PostgreSQL 11.22 手册

PostgreSQL 全球开发组

翻译：冯若航[1]，Pigsty 项目组 [2]

[1] https://vonng.com
[2] https://pigsty.cc

PostgreSQL 11.22 手册

PostgreSQL 全球开发组
翻译：冯若航[1]，Pigsty 项目组[2]
版权 © 1996-2023 PostgreSQL 全球开发组

法律声明

PostgreSQL 版权所有 © 1996-2023 PostgreSQL 全球开发组。

Postgres 95 版权所有 © 1994-5 加利福尼亚大学董事会。

特此授权，可为任何目的使用、复制、修改和分发本软件及其文档，无需付费，也无需书面协议，但条件是所有副本中都必须包含上述版权声明、本段以及后续两段内容。

在任何情况下，加利福尼亚大学都不对任何一方承担因使用本软件及其文档而产生的任何直接、间接、特殊、附带或后果性损害赔偿责任，包括利润损失，即使加利福尼亚大学已被告知此类损害发生的可能性。

加利福尼亚大学明确不作任何担保，包括但不限于适销性和特定用途适用性的默示担保。本协议项下提供的软件按“原样”提供，加利福尼亚大学没有义务提供维护、支持、更新、增强或修改。

[1] https://vonng.com
[2] https://pigsty.cc

目录

前言 xxxiii
1. 什么是 PostgreSQL? xxxiii
2. PostgreSQL 简史 xxxiv
2.1. 伯克利 POSTGRES 项目 xxxiv
2.2. Postgres 95 xxxiv
2.3. PostgreSQL xxxv
3. 约定 xxxv
4. 更多信息 xxxv
5. 缺陷报告指南 xxxvi
5.1. 识别缺陷 xxxvi
5.2. 报告哪些内容 xxxvi
5.3. 到哪里报告缺陷 xxxviii
I. 教程 1
1. 从头开始 3
1.1. 安装 3
1.2. 架构基础 3
1.3. 创建数据库 3
1.4. 访问数据库 5
2. SQL 语言 7
2.1. 简介 7
2.2. 概念 7
2.3. 创建一个新表 8
2.4. 向表中填充行 8
2.5. 查询表 9
2.6. 表之间的连接 11
2.7. 聚合函数 13
2.8. 更新 16
2.9. 删除 17
3. 高级特性 18
3.1. 简介 18
3.2. 视图 18
3.3. 外键 18
3.4. 事务 19
3.5. 窗口函数 21
3.6. 继承 24
3.7. 小结 26
II. SQL 语言 27
4. SQL 语法 35
4.1. 词法结构 35
4.2. 值表达式 44
4.3. 调用函数 59
5. 数据定义 62
5.1. 表基础 62
5.2. 默认值 63
5.3. 约束 64
5.4. 系统列 73
5.5. 修改表 74
5.6. 权限 77
5.7. 行安全性策略 78
5.8. 模式 84
5.9. 继承 89

5.10. 表分区 92
5.11. 外部数据 105
5.12. 其他数据库对象 105
5.13. 依赖跟踪 106
6. 数据操纵 108
6.1. 插入数据 108
6.2. 更新数据 109
6.3. 删除数据 110
6.4. 从被修改的行中返回数据 110
7. 查询 112
7.1. 概述 112
7.2. 表表达式 112
7.3. 选择列表 128
7.4. 组合查询 130
7.5. 行排序 131
7.6. LIMIT 和 OFFSET 132
7.7. VALUES 列表 133
7.8. WITH 查询（公共表表达式） 134
8. 数据类型 141
8.1. 数字类型 142
8.2. 货币类型 147
8.3. 字符类型 147
8.4. 二进制数据类型 149
8.5. 日期/时间类型 151
8.6. 布尔类型 161
8.7. 枚举类型 162
8.8. 几何类型 164
8.9. 网络地址类型 167
8.10. 位串类型 169
8.11. 文本检索类型 170
8.12. UUID 类型 173
8.13. XML 类型 174
8.14. JSON 类型 176
8.15. 数组 183
8.16. 复合类型 194
8.17. 范围类型 201
8.18. 域类型 206
8.19. 对象标识符类型 207
8.20. pg_lsn 类型 208
8.21. 伪类型 209
9. 函数和操作符 211
9.1. 逻辑操作符 211
9.2. 比较函数和操作符 211
9.3. 数学函数和操作符 214
9.4. 字符串函数和操作符 220
9.5. 二进制串函数和操作符 235
9.6. 位串函数和操作符 239
9.7. 模式匹配 240
9.8. 数据类型格式化函数 255
9.9. 日期/时间函数和操作符 262
9.10. 枚举支持函数 277
9.11. 几何函数和操作符 278
9.12. 网络地址函数和操作符 281

9.13. 文本检索函数和操作符 285
9.14. XML 函数 293
9.15. JSON 函数和操作符 307
9.16. 序列操作函数 317
9.17. 条件表达式 319
9.18. 数组函数和操作符 322
9.19. 范围函数和操作符 326
9.20. 聚合函数 329
9.21. 窗口函数 338
9.22. 子查询表达式 340
9.23. 行和数组比较 342
9.24. 集合返回函数 345
9.25. 系统信息函数 348
9.26. 系统管理函数 363
9.27. 触发器函数 378
9.28. 事件触发器函数 378
10. 类型转换 381
10.1. 概述 381
10.2. 操作符 382
10.3. 函数 386
10.4. 值存储 390
10.5. UNION、CASE 及相关结构 391
10.6. SELECT 输出列 393
11. 索引 394
11.1. 简介 394
11.2. 索引类型 395
11.3. 多列索引 397
11.4. 索引和 ORDER BY 397
11.5. 组合多个索引 398
11.6. 唯一索引 399
11.7. 表达式索引 399
11.8. 部分索引 400
11.9. 仅索引扫描和覆盖索引 403
11.10. 操作符类和操作符族 406
11.11. 索引和排序规则 407
11.12. 检查索引使用情况 408
12. 全文检索 409
12.1. 简介 409
12.2. 表和索引 413
12.3. 控制文本检索 415
12.4. 附加特性 423
12.5. 解析器 428
12.6. 词典 430
12.7. 配置示例 440
12.8. 测试和调试文本检索 442
12.9. 文本检索的首选索引类型 446
12.10. psql 支持 447
12.11. 限制 451
13. 并发控制 452
13.1. 简介 452
13.2. 事务隔离 452
13.3. 显式锁定 457
13.4. 应用级别的数据一致性检查 463

13.5. 注意事项 464
13.6. 锁定和索引 465
14. 性能提示 466
14.1. 使用 EXPLAIN 466
14.2. 规划器使用的统计信息 477
14.3. 用显式 JOIN 子句控制规划器 480
14.4. 填充一个数据库 482
14.5. 非持久设置 485
15. 并行查询 486
15.1. 并行查询如何工作 486
15.2. 何时可以使用并行查询？ 486
15.3. 并行计划 487
15.4. 并行安全性 489
III. 服务器管理 491
16. 从源代码安装 498
16.1. 简要说明 498
16.2. 需求 498
16.3. 获取源代码 500
16.4. 安装过程 500
16.5. 安装后设置 512
16.6. 受支持的平台 513
16.7. 平台相关说明 514
17. 在 Windows 上从源代码安装 522
17.1. 使用 Visual C++ 或 Microsoft Windows SDK 构建 522
18. 服务器设置和操作 528
18.1. PostgreSQL 用户账户 528
18.2. 创建一个数据库集簇 528
18.3. 启动数据库服务器 530
18.4. 管理内核资源 533
18.5. 关闭服务器 542
18.6. 升级 PostgreSQL 集簇 543
18.7. 防止服务器欺骗 546
18.8. 加密选项 546
18.9. 使用 SSL 的安全 TCP/IP 连接 547
18.10. 通过 SSH 隧道建立安全 TCP/IP 连接 551
18.11. 在 Windows 上注册事件日志 552
19. 服务器配置 554
19.1. 设置参数 554
19.2. 文件位置 558
19.3. 连接和认证 558
19.4. 资源消耗 564
19.5. 预写式日志 570
19.6. 复制 575
19.7. 查询规划 580
19.8. 错误报告和日志 586
19.9. 运行时统计数据 597
19.10. 自动清理 598
19.11. 客户端连接默认值 600
19.12. 锁管理 609
19.13. 版本和平台兼容性 610
19.14. 错误处理 612
19.15. 预置选项 612
19.16. 自定义选项 614

19.17. 开发人员选项 614
19.18. 短选项 617
20. 客户端认证 619
20.1. pg_hba.conf 文件 619
20.2. 用户名映射 625
20.3. 认证方法 626
20.4. 信任认证 627
20.5. 密码认证 627
20.6. GSSAPI 认证 628
20.7. SSPI 认证 629
20.8. Ident 认证 630
20.9. Peer 认证 631
20.10. LDAP 认证 631
20.11. RADIUS 认证 634
20.12. 证书认证 635
20.13. PAM 认证 635
20.14. BSD 认证 636
20.15. 认证问题 636
21. 数据库角色 638
21.1. 数据库角色 638
21.2. 角色属性 639
21.3. 角色成员资格 640
21.4. 删除角色 642
21.5. 默认角色 643
21.6. 函数安全性 644
22. 管理数据库 645
22.1. 概述 645
22.2. 创建数据库 645
22.3. 模板数据库 646
22.4. 数据库配置 648
22.5. 删除数据库 648
22.6. 表空间 648
23. 本地化 651
23.1. 区域设置支持 651
23.2. 排序规则支持 653
23.3. 字符集支持 659
24. 日常数据库维护任务 665
24.1. 日常清理 665
24.2. 日常重建索引 672
24.3. 日志文件维护 673
25. 备份和恢复 675
25.1. SQL 转储 675
25.2. 文件系统级备份 678
25.3. 持续归档和时间点恢复（PITR） 679
26. 高可用、负载均衡和复制 691
26.1. 不同方案的比较 691
26.2. 日志传送备库 694
26.3. 故障切换 701
26.4. 日志传送的另一种方法 702
26.5. 热备 703
27. 恢复配置 710
27.1. 归档恢复设置 710
27.2. 恢复目标设置 711

27.3. 备库设置 712
28. 监控数据库活动 714
28.1. 标准 Unix 工具 714
28.2. 统计收集器 715
28.3. 查看锁 740
28.4. 进度报告 740
28.5. 动态追踪 742
29. 监控磁盘使用情况 751
29.1. 确定磁盘使用情况 751
29.2. 磁盘已满故障 752
30. 可靠性与预写式日志 753
30.1. 可靠性 753
30.2. 预写式日志（WAL） 754
30.3. 异步提交 755
30.4. WAL 配置 756
30.5. WAL 内部机制 758
31. 逻辑复制 760
31.1. 发布 760
31.2. 订阅 761
31.3. 冲突 762
31.4. 限制 762
31.5. 架构 762
31.6. 监控 763
31.7. 安全性 763
31.8. 配置设置 764
31.9. 快速设置 764
32. 即时编译（JIT） 765
32.1. 什么是 JIT 编译？ 765
32.2. 何时使用 JIT？ 765
32.3. 配置 766
32.4. 可扩展性 767
33. 回归测试 768
33.1. 运行测试 768
33.2. 测试结果评估 771
33.3. 变体比较文件 773
33.4. TAP 测试 774
33.5. 测试覆盖率检查 775
IV. 客户端接口 776
34. libpq - C 库 781
34.1. 数据库连接控制函数 781
34.2. 连接状态函数 794
34.3. 命令执行函数 801
34.4. 异步命令处理 817
34.5. 逐行检索查询结果 821
34.6. 取消进行中的查询 822
34.7. 快速路径接口 823
34.8. 异步通知 824
34.9. COPY 命令相关的函数 825
34.10. 控制函数 829
34.11. 杂项函数 831
34.12. 通知处理 834
34.13. 事件系统 835
34.14. 环境变量 841

34.15. 密码文件 843
34.16. 连接服务文件 843
34.17. 连接参数的 LDAP 查找 844
34.18. SSL 支持 845
34.19. 多线程程序中的行为 848
34.20. 构建 libpq 程序 849
34.21. 示例程序 850
35. 大对象 860
35.1. 简介 860
35.2. 实现特性 860
35.3. 客户端接口 860
35.4. 服务器端函数 864
35.5. 示例程序 866
36. ECPG - C 中的嵌入式 SQL 871
36.1. 概念 871
36.2. 管理数据库连接 871
36.3. 运行 SQL 命令 875
36.4. 使用主变量 877
36.5. 动态 SQL 893
36.6. pgtypes 库 895
36.7. 使用描述符区域 909
36.8. 错误处理 924
36.9. 预处理器指令 932
36.10. 处理嵌入式 SQL 程序 934
36.11. 库函数 935
36.12. 大对象 936
36.13. C++ 应用 937
36.14. 嵌入式 SQL 命令 941
36.15. Informix 兼容模式 963
36.16. Oracle 兼容模式 980
36.17. 内部 981
37. 信息模式 983
37.1. 模式 983
37.2. 数据类型 983
37.3. information_schema_catalog_name 984
37.4. administrable_role_authorizations 984
37.5. applicable_roles 984
37.6. attributes 985
37.7. character_sets 987
37.8. check_constraint_routine_usage 988
37.9. check_constraints 988
37.10. collations 988
37.11. collation_character_set_applicability 989
37.12. column_domain_usage 989
37.13. column_options 990
37.14. column_privileges 990
37.15. column_udt_usage 990
37.16. columns 991
37.17. constraint_column_usage 994
37.18. constraint_table_usage 994
37.19. data_type_privileges 995
37.20. domain_constraints 995
37.21. domain_udt_usage 996

37.22. domains 996
37.23. element_types 998
37.24. enabled_roles 1000
37.25. foreign_data_wrapper_options 1000
37.26. foreign_data_wrappers 1001
37.27. foreign_server_options 1001
37.28. foreign_servers 1001
37.29. foreign_table_options 1002
37.30. foreign_tables 1002
37.31. key_column_usage 1002
37.32. parameters 1003
37.33. referential_constraints 1004
37.34. role_column_grants 1005
37.35. role_routine_grants 1005
37.36. role_table_grants 1006
37.37. role_udt_grants 1007
37.38. role_usage_grants 1007
37.39. routine_privileges 1007
37.40. routines 1008
37.41. schemata 1012
37.42. sequences 1012
37.43. sql_features 1013
37.44. sql_implementation_info 1013
37.45. sql_languages 1014
37.46. sql_packages 1014
37.47. sql_parts 1015
37.48. sql_sizing 1015
37.49. sql_sizing_profiles 1015
37.50. table_constraints 1016
37.51. table_privileges 1016
37.52. tables 1017
37.53. transforms 1018
37.54. triggered_update_columns 1018
37.55. triggers 1019
37.56. udt_privileges 1020
37.57. usage_privileges 1020
37.58. user_defined_types 1021
37.59. user_mapping_options 1022
37.60. user_mappings 1022
37.61. view_column_usage 1023
37.62. view_routine_usage 1023
37.63. view_table_usage 1024
37.64. views 1024
V. 服务器编程 1026
38. 扩展 SQL 1032
38.1. 可扩展性如何运作 1032
38.2. PostgreSQL 类型系统 1032
38.3. 用户定义的函数 1034
38.4. 用户定义的过程 1034
38.5. 查询语言（SQL）函数 1034
38.6. 函数重载 1053
38.7. 函数易变性分类 1053

38.8. 过程语言函数 1055
38.9. 内部函数 1055
38.10. C 语言函数 1055
38.11. 用户定义的聚合 1077
38.12. 用户定义的类型 1084
38.13. 用户定义的操作符 1089
38.14. 操作符优化信息 1089
38.15. 索引扩展接口 1093
38.16. 将相关对象打包成扩展 1106
38.17. 扩展构建基础设施 1113
39. 触发器 1118
39.1. 触发器行为概述 1118
39.2. 数据更改的可见性 1120
39.3. 用 C 编写触发器函数 1121
39.4. 一个完整的触发器示例 1124
40. 事件触发器 1128
40.1. 事件触发器行为概述 1128
40.2. 事件触发器触发矩阵 1129
40.3. 用 C 编写事件触发器函数 1133
40.4. 完整的事件触发器示例 1134
40.5. 表重写事件触发器示例 1136
41. 规则系统 1137
41.1. 查询树 1137
41.2. 视图和规则系统 1139
41.3. 物化视图 1147
41.4. INSERT、UPDATE 和 DELETE 规则 1150
41.5. 规则和权限 1162
41.6. 规则和命令状态 1164
41.7. 规则与触发器 1165
42. 过程语言 1169
42.1. 安装过程语言 1169
43. PL/pgSQL − SQL 过程语言 1172
43.1. 概述 1172
43.2. PL/pgSQL 的结构 1173
43.3. 声明 1174
43.4. 表达式 1181
43.5. 基本语句 1181
43.6. 控制结构 1190
43.7. 游标 1206
43.8. 事务管理 1212
43.9. 错误和消息 1214
43.10. 触发器函数 1216
43.11. PL/pgSQL 内部机制 1225
43.12. PL/pgSQL 开发提示 1229
43.13. 从 Oracle PL/SQL 移植 1232
44. PL/Tcl - Tcl 过程语言 1242
44.1. 概述 1242
44.2. PL/Tcl 函数和参数 1242
44.3. PL/Tcl 中的数据值 1244
44.4. PL/Tcl 中的全局数据 1245
44.5. 从 PL/Tcl 访问数据库 1245
44.6. PL/Tcl 中的触发器函数 1247
44.7. PL/Tcl 中的事件触发器函数 1249

44.8. PL/Tcl 中的错误处理 1250
44.9. PL/Tcl 中的显式子事务 1251
44.10. 事务管理 1251
44.11. PL/Tcl 配置 1252
44.12. Tcl 过程名 1252
45. PL/Perl - Perl 过程语言 1253
45.1. PL/Perl 函数和参数 1253
45.2. PL/Perl 中的数据值 1258
45.3. 内置函数 1258
45.4. PL/Perl 中的全局值 1264
45.5. 受信任与不受信任的 PL/Perl 1265
45.6. PL/Perl 触发器 1266
45.7. PL/Perl 事件触发器 1268
45.8. PL/Perl 内部机制 1268
46. PL/Python - Python 过程语言 1271
46.1. Python 2 与 Python 3 1271
46.2. PL/Python 函数 1272
46.3. 数据值 1273
46.4. 共享数据 1280
46.5. 匿名代码块 1280
46.6. 触发器函数 1280
46.7. 数据库访问 1281
46.8. 显式子事务 1285
46.9. 事务管理 1287
46.10. 辅助函数 1287
46.11. 环境变量 1288
47. 服务器编程接口 1290
47.1. 接口函数 1290
47.2. 接口支持函数 1325
47.3. 内存管理 1334
47.4. 事务管理 1344
47.5. 数据更改的可见性 1347
47.6. 示例 1347
48. 后台工作进程 1351
49. 逻辑解码 1354
49.1. 逻辑解码示例 1354
49.2. 逻辑解码概念 1356
49.3. 流复制协议接口 1357
49.4. 逻辑解码 SQL 接口 1357
49.5. 与逻辑解码相关的系统目录 1357
49.6. 逻辑解码输出插件 1357
49.7. 逻辑解码输出写入器 1361
49.8. 逻辑解码的同步复制支持 1361
50. 复制进度跟踪 1363
VI. 参考 1364
I. SQL 命令 1370
ABORT 1374
ALTER AGGREGATE 1375
ALTER COLLATION 1377
ALTER CONVERSION 1379
ALTER DATABASE 1381
ALTER DEFAULT PRIVILEGES 1384
ALTER DOMAIN 1387
ALTER EVENT TRIGGER 1391

ALTER EXTENSION 1392
ALTER FOREIGN DATA WRAPPER 1395
ALTER FOREIGN TABLE 1397
ALTER FUNCTION 1402
ALTER GROUP 1406
ALTER INDEX 1408
ALTER LANGUAGE 1411
ALTER LARGE OBJECT 1412
ALTER MATERIALIZED VIEW 1413
ALTER OPERATOR 1415
ALTER OPERATOR CLASS 1417
ALTER OPERATOR FAMILY 1419
ALTER POLICY 1423
ALTER PROCEDURE 1425
ALTER PUBLICATION 1428
ALTER ROLE 1430
ALTER ROUTINE 1434
ALTER RULE 1436
ALTER SCHEMA 1437
ALTER SEQUENCE 1438
ALTER SERVER 1441
ALTER STATISTICS 1443
ALTER SUBSCRIPTION 1444
ALTER SYSTEM 1446
ALTER TABLE 1448
ALTER TABLESPACE 1463
ALTER TEXT SEARCH CONFIGURATION 1465
ALTER TEXT SEARCH DICTIONARY 1467
ALTER TEXT SEARCH PARSER 1469
ALTER TEXT SEARCH TEMPLATE 1470
ALTER TRIGGER 1471
ALTER TYPE 1473
ALTER USER 1477
ALTER USER MAPPING 1478
ALTER VIEW 1480
ANALYZE 1482
BEGIN 1485
CALL 1487
CHECKPOINT 1488
CLOSE 1489
CLUSTER 1491
COMMENT 1493
COMMIT 1497
COMMIT PREPARED 1498
COPY 1499
CREATE ACCESS METHOD 1509
CREATE AGGREGATE 1510
CREATE CAST 1517
CREATE COLLATION 1522
CREATE CONVERSION 1524
CREATE DATABASE 1526
CREATE DOMAIN 1529
CREATE EVENT TRIGGER 1532
CREATE EXTENSION 1534

CREATE FOREIGN DATA WRAPPER 1537
CREATE FOREIGN TABLE 1539
CREATE FUNCTION 1543
CREATE GROUP 1552
CREATE INDEX 1553
CREATE LANGUAGE 1561
CREATE MATERIALIZED VIEW 1564
CREATE OPERATOR 1566
CREATE OPERATOR CLASS 1569
CREATE OPERATOR FAMILY 1572
CREATE POLICY 1573
CREATE PROCEDURE 1579
CREATE PUBLICATION 1582
CREATE ROLE 1584
CREATE RULE 1589
CREATE SCHEMA 1592
CREATE SEQUENCE 1595
CREATE SERVER 1599
CREATE STATISTICS 1601
CREATE SUBSCRIPTION 1603
CREATE TABLE 1606
CREATE TABLE AS 1626
CREATE TABLESPACE 1629
CREATE TEXT SEARCH CONFIGURATION 1631
CREATE TEXT SEARCH DICTIONARY 1633
CREATE TEXT SEARCH PARSER 1635
CREATE TEXT SEARCH TEMPLATE 1637
CREATE TRANSFORM 1639
CREATE TRIGGER 1641
CREATE TYPE 1648
CREATE USER 1657
CREATE USER MAPPING 1658
CREATE VIEW 1660
DEALLOCATE 1665
DECLARE 1666
DELETE 1670
DISCARD 1673
DO 1675
DROP ACCESS METHOD 1677
DROP AGGREGATE 1678
DROP CAST 1680
DROP COLLATION 1681
DROP CONVERSION 1682
DROP DATABASE 1683
DROP DOMAIN 1684
DROP EVENT TRIGGER 1685
DROP EXTENSION 1686
DROP FOREIGN DATA WRAPPER 1687
DROP FOREIGN TABLE 1688
DROP FUNCTION 1689
DROP GROUP 1691
DROP INDEX 1692
DROP LANGUAGE 1694
DROP MATERIALIZED VIEW 1696

DROP OPERATOR 1697
DROP OPERATOR CLASS 1699
DROP OPERATOR FAMILY 1701
DROP OWNED 1703
DROP POLICY 1704
DROP PROCEDURE 1705
DROP PUBLICATION 1707
DROP ROLE 1708
DROP ROUTINE 1710
DROP RULE 1711
DROP SCHEMA 1712
DROP SEQUENCE 1714
DROP SERVER 1715
DROP STATISTICS 1716
DROP SUBSCRIPTION 1717
DROP TABLE 1719
DROP TABLESPACE 1721
DROP TEXT SEARCH CONFIGURATION 1722
DROP TEXT SEARCH DICTIONARY 1723
DROP TEXT SEARCH PARSER 1724
DROP TEXT SEARCH TEMPLATE 1725
DROP TRANSFORM 1726
DROP TRIGGER 1727
DROP TYPE 1729
DROP USER 1730
DROP USER MAPPING 1731
DROP VIEW 1732
END 1733
EXECUTE 1734
EXPLAIN 1735
FETCH 1740
GRANT 1744
IMPORT FOREIGN SCHEMA 1751
INSERT 1753
LISTEN 1760
LOAD 1762
LOCK 1763
MOVE 1766
NOTIFY 1768
PREPARE 1770
PREPARE TRANSACTION 1773
REASSIGN OWNED 1775
REFRESH MATERIALIZED VIEW 1776
REINDEX 1778
RELEASE SAVEPOINT 1781
RESET 1783
REVOKE 1784
ROLLBACK 1788
ROLLBACK PREPARED 1789
ROLLBACK TO SAVEPOINT 1790
SAVEPOINT 1792
SECURITY LABEL 1794
SELECT 1797
SELECT INTO 1818

SET ... 1820
SET CONSTRAINTS ... 1823
SET ROLE ... 1825
SET SESSION AUTHORIZATION ... 1827
SET TRANSACTION ... 1829
SHOW ... 1832
START TRANSACTION ... 1834
TRUNCATE ... 1835
UNLISTEN ... 1837
UPDATE ... 1839
VACUUM ... 1844
VALUES ... 1847
II. PostgreSQL 客户端应用 ... 1850
clusterdb ... 1851
createdb ... 1854
createuser ... 1857
dropdb ... 1862
dropuser ... 1865
ecpg ... 1868
pg_basebackup ... 1871
pgbench ... 1878
pg_config ... 1896
pg_dump ... 1899
pg_dumpall ... 1911
pg_isready ... 1918
pg_receivewal ... 1920
pg_recvlogical ... 1924
pg_restore ... 1928
psql ... 1937
reindexdb ... 1977
vacuumdb ... 1980
III. PostgreSQL 服务器端应用 ... 1984
initdb ... 1985
pg_archivecleanup ... 1989
pg_controldata ... 1991
pg_ctl ... 1992
pg_resetwal ... 1998
pg_rewind ... 2002
pg_test_fsync ... 2005
pg_test_timing ... 2006
pg_upgrade ... 2010
pg_verify_checksums ... 2018
pg_waldump ... 2019
postgres ... 2022
postmaster ... 2029
VII. 内部 ... 2030
51. PostgreSQL 内部概述 ... 2036
51.1. 查询的路径 ... 2036
51.2. 连接是如何建立的 ... 2036
51.3. 解析器阶段 ... 2037
51.4. PostgreSQL 规则系统 ... 2038
51.5. 规划器/优化器 ... 2038
51.6. 执行器 ... 2039
52. 系统目录 ... 2041

52.1. 概述 2041
52.2. pg_aggregate 2043
52.3. pg_am 2044
52.4. pg_amop 2045
52.5. pg_amproc 2046
52.6. pg_attrdef 2046
52.7. pg_attribute 2047
52.8. pg_authid 2049
52.9. pg_auth_members 2050
52.10. pg_cast 2050
52.11. pg_class 2051
52.12. pg_collation 2054
52.13. pg_constraint 2055
52.14. pg_conversion 2057
52.15. pg_database 2058
52.16. pg_db_role_setting 2059
52.17. pg_default_acl 2059
52.18. pg_depend 2060
52.19. pg_description 2061
52.20. pg_enum 2062
52.21. pg_event_trigger 2062
52.22. pg_extension 2063
52.23. pg_foreign_data_wrapper 2063
52.24. pg_foreign_server 2064
52.25. pg_foreign_table 2064
52.26. pg_index 2065
52.27. pg_inherits 2067
52.28. pg_init_privs 2067
52.29. pg_language 2068
52.30. pg_largeobject 2069
52.31. pg_largeobject_metadata 2069
52.32. pg_namespace 2069
52.33. pg_opclass 2070
52.34. pg_operator 2070
52.35. pg_opfamily 2071
52.36. pg_partitioned_table 2071
52.37. pg_pltemplate 2072
52.38. pg_policy 2073
52.39. pg_proc 2074
52.40. pg_publication 2077
52.41. pg_publication_rel 2077
52.42. pg_range 2077
52.43. pg_replication_origin 2078
52.44. pg_rewrite 2078
52.45. pg_seclabel 2079
52.46. pg_sequence 2079
52.47. pg_shdepend 2080
52.48. pg_shdescription 2081
52.49. pg_shseclabel 2081
52.50. pg_statistic 2082
52.51. pg_statistic_ext 2083
52.52. pg_subscription 2084

52.53. pg_subscription_rel 2084
52.54. pg_tablespace 2085
52.55. pg_transform 2085
52.56. pg_trigger 2086
52.57. pg_ts_config 2087
52.58. pg_ts_config_map 2088
52.59. pg_ts_dict 2088
52.60. pg_ts_parser 2089
52.61. pg_ts_template 2089
52.62. pg_type 2090
52.63. pg_user_mapping 2095
52.64. 系统视图 2095
52.65. pg_available_extensions 2096
52.66. pg_available_extension_versions 2097
52.67. pg_config 2097
52.68. pg_cursors 2097
52.69. pg_file_settings 2098
52.70. pg_group 2099
52.71. pg_hba_file_rules 2099
52.72. pg_indexes 2100
52.73. pg_locks 2100
52.74. pg_matviews 2103
52.75. pg_policies 2103
52.76. pg_prepared_statements 2104
52.77. pg_prepared_xacts 2104
52.78. pg_publication_tables 2105
52.79. pg_replication_origin_status 2105
52.80. pg_replication_slots 2105
52.81. pg_roles 2106
52.82. pg_rules 2107
52.83. pg_seclabels 2107
52.84. pg_sequences 2108
52.85. pg_settings 2109
52.86. pg_shadow 2110
52.87. pg_stats 2111
52.88. pg_tables 2113
52.89. pg_timezone_abbrevs 2113
52.90. pg_timezone_names 2114
52.91. pg_user 2114
52.92. pg_user_mappings 2114
52.93. pg_views 2115
53. 前端/后端协议 2116
53.1. 概述 2116
53.2. 消息流 2117
53.3. SASL 认证 2129
53.4. 流复制协议 2130
53.5. 逻辑流复制协议 2137
53.6. 消息数据类型 2138
53.7. 消息格式 2139
53.8. 错误和通知消息字段 2157
53.9. 逻辑复制消息格式 2158
53.10. 自协议 2.0 以来的变更总结 2163

54. PostgreSQL 编码约定 2165
54.1. 格式 2165
54.2. 在服务器内部报告错误 2165
54.3. 错误消息风格指南 2168
54.4. 其他编码约定 2172
55. 本地语言支持 2175
55.1. 给翻译者 2175
55.2. 给程序员 2177
56. 编写过程语言调用处理器 2181
57. 编写外部数据包装器 2184
57.1. 外部数据包装器函数 2184
57.2. 外部数据包装器回调例程 2184
57.3. 外部数据包装器辅助函数 2198
57.4. 外部数据包装器查询规划 2199
57.5. 外部数据包装器中的行锁定 2201
58. 编写一种表采样方法 2203
58.1. 采样方法支持函数 2203
59. 编写自定义扫描提供者 2206
59.1. 创建自定义扫描路径 2206
59.2. 创建自定义扫描计划 2208
59.3. 执行自定义扫描 2209
60. 遗传查询优化器 2211
60.1. 将查询处理看成是一个复杂的优化问题 2211
60.2. 遗传算法 2211
60.3. PostgreSQL 中的遗传查询优化（GEQO） 2212
60.4. 进一步阅读 2213
61. 索引访问方法接口定义 2214
61.1. 索引的基本 API 结构 2214
61.2. 索引访问方法函数 2217
61.3. 索引扫描 2222
61.4. 索引锁定注意事项 2223
61.5. 索引唯一性检查 2224
61.6. 索引代价估算函数 2225
62. 通用 WAL 记录 2229
63. B-树索引 2231
63.1. 简介 2231
63.2. B-树操作符类的行为 2231
63.3. B-树支持函数 2232
63.4. 实现 2233
64. GiST 索引 2234
64.1. 简介 2234
64.2. 内置操作符类 2234
64.3. 可扩展性 2235
64.4. 实现 2245
64.5. 示例 2245
65. SP-GiST 索引 2246
65.1. 简介 2246
65.2. 内置操作符类 2246
65.3. 可扩展性 2246
65.4. 实现 2255
65.5. 示例 2256
66. GIN 索引 2257
66.1. 简介 2257
66.2. 内置操作符类 2257

66.3. 可扩展性 2257
66.4. 实现 2260
66.5. GIN 提示和技巧 2261
66.6. 限制 2262
66.7. 示例 2262
67. BRIN 索引 2263
67.1. 简介 2263
67.2. 内置操作符类 2263
67.3. 可扩展性 2264
68. Hash 索引 2268
68.1. 概述 2268
68.2. 实现 2268
69. 数据库物理存储 2270
69.1. 数据库文件布局 2270
69.2. TOAST 2272
69.3. 空闲空间映射 2274
69.4. 可见性映射 2274
69.5. 初始化分支 2275
69.6. 数据库页面布局 2275
69.7. 堆内元组（HOT） 2277
70. 系统目录声明与初始内容 2279
70.1. 系统目录声明规则 2279
70.2. 系统目录初始数据 2280
70.3. BKI 文件格式 2283
70.4. BKI 命令 2284
70.5. 引导 BKI 文件的结构 2285
70.6. BKI 示例 2286
71. 规划器如何使用统计信息 2287
71.1. 行估计示例 2287
71.2. 多元统计信息示例 2293
71.3. 规划器统计信息与安全 2294
VIII. 附录 2296
A. PostgreSQL 错误代码 2304
B. 日期/时间支持 2312
B.1. 日期/时间输入解释 2312
B.2. 处理无效或有歧义的时间戳 2313
B.3. 日期/时间关键字 2314
B.4. 日期/时间配置文件 2315
B.5. POSIX 时区规范 2316
B.6. 历法的历史 2318
B.7. 儒略日 2319
C. SQL 关键字 2320
D. SQL 符合性 2342
D.1. 支持的特性 2343
D.2. 不支持的特性 2355
D.3. XML 的限制以及对 SQL/XML 的符合性 2365
E. 发布说明 2368
E.1. 发布版本 11.22 2368
E.2. 发布版本 11.21 2371
E.3. 发布版本 11.20 2374
E.4. 发布版本 11.19 2378
E.5. 发布版本 11.18 2381
E.6. 发布版本 11.17 2384
E.7. 发布版本 11.16 2388

E.8. 发布版本 11.15 2391
E.9. 发布版本 11.14 2393
E.10. 发布版本 11.13 2399
E.11. 发布版本 11.12 2404
E.12. 发布版本 11.11 2407
E.13. 发布版本 11.10 2412
E.14. 发布版本 11.9 2415
E.15. 发布版本 11.8 2419
E.16. 发布版本 11.7 2423
E.17. 发布版本 11.6 2428
E.18. 发布版本 11.5 2433
E.19. 发布版本 11.4 2436
E.20. 发布版本 11.3 2439
E.21. 发布版本 11.2 2444
E.22. 发布版本 11.1 2449
E.23. 发布版本 11 2451
E.24. 先前版本 2470
F. 额外提供的模块 2471
F.1. adminpack 2473
F.2. amcheck 2474
F.3. auth_delay 2477
F.4. auto_explain 2478
F.5. bloom 2481
F.6. btree_gin 2485
F.7. btree_gist 2486
F.8. citext 2488
F.9. cube 2491
F.10. dblink 2496
F.11. dict_int 2527
F.12. dict_xsyn 2528
F.13. earthdistance 2530
F.14. file_fdw 2532
F.15. fuzzystrmatch 2535
F.16. hstore 2538
F.17. intagg 2546
F.18. intarray 2548
F.19. isn 2551
F.20. lo 2555
F.21. ltree 2557
F.22. pageinspect 2564
F.23. passwordcheck 2572
F.24. pg_buffercache 2573
F.25. pgcrypto 2575
F.26. pg_freespacemap 2587
F.27. pg_prewarm 2589
F.28. pgrowlocks 2591
F.29. pg_stat_statements 2593
F.30. pgstattuple 2598
F.31. pg_trgm 2603
F.32. pg_visibility 2609
F.33. postgres_fw 2611
F.34. seg 2617
F.35. sepgsql 2620
F.36. spi 2628

F.37. sslinfo 2630
F.38. tablefunc 2632
F.39. tcn 2642
F.40. test_decoding 2643
F.41. tsm_system_rows 2644
F.42. tsm_system_time 2645
F.43. unaccent 2646
F.44. uuid-ossp 2649
F.45. xml2 2651
G. 额外提供的程序 2655
G.1. 客户端应用 2655
G.2. 服务器应用 2662
H. 外部项目 2667
H.1. 客户端接口 2667
H.2. 管理工具 2667
H.3. 过程语言 2667
H.4. 扩展 2667
I. 源代码仓库 2668
I.1. 通过 Git 获取源代码 2668
J. 文档 2669
J.1. DocBook 2669
J.2. 工具集 2669
J.3. 构建文档 2671
J.4. 文档编写 2673
J.5. 风格指南 2673
K. 缩略词 2675
L. 过时或重命名的功能 2682
L.1. `pg_xlogdump` 重命名为 `pg_waldump` 2682
L.2. `pg_resetxlog` 重命名为 `pg_resetwal` 2682
L.3. `pg_receivexlog` 重命名为 `pg_receivewal` 2682
参考书目 2683
索引 2685

插图清单

60.1. 遗传算法的结构图 2212

表格清单

4.1. 反斜线转义序列 37
4.2. 操作符优先级（从高到低） 43
8.1. 数据类型 141
8.2. 数字类型 142
8.3. 货币类型 147
8.4. 字符类型 147
8.5. 特殊字符类型 149
8.6. 二进制数据类型 149
8.7. `bytea` 字面量中需要转义的字节 150
8.8. `bytea` 输出转义字节 151
8.9. 日期/时间类型 151
8.10. 日期输入 153
8.11. 时间输入 154
8.12. 时区输入 154
8.13. 特殊日期/时间输入 156
8.14. 日期/时间输出风格 156
8.15. 日期顺序习惯 157
8.16. ISO 8601 间隔单位缩写 159
8.17. 间隔输入 160
8.18. 间隔输出风格示例 161
8.19. 布尔数据类型 161
8.20. 几何类型 164
8.21. 网络地址类型 167
8.22. `cidr` 类型输入示例 167
8.23. JSON 基本类型及其对应的 PostgreSQL 类型 177
8.24. 对象标识符类型 207
8.25. 伪类型 209
9.1. 比较操作符 211
9.2. 比较谓词 212
9.3. 比较函数 214
9.4. 数学操作符 214
9.5. 数学函数 215
9.6. 随机函数 219
9.7. 三角函数 219
9.8. SQL 字符串函数和操作符 220
9.9. 其他字符串函数 222
9.10. 内置转换 230
9.11. SQL 二进制串函数和操作符 236
9.12. 其他二进制串函数 237
9.13. 位串操作符 239
9.14. 正则表达式匹配操作符 242
9.15. 正则表达式原子 247
9.16. 正则表达式量词 248
9.17. 正则表达式约束 248
9.18. 正则表达式字符输入转义 250
9.19. 正则表达式字符类简写转义 250
9.20. 正则表达式约束转义 251
9.21. 正则表达式反向引用 251
9.22. ARE 嵌入选项字母 252
9.23. 格式化函数 255
9.24. 用于日期/时间格式化的模板模式 256

9.25. 用于日期/时间格式化的模板模式修饰符 258
9.26. 用于数值格式化的模板模式 259
9.27. 用于数值格式化的模板模式修饰符 260
9.28. to_char 示例 261
9.29. 日期/时间操作符 262
9.30. 日期/时间函数 263
9.31. AT TIME ZONE 变体 274
9.32. 枚举支持函数 277
9.33. 几何操作符 278
9.34. 几何函数 280
9.35. 几何类型转换函数 280
9.36. cidr 和 inet 操作符 282
9.37. cidr 和 inet 函数 282
9.38. macaddr 函数 285
9.39. macaddr8 函数 285
9.40. 文本检索操作符 285
9.41. 文本检索函数 287
9.42. 文本检索调试函数 292
9.43. json 和 jsonb 操作符 307
9.44. 附加的 jsonb 操作符 309
9.45. JSON 创建函数 310
9.46. JSON 处理函数 311
9.47. 序列函数 317
9.48. 数组操作符 322
9.49. 数组函数 323
9.50. 范围操作符 327
9.51. 范围函数 328
9.52. 通用聚合函数 329
9.53. 用于统计的聚合函数 333
9.54. 有序集聚合函数 336
9.55. 假想集聚合函数 337
9.56. 分组操作 337
9.57. 通用窗口函数 338
9.58. 序列生成函数 345
9.59. 下标生成函数 346
9.60. 会话信息函数 348
9.61. 访问权限查询函数 351
9.62. 模式可见性查询函数 354
9.63. 系统目录信息函数 355
9.64. 索引列属性 357
9.65. 索引属性 358
9.66. 索引访问方法属性 358
9.67. 对象信息和寻址函数 359
9.68. 注释信息函数 360
9.69. 事务 ID 和快照 360
9.70. 快照组件 361
9.71. 已提交事务信息 362
9.72. 控制数据函数 362
9.73. pg_control_checkpoint 输出列 362
9.74. pg_control_system 输出列 363
9.75. pg_control_init 输出列 363
9.76. pg_control_recovery 输出列 363
9.77. 配置设定函数 364

9.78. 服务器信号函数 364
9.79. 备份控制函数 365
9.80. 恢复信息函数 367
9.81. 恢复控制函数 368
9.82. 快照同步函数 368
9.83. 复制 SQL 函数 369
9.84. 数据库对象尺寸函数 371
9.85. 数据库对象位置函数 373
9.86. 排序规则管理函数 373
9.87. 索引维护函数 374
9.88. 通用文件访问函数 375
9.89. 咨询锁函数 376
9.90. 表重写信息 380
12.1. 默认解析器的词元类型 428
13.1. 事务隔离级别 453
13.2. 冲突的锁模式 459
13.3. 冲突的行级锁 461
18.1. System V IPC 参数 533
18.2. SSL 服务器文件用法 549
19.1. synchronous_commit 模式 572
19.2. 消息严重级别 590
19.3. 短选项对照 617
21.1. 默认角色 643
23.1. PostgreSQL 字符集 659
23.2. 客户端/服务器字符集转换 662
26.1. 高可用、负载均衡和复制特性矩阵 693
28.1. 动态统计视图 716
28.2. 已收集统计信息的视图 716
28.3. pg_stat_activity 视图 718
28.4. wait_event 描述 721
28.5. pg_stat_replication 视图 728
28.6. pg_stat_wal_receiver 视图 731
28.7. pg_stat_subscription 视图 732
28.8. pg_stat_ssl 视图 732
28.9. pg_stat_archiver 视图 733
28.10. pg_stat_bgwriter 视图 733
28.11. pg_stat_database 视图 733
28.12. pg_stat_database_conflicts 视图 734
28.13. pg_stat_all_tables 视图 735
28.14. pg_stat_all_indexes 视图 736
28.15. pg_statio_all_tables 视图 736
28.16. pg_statio_all_indexes 视图 737
28.17. pg_statio_all_sequences 视图 737
28.18. pg_stat_user_functions 视图 737
28.19. 附加统计函数 738
28.20. 按后端统计函数 739
28.21. pg_stat_progress_vacuum 视图 740
28.22. VACUUM 阶段 741
28.23. 内置 DTrace 探针 742
28.24. 探针参数中使用的已定义类型 747
34.1. SSL 模式描述 847
34.2. libpq/客户端 SSL 文件用法 847
35.1. 面向 SQL 的大对象函数 865

36.1. PostgreSQL 数据类型与 C 变量类型的映射 880
36.2. PGTYPESdate_from_asc 的合法输入格式 899
36.3. PGTYPESdate_fmt_asc 的合法输入格式 902
36.4. rdefmtdate 的合法输入格式 902
36.5. PGTYPEStimestamp_from_asc 的合法输入格式 903
37.1. information_schema_catalog_name 列 984
37.2. administrable_role_authorizations 列 984
37.3. applicable_roles 列 984
37.4. attributes 列 985
37.5. character_sets 列 987
37.6. check_constraint_routine_usage 列 988
37.7. check_constraints 列 988
37.8. collations 列 989
37.9. collation_character_set_applicability 列 989
37.10. column_domain_usage 列 989
37.11. column_options 列 990
37.12. column_privileges 列 990
37.13. column_udt_usage 列 991
37.14. columns 列 991
37.15. constraint_column_usage 列 994
37.16. constraint_table_usage 列 995
37.17. data_type_privileges 列 995
37.18. domain_constraints 列 995
37.19. domain_udt_usage 列 996
37.20. domains 列 996
37.21. element_types 列 998
37.22. enabled_roles 列 1000
37.23. foreign_data_wrapper_options 列 1000
37.24. foreign_data_wrappers 列 1001
37.25. foreign_server_options 列 1001
37.26. foreign_servers 列 1001
37.27. foreign_table_options 列 1002
37.28. foreign_tables 列 1002
37.29. key_column_usage 列 1002
37.30. parameters 列 1003
37.31. referential_constraints 列 1004
37.32. role_column_grants 列 1005
37.33. role_routine_grants 列 1006
37.34. role_table_grants 列 1006
37.35. role_udt_grants 列 1007
37.36. role_usage_grants 列 1007
37.37. routine_privileges 列 1008
37.38. routines 列 1008
37.39. schemata 列 1012
37.40. sequences 列 1012
37.41. sql_features 列 1013
37.42. sql_implementation_info 列 1014
37.43. sql_languages 列 1014
37.44. sql_packages 列 1014
37.45. sql_parts 列 1015
37.46. sql_sizing 列 1015
37.47. sql_sizing_profiles 列 1016

37.48. table_constraints 列 1016
37.49. table_privileges 列 1016
37.50. tables 列 1017
37.51. transforms 列 1018
37.52. triggered_update_columns 列 1018
37.53. triggers 列 1019
37.54. udt_privileges 列 1020
37.55. usage_privileges 列 1021
37.56. user_defined_types 列 1021
37.57. user_mapping_options 列 1022
37.58. user_mappings 列 1023
37.59. view_column_usage 列 1023
37.60. view_routine_usage 列 1023
37.61. view_table_usage 列 1024
37.62. views 列 1024
38.1. 内置 SQL 类型等效的 C 类型 1058
38.2. B-树策略 1094
38.3. Hash 策略 1094
38.4. GiST 二维 “R 树” 策略 1095
38.5. SP-GiST 点策略 1095
38.6. GIN 数组策略 1095
38.7. BRIN Minmax 策略 1096
38.8. B-树支持函数 1096
38.9. Hash 支持函数 1096
38.10. GiST 支持函数 1097
38.11. SP-GiST 支持函数 1097
38.12. GIN 支持函数 1097
38.13. BRIN 支持函数 1098
40.1. 按命令标签分类的事件触发器支持 1129
43.1. 可用的诊断项 1188
43.2. 错误诊断项 1205
241. 按命令类型应用的策略 1576
242. 自动变量 1886
243. 按优先级从低到高排列的 pgbench 操作符 1887
244. pgbench 函数 1888
52.1. 系统目录 2041
52.2. pg_aggregate 列 2043
52.3. pg_am 列 2044
52.4. pg_amop 列 2045
52.5. pg_amproc 列 2046
52.6. pg_attrdef 列 2046
52.7. pg_attribute 列 2047
52.8. pg_authid 列 2049
52.9. pg_auth_members 列 2050
52.10. pg_cast 列 2051
52.11. pg_class 列 2051
52.12. pg_collation 列 2054
52.13. pg_constraint 列 2055
52.14. pg_conversion 列 2057
52.15. pg_database 列 2058
52.16. pg_db_role_setting 列 2059
52.17. pg_default_acl 列 2060

52.18. pg_depend 列 2060
52.19. pg_description 列 2062
52.20. pg_enum 列 2062
52.21. pg_event_trigger 列 2062
52.22. pg_extension 列 2063
52.23. pg_foreign_data_wrapper 列 2064
52.24. pg_foreign_server 列 2064
52.25. pg_foreign_table 列 2065
52.26. pg_index 列 2065
52.27. pg_inherits 列 2067
52.28. pg_init_privs 列 2067
52.29. pg_language 列 2068
52.30. pg_largeobject 列 2069
52.31. pg_largeobject_metadata 列 2069
52.32. pg_namespace 列 2069
52.33. pg_opclass 列 2070
52.34. pg_operator 列 2070
52.35. pg_opfamily 列 2071
52.36. pg_partitioned_table 列 2072
52.37. pg_pltemplate 列 2072
52.38. pg_policy 列 2073
52.39. pg_proc 列 2074
52.40. pg_publication 列 2077
52.41. pg_publication_rel 列 2077
52.42. pg_range 列 2077
52.43. pg_replication_origin 列 2078
52.44. pg_rewrite 列 2078
52.45. pg_seclabel 列 2079
52.46. pg_sequence 列 2079
52.47. pg_shdepend 列 2080
52.48. pg_shdescription 列 2081
52.49. pg_shseclabel 列 2081
52.50. pg_statistic 列 2082
52.51. pg_statistic_ext 列 2083
52.52. pg_subscription 列 2084
52.53. pg_subscription_rel 列 2085
52.54. pg_tablespace 列 2085
52.55. pg_transform 列 2085
52.56. pg_trigger 列 2086
52.57. pg_ts_config 列 2088
52.58. pg_ts_config_map 列 2088
52.59. pg_ts_dict 列 2088
52.60. pg_ts_parser 列 2089
52.61. pg_ts_template 列 2089
52.62. pg_type 列 2090
52.63. typcategory 代码 2094
52.64. pg_user_mapping 列 2095
52.65. 系统视图 2095
52.66. pg_available_extensions 列 2096
52.67. pg_available_extension_versions 列 2097
52.68. pg_config 列 2097
52.69. pg_cursors 列 2098

52.70. pg_file_settings 列 2098
52.71. pg_group 列 2099
52.72. pg_hba_file_rules 列 2099
52.73. pg_indexes 列 2100
52.74. pg_locks 列 2100
52.75. pg_matviews 列 2103
52.76. pg_policies 列 2103
52.77. pg_prepared_statements 列 2104
52.78. pg_prepared_xacts 列 2104
52.79. pg_publication_tables 列 2105
52.80. pg_replication_origin_status 列 2105
52.81. pg_replication_slots 列 2105
52.82. pg_roles 列 2106
52.83. pg_rules 列 2107
52.84. pg_seclabels 列 2108
52.85. pg_sequences 列 2108
52.86. pg_settings 列 2109
52.87. pg_shadow 列 2111
52.88. pg_stats 列 2111
52.89. pg_tables 列 2113
52.90. pg_timezone_abbrevs 列 2113
52.91. pg_timezone_names 列 2114
52.92. pg_user 列 2114
52.93. pg_user_mappings 列 2114
52.94. pg_views 列 2115
64.1. 内置 GiST 操作符类 2234
65.1. 内置 SP-GiST 操作符类 2246
66.1. 内置 GIN 操作符类 2257
67.1. 内置 BRIN 操作符类 2264
67.2. Minmax 操作符类的函数和支持编号 2266
67.3. Inclusion 操作符类的函数和支持编号 2266
69.1. PGDATA 的内容 2270
69.2. 页面布局 2275
69.3. PageHeaderData 布局 2276
69.4. HeapTupleHeaderData 布局 2277
A.1. PostgreSQL 错误代码 2304
B.1. 月份名称 2314
B.2. 星期名称 2314
B.3. 日期/时间字段修饰符 2314
C.1. SQL 关键字 2320
F.1. adminpack 函数 2473
F.2. cube 外部表示 2491
F.3. cube 操作符 2491
F.4. cube 函数 2492
F.5. 基于立方体的地球距离函数 2530
F.6. 基于点的地球距离操作符 2531
F.7. hstore 操作符 2539
F.8. hstore 函数 2540
F.9. intarray 函数 2548
F.10. intarray 操作符 2548
F.11. isn 数据类型 2551
F.12. isn 函数 2552

F.13. ltree 操作符 2558
F.14. ltree 函数 2559
F.15. pg_buffercache 列 2573
F.16. crypt() 支持的算法 2576
F.17. crypt() 的迭代计数 2576
F.18. Hash 算法速度 2577
F.19. 有无 OpenSSL 时的功能概要 2584
F.20. pgrowlocks 输出列 2591
F.21. pg_stat_statements 列 2593
F.22. pgstattuple 输出列 2598
F.23. pgstattuple_approx 输出列 2601
F.24. pg_trgm 函数 2603
F.25. pg_trgm 操作符 2604
F.26. seg 外部表示 2617
F.27. 合法 seg 输入示例 2618
F.28. seg GiST 操作符 2618
F.29. sepgsql 函数 2626
F.30. tablefunc 函数 2632
F.31. connectby 参数 2639
F.32. 用于生成 UUID 的函数 2649
F.33. 返回 UUID 常量的函数 2649
F.34. 函数 2651
F.35. xpath_table 参数 2652

范例清单

8.1. 使用字符类型 149
8.2. 使用 boolean 类型 162
8.3. 使用位串类型 170
9.1. 转换 SQL/XML 输出到 HTML 的 XSLT 样式表 307
10.1. 阶乘操作符类型解析 384
10.2. 字符串连接操作符类型解析 384
10.3. 绝对值与取反操作符类型解析 385
10.4. 数组包含操作符类型解析 386
10.5. 域类型上的自定义操作符 386
10.6. round 函数参数类型解析 388
10.7. 可变参数函数解析 389
10.8. substr 函数类型解析 389
10.9. character 的存储类型转换 391
10.10. UNION 中未充分指定类型时的类型解析 392
10.11. 简单 UNION 中的类型解析 392
10.12. 次序对调的 UNION 中的类型解析 392
10.13. 嵌套 UNION 中的类型解析 393
11.1. 建立一个部分索引以排除常见值 400
11.2. 建立一个部分索引以排除不感兴趣的值 401
11.3. 建立一个部分唯一索引 402
11.4. 不要把部分索引当作分区的替代品 403
20.1. 示例 pg_hba.conf 项 624
20.2. 示例 pg_ident.conf 文件 626
34.1. libpq 示例程序 1 850
34.2. libpq 示例程序 2 852
34.3. libpq 示例程序 3 855
35.1. 用 libpq 操作大对象的示例程序 866
36.1. 示例 SQLDA 程序 921
36.2. 访问大对象的 ECPG 程序 936
42.1. 手工安装 PL/Perl 1170
43.1. 在动态查询中为值加引号 1186
43.2. UPDATE/INSERT 的异常 1204
43.3. 一个 PL/pgSQL 触发器函数 1218
43.4. 一个用于审计的 PL/pgSQL 触发器函数 1218
43.5. 一个用于审计的 PL/pgSQL 视图触发器函数 1219
43.6. 一个用于维护汇总表的 PL/pgSQL 触发器函数 1221
43.7. 使用过渡表审计 1223
43.8. 一个 PL/pgSQL 事件触发器函数 1225
43.9. 从 PL/SQL 移植一个简单的函数到 PL/pgSQL 1233
43.10. 从 PL/SQL 移植一个创建另一个函数的函数到 PL/pgSQL 1234
43.11. 从 PL/SQL 移植一个带有字符串操作以及 OUT 参数的过程到 PL/pgSQL 1235
43.12. 从 PL/SQL 移植一个过程到 PL/pgSQL 1238
F.1. 为 PostgreSQL CSV 日志创建外部表 2533

前言

本书是 PostgreSQL 的官方文档。它由 PostgreSQL 开发人员及其他志愿者在 PostgreSQL 软件开发过程中同步编写。它描述了当前版本的 PostgreSQL 正式支持的全部功能。

为了便于管理有关 PostgreSQL 的大量信息，本书被组织为若干部分。每一部分都面向不同类型的用户，或者面向处于不同PostgreSQL 使用阶段的用户：

- 第 I 部分是面向新用户的非正式介绍。
- 第 II 部分记述了 SQL 查询语言环境，包括数据类型、函数以及用户级性能调优。每个 PostgreSQL 用户都应阅读这一部分。
- 第 III 部分描述了服务器的安装和管理。无论是为自己使用还是为他人提供服务，凡是运行 PostgreSQL 服务器的人都应阅读这一部分。
- 第 IV 部分描述 PostgreSQL 客户端程序的编程接口。
- 第 V 部分包含面向高级用户的服务器可扩展能力相关信息，其中的主题包括用户定义数据类型和函数。
- 第 VI 部分包含有关 SQL 命令、客户端程序和服务器程序的参考信息。该部分以按命令或程序组织的结构化信息为其他部分提供支持。
- 第 VII 部分包含可能对 PostgreSQL 开发人员有用的各类信息。

1. 什么是 PostgreSQL?

PostgreSQL 是一种基于 POSTGRES, Version 4.2[1] 的对象关系数据库管理系统（ORDBMS），后者由加州大学伯克利分校计算机科学系开发。POSTGRES 首创了许多概念，而这些概念直到很久之后才在某些商业数据库系统中出现。

PostgreSQL 是这套原始伯克利代码的开源后继版本。它支持 SQL 标准中的很大一部分，并提供了许多现代特性：

- 复杂查询
- 外键
- 触发器
- 可更新视图
- 事务完整性
- 多版本并发控制

此外，PostgreSQL 还允许用户以多种方式扩展，例如添加新的：

- 数据类型
- 函数
- 操作符
- 聚合函数
- 索引方法
- 过程语言

由于采用宽松的许可证，任何人都可以出于任何目的免费使用、修改和分发PostgreSQL，无论是私人、商业还是学术用途。

[1] https://dsf.berkeley.edu/postgres.html

2. PostgreSQL 简史

如今名为 PostgreSQL 的对象关系数据库管理系统，源自加州大学伯克利分校编写的POSTGRES软件包。经过数十年的发展，PostgreSQL 如今已成为当今最先进的开源数据库。

2.1. 伯克利 POSTGRES 项目

由 Michael Stonebraker 教授领导的 POSTGRES 项目，得到了国防高级研究计划局（DARPA）、陆军研究办公室（ARO）、国家科学基金会（NSF）以及 ESL, Inc. 的资助。POSTGRES 的实现始于 1986 年。该系统的最初概念见于[ston86]，而最初数据模型的定义见于[rowe87]。当时规则系统的设计见于[ston87a]。存储管理器的设计动机和体系结构详见[ston87b]。

自那以后，POSTGRES 经历了几次重大版本发布。第一个“演示版”系统于 1987 年开始运行，并在 1988 年的 ACM-SIGMOD 大会上进行了展示。版本 1 在[ston90a] 中有描述，并于 1989 年 6 月发布给少数外部用户。针对第一代规则系统所受到的批评（[ston89]），规则系统被重新设计（[ston90b]），版本 2 连同新的规则系统于 1990 年 6 月发布。版本 3 于 1991 年问世，增加了对多个存储管理器的支持、改进后的查询执行器以及重写后的规则系统。此后直到 Postgres 95（见下文）之前的后续版本，主要聚焦于可移植性和可靠性。

POSTGRES 被用于实现许多不同的研究和生产应用。这些应用包括金融数据分析系统、喷气发动机性能监测软件包、小行星跟踪数据库、医疗信息数据库以及若干地理信息系统。POSTGRES 还在多所大学中被用作教学工具。最后，Illustra Information Technologies（后来并入 Informix[2]，该公司现归 IBM[3] 所有）接手了这套代码并将其商业化。在 1992 年末，POSTGRES 成为 Sequoia 2000 科学计算项目[4]的主要数据管理器。

1993 年期间，外部用户社区的规模几乎翻了一番。人们越来越清楚地意识到，对原型代码的维护和支持工作占用了大量本应用于数据库研究的时间。为了减轻这一支持负担，伯克利 POSTGRES 项目在版本 4.2 时正式结束。

2.2. Postgres 95

1994 年，Andrew Yu 和 Jolly Chen 为 POSTGRES 添加了一个 SQL 语言解释器。随后，它以 Postgres 95 这一新名称发布到网络上，作为最初的 POSTGRES 伯克利代码的开源后继者自行发展。

Postgres 95 的代码完全采用 ANSI C 编写，并将体积缩减了 25%。许多内部更改提升了性能和可维护性。与 POSTGRES 4.2 版相比，Postgres 95 1.0.x 版本在 Wisconsin Benchmark 上快了大约 30-50%。除了错误修复之外，主要增强还包括：

- 查询语言 PostQUEL 被 SQL 所取代（在服务器中实现）。（接口库 libpq 的名称来源于 PostQUEL。）直到 PostgreSQL（见下文）才支持子查询，但在 Postgres 95 中可以用用户定义的 SQL 函数来模拟。聚合函数也得到了重新实现，并增加了对 `GROUP BY` 查询子句的支持。
- 新增了一个用于交互式 SQL 查询的程序（psql），它使用了 GNU Readline。这在很大程度上取代了旧的 monitor 程序。
- 新增的前端库 `libpgtcl` 支持基于 Tcl 的客户端。示例 shell `pgtclsh` 提供了新的 Tcl 命令，用于让 Tcl 程序与 Postgres 95 服务器交互。

[2] https://www.ibm.com/analytics/informix
[3] https://www.ibm.com/
[4] http://meteora.ucsd.edu/s2k/s2k_home.html

- 大对象接口经过了全面改造。反转大对象成为存储大对象的唯一机制。（反转文件系统已被移除。）
- 实例级规则系统被移除。规则仍然可以作为重写规则使用。
- 源代码中附带了一份简短教程，介绍常规 SQL 特性以及 Postgres 95 的特性。
- 构建时使用的是 GNU make（而不是 BSD make）。此外，Postgres 95 可以用未经修改的 GCC 编译（双精度浮点数的数据对齐问题已修复）。

2.3. PostgreSQL

到了 1996 年，很明显“Postgres 95”这个名字经不起时间的考验。我们选择了一个新名字 PostgreSQL，以体现最初的 POSTGRES 与后来具备 SQL 能力的版本之间的关系。同时，我们将版本号从 6.0 开始，使编号重新回到最初由伯克利 POSTGRES 项目开启的序列中。

许多人出于传统，或因为更容易发音，仍然将PostgreSQL 称为“Postgres”（如今很少全部大写）。这一用法作为昵称或别名已被广泛接受。

在开发 Postgres 95 期间，重点是识别并理解服务器代码中已有的问题。到了 PostgreSQL 阶段，重点则转向增强特性和能力，不过各个方面的工作仍在继续。

自那以后 PostgreSQL 发生了哪些变化，可见附录 E。

3. 约定

以下约定用于命令概要：方括号（`[`和`]`）表示可选部分。大括号（`{`和`}`）和竖线（`|`）表示必须选择其中一个替代项。点（`...`）表示前面的元素可以重复。其他所有符号（包括圆括号）都应按字面含义理解。

在有助于增强清晰度时，SQL 命令前会加上提示符`=>`，shell 命令前会加上提示符`$`。不过，通常并不显示提示符。

一般来说，管理员是负责安装和运行服务器的人。用户则可以是任何正在使用或希望使用 PostgreSQL 系统任一部分的人。不应把这些术语理解得过于狭隘；本书并不对系统管理流程作固定预设。

4. 更多信息

除了本文档，也就是本书之外，还有其他关于PostgreSQL 的资源：

Wiki

PostgreSQL wiki[5] 收录了该项目的 FAQ[6]（常见问题解答）列表、TODO[7] 列表，以及许多其他主题的详细信息。

网站

PostgreSQL 网站[8]提供最新版本的详细信息，以及其他能让你在工作中或日常使用 PostgreSQL 时更加高效的信息。

[5] https://wiki.postgresql.org
[6] https://wiki.postgresql.org/wiki/Frequently_Asked_Questions
[7] https://wiki.postgresql.org/wiki/Todo
[8] https://www.postgresql.org

邮件列表

> 邮件列表是一个让你的问题得到解答、与其他用户分享经验以及联系开发者的好地方。详情请参阅PostgreSQL 网站。

你自己!

> PostgreSQL 是一个开源项目。因此，它依赖用户社区提供持续支持。随着你开始使用 PostgreSQL，无论是通过本文档还是通过邮件列表，你都会依赖他人的帮助。请考虑把你的知识回馈给社区。阅读邮件列表并回答问题。如果你学到了一些文档中尚未包含的内容，就把它写出来并贡献出去。如果你为代码增加了功能，也请将其贡献出来。

5. 缺陷报告指南

当你在 PostgreSQL 中发现缺陷时，我们希望得知此事。你的缺陷报告对于提高 PostgreSQL 的可靠性十分重要，因为即使再怎么小心，也无法保证 PostgreSQL 的每一部分都能在每个平台、任何情况下正常工作。

以下建议旨在帮助你组织缺陷报告，以便能够高效处理。没有人必须遵循这些建议，但这样做通常对各方都有好处。

我们无法承诺立刻修复每一个缺陷。如果该缺陷明显、严重，或者影响大量用户，很可能会有人去调查它。也可能我们会让你升级到更新的版本，看看在新版本中是否也会出现同样的缺陷。或者，我们可能会认定，在某些计划中的重大重写完成之前，这个缺陷无法修复。再或者，它只是过于棘手，而当前还有更重要的事情要处理。如果你需要立即获得帮助，请考虑购买商业支持合同。

5.1. 识别缺陷

在报告缺陷之前，请反复阅读文档，以确认你确实可以完成正在尝试做的事情。如果从文档中无法明确看出某件事是否可行，也请报告；那就是文档中的缺陷。如果程序实际做的事情与文档描述不同，那就是缺陷。这可能包括但不限于以下情况：

- 程序因致命信号而终止，或在终止时给出表明程序本身存在问题的操作系统错误消息。（反例可能是“磁盘满”之类的消息，因为那得由你自己解决。）
- 程序对给定输入产生了错误的输出。
- 程序拒绝接受有效输入（按文档定义）。
- 程序接受了无效输入，却没有给出提示或错误消息。但请记住，你认为的无效输入，可能正是我们认为的扩展功能，或者是对传统做法的兼容。
- 在受支持的平台上，PostgreSQL 无法按照说明完成编译、构建或安装。

这里的“程序”指的是任何可执行程序，不只是后端进程。

运行缓慢或消耗大量资源，并不一定就是缺陷。请阅读文档，或在某个邮件列表中寻求调优应用程序的帮助。不符合 SQL 标准，也不一定是缺陷，除非明确声称该特定特性符合标准。

在继续之前，请查看 TODO 列表和 FAQ，确认你的缺陷是否已经是已知问题。如果你看不懂 TODO 列表中的信息，也请报告你的问题。至少我们可以把 TODO 列表写得更清楚。

5.2. 报告哪些内容

关于缺陷报告，最重要的一点是陈述全部事实，而且只陈述事实。不要猜测你觉得哪里出了问题、“它看起来像是在做什么”，或者程序的哪一部分有错。如果你不熟悉实现，十有八九会猜

错，对我们毫无帮助。即使你熟悉实现，有根据的解释也只能是事实的有益补充，而不能替代事实。如果我们要修复这个缺陷，仍然必须先亲自重现它。陈述这些基本事实并不难做（你大概可以直接从屏幕上复制粘贴），但太多时候，重要细节会被遗漏，因为有人觉得它们不重要，或者认为即使不说清楚，报告也一样能被理解。

以下各项应包含在每一个缺陷报告中：

- 重现问题所需的精确步骤序列，而且必须是从程序启动开始。这些步骤应当是自包含的；如果输出依赖表中的数据，那么仅仅发来一条 `SELECT` 语句，而不附带前面的`CREATE TABLE` 和 `INSERT` 语句，是不够的。我们没有时间去逆向推导你的数据库模式；如果要靠我们自己编造数据，多半会错过问题。

  对于 SQL 相关问题，测试用例的最佳形式是一个可由psql 前端运行、并能展示问题的文件。（请务必确保你的`~/.psqlrc` 启动文件中没有任何内容。）创建这个文件的一个简便方法，是用pg_dump 导出搭建场景所需的表声明和数据，再附上触发问题的查询。我们鼓励你尽量缩小示例规模，但这并非绝对必要。如果缺陷可重现，我们无论如何都能找出来。

  如果你的应用使用的是别的客户端接口，例如PHP，那么请尽量隔离出触发问题的查询。我们大概不会为了重现你的问题去搭建一个 Web 服务器。无论如何，都请记得提供精确的输入文件；不要笼统地猜测问题发生在“大文件”或“中等大小的数据库”等情况下，因为这些信息不够精确，派不上用场。

- 你实际得到的输出。请不要只说它“没起作用”或“崩溃了”。如果有错误消息，请把它贴出来，即使你看不懂。如果程序因操作系统错误而终止，请说明是哪一种错误。如果什么都没发生，也请说清楚。即使你的测试用例导致程序崩溃，或出现其他显而易见的问题，这种情况在我们的平台上也未必会发生。如果可以，最简单的做法就是直接从终端复制输出。

  > **注意**
  >
  > 如果你要报告错误消息，请尽量获取其最详细的形式。在psql 中，请预先执行`\set VERBOSITY verbose`。如果你是从服务器日志中提取消息，请将运行时参数 log_error_verbosity 设为 `verbose`，以便记录全部细节。

  > **注意**
  >
  > 在致命错误的情况下，客户端报告的错误消息可能不包含全部可用信息。也请查看数据库服务器的日志输出。如果你平时没有保留服务器日志，现在正是开始这样做的好时机。

- 你期望得到的输出，也非常重要。如果你只是写“这个命令给了我那个输出。”或“这不是我期望的。”，我们可能会自己运行一遍，扫一眼输出，然后觉得它看起来没问题，也正是我们所期望的。我们不应该花时间去揣摩你的命令背后的确切语义。尤其不要只是说“这不是 SQL 规定的 /Oracle 所做的。”从 SQL 标准中找出正确行为并不是一件轻松的事，而且我们也不都了解其他所有关系数据库的行为。（如果你的问题是程序崩溃，显然可以省略这一项。）

- 任何命令行选项和其他启动选项，包括所有被你改动过、且与问题相关的环境变量或配置文件。同样，请提供精确的信息。如果你使用的是一种预打包发行版，并且它会在系统启动时自动启动数据库服务器，那么你应当尽量弄清楚它是怎么做到的。

- 任何偏离安装说明的做法。

- PostgreSQL 的版本。你可以运行命令 `SELECT version();`来查明所连接服务器的版本。大多数可执行程序也支持 `--version` 选项；至少 `postgres --version` 和 `psql --version` 应当可以工作。如果这个函数或这些选项不存在，就说明你的版本已经过于陈旧，应当升级。如果你运行的是预打包版本，例如 RPM，请说明这一点，并附上该软件包可能带有的子版本号。如果你说的是一个 Git 快照，也请说明，并给出提交 hash 值。

  如果你的版本早于 11.22，我们几乎肯定会建议你升级。每个新版本都会包含大量缺陷修复和改进，因此你在旧版 PostgreSQL 中遇到的缺陷，很可能已经被修复。对于使用旧版 PostgreSQL 的站点，我们只能提供有限支持；如果你需要超出这一范围的帮助，请考虑购买商业支持合同。

- 平台信息。这包括内核名称和版本、C 库、处理器、内存信息等。在大多数情况下，报告供应商和版本就足够了，但不要想当然地认为所有人都知道“Debian”具体包含什么，或者所有人都运行在 x86_64 上。如果你遇到的是安装问题，那么你机器上的工具链信息（编译器、make 等）也是必需的。

不要担心你的缺陷报告会变得很长；这很正常。第一次就把所有内容都报告出来，总比让我们反复追问细节要好。另一方面，如果你的输入文件非常大，先问问是否有人愿意看一看，也是合理的。这里有一篇文章[9]，概述了更多关于报告缺陷的建议。

不要把全部时间都花在找出输入中的哪些变化会让问题消失上。这多半无助于解决问题。如果最后发现这个缺陷无法立刻修复，你仍然有时间去寻找并分享变通办法。还有，再强调一次，不要浪费时间去猜测这个缺陷为什么存在。我们迟早会查明原因。

在编写缺陷报告时，请避免使用含混的术语。整个软件包叫作“PostgreSQL”，有时简称“Postgres”。如果你特指后端进程，请明确说出来，不要只是说“PostgreSQL 崩溃了”。单个后端进程崩溃，与父“postgres”进程崩溃，是完全不同的事情；当你指的是某个单独的后端进程挂掉时，请不要说“服务器崩溃了”，反过来也一样。此外，客户端程序，例如交互式前端“psql”，与后端完全是分开的。请尽量明确问题是在客户端还是服务器端。

5.3. 到哪里报告缺陷

一般来说，请将缺陷报告发送到缺陷报告邮件列表 `<pgsql-bugs@lists.postgresql.org>`。请为电子邮件使用描述性的主题，必要时可以摘取部分错误消息。

另一种方法是填写项目网站[10]上提供的缺陷报告网页表单。通过这种方式提交的缺陷报告，也会被发送到 `<pgsql-bugs@lists.postgresql.org>` 邮件列表。

如果你的缺陷报告涉及安全影响，而且你不希望它立刻出现在公开归档中，请不要发送到 `pgsql-bugs`。安全问题可以私下报告给 `<security@postgresql.org>`。

不要将缺陷报告发送到任何用户邮件列表，例如 `<pgsql-sql@lists.postgresql.org>` 或 `<pgsql-general@lists.postgresql.org>`。这些邮件列表用于回答用户问题，其订阅者通常并不希望收到缺陷报告。更重要的是，他们也不太可能去修复这些缺陷。

此外，请不要把报告发送到开发者邮件列表 `<pgsql-hackers@lists.postgresql.org>`。这个列表是用来讨论 PostgreSQL 开发的，最好把缺陷报告和这类讨论分开。如果这个问题需要更深入的审查，我们可能会选择在 `pgsql-hackers` 上继续讨论你的缺陷报告。

如果你遇到的是文档问题，报告它的最佳地点是文档邮件列表 `<pgsql-docs@lists.postgresql.org>`。请明确指出你对文档的哪一部分不满意。

[9] https://www.chiark.greenend.org.uk/~sgtatham/bugs.html
[10] https://www.postgresql.org/

如果你的缺陷是在某个不受支持平台上出现的可移植性问题，请发送邮件到 <pgsql-hackers@lists.postgresql.org>，这样我们（以及你）就可以共同推进 PostgreSQL 在该平台上的移植工作。

> **注意**
>
> 由于垃圾邮件泛滥，上述所有列表对于未订阅者发来的邮件都会先进行审核。这意味着邮件送达前会有一定延迟。如果你希望订阅这些列表，请访问https://lists.postgresql.org/获取说明。

部分 I. 教程

欢迎阅读 PostgreSQL 教程。下面几章旨在向不熟悉其中任一方面的新手简单介绍 PostgreSQL、关系数据库概念以及 SQL 语言。我们只假定读者具备基本的计算机使用常识，不要求具备特定的 Unix 或编程经验。本部分主要是想让读者通过动手实践了解PostgreSQL 系统的一些重要方面。它并不试图对所涵盖的主题作全面论述。

在学完本教程后，你可能会希望阅读第 II 部分部分，以更系统地掌握 SQL 语言；或者阅读第 IV 部分，了解如何使用PostgreSQL 开发应用程序。自己搭建并管理服务器的读者也应阅读第 III 部分。

目录

1. 从头开始 3
    1.1. 安装 3
    1.2. 架构基础 3
    1.3. 创建数据库 3
    1.4. 访问数据库 5
2. SQL 语言 7
    2.1. 简介 7
    2.2. 概念 7
    2.3. 创建一个新表 8
    2.4. 向表中填充行 8
    2.5. 查询表 9
    2.6. 表之间的连接 11
    2.7. 聚合函数 13
    2.8. 更新 16
    2.9. 删除 17
3. 高级特性 18
    3.1. 简介 18
    3.2. 视图 18
    3.3. 外键 18
    3.4. 事务 19
    3.5. 窗口函数 21
    3.6. 继承 24
    3.7. 小结 26

第 1 章 从头开始

1.1. 安装

当然，在你开始使用PostgreSQL 之前，必须先安装它。PostgreSQL 可能已经安装在你的站点上，可能是因为它随操作系统发行版一起提供，也可能是因为系统管理员已经安装了它。如果是这样，你应该从操作系统文档或系统管理员那里了解如何访问PostgreSQL。

如果你不确定 PostgreSQL 是否已经可用，或者是否可以用它来做实验，那么你可以自行安装。这并不难，而且也是一个不错的练习。PostgreSQL 可由任何非特权用户安装，不需要超级用户（root）权限。

如果你要自己安装 PostgreSQL，请参阅第 16 章中的安装说明，待安装完成后再回到本指南。务必仔细遵循关于设置适当环境变量的小节。

如果你的站点管理员没有按默认方式完成设置，你可能还需要做一些额外工作。例如，如果数据库服务器所在的机器是远程机器，你就需要把`PGHOST` 环境变量设置为数据库服务器机器的名称。环境变量 `PGPORT` 也可能需要设置。归根结底就是：如果你试着启动某个应用程序，而它抱怨无法连接到数据库，那么你应该去咨询站点管理员；如果你自己就是管理员，那就查阅文档以确保你的环境已经正确设置。如果你没有理解上一段的意思，请阅读下一节。

1.2. 架构基础

在继续之前，你应该先了解PostgreSQL 的基本系统架构。了解 PostgreSQL 各部分如何交互，会让本章更容易理解。

用数据库术语来说，PostgreSQL 采用客户端/服务器模型。一次 PostgreSQL 会话由以下协同工作的进程（程序）组成：

- 一个服务器进程，负责管理数据库文件、接受客户端应用到数据库的连接，并代表客户端执行数据库操作。数据库服务器程序叫做`postgres`。
- 用户用来执行数据库操作的客户端（前端）应用。客户端应用的形式可以非常多样：可以是一个文本工具，也可以是一个图形应用程序，或者是一个通过访问数据库来显示网页的 Web 服务器，还可以是专门的数据库维护工具。有些客户端应用随PostgreSQL 发行版一起提供；大多数则由用户开发。

和典型的客户端/服务器应用一样，客户端和服务器可以位于不同的主机上。在这种情况下，它们通过 TCP/IP 网络连接通信。你应该记住这一点，因为在客户端机器上可访问的文件，在数据库服务器机器上可能不可访问（或者只能使用不同的文件名访问）。

PostgreSQL 服务器可以处理来自客户端的多个并发连接。为此，它会为每个连接启动（“forks”）一个新进程。从那时起，客户端和新的服务器进程就无需原始 `postgres` 进程介入而直接通信。因此，这个主服务器进程始终在运行，等待客户端连接，而客户端及其关联的服务器进程则不断创建和退出。（当然，这一切对用户都是不可见的。这里只是为求完整而提及。）

1.3. 创建数据库

检验你能否访问数据库服务器的第一步，就是试着创建一个数据库。一个正在运行的PostgreSQL 服务器可以管理许多数据库。通常会为每个项目或每个用户使用一个独立的数据库。

你的站点管理员可能已经为你创建了一个可用的数据库。如果是这样，你就可以省略这一步，直接跳到下一节。

要创建一个新数据库（本例中命名为 mydb），请使用以下命令：

```
$ createdb mydb
```

如果这条命令没有任何输出，就说明这一步成功了，你可以跳过本节余下的内容。

如果你看到类似下面这样的信息：

```
createdb: command not found
```

那么说明 PostgreSQL 没有正确安装。要么根本没有安装，要么就是你的 shell 搜索路径没有把它包含进去。试着改用绝对路径调用该命令：

```
$ /usr/local/pgsql/bin/createdb mydb
```

你所在站点上的路径可能不同。请联系站点管理员，或者查看安装说明来修正这个问题。

另一种响应可能是这样：

```
createdb: could not connect to database postgres: could not connect to server: No such file or
 directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

这意味着服务器尚未启动，或者它没有在 createdb 期望的位置启动。同样，请查看安装说明或咨询管理员。

另一种响应可能是这样：

```
createdb: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL:  role "joe"
 does not exist
```

其中提到了你自己的登录名。如果管理员没有为你创建 PostgreSQL 用户帐号，就会发生这种情况。（PostgreSQL 用户帐号与操作系统用户帐号是分开的。）如果你是管理员，请参阅第 21 章了解如何创建帐号。你需要切换成安装 PostgreSQL 时所用的操作系统用户（通常是 postgres），才能创建第一个用户帐号。也可能分配给你的PostgreSQL 用户名与操作系统用户名不同；在这种情况下，你需要使用-U 选项，或者设置PGUSER 环境变量，来指定你的 PostgreSQL 用户名。

如果你有用户帐号，但它没有创建数据库所需的权限，那么你会看到下面的信息：

```
createdb: error: database creation failed: ERROR:  permission denied to create database
```

并非每个用户都被授权创建新数据库。如果 PostgreSQL 拒绝为你创建数据库，那么站点管理员就需要授予你创建数据库的权限。遇到这种情况时，请咨询站点管理员。如果PostgreSQL 是你自己安装的，那么在本教程中，你应当以启动服务器时所用的用户帐号登录。[1]

[1] 这之所以可行，是因为PostgreSQL 用户名与操作系统用户帐号是分开的。当你连接到数据库时，可以选择以哪个PostgreSQL 用户名连接；如果不指定，默认就会使用你当前操作系统帐号的名称。恰好总会有一个PostgreSQL 用户帐号，其名称与启动服务器的操作系统用户相同，而该用户也总是有权创建数据库。除了直接以该用户身份登录之外，你还可以在各处使用-U 选项来选择连接时使用的 PostgreSQL 用户名。

你也可以创建其他名称的数据库。PostgreSQL 允许你在一个给定站点上创建任意数量的数据库。数据库名的首字符必须是字母，并且长度限制为 63 字节。一种方便的做法是创建一个与当前用户名同名的数据库。许多工具都将该数据库名视为默认值，因此这样可以少打一些字。要创建这个数据库，只需输入：

```
$ createdb
```

如果你不再想使用某个数据库，可以删除它。例如，如果你是数据库mydb 的所有者（创建者），就可以使用下面的命令销毁它：

```
$ dropdb mydb
```

（对于这条命令，数据库名不会默认成用户帐号名，你始终需要显式指定它。）这个操作会从物理上删除与该数据库关联的所有文件，而且无法撤销，因此务必三思而后行。

关于 createdb 和 dropdb 的更多信息，可分别参见createdb 和 dropdb。

1.4. 访问数据库

一旦你创建了数据库，你就可以通过以下方式访问它：

- 运行 PostgreSQL 的交互式终端程序 psql，它允许你以交互方式输入、编辑并执行SQL 命令。
- 使用现有的图形前端工具，例如pgAdmin，或者使用具备ODBC 或 JDBC 支持的办公套件来创建和操作数据库。这些方式不在本教程讨论范围内。
- 使用若干可用语言绑定之一编写自定义应用程序。关于这些方式的进一步讨论，请见第 IV 部分。

你可能需要启动 psql 来试验本教程中的示例。可以通过输入下面的命令连接到 mydb 数据库：

```
$ psql mydb
```

如果你不提供数据库名，那么它默认就是你的用户帐号名。在上一节使用createdb 时，你已经见识过这种机制了。

在 psql 中，你将看到下面的欢迎信息：

```
psql (11.22)
Type "help" for help.

mydb=>
```

最后一行也可能是：

```
mydb=#
```

这表示你是数据库超级用户；如果这个 PostgreSQL 实例是你自己安装的，那很可能就是这种情况。超级用户意味着你不受访问控制限制。对本教程而言，这一点并不重要。

如果你在启动 psql 时遇到问题，请回到前一节。createdb 和 psql 的诊断信息很相似，如果前者能工作，后者也应该能工作。

psql 打印出的最后一行就是提示符，它表示psql 正在等待你的输入，你可以将SQL 查询输入到 psql 维护的工作区中。试试这些命令：

```
mydb=> SELECT version();
                                                  version
-------------------------------------------------------------------------------------------
 PostgreSQL 11.22 on x86_64-pc-linux-gnu, compiled by gcc (Debian 4.9.2-10) 4.9.2, 64-bit
(1 row)

mydb=> SELECT current_date;
    date
------------
 2016-01-07
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

psql 程序有一些不属于 SQL 命令的内部命令。它们以反斜线 “\” 开头。例如，你可以输入下面的命令来查看各种PostgreSQL SQL 命令的语法帮助：

```
mydb=> \h
```

要退出 psql，输入：

```
mydb=> \q
```

psql 将退出并返回到命令 shell。（要了解更多内部命令，可输入\?，在psql 提示符下执行。）psql 的完整功能见 psql。在本教程中，我们不会明确使用这些功能，但在有帮助时你可以自行使用它们。

第 2 章 SQL 语言

2.1. 简介

本章概述如何使用 SQL 执行简单操作。本教程仅供你入门之用，绝不是一份完整的SQL 教程。已有许多关于 SQL 的书籍，包括[melt93] 和[date97]。你应当知道，PostgreSQL 的某些语言特性是对标准的扩展。

在下面的示例中，我们假定你已经按照前一章所述创建了名为`mydb` 的数据库，并且已经能够启动psql。

本手册中的示例也可以在 PostgreSQL 源代码发行包的 `src/tutorial/`目录中找到。（PostgreSQL 的二进制发行包可能不提供这些文件。）要使用这些文件，请先切换到该目录并运行 make：

```
$ cd .../src/tutorial
$ make
```

这会创建脚本并编译包含用户定义函数和类型的 C 文件。然后，要开始本教程，请执行以下操作：

```
$ psql -s mydb

...

mydb=> \i basics.sql
```

`\i` 命令会从指定文件中读取命令。`psql` 的 `-s` 选项会让你进入单步模式，在将每条语句发送给服务器之前都会暂停。本节使用的命令都位于文件`basics.sql` 中。

2.2. 概念

PostgreSQL 是一种关系数据库管理系统（RDBMS）。这意味着它是一个用于管理存储在关系中的数据的系统。关系本质上是表的数学术语。如今，把数据存储在表中的观念已经十分常见，似乎显而易见，但组织数据库的方式还有很多。类 Unix 操作系统中的文件和目录就是层次数据库的一个例子。较新的发展则是面向对象数据库。

每个表都是一个有名字的行集合。给定表中的每一行都有相同的一组具名列，并且每一列都具有特定的数据类型。虽然列在每一行中的顺序是固定的，但一定要记住，SQL 并不以任何方式保证表内行的顺序（虽然可以为了显示而对它们显式排序）。

表被归入数据库，而由单个PostgreSQL 服务器实例管理的数据库集合构成一个数据库集簇。

2.3. 创建一个新表

你可以通过指定表名以及各列的列名和类型来创建一个新表：

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- 最低温度
    temp_hi         int,           -- 最高温度
    prcp            real,          -- 降水量
    date            date
);
```

你可以在 `psql` 中按这种换行格式输入该命令。`psql` 会识别出该命令在分号出现之前尚未结束。

在 SQL 命令中可以随意使用空白字符（即空格、制表符和换行符）。这意味着你可以按与上面不同的方式对齐该命令，甚至把它全部写在一行上。两个连字符（“--”）表示注释。从它们开始直到行尾的内容都会被忽略。SQL 对关键字和标识符不区分大小写，除非用双引号括起标识符以保留大小写（上面的例子没有这么做）。

`varchar(80)` 指定一种数据类型，它可以存储长度不超过 80 个字符的任意字符串。`int` 是普通的整数类型。`real` 是一种用于存储单精度浮点数的类型。`date` 应当不言自明。（是的，类型为`date` 的列也叫 `date`。这可能方便，也可能令人困惑 – 由你自己决定。）

PostgreSQL 支持标准的 SQL 类型 `int`、`smallint`、`real`、`double precision`、`char(`*N*`)`、`varchar(`*N*`)`、`date`、`time`、`timestamp` 和 `interval`，还支持其他一些通用类型以及丰富的几何类型。PostgreSQL 可以通过任意数量的用户定义数据类型进行定制。因此，类型名在语法中并不是关键字，只有在为了支持SQL 标准中的特殊情况而必须如此时才是例外。

第二个示例将存储城市及其关联的地理位置：

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

`point` 类型就是 PostgreSQL 特有数据类型的一个例子。

最后还要提一句，如果你不再需要某个表，或者想用不同的定义重新创建它，可以使用下面的命令把它移除：

```
DROP TABLE tablename;
```

2.4. 向表中填充行

`INSERT` 语句用于向表中填充行：

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

请注意，所有数据类型的输入格式都相当直观。那些并非简单数值的常量通常必须用单引号（'）括起，就像这个例子一样。`date` 类型实际上可接受的格式相当灵活，不过在本教程中，我们将坚持使用这里展示的这种无歧义格式。

point 类型要求一个坐标对作为输入，如下所示：

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

到目前为止所用的语法要求你记住列的顺序。另一种语法允许你显式列出列：

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

如果愿意，你可以按不同顺序列出列，甚至省略某些列，例如降水量未知时：

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

许多开发者认为，与依赖隐含顺序相比，显式列出列是一种更好的风格。

请把上面展示的所有命令都输入一遍，这样你在下面各节中才有一些数据可供操作。

也可以使用 COPY 从纯文本文件装载大量数据。这通常更快，因为COPY 命令针对这种用途进行了优化，不过灵活性不如INSERT。例如：

```
COPY weather FROM '/home/user/weather.txt';
```

其中，源文件必须可以在运行后端进程的机器上访问，而不是客户端机器，因为后端进程直接读取该文件。有关COPY 命令的更多信息，请参见 COPY。

2.5. 查询表

要从表中检索数据，就要查询该表。SQL 的 SELECT 语句就是用来做这件事的。该语句分为选择列表（列出要返回的列）、表列表（列出从哪些表中检索数据）以及可选的限定条件（指定限制条件的部分）。例如，要检索表 weather 的所有行，键入：

```
SELECT * FROM weather;
```

这里* 是“所有列”的缩写。[1]因此，使用下面的语句也会得到相同结果：

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

输出应该是：

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

[1] 虽然 SELECT * 对于即席查询很有用，但在生产代码中通常认为这是糟糕的风格，因为给表增加一个列就会改变结果。

你可以在选择列表中编写表达式，而不仅仅是简单的列引用。例如，你可以这样做：

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

这样应该得到：

```
     city      | temp_avg |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

注意这里 AS 子句如何用于重新命名输出列。（AS 子句是可选的。）

查询可以被“限定”，方法是添加一个指定所需行的WHERE 子句。WHERE 子句包含一个布尔（真值）表达式，只有使该布尔表达式为真的行才会被返回。在限定条件中可以使用常见的布尔操作符（AND、OR 和 NOT）。例如，下面的查询检索 San Francisco 在雨天的天气记录：

```
SELECT * FROM weather
    WHERE city = 'San Francisco' AND prcp > 0.0;
```

结果：

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

你可以要求查询结果按排序后的顺序返回：

```
SELECT * FROM weather
    ORDER BY city;
```

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 Hayward       |      37 |      54 |      | 1994-11-29
 San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

在这个例子中，排序顺序没有完全指定，因此两个 San Francisco 行可能以任意次序出现。但如果你使用下面的语句，就总会得到上面显示的结果：

```
SELECT * FROM weather
    ORDER BY city, temp_lo;
```

你可以要求从查询结果中去除重复行：

```
SELECT DISTINCT city
    FROM weather;
```

```
     city
---------------
 Hayward
 San Francisco
(2 rows)
```

这里同样要说明，结果行的顺序可能发生变化。你可以把DISTINCT 和 ORDER BY 一起使用，以确保结果一致：[2]

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

2.6. 表之间的连接

到目前为止，我们的查询每次只访问一个表。查询可以同时访问多个表，也可以访问同一个表并同时处理其中的多行。一次访问同一个表或不同表中多行的查询称为连接查询。例如，假设要列出所有天气记录及其对应城市的位置。为此，需要比较city 列在 weather 表各行中的值与 name 列在 cities 表所有行中的值，选出这些值匹配的行对。

> **注意**
>
> 这只是一个概念模型。连接通常会以比真正比较每一个可能的行对更高效的方式执行，不过这对用户是不可见的。

下面的查询可以完成这一点：

```
SELECT *
    FROM weather, cities
    WHERE city = name;
```

```
     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

[2] 在一些数据库系统中，包括较老版本的PostgreSQL，DISTINCT 的实现会自动对行排序，因此 ORDER BY 并非必需。但这并不是 SQL 标准的要求，而且当前的 PostgreSQL 并不保证 DISTINCT 会使行按顺序排列。

注意结果集中的两点：

- 没有与城市 Hayward 对应的结果行。这是因为 `cities` 表中没有与 Hayward 匹配的项，所以连接会忽略`weather` 表中的不匹配行。我们很快就会看到如何修正这一点。
- 有两列都包含城市名。这是正确的，因为`weather` 和 `cities` 表的列列表被串接在一起。不过在实际中这通常并非所愿，因此你很可能会希望显式列出输出列，而不是使用`*`：

```
SELECT city, temp_lo, temp_hi, prcp, date, location
    FROM weather, cities
    WHERE city = name;
```

练习：. 试着确定省略 `WHERE` 子句时，这条查询的含义。

由于这些列的名称都不同，解析器会自动判断它们属于哪个表。如果两个表中有重名列，你就需要限定列名来说明你究竟想要哪一个，例如：

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather, cities
    WHERE cities.name = weather.city;
```

在连接查询中限定所有列名通常被认为是一种好的风格，这样即使以后某个表中又添加了同名列，查询也不会失败。

目前所见的连接查询，也可以采用下面这种形式编写：

```
SELECT *
    FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

这种语法不如上面的语法常用，不过这里展示它是为了帮助理解后续主题。

现在来看看如何把 Hayward 的记录找回来。希望查询扫描 `weather` 表，并为其中每一行查找匹配的`cities` 行。如果找不到匹配行，希望用一些“空值”代替 `cities` 表中的列。这种查询称为外连接。（此前所见的连接都是内连接。）命令如下：

```
SELECT *
    FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 Hayward       |      37 |      54 |      | 1994-11-29 |               |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

这条查询称为左外连接，因为连接操作符左边的表，其每一行都至少在输出中出现一次，而右边的表只有与左表某行匹配的行才会输出。在输出某个没有右表匹配行的左表行时，右表的各列会用空值（null）代替。

练习：. 还有右外连接和全外连接。试着找出它们能做什么。

也可以把一个表与它自身连接，这称为自连接。例如，假设要找出处于其他天气记录温度范围内的所有天气记录。为此，需要比较temp_lo 和 temp_hi 两列在每个 weather 行中的值与 temp_lo 和 temp_hi 两列在其他所有 weather 行中的值。可以使用下面的查询：

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

这里把 weather 表重新标记为 W1 和 W2，以区分连接的左侧和右侧。在其他查询中也可以使用这种别名来减少输入，例如：

```
SELECT *
    FROM weather w, cities c
    WHERE w.city = c.name;
```

你会经常遇到这种缩写方式。

2.7. 聚合函数

与大多数其他关系数据库产品一样，PostgreSQL 支持聚合函数。聚合函数从多行输入中计算出单个结果。例如，有一些聚合函数可在一组行上计算count（计数）、sum（和）、avg（平均值）、max（最大值）和 min（最小值）。

例如，我们可以用下面的语句找出所有记录中最低温度读数的最高值：

```
SELECT max(temp_lo) FROM weather;

 max
-----
  46
(1 row)
```

如果我们想知道该读数出现在哪个城市（或哪些城市），可能会尝试：

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);     错误
```

但这样行不通，因为聚合函数max 不能用于 WHERE 子句中（存在这个限制是因为 WHERE 子句决定哪些行会被纳入聚合计算；因此显然它必须在聚合函数计算之前求值。）不过，这类情况下通常可以把查询改写成能够得到所需结果的形式，这里我们通过使用子查询来做到这一点：

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);


     city
---------------
 San Francisco
(1 row)
```

这样做是可以的，因为子查询是一次独立的计算，它会与外层查询分开计算自己的聚合。

聚合与 GROUP BY 子句结合使用时也非常有用。例如，我们可以得到每个城市的读数个数，以及该城市观测到的最低温度中的最高值：

```
SELECT city, count(*), max(temp_lo)
    FROM weather
    GROUP BY city;
```

```
     city      | count | max
---------------+-------+-----
 Hayward       |     1 |  37
 San Francisco |     2 |  46
(2 rows)
```

这样每个城市都会得到一行输出。每个聚合结果都是在与该城市匹配的表行上计算出来的。我们可以使用 HAVING 来过滤这些分组行：

```
SELECT city, count(*), max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

```
  city   | count | max
---------+-------+-----
 Hayward |     1 |  37
(1 row)
```

这样只会得到所有 temp_lo 值都低于 40 的城市的结果。最后，如果我们只关心名称以“S”开头的城市，可以这样做：

```
SELECT city, count(*), max(temp_lo)
    FROM weather
    WHERE city LIKE 'S%'            -- 1
    GROUP BY city;
```

```
     city      | count | max
---------------+-------+-----
 San Francisco |     2 |  46
(1 row)
```

1 LIKE 操作符执行模式匹配，详细说明见第 9.7 节。

理解聚合与 SQL 的 WHERE 和 HAVING 子句之间的相互作用非常重要。WHERE 和 HAVING 的根本区别在于：WHERE 在分组和聚合计算之前选择输入行（因此它控制哪些行进入聚合计算），而HAVING 在分组和聚合计算之后选择分组行。因此，WHERE 子句中不能包含聚合函数；试图用聚合来决定哪些行应该成为聚合的输入是没有意义的。另一方面，HAVING 子句总是包含聚合函数。（严格地说，也允许编写不使用聚合的HAVING 子句，但那很少有用。同样的条件在WHERE 阶段使用会更高效。）

在前面的例子中，我们可以在WHERE 中应用城市名限制，因为它不需要聚合。这样比把限制放到 HAVING 中更高效，因为我们避免了对所有未通过WHERE 检查的行进行分组和聚合计算。

选择哪些行进入聚合计算的另一种方法是使用 FILTER，这是一个按聚合分别指定的选项：

```
SELECT city, count(*) FILTER (WHERE temp_lo < 45), max(temp_lo)
    FROM weather
    GROUP BY city;
```

```
     city      | count | max
---------------+-------+-----
 Hayward       |     1 |  37
 San Francisco |     1 |  46
(2 rows)
```

FILTER 和 WHERE 非常相似，不同之处在于，它只从其所附加到的那个特定聚合函数的输入中移除行。这里，count 聚合只统计 temp_lo 低于 45 的行；但 max 聚合仍然应用于所有行，因此它仍会得到 46 这个读数。

2.8. 更新

你可以使用 UPDATE 命令更新已有的行。假设你发现 11 月 28 日之后的温度读数全部偏差了 2 度，那么就可以按下面的方式改正数据：

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

看看数据的新状态：

```
SELECT * FROM weather;
```

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. 删除

可以使用 DELETE 命令从表中移除行。假设你不再关心 Hayward 的天气，那么可以用下面的方法把这些行从表中删除：

```
DELETE FROM weather WHERE city = 'Hayward';
```

所有属于 Hayward 的天气记录都会被删除。

```
SELECT * FROM weather;
```

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

对如下形式的语句必须保持警惕：

```
DELETE FROM tablename;
```

如果没有限定条件，DELETE 将从指定表中删除所有行，使其变为空表。系统在执行前不会请求你确认！

第 3 章 高级特性

3.1. 简介

在前一章中，我们已经介绍了如何使用SQL 在 PostgreSQL 中存储和访问数据的基础知识。现在我们将讨论 SQL 的一些更高级特性，它们能够简化管理，并防止数据丢失或损坏。最后，我们还将看看一些PostgreSQL 扩展。

本章有时会引用第 2 章中的示例，对其加以修改或改进，因此事先读过那一章会很有帮助。本章中的一些示例也可以在教程目录中的`advanced.sql` 文件中找到。该文件还包含一些要装载的样例数据，这里不再重复。（关于如何使用该文件，参见第 2.1 节。）

3.2. 视图

回过头看第 2.6 节中的查询。假设天气记录和城市位置的组合清单对你的应用特别有用，但你又不想每次需要它时都键入这条查询。你可以在该查询之上创建一个视图，为该查询命名，以后就可以像引用普通表一样引用它：

```
CREATE VIEW myview AS
    SELECT name, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

大量使用视图是良好的 SQL 数据库设计的一个关键方面。视图允许你通过一致的接口封装表结构的细节，而这些细节可能会随着应用演进而变化。

几乎凡是真实表可以使用的地方，都可以使用视图。在其他视图之上再构建视图也很常见。

3.3. 外键

回想第 2 章中的 `weather` 和 `cities` 表。考虑这样一个问题：你希望确保没有人能向 `weather` 表中插入在 `cities` 表里没有匹配项的行。这称为维护数据的引用完整性。在较为简陋的数据库系统中，这通常是通过先查看`cities` 表、检查是否存在匹配记录，然后再插入新的`weather` 记录或拒绝插入来实现的（如果系统支持这种做法的话）。这种做法有很多问题，而且很不方便，所以PostgreSQL 可以替你完成这件事。

新的表声明如下：

```
CREATE TABLE cities (
        name     varchar(80) primary key,
        location point
);

CREATE TABLE weather (
        city      varchar(80) references cities(name),
        temp_lo   int,
        temp_hi   int,
        prcp      real,
        date      date
);
```

现在试着插入一条无效记录：

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

```
ERROR:  insert or update on table "weather" violates foreign key constraint "weather_city_fkey"
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

外键的行为可以针对应用进行精细调整。本教程不再超出这个简单例子继续展开，更多信息请参见第 5 章。正确使用外键肯定能提高数据库应用的质量，因此强烈建议你了解它们。

3.4. 事务

事务是所有数据库系统中的一个基本概念。事务的要点在于，它把多个步骤打包成一个单一的、要么全部成功要么全部失败的操作。步骤之间的中间状态对其他并发事务不可见；如果发生某种故障使事务无法完成，那么其中任何一步都不会对数据库产生影响。

例如，考虑一个银行数据库，其中保存着各个客户账户的余额，以及各营业网点的总存款余额。假设我们要记录一笔从 Alice 的账户向 Bob 的账户支付 100.00 美元的款项。大幅简化后，相应的 SQL 命令可能是：

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

这些命令的细节在这里并不重要；重要的是，为了完成这个相当简单的操作，涉及了若干次彼此独立的更新。银行管理人员会希望确信这些更新要么全部发生，要么一个也不发生。显然，不能因为系统故障导致 Bob 收到了 100.00 美元，而 Alice 那边却没有被扣款。反过来，如果 Alice 被扣了款而 Bob 没有入账，她也不会满意。我们需要保证：如果操作进行到一半出了问题，到目前为止已执行的步骤都不会生效。把这些更新归为一个事务，就能提供这种保证。事务被称为原子的：从其他事务的角度看，它要么完整发生，要么根本不发生。

我们还希望保证，一旦事务完成并得到数据库系统确认，它确实已经被永久记录下来，即便紧接着发生崩溃也不会丢失。例如，如果我们正在记录 Bob 的一次现金提款，我们当然不希望他刚走

出银行大门，对他账户的扣款就在崩溃后消失。事务型数据库保证，在把事务报告为完成之前，会先将该事务所做的全部更新记入永久存储（即磁盘）。

事务型数据库的另一个重要性质与原子更新的概念密切相关：当多个事务并发运行时，每个事务都不应该看到其他事务尚未完成的更改。例如，如果某个事务正在统计所有营业网点的余额，就不能让它只算进 Alice 所在网点的扣款，却没有算进 Bob 所在网点的入账，反过来也不行。因此，事务不仅在对数据库的持久影响上必须是全有或全无的，在其发生过程中的可见性上也必须如此。一个未结束事务到目前为止所做的更新，对其他事务不可见；等到该事务完成时，所有更新会同时变得可见。

在 PostgreSQL 中，可以通过将事务中的 SQL 命令置于`BEGIN` 和 `COMMIT` 命令之间来建立一个事务。因此，我们的银行事务实际上会是这样：

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- 其他操作
COMMIT;
```

如果事务执行到一半时，我们决定不想提交了（例如刚刚发现 Alice 的余额变成了负数），就可以发出`ROLLBACK` 而不是 `COMMIT`，这样到目前为止的全部更新都会被取消。

PostgreSQL 实际上把每条 SQL 语句都视为在一个事务中执行。如果你没有显式发出 `BEGIN`，那么每条独立语句外围都会隐式包上一对`BEGIN` 和（若成功）`COMMIT`。由 `BEGIN` 和 `COMMIT` 包围起来的一组语句，有时称为事务块。

> **注意**
>
> 某些客户端库会自动发出 `BEGIN` 和 `COMMIT` 命令，因此你可能在没有主动要求的情况下，也得到了事务块的效果。请查阅你所使用接口的文档。

也可以借助保存点，以更细粒度控制事务中的语句。保存点允许你有选择地丢弃事务的一部分，同时提交其余部分。使用`SAVEPOINT` 定义保存点之后，如有需要，可以用`ROLLBACK TO` 回滚到该保存点。事务中从定义保存点到回滚到它之间所做的数据库修改都会被丢弃，而早于该保存点的修改会被保留。

回滚到某个保存点之后，该保存点仍然保持定义状态，因此你可以多次回滚到它。反过来，如果你确定不再需要回滚到某个保存点，可以将其释放，以便系统回收一些资源。请记住，无论是释放某个保存点，还是回滚到某个保存点，都会自动释放在它之后定义的所有保存点。

所有这些都发生在事务块内部，因此其他数据库会话都看不到。等到你提交整个事务块时，被提交的动作才会作为一个整体对其他会话可见，而被回滚的动作则永远不会可见。

回到银行数据库，假设我们从 Alice 的账户扣除了 100.00 美元，并记入 Bob 的账户，后来却发现其实应该记入 Wally 的账户。我们可以像下面这样利用保存点来处理：

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- 糟糕……撤销刚才的操作，改用 Wally 的账户
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

当然，这个例子是过度简化的，但在事务块中借助保存点可以进行大量控制。此外，对于一个由于错误而被系统置为中止状态的事务块，要重新取得控制权，唯一的办法就是使用 ROLLBACK TO；否则就只能把整个事务完全回滚并重新开始。

3.5. 窗口函数

窗口函数会在一组与当前行存在某种关联的表行上执行计算。这与聚合函数能够完成的计算类型类似。不过，窗口函数不会像非窗口聚合调用那样，把多行分组为一条输出行。相反，各行仍然保留各自的独立身份。在幕后，窗口函数能够访问的不仅仅是查询结果中的当前行。

下面的例子说明如何将每个雇员的工资与其所在部门的平均工资进行比较：

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
  depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

前三个输出列直接来自表 empsalary，并且该表中的每一行对应一条输出行。第四列表示在所有depname 值与当前行相同的表行上求得的平均值。（这其实与非窗口的 avg 聚合是同一个函数，但OVER 子句使它被当作窗口函数处理，并在窗口帧上计算。）

窗口函数调用总是包含一个紧跟在窗口函数名和参数之后的 OVER 子句。这正是它在语法上区别于普通函数或非窗口聚合的地方。OVER 子句精确决定如何划分查询中的行，以供窗口函数处理。OVER 内部的 PARTITION BY 子句将具有相同 PARTITION BY 表达式值的行划分为组，也就是分区。对于每一行，窗口函数都是在与当前行处于同一分区的那些行上计算的。

你也可以在 OVER 内部使用 ORDER BY 来控制窗口函数处理行的顺序。（窗口中的 ORDER BY 甚至不必与行的输出顺序一致。）下面是一个例子：

```
SELECT depname, empno, salary,
       rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
```

```
  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

如上所示，rank 函数按 ORDER BY 子句定义的顺序，为当前行所在分区内每个不同的 ORDER BY 值生成一个数值排名。rank 不需要显式参数，因为它的行为完全由OVER 子句决定。

窗口函数所考虑的行，是查询的FROM 子句产生并经 WHERE、GROUP BY 和 HAVING 子句（如果有）过滤后的那个“虚拟表”中的行。例如，由于不满足WHERE 条件而被删除的行，不会被任何窗口函数看到。一个查询可以包含多个窗口函数，它们可以通过不同的OVER 子句以不同方式划分数据，但它们都作用于这个虚拟表所定义的同一组行。

我们已经看到，如果行的顺序并不重要，就可以省略ORDER BY。PARTITION BY 也可以省略，这时所有行构成一个分区。

与窗口函数相关的另一个重要概念是：对于每一行，在其所在分区内有一组行，称为它的窗口帧。有些窗口函数只作用于窗口帧中的行，而不是整个分区中的所有行。默认情况下，如果提供了ORDER BY，则窗口帧包含从分区起始处直到当前行的所有行，再加上根据ORDER BY 子句与当前行相等的所有后续行。如果省略 ORDER BY，默认窗口帧则包含该分区中的所有行。[1]下面是一个使用 sum 的例子：

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

[1] 还可以用其他方式定义窗口帧，但本教程不涉及这些选项。细节见第 4.2.8 节。

```
 salary |  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

上面由于 OVER 子句中没有 ORDER BY，窗口帧与分区相同；而在没有PARTITION BY 的情况下，这个分区就是整张表。换言之，每个求和都是在整张表上计算的，因此每条输出行得到的结果都相同。但是如果加上ORDER BY 子句，结果就会大不一样：

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

```
 salary |  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

这里的求和是从第一条（最低）工资一直累加到当前行，包括与当前行工资相同的所有重复值（注意那些重复工资对应的结果）。

窗口函数只允许出现在查询的 SELECT 列表和 ORDER BY 子句中。它们不允许出现在其他地方，例如GROUP BY、HAVING 和 WHERE 子句中。这是因为从逻辑上讲，它们在这些子句处理完成之后才执行。另外，窗口函数在非窗口聚合函数之后执行。这意味着在窗口函数的参数中包含聚合函数调用是合法的，反过来则不行。

如果需要在窗口计算执行之后再过滤或分组行，可以使用子查询。例如：

```
SELECT depname, empno, salary, enroll_date
FROM
  (SELECT depname, empno, salary, enroll_date,
     rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
     FROM empsalary
  ) AS ss
WHERE pos < 3;
```

上面的查询只显示内部查询中 `rank` 小于 3 的那些行。

当查询涉及多个窗口函数时，可以为每个函数分别写一个独立的`OVER` 子句；但如果多个函数都需要相同的窗口行为，这样写既重复，又容易出错。替代方法是，在`WINDOW` 子句中给每一种窗口行为命名，然后在`OVER` 中引用它。例如：

```
SELECT sum(salary) OVER w, avg(salary) OVER w
  FROM empsalary
  WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

关于窗口函数的更多细节可以在第 4.2.8 节、第 9.21 节、第 7.2.5 节以及 SELECT 参考页中找到。

3.6. 继承

继承是面向对象数据库中的一个概念。它为数据库设计打开了一些有趣的新可能性。

让我们创建两个表：表 `cities` 和表 `capitals`。很自然，州首府也是城市，因此当你列出所有城市时，应该有某种办法能隐式地把州首府也显示出来。如果你很有巧思，也许会想出这样的方案：

```
CREATE TABLE capitals (
  name       text,
  population real,
  elevation  int,    -- (单位为英尺)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,
  elevation  int     -- (单位为英尺)
);

CREATE VIEW cities AS
  SELECT name, population, elevation FROM capitals
    UNION
  SELECT name, population, elevation FROM non_capitals;
```

就查询而言，这样做还行，但一旦需要更新多行，它就会变得很麻烦。

更好的解决办法是：

```
CREATE TABLE cities (
  name       text,
  population real,
  elevation  int     -- （单位为英尺）
);

CREATE TABLE capitals (
  state      char(2) UNIQUE NOT NULL
) INHERITS (cities);
```

在这种情况下，capitals 的一行会从它的父表 cities 继承全部列（name、population 和 elevation）。列 name 的类型是 text，这是PostgreSQL 内置的一种变长字符串类型。capitals 表还有一个额外的列 state，用来表示所在州的缩写。在PostgreSQL 中，一个表可以从零个或多个其他表继承。

例如，下面的查询会找出所有海拔超过 500 英尺的城市名称，其中也包括州首府：

```
SELECT name, elevation
  FROM cities
  WHERE elevation > 500;
```

返回：

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
(3 rows)
```

另一方面，下面的查询会找出所有海拔超过 500 英尺且不是州首府的城市：

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;
```

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
(2 rows)
```

这里，放在cities 前面的 ONLY 表示该查询只针对 cities 表执行，而不包括继承层次中位于 cities 之下的表。我们前面已经讨论过的许多命令 – SELECT、UPDATE 和 DELETE – 都支持这种 ONLY 记法。

注意

尽管继承经常很有用，但它尚未与唯一约束或外键集成，这限制了它的实用性。更多细节见第 5.9 节。

3.7. 小结

PostgreSQL 还有许多特性在这个面向 SQL 新用户的入门教程中没有涉及。本书余下部分会更详细地讨论这些特性。

如果你觉得还需要更多入门材料，请访问 PostgreSQL 官方网站[2]，其中有指向更多资源的链接。

[2] https://www.postgresql.org

部分 II. SQL 语言

本部分描述 SQL 语言在 PostgreSQL 中的用法。我们首先介绍 SQL 的一般语法，然后说明如何创建保存数据的结构、如何填充数据库以及如何查询数据库。中间部分列出可在SQL 命令中使用的数据类型和函数。最后，我们讨论数据库调优中的若干重要方面。

本部分的内容安排方式使初学者能够从头到尾依次阅读，并在不必过多向后查阅的情况下充分理解这些主题。各章力求自成体系，因此高级用户也可以按需单独阅读。相关内容按主题单元以叙述方式展开。希望获得某个命令完整描述的读者，建议查阅第 VI 部分。

读者应当知道如何连接到 PostgreSQL 数据库并发出 SQL 命令。不熟悉这些内容的读者，建议先阅读第 I 部分。SQL 命令通常通过 PostgreSQL 的交互式终端程序 psql 输入，但也可以使用具有类似功能的其他程序。

目录

4. SQL 语法 35
4.1. 词法结构 35
4.1.1. 标识符和关键词 35
4.1.2. 常量 37
4.1.3. 操作符 41
4.1.4. 特殊字符 42
4.1.5. 注释 42
4.1.6. 操作符优先级 42
4.2. 值表达式 44
4.2.1. 列引用 45
4.2.2. 位置参数 45
4.2.3. 下标 45
4.2.4. 字段选择 46
4.2.5. 操作符调用 46
4.2.6. 函数调用 46
4.2.7. 聚合表达式 47
4.2.8. 窗口函数调用 50
4.2.9. 类型转换 52
4.2.10. 排序规则表达式 53
4.2.11. 标量子查询 53
4.2.12. 数组构造器 54
4.2.13. 行构造器 56
4.2.14. 表达式计算规则 57
4.3. 调用函数 59
4.3.1. 使用位置记法 60
4.3.2. 使用命名记法 60
4.3.3. 使用混合记法 61
5. 数据定义 62
5.1. 表基础 62
5.2. 默认值 63
5.3. 约束 64
5.3.1. 检查约束 64
5.3.2. 非空约束 67
5.3.3. 唯一约束 68
5.3.4. 主键 69
5.3.5. 外键 70
5.3.6. 排他约束 73
5.4. 系统列 73
5.5. 修改表 74
5.5.1. 增加列 74
5.5.2. 移除列 75
5.5.3. 增加约束 75
5.5.4. 移除约束 76
5.5.5. 更改列的默认值 76
5.5.6. 修改列的数据类型 76
5.5.7. 重命名列 77
5.5.8. 重命名表 77
5.6. 权限 77
5.7. 行安全性策略 78
5.8. 模式 84
5.8.1. 创建模式 85

5.8.2. 公共模式 86
5.8.3. 模式搜索路径 86
5.8.4. 模式和权限 87
5.8.5. 系统目录模式 87
5.8.6. 使用方式 88
5.8.7. 可移植性 88
5.9. 继承 89
5.9.1. 注意事项 92
5.10. 表分区 92
5.10.1. 概述 92
5.10.2. 声明式分区 93
5.10.3. 使用继承的分区 98
5.10.4. 分区剪枝 102
5.10.5. 分区和约束排除 104
5.10.6. 声明式分区最佳实践 104
5.11. 外部数据 105
5.12. 其他数据库对象 105
5.13. 依赖跟踪 106
6. 数据操纵 108
6.1. 插入数据 108
6.2. 更新数据 109
6.3. 删除数据 110
6.4. 从被修改的行中返回数据 110
7. 查询 112
7.1. 概述 112
7.2. 表表达式 112
7.2.1. FROM 子句 113
7.2.2. WHERE 子句 121
7.2.3. GROUP BY 和 HAVING 子句 122
7.2.4. GROUPING SETS、CUBE 和 ROLLUP 125
7.2.5. 窗口函数处理 128
7.3. 选择列表 128
7.3.1. 选择列表项 128
7.3.2. 列标签 129
7.3.3. DISTINCT 129
7.4. 组合查询 130
7.5. 行排序 131
7.6. LIMIT 和 OFFSET 132
7.7. VALUES 列表 133
7.8. WITH 查询（公共表表达式） 134
7.8.1. WITH 中的 SELECT 135
7.8.2. WITH 中的数据修改语句 139
8. 数据类型 141
8.1. 数字类型 142
8.1.1. 整数类型 143
8.1.2. 任意精度数值 143
8.1.3. 浮点类型 144
8.1.4. serial 类型 146
8.2. 货币类型 147
8.3. 字符类型 147
8.4. 二进制数据类型 149
8.4.1. bytea 的十六进制格式 150
8.4.2. bytea 的转义格式 150

8.5. 日期/时间类型 151
8.5.1. 日期/时间输入 153
8.5.2. 日期/时间输出 156
8.5.3. 时区 157
8.5.4. 间隔输入 158
8.5.5. 间隔输出 160
8.6. 布尔类型 161
8.7. 枚举类型 162
8.7.1. 枚举类型的声明 162
8.7.2. 排序 163
8.7.3. 类型安全性 163
8.7.4. 实现细节 164
8.8. 几何类型 164
8.8.1. 点 165
8.8.2. 直线 165
8.8.3. 线段 165
8.8.4. 方框 165
8.8.5. 路径 166
8.8.6. 多边形 166
8.8.7. 圆 166
8.9. 网络地址类型 167
8.9.1. `inet` 167
8.9.2. `cidr` 167
8.9.3. `inet` 与 `cidr` 168
8.9.4. `macaddr` 168
8.9.5. `macaddr8` 169
8.10. 位串类型 169
8.11. 文本检索类型 170
8.11.1. `tsvector` 171
8.11.2. `tsquery` 172
8.12. UUID 类型 173
8.13. XML 类型 174
8.13.1. 创建 XML 值 175
8.13.2. 编码处理 175
8.13.3. 访问 XML 值 176
8.14. JSON 类型 176
8.14.1. JSON 输入和输出语法 177
8.14.2. 有效地设计 JSON 文档 178
8.14.3. `jsonb` 包含与存在 179
8.14.4. `jsonb` 索引 180
8.14.5. 转换 183
8.15. 数组 183
8.15.1. 数组类型的声明 184
8.15.2. 数组值输入 185
8.15.3. 访问数组 186
8.15.4. 修改数组 189
8.15.5. 在数组中搜索 192
8.15.6. 数组输入和输出语法 193
8.16. 复合类型 194
8.16.1. 复合类型的声明 195
8.16.2. 构造复合值 196
8.16.3. 访问复合类型 197
8.16.4. 修改复合类型 197

8.16.5. 在查询中使用复合类型 198
8.16.6. 复合类型的输入和输出语法 200
8.17. 范围类型 201
8.17.1. 内置范围类型 201
8.17.2. 示例 202
8.17.3. 包含界限与排除界限 202
8.17.4. 无限（无界）范围 202
8.17.5. 范围输入/输出 203
8.17.6. 构造范围 204
8.17.7. 离散范围类型 204
8.17.8. 定义新的范围类型 204
8.17.9. 索引 205
8.17.10. 范围上的约束 206
8.18. 域类型 206
8.19. 对象标识符类型 207
8.20. pg_lsn 类型 208
8.21. 伪类型 209
9. 函数和操作符 211
9.1. 逻辑操作符 211
9.2. 比较函数和操作符 211
9.3. 数学函数和操作符 214
9.4. 字符串函数和操作符 220
9.4.1. format 233
9.5. 二进制串函数和操作符 235
9.6. 位串函数和操作符 239
9.7. 模式匹配 240
9.7.1. LIKE 240
9.7.2. SIMILAR TO 正则表达式 241
9.7.3. POSIX 正则表达式 242
9.8. 数据类型格式化函数 255
9.9. 日期/时间函数和操作符 262
9.9.1. EXTRACT, date_part 269
9.9.2. date_trunc 273
9.9.3. AT TIME ZONE 274
9.9.4. 当前日期/时间 275
9.9.5. 延时执行 277
9.10. 枚举支持函数 277
9.11. 几何函数和操作符 278
9.12. 网络地址函数和操作符 281
9.13. 文本检索函数和操作符 285
9.14. XML 函数 293
9.14.1. 产生 XML 内容 293
9.14.2. XML 谓词 298
9.14.3. 处理 XML 299
9.14.4. 将表映射到 XML 303
9.15. JSON 函数和操作符 307
9.16. 序列操作函数 317
9.17. 条件表达式 319
9.17.1. CASE 320
9.17.2. COALESCE 321
9.17.3. NULLIF 321
9.17.4. GREATEST 和 LEAST 322
9.18. 数组函数和操作符 322

9.19. 范围函数和操作符 326
9.20. 聚合函数 329
9.21. 窗口函数 338
9.22. 子查询表达式 340
9.22.1. EXISTS 340
9.22.2. IN 340
9.22.3. NOT IN 341
9.22.4. ANY/SOME 341
9.22.5. ALL 342
9.22.6. 单一行比较 342
9.23. 行和数组比较 342
9.23.1. IN 342
9.23.2. NOT IN 343
9.23.3. ANY/SOME（数组） 343
9.23.4. ALL（数组） 344
9.23.5. 行构造器比较 344
9.23.6. 复合类型比较 345
9.24. 集合返回函数 345
9.25. 系统信息函数 348
9.26. 系统管理函数 363
9.26.1. 配置设定函数 363
9.26.2. 服务器信号函数 364
9.26.3. 备份控制函数 365
9.26.4. 恢复控制函数 367
9.26.5. 快照同步函数 368
9.26.6. 复制函数 369
9.26.7. 数据库对象管理函数 371
9.26.8. 索引维护函数 374
9.26.9. 通用文件访问函数 374
9.26.10. 咨询锁函数 376
9.27. 触发器函数 378
9.28. 事件触发器函数 378
9.28.1. 在命令结束处捕捉更改 378
9.28.2. 处理被 DDL 命令删除的对象 379
9.28.3. 处理表重写事件 380
10. 类型转换 381
10.1. 概述 381
10.2. 操作符 382
10.3. 函数 386
10.4. 值存储 390
10.5. UNION、CASE 及相关结构 391
10.6. SELECT 输出列 393
11. 索引 394
11.1. 简介 394
11.2. 索引类型 395
11.3. 多列索引 397
11.4. 索引和 ORDER BY 397
11.5. 组合多个索引 398
11.6. 唯一索引 399
11.7. 表达式索引 399
11.8. 部分索引 400
11.9. 仅索引扫描和覆盖索引 403
11.10. 操作符类和操作符族 406

11.11. 索引和排序规则 407
11.12. 检查索引使用情况 408
12. 全文检索 409
12.1. 简介 409
12.1.1. 什么是文档? 410
12.1.2. 基本文本匹配 410
12.1.3. 配置 412
12.2. 表和索引 413
12.2.1. 搜索表 413
12.2.2. 创建索引 414
12.3. 控制文本检索 415
12.3.1. 解析文档 415
12.3.2. 解析查询 416
12.3.3. 搜索结果排名 419
12.3.4. 高亮结果 421
12.4. 附加特性 423
12.4.1. 操纵文档 423
12.4.2. 操纵查询 423
12.4.3. 用于自动更新的触发器 426
12.4.4. 收集文档统计数据 428
12.5. 解析器 428
12.6. 词典 430
12.6.1. 停用词 431
12.6.2. 简单词典 431
12.6.3. 同义词词典 433
12.6.4. 分类词典 434
12.6.5. Ispell 词典 438
12.6.6. Snowball 词典 440
12.7. 配置示例 440
12.8. 测试和调试文本检索 442
12.8.1. 配置测试 442
12.8.2. 解析器测试 444
12.8.3. 词典测试 445
12.9. 文本检索的首选索引类型 446
12.10. psql 支持 447
12.11. 限制 451
13. 并发控制 452
13.1. 简介 452
13.2. 事务隔离 452
13.2.1. 读已提交隔离级别 453
13.2.2. 可重复读隔离级别 454
13.2.3. 可串行化隔离级别 455
13.3. 显式锁定 457
13.3.1. 表级锁 458
13.3.2. 行级锁 460
13.3.3. 页级锁 461
13.3.4. 死锁 461
13.3.5. 咨询锁 462
13.4. 应用级别的数据一致性检查 463
13.4.1. 用可串行化事务强制一致性 463
13.4.2. 使用显式阻塞锁强制一致性 464
13.5. 注意事项 464
13.6. 锁定和索引 465
14. 性能提示 466

14.1. 使用 EXPLAIN 466
14.1.1. EXPLAIN 基础 466
14.1.2. EXPLAIN ANALYZE 472
14.1.3. 注意事项 476
14.2. 规划器使用的统计信息 477
14.2.1. 单列统计信息 477
14.2.2. 扩展统计信息 478
14.3. 用显式 JOIN 子句控制规划器 480
14.4. 填充一个数据库 482
14.4.1. 禁用自动提交 482
14.4.2. 使用 COPY 483
14.4.3. 移除索引 483
14.4.4. 移除外键约束 483
14.4.5. 增加 maintenance_work_mem 483
14.4.6. 增加 max_wal_size 483
14.4.7. 禁用 WAL 归档和流复制 483
14.4.8. 事后运行 ANALYZE 484
14.4.9. 关于 pg_dump 的一些注记 484
14.5. 非持久设置 485
15. 并行查询 486
15.1. 并行查询如何工作 486
15.2. 何时可以使用并行查询？ 486
15.3. 并行计划 487
15.3.1. 并行扫描 488
15.3.2. 并行连接 488
15.3.3. 并行聚合 488
15.3.4. 并行追加 489
15.3.5. 并行计划提示 489
15.4. 并行安全性 489
15.4.1. 为函数和聚合指定并行标签 490

第 4 章 SQL 语法

本章描述 SQL 的语法。它是理解后续各章的基础，后续各章将详细介绍如何使用 SQL 命令定义和修改数据。

我们也建议已经熟悉 SQL 的用户仔细阅读本章，因为其中包含一些在各 SQL 数据库之间实现并不一致的规则和概念，以及一些 PostgreSQL 特有的规则和概念。

4.1. 词法结构

SQL 输入由一系列命令组成。一个命令由一系列词元组成，并以分号（“;”）终止。输入流的结束也会终止一个命令。哪些词元是合法的，取决于具体命令的语法。

一个词元可以是关键词、标识符、带引号的标识符、字面量（或常量），或者特殊字符符号。词元通常由空白（空格、制表符、换行）分隔，但如果不存在歧义，则不必如此（一般只有特殊字符与其他类型的词元相邻时才会这样）。

例如，下面是一个（语法上）合法的 SQL 输入：

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

这是一个由三个命令组成的序列，每行一个命令（虽然这并非必须；一行中可以包含多个命令，命令也可以有意义地跨多行书写）。

此外，注释也可以出现在 SQL 输入中。它们不是词元，实际上等价于空白。

在哪些词元标识命令、哪些词元是操作数或参数这一点上，SQL 语法并不十分一致。最前面的几个词元通常是命令名，因此在上面的示例中，我们通常会说是“SELECT”、“UPDATE”和“INSERT”命令。但是例如 `UPDATE` 命令总是要求在特定位置出现一个 `SET` 词元，而这种形式的 `INSERT` 也要求有一个 `VALUES` 才算完整。每个命令的精确语法规则见第 VI 部分。

4.1.1. 标识符和关键词

上例中的 `SELECT`、`UPDATE` 或 `VALUES` 这类词元都是关键词，也就是在 SQL 语言中具有固定含义的词。词元 `MY_TABLE` 和 `A` 则是标识符的例子。它们标识表、列或其他数据库对象的名称，这取决于它们所在的命令。因此，它们有时也简称为“名称”。关键词和标识符具有相同的词法结构，这意味着如果不了解该语言，就无法判断一个词元究竟是标识符还是关键词。完整的关键词列表可以在附录 C 中找到。

SQL 标识符和关键词必须以字母（`a-z`，也包括带变音符的字母和非拉丁字母）或下划线（`_`）开头。标识符或关键词中的后续字符可以是字母、下划线、数字（`0-9`）或美元符号（`$`）。注意，按照 SQL 标准的字面规定，标识符中不允许使用美元符号，因此它可能会降低应用的可移植性。SQL 标准不会定义包含数字、以下划线开头或以下划线结尾的关键词，因此这种形式的标识符可以安全地避免与未来标准扩展发生冲突。

系统对标识符最多只使用 `NAMEDATALEN`-1 个字节；在命令中可以写更长的名称，但它们会被截断。默认情况下，`NAMEDATALEN` 为 64，因此标识符的最大长度是 63 字节。如果这个限制有问题，可以通过修改 `src/include/pg_config_manual.h` 中的 `NAMEDATALEN` 常量来提高。

关键词和未加引号的标识符不区分大小写。因此：

```
UPDATE MY_TABLE SET A = 5;
```

也可以等价地写成：

```
uPDaTE my_TabLE SeT a = 5;
```

一种常见约定是关键词使用大写、名称使用小写，例如：

```
UPDATE my_table SET a = 5;
```

还有第二类标识符：定界标识符或带引号的标识符。它通过把任意字符序列括在双引号中形成（"）。定界标识符始终是标识符，绝不会是关键词。因此，"select" 可以用来引用名为 “select” 的列或表，而不加引号的 select 会被视为关键词，因此在期望表名或列名的位置使用时会导致解析错误。这个示例可以用带引号的标识符写成：

```
UPDATE "my_table" SET "a" = 5;
```

带引号的标识符可以包含除编码为零的字符之外的任意字符。（如果要包含双引号，就写两个双引号。）这使得可以构造原本不可能的表名或列名，例如包含空格或和号的名称。长度限制仍然适用。

带引号标识符的一种变体允许包含按代码点标识的转义 Unicode 字符。这种变体在开头双引号之前紧接着写 U&（大写或小写的字母 U 后接和号），中间没有空格，例如 U&"foo"。（注意这会与操作符 & 产生歧义。请在该操作符两侧加空格以避免这个问题。）在引号内，可以通过写一个反斜线后跟 4 位十六进制代码点编号，或者一个反斜线后跟加号再跟 6 位十六进制代码点编号，来指定 Unicode 字符。例如，标识符 "data" 可以写成：

```
U&"d\0061t\+000061"
```

下面这个稍微复杂一些的示例，用西里尔字母写出了俄语单词 “slon”（大象）：

```
U&"\0441\043B\043E\043D"
```

如果想使用反斜线以外的转义字符，可以在字符串后用 UESCAPE 子句指定，例如：

```
U&"d!0061t!+000061" UESCAPE '!'
```

转义字符可以是除十六进制数字、加号、单引号、双引号或空白字符之外的任何单个字符。注意，转义字符要写在单引号内，而不是双引号内。

如果要在标识符中按字面包含转义字符本身，把它写两次即可。

Unicode 转义语法只有在服务器编码为 UTF8 时才有效。使用其他服务器编码时，只能指定 ASCII 范围内的代码点（不超过 \007F）。4 位或 6 位转义形式都可以用来指定 UTF-16 代理对，以组合出代码点大于 U+FFFF 的字符，尽管从技术上说，由于有 6 位形式，这样做并无必要。（代理对不会被直接存储，而是会合并成一个代码点，然后以 UTF-8 编码。）

给标识符加上引号也会使其区分大小写，而不加引号的名称总是会被折叠成小写。例如，标识符 FOO、foo 和 "foo" 在 PostgreSQL 中被视为相同，但 "Foo" 和 "FOO" 与这三个都不同，彼此之间也不同。（PostgreSQL 将不加引号的名称折叠为小写，这与 SQL 标准不兼容；标准规定，不加引号的名称应折叠为大写。因此，按照标准，foo 应当等价于 "FOO"，而不是 "foo"。如果你想编写可移植的应用，建议对某个特定名称要么始终加引号，要么始终不加引号。）

4.1.2. 常量

在 PostgreSQL 中有三种隐式类型常量：字符串、位串和数字。也可以为常量指定显式类型，这样系统就能更准确地表示并更高效地处理它们。这些方式将在后续小节中讨论。

4.1.2.1. 字符串常量

SQL 中的字符串常量是由单引号（'）括起的任意字符序列，例如 'This is a string'。要在字符串常量中包含单引号字符，请写两个相邻的单引号，例如 'Dianne''s horse'。注意，这不同于双引号字符（"）。

两个只由空白及至少一个新行分隔的字符串常量会被连接在一起，并且将作为一个写在一起的字符串常量来对待。例如：

```
SELECT 'foo'
'bar';
```

等同于：

```
SELECT 'foobar';
```

但是：

```
SELECT 'foo'      'bar';
```

则不是合法的语法（这种有些奇怪的行为是 SQL 指定的，PostgreSQL 遵循了该标准）。

4.1.2.2. C 风格转义的字符串常量

PostgreSQL 也接受“转义”字符串常量，这也是 SQL 标准的一个扩展。一个转义字符串常量可以通过在开单引号前面写一个字母E（大写或小写形式）来指定，例如E'foo'（当一个转义字符串常量跨行时，只在第一个开引号之前写E）。在一个转义字符串内部，一个反斜线字符（\）会开始一个 C 风格的反斜线转义序列，在其中反斜线和后续字符的组合表示一个特殊的字节值（如表 4.1 中所示）。

表 4.1. 反斜线转义序列

反斜线转义序列	解释
\b	退格
\f	换页
\n	换行
\r	回车
\t	制表符
o, *oo*, *ooo* (*o* = 0 - 7)	八进制字节值
\x*h*, \x*hh* (*h* = 0 - 9, A - F)	十六进制字节值

反斜线转义序列	解释
\u*xxxx*, \U*xxxxxxxx* (*x* = 0 - 9, A - F)	16 或 32 位十六进制 Unicode 字符值

跟随在一个反斜线后面的任何其他字符被当做其字面意思。因此，要包括一个反斜线字符，请写两个反斜线（\\）。在一个转义字符串中包括一个单引号除了普通方法'' 之外，还可以写成 \'。

你有责任确保所创建的字节序列能够构成服务器字符集编码中的有效字符，尤其是在使用八进制或十六进制转义时。当服务器编码为 UTF-8 时，应改用 Unicode 转义或者第 4.1.2.3 节中说明的另一种 Unicode 转义语法。（另一种做法是手工进行 UTF-8 编码并写出各字节，但那会非常繁琐。）

Unicode 转义语法只有在服务器编码为 UTF8 时才能完全有效。使用其他服务器编码时，只能指定 ASCII 范围内的代码点（不超过 \u007F）。4 位和 8 位形式都可以用来指定 UTF-16 代理对，以组合出代码点大于 U+FFFF 的字符，尽管从技术上说，由于有 8 位形式，这样做并无必要。（在服务器编码为 UTF8 时使用代理对，它们会先合并成一个代码点，然后以 UTF-8 编码。）

> **小心**
>
> 如果配置参数 standard_conforming_strings 为 off，那么PostgreSQL 对常规字符串常量和转义字符串常量中的反斜线转义都识别。不过，从PostgreSQL 9.1 开始，该参数的默认值为on，意味着只在转义字符串常量中识别反斜线转义。这种行为更兼容标准，但是可能使依赖历史行为（始终识别反斜线转义）的应用无法正常工作。作为一种变通，你可以设置该参数为off，但最好修改应用，不再使用反斜线转义。如果你需要使用一个反斜线转义来表示一个特殊字符，为该字符串常量写上一个E。
>
> 在 standard_conforming_strings 之外，配置参数escape_string_warning 和 backslash_quote 也决定了如何对待字符串常量中的反斜线。

编码为零的字符不能出现在字符串常量中。

4.1.2.3. 带有 Unicode 转义的字符串常量

PostgreSQL 还支持另一种字符串转义语法，它允许用代码点指定任意 Unicode 字符。Unicode 转义字符串常量在开引号前紧接着写 U&（大写或小写字母 U 后跟和号），中间没有任何空白，例如 U&'foo'。（注意这会与操作符 & 产生歧义。请在该操作符两侧加空格以避免这个问题。）在引号内，可以通过写一个反斜线后跟 4 位十六进制代码点编号，或者一个反斜线后跟加号再跟 6 位十六进制代码点编号，来指定 Unicode 字符。例如，字符串 'data' 可以写成

```
U&'d\0061t\+000061'
```

下面这个稍微复杂一些的示例，用西里尔字母写出了俄语单词“slon”（大象）：

```
U&'\0441\043B\043E\043D'
```

如果想要一个不是反斜线的转义字符，可以在字符串之后使用UESCAPE 子句来指定，例如：

```
U&'d!0061t!+000061' UESCAPE '!'
```

转义字符可以是除十六进制数字、加号、单引号、双引号或空白字符之外的任意单个字符。

Unicode 转义语法只有在服务器编码为 UTF8 时才有效。使用其他服务器编码时，只能指定 ASCII 范围内的代码点（不超过 \007F）。4 位和 6 位形式都可以用来指定 UTF-16 代理对，以组合出代码点大于 U+FFFF 的字符，尽管从技术上说，由于有 6 位形式，这样做并无必要。（在服务器编码为 UTF8 时使用代理对，它们会先合并成一个代码点，然后以 UTF-8 编码。）

此外，字符串常量的 Unicode 转义语法只有在配置参数standard_conforming_strings 开启时才有效。这是因为否则这种语法可能会混淆解析 SQL 语句的客户端，从而导致 SQL 注入以及类似的安全问题。如果该参数被设置为 off，这种语法将被拒绝并报错。

如果要在字符串中按字面包含转义字符本身，把它写两次即可。

4.1.2.4. 美元引用的字符串常量

虽然指定字符串常量的标准语法通常很方便，但当目标字符串中包含很多单引号时，就会变得难以阅读，因为每个单引号都必须写成两个。为了在这种情况下让查询更易读，PostgreSQL 提供了另一种编写字符串常量的方式，称为“美元引用”（dollar quoting）。一个美元引用字符串常量由美元符号（$）、一个由零个或多个字符组成的可选“标签”、另一个美元符号、组成字符串内容的任意字符序列、一个美元符号、开始该美元引用时使用的同一个标签以及最后一个美元符号组成。例如，下面是使用美元引用指定字符串“Dianne's horse”的两种不同写法：

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

注意，在美元引用字符串内部，单引号可以直接使用而无须转义。事实上，美元引用字符串内部的任何字符都不会被转义：字符串内容总是按字面方式书写。反斜线不是特殊字符，美元符号也不是，除非它们构成了与开头标签匹配的序列。

可以通过在每个嵌套层级选择不同的标签，来嵌套美元引用字符串常量。这种做法最常见于编写函数定义时。例如：

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

这里，序列 q[\t\r\n\v\\]q 表示一个美元引用的字面字符串 [\t\r\n\v\\]，当函数体被 PostgreSQL 执行时，它会被识别。但由于该序列与外层美元引用定界符 $function$ 不匹配，就外层字符串而言，它只不过是常量中的一部分字符。

美元引用字符串的标签（如果有）遵循与未加引号标识符相同的规则，但不能包含美元符号。标签区分大小写，因此 tagString contenttag 是正确的，而 TAGString content tag 则不正确。

如果美元引用字符串位于某个关键词或标识符之后，必须用空白把它们分隔开，否则美元引用定界符会被当作前一个标识符的一部分。

美元引用不是 SQL 标准的一部分，但在书写复杂字符串字面量时，它往往比符合标准的单引号语法更方便。当要表示的字符串常量位于其他常量内部时，它尤其有用，这种情况在过程函数定义中很常见。如果使用单引号语法，上一个示例中的每个反斜线都必须写成四个反斜线；在解析原始字符串常量时它们会缩减为两个反斜线，而在函数执行期间重新解析内层字符串常量时又会变成一个。

4.1.2.5. 位串常量

位串常量看起来像常规字符串常量在开引号之前（中间无空白）加了一个 `B`（大写或小写形式），例如`B'1001'`。位串常量中允许的字符只有 `0` 和 `1`。

或者，也可以用十六进制记法指定位串常量，即在开头写一个 `X`（大写或小写形式），例如 `X'1FF'`。这种记法等价于把每个十六进制位替换成四个二进制位得到的位串常量。

两种形式的位串常量可以以常规字符串常量相同的方式跨行继续。美元引用不能被用在位串常量中。

4.1.2.6. 数字常量

数字常量接受下列一般形式：

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

其中 *digits* 是一个或多个十进制数字（0 到 9）。若使用小数点，则小数点前后至少有一侧必须有数字。若使用指数标记（`e`），其后至少必须有一位数字。常量中不能嵌入空格或其他字符。请注意，前导正号或负号实际上不属于常量本身，而是作用于常量的操作符。

这些是合法数字常量的示例：

```
42
3.5
4.
.001
5e2
1.925e-3
```

如果一个不包含小数点和指数的数字常量的值适合类型 `integer`（32 位），它首先被假定为类型`integer`。否则如果它的值适合类型 `bigint`（64 位），它被假定为类型`bigint`。再否则它会被取做类型 `numeric`。包含小数点和/或指数的常量总是首先被假定为类型 `numeric`。

为数字常量最初指定的数据类型，只是类型解析算法的起点。在大多数情况下，常量会根据上下文被自动强制转换成最合适的类型。必要时，你可以通过类型转换强制把一个数值解释为特定的数据类型。例如，可以通过下面的写法强制把一个数值当作类型 `real`（`float4`）处理：

```
REAL '1.23'  -- 字符串形式
1.23::REAL   -- PostgreSQL（历史）形式
```

这些实际上只是下面将要讨论的一般类型转换记法的特例。

4.1.2.7. 其他类型的常量

任意类型的常量都可以用下列记法之一输入：

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

字符串常量的文本会被传递给名为 *type* 的类型的输入转换例程。其结果是一个指定类型的常量。如果该常量必须具有的类型不存在歧义（例如，它被直接赋给某个表列时），就可以省略显式类型转换，在这种情况下它会被自动强制转换。

字符串常量可以使用常规 SQL 记法或美元引用书写。

也可以用类似函数调用的语法来指定类型强制转换：

```
typename ( 'string' )
```

但并非所有类型名都可以这样使用，详见第 4.2.9 节。

如第 4.2.9 节中所述，`::`、`CAST()` 以及函数调用语法也可以用来指定任意表达式的运行时类型转换。为避免语法歧义，*type* '*string*' 语法只能用来指定简单字面常量的类型。*type* '*string*' 语法的另一个限制是不能用于数组类型；要指定数组常量的类型，请使用 `::` 或 `CAST()`。

`CAST()` 语法符合 SQL。*type* '*string*' 语法是该标准的一般化：SQL 指定这种语法只用于一些数据类型，但是PostgreSQL 允许它用于所有类型。带有 `::` 的语法是 PostgreSQL 的历史用法，就像函数调用语法一样。

4.1.3. 操作符

操作符名称由下列列表中的字符组成，长度最多为 `NAMEDATALEN`-1 个字符（默认是 63）：

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

不过，操作符名称也有一些限制：

- `--` 和 `/*` 不能出现在操作符名称中的任何位置，因为它们会被视为注释的开始。
- 多字符操作符名称不能以 `+` 或 `-` 结尾，除非该名称中还至少包含下列字符中的一个：

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  例如，`@-` 是合法的操作符名称，但 `*-` 不是。这个限制使 PostgreSQL 能够在不要求词元之间必须有空格的情况下解析符合 SQL 的查询。

当使用非 SQL 标准的操作符名时，你通常需要用空格分隔相邻的操作符来避免歧义。例如，如果你定义了一个名为`@`的左一元操作符，你不能写`X*@Y`，你必须写`X* @Y` 来确保 PostgreSQL 把它读作两个操作符名而不是一个。

4.1.4. 特殊字符

某些非字母数字字符具有不同于操作符的特殊含义。关于它们的详细用法，可以在描述相应语法元素的位置找到。本节只是为了提示它们的存在，并概述这些字符的用途。

- 美元符号（`$`）后跟数字时，用来表示函数定义体或预备语句中的位置参数。在其他上下文中，美元符号可以是标识符的一部分，也可以是美元引用字符串常量的一部分。
- 圆括号（`()`）具有它们通常的含义，用来对表达式分组并确定运算优先级。在某些情况中，圆括号被要求作为一个特定 SQL 命令的固定语法的一部分。
- 方括号（`[]`）被用来选择一个数组中的元素。更多关于数组的信息见第 8.15 节。
- 逗号（`,`）被用在某些语法结构中来分割一个列表的元素。
- 分号（`;`）结束一个 SQL 命令。它不能出现在一个命令中间的任何位置，除了在一个字符串常量中或者一个带引号的标识符中。
- 冒号（`:`）被用来从数组中选择“切片”（见第 8.15 节）。在某些 SQL 的“方言”（例如嵌入式 SQL）中，冒号被用来作为变量名的前缀。
- 星号（`*`）在某些上下文中用来表示表行或复合值的所有字段。当它被用作聚合函数的参数时，还有一种特殊含义，即该聚合不需要任何显式参数。
- 句点（`.`）被用在数字常量中，并且被用来分割模式、表和列名。

4.1.5. 注释

注释是一串以双连字符开始并延伸到行尾的字符，例如：

```
-- 这是一条标准 SQL 注释
```

另外，也可以使用 C 风格注释块：

```
/* 多行注释
 * 包含嵌套：/* 嵌套块注释 */
 */
```

这里该注释开始于`/*`并且延伸到匹配出现的`*/`。这些注释块可按照 SQL 标准中指定的方式嵌套，但和 C 中不同。这样我们可以注释掉一大段可能包含注释块的代码。

在进一步进行语法分析之前，注释会从输入流中移除，并实际被替换为空白。

4.1.6. 操作符优先级

表 4.2 展示了 PostgreSQL 中操作符的优先级和结合性。大多数操作符具有相同的优先级，并且是左结合的。操作符的优先级和结合性是固定写在解析器中的。

在组合使用二元和一元操作符时，有时需要添加圆括号。例如：

```
SELECT 5 ! - 6;
```

将被解析为：

```
SELECT 5 ! (- 6);
```

因为解析器并不知道 – 等到知道时已经太迟了 – !被定义为后缀操作符，而不是中缀操作符。要在这种情况下得到所需的行为，必须写成：

```
SELECT (5 !) - 6;
```

这是为可扩展性付出的代价。

表 4.2. 操作符优先级（从高到低）

操作符/元素	结合性	描述
.	左	表/列名分隔符
::	左	PostgreSQL-风格的类型转换
[ ]	左	数组元素选择
+ -	右	一元正号、一元负号
^	左	求幂
* / %	左	乘、除、取模
+ -	左	加、减
（任意其他操作符）	左	所有其他内置以及用户定义的操作符
BETWEEN IN LIKE ILIKE SIMILAR		范围包含、集合成员关系、字符串匹配
< > = <= >= <>		比较操作符
IS ISNULL NOTNULL		IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM 等。
NOT	右	逻辑否定
AND	左	逻辑合取
OR	左	逻辑析取

注意，这些操作符优先级规则也适用于名称与上述内置操作符相同的用户定义操作符。例如，如果你为某种自定义数据类型定义了一个 “+” 操作符，那么无论你的操作符做什么，它都将具有与内置 “+” 操作符相同的优先级。

当一个模式限定的操作符名被用在 OPERATOR 语法中时，如下面的示例：

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

OPERATOR 结构在优先级上会被视为表 4.2 中所示的 “任意其他操作符” 。无论 OPERATOR() 中指定的是哪个具体操作符，这一点都成立。

注意

PostgreSQL 9.5 之前的版本使用的操作符优先级规则略有不同。特别是，<=、>= 和 <> 过去被当作普通操作符；IS 测试过去具有较高的优先级；而 NOT BETWEEN 和相关结构的处理不一致，在某些情况下被认为具有 NOT 而不是 BETWEEN 的优先级。为了更好地符合 SQL 标准，并减少对逻辑等价结构的不一致处理造成的困惑，这些规则得到了修改。在大部分情况下，这些变化不会导致行为变化，或者可能会产生“no such operator”错误，但可以通过增加圆括号解决。不过在一些极端情况下，查询可能在没有报告任何解析错误的情况下改变行为。如果你担心这些变化是否悄悄破坏了某些功能，可以在开启配置参数operator_precedence_warning 的情况下测试应用程序，查看是否记录了任何警告。

4.2. 值表达式

值表达式被用于各种各样的环境中，例如在SELECT 命令的目标列表中，作为INSERT 或 UPDATE 中的新列值，或者作为若干命令中的搜索条件。为了与表表达式（其结果是一张表）区分开来，值表达式的结果有时被称为标量。因此，值表达式也称为标量表达式（甚至简称为表达式）。表达式语法允许使用算术、逻辑、集合以及其他操作，从基本部分计算出值。

一个值表达式是下列之一：

- 一个常量或字面值
- 一个列引用
- 在一个函数定义体或预备语句中的一个位置参数引用
- 一个下标表达式
- 一个字段选择表达式
- 一个操作符调用
- 一个函数调用
- 一个聚合表达式
- 一个窗口函数调用
- 一个类型转换
- 一个排序规则表达式
- 一个标量子查询
- 一个数组构造器
- 一个行构造器
- 另一个圆括号中的值表达式（用于对子表达式分组并覆盖优先级）

在这个列表之外，还有一些结构可以被分类为一个表达式，但是它们不遵循任何一般语法规则。这些通常具有一个函数或操作符的语义并且在第 9 章中的合适位置解释。一个示例是 IS NULL 子句。

我们已经在第 4.1.2 节中讨论过常量。下面的小节会讨论剩下的选项。

4.2.1. 列引用

一个列可以以下面的形式被引用：

correlation.*columnname*

correlation 是一个表的名字（可能带有模式名限定），或者是在 FROM 子句中为某个表定义的别名。如果该列名在当前查询所用的所有表中都是唯一的，则关联名称和分隔用的句点都可以省略（另见第 7 章）。

4.2.2. 位置参数

位置参数引用用来表示一个由 SQL 语句外部提供的值。参数可用于 SQL 函数定义和预备语句中。某些客户端库还支持把数据值与 SQL 命令字符串分开指定，在这种情况下，参数就用来引用这些命令字符串之外的数据值。参数引用的形式是：

$*number*

例如，考虑一个函数dept 的定义：

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

这里 $1 引用函数被调用时第一个函数参数的值。

4.2.3. 下标

如果一个表达式得到了一个数组类型的值，那么可以抽取出该数组值的一个特定元素：

expression[*subscript*]

或者抽取出多个相邻元素（一个“数组切片”）：

expression[*lower_subscript*:*upper_subscript*]

（这里，方括号[ ] 表示其字面意思）。每一个 *subscript* 自身是一个表达式，它将舍入到最接近的整数值。

通常，数组*expression* 必须加上圆括号，但如果要加下标的表达式只是列引用或位置参数，则圆括号可以省略。另外，当原始数组是多维数组时，多个下标可以连写。例如：

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

最后一个示例中的圆括号是必需的。详见第 8.15 节。

4.2.4. 字段选择

如果一个表达式得到一个复合类型（行类型）的值，那么可以抽取该行的指定字段：

expression.*fieldname*

通常，行*expression*必须加上圆括号，但如果要取字段的表达式只是表引用或位置参数，则圆括号可以省略。例如：

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

（因此，限定列引用实际上只是字段选择语法的一种特例。）一种重要的特例是从复合类型的表列中抽取字段：

```
(compositecol).somefield
(mytable.compositecol).somefield
```

这里需要圆括号来显示 `compositecol` 是一个列名而不是一个表名，在第二种情况中则是显示 `mytable` 是一个表名而不是一个模式名。

你可以通过书写 `.*` 来请求一个复合值的所有字段：

```
(compositecol).*
```

这种记法的行为根据上下文会有不同，详见第 8.16.5 节。

4.2.5. 操作符调用

对于一次操作符调用，有三种可能的语法：

expression operator expression（二元中缀操作符）
operator expression（一元前缀操作符）
expression operator（一元后缀操作符）

其中 *operator* 词元遵循第 4.1.3 节的语法规则，或者是关键词 `AND`，`OR` 和 `NOT` 之一，或者是一个如下形式的模式限定操作符名：

OPERATOR(*schema*.*operatorname*)

具体存在哪些操作符以及它们是一元还是二元，取决于系统或用户定义了哪些操作符。第 9 章描述了内置操作符。

4.2.6. 函数调用

函数调用的语法是：函数名（可能带有模式名限定）后跟一组放在圆括号内的参数列表：

function_name ([*expression* [, *expression* ...]])

例如，下面会计算 2 的平方根：

```
sqrt(2)
```

内置函数的列表在第 9 章中。其他函数可以由用户增加。

当在某些用户不信任其他用户的数据库中发出查询时，在编写函数调用时应遵守第 10.3 节中的安全防范措施。

参数也可以选择附带名称。详见第 4.3 节。

> **注意**
>
> 接受单个复合类型参数的函数，可以选择使用字段选择语法来调用；反过来，字段选择也可以写成函数风格。也就是说，`col(table)` 与 `table.col` 这两种记法可以互换。这种行为不是 SQL 标准的一部分，但 PostgreSQL 提供了它，因为这允许使用函数来模拟 “计算字段” 。详见第 8.16.5 节。

4.2.7. 聚合表达式

一个聚合表达式表示在由一个查询选择的行上应用一个聚合函数。一个聚合函数将多个输入减少到一个单一输出值，例如对输入的求和或平均。一个聚合表达式的语法是下列之一：

```
aggregate_name (expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name (ALL expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clause )
 ]
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_
clause ) ]
aggregate_name ( * ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP ( order_by_clause ) [ FILTER (
 WHERE filter_clause ) ]
```

这里 *aggregate_name* 是一个先前定义的聚合（可能带有模式名限定），而 *expression* 是任意值表达式，只要其本身不包含聚合表达式或窗口函数调用即可。可选的 *order_by_clause* 和 *filter_clause* 如下所述。

第一种形式的聚合表达式为每一个输入行调用一次聚合。第二种形式和第一种相同，因为ALL 是默认选项。第三种形式为输入行中表达式的每一个非重复值（对于多个表达式，则是每一个非重复的值组合）调用一次聚合。第四种形式为每一个输入行调用一次聚合，因为没有特定的输入值被指定，它通常只对于`count(*)` 聚合函数有用。最后一种形式被用于有序集聚合函数，其描述如下。

大部分聚合函数忽略值为 null 的输入，因此，一个或多个表达式的结果为 null 的行会被丢弃。除非另有说明，所有内置聚合都可认为具有这种行为。

例如，`count(*)` 得到输入行的总数。`count(f1)` 得到输入行中 `f1` 为非 null 的行数，因为 `count` 忽略空值。而 `count(distinct f1)` 得到 `f1` 的非 null 非重复值的数量。

通常，输入行以未指定的顺序传递给聚合函数。在许多情况下这无关紧要；例如，`min` 无论以什么顺序接收输入都会产生相同的结果。但是，有些聚合函数（例如 `array_agg` 和 `string_agg`）的结果取决于输入行的顺序。使用这类聚合时，可选的 *`order_by_clause`* 可以用来指定所需的顺序。*`order_by_clause`* 的语法与查询级 `ORDER BY` 子句相同，详见第 7.5 节，但其中的表达式只能是表达式，不能是输出列的名称或编号。例如：

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

对于多参数聚合函数，请注意 `ORDER BY` 子句应放在所有聚合参数之后。例如，应写为：

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

而不是：

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- 不正确
```

后者在语法上是合法的，但它表示对单参数聚合函数的调用，并带有两个 `ORDER BY` 键（第二个键是常量，基本无意义）。

如果在 `DISTINCT` 之外还指定了 *`order_by_clause`*，那么所有 `ORDER BY` 表达式都必须与聚合的常规参数相匹配；也就是说，不能根据未包含在 `DISTINCT` 列表中的表达式排序。

> **注意**
>
> 在聚合函数中同时指定 `DISTINCT` 和 `ORDER BY` 的能力是 PostgreSQL 的扩展。

按照到目前为止的描述，在一般用途和统计类聚合里，如果需要对输入行排序，可以把 `ORDER BY` 放在该聚合的常规参数列表中；此时排序是可选的。有一类聚合函数称为有序集聚合，它们必须带有 *`order_by_clause`*，通常是因为这些聚合只有在输入行具有特定顺序时其计算才有意义。有序集聚合的典型例子包括排名和百分位点计算。对于有序集聚合，按照上面最后一种语法，*`order_by_clause`* 要写在 `WITHIN GROUP (...)` 之中。*`order_by_clause`* 中的表达式会像普通聚合参数一样，对每个输入行计算一次，按 *`order_by_clause`* 的要求排序，然后作为输入参数传给聚合函数。（这不同于不带 `WITHIN GROUP` 的 *`order_by_clause`*；在那种情况下，表达式结果不会被视为聚合函数的参数。）如果在 `WITHIN GROUP` 之前还有参数表达式，它们称为直接参数，以区别于列在 *`order_by_clause`* 中的聚合参数。与普通聚合参数不同，直接参数在每次聚合调用中只计算一次，而不是对每个输入行都计算一次。这意味着，只有在这些变量被 `GROUP BY` 分组时，直接参数中才能包含它们；这一限制与这些直接参数根本不在聚合表达式内时是一样的。直接参数通常用于诸如百分位分数之类的值，因为这类值只有在每次聚合计算中作为单一值时才有意义。直接参数列表可以为空；在这种情况下，应写成 `()` 而不是 `(*)`。（实际上 PostgreSQL 两种写法都接受，但只有前者符合 SQL 标准。）

有序集聚合调用的一个示例如下：

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_cont
-----------------
           50489
```

这会得到表 households 中 income 列的第 50 百分位点，也就是中位数。在这里，0.5 是一个直接参数；如果百分位分数在不同行之间变化，那就没有意义了。

如果指定了 FILTER，那么只有那些对 *filter_clause* 求值为真的输入行才会传给聚合函数；其他行会被丢弃。例如：

```
SELECT
    count(*) AS unfiltered,
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

预定义的聚合函数在第 9.20 节中描述。其他聚合函数可以由用户增加。

一个聚合表达式只能出现在 SELECT 命令的结果列表或是 HAVING 子句中。在其他子句（如 WHERE）中禁止使用它，因为那些子句的计算在逻辑上是在聚合的结果被形成之前。

当一个聚合表达式出现在一个子查询中（见第 4.2.11 节和第 9.22 节），聚合通常在该子查询的行上被计算。但是如果该聚合的参数（以及*filter_clause*，如果有）只包含外层变量则有一个例外：该聚合则属于最近的那个外层，并且会在那个查询的行上被计算。该聚合表达式从整体上则是对其所出现于的子查询的一种外层引用，并且在那个子查询的任意一次计算中都作为一个常量。只出现在结果列表或HAVING 子句的限制适用于该聚合所属的查询层次。

4.2.8. 窗口函数调用

一个窗口函数调用表示在一个查询选择的行的某个部分上应用一个聚合类的函数。和非窗口聚合函数调用不同，它并不要求将选中的行分组为单个输出行 – 在查询输出中每一个行仍保持独立。不过，窗口函数能够根据窗口函数调用的分组声明（PARTITION BY 列表）访问属于当前行所在分组中的所有行。一个窗口函数调用的语法是下列之一：

```
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ]
 OVER window_name
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ] OVER (
 window_definition )
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER window_name
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER ( window_definition )
```

其中 *window_definition* 的语法是

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

可选的 *frame_clause* 是下列之一

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

其中 *frame_start* 和 *frame_end* 可以是下面形式中的一种

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

而 *frame_exclusion* 可以是下列之一

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

这里，*expression* 表示任何自身不含有窗口函数调用的值表达式。

window_name 是对定义在查询的 WINDOW 子句中的一个命名窗口声明的引用。还可以使用在 WINDOW 子句中定义命名窗口的相同语法在圆括号内给定一个完整的 *window_definition*，详见SELECT 参考页。值得指出的是，OVER wname 并不严格地等价于 OVER (wname ...)，后者表示复制并修改窗口定义，并且在被引用窗口声明包括一个帧子句时会被拒绝。

PARTITION BY 选项把查询中的行分组成分区，窗口函数会分别处理这些分区。PARTITION BY 的工作方式类似于查询级别的 GROUP BY 子句，不过它的表达式始终只是表达式，不能是输出列名或列编号。如果没有 PARTITION BY，该查询生成的所有行都会被当作单个分区处理。ORDER BY 选项决定窗口函数处理某个分区中的行时所采用的顺序。它的工作方式也类似于查询级别的 ORDER BY 子句，但同样不能使用输出列名或列编号。如果没有 ORDER BY，行将按未指定的顺序处理。

frame_clause 指定构成窗口帧的行集合，它是当前分区的一个子集，供那些作用于帧而不是整个分区的窗口函数使用。帧中的行集合会随着哪一行是当前行而变化。在RANGE、ROWS 或者 GROUPS 模式中可以指定帧，在每一种情况下，帧的范围都是从*frame_start* 到 *frame_end*。如果 *frame_end* 被省略，则末尾默认为CURRENT ROW。

UNBOUNDED PRECEDING 的一个 *frame_start* 表示该帧开始于分区的第一行，类似地 UNBOUNDED FOLLOWING 的一个 *frame_end* 表示该帧结束于分区的最后一行。

在 RANGE 或 GROUPS 模式中，CURRENT ROW 的一个 *frame_start* 表示帧开始于当前行的第一个同等行（被窗口的 ORDER BY 子句排序为与当前行等效的行），而CURRENT ROW 的一个 *frame_end* 表示帧结束于当前行的最后一个同等行。在 ROWS 模式中，CURRENT ROW 就表示当前行。

在 *offset* PRECEDING 以及 *offset* FOLLOWING 帧选项中，*offset* 必须是一个不包含任何变量、聚合函数或者窗口函数的表达式。*offset* 的含义取决于帧模式：

- 在 ROWS 模式中，*offset* 必须得到一个非空、非负的整数，并且该选项表示帧开始或结束于当前行之前或者之后指定数量的行。
- 在 GROUPS 模式中，*offset* 也必须得到一个非空、非负的整数，并且该选项表示帧开始或结束于当前行的同等行组之前或者之后指定数量的同等行组，这里同等行组是在ORDER BY 顺序中等效的行集合（要使用 GROUPS 模式，在窗口定义中就必须有一个ORDER BY 子句）。
- 在 RANGE 模式中，这些选项要求ORDER BY 子句正好指定一列。*offset* 指定当前行中那一列的值与它在该帧中前面或后面的行中的列值的最大差值。*offset* 表达式的数据类型会随着排序列的数据类型而变化。对于数字的排序列，它通常是与排序列相同的类型，但对于日期时间排序列它是一个interval。例如，如果排序列是类型date 或者 timestamp，我们可以写 RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWING。*offset* 仍然要求是非空且非负，不过“非负”的含义取决于它的数据类型。

在任何一种情况下，到帧边界的距离都受限于到分区边界的距离，因此对于靠近分区两端的行来说，帧可能会包含比较少的行。

注意在 ROWS 以及 GROUPS 模式中，0 PRECEDING 和 0 FOLLOWING 与 CURRENT ROW 等效。通常在 RANGE 模式中，这个结论也成立（只要有一种合适的、与数据类型相关的“零”的含义）。

frame_exclusion 选项允许当前行周围的行被排除在帧之外，即便根据帧的开始和结束选项应该把它们包括在帧中。EXCLUDE CURRENT ROW 会把当前行排除在帧之外。EXCLUDE GROUP 会把当前行以及它在顺序上的同等行都排除在帧之外。EXCLUDE TIES 把当前行的任何同等行都从帧中排除，但不排除当前行本身。EXCLUDE NO OTHERS 只是明确地指定不排除当前行或其同等行的这种默认行为。

默认的帧选项是 RANGE UNBOUNDED PRECEDING，它和RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW 相同。如果使用 ORDER BY，这会把该帧设置为从分区开始一直到当前行的最后一个ORDER BY 同等行的所有行。如果不使用 ORDER BY，就意味着分区中所有的行都被包括在窗口帧中，因为所有行都成为了当前行的同等行。

限制是 *frame_start* 不能是 UNBOUNDED FOLLOWING、*frame_end* 不能是 UNBOUNDED PRECEDING，并且在上述*frame_start* 和 *frame_end* 选项的列表中 *frame_end* 选择不能早于 *frame_start* 选择出现 – 例如不允许 RANGE BETWEEN CURRENT ROW AND *offset* PRECEDING，但允许ROWS BETWEEN 7 PRECEDING AND 8 PRECEDING，虽然它不会选择任何行。

如果指定了 FILTER，那么只有对*filter_clause* 计算为真的输入行会被交给该窗口函数，其他行会被丢弃。只有是聚合的窗口函数才接受FILTER。

内置的窗口函数在表 9.57 中介绍。用户可以加入其他窗口函数。此外，任何内置的或者用户定义的通用聚合或者统计性聚合都可以被用作窗口函数（有序集和假想集聚合当前不能被用作窗口函数）。

使用* 的语法被用来把无参数聚合函数当作窗口函数调用，例如count(*) OVER (PARTITION BY x ORDER BY y)。星号（*）通常不被用于专用窗口函数。专用窗口函数不允许在函数参数列表中使用 DISTINCT 或 ORDER BY。

只有在 SELECT 列表和查询的 ORDER BY 子句中才允许窗口函数调用。

更多关于窗口函数的信息可以在第 3.5 节、第 9.21 节以及第 7.2.5 节中找到。

4.2.9. 类型转换

一个类型转换指定从一种数据类型到另一种数据类型的转换。PostgreSQL 接受两种等价的类型转换语法：

```
CAST ( expression AS type )
expression::type
```

CAST 语法遵从 SQL，而用::的语法是 PostgreSQL 的历史用法。

当类型转换被应用于一种已知类型的值表达式时，它表示一次运行时类型转换。只有在已经定义了合适的类型转换操作时，该类型转换才会成功。注意，这与常量上的类型转换（如第 4.1.2.7 节中所示）略有不同。应用于未修饰字符串字面量的类型转换，表示为一个字面常量值赋予初始类型，因此它对任意类型都能成功（前提是该字符串字面量的内容符合该数据类型的输入语法）。

如果一个值表达式必须产生的类型没有歧义（例如当它被赋给一个表列时），通常可以省略显式类型转换，在这种情况下系统会自动应用类型转换。但是，自动类型转换只会对那些在系统目录中被标记为“可以隐式应用”的类型转换执行。其他类型转换必须使用显式类型转换语法来调用。施加这一限制的目的是防止出人意料的转换被悄无声息地应用。

还可以用类似函数调用的语法来指定类型转换：

```
typename ( expression )
```

不过，这只对那些名称本身也可作为函数名使用的类型有效。例如，double precision 不能用这种方式，但等效的 float8 可以。另外，由于语法冲突，名称 interval、time 和 timestamp 只有在使用双引号引用时才能采用这种写法。因此，函数风格的类型转换语法会带来不一致性，通常应尽量避免。

> **注意**
>
> 函数风格的语法事实上只是一次函数调用。当两种标准类型转换语法之一被用来做一次运行时转换时，它将在内部调用一个已注册的函数来执行该转换。按照惯例，这些转换函数具有和它们的输出类型相同的名字，并且因此“函数风格的语法”无非是对底层转换函数的一次直接调用。显然，一个可移植的应用不应当依赖于它。详见 CREATE CAST。

4.2.10. 排序规则表达式

COLLATE 子句会覆盖表达式的排序规则。它附加在所作用的表达式之后：

```
expr COLLATE collation
```

这里 *collation* 可以是一个带模式名限定的标识符。COLLATE 子句比操作符绑定得更紧，必要时可以使用圆括号。

如果没有显式指定排序规则，数据库系统会从表达式所涉及的列中得到一个排序规则，如果该表达式没有涉及列，则会默认采用数据库的默认排序规则。

COLLATE 子句的两种常见用途是覆盖 ORDER BY 子句中的排序顺序，例如：

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

以及覆盖结果受区域设置影响的函数或操作符调用的排序规则，例如：

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

注意，在后一种情况中，COLLATE 子句是附加到我们希望影响的那个操作符的某个输入参数上的。把 COLLATE 子句附加到该操作符或函数调用的哪个参数上并不重要，因为操作符或函数实际采用的排序规则是综合所有参数得出的，而显式的 COLLATE 子句会覆盖其他所有参数的排序规则。（不过，若把不匹配的 COLLATE 子句附加到多个参数上，则会报错。详见第 23.2 节。）因此，这会得到与前一个示例相同的结果：

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

但是这是一个错误：

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

因为它尝试把一个排序规则应用到> 操作符的结果，而它的数据类型是不支持排序规则的数据类型boolean。

4.2.11. 标量子查询

标量子查询是一个放在圆括号中的普通 SELECT 查询，它恰好返回一行一列（关于如何编写查询，见第 7 章）。该 SELECT 查询会被执行，其返回的单个值会用在外围值表达式中。把一个返回多于一行或多于一列的查询当作标量子查询来用是错误的。（但如果在某次具体执行中该子查询没有返回任何行，则不算错误；其标量结果会被视为空值。）该子查询可以引用外围查询中的变量，这些变量在该子查询的每一次求值期间都会作为常量。有关其他涉及子查询的表达式，另见第 9.22 节。

例如，下列语句会寻找每个州中最大的城市人口：

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
    FROM states;
```

4.2.12. 数组构造器

数组构造器是一个表达式，它使用其成员元素的值来构造一个数组值。简单的数组构造器由关键字 ARRAY、左方括号 [、一个用于指定数组元素值的表达式列表（以逗号分隔）以及最后的右方括号] 组成。例如：

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

默认情况下，数组元素类型是成员表达式的公共类型，其确定规则与 UNION 或 CASE 结构相同（见第 10.5 节）。你可以通过把数组构造器显式转换为所需类型来覆盖这一点，例如：

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

这与把每个表达式分别转换为数组元素类型具有相同的效果。关于类型转换的更多信息，见第 4.2.9 节。

多维数组值可以通过嵌套数组构造器来构造。在内部构造器中，关键字 ARRAY 可以省略。例如，下面两种写法产生相同的结果：

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

由于多维数组必须是长方形的，因此同一级别的内部构造器必须生成维度完全相同的子数组。施加到外层 ARRAY 构造器上的任何类型转换，都会自动传播到所有内层构造器。

多维数组构造器中的元素可以是任何能够生成适当类型数组的表达式，而不仅仅是子 ARRAY 构造。例如：

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
-----------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

你也可以构造空数组，但由于不可能存在没有类型的数组，因此必须把空数组显式转换为所需类型。例如：

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

也可以根据子查询的结果构造数组。在这种形式中，数组构造器写成关键字 ARRAY 后跟一个圆括号括起来的（而不是方括号括起来的）子查询。例如：

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                                  array
------------------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412,2413}
(1 row)

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
             array
----------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

子查询必须返回单列。如果子查询的输出列是非数组类型，那么生成的一维数组会为子查询结果中的每一行包含一个元素，其元素类型与子查询输出列的类型相匹配。如果子查询的输出列是数组类型，那么结果将是相同类型但高一维的数组；在这种情况下，所有子查询行都必须产生维度相同的数组，否则结果就不会是长方形。

用 ARRAY 构造出来的数组值，其下标总是从 1 开始。关于数组的更多信息，见第 8.15 节。

4.2.13. 行构造器

行构造器是一种表达式，它使用成员字段的值来构造一个行值（也称为复合值）。行构造器由关键字 `ROW`、左圆括号、零个或多个用于指定行各字段值的表达式（以逗号分隔），以及最后的右圆括号组成。例如：

```
SELECT ROW(1,2.5,'this is a test');
```

当列表中有多个表达式时，关键字 `ROW` 是可选的。

一个行构造器可以包括语法 *`rowvalue`*`.*`，它将被扩展为该行值的元素的一个列表，就像在 `SELECT` 列表的顶层（见第 8.16.5 节）中使用 `.*` 时发生的事情一样。例如，如果表`t` 有列 `f1` 和 `f2`，那么这些是相同的：

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **注意**
>
> 在 PostgreSQL 8.2 以前，`.*` 语法不会在行构造器中展开，因此写成 `ROW(t.*, 42)` 会创建一个具有两个字段的行，其中第一个字段是另一个行值。新的行为通常更有用。如果你需要旧版本那种嵌套行值的行为，写内层行值时不要使用 `.*`，例如 `ROW(t, 42)`。

默认情况下，ROW 表达式创建的值具有匿名记录类型。如有需要，可以把它转换为具名复合类型，也就是某个表的行类型，或者使用 CREATE TYPE AS 创建的复合类型。为避免歧义，可能需要显式类型转换。例如：

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- 只有一个 getf1()，因此不需要类型转换
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- 现在需要用类型转换指明要调用哪个函数：
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

行构造器可以用来构造复合值，以存储在复合类型的表列中，或者传递给接受复合参数的函数。此外，还可以比较两个行值，或者用 IS NULL 或 IS NOT NULL 来测试一个行，例如：

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.*) IS NULL FROM table;  -- 检测所有字段均为 null 的行
```

更多详情见第 9.23 节。行构造器也可以与子查询结合使用，详见第 9.22 节。

4.2.14. 表达式计算规则

子表达式的计算顺序是未定义的。特别是，操作符或函数的输入不一定会按照从左到右或任何其他固定顺序计算。

此外，如果一个表达式的结果可以通过只计算其一部分来决定，那么其他子表达式可能完全不需要被计算。例如，如果我们写：

```
SELECT true OR somefunc();
```

那么 somefunc() 将（可能）完全不被调用。如果我们写成下面这样也是一样：

```
SELECT somefunc() OR true;
```

注意这和一些编程语言中布尔操作符从左至右的“短路”不同。

因此，在复杂表达式中使用带有副作用的函数是不明智的。在WHERE 和 HAVING 子句中依赖副作用或计算顺序尤其危险，因为在建立一个执行计划时这些子句会被广泛地重新处理。这些子句中布尔表达式（AND/OR/NOT 的组合）可能会以布尔代数定律所允许的任何方式被重组。

当有必要强制计算顺序时，可以使用一个CASE 结构（见第 9.17 节）。例如，在一个WHERE 子句中使用下面的方法尝试避免除零是不可靠的：

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

但是这是安全的：

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

一个以这种风格使用的 CASE 结构将使得优化尝试失败，因此只有必要时才这样做（在这个特别的示例中，最好通过写y > 1.5*x 来回避这个问题）。

不过，CASE 并不是这类问题的万灵药。上述技术的一个限制是，它无法阻止常量子表达式被提前计算。如第 38.7 节中所述，当查询被规划而不是被执行时，被标记成 IMMUTABLE 的函数和操作符可以被计算。因此

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

很可能会导致一次除零失败，因为规划器尝试简化常量子表达式。即便是表中的每一行都有 x > 0（这样运行时永远不会进入到 ELSE 分支）也是这样。

虽然这个例子看起来有些可笑，但在函数内部执行的查询中，也可能出现一些看上去并未明显涉及常量的相关情况，因为函数参数值和局部变量值为了规划目的，可能会作为常量插入到查询中。例如，在 PL/pgSQL 函数中，使用 IF-THEN-ELSE 语句来保护一项有风险的计算，要比仅仅把它嵌套进一个 CASE 表达式安全得多。

另一个同类型的限制是，一个CASE 无法阻止其所包含的聚合表达式的计算，因为在考虑SELECT 列表或 HAVING 子句中的其他表达式之前，会先计算聚合表达式。例如，下面的查询可能会导致除零错误，尽管看起来似乎已经对这种情况做了防护：

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
    FROM departments;
```

min() 和 avg() 聚合会在所有输入行上同时计算，因此如果任何行有employees 等于零，在有机会测试 min() 的结果之前，就会发生除零错误。取而代之的是，可以使用一个WHERE 或 FILTER 子句来首先阻止有问题的输入行到达一个聚合函数。

4.3. 调用函数

PostgreSQL 允许对带命名参数的函数使用位置记法或命名记法来调用。命名记法对于拥有大量参数的函数尤其有用，因为它能让形参与实参之间的对应关系更明确、更可靠。在位置记法中，函数调用中的参数值按照它们在函数声明中定义的顺序书写；在命名记法中，实参与函数参数按名称匹配，因此可以按任意顺序书写。对于每一种记法，还要考虑函数参数类型的影响，这一点在第 10.3 节中有说明。

在任意一种记法中，在函数声明中给出了默认值的参数都可以在调用中省略。不过这在命名记法中特别有用，因为任意组合的参数都可以被省略；而在位置记法中，参数只能从右向左省略。

PostgreSQL 也支持混合记法，它结合了位置记法和命名记法。在这种情况下，位置参数先写，命名参数写在后面。

下面的示例将展示这三种记法的用法，所用函数定义如下：

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

函数 concat_lower_or_upper 有两个必需参数 a 和 b。此外还有一个可选参数 uppercase，其默认值为 false。a 和 b 的输入将被连接起来，并根据 uppercase 参数的值转换成大写或小写。该函数定义的其余细节在这里并不重要（详见第 38 章）。

4.3.1. 使用位置记法

位置记法是在 PostgreSQL 中向函数传递参数的传统机制。例如：

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

所有参数都按顺序给出。由于 `uppercase` 被指定为 `true`，因此结果是大写。另一个例子是：

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

这里省略了 `uppercase` 参数，因此它接收默认值 `false`，结果是小写输出。在位置记法中，只要参数具有默认值，就可以从右向左依次省略。

4.3.2. 使用命名记法

在命名记法中，每个参数名都使用 `=>` 与其参数表达式分隔。例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

同样，`uppercase` 参数被省略，因此它被隐式设置为 `false`。使用命名记法的一个优点是，参数可以按任意顺序指定，例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

为了向后兼容，也支持基于 `:=` 的旧语法：

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

4.3.3. 使用混合记法

混合记法结合了位置记法和命名记法。不过，如前所述，命名参数不能出现在位置参数之前。例如：

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

在上面的查询中，参数 `a` 和 `b` 采用位置方式指定，而 `uppercase` 采用命名方式指定。在这个示例中，这样做除了起到一定说明作用外并没有太多额外价值。但对于拥有大量带默认值参数的更复杂函数，命名记法或混合记法可以大幅减少书写量，并降低出错机会。

> **注意**
>
> 命名调用记法和混合调用记法目前不能用于调用聚合函数（但当聚合函数被用作窗口函数时，它们是可以使用的）。

第 5 章 数据定义

本章介绍如何创建用于保存数据的数据库结构。在关系数据库中，原始数据存储在表中，因此本章的大部分内容都在说明如何创建和修改表，以及有哪些特性可用于控制表中存储的数据。随后，我们将讨论如何把表组织到模式中，以及如何向表分配权限。最后，我们会简要介绍一些同样会影响数据存储的特性，例如继承、表分区、视图、函数和触发器。

5.1. 表基础

关系数据库中的表很像纸面上的表格：它由行和列组成。列的数量和顺序是固定的，每一列都有名称。行的数量则是可变的，它反映了某一时刻存储了多少数据。SQL 不对表中行的顺序作任何保证。读取表时，除非显式要求排序，否则行会以未指定的顺序出现，参见第 7 章。此外，SQL 不会为行分配唯一标识符，因此一个表中可能出现多行完全相同的记录。这是 SQL 所依据的数学模型带来的结果，但通常并不理想。本章稍后将看到如何处理这个问题。

每一列都有一种数据类型。数据类型限制了可以赋给该列的可能值集合，并为列中存储的数据赋予语义，从而使其能够用于计算。例如，声明为数值类型的列不能接受任意文本字符串，存储在这类列中的数据可以参与数学运算。相比之下，声明为字符串类型的列几乎能接受任何种类的数据，但并不适合用于数学计算，尽管仍可进行字符串拼接等其他操作。

PostgreSQL 提供了相当丰富的一组内置数据类型，适用于许多应用场景。用户也可以定义自己的数据类型。大多数内置数据类型的名称和语义都很直观，因此我们把详细说明留到第 8 章。一些常用的数据类型包括：表示整数的 `integer`、表示可能带小数部分数值的`numeric`、表示字符串的`text`、表示日期的`date`、表示一天中时间值的`time`，以及同时包含日期和时间的`timestamp`。

要创建一个表，可使用名副其实的CREATE TABLE 命令。在该命令中，至少要指定新表的名称、各列的名称以及每列的数据类型。例如：

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

这会创建一个名为 `my_first_table` 的表，它有两列。第一列名为`first_column`，数据类型为`text`；第二列名为 `second_column`，类型为`integer`。表名和列名遵循第 4.1.1 节中解释的标识符语法。类型名通常也是标识符，但也有少数例外。注意，列列表以逗号分隔，并用圆括号括起来。

当然，前面的例子相当刻意。通常，你会为表和列起能体现其所存储数据含义的名称。下面看一个更实际的例子：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

（`numeric` 类型可以存储小数部分，这对金额很常见。）

> **提示**
>
> 在创建许多相互关联的表时，最好为表和列选择一致的命名模式。例如，表名可以使用单数名词或复数名词，两种做法都各有支持者。

一个表能包含的列数是有限的。根据列类型不同，这个上限在 250 到 1600 之间。不过，定义一个接近这个数量的表极不常见，而且其设计往往值得商榷。

如果不再需要某个表，可以使用DROP TABLE 命令删除它。例如：

```
DROP TABLE my_first_table;
DROP TABLE products;
```

尝试删除不存在的表会报错。不过，在 SQL 脚本文件中，人们常常在创建每个表之前无条件地尝试删除它，并忽略可能出现的错误消息，这样脚本无论表是否存在都能工作。（如果愿意，也可以使用`DROP TABLE IF EXISTS` 变体来避免错误消息，但这不是标准 SQL。）

如果我们需要修改一个已经存在的表，请参考本章稍后的第 5.5 节。

利用到目前为止所讨论的工具，我们已经可以创建功能完备的表。本章其余内容关注如何向表定义添加特性，以确保数据完整性、安全性或使用便利。如果你现在就急着向表中填充数据，可以先跳到第 6 章，稍后再回来阅读本章其余内容。

5.2. 默认值

列可以被赋予默认值。当创建新行时，如果没有为某些列指定值，这些列就会填入各自的默认值。数据操作命令也可以显式要求把某列设为其默认值，而不必知道默认值究竟是什么。（数据操作命令详见第 6 章。）

如果没有显式指定默认值，则默认值是空值。这通常是合理的，因为空值可以被视为表示未知数据。

在表定义中，默认值写在列的数据类型之后。例如：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

默认值也可以是表达式。每次需要插入默认值时都会对其求值（不是在创建表时）。一个常见例子是让 timestamp 列的默认值为 CURRENT_TIMESTAMP，这样它会被设为插入该行时的时间。另一个常见例子是为每一行生成一个“序列号”。在 PostgreSQL 中，这通常可以这样实现：

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'),
    ...
);
```

其中 nextval() 函数从序列对象（见第 9.16 节）中依次取值。这种写法非常常见，因此还有一种专门的简写形式：

```
CREATE TABLE products (
    product_no SERIAL,
    ...
);
```

SERIAL 简写会在第 8.1.4 节进一步讨论。

5.3. 约束

数据类型是一种限制能够存储在表中的数据类别的方法。但是对于很多应用来说，它们提供的约束太粗糙。例如，一个包含产品价格的列应该只接受正值。但是没有任何一种标准数据类型只接受正值。另一个问题是我们可能需要根据其他列或行来约束一个列中的数据。例如，在一个包含产品信息的表中，对于每个产品编号应该只有一行。

为此，SQL 允许我们在列和表上定义约束。约束让我们能够按照需要控制表中的数据。如果用户试图在列中存储违反约束的数据，就会报错。即使该值来自默认值定义，这条规则也同样适用。

5.3.1. 检查约束

检查约束是最通用的约束类型。它允许我们指定某一列中的值必须满足一个布尔（真值）表达式。例如，要要求产品价格为正值，可以使用：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

如你所见，约束定义和默认值定义一样都写在数据类型之后。默认值和约束的先后顺序没有影响。检查约束由关键字CHECK 以及其后放在圆括号中的表达式组成。检查约束表达式应当涉及被约束的列，否则这个约束就没有太大意义。

我们也可以为约束单独指定一个名称。这样可以让错误消息更清晰，也便于在需要修改约束时引用它。语法如下：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

要指定一个命名约束，在约束名标识符前写关键字CONSTRAINT，再在其后写约束定义即可。（如果没有用这种方式指定约束名，系统会为你选择一个。）

一个检查约束也可以引用多个列。例如我们存储一个普通价格和一个打折后的价格，而我们希望保证打折后的价格低于普通价格：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

前两个约束看起来应该很熟悉。第三个则使用了一种新语法。它并没有依附在一个特定的列，而是作为一个独立的项出现在逗号分隔的列列表中。列定义和这种约束定义可以以混合的顺序出现在列表中。

我们将前两个约束称为列约束，而第三个约束为表约束，因为它独立于任何一个列定义。列约束也可以写成表约束，但反过来不一定可行，因为列约束应当只引用它所依附的那一列（PostgreSQL 并不强制要求这个规则，但是如果我们希望表定义能够在其他数据库系统中工作，那就应该遵循它）。上述示例也可以写成：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

甚至是：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

这只是口味的问题。

表约束也可以用与列约束相同的方法来指定名称：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

需要注意的是，当检查表达式的值为真或空值时，检查约束就被视为满足。由于当任一操作数为空时，大多数表达式都会计算为空值，所以检查约束不会阻止受约束列中出现空值。要确保某列不包含空值，可以使用下一节介绍的非空约束。

> **注意**
>
> PostgreSQL 不支持引用除了正在检查的新行或更新行之外的表数据的 CHECK 约束。虽然违反此规则的 CHECK 约束在简单测试中可能有效，但无法保证数据库不会达到约束条件为假的状态（由于其他行的后续更改）。这将导致数据库转储和恢复失败。即使完整的数据库状态与约束一致，恢复也可能失败，因为行未按满足约束的顺序加载。如果可能的话，使用UNIQUE、EXCLUDE 或 FOREIGN KEY 约束来表示跨行和跨表的限制。

如果你需要的是在插入行时针对其他行做一次性检查，而不是持续维护一致性保证，可以使用自定义触发器来实现。（这种方法避免了转储/恢复问题，因为pg_dump 在恢复数据之后才重新安装触发器，因此在转储/恢复期间不会强制执行检查。）

注意

PostgreSQL 假定 CHECK 约束的条件是不可变的，也就是说，对于同一输入行它们始终会给出相同的结果。正是这个假设，才使得只需要在插入或更新行时检查CHECK 约束，而不必在其他时间检查。（上面关于不引用其他表数据的警告实际上是此限制的特殊情况。）

一种常见的破坏这种假设的方式，是在CHECK 表达式中引用用户定义函数，然后改变该函数的行为。PostgreSQL 不会禁止这样做，但它不会注意到表中是否有行现在违反了CHECK 约束。这将导致后续的数据库转储和恢复操作失败。处理这种变化的推荐方法是删除约束（使用 ALTER TABLE），调整函数定义，然后重新添加约束，从而重新检查所有表行。

5.3.2. 非空约束

非空约束只是指定某列不能取空值。语法示例如下：

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

非空约束总是写成列约束。从功能上看，非空约束等价于创建检查约束CHECK (*column_name* IS NOT NULL)，但在PostgreSQL 中创建显式的非空约束效率更高。缺点是无法为以这种方式创建的非空约束显式指定名称。

当然，一列可以有多个约束。只需将约束逐个写出：

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

顺序并不重要，也不一定决定检查约束的顺序。

该 NOT NULL 约束有一个反面形式：NULL 约束。这并不表示该列必须为空值，那显然毫无用处。它只是显式选择列可以为空的默认行为。NULL 约束不属于 SQL 标准，因此不应在需要可移植性的应用中使用。（它之所以被加入PostgreSQL，只是为了兼容某些其他数据库系统。）不过，有些用户喜欢它，因为它让在脚本文件中切换该约束变得比较容易。例如，可以从下面的定义开始：

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

然后在需要的地方插入 NOT 关键字。

> **提示**
>
> 在大多数数据库设计中，多数列都应标记为非空。

5.3.3. 唯一约束

唯一约束保证某一列或某一组列中保存的数据在整个表的所有行之间都是唯一的。写成列约束时的语法是：

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

写成表约束时则是：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

要为一组列定义一个唯一约束，把它写作一个表级约束，列名用逗号分隔：

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

这指定这些列的组合值在整个表的范围内是唯一的，但其中任意一列的值并不需要是（一般也不是）唯一的。

我们可以按通常的方式为唯一约束指定名称：

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

增加一个唯一约束会在约束中列出的列或列组上自动创建一个唯一 B-树索引。只覆盖某些行的唯一性限制不能写成唯一约束，但可以通过创建唯一的部分索引来强制这种限制。

通常情况下，如果表中有多行，且约束所包含的所有列的值都相等，就违反了唯一约束。不过，在这种比较中，两个空值从不被视为相等。这意味着即使存在唯一约束，也可以存储重复行，只要其中至少一个受约束列含有空值。此行为符合 SQL 标准，但我们听说其他 SQL 数据库可能不遵循此规则。因此，开发需要可移植性的应用时要小心。

5.3.4. 主键

一个主键约束表示可以用作表中行的唯一标识符的一个列或者一组列。这要求那些值都是唯一的并且非空。因此，下面的两个表定义接受相同的数据：

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

主键也可以包含多于一个列，其语法和唯一约束相似：

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

增加一个主键会自动在主键中列出的列或列组上创建一个唯一 B-树索引，并强制这些列被标记为 NOT NULL。

一个表最多只能有一个主键（可以有任意数量的唯一和非空约束，它们可以达到和主键几乎一样的功能，但只能有一个被标识为主键）。关系数据库理论要求每一个表都要有一个主键。PostgreSQL 并未强制要求这一点，但通常最好遵循它。

主键对于文档和客户端应用都是有用的。例如，一个允许修改行值的 GUI 应用可能需要知道一个表的主键，以便能唯一地标识行。如果定义了主键，数据库系统也有多种方法来利用主键。例如，主键定义了外键要引用的默认目标列。

5.3.5. 外键

一个外键约束指定一列（或一组列）中的值必须与另一个表中某一行的对应值匹配。我们说这维持了两个关联表之间的引用完整性。

例如我们有一个使用过多次的产品表：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

让我们假设我们还有一个存储这些产品订单的表。我们希望保证订单表中只包含真正存在的产品的订单。因此我们在订单表中定义一个引用产品表的外键约束：

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

现在就不可能创建包含不存在于产品表中的 `product_no` 值（非空）的订单。

我们说在这种情况下，订单表是引用表而产品表是被引用表。相应地，也有引用和被引用列的说法。

我们也可以把上述命令简写为：

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

因为如果缺少列的列表，则被引用表的主键将被用作被引用列。

你可以按常规方式为外键约束指定自己的名称。

一个外键也可以约束和引用一组列。照例，它需要被写成表约束的形式。下面是一个示例：

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

当然，被约束列的数量和类型应该匹配被引用列的数量和类型。

有时，让外键约束的“其它表”是同一个表会很有用；这称为自引用外键。例如，如果希望表中的行代表树结构的节点，可以写成：

```
CREATE TABLE tree (
    node_id integer PRIMARY KEY,
    parent_id integer REFERENCES tree,
    name text,
    ...
);
```

顶层节点的 `parent_id` 为 NULL，而非 NULL 的`parent_id` 条目则受到约束，必须引用该表中的有效行。

一个表可以有多个外键约束。这可用于实现表之间的多对多关系。例如，我们已经有了关于产品和订单的表，但现在希望一个订单可以包含多种产品（上面的结构并不允许这样做）。我们可以使用下面这种表结构：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

注意在最后一个表中主键和外键之间有重叠。

我们知道外键不允许创建与任何产品都不相关的订单。但如果一个产品在一个引用它的订单创建之后被移除会发生什么？SQL 允许我们处理这种情况。直观上，我们有几种选项：

- 不允许删除一个被引用的产品
- 同时也删除引用产品的订单
- 其他？

为了说明这些，让我们在上面的多对多关系示例中实现下面的策略：当某人希望移除一个仍然被一个订单引用（通过order_items）的产品时，我们阻止它。如果某人移除一个订单，订单项也同时被移除：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

限制删除和级联删除是最常见的两种选项。RESTRICT 阻止删除被引用的行。NO ACTION 表示检查约束时若仍存在引用行，就会报错；未指定任何选项时，这是默认行为。（这两种选项的根本区别在于，NO ACTION 允许将检查延迟到事务稍后进行，而RESTRICT 不允许。）CASCADE 指定当被引用行被删除时，引用它的行也应自动删除。另有两个选项：SET NULL 和SET DEFAULT。它们会在被引用行删除时，分别把引用行中的引用列设为空值或其默认值。注意，这并不能让你免于遵守其他约束。例如，如果某个动作指定了SET DEFAULT，但默认值本身不满足外键约束，则该操作仍会失败。

与 ON DELETE 类似，还有ON UPDATE，当被引用列发生变化（更新）时就会触发。可用动作相同。在这种情况下，CASCADE 表示应把被引用列更新后的值复制到引用行中。

正常情况下，如果引用行的任意一个引用列为空，它就不需要满足外键约束。如果在外键声明中加入MATCH FULL，引用行只有在所有引用列都为空时才不需要满足约束（因此空值和非空值混合的情况必定不满足MATCH FULL 约束）。如果不希望引用行能够避开外键约束，应将引用列声明为NOT NULL。

外键必须引用作为主键或组成唯一约束的列。这意味着被引用列总是具有索引（即主键或唯一约束的底层索引），因此可以高效检查引用行是否存在匹配项。由于从被引用表中DELETE 行或 UPDATE 被引用列时，需要扫描引用表以查找匹配旧值的行，因此通常也建议为引用列建立索引。由于这并不总是必需，而且索引方式也有很多种，因此声明外键约束时不会自动在引用列上创建索引。

更多关于更新和删除数据的信息请见第 6 章。外键约束的语法描述请参考 CREATE TABLE。

5.3.6. 排他约束

排他约束保证：对任意两行，若对指定列或表达式使用指定操作符进行比较，则这些操作符比较中至少有一个会返回假或空值。语法如下：

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

详见 `CREATE TABLE ... CONSTRAINT ... EXCLUDE`。

增加一个排他约束会自动创建约束声明中指定类型的索引。

5.4. 系统列

每个表都有几个由系统隐式定义的系统列。因此，这些名称不能用作用户定义列的名称。（注意，这些限制与名称是否为关键字无关；给名称加引号也不能绕过这些限制。）你其实不必关心这些列的细节，只需要知道它们存在即可。

`oid`

行的对象标识符（对象 ID）。只有使用 `WITH OIDS` 创建表，或创建时设置了 default_with_oids 配置变量，才存在此列。该列的类型是`oid`（与列同名）；关于该类型的更多信息，参见第 8.19 节。

`tableoid`

包含该行的表的 OID。对于从继承层次（见第 5.9 节）中查询数据的场景，这一列尤其方便，因为如果没有它，就很难判断一行究竟来自哪个具体表。`tableoid` 可以与 `pg_class` 的 `oid` 列连接，以取得表名。

`xmin`

插入该行版本的事务 ID。（行版本是某一行的一个具体状态；对同一逻辑行的每次更新都会创建一个新的行版本。）

`cmin`

插入事务中的命令标识符（从 0 开始）。

`xmax`

删除事务的标识（事务 ID）；对于未删除的行版本则为 0。对于一个可见的行版本，该列也可能是非零值。这通常表示删除事务尚未提交，或者一次删除尝试被回滚了。

`cmax`

删除事务中的命令标识符，或者为 0。

`ctid`

行版本在其表中的物理位置。注意尽管 `ctid` 可以被用来非常快速地定位行版本，但是一个行的`ctid` 会在被更新或者被 `VACUUM FULL` 移动时改变。因此，`ctid` 不适合用作长期行标识符。应使用 OID，或者更好地使用用户定义的序列号，来标识逻辑行。

OID 是 32 位的量，由整个集簇共用的单个计数器分配。在大型或长期运行的数据库中，该计数器可能发生回卷。因此，除非采取了确保唯一性的措施，否则不应假定 OID 唯一。如果需要标识表中的行，强烈建议使用序列生成器。不过，只要采取一些额外的预防措施，也可以使用 OID：

- 对于使用 OID 标识行的每个表，都应在其 OID 列上创建唯一约束。有这样的唯一约束（或唯一索引）时，系统会确保不生成与已有行相同的 OID。（当然，这只有在表的行数小于 2^{32}（40 亿）时才有可能；实际上，表的大小最好远小于此值，否则性能可能受影响。）
- 绝不能假定 OID 在不同表之间也唯一；如果需要数据库范围的标识符，应将`tableoid` 与行 OID 组合使用。
- 当然，相关表必须以`WITH OIDS` 创建。从 PostgreSQL 8.1 开始，默认值是 `WITHOUT OIDS`。

事务 ID 也是 32 位的量。在一个长期运行的数据库中，事务 ID 可能会回卷。只要采取适当的维护措施，这并不是致命问题，详见第 24 章。不过，从长期来看（超过十亿个事务）依赖事务 ID 的唯一性是不明智的。

命令标识符也是 32 位的量。这为单个事务中的 SQL 命令数设置了一个硬上限：2^{32}（40 亿）。在实践中，这个限制并不是问题 – 注意，这里限制的是SQL 命令的数量，而不是处理的行数。另外，只有真正修改数据库内容的命令才会消耗命令标识符。

5.5. 修改表

创建表之后，如果你意识到自己犯了错误，或者应用需求发生了变化，可以删掉表再重新创建。但如果表中已经有数据，或者该表已被其他数据库对象引用（例如被外键约束引用），这样做就不方便了。因此，PostgreSQL 提供了一组命令来修改现有表。注意，这在概念上不同于修改表中保存的数据：这里关注的是修改表的定义，也就是表的结构。

利用这些命令，我们可以：

- 增加列
- 移除列
- 增加约束
- 移除约束
- 修改默认值
- 修改列数据类型
- 重命名列
- 重命名表

所有这些动作都由 ALTER TABLE 命令执行，其参考页面中包含更详细的信息。

5.5.1. 增加列

要添加一列，可以使用这样的命令：

```
ALTER TABLE products ADD COLUMN description text;
```

新列最初会填入给定的默认值（如果没有指定 `DEFAULT` 子句，则填入空值）。

> **提示**
>
> 从 PostgreSQL 11 开始，添加带有常量默认值的列时，在执行ALTER TABLE 语句时不再需要更新表中的每一行。相反，该默认值会在下一次访问该行时返回，并在表被重写时应用，因此即使面对大表，ALTER TABLE 也会非常快。
>
> 然而，如果默认值是易失性的（例如clock_timestamp()），则每一行都需要更新为执行ALTER TABLE 时计算出的值。为避免潜在的长时间更新操作，特别是在你本来就打算用大多数非默认值填充该列时，更好的做法可能是先添加一个没有默认值的列，用 UPDATE 填入正确的值，然后再按下文所述添加所需的默认值。

也可以同时为该列定义约束，使用常规语法：

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

实际上，凡是在CREATE TABLE 中可用于列描述的选项，在这里都可以使用。不过要记住，默认值必须满足给定约束，否则ADD 就会失败。另一种做法是先把新列正确填好，再在之后添加约束（见下文）。

5.5.2. 移除列

要删除一列，使用如下命令：

```
ALTER TABLE products DROP COLUMN description;
```

该列中的数据会消失，涉及该列的表约束也会被删除。不过，如果该列被其他表的外键约束引用，PostgreSQL 不会静默删除该约束。你可以通过添加 CASCADE 来授权删除所有依赖该列的对象：

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

关于这个操作背后的一般性机制请见第 5.13 节。

5.5.3. 增加约束

添加约束时使用表约束语法。例如：

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

非空约束不能写成表约束，添加时应使用以下语法：

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

系统会立即检查约束，因此只有表中的数据满足约束，才能将其添加。

5.5.4. 移除约束

要删除约束，需要知道它的名称。如果曾为它指定名称，这很容易；否则系统会分配一个自动生成的名称，需要将其查出来。psql 的命令`\d tablename`对此很有帮助；其他接口也可能提供检查表细节的方式。然后执行以下命令：

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

（如果处理的是自动生成的约束名，例如`$2`，别忘了加上双引号，使其成为有效标识符。）

和删除列一样，如果要删除某些其他对象所依赖的约束，也需要加上`CASCADE`。例如，外键约束就依赖于被引用列上的唯一约束或主键约束。

除非空约束之外，所有类型的约束都可以用相同方式删除。要删除非空约束，使用：

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

（请记住，非空约束没有名称。）

5.5.5. 更改列的默认值

要为一个列设置一个新默认值，使用命令：

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

注意这不会影响表中任何已经存在的行，它只是为未来的`INSERT`命令改变了默认值。

要移除任何默认值，使用：

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

这等同于将默认值设置为空值。相应地，试图删除一个未被定义的默认值并不会引发错误，因为默认值已经被隐式地设置为空值。

5.5.6. 修改列的数据类型

为了将一个列转换为一种不同的数据类型，使用如下命令：

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

只有当列中的每一个项都能通过一个隐式类型转换为新的类型时该操作才能成功。如果需要一种更复杂的转换，应该加上一个`USING`子句来指定应该如何把旧值转换为新值。

PostgreSQL 将尝试把列的默认值（如果有）转换为新类型，其他涉及到该列的任何约束也是一样。但是这些转换可能失败或者产生奇特的结果。因此通常最好在修改类型之前先删除该列上所有的约束，然后在修改完类型后重新加上相应修改过的约束。

5.5.7. 重命名列

要重命名一个列：

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.5.8. 重命名表

要重命名一个表：

```
ALTER TABLE products RENAME TO items;
```

5.6. 权限

一旦一个对象被创建，它会被分配一个所有者。所有者通常是执行创建语句的角色。对于大部分类型的对象，初始状态下只有所有者（或者超级用户）能够对该对象做任何事情。为了允许其他角色使用它，必须分配权限。

有不同种类的权限：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、TRIGGER、CREATE、CONNECT、TEMPORARY、EXECUTE 和 USAGE。适用于特定对象的权限取决于对象的类型（表、函数等）。关于PostgreSQL 支持的不同权限类型的完整信息，请参阅 GRANT 参考页。后续的各节和各章还将展示这些权限是如何使用的。

修改或销毁对象的权利始终只属于该对象的拥有者。

可以使用适合对象类型的 ALTER 命令，将对象分配给新的拥有者，例如 ALTER TABLE。超级用户始终可以这样做；普通角色只有同时是对象的当前拥有者（或拥有者角色的成员），并且是新拥有者角色的成员，才能这样做。

分配权限使用 GRANT 命令。例如，如果joe 是一个已有角色，并且accounts 是一个已有表，则可以用以下命令授予更新该表的权限：

```
GRANT UPDATE ON accounts TO joe;
```

用 ALL 代替某个具体权限，会授予与该对象类型相关的全部权限。

特殊的“角色”名 PUBLIC 可用于向系统中的每个角色授予权限。此外，当数据库用户很多时，可以设置“组”角色来协助管理权限 – 详见第 21 章。

要撤销权限，使用名称贴切的REVOKE 命令：

```
REVOKE ALL ON accounts FROM PUBLIC;
```

对象拥有者的特殊权限（即执行 DROP，GRANT，REVOKE 等操作的权利）始终隐含在拥有者身份中，不能授予或撤销。不过，对象拥有者可以选择撤销自己的普通权限，例如将表设为对自己和其他人都只读。

通常，只有对象的拥有者（或超级用户）可以授予或撤销对象上的权限。不过，也可以在授予权限时“附带授权选项”，使接收者有权进一步将该权限授予其他角色。如果之后撤销了授权选项，所有直接或经过授权链从该接收者取得此权限的角色都会失去权限。详见GRANT 和 REVOKE 参考页。

5.7. 行安全性策略

除了可通过 GRANT 使用的 SQL 标准权限系统之外，表还可以拥有行安全性策略，用来按用户限制普通查询可以返回哪些行，以及数据修改命令可以插入、更新或删除哪些行。这一特性也称为行级安全。默认情况下，表没有任何策略，因此如果某个用户按照 SQL 权限系统拥有访问该表的权限，那么表中的所有行对查询或更新来说都是同等可用的。

当在表上启用行安全性时（使用 ALTER TABLE ... ENABLE ROW LEVEL SECURITY），所有针对该表选择行或修改行的普通访问都必须得到某条行安全性策略的允许。（不过，表拥有者通常不受行安全性策略约束。）如果该表没有任何策略，则会采用一条默认拒绝策略，也就是说所有行都不可见，也不能被修改。作用于整张表的操作，例如TRUNCATE 和 REFERENCES，不受行安全性约束。

行安全性策略可以针对特定命令、特定角色，或者同时针对两者。策略可以指定适用于ALL 命令，或者适用于SELECT、INSERT、UPDATE 或 DELETE。一条策略也可以分配给多个角色，并且正常的角色成员关系与继承规则同样适用。

要指定根据某条策略哪些行可见或可修改，需要提供一个返回布尔结果的表达式。对每一行来说，在计算来自用户查询的任何条件或函数之前，都会先计算这个表达式。（这条规则的唯一例外是leakproof 函数，它们被保证不会泄露信息；优化器可能会选择在行安全性检查之前应用这类函数。）表达式结果不是true 的行不会被处理。你还可以分别指定独立的表达式，以便独立控制哪些行可见，以及哪些行允许被修改。策略表达式作为查询的一部分运行，并使用执行该查询的用户的权限；不过，可以借助安全性定义者函数访问调用用户本来无权访问的数据。

超级用户以及带有 BYPASSRLS 属性的角色在访问表时总是会绕过行安全性系统。表拥有者通常也会绕过行安全性，不过表拥有者可以通过ALTER TABLE ... FORCE ROW LEVEL SECURITY 选择让自己也受行安全性约束。

启用或禁用行安全性，以及向表添加策略，始终都只属于表拥有者的权限。

策略的创建可以使用 CREATE POLICY 命令，策略的修改可以使用ALTER POLICY 命令，而策略的删除可以使用DROP POLICY 命令。要为一个给定表启用或者禁用行安全性，可以使用ALTER TABLE 命令。

每条策略都有一个名称，而且同一张表可以定义多条策略。由于策略是表级对象，因此同一张表上的每条策略都必须具有唯一名称。不同的表则可以拥有同名策略。

当多条策略适用于某个给定查询时，它们会通过OR（对宽松策略，默认类型）或AND（对限制性策略）组合在一起。这，类似于某个角色拥有其所属全部角色权限的规则。下文还会进一步讨论宽松策略与限制性策略。

作为一个简单的示例，这里是如何在account 关系上创建一条策略以允许只有 managers 角色的成员能访问行，并且只能访问它们账户的行：

```
CREATE TABLE accounts (manager text, company text, contact_email text);

ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY account_managers ON accounts TO managers
    USING (manager = current_user);
```

上面的策略会隐式提供一个与其 USING 子句相同的 WITH CHECK 子句，因此该约束既作用于命令选中的行（也就是说，经理不能SELECT、UPDATE 或 DELETE 属于其他经理的现有行），也作用于命令修改的行（也就是说，不能通过INSERT 或 UPDATE 创建属于其他经理的行）。

如果没有指定角色或者使用了特殊的用户名 PUBLIC，则该策略适用于系统上所有的用户。要允许所有用户访问users 表中属于他们自己的行，可以使用一条简单的策略：

```
CREATE POLICY user_policy ON users
    USING (user_name = current_user);
```

这个示例的效果和前一个类似。

为了对增加到表中的行使用与可见行不同的策略，可以组合多条策略。这一对策略将允许所有用户查看users 表中的所有行，但只能修改他们自己的行：

```
CREATE POLICY user_sel_policy ON users
    FOR SELECT
    USING (true);
CREATE POLICY user_mod_policy ON users
    USING (user_name = current_user);
```

在 SELECT 命令中，这两条策略会使用OR 组合，最终效果就是可以选中所有行。在其他命令类型中，只有第二条策略适用，因此效果和之前相同。

也可以用 ALTER TABLE 命令禁用行安全性。禁用行安全性不会移除定义在表上的任何策略，它们只是被简单地忽略。然后该表中的所有行都是可见的并且可修改，服从于标准的 SQL 权限系统。

下面是一个更大的示例，展示这项特性如何用于生产环境。表 passwd 模拟了一个 Unix 密码文件：

```
-- Simple passwd-file based example
CREATE TABLE passwd (
  user_name             text UNIQUE NOT NULL,
  pwhash                text,
  uid                   int  PRIMARY KEY,
  gid                   int  NOT NULL,
  real_name             text NOT NULL,
  home_phone            text,
  extra_info            text,
  home_dir              text NOT NULL,
  shell                 text NOT NULL
);

CREATE ROLE admin;  -- Administrator
CREATE ROLE bob;    -- Normal user
CREATE ROLE alice;  -- Normal user

-- Populate the table
INSERT INTO passwd VALUES
  ('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/dash');
INSERT INTO passwd VALUES
  ('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/zsh');
INSERT INTO passwd VALUES
  ('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/bin/zsh');

-- Be sure to enable row level security on the table
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;
```

```
-- Create policies
-- Administrator can see all rows and add any rows
CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK (true);
-- Normal users can view all rows
CREATE POLICY all_view ON passwd FOR SELECT USING (true);
-- Normal users can update their own records, but
-- limit which shells a normal user is allowed to set
CREATE POLICY user_mod ON passwd FOR UPDATE
  USING (current_user = user_name)
  WITH CHECK (
    current_user = user_name AND
    shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/tcsh')
  );

-- Allow admin all normal rights
GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;
-- Users only get select access on public columns
GRANT SELECT
  (user_name, uid, gid, real_name, home_phone, extra_info, home_dir, shell)
  ON passwd TO public;
-- Allow users to update certain columns
GRANT UPDATE
  (pwhash, real_name, home_phone, extra_info, shell)
  ON passwd TO public;
```

对于任意安全性设置来说，重要的是测试并确保系统的行为符合预期。使用上述的示例，下面展示了权限系统工作正确：

```
-- admin can view all rows and fields
postgres=> set role admin;
SET
postgres=> table passwd;
 user_name | pwhash | uid | gid | real_name |  home_phone  | extra_info | home_dir    |   shell
-----------+--------+-----+-----+-----------+--------------+------------+-------------+--------
---
 admin     | xxx    |   0 |   0 | Admin     | 111-222-3333 |            | /root       | /bin/
dash
 bob       | xxx    |   1 |   1 | Bob       | 123-456-7890 |            | /home/bob   | /bin/
zsh
 alice     | xxx    |   2 |   1 | Alice     | 098-765-4321 |            | /home/alice | /bin/
zsh
(3 rows)

-- Test what Alice is able to do
postgres=> set role alice;
SET
postgres=> table passwd;
ERROR:  permission denied for table passwd
postgres=> select user_name,real_name,home_phone,extra_info,home_dir,shell from passwd;
 user_name | real_name |  home_phone  | extra_info | home_dir    |   shell
-----------+-----------+--------------+------------+-------------+-----------
 admin     | Admin     | 111-222-3333 |            | /root       | /bin/dash
 bob       | Bob       | 123-456-7890 |            | /home/bob   | /bin/zsh
 alice     | Alice     | 098-765-4321 |            | /home/alice | /bin/zsh
(3 rows)

postgres=> update passwd set user_name = 'joe';
ERROR:  permission denied for table passwd
-- Alice is allowed to change her own real_name, but no others
postgres=> update passwd set real_name = 'Alice Doe';
UPDATE 1
postgres=> update passwd set real_name = 'John Doe' where user_name = 'admin';
UPDATE 0
postgres=> update passwd set shell = '/bin/xx';
ERROR:  new row violates WITH CHECK OPTION for "passwd"
postgres=> delete from passwd;
ERROR:  permission denied for table passwd
postgres=> insert into passwd (user_name) values ('xxx');
ERROR:  permission denied for table passwd
-- Alice can change her own password; RLS silently prevents updating other rows
postgres=> update passwd set pwhash = 'abc';
UPDATE 1
```

到目前为止，我们构造的所有策略都是宽松策略，也就是说，当多条策略都适用时，它们会通过“OR”布尔操作符组合起来。虽然宽松策略也可以构造得只在预期情况下允许访问行，但有时把宽松策略与限制性策略组合起来会更简单；后者是记录必须通过的策略，并且它们会通过“AND”布尔操作符组合。在上面的示例基础上，我们再增加一条限制性策略，要求管理员必须通过本地 Unix 套接字连接，才能访问`passwd`表中的记录：

```
CREATE POLICY admin_local_only ON passwd AS RESTRICTIVE TO admin
    USING (pg_catalog.inet_client_addr() IS NULL);
```

然后，由于这条限制性策略的存在，我们可以看到从网络连接进来的管理员将无法看到任何记录：

```
=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> select inet_client_addr();
 inet_client_addr
------------------
 127.0.0.1
(1 row)

=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> TABLE passwd;
 user_name | pwhash | uid | gid | real_name | home_phone | extra_info | home_dir | shell
-----------+--------+-----+-----+-----------+------------+------------+----------+-------
(0 rows)

=> UPDATE passwd set pwhash = NULL;
UPDATE 0
```

引用完整性检查，例如唯一约束或主键约束以及外键引用，总是会绕过行级安全，以确保数据完整性得到维护。在设计模式和行级策略时必须小心，避免通过这类引用完整性检查形成“隐蔽通道”并泄露信息。

在某些场景下，确保没有应用行安全性很重要。例如在做备份时，如果行安全性静默地导致某些行被从备份中省略，那将是灾难性的。在这种情况下，可以把row_security 配置参数设置为 `off`。这本身并不会绕过行安全性；它的作用是，只要某个查询结果本会被策略过滤，就直接抛出错误。这样就可以调查并修复出错原因。

在上面的示例中，策略表达式只考虑了要被访问或更新的行中的当前值。这是最简单并且性能最好的情况。如果可能，最好设计行安全性应用以这种方式工作。如果需要参考其他行或者其他表来做出策略决定，可以在策略表达式中通过使用子`SELECT` 或包含 `SELECT` 的函数来实现。不

过要注意这类访问可能会导致竞争条件，在不小心的情况下这可能会导致信息泄露。作为一个示例，考虑下面的表设计：

```
-- definition of privilege groups
CREATE TABLE groups (group_id int PRIMARY KEY,
                     group_name text NOT NULL);

INSERT INTO groups VALUES
  (1, 'low'),
  (2, 'medium'),
  (5, 'high');

GRANT ALL ON groups TO alice;  -- alice is the administrator
GRANT SELECT ON groups TO public;

-- definition of users' privilege levels
CREATE TABLE users (user_name text PRIMARY KEY,
                    group_id int NOT NULL REFERENCES groups);

INSERT INTO users VALUES
  ('alice', 5),
  ('bob', 2),
  ('mallory', 2);

GRANT ALL ON users TO alice;
GRANT SELECT ON users TO public;

-- table holding the information to be protected
CREATE TABLE information (info text,
                          group_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES
  ('barely secret', 1),
  ('slightly secret', 2),
  ('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- a row should be visible to/updatable by users whose security group_id is
-- greater than or equal to the row's group_id
CREATE POLICY fp_s ON information FOR SELECT
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));
CREATE POLICY fp_u ON information FOR UPDATE
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));

-- we rely only on RLS to protect the information table
GRANT ALL ON information TO public;
```

现在假设 `alice` 想修改 “slightly secret” 这条信息，但认为`mallory` 不应该看到这一行的新内容，因此她这样做：

```
BEGIN;
UPDATE users SET group_id = 1 WHERE user_name = 'mallory';
UPDATE information SET info = 'secret from mallory' WHERE group_id = 2;
COMMIT;
```

这看起来是安全的；似乎不存在 mallory 能够看到 “secret from mallory” 这个字符串的窗口。不过，这里存在一个竞争条件。如果mallory 正在并发执行例如：

```
SELECT * FROM information WHERE group_id = 2 FOR UPDATE;
```

并且她的事务处于 READ COMMITTED 模式，那么她就有可能看到 “secret from mallory” 。这种情况会在她的事务恰好在 alice 之后到达 information 中的那一行时发生。它会阻塞并等待 alice 的事务提交，然后由于FOR UPDATE 子句而取到更新后的行内容。但是，对于来自 users 的隐式 SELECT，它不会取到更新后的行，因为该子SELECT 没有 FOR UPDATE；相反，users 中的那一行是用查询开始时取得的快照读取的。因此，策略表达式测试的是mallory 权限级别的旧值，并允许她看到更新后的行。

有多种方法能解决这个问题。一种简单的答案是在行安全性策略中的子 SELECT 里使用 SELECT ... FOR SHARE。不过，这要求在被引用表（这里是users）上授予 UPDATE 权限给受影响的用户，这可能不是我们想要的（但是另一条行安全性策略可能被应用来阻止它们实际使用这个权限，或者子SELECT 可能被嵌入到一个安全性定义者函数中）。还有，在被引用的表上过多并发地使用行共享锁可能会导致性能问题，特别是表更新比较频繁时。另一种解决方案（如果被引用表上的更新不频繁就可行）是在更新被引用表时对它取一个ACCESS EXCLUSIVE 锁，这样就没有并发事务能够检查旧的行值了。或者我们可以在提交对被引用表的更新之后、在做依赖于新安全性情况的更改之前等待所有并发事务结束。

更多细节请见 CREATE POLICY 和 ALTER TABLE。

5.8. 模式

一个 PostgreSQL 数据库集簇中包含一个或更多命名的数据库。角色和一些其他对象类型被整个集簇共享，连接到服务器的客户端只能访问单个数据库中的数据，在连接请求中指定的那一个。

> **注意**
>
> 一个集簇的用户不一定拥有访问集簇中每一个数据库的权限。角色名的共享意味着，同一集簇的两个数据库中不能存在两个不同的、都名为joe 的角色。但系统可以被配置为只允许 joe 访问某些数据库。

一个数据库包含一个或多个命名的模式，模式中又包含表。模式还包含其他类型的命名对象，包括数据类型、函数和操作符。相同的对象名可以在不同模式中重复使用而不发生冲突；例如，schema1 和 myschema 都可以包含名为 mytable 的表。与数据库不同，模式并不是被严格隔离的：只要拥有相应权限，用户就可以访问其所连接数据库中任意模式里的对象。

下面是一些使用模式的原因：

- 允许多个用户使用一个数据库并且不会互相干扰。
- 将数据库对象组织成逻辑组以便更容易管理。
- 第三方应用的对象可以放在独立的模式中，这样它们就不会与其他对象的名称发生冲突。

模式类似于操作系统层的目录，但是模式不能嵌套。

5.8.1. 创建模式

要创建一个模式，可使用CREATE SCHEMA 命令，并且给出选择的模式名称。例如：

```
CREATE SCHEMA myschema;
```

在一个模式中创建或访问对象，需要使用由模式名和表名构成的限定名，模式名和表名之间以点号分隔：

```
schema.table
```

在任何需要一个表名的地方都可以这样用，包括表修改命令和后续章节要讨论的数据访问命令（为了简洁我们在这里只谈到表，但是这种方式对其他类型的命名对象同样有效，例如类型和函数）。

事实上，还有更加通用的语法：

```
database.schema.table
```

也可以使用，但是目前它只是在形式上与 SQL 标准兼容。如果我们写一个数据库名称，它必须是我们正在连接的数据库。

因此，如果要在一个新模式中创建一个表，可用：

```
CREATE TABLE myschema.mytable (
 ...
);
```

要删除一个为空的模式（其中的所有对象已经被删除），可用：

```
DROP SCHEMA myschema;
```

要删除一个模式以及其中包含的所有对象，可用：

```
DROP SCHEMA myschema CASCADE;
```

有关于此的更一般的机制请参见第 5.13 节。

我们常常希望创建一个由其他人所拥有的模式（因为这是将用户的操作限制在明确定义的名字空间中的方法之一）。其语法是：

```
CREATE SCHEMA schema_name AUTHORIZATION user_name;
```

我们甚至可以省略模式名称，在此种情况下模式名称将会使用用户名，参见第 5.8.6 节。

以 pg_开头的模式名被保留用于系统目的，所以不能被用户所创建。

5.8.2. 公共模式

在前面的小节中，我们创建表时都没有指定模式名。默认情况下，这些表（以及其他对象）会自动放入名为 “public” 的模式中。每个新数据库都包含这样一个模式。因此，下面两条命令是等效的：

```
CREATE TABLE products ( ... );
```

以及：

```
CREATE TABLE public.products ( ... );
```

5.8.3. 模式搜索路径

限定名写起来很繁琐，而且通常最好不要把某个特定模式名硬编码到应用中。因此，表通常通过非限定名来引用，也就是只写表名。系统会沿着一条搜索路径来决定该名称指的是哪个表；搜索路径就是要查找的一组模式列表。搜索路径中第一个匹配的表会被视为目标表。如果搜索路径中没有任何匹配，就会报错，即使数据库的其他模式中存在同名表也是如此。

在不同模式中创建命名相同的对象的能力使得编写每次都准确引用相同对象的查询变得复杂。这也使得用户有可能更改其他用户查询的行为，不管是出于恶意还是无意。由于未经限定的名称在查询中以及在PostgreSQL 内部的广泛使用，在`search_path` 中增加一个模式实际上是信任所有在该模式中具有 `CREATE` 权限的用户。在你运行一个普通查询时，如果恶意用户可以在搜索路径的模式中创建对象，那么他们就可以取得控制权，以你的身份执行任意 SQL 函数。

搜索路径中的第一个模式被称为当前模式。除了是第一个被搜索的模式外，如果`CREATE TABLE` 命令没有指定模式名，它将是新创建表所在的模式。

要显示当前搜索路径，可使用下面的命令：

```
SHOW search_path;
```

在默认设置下这将返回：

```
 search_path
--------------
 "$user", public
```

第一个元素说明一个和当前用户同名的模式会被搜索。如果不存在这个模式，该项将被忽略。第二个元素指向我们已经见过的公共模式。

搜索路径中第一个存在的模式，是创建新对象时的默认位置。这就是默认情况下对象会被创建在公共模式中的原因。当对象在其他任何未限定模式的上下文中被引用时（表修改、数据修改或查询命令），系统会沿搜索路径查找，直到找到匹配对象为止。因此，在默认配置中，任何非限定访问仍然只能指向公共模式。

要把新模式放在搜索路径中，我们可以使用：

```
SET search_path TO myschema,public;
```

（我们在这里省略了 $user，因为我们并不立即需要它）。然后我们可以删除该表而无需使用模式进行限定：

```
DROP TABLE mytable;
```

同样，由于myschema 是路径中的第一个元素，新对象默认也会创建在其中。

我们也可以这样写：

```
SET search_path TO myschema;
```

这样一来，在没有显式限定时，我们就不再能访问公共模式了。公共模式本身并无特殊之处，只是默认存在而已；它同样可以被删除。

其他操作模式搜索路径的方法请见第 9.25 节。

搜索路径对于数据类型名称、函数名称和操作符名称的作用与表名一样。数据类型和函数名称可以使用和表名完全相同的限定方式。如果我们需要在一个表达式中写一个限定的操作符名称，我们必须写成一种特殊的形式：

```
OPERATOR(schema.operator)
```

这是为了避免句法歧义。例如：

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

实际上我们通常都会依赖于搜索路径来查找操作符，因此没有必要去写如此“丑陋”的东西。

5.8.4. 模式和权限

默认情况下，用户无法访问他们不拥有的模式中的任何对象。要允许这样做，模式的所有者必须授予该模式上的USAGE 权限。要允许用户使用模式中的对象，可能还需要根据对象的类型授予相应的权限。

也可以允许用户在他人的模式中创建对象。为此，需要授予该模式上的CREATE 权限。注意，默认情况下，每个人都具有CREATE 和 USAGE 权限，它们作用于模式public。这使所有能够连接到某个数据库的用户都可以在它的 public 模式中创建对象。一些使用方式要求撤销此权限：

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

（第一个“public”是模式名，第二个“public”表示“每个用户”。前一种用法是标识符，后一种是关键字，因此大小写不同；请回顾第 4.1.1 节。）

5.8.5. 系统目录模式

除 public 和用户创建的模式之外，每一个数据库还包括一个pg_catalog 模式，它包含了系统表和所有内置的数据类型、函数以及操作符。pg_catalog 总是搜索路径的一个有效部分。如果

没有在路径中显式地包括该模式，它将在路径中的模式之前被搜索。这保证了内置的名称总是能被找到。然而，如果我们希望用用户定义的名称覆盖内置的名称，可以显式地将pg_catalog 放在搜索路径的末尾。

由于系统表名以 pg_开头，因此最好避免使用这样的名称，以免将来某个版本定义出与你的表同名的系统表。（使用默认搜索路径时，对你的表名的非限定引用就会被解析为该系统表。）系统表会继续遵循以pg_开头的约定，因此只要用户避免使用pg_前缀，它们就不会与未经限定的用户表名发生冲突。

5.8.6. 使用方式

模式可以用多种方式组织数据。模式的安全使用方式可防止不受信任的用户改变其他用户查询的行为。如果数据库未采用安全的模式使用方式，希望安全查询该数据库的用户应在每个会话开始时采取保护措施。具体而言，应在每个会话开始时将search_path 设为空字符串，或者以其他方式移除非超级用户可写的模式，使它们不再出现在以下路径中：search_path。默认配置很容易支持以下几种使用方式：

- 将普通用户限制在各自私有的模式中。为此，执行REVOKE CREATE ON SCHEMA public FROM PUBLIC，并为每个用户创建一个与其同名的模式。请记住，默认搜索路径以$user 开头，它会解析为用户名。因此，如果每个用户都有单独的模式，他们默认访问的就是自己的模式。在已经有不受信任用户登录过的数据库中采用这种方式后，应考虑检查 public 模式中是否存在与pg_catalog 模式中的对象同名的对象。除非不受信任的用户是数据库拥有者或持有CREATEROLE 权限，否则这是一种安全的模式使用方式；如果存在上述情况，则没有安全的模式使用方式。

- 通过修改 postgresql.conf，或执行ALTER ROLE ALL SET search_path = "$user"，将 public 模式从默认搜索路径中移除。每个人仍可在 public 模式中创建对象，但只有限定名才会选中这些对象。虽然限定的表引用没有问题，但调用 public 模式中的函数会不安全或不可靠。如果要在 public 模式中创建函数或扩展，请改用第一种方式。否则，它和第一种方式一样，除非不受信任的用户是数据库拥有者或持有CREATEROLE 权限，否则就是安全的。

- 保持默认设置。所有用户都会隐式访问 public 模式。这模拟了根本没有模式可用的情况，使从不支持模式的环境迁移过来时更加平滑。不过，这绝不是一种安全的方式。它只适用于数据库只有单个用户，或只有少数彼此信任的用户的场景。

对于任何一种模式，如果要安装共享应用（所有人都要使用的表、第三方提供的附加函数等），可以把它们放进单独的模式中。记得授予适当的权限，以便其他用户能够访问它们。这样，用户既可以通过带模式名的限定名来引用这些附加对象，也可以按自己的需要把这些附加模式加入搜索路径。

5.8.7. 可移植性

在 SQL 标准中，不存在同一模式中的对象由不同用户拥有这一概念。此外，有些实现不允许创建与其所有者名称不同的模式。事实上，在那些只实现了标准中基本模式支持的数据库系统中，模式和用户这两个概念几乎是等价的。因此，很多用户认为限定名实际上就是 *user_name*.*table_name*。如果你为每个用户都创建一个独立模式，PostgreSQL 的行为实际上也会是这样。

同样，SQL 标准中也没有public 模式这一概念。为了尽可能符合标准，你不应使用public 模式。

当然，某些 SQL 数据库系统可能根本没有实现模式，或者提供（很可能是有限制地）允许跨数据库访问的命名空间。如果需要使用这样的系统，为了获得最好的可移植性，最好不要使用模式。

5.9. 继承

PostgreSQL 实现了表继承，这对数据库设计者来说是一种有用的工具（SQL:1999 及其后的版本定义了一种类型继承特性，但和这里介绍的继承有很大的不同）。

让我们从一个示例开始：假设我们要为城市建立一个数据模型。每个州有很多城市，但只有一个首府。我们希望能够快速检索任意特定州的首府城市。这可以通过创建两个表来实现：一个用于州首府，另一个用于非首府城市。然而，当我们想要查询某个城市的数据，而不关心它是不是首府时，会发生什么？继承特性将有助于解决这个问题。我们可以将`capitals` 表定义为继承自 `cities` 表：

```
CREATE TABLE cities (
    name            text,
    population      float,
    elevation       int     -- 单位为英尺
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

在这种情况下，`capitals` 表继承了它的父表 `cities` 的所有列。州首府还有一个额外的列 `state` 用来表示它所属的州。

在 PostgreSQL 中，一个表可以从 0 个或者多个其他表继承，而对一个表的查询则可以引用一个表的所有行或者该表的所有行加上它所有的后代表。默认情况是后一种行为。例如，下面的查询将查找所有高度高于 500 英尺的城市的名称，包括州首府：

```
SELECT name, elevation
    FROM cities
    WHERE elevation > 500;
```

对于来自 PostgreSQL 教程（见第 2.1 节）的示例数据，它将返回：

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
```

另一方面，下面的查询将找到所有高度超过 500 英尺且不是州首府的城市：

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;

   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
```

这里的 ONLY 关键词指示查询只被应用于 cities 上，而其他在继承层次中位于cities 之下的其他表都不会被该查询涉及。很多我们已经讨论过的命令（如 SELECT、UPDATE 和 DELETE）都支持 ONLY 关键词。

我们也可以在表名后写上一个* 来显式地将后代表包括在查询范围内：

```
SELECT name, elevation
    FROM cities*
    WHERE elevation > 500;
```

写* 并非必需，因为这种行为始终是默认的。不过，为了兼容那些允许修改默认行为的较旧版本，现在仍然支持这种语法。

在某些情况下，我们可能希望知道一个特定行来自于哪个表。每个表中的系统列tableoid 可以告诉我们行来自于哪个表：

```
SELECT c.tableoid, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

将会返回：

```
 tableoid |   name    | elevation
----------+-----------+-----------
   139793 | Las Vegas |      2174
   139793 | Mariposa  |      1953
   139798 | Madison   |       845
```

（如果重新生成这个结果，可能会得到不同的 OID 数字。）通过与pg_class 进行连接可以看到实际的表名：

```
SELECT p.relname, c.name, c.elevation
FROM cities c, pg_class p
WHERE c.elevation > 500 AND c.tableoid = p.oid;
```

将会返回：

```
 relname  |   name    | elevation
----------+-----------+-----------
 cities   | Las Vegas |      2174
 cities   | Mariposa  |      1953
 capitals | Madison   |       845
```

得到同样效果的另一种方法，是使用regclass 别名类型，它会以符号形式打印表的 OID：

```
SELECT c.tableoid::regclass, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

继承不会自动地将来自 INSERT 或 COPY 命令的数据传播到继承层次中的其他表中。在我们的示例中，下面的INSERT 语句将会失败：

```
INSERT INTO cities (name, population, elevation, state)
VALUES ('Albany', NULL, NULL, 'NY');
```

我们也许会希望数据能以某种方式被路由到 capitals 表中，但这不会发生：INSERT 总是向指定的表中插入。在某些情况下，可以通过使用一个规则（见第 41 章）将插入动作重定向。但是这对上面的情况并没有帮助，因为cities 表根本就不包含 state 列，因而这个命令会在触发规则之前就被拒绝。

父表上的所有检查约束和非空约束都将自动被它的后代所继承，除非显式地指定了NO INHERIT 子句。其他类型的约束（唯一、主键和外键约束）则不会被继承。

一个表可以从超过一个的父表继承，在这种情况下它拥有父表们所定义的列的并集。任何定义在子表上的列也会被加入到其中。如果在这个集合中出现重名列，那么这些列将被“合并”，这样在子表中只会有一个这样的列。重名列能被合并的前提是这些列必须具有相同的数据类型，否则会导致错误。可继承的检查约束和非空约束会以类似的方式被合并。例如，如果合并成一个合并列的任一列定义被标记为非空，则该合并列会被标记为非空。如果检查约束的名称相同，则它们会被合并，但如果它们的条件不同则合并会失败。

表继承通常在创建子表时建立，即通过CREATE TABLE 语句中的 INHERITS 子句。已经创建且定义兼容的表也可以通过 ALTER TABLE 的 INHERIT 变体再增加一个新的父表关系。要这么做，新子表必须已经包含与父表同名且数据类型相同的列。子表还必须包含名称和检查表达式与父表相同的检查约束。类似地，也可以使用ALTER TABLE 的 NO INHERIT 变体，从子表中移除一条继承链接。动态添加和移除继承链接可用于实现表分区（见第 5.10 节）。

一种方便的方法是在 CREATE TABLE 中使用 LIKE 子句，创建一个兼容的表，以便以后将其用作新的子表。这样会创建一个与源表具有相同列的新表。如果源表上定义了任何CHECK 约束，就应指定LIKE 的 INCLUDING CONSTRAINTS 选项，因为新子表必须具备与父表匹配的约束，才会被视为兼容。

当有任何一个子表存在时，父表不能被删除。当子表的列或者检查约束继承于父表时，它们也不能被删除或修改。如果希望移除一个表和它的所有后代，一种简单的方法是使用CASCADE 选项删除父表（见第 5.13 节）。

ALTER TABLE 会把列的数据定义或检查约束上的任何变化沿着继承层次向下传播。同样，删除被其他表依赖的列只能使用CASCADE 选项。对于同名列的合并与拒绝，ALTER TABLE 遵循与 CREATE TABLE 相同的规则。

继承查询只对父表进行访问权限检查。因此，例如授予cities 表上的 UPDATE 权限，就意味着通过cities 访问时，也有权更新capitals 表中的行。这保留了数据（也）位于父表中的表象。不过，未经额外授权，不能直接更新capitals 表。这条规则有两个例外：TRUNCATE 和 LOCK TABLE。无论直接处理子表，还是对父表执行这些命令并递归处理子表，都会检查子表的权限。

类似地，在继承查询期间，父表的行安全策略（见第 5.7 节）会应用于来自子表的行。子表自己的策略（如果有）只会在查询中明确指定该子表时应用；此时，其父表的任何策略都会被忽略。

外部表（见第 5.11 节）也可以是继承层次中的一部分，既可作为父表也可作为子表，就像常规表一样。如果一个外部表是继承层次的一部分，那么任何不被该外部表支持的操作也不被整个层次所支持。

5.9.1. 注意事项

注意，并非所有 SQL 命令都能作用于继承层次。用于数据查询、数据修改或模式修改的命令（例如SELECT、UPDATE、DELETE、大多数ALTER TABLE 变体，但不包括INSERT 或 ALTER TABLE ... RENAME）通常默认包含子表，并支持使用ONLY 记法将其排除。用于数据库维护和调优的命令（例如 REINDEX、VACUUM）通常只作用于独立的物理表，不支持沿继承层次递归。各条命令的具体行为都记录在相应参考页中（SQL 命令）。

继承特性的一个严重限制是，索引（包括唯一约束）和外键约束只作用于单个表，而不作用于其继承子表。对于外键约束的引用端和被引用端，这一点都成立。因此，沿用上面的示例：

- 如果我们把 cities.name 声明为 UNIQUE 或 PRIMARY KEY，这并不能阻止capitals 表中出现与 cities 中城市同名的行。而且这些重复行默认还会出现在针对 cities 的查询结果中。事实上，默认情况下capitals 根本没有唯一约束，因此它可以包含多行同名记录。你当然可以给capitals 添加唯一约束，但这仍无法阻止相对于cities 的重复。

- 相似地，如果我们指定cities.name REFERENCES 某个其他表，该约束不会自动地传播到capitals。在此种情况下，我们可以变通地在capitals 上手工创建一个相同的 REFERENCES 约束。

- 如果让另一个表的某列 REFERENCES cities(name)，那么该表可以包含城市名称，但不能包含首府名称。对于这种情况，并没有什么好的变通办法。

某些未为继承层次实现的功能，却已经为声明式分区实现了。因此，在决定使用旧式继承分区是否适合你的应用时，需要非常小心。

5.10. 表分区

PostgreSQL 支持基本的表分区。本节介绍为什么以及如何把分区作为数据库设计的一部分来实现。

5.10.1. 概述

分区是指把逻辑上的一个大表拆分成一些较小的物理片段。分区可以带来若干好处：

- 在某些情况下查询性能能够显著提升，特别是当表中频繁访问的行大多集中在一个或少数几个分区时。分区有效地替代了索引树的上层，使得索引中频繁使用的部分更有可能容纳在内存中。

- 当查询或更新需要访问单个分区中的很大一部分数据时，顺序扫描该分区可以提高性能，优于使用索引并在整个表中执行分散的随机读取。

- 如果在分区设计中考虑到了这种使用模式，就可以通过添加或移除分区来完成批量加载和删除。使用DROP TABLE 删除单个分区，或执行ALTER TABLE DETACH PARTITION 操作，都比批量操作快得多。这些命令还完全避免了批量DELETE 所导致的 VACUUM 开销。

- 很少使用的数据可以被迁移到便宜且较慢的存储介质上。

通常，只有表本来会很大时，分区带来的这些好处才值得考虑。一个表在什么情况下会从分区中受益取决于具体应用；一条经验法则是，当表的大小超过数据库服务器的物理内存时，分区通常会带来好处。

PostgreSQL 对下列分区形式提供了内置支持：

范围分区

表按照某个键列或一组键列划分为若干“范围”，不同分区的范围彼此不重叠。例如，可以按日期范围进行分区，或者按某个特定业务对象的标识符范围进行分区。每个范围的边界都被理解为左端包含、右端不包含。例如，如果一个分区的范围是从`1` 到 `10`，而下一个分区是从 `10` 到 `20`，那么值`10` 属于第二个分区，而不是第一个分区。

列表分区

通过显式列出每个分区中出现的键值来对表进行分区。

Hash 分区

通过为每个分区指定模数和余数来对表进行分区。每个分区保存的都是这样一些行：这些行的分区键 hash 值除以指定模数后，会得到该分区指定的余数。

如果你的应用需要使用上面所列之外的分区形式，可以使用诸如继承和`UNION ALL` 视图之类的替代方法。这些方法很灵活，但是却缺少内置声明式分区的一些性能优势。

5.10.2. 声明式分区

PostgreSQL 允许你声明将一个表划分为多个分区。被划分的表称为分区表。该声明包括前文所述的分区方法，以及用作分区键的列或表达式列表。

分区表本身是一个“虚拟”表，没有自己的存储空间。相反，存储属于分区，这些分区是与分区表关联的普通表。每个分区存储由其分区边界定义的数据子集。插入到分区表中的所有行都将根据分区键列的值路由到适当的一个分区。如果一行不再满足其原始分区的分区边界，则更新该行的分区键将导致该行移动到另一个分区中。

分区本身可能被定义为分区表，从而导致子分区。尽管所有分区必须具有与其分区父级相同的列，但分区可能具有自己的与其他分区不同的索引、约束以及默认值。创建分区表及分区的更多细节请见 CREATE TABLE。

不可能将常规表转换为分区表，反之亦然。但是，可以把现有的常规表或分区表加入某个分区表作为分区，或者从分区表中移除分区并将其转换为独立表；这可以简化并加快许多维护过程。有关`ATTACH PARTITION` 和 `DETACH PARTITION` 子命令的详细信息，请参见 ALTER TABLE。

分区也可以是外部表，但是需要非常小心，因为这时用户需要确保外部表的内容符合分区规则。还有一些其他限制。更多信息请参见 CREATE FOREIGN TABLE。

5.10.2.1. 示例

假定我们正在为一个大型的冰激凌公司构建数据库。该公司每天测量最高温度以及每个区域的冰激凌销售情况。概念上，我们想要一个这样的表：

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

我们知道大部分查询只会访问上周的、上月的或者上季度的数据，因为这个表的主要用途是为管理层准备在线报告。为了减少需要被存放的旧数据量，我们决定只保留最近 3 年的数据。在每个月的开始我们将去除掉最早的那个月的数据。在这种情况下我们可以使用分区技术来帮助我们满足对 measurement 表的所有不同需求。

要在这种情况下使用声明式分区，可采用下面的步骤：

1. 通过指定 `PARTITION BY` 子句把 `measurement` 表创建为分区表，该子句包括分区方法（这个示例中是`RANGE`）以及用作分区键的列列表。

   ```
   CREATE TABLE measurement (
       city_id         int not null,
       logdate         date not null,
       peaktemp        int,
       unitsales       int
   ) PARTITION BY RANGE (logdate);
   ```

2. 创建分区。每个分区的定义必须指定与父表的分区方法和分区键对应的边界。请注意，指定边界如果使得新分区的值与一个或多个现有分区中的值重叠将导致错误。

   分区以普通 PostgreSQL 表（或者可能是外部表）的方式创建。可以为每个分区单独指定表空间和存储参数。

在我们的示例中，每个分区应该保存一个月的数据，以满足一次删除一个月数据的要求。因此，这些命令可能看起来像：

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

CREATE TABLE measurement_y2006m03 PARTITION OF measurement
    FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');

...
CREATE TABLE measurement_y2007m11 PARTITION OF measurement
    FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');

CREATE TABLE measurement_y2007m12 PARTITION OF measurement
    FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')
    TABLESPACE fasttablespace;

CREATE TABLE measurement_y2008m01 PARTITION OF measurement
    FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')
    WITH (parallel_workers = 4)
    TABLESPACE fasttablespace;
```

（回想一下，相邻分区可以共享一个边界值，因为范围上限被视为不包含的边界。）

如果你打算实现子分区，再次在创建分区的命令中指定PARTITION BY 子句，例如：

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')
    PARTITION BY RANGE (peaktemp);
```

在创建了 measurement_y2006m02 的分区之后，任何插入到measurement 并被路由到 measurement_y2006m02 的数据（或者直接插入到 measurement_y2006m02、且满足该分区约束的数据）都会再根据peaktemp 列进一步路由到 measurement_y2006m02 的某个分区。指定的分区键可以与父分区的分区键重叠，不过在指定子分区边界时要注意，它所接受的数据集合必须是该分区自身边界所允许数据集合的一个子集；系统不会尝试检查这一点。

将没有映射到任何现有分区的数据插入父表将导致错误；必须手动添加适当的分区。

不需要手动创建描述分区边界条件的表约束。此类约束将自动创建。

3. 在分区表的键列上创建索引，并创建任何其他需要的索引。（键索引不是严格必需的，但在大多数情况下它是有用的。）会自动在每个分区上创建一个匹配的索引，你稍后创建或附加的任何分区也会有这样的索引。在分区表上声明的索引或唯一约束与分区表一样都是“虚拟的”：实际数据位于各个分区表的子索引中。

```
CREATE INDEX ON measurement (logdate);
```

4. 确保 enable_partition_pruning 配置参数在 postgresql.conf 中没有被禁用。如果被禁用，查询将不会按照想要的方式被优化。

在上面的示例中，我们会每个月创建一个新分区，因此写一个脚本来自动生成所需的 DDL 会更好。

5.10.2.2. 分区维护

通常在初始定义分区表时建立的分区并非保持静态不变。移除存放旧数据的分区并且为新数据周期性地增加新分区的需求比比皆是。分区的最大好处之一就是可以通过操纵分区结构来近乎瞬时地执行这类让人头痛的任务，而不是物理地移动大量数据。

移除旧数据最简单的选择是删除掉不再需要的分区：

```
DROP TABLE measurement_y2006m02;
```

这可以非常快速地删除数百万行记录，因为它不需要逐行删除。不过要注意，上面的命令需要在父表上获取ACCESS EXCLUSIVE 锁。

另一个通常更合适的选择是将分区从分区表中分离，但保留它作为独立表的可访问性：

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;
```

这样就可以在删除数据前继续对其执行操作。例如，这时通常适合使用COPY，pg_dump 或类似工具备份数据，也适合将数据聚合为更紧凑的形式、执行其他数据操作或运行报表。

类似地，可以添加新分区来处理新数据。可以像上面创建初始分区那样，在分区表中创建空分区：

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
    TABLESPACE fasttablespace;
```

另一种有时更方便的做法是先在分区结构之外创建新表，之后再将其变为正式分区。这样，新数据在出现在分区表中之前，就可以先被载入、检查和转换。CREATE TABLE ... LIKE 选项有助于避免繁琐地重复父表定义：

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
  TABLESPACE fasttablespace;

ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```

在运行 ATTACH PARTITION 命令之前，建议在要附加的表上创建一个匹配预期分区约束的 CHECK 约束，如上所示。这样，系统就能够跳过验证隐式分区约束所需的扫描。没有CHECK 约束，表将在持有父表的ACCESS EXCLUSIVE 锁的情况下进行扫描以验证分区约束。建议在 ATTACH PARTITION 完成后删除现在多余的 CHECK 约束。

如上所述，可以在分区表上创建索引，使其自动应用于整个层次。这很方便，因为不仅已有分区会建立索引，将来创建的分区也会建立索引。一个限制是，创建这种分区索引时不能使用CONCURRENTLY 限定词。为避免长时间持锁，可以使用CREATE INDEX ON ONLY 分区表；这样的索引会被标记为无效，并且不会自动应用到分区。可以在各个分区上使用 CONCURRENTLY 分别创建索引，然后将它们附加到父表索引，所用命令为ALTER INDEX .. ATTACH PARTITION。当所有分区的索引都附加到父索引后，父索引就会自动标记为有效。例如：

```
CREATE INDEX measurement_usls_idx ON ONLY measurement (unitsales);

CREATE INDEX measurement_usls_200602_idx
    ON measurement_y2006m02 (unitsales);
ALTER INDEX measurement_usls_idx
    ATTACH PARTITION measurement_usls_200602_idx;
...
```

此技术也可以用于 UNIQUE 和 PRIMARY KEY 约束；索引会在创建约束时隐式创建。例如：

```
ALTER TABLE ONLY measurement ADD UNIQUE (city_id, logdate);

ALTER TABLE measurement_y2006m02 ADD UNIQUE (city_id, logdate);
ALTER INDEX measurement_city_id_logdate_key
    ATTACH PARTITION measurement_y2006m02_city_id_logdate_key;
...
```

5.10.2.3. 限制

分区表有以下限制：

- 要在分区表上创建唯一或主键约束，分区键不能包含任何表达式或函数调用，约束的列必须包括所有分区键列。存在此限制是因为构成约束的各个索引只能直接在各自的分区内强制唯一性；因此，分区结构本身必须保证不同分区中没有重复数据。
- 无法创建跨越整个分区表的排他约束，只能在每个叶子分区上分别设置这种约束。同样，这一限制源于无法强制执行跨分区的约束。
- 虽然分区表支持主键，但不支持引用分区表的外键。（支持从分区表引用其他表的外键。）
- 如果需要 BEFORE ROW 触发器，必须在各个分区上定义，而不能在分区表上定义。
- 在同一分区树中不允许混合临时和永久关系。因此，如果分区表是永久的，那么它的分区也必须是永久的；同样，如果分区表是临时的，那么它的分区也必须是临时的。在使用临时关系时，分区树的所有成员必须来自同一个会话。

各个分区在内部通过继承链接到其分区表。不过，并非所有通用的继承特性都能用于声明式分区表及其分区，如下所述。值得注意的是，一个分区除了其所属的分区表之外不能再有其他父表，也不能同时从分区表和普通表继承。这意味着分区表及其分区永远不会与普通表共享继承层次。

由于分区表及其分区所组成的分区层次仍然是继承层次，tableoid 以及所有普通继承规则仍然适用，具体见第 5.9 节，但有以下例外：

- 分区不能具有父级中不存在的列。在使用 CREATE TABLE 创建分区时无法指定列，也无法使用ALTER TABLE 在事后添加列到分区。只有当表的列与父级完全匹配（包括任何 oid 列）时，才能使用ALTER TABLE ... ATTACH PARTITION 将表添加为分区。

- 分区表的 CHECK 和 NOT NULL 约束始终被所有分区继承。标记为 NO INHERIT 的 CHECK 约束不允许在分区表上创建。如果分区的列上存在与父表中相同的 NOT NULL 约束，则无法删除该约束。
- 只有在没有分区存在时，才支持在分区表上使用ONLY 添加或删除约束。一旦存在分区，使用 ONLY 将导致错误。可以在分区上添加约束，并且（如果这些约束在父表中不存在）可以删除它们。
- 由于分区表本身没有任何数据，因此尝试在分区表上使用TRUNCATE ONLY 将始终返回错误。

5.10.3. 使用继承的分区

虽然内置的声明式分区适用于大多数常见场景，但有时更灵活的方法可能更有用。可以使用表继承来实现分区，它允许使用声明式分区所不支持的一些功能，例如：

- 对声明式分区来说，分区必须具有和分区表正好相同的列集合，而在表继承中，子表可以有父表中没有出现过的额外列。
- 表继承允许多继承。
- 声明式分区只支持范围、列表和 hash 分区，而表继承允许按用户选择的方式对数据进行分区。（不过注意，如果约束排除无法有效剪枝子表，查询性能可能会很差。）
- 某些操作在使用声明式分区时，比使用表继承时需要更强的锁。例如，向分区表添加分区或从中移除分区，都需要在父表上取得ACCESS EXCLUSIVE 锁，而对于普通继承，SHARE UPDATE EXCLUSIVE 锁就足够了。

5.10.3.1. 示例

本示例构建与上述声明式分区示例等价的分区结构。请按以下步骤操作：

1. 创建“根”表，所有“子”表都从它继承。该表不包含数据。除非希望检查约束同样应用于所有子表，否则不要在此表上定义任何检查约束。在它上面定义索引或唯一约束也没有意义。在本示例中，根表就是最初定义的measurement 表：

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

2. 创建若干“子”表，每个都从根表继承。通常，这些表不会在从根表继承的列集合之外增加任何列。和声明式分区一样，这些表就是普通的PostgreSQL 表（或者外部表）。

```
CREATE TABLE measurement_y2006m02 () INHERITS (measurement);
CREATE TABLE measurement_y2006m03 () INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 () INHERITS (measurement);
CREATE TABLE measurement_y2007m12 () INHERITS (measurement);
CREATE TABLE measurement_y2008m01 () INHERITS (measurement);
```

3. 为子表增加不重叠的表约束来定义每个分区允许的键值。

典型示例如下：

```
CHECK ( x = 1 )
CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
CHECK ( outletID >= 100 AND outletID < 200 )
```

确保约束能够保证不同子表允许的键值之间没有重叠。一个常见的错误是设置如下范围约束：

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

这是错误的，因为无法确定键值 200 属于哪个子表。范围应当采用以下方式定义：

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
) INHERITS (measurement);

...
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
) INHERITS (measurement);
```

4. 对于每个子表，在键列上创建一个索引，以及任何想要的其他索引。

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
```

5. 我们希望应用程序能够执行 `INSERT INTO measurement ...`，并使数据被重定向到适当的子表。可以在根表上附加合适的触发器函数来实现这一点。如果数据只添加到最新的子表，可以使用非常简单的触发器函数：

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

创建函数后，创建一个调用该触发器函数的触发器：

```
CREATE TRIGGER insert_measurement_trigger
    BEFORE INSERT ON measurement
    FOR EACH ROW EXECUTE FUNCTION measurement_insert_trigger();
```

我们必须每月重新定义触发器函数，使它总是向当前子表插入。不过，触发器定义不需要更新。

我们可能希望在插入数据时，由服务器自动确定应将该行添加到哪个子表。可以使用更复杂的触发器函数来实现，例如：

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.logdate >= DATE '2006-02-01' AND
         NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
            NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
            NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range.  Fix the measurement_insert_trigger()
 function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

触发器定义与之前相同。注意，每个`IF`测试必须与其子表的`CHECK`约束完全匹配。

虽然该函数比单月形式更复杂，但并不需要同样频繁地更新，因为可以在需要某个分支之前就把它加入。

> **注意**
>
> 在实践中，如果大部分插入都会进入最新的子表，最好先检查它。为了简洁，我们为触发器的检查采用了和本例中其他部分一致的顺序。

将插入重定向到适当子表的另一种方法，是在根表上设置规则来代替触发器。例如：

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

规则的开销明显高于触发器，但这种开销每个查询只支付一次，而不是每行一次，因此在批量插入时，这种方法可能有优势。不过，在大多数情况下，触发器方法的性能更好。

注意 `COPY` 会忽略规则。如果想要使用 `COPY` 插入数据，则需要拷贝到正确的子表而不是直接放在根表中。`COPY` 会引发触发器，因此在使用触发器方法时可以正常使用它。

规则方法的另一个缺点是，如果规则集合无法覆盖插入日期，则没有简单的方法能够强制产生错误，数据将会无声无息地进入到根表中。

6. 确认 constraint_exclusion 配置参数在 `postgresql.conf` 中没有被禁用，否则可能会不必要地访问子表。

如我们所见，一个复杂的表层次可能需要大量的 DDL。在上面的示例中，我们可能为每个月创建一个新的子表，因此编写一个脚本来自动生成所需要的 DDL 可能会更好。

5.10.3.2. 继承分区的维护

要快速移除旧数据，只需要简单地去掉不再需要的子表：

```
DROP TABLE measurement_y2006m02;
```

要从继承层次表中去掉子表，但还是把它当做一个表保留：

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

要增加一个新子表来处理新数据，可以像上面创建的原始子表那样创建一个空的子表：

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

或者，用户可能想要创建新子表并且在将它加入到表层次之前填充它。这允许在数据对父表上的查询可见之前，先对其进行装载、检查和转换。

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.10.3.3. 注意事项

使用继承实现分区时，有以下注意事项：

- 没有自动的方法验证所有的 CHECK 约束之间是否互斥。编写代码来产生子表以及创建和修改相关对象比手写命令要更加安全。
- 这里展示的方案假定行的键列值永不改变，或者至少不会改变到必须把该行移入另一个分区的程度。由于CHECK 约束的存在，试图那样做的UPDATE 将会失败。如果需要处理这种情况，可以在子表上放置合适的更新触发器，但这会让整个结构的管理复杂得多。
- 如果手动执行 VACUUM 或 ANALYZE 命令，不要忘记需要在每个子表上分别运行。例如，以下命令：

  ```
  ANALYZE measurement;
  ```

  只会处理根表。
- 带有 ON CONFLICT 子句的 INSERT 语句不太可能按照预期工作，因为只有在指定的目标关系而不是其子关系上发生唯一违背时才会采取ON CONFLICT 动作。
- 将会需要触发器或者规则将行路由到想要的子表中，除非应用明确地知道分区方案。编写触发器可能会很复杂，并且会比声明式分区在内部执行的元组路由慢很多。

5.10.4. 分区剪枝

分区剪枝是一种提升声明式分区表性能的查询优化技术。例如：

```
SET enable_partition_pruning = on;                 -- the default
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

如果没有分区剪枝，上面的查询将会扫描measurement 表的每一个分区。如果启用了分区剪枝，规划器会检查每个分区的定义，并证明某个分区不需要扫描，因为它不可能包含满足查询 WHERE 子句的任何行。当规划器能够证明这一点时，就会把该分区从查询计划中排除（剪枝）。

通过使用 EXPLAIN 命令和 enable_partition_pruning 配置参数，可以显示进行了分区剪枝的计划与未剪枝计划的差异。对于这种表结构，典型的未优化计划如下：

```
SET enable_partition_pruning = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=188.76..188.77 rows=1 width=8)
   ->  Append  (cost=0.00..181.05 rows=3085 width=0)
         ->  Seq Scan on measurement_y2006m02  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m11  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2007m12  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

某些或全部分区可能使用索引扫描，而不是全表顺序扫描，但这里的重点是，为回答该查询，根本不需要扫描较旧的分区。启用分区剪枝后，可以得到代价明显更低、结果却相同的计划：

```
SET enable_partition_pruning = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=37.75..37.76 rows=1 width=8)
   ->  Append  (cost=0.00..36.21 rows=617 width=0)
         ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

注意，分区剪枝仅由分区键隐式定义的约束所驱动，而不是由索引的存在驱动。因此，没有必要在键列上定义索引。是否需要为一个给定分区创建索引取决于预期的查询扫描该分区时会扫描大部分还是小部分。索引有助于后一种情况，对前一种情况则没有帮助。

分区剪枝不仅可以在规划查询时进行，也可以在执行期间进行。这很有用，因为当子句中的表达式在查询规划时还无法确定其值时，它可以剪掉更多分区。例如，使用定义在PREPARE 语句中的参数、使用从子查询获得的值，或者在嵌套循环连接的内侧使用参数化值。执行期间的分区剪枝可以在以下任一时机进行：

- 在查询计划的初始化阶段。对于在执行初始化阶段就已知的参数值，可以在这里执行分区剪枝。这个阶段被剪掉的分区不会出现在查询的EXPLAIN 或 EXPLAIN ANALYZE 结果中。通过观察 EXPLAIN 输出中的 “Subplans Removed” 属性，可以确定这个阶段被移除的分区数。
- 在查询计划的实际执行期间。这里可以使用只有在实际查询执行时才能知道的值执行分区剪枝。这包括来自子查询的值以及来自执行时参数的值（例如来自于参数化嵌套循环连接的参数）。由于在查询执行期间这些参数的值可能会改变多次，所以只要分区剪枝使用到的执行参数发生改变，就会执行一次分区剪枝。要判断分区是否在这个阶段被剪枝，需要仔细地观察EXPLAIN ANALYZE 输出中的 loops 属性。对应于不同分区的子计划可以具有不同的值，这取决于在执行期间每个分区被剪枝的次数。如果每次都被剪枝，有些分区可能会显示为 (never executed)。

可以使用 enable_partition_pruning 设置禁用分区剪枝。

> **注意**
>
> 执行时分区剪枝目前只会发生在 Append 节点类型上，而不会发生在MergeAppend 或 ModifyTable 节点上。这很可能在 PostgreSQL 的未来版本中改变。

5.10.5. 分区和约束排除

约束排除是一种与分区剪枝类似的查询优化技术。虽然它主要用于采用传统继承方法实现的分区，但也可以用于其他用途，包括声明式分区。

约束排除以非常类似于分区剪枝的方式工作，不过它使用每个表的CHECK 约束 – 这也是它得名的原因 – 而分区剪枝使用表的分区边界，分区边界仅存在于声明式分区的情况中。另一点不同之处是约束排除仅在规划时应用，在执行时不会尝试移除分区。

约束排除使用 CHECK 约束，这使它比分区剪枝慢，但有时也可以成为一种优势：除了内部分区边界外，声明式分区表也可以定义约束，因此约束排除可能从查询计划中进一步排除分区。

constraint_exclusion 的默认（也是推荐的）设置不是 on 也不是 off，而是一种被称为 partition 的中间设置，这会导致该技术仅被应用于可能工作在继承分区表上的查询。on 设置导致规划器检查所有查询中的 CHECK 约束，甚至是那些不太可能受益的简单查询。

下列提醒适用于约束排除：

- 约束排除只在查询规划期间应用；不像分区剪枝那样，它不会在查询执行期间应用。
- 只有查询的 WHERE 子句包含常量（或者外部提供的参数）时，约束排除才能有效果。例如，针对一个非不可变函数（如CURRENT_TIMESTAMP）的比较不能被优化，因为规划器不知道该函数的值在运行时会落到哪个子表中。
- 保持分区约束尽量简单，否则规划器可能无法证明哪些子表不需要访问。如前面的示例所示，对列表分区使用简单的等值条件，对范围分区使用简单的范围测试。一条很好的经验法则是：分区约束应只包含分区列与常量之间使用 B-树可索引操作符的比较，因为只有 B-树可索引列才允许出现在分区键中。
- 约束排除期间会检查父表的所有子表上的所有约束，因此大量的子表很可能明显地增加查询规划时间。因此，传统的基于继承的分区在子表数量不超过大约一百个时可以很好地工作，不要尝试使用成千上万个子表。

5.10.6. 声明式分区最佳实践

应当谨慎选择如何对表进行分区，因为糟糕的设计会对查询规划和执行性能产生负面影响。

最重要的设计决策之一是选择对数据进行分区的列或者列的组合。通常最佳选择是按最常出现在分区表上执行的查询的 WHERE 子句中的列或列集合进行分区。与分区边界约束兼容的 WHERE 子句可用于剪枝不需要的分区。但是，你可能会被迫根据PRIMARY KEY 或 UNIQUE 约束的要求做出其他决策。在规划分区策略时，删除不需要的数据也是需要考虑的一个因素。整个分区可以相当快地分离出去，因此把分区策略设计成让一次需要删除的所有数据都位于单个分区中，往往是有益的。

选择表应划分成多少个分区，也是一个关键决策。没有足够的分区可能意味着索引仍然太大，数据局部性仍然较差，这可能导致缓存命中率很低。但是，把表分成过多分区也会带来问题。在查询规划和执行期间，分区过多可能意味着规划时间更长、内存消耗更高，下面还会进一步说明。在选择如何分区时，也必须考虑将来可能发生的变化。例如，如果你选择为每个客户建立一个分

区，而当前只有少量大客户，那么就应考虑几年后可能变成拥有大量小客户的情形。在这种情况下，最好选择按HASH 分区并且选择合理数量的分区，而不是尝试按 LIST 进行分区，并希望客户数量的增长不会超出按数据分区的实际范围。

子分区有助于进一步拆分那些预计会比其他分区更大的分区。另一种选择是在分区键中使用多列范围分区。这两种做法都很容易导致分区数量过多，因此应当克制使用。

考虑查询规划和执行期间的分区开销也很重要。查询规划器通常能够较好地处理多达几百个分区的层次结构，前提是典型查询能让规划器剪枝掉除少量分区之外的所有分区。随着添加更多分区，规划时间会变长，内存消耗会更高。对于UPDATE 和 DELETE 命令尤其如此。担心拥有大量分区的另一个原因是，服务器的内存消耗可能会随着时间的推移而显著增加，特别是如果许多会话接触大量分区。这是因为每个分区都需要将其元数据加载到接触它的每个会话的本地内存中。

对于数据仓库类型工作负载，使用比 OLTP 类型工作负载更多的分区可能是合理的。通常，在数据仓库中，查询规划时间相对不那么重要，因为大多数处理时间都花在查询执行期间。对于这两种类型的工作负载，尽早做出正确的决策非常重要，因为重新分区大量数据可能会非常缓慢。模拟预期工作负载通常有利于优化分区策略。永远不要只是假设更多的分区比更少的分区更好，反之亦然。

5.11. 外部数据

PostgreSQL 实现了部分的 SQL/MED 规定，允许我们使用普通 SQL 查询来访问位于 PostgreSQL 之外的数据。这种数据被称为外部数据（注意这种用法不要和外键混淆，后者是数据库中的一种约束）。

外部数据可以借助外部数据包装器来访问。外部数据包装器是一个库，它可以与外部数据源通信，并隐藏连接数据源以及从中获取数据的细节。在contrib 模块中有一些外部数据包装器，参见附录 F。其他类型的外部数据包装器可以在第三方产品中找到。如果这些现有的外部数据包装器都不能满足需要，也可以自行编写，参见第 57 章。

要访问外部数据，我们需要创建一个外部服务器对象，它根据为其提供支持的外部数据包装器所使用的一组选项定义如何连接到某个特定的外部数据源。然后还需要创建一个或多个外部表，用于定义远程数据的结构。外部表在查询中可以像普通表一样使用，但它在 PostgreSQL 服务器中并不存储数据。每次使用外部表时，PostgreSQL 都会要求外部数据包装器从外部数据源获取数据；而在更新命令的场景下，则要求其把数据发送到外部数据源。

访问远程数据可能需要向外部数据源进行认证。这些信息可以通过一个用户映射提供，它会基于当前的PostgreSQL 角色提供附加数据，例如用户名和密码。

更多信息请见 CREATE FOREIGN DATA WRAPPER、CREATE SERVER、CREATE USER MAPPING、CREATE FOREIGN TABLE、以及 IMPORT FOREIGN SCHEMA。

5.12. 其他数据库对象

表是关系数据库结构中的核心对象，因为它们承载着数据。但它们并不是数据库中唯一存在的对象。还可以创建许多其他种类的对象，以便让数据的使用和管理更加高效或方便。本章不会讨论这些对象，但这里给出一个列表，让你了解有哪些可能性：

- 视图
- 函数、过程和操作符
- 数据类型和域
- 触发器和重写规则

这些主题的详细信息请见第 V 部分。

5.13. 依赖跟踪

当我们创建涉及许多带外键约束的表、视图、触发器、函数等对象的复杂数据库结构时，我们隐式地创建了一张对象之间的依赖关系网。例如，具有一个外键约束的表依赖于它所引用的表。

为了保证整个数据库结构的完整性，PostgreSQL 确保我们无法删除仍然被其他对象依赖的对象。例如，尝试删除第 5.3.5 节中的产品表会导致一个如下的错误消息，因为有订单表依赖于产品表：

```
DROP TABLE products;

ERROR:  cannot drop table products because other objects depend on it
DETAIL:  constraint orders_product_no_fkey on table orders depends on table products
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

该错误消息包含了一个有用的提示：如果我们不想一个一个去删除所有的依赖对象，我们可以执行：

```
DROP TABLE products CASCADE;
```

这样所有的依赖对象将被移除，同样依赖于它们的任何对象也会被递归删除。在这种情况下，订单表不会被移除，但是它的外键约束会被移除。之所以在这里会停下，是因为没有什么依赖着外键约束（如果希望检查`DROP ... CASCADE` 会干什么，运行不带`CASCADE` 的 `DROP` 并阅读 `DETAIL` 输出）。

PostgreSQL 中几乎所有 `DROP` 命令都支持指定 `CASCADE`。当然，可能出现的依赖关系形态会随着对象类型不同而变化。你也可以写`RESTRICT` 来代替 `CASCADE`，从而得到默认行为，也就是阻止删除任何仍被其他对象依赖的对象。

> **注意**
>
> 根据 SQL 标准，在`DROP` 命令中指定 `RESTRICT` 或 `CASCADE` 是被要求的。但没有哪个数据库系统真正强制了这个规则，但是不同的系统中两种默认行为都是可能的。

如果一个 `DROP` 命令列出了多个对象，只有在存在指定对象构成的组之外的依赖关系时才需要 `CASCADE`。例如，如果发出命令`DROP TABLE tab1, tab2` 且存在从 `tab2` 到 `tab1` 的外键引用，那么就不需要`CASCADE` 即可成功执行。

对于用户定义函数，PostgreSQL 会跟踪与函数外部可见属性有关的依赖，例如参数和结果类型，但不会跟踪那些只有检查函数体才能得知的依赖。例如，考虑以下情况：

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
                             'green', 'blue', 'purple');

CREATE TABLE my_colors (color rainbow, note text);

CREATE FUNCTION get_color_note (rainbow) RETURNS text AS
  'SELECT note FROM my_colors WHERE color = $1'
  LANGUAGE SQL;
```

（关于 SQL 语言函数的说明，请参阅第 38.5 节。）PostgreSQL 会知道 `get_color_note` 函数依赖于 `rainbow` 类型：删除该类型将迫使系统删除函数，因为它的参数类型将不再有定义。但是，PostgreSQL 不会认为 `get_color_note` 依赖于 `my_colors` 表，因此不会在删除该表时删除函数。这种做法有缺点，也有好处。即使表不存在，函数在某种意义上仍然有效，尽管执行它会报错；创建一个同名新表后，函数就可以重新工作。

第 6 章 数据操纵

前一章讨论了如何创建用于保存数据的表和其他结构。现在该把数据填入表中了。本章介绍如何插入、更新和删除表数据。下一章则终于会讲到如何从数据库中取回那些久寻不得的数据。

6.1. 插入数据

创建表时，其中不包含任何数据。在数据库真正派上用场之前，首先要做的就是插入数据。从概念上说，数据一次插入一行。当然也可以插入多行，但不能插入少于一行的数据。即使只知道部分列值，也必须创建完整的一行。

要创建新行，可使用INSERT 命令。该命令需要提供表名和列值。例如，考虑第 5 章中的 products 表：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

插入一行的示例命令如下：

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

数据值按列在表中出现的顺序列出，并以逗号分隔。通常，数据值会是字面量（常量），但也允许使用标量表达式。

上述语法的缺点在于你必须知道表中列的顺序。为了避免这一点，你也可以显式列出列。例如，下面两条命令的效果都与上面的命令相同：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

许多用户认为，总是列出列名是一种好习惯。

如果你没有所有列的值，可以省略其中一些列。在这种情况下，这些列将填入各自的默认值。例如：

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

第二种形式是 PostgreSQL 的一个扩展。它从左到右用给出的值填充各列，其余列则取默认值。

为求清晰，你也可以显式请求默认值，既可以针对单个列，也可以针对整行：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

你可以在一条命令中插入多行：

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

也可以插入一个查询的结果（它可能返回零行、一行或多行）：

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = 'today';
```

这使你能够利用 SQL 查询机制的全部能力（见第 7 章）来计算要插入的行。

> **提示**
>
> 当需要一次插入大量数据时，可以考虑使用COPY 命令。它不像 INSERT 命令那样灵活，但效率更高。关于如何改善批量装载性能的更多信息，请参阅第 14.4 节。

6.2. 更新数据

修改数据库中已有的数据称为更新。你可以更新单独的行、表中的所有行，或者所有行中的一个子集。每一列都可以单独更新；其他列不受影响。

要更新现有的行，可使用UPDATE 命令。这需要三部分信息：

1. 要更新的表名和列名
2. 列的新值
3. 要更新哪一行（或哪些行）

回想一下第 5 章中的内容，SQL 通常并不为行提供唯一标识符。因此，并不总能直接指定要更新哪一行。取而代之的是，你要指定一行必须满足哪些条件才会被更新。只有当表中存在主键时（不论你是否声明了它），才能通过选择一个匹配主键的条件来可靠地定位单独的行。图形化数据库访问工具正是依赖这一事实来允许你逐行更新。

例如，这条命令把所有价格为 5 的产品的价格更新为 10：

```
UPDATE products SET price = 10 WHERE price = 5;
```

这可能更新零行、一行或多行。尝试执行一个匹配不到任何行的更新并不算错误。

让我们仔细看看这条命令。首先是关键字 UPDATE，后面跟着表名。和平常一样，表名可以用模式限定；否则就在搜索路径中查找。接着是关键字 SET，后面跟着列名、一个等号以及新的列值。新列值可以是任意标量表达式，而不仅仅是常量。例如，如果你想把所有产品的价格提高 10%，可以使用：

```
UPDATE products SET price = price * 1.10;
```

如你所见，用于新值的表达式也可以引用该行中的现有值。我们还省略了WHERE 子句。如果省略它，就意味着更新表中的所有行。如果给出了它，则只有匹配 WHERE 条件的那些行才会被更新。请注意，SET 子句中的等号表示赋值，而 WHERE 子句中的等号表示比较，但这不会造成歧义。当然，WHERE 条件不必是等值测试。还有许多其他操作符可用（见第 9 章）。不过，该表达式必须计算出一个布尔结果。

你可以在一条 UPDATE 命令中更新多个列，只需在 SET 子句中列出多个赋值。例如：

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. 删除数据

到目前为止，我们已经解释了如何向表中添加数据以及如何修改数据。接下来要讨论的是如何删除不再需要的数据。正如添加数据只能按整行进行一样，从表中删除数据也只能按整行进行。前一节已经说明，SQL 不提供直接寻址单独行的方法。因此，只能通过指定待删除行必须满足的条件来删除它们。如果表中有主键，那么你可以指定精确的一行；但你也可以删除匹配某个条件的一组行，或者一次删除表中的所有行。

可以使用 DELETE 命令删除行；其语法与 UPDATE 命令非常相似。例如，要删除 products 表中所有价格为 10 的行，可以使用：

```
DELETE FROM products WHERE price = 10;
```

如果你只是写：

```
DELETE FROM products;
```

那么表中的所有行都会被删除！程序员须当心。

6.4. 从被修改的行中返回数据

有时候，在修改行的同时取得这些行中的数据会很有用。INSERT、UPDATE、DELETE 命令都支持可选的 RETURNING 子句。使用 RETURNING 可以避免额外执行一次数据库查询来收集这些数据；当原本难以可靠地识别被修改的行时，这一点尤其有价值。

RETURNING 子句允许的内容与 SELECT 命令的输出列表相同（见第 7.3 节）。它可以包含命令目标表的列名，也可以包含使用这些列的值表达式。一个常见的简写是 RETURNING *，它按顺序选择目标表的全部列。

在 INSERT 中，RETURNING 可用的数据是插入后的行。对于简单的插入，这并没有太大用处，因为它只是重复客户端提供的数据。但在依赖计算得出的默认值时，它就会非常方便。例如，当使用 serial 列提供唯一标识符时，RETURNING 可以返回分配给新行的 ID：

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);

INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;
```

RETURNING 子句配合 INSERT ... SELECT 也非常有用。

在 UPDATE 中，RETURNING 可用的数据是被修改行的新内容。例如：

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

在 DELETE 中，RETURNING 可用的数据是被删除行的内容。例如：

```
DELETE FROM products
  WHERE obsoletion_date = 'today'
  RETURNING *;
```

如果目标表上有触发器（第 39 章），那么 RETURNING 可用的数据就是经触发器修改后的行。因此，查看由触发器计算出的列，也是 RETURNING 的另一个常见用例。

第 7 章 查询

前面的章节解释了如何创建表、如何用数据填充它们，以及如何操纵这些数据。现在我们终于可以讨论如何从数据库中检索数据了。

7.1. 概述

从数据库检索数据的过程或用于检索数据的命令称为查询。在 SQL 中，SELECT 命令用于指定查询。SELECT 命令的一般语法为：

```
[WITH with_queries] SELECT select_list FROM table_expression [sort_specification]
```

以下各节将详细描述选择列表、表表达式和排序说明。WITH 查询是一项高级特性，因此放在最后介绍。

一种简单查询的形式如下：

```
SELECT * FROM table1;
```

假设存在一个名为 table1 的表，则这条命令会从table1 中检索所有行以及所有用户定义列。（具体的检索方式取决于客户端应用。例如，psql 程序会在屏幕上显示一个 ASCII 形式的表格，而客户端库则会提供从查询结果中提取单个值的函数。）选择列表说明* 表示表表达式所提供的全部列。选择列表也可以只选择部分可用列，或者利用这些列进行计算。例如，如果table1 有名为 a、b 和 c 的列（还可能有其他列），那么可以写出下面的查询：

```
SELECT a, b + c FROM table1;
```

（假设 b 和 c 都是数值数据类型）。有关更多细节请参见第 7.3 节。

FROM table1 是一种简单的表表达式：它只读取一个表。通常，表表达式可以是由基本表、连接和子查询组成的复杂结构。不过，你也可以完全省略表表达式，把SELECT 命令当成计算器来使用：

```
SELECT 3 * 4;
```

如果选择列表中的表达式会返回变化的结果，这种用法就更有用了。例如，你可以这样调用函数：

```
SELECT random();
```

7.2. 表表达式

表表达式用于计算出一个表。表表达式包含一个 FROM 子句，后面可以根据需要跟上WHERE、GROUP BY 和 HAVING 子句。最简单的表表达式只是引用磁盘上的一个表，即所谓的基本表；但也可以使用更复杂的表达式，以多种方式修改或组合基本表。

表表达式中可选的 WHERE、GROUP BY 和 HAVING 子句指定了一个连续转换的流水线，这些转换作用于由FROM 子句派生出的表。所有这些转换都会生成一个虚拟表，该表提供的各行会传递给选择列表，以计算查询的输出行。

7.2.1. FROM 子句

此 FROM 子句从逗号分隔的表引用列表所给出的一个或多个其他表推导出一个表。

```
FROM table_reference [, table_reference [, ...]]
```

表引用可以是表名（可能带模式限定），也可以是子查询、JOIN 结构等派生表，或它们的复杂组合。如果FROM 子句中列出了多个表引用，这些表会进行交叉连接（即形成各表行的笛卡尔积，见下文）。FROM 列表的结果是一个中间虚拟表，随后可以用WHERE，GROUP BY 和 HAVING 子句对它进行变换，最终得到整个表表达式的结果。

当表引用命名的是表继承层次中的父表时，除非在表名前加上关键字ONLY，否则该表引用不仅会产生该表中的行，还会产生其所有后代表中的行。不过，这种引用只会产生该命名表中出现的列 – 子表中新增的列会被忽略。

也可以不在表名前写 ONLY，而是在表名后面写上*，显式指定要包含后代表。如今继续使用这种语法并没有实际理由，因为包含后代表现在本来就是默认行为。不过，为了兼容旧版本，仍然支持这种语法。

7.2.1.1. 连接表

一个连接表是根据特定的连接类型的规则从两个其它表（真实表或派生表）中派生的表。目前支持内连接、外连接和交叉连接。一个连接表的一般语法是：

```
T1 join_type T2 [ join_condition ]
```

所有类型的连接都可以被链在一起或者嵌套：*T1* 和 *T2* 都可以是连接表。在 JOIN 子句周围可以使用圆括号来控制连接顺序。如果不使用圆括号，JOIN 子句会从左至右嵌套。

连接类型

交叉连接

```
T1 CROSS JOIN T2
```

对来自于 *T1* 和 *T2* 的行的每一种可能的组合（即笛卡尔积），连接表将包含这样一行：它由所有*T1* 里面的列后面跟着所有 *T2* 里面的列构成。如果两个表分别有 N 和 M 行，连接表将有 N * M 行。

FROM *T1* CROSS JOIN *T2* 等效于 FROM *T1* INNER JOIN *T2* ON TRUE（见下文）。它也等效于 FROM *T1*,*T2*。

注意

当出现两个以上的表时，后一种等价关系并不严格成立，因为JOIN 的绑定强于逗号。例如 FROM *T1* CROSS JOIN *T2* INNER JOIN *T3* ON *condition* 和 FROM *T1*,*T2* INNER JOIN *T3* ON *condition* 并不完全相同，因为第一种情况中的 *condition* 可以引用 *T1*，而第二种情况中则不行。

限定连接

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

INNER 和 OUTER 在所有形式中都是可选的。INNER 是默认值；LEFT、RIGHT 和 FULL 表示外连接。

连接条件在 ON 或 USING 子句中指定，或者用关键字NATURAL 隐含地指定。连接条件决定来自两个源表中的哪些行是“匹配”的，这些我们将在后文详细解释。

限定连接有以下几种类型：

INNER JOIN

对于 T1 的每一行 R1，生成的连接表都有一行对应 T2 中的每一个满足和 R1 的连接条件的行。

LEFT OUTER JOIN

首先，执行一次内连接。然后，为 T1 中每一个无法在连接条件上匹配 T2 里任何一行的行返回一个连接行，该连接行中 T2 的列用空值补齐。因此，生成的连接表里为来自 T1 的每一行都至少包含一行。

RIGHT OUTER JOIN

首先，执行一次内连接。然后，为 T2 中每一个无法在连接条件上匹配 T1 里任何一行的行返回一个连接行，该连接行中 T1 的列用空值补齐。这与左连接相反：生成的连接表里为来自 T2 的每一行都至少包含一行。

FULL OUTER JOIN

首先，执行一次内连接。然后，为 T1 中每一个无法在连接条件上匹配 T2 里任何一行的行返回一个连接行，该连接行中 T2 的列用空值补齐。同样，为 T2 中每一个无法在连接条件上匹配 T1 里任何一行的行返回一个连接行，该连接行中 T1 的列用空值补齐。

ON 子句是最通用的连接条件形式：它接收一个和 WHERE 子句里用的一样的布尔值表达式。如果两个分别来自 *T1* 和 *T2* 的行在 ON 表达式上运算的结果为真，那么它们就算是匹配的行。

USING 是个缩写符号，它允许你利用特殊的情况：连接的两端都具有相同的连接列名。它接受共享列名的一个逗号分隔列表，并且为其中每一个共享列构造一个包含等值比较的连接条件。例如用USING (a, b) 连接 *T1* 和 *T2* 会产生连接条件 ON *T1*.a = *T2*.a AND *T1*.b = *T2*.b。

更进一步，JOIN USING 的输出会废除冗余列：不需要把匹配上的列都打印出来，因为它们必须具有相等的值。不过JOIN ON 会先产生来自 *T1* 的所有列，后面跟上所有来自*T2* 的列；而 JOIN USING 会先按列出的顺序为每一个列对产生一个输出列，然后先跟上来自*T1* 的剩余列，最后跟上来自*T2* 的剩余列。

最后，NATURAL 是 USING 的一种缩写形式：它会形成一个 USING 列表，其中包含两个输入表中共同出现的所有列名。和USING 一样，这些列在输出表中只出现一次。如果没有公共列名，NATURAL JOIN 的行为就与 JOIN ... ON TRUE 相同，产生一个叉积连接。

> **注意**
>
> 对于被连接的关系发生列变化的情况，USING 相当安全，因为只有列出的列才会被合并。NATURAL 的风险则大得多，因为只要任一关系的模式变更导致出现新的同名列，连接就会把这个新列也纳入合并。

综合起来看，假设我们有表t1：

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

和 t2：

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

然后我们用不同的连接方式可以获得各种结果：

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
```

```
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

用 ON 指定的连接条件也可以包含与连接不直接相关的条件。这种功能可能对某些查询很有用，但是需要我们仔细想清楚。例如：

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

注意把限制放在 WHERE 子句中会产生不同的结果：

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

这是因为放在 ON 子句中的约束会在连接之前处理，而放在WHERE 子句中的约束则会在连接之后处理。这对内连接无关紧要，但对外连接影响很大。

7.2.1.2. 表和列别名

你可以给表以及复杂的表引用指定一个临时名字，以便在查询的其余部分引用该派生表。这被称为表别名。

要创建一个表别名，我们可以写：

```
FROM table_reference AS alias
```

或者

```
FROM table_reference alias
```

AS 关键字只是可选的语法噪音。*alias* 可以是任意标识符。

表别名的典型应用是给长表名赋予比较短的标识符，好让连接子句更易读。例如：

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num;
```

对于当前查询而言，别名成为该表引用的新名称–不允许在查询的其他地方通过原名称引用该表。因此，以下写法无效：

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;    -- 错误
```

表别名主要用于简化符号，但是当把一个表连接到它自身时必须使用别名，例如：

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
```

此外，当表引用是子查询时也必须使用别名（参见第 7.2.1.3 节）。

圆括弧用于解决歧义。在下面的示例中，第一个语句将把别名b 赋给 my_table 的第二个实例，但是第二个语句把别名赋给连接的结果：

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

另外一种给表指定别名的形式是给表的列赋予临时名字，就像给表本身指定别名一样：

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

如果指定的列别名比表里实际的列少，那么剩下的列就没有被重命名。这种语法对于自连接或子查询特别有用。

如果用这些形式中的任何一种给一个 JOIN 子句的输出附加了一个别名，那么该别名就会隐藏 JOIN 内部的原始名称。例如：

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

是合法 SQL，但是：

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

是不合法的：表别名 a 在别名 c 外面是看不到的。

7.2.1.3. 子查询

指定派生表的子查询必须用圆括号括起来，并且必须赋予一个表别名（参见第 7.2.1.2 节）。例如：

```
FROM (SELECT * FROM table1) AS alias_name
```

这个示例等效于 FROM table1 AS alias_name。更有趣的情况是在子查询里面有分组或聚合的时候，子查询不能被简化为一个简单的连接。

一个子查询也可以是一个 VALUES 列表：

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS names(first, last)
```

同样，表别名是必须的。为VALUES 列表中的列分配别名是可选的，但这是一个好习惯。更多信息可参见第 7.7 节。

7.2.1.4. 表函数

表函数是那些生成行集合的函数，这些行可以由基础类型（标量类型）组成，也可以由复合数据类型（表行）组成。它们在查询的FROM 子句中用法类似于表、视图或子查询。表函数返回的列可以像表、视图或子查询的列一样，出现在SELECT、JOIN 或 WHERE 子句中。

也可以使用 ROWS FROM 语法把多个表函数组合起来，并列返回各函数的结果列；这种情况下，结果行数等于返回行数最多的那个函数的结果行数，较小的结果会用空值填充到相同长度。

```
function_call [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ])]]
ROWS FROM( function_call [, ... ] ) [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ...
 ])]]
```

如果指定了 WITH ORDINALITY 子句，一个额外的 bigint 类型的列将会被增加到函数的结果列中。这个列对函数结果集的行进行编号，编号从 1 开始（这是对 SQL 标准语法 UNNEST ... WITH ORDINALITY 的一般化）。默认情况下，序数列被称为ordinality，但也可以通过使用一个 AS 子句给它分配一个不同的列名。

调用特殊的表函数 UNNEST 可以使用任意数量的数组参数，它会返回对应的列数，就好像在每一个参数上单独调用 UNNEST（第 9.18 节）并且使用 ROWS FROM 结构把它们组合起来。

```
UNNEST( array_expression [, ... ] ) [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ...
 ])]]
```

如果没有指定 *table_alias*，该函数名将被用作表名。在ROWS FROM() 结构的情况中，会使用第一个函数名。

如果没有提供列的别名，那么对于一个返回基础类型的函数，列名也与该函数名相同。对于一个返回复合类型的函数，结果列会从该类型的属性得到名称。

示例：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                      );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

有时候，定义一个能够根据调用方式返回不同列集合的表函数很有用。为支持这一点，可以将不带OUT 参数的表函数声明为返回伪类型 record。当此类函数在查询中使用时，必须在查询本身中指定预期的行结构，这样系统才能知道如何解析和规划该查询。这种语法如下：

```
function_call [AS] alias (column_definition [, ... ])
function_call AS [alias] (column_definition [, ... ])
ROWS FROM( ... function_call AS (column_definition [, ... ]) [, ... ] )
```

在不使用 ROWS FROM() 语法时，*column_definition* 列表会取代原本可附加到 FROM 项上的列别名列表，列定义中的名称就起到列别名的作用。在使用ROWS FROM() 语法时，可以为每一个成员函数单独附着一个 *column_definition* 列表；或者在只有一个成员函数并且没有

WITH ORDINALITY 子句的情况下，可以在 ROWS FROM() 后面写一个 *column_definition* 列表来取代一个列别名列表。

考虑下面的示例：

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

dblink 函数（dblink 模块的一部分）执行远程查询。它被声明为返回 record，因为它可能用于任意类型的查询。实际的列集必须在调用它的查询中指定，这样解析器才知道像* 这样的写法应当扩展成什么。

此示例使用 ROWS FROM：

```
SELECT *
FROM ROWS FROM
    (
        json_to_recordset('[{"a":40,"b":"foo"},{"a":"100","b":"bar"}]')
            AS (a INTEGER, b TEXT),
        generate_series(1, 3)
    ) AS x (p, q, s)
ORDER BY p;

  p  |  q  | s
-----+-----+---
  40 | foo | 1
 100 | bar | 2
     |     | 3
```

它把两个函数组合成一个 FROM 目标。json_to_recordset() 被指定返回两列，第一列为 integer，第二列为text。generate_series() 的结果则直接使用。ORDER BY 子句会把列值按整数排序。

7.2.1.5. LATERAL 子查询

可以在出现于 FROM 中的子查询前放置关键词 LATERAL。这允许它们引用前面的 FROM 项提供的列（如果没有 LATERAL，每一个子查询将被独立计算，并且因此不能交叉引用其他FROM 项）。

出现在 FROM 中的表函数的前面也可以被放上关键词 LATERAL，但对于函数该关键词是可选的，在任何情况下函数的参数都可以包含对前面的FROM 项提供的列的引用。

LATERAL 项可以出现在 FROM 列表的顶层，也可以出现在JOIN 树中。在后一种情况下，如果它位于某个JOIN 的右侧，也可以引用该连接左侧的任何项。

当 FROM 项包含 LATERAL 交叉引用时，求值过程如下：对于提供被引用列的FROM 项中的每一行，或者对于多个FROM 项共同提供这些列时对应的每一组行，都会用该行或该组行中的列值来计算LATERAL 项。得到的结果行再像通常那样与生成它们的行做连接。对于源表中的每一行或每一组行，这一过程都会重复。

LATERAL 的一个简单示例：

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

这不是非常有用，因为它和一种更简单的形式得到的结果完全一样：

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

在必须要使用交叉引用列来计算那些即将要被连接的行时，LATERAL 是最有用的。一种常用的应用是为一个返回集合的函数提供一个参数值。例如，假设vertices(polygon) 返回一个多边形的顶点集合，我们可以这样标识存储在一个表中的多边形中靠近的顶点：

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

这个查询也可以被写成：

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

或者写成其他几种等价的形式。（正如前面提到的，这个示例中的LATERAL 关键字并非必需，但这里写出来是为了更清楚。）

把 LEFT JOIN 用于 LATERAL 子查询往往特别方便，这样即使LATERAL 子查询没有为源行产生任何结果行，源行也仍会出现在结果中。例如，如果get_product_names() 返回某个制造商生产的产品名称，而我们表中的某些制造商当前没有生产任何产品，就可以像下面这样找出它们：

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true
WHERE pname IS NULL;
```

7.2.2. WHERE 子句

此 WHERE 子句的语法是：

```
WHERE search_condition
```

其中，*search_condition* 可以是任意值表达式（见第 4.2 节），只要其返回值的类型为boolean。

在完成 FROM 子句的处理之后，生成的虚拟表中的每一行都会根据搜索条件进行检查。如果条件结果为真，该行就会保留在输出表中；否则（也就是说，如果结果为假或空）就会被丢弃。搜索条件通常至少会引用FROM 子句生成的表中的某一列；这并非必须，但若不引用这些列，WHERE 子句通常就没什么用了。

注意

内连接的连接条件既可以写在 WHERE 子句也可以写在 JOIN 子句里。例如，这些表表达式是等效的：

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

和：

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

或者可能还有：

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

选择其中哪一种主要只是风格问题。FROM 子句中的 JOIN 语法虽然属于 SQL 标准，但在其他 SQL 数据库管理系统中可能没有那么容易移植。对于外连接则没有选择：它们必须放在FROM 子句中完成。外连接的 ON 或 USING 子句与 WHERE 条件并不等价，因为它除了会在最终结果中移除行之外，还会为不匹配的输入行增加结果行。

这里是一些 WHERE 子句的示例：

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

在上面的示例中，fdt 是从 FROM 子句派生出来的表。不符合 WHERE 子句搜索条件的行会从 fdt 中被剔除。请注意，这里把标量子查询当作值表达式来使用。和其他查询一样，子查询中也可以使用复杂的表表达式。同时还要注意，子查询中也引用了fdt。只有当 c1 同时也是子查询输入表的某个列名时，才必须把它写成fdt.c1。不过，即使不是必须的，限定列名通常也能使语句更清晰。这个示例展示了外层查询的列名作用域如何扩展到其内层查询中。

7.2.3. GROUP BY 和 HAVING 子句

通过 WHERE 过滤之后，派生出来的输入表还可以使用GROUP BY 子句进行分组，并通过HAVING 子句排除某些分组行。

```
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference [, grouping_column_reference]...
```

此 GROUP BY 子句子句用于把表中在所列出的列上具有相同值的行归为一组。列出的顺序无关紧要。这样做的效果是把每组具有共同值的行合并成一个组行，用来代表该组中的所有行。这既可以消除输出中的冗余，也可以计算适用于这些组的聚合。例如：

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

在第二个查询里，我们不能写成SELECT * FROM test1 GROUP BY x，因为列y 没有一个单一值可以与每个分组对应。被分组的列可以在选择列表中引用是因为它们在每个组都有单一的值。

通常，如果一个表被分了组，那么没有在GROUP BY 中列出的列都不能被引用，除非在聚合表达式中被引用。一个用聚合表达式的示例是：

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

这里的 sum 是一个聚合函数，它在整个组上计算出一个单一值。有关可用的聚合函数的更多信息可以在第 9.20 节中找到。

> **提示**
>
> 不带聚合表达式的分组实际上是在计算某一列的非重复值集合。这也可以用 DISTINCT 子句实现（参阅第 7.3.3 节）。

这里是另外一个示例：它计算每种产品的总销售额（而不是所有产品的总销售额）：

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

在这个示例里，列product_id、p.name 和 p.price 必须在 GROUP BY 子句里，因为它们都在查询的选择列表里被引用到（但见下文）。列s.units 不必在 GROUP BY 列表里，因为它只是在一个聚合表达式（sum(...)）里使用，它代表某种产品的销量。对于每种产品，这个查询都返回一行关于该产品所有销售额的汇总。

如果 products 表的设计使得 product_id 是主键，那么在上面的示例中只按product_id 分组就足够了，因为名称和价格都函数依赖于产品 ID，因此对于每个产品 ID 分组该返回哪个名称和价格值不会有歧义。

在严格的 SQL 里，GROUP BY 只能对源表的列进行分组，但PostgreSQL 把这个扩展为也允许 GROUP BY 去根据选择列表中的列分组。也允许对值表达式进行分组，而不仅是简单的列名。

如果一个表已经用 GROUP BY 子句分了组，而你只对其中某些组感兴趣，那么就可以使用 HAVING 子句。它很像 WHERE 子句，用于从结果中删除某些组。其语法是：

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

在 HAVING 子句中的表达式可以引用分组的表达式和未分组的表达式（后者必须涉及一个聚合函数）。

示例：

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

再次，一个更现实的示例：

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

在上面的示例里，WHERE 子句通过一个未分组的列来选择数据行（该表达式仅对最近四周内发生的销售为真），而HAVING 子句则把输出限制为总销售收入超过 5000 的组。请注意，聚合表达式不必在查询的所有部分都完全相同。

如果一个查询包含聚合函数调用，但没有GROUP BY 子句，分组仍然会发生：结果是一个单一组行（如果该单一组行随后又被HAVING 消除，则可能根本没有结果行）。如果查询包含HAVING 子句，即使没有任何聚合函数调用或GROUP BY 子句，情况也是如此。

7.2.4. GROUPING SETS、CUBE 和 ROLLUP

使用分组集的概念可以实现比上述更加复杂的分组操作。由 FROM 和 WHERE 子句选出的数据被按照每一个指定的分组集单独分组，按照简单GROUP BY 子句对每一个分组计算聚合，然后返回结果。例如：

```
=> SELECT * FROM items_sold;
 brand | size | sales
-------+------+-------
 Foo   | L    |  10
 Foo   | M    |  20
 Bar   | M    |  15
 Bar   | L    |  5
(4 rows)

=> SELECT brand, size, sum(sales) FROM items_sold GROUP BY GROUPING SETS ((brand), (size), ());
 brand | size | sum
-------+------+-----
 Foo   |      |  30
 Bar   |      |  20
       | L    |  15
       | M    |  35
       |      |  50
(5 rows)
```

GROUPING SETS 的每一个子列表可以指定零个或者多个列或者表达式，它们将按照直接出现在GROUP BY 子句中同样的方式被解释。一个空的分组集表示所有的行都要被聚合到一个单一分组（即使没有输入行存在也会被输出）中，这就像前面所说的没有GROUP BY 子句的聚合函数的情况一样。

对于分组列或表达式没有出现在其中的分组集的结果行，对分组列或表达式的引用会被空值所替代。要区分一个特定的输出行来自于哪个分组，请见表 9.56。

PostgreSQL 中提供了一种简化方法来指定两种常用类型的分组集。下面形式的子句

```
ROLLUP ( e1, e2, e3, ... )
```

表示给定的表达式列表及其所有前缀（包括空列表），因此它等效于

```
GROUPING SETS (
    ( e1, e2, e3, ... ),
    ...
    ( e1, e2 ),
    ( e1 ),
    ( )
)
```

这通常被用来分析层次数据，例如按部门、事业部和全公司汇总工资。

下面形式的子句

```
CUBE ( e1, e2, ... )
```

表示给定的列表及其可能的子集（即幂集）。因此

```
CUBE ( a, b, c )
```

等效于

```
GROUPING SETS (
    ( a, b, c ),
    ( a, b    ),
    ( a,    c ),
    ( a       ),
    (    b, c ),
    (    b    ),
    (       c ),
    (         )
)
```

CUBE 或 ROLLUP 子句中的元素可以是表达式或者圆括号中的元素子列表。在后一种情况下，为了生成各个分组集，这些子列表会被当作单一单元处理。例如：

```
CUBE ( (a, b), (c, d) )
```

等效于

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b       ),
    (       c, d ),
    (            )
)
```

并且

```
ROLLUP ( a, (b, c), d )
```

等效于

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c    ),
    ( a          ),
    (            )
)
```

CUBE 和 ROLLUP 可以被直接用在 GROUP BY 子句中，也可以被嵌套在一个 GROUPING SETS 子句中。如果一个 GROUPING SETS 子句被嵌套在另一个同类子句中，效果和把内层子句的所有元素直接写在外层子句中一样。

如果在一个 GROUP BY 子句中指定了多个分组项，那么最终的分组集列表是这些项的笛卡尔积。例如：

```
GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))
```

等效于

```
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),    (a, b, e),
    (a, c, d),    (a, c, e),
    (a, d),       (a, e)
)
```

> **注意**
>
> 在表达式中，结构(a, b)通常会被识别为一个行构造器。在GROUP BY子句中，这不会在表达式的顶层应用，并且(a, b)会按照上面所说的被解析为一个表达式的列表。如果出于某种原因你在分组表达式中需要一个行构造器，请使用ROW(a, b)。

7.2.5. 窗口函数处理

如果查询包含任何窗口函数（见第 3.5 节、第 9.21 节和第 4.2.8 节），这些函数将在任何分组、聚合和HAVING过滤被执行之后被计算。也就是说如果查询使用了任何聚合、GROUP BY或HAVING，则窗口函数看到的行是分组行而不是来自于FROM/WHERE的原始表行。

当使用多个窗口函数时，窗口定义中具有语法等价的PARTITION BY和ORDER BY子句的所有窗口函数，都保证在数据的一次遍历中求值。因此，即使ORDER BY不能唯一确定排序顺序，它们看到的排序顺序也相同。不过，对于具有不同PARTITION BY或ORDER BY定义的函数，其求值不作保证。（在这种情况下，窗口函数的各次求值遍历之间通常需要排序步骤，而排序不保证保留那些在其ORDER BY看来等价的行的顺序。）

目前，窗口函数总是要求输入数据预先排好序，因此查询输出总会按照某个窗口函数的PARTITION BY/ORDER BY子句排序。不过，不建议依赖这一点。如果你希望确保结果按特定方式排序，请显式使用顶层的ORDER BY子句。

7.3. 选择列表

如前面几节所示，SELECT命令中的表表达式会构造出一个中间虚拟表，其中可能涉及组合表和视图、消除行、分组等操作。这个表随后交由选择列表处理。选择列表决定中间表中的哪些列会实际输出。

7.3.1. 选择列表项

最简单的选择列表是*，它会输出表表达式生成的所有列。否则，选择列表就是一个逗号分隔的值表达式列表（定义见第 4.2 节）。例如，它可以是一个列名列表：

```
SELECT a, b, c FROM ...
```

列名a、b和c要么是FROM子句中所引用表的实际列名，要么是像第 7.2.1.2 节所述为它们指定的别名。选择列表中可用的名字空间与WHERE子句相同；如果使用了分组，则与HAVING子句相同。

如果超过一个表有同样的列名，那么你还必须给出表名字，如：

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

在使用多个表时，要求一个特定表的所有列也是有用的：

```
SELECT tbl1.*, tbl2.a FROM ...
```

更多有关 *table_name*.* 记号的内容请参考第 8.16.5 节。

如果在选择列表中使用任意值表达式，那么从概念上说，它会在返回表中增加一个新的虚拟列。该值表达式会对每个结果行计算一次，并用该行中的值替换其中的列引用。不过，选择列表中的表达式并不一定要引用FROM 子句表表达式中的列；例如，它也可以是任意常量算术表达式。

7.3.2. 列标签

选择列表中的项可以被赋予名字，以供后续处理使用，例如在ORDER BY 子句中引用，或者供客户端应用显示。例如：

```
SELECT a AS value, b + c AS sum FROM ...
```

如果没有使用 AS 指定输出列名，系统就会分配一个默认列名。对于简单的列引用，它就是被引用列的名字；对于函数调用，它就是函数的名字；对于复杂表达式，系统则会生成一个通用名称。

此 AS 关键字是可选的，但前提是新列名不能与任何PostgreSQL 关键字相同（见附录 C）。为了避免意外使用关键字，可以用双引号括住列名。例如，VALUE 是关键字，因此下面的写法无效：

```
SELECT a value, b + c AS sum FROM ...
```

但下面的写法有效：

```
SELECT a "value", b + c AS sum FROM ...
```

为防范将来可能新增的关键字，建议始终写出AS，或者用双引号括住输出列名。

> **注意**
>
> 输出列的命名和在 FROM 子句里的命名是不一样的（参阅第 7.2.1.2 节）。它实际上允许你对同一个列命名两次，但是在选择列表中分配的名字是要传递下去的名字。

7.3.3. DISTINCT

在处理完选择列表之后，结果表还可以选择性地删除重复行。我们可以直接在SELECT 后面写上DISTINCT 关键字来指定：

```
SELECT DISTINCT select_list ...
```

（如果不用 DISTINCT，则可以用ALL 关键字来显式指定保留所有行这一默认行为。）

显然，如果两行至少在一个列值上不同，就认为它们是不同的。空值在这种比较中被认为是相等的。

另外，我们还可以用任意表达式来判断什么行可以被认为是可区分的：

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

这里的 *expression* 是一个对所有行求值的任意值表达式。如果某一组行在这些表达式上的值都相等，那么它们就会被视为重复行，因此只保留该组中的第一行。请注意，某一组里的“第一行”是不可预测的，除非查询在足够多的列上进行了排序，以保证到达DISTINCT 过滤器的行顺序是唯一的。（DISTINCT ON 的处理发生在 ORDER BY 排序之后。）

DISTINCT ON 子句不是 SQL 标准的一部分，有时会被认为风格不佳，因为它的结果可能具有不确定性。通过审慎使用GROUP BY 以及 FROM 中的子查询，可以避免使用这种结构，但它往往是最方便的替代方案。

7.4. 组合查询

两个查询的结果可以用集合操作并、交、差进行组合。语法是

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

query1 和 *query2* 都是可以使用以上所有特性的查询。

UNION 有效地把 *query2* 的结果附加到 *query1* 的结果上（不过我们不能保证这就是这些行实际被返回的顺序）。此外，它将删除结果中所有重复的行，就像DISTINCT 做的那样，除非你使用了UNION ALL。

INTERSECT 返回那些同时存在于 *query1* 和 *query2* 的结果中的行，除非声明了 INTERSECT ALL，否则所有重复行都被消除。

EXCEPT 返回所有在 *query1* 结果中但不在 *query2* 结果中的行。（这有时被称为两个查询的差。）同样，除非使用了EXCEPT ALL，否则重复行都会被消除。

为了计算两个查询的并、交、差，这两个查询必须是“并操作兼容的”，也就意味着它们都返回同样数量的列，并且对应的列有兼容的数据类型，如第 10.5 节中描述的那样。

集合操作可以组合使用，例如：

```
query1 UNION query2 EXCEPT query3
```

等价于：

```
(query1 UNION query2) EXCEPT query3
```

如这里所示，你可以使用括号控制求值顺序。如果没有括号，UNION 和 EXCEPT 按从左到右结合，但INTERSECT 的绑定强于这两个操作符。因此

```
query1 UNION query2 INTERSECT query3
```

意思是

```
query1 UNION (query2 INTERSECT query3)
```

你也可以用括号包裹一个单独的 *query*。如果 *query* 需要使用后续部分中讨论的任何子句（例如 LIMIT），那么这一点非常重要。没有括号，要么会出现语法错误，要么该子句会被理解为适用于集合操作的输出，而不是它的输入之一。例如，

```
SELECT a FROM b UNION SELECT x FROM y LIMIT 10
```

是可以接受的，但它的意思是

```
(SELECT a FROM b UNION SELECT x FROM y) LIMIT 10
```

不是

```
SELECT a FROM b UNION (SELECT x FROM y LIMIT 10)
```

7.5. 行排序

当一个查询生成输出表之后（也就是处理完选择列表之后），还可以选择对其排序。如果没有选择排序，行将按未指定的顺序返回。这时的实际顺序取决于扫描和连接计划类型以及磁盘上的物理顺序，但绝不能依赖这些因素。只有显式选择排序步骤，才能保证特定的输出顺序。

ORDER BY 子句指定了排序顺序：

```
SELECT select_list
    FROM table_expression
    ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
             [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

排序表达式可以是任何在查询选择列表中合法的表达式。例如：

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

当指定了多个表达式时，后面的值将用于对那些在前面值上相等的行继续排序。每个表达式后面都可以选择性地写上ASC 或 DESC 关键字，以指定升序或降序排序方向。ASC 是默认值。升序会把较小的值排在前面，而“较小”是由<操作符定义的。类似地，降序则由> 操作符定义。[1]

NULLS FIRST 和 NULLS LAST 选项可用于决定空值在排序顺序中是位于非空值之前还是之后。默认情况下，空值排序时被认为大于任何非空值；也就是说，在DESC 顺序下默认是NULLS FIRST，否则默认是NULLS LAST。

注意顺序选项是对每一个排序列独立考虑的。例如 ORDER BY x, y DESC 表示 ORDER BY x ASC, y DESC，而和ORDER BY x DESC, y DESC 不同。

一个 *sort_expression* 也可以是输出列的列标签或编号，例如：

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

这两条查询都按第一个输出列排序。注意，输出列名必须单独出现，不能用于表达式中–例如，下面的写法是不正确的：

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          -- 错误
```

这一限制是为了减少歧义。如果 ORDER BY 项是一个简单名称，既可能匹配输出列名，也可能匹配表表达式中的列，那么仍会有歧义。这种情况下使用输出列。只有在使用AS 将输出列重命名为其他表列的名称时，才会造成这种混淆。

ORDER BY 可以被应用于 UNION、INTERSECT 或 EXCEPT 组合的结果，但是在这种情况中它只被允许根据输出列名或编号排序，而不能根据表达式排序。

7.6. LIMIT 和 OFFSET

LIMIT 和 OFFSET 允许你只检索查询剩余部分产生的行的一部分：

```
SELECT select_list
    FROM table_expression
    [ ORDER BY ... ]
    [ LIMIT { number | ALL } ] [ OFFSET number ]
```

[1] 事实上，PostgreSQL 会使用该表达式数据类型的默认 B-树操作符类来决定 ASC 和 DESC 的排序顺序。按惯例，数据类型会被设置成由<和> 操作符来对应这种排序顺序，不过用户定义数据类型的设计者也可以选择不同的做法。

如果给出了一个限制计数，那么会返回数量不超过该限制的行（但可能更少些，因为查询本身可能生成的行数就比较少）。LIMIT ALL 的效果和省略 LIMIT 子句一样，就像是LIMIT 带有 NULL 参数一样。

OFFSET 说明在开始返回结果行之前要跳过多少行。OFFSET 0 的效果与省略 OFFSET 子句相同，OFFSET 带有 NULL 参数时也是如此。

如果 OFFSET 和 LIMIT 都出现，那么在开始返回LIMIT 行之前，会先跳过OFFSET 行。

如果使用 LIMIT，那么用一个ORDER BY 子句把结果行约束成一个唯一的顺序是很重要的。否则你就会拿到一个不可预料的该查询的行的子集。你要的可能是第十到第二十行，但以什么顺序的第十到第二十？除非你指定了ORDER BY，否则顺序是不知道的。

查询优化器在生成查询计划时会考虑 LIMIT，因此，根据你为LIMIT 和 OFFSET 提供的值，很可能生成不同的计划（产生不同的行顺序）。因此，使用不同的LIMIT/OFFSET 值选择查询结果的不同子集将生成不一致的结果，除非你用ORDER BY 强制一个可预测的顺序。这并非 bug，这是一个很自然的结果，因为 SQL 没有许诺把查询的结果按照任何特定的顺序发出，除非用了 ORDER BY 来约束顺序。

被 OFFSET 子句跳过的行仍然需要在服务器内部计算；因此，较大的OFFSET 可能效率不高。

7.7. VALUES 列表

VALUES 提供了一种生成“常量表”的方法，这种表可以在查询中使用，而不必真的在磁盘上创建并填充一个表。语法是：

```
VALUES ( expression [, ...] ) [, ...]
```

每个用圆括号括起来的表达式列表都会生成表中的一行。各个列表必须具有相同数量的元素（即表中的列数），并且每个列表中对应位置的项必须具有兼容的数据类型。结果中每一列的实际数据类型按照与UNION 相同的规则确定（参见第 10.5 节）。

一个示例：

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

将会返回一个有两列三行的表。它实际上等效于：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

在默认情况下，PostgreSQL 将 column1、column2 等名字分配给一个 VALUES 表的列。这些列名不是由 SQL 标准指定的，并且不同的数据库系统的做法也不同，因此通常最好使用表别名列表来重写这些默认的名字，像这样：

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t (num,letter);
 num | letter
-----+--------
   1 | one
   2 | two
   3 | three
(3 rows)
```

从语法上说，后面跟有表达式列表的VALUES 列表被视为等同于

```
SELECT select_list FROM table_expression
```

并且可以出现在 SELECT 能出现的任何地方。例如，你可以把它用作UNION 的一部分，或者附加一个*sort_specification*（ORDER BY、LIMIT 和/或 OFFSET）给它。VALUES 最常见的用途是作为一个 INSERT 命令的数据源，其次常见的用途是作为子查询。

更多信息请见 VALUES。

7.8. WITH 查询（公共表表达式）

WITH 提供了一种为更大查询编写辅助语句的方法。这些语句通常被称为公共表表达式或 CTE，可以视为仅在单个查询期间存在的临时表定义。WITH 子句中的每个辅助语句都可以是 SELECT、INSERT、UPDATE 或 DELETE；而 WITH 子句本身则附加到一个主语句上，该主语句同样可以是 SELECT、INSERT、UPDATE 或 DELETE。

7.8.1. WITH 中的 SELECT

WITH 中 SELECT 的基本价值在于把复杂查询分解成更简单的部分。一个示例如下：

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

该查询只显示高销售区域中各产品的销售总额。WITH 子句定义了两个辅助语句 regional_sales 和 top_regions，其中regional_sales 的输出被 top_regions 使用，而top_regions 的输出又被主 SELECT 查询使用。这个示例也可以不借助 WITH 来编写，但那就需要两层嵌套的子SELECT，可读性会稍差一些。

可选的 RECURSIVE 修饰符会让 WITH 从单纯的语法便利变成一种能够实现某些功能的特性，而这些功能在标准 SQL 中无法通过其他方式实现。通过使用RECURSIVE，WITH 查询可以引用自己的输出。下面这个查询是一个非常简单的示例，它计算从 1 到 100 的整数之和：

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

递归 WITH 查询的一般形式总是先写一个非递归项，再写UNION（或 UNION ALL），然后写一个递归项；其中只有递归项可以包含对查询自身输出的引用。这样的查询执行过程如下：

递归查询求值

1. 计算非递归项。对 UNION（但不对 UNION ALL），抛弃重复行。把所有剩余的行包括在递归查询的结果中，并且也把它们放在一个临时的工作表中。

2. 只要工作表不为空，重复下列步骤：

    a. 计算递归项，用当前工作表的内容替换递归自引用。对UNION（不是 UNION ALL），抛弃重复行以及那些与之前结果行重复的行。将剩下的所有行包括在递归查询的结果中，并且也把它们放在一个临时的中间表中。

    b. 用中间表的内容替换工作表的内容，然后清空中间表。

> **注意**
>
> 虽然 RECURSIVE 允许递归指定查询，但这类查询在内部是以迭代方式求值的。

在上面的示例中，工作表在每一步都只有一行，并且在连续步骤中依次取值 1 到 100。到了第 100 步，由于WHERE 子句不再产生输出，因此查询终止。

递归查询通常用于处理层次结构或树形结构数据。下面这个查询是一个有用的示例：给定一个只显示直接包含关系的表，找出某个产品的所有直接和间接子部件：

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity * pr.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

使用递归查询时，必须确保查询的递归部分最终会不返回任何元组，否则查询会无限循环。有时，可以用UNION 代替 UNION ALL，通过丢弃与先前输出行重复的行来做到这一点。不过，循环往往并不涉及完全重复的输出行：可能需要只检查一个或几个字段，以判断是否曾到达同一点。处理这种情况的标准方法是计算一个包含已访问值的数组。例如，考虑以下查询，它搜索表 graph，使用其中的link 字段：

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 1
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

如果 link 关系包含循环，此查询就会循环。由于需要输出 “depth”，仅把UNION ALL 改为 UNION 并不能消除循环。我们需要判断在沿特定链接路径前进时，是否再次到达了同一行。为这个容易循环的查询添加path 和 cycle 两列：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
      ARRAY[g.id],
      false
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      path || g.id,
      g.id = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

除了防止循环，数组值本身也常常很有用，因为它表示到达某一行所经过的 “路径”。

在更一般的情形下，如果需要检查多个字段才能识别一个环，就要使用行数组。例如，如果需要比较f1 和 f2 字段：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
      ARRAY[ROW(g.f1, g.f2)],
      false
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      path || ROW(g.f1, g.f2),
      ROW(g.f1, g.f2) = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

> **提示**
>
> 在通常只有一个字段需要检查即可识别环的情况下，可以省略ROW() 语法。这样就能使用简单数组而不是复合类型数组，从而提高效率。

> **提示**
>
> 递归查询求值算法按广度优先搜索顺序产生输出。可以让外层查询按这样构造的“路径”列进行 ORDER BY 排序，以深度优先搜索顺序显示结果。

当你不确定查询是否可能循环时，一个测试查询的有用技巧是在父查询中放一个LIMIT。例如，这个查询没有LIMIT 时会永远循环：

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

这之所以有效，是因为PostgreSQL 的实现只会计算父查询实际取到的那些 WITH 查询结果行。不建议在生产环境中使用这个技巧，因为其他系统可能采用不同的工作方式。同样，如果让外层查询对递归查询结果排序，或者把它们与其他表做连接，这个技巧通常就不会生效，因为在这些情况下外层查询往往会尝试获取WITH 查询的全部输出。

WITH 查询的一个有用特性是，在父查询每次执行时，它只求值一次，即使父查询或同级WITH 查询对它有多次引用也是如此。因此，在多处需要的昂贵计算可以放在一个WITH 查询中，以避免重复工作。另一个用途是防止带副作用的函数被意外求值多次。不过，另一方面，与普通子查询相比，优化器更难把父查询中的限制下推到WITH 查询中。WITH 查询通常会按书写的样子求值，不会抑制父查询稍后可能丢弃的行。（不过，如上所述，如果对该查询的引用只需要有限数量的行，求值可能会提前停止。）

上面的示例仅显示了 WITH 与 SELECT 一起使用，但它可以以相同的方式附加到INSERT、UPDATE 或 DELETE。在每种情况下，它都等效于提供了可在主命令中引用的临时表。

7.8.2. WITH 中的数据修改语句

你可以在 WITH 中使用数据修改语句（INSERT、UPDATE 或 DELETE）。这样就可以在同一查询中执行多个不同的操作。例如：

```
WITH moved_rows AS (
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

这个查询实际上将行从 products 移动到 products_log。在 WITH 中的 DELETE 从 products 中删除指定的行，通过其RETURNING 子句返回它们的内容；然后主查询读取该输出并将其插入到 products_log 中。

上述示例中有一个细节值得注意：WITH 子句附加在 INSERT 上，而不是附加在INSERT 内部的子 SELECT 上。这是必要的，因为数据修改语句只允许出现在附着于顶层语句的WITH 子句中。不过，普通WITH 的可见性规则仍然适用，因此仍然可以在该子SELECT 中引用 WITH 语句的输出。

正如上述示例所示，WITH 中的数据修改语句通常带有 RETURNING 子句（见第 6.4 节）。构成查询其余部分可引用的临时表的，是RETURNING 子句的输出，而不是数据修改语句的目标表。如果 WITH 中的数据修改语句缺少 RETURNING 子句，那么它就不会形成临时表，也无法被查询的其他部分引用。尽管如此，这样的语句仍然会被执行。下面是一个并不特别有用的示例：

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

这个示例将从表 foo 和 bar 中移除所有行。被报告给客户端的受影响行的数目只会包括从 bar 中移除的行。

数据修改语句中不允许递归自引用。在某些情况下，可以通过引用递归WITH 的输出来绕过这一限制，例如：

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

这个查询会移除一个产品的所有直接和间接子部件。

WITH 中的数据修改语句只会执行一次，并且总会执行到完成，而不管主查询是否读取了它们的全部输出，甚至是否读取了任何输出。注意这与WITH 中 SELECT 的规则不同：正如前一小节所述，SELECT 只会执行到主查询实际需要其输出为止。

WITH 中的子语句彼此之间以及与主查询是并发执行的。因此，在WITH 中使用数据修改语句时，实际更新发生的顺序是不可预知的。所有语句都使用同一个快照执行（参见第 13 章），因此它们无法“看见”彼此对目标表产生的影响。这减轻了实际行更新顺序不可预知所带来的影响，也意味着RETURNING 数据是在不同 WITH 子语句与主查询之间传递更改的唯一方式。例如，在

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

外层 SELECT 返回的将是 UPDATE 操作发生之前的原始价格，而在

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

外层 SELECT 将返回更新后的数据。

在单个语句中试图更新同一行两次是不受支持的。只会发生一次修改，但很难（有时根本无法）可靠地预测究竟是哪一次。这同样适用于删除在同一语句中已经被更新过的行：只有更新会生效。因此，通常应避免在一个语句中两次修改同一行。特别要避免编写那些可能影响主语句或同级子语句所修改行的WITH 子语句，因为这类语句的效果将不可预测。

当前，用作WITH 中数据修改语句目标的任何表不能有条件规则、ALSO 规则或扩展到多个语句的 INSTEAD 规则。

第 8 章 数据类型

PostgreSQL 提供了丰富的原生数据类型。用户可以使用 CREATE TYPE 命令向 PostgreSQL 添加新类型。

表 8.1 展示了所有内置的通用数据类型。“别名”列中列出的多数可选名称，都是 PostgreSQL 出于历史原因在内部使用的名称。此外，还有一些内部使用或已废弃的类型也可用，但未在此列出。

表 8.1. 数据类型

名字	别名	描述
bigint	int8	有符号的 8 字节整数
bigserial	serial8	自动递增的 8 字节整数
bit [(*n*)]		定长位串
bit varying [(*n*)]	varbit [(*n*)]	变长位串
boolean	bool	逻辑布尔值（真/假）
box		平面上的矩形框
bytea		二进制数据（“字节数组”）
character [(*n*)]	char [(*n*)]	定长字符串
character varying [(*n*)]	varchar [(*n*)]	变长字符串
cidr		IPv4 或 IPv6 网络地址
circle		平面上的圆
date		日历日期（年、月、日）
double precision	float8	双精度浮点数（8 字节）
inet		IPv4 或 IPv6 主机地址
integer	int, int4	有符号 4 字节整数
interval [*fields*] [(*p*)]		时间段
json		文本 JSON 数据
jsonb		二进制 JSON 数据，已分解
line		平面上的无限直线
lseg		平面上的线段
macaddr		MAC（媒体访问控制）地址
macaddr8		MAC（媒体访问控制）地址（EUI-64 格式）
money		货币额
numeric [(*p*, *s*)]	decimal [(*p*, *s*)]	可选择精度的精确数值
path		平面上的几何路径
pg_lsn		PostgreSQL 日志序列号
point		平面上的几何点
polygon		平面上的封闭几何路径
real	float4	单精度浮点数（4 字节）

名字	别名	描述
smallint	int2	有符号 2 字节整数
smallserial	serial2	自动递增的 2 字节整数
serial	serial4	自动递增的 4 字节整数
text		变长字符串
time [(*p*)] [without time zone]		一天中的时间（无时区）
time [(*p*)] with time zone	timetz	一天中的时间，包括时区
timestamp [(*p*)] [without time zone]		日期和时间（无时区）
timestamp [(*p*)] with time zone	timestamptz	日期和时间，包括时区
tsquery		文本检索查询
tsvector		文本检索文档
txid_snapshot		用户级事务 ID 快照
uuid		通用唯一标识符
xml		XML 数据

兼容性

下列类型（或它们的这些拼写形式）由 SQL 规定：bigint、bit、bit varying、boolean、char、character varying、character、varchar、date、double precision、integer、interval、numeric、decimal、real、smallint、time（有时区或无时区）、timestamp（有时区或无时区）、xml。

每种数据类型都有一种由其输入和输出函数决定的外部表示。许多内置类型的外部格式都很直观。不过，也有一些类型是 PostgreSQL 所特有的，例如几何路径；还有一些类型可能存在多种可能的格式，例如日期/时间类型。有些输入和输出函数并不可逆，也就是说，输出函数的结果与原始输入相比可能会丢失精度。

8.1. 数字类型

数字类型包括 2、4、8 字节的整数，4、8 字节的浮点数，以及可选精度的小数。表 8.2 列出了所有可用类型。

表 8.2. 数字类型

名字	存储尺寸	描述	范围
smallint	2 字节	小范围整数	-32768 到 +32767
integer	4 字节	整数的典型选择	-2147483648 到 +2147483647
bigint	8 字节	大范围整数	-9223372036854775808 到 +9223372036854775807
decimal	可变	用户指定精度，精确	最高小数点前 131072 位，以及小数点后 16383 位

名字	存储尺寸	描述	范围
numeric	可变	用户指定精度，精确	最高小数点前 131072 位，以及小数点后 16383 位
real	4 字节	可变精度，不精确	6 位十进制精度
double precision	8 字节	可变精度，不精确	15 位十进制精度
smallserial	2 字节	自动递增的小整数	1 到 32767
serial	4 字节	自动递增的整数	1 到 2147483647
bigserial	8 字节	自动递增的大整数	1 到 9223372036854775807

数字类型常量的语法在第 4.1.2 节里描述。数字类型有一整套对应的算术操作符和函数。相关信息请参考第 9 章。下面的几节详细描述这些类型。

8.1.1. 整数类型

类型 smallint、integer 和 bigint 用于存储不同范围的整数，也就是没有小数部分的数。试图存储超出允许范围的值会导致错误。

常用的类型是 integer，因为它提供了在范围、存储空间和性能之间的最佳平衡。一般只有在磁盘空间紧张的时候才使用 smallint 类型。bigint 则设计用于 integer 的范围不够的情况。

SQL 只规定了整数类型 integer（或 int）、smallint 和 bigint。类型 int2、int4 和 int8 都是扩展，也在某些其他SQL 数据库系统中使用。

8.1.2. 任意精度数值

类型 numeric 可以存储非常多位的数字。我们特别建议将它用于货币金额和其它要求计算准确的数量。numeric 值的计算在可能的情况下会得到准确的结果，例如加法、减法、乘法。不过，numeric 类型上的算术运算比整数类型或者下一节描述的浮点数类型要慢很多。

我们在下文中使用以下术语：精度（precision）是一个 numeric 值中有效数字的总位数，也就是小数点两侧数字的总数。小数位数（scale）是小数部分中位于小数点右侧的十进制位数。因此，数值 23.5141 的精度为 6，小数位数为 4。整数可以认为其小数位数为 0。

可以配置 numeric 列的最大精度和最大小数位数。要声明 numeric 类型的列，使用以下语法：

```
NUMERIC(precision, scale)
```

精度必须为正数，小数位数必须为零或正数。也可以使用：

```
NUMERIC(precision)
```

这会选择小数位数为 0。指定：

```
NUMERIC
```

而不指定任何精度或小数位数，会创建一个可存储任意精度和小数位数数值的列，直到达到实现对精度的限制。这种列不会把输入值强制转换为某个特定的小数位数，而声明了小数位数的 numeric 列会将输入值强制转换为该小数位数。（SQL 标准要求默认小数位数为 0，即强制转换为整数精度。我们认为这没有多少用处。如果你关心可移植性，请始终显式指定精度和小数位数。）

> **注意**
>
> 在类型声明中显式指定精度时，所允许的最大值为 1000；未指定精度的NUMERIC 受表 8.2 中所述限制的约束。

如果要存储的值的小数位数大于该列声明的小数位数，系统会把该值舍入到指定的小数位数。然后，如果小数点左侧的位数超过了声明的精度减去声明的小数位数，就会报错。

数值在物理上存储时不会保留多余的前导零或尾随零。因此，列上声明的精度和小数位数只是最大值，而不是固定分配的空间（从这个意义上说，numeric 更像 varchar(*n*)，而不像 char(*n*)）。实际存储需求是每四个十进制数字组占两个字节，再加上 3 到 8 字节的开销。

除了普通数值外，numeric 类型还允许特殊值 NaN，表示“非数”。对 NaN 的任何运算都会产生另一个 NaN。在 SQL 命令中将该值写为常量时，必须加上引号，例如UPDATE table SET x = 'NaN'。输入时，字符串NaN 的识别不区分大小写。

> **注意**
>
> 在大多数“非数字”概念的实现中，NaN 都被认为不等于任何其他数值（包括 NaN 本身）。为了让 numeric 值能够排序并用于基于树的索引，PostgreSQL 将 NaN 值视为彼此相等，并且大于所有非 NaN 值。

类型 decimal 和 numeric 是等效的。两种类型都是 SQL 标准的一部分。

进行舍入时，numeric 类型在遇到恰好处于中间的值时，会朝远离零的方向舍入；而（在大多数机器上）real 和 double precision 类型会把这类值舍入到最近的偶数。例如：

```
SELECT x,
  round(x::numeric) AS num_round,
  round(x::double precision) AS dbl_round
FROM generate_series(-3.5, 3.5, 1) as x;
  x   | num_round | dbl_round
------+-----------+-----------
 -3.5 |        -4 |        -4
 -2.5 |        -3 |        -2
 -1.5 |        -2 |        -2
 -0.5 |        -1 |        -0
  0.5 |         1 |         0
  1.5 |         2 |         2
  2.5 |         3 |         2
  3.5 |         4 |         4
(8 rows)
```

8.1.3. 浮点类型

数据类型 real 和 double precision 是不精确的、变精度的数字类型。实际上，在底层处理器、操作系统和编译器支持的范围内，这两种类型通常实现了IEEE 754 二进制浮点算术标准（分别对应单精度和双精度）。

所谓不精确，是指某些值无法被精确转换为内部格式，只能以近似值存储，因此存储并再取出一个值时，可能会看到轻微差异。如何处理这类误差以及它们在计算中如何传播，是数学和计算机科学中的一个独立领域，这里不再展开，只强调以下几点：

- 如果你要求准确的存储和计算（例如计算货币金额），应使用`numeric` 类型。
- 如果你想用这些类型进行任何重要的复杂计算，尤其是依赖边界情形（无穷大、下溢）特定行为的计算，那么应该仔细评估其实现。
- 比较两个浮点值是否相等，未必总能得到符合预期的结果。

在大多数平台上，`real` 类型的范围至少为 1E-37 到 1E+37，精度至少为 6 位十进制数字。`double precision` 类型的范围通常约为 1E-307 到 1E+308，精度至少为 15 位数字。过大或过小的值会导致错误。如果输入数字的精度过高，可能会发生舍入。过于接近零且无法表示为与零不同的数值，会导致下溢错误。

> **注意**
>
> extra_float_digits 设置控制浮点值转换为文本输出时所包含的额外有效数字位数。使用默认值 `0` 时，输出在 PostgreSQL 支持的每个平台上都相同。增大该值会使输出更精确地表示所存储的值，但可能不具有可移植性。

除了普通数值外，浮点类型还有几个特殊值：

```
Infinity
-Infinity
NaN
```

它们分别表示 IEEE 754 的特殊值“无穷大”、“负无穷大”和“非数”。（在浮点运算不遵循 IEEE 754 的机器上，这些值可能无法按预期工作。）在 SQL 命令中将这些值写为常量时，必须加上引号，例如`UPDATE table SET x = '-Infinity'`。输入时，这些字符串的识别不区分大小写。

> **注意**
>
> IEEE 754 规定，`NaN` 不应与任何其他浮点值（包括 `NaN`）相等。为了允许浮点值被排序，并可用于基于树的索引，PostgreSQL 将 `NaN` 视为彼此相等，并且大于所有非 `NaN` 值。

PostgreSQL 也支持 SQL 标准记法 `float` 和 `float(`*p*`)` 来指定不精确数值类型。这里，*p* 指定可接受的最小二进制精度位数。PostgreSQL 将 `float(1)` 到 `float(24)` 视为选择 `real` 类型，而 `float(25)` 到 `float(53)` 则选择 `double precision`。*p* 超出允许范围会报错。未指定精度的 `float` 视为 `double precision`。

> **注意**
>
> 假定 real 和 double precision 的尾数分别恰好有 24 位和 53 位，对于 IEEE 标准浮点实现而言是正确的。在非 IEEE 平台上可能会有少许偏差，但为简单起见，所有平台都使用相同的p 取值范围。

8.1.4. serial 类型

> **注意**
>
> 本节介绍 PostgreSQL 特有的自动递增列创建方法。另一种方法是使用 SQL 标准的标识列特性，参见 CREATE TABLE。

smallserial、serial 和 bigserial 并不是真正的数据类型，它们只是为了创建唯一标识符列而提供的记法便利（类似于其他一些数据库支持的 AUTO_INCREMENT 属性）。在当前实现中，下列语句：

```
CREATE TABLE tablename (
    colname SERIAL
);
```

等价于以下语句：

```
CREATE SEQUENCE tablename_colname_seq AS integer;
CREATE TABLE tablename (
    colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

这样就创建了一个整数列，并把它的默认值设置为从序列生成器中取值。同时还会加上 NOT NULL 约束，以确保不能插入空值。（在大多数情况下，你可能还会希望再加上UNIQUE 或 PRIMARY KEY 约束，以防意外插入重复值，但这不会自动发生。）最后，该序列会被标记为“属于”该列，这样当列或表被删除时，序列也会随之删除。

> **注意**
>
> 因为 smallserial、serial 和 bigserial 是用序列实现的，所以即使没有删除任何行，列中出现的值序列也可能存在“空洞”或缺口。即使包含该值的行从未成功插入表中，从序列分配出去的值仍会被视为“已使用”。例如，如果插入事务回滚，就会发生这种情况。详细信息见第 9.16 节中的 nextval()。

要向 serial 列插入序列中的下一个值，应指定让serial 列使用其默认值。这既可以通过在 INSERT 语句的列表中省略该列来实现，也可以通过使用 DEFAULT 关键字来实现。

类型名 serial 和 serial4 是等价的：二者都会创建 integer 列。类型名 bigserial 和 serial8 的工作方式相同，只是它们创建的是 bigint 列。如果预计表在其生命周期内会使用

超过 2^{31} 个标识符，就应使用 bigserial。类型名 smallserial 和 serial2 也同理，只是它们创建的是 smallint 列。

为 serial 列创建的序列会在其所属列被删除时自动删除。你也可以在不删除该列的情况下删除该序列，但这会强制移除该列的默认值表达式。

8.2. 货币类型

money 类型以固定的小数精度存储货币金额；参见表 8.3。小数精度由数据库的 lc_monetary 设置决定。表中显示的范围假定有两位小数。输入支持多种格式，包括整数字面量和浮点数字面量，以及典型的货币格式，例如 '$1,000.00'。输出通常也采用后一种形式，但会受到区域设置影响。

表 8.3. 货币类型

名字	存储尺寸	描述	范围
money	8 字节	货币额	-92233720368547758.08 到 +92233720368547758.07

由于该数据类型的输出依赖区域设置，因此在 lc_monetary 设置不同的数据库之间装入 money 数据时，可能无法正常工作。为了避免这类问题，在将转储恢复到新数据库之前，应确保新数据库的 lc_monetary 设置与原数据库相同或等效。

数据类型 numeric、int 和 bigint 的值可以转换为 money。从数据类型 real 和 double precision 的转换可以通过先转换为 numeric 来实现，例如：

```
SELECT '12.34'::float8::numeric::money;
```

但是，我们不推荐这样做。浮点数不应该被用来处理货币，因为浮点数可能会有舍入误差。

一个 money 值可以在不损失精度的情况下转换为 numeric。转换到其他类型可能会丢失精度，并且必须采用两个阶段完成：

```
SELECT '52093.89'::money::numeric::float8;
```

一个 money 值除以一个整数值时，会朝零方向截去小数部分。要得到舍入后的结果，可以除以一个浮点值，或者在除法前先把 money 转换为 numeric，再在除法后转换回 money（如果要避免精度丢失风险，后一种做法更好）。当一个 money 值被另一个 money 值除时，结果是 double precision（即一个纯数字，而不是金额），因为在除法中货币单位被约掉了。

8.3. 字符类型

表 8.4. 字符类型

名字	描述
character varying(*n*), varchar(*n*)	有长度限制的变长
character(*n*), char(*n*)	定长，空格填充
text	无限长度的变长

表 8.4 显示了在 PostgreSQL 里可用的一般用途的字符类型。

SQL 定义了两种主要字符类型：`character varying(n)` 和 `character(n)`，其中 *n* 是正整数。这两种类型都可以存储长度最多为 *n* 个字符（而不是字节）的字符串。试图把更长的字符串存入这些类型的列时会报错，除非超出的字符全部都是空格，在这种情况下字符串会被截断到最大长度。（这一多少有些奇怪的例外是 SQL 标准要求的。）如果要存储的字符串短于声明长度，`character` 类型的值会用空格填充；`character varying` 类型则只是简单地存储较短的字符串。

如果显式把一个值转换为 `character varying(n)` 或 `character(n)`，那么超长的值会被截断为 *n* 个字符而不会报错。（这同样是 SQL 标准要求的。）

`varchar(n)` 和 `char(n)` 分别是 `character varying(n)` 和 `character(n)` 的别名。如果指定了长度，该长度必须大于零，且不能超过 10485760。`character` 若不带长度说明则等同于 `character(1)`。如果 `character varying` 不带长度说明，则该类型接受任意长度的字符串。这是 PostgreSQL 的扩展。

此外，PostgreSQL 还提供 `text` 类型，用于存储任意长度的字符串。虽然 `text` 类型不在 SQL 标准中，但其他若干 SQL 数据库管理系统也提供了它。

`character` 类型的值在物理存储时会在右侧用空格填充到指定宽度 *n*，并且也会按这种形式存储和显示。但是，在比较两个 `character` 值时，尾随空格在语义上被视为不重要并会被忽略。在空白字符有区分意义的排序规则中，这种行为可能产生意外结果；例如，`SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)` 会返回真，即使 `C` 区域设置认为空格大于换行符。把 `character` 值转换为其他字符串类型之一时，尾随空格会被移除。请注意，在 `character varying` 和 `text` 值中，以及进行模式匹配（即 `LIKE` 和正则表达式）时，尾随空格在语义上是有意义的。

可以存储在这些数据类型中的字符由数据库字符集确定，该数据库字符集在创建数据库时选择。无论特定的字符集是什么，都无法存储编码值为零的字符（有时称为 NUL）。有关更多信息，请参阅第 23.3 节。

短字符串（最长 126 字节）的存储需求是 1 个字节，再加上实际字符串本身；对 `character` 而言，这其中还包括填充的空格。更长的字符串则需要 4 个字节的额外开销，而不是 1 个字节。长字符串会被系统自动压缩，因此在磁盘上的实际占用可能更小。非常长的值还会被存储在后台表中，以免影响对较短列值的快速访问。无论如何，可存储的最长字符串大约为 1 GB。（数据类型声明中 *n* 允许的最大值比这还小。更改这个限制并没有意义，因为在多字节字符编码下，字符数和字节数可能差异很大。如果你想存储没有明确上限的长字符串，应使用 `text` 或未指定长度的 `character varying`，而不是随意给出一个长度上限。）

> **提示**
>
> 这三种类型之间没有性能差别，除了使用空格填充类型时会占用更多存储空间，以及在写入带长度约束的列时需要少量额外 CPU 周期来检查长度。虽然在某些其他数据库系统中，`character(n)` 可能有一定性能优势，但在 PostgreSQL 中并不存在这种优势。事实上，由于额外的存储开销，`character(n)` 通常反而是三者中最慢的。大多数情况下，应优先使用 `text` 或 `character varying`。

关于字符串常量的语法，请参见第 4.1.2.1 节；关于可用的操作符和函数，请参见第 9 章。

例 8.1. 使用字符类型

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1

  a   | char_length
------+-------------
 ok   |           2


CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- 显式截断
SELECT b, char_length(b) FROM test2;

   b    | char_length
-------+-------------
 ok    |           2
 good  |           5
 too l |           5
```

1 函数 `char_length` 在第 9.4 节中讨论。

PostgreSQL 中还有两种固定长度字符类型，如表 8.5 所示。`name` 类型仅用于在内部系统目录中存储标识符，并非供一般用户使用。它的长度目前定义为 64 字节（63 个可用字符加结束符），但在 `C` 源代码中应使用常量 `NAMEDATALEN` 来引用。这个长度是在编译时设定的（因此可以针对特殊用途调整）；默认最大长度在未来版本中可能会变化。类型 `"char"`（注意带引号）不同于 `char(1)`，因为它只使用 1 个字节存储。它在系统目录中被用作一种简单的枚举类型。

表 8.5. 特殊字符类型

名字	存储尺寸	描述
`"char"`	1 字节	单字节内部类型
`name`	64 字节	用于对象名的内部类型

8.4. 二进制数据类型

`bytea` 数据类型允许存储二进制串，参见表 8.6。

表 8.6. 二进制数据类型

名字	存储尺寸	描述
`bytea`	1 或 4 字节外加真正的二进制串	变长二进制串

二进制串是八位组（或字节）的序列。二进制串与字符串有两点区别。首先，二进制串明确允许存储值为零的字节以及其他“不可打印”的字节（通常指十进制范围 32 到 126 之外的字节）。而字符串不允许零字节，也不允许那些按数据库所选字符集编码看属于非法的其他字节值或字节

序列。其次，对二进制串的操作处理的是实际字节，而字符串的处理则取决于区域设置。简单地说，二进制串适合存储程序员视为“裸字节”的数据，而字符串适合存储文本。

bytea 类型支持两种用于输入和输出的格式：“十六进制”格式和 PostgreSQL 的历史的“转义”格式。在输入时这两种格式总是会被接受。输出格式则取决于配置参数 bytea_output，其默认值为十六进制（注意十六进制格式是在PostgreSQL 9.0 中被引入的，早期的版本和某些工具无法理解它）。

SQL 标准定义了一种不同的二进制串类型，叫做BLOB 或者 BINARY LARGE OBJECT。其输入格式和 bytea 不同，但是提供的函数和操作符大多一样。

8.4.1. bytea 的十六进制格式

“十六进制”格式把二进制数据编码为每字节两个十六进制数字，最高有效半字节在前。整个串以前缀 \x 开头（以便与转义格式区分）。在某些上下文中，这个开头的反斜线可能需要通过双写进行转义（见第 4.1.2.1 节）。作为输入时，十六进制数字可以使用大写或小写，并且每两个数字组成一组，组与组之间允许出现空白（但组内以及起始的 \x 序列中不能有空白）。十六进制格式与大量外部应用和协议兼容，并且通常比转义格式转换得更快，因此更推荐使用。

示例：

```
SET bytea_output = 'hex';

SELECT '\xDEADBEEF'::bytea;
   bytea
------------
 \xdeadbeef
```

8.4.2. bytea 的转义格式

“转义”格式是 bytea 类型在 PostgreSQL 中的传统格式。它采用把二进制串表示为 ASCII 字符序列的方式，同时把那些不能表示为 ASCII 字符的字节转换为特殊的转义序列。如果从应用角度看，把字节当作字符表示是合理的，那么这种表示法会比较方便。但在实际中它往往让人困惑，因为它模糊了二进制串和字符串之间的区别，而且所选用的转义机制也比较笨拙。因此，对大多数新应用来说，最好避免使用这种格式。

在转义格式中输入 bytea 值时，某些字节值必须转义，而所有字节值都可以转义。通常，转义一个字节的方法是把它写成三位八进制值，并在前面加一个反斜线。反斜线本身（十进制字节值 92）也可以写成双反斜线。表 8.7 展示了必须转义的字符，并给出了可用的替代转义序列。

表 8.7. bytea 字面量中需要转义的字节

十进制字节值	描述	转义输入表示	示例	十六进制表示
0	零值字节	'\000'	SELECT '\000'::bytea;	\x00
39	单引号	'''' 或'\047'	SELECT ''''::bytea;	\x27
92	反斜线	'\\' 或'\134'	SELECT '\\'::bytea;	\x5c
0 到 31 和 127 到 255	“不可打印的”字节	'*xxx*'（八进制值）	SELECT '\001'::bytea;	\x01

是否必须转义这些不可打印字节，会因区域设置不同而有所差异。在某些情况下，你可以不转义它们。

如表 8.7 所示，单引号必须成对写出的原因，在于这对 SQL 命令中的任何字符串常量都成立。通用的字符串常量解析器会去掉最外层单引号，并把任意成对的单引号缩减为一个数据字符。因此，`bytea` 输入函数实际只会看到一个单引号，并将其视为普通数据字符。不过，`bytea` 输入函数会把反斜线视为特殊字符，而表 8.7 中展示的其他行为也是由这个函数实现的。

在某些上下文中，反斜线必须比上面显示的再多写一倍，因为通用的字符串常量解析器也会把成对反斜线缩减为一个数据字符；参见第 4.1.2.1 节。

默认情况下，`bytea` 字节会以 `hex` 格式输出。如果把 bytea_output 改为 `escape`，“不可打印”字节会被转换成等价的三位八进制值，并在前面加一个反斜线。大多数“可打印”字节则会按其在客户端字符集中的标准表示输出，例如：

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;
     bytea
----------------
 abc klm *\251T
```

十进制值 92（反斜线）的字节在输出中会被双写。详情见表 8.8。

表 8.8. `bytea` 输出转义字节

十进制字节值	描述	转义的输出表示	示例	输出结果
92	反斜线	`\\`	`SELECT '\134'::bytea;`	`\\`
0 到 31 和 127 到 255	“不可打印的”字节	`\`*xxx*（八进制值）	`SELECT '\001'::bytea;`	`\001`
32 到 126	“可打印的”字节	客户端字符集表示	`SELECT '\176'::bytea;`	~

取决于你所使用的 PostgreSQL 前端，在转义和反转义 `bytea` 串时可能还需要做额外工作。例如，如果你的接口会自动转换换行和回车，那么你可能还需要对它们进行转义。

8.5. 日期/时间类型

PostgreSQL 支持完整的一组 SQL 日期和时间类型，如表 8.9 所示。这些数据类型上可用的操作见第 9.9 节。日期按格里高利历计算，即使是在该历法启用之前的年份也是如此（更多信息见第 B.6 节）。

表 8.9. 日期/时间类型

名字	存储尺寸	描述	最小值	最大值	解析度
`timestamp [ (`*p*`) ] [ without time zone ]`	8 字节	包括日期和时间（无时区）	4713 BC	294276 AD	1 微秒

名字	存储尺寸	描述	最小值	最大值	解析度
timestamp [(*p*)] with time zone	8 字节	包括日期和时间，有时区	4713 BC	294276 AD	1 微秒
date	4 字节	日期（没有一天中的时间）	4713 BC	5874897 AD	1 日
time [(*p*)] [without time zone]	8 字节	一天中的时间（无日期）	00:00:00	24:00:00	1 微秒
time [(*p*)] with time zone	12 字节	一天中的时间（无日期），带时区	00:00:00+1559	24:00:00-1559	1 微秒
interval [*fields*] [(*p*)]	16 字节	时间间隔	-178000000 年	178000000 年	1 微秒

> **注意**
>
> SQL 标准要求仅写 timestamp 时，应等效于 timestamp without time zone，而 PostgreSQL 也遵循这种行为。timestamptz 被接受为 timestamp with time zone 的简写，这是 PostgreSQL 的扩展。

time、timestamp 和 interval 都接受一个可选精度值 *p*，用来指定在秒字段中保留多少位小数。默认情况下，对精度没有显式上界。*p* 的允许范围是 0 到 6。

interval 类型还有一个附加选项，可以通过写出下面这些短语之一来限制所存储字段的集合：

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

注意，如果同时指定了 *fields* 和 *p*，那么 *fields* 必须包含 SECOND，因为精度只作用于秒。

time with time zone 类型由 SQL 标准定义，但其定义具有一些会让人怀疑其实用性的特性。在大多数情况下，date、time、timestamp without time zone 和 timestamp with time zone 的组合，就足以提供任何应用所需的完整日期/时间功能。

类型 abstime 和 reltime 是内部使用的低精度类型。不推荐在应用中使用这些类型；这些内部类型可能在将来的版本中消失。

8.5.1. 日期/时间输入

日期和时间输入几乎接受任何合理的格式，包括 ISO 8601、与 SQL 兼容的格式、传统 POSTGRES 格式等。对于某些格式，日期输入中日、月、年的顺序可能存在歧义，因此支持指定这些字段的预期顺序。将DateStyle 参数设置为 `MDY`，表示采用“月-日-年”解释；设置为 `DMY` 表示采用“日-月-年”解释；而 `YMD` 表示采用“年-月-日”解释。

PostgreSQL 在处理日期/时间输入方面，比 SQL 标准要求的更灵活。有关日期/时间输入的精确解析规则，以及可识别的文本字段（包括月份、星期几和时区），请参见附录 B。

请记住，任何日期或时间字面值输入都必须像文本字符串一样用单引号括起来。更多信息请参见第 4.1.2.7 节。SQL 要求使用下列语法：

```
type [ (p) ] 'value'
```

其中 *p* 是可选的精度说明，给出秒字段中保留的小数位数。精度可用于 `time`、`timestamp` 和 `interval` 类型，范围是 0 到 6。如果在常量声明中没有指定精度，则默认采用该字面值本身的精度（但不超过 6 位）。

8.5.1.1. 日期

表 8.10 显示了 `date` 类型可能的输入方式。

表 8.10. 日期输入

示例	描述
1999-01-08	ISO 8601；任何模式下的 1 月 8 日（推荐格式）
January 8, 1999	在任何`datestyle` 输入模式下都无歧义
1/8/1999	`MDY` 模式中的 1 月 8 日；`DMY` 模式中的 8 月 1 日
1/18/1999	`MDY` 模式中的 1 月 18 日；在其他模式中被拒绝
01/02/03	`MDY` 模式中的 2003 年 1 月 2 日；`DMY` 模式中的 2003 年 2 月 1 日；`YMD` 模式中的 2001 年 2 月 3 日
1999-Jan-08	任何模式下的 1 月 8 日
Jan-08-1999	任何模式下的 1 月 8 日
08-Jan-1999	任何模式下的 1 月 8 日
99-Jan-08	`YMD` 模式中的 1 月 8 日，否则报错
08-Jan-99	1 月 8 日，但在`YMD` 模式中会报错
Jan-08-99	1 月 8 日，但在`YMD` 模式中会报错
19990108	ISO 8601；任何模式中的 1999 年 1 月 8 日
990108	ISO 8601；任何模式中的 1999 年 1 月 8 日
1999.008	年和一年中的第几天
J2451187	儒略日
January 8, 99 BC	公元前 99 年

8.5.1.2. 时间

一天中的时间类型包括 `time [ (p) ] without time zone` 和 `time [ (p) ] with time zone`。单独写 `time` 等效于 `time without time zone`。

这些类型的有效输入由一个一天中的时间，加上一个可选时区组成（见表 8.11 和表 8.12）。如果在 `time without time zone` 的输入中指定了时区，它会被静默忽略。你也可以指定一个日期，但它同样会被忽略，除非你使用了涉及夏令时规则的时区名称，例如 `America/New_York`。在这种情况下，必须指定日期，以便确定应适用标准时间还是夏令时。相应的时区偏移会被记录到 `time with time zone` 值中，并按存储时的样子输出；它不会根据当前活动时区进行调整。

表 8.11. 时间输入

示例	描述
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601
`040506`	ISO 8601
`04:05 AM`	和 04:05 一样，AM 并不影响值
`04:05 PM`	和 16:05 一样，输入的小时必须为 <= 12
`04:05:06.789-8`	ISO 8601，时区以 UTC 偏移表示
`04:05:06-08:00`	ISO 8601，时区以 UTC 偏移表示
`04:05-08:00`	ISO 8601，时区以 UTC 偏移表示
`040506-08`	ISO 8601，时区以 UTC 偏移表示
`040506+0730`	ISO 8601，时区的 UTC 偏移不是整小时
`040506+07:30:00`	UTC 偏移精确到秒（ISO 8601 中不允许）
`04:05:06 PST`	缩写指定的时区
`2003-04-12 04:05:06 America/New_York`	全名指定的时区

表 8.12. 时区输入

示例	描述
`PST`	缩写（太平洋标准时间）
`America/New_York`	完整时区名
`PST8PDT`	POSIX 风格的时区声明
`-8:00:00`	PST 的 UTC 偏移
`-8:00`	PST 的 UTC 偏移量（ISO 8601 扩展格式）
`-800`	PST 的 UTC 偏移量（ISO 8601 基本格式）
`-8`	PST 的 UTC 偏移量（ISO 8601 基本格式）
`zulu`	UTC 的军方缩写
`z`	`zulu` 的缩写形式（也在 ISO 8601 中）

关于如何指定时区，参见第 8.5.3 节。

8.5.1.3. 时间戳

时间戳类型的有效输入由一个日期和时间的串接组成，后面跟着一个可选时区，以及一个可选的 AD 或 BC（另外，AD/BC 也可以出现在时区前面，但这种顺序并不推荐）。因此：

```
1999-01-08 04:05:06
```

和：

```
1999-01-08 04:05:06 -8:00
```

都是遵循 ISO 8601 标准的有效值。另外，广泛使用的下列格式：

```
January 8 04:05:06 1999 PST
```

也被支持。

按照 SQL 标准，timestamp without time zone 和 timestamp with time zone 字面量的区别在于，时间后是否有 “+” 或 “-” 符号及其后的时区偏移。因此，按照该标准，

```
TIMESTAMP '2004-10-19 10:23:54'
```

是 timestamp without time zone，而

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

是 timestamp with time zone。PostgreSQL 在确定字符串字面量的类型之前，从不检查其内容，因此会把上述两者都视为timestamp without time zone。为确保字面量被视为 timestamp with time zone，应为它显式指定正确类型：

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

若字面量已经被确定为 timestamp without time zone，PostgreSQL 会静默忽略任何时区标记。也就是说，所得值来自输入值中的日期/时间字段，不会根据时区调整。

对于 timestamp with time zone，内部存储的值始终使用 UTC（协调世界时，传统上称为格林尼治标准时间，GMT）。对于显式指定时区的输入值，会使用该时区适当的偏移将它转换为 UTC。如果输入字符串中没有说明时区，则假定它属于系统TimeZone 参数指定的时区，并使用 timezone 时区的偏移将它转换为 UTC。

当输出一个 timestamp with time zone 值时，它总会从 UTC 转换到当前 timezone 时区，并显示为该时区的本地时间。若要查看其他时区的时间，可以修改 timezone，或者使用 AT TIME ZONE 构造（见第 9.9.3 节）。

在 timestamp without time zone 和 timestamp with time zone 之间转换时，通常假定 timestamp without time zone 值应被解释为，或输出为，timezone 本地时间。要为该转换指定不同的时区，可以使用 AT TIME ZONE。

8.5.1.4. 特殊值

为了方便起见，PostgreSQL 支持若干特殊的日期/时间输入值，如表 8.13 所示。infinity 和 -infinity 在系统内部有特殊表示，并且输出时会保持不变；其余值则只是记法上的简写，在读取时会被转换成普通日期/时间值。（特别是，now 及相关字符串在被读取后会立刻转换成某个特定的时间值。）这些值在 SQL 命令中作为常量使用时，都必须用单引号括起来。

表 8.13. 特殊日期/时间输入

输入串	合法类型	描述
epoch	date, timestamp	1970-01-01 00:00:00+00（Unix 系统时间 0）
infinity	date, timestamp	晚于所有其他时间戳
-infinity	date, timestamp	早于所有其他时间戳
now	date, time, timestamp	当前事务的开始时间
today	date, timestamp	今天午夜（00:00）
tomorrow	date, timestamp	明天午夜（00:00）
yesterday	date, timestamp	昨天午夜（00:00）
allballs	time	00:00:00.00 UTC

以下与 SQL 兼容的函数也可用于获取相应数据类型的当前时间值：CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME 和 LOCALTIMESTAMP。（参见第 9.9.4 节。）请注意，这些是 SQL 函数，不会在日期/时间输入字符串中被识别。

> **小心**
>
> 虽然输入字符串 now、today、tomorrow 和 yesterday 可用于交互式 SQL 命令，但当命令被保存以便稍后执行时，例如在预备语句、视图和函数定义中，它们的行为可能令人意外。这些字符串可能在读取时就被转换成某个具体时间值，而该值即使后来已经过时，也会继续被使用。在这类上下文中，应改用某个 SQL 函数。例如，CURRENT_DATE + 1 比 'tomorrow'::date 更安全。

8.5.2. 日期/时间输出

日期/时间类型的输出格式可以设为四种样式之一：ISO 8601、SQL（Ingres）、传统的 POSTGRES（Unix date 格式）或 German。默认是 ISO 格式。（SQL 标准要求使用 ISO 8601 格式。之所以存在名为 “SQL” 的输出格式，只是历史原因。）表 8.14 展示了各种输出样式的示例。date 和 time 类型的输出通常只包含示例中的日期部分或时间部分。不过，POSTGRES 样式对纯日期值仍会采用 ISO 格式输出。

表 8.14. 日期/时间输出风格

风格声明	描述	示例
ISO	ISO 8601，SQL 标准	1997-12-17 07:37:16-08
SQL	传统样式	12/17/1997 07:37:16.00 PST
Postgres	原始样式	Wed Dec 17 07:37:16 1997 PST
German	地区样式	17.12.1997 07:37:16.00 PST

> **注意**
>
> ISO 8601 规定使用大写字母 `T` 来分隔日期和时间。PostgreSQL 在输入时接受该格式，但在输出时使用空格而不是`T`，如上所示。这是为了可读性和与 RFC 3339 以及其他一些数据库系统的一致性。

SQL 和 POSTGRES 风格中，如果指定了 DMY 字段顺序，“日”将出现在“月”之前，否则“月”出现在“日”之前（有关该设置如何影响输入值的解释，请参考第 8.5.1 节）。表 8.15 给出了示例。

表 8.15. 日期顺序习惯

datestyle 设置	**输入顺序**	**示例输出**
`SQL, DMY`	日/月/年	`17/12/1997 15:37:16.00 CET`
`SQL, MDY`	月/日/年	`12/17/1997 07:37:16.00 PST`
`Postgres, DMY`	日/月/年	`Wed 17 Dec 07:37:16 1997 PST`

在 ISO 样式中，时区总是显示为相对于 UTC 的有符号数字偏移，格林尼治以东的时区使用正号。如果偏移是整小时，就显示为 *hh*；如果是整分钟，则显示为 *hh*:*mm*；否则显示为 *hh*:*mm*:*ss*。（第三种情况在任何现代时区标准下都不可能出现，但在处理标准化时区被采用之前的时间戳时可能会看到。）在其他日期样式中，如果当前时区有通用的字母缩写，就会显示该缩写；否则会以 ISO 8601 基本格式的有符号数字偏移显示（*hh* 或 *hhmm*）。

用户可以通过 `SET datestyle` 命令、`postgresql.conf` 配置文件中的 DateStyle 参数，或者服务器端或客户端上的 `PGDATESTYLE` 环境变量来选择日期/时间样式。

格式化函数 `to_char`（见第 9.8 节）也可以作为一个更灵活的方式来格式化日期/时间输出。

8.5.3. 时区

时区及其约定不仅受地球几何形状影响，也受政治决定影响。世界各地的时区在 20 世纪逐渐趋于标准化，但仍然容易发生任意变化，尤其是夏令时规则方面。PostgreSQL 使用广泛采用的 IANA（Olson）时区数据库来获取历史时区规则信息。对于未来时间，则假定某个时区最新已知的规则会无限期地持续下去。

PostgreSQL 努力在典型用法上与 SQL 标准定义保持兼容。不过，SQL 标准在日期和时间类型及其能力方面存在一些奇怪的混搭。两个显而易见的问题是：

- 尽管 `date` 类型不能有关联的时区，`time` 类型却可以。但现实世界中的时区如果不同时关联日期和时间，几乎没有意义，因为偏移量可能会随着夏令时切换而在一年中发生变化。
- 默认时区被指定为相对于 UTC 的一个固定数值偏移。因此，在跨越 DST 边界做日期/时间算术时，根本无法适应夏令时变化。

为了克服这些困难，我们建议在使用时区时采用同时包含日期和时间的日期/时间类型。我们不建议使用 `time with time zone` 类型（尽管 PostgreSQL 出于兼容旧应用以及遵循 SQL 标准的考虑而支持它）。PostgreSQL 对于任何只包含日期或时间的类型，都会假定其使用本地时区。

在系统内部，所有带时区的日期和时间都以 UTC 存储。显示给客户端之前，它们会被转换为由 TimeZone 配置参数指定的本地时间。

PostgreSQL 允许使用三种不同形式来指定时区：

- 完整时区名称，例如 `America/New_York`。识别到的时区名称列在 `pg_timezone_names` 视图中（见第 52.90 节）。PostgreSQL 为此使用广泛采用的 IANA 时区数据，因此同样的时区名称通常也会被其他软件识别。
- 时区缩写，例如 `PST`。与完整时区名称不同，这种指定方式只定义了一个特定的 UTC 偏移，而完整时区名称还可能隐含一套夏令时切换规则。识别到的缩写列在 `pg_timezone_abbrevs` 视图中（见第 52.89 节）。你不能把 TimeZone 或 log_timezone 配置参数设置为时区缩写，但可以在日期/时间输入值中以及与 `AT TIME ZONE` 操作符一起使用缩写。
- 除了时区名称和缩写之外，PostgreSQL 还接受 POSIX 风格的时区说明，见第 B.5 节。这个选项通常不如使用具名时区更合适，但如果没有合适的 IANA 时区条目可用，它可能就是必需的。

简而言之，这就是缩写和全名的区别：缩写表示某个特定的 UTC 偏移，而很多完整名称隐含了当地的夏令时规则，因此可能对应两个不同的 UTC 偏移。例如，`2014-06-04 12:00 America/New_York` 表示纽约当地时间中午；在这个具体日期，它属于东部夏令时（UTC-4），所以 `2014-06-04 12:00 EDT` 指定的是同一个时刻。但 `2014-06-04 12:00 EST` 指定的是东部标准时中午（UTC-5），无论当天名义上是否正在实行夏令时。

更复杂的是，一些司法辖区在不同时间使用同一时区缩写来表示不同的 UTC 偏移；例如在莫斯科，`MSK` 在某些年份表示 UTC+3，在另一些年份则表示 UTC+4。PostgreSQL 会按照该缩写在所给日期上的含义（或最近一次的含义）来解释这类缩写；但与上面的 `EST` 例子一样，这并不一定等同于该日期的当地民用时间。

无论哪种形式，时区名称及其缩写都不区分大小写。（这是对 PostgreSQL 8.2 之前版本的一项改动；在那些版本中，时区名在某些环境下区分大小写，而在另一些环境下则不区分。）

时区名称和缩写并不是硬编码在服务器中的；它们来自安装目录下 `.../share/timezone/` 和 `.../share/timezonesets/` 子目录中的配置文件（见第 B.4 节）。

TimeZone 配置参数可以在 `postgresql.conf` 文件中设置，也可以通过第 19 章中说明的其他标准方式设置。另外，还有一些特殊的设置方法：

- SQL 命令 `SET TIME ZONE` 用于设置会话的时区。它是 `SET TIMEZONE TO` 的另一种写法，语法上更符合 SQL 规范。
- `PGTZ` 环境变量会被 libpq 客户端用于在连接到服务器时发送一条 `SET TIME ZONE` 命令。

8.5.4. 间隔输入

`interval` 值可以使用下列详细语法书写：

```
[@] quantity unit [quantity unit...] [direction]
```

其中 *`quantity`* 是一个数字（可以带有符号）；*`unit`* 是 `microsecond`、`millisecond`、`second`、`minute`、`hour`、`day`、`week`、`month`、`year`、`decade`、`century`、`millennium` 或它们的缩写或复数形式；*`direction`* 可以是 `ago` 或为空。at 符号（`@`）只是可选的噪声。不同单位的数量会按适当的符号规则隐式相加。`ago` 会将所有字段取反。如果 IntervalStyle 被设置为 `postgres_verbose`，该语法也会用于间隔输出。

日、小时、分钟和秒的数量也可以不写显式单位标记。例如，'1 12:59:10' 会被读作 '1 day 12 hours 59 min 10 sec'。同样，年和月的组合也可以用一个连字符表示，例如 '200-10' 会被读作 '200 years 10 months'。（事实上，这些较短形式正是 SQL 标准唯一允许的形式，并且在 IntervalStyle 被设置为 sql_standard 时也用于输出。）

间隔值也可以写成 ISO 8601 时间间隔，使用标准第 4.4.3.2 节的“带标志符的格式”，或第 4.4.3.3 节的“替代格式”。带标志符的格式如下：

```
P quantity unit [ quantity unit ...] [ T [ quantity unit ...]]
```

字符串必须以 P 开头，并且可以包含一个 T 来引出一天中的时间单位。可用的单位缩写见表 8.16。单位可以省略，也可以按任意顺序出现，但小于一天的单位必须出现在 T 之后。特别是，M 的含义取决于它是在 T 之前还是之后。

表 8.16. ISO 8601 间隔单位缩写

缩写	含义
Y	年
M	月（在日期部分中）
W	周
D	日
H	小时
M	分钟（在时间部分中）
S	秒

如果使用替代格式：

```
P [ years-months-days ] [ T hours:minutes:seconds ]
```

串必须以 P 开始，并且一个T 分隔间隔的日期和时间部分。其值按照类似于 ISO 8601 日期的数字给出。

当编写带有 *fields* 说明的间隔常量，或者把字符串赋给一个定义时带有 *fields* 说明的间隔列时，未标记数量的解释方式取决于 *fields*。例如，INTERVAL '1' YEAR 会被读作 1 年，而 INTERVAL '1' 表示 1 秒。此外，位于 *fields* 说明所允许的最小字段“右侧”的字段值会被静默丢弃。例如，写 INTERVAL '1 day 2:03:04' HOUR TO MINUTE 会导致秒字段被丢弃，而不是日字段。

根据 SQL 标准，一个间隔值的所有字段都必须带有相同符号，因此一个前导负号会作用于所有字段；例如间隔字面值 '-1 2:03:04' 中的负号会同时作用于日、小时、分钟和秒。PostgreSQL 允许各字段具有不同符号，并且传统上认为文本表示中的每个字段都带有各自独立的符号，因而在这个例子里，小时、分钟和秒部分会被视为正值。如果 IntervalStyle 被设置为 sql_standard，则会把前导符号视为作用于所有字段（但前提是没有出现额外符号）；否则将采用传统的 PostgreSQL 解释。为了避免混淆，我们建议只要有任何字段为负值，就为每个字段都显式写出符号。

字段值可以带小数部分，例如'1.5 weeks' 或'01:02:03.45'。不过，由于时间间隔内部仅存储三种整数单位（月、日、微秒），小数单位必须转入更小的单位。大于月的单位的小数部分会被截断为整数个月，例如'1.5 years' 会变成'1 year 6 mons'。周和日的

小数部分会按每月 30 天、每天 24 小时计算为整数天和微秒，例如'1.75 months' 会变成 1 mon 22 days 12:00:00。输出时，只有秒会显示小数部分。

表 8.17 展示了一些有效 interval 输入的示例。

表 8.17. 间隔输入

示例	描述
1-2	SQL 标准格式：1 年 2 个月
3 4:05:06	SQL 标准格式：3 日 4 小时 5 分钟 6 秒
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	传统 Postgres 格式：1 年 2 个月 3 日 4 小时 5 分钟 6 秒钟
P1Y2M3DT4H5M6S	ISO 8601 “带标志符的格式”：含义同上
P0001-02-03T04:05:06	ISO 8601 的 “替代格式”：含义同上

在内部，interval 值以月、日和微秒存储。这是因为一个月的天数各不相同，而涉及夏令时调整时，一天可能有 23 或 25 小时。月和日字段为整数，微秒字段则可以存储小数秒。由于时间间隔通常由常量字符串或timestamp 相减创建，这种存储方式在大多数情况下都很有效，但可能产生意外结果：

```
SELECT EXTRACT(hours from '80 minutes'::interval);
 date_part
-----------
         1

SELECT EXTRACT(days from '80 hours'::interval);
 date_part
-----------
         0
```

函数 justify_days 和 justify_hours 可用于调整超出正常范围的日和小时。

8.5.5. 间隔输出

间隔类型的输出格式可以通过 SET intervalstyle 命令设置为以下四种风格之一：sql_standard、postgres、postgres_verbose 或 iso_8601。默认值是 postgres 格式。表 8.18 展示了每种输出风格的示例。

如果间隔值满足 SQL 标准的限制条件（仅有年-月或仅有日-时间，且不混合正负分量），sql_standard 风格会生成符合 SQL 标准的间隔字面值输出。否则，输出将表现为一个标准的年-月字面值串后跟一个日-时间字面值串，并显式加上符号，以消除正负混合间隔的歧义。

postgres 样式的输出与 PostgreSQL 8.4 之前版本在 DateStyle 参数设为 ISO 时的输出一致。

postgres_verbose 样式的输出与 PostgreSQL 8.4 之前版本在 DateStyle 参数设为非 ISO 输出时的结果一致。

iso_8601 风格的输出符合 ISO 8601 标准 4.4.3.2 节描述的 “带标志符的格式”。

表 8.18. 间隔输出风格示例

风格声明	年-月间隔	日-时间间隔	混合间隔
sql_standard	1-2	3 4:05:06	-1-2 +3 -4:05:06
postgres	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
postgres_verbose	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

8.6. 布尔类型

PostgreSQL 提供标准的 SQL 类型 boolean，参见表 8.19。boolean 可以有多个状态：“true（真）”、“false（假）”和第三种状态“unknown（未知）”，未知状态由SQL 空值表示。

表 8.19. 布尔数据类型

名字	存储尺寸	描述
boolean	1 字节	状态为真或假

在 SQL 查询中，布尔常量可以用 SQL 关键字 TRUE、FALSE 和 NULL 表示。

boolean 类型的输入函数接受以下表示“真”状态的字符串：

```
true
yes
on
1
```

以及以下表示“假”状态的字符串：

```
false
no
off
0
```

这些字符串的唯一前缀也同样可以接受，例如 t 或 n。前导或尾随空白会被忽略，并且大小写不敏感。

boolean 类型的输出函数总是产生 t 或 f，如例 8.2 所示。

例 8.2. 使用 boolean 类型

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

在 SQL 查询中，最好使用关键字 `TRUE` 和 `FALSE` 来书写布尔常量（这与 SQL 兼容）。不过，你也可以使用遵循第 4.1.2.7 节中所述通用字符串常量语法的字符串来表示，例如 `'yes'::boolean`。

注意，语法分析器会自动将 `TRUE` 和 `FALSE` 理解为 `boolean` 类型，但 `NULL` 不会，因为它可以是任意类型。因此在某些上下文中，你可能需要显式将 `NULL` 转换为 `boolean`，例如 `NULL::boolean`。反过来，如果语法分析器能够判定某个字符串字面量必定是 `boolean` 类型，则也可以省略这类转换。

8.7. 枚举类型

枚举（enum）类型是由一个静态的有序值集合构成的数据类型。它们等效于很多编程语言所支持的 `enum` 类型。枚举类型的一个示例可以是一周中的各天，或者一个数据的状态值集合。

8.7.1. 枚举类型的声明

可以使用 CREATE TYPE 命令创建枚举类型，例如：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

创建之后，该 enum 类型就可以像任何其他类型一样在表定义和函数定义中使用：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

8.7.2. 排序

enum 类型中各值的顺序，就是创建该类型时列出的顺序。枚举支持所有标准比较操作符以及相关聚合函数。例如：

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
 name  | current_mood
-------+--------------
 Moe   | happy
 Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name  | current_mood
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
 name
-------
 Larry
(1 row)
```

8.7.3. 类型安全性

每一种枚举数据类型都是独立的并且不能和其他枚举类型相比较。看这样一个示例：

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

如果你确实需要做这类事情，可以编写自定义操作符，或者在查询中添加显式类型转换：

```
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |          4
(1 row)
```

8.7.4. 实现细节

枚举标签是大小写敏感的，因此'happy'与'HAPPY'是不同的。标签中的空白也是有意义的。

尽管枚举类型的主要目的是用于值的静态集合，但也有方法在现有枚举类型中增加新值和重命名值（见 ALTER TYPE）。不能从枚举类型中去除现有的值，也不能更改这些值的排序顺序；除非删除并重建枚举类型，否则无法这样做。

一个枚举值在磁盘上占据 4 个字节。一个枚举值的文本标签的长度受限于 NAMEDATALEN 设置，该设置被编译在PostgreSQL 中，在标准编译下它表示最多 63 字节。

从内部枚举值到文本标签的转换关系被保存在系统目录 pg_enum 中。直接查询该目录可能会有所帮助。

8.8. 几何类型

几何数据类型表示二维的空间物体。表 8.20 展示了 PostgreSQL 中可以用的几何类型。

表 8.20. 几何类型

名字	存储尺寸	描述	表示
point	16 字节	平面上的点	(x,y)
line	32 字节	无限直线	{A,B,C}
lseg	32 字节	有限线段	((x1,y1),(x2,y2))
box	32 字节	矩形框	((x1,y1),(x2,y2))
path	16+16n 字节	封闭路径（类似于多边形）	((x1,y1),...)
path	16+16n 字节	开放路径	[(x1,y1),...]
polygon	40+16n 字节	多边形（类似于封闭路径）	((x1,y1),...)
circle	24 字节	圆	<(x,y),r>（圆心和半径）

我们提供了丰富的函数和操作符来进行各种几何操作，例如缩放、平移、旋转以及求交等，详见第 9.11 节。

8.8.1. 点

点是几何类型的基本二维构造块。用下面任一种语法描述 `point` 类型的值：

```
( x , y )
  x , y
```

其中 *x* 和 *y* 分别是坐标，都是浮点数。

点使用第一种语法输出。

8.8.2. 直线

直线由线性方程 *A*x + *B*y + *C* = 0 表示，其中*A* 和 *B* 不能同时为零。类型 `line` 的值采用以下形式输入和输出：

```
{ A, B, C }
```

另外，还可以用下列任一形式输入：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中 (*x1*,*y1*) 和 (*x2*,*y2*) 是直线上不同的两点。

8.8.3. 线段

线段用一对线段的端点来表示。`lseg` 类型的值用下面任一种语法指定：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中(*x1*,*y1*) 和 (*x2*,*y2*) 是线段的端点。

线段使用第一种语法输出。

8.8.4. 方框

方框用其对角的点对表示。`box` 类型的值使用下面任一种语法指定：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中(*x1*,*y1*) 和 (*x2*,*y2*) 是方框的对角点。

方框使用第二种语法输出。

在输入时可以提供任意两个对角，但是值将根据需要被重新排列，并依次存储为右上角和左下角。

8.8.5. 路径

路径由一系列连接的点组成。路径可能是开放的，也就是认为列表中第一个点和最后一个点没有被连接起来；也可能是封闭的，这时认为第一个和最后一个点被连接起来。

`path` 类型的值用下面任一种语法指定：

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

其中的点是组成路径的线段的端点。方括弧（`[]`）表示一个开放的路径，圆括弧（`()`）表示一个封闭的路径。如第三种到第五种语法所示，当最外面的圆括号被省略时，路径将被假定为封闭。

路径的输出视情况使用第一种或第二种语法。

8.8.6. 多边形

多边形由点的列表（多边形的顶点）表示。多边形与封闭路径非常相似，但存储方式不同，并且有自己的支持例程。

`polygon` 类型的值用下列任一种语法指定：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

其中的点是组成多边形边界的线段的端点。

多边形的输出使用第一种语法。

8.8.7. 圆

圆由一个圆心和一个半径代表。`circle` 类型的值用下面任一种语法指定：

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

其中 `(`*x*`,`*y*`)` 是圆心，而 *r* 是圆的半径。

圆的输出用第一种语法。

8.9. 网络地址类型

PostgreSQL 提供了用于存储 IPv4、IPv6 和 MAC 地址的数据类型，如表 8.21 所示。使用这些数据类型来存储网络地址，比使用纯文本类型更好，因为它们提供了输入错误检查以及专门的操作符和函数（见第 9.12 节）。

表 8.21. 网络地址类型

名字	存储尺寸	描述
`cidr`	7 或 19 字节	IPv4 和 IPv6 网络
`inet`	7 或 19 字节	IPv4 和 IPv6 主机以及网络
`macaddr`	6 字节	MAC 地址
`macaddr8`	8 字节	MAC 地址（EUI-64 格式）

在对 `inet` 或 `cidr` 数据类型排序时，IPv4 地址总是排在 IPv6 地址之前，即使后者是封装或映射 IPv4 地址所得的 IPv6 地址，例如 ::10.2.3.4 或 ::ffff:10.4.3.2。

8.9.1. `inet`

`inet` 在一个字段中保存 IPv4 或 IPv6 主机地址，以及可选的子网信息。子网由主机地址中包含的网络地址位数（即“网络掩码”）表示。如果网络掩码为 32 且地址是 IPv4，那么该值表示的不是子网，而只是单个主机。在 IPv6 中，地址长度为 128 位，因此 128 位表示一个唯一的主机地址。请注意，如果你只想接受网络地址，应使用 `cidr` 类型，而不是 `inet`。

该类型的输入格式为 *`address/y`*，其中 *`address`* 是 IPv4 或 IPv6 地址，而 *`y`* 是网络掩码中的位数。如果省略 *`/y`* 部分，则 IPv4 的网络掩码取 32，IPv6 的网络掩码取 128，因此该值表示单个主机。显示时，如果 *`/y`* 部分表示的是单个主机，则它不会被显示出来。

8.9.2. `cidr`

`cidr` 类型保存 IPv4 或 IPv6 网络说明。输入和输出格式遵循无类别域间路由（CIDR）约定。指定网络的格式为 *`address/y`*，其中 *`address`* 是网络的 IPv4 或 IPv6 表示，而 *`y`* 是网络掩码中的位数。如果省略 *`y`*，则会按照旧式的分类网络编号系统假设来计算，但至少会大到足以覆盖输入中写出的所有八位组。若指定的网络地址在所给网络掩码右侧仍有置位比特，则会报错。

表 8.22 展示了一些示例。

表 8.22. `cidr` 类型输入示例

`cidr` 输入	`cidr` 输出	`abbrev(cidr)`
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16

cidr 输入	cidr 输出	abbrev(cidr)
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. inet 与 cidr

inet 和 cidr 两种数据类型的本质区别在于：inet 接受网络掩码右侧仍含有非零位的值，而 cidr 不接受。例如，192.168.0.1/24 对 inet 是有效的，但对 cidr 无效。

> **提示**
>
> 如果你不喜欢 inet 或 cidr 值的输出格式，可以试试 host、text 和 abbrev 函数。

8.9.4. macaddr

macaddr 类型存储 MAC 地址，例如以太网卡硬件地址（尽管 MAC 地址还用于其它用途）。可以接受下列格式的输入：

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

这些示例指定的都是同一个地址。十六进制数字 a 到 f 可使用大小写任意形式。输出总是采用上面展示的第一种形式。

IEEE 标准 802-2001 指定了 MAC 地址的第二种形式（带连字符）作为规范形式，并指定第一种形式（带冒号）用于位反转、最高位在前的表示法，因此 08-00-2b-01-02-03 = 10:00:D4:80:40:C0。这种约定现在被广泛忽略，仅适用于过时的网络协议（如 Token Ring）。PostgreSQL 不提供位反转的功能；所有接受的格式都使用规范的 LSB 顺序。

剩下的五种输入格式不属于任何标准。

8.9.5. macaddr8

此 macaddr8 类型以 EUI-64 格式存储 MAC 地址，例如常见的以太网卡硬件地址（尽管 MAC 地址也有其他用途）。该类型既接受 6 字节也接受 8 字节的 MAC 地址，并以 8 字节格式存储。给定的 6 字节格式 MAC 地址会以 8 字节格式存储，其中第 4 和第 5 字节分别设为 FF 和 FE。注意，IPv6 使用修改后的 EUI-64 格式，从 EUI-48 转换后应将第 7 位设为 1。提供的函数 macaddr8_set7bit 可完成这一更改。一般来说，接受由成对十六进制数字（在字节边界上）构成的输入，各对之间可以统一使用':'、'-' 或'.' 中的一种分隔。十六进制数字的数量必须为 16 个（8 字节）或 12 个（6 字节）。开头和结尾的空白会被忽略。以下是可以接受的输入格式示例：

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
'08002b-0102030405'
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
'08002b0102030405'
```

这些示例都指定同一个地址。十六进制数字 a 到 f 均可使用大写或小写。输出总是采用所示的第一种形式。上述最后六种输入格式不属于任何标准。要将传统 EUI-48 格式的 48 位 MAC 地址转换为修改后的 EUI-64 格式，以用作 IPv6 地址的主机部分，可以使用macaddr8_set7bit，如下所示：

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');

    macaddr8_set7bit
-------------------------
 0a:00:2b:ff:fe:01:02:03
(1 row)
```

8.10. 位串类型

位串是由 1 和 0 组成的字符串。它们可用于存储或展示位掩码。SQL 中有两种位类型：bit(*n*) 和 bit varying(*n*)，其中 *n* 是正整数。

bit 类型的数据长度必须与 *n* 完全一致；试图存储更短或更长的位串都会报错。bit varying 数据则是长度上限为 *n* 的变长类型，更长的串会被拒绝。不带长度的 bit 等效于 bit(1)，而不带长度说明的 bit varying 表示长度不受限。

> **注意**
>
> 如果显式地将一个位串值转换为 bit(*n*)，它会在右侧被截断或补零，直到长度恰好为 *n* 位，且不会报错。类似地，如果显式地将一个位串值转换为 bit varying(*n*)，而其长度超过 *n* 位，则会在右侧被截断。

关于位串常量的语法信息，请参见第 4.1.2.5 节。位逻辑操作符和字符串操作函数也可用；参见第 9.6 节。

例 8.3. 使用位串类型

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');

ERROR:  bit string length 2 does not match type bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

  a  |  b
-----+-----
 101 | 00
 100 | 101
```

一个位串值每 8 位需要 1 个字节，再加上 5 或 8 个字节的额外开销，具体取决于串的长度。（不过，长值可能会被压缩或移到行外存储，与第 8.3 节中对字符串的说明相同。）

8.11. 文本检索类型

PostgreSQL 提供了两种专为支持全文检索而设计的数据类型。所谓全文检索，是指在一组自然语言文档中查找最匹配某个查询的文档。`tsvector` 类型以适合文本检索的优化形式表示文档，`tsquery` 类型则表示文本查询。关于这一功能的详细解释见第 12 章；相关函数和操作符的概览见第 9.13 节。

8.11.1. tsvector

tsvector 值是一个排好序且互不重复的词位（lexeme）列表，这些词已经过正规化，以便把同一单词的不同变体合并起来（详见第 12 章）。排序和去重会在输入时自动完成，如下例所示：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

若要表示包含空白或标点的词位，请用引号将它们括起来：

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                 tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

（本例及下例使用美元引用的字符串常量，以避免在字符串内部必须双写引号所造成的混淆。）嵌入的引号和反斜线必须双写：

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                    tsvector
-------------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

还可以为词位附加整数形式的位置：

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
                                 tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

一个位置通常表示源词在文档中的位置。位置信息可用于邻近度排序。位置值可以位于 1 到 16383 之间；更大的数字会被静默设为 16383。同一词位的重复位置会被丢弃。

带有位置的词位还可以进一步附加一个权重标签，其值可以是 A、B、C 或 D。D 是默认值，因此在输出中不会显示：

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

权重通常用于反映文档结构，例如把标题中的词和正文中的词区分开来。文本检索排序函数可以为不同的权重标记分配不同优先级。

必须认识到，tsvector 类型本身并不会执行任何词语正规化；它假定输入的词已经按照应用需求完成正规化。例如：

```
SELECT 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

对于大多数英文全文检索应用来说，上述词会被视为尚未正规化，但 tsvector 并不在意。原始文档文本通常应先经过 to_tsvector，以按检索需要对词语进行正规化：

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

更多细节仍请参见第 12 章。

8.11.2. tsquery

tsquery 值存储要搜索的词位，并可用布尔操作符 & （AND）、| （OR）和 ! （NOT）将它们组合起来，也可使用短语搜索操作符 <-> （FOLLOWED BY）。此外，FOLLOWED BY 还有一种变体 <*N*>，其中 *N* 是整数常量，用于指定被搜索的两个词位之间的距离。<-> 等效于 <1>。

可以使用圆括号强制指定这些操作符的分组方式。若没有圆括号，! （NOT）的绑定最紧，其次是 <-> （FOLLOWED BY），再其次是 & （AND），最后是 | （OR）。

以下是一些示例：

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

可选地，tsquery 中的词位可以用一个或多个权重字母标注，这会限制它们只匹配在 tsvector 中带有这些权重之一的词位：

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

此外，tsquery 中的词位还可以带上 * 标签来指定前缀匹配：

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

这个查询将匹配 tsvector 中任何以 “super” 开头的词位。

引号的使用规则与前面介绍 tsvector 时相同；同样，与 tsvector 一样，任何需要的词语正规化都必须在转换为 tsquery 类型之前完成。to_tsquery 函数很适合用来实现这种正规化：

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

请注意，to_tsquery 会像处理其他词一样处理前缀，这意味着下面的比较会返回真：

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
 t
```

因为 postgres 会被词干化为 postgr：

```
SELECT to_tsvector( 'postgraduate' ), to_tsquery( 'postgres:*' );
  to_tsvector  | to_tsquery
---------------+------------
 'postgradu':1 | 'postgr':*
```

因而它能够匹配 postgraduate 词干化后的形式。

8.12. UUID 类型

uuid 数据类型用于存储由 RFC 4122、ISO/IEC 9834-8:2005 及相关标准定义的通用唯一标识符（UUID）。（有些系统把这种数据类型称为全局唯一标识符，或 GUID。）这种标识符是一个 128 位的量，由某种算法生成，该算法的设计目标是：使已知宇宙中其他人用同一算法生成出相同标识符的概率极低。因此，对于分布式系统而言，这类标识符能比序列生成器提供更好的唯一性保证，因为序列生成器仅在单个数据库内唯一。

UUID 写作一串小写十六进制数字，并用连字符分隔成若干组：先是 8 位一组，接着是三个 4 位组，最后是一个 12 位组。总共 32 个十六进制数字表示 128 个二进制位。标准形式的 UUID 例如：

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

PostgreSQL 也接受其他输入形式：可以使用大写字母、用花括号包围标准格式、省略部分或全部连字符，或者在任意 4 位分组后额外加上连字符。例如：

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

输出总是采用标准形式。

PostgreSQL 为 UUID 提供存储和比较函数，但核心数据库不包含生成 UUID 的函数，因为没有一种算法适合所有应用。uuid-ossp 模块提供实现多种标准算法的函数。pgcrypto 模块也提供生成随机 UUID 的函数。或者，可以由客户端应用或通过服务器端函数调用的其他库来生成 UUID。

8.13. XML 类型

`xml` 数据类型可用于存储 XML 数据。与把 XML 数据存储在 `text` 字段中相比，它的优势在于会检查输入值的良构性，并且提供了可执行类型安全操作的支持函数；见第 9.14 节。使用该数据类型要求构建时使用 `configure --with-libxml`。

`xml` 类型既可以存储 XML 标准所定义的良构“文档”，也可以存储“内容”片段；后者是参照 XQuery 和 XPath 数据模型中更宽松的“文档节点”[1]概念来定义的。粗略地说，这意味着内容片段可以拥有多个顶层元素或字符节点。表达式 *`xmlvalue`* `IS DOCUMENT` 可用于判断某个特定的 `xml` 值是完整文档，还是仅仅是一个内容片段。

关于 `xml` 数据类型的限制和兼容性说明，见第 D.3 节。

[1] https://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/#DocumentNode

8.13.1. 创建 XML 值

要从字符数据生成 xml 类型的值，可以使用函数 xmlparse：

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

示例：

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></
book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

按照 SQL 标准，这是把字符串转换为 XML 值的唯一方法；但 PostgreSQL 特有的下列语法：

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

也可以使用。

即使输入值指定了文档类型声明（DTD），xml 类型也不会根据 DTD 来验证输入值。目前也没有内置支持可依据其他 XML 模式语言（如 XML Schema）来执行验证。

反向操作，也就是从 xml 生成字符串值，则使用函数 xmlserialize：

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

type 可以是 character、character varying 或 text（或它们的别名）。同样，根据 SQL 标准，这是在 xml 与字符类型之间进行转换的唯一方式，但 PostgreSQL 也允许你直接进行类型转换。

当字符串值在不经过 XMLPARSE 或 XMLSERIALIZE 的情况下与 xml 类型互相转换时，选择 DOCUMENT 还是 CONTENT 由会话配置参数 “XML 选项” 决定，可以使用标准命令设置：

```
SET XML OPTION { DOCUMENT | CONTENT };
```

或使用更接近 PostgreSQL 风格的语法

```
SET xmloption TO { DOCUMENT | CONTENT };
```

默认值是 CONTENT，因此允许所有形式的 XML 数据。

8.13.2. 编码处理

在客户端、服务器以及其间传输的 XML 数据上处理多种字符编码时，必须格外小心。使用文本模式向服务器发送查询并把查询结果返回给客户端时（这是通常使用的模式），PostgreSQL 会将客户端与服务器之间传输的所有字符数据转换为目标端的字符编码，参见第 23.3 节。这也包括表示 XML 值的字符串，如上例所示。这通常意味着，由于字符数据在客户端和服务器之间传输时可能被转换为其他编码，XML 数据中包含的编码声明可能会失效，因为内嵌的编码声明本身并不会被修改。为处理这种情况，表示 xml 类型输入值的字符串中所包含的编码声明会被忽略，其内容被假定为当前服务器编码。因此，为了正确处理，客户端发出的 XML 数据字符串必须采用当前客户端编码。客户端负责在将文档发送给服务器之前把它们转换为当前客户端编码，或适当调整客

户端编码。输出时，xml 类型值不会带有编码声明，而客户端应假定所有数据都采用当前客户端编码。

若使用二进制模式把查询参数传给服务器并把查询结果返回客户端，则不会执行编码转换，因此情况有所不同。在这种情况下，XML 数据中的编码声明会被识别；如果缺少编码声明，则该数据会被假定为 UTF-8（这是 XML 标准的要求；另请注意 PostgreSQL 不支持 UTF-16）。输出时，除非客户端编码是 UTF-8（此时编码声明会被省略），否则数据会带有一个指明客户端编码的编码声明。

不言而喻，在 PostgreSQL 中处理 XML 数据时，如果 XML 数据编码、客户端编码和服务器编码三者相同，不仅更高效，也更不容易出错。由于 XML 数据在内部以 UTF-8 处理，如果服务器编码也是 UTF-8，则处理效率最高。

> **小心**
>
> 当服务器编码不是 UTF-8 时，某些与 XML 相关的函数在非 ASCII 数据上可能完全无法工作。尤其是 xmltable() 和 xpath()，这是一个已知问题。

8.13.3. 访问 XML 值

xml 数据类型有些特殊，因为它不提供任何比较操作符。这是因为对 XML 数据并不存在良定义且通用的比较算法。其结果是，你无法通过把某个 xml 列与搜索值比较来检索行。因此，XML 值通常应伴随一个独立的键字段，例如 ID。另一种比较 XML 值的办法，是先把它们转换成字符串；但请注意，字符串比较对 XML 的比较需求通常帮助不大。

由于 xml 数据类型没有可用的比较操作符，因此无法直接在这种类型的列上创建索引。如果需要在 XML 中快速搜索，可行方案包括：将表达式转换为字符串类型后为其建立索引，或者为某个 XPath 表达式建立索引。当然，实际查询也必须相应调整为使用该被索引的表达式。

PostgreSQL 的文本检索功能也可用于加速 XML 数据的全文检索。不过，目前 PostgreSQL 发行版中仍缺少所需的预处理支持。

8.14. JSON 类型

JSON 数据类型用于存储 JSON（JavaScript Object Notation）数据，如 RFC 7159[2] 所定义。这类数据也可以存储为 text，但 JSON 数据类型的优势在于会强制每个存储值都符合 JSON 规则。此外，对于存储在这些数据类型中的数据，还提供了各种 JSON 专用的函数和操作符；见第 9.15 节。

有两种 JSON 数据类型：json 和 jsonb。它们接受的输入值集合几乎相同。实际使用中的主要区别是效率。json 数据类型保存输入文本的精确副本，处理函数每次执行时都必须重新解析；而 jsonb 数据以分解后的二进制格式存储，额外的转换开销使输入稍慢，但无需重新解析，因此处理速度明显更快。jsonb 还支持索引，这可能带来显著优势。

由于 json 类型存储的是输入文本的精确副本，因此它会保留词元之间在语义上无关紧要的空白，以及 JSON 对象内部键的顺序。此外，如果值中的某个 JSON 对象包含同一个键多次，所有键/值对都会被保留下来（处理函数会将最后一个值视为生效值）。相比之下，jsonb 不保留空白，不保留对象键的顺序，也不保留重复的对象键。如果输入中指定了重复的键，则只保留最后一个值。

[2] https://tools.ietf.org/html/rfc7159

一般而言，大多数应用都应优先将 JSON 数据存储为 `jsonb`，除非存在相当特殊的需求，例如遗留系统对对象键顺序的假设。

PostgreSQL 的每个数据库只允许使用一种字符集编码。因此，除非数据库编码为 UTF8，否则 JSON 类型无法严格遵循 JSON 规范。直接包含数据库编码无法表示的字符会失败；反过来，数据库编码能够表示但 UTF8 无法表示的字符则会被允许。

RFC 7159 允许 JSON 字符串包含以 `\u`*XXXX* 表示的 Unicode 转义序列。`json` 类型的输入函数允许 Unicode 转义，而不管数据库使用什么编码，并且只检查语法是否正确（即 `\u` 后面是否有四位十六进制数字）。但 `jsonb` 的输入函数更严格：除非数据库编码为 UTF8，否则不允许非 ASCII 字符（大于 `U+007F` 的字符）的 Unicode 转义。`jsonb` 类型也会拒绝 `\u0000`（因为它无法在 PostgreSQL 的 `text` 类型中表示），并要求正确使用 Unicode 代理对来表示 Unicode 基本多文种平面之外的字符。有效的 Unicode 转义会转换为等价的 ASCII 或 UTF8 字符进行存储，包括将代理对合并为单个字符。

> **注意**
>
> 第 9.15 节中描述的许多 JSON 处理函数会将 Unicode 转义转换为普通字符，因此即使输入的类型为 `json` 而非 `jsonb`，也会抛出上述相同类型的错误。`json` 输入函数不做这些检查，可以视为历史遗留行为，不过它确实允许在非 UTF8 数据库编码下简单地存储 JSON Unicode 转义，而不进行处理。一般而言，应尽可能避免将 JSON 中的 Unicode 转义与非 UTF8 数据库编码混用。

当把文本形式的 JSON 输入转换为 `jsonb` 时，RFC 7159 描述的基本类型实际上会映射到原生的 PostgreSQL 类型上，如表 8.23 所示。因此，什么样的数据构成有效的 `jsonb` 会有一些额外但较小的限制，这些限制不适用于 `json` 类型，也不适用于抽象意义上的 JSON；它们对应于底层数据类型可表示范围的限制。特别地，`jsonb` 会拒绝超出 PostgreSQL `numeric` 数据类型范围的数字，而 `json` 不会。RFC 7159 允许这种由实现定义的限制。不过在实践中，这类问题更可能出现在其他实现中，因为通常会把 JSON 的 `number` 基本类型表示为 IEEE 754 双精度浮点数（RFC 7159 明确预见并允许了这一点）。当把 JSON 用作与这类系统交换数据的格式时，应考虑与原先由 PostgreSQL 存储的数据相比丢失数值精度的风险。

另一方面，正如表中所指出的那样，JSON 基本类型的输入格式还有一些轻微限制，而对应的 PostgreSQL 类型并没有这些限制。

表 8.23. JSON 基本类型及其对应的 PostgreSQL 类型

JSON 基本类型	PostgreSQL 类型	说明
`string`	`text`	不允许 `\u0000`；如果数据库编码不是 UTF8，也不允许非 ASCII 字符的 Unicode 转义
`number`	`numeric`	不允许 `NaN` 和 `infinity` 值
`boolean`	`boolean`	只接受小写拼写 `true` 和 `false`
`null`	（无）	SQL `NULL` 是不同的概念

8.14.1. JSON 输入和输出语法

JSON 数据类型的输入/输出语法遵循 RFC 7159。

以下都是有效的 json（或 jsonb）表达式：

```
-- 简单的标量/基本值
-- 基本值可以是数字、加引号的字符串、true、false 或 null
SELECT '5'::json;

-- 包含零个或多个元素的数组（元素不必是同一类型）
SELECT '[1, 2, "foo", null]'::json;

-- 包含键值对的对象
-- 注意，对象的键必须始终是加引号的字符串
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- 数组和对象可以任意嵌套
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

如前所述，当一个 JSON 值被输入后又在不进行任何额外处理的情况下输出时，json 会输出与输入完全相同的文本，而 jsonb 不会保留诸如空白这类语义上无关紧要的细节。例如，请注意这里的差异：

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                       json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                       jsonb
--------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

一个值得注意的语义无关细节是，在 jsonb 中，数字会按照底层 numeric 类型的行为输出。在实践中，这意味着使用 E 记数法输入的数字在输出时将不再使用这种写法，例如：

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;
         json          |          jsonb
-----------------------+-------------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

不过，正如这个例子所示，jsonb 会保留小数部分末尾的零，尽管对于等值检查之类的用途来说，这些零在语义上并不重要。

8.14.2. 有效地设计 JSON 文档

以 JSON 形式表示数据，可能比传统的关系数据模型灵活得多，这在需求变化较大的环境中尤其有吸引力。这两种方法完全可能在同一个应用中共存并互为补充。但是，即使对于追求最大灵活性的应用，也仍然建议 JSON 文档拥有某种相对固定的结构。这种结构通常并不受强制约束（尽管也可以用声明式方式强制某些业务规则），但具有可预测的结构会让编写查询更容易，从而能够有效地汇总表中一组“文档”（数据项）。

当 JSON 数据存储在表中时，它与任何其他数据类型一样，都要面对相同的并发控制考量。虽然存储大型文档是可行的，但要记住，任何更新都会在整行上获取一个行级锁。应考虑将 JSON 文档限制在可管理的大小，以减少更新事务之间的锁争用。理想情况下，每个 JSON 文档都应表示一个原子数据项，根据业务规则，无法再合理地将它细分为更小且可独立修改的数据项。

8.14.3. jsonb 包含与存在

测试包含是 jsonb 的一项重要能力。对于 json 类型，则没有与之对应的一组功能。包含测试用于检查一个 jsonb 文档中是否包含另一个文档。除特别说明外，下面这些示例都返回真：

```
-- 简单的标量/基本值只包含相同的值:
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- 左侧数组包含右侧数组:
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- 数组元素的顺序无关紧要，因此下面的结果也为真:
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- 重复的数组元素也无关紧要:
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- 左侧对象包含右侧只有一个键值对的对象:
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}'::jsonb @> '{"version": 9.4}':
:jsonb;

-- 左侧数组被认为不包含右侧数组，
-- 尽管其中嵌套了一个类似的数组:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- 结果为假

-- 但增加一层嵌套后，就构成了包含关系:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- 同样，下面也不构成包含关系:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;  -- 结果为假

-- 包含由顶层键和空对象构成的对象:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

一般原则是，被包含对象在结构和数据内容上都必须与包含对象匹配；必要时，可以从包含对象中丢弃某些不匹配的数组元素或对象键/值对后再进行这种匹配。但要记住，在进行包含匹配时，数组元素的顺序并不重要，重复的数组元素实际上也只会被考虑一次。

对于结构必须匹配这一一般原则，有一个特殊例外：数组可以包含一个基本值：

```
-- 这个数组包含该基本字符串值:
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- 这个例外并不对称——下面的结果表示不包含:
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- 结果为假
```

jsonb 还有一个存在操作符，它可看作包含的一种变体：它测试某个字符串（以 text 值给出）是否在 jsonb 值的顶层作为对象键或数组元素出现。除特别说明外，下面这些示例都返回真：

```
-- 字符串作为数组元素存在：
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- 字符串作为对象键存在：
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- 不考虑对象的值：
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- 结果为假

-- 与包含关系一样，存在性必须在顶层匹配：
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- 结果为假

-- 如果字符串与一个基本 JSON 字符串匹配，则认为它存在：
SELECT '"foo"'::jsonb ? 'foo';
```

当涉及很多键或元素时，JSON 对象比数组更适合用于测试包含或存在，因为对象与数组不同，内部已针对搜索做了优化，不需要进行线性搜索。

> **提示**
>
> 由于 JSON 包含是嵌套的，因此适当的查询可以跳过对子对象的显式选择。例如，假设我们有一个 doc 列，其顶层是对象，而且大多数对象都带有 tags 字段，该字段中包含子对象数组。下面这个查询会找出同时出现包含 "term":"paris" 的子对象和包含 "term":"food" 的子对象的项，同时忽略 tags 数组之外的任何此类键：
>
> ```
> SELECT doc->'site_name' FROM websites
>   WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
> ```
>
> 例如，也可以用下面的写法完成同样的事情：
>
> ```
> SELECT doc->'site_name' FROM websites
>   WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
> ```
>
> 但这种做法的灵活性较差，而且通常效率也更低。
>
> 另一方面，JSON 的存在操作符并不是嵌套的：它只会在 JSON 值的顶层查找指定的键或数组元素。

各种包含和存在操作符，以及所有其他 JSON 操作符和函数，均见第 9.15 节。

8.14.4. jsonb 索引

GIN 索引可用于高效搜索大量 jsonb 文档（数据项）中出现的键或键/值对。提供了两种 GIN “操作符类”，它们在性能和灵活性之间提供不同的权衡。

对于 jsonb，默认 GIN 操作符类支持使用顶层键存在操作符?、?& 和?|以及路径/值存在操作符@> 的查询。（这些操作符所实现语义的详情，参见表 9.44。）使用此操作符类创建索引的示例如下：

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

非默认的 GIN 操作符类 jsonb_path_ops 仅支持为@> 操作符建立索引。使用此操作符类创建索引的示例如下：

```
CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);
```

假设有一个表，用于存储从第三方 Web 服务检索到的 JSON 文档，而且该服务的模式定义已有文档说明。一个典型文档如下：

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
    "address": "178 Howard Place, Gulf, Washington, 702",
    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}
```

我们把这些文档存储在名为 api 的表中，存放于名为 jdoc 的 jsonb 列里。如果在该列上创建了 GIN 索引，那么下面这样的查询就可以利用这个索引：

```
-- 查找键 "company" 的值为 "Magnafone" 的文档
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnafone"}';
```

但是，类似下面这样的查询就无法使用该索引，因为虽然操作符 ? 可索引，但它并未直接应用到被索引的列 jdoc 上：

```
-- 查找键 "tags" 包含键或数组元素 "qui" 的文档
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

不过，只要适当地使用表达式索引，上述查询也可以利用索引。如果经常查询 "tags" 键中的特定项，那么定义如下索引可能是值得的：

```
CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));
```

现在，WHERE 子句 jdoc -> 'tags' ? 'qui' 会被识别为把可索引操作符 ? 应用于被索引表达式 jdoc -> 'tags'。（表达式索引的更多信息见第 11.7 节。）

另一种查询方法是利用包含，例如：

```
-- 查找键 "tags" 包含数组元素 "qui" 的文档
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';
```

jdoc 列上的简单 GIN 索引可以支持这个查询。但要注意，这样的索引会存储 jdoc 列中每个键和值的副本，而前一个例子中的表达式索引只存储 tags 键下出现的数据。虽然简单索引方法灵活得多（因为它支持对任意键的查询），但有针对性的表达式索引通常会比简单索引更小，搜索起来也更快。

虽然 jsonb_path_ops 操作符类只支持使用 @> 操作符的查询，但与默认操作符类 jsonb_ops 相比，它具有显著的性能优势。对于相同数据，jsonb_path_ops 索引通常比 jsonb_ops 索引小得多，搜索的针对性也更强，特别是当查询包含数据中频繁出现的键时。因此，搜索操作的性能通常优于默认操作符类。

jsonb_ops 与 jsonb_path_ops GIN 索引之间的技术差异在于，前者会为数据中的每个键和值分别创建独立的索引项，而后者只会为数据中的每个值创建索引项。[3]基本上，每个 jsonb_path_ops 索引项都是该值连同通向该值的键一起计算出的 hash。例如，要索引 {"foo": {"bar": "baz"}}，会创建一个单独的索引项，其 hash 值中同时纳入 foo、bar 和 baz 这三者。因此，查找这一结构的包含查询会得到一次非常精确的索引搜索；但完全没有办法据此找出 foo 是否作为键出现。另一方面，jsonb_ops 索引会分别创建三个索引项来表示 foo、bar 和 baz；然后为了执行包含查询，它会查找包含这三个项的行。尽管 GIN 索引可以相当高效地执行这种 AND 搜索，但它仍然会比等效的 jsonb_path_ops 搜索更不精确、也更慢，尤其是在包含这三个索引项中任意一个的行数非常多时。

jsonb_path_ops 方法的一个缺点是，它不会为不包含任何值的 JSON 结构产生索引项，例如 {"a": {}}。如果请求查找包含此类结构的文档，就需要执行一次全索引扫描，这会相当慢。因此，jsonb_path_ops 并不适合经常执行此类搜索的应用。

[3] 在这里，术语“值”也包括数组元素，尽管在 JSON 术语中，有时会把数组元素与对象中的值区分开来。

jsonb 也支持 btree 和 hash 索引。通常只有在需要检查完整 JSON 文档是否相等时，这些索引才有用。对于btree 排序，jsonb 数据的顺序很少值得关注，但为求完整，列出如下：

对象 > 数组 > 布尔值 > 数字 > 字符串 > *Null*

包含 n 个键值对的对象 > 包含 n - 1 个键值对的对象

包含 n 个元素的数组 > 包含 n - 1 个元素的数组

键值对数量相等的对象按以下顺序比较：

键-1，值-1，键-2 ...

注意，对象键按其存储顺序比较；尤其是，较短的键存储在较长的键之前，因此可能产生不直观的结果，例如：

```
{ "aa": 1, "c": 1} > {"b": 1, "d": 1}
```

类似地，元素数量相等的数组按以下顺序比较：

元素-1，元素-2 ...

JSON 基本值使用与其底层 PostgreSQL 数据类型相同的规则进行比较。字符串使用数据库的默认排序规则进行比较。

8.14.5. 转换

还提供了一些附加扩展，为不同过程语言实现了 jsonb 类型的转换。

PL/Perl 的扩展名为 jsonb_plperl 和 jsonb_plperlu。如果使用它们，jsonb 值会按适当方式映射为 Perl 数组、hash 和标量。

PL/Python 的扩展分别名为 jsonb_plpythonu、jsonb_plpython2u 和 jsonb_plpython3u（关于 PL/Python 的命名约定请见第 46.1 节）。如果使用它们，jsonb 值会按适当方式映射为 Python 字典、列表和标量。

8.15. 数组

PostgreSQL 允许把表的列定义为变长多维数组。可以创建内置或用户定义的基础类型、枚举类型、复合类型、范围类型或域的数组。

8.15.1. 数组类型的声明

为了说明数组类型的用法，我们创建下面这个表：

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

如上所示，数组数据类型通过在数组元素的数据类型名后附加方括号（`[]`）来命名。上述命令将创建一个名为 `sal_emp` 的表，其中有一个 `text` 类型的列（`name`）、一个表示员工各季度薪资的一维 `integer` 数组（`pay_by_quarter`），以及一个表示员工每周日程安排的二维 `text` 数组（`schedule`）。

`CREATE TABLE` 的语法允许指定数组的确切大小，例如：

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

不过，当前实现会忽略给出的任何数组大小限制，也就是说，其行为与未指定长度的数组相同。

当前实现也不会强制执行所声明的维度数。对于某一特定元素类型的数组，无论大小或维度数如何，都会被视为同一种类型。因此，在 `CREATE TABLE` 中声明数组大小或维度数仅仅是文档说明；它不会影响运行时行为。

另一种使用关键字 `ARRAY`、并且符合 SQL 标准的语法可用于一维数组。`pay_by_quarter` 也可以定义为：

```
    pay_by_quarter  integer ARRAY[4],
```

或者，如果不指定数组大小：

```
    pay_by_quarter  integer ARRAY,
```

不过，与前面一样，PostgreSQL 在任何情况下都不会强制执行大小限制。

8.15.2. 数组值输入

要把数组值写成字面常量，请将元素值放在花括号内，并用逗号分隔。（如果你了解 C，这与 C 中初始化结构体的语法有些类似。）你可以给任意元素值加上双引号；如果它包含逗号或花括号，则必须这样做。（更多细节见下文。）因此，数组常量的一般格式如下：

```
'{ val1 delim val2 delim ... }'
```

其中 *delim* 是该类型的分隔符字符，它记录在其 pg_type 条目中。在 PostgreSQL 发行版提供的标准数据类型里，除类型 box 使用分号（;）外，其余都使用逗号（,）。每个 *val* 要么是数组元素类型的常量，要么是一个子数组。数组常量的一个示例是：

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

这个常量是一个 3 x 3 的二维数组，由三个整数子数组构成。

要把数组常量中的某个元素设为 NULL，请把该元素值写成 NULL。（NULL 的任意大小写变体都可以。）如果你想要实际的字符串值“NULL”，就必须为它加上双引号。

（这类数组常量实际上只是第 4.1.2.7 节中讨论的通用类型常量的一种特例。该常量最初会被当作字符串处理，然后传递给数组输入转换例程。可能需要显式指定类型。）

现在我们来看几个 INSERT 语句：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

前两条插入语句的结果如下：

```
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |                  schedule
-------+---------------------------+-------------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

多维数组的各个子数组在每个维度上的长度必须匹配。不匹配会导致错误，例如：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ERROR:  multidimensional arrays must have array expressions with matching dimensions
```

也可以使用 ARRAY 构造器语法：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

注意，数组元素是普通的 SQL 常量或表达式；例如，字符串字面值用单引号而不是双引号包围，而在数组字面量中则会用双引号。关于 ARRAY 构造器语法的更多细节见第 4.2.12 节。

8.15.3. 访问数组

现在，我们可以对该表执行一些查询。首先，演示如何访问数组中的单个元素。下面的查询取回第二季度薪资发生变化的员工姓名：

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

数组下标写在方括号内。默认情况下，PostgreSQL 对数组采用从 1 开始的编号约定，也就是说，一个有 *n* 个元素的数组从 array[1] 开始，到 array[*n*] 结束。

下面这个查询取回所有员工第三季度的薪资：

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

我们还可以访问数组的任意矩形切片，即子数组。数组切片通过在一个或多个数组维度上写成 *lower-bound*:*upper-bound* 的形式来表示。例如，下面这个查询取回 Bill 在一周中前两天里每天的首项日程：

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

如果任何一个维度写成切片形式，也就是包含冒号，那么所有维度都会被当作切片处理。任何只有单个数字（没有冒号）的维度都会被视为从 1 到该数字指定的范围。例如，`[2]` 会被当作 `[1:2]`，如下例所示：

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

                  schedule
-------------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

为了避免与非切片情况混淆，最好对所有维度都使用切片语法，例如写成 `[1:2][1:1]`，而不是 `[2][1:1]`。

切片说明符中的 *lower-bound* 和/或 *upper-bound* 可以省略；缺失的边界会分别由数组下标的下界或上界代替。例如：

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{lunch},{presentation}}
(1 row)

SELECT schedule[:][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

如果数组本身或任一下标表达式为 NULL，则数组下标表达式将返回 NULL。此外，如果下标超出数组边界，也会返回 NULL（这种情况不会报错）。例如，如果 `schedule` 当前的维度是 `[1:3][1:2]`，那么引用 `schedule[3][3]` 会得到 NULL。类似地，使用错误数量的下标访问数组，得到的也是 NULL 而不是错误。

同样地，如果数组本身或任一下标表达式为 NULL，数组切片表达式也会返回 NULL。不过，在其他情况下，例如选择一个完全位于当前数组边界之外的数组切片时，切片表达式返回的是空（零维）数组而不是 NULL。（这与非切片行为不一致，是出于历史原因。）如果所请求的切片与数组边界仅部分重叠，那么它会被静默缩减为重叠区域，而不是返回 NULL。

任意数组值的当前维度都可以用 array_dims 函数取回：

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

array_dims 产生的是一个 text 结果，这对人工阅读比较方便，但对程序来说可能不太方便。也可以用 array_upper 和 array_lower 取回维度信息，它们分别返回指定数组维度的上界和下界：

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

array_length 会返回指定数组维度的长度：

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_length
--------------
            2
(1 row)
```

cardinality 返回数组跨所有维度的元素总数。它实际上就是调用 unnest 会产生的行数：

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';

 cardinality
-------------
           4
(1 row)
```

8.15.4. 修改数组

数组值可以被整体替换：

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

或者使用 `ARRAY` 表达式语法：

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

也可以更新数组中的单个元素：

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

或者更新其中一个切片：

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

也可以使用省略 *lower-bound* 和/或 *upper-bound* 的切片语法，但前提是被更新的数组值不是 NULL，也不是零维数组（否则就没有现有的下标边界可供替代）。

已存储的数组值可以通过给尚不存在的元素赋值来扩展。原有元素与新赋值元素之间的任何位置都将用 NULL 填充。例如，如果数组 `myarray` 当前有 4 个元素，那么在一次更新把值赋给 `myarray[6]` 之后，它将有 6 个元素；`myarray[5]` 将包含 NULL。目前，以这种方式扩展只允许用于一维数组，不允许用于多维数组。

带下标的赋值也允许创建不使用从 1 开始下标的数组。例如，可以给 `myarray[-2:7]` 赋值，从而创建一个下标值范围为 -2 到 7 的数组。

新的数组值也可以使用连接操作符 `||` 构造：

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

连接操作符允许把单个元素添加到一维数组的开头或末尾。它也接受两个 *N* 维数组，或者一个 *N* 维数组与一个 *N+1* 维数组。

当把单个元素添加到一维数组的开头或末尾时，结果数组会保留该数组操作数的下界下标。例如：

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

当连接两个维度数相同的数组时，结果会保留左侧操作数外层维度的下界下标。结果数组由左侧操作数的所有元素后跟右侧操作数的所有元素组成。例如：

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

当把一个 *N* 维数组添加到一个 *N+1* 维数组的开头或末尾时，其结果与前面“单个元素与数组相连”的情况类似。每个 *N* 维子数组本质上都是该 *N+1* 维数组外层维度中的一个元素。例如：

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

也可以使用 array_prepend、array_append 或 array_cat 函数来构造数组。前两个只支持一维数组，而 array_cat 支持多维数组。示例如下：

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

在简单情况下，优先使用上面讨论的连接操作符，而不是直接调用这些函数。不过，由于连接操作符被重载以同时服务于这三种情形，所以在某些场景下使用这些函数之一有助于避免歧义。例如，考虑：

```
SELECT ARRAY[1, 2] || '{3, 4}';  -- 未指定类型的字面量被当作数组
 ?column?
-----------
 {1,2,3,4}

SELECT ARRAY[1, 2] || '7';                  -- 这个字面量也一样
ERROR:  malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL;                 -- 未加类型修饰的 NULL 也一样
 ?column?
----------
 {1,2}
(1 row)

SELECT array_append(ARRAY[1, 2], NULL);     -- 这可能才是原本想表达的意思
 array_append
--------------
 {1,2,NULL}
```

在上面的示例中，解析器看到连接操作符的一侧是整数数组，另一侧是类型未定的常量。它用来解析该常量类型的启发式规则是假定其类型与该操作符另一侧的输入相同，在这里就是整数数组。因此，连接操作符会被假定为表示 `array_cat`，而不是 `array_append`。如果这是错误的选择，可以通过把常量显式转换为数组的元素类型来修正；但显式使用 `array_append` 也许是更好的解决方案。

8.15.5. 在数组中搜索

要在数组中搜索某个值，就必须检查每一个值。如果你知道数组的大小，这可以手工完成。例如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

不过，对于大型数组，这样很快就会变得繁琐；而在数组大小未知时，这也没有帮助。另一种方法见第 9.23 节。上面的查询可以改写为：

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

此外，如果要查找数组中所有值都等于 10000 的行，可以使用：

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

另外，也可以使用 generate_subscripts 函数。例如：

```
SELECT * FROM
   (SELECT pay_by_quarter,
           generate_subscripts(pay_by_quarter, 1) AS s
      FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

关于该函数的说明见表 9.59。

也可以使用 && 操作符来搜索数组，它会检查左操作数是否与右操作数有重叠。例如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

关于该操作符和其他数组操作符的更多说明见第 9.18 节。如第 11.2 节所述，也可以通过适当的索引加速。

也可以使用 array_position 和 array_positions 函数在数组中搜索特定值。前者返回某个值在数组中首次出现位置的下标；后者返回一个数组，其中包含该值在数组中所有出现位置的下标。例如：

```
SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon');
 array_positions
-----------------
 2

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
```

> **提示**
>
> 数组不是集合；搜索特定数组元素可能是数据库设计不当的信号。可以考虑使用一个独立的表，让原本会成为数组元素的每个项各占一行。这样会更容易搜索，而且在元素数量很多时通常也有更好的可伸缩性。

8.15.6. 数组输入和输出语法

数组值的外部文本表示由若干项构成，这些项会按照数组元素类型的 I/O 转换规则进行解释，再加上一些表示数组结构的修饰。修饰部分包括数组值外层的花括号（{ 和 }），以及相邻项之间的分隔符字符。分隔符字符通常是逗号（,），但也可能是别的字符：它由数组元素类型的 typdelim 设置决定。在 PostgreSQL 发行版提供的标准数据类型中，除类型 box 使用分号（;）外，其余都使用逗号。在多维数组中，每个维度（行、平面、立方体等）都有自己的一层花括号，并且同一级别相邻的、由花括号括起的各项之间必须写出分隔符。

如果元素值是空字符串、包含花括号、分隔符字符、双引号、反斜线或空白，或者与单词 NULL 匹配，数组输出例程就会在元素值外面加上双引号。元素值中嵌入的双引号和反斜线会用反斜线转义。对于数字数据类型，可以安全地假定不会出现双引号；但对于文本数据类型，则应准备好处理带引号和不带引号两种情况。

默认情况下，数组各维度的下界下标值都设为 1。要表示具有其他下界的数组，可以在写出数组内容之前显式指定数组下标范围。这种修饰由包围每个数组维度上下界的方括号（[]）构成，中间以冒号（:）作为分隔符字符。数组维度修饰后面再跟一个等号（=）。例如：

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

只有当一个或多个下界不等于 1 时，数组输出例程才会在结果中包含显式维度信息。

如果某个元素写入的值是 NULL（任何大小写变体都可以），该元素就会被视为 NULL。若存在任何引号或反斜线，则会禁用这种行为，从而可以输入字面字符串值“NULL”。另外，为了与 PostgreSQL 8.2 之前的版本保持向后兼容，可以将array_nulls 配置参数设为 off，从而禁止把 NULL 识别为 NULL。

如前所述，在写数组值时，可以给任意单个数组元素加上双引号。如果元素值不加引号就会让数组值解析器产生混淆，那么你必须这样做。例如，包含花括号、逗号（或者该数据类型的分隔符字符）、双引号、反斜线，或者前后带有空白的元素值，必须使用双引号。空字符串以及与单词 NULL 匹配的字符串也必须加引号。要在一个带双引号的数组元素值中放入双引号或反斜线，需要在它前面加反斜线。或者，你也可以避免使用引号，而改用反斜线转义来保护所有原本会被当作数组语法的数据字符。

你可以在左花括号之前或右花括号之后添加空白。也可以在任意单个项字符串之前或之后添加空白。在所有这些情况下，这些空白都会被忽略。不过，双引号元素内部的空白，或者元素中夹在两个非空白字符之间的空白，不会被忽略。

> **提示**
>
> 在 SQL 命令中写数组值时，ARRAY 构造器语法（见第 4.2.12 节）通常比数组字面量语法更容易使用。在 ARRAY 中，各个元素值的写法与它们不作为数组成员时完全相同。

8.16. 复合类型

复合类型表示一行或一条记录的结构；本质上它就是字段名及其数据类型的列表。PostgreSQL 允许像使用简单类型那样，在许多相同的场合使用复合类型。例如，表中的一列可以声明为复合类型。

8.16.1. 复合类型的声明

下面是两个定义复合类型的简单示例：

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

这种语法与 CREATE TABLE 类似，但只能指定字段名和类型；目前还不能包含约束（例如NOT NULL）。注意，AS 关键字必不可少；如果没有它，系统会认为你想使用另一种 CREATE TYPE 命令，并产生令人费解的语法错误。

定义了这些类型之后，我们可以用它们来创建表：

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

或者创建函数：

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

每当你创建一个表时，系统也会自动创建一个与该表同名的复合类型，用来表示该表的行类型。例如，如果我们这样写：

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

那么上面展示的同一个 inventory_item 复合类型也会顺带生成，并且同样可以像前面那样使用。不过，当前实现有一个重要限制：由于复合类型本身不关联任何约束，表定义中给出的那些约束在表之外的复合类型值上并不适用。（解决办法是：在该复合类型之上创建一个域，并把所需的约束作为该域的CHECK 约束。）

8.16.2. 构造复合值

要把复合值写成字面常量，请将各字段值放在圆括号内，并用逗号分隔。你可以给任意字段值加双引号；如果它包含逗号或圆括号，则必须这样做。（更多细节见下文。）因此，复合常量的一般格式如下：

```
'( val1 , val2 , ... )'
```

一个示例是：

```
'("fuzzy dice",42,1.99)'
```

这就是上文定义的 `inventory_item` 类型的一个合法值。要让某个字段为 NULL，就在列表中对应的位置什么也不写。例如，这个常量指定第三个字段为 NULL：

```
'("fuzzy dice",42,)'
```

如果想写空字符串而不是 NULL，请写双引号：

```
'("",42,)'
```

这里第一个字段是非 NULL 的空字符串，第三个字段是 NULL。

（这些常量实际上只是第 4.1.2.7 节中讨论的通用类型常量的一种特例。该常量最初会被当作字符串处理，然后传递给复合类型输入转换例程。可能需要显式指定类型，以指明应将该常量转换为哪种类型。）

`ROW` 表达式语法也可以用于构造复合值。在大多数情况下，它比字符串字面量语法简单得多，因为你不必担心多层引号。我们在上文已经用过这种方法：

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

只要表达式中有多个字段，ROW 关键字实际上是可选的，因此这些可以简写为：

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

关于 `ROW` 表达式语法的更多细节，见第 4.2.13 节。

8.16.3. 访问复合类型

要访问复合列中的某个字段，可以写一个点号再加字段名，这很像通过表名选取字段。实际上，它与通过表名选取字段太像了，以至于你通常必须使用圆括号，以免让解析器混淆。例如，你可能尝试从示例表`on_hand`中选取一些子字段：

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

这不会起作用，因为根据 SQL 语法规则，名称`item`会被当成表名，而不是`on_hand`的列名。你必须写成这样：

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

或者，如果你还需要使用表名（例如在多表查询中），可以这样写：

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

现在，加上括号的对象就会被正确解释为对`item`列的引用，然后就可以从中选出子字段。

无论何时从复合值中选择字段，都会遇到类似的语法问题。例如，要从一个返回复合值的函数结果中只选取一个字段，你需要这样写：

```
SELECT (my_func(...)).field FROM ...
```

如果没有额外的圆括号，这将生成一个语法错误。

特殊字段名`*`表示“所有字段”，其进一步解释见第 8.16.5 节。

8.16.4. 修改复合类型

下面是一些插入和更新复合列时正确语法的示例。先看插入或更新整个列值的情况：

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

第一个示例省略了`ROW`，第二个示例使用了它；两种写法都可以。

我们也可以更新复合列中的单个子字段：

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

注意，这里不需要（事实上也不能）给紧跟在`SET`后面的列名加圆括号，但在等号右侧的表达式中引用同一列时，则需要加圆括号。

我们也可以把子字段指定为`INSERT`的目标：

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

如果我们没有为该列的所有子字段提供值，其余子字段就会填充为 NULL 值。

8.16.5. 在查询中使用复合类型

在查询中，复合类型有多种特殊的语法规则和行为。这些规则提供了有用的简写形式，但如果不了解背后的逻辑，也可能让人困惑。

在 PostgreSQL 中，查询中对表名（或别名）的引用，实际上就是对该表当前行的复合值的引用。例如，如果我们有一个如上文所示的表 `inventory_item`，就可以写：

```
SELECT c FROM inventory_item c;
```

这个查询会产生一个单独的复合值列，因此我们可能得到如下输出：

```
           c
------------------------
 ("fuzzy dice",42,1.99)
(1 row)
```

不过要注意，简单名称会先与列名匹配，再与表名匹配，因此这个示例之所以可行，只是因为该查询涉及的表中没有名为`c` 的列。

普通的限定列名语法 *`table_name.column_name`* 可以理解为对该表当前行的复合值进行字段选择。（出于效率原因，实际上并不是这样实现的。）

当我们写

```
SELECT c.* FROM inventory_item c;
```

时，根据 SQL 标准，应该得到把该表内容展开为独立列后的结果：

```
    name     | supplier_id | price
-------------+-------------+-------
 fuzzy dice |          42 |  1.99
(1 row)
```

就好像查询写成了

```
SELECT c.name, c.supplier_id, c.price FROM inventory_item c;
```

PostgreSQL 会对任何结果为复合类型的表达式应用这种展开行为，不过正如上文所示，只要`.*` 所作用的值不是简单表名，就需要给该值加圆括号。例如，如果`myfunc()` 是一个返回复合类型的函数，该复合类型有`a`、`b` 和 `c` 三列，那么下面两个查询的结果相同：

```
SELECT (myfunc(x)).* FROM some_table;
SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some_table;
```

> **提示**
>
> PostgreSQL 处理列展开时，实际上会把第一种形式转换成第二种形式。因此，在这个示例中，无论使用哪种语法，myfunc() 每行都会被调用三次。如果它是一个开销较大的函数，你可能希望避免这种情况，可以使用如下查询：
>
> ```
> SELECT m.* FROM some_table, LATERAL myfunc(x) AS m;
> ```
>
> 把该函数放在 LATERAL FROM 项中，可以避免它在每行上被调用多次。m.* 仍会展开为 m.a, m.b, m.c，但现在这些变量只是对该FROM 项输出的引用。（这里的 LATERAL 关键字其实是可选的，不过我们把它写出来，是为了明确该函数会从some_table 中取得 x。）

当 *composite_value*.* 出现在 SELECT 输出列表、INSERT/UPDATE/DELETE 中的 RETURNING 列表、VALUES 子句或行构造器的顶层时，就会产生这种列展开行为。在所有其他上下文中（包括嵌套在上述结构之内时），给复合值附加 .* 不会改变其值，因为它表示“所有列”，因此结果仍然是同一个复合值。例如，如果somefunc() 接受一个复合值参数，这些查询就是等价的：

```
SELECT somefunc(c.*) FROM inventory_item c;
SELECT somefunc(c) FROM inventory_item c;
```

在这两种情况下，inventory_item 的当前行都会作为单个复合值参数传递给该函数。尽管 .* 在这种场合并不起作用，使用它仍是一种良好风格，因为它能明确表明这里想要的是复合值。特别是，解析器会把c.* 中的 c 视为表名或别名，而不是列名，因此不会产生歧义；而没有 .* 时，就不清楚c 究竟表示表名还是列名，而且如果存在名为c 的列，实际上会优先按列名解释。

另一个说明这些概念的例子是，下面这些查询的含义都相同：

```
SELECT * FROM inventory_item c ORDER BY c;
SELECT * FROM inventory_item c ORDER BY c.*;
SELECT * FROM inventory_item c ORDER BY ROW(c.*);
```

所有这些 ORDER BY 子句都指定了该行的复合值，因此会按照第 9.23.6 节中描述的规则对行进行排序。不过，如果inventory_item 包含一个名为 c 的列，第一种情况就会不同于其他情况，因为它表示只按那一列排序。按照前面展示的列名，下面这些查询也与上述查询等效：

```
SELECT * FROM inventory_item c ORDER BY ROW(c.name, c.supplier_id, c.price);
SELECT * FROM inventory_item c ORDER BY (c.name, c.supplier_id, c.price);
```

（最后一种情况使用的是省略了关键字 ROW 的行构造器。）

另一种与复合值有关的特殊语法行为是，我们可以使用函数记法来提取复合值中的字段。简单来说，记法*field*(*table*) 和 *table*.*field* 可以互换。例如，这些查询是等价的：

```
SELECT c.name FROM inventory_item c WHERE c.price > 1000;
SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;
```

此外，如果我们有一个接受单个复合类型参数的函数，也可以用这两种记法来调用它。这些查询都等价：

```
SELECT somefunc(c) FROM inventory_item c;
SELECT somefunc(c.*) FROM inventory_item c;
SELECT c.somefunc FROM inventory_item c;
```

函数记法与字段记法之间的这种等价性，使得我们可以通过在复合类型上使用函数来实现“计算字段”。使用上面最后一种查询形式的应用程序，无需直接知道somefunc 并不是该表中的真实列。

> **提示**
>
> 由于这种行为，不宜让接受单个复合类型参数的函数与该复合类型中的任何字段同名。如果存在歧义，在使用字段名语法时会选择字段名解释，而在使用函数调用语法时会选择函数解释。不过，在 11 之前的PostgreSQL 版本中，除非调用的语法要求它必须是函数调用，否则总是选择字段名解释。在旧版本中，强制按函数解释的一种方法是为函数名加模式限定，也就是写成*schema*.*func*(*compositevalue*)。

8.16.6. 复合类型的输入和输出语法

复合值的外部文本表示由两部分组成：一部分是按照各字段类型的 I/O 转换规则解释的项，另一部分是表明复合结构的附加符号。这些附加符号包括包围整个值的圆括号（(和)），以及相邻项之间的逗号（,）。圆括号外部的空白会被忽略；但在圆括号内部，空白会被视为字段值的一部分，其是否有意义取决于该字段数据类型的输入转换规则。例如，在

```
'(  42)'
```

中，如果字段类型是 integer，则空白会被忽略；如果是 text，则不会被忽略。

如前所示，在写复合值时，你可以给任意单个字段值加双引号。如果字段值本身可能让复合值解析器混淆，则必须这样做。特别是，包含圆括号、逗号、双引号或反斜杠的字段必须用双引号括起来。要在带引号的复合字段值中写入双引号或反斜杠，需要在其前面加一个反斜杠。（另外，带双引号的字段值内部成对出现的双引号会被视为一个双引号字符，这与 SQL 字面字符串中单引号的规则类似。）或者，你也可以完全不使用引号，而改用反斜杠转义，保护所有原本会被当作复合语法的数据字符。

完全空的字段值（即逗号或圆括号之间一个字符也没有）表示 NULL。要写一个空字符串值而不是 NULL，可以写成""。

如果字段值是空字符串，或者包含圆括号、逗号、双引号、反斜杠或空白字符，复合类型输出例程就会在其周围加上双引号。（对空白字符这样做并非必需，但有助于提高可读性。）嵌入字段值中的双引号和反斜杠会被加倍。

注意

记住，你在 SQL 命令中写的内容会先被解释为字符串字面量，然后才会被解释为复合值。这会使所需的反斜杠数量翻倍（假定使用的是转义字符串语法）。例如，要在复合值中插入一个包含双引号和反斜杠的`text` 字段，需要写成：

```
INSERT ... VALUES ('("\"\\")');
```

字符串字面量处理器会去掉一层反斜杠，因此传到复合值解析器时看起来是`("\"\\")`。随后，送入`text` 数据类型输入例程的字符串就变成了`"\`。（如果我们使用的数据类型的输入例程也会把反斜杠当作特殊字符处理，例如`bytea`，那么为了在存储的复合字段中得到一个反斜杠，命令里可能需要多达八个反斜杠。）美元引用（见第 4.1.2.4 节）可用于避免反斜杠加倍的需要。

提示

在 SQL 命令中编写复合值时，`ROW` 构造器语法通常比复合字面量语法更容易使用。在 `ROW` 中，各个字段值的写法与它们不是复合成员时完全相同。

8.17. 范围类型

范围类型是表示某种元素类型的值范围的数据类型（该元素类型称为范围的子类型（subtype））。例如，`timestamp` 的范围可用于表示会议室被预订的时间范围。在这种情况下，数据类型为`tsrange`（“timestamp range” 的缩写），而`timestamp` 就是其子类型。子类型必须具有全序，这样才能明确定义元素值是位于某个值范围之内、之前还是之后。

范围类型很有用，因为它们能用单个范围值表示许多元素值，也能清晰地表达诸如范围重叠之类的概念。将时间和日期范围用于日程安排，是最清晰的例子；但价格区间、仪器的测量范围等场景也同样有用。

8.17.1. 内置范围类型

PostgreSQL 自带下列内置范围类型：

- `int4range` – `integer` 的范围
- `int8range` – `bigint` 的范围
- `numrange` – `numeric` 的范围
- `tsrange` – `timestamp without time zone` 的范围
- `tstzrange` – `timestamp with time zone` 的范围
- `daterange` – `date` 的范围

此外，你还可以定义自己的范围类型；详见 CREATE TYPE。

8.17.2. 示例

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- 包含关系
SELECT int4range(10, 20) @> 3;

-- 重叠
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- 提取上界
SELECT upper(int8range(15, 25));

-- 计算交集
SELECT int4range(10, 20) * int4range(15, 25);

-- 范围是否为空?
SELECT isempty(numrange(1, 5));
```

范围类型上的操作符和函数的完整列表见表 9.50 和表 9.51。

8.17.3. 包含界限与排除界限

每个非空范围都有两个界限：下界和上界。位于这两个值之间的所有点都包含在该范围内。包含界限表示边界点本身也包含在范围内，而排除界限则表示边界点不包含在范围内。

在范围的文本形式中，包含下界用“`[`”表示，排除下界用“`(`”表示。同样，包含上界用“`]`”表示，排除上界用“`)`”表示。更多细节见第 8.17.5 节。

函数 `lower_inc` 和 `upper_inc` 分别测试范围值的下界和上界是否包含在内。

8.17.4. 无限（无界）范围

范围的下界可以省略，这意味着所有小于上界的值都包含在范围内，例如`(,3]`。同样，如果省略范围的上界，则所有大于下界的值都包含在范围内。如果上下界都被省略，则该元素类型的所有值都被认为处于该范围内。把缺失的界限指定为包含，会自动转换为排除，例如`[,]` 会转换为`(,)`。你可以把这些缺失的值看作正负无穷大，但它们是特殊的范围类型值，并且被认为超出了任何范围元素类型的正负无穷大值。

具有“无穷大”概念的元素类型可以将其用作显式界限值。例如，对于时间戳范围，`[today,infinity)` 不包括特殊的 `timestamp` 值 `infinity`，而`[today,infinity]` 则包括它，`[today,)` 和`[today,]` 也一样。

函数 `lower_inf` 和 `upper_inf` 分别测试范围是否没有下界和上界。

8.17.5. 范围输入/输出

范围值的输入必须遵循下列模式之一：

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

如前所述，圆括号或方括号指示上下界是排除还是包含。注意最后一种模式是`empty`，它表示空范围（即不包含任何点的范围）。

lower-bound 可以是子类型的合法输入字符串，也可以留空以表示没有下界。同样，*upper-bound* 也可以是子类型的合法输入字符串，或者留空以表示没有上界。

每个界限值都可以用"（双引号）字符括起来。如果界限值包含圆括号、方括号、逗号、双引号或反斜线，这样做就是必须的，因为否则这些字符会被视为范围语法的一部分。要在带引号的界限值中写入双引号或反斜线，需要在前面加一个反斜线。（另外，在双引号括起来的界限值中，两个连续的双引号表示一个双引号字符，这与 SQL 字符串字面量中单引号的规则类似。）或者，你也可以不使用引号，而是用反斜线转义所有本来会被当作范围语法的字符。另外，如果要把空字符串写成界限值，应写成""，因为什么都不写表示无限界限。

范围值前后允许有空白，但圆括号或方括号之间的任何空白都会被视为下界或上界值的一部分。（取决于元素类型，这些空白可能有意义，也可能没有意义。）

> **注意**
>
> 这些规则与在复合类型字面量中书写字段值的规则非常相似。更多说明见第 8.16.6 节。

示例：

```
-- 包含 3，不包含 7，并包含中间的所有点
SELECT '[3,7)'::int4range;

-- 既不包含 3，也不包含 7，但包含中间的所有点
SELECT '(3,7)'::int4range;

-- 只包含单个点 4
SELECT '[4,4]'::int4range;

-- 不包含任何点（并将被规范化为 'empty'）
SELECT '[4,4)'::int4range;
```

8.17.6. 构造范围

每种范围类型都有一个与范围类型同名的构造函数。使用构造函数通常比书写范围字面量更方便，因为这样无需为界限值额外加引号。构造函数接受两个或三个参数。两个参数的形式构造标准形式的范围（下界包含，上界排除），而三个参数的形式则按第三个参数指定的界限形式构造范围。第三个参数必须是下列字符串之一：“()”、“(]”、“[)”或者“[]”。例如：

```
-- 完整形式为：下界、上界，以及指示界限包含性/排除性的文本参数。
SELECT numrange(1.0, 14.0, '(]');

-- 如果省略第三个参数，则假定为 '[)'。
SELECT numrange(1.0, 14.0);

-- 虽然这里指定的是 '(]'，但显示时该值会转换为规范形式，因为 int8range 是离散范围类型（见下文）。
SELECT int8range(1, 14, '(]');

-- 任一界限使用 NULL 都会使该侧无界。
SELECT numrange(NULL, 2.2);
```

8.17.7. 离散范围类型

离散范围是指其元素类型具有明确定义的“步长”，例如integer 或 date。在这类类型中，如果两个元素之间不存在合法值，就可以说它们是相邻的。这与连续范围形成对比：在连续范围中，两个给定值之间总是（或几乎总是）可以识别出其他元素值。例如，基于numeric 的范围是连续的，基于timestamp 的范围也是如此。（尽管 timestamp 的精度有限，因此理论上可以视为离散的，但通常最好仍将其视为连续的，因为步长通常并不是关注点。）

理解离散范围类型的另一种方式是：对每个元素值，都有一个明确的“下一个”或“上一个”值。知道这一点后，就可以把原先给定的元素值替换成其下一个或上一个值，从而在范围界限的包含表示和排除表示之间相互转换。例如，在整数范围类型中，[4,8] 和(3,9) 表示相同的值集合；但对基于 numeric 的范围则并非如此。

离散范围类型应当具有一个规范化函数，该函数知道元素类型所期望的步长。规范化函数负责将范围类型中的等价值转换成完全一致的表示形式，特别是对包含或排除界限保持一致。如果未指定规范化函数，那么格式不同的范围即使实际上表示的是同一组值，也总会被视为不相等。

内置范围类型 int4range、int8range 和 daterange 都使用一种规范形式：包含下界而排除上界，也就是[)。不过，用户定义的范围类型也可以采用其他约定。

8.17.8. 定义新的范围类型

用户可以定义自己的范围类型。最常见的原因，是希望对内置范围类型中未提供的子类型使用范围。例如，要定义一个子类型为float8 的新范围类型：

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

因为 float8 没有有意义的“步长”，所以在这个示例中我们没有定义规范化函数。

定义自己的范围类型还允许你指定不同的子类型 B-树操作符类或排序规则，从而改变用于判定哪些值落入给定范围的排序顺序。

如果认为子类型的值是离散而非连续的，则CREATE TYPE 命令应指定 canonical 函数。规范化函数接受一个输入的范围值，并必须返回一个等价的范围值，该值的界限和格式可能不同。对于表示同一组值的两个范围，例如整数范围[1, 7] 和[1, 8)，规范化输出必须相同。选择哪一种表示作为规范形式并不重要，只要不同格式的两个等价值总是会被映射为同一种格式的相同值即可。除了调整界限的包含/排除格式之外，如果期望的步长大于子类型能够存储的精度，规范化函数还可能对界限值进行舍入。例如，基于timestamp 的范围类型可以定义为步长为一小时，在这种情况下，规范化函数需要把不是一小时整数倍的界限值舍入，或者直接抛出错误。

此外，任何打算与 GiST 或 SP-GiST 索引一起使用的范围类型都应定义子类型差值函数，即 subtype_diff 函数。（没有 subtype_diff 时索引仍可工作，但效率很可能明显低于提供了差值函数的情况。）子类型差值函数接受两个子类型输入值，并返回它们的差值（即*X* 减 *Y*），结果表示为一个float8 值。在上面的示例中，可以使用函数float8mi，它是常规float8 减法操作符的底层实现；但对于其他子类型，则需要某种类型转换。此外，可能还需要仔细考虑如何把差异表示为数字。尽可能地，subtype_diff 函数应与所选操作符类和排序规则所隐含的排序顺序一致；也就是说，只要其第一个参数按该排序顺序大于第二个参数，它的结果就应该为正。

下面给出一个不那么简化的 subtype_diff 函数示例：

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);

SELECT '[11:10, 23:00]'::timerange;
```

关于创建范围类型的更多信息，见 CREATE TYPE。

8.17.9. 索引

可以为范围类型的表列创建 GiST 和 SP-GiST 索引。例如，创建 GiST 索引：

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

GiST 或 SP-GiST 索引可以加速涉及以下范围操作符的查询：=、&&、<@、@>、<<、>>、-|-、&<和&>（参见表 9.50 了解更多信息）。

此外，也可以为范围类型的表列创建 B-树和 hash 索引。对于这些索引类型，基本上唯一有用的范围操作就是等值。系统为范围值定义了对应<和> 操作符的 B-树排序顺序，但这种顺序相当任意，在现实中通常并没有什么用处。范围类型的 B-树和 hash 支持主要是为了允许在查询内部进行排序和 hash 运算，而不是用于创建实际的索引。

8.17.10. 范围上的约束

虽然 UNIQUE 是标量值的一种自然约束，但它通常并不适用于范围类型。相反，排他约束往往更合适（见CREATE TABLE ... CONSTRAINT ... EXCLUDE）。排他约束允许在范围类型上指定诸如“不重叠”之类的约束。例如：

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

该约束会阻止表中同时存在任何重叠值：

```
INSERT INTO reservation VALUES
    ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
    ('[2010-01-01 14:45, 2010-01-01 15:45)');
ERROR:  conflicting key value violates exclusion constraint "reservation_during_excl"
DETAIL:  Key (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00")) conflicts
with existing key (during)=(["2010-01-01 11:30:00","2010-01-01 15:00:00")).
```

你可以使用 btree_gist 扩展在普通标量数据类型上定义排他约束，然后再将其与范围上的排他约束结合起来，以获得最大的灵活性。例如，在安装了btree_gist 之后，下面的约束仅在会议室编号相等时才会拒绝重叠的范围：

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR:  conflicting key value violates exclusion constraint "room_reservation_room_during_excl"
DETAIL:  Key (room, during)=(123A, ["2010-01-01 14:30:00","2010-01-01 15:30:00")) conflicts
with existing key (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01 15:00:00")).

INSERT INTO room_reservation VALUES
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

8.18. 域类型

域是一种用户定义的数据类型，它基于另一种底层类型。根据需要，它可以有约束来限制其有效值为底层类型所允许值的一个子集。除此之外，它的行为与底层类型相同 – 例如，任何适用于底层类型的操作符或函数都对该域类型有效。底层类型可以是任何内置或者用户定义的基础类型、枚举类型、数组类型、复合类型、范围类型或者另一个域。

例如，我们可以基于整数创建一个只接受正整数的域：

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);
CREATE TABLE mytable (id posint);
INSERT INTO mytable VALUES(1);   -- 成功
INSERT INTO mytable VALUES(-1);  -- 失败
```

当底层类型的一个操作符或函数被应用于一个域值时，域会被自动向下转换为底层类型。因此，mytable.id - 1 的结果会被认为是类型 integer 而不是 posint。我们可以写成 (mytable.id - 1)::posint 来把结果转换回 posint，这会导致域的约束被重新检查。在这种情况下，如果该表达式被应用于一个值为 1 的id 就会报错。把底层类型的值赋给域类型的一个字段或者变量不需要写显式的类型转换，但是域的约束将会被检查。

更多信息请参考 CREATE DOMAIN。

8.19. 对象标识符类型

对象标识符（OID）在 PostgreSQL 内部被用作若干系统表的主键。除非建表时指定 WITH OIDS，或启用配置变量 default_with_oids，否则不会向用户创建的表添加 OID。类型oid 表示一个对象标识符。还有若干 oid 的别名类型：regproc、regprocedure、regoper、regoperator、regclass、regtype、regrole、regnamespace、regconfig 和 regdictionary。表 8.24 给出了概要说明。

oid 类型目前实现为无符号 4 字节整数。因此，在大型数据库中，它不足以提供数据库范围内的唯一性，甚至在大型的单个表中也无法保证唯一性。因此，不推荐把用户创建表的 OID 列用作主键。OID 最好仅用于引用系统表。

oid 类型本身除比较之外几乎没有其他操作。不过，它可以转换为整数，然后再用标准整数操作符进行处理。（这样做时要注意有符号与无符号可能造成的混淆。）

OID 别名类型除了专门的输入和输出例程外，没有自己的操作。这些例程能够接受并显示系统对象的符号名称，而不是 oid 类型所使用的原始数值。别名类型简化了对象 OID 值的查找。例如，要查看与表 mytable 相关的 pg_attribute 行，可以写成：

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

而不是：

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

虽然这样看起来也不算太糟，但其实仍然过于简化。如果不同模式中有多个名为 mytable 的表，就必须写一个复杂得多的子查询来选出正确的 OID。regclass 的输入转换器会按照模式路径设置来处理表查找，因此能够自动完成“正确的事情”。同样，把表的 OID 转换成 regclass，也很适合用来以符号形式显示数值 OID。

表 8.24. 对象标识符类型

名字	引用	描述	值示例
oid	任意	数字形式的对象标识符	564182
regproc	pg_proc	函数名称	sum

名字	引用	描述	值示例
regprocedure	pg_proc	带参数类型的函数	sum(int4)
regoper	pg_operator	操作符名称	+
regoperator	pg_operator	带参数类型的操作符	*(integer,integer) 或-(NONE,integer)
regclass	pg_class	关系名称	pg_type
regtype	pg_type	数据类型名称	integer
regrole	pg_authid	角色名	smithee
regnamespace	pg_namespace	命名空间名称	pg_catalog
regconfig	pg_ts_config	文本检索配置	english
regdictionary	pg_ts_dict	文本检索词典	simple

对于按命名空间分组的对象，所有 OID 别名类型都接受模式限定名称；如果不加限定就无法在当前搜索路径中找到对象，输出时也会显示模式限定名称。regproc 和 regoper 别名类型只接受唯一的（未重载的）输入名称，因此用途有限；对于大多数用途，regprocedure 或 regoperator 更合适。对于 regoperator，通过将未使用的操作数写为NONE 来标识一元操作符。

大多数 OID 别名类型还有一个附加特性，就是会创建依赖关系。如果这些类型的常量出现在存储的表达式中（例如列默认表达式或视图），它就会对被引用对象建立依赖。例如，如果某列的默认表达式是 nextval('my_seq'::regclass)，PostgreSQL 就会知道该默认表达式依赖于序列 my_seq，因此系统不允许在尚未移除该默认表达式时删除序列。（regrole 是这一特性的唯一例外：该类型的常量不允许出现在存储表达式中。）

> **注意**
>
> OID 别名类型并不完全遵循事务隔离规则。规划器也会把它们视为简单常量，这可能导致次优的规划。

系统使用的另一种标识符类型是 xid，即事务（缩写为xact）标识符。这是系统列 xmin 和 xmax 的数据类型。事务标识符是 32 位的量。

系统使用的第三种标识符类型是 cid，也就是命令标识符。这是系统列 cmin 和 cmax 的数据类型。命令标识符同样是 32 位量。

系统使用的最后一种标识符类型是 tid，也就是元组标识符（行标识符）。这是系统列 ctid 的数据类型。一个元组 ID 是一对值（块号、块内元组索引），用来标识该行在其所属表中的物理位置。

（系统列的更多说明见第 5.4 节。）

8.20. pg_lsn 类型

pg_lsn 数据类型可用于存储 LSN（日志序列号）数据。LSN 是指向 WAL 中某个位置的指针。该类型是 XLogRecPtr 的一种表示形式，也是 PostgreSQL 的内部系统类型之一。

在内部，LSN 是一个 64 位整数，表示预写式日志流中的字节位置。它打印为两个最多 8 位的十六进制数，中间以斜线分隔，例如 16/B374D848。pg_lsn 类型支持标准比较操作符，如 = 和 >。两个 LSN 可以用 - 操作符相减，结果是这两个预写式日志位置之间相隔的字节数。

8.21. 伪类型

PostgreSQL 的类型系统中包含一些特殊用途的条目，统称为伪类型。伪类型不能作为列数据类型使用，但可以用于声明函数的参数类型或结果类型。每一种伪类型都适用于这样的场景：函数的行为并不对应于简单地接受或返回某个特定 SQL 数据类型的值。表 8.25 列出了现有的伪类型。

表 8.25. 伪类型

名字	描述
any	表示一个函数可以接受任意输入数据类型。
anyelement	表示一个函数可以接受任意数据类型（参见第 38.2.5 节）。
anyarray	表示一个函数可以接受任意数组数据类型（参见第 38.2.5 节）。
anynonarray	表示一个函数可以接受任意非数组数据类型（参见第 38.2.5 节）。
anyenum	表示一个函数可以接受任意枚举数据类型（参见第 38.2.5 节和第 8.7 节）。
anyrange	表示一个函数可以接受任意范围数据类型（参见第 38.2.5 节和第 8.17 节）。
cstring	表示一个函数接受或返回一个以空字符结尾的 C 字符串。
internal	表示一个函数接受或返回一个服务器内部数据类型。
language_handler	表示过程语言调用处理器被声明为返回 language_handler。
fdw_handler	表示外部数据包装器处理器被声明为返回 fdw_handler。
index_am_handler	表示索引访问方法处理器被声明为返回 index_am_handler。
tsm_handler	表示表采样方法处理器被声明为返回 tsm_handler。
record	标识一个接受或返回未指定行类型的函数。
trigger	表示触发器函数被声明为返回 trigger。
event_trigger	表示事件触发器函数被声明为返回 event_trigger。
pg_ddl_command	标识一种对事件触发器可用的 DDL 命令的表达。
void	表示一个函数不返回值。
unknown	标识一种尚未解析出的类型，例如未修饰的字符串字面值的类型。
opaque	一个已废弃的类型名称，以前用于上述多种用途。

用 C 编写的函数（无论是内置的还是动态装载的）都可以声明为接受或返回这些伪类型中的任意一种。函数作者必须自行确保，当伪类型被用作参数类型时，该函数的行为仍然安全。

用过程语言编写的函数，只有在其实现语言允许的情况下才能使用伪类型。目前，大多数过程语言都禁止把伪类型用作参数类型，而只允许把void 和 record 用作结果类型（如果函数被用作触发器或事件触发器，则也允许trigger 或 event_trigger）。有些语言还支持使用 anyelement、anyarray、anynonarray、anyenum 和 anyrange 实现多态函数。

`internal` 伪类型用于声明那些只应由数据库系统内部调用、而不应被 SQL 查询直接调用的函数。如果一个函数至少有一个 `internal` 类型参数，那么它就不能从 SQL 中调用。为了保持这一限制的类型安全性，必须遵守这样一条编码规则：除非函数至少有一个 `internal` 类型参数，否则不要声明任何返回 `internal` 的函数。

第 9 章 函数和操作符

PostgreSQL 为内置数据类型提供了大量函数和操作符。用户也可以按第 V 部分中的说明定义自己的函数和操作符。psql 命令`\df` 和`\do` 可分别用于列出所有可用的函数和操作符。

如果你关心可移植性，请注意，本章描述的大多数函数和操作符，除了最简单的算术和比较操作符以及少数明确标记的函数之外，都不是SQL 标准规定的。不过，其中一些扩展功能也存在于其他SQL 数据库管理系统中，而且在很多情况下，各实现之间的这类功能是兼容且一致的。本章也并非详尽无遗；手册的相关章节中还有其他函数。

9.1. 逻辑操作符

常用的逻辑操作符有：

```
AND
OR
NOT
```

SQL 使用三值的逻辑系统，包括真、假和`null`，它表示“未知”。观察下面的真值表：

a	*b*	*a* **AND** *b*	*a* **OR** *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	**NOT** *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

操作符 `AND` 和 `OR` 是可交换的，也就是说，你可以交换左右操作数而不影响结果。但请参见第 4.2.14 节获取有关子表达式计算顺序的更多信息。

9.2. 比较函数和操作符

常见的比较操作符都可用，如表 9.1 所示。

表 9.1. 比较操作符

操作符	描述
<	小于
>	大于
<=	小于或等于
>=	大于或等于
=	等于

操作符	描述
<> 或 !=	不等于

> **注意**
>
> != 操作符在解析器阶段会转换为 <>。不可能实现行为不同的 != 和 <> 操作符。

比较操作符适用于所有相关的数据类型。所有比较操作符都是返回 boolean 类型值的二元操作符；类似 1 < 2 < 3 的表达式是无效的（因为没有用于比较布尔值和 3 的 < 操作符）。

如表 9.2 所示，也有一些比较谓词。它们的行为和操作符很像，但是具有 SQL 标准所要求的特殊语法。

表 9.2. 比较谓词

谓词	描述
a BETWEEN *x* AND *y*	位于两者之间
a NOT BETWEEN *x* AND *y*	不位于两者之间
a BETWEEN SYMMETRIC *x* AND *y*	对比较值排序后，位于两者之间
a NOT BETWEEN SYMMETRIC *x* AND *y*	对比较值排序后，不位于两者之间
a IS DISTINCT FROM *b*	不等于，将 null 视为普通值
a IS NOT DISTINCT FROM *b*	等于，将 null 视为普通值
expression IS NULL	为 null
expression IS NOT NULL	不为 null
expression ISNULL	为 null（非标准语法）
expression NOTNULL	不为 null（非标准语法）
boolean_expression IS TRUE	为真
boolean_expression IS NOT TRUE	为假或未知
boolean_expression IS FALSE	为假
boolean_expression IS NOT FALSE	为真或未知
boolean_expression IS UNKNOWN	为未知
boolean_expression IS NOT UNKNOWN	为真或假

该 BETWEEN 谓词可以简化范围测试：

```
a BETWEEN x AND y
```

等价于

```
a >= x AND a <= y
```

注意，BETWEEN 将两个端点值都视为包含在范围内。NOT BETWEEN 执行相反的比较：

```
a NOT BETWEEN x AND y
```

等价于

```
a < x OR a > y
```

BETWEEN SYMMETRIC 类似于 BETWEEN，但不要求AND 左侧的参数小于或等于右侧的参数。如果不是这样，这两个参数会自动交换，以确保表示的范围始终非空。

当任一输入为 null 时，普通的比较操作符会得到 null（表示“未知”），而不是真或假。例如，7 = NULL 得到 null，7 <> NULL 也一样。如果这种行为不合适，可以使用IS [NOT] DISTINCT FROM 谓词：

```
a IS DISTINCT FROM b
a IS NOT DISTINCT FROM b
```

对于非 null 输入，IS DISTINCT FROM 和<> 操作符一样。不过，如果两个输入都为 null，它会返回假。而如果只有一个输入为 null，它会返回真。类似地，IS NOT DISTINCT FROM 对于非 null 输入的行为与=相同，但是当两个输入都为 null 时它返回真，并且当只有一个输入为 null 时返回假。因此，这些谓词实际上把 null 当作一种普通数据值，而不是“未知”。

要检查一个值是否为 null，使用下面的谓词：

```
expression IS NULL
expression IS NOT NULL
```

或者等效但非标准的谓词：

```
expression ISNULL
expression NOTNULL
```

不要写 *expression* = NULL，因为NULL 并不“等于”NULL。（null 值表示未知值，而我们并不知道两个未知值是否相等。）

> **提示**
>
> 有些应用可能期望表达式 *expression* = NULL 在 *expression* 求值为 null 时返回真。我们强烈建议此类应用修改为遵循 SQL 标准。但是，如果无法这样修改，那么可以使用配置变量 transform_null_equals。如果将其打开，PostgreSQL 会把 x = NULL 子句转换成 x IS NULL。

如果 *expression* 是行值，那么当行表达式本身为 null 或其所有字段都为 null 时，IS NULL 为真；而当行表达式本身非 null 且其所有字段都非 null 时，IS NOT NULL 为真。由于这种行为，IS NULL 和 IS NOT NULL 并不总是对行值表达式返回相反的结果；特别是，一个同时包含 null 和非 null 字段的行值表达式会对这两种测试都返回假。在某些情况下，写成 *row* IS DISTINCT FROM NULL 或者 *row* IS NOT DISTINCT FROM NULL 可能更合适，因为它们只会检查整个行值是否为 null，而不会再对行字段做额外测试。

布尔值也可以使用下列谓词进行测试：

```
boolean_expression IS TRUE
boolean_expression IS NOT TRUE
boolean_expression IS FALSE
boolean_expression IS NOT FALSE
boolean_expression IS UNKNOWN
boolean_expression IS NOT UNKNOWN
```

这些谓词总是返回真或假，从不返回空值，即使操作数为 null 也是如此。null 输入被当作逻辑值“未知”。请注意，IS UNKNOWN 和 IS NOT UNKNOWN 实际上分别等同于 IS NULL 和 IS NOT NULL，只是输入表达式必须是布尔类型。

如表 9.3 中所示，也有一些比较相关的函数可用。

表 9.3. 比较函数

函数	描述	示例	示例结果
num_nonnulls(VARIADIC "any")	返回非 null 参数的数量	num_nonnulls(1, NULL, 2)	2
num_nulls(VARIADIC "any")	返回 null 参数的数量	num_nulls(1, NULL, 2)	1

9.3. 数学函数和操作符

PostgreSQL 为很多类型提供了数学操作符。对于那些没有标准数学惯例的类型（如日期/时间类型），我们将在后续小节中描述实际的行为。

表 9.4 列出了可用的数学操作符。

表 9.4. 数学操作符

操作符	描述	示例	结果
+	加法	2 + 3	5
-	减法	2 - 3	-1
*	乘法	2 * 3	6

操作符	描述	示例	结果
/	除法（整数除法会截断结果）	4 / 2	2
%	取模（余数）	5 % 4	1
^	求幂（从左向右结合）	2.0 ^ 3.0	8
|/	平方根	|/ 25.0	5
||/	立方根	||/ 27.0	3
!	阶乘（已弃用，改用 factorial()）	5 !	120
!!	以前缀操作符形式求阶乘（已弃用，改用 factorial()）	!! 5	120
@	绝对值	@ -5.0	5
&	按位与	91 & 15	11
|	按位或	32 | 3	35
#	按位异或	17 # 5	20
~	按位非	~1	-2
<<	按位左移	1 << 4	16
>>	按位右移	8 >> 2	2

按位操作符仅适用于整数数据类型，也可用于位串类型 bit 和 bit varying，如表 9.13 所示。

表 9.5 列出了可用的数学函数。表中的 dp 表示 double precision。许多函数提供了参数类型不同的多种形式。除非另有说明，函数的每种形式都返回与其参数相同的数据类型。处理 double precision 数据的函数大多基于主机系统的 C 库实现；因此，其精度和边界情况下的行为可能因主机系统而异。

表 9.5. 数学函数

函数	abs(*x*)
返回类型	（与输入相同）
描述	绝对值
示例	abs(-17.4)
结果	17.4
函数	cbrt(dp)
返回类型	dp
描述	立方根
示例	cbrt(27.0)
结果	3
函数	ceil(dp or numeric)
返回类型	（与输入相同）
描述	大于或等于参数的最小整数

示例	`ceil(-42.8)`
结果	`-42`
函数	`ceiling(dp or numeric)`
返回类型	（与输入相同）
描述	大于或等于参数的最小整数（与 `ceil` 相同）
示例	`ceiling(-95.3)`
结果	`-95`
函数	`degrees(dp)`
返回类型	`dp`
描述	将弧度转换为角度
示例	`degrees(0.5)`
结果	`28.6478897565412`
函数	`div(y numeric, x numeric)`
返回类型	`numeric`
描述	*y*/*x* 的整数商
示例	`div(9,4)`
结果	`2`
函数	`exp(dp or numeric)`
返回类型	（与输入相同）
描述	指数函数
示例	`exp(1.0)`
结果	`2.71828182845905`
函数	`factorial(bigint)`
返回类型	`numeric`
描述	阶乘
示例	`factorial(5)`
结果	`120`
函数	`floor(dp or numeric)`
返回类型	（与输入相同）
描述	小于或等于参数的最大整数
示例	`floor(-42.8)`
结果	`-43`
函数	`ln(dp or numeric)`
返回类型	（与输入相同）
描述	自然对数

示例	ln(2.0)
结果	0.693147180559945
函数	log(dp or numeric)
返回类型	（与输入相同）
描述	以 10 为底的对数
示例	log(100.0)
结果	2
函数	log(*b* numeric, *x* numeric)
返回类型	numeric
描述	以 *b* 为底的对数
示例	log(2.0, 64.0)
结果	6.0000000000
函数	mod(*y*, *x*)
返回类型	（与参数类型相同）
描述	*y*/*x* 的余数
示例	mod(9,4)
结果	1
函数	pi()
返回类型	dp
描述	常数 “π”
示例	pi()
结果	3.14159265358979
函数	power(*a* dp, *b* dp)
返回类型	dp
描述	*a* 的*b* 次幂
示例	power(9.0, 3.0)
结果	729
函数	power(*a* numeric, *b* numeric)
返回类型	numeric
描述	*a* 的*b* 次幂
示例	power(9.0, 3.0)
结果	729
函数	radians(dp)
返回类型	dp

描述	将角度转换为弧度
示例	`radians(45.0)`
结果	`0.785398163397448`
函数	`round(dp or numeric)`
返回类型	（与输入相同）
描述	舍入到最接近的整数
示例	`round(42.4)`
结果	`42`
函数	`round(v numeric, s int)`
返回类型	`numeric`
描述	舍入到 *s* 位小数
示例	`round(42.4382, 2)`
结果	`42.44`
函数	`scale(numeric)`
返回类型	`integer`
描述	参数的小数位数（小数部分的十进制位数）
示例	`scale(8.41)`
结果	`2`
函数	`sign(dp or numeric)`
返回类型	（与输入相同）
描述	参数的符号（-1、0、+1）
示例	`sign(-8.4)`
结果	`-1`
函数	`sqrt(dp or numeric)`
返回类型	（与输入相同）
描述	平方根
示例	`sqrt(2.0)`
结果	`1.4142135623731`
函数	`trunc(dp or numeric)`
返回类型	（与输入相同）
描述	向零截断
示例	`trunc(42.8)`
结果	`42`
函数	`trunc(v numeric, s int)`
返回类型	`numeric`

描述	截断到 *s* 位小数
示例	`trunc(42.4382, 2)`
结果	`42.43`
函数	`width_bucket(operand dp, b1 dp, b2 dp, count int)`
返回类型	`int`
描述	返回 *operand* 在直方图中所属的桶编号；该直方图将 *b1* 至 *b2* 的范围划分为 *count* 个等宽桶。对于范围之外的输入，返回 `0` 或 *count*+1
示例	`width_bucket(5.35, 0.024, 10.06, 5)`
结果	`3`
函数	`width_bucket(operand numeric, b1 numeric, b2 numeric, count int)`
返回类型	`int`
描述	返回 *operand* 在直方图中所属的桶编号；该直方图将 *b1* 至 *b2* 的范围划分为 *count* 个等宽桶。对于范围之外的输入，返回 `0` 或 *count*+1
示例	`width_bucket(5.35, 0.024, 10.06, 5)`
结果	`3`
函数	`width_bucket(operand anyelement, thresholds anyarray)`
返回类型	`int`
描述	根据列出各桶下界的数组，返回 *operand* 所属的桶编号；对于小于第一个下界的输入，返回 `0`。*thresholds* 数组必须按升序排序，否则将产生意外结果
示例	`width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])`
结果	`2`

表 9.6 展示了用于产生随机数的函数。

表 9.6. 随机函数

函数	返回类型	描述
`random()`	`dp`	范围 0.0 <= x < 1.0 内的随机值
`setseed(dp)`	`void`	为后续 `random()` 调用设置种子（值在 -1.0 与 1.0 之间，包含端点）

`random()` 返回值的特性取决于系统实现。它不适用于加密应用；替代方案见 pgcrypto 模块。

最后，表 9.7 列出了可用的三角函数。所有三角函数的参数和返回值都是 `double precision` 类型。每个三角函数都有两种变体，一种以弧度度量角，另一种以角度度量角。

表 9.7. 三角函数

函数（弧度）	函数（角度）	描述
`acos(x)`	`acosd(x)`	反余弦
`asin(x)`	`asind(x)`	反正弦

函数（弧度）	函数（角度）	描述
atan(*x*)	atand(*x*)	反正切
atan2(*y*, *x*)	atan2d(*y*, *x*)	*y*/*x* 的反正切
cos(*x*)	cosd(*x*)	余弦
cot(*x*)	cotd(*x*)	余切
sin(*x*)	sind(*x*)	正弦
tan(*x*)	tand(*x*)	正切

注意

另一种使用以角度度量的角的方法是使用早前展示的单位转换函数 radians() 和 degrees()。不过，使用基于角度的三角函数更好，因为这类方法能避免sind(30) 等特殊情况下的舍入误差。

9.4. 字符串函数和操作符

本节描述用于检查和操作字符串值的函数和操作符。这里的字符串包括 character、character varying 和 text 类型的值。除非另有说明，下面列出的所有函数都适用于这些类型，但使用 character 类型时，应注意自动填充空格可能带来的影响。一些函数也为位串类型提供了原生实现。

SQL 定义了一些使用关键字而不是逗号来分隔参数的字符串函数。详情请见表 9.8。PostgreSQL 也提供了这些函数使用常规函数调用语法的版本（参阅表 9.9）。

注意

在 PostgreSQL 8.3 之前，由于某些非字符串数据类型到 text 的隐式强制转换，这些函数也会悄然接受那些类型的值。由于这些强制转换经常产生出人意料的行为，它们已被移除。不过，字符串串接操作符（||）仍接受非字符串输入，只要至少一个输入属于字符串类型，如表 9.8 所示。对于其他情况，如果需要重现以前的行为，可添加到 text 的显式强制转换。

表 9.8. SQL 字符串函数和操作符

函数	*string* || *string*
返回类型	text
描述	字符串串接
示例	'Post' || 'greSQL'
结果	PostgreSQL
函数	*string* || *non-string* 或 *non-string* || *string*
返回类型	text
描述	一个输入为非字符串的字符串串接
示例	'Value: ' || 42

结果	`Value: 42`
函数	`bit_length(`*`string`*`)`
返回类型	`int`
描述	字符串中的位数
示例	`bit_length('jose')`
结果	`32`
函数	`char_length(`*`string`*`)` 或 `character_length(`*`string`*`)`
返回类型	`int`
描述	字符串中的字符数
示例	`char_length('jose')`
结果	`4`
函数	`lower(`*`string`*`)`
返回类型	`text`
描述	将字符串转换为小写
示例	`lower('TOM')`
结果	`tom`
函数	`octet_length(`*`string`*`)`
返回类型	`int`
描述	字符串中的字节数
示例	`octet_length('jose')`
结果	`4`
函数	`overlay(`*`string`* `placing` *`string`* `from int [for int])`
返回类型	`text`
描述	替换子字符串
示例	`overlay('Txxxxas' placing 'hom' from 2 for 4)`
结果	`Thomas`
函数	`position(`*`substring`* `in` *`string`*`)`
返回类型	`int`
描述	指定子字符串的位置
示例	`position('om' in 'Thomas')`
结果	`3`
函数	`substring(`*`string`* `[from int] [for int])`
返回类型	`text`
描述	提取子字符串

示例	`substring('Thomas' from 2 for 3)`
结果	`hom`
函数	`substring(`*`string`* `from` *`pattern`*`)`
返回类型	`text`
描述	提取匹配 POSIX 正则表达式的子字符串。有关模式匹配的更多信息，参见第 9.7 节。
示例	`substring('Thomas' from '...$')`
结果	`mas`
函数	`substring(`*`string`* `from` *`pattern`* `for` *`escape`*`)`
返回类型	`text`
描述	提取匹配 SQL 正则表达式的子字符串。有关模式匹配的更多信息，参见第 9.7 节。
示例	`substring('Thomas' from '%#"o_a#"_' for '#')`
结果	`oma`
函数	`trim([leading | trailing | both] [`*`characters`*`] from` *`string`*`)`
返回类型	`text`
描述	从 *`string`* 的开头、结尾或两端（默认为 `both`）移除仅由 *`characters`* 中字符（默认为空格）组成的最长字符串
示例	`trim(both 'xyz' from 'yxTomxx')`
结果	`Tom`
函数	`trim([leading | trailing | both] [from]` *`string`* `[,` *`characters`*`] )`
返回类型	`text`
描述	`trim()` 的非标准语法
示例	`trim(both from 'yxTomxx', 'xyz')`
结果	`Tom`
函数	`upper(`*`string`*`)`
返回类型	`text`
描述	将字符串转换为大写
示例	`upper('tom')`
结果	`TOM`

还有其他字符串操作函数，列在表 9.9 中。其中一些用于内部实现表 9.8 中列出的 SQL 标准字符串函数。

表 9.9. 其他字符串函数

函数	`ascii(`*`string`*`)`
返回类型	`int`
描述	参数首字符的 ASCII 码。对于 UTF8，返回该字符的 Unicode 码点。对于其他多字节编码，参数必须是 ASCII 字符。

示例	`ascii('x')`
结果	`120`
函数	`btrim(string text [, characters text])`
返回类型	`text`
描述	从 *string* 的开头和结尾移除仅由 *characters* 中字符（默认为空格）组成的最长字符串
示例	`btrim('xyxtrimyyx', 'xyz')`
结果	`trim`
函数	`chr(int)`
返回类型	`text`
描述	指定编码值对应的字符。对于 UTF8，参数被视为 Unicode 码点。对于其他多字节编码，参数必须指定一个 ASCII 字符。不允许 NULL（0）字符，因为文本数据类型无法存储这样的字节。
示例	`chr(65)`
结果	`A`
函数	`concat(str "any" [, str "any" [, ...] ])`
返回类型	`text`
描述	串接所有参数的文本表示。忽略 NULL 参数。
示例	`concat('abcde', 2, NULL, 22)`
结果	`abcde222`
函数	`concat_ws(sep text, str "any" [, str "any" [, ...] ])`
返回类型	`text`
描述	使用分隔符串接除第一个参数以外的所有参数。第一个参数用作分隔字符串。忽略 NULL 参数。
示例	`concat_ws(',', 'abcde', 2, NULL, 22)`
结果	`abcde,2,22`
函数	`convert(string bytea, src_encoding name, dest_encoding name)`
返回类型	`bytea`
描述	将字符串转换为 *dest_encoding*。原编码由 *src_encoding* 指定。*string* 必须是该编码下的有效字符串。可以用 `CREATE CONVERSION` 定义转换。此外，也有一些预定义的转换。可用的转换见表 9.10。
示例	`convert('text_in_utf8', 'UTF8', 'LATIN1')`
结果	以 Latin-1 编码（ISO 8859-1）表示的 `text_in_utf8`
函数	`convert_from(string bytea, src_encoding name)`
返回类型	`text`
描述	将字符串转换为数据库编码。原编码由 *src_encoding* 指定。*string* 必须是该编码下的有效字符串。

示例	convert_from('text_in_utf8', 'UTF8')
结果	以当前数据库编码表示的 text_in_utf8
函数	convert_to(*string* text, *dest_encoding* name)
返回类型	bytea
描述	将字符串转换为 *dest_encoding*。
示例	convert_to('some text', 'UTF8')
结果	以 UTF8 编码表示的 some text
函数	decode(*string* text, *format* text)
返回类型	bytea
描述	从 *string* 中的文本表示解码二进制数据。*format* 的选项与 encode 相同。
示例	decode('MTIzAAE=', 'base64')
结果	\x3132330001
函数	encode(*data* bytea, *format* text)
返回类型	text
描述	将二进制数据编码为文本表示。支持的格式为：base64、hex、escape。escape 将零字节和最高位为 1 的字节转换为八进制序列（*nnn*），并将反斜杠双写。
示例	encode('123\000\001', 'base64')
结果	MTIzAAE=
函数	format(*formatstr* text [, *formatarg* "any" [, ...]])
返回类型	text
描述	根据格式字符串格式化参数。此函数类似于 C 函数 sprintf。参见第 9.4.1 节。
示例	format('Hello %s, %1$s', 'World')
结果	Hello World, World
函数	initcap(*string*)
返回类型	text
描述	将每个单词的首字母转换为大写，其余字母转换为小写。单词是由非字母数字字符分隔的字母数字字符序列。
示例	initcap('hi THOMAS')
结果	Hi Thomas
函数	left(*str* text, *n* int)
返回类型	text
描述	返回字符串中的前 *n* 个字符。当 *n* 为负数时，返回除最后 |*n*| 个字符以外的所有字符。
示例	left('abcde', 2)
结果	ab

函数	length(*string*)
返回类型	int
描述	*string* 中的字符数
示例	length('jose')
结果	4
函数	length(*string* bytea, *encoding* name)
返回类型	int
描述	采用给定 *encoding* 的 *string* 中的字符数。*string* 必须是该编码下的有效字符串。
示例	length('jose', 'UTF8')
结果	4
函数	lpad(*string* text, *length* int [, *fill* text])
返回类型	text
描述	在 *string* 前面添加字符 *fill*（默认为空格），将其填充到长度 *length*。如果 *string* 已经长于 *length*，则从右侧截断。
示例	lpad('hi', 5, 'xy')
结果	xyxhi
函数	ltrim(*string* text [, *characters* text])
返回类型	text
描述	从 *string* 的开头移除仅由 *characters* 中字符（默认为空格）组成的最长字符串
示例	ltrim('zzzytest', 'xyz')
结果	test
函数	md5(*string*)
返回类型	text
描述	计算 *string* 的 MD5 hash，并以十六进制返回结果
示例	md5('abc')
结果	900150983cd24fb0 d6963f7d28e17f72
函数	parse_ident(*qualified_identifier* text [, *strictmode* boolean DEFAULT true])
返回类型	text[]
描述	将 *qualified_identifier* 拆分为标识符数组，移除各标识符的引号。默认情况下，最后一个标识符后的多余字符会被视为错误；但如果第二个参数为 false，则忽略这些多余字符。（此行为适合解析函数等对象的名称。）注意，此函数不会截断过长的标识符。如需截断，可以将结果强制转换为 name[]。
示例	parse_ident('"SomeSchema".someTable')
结果	{SomeSchema,sometable}
函数	pg_client_encoding()
返回类型	name

描述	当前客户端编码名称
示例	pg_client_encoding()
结果	SQL_ASCII
函数	quote_ident(*string* text)
返回类型	text
描述	为给定字符串添加适当的引号并返回，使其可用作 SQL 语句字符串中的标识符。仅在必要时添加引号（即字符串包含不能用于标识符的字符，或会被大小写折叠时）。内嵌引号会适当地双写。另见例 43.1。
示例	quote_ident('Foo bar')
结果	"Foo bar"
函数	quote_literal(*string* text)
返回类型	text
描述	为给定字符串添加适当的引号并返回，使其可用作 SQL 语句字符串中的字符串字面量。内嵌单引号和反斜杠会被适当地双写。注意，quote_literal 在输入为 null 时返回 null；如果参数可能为 null，quote_nullable 通常更合适。另见例 43.1。
示例	quote_literal(E'O\'Reilly')
结果	'O''Reilly'
函数	quote_literal(*value* anyelement)
返回类型	text
描述	将给定值强制转换为文本，然后作为字面量加引号。内嵌单引号和反斜杠会被适当地双写。
示例	quote_literal(42.5)
结果	'42.5'
函数	quote_nullable(*string* text)
返回类型	text
描述	为给定字符串添加适当的引号并返回，使其可用作 SQL 语句字符串中的字符串字面量；如果参数为 null，则返回 NULL。内嵌单引号和反斜杠会被适当地双写。另见例 43.1。
示例	quote_nullable(NULL)
结果	NULL
函数	quote_nullable(*value* anyelement)
返回类型	text
描述	将给定值强制转换为文本，然后作为字面量加引号；如果参数为 null，则返回 NULL。内嵌单引号和反斜杠会被适当地双写。
示例	quote_nullable(42.5)
结果	'42.5'
函数	regexp_match(*string* text, *pattern* text [, *flags* text])
返回类型	text[]

描述	返回 POSIX 正则表达式首次匹配 *string* 时捕获的子字符串。更多信息参见第 9.7.3 节。
示例	`regexp_match('foobarbequebaz', '(bar)(beque)')`
结果	`{bar,beque}`
函数	`regexp_matches(`*string* `text,` *pattern* `text [,` *flags* `text])`
返回类型	`setof text[]`
描述	返回 POSIX 正则表达式匹配 *string* 时捕获的子字符串。更多信息参见第 9.7.3 节。
示例	`regexp_matches('foobarbequebaz', 'ba.', 'g')`

结果	`{bar}`
	`{baz}`
	（2 行）

函数	`regexp_replace(`*string* `text,` *pattern* `text,` *replacement* `text [,` *flags* `text])`
返回类型	`text`
描述	替换匹配 POSIX 正则表达式的子字符串。更多信息参见第 9.7.3 节。
示例	`regexp_replace('Thomas', '.[mN]a.', 'M')`
结果	`ThM`
函数	`regexp_split_to_array(`*string* `text,` *pattern* `text [,` *flags* `text ])`
返回类型	`text[]`
描述	使用 POSIX 正则表达式作为分隔符拆分 *string*。更多信息参见第 9.7.3 节。
示例	`regexp_split_to_array('hello world', '\s+')`
结果	`{hello,world}`
函数	`regexp_split_to_table(`*string* `text,` *pattern* `text [,` *flags* `text])`
返回类型	`setof text`
描述	使用 POSIX 正则表达式作为分隔符拆分 *string*。更多信息参见第 9.7.3 节。
示例	`regexp_split_to_table('hello world', '\s+')`

结果	`hello`
	`world`
	（2 行）

函数	`repeat(`*string* `text,` *number* `int)`
返回类型	`text`
描述	将 *string* 重复指定的 *number* 次
示例	`repeat('Pg', 4)`
结果	`PgPgPgPg`

函数	replace(*string* text, *from* text, *to* text)
返回类型	text
描述	将 *string* 中每次出现的子字符串 *from* 替换为子字符串 *to*
示例	replace('abcdefabcdef', 'cd', 'XX')
结果	abXXefabXXef
函数	reverse(*str*)
返回类型	text
描述	返回反转后的字符串。
示例	reverse('abcde')
结果	edcba
函数	right(*str* text, *n* int)
返回类型	text
描述	返回字符串中的最后 *n* 个字符。当 *n* 为负数时，返回除前 |*n*| 个字符以外的所有字符。
示例	right('abcde', 2)
结果	de
函数	rpad(*string* text, *length* int [, *fill* text])
返回类型	text
描述	在 *string* 后面追加字符 *fill*（默认为空格），将其填充到长度 *length*。如果 *string* 已经长于 *length*，则截断它。
示例	rpad('hi', 5, 'xy')
结果	hixyx
函数	rtrim(*string* text [, *characters* text])
返回类型	text
描述	从 *string* 的结尾移除仅由 *characters* 中字符（默认为空格）组成的最长字符串
示例	rtrim('testxxzx', 'xyz')
结果	test
函数	split_part(*string* text, *delimiter* text, *field* int)
返回类型	text
描述	按 *delimiter* 拆分 *string*，并返回指定字段（从一开始计数）
示例	split_part('abc~@~def~@~ghi', '~@~', 2)
结果	def
函数	strpos(*string*, *substring*)
返回类型	int
描述	指定子字符串的位置（与 position(*substring* in *string*) 相同，但注意参数顺序相反）

示例	strpos('high', 'ig')
结果	2
函数	substr(*string*, *from* [, *count*])
返回类型	text
描述	提取子字符串（与 substring(*string* from *from* for *count*) 相同）
示例	substr('alphabet', 3, 2)
结果	ph
函数	starts_with(*string*, *prefix*)
返回类型	bool
描述	如果 *string* 以 *prefix* 开始就返回真。
示例	starts_with('alphabet', 'alph')
结果	t
函数	to_ascii(*string* text [, *encoding* text])
返回类型	text
描述	将 *string* 从其他编码转换为 ASCII（只支持从 LATIN1、LATIN2、LATIN9 和 WIN1250 编码转换）
示例	to_ascii('Karel')
结果	Karel
函数	to_hex(*number* int or bigint)
返回类型	text
描述	将 *number* 转换为等价的十六进制表示
示例	to_hex(2147483647)
结果	7fffffff
函数	translate(*string* text, *from* text, *to* text)
返回类型	text
描述	将 *string* 中与 *from* 集合中某字符匹配的每个字符替换为 *to* 集合中的对应字符。如果 *from* 比 *to* 长，则移除输入中出现的 *from* 中的多余字符。
示例	translate('12345', '143', 'ax')
结果	a2x5

concat、concat_ws 和 format 是可变参数函数，因此可以把要串接或格式化的值作为一个标记了VARIADIC 关键字的数组进行传递（见第 38.5.5 节）。数组的元素被当作函数的独立普通参数一样处理。如果可变参数数组为 NULL，concat 和 concat_ws 返回 NULL，但format 把 NULL 当作一个零元素数组。

另见第 9.20 节中的聚合函数 string_agg。

表 9.10. 内置转换

转换名称 [a]	源编码	目标编码
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf8	SQL_ASCII	UTF8
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8

转换名称 [a]	源编码	目标编码
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
windows_1258_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_ascii	UTF8	SQL_ASCII
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1

转换名称 [a]	源编码	目标编码
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_windows_1258	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8

转换名称 [a]	源编码	目标编码
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
utf8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

[a] 转换名称遵循标准命名规则：将源编码正式名称中的所有非字母数字字符替换为下划线，再接上 _to_，然后接上按相同方式处理的目标编码名称。因此，这些名称可能与惯用的编码名称不同。

9.4.1. format

函数 format 根据一个格式字符串产生格式化的输出，其形式类似于 C 函数sprintf。

```
format(formatstr text [, formatarg "any" [, ...] ])
```

formatstr 是指定结果格式的字符串。格式字符串中的文本会直接复制到结果中，但格式说明符所在的位置除外。格式说明符充当字符串中的占位符，定义如何格式化后续函数参数并将其插入结果。每个*formatarg* 参数都按照其数据类型通常的输出规则转换为文本，再根据格式说明符进行格式化并插入结果字符串。

格式说明符以% 字符开头，格式如下：

```
%[position][flags][width]type
```

其中各组成字段为：

position（可选）

一个形式为 *n*$的字符串，其中*n* 是要打印的参数的索引。索引 1 表示 *formatstr* 之后的第一个参数。如果 *position* 被省略，默认会使用序列中的下一个参数。

flags（可选）

用于控制格式说明符输出格式的附加选项。目前唯一支持的标志是减号（-），它使格式说明符的输出左对齐。只有同时指定了*width* 字段时，它才有效。

width（可选）

指定用于显示格式说明符输出的最小字符数。输出将被在左部或右部（取决于-标志）用空格填充以保证充满该宽度。太小的宽度设置不会导致输出被截断，但是会被简单地忽略。宽度可以使用下列形式之一指定：一个正整数；一个星号（*）表示使用下一个函数参数作为宽度；或者一个形式为*n*$的字符串表示使用第 *n* 个函数参数作为宽度。

如果宽度来自一个函数参数，会先使用该宽度参数，再使用作为格式说明符值的参数。如果宽度参数为负数，结果会在长度为abs(*width*) 的字段中左对齐（如同指定了-标志）。

type（必需）

格式转换的类型，用于产生格式说明符的输出。支持下面的类型：

- s 将参数值格式化为一个简单字符串。空值会被视为空字符串。
- I 将参数值视作 SQL 标识符，并在必要时用双引号包围它。如果参数为 null，则会报错（等效于quote_ident）。
- L 将参数值作为 SQL 字面量加引号。null 值显示为不带引号的字符串 NULL（等效于 quote_nullable）。

除了以上所述的格式说明符之外，要输出一个字面形式的% 字符，可以使用特殊序列%%。

下面是一些基本格式转换的示例：

```
SELECT format('Hello %s', 'World');
结果: Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
结果: Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');
结果: INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');
结果: INSERT INTO locations VALUES('C:\Program Files')
```

下面是使用 *width* 字段和-标志的示例：

```
SELECT format('|%10s|', 'foo');
结果: |       foo|

SELECT format('|%-10s|', 'foo');
结果: |foo       |

SELECT format('|%*s|', 10, 'foo');
结果: |       foo|

SELECT format('|%*s|', -10, 'foo');
结果: |foo       |

SELECT format('|%-*s|', 10, 'foo');
结果: |foo       |

SELECT format('|%-*s|', -10, 'foo');
结果: |foo       |
```

这些示例展示了如何使用 *position* 字段：

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
结果: Testing three, two, one

SELECT format('|%*2$s|', 'foo', 10, 'bar');
结果: |       bar|

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
结果: |       foo|
```

不同于标准 C 函数 `sprintf`，PostgreSQL 的 `format` 函数允许在同一个格式字符串中，混合使用带有或不带有*position* 字段的格式说明符。不带 *position* 字段的格式说明符，总是使用最后一个已使用参数之后的下一个参数。此外，`format` 函数不要求格式字符串使用全部函数参数。例如：

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
结果: Testing three, two, three
```

对于安全地构造动态 SQL 语句，`%I` 和 `%L` 格式说明符特别有用。参见例 43.1。

9.5. 二进制串函数和操作符

本节描述用于检查和操作 `bytea` 类型值的函数和操作符。

SQL 定义了一些使用关键字而不是逗号来分隔参数的字符串函数。详情请见表 9.11。PostgreSQL 也提供了这些函数使用常规函数调用语法的版本（参阅表 9.12）。

注意

本页显示的示例结果假定服务器参数 `bytea_output` 设为 `escape`（传统的 PostgreSQL 格式）。

表 9.11. SQL 二进制串函数和操作符

函数	*string* `||` *string*
返回类型	bytea
描述	字符串串接
示例	`'\\Post'::bytea || '\047gres\000'::bytea`
结果	`\\Post'gres\000`
函数	`octet_length(`*string*`)`
返回类型	int
描述	二进制串中的字节数
示例	`octet_length('jo\000se'::bytea)`
结果	5
函数	`overlay(`*string* `placing` *string* `from int [for int])`
返回类型	bytea
描述	替换子字符串
示例	`overlay('Th\000omas'::bytea placing '\002\003'::bytea from 2 for 3)`
结果	`T\\002\\003mas`
函数	`position(`*substring* `in` *string*`)`
返回类型	int
描述	指定子字符串的位置
示例	`position('\000om'::bytea in 'Th\000omas'::bytea)`
结果	3
函数	`substring(`*string* `[from int] [for int])`
返回类型	bytea
描述	提取子字符串
示例	`substring('Th\000omas'::bytea from 2 for 3)`
结果	`h\000o`
函数	`trim([both]` *bytes* `from` *string*`)`
返回类型	bytea
描述	从 *string* 的开头和结尾移除仅由 *bytes* 中字节组成的最长字符串

示例	`trim('\000\001'::bytea from '\000Tom\001'::bytea)`
结果	`Tom`

还有一些二进制串处理函数可以使用，在表 9.12 列出。其中有一些是在内部使用，用于实现表 9.11 列出的 SQL 标准串函数。

表 9.12. 其他二进制串函数

函数	`btrim(`*`string`* `bytea, `*`bytes`* `bytea)`
返回类型	`bytea`
描述	从 *`string`* 的开头和结尾移除仅由 *`bytes`* 中字节组成的最长字符串
示例	`btrim('\000trim\001'::bytea, '\000\001'::bytea)`
结果	`trim`
函数	`decode(`*`string`* `text, `*`format`* `text)`
返回类型	`bytea`
描述	从 *`string`* 中的文本表示解码二进制数据。*`format`* 的选项与 `encode` 相同。
示例	`decode('123\000456', 'escape')`
结果	`123\000456`
函数	`encode(`*`data`* `bytea, `*`format`* `text)`
返回类型	`text`
描述	将二进制数据编码为文本表示。支持的格式为：`base64`、`hex`、`escape`。`escape` 将零字节和最高位为 1 的字节转换为八进制序列（*`\nnn`*），并将反斜杠双写。
示例	`encode('123\000456'::bytea, 'escape')`
结果	`123\000456`
函数	`get_bit(`*`string`*`, `*`offset`*`)`
返回类型	`int`
描述	从字符串中提取位
示例	`get_bit('Th\000omas'::bytea, 45)`
结果	`1`
函数	`get_byte(`*`string`*`, `*`offset`*`)`
返回类型	`int`
描述	从字符串中提取字节
示例	`get_byte('Th\000omas'::bytea, 4)`
结果	`109`
函数	`length(`*`string`*`)`
返回类型	`int`
描述	二进制串的长度

示例	`length('jo\000se'::bytea)`
结果	`5`
函数	`md5(`*`string`*`)`
返回类型	`text`
描述	计算 *`string`* 的 MD5 hash，并以十六进制返回结果
示例	`md5('Th\000omas'::bytea)`
结果	`8ab2d3c9689aaf18b4958c334c82d8b1`
函数	`set_bit(`*`string`*`, `*`offset`*`, `*`newvalue`*`)`
返回类型	`bytea`
描述	设置字符串中的位
示例	`set_bit('Th\000omas'::bytea, 45, 0)`
结果	`Th\000omAs`
函数	`set_byte(`*`string`*`, `*`offset`*`, `*`newvalue`*`)`
返回类型	`bytea`
描述	设置字符串中的字节
示例	`set_byte('Th\000omas'::bytea, 4, 64)`
结果	`Th\000o@as`
函数	`sha224(bytea)`
返回类型	`bytea`
描述	SHA-224 hash
示例	`sha224('abc')`
结果	`\x23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7`
函数	`sha256(bytea)`
返回类型	`bytea`
描述	SHA-256 hash
示例	`sha256('abc')`
结果	`\xba7816bf8f01cfea414140de5dae2223` `b00361a396177a9cb410ff61f20015ad`
函数	`sha384(bytea)`
返回类型	`bytea`
描述	SHA-384 hash
示例	`sha384('abc')`
结果	`\xcb00753f45a35e8bb5a03d699ac65007` `272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7`

函数	sha512(bytea)
返回类型	bytea
描述	SHA-512 hash
示例	sha512('abc')
结果	\xddaf35a193617abacc417349ae204131 12e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd 454d4423643ce80e2a9ac94fa54ca49f

函数 get_byte 和 set_byte 把二进制串中的第一个字节编号为字节 0。函数 get_bit 和 set_bit 在每一个字节中从右边起计数位；例如位 0 是第一个字节的最低有效位，而位 15 是第二个字节的最高有效位。

注意，由于历史原因，md5 函数返回十六进制编码的 text 值，而 SHA-2 函数返回 bytea。可以使用 encode 和 decode 在两者之间转换，例如使用 encode(sha256('abc'), 'hex') 得到十六进制编码的文本表示。

参见第 9.20 节中的聚合函数 string_agg 以及第 35.4 节中的大对象函数。

9.6. 位串函数和操作符

本节描述用于检查和操作位串的函数和操作符，也就是 bit 和 bit varying 类型的值。除常用比较操作符外，还可使用表 9.13 中的操作符。&、| 和 # 的位串操作数长度必须相同。移位时会保留字符串的原始长度，如示例所示。

表 9.13. 位串操作符

操作符	描述	示例	结果
||	串接	B'10001' || B'011'	10001011
&	按位与	B'10001' & B'01101'	00001
|	按位或	B'10001' | B'01101'	11101
#	按位异或	B'10001' # B'01101'	11100
~	按位非	~ B'10001'	01110
<<	按位左移	B'10001' << 3	01000
>>	按位右移	B'10001' >> 2	00100

以下 SQL 标准函数既适用于位串，也适用于字符串：length、bit_length、octet_length、position、substring、overlay。

以下函数既适用于位串，也适用于二进制串：get_bit、set_bit。处理位串时，这些函数将字符串的第一位（最左侧的位）编号为位 0。

此外，还可以在整数值与bit 类型之间进行类型转换。例如：

```
44::bit(10)                    0000101100
44::bit(3)                     100
cast(-44 as bit(12))           111111010100
'1110'::bit(4)::integer        14
```

注意，仅转换为 “bit” 意味着转换为 bit(1)，因此只会得到该整数的最低有效位。

> **注意**
>
> 将整数转换为 bit(n) 会复制最右侧的 n 位。将整数转换为宽度大于该整数自身的位串时，会在左侧进行符号扩展。

9.7. 模式匹配

PostgreSQL 提供了三种独立的模式匹配方法：传统的 SQL LIKE 操作符、较新的SIMILAR TO 操作符（在 SQL:1999 中加入），以及POSIX 风格的正则表达式。除了用于判断 “这个字符串是否匹配这个模式？” 的基本操作符外，还提供了提取或替换匹配子字符串、在匹配位置分割字符串的函数。

> **提示**
>
> 如果你的模式匹配的要求超出了这些，请考虑用 Perl 或 Tcl 写一个用户定义的函数。

> **小心**
>
> 虽然大多数正则表达式搜索都能很快完成，但特意构造的正则表达式可能需要任意长的处理时间和任意多的内存。接受来自恶意来源的正则表达式搜索模式时应当谨慎。如果必须这样做，建议设置语句超时。
>
> 使用 SIMILAR TO 模式进行搜索具有同样的安全风险，因为SIMILAR TO 提供了许多与 POSIX 风格正则表达式相同的能力。
>
> LIKE 搜索比另外两种方法简单得多，因此，当模式可能来自恶意来源时，使用它更安全。

9.7.1. LIKE

string LIKE *pattern* [ESCAPE *escape-character*]
string NOT LIKE *pattern* [ESCAPE *escape-character*]

如果该 *string* 匹配了提供的 *pattern*，那么LIKE 表达式返回真（和预期的一样，如果LIKE 返回真，那么NOT LIKE 表达式返回假，反之亦然。一个等效的表达式是NOT (*string* LIKE *pattern*)）。

如果 *pattern* 不包含百分号或下划线，那么该模式只表示它本身的字符串；在这种情况下，LIKE 的行为就像等号操作符。*pattern* 中的下划线（_）代表（匹配）任意单个字符；百分号（%）匹配任意由零个或多个字符组成的序列。

一些示例：

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

LIKE 模式匹配总是覆盖整个字符串。因此，如果想要匹配字符串内任意位置上的一个序列，该模式就必须以百分号开头并以百分号结尾。

要匹配字面量下划线或百分号而不是把它们当作通配符，*pattern* 中相应的字符前面必须带有转义字符。默认的转义字符是反斜线，但也可以使用ESCAPE 子句选择其他转义字符。要匹配转义字符本身，请写两个转义字符。

> **注意**
>
> 如果你关掉了 standard_conforming_strings，你在字符串常量中写的任何反斜线都需要被双写。详见第 4.1.2.1 节。

也可以通过写 ESCAPE '' 来选择不使用转义字符。这会禁用转义机制，从而无法关闭模式中下划线和百分号的特殊含义。

可以用关键字 ILIKE 代替 LIKE，使匹配根据当前区域设置忽略大小写。这不属于 SQL 标准，而是 PostgreSQL 的扩展。

操作符~~等效于 LIKE，而~~* 对应 ILIKE。还有 !~~和!~~* 操作符分别代表 NOT LIKE 和 NOT ILIKE。所有这些操作符都是 PostgreSQL 特有的。你可能会在 EXPLAIN 输出和类似的地方看到这些操作符名称，因为解析器实际上将LIKE 等翻译成这些操作符。

在 PostgreSQL 语法中，LIKE、ILIKE、NOT LIKE 和 NOT ILIKE 通常被当作操作符；例如，它们可以用于 *expression operator* ANY (*subquery*) 构造，但其中不能包含 ESCAPE 子句。在某些不常见的情况下，可能需要改用底层操作符名称。

还有前缀操作符 ^@ 和相应的 starts_with 函数，可用于只需匹配字符串开头的情况。

9.7.2. SIMILAR TO 正则表达式

```
string SIMILAR TO pattern [ESCAPE escape-character]
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```

SIMILAR TO 操作符根据其模式是否匹配给定字符串返回真或假。它与 LIKE 类似，但按照 SQL 标准定义的正则表达式来解释模式。SQL 正则表达式是 LIKE 表示法和常见正则表达式表示法的一种奇特结合。

与 LIKE 类似，SIMILAR TO 操作符只有在其模式匹配整个字符串时才算成功；这一点不同于普通正则表达式，后者可以匹配字符串的任意部分。与LIKE 相同，SIMILAR TO 也使用_和% 作为通配符，分别匹配任意单个字符和任意字符串（分别类似于 POSIX 正则表达式中的.和.*）。

除了这些从 LIKE 借用的功能之外，SIMILAR TO 支持下面这些从 POSIX 正则表达式借用的模式匹配元字符：

- |表示选择（两个候选之一）。
- * 表示重复前面的项零次或更多次。
- +表示重复前面的项一次或更多次。
- ?表示重复前面的项零次或一次。
- {*m*}表示重复前面的项刚好 *m* 次。
- {*m*,}表示重复前面的项 *m* 次或更多次。
- {*m*,*n*}表示重复前面的项至少 *m* 次并且不超过 *n* 次。
- 可以使用圆括号() 把多个项组合成一个逻辑项。
- 一个方括号表达式[...] 声明一个字符类，就像 POSIX 正则表达式一样。

注意点号（.）不是 SIMILAR TO 的一个元字符。

与 LIKE 一样，反斜杠会禁用这些元字符的特殊含义；也可以用 ESCAPE 指定其他转义字符。

下面是一些示例：

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

带三个参数的 substring 函数，即 substring(*string* from *pattern* for *escape-character*)，用于提取匹配 SQL 正则表达式模式的子串。与 SIMILAR TO 一样，指定模式必须匹配整个数据字符串，否则函数失败并返回空值。要指出成功时应返回的模式部分，模式中必须出现两次后跟双引号（"）的转义字符。函数返回匹配这两个标记之间的模式部分的文本。

一些示例，使用#" 定界返回串：

```
substring('foobar' from '%#"o_b#"%' for '#')   oob
substring('foobar' from '#"o_b#"%' for '#')    NULL
```

9.7.3. POSIX 正则表达式

表 9.14 列出了所有可用于 POSIX 正则表达式模式匹配的操作符。

表 9.14. 正则表达式匹配操作符

操作符	描述	示例
~	匹配正则表达式，区分大小写	'thomas' ~ '.*thomas.*'
~*	匹配正则表达式，不区分大小写	'thomas' ~* '.*Thomas.*'
!~	不匹配正则表达式，区分大小写	'thomas' !~ '.*Thomas.*'

操作符	描述	示例
!~*	不匹配正则表达式，不区分大小写	'thomas' !~* '.*vadim.*'

POSIX 正则表达式提供了比 LIKE 和 SIMILAR TO 操作符更强大的模式匹配方式。许多 Unix 工具，例如egrep、sed 或 awk，都使用与这里描述的模式匹配语言相似的语言。

正则表达式是一个字符序列，是定义一组字符串（一个正则集）的简写。如果字符串属于正则表达式描述的正则集，就称该字符串匹配此正则表达式。与LIKE 一样，模式中的字符精确匹配字符串中的字符，除非该模式字符在正则表达式语言中有特殊含义 – 但正则表达式使用的特殊字符与LIKE 不同。与 LIKE 模式不同，正则表达式可以匹配字符串中的任意位置，除非显式将其锚定到字符串开头或末尾。

一些示例：

```
'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^(b|c)' false
```

POSIX 模式语言的详细描述见下文。

带两个参数的 substring 函数，即 substring(*string* from *pattern*)，用于提取匹配 POSIX 正则表达式模式的子串。如果没有匹配，它返回空值；否则返回匹配模式的那部分文本。但是，如果模式包含圆括号，则返回匹配第一个括号子表达式（左圆括号最先出现的那个）的文本。如果要在表达式内部使用圆括号而不触发这个例外，可以在整个表达式外再加一对圆括号。如果需要在想要提取的子表达式之前使用圆括号，请参见下文介绍的非捕获圆括号。

一些示例：

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

regexp_replace 函数用新文本替换匹配 POSIX 正则表达式模式的子字符串。它的语法为 regexp_replace(*source*, *pattern*, *replacement* [, *flags*])。如果没有与 *pattern* 匹配的内容，则原样返回 *source* 字符串。如果存在匹配，则返回将匹配子字符串替换为 *replacement* 字符串后的 *source* 字符串。*replacement* 字符串可以包含 *n*，其中 *n* 为 1 至 9，表示应插入与模式中第 *n* 个圆括号子表达式匹配的源子字符串；它也可以包含 \&，表示应插入与整个模式匹配的子字符串。如果需要在替换文本中放置字面的反斜杠，应写成 \\。*flags* 参数是可选的文本字符串，其中包含零个或多个改变函数行为的单字母标志。标志 i 指定不区分大小写的匹配，而标志 g 指定替换每个匹配的子字符串，而不仅是第一个。支持的标志（不包括 g）在表 9.22 中描述。

一些示例：

```
regexp_replace('foobarbaz', 'b..', 'X')
                                   fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                   fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
                                   fooXarYXazY
```

`regexp_match` 函数返回一个 text 数组，包含字符串首次匹配 POSIX 正则表达式模式时捕获的子字符串。它的语法为 `regexp_match(`*`string`*`,` *`pattern`* `[,` *`flags`* `])`。如果没有匹配，结果为`NULL`。如果找到匹配，并且*`pattern`* 不包含圆括号子表达式，则结果是包含与整个模式匹配的子字符串的单元素 text 数组。如果找到匹配，并且*`pattern`* 包含圆括号子表达式，则结果是一个 text 数组，其中第*`n`* 个元素是匹配 *`pattern`* 的第 *`n`* 个圆括号子表达式的子字符串（不包括“非捕获”括号；详情见下文）。*`flags`* 参数是一个可选的文本字符串，其中包含零个或多个单个字母标志，用于更改函数的行为。支持的标志在表 9.22 中描述。

下面是一些示例：

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
 regexp_match
--------------
 {barbeque}
(1 row)

SELECT regexp_match('foobarbequebaz', '(bar)(beque)');
 regexp_match
--------------
 {bar,beque}
(1 row)
```

通常，如果只想得到整个匹配的子字符串，或在没有匹配时得到`NULL`，可以写成如下形式：

```
SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
 regexp_match
--------------
 barbeque
(1 row)
```

`regexp_matches` 函数返回一个 text 数组的集合，其中包含字符串匹配 POSIX 正则表达式模式时捕获的子字符串。它具有与`regexp_match` 相同的语法。如果没有匹配，则此函数不返回任何行；如果有匹配且未给出`g` 标志，则返回一行；如果有*`N`* 个匹配且给出 `g` 标志，则返回*`N`* 行。每个返回的行都是一个 text 数组，其中包含整个匹配的子字符串或与*`pattern`* 的括号子表达式匹配的子字符串，就像`regexp_match` 中描述的那样。`regexp_matches` 接受在表 9.22 中显示的所有标志，还接受`g` 标志，该标志命令它返回所有匹配项，而不仅仅是第一个匹配项。

一些示例：

```
SELECT regexp_matches('foo', 'not there');
 regexp_matches
----------------
(0 rows)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

> **提示**
>
> 在大部分情况下，`regexp_matches()` 应该与 `g` 标志一起使用，因为如果只是想要第一个匹配，使用`regexp_match()` 会更加简单高效。不过，`regexp_match()` 仅存在于 PostgreSQL 版本 10 以及更高的版本中。当在较老的版本中使用时，一种常用的技巧是把 `regexp_matches()` 调用放在子查询中，例如：
>
> ```
> SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
> ```
>
> 如果有一个匹配，则这个语句会产生一个 text 数组，否则返回`NULL`，这和 `regexp_match()` 的做法一样。如果没有子查询，这个查询对于没有匹配的表行根本不会产生输出，这通常不是预期行为。

`regexp_split_to_table` 把一个 POSIX 正则表达式模式当作一个定界符来分割字符串。它的语法形式是 `regexp_split_to_table(`*`string`*`, `*`pattern`*` [, `*`flags`*` ])`。如果没有与 *`pattern`* 的匹配，该函数返回*`string`*。如果至少有一个匹配，对每一个匹配它都返回从上一个匹配的末尾（或者串的开头）到这次匹配开头之间的文本。当没有更多匹配时，它返回从上一次匹配的末尾到串末尾之间的文本。*`flags`* 参数是一个可选的文本串，它包含零个或更多单字母标志，这些标志可以改变该函数的行为。`regexp_split_to_table` 能支持的标志在表 9.22 中描述。

`regexp_split_to_array` 函数的行为和 `regexp_split_to_table` 相同，不过`regexp_split_to_array` 会把它的结果以一个 `text` 数组的形式返回。它的语法是 `regexp_split_to_array(`*`string`*`, `*`pattern`*` [, `*`flags`*` ])`。这些参数和 `regexp_split_to_table` 的相同。

下面是一些示例：

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog', '\s+') AS
 foo;
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s+');
              regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*') AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
 b
 r
 o
 w
 n
 f
 o
 x
(16 rows)
```

正如最后一个示例所示，正则表达式分割函数会忽略出现在字符串开头或结尾或紧跟在前一个匹配项之后的零长度匹配。这与`regexp_match`和`regexp_matches`实现的严格的正则表达式匹配定义相矛盾，但在实践中通常是最方便的行为。其他软件系统如 Perl 使用类似的定义。

9.7.3.1. 正则表达式细节

PostgreSQL 的正则表达式是使用 Henry Spencer 写的一个包来实现的。下面的正则表达式的大部分描述都是从他的手册页中逐字拷贝过来的。

正则表达式（RE），在POSIX 1003.2 中定义，它有两种形式：扩展的 RE 或者是 ERE（大概地说就是那些在`egrep`里的），基本的 RE 或者是 BRE（大概地说就是那些在`ed`里的）。PostgreSQL 支持两种形式，并且还实现了一些 POSIX 标准中没有但是在类似 Perl 或者 Tcl 这样的语言中得到广泛应用的一些扩展。使用了那些非 POSIX 扩展的RE 叫高级 RE，或者本文档里说的ARE。ARE 几乎完全是 ERE 的超集，但是 BRE 有几个符号上的不兼容（以及更多的限制）。我们首先描述 ARE 和 ERE 形式，描述那些只适用于 ARE 的特性，然后描述 BRE 的区别是什么。

> **注意**
>
> PostgreSQL 最初总是假定正则表达式遵循 ARE 规则。不过，也可以像第 9.7.3.4 节中所述那样，在 RE 模式前加上一个嵌入选项，从而改用限制更多的 ERE 或 BRE 规则。这对于需要精确遵循POSIX 1003.2 规则的应用有助于保持兼容性。

一个正则表达式被定义为一个或更多分支，它们之间被 | 分隔。只要能匹配其中一个分支的东西都能匹配正则表达式。

一个分支是零个或多个量化原子或约束，连接在一起。它匹配第一个的匹配项，然后是第二个的匹配项，依此类推；一个空分支匹配空字符串。

一个量化原子是一个原子，后面可以跟一个量词。没有量词时，匹配一次原子所匹配的内容；有量词时，按量词指定的次数匹配原子所匹配的内容。原子可以是表 9.15 列出的任何一种形式。可用量词及其含义见表 9.16。

一个约束匹配一个空串，但只是在满足特定条件下才匹配。约束可以在能够使用原子的地方使用，只是它不能跟着量词。简单的约束在表 9.17 里显示；更多的约束稍后描述。

表 9.15. 正则表达式原子

原子	描述
(*re*)	（其中*re* 是任意正则表达式）匹配*re* 所匹配的内容，并记录该匹配，以备输出结果
(?:*re*)	同上，但不记录匹配结果（“非捕获”圆括号；仅适用于 ARE）
.	匹配任意单个字符
[*chars*]	一个方括号表达式，匹配*chars* 中的任意一个（详见第 9.7.3.2 节）
k	（其中*k* 既不是字母也不是数字）把该字符视为普通字符并匹配它，例如，\\ 匹配反斜线字符
c	其中*c* 是字母或数字（后面可能还有其他字符），这是一个转义，参见第 9.7.3.3 节（仅适用于 ARE；在 ERE 和 BRE 中，它匹配*c*）
{	如果后面跟着非数字字符，则匹配左花括号{；如果后面跟着数字，则是*bound* 的开头（见下文）
x	其中*x* 是一个没有其它意义的单个字符，则匹配该字符

RE 不能以反斜线（\）结尾。

> **注意**
>
> 如果你关掉了 standard_conforming_strings，你在字符串常量中写的任何反斜线都需要被双写。详见第 4.1.2.1 节。

表 9.16. 正则表达式量词

量词	匹配
*	一个由原子的 0 次或更多次匹配组成的序列
+	一个由原子的 1 次或更多次匹配组成的序列
?	一个由原子的 0 次或 1 次匹配组成的序列
{*m*}	一个由原子的正好*m* 次匹配组成的序列
{*m*,}	一个由原子的*m* 次或更多次匹配组成的序列
{*m*,*n*}	一个由原子的从*m* 次到*n* 次（包括）匹配组成的序列；*m* 不能超过*n*
*?	* 的非贪婪版本
+?	+的非贪婪版本
??	?的非贪婪版本
{*m*}?	{*m*}的非贪婪版本
{*m*,}?	{*m*,}的非贪婪版本
{*m*,*n*}?	{*m*,*n*}的非贪婪版本

使用{ . . . }的形式被称作范围。一个范围内的数字 *m* 和 *n* 都是无符号十进制整数，允许的数值从 0 到 255（包含）。

非贪婪量词（仅适用于 ARE）与对应的普通（贪婪）量词匹配相同的可能内容，但优先选择最少的匹配次数，而不是最多的匹配次数。详见第 9.7.3.5 节。

> **注意**
>
> 一个量词不能紧跟在另外一个量词后面，例如** 是非法的。量词不能作为表达式或者子表达式的开头，也不能跟在^或者 | 后面。

表 9.17. 正则表达式约束

约束	描述
^	在字符串开头匹配
$	在字符串末尾匹配
(?=*re*)	正向先行断言在这样的位置匹配：存在从该位置开始且匹配*re* 的子字符串（仅适用于 ARE）
(?!*re*)	负向先行断言在这样的位置匹配：不存在任何从该位置开始且匹配*re* 的子字符串（仅适用于 ARE）
(?<=*re*)	正向后行断言在这样的位置匹配：存在以该位置结束且匹配*re* 的子字符串（仅适用于 ARE）
(?<!*re*)	负向后行断言在这样的位置匹配：不存在任何以该位置结束且匹配*re* 的子字符串（仅适用于 ARE）

先行和后行约束不能包含反向引用（参见第 9.7.3.3 节），其中的所有圆括号都视为非捕获圆括号。

9.7.3.2. 方括号表达式

方括号表达式是用[]括起来的字符列表。通常，它匹配列表中的任意单个字符（但请参见下文）。如果列表以^开头，则匹配任意不在列表剩余部分中的单个字符。如果列表中的两个字符用-分隔，则表示排序序列中这两个字符之间的完整字符范围（包含两个端点）；例如，ASCII 中的[0-9]匹配任意十进制数字。两个范围共享一个端点是不合法的，例如a-c-e。范围高度依赖排序序列，因此可移植程序应避免依赖它们。

想在列表中包含字面字符]，可以让它做列表的首字符（如果使用了^，需要放在其后）。想在列表中包含字面字符-，可以让它做列表的首字符或者尾字符，或者一个范围的第二个端点。想在列表中把字面字符-当做范围的起点，把它用[.和.]包围起来，这样它就成为一个排序元素（见下文）。除了这些字符本身、一些用[的组合（见下段）以及转义（只在 ARE 中有效）以外，所有其它特殊字符在方括号表达式里都失去它们的特殊含义。特别是，在 ERE 和 BRE 规则下\不是特殊的，但在 ARE 里，它是特殊的（引入一个转义）。

在一个方括号表达式里，一个排序元素（一个字符、一个被当做一个单一字符排序的多字符序列或者一个表示上面两种情况的排序序列名称）包含在[.和.]里面的时候表示该排序元素的字符序列。该序列被当做该方括号列表的一个单一元素。这允许一个包含多字符排序元素的方括号表达式去匹配多于一个字符，例如，如果排序序列包含一个ch排序元素，那么 RE [[.ch.]]*c 匹配 chchcc 的头五个字符。

> **注意**
>
> PostgreSQL 当前不支持多字符排序元素。这些信息描述了将来可能有的行为。

在方括号表达式里，包围在[=和=]里的排序元素是一个等价类，代表等效于那一个的所有排序元素的字符序列，包括它本身（如果没有其它等效排序元素，那么就好像封装定界符是[.和.]）。例如，如果o和^是一个等价类的成员，那么[[=o=]]、[[=^=]]和[o^]都是同义的。一个等价类不能是一个范围的端点。

在方括号表达式中，用[:和:]括起的字符类名表示属于该类的所有字符的列表。标准字符类名包括：alnum、alpha、blank、cntrl、digit、graph、lower、print、punct、space、upper、xdigit。它们表示 ctype 中定义的字符类。区域设置可以提供其他字符类。字符类不能用作范围的端点。

方括号表达式有两个特例：[[:<:]]和[[:>:]]是约束，分别匹配单词开头和结尾的空字符串。单词定义为一个前后都没有单词字符的单词字符序列。单词字符是alnum字符（由 ctype 定义）或下划线。这是与 POSIX 1003.2 兼容但未由其规定的扩展，在打算移植到其他系统的软件中应谨慎使用。通常更适合使用下文介绍的约束转义；它们并不更标准，但更容易键入。

9.7.3.3. 正则表达式转义

转义是以\开头，后面跟着一个字母或数字字符的特殊序列。转义有好几种变体：字符输入转义、字符类简写转义、约束转义以及反向引用。在 ARE 里，如果一个\后面跟着一个字母或数字，但是并未组成一个合法的转义，那么它是非法的。在 ERE 中没有转义：在方括号表达式之外，一个后面跟着字母或数字字符的\只是表示该字符是一个普通的字符，而且在一个方括号表达式里，\是一个普通的字符（后者是 ERE 与 ARE 之间唯一的实际不兼容之处）。

字符输入转义便于在 RE 中指定不可打印或其他不便输入的字符，见表 9.18。

字符类简写转义用来提供一些常用的字符类简写。它们显示在表 9.19 中。

约束转义是以转义形式书写的约束，在满足特定条件时匹配空字符串，见表 9.20。

反向引用（*n*）匹配数字 *n* 指定的被前面的圆括号子表达式匹配的同一个串（参阅表 9.21）。例如，([bc])\1 匹配 bb 或者 cc，但是不匹配bc 或者 cb。RE 中子表达式必须完全在反向引用前面。子表达式以它们的左圆括号的顺序编号。非捕获圆括号并不定义子表达式。

表 9.18. 正则表达式字符输入转义

转义	描述
\a	警告（响铃）字符，和 C 中一样
\b	退格，和 C 中一样
\B	反斜线（\）的同义词，用来减少双写反斜线
\c*X*	（其中*X* 是任意字符）低序 5 位和*X* 相同的字符，它的其他位都是零
\e	排序序列名称为ESC 的字符；若不存在这样的字符，则使用八进制值为033 的字符
\f	换页，和 C 中一样
\n	换行符，与 C 中相同
\r	回车，和 C 中一样
\t	水平制表符，和 C 中一样
\u*wxyz*	（其中*wxyz* 正好是四个十六进制位）十六进制值为0x*wxyz* 的字符
\U*stuvwxyz*	（其中*stuvwxyz* 正好是八个十六进制位）十六进制值为0x*stuvwxyz* 的字符
\v	垂直制表符，和 C 中一样
\x*hhh*	（其中*hhh* 是十六进制位的任意序列）十六进制值为0x*hhh* 的字符（一个单一字符，不管用了多少个十六进制位）
\0	值为0（空字节）的字符
xy	（其中*xy* 正好是两个八进制位，并且不是一个反向引用）八进制值为0*xy* 的字符
xyz	（其中*xyz* 正好是三个八进制位，并且不是一个反向引用）八进制值为0*xyz* 的字符

十六进制位是 0-9、a-f 和 A-F。八进制位是 0-7。

指定 ASCII 范围（0-127）之外的值的数字字符输入转义的含义取决于数据库编码。当编码是 UTF-8 时，转义值等价于 Unicode 代码点，例如 \u1234 表示字符 U+1234。对于其他多字节编码，字符输入转义通常只是指定该字符的字节值的串接。如果该转义值不对应数据库编码中的任何合法字符，将不会发生错误，但是它不会匹配任何数据。

字符输入转义总是被当作普通字符。例如，\135 是 ASCII 中的]，但\135 并不终止一个方括号表达式。

表 9.19. 正则表达式字符类简写转义

转义	描述
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_]（注意包括下划线）

转义	描述
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]_]（注意包括下划线）

在方括号表达式中，\d、\s 和 \w 会失去其外层方括号，而 \D、\S 和 \W 则是非法的。（因此，例如 [a-c\d] 等同于 [a-c[:digit:]]。此外，等同于 [a-c^[:digit:]] 的 [a-c\D] 是非法的。）

表 9.20. 正则表达式约束转义

转义	描述
\A	只在串开头匹配（与^的不同请参见第 9.7.3.5 节）
\m	只在一个词的开头匹配
\M	只在一个词的末尾匹配
\y	只在一个词的开头或末尾匹配
\Y	只在不属于单词开头或末尾的位置匹配
\Z	只在串的末尾匹配（与$的不同请参见第 9.7.3.5 节）

单词的定义与上文[[:<:]] 和[[:>:]] 的说明相同。方括号表达式中不允许使用约束转义。

表 9.21. 正则表达式反向引用

转义	描述
m	（其中*m* 是一个非零数字）一个到第*m* 个子表达式的反向引用
mnn	（其中*m* 是一个非零数字，并且*nn* 是后续的若干数字，并且十进制值*mnn* 不大于此前已出现的捕获右圆括号数）一个到第*mnn* 个子表达式的反向引用

> **注意**
>
> 八进制字符输入转义与反向引用之间存在固有歧义，按上文提到的启发式规则解决：前导零始终表示八进制转义。单个非零数字，如果后面没有其他数字，始终视为反向引用。不以零开头的多位数字序列，如果前面已有相应的子表达式（即该数字在反向引用的合法范围内），则视为反向引用，否则视为八进制转义。

9.7.3.4. 正则表达式元语法

除了上面描述的主要语法之外，还有几种特殊形式和杂项语法。

RE 可以以两种特殊的引导前缀之一开头。如果 RE 以***:开头，余下部分就被视为 ARE。（这在PostgreSQL 中通常没有影响，因为 RE 默认被视为 ARE；但如果通过正则表达式函数的*flags* 参数指定了 ERE 或 BRE 模式，它就会产生影响。）如果 RE 以***=开头，余下部分就作为按字面解释的字符串处理，所有字符都视为普通字符。

一个 ARE 可以以嵌入选项开头：一个序列(?*xyz*)（这里的 *xyz* 是一个或多个字母字符）声明影响剩余 RE 的选项。这些选项覆盖任何先前确定的选项 – 特别地，它们可以覆盖一个正则表达

式操作符隐含的大小写敏感的行为，或者覆盖正则表达式函数的*flags*参数所指定的选项。可用的选项字母在表 9.22 中显示。注意这些同样的选项字母也被用在正则表达式函数的 *flags* 参数中。

表 9.22. ARE 嵌入选项字母

选项	描述
b	RE 的剩余部分是一个 BRE
c	大小写敏感的匹配（覆盖操作符类型）
e	RE 的剩余部分是一个 ERE
i	大小写不敏感的匹配（见第 9.7.3.5 节）（覆盖操作符类型）
m	n 的历史原因的同义词
n	换行敏感的匹配（见第 9.7.3.5 节）
p	部分换行敏感的匹配（见第 9.7.3.5 节）
q	RE 的剩余部分按字面（“加引号”）解释，所有字符都视为普通字符
s	非换行敏感的匹配（默认）
t	紧凑语法（默认；见下文）
w	逆部分换行敏感（“怪异”）的匹配（见第 9.7.3.5 节）
x	扩展语法（见下文）

嵌入选项从结束该序列的)处开始生效。它们只能出现在 ARE 的开头（如果存在***:引导前缀，则位于该前缀之后）。

除了通常的紧凑 RE 语法（其中所有字符都有意义）之外，还有一种扩展语法，可以通过指定嵌入的x选项来使用。在扩展语法中，RE 中的空白字符会被忽略，同样被忽略的还有#与其后的换行符（或 RE 末尾）之间的所有字符。这使得复杂的 RE 可以分段并添加注释。此基本规则有三个例外：

- 空白字符或 # 的前面若有 \，该字符就会被保留。
- 方括号表达式里的空白或者#将被保留
- 在多字符符号里面不能出现空白和注释，例如(?:

在这里，空白字符包括空格、制表符、换行符，以及属于*space*字符类的任何字符。

最后，在 ARE 里，方括号表达式外面，序列(?#*ttt*)（其中 *ttt* 是任意不包含一个)的文本）是一个注释，它被完全忽略。同样，这样的东西是不允许出现在多字符符号的字符中间的，例如(?:。这种注释更像是一种历史产物而不是一种有用的设施，并且它们的使用已经被废弃；请使用扩展语法来替代。

如果指定了开头的***=引导前缀，那么这些元语法扩展都不能使用，因为这表示把用户输入作为按字面解释的字符串，而非 RE 处理。

9.7.3.5. 正则表达式匹配规则

在 RE 可以在给定串中匹配多于一个子串的情况下，RE 匹配串中最靠前的那个子串。如果 RE 可以匹配在那个位置开始的多个子串，要么是取最长的子串，要么是最短的，具体哪种，取决于 RE 是贪婪的还是非贪婪的。

一个 RE 是否贪婪取决于下面规则：

- 大多数原子以及所有约束，都没有贪婪属性（因为它们毕竟无法匹配长度不定的文本）。
- 在一个 RE 周围加上圆括号并不会改变其贪婪性。
- 带一个固定重复次数量词（`{m}`或者`{m}?`）的量化原子和原子自身具有同样的贪婪性（可能是没有）。
- 一个带其他普通的量词（包括`{m,n}`中 *m* 等于 *n* 的情况）的量化原子是贪婪的（首选最长匹配）。
- 一个带非贪婪量词（包括`{m,n}?`中 *m* 等于 *n* 的情况）的量化原子是非贪婪的（首选最短匹配）。
- 一个分支 – 也就是说，一个没有顶级 | 操作符的 RE – 和它里面的第一个有贪婪属性的量化原子有着同样的贪婪性。
- 一个由 | 操作符连接起来的两个或者更多分支组成的 RE 总是贪婪的。

上面的规则所描述的贪婪属性不仅仅适用于独立的量化原子，而且也适用于包含量化原子的分支和整个 RE。这里的意思是，匹配是按照分支或者整个 RE 作为一个整体匹配最长或者最短的可能子串。一旦整个匹配的长度确定，那么匹配任意特定子表达式的部分就基于该子表达式的贪婪属性进行判断，在 RE 里面靠前的子表达式的优先级高于靠后的子表达式。

一个相应的示例：

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
```
结果：`123`
```
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
```
结果：`1`

在第一个示例里，RE 作为整体是贪婪的，因为`Y*` 是贪婪的。它可以匹配从 `Y` 开始的东西，并且它匹配从这个位置开始的最长的串，也就是，`Y123`。输出是这里的圆括号包围的部分，或者说是`123`。在第二个示例里，RE 总体上是一个非贪婪的 RE，因为`Y*?`是非贪婪的。它可以匹配从 `Y` 开始的最短的子串，也就是说`Y1`。子表达式`[0-9]{1,3}`是贪婪的，但是它不能修改总体匹配长度的决定；因此它被迫只匹配`1`。

简而言之，如果一个 RE 同时包含贪婪和非贪婪的子表达式，那么总的匹配长度要么是尽可能长，要么是尽可能短，这取决于给整个 RE 赋予的属性。给子表达式赋予的属性只影响在这个匹配里，各个子表达式相对于其他子表达式能“吃掉”多少内容。

量词{1,1}和{1,1}?可以分别用于在一个子表达式或者整个 RE 上强制贪婪或者非贪婪。当需要整个 RE 具有不同于从其元素中推导出的贪婪属性时，这很有用。例如，假设我们尝试将一个包含一些数字的字符串分隔成数字以及在它们之前和之后的部分，我们可能会尝试这样做：

```
SELECT regexp_match('abc01234xyz', '(.*)(\d+)(.*)');
结果：{abc0123,4,xyz}
```

这不会有用：第一个.* 是贪婪的，因此它会“吃掉”尽可能多的字符而留下\d+去匹配在最后一个可能位置上的最后一个数字。我们可能会通过让它变成非贪婪来修复：

```
SELECT regexp_match('abc01234xyz', '(.*?)(\d+)(.*)');
结果：{abc,0,""}
```

这也不会有用：因为现在 RE 作为整体来说是非贪婪的，因此它会尽快结束全部的匹配。我们可以通过强制 RE 整体是贪婪的来得到我们想要的：

```
SELECT regexp_match('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}');
结果：{abc,01234,xyz}
```

将 RE 的整体贪婪性与各组件自身的贪婪性分开控制，为处理变长模式提供了很大的灵活性。

在决定更长或者更短的匹配时，匹配长度是以字符衡量的，而不是排序元素。一个空串会被认为比什么都不匹配长。例如：bb* 匹配 abbbc 的中间三个字符；(week|wee)(night|knights) 匹配 weeknights 的所有十个字符；而 (.*).* 匹配 abc 的时候，圆括号包围的子表达式匹配所有三个字符；当 (a*)* 被拿来匹配 bc 时，整个 RE 和圆括号子表达式都匹配一个空串。

如果指定不区分大小写的匹配，其效果近似于字母表中的所有大小写差别都消失了。当存在大小写形式的字母作为普通字符出现在方括号表达式之外时，实际上会转换为包含其大小写形式的方括号表达式，例如x 变成[xX]。当它出现在方括号表达式内部时，其所有大小写形式都会加入该表达式，例如[x] 变成[xX]，[^x] 变成[^xX]。

如果指定了换行敏感的匹配，.和使用^的方括号表达式将永远不会匹配换行字符（这样，匹配就不会跨越行，除非 RE 显式安排了跨行匹配）并且^和$除了分别匹配串开头和结尾之外，还将分别匹配换行后面和前面的空串。但是 ARE 转义\A 和\Z 仍然只匹配串的开头和结尾。

如果指定了部分换行敏感的匹配，那么它影响.和方括号表达式，这个时候和换行敏感的匹配一样，但是不影响^和$。

如果指定了逆部分换行敏感匹配，那么它影响^和$，其作用和在换行敏感的匹配里一样，但是不影响.和方括号表达式。这个并不是很有用，只是为了满足对称性而提供的。

9.7.3.6. 限制和兼容性

在这个实现里，对 RE 的长度没有特别的限制。但是，那些希望高移植性的程序应该避免使用长度超过 256 字节的 RE，因为 POSIX 兼容的实现可以拒绝接受这样的 RE。

ARE 与 POSIX ERE 实际不兼容的唯一特性是：\ 在方括号表达式中不会失去特殊含义。其他所有 ARE 特性所使用的语法，在 POSIX ERE 中都是非法的，或其效果未定义或未指定；引导前缀的 *** 语法同样不属于 POSIX 的 BRE 或 ERE 语法。

许多 ARE 扩展借鉴自 Perl，但其中一些经过了整理和修改，也未实现少数 Perl 扩展。需要注意的不兼容之处包括\b、\B、不对末尾换行符作特殊处理、取反的方括号表达式也受换行敏感匹配

影响、先行和后行约束中对圆括号及反向引用的限制，以及采用最长或最短匹配而非首次匹配的语义。

ARE 与旧版 ERE 语法存在两点重要的不兼容，这里比较的是 7.4 之前的PostgreSQL：

- 在 ARE 中，\ 后跟字母或数字字符时，要么表示转义，要么是错误；而在以前的版本中，它只是书写该字母或数字字符的另一种方式。这应该不是什么大问题，因为以前的版本没有理由使用这样的序列。
- 在 ARE 中，\ 在 [] 内仍是特殊字符，因此方括号表达式内的字面 \ 必须写成 \\。

9.7.3.7. 基本正则表达式

BRE 在几个方面和 ERE 不太一样。在 BRE 中，|、+和?都是普通字符并且没有与它们功能等价的东西。范围的定界符是\{和\}，而{和}本身是普通字符。嵌套的子表达式的圆括号是\(和\)，而(和)自身是普通字符。除非在 RE 开头或者是圆括号子表达式开头，^都是一个普通字符。除非在 RE 结尾或者是圆括号子表达式的结尾，$是一个普通字符。如果* 出现在 RE 开头或者是圆括号封装的子表达式开头（前面可能有^），那么它是个普通字符。最后，可以用单数字的反向引用，\<和\> 分别是[[:<:]] 和[[:>:]] 的同义词；在 BRE 中没有其它可用的转义。

9.8. 数据类型格式化函数

PostgreSQL 格式化函数提供一套强大的工具用于把各种数据类型（日期/时间、整数、浮点数、数值）转换成格式化的字符串以及反过来从格式化的字符串转换成指定的数据类型。表 9.23 列出了这些函数。这些函数都遵循一个公共的调用规范：第一个参数是待格式化的值，而第二个是一个定义输出或输入格式的模板。

表 9.23. 格式化函数

函数	返回类型	描述	示例
to_char(timestamp, text)	text	将时间戳转换为字符串	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	将时间间隔转换为字符串	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	将整数转换为字符串	to_char(125, '999')
to_char(double precision, text)	text	将 real/double precision 转换为字符串	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	将 numeric 转换为字符串	to_char(-125.8, '999D99S')
to_date(text, text)	date	将字符串转换为日期	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	将字符串转换为 numeric	to_number('12,454.8-', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	将字符串转换为时间戳	to_timestamp('05 Dec 2000', 'DD Mon YYYY')

> **注意**
>
> 还有一个接受单个参数的 to_timestamp 函数；参见表 9.30。

> **提示**
>
> to_timestamp 和 to_date 存在的目的是为了处理无法通过简单类型转换直接处理的输入格式。对于大部分标准的日期/时间格式，只需把源字符串类型转换为所需的数据类型即可，并且简单很多。类似地，对于标准的数字表示形式，to_number 也是没有必要的。

在一个 to_char 输出模板串中，一些特定的模式可以被识别并且被替换成基于给定值的被恰当地格式化的数据。任何不属于模板模式的文本都简单地照字面拷贝。同样，在一个输入模板串里（对其他函数），模板模式标识由输入数据串提供的值。如果在模板字符串中有不是模板模式的字符，输入数据字符串中的对应字符会被简单地跳过（不管它们是否等于模板字符串字符）。

表 9.24 展示了可以用于格式化日期和时间值的模板模式。

表 9.24. 用于日期/时间格式化的模板模式

模式	描述
HH	一天中的小时（01-12）
HH12	一天中的小时（01-12）
HH24	一天中的小时（00-23）
MI	分钟（00-59）
SS	秒（00-59）
MS	毫秒（000-999）
US	微秒（000000-999999）
SSSS	自午夜起的秒数（0-86399）
AM, am, PM 或 pm	上午/下午标记（不带句点）
A.M., a.m., P.M. 或 p.m.	上午/下午标记（带句点）
Y,YYY	带逗号的年（4 位或者更多位）
YYYY	年（4 位或者更多位）
YYY	年的最后 3 位数字
YY	年的最后 2 位数字
Y	年的最后 1 位数字
IYYY	ISO 8601 周编号方式的年（4 位或更多位）
IYY	ISO 8601 周编号方式的年的最后 3 位数字
IY	ISO 8601 周编号方式的年的最后 2 位数字
I	ISO 8601 周编号方式的年的最后 1 位数字
BC, bc, AD 或 ad	纪元指示器（不带句号）
B.C., b.c., A.D. 或 a.d.	纪元指示器（带句号）
MONTH	大写的月份全称（空格补齐到 9 字符）

模式	描述
Month	首字母大写的月份全称（空格补齐到 9 字符）
month	小写的月份全称（空格补齐到 9 字符）
MON	简写的大写形式的月名（英文 3 字符，本地化长度可变）
Mon	简写的首字母大写形式的月名（英文 3 字符，本地化长度可变）
mon	简写的小写形式的月名（英文 3 字符，本地化长度可变）
MM	月份编号（01-12）
DAY	大写的星期全称（空格补齐到 9 字符）
Day	首字母大写的星期全称（空格补齐到 9 字符）
day	小写的星期全称（空格补齐到 9 字符）
DY	大写的星期简称（英语 3 字符，本地化长度可变）
Dy	首字母大写的星期简称（英语 3 字符，本地化长度可变）
dy	小写的星期简称（英语 3 字符，本地化长度可变）
DDD	一年中的第几天（001-366）
IDDD	ISO 8601 周编号年中的第几天（001-371；一年的第 1 天是第一个 ISO 周的星期一）
DD	月内日序数（01-31）
D	星期几，周日 (1) 到周六 (7)
ID	ISO 8601 星期几，周一 (1) 到周日 (7)
W	月内周序数（1-5）（第一周从该月第一天开始）
WW	一年中的第几周（1-53）（第一周从该年第一天开始）
IW	ISO 8601 周编号年中的第几周（01-53；该年的第一个星期四位于第 1 周）
CC	世纪（2 位数）（21 世纪开始于 2001-01-01）
J	儒略日（从本地午夜的公元前 4714 年 11 月 24 日开始的整数日数；参见第 B.7 节）
Q	季度
RM	以大写罗马数字表示的月份（I-XII；I=一月）
rm	以小写罗马数字表示的月份（i-xii；i=一月）
TZ	大写形式的时区缩写（仅在to_char 中支持）
tz	小写形式的时区缩写（仅在to_char 中支持）
TZH	时区的小时
TZM	时区的分钟
OF	相对于 UTC 的时区偏移（仅在to_char 中支持）

修饰符可以被应用于模板模式来修改它们的行为。例如，FMMonth 就是带着 FM 修饰符的 Month 模式。表 9.25 展示了可用于日期/时间格式化的修饰符模式。

表 9.25. 用于日期/时间格式化的模板模式修饰符

修饰符	描述	示例
FM 前缀	填充模式（抑制前导零和填充的空格）	FMMonth
TH 后缀	大写形式的序数后缀	DDTH，例如，12TH
th 后缀	小写形式的序数后缀	DDth，例如，12th
FX 前缀	固定格式全局选项（见使用须知）	FX Month DD Day
TM 前缀	翻译模式（根据lc_time 输出本地化的星期名和月份名）	TMMonth
SP 后缀	拼写模式（未实现）	DDSP

日期/时间格式化的使用注意事项：

- FM 抑制了在模式输出中添加前导零和尾随空格的行为，这些前导零和尾随空格本来会被添加以使输出成为固定宽度。在PostgreSQL 中，FM 仅修改下一个格式说明，而在 Oracle 中FM 影响所有后续格式说明，并且重复的FM 修饰符切换填充模式的开启和关闭。

- TM 不包含尾随空格。to_timestamp 和 to_date 会忽略 TM 修饰符。

- to_timestamp 和 to_date 会跳过输入字符串中的多个空格，除非使用 FX 选项。例如，to_timestamp('2000   JUN', 'YYYY MON') 可以工作，但 to_timestamp('2000   JUN', 'FXYYYY MON') 会报错，因为 to_timestamp 只接受一个空格。FX 必须指定为模板中的第一项。

- 普通文本可以出现在 to_char 模板中，并会按字面输出。可以用双引号括起子串，使其即使包含模式关键字也强制按字面文本解释。例如，在'"Hello Year "YYYY' 中，YYYY 会被年份数据替换，但Year 中单独的 Y 不会被替换。在 to_date、to_number 和 to_timestamp 中，字面文本和双引号字符串会跳过与该字符串所含字符数相同数量的输入字符；例如，"XX" 跳过两个输入字符（无论它们是否为 XX）。

- 如果您想在输出中使用双引号，必须在其前面加上反斜杠，例如'\"YYYY Month\"'。反斜杠在双引号之外不起特殊作用。在双引号字符串内部，反斜杠会使下一个字符被按字面解释，无论是什么（但除非下一个字符是双引号或另一个反斜杠，否则没有特殊效果）。

- 在 to_timestamp 和 to_date 中，如果年份格式规范少于四位数字，例如YYY，并且提供的年份少于四位数字，年份将被调整为最接近 2020 年的年份，例如95 变为 1995 年。

- 在 to_timestamp 和 to_date 中，负年份被视为 BC 纪元。如果同时写入负年份和显式的BC 字段，则再次得到 AD。年份零被视为公元前 1 年。

- 在 to_timestamp 和 to_date 中，YYYY 转换在处理超过 4 位数字的年份时有限制。您必须在 YYYY 后使用一些非数字字符或模板，否则年份总是被解释为 4 位数字。例如（使用年份 20000）：to_date('200001131', 'YYYYMMDD') 将被解释为 4 位年份；而应该在年份后使用非数字分隔符，如 to_date('20000-1131', 'YYYY-MMDD') 或 to_date('20000Nov31', 'YYYYMonDD')。

- 在 to_timestamp 和 to_date 中，如果存在YYY、YYYY 或 Y,YYY 字段，则会接受但忽略 CC（世纪）字段。如果 CC 与 YY 或 Y 一起使用，则结果将计算为指定世纪中的那一年。如果指定了世纪但未指定年份，则假定为该世纪的第一年。

- 在 to_timestamp 和 to_date 中，星期几的名称或数字（DAY，D，以及相关字段类型）是被接受的，但在计算结果时会被忽略。同样适用于季度（Q）字段。

- 在 to_timestamp 和 to_date 中，ISO 8601 周编号日期（与公历日期不同）可以通过两种方式之一指定：

  - 年份、周编号和星期几：例如to_date('2006-42-4', 'IYYY-IW-ID') 返回日期 2006-10-19。如果省略星期几，则假定为 1（星期一）。

  - 年份和年内日序数：例如 to_date('2006-291', 'IYYY-IDDD') 也返回 2006-10-19。

  尝试使用 ISO 8601 周编号字段和公历日期字段的混合输入日期是荒谬的，并将导致错误。在 ISO 8601 周编号年的背景下，“月份”或“月内日序数”的概念没有意义。在公历年的背景下，ISO 周没有意义。

  > **小心**
  >
  > 虽然 to_date 会拒绝混合使用公历和 ISO 周编号日期字段，但to_char 不会，因为输出格式规范如YYYY-MM-DD (IYYY-IDDD) 可能很有用。但要避免编写类似 IYYY-MM-DD 的内容；那会在年初附近产生令人惊讶的结果。（有关更多信息，请参见第 9.9.1 节。）

- 在 to_timestamp 函数中，毫秒（MS）或微秒（US）字段被用作小数点后的秒数位。例如 to_timestamp('12.3', 'SS.MS') 不是 3 毫秒，而是 300，因为转换将其视为 12 + 0.3 秒。因此，对于格式SS.MS，输入值12.3、12.30 和 12.300 指定相同数量的毫秒。要获得三毫秒，必须写成12.003，转换将其视为 12 + 0.003 = 12.003 秒。

  这是一个更复杂的示例：to_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US') 为 15 小时 12 分钟，秒数为 2 秒 + 20 毫秒 + 1230 微秒 = 2.021230 秒。

- to_char(..., 'ID') 的星期几编号与 extract(isodow from ...) 函数匹配，但 to_char(..., 'D') 的星期几编号与 extract(dow from ...) 的不一致。

- to_char(interval) 格式化 HH 和 HH12，如在 12 小时制时钟上显示，例如零小时和 36 小时都输出为12，而HH24 输出完整的小时值，在interval 值中可以超过 23。

表 9.26 展示了可以用于格式化数值的模板模式。

表 9.26. 用于数值格式化的模板模式

模式	描述
9	数位（非有效位可以被省略）
0	数位（即便是非有效位也不会被省略）
.（句点）	小数点
,（逗号）	分组（千）分隔符
PR	尖括号内的负值
S	紧贴数值的正负号（使用区域设置）
L	货币符号（使用区域设置）
D	小数点（使用区域设置）
G	分组分隔符（使用区域设置）
MI	在指定位置的负号（如果数字 < 0）

模式	描述
PL	在指定位置的正号（如果数字 > 0）
SG	在指定位置的正/负号
RN	罗马数字（输入在 1 和 3999 之间）
TH 或 th	序数后缀
V	移动指定位数（参阅注解）
EEEE	科学记数的指数

数值格式化的使用注意事项：

- 0 指定一个数字位置，即使它包含前导/尾随零，也将始终打印出来。9 也指定一个数字位置，但如果它是一个前导零，则将被替换为一个空格，而如果它是一个尾随零并且指定了填充模式，则将被删除。（对于to_number()，这两个模式字符是等效的。）
- 如果格式指定的小数位数少于被格式化数值的小数位数，to_char() 会将该数值舍入到指定的小数位数。
- 模式字符 S、L、D 和 G 表示当前区域设置定义的正负号、货币符号、小数点和千位分隔符字符（参见lc_monetary 和 lc_numeric）。模式字符句点和逗号表示这些确切字符，具有小数点和千位分隔符的含义，不受区域设置影响。
- 如果 to_char() 的模式中没有明确指定正负号的位置，就会为正负号保留一列，并使其紧贴数值（紧靠数值左侧）。如果S 紧邻若干个 9 的左侧，它同样会紧贴数值。
- 使用 SG、PL 或 MI 格式化的正负号不紧贴数值；例如，to_char(-12, 'MI9999') 会产生'- 12'，但to_char(-12, 'S9999') 会产生' -12'。（Oracle 实现不允许在 9 之前使用 MI，而是要求9 在 MI 之前。）
- TH 不会转换小于零的值，也不会转换小数。
- PL，SG 和 TH 是 PostgreSQL 的扩展。
- 在 to_number 函数中，如果使用非数据模板模式，如L 或 TH，则会跳过相应数量的输入字符，无论它们是否与模板模式匹配，除非它们是数据字符（即数字、正负号、小数点或逗号）。例如，TH 会跳过两个非数据字符。
- V 与 to_char 一起，将输入值乘以10^*n*，其中*n* 是跟在 V 后面的数字位数。V 与 to_number 一起以类似的方式进行除法。to_char 和 to_number 不支持与小数点结合使用的 V（例如，不允许使用99.9V99）。
- EEEE（科学计数法）不能与任何其他格式模式或修饰符结合使用，除了数字和小数点模式之外，必须位于格式字符串的末尾（例如，9.99EEEE 是一个有效模式）。

某些修饰符可以被应用到任何模板来改变其行为。例如，FM99.99 是带有 FM 修饰符的 99.99 模式。表 9.27 中展示了用于数值格式化的模式修饰符。

表 9.27. 用于数值格式化的模板模式修饰符

修饰符	描述	示例
FM 前缀	填充模式（抑制尾随零和填充的空白）	FM99.99
TH 后缀	大写序数后缀	999TH

修饰符	描述	示例
th 后缀	小写序数后缀	999th

表 9.28 展示了一些使用 to_char 函数的示例。

表 9.28. to_char 示例

表达式	结果
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(-0.1, 'FM90.99')	'-0.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485'
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'

表达式	结果
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

9.9. 日期/时间函数和操作符

表 9.30 展示了可用于处理日期/时间值的函数，其细节在随后的小节中描述。表 9.29 演示了基本算术操作符（+、* 等）的行为。而与格式化相关的函数，可以参考第 9.8 节。你应当熟悉第 8.5 节中的日期/时间数据类型的背景知识。

此外，表 9.1 中显示的常用比较操作符也适用于日期/时间类型。日期和时间戳（带或不带时区）都是可比较的，而时间（带或不带时区）和间隔只能与相同数据类型的其他值进行比较。将不带时区的时间戳与带时区的时间戳进行比较时，前者的值假定是在TimeZone 配置参数指定的时区中给出的，并被转换到 UTC，以便与后者的值进行比较（后者在内部已用 UTC 表示）。类似地，日期值会被假定表示TimeZone 区域中的午夜，当它与时间戳进行比较时。

所有下文描述的接受 time 或 timestamp 输入的函数和操作符实际上都有两种变体：一种接收 time with time zone 或 timestamp with time zone，另外一种接受 time without time zone 或者 timestamp without time zone。为了简化，这些变种没有被独立地展示。此外，+和* 操作符都是可交换的操作符对（例如，date + integer 和 integer + date）；我们只显示每一对中的一个。

表 9.29. 日期/时间操作符

操作符	示例	结果
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3'（天）
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'

操作符	示例	结果
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

表 9.30. 日期/时间函数

函数	age(timestamp, timestamp)
返回类型	interval
描述	将两个参数相减，生成一个使用年和月，而不是只用日的“符号化”的结果
示例	age(timestamp '2001-04-10', timestamp '1957-06-13')
结果	43 years 9 mons 27 days
函数	age(timestamp)
返回类型	interval
描述	从 current_date（午夜）减去
示例	age(timestamp '1957-06-13')
结果	43 years 8 mons 3 days
函数	clock_timestamp()
返回类型	timestamp with time zone
描述	当前日期和时间（在语句执行期间变化）；参见第 9.9.4 节
示例	
结果	
函数	current_date
返回类型	date
描述	当前日期；参见第 9.9.4 节
示例	
结果	

函数	current_time
返回类型	time with time zone
描述	一天中的当前时刻；参见第 9.9.4 节
示例	
结果	
函数	current_timestamp
返回类型	timestamp with time zone
描述	当前日期和时间（当前事务开始时）；参见第 9.9.4 节
示例	
结果	
函数	date_part(text, timestamp)
返回类型	double precision
描述	获取子字段（等价于 extract）；参见第 9.9.1 节
示例	date_part('hour', timestamp '2001-02-16 20:38:40')
结果	20
函数	date_part(text, interval)
返回类型	double precision
描述	获取子字段（等价于 extract）；参见第 9.9.1 节
示例	date_part('month', interval '2 years 3 months')
结果	3
函数	date_trunc(text, timestamp)
返回类型	timestamp
描述	截断到指定精度；另见第 9.9.2 节
示例	date_trunc('hour', timestamp '2001-02-16 20:38:40')
结果	2001-02-16 20:00:00
函数	date_trunc(text, interval)
返回类型	interval
描述	截断到指定精度；另见第 9.9.2 节
示例	date_trunc('hour', interval '2 days 3 hours 40 minutes')
结果	2 days 03:00:00
函数	extract(*field* from timestamp)
返回类型	double precision
描述	获取子字段；参见第 9.9.1 节
示例	extract(hour from timestamp '2001-02-16 20:38:40')
结果	20

函数	`extract(`*field* `from interval)`
返回类型	`double precision`
描述	获取子字段；参见第 9.9.1 节
示例	`extract(month from interval '2 years 3 months')`
结果	`3`
函数	`isfinite(date)`
返回类型	`boolean`
描述	测试日期是否有限（不是正负无穷）
示例	`isfinite(date '2001-02-16')`
结果	`true`
函数	`isfinite(timestamp)`
返回类型	`boolean`
描述	测试时间戳是否有限（不是正负无穷）
示例	`isfinite(timestamp '2001-02-16 21:28:30')`
结果	`true`
函数	`isfinite(interval)`
返回类型	`boolean`
描述	测试时间间隔是否有限
示例	`isfinite(interval '4 hours')`
结果	`true`
函数	`justify_days(interval)`
返回类型	`interval`
描述	调整时间间隔，使 30 天的时段表示为月
示例	`justify_days(interval '35 days')`
结果	`1 mon 5 days`
函数	`justify_hours(interval)`
返回类型	`interval`
描述	调整时间间隔，使 24 小时的时段表示为天
示例	`justify_hours(interval '27 hours')`
结果	`1 day 03:00:00`
函数	`justify_interval(interval)`
返回类型	`interval`
描述	使用 `justify_days` 和 `justify_hours` 调整时间间隔，并额外调整符号
示例	`justify_interval(interval '1 mon -1 hour')`

结果	`29 days 23:00:00`
函数	`localtime`
返回类型	`time`
描述	一天中的当前时刻；参见第 9.9.4 节
示例	
结果	
函数	`localtimestamp`
返回类型	`timestamp`
描述	当前日期和时间（当前事务开始时）；参见第 9.9.4 节
示例	
结果	
函数	`make_date(`*`year`* `int,` *`month`* `int,` *`day`* `int)`
返回类型	`date`
描述	从年、月和日字段创建日期
示例	`make_date(2013, 7, 15)`
结果	`2013-07-15`
函数	`make_interval(`*`years`* `int DEFAULT 0,` *`months`* `int DEFAULT 0,` *`weeks`* `int DEFAULT 0,` *`days`* `int DEFAULT 0,` *`hours`* `int DEFAULT 0,` *`mins`* `int DEFAULT 0,` *`secs`* `double precision DEFAULT 0.0)`
返回类型	`interval`
描述	从年、月、周、日、小时、分钟和秒字段创建时间间隔
示例	`make_interval(days => 10)`
结果	`10 days`
函数	`make_time(`*`hour`* `int,` *`min`* `int,` *`sec`* `double precision)`
返回类型	`time`
描述	从小时、分钟和秒字段创建时间
示例	`make_time(8, 15, 23.5)`
结果	`08:15:23.5`
函数	`make_timestamp(`*`year`* `int,` *`month`* `int,` *`day`* `int,` *`hour`* `int,` *`min`* `int,` *`sec`* `double precision)`
返回类型	`timestamp`
描述	从年、月、日、小时、分钟和秒字段创建时间戳
示例	`make_timestamp(2013, 7, 15, 8, 15, 23.5)`
结果	`2013-07-15 08:15:23.5`

函数	make_timestamptz(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec* double precision, [*timezone* text])
返回类型	timestamp with time zone
描述	从年、月、日、小时、分钟和秒字段创建带时区的时间戳；如果未指定 *timezone*，则使用当前时区。
示例	make_timestamptz(2013, 7, 15, 8, 15, 23.5)
结果	2013-07-15 08:15:23.5+01
函数	now()
返回类型	timestamp with time zone
描述	当前日期和时间（当前事务开始时）；参见第 9.9.4 节
示例	
结果	
函数	statement_timestamp()
返回类型	timestamp with time zone
描述	当前日期和时间（当前语句开始时）；参见第 9.9.4 节
示例	
结果	
函数	timeofday()
返回类型	text
描述	当前的日期和时间（类似 clock_timestamp，但是采用 text 字符串）；参见第 9.9.4 节
示例	
结果	
函数	transaction_timestamp()
返回类型	timestamp with time zone
描述	当前日期和时间（当前事务开始时）；参见第 9.9.4 节
示例	
结果	
函数	to_timestamp(double precision)
返回类型	timestamp with time zone
描述	将 Unix 纪元时间（自 1970-01-01 00:00:00+00 起的秒数）转换为时间戳
示例	to_timestamp(1284352323)
结果	2010-09-13 04:32:03+00

除了这些函数以外，还支持 SQL 操作符OVERLAPS：

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

这个表达式在两个时间段（用它们的端点定义）重叠的时候得到真，当它们不重叠时得到假。端点可以用一对日期、时间或者时间戳来指定；或者是用一个后面跟着一个间隔的日期、时间或时间戳来指定。当一对值被提供时，起点或终点都可以被写在前面，OVERLAPS 会自动地把较早的值作为起点。每一个时间段被认为是表示半开区间 *start* <= *time* < *end*，除非*start* 和 *end* 相等，这种情况下它表示单个时刻。例如这表示两个只有一个共同端点的时间段不重叠。

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
结果: true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
结果: false
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
结果: false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
结果: true
```

当把一个 interval 值加到某个时间戳上（或从该时间戳中减去一个 interval 值）时，如果该时间戳为timestamp with time zone 类型，天数部分会相应增加或减少timestamp with time zone 的日期，变化天数为所指定的天数，而一天中的时刻保持不变。当跨越夏令时变化时（会话时区设为识别夏令时的时区），这意味着interval '1 day' 不一定等于interval '24 hours'。例如，当会话时区设置为America/Denver：

```
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '1 day';
结果: 2005-04-03 12:00:00-06
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '24 hours';
结果: 2005-04-03 13:00:00-06
```

出现这种情况，是因为夏令时变化跳过了一个小时；变化发生的时间为 2005-04-03 02:00:00，所在时区为America/Denver。

注意，age 返回的 months 字段可能存在歧义，因为不同月份的天数不同。PostgreSQL 在计算不足整月的部分时，会采用两个日期中较早的那个日期所在的月份。例如：age('2004-06-01', '2004-04-30') 使用 4 月得到 1 mon 1 day，而如果使用 5 月则会得到 1 mon 2 days，因为 5 月有 31 天，而 4 月只有 30 天。

日期和时间戳的减法也可能很复杂。一种概念上简单的方法是，先用EXTRACT(EPOCH FROM ...) 将各值转换为秒数，然后将结果相减；这样得到的是两个值之间的秒数。这种方法会针对每个月的天数、时区变化和夏令时变化进行调整。用 “-” 操作符将日期或时间戳值相减，会返回两个值之间的天数（每一天为 24 小时）和时/分/秒，也会作相同的调整。age 函数返回年、月、

日和时/分/秒，它会逐字段相减，然后调整负值字段。以下查询显示了这些方法的差异。示例结果在timezone = 'US/Eastern' 设置下产生；所用的两个日期之间发生了夏令时切换：

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');
结果: 10537200
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
        EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
        / 60 / 60 / 24;
结果: 121.958333333333
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01 12:00:00';
结果: 121 days 23:00:00
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz '2013-03-01 12:00:00');
结果: 4 mons
```

9.9.1. EXTRACT, date_part

```
EXTRACT(field FROM source)
```

该 extract 函数从日期/时间值中提取年份、小时等子字段。*source* 必须是以下类型的值表达式：timestamp，time，或interval。（date 类型的表达式会转换为 timestamp，因此也可以使用。）*field* 是一个标识符或字符串，用于选择从源值中提取的字段。extract 函数返回的值的类型为 double precision。以下是有效的字段名称：

century

世纪

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
结果: 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
结果: 21
```

第一个世纪始于公元 0001-01-01 00:00:00，尽管当时的人们并不知道。这一定义适用于所有采用格里高利历的国家。世纪编号没有 0，而是从 -1 世纪直接跳到 1 世纪。如果你对此有异议，请向梵蒂冈罗马圣彼得大教堂的教皇投诉。

day

对于 timestamp 值，表示月份中的日期字段（1 - 31）；对于 interval 值，表示天数

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
结果: 16
SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
结果: 40
```

decade

年份字段除以 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
结果: 200
```

dow

一周的日子从星期天（0）到星期六（6）

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：5

请注意 extract 函数的星期几编号与 to_char(..., 'D') 函数不同。

doy

一年中的第几天（1 - 365/366）

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：47

epoch

对于 timestamp with time zone 值，自 1970-01-01 00:00:00 UTC 以来的秒数（早于该时刻的时间戳对应负值）；对于date 和 timestamp 值，自 1970-01-01 00:00:00 以来的名义秒数，不考虑时区或夏令时规则；对于interval 值，间隔中的总秒数

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
```
结果：982384720.12
```
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40.12');
```
结果：982355920.12
```
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
```
结果：442800

您可以使用 to_timestamp 将一个 epoch 值转换回 timestamp with time zone：

```
SELECT to_timestamp(982384720.12);
```
结果：2001-02-17 04:38:40.12+00

注意，将to_timestamp 应用于从 date 或 timestamp 值中提取的 epoch 值可能会产生误导性的结果：结果实际上会假定原始值是以 UTC 时间给出的，这可能并非事实。

hour

小时字段（0 - 23）

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：20

isodow

一周的日子从星期一（1）到星期日（7）

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
```
结果：7

这与 dow 相同，除了星期天。这匹配ISO 8601 的星期几编号。

isoyear

日期所属的 ISO 8601 周编号年份（不适用于间隔）

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
```
结果：2005
```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
```
结果：2006

每个 ISO 8601 周编号年都从包含 1 月 4 日的那一周的星期一开始，因此在 1 月初或 12 月末，ISO 年可能与格里高利年不同。更多信息请参见 week 字段。

PostgreSQL 8.3 之前的版本不支持此字段。

julian

与日期或时间戳对应的儒略日（不适用于间隔）。非本地午夜的时间戳会产生带小数部分的值。更多信息请参见第 B.7 节。

```
SELECT EXTRACT(JULIAN FROM DATE '2006-01-01');
```
结果：2453737
```
SELECT EXTRACT(JULIAN FROM TIMESTAMP '2006-01-01 12:00');
```
结果：2453737.5

microseconds

秒字段，包括小数部分，乘以 1 000 000；注意这包括完整的秒数

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
```
结果：28500000

millennium

千年

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：3

1900 年代的年份属于第二个千年。第三个千年始于 2001 年 1 月 1 日。

milliseconds

秒字段，包括小数部分，乘以 1000。请注意，这包括完整的秒数。

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
```
结果：28500

minute

分钟字段（0 - 59）

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：38

`month`

对于 `timestamp` 值，表示一年中的月份编号（1 - 12）；对于 `interval` 值，表示月数对 12 取模（0 - 11）

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：2
```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
```
结果：3
```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
```
结果：1

`quarter`

日期所在的一年中的季度（1 - 4）

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：1

`second`

秒字段，包括小数部分（0 - 59[1]）

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：40
```
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
```
结果：28.5

`timezone`

与 UTC 的时区偏移量，以秒为单位。正值对应于 UTC 东部的时区，负值对应于 UTC 西部的时区。（从技术上讲，PostgreSQL 不使用 UTC，因为不处理闰秒。）

`timezone_hour`

时区偏移的小时部分

`timezone_minute`

时区偏移的分钟部分

`week`

一年中按 ISO 8601 周编号体系计算的周序号。根据定义，ISO 周从周一开始，一年的第一周包含该年的 1 月 4 日。换句话说，一年的第一个星期四在该年的第 1 周。

[1] 如果操作系统实现了闰秒，则为 60

在 ISO 周编号体系中，1 月初的日期可能属于前一年的第 52 周或第 53 周，而 12 月末的日期可能属于下一年的第一周。例如，2005-01-01 属于 2004 年的第 53 周，2006-01-01 属于 2005 年的第 52 周，而2012-12-31 属于 2013 年的第一周。建议将 isoyear 字段与 week 一起使用，以获得一致的结果。

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
```

结果：7

year

年份字段。请记住，没有0 AD，所以把BC 年份从 AD 年份中减去时需要小心。

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
```

结果：2001

注意

当输入值为 +/-Infinity 时，extract 对于单调递增的字段（epoch、julian、year、isoyear、decade、century 以及 millennium）返回 +/-Infinity。对于其他字段返回 NULL。PostgreSQL 9.6 之前的版本对所有输入无穷的情况都返回零。

extract 函数主要的用途是做计算性处理。对于用于显示的日期/时间值格式化，参阅第 9.8 节。

date_part 函数仿照传统的 Ingres 实现，后者对应SQL 标准的 extract 函数：

```
date_part('field', source)
```

注意，此处的*field* 参数必须是字符串值，而不能是名称。date_part 的有效字段名与 extract 相同。

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
```

结果：16

```
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
```

结果：4

9.9.2. date_trunc

date_trunc 函数在概念上和用于数字的 trunc 函数类似。

```
date_trunc('field', source)
```

source 是 timestamp 或 interval 类型的值表达式。（类型为 date 和 time 的值会自动转换为 timestamp 或 interval，分别对应这两种输入类型。）*field* 选择输入值的截断精度。返回值的类型是 timestamp 或 interval，所有小于所选精度的字段都设为零（日和月则设为一）。

field 的有效值是：

```
microseconds
milliseconds
second
minute
hour
day
week
month
quarter
year
decade
century
millennium
```

示例：

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
```
结果：`2001-02-16 20:00:00`

```
SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
```
结果：`2001-01-01 00:00:00`

9.9.3. AT TIME ZONE

`AT TIME ZONE` 可在不带时区和带时区的时间戳之间相互转换，也可将时间值转换到不同的时区。表 9.31 展示了它的各种变体。

表 9.31. AT TIME ZONE 变体

表达式	返回类型	描述
`timestamp without time zone AT TIME ZONE` *zone*	`timestamp with time zone`	将给定的不带时区时间戳视为指定时区中的时间
`timestamp with time zone AT TIME ZONE` *zone*	`timestamp without time zone`	将给定的带时区时间戳转换为新时区的时间，结果不带时区标识
`time with time zone AT TIME ZONE` *zone*	`time with time zone`	将给定的带时区时间转换为新时区的时间

在这些表达式里，所需的时区 *zone* 可以指定为文本值（例如 `'America/Los_Angeles'`），也可以指定为一个间隔值（例如 `INTERVAL '-08:00'`）。在文本情况下，时区名称可以按第 8.5.3 节中描述的任意方式指定。

示例（假设本地时区为 America/Los_Angeles）：

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
结果: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver';
结果: 2001-02-16 18:38:40

SELECT TIMESTAMP '2001-02-16 20:38:40-05' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE 'America/
Chicago';
结果: 2001-02-16 05:38:40
```

第一个示例为不带时区的值添加时区，并使用当前TimeZone 设置显示该值。第二个示例将带时区的时间戳值移到指定时区，并返回不带时区的值。这样就可以存储和显示与当前TimeZone 设置不同的值。第三个示例将东京时间转换为芝加哥时间。将 time 值转换到其他时区时，由于没有提供日期，会使用当前生效的时区规则。

函数 timezone(*zone*, *timestamp*) 等效于符合 SQL 标准的结构 *timestamp* AT TIME ZONE *zone*。

9.9.4. 当前日期/时间

PostgreSQL 提供了许多返回当前日期和时间的函数。这些 SQL 标准的函数全部都按照当前事务的开始时刻返回值：

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

CURRENT_TIME 和 CURRENT_TIMESTAMP 返回带时区的值；LOCALTIME 和 LOCALTIMESTAMP 返回不带时区的值。

CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME 和 LOCALTIMESTAMP 可以有选择地接受一个精度参数，该精度会使结果的秒字段舍入到指定的小数位数。如果没有精度参数，结果将给出可用的全部精度。

一些示例：

```
SELECT CURRENT_TIME;
结果： 14:39:53.662522-05
SELECT CURRENT_DATE;
结果： 2001-12-23
SELECT CURRENT_TIMESTAMP;
结果： 2001-12-23 14:39:53.662522-05
SELECT CURRENT_TIMESTAMP(2);
结果： 2001-12-23 14:39:53.66-05
SELECT LOCALTIMESTAMP;
结果： 2001-12-23 14:39:53.662522
```

因为这些函数全部都按照当前事务的开始时刻返回结果，所以它们的值在事务运行的整个期间内都不改变。我们认为这是一个特性：目的是为了允许一个事务在“当前”时间上有一致的概念，这样在同一个事务里的多个修改可以保持同样的时间戳。

> **注意**
>
> 其他数据库系统可能会更频繁地推进这些值。

PostgreSQL 还提供了返回当前语句的开始时间以及调用该函数时的实际当前时间的函数。这些非 SQL 标准的函数列表如下：

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

`transaction_timestamp()` 等价于 `CURRENT_TIMESTAMP`，但是其命名清楚地反映了它的返回值。`statement_timestamp()` 返回当前语句的开始时刻（更准确地说，是接收到客户端最近一条命令消息的时间）。`statement_timestamp()` 和 `transaction_timestamp()` 在一个事务的第一条命令期间返回值相同，但是在随后的命令中却不一定相同。`clock_timestamp()` 返回真正的当前时间，因此它的值甚至在同一条 SQL 命令中都会变化。`timeofday()` 是一个有历史原因的 PostgreSQL 函数。和 `clock_timestamp()` 相似，它也返回真实的当前时间，但是它的结果是一个格式化的`text` 串，而不是`timestamp with time zone` 值。`now()` 是 PostgreSQL 中与 `transaction_timestamp()` 等价的传统函数。

所有日期/时间数据类型也都接受特殊字面值 `now` 来指定当前日期和时间（同样解释为事务开始时间）。因此，下面三种写法都返回相同的结果：

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- 但请参阅下面的提示
```

> **提示**
>
> 当指定以后要计算的值时，不要使用第三种形式，例如在表列的`DEFAULT` 子句中。系统将在分析这个常量的时候把 `now` 转换为一个 `timestamp`，这样需要默认值时就会得到创建表的时间！而前两种形式要到实际使用默认值的时候才被计算，因为它们是函数调用。因此它们可以给出每次插入行的时刻。（参见第 8.5.1.4 节。）

9.9.5. 延时执行

以下函数可用于延迟服务器进程的执行：

```
pg_sleep(seconds)
pg_sleep_for(interval)
pg_sleep_until(timestamp with time zone)
```

pg_sleep 使当前会话的进程休眠 *seconds* 秒。*seconds* 的类型为 double precision，因此可以指定带小数部分的秒数作为延迟时间。pg_sleep_for 便于指定较长的休眠时间，其参数类型为interval。pg_sleep_until 是在需要指定唤醒时间时使用的便利函数。例如：

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

> **注意**
>
> 有效的休眠时间间隔精度与平台有关，0.01 秒是常见值。休眠延迟将至少持续指定的时长，也有可能由于服务器负荷而比指定的时间长。特别地，pg_sleep_until 并不保证能刚好在指定的时刻被唤醒，但它不会在比指定时刻早的时候醒来。

> **警告**
>
> 请确保在调用 pg_sleep 或者其变体时，你的会话没有持有不必要的锁。否则其它会话可能必须等待你的休眠会话，因而减慢整个系统速度。

9.10. 枚举支持函数

对于枚举类型（见第 8.7 节），有些函数可以避免硬编码枚举类型中的特定值，使程序更简洁。这些函数列在表 9.32 中。以下示例假定枚举类型按如下方式创建：

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

表 9.32. 枚举支持函数

函数	描述	示例	示例结果
enum_first(anyenum)	返回输入枚举类型的第一个值。	enum_first(null::rainbow)	red
enum_last(anyenum)	返回输入枚举类型的最后一个值。	enum_last(null::rainbow)	purple
enum_range(anyenum)	将输入枚举类型的所有值作为一个有序的数组返回。	enum_range(null::rainbow)	{red,orange,yellow,green,blue,purple}

函数	描述	示例	示例结果
enum_range(anyenum, anyenum)	以有序数组返回两个给定枚举值之间的范围。两个值必须来自同一枚举类型。如果第一个参数为 null，结果从该枚举类型的第一个值开始；如果第二个参数为 null，结果以该枚举类型的最后一个值结束。	enum_range('orange'::rainbow, 'green'::rainbow)	{orange,yellow,green}
		enum_range(NULL, 'green'::rainbow)	{red,orange,yellow,green}
		enum_range('orange'::rainbow, NULL)	{orange,yellow,green,blue,purple}

请注意，除了enum_range 的双参数形式外，这些函数都忽略传入的具体值，只关心其声明的数据类型。传入 null 或该类型的某个具体值，结果都相同。通常会将这些函数用于表列或函数参数，而不是像示例那样使用硬编码的类型名。

9.11. 几何函数和操作符

几何类型 point、box、lseg、line、path、polygon 和 circle 有大量内置支持函数和操作符，如表 9.33、表 9.34 和表 9.35 中所示。

> **小心**
>
> 注意，“相同”操作符 ~= 表示 point、box、polygon 和 circle 类型通常意义上的相等。其中一些类型也有 = 操作符，但 = 仅比较面积是否相等。这些类型的其他标量比较操作符（<= 等）同样比较面积。

表 9.33. 几何操作符

操作符	描述	示例
+	平移	box '((0,0),(1,1))' + point '(2.0,0)'
-	平移	box '((0,0),(1,1))' - point '(2.0,0)'
*	缩放/旋转	box '((0,0),(1,1))' * point '(2.0,0)'
/	缩放/旋转	box '((0,0),(2,2))' / point '(2.0,0)'
#	相交的点或矩形框	box '((1,-1),(-1,1))' # box '((1,1),(-2,-2))'
#	路径或多边形中的点数	# path '((1,0),(0,1),(-1,0))'
@-@	长度或周长	@-@ path '((0,0),(1,0))'
@@	中心	@@ circle '((0,0),10)'
##	第二个操作数上距离第一个操作数最近的点	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	两者之间的距离	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	是否重叠？（有一个公共点即为真。）	box '((0,0),(1,1))' && box '((0,0),(2,2))'

操作符	描述	示例
<<	是否严格位于左侧?	circle '((0,0),1)' << circle '((5,0),1)'
>>	是否严格位于右侧?	circle '((5,0),1)' >> circle '((0,0),1)'
&<	是否未超出对方的右边界?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	是否未超出对方的左边界?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<|	是否严格位于下方?	box '((0,0),(3,3))' <<| box '((3,4),(5,5))'
|>>	是否严格位于上方?	box '((3,4),(5,5))' |>> box '((0,0),(3,3))'
&<|	是否未超出对方的上边界?	box '((0,0),(1,1))' &<| box '((0,0),(2,2))'
|&>	是否未超出对方的下边界?	box '((0,0),(3,3))' |&> box '((0,0),(2,2))'
<^	是否位于下方（允许接触）?	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	是否位于上方（允许接触）?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	是否相交?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	是否水平?	?- lseg '((-1,0),(1,0))'
?-	是否水平对齐?	point '(1,0)' ?- point '(0,0)'
?|	是否竖直?	?| lseg '((-1,0),(1,0))'
?|	是否竖直对齐?	point '(0,1)' ?| point '(0,0)'
?-|	是否互相垂直?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
?||	是否平行?	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@>	是否包含?	circle '((0,0),2)' @> point '(1,1)'
<@	是否位于内部或边界上?	point '(1,1)' <@ circle '((0,0),2)'
~=	是否相同?	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

注意

在 PostgreSQL 8.2 之前，包含操作符 @> 和 <@ 分别称为 ~ 和 @。这些名称仍然可用，但已被弃用，最终将被移除。

表 9.34. 几何函数

函数	返回类型	描述	示例
area(*object*)	double precision	面积	area(box '((0,0),(1,1))')
center(*object*)	point	中心	center(box '((0,0),(1,2))')
diameter(circle)	double precision	圆的直径	diameter(circle '((0,0),2.0)')
height(box)	double precision	矩形框的竖直尺寸	height(box '((0,0),(1,1))')
isclosed(path)	boolean	是否为闭合路径?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	是否为开放路径?	isopen(path '[(0,0),(1,1),(2,0)]')
length(*object*)	double precision	长度	length(path '((-1,0),(1,0))')
npoints(path)	int	点数	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	int	点数	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	将路径转换为闭合路径	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	将路径转换为开放路径	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	圆的半径	radius(circle '((0,0),2.0)')
width(box)	double precision	矩形框的水平尺寸	width(box '((0,0),(1,1))')

表 9.35. 几何类型转换函数

函数	返回类型	描述	示例
box(circle)	box	将圆转换为矩形框	box(circle '((0,0),2.0)')
box(point)	box	将点转换为空矩形框	box(point '(0,0)')
box(point, point)	box	将点转换为矩形框	box(point '(0,0)', point '(1,1)')
box(polygon)	box	将多边形转换为矩形框	box(polygon '((0,0),(1,1),(2,0))')
bound_box(box, box)	box	将两个矩形框转换为边界框	bound_box(box '((0,0),(1,1))', box '((3,3),(4,4))')
circle(box)	circle	将矩形框转换为圆	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	由圆心和半径构造圆	circle(point '(0,0)', 2.0)

函数	返回类型	描述	示例
circle(polygon)	circle	将多边形转换为圆	circle(polygon '((0,0),(1,1),(2,0))')
line(point, point)	line	由点构造直线	line(point '(-1,0)', point '(1,0)')
lseg(box)	lseg	将矩形框的对角线转换为线段	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	由点构造线段	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	path	将多边形转换为路径	path(polygon '((0,0),(1,1),(2,0))')
point(double precision, double precision)	point	构造点	point(23.4, -44.5)
point(box)	point	矩形框的中心	point(box '((-1,0),(1,0))')
point(circle)	point	圆的中心	point(circle '((0,0),2.0)')
point(lseg)	point	线段的中心	point(lseg '((-1,0),(1,0))')
point(polygon)	point	多边形的中心	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	将矩形框转换为 4 个顶点的多边形	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	将圆转换为 12 个顶点的多边形	polygon(circle '((0,0),2.0)')
polygon(*npts*, circle)	polygon	将圆转换为 *npts* 个顶点的多边形	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	将路径转换为多边形	polygon(path '((0,0),(1,1),(2,0))')

可以把一个 point 当作下标为 0 和 1 的数组，访问它的两个数值分量。例如，如果t.p 是一个 point 列，那么SELECT p[0] FROM t 检索 X 坐标而 UPDATE t SET p[1] = ...改变 Y 坐标。同样，box 或者 lseg 类型的值可以当作两个 point 值组成的数组看待。

area 函数适用于 box、circle 和 path 类型。对于 path 数据类型，只有当 path 中的点构成的路径不自相交时，area 函数才能工作。例如，path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH 无法使用；但下面这个视觉上相同的 path '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH 可以使用。如果难以理解自相交与非自相交 path 的区别，可以把上述两条 path 并排画在方格纸上。

9.12. 网络地址函数和操作符

表 9.36 列出了可用于 cidr 和 inet 类型的操作符。操作符 <<、<<=、>>、>>= 和 && 测试子网包含关系。它们只考虑两个地址的网络部分（忽略主机部分），并判断一个网络是否与另一个网络相同，或是另一个网络的子网。

表 9.36. cidr 和 inet 操作符

操作符	描述	示例
<	小于	inet '192.168.1.5' < inet '192.168.1.6'
<=	小于或等于	inet '192.168.1.5' <= inet '192.168.1.5'
=	等于	inet '192.168.1.5' = inet '192.168.1.5'
>=	大于或等于	inet '192.168.1.5' >= inet '192.168.1.5'
>	大于	inet '192.168.1.5' > inet '192.168.1.4'
<>	不等于	inet '192.168.1.5' <> inet '192.168.1.4'
<<	被包含	inet '192.168.1.5' << inet '192.168.1/24'
<<=	被包含于或等于	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	包含	inet '192.168.1/24' >> inet '192.168.1.5'
>>=	包含或等于	inet '192.168.1/24' >>= inet '192.168.1/24'
&&	包含或被包含于	inet '192.168.1/24' && inet '192.168.1.80/28'
~	按位非	~ inet '192.168.1.6'
&	按位与	inet '192.168.1.6' & inet '0.0.0.255'
|	按位或	inet '192.168.1.6' | inet '0.0.0.255'
+	加法	inet '192.168.1.6' + 25
-	减法	inet '192.168.1.43' - 36
-	减法	inet '192.168.1.43' - inet '192.168.1.19'

表 9.37 列出了可用于 cidr 和 inet 类型的函数。abbrev、host 和 text 函数主要用于提供其他显示格式。

表 9.37. cidr 和 inet 函数

函数	abbrev(inet)
返回类型	text
描述	以文本表示的缩略显示格式
示例	abbrev(inet '10.1.0.0/16')
结果	10.1.0.0/16

函数	abbrev(cidr)
返回类型	text
描述	以文本表示的缩略显示格式
示例	abbrev(cidr '10.1.0.0/16')
结果	10.1/16
函数	broadcast(inet)
返回类型	inet
描述	网络的广播地址
示例	broadcast('192.168.1.5/24')
结果	192.168.1.255/24
函数	family(inet)
返回类型	int
描述	提取地址族；IPv4 为 4，IPv6 为 6
示例	family('::1')
结果	6
函数	host(inet)
返回类型	text
描述	以文本形式提取 IP 地址
示例	host('192.168.1.5/24')
结果	192.168.1.5
函数	hostmask(inet)
返回类型	inet
描述	构造网络的主机掩码
示例	hostmask('192.168.23.20/30')
结果	0.0.0.3
函数	masklen(inet)
返回类型	int
描述	提取网络掩码长度
示例	masklen('192.168.1.5/24')
结果	24
函数	netmask(inet)
返回类型	inet
描述	构造网络的网络掩码
示例	netmask('192.168.1.5/24')
结果	255.255.255.0

函数	`network(inet)`
返回类型	`cidr`
描述	提取地址的网络部分
示例	`network('192.168.1.5/24')`
结果	`192.168.1.0/24`
函数	`set_masklen(inet, int)`
返回类型	`inet`
描述	设置 `inet` 值的网络掩码长度
示例	`set_masklen('192.168.1.5/24', 16)`
结果	`192.168.1.5/16`
函数	`set_masklen(cidr, int)`
返回类型	`cidr`
描述	设置 `cidr` 值的网络掩码长度
示例	`set_masklen('192.168.1.0/24'::cidr, 16)`
结果	`192.168.0.0/16`
函数	`text(inet)`
返回类型	`text`
描述	以文本形式提取 IP 地址和网络掩码长度
示例	`text(inet '192.168.1.5')`
结果	`192.168.1.5/32`
函数	`inet_same_family(inet, inet)`
返回类型	`boolean`
描述	这些地址是否属于同一地址族?
示例	`inet_same_family('192.168.1.5/24', '::1')`
结果	`false`
函数	`inet_merge(inet, inet)`
返回类型	`cidr`
描述	包含两个给定网络的最小网络
示例	`inet_merge('192.168.1.5/24', '192.168.2.5/24')`
结果	`192.168.0.0/22`

任何 `cidr` 值都可以隐式或显式地转换为 `inet`；因此，上述处理 `inet` 的函数也适用于 `cidr` 值。（如果 `inet` 和 `cidr` 各有单独的函数，是因为这两种情况下的行为应当不同。）也允许将 `inet` 值转换为 `cidr`。这样做时，网络掩码右侧的所有位都会被直接置零，以创建有效的 `cidr` 值。此外，还可以用普通的类型转换语法将文本值转换为 `inet` 或 `cidr`：例如 `inet(`*`expression`*`)` 或 *`colname`*`::cidr`。

表 9.38 列出了可用于 macaddr 类型的函数。函数 trunc(macaddr) 返回将最后 3 个字节置零的 MAC 地址。可以用余下的前缀确定制造商。

表 9.38. macaddr 函数

函数	trunc(macaddr)
返回类型	macaddr
描述	将最后 3 个字节设为零
示例	trunc(macaddr '12:34:56:78:90:ab')
结果	12:34:56:00:00:00

macaddr 类型还支持按字典顺序比较的标准关系操作符（>、<= 等），以及用于按位非、与、或的位运算操作符（~、& 和 |）。

表 9.39 列出了可用于 macaddr8 类型的函数。函数 trunc(macaddr8) 返回将最后 5 个字节置零的 MAC 地址。可以用余下的前缀确定制造商。

表 9.39. macaddr8 函数

函数	trunc(macaddr8)
返回类型	macaddr8
描述	将最后 5 个字节设为零
示例	trunc(macaddr8 '12:34:56:78:90:ab:cd:ef')
结果	12:34:56:00:00:00:00:00
函数	macaddr8_set7bit(macaddr8)
返回类型	macaddr8
描述	将第 7 位设为 1，即修订的 EUI-64 格式，以便用于 IPv6 地址
示例	macaddr8_set7bit(macaddr8 '00:34:56:ab:cd:ef')
结果	02:34:56:ff:fe:ab:cd:ef

macaddr8 类型还支持用于排序的标准关系操作符（>、<= 等），以及用于按位非、与、或的位运算操作符（~、& 和 |）。

9.13. 文本检索函数和操作符

表 9.40、表 9.41 以及表 9.42 总结了为全文检索提供的函数和操作符。PostgreSQL 的文本检索功能的详细解释可参考第 12 章。

表 9.40. 文本检索操作符

操作符	@@
返回类型	boolean
描述	tsvector 是否匹配 tsquery?
示例	to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')

结果	t
操作符	@@@
返回类型	boolean
描述	@@ 的已弃用同义写法
示例	to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')
结果	t
操作符	||
返回类型	tsvector
描述	串接 tsvector
示例	'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector
结果	'a':1 'b':2,5 'c':3 'd':4
操作符	&&
返回类型	tsquery
描述	将 tsquery 进行 AND 组合
示例	'fat | rat'::tsquery && 'cat'::tsquery
结果	('fat' | 'rat') & 'cat'
操作符	||
返回类型	tsquery
描述	将 tsquery 进行 OR 组合
示例	'fat | rat'::tsquery || 'cat'::tsquery
结果	('fat' | 'rat') | 'cat'
操作符	!!
返回类型	tsquery
描述	对 tsquery 取反
示例	!! 'cat'::tsquery
结果	!'cat'
操作符	<->
返回类型	tsquery
描述	tsquery 后跟 tsquery
示例	to_tsquery('fat') <-> to_tsquery('rat')
结果	'fat' <-> 'rat'
操作符	@>
返回类型	boolean
描述	tsquery 是否包含另一个查询？

示例	`'cat'::tsquery @> 'cat & rat'::tsquery`
结果	`f`
操作符	`<@`
返回类型	`boolean`
描述	`tsquery` 是否被包含于另一个查询？
示例	`'cat'::tsquery <@ 'cat & rat'::tsquery`
结果	`t`

> **注意**
>
> `tsquery` 包含操作符只考虑两个查询中列出的词位，忽略组合操作符。

除了表中列出的操作符，`tsvector` 和 `tsquery` 类型还定义了普通的 B-树比较操作符（=、< 等）。这些操作符对文本检索用处不大，但可以用于其他用途，例如在这些类型的列上建立唯一索引。

表 9.41. 文本检索函数

函数	`array_to_tsvector(text[])`
返回类型	`tsvector`
描述	将词位数组转换为 `tsvector`
示例	`array_to_tsvector('{fat,cat,rat}'::text[])`
结果	`'cat' 'fat' 'rat'`
函数	`get_current_ts_config()`
返回类型	`regconfig`
描述	获取默认的文本检索配置
示例	`get_current_ts_config()`
结果	`english`
函数	`length(tsvector)`
返回类型	`integer`
描述	`tsvector` 中的词位数
示例	`length('fat:2,4 cat:3 rat:5A'::tsvector)`
结果	`3`
函数	`numnode(tsquery)`
返回类型	`integer`
描述	`tsquery` 中的词位数与操作符数之和
示例	`numnode('(fat & rat) | cat'::tsquery)`

结果	5
函数	plainto_tsquery([*config* regconfig ,] *query* text)
返回类型	tsquery
描述	生成 tsquery，忽略标点符号
示例	plainto_tsquery('english', 'The Fat Rats')
结果	'fat' & 'rat'
函数	phraseto_tsquery([*config* regconfig ,] *query* text)
返回类型	tsquery
描述	生成搜索短语的 tsquery，忽略标点符号
示例	phraseto_tsquery('english', 'The Fat Rats')
结果	'fat' <-> 'rat'
函数	websearch_to_tsquery([*config* regconfig ,] *query* text)
返回类型	tsquery
描述	从网络搜索风格的查询生成 tsquery
示例	websearch_to_tsquery('english', '"fat rat" or rat')
结果	'fat' <-> 'rat' | 'rat'
函数	querytree(*query* tsquery)
返回类型	text
描述	获取 tsquery 中可索引的部分
示例	querytree('foo & ! bar'::tsquery)
结果	'foo'
函数	setweight(*vector* tsvector, *weight* "char")
返回类型	tsvector
描述	为 *vector* 中的每个元素赋予 *weight*
示例	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')
结果	'cat':3A 'fat':2A,4A 'rat':5A
函数	setweight(*vector* tsvector, *weight* "char", *lexemes* text[])
返回类型	tsvector
描述	为 *vector* 中列在 *lexemes* 内的元素赋予 *weight*
示例	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A', '{cat,rat}')
结果	'cat':3A 'fat':2,4 'rat':5A
函数	strip(tsvector)
返回类型	tsvector
描述	从 tsvector 中移除位置和权重

示例	strip('fat:2,4 cat:3 rat:5A'::tsvector)
结果	'cat' 'fat' 'rat'
函数	to_tsquery([*config* regconfig ,] *query* text)
返回类型	tsquery
描述	正规化单词并转换为 tsquery
示例	to_tsquery('english', 'The & Fat & Rats')
结果	'fat' & 'rat'
函数	to_tsvector([*config* regconfig ,] *document* text)
返回类型	tsvector
描述	将文档文本转换为 tsvector
示例	to_tsvector('english', 'The Fat Rats')
结果	'fat':2 'rat':3
函数	to_tsvector([*config* regconfig ,] *document* json(b))
返回类型	tsvector
描述	将文档中的每个字符串值转换为 tsvector，再按文档顺序串接，生成单个 tsvector
示例	to_tsvector('english', '{"a": "The Fat Rats"}'::json)
结果	'fat':2 'rat':3
函数	json(b)_to_tsvector([*config* regconfig,] *document* json(b), *filter* json(b))
返回类型	tsvector
描述	将文档中由 *filter* 指定的每个值转换为 tsvector，再按文档顺序串接，生成单个 tsvector。*filter* 是一个 jsonb 数组，列出需要包含在结果 tsvector 中的元素种类。*filter* 的可用值为 "string"（包含所有字符串值）、"numeric"（以字符串形式包含所有数值）、"boolean"（以字符串 "true"/"false" 的形式包含所有布尔值）、"key"（包含所有键）或 "all"（包含上述全部内容）。可以组合这些值，例如包含所有字符串值和数值。
示例	json_to_tsvector('english', '{"a": "The Fat Rats", "b": 123}'::json, '["string", "numeric"]')
结果	'123':5 'fat':2 'rat':3
函数	ts_delete(*vector* tsvector, *lexeme* text)
返回类型	tsvector
描述	从 *vector* 中移除给定的 *lexeme*
示例	ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat')
结果	'cat':3 'rat':5A
函数	ts_delete(*vector* tsvector, *lexemes* text[])
返回类型	tsvector
描述	从 *vector* 中移除 *lexemes* 所列词位的所有出现

示例	ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat'])
结果	'cat':3
函数	ts_filter(*vector* tsvector, *weights* "char"[])
返回类型	tsvector
描述	仅从 *vector* 中选出具有给定 *weights* 的元素
示例	ts_filter('fat:2,4 cat:3b rat:5A'::tsvector, '{a,b}')
结果	'cat':3B 'rat':5A
函数	ts_headline([*config* regconfig,] *document* text, *query* tsquery [, *options* text])
返回类型	text
描述	显示查询匹配内容
示例	ts_headline('x y z', 'z'::tsquery)
结果	x y z
函数	ts_headline([*config* regconfig,] *document* json(b), *query* tsquery [, *options* text])
返回类型	text
描述	显示查询匹配内容
示例	ts_headline('{"a":"x y z"}'::json, 'z'::tsquery)
结果	{"a":"x y z"}
函数	ts_rank([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])
返回类型	float4
描述	计算文档相对于查询的排名分值
示例	ts_rank(textsearch, query)
结果	0.818
函数	ts_rank_cd([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])
返回类型	float4
描述	使用覆盖密度计算文档相对于查询的排名分值
示例	ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)
结果	2.01317
函数	ts_rewrite(*query* tsquery, *target* tsquery, *substitute* tsquery)
返回类型	tsquery
描述	将查询中的 *target* 替换为 *substitute*
示例	ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)
结果	'b' & ('foo' | 'bar')

函数	ts_rewrite(*query* tsquery, *select* text)
返回类型	tsquery
描述	使用 SELECT 命令提供的目标和替代内容进行替换
示例	SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')
结果	'b' & ('foo' | 'bar')
函数	tsquery_phrase(*query1* tsquery, *query2* tsquery)
返回类型	tsquery
描述	构造搜索 *query1* 后跟 *query2* 的查询（与 <-> 操作符相同）
示例	tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'))
结果	'fat' <-> 'cat'
函数	tsquery_phrase(*query1* tsquery, *query2* tsquery, *distance* integer)
返回类型	tsquery
描述	构造搜索 *query1* 后跟 *query2*、间距为 *distance* 的查询
示例	tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10)
结果	'fat' <10> 'cat'
函数	tsvector_to_array(tsvector)
返回类型	text[]
描述	将 tsvector 转换为词位数组
示例	tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector)
结果	{cat,fat,rat}
函数	tsvector_update_trigger()
返回类型	trigger
描述	自动更新 tsvector 列的触发器函数
示例	CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)
结果	
函数	tsvector_update_trigger_column()
返回类型	trigger
描述	自动更新 tsvector 列的触发器函数
示例	CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, configcol, title, body)
结果	
函数	unnest(tsvector, OUT *lexeme* text, OUT *positions* smallint[], OUT *weights* text)
返回类型	setof record
描述	将 tsvector 展开为一组行

示例	`unnest('fat:2,4 cat:3 rat:5A'::tsvector)`
结果	`(cat,{3},{D}) ...`

> **注意**
>
> 所有接受一个可选的 `regconfig` 参数的文本检索函数在省略该参数时，会使用由 default_text_search_config 指定的配置。

表 9.42 中的函数被单独列出，因为它们通常不被用于日常的文本检索操作。它们有助于开发和调试新的文本检索配置。

表 9.42. 文本检索调试函数

函数	`ts_debug([ config regconfig, ] document text, OUT alias text, OUT description text, OUT token text, OUT dictionaries regdictionary[], OUT dictionary regdictionary, OUT lexemes text[])`
返回类型	`setof record`
描述	测试配置
示例	`ts_debug('english', 'The Brightest supernovaes')`
结果	`(asciiword,"Word, all ASCII",The,{english_stem},english_stem,{}) ...`
函数	`ts_lexize(dict regdictionary, token text)`
返回类型	`text[]`
描述	测试词典
示例	`ts_lexize('english_stem', 'stars')`
结果	`{star}`
函数	`ts_parse(parser_name text, document text, OUT tokid integer, OUT token text)`
返回类型	`setof record`
描述	测试解析器
示例	`ts_parse('default', 'foo - bar')`
结果	`(1,foo) ...`
函数	`ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text)`
返回类型	`setof record`
描述	测试解析器
示例	`ts_parse(3722, 'foo - bar')`
结果	`(1,foo) ...`

函数	ts_token_type(*parser_name* text, OUT *tokid* integer, OUT *alias* text, OUT *description* text)
返回类型	setof record
描述	获取解析器定义的词元类型
示例	ts_token_type('default')
结果	(1,asciiword,"Word, all ASCII") ...
函数	ts_token_type(*parser_oid* oid, OUT *tokid* integer, OUT *alias* text, OUT *description* text)
返回类型	setof record
描述	获取解析器定义的词元类型
示例	ts_token_type(3722)
结果	(1,asciiword,"Word, all ASCII") ...
函数	ts_stat(*sqlquery* text, [*weights* text,] OUT *word* text, OUT *ndoc* integer, OUT *nentry* integer)
返回类型	setof record
描述	获取 tsvector 列的统计信息
示例	ts_stat('SELECT vector from apod')
结果	(foo,10,15) ...

9.14. XML 函数

本节中描述的函数以及类函数表达式都作用于类型 xml 的值。关于 xml 类型的详细信息请参见第 8.13 节。用于在值与类型 xml 之间转换的类函数表达式 xmlparse 和 xmlserialize 记录在该节中，而不是本节。

使用大部分这些函数要求 PostgreSQL 使用了 configure --with-libxml 进行编译。

9.14.1. 产生 XML 内容

有一组函数和类函数的表达式可以用来从 SQL 数据产生 XML 内容。它们特别适合于将查询结果格式化成 XML 文档以便于在客户端应用中处理。

9.14.1.1. xmlcomment

xmlcomment(*text*)

函数 xmlcomment 创建一个 XML 值，其中包含以指定文本为内容的 XML 注释。该文本不能包含“--”，也不能以“-”结尾，以确保构造出的 XML 注释有效。如果参数为空值，结果也为空值。

示例：

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

9.14.1.2. xmlconcat

```
xmlconcat(xml[, ...])
```

函数 xmlconcat 将由各个独立 XML 值组成的列表串接成一个单独的值，这个值包含一个 XML 内容片断。空值会被忽略，只有当没有参数为非空时结果才为空。

示例：

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

如果 XML 声明存在，它们会按照下面的方式被组合。如果所有的参数值都有相同的 XML 版本声明，该版本将被用在结果中，否则将不使用版本。如果所有参数值有独立声明值“yes”，那么该值将被用在结果中。如果所有参数值都有一个独立声明值并且至少有一个为“no”，则“no”被用在结果中。否则结果中将没有独立声明。如果确定结果需要独立声明但不需要版本声明，将会使用一个版本 1.0 的版本声明，因为 XML 要求一个 XML 声明要包含一个版本声明。编码声明会被忽略并且在所有情况中都会被移除。

示例：

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/>');

            xmlconcat
----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

9.14.1.3. xmlelement

```
xmlelement(name name [, xmlattributes(value [AS attname] [, ... ])] [, content, ...])
```

表达式 xmlelement 使用给定的名称、属性和内容生成一个 XML 元素。

示例：

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

             xmlelement
------------------------------------
 <foo bar="2007-01-26">content</foo>
```

不是合法 XML 名字的元素名和属性名将被转义，转义的方法是将不合法的字符用序列 *_xHHHH_* 替换，其中 *HHHH* 是被替换字符的 Unicode 代码点的十六进制表示。例如：

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

             xmlelement
-----------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

如果属性值是一个列引用，则不需要指定一个显式的属性名，在这种情况下列的名字将被默认用于属性的名字。在其他情况下，属性必须被给定一个显式名称。因此这个示例是合法的：

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

但是下面这些不合法：

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

如果指定了元素内容，它们将被根据其数据类型格式化。如果内容本身也是类型xml，就可以构建复杂的 XML 文档。例如：

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

其他类型的内容将被格式化为合法的 XML 字符数据。这意味着字符 <、> 和 & 将被转换为实体。二进制数据（数据类型bytea）将被表示成 base64 或十六进制编码，具体取决于配置参数 xmlbinary 的设置。为了使 PostgreSQL 的映射与 SQL:2006 及以后标准中指定的映射保持一致，个别数据类型的特殊行为将不断发展，正如第 D.3.1.3 节中讨论的那样。

9.14.1.4. xmlforest

```
xmlforest(content [AS name] [, ...])
```

表达式 xmlforest 使用给定的名称和内容生成由元素构成的 XML 森林（序列）。

示例：

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

                                       xmlforest
-----------------------------------------------------------------------------------------
 <table_name>pg_authid</table_name><column_name>rolname</column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</column_name>
 ...
```

如第二个示例所示，如果内容值是一个列引用，可以省略元素名称，此时默认使用列名。否则，必须指定名称。

如上文 xmlelement 所示，非法 XML 名字的元素名会被转义。相似地，内容数据也会被转义来产生合法的 XML 内容，除非它已经是一个xml 类型。

注意如果 XML 森林由多于一个元素组成，那么它不是合法的 XML 文档，因此在xmlelement 中包装 xmlforest 表达式会有用处。

9.14.1.5. xmlpi

```
xmlpi(name target [, content])
```

表达式 xmlpi 创建一个 XML 处理指令。如果提供了内容，其中不得包含字符序列 ?>。

示例：

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

9.14.1.6. xmlroot

```
xmlroot(xml, version text | no value [, standalone yes|no|no value])
```

表达式 xmlroot 修改一个 XML 值的根结点的属性。如果指定了一个版本，它会替换根节点的版本声明中的值；如果指定了一个独立设置，它会替换根节点的独立声明中的值。

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
----------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

9.14.1.7. xmlagg

```
xmlagg(xml)
```

和这里描述的其他函数不同，函数xmlagg 是一个聚合函数。它将聚合函数调用的输入值串接起来，非常像xmlconcat 所做的事情，除了串接是跨行发生的而不是在单一行的多个表达式上发生。聚合函数的更多信息请见第 9.20 节。

示例：

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
----------------------
 <foo>abc</foo><bar/>
```

为了决定串接的顺序，可以为聚合调用增加一个ORDER BY 子句，如第 4.2.7 节中所述。例如：

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

以前的版本曾推荐使用下列非标准方法，它在某些情况下可能仍然有用：

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

9.14.2. XML 谓词

这一节描述的表达式检查 xml 值的属性。

9.14.2.1. IS DOCUMENT

```
xml IS DOCUMENT
```

如果参数 XML 值是一个正确的 XML 文档，则IS DOCUMENT 返回真，如果不是则返回假（即它是一个内容片断），或者是参数为空时返回空。文档和内容片断之间的区别请见第 8.13 节。

9.14.2.2. IS NOT DOCUMENT

```
xml IS NOT DOCUMENT
```

如果参数中的 XML 值是一个正确的 XML 文档，那么表达式IS NOT DOCUMENT 返回假，否则返回真（也就是说它是一个内容片段），如果参数为空则返回空。

9.14.2.3. XMLEXISTS

```
XMLEXISTS(text PASSING [BY REF] xml [BY REF])
```

函数 xmlexists 对一个 XPath 1.0 表达式（第一个参数）求值，以传递的 XML 值作为其上下文项。如果求值结果产生一个空节点集，该函数返回 false，如果产生任何其他值，则返回 true。如果任何参数为空，则函数返回 null。作为上下文项传递的非空值必须是一个 XML 文档，而不是内容片段或任何非 XML 值。

示例：

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY REF '<towns><town>Toronto</
town><town>Ottawa</town></towns>');

 xmlexists
------------
 t
(1 row)
```

PostgreSQL 接受 BY REF 子句，但会忽略它们，如第 D.3.2 节所述。在 SQL 标准中，xmlexists 函数计算 XML 查询语言表达式，但 PostgreSQL 只允许 XPath 1.0 表达式，如第 D.3.1 节所述。

9.14.2.4. xml_is_well_formed

```
xml_is_well_formed(text)
xml_is_well_formed_document(text)
xml_is_well_formed_content(text)
```

这些函数检查一个 text 串是不是一个良构的 XML，返回一个布尔结果。xml_is_well_formed_document 检查一个良构的文档，而xml_is_well_formed_content 检查良构的内容。如果 xmloption 配置参数被设置为 DOCUMENT，xml_is_well_formed 会做第一个函数的工作；如果配置参数被设置为 CONTENT，它会做第二个函数的工作。这意味着 xml_is_well_formed 对于检查一个到类型 xml 的简单类型转换是否会成功非常有用，而其他两个函数对于检查XMLPARSE 的对应变体是否会成功有用。

示例：

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</pg:
foo>');
 xml_is_well_formed_document
-----------------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</my:
foo>');
 xml_is_well_formed_document
-----------------------------
 f
(1 row)
```

最后一个示例表明，这些检查也会检查命名空间是否正确匹配。

9.14.3. 处理 XML

为了处理 xml 类型的值，PostgreSQL 提供了用于计算 XPath 1.0 表达式的xpath 和 xpath_exists 函数，以及XMLTABLE 表函数。

9.14.3.1. xpath

xpath(*xpath*, *xml* [, *nsarray*])

函数 xpath 根据 XML 值 *xml* 计算 XPath 1.0 表达式 *xpath*（一个 text 值）。它返回一个 XML 值数组，对应于 XPath 表达式产生的结点集合。如果 XPath 表达式返回标量值而不是结点集合，则返回只含一个元素的数组。

第二个参数必须是一个良构的 XML 文档。特殊地，它必须有一个单一根结点元素。

该函数可选的第三个参数是一个名字空间映射数组。这个数组应为二维 text 数组，其第二维长度等于 2（也就是说，它应当是一个数组的数组，其中每个子数组都恰好包含 2 个元素）。每个数组项的第一个元素是名字空间名称（别名），第二个元素是名字空间 URI。这个数组中提供的

别名不要求与 XML 文档本身中使用的别名相同；换句话说，在 XML 文档中以及在xpath 函数的上下文中，别名都是局部的。

示例：

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

要处理默认（匿名）命名空间，做这样的事情：

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

9.14.3.2. xpath_exists

```
xpath_exists(xpath, xml [, nsarray])
```

函数 xpath_exists 是 xpath 函数的一种特殊形式。这个函数不是返回满足 XPath 1.0 表达式的各个 XML 值，它返回一个布尔值表示查询是否被满足（具体来说，它是否产生了空节点集以外的任何值）。这个函数等价于标准的XMLEXISTS 谓词，不过它还提供了对一个名字空间映射参数的支持。

示例：

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
                    ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

9.14.3.3. xmltable

```
xmltable( [XMLNAMESPACES(namespace uri AS namespace name[, ...]), ]
          row_expression PASSING [BY REF] document_expression [BY REF]
          COLUMNS name { type [PATH column_expression] [DEFAULT default_expression] [NOT NULL |
 NULL]
                       | FOR ORDINALITY }
                  [, ...]
)
```

xmltable 函数根据给定的 XML 值、用于提取行的 XPath 过滤器和一组列定义生成表。

可选的 XMLNAMESPACES 子句是一个以逗号分隔的命名空间列表。它指定文档中使用的 XML 命名空间及其别名。目前不支持指定默认命名空间。

必需的参数 *row_expression* 是一个 XPath 1.0 表达式，以 *document_expression* 作为上下文项进行计算，从而获得一个 XML 结点集合。xmltable 将这些结点转换为输出行。如果 *document_expression* 为空值，或者 *row_expression* 产生空结点集合或结点集合以外的任何值，则不会产生任何行。

document_expression 为 *row_expression* 提供上下文项。它必须是良构的 XML 文档；不接受片段或森林。BY REF 子句会被接受，但会被忽略，如第 D.3.2 节所述。在 SQL 标准中，xmltable 函数计算 XML 查询语言表达式，但 PostgreSQL 只允许 XPath 1.0 表达式，如第 D.3.1 节所述。

必需的 COLUMNS 子句指定输出表的列列表。每一项描述一列，其格式见上面的语法概要。列名和类型是必需的；路径、默认值和可空性子句是可选的。

被标记为 FOR ORDINALITY 的列将按照从 *row_expression* 的结果节点集中检索到的节点的顺序，从 1 开始，填充行号。最多只能有一个列被标记为FOR ORDINALITY。

> **注意**
>
> XPath 1.0 并没有为节点集中的节点指定顺序，因此依赖特定结果顺序的代码将取决于实现。详情请参见第 D.3.1.2 节。

列的 *column_expression* 是一个 XPath 1.0 表达式，它对每一行都要进行求值，并以*row_expression* 结果中的当前节点作为其上下文项，以找到列的值。如果没有给出 *column_expression*，那么列名被用作隐式路径。

如果列的 XPath 表达式返回非 XML 值（在 XPath 1.0 中仅限字符串、布尔值或双精度值），且该列的 PostgreSQL 类型不是 xml，则会按照将该值的字符串表示赋给 PostgreSQL 类型的方式设置该列。在此版本中，XPath 布尔值或双精度结果必须显式转换为字符串（也就是在原列表达式外包裹 XPath 1.0 的 string 函数）；这样 PostgreSQL 才能成功地将字符串赋给布尔或双精度类型的 SQL 结果列。这些转换规则与 SQL 标准不同，如第 D.3.1.3 节所述。

在此版本中，对于类型为 xml 的 SQL 结果列，或者求值结果为 XML 类型的列 XPath 表达式（无论输出列的 SQL 类型是什么），都按第 D.3.2 节所述处理；此行为在 PostgreSQL 12 中有重大变化。

如果路径表达式为给定行返回一个空节点集（通常情况下，当它不匹配时），该列将被设置为 NULL，除非指定了*default_expression*；如果指定了该表达式，则使用对它求值所得到的值。

列可以标记为 NOT NULL。如果某个 NOT NULL 列的 *column_expression* 没有匹配项，并且未指定DEFAULT（或者指定的 *default_expression* 求值结果也为空），就会报错。

default_expression 并不在调用 xmltable 时立即求值，而是在每次该列需要默认值时才求值。如果表达式是稳定的或不可变的，则可能跳过重复求值。因此，你可以在 *default_expression* 中使用 nextval 这样的易失性函数。

示例：

```
CREATE TABLE xmldata AS SELECT
xml $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>AU</COUNTRY_ID>
    <COUNTRY_NAME>Australia</COUNTRY_NAME>
  </ROW>
  <ROW id="5">
    <COUNTRY_ID>JP</COUNTRY_ID>
    <COUNTRY_NAME>Japan</COUNTRY_NAME>
    <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
    <SIZE unit="sq_mi">145935</SIZE>
  </ROW>
  <ROW id="6">
    <COUNTRY_ID>SG</COUNTRY_ID>
    <COUNTRY_NAME>Singapore</COUNTRY_NAME>
    <SIZE unit="sq_km">697</SIZE>
  </ROW>
</ROWS>
$$ AS data;

SELECT xmltable.*
  FROM xmldata,
       XMLTABLE('//ROWS/ROW'
                PASSING data
                COLUMNS id int PATH '@id',
                        ordinality FOR ORDINALITY,
                        "COUNTRY_NAME" text,
                        country_id text PATH 'COUNTRY_ID',
                        size_sq_km float PATH 'SIZE[@unit = "sq_km"]',
                        size_other text PATH
                             'concat(SIZE[@unit!="sq_km"], " ", SIZE[@unit!="sq_km"]/@unit)',
                        premier_name text PATH 'PREMIER_NAME' DEFAULT 'not specified') ;

 id | ordinality | COUNTRY_NAME | country_id | size_sq_km |  size_other  | premier_name
----+------------+--------------+------------+------------+--------------+--------------
  1 |          1 | Australia    | AU         |            |              | not specified
  5 |          2 | Japan        | JP         |            | 145935 sq_mi | Shinzo Abe
  6 |          3 | Singapore    | SG         |        697 |              | not specified
```

下面的示例展示了多个 text() 节点的连接、将列名用作 XPath 过滤器，以及对空白、XML 注释和处理指令的处理方式：

```
CREATE TABLE xmlelements AS SELECT
xml $$
  <root>
   <element>  Hello<!-- xyxxz -->2a2<?aaaaa?> <!--x-->  bbb<x>xxx</x>CC  </element>
  </root>
$$ AS data;

SELECT xmltable.*
  FROM xmlelements, XMLTABLE('/root' PASSING data COLUMNS element text);
       element
----------------------
   Hello2a2   bbbCC
```

下面的示例展示了如何使用 XMLNAMESPACES 子句指定用在 XML 文档以及 XPath 表达式中的名字空间列表：

```
WITH xmldata(data) AS (VALUES ('
<example xmlns="http://example.com/myns" xmlns:B="http://example.com/b">
 <item foo="1" B:bar="2"/>
 <item foo="3" B:bar="4"/>
 <item foo="4" B:bar="5"/>
</example>'::xml)
)
SELECT xmltable.*
  FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns' AS x,
                              'http://example.com/b' AS "B"),
              '/x:example/x:item'
                 PASSING (SELECT data FROM xmldata)
                 COLUMNS foo int PATH '@foo',
                   bar int PATH '@B:bar');
 foo | bar
-----+-----
   1 |   2
   3 |   4
   4 |   5
(3 rows)
```

9.14.4. 将表映射到 XML

以下函数将关系表的内容映射为 XML 值，可以将它们视为 XML 导出功能：

```
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean,
              tableforest boolean, targetns text)
```

每个函数的返回类型都是 xml。

table_to_xml 映射由参数 *tbl* 传递的命名表的内容。regclass 类型接受使用常见标记标识表的字符串，包括可选的模式限定和双引号。query_to_xml 执行由参数 *query* 传递的查询并且映射结果集。cursor_to_xml 从 *cursor* 指定的游标中取出指定数量的行。如果需要映射一个大型的表，我们推荐这种变体，因为每一个函数都是在内存中构建结果值的。

如果 *tableforest* 为假，则结果的 XML 文档看起来像这样：

```
<tablename>
  <row>
    <columnname1>data</columnname1>
    <columnname2>data</columnname2>
  </row>

  <row>
    ...
  </row>

  ...
</tablename>
```

如果 *tableforest* 为真，结果是一个看起来像这样的 XML 内容片断：

```
<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...
```

如果没有表名可用，在映射一个查询或一个游标时，在第一种格式中使用串`table`，在第二种格式中使用`row`。

这几种格式的选择由用户决定。第一种格式是一个正确的 XML 文档，它在很多应用中都很重要。如果结果值要被重组为一个文档，第二种格式在`cursor_to_xml` 函数中更有用。前文讨论的产生 XML 内容的函数（特别是 `xmlelement`）可以被用来把结果修改成符合用户的要求。

数据值会被以前文的函数 `xmlelement` 中描述的相同方法映射。

参数 *nulls* 决定空值是否会被包含在输出中。如果为真，列中的空值被表示为：

```
<columnname xsi:nil="true"/>
```

其中 `xsi` 是 XML 模式实例的 XML 名字空间前缀。一个合适的名字空间声明将被加入到结果值中。如果为假，包含空值的列将被从输出中忽略掉。

参数 *targetns* 指定想要的结果的 XML 名字空间。如果不需要特定的名字空间，应传入一个空字符串。

下面的函数返回 XML 模式文档，这些文档描述上述对应函数所执行的映射：

```
table_to_xmlschema ( table regclass, nulls boolean,
                     tableforest boolean, targetns text ) → xml
query_to_xmlschema ( query text, nulls boolean,
                     tableforest boolean, targetns text ) → xml
cursor_to_xmlschema ( cursor refcursor, nulls boolean,
                      tableforest boolean, targetns text ) → xml
```

必须传入相同的参数，才能得到彼此匹配的 XML 数据映射和 XML 模式文档。

下面的函数产生 XML 数据映射和对应的 XML 模式，并把产生的结果链接在一起放在一个文档（或森林）中。在要求自包含和自描述的结果时，它们非常有用：

```
table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
```

此外，还可以使用以下函数，为整个模式或整个当前数据库生成类似的映射：

```
schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)
```

注意，这些函数可能生成大量数据，需要在内存中构建。请求大型模式或数据库的内容映射时，可以考虑分别映射各个表，甚至通过游标来完成。

一个模式内容映射的结果看起来像这样：

```
<schemaname>

table1-mapping

table2-mapping

...

</schemaname>
```

其中一个表映射的格式取决于上文解释的 *tableforest* 参数。

一个数据库内容映射的结果看起来像这样：

```
<dbname>

<schema1name>
  ...
</schema1name>

<schema2name>
  ...
</schema2name>

...

</dbname>
```

其中的模式映射如上所述。

作为一个使用这些函数产生的输出的示例，例 9.1 展示了一个 XSLT 样式表，它将`table_to_xml_and_xmlschema` 的输出转换为一个以表格形式呈现表数据的 HTML 文档。以一种相似的方式，这些函数的结果可以被转换成其他基于 XML 的格式。

例 9.1. 转换 SQL/XML 输出到 HTML 的 XSLT 样式表

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/xsd:
element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/xsd:sequence/xsd:
element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

9.15. JSON 函数和操作符

表 9.43 列出了可用于两种 JSON 数据类型的操作符（参见第 8.14 节）。

表 9.43. `json` 和 `jsonb` 操作符

操作符	`->`
右操作数类型	`int`
描述	获取 JSON 数组元素（索引从零开始，负整数从末尾计数）
示例	`'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2`
示例结果	`{"c":"baz"}`
操作符	`->`
右操作数类型	`text`

描述	按键获取 JSON 对象字段
示例	`'{"a": {"b":"foo"}}'::json->'a'`
示例结果	`{"b":"foo"}`
操作符	`->>`
右操作数类型	`int`
描述	以 `text` 形式获取 JSON 数组元素
示例	`'[1,2,3]'::json->>2`
示例结果	`3`
操作符	`->>`
右操作数类型	`text`
描述	以 `text` 形式获取 JSON 对象字段
示例	`'{"a":1,"b":2}'::json->>'b'`
示例结果	`2`
操作符	`#>`
右操作数类型	`text[]`
描述	获取指定路径处的 JSON 对象
示例	`'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'`
示例结果	`{"c": "foo"}`
操作符	`#>>`
右操作数类型	`text[]`
描述	以 `text` 形式获取指定路径处的 JSON 对象
示例	`'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'`
示例结果	`3`

注意

这些操作符针对 `json` 和 `jsonb` 类型都有相应的变体。字段、元素和路径提取操作符返回的类型与其左侧输入相同（`json` 或 `jsonb`），但标明返回 `text` 的操作符会将值转换为文本。如果 JSON 输入的结构不符合请求，例如所需元素不存在，字段、元素和路径提取操作符会返回 NULL，而不会失败。接受整数 JSON 数组下标的字段、元素和路径提取操作符都支持使用负下标从数组末尾计数。

表 9.1 中给出的常规比较操作符也可用于 `jsonb`，但不适用于`json`。比较操作符遵循 B-树操作的排序规则，详见第 8.14.4 节。另请参阅第 9.20 节，了解聚合函数`json_agg` 如何将记录值聚合为 JSON，以及聚合函数`json_object_agg` 如何将值对聚合为 JSON 对象，还有它们对应的 `jsonb` 函数，`jsonb_agg` 和 `jsonb_object_agg`。

还有一些操作符只适用于 jsonb，如表 9.44 所示。其中许多操作符可以通过 jsonb 操作符类使用索引。关于 jsonb 包含与存在语义的完整说明，请参见第 8.14.3 节。第 8.14.4 节介绍了如何使用这些操作符有效地为 jsonb 数据建立索引。

表 9.44. 附加的 jsonb 操作符

操作符	右操作数类型	描述	示例
@>	jsonb	左侧 JSON 值是否在顶层包含右侧 JSON 路径/值条目?	'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb
<@	jsonb	左侧 JSON 路径/值条目是否包含在右侧 JSON 值的顶层?	'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb
?	text	该字符串是否作为顶层键存在于 JSON 值中?	'{"a":1, "b":2}'::jsonb ? 'b'
?|	text[]	这些数组字符串中是否有任意一个作为顶层键存在?	'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']
?&	text[]	这些数组字符串是否都作为顶层键存在?	'["a", "b"]'::jsonb ?& array['a', 'b']
||	jsonb	将两个 jsonb 值串接为一个新的 jsonb 值	'["a", "b"]'::jsonb || '["c", "d"]'::jsonb
-	text	从左操作数中删除键/值对或字符串元素。键/值对按其键进行匹配。	'{"a": "b"}'::jsonb - 'a'
-	text[]	从左操作数中删除多个键/值对或字符串元素。键/值对按其键进行匹配。	'{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[]
-	integer	删除指定索引的数组元素（负整数从末尾计数）。如果顶层容器不是数组，则抛出错误。	'["a", "b"]'::jsonb - 1
#-	text[]	删除指定路径处的字段或元素（对于 JSON 数组，负整数从末尾计数）	'["a", {"b":1}]'::jsonb #- '{1,b}'

> **注意**
>
> || 操作符连接两个 JSON 对象时，会生成一个包含两者键的并集的对象；遇到重复键时，采用第二个对象的值。其他情况都会生成 JSON 数组：首先将任何非数组输入转换为单元素数组，然后连接两个数组。该操作不递归，只合并顶层数组或对象结构。

表 9.45 列出了可用于创建 json 和 jsonb 值的函数。（row_to_json 和 array_to_json 函数没有对应的 jsonb 函数，但 to_jsonb 函数提供了大致相同的功能。）

表 9.45. JSON 创建函数

函数	描述	示例	示例结果
to_json(anyelement) to_jsonb(anyelement)	将值作为 json 或 jsonb 返回。数组和复合值分别递归转换为数组和对象；否则，如果存在从该类型到 json 的类型转换，则使用该转换函数执行转换；否则生成标量值。对于数值、布尔值或 null 以外的任何标量类型，将使用其文本表示，并使其成为有效的 json 或 jsonb 值。	to_json('Fred said "Hi."'::text)	"Fred said \"Hi.\""
array_to_json(anyarray [, pretty_bool])	将数组作为 JSON 数组返回。PostgreSQL 多维数组会变成由数组组成的 JSON 数组。如果 *pretty_bool* 为真，则在第一维元素之间添加换行。	array_to_json('{{1,5},{99,100}}'::int[])	[[1,5],[99,100]]
row_to_json(record [, pretty_bool])	将行作为 JSON 对象返回。如果 *pretty_bool* 为真，则在第一层元素之间添加换行。	row_to_json(row(1,'foo'))	{"f1":1,"f2":"foo"}
json_build_array(VARIADIC "any") jsonb_build_array(VARIADIC "any")	从可变参数列表构造 JSON 数组，各元素可以具有不同类型。	json_build_array(1,2,'3',4,5)	[1, 2, "3", 4, 5]
json_build_object(VARIADIC "any") jsonb_build_object(VARIADIC "any")	从可变参数列表构造 JSON 对象。按惯例，参数列表由键和值交替组成。	json_build_object('foo',1,'bar',2)	{"foo": 1, "bar": 2}
json_object(text[]) jsonb_object(text[])	从 text 数组构造 JSON 对象。该数组必须是一维且包含偶数个成员，此时将成员按交替的键/值对处理；或者是二维数组，且每个内部数组恰好有两个元素，将这两个元素作为一个键/值对。	json_object('{a, 1, b, "def", c, 3.5}') json_object('{{a, 1},{b, "def"},{c, 3.5}}')	{"a": "1", "b": "def", "c": "3.5"}
json_object(keys text[], values text[]) jsonb_object(keys text[], values text[])	这种形式的 json_object 从两个独立数组中成对获取键和值。除此之外，它与单参数形式完全相同。	json_object('{a, b}', '{1,2}')	{"a": "1", "b": "2"}

> **注意**
>
> 除了提供美化输出选项外，`array_to_json` 和 `row_to_json` 的行为与 `to_json` 相同。针对 `to_json` 描述的行为同样适用于其他 JSON 创建函数转换的每个值。

> **注意**
>
> hstore 扩展提供了从 `hstore` 到 `json` 的类型转换，因此经由 JSON 创建函数转换的 `hstore` 值会表示为 JSON 对象，而不是基本的字符串值。

表 9.46 显示可用于处理 `json` 和 `jsonb` 值的函数。

表 9.46. JSON 处理函数

函数	`json_array_length(json)`
	
	`jsonb_array_length(jsonb)`

返回类型	`int`
描述	返回最外层 JSON 数组的元素数量。
示例	`json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]')`
示例结果	`5`

函数	`json_each(json)`
	
	`jsonb_each(jsonb)`

返回类型	`setof key text, value json`
	
	`setof key text, value jsonb`

描述	将最外层 JSON 对象展开为一组键/值对。
示例	`select * from json_each('{"a":"foo", "b":"bar"}')`

示例结果	`key | value`
	`-----+-------`
	`a   | "foo"`
	`b   | "bar"`

函数	`json_each_text(json)`
	
	`jsonb_each_text(jsonb)`

返回类型	`setof key text, value text`
描述	将最外层 JSON 对象展开为一组键/值对。返回的值为 `text` 类型。
示例	`select * from json_each_text('{"a":"foo", "b":"bar"}')`

示例结果	``` key | value -----+------- a | foo b | bar ```

函数	`json_extract_path(from_json json, VARIADIC path_elems text[])`
	
	`jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])`

返回类型	`json`
	
	`jsonb`

描述	返回 *path_elems* 指向的 JSON 值（等价于 #> 操作符）。
示例	`json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4')`
示例结果	`{"f5":99,"f6":"foo"}`

函数	`json_extract_path_text(from_json json, VARIADIC path_elems text[])`
	
	`jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])`

返回类型	`text`
描述	以 `text` 形式返回 *path_elems* 指向的 JSON 值（等价于 #>> 操作符）。
示例	`json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4', 'f6')`
示例结果	`foo`

函数	`json_object_keys(json)`
	
	`jsonb_object_keys(jsonb)`

返回类型	`setof text`
描述	返回最外层 JSON 对象的键集合。
示例	`json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}')`
示例结果	``` json_object_keys ------------------ f1 f2 ```

函数	`json_populate_record(base anyelement, from_json json)`
	
	`jsonb_populate_record(base anyelement, from_json jsonb)`

返回类型	`anyelement`
描述	将 *from_json* 中的对象展开为一行，其列与 *base* 定义的记录类型相匹配（见下注）。
示例	`select * from json_populate_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}}')`

示例结果	``` a | b | c ---+-----------+------------- 1 | {2,"a b"} | (4,"a b c") ```

函数	`json_populate_recordset(base anyelement, from_json json)`
	
	`jsonb_populate_recordset(base anyelement, from_json jsonb)`

返回类型	`setof anyelement`
描述	将 *from_json* 中最外层的对象数组展开为一组行，其列与 *base* 定义的记录类型相匹配（见下注）。
示例	`select * from json_populate_recordset(null::myrowtype, '[{"a":1,"b":2},{"a":3,"b":4}]')`
示例结果	``` a | b ---+--- 1 | 2 3 | 4 ```

函数	`json_array_elements(json)`
	
	`jsonb_array_elements(jsonb)`

返回类型	`setof json`
	
	`setof jsonb`

描述	将 JSON 数组展开为一组 JSON 值。
示例	`select * from json_array_elements('[1,true, [2,false]]')`
示例结果	``` value ----------- 1 true [2,false] ```

函数	`json_array_elements_text(json)`
	
	`jsonb_array_elements_text(jsonb)`

返回类型	`setof text`
描述	将 JSON 数组展开为一组 `text` 值。
示例	`select * from json_array_elements_text('["foo", "bar"]')`
示例结果	``` value ----------- foo bar ```

函数	`json_typeof(json)`
	
	`jsonb_typeof(jsonb)`

返回类型	`text`

描述	以文本字符串形式返回最外层 JSON 值的类型。可能的类型为 `object`、`array`、`string`、`number`、`boolean` 和 `null`。
示例	`json_typeof('-123.4')`
示例结果	`number`

函数	`json_to_record(json)`
	
	`jsonb_to_record(jsonb)`

返回类型	`record`
描述	从 JSON 对象构造任意记录（见下注）。与所有返回 `record` 的函数一样，调用者必须使用 `AS` 子句显式定义记录结构。
示例	`select * from json_to_record('{"a":1,"b":[1,2,3],"c":[1,2,3], "e":"bar","r": {"a": 123, "b": "a b c"}}') as x(a int, b text, c int[], d text, r myrowtype)`

示例结果	`a |  b  |  c  | d |  r`
	`---+---------+---------+---+---------------`
	`1 | [1,2,3] | {1,2,3} |  | (123,"a b c")`

函数	`json_to_recordset(json)`
	
	`jsonb_to_recordset(jsonb)`

返回类型	`setof record`
描述	从 JSON 对象数组构造任意记录集合（见下注）。与所有返回 `record` 的函数一样，调用者必须使用 `AS` 子句显式定义记录结构。
示例	`select * from json_to_recordset('[{"a":1,"b":"foo"},{"a":"2", "c":"bar"}]') as x(a int, b text);`

示例结果	`a |  b`
	`---+-----`
	`1 | foo`
	`2 |`

函数	`json_strip_nulls(from_json json)`
	
	`jsonb_strip_nulls(from_json jsonb)`

返回类型	`json`
	
	`jsonb`

描述	返回移除了所有值为 null 的对象字段的 *`from_json`*。其他 null 值保持不变。
示例	`json_strip_nulls('[{"f1":1,"f2":null},2,null,3]')`
示例结果	`[{"f1":1},2,null,3]`
函数	`jsonb_set(target jsonb, path text[], new_value jsonb [, create_missing boolean])`
返回类型	`jsonb`
描述	返回将 *`path`* 指定部分替换为 *`new_value`* 后的 *`target`*；如果 *`create_missing`* 为真（默认为 `true`），且 *`path`* 指定的项不存在，则添加 *`new_value`*。与面向路径的操作符一样，*`path`* 中的负整数从 JSON 数组末尾计数。

示例	`jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}','[2,3,4]', false)`
	
	`jsonb_set('[{"f1":1,"f2":null},2]', '{0,f3}','[2,3,4]')`

示例结果	`[{"f1":[2,3,4],"f2":null},2,null,3]`
	
	`[{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]`

函数	`jsonb_insert(target jsonb, path text[], new_value jsonb [, insert_after boolean])`
返回类型	`jsonb`
描述	返回插入 *new_value* 后的 *target*。如果 *path* 指定的 *target* 部分位于 JSONB 数组中，则将 *new_value* 插入目标之前；若 *insert_after* 为真，则插入目标之后（默认为 `false`）。如果 *path* 指定的 *target* 部分位于 JSONB 对象中，则仅在 *target* 不存在时插入 *new_value*。与面向路径的操作符一样，*path* 中的负整数从 JSON 数组末尾计数。

示例	`jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"')`
	
	`jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true)`

示例结果	`{"a": [0, "new_value", 1, 2]}`
	
	`{"a": [0, 1, "new_value", 2]}`

函数	`jsonb_pretty(from_json jsonb)`
返回类型	`text`
描述	将 *from_json* 作为带缩进的 JSON 文本返回。
示例	`jsonb_pretty('[{"f1":1,"f2":null},2,null,3]')`

示例结果	`[`
	`  {`
	`  "f1": 1,`
	`  "f2": null`
	`  },`
	`  2,`
	`  null,`
	`  3`
	`]`

注意

这些函数和操作符中有许多会将 JSON 字符串中的 Unicode 转义转换为相应的单个字符。对于 `jsonb` 输入，这不成问题，因为转换已经完成；但对于 `json` 输入，这可能引发错误，如第 8.14 节所述。

注意

函数 `json[b]_populate_record`, `json[b]_populate_recordset`, `json[b]_to_record` 和 `json[b]_to_recordset` 处理 JSON 对象或对象数组，提取

与输出行类型的列名同名的键所对应的值。不对应任何输出列名的对象字段会被忽略，而不匹配任何对象字段的输出列会填入空值。将 JSON 值转换为输出列的 SQL 类型时，依次应用以下规则：

- JSON 空值在任何情况下都转换为 SQL 空值。
- 如果输出列的类型是 `json` 或 `jsonb`，则 JSON 值会被原样保留。
- 如果输出列是复合(行)类型，且 JSON 值是 JSON 对象，则该对象的字段会通过递归应用这些规则，被转换为输出行类型的各列。
- 同样，如果输出列是数组类型，而 JSON 值是 JSON 数组，则会通过递归应用这些规则，把 JSON 数组的元素转换为输出数组的元素。
- 否则，如果 JSON 值是字符串字面量，则将字符串的内容传给该列数据类型的输入转换函数。
- 否则，JSON 值的普通文本表示会被送入该列数据类型的输入转换函数。

虽然这些函数的示例使用常量，但典型用法是在 `FROM` 子句中引用一个表，并将它的某个 `json` 或 `jsonb` 列用作函数参数。随后可在查询的其他部分（如 `WHERE` 子句和目标列表）引用提取出的键值。与使用逐键操作符分别提取相比，以这种方式提取多个值可以提高性能。

注意

`jsonb_set` 和 `jsonb_insert` 的 `path` 参数中，除最后一项外的所有项都必须已存在于 `target` 中。如果 `create_missing` 为假，`jsonb_set` 的 `path` 参数的所有项都必须存在。如果不满足这些条件，则原样返回 `target`。

如果路径的最后一项是对象键，当该键不存在时会创建它，并赋予新值。如果路径的最后一项是数组下标，正值从左侧计数，负值从右侧计数来确定要设置的项；`-1` 表示最右侧的元素，以此类推。如果该项超出 -array_length .. array_length -1 的范围，且 create_missing 为真，则在下标为负时将新值添加到数组开头，为正时添加到数组末尾。

注意

不要将 `json_typeof` 函数返回的 `null` 与 SQL NULL 混淆。调用 `json_typeof( 'null'::json)` 会返回 `null`，而调用 `json_typeof(NULL::json)` 会返回 SQL NULL。

注意

如果 `json_strip_nulls` 的参数中有任何对象包含重复字段名，则结果的语义可能有所不同，具体取决于这些字段的出现顺序。`jsonb_strip_nulls` 没有这一问题，因为 `jsonb` 值不会包含重复的对象字段名。

9.16. 序列操作函数

本节描述用于操作序列对象（也称为序列生成器，或简称序列）的函数。序列对象是使用CREATE SEQUENCE 创建的特殊单行表。序列对象通常用于为表中的行生成惟一标识符。在表 9.47 中列出的序列函数，提供了简单的、多用户安全方法，用于从序列对象中获取连续的序列值。

表 9.47. 序列函数

函数	返回类型	描述
currval(regclass)	bigint	返回最近对指定序列调用 nextval 得到的值
lastval()	bigint	返回最近对任意序列调用 nextval 得到的值
nextval(regclass)	bigint	推进序列并返回新值
setval(regclass, bigint)	bigint	设置序列的当前值
setval(regclass, bigint, boolean)	bigint	设置序列的当前值和 is_called 标志

序列函数要操作的序列由一个 regclass 参数指定，该参数就是序列在pg_class 系统目录中的 OID。不过，不必手动查找 OID，因为regclass 数据类型的输入转换函数会替你完成这项工作。只需将序列名用单引号括起来，使其看上去像一个字面常量。为兼容普通SQL 名称的处理方式，除非序列名被双引号括起，否则字符串会转换为小写。因此：

```
nextval('foo')      操作序列 foo
nextval('FOO')      操作序列 foo
nextval('"Foo"')    操作序列 Foo
```

必要时，序列名可以带模式限定：

```
nextval('myschema.foo')      操作 myschema.foo
nextval('"myschema".foo')    同上
nextval('foo')               在搜索路径中查找 foo
```

参见第 8.19 节以获取更多有关以下类型的信息：regclass。

> **注意**
>
> 在 PostgreSQL 8.1 之前，序列函数的参数类型是 text，而不是 regclass；上述从文本字符串到 OID 值的转换会在每次调用时于运行期间执行。为保持向后兼容，这种能力仍然存在，但内部现在会在调用函数前，通过从 text 到 regclass 的隐式强制转换来处理。

将序列函数的参数写成不加修饰的字符串字面量时，它会成为以下类型的常量：`regclass`。由于它实际上只是一个 OID，即使后来发生重命名、模式变更等情况，它仍会指向最初标识的序列。这种“早绑定”行为通常适用于列默认值和视图中的序列引用。但有时你可能希望采用“后绑定”，在运行时解析序列引用。要获得后绑定行为，应强制将常量存储为`text` 常量，而不是`regclass`：

`nextval('foo'::text)` *foo* 运行时被查找

请注意，后绑定是下列旧版本唯一支持的行为：PostgreSQL 8.1 之前的版本，因此可能需要这样做来保留旧应用的语义。

当然，序列函数的参数既可以是常量，也可以是表达式。如果它是文本表达式，隐式强制转换就会导致运行时查找。

可用的序列函数如下：

`nextval`

将序列对象推进到下一个值并返回该值。这个操作是原子的：即使多个会话并发执行 `nextval`，每个会话也会安全地获得一个不同的序列值。

如果使用默认参数创建序列对象，连续调用 `nextval` 会返回从 1 开始的连续值。可以在 CREATE SEQUENCE 命令中使用特殊参数获得其他行为；更多信息请参见该命令的参考页。

这个函数需要序列上的 `USAGE` 或 `UPDATE` 权限。

`currval`

返回当前会话中最近一次针对该序列调用 `nextval` 所获得的值。（如果当前会话从未针对该序列调用过 `nextval`，则会报错。）由于返回的是会话局部值，无论其他会话是否在当前会话调用之后执行过 `nextval`，结果都是可预测的。

这个函数需要序列上的 `USAGE` 或 `SELECT` 权限。

`lastval`

返回当前会话中最近一次调用 `nextval` 所返回的值。此函数与 `currval` 相同，但不接受序列名参数，而是引用当前会话中最近一次调用 `nextval` 时所操作的序列。如果当前会话尚未调用过 `nextval`，调用 `lastval` 会报错。

该函数在最后使用的序列上需要 `USAGE` 或 `SELECT` 权限。

setval

重置序列对象的计数器值。双参数形式将序列的 last_value 字段设为指定值，并将其 is_called 字段设为 true，这意味着下次调用nextval 会在返回值之前推进序列。currval 报告的值也会设为指定值。在三参数形式中，is_called 可以设为 true 或 false。true 与双参数形式具有相同的效果。如果将它设为 false，下次调用nextval 将恰好返回指定值，并从再下一次调用nextval 时开始推进序列。此外，在这种情况下，currval 报告的值不会改变。例如：

```
SELECT setval('foo', 42);           下一次 nextval 将返回 43
SELECT setval('foo', 42, true);     同上
SELECT setval('foo', 42, false);    下一次 nextval 将返回 42
```

函数 setval 返回的结果就是其第二个参数的值。

这个函数在序列上需要 UPDATE 权限。

小心

为了避免阻塞从相同序列获取数字的并发事务，nextval 获得的值如果调用事务后续中止，就不会被重新使用。这意味着事务中止或数据库崩溃可能导致分配值序列中的间隙。这也可能发生在没有事务中止的情况下。例如，带有ON CONFLICT 子句的 INSERT 将计算要插入的元组，包括执行任何必需的nextval 调用，然后才检测到可能导致其遵循 ON CONFLICT 规则的任何冲突。因此，PostgreSQL 序列对象不能用于获取“无间隙”序列。

同样，setval 函数所做的序列状态更改立即对其他事务可见，并且如果调用事务回滚，则不会被撤消。

如果在包含 nextval 或 setval 调用的事务提交之前，数据库集簇崩溃，那么序列状态更改可能尚未传递到持久存储，因此在集簇重新启动后，无法确定序列是保持其原始状态还是更新状态。对于数据库内部使用序列而言，这是无害的，因为未提交事务的其他影响也不会可见。但是，如果您希望将序列值用于持久性的数据库外部用途，请确保在这样做之前已经提交了nextval 调用。

9.17. 条件表达式

本节描述在 PostgreSQL 中可用的 SQL 兼容的条件表达式。

提示

如果你的需求超过这些条件表达式的能力，你可能会希望用一种更富表现力的编程语言写一个服务器端函数。

9.17.1. CASE

SQL CASE 表达式是一种通用的条件表达式，类似于其它编程语言中的 if/else 语句：

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

CASE 子句可以用于任何表达式可以出现的地方。每一个 *condition* 是一个返回 boolean 结果的表达式。如果结果为真，那么CASE 表达式的结果就是紧随该条件后的 *result*，并且剩下的 CASE 表达式不会被处理。如果条件的结果不为真，那么以相同方式搜寻任何随后的WHEN 子句。如果没有 WHEN *condition* 为真，那么CASE 表达式的值就是在 ELSE 子句里的 *result*。如果省略了 ELSE 子句而且没有条件为真，结果为 null。

示例：

```
SELECT * FROM test;

 a
---
 1
 2
 3

SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

所有 *result* 表达式的数据类型都必须可以转换成单一的输出类型。参阅第 10.5 节获取细节。

下面这个“简单”形式的 CASE 表达式是上述通用形式的一个变种：

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

第一个 *expression* 会被计算，然后与所有在WHEN 子句中的每一个 *value* 对比，直到找到一个相等的。如果没有找到匹配的，则返回在ELSE 子句中的 *result*（或者 null 值）。这类似于 C 里的 switch 语句。

上面的示例可以用简单 CASE 语法来写：

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

CASE 表达式并不计算任何无助于判断结果的子表达式。例如，下面是一个可以避免被零除错误的方法：

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

> **注意**
>
> 如第 4.2.14 节所述，在多种情况下，表达式中的子表达式会在不同阶段求值，因此“CASE 只计算必要的子表达式”这一原则并非绝对成立。例如，常量子表达式1/0 通常会在规划时导致除零错误，即便它位于一个运行时永远不会进入的CASE 分支中也是如此。

9.17.2. COALESCE

```
COALESCE(value [, ...])
```

COALESCE 函数返回参数中第一个不为 null 的值。只有所有参数都为 null 时，才返回 null。它常用于在检索数据以供显示时，用默认值替换 null 值。例如：

```
SELECT COALESCE(description, short_description, '(none)') ...
```

此表达式返回 description，前提是它不为 null，否则返回short_description，前提是它不为 null，否则返回(none)。

所有参数都必须能转换为同一个数据类型，它将是结果的类型（详情参见第 10.5 节）。

和 CASE 表达式一样，COALESCE 只计算确定结果所需的参数；也就是说，在第一个不为 null 的参数右边的参数不会被计算。这个 SQL 标准函数提供了类似于NVL 和 IFNULL 的能力，它们被用在某些其他数据库系统中。

9.17.3. NULLIF

```
NULLIF(value1, value2)
```

当 *value1* 和 *value2* 相等时，NULLIF 返回 null。否则它返回 *value1*。这些可以用于执行前文给出的 COALESCE 示例的逆操作：

```
SELECT NULLIF(value, '(none)') ...
```

在这个示例中，如果value 是(none)，将返回 null，否则返回value 的值。

这两个参数必须具有可比较的类型。具体来说，它们的比较与你写的 *value1 = value2* 完全一样，因此必须有一个合适的=操作符可用。

结果的类型与第一个参数相同，但有一点细微的区别。实际上返回的是隐含 =操作符的第一个参数，在某些情况下，它将被提升以匹配第二个参数的类型。例如，NULLIF(1, 2.2) 生成numeric，因为没有integer = numeric 操作符，只有numeric = numeric。

9.17.4. GREATEST 和 LEAST

```
GREATEST(value [, ...])
```

```
LEAST(value [, ...])
```

GREATEST 和 LEAST 函数从由任意数量的表达式组成的列表中选取最大值或最小值。这些表达式都必须能转换为同一个数据类型，该类型将作为结果类型（详情参见第 10.5 节）。列表中的 NULL 值会被忽略。只有所有表达式的求值结果都为 NULL 时，结果才为 NULL。

请注意 GREATEST 和 LEAST 都未包含在 SQL 标准中，但却是很常见的扩展。某些其他数据库让它们在任何参数为 NULL 时返回 NULL，而不是在所有参数都为 NULL 时才返回 NULL。

9.18. 数组函数和操作符

表 9.48 列出了可用于数组类型的操作符。

表 9.48. 数组操作符

操作符	描述	示例	结果
=	等于	ARRAY[1.1,2.1,3.1] ::int[] = ARRAY[1, 2,3]	t
<>	不等于	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	小于	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	大于	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	小于等于	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	大于等于	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
@>	包含	ARRAY[1,4,3] @> ARRAY[3,1,3]	t

操作符	描述	示例	结果
<@	被包含	ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6]	t
&&	重叠（有公共元素）	ARRAY[1,4,3] && ARRAY[2,1]	t
||	数组与数组串接	ARRAY[1,2,3] || ARRAY[4,5,6]	{1,2,3,4,5,6}
||	数组与数组串接	ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
||	元素与数组串接	3 || ARRAY[4,5,6]	{3,4,5,6}
||	数组与元素串接	ARRAY[4,5,6] || 7	{4,5,6,7}

数组排序操作符（<、>= 等）使用元素数据类型的默认 B-树比较函数，逐个元素比较数组内容，并根据第一个差异决定排序顺序。多维数组的元素按行优先顺序访问（最后一个下标变化最快）。如果两个数组的内容相同，但维度信息不同，则由维度信息中的第一个差异决定排序顺序。（这与 PostgreSQL 8.2 之前的版本不同：旧版本会将内容相同的两个数组视为相等，即使它们的维数或下标范围不同。）

数组包含操作符（<@ 和 @>）在一个数组的每个元素都出现在另一个数组中时，认为前者包含在后者中。重复元素不会被特殊处理，因此 ARRAY[1] 和 ARRAY[1,1] 被认为彼此包含。

参阅第 8.15 节获取有关数组操作符行为的更多细节。有关哪些操作符支持索引操作，请参阅第 11.2 节。

表 9.49 展示了可以用于数组类型的函数。参阅第 8.15 节获取更多信息以及使用这些函数的示例。

表 9.49. 数组函数

函数	array_append(anyarray, anyelement)
返回类型	anyarray
描述	向数组末尾追加一个元素
示例	array_append(ARRAY[1,2], 3)
结果	{1,2,3}
函数	array_cat(anyarray, anyarray)
返回类型	anyarray
描述	串接两个数组
示例	array_cat(ARRAY[1,2,3], ARRAY[4,5])
结果	{1,2,3,4,5}
函数	array_ndims(anyarray)
返回类型	int
描述	返回数组的维数
示例	array_ndims(ARRAY[[1,2,3], [4,5,6]])

结果	2
函数	array_dims(anyarray)
返回类型	text
描述	返回数组维度的文本表示
示例	array_dims(ARRAY[[1,2,3], [4,5,6]])
结果	[1:2][1:3]
函数	array_fill(anyelement, int[] [, int[]])
返回类型	anyarray
描述	返回按给定值和维度初始化的数组，可选地使用不为 1 的下界
示例	array_fill(7, ARRAY[3], ARRAY[2])
结果	[2:4]={7,7,7}
函数	array_length(anyarray, int)
返回类型	int
描述	返回数组指定维度的长度
示例	array_length(array[1,2,3], 1)
结果	3
函数	array_lower(anyarray, int)
返回类型	int
描述	返回数组指定维度的下界
示例	array_lower('[0:2]={1,2,3}'::int[], 1)
结果	0
函数	array_position(anyarray, anyelement [, int])
返回类型	int
描述	返回第二个参数在数组中第一次出现的下标，从第三个参数指定的元素或第一个元素开始搜索（数组必须是一维的）
示例	array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon')
结果	2
函数	array_positions(anyarray, anyelement)
返回类型	int[]
描述	返回第二个参数在第一个参数所给数组中所有出现位置的下标数组（数组必须是一维的）
示例	array_positions(ARRAY['A','A','B','A'], 'A')
结果	{1,2,4}
函数	array_prepend(anyelement, anyarray)
返回类型	anyarray

描述	向数组开头添加一个元素
示例	`array_prepend(1, ARRAY[2,3])`
结果	`{1,2,3}`
函数	`array_remove(anyarray, anyelement)`
返回类型	`anyarray`
描述	从数组中移除所有等于给定值的元素（数组必须是一维的）
示例	`array_remove(ARRAY[1,2,3,2], 2)`
结果	`{1,3}`
函数	`array_replace(anyarray, anyelement, anyelement)`
返回类型	`anyarray`
描述	将数组中每个等于给定值的元素替换为新值
示例	`array_replace(ARRAY[1,2,5,4], 5, 3)`
结果	`{1,2,3,4}`
函数	`array_to_string(anyarray, text [, text])`
返回类型	`text`
描述	使用给定分隔符和可选的 null 替代字符串串接数组元素
示例	`array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')`
结果	`1,2,3,*,5`
函数	`array_upper(anyarray, int)`
返回类型	`int`
描述	返回数组指定维度的上界
示例	`array_upper(ARRAY[1,8,3,7], 1)`
结果	`4`
函数	`cardinality(anyarray)`
返回类型	`int`
描述	返回数组的元素总数；数组为空时返回 0
示例	`cardinality(ARRAY[[1,2],[3,4]])`
结果	`4`
函数	`string_to_array(text, text [, text])`
返回类型	`text[]`
描述	使用给定分隔符和可选的 null 替代字符串将字符串拆分为数组元素
示例	`string_to_array('xx~^~yy~^~zz', '~^~', 'yy')`
结果	`{xx,NULL,zz}`

函数	`unnest(anyarray)`
返回类型	`setof anyelement`
描述	将数组展开为一组行
示例	`unnest(ARRAY[1,2])`
结果	1 2 (2 rows)
函数	`unnest(anyarray, anyarray [, ...])`
返回类型	`setof anyelement, anyelement [, ...]`
描述	将多个数组（类型可以不同）展开为一组行。仅允许在 FROM 子句中使用；参见第 7.2.1.4 节
示例	`unnest(ARRAY[1,2],ARRAY['foo','bar','baz'])`
结果	1 foo 2 bar NULL baz (3 rows)

在 `array_position` 和 `array_positions` 中，使用 `IS NOT DISTINCT FROM` 语义将每个数组元素与要搜索的值进行比较。

在 `array_position` 中，如果找不到该值，则返回 `NULL`。

在 `array_positions` 中，只有数组为 `NULL` 时才返回 `NULL`；如果数组中找不到该值，则返回空数组。

在 `string_to_array` 中，如果分隔符参数为 NULL，输入字符串中的每个字符都会成为结果数组中的独立元素。如果分隔符为空字符串，则将整个输入字符串作为单元素数组返回。否则，在分隔符字符串的每次出现处拆分输入字符串。

在 `string_to_array` 中，如果省略 null 替代字符串参数，或该参数为 NULL，则不会将任何输入子串替换为 NULL。在 `array_to_string` 中，如果省略 null 替代字符串参数，或该参数为 NULL，则直接跳过数组中的所有 null 元素，不在输出字符串中表示它们。

> **注意**
>
> `string_to_array` 的行为与 PostgreSQL 9.1 之前的版本有两点不同。首先，当输入字符串长度为零时，它返回空数组（零元素），而不是 NULL。其次，如果分隔符字符串为 NULL，该函数会将输入拆分为单独的字符，而不是像以前那样返回 NULL。

也可参见第 9.20 节了解用于数组的聚合函数 `array_agg`。

9.19. 范围函数和操作符

范围类型的概述可参见第 8.17 节。

表 9.50 列出了可用于范围类型的操作符。

表 9.50. 范围操作符

操作符	描述	示例	结果
=	等于	int4range(1,5) = '[1,4]'::int4range	t
<>	不等于	numrange(1.1,2.2) <> numrange(1.1,2.3)	t
<	小于	int4range(1,10) < int4range(2,3)	t
>	大于	int4range(1,10) > int4range(1,5)	t
<=	小于等于	numrange(1.1,2.2) <= numrange(1.1,2.2)	t
>=	大于等于	numrange(1.1,2.2) >= numrange(1.1,2.0)	t
@>	包含范围	int4range(2,4) @> int4range(2,3)	t
@>	包含元素	'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp	t
<@	范围被包含于	int4range(2,4) <@ int4range(1,7)	t
<@	元素被包含于	42 <@ int4range(1,7)	f
&&	重叠（有公共点）	int8range(3,7) && int8range(4,12)	t
<<	严格位于左侧	int8range(1,10) << int8range(100,110)	t
>>	严格位于右侧	int8range(50,60) >> int8range(20,30)	t
&<	不超出其右边界	int8range(1,20) &< int8range(18,20)	t
&>	不超出其左边界	int8range(7,20) &> int8range(5,10)	t
-|-	与之相邻	numrange(1.1,2.2) -|- numrange(2.2,3.3)	t
+	并集	numrange(5,15) + numrange(10,20)	[5,20)
*	交集	int8range(5,15) * int8range(10,20)	[10,15)
-	差集	int8range(5,15) - int8range(10,20)	[5,10)

简单比较操作符 <、>、<= 和 >= 首先比较下界，只有下界相等时才比较上界。这些比较对范围通常用处不大，但提供这些操作符可以在范围上建立 B-树索引。

涉及空范围时，左侧、右侧和相邻操作符总是返回假；也就是说，空范围不被视为位于任何其他范围之前或之后。

如果结果范围需要包含两个不相交的子范围，并集和差集操作符会失败，因为这样的范围无法表示。

表 9.51 列出了可用于范围类型的函数。

表 9.51. 范围函数

函数	`lower(anyrange)`
返回类型	范围的元素类型
描述	范围的下界
示例	`lower(numrange(1.1,2.2))`
结果	`1.1`
函数	`upper(anyrange)`
返回类型	范围的元素类型
描述	范围的上界
示例	`upper(numrange(1.1,2.2))`
结果	`2.2`
函数	`isempty(anyrange)`
返回类型	`boolean`
描述	范围是否为空？
示例	`isempty(numrange(1.1,2.2))`
结果	`false`
函数	`lower_inc(anyrange)`
返回类型	`boolean`
描述	是否包含下界？
示例	`lower_inc(numrange(1.1,2.2))`
结果	`true`
函数	`upper_inc(anyrange)`
返回类型	`boolean`
描述	是否包含上界？
示例	`upper_inc(numrange(1.1,2.2))`
结果	`false`

函数	`lower_inf(anyrange)`
返回类型	`boolean`
描述	下界是否无穷?
示例	`lower_inf('(,)'::daterange)`
结果	`true`
函数	`upper_inf(anyrange)`
返回类型	`boolean`
描述	上界是否无穷?
示例	`upper_inf('(,)'::daterange)`
结果	`true`
函数	`range_merge(anyrange, anyrange)`
返回类型	`anyrange`
描述	包含两个给定范围的最小范围
示例	`range_merge('[1,2)'::int4range, '[3,4)'::int4range)`
结果	`[1,4)`

如果范围为空，或请求的边界是无限的，`lower` 和 `upper` 函数返回 null。对于空范围，`lower_inc`、`upper_inc`、`lower_inf` 和 `upper_inf` 函数都返回假。

9.20. 聚合函数

聚合函数根据一组输入值计算出单个结果。内置的通用聚合函数列于表 9.52，统计聚合函数列于表 9.53。内置的组内有序集聚合函数列于表 9.54，内置的组内假想集聚合函数列于表 9.55。与聚合函数密切相关的分组操作列于表 9.56。聚合函数的特殊语法说明见第 4.2.7 节。更多入门信息请参见第 2.7 节。

表 9.52. 通用聚合函数

函数	`array_agg(`*`expression`*`)`
参数类型	任意非数组类型
返回类型	参数类型的数组
部分模式	否
描述	将输入值（包括 null）串接为数组
函数	`array_agg(`*`expression`*`)`
参数类型	任意数组类型
返回类型	与参数数据类型相同
部分模式	否
描述	将输入数组串接为维数多一维的数组（所有输入的维数必须相同，且不能是空数组或 null）

函数	avg(*expression*)
参数类型	smallint、int、bigint、real、double precision、numeric 或 interval
返回类型	整数类型参数返回 numeric，浮点参数返回 double precision，其他情况与参数数据类型相同
部分模式	是
描述	所有非空输入值的平均值（算术平均值）
函数	bit_and(*expression*)
参数类型	smallint、int、bigint 或 bit
返回类型	与参数数据类型相同
部分模式	是
描述	所有非空输入值的按位与；没有非空输入时为 null
函数	bit_or(*expression*)
参数类型	smallint、int、bigint 或 bit
返回类型	与参数数据类型相同
部分模式	是
描述	所有非空输入值的按位或；没有非空输入时为 null
函数	bool_and(*expression*)
参数类型	bool
返回类型	bool
部分模式	是
描述	所有输入值都为真时返回真，否则返回假
函数	bool_or(*expression*)
参数类型	bool
返回类型	bool
部分模式	是
描述	至少一个输入值为真时返回真，否则返回假
函数	count(*)
参数类型	
返回类型	bigint
部分模式	是
描述	输入行数
函数	count(*expression*)
参数类型	任意
返回类型	bigint
部分模式	是

描述	*expression* 的值不为 null 的输入行数
函数	every(*expression*)
参数类型	bool
返回类型	bool
部分模式	是
描述	等价于 bool_and
函数	json_agg(*expression*)
参数类型	any
返回类型	json
部分模式	否
描述	将值（包括 null）聚合为 JSON 数组
函数	jsonb_agg(*expression*)
参数类型	any
返回类型	jsonb
部分模式	否
描述	将值（包括 null）聚合为 JSON 数组
函数	json_object_agg(*name*, *value*)
参数类型	(any, any)
返回类型	json
部分模式	否
描述	将名称/值对聚合为 JSON 对象；值可以为 null，名称不能为 null
函数	jsonb_object_agg(*name*, *value*)
参数类型	(any, any)
返回类型	jsonb
部分模式	否
描述	将名称/值对聚合为 JSON 对象；值可以为 null，名称不能为 null
函数	max(*expression*)
参数类型	任意数值、字符串、日期/时间、网络或枚举类型，或这些类型的数组
返回类型	与参数类型相同
部分模式	是
描述	所有非空输入值中 *expression* 的最大值
函数	min(*expression*)
参数类型	任意数值、字符串、日期/时间、网络或枚举类型，或这些类型的数组
返回类型	与参数类型相同
部分模式	是

描述	所有非空输入值中 *expression* 的最小值
函数	string_agg(*expression*, *delimiter*)
参数类型	(text, text) 或 (bytea, bytea)
返回类型	与参数类型相同
部分模式	否
描述	将非空输入值串接为字符串，以分隔符分隔
函数	sum(*expression*)
参数类型	smallint、int、bigint、real、double precision、numeric、interval 或 money
返回类型	smallint 或 int 参数返回 bigint，bigint 参数返回 numeric，其他情况与参数数据类型相同
部分模式	是
描述	对所有非空输入值的 *expression* 求和
函数	xmlagg(*expression*)
参数类型	xml
返回类型	xml
部分模式	否
描述	串接非空 XML 值（另见第 9.14.1.7 节）

需要注意，除了count 之外，这些函数在没有选中任何行时都会返回空值。特别地，sum 在没有输入行时返回空值，而不是预期中的零；array_agg 在没有输入行时返回空值，而不是空数组。必要时，可以用coalesce 函数把空值替换成零或空数组。

支持部分模式的聚合函数可以参与并行聚合等多种优化。

注意

布尔聚合 bool_and 和 bool_or 对应于标准 SQL 聚合 every 和 any 或 some。至于 any 和 some，标准语法似乎存在歧义：

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

此处的 ANY 既可以被视为引入一个子查询，也可以在该子查询返回一行布尔值时被视为聚合函数。因此，不能将标准名称用于这些聚合。

> **注意**
>
> 习惯于其他 SQL 数据库管理系统的用户，可能会对count 聚合用于整个表时的性能感到失望。如下查询：
>
> ```
> SELECT count(*) FROM sometable;
> ```
>
> 所需的工作量与表大小成正比：PostgreSQL 必须扫描整个表，或者完整扫描一个包含表中所有行的索引。

聚合函数 array_agg、json_agg、jsonb_agg、json_object_agg、jsonb_object_agg、string_agg 和 xmlagg，以及类似的用户定义聚合函数，其结果值会随输入值的顺序发生实质性变化。默认情况下，输入顺序未指定，但可以在聚合调用中写入 ORDER BY 子句来控制，如第 4.2.7 节所示。也可以用已排序的子查询提供输入值，这通常也能奏效。例如：

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

需要注意，如果外层查询包含连接等额外处理，这种方法可能失效，因为子查询的输出可能在计算聚合之前被重新排序。

表 9.53 列出了统计分析常用的聚合函数。（将它们单独列出只是为了避免让更常用的聚合函数列表过于拥挤。）说明中提到的 *N* 表示所有输入表达式均非空的输入行数。在所有情况下，如果计算没有意义，例如 *N* 为零，都会返回空值。

表 9.53. 用于统计的聚合函数

函数	corr(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	相关系数
函数	covar_pop(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	总体协方差
函数	covar_samp(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	样本协方差

函数	regr_avgx(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	自变量的平均值（sum(*X*)/*N*）
函数	regr_avgy(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	因变量的平均值（sum(*Y*)/*N*）
函数	regr_count(*Y*, *X*)
参数类型	double precision
返回类型	bigint
部分模式	是
描述	两个表达式都非空的输入行数
函数	regr_intercept(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	由（*X*, *Y*）数值对确定的最小二乘拟合线性方程的 y 轴截距
函数	regr_r2(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	相关系数的平方
函数	regr_slope(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	由（*X*, *Y*）数值对确定的最小二乘拟合线性方程的斜率
函数	regr_sxx(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	sum(*X*^2) - sum(*X*)^2/*N*（自变量的“平方和”）

函数	regr_sxy(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	sum(*X***Y*) - sum(*X*) * sum(*Y*)/*N*（自变量与因变量的“乘积和”）
函数	regr_syy(*Y*, *X*)
参数类型	double precision
返回类型	double precision
部分模式	是
描述	sum(*Y*^2) - sum(*Y*)^2/*N*（因变量的“平方和”）
函数	stddev(*expression*)
参数类型	smallint、int、bigint、real、double precision 或 numeric
返回类型	浮点参数返回 double precision，其他情况返回 numeric
部分模式	是
描述	stddev_samp 的历史别名
函数	stddev_pop(*expression*)
参数类型	smallint、int、bigint、real、double precision 或 numeric
返回类型	浮点参数返回 double precision，其他情况返回 numeric
部分模式	是
描述	输入值的总体标准差
函数	stddev_samp(*expression*)
参数类型	smallint、int、bigint、real、double precision 或 numeric
返回类型	浮点参数返回 double precision，其他情况返回 numeric
部分模式	是
描述	输入值的样本标准差
函数	variance(*expression*)
参数类型	smallint、int、bigint、real、double precision 或 numeric
返回类型	浮点参数返回 double precision，其他情况返回 numeric
部分模式	是
描述	var_samp 的历史别名
函数	var_pop(*expression*)
参数类型	smallint、int、bigint、real、double precision 或 numeric
返回类型	浮点参数返回 double precision，其他情况返回 numeric
部分模式	是
描述	输入值的总体方差（总体标准差的平方）

函数	var_samp(*expression*)
参数类型	smallint、int、bigint、real、double precision 或 numeric
返回类型	浮点参数返回 double precision，其他情况返回 numeric
部分模式	是
描述	输入值的样本方差（样本标准差的平方）

表 9.54 列出了一些使用有序集聚合语法的聚合函数。这些函数有时被称为“逆分布”函数。

表 9.54. 有序集聚合函数

函数	直接参数类型	聚合参数类型	返回类型	部分模式	描述
mode() WITHIN GROUP (ORDER BY *sort_expression*)		任意可排序类型	与排序表达式相同	否	返回出现次数最多的输入值（若有多个结果同样频繁，则任意选择第一个）
percentile_cont(*fraction*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision	double precision 或 interval	与排序表达式相同	否	连续百分位点：返回排序中与指定比例对应的值，必要时在相邻输入项之间插值
percentile_cont(*fractions*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision[]	double precision 或 interval	排序表达式类型的数组	否	多个连续百分位点：返回形状与 *fractions* 参数一致的结果数组，将每个非空元素替换为与该百分位点对应的值
percentile_disc(*fraction*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision	任意可排序类型	与排序表达式相同	否	离散百分位数：返回排序位置等于或超过指定比例的第一个输入值
percentile_disc(*fractions*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision[]	任意可排序类型	排序表达式类型的数组	否	多个离散百分位数：返回形状与 *fractions* 参数一致的结果数组，将每个非空元素替换为与该百分位数对应的输入值

表 9.54 中列出的所有聚合函数都忽略其排序输入中的空值。对于接受 *fraction* 参数的函数，该比例值必须在 0 和 1 之间，否则会报错。但 null 比例值只会产生 null 结果。

表 9.55 中列出的每个聚合函数都与第 9.21 节中定义的同名窗口函数相关联。对于每个函数，如果将由 *args* 构造的“假想”行加入由 *sorted_args* 计算出的已排序行组，聚合结果就是相应窗口函数会为该行返回的值。

表 9.55. 假想集聚合函数

函数	直接参数类型	聚合参数类型	返回类型	部分模式	描述
rank(*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	bigint	否	假设行的排名，重复行会造成排名空缺
dense_rank(*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	bigint	否	假设行的排名，没有空缺
percent_rank(*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	double precision	否	假设行的相对排名，范围为 0 到 1
cume_dist(*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	double precision	否	假设行的相对排名，范围为 1/*N* 到 1

对于每个假想集聚合函数，*args* 中给出的直接参数列表必须与 *sorted_args* 中给出的聚合参数在数量和类型上匹配。与大多数内置聚合不同，这些聚合不是严格的，也就是说，它们不会丢弃包含空值的输入行。空值按 ORDER BY 子句指定的规则排序。

表 9.56. 分组操作

函数	返回类型	描述
GROUPING(*args...*)	integer	表示哪些参数未包含在当前分组集中的整数位掩码

分组操作与分组集配合使用（参见第 7.2.4 节），以区分结果行。传给GROUPING 操作的参数不会实际求值，但它们必须与同一查询层级的GROUP BY 子句中的表达式完全匹配。位的分配方式是将最右侧的参数对应到最低有效位；如果相应表达式包含在生成结果行的分组集的分组条件中，该位为 0，否则为 1。例如：

```
=> SELECT * FROM items_sold;
 make  | model | sales
-------+-------+-------
 Foo   | GT    |  10
 Foo   | Tour  |  20
 Bar   | City  |  15
 Bar   | Sport |  5
(4 rows)

=> SELECT make, model, GROUPING(make,model), sum(sales) FROM items_sold GROUP BY ROLLUP(make,
model);
 make  | model | grouping | sum
-------+-------+----------+-----
 Foo   | GT    |        0 | 10
 Foo   | Tour  |        0 | 20
 Bar   | City  |        0 | 15
 Bar   | Sport |        0 | 5
 Foo   |       |        1 | 30
 Bar   |       |        1 | 20
       |       |        3 | 50
(7 rows)
```

9.21. 窗口函数

窗口函数提供了对与当前查询行相关的一组行执行计算的能力。该特性的介绍请参见第 3.5 节，语法细节请参见第 4.2.8 节。

内置的窗口函数列在表 9.57 中。注意，这些函数必须使用窗口函数语法调用，也就是说，需要一个OVER 子句。

除了这些函数外，任何内置或用户定义的通用或统计聚合函数（即不属于有序集或假想集聚合的函数）都可以用作窗口函数；内置聚合函数列表见第 9.20 节。聚合函数只有在调用后跟随 OVER 子句时才作为窗口函数；否则作为非窗口聚合，为整个集合返回一行。

表 9.57. 通用窗口函数

函数	返回类型	描述
row_number()	bigint	当前行在其分区内的编号，从 1 开始计数
rank()	bigint	当前行的排名，允许空缺；等于其第一个同等行的 row_number
dense_rank()	bigint	当前行的排名，没有空缺；此函数对同等行组计数
percent_rank()	double precision	当前行的相对排名：（rank - 1）/（分区总行数 - 1）
cume_dist()	double precision	累积分布：（分区中位于当前行之前或与之同等的行数）/ 分区总行数
ntile(*num_buckets* integer)	integer	范围为 1 到参数值的整数，将分区尽可能均等地划分

函数	返回类型	描述
lag(*value* anyelement [, *offset* integer [, *default* anyelement]])	与 *value* 相同的类型	返回在分区内当前行之前 *offset* 行处计算的 *value*；如果没有这样的行，则返回 *default*（其类型必须与 *value* 相同）。*offset* 和 *default* 都针对当前行求值。如果省略，*offset* 默认为 1，*default* 默认为 null
lead(*value* anyelement [, *offset* integer [, *default* anyelement]])	与 *value* 相同的类型	返回在分区内当前行之后 *offset* 行处计算的 *value*；如果没有这样的行，则返回 *default*（其类型必须与 *value* 相同）。*offset* 和 *default* 都针对当前行求值。如果省略，*offset* 默认为 1，*default* 默认为 null
first_value(*value* any)	与 *value* 相同的类型	返回在窗口帧第一行处计算的 *value*
last_value(*value* any)	与 *value* 相同的类型	返回在窗口帧最后一行处计算的 *value*
nth_value(*value* any, *nth* integer)	与 *value* 相同的类型	返回在窗口帧第 *nth* 行处计算的 *value*（从 1 开始计数）；没有这样的行时返回 null

在表 9.57 中列出的所有函数都依赖于相关窗口定义的 ORDER BY 子句指定的排序顺序。仅考虑 ORDER BY 列时不能区分的行被称为是同等行。定义的这四个排名函数（包括 cume_dist），对于同等行组的所有行的答案相同。

注意 first_value、last_value 和 nth_value 只考虑“窗口帧”内的行，它默认情况下包含从分区的开始行直到当前行的最后一个同等行。这对last_value 可能不会给出有用的结果，有时对nth_value 也一样。你可以通过向 OVER 子句增加一个合适的帧声明（RANGE、ROWS 或 GROUPS）来重定义帧。关于帧声明的更多信息请参考第 4.2.8 节。

当一个聚合函数被用作窗口函数时，它将在当前行的窗口帧内的行上聚合。一个使用ORDER BY 和默认窗口帧定义的聚合产生一种“累计求和”类型的行为，这可能是或者不是想要的结果。为了获取在整个分区上的聚合，省略ORDER BY 或者使用 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING。其它窗口帧声明可以用来获得其它的效果。

> **注意**
>
> SQL 标准为 lead、lag、first_value、last_value 和 nth_value 定义了一个 RESPECT NULLS 或 IGNORE NULLS 选项。这在 PostgreSQL 中没有实现：行为总是与标准的默认相同，即RESPECT NULLS。同样，标准中用于nth_value 的 FROM FIRST 或 FROM LAST 选项没有实现：只支持默认的 FROM FIRST 行为（你可以通过反转 ORDER BY 的排序达到 FROM LAST 的结果）。

cume_dist 计算分区中小于或等于当前行及其同等行的行所占的比例；percent_rank 则假设当前行不存在于分区中，计算分区中小于当前行的行所占的比例。

9.22. 子查询表达式

本节描述 PostgreSQL 中可用的 SQL 兼容的子查询表达式。本节介绍的所有表达式形式都返回布尔值（真/假）结果。

9.22.1. EXISTS

```
EXISTS (subquery)
```

EXISTS 的参数是一个任意的 SELECT 语句，或者说子查询。系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，那么EXISTS 的结果就为“真”；如果子查询没有返回行，那么EXISTS 的结果是“假”。

子查询可以引用外层查询的变量，这些变量在该子查询的任何一次计算中都起常量的作用。

这个子查询通常只是运行到能判断它是否可以返回至少一行为止，而不是等到全部结束。在这里写任何有副作用的子查询都是不明智的（例如调用序列函数）；这些副作用是否发生是很难判断的。

因为结果只取决于是否会返回行，而不取决于这些行的内容，所以这个子查询的输出列表通常是无关紧要的。一个常用的编码习惯是用EXISTS(SELECT 1 WHERE ...) 的形式写所有的EXISTS 测试。不过这条规则有例外，例如那些使用INTERSECT 的子查询。

下面这个简单的示例类似在 col2 上的一次内连接，但是它为每个 tab1 的行最多生成一行输出，即使存在多个匹配tab2 的行也如此：

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.22.2. IN

```
expression IN (subquery)
```

右手边是一个圆括号括起来的子查询，它必须恰好返回一列。左手边表达式会被求值，并与子查询结果逐行比较。如果找到任意相等的子查询行，那么IN 的结果就是“真”。如果没有找到相等行，那么结果就是“假”（包括子查询不返回任何行的情况）。

请注意，如果左侧表达式得到空值，或者右侧没有相等的值且至少有一行得到空值，则IN 结构的结果将是空值，而不是假。这符合 SQL 对空值布尔组合的一般规则。

和 EXISTS 一样，假定子查询一定会完整运行并不明智。

```
row_constructor IN (subquery)
```

这种形式的 IN 左手边是一个行构造器，如第 4.2.13 节中所述。右手边是一个圆括号括起来的子查询，它必须返回与左手边行中表达式数量完全相同的列数。左手边表达式会被求值，并与子查询结果逐行比较。如果找到任意相等的子查询行，则IN 的结果为“真”。如果没有找到相等行，那么结果为“假”（包括子查询不返回任何行的情况）。

通常，表达式或者子查询行里的空值是按照 SQL 布尔表达式的一般规则进行组合的。如果两个行对应的成员都非空并且相等，那么认为这两行相等；如果任意对应成员为非空且不等，那么这

两行不等；否则这样的行比较的结果是未知（空值）。如果所有行的结果要么是不等，要么是空值，并且至少有一个空值，那么IN 的结果是空值。

9.22.3. NOT IN

expression NOT IN (*subquery*)

右手边是一个圆括号括起来的子查询，它必须恰好返回一列。左手边表达式会被求值，并与子查询结果逐行比较。如果只找到不相等的子查询行（包括子查询不返回任何行的情况），那么NOT IN 的结果是“真”。如果找到任何相等行，则结果为“假”。

请注意，如果左侧表达式得到空值，或者右侧没有相等的值且至少有一行得到空值，则NOT IN 结构的结果将是空值，而不是真。这符合 SQL 对空值布尔组合的一般规则。

和 EXISTS 一样，假定子查询一定会完整运行并不明智。

row_constructor NOT IN (*subquery*)

这种形式的 NOT IN 左侧是一个行构造器，如第 4.2.13 节中所述。右侧是一个用圆括号括起来的子查询，必须返回与左侧行中表达式数量完全相同的列数。对左侧表达式求值后，将其按行与子查询结果的每一行比较。如果只找到不相等的子查询行（包括子查询不返回任何行的情况），则NOT IN 的结果为“真”。如果找到任何相等行，则结果为“假”。

通常，表达式或者子查询行里的空值是按照 SQL 布尔表达式的一般规则进行组合的。如果两个行对应的成员都非空并且相等，那么认为这两行相等；如果任意对应成员为非空且不等，那么这两行不等；否则这样的行比较的结果是未知（空值）。如果所有行的结果要么是不等，要么是空值，并且至少有一个空值，那么NOT IN 的结果是空值。

9.22.4. ANY/SOME

expression operator ANY (*subquery*)
expression operator SOME (*subquery*)

这种形式的右侧是一个用括号括起来的子查询，它必须恰好返回一列。左侧表达式会被求值，并使用给定的 *operator* 与子查询结果的每一行进行比较。该操作符必须产生布尔结果。如果得到任何真值结果，那么ANY 的结果就是“真”。如果没有找到真值结果，那么结果是“假”（包括子查询没有返回任何行的情况）。

SOME 是 ANY 的同义词。IN 等价于= ANY。

请注意，如果没有任何比较返回真，并且至少有一个右侧行的操作符结果为空值，则ANY 结构的结果将是空值，而不是假。这符合 SQL 对空值布尔组合的一般规则。

和 EXISTS 一样，假定子查询一定会完整运行并不明智。

row_constructor operator ANY (*subquery*)
row_constructor operator SOME (*subquery*)

这种形式的 ANY 左侧是一个行构造器，如第 4.2.13 节所述。右侧是一个用括号括起来的子查询，它必须返回与左侧行中表达式数量完全相同的列数。左侧表达式会被求值，并使用给定的 *operator* 与子查询结果的每一行逐行比较。如果比较对任何子查询行返回真，则ANY 的结

果为“真”。如果比较对每一个子查询行都返回假，则结果为“假”（包括子查询不返回行的情况）。如果比较对任何行都不返回真，并且至少有一次比较返回 NULL，则结果为 NULL。

关于行构造器比较的详细含义请见第 9.23.5 节。

9.22.5. ALL

expression operator ALL (*subquery*)

右侧是一个用圆括号括起来的子查询，必须恰好返回一列。左侧表达式会被求值，并使用给定的 *operator* 与子查询结果的每一行进行比较。该操作符必须产生布尔结果。如果所有行都得到真（包括子查询不返回任何行的情况），则ALL 的结果为“真”。如果得到任何假值结果，则结果为“假”。如果与任何子查询行的比较都不返回假，并且至少有一次比较返回 NULL，则结果为 NULL。

NOT IN 等价于<> ALL。

和 EXISTS 一样，假定子查询一定会完整运行并不明智。

row_constructor operator ALL (*subquery*)

这种形式的 ALL 左侧是一个行构造器，如第 4.2.13 节所述。右侧是一个用括号括起来的子查询，它必须返回与左侧行中表达式一样多的列。左侧表达式会被求值，并使用给定的 *operator* 与子查询结果逐行比较。如果该比较对所有子查询行都返回真，那么ALL 的结果就是“真”（包括子查询没有返回任何行的情况）。如果对任何子查询行的比较返回假，则结果为“假”。如果比较对任何子查询行都不返回假，并且至少有一次比较返回 NULL，则结果为 NULL。

关于行构造器比较的详细含义请见第 9.23.5 节。

9.22.6. 单一行比较

row_constructor operator (*subquery*)

左侧是一个行构造器，如第 4.2.13 节所述。右侧是一个用圆括号括起来的子查询，必须返回与左侧行中表达式数量完全相同的列数。此外，该子查询不能返回超过一行；如果它返回零行，则结果为空值。对左侧求值后，将所得的行与子查询返回的唯一一行进行比较。

关于行构造器比较的详细含义请见第 9.23.5 节。

9.23. 行和数组比较

本节描述几个特殊的结构，用于在值的组之间进行多重比较。这些形式语法上和前面一节的子查询形式相关，但是不涉及子查询。涉及数组子表达式的形式是PostgreSQL 的扩展；其余形式是 SQL 兼容的。本节介绍的所有表达式形式都返回布尔（Boolean）结果（真/假）。

9.23.1. IN

expression IN (*value* [, ...])

右侧是一个用圆括号括起来的表达式列表。如果左侧表达式的结果等于右侧任一表达式的结果，则结果为“真”。这等价于以下写法：

```
expression = value1
OR
expression = value2
OR
...
```

请注意如果左手边表达式得到空值，或者没有相等的右手边值并且至少有一个右手边的表达式得到空值，那么IN 结构的结果将为空值，而不是假。这符合 SQL 处理空值的布尔组合的一般规则。

9.23.2. NOT IN

```
expression NOT IN (value [, ...])
```

右侧是一个用圆括号括起来的表达式列表。如果左侧表达式的结果与右侧所有表达式的结果都不相等，则结果为“真”。这等价于以下写法：

```
expression <> value1
AND
expression <> value2
AND
...
```

请注意如果左手边表达式得到空值，或者没有相等的右手边值并且至少有一个右手边的表达式得到空值，那么NOT IN 结构的结果将为空值，而不是我们可能天真地认为的真值。这符合 SQL 处理空值的布尔组合的一般规则。

> **提示**
>
> x NOT IN y 在所有情况下都等效于 NOT (x IN y)。但是，在处理空值的时候，用 NOT IN 比用 IN 更可能迷惑新手。最好尽可能用正逻辑来表达你的条件。

9.23.3. ANY/SOME（数组）

```
expression operator ANY (array expression)
expression operator SOME (array expression)
```

右侧是一个用括号括起来的表达式，它必须产生一个数组值。左侧表达式会被求值，并使用给定的*operator* 与数组的每个元素进行比较，该操作符必须产生布尔结果。如果得到了任何真值结果，那么ANY 的结果是“真”。如果没有找到真值结果（包括数组有零个元素的情况），那么结果是“假”。

如果数组表达式得到的是 null 数组，那么ANY 的结果将为 null。如果左手边的表达式得到 null，ANY 通常也为 null（尽管非严格比较操作符可能得到不同结果）。另外，如果右手边数组包含任何 null 元素，并且没有得到真值比较结果，那么ANY 的结果将为 null 而不是假（同样假设这里使用的是严格比较操作符）。这符合 SQL 处理 null 值布尔组合的一般规则。

SOME 是 ANY 的同义词。

9.23.4. ALL（数组）

expression operator ALL (*array expression*)

右侧是一个用括号括起来的表达式，它必须产生一个数组值。左侧表达式会被求值，并使用给定的*operator* 与数组的每个元素进行比较，该操作符必须产生布尔结果。如果所有比较都得到真值结果，那么ALL 的结果是“真”（包括数组有零个元素的情况）。如果有任何假值结果，那么结果是“假”。

如果数组表达式得到的是 null 数组，那么ALL 的结果将为 null。如果左手边的表达式得到 null，ALL 通常也为 null（尽管非严格比较操作符可能得到不同结果）。另外，如果右手边数组包含任何 null 元素，并且没有得到假值比较结果，那么ALL 的结果将为 null 而不是真（同样假设这里使用的是严格比较操作符）。这符合 SQL 处理 null 值布尔组合的一般规则。

9.23.5. 行构造器比较

row_constructor operator row_constructor

两侧都是行构造器，如第 4.2.13 节所述。两个行构造器必须具有相同数量的字段。给定的*operator* 应用于每一对对应字段。（由于字段可能具有不同的类型，因此可能为各对字段选择不同的具体操作符。）所选操作符必须是某个 B-树操作符类的成员，或者是 B-树操作符类中=成员的否定操作符。这意味着只有当 *operator* 是=、<>、<、<=、>、>=，或具有与这些操作符类似的语义时，才可以进行行构造器比较。

=和<> 情况略有不同。如果两行的所有对应成员都是非空且相等则这两行被认为相等；如果任何对应成员是非空但是不相等则这两行不相等；否则行比较的结果为未知（空值）。

对于<、<=、> 和>=这几种情况，会从左到右比较各行元素，一旦找到一对不相等或含有 null 的元素就立即停止。如果这对元素中的任意一个为 null，那么行比较的结果就是未知（null）；否则，这对元素的比较结果决定整个行比较的结果。例如，ROW(1,2,NULL) < ROW(1,3,0) 的结果为真，而不是 null，因为第三对元素并不会被考虑。

> **注意**
>
> 在 PostgreSQL 8.2 之前，<、<=、> 和>=这几种情况并不是按照 SQL 规范处理的。像 ROW(a,b) < ROW(c,d) 这样的比较会被实现为 a < c AND b < d，而正确行为应当等价于a < c OR (a = c AND b < d)。

row_constructor IS DISTINCT FROM *row_constructor*

这个结构与<> 行比较相似，但是它对于空值输入不会得到空值。任何空值被认为和任何非空值不相等（有区别），并且任意两个空值被认为相等（无区别）。因此结果将总是为真或为假，永远不会是空值。

row_constructor IS NOT DISTINCT FROM *row_constructor*

这个结构与=行比较相似，但是它对于空值输入不会得到空值。任何空值被认为和任何非空值不相等（有区别），并且任意两个空值被认为相等（无区别）。因此结果将总是为真或为假，永远不会是空值。

9.23.6. 复合类型比较

record operator record

SQL 规范要求在结果依赖于比较两个 NULL 值或者一个 NULL 与一个非 NULL 时行比较返回 NULL。PostgreSQL 只有在比较两个行构造器（如第 9.23.5 节）的结果或者比较一个行构造器与一个子查询的输出时才这样做（如第 9.22 节中所述）。在其他比较两个复合类型值的环境中，两个 NULL 字段值被认为相等，并且一个 NULL 被认为大于一个非 NULL。为了得到复合类型的一致的排序和索引行为，这样做是必要的。

对两侧分别求值后，按行进行比较。当*operator* 是 =、<>、<、<=、> 或者 >=时或者具有与这些类似的语义时，允许复合类型的比较（更准确地说，如果一个操作符是一个 B-树操作符类的成员，或者是一个 B-树操作符类的=成员的否定操作符，它就可以是一个行比较操作符）。上述操作符的默认行为与用于行构造器（见第 9.23.5 节）的 `IS [ NOT ] DISTINCT FROM` 相同。

为了支持包含无默认 B-树操作符类的元素的行匹配，为复合类型比较定义了下列操作符：*=、*<>、*<、*<=、*> 以及 *>=。这些操作符比较两行的内部二进制表示。即使两行用相等操作符的比较为真，两行也可能具有不同的二进制表示。使用这些比较操作符得到的行排序是确定的，但除此之外没有其他意义。这些操作符在内部被用于物化视图并且可能对其他如复制之类的特殊功能有用，但是它们并不打算用在书写查询这类普通用途中。

9.24. 集合返回函数

本节描述那些可能返回多于一行的函数。目前这个类中被使用最广泛的是序列生成函数，如表 9.58 和表 9.59 所述。其他更特殊的集合返回函数在本手册的其他地方描述。组合多个集合返回函数的方法可见第 7.2.1.4 节。

表 9.58. 序列生成函数

函数	参数类型	返回类型	描述
`generate_series(` *start*, *stop*`)`	`int`、`bigint` 或 `numeric`	`setof int`、`setof bigint` 或 `setof numeric`（与参数类型相同）	以 1 为步长，从 *start* 到 *stop* 生成一系列值
`generate_series(` *start*, *stop*, *step*`)`	`int`、`bigint` 或 `numeric`	`setof int`、`setof bigint` 或 `setof numeric`（与参数类型相同）	以 *step* 为步长，从 *start* 到 *stop* 生成一系列值
`generate_series(` *start*, *stop*, *step* `interval)`	`timestamp` 或 `timestamp with time zone`	`setof timestamp` 或 `setof timestamp with time zone`（与参数类型相同）	以 *step* 为步长，从 *start* 到 *stop* 生成一系列值

当 *step* 为正数时，如果*start* 大于 *stop*，则返回零行。反之，当*step* 为负数时，如果 *start* 小于 *stop* 则返回零行。如果输入为 `NULL`，也返回零行。如果*step* 为零，则会报错。下面是一些示例：

```
SELECT * FROM generate_series(2,4);
 generate_series
```

```
-----------------
               2
               3
               4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

SELECT generate_series(1.1, 4, 1.3);
 generate_series
-----------------
             1.1
             2.4
             3.7
(3 rows)

-- 此示例使用日期加整数的操作符
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
   dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

表 9.59. 下标生成函数

函数	返回类型	描述
generate_subscripts(*array anyarray*, *dim int*)	setof int	生成由给定数组下标组成的序列。

函数	返回类型	描述
generate_subscripts(*array anyarray*, *dim int*, *reverse boolean*)	setof int	生成由给定数组下标组成的序列。当 *reverse* 为真时，按逆序返回该序列。

generate_subscripts 是一个便利函数，用于生成给定数组在指定维度上的有效下标集合。对于没有所请求维度的数组或 NULL 数组，返回零行（但数组中的 NULL 元素仍会返回有效下标）。下面是一些示例：

```
-- 基本用法
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
 4
(4 rows)

-- 展示数组、下标和下标对应的值
-- 需要使用子查询
SELECT * FROM arrays;
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
     array     | subscript | value
---------------+-----------+-------
 {-1,-2}       |         1 |    -1
 {-1,-2}       |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)

-- 展开二维数组
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
```

```
       3
       4
(4 rows)
```

当 FROM 子句中的函数后面加上 WITH ORDINALITY 时，一个bigint 列会追加到输出中，其值从 1 开始，对函数输出的每一行递增 1。这种方式对集合返回函数尤其有用，例如unnest()。

```
-- 集合返回函数与 WITH ORDINALITY
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls,n);
       ls         | n
------------------+----
 pg_serial        |  1
 pg_twophase      |  2
 postmaster.opts  |  3
 pg_notify        |  4
 postgresql.conf  |  5
 pg_tblspc        |  6
 logfile          |  7
 base             |  8
 postmaster.pid   |  9
 pg_ident.conf    | 10
 global           | 11
 pg_xact          | 12
 pg_snapshots     | 13
 pg_multixact     | 14
 PG_VERSION       | 15
 pg_wal           | 16
 pg_hba.conf      | 17
 pg_stat_tmp      | 18
 pg_subtrans      | 19
(19 rows)
```

9.25. 系统信息函数

表 9.60 列出了多个用于提取会话和系统信息的函数。

除了本节列出的函数外，还有许多与统计系统相关的函数，这些函数也提供系统信息。有关更多信息，请参见第 28.2.3 节。

表 9.60. 会话信息函数

名称	返回类型	描述
current_catalog	name	当前数据库名称（在 SQL 标准中称为“目录”）
current_database()	name	当前数据库名称
current_query()	text	客户端提交的当前执行查询文本（可能包含多条语句）
current_role	name	等价于 current_user
current_schema[()]	name	当前模式名称
current_schemas(boolean)	name[]	搜索路径中的模式名称，可选地包含隐式模式

名称	返回类型	描述
current_user	name	当前执行上下文的用户名
inet_client_addr()	inet	远端连接地址
inet_client_port()	int	远端连接端口
inet_server_addr()	inet	本地连接地址
inet_server_port()	int	本地连接端口
pg_backend_pid()	int	服务当前会话的服务器进程的进程 ID
pg_blocking_pids(int)	int[]	阻止指定服务器进程获取锁的进程 ID
pg_conf_load_time()	timestamp with time zone	配置加载时间
pg_current_logfile([text])	text	日志收集器当前使用的主要日志文件名，或指定格式的日志文件名
pg_my_temp_schema()	oid	会话临时模式的 OID；不存在时为 0
pg_is_other_temp_schema(oid)	boolean	该模式是否为另一个会话的临时模式?
pg_jit_available()	boolean	JIT 编译器扩展是否可用（参见第 32 章），且jit 配置参数是否设为 on。
pg_listening_channels()	setof text	会话当前监听的通道名称
pg_notification_queue_ usage()	double	异步通知队列当前已占用的比例（0-1）
pg_postmaster_start_time( )	timestamp with time zone	服务器启动时间
pg_safe_snapshot_blocki ng_pids(int)	int[]	阻止指定服务器进程获取安全快照的进程 ID
pg_trigger_depth()	int	PostgreSQL 触发器的当前嵌套层级（如果不是从触发器内部直接或间接调用，则为 0）
session_user	name	会话用户名
user	name	等价于 current_user
version()	text	PostgreSQL 版本信息。机器可读的版本另见 server_version_ num。

注意

current_catalog、current_role、current_schema、current_user、session_user 和 user 在 SQL 中具有特殊语法：调用时不得在后面加圆括号。在 PostgreSQL 中，current_schema 可以选择加圆括号，其他函数则不可以。

session_user 通常是发起当前数据库连接的用户，但超级用户可以用SET SESSION AUTHORIZATION 修改此设置。current_user 是用于权限检查的用户标识，通常等于会话用户，但可以用SET ROLE 更改。在执行具有 SECURITY DEFINER 属性的函数期间，它也会改变。用 Unix 的术语来说，会话用户是“真实用户”，当前用户是“有效用户”。current_role

和 user 是 current_user 的同义词。（SQL 标准区分 current_role 和 current_user，但PostgreSQL 不区分，因为它将用户和角色统一为同一种实体。）

current_schema 返回搜索路径中第一个模式的名称（如果搜索路径为空，则返回空值）。在创建表或其他命名对象时，如果未指定目标模式，就会使用该模式。current_schemas(boolean) 返回当前搜索路径中所有模式名称的数组。布尔选项决定是否在返回的搜索路径中包含 pg_catalog 等隐式包含的系统模式。

> **注意**
>
> 可以在运行时修改搜索路径，命令如下：
>
> ```
> SET search_path TO schema [, schema, ...]
> ```

inet_client_addr 返回当前客户端的 IP 地址，inet_client_port 返回端口号。inet_server_addr 返回服务器接受当前连接所用的 IP 地址，inet_server_port 返回端口号。如果当前连接通过 Unix 域套接字建立，这些函数都返回 NULL。

pg_blocking_pids 返回一个数组，包含阻塞指定进程 ID 对应的服务器进程的会话进程 ID；如果不存在这样的服务器进程，或该进程未被阻塞，则返回空数组。一个服务器进程会在以下情况下阻塞另一个进程：它持有与被阻塞进程请求的锁冲突的锁（硬阻塞）；或者它正在等待一个会与被阻塞进程请求的锁冲突的锁，并且在等待队列中位于被阻塞进程之前（软阻塞）。使用并行查询时，即使实际持锁或等待锁的是子工作进程，结果也始终列出客户端可见的进程 ID（即 pg_backend_pid 的结果）。因此，结果中可能出现重复的 PID。另外，如果持有冲突锁的是一个预备事务，该函数会在结果中用进程 ID 0 表示它。频繁调用此函数可能影响数据库性能，因为它需要短暂地独占访问锁管理器的共享状态。

pg_conf_load_time 返回服务器配置文件最近一次加载的时间，类型为 timestamp with time zone。（如果当时当前会话已经存在，则返回该会话自身重新读取配置文件的时间，因此不同会话中的时间会略有不同。否则，返回 postmaster 进程重新读取配置文件的时间。）

pg_current_logfile 以 text 返回日志收集器当前使用的日志文件路径。路径包括 log_directory 目录和日志文件名。必须启用日志收集，否则返回值为 NULL。当存在多个格式不同的日志文件时，不带参数调用 pg_current_logfile 会按 stderr、csvlog 的顺序查找，返回找到的第一个格式的文件路径。如果没有任何日志文件采用这些格式，则返回 NULL。要请求特定文件格式，可向可选参数传入 text 类型的 csvlog 或 stderr。如果请求的日志格式未配置在 log_destination 中，返回值为 NULL。pg_current_logfile 反映 current_logfiles 文件的内容。

pg_my_temp_schema 返回当前会话的临时模式的 OID；如果没有临时模式（因为尚未创建任何临时表），则返回零。如果给定 OID 是另一个会话的临时模式的 OID，pg_is_other_temp_schema 返回真。（这可以用于从系统目录显示结果中排除其他会话的临时表等场景。）

pg_listening_channels 返回当前会话正在监听的异步通知通道名称集合。pg_notification_queue_usage 返回待处理通知当前占用的空间占通知总可用空间的比例，是一个范围为 0-1 的 double 值。更多信息请参见 LISTEN 和 NOTIFY。

pg_postmaster_start_time 返回服务器启动的时间，类型为 timestamp with time zone。

pg_safe_snapshot_blocking_pids 返回一个数组，包含阻止指定进程 ID 对应的服务器进程获取安全快照的会话进程 ID；如果不存在这样的服务器进程，或该进程未被阻塞，则返回空数组。运行 SERIALIZABLE 事务的会话会阻止 SERIALIZABLE READ ONLY DEFERRABLE 事务获取快照，直到后者确定可以安全地不获取任何谓词锁。关于可序列化事务和可延迟事务的更多信息，请参见第 13.2.3 节。频繁调用此函数可能影响数据库性能，因为它需要短暂访问谓词锁管理器的共享状态。

version 返回一个描述 PostgreSQL 服务器版本的字符串。也可以通过 server_version 获取此信息，或者通过server_version_num 获取机器可读的版本。软件开发者应使用 server_version_num（自 8.2 起提供）或 PQserverVersion，而不是解析文本版本。

表 9.61 列出了允许用户以编程方式查询对象访问权限的函数。关于权限的更多信息，请参见第 5.6 节。

表 9.61. 访问权限查询函数

名称	返回类型	描述
has_any_column_privilege(*user*, *table*, *privilege*)	boolean	用户是否对表的至少一列具有权限
has_any_column_privilege(*table*, *privilege*)	boolean	当前用户是否对表的至少一列具有权限
has_column_privilege(*user*, *table*, *column*, *privilege*)	boolean	用户是否具有列权限
has_column_privilege(*table*, *column*, *privilege*)	boolean	当前用户是否具有列权限
has_database_privilege(*user*, *database*, *privilege*)	boolean	用户是否具有数据库权限
has_database_privilege(*database*, *privilege*)	boolean	当前用户是否具有数据库权限
has_foreign_data_wrapper_privilege(*user*, *fdw*, *privilege*)	boolean	用户是否具有外部数据包装器权限
has_foreign_data_wrapper_privilege(*fdw*, *privilege*)	boolean	当前用户是否具有外部数据包装器权限
has_function_privilege(*user*, *function*, *privilege*)	boolean	用户是否具有函数权限
has_function_privilege(*function*, *privilege*)	boolean	当前用户是否具有函数权限
has_language_privilege(*user*, *language*, *privilege*)	boolean	用户是否具有语言权限

名称	返回类型	描述
has_language_privilege(*language*, *privilege*)	boolean	当前用户是否具有语言权限
has_schema_privilege(*user*, *schema*, *privilege*)	boolean	用户是否具有模式权限
has_schema_privilege(*schema*, *privilege*)	boolean	当前用户是否具有模式权限
has_sequence_privilege (*user*, *sequence*, *privilege*)	boolean	用户是否具有序列权限
has_sequence_privilege(*sequence*, *privilege*)	boolean	当前用户是否具有序列权限
has_server_privilege(*user*, *server*, *privilege*)	boolean	用户是否具有外部服务器权限
has_server_privilege(*server*, *privilege*)	boolean	当前用户是否具有外部服务器权限
has_table_privilege(*user*, *table*, *privilege*)	boolean	用户是否具有表权限
has_table_privilege(*table*, *privilege*)	boolean	当前用户是否具有表权限
has_tablespace_privilege (*user*, *tablespace*, *privilege*)	boolean	用户是否具有表空间权限
has_tablespace_privilege(*tablespace*, *privilege*)	boolean	当前用户是否具有表空间权限
has_type_privilege(*user*, *type*, *privilege*)	boolean	用户是否具有类型权限
has_type_privilege(*type*, *privilege*)	boolean	当前用户是否具有类型权限
pg_has_role(*user*, *role*, *privilege*)	boolean	用户是否具有角色权限
pg_has_role(*role*, *privilege*)	boolean	当前用户是否具有角色权限
row_security_active(*table*)	boolean	该表的行级安全是否对当前用户生效

has_table_privilege 检查用户是否可以以某种方式访问表。用户参数可以是名称、OID（pg_authid.oid），public（表示 PUBLIC 伪角色）；如果省略该参数，则使用 current_user。表可以通过名称或 OID 指定。（因此，has_table_privilege 实际上有六种变体，可以通过参数的数量和类型来区分。）通过名称指定时，如有需要，可以用模式限定名称。所需访问权限类型由文本字符串指定，其值必须为以下值之一：SELECT，INSERT，UPDATE，DELETE，TRUNCATE，REFERENCES，或TRIGGER。还可以添加 WITH GRANT OPTION 到权限类型后，以检查该权限是否附带授予选项。也可以用逗号分隔列出多个权限类型；只要拥有列出的任一权限，结果就为true。（权限字符串不区分大小写，权限名之间允许有额外的空白，但权限名内部不允许。）下面是一些示例：

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');
```

has_sequence_privilege 检查用户是否可以以某种方式访问序列。其参数形式与 has_table_privilege 类似。所需访问权限类型必须为 USAGE、SELECT 或 UPDATE 之一。

has_any_column_privilege 检查用户是否可以以某种方式访问表的至少一列。其参数形式与 has_table_privilege 类似，但所需访问权限类型必须为 SELECT、INSERT、UPDATE 或 REFERENCES 的某种组合。注意，在表级别拥有这些权限中的任意一种，就隐式地对表的每一列拥有该权限，因此对于相同参数，如果 has_table_privilege 返回 true，has_any_column_privilege 也总是返回真。但是，只要至少有一列获得该权限的列级授权，has_any_column_privilege 也会成功。

has_column_privilege 检查用户是否可以以某种方式访问列。其参数形式与 has_table_privilege 类似，但还可以通过列名或属性编号指定列。所需访问权限类型必须为 SELECT、INSERT、UPDATE 或 REFERENCES 的某种组合。注意，在表级别拥有这些权限中的任意一种，就隐式地对表的每一列拥有该权限。

has_database_privilege 检查用户是否可以以某种方式访问数据库。其参数形式与 has_table_privilege 类似。所需访问权限类型必须为 CREATE、CONNECT、TEMPORARY 或 TEMP（等同于 TEMPORARY）的某种组合。

has_function_privilege 检查用户是否可以以某种方式访问函数。其参数形式类似于 has_table_privilege。通过文本字符串而不是 OID 指定函数时，允许的输入与 regprocedure 数据类型相同（参见第 8.19 节）。所需访问权限类型必须为 EXECUTE。例如：

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

has_foreign_data_wrapper_privilege 检查用户是否可以以某种方式访问外部数据包装器。其参数形式与 has_table_privilege 类似。所需访问权限类型必须为 USAGE。

has_language_privilege 检查用户是否可以以某种方式访问过程语言。其参数形式与 has_table_privilege 类似。所需访问权限类型必须为 USAGE。

has_schema_privilege 检查用户是否可以以某种方式访问模式。其参数形式与 has_table_privilege 类似。所需访问权限类型必须为 CREATE 或 USAGE 的某种组合。

has_server_privilege 检查用户是否可以以某种方式访问外部服务器。其参数形式与 has_table_privilege 类似。所需访问权限类型必须为 USAGE。

has_tablespace_privilege 检查用户是否可以以某种方式访问表空间。其参数形式与 has_table_privilege 类似。所需访问权限类型必须为 CREATE。

has_type_privilege 检查用户是否可以以某种方式访问类型。其参数形式与 has_table_privilege 类似。通过文本字符串而不是 OID 指定类型时，允许的输入与 regtype 数据类型相同（参见第 8.19 节）。所需访问权限类型必须为 USAGE。

pg_has_role 检查用户是否可以以某种方式访问角色。其参数形式与 has_table_privilege 类似，但不允许使用 public 作为用户名。所需访问权限类型必须为 MEMBER 或 USAGE 的某种组合。MEMBER 表示直接或间接地属于该角色（即有权执行 SET ROLE），而 USAGE 表示无需执行 SET ROLE 就能立即使用该角色的权限。

row_security_active 检查在 current_user 和当前环境的上下文中，指定表的行级安全是否生效。可以通过名称或 OID 指定表。

表 9.62 列出了判断某个特定对象是否可见的函数，其判断依据是当前模式搜索路径。例如，如果表所在的模式位于搜索路径中，并且在搜索路径的更前面没有同名表，就称该表可见。这等价于说，可以只通过表名引用该表，而不必显式地用模式限定。要列出所有可见表的名称：

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

表 9.62. 模式可见性查询函数

名称	返回类型	描述
pg_collation_is_visible(*collation_oid*)	boolean	排序规则是否在搜索路径中可见
pg_conversion_is_visible(*conversion_oid*)	boolean	转换是否在搜索路径中可见
pg_function_is_visible(*function_oid*)	boolean	函数是否在搜索路径中可见
pg_opclass_is_visible(*opclass_oid*)	boolean	操作符类是否在搜索路径中可见
pg_operator_is_visible(*operator_oid*)	boolean	操作符是否在搜索路径中可见
pg_opfamily_is_visible(*opclass_oid*)	boolean	操作符族是否在搜索路径中可见
pg_statistics_obj_is_visible(*stat_oid*)	boolean	统计信息对象是否在搜索路径中可见
pg_table_is_visible(*table_oid*)	boolean	表是否在搜索路径中可见
pg_ts_config_is_visible(*config_oid*)	boolean	全文检索配置是否在搜索路径中可见
pg_ts_dict_is_visible(*dict_oid*)	boolean	全文检索词典是否在搜索路径中可见
pg_ts_parser_is_visible(*parser_oid*)	boolean	全文检索解析器是否在搜索路径中可见
pg_ts_template_is_visible(*template_oid*)	boolean	全文检索模板是否在搜索路径中可见
pg_type_is_visible(*type_oid*)	boolean	类型（或域）是否在搜索路径中可见

每个函数检查一种数据库对象的可见性。注意，pg_table_is_visible 也可用于视图、物化视图、索引、序列和外部表；pg_function_is_visible 也可用于过程和聚合；pg_type_is_visible 也可用于域。对于函数和操作符，如果搜索路径更前面没有名称和参数数据类型都相同的对象，那么路径中的该对象就是可见的。对于操作符类，会同时考虑名称和关联的索引访问方法。

所有这些函数都需要用对象 OID 标识要检查的对象。如果想按名称测试对象，使用 OID 别名类型会很方便（`regclass`, `regtype`, `regprocedure`, `regoperator`, `regconfig`, 或 `regdictionary`），例如：

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

注意，用这种方式测试不带模式限定的类型名并没有太大意义：只要该名称能够被识别，它就必然可见。

表 9.63 列出从系统目录中提取信息的函数。

表 9.63. 系统目录信息函数

名称	返回类型	描述
format_type(*type_oid*, *typemod*)	text	获取数据类型的 SQL 名称
pg_get_constraintdef(*constraint_oid*)	text	获取约束定义
pg_get_constraintdef(*constraint_oid*, *pretty_bool*)	text	获取约束定义
pg_get_expr(*pg_node_tree*, *relation_oid*)	text	反编译表达式的内部形式，假定其中所有 Var 节点都引用第二个参数指定的关系
pg_get_expr(*pg_node_tree*, *relation_oid*, *pretty_bool*)	text	反编译表达式的内部形式，假定其中所有 Var 节点都引用第二个参数指定的关系
pg_get_functiondef(*func_oid*)	text	获取函数或过程的定义
pg_get_function_arguments(*func_oid*)	text	获取函数或过程定义的参数列表（包含默认值）
pg_get_function_identity_arguments(*func_oid*)	text	获取用于标识函数或过程的参数列表（不含默认值）
pg_get_function_result(*func_oid*)	text	获取函数的 `RETURNS` 子句（对于过程返回 null）
pg_get_indexdef(*index_oid*)	text	获取索引的 `CREATE INDEX` 命令
pg_get_indexdef(*index_oid*, *column_no*, *pretty_bool*)	text	获取索引的 `CREATE INDEX` 命令；当 *column_no* 非零时，仅获取一个索引列的定义
pg_get_keywords()	setof record	获取 SQL 关键字及其类别的列表
pg_get_ruledef(*rule_oid*)	text	获取规则的 `CREATE RULE` 命令
pg_get_ruledef(*rule_oid*, *pretty_bool*)	text	获取规则的 `CREATE RULE` 命令
pg_get_serial_sequence(*table_name*, *column_name*)	text	获取 serial 列或标识列使用的序列名称
pg_get_statisticsobjdef(*statobj_oid*)	text	获取扩展统计信息对象的 `CREATE STATISTICS` 命令

名称	返回类型	描述
pg_get_triggerdef(*trigger_oid*)	text	获取触发器的 CREATE [CONSTRAINT] TRIGGER 命令
pg_get_triggerdef(*trigger_oid*,*pretty_bool*)	text	获取触发器的 CREATE [CONSTRAINT] TRIGGER 命令
pg_get_userbyid(*role_oid*)	name	获取具有给定 OID 的角色名称
pg_get_viewdef(*view_name*)	text	获取视图或物化视图的底层 SELECT 命令（已弃用）
pg_get_viewdef(*view_name*, *pretty_bool*)	text	获取视图或物化视图的底层 SELECT 命令（已弃用）
pg_get_viewdef(*view_oid*)	text	获取视图或物化视图的底层 SELECT 命令
pg_get_viewdef(*view_oid*, *pretty_bool*)	text	获取视图或物化视图的底层 SELECT 命令
pg_get_viewdef(*view_oid*, *wrap_column_int*)	text	获取视图或物化视图的底层 SELECT 命令；包含字段的行按指定列数折行，并隐含启用美化输出
pg_index_column_has_property(*index_oid*, *column_no*, *prop_name*)	boolean	测试索引列是否具有指定属性
pg_index_has_property(*index_oid*, *prop_name*)	boolean	测试索引是否具有指定属性
pg_indexam_has_property(*am_oid*, *prop_name*)	boolean	测试索引访问方法是否具有指定属性
pg_options_to_table(*reloptions*)	setof record	获取存储选项的名称/值对集合
pg_tablespace_databases(*tablespace_oid*)	setof oid	获取在该表空间中具有对象的数据库 OID 集合
pg_tablespace_location(*tablespace_oid*)	text	获取该表空间在文件系统中的路径
pg_typeof(*any*)	regtype	获取任意值的数据类型
collation for (*any*)	text	获取参数的排序规则
to_regclass(*rel_name*)	regclass	获取指定名称关系的 OID
to_regproc(*func_name*)	regproc	获取指定名称函数的 OID
to_regprocedure(*func_name*)	regprocedure	获取指定名称函数的 OID
to_regoper(*operator_name*)	regoper	获取指定名称操作符的 OID
to_regoperator(*operator_name*)	regoperator	获取指定名称操作符的 OID
to_regtype(*type_name*)	regtype	获取指定名称类型的 OID
to_regnamespace(*schema_name*)	regnamespace	获取指定名称模式的 OID
to_regrole(*role_name*)	regrole	获取指定名称角色的 OID

format_type 返回数据类型的 SQL 名称，该类型由其类型 OID 和可能存在的类型修饰符标识。如果不知道具体的类型修饰符，请为类型修饰符参数传入 NULL。

pg_get_keywords 返回一组记录，描述服务器识别的 SQL 关键字。word 列包含关键字。catcode 列包含类别代码：U 表示非保留关键字，C 表示列名，T 表示类型名或函数名，R 表示保留关键字。catdesc 列包含描述该类别的字符串，该字符串可能已经过本地化。

pg_get_constraintdef、pg_get_indexdef、pg_get_ruledef、pg_get_statisticsobjdef 和 pg_get_triggerdef 分别重建约束、索引、规则、扩展统计信息对象或触发器的创建命令。（注意，这是通过反编译重建的，并非命令的原始文本。）pg_get_expr 反编译单个表达式的内部形式，例如列的默认值。这在检查系统目录内容时可能很有用。如果表达式可能包含 Var 节点，请将它们所引用的关系的 OID 指定为第二个参数；如果预计不含 Var 节点，传入零即可。pg_get_viewdef 重建定义视图的 SELECT 查询。这些函数中的大多数有两种变体，其中一种可以选择“美化输出”结果。美化格式更易读，但默认格式更可能被未来版本的 PostgreSQL 以相同方式解释；转储时应避免使用美化输出。向美化输出参数传入 false，所得结果与完全没有该参数的变体相同。

pg_get_functiondef 返回一个函数的完整 CREATE OR REPLACE FUNCTION 语句。pg_get_function_arguments 以 CREATE FUNCTION 中所需的形式返回函数参数列表。pg_get_function_result 类似地返回该函数相应的 RETURNS 子句。pg_get_function_identity_arguments 返回标识函数所需的参数列表，例如以 ALTER FUNCTION 中所需的形式返回。此形式省略默认值。

pg_get_serial_sequence 返回与列关联的序列名称；如果该列没有关联序列，则返回 NULL。如果列是标识列，关联序列就是为标识列在内部创建的序列。对于使用某种 serial 类型（serial，smallserial，bigserial）创建的列，关联序列就是为该 serial 列定义创建的序列。在后一种情况下，可以使用ALTER SEQUENCE OWNED BY 修改或移除这种关联。（该函数也许应该叫作 pg_get_owned_sequence；它当前的名称反映了它通常用于 serial 或 bigserial 列这一事实。）第一个输入参数是可以带有模式名的表名，第二个参数是列名。由于第一个参数可能包含模式和表，它不会被当作双引号括起的标识符处理，因此默认转换为小写；第二个参数只包含列名，会被当作带双引号的标识符处理并保留大小写。函数返回的值采用适于传递给序列函数的格式（参见第 9.16 节）。典型用法是读取标识列或 serial 列的序列当前值，例如：

```
SELECT currval(pg_get_serial_sequence('sometable', 'id'));
```

pg_get_userbyid 根据给定 OID 获取角色名称。

pg_index_column_has_property、pg_index_has_property 和 pg_indexam_has_property 返回指定的索引列、索引或索引访问方法是否具有给定名称的属性。如果属性名未知或不适用于该对象，或者 OID 或列号不能标识有效对象，则返回 NULL。列属性见表 9.64，索引属性见表 9.65，访问方法属性见表 9.66。（注意，扩展提供的访问方法可以为其索引定义额外的属性名。）

表 9.64. 索引列属性

名称	描述
asc	在向前扫描时列是按照升序排列吗？
desc	在向前扫描时列是按照降序排列吗？
nulls_first	在向前扫描时列排序会把空值排在前面吗？
nulls_last	在向前扫描时列排序会把空值排在最后吗？
orderable	列具有已定义的排序顺序吗？
distance_orderable	能否按“距离”操作符的结果有序地扫描该列，例如 ORDER BY col <-> constant。

名称	描述
returnable	列值是否可以通过一次仅索引扫描返回?
search_array	列是否原生支持col = ANY(array) 搜索?
search_nulls	列是否支持IS NULL 和IS NOT NULL 搜索?

表 9.65. 索引属性

名称	描述
clusterable	索引是否可以用于CLUSTER 命令?
index_scan	索引是否支持普通扫描（非位图）?
bitmap_scan	索引是否支持位图扫描?
backward_scan	在扫描中扫描方向能否被更改（为了支持游标上无需物化的FETCH BACKWARD）?

表 9.66. 索引访问方法属性

名称	描述
can_order	访问方法是否支持ASC、DESC 以及CREATE INDEX 中的有关关键词?
can_unique	访问方法是否支持唯一索引?
can_multi_col	访问方法是否支持多列索引?
can_exclude	访问方法是否支持排除约束?
can_include	访问方法是否支持CREATE INDEX 的INCLUDE 子句?

向 pg_options_to_table 传入 pg_class.reloptions 或 pg_attribute.attoptions 时，它返回存储选项名称/值对（*option_name*/*option_value*）的集合。

pg_tablespace_databases 用于检查表空间。它返回在该表空间中存储了对象的数据库 OID 集合。如果此函数返回任何行，则说明该表空间不为空，不能删除。要显示存放在该表空间中的具体对象，需要连接到 pg_tablespace_databases 标识的数据库，并查询它们的 pg_class 系统目录。

pg_typeof 返回所传入值的数据类型的 OID。这有助于排查问题或动态构造 SQL 查询。函数声明的返回类型是 regtype，它是一种 OID 别名类型（参见第 8.19 节）；这意味着它在比较时与 OID 相同，但显示为类型名。例如：

```
SELECT pg_typeof(33);

 pg_typeof
-----------
 integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
--------
      4
(1 row)
```

表达式 `collation for` 返回所传入值的排序规则。例如：

```
SELECT collation for (description) FROM pg_description LIMIT 1;
 pg_collation_for
------------------
 "default"
(1 row)

SELECT collation for ('foo' COLLATE "de_DE");
 pg_collation_for
------------------
 "de_DE"
(1 row)
```

返回值可能带有引号和模式限定。如果不能为参数表达式推导出排序规则，则返回空值。如果参数的数据类型不支持排序规则，则会报错。

`to_regclass`、`to_regproc`、`to_regprocedure`、`to_regoper`、`to_regoperator`、`to_regtype`、`to_regnamespace` 和 `to_regrole` 函数将以 `text` 给出的关系、函数、操作符、类型、模式和角色名称，分别转换为 `regclass`、`regproc`、`regprocedure`、`regoper`、`regoperator`、`regtype`、`regnamespace` 和 `regrole` 类型的对象。这些函数与从文本进行类型转换的区别是：它们不接受数值 OID，而且在找不到名称时（对于 `to_regproc` 和 `to_regoper`，还包括给定名称匹配多个对象时）返回空值，而不会报错。

表 9.67 列出了与数据库对象标识和寻址有关的函数。

表 9.67. 对象信息和寻址函数

名称	返回类型	描述
`pg_describe_object(`*`classid`* `oid, `*`objid`* `oid, `*`objsubid`* `integer)`	`text`	获取数据库对象的描述
`pg_identify_object(`*`classid`* `oid, `*`objid`* `oid, `*`objsubid`* `integer)`	*`type`* `text, `*`schema`* `text, `*`name`* `text, `*`identity`* `text`	获取数据库对象的标识
`pg_identify_object_as_address(`*`classid`* `oid, `*`objid`* `oid, `*`objsubid`* `integer)`	*`type`* `text, `*`object_names`* `text[], `*`object_args`* `text[]`	获取数据库对象地址的外部表示
`pg_get_object_address(`*`type`* `text, `*`object_names`* `text[], `*`object_args`* `text[])`	*`classid`* `oid, `*`objid`* `oid, `*`objsubid`* `integer`	根据外部表示获取数据库对象的地址

`pg_describe_object` 返回数据库对象的文本描述，对象由系统目录 OID、对象 OID 和子对象 ID 指定（例如表中的列号；引用整个对象时，子对象 ID 为零）。该描述供人阅读，并可能根据服务器配置被翻译。这有助于确定存储在 `pg_depend` 系统目录中的对象标识。

`pg_identify_object` 返回一行，其中包含足以唯一标识数据库对象的信息，该对象由系统目录 OID、对象 OID 和子对象 ID 指定。这些信息供机器读取，永远不会被翻译。*`type`* 标识数据库对象的类型；*`schema`* 是对象所属的模式名，对于不属于模式的对象类型则为 `NULL`；如果对象名（以及适用时的模式名）足以唯一标识该对象，*`name`* 就是对象名，并在必要时加引号，否则为

NULL；*identity* 是完整的对象标识，其具体格式取决于对象类型，格式中的每个名称都会根据需要加上模式限定和引号。

pg_identify_object_as_address 返回一行，其中包含足以唯一标识数据库对象的信息，该对象由系统目录 OID、对象 OID 和子对象 ID 指定。返回的信息与当前服务器无关，也就是说，它也能用于标识另一台服务器上名称相同的对象。*type* 标识数据库对象的类型；*object_names* 和 *object_args* 是 text 数组，共同构成对该对象的引用。将这三个值传给 pg_get_object_address 可以获得对象的内部地址。此函数执行 pg_get_object_address 的逆操作。

pg_get_object_address 返回一行，其中包含足以唯一标识数据库对象的信息，该对象由其类型、对象名称数组和参数数组指定。返回的值就是 pg_depend 等系统目录中使用的值，也可以传给 pg_identify_object 或 pg_describe_object 等其他系统函数。*classid* 是包含该对象的系统目录的 OID；*objid* 是对象自身的 OID；*objsubid* 是子对象 ID，没有子对象时为零。此函数执行 pg_identify_object_as_address 的逆操作。

表 9.68 中的函数用于提取此前通过 COMMENT 命令存储的注释。如果找不到与指定参数对应的注释，则返回空值。

表 9.68. 注释信息函数

名称	返回类型	描述
col_description(*table_oid*, *column_number*)	text	获取表列的注释
obj_description(*object_oid*, *catalog_name*)	text	获取数据库对象的注释
obj_description(*object_oid*)	text	获取数据库对象的注释（已弃用）
shobj_description(*object_oid*, *catalog_name*)	text	获取共享数据库对象的注释

col_description 返回表列的注释，列由所属表的 OID 和列号指定。（obj_description 不能用于表列，因为列没有自身的 OID。）

obj_description 的双参数形式返回数据库对象的注释，对象由其 OID 和所在系统目录的名称指定。例如，obj_description(123456,'pg_class') 会获取 OID 为 123456 的表的注释。obj_description 的单参数形式只需要对象 OID。该形式已被弃用，因为无法保证 OID 在不同系统目录之间唯一，因而可能返回错误的注释。

shobj_description 的用法与 obj_description 相同，但用于获取共享对象的注释。有些系统目录由数据库集簇中的所有数据库全局共享，其中对象的描述也全局存储。

表 9.69 中的函数以可导出的形式提供服务器事务信息。这些函数主要用于确定两个快照之间有哪些事务提交。

表 9.69. 事务 ID 和快照

名称	返回类型	描述
txid_current()	bigint	获取当前事务 ID；如果当前事务尚无 ID，则分配一个新的 ID

名称	返回类型	描述
txid_current_if_assigned()	bigint	与 txid_current() 相同，但尚未分配事务 ID 时返回 null，而不是分配新的事务 ID
txid_current_snapshot()	txid_snapshot	获取当前快照
txid_snapshot_xip(*txid_snapshot*)	setof bigint	获取快照中正在进行的事务 ID
txid_snapshot_xmax(*txid_snapshot*)	bigint	获取快照的 xmax
txid_snapshot_xmin(*txid_snapshot*)	bigint	获取快照的 xmin
txid_visible_in_snapshot(*bigint*, *txid_snapshot*)	boolean	该事务 ID 是否在快照中可见？（不要用于子事务 ID）
txid_status(*bigint*)	text	报告给定事务的状态：committed、aborted、in progress；如果事务 ID 太旧，则为 null

内部事务 ID 类型（xid）为 32 位，每经过约 40 亿个事务就会回卷一次。不过，这些函数导出的是用“纪元”计数器扩展的 64 位格式，在一次安装的整个使用期间都不会回卷。这些函数使用的 txid_snapshot 数据类型存储某一时刻的事务 ID 可见性信息。其组成部分见表 9.70。

表 9.70. 快照组件

名称	描述
xmin	仍然活动的最早事务 ID（txid）。所有更早的事务要么已经提交且可见，要么已经回滚而失效。
xmax	第一个尚未分配的 txid。所有大于或等于此值的 txid 在快照时尚未开始，因此不可见。
xip_list	快照时活动的 txid。该列表仅包含 xmin 与 xmax 之间的活动 txid；可能存在大于 xmax 的活动 txid。满足 xmin <= txid < xmax 且不在此列表中的 txid 在快照时已经完成，因此根据其提交状态，要么可见，要么失效。该列表不包含子事务的 txid。

txid_snapshot 的文本表示为 *xmin*:*xmax*:*xip_list*。例如，10:20:10,14,15 表示 xmin=10, xmax=20, xip_list=10, 14, 15。

txid_status(bigint) 报告近期事务的提交状态。如果在 COMMIT 进行过程中应用与数据库服务器断开连接，应用可以用它判断事务是提交了还是中止了。只要事务足够新，系统仍保留其提交状态，就会将状态报告为 in progress、committed 或 aborted。如果事务已足够旧，系统中不再有对它的引用，且提交状态信息已被丢弃，则此函数返回 NULL。注意，预备事务被报告为 in progress；如果需要确定某个 txid 是否为预备事务，应用必须检查 pg_prepared_xacts。

表 9.71 中的函数提供已提交事务的信息，主要是事务何时提交。只有启用 track_commit_timestamp 配置选项后，这些函数才会提供有用的数据，并且只针对启用该选项之后提交的事务。

表 9.71. 已提交事务信息

名称	返回类型	描述
pg_xact_commit_timestamp(*xid*)	timestamp with time zone	获取事务的提交时间戳
pg_last_committed_xact()	*xid* xid, *timestamp* timestamp with time zone	获取最近提交事务的事务 ID 和提交时间戳

表 9.72 中的函数显示在 initdb 期间初始化的信息，例如系统目录版本。它们也显示有关预写式日志和检查点处理的信息。这些信息适用于整个数据库集簇，而非某个特定数据库。它们与 pg_controldata 从相同来源提供大部分相同的信息，但采用更适合 SQL 函数的形式。

表 9.72. 控制数据函数

名称	返回类型	描述
pg_control_checkpoint()	record	返回当前检查点状态的信息。
pg_control_system()	record	返回当前控制文件状态的信息。
pg_control_init()	record	返回集簇初始化状态的信息。
pg_control_recovery()	record	返回恢复状态的信息。

pg_control_checkpoint 返回一条记录，其内容见表 9.73。

表 9.73. pg_control_checkpoint 输出列

列名称	数据类型
checkpoint_lsn	pg_lsn
redo_lsn	pg_lsn
redo_wal_file	text
timeline_id	integer
prev_timeline_id	integer
full_page_writes	boolean
next_xid	text
next_oid	oid
next_multixact_id	xid
next_multi_offset	xid
oldest_xid	xid
oldest_xid_dbid	oid
oldest_active_xid	xid
oldest_multi_xid	xid
oldest_multi_dbid	oid
oldest_commit_ts_xid	xid
newest_commit_ts_xid	xid
checkpoint_time	timestamp with time zone

pg_control_system 返回一条记录，其内容见表 9.74。

表 9.74. pg_control_system 输出列

列名称	数据类型
pg_control_version	integer
catalog_version_no	integer
system_identifier	bigint
pg_control_last_modified	timestamp with time zone

pg_control_init 返回一条记录，其内容见表 9.75。

表 9.75. pg_control_init 输出列

列名称	数据类型
max_data_alignment	integer
database_block_size	integer
blocks_per_segment	integer
wal_block_size	integer
bytes_per_wal_segment	integer
max_identifier_length	integer
max_index_columns	integer
max_toast_chunk_size	integer
large_object_chunk_size	integer
float4_pass_by_value	boolean
float8_pass_by_value	boolean
data_page_checksum_version	integer

pg_control_recovery 返回一条记录，其内容见表 9.76。

表 9.76. pg_control_recovery 输出列

列名称	数据类型
min_recovery_end_lsn	pg_lsn
min_recovery_end_timeline	integer
backup_start_lsn	pg_lsn
backup_end_lsn	pg_lsn
end_of_backup_record_required	boolean

9.26. 系统管理函数

这一节描述的函数被用来控制和监视一个 PostgreSQL 安装。

9.26.1. 配置设定函数

表 9.77 展示了那些可以用于查询以及修改运行时配置参数的函数。

表 9.77. 配置设定函数

名称	返回类型	描述
current_setting(*setting_name* [, *missing_ok*])	text	获取配置项的当前值
set_config(*setting_name*, *new_value*, *is_local*)	text	设置参数并返回新值

函数 current_setting 返回设置 *setting_name* 的当前值。它对应于 SQL 命令 SHOW。例如：

```
SELECT current_setting('datestyle');

 current_setting
-----------------
 ISO, MDY
(1 row)
```

如果不存在名为 *setting_name* 的设置，current_setting 会报错，除非提供了 *missing_ok* 且其值为 true。

set_config 将参数 *setting_name* 设置为 *new_value*。如果 *is_local* 为 true，新值仅适用于当前事务。如果要让新值适用于当前会话，请改用false。该函数对应于 SQL 命令 SET。例如：

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
 off
(1 row)
```

9.26.2. 服务器信号函数

在表 9.78 中展示的函数向其它服务器进程发送控制信号。默认情况下这些函数只能被超级用户使用，但是如果需要，可以利用GRANT 把访问权限授予给其他用户（注明的例外除外）。

表 9.78. 服务器信号函数

名称	返回类型	描述
pg_cancel_backend(*pid* int)	boolean	取消后端的当前查询。如果调用角色是被取消后端所属角色的成员，或者调用角色被授予了 pg_signal_backend，也允许执行此操作；但只有超级用户才能取消超级用户的后端。
pg_reload_conf()	boolean	使服务器进程重新加载配置文件
pg_rotate_logfile()	boolean	轮转服务器日志文件

名称	返回类型	描述
pg_terminate_backend(*pid* int)	boolean	终止后端。如果调用角色是被终止后端所属角色的成员，或者调用角色被授予了 pg_signal_backend，也允许执行此操作；但只有超级用户才能终止超级用户的后端。

这些函数在成功时都返回 true，否则返回 false。

pg_cancel_backend 和 pg_terminate_backend 向由进程 ID 标识的后端进程发送信号（分别是 SIGINT 或 SIGTERM）。一个活动后端的进程 ID 可以从 pg_stat_activity 视图的 pid 列中找到，或者通过在服务器上列出postgres 进程（在 Unix 上使用 ps 或者在 Windows 上使用任务管理器）得到。一个活动后端的角色可以在 pg_stat_activity 视图的 usename 列中找到。

pg_reload_conf 向服务器发送 SIGHUP 信号，使所有服务器进程重新加载配置文件。

pg_rotate_logfile 向日志文件管理器发送信号，使其立即切换到新的输出文件。只有内置日志收集器正在运行时才有效，否则不存在日志文件管理器子进程。

9.26.3. 备份控制函数

表 9.79 中列出的函数有助于进行在线备份。这些函数不能在恢复期间执行（非排他模式的 pg_start_backup、非排他模式的 pg_stop_backup、pg_is_in_backup、pg_backup_start_time 和 pg_wal_lsn_diff 除外）。

表 9.79. 备份控制函数

名称	返回类型	描述
pg_create_restore_point(*name* text)	pg_lsn	创建用于恢复的命名点（默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数）
pg_current_wal_flush_lsn()	pg_lsn	获取当前预写式日志刷盘位置
pg_current_wal_insert_lsn()	pg_lsn	获取当前预写式日志插入位置
pg_current_wal_lsn()	pg_lsn	获取当前预写式日志写入位置
pg_start_backup(*label* text [, *fast* boolean [, *exclusive* boolean]])	pg_lsn	准备执行在线备份（默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数）
pg_stop_backup()	pg_lsn	结束排他在线备份（默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数）
pg_stop_backup(*exclusive* boolean [, *wait_for_archive* boolean])	setof record	结束排他或非排他在线备份（默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数）
pg_is_in_backup()	bool	如果在线排他备份仍在进行，则为真。
pg_backup_start_time()	timestamp with time zone	获取正在进行的在线排他备份的开始时间。

名称	返回类型	描述
pg_switch_wal()	pg_lsn	强制切换到新的预写式日志文件（默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数）
pg_walfile_name(*lsn* pg_lsn)	text	将预写式日志位置转换为文件名
pg_walfile_name_offset(*lsn* pg_lsn)	text, integer	将预写式日志位置转换为文件名和文件内的十进制字节偏移量
pg_wal_lsn_diff(*lsn* pg_lsn, *lsn* pg_lsn)	numeric	计算两个预写式日志位置之差

pg_start_backup 接受任意用户定义的备份标签。（通常是备份转储文件将要保存的名称。）在排他模式下，函数将备份标签文件（backup_label）以及在 pg_tblspc/目录中存在链接时生成的表空间映射文件（tablespace_map）写入数据库集簇的数据目录，执行检查点，然后以文本形式返回备份的起始预写式日志位置。用户可以忽略该返回值，提供它只是因为它可能有用。在非排他模式下，这些文件的内容改由pg_stop_backup 函数返回，并应由调用者写入备份。

```
postgres=# select pg_start_backup('label_goes_here');
 pg_start_backup
-----------------
 0/D4445B8
(1 row)
```

还可以提供第二个可选参数，其类型为boolean。如果该参数为 true，则尽快执行 pg_start_backup。这会强制立即执行检查点，使 I/O 操作量陡增，并降低并发执行的查询的速度。

在排他备份中，pg_stop_backup 删除标签文件，以及 pg_start_backup 创建的 tablespace_map 文件（如果存在）。在非排他备份中，backup_label 和 tablespace_map 的内容在函数结果中返回，应将它们写入备份中的文件，而不是数据目录中的文件。还有一个 boolean 类型的可选第二参数。如果为假，pg_stop_backup 会在备份完成后立即返回，不等待 WAL 归档。这种行为只适用于自行监控 WAL 归档的备份软件；否则，可能缺少使备份保持一致所需的 WAL，导致备份无法使用。如果该参数为真，且归档已启用，pg_stop_backup 会等待 WAL 归档；在备库上，这意味着只有 archive_mode = always 时才会等待。如果主库的写入活动很少，可以在主库上运行 pg_switch_wal，触发立即切换日志段。

在主库上执行时，该函数还会在预写式日志归档区域创建备份历史文件。历史文件包括传给 pg_start_backup 的标签、备份的起止预写式日志位置以及备份的起止时间。返回值是备份的结束预写式日志位置（同样可以忽略）。记录结束位置后，当前预写式日志插入点会自动推进到下一个预写式日志文件，使包含结束位置的预写式日志文件可以立即归档，从而完成备份。

pg_switch_wal 切换到下一个预写式日志文件，使当前文件可以归档（假设正在使用连续归档）。返回值是刚完成的预写式日志文件中的结束预写式日志位置加 1。如果自上次切换预写式日志以来没有发生任何预写式日志活动，pg_switch_wal 不执行任何操作，并返回当前使用的预写式日志文件的起始位置。

pg_create_restore_point 创建一条可用作恢复目标的命名预写式日志记录，并返回相应的预写式日志位置。随后可以在recovery_target_name 中使用该名称，指定恢复到哪一点。应避免创建多个同名恢复点，因为恢复会在第一个名称匹配恢复目标的恢复点停止。

pg_current_wal_lsn 显示当前预写式日志写入位置，格式与上述函数相同。类似地，pg_current_wal_insert_lsn 显示当前预写式日志插入位置，pg_current_wal_flush_lsn 显示当前预写式日志刷盘位置。插入位置是预写式日志在任意时刻的“逻辑”末尾；写入位置是实际从服务器内部缓冲区写出的内容的末尾；刷盘位置则是保证已经写入持久存储的位置。写入位置是能从服务器外部检查到的内容的末尾，如果要归档尚未写满的预写式日志文件，通常需要这个位置。插入位置和刷盘位置主要用于服务器调试。这些都是只读操作，不需要超级用户权限。

可以使用 pg_walfile_name_offset 从上述任一函数的结果中提取相应的预写式日志文件名和字节偏移。例如：

```
postgres=# SELECT * FROM pg_walfile_name_offset(pg_stop_backup());
        file_name         | file_offset
--------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

类似地，pg_walfile_name 仅提取预写式日志文件名。当指定的预写式日志位置恰好位于预写式日志文件边界时，这两个函数都返回前一个预写式日志文件的名称。这通常正是管理预写式日志归档时所需的行为，因为前一个文件是当前需要归档的最后一个文件。

pg_wal_lsn_diff 计算两个预写式日志位置之间的字节差。可以将它与 pg_stat_replication 或表 9.79 中的某些函数配合使用，以获取复制延迟。

有关正确使用这些函数的详细信息，参见第 25.3 节。

9.26.4. 恢复控制函数

表 9.80 中展示的函数提供有关备库当前状态的信息。这些函数可以在恢复或普通运行过程中被执行。

表 9.80. 恢复信息函数

名称	返回类型	描述
pg_is_in_recovery()	bool	如果恢复仍在进行，则为真。
pg_last_wal_receive_lsn()	pg_lsn	获取流复制最近接收并同步到磁盘的预写式日志位置。在流复制进行期间，该值单调增加。如果恢复已完成，该值将保持为恢复期间最后接收并同步到磁盘的 WAL 记录的位置。如果禁用了流复制，或者流复制尚未开始，此函数返回 NULL。
pg_last_wal_replay_lsn()	pg_lsn	获取恢复期间最近重放的预写式日志位置。如果恢复仍在进行，该值单调增加。如果恢复已完成，该值将保持为该次恢复期间最后应用的 WAL 记录的位置。如果服务器未经恢复而正常启动，此函数返回 NULL。

名称	返回类型	描述
pg_last_xact_replay_timestamp()	timestamp with time zone	获取恢复期间最近重放事务的时间戳，即该事务的提交或中止 WAL 记录在主库上生成的时间。如果恢复期间尚未重放任何事务，此函数返回 NULL。否则，如果恢复仍在进行，该值单调增加。如果恢复已完成，该值将保持为该次恢复期间最后应用的事务的时间戳。如果服务器未经恢复而正常启动，此函数返回 NULL。

表 9.81 列出的函数用于控制恢复进度。这些函数只能在恢复期间执行。

表 9.81. 恢复控制函数

名称	返回类型	描述
pg_is_wal_replay_paused()	bool	如果恢复已暂停，则为真。
pg_wal_replay_pause()	void	立即暂停恢复（默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数）。
pg_wal_replay_resume()	void	如果恢复已暂停，则重新开始恢复（默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数）。

恢复暂停时，不再应用数据库更改。如果处于热备状态，所有新查询都会看到同一个一致的数据库快照，并且在恢复继续之前不会再产生查询冲突。

如果禁用了流复制，则暂停状态可能会无限期地持续下去，不会出现问题。如果正在进行流复制，那么将继续接收 WAL 记录，这将最终填满可用磁盘空间，这取决于暂停持续时间、WAL 生成速度和可用磁盘空间。

9.26.5. 快照同步函数

PostgreSQL 允许数据库会话同步它们的快照。一个快照决定对于正在使用该快照的事务哪些数据是可见的。当两个或者更多个会话需要看到数据库中的相同内容时，就需要同步快照。如果两个会话独立开始其事务，就总是有可能有某个第三事务在两个START TRANSACTION 命令的执行之间提交，这样其中一个会话就可以看到该事务的效果而另一个则看不到。

为了解决这个问题，PostgreSQL 允许一个事务导出它正在使用的快照。只要导出快照的事务仍然保持打开，其他事务可以导入它的快照，并且因此可以保证它们可以看到和第一个事务看到的完全一样的数据库视图。但是注意这些事务中的任何一个对数据库所作的更改对其他事务仍然保持不可见，和未提交事务所作的修改一样。因此这些事务是针对以前存在的数据同步，而对由它们自己所作的更改则采取正常的动作。

如表 9.82 中所示，快照通过pg_export_snapshot 函数导出，并且通过SET TRANSACTION 命令导入。

表 9.82. 快照同步函数

名称	返回类型	描述
pg_export_snapshot()	text	保存当前快照并返回其标识符

函数 pg_export_snapshot 保存当前快照，并返回标识该快照的 text 字符串。必须在数据库之外将此字符串传递给要导入该快照的客户端。快照只能在导出它的事务结束之前导入。如有需要，一个事务可以导出多个快照。注意，这只对 READ COMMITTED 事务有用，因为在 REPEATABLE READ 及更高隔离级别中，事务在整个存续期间都使用同一个快照。事务一旦导出过快照，就不能再通过PREPARE TRANSACTION 进行预备。

关于如何使用导出快照的详细说明，请参见 SET TRANSACTION。

9.26.6. 复制函数

表 9.83 中的函数用于控制复制功能并与之交互。相关功能的信息见第 26.2.5 节、第 26.2.6 节和第 50 章。复制源函数仅限超级用户使用。复制槽函数仅限超级用户和具有 REPLICATION 权限的用户使用。

很多这些函数在复制协议中都有等价的命令，见第 53.4 节。

第 9.26.3 节、第 9.26.4 节和第 9.26.5 节中描述的函数也与复制相关。

表 9.83. 复制 SQL 函数

函数	返回类型	描述
pg_create_physical_replication_slot(*slot_name* name [, *immediately_reserve* boolean, *temporary* boolean])	(*slot_name* name, *lsn* pg_lsn)	创建名为 *slot_name* 的新物理复制槽。可选的第二个参数为 true 时，指定立即为此复制槽保留 LSN；否则在流复制客户端首次连接时保留 LSN。从物理槽流式传输更改只能使用流复制协议 – 参见第 53.4 节。可选的第三个参数 *temporary* 为真时，指定该槽不永久存储到磁盘，且仅供当前会话使用。发生任何错误时，临时槽也会被释放。此函数对应复制协议命令 CREATE_REPLICATION_SLOT ... PHYSICAL。
pg_drop_replication_slot(*slot_name* name)	void	删除名为 *slot_name* 的物理或逻辑复制槽。与复制协议命令 DROP_REPLICATION_SLOT 相同。对于逻辑槽，必须连接到创建该槽的同一个数据库来调用。
pg_create_logical_replication_slot(*slot_name* name, *plugin* name [, *temporary* boolean])	(*slot_name* name, *lsn* pg_lsn)	使用输出插件 *plugin* 创建名为 *slot_name* 的新逻辑（解码）复制槽。可选的第三个参数 *temporary* 为真时，指定该槽不永久存储到磁盘，且仅供当前会话使用。发生任何错误时，临时槽也会被释放。调用此函数与复制协议命令 CREATE_REPLICATION_SLOT ... LOGICAL 效果相同。

函数	返回类型	描述
pg_logical_slot_get_changes(*slot_name* name, *upto_lsn* pg_lsn, *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* pg_lsn, *xid* xid, *data* text)	返回槽 *slot_name* 中自上次消费更改的位置起的更改。如果 *upto_lsn* 和 *upto_nchanges* 都为 NULL，逻辑解码会持续到 WAL 末尾。如果 *upto_lsn* 非 NULL，解码仅包含在指定 LSN 之前提交的事务。如果 *upto_nchanges* 非 NULL，解码产生的行数超过指定值时就会停止。不过，实际返回行数可能更大，因为只有在添加完对每个新事务提交进行解码所产生的行后，才会检查此限制。
pg_logical_slot_peek_changes(*slot_name* name, *upto_lsn* pg_lsn, *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* pg_lsn, *xid* xid, *data* text)	行为就像pg_logical_slot_get_changes() 函数，不过改变不会被消费，即在未来的调用中还会返回这些改变。
pg_logical_slot_get_binary_changes(*slot_name* name, *upto_lsn* pg_lsn, *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* pg_lsn, *xid* xid, *data* bytea)	行为就像pg_logical_slot_get_changes() 函数，不过改变会以bytea 返回。
pg_logical_slot_peek_binary_changes(*slot_name* name, *upto_lsn* pg_lsn, *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* pg_lsn, *xid* xid, *data* bytea)	行为与 pg_logical_slot_get_changes() 相同，但更改以 bytea 返回，且不会被消费；也就是说，后续调用还会再次返回这些更改。
pg_replication_slot_advance(*slot_name* name, *upto_lsn* pg_lsn)	(*slot_name* name, *end_lsn* pg_lsn) bool	推进名为 *slot_name* 的复制槽当前已确认位置。该槽不会后退，也不会越过当前插入位置。返回槽名及其实际推进到的位置。如果发生了推进，更新后的槽信息会在后续检查点写出。发生崩溃时，槽可能回到更早的位置。
pg_replication_origin_create(*node_name* text)	oid	以给定外部名称创建复制源，并返回分配给它的内部 ID。
pg_replication_origin_drop(*node_name* text)	void	删除先前创建的复制源，包括关联的重放进度。
pg_replication_origin_oid(*node_name* text)	oid	按名称查找复制源并返回内部 ID。如果找不到这样的复制源，则返回 NULL。
pg_replication_origin_session_setup(*node_name* text)	void	将当前会话标记为从给定复制源重放，以便跟踪重放进度。使用 pg_replication_origin_session_reset 撤销。仅当先前未配置复制源时才能使用。
pg_replication_origin_session_reset()	void	取消 pg_replication_origin_session_setup() 的效果。
pg_replication_origin_session_is_setup()	bool	当前会话是否已配置复制源?

函数	返回类型	描述
pg_replication_origin_session_progress(*flush* bool)	pg_lsn	返回当前会话所配置复制源的重放位置。*flush* 参数决定是否保证对应的本地事务已刷盘。
pg_replication_origin_xact_setup(*origin_lsn* pg_lsn, *origin_timestamp* timestamptz)	void	将当前事务标记为正在重放一个已在给定 LSN 和时间戳提交的事务。仅当先前已用 pg_replication_origin_session_setup() 配置复制源时才能调用。
pg_replication_origin_xact_reset()	void	取消 pg_replication_origin_xact_setup() 的效果。
pg_replication_origin_advance(*node_name* text, *lsn* pg_lsn)	void	将给定结点的复制进度设置为给定位置。这主要用于设置初始位置，或在配置更改等情况之后设置新位置。请注意，疏忽使用此函数可能导致复制数据不一致。
pg_replication_origin_progress(*node_name* text, *flush* bool)	pg_lsn	返回给定复制源的重放位置。*flush* 参数决定是否保证对应的本地事务已刷盘。
pg_logical_emit_message(*transactional* bool, *prefix* text, *content* text)	pg_lsn	发出文本逻辑解码消息。这可用于通过 WAL 向逻辑解码插件传递通用消息。*transactional* 参数指定消息是作为当前事务的一部分，还是立即写入并在逻辑解码读到该记录时立即解码。*prefix* 是文本前缀，便于逻辑解码插件识别其关注的消息。*content* 是消息文本。
pg_logical_emit_message(*transactional* bool, *prefix* text, *content* bytea)	pg_lsn	发出二进制逻辑解码消息。这可用于通过 WAL 向逻辑解码插件传递通用消息。*transactional* 参数指定消息是作为当前事务的一部分，还是立即写入并在逻辑解码读到该记录时立即解码。*prefix* 是文本前缀，便于逻辑解码插件识别其关注的消息。*content* 是消息的二进制内容。

9.26.7. 数据库对象管理函数

表 9.84 中的函数计算数据库对象的磁盘空间用量。

表 9.84. 数据库对象尺寸函数

名称	返回类型	描述
pg_column_size(any)	int	存储某个值所使用的字节数（可能已压缩）
pg_database_size(oid)	bigint	指定 OID 的数据库占用的磁盘空间
pg_database_size(name)	bigint	指定名称的数据库占用的磁盘空间
pg_indexes_size(regclass)	bigint	指定表的索引占用的磁盘空间总量

名称	返回类型	描述
pg_relation_size(*relation* regclass, *fork* text)	bigint	指定表或索引的指定分支（'main'、'fsm'、'vm' 或 'init'）占用的磁盘空间
pg_relation_size(*relation* regclass)	bigint	pg_relation_size(..., 'main') 的简写
pg_size_bytes(text)	bigint	将带有大小单位的人类可读格式转换为字节数
pg_size_pretty(bigint)	text	将以 64 位整数表示的字节大小转换为带有大小单位的人类可读格式
pg_size_pretty(numeric)	text	将以 numeric 值表示的字节大小转换为带有大小单位的人类可读格式
pg_table_size(regclass)	bigint	指定表占用的磁盘空间，不包括索引（但包括 TOAST、空闲空间映射和可见性映射）
pg_tablespace_size(oid)	bigint	指定 OID 的表空间占用的磁盘空间
pg_tablespace_size(name)	bigint	指定名称的表空间占用的磁盘空间
pg_total_relation_size(regclass)	bigint	指定表占用的磁盘空间总量，包括所有索引和 TOAST 数据

pg_column_size 显示存储任意单个数据值所用的空间。

pg_total_relation_size 接受表或 TOAST 表的 OID 或名称，返回该表占用的磁盘总空间，包括所有关联索引。此函数等同于 pg_table_size + pg_indexes_size。

pg_table_size 接受表的 OID 或名称，返回该表所需的磁盘空间，不包括索引。（包括 TOAST 空间、空闲空间映射和可见性映射。）

pg_indexes_size 接受表的 OID 或名称，返回该表所有附属索引占用的磁盘总空间。

pg_database_size 和 pg_tablespace_size 接受数据库或表空间的 OID 或名称，返回其中占用的磁盘总空间。使用 pg_database_size 必须对指定数据库拥有 CONNECT 权限（默认授予），或属于 pg_read_all_stats 角色。使用 pg_tablespace_size 必须对指定表空间拥有 CREATE 权限，或属于 pg_read_all_stats 角色；但如果该表空间是当前数据库的默认表空间，则不受此限制。

pg_relation_size 接受表、索引或 TOAST 表的 OID 或名称，并以字节为单位返回该关系某个分支的磁盘大小。（注意，大多数情况下使用更高层的函数会更方便，例如 pg_total_relation_size 或 pg_table_size，它们会将所有分支的大小相加。）只带一个参数时，返回关系的主数据分支大小。可以提供第二个参数来指定要检查的分支：

- 'main' 返回关系的主数据分支大小。
- 'fsm' 返回关系所关联的空闲空间映射的大小（参见第 69.3 节）。
- 'vm' 返回关系所关联的可见性映射的大小（参见第 69.4 节）。
- 'init' 返回关系所关联的初始化分支的大小（如果存在）。

pg_size_pretty 可以将其他函数的结果格式化为便于阅读的形式，适当地使用 bytes、kB、MB、GB 或 TB 单位。

pg_size_bytes 可以将便于阅读的大小字符串转换为字节数。输入可以使用 bytes、kB、MB、GB 或 TB 单位，解析时不区分大小写。如果没有指定单位，则默认为字节。

> **注意**
>
> pg_size_pretty 和 pg_size_bytes 使用的 kB、MB、GB 和 TB 单位按 2 的幂定义，而不是 10 的幂。因此，1kB 为 1024 字节，1MB 为 1024^2 = 1048576 字节，以此类推。

上述操作表和索引的函数接受一个 regclass 参数，它是该表或索引在pg_class 系统目录中的 OID。你不必手工去查找该 OID，因为regclass 数据类型的输入转换器会为你代劳。只写包围在单引号内的表名，这样它看起来像一个字面量。为了与普通SQL 名称的处理相兼容，该字符串将被转换为小写形式，除非其中在表名周围包含双引号。

如果向上述某个函数传入的 OID 不代表现有对象，则返回 NULL。

表 9.85 中展示的函数帮助标识数据库对象相关的磁盘文件。

表 9.85. 数据库对象位置函数

名称	返回类型	描述
pg_relation_filenode(*relation* regclass)	oid	指定关系的文件节点编号
pg_relation_filepath(*relation* regclass)	text	指定关系的文件路径名
pg_filenode_relation(*tablespace* oid, *filenode* oid)	regclass	查找与给定表空间和文件节点关联的关系

pg_relation_filenode 接受表、索引、序列或 TOAST 表的 OID 或名称，返回当前分配给它的“文件节点”编号。文件节点是关系文件名的基本组成部分（更多信息见第 69.1 节）。对于大多数表，结果与 pg_class.relfilenode 相同；但某些系统目录的 relfilenode 为零，必须使用此函数获取正确值。如果传入的关系没有存储空间（例如视图），函数返回 NULL。

pg_relation_filepath 与 pg_relation_filenode 类似，但返回关系的完整文件路径名（相对于数据库集簇的数据目录 PGDATA）。

pg_filenode_relation 执行 pg_relation_filenode 的逆操作。给定“表空间” OID 和“文件节点”，它返回关联关系的 OID。对于位于数据库默认表空间中的表，表空间可以指定为 0。

表 9.86 列出用于管理排序规则的函数。

表 9.86. 排序规则管理函数

名称	返回类型	描述
pg_collation_actual_version(oid)	text	从操作系统返回排序规则的实际版本
pg_import_system_collations(*schema* regnamespace)	integer	导入操作系统的排序规则

pg_collation_actual_version 返回当前安装在操作系统中的排序规则对象的实际版本。如果它与 pg_collation.collversion 中的值不同，则可能需要重建依赖该排序规则的对象。另见 ALTER COLLATION。

pg_import_system_collations 根据操作系统中找到的所有区域设置，向系统目录 pg_collation 添加排序规则。initdb 使用的就是此函数；更多信息见第 23.2.2 节。如果以后在操作系统中安装了其他区域设置，可以再次运行此函数，为新区域设置添加排序规则。与 pg_collation 中现有条目匹配的区域设置会被跳过。（但此函数不会删除基于操作系统中已不存在的区域设置的排序规则对象。）*schema* 参数通常为 pg_catalog，但并非必须如此；也可以将排序规则安装到其他模式中。函数返回新建的排序规则对象数量。此函数仅限超级用户使用。

9.26.8. 索引维护函数

表 9.87 列出了可用于索引维护任务的函数。这些函数不能在恢复期间执行，仅限超级用户和给定索引的所有者使用。

表 9.87. 索引维护函数

名称	返回类型	描述
brin_summarize_new_values(*index* regclass)	integer	对尚未摘要的页范围进行摘要
brin_summarize_range(*index* regclass, *blockNumber* bigint)	integer	如果给定块所在的页范围尚未摘要，则对其进行摘要
brin_desummarize_range(*index* regclass, *blockNumber* bigint)	integer	如果给定块所在的页范围已被摘要，则移除其摘要
gin_clean_pending_list(*index* regclass)	bigint	将 GIN 待处理列表条目移入主索引结构

brin_summarize_new_values 接受 BRIN 索引的 OID 或名称，检查索引，找出基表中尚未生成索引摘要的页面范围；对于每个这样的范围，它通过扫描表页面创建新的摘要索引元组。函数返回插入索引的新页面范围摘要数量。brin_summarize_range 执行相同的操作，但只对覆盖给定块号的范围生成摘要。

gin_clean_pending_list 接受 GIN 索引的 OID 或名称，通过将待处理列表中的条目批量移到主 GIN 数据结构中，清理指定索引的待处理列表。它返回从待处理列表中移除的页面数。注意，如果参数是禁用了 fastupdate 选项的 GIN 索引，则不执行清理并返回 0，因为该索引没有待处理列表。关于待处理列表和 fastupdate 选项的详细信息，请参见第 66.4.1 节和第 66.5 节。

9.26.9. 通用文件访问函数

表 9.88 中展示的函数提供了对数据库服务器所在机器上的文件的本地访问。只能访问数据库集簇目录以及 log_directory 中的文件，除非用户被授予了角色pg_read_server_files。使用相对路径访问集簇目录里面的文件，以及匹配 log_directory 配置设置的路径访问日志文件。

注意，授予用户 pg_read_file() 或相关函数的 EXECUTE 权限，会允许他们读取服务器上数据库能够读取的任何文件，而且这些读取会绕过数据库内部的所有权限检查。这意味着具有这种访问权限的用户能够读取包含认证信息的 pg_authid 表的内容，以及数据库中的任何文件。因此，授予这些函数的访问权限时应仔细考虑。

表 9.88. 通用文件访问函数

名称	返回类型	描述
pg_ls_dir(*dirname* text [, *missing_ok* boolean, *include_dot_dirs* boolean])	setof text	列出目录内容。默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数。
pg_ls_logdir()	setof record	列出日志目录中文件的名称、大小和最后修改时间。pg_monitor 角色成员拥有访问权限，也可向其他非超级用户角色授予访问权限。
pg_ls_waldir()	setof record	列出 WAL 目录中文件的名称、大小和最后修改时间。pg_monitor 角色成员拥有访问权限，也可向其他非超级用户角色授予访问权限。
pg_read_file(*filename* text [, *offset* bigint, *length* bigint [, *missing_ok* boolean]])	text	返回文本文件的内容。默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数。
pg_read_binary_file(*filename* text [, *offset* bigint, *length* bigint [, *missing_ok* boolean]])	bytea	返回文件内容。默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数。
pg_stat_file(*filename* text[, *missing_ok* boolean])	record	返回文件信息。默认仅限超级用户，但可以向其他用户授予 EXECUTE 权限以运行此函数。

这些函数中的一些接受可选参数 *missing_ok*，用于指定文件或目录不存在时的行为。如果为 true，函数返回 NULL（但 pg_ls_dir 返回空结果集）。如果为 false，则会报错。默认值为 false。

pg_ls_dir 返回指定目录中所有文件的名称（也包括目录和其他特殊文件）。*include_dot_dirs* 指示结果集是否包含 "." 和 ".." 。默认不包含它们（false）；但当 *missing_ok* 为 true 时，包含它们有助于区分空目录和不存在的目录。

pg_ls_logdir 返回日志目录中每个文件的名称、大小和最后修改时间（mtime）。默认情况下，只有超级用户和 pg_monitor 角色的成员可以使用此函数。可以使用 GRANT 授权其他用户访问。不会显示名称以点开头的文件、目录和其他特殊文件。

pg_ls_waldir 返回预写式日志（WAL）目录中每个文件的名称、大小和最后修改时间（mtime）。默认情况下，只有超级用户和 pg_monitor 角色的成员可以使用此函数。可以使用 GRANT 授权其他用户访问。不会显示名称以点开头的文件、目录和其他特殊文件。

pg_read_file 从给定的 *offset* 开始返回文本文件的一部分，最多返回 *length* 字节（如果先到达文件末尾，则返回更少的字节）。如果 *offset* 为负，则它相对于文件末尾。如果省略 *offset* 和 *length*，则返回整个文件。从文件读取的字节按服务器编码解释为字符串；如果它们在该编码下无效，则会报错。

pg_read_binary_file 类似于 pg_read_file，但结果是bytea 值，因此不执行编码检查。与convert_from 函数配合使用，可以按指定编码读取文件：

```
SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');
```

pg_stat_file 返回一条记录，包含文件大小、最后访问时间戳、最后修改时间戳、最后文件状态更改时间戳（仅限 Unix 平台）、文件创建时间戳（仅限 Windows），以及一个boolean 值，表示它是否是目录。典型用法包括：

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

9.26.10. 咨询锁函数

表 9.89 中展示的函数管理咨询锁。有关正确使用这些函数的细节请参考第 13.3.5 节。

表 9.89. 咨询锁函数

名称	返回类型	描述
pg_advisory_lock(*key* bigint)	void	获取排他会话级咨询锁
pg_advisory_lock(*key1* int, *key2* int)	void	获取排他会话级咨询锁
pg_advisory_lock_shared(*key* bigint)	void	获取共享会话级咨询锁
pg_advisory_lock_shared(*key1* int, *key2* int)	void	获取共享会话级咨询锁
pg_advisory_unlock(*key* bigint)	boolean	释放排他会话级咨询锁
pg_advisory_unlock(*key1* int, *key2* int)	boolean	释放排他会话级咨询锁
pg_advisory_unlock_all()	void	释放当前会话持有的所有会话级咨询锁
pg_advisory_unlock_shared(*key* bigint)	boolean	释放共享会话级咨询锁
pg_advisory_unlock_shared(*key1* int, *key2* int)	boolean	释放共享会话级咨询锁
pg_advisory_xact_lock(*key* bigint)	void	获取排他事务级咨询锁
pg_advisory_xact_lock(*key1* int, *key2* int)	void	获取排他事务级咨询锁
pg_advisory_xact_lock_shared(*key* bigint)	void	获取共享事务级咨询锁
pg_advisory_xact_lock_shared(*key1* int, *key2* int)	void	获取共享事务级咨询锁

名称	返回类型	描述
pg_try_advisory_lock(*key* bigint)	boolean	如果可用，则获取排他会话级咨询锁
pg_try_advisory_lock(*key1* int, *key2* int)	boolean	如果可用，则获取排他会话级咨询锁
pg_try_advisory_lock_shared(*key* bigint)	boolean	如果可用，则获取共享会话级咨询锁
pg_try_advisory_lock_shared(*key1* int, *key2* int)	boolean	如果可用，则获取共享会话级咨询锁
pg_try_advisory_xact_lock(*key* bigint)	boolean	如果可用，则获取排他事务级咨询锁
pg_try_advisory_xact_lock(*key1* int, *key2* int)	boolean	如果可用，则获取排他事务级咨询锁
pg_try_advisory_xact_lock_shared(*key* bigint)	boolean	如果可用，则获取共享事务级咨询锁
pg_try_advisory_xact_lock_shared(*key1* int, *key2* int)	boolean	如果可用，则获取共享事务级咨询锁

pg_advisory_lock 锁定应用定义的资源，可以用一个 64 位键值或两个 32 位键值标识资源（注意，这两个键空间不重叠）。如果另一个会话已持有相同资源标识符上的锁，此函数会等待，直到该资源可用。该锁是排他的。多个锁请求会叠加，因此如果同一资源被锁定了三次，就必须解锁三次，才能释放给其他会话使用。

pg_advisory_lock_shared 的行为与 pg_advisory_lock 相同，但该锁可以与其他请求共享锁的会话共享。只有试图获取排他锁的会话会被阻挡。

pg_try_advisory_lock 与 pg_advisory_lock 类似，但不会等待锁变得可用。它要么立即获得锁并返回 true，要么在无法立即获得锁时返回 false。

pg_try_advisory_lock_shared 的行为与 pg_try_advisory_lock 相同，但尝试获取的是共享锁，而不是排他锁。

pg_advisory_unlock 释放先前获得的会话级排他咨询锁。如果成功释放锁，则返回 true。如果未持有该锁，则返回 false，服务器还会报告一条 SQL 警告。

pg_advisory_unlock_shared 的行为与 pg_advisory_unlock 相同，但释放的是会话级共享咨询锁。

pg_advisory_unlock_all 释放当前会话持有的所有会话级咨询锁。（会话结束时会隐式调用此函数，即使客户端非正常断开连接也是如此。）

pg_advisory_xact_lock 的行为与 pg_advisory_lock 相同，但锁会在当前事务结束时自动释放，不能显式释放。

pg_advisory_xact_lock_shared 的行为与 pg_advisory_lock_shared 相同，但锁会在当前事务结束时自动释放，不能显式释放。

pg_try_advisory_xact_lock 的行为与 pg_try_advisory_lock 相同，但如果获得了锁，该锁会在当前事务结束时自动释放，不能显式释放。

pg_try_advisory_xact_lock_shared 的行为与 pg_try_advisory_lock_shared 相同，但如果获得了锁，该锁会在当前事务结束时自动释放，不能显式释放。

9.27. 触发器函数

目前 PostgreSQL 提供一个内置触发器函数 suppress_redundant_updates_trigger，它会阻止那些实际上不会改变行中数据的更新。通常的行为则是无论数据是否改变都执行更新。（这种通常行为不需要检查，因此更新运行得更快，并且在某些情况下也有用。）

理想情况下，应尽量避免执行实际上不会改变记录中数据的更新。冗余更新会耗费大量不必要的时间，尤其是在需要改动许多索引时；它们还会使死行占用空间，最终需要通过清理回收。但是，在客户端代码中检测这种情况并不总是容易，甚至可能无法做到，而编写用于检测的表达式也容易出错。另一种办法是使用 suppress_redundant_updates_trigger，跳过不改变数据的更新。但使用时应当谨慎。此触发器处理每条记录的耗时虽小，却不可忽略，因此如果一次更新涉及的大多数记录确实会改变，使用此触发器反而会使更新更慢。

该 suppress_redundant_updates_trigger 函数可以按如下方式添加到表中：

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE FUNCTION suppress_redundant_updates_trigger();
```

在大多数情况下，应让该触发器对每一行最后触发。由于触发器按名称顺序触发，应选择一个排序位于表上其他所有触发器名称之后的名称。

有关创建触发器的更多信息，请参考 CREATE TRIGGER。

9.28. 事件触发器函数

PostgreSQL 提供了这些辅助函数来从事件触发器检索信息。

更多有关事件触发器的信息请见第 40 章。

9.28.1. 在命令结束处捕捉更改

当在一个 ddl_command_end 事件触发器的函数中调用时，pg_event_trigger_ddl_commands 返回被每一个用户动作执行的 DDL 命令的列表。如果在其他任何环境中调用这个函数，会发生错误。pg_event_trigger_ddl_commands 为每一个被执行的基本命令返回一行，某些由单条 SQL 语句构成的命令可能会返回多于一行。这个函数返回下面的列：

名称	类型	描述
classid	oid	对象所属系统目录的 OID
objid	oid	对象本身的 OID
objsubid	integer	子对象 ID（例如列的属性编号）

名称	类型	描述
command_tag	text	命令标签
object_type	text	对象的类型
schema_name	text	对象所属模式的名称（若有）；否则为NULL。不加引号。
object_identity	text	对象标识的文本表示形式，带模式限定。标识中包含的每个标识符在必要时都会加引号。
in_extension	bool	如果该命令是一个扩展脚本的一部分则为真
command	pg_ddl_command	命令的完整内部表示，不能直接输出，但可以将其传给其他函数以获取关于该命令的不同信息。

9.28.2. 处理被 DDL 命令删除的对象

在命令的 sql_drop 事件中调用 pg_event_trigger_dropped_objects 时，它返回该命令删除的所有对象的列表。如果在其他上下文中调用，pg_event_trigger_dropped_objects 会报错。pg_event_trigger_dropped_objects 返回以下列：

名称	类型	描述
classid	oid	对象原先所属系统目录的 OID
objid	oid	对象本身的 OID
objsubid	integer	子对象 ID（例如列的属性编号）
original	bool	如果这是删除中的一个根对象则为真
normal	bool	如果依赖图中存在指向该对象的普通依赖关系，则为真。
is_temporary	bool	如果该对象是一个临时对象则为真
object_type	text	对象的类型
schema_name	text	对象原先所属模式的名称（若有）；否则为NULL。不加引号。
object_name	text	如果模式和名称的组合可用作该对象的唯一标识符，则为对象名称；否则为NULL。不加引号，并且该名称永远不带模式限定。
object_identity	text	对象标识的文本表示形式，带模式限定。标识中包含的每个标识符在必要时都会加引号。
address_names	text[]	一个数组，可与object_type 和address_args 一起，通过pg_get_object_address() 函数在包含同类同名对象的远程服务器上重建该对象地址。
address_args	text[]	上述address_names 的补充。

pg_event_trigger_dropped_objects 可以被这样用在一个事件触发器中：

```
CREATE FUNCTION test_event_trigger_for_drops()
        RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
    obj record;
BEGIN
    FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % %.% %',
                     tg_tag,
                     obj.object_type,
                     obj.schema_name,
                     obj.object_name,
                     obj.object_identity;
    END LOOP;
END;
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
   ON sql_drop
   EXECUTE FUNCTION test_event_trigger_for_drops();
```

9.28.3. 处理表重写事件

表 9.90 中所示的函数提供有关刚刚为其触发 table_rewrite 事件的表的信息。如果在任何其他环境中调用，会发生错误。

表 9.90. 表重写信息

名称	返回类型	描述
pg_event_trigger_table_rewrite_oid()	Oid	即将重写的表的 OID。
pg_event_trigger_table_rewrite_reason()	int	解释重写原因的原因代码。代码的确切含义取决于版本。

该 pg_event_trigger_table_rewrite_oid 函数可以按如下方式用于事件触发器：

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
BEGIN
  RAISE NOTICE 'rewriting table % for reason %',
                pg_event_trigger_table_rewrite_oid()::regclass,
                pg_event_trigger_table_rewrite_reason();
END;
$$;

CREATE EVENT TRIGGER test_table_rewrite_oid
                  ON table_rewrite
   EXECUTE FUNCTION test_event_trigger_table_rewrite_oid();
```

第 10 章 类型转换

SQL 语句可能有意或无意地要求在同一表达式中混用不同的数据类型。PostgreSQL 提供了丰富的机制来求值这类混合类型表达式。

在很多情况下，用户并不需要理解类型转换机制的细节。不过，PostgreSQL 执行的隐式类型转换会影响查询结果。必要时，可以使用显式类型转换来控制这些结果。

本章介绍 PostgreSQL 的类型转换机制和约定。有关特定数据类型以及允许使用的函数和操作符的更多信息，请参阅第 8 章和第 9 章中的相关章节。

10.1. 概述

SQL 是一种强类型语言。也就是说，每个数据项都关联着一个决定其行为和允许用法的数据类型。PostgreSQL 拥有可扩展的类型系统，比其他SQL 实现更通用也更灵活。因此，PostgreSQL 中的大多数类型转换行为都由通用规则控制，而不是依赖临时性的启发式规则。这使得即使表达式中包含用户定义类型，也能够使用混合类型表达式。

PostgreSQL 扫描器/解析器将词法元素分成五个基本类别：整数、非整数数字、字符串、标识符和关键字。大多数非数值类型的常量会先被归类为字符串。SQL 语言定义允许给字符串指定类型名，而PostgreSQL 可以利用这一机制让解析器从一开始就沿着正确的路径处理。例如，查询：

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

包含两个字面量，它们的类型分别是`text` 和 `point`。如果没有为某个字符串字面量指定类型，那么它最初会被赋予占位类型`unknown`，并在后文所述的后续阶段解析。

在 PostgreSQL 解析器中，有四种基本的SQL 结构需要采用不同的类型转换规则：

函数调用

PostgreSQL 类型系统的很大一部分建立在一套丰富的函数之上。函数可以有一个或多个参数。由于 PostgreSQL 支持函数重载，单凭函数名无法唯一确定要调用哪个函数；解析器必须根据所提供参数的数据类型选择正确的函数。

操作符

PostgreSQL 允许表达式使用前缀和后缀一元（单参数）操作符，以及二元（双参数）操作符。与函数一样，操作符也可以重载，因此同样存在选择正确操作符的问题。

值存储

SQL 的 `INSERT` 和 `UPDATE` 语句会把表达式的结果放入表中。语句中的表达式必须与目标列的类型匹配，并且在必要时转换为目标列类型。

UNION、CASE 和相关结构

由于经 UNION 合并的 SELECT 语句的所有查询结果都必须出现在同一组列中，因此每个 SELECT 子句的结果类型必须彼此匹配，并转换为统一的一组类型。类似地，CASE 结构中的结果表达式必须转换为某种公共类型，这样整个CASE 表达式才有确定的输出类型。其他一些结构，如ARRAY[] 以及 GREATEST 和 LEAST 函数，也同样需要为若干子表达式确定公共类型。

系统目录保存了哪些数据类型之间存在哪些类型转换，以及如何执行这些转换的信息。用户还可以使用CREATE CAST 命令添加额外的类型转换。（这通常与定义新数据类型一并完成。内置类型之间的类型转换集合经过精心设计，最好不要改动。）

解析器还提供了一种额外的启发式规则，可在存在隐式类型转换的一组类型之间更准确地判定适当的转换行为。数据类型被划分为若干基本的类型分类，包括boolean、numeric、string、bitstring、datetime、timespan、geometric、network 以及用户定义类型。（列表见表 52.63；不过也可以创建自定义类型分类。）每个分类中可以有一个或多个首选类型，当存在多种可能类型可选时，会优先选择它们。通过仔细选择首选类型和可用的隐式类型转换，可以确保有歧义的表达式（即存在多个候选解析方案的表达式）能够以有用的方式被解析。

所有类型转换规则都以以下几个原则为基础：

- 隐式转换绝不应产生令人意外或无法预测的结果。
- 如果查询不需要隐式类型转换，解析器和执行器就不应承担额外开销。也就是说，如果查询本身书写正确且类型已经匹配，那么执行时不应让解析器额外耗时，也不应在查询中引入不必要的隐式类型转换调用。
- 此外，如果某个查询通常需要先对参数做隐式类型转换才能调用某个函数，而此后用户又定义了一个参数类型正确的新函数，那么解析器应改为使用这个新函数，而不再通过隐式转换去调用旧函数。

10.2. 操作符

操作符表达式所引用的具体操作符按照下述过程确定。注意，该过程还会间接受到相关操作符优先级的影响，因为优先级决定了哪些子表达式会被视为哪些操作符的输入。详见第 4.1.6 节。

操作符类型解析

1. 从系统目录 pg_operator 中选取要考虑的操作符。如果使用的是未带模式限定的操作符名（通常如此），则考虑当前搜索路径中可见且名称和参数个数匹配的操作符（见第 5.8.3 节）。如果给出了限定名，则只考虑指定模式中的操作符。
    - (可选的) 如果搜索路径中找到多个参数类型完全相同的操作符，只考虑路径中最早出现的那一个。参数类型不同的操作符则不受搜索路径位置影响，处于同等地位。
2. 检查是否存在一个恰好接受输入参数类型的操作符。如果存在（在被考虑的操作符集合中至多只有一个精确匹配），就使用它。若通过限定名调用（这种情况并不常见）某个允许不受信任的用户创建对象的模式中的操作符，而又缺少精确匹配，就会带来安全隐患[1]。在这种情况下，应通过对参数进行类型转换来强制获得精确匹配。

[1] 对于未带模式限定的名称，不会出现这种隐患，因为包含允许不受信任的用户创建对象的模式的搜索路径不是一种模式的安全使用方式。

a. (可选的) 如果二元操作符调用的一个参数是 `unknown` 类型，则在本次检查中假定它与另一个参数类型相同。涉及两个`unknown` 输入的调用，或带有`unknown` 输入的一元操作符，在这一步都不可能找到匹配。

b. (可选的) 如果二元操作符调用的一个参数是 `unknown` 类型，而另一个是域类型，则接着检查是否存在两边都恰好接受该域基础类型的操作符；如果有，就使用它。

3. 寻找最佳匹配。

a. 丢弃那些输入类型不匹配，且也无法通过隐式类型转换实现匹配的候选操作符。为此，假定`unknown` 字面量可以转换为任何类型。如果只剩一个候选操作符，就使用它；否则继续下一步。

b. 如果任何输入参数是域类型，则在后续所有步骤中都把它当作该域的基础类型。这样可以确保在解决操作符歧义时，域的行为与其基础类型一致。

c. 遍历所有候选操作符，保留那些在输入类型上精确匹配数最多的候选项。如果没有任何候选项存在精确匹配，则保留全部候选项。如果只剩一个候选操作符，就使用它；否则继续下一步。

d. 遍历所有候选操作符，在需要类型转换的位置上，保留接受首选类型（属于输入数据类型的类型分类）的位置数最多的候选项。如果没有候选项接受首选类型，则保留全部候选项。如果只剩一个候选操作符，就使用它；否则继续下一步。

e. 如果有任何输入参数是 `unknown`，就检查剩余候选操作符在这些参数位置上接受哪些类型分类。在每个位置上，如果有任何候选项接受`string` 分类，就选择该分类。（这种偏向字符串的做法是合适的，因为`unknown` 类型的字面量看起来像字符串。）否则，如果所有剩余候选项都接受同一种类型分类，就选择该分类；否则失败，因为在缺少更多线索的情况下无法推断出正确选择。然后丢弃不接受所选类型分类的候选项。此外，如果有任何候选项接受该分类中的首选类型，就丢弃在该参数上接受非首选类型的候选项。如果没有候选项通过这些测试，则保留全部候选项。如果只剩一个候选操作符，就使用它；否则继续下一步。

f. 如果同时存在 `unknown` 参数和已知类型参数，且所有已知类型参数都具有相同类型，则假定`unknown` 参数也属于该类型，并检查哪些候选项能在`unknown` 参数位置接受该类型。如果恰好只有一个候选项通过测试，就使用它；否则失败。

下面是一些示例。

例 10.1. 阶乘操作符类型解析

标准目录中只定义了一个阶乘操作符（后缀 !），它接受一个类型为 bigint 的参数。扫描器为以下查询表达式中的参数赋予初始类型 integer：

```
SELECT 40 ! AS "40 factorial";

                  40 factorial
-------------------------------------------------
 815915283247897734345611269596115894272000000000
(1 row)
```

因此，解析器会对操作数进行类型转换，该查询等价于：

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

例 10.2. 字符串连接操作符类型解析

处理字符串类型以及复杂的扩展类型时，常常会使用类似字符串的语法。未指定类型的字符串会与可能的操作符候选项进行匹配。

一个参数未指定类型的例子：

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

在这种情况下，解析器会查看是否存在一个两边参数都接受text 的操作符。既然存在，它就会假定第二个参数应解释为text 类型。

下面是两个未指定类型值的连接：

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

这里对使用哪种类型没有初始提示，因为查询中没有指定任何类型。因此，解析器会查找所有候选操作符，并发现既有接受字符串分类输入的候选项，也有接受位串分类输入的候选项。由于在可用时优先选择字符串分类，所以会选中该分类，再使用字符串的首选类型text 作为解析这些unknown 类型字面量的具体类型。

例 10.3. 绝对值与取反操作符类型解析

PostgreSQL 操作符目录中为前缀操作符@提供了多个条目，它们分别实现各种数值数据类型的绝对值操作。其中一个条目对应float8，它是数值分类中的首选类型。因此，当遇到一个unknown 输入时，PostgreSQL 会使用该条目：

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

这里，系统在应用所选操作符之前，已经把unknown 类型字面量隐式解析为 float8。我们可以验证使用的确实是 float8，而不是其他类型：

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

另一方面，前缀操作符~（按位取反）只对整数数据类型定义，并没有为float8 定义。因此，如果对~试一个类似的例子，会得到：

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You might need to add
explicit type casts.
```

这是因为系统无法决定几个可能的~操作符中应优先选择哪一个。我们可以通过显式类型转换来帮助它：

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

例 10.4. 数组包含操作符类型解析

下面是另一个解析带有一个已知类型输入和一个未知类型输入的操作符的例子：

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset
-----------
 t
(1 row)
```

PostgreSQL 操作符目录中为中缀操作符<@定义了多个条目，但左侧能够接受整数数组的只有两种：数组包含（anyarray <@ anyarray）和范围包含（anyelement <@ anyrange）。由于这些多态伪类型（见第 8.21 节）都不被视为首选类型，解析器无法据此消除歧义。不过，步骤 3.f 要求它假定 unknown 类型字面量与另一输入具有相同类型，也就是整数数组。这样两个操作符中只有一个能够匹配，因此会选择数组包含。（如果选中范围包含，就会报错，因为该字符串的格式并不是合法的范围字面量。）

例 10.5. 域类型上的自定义操作符

用户有时会尝试声明只适用于某个域类型的操作符。这是可行的，但远没有看上去那样有用，因为操作符解析规则被设计成优先选择作用于域基础类型的操作符。考虑下面的例子：

```
CREATE DOMAIN mytext AS text CHECK(...);
CREATE FUNCTION mytext_eq_text (mytext, text) RETURNS boolean AS ...;
CREATE OPERATOR = (procedure=mytext_eq_text, leftarg=mytext, rightarg=text);
CREATE TABLE mytable (val mytext);

SELECT * FROM mytable WHERE val = 'foo';
```

这个查询不会使用自定义操作符。解析器首先会检查是否存在 mytext = mytext 操作符（步骤 2.a），但并不存在；然后它会考虑该域的基础类型text，并检查是否存在text = text 操作符（步骤 2.b），而这是存在的；于是它会把unknown 类型的字面量解析为 text，并使用text = text 操作符。要让自定义操作符被使用，唯一的方法是显式地对该字面量做类型转换：

```
SELECT * FROM mytable WHERE val = text 'foo';
```

这样就会根据精确匹配规则立即找到 mytext = text 操作符。如果解析过程进入最佳匹配规则，它们会主动排斥作用于域类型的操作符。如果不是这样，这类操作符会导致过多的操作符歧义失败，因为类型转换规则总是把域视为可以转换到其基础类型或从其基础类型转换而来，因此域操作符会在所有与基础类型上同名操作符相同的场景中都被视为可用。

10.3. 函数

函数调用所引用的具体函数按照下述过程确定。

函数类型解析

1. 从系统目录 pg_proc 中选取要考虑的函数。如果使用的是未带模式限定的函数名，则考虑当前搜索路径中可见且名称和参数个数匹配的函数（见第 5.8.3 节）。如果给出了限定名，则只考虑指定模式中的函数。

a. (可选的) 如果搜索路径中找到多个参数类型完全相同的函数，只考虑路径中最早出现的那一个。参数类型不同的函数则不受搜索路径位置影响，处于同等地位。

b. (可选的) 如果函数声明了一个 VARIADIC 数组参数，而调用时没有使用VARIADIC 关键字，则把数组参数视为一个或多个其元素类型的参数，个数按匹配该调用的需要确定。扩展之后，该函数的有效参数类型可能与某个非VARIADIC 函数完全相同。在这种情况下，使用搜索路径中较早出现的函数；如果两个函数位于同一模式，则优先选择非 VARIADIC 函数。

   通过限定名调用某个允许不受信任的用户创建对象的模式中找到的可变参数函数时，会带来安全隐患[2]。恶意用户可以借此劫持调用，并以你的身份执行任意 SQL 函数。改为使用带有VARIADIC 关键字的调用可以绕过这一隐患。对于填充 VARIADIC "any" 参数的调用，往往没有包含VARIADIC 关键字的等价写法。要安全地发出这类调用，该函数所在模式必须只允许受信任的用户创建对象。

c. (可选的) 带有参数默认值的函数，被认为可以匹配任何省略了零个或多个可使用默认值的参数位置的调用。如果不止一个这样的函数能匹配某个调用，则使用搜索路径中最早出现的那一个。如果同一模式中存在两个或更多这样的函数，并且它们在未使用默认值的参数位置上的参数类型相同（如果它们具有不同的带有默认值的参数集合，这种情况是可能的），系统就无法判断该偏好哪一个；如果找不到更好的匹配，就会报“有歧义的函数调用”错误。

   通过限定名[2]调用某个允许不受信任的用户创建对象的模式中的任意函数时，会带来可用性隐患。恶意用户可以用现有函数的名字创建一个新函数，复制该函数的参数，并附加带有默认值的新参数。这样就会阻止对原始函数的新调用。为避免这种隐患，应把函数放在只允许受信任的用户创建对象的模式中。

2. 检查是否存在一个恰好接受输入参数类型的函数。如果存在（在被考虑的函数集合中至多只有一个精确匹配），就使用它。若通过限定名[2]调用某个允许不受信任的用户创建对象的模式中的函数，而又缺少精确匹配，就会带来安全隐患。在这种情况下，应通过对参数进行类型转换来强制获得精确匹配。（涉及unknown 的情况在这一步永远找不到匹配。）

3. 如果没有找到精确匹配，就检查该函数调用是否看起来像一种特殊的类型转换请求。这种情况发生在：函数调用只有一个参数，且函数名与某个数据类型的（内部）名称相同。此外，函数参数必须是unknown 类型字面量，或者是一种可以二进制强制转换为该命名数据类型的类型，或者是一种可以通过应用其 I/O 函数转换为该命名数据类型的类型（也就是说，该转换要么转到某种标准字符串类型，要么从某种标准字符串类型转来）。满足这些条件时，该函数调用会被当作一种CAST 形式。[3]

4. 寻找最佳匹配。

   a. 丢弃那些输入类型不匹配，且也无法通过隐式类型转换实现匹配的候选函数。为此，假定unknown 字面量可以转换为任何类型。如果只剩一个候选函数，就使用它；否则继续下一步。

   b. 如果任何输入参数是域类型，则在后续所有步骤中都把它当作该域的基础类型。这样可以确保在解决函数歧义时，域的行为与其基础类型一致。

[2] 对于未带模式限定的名称，不会出现这种隐患，因为包含允许不受信任的用户创建对象的模式的搜索路径不是一种模式的安全使用方式。

[3] 之所以有这一步，是为了在不存在实际类型转换函数的情况下仍支持函数风格的类型转换写法。如果存在类型转换函数，按惯例它会以其输出类型命名，因此无需为此设置特殊情况。更多说明见 CREATE CAST。

c. 遍历所有候选函数，保留那些在输入类型上精确匹配数最多的候选项。如果没有任何候选项存在精确匹配，则保留全部候选项。如果只剩一个候选函数，就使用它；否则继续下一步。

d. 遍历所有候选函数，在需要类型转换的位置上，保留接受首选类型（属于输入数据类型的类型分类）的位置数最多的候选项。如果没有候选项接受首选类型，则保留全部候选项。如果只剩一个候选函数，就使用它；否则继续下一步。

e. 如果有任何输入参数是 unknown，就检查剩余候选函数在这些参数位置上接受哪些类型分类。在每个位置上，如果有任何候选项接受string 分类，就选择该分类。（这种偏向字符串的做法是合适的，因为unknown 类型的字面量看起来像字符串。）否则，如果所有剩余候选项都接受同一种类型分类，就选择该分类；否则失败，因为在缺少更多线索的情况下无法推断出正确选择。然后丢弃不接受所选类型分类的候选项。此外，如果有任何候选项接受该分类中的首选类型，就丢弃在该参数上接受非首选类型的候选项。如果没有候选项通过这些测试，则保留全部候选项。如果只剩一个候选函数，就使用它；否则继续下一步。

f. 如果同时存在 unknown 参数和已知类型参数，且所有已知类型参数都具有相同类型，则假定unknown 参数也属于该类型，并检查哪些候选项能在unknown 参数位置接受该类型。如果恰好只有一个候选项通过测试，就使用它；否则失败。

注意，操作符类型解析和函数类型解析的“最佳匹配”规则是完全相同的。下面是一些示例。

例 10.6. round 函数参数类型解析

只有一个接受两个参数的 round 函数；它的第一个参数类型是 numeric，第二个参数类型是 integer。因此下面的查询会自动把类型为 integer 的第一个参数转换为 numeric：

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

该查询实际上会被解析器改写成：

```
SELECT round(CAST (4 AS numeric), 4);
```

由于带小数点的数字常量最初会被赋予 numeric 类型，下面的查询不需要类型转换，因此可能稍微更高效一些：

```
SELECT round(4.0, 4);
```

例 10.7. 可变参数函数解析

```
CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS int
  LANGUAGE sql AS 'SELECT 1';
CREATE FUNCTION
```

这个函数接受但不要求使用 VARIADIC 关键字。它既能接受 integer 参数，也能接受numeric 参数：

```
SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                1 |                1 |                1
(1 row)
```

但是，如果有更具体的函数可用，第一和第二个调用会优先选择它们：

```
CREATE FUNCTION public.variadic_example(numeric) RETURNS int
  LANGUAGE sql AS 'SELECT 2';
CREATE FUNCTION

CREATE FUNCTION public.variadic_example(int) RETURNS int
  LANGUAGE sql AS 'SELECT 3';
CREATE FUNCTION

SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                3 |                2 |                1
(1 row)
```

在默认配置下，且只存在第一个函数时，第一和第二个调用并不安全。任何用户都可以通过创建第二个或第三个函数来劫持它们。第三个调用由于参数类型精确匹配且使用了VARIADIC 关键字，因此是安全的。

例 10.8. substr 函数类型解析

存在多个 substr 函数，其中一个接受text 和 integer。如果以未指定类型的字符串常量调用，系统会选择接受首选类型分类string 参数的候选函数（也就是 text 类型的那个）。

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

如果字符串被声明为 varchar 类型，例如它来自表中的某一列，那么解析器会尝试把它转换为 text：

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

解析器会把它改写成如下形式：

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **注意**
>
> 解析器从 pg_cast 目录得知 text 与 varchar 二进制兼容，这意味着把其中一种类型传给接受另一种类型的函数时，无需做任何物理转换。因此，这种情况下实际上不会插入类型转换调用。

如果用 integer 类型参数调用该函数，解析器会尝试把它转换为text：

```
SELECT substr(1234, 3);
ERROR:  function substr(integer, integer) does not exist
HINT:  No function matches the given name and argument types. You might need
to add explicit type casts.
```

这行不通，因为integer 并没有到 text 的隐式类型转换。不过，显式类型转换可以：

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
     34
(1 row)
```

10.4. 值存储

要插入表中的值会按照以下步骤转换为目标列的数据类型。

值存储的类型转换

1. 检查是否与目标类型精确匹配。

2. 否则，尝试把表达式转换为目标类型。如果这两种类型之间在pg_cast 目录中登记了赋值类型转换（见 CREATE CAST），就可以这样做。另一种情况是，如果表达式是unknown 类型字面量，则会把字符串字面量的内容送入目标类型的输入转换例程。

3. 检查目标类型是否有尺寸调整类型转换。尺寸调整类型转换是该类型到其自身的类型转换。如果在 pg_cast 目录中找到了这种转换，就在把表达式存入目标列之前先应用它。此类类型转换的实现函数总是额外接受一个integer 参数，用来接收目标列的atttypmod 值（通常是其声明长度，尽管不同数据类型对atttypmod 的解释各不相同）；它还可能接受第三个 boolean 参数，用来说明该类型转换是显式还是隐式的。类型转换函数负责应用所有与长度相关的语义，例如长度检查或截断。

例 10.9. character 的存储类型转换

对于声明为 character(20) 的目标列，下面的语句表明存储值会被调整到正确的长度：

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
 abcdef               |           20
(1 row)
```

这里实际发生的是，两个unknown 类型字面量默认都被解析为 text，从而使||操作符被解析为 text 的连接操作。然后，操作符得到的text 结果被转换为 bpchar（“用空格填充的字符”，blank-padded char，即character 数据类型的内部名称），以匹配目标列类型。（由于从 text 到 bpchar 的转换是二进制可强制转换的，这一步不会插入任何真实的函数调用。）最后，系统从系统目录中找到尺寸调整函数bpchar(bpchar, integer, boolean)，并把它应用到操作符结果以及存储列长度上。这个特定于类型的函数会执行所需的长度检查并补齐填充空格。

10.5. UNION、CASE 及相关结构

SQL 的 UNION 结构必须让可能不同的类型彼此匹配，以形成单一结果集。该解析算法会分别应用到联合查询的每个输出列。INTERSECT 和 EXCEPT 结构以与 UNION 相同的方式解析不同类型。其他一些结构，包括CASE、ARRAY、VALUES 以及 GREATEST 和 LEAST 函数，也使用完全相同的算法来匹配其组成表达式并选择结果数据类型。

UNION、CASE 及相关结构的类型解析

1. 如果所有输入都是同一类型，且该类型不是unknown，就解析为该类型。

2. 如果任何输入是域类型，则在后续所有步骤中都把它当作该域的基础类型。[4]

3. 如果所有输入都是 unknown 类型，则解析为text 类型（字符串分类的首选类型）。否则，在剩余规则中忽略unknown 输入。

4. 如果非 unknown 输入并非全都属于同一类型分类，则失败。

[4] 这在某种程度上类似于操作符和函数对域输入的处理方式；只要用户确保所有输入都隐式或显式地正好属于该域类型，这种行为就能让域类型在UNION 或类似结构中得以保留。否则，就会使用该域的基础类型。

5. 选择第一个非 unknown 输入的类型作为候选类型，然后按从左到右的顺序考虑其余每个非 unknown 输入类型。[5]如果候选类型可以隐式转换为另一类型，而反过来不行，则把另一类型选为新的候选类型。然后继续考虑剩余输入。如果在此过程的任何阶段选中了首选类型，就停止继续考虑其他输入。

6. 把所有输入都转换为最终候选类型。如果某个输入类型到候选类型不存在隐式转换，则失败。

下面是一些示例。

例 10.10. UNION 中未充分指定类型时的类型解析

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

这里，unknown 类型字面量'b' 会被解析为 text 类型。

例 10.11. 简单 UNION 中的类型解析

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

字面量 1.2 的类型是 numeric，而integer 值 1 可以隐式转换为 numeric，因此使用该类型。

例 10.12. 次序对调的 UNION 中的类型解析

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

这里，由于real 类型不能隐式转换为 integer，而integer 可以隐式转换为 real，因此 UNION 结果类型被解析为 real。

[5] 出于历史原因，CASE 会把其 ELSE 子句（如果有）视为“第一个”输入，THEN 子句则在其后考虑。其他所有情况中，“从左到右”都指表达式在查询文本中出现的顺序。

例 10.13. 嵌套 UNION 中的类型解析

```
SELECT NULL UNION SELECT NULL UNION SELECT 1;

ERROR:  UNION types text and integer cannot be matched
```

之所以失败，是因为PostgreSQL 把多个 UNION 视为由成对操作组成的嵌套；也就是说，这个输入等同于：

```
(SELECT NULL UNION SELECT NULL) UNION SELECT 1;
```

根据上述规则，内层UNION 会被解析为输出 text 类型。随后外层 UNION 的输入类型分别是 text 和 integer，于是就得到上面的错误。解决办法是确保最左边的UNION 至少有一个输入属于期望的结果类型。

INTERSECT 和 EXCEPT 也同样按成对方式解析。不过，本节介绍的其他结构会在一次解析步骤中同时考虑它们的所有输入。

10.6. SELECT 输出列

前面各节给出的规则会为 SQL 查询中的所有表达式赋予非 unknown 数据类型，但作为 SELECT 命令的简单输出列出现的未指定类型字面量除外。例如，在

```
SELECT 'Hello World';
```

中，没有任何信息可用于确定应将该字符串字面量视为什么类型。在这种情况下，PostgreSQL 会回退为将该字面量的类型解析为 text。

当 SELECT 是 UNION（或 INTERSECT、EXCEPT）结构的一端，或者出现在INSERT ... SELECT 中时，这条规则不适用，因为前面各节给出的规则具有更高优先级。在前一种情况下，未指定类型字面量的类型可以从UNION 的另一侧取得；在后一种情况下，则从目标列取得。

为此目的，RETURNING 列表与 SELECT 输出列表一视同仁。

> **注意**
>
> 在 PostgreSQL 10 之前，这条规则并不存在，SELECT 输出列表中的未指定类型字面量会被保留为 unknown 类型。这会带来各种不良后果，因此后来做了修改。

第 11 章 索引

索引是增强数据库性能的一种常见方式。索引允许数据库服务器比没有索引时更快地找到并检索特定行。但索引也会给整个数据库系统带来额外开销，因此应当合理使用。

11.1. 简介

假设我们有一个类似如下的表：

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

并且应用会发出很多如下形式的查询：

```
SELECT content FROM test1 WHERE id = constant;
```

如果没有预先准备，系统就必须逐行扫描整个`test1` 表，以找出所有匹配项。如果`test1` 中有很多行，而这种查询只会返回很少几行（甚至可能一行也没有，或者只有一行），这显然是低效的办法。但是如果系统被要求维护一个基于`id` 列的索引，它就可以用更高效的方法定位匹配行。例如，它可能只需要在一棵搜索树中向下走几层。

大多数非虚构类书籍也采用了类似的方法：读者经常查找的术语和概念会被收集到书末按字母顺序排列的索引中。感兴趣的读者可以相对快速地浏览索引并翻到相应页面，而不必通读整本书才能找到自己感兴趣的内容。正如作者需要预判读者可能查找哪些条目一样，数据库程序员也需要预见哪些索引会有用。

如前所述，下列命令可用于在`id` 列上创建索引：

```
CREATE INDEX test1_id_index ON test1 (id);
```

`test1_id_index` 这个名字可以自由选择，但最好选一个以后仍能让你记住该索引用途的名字。

要删除索引，使用`DROP INDEX` 命令。索引可以在任何时候添加到表上，也可以在任何时候从表上删除。

一旦创建了索引，就无需再进行额外干预：系统会在表被修改时更新索引，并且会在它认为这样做比顺序扫描表更高效时，在查询中使用该索引。不过，你可能仍需要定期运行`ANALYZE` 命令来更新统计信息，以便查询规划器作出更有根据的决策。关于如何判断某个索引是否被使用，以及规划器何时和为何可能选择不使用索引，可参见第 14 章。

索引还可以让带有搜索条件的 `UPDATE` 和 `DELETE` 命令受益。索引也可以用于连接搜索。因此，定义在连接条件组成列上的索引也能显著加速带连接的查询。

在大表上创建索引可能需要很长时间。默认情况下，PostgreSQL 允许在创建索引期间并行执行读取（`SELECT` 语句），但写入（`INSERT`、`UPDATE`、`DELETE`）会被阻塞，直到索引构建完成。在生产环境中，这往往不可接受。可以允许在创建索引时并行写入，但有若干注意事项需要了解，详见并发构建索引。

索引创建之后，系统还必须使其与表保持同步。这会给数据操作增加开销。索引还可能阻止创建堆内元组。因此，那些在查询中很少使用或从不使用的索引应当被移除。

11.2. 索引类型

PostgreSQL 提供了多种索引类型：B-树、Hash、GiST、SP-GiST、GIN、BRIN，以及扩展bloom。每种索引类型都采用不同的算法，分别最适合不同类型的查询。默认情况下，`CREATE INDEX` 命令创建的是 B-树索引，它适用于最常见的场景。

B-树能够处理可以按某种顺序排序的数据上的等值查询和范围查询。特别是，只要已索引列参与了下列任一操作符的比较，PostgreSQL 查询规划器就会考虑使用 B-树索引：

```
<
<=
=
>=
>
```

与这些操作符组合等价的构造，例如`BETWEEN` 和 `IN`，也可以通过 B-树索引搜索来实现。此外，索引列上的`IS NULL` 或 `IS NOT NULL` 条件也可以配合 B-树索引使用。

如果模式是常量，并且锚定在字符串起始位置，优化器也可以对涉及模式匹配操作符`LIKE` 和`~`的查询使用 B-树索引，例如`col LIKE 'foo%'` 或 `col ~ '^foo'`，但不能用于 `col LIKE '%bar'`。不过，如果你的数据库没有使用 C 区域设置，就需要以一个特殊操作符类来创建该索引，才能支持模式匹配查询的索引化，详见下文第 11.10 节。B-树索引也可以用于 `ILIKE` 和`~*`，但前提是模式以非字母字符开头，也就是不会受到大小写转换影响的字符。

B-树索引还可以用于按排序顺序取回数据。这并不总是比简单扫描再排序更快，但通常会很有帮助。

Hash 索引只能处理简单的等值比较。只要已索引列参与的是使用`=`操作符的比较，查询规划器就会考虑使用 Hash 索引。使用以下命令创建 Hash 索引：

```
CREATE INDEX name ON table USING HASH (column);
```

GiST 索引并不是某一种单独的索引，而是一种基础设施，可在其中实现许多不同的索引策略。因此，GiST 索引可使用哪些具体操作符，取决于所采用的索引策略（即操作符类）。例如，PostgreSQL 标准发布版中包含了若干二维几何数据类型的 GiST 操作符类，它们支持使用下列操作符的索引化查询：

```
<<
&<
&>
>>
<<|
&<|
|&>
|>>
@>
<@
~=
&&
```

（这些操作符的含义见第 9.11 节。）标准发布版自带的 GiST 操作符类记录在表 64.1 中。还有许多其他 GiST 操作符类可在 `contrib` 集合或独立项目中获得。更多信息见第 64 章。

GiST 索引还能够优化“最近邻”搜索，例如：

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

这会找出距离给定目标点最近的十个地点。是否能做到这一点，同样取决于所使用的具体操作符类。在表 64.1 中，可以按这种方式使用的操作符列在“排序操作符”这一列中。

与 GiST 一样，SP-GiST 索引也提供一种支持多种搜索的基础设施。SP-GiST 允许实现大量不同的、非平衡的、基于磁盘的数据结构，例如四叉树、k-d 树和基数树（trie）。例如，PostgreSQL 标准发布版中包含了用于二维点的 SP-GiST 操作符类，它支持使用下列操作符的索引化查询：

```
<<
>>
~=
<@
<^
>^
```

（这些操作符的含义见第 9.11 节。）标准发布版自带的 SP-GiST 操作符类记录在表 65.1 中。更多信息见第 65 章。

GIN 索引是“倒排索引”，适用于包含多个组成值的数据值，例如数组。倒排索引会为每个组成值保存单独的项，因此能够高效处理测试特定组成值是否存在的查询。

与 GiST 和 SP-GiST 一样，GIN 也能支持多种不同的用户定义索引策略，GIN 索引可使用哪些具体操作符同样取决于索引策略。例如，PostgreSQL 标准发布版中包含一个用于数组的 GIN 操作符类，它支持使用下列操作符的索引化查询：

```
<@
@>
=
&&
```

（这些操作符的含义见第 9.18 节。）标准发布版自带的 GIN 操作符类记录在表 66.1 中。还有许多其他 GIN 操作符类可在 `contrib` 集合或独立项目中获得。更多信息见第 66 章。

BRIN 索引（块范围索引，Block Range Indexes）存储的是关于表中连续物理块范围内所保存值的摘要信息。与 GiST、SP-GiST 和 GIN 一样，BRIN 也可以支持多种不同的索引策略，而 BRIN 索引可使用哪些具体操作符取决于所采用的索引策略。对于具有线性排序顺序的数据类型，每个块范围上被索引的数据对应于该列值的最小值和最大值。这支持使用下列操作符的索引化查询：

```
<
<=
=
>=
>
```

标准发布版自带的 BRIN 操作符类记录在表 67.1 中。更多信息见第 67 章。

11.3. 多列索引

一个索引可以定义在表的多个列上。例如，如果你有一个如下形式的表：

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

（假设你把/dev 目录保存在数据库里.....）并且经常发出如下查询：

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

那么将 major 和 minor 两列一起建一个索引可能是合适的，例如：

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

当前，只有 B-树、GiST、GIN 和 BRIN 索引类型支持多列索引。最多可以指定 32 列。（这个限制可以在构建 PostgreSQL 时修改；参见文件 pg_config_manual.h。）

多列 B-树索引可以用于涉及索引任意列子集的查询条件，但当对前导（最左）列存在约束时，索引效率最高。精确的规则是：对前导列的等值约束，再加上第一个没有等值约束的列上的任何不等约束，会被用来限制索引扫描的范围。这些列右侧列上的约束会在索引中进行检查，因此可以减少访问表本体的次数，但不会缩小必须扫描的索引范围。例如，给定一个 (a, b, c) 上的索引和查询条件 WHERE a = 5 AND b >= 42 AND c < 77，索引必须从第一个 a = 5 且 b = 42 的条目扫描到最后一个 a = 5 的条目。c >= 77 的索引项会被排除，但仍然必须扫描这些项。原则上，这个索引也可以用于只对 b 和/或 c 有约束、而对 a 没有约束的查询，但必须扫描整个索引，因此在大多数情况下，规划器会倾向于顺序扫描表，而不是使用该索引。

多列 GiST 索引可以用于涉及索引任意列子集的查询条件。附加列上的条件会限制索引返回的项，但决定索引需要扫描多少内容的，最重要的仍是第一列上的条件。如果第一列只有很少几个非重复值，即使其他列有很多非重复值，GiST 索引也会相对低效。

多列 GIN 索引可以用于涉及索引任意列子集的查询条件。与 B-树或 GiST 不同，无论查询条件使用的是哪些索引列，GIN 的索引搜索效果都一样。

多列 BRIN 索引可以用于涉及索引任意列子集的查询条件。和 GIN 一样、不同于 B-树或 GiST，无论查询条件使用的是哪些索引列，索引搜索效果都一样。在单个表上使用多个 BRIN 索引，而不是使用一个多列 BRIN 索引的唯一理由，是需要不同的pages_per_range 存储参数。

当然，每一列都必须配合适合该索引类型的操作符使用；涉及其他操作符的子句不会被考虑。

多列索引应谨慎使用。在大多数情况下，单列索引已经足够，而且更省空间、也更省时间。除非表的使用方式极其程式化，否则超过三列的索引通常不会有帮助。关于不同索引配置优缺点的讨论，还可参见第 11.5 节和第 11.9 节。

11.4. 索引和 ORDER BY

除了简单地找到查询要返回的行之外，索引还可能能够以某种特定排序顺序交付这些行。这使得查询中的ORDER BY 要求无需额外排序步骤即可满足。在 PostgreSQL 当前支持的索引类型中，只有 B-树能产生有序输出，其他索引类型返回匹配行的顺序则未指定，并且依赖具体实现。

规划器在满足 ORDER BY 要求时，会考虑两种方案：扫描一个与该要求匹配的可用索引，或者按物理顺序扫描表再显式排序。对于需要扫描表中很大一部分内容的查询，显式排序通常会比使用索引更快，因为它遵循顺序访问模式，需要的磁盘 I/O 更少。只需取出少量行时，索引更有价值。一个重要的特例是ORDER BY 与 LIMIT *n* 组合使用：显式排序必须处理所有数据才能找出最前面的 *n* 行，而如果有一个与ORDER BY 匹配的索引，就可以直接取回前*n* 行，而完全不必扫描剩余数据。

默认情况下，B-树索引按升序存储其项，并将空值放在最后。这意味着，对列x 上的索引进行前向扫描，会产生满足ORDER BY x 的输出（更完整地说，是ORDER BY x ASC NULLS LAST）。索引也可以反向扫描，从而产生满足ORDER BY x DESC 的输出（更完整地说，是 ORDER BY x DESC NULLS FIRST，因为NULLS FIRST 是 ORDER BY DESC 的默认行为）。

你可以在创建 B-树索引时通过指定 ASC、DESC、NULLS FIRST 和/或 NULLS LAST 选项来调整索引的排序方式，例如：

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

一个按升序且空值在前存储的索引，根据扫描方向不同，可以满足ORDER BY x ASC NULLS FIRST 或 ORDER BY x DESC NULLS LAST。

你可能会疑惑，既然借助两个选项再加上反向扫描的可能性就能覆盖所有ORDER BY 变体，为什么还要提供四个选项。对于单列索引，这些选项确实是冗余的，但对于多列索引它们就可能很有用。考虑一个(x, y) 上的两列索引：前向扫描时它能满足 ORDER BY x, y，反向扫描时能满足ORDER BY x DESC, y DESC。但应用也可能经常需要 ORDER BY x ASC, y DESC。普通索引无法产生这种顺序，而如果把索引定义成(x ASC, y DESC) 或(x DESC, y ASC)，就可以做到。

显然，具有非默认排序方式的索引是一项相当专门的特性，但在某些查询上它们有时能带来巨大的加速。是否值得维护这样的索引，取决于你需要这种特殊排序的查询出现得有多频繁。

11.5. 组合多个索引

单次索引扫描只能利用那些在索引列上使用了该索引操作符类中操作符、并且彼此以AND 连接的查询子句。例如，给定一个(a, b) 上的索引，查询条件WHERE a = 5 AND b = 6 可以使用该索引，而WHERE a = 5 OR b = 6 这样的查询就不能直接使用该索引。

幸运的是，PostgreSQL 能够组合多个索引（包括同一个索引的多次使用），以处理单次索引扫描无法实现的情况。系统可以在多个索引扫描之间形成AND 和 OR 条件。例如，WHERE x = 42 OR x = 47 OR x = 53 OR x = 99 这样的查询，可以拆分成对x 上的一个索引进行四次独立扫描，每次扫描使用一个查询子句。然后将这些扫描的结果做 OR 运算以得到最终结果。另一个例子是，如果我们在x 和 y 上分别有索引，那么WHERE x = 5 AND y = 6 这样的查询，其一种可能实现方式就是分别用两个索引处理相应条件，然后将索引结果做 AND 运算，以确定结果行。

为了组合多个索引，系统会扫描每个所需索引，并在内存中准备一个位图，给出满足该索引条件的表行位置。然后按查询需要对这些位图进行 AND 和 OR 运算。最后，再访问并返回实际的表行。表行是按物理顺序访问的，因为位图就是这样布局的；这意味着原始索引中的任何排序都会丢失，所以如果查询带有ORDER BY 子句，就需要单独的排序步骤。也正因为如此，再加上每多一次索引扫描都会增加额外时间，规划器有时会选择使用简单的单一索引扫描，即使还有其他本来也可以利用的索引。

除了最简单的应用之外，通常会有多种可能有用的索引组合，数据库开发者必须权衡决定提供哪些索引。有时多列索引最好，但有时创建独立索引并依赖索引组合功能会更合适。例如，如果你的工作负载包含一组查询：有时只涉及列x，有时只涉及列y，有时同时涉及两列，那么你可以选择分别在x 和 y 上创建两个独立索引，并依赖索引组合来处理同时使用两列的查询。你也可以创建一个(x, y) 上的多列索引。对于同时涉及两列的查询，这个索引通常会比索引组合更高效，但正如第 11.3 节中所讨论的，它对于只涉及y 的查询几乎没有用处，因此不应只创建这一个索引。将这个多列索引与一个单独的y 索引组合起来，能够取得相当好的效果。对于只涉及x 的查询，多列索引也可以使用，但它会比单独的x 索引更大，因此更慢。最后一种选择是同时创建这三个索引，但这大概只有在该表被搜索的频率远高于被更新的频率，并且三类查询都很常见时才合理。如果其中一种查询远没有另外两种常见，那么你大概只需创建最适合常见查询类型的两个索引即可。

11.6. 唯一索引

索引还可以用于强制列值唯一，或者强制多个列组合值唯一。

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

当前，只有 B-树索引可以声明为唯一。

当一个索引被声明为唯一时，就不允许存在多个具有相同索引值的表行。空值不被视为相等。多列唯一索引只会拒绝那种在多行中所有索引列都相等的情况。

当为表定义唯一约束或主键时，PostgreSQL 会自动创建一个唯一索引。该索引覆盖构成主键或唯一约束的列（必要时是多列索引），并且正是它在强制执行该约束。

> **注意**
>
> 无需手工在唯一列上创建索引；那样做只会重复自动创建的索引。

11.7. 表达式索引

索引列不一定非得是底层表中的某一列，也可以是一个由表中一列或多列计算得到的函数或标量表达式。这个特性有助于基于计算结果来快速访问表。

例如，进行大小写不敏感比较的一种常见方式，是使用lower 函数：

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

如果为 lower(col1) 函数的结果定义了索引，这个查询就可以使用该索引：

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

如果我们把这个索引声明为 UNIQUE，那么它不仅会阻止创建那些col1 值仅在大小写上不同的行，也会阻止创建col1 值完全相同的行。因此，表达式索引可用于强制那些无法定义为简单唯一约束的约束。

再例如，如果你经常执行如下查询：

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

那么创建一个如下索引可能是值得的：

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

正如第二个示例所示，`CREATE INDEX` 命令的语法通常要求对索引表达式写上圆括号。当表达式只是一个函数调用时，则可以像第一个示例那样省略这些圆括号。

维护索引表达式的代价相对较高，因为每次插入一行以及每次非 HOT 更新都必须计算派生表达式。不过，在使用索引搜索时，索引表达式不会被重新计算，因为它们已经存储在索引中。在上面的两个示例里，系统看到的查询只是`WHERE indexedcolumn = 'constant'`，因此搜索速度与其他简单索引查询相同。因此，当检索速度比插入和更新速度更重要时，表达式索引会很有用。

11.8. 部分索引

部分索引是建立在表的一个子集上的索引；这个子集由一个条件表达式定义（称为部分索引的谓词）。索引中只包含满足该谓词的表行的项。部分索引是一种专门特性，但有几种场景下它会很有用。

使用部分索引的一个主要原因是避免索引常见值。由于搜索常见值（即占全部表行百分之几以上的值）的查询反正也不会使用索引，因此完全没有必要把这些行保留在索引里。这会减小索引尺寸，从而加快那些确实会使用该索引的查询。它也会加快很多表更新操作，因为索引并不需要在所有情况下都更新。例 11.1 展示了这种思路的一种可能应用。

例 11.1. 建立一个部分索引以排除常见值

假设你把 Web 服务器访问日志存储在数据库中。大多数访问来自你所在组织的 IP 地址范围，但也有一些来自其他地方（例如使用拨号连接的员工）。如果你按 IP 搜索时主要关心外部访问，那么你大概没有必要为对应于组织内子网的 IP 范围建索引。

假设有一个如下表：

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

要创建一个适合这个例子的部分索引，可使用如下命令：

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

一个能够使用此索引的典型查询如下：

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

以下查询无法使用此索引：

```
SELECT *
FROM access_log
WHERE client_ip = inet '192.168.100.23';
```

请注意，这类部分索引要求常见值事先可知，因此最适合用于数据分布不会变化的场景。也可以偶尔重建这类索引以适应新的数据分布，但那会增加维护工作量。

部分索引的另一种可能用途，是把典型查询工作负载不感兴趣的值排除在索引之外，如例 11.2 所示。这样会得到与上面相同的好处，但也意味着这些“不感兴趣”的值无法通过该索引访问，即使在那种情况下索引扫描可能是有利的。显然，为这种场景建立部分索引需要大量的谨慎和实验。

例 11.2. 建立一个部分索引以排除不感兴趣的值

如果你有一张表，其中同时包含已开账单和未开账单的订单，而未开账单订单只占整张表的一小部分，却是最常被访问的那些行，那么你可以通过只为未开账单的行创建索引来提升性能。创建该索引的命令如下：

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

一个可能会使用该索引的查询是：

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

不过，这个索引也可以用于那些完全不涉及`order_nr`的查询，例如：

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

这不像在 `amount` 列上建立部分索引那样高效，因为系统必须扫描整个索引。不过，如果未开账单的订单相对较少，那么即使只是用这个部分索引来找出未开账单订单，也可能是值得的。

请注意，下面这个查询不能使用该索引：

```
SELECT * FROM orders WHERE order_nr = 3501;
```

因为订单 3501 可能属于已开账单订单，也可能属于未开账单订单。

例 11.2 也说明了：索引列和谓词中使用的列不必一致。PostgreSQL 支持带任意谓词的部分索引，只要其中只涉及正在建立索引的那张表的列。不过要记住，谓词必须与那些希望从该索引受益的查询中使用的条件相匹配。更准确地说，只有当系统能够识别出查询的`WHERE` 条件在数学上蕴含该索引的谓词时，部分索引才能用于该查询。PostgreSQL 并没有一个复杂的定理证明器，来识别那些写法不同但数学上等价的表达式。（不仅构建这样一个通用定理证明器极其困难，而

且它很可能也会慢到失去实际用途。）系统可以识别简单的不等式蕴含，例如“x < 1”蕴含“x < 2”；否则，谓词条件必须与查询WHERE 条件的某一部分完全匹配，否则索引不会被识别为可用。匹配发生在查询规划阶段，而不是运行时。因此，参数化查询子句无法与部分索引配合工作。例如，一个带参数的预备查询可能写成“x < ?”，而它无法保证在参数的所有可能取值下都蕴含“x < 2”。

部分索引的第三种可能用途，甚至不要求索引被查询使用。这里的思路是像例 11.3 那样，在表的一个子集上创建唯一索引。这样就能在满足索引谓词的那些行之间强制唯一性，而不会约束不满足谓词的行。

例 11.3. 建立一个部分唯一索引

假设我们有一张描述测试结果的表。我们希望确保对于给定的测试对象和目标组合，只有一条“成功”记录，但可以有任意多条“失败”记录。实现方法之一如下：

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

当成功测试很少而失败测试很多时，这是一种特别高效的方法。也可以通过创建一个带IS NULL 限制的唯一部分索引，来让某一列只允许出现一个空值。

最后，部分索引还可以用来影响系统的查询计划选择。同样，分布异常的数据集可能导致系统在实际上不应该使用索引的时候选择使用它。在这种情况下，可以把索引建成对那个有问题的查询不可用。通常，PostgreSQL 会对索引使用作出合理选择（例如，它会在检索常见值时避免使用索引，因此前面的例子实际上只是节省索引空间，而不是为了避免使用索引所必需的），如果出现明显错误的计划选择，那就应当提交 bug 报告。

请记住，建立部分索引意味着你至少与查询规划器一样了解情况，尤其是你知道索引何时可能是有利的。形成这种认识需要经验，以及对PostgreSQL 中索引工作方式的理解。在大多数情况下，部分索引相比普通索引的优势都很小。有些场景下它们甚至会适得其反，如例 11.4 所示。

例 11.4. 不要把部分索引当作分区的替代品

你可能会想创建一大组互不重叠的部分索引，例如

```
CREATE INDEX mytable_cat_1 ON mytable (data) WHERE category = 1;
CREATE INDEX mytable_cat_2 ON mytable (data) WHERE category = 2;
CREATE INDEX mytable_cat_3 ON mytable (data) WHERE category = 3;
...
CREATE INDEX mytable_cat_N ON mytable (data) WHERE category = N;
```

这是个坏主意！几乎可以肯定，你最好建立一个单独的非部分索引，例如：

```
CREATE INDEX mytable_cat_data ON mytable (category, data);
```

（把 category 列放在前面，原因见第 11.3 节。）虽然在这个较大的索引中搜索，可能要比在较小索引中搜索多下降几层树，但这几乎肯定仍然比规划器为了从这些部分索引中选出合适的一个所付出的代价更低。问题的核心在于，系统并不了解这些部分索引之间的关系，因此会费力地逐个测试它们，看它们是否适用于当前查询。

如果你的表大到单个索引确实不是好主意，那么你应该考虑改用分区（见第 5.10 节）。借助这种机制，系统能够理解这些表和索引彼此不重叠，因此有可能获得好得多的性能。

关于部分索引的更多信息可参见[ston89b]、[olson93] 和[seshadri95]。

11.9. 仅索引扫描和覆盖索引

PostgreSQL 中的所有索引都是二级索引，也就是说，每个索引都与表的主数据区分开存储（在 PostgreSQL 术语中，这个主数据区称为表的堆）。这意味着，在普通索引扫描中，每次取回一行都需要同时从索引和堆中取数据。此外，尽管满足某个可索引`WHERE`条件的索引项通常在索引中彼此接近，但它们引用的表行却可能分布在堆中的任何位置。因此，索引扫描的堆访问部分会涉及大量对堆的随机访问，这可能很慢，尤其是在传统旋转介质上。（正如第 11.5 节中所述，位图扫描试图通过按排序顺序进行堆访问来缓解这项代价，但那也只能缓解到一定程度。）

为了解决这个性能问题，PostgreSQL 支持仅索引扫描，它可以仅凭索引而不访问堆来回答查询。基本思路是直接从每个索引项中返回值，而不是再去查对应的堆项。要使用这种方法，有两个根本限制：

1. 索引类型必须支持仅索引扫描。B-树索引总是支持。GiST 和 SP-GiST 索引对某些操作符类支持仅索引扫描，但对另一些则不支持。其他索引类型则完全不支持。底层要求是，索引必须实际存储原始数据值，或者至少能够重建出每个索引项对应的原始数据值。反例是 GIN 索引，它不能支持仅索引扫描，因为每个索引项通常只保存原始数据值的一部分。

2. 查询只能引用存储在索引中的列。例如，假设某个表的x 和 y 列上有一个索引，且该表还有一列z，那么下面这些查询可以使用仅索引扫描：

```
SELECT x, y FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND y < 42;
```

   但下面这些查询则不能：

```
SELECT x, z FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND z < 42;
```

   （表达式索引和部分索引会让这条规则变得更复杂，下文会讨论。）

如果这两个基本要求满足，那么查询所需的所有数据值都能从索引中取得，因此从物理上说仅索引扫描是可行的。不过，在PostgreSQL 中，任何表扫描还有一个额外要求：它必须验证每个取回的行对该查询的 MVCC 快照是否“可见”，如第 13 章所述。可见性信息并不保存在索引项中，而只保存在堆项中；因此乍看之下，似乎每次取回行无论如何都要访问堆。这在表行最近被修改过时的确如此。然而，对于很少变化的数据，这个问题有办法绕开。PostgreSQL 会跟踪表堆中每个页面是否其中所有行都已经足够老，以至于对当前和未来所有事务都可见。这个信息保存在该表的可见性映射中的一个位里。仅索引扫描在找到候选索引项后，会检查对应堆页面的可见性映射位。如果该位已设置，那么这行就已知可见，数据可以直接返回而无需进一步工作。如果没有设置，就必须访问堆项来判断该行是否可见，这样相对标准索引扫描就没有性能优势。即使在成功的情况下，这种做法也是用访问可见性映射来替代访问堆；但由于可见性映射比它描述的堆小四个数量级，访问它所需的物理 I/O 要少得多。在大多数场景下，可见性映射始终都会缓存于内存中。

简而言之，尽管满足那两个基本要求时就有可能使用仅索引扫描，但只有当表中相当一部分堆页的“全部可见”（all-visible）映射位已被设置时，它才会带来收益。不过，很多表都会有相当大一部分行长期不变，因此这种扫描方式在实践中非常有用。

为了有效利用仅索引扫描特性，你可以选择创建一个覆盖索引，也就是专门设计来包含某一类高频查询所需列的索引。由于查询通常需要取回的列不止是用于搜索的那些列，PostgreSQL 允许你创建一种索引，其中某些列只是“负载”，而不是搜索键的一部分。这是通过添加一个 INCLUDE 子句列出这些额外列来实现的。例如，如果你经常运行如下查询：

```
SELECT y FROM tab WHERE x = 'key';
```

传统上，为了加速这类查询，通常会只在x 上创建一个索引。但如果将索引定义为：

```
CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

那么这些查询就可以以仅索引扫描的方式完成，因为y 可以直接从索引中取得，而不必访问堆。

由于列 y 不是索引搜索键的一部分，因此它不必属于该索引能够处理的数据类型；它只是被存储在索引中，而不会被索引机制解释。此外，如果该索引是唯一索引，即

```
CREATE UNIQUE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

那么唯一性条件只作用于列 x，而不作用于x 和 y 的组合。（INCLUDE 子句也可以写在 UNIQUE 和 PRIMARY KEY 约束中，作为建立这种索引的另一种语法。）

向索引中添加非键负载列时，明智的做法是保持保守，尤其是宽列。如果某个索引元组超过该索引类型允许的最大尺寸，数据插入就会失败。无论如何，非键列都会复制索引对应表中的数据并让索引膨胀，因此可能拖慢搜索。另外请记住，除非表变化足够缓慢，以至于仅索引扫描很可能无需访问堆，否则把负载列包含在索引中并没有太大意义。如果反正必须访问堆元组，那么顺便从那里取出该列的值也不会增加额外代价。其他限制包括：当前不支持把表达式作为内含列；并且当前只有 B-树索引支持内含列。

在 PostgreSQL 支持 `INCLUDE` 功能之前，人们有时会将负载列写成普通索引列来创建覆盖索引，也就是写成

```
CREATE INDEX tab_x_y ON tab(x, y);
```

尽管他们从未打算将 `y` 用作 `WHERE` 子句的一部分。只要这些额外列位于末尾，这样做就没有问题；将它们放在前面是不明智的，原因见第 11.3 节。但是，如果希望索引强制键列唯一，这种方法就不适用了。另外，将不参与搜索的列明确标记为`INCLUDE` 列，还能使索引稍小一些，因为这些列不需要存储在 B-树的上层。

原则上，仅索引扫描也可以和表达式索引一起使用。例如，给定一个`f(x)` 上的索引，其中`x` 是表的一列，那么按理说应该可以把

```
SELECT f(x) FROM tab WHERE f(x) < 1;
```

执行成一次仅索引扫描；如果 `f()` 是一个计算代价很高的函数，这会非常有吸引力。不过，PostgreSQL 的规划器目前在这种情况上还不够聪明。它只会在查询所需的所有列都能从索引取得时，才认为查询可能通过仅索引扫描执行。在这个例子里，除了在`f(x)` 这个上下文中，`x` 本身并不需要，但规划器意识不到这一点，因此得出无法做仅索引扫描的结论。如果仅索引扫描看起来足够值得，可以通过把`x` 作为内含列加进去来绕过这一点，例如：

```
CREATE INDEX tab_f_x ON tab (f(x)) INCLUDE (x);
```

还有一个额外注意事项：如果目标是避免重新计算 `f(x)`，那么规划器不一定会把那些不在可索引`WHERE` 子句中的 `f(x)` 用法与索引列匹配起来。对于上面展示的简单查询，它通常能做对，但对于涉及连接的查询则未必。未来版本的PostgreSQL 可能会修复这些不足。

部分索引与仅索引扫描之间也有有趣的相互作用。考虑例 11.3 中展示的这个部分索引：

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

原则上，我们可以在这个索引上做仅索引扫描，以满足如下查询：

```
SELECT target FROM tests WHERE subject = 'some-subject' AND success;
```

但这里有个问题：`WHERE` 子句引用了 `success`，而它并不能作为索引的结果列取得。尽管如此，仍然可能做仅索引扫描，因为执行计划在运行时不需要重新检查`WHERE` 子句的这一部分：索引中找到的所有项都必然满足 `success = true`，因此计划里无需显式检查它。PostgreSQL 9.6 及更高版本能够识别这种情况，并允许生成仅索引扫描；更早的版本则不能。

11.10. 操作符类和操作符族

一个索引定义可以为索引的每一列指定一个操作符类。

```
CREATE INDEX name ON table (column opclass [ ( opclass_options ) ] [sort options] [, ...]);
```

操作符类标识该索引在该列上要使用哪些操作符。例如，类型int4 上的一个 B-树索引会使用 int4_ops 类；这个操作符类包含对 int4 值进行比较所需的函数。实践中，列数据类型的默认操作符类通常已经足够。之所以存在操作符类，主要是因为对某些数据类型来说，可能存在多种有意义的索引行为。例如，我们可能希望按绝对值排序一种复数数据类型，也可能希望按实部排序。我们可以通过为该数据类型定义两个操作符类，并在创建索引时选择合适的那个来做到这一点。操作符类决定了基础排序顺序（之后还可以通过添加排序选项COLLATE、ASC/DESC 和/或 NULLS FIRST/NULLS LAST 来修改）。

除了默认操作符类之外，还有一些内置操作符类：

- 操作符类 text_pattern_ops、varchar_pattern_ops 和 bpchar_pattern_ops 分别支持类型 text、varchar 和 char 上的 B-树索引。它们与默认操作符类的区别在于，值是严格按字符逐个比较，而不是按照区域设置相关的排序规则比较。因此，当数据库未使用标准 “C” 区域设置时，这些操作符类适用于涉及模式匹配表达式（LIKE 或 POSIX 正则表达式）的查询。例如，你可以像这样为一个varchar 列建立索引：

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  注意，如果你希望涉及普通<、<=、> 或>=比较的查询也能使用索引，那么还应该再创建一个使用默认操作符类的索引。这类查询不能使用*xxx*_pattern_ops 操作符类。（不过普通等值比较是可以使用这些操作符类的。）同一列上可以创建多个使用不同操作符类的索引。如果你使用的是 C 区域设置，则不需要*xxx*_pattern_ops 操作符类，因为带默认操作符类的索引在 C 区域设置下就可以用于模式匹配查询。

下面的查询会显示所有已定义的操作符类：

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

操作符类其实只是一个更大结构的子集，这个结构称为操作符族。当若干数据类型具有相似行为时，定义跨数据类型操作符并让索引支持它们，往往会很有用。为此，每种类型对应的操作符类都必须归入同一个操作符族。跨类型操作符属于该族，但不与该族中任何单独一个类关联。

前一个查询的扩展版本会显示每个操作符类所属的操作符族：

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opf.opfname AS opfamily_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc, pg_opfamily opf
    WHERE opc.opcmethod = am.oid AND
          opc.opcfamily = opf.oid
    ORDER BY index_method, opclass_name;
```

下面这个查询会显示所有已定义的操作符族，以及每个族中包含的全部操作符：

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
          amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

11.11. 索引和排序规则

每个索引列只能支持一种排序规则。如果你需要关心多种排序规则，就可能需要多个索引。

考虑以下语句：

```
CREATE TABLE test1c (
    id integer,
    content varchar COLLATE "x"
);

CREATE INDEX test1c_content_index ON test1c (content);
```

该索引会自动使用底层列的排序规则。因此，下面这种形式的查询：

```
SELECT * FROM test1c WHERE content > constant;
```

可以使用该索引，因为比较默认会使用该列的排序规则。不过，这个索引不能加速涉及其他排序规则的查询。因此，如果你也关心如下形式的查询：

```
SELECT * FROM test1c WHERE content > constant COLLATE "y";
```

那么可以额外创建一个支持"y"排序规则的索引，例如：

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE "y");
```

11.12. 检查索引使用情况

尽管 PostgreSQL 中的索引不需要维护或调优，但检查真实查询工作负载到底实际使用了哪些索引，仍然很重要。检查某个单独查询的索引使用情况，可以使用EXPLAIN 命令；它在这方面的应用见第 14.1 节。也可以像第 28.2 节中描述的那样，在运行中的服务器上收集索引使用的总体统计信息。

很难给出一个确定应该创建哪些索引的通用流程。前面各节的示例已经展示了若干典型情形。通常需要做大量实验。下面这部分会给出一些相关提示：

- 一定要先运行 ANALYZE。这个命令会收集表中值分布的统计信息。估计查询会返回多少行，需要这些信息；而规划器要为每种可能的查询计划分配现实的代价，也需要这些信息。如果没有真实统计信息，系统就会假定一些默认值，而这些默认值几乎肯定并不准确。因此，在没有运行`ANALYZE` 的情况下去检查应用的索引使用情况，基本上是徒劳的。更多信息见第 24.1.3 节和第 24.1.6 节。

- 用真实数据做实验。使用测试数据来设置索引，只会告诉你测试数据需要什么索引，仅此而已。

  特别要避免使用非常小的测试数据集。虽然在 100000 行里选 1000 行可能适合用索引，但在 100 行里选 1 行几乎不适合，因为这 100 行大概都能放进一个磁盘页，而没有任何计划能比顺序取一个磁盘页更快。

  另外，在编造测试数据时也要小心；如果应用尚未投入生产，这往往无法避免。非常相似、完全随机、或者按排序顺序插入的值，都会让统计信息偏离真实数据应有的分布。

- 当索引没有被使用时，强制使用它们对测试会很有帮助。有一些运行时参数可以关闭不同的计划类型（见第 19.7.1 节）。例如，关闭顺序扫描（`enable_seqscan`）和嵌套循环连接（`enable_nestloop`）这两种最基本的计划，就会迫使系统使用不同的计划。如果系统仍然选择顺序扫描或嵌套循环连接，那么索引未被使用很可能有更根本的原因，例如查询条件并不匹配该索引。（前面各节已经解释了什么样的查询可以使用什么样的索引。）

- 如果强制使用索引后确实使用了索引，那么就有两种可能：要么系统是对的，使用索引确实不合适；要么查询计划的代价估计并未反映现实。因此，你应该分别在有索引和无索引的情况下对查询计时。`EXPLAIN ANALYZE` 命令在这里会很有帮助。

- 如果发现代价估计是错的，那么又有两种可能。总代价是根据每个计划节点的逐行代价乘以该计划节点的选择率估计计算出来的。计划节点的估计代价可以通过运行时参数来调整（见第 19.7.2 节）。而选择率估计不准确，则是因为统计信息不足。可以尝试通过调整统计信息收集参数来改进这一点（见 ALTER TABLE）。

  如果你无法把这些代价调得更合适，那么可能就不得不退而求其次，显式强制使用索引。你也可能希望联系PostgreSQL 开发者来研究这个问题。

第 12 章 全文检索

12.1. 简介

全文检索（或简称文本检索）提供了识别满足查询条件的自然语言文档的能力，并且可按它们与查询的相关度进行排序。最常见的搜索类型，是找出所有包含给定查询词的文档，并按它们与查询的相似性排序返回。查询和相似性的概念都很灵活，取决于具体应用。最简单的搜索把查询视为一组词，把相似性视为查询词在文档中的出现频率。

文本检索操作符在数据库中已经存在很多年了。PostgreSQL 为文本数据类型提供了 `~`、`~*`、`LIKE` 和 `ILIKE` 操作符，但它们缺少现代信息系统所要求的许多关键特性：

- 缺少语言学支持，即便对英语也是如此。正则表达式并不足够，因为它们难以轻松处理词形变化，例如 `satisfies` 和 `satisfy`。搜索 `satisfy` 时，你大概也希望找到包含 `satisfies` 的文档，但实际上可能会漏掉它们。虽然可以用 `OR` 搜索多个派生形式，但这样既繁琐又容易出错（有些词甚至可能有数千种派生形式）。
- 它们不会对搜索结果排序（排名），因此当找到成千上万条匹配文档时就难以实用。
- 它们往往很慢，因为没有索引支持，所以每次搜索都必须处理全部文档。

全文索引允许先对文档进行预处理，并保存索引以供后续快速搜索。预处理包括：

将文档解析成词元。识别出不同类别的词元很有帮助，例如数字、单词、复合词、电子邮件地址等，这样就可以分别处理。原则上，词元类别取决于具体应用，但在大多数场景中，一套预定义的类别已经足够。PostgreSQL 使用解析器执行这一步。系统提供了标准解析器，也可以根据特定需求创建自定义解析器。

将词元转换成词位。词位和词元一样都是字符串，但它已经过正规化，使同一个词的不同形式归为一致。例如，正规化几乎总是包含把大写字母折叠为小写，也常常会去掉后缀（如英语中的 `s` 或 `es`）。这样，搜索时无需繁琐地输入所有可能的变体，就能匹配同一个词的不同形式。此外，这一步通常还会去除停用词，也就是那些过于常见、对搜索没有帮助的词。（简言之，词元是文档文本的原始片段，而词位则是被认为适合用于索引和搜索的词。）PostgreSQL 使用词典执行这一步。系统提供了多种标准词典，也可以按需创建自定义词典。

以适合搜索的形式存储预处理后的文档。例如，每个文档都可以表示为由正规化词位组成的有序数组。除词位之外，通常还希望存储位置信息，以便用于邻近排名；这样，查询词分布更“密集”的文档会比查询词分散出现的文档得到更高的排名。

词典允许对词元的正规化方式进行细粒度控制。借助合适的词典，你可以：

- 定义不应该被索引的停用词。
- 使用 Ispell 把同义词映射为同一个词。
- 使用分类词典把短语映射为同一个词。
- 使用 Ispell 词典把一个词的不同变体映射到一种规范形式。
- 使用 Snowball 词干分析规则把一个词的不同变体映射到一种规范形式。

系统提供了数据类型 `tsvector` 用于存储预处理后的文档，也提供了数据类型 `tsquery` 用于表示处理过的查询（第 8.11 节）。围绕这两种数据类型还有许多函数和操作符（第 9.13 节），其中最重要的是匹配操作符 `@@`，我们将在第 12.1.2 节中介绍。全文检索还可以借助索引加速（第 12.9 节）。

12.1.1. 什么是文档?

文档是全文检索系统中的搜索单位，例如一篇杂志文章或一封电子邮件。文本检索引擎必须能够解析文档，并保存词位（关键字）与其所属文档之间的关联。随后，就可以利用这些关联来搜索包含查询词的文档。

在 PostgreSQL 中进行搜索时，文档通常是数据库表某一行中的一个文本字段，或者是这类字段的组合（串接），这些字段可能分布在多个表里，也可能是动态取得的。换句话说，文档可以由不同部分拼接而成以供索引，而且未必会以一个整体存储在任何地方。例如：

```
SELECT title || ' ' ||  author || ' ' ||  abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document
FROM messages m, docs d
WHERE m.mid = d.did AND m.mid = 12;
```

> **注意**
>
> 实际上，这些示例查询中应使用 `coalesce`，以避免某个单独的 `NULL` 属性导致整个文档结果变成 `NULL`。

另一种做法是把文档存储为文件系统中的普通文本文件。在这种情况下，数据库可以用于保存全文索引并执行搜索，而文档本身则借助某个唯一标识符从文件系统中取回。不过，从数据库外部检索文件需要超级用户权限或特殊函数支持，因此这种方法通常不如把所有数据都保存在 PostgreSQL 内部方便。此外，把所有内容都放在数据库中，也便于访问文档元数据来辅助索引和展示。

为了进行文本检索，每个文档都必须被化简为预处理后的 `tsvector` 格式。搜索和排名完全基于文档的 `tsvector` 表示来执行 – 只有当文档被选中并准备展示给用户时，才需要取回原始文本。因此，我们常常把 `tsvector` 直接称作文档，但它当然只是完整文档的一种紧凑表示。

12.1.2. 基本文本匹配

在 PostgreSQL 中，全文检索基于匹配操作符`@@`。如果一个 `tsvector`（文档）匹配一个 `tsquery`（查询），它就返回`true`。哪一种数据类型写在前面并不重要：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;
 ?column?
----------
 f
```

正如上例所示，tsquery 并不只是原始文本，tsvector 也不是。tsquery 包含搜索词，这些搜索词必须已经是正规化后的词位，并且可以用 AND、OR、NOT 和 FOLLOWED BY 操作符把多个搜索词组合起来。（语法细节见第 8.11.2 节。）to_tsquery、plainto_tsquery 和 phraseto_tsquery 有助于把用户输入的文本转换为合适的 tsquery，其主要工作就是对文本中的词做正规化。类似地，to_tsvector 用于解析并正规化文档字符串。因此在实践中，文本检索匹配更像是这样：

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
 ?column?
----------
 t
```

请注意，如果写成

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

就不会匹配成功，因为这里不会对单词 rats 做正规化。tsvector 的元素是词位，被假定为已经正规化，因此 rats 不会匹配 rat。

@@操作符也支持 text 输入，因此在简单场景下可以跳过把文本字符串显式转换为tsvector 或 tsquery。可用的形式有：

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

前两种形式我们已经见过。text @@ tsquery 等价于 to_tsvector(x) @@ y。text @@ text 等价于 to_tsvector(x) @@ plainto_tsquery(y)。

在 tsquery 中，&（AND）操作符指定它的两个参数都必须出现在文档中才表示匹配。类似地，|（OR）操作符指定至少一个参数必须出现，而!（NOT）操作符指定它的参数不出现才能匹配。例如，查询fat & ! rat 匹配包含 fat 但不包含 rat 的文档。

借助<->（FOLLOWED BY）tsquery 操作符，也可以搜索短语。只有当它的参数在文档中有相邻且顺序符合要求的匹配时，查询才算匹配。例如：

```
SELECT to_tsvector('fatal error') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 t

SELECT to_tsvector('error is not fatal') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 f
```

FOLLOWED BY 操作符还有一个更一般的形式 <*N*>，其中 *N* 是表示匹配词位位置差的整数。<1> 与 <-> 相同，而 <2> 允许两个匹配之间恰好出现一个其他词位，依此类推。phraseto_tsquery 函数利用这种操作符构造 tsquery，从而在某些词是停用词时仍能匹配多词短语。例如：

```
SELECT phraseto_tsquery('cats ate rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <-> 'rat'

SELECT phraseto_tsquery('the cats ate the rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <2> 'rat'
```

有一个特殊情形有时很有用：<0> 可用于要求两个模式匹配同一个词。

圆括号可以被用来控制 tsquery 操作符的嵌套。如果没有圆括号，|的计算优先级最低，然后从低到高依次是&、<->、!。

值得注意的是，当 AND/OR/NOT 操作符位于 FOLLOWED BY 操作符的参数内部时，它们的语义与位于外部时会有细微差别，因为在 FOLLOWED BY 中，匹配的确切位置是有意义的。例如，通常 !x 只匹配完全不包含 x 的文档；但 !x <-> y 会在 y 前面没有紧邻一个 x 时匹配，文档其他位置出现的 x 并不会阻止匹配。再例如，x & y 通常只要求 x 和 y 都在文档某处出现，而 (x & y) <-> z 则要求 x 和 y 在同一位置匹配，并且紧挨在 z 之前。因此，这个查询的行为不同于 x <-> z & y <-> z，后者会匹配同时包含两个独立序列 x z 和 y z 的文档。（按字面写出时，这个特定查询其实没有什么用，因为 x 和 y 不可能在同一位置匹配；但在更复杂的场景中，例如前缀匹配模式，这种形式就可能有用。）

12.1.3. 配置

前述的都是简单的文本检索示例。正如前面所提到的，全文检索功能包括做更多事情的能力：跳过索引特定词（停用词）、处理同义词并使用更高级的解析，例如不局限于按空白划分的解析。这个功能由文本检索配置控制。PostgreSQL 中有多种语言的预定义配置，并且你可以很容易地创建你自己的配置（psql 的\dF 命令显示所有可用的配置）。

在安装期间会选择一个合适的配置，并据此在postgresql.conf 中设置 default_text_search_config。如果整个集簇都使用同一种文本检索配置，你可以直接使用 postgresql.conf 中的这个值。若要在整个集簇中使用不同配置，但保证每个数据库内部使用同一种配置，可以使用ALTER DATABASE ... SET。否则，你也可以在每个会话中设置 default_text_search_config。

依赖一个配置的每一个文本检索函数都有一个可选的 `regconfig` 参数，因此要使用的配置可以被显式指定。只有当省略这个参数时，`default_text_search_config` 才被使用。

为了便于构建自定义文本检索配置，配置由更简单的数据库对象组合而成。PostgreSQL 的文本检索功能提供了四类与配置相关的数据库对象：

- 文本检索解析器将文档拆分成词元并对每个词元分类（例如，作为词或者数字）。
- 文本检索词典将词元转变成正规化的形式并拒绝停用词。
- 文本检索模板提供位于词典底层的函数（词典只需指定一个模板及其一组参数）。
- 文本检索配置选择一个解析器和一组用于将解析器产生的词元正规化的词典。

文本检索解析器和模板由底层 C 函数构成，因此开发新的解析器和模板需要 C 编程能力，将它们安装到数据库中则需要超级用户权限。（PostgreSQL 发行版的 `contrib/` 目录中有附加解析器和模板的示例。）词典和配置只是为底层解析器和模板设置参数并将它们连接起来，因此创建新词典或配置不需要特殊权限。本章后面会给出创建自定义词典和配置的示例。

12.2. 表和索引

在前一节中的示例演示了使用简单常量字符串进行全文匹配。本节展示如何搜索表数据，以及可选择地使用索引。

12.2.1. 搜索表

即使没有索引，也可以执行全文检索。下面这个简单查询会输出每一行的`title`，其中对应的 `body` 字段包含单词 `friend`：

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

这还会找到相关词，例如`friends` 和 `friendly`，因为这些词都会被约简为同一个正规化词位。

上述查询指定使用 `english` 配置来解析并正规化字符串。我们也可以省略配置参数：

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

该查询将使用由 default_text_search_config 设置的配置。

更复杂一点的例子，是选出最近的 10 个文档，它们的 `title` 和 `body` 合在一起包含 `create` 和 `table`：

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

为简洁起见，这里省略了 `coalesce` 调用；如果希望在这两个字段之一为 `NULL` 时，另一字段仍能参与搜索，就需要加上它。

虽然这些查询在没有索引的情况下也能工作，但除偶尔的临时搜索外，大多数应用都会觉得这种方式太慢。文本检索在实际使用中通常都需要建立索引。

12.2.2. 创建索引

我们可以创建一个 GIN 索引（第 12.9 节）来加速文本检索：

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', body));
```

注意这里使用的是 to_tsvector 的双参数版本。只有显式指定配置名称的文本检索函数，才能用于表达式索引（第 11.7 节）。这是因为索引内容必须不受 default_text_search_config 的影响。否则，索引内容就可能不一致，因为不同的索引项可能包含使用不同文本检索配置创建的 tsvector，而且无法判断各自使用了哪一种配置。这样的索引也不可能被正确地转储和恢复。

由于上面的索引使用了 to_tsvector 的双参数版本，因此只有同样使用相同配置名的双参数版 to_tsvector 查询，才能使用该索引。也就是说，WHERE to_tsvector('english', body) @@ 'a & b' 可以使用该索引，而 WHERE to_tsvector(body) @@ 'a & b' 则不能。这样可以保证索引只会和创建索引项时所用的同一配置配合使用。

还可以建立更复杂的表达式索引，其中配置名由另一个列指定，例如：

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector(config_name, body));
```

这里 config_name 是 pgweb 表中的一个列。这样就允许在同一个索引中混合使用不同配置，同时记录每个索引项使用的是哪一种配置。例如，如果文档集合中包含不同语言的文档，这就会很有用。同样，打算使用该索引的查询也必须写成对应的形式，例如 WHERE to_tsvector(config_name, body) @@ 'a & b'。

索引甚至可以串接多个列：

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', title || ' ' || body));
```

另一种方法是创建一个单独的 tsvector 列来保存 to_tsvector 的输出。下面的示例把 title 和 body 串接起来，并用coalesce 保证一个字段仍然可以被建立索引，即使另一个字段为NULL：

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col =
     to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''));
```

然后我们创建一个 GIN 索引来加速搜索：

```
CREATE INDEX textsearch_idx ON pgweb USING GIN (textsearchable_index_col);
```

现在可以执行快速全文检索了：

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

当使用一个单独的列来存储 tsvector 表示时，需要创建一个触发器来使 tsvector 列保持最新，以应对 title 或 body 的任何更改。第 12.4.3 节说明了如何做到这一点。

与表达式索引相比，单独列方法的一个优点是，为了利用索引，查询中不必显式指定文本检索配置。正如上面的例子所示，查询可以依赖default_text_search_config。另一个优点是搜索会更快，因为它不必重新执行to_tsvector 调用来验证索引匹配（使用 GiST 索引时这一点比使用 GIN 索引时更重要；见第 12.9 节）。不过，表达式索引方法更容易设置，而且占用更少磁盘空间，因为tsvector 表示并没有被显式存储。

12.3. 控制文本检索

要实现全文检索，必须有函数能够从文档创建 tsvector，并从用户查询创建 tsquery。此外，我们还希望结果能按有意义的顺序返回，因此还需要函数根据文档与查询的相关性进行比较。同样重要的，是把结果良好地展示出来。PostgreSQL 为这些能力都提供了支持。

12.3.1. 解析文档

PostgreSQL 提供了 to_tsvector 函数，用于把文档转换成 tsvector 数据类型。

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

to_tsvector 会把文本文档解析为词元，将词元归约为词位，并返回一个 tsvector，其中列出各词位及其在文档中的位置。文档会按指定的或默认的文本检索配置进行处理。下面是一个简单示例：

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

在上面的示例中可以看到，结果 tsvector 不包含 a、on 和 it，单词 rats 变成了 rat，标点符号 - 也被忽略了。

to_tsvector 在内部会调用解析器，把文档文本拆分成词元并为每个词元分配类型。对于每个词元，系统都会查询一个词典列表（第 12.6 节），而这个列表会随词元类型而变化。第一个能够识别该词元的词典，会输出一个或多个正规化后的词位来表示它。例如，rats 之所以变成 rat，是因为某个词典识别出 rats 是 rat 的复数形式。某些词会被识别为停用词（第 12.6.1 节），于是被忽略，因为它们出现得过于频繁，对搜索没有帮助。在这个示例中，a、on 和 it 就是停用词。如果列表中的词典都无法识别某个词元，它也会被忽略。示例里的标点符号 - 就属于这种情况，因为其词元类型（Space symbols）实际上没有分配任何词典，也就是说，空白类词元永远不会被索引。解析器、词典以及需要索引哪些词元类型，都由所选的文本检索配置（第 12.7 节）决定。一个数据库里可以同时存在多种不同配置，并且系统已经为多种语言提供了预定义配置。在本例中，我们使用的是英语的默认配置 english。

setweight 函数可用于给 tsvector 中的项打上指定的权重标签，权重可以是四个字母之一：A、B、C 或 D。这通常用于标记来自文档不同部分的项，例如标题和正文。稍后，这些信息可以用于搜索结果排名。

由于 to_tsvector(NULL) 会返回 NULL，因此只要字段可能为 NULL，就建议使用 coalesce。下面是从结构化文档创建 tsvector 的推荐方法：

```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

这里我们使用 setweight 为最终 tsvector 中每个词位标注来源，然后再用 tsvector 连接操作符 || 把这些已标注的 tsvector 值合并起来（关于这些操作的细节，见第 12.4.1 节）。

12.3.2. 解析查询

PostgreSQL 提供了 to_tsquery、plainto_tsquery、phraseto_tsquery 以及 websearch_to_tsquery，用于把查询转换成 tsquery 数据类型。to_tsquery 提供的特性比 plainto_tsquery 和 phraseto_tsquery 更丰富，但对输入也更严格。websearch_to_tsquery 则是 to_tsquery 的简化版本，使用一种类似 Web 搜索引擎的替代语法。

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

to_tsquery 根据 *querytext* 创建一个 tsquery 值。输入必须由单个词元组成，这些词元之间使用 tsquery 操作符 & （AND）、| （OR）、! （NOT）和 <-> （FOLLOWED BY）分隔，并可使用圆括号分组。换句话说，传给 to_tsquery 的输入，必须已经遵循第 8.11.2 节中描述的 tsquery 一般输入规则。不同之处在于，基础 tsquery 输入是按字面接受词元，而 to_tsquery 会使用指定或默认配置，把每个词元正规化为词位，并丢弃那些按该配置被判定为停用词的词元。例如：

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

与基础 tsquery 输入一样，可以为每个词位附加权重，以限制它只匹配具有这些权重的 tsvector 词位。例如：

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
```

此外，还可以在词位后附加* 来指定前缀匹配：

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

这样的词位将匹配 tsvector 中任何以给定字符串开头的单词。

to_tsquery 也可以接受单引号括起来的短语。当配置中包含可能在这类短语上触发的分类词典时，这一点尤其有用。在下面的例子中，一个分类词典包含规则 supernovae stars : sn：

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

如果不加引号，对于那些未被 AND、OR 或 FOLLOWED BY 操作符分隔的词元，to_tsquery 会产生语法错误。

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

plainto_tsquery 把未格式化的文本 *querytext* 转换成一个 tsquery 值。该文本会像 to_tsvector 那样被解析并正规化，然后在保留下来的词之间插入& （AND）tsquery 操作符。

示例：

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

注意，plainto_tsquery 不会识别其输入中的 tsquery 操作符、权重标签或前缀匹配标签：

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

这里，输入中的所有标点符号都被当作空白符丢弃。

```
phraseto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

phraseto_tsquery 的行为很像 plainto_tsquery，不过它会在保留下来的词之间插入<->（FOLLOWED BY）操作符，而不是& （AND）操作符。此外，停用词也不是简单地丢弃，而是通过插入<*N*> 操作符（而不是<-> 操作符）来体现。在搜索精确词位序列时，这个函数很有用，因为 FOLLOWED BY 操作符不仅检查所有词位是否存在，还检查词位的顺序。

示例：

```
SELECT phraseto_tsquery('english', 'The Fat Rats');
 phraseto_tsquery
------------------
 'fat' <-> 'rat'
```

与 plainto_tsquery 一样，phraseto_tsquery 函数也不会识别输入中的 tsquery 操作符、权重标签或前缀匹配标签：

```
SELECT phraseto_tsquery('english', 'The Fat & Rats:C');
      phraseto_tsquery
-----------------------------
 'fat' <-> 'rat' <-> 'c'
```

```
websearch_to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

websearch_to_tsquery 创建一个 tsquery 值，其来源为*querytext*，采用一种替代语法，其中简单的无格式文本就是一个有效查询。不同于plainto_tsquery 和 phraseto_tsquery，它还识别某些操作符。此外，这个函数不应引发语法错误，因此可以直接使用用户提供的原始输入进行搜索。支持以下语法：

- 未加引号的文本：不在引号中的文本会被转换为由& 操作符分隔的词，就像经过 plainto_tsquery 处理一样。
- "加引号的文本"：引号中的文本会被转换为由<-> 操作符分隔的词，就像经过 phraseto_tsquery 处理一样。

- OR：逻辑或会转换为 | 操作符。
- -：逻辑非操作符，会转换为 ! 操作符。

示例：

```
SELECT websearch_to_tsquery('english', 'The fat rats');
 websearch_to_tsquery
----------------------
 'fat' & 'rat'
(1 row)

SELECT websearch_to_tsquery('english', '"supernovae stars" -crab');
       websearch_to_tsquery
----------------------------------
 'supernova' <-> 'star' & !'crab'
(1 row)

SELECT websearch_to_tsquery('english', '"sad cat" or "fat rat"');
       websearch_to_tsquery
-----------------------------------
 'sad' <-> 'cat' | 'fat' <-> 'rat'
(1 row)

SELECT websearch_to_tsquery('english', 'signal -"segmentation fault"');
         websearch_to_tsquery
---------------------------------------
 'signal' & !( 'segment' <-> 'fault' )
(1 row)

SELECT websearch_to_tsquery('english', '""" )( dummy \\ query <->');
 websearch_to_tsquery
----------------------
 'dummi' & 'queri'
(1 row)
```

12.3.3. 搜索结果排名

排名旨在衡量文档与特定查询的相关程度，以便在匹配很多时优先显示最相关的结果。PostgreSQL 提供了两种预定义的排名函数，它们会综合考虑词法信息、邻近关系和结构信息；也就是说，会考虑查询词在文档中出现的频率、这些词彼此之间的距离，以及这些词所处文档部分的重要程度。不过，相关性这一概念本身就比较模糊，而且高度依赖具体应用。不同应用可能还需要额外信息参与排名，例如文档修改时间。内置排名函数仅仅是示例。你可以编写自己的排名函数，或者把它们的结果与其他因素结合起来，以满足特定需求。

目前可用的两种排名函数是：

```
ts_rank([ weights float4[], ] vector tsvector, query tsquery [,
normalization integer ]) returns float4
```

根据向量中匹配词位的频率对向量进行排名。

`ts_rank_cd([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4`

该函数为给定的文档向量和查询计算覆盖密度排名。该方法见 Clarke、Cormack 和 Tudhope 于 1999 年发表于期刊 “Information Processing and Management” 的文章 “Relevance Ranking for One to Three Term Queries” 。覆盖密度排名与 `ts_rank` 类似，但还会考虑匹配词位彼此之间的接近程度。

该函数需要词位位置信息才能完成计算。因此，它会忽略 `tsvector` 中任何 “被剥离的” 词位。如果输入中根本没有未剥离的词位，结果就会是零（有关 `strip` 函数以及 `tsvector` 中位置信息的更多内容，见第 12.4.1 节）。

对这两个函数来说，可选的 *`weights`* 参数允许根据词实例的标注情况赋予它们不同权重。权重数组按如下顺序指定各类词的权重：

`{D-权重， C-权重， B-权重， A-权重}`

如果没有提供 *`weights`*，则使用如下默认值：

`{0.1, 0.2, 0.4, 1.0}`

通常，权重用于标记来自文档特定区域的词，例如标题或开头摘要中的词，从而使它们相较于正文中的词被赋予更高或更低的重要性。

由于较长的文档更有机会包含查询词，因此把文档大小纳入考量是合理的。例如，一个一百词的文档里某个搜索词出现五次，通常会比一个一千词的文档里同一搜索词也只出现五次更相关。两种排名函数都接受一个整数 *`normalization`* 选项，用于指定文档长度是否影响排名，以及具体如何影响。该整数选项控制多种行为，因此它是一个位掩码：你可以使用 `|` 指定一种或多种行为（例如 `2|4`）。

- 0（默认值）忽略文档长度
- 1 用 1 + 文档长度的对数除排名
- 2 用文档长度除排名
- 4 用匹配范围之间的平均调和距离除排名（仅由 `ts_rank_cd` 实现）
- 8 用文档中不同词的数量除排名
- 16 用 1 + 文档中不同词数量的对数除排名
- 32 用排名 + 1 除排名

如果指定了多个标志位，这些变换将按上面列出的顺序依次应用。

需要注意的是，排名函数不会使用任何全局信息，因此不可能像有时人们希望的那样，给出一种能够公平归一化到 1% 或 100% 的结果。归一化选项 32（`rank/(rank+1)`）可以把所有排名缩放到 0 到 1 的范围内，但这当然只是表面上的变化，不会影响搜索结果的顺序。

下面是只选择排名最高的十个匹配的示例：

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |   rank
-----------------------------------------------+----------
 Neutrinos in the Sun                          |      3.1
 The Sudbury Neutrino Detector                 |      2.4
 A MACHO View of Galactic Dark Matter          |  2.01317
 Hot Gas and Dark Matter                       |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos                   |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter                       |   1.6123
 Ice Fishing for Cosmic Neutrinos              |      1.6
 Weak Lensing Distorts the Universe            | 0.818218
```

下面是使用归一化排名的同一示例：

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |        rank
-----------------------------------------------+-------------------
 Neutrinos in the Sun                          | 0.756097569485493
 The Sudbury Neutrino Detector                 | 0.705882361190954
 A MACHO View of Galactic Dark Matter          | 0.668123210574724
 Hot Gas and Dark Matter                       |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                   | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter     | 0.650072921219637
 Hot Gas and Dark Matter                       | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos              | 0.615384618911517
 Weak Lensing Distorts the Universe            | 0.450010798361481
```

排名计算可能非常昂贵，因为它需要访问每个匹配文档的 tsvector，这往往会受 I/O 限制而变慢。不幸的是，这几乎无法避免，因为实际查询常常会产生大量匹配。

12.3.4. 高亮结果

在呈现搜索结果时，理想的方式是展示每个文档的一段摘录，并说明它与查询的关系。通常，搜索引擎会在文档片段中标出搜索词。PostgreSQL 提供了 ts_headline 函数来实现这一功能。

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text ]) returns text
```

ts_headline 接收文档和查询，并返回文档中一段高亮查询词的摘录。用于解析文档的配置可通过 *config* 指定；若省略 *config*，则使用 default_text_search_config 配置。

如果指定了 *options* 字符串，它必须由一个或多个用逗号分隔的*option=value* 对组成。可用选项有：

- MaxWords、MinWords（整数）：这两个数值决定输出摘要的最长和最短长度。默认值分别为 35 和 15。
- ShortWord（整数）：长度不超过该值的词，如果不是查询词，就会从摘要的开头和结尾处被丢弃。默认值 3 会去掉常见的英语冠词。
- HighlightAll（布尔值）：如果为 true，则整个文档都会被用作摘要，而忽略前面三个参数。默认值是 false。
- MaxFragments（整数）：要显示的最大文本片段数。默认值 0 选择一种非基于片段的摘要生成方法。大于 0 的值选择基于片段的摘要生成方法（见下文）。
- StartSel、StopSel（字符串）：用于界定文档中查询词的字符串，以便与摘录中的其他词区分。默认值分别为“”和“”，可以用于 HTML 输出。
- FragmentDelimiter（字符串）：当显示多个片段时，片段之间将以该字符串分隔。默认值是“ ... ”。

这些选项名不区分大小写。如果字符串值包含空格或逗号，则必须用双引号括起来。

在非片段模式下，ts_headline 会为给定的 *query* 定位匹配，并从中选择一处显示，优先选择在允许摘要长度内包含更多查询词的匹配。在基于片段的摘要生成模式下，ts_headline 会定位查询匹配，并把每个匹配切分成多个不超过 MaxWords 个词的“片段”，优先选择包含更多查询词的片段，并在可能时把片段“扩展”到周围词语。因此，当查询匹配跨越文档中较大区域，或希望显示多个匹配时，片段模式会更有用。无论哪种模式，如果无法识别出查询匹配，都会显示由文档前 MinWords 个词组成的单个片段。

例如：

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('english', 'query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms                       +
 and return them in order of their <b>similarity</b> to the+
 <b>query</b>.

SELECT ts_headline('english',
  'Search terms may occur
many times in a document,
requiring ranking of the search matches to decide which
occurrences to display in the result.',
  to_tsquery('english', 'search & term'),
  'MaxFragments=10, MaxWords=7, MinWords=3, StartSel=<<, StopSel=>>');
                        ts_headline
------------------------------------------------------------
 <<Search>> <<terms>> may occur                            +
 many times ... ranking of the <<search>> matches to decide
```

ts_headline 使用的是原始文档，而不是 tsvector 摘要，因此它可能较慢，使用时应当谨慎。

12.4. 附加特性

本节介绍一些在文本检索中很有用的附加函数和操作符。

12.4.1. 操纵文档

第 12.3.1 节展示了原始文本文档如何被转换成 `tsvector` 值。PostgreSQL 也提供了用于操纵已经处于 `tsvector` 形式文档的函数和操作符。

`tsvector || tsvector`

`tsvector` 连接操作符返回一个向量，它结合了两个参数向量中的词位和位置信息。位置和权重标签在连接过程中会被保留。右侧向量中的位置会按左侧向量中出现的最大位置做偏移，因此结果几乎等同于对两个原始文档字符串连接后的结果执行 `to_tsvector`。（这种等价并不完全成立，因为从左侧参数末尾移除的停用词不会影响结果；而如果直接做文本连接，它们本来会影响右侧参数中词位的位置。）

相比先连接文本再应用 `to_tsvector`，直接连接向量的一个优点是，你可以用不同配置来解析文档的不同部分。此外，由于 `setweight` 会以相同方式标记给定向量中的全部词位，如果希望给文档不同部分打上不同权重，就必须先解析文本并调用 `setweight`，然后再做连接。

`setweight(`*`vector`* `tsvector, `*`weight`* `"char") returns tsvector`

`setweight` 返回输入向量的一个副本，其中每个位置都被标注为给定的 *`weight`*：`A`、`B`、`C` 或 `D`（`D` 是新向量的默认值，因此不会在输出中显示）。这些标签在向量连接时会被保留，从而允许排名函数对来自文档不同部分的词赋予不同权重。

注意权重标签是应用到位置而不是词位。如果输入向量已经被剥离了位置，则`setweight` 什么也不会做。

`length(`*`vector`* `tsvector) returns integer`

返回存储在向量中的词位数。

`strip(`*`vector`* `tsvector) returns tsvector`

返回一个向量，其中列出与给定向量相同的词位，但不包含任何位置或权重信息。其结果通常比未剥离的向量小得多，但也没那么有用。相关度排名在已剥离向量上的效果不如未剥离向量。此外，`<->`（FOLLOWED BY）`tsquery` 操作符永远不会匹配已剥离的输入，因为它无法确定词位出现之间的距离。

表 9.41 中有 `tsvector` 相关函数的完整列表。

12.4.2. 操纵查询

第 12.3.2 节展示了原始文本查询如何被转换成 `tsquery` 值。PostgreSQL 也提供了用于操纵已经处于 `tsquery` 形式查询的函数和操作符。

`tsquery && tsquery`

返回两个给定查询的 AND 组合。

`tsquery || tsquery`

返回两个给定查询的 OR 组合。

`!! tsquery`

返回给定查询的否定（NOT）。

`tsquery <-> tsquery`

返回一个查询，它使用 `<->`（FOLLOWED BY）`tsquery` 操作符，搜索第一个给定查询的匹配后紧跟着第二个给定查询的匹配。例如：

```
SELECT to_tsquery('fat') <-> to_tsquery('cat | rat');
          ?column?
----------------------------
 'fat' <-> ( 'cat' | 'rat' )
```

`tsquery_phrase(`*`query1`* `tsquery,` *`query2`* `tsquery [,` *`distance`* `integer ]) returns tsquery`

返回一个查询，它使用 `<`*`N`*`> tsquery` 操作符，搜索第一个给定查询的匹配，并在其后距离正好为 *`distance`* 个词位处搜索第二个给定查询的匹配。例如：

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);
  tsquery_phrase
------------------
 'fat' <10> 'cat'
```

`numnode(`*`query`* `tsquery) returns integer`

返回 `tsquery` 中的节点数（词位加操作符）。这个函数可用于判断 *`query`* 是否有意义（返回值 > 0），或者是否只包含停用词（返回 0）。例如：

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE:  query contains only stopword(s) or doesn't contain lexeme(s), ignored
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

`querytree(`*`query`* `tsquery) returns text`

返回一个 `tsquery` 中可用于搜索索引的部分。此函数可用于检测无法使用索引的查询，例如只包含停用词或只包含否定词项的查询。例如：

```
SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
```

12.4.2.1. 查询重写

ts_rewrite 函数族会在给定的 tsquery 中搜索目标子查询的出现，并把每一次出现都替换为替换子查询。本质上，这就是面向 tsquery 的一种子串替换。目标与替换的组合可以看作一条查询重写规则。一组这样的重写规则可以成为非常强大的搜索辅助工具。例如，你可以利用同义词扩展搜索（如 new york、big apple、nyc、gotham），或者收窄搜索，把用户引向某个热门主题。该特性与分类词典（第 12.6.4 节）在功能上有一定重叠。不过，重写规则可以随时修改而无需重建索引，而更新分类词典则必须重建索引后才能生效。

```
ts_rewrite (query tsquery, target tsquery, substitute tsquery)
 returns tsquery
```

这种形式的 ts_rewrite 只应用一条重写规则：无论 *target* 在 *query* 中的何处出现，都用 *substitute* 替换它。例如：

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
```

```
ts_rewrite (query tsquery, select text) returns tsquery
```

这种形式的 ts_rewrite 接受一个起始 *query* 和一个以文本字符串给出的 SQL *select* 命令。该 *select* 必须产生两列 tsquery 类型的值。对于 *select* 结果中的每一行，第一列值（目标）在当前 *query* 值中的所有出现都会被第二列值（替换）取代。例如：

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

注意，当以这种方式应用多个重写规则时，应用顺序可能很重要；因此在实际使用中，你通常会要求源查询按某个排序键 ORDER BY。

我们来看一个现实中的天文示例。我们将使用表驱动的重写规则来扩展查询 supernovae：

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
          ts_rewrite
---------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

我们只需更新表，就可以修改这些重写规则：

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
                 ts_rewrite
--------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

当重写规则很多时，重写过程可能会很慢，因为它需要检查每一条规则是否可能匹配。为了过滤掉显然不可能命中的规则，可以使用 tsquery 类型的包含操作符。在下面的例子中，我们只选择那些可能匹配原始查询的规则：

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery @> t');
 ts_rewrite
------------
 'b' & 'c'
```

12.4.3. 用于自动更新的触发器

当使用单独一列来存储文档的 tsvector 表示时，就需要创建触发器，以便在文档内容列发生变化时更新该 tsvector 列。系统为此提供了两个内置触发器函数，你也可以自行编写触发器函数。

```
tsvector_update_trigger(tsvector_column_name, config_name, text_column_name [, ... ])
tsvector_update_trigger_column(tsvector_column_name, config_column_name, text_column_name [, ..
. ])
```

这些触发器函数会在 CREATE TRIGGER 命令中给定参数的控制下，从一个或多个文本列自动计算出一个 tsvector 列。下面是它们的一个使用示例：

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE FUNCTION
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title    |         body          |            tsv
------------+-----------------------+---------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
   title    |         body
------------+-----------------------
 title here | the body text is here
```

创建该触发器之后，对title 或 body 的任何修改都会自动反映到 tsv 中，而应用程序无需关心这一点。

第一个触发器参数必须是要更新的 tsvector 列名。第二个参数指定执行转换时要使用的文本检索配置。对于 tsvector_update_trigger，配置名直接作为第二个触发器参数给出。如上所示，它必须带模式限定，这样触发器行为就不会随着 search_path 的变化而变化。对于 tsvector_update_trigger_column，第二个触发器参数则是另一个表列的名称，该列必须是 regconfig 类型。这样就可以按行选择配置。其余参数是文本列的名称（类型为 text、varchar 或 char），它们会按给定顺序并入文档。NULL 值会被跳过（但其他列仍会被索引）。

这些内置触发器的一个限制是，它们会以相同方式处理所有输入列。若要对列做不同处理 – 例如让标题和正文具有不同权重 – 就需要编写自定义触发器。下面是一个使用 PL/pgSQL 作为触发器语言的示例：

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('pg_catalog.english', coalesce(new.title,'')), 'A') ||
     setweight(to_tsvector('pg_catalog.english', coalesce(new.body,'')), 'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
    ON messages FOR EACH ROW EXECUTE FUNCTION messages_trigger();
```

请记住，在触发器中创建 tsvector 值时，明确指定配置名称非常重要，这样列的内容才不会受到 default_text_search_config 变化的影响。否则，很可能会导致问题，例如在转储并恢复之后搜索结果发生变化。

12.4.4. 收集文档统计数据

ts_stat 可用于检查配置，并寻找候选停用词。

```
ts_stat(sqlquery text, [ weights text, ]
        OUT word text, OUT ndoc integer,
        OUT nentry integer) returns setof record
```

sqlquery 是一个文本值，其中包含一条必须返回单一 tsvector 列的 SQL 查询。ts_stat 会执行该查询，并返回该 tsvector 数据中每个不同词位（单词）的统计信息。返回的列如下：

- *word* text — 一个词位的值
- *ndoc* integer — 词出现过的文档（tsvector）的数量
- *nentry* integer — 词出现的总次数

如果提供了 *weights*，则只统计具有这些权重之一的词的出现次数。

例如，要在一个文档集合中查找十个最频繁的词：

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

同样的查询，但只统计权重为 A 或 B 的出现次数：

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

12.5. 解析器

文本检索解析器负责把未处理的文档文本划分成词元并标识每个词元的类型，而可能的类型集合由解析器本身定义。注意，解析器完全不会修改文本 — 它只是识别看似合理的词边界。由于作用范围有限，相比自定义词典，对应用相关的自定义解析器的需求没有那么强烈。目前PostgreSQL只提供一种内置解析器，而它已经被证明对广泛的应用都很有用。

内置解析器被称为 pg_catalog.default。它识别 23 种词元类型，如表 12.1 所示。

表 12.1. 默认解析器的词元类型

别名	描述	示例
asciiword	单词，全部由 ASCII 字母组成	elephant
word	单词，全部由字母组成	mañana
numword	单词，字母和数字	beta1
asciihword	带连字符的单词，全部为 ASCII 字符	up-to-date
hword	带连字符的单词，由字母组成	lógico-matemática
numhword	带连字符的单词，字母和数字	postgresql-beta1
hword_asciipart	带连字符的单词部分，全部为 ASCII 字符	在 postgresql-beta1 上下文中的 postgresql

别名	描述	示例
hword_part	带连字符的单词部分，全部由字母组成	在 lógico-matemática 上下文中的 lógico 或 matemática
hword_numpart	带连字符的单词部分，字母和数字	在 postgresql-beta1 上下文中的 beta1
email	电子邮件地址	foo@example.com
protocol	协议头部	http://
url	URL	example.com/stuff/index.html
host	主机	example.com
url_path	URL 路径	/stuff/index.html，在 URL 上下文中
file	文件或路径名	/usr/local/foo.txt，如果不在 URL 中
sfloat	科学记数法	-1.234e56
float	十进制记数法	-1.234
int	有符号整数	-1234
uint	无符号整数	1234
version	版本号	8.3.0
tag	XML 标签	<a href="dictionaries.html">
entity	XML 实体	&
blank	空白符号	（未被识别为其他类型的空白或标点符号）

注意

解析器对“字母”的界定由数据库的区域设置决定，具体是lc_ctype。只包含基本 ASCII 字母的词被报告为一个单独的词元类型，因为有时区分它们会很有用。在大部分欧洲语言中，词元类型word 和 asciiword 应该被同样对待。

email 不支持所有由 RFC 5322 定义的有效电子邮件字符。具体来说，电子邮件用户名仅支持句点、短横线和下划线这些非字母数字字符。

解析器有可能从同一段文本中产生重叠的词元。举例来说，一个带连字符的词既会被报告为整个单词，也会被报告为各个组成部分：

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |               description                |     token
-----------------+------------------------------------------+---------------
 numhword         | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank            | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank            | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

这种行为是可取的，因为它既允许针对整个复合词搜索，也允许针对各个组成部分搜索。下面是另一个有启发性的示例：

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/index.html');
  alias   |  description  |           token
----------+---------------+----------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

12.6. 词典

词典用于消除不应参与搜索的词（停用词），并用于对词进行正规化，以便同一个词的不同派生形式能够匹配。成功完成正规化的词被称为词位。除了改善搜索质量，正规化和移除停用词还会减小文档的`tsvector` 表示，从而提高性能。正规化并不总是具有语言学意义，而且通常依赖于应用的语义。

一些正规化的示例：

- 语言学上的例子 – Ispell 词典尝试把输入词约简为一种正规化形式；词干分析器词典则去掉词尾
- URL 地址可以规范化，以便让等价的 URL 匹配：
  - http://www.pgsql.ru/db/mw/index.html
  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html
- 颜色名可以被它们的十六进制值替换，例如`red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF`
- 如果要索引数字，我们可以去掉某些小数位，以缩小可能数值的范围。因此，如果只保留小数点后两位，那么例如 3.14159265359、3.1415926 和 3.14 在正规化后就会变成相同的值。

一个词典是一个程序，它接受一个词元作为输入，并返回：

- 如果输入的词元对词典是已知的，则返回一个词位数组（注意一个词元可能产生多于一个词位）
- 一个带有 `TSL_FILTER` 标志的单个词位，它会用一个新词元替换原始词元，并把它传递给后续词典（执行这种工作的词典称为过滤词典）
- 如果词典知道该词元但它是一个停用词，则返回一个空数组
- 如果词典不识别该输入词元，则返回`NULL`

PostgreSQL 为许多语言提供了预定义的词典。也有多种预定义模板可以被用于创建带自定义参数的新词典。每一种预定义词典模板在下面描述。如果没有合适的现有模板，可以创建新的；示例见PostgreSQL 发行版的 `contrib/`目录。

文本检索配置把一个解析器与一组用于处理解析器输出词元的词典绑定在一起。对于解析器可能返回的每一种词元类型，配置都会指定一个单独的词典列表。当解析器找到该类型的词元时，会按顺序依次查询列表中的每个词典，直到有某个词典把它识别为已知词。如果它被识别为停用词，或者没有任何词典识别它，那么该词元就会被丢弃，既不会建立索引，也不会参与搜索。通常，第一个返回非 `NULL` 输出的词典就决定结果，后续词典不会再被查询；但过滤词典可以把给定单词替换为一个修改后的单词，再传递给后续词典。

配置词典列表的一般规则是，把适用范围最窄、最专门的词典放在最前面，然后是更通用的词典，最后以一个非常通用的词典收尾，例如 Snowball 词干分析器，或能识别所有内容的 `simple`。例如，对于一个天文学相关搜索（配置名为 `astro_en`），可以把词元类型 `asciiword`（ASCII 词）绑定到一个天文学术语同义词词典、一个通用英语词典，以及一个 Snowball 英语词干分析器：

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

过滤词典可以放在列表中的任何位置，但放在末尾没有用处。过滤词典可用于先对词做部分正规化，以简化后续词典的工作。例如，可以用过滤词典去掉带重音字母中的重音符号，就像 unaccent 模块所做的那样。

12.6.1. 停用词

停用词是非常常见、几乎出现在每个文档中的词，没有区分价值。因此，在全文检索中可以忽略它们。例如，每篇英文文本都包含像 `a` 和 `the` 这样的词，所以把它们存储在索引中没有用处。不过，停用词确实会影响 `tsvector` 中的位置，而这又会影响排名：

```
SELECT to_tsvector('english', 'in the list of stop words');
        to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

缺失的位置 1、2、4 就是由停用词造成的。对包含停用词和不包含停用词的文档，计算出来的排名会明显不同：

```
SELECT ts_rank_cd (to_tsvector('english', 'in the list of stop words'), to_tsquery('list &
 stop'));
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english', 'list stop words'), to_tsquery('list & stop'));
 ts_rank_cd
------------
        0.1
```

如何处理停用词取决于具体词典。例如，`ispell` 词典会先正规化单词，再查看停用词列表，而 `Snowball` 词干分析器则先检查停用词列表。这种不同行为是为了尽量减少噪声。

12.6.2. 简单词典

`simple` 词典模板的工作方式是先把输入词元转换为小写，再根据停用词文件检查它。如果在文件中找到该词元，就返回一个空数组，从而丢弃该词元；否则，返回其小写形式作为正规化后的词位。或者，也可以把该词典配置为把非停用词报告为未识别，从而允许它们传递给列表中的下一个词典。

下面是一个使用 simple 模板的词典定义示例：

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

其中，english 是停用词文件的基名。文件完整名称将是 $SHAREDIR/tsearch_data/english.stop，其中 $SHAREDIR 表示 PostgreSQL 安装的共享数据目录，通常是 /usr/local/share/postgresql（如果不确定，可用 pg_config --sharedir 来确定）。文件格式只是一个单词列表，每行一个单词。空行和行尾空格会被忽略，大写会折叠为小写，但不会对文件内容做其他处理。

现在我们可以测试这个词典：

```
SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

如果在停用词文件中找不到该词，我们也可以选择返回 NULL，而不是返回它的小写形式。这种行为可以通过把词典的 Accept 参数设置为 false 来启用。继续上面的例子：

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------


SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

在默认设置 Accept = true 下，只有把 simple 词典放在词典列表末尾才有意义，因为它不会把任何词元传递给后续词典。相反，Accept = false 只有在后面至少还有一个词典时才有用。

> **小心**
>
> 大多数词典类型都依赖配置文件，例如停用词文件。这些文件必须采用 UTF-8 编码保存。当它们被读入服务器时，如果数据库编码不同，就会被转换为实际的数据库编码。

> **小心**
>
> 通常，词典配置文件在某个数据库会话中第一次被使用时，只会被读取一次。如果你修改了配置文件，并希望强制现有会话加载新内容，可以对该词典执行一条 ALTER TEXT SEARCH DICTIONARY 命令。这可以是一条“假”更新，也就是并不实际修改任何参数值的更新。

12.6.3. 同义词词典

这个词典模板用于创建把一个单词替换为同义词的词典。不支持短语（对此请使用分类词典模板第 12.6.4 节）。同义词词典可以用来克服语言学上的问题，例如防止英语词干分析词典把“Paris”约简成“pari”。只要在同义词词典中加入一行 Paris paris，并把该词典放在 english_stem 词典之前即可。例如：

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        | dictionary | lexemes
-----------+-----------------+-------+---------------------------+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
```

synonym 模板要求的唯一参数是 SYNONYMS，它是其配置文件的基名 – 上例中的 my_synonyms。该文件的完整名称将是 $SHAREDIR/tsearch_data/my_synonyms.syn（其中 $SHAREDIR 表示 PostgreSQL 安装的共享数据目录）。该文件格式是每行一个要被替换的词，后面跟着它的同义词，用空白分隔。空行和结尾的空格会被忽略。

synonym 模板还有一个可选参数 CaseSensitive，默认值为false。当 CaseSensitive 为 false 时，同义词文件中的词会像输入词元一样折叠为小写。当它为true 时，词和词元都不会被折叠为小写，而是按原样比较。

可以在配置文件中把星号（*）放在同义词末尾，表示该同义词是一个前缀。当该条目用于 to_tsvector() 时，星号会被忽略；但当它用于 to_tsquery() 时，结果将是一个带有前缀匹配标记的查询项（见第 12.3.2 节）。例如，假设在 $SHAREDIR/tsearch_data/synonym_sample.syn 中有以下条目：

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

那么将得到如下结果：

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn', 'indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst', 'indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst', 'indices');
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
---------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst', 'indices');
 ?column?
----------
 t
(1 row)
```

12.6.4. 分类词典

一个分类词典（有时被简写成 TZ）是一个词的集合，其中包括了词与短语之间的联系，即广义词（BT）、狭义词（NT）、首选词、非首选词、相关词等。

分类词典基本上会用一个首选词替换所有非首选词，并且也可以选择保留原始词用于索引。PostgreSQL 的分类词典的当前实现是同义词词典的一个扩展，并增加了短语支持。一个分类词典需要一个采用以下格式的配置文件：

```
# 这是一条注释
示例词（一个或多个） ： 索引词（一个或多个）
更多示例词（一个或多个） ： 更多索引词（一个或多个）
...
```

其中冒号（:）符号用作短语及其替换词之间的分隔符。

分类词典会使用一个子词典（在词典配置中指定）在检查短语匹配之前正规化输入文本。只能选择一个子词典。如果子词典无法识别某个词，就会报错。在这种情况下，你应当避免使用该词，或者让子词典学会它。你可以在某个被索引词的开头放置一个星号（*），以跳过将子词典应用于该词，但所有样例词都必须能被子词典识别。

如果有多个短语匹配输入，分类词典会选择最长的匹配；长度相同时使用最后的定义。

由子词典识别的特定停用词不能够被指定；改用?标记可出现任意停用词的位置。例如，假定根据子词典a 和 the 是停用词：

```
? one ? two : swsw
```

匹配 a one the two 和 the one a two；两者都将被 swsw 替换。

由于分类词典具备识别短语的能力，因此它必须记住自身状态并与解析器交互。分类词典会利用这些映射关系来判断自己是应当处理下一个词，还是停止累积。分类词典必须经过仔细配置。例如，如果分类词典只被映射到 asciiword 词元，那么像 one 7 这样的分类词典定义就无法工作，因为词元类型 uint 并没有映射给该分类词典。

> **小心**
>
> 分类词典会在建立索引时使用，因此其参数发生任何变化都要求重新建立索引。对于大多数其他词典类型，像增加或移除停用词这样的细小改动则不会强制重新建立索引。

12.6.4.1. 分类词典配置

要定义一个新的分类词典，可使用thesaurus 模板。例如：

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

这里：

- thesaurus_simple 是新词典的名称
- mythesaurus 是分类词典配置文件的基名（它的全名将是 $SHAREDIR/tsearch_data/mythesaurus.ths，其中$SHAREDIR 表示安装的共享数据目录）。
- pg_catalog.english_stem 是要用于分类词典正规化的子词典（这里是一个 Snowball 英语词干分析器）。注意子词典将拥有它自己的配置（例如停用词），但这里没有展示。

现在可以在配置中把分类词典 thesaurus_simple 绑定到想要的词元类型上，例如：

```
ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_simple;
```

12.6.4.2. 分类词典示例

考虑一个简单的天文分类词典 thesaurus_astro，其中包含一些天文单词组合：

```
supernovae stars : sn
crab nebulae : crab
```

下面我们创建一个词典，并把一些词元类型绑定到天文分类词典和英语词干分析器上：

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

现在我们可以看看它是如何工作的。ts_lexize 对测试分类词典并不十分有用，因为它把输入当作一个单独的词元。相反，我们可以使用 plainto_tsquery 和 to_tsvector，它们会把输入字符串拆分成多个词元：

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

原则上，如果把参数用引号括起来，也可以使用 to_tsquery：

```
SELECT to_tsquery('''supernova star''');
 to_tsquery
------------
 'sn'
```

请注意，supernova star 能匹配 thesaurus_astro 中的 supernovae stars，因为我们在分类词典定义中指定了 english_stem 词干分析器。它去掉了词尾的 e 和 s。

如果既要为替换词建立索引，也要为原始短语建立索引，只需把原始短语也写到定义右侧即可：

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

12.6.5. Ispell 词典

Ispell 词典模板支持形态词典，它可以把一个词的许多不同语言学形式正规化为同一个词位。例如，一个英语 Ispell 词典可以匹配搜索词 `bank` 的所有变格和变位形式，例如 `banking`、`banked`、`banks`、`banks'` 和 `bank's`。

标准 PostgreSQL 发行版不包含任何 Ispell 配置文件。大量语言的词典可从 Ispell[1] 获取。此外，还支持一些更现代的词典文件格式 – MySpell[2] (OO < 2.0.1) 和 Hunspell[3](OO >= 2.0.2)。在 OpenOffice Wiki[4] 上有大量词典可用。

要创建一个 Ispell 词典，请执行以下步骤：

- 下载词典配置文件。OpenOffice 扩展文件的扩展名是 `.oxt`。需要提取 `.aff` 和 `.dic` 文件，将扩展名改为 `.affix` 和 `.dict`。对于某些词典文件，还需要使用下面的命令把字符转换成 UTF-8 编码（例如挪威语词典）：

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

- 复制文件到 `$SHAREDIR/tsearch_data` 目录
- 用下面的命令把文件载入到 PostgreSQL：

```
CREATE TEXT SEARCH DICTIONARY english_hunspell (
    TEMPLATE = ispell,
    DictFile = en_us,
    AffFile = en_us,
    Stopwords = english);
```

这里，`DictFile`、`AffFile` 和 `StopWords` 指定词典、词缀和停用词文件的基名。停用词文件的格式和前面解释的`simple` 词典类型相同。其他文件的格式在这里没有指定，但是也可以从上面提到的网站获得。

Ispell 词典通常只能识别有限的词，因此其后应跟着另一个覆盖范围更广的词典；例如，一个 Snowball 词典，它可以识别所有输入。

Ispell 的 `.affix` 文件具有下面的结构：

```
prefixes
flag *A:
    .           >   RE      # 例如 enter > reenter
suffixes
flag T:
    E           >   ST      # 例如 late > latest
    [^AEIOU]Y   >   -Y,IEST # 例如 dirty > dirtiest
    [AEIOU]Y    >   EST     # 例如 gray > grayest
    [^EY]       >   EST     # 例如 small > smallest
```

[1] https://www.cs.hmc.edu/~geoff/ispell.html
[2] https://en.wikipedia.org/wiki/MySpell
[3] https://hunspell.github.io/
[4] https://wiki.openoffice.org/wiki/Dictionaries

.dict 文件具有下面的结构：

```
lapse/ADGRS
lard/DGRS
large/PRTY
lark/MRS
```

.dict 文件的格式是：

基本词形/词缀类名

在.affix 文件中，每一个词缀标志以下面的格式描述：

条件 > [-删除的字母,] 添加的词缀

这里的条件具有和正则表达式相似的格式。它可以使用分组[...] 和[^...]。例如，[AEIOU]Y 表示词的最后一个字母是"y" 并且倒数第二个字母是"a"、"e"、"i"、"o" 或者"u"。[^EY] 表示最后一个字母既不是"e" 也不是"y"。

Ispell 词典支持划分复合词，这是一个有用的特性。注意词缀文件应该用compoundwords controlled 语句指定一个特殊标志，用于标记词典中可以参与构成复合词的词：

```
compoundwords  controlled z
```

下面是挪威语的一些示例：

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

MySpell 格式是 Hunspell 格式的一个子集。Hunspell 的.affix 文件具有下面的结构：

```
PFX A Y 1
PFX A   0     re          .
SFX T N 4
SFX T   0     st          e
SFX T   y     iest        [^aeiou]y
SFX T   0     est         [aeiou]y
SFX T   0     est         [^ey]
```

一个词缀类的第一行是首部。首部后面列出了词缀规则的字段：

- 参数名（PFX 或者 SFX）
- 标志（词缀类的名称）
- 从该词的开始（前缀）或者结尾（后缀）剥离字符
- 增加词缀
- 和正则表达式格式类似的条件。

.dict 文件看起来和 Ispell 的.dict 文件相似：

```
larder/M
lardy/RT
large/RSPMYT
largehearted
```

> **注意**
>
> MySpell 不支持复合词。Hunspell 则对复合词有更好的支持。当前，PostgreSQL 只实现了 Hunspell 中基本的复合词操作。

12.6.6. Snowball 词典

Snowball 词典模板基于 Martin Porter 的一个项目，他是流行的英语 Porter 词干分析算法的发明者。Snowball 现在对许多语言提供词干分析算法（详见Snowball 站点[5]）。每种算法都知道如何把其语言中词的常见变体约简为基本拼写形式，也就是词干。一个 Snowball 词典要求一个 language 参数来标识要用哪种词干分析器，并且可以选择指定一个stopword 文件名来给出一个要被消除的词列表（PostgreSQL 的标准停用词列表也是由 Snowball 项目提供的）。例如，有一个内置的定义等效于

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

停用词文件格式和已经解释的一样。

一个 Snowball 词典识别所有输入，不管它能不能简化该词，因此它应当被放置在词典列表的最后。把它放在任何其他词典前面是没有用处的，因为一个词元永远不会穿过它而进入到下一个词典。

12.7. 配置示例

一个文本检索配置指定了将一个文档转换成一个 tsvector 所需的所有选项：用于把文本分解成词元的解析器，以及用于将每一个词元转换成词位的词典。每一次 to_tsvector 或 to_tsquery 的调用都需要一个文本检索配置来执行其处理。配置参数 default_text_search_config 指定了默认配置的名称，如果省略了显式的配置参数，文本检索函数将会使用它。它可以在postgresql.conf 中设置，或者使用SET 命令为一个单独的会话设置。

有一些预定义的文本检索配置可用，并且你可以容易地创建自定义的配置。为了便于管理文本检索对象，可以使用一组SQL 命令，并且有多个psql 命令可以显示有关文本检索对象（第 12.10 节）的信息。

[5] https://snowballstem.org/

作为一个示例，我们将创建一个配置pg，从复制内置的english 配置开始：

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

我们将使用一个 PostgreSQL 相关的同义词列表，并将它存储在$SHAREDIR/tsearch_data/pg_dict.syn 中。文件内容看起来像：

```
postgres    pg
pgsql       pg
postgresql  pg
```

我们定义同义词词典如下：

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

接下来我们注册 Ispell 词典 english_ispell，它有其自己的配置文件：

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

现在我们可以在配置 pg 中建立词的映射：

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

我们选择不索引或搜索某些内置配置确实处理的词元类型：

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

现在我们可以测试我们的配置：

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

下一步是把会话设置为使用这个新配置，该配置是在 public 模式中创建的：

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.pg
```

12.8. 测试和调试文本检索

一个自定义文本检索配置的行为很容易让人困惑。本节中描述的函数对于测试文本检索对象有用。你可以测试一个完整的配置，或者独立测试解析器和词典。

12.8.1. 配置测试

函数 ts_debug 可方便地测试一个文本检索配置。

```
ts_debug([ config regconfig, ] document text,
         OUT alias text,
         OUT description text,
         OUT token text,
         OUT dictionaries regdictionary[],
         OUT dictionary regdictionary,
         OUT lexemes text[])
         returns setof record
```

ts_debug 显示 *document* 的每一个词元的信息，词元由解析器产生并由配置的词典处理过。该函数使用由*config* 指定的配置，如果省略该参数则使用default_text_search_config 指定的配置。

ts_debug 为解析器在文本中标识的每一个词元返回一行。被返回的列是：

- *alias* text – 词元类型的短名称
- *description* text – 词元类型的描述
- *token* text – 词元的文本
- *dictionaries* regdictionary[] – 配置为这种词元类型选择的词典
- *dictionary* regdictionary – 识别该词元的词典，如果没有词典能识别则为NULL
- *lexemes* text[] – 识别该词元的词典产生的词位，如果没有词典能识别则为NULL；一个空数组（{}）表示该词元被识别为一个停用词

以下是一个简单的示例：

```
SELECT * FROM ts_debug('english', 'a fat  cat sat on a mat - it ate a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

为了做更完整的演示，我们先为英语创建一个`public.english`配置和 Ispell 词典：

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;
```

```
SELECT * FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |   description   |    token    |         dictionaries          |   dictionary   |
 lexemes
-----------+-----------------+-------------+-------------------------------+----------------+--
-----------
 asciiword | Word, all ASCII | The         | {english_ispell,english_stem} | english_ispell | {
}
 blank     | Space symbols   |             | {}                            |                |
 asciiword | Word, all ASCII | Brightest   | {english_ispell,english_stem} | english_ispell | {
bright}
 blank     | Space symbols   |             | {}                            |                |
 asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} | english_stem   | {
supernova}
```

在这个示例中，词`Brightest`被解析器识别为一个 ASCII 词（别名`asciiword`）。对于这种词元类型，词典列表是`english_ispell`和`english_stem`。该词被`english_ispell`识别，并被这个词典归约为词位`bright`。词`supernovaes`对`english_ispell`词典来说是未

知的，因此它会被传递给下一个词典；幸运的是，它随后被识别了。（实际上，english_stem 是一个 Snowball 词典，它能够识别所有输入；这也是为什么它被放在词典列表末尾。）

词 The 被 english_ispell 词典识别为一个停用词（第 12.6.1 节）并且将不会被索引。空格也被丢弃，因为该配置没有为它们提供词典。

你可以通过明确指定要查看的列来减少输出宽度：

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

12.8.2. 解析器测试

下列函数允许直接测试一个文本检索解析器。

```
ts_parse(parser_name text, document text,
         OUT tokid integer, OUT token text) returns setof record
ts_parse(parser_oid oid, document text,
         OUT tokid integer, OUT token text) returns setof record
```

ts_parse 解析给定的 *document*，并返回一组记录，每个由解析过程产生的词元对应一条记录。每条记录都包含一个 tokid，显示所分配的词元类型，以及一个 token，即该词元的文本。例如：

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(parser_name text, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
ts_token_type(parser_oid oid, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
```

ts_token_type 返回一个表，描述指定解析器能够识别的每一种词元类型。对于每种词元类型，该表给出解析器用来标记该类词元的整数 tokid、在配置命令中命名该词元类型的 alias，以及简短的 description。例如：

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------+-----------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
```

12.8.3. 词典测试

ts_lexize 函数便于测试词典。

```
ts_lexize(dict regdictionary, token text) returns text[]
```

如果输入的 *token* 是该词典已知的，则ts_lexize 返回一个词位数组；如果词元是词典已知的但是它是一个停用词，则返回一个空数组；或者如果它对词典是未知词，则返回NULL。

示例：

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

注意

ts_lexize 函数期望的是单个词元，而不是文本。下面这种情况可能会让人困惑：

```
SELECT ts_lexize('thesaurus_astro', 'supernovae stars') is null;
 ?column?
----------
 t
```

分类词典 thesaurus_astro 确实知道短语 supernovae stars，但 ts_lexize 会失败，因为它不会解析输入文本，而是把它当作一个单独的词元。测试分类词典时应使用 plainto_tsquery 或 to_tsvector，例如：

```
SELECT plainto_tsquery('supernovae stars');
 plainto_tsquery
-----------------
 'sn'
```

12.9. 文本检索的首选索引类型

有两种索引可以用来加速全文检索：GIN 和 GiST。请注意，索引对于全文检索并非强制要求，但在定期搜索某一列的情况下，通常是可取的。

要创建这样的索引，可以采用下面的任一种方式：

```
CREATE INDEX name ON table USING GIN (column);
```

创建基于 GIN（广义倒排索引）的索引。*column* 必须是 tsvector 类型。

```
CREATE INDEX name ON table USING GIST (column);
```

创建基于 GiST（广义搜索树）的索引。*column* 可以是 tsvector 或 tsquery 类型。

GIN 索引是文本检索的首选索引类型。作为倒排索引，每个词（词位）在其中都有一个索引项，其中有压缩过的匹配位置的列表。多词搜索可以找到第一个匹配，然后使用该索引移除缺少额外词的行。GIN 索引只存储 tsvector 值的词（词位），并且不存储它们的权重标签。因此，在使用涉及权重的查询时需要一次在表行上的重新检查。

一个 GiST 索引是有损的，这表示索引可能产生假匹配，并且有必要检查真实的表行来消除这种假匹配（PostgreSQL 在需要时会自动做这一步）。GiST 索引之所以是有损的，是因为每一个文档在索引中被表示为一个定长的签名。该签名通过将每个词 hash 到一个 n 位串中的一位，再将所有这些位进行 OR 运算来生成，结果是一个 n 位的文档签名。当两个词 hash 到同一个位位置时就会产生假匹配。如果查询中所有词都有匹配（真或假），则必须检索表行查看匹配是否正确。

有损性导致的性能下降归因于不必要的表记录（即被证实为假匹配的记录）获取。因为表记录的随机访问是较慢的，这限制了 GiST 索引的实用性。假匹配的可能性取决于几个因素，特别是不同词的数量，因此推荐使用词典来缩减这个数量。

注意 GIN 索引的构建时间常常可以通过增加 maintenance_work_mem 来缩短，而GiST 索引的构建时间则对该参数不敏感。

对大集合分区并正确使用 GIN 和 GiST 索引允许实现带在线更新的快速搜索。分区可以在数据库层面上使用表继承来完成，或者是通过将文档分布在服务器上并收集外部的搜索结果，例如通过外部数据访问。后者是可能的，因为排名函数只使用本地信息。

12.10. psql 支持

关于文本检索配置对象的信息可以在 psql 中使用一组命令获得：

```
\dF{d,p,t}[+] [PATTERN]
```

可选的+能产生更多细节。

可选参数 *PATTERN* 可以是文本检索对象的名称，也可以可选地带上模式限定。如果省略 *PATTERN*，则会显示所有可见对象的信息。*PATTERN* 还可以是正则表达式，并且可以为模式名和对象名分别提供独立的匹配模式。下面的示例说明了这一点：

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
 public | fulltext_cfg |


=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name        | Description
----------+--------------+-------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

可用命令如下：

```
\dF[+] [PATTERN]
```

列出文本检索配置（加上 + 可显示更多细节）。

```
=> \dF russian
            List of text search configurations
   Schema   |  Name   |            Description
------------+---------+-----------------------------------
 pg_catalog | russian | configuration for russian language

=> \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

\dFd[+] [PATTERN]

列出文本检索词典（加上 + 可显示更多细节）。

```
=> \dFd
                           List of text search dictionaries
   Schema   |      Name       |                        Description
------------+-----------------+-------------------------------------------------------------
 pg_catalog | danish_stem     | snowball stemmer for danish language
 pg_catalog | dutch_stem      | snowball stemmer for dutch language
 pg_catalog | english_stem    | snowball stemmer for english language
 pg_catalog | finnish_stem    | snowball stemmer for finnish language
 pg_catalog | french_stem     | snowball stemmer for french language
 pg_catalog | german_stem     | snowball stemmer for german language
 pg_catalog | hungarian_stem  | snowball stemmer for hungarian language
 pg_catalog | italian_stem    | snowball stemmer for italian language
 pg_catalog | norwegian_stem  | snowball stemmer for norwegian language
 pg_catalog | portuguese_stem | snowball stemmer for portuguese language
 pg_catalog | romanian_stem   | snowball stemmer for romanian language
 pg_catalog | russian_stem    | snowball stemmer for russian language
 pg_catalog | simple          | simple dictionary: just lower case and check for stopword
 pg_catalog | spanish_stem    | snowball stemmer for spanish language
 pg_catalog | swedish_stem    | snowball stemmer for swedish language
 pg_catalog | turkish_stem    | snowball stemmer for turkish language
```

```
\dFp[+] [PATTERN]
```

列出文本检索解析器（加上 + 可显示更多细节）。

```
=> \dFp
        List of text search parsers
   Schema   |  Name   |     Description
------------+---------+---------------------
 pg_catalog | default | default word parser
=> \dFp+
    Text search parser "pg_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

        Token types for parser "pg_catalog.default"
   Token name    |               Description
-----------------+------------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | XML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

```
\dFt[+] [PATTERN]
```

列出文本检索模板（加上 + 可显示更多细节）。

```
=> \dFt
                            List of text search templates
   Schema   |   Name    |                        Description
------------+-----------+------------------------------------------------------------
 pg_catalog | ispell    | ispell dictionary
 pg_catalog | simple    | simple dictionary: just lower case and check for stopword
 pg_catalog | snowball  | snowball stemmer
 pg_catalog | synonym   | synonym dictionary: replace word by its synonym
 pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution
```

12.11. 限制

目前，PostgreSQL 的文本检索功能存在以下限制：

- 每个词位的长度必须小于 2K 字节
- `tsvector` 的长度（词位加位置）必须小于 1 兆字节
- 词位数量必须小于 2^{64}
- `tsvector` 中的位置值必须大于 0 且不超过 16,383
- <*N*>（FOLLOWED BY）`tsquery` 操作符中的匹配距离不能超过 16,384
- 每个词位的位置数不能超过 256 个
- `tsquery` 中的节点数（词位加操作符）必须小于 32,768

为了对比，PostgreSQL 8.1 的文档包含 10,441 个不同词，总数 335,420 个词，并且最频繁的词 “postgresql” 在 655 个文档中被提到 6,127 次。

另一个示例 – PostgreSQL 的邮件列表归档在 461,020 条消息的 57,491,343 个词位中包含 910,989 个不同词。

第 13 章 并发控制

本章描述 PostgreSQL 数据库系统在两个或更多会话试图同时访问同一数据时的行为。在这种情况下，目标是在保持严格数据完整性的同时，为所有会话提供高效访问。每一个数据库应用开发者都应该熟悉本章涉及的主题。

13.1. 简介

PostgreSQL 为开发者提供了一组丰富的工具来管理对数据的并发访问。在内部，数据一致性是通过使用多版本模型（多版本并发控制，MVCC）来维护的。这意味着，每条 SQL 语句看到的都是某个较早时刻的数据快照（即一个数据库版本），而不受底层数据当前状态的影响。这样可以防止语句看到由并发事务在同一数据行上执行更新所产生的不一致数据，从而为每个数据库会话提供事务隔离。MVCC 摒弃了传统数据库系统的加锁方法，尽量减少锁争用，以便在多用户环境中获得合理的性能。

使用 MVCC 并发控制模型而不是加锁的主要优点在于，在 MVCC 中，为查询（读取）数据而获得的锁不会与为写入数据而获得的锁冲突，因此读取不会阻塞写入，写入也不会阻塞读取。PostgreSQL 甚至在通过创新性的可串行化快照隔离（SSI）级别提供最严格的事务隔离时，也保持了这一保证。

PostgreSQL 也提供表级和行级锁设施，供那些通常不需要完整事务隔离、并且倾向于显式管理特定冲突点的应用使用。不过，正确使用 MVCC 通常会比加锁提供更好的性能。此外，由应用定义的咨询锁还提供了一种获取不依附于单个事务的锁的机制。

13.2. 事务隔离

SQL 标准定义了四个事务隔离级别。最严格的是可串行化，标准将它定义为：一组可串行化事务的任意并发执行，都保证与按照某种顺序一次只运行一个事务的效果相同。其他三个级别则按现象定义，这些现象源自并发事务之间的相互作用，而在各自级别上这些现象不得发生。标准指出，由于可串行化的定义，这些现象在该级别下都不可能出现。（这并不奇怪，如果事务的效果必须与一次只运行一个事务保持一致，又怎么可能看到由相互作用引起的任何现象呢？）

在各个级别上被禁止的现象有：

脏读

一个事务读取了由另一个并发未提交事务写入的数据。

不可重复读

一个事务重新读取它之前读过的数据，发现这些数据已经被另一个事务修改（且该事务已在初次读取之后提交）。

幻读

一个事务重新执行一个返回满足某个搜索条件的一组行的查询，却发现由于另一个最近提交的事务，满足该条件的行集已经发生了变化。

串行化异常

一组事务成功提交后的结果，与将这些事务逐个按某种顺序执行的任何可能结果都不一致。

SQL 标准以及 PostgreSQL 实现的事务隔离级别见表 13.1。

表 13.1. 事务隔离级别

隔离级别	脏读	不可重复读	幻读	串行化异常
读未提交	允许，但 PostgreSQL 中不会发生	可能	可能	可能
读已提交	不可能	可能	可能	可能
可重复读	不可能	不可能	允许，但 PostgreSQL 中不会发生	可能
可串行化	不可能	不可能	不可能	不可能

在 PostgreSQL 中，你可以请求四种标准事务隔离级别中的任意一种，但在内部只实现了三种不同的隔离级别；也就是说，PostgreSQL 的读未提交模式与读已提交模式行为相同。这是因为，要把标准隔离级别映射到 PostgreSQL 的多版本并发控制架构上，这是唯一合理的做法。

该表还显示，PostgreSQL 的可重复读实现不允许幻读。这在 SQL 标准下是可以接受的，因为标准规定的是某些隔离级别下哪些异常不得发生；提供更高的保证也是允许的。可用隔离级别的行为将在下面各小节中详细说明。

要设置事务的事务隔离级别，请使用命令 SET TRANSACTION。

> **重要**
>
> 某些 PostgreSQL 数据类型和函数在事务行为方面有特殊规则。特别是，对序列的修改（以及使用 `serial` 声明的列所对应的计数器）会立即对所有其他事务可见，并且即使执行该修改的事务中止，这些修改也不会被回滚。见第 9.16 节和第 8.1.4 节。

13.2.1. 读已提交隔离级别

读已提交是 PostgreSQL 中默认的隔离级别。当事务使用这一隔离级别时，`SELECT` 查询（不带 `FOR UPDATE/SHARE` 子句）只能看到查询开始之前已经提交的数据；它既看不到未提交的数据，也看不到查询执行期间并发事务提交的更改。实际上，`SELECT` 查询看到的是该查询开始运行瞬间的数据库快照。不过，`SELECT` 能看到其自身事务中先前执行的更新效果，即使这些更新尚未提交。还要注意，即使两个连续的 `SELECT` 命令位于同一个事务中，如果其他事务在第一个 `SELECT` 开始之后、第二个 `SELECT` 开始之前提交了更改，这两个命令也可能看到不同的数据。

`UPDATE`、`DELETE`、`SELECT FOR UPDATE` 和 `SELECT FOR SHARE` 命令在搜索目标行时的行为与 `SELECT` 相同：它们只会找到在命令开始时已经提交的目标行。不过，当找到这样的目标行时，它可能已经被其他并发事务更新、删除或者锁定。在这种情况下，即将执行更新的事务会等待第一个更新事务提交或回滚（如果它仍在进行中）。如果第一个更新事务回滚，那么它的作用将被撤销，第二个更新事务就可以继续更新最初找到的行。如果第一个更新事务提交了，那么若该行已被第一个更新者删除，第二个更新事务就会忽略该行；否则，第二个更新者将尝试在该行的已更新版本上应用自己的操作。该命令的搜索条件（`WHERE` 子句）会被重新计算，以判断该行的已更新版本是否仍然符合搜索条件。如果符合，则第二个更新者会基于该行的已更新版本继续执行操作。对于 `SELECT FOR UPDATE` 和 `SELECT FOR SHARE`，这意味着被锁住并返回给客户端的是该行的已更新版本。

带有 `ON CONFLICT DO UPDATE` 子句的 `INSERT` 的行为类似。在读已提交模式下，每个拟插入的行要么插入成功，要么转而更新；除非出现无关错误，否则这两种结果之一是有保证的。如果冲突来自另一个事务，而其效果对 `INSERT` 尚不可见，则 `UPDATE` 子句仍将作用于该行，即使从通常意义上说，该命令可能看不到该行的任何版本。

带有 ON CONFLICT DO NOTHING 子句的 INSERT 也可能因为另一个事务的结果而不插入某一行，即使该事务的效果对 INSERT 的快照不可见。同样，这种情况只会出现在读已提交模式下。

由于上述规则，更新命令可能看到一个不一致的快照：它能够看到并发更新命令在它试图更新的同一行上的效果，却看不到这些命令对数据库中其他行的影响。这种行为使得读已提交模式不适合涉及复杂搜索条件的命令；不过，对于较简单的场景它恰到好处。例如，考虑以如下事务更新银行余额：

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

如果两个这样的事务并发地尝试更改账户 12345 的余额，我们显然希望第二个事务从该账户行的已更新版本开始。因为每个命令只影响一个预先确定的行，让它看到该行的已更新版本不会造成任何麻烦的不一致。

在读已提交模式下，更复杂的用法可能产生不理想的结果。例如，考虑一个 DELETE 命令，另一个命令正在修改数据，使某些行开始满足其筛选条件、另一些行不再满足。假设 website 是一个有两行的表，其中 website.hits 分别等于 9 和 10：

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- 从另一个会话运行：  DELETE FROM website WHERE hits = 10;
COMMIT;
```

该 DELETE 将不会产生任何效果，即使在 UPDATE 之前和之后都存在一行 website.hits = 10。这是因为更新前值为 9 的那一行被跳过了，而当 UPDATE 完成并且 DELETE 获得锁时，新行值已经不再是 10 而是 11，因此不再匹配条件。

因为在读已提交模式中，每个命令都从一个新快照开始，而该快照包含截至当时已提交的所有事务，所以同一事务中的后续命令无论如何都会看到并发事务已提交的效果。上面真正的问题在于，单个命令是否能看到数据库的绝对一致视图。

读已提交模式所提供的部分事务隔离对于许多应用来说已经足够，而且这种模式既快速又容易使用。不过，它并不能满足所有场景。执行复杂查询和更新的应用，可能需要比读已提交模式所提供的更严格一致的数据库视图。

13.2.2. 可重复读隔离级别

可重复读隔离级别只能看到事务开始前已提交的数据；它看不到未提交的数据，也看不到事务执行期间并发事务提交的更改。（不过，每个查询都能看到其自身事务中先前执行的更新效果，即使这些更新尚未提交。）这比 SQL 标准对该隔离级别的要求更强，并且除了串行化异常之外，能够防止表 13.1 中描述的所有现象。正如前面提到的，这正是标准明确允许的，因为标准只描述每个隔离级别必须提供的最低保护。

这一点与读已提交不同：在可重复读事务中，查询看到的是该事务中第一条非事务控制语句开始时的快照，而不是事务中当前语句开始时的快照。因此，在单个事务中的连续 SELECT 命令会看到相同的数据，也就是说，它们不会看到在自身事务开始后由其他事务提交的更改。

使用这一隔离级别的应用必须准备好在发生串行化失败时重试事务。

UPDATE、DELETE、SELECT FOR UPDATE 和 SELECT FOR SHARE 命令在搜索目标行方面与 SELECT 相同：它们只会找到在事务开始时已提交的目标行。然而，这样的目标行可能在找到时已被另一个并发事务更新（或删除或锁定）。在这种情况下，可重复读事务将等待第一个更新事务提交或回滚（如果仍在进行）。如果第一个更新者回滚，则其效果将被取消，可重复读事务可以继续更新最初找到的行。但如果第一个更新者提交了（并实际更新或删除了行，而不仅仅是锁定了它），那么可重复读事务将被回滚，并显示以下消息：

```
ERROR:  could not serialize access due to concurrent update
```

因为可重复读事务不能修改或锁定在其开始之后已被其他事务更改的行。

当应用收到这条错误消息时，应当中止当前事务，并从头重试整个事务。第二次执行时，该事务会把先前已提交的更改视为其初始数据库视图的一部分，因此以该行的新版本作为新事务更新的起点时，就不存在逻辑冲突。

注意，只有更新事务才可能需要重试；只读事务永远不会发生串行化冲突。

可重复读模式提供了严格的保证，即每个事务都看到数据库的一个完全稳定的视图。不过，这个视图并不一定总能与同一级别并发事务的某种串行（一次一个）执行保持一致。例如，即使是该级别的只读事务，也可能看到一条表明某批次已完成的控制记录，却看不到逻辑上属于该批次的某条明细记录，因为创建该明细记录的事务读取了控制记录的较早版本。如果不仔细使用显式锁来阻塞冲突事务，试图依靠运行在这一隔离级别的事务来强制业务规则，往往无法正确工作。

可重复读隔离级别是通过一种技术实现的，这种技术在学术数据库文献中以及某些其他数据库产品中被称为快照隔离。与使用会降低并发性的传统加锁技术的系统相比，其行为和性能可能会表现出差异。有些其他系统甚至把可重复读和快照隔离作为两个行为不同的独立隔离级别提供。用来区分这两种技术的允许现象，直到 SQL 标准制定之后才被数据库研究人员正式定义，并且超出了本手册的范围。完整讨论请参阅[berenson95]。

> **注意**
>
> 在 PostgreSQL 9.1 之前，请求可串行化事务隔离级别会得到与这里描述完全相同的行为。若要保留旧式的可串行化行为，现在应请求可重复读。

13.2.3. 可串行化隔离级别

可串行化隔离级别提供最严格的事务隔离。该级别为所有已提交事务模拟串行执行；也就是说，效果就像这些事务不是并发执行，而是一个接一个地串行执行一样。不过，与可重复读级别一样，使用该级别的应用必须准备好在发生串行化失败时重试事务。实际上，这个隔离级别的工作方式与可重复读完全相同，只是它还会监视那些可能使一组并发可串行化事务的执行结果，与这些事务任何可能的串行（一次一个）执行都不一致的条件。这种监视不会引入超出可重复读已有阻塞之外的任何阻塞，但会带来一定开销；一旦检测到可能导致串行化异常的条件，就会触发串行化失败。

例如，考虑表mytab，它最初包含以下内容：

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

假设可串行化事务 A 执行以下计算：

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

然后将结果（30）作为新行的 value 插入，且该行满足class = 2。与此同时，可串行化事务 B 执行以下计算：

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

得到结果 300，并将它插入一个新行，该行满足class = 1。然后两个事务都尝试提交。如果任一事务运行在可重复读隔离级别，两个事务都可以提交；但由于不存在与该结果一致的串行执行顺序，使用可串行化事务时，只会允许一个事务提交，另一个则会被回滚，并收到以下消息：

```
ERROR:  could not serialize access due to read/write dependencies among transactions
```

这是因为，如果 A 在 B 之前执行，B 算出的总和应该是 330，而不是 300；同样，按另一种顺序执行，A 算出的总和也会不同。

当依赖可串行化事务来防止异常时，重要的是：从永久用户表读取的任何数据，在读取它的事务成功提交之前，都不应视为有效。即使对于只读事务也是如此；唯一的例外是可延迟只读事务中读取的数据，它一经读出就可视为有效，因为这种事务会等到能够获取一个保证不存在此类问题的快照后才开始读取数据。在所有其他情况下，应用不能依赖后来被中止事务中读到的结果；相反，应重试事务直到成功。

为了保证真正的可串行性，PostgreSQL 使用了谓词锁，也就是说，系统会保留一些锁，以便判断某个写操作如果先发生，是否会影响并发事务先前读取的结果。在 PostgreSQL 中，这些锁不会造成任何阻塞，因此不会参与形成死锁。它们用于识别并标记并发可串行化事务之间的依赖关系，而这些依赖在某些组合下可能导致串行化异常。相比之下，想要保证数据一致性的读已提交或可重复读事务，可能需要在整张表上加锁，这会阻塞其他试图使用该表的用户；或者它可能需要使用 SELECT FOR UPDATE 或 SELECT FOR SHARE，而这些做法不仅可能阻塞其他事务，还会造成磁盘访问。

与大多数其他数据库系统一样，PostgreSQL 中的谓词锁以事务实际访问的数据为基础。这些锁会出现在 pg_locks 系统视图中，其 mode 为 SIReadLock。查询执行期间获取的具体锁取决于查询所使用的计划，并且在事务过程中，多个更细粒度的锁（如元组锁）可能会合并成较少的粗粒度锁（如页锁），以防耗尽用于跟踪锁的内存。如果某个 READ ONLY 事务检测到不再可能发生会导致串行化异常的冲突，它可以在完成前释放其 SIRead 锁。事实上，READ ONLY 事务通常在启动时就能确定这一点，从而避免获取任何谓词锁。如果你显式请求一个 SERIALIZABLE READ ONLY DEFERRABLE 事务，它会阻塞，直到能够确立这一事实为止。（这是唯一一种可串行化事务会阻塞而可重复读事务不会阻塞的情况。）另一方面，SIRead 锁往往需要保留到事务提交之后，直到与之重叠的读写事务完成。

统一采用可串行化事务可以简化开发。任何一组成功提交的并发可串行化事务，都保证与把它们一次一个运行的效果相同；这意味着，如果你能够证明某个事务在单独运行时会做正确的事，那么即使不了解其他事务会做什么，也可以相信它在任何可串行化事务的混合执行中也会做正确的事，否则它就不会成功提交。重要的是，采用这种技术的环境应当有一套通用机制来处理串行化失败（其 SQLSTATE 值总是 '40001'），因为很难准确预测究竟哪些事务会对读/写依赖关系有贡献，并因此需要被回滚来防止串行化异常。监视读/写依赖关系会带来开销，被串行化失败中止的事务重新启动也会带来开销；但在把这些开销与显式锁以及 `SELECT FOR UPDATE` 或 `SELECT FOR SHARE` 所涉及的成本和阻塞进行权衡之后，可串行化事务在某些环境中仍然是性能最佳的选择。

虽然 PostgreSQL 的可串行化事务隔离级别只允许那些能够证明存在等价串行执行顺序的并发事务提交，但它并不总能阻止某些在真正串行执行中不会出现的错误被报告。特别是，即使在尝试插入某个键之前已经显式检查过该键不存在，仍然可能看到由于重叠执行的可串行化事务引起的唯一约束违例。要避免这种问题，必须确保所有插入潜在冲突键的可串行化事务，都先显式检查自己是否可以这样做。例如，设想一个要求用户输入新键的应用，它会先尝试查询用户给定的键以检查其是否已经存在，或者通过选取当前最大键再加一来生成新键。如果某些可串行化事务不遵循这一协议而直接插入新键，那么即使在这些并发事务的串行执行中本不会发生唯一约束违例，系统仍可能报告唯一约束违例。

当依赖可串行化事务进行并发控制时，为了获得最佳性能，应考虑以下问题：

- 在可能时声明事务为 `READ ONLY`。
- 控制活动连接的数量，并在需要时使用连接池。这始终是重要的性能考量，但在使用可串行化事务的繁忙系统中尤为重要。
- 单个事务中只包含为保证完整性所必需的内容。
- 不要让连接不必要地“闲置在事务中”。配置参数 idle_in_transaction_session_timeout 可用于自动断开这类拖延的会话。
- 在那些已经由可串行化事务自动提供保护的地方，消除不再需要的显式锁、`SELECT FOR UPDATE` 和 `SELECT FOR SHARE`。
- 当系统因谓词锁表内存不足而被迫把多个页级谓词锁合并为单个关系级谓词锁时，串行化失败的比例可能会上升。你可以通过增加 max_pred_locks_per_transaction、max_pred_locks_per_relation 和/或 max_pred_locks_per_page 来避免这种情况。
- 顺序扫描总是需要关系级谓词锁。这可能导致串行化失败的比例上升。通过降低 random_page_cost 和/或提高 cpu_tuple_cost 来鼓励使用索引扫描，可能对此有所帮助。务必在事务回滚和重启次数减少所带来的收益，与查询执行时间整体变化之间进行权衡。

可串行化隔离级别是通过一种在学术数据库文献中称为可串行化快照隔离的技术实现的，它在快照隔离的基础上增加了对串行化异常的检查。与使用传统加锁技术的其他系统相比，可能会观察到行为和性能方面的一些差异。详细信息请参阅[ports12]。

13.3. 显式锁定

PostgreSQL 提供了多种锁模式，用于控制对表中数据的并发访问。在 MVCC 无法给出期望行为的场景中，这些模式可用于由应用自行控制的加锁。此外，大多数 PostgreSQL 命令也会自动获取适当模式的锁，以确保在命令执行期间，被引用的表不会以不兼容的方式被删除或修改。（例如，`TRUNCATE` 无法安全地与同一张表上的其他操作并发执行，因此它会在该表上获取 `ACCESS EXCLUSIVE` 锁来强制实现这一点。）

要查看数据库服务器中当前尚未释放的锁列表，可以使用 pg_locks 系统视图。有关监控锁管理器子系统状态的更多信息，请参见第 28 章。

13.3.1. 表级锁

下面的列表给出了可用的锁模式，以及它们在 PostgreSQL 中被自动使用的场景。你也可以通过命令 LOCK 显式获取其中任意一种锁。请记住，这些锁模式全部都是表级锁，即使名称中包含“row”一词；这些名称只是历史遗留。在某种程度上，这些名称反映了各锁模式的典型用途 – 但它们的语义是完全相同的。一个锁模式与另一个锁模式真正的区别，只在于它与哪些锁模式冲突（见表 13.2）。两个事务不能在同一时刻在同一张表上持有相互冲突模式的锁。（不过，事务永远不会与自己冲突。例如，它可能先获取 ACCESS EXCLUSIVE 锁，随后又在同一张表上获取 ACCESS SHARE 锁。）不冲突的锁模式可以被多个事务同时持有。特别要注意，有些锁模式与自身冲突（例如，一次只能有一个事务持有 ACCESS EXCLUSIVE 锁），而另一些锁模式并不与自身冲突（例如，可以有多个事务持有 ACCESS SHARE 锁）。

表级锁模式

ACCESS SHARE (AccessShareLock)

只与 ACCESS EXCLUSIVE 锁模式冲突。

SELECT 命令会在被引用的表上获取这种模式的锁。通常，任何只读取表而不修改它的查询都会获取这种锁模式。

ROW SHARE (RowShareLock)

与 EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。

SELECT FOR UPDATE 和 SELECT FOR SHARE 命令会在目标表上获取这种模式的锁（此外，对于引用到但没有用 FOR UPDATE/FOR SHARE 选取的其他表，还会获取 ACCESS SHARE 锁）。

ROW EXCLUSIVE (RowExclusiveLock)

与 SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。

UPDATE、DELETE 和 INSERT 命令会在目标表上获取这种锁模式（此外，对任何其他被引用的表还会获取 ACCESS SHARE 锁）。通常，任何修改表中数据的命令都会获取这种锁模式。

SHARE UPDATE EXCLUSIVE (ShareUpdateExclusiveLock)

与 SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。这种模式用于保护表不受并发模式更改和 VACUUM 运行的影响。

由 VACUUM（不带 FULL）、ANALYZE、CREATE INDEX CONCURRENTLY、CREATE STATISTICS、COMMENT ON、ALTER TABLE VALIDATE 和其他 ALTER TABLE 变体获取（完整细节见 ALTER TABLE）。

SHARE (ShareLock)

与 ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。这种模式用于保护表不受并发数据更改的影响。

由 CREATE INDEX（不带 CONCURRENTLY）获取。

SHARE ROW EXCLUSIVE (ShareRowExclusiveLock)

与 ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。这种模式用于保护表不受并发数据更改的影响，并且是自排他的，因此同一时刻只能有一个会话持有它。

由 CREATE COLLATION、CREATE TRIGGER 和多种形式的 ALTER TABLE 获取（见 ALTER TABLE）。

EXCLUSIVE (ExclusiveLock)

与 ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。这种模式只允许并发的 ACCESS SHARE 锁，也就是说，只有对该表的读操作可以与持有这种锁模式的事务并行执行。

由 REFRESH MATERIALIZED VIEW CONCURRENTLY 获取。

ACCESS EXCLUSIVE (AccessExclusiveLock)

与所有模式的锁冲突（ACCESS SHARE、ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE）。这种模式保证持有者是以任何方式访问该表的唯一事务。

由 DROP TABLE、TRUNCATE、REINDEX、CLUSTER、VACUUM FULL 和不带 CONCURRENTLY 的 REFRESH MATERIALIZED VIEW 命令获取。多种形式的 ALTER TABLE 也会获取这一层级的锁。这也是未显式指定模式的 LOCK TABLE 语句的默认锁模式。

> **提示**
>
> 只有 ACCESS EXCLUSIVE 锁才会阻塞 SELECT（不带 FOR UPDATE/SHARE）语句。

锁一旦被获取，通常会一直持有到事务结束。但是，如果锁是在建立保存点之后才获取的，那么在回滚到该保存点时，这个锁会立即释放。这与 ROLLBACK 会取消保存点之后所有命令效果的原则是一致的。相同的原则也适用于在 PL/pgSQL 异常块中获取的锁：从该块中因错误跳出时，会释放在块中获取的锁。

表 13.2. 冲突的锁模式

请求的锁模式	当前的锁模式							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X

请求的锁模式	当前的锁模式							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

13.3.2. 行级锁

除了表级锁之外，还有行级锁。下面列出了这些锁，以及 PostgreSQL 会在哪些场景下自动使用它们。行级锁的完整冲突表见表 13.3。请注意，一个事务可以在同一行上持有彼此冲突的锁，甚至可以出现在不同子事务中；除此之外，两个事务不可能在同一行上持有彼此冲突的锁。行级锁不会影响数据查询；它们只会阻塞对同一行的写入者和加锁者。与表级锁一样，行级锁会在事务结束时或回滚到保存点时释放。

行级锁模式

`FOR UPDATE`

> `FOR UPDATE` 会导致 `SELECT` 语句检索到的行像将要被更新一样被锁定。这会阻止它们在当前事务结束之前被其他事务锁定、修改或删除。也就是说，其他尝试对这些行执行 `UPDATE`、`DELETE`、`SELECT FOR UPDATE`、`SELECT FOR NO KEY UPDATE`、`SELECT FOR SHARE` 或 `SELECT FOR KEY SHARE` 的事务，都会被阻塞直到当前事务结束。反过来，`SELECT FOR UPDATE` 也会等待已经在同一行上执行过上述任一命令的并发事务，并随后锁定并返回更新后的那一行（或者不返回任何行，因为该行可能已被删除）。不过，在 `REPEATABLE READ` 或 `SERIALIZABLE` 事务中，如果要锁定的行自事务开始以来已经发生变化，就会抛出错误。进一步讨论见第 13.4 节。
>
> `FOR UPDATE` 锁模式也会被对某一行的任何 `DELETE` 获取，以及会修改某些列值的 `UPDATE` 获取。目前，就 `UPDATE` 而言，这里考虑的列是指其上存在可供外键使用的唯一索引的列（因此不考虑部分索引和表达式索引），但这在将来可能会变化。

`FOR NO KEY UPDATE`

> 其行为类似于 `FOR UPDATE`，但获取的锁较弱：这种锁不会阻塞试图在同一行上获取锁的 `SELECT FOR KEY SHARE` 命令。凡是不会获得 `FOR UPDATE` 锁的 `UPDATE`，都会获得这种锁模式。

`FOR SHARE`

行为与 `FOR NO KEY UPDATE` 类似，不过它在每个检索到的行上获取的是共享锁而不是排他锁。共享锁会阻塞其他事务在这些行上执行 `UPDATE`、`DELETE`、`SELECT FOR UPDATE` 或 `SELECT FOR NO KEY UPDATE`，但不会阻止它们执行 `SELECT FOR SHARE` 或 `SELECT FOR KEY SHARE`。

`FOR KEY SHARE`

行为与 `FOR SHARE` 类似，不过锁更弱：`SELECT FOR UPDATE` 会被阻塞，但 `SELECT FOR NO KEY UPDATE` 不会。键共享锁会阻塞其他事务执行 `DELETE` 或任何会修改键值的 `UPDATE`，但不会阻塞其他 `UPDATE`，也不会阻止 `SELECT FOR NO KEY UPDATE`、`SELECT FOR SHARE` 或 `SELECT FOR KEY SHARE`。

PostgreSQL 不会在内存中保存任何关于已修改行的信息，因此一次加锁的行数没有限制。不过，锁定一行可能会导致一次磁盘写；例如，`SELECT FOR UPDATE` 会修改被选中的行以标记它们已被锁定，因此会产生磁盘写入。

表 13.3. 冲突的行级锁

请求的锁模式	当前的锁模式			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

13.3.3. 页级锁

除了表级锁和行级锁之外，还使用页级共享/排他锁来控制对共享缓冲池中表页面的读写访问。这些锁会在取出或更新某一行后立即释放。应用开发者通常不需要关心页级锁，这里提到它们只是为了完整性。

13.3.4. 死锁

使用显式锁可能会增加死锁的发生概率。所谓死锁，是指两个（或更多）事务各自持有对方想要的锁。例如，如果事务 1 在表 A 上获得了排他锁，然后试图再获取表 B 上的排他锁，而事务 2 已经持有表 B 上的排他锁，现在又想获取表 A 上的排他锁，那么两者都无法继续进行。PostgreSQL 会自动检测死锁，并通过中止其中一个事务来解决这个问题，使其他事务得以继续完成。（具体会中止哪个事务很难预测，也不应依赖这种预测。）

还要注意，死锁也可能由于行级锁而发生（因此，即使没有使用显式锁，它们也可能出现）。考虑下面这种情况：两个并发事务都在修改一个表。第一个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

这会在指定账号的那一行上获取一个行级锁。然后第二个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

第一条 `UPDATE` 语句成功地在指定行上获取了一个行级锁，因此它成功更新了那一行。然而，第二条 `UPDATE` 语句发现它试图更新的行已经被锁住了，于是它等待持有该锁的事务结束。此时，事务二正在等待事务一结束之后才能继续执行。现在，事务一执行：

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

事务一试图在指定行上获取一个行级锁，但它做不到，因为事务二已经持有了这个锁。于是它必须等待事务二完成。这样一来，事务一被事务二阻塞，而事务二又被事务一阻塞：死锁条件形成了。PostgreSQL 会检测到这种情况，并中止其中一个事务。

防止死锁的最好办法通常是避免它们出现，也就是确保所有使用同一数据库的应用都以一致的顺序在多个对象上获取锁。在上面的例子中，如果两个事务都是按相同顺序更新这些行，就不会发生死锁。还应确保事务在某个对象上获取的第一个锁，就是该对象所需的最严格锁模式。如果事先无法验证这一点，那么就应通过重试因死锁而中止的事务来动态处理死锁。

只要没有检测到死锁，寻求表级锁或行级锁的事务就会无限期等待冲突锁被释放。这意味着让应用长时间保持事务打开并不是好主意（例如等待用户输入时）。

13.3.5. 咨询锁

PostgreSQL 提供了一种创建由应用自行定义含义的锁的方法。这类锁被称为咨询锁，因为系统并不强制要求使用它们 – 是否正确使用完全取决于应用。咨询锁对于那些与 MVCC 模型不太契合的加锁策略非常有用。例如，咨询锁的一个常见用途是模拟所谓“平面文件”数据管理系统中典型的悲观锁策略。虽然也可以通过在表中存储一个标志位来达到相同目的，但咨询锁速度更快，可以避免表膨胀，并且会在会话结束时由服务器自动清理。

在 PostgreSQL 中获取咨询锁有两种方式：会话级和事务级。会话级咨询锁一旦获取，就会一直保持到显式释放或会话结束。不同于标准锁请求，会话级咨询锁请求不遵守事务语义：在随后回滚的事务中获取的锁，回滚后仍会保持；同样，即使发出解锁请求的事务后来失败，解锁依然有效。一个进程可以多次获取同一个锁；每成功一次加锁请求，都必须有对应的解锁请求，锁才会真正释放。另一方面，事务级锁请求更像普通锁请求：它们会在事务结束时自动释放，而且没有显式解锁操作。对于短期使用咨询锁，这种行为通常比会话级行为更方便。针对同一咨询锁标识符的会话级和事务级锁请求，会按预期相互阻塞。如果某个会话已经持有给定的咨询锁，那么它发出的附加请求总会成功，即使其他会话正在等待该锁；无论现有锁和新请求属于会话级还是事务级，都是如此。

和 PostgreSQL 中的所有锁一样，任何会话当前持有的咨询锁完整列表，都可以在`pg_locks` 系统视图中找到。

咨询锁和普通锁都存储在一个共享内存池中，其大小由配置变量max_locks_per_transaction 和 max_connections 定义。必须小心不要耗尽这部分内存，否则服务器将根本无法再授予任何锁。这也为服务器可授予的咨询锁数量设定了上限，具体通常在数万到数十万个之间，取决于服务器的配置。

在某些使用咨询锁的方法中，特别是涉及显式排序和 LIMIT 子句的查询，必须注意控制由于 SQL 表达式求值顺序而获取的锁。例如：

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- 可以
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- 危险!
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- 可以
```

在上面的查询中，第二种形式是危险的，因为不能保证在执行加锁函数之前先应用 LIMIT。这可能导致获取了一些应用并未预期的锁，因此应用也不会去释放它们（直到会话结束）。从应用的角度看，这类锁会变成悬而未清的锁，尽管仍然可以在 pg_locks 中看到。

用于操作咨询锁的函数见第 9.26.10 节。

13.4. 应用级别的数据一致性检查

用读已提交事务强制执行有关数据完整性的业务规则非常困难，因为数据视图会随每条语句而变化，而且一旦发生写冲突，即使是单条语句也未必局限于该语句自己的快照。

虽然可重复读事务在整个执行期间都具有稳定的数据视图，但在使用 MVCC 快照进行数据一致性检查时，还存在一个微妙问题，即所谓的读/写冲突。如果一个事务写入数据，而并发事务试图读取相同的数据（无论是在写入之前还是之后），它都看不到另一个事务的工作。于是，读取者看起来就像是先执行的一方，而不管究竟是谁先启动，也不管谁先提交。如果事情只停留在这一步，就没有问题；但如果读取者还写入了数据，而这些数据又被另一个并发事务读取，那么现在就会有一个事务看起来像是在前面提到的任一事务之前执行。如果看起来最后执行的事务实际上最先提交，就很容易在事务执行顺序图中形成一个环。一旦出现这样的环，不借助额外机制，完整性检查就无法正确工作。

正如第 13.2.3 节中提到的，可串行化事务其实就是在可重复读事务的基础上，增加了对危险读/写冲突模式的非阻塞监控。当检测到某种模式可能在表面执行顺序中形成一个环时，其中一个相关事务就会被回滚，以打破这个环。

13.4.1. 用可串行化事务强制一致性

如果对所有写操作，以及所有需要一致数据视图的读操作都使用可串行化事务隔离级别，那么不需要额外工作就能确保一致性。从其他环境移植而来、按使用可串行化事务来保证一致性而编写的软件，在这方面应当能够在 PostgreSQL 中“正常工作”。

使用这种技术时，如果应用软件通过某个框架运行，并由该框架自动重试因串行化失败而回滚的事务，就可以避免给应用程序员带来不必要的负担。把 default_transaction_isolation 设置为 serializable 可能是个好主意。通过在触发器中检查事务隔离级别来采取某些措施，以确保不会因为疏忽或为了绕过完整性检查而使用其他事务隔离级别，也是明智的。

性能方面的建议见第 13.2.3 节。

> **警告**
>
> 利用可串行化事务提供的这一层完整性保护，尚未扩展到热备模式（第 26.5 节）。因此，使用热备的用户可能希望在主库上使用可重复读和显式锁定。

13.4.2. 使用显式阻塞锁强制一致性

当存在非可串行化写入时，要确保某一行当前仍然有效，并保护它不受并发更新影响，就必须使用 SELECT FOR UPDATE、SELECT FOR SHARE 或适当的 LOCK TABLE 语句。（SELECT FOR UPDATE 和 SELECT FOR SHARE 只锁定返回的行以防止并发更新，而 LOCK TABLE 会锁住整张表。）从其他环境向 PostgreSQL 迁移应用时，应当考虑这一点。

对于从其他环境迁移而来的用户，还需要注意：SELECT FOR UPDATE 并不能保证并发事务不会更新或删除被选中的行。要在 PostgreSQL 中做到这一点，必须真正去更新该行，即使没有任何值需要改变。SELECT FOR UPDATE 只是临时阻塞其他事务，使它们不能获取同样的锁，也不能执行会影响被锁定行的 UPDATE 或 DELETE；但是一旦持有该锁的事务提交或回滚，被阻塞的事务就会继续执行冲突操作，除非在持锁期间已经对该行执行了实际的 UPDATE。

在非可串行化的 MVCC 环境下，全局有效性检查需要额外考虑。例如，一个银行应用可能希望检查一个表中的贷方金额总和等于另一个表中的借方金额总和，而这两个表都在被活跃更新。在读已提交模式下，比较两个连续的 SELECT sum(...) 命令的结果并不可靠，因为第二个查询很可能会包含第一个查询没有统计到的事务提交结果。在单个可重复读事务中完成这两次求和，只能准确反映在该可重复读事务开始之前已提交事务的效果 – 但等到结果交付时，人们完全可能合理地怀疑这个答案是否仍然相关。如果可重复读事务本身在尝试进行一致性检查之前已经应用了某些更改，那么这种检查的意义就更加值得商榷，因为此时它包含了事务开始后的一部分而不是全部更改。在这种情况下，谨慎的做法可能是锁定执行检查所需的所有表，以获得当前真实状态的无可争议图景。SHARE 模式（或更高）的锁能够保证，在被锁定的表中，除了当前事务自身的更改之外，没有其他未提交更改。

注意，如果依赖显式锁定来防止并发更改，就应当使用读已提交模式，或者在可重复读模式下小心地在执行查询之前先获取锁。可重复读事务获得的锁能够保证没有其他修改该表的事务仍在运行，但如果该事务看到的快照早于获取锁的时刻，那么它看到的快照也可能早于表中某些现在已经提交的更改。可重复读事务的快照实际上是在其第一条查询或数据修改命令（SELECT、INSERT、UPDATE、DELETE）开始时被冻结的，因此可以在快照冻结之前显式获取锁。

13.5. 注意事项

一些 DDL 命令（目前只有 TRUNCATE 和会重写表的 ALTER TABLE 形式）不是 MVCC 安全的。这意味着在截断或重写提交之后，如果并发事务使用的是在该 DDL 命令提交之前取得的快照，那么该表对它们来说会表现为空表。这只会对那些在 DDL 命令开始前没有访问过相关表的事务造成问题 – 任何在 DDL 命令开始前访问过该表的事务，都会持有至少一个 ACCESS SHARE 表锁，从而阻塞该 DDL 命令直到该事务完成。因此，这些命令不会在针对目标表的连续查询中造成明显的表内容不一致，但它们可能会导致目标表内容与数据库中其他表内容之间出现可见的不一致。

对可串行化事务隔离级别的支持尚未添加到热备复制目标（见第 26.5 节）。当前热备模式下支持的最严格隔离级别是可重复读。虽然在主库上把所有永久数据库写操作都放在可串行化事务中执行，能够确保所有备库最终达到一致状态，但在备库上运行的可重复读事务，有时仍可能看到一种瞬态状态，而这种状态与主库上事务的任何串行执行都不一致。

对系统目录的内部访问并不是按当前事务的隔离级别执行的。这意味着新创建的数据库对象（例如表）对于并发的可重复读和可串行化事务是可见的，即使这些对象中包含的行并不可见。相反，在较高隔离级别下，显式检查系统目录的查询不会看到表示并发创建数据库对象的那些行。

13.6. 锁定和索引

尽管 PostgreSQL 为表数据提供了非阻塞的读写访问，但 PostgreSQL 当前实现的索引访问方法并不是每一种都能提供非阻塞的读写访问。各种索引类型的处理方式如下：

B-树、GiST 和 SP-GiST 索引

读写访问使用短期的页级共享/排他锁。每个索引行在被取出或插入之后，锁都会立即释放。这些索引类型提供最高的并发性，并且不会产生死锁。

Hash 索引

读写访问使用的是 Hash 桶级共享/排他锁。锁会在整个 Hash 桶处理完成后释放。桶级锁比索引级锁具有更好的并发性，但也可能产生死锁，因为它们的持有时间比一次索引操作更长。

GIN 索引

读写访问使用短期的页级共享/排他锁。每个索引行在被取出或插入之后，锁都会立即释放。但要注意，插入一个使用 GIN 索引的值通常会导致每行产生多个索引键插入，因此 GIN 可能为了插入单个值而做大量工作。

目前，B-树索引为并发应用提供了最佳性能；由于它们还比 Hash 索引拥有更多特性，因此对于需要为标量数据建立索引的并发应用，推荐使用 B-树索引类型。处理非标量数据时，B-树就不再适用，此时应改用 GiST、SP-GiST 或 GIN 索引。

第 14 章 性能提示

查询性能可能受许多因素影响。其中一些因素可以由用户控制，另一些则属于系统底层设计的基本特性。本章提供一些帮助理解和调优PostgreSQL 性能的提示。

14.1. 使用 EXPLAIN

PostgreSQL 会为收到的每个查询制定一个查询计划。选择与查询结构和数据特性相匹配的正确计划，对获得良好的性能至关重要，因此系统内置了一个复杂的规划器来尽量选出好的计划。可以使用 EXPLAIN 命令查看规划器为任意查询生成的查询计划。读懂计划是一门需要经验积累的技能，本节将尝试介绍其中的基础知识。

本节中的示例取自执行过 VACUUM ANALYZE 的回归测试数据库，使用的是 9.3 开发版本源码。如果自行尝试这些示例，通常应当能得到相近的结果，但估计代价和行计数可能略有差异，因为 ANALYZE 生成的统计信息来自随机采样而非精确计数，而且代价本身在某种程度上也依赖于平台。

这些示例使用 EXPLAIN 默认的 “text” 输出格式，它紧凑且便于人工阅读。如果希望将EXPLAIN 的输出交给程序做进一步分析，则应改用机器可读的输出格式（XML、JSON 或 YAML）。

14.1.1. EXPLAIN 基础

查询计划的结构是一棵由计划节点组成的树。树的最底层是扫描节点，它们从表中返回原始行。不同的表访问方法对应不同类型的扫描节点，例如顺序扫描、索引扫描和位图索引扫描。也有一些并非表的行来源，例如VALUES 子句和 FROM 中的返回集合函数，它们也各自有对应的扫描节点类型。如果查询需要对原始行执行连接、聚合、排序或其他操作，那么扫描节点之上还会出现额外的节点来完成这些操作。同样，这些操作通常不止一种实现方式，因此这里也会出现不同的节点类型。EXPLAIN 会为计划树中的每个节点输出一行，显示基本节点类型以及规划器对该计划节点执行代价的估计值。还可能出现相对节点摘要行缩进的附加行，用来显示该节点的更多属性。第一行，也就是最顶层节点的摘要行，给出了整个计划的估计总执行代价；规划器力图最小化的正是这个数字。

下面是一个简单的示例，仅用于展示输出的样子：

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

由于这个查询没有 WHERE 子句，它必须扫描表中的所有行，因此规划器选择了一个简单的顺序扫描计划。圆括号中的数字从左到右依次表示：

- 估计启动代价。这是输出阶段开始前需要消耗的时间，例如排序节点执行排序所需的时间。
- 估计总代价。这里假定该计划节点会运行到结束，也就是取回所有可用的行。实际中某个节点的父节点可能会在尚未读完所有可用行之前提前停止（见后文的LIMIT 示例）。
- 该计划节点输出行数的估计值。同样，也是假定该节点会运行到结束。
- 预计该计划节点输出行的平均宽度，以字节计。

这些代价使用由规划器代价参数决定的任意单位来衡量（见第 19.7.2 节）。传统上通常以磁盘页读取作为代价单位；也就是说，惯例上将seq_page_cost 设为 1.0，其他代价参数都相对它来设定。本节中的示例都使用默认代价参数。

需要理解的一点是，上层节点的代价包含了其所有子节点的代价。还要注意，这个代价只反映规划器关心的内容。特别是，它没有考虑将结果行传输给客户端所消耗的时间，而这在实际耗时中可能是重要因素；但规划器会忽略这项代价，因为它无法通过改变计划来影响它。（我们相信每个正确的计划都会输出相同的行集。）

rows 值有些容易误解，因为它不是计划节点处理或扫描过的行数，而是该节点输出的行数。由于在该节点上应用的WHERE 条件会过滤掉一部分扫描到的行，这个数字通常小于扫描行数。理想情况下，顶层的行数估计应当接近查询实际返回、更新或删除的行数。

回到我们的示例：

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

这些数字的来源非常直接。如果执行：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

可以看到 tenk1 有 358 个磁盘页和 10000 行。估计代价按如下公式计算：（读取的磁盘页数 * seq_page_cost）+（扫描的行数 * cpu_tuple_cost）。默认情况下，seq_page_cost 为 1.0，cpu_tuple_cost 为 0.01，因此估计代价就是 (358 * 1.0) + (10000 * 0.01) = 458。

现在给查询增加一个 WHERE 条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..470.00 rows=7000 width=244)
   Filter: (unique1 < 7000)
```

注意，EXPLAIN 输出显示，WHERE 子句作为附加在 Seq Scan 计划节点上的“过滤器”条件被应用。这意味着该计划节点会对扫描到的每一行检查该条件，只输出满足条件的那些行。由于有了 WHERE 子句，估计输出行数降低了；但扫描仍然必须访问全部 10000 行，所以代价没有下降。实际上它还略微上升了一些，准确地说是上升了 10000 * cpu_operator_cost，以反映额外检查 WHERE 条件所需的 CPU 时间。

这条查询实际选出的行数是 7000，但估计的rows 只是近似值。如果重复这个实验，很可能会得到略有不同的估计值。此外，由于ANALYZE 生成的统计信息来自该表的随机采样，这个估计值可能在每次执行ANALYZE 之后发生变化。

现在把条件收紧一些：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                 QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.06..224.98 rows=100 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0)
         Index Cond: (unique1 < 100)
```

这里规划器决定采用两步计划：子计划节点访问索引，找出满足索引条件的行位置，然后上层计划节点再从表中实际取回这些行。单独取回行比顺序读取昂贵得多，但因为不必访问表中的所有页面，这样仍然比顺序扫描便宜。（之所以采用两层计划，是因为上层计划节点会在读取前先将索引识别出的行位置按物理顺序整理，从而尽量降低零散取页的代价。节点名称中的“位图”就是完成这种整理的机制。）

现在向 `WHERE` 子句添加另一个条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                 QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

新增的条件 `stringu1 = 'xxx'` 降低了估计输出行数，却没有降低代价，因为仍然必须访问同一组行。注意，`stringu1` 子句不能用作索引条件，因为该索引只建立在`unique1` 列上。它只能作为过滤条件，应用到通过索引取出的行上。因此，代价实际上略有增加，以反映这项额外检查。

在某些情况下，规划器更倾向于一个“简单”索引扫描计划：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                 QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

在这种计划中，表行按索引顺序取出，因此读取代价更高，但行数很少，不值得为排序行位置付出额外代价。对于仅取出一行的查询，最常见到这种计划。它也经常用于带有以下条件的查询：`ORDER BY` 与索引顺序匹配，因为这时无需额外排序步骤就能满足`ORDER BY`。

如果在 WHERE 引用的多个列上分别建有索引，规划器可能选择将这些索引做 AND 或 OR 组合：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

但这需要访问两个索引，因此与只使用一个索引、把另一个条件作为过滤条件相比，并不一定更划算。如果调整条件中的范围，就会看到计划相应改变。

下面用一个示例说明 LIMIT：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

这与上面的查询相同，只是加上了LIMIT，因此不必检索全部行，规划器也就改变了选择。注意，Index Scan 节点的总代价和行计数显示得像是它会运行到结束一样；但 Limit 节点预计在只取到其中五分之一的行后就会停止，因此它的总代价也只有前者的五分之一，这才是该查询真正的估计代价。之所以更偏好这个计划，而不是在前一个计划之上再加一个 Limit 节点，是因为后者仍然无法避免位图扫描的启动代价，那样总代价仍会高于 25。

使用之前讨论的列，尝试连接两个表：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

在这个计划中，有一个嵌套循环连接节点，它的两个输入，也就是两个子节点，都是表扫描。节点摘要行的缩进反映了计划树结构。连接的第一个子节点，也就是“外侧”子节点，是一个与前面见过的位图扫描类似的节点。它的代价和行计数与SELECT ... WHERE unique1 < 10 得到的结果相同，因为WHERE 子句 unique1 < 10 正是在该节点上应用的。t1.unique2 = t2.unique2 子句此时还无关，因此不会影响外侧扫描的行计数。嵌套循环连接节点会对从外侧子节点得到的每一行执行一次第二个，也就是“内侧”子节点。当前外侧行中的列值可以代入内侧扫描；这里外侧行的 t1.unique2 值可用，因此得到的计划和代价与前面看到的简单SELECT ... WHERE t2.unique2 = *constant* 情形类似。（由于预期在对 t2 反复执行索引扫描期间会发生缓存命中，估计代价实际上比前面看到的略低一些。）随后，循环节点的代价建立在外侧

扫描代价之上，再加上每个外侧行都要执行一次内侧扫描的代价（这里是 10 * 7.91），以及少量连接处理的 CPU 时间。

本例中，连接的输出行数等于两个扫描行数的乘积，但并非所有情况都如此，因为可能还有额外的WHERE 子句同时涉及两个表，因此只能在连接处应用，不能应用到任一输入扫描。下面是一个示例：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

                                      QUERY PLAN
--------------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46 rows=10 width=244)
               Index Cond: (unique2 < 10)
```

条件 t1.hundred < t2.hundred 不能在 tenk2_unique2 索引中检验，因此应用在连接节点上。这会减少连接节点的估计输出行数，但不改变任一输入扫描。

注意，这里规划器通过在连接的内侧关系之上放置一个 Materialize 计划节点，选择将其“物化”。这意味着 t2 索引扫描只会执行一次，尽管嵌套循环连接节点需要读取那份数据十次，也就是外侧关系的每一行都要读取一次。Materialize 节点会在读取数据时将其保存在内存中，并在之后的每次遍历中从内存返回这些数据。

在处理外连接时，可能会看到连接计划节点同时带有“Join Filter”和普通“Filter”条件。Join Filter 条件来自外连接的 ON 子句，因此某一行即使未通过 Join Filter，仍可能作为一条补齐 null 的行被输出。但普通 Filter 条件是在外连接规则应用之后再执行的，因此会无条件移除行。在内连接中，这两类过滤条件在语义上没有区别。

如果稍微改变查询的选择率，可能得到完全不同的连接计划：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------------
 Hash Join  (cost=230.47..713.98 rows=101 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=229.20..229.20 rows=101 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
                     Index Cond: (unique1 < 100)
```

这里规划器选择了哈希连接：先把一个表的行放入内存中的 hash 表，然后扫描另一个表，并对其中每一行到 hash 表中查找匹配。再次注意缩进如何反映计划结构：tenk1 上的位图扫描是 Hash 节点的输入，Hash 节点据此构造 hash 表；随后该 hash 表被返回给 Hash Join 节点，后者从其外侧子计划读取行，并对每一行在 hash 表中进行查找。

另一种可能的连接类型是归并连接，如下所示：

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
         Sort Key: t2.unique2
         ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000 width=244)
```

归并连接要求输入数据按连接键排序。在这个计划中，tenk1 的数据通过索引扫描按正确顺序访问行来完成排序，而 onek 则更适合顺序扫描后再排序，因为需要访问该表中的更多行。（对大量行进行排序时，顺序扫描加排序往往优于索引扫描，因为索引扫描需要非顺序的磁盘访问。）

查看其他候选计划的一种方法，是使用第 19.7.1 节中介绍的启用/禁用标志，强制规划器忽略它认为代价最低的策略。（这是一种粗略但有用的工具。另请参见第 14.3 节。）例如，如果不确信顺序扫描加排序是处理上例中onek 表的最佳方法，可以尝试：

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79 rows=1000 width=244)
```

这表明规划器认为，通过索引扫描来排序onek，代价比顺序扫描加排序大约高 12%。当然，接下来的问题是它的判断是否正确。可以使用EXPLAIN ANALYZE 来调查，下文将详细说明。

14.1.2. EXPLAIN ANALYZE

可以使用 EXPLAIN 的 ANALYZE 选项来检查规划器估计值的准确性。启用该选项后，EXPLAIN 会实际执行查询，然后显示各个计划节点累计的真实行计数和真实运行时间，同时也保留普通 EXPLAIN 会显示的估计值。例如，可能会得到如下结果：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.50 rows=10 width=488) (actual time=0.017..0.051 rows=10.00 loops=
1)
   Buffers: shared hit=36 read=6
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.38 rows=10 width=244) (actual time=0.009..
0.017 rows=10.00 loops=1)
         Recheck Cond: (unique1 < 10)
         Heap Blocks: exact=10
         Buffers: shared hit=3 read=5 written=4
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0) (actual
 time=0.004..0.004 rows=10.00 loops=1)
               Index Cond: (unique1 < 10)
               Index Searches: 1
               Buffers: shared hit=2
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.90 rows=1 width=244) (actual
 time=0.003..0.003 rows=1.00 loops=10)
         Index Cond: (unique2 = t1.unique2)
         Index Searches: 10
         Buffers: shared hit=24 read=6
 Planning:
   Buffers: shared hit=15 dirtied=9
 Planning Time: 0.485 ms
 Execution Time: 0.073 ms
```

注意，“actual time” 值以毫秒的真实时间表示，而cost 估计值则以任意单位表示，因此二者通常不会直接对应。通常最值得关注的是估计行数是否与实际情况足够接近。在这个示例中，估计值完全命中，但在实践中这其实相当少见。

在某些查询计划中，一个子计划节点可能会执行多次。例如，上面那个嵌套循环计划中的内侧索引扫描会对外侧的每一行执行一次。在这种情况下，loops 值报告的是该节点的总执行次数，而 actual time 和 rows 显示的是每次执行的平均值。这样做是为了让这些数字更容易与代价估计的展示方式相比较。将它们乘以loops 值，就能得到该节点实际消耗的总时间。在上面的示例中，执行tenk2 上的索引扫描总共花费了 0.220 毫秒。

在某些情况下，EXPLAIN ANALYZE 会显示除计划节点执行时间和行计数之外的额外执行统计信息。例如，Sort 和 Hash 节点会提供更多信息：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------
---------------------------------------------
 Sort  (cost=713.05..713.30 rows=100 width=488) (actual time=2.995..3.002 rows=100.00 loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 74kB
   Buffers: shared hit=440
   ->  Hash Join  (cost=226.23..709.73 rows=100 width=488) (actual time=0.515..2.920 rows=100.
00 loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         Buffers: shared hit=437
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual time=0.026.
.1.790 rows=10000.00 loops=1)
               Buffers: shared hit=345
         ->  Hash  (cost=224.98..224.98 rows=100 width=244) (actual time=0.476..0.477 rows=100.
00 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 35kB
               Buffers: shared hit=92
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.06..224.98 rows=100 width=244) (actual
 time=0.030..0.450 rows=100.00 loops=1)
                     Recheck Cond: (unique1 < 100)
                     Heap Blocks: exact=90
                     Buffers: shared hit=92
                     ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0)
 (actual time=0.013..0.013 rows=100.00 loops=1)
                           Index Cond: (unique1 < 100)
                           Index Searches: 1
                           Buffers: shared hit=2
 Planning:
   Buffers: shared hit=12
 Planning Time: 0.187 ms
 Execution Time: 3.036 ms
```

Sort 节点会显示所用的排序方法，特别是排序是在内存中完成还是落到磁盘上，以及所需的内存或磁盘空间。Hash 节点则会显示 hash 桶和批次的数量，以及 hash 表使用内存的峰值。（如果批次数超过 1，还会涉及磁盘空间使用，但这里不会显示出来。）

另一种附加信息是被过滤条件排除的行数：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                QUERY PLAN
-------------------------------------------------------------------------------------------------
----------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual time=0.016..5.107 rows=7000
 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning time: 0.083 ms
 Execution time: 5.905 ms
```

对于应用在连接节点上的过滤条件，这些计数尤其有价值。只有当至少一个扫描行（对于连接节点，则是一个潜在连接行对）被过滤条件排除时，才会出现“Rows Removed”这一行。

在“有损”索引扫描中，也会出现类似于过滤条件的情况。例如，考虑下面这个查找包含指定点的多边形的查询：

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
-------
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual time=0.044..0.044 rows=0
 loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Planning time: 0.040 ms
 Execution time: 0.083 ms
```

规划器认为（而且完全正确）这个示例表太小，不值得使用索引扫描，因此得到的是普通顺序扫描，其中所有行都被过滤条件排除。但如果强制使用索引扫描，就会看到：

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                           QUERY PLAN
-----------------------------------------------------------------------------------------------------------
---------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=32) (actual time=0.
062..0.062 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Planning time: 0.034 ms
 Execution time: 0.144 ms
```

这里可以看到，索引返回了一个候选行，但随后在重新检查索引条件时被排除。这是因为 GiST 索引对于多边形包含关系测试是“有损”的：它实际返回的是包含与目标重叠的多边形的行，然后必须对这些行进行精确的包含关系测试。

EXPLAIN 有一个 BUFFERS 选项，可以与ANALYZE 一起使用，以获取更多运行时统计信息：

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------
---------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244) (actual time=0.323..0.342
 rows=10 loops=1)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Buffers: shared hit=15
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309 rows=0 loops=1)
         Buffers: shared hit=7
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual
 time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
               Buffers: shared hit=2
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0) (actual
 time=0.227..0.227 rows=999 loops=1)
               Index Cond: (unique2 > 9000)
               Buffers: shared hit=5
 Planning time: 0.088 ms
 Execution time: 0.423 ms
```

由 BUFFERS 提供的数值有助于找出查询中 I/O 最密集的部分。

请记住，由于EXPLAIN ANALYZE 会实际执行查询，任何副作用都会照常发生，即使查询原本输出的结果被丢弃，改为打印EXPLAIN 数据。如果想分析会修改数据的查询而不改变表，可以在之后回滚命令，例如：

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

                                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
---------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.628 rows=0
 loops=1)
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=0.101..0.
439 rows=100 loops=1)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual
 time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
 Planning time: 0.079 ms
 Execution time: 14.727 ms

ROLLBACK;
```

如本例所示，当查询是INSERT、UPDATE 或 DELETE 命令时，真正执行表修改的工作由顶层的 Insert、Update 或 Delete 计划节点完成。该节点下面的计划节点负责定位旧行和/或计算新数据。因此，上面看到的是前文已经介绍过的同类位图表扫描，它的输出被送入一个负责存储更新后行的 Update 节点。值得注意的是，虽然数据修改节点可能占用相当可观的运行时间（这里它消耗了大部分时间），但规划器目前不会为这部分工作额外增加任何代价估计。这是因为这项工作对每个正确的查询计划都相同，因此不会影响规划决策。

当一个 UPDATE 或 DELETE 命令影响到继承层次结构时，输出可能如下所示：

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Update on parent  (cost=0.00..24.53 rows=4 width=14)
   Update on parent
   Update on child1
   Update on child2
   Update on child3
   ->  Seq Scan on parent  (cost=0.00..0.00 rows=1 width=14)
         Filter: (f1 = 101)
   ->  Index Scan using child1_f1_key on child1  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child2_f1_key on child2  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child3_f1_key on child3  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
```

在本例中，Update 节点需要考虑三个子表，以及最初指定的父表。因此有四个输入扫描子计划，每个表一个。为清晰起见，Update 节点标注了将被更新的具体目标表，顺序与相应子计划一致。（这些标注是在PostgreSQL 9.5 中新增的；在更早的版本中，读者必须通过检查子计划来推断目标表。）

EXPLAIN ANALYZE 显示的 Planning time，是从已解析的查询生成查询计划并完成优化所花费的时间；其中不包括解析和重写。

EXPLAIN ANALYZE 显示的 Execution time 包括执行器启动和关闭所需的时间，以及运行所有已触发触发器所花费的时间，但不包括解析、重写和规划时间。若存在BEFORE 触发器，其执行时间会计入相关的 Insert、Update 或 Delete 节点；而AFTER 触发器的时间不会计入那里，因为AFTER 触发器是在整个计划完成之后才触发的。每个触发器，无论是BEFORE 还

是 AFTER，其总耗时也会单独显示。注意，延迟约束触发器要到事务结束时才会执行，因此 EXPLAIN ANALYZE 完全不会把它们计入其中。

14.1.3. 注意事项

EXPLAIN ANALYZE 测得的运行时间，可能以两种重要方式偏离同一查询的正常执行时间。第一，由于不会向客户端传递任何输出行，因此网络传输开销和 I/O 转换开销均不会被计入。第二，EXPLAIN ANALYZE 附加的测量开销本身可能相当可观，尤其是在操作系统调用 gettimeofday() 较慢的机器上。可以使用 pg_test_timing 工具来测量系统上的计时开销。

不应将 EXPLAIN 结果外推到与实际测试场景差异很大的情况。例如，在一个很小的表上得到的结果，不能假定也适用于大型表。规划器的代价估计不是线性的，因此它可能会为更大或更小的表选择不同的计划。一个极端示例是：对于只占用一个磁盘页的表，无论索引是否可用，几乎总会得到顺序扫描计划。规划器认识到，无论如何处理该表都需要一次磁盘页读取，因此再额外读取页面去查看索引并没有价值。（前面的polygon_tbl 示例已经展示过这种情况。）

在某些情况下，实际值和估计值不会很好地对应，但这并不表示出现了问题。其中一种情况是，计划节点的执行会被LIMIT 或类似效果提前截断。例如，在前面用过的LIMIT 查询中：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------
--------------------------------
 Limit  (cost=0.29..14.33 rows=2 width=244) (actual time=0.051..0.071 rows=2.00 loops=1)
   Buffers: shared hit=16
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..70.50 rows=10 width=244) (actual
 time=0.051..0.070 rows=2.00 loops=1)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
         Index Searches: 1
         Buffers: shared hit=16
 Planning Time: 0.077 ms
 Execution Time: 0.086 ms
```

Index Scan 节点的估计代价和行计数显示得像是它会运行到结束一样。但实际上，Limit 节点在拿到两行之后就停止请求更多行，因此实际行计数只有 2，运行时间也低于代价估计所暗示的水平。这并不是估计错误，只是估计值与真实值的显示方式不同而已。

归并连接也会产生一些容易误导人的计量现象。如果归并连接已经耗尽其中一个输入，且另一个输入的下一个键值大于已耗尽输入的最后一个键值，就会停止读取这个尚未耗尽的输入；在这种情况下，不可能再有匹配，因此不必扫描它的剩余部分。这会导致某个子节点没有被完整读取，其结果与前面提到的LIMIT 情况类似。此外，如果外侧（第一个）子节点包含带有重复键值的行，内侧（第二个）子节点会回退，并重新扫描与该键值匹配的那部分行。EXPLAIN ANALYZE 会把这些对同一内侧行的重复输出，统计得像是真实的额外行一样。当外侧存在大量重复值时，内侧子计划节点报告出的实际行计数，可能会明显大于内侧关系中真实存在的行数。

由于实现上的限制，BitmapAnd 和 BitmapOr 节点总是报告其实际行计数为零。

通常，EXPLAIN 会显示规划器生成的每一个计划节点。不过在某些情况下，执行器能够根据那些在规划阶段尚不可用的参数值判断某些节点根本不需要执行，因为它们不可能产生任何行。（目前，这种情况只会发生在扫描分区表的 Append 节点的子节点上。）当出现这种情况时，这些计划节点会从EXPLAIN 输出中省略，转而显示一条Subplans Removed: *N* 标注。

14.2. 规划器使用的统计信息

14.2.1. 单列统计信息

如上一节所见，查询规划器需要估计查询将检索多少行，才能对查询计划做出良好选择。本节简要介绍系统用于这些估计的统计信息。

统计信息的一部分是各个表和索引的条目总数，以及各个表和索引占用的磁盘块数。这些信息保存在表pg_class 的 reltuples 和 relpages 列中。可以使用类似下面的查询来查看：

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

这里可以看到，tenk1 包含 10000 行，它的索引也一样，但索引比表小得多（这并不意外）。

出于效率考虑，reltuples 和 relpages 不会实时更新，因此它们通常包含有些过时的值。它们会在VACUUM、ANALYZE 以及少数 DDL 命令（如 CREATE INDEX）执行时被更新。不扫描全表的 VACUUM 或 ANALYZE 操作（这很常见）会根据其实际扫描到的那一部分增量更新 reltuples 计数，因此得到的是近似值。无论如何，规划器都会将它在pg_class 中找到的值按当前物理表大小进行缩放，从而得到更接近实际情况的近似值。

大多数查询只会检索表中一部分行，因为它们通过WHERE 子句限制了需要检查的行。因此，规划器需要估算WHERE 子句的选择率，也就是满足WHERE 子句中各个条件的行所占比例。完成这项任务所需的信息存储在 pg_statistic 系统目录中。pg_statistic 中的条目由 ANALYZE 和 VACUUM ANALYZE 命令更新，而且即使刚更新完，也始终只是近似值。

手动检查统计信息时，与其直接查看pg_statistic，不如查看它的视图pg_stats。pg_stats 旨在让统计信息更易读。此外，pg_stats 所有用户都可以读取，而pg_statistic 只有超级用户可以读取。（这可以防止没有相应权限的用户通过统计信息获知其他用户表中的内容。pg_stats 视图被限制为只显示当前用户有权读取的表的相关行。）例如，可以执行：

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------+------------------------------------
 name    | f         |  -0.363388 | I- 580                          Ramp+
         |           |            | I- 880                          Ramp+
         |           |            | Sp Railroad                         +
         |           |            | I- 580                              +
         |           |            | I- 680                          Ramp
 name    | t         |  -0.284859 | I- 880                          Ramp+
         |           |            | I- 580                          Ramp+
         |           |            | I- 680                          Ramp+
         |           |            | I- 580                              +
         |           |            | State Hwy 13                    Ramp
(2 rows)
```

注意，同一列显示了两行，其中一行对应从road 表开始的整个继承层次结构（inherited =t），另一行只包含road 表本身（inherited=f）。

ANALYZE 在 pg_statistic 中存储的信息量，特别是每列most_common_vals 中的最大项数和 histogram_bounds 数组的大小，可以使用ALTER TABLE SET STATISTICS 命令按列设置，也可以通过设置配置变量default_statistics_target 进行全局设置。目前默认上限是 100 项。提高这一上限可能让规划器做出更准确的估计，尤其是对于数据分布不规则的列；代价则是 pg_statistic 占用更多空间，并且计算估计值所需时间也会略有增加。相反，对于数据分布较简单的列，较低的上限可能已经足够。

更多规划器对统计信息的使用可参阅第 71 章。

14.2.2. 扩展统计信息

由于查询子句中使用的多个列彼此相关，慢查询选择了糟糕执行计划的情况并不少见。规划器通常会假设多个条件彼此独立，而当列值存在相关性时，这一假设就不成立了。常规统计信息天然是按单列组织的，因此无法表达跨列相关性的信息。不过，PostgreSQL 能够计算多元统计信息，从而捕捉这类信息。

由于可能的列组合数量极其庞大，因此自动计算多元统计信息并不切实际。可以创建扩展统计信息对象，更常见的简称是统计信息对象，来指示服务器收集一组感兴趣列上的统计信息。

统计信息对象可使用 CREATE STATISTICS 命令创建。创建这样的对象，只是建立一个表示 “对此类统计信息感兴趣” 的目录项。实际的数据收集由ANALYZE 执行，无论是手工执行还是后台自动分析。收集到的值可以在pg_statistic_ext 目录中查看。

ANALYZE 使用与计算常规单列统计信息相同的表行样本来计算扩展统计信息。由于样本大小会随着表或列的统计信息目标值增大而增大，如前一节所述，较大的统计信息目标通常会带来更准确的扩展统计信息，同时也意味着需要花费更多时间来计算它们。

下面的小节介绍当前支持的扩展统计信息类型。

14.2.2.1. 函数依赖

最简单的一类扩展统计信息跟踪函数依赖，这是数据库范式定义中使用的概念。如果知道列a 的值就足以确定列 b 的值，也就是说，不存在两行具有相同的a 值却拥有不同的 b 值，那么就称列 b 函数依赖于列 a。在完全规范化的数据库中，函数依赖应当只存在于主键和超键上。不过在实践中，很多数据集出于各种原因并未完全规范化，常见情形之一就是为了性能而有意进行反规范化。即使在完全规范化的数据库中，某些列之间也可能存在部分相关性，这可以表现为部分函数依赖。

函数依赖的存在会直接影响某些查询中估计值的准确性。如果一个查询同时包含独立列和依赖列上的条件，那么依赖列上的条件不会进一步缩小结果集。但如果缺少函数依赖的信息，查询规划器就会假定这些条件彼此独立，从而低估结果集大小。

为了向规划器提供函数依赖信息，ANALYZE 可以收集跨列依赖程度的度量。若要评估所有列组之间的依赖程度，代价会高到难以承受，因此数据收集仅限于那些在同一个统计信息对象中共同出现的列组，这由dependencies 选项定义。建议只为强相关的列组创建 dependencies 统计信息，以避免在ANALYZE 和后续查询规划中产生不必要的开销。

下面是收集函数依赖统计信息的示例：

```
CREATE STATISTICS stts (dependencies) ON zip, city FROM zipcodes;

ANALYZE zipcodes;

SELECT stxname, stxkeys, stxdependencies
  FROM pg_statistic_ext
  WHERE stxname = 'stts';
 stxname | stxkeys |              stxdependencies
---------+---------+-------------------------------------------
 stts    | 1 5     | {"1 => 5": 1.000000, "5 => 1": 0.423130}
(1 row)
```

这里可以看到，第 1 列（邮政编码）完全决定第 5 列（城市），因此系数为 1.0；而城市仅在约 42% 的情况下能决定邮政编码，这意味着许多城市（58%）对应不止一个邮政编码。

在为涉及函数依赖列的查询计算选择率时，规划器会使用依赖系数来调整针对条件的选择率估计，这样就不会产生低估。

14.2.2.1.1. 函数依赖的限制

当前只有在考虑简单等值条件（将列与常量值比较）时，函数依赖才适用。不会使用它们来改进比较两个列或者比较列和表达式的等值条件的估计，也不会用它们来改进范围子句、LIKE 或者任何其他类型的条件。

在利用函数依赖做估计时，规划器假定涉及列上的条件彼此兼容，因此是冗余的。如果它们并不兼容，正确估计本应为零行，但这种可能性不会被考虑。例如，给定如下查询

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '94105';
```

规划器将会忽视 `city` 子句，因为它不改变选择率，这是正确的。不过，即便实际上没有任何行满足下面的查询，规划器也会做出同样的假设

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '90210';
```

然而，函数依赖统计信息提供的信息不足以据此断定结果为零行。

在很多实际场景中，这种假设通常是成立的。例如，应用程序中可能存在一个图形界面，只允许选择彼此兼容的城市和邮编值用于查询。但如果不是这样，函数依赖可能就不是一个可行的选项。

14.2.2.2. 多元非重复值计数

单列统计信息存储每一列中非重复值的数量。在组合多个列（例如 `GROUP BY a, b`）时，如果规划器只有单列统计数据，则对非重复值数量的估计常常会错误，导致选择不好的计划。

为了改进这种估计，`ANALYZE` 可以为列组收集非重复值统计信息。和以前一样，为每一种可能的列组合做这件事情是不切实际的，因此只会为一起出现在一个统计信息对象（用`ndistinct` 选项定义）中的列组收集数据。将会为所列出的列中两个或更多列的每一种可能组合收集数据。

继续前面的示例，邮政编码表中的非重复值计数可能如下所示：

```
CREATE STATISTICS stts2 (ndistinct) ON zip, state, city FROM zipcodes;

ANALYZE zipcodes;

SELECT stxkeys AS k, stxndistinct AS nd
  FROM pg_statistic_ext
  WHERE stxname = 'stts2';
-[ RECORD 1 ]-------------------------------------------------------
k  | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5": 33178}
(1 row)
```

这表示有三种列组合各有 33178 个非重复值：邮政编码与州；邮政编码与城市；以及邮政编码、城市与州（由于邮政编码本身在该表中唯一，三者相等是符合预期的）。另一方面，城市与州的组合只有 27435 个非重复值。

建议只对实际用于分组且其分组数错误估计导致了糟糕计划的列组合创建 `ndistinct` 统计信息对象。否则，`ANALYZE` 的处理时间只会被白白浪费。

14.3. 用显式 JOIN 子句控制规划器

可以在一定程度上用显式 `JOIN` 语法控制查询规划器。要理解这一点为何重要，先需要一些背景知识。

在一个简单的连接查询中，例如：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

规划器可以自由地按任意顺序连接这些表。例如，它可以生成一个查询计划，先利用WHERE 条件 a.id = b.id 将 A 连接到 B，然后再利用另一个WHERE 条件把 C 连接到这个结果上。也可以先连接 B 和 C，再把 A 连接到所得结果上。甚至还可以先连接 A 和 C，再与 B 连接，但这样效率会很低，因为必须先形成 A 和 C 的完整笛卡尔积，而WHERE 子句中并没有可用于优化这一连接的条件。（PostgreSQL 执行器中的所有连接都发生在两个输入表之间，因此结果必须以这些形式之一逐步构造出来。）关键在于，这些不同的连接可能性在语义上是等价的，但执行代价可能相差极大。因此，规划器会探索它们，力图找出最高效的查询计划。

当查询只涉及两个或三个表时，需要考虑的连接顺序并不多。但可能的连接顺序数量会随着表数增加而呈指数增长。输入表超过十个左右之后，对所有可能性做穷举搜索实际上就不再可行，甚至六七个表也可能让规划耗时长得令人厌烦。输入表过多时，PostgreSQL 规划器会从穷举搜索切换到一种遗传概率搜索，只考虑有限数量的可能性。（切换阈值由运行时参数geqo_threshold控制。）遗传搜索耗时更少，但并不一定能找到最优计划。

当查询涉及外连接时，规划器比处理普通（内）连接时拥有更小的自由度。例如，考虑：

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

尽管这个查询的约束表面上与前一个非常相似，但它们的语义不同，因为如果 A 中有某一行无法匹配 B 和 C 连接结果中的任何行，该行仍然必须被输出。因此这里规划器对连接顺序没有选择：它必须先连接 B 和 C，再把 A 连接到该结果上。相应地，这个查询比前一个查询需要更少的规划时间。在其他情况下，规划器可能会判断多种连接顺序都是安全的。例如：

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

将 A 首先连接到 B 或 C 都是有效的。当前，只有FULL JOIN 完全约束连接顺序。大多数涉及 LEFT JOIN 或 RIGHT JOIN 的实际情况都在某种程度上可以被重新排列。

显式内连接语法（INNER JOIN、CROSS JOIN 或无修饰的 JOIN）在语义上和 FROM 中列出输入关系是一样的，因此它不约束连接顺序。

即使大多数类型的 JOIN 并不会完全约束连接顺序，仍然可以指示PostgreSQL 查询规划器将所有 JOIN 子句都当作带有连接顺序约束来处理。例如，下面三个查询在逻辑上是等价的：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

但如果告诉规划器遵循 JOIN 顺序，那么第二个和第三个查询在规划上会比第一个花费更少时间。对于只有三个表的连接来说，这种效果微不足道；但当表很多时，它可能非常关键。

要强制规划器遵循显式 JOIN 给出的连接顺序，可以将运行时参数join_collapse_limit 设置为 1。（其他可能值见下文讨论。）

不必为了缩短搜索时间而完全约束连接顺序，因为可以在普通FROM 列表的某一项中使用 JOIN 操作符。例如：

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

如果设置 join_collapse_limit = 1，就会强制规划器先将 A 连接到 B，然后再与其他表连接，但不会进一步约束它的选择。在这个示例中，可能的连接顺序数量减少到原来的五分之一。

以这种方式约束规划器的搜索，是一种既可减少规划时间、又可引导规划器生成更好查询计划的实用技巧。如果规划器默认选择了糟糕的连接顺序，可以通过JOIN 语法强制它采用更好的顺序，前提当然是确实知道哪个顺序更好。建议进行实验。

与此密切相关、同样会影响规划时间的另一个问题，是将子查询折叠进父查询。例如：

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

这种情况可能出现在使用包含连接的视图时；该视图的 SELECT 规则会被插入到引用视图的位置，从而得到一个与上面非常相似的查询。通常，规划器会尝试把子查询折叠进父查询，得到：

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

这通常会生成比单独规划子查询更好的计划。（例如，外层WHERE 条件可能会先把 X 连接到 A，从而消除 A 中的大量行，也就避免了形成子查询完整逻辑输出的需要。）但与此同时，规划时间也增加了；这里把两个彼此独立的三路连接问题，替换成了一个五路连接问题。由于可能性数量呈指数增长，这种差别可能非常大。为了避免陷入巨大的连接搜索问题，如果折叠子查询会使父查询产生超过from_collapse_limit 个 FROM 项，规划器就会尝试通过停止提升子查询来规避这一点。可以通过调高或调低这个运行时参数，在规划时间和计划质量之间做权衡。

from_collapse_limit 和 join_collapse_limit 名称相似，因为它们做的几乎是同一件事：一个控制规划器何时将子查询“平面化”，另一个控制何时将显式连接平面化。通常，要么把 join_collapse_limit 设置为与 from_collapse_limit 相同，这样显式连接与子查询的行为类似；要么把join_collapse_limit 设置为 1，如果想使用显式连接控制连接顺序。不过，也可以把它们设置为不同的值，以更细致地调节规划时间与运行时间之间的平衡。

14.4. 填充一个数据库

初次填充数据库时，可能需要插入大量数据。本节给出一些使这一过程尽可能高效的建议。

14.4.1. 禁用自动提交

使用多个 INSERT 时，应关闭自动提交，只在最后提交一次。（在普通 SQL 中，这意味着开始时发出BEGIN，结束时发出COMMIT。某些客户端库可能会替调用方完成这件事，在这种情况下需要确认它们确实会在所需的时候这样做。）如果允许每次插入都单独提交，PostgreSQL 就必须为每一行的加入执行大量额外工作。把所有插入都放在一个事务中的另一个好处是：如果其中某一行插入失败，那么此前插入的所有行都会被回滚，这样就不会留下部分装载的数据。

14.4.2. 使用 COPY

使用 COPY 在一条命令中装载所有记录，而不是使用一系列INSERT 命令。COPY 命令针对装载大量行做了优化；它不如 INSERT 灵活，但在大规模数据装载时开销显著更小。由于COPY 是一条单独的命令，因此采用这种方法填充表时无须关闭自动提交。

如果不能使用 COPY，那么用PREPARE 创建一个预备 INSERT 语句，再按需多次执行EXECUTE 也会有所帮助。这样可以避免重复解析和规划 INSERT 的开销。不同接口以不同方式提供这一功能，可参阅接口文档中关于“预备语句”的说明。

请注意，在装载大量行时，使用COPY 几乎总是比使用 INSERT 更快，即使已经使用了 PREPARE，并把多次插入批量放入同一个事务中也是如此。

当 COPY 与更早的 CREATE TABLE 或 TRUNCATE 命令处于同一事务中时，速度最快。在这种情况下，不需要写 WAL，因为一旦出错，包含新装载数据的文件反正也会被移除。不过，这一点只适用于wal_level 为 minimal 的非分区表；否则所有命令都必须写 WAL。

14.4.3. 移除索引

如果正在装载一个新创建的表，最快的方法是先创建表，用COPY 批量装载数据，然后再创建该表所需的索引。在已有数据的表上创建索引，要比在每行装载时对索引做增量更新更快。

如果正在向现有表加入大量数据，那么删除索引、装载数据、再重建索引可能是更好的方案。当然，在索引缺失期间，其他数据库用户的性能可能会下降。删除唯一索引之前也必须慎重，因为唯一约束提供的错误检查会在索引缺失期间丧失。

14.4.4. 移除外键约束

与索引类似，“批量”检查外键约束比逐行检查更高效。因此，先删除外键约束、装载数据、再重建约束可能很有用。同样，这里也需要在装载速度与约束缺失期间失去错误检查之间做权衡。

更重要的是，当在已有外键约束的情况下向表中装载数据时，每一行新数据都需要在服务器待处理的触发器事件列表中占一个条目，因为外键约束检查是通过触发器触发完成的。装载数百万行可能导致触发器事件队列溢出可用内存，造成无法接受的交换，甚至让命令直接失败。因此在装载大量数据时，删除并重新应用外键可能是必须的，而不仅仅是期望如此。如果不能临时移除约束，唯一的替代办法可能就是把装载操作拆分成更小的事务。

14.4.5. 增加 maintenance_work_mem

在装载大量数据时，临时增大maintenance_work_mem 配置变量可以提升性能。这个参数也有助于加速 CREATE INDEX 和 ALTER TABLE ADD FOREIGN KEY 命令。它对 COPY 本身帮助不大，因此这个建议只有在采用前述一种或两种技巧时才有意义。

14.4.6. 增加 max_wal_size

临时增大 max_wal_size 配置变量，也可以让大规模数据装载更快。这是因为向PostgreSQL 中装载大量数据，会导致检查点比平常更频繁地发生，而正常频率由checkpoint_timeout 配置变量指定。每次发生检查点时，所有脏页都必须刷写到磁盘。通过在批量装载期间临时增大 max_wal_size，可以减少所需的检查点次数。

14.4.7. 禁用 WAL 归档和流复制

当在使用 WAL 归档或流复制的安装中载入大量数据时，完成装载后重新做一次基础备份，可能比处理大量增量 WAL 数据更快。为了避免在装载期间记录这些增量 WAL，可以通过将wal_level 设为 minimal、将archive_mode 设为 off，并将max_wal_senders 设为零，来禁用归档和流复制。但请注意，修改这些设置需要重启服务器。

这样做除了省去归档进程或 WAL 发送进程处理 WAL 数据的时间外，还能使某些命令执行得更快，因为这些命令被设计为在wal_level 为 minimal 时完全不写 WAL。（它们可以在结束时执行一次 fsync，以比写 WAL 更低的代价保证崩溃安全性。）这适用于以下命令：

- CREATE TABLE AS SELECT
- CREATE INDEX（以及 ALTER TABLE ADD PRIMARY KEY 等变体）
- ALTER TABLE SET TABLESPACE
- CLUSTER
- COPY FROM，前提是目标表在同一事务中已被创建或截断

14.4.8. 事后运行 ANALYZE

每当显著改变了表中数据的分布，都强烈建议运行 ANALYZE。这也包括向表中批量装载大量数据。运行 ANALYZE（或 VACUUM ANALYZE）可以确保规划器掌握该表的最新统计信息。如果没有统计信息，或者统计信息已经过时，规划器在生成查询计划时就可能做出糟糕决定，从而导致相关表性能不佳。注意，如果启用了自动清理守护进程，它可能会自动运行ANALYZE；详见第 24.1.3 节和第 24.1.6 节。

14.4.9. 关于 pg_dump 的一些注记

由 pg_dump 生成的转储脚本会自动应用上面若干条指导原则，但并非全部。若要尽可能快速地还原pg_dump 的转储，仍需手动做一些额外操作。（注意，这些要点适用于还原转储，而不是创建转储。无论是使用 psql 加载文本转储，还是使用pg_restore 从 pg_dump 归档文件加载，相关要点都是一样的。）

默认情况下，pg_dump 使用 COPY；而当它生成完整的模式加数据转储时，也会小心地先装载数据，再创建索引和外键。因此在这种情况下，上述若干指导原则已经被自动处理。剩下需要做的是：

- 为 maintenance_work_mem 和 max_wal_size 设置适当的（即比正常值大的）值。
- 如果使用 WAL 归档或流复制，可以考虑在恢复期间禁用它们。为此，请在载入转储之前将 archive_mode 设为 off、将wal_level 设为 minimal，并将max_wal_senders 设为零。恢复完成后，再把这些设置改回正确的值，并重新做一次基础备份。
- 试验 pg_dump 和 pg_restore 的并行转储与恢复模式，找出最优的并发任务数量。通过-j 选项进行并行转储和恢复，通常会比串行模式获得高得多的性能。
- 考虑是否应当把整个转储作为单个事务来恢复。要这样做，请把-1 或 --single-transaction 命令行选项传给 psql 或 pg_restore。使用这种模式时，即使是很小的错误也会回滚整个恢复过程，可能丢掉数小时的处理成果。视数据之间的关联程度而定，这种做法未必一定比手工清理更可取。如果使用单个事务并关闭 WAL 归档，COPY 命令会运行得最快。
- 如果在数据库服务器上有多个 CPU 可用，可以考虑使用pg_restore 的 --jobs 选项。这允许并行数据载入和索引创建。
- 之后运行 ANALYZE。

仅包含数据的转储仍然会使用 COPY，但它不会删除或重建索引，通常也不会处理外键。[1]因此，在装载纯数据转储时，如果想采用这些技术，就需要自行负责删除并重建索引与外键。装载数据期间增大max_wal_size 仍然有益，但没有必要同时增大maintenance_work_mem；后者更适合留到之后手工重建索引和外键时再调大。完成后也别忘了执行 ANALYZE；详见第 24.1.3 节和第 24.1.6 节。

14.5. 非持久设置

持久性是一种数据库特性，保证即使服务器崩溃或断电，已提交的事务也会被记录下来。但持久性会带来显著的数据库开销，因此如果你的应用场景不需要这种保证，可以将PostgreSQL 配置为以快得多的速度运行。以下配置更改可以在这些情况下提高性能。除下文特别注明的情况外，数据库软件崩溃时仍然保证持久性；使用这些设置时，只有操作系统突然停止运行才会造成数据丢失或损坏的风险。

- 将数据库集簇的数据目录放在一个内存支持的文件系统上（即 RAM 磁盘）。这消除了所有的数据库磁盘 I/O，但将数据存储限制到可用的内存量（可能有交换区）。
- 关闭 fsync；不需要将数据刷入磁盘。
- 关闭 synchronous_commit；这样可能无需在每次提交时强制将 WAL 写入磁盘。这种设置会在数据库崩溃时带来事务丢失的风险（但不会造成数据损坏）。
- 关闭 full_page_writes；不需要警惕部分页面写入。
- 增加 max_wal_size 和 checkpoint_timeout；这会降低检查点频率，但会增加/pg_wal 的存储需求。
- 创建不记录 WAL 的表来避免 WAL 写入，不过这样会使表在崩溃时不再安全。

[1] 可以通过使用 --disable-triggers 选项达到禁用外键的效果 – 但要注意，这样做是取消外键验证，而不仅仅是推迟它。因此如果使用该选项，就有可能插入坏数据。

第 15 章 并行查询

PostgreSQL 可以制定能够利用多个 CPU、从而更快回答查询的查询计划。这一特性称为并行查询。许多查询无法从并行查询中获益，要么是因为当前实现存在限制，要么是因为根本不存在比串行查询计划更快的可行计划。不过，对于能够从中受益的查询，并行查询带来的加速通常非常明显。许多查询在使用并行查询时，运行速度可以达到原来的两倍以上，有些甚至可以达到四倍或更高。那些访问大量数据但只向用户返回少量行的查询，通常最能从并行查询中获益。本章将解释并行查询的工作方式以及可以在哪些情况下使用它，从而帮助希望利用这一特性的用户了解可以期待的效果。

15.1. 并行查询如何工作

当优化器确定并行查询是某个特定查询最快的执行策略时，它会创建一个包含 Gather 或 Gather Merge 节点的查询计划。下面是一个简单示例：

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
                                     QUERY PLAN
------------------------------------------------------------------------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33 rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

无论在哪种情况下，`Gather` 或 `Gather Merge` 节点都恰好有一个子计划，也就是将以并行方式执行的那部分计划。如果 `Gather` 或 `Gather Merge` 节点位于计划树的最顶端，那么整个查询都会并行执行；如果它位于计划树中的其他位置，那么只有它下方的那部分计划会并行运行。在上面的示例中，查询只访问一个表，因此除了 `Gather` 节点本身之外只有一个计划节点；由于该计划节点是 `Gather` 节点的子节点，所以它会并行运行。

通过使用 EXPLAIN，你可以看到规划器选择的工作进程数量。当查询执行到 `Gather` 节点时，实现用户会话的进程会请求与规划器所选数量相同的后台工作进程。规划器会考虑使用的后台工作进程数量，最多不超过 max_parallel_workers_per_gather。任意时刻可存在的后台工作进程总数同时受到 max_worker_processes 和 max_parallel_workers 的限制。因此，并行查询实际运行时可能使用比计划更少的工作进程，甚至完全没有工作进程。最优计划可能依赖于可用工作进程的数量，因此这会导致查询性能不佳。如果这种情况经常发生，可以考虑增大 `max_worker_processes` 和 `max_parallel_workers`，让更多工作进程能够同时运行；或者降低 `max_parallel_workers_per_gather`，使规划器请求更少的工作进程。

为某个并行查询成功启动的每个后台工作进程都会执行计划的并行部分。领导者也会执行这部分计划，但它还承担着额外职责：必须读取所有由工作进程生成的元组。当计划的并行部分只生成少量元组时，领导者通常会表现得很像一个额外的工作进程，从而加快查询执行。反过来，当计划的并行部分生成大量元组时，领导者可能几乎完全忙于读取工作进程生成的元组，并执行位于 `Gather` 或 `Gather Merge` 节点之上的计划节点所要求的后续处理。在这种情况下，领导者实际参与执行计划并行部分的工作就会很少。

当计划并行部分的顶层节点是 `Gather Merge` 而不是 `Gather` 时，这表示每个执行计划并行部分的进程都会按排序顺序生成元组，而领导者会执行保持顺序的归并。相比之下，`Gather` 会以任意方便的顺序从工作进程读取元组，从而破坏原本可能存在的排序顺序。

15.2. 何时可以使用并行查询？

有几种设置会导致查询规划器在任何情况下都不生成并行查询计划。要让系统能够生成任何并行查询计划，下列设置必须按要求配置。

- max_parallel_workers_per_gather 必须设置为大于零的值。这只是更一般原则的一个特例，即所使用的工作进程数量不应超过通过 `max_parallel_workers_per_gather` 配置的数量。
- dynamic_shared_memory_type 必须设为 `none` 以外的值。并行查询需要动态共享内存，以便在协作进程之间传递数据。

此外，系统不能运行在单用户模式下。因为在这种情况下整个数据库系统作为单个进程运行，所以没有后台工作进程可用。

即使系统通常可以为某个给定查询生成并行查询计划，只要下面任一条件成立，规划器都不会为该查询生成并行查询计划：

- 查询会写入任何数据或者锁定任何数据库行。如果一个查询在顶层或 CTE 中包含数据修改操作，那么不会为该查询生成并行计划。作为例外，创建新表并向其中填充数据的命令 `CREATE TABLE ... AS`、`SELECT INTO` 和 `CREATE MATERIALIZED VIEW` 可以使用并行计划。
- 查询在执行过程中可能会被挂起。只要系统认为可能发生部分执行或增量执行，就不会生成并行计划。例如，使用 DECLARE CURSOR 创建的游标绝不会使用并行计划。类似地，形如 `FOR x IN query LOOP .. END LOOP` 的 PL/pgSQL 循环也绝不会使用并行计划，因为并行查询系统无法验证在并行查询处于活动状态时执行循环中的代码是否安全。
- 查询使用了任何被标记为 `PARALLEL UNSAFE` 的函数。大多数系统定义的函数都被标记为 `PARALLEL SAFE`，但用户定义的函数默认被标记为 `PARALLEL UNSAFE`。参见第 15.4 节中的讨论。
- 该查询运行在另一个已经并行执行的查询内部。例如，如果一个由并行查询调用的函数自己又发出一个 SQL 查询，那么该查询将绝不会使用并行计划。这是当前实现的一个限制，但未必值得移除，因为那样可能导致单个查询使用大量进程。
- 事务隔离级别为可串行化。这是当前实现的限制。

即使某个特定查询已经生成了并行查询计划，在执行时仍然有若干情况会导致该计划无法并行执行。如果发生这种情况，领导者将完全独自执行 `Gather` 节点以下的那部分计划，几乎就像 `Gather` 节点根本不存在一样。满足下列任一条件时，就会出现这种情况：

- 由于后台工作进程总数不能超过 max_worker_processes，因此无法获得后台工作进程。
- 由于为并行查询启动的后台工作进程总数不能超过 max_parallel_workers，因此无法获得后台工作进程。
- 客户端发送了带有非零提取计数的 Execute 消息。请参见扩展查询协议中的讨论。由于 libpq 目前没有提供发送此类消息的方法，因此这种情况只会出现在不依赖 libpq 的客户端中。如果这种情况经常发生，那么在可能发生这种情况的会话中将max_parallel_workers_per_gather 设置为零可能是个好主意，这样可以避免生成那些在串行运行时可能次优的查询计划。
- 事务隔离级别为可串行化。通常不会出现这种情况，因为在事务隔离级别为可串行化时，不会生成并行查询计划。但是，如果在计划生成之后、执行之前将事务隔离级别改为可串行化，就可能出现这种情况。

15.3. 并行计划

由于每个工作进程都会将计划的并行部分执行到底，因此不能简单地拿一个普通查询计划并让多个工作进程同时运行。那样每个工作进程都会生成完整输出结果集的一份副本，所以查询不仅不

会比平常更快，反而会产生错误结果。相反，计划的并行部分必须是查询优化器内部所说的部分计划；也就是说，它必须被构造为使执行该计划的每个进程只生成输出行的一个子集，并且保证每一条所需输出行都恰好由某个协作进程生成一次。一般来说，这意味着查询驱动表上的扫描必须是并行感知扫描。

15.3.1. 并行扫描

目前支持下列类型的并行感知表扫描。

- 在并行顺序扫描中，表的块会被分配给各个协作进程。每次只分配一个块，因此对表的访问仍然是顺序的。
- 在并行位图堆扫描中，会选择一个进程作为领导者。该进程执行一个或多个索引扫描，并构建一个位图，以指示需要访问哪些表块。然后这些块会像并行顺序扫描那样分配给协作进程。换句话说，堆扫描是并行执行的，但底层索引扫描不是。
- 在并行索引扫描或并行仅索引扫描中，协作进程轮流从索引中读取数据。目前，并行索引扫描仅支持 B-树索引。每个进程都会认领一个索引块，并扫描并返回该块所引用的全部元组；与此同时，其他进程可以从不同的索引块返回元组。并行 B-树扫描的结果在每个工作进程内部都会按排序顺序返回。

其他扫描类型，例如非 B-树索引扫描，将来也可能支持并行扫描。

15.3.2. 并行连接

与非并行计划一样，驱动表可以通过嵌套循环、哈希连接或归并连接与一个或多个其他表连接。连接的内侧可以是规划器支持的任何类型的非并行计划，只要它能够安全地在并行工作进程中运行。根据连接类型，内侧也可以是并行计划。

- 在嵌套循环连接中，内侧始终是非并行的。尽管它会被完整执行，但如果内侧是索引扫描，这种方式仍然是高效的，因为外侧元组以及在索引中查找值的循环会分摊到多个协作进程上。
- 在归并连接中，内侧始终是非并行计划，因此会被完整执行。这可能效率不高，尤其是在必须执行排序时，因为工作量和结果数据会在每个协作进程中重复。
- 在哈希连接（没有“并行”前缀）中，每个协作进程都会完整执行内侧，以构建同样的 hash 表副本。如果 hash 表很大，或者该计划代价很高，这种方式可能效率不高。在并行哈希连接中，内侧是一个并行 hash，它会将构建共享 hash 表的工作分摊到多个协作进程上。

15.3.3. 并行聚合

PostgreSQL 通过分两个阶段进行聚合来支持并行聚合。首先，每个参与查询并行部分的进程执行一个聚合步骤，为该进程所见到的每个分组产生一个部分结果。这在计划中体现为一个 `Partial Aggregate` 节点。然后，部分结果通过 `Gather` 或 `Gather Merge` 传送给领导者。最后，领导者会把来自所有工作进程的结果再次聚合，以产生最终结果。这在计划中体现为一个 `Finalize Aggregate` 节点。

由于 `Finalize Aggregate` 节点运行在领导者进程上，因此对于那些相对于输入行数会产生较多分组的查询，查询规划器会认为它不太有利。例如，在最坏情况下，`Finalize Aggregate` 节点看到的分组数可能与所有工作进程在 `Partial Aggregate` 阶段看到的输入行数一样多。对于这种情况，使用并行聚合显然不会带来性能收益。查询规划器会在规划过程中考虑这一点，因此在这种场景下不太可能选择并行聚合。

并行聚合并非在所有情况下都受支持。每个聚合都必须是并行安全的，并且必须具有合并函数。如果该聚合具有类型为 `internal` 的转移状态，那么它还必须具有序列化和反序列化函数。更多细节请参见 CREATE AGGREGATE。如果任何聚合函数调用包含 `DISTINCT` 或 `ORDER BY` 子句，

则不支持并行聚合。对于有序集聚合，或者当查询涉及 GROUPING SETS 时，也不支持并行聚合。只有当查询涉及的所有连接也都属于计划并行部分时，才能使用并行聚合。

15.3.4. 并行追加

每当 PostgreSQL 需要将来自多个源的行合并成一个结果集时，它就会使用 Append 或 MergeAppend 计划节点。这种情况常见于实现 UNION ALL 或扫描分区表时。这样的节点和其他任何计划中的情形一样，也可以用于并行计划。不过，在并行计划中，规划器也可能改用 Parallel Append 节点。

当 Append 节点用于并行计划时，每个进程都会按照子计划出现的顺序执行它们，因此所有参与进程会协作执行第一个子计划直到其完成，然后再大致同时转向第二个子计划。而当使用 Parallel Append 时，执行器会尽可能均匀地将参与进程分散到各个子计划上，从而使多个子计划能够同时执行。这样既避免了争用，也避免让那些从未执行某个子计划的进程承担其启动代价。

此外，普通的 Append 节点在并行计划中只能拥有部分子计划，而 Parallel Append 节点则既可以拥有部分子计划，也可以拥有非部分子计划。非部分子计划只能由单个进程扫描，因为重复扫描会产生重复结果。因此，即使没有高效的部分计划可用，涉及追加多个结果集的计划仍然可以实现粗粒度并行。比如，考虑一个针对分区表的查询，而该查询只有通过某个不支持并行扫描的索引才能高效实现。规划器可能会选择由普通 Index Scan 计划组成的 Parallel Append；每个单独的索引扫描仍然必须由单个进程执行到底，但不同的扫描可以由不同进程同时执行。

enable_parallel_append 可用于禁用这一特性。

15.3.5. 并行计划提示

如果一个本来预期会生成并行计划的查询却没有生成并行计划，可以尝试降低 parallel_setup_cost 或 parallel_tuple_cost。当然，这样得到的计划可能会比规划器原本偏好的串行计划更慢，但并不总是如此。如果即使把这些设置调得很低（例如都设为零）之后仍然得不到并行计划，那么可能存在某些原因使查询规划器无法为该查询生成并行计划。关于可能的原因，请参见第 15.2 节和第 15.4 节。

在执行并行计划时，可以使用 EXPLAIN (ANALYZE, VERBOSE) 显示每个计划节点的逐工作进程统计信息。这有助于判断工作是否在各个计划节点之间均匀分布，以及更全面地理解该计划的性能特征。

15.4. 并行安全性

规划器会将查询中涉及的操作分类为并行安全、并行受限或并行不安全。并行安全的操作不会与并行查询的使用产生冲突。并行受限的操作不能在并行工作进程中执行，但可以在启用并行查询时由领导者执行。因此，并行受限的操作绝不能出现在 Gather 或 Gather Merge 节点之下，但可以出现在包含这类节点的计划的其他位置。并行不安全的操作在并行查询启用时完全不能执行，连领导者中也不行。当一个查询包含任何并行不安全的内容时，并行查询对该查询会被完全禁用。

下列操作总是并行受限的：

- 公共表表达式（CTE）的扫描。
- 临时表的扫描。
- 外部表的扫描，除非外部数据包装器提供了 IsForeignScanParallelSafe API，并明确指出并行执行是安全的。

- 附带 InitPlan 的计划节点。
- 引用相关 SubPlan 的计划节点。

15.4.1. 为函数和聚合指定并行标签

规划器无法自动判断一个用户定义的函数或聚合究竟是并行安全、并行受限还是并行不安全，因为这需要预测该函数可能执行的每一种操作。一般来说，这等价于停机问题，因此是不可能做到的。即使对于那些理论上可能判定出来的简单函数，我们也不会尝试，因为那样做代价高昂而且容易出错。相反，所有用户定义的函数都会被假定为并行不安全，除非另有标记。在使用CREATE FUNCTION 或 ALTER FUNCTION 时，可以通过指定 PARALLEL SAFE、PARALLEL RESTRICTED 或 PARALLEL UNSAFE 来设置标记。在使用 CREATE AGGREGATE 时，可以将 PARALLEL 选项的值指定为 SAFE、RESTRICTED 或 UNSAFE。

如果函数或聚合会写入数据库、访问序列、修改事务状态（即使只是临时修改，例如 PL/pgSQL 函数建立 EXCEPTION 块来捕获错误），或者对设置作出持久更改，那么它们必须标记为 PARALLEL UNSAFE。类似地，如果函数访问临时表、客户端连接状态、游标、预备语句，或者系统无法在工作进程之间同步的各种后端本地状态，那么它必须标记为 PARALLEL RESTRICTED。例如，setseed 和 random 就因为最后一个原因而属于并行受限。

一般来说，如果某个函数实际上是受限或不安全的，却被标记为安全，或者实际上是不安全的，却被标记为受限，那么在并行查询中使用它时可能会抛出错误，或者产生错误结果。如果 C 语言函数被错误标记，理论上它甚至可能表现出完全未定义的行为，因为系统无法保护自己免受任意 C 代码的影响。不过，在最可能发生的情况下，结果通常也不会比其他任何函数更糟。如果有疑虑，最好还是将函数标记为 UNSAFE。

如果在并行工作进程中执行的函数获取了领导者并未持有的锁，例如通过查询该查询中未引用的表，那么这些锁会在工作进程退出时释放，而不是在事务结束时释放。如果你编写了这样一个函数，并且这种行为差异对你很重要，那么应将此类函数标记为 PARALLEL RESTRICTED，以确保它们只在领导者中执行。

请注意，查询规划器不会为了得到更优的计划，而考虑推迟计算查询中涉及的并行受限函数或聚合。因此，如果某个应用于特定表的 WHERE 子句是并行受限的，查询规划器就不会考虑在计划的并行部分中对该表执行扫描。在某些情况下，把该表扫描纳入查询的并行部分，并将 WHERE 子句的计算推迟到 Gather 节点之上，可能是可行的，甚至更高效。然而，规划器不会这样做。

部分 III. 服务器管理

本部分涵盖 PostgreSQL 管理员关心的主题，包括软件安装、服务器配置、用户和数据库管理以及维护任务。任何运行PostgreSQL 服务器的人，即使仅供个人使用，尤其是在生产环境中，都应熟悉这些主题。

本部分的内容大体按照新用户应当阅读的顺序来安排。各章自成体系，也可按需单独阅读。相关内容按主题单元以叙述方式展开。希望获得某个命令完整描述的读者，建议查阅第 VI 部分。

开头几章写得无需预备知识即可理解，因此需要自行搭建服务器的新用户可以从这里开始探索。本部分其余内容涉及调优和管理；这些材料假定读者已经熟悉PostgreSQL 数据库系统的一般用法。建议读者再参阅第 I 部分和第 II 部分部分，以获取更多信息。

目录

16. 从源代码安装 498
16.1. 简要说明 498
16.2. 需求 498
16.3. 获取源代码 500
16.4. 安装过程 500
16.5. 安装后设置 512
16.5.1. 共享库 512
16.5.2. 环境变量 513
16.6. 受支持的平台 513
16.7. 平台相关说明 514
16.7.1. AIX 514
16.7.2. Cygwin 517
16.7.3. HP-UX 518
16.7.4. macOS 518
16.7.5. MinGW/原生 Windows 519
16.7.6. Solaris 520
17. 在 Windows 上从源代码安装 522
17.1. 使用 Visual C++ 或 Microsoft Windows SDK 构建 522
17.1.1. 需求 523
17.1.2. 64 位 Windows 的特殊注意事项 525
17.1.3. 构建 525
17.1.4. 清理和安装 525
17.1.5. 运行回归测试 526
17.1.6. 构建文档 527
18. 服务器设置和操作 528
18.1. PostgreSQL 用户账户 528
18.2. 创建一个数据库集簇 528
18.2.1. 二级文件系统的使用 529
18.2.2. 使用网络文件系统 529
18.3. 启动数据库服务器 530
18.3.1. 服务器启动失败 532
18.3.2. 客户端连接问题 532
18.4. 管理内核资源 533
18.4.1. 共享内存和信号量 533
18.4.2. systemd RemoveIPC 538
18.4.3. 资源限制 539
18.4.4. Linux 内存过量分配 540
18.4.5. Linux 大页 542
18.5. 关闭服务器 542
18.6. 升级 PostgreSQL 集簇 543
18.6.1. 通过 pg_dumpall 升级数据 544
18.6.2. 通过 pg_upgrade 升级数据 545
18.6.3. 通过复制升级数据 546
18.7. 防止服务器欺骗 546
18.8. 加密选项 546
18.9. 使用 SSL 的安全 TCP/IP 连接 547
18.9.1. 基本设置 547
18.9.2. OpenSSL 配置 548
18.9.3. 使用客户端证书 548
18.9.4. SSL 服务器文件用法 549
18.9.5. 创建证书 549

18.10. 通过 SSH 隧道建立安全 TCP/IP 连接 551
18.11. 在 Windows 上注册事件日志 552
19. 服务器配置 554
19.1. 设置参数 554
19.1.1. 参数名称和值 554
19.1.2. 通过配置文件影响参数 554
19.1.3. 通过 SQL 影响参数 555
19.1.4. 通过 Shell 影响参数 556
19.1.5. 管理配置文件内容 556
19.2. 文件位置 558
19.3. 连接和认证 558
19.3.1. 连接设置 558
19.3.2. 认证 561
19.3.3. SSL 562
19.4. 资源消耗 564
19.4.1. 内存 564
19.4.2. 磁盘 566
19.4.3. 内核资源使用 566
19.4.4. 基于代价的清理延迟 566
19.4.5. 后台写入器 567
19.4.6. 异步行为 568
19.5. 预写式日志 570
19.5.1. 设置 570
19.5.2. 检查点 574
19.5.3. 归档 575
19.6. 复制 575
19.6.1. 发送服务器 575
19.6.2. 主库 576
19.6.3. 备库 578
19.6.4. 订阅者 579
19.7. 查询规划 580
19.7.1. 规划器方法配置 580
19.7.2. 规划器代价常量 581
19.7.3. 遗传查询优化器 583
19.7.4. 其他规划器选项 584
19.8. 错误报告和日志 586
19.8.1. 日志记录到哪里 586
19.8.2. 什么时候记录日志 589
19.8.3. 记录哪些内容 591
19.8.4. 使用 CSV 格式的日志输出 596
19.8.5. 进程标题 597
19.9. 运行时统计数据 597
19.9.1. 查询和索引统计信息收集器 597
19.9.2. 统计监控 598
19.10. 自动清理 598
19.11. 客户端连接默认值 600
19.11.1. 语句行为 600
19.11.2. 区域设置和格式化 605
19.11.3. 共享库预载入 606
19.11.4. 其他默认值 608
19.12. 锁管理 609
19.13. 版本和平台兼容性 610
19.13.1. 以前的 PostgreSQL 版本 610
19.13.2. 平台和客户端兼容性 611

19.14. 错误处理 612
19.15. 预置选项 612
19.16. 自定义选项 614
19.17. 开发人员选项 614
19.18. 短选项 617
20. 客户端认证 619
20.1. pg_hba.conf 文件 619
20.2. 用户名映射 625
20.3. 认证方法 626
20.4. 信任认证 627
20.5. 密码认证 627
20.6. GSSAPI 认证 628
20.7. SSPI 认证 629
20.8. Ident 认证 630
20.9. Peer 认证 631
20.10. LDAP 认证 631
20.11. RADIUS 认证 634
20.12. 证书认证 635
20.13. PAM 认证 635
20.14. BSD 认证 636
20.15. 认证问题 636
21. 数据库角色 638
21.1. 数据库角色 638
21.2. 角色属性 639
21.3. 角色成员资格 640
21.4. 删除角色 642
21.5. 默认角色 643
21.6. 函数安全性 644
22. 管理数据库 645
22.1. 概述 645
22.2. 创建数据库 645
22.3. 模板数据库 646
22.4. 数据库配置 648
22.5. 删除数据库 648
22.6. 表空间 648
23. 本地化 651
23.1. 区域设置支持 651
23.1.1. 概述 651
23.1.2. 行为 652
23.1.3. 问题 653
23.2. 排序规则支持 653
23.2.1. 概念 653
23.2.2. 管理排序规则 655
23.3. 字符集支持 659
23.3.1. 支持的字符集 659
23.3.2. 设置字符集 661
23.3.3. 服务器和客户端之间的自动字符集转换 662
23.3.4. 进一步阅读 664
24. 日常数据库维护任务 665
24.1. 日常清理 665
24.1.1. 清理基础 665
24.1.2. 回收磁盘空间 666
24.1.3. 更新规划器统计信息 667

24.1.4. 更新可见性映射 668
24.1.5. 防止事务 ID 回卷失败 668
24.1.6. 自动清理守护进程 671
24.2. 日常重建索引 672
24.3. 日志文件维护 673
25. 备份和恢复 675
25.1. SQL 转储 675
25.1.1. 恢复转储 676
25.1.2. 使用 pg_dumpall 677
25.1.3. 处理大型数据库 677
25.2. 文件系统级备份 678
25.3. 持续归档和时间点恢复（PITR） 679
25.3.1. 设置 WAL 归档 680
25.3.2. 进行基础备份 682
25.3.3. 使用低级 API 进行基础备份 682
25.3.4. 使用持续归档备份进行恢复 686
25.3.5. 时间线 687
25.3.6. 建议和示例 688
25.3.7. 注意事项 690
26. 高可用、负载均衡和复制 691
26.1. 不同方案的比较 691
26.2. 日志传送备库 694
26.2.1. 规划 694
26.2.2. 备库操作 694
26.2.3. 为备库准备主库 695
26.2.4. 设置备库 695
26.2.5. 流复制 696
26.2.6. 复制槽 697
26.2.7. 级联复制 698
26.2.8. 同步复制 698
26.2.9. 在备库中持续归档 701
26.3. 故障切换 701
26.4. 日志传送的另一种方法 702
26.4.1. 实现 703
26.4.2. 基于记录的日志传送 703
26.5. 热备 703
26.5.1. 用户概览 704
26.5.2. 处理查询冲突 705
26.5.3. 管理员概览 707
26.5.4. 热备参数参考 709
26.5.5. 注意事项 709
27. 恢复配置 710
27.1. 归档恢复设置 710
27.2. 恢复目标设置 711
27.3. 备库设置 712
28. 监控数据库活动 714
28.1. 标准 Unix 工具 714
28.2. 统计收集器 715
28.2.1. 统计信息收集配置 715
28.2.2. 查看统计信息 715
28.2.3. 统计函数 738
28.3. 查看锁 740
28.4. 进度报告 740
28.4.1. VACUUM 进度报告 740

28.5. 动态追踪 742
28.5.1. 为动态追踪编译 742
28.5.2. 内置探针 742
28.5.3. 使用探针 748
28.5.4. 定义新探针 749
29. 监控磁盘使用情况 751
29.1. 确定磁盘使用情况 751
29.2. 磁盘已满故障 752
30. 可靠性与预写式日志 753
30.1. 可靠性 753
30.2. 预写式日志（WAL） 754
30.3. 异步提交 755
30.4. WAL 配置 756
30.5. WAL 内部机制 758
31. 逻辑复制 760
31.1. 发布 760
31.2. 订阅 761
31.2.1. 复制槽管理 761
31.3. 冲突 762
31.4. 限制 762
31.5. 架构 762
31.5.1. 初始快照 763
31.6. 监控 763
31.7. 安全性 763
31.8. 配置设置 764
31.9. 快速设置 764
32. 即时编译（JIT） 765
32.1. 什么是 JIT 编译？ 765
32.1.1. JIT 加速的操作 765
32.1.2. 内联 765
32.1.3. 优化 765
32.2. 何时使用 JIT？ 765
32.3. 配置 766
32.4. 可扩展性 767
32.4.1. 对扩展的内联支持 767
32.4.2. 可插拔的 JIT 提供者 767
33. 回归测试 768
33.1. 运行测试 768
33.1.1. 针对临时安装运行测试 768
33.1.2. 针对现有安装运行测试 769
33.1.3. 附加测试套件 769
33.1.4. 区域设置和编码 770
33.1.5. 额外测试 771
33.1.6. 测试热备 771
33.2. 测试结果评估 771
33.2.1. 错误消息差异 772
33.2.2. 区域设置差异 772
33.2.3. 日期和时间差异 772
33.2.4. 浮点差异 772
33.2.5. 行顺序差异 773
33.2.6. 栈深度不足 773
33.2.7. “random” 测试 773
33.2.8. 配置参数 773
33.3. 变体比较文件 773

33.4. TAP 测试 774
33.5. 测试覆盖率检查 775

第 16 章 从源代码安装

本章介绍如何使用源代码发布包安装 PostgreSQL。（如果你安装的是预打包的发行版，例如 RPM 或 Debian 软件包，请忽略本章，改为阅读打包者提供的说明。）

16.1. 简要说明

```
./configure
make
su
make install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

详细说明见本章的其余部分。

16.2. 需求

一般来说，现代的 Unix 兼容平台应该能够运行 PostgreSQL。发布时经过具体测试的平台列于下方第 16.6 节。发行包的 `doc` 子目录中有若干平台特定的 FAQ 文档，遇到问题时可以查阅。

构建 PostgreSQL 需要下列软件包：

- 需要 GNU make 3.80 或更高版本；其他 make 程序或较旧版本的 GNU make 都无法工作。（GNU make 有时会以 `gmake` 这个名字安装。）要检查是否为 GNU make，请输入：

  ```
  make --version
  ```

- 你需要一个 ISO/ANSI C 编译器（至少符合 C89）。推荐使用较新的 GCC 版本，不过已知 PostgreSQL 也可以使用来自不同厂商的多种编译器构建。

- 需要 tar 来解开源代码发布包，此外还需要 gzip 或 bzip2 之一。

- 默认使用 GNU Readline 库。它可以让 psql（PostgreSQL 命令行 SQL 解释器）记住你输入的每一条命令，并允许你使用方向键回忆和编辑此前的命令。这非常有用，强烈推荐。如果你不想使用它，则必须给 `configure` 指定 `--without-readline` 选项。作为替代方案，你通常也可以使用 BSD 许可的 `libedit` 库，它最初是在 NetBSD 上开发的。`libedit` 库与 GNU Readline 兼容；如果找不到 `libreadline`，或者在 `configure` 中使用了 `--with-libedit-preferred` 选项，就会使用它。如果你使用的是基于软件包的 Linux 发行版，请注意，如果你的发行版把它们拆成单独的软件包，那么需要同时安装 `readline` 和 `readline-devel`。

- 默认使用 zlib 压缩库。如果你不想使用它，则必须给 `configure` 指定 `--without-zlib` 选项。使用这个选项将禁用 pg_dump 和 pg_restore 对压缩归档的支持。

下列软件包是可选的。默认配置不需要它们，但在启用某些构建选项时会用到，如下所述：

- 要构建服务器端编程语言 PL/Perl，你需要一个完整的 Perl 安装，包括 `libperl` 库和头文件。最低要求版本是 Perl 5.8.3。由于 PL/Perl 是一个共享库，在大多数平台上，`libperl` 也必须是共享库。这在较新的 Perl 版本中似乎是默认行为，但在更早版本中并非如此；无论如何，这取决于你所在环境中安装 Perl 的人是如何选择的。如果选择构建 PL/Perl，但 `configure` 找不到共享的 `libperl`，则配置将失败。在这种情况下，你必须手动重新构建并安装 Perl，才能构建 PL/Perl。在配置 Perl 时，请要求生成共享库。

  如果你打算不仅仅是偶尔使用 PL/Perl，应确保 Perl 安装是在启用了 `usemultiplicity` 选项的情况下构建的（`perl -V` 会显示这一点）。

- 要构建 PL/Python 服务器端编程语言，你需要带有头文件以及 sysconfig 模块的 Python 安装。最低要求版本是 Python 2.7。支持 3.2 及更高版本的 Python 3；但使用 Python 3 时请参见第 46.1 节。

  由于 PL/Python 是一个共享库，在大多数平台上，`libpython` 也必须是共享库。默认情况下，从源代码构建的 Python 安装并非如此，但很多操作系统发行版提供了共享库。如果选择构建 PL/Python，但 `configure` 找不到共享的 `libpython`，则配置将失败。这可能意味着你要么需要安装额外的软件包，要么需要重新构建（部分）Python 安装，以提供这个共享库。从源代码构建时，请在运行 Python 的 configure 时加上 `--enable-shared` 标志。

- 要构建 PL/Tcl 过程语言，你当然需要安装 Tcl。最低要求版本是 Tcl 8.4。

- 要启用本地语言支持（NLS），也就是以非英语语言显示程序消息的能力，你需要一个 Gettext API 的实现。某些操作系统已内置此功能（例如 Linux、NetBSD、Solaris）；对于其他系统，你可以从http://www.gnu.org/software/gettext/下载附加软件包。如果你使用的是 GNU C 库中的 Gettext 实现，那么还需要 GNU Gettext 软件包来提供某些实用程序。如果使用其他实现，则不需要它。

- 如果你希望支持加密的客户端连接，就需要 OpenSSL。最低要求版本是 0.9.8。

- 如果你希望支持使用相应服务进行认证，就需要 Kerberos、OpenLDAP 和/或 PAM。

- 要构建 PostgreSQL 文档，还需要满足另一组要求；见第 J.2 节。

如果你从 Git 树构建，而不是使用已发布的源码包，或者你要进行服务器开发，还需要以下软件包：

- 从 Git 检出构建，或者修改了扫描器和解析器的实际定义文件时，需要 Flex 和 Bison。如果需要它们，请确保使用 Flex 2.5.31 或更高版本，以及 Bison 1.875 或更高版本。不能使用其他 lex 和 yacc 程序。

- 从 Git 检出构建，或者修改了使用 Perl 脚本的任何构建步骤的输入文件时，需要 Perl 5.8.3 或更高版本。在 Windows 上构建时，无论如何都需要 Perl。运行某些测试套件也需要 Perl。

如果你需要获取 GNU 软件包，可以从本地的 GNU 镜像站点获取（站点列表见 https://www.gnu.org/prep/ftp），或者直接从ftp://ftp.gnu.org/gnu/获取。

还要检查磁盘空间是否充足。编译期间，源码树需要约 100 MB，安装目录需要约 20 MB。空数据库集簇约占 35 MB；数据库所需空间大约是存储相同数据的纯文本文件的五倍。如果要运行回归测试，还会临时额外需要最多 150 MB。使用 `df` 命令检查可用磁盘空间。

16.3. 获取源代码

要获取 PostgreSQL 11.22 的源码，请访问网站的下载区：https://www.postgresql.org/download/。你应该取得一个名为 `postgresql-11.22.tar.gz` 或 `postgresql-11.22.tar.bz2` 的文件。取得文件后，将其解包：

```
gunzip postgresql-11.22.tar.gz
tar xf postgresql-11.22.tar
```

（可使用 `bunzip2` 代替 `gunzip` 来解压 `.bz2` 文件。）这会在当前目录下创建目录 `postgresql-11.22`，其中包含 PostgreSQL 源码。切换到该目录，继续完成其余安装过程。

你也可以直接从版本控制仓库获取源码，参见附录 I。

16.4. 安装过程

1. **配置**

   安装过程的第一步是为你的系统配置源码树，并选择所需选项。这通过运行以下脚本完成：`configure`。对于默认安装，只需输入：

   ```
   ./configure
   ```

   此脚本会运行一系列测试，确定各个依赖系统的变量的值，并检测操作系统的特殊之处，最后在构建树中创建若干文件来记录结果。你也可以运行 `configure` 时使用源码树之外的目录，以将构建目录单独存放。这种过程也称为 VPATH 构建。方法如下：

   ```
   mkdir build_dir
   cd build_dir
   /path/to/source/tree/configure [options go here]
   make
   ```

   默认配置将构建服务器和实用程序，以及所有只需要 C 编译器的客户端应用程序和接口。默认情况下，所有文件都会安装到 `/usr/local/pgsql` 之下。

   你可以通过提供以下一个或多个命令行选项来定制构建和安装过程，这些选项传给 `configure`：

   `--prefix=`*`PREFIX`*

   把所有文件安装到目录 *`PREFIX`* 下，而不是 `/usr/local/pgsql` 下。实际文件会安装到各个子目录中；不会有任何文件直接安装到 *`PREFIX`* 目录中。

   如果有特殊需求，还可以通过以下选项分别定制各个子目录。不过，如果保留它们的默认值，安装就可以重定位，也就是说，安装完成后可以移动目录。（`man` 和 `doc` 的位置不受此影响。）

   对于可重定位的安装，可以考虑使用 `configure` 的 `--disable-rpath` 选项。此外，还需要告知操作系统如何查找共享库。

`--exec-prefix=`*EXEC-PREFIX*

你可以把与体系结构相关的文件安装到与 *PREFIX* 不同的前缀 *EXEC-PREFIX* 下。这对于在多台主机之间共享与体系结构无关的文件很有用。如果省略该选项，那么 *EXEC-PREFIX* 会被设为与 *PREFIX* 相同，与体系结构相关和无关的文件都会安装到同一棵目录树下，这通常正是你想要的。

`--bindir=`*DIRECTORY*

指定可执行程序的目录。默认是 *EXEC-PREFIX*`/bin`，通常也就是 `/usr/local/pgsql/bin`。

`--sysconfdir=`*DIRECTORY*

设置各种配置文件的目录，默认是 *PREFIX*`/etc`。

`--libdir=`*DIRECTORY*

设置安装库和动态可加载模块的位置。默认是 *EXEC-PREFIX*`/lib`。

`--includedir=`*DIRECTORY*

设置安装 C 和 C++ 头文件的目录。默认是 *PREFIX*`/include`。

`--datarootdir=`*DIRECTORY*

设置多种只读数据文件的根目录。这只会为后续某些选项设置默认值。默认是 *PREFIX* `/share`。

`--datadir=`*DIRECTORY*

设置已安装程序使用的只读数据文件目录。默认是 *DATAROOTDIR*。注意，这与数据库文件实际存放的位置无关。

`--localedir=`*DIRECTORY*

设置安装区域设置数据的目录，尤其是消息翻译目录文件。默认是 *DATAROOTDIR* `/locale`。

`--mandir=`*DIRECTORY*

PostgreSQL 附带的手册页将安装到这个目录下各自对应的 `man`*x* 子目录中。默认是 *DATAROOTDIR*`/man`。

`--docdir=`*DIRECTORY*

设置安装文档文件（不包括“man”页）的根目录。这只会为后续选项设置默认值。该选项的默认值是 *DATAROOTDIR*`/doc/postgresql`。

`--htmldir=`*DIRECTORY*

PostgreSQL 的 HTML 格式文档将安装到这个目录。默认是 *DATAROOTDIR*。

> **注意**
>
> 为了能把 PostgreSQL 安装到共享安装位置（例如 `/usr/local/include`），同时又不干扰系统其他部分的名字空间，我们做了特别处理。首先，除非完整展开后的目录名已经包含字符串“`postgres`”或“`pgsql`”，否则会自动把字符串“`/postgresql`”追加到 `datadir`、`sysconfdir` 和 `docdir` 上。例如，如果你选择 `/usr/local` 作为前缀，那么文档会安装到 `/usr/local/doc/postgresql`；但如果前缀是 `/opt/postgres`，那么它会安装到 `/opt/postgres/doc`。客户端接口的公共 C 头文件会安装到 `includedir` 中，并且不会污染名字空间。内部头文件和服务器头文件则会安装到 `includedir` 下的私有目录中。关于如何访问这些头文件，请参见各接口自己的文档。最后，如果有需要，也会在 `libdir` 下创建私有子目录，用于存放动态可加载模块。

`--with-extra-version=`*`STRING`*

在 PostgreSQL 版本号后追加 *`STRING`*。例如，可以用它为从未发布的 Git 快照构建或包含自定义补丁的二进制文件添加额外版本字符串，如 `git describe` 标识符或发行版软件包的发布编号。

`--with-includes=`*`DIRECTORIES`*

`DIRECTORIES` 是一个以冒号分隔的目录列表，这些目录会被加入编译器搜索头文件的路径中。如果你把可选软件包（如 GNU Readline）安装在非标准位置，就必须使用此选项，并且通常还要使用对应的 `--with-libraries` 选项。

例如：`--with-includes=/opt/gnu/include:/usr/sup/include`。

`--with-libraries=`*`DIRECTORIES`*

`DIRECTORIES` 是一个以冒号分隔的目录列表，用于搜索库文件。如果你把软件包安装在非标准位置，很可能需要使用这个选项（以及对应的 `--with-includes`）。

例如：`--with-libraries=/opt/gnu/lib:/usr/sup/lib`。

`--enable-nls[=`*`LANGUAGES`*`]`

启用本地语言支持（NLS），也就是以非英语语言显示程序消息的能力。*`LANGUAGES`* 是可选的、以空格分隔的语言代码列表，例如 `--enable-nls='de fr'`。（你的列表与实际提供的翻译集合之间的交集会自动计算。）如果你不指定列表，则会安装所有可用的翻译。

要使用此选项，需要一个 Gettext API 实现；参见上文。

`--with-pgport=`*`NUMBER`*

将 *`NUMBER`* 设为服务器和客户端的默认端口号。默认值是 5432。端口号以后始终都可以修改，但如果在这里指定，那么服务器和客户端都会编译进同一个默认值，这可能很方便。通常选择非默认值的唯一合理原因，是你打算在同一台机器上运行多个 PostgreSQL 服务器。

`--with-perl`

构建 PL/Perl 服务器端语言。

`--with-python`

构建 PL/Python 服务器端语言。

`--with-tcl`

构建 PL/Tcl 服务器端语言。

`--with-tclconfig=`*`DIRECTORY`*

Tcl 会安装文件 `tclConfig.sh`，其中包含构建与 Tcl 交互的模块所需的配置信息。通常会在已知位置自动找到此文件，但如果要使用其他版本的 Tcl，可以指定查找它的目录。

`--with-gssapi`

构建时支持 GSSAPI 认证。在许多系统上，GSSAPI 系统（通常是 Kerberos 安装的一部分）并不安装在默认搜索的位置（如 `/usr/include`、`/usr/lib`），因此除了此选项，还必须使用 `--with-includes` 和 `--with-libraries`。`configure` 会检查所需的头文件和库，确保 GSSAPI 安装满足要求，然后才会继续。

`--with-krb-srvnam=`*`NAME`*

GSSAPI 使用的 Kerberos 服务主体的默认名称。默认值为 `postgres`。通常没有理由更改，除非使用 Windows 环境，此时必须将其设为大写的 `POSTGRES`。

`--with-llvm`

构建时支持基于 LLVM 的 JIT 编译（见第 32 章）。这要求已安装 LLVM 库。当前 LLVM 的最低要求版本是 3.9。

`llvm-config` 用于查找所需的编译选项。首先查找 `llvm-config`，然后查找 `llvm-config-$major-$minor`（遍历所有受支持版本），搜索位置为 `PATH`。如果这样无法找到正确的二进制文件，请使用 `LLVM_CONFIG` 指定正确程序的路径，即 `llvm-config`。例如：

```
./configure ... --with-llvm LLVM_CONFIG='/path/to/llvm/bin/llvm-config'
```

LLVM 支持要求有兼容的 `clang` 编译器（必要时通过环境变量 `CLANG` 指定），以及可工作的 C++ 编译器（必要时通过环境变量 `CXX` 指定）。

`--with-icu`

构建时支持 ICU 库。这要求已安装 ICU4C 软件包。当前 ICU4C 的最低要求版本是 4.2。

默认情况下，pkg-config 用于查找所需的编译选项。此方式支持 ICU4C 4.6 及更高版本。对于更早的版本，或者当 pkg-config 不可用时，可以将变量 `ICU_CFLAGS` 和 `ICU_LIBS` 指定给 `configure`，如下例所示：

```
./configure ... --with-icu ICU_CFLAGS='-I/some/where/include' ICU_LIBS='-L/some/where/
lib -licui18n -licuuc -licudata'
```

（如果 ICU4C 位于编译器的默认搜索路径中，仍需要指定一个非空字符串，以避免使用 pkg-config，例如 `ICU_CFLAGS=' '`。）

`--with-openssl`

构建时支持 SSL（加密）连接。这要求已安装 OpenSSL 软件包。`configure` 会检查所需的头文件和库，确保 OpenSSL 安装满足要求，然后才会继续。

`--with-pam`

构建时支持 PAM（可插拔认证模块）。

`--with-bsd-auth`

构建时支持 BSD 认证。（BSD 认证框架目前仅在 OpenBSD 上可用。）

`--with-ldap`

构建时支持 LDAP 认证和连接参数查找（更多信息见第 34.17 节和第 20.10 节）。在 Unix 上，这要求安装 OpenLDAP 软件包。在 Windows 上，使用默认的 WinLDAP 库。`configure` 会检查所需的头文件和库，以确认你的 OpenLDAP 安装满足要求后才继续。

`--with-systemd`

构建时支持 systemd 服务通知。如果服务器二进制文件由 systemd 启动，这可以改善集成；其他情况下没有影响；更多信息参见第 18.3 节。要使用此选项，需要安装 libsystemd 及相关头文件。

`--without-readline`

禁止使用 Readline 库（也禁止 libedit）。此选项会禁用 psql 中的命令行编辑和历史记录，因此不建议使用。

`--with-libedit-preferred`

优先使用 BSD 许可的 libedit 库，而不是 GPL 许可的 Readline。只有在两个库都已安装时，该选项才有意义；在那种情况下，默认行为是使用 Readline。

`--with-bonjour`

构建时支持 Bonjour。这要求操作系统支持 Bonjour。建议在 macOS 上启用。

`--with-uuid=`*`LIBRARY`*

构建 uuid-ossp 模块（它提供生成 UUID 的函数），并使用指定的 UUID 库。*`LIBRARY`* 必须是以下之一：

- `bsd`，使用 FreeBSD 以及其他某些 BSD 衍生系统中的 UUID 函数

- `e2fs`，使用 `e2fsprogs` 项目创建的 UUID 库；该库存在于大多数 Linux 系统和 mac OS 中，也可用于其他平台
- `ossp`，使用 OSSP UUID 库 [1]

`--with-ossp-uuid`

这是 `--with-uuid=ossp` 的过时等价写法。

`--with-libxml`

构建时支持 libxml2，以启用 SQL/XML 支持。这一特性要求 libxml2 2.6.23 或更高版本。

为检测所需的编译器和链接器选项，PostgreSQL 会查询 `pkg-config`，前提是已安装该程序且其中有 libxml2 的信息。否则，如果找到 libxml2 安装的 `xml2-config` 程序，就会使用它。优先使用 `pkg-config`，因为它能更好地处理多架构安装。

如果 libxml2 安装在不常见的位置，你可以设置与 `pkg-config` 相关的环境变量（见其文档），或把环境变量 `XML2_CONFIG` 设为指向该 libxml2 安装对应的 `xml2-config` 程序，或者设置变量 `XML2_CFLAGS` 和 `XML2_LIBS`。（如果安装了 `pkg-config`，那么要覆盖它对 libxml2 位置的判断，你必须设置 `XML2_CONFIG`，或同时把 `XML2_CFLAGS` 和 `XML2_LIBS` 设为非空字符串。）

`--with-libxslt`

构建 xml2 模块时使用 libxslt。xml2 依赖此库对 XML 执行 XSL 转换。

`--disable-float4-byval`

禁用 float4 值的“按值”传递，改为“按引用”传递。此选项会降低性能，但为了兼容用 C 编写并使用“版本 0”调用约定的旧用户定义函数，可能需要使用它。更好的长期解决办法是更新这些函数，使其使用“版本 1”调用约定。

`--disable-float8-byval`

禁用 float8 值的“按值”传递，改为“按引用”传递。此选项会降低性能，但为了兼容用 C 编写并使用“版本 0”调用约定的旧用户定义函数，可能需要使用它。更好的长期解决办法是更新这些函数，使其使用“版本 1”调用约定。注意，此选项不仅影响 float8，还影响 int8 和 timestamp 等相关类型。在 32 位平台上，默认使用 `--disable-float8-byval`，且不允许选择 `--enable-float8-byval`。

`--with-segsize=`*`SEGSIZE`*

设置段大小，单位为 GB。大型表会拆分为多个操作系统文件，每个文件的大小等于段大小。这避免了许多平台上的文件大小限制问题。默认段大小为 1 GB，在所有支持的平台上都是安全的。如果操作系统支持“大文件”（如今大多数系统都支持），可以使用更大的段大小。这有助于减少处理超大表时消耗的文件描述符数量。但应注意，不要选择超出平台和拟用文件系统支持范围的值。你可能要使用的其他工具（如 tar）也可能限制可用的文件大小。建议将此值设为 2 的幂，但这并非硬性要求。注意，更改此值需要执行 initdb。

[1] http://www.ossp.org/pkg/lib/uuid/

`--with-blocksize=`*`BLOCKSIZE`*

设置块大小，单位为 KB。这是表内存储和 I/O 的单位。默认值 8 KB 适用于大多数情况；在特殊情况下，其他值可能有用。该值必须是 1 到 32（KB）之间的 2 的幂。注意，更改此值需要执行 initdb。

`--with-wal-blocksize=`*`BLOCKSIZE`*

设置 WAL 块大小，单位为 KB。这是 WAL 日志内存储和 I/O 的单位。默认值 8 KB 适用于大多数情况；在特殊情况下，其他值可能有用。该值必须是 1 到 64（KB）之间的 2 的幂。注意，更改此值需要执行 initdb。

`--disable-spinlocks`

即使 PostgreSQL 不支持该平台的 CPU 自旋锁，也允许构建成功。缺少自旋锁支持会导致性能不佳；因此，仅应在构建中止并告知平台缺少自旋锁支持时使用此选项。如果在你的平台上构建 PostgreSQL 必须使用此选项，请向 PostgreSQL 开发者报告该问题。

`--disable-strong-random`

即使 PostgreSQL 不支持该平台上的强随机数，也允许构建成功。某些认证协议以及 pgcrypto 模块中的某些例程需要随机数来源。`--disable-strong-random` 会禁用要求密码学强随机数的功能，并使用较弱的伪随机数生成器来生成认证盐值和查询取消密钥。这可能降低认证的安全性。

`--disable-thread-safety`

禁用客户端库的线程安全性。这会使 libpq 和 ECPG 程序中的并发线程无法安全地控制各自私有的连接句柄。

`--with-system-tzdata=`*`DIRECTORY`*

PostgreSQL 自带了日期和时间操作所需的时区数据库。这个时区数据库实际上与很多操作系统（如 FreeBSD、Linux 和 Solaris）提供的 IANA 时区数据库兼容，因此再次安装它是多余的。使用此选项时，将使用位于 *`DIRECTORY`* 的、由系统提供的时区数据库，而不是 PostgreSQL 源码发布中自带的那一份。*`DIRECTORY`* 必须是绝对路径。在某些操作系统上，`/usr/share/zoneinfo` 是一个可能的目录。请注意，安装过程不会检测时区数据是否不匹配或有误。如果你使用该选项，建议运行回归测试，以验证你指定的时区数据能够与 PostgreSQL 正常配合工作。

这个选项主要面向那些非常了解其目标操作系统的二进制包发布者。使用该选项的主要优点是，当本地众多夏令时规则中的任何一条发生变化时，PostgreSQL 软件包都无需升级。另一个优点是，如果安装时不需要构建时区数据库文件，那么 PostgreSQL 的交叉编译也会更直接。

`--without-zlib`

禁止使用 Zlib 库。这会禁用 pg_dump 和 pg_restore 对压缩归档的支持。此选项仅用于没有该库的罕见系统。

`--enable-debug`

将所有程序和库编译为带调试符号的版本。这意味着你可以在调试器中运行程序，以分析问题。这会显著增大安装后的可执行文件大小，而且在非 GCC 编译器上，通常还会禁

用编译器优化，从而导致变慢。不过，保留这些符号对于处理可能出现的各种问题极其有帮助。目前，只有在你使用 GCC 的情况下，才建议在生产安装中使用该选项。但如果你在做开发工作或运行测试版，就应始终启用它。

`--enable-coverage`

如果使用 GCC，所有程序和库都会带代码覆盖率测试插桩进行编译。运行时，它们会在构建目录中生成包含代码覆盖率指标的文件。更多信息见第 33.5 节。该选项只适用于 GCC，并且仅在开发工作中使用。

`--enable-profiling`

如果使用 GCC，所有程序和库都会编译为可进行性能分析的形式。后端退出时，会创建一个子目录，其中包含用于性能分析的 `gmon.out` 文件。此选项仅用于使用 GCC 进行开发工作时。

`--enable-cassert`

在服务器中启用断言检查，用于测试许多“不可能发生”的条件。这对代码开发非常有价值，但这些测试可能会显著拖慢服务器速度。此外，启用这些测试并不一定会增强服务器稳定性！断言检查并未按严重程度分类，因此即使某个 bug 相对无害，只要触发了断言失败，仍会导致服务器重启。该选项不建议用于生产环境，但如果你在做开发工作或运行测试版，就应当启用它。

`--enable-depend`

启用自动依赖跟踪。启用后，makefile 会在任何头文件被修改时，重新构建所有受影响的目标文件。如果你在做开发工作，这很有用；但如果你只是打算编译一次并安装，这只是额外的开销。目前该选项只在 GCC 下有效。

`--enable-dtrace`

将 PostgreSQL 编译为支持动态跟踪工具 DTrace。更多信息见第 28.5 节。

可以设置环境变量 `DTRACE` 来指定 `dtrace` 程序。这通常是必要的，因为 `dtrace` 一般安装在 `/usr/sbin` 下，而该目录可能不在搜索路径中。

对于 `dtrace` 程序，可在以下环境变量中指定额外命令行选项：`DTRACEFLAGS`。在 Solaris 上，要在 64 位二进制文件中包含 DTrace 支持，必须指定 `DTRACEFLAGS="-64"` 给 configure。例如，使用 GCC 编译器时：

```
./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='-64' ...
```

使用 Sun 的编译器时：

```
./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --enable-dtrace DTRACEFLAGS='
-64' ...
```

`--enable-tap-tests`

启用使用 Perl TAP 工具的测试。这要求安装 Perl 以及 Perl 模块 `IPC::Run`。更多信息见第 33.4 节。

如果你希望使用不同于 configure 所选的 C 编译器，可以将环境变量 CC 设为所选程序。默认情况下，configure 会在 gcc 可用时选择它，否则选择平台默认编译器（通常为 cc）。类似地，如有需要，可以使用 CFLAGS 变量覆盖默认编译器标志。

可以在 configure 命令行上指定环境变量，例如：

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

下面列出可以按这种方式设置的重要变量：

BISON

Bison 程序

CC

C 编译器

CFLAGS

传给 C 编译器的选项

CLANG

在使用 --with-llvm 编译时，用于处理源代码以进行内联的 clang 程序路径

CPP

C 预处理器

CPPFLAGS

传给 C 预处理器的选项

CXX

C++ 编译器

CXXFLAGS

传给 C++ 编译器的选项

DTRACE

dtrace 程序的位置

DTRACEFLAGS

传给 dtrace 程序的选项

FLEX

Flex 程序

LDFLAGS

用于链接可执行文件或共享库的选项

LDFLAGS_EX

仅用于链接可执行文件的附加选项

LDFLAGS_SL

仅用于链接共享库的附加选项

LLVM_CONFIG

用于定位 LLVM 安装的 llvm-config 程序

MSGFMT

用于本地语言支持的 msgfmt 程序

PERL

Perl 解释器程序。它将用于确定构建 PL/Perl 所需的依赖。默认值是 perl。

PYTHON

Python 解释器程序。它将用于确定构建 PL/Python 所需的依赖。此外，这里指定（或以其他方式隐式选择）Python 2 还是 3，决定了可用的 PL/Python 语言变体。更多信息参见第 46.1 节。如果未设置，则会按以下顺序探测：python python3 python2。

TCLSH

Tcl 解释器程序。它将用于确定构建 PL/Tcl 所需的依赖，并会被代入 Tcl 脚本中。

XML2_CONFIG

用于定位 libxml2 安装的 xml2-config 程序

有时，事后补充编译器标志会很有用；原有标志由 configure 选定。一个重要的例子是，gcc 的 -Werror 选项不能包含在传给 configure 的 CFLAGS 中，因为这会破坏 configure 的许多内置测试。要添加此类标志，请将其放入 COPT 环境变量，再运行 make。COPT 的内容会被附加到 configure 设置好的 CFLAGS 和 LDFLAGS 上。例如，你可以执行：

```
make COPT='-Werror'
```

或者：

```
export COPT='-Werror'
make
```

> **注意**
>
> 开发服务器内部代码时，建议使用 configure 选项 `--enable-cassert`（开启许多运行时错误检查）和 `--enable-debug`（提高调试工具的实用性）。
>
> 如果使用 GCC，最好使用至少 `-O1` 的优化级别进行构建，因为不使用优化（`-O0`）会关闭一些重要的编译器警告（例如未初始化变量的使用）。不过，非零优化级别会让调试变得更复杂，因为单步执行编译后的代码时，通常无法与源代码行一一对应。如果你在调试优化后的代码时感到困惑，可以把感兴趣的特定文件用 `-O0` 重新编译。一个简单方法是向 make 传递选项：`make PROFILE=-O0 file.o`。
>
> `COPT` 和 `PROFILE` 环境变量实际上会被 PostgreSQL 的 makefile 以完全相同的方式处理。用哪个只是个人偏好问题，但开发者常见的习惯是把 `PROFILE` 用于一次性的标志调整，而 `COPT` 则可能长期保持设置。

2. **构建**

   要开始构建，输入以下任意一条命令：

   ```
   make
   make all
   ```

   （记得使用 GNU make。）构建需要几分钟，具体取决于硬件。显示的最后一行应该是：

   ```
   All of PostgreSQL successfully made. Ready to install.
   ```

   如果要构建所有可构建的内容，包括文档（HTML 和手册页）及附加模块（`contrib`），请改为输入：

   ```
   make world
   ```

   显示的最后一行应该是：

   ```
   PostgreSQL, contrib, and documentation successfully made. Ready to install.
   ```

   如果你希望构建所有可构建的内容，包括附加模块（`contrib`），但不包括文档，则改为输入：

   ```
   make world-bin
   ```

   如果你希望从另一个 makefile 中调用构建，而不是手工执行，那么必须取消设置 `MAKELEVEL` 或将其设为零，例如：

   ```
   build-postgresql:
           $(MAKE) -C postgresql MAKELEVEL=0 all
   ```

   否则可能会出现奇怪的错误信息，通常是关于缺少头文件。

3. **回归测试**

   如果你希望在安装之前测试新构建的服务器，可以在此时运行回归测试。回归测试是一套测试，用来验证 PostgreSQL 是否按开发者预期的方式在你的机器上运行。输入：

   ```
   make check
   ```

   （这不能以 root 身份运行；请使用非特权用户。）关于如何解释测试结果的详细信息，见第 33 章。在以后的任何时候，你都可以再次执行同样的命令来重复此测试。

4. **安装文件**

   > **注意**
   >
   > 如果你是在升级已有系统，请务必阅读第 18.6 节，其中包含关于升级集簇的说明。

   要安装 PostgreSQL，请输入：

   ```
   make install
   ```

   这会把文件安装到步骤 1 中指定的目录。请确保你有权限写入该区域。通常这一步需要以 root 身份执行。另一种做法是预先创建目标目录，并安排授予适当权限。

   要安装文档（HTML 和手册页），请输入：

   ```
   make install-docs
   ```

   如果你上面执行的是 world 构建，则改为输入：

   ```
   make install-world
   ```

   这也会安装文档。

   如果你上面执行的是不含文档的 world 构建，则改为输入：

   ```
   make install-world-bin
   ```

   你可以用 `make install-strip` 代替 `make install`，在安装时剥离可执行文件和库中的符号。这会节省一些空间。如果你在构建时启用了调试支持，那么剥离操作实际上会移除调试支持，因此只有在不再需要调试时才应该这样做。`install-strip` 会尽量合理地节省空间，但它并不知道如何从每个可执行文件中去掉所有不需要的字节，所以如果你希望尽可能节省所有磁盘空间，仍需要手工处理。

   标准安装会提供客户端应用开发以及服务器端程序开发（例如用 C 编写自定义函数或数据类型）所需的全部头文件。（在 PostgreSQL 8.0 之前，后者需要单独执行 `make install-all-headers` 命令，但此步骤现已并入标准安装。）

仅客户端安装：. 如果你只想安装客户端应用程序和接口库，可以使用以下命令：

```
make -C src/bin install
make -C src/include install
make -C src/interfaces install
make -C doc install
```

`src/bin` 中有少量仅供服务器使用的二进制文件，不过它们都很小。

卸载：. 要撤销安装，请使用命令 `make uninstall`。不过，这不会删除任何已创建的目录。

清理：. 安装完成后，你可以使用命令 `make clean` 从源码树中删除构建产生的文件，以释放磁盘空间。这会保留 `configure` 程序生成的文件，以便你之后可以用 `make` 重新构建全部内容。要把源码树重置到发布时的状态，请使用 `make distclean`。如果你打算在同一个源码树中为多个平台构建，就必须为每个平台执行这一步并重新配置。（另一种做法是为每个平台使用单独的构建树，这样源码树就不会被修改。）

如果你已经执行了一次构建，随后发现 `configure` 选项有误，或者修改了任何会被 `configure` 检查的内容（例如软件升级），那么在重新配置和重新构建之前先执行 `make distclean` 是一个好主意。否则，你对配置选项所做的更改可能不会传播到所有需要它们的地方。

16.5. 安装后设置

16.5.1. 共享库

在某些使用共享库的系统上，你需要告诉系统如何找到新安装的共享库。不需要这样做的系统包括 FreeBSD、HP-UX、Linux、NetBSD、OpenBSD 和 Solaris。

设置共享库搜索路径的方法因平台而异，但最常见的方法是按如下方式设置环境变量 `LD_LIBRARY_PATH`。对于 Bourne shell（`sh`、`ksh`、`bash`、`zsh`）：

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

或者对于 `csh` 或 `tcsh`：

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

请把这里的 `/usr/local/pgsql/lib` 替换为你在步骤 1 中通过 `--libdir` 设置的值。你应该把这些命令放入 shell 启动文件，例如 `/etc/profile` 或 `~/.bash_profile`。关于这种方法的注意事项，有一些不错的资料可参见 http://xahlee.info/UnixResource_dir/_/ldpath.html。

在某些系统上，更好的做法可能是在构建之前设置环境变量 `LD_RUN_PATH`。

在 Cygwin 上，请把库目录加入 `PATH`，或者把 `.dll` 文件移到 `bin` 目录中。

如果不确定，请参考你系统的手册页（可能是 ld.so 或 rld）。如果你随后收到类似下面这样的消息：

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

那就说明这一步确实是必需的。到那时再处理即可。

如果你使用的是 Linux 且拥有 root 权限，可以在安装后运行：

```
/sbin/ldconfig /usr/local/pgsql/lib
```

（或者等效目录），这样可以让运行时链接器更快地找到共享库。更多信息请参见 ldconfig 的手册页。在 FreeBSD、NetBSD 和 OpenBSD 上，相应命令是：

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

其他系统尚不清楚是否有等效命令。

16.5.2. 环境变量

如果你安装到了 /usr/local/pgsql，或者安装到了其他默认不会搜索程序的目录，那么应该把 /usr/local/pgsql/bin（或者你在步骤 1 中通过 --bindir 设置的值）加入你的 PATH。严格来说，这不是必需的，但它会让 PostgreSQL 的使用方便得多。

为此，请把下面内容加到你的 shell 启动文件中，例如 ~/.bash_profile（如果你希望影响所有用户，则用 /etc/profile）：

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

如果你使用的是 csh 或 tcsh，则使用这条命令：

```
set path = ( /usr/local/pgsql/bin $path )
```

要让系统能够找到 man 文档，除非你安装到了默认会搜索的位置，否则需要把类似下面的内容加到 shell 启动文件中：

```
MANPATH=/usr/local/pgsql/share/man:$MANPATH
export MANPATH
```

环境变量 PGHOST 和 PGPORT 会告诉客户端应用数据库服务器的主机和端口，从而覆盖编译时内置的默认值。如果你准备从远端运行客户端应用，那么让每个打算使用数据库的用户都设置 PGHOST 会比较方便。不过这不是必需的；对于大多数客户端程序，也可以通过命令行选项传递这些设置。

16.6. 受支持的平台

如果代码中包含了使某个平台（也就是一种 CPU 架构和操作系统的组合）工作的相关处理，并且最近已经在该平台上验证过能够成功构建并通过回归测试，那么 PostgreSQL 开发社区就认为该

平台受支持。目前，大多数平台兼容性测试都是由 PostgreSQL 构建农场[2]中的测试机器自动完成的。如果你有兴趣在某个平台上使用 PostgreSQL，而该平台尚未出现在构建农场中，但代码已经能够工作或可以被修改得可工作，我们强烈鼓励你设置一个构建农场成员机器，以便持续确保兼容性。

一般来说，可以预期 PostgreSQL 能在以下 CPU 架构上工作：x86、x86_64、IA64、Power PC、PowerPC 64、S/390、S/390x、Sparc、Sparc 64、ARM、MIPS、MIPSEL 和 PA-RISC。代码中包含对 M68K、M32R 和 VAX 的支持，但尚不清楚这些架构近期是否经过测试。对于不受支持的 CPU 类型，通常可以通过配置 `--disable-spinlocks` 来构建，但性能会很差。

可以预期 PostgreSQL 能在以下操作系统上工作：Linux（所有近期发行版）、Windows（Win2000 SP4 及更高版本）、FreeBSD、OpenBSD、NetBSD、macOS、AIX、HP/UX 和 Solaris。其他类 Unix 系统也可能可用，但目前未在测试。大多数情况下，某个操作系统所支持的全部 CPU 架构也都可以工作。尤其是在使用较旧系统时，请查看下面的第 16.7 节，看看是否有针对你的操作系统的特别说明。

如果你在某个平台上遇到安装问题，而根据近期构建农场的结果该平台是受支持的，请把问题报告到 `<pgsql-bugs@lists.postgresql.org>`。如果你有兴趣把 PostgreSQL 移植到一个新平台，那么 `<pgsql-hackers@lists.postgresql.org>` 是合适的讨论地点。

16.7. 平台相关说明

本节记录 PostgreSQL 安装和设置时的一些平台相关附加问题。请务必阅读安装说明，尤其是第 16.2 节。此外，也请参阅第 33 章中关于如何解释回归测试结果的说明。

这里未列出的平台，目前没有已知的平台特定安装问题。

16.7.1. AIX

PostgreSQL 可以在 AIX 上运行，但正确安装可能比较困难。AIX 4.3.3 至 6.1 被视为受支持版本。你可以使用 GCC 或原生 IBM 编译器 `xlc`。一般来说，使用较新的 AIX 和 PostgreSQL 版本会有所帮助。关于哪些 AIX 版本已知可用的最新信息，请查看构建农场。

对于受支持的 AIX 版本，建议至少达到以下修复级别：

AIX 4.3.3

维护级别 11，加上 ML11 后续补丁包

AIX 5.1

维护级别 9，加上 ML9 后续补丁包

AIX 5.2

技术级别 10，服务包 3

AIX 5.3

技术级别 7

AIX 6.1

基础级别

[2] https://buildfarm.postgresql.org/

要检查当前修复级别，在 AIX 4.3.3 至 AIX 5.2 ML 7 上使用 `oslevel -r`，在更高版本上使用 `oslevel -s`。

如果将 Readline 或 libz 安装在 `/usr/local`，除了自己的选项外，还应使用以下 `configure` 标志：`--with-includes=/usr/local/include --with-libraries=/usr/local/lib`。

16.7.1.1. GCC 问题

在 AIX 5.3 上，使用 GCC 编译和运行 PostgreSQL 曾出现一些问题。

应使用晚于 3.3.2 的 GCC 版本，尤其是在使用预打包版本时。我们使用 4.0.1 的效果很好。较早版本的问题似乎更多与 IBM 打包 GCC 的方式有关，而非 GCC 本身的问题，因此如果自行编译 GCC，使用较早版本的 GCC 也很可能成功。

16.7.1.2. Unix 域套接字不可用

AIX 5.3 存在一个问题：`sockaddr_storage` 的大小定义不足。在 5.3 版中，IBM 增大了 Unix 域套接字地址结构 `sockaddr_un` 的大小，却未相应增大 `sockaddr_storage`。结果是，尝试通过 Unix 域套接字使用 PostgreSQL 时，libpq 会写出该数据结构的边界。TCP/IP 连接正常，但 Unix 域套接字无法正常工作，导致回归测试无法运行。

此问题已报告给 IBM，缺陷报告编号为 PMR29657。升级到维护级别 5300-03 或更高版本即可包含此修复。一种快速解决办法是将 `/usr/include/sys/socket.h` 中的 `_SS_MAXSIZE` 改为 1025。无论采用哪种方法，在取得修正后的头文件后，都应重新编译 PostgreSQL。

16.7.1.3. 互联网地址问题

PostgreSQL 依赖系统的 `getaddrinfo` 函数解析 `listen_addresses`、`pg_hba.conf` 等位置中的 IP 地址。较旧的 AIX 版本在此函数中存在多种缺陷。如果遇到与这些设置有关的问题，更新到上面所列的适当 AIX 修复级别应该能够解决。

一位用户报告：

在 AIX 5.3 上部署 PostgreSQL 8.1 时，我们不时遇到统计信息收集器“莫名其妙地”无法成功启动的问题。这似乎是 IPv6 实现中意外行为所致。看起来在 AIX 5.3 上，PostgreSQL 与 IPv6 配合得不太好。

以下任意操作都能“修复”此问题。

- 删除 localhost 的 IPv6 地址：

```
(as root)
# ifconfig lo0 inet6 ::1/0 delete
```

- 从网络服务中移除 IPv6。在 AIX 上，文件 `/etc/netsvc.conf` 大致相当于 Solaris/Linux 上的 `/etc/nsswitch.conf`。AIX 上的默认设置如下：

  ```
  hosts=local,bind
  ```

  将其替换为：

  ```
  hosts=local4,bind4
  ```

  以停用对 IPv6 地址的搜索。

  警告

  这实际上只是针对 IPv6 支持不成熟所引发问题的临时解决办法，而 IPv6 支持在 AIX 5.3 的各次发布中已有明显改进。此方法在 AIX 5.3 上有效，但并不是优雅的解决方案。据报告，在 IPv6 支持更成熟的 AIX 6.1 上，这种做法不仅没有必要，还会导致问题。

16.7.1.4. 内存管理

AIX 的内存管理方式有些特殊。服务器可能有数 GB 乃至更多的空闲 RAM，但运行应用程序时仍会出现内存不足或地址空间错误。例如，加载扩展可能因不寻常的错误而失败。以下是以 PostgreSQL 安装所有者身份运行的例子：

```
=# CREATE EXTENSION plperl;
ERROR:  could not load library "/opt/dbs/pgsql/lib/plperl.so": A memory address is not in the
 address space for the process.
```

以 PostgreSQL 安装所属组中非所有者的成员身份运行：

```
=# CREATE EXTENSION plperl;
ERROR:  could not load library "/opt/dbs/pgsql/lib/plperl.so": Bad address
```

另一个例子是 PostgreSQL 服务器日志中的内存不足错误，此时每次接近或超过 256 MB 的内存分配都会失败。

这些问题的总体原因是服务器进程使用的默认位数和内存模型。默认情况下，在 AIX 上构建的所有二进制文件都是 32 位的。这与硬件类型或所用内核无关。这些 32 位进程的内存上限为 4 GB，按几种模型之一以 256 MB 的段布局。默认情况下，堆与栈共享一个段，因此堆可用空间不足 256 MB。

对于上面的 `plperl` 示例，请检查你的 umask 和 PostgreSQL 安装中二进制文件的权限。该示例涉及的二进制文件是 32 位的，安装时使用模式 750 而非 755。由于权限以这种方式设置，只有所有者或所属组的成员可以加载该库。因为它不是所有用户可读，加载器会将该对象放入进程的堆中，而不是通常应放置的共享库段。

这个问题的“理想”解决办法是使用 64 位构建的 PostgreSQL，但这并非总是可行，因为采用 32 位处理器的系统可以构建 64 位二进制文件，却无法运行它们。

如果需要 32 位二进制文件，请在启动 PostgreSQL 服务器之前，将 `LDR_CNTRL` 设为 `MAXDATA=0x`*n*`0000000`，其中 1 <= n <= 8，并尝试不同的值及 `postgresql.conf` 设置，找到能令人满意地工作的配置。这样使用 `LDR_CNTRL` 会告知 AIX，为服务器的堆预留 `MAXDATA` 字节，按 256 MB 的段分配。找到可用配置后，可以使用 `ldedit` 修改二进制文件，使其默

认使用所需的堆大小。也可以重新构建 PostgreSQL，传入 `configure LDFLAGS="-Wl,-bmaxdata:0x*n*0000000"` 来达到相同效果。

对于 64 位构建，将 `OBJECT_MODE` 设为 64，并向 `configure` 传入 `CC="gcc -maix64"` 和 `LDFLAGS="-Wl,-bbigtoc"`。（`xlc` 的选项可能不同。）如果未导出 `OBJECT_MODE`，构建可能因链接器错误而失败。设置 `OBJECT_MODE` 后，它会告知 AIX 的 `ar`、`as` 和 `ld` 等构建工具默认处理哪种对象。

默认情况下，可能发生分页空间过量分配。虽然我们尚未见过这种情况，但当内存耗尽且访问了过量分配的空间时，AIX 会终止进程。我们遇到过最接近的情况是，系统认为没有足够内存容纳另一个进程，导致 fork 失败。与 AIX 的许多其他部分一样，如果这成为问题，可以在系统或进程级别配置分页空间分配方式和内存不足时终止进程的行为。

16.7.2. Cygwin

可以使用 Cygwin 这个 Windows 上的类 Linux 环境来构建 PostgreSQL，但这种方式不如原生 Windows 构建（见第 17 章），且如今已不再推荐在 Cygwin 下运行服务器。

从源码构建时，请按照常规安装过程操作（即 `./configure; make` 等），同时注意以下 Cygwin 特有的差异：

- 请把路径设置成优先使用 Cygwin 的 bin 目录，而不是 Windows 工具目录。这有助于避免编译问题。
- 不支持 `adduser` 命令；请使用 Windows NT、2000 或 XP 中相应的用户管理应用。如果不是这些系统，则跳过此步骤。
- 不支持 `su` 命令；请在 Windows NT、2000 或 XP 上使用 ssh 模拟 su。如果不是这些系统，则跳过此步骤。
- 不支持 OpenSSL。
- 请启动 `cygserver` 以支持共享内存。为此，请输入命令 `/usr/sbin/cygserver &`。每次启动 PostgreSQL 服务器或初始化数据库集簇（`initdb`）时，该程序都必须在运行中。默认的 `cygserver` 配置可能需要修改（例如增大 `SEMMNS`），以防止 PostgreSQL 因系统资源不足而失败。
- 在某些使用非 C 区域设置的系统上，构建可能会失败。要修复这一点，请在构建前执行 `export LANG=C.utf8` 把区域设置改为 C，安装完 PostgreSQL 后再把它恢复为之前的设置。
- 并行回归测试（`make check`）可能因 `listen()` 的待处理连接队列溢出而误报回归测试失败；队列溢出会导致连接被拒绝的错误或挂起。可以使用 make 变量 `MAX_CONNECTIONS` 限制连接数，方法如下：

  ```
  make MAX_CONNECTIONS=5 check
  ```

  （在某些系统上，并发连接数最高可达约 10 个。）

可以把 `cygserver` 和 PostgreSQL 服务器安装为 Windows NT 服务。关于具体做法，请参阅 Cygwin 上 PostgreSQL 二进制包附带的 `README` 文档。它安装在 `/usr/share/doc/Cygwin` 目录中。

16.7.3. HP-UX

如果系统补丁级别和构建工具合适，PostgreSQL 7.3+ 应能在运行 HP-UX 10.X 或 11.X 的 Series 700/800 PA-RISC 机器上工作。至少有一位开发者经常在 HP-UX 10.20 上测试，我们也收到过在 HP-UX 11.00 和 11.11 上成功安装的报告。

除了 PostgreSQL 源码发行包，还需要 GNU make（HP 的 make 不可用），以及 GCC 或 HP 的完整 ANSI C 编译器。如果打算从 Git 源码而非发行 tar 包构建，还需要 Flex（GNU lex）和 Bison（GNU yacc）。我们也建议确保 HP 补丁相对较新。至少，在 HP-UX 11.11 上构建 64 位二进制文件时，可能需要 PHSS_30966（11.11）或其后续补丁，否则 `initdb` 可能挂起：

```
PHSS_30966  s700_800 ld(1) and linker tools cumulative patch
```

一般而言，应安装最新的 libc 和 ld/dld 补丁；如果使用 HP 的 C 编译器，也应安装最新的编译器补丁。请访问 HP 的支持站点，例如 ftp://us-ffs.external.hp.com/，免费获取最新补丁。

如果在 PA-RISC 2.0 机器上构建，并希望使用 GCC 生成 64 位二进制文件，则必须使用 64 位版本的 GCC。

如果在 PA-RISC 2.0 机器上构建，并希望编译后的二进制文件能在 PA-RISC 1.1 机器上运行，需要在 `CFLAGS` 中指定 `+DAportable`。

如果在 HP-UX Itanium 机器上构建，需要最新的 HP ANSI C 编译器及其依赖补丁或后续补丁：

```
PHSS_30848  s700_800 HP C Compiler (A.05.57)
PHSS_30849  s700_800 u2comp/be/plugin library Patch
```

如果同时安装了 HP 的 C 编译器和 GCC，可以在运行 `configure` 时显式选择要使用的编译器：

```
./configure CC=cc
```

这会选择 HP 的 C 编译器；或者执行：

```
./configure CC=gcc
```

这会选择 GCC。如果省略此设置，configure 在可以选择时会选用 `gcc` 作为编译器。

默认安装目标位置为 `/usr/local/pgsql`，你可能希望将其改为 `/opt` 下的某个位置。如果是这样，请为 `configure` 使用 `--prefix` 选项。

在回归测试中，几何测试可能存在一些低位数字差异，具体取决于所用编译器和数学库的版本。任何其他错误都值得怀疑。

16.7.4. macOS

要在 macOS 上从源代码构建 PostgreSQL，你需要安装 Apple 的命令行开发工具，可通过执行下列命令完成：

```
xcode-select --install
```

（注意，这会弹出一个 GUI 对话框要求确认。）你也可以视需要另外安装 Xcode。

在较新的 macOS 版本中，需要把 “sysroot” 路径嵌入到用于查找某些系统头文件的 include 开关中。这会导致 configure 脚本的输出，因为 configure 时所用 SDK 版本不同而变化。在简单场景下这通常不是问题；但如果你要做的是类似于在与服务器代码构建机器不同的另一台机器上构建扩展，就可能需要强制使用不同的 sysroot 路径。要这样做，请设置 `PG_SYSROOT`，例如：

```
make PG_SYSROOT=/desired/path all
```

要找出你机器上的合适路径，请运行：

```
xcrun --show-sdk-path
```

请注意，使用与构建核心服务器时不同的 sysroot 版本来构建扩展并不值得推荐；最坏情况下，这可能导致难以调试的 ABI 不一致。

你也可以在配置时通过向 configure 指定 `PG_SYSROOT`，选择非默认的 sysroot 路径：

```
./configure ... PG_SYSROOT=/desired/path
```

这主要适用于针对其他 macOS 版本进行交叉编译。不能保证生成的可执行文件能在当前主机上运行。

如果要完全禁止 `-isysroot` 选项，请使用：

```
./configure ... PG_SYSROOT=none
```

（任何不存在的路径名都可以。）如果你希望使用非 Apple 编译器进行构建，这可能会有用，但请注意，这种情况并未经过 PostgreSQL 开发者测试，也不受支持。

macOS 的 “系统完整性保护”（SIP）功能会破坏 `make check`，因为它会阻止将所需的 `DYLD_LIBRARY_PATH` 设置传给被测试的可执行文件。可以在 `make check` 之前先执行 `make install` 来规避这一问题。不过，大多数 Postgres 开发者会直接关闭 SIP。

16.7.5. MinGW/原生 Windows

Windows 版 PostgreSQL 可以使用 MinGW（用于 Microsoft 操作系统的类 Unix 构建环境）构建，也可以使用 Microsoft 的 Visual C++ 编译器套件构建。MinGW 构建方式使用本章介绍的常规构建系统；Visual C++ 构建方式完全不同，详见第 17 章。后者是完全原生的构建，不使用 MinGW 等附加软件。PostgreSQL 主网站上提供现成的安装程序。

原生 Windows 移植版本要求 32 位或 64 位的 Windows 2000 或更高版本。更早的操作系统没有足够的基础设施（但可以在其上使用 Cygwin）。可以从 http://www.mingw.org/下载 MinGW 这一类 Unix 构建工具，以及 MSYS 这一运行 `configure` 等 shell 脚本所需的 Unix 工具集。运行生成的二进制文件不需要它们；只有创建二进制文件时才需要。

要使用 MinGW 构建 64 位二进制文件，请从https://mingw-w64.org/安装 64 位工具集，将其 bin 目录放入 `PATH`，并使用 `--host=x86_64-w64-mingw32` 选项运行 `configure`。

安装好所有组件后，建议在 `CMD.EXE` 下运行 psql，因为 MSYS 控制台存在缓冲问题。

16.7.5.1. 在 Windows 上收集崩溃转储

如果 PostgreSQL 在 Windows 上崩溃，它能够生成 minidumps，可用于追踪崩溃原因，类似于 Unix 上的核心转储。这些转储可以使用 Windows Debugger Tools 或 Visual Studio 读取。

要在 Windows 上启用转储生成，请在集簇数据目录中创建一个名为 `crashdumps` 的子目录。随后，转储会以唯一名称写入该目录，该名称基于崩溃进程的标识符以及崩溃发生时的当前时间。

16.7.6. Solaris

PostgreSQL 在 Solaris 上有良好支持。你的操作系统越新，遇到的问题越少；详情见下文。

16.7.6.1. 所需工具

你可以使用 GCC 或 Sun 编译器套件来构建。为了获得更好的代码优化，在 SPARC 架构上强烈推荐使用 Sun 编译器。我们收到过使用 GCC 2.95.1 时出现问题的报告；建议使用 GCC 2.95.3 或更高版本。如果使用 Sun 编译器，请注意不要选择 `/usr/ucb/cc`；应使用 `/opt/SUNWspro/bin/cc`。

你可以从 https://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/下载 Sun Studio。许多 GNU 工具已经集成到 Solaris 10 中，或者包含在 Solaris 配套光盘中。如果你需要适用于较旧 Solaris 版本的软件包，可以到http://www.sunfreeware.com 查找这些工具。如果你更想要源码，请看 https://www.gnu.org/prep/ftp。

16.7.6.2. configure 报告测试程序失败

如果 `configure` 报告某个测试程序失败，这多半是因为运行时链接器找不到某些库，通常是 libz、libreadline，或其他非标准库如 libssl。要把它指向正确位置，请在 `configure` 命令行中设置环境变量 `LDFLAGS`，例如：

```
configure ... LDFLAGS="-R /usr/sfw/lib:/opt/sfw/lib:/usr/local/lib"
```

更多信息请参见 ld 手册页。

16.7.6.3. 64 位构建有时会崩溃

在 Solaris 7 及更早版本上，64 位 libc 的 `vsnprintf` 例程存在缺陷，导致 PostgreSQL 不定期发生核心转储。已知最简单的解决办法是强制 PostgreSQL 使用自带的 `vsnprintf`，而不是库中的版本。为此，在运行 `configure` 后，编辑由 `configure` 生成的文件：在 `src/Makefile.global` 中，将以下行：

```
LIBOBJS =
```

改为：

```
LIBOBJS = snprintf.o
```

（此变量中可能已经列有其他文件，顺序无关紧要。）然后照常构建。

16.7.6.4. 为获得最佳性能而编译

在 SPARC 架构上，强烈推荐使用 Sun Studio 进行编译。可以尝试使用 `-xO5` 优化标志，以生成显著更快的二进制文件。不要使用任何会改变浮点运算行为以及 `errno` 处理行为的标志（例如 `-fast`）。这些标志可能导致 PostgreSQL 出现不符合标准的行为，例如在日期和时间计算中。

如果你没有理由在 SPARC 上使用 64 位二进制文件，那么优先选择 32 位版本。64 位运算更慢，而且 64 位二进制文件也比 32 位变体慢。另一方面，在 AMD64 CPU 家族上，32 位代码并不是原生模式，因此 32 位代码在该 CPU 家族上会明显更慢。

16.7.6.5. 使用 DTrace 跟踪 PostgreSQL

是的，可以使用 DTrace。更多信息见第 28.5 节。

如果发现链接 `postgres` 可执行文件时中止，并出现类似以下错误消息：

```
Undefined                       first referenced
 symbol                             in file
AbortTransaction                    utils/probes.o
CommitTransaction                   utils/probes.o
ld: fatal: Symbol referencing errors. No output written to postgres
collect2: ld returned 1 exit status
make: *** [postgres] Error 1
```

则说明安装的 DTrace 太旧，无法处理静态函数中的探针。需要 Solaris 10u4 或更高版本。

第 17 章 在 Windows 上从源代码安装

建议大多数用户下载适用于 Windows 的二进制发行版，该发行版以图形化安装程序包的形式提供，可从PostgreSQL 网站获取。从源码构建仅面向开发 PostgreSQL 或扩展的人员。

在 Windows 上构建 PostgreSQL 有几种不同的方法。使用 Microsoft 工具进行构建的最简单方式，是安装Visual Studio 2022 并使用其自带的编译器。也可以使用完整的 Microsoft Visual C++ 2005 to 2022 进行构建。在某些情况下，除编译器之外还需要安装Windows SDK。

也可以使用 MinGW 提供的 GNU 编译器工具构建 PostgreSQL，或者在较旧版本的Windows 上使用 Cygwin。

使用 MinGW 或 Cygwin 构建时，采用的是常规构建系统，见第 16 章以
及第 16.7.5 节和第 16.7.2 节中的专门说明。要在这些环境中生成原生 64 位二进制文件，请使用 MinGW-w64 工具。这些工具也可用于在其他主机（例如 Linux 和 macOS）上为 32 位和 64 位 Windows 目标进行交叉编译。不建议使用 Cygwin 运行生产服务器，它只应用于原生构建无法工作的较旧版本Windows，例如Windows 98。官方二进制文件是使用 Visual Studio 构建的。

原生构建的 psql 不支持命令行编辑。Cygwin 构建支持命令行编辑，因此如果需要在Windows 上交互式使用 psql，应当使用它。

17.1. 使用 Visual C++ 或 Microsoft Windows SDK 构建

PostgreSQL 可以使用 Microsoft 的 Visual C++ 编译器套件构建。这些编译器既可以来自 Visual Studio、Visual Studio Express，也可以来自某些版本的Microsoft Windows SDK。如果你还没有配置好 Visual Studio 环境，最简单的办法是使用Visual Studio 2022 中的编译器，或者 Windows SDK 10 中的编译器，这两者都可从 Microsoft 免费下载。

使用 Microsoft 编译器套件既可以构建 32 位版本，也可以构建 64 位版本。32 位 PostgreSQL 可以使用 Visual Studio 2005 到 Visual Studio 2022 构建，也可以使用独立发布的 Windows SDK 6.0 到 10 版本。64 位 PostgreSQL 支持使用 6.0a 到 10 版本的 Microsoft Windows SDK 或 Visual Studio 2008 及以上版本构建。使用 Visual Studio 2005 到 Visual Studio 2013 构建时，编译最低支持到 Windows XP 和 Windows Server 2003。使用 Visual Studio 2015 构建时，最低支持到 Windows Vista 和 Windows Server 2008。使用 Visual Studio 2017 到 Visual Studio 2022 构建时，最低支持到 Windows 7 SP1 和 Windows Server 2008 R2 SP1。

使用 Visual C++或 Platform SDK 构建所需的工具位于 `src/tools/msvc` 目录中。构建时，请确保系统 PATH 中没有来自MinGW 或 Cygwin 的工具。同时，也要确保所需的 Visual C++ 工具都已在 PATH 中可用。在Visual Studio 中，启动Visual Studio Command Prompt。如果你想构建 64 位版本，就必须使用 64 位版本的命令提示符，反之亦然。在 Microsoft Windows SDK 中，启动开始菜单中 SDK 项下列出的 CMD shell。在较新的 SDK 版本中，可以使用 `setenv` 命令更改目标 CPU 架构、构建类型以及目标操作系统，例如使用 `setenv /x86 /release /xp` 来面向 Windows XP 或更高版本进行 32 位发行版构建。`setenv` 的其他选项见 `/?`。所有命令都应在 `src\tools\msvc` 目录中运行。

在构建之前，你可以创建config.pl 文件，以反映你想更改的任何配置选项，或要使用的任何第三方库的路径。完整配置是先读取并解析config_default.pl 文件，再应用来自 config.pl 的任何更改而得到的。例如，要指定Python 安装位置，可在config.pl 中写入以下内容：

```
$config->{python} = 'c:\python26';
```

你只需要指定那些与 config_default.pl 中不同的参数。

如果你需要设置其他环境变量，请创建一个名为buildenv.pl 的文件，并将所需命令放入其中。例如，如果 bison 不在 PATH 中，可以创建一个包含以下内容的文件，以将其路径加入 PATH：

```
$ENV{PATH}=$ENV{PATH} . ';c:\some\where\bison\bin';
```

如果要向 Visual Studio 构建命令（msbuild 或 vcbuild）传递额外的命令行参数：

```
$ENV{MSBFLAGS}="/m";
```

17.1.1. 需求

构建 PostgreSQL 需要以下附加产品。请使用 config.pl 文件指定这些库所在的目录。

Microsoft Windows SDK

如果你的构建环境未附带受支持版本的 Microsoft Windows SDK，建议升级到最新版本（目前为版本 10），可从https://www.microsoft.com/download 下载。

你必须始终包含 SDK 中的 Windows Headers and Libraries 部分。如果你安装的 Windows SDK 包含 Visual C++ Compilers，则无需Visual Studio 即可构建。请注意，从 8.0a 版本起，Windows SDK 不再附带完整的命令行构建环境。

ActiveState Perl

运行构建生成脚本需要 ActiveState Perl。MinGW 或 Cygwin Perl 都无法工作。它还必须出现在 PATH 中。二进制文件可从 https://www.activestate.com 下载（注意：需要 5.8.3 或更高版本，免费的标准发行版即可）。

以下附加产品并非入门所必需，但若要构建完整的软件包则需要它们。请使用config.pl 文件指定这些库所在的目录。

ActiveState TCL

构建 PL/Tcl 需要它（注意：需要 8.4 版本，免费的标准发行版即可）。

Bison 和 Flex

从 Git 构建需要 Bison 和 Flex，但从发布文件构建则不需要。只有 Bison 1.875 或 2.2 及以上版本可以工作。Flex 必须为 2.5.31 或更高版本。

Bison 和 Flex 都包含在 msys 工具套件中，可从http://www.mingw.org/wiki/MSYS 获取，它是MinGW 编译器套件的一部分。

除非它们已经在 PATH 中，否则你需要把包含`flex.exe` 和 `bison.exe` 的目录加入 `buildenv.pl` 中的 PATH 环境变量。对于 MinGW，该目录是你的 MinGW 安装目录下的 `\msys\1.0\bin` 子目录。

> **注意**
>
> 来自 GnuWin32 的 Bison 发行版似乎存在一个 bug：如果安装在名称中含有空格的目录中，Bison 就会工作异常，例如英文安装中的默认位置`C:\Program Files\GnuWin32`。请考虑安装到 `C:\GnuWin32`，或者在 PATH 环境设置中使用指向 GnuWin32 的 NTFS 短文件名路径（例如`C:\PROGRA~1\GnuWin32`）。

> **注意**
>
> PostgreSQL FTP 站点提供、且旧版文档提及的过时 `winflex` 二进制文件，在 64 位 Windows 主机上会报错 “flex: fatal internal error, exec failed” 。请改用 MSYS 中的 Flex。

Diff

运行回归测试需要 Diff，可从http://gnuwin32.sourceforge.net 下载。

Gettext

构建带有 NLS 支持的版本需要 Gettext，可从http://gnuwin32.sourceforge.net 下载。请注意，二进制文件、依赖项以及开发文件都是必需的。

MIT Kerberos

支持 GSSAPI 认证需要 MIT Kerberos。MIT Kerberos 可从 https://web.mit.edu/Kerberos/dist/index.html 下载。

libxml2 和 libxslt

XML 支持需要它们。二进制文件可从 https://zlatkovic.com/pub/libxml 下载，源代码可从http://xmlsoft.org 获取。请注意，libxml2 需要 iconv，而它可从相同的下载位置获得。

OpenSSL

SSL 支持需要它。二进制文件可从 https://slproweb.com/products/Win32OpenSSL.html 下载，源代码可从https://www.openssl.org 获取。

ossp-uuid

支持 UUID-OSSP 需要它（仅 contrib）。源代码可从 http://www.ossp.org/pkg/lib/uuid/下载。

Python

构建 PL/Python 需要它。二进制文件可从 https://www.python.org 下载。

zlib

pg_dump 和 pg_restore 中的压缩支持需要它。二进制文件可从 https://www.zlib.net 下载。

17.1.2. 64 位 Windows 的特殊注意事项

PostgreSQL 在 64 位 Windows 上只会为 x64 架构构建，不支持 Itanium 处理器。

不支持在同一个构建树中混用 32 位和 64 位版本。构建系统会自动检测自己是在 32 位还是 64 位环境中运行，并据此构建 PostgreSQL。因此，在开始构建之前启动正确的命令提示符非常重要。

若要使用服务器端第三方库，例如python 或 OpenSSL，该库也必须是 64 位的。不支持在 64 位服务器中加载 32 位库。PostgreSQL 支持的若干第三方库可能仅提供 32 位版本，在这种情况下，它们不能与 64 位 PostgreSQL 一起使用。

17.1.3. 构建

要以发布配置（默认值）构建 PostgreSQL 的全部内容，请运行以下命令：

```
build
```

要以调试配置构建 PostgreSQL 的全部内容，请运行以下命令：

```
build DEBUG
```

若只构建单个项目，例如 psql，请运行以下命令：

```
build psql
build DEBUG psql
```

若要将默认构建配置改为调试模式，请在`buildenv.pl` 文件中加入以下内容：

```
$ENV{CONFIG}="Debug";
```

也可以在 Visual Studio 图形界面中进行构建。在这种情况下，你需要先从命令提示符运行：

```
perl mkvcbuild.pl
```

然后在 Visual Studio 中打开生成的 `pgsql.sln`（位于源码树根目录）。

17.1.4. 清理和安装

大多数时候，Visual Studio 的自动依赖跟踪都会处理好变更过的文件。但如果变更较大，你可能需要清理安装。要执行此操作，只需运行`clean.bat` 命令，它会自动清除所有生成的文件。你也可以带上*`dist`* 参数运行它，此时它的行为类似于**`make distclean`**，并且也会删除 flex/bison 的输出文件。

默认情况下，所有文件都会写入debug 或 release 目录下的某个子目录。若要按标准布局安装这些文件，并同时生成初始化和使用数据库所需的文件，请运行以下命令：

```
install c:\destination\directory
```

如果你只想安装客户端应用程序和接口库，则可以使用以下命令：

```
install c:\destination\directory client
```

17.1.5. 运行回归测试

要运行回归测试，请先确保已完成所有必需部分的构建。此外，还要确保加载系统各部分所需的 DLL（例如过程语言所用的 Perl 和 Python DLL）都能在系统路径中找到。如果不能，请通过 buildenv.pl 文件设置路径。要运行测试，请在 src\tools\msvc 目录中执行以下命令之一：

```
vcregress check
vcregress installcheck
vcregress plcheck
vcregress contribcheck
vcregress modulescheck
vcregress ecpgcheck
vcregress isolationcheck
vcregress bincheck
vcregress recoverycheck
vcregress taptest
vcregress upgradecheck
```

要更改所用的测试调度方式（默认为 parallel），请将其追加到命令行，例如：

```
vcregress check serial
```

vcregress taptest 可用于运行目标目录中的 TAP 测试，例如：

```
vcregress taptest src\bin\initdb\
```

有关回归测试的更多信息，参见第 33 章。

对客户端程序运行回归测试时使用 vcregress bincheck，对恢复测试使用vcregress recoverycheck，或者运行由vcregress taptest 指定的 TAP 测试时，需要额外安装一个 Perl 模块：

IPC::Run

撰写本文时，IPC::Run 既未包含在 ActiveState Perl 安装中，也未包含在 ActiveState Perl 软件包管理器（PPM）的库中。要安装它，请下载以下文件：IPC-Run-<version>.tar.gz，这是 CPAN 提供的源码归档，地址为 https://metacpan.org/release/IPC-Run，然后将其解压。编辑 buildenv.pl 文件，并添加 PERL5LIB 变量，使其指向解压后归档中的 lib 子目录。例如：

```
$ENV{PERL5LIB}=$ENV{PERL5LIB} . ';c:\IPC-Run-0.94\lib';
```

部分 TAP 测试依赖一些外部命令，相关测试会根据这些命令是否可用而选择性触发。以下各变量都可以在 buildenv.pl 中设置或取消设置：

GZIP_PROGRAM

指向 gzip 命令的路径。默认值为 gzip，即在PATH 中找到的命令。

TAR

指向 tar 命令的路径。默认值为 tar，即在PATH 中找到的命令。

17.1.6. 构建文档

将 PostgreSQL 文档构建为 HTML 格式，需要若干工具和文件。请为这些文件创建一个根目录，并将它们存放在下列子目录中。

OpenJade 1.3.1-2

从 https://sourceforge.net/projects/openjade/files/openjade/1.3.1/openjade-1_3_1-2-bin.zip/download 下载，并解压到子目录 openjade-1.3.1 中。

DocBook DTD 4.2

从 https://www.oasis-open.org/docbook/sgml/4.2/docbook-4.2.zip 下载，并解压到子目录 docbook 中。

ISO 字符实体

从 https://www.oasis-open.org/cover/ISOEnts.zip 下载，并解压到子目录 docbook 中。

编辑 buildenv.pl 文件，并添加一个变量来指定根目录的位置，例如：

```
$ENV{DOCROOT}='c:\docbook';
```

要构建文档，请运行命令 builddoc.bat。注意，为了生成索引，实际上会构建两次。生成的 HTML 文件位于 doc\src\sgml。

第 18 章 服务器设置和操作

本章讨论如何设置和运行数据库服务器，以及它与操作系统的交互。

18.1. PostgreSQL 用户账户

与任何可从外部访问的服务器守护进程一样，建议使用单独的用户账户运行 PostgreSQL。该用户账户应该只拥有服务器管理的数据，不应与其他守护进程共用。（例如，使用 nobody 用户就不是好主意。）不建议将可执行文件安装为由该用户所有，因为系统一旦被攻破，就可能修改自己的二进制文件。

要在系统中添加一个 Unix 用户账户，请查找useradd 或 adduser 命令。通常使用的用户名是 postgres，本书也都按此假定，不过你也可以使用其他名称。

18.2. 创建一个数据库集簇

在做任何事情之前，必须先在磁盘上初始化一个数据库存储区域。我们把它称为数据库集簇。（SQL 标准使用术语目录集簇（catalog cluster）。）数据库集簇是由一个正在运行的数据库服务器实例管理的一组数据库。初始化之后，数据库集簇中会包含一个名为postgres 的数据库，作为供工具、用户和第三方应用使用的默认数据库。数据库服务器本身并不要求该postgres 数据库存在，但许多外部工具程序都假定它存在。初始化期间，集簇中还会创建另一个数据库，名为template1。顾名思义，它会作为后续新建数据库的模板；不应将它用于实际工作。（关于在集簇中创建新数据库的信息，见第 22 章。）

从文件系统角度看，数据库集簇就是一个目录，所有数据都存储在其下。我们将它称为数据目录或数据区域。数据存放位置完全由你决定，没有默认值；常用位置包括 /usr/local/pgsql/data 或/var/lib/pgsql/data。要初始化数据库集簇，请使用命令 initdb，该命令随以下产品安装：PostgreSQL。数据库集簇在文件系统中的目标位置通过 -D 选项指定，例如：

```
$ initdb -D /usr/local/pgsql/data
```

注意，必须先登录 PostgreSQL 用户账户，再执行此命令。该账户已在上一节介绍。

> **提示**
>
> 除了使用 -D 选项，也可以设置环境变量 PGDATA。

或者，也可以通过pg_ctl 程序运行 initdb，如下所示：

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

如果你打算用 pg_ctl 来启动和停止服务器（见第 18.3 节），这种方式可能更直观，因为这样 pg_ctl 就会成为你管理数据库服务器实例时使用的唯一命令。

如果指定的目录尚不存在，initdb 会尝试创建它。当然，如果initdb 对其父目录没有写权限，这会失败。通常建议让PostgreSQL 用户不仅拥有数据目录，也拥有其父目录，这样就不会有这个问题。如果目标父目录也不存在，则需要先创建它；如果祖父目录不可写，就需要使用 root 权限。因此，过程可能如下所示：

```
root# mkdir /usr/local/pgsql
root# chown postgres /usr/local/pgsql
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

如果数据目录已存在且其中已有文件，initdb 会拒绝运行；这是为了防止意外覆盖现有安装。

由于数据目录包含数据库中存储的全部数据，必须确保其不被未授权访问。因此，initdb 会撤销除 PostgreSQL 用户之外其他人的访问权限，也可以选择保留组访问权限。启用组访问时仅为只读。这允许与集簇属主同组的非特权用户备份集簇数据，或执行其他只需要读权限的操作。

请注意，在现有集簇上启用或禁用组访问时，必须先关闭集簇，并在重新启动PostgreSQL 前为所有目录和文件设置正确的权限模式。否则，数据目录中可能会混杂多种权限模式。对于只允许属主访问的集簇，适当的模式是目录为0700、文件为0600。对于还允许同组用户读取的集簇，适当的模式是目录为0750、文件为0640。

不过，尽管目录内容是安全的，默认的客户端认证设置却允许任何本地用户连接到数据库，甚至成为数据库超级用户。如果你不信任其他本地用户，我们建议在运行 initdb 时使用 -W、--pwprompt 或 --pwfile 之一，为数据库超级用户指定密码。另外，指定 -A md5 或 -A password，以避免使用默认的 trust 认证方式；或者在运行 initdb 之后、首次启动服务器之前修改生成的 pg_hba.conf 文件。（其他合理方法包括使用 peer 认证，或使用文件系统权限限制连接。更多信息见第 20 章。）

initdb 还会初始化数据库集簇的默认区域设置。通常，它会直接采用环境中的区域设置，并将其应用于初始化的数据库。可以为数据库指定不同的区域设置；更多信息见第 23.1 节。特定数据库集簇中使用的默认排序顺序由 initdb 设置，虽然你可以使用不同的排序顺序创建新数据库，但 initdb 创建的模板数据库所使用的顺序，只有删除并重建这些数据库才能更改。使用 C 或 POSIX 以外的区域设置还会影响性能。因此，一开始就作出正确选择很重要。

initdb 还会为数据库集簇设置默认字符集编码。通常应选择与区域设置相匹配的字符集编码。详见第 23.3 节。

非 C 和非 POSIX 的区域设置依赖操作系统的排序规则库来决定字符集排序。这会控制索引中存储键值的顺序。因此，无论是通过快照恢复、二进制流复制、切换到不同操作系统，还是升级操作系统，都不能让一个集簇切换到不兼容版本的排序规则库。

18.2.1. 二级文件系统的使用

很多安装会把数据库集簇建在机器“根”卷以外的文件系统（卷）上。如果这样做，不建议把二级卷的最顶层目录（挂载点）直接用作数据目录。最佳实践是在挂载点目录下创建一个由 PostgreSQL 用户拥有的目录，再在其下创建数据目录。这样可以避免权限问题，尤其是在执行 pg_upgrade 之类的操作时；同时也能确保二级卷脱机时以干净的方式失败。

18.2.2. 使用网络文件系统

许多安装会在网络文件系统上创建数据库集簇。有时通过 NFS 完成，有时则使用内部采用 NFS 的网络附加存储（NAS）设备。PostgreSQL 不会对 NFS 文件系统做特殊处理，也就是说，它假定 NFS 的行为与本地连接的驱动器完全一致。如果客户端或服务器端的 NFS 实现不提供标准文件系统语义，就可能导致可靠性问题（参见 https://www.time-travellers.org/shane/papers/

NFS_considered_harmful.html）。具体而言，对 NFS 服务器的延迟（异步）写入可能导致数据损坏。如果可能，应以同步方式（不使用缓存）挂载 NFS 文件系统，以避免这一风险。此外，不建议对 NFS 文件系统使用软挂载。

存储区域网络（SAN）通常使用 NFS 以外的通信协议，可能存在也可能不存在此类风险。建议查阅供应商文档中有关数据一致性保证的说明。PostgreSQL 的可靠性不可能高于它所使用的文件系统。

18.3. 启动数据库服务器

在任何人访问数据库之前，必须先启动数据库服务器。数据库服务器程序名为 `postgres`。其中，`postgres` 程序必须知道应到哪里查找它要使用的数据。这通过 `-D` 选项指定。因此，启动服务器最简单的方法是：

```
$ postgres -D /usr/local/pgsql/data
```

这会让服务器在前台运行。必须先登录 PostgreSQL 用户账户再执行此操作。如果不使用 `-D`，服务器会尝试使用环境变量所指定的数据目录，该变量为 `PGDATA`。如果也未提供该变量，则会失败。

通常最好在后台启动 `postgres`。做法是使用常见的 Unix shell 语法：

```
$ postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

如上所示，把服务器的stdout 和 stderr 输出保存到某个地方非常重要。这有助于审计，也有助于诊断问题。（关于日志文件处理的更深入讨论，见第 24.3 节。）

`postgres` 还接受许多其他命令行选项。更多信息请见 postgres 参考页以及下面的第 19 章。

这些 shell 语法很快就会让人觉得繁琐。因此提供了包装器程序 pg_ctl 来简化一些任务。例如：

```
pg_ctl start -l logfile
```

这会在后台启动服务器，并把输出写入指定的日志文件。这里的 `-D` 选项含义与在 `postgres` 中相同。`pg_ctl` 还可用于停止服务器。

通常，你会希望在计算机启动时就启动数据库服务器。自动启动脚本依赖于操作系统。PostgreSQL 在 `contrib/start-scripts` 目录中附带了一些示例脚本。安装这些脚本需要 root 权限。

不同系统在引导时启动守护进程的惯例各不相同。许多系统有`/etc/rc.local` 或`/etc/rc.d/rc.local` 文件，其他系统则使用`init.d` 或 `rc.d` 目录。无论采用哪种方式，服务器都必须由PostgreSQL 用户账户而不是 root 或其他用户来启动。因此，你大概应该在命令中使用`su postgres -c '...'` 这种形式。例如：

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

下面是一些与操作系统相关的补充建议。（在每种情况下，请确保使用正确的安装目录和用户名；这里展示的是通用值。）

- 对于 FreeBSD，请查看PostgreSQL 源码发布包中的 `contrib/start-scripts/freebsd` 文件。

- 在 OpenBSD 上，把以下内容加入/etc/rc.local：

```
if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
    su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/postgresql/log -D /usr/
local/pgsql/data'
    echo -n ' postgresql'
fi
```

- 在 Linux 系统上，可以把

  ```
  /usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
  ```

  加入/etc/rc.d/rc.local 或/etc/rc.local，或者查看PostgreSQL 源码发布包中的 contrib/start-scripts/linux 文件。

  使用 systemd 时，可以采用下面的服务单元文件（例如放在/etc/systemd/system/postgresql.service）：

  ```
  [Unit]
  Description=PostgreSQL database server
  Documentation=man:postgres(1)
  After=network-online.target
  Wants=network-online.target

  [Service]
  Type=notify
  User=postgres
  ExecStart=/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
  ExecReload=/bin/kill -HUP $MAINPID
  KillMode=mixed
  KillSignal=SIGINT
  TimeoutSec=infinity

  [Install]
  WantedBy=multi-user.target
  ```

  使用 Type=notify 要求服务器二进制文件在构建时启用了 configure --with-systemd。

  请仔细考虑超时设置。撰写本文时，systemd 的默认超时为 90 秒，如果进程在这段时间内未报告已就绪，它就会将其杀死。但一个在启动时可能需要执行崩溃恢复的PostgreSQL 服务器，准备就绪可能远不止这么久。建议值infinity 可以禁用该超时逻辑。

- 在 NetBSD 上，可根据偏好使用FreeBSD 或 Linux 的启动脚本。

- 在 Solaris 上，创建一个名为/etc/init.d/postgresql 的文件，其中包含以下内容：

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data"
  ```

  然后，在/etc/rc3.d 中为它创建一个名为 S99postgresql 的符号链接。

当服务器运行时，其PID 会保存在数据目录中的 postmaster.pid 文件里。这可以防止多个服务器实例在同一个数据目录上运行，也可以用来关闭服务器。

18.3.1. 服务器启动失败

服务器启动失败有几种常见原因。请检查服务器日志文件，或者手工启动服务器（不要重定向标准输出和标准错误），看看出现了什么错误消息。下面将更详细地解释一些最常见的错误消息。

```
LOG:  could not bind IPv4 address "127.0.0.1": Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds and
 retry.
FATAL:  could not create any TCP/IP sockets
```

这通常就是字面上的意思：你试图在一个已有服务器运行的端口上再启动另一个服务器。不过，如果内核错误消息不是Address already in use或类似变体，也可能是别的问题。例如，试图在一个保留端口上启动服务器，可能会得到类似下面的消息：

```
$ postgres -p 666
LOG:  could not bind IPv4 address "127.0.0.1": Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

像下面这样的消息：

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

可能表示内核对共享内存大小的限制小于 PostgreSQL 尝试创建的工作区（本例中为 4011376640 字节）。也可能表示内核中根本没有配置 System V 风格的共享内存支持。作为临时解决办法，可以尝试以少于通常数量的缓冲区启动服务器（shared_buffers）。最终还是需要重新配置内核，增大允许的共享内存大小。如果尝试在同一台机器上启动多个服务器，而它们请求的总空间超出内核限制，也可能看到这条消息。

像下面这样的错误：

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

并不意味着你已经用光了磁盘空间。它表示你的内核对 System V 信号量数量的限制，小于PostgreSQL 想要创建的数量。和上面一样，你也许可以通过减少允许的连接数（max_connections）来暂时绕过这个问题，但最终还是应该提高内核限制。

如果收到 “illegal system call” 错误，很可能是内核根本不支持共享内存或信号量。在这种情况下，唯一的办法就是重新配置内核以启用这些功能。

关于配置 System V IPC 功能的细节请见第 18.4.1 节。

18.3.2. 客户端连接问题

客户端一侧可能出现的错误种类很多，并且依赖于具体应用，但其中有一些可能直接与服务器的启动方式有关。除下面列出的几种情况外，其他问题应查阅相应客户端应用的文档。

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

这种常见的失败表示 “我找不到可通信的服务器”。尝试 TCP/IP 通信时，它就会显示为上述形式。一个常见错误是忘记配置服务器以允许 TCP/IP 连接。

另一种情况是，尝试通过 Unix 域套接字与本地服务器通信时，会得到以下消息：

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

最后一行有助于确认客户端是否正尝试连接到正确的位置。如果那里实际上没有运行服务器，内核错误消息通常会像示例那样是 `Connection refused` 或 `No such file or directory`。（要注意，在这里 `Connection refused` 并不表示服务器收到了你的连接请求并拒绝了它；那种情况会产生另一条消息，如第 20.15 节所示。）其他错误消息，例如 `Connection timed out`，可能表示更底层的问题，例如网络不通。

18.4. 管理内核资源

PostgreSQL 有时会达到操作系统的各种资源上限，尤其是在同一系统上运行多个服务器实例，或在非常大型的安装环境中时更是如此。本节解释PostgreSQL 使用的内核资源，以及你可以采取哪些步骤来解决与内核资源消耗相关的问题。

18.4.1. 共享内存和信号量

PostgreSQL 要求操作系统提供进程间通信（IPC）机制，特别是共享内存和信号量。Unix 派生系统通常提供“System V”IPC、“POSIX”IPC，或者两者兼有。Windows 对这些特性有自己的实现，这里不作讨论。

完全缺少这些功能时，通常会在服务器启动时出现“Illegal system call”错误。在这种情况下，除了重新配置内核别无选择。没有它们，PostgreSQL 就无法工作。不过，在现代操作系统中，这种情况很少见。

服务器启动时，PostgreSQL 通常分配极少量的 System V 共享内存，以及大量 POSIX（`mmap`）共享内存。此外，服务器启动时还会创建大量信号量，这些信号量可以是 System V 风格或 POSIX 风格。目前，Linux 和 FreeBSD 系统使用 POSIX 信号量，而其他平台使用 System V 信号量。

> **注意**
>
> 在 PostgreSQL 9.3 之前，只使用 System V 共享内存，因此启动服务器所需的 System V 共享内存量大得多。如果你运行的是较旧版本的服务器，请查阅相应服务器版本的文档。

System V IPC 功能通常受系统范围的分配限制约束。当 PostgreSQL 超出这些限制之一时，服务器会拒绝启动，并应留下带有指导性的错误消息，说明问题所在以及应如何处理（另见第 18.3.1 节）。相关内核参数在不同系统上的命名基本一致，表 18.1 给出了概览；不过，设置它们的方法却各不相同。下面给出一些平台上的建议。

表 18.1. System V IPC 参数

名称	描述	运行一个PostgreSQL实例所需的值
`SHMMAX`	共享内存段的最大尺寸（字节）	至少 1kB，但是默认值通常高得多
`SHMMIN`	共享内存段的最小尺寸（字节）	1

名称	描述	运行一个PostgreSQL实例所需的值
SHMALL	可用共享内存的总量（字节或页面）	如果是字节，同SHMMAX；如果是页面，为ceil(SHMMAX/PAGE_SIZE)，加上其他应用程序的空间
SHMSEG	每个进程的最大共享内存段数目	只需要 1 段，但是默认值高很多
SHMMNI	系统范围内的最大共享内存段数目	像SHMSEG 外加其他应用的空间
SEMMNI	信号量标识符（即，集合）的最大数目	至少 ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16)，并为其他应用保留空间
SEMMNS	系统范围内的最大信号量数目	ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16) * 17，并为其他应用保留空间
SEMMSL	每个集合中信号量的最大数目	至少 17
SEMMAP	信号量映射中的项数	见文本
SEMVMX	信号量的最大值	至少 1000（默认值常常是 32767，如非必要不要更改）

PostgreSQL 的每个服务器实例都需要少量 System V 共享内存（在 64 位平台上通常为 48 字节）。大多数现代操作系统都能轻松分配这个数量。但是，如果运行许多服务器实例，或者其他应用程序也在使用 System V 共享内存，可能需要增大 SHMALL，它是系统范围内 System V 共享内存的总量。注意，在许多系统上，SHMALL 以页而非字节为单位。

不太可能出问题的是共享内存段的最小尺寸（SHMMIN），对PostgreSQL 来说应该最多大约是 32 字节（通常只是 1）。而系统范围（SHMMNI）或每个进程（SHMSEG）的最大共享内存段数目不太可能会导致问题，除非你的系统把它们设成零。

使用 System V 信号量时，PostgreSQL 为每个允许的连接（max_connections）、自动清理工作进程（autovacuum_max_workers）和后台进程（max_worker_processes）各使用一个信号量，每 16 个组成一组。每组还包含第 17 个信号量，其中存放一个“魔数”，用来检测与其他应用程序使用的信号量集的冲突。系统中的信号量最大数量由 SEMMNS 设置，因此它必须至少等于 max_connections、autovacuum_max_workers 和 max_worker_processes 之和，再为每 16 个允许的连接和工作进程额外增加一个信号量（参见表 18.1 中的公式）。参数 SEMMNI 限制系统中同时存在的信号量集的数量。因此，此参数必须至少为 ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16)。降低允许的连接数，可以临时规避 semget 函数的失败；此类失败通常会报告令人困惑的“No space left on device”。

在某些情况下，可能还需要增大SEMMAP，使其至少与SEMMNS 处于同一数量级。如果系统提供这个参数（很多系统没有），它定义的是信号量资源映射的大小；映射中每个连续的可用信号量块都需要占用一项。每当一个信号量集合被释放时，它要么会并入与该释放块相邻的现有项，要么会登记为一个新的映射项。如果映射已满，被释放的信号量就会丢失（直到重启）。因此，随着时间推移，信号量空间碎片化可能会导致可用信号量少于应有数量。

与“信号量撤销”有关的其他各种设置，如SEMMNU 和 SEMUME，不会影响PostgreSQL。

使用 POSIX 信号量时，所需数量与 System V 相同，即每个允许的连接（max_connections）、自动清理工作进程（autovacuum_max_workers）和后台进程（max_worker_processes）各需一个信号量。在优先使用此实现的平台上，POSIX 信号量数量没有特定的内核限制。

AIX

至少从 5.1 版起，应该不需要对 `SHMMAX` 等参数进行任何特殊配置，因为它似乎已配置为允许将全部内存用作共享内存。这也是 DB/2 等其他数据库常用的配置。

不过，可能需要修改 `/etc/security/limits` 中的全局 `ulimit` 信息，因为文件大小（`fsize`）和文件数量（`nofiles`）的默认硬限制可能过低。

FreeBSD

可以使用 `sysctl` 或 `loader` 接口更改默认 IPC 配置。下列参数可用 `sysctl` 设置：

```
# sysctl kern.ipc.shmall=32768
# sysctl kern.ipc.shmmax=134217728
```

要让这些设置在重启之后也保持，请修改`/etc/sysctl.conf`。

这些信号量相关设置对于 `sysctl` 是只读的；设置它们时，可修改 `/boot/loader.conf`：

```
kern.ipc.semmni=256
kern.ipc.semmns=512
```

修改该文件后，需要重启才能使新设置生效。

你可能还希望配置内核，把共享内存锁定在 RAM 中，防止其被换出到交换区。这可以通过 `sysctl` 设置 `kern.ipc.shm_use_phys` 来实现。

如果通过启用 sysctl 的 `security.jail.sysvipc_allowed` 在 FreeBSD jail 中运行，应让不同 jail 中的 postmaster 使用不同的操作系统用户。这能提高安全性，因为它可以防止非 root 用户干扰不同 jail 中的共享内存或信号量，也能让 PostgreSQL 的 IPC 清理代码正常工作。（在 FreeBSD 6.0 及更高版本中，IPC 清理代码无法正确检测其他 jail 中的进程，导致无法在不同 jail 中使用同一端口运行 postmaster。）

4.0 之前的 FreeBSD 版本与旧版 OpenBSD 的行为相同（见下文）。

NetBSD

在 NetBSD 5.0 及更高版本中，可以通过以下命令调整 IPC 参数：`sysctl`。例如：

```
# sysctl -w kern.ipc.semmni=100
```

要让这些设置在重启后仍然生效，请修改 `/etc/sysctl.conf`。

通常需要增大 `kern.ipc.semmni` 和 `kern.ipc.semmns`，因为 NetBSD 对这两项的默认设置小得令人担忧。

你可能还希望配置内核，把共享内存锁定在 RAM 中，防止其被换出到交换区。这可以通过 `sysctl` 设置 `kern.ipc.shm_use_phys` 来实现。

5.0 之前的 NetBSD 版本与旧版 OpenBSD 的行为相同（见下文），但设置内核参数时应使用关键字 `options`，而非 `option`。

OpenBSD

在 OpenBSD 3.3 及更高版本中，可以通过以下命令调整 IPC 参数：`sysctl`。例如：

```
# sysctl kern.seminfo.semmni=100
```

要让这些设置在重启后仍然生效，请修改 `/etc/sysctl.conf`。

通常需要增大 `kern.seminfo.semmni` 和 `kern.seminfo.semmns`，因为 OpenBSD 对这两项的默认设置小得令人担忧。

在较旧的 OpenBSD 版本中，需要构建自定义内核才能更改 IPC 参数。请确保以下选项：`SYSVSHM` 和 `SYSVSEM` 也已启用（它们默认启用）。以下示例展示了如何在内核配置文件中设置各项参数：

```
option          SYSVSHM
option          SHMMAXPGS=4096
option          SHMSEG=256

option          SYSVSEM
option          SEMMNI=256
option          SEMMNS=512
option          SEMMNU=256
```

HP-UX

默认设置通常足以满足普通安装的需求。在 HP-UX 10 上，`SEMMNS` 的出厂默认值为 128，对较大的数据库站点来说可能过低。

可以在 System Administration Manager（SAM）的 Kernel Configuration → Configurable Parameters 中设置 IPC 参数。完成后选择 Create A New Kernel。

Linux

默认最大段大小为 32 MB，默认最大总大小为 2097152 页。除使用“大页”的特殊内核配置外，页大小几乎总是 4096 字节（使用 `getconf PAGE_SIZE` 验证）。

可以通过以下接口更改共享内存大小设置：`sysctl`。例如，要允许 16 GB：

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

此外，可以将这些设置保存在以下文件中，使其在重启后仍然生效：`/etc/sysctl.conf`。强烈建议这样做。

很旧的发行版可能没有 `sysctl` 程序，但可以通过操作 `/proc` 文件系统进行等效更改：

```
$ echo 17179869184 >/proc/sys/kernel/shmmax
$ echo 4194304 >/proc/sys/kernel/shmall
```

其他默认值都相当充裕，通常不需要更改。

macOS

在 macOS 中配置共享内存的推荐方法是创建一个名为/etc/sysctl.conf 的文件，其中包含这样的变量赋值：

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

注意，在某些 macOS 版本中，全部五个共享内存参数都必须在/etc/sysctl.conf 中设置，否则这些值会被忽略。

请注意，近期 macOS 版本会忽略将 SHMMAX 设为非 4096 整数倍的值的尝试。

在这个平台上，SHMALL 以 4 kB 的页为单位。

在较旧的 macOS 版本中，必须重启才能使共享内存参数的变更生效。从 10.5 起，除了 SHMMNI，都可以使用 sysctl 在线更改。但最好仍通过 /etc/sysctl.conf 设置所需的值，以便重启后保留这些值。

文件 /etc/sysctl.conf 只在 macOS 10.3.9 及更高版本中生效。如果运行的是更早的 10.3.x 版本，必须编辑文件 /etc/rc 并更改以下命令中的值：

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

注意，/etc/rc 通常会被 macOS 系统更新覆盖，因此每次更新后可能都需要重新进行这些编辑。

在 macOS 10.2 及更早版本中，请改为编辑 /System/Library/StartupItems/SystemTuning/SystemTuning 文件中的这些命令。

Solaris 2.6 至 2.9（Solaris 6 至 Solaris 9）

可以在以下文件中更改相关设置：`/etc/system`。例如：

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

需要重启才能使变更生效。另请参见 http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html，了解旧版 Solaris 中共享内存的信息。

Solaris 2.10（Solaris 10）及更高版本
OpenSolaris

在 Solaris 10 及更高版本和 OpenSolaris 中，默认共享内存和信号量设置足以满足大多数 PostgreSQL 应用的需求。Solaris 现在默认将 `SHMMAX` 设为系统 RAM 的四分之一。要进一步调整此设置，请使用与以下用户关联的项目设置：`postgres`。例如，执行以下命令时使用的用户为 `root`：

```
projadd -c "PostgreSQL DB User" -K "project.max-shm-memory=(privileged,8GB,deny)" -U
 postgres -G postgres user.postgres
```

该命令会新增 `user.postgres` 项目，并把`postgres` 用户的共享内存上限设为 8GB；它会在该用户下次登录时或重启 PostgreSQL 时（不是重新加载）生效。上述示例假定 PostgreSQL 由 `postgres` 组中的 `postgres` 用户运行。不需要重启操作系统。

对于会承载大量连接的数据库服务器，我们还建议修改以下内核设置：

```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
project.max-msg-ids=(priv,4096,deny)
```

此外，如果你正在某个区（zone）中运行PostgreSQL，可能也需要提高该区的资源使用限制。关于`projects` 和 `prctl` 的更多信息，请参见System Administrator's Guide 中的 "Chapter 2: Projects and Tasks"。

18.4.2. systemd RemoveIPC

如果使用 systemd，必须注意不要让操作系统过早移除 IPC 资源（包括共享内存）。这在从源码安装 PostgreSQL 时尤其值得关注。使用发行版软件包的用户较不容易受到影响，因为此时 `postgres` 用户通常会被创建为系统用户。

`logind.conf` 中的 `RemoveIPC` 设置控制当用户完全注销时是否移除 IPC 对象。系统用户不受此限制。原版 systemd 默认开启该设置，但某些操作系统发行版默认将其关闭。

当此设置开启时，一个常见现象是，用于并行查询执行的共享内存对象会在看似随机的时间被删除，导致在尝试打开或删除它们时出现错误和警告，例如

```
WARNING:  could not remove shared memory segment "/PostgreSQL.1450751626": No such file or
 directory
```

不同类型的 IPC 对象（共享内存与信号量、System V 与 POSIX）在 systemd 中的处理略有不同，因此你可能会发现某些 IPC 资源不会像其他资源那样被删除。但依赖这些细微差别并不可取。

“用户注销”可能会作为维护作业的一部分发生，也可能在管理员以`postgres` 用户登录或执行类似操作时手工触发，因此通常很难彻底防止。

什么算作“系统用户”，是在编译 systemd 时依据 `/etc/login.defs` 中的 `SYS_UID_MAX` 设置确定的。

打包和部署脚本应当谨慎地使用 `useradd -r`、`adduser --system` 或其等价方式，把 `postgres` 用户创建为系统用户。

或者，如果该用户账户创建得不正确，或无法更改，建议在`/etc/systemd/logind.conf` 或其他适当的配置文件中设置

```
RemoveIPC=no
```

> **小心**
>
> 上述两件事至少要确保做到其中之一，否则 PostgreSQL 服务器会变得非常不可靠。

18.4.3. 资源限制

类 Unix 操作系统会施加多种资源限制，这些限制可能干扰 PostgreSQL 服务器的运行。其中尤其重要的是：每个用户可用的进程数限制、每个进程可打开文件数限制，以及每个进程可用内存量限制。每一种限制都有“硬”限制和“软”限制。实际生效的是软限制，但用户可以自行把它调高到不超过硬限制；硬限制则只能由 root 用户修改。系统调用`setrlimit` 负责设置这些参数。shell 的内置命令 `ulimit`（Bourne shell）或 `limit`（csh）可用于在命令行控制资源限制。在 BSD 派生系统上，`/etc/login.conf` 文件控制登录时设置的各种资源限制。详细信息请参阅操作系统文档。相关参数有 `maxproc`、`openfiles` 和 `datasize`。例如：

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

（`-cur` 表示软限制；把它改为`-max` 则表示设置硬限制。）

内核还可能对某些资源施加系统范围的限制。

- 在 Linux 上，内核参数 `fs.file-max` 确定内核支持的最大打开文件数。可以使用 `sysctl -w fs.file-max=`*`N`* 来修改它。若要在重启后保持该设置，请在`/etc/sysctl.conf` 中

添加相应赋值。每个进程的文件数上限是在内核编译时固定的；请参阅 `/usr/src/linux/Documentation/proc.txt` 获取更多信息。

PostgreSQL 服务器为每个连接使用一个进程，因此你至少应提供与允许连接数相同数量的进程，再加上系统其他部分所需的进程数。通常这不是问题，但如果你在一台机器上运行多个服务器，资源可能就会变得紧张。

打开文件数的出厂默认限制通常被设为“对共享环境友好”的值，也就是允许许多用户在一台机器上共存，而不会占用不成比例的系统资源。如果你在一台机器上运行很多服务器，这也许正合适；但在专用服务器上，你可能会希望提高这个限制。

另一方面，有些系统允许单个进程打开非常多的文件；如果不止少数几个进程都这么做，就很容易超过系统范围的限制。如果你遇到这种情况，又不想修改系统范围的限制，可以设置 PostgreSQL 的 max_files_per_process 配置参数来限制其打开文件的消耗。

另一个在支持大量客户端连接时可能需要关注的内核限制，是套接字连接队列的最大长度。如果在很短时间内到达的连接请求超过了这个数量，那么其中一些请求可能会在 postmaster 来得及处理之前就被拒绝，于是客户端会收到不太有帮助的连接失败错误，例如“Resource temporarily unavailable”或“Connection refused”。在许多平台上，默认队列长度限制是 128。要提高它，请通过sysctl 调整相应的内核参数，然后重新启动 postmaster。该参数在不同系统上的名称不同：在 Linux 上是`net.core.somaxconn`，在较新的 FreeBSD 上是`kern.ipc.soacceptqueue`，在 macOS 和其他 BSD 变体上是`kern.ipc.somaxconn`。

18.4.4. Linux 内存过量分配

在 Linux 2.4 及更高版本中，默认的虚拟内存行为对 PostgreSQL 并非最佳。由于内核实现内存过量分配的方式，如果 PostgreSQL 或其他进程的内存需求导致系统耗尽虚拟内存，内核可能会终止 PostgreSQL 的 postmaster（主管服务器进程）。

如果发生这种情况，你会看到类似下面的内核消息（至于应到哪里查看此类消息，请参阅你的系统文档和配置）：

```
Out of Memory: Killed process 12345 (postgres).
```

这表示 `postgres` 进程因内存压力而被终止。尽管现有数据库连接仍会继续正常运行，但新连接将不再被接受。要恢复服务，必须重启PostgreSQL。

避免该问题的一种办法，是让PostgreSQL 运行在一台你能确定不会被其他进程耗尽内存的机器上。如果内存紧张，增加操作系统交换空间也有助于避免这个问题，因为内存不足（OOM）杀手只有在物理内存和交换空间都耗尽时才会被触发。

如果导致系统耗尽内存的正是 PostgreSQL 自身，那么你可以通过调整配置来避免该问题。在某些情况下，调低与内存相关的配置参数会有帮助，尤其是 `shared_buffers` 和 `work_mem`。在其他情况下，允许数据库服务器接受过多连接也可能导致这一问题。很多时候，更好的做法可能是减小 `max_connections`，并转而使用外部连接池软件。

在 Linux 2.6 及更高版本中，可以修改内核的行为，使其不会“过量分配”内存。虽然此设置无法完全阻止 OOM 杀手[1]被调用，但会显著降低发生概率，从而使系统行为更健壮。方法是选择严格的过量分配模式，使用以下命令：`sysctl`：

```
sysctl -w vm.overcommit_memory=2
```

或者在以下文件中加入等效条目：`/etc/sysctl.conf`。你可能还希望修改相关设置 `vm.overcommit_ratio`。详细信息参见内核文档文件 https://www.kernel.org/doc/Documentation/vm/overcommit-accounting。

另一种方法可在修改或不修改 `vm.overcommit_memory` 的情况下使用：把 postmaster 进程专属的 OOM 评分调整值设为 `-1000`，从而保证它不会成为 OOM 杀手的目标。最简单的做法是在 postmaster 启动脚本中、调用 postmaster 之前执行：

```
echo -1000 > /proc/self/oom_score_adj
```

请注意，这个操作必须以 root 身份完成，否则不会生效；因此，由 root 拥有的启动脚本是最容易执行该操作的位置。如果这样做，还应在调用 postmaster 之前，在启动脚本中设置以下环境变量：

```
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
export PG_OOM_ADJUST_VALUE=0
```

这些设置会使 postmaster 子进程以常规的 OOM 评分调整值零运行，以便 OOM 杀手在需要时仍可将它们作为目标。如果希望子进程以其他 OOM 评分调整值运行，也可以为 `PG_OOM_ADJUST_VALUE` 指定其他值。（也可以省略 `PG_OOM_ADJUST_VALUE`，此时默认为零。）如果不设置 `PG_OOM_ADJUST_FILE`，子进程就会和 postmaster 使用相同的 OOM 评分调整值，这并不明智，因为这样做的目的正是确保 postmaster 获得优先保护。

较旧的 Linux 内核不提供 `/proc/self/oom_score_adj`，但可能提供名为 `/proc/self/oom_adj` 的同类功能的早期版本。其工作方式相同，只是禁用值为 `-17` 而非 `-1000`。

> **注意**
>
> 据报告，某些厂商的 Linux 2.4 内核包含 2.6 版过量分配 `sysctl` 参数的早期实现。但是，在没有相关代码的 2.4 内核上将 `vm.overcommit_memory` 设为 2，会使情况更糟，而非更好。建议在 2.4 安装上尝试之前，检查实际内核源码（参见文件 `mm/mmap.c` 中的函数 `vm_enough_memory`），确认内核支持的功能。存在 `overcommit-accounting` 文档文件，不能作为该功能已存在的证据。如有任何疑问，请咨询内核专家或内核供应商。

[1] https://lwn.net/Articles/104179/

18.4.5. Linux 大页

使用大块连续内存时，大页可以降低开销；PostgreSQL 就采用这种内存使用方式，尤其是将shared_buffers 设为较大值时。要让 PostgreSQL 使用此功能，内核需要配置为 `CONFIG_HUGETLBFS=y` 和 `CONFIG_HUGETLB_PAGE=y`。还必须调整内核设置 `vm.nr_hugepages`。要估算所需的大页数量，请启动 PostgreSQL 时先禁用大页，然后检查 postmaster 的匿名共享内存段大小以及系统的大页大小，使用的接口是 `/proc` 文件系统。操作可能如下所示：

```
$ head -1 $PGDATA/postmaster.pid
4170
$ pmap 4170 | awk '/rw-s/ && /zero/ {print $2}'
6490428K
$ grep ^Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
```

`6490428` / `2048` 约等于 `3169.154`，因此本例至少需要 `3170` 个大页，可以这样设置：

```
$ sysctl -w vm.nr_hugepages=3170
```

如果机器上的其他程序也需要大页，应使用更大的设置值。不要忘记将此设置添加到 `/etc/sysctl.conf`，使其在重启后重新生效。

有时内核无法立即分配所需数量的大页，因此可能需要重复执行该命令或重启。（刚重启后，机器的大部分内存应该都可以转换为大页。）要确认大页分配情况，请使用：

```
$ grep Huge /proc/meminfo
```

可能还需要通过 sysctl 设置 `vm.hugetlb_shm_group`，授予数据库服务器的操作系统用户使用大页的权限，以及/或者通过 `ulimit -l` 授予其锁定内存的权限。

PostgreSQL 对大页的默认行为是：只要可能就使用它们；如果失败，则回退到普通页。要强制使用大页，可以在 `postgresql.conf` 中将 huge_pages 设为 `on`。请注意，在这种设置下，如果没有足够的大页可用，PostgreSQL 将无法启动。

关于 Linux 大页特性的详细说明，请参见 https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt。

18.5. 关闭服务器

有几种方法可以关闭数据库服务器。通过向主管进程发送不同的信号，可以控制关闭类型，该进程为 `postgres`。

SIGTERM

> 这是智能关闭模式。收到 SIGTERM 后，服务器禁止新连接，但允许现有会话正常结束工作。仅在所有会话终止后才会关闭。如果服务器处于在线备份模式，还会等待该模式结束。在线备份模式生效期间，仍允许新的超级用户连接（这一例外允许超级用户连接以结束在线备份模式），但不允许其他新连接。如果服务器在请求智能关闭时处于恢复状态，则只有在所有常规会话终止后，恢复和流复制才会停止。

SIGINT

这是快速关闭模式。服务器禁止新连接并向所有现有服务器进程发送 SIGTERM，这将导致它们中止当前事务并迅速退出。然后等待所有服务器进程退出，最后关闭。如果服务器处于在线备份模式，该模式将被终止，导致备份不可用。

SIGQUIT

这是立即关闭模式。服务器将向所有子进程发送 SIGQUIT 并等待它们终止。如果有任何进程在 5 秒内未终止，它们将被发送 SIGKILL。一旦所有子进程退出，主管服务器进程将立即退出，而不进行正常的数据库关闭处理。这会导致下次启动时通过重放 WAL 日志执行恢复。仅建议在紧急情况下使用。

pg_ctl 程序提供了一个便于发送这些信号以关闭服务器的接口。另外，在非 Windows 系统上，你也可以用`kill` 直接发送这些信号。可以用 `ps` 程序，或者从数据目录中的`postmaster.pid` 文件找到 `postgres` 进程的 PID。例如，要执行一次快速关闭：

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

重要

最好不要使用 SIGKILL 关闭服务器。这样做会阻止服务器释放共享内存和信号量，因此在启动新服务器之前，可能必须手动释放它们。此外，SIGKILL 会终止 `postgres` 进程，使其没有机会将信号转发给子进程，因此还必须手动逐个终止子进程。

要终止单个会话并允许其他会话继续运行，可使用`pg_terminate_backend()`（参阅表 9.78），或者向与该会话相关的子进程发送SIGTERM 信号。

18.6. 升级 PostgreSQL 集簇

本节讨论如何把数据库数据从一个 PostgreSQL 发行版升级到较新的发行版。

当前 PostgreSQL 的版本号由主版本号和次版本号组成。例如，在版本号 10.1 中，10 是主版本号，1 是次版本号，这意味着它是主版本 10 的第一个次版本。对于 10.0 之前的PostgreSQL 版本，版本号由三个数字组成，例如 9.5.3。在这些情况下，主版本由版本号的前两组数字组成，例如 9.5；次版本是第三个数字，例如 3，这意味着它是主版本 9.5 的第三个次版本。

次版本从不改变内部存储格式，并且始终与同一主版本号中的更早和更晚次版本兼容。例如，10.1 与 10.0 和 10.6 兼容；同样地，9.5.3 与 9.5.0、9.5.1 和 9.5.6 兼容。要在兼容版本之间更新，只需在服务器关闭时替换可执行文件，然后重启服务器即可。数据目录保持不变，次版本升级就是这么简单。

对于 PostgreSQL 的主版本，内部数据存储格式可能会发生变化，因此升级会复杂得多。将数据迁移到新的主版本的传统方法是转储并恢复数据库，不过这可能比较慢。更快的方法是 pg_upgrade。此外，也可以使用复制方法，如下所述。

新的主版本通常还会引入一些用户可见的不兼容变化，因此应用程序可能也需要做出修改。所有用户可见的变化都列在发行说明（附录 E）中；请特别留意标为“迁移（Migration）”的小节。虽然你可以从一个主版本直接升级到另一个主版本，而不必逐个升级中间版本，但仍应阅读所有中间版本的主版本发行说明。

谨慎的用户会希望在完全切换之前，先在新版本上测试客户端应用。因此，同时安装旧版本和新版本通常是个好主意。测试PostgreSQL 主版本升级时，可以考虑以下几类可能的变化：

管理

管理员可用于监控和控制服务器的功能，在每个主版本中常常都会变化并有所增强。

SQL

通常这意味着 SQL 命令能力会增加，而行为不会变化，除非发行说明中特别提到。

库 API

通常像 libpq 这样的库只会增加新功能，除非发行说明中特别说明。

系统目录

系统目录的变化通常只影响数据库管理工具。

服务器端 C 语言 API

这涉及使用 C 语言编写的后端函数 API 的变更。这类变更会影响那些深入引用服务器内部后端函数的代码。

18.6.1. 通过 pg_dumpall 升级数据

一种升级方法是从某个主版本的 PostgreSQL 转储数据，再在另一个版本中恢复。要这样做，必须使用逻辑备份工具，例如pg_dumpall；文件系统级备份方法不起作用。（系统中有相应检查，会阻止你在不兼容版本的PostgreSQL 上使用某个数据目录，因此即便误用错误版本的服务器去启动某个数据目录，通常也不会造成太大损害。）

建议使用来自较新版本 PostgreSQL 的 pg_dump 和 pg_dumpall 程序，以利用这些程序可能包含的改进。当前版本的转储程序可以读取从 8.0 起任意服务器版本的数据。

以下说明假定你现有的安装位于`/usr/local/pgsql` 目录下，数据区域位于`/usr/local/pgsql/data`。请根据实际情况替换成你的路径。

1. 如果是在制作备份，请确认数据库此时没有正在进行更新。更新操作不会影响备份的完整性，但那些变更当然不会包含在备份中。必要时，可修改`/usr/local/pgsql/data/pg_hba.conf`（或等效文件）中的访问权限，禁止除你之外的其他人访问数据库。有关访问控制的更多信息见第 20 章。

   要备份整个数据库安装，请输入：

   ```
   pg_dumpall > outputfile
   ```

   制作备份时，你可以使用当前正在运行版本中的pg_dumpall 命令，详见第 25.1.2 节。不过，为了获得最佳结果，尽量使用PostgreSQL 11.22 自带的 pg_dumpall 命令，因为这一版本相较旧版本包含了缺陷修复和改进。这个建议看起来可能有些反常，因为你这时还没有安装新版本；但如果你计划让新旧版本并行安装，遵循这一建议是明智的。在那种情况下，你可以先正常完成新版本安装，稍后再迁移数据，这样也能减少停机时间。

2. 关闭旧服务器：

   ```
   pg_ctl stop
   ```

   在那些会在开机时自动启动 PostgreSQL 的系统上，可能已经有某个启动脚本能完成同样的工作。例如，在Red Hat Linux 系统上，也许下面这条命令就可以：

   ```
   /etc/rc.d/init.d/postgresql stop
   ```

   有关服务器启动和停止的细节见第 18 章。

3. 如果是从备份恢复，请重命名或删除旧的安装目录，前提是它不是按版本区分的目录。与其删除，不如重命名，这样如果你遇到问题需要回退，它仍然还在。请记住，该目录可能会占用相当可观的磁盘空间。要重命名该目录，可以使用类似下面的命令：

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

   （请确保把该目录作为一个整体移动，这样相对路径才会保持不变。）

4. 按照第 16.4 节安装新版本的 PostgreSQL。

5. 如有需要，创建一个新的数据库集簇。请记住，执行这些命令时必须登录到专用数据库用户账户（如果你正在升级，就已经拥有这个账户）。

   ```
   /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
   ```

6. 恢复先前的 `pg_hba.conf`，以及你对`postgresql.conf` 所做的任何修改。

7. 启动数据库服务器，同样要使用该专门的数据库用户账户：

   ```
   /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
   ```

8. 最后，使用以下命令从备份恢复数据：

   ```
   /usr/local/pgsql/bin/psql -d postgres -f outputfile
   ```

   这里应使用新版 psql。

若要把停机时间降到最低，可以把新服务器安装到不同目录中，并让新旧两个服务器在不同端口上并行运行。这样就可以使用类似下面的命令：

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

来传输数据。

18.6.2. 通过 pg_upgrade 升级数据

pg_upgrade 模块允许把一次安装从一个 PostgreSQL 主版本 “就地” 迁移到另一个主版本。升级可在几分钟内完成，特别是在使用`--link` 模式时更是如此。它需要执行与上面 pg_dumpall

方法类似的步骤，例如启动/停止服务器、运行initdb 等。pg_upgrade 的文档概述了所需步骤。

18.6.3. 通过复制升级数据

也可以使用逻辑复制方法，创建一个运行较新版本PostgreSQL 的备库。之所以可行，是因为逻辑复制支持在不同主版本的PostgreSQL 之间进行复制。该备库可以与旧服务器部署在同一台机器上，也可以部署在不同机器上。一旦它与主库（运行旧版本PostgreSQL）同步，就可以切换主备角色，让备库提升为主库，并关闭旧的数据库实例。这种切换使升级只需几秒钟的停机时间。

这种升级方法既可以使用内置的逻辑复制功能，也可以使用外部逻辑复制系统，例如pglogical、Slony、Londiste 和 Bucardo。

18.7. 防止服务器欺骗

当服务器正在运行时，恶意用户不可能取代正常的数据库服务器。然而，当服务器关闭时，本地用户却可能通过启动自己的服务器来冒充正常服务器。这个伪造服务器可以读取客户端发出的密码和查询语句，但由于PGDATA 目录仍然受目录权限保护，它无法返回任何真实数据。之所以会出现这种冒充，是因为任何用户都可以启动一个数据库服务器；而客户端除非经过专门配置，否则无法识别服务器是否为伪造。

防止 local 连接被欺骗的一种办法，是使用一个 Unix 域套接字目录（unix_socket_directories），并且只允许受信任的本地用户对该目录拥有写权限。这样可以防止恶意用户在其中创建自己的套接字文件。如果你担心某些应用程序仍会引用/tmp 中的套接字文件，因此仍然容易受到欺骗，那么可以在操作系统启动时创建一个符号链接/tmp/.s.PGSQL.5432，使其指向迁移后的套接字文件。你可能还需要修改/tmp 清理脚本，防止该符号链接被删除。

对于 local 连接，另一个选项是让客户端使用requirepeer，指定连接到该套接字的服务器进程必须由哪个用户拥有。

要防止 TCP 连接上的服务器欺骗，最佳办法是使用 SSL 证书，并确保客户端检查服务器证书。为此，服务器必须配置为仅接受 hostssl 连接（第 20.1 节），并且具备 SSL 密钥和证书文件（第 18.9 节）。TCP 客户端必须使用 sslmode=verify-ca 或 verify-full 连接，并安装适当的根证书文件（第 34.18.1 节）。

18.8. 加密选项

PostgreSQL 提供了多个层次的加密，并在防止数据因数据库服务器被盗、不诚实的管理员或不安全网络等原因而泄露方面提供了很高的灵活性。为了保护医疗记录、金融交易等敏感数据，也可能需要使用加密。

密码加密

数据库用户密码以 hash 形式存储（由 password_encryption 控制），因此管理员无法得知分配给用户的实际密码。如果客户端认证使用 SCRAM 或 MD5，明文密码甚至不会暂时出现在服务器端，因为客户端会在通过网络发送前完成加密。推荐使用 SCRAM，因为它是互联网标准，并且比 PostgreSQL 特有的 MD5 认证协议更安全。

指定列加密

pgcrypto 模块允许以加密形式存储特定字段。这在只有部分数据敏感时很有用。客户端提供解密密钥，数据在服务器端解密后再发送给客户端。

当数据被解密并在服务器与客户端之间传输时，解密后的数据和解密密钥会在服务器端短暂存在。这就给那些可以完全访问数据库服务器的人（例如系统管理员）提供了一个短暂的机会来截获密钥和数据。

数据分区加密

存储加密可以在文件系统层或块设备层实现。Linux 的文件系统加密方案包括 eCryptfs 和 EncFS，而 FreeBSD 使用 PEFS。块级或整盘加密方案则包括 Linux 上的 dm-crypt + LUKS，以及 FreeBSD 上的 GEOM 模块 geli 和 gbde。许多其他操作系统也支持这类功能，包括 Windows。

这种机制可以防止在整台计算机或磁盘被盗时，从驱动器中直接读取未加密数据。但它无法防御文件系统已挂载时的攻击，因为在挂载之后，操作系统会提供数据的解密视图。不过，要挂载该文件系统，就必须以某种方式把加密密钥提供给操作系统，而有时这个密钥就存放在挂载该磁盘的主机上的某处。

跨网络加密数据

SSL 连接会加密通过网络发送的全部数据：密码、查询和返回的数据。`pg_hba.conf` 文件允许管理员指定哪些主机可以使用非加密连接（`host`），哪些主机必须使用 SSL 加密连接（`hostssl`）。此外，客户端可以指定仅通过 SSL 连接到服务器。也可以使用 Stunnel 或 SSH 加密传输。

SSL 主机认证

客户端和服务器都可以向对方提供 SSL 证书。这需要双方做一些额外配置，但它提供的认证强度高于单纯依赖密码。它可以防止某台计算机短暂冒充服务器，只为读取客户端发送的密码；同时也有助于防御“中间人”攻击，即某台位于客户端与服务器之间的计算机冒充服务器，读取并转发双方之间的全部数据。

客户端加密

如果服务器所在机器的系统管理员不可信，那么就有必要由客户端自行加密数据。这样一来，未加密的数据就永远不会出现在数据库服务器上。数据在发送给服务器之前就在客户端加密，而查询结果也必须在客户端解密后才能使用。

18.9. 使用 SSL 的安全 TCP/IP 连接

PostgreSQL 原生支持使用 SSL 连接来加密客户端与服务器之间的通信，以提高安全性。这要求客户端和服务器系统上都安装了OpenSSL，并且在构建PostgreSQL 时启用了该支持（见第 16 章）。

18.9.1. 基本设置

如果构建时启用了 SSL 支持，那么在 `postgresql.conf` 中将参数 ssl 设为 `on`，就可以在启动 PostgreSQL 服务器时启用 SSL。服务器会在同一个 TCP 端口上同时监听普通连接和 SSL 连接，并与每个连接进来的客户端协商是否使用 SSL。默认情况下，这由客户端决定；关于如何把服务器配置为要求某些或全部连接必须使用 SSL，请参阅第 20.1 节。

要以 SSL 模式启动服务器，必须存在包含服务器证书和私钥的文件。默认情况下，这些文件应分别命名为`server.crt` 和 `server.key`，并放在服务器的数据目录中；不过也可以通过配置参数ssl_cert_file 和 ssl_key_file 指定其他名称和位置。

在 Unix 系统上，`server.key` 的权限必须禁止组和其他用户访问；可以通过 `chmod 0600 server.key` 实现。或者，该文件也可以由 root 拥有并授予组读权限（即

0640）。这种设置适用于证书和密钥文件由操作系统统一管理的安装方式。运行 PostgreSQL 服务器的用户此时应属于有权访问这些证书和密钥文件的组。

如果数据目录允许组读取访问，那么为了满足上面概述的安全要求，证书文件可能需要放在数据目录之外。通常，启用组访问权限是为了允许非特权用户备份数据库；在这种情况下，备份软件将无法读取证书文件，并且很可能会报错。

如果私钥受口令保护，服务器会提示输入该口令，并在输入之前不会启动。默认情况下，使用口令会禁用无需重启服务器即可更改 SSL 配置的能力，不过参见 ssl_passphrase_command_supports_reload。此外，在 Windows 上完全无法使用带口令保护的私钥。

server.crt 中的第一个证书必须是服务器证书，因为它必须与服务器私钥相匹配。“中间”证书颁发机构的证书也可以追加到该文件中。假设根证书和中间证书是使用 v3_ca 扩展创建的（这会将证书的 CA 基本约束设为 true），这样做就可以避免在客户端上存储中间证书。同时，这也使中间证书更容易单独过期。

无需把根证书加入 server.crt。相反，客户端必须持有服务器证书链对应的根证书。

18.9.2. OpenSSL 配置

PostgreSQL 会读取系统范围的 OpenSSL 配置文件。默认情况下，该文件名为openssl.cnf，位于openssl version -d 报告的目录中。可以通过把环境变量 OPENSSL_CONF 设为所需配置文件的名称，覆盖这一默认值。

OpenSSL 支持种类繁多、强度不一的密码套件和认证算法。虽然可以在OpenSSL 配置文件中指定密码套件列表，但你也可以通过修改postgresql.conf 中的 ssl_ciphers，专门为数据库服务器指定要使用的密码套件。

> **注意**
>
> 使用 NULL-SHA 或 NULL-MD5 密码套件，可以在没有加密开销的情况下完成认证。不过，中间人仍然能够读取并转发客户端与服务器之间的通信。此外，与认证的开销相比，加密开销本身很小。基于这些原因，不建议使用 NULL 密码套件。

18.9.3. 使用客户端证书

要要求客户端提供受信任的证书，请把你信任的根证书颁发机构（CA）的证书放入数据目录中的某个文件，在 postgresql.conf 中将参数 ssl_ca_file 设为该文件名，并在 pg_hba.conf 中相应的 hostssl 行上添加认证选项 clientcert=1。这样，SSL 连接启动时就会向客户端请求证书。（关于如何在客户端设置证书，请参见第 34.18 节。）服务器会验证客户端证书是否由某个受信任的证书颁发机构签名。

如果你希望避免在客户端上存储中间证书，那么与现有根证书构成链的中间证书也可以出现在 ssl_ca_file 文件中（前提是根证书和中间证书是使用 v3_ca 扩展创建的）。如果设置了参数 ssl_crl_file，还会检查证书吊销列表（CRL）条目。

clientcert 认证选项适用于所有认证方法，但只适用于 pg_hba.conf 中指定为 hostssl 的行。未指定 clientcert 或将其设为 0 时，如果配置了 CA 文件，服务器仍会依据该文件验证客户端提供的证书，但不会强制要求客户端提供证书。

如果正在设置客户端证书，可以考虑使用 `cert` 认证方法，让证书在提供连接安全性的同时也控制用户认证。详情参见第 20.12 节。（使用 `cert` 认证方法时，不必显式指定 `clientcert=1`。）

18.9.4. SSL 服务器文件用法

表 18.2 概括了服务器端 SSL 设置相关的文件。（表中显示的是默认文件名；本地实际配置的名称可能不同。）

表 18.2. SSL 服务器文件用法

文件	内容	效果
ssl_cert_file (`$PGDATA/server.crt`)	服务器证书	发送给客户端，用于表明服务器身份
ssl_key_file (`$PGDATA/server.key`)	服务器私钥	证明服务器证书是其所有者发送的，并不说明证书所有者是值得信任的
ssl_ca_file	受信任的证书颁发机构	检查客户端证书是由一个受信任的证书颁发机构签名的
ssl_crl_file	被证书颁发机构吊销的证书	客户端证书不能出现在这个列表上

服务器会在启动时以及每次重新加载配置时读取这些文件。在 Windows 系统上，每当为新的客户端连接生成新的后端进程时，也会重新读取它们。

如果在服务器启动时检测到这些文件有错误，服务器将拒绝启动。但如果在重新加载配置时检测到错误，这些文件会被忽略，旧的 SSL 配置将继续使用。在Windows 系统上，如果在后端启动时检测到这些文件有错误，该后端将无法建立 SSL 连接。在所有这些情况下，错误都会记录到服务器日志中。

18.9.5. 创建证书

要为服务器创建一个有效期为 365 天的简单自签名证书，可以使用下面的OpenSSL 命令，并将 *`dbhost.yourdomain.com`* 替换为服务器主机名：

```
openssl req -new -x509 -days 365 -nodes -text -out server.crt \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

然后执行：

```
chmod og-rwx server.key
```

如果该文件权限比这更宽松，服务器会拒绝使用它。关于如何创建服务器私钥和证书的更多细节，请参阅OpenSSL 文档。

虽然自签名证书可用于测试，但在生产环境中应使用由证书颁发机构（CA，通常是企业范围内的根 CA）签名的证书。

要创建一个可由客户端验证其身份的服务器证书，首先要创建证书签名请求（CSR）以及公钥/私钥文件：

```
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
```

然后，使用该密钥对请求进行签名，以创建一个根证书颁发机构（这里使用的是Linux 上 OpenSSL 配置文件的默认位置）：

```
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

最后，创建一个由新根证书颁发机构签名的服务器证书：

```
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key

openssl x509 -req -in server.csr -text -days 365 \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out server.crt
```

`server.crt` 和 `server.key` 应存放在服务器上，而`root.crt` 应存放在客户端上，以便客户端能够验证服务器的叶子证书是否由其受信任的根证书签名。`root.key` 应离线保存，以供将来签发证书时使用。

也可以创建包含中间证书的信任链：

```
# root
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt

# intermediate
openssl req -new -nodes -text -out intermediate.csr \
  -keyout intermediate.key -subj "/CN=intermediate.yourdomain.com"
chmod og-rwx intermediate.key
openssl x509 -req -in intermediate.csr -text -days 1825 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out intermediate.crt

# leaf
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
openssl x509 -req -in server.csr -text -days 365 \
  -CA intermediate.crt -CAkey intermediate.key -CAcreateserial \
  -out server.crt
```

`server.crt` 和 `intermediate.crt` 应拼接为一个证书文件包并存放在服务器上。`server.key` 也应存放在服务器上。`root.crt` 应存放在客户端上，以便客户端验证服务器的叶子证书是否由一条可追溯到其受信任根证书的证书链签名。`root.key` 和 `intermediate.key` 应离线保存，以供将来签发证书时使用。

18.10. 通过 SSH 隧道建立安全 TCP/IP 连接

可以使用 SSH 来加密客户端与 PostgreSQL 服务器之间的网络连接。如果配置得当，即使客户端本身不支持 SSL，这种方法也能提供足够安全的网络连接。

首先确认在 PostgreSQL 服务器所在的同一台机器上有一个正常运行的 SSH 服务器，并且你可以使用`ssh` 以某个用户身份登录；然后就可以建立通向远程服务器的安全隧道。安全隧道会监听本地端口，并把所有流量转发到远程机器上的某个端口。发送到该远程端口的流量可以到达其`localhost` 地址，或者在需要时到达其他绑定地址；对远端来说，这些流量看起来并不是来自你的本地机器。下面这个命令会创建一条从客户端机器到远程机器`foo.com` 的安全隧道：

```
ssh -L 63333:localhost:5432 joe@foo.com
```

`-L` 参数中的第一个数字 63333 是隧道在本地的端口号；它可以是任何未使用的端口。（IANA 保留 49152 到 65535 端口供私用。）其后的名称或 IP 地址是你要连接的远程绑定地址，这里是默认值`localhost`。第二个数字 5432 是隧道远端的端口，例如数据库服务器所使用的端口。要通过这条隧道连接数据库服务器，只需连接本地机器上的 63333 端口：

```
psql -h localhost -p 63333 postgres
```

对数据库服务器来说，它看到的是来自主机`foo.com` 上用户 `joe` 对 `localhost` 绑定地址的连接，并将应用为该用户到该绑定地址所配置的认证方式。请注意，服务器不会认为这是 SSL 加密连接，因为实际上SSH 服务器与 PostgreSQL 服务器之间并没有加密。不过这通常不会带来额外安全风险，因为两者位于同一台机器上。

要让这条隧道建立成功，你必须有权通过 `ssh` 以 `joe@foo.com` 身份连接，就像尝试使用 `ssh` 创建终端会话一样。

你也可以把端口转发设成这样：

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

但这样一来，数据库服务器会把该连接看作是到达其`foo.com` 绑定地址的连接，而默认设置 `listen_addresses = 'localhost'` 并不会监听该地址。这通常不是你想要的结果。

如果你必须经由某个登录主机“跳转”到数据库服务器，那么一种可能的设置如下：

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

请注意，这种从`shell.foo.com` 到 `db.foo.com` 的连接不会受到 SSH 隧道的加密保护。当网络受到各种限制时，SSH 还提供了许多其他配置方式。详情请参阅 SSH 文档。

> **提示**
>
> 还有一些其他应用也能提供安全隧道，其思路与刚才描述的 SSH 方法类似。

18.11. 在 Windows 上注册事件日志

要向操作系统注册一个 Windows 事件日志库，请执行以下命令：

```
regsvr32 pgsql_library_directory/pgevent.dll
```

这会创建被事件查看器使用的注册表项，默认事件源命名为`PostgreSQL`。

若要指定不同的事件源名称（见 event_source），请使用/n 和/i 选项：

regsvr32 /n /i:*event_source_name pgsql_library_directory*/pgevent.dll

要从操作系统中注销该事件日志库，请执行以下命令：

regsvr32 /u [/i:*event_source_name*] *pgsql_library_directory*/pgevent.dll

> **注意**
>
> 要在数据库服务器中启用事件日志，请在postgresql.conf 中修改 log_destination，使其包含eventlog。

第 19 章 服务器配置

有许多配置参数会影响数据库系统的行为。本章第一节将介绍如何与配置参数交互。后续各节将详细讨论每个参数。

19.1. 设置参数

19.1.1. 参数名称和值

所有参数名都是大小写不敏感的。每个参数都可以接受五种类型之一的值：布尔、字符串、整数、浮点数或枚举。该类型决定了设置该参数的语法：

- 布尔：值可以被写成 `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0`（都是大小写不敏感的）或者这些值的任何无歧义前缀。
- 字符串：通常值被包括在单引号内，值内部的任何单引号都需要被双写。不过，如果值是一个简单数字或者标识符，引号通常可以被省略。
- 数字（整数和浮点数）：只有浮点数参数才允许使用小数点。不要使用千位分隔符。不要求使用引号。
- 带单位的数字：一些数字参数具有隐含单位，因为它们描述的是内存或时间量。单位可能是字节、千字节、块（通常为 8 千字节）、毫秒、秒或分钟。这类设置若给出不带单位的数字值，就会使用该设置的默认单位，可以通过 `pg_settings.unit` 了解该默认单位。为了方便，也可以显式指定单位，例如把时间值写成 `'120 ms'`，系统会将其转换为该参数的实际单位。注意，要使用这一特性，值必须写成字符串（带引号）。单位名称区分大小写，并且数字值与单位之间可以有空白。
  - 可用的内存单位是 `B`（字节）、`kB`（千字节）、`MB`（兆字节）、`GB`（吉字节）和 `TB`（太字节）。内存单位的乘数是 1024，而不是 1000。
  - 可用的时间单位是 `ms`（毫秒）、`s`（秒）、`min`（分钟）、`h`（小时）和 `d`（天）。
- 枚举：枚举类型的参数以与字符串参数相同的方式指定，但被限制到一组有限的值。这样一个参数可用的值可以在`pg_settings.enumvals` 中找到。枚举参数值是大小写无关的。

19.1.2. 通过配置文件影响参数

设置这些参数最基本的方法是编辑文件 `postgresql.conf`，它通常位于数据目录中。在数据库集簇目录初始化时，会安装该文件的一个默认副本。其内容示例如下：

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

每行指定一个参数。名称和值之间的等号是可选的。空白不重要（引号括起的参数值内部除外），空行会被忽略。井号（`#`）表示该行余下部分是注释。不是简单标识符或数字的参数值必须用单引号括起。要在参数值中嵌入单引号，可以写两个单引号（推荐）或使用反斜线转义单引号。如果文件包含相同参数的多个条目，则忽略除最后一个之外的所有条目。

以这种方式设定的参数为集簇提供了默认值。除非这些设置被覆盖，活动会话看到的就是这些设置。下面的小节描述了管理员或用户覆盖这些默认值的方法。

主服务器进程每次收到 SIGHUP 信号（最简单的方法是从命令行运行 `pg_ctl reload` 或调用 SQL 函数 `pg_reload_conf()` 来发送这个信号）后都会重新读取这个配置文件。主服务器进程还会把这个信号传播给所有正在运行的服务器进程，这样现有的会话也能采用新值（要等待它们完成当前正在执行的客户端命令之后才会发生）。另外，你可以直接向一个单一服务器进程发送该信号。有些参数只能在服务器启动时设置，在配置文件中对这些条目的修改将被忽略，直到下次服务器重启。配置文件中的非法参数设置也会在SIGHUP 处理过程中被忽略（但是会记录日志）。

除了 `postgresql.conf` 之外，PostgreSQL 数据目录还包含文件 `postgresql.auto.conf`，它与 `postgresql.conf` 采用相同的格式，但设计为自动编辑而非手工编辑。这个文件保存了通过ALTER SYSTEM 命令提供的设置。每当读取 `postgresql.conf` 时，也会读取该文件，并以同样的方式使其中设置生效。`postgresql.auto.conf` 中的设置会覆盖 `postgresql.conf` 中的设置。

外部工具也可以修改 `postgresql.auto.conf`。不建议在服务器运行时这样做，因为并发的 `ALTER SYSTEM` 命令可能会覆盖这些更改。这类工具可能只是简单地在文件末尾追加新设置，也可能选择删除重复设置和/或注释（正如 `ALTER SYSTEM` 那样）。

系统视图 `pg_file_settings` 可以有助于对配置文件中的更改进行提前测试，或者在SIGHUP 信号没有达到预期效果时用来诊断问题。

19.1.3. 通过 SQL 影响参数

PostgreSQL 提供了三个 SQL 命令来建立配置默认值。已经提到过的 ALTER SYSTEM 命令提供了一种改变全局默认值的从 SQL 可访问的方法；它在功效上等效于编辑 `postgresql.conf`。此外，还有两个命令可以针对每个数据库或者每个角色设置默认值：

- ALTER DATABASE 命令允许针对各个数据库覆盖全局设置。
- ALTER ROLE 命令允许用针对特定用户设置的值来覆盖全局设置和数据库设置。

只有当开始一个新的数据库会话时，用`ALTER DATABASE` 和 `ALTER ROLE` 设置的值才会被应用。它们会覆盖从配置文件或服务器命令行获得的值，并且作为该会话后续的默认值。注意某些设置在服务器启动后不能被更改，并且因此不能被这些命令（或者下文列举的命令）设置。

一旦一个客户端连接到数据库，PostgreSQL 会提供两个额外的 SQL 命令（以及等效的函数）用以影响会话本地的配置设置：

- SHOW 命令允许查看所有参数的当前值。对应的函数为 `current_setting(setting_name text)`。
- SET 命令允许修改可在会话本地设置的参数的当前值；它对其他会话没有影响。对应的函数为 `set_config(setting_name, new_value, is_local)`。

此外，系统视图`pg_settings` 可以被用来查看和改变会话本地的值：

- 查询这个视图与使用 `SHOW ALL` 相似，但是可以提供更多细节。它也更加灵活，因为可以为它指定过滤条件或者把它与其他关系进行连接。

- 在这个视图上使用 UPDATE 并且指定更新 `setting` 列，其效果等同于发出`SET` 命令。例如，下面的命令

  ```
  SET configuration_parameter TO DEFAULT;
  ```

  等效于：

  ```
  UPDATE pg_settings SET setting = reset_val WHERE name = 'configuration_parameter';
  ```

19.1.4. 通过 Shell 影响参数

除了设置全局默认值或在数据库、角色级别覆盖默认值之外，你还可以通过 shell 工具把设置传递给PostgreSQL。服务器和 libpq 客户端库都能通过 shell 接受参数值。

- 在服务器启动期间，可以通过`-c` 命令行参数把参数设置传递给 `postgres` 命令。例如：

  ```
  postgres -c log_connections=yes -c log_destination='syslog'
  ```

  这种方式提供的设置会覆盖通过 `postgresql.conf` 或者 `ALTER SYSTEM` 提供的设置，因此除了重启服务器之外无法从全局上改变它们。

- 当通过 libpq 启动一个客户端会话时，可以使用`PGOPTIONS` 环境变量指定参数设置。这种方式建立的设置构成了会话生存期间的默认值，但是不会影响其他的会话。由于历史原因，`PGOPTIONS` 的格式和启动 `postgres` 命令时用到的相似，特别是`-c` 标志必须被指定。例如：

  ```
  env PGOPTIONS="-c geqo=off -c statement_timeout=5min" psql
  ```

  通过 shell 或者其他方式，其他客户端和库可能提供它们自己的机制，以便允许用户在不直接使用 SQL 命令的前提下修改会话设置。

19.1.5. 管理配置文件内容

PostgreSQL 提供了一些特性用于把复杂的 `postgresql.conf` 文件分解成子文件。在管理多个具有相关但不完全相同配置的服务器时，这些特性特别有用。

除了单个参数设置之外，`postgresql.conf` 文件还可以包含 include 指令，用来指定另一个要读取和处理的文件，就像把该文件插入到配置文件的这个位置一样。这一特性允许把一个配置文件拆分成多个物理上独立的部分。include 指令的形式如下：

```
include 'filename'
```

如果文件名不是绝对路径，则会被解释为相对于引用它的配置文件所在目录的路径。include 可以嵌套。

还有一个 `include_if_exists` 指令，其行为与 `include` 相同，但在被引用文件不存在或无法读取时有所不同。普通的 `include` 会将其视为错误，而 `include_if_exists` 只会记录一条消息并继续处理引用它的配置文件。

postgresql.conf 文件也可以包含 include_dir 指令，用来指定一个应被包含的配置文件目录。其用法如下：

```
include_dir 'directory'
```

非绝对目录名会被解释为相对于引用它的配置文件所在目录的路径。在指定目录中，只有名称以 .conf 结尾的非目录文件才会被包含。以 . 开头的文件名也会被忽略，以避免在某些平台上误处理隐藏文件。包含目录中的多个文件会按文件名顺序处理（依据 C 区域规则排序，即数字在字母之前，大写字母在小写字母之前）。

包含文件或目录可以用来在逻辑上分隔数据库配置的各个部分，而不是用一个很大的 postgresql.conf 文件。考虑一个有两台数据库服务器的公司，每一个都有不同的内存量。两者很可能会共享部分配置，例如日志设置。但是两者关于内存的参数将会不同。并且还可能会有服务器相关的自定义。一种管理这类情况的方法是将你的站点的自定义配置修改分成三个文件。你可以把下面的内容加入到你的postgresql.conf 文件末尾来包含它们：

```
include 'shared.conf'
include 'memory.conf'
include 'server.conf'
```

所有的系统将会有相同的 shared.conf。每个有特定内存量的服务器可以共享相同的 memory.conf。你可能对所有 8GB 内存的服务器有一个，而对那些 16GB 内存的服务器有另一个。并且最后server.conf 可以装有真正服务器相关的配置信息。

另一种做法是创建一个配置文件目录，并把这些信息放到其中的文件里。例如，一个conf.d 目录可以在 postgresql.conf 的末尾被引用：

```
include_dir 'conf.d'
```

然后你可以这样命名 conf.d 目录中的文件：

```
00shared.conf
01memory.conf
02server.conf
```

这种命名习惯建立了这些文件将被载入的清晰顺序。这是很重要的，因为在服务器读取配置文件时，对于一个特定的参数只有最后碰到的一个设置才会被使用。在这个示例中，conf.d/02server.conf 设置的东西将会覆盖在 conf.d/01memory.conf 中相同参数的值。

你还可以使用这种配置目录方法，在命名文件时更有描述性：

```
00shared.conf
01memory-8GB.conf
02server-foo.conf
```

这种形式的安排为每个配置文件变体给定了一个唯一的名称。当多个服务器把它们的配置全部存储在一个位置（例如在一个版本控制仓库中）时，这可以帮助消除歧义（在版本控制下存储数据库配置文件是另一个值得考虑的好方法）。

19.2. 文件位置

除了已经提到过的 postgresql.conf 文件之外，PostgreSQL 还使用另外两个手工编辑的配置文件，它们控制客户端认证（其使用在第 20 章中讨论）。默认情况下，所有三个配置文件都存放在数据库集簇的数据目录中。本节描述的参数允许配置文件放在别的地方（这么做可以简化管理，特别是如果配置文件被独立放置，可以很容易保证它得到恰当的备份）。

data_directory (string)

指定用于数据存储的目录。这个参数只能在服务器启动时设置。

config_file (string)

指定主服务器配置文件（通常叫 postgresql.conf）。这个参数只能在 postgres 命令行上设置。

hba_file (string)

指定基于主机认证配置文件（通常叫 pg_hba.conf）。这个参数只能在服务器启动的时候设置。

ident_file (string)

指定用于用户名称映射的配置文件（通常叫 pg_ident.conf）。这个参数只能在服务器启动的时候设置。另见第 20.2 节。

external_pid_file (string)

指定服务器应创建的额外进程 ID（PID）文件的名称，供服务器管理程序使用。这个参数只能在服务器启动的时候设置。

在默认安装中不会显式设置以上参数。相反，命令行参数-D 或者环境变量 PGDATA 指定数据目录，并且上述配置文件都能在数据目录中找到。

如果你想把配置文件放在别的地方而不是数据目录中，那么postgres -D 命令行选项或者环境变量 PGDATA 必须指向包含配置文件的目录，并且postgresql.conf 中（或者命令行上）的 data_directory 参数必须设置为数据目录的实际位置。请注意，data_directory 将覆盖 -D 和 PGDATA 指定的数据目录位置，但是不覆盖配置文件的位置。

如果你愿意，可以使用选项config_file、hba_file 和/或 ident_file 单独指定配置文件名称和位置。config_file 只能在 postgres 命令行上指定，但是其他参数可以在主配置文件中设置。如果所有三个参数外加data_directory 被显式地设置，则不必指定-D 或 PGDATA。

在设置任何这些参数时，相对路径将被解释为相对于启动 postgres 时所在目录的路径。

19.3. 连接和认证

19.3.1. 连接设置

listen_addresses (string)

指定服务器用于监听来自客户端应用程序的连接的 TCP/IP 地址。该值采用逗号分隔的主机名和/或数字 IP 地址的形式。特殊条目* 对应于所有可用的 IP 接口。条目 0.0.0.0 允许监听所有 IPv4 地址，::允许监听所有 IPv6 地址。如果列表为空，则服务器不会在任何 IP 接口

上监听，此时只能使用 Unix 域套接字进行连接。如果列表不为空，则服务器将在至少一个 TCP/IP 地址上可以监听时启动。对于任何无法打开的 TCP/IP 地址，将发出警告。默认值为 localhost，仅允许进行本地 TCP/IP “回环” 连接。

在客户端认证（第 20 章）允许对谁可以访问服务器进行细粒度控制的同时，`listen_addresses` 控制哪些接口接受连接尝试，这可以帮助防止在不安全的网络接口上重复恶意连接请求。此参数只能在服务器启动时设置。

`port` (`integer`)

服务器监听的 TCP 端口，默认是 5432。请注意，服务器监听的所有 IP 地址都使用同一个端口号。此参数只能在服务器启动时设置。

`max_connections` (`integer`)

决定数据库服务器允许的最大并发连接数。默认值通常是 100 个连接，但如果内核设置不支持（在 initdb 期间确定），则可能更少。这个参数只能在服务器启动时设置。

当运行一个备库时，你必须设置这个参数等于或大于主库上的参数。否则，备库上将不允许查询。

`superuser_reserved_connections` (`integer`)

决定为 PostgreSQL 超级用户连接保留多少个连接 “槽” 。同时活跃的连接数最多始终只能达到 max_connections。当活跃并发连接数至少达到 `max_connections` 减去 `superuser_reserved_connections` 时，新连接将只接受超级用户，并且不再接受新的复制连接。

默认值是 3 个连接。该值必须小于 `max_connections` 减去 max_wal_senders。这个参数只能在服务器启动时设置。

`unix_socket_directories` (`string`)

指定服务器用于监听来自客户端应用的连接的 Unix 域套接字目录。通过列出用逗号分隔的多个目录可以建立多个套接字。项之间的空白被忽略，如果你需要在名字中包括空白或逗号，在目录名周围放上双引号。一个空值指定在任何 Unix 域套接字上都不监听，在这种情况中只能使用 TCP/IP 套接字来连接到服务器。默认值通常是 `/tmp`，但可以在构建时更改。这个参数只能在服务器启动时设置。

除了套接字文件本身（名为 `.s.PGSQL.`*`nnnn`*，其中*`nnnn`* 是服务器的端口号），一个名为 `.s.PGSQL.`*`nnnn`*`.lock` 的普通文件会在每一个 `unix_socket_directories` 目录中被创建。任何一个都不应该被手工移除。

Windows 没有 Unix 域套接字，因此此参数在 Windows 上没有意义。

`unix_socket_group` (`string`)

设置 Unix 域套接字的所属组（套接字的所属用户总是启动服务器的用户）。可以与选项 `unix_socket_permissions` 一起用于对 Unix 域连接进行访问控制。默认是一个空字符串，表示服务器用户的默认组。这个参数只能在服务器启动时设置。

Windows 没有 Unix 域套接字，因此此参数在 Windows 上没有意义。

`unix_socket_permissions` (`integer`)

设置 Unix 域套接字的访问权限。Unix 域套接字使用通常的 Unix 文件系统权限集。参数值应是以 `chmod` 和 `umask` 系统调用所接受格式指定的数字权限模式。（要使用惯用的八进制格式，数字必须以 `0`（零）开头。）

默认权限是 `0777`，表示任何人都可以连接。合理的其他取值包括 `0770`（仅属主和所属组，另见 `unix_socket_group`）和 `0700`（仅属主）。（注意，对 Unix 域套接字而言，只有写权限起作用，因此设置或撤销读权限和执行权限没有意义。）

此访问控制机制独立于第 20 章中描述的机制。

此参数只能在服务器启动时设置。

此参数对完全忽略套接字权限的系统无效，尤其是 Solaris（截至 Solaris 10）。在这些系统上，可以将 `unix_socket_directories` 指向一个仅向目标用户授予搜索权限的目录，以达到类似效果。Windows 没有 Unix 域套接字，因此此参数在 Windows 上也没有意义。

`bonjour` (`boolean`)

启用通过 Bonjour 通告服务器存在的功能。默认值为关闭。此参数只能在服务器启动时设置。

`bonjour_name` (`string`)

指定 Bonjour 服务名。空字符串 `''`（默认值）表示使用计算机名。如果编译时未启用 Bonjour 支持，则此参数会被忽略。此参数只能在服务器启动时设置。

`tcp_keepalives_idle` (`integer`)

指定在没有活动多少秒之后，TCP 应向客户端发送 keepalive 消息。值 0 表示使用系统默认值。此参数仅在支持`TCP_KEEPIDLE` 或等效套接字选项的系统以及 Windows 上可用；在其他系统上，它必须为零。在通过 Unix 域套接字连接的会话中，此参数会被忽略，并始终读作零。

> **注意**
>
> 在 Windows 上，值 0 会将此参数设置为 2 小时，因为 Windows 不提供读取系统默认值的方法。

`tcp_keepalives_interval` (`integer`)

指定未被客户端确认收到的 TCP keepalive 消息在多少秒后应被重传。值 0 表示使用系统默认值。此参数仅在支持 `TCP_KEEPINTVL` 或等效套接字选项的系统以及 Windows 上可用；在其他系统上，它必须为零。在通过 Unix 域套接字连接的会话中，此参数会被忽略，并始终读作零。

> **注意**
>
> 在 Windows 上，值 0 会将此参数设置为 1 秒，因为 Windows 不提供读取系统默认值的方法。

`tcp_keepalives_count` (`integer`)

指定在服务器与客户端之间的连接被视为中断之前，可以丢失多少个 TCP keepalive 消息。值 0 表示使用系统默认值。这个参数只有在支持 `TCP_KEEPCNT` 或等效套接字选项的系统上才可用；在其他系统上，它必须为零。在通过 Unix 域套接字连接的会话中，这个参数会被忽略，并始终读作零。

> **注意**
>
> Windows 不支持此参数，它必须为零。

19.3.2. 认证

`authentication_timeout` (`integer`)

允许完成客户端认证的最长时间，以秒为单位。如果一个客户端没有在这段时间里完成认证协议，服务器将关闭连接。这样就避免了出问题的客户端无限制地占有一个连接。默认值是 1 分钟（`1m`）。这个参数只能在服务器命令行上或者在 `postgresql.conf` 文件中设置。

`password_encryption` (`enum`)

在 CREATE ROLE 或 ALTER ROLE 中指定密码时，此参数决定用于加密密码的算法。默认值为 `md5`，它将密码存储为 MD5 hash（也接受 `on`，作为 `md5` 的别名）。将此参数设置为 `scram-sha-256` 将使用 SCRAM-SHA-256 加密密码。

注意，较旧的客户端可能不支持 SCRAM 认证机制，因此无法使用以 SCRAM-SHA-256 加密的密码。详情参见第 20.5 节。

`krb_server_keyfile` (`string`)

设置服务器的 Kerberos 密钥文件的位置。详情请参考第 20.6 节。这个参数只能在 `postgresql.conf` 文件中或者服务器命令行上设置。

`krb_caseins_users` (`boolean`)

设置是否应该以大小写不敏感的方式对待 GSSAPI 用户名。默认值是 `off`（大小写敏感）。这个参数只能在 `postgresql.conf` 文件中或者服务器命令行上设置。

`db_user_namespace` (`boolean`)

此参数启用各数据库独立的用户名。默认关闭。此参数只能在 `postgresql.conf` 文件中或在服务器命令行上设置。

如果启用此参数，你应以*`username@dbname`* 的形式创建用户。连接的客户端传入 *`username`* 时，会在用户名后附加`@`和数据库名，然后由服务器查找此数据库专属的用户名。请注意，在 SQL 环境中创建名称包含`@`的用户时，需要用引号括起用户名。

启用此参数后，仍然可以创建普通的全局用户。只需在客户端指定用户名时附加`@`，例如 `joe@`。服务器查找用户名之前会去掉`@`。

`db_user_namespace` 会使客户端和服务器的用户名表示形式不同。认证检查始终使用服务器端的用户名，因此认证方法必须针对服务器端的用户名配置，而不是客户端的用户名。由

于md5 在客户端和服务器上都使用用户名作为盐值，md5 不能与 db_user_namespace 一起使用。

> **注意**
>
> 此特性旨在作为找到完整解决方案之前的临时措施。届时将移除此选项。

19.3.3. SSL

关于 SSL 的设置，参见第 18.9 节。

ssl (boolean)

启用 SSL 连接。这个参数只能在 postgresql.conf 文件中或者服务器命令行上设置。默认值是 off。

ssl_ca_file (string)

指定包含 SSL 服务器证书颁发机构（CA）的文件名。相对路径是相对于数据目录的。这个参数只能在 postgresql.conf 文件中或者服务器命令行上设置。默认值为空，表示没有载入 CA 文件，并且客户端证书验证没有被执行。

ssl_cert_file (string)

指定包含 SSL 服务器证书的文件名。相对路径是相对于数据目录的。这个参数只能在 postgresql.conf 文件中或者服务器命令行上设置。默认值是 server.crt。

ssl_crl_file (string)

指定包含 SSL 客户端证书吊销列表（CRL）的文件名。相对路径是相对于数据目录的。此参数只能在 postgresql.conf 文件或服务器命令行中设置。默认为空，表示不加载 CRL 文件。

ssl_key_file (string)

指定包含 SSL 服务器私钥的文件名。相对路径是相对于数据目录的。这个参数只能在 postgresql.conf 文件中或者服务器命令行上设置。默认值是 server.key。

ssl_ciphers (string)

指定允许 SSL 连接使用的 SSL 密码套件列表。该设置的语法及支持的值列表，可参见 OpenSSL 包中的 ciphers 手册页。此设置只影响使用 TLS 1.2 及更低版本的连接。目前没有控制 TLS 1.3 连接所用密码套件的设置。默认值是 HIGH:MEDIUM:+3DES:!aNULL。除非你有特定的安全需求，否则该默认值通常是合理的选择。

这个参数只能在 postgresql.conf 文件中或者服务器命令行上设置。

默认值的解释：

HIGH

使用 HIGH 组中密码算法（例如 AES、Camellia、3DES）的密码套件。

MEDIUM

使用 MEDIUM 组中密码算法（例如 RC4、SEED）的密码套件。

+3DES

OpenSSL 对 HIGH 的默认排序有问题，因为它将 3DES 排在 AES128 之前。这是不正确的，因为 3DES 的安全性低于 AES128，并且速度也慢得多。+3DES 会把它重新排序到其他所有 HIGH 和 MEDIUM 密码套件之后。

!aNULL

禁用不进行认证的匿名密码套件。这类密码套件容易遭受中间人攻击，因此不应使用。

可用的密码套件细节可能会随着 OpenSSL 版本变化。可使用命令 openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL' 来查看当前安装的 OpenSSL 版本的实际细节。注意这个列表是根据服务器密钥类型在运行时过滤过的。

ssl_prefer_server_ciphers (boolean)

指定是否使用服务器的 SSL 密码套件优先顺序，而非客户端的优先顺序。此参数只能在 postgresql.conf 文件中或服务器命令行上设置。默认值为 true。

旧版 PostgreSQL 没有此设置，始终采用客户端的优先顺序。此设置主要用于与这些旧版本保持向后兼容。采用服务器的优先顺序通常更好，因为服务器更可能得到适当配置。

ssl_ecdh_curve (string)

指定在 ECDH 密钥交换中使用的曲线名称。所有连接的客户端都必须支持该曲线。它不必与服务器椭圆曲线密钥使用的曲线相同。此参数只能在 postgresql.conf 文件中或服务器命令行上设置。默认值为 prime256v1。

OpenSSL 中最常见的曲线名称为：prime256v1（NIST P-256）、secp384r1（NIST P-384）、secp521r1（NIST P-521）。可以用 openssl ecparam -list_curves 命令显示可用曲线的完整列表，但其中并非所有曲线都能用于 TLS。

ssl_dh_params_file (string)

指定包含 Diffie-Hellman 参数的文件名，这些参数用于所谓临时 DH 系列的 SSL 密码套件。默认值为空，此时使用编译时内置的默认 DH 参数。如果攻击者破解了众所周知的内置 DH 参数，使用自定义 DH 参数可以降低由此带来的风险。可以用命令 openssl dhparam -out dhparams.pem 2048 创建自己的 DH 参数文件。

这个参数只能在 postgresql.conf 文件中或者服务器命令行上设置。

ssl_passphrase_command (string)

设置获取用于解密私钥等 SSL 文件的口令时要调用的外部命令。默认值为空，表示使用内置的提示机制。

该命令必须将口令输出到标准输出，并以退出码 0 结束。参数值中的 %p 会被替换为提示字符串。（要表示字面值 %，应写成 %%。）提示字符串可能包含空白，因此务必正确加引号。如果输出末尾有一个换行符，会将其去掉。

该命令不一定要提示用户输入口令，也可以从文件读取口令、从钥匙链获取口令等。用户应自行确保所选机制足够安全。

此参数只能在 postgresql.conf 文件中或服务器命令行上设置。

ssl_passphrase_command_supports_reload (boolean)

此参数决定：配置重载期间，如果密钥文件需要口令，是否也调用 ssl_passphrase_command 设置的口令命令。如果此参数为 false（默认值），则重载时会忽略 ssl_passphrase_command，如果需要口令，就不会重载 SSL 配置。此设置适合需要通过 TTY 提示用户的命令，因为服务器运行期间可能无法使用 TTY。例如，如果从文件中获取口令，则适合将此参数设为 true。

此参数只能在 postgresql.conf 文件中或服务器命令行上设置。

19.4. 资源消耗

19.4.1. 内存

shared_buffers (integer)

设置数据库服务器用于共享内存缓冲区的内存量。默认值通常为 128 兆字节（128MB），但如果内核设置不支持，则可能更小（在 initdb 期间确定）。此设置必须至少为 128 千字节。（BLCKSZ 的非默认值会改变该最小值。）不过，要获得良好性能，通常需要远高于该最小值的设置。此参数只能在服务器启动时设置。

如果专用数据库服务器具有 1GB 或更多内存，shared_buffers 的合理初始值是系统内存的 25%。对于某些工作负载，将 shared_buffers 设得更大也有效，但由于 PostgreSQL 同时依赖操作系统缓存，将超过 40% 的内存分配给 shared_buffers 不太可能比更小的值效果更好。将 shared_buffers 设得更大时，通常还需要相应增加 max_wal_size，以便将大量新数据或已修改数据的写入过程分散到更长的时间内。

对于内存少于 1GB 的系统，适合使用更小的内存比例，以便为操作系统留出足够空间。

huge_pages (enum)

控制是否为主共享内存区域请求巨型页。有效值是 try（默认）、on 以及 off。如果 huge_pages 被设置为 try，则服务器将尝试请求巨型页，但是如果失败会退回到默认的方式。如果为on，请求巨型页失败将使得服务器无法启动。如果为off，则不会请求巨型页。

当前，只有 Linux 和 Windows 上支持这个设置。在其他系统上这个参数被设置为try 时，它会被忽略。

巨型页的使用会导致更小的页表以及花费在内存管理上的 CPU 时间更少，从而提高性能。更多有关 Linux 上使用巨型页的细节请见第 18.4.5 节。

巨型页在 Windows 上被称为大页面。要使用大页面，需要为运行PostgreSQL 的 Windows 用户账号分配在内存中锁定页面的用户权限。可以使用 Windows 的组策略工具（gpedit.msc）来分配用户权限在内存中锁定页面。为了在命令窗口以独立进程（而不是 Windows 服务）的方式启动数据库服务器，命令窗口必须以管理员身份运行或者禁用用户访问控制（UAC）。当 UAC 被启用时，普通的命令窗口会在启动时收回用户权限在内存中锁定页面。

注意这种设置仅影响主共享内存区域。Linux、FreeBSD 以及 Illumos 之类的操作系统也能为普通内存分配自动使用巨型页（也被称为“超级”页或者“大”页面），而不需要来自 PostgreSQL 的显式请求。在 Linux 上，这被称为“transparent huge pages”（THP，透明巨型页）。已知这种特性对某些 Linux 版本上的某些用户会导致PostgreSQL 的性能退化，因此当前并不鼓励使用它（与huge_pages 的显式使用不同）。

temp_buffers (integer)

设置每个数据库会话使用的临时缓冲区的最大数量。这些是会话本地的缓冲区，仅用于访问临时表。默认值为 8 兆字节（8MB）。可以在单个会话内更改此设置，但必须在该会话首次使用临时表之前更改；此后尝试更改该值，对该会话不会产生影响。

会话会按需分配临时缓冲区，上限为 temp_buffers。对于实际不需要很多临时缓冲区的会话，将此参数设得较大时，开销仅为 temp_buffers 每增加一就多分配一个缓冲区描述符，约为 64 字节。不过，如果实际使用了某个缓冲区，还会为它额外消耗 8192 字节（一般而言为 BLCKSZ 字节）。

max_prepared_transactions (integer)

设置可同时处于“预备”状态的事务的最大数量（见 PREPARE TRANSACTION）。将此参数设为零（默认值）会禁用预备事务功能。此参数只能在服务器启动时设置。

如果不打算使用预备事务，应将此参数设为零，以防意外创建预备事务。如果使用预备事务，通常应将 max_prepared_transactions 设为不小于 max_connections 的值，以便每个会话都能有一个待处理的预备事务。

运行备库时，必须将此参数设为与主库相同或更大的值。否则，备库上将不允许执行查询。

work_mem (integer)

指定内部排序操作和 hash 表在写入临时磁盘文件之前可使用的内存量。默认值为 4 兆字节（4MB）。请注意，复杂查询可能同时执行多个排序和 hash 操作，每个操作在开始向临时文件写入数据之前，通常都可以使用此值指定的内存量。此外，多个正在运行的会话也可能并发执行此类操作。因此，使用的总内存量可能是 work_mem 值的数倍；选择此值时必须考虑这一点。排序操作用于 ORDER BY、DISTINCT 和归并连接。Hash 表用于哈希连接、基于 hash 的聚合、以及基于 hash 的 IN 子查询处理。

maintenance_work_mem (integer)

指定维护操作（如 VACUUM、CREATE INDEX 和 ALTER TABLE ADD FOREIGN KEY）可使用的最大内存量。默认值为 64 兆字节（64MB）。由于一个数据库会话一次只能执行一个此类操作，而一个数据库系统通常也不会并发运行很多此类操作，因此可以安全地将该值设得远大于 work_mem。更大的设置可能改善清理和恢复数据库转储的性能。

注意，自动清理运行时，最多可能分配此内存量的autovacuum_max_workers 倍，因此不要将默认值设得过高。单独设置autovacuum_work_mem 可能有助于控制这一点。

注意，在收集死元组标识符时，VACUUM 最多只能使用 1GB 内存。

autovacuum_work_mem (integer)

指定每个自动清理工作进程可使用的最大内存量。默认值为 -1，表示改用 maintenance_work_mem 的值。该设置不影响其他上下文中运行的 VACUUM 的行为。此参数只能在 postgresql.conf 文件中或服务器命令行上设置。

在收集死元组标识符时，自动清理最多只能使用 1GB 内存，因此将 autovacuum_work_mem 设得更高，不会影响自动清理扫描表时能收集的死元组数量。

max_stack_depth (integer)

指定服务器执行栈的最大安全深度。此参数的理想设置是由内核强制执行的实际栈大小限制（如由 ulimit -s 或本地等效设置），减去大约一兆字节的安全余量。需要安全余量是因

为服务器中并非每个例程都检查栈深度，而只在可能递归的关键例程（如表达式求值）中检查。默认设置为两兆字节（2MB），这是保守且不太可能引起崩溃的小值。但是，这可能太小，无法执行复杂函数。只有超级用户才能更改此设置。

把 max_stack_depth 参数设置得高于实际的内核限制将意味着一个失控的递归函数可能会导致一个独立的后端进程崩溃。在 PostgreSQL 能够检测内核限制的平台上，服务器将不允许把这个参数设置为一个不安全的值。不过，并非所有平台都能提供该信息，所以我们还是建议你在选择值时要小心。

dynamic_shared_memory_type (enum)

指定服务器应使用的动态共享内存实现。可选值为 posix（使用 shm_open 分配的 POSIX 共享内存）、sysv（通过 shmget 分配的 System V 共享内存）、windows（Windows 共享内存）、mmap（使用存放在数据目录中的内存映射文件模拟共享内存），以及 none（禁用此功能）。并非所有平台都支持所有值；第一个受支持的选项是该平台的默认值。mmap 不是任何平台的默认选项，通常不建议使用，因为操作系统可能会反复将修改过的页面写回磁盘，增加系统 I/O 负载；不过，在调试、将 pg_dynshmem 目录存放在 RAM 磁盘上，或其他共享内存设施不可用时，它可能有用。

19.4.2. 磁盘

temp_file_limit (integer)

指定一个进程可用于临时文件的最大磁盘空间，例如排序和 hash 临时文件，或保留游标的存储文件。尝试超过此限制的事务将被取消。该值以千字节为单位。-1（默认值）表示没有限制。只有超级用户才能更改此设置。

此设置限制单个 PostgreSQL 进程在任意时刻使用的所有临时文件的总空间。需要注意，显式临时表所用的磁盘空间不计入该上限；计入的是查询执行过程中内部使用的临时文件。

19.4.3. 内核资源使用

max_files_per_process (integer)

设置每个服务器子进程允许同时打开的最大文件数量。默认值为一千个文件。如果内核强制实施了安全的每进程上限，就不必担心此设置。但在某些平台上（尤其是大多数 BSD 系统），内核允许单个进程打开的文件数量很大，如果很多进程都尝试打开这么多文件，就会远超系统实际能够支持的总量。如果遇到“Too many open files”（打开的文件过多）错误，可尝试减小此设置。此参数只能在服务器启动时设置。

19.4.4. 基于代价的清理延迟

执行 VACUUM 和 ANALYZE 命令期间，系统维护一个内部计数器，记录已执行的各种 I/O 操作的估算代价。当累计代价达到上限（由 vacuum_cost_limit 指定）时，执行该操作的进程会休眠一小段时间，时长由 vacuum_cost_delay 指定。随后重置计数器并继续执行。

此功能让管理员能够降低这些命令对并发数据库活动的 I/O 影响。在许多情况下，VACUUM 和 ANALYZE 等维护命令是否快速完成并不重要，但避免它们显著干扰系统执行其他数据库操作的能力通常很重要。基于代价的清理延迟为管理员提供了实现这一点的方法。

对于手动执行的 VACUUM 命令，此功能默认禁用。要启用它，将 vacuum_cost_delay 变量设为非零值。

vacuum_cost_delay (integer)

超过代价上限后，进程将休眠的时长，单位为毫秒。默认值为零，表示禁用基于代价的清理延迟功能。正值会启用基于代价的清理。注意，在许多系统上，休眠延迟的有效分辨率为 10 毫秒；将 vacuum_cost_delay 设为不是 10 的倍数的值，可能与将它设为下一个更大的 10 的倍数效果相同。

使用基于代价的清理时，vacuum_cost_delay 的合适值通常较小，例如 10 或 20 毫秒。调整清理的资源消耗时，最好更改其他清理代价参数。

vacuum_cost_page_hit (integer)

清理在共享缓冲区缓存中找到的缓冲区时所计入的估算代价。它表示锁定缓冲池、查找共享 hash 表和扫描页内容的代价。默认值为 1。

vacuum_cost_page_miss (integer)

清理必须从磁盘读取的缓冲区时所计入的估算代价。它表示锁定缓冲池、查找共享 hash 表、从磁盘读取所需数据块并扫描其内容所需的工作量。默认值为 10。

vacuum_cost_page_dirty (integer)

清理操作修改原本干净的数据块时所计入的估算代价。它表示再次将脏块刷盘所需的额外 I/O。默认值为 20。

vacuum_cost_limit (integer)

会使清理进程休眠的累计代价。默认值为 200。

> **注意**
>
> 某些操作持有关键的锁，因此应尽快完成。这些操作期间不会发生基于代价的清理延迟，所以累计代价可能远超指定上限。为避免此时出现无益的长时间延迟，实际延迟按 vacuum_cost_delay * accumulated_balance / vacuum_cost_limit 计算，但最大不超过 vacuum_cost_delay * 4。

19.4.5. 后台写入器

有一个独立的服务器进程，称为后台写入器，负责写出“脏”（新的或修改过的）共享缓冲区。当干净的共享缓冲区数量似乎不足时，后台写入器会将一些脏缓冲区写入文件系统，并将其标记为干净。这可以降低处理用户查询的服务器进程找不到干净缓冲区、因而不得不自行写出脏缓冲区的可能性。不过，后台写入器确实会使总体 I/O 负载有所增加：反复变脏的页面原本可能在每个检查点间隔中只写出一次，而后台写入器可能在同一间隔内随着它变脏而多次写出。本节参数可用于根据实际需求调整此行为。

bgwriter_delay (integer)

指定后台写入器各轮活动之间的延迟。每一轮中，写入器会对一定数量的脏缓冲区发出写操作（由下面的参数控制），然后休眠 bgwriter_delay 毫秒，再重复此过程。不过，当缓冲池中没有脏缓冲区时，它会进入更长的休眠，而不受 bgwriter_delay 限制。默认值为 200 毫秒（200ms）。注意，在许多系统上，休眠延迟的有效分辨率为 10 毫秒；将 bgwriter_delay 设为不是 10 的倍数的值，可能与将它设为下一个更大的 10 的倍数效果相同。此参数只能在 postgresql.conf 文件中或服务器命令行上设置。

bgwriter_lru_maxpages (integer)

后台写入器每轮写出的缓冲区数量不会超过此值。设为零会禁用后台写入。（由另一个独立的专用辅助进程管理的检查点不受影响。）默认值为 100 个缓冲区。此参数只能在 postgresql.conf 文件中或服务器命令行上设置。

bgwriter_lru_multiplier (floating point)

每轮写出的脏缓冲区数量取决于最近几轮服务器进程所需的新缓冲区数量。将近期平均需求乘以 bgwriter_lru_multiplier，即可估算下一轮所需的缓冲区数量。写入器会写出脏缓冲区，直到可用的干净且可重用缓冲区达到这一数量。（不过，每轮写出的缓冲区数量不会超过 bgwriter_lru_maxpages。）因此，设为 1.0 表示采用“恰好及时”策略，写出的缓冲区数量恰好等于预测需求量。更大的值可为需求突增留出余量，而更小的值则有意将部分写操作留给服务器进程执行。默认值为 2.0。此参数只能在 postgresql.conf 文件中或服务器命令行上设置。

bgwriter_flush_after (integer)

每当后台写入器写出的数据超过 bgwriter_flush_after 字节时，就尝试强制操作系统将这些写操作发往底层存储。这样可以限制内核页缓存中的脏数据量，降低在检查点结束时调用 fsync，或操作系统在后台以较大批次写回数据时发生停顿的可能性。通常，这能大幅降低事务延迟，但在某些情况下性能可能下降，尤其是工作负载大于shared_buffers 而小于操作系统页缓存时。此设置在某些平台上可能没有效果。有效范围为 0（禁用强制写回）至 2MB。Linux 上的默认值为 512kB，其他平台为 0。（如果 BLCKSZ 不是 8kB，默认值和最大值将按比例变化。）此参数只能在 postgresql.conf 文件中或服务器命令行上设置。

较小的 bgwriter_lru_maxpages 和 bgwriter_lru_multiplier 可以降低后台写入器造成的额外 I/O 负载，但也会增加服务器进程必须自行发出写操作的可能性，从而延迟交互式查询。

19.4.6. 异步行为

effective_io_concurrency (integer)

设置 PostgreSQL 预期可以同时执行的并发磁盘 I/O 操作数量。提高此值会增加单个 PostgreSQL 会话尝试并行发起的 I/O 操作数量。允许的范围为 1 至 1000，或设为零以禁用异步 I/O 请求。目前，此设置仅影响位图堆扫描。

对于磁盘，可以将为数据库提供存储的 RAID 0 条带或 RAID 1 镜像中的独立磁盘数量作为合理初始值。（对于 RAID 5，不应计入校验盘。）不过，如果数据库经常忙于执行并发会话发出的多个查询，较小的值可能就足以使磁盘阵列保持繁忙。超过使磁盘保持繁忙所需的值只会增加 CPU 开销。SSD 和其他基于内存的存储通常可以处理大量并发请求，因此最佳值可能达到数百。

异步 I/O 依赖于有效的 posix_fadvise 函数，而某些操作系统缺少此函数。如果该函数不存在，将此参数设为任何非零值都会报错。在某些操作系统（如 Solaris）上，该函数虽然存在，却实际上不做任何事情。

支持此功能的系统上默认值为 1，其他系统为 0。对于位于特定表空间中的表，可以通过设置同名的表空间参数覆盖此值（见 ALTER TABLESPACE）。

max_worker_processes (integer)

设置系统能够支持的后台进程的最大数量。此参数只能在服务器启动时设置。默认值为 8。

运行备库时，必须将此参数设为与主库相同或更大的值。否则，备库上将不允许执行查询。

更改此值时，也应考虑调整 max_parallel_workers、max_parallel_maintenance_workers、max_parallel_workers_per_gather。

max_parallel_workers_per_gather (integer)

设置单个 Gather 或 Gather Merge 节点能够启动的工作进程的最大数量。并行工作进程取自 max_worker_processes 建立的进程池，并受到max_parallel_workers 的限制。注意，运行时实际可用的工作进程可能不足请求的数量。此时，计划会使用少于预期的工作进程运行，效率可能较低。默认值为 2。设为 0 会禁用并行查询执行。

注意，并行查询消耗的资源可能远多于非并行查询，因为每个工作进程都是完全独立的进程，对系统的影响大致相当于额外增加一个用户会话。选择此设置的值，以及配置其他控制资源使用的设置（如 work_mem）时，都应考虑这一点。work_mem 等资源限制分别应用于每个工作进程，因此所有进程的总资源用量可能远高于单个进程通常的用量。例如，使用 4 个工作进程的并行查询，其 CPU 时间、内存、I/O 带宽等用量可能达到完全不使用工作进程的查询的 5 倍。

并行查询的更多信息参见第 15 章。

max_parallel_maintenance_workers (integer)

设置单个工具命令能够启动的并行工作进程的最大数量。目前，唯一支持使用并行工作进程的工具命令是 CREATE INDEX，且仅在构建 B-树索引时支持。并行工作进程取自 max_worker_processes 建立的进程池，并受到max_parallel_workers 的限制。注意，运行时实际可用的工作进程可能不足请求的数量。此时，工具操作会使用少于预期的工作进程运行。默认值为 2。设为 0 会禁止工具命令使用并行工作进程。

注意，并行工具命令的内存消耗不应明显高于等效的非并行操作。这与并行查询的策略不同，后者的资源限制通常分别应用于每个工作进程。并行工具命令将 maintenance_work_mem 视为整个工具命令的资源上限，而不论使用多少个并行工作进程。不过，并行工具命令仍可能消耗多得多的 CPU 资源和 I/O 带宽。

max_parallel_workers (integer)

设置系统能够为并行操作提供的工作进程的最大数量。默认值为 8。增大或减小此值时，也应考虑调整 max_parallel_maintenance_workers、max_parallel_workers_per_gather。此外，将此值设得大于max_worker_processes 没有效果，因为并行工作进程取自该设置建立的工作进程池。

backend_flush_after (integer)

每当单个后端写出的数据超过 backend_flush_after 字节时，就尝试强制操作系统将这些写操作发往底层存储。这样可以限制内核页缓存中的脏数据量，降低在检查点结束时调用 fsync，或操作系统在后台以较大批次写回数据时发生停顿的可能性。通常，这能大幅降低事务延迟，但在某些情况下性能可能下降，尤其是工作负载大于shared_buffers 而小于操作系统页缓存时。此设置在某些平台上可能没有效果。有效范围为 0（禁用强制写回）至 2MB。默认值为 0，即不强制写回。（如果 BLCKSZ 不是 8kB，最大值将按比例变化。）

old_snapshot_threshold (integer)

设置快照在使用时不会发生 snapshot too old 错误的最短可用时间。此参数只能在服务器启动时设置。

超过该阈值后，旧数据可以被清理掉。这有助于在快照长期保持使用时防止膨胀。为了避免因清理本应对该快照可见的数据而得到错误结果，当快照年龄超过该阈值，且该快照被用于读取自其建立以来已被修改过的页面时，就会报错。

值 -1 会禁用此功能，也是默认值。对生产环境而言，有用的取值大概从几个小时到几天不等。此设置会被强制调整为分钟粒度；较小的值（例如 0 或 1min）之所以被允许，只是因为它们有时可用于测试。虽然允许设置到 60d 这么高，但在许多工作负载中，严重膨胀或事务 ID 回卷可能会在更短时间内发生。

启用此功能后，关系末尾释放出来的空间不能返还给操作系统，因为那样可能会移除检测 snapshot too old 条件所需的信息。分配给某个关系的全部空间仍归属于该关系，只能在该关系内重用，除非显式释放（例如使用 VACUUM FULL）。

此设置不保证在任何特定情况下都一定会报错。实际上，如果仍能从某个对象（例如已物化结果集的游标）生成正确结果，那么即使被引用表中的底层行已被清理掉，也不会报错。有些表不能安全地提前清理，因此不受此设置影响，例如系统目录。对于这类表，此设置既不会减少膨胀，也不会在扫描时引入 snapshot too old 错误的可能性。

19.5. 预写式日志

参阅第 30.4 节获取调节这些设置的额外信息。

19.5.1. 设置

wal_level (enum)

wal_level 决定多少信息写入到 WAL 中。默认值是 replica，它会写入足够的数据以支持 WAL 归档和复制，包括在备库上运行只读查询。minimal 会去掉除从崩溃或者立即关机中进行恢复所需的信息之外的所有记录。最后，logical 会增加支持逻辑解码所需的信息。每个层次包括所有更低层次记录的信息。这个参数只能在服务器启动时设置。

在 minimal 级别，可以安全地跳过一些批量操作的 WAL 日志记录，使这些操作快得多（参见第 14.4.7 节）。可以应用此优化的操作包括：

CREATE TABLE AS
CREATE INDEX
CLUSTER
向同一事务中创建或截断的表执行 COPY

但 minimal 级别的 WAL 不包含足够的信息来从基础备份和 WAL 日志重建数据，因此必须使用replica 或更高级别来启用 WAL 归档（archive_mode）和流复制。

在 logical 级别上，记录与replica 相同的信息，以及从 WAL 中提取逻辑变更集所需的信息。使用logical 级别会增加 WAL 的数量，特别是如果许多表被配置为 REPLICA IDENTITY FULL，并且执行了许多UPDATE 和 DELETE 语句。

在 9.6 之前的版本中，这个参数也允许值archive 和 hot_standby。现在仍然接受这些值，但是它们会被映射到replica。

fsync (boolean)

如果打开这个参数，PostgreSQL 服务器将尝试确保更新被物理地写入到磁盘，做法是发出 fsync() 系统调用或者使用多种等价的方法（见 wal_sync_method）。这保证了数据库集簇在一次操作系统或者硬件崩溃后能恢复到一个一致的状态。

虽然关闭 fsync 常常可以得到性能上的收益，但当发生断电或系统崩溃时可能造成不可恢复的数据损坏。因此，只有在能很容易地从外部数据中重建整个数据库时才建议关闭fsync。

可以安全关闭 fsync 的情形包括：从备份文件初始装载一个新数据库集簇；用数据库集簇处理一批数据，处理后就丢弃并重建该数据库；或者使用经常重建且不用于故障切换的只读数据库克隆。仅有高质量硬件不足以成为关闭fsync 的理由。

为确保将 fsync 从关闭改为打开后能够可靠恢复，必须将内核中所有已修改的缓冲区强制写入持久存储。可以在集簇已关闭或fsync 已开启时，通过运行initdb --sync-only、运行sync、卸载文件系统或重启服务器来完成。

在很多情况下，为非关键事务关闭 synchronous_commit，可以获得关闭fsync 所带来的大部分潜在性能收益，同时避免伴随的数据损坏风险。

fsync 只能在 postgresql.conf 文件中或在服务器命令行上设置。如果你关闭这个参数，请也考虑关闭 full_page_writes。

synchronous_commit (enum)

指定数据库服务器向客户端返回“成功”指示之前，必须完成多少 WAL 处理。有效值为 remote_apply、on（默认值）、remote_write、local 和 off。

如果 synchronous_standby_names 为空，只有on 和 off 两种设置有意义；remote_apply、remote_write 和 local 提供的本地同步级别都与 on 相同。所有非 off 模式在本地都会等待 WAL 刷写到磁盘。在 off 模式下则无需等待，因此，向客户端报告成功后，可能还要经过一段时间，才能保证事务不会因服务器崩溃而丢失。（最大延迟为wal_writer_delay 的三倍。）与 fsync 不同，将此参数设为off 不会带来数据库不一致的风险：操作系统或数据库崩溃可能会使一些最近报告已提交的事务丢失，但数据库状态会与这些事务已正常中止时完全相同。因此，当性能比完全确保事务持久性更重要时，关闭 synchronous_commit 可以是一种有用的替代方案。更多讨论见第 30.3 节。

如果 synchronous_standby_names 非空，synchronous_commit 还控制事务提交是否等待备库处理其 WAL 记录。

设为 remote_apply 时，提交会等待当前同步备库回复，确认已收到并应用该事务的提交记录，使其对备库上的查询可见，并且已将其写入备库的持久存储。由于需要等待 WAL 重放，这会比之前的设置产生大得多的提交延迟。设为on 时，提交会等待当前同步备库回复，确认已收到事务的提交记录，并已将其刷写到持久存储。这能保证事务不会丢失，除非主库和所有同步备库的数据库存储都损坏。设为remote_write 时，提交会等待当前同步备库回复，确认已收到事务的提交记录，并已将其写入各自的文件系统。此设置能保证备库上的 PostgreSQL 实例崩溃时数据不丢失，但不能保证备库发生操作系统级别崩溃时数据不丢失，因为数据未必已写入备库的持久存储。设为local 时，提交会等待本地刷盘，但不等待复制。使用同步复制时通常不希望采用这种设置，提供它是为了使选项完整。

此参数可以随时更改；每个事务的行为由提交时生效的设置决定。因此，让一些事务同步提交、另一些事务异步提交是可行且有用的。例如，当默认设置要求同步提交时，可以在一个包含多条语句的事务中执行SET LOCAL synchronous_commit TO OFF，使该事务异步提交。

表 19.1 汇总了 synchronous_commit 各种设置具备的能力。

表 19.1. synchronous_commit 模式

synchronous_commit 设置	本地提交持久性	PG 崩溃后备库提交持久性	OS 崩溃后备库提交持久性	备库查询一致性
remote_apply	•	•	•	•
on	•	•	•	
remote_write	•	•		
local	•			
off				

`wal_sync_method` (`enum`)

用于将 WAL 更新强制写入磁盘的方法。如果 `fsync` 关闭，此设置就没有作用，因为 WAL 文件更新根本不会被强制写入磁盘。可选值为：

- `open_datasync`（用 `open()` 选项 `O_DSYNC` 写 WAL 文件）
- `fdatasync`（在每次提交时调用 `fdatasync()`）
- `fsync`（在每次提交时调用 `fsync()`）
- `fsync_writethrough`（在每次提交时调用 `fsync()`，强制穿透任何磁盘写缓存）
- `open_sync`（用 `open()` 选项 `O_SYNC` 写 WAL 文件）

`open_*` 选项还会使用 `O_DIRECT`（如果可用）。不是在所有平台上都能使用所有这些选择。默认值是列表中第一个被平台支持的那个，不过`fdatasync` 是 Linux 和 FreeBSD 中的默认值。默认值不一定最合适；可能需要更改此设置或系统配置的其他方面，以确保崩溃时的数据安全或达到最佳性能。这些方面在第 30.1 节中讨论。这个参数只能在 `postgresql.conf` 文件中或在服务器命令行上设置。

`full_page_writes` (`boolean`)

启用此参数时，PostgreSQL 服务器会在检查点之后首次修改每个磁盘页面时，将该页面的全部内容写入 WAL。这样做是因为，操作系统崩溃时正在进行的页面写入可能只完成了一部分，导致磁盘页面混有新旧数据。通常存储在 WAL 中的行级变更数据不足以在崩溃恢复时完整还原这样的页面。保存整页镜像能保证正确恢复页面，但会增加必须写入 WAL 的数据量。（由于 WAL 重放总是从检查点开始，只需在检查点之后首次修改每个页面时这样做。因此，减少整页写入开销的一种方法是增大检查点间隔参数。）

关闭此参数可以加快正常操作，但系统故障后可能出现不可恢复的数据损坏或静默数据损坏。风险与关闭`fsync` 类似，虽然较小，但也只有在建议关闭`fsync` 的相同情形下，才应关闭此参数。

关闭这个选项并不影响用于时间点恢复（PITR）的 WAL 归档使用（见第 25.3 节）。

这个参数只能在 `postgresql.conf` 文件中或在服务器命令行上设置。默认值是 `on`。

`wal_log_hints` (`boolean`)

当这个参数为 `on` 时，PostgreSQL 服务器在检查点之后首次修改页面时把该磁盘页面的整个内容都写入 WAL，即使对所谓的提示位做非关键修改也会这样做。

如果启用了数据校验和，提示位更新总是会被 WAL 记录并且这个设置会被忽略。你可以用这个设置来测试：如果数据库启用数据校验和，会产生多少额外的 WAL 记录。

这个参数只能在服务器启动时设置。默认值是 off。

wal_compression (boolean)

当这个参数为 on 时，PostgreSQL 服务器在 full_page_writes 打开时或在基础备份期间，压缩写入 WAL 的整页镜像。在 WAL 重放期间将对压缩的整页镜像进行解压。默认值是 off。只有超级用户能更改这个设置。

开启此参数可以减少 WAL 数据量，而且不会增加不可恢复的数据损坏风险；代价是在记录 WAL 时压缩、重放 WAL 时解压会额外消耗一些 CPU。

wal_buffers (integer)

用于还未写入磁盘的 WAL 数据的共享内存量。默认值 -1 选择等于 shared_buffers 的 1/32 的尺寸（大约 3%），但是不小于64kB 也不大于 WAL 段的尺寸（通常为 16MB）。如果自动的选择太大或太小可以手工设置该值，但是任何小于32kB 的正值都将被当作 32kB。这个参数只能在服务器启动时设置。

在每次事务提交时，WAL 缓冲区的内容被写出到磁盘，因此极大的值不太可能带来显著收益。不过，把这个值设置为至少几个兆字节可以在一个繁忙的服务器（其中很多客户端会在同一时间提交）上提高写性能。由默认设置 -1 选择的自动调节将在大部分情况下得到合理的结果。

wal_writer_delay (integer)

指定 WAL 写入器刷写 WAL 的频繁程度。刷写 WAL 之后，它会休眠wal_writer_delay 毫秒，除非被异步提交的事务唤醒。如果上一次刷写距今少于wal_writer_delay 毫秒，并且此后产生的 WAL 少于wal_writer_flush_after 字节，则只将 WAL 写入操作系统，而不刷写到磁盘。默认值为 200 毫秒（200ms）。请注意，许多系统的休眠延迟有效分辨率为 10 毫秒；将wal_writer_delay 设置为非 10 的倍数的值，可能与设置为下一个更大的 10 的倍数具有相同效果。此参数只能在postgresql.conf 文件中或在服务器命令行上设置。

wal_writer_flush_after (integer)

指定 WAL 写入器刷写 WAL 的频繁程度。如果上一次刷写距今少于 wal_writer_delay 毫秒，并且此后产生的 WAL 少于wal_writer_flush_after 字节，则只将 WAL 写入操作系统，而不刷写到磁盘。如果wal_writer_flush_after 设置为 0，则立即刷写 WAL 数据。默认值为1MB。此参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

commit_delay (integer)

设置 commit_delay 会在发起 WAL 刷盘之前添加以微秒为单位的时间延迟。如果系统负载足够高，使得在给定时间间隔内有更多事务准备提交，这可以通过允许更多事务通过一次 WAL 刷盘来提高组提交吞吐量。不过，每次 WAL 刷盘的延迟也会因此增加，最多增加 commit_delay 微秒。如果没有其他事务准备提交，等待就没有意义，因此仅当即将发起刷盘时至少还有 commit_siblings 个其他活动事务，才会等待。此外，如果禁用了fsync，也不会等待。默认commit_delay 为零（无延迟）。只有超级用户能更改这个设置。

在 PostgreSQL 的 9.3 发布之前，commit_delay 的行为不同并且效果更差：它只影响提交，而不是所有 WAL 刷写，并且即使 WAL 刷盘更早完成，也会等待整个配置的延迟时间。从PostgreSQL 9.3 开始，第一个准备好刷写的进程会等待配置的间隔，而后续的进程只等到领先者完成刷写操作。

commit_siblings (integer)

执行 commit_delay 延迟前要求的并发活动事务的最小数目。大一些的值会导致在延迟间隔期间更可能有至少另外一个事务准备好提交。默认值是五个事务。

19.5.2. 检查点

checkpoint_timeout (integer)

自动 WAL 检查点之间的最长时间，以秒为单位。合理的范围在 30 秒到 1 天之间。默认是 5 分钟（5min）。增加这个参数的值可能会增加崩溃恢复所需的时间。这个参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

checkpoint_completion_target (floating point)

指定检查点完成的目标，作为检查点之间总时间的一部分。默认是 0.5。这个参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

checkpoint_flush_after (integer)

当执行检查点时写入的数据量超过 checkpoint_flush_after 字节时，就尝试强制 OS 把这些写发送到底层存储。这样做将会限制内核页面高速缓存中的脏数据数量，降低在检查点末尾发出fsync 或者 OS 在后台大批量写回数据时被卡住的可能性。这通常能显著降低事务延迟，但是也有一些情况（特别是负载超过shared_buffers 但小于 OS 页面高速缓存）的性能会降低。这种设置可能会在某些平台上没有效果。合法的范围在 0（禁用强制写回）和 2MB 之间。Linux 上的默认值是 256kB，其他平台上是0（如果 BLCKSZ 不是 8kB，则默认值和最大值会按比例缩放到它）。这个参数只能在postgresql.conf 文件中或者服务器命令行上设置。

checkpoint_warning (integer)

如果由于填充 WAL 段文件导致的检查点之间的间隔低于这个参数表示的秒数，那么就向服务器日志写一个消息（它建议增加max_wal_size 的值）。默认值是 30 秒（30s）。零则关闭警告。如果 checkpoint_timeout 低于 checkpoint_warning，则不会有警告产生。这个参数只能在postgresql.conf 文件中或在服务器命令行上设置。

max_wal_size (integer)

在自动检查点期间允许 WAL 增长的最大大小。这是一个软限制；在特殊情况下，如重负载、失败的archive_command，或高wal_keep_segments 设置下，WAL 大小可能会超过 max_wal_size。默认值为 1 GB。增加此参数可能会增加崩溃恢复所需的时间。此参数只能在 postgresql.conf 文件或服务器命令行中设置。

min_wal_size (integer)

只要 WAL 磁盘用量保持在这个设置之下，在检查点时旧的 WAL 文件总是被回收以便未来使用，而不是直接被删除。这可以被用来确保有足够的 WAL 空间被保留来应付 WAL 使用的高峰，例如运行大型的批处理任务。默认是 80 MB。这个参数只能在postgresql.conf 或者服务器命令行中设置。

19.5.3. 归档

archive_mode (enum)

当启用 archive_mode 时，完成的 WAL 段会通过设置archive_command 发送到归档存储。除了用于禁用归档的 off 外，还有两种模式：on 和 always。在正常操作期间，这两种模式之间没有区别，但当设置为always 时，WAL 归档程序在归档恢复或备库模式下也会被启用。在always 模式下，从归档中恢复的所有文件或通过流复制传输的文件将被再次归档。详细信息请参见第 26.2.9 节。

archive_mode 和 archive_command 是单独的变量，这样archive_command 可以在不离开归档模式的情况下进行更改。此参数只能在服务器启动时设置。当 wal_level 设置为 minimal 时，无法启用archive_mode。

archive_command (string)

本地 shell 命令被执行来归档一个完成的 WAL 文件段。字符串中的任何 %p 被替换成要被归档的文件的路径名，而%f 只被文件名替换（路径名是相对于服务器的工作目录，即集簇的数据目录）。如果要在命令里嵌入一个真正的% 字符，可以使用%%。有一点很重要，该命令只在成功时返回一个零作为退出状态。更多信息请见第 25.3.1 节。

这个参数只能在 postgresql.conf 文件或服务器命令行中设置。除非在服务器启动时启用了 archive_mode，否则将被忽略。如果archive_command 是空字符串（默认值），而 archive_mode 已启用，WAL 归档将暂时被禁用，但服务器将继续积累 WAL 段文件，期望很快会提供命令。将archive_command 设置为一个什么都不做但返回 true 的命令，例如 /bin/true（Windows 上为 REM），实际上禁用了归档，但也破坏了用于归档恢复所需的 WAL 文件链，因此只应在不寻常的情况下使用。

archive_timeout (integer)

archive_command 只针对已完成的 WAL 段调用。因此，如果您的服务器生成的 WAL 流量较少（或者在这样做时有间歇期），在事务完成和安全记录到归档存储之间可能会有很长的延迟。为了限制未归档数据的年龄，您可以将archive_timeout 设置为强制服务器定期切换到新的 WAL 段文件。当此参数大于零时，只要自上次段文件切换以来经过了指定的秒数，并且存在任何数据库活动，包括单个检查点（如果没有数据库活动，则跳过检查点），服务器将切换到新的段文件。请注意，由于强制切换而提前关闭的归档文件仍然与完全填满的文件长度相同。因此，使用非常短的archive_timeout 是不明智的，它会使您的归档存储膨胀。通常，将archive_timeout 设置为一分钟左右是合理的。如果您希望数据比这更快地从主库复制出来，您应该考虑使用流复制而不是归档。此参数只能在postgresql.conf 文件或服务器命令行中设置。

19.6. 复制

这些设置控制内置流复制特性（见第 26.2.5 节）的行为。服务器将可以是主库或备库。主库能发送数据，而备库总是被复制数据的接收者。当使用级联复制（见第 26.2.7 节）时，备库也可以是发送者，同时也是接收者。这些参数主要用于发送服务器和备库，尽管某些只在主库上有意义。如果有必要，设置可以在集簇中变化而不出问题。

19.6.1. 发送服务器

这些参数可以在任何发送复制数据给一个或多个备库的服务器上设置。主库总是一个发送服务器，因此这些参数总是要在主库上设置。这些参数的角色和含义不会在一个备库变成主库后改变。

max_wal_senders (integer)

指定来自备库或流式基础备份客户端的并发连接的最大数量（即同时运行 WAL 发送进程的最大数）。默认值是 10。值 0 意味着禁用复制。WAL 发送进程计入总连接数，因此此参数的值必须小于max_connections 减去 superuser_reserved_connections。流式客户端突然断开连接可能留下一个孤立连接槽，直到达到超时，因此此参数应设置得略高于预期的最大客户端数，使断开连接的客户端能够立即重新连接。此参数只能在服务器启动时设置。此外，wal_level 必须设置为 replica 或更高级别，才允许来自备库的连接。

max_replication_slots (integer)

指定服务器可以支持的复制槽（见第 26.2.6 节）最大数量。默认值为 10。这个参数只能在服务器启动时设置。将它设置为一个比当前已有复制槽要少的值会阻碍服务器启动。此外，要允许使用复制槽，wal_level 必须被设置为 replica 或更高。

在订阅者端，指定可以同时跟踪多少个复制源（参见第 50 章），从而限制可以在服务器上创建的逻辑复制订阅数。如果将其设置为低于当前正在跟踪的复制源数量的值（该数量反映在 pg_replication_origin_status 中，而不是 pg_replication_origin 中），服务器将无法启动。

wal_keep_segments (integer)

指定在 pg_wal 目录中保留的过去日志文件段的最小数量，以便备库可能需要获取它们来进行流复制。每个段通常为 16 兆字节。如果连接到发送服务器的备库落后超过 wal_keep_segments 个段，发送服务器可能会移除备库仍需要的 WAL 段，在这种情况下，复制连接将被终止。下游连接最终也会因此失败。（但是，如果使用了 WAL 归档，备库可以通过从归档中获取该段来恢复。）

此设置只指定在 pg_wal 中保留的最小段数；系统可能需要为 WAL 归档或从检查点恢复而保留更多段。如果 wal_keep_segments 为零（默认值），系统不会为备库额外保留任何段，因此备库可用的旧 WAL 段数取决于前一个检查点的位置和 WAL 归档的状态。此参数只能在postgresql.conf 文件中或在服务器命令行上设置。

wal_sender_timeout (integer)

终止处于非活动状态超过指定毫秒数的复制连接。这有助于发送服务器检测备库崩溃或网络中断。值零禁用超时机制。此参数只能在 postgresql.conf 文件中或在服务器命令行上设置。默认值为 60 秒。

track_commit_timestamp (boolean)

记录事务的提交时间。该参数只能在 postgresql.conf 文件中或在服务器命令行上设置。默认值是 off。

19.6.2. 主库

这些参数可以在向一个或多个备库发送复制数据的主库上设置。除这些参数外，还必须在主库上适当设置 wal_level，也可以选择启用 WAL 归档（见第 19.5.3 节）。这些参数在备库上的取值不影响备库运行，不过也可以预先设置，以备将来提升为主库。

synchronous_standby_names (string)

如第 26.2.8 节所述，这个参数指定一个支持同步复制的备库的列表。将有一个或多个活动的同步备库，在这些备库确认收到它们的数据之后，等待提交的事务将被允许继续下去。同步备库是那些名字出现在这个列表中，并且当前已连接并且正在实时流式传输数据（如

pg_stat_replication 视图中 streaming 的状态所示）的服务器。指定多于一台同步备库可以得到非常高的可用性并且能防止数据丢失。

用于这一目的的备库名称是其连接信息中设置的 application_name。对于物理复制备库，应在recovery.conf 的 primary_conninfo 设置中指定；默认值为 walreceiver。对于逻辑复制，可以在订阅的连接信息中设置，默认值为订阅名称。对于其他复制流消费者，请参考其文档。

这个参数使用下面的语法之一来指定一个备库列表：

```
[FIRST] num_sync ( standby_name [, ...] )
ANY num_sync ( standby_name [, ...] )
standby_name [, ...]
```

其中 *num_sync* 是事务需要等待其回复的同步备库的数量，*standby_name* 是一个备库的名称。FIRST 以及 ANY 指定从所列服务器中选取同步备库的方法。

关键词 FIRST 加上 *num_sync* 指定一种基于优先级的同步复制，并且会让事务提交等待，直到它们的 WAL 记录被复制到基于优先级选择的*num_sync* 台同步备库上为止。例如，设置FIRST 3 (s1, s2, s3, s4) 将导致每次提交都等待来自三台较高优先级的备库的答复，这三台备库将从备库s1、s2、s3 以及 s4 中选出。在该列表中出现较早的备库将被给予较高的优先级，并且将被考虑为同步备库。列表中出现的其他备库表示潜在的同步备库。如果当前的任何同步备库因为某种原因断开连接，它将立刻被下一个最高优先级的备库替代。关键词FIRST 是可选的。

关键词 ANY 加上 *num_sync* 指定一种基于法定人数的同步复制，并且会让事务提交等待，直到它们的 WAL 记录被复制到所列出备库中的至少 *num_sync* 台上为止。例如，设置 ANY 3 (s1, s2, s3, s4) 将导致每次提交会在收到 s1、s2、s3 以及 s4 中任意三台备库的回答后立刻继续下去。

FIRST 和 ANY 是大小写不敏感的。如果这些关键词被用作备库的名字，其*standby_name* 必须被放在双引号内。

PostgreSQL 版本 9.6 之前使用过第三种语法，目前也仍然支持。它和FIRST 和 *num_sync* 等于 1 的第一种语法相同。例如，FIRST 1 (s1, s2) 和 s1, s2 具有相同的含义：s1 或者 s2 会被选中作为同步备库。

特殊项* 匹配任意备库名称。

没有机制强制备库名称的唯一性。在出现重复的情况下，匹配的备库之一将被认为是较高优先级，不过无法弄清到底是哪一个。

> **注意**
>
> 每一个 *standby_name* 都应该具有合法 SQL 标识符的形式，除非它是*。如果必要你可以使用双引号。但是注意在比较 *standby_name* 和备库应用程序名称时是大小写不敏感的（不管有没有双引号）。

如果这里没有指定同步备库名称，那么不启用同步复制并且事务提交将不会等待复制。这是默认的配置。即便当同步复制被启用时，个体事务也可以被配置为不等待复制，做法是将 synchronous_commit 参数设置为 local 或 off。

这个参数只能在 postgresql.conf 文件中或通过服务器命令行进行设置。

vacuum_defer_cleanup_age (integer)

指定 VACUUM 和 HOT 更新延迟清理死行版本的事务数。默认值为零个事务，意味着可以尽快移除死行版本，也就是在它们不再对任何打开的事务可见时立即移除。如第 26.5 节所述，在为热备服务器提供支持的主库上，你可能希望将此参数设置为非零值。这让备库上的查询有更多时间完成，而不会因过早清理行而发生冲突。但是，由于此值以主库上发生的写事务数计量，很难预测会给备库查询带来多少额外的宽限时间。此参数只能在postgresql.conf 文件中或在服务器命令行上设置。

你也应考虑在备库上设置 hot_standby_feedback，作为使用此参数的替代方案。

这不会阻止清理已经达到 old_snapshot_threshold 指定年龄的死行。

19.6.3. 备库

这些设置控制备库接收复制数据时的行为。它们在主库的值是无关的。

hot_standby (boolean)

指定在恢复期间，你是否能够连接并运行查询，如第 26.5 节中所述。默认值是 on。这个参数只能在服务器启动时设置。它只在归档恢复期间或备库模式下才有效。

max_standby_archive_delay (integer)

当热备处于活动状态时，此参数确定备库在取消与即将应用的 WAL 条目冲突的备库查询之前应等待多长时间，如第 26.5.2 节中所述。max_standby_archive_delay 在从 WAL 归档中读取 WAL 数据时适用（因此不是当前的）。如果未指定单位，则将其视为毫秒。默认值为 30 秒。值为-1 允许备库永远等待冲突查询完成。此参数只能在postgresql.conf 文件或服务器命令行中设置。

注意，max_standby_archive_delay 并不等同于查询在被取消前可以运行的最长时间；它表示应用任意一个 WAL 段的数据所允许的最长总时间。因此，如果某个查询先前在处理该 WAL 段时已造成显著延迟，后续冲突查询的宽限时间就会短得多。

max_standby_streaming_delay (integer)

当热备处于活动状态时，此参数确定备库在取消与即将应用的 WAL 条目冲突的备库查询之前应等待多长时间，如第 26.5.2 节中所述。max_standby_streaming_delay 在通过流复制接收 WAL 数据时应用。如果未指定单位，则将其视为毫秒。默认值为 30 秒。值为-1 允许备库永远等待冲突查询完成。此参数只能在postgresql.conf 文件或服务器命令行中设置。

注意，max_standby_streaming_delay 并不等同于查询在被取消前可以运行的最长时间；它表示从主库接收到 WAL 数据后，允许用于应用这些数据的最长总时间。因此，如果某个查询已造成显著延迟，后续冲突查询的宽限时间就会短得多，直到备库再次赶上进度。

wal_receiver_status_interval (integer)

指定在备库上的 WAL 接收进程向主库或上游备库发送有关复制进度的信息的最小频度，它可以使用pg_stat_replication 视图看到。备库将报告最后写入的预写日志位置、最后刷盘的位置以及最后应用的位置。这个参数的值是报告之间的最大间隔，以秒为单位。每次写入或刷盘位置改变时会发送状态更新，或者至少按这个参数指定的频度发送。因此，应用位置可能比真实位置略微滞后。默认值是 10 秒。将此参数设置为零会完全禁用状态更新。这个参数只能在postgresql.conf 文件中或在服务器命令行上设置。

hot_standby_feedback (boolean)

指定一个热备机是否将会向主库或上游备库发送有关于备库上当前正被执行的查询的反馈。这个参数可以被用来消除由清理记录引起的查询取消，但是可能导致在主库上用于某些负载的数据库膨胀。反馈消息的发送频度不会高于每个wal_receiver_status_interval 周期发送一次。默认值是 off。这个参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

如果使用级联复制，反馈将被向上游传递直到它最后到达主库。备库在接收到反馈之后除了传递给上游不会做任何其他操作。

此设置不会覆盖主库上 old_snapshot_threshold 的行为。如果备库上的快照超过了主库的快照年龄阈值，它可能失效，从而导致备库上的事务被取消。这是因为 old_snapshot_threshold 旨在为死行版本导致表膨胀的持续时间设定一个绝对限制，而备库的配置不应允许违反这个限制。

wal_receiver_timeout (integer)

终止处于非活动状态超过指定毫秒数的复制连接。这有助于接收数据的备库检测主库节点崩溃或网络中断。值零禁用超时机制。此参数只能在 postgresql.conf 文件中或在服务器命令行上设置。默认值为 60 秒。

wal_retrieve_retry_interval (integer)

指定当从任何来源（流复制、本地pg_wal 或者 WAL 归档）都得不到 WAL 数据时，备库应该等待多久才去重新尝试获取 WAL 数据。如果指定值时没有单位，则以毫秒为单位。默认值是 5 秒。这个参数只能在postgresql.conf 文件或者服务器命令行中设置。

当恢复中的节点需要控制等待新 WAL 数据可用的时间时，此参数很有用。例如，在归档恢复中，减小此参数的值可以让恢复更快地检测到新的 WAL 日志文件。在 WAL 活动较少的系统上，增大此值会减少访问 WAL 归档所需的请求数；这在例如按基础设施访问次数计量的云环境中很有用。

19.6.4. 订阅者

这些设置控制逻辑复制订阅者的行为。它们在发布者上的值与此无关。

注意，配置参数wal_receiver_timeout、wal_receiver_status_interval 以及 wal_retrieve_retry_interval 也影响逻辑复制工作进程。

max_logical_replication_workers (int)

指定逻辑复制工作进程的最大数量，其中包括应用工作进程和表同步工作进程。

逻辑复制工作进程来自 max_worker_processes 定义的进程池。

默认值是 4。此参数只能在服务器启动时设置。

max_sync_workers_per_subscription (integer)

每个订阅允许的最大同步工作进程数。该参数控制订阅初始化时或新增表时初始数据复制的并行度。

目前，每个表只能有一个同步工作进程。

同步工作进程来自 max_logical_replication_workers 定义的进程池。

默认值是 2。此参数只能在 postgresql.conf 文件中或服务器命令行上设置。

19.7. 查询规划

19.7.1. 规划器方法配置

这些配置参数提供了一种较为粗糙的方法，用于影响查询优化器所选择的查询计划。如果优化器为某个特定查询选择的默认计划并不理想，一种临时解决方案是使用这些配置参数，强制优化器选择另一种计划。改善优化器所选计划质量的更好办法包括调整规划器代价常数（见第 19.7.2 节）、手工运行 ANALYZE、增加default_statistics_target 配置参数的值，以及使用 ALTER TABLE SET STATISTICS 增加为特定列收集的统计信息量。

enable_bitmapscan (boolean)

启用或禁用查询规划器对位图扫描计划类型的使用。默认值为 on。

enable_gathermerge (boolean)

启用或禁用查询规划器对收集归并计划类型的使用。默认值为 on。

enable_hashagg (boolean)

启用或禁用查询规划器对 hash 聚合计划类型的使用。默认值为 on。

enable_hashjoin (boolean)

启用或禁用查询规划器对哈希连接计划类型的使用。默认值为 on。

enable_indexscan (boolean)

启用或禁用查询规划器对索引扫描计划类型的使用。默认值为 on。

enable_indexonlyscan (boolean)

启用或禁用查询规划器对仅索引扫描计划类型的使用（参见第 11.9 节）。默认值为 on。

enable_material (boolean)

允许或者禁止查询规划器使用物化。它不可能完全禁用物化，但是关闭这个变量将阻止规划器插入物化节点，除非为了保证正确性。默认值是on。

enable_mergejoin (boolean)

启用或禁用查询规划器对归并连接计划类型的使用。默认值为 on。

enable_nestloop (boolean)

允许或禁止查询规划器使用嵌套循环连接计划。它不可能完全禁止嵌套循环连接，但是关闭这个变量将使得规划器尽可能优先使用其他方法。默认值是on。

enable_parallel_append (boolean)

启用或禁用查询规划器对支持并行执行的 append 计划类型的使用。默认值为 on。

enable_parallel_hash (boolean)

启用或禁用查询规划器对使用并行 hash 的哈希连接计划类型的使用。如果没有同时启用哈希连接计划，此参数不起作用。默认值为 on。

enable_partition_pruning (boolean)

允许或者禁止查询规划器从查询计划中消除一个分区表的分区。这也控制着规划器产生允许执行器在查询执行期间移除（忽略）分区的查询计划的能力。默认值是 on。详情请参考第 5.10.4 节。

enable_partitionwise_join (boolean)

启用或禁用查询规划器对分区连接的使用，使分区表之间的连接可以通过连接相匹配的分区来完成。分区连接目前仅适用于连接条件包含所有分区键的情况；这些分区键必须具有相同的数据类型，并且子分区集合必须完全匹配。由于分区连接规划可能在规划期间使用明显更多的 CPU 时间和内存，默认值为 off。

enable_partitionwise_aggregate (boolean)

启用或禁用查询规划器对分区分组或聚合的使用，使分区表上的分组或聚合可以在各个分区上分别执行。如果 GROUP BY 子句不包含分区键，则只能在各分区上执行部分聚合，随后还必须进行最终处理。由于分区分组或聚合可能在规划期间使用明显更多的 CPU 时间和内存，默认值为 off。

enable_seqscan (boolean)

允许或禁止查询规划器使用顺序扫描计划类型。它不可能完全禁止顺序扫描，但是关闭这个变量将使得规划器尽可能优先使用其他方法。默认值是on。

enable_sort (boolean)

允许或禁止查询规划器使用显式排序步骤。它不可能完全禁止显式排序，但是关闭这个变量将使得规划器尽可能优先使用其他方法。默认值是on。

enable_tidscan (boolean)

允许或禁止查询规划器使用 TID 扫描计划类型。默认值是 on。

19.7.2. 规划器代价常量

这一节中描述的代价变量可以按照任意尺度衡量。我们只关心它们的相对值，将它们以相同的因子缩放不会影响规划器的选择。默认情况下，这些代价变量是基于顺序页面获取的代价的，即 seq_page_cost 被设置为 1.0 并且其他代价变量都参考它来设置。不过你可以使用你喜欢的不同尺度，例如在一个特定机器上以毫秒为单位的实际执行时间。

> **注意**
>
> 没有明确的方法可以确定代价变量的理想值。最好将它们视为某个数据库安装实例所接收的全部查询组合的平均值。因此，仅凭少数几次试验就修改它们是非常冒险的。

`seq_page_cost`(`floating point`)

设置规划器对一系列顺序磁盘页面读取中单次读取的代价估计。默认值是 1.0。对于某个表空间内的表和索引，可以通过设置该表空间的同名参数来覆盖此值（见 ALTER TABLESPACE）。

`random_page_cost`(`floating point`)

设置规划器对一次非顺序磁盘页面读取的代价估计。默认值是 4.0。对于某个表空间内的表和索引，可以通过设置该表空间的同名参数来覆盖此值（见 ALTER TABLESPACE）。

减少这个值（相对于 `seq_page_cost`）将导致系统更倾向于索引扫描；提高它将让索引扫描看起来相对更昂贵。你可以一起提高或降低两个值来改变磁盘 I/O 代价相对于 CPU 代价的重要性，后者由下列参数描述。

对机械磁盘存储的随机访问通常远不止比顺序访问贵四倍。不过，仍使用较低的默认值（4.0），因为假定对磁盘的大多数随机访问（例如索引读取）都将在缓存中命中。可以将默认值理解为：假设随机访问比顺序访问慢 40 倍，同时预期 90% 的随机读取能够命中缓存。

如果你认为 90% 的缓存命中率不符合你的工作负载，可以增大 random_page_cost，以更好地反映随机存储读取的真实代价。对应地，如果你的数据很可能完全缓存在内存中，例如数据库小于服务器总内存，则降低 random_page_cost 可能更合适。对于随机读取代价相对于顺序读取较低的存储，例如固态硬盘，也可以用较低的 random_page_cost 值来更好地建模，例如`1.1`。

> **提示**
>
> 尽管系统允许将 `random_page_cost` 设置得小于 `seq_page_cost`，但这不符合实际物理情况。不过，如果数据库完全缓存在 RAM 中，将它们设置为相等是合理的，因为此时非顺序访问页面不会产生额外代价。同样，对于大部分数据已缓存的数据库，应相对于 CPU 参数降低这两个值，因为读取一个已在 RAM 中的页面的代价远小于通常的页面读取代价。

`cpu_tuple_cost`(`floating point`)

设置规划器对一次查询中处理每一行的代价估计。默认值是 0.01。

`cpu_index_tuple_cost`(`floating point`)

设置规划器对一次索引扫描中处理每一个索引项的代价估计。默认值是 0.005。

`cpu_operator_cost`(`floating point`)

设置规划器对于一次查询中处理每个操作符或函数的代价估计。默认值是 0.0025。

`parallel_setup_cost`(`floating point`)

设置规划器对启动并行工作进程的代价估计。默认是 1000。

`parallel_tuple_cost`(`floating point`)

设置规划器对于从一个并行工作进程传递一个元组给另一个进程的代价估计。默认是 0.1。

min_parallel_table_scan_size (integer)

设置考虑使用并行扫描时必须扫描的最小表数据量。对于并行顺序扫描，被扫描的表数据量总是等于表的尺寸，但是在使用索引时，被扫描的表数据量通常会更小。默认值是 8 兆字节（8MB）。

min_parallel_index_scan_size (integer)

设置考虑使用并行扫描时必须扫描的最小索引数据量。注意，并行索引扫描通常不会访问整个索引；这里考虑的是规划器认为扫描实际会访问的页面数量。默认值是 512 千字节（512kB）。

effective_cache_size (integer)

设置规划器对一个单一查询可用的有效磁盘缓存尺寸的假设。这个参数会被考虑在使用一个索引的代价估计中，更高的数值会使得索引扫描更可能被使用，更低的数值会使得顺序扫描更可能被使用。在设置这个参数时，你还应该考虑PostgreSQL 的共享缓冲区以及将被用于PostgreSQL 数据文件的内核磁盘缓存，尽管有些数据可能在两个地方都存在。另外，还要考虑预计在不同表上的并发查询数目，因为它们必须共享可用的空间。这个参数对PostgreSQL 分配的共享内存尺寸没有影响，它也不会预留内核磁盘缓存，它只用于估计的目的。系统也不会假设在查询之间数据会保留在磁盘缓存中。默认值是 4 吉字节（4GB）。

jit_above_cost (floating point)

设置激活 JIT 编译的查询代价，如果查询代价超过这个值就会激活 JIT 编译（如果启用了 JIT，见第 32 章）。执行 JIT 会消耗一些规划时间，但是能够加速查询执行。将这个值设置为-1 会禁用 JIT 编译。默认值是 100000。

jit_inline_above_cost (floating point)

设置 JIT 编译尝试内联函数和操作符的查询代价阈值，如果查询代价超过这个值，JIT 编译就会尝试内联。内联会增加规划时间，但是可以提高执行速度。将这个参数设置成小于 jit_above_cost 是没有意义的。将这个参数设置为-1 会禁用内联。默认值是 500000。

jit_optimize_above_cost (floating point)

设置 JIT 编译应用优化的查询代价阈值，如果查询代价超过这个值，JIT 编译就会应用开销较大的优化。这类优化会增加规划时间，但可以提高执行速度。将这个参数设置成小于 jit_above_cost 是没有意义的，并且将它设置成大于jit_inline_above_cost 也不太可能有益。将这个参数设置为-1 会禁用开销较大的优化。默认值是 500000。

19.7.3. 遗传查询优化器

遗传查询优化器（GEQO）是一种使用启发式搜索进行查询规划的算法。它可以缩短复杂查询（连接很多关系的查询）的规划时间，代价是生成的计划有时不如常规穷举搜索算法找到的计划。更多信息见第 60 章。

geqo (boolean)

允许或禁止遗传查询优化。默认是启用。在生产环境中通常最好不要关闭它。geqo_threshold 变量提供了对 GEQO 更细粒度的控制。

geqo_threshold (integer)

对涉及至少这么多个 FROM 项的查询，使用遗传查询优化来生成计划。（注意，一个 FULL OUTER JOIN 结构只计为一个 FROM 项。）默认值是 12。对于更简单的查询，通常

最好使用常规的穷举搜索规划器；但对于涉及很多表的查询，穷举搜索花费的时间太长，往往超过执行次优计划所增加的耗时。因此，以查询规模为阈值，可以方便地控制 GEQO 的使用。

`geqo_effort` (`integer`)

控制 GEQO 中规划时间和查询计划质量之间的权衡。此变量必须是 1 到 10 之间的整数。默认值是 5。较大的值会增加查询规划所用的时间，但也会提高选择高效查询计划的可能性。

`geqo_effort` 实际并不直接做任何事情；它只是被用来计算其他影响 GEQO 行为的变量（如下所述）的默认值。如果你愿意，你可以手工设置其他参数。

`geqo_pool_size` (`integer`)

控制 GEQO 使用的池大小，它就是遗传种群中的个体数目。它必须至少为 2，且有用的值通常在 100 到 1000 之间。如果它被设置为零（默认设置）则会基于`geqo_effort` 和查询中表的数量选择一个合适的值。

`geqo_generations` (`integer`)

控制 GEQO 使用的代数，即算法的迭代次数。它必须至少为 1，通常有用的值与池大小处于相同范围。如果设置为零（默认设置），则根据`geqo_pool_size` 选择合适的值。

`geqo_selection_bias` (`floating point`)

控制 GEQO 使用的选择偏好。选择偏好是种群中的选择压力。值可以是 1.5 到 2.0 之间，后者是默认值。

`geqo_seed` (`floating point`)

控制 GEQO 使用的随机数生成器的初始值，随机数生成器用于在连接顺序搜索空间中选择随机路径。该值可以从 0（默认值）到 1。改变该值会改变所探索的连接路径集合，最终找到的最佳路径可能更好，也可能更差。

19.7.4. 其他规划器选项

`default_statistics_target` (`integer`)

为没有通过 `ALTER TABLE SET STATISTICS` 设置列相关目标的表列设置默认统计目标。更大的值增加了需要做 `ANALYZE` 的时间，但是可能会改善规划器的估计质量。默认值是 100。有关PostgreSQL 查询规划器使用的统计信息的更多内容，请参考第 14.2 节。

`constraint_exclusion` (`enum`)

控制查询规划器对表约束的使用，以优化查询。`constraint_exclusion` 的允许值是 `on`（对所有表检查约束）、`off`（从不检查约束）和 `partition`（只对继承的子表和 `UNION ALL` 子查询检查约束）。`partition` 是默认设置。它通常与传统的继承树一起使用来提高性能。

当此参数允许对某个表使用约束排除时，规划器会比较查询条件与该表的CHECK 约束，并且忽略那些条件违反约束的表扫描。例如：

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

在启用约束排除时，这个SELECT 将完全不会扫描 child1000，从而提高性能。

目前，约束排除仅在通常用于通过继承树实现表分区的情况下默认启用。为所有表启用它会增加额外的规划开销，这在简单查询上相当明显，而且通常不会为简单查询带来好处。如果没有通过传统继承方式进行分区的表，你可能希望完全关闭它。（注意，分区表的等效功能由另一个参数enable_partition_pruning 控制。）

更多关于使用约束排除实现分区的信息请参阅第 5.10.5 节。

cursor_tuple_fraction (floating point)

设置规划器对将被检索的一个游标的行的比例的估计。默认值是 0.1。更小的值使得规划器偏向为游标使用“快速开始”计划，它将很快地检索前几行但是可能需要很长时间来获取所有行。更大的值强调总的估计时间。最大设置为 1.0，游标将和普通查询完全一样地被规划，只考虑总估计时间并且不考虑前几行会被多快地返回。

from_collapse_limit (integer)

如果生成的 FROM 列表不超过这么多项，规划器将把子查询融合到上层查询。较小的值可以减少规划时间，但是可能会生成较差的查询计划。默认值是 8。详见第 14.3 节。

将这个值设置为 geqo_threshold 或更大，可能触发使用 GEQO 规划器，从而产生非最优计划。见第 19.7.3 节。

jit (boolean)

决定如果可用（见第 32 章），PostgreSQL 是否可以使用 JIT 编译。默认值是 off。

join_collapse_limit (integer)

如果得出的列表中不超过这么多项，那么规划器将把显式JOIN（除了 FULL JOIN）结构重写到 FROM 项列表中。较小的值可减少规划时间，但是可能会生成差些的查询计划。

默认情况下，这个变量被设置成和from_collapse_limit 相同，这样适合大多数使用。把它设置为 1 可避免任何显式JOIN 的重排序。因此查询中指定的显式连接顺序就是关系被连接的实际顺序。因为查询规划器并不是总能选取最优的连接顺序，高级用户可以选择暂时把这个变量设置为 1，然后显式地指定他们想要的连接顺序。更多信息请见第 14.3 节。

将这个值设置为 geqo_threshold 或更大，可能触发使用 GEQO 规划器，从而产生非最优计划。见第 19.7.3 节。

parallel_leader_participation (boolean)

允许领导者进程执行 Gather 和 Gather Merge 节点下的查询计划，而不是等待工作进程。默认值为 on。将此值设置为 off，可以降低工作进程因领导者读取元组不够快而被阻塞的可能性，但在产生第一批元组之前，领导者进程必须等待工作进程启动。领导者对性能的帮助或妨碍程度取决于计划类型、工作进程数量和查询持续时间。

force_parallel_mode (enum)

允许为测试目的使用并行查询，即使预期不会带来性能收益。force_parallel_mode 允许的值包括 off（仅在预期能提高性能时使用并行模式）、on（对所有被认为可以安全并行的查询强制使用并行查询），以及 regress（类似于 on，但还有下文说明的额外行为变化）。

更具体地说，将此值设置为on 会在任何看起来可以安全并行的查询计划顶部添加一个 Gather 节点，让查询在并行工作进程中运行。即使没有可用的并行工作进程或无法使用并行工作进程，在并行查询上下文中不允许的操作（例如启动子事务）也会被禁止，除非规划器认为这会使查询失败。如果设置此选项后出现失败或意外结果，查询使用的某些函数可能需要被标记为PARALLEL UNSAFE（也可能是 PARALLEL RESTRICTED）。

将此值设置为 regress 具有设置为 on 的所有效果，另外还有一些旨在方便自动回归测试的效果。通常，来自并行工作进程的消息会包含一行说明这一点的上下文信息，但regress 设置会抑制此行，使输出与非并行执行时相同。此外，此设置添加到计划中的Gather 节点会在 EXPLAIN 输出中隐藏，使输出与此设置为off 时的输出一致。

19.8. 错误报告和日志

19.8.1. 日志记录到哪里

log_destination (string)

PostgreSQL 支持多种记录服务器消息的方法，包括stderr，csvlog 和 syslog。在 Windows 上，还支持eventlog。将此参数设为所需日志目的地的逗号分隔列表。默认只将日志记录到 stderr。此参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

如果 csvlog 被包括在 log_destination 中，日志项会以“逗号分隔值”（CSV）格式被输出，这样可以很方便地把日志载入到程序中。详见第 19.8.4 节。要产生 CSV 格式的日志输出，必须启用 logging_collector。

如果包含 stderr 或 csvlog，就会创建文件current_logfiles，记录日志收集器当前使用的日志文件位置及相关日志目的地。这样便于找到实例当前使用的日志。以下是此文件内容的示例：

```
stderr log/postgresql.log
csvlog log/postgresql.csv
```

current_logfiles 会在日志轮转产生新日志文件时重新创建，也会在重新载入 log_destination 时重新创建。如果 stderr 和 csvlog 均未包含在 log_destination 中，或者禁用了日志收集器，则会移除此文件。

> **注意**
>
> 在大多数 Unix 系统上，你将需要修改系统的syslog 守护进程的配置来使用 log_destination 的 syslog 选项。PostgreSQL 可以在 syslog 设施 LOCAL0 到 LOCAL7 中记录（见 syslog_facility），但是大部分平台上的默认syslog 配置会丢弃所有这种消息。你将需要增加这样的内容：
>
> ```
> local0.*    /var/log/postgresql
> ```
>
> 到 syslog 守护进程的配置文件来让它工作。
>
> 在 Windows 上，当你使用log_destination 的 eventlog 选项时，你应该在操作系统中注册一个事件源及其库，这样 Windows 事件查看器能够清楚地显示事件日志消息。详见第 18.11 节。

logging_collector (boolean)

这个参数启用日志收集器，它是一个捕捉被发送到stderr 的日志消息的后台进程，并且它会将这些消息重定向到日志文件中。这种方法比记录到syslog 通常更有用，因为某些类型的消息可能不会在syslog 输出中出现（一个常见的示例是动态链接器错误消息；另一个示例是由 archive_command 等脚本产生的错误消息）。这个参数只能在服务器启动时设置。

> **注意**
>
> 也可以不使用日志收集器而把日志记录到 stderr，日志消息将只会去到服务器的 stderr 被定向到的位置。不过，那种方法只适合于低日志量，因为它没有提供便捷的方法来轮转日志文件。还有，在某些平台上，不使用日志收集器可能会导致日志输出丢失或混杂，因为多个进程并发写入同一个日志文件时会覆盖彼此的输出。

> **注意**
>
> 日志收集器被设计成从来不会丢失消息。这意味着在极高的负载下，如果服务器进程试图在收集器已经落后时发送更多的日志消息，那么它可能会被阻塞。相反，syslog 倾向于在无法写入消息时丢掉消息，这意味着在这样的情况下它可能会无法记录某些消息，但是它不会阻塞系统的其他部分。

log_directory (string)

当 logging_collector 被启用时，这个参数决定日志文件将被在哪个目录下创建。它可以被指定为一个绝对路径，也可以被指定为一个相对于集簇数据目录的相对路径。这个参数只能在postgresql.conf 文件中或在服务器命令行上设置。默认是 log。

log_filename (string)

当 logging_collector 被启用时，这个参数设置被创建的日志文件的文件名。该值被视为一种strftime 模式，因此% 转义可以被用来指定根据时间变化的文件名（注意如果有任何依赖时区的% 转义，计算将在由log_timezone 指定的时区中完成）。被支持的% 转义和开

放组织的 strftime[1] 说明中列举的类似。注意系统的 strftime 不会被直接使用，因此平台相关（非标准）的扩展无法工作。默认是postgresql-%Y-%m-%d_%H%M%S.log。

如果你不使用转义来指定一个文件名，你应该计划使用一个日志轮转工具来避免最终填满整个磁盘。在 8.4 发行之前，如果不存在% 转义，PostgreSQL 将追加新日志文件创建时的纪元时间戳，但是现在已经不再这样做了。

如果在 log_destination 中启用了 CSV 格式输出，.csv 将会被追加到时间戳日志文件名中来创建 CSV 格式输出（如果 log_filename 以.log 结尾，该后缀会被替换）。

这个参数只能在 postgresql.conf 文件中或通过服务器命令行进行设置。

log_file_mode (integer)

在 Unix 系统上，当logging_collector 被启用时，这个参数设置日志文件的权限（在微软 Windows 上这个参数将被忽略）。这个参数值应当是一个数字形式的模式，它可以被 chmod 和 umask 系统调用接受（要使用通常的八进制格式，该数字必须以一个0（零）开始）。

默认的权限是 0600，表示只有服务器拥有者才能读取或写入日志文件。其他常用的设置是 0640，它允许拥有者的组成员读取文件。不过要注意你需要修改log_directory 为将文件存储在集簇数据目录之外的某个位置，才能利用这个设置。在任何情况下，让日志文件变成任何人都可读是不明智的，因为日志文件中可能包含敏感数据。

这个参数只能在 postgresql.conf 文件中或通过服务器命令行进行设置。

log_rotation_age (integer)

当 logging_collector 被启用时，这个参数决定单个日志文件的最长使用时间。经过指定的分钟数之后，将创建一个新的日志文件。将这个参数设置为零将禁用基于时间的新日志文件创建。这个参数只能在postgresql.conf 文件中或在服务器命令行上设置。

log_rotation_size (integer)

当 logging_collector 被启用时，这个参数决定单个日志文件的最大尺寸。当指定千字节数的数据被写入一个日志文件后，将创建一个新的日志文件。设置为零时将禁用基于大小创建新的日志文件。这个参数只能在postgresql.conf 文件中或在服务器命令行上设置。

log_truncate_on_rotation (boolean)

当 logging_collector 被启用时，这个参数将导致PostgreSQL 截断（覆盖而不是追加）任何已有的同名日志文件。不过，截断只在一个新文件由于基于时间的轮转被打开时发生，在服务器启动或基于尺寸的轮转时不会发生。如果被关闭，在所有情况下以前存在的文件将被追加。例如，使用这个设置和一个类似postgresql-%H.log 的 log_filename 将导致产生 24 个每小时的日志文件，并且循环地覆盖它们。这个参数只能在postgresql.conf 文件中或在服务器命令行上设置。

示例：保留 7 天的日志，每天一个日志文件，命名为server_log.Mon，server_log.Tue，等等，并自动用本周的日志覆盖上周的日志，将log_filename 设置为 server_log.%a，将log_truncate_on_rotation 设置为 on，将 log_rotation_age 设置为 1440。

[1] https://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html

示例：要保留 24 小时的日志，每个小时一个日志文件，如果日志文件尺寸超过 1GB，也会提前轮转。可以这样做：将log_filename 设置为 server_log.%H%M、将log_truncate_on_rotation 设置为 on、将log_rotation_age 设置为 60 并且将 log_rotation_size 设置为 1000000。在 log_filename 中包含 %M，可让按文件大小触发的轮转选用与整点初始文件名不同的新文件名。

syslog_facility (enum)

当启用了向 syslog 记录时，这个参数决定要使用的syslog “设施” 。你可以在 LOCAL0、LOCAL1、LOCAL2、LOCAL3、LOCAL4、LOCAL5、LOCAL6、LOCAL7 中选择，默认值是 LOCAL0。还请参阅系统的 syslog 守护进程的文档。这个参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

syslog_ident (string)

当启用了向 syslog 记录时，这个参数决定用来标识syslog 中的 PostgreSQL 消息的程序名。默认值是 postgres。这个参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

syslog_sequence_numbers (boolean)

当日志被记录到 syslog 并且这个设置为 on（默认）时，每一个消息会被加上一个增长的序号作为前缀（例如[2]）。这种行为避开了很多 syslog 实现默认采用的 “--- 上一个消息重复 N 次 ---” 形式。在现代 syslog 实现中，抑制重复消息是可以配置的（例如rsyslog 中的 $RepeatedMsgReduction），因此这个参数可能不是必需的。此外，如果你真的想抑制重复消息，你可以把这个参数设置为 off。

这个参数只能在 postgresql.conf 文件或者服务器命令行上设置。

syslog_split_messages (boolean)

当启用把日志记录到 syslog 时，这个参数决定消息如何送达 syslog。当设置为 on（默认）时，消息会被分成行，并且长的行也会被划分以便能够放到 1024 字节中，这是传统 syslog 实现的一种典型尺寸限制。当设置为 off 时，PostgreSQL 服务器日志消息会被原样送达 syslog 服务，而处理可能的大体量消息的任务由 syslog 服务负责。

如果 syslog 最终被记录到一个文本文件中，那么两种设置的效果是一样的，但最好设置为 on，因为大部分 syslog 实现要么不能处理大型消息，要么需要做特殊的配置以处理大型消息。但是如果 syslog 最终写入到某种其他媒介，让消息保持逻辑上的完整性可能是必要的，也可能更有用。

这个参数只能在 postgresql.conf 文件中或通过服务器命令行进行设置。

event_source (string)

当启用了向事件日志记录时，这个参数决定用来标识日志中PostgreSQL 消息的程序名。默认值是 PostgreSQL。这个参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

19.8.2. 什么时候记录日志

log_min_messages (enum)

控制将哪些消息级别写入服务器日志。有效值为 DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL 和 PANIC。每个级别包

括其后的所有级别。级别越高，发送到日志的消息越少。默认值为WARNING。请注意，在 client_min_messages 中，LOG 的排名不同。只有超级用户能更改这个设置。

log_min_error_statement (enum)

控制在服务器日志中记录哪些导致错误条件的 SQL 语句。对于达到指定严重级别或更高级别的消息，其日志条目中会包含当前 SQL 语句。有效值为DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、ERROR、LOG、FATAL 和 PANIC。默认值为 ERROR，这意味着导致错误、日志消息、致命错误或紧急情况的语句将被记录。要有效地关闭记录失败的语句，将此参数设置为PANIC。只有超级用户能更改这个设置。

log_min_duration_statement (integer)

如果一条已完成语句的运行时间至少达到指定毫秒数，就记录其持续时间。将此值设置为零会打印所有语句的持续时间。负一（默认值）禁用语句持续时间记录。例如，如果设置为 250ms，则会记录所有运行 250ms 或更长时间的 SQL 语句。启用此参数有助于发现应用中未优化的查询。只有超级用户能更改这个设置。

对于使用扩展查询协议的客户端，Parse、Bind 和 Execute 步骤的持续时间将被独立记录。

> **注意**
>
> 当把这个选项和 log_statement 一起使用时，已经被log_statement 记录的语句文本不会在持续时间日志消息中重复。如果你没有使用 syslog，我们推荐你使用 log_line_prefix 记录 PID 或会话 ID，这样你可以使用进程 ID 或会话 ID 把语句消息链接到后来的持续时间消息。

表 19.2 解释了 PostgreSQL 所使用的消息严重级别。如果日志输出被发送到 syslog 或 Windows 的 eventlog，严重级别会按照表中所示进行转换。

表 19.2. 消息严重级别

严重性	用法	syslog	eventlog
DEBUG1..DEBUG5	为开发者提供逐级更加详细的信息。	DEBUG	INFORMATION
INFO	提供用户隐式要求的信息，例如来自VACUUM VERBOSE 的输出。	INFO	INFORMATION
NOTICE	提供可能对用户有用的信息，例如长标识符截断提示。	NOTICE	INFORMATION
WARNING	提供可能出现的问题的警告，例如在一个事务块外COMMIT。	NOTICE	WARNING
ERROR	报告一个导致当前命令中断的错误。	WARNING	ERROR
LOG	报告管理员可能感兴趣的信息，例如检查点活动。	INFO	INFORMATION
FATAL	报告一个导致当前会话中断的错误。	ERR	ERROR

严重性	用法	syslog	eventlog
PANIC	报告一个导致所有数据库会话中断的错误。	CRIT	ERROR

19.8.3. 记录哪些内容

> **注意**
>
> 你选择记录的内容可能会影响安全性；请参见第 24.3 节。

application_name (string)

application_name 可以是任意小于 NAMEDATALEN 个字符（标准编译中是 64 个字符）的字符串。应用通常在连接服务器时设置此值。该名称将被显示在 pg_stat_activity 视图中并被包括在 CSV 日志项中。也可以通过 log_line_prefix 将其包括在普通日志项中。只有可打印 ASCII 字符能被使用在 application_name 之中。其他字符将被替换为问号（?）。

debug_print_parse (boolean)
debug_print_rewritten (boolean)
debug_print_plan (boolean)

这些参数将会让多种调试输出被发出。当被设置时，它们为每一个被执行的查询打印结果分析树、查询重写器输出或执行计划。这些消息在LOG 消息级别上被发出，因此默认情况下它们将出现在服务器日志中但不会被发送到客户端。你可以通过调整client_min_messages 和/或 log_min_messages 来改变这种情况。这些参数默认是关闭的。

debug_pretty_print (boolean)

当被设置时，debug_pretty_print 会缩进由 debug_print_parse、debug_print_rewritten 或 debug_print_plan 产生的输出。这将导致比关闭参数时使用的“紧凑”模式可读性更强但是更长的输出。它默认是打开的。

log_checkpoints (boolean)

导致检查点和重启点在服务器日志中记录。日志消息中包括一些统计信息，包括写入的缓冲区数量和写入它们所花费的时间。此参数只能在 postgresql.conf 文件或服务器命令行中设置。默认值为关闭。

log_connections (boolean)

记录每次尝试连接服务器的操作，以及客户端认证的成功完成。只有超级用户可以在会话开始时更改此参数，并且在会话中完全无法更改它。默认值为off。

> **注意**
>
> 某些客户端程序（例如 psql）在判断是否需要密码时会尝试连接两次，因此重复的“收到连接”消息并不一定表示一个错误。

log_disconnections (boolean)

导致会话终止被记录。日志输出提供类似于 log_connections 的信息，以及会话的持续时间。只有超级用户可以在会话开始时更改此参数，而且在会话中根本无法更改。默认值为 off。

log_duration (boolean)

记录每个已完成语句的持续时间。默认值为 off。只有超级用户能更改这个设置。

对于使用扩展查询协议的客户端，Parse、Bind 和 Execute 步骤的持续时间将被独立记录。

> **注意**
>
> 启用这个选项和设置 log_min_duration_statement 为零之间的区别是，超过log_min_duration_statement 指定的时长会强制记录查询文本，而这个选项不会。因此，如果log_duration 为 on 并且 log_min_duration_statement 为正值，所有持续时间都将被记录，但是只有超过阈值的语句才会被记录查询文本。这种行为有助于在高负载安装中收集统计信息。

log_error_verbosity (enum)

控制在服务器日志中记录的每条消息的详细程度。有效值为 TERSE，DEFAULT 和 VERBOSE，它们依次在显示的消息中增加更多字段。TERSE 不包括 DETAIL，HINT，QUERY 和 CONTEXT 错误信息的记录。VERBOSE 输出包括 SQLSTATE 错误代码（另请参见附录 A）以及生成错误的源代码文件名、函数名和行号。只有超级用户能更改这个设置。

log_hostname (boolean)

默认情况下，连接日志消息只显示连接主机的 IP 地址。打开这个参数将导致也记录主机名。注意根据你的主机名解析设置，这可能会导致不可忽视的性能开销。这个参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

log_line_prefix(string)

这是一个 printf 风格的字符串，会输出在每个日志行的开头。% 字符用于引入“转义序列”，它们会被替换为下表所述的状态信息。无法识别的转义会被忽略。其他字符会直接复制到日志行中。有些转义只被会话进程识别，后台进程（例如主服务器进程）会将其视为空。在 % 之后、选项之前指定数值字面量，可以使状态信息左对齐或右对齐。负值会在状态信息的右侧填充空格，使其达到最小宽度；正值则在左侧填充。填充有助于提高日志文件的可读性。此参数只能在postgresql.conf 文件中或在服务器命令行上设置。默认值为'%m [%p] '，它会记录时间戳和进程 ID。

转义	效果	只限会话
%a	应用名	是
%u	用户名	是
%d	数据库名	是
%r	远程主机名或 IP 地址，以及远程端口	是
%h	远程主机名或 IP 地址	是
%p	进程 ID	否
%t	无毫秒的时间戳	否
%m	带毫秒的时间戳	否
%n	带毫秒精度的 Unix 时间戳	否
%i	命令标签：会话当前命令的类型	是
%e	SQLSTATE 错误代码	否
%c	会话 ID：见下文	否
%l	对每个会话或进程的日志行号，从 1 开始	否
%s	进程开始的时间戳	否
%v	虚拟事务 ID (backendID/local XID)	否
%x	事务 ID（如果未分配则为 0）	否
%q	不产生输出，但是告诉非会话进程在字符串的这一点停止；会话进程忽略	否
%%	字面字符 %	否

该 %c 转义会打印一个近乎唯一的会话标识符，由两个以点分隔的 4 字节十六进制数（不含前导零）组成。这两个数分别是进程启动时间和进程 ID，因此%c 也可以用作节省空间的方式来打印这些信息。例如，要从pg_stat_activity 生成会话标识符，可以使用以下查询：

```
SELECT to_hex(trunc(EXTRACT(EPOCH FROM backend_start))::integer) || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

> **提示**
>
> 如果你为 log_line_prefix 设置了非空值，你通常应该让它的最后一个字符为空格，这样用以提供和日志行的剩余部分的视觉区别。也可以使用标点符号。

> **提示**
>
> Syslog 产生自己的时间戳和进程 ID 信息，因此如果你记录到syslog 你可能不希望包括那些转义。

> **提示**
>
> 该 %q 转义在包含只有会话（后端）上下文才提供的信息时很有用，例如用户名或数据库名。比如：
>
> ```
> log_line_prefix = '%m [%p] %q%u@%d/%a '
> ```

log_lock_waits (boolean)

控制会话为获取锁而等待的时间超过 deadlock_timeout 时是否生成日志消息。这对于确定锁等待是否导致性能不佳很有用。默认值为 off。只有超级用户能更改这个设置。

log_statement (enum)

控制哪些 SQL 语句被记录。有效值是 none (off)、ddl、mod 和 all（所有语句）。ddl 记录所有数据定义语句，例如CREATE、ALTER 和 DROP 语句。mod 记录所有 ddl 语句，外加INSERT、UPDATE、DELETE、TRUNCATE 和 COPY FROM 等数据修改语句。如果 PREPARE、EXECUTE 和 EXPLAIN ANALYZE 包含合适类型的命令，它们也会被记录。对于使用扩展查询协议的客户端，当收到一个 Execute 消息时会产生日志并且会包括 Bind 参数的值（任何内嵌的单引号会被双写）。

默认值为 none。只有超级用户能更改这个设置。

> **注意**
>
> 即使使用 log_statement = all 设置，包含简单语法错误的语句也不会被记录。这是因为只有在完成基本语法解析并确定了语句类型之后才会发出日志消息。在扩展查询协议的情况下，在 Execute 阶段之前（即在解析分析或规划期间）出错的语句也不会被记录。将log_min_error_statement 设置为 ERROR（或更低）来记录这种语句。
>
> 记录的语句可能会透露敏感数据，甚至包含明文密码。

log_replication_commands (boolean)

每个复制命令都会被记录在服务器日志中。有关复制命令的更多信息，请参见第 53.4 节。默认值为 off。只有超级用户能更改这个设置。

log_temp_files (integer)

控制临时文件名和大小的日志记录。临时文件可以用于排序、hash 和临时查询结果。每当删除临时文件时都会发出日志记录。值为零时记录所有临时文件信息，而正值仅记录大小大于或等于指定千字节数的文件。默认设置为-1，禁用此类日志记录。只有超级用户能更改这个设置。

log_timezone (string)

设置在服务器日志中写入的时间戳的时区。和 TimeZone 不同，这个值是集簇范围的，因此所有会话将报告一致的时间戳。内置默认值是GMT，但是通常会被在postgresql.conf 中覆盖。initdb 将安装一个对应于其系统环境的设置。详见第 8.5.3 节。这个参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

19.8.4. 使用 CSV 格式的日志输出

将 csvlog 加入 log_destination 列表中，可以方便地将日志文件导入数据库表。此选项以逗号分隔值（CSV）格式输出日志行，包含以下列：带毫秒的时间戳、用户名、数据库名、进程 ID、客户端主机:端口号、会话 ID、会话内行号、命令标签、会话开始时间、虚拟事务 ID、常规事务 ID、错误严重性、SQLSTATE 代码、错误消息、错误消息详情、提示、引发错误的内部查询（如果有）、该内部查询中错误位置的字符数、错误上下文、引发错误的用户查询（如果有且由 log_min_error_statement 启用）、该用户查询中错误位置的字符数、错误在 PostgreSQL 源代码中的位置（如果log_error_verbosity 设置为 verbose）和应用名称。以下是用于存储 CSV 格式日志输出的示例表定义：

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  PRIMARY KEY (session_id, session_line_num)
);
```

使用 COPY FROM 命令将一个日志文件导入到这个表中：

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

也可以作为外部表访问该文件，使用提供的file_fdw 模块。

你可以做一些事情来简化导入 CSV 日志文件：

1. 设置 log_filename 和 log_rotation_age，为日志文件提供一致且可预测的命名方案。这样就能预测文件名，并知道单个日志文件何时已完成写入、可以导入。
2. 将 log_rotation_size 设置为 0 来禁用基于尺寸的日志轮转，因为它使得日志文件名难以预测。

3. 将 log_truncate_on_rotation 设置为 on，这样在同一个文件中旧日志数据不会与新数据混杂。

4. 上述表定义包括一个主键声明。这有助于避免意外地两次导入相同的信息。COPY 命令一次提交所有它导入的数据，因此任何错误将导致整个导入失败。如果你导入一个部分完成的日志文件并且稍后当它完全完成后再次导入，主键冲突将导致导入失败。请等到日志完成且被关闭之后再导入。这个过程也可以避免意外地导入部分完成的行，这种行也将导致COPY 失败。

19.8.5. 进程标题

这些设置控制服务器进程的进程标题如何修改。通常可以通过 ps 等程序查看进程标题，在 Windows 上则可以使用Process Explorer。详情参见第 28.1 节。

cluster_name (string)

设置此集簇所有服务器进程的进程标题中显示的集簇名称。这个名称可以是任何长度少于 NAMEDATALEN 个字符（在标准编译中是 64 字符）的任何字符串。只有可打印的 ASCII 字符能被用在 cluster_name 值中。其他字符将被替换为问号（?）。如果这个参数被设置为空字符串''（也是默认值），将不会显示名称。这个参数只能在服务器启动时设置。

update_process_title (boolean)

启用后，每次服务器接收到新的 SQL 命令时都会更新进程标题。在大多数平台上，默认情况下此设置为on，但在 Windows 上默认为off，因为该平台更新进程标题的开销较大。只有超级用户能更改这个设置。

19.9. 运行时统计数据

19.9.1. 查询和索引统计信息收集器

这些参数控制服务器范围内的统计信息收集功能。启用统计信息收集后，产生的数据可以通过 pg_stat 和 pg_statio 系列系统视图进行访问。更多信息请参阅第 28 章。

track_activities (boolean)

启用对每个会话当前执行命令的信息收集，包括命令开始执行的时间。此参数默认为开启状态。请注意，即使启用了此参数，只有超级用户、具有 pg_read_all_stats 角色权限的角色和拥有被报告会话的用户（包括属于他们具有权限的角色的会话）才能看到此信息，因此不应构成安全风险。只有超级用户能更改这个设置。

track_activity_query_size (integer)

为每个活动会话指定存储当前执行命令的文本所预留的字节数，它们被用于 pg_stat_activity.query 字段。默认值是 1024 字节。这个参数只能在服务器启动时被设置。

track_counts (boolean)

启用对数据库活动的统计信息收集。此参数默认为开启，因为自动清理守护进程需要这些收集到的信息。只有超级用户能更改这个设置。

track_io_timing (boolean)

启用数据库 I/O 调用的计时。默认情况下，此参数处于关闭状态，因为它将重复查询操作系统的当前时间，这可能会在某些平台上造成显著的开销。你可以使用pg_test_timing 工具来测量系统上计时的开销。I/O 计时信息显示在 pg_stat_database 中，也显示在使用 BUFFERS 选项的 EXPLAIN 输出中，并由pg_stat_statements 提供。只有超级用户能更改这个设置。

track_functions (enum)

启用函数调用次数和耗时的跟踪。指定 pl 以仅跟踪过程语言函数，all 以同时跟踪 SQL 和 C 语言函数。默认值为 none，即禁用函数统计跟踪。只有超级用户能更改这个设置。

注意

简单到足以被“内联”到调用查询中的 SQL 语言函数不会被跟踪，而不管这个设置。

stats_temp_directory (string)

设置存放临时统计信息数据的目录。它既可以是相对于数据目录的路径，也可以是绝对路径。默认值是 pg_stat_tmp。将其指向基于 RAM 的文件系统可以减少物理 I/O 需求，并可能提升性能。这个参数只能在postgresql.conf 文件中或服务器命令行上设置。

19.9.2. 统计监控

log_statement_stats (boolean)
log_parser_stats (boolean)
log_planner_stats (boolean)
log_executor_stats (boolean)

对于每个查询，将各自模块的性能统计输出到服务器日志中。这是一个简单的性能分析工具，类似于 Unix getrusage() 操作系统功能。log_statement_stats 报告整个语句的统计信息，而其他选项报告每个模块的统计信息。log_statement_stats 不能与任何单独模块选项一起启用。所有这些选项默认情况下都是禁用的。只有超级用户才能更改这些设置。

19.10. 自动清理

这些设置控制 autovacuum 特性的行为。更多信息见第 24.1.6 节。注意，其中许多设置可以按表覆盖，参见存储参数。

autovacuum (boolean)

控制服务器是否运行自动清理启动器后台进程。默认为开启，不过要使自动清理正常工作，还需要启用 track_counts。该参数只能在 postgresql.conf 文件或服务器命令行中设置，不过，通过更改表存储参数可以为表禁用自动清理。

注意即使该参数被禁用，系统也会在需要防止事务 ID 回卷时发起自动清理进程。详情请见第 24.1.5 节。

log_autovacuum_min_duration (integer)

当自动清理执行的操作运行时间至少达到指定毫秒数时，就会记录该操作。将此设置为零会记录所有自动清理操作。负一（默认值）会禁用记录自动清理操作。例如，如果将其设置为 250ms，则所有运行时间为 250ms 或更长的自动清理和分析都将被记录。此外，当此参数设置为任何非-1 值时，如果由于冲突锁或关系被并发删除而跳过自动清理操作，则会记录消息。启用此参数可帮助跟踪自动清理活动。此参数只能在postgresql.conf 文件或服务器命令行中设置；但可以通过更改表存储参数来覆盖对单个表的设置。

autovacuum_max_workers (integer)

指定任一时刻可能运行的自动清理进程（自动清理启动器除外）的最大数量。默认值为三个。这个参数只能在服务器启动时设置。

autovacuum_naptime (integer)

指定自动清理在任意给定数据库上各次运行之间的最小间隔。在每一轮中后台进程检查数据库并根据需要为数据库中的表发出 VACUUM 和 ANALYZE 命令。延迟以秒为单位。默认值为 1 分钟（1min）。该参数只能在 postgresql.conf 文件或在服务器命令行上设置。

autovacuum_vacuum_threshold (integer)

指定能在一个表上触发 VACUUM 的被更新或被删除元组的最小数量。默认值为 50 个元组。该参数只能在 postgresql.conf 文件或在服务器命令行上设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

autovacuum_analyze_threshold (integer)

指定能在一个表上触发 ANALYZE 的被插入、被更新或被删除元组的最小数量。默认值为 50 个元组。该参数只能在postgresql.conf 文件或在服务器命令行上设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

autovacuum_vacuum_scale_factor (floating point)

指定一个表尺寸的分数，在决定是否触发VACUUM 时将它加到 autovacuum_vacuum_threshold 上。默认值为 0.2（表尺寸的 20%）。该参数只能在 postgresql.conf 文件或在服务器命令行上设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

autovacuum_analyze_scale_factor (floating point)

指定一个表尺寸的分数，在决定是否触发ANALYZE 时将它加到 autovacuum_analyze_threshold 上。默认值为 0.1（表尺寸的 10%）。该参数只能在 postgresql.conf 文件或在服务器命令行上设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

autovacuum_freeze_max_age (integer)

指定在必须强制执行 VACUUM 以防止表内事务 ID 回卷之前，一个表的 pg_class.relfrozenxid 字段可以达到的最大年龄（以事务计）。注意即使自动清理被禁用，系统也会启动自动清理进程来防止回卷。

清理也允许从 pg_xact 子目录中移除旧文件，这也是为什么默认值被设置为较低的 2 亿事务。该参数只能在服务器启动时设置，但是对于个别表可以通过修改表存储参数来降低该设置。详见第 24.1.5 节。

autovacuum_multixact_freeze_max_age (integer)

指定在必须强制执行 VACUUM 以防止表内多事务 ID 回卷之前，一个表的 pg_class .relminmxid 字段可以达到的最大年龄（以多事务计）。注意即使自动清理被禁用，系统也会启动自动清理进程来防止回卷。

清理多事务也允许从 pg_multixact/members 和 pg_multixact/offsets 子目录中移除旧文件，这也是为什么默认值被设置为较低的 4 亿个多事务。该参数只能在服务器启动时设置，但是对于个别表可以通过修改表存储参数来降低该设置。详见第 24.1.5.1 节。

autovacuum_vacuum_cost_delay (integer)

指定用于自动 VACUUM 操作中的代价延迟值。如果指定-1，则使用vacuum_cost_delay 值。默认值为 20 毫秒。该参数只能在 postgresql.conf 文件或在服务器命令行上设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

autovacuum_vacuum_cost_limit (integer)

指定用于自动 VACUUM 操作中的代价限制值。如果指定-1（默认值），则使用 vacuum_cost_limit 值。注意该值被按比例地分配到运行中的自动清理工作进程上（如果有多个），因此每一个工作进程的限制值之和不会超过这个变量中的值。该参数只能在 postgresql.conf 文件或在服务器命令行上设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

19.11. 客户端连接默认值

19.11.1. 语句行为

client_min_messages (enum)

控制被发送给客户端的消息级别。有效值是 DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、LOG、NOTICE、WARNING、ERROR。每个级别都包括其后的所有级别。级别越靠后，被发送的消息越少。默认值是NOTICE。注意 LOG 在这里的排序与 log_min_messages 中的不同。

INFO 级别的消息总是被发送到客户端。

search_path (string)

这个变量指定当一个对象（表、数据类型、函数等）被用一个无模式限定的简单名称引用时，搜索该对象时的模式顺序。当在不同模式中有同名对象时，将使用第一个在搜索路径中被找到的对象。一个不属于搜索路径中任何一个模式的对象只能通过用限定名（带点号）指定包含它的模式来引用。

search_path 的值必须是一个逗号分隔的模式名列表。任何不是一个已有模式的名称，或者是一个用户不具有USAGE 权限的模式，将被静默忽略。

如果列表项之一是特殊名 $user，则具有CURRENT_USER 返回的名字的模式将取代它（如果有这样一个模式并且该用户有该模式的 USAGE 权限；如果没有，$user 会被忽略）。

系统目录模式 pg_catalog 总是被搜索，不管它是否在搜索路径中被提及。如果它在路径中被提及，那么它将被按照路径指定的顺序搜索。如果pg_catalog 不在路径中，则它将在任何路径项之前被搜索。

同样，如果存在，当前会话的临时表模式pg_temp_*nnn* 总是被搜索。可以通过使用别名 pg_temp 在路径中显式列出它。如果它没有列在路径中，则首先搜索它（甚至在 pg_catalog 之前）。然而，临时模式仅用于搜索关系（表、视图、序列等）和数据类型名称。它永远不会用于搜索函数或操作符名称。

当对象创建时没有指定一个特定目标模式，它们将被放置在search_path 中第一个合法模式中。如果搜索路径为空将报告一个错误。

这个参数的默认值是"$user", public。这种设置支持共享使用数据库（用户没有私有模式，所有人共享使用public）、每个用户拥有私有模式，以及二者的组合。还可以通过全局或针对每个用户修改默认搜索路径设置来获得其他效果。

更多有关模式处理的信息，请参考第 5.8 节。特别地，只有当数据库只有一个用户或者有少数的相互信任的用户时，默认配置是合适的。

搜索路径的当前有效值可以通过 SQL 函数 current_schemas 检查（见第 9.25 节）。它和检查 search_path 的值不太一样，因为current_schemas 显示出现在 search_path 中的项如何被解析。

row_security (boolean)

这个变量控制是否以抛出一个错误来代替应用一条行安全性策略。在设置为 on 时，策略正常应用。在设置为off 时，原本会应用至少一条策略的查询就会失败。默认为on。受限的行可见性可能导致不正确的结果时，可将其改成off。例如，pg_dump 默认会做这种更改。这个变量对能绕过每一条行安全性策略的角色（即超级用户和具有 BYPASSRLS 属性的角色）没有效果。

更多关于行安全性策略的信息请见 CREATE POLICY。

default_tablespace (string)

这个变量指定当一个 CREATE 命令没有显式指定一个表空间时，创建对象（表和索引）的默认表空间。

该值要么是一个表空间的名字，要么是一个指定使用当前数据库默认表空间的空字符串。如果该值和任何现有表空间的名字都不匹配，PostgreSQL 将自动使用当前数据库的默认表空间。如果指定了一个非默认的表空间，用户必须对它有CREATE 权限，否则创建尝试将失败。

这个变量不被用于临时表，对临时表会使用 temp_tablespaces。

创建数据库时也不会使用这个变量。默认情况下，一个新数据库会从它的模板数据库继承其表空间设置。

有关表空间的更多的信息，请见第 22.6 节。

temp_tablespaces (string)

这个变量指定当一个 CREATE 命令没有显式指定一个表空间时，创建临时对象（临时表和临时表上的索引）的默认表空间。用于排序大型数据集的临时文件也被创建在这些表空间中。

该值是一个表空间名字的列表。当列表中有多于一个名称时，每次一个临时对象被创建时 PostgreSQL 随机选择列表中的一个成员。例外是在一个事务中，连续创建的临时对象被依次放置在列表中的连续表空间中。如果列表中被选中的元素是一个空字符串，PostgreSQL 将自动使用当前数据库的默认表空间。

当 temp_tablespaces 被交互式地设置时，指定一个不存在的表空间是一种错误，指定一个用户不具有CREATE 权限的表空间也同样是错误。不过，当使用一个之前设置的值时，不存在的表空间会被忽略，就像用户缺少CREATE 权限的表空间一样。特别是，使用一个在 postgresql.conf 中设置的值时，这条规则起效。

默认值是一个空字符串，它使得所有临时对象被创建在当前数据库的默认表空间中。

参阅 default_tablespace。

check_function_bodies (boolean)

这个参数通常为打开。当设置为 off 时，它禁用CREATE FUNCTION 期间对函数体字符串的验证。禁用验证避免了验证处理的副作用，也避免前向引用等问题导致的误报。在代表其他用户载入函数之前设置这个参数为off；pg_dump 会自动这样做。

default_transaction_isolation (enum)

每个 SQL 事务都有一个隔离级别，可以是“读未提交”、“读已提交”、“可重复读”或者“可串行化”。这个参数控制每个新事务的默认隔离级别。默认是“读已提交”。

更多信息请参阅第 13 章和 SET TRANSACTION。

default_transaction_read_only (boolean)

一个只读的 SQL 事务不能修改非临时表。这个参数控制每个新事务的默认只读状态。默认是 off（读/写）。

更多信息请参考 SET TRANSACTION。

default_transaction_deferrable (boolean)

当运行在 serializable 隔离级别时，一个可延迟只读 SQL 事务可能在获准继续之前被延迟一段时间。但是，一旦它开始执行就不会产生任何用来保证可串行性的开销；因此串行化代码将没有任何理由因为并发更新而强制它中止，使得这个选项适合于长时间运行的只读事务。

这个参数控制每个新事务的默认可延迟状态。目前它对读写事务或者那些运行在低于 serializable 隔离级别上的事务无效。默认值是 off。

更多信息请参考 SET TRANSACTION。

transaction_isolation (enum)

此参数反映当前事务的隔离级别。在每个事务开始时，它被设置为 default_transaction_isolation 的当前值。任何后续更改它的尝试都相当于 SET TRANSACTION 命令。

transaction_read_only (boolean)

此参数反映当前事务的只读状态。在每个事务的开始，它被设置为 default_transaction_read_only 的当前值。任何后续更改它的尝试都等同于 SET TRANSACTION 命令。

transaction_deferrable (boolean)

此参数反映当前事务的可延迟性状态。在每个事务的开始，它被设置为 default_transaction_deferrable 的当前值。任何后续更改它的尝试都等同于 SET TRANSACTION 命令。

session_replication_role (enum)

控制当前会话中复制相关触发器和规则的触发。设置此变量需要超级用户权限，并会丢弃任何先前缓存的查询计划。可用值为origin（默认值）、replica 和 local。

这个设置的预期用途是由逻辑复制系统在应用所复制的更改时将它设置为 replica。其效果将是触发器和规则（没有对其默认配置做修改）在副本上将不会被触发。更多信息请参考 ALTER TABLE 的子句 ENABLE TRIGGER 以及 ENABLE RULE。

PostgreSQL 在内部会把设置 origin 和 local 同样对待。第三方复制系统可能会把这两个值用于其内部目的，例如把local 用来标出一个不应复制其更改的会话。

因为外键被实现为触发器，将这个参数设置为replica 还会禁用所有的外键检查，如果使用不当可能会让数据处于一种不一致的状态。

statement_timeout (integer)

中止任何使用了超过指定毫秒数的语句，从客户端命令到达服务器时开始计时。如果log_min_error_statement 被设置为 ERROR 或更低，语句如果超时也会被记录。一个零值（默认）将禁用超时。

我们不推荐在 postgresql.conf 中设置 statement_timeout，因为它会影响所有会话。

lock_timeout (integer)

如果任何语句在试图获取表、索引、行或其他数据库对象上的锁时等待超过指定的毫秒数，该语句将被中止。该时间限制独立地应用于每一次锁获取尝试。该限制会应用到显式锁定请求（如LOCK TABLE 或不带 NOWAIT 的 SELECT FOR UPDATE）和隐式获得的锁。一个零值（默认）将禁用超时。

与 statement_timeout 不同，这个超时只在等待锁时发生。注意如果 statement_timeout 为非零，设置lock_timeout 为相同或更大的值没有意义，因为语句超时将总是第一个被触发。如果log_min_error_statement 被设置为 ERROR 或更低，超时的语句将被记录。

我们不推荐在 postgresql.conf 中设置 lock_timeout，因为它会影响所有会话。

idle_in_transaction_session_timeout (integer)

终止任何具有打开事务、且闲置时间超过指定的毫秒数的会话。这使得该会话持有的锁能够被释放，连接槽能够被重用，也使得仅对此事务可见的元组能够被清理。更多详情参见第 24.1 节。

默认值 0 禁用此特性。

vacuum_freeze_table_age (integer)

如果表的 pg_class.relfrozenxid 字段达到此设置指定的年龄，VACUUM 就会执行一次激进扫描。激进扫描与常规 VACUUM 不同，它会访问每一个可能包含未冻结 XID

或 MXID 的页面，而不仅仅是那些可能包含死元组的页面。默认值为 1.5 亿个事务。尽管用户可以将该值设置在 0 到 20 亿之间，VACUUM 仍会悄悄将其有效值限制为不超过 autovacuum_freeze_max_age 的 95%，以便在针对该表启动防回卷自动清理之前，周期性手工VACUUM 仍有机会运行。详见第 24.1.5 节。

vacuum_freeze_min_age (integer)

指定 VACUUM 在扫描表时用来决定是否冻结行版本的截止年龄（以事务计）。默认值是 5000 万个事务。尽管用户可以将该值设置为 0 到 10 亿之间的任意值，VACUUM 会悄悄将有效值限制为不超过 autovacuum_freeze_max_age 的一半，这样强制 autovacuum 之间就不会间隔过短。更多信息请参见第 24.1.5 节。

vacuum_multixact_freeze_table_age (integer)

如果表的 pg_class.relminmxid 字段达到此设置指定的年龄，VACUUM 就会执行一次激进扫描。激进扫描与常规 VACUUM 不同，它会访问每一个可能包含未冻结 XID 或 MXID 的页面，而不仅仅是那些可能包含死元组的页面。默认值为 1.5 亿个多事务。尽管用户可以将该值设置在 0 到 20 亿之间，VACUUM 仍会悄悄将其有效值限制为不超过 autovacuum_multixact_freeze_max_age 的 95%，以便在针对该表启动防回卷清理之前，周期性手工VACUUM 仍有机会运行。详见第 24.1.5.1 节。

vacuum_multixact_freeze_min_age (integer)

指定 VACUUM 在扫描表时用来决定是否将多事务 ID 替换为较新的事务 ID 或多事务 ID 的截止年龄（以多事务计）。默认值是 500 万个多事务。尽管用户可以将该值设置为 0 到 10 亿之间的任意值，VACUUM 会悄悄将有效值限制为不超过 autovacuum_multixact_freeze_max_age 的一半，这样强制 autovacuum 之间就不会间隔过短。更多信息请参见第 24.1.5.1 节。

vacuum_cleanup_index_scale_factor (floating point)

指定可插入的元组数占上次统计信息收集时计数的堆元组总数的比例，在此范围内不会导致 VACUUM 在收尾清理阶段扫描索引。此设置当前仅适用于 B-树索引。

即使没有从堆中删除元组，只要满足以下任一条件，VACUUM 仍会在收尾清理阶段扫描 B-树索引：索引统计信息已过时，或者索引包含可在收尾清理期间回收的已删除页面。如果新插入的元组数超过上次统计信息收集时检测到的堆元组总数的vacuum_cleanup_index_scale_factor 比例，则认为索引统计信息已过时。堆元组总数存储在索引元页面中。请注意，只有在VACUUM 未发现死元组之后，元页面才会包含此数据，因此，只有第二次及后续 VACUUM 周期未检测到死元组时，才能跳过收尾清理阶段的 B-树索引扫描。

此值的范围是 0 到 10000000000。当 vacuum_cleanup_index_scale_factor 设置为 0 时，VACUUM 在收尾清理期间永远不会跳过索引扫描。默认值为 0.1。

bytea_output (enum)

设置 bytea 类型值的输出格式。有效值是 hex（默认）和 escape（传统的 PostgreSQL 格式）。详见第 8.4 节。不管这个设置的值如何，bytea 类型总是接受这两种格式的输入。

xmlbinary (enum)

设置二进制值如何被编码为 XML。例如，这适用于通过xmlelement 函数或 xmlforest 函数将 bytea 值转换到 XML 值。可能的值有 base64 和 hex，它们都是用 XML 模式标准定义的。默认值是base64。更多关于 XML 相关函数的信息可参阅第 9.14 节。

这里的实际选择主要取决于偏好，只受客户端应用中可能存在的限制的约束。两种方法都支持所有可能的值，尽管十六进制编码会比 base64 编码略大。

`xmloption` (`enum`)

设置在 XML 与字符串值之间进行转换时，隐含采用`DOCUMENT` 还是 `CONTENT`。有关说明参见第 8.13 节。有效值是 `DOCUMENT` 和 `CONTENT`。默认值是 `CONTENT`。

根据 SQL 标准，设置这个选项的命令是：

```
SET XML OPTION { DOCUMENT | CONTENT };
```

这种语法在 PostgreSQL 也可用。

`gin_pending_list_limit` (`integer`)

设置 `fastupdate` 被启用时可以使用的 GIN 索引的待处理列表的最大尺寸。如果该列表增长到超过这个最大尺寸，会通过批量将其中的项移入索引的主 GIN 数据结构来清理列表。默认值是四兆字节（`4MB`）。可以通过更改索引的存储参数来为个别 GIN 索引覆盖这个设置。更多信息请见第 66.4.1 节和第 66.5 节。

19.11.2. 区域设置和格式化

`DateStyle` (`string`)

设置日期和时间值的显示格式，以及解释有歧义的日期输入值的规则。由于历史原因，这个变量包含两个独立的部分：输出格式声明（`ISO`、`Postgres`、`SQL` 或 `German`）、输入/输出的年/月/日顺序（`DMY`、`MDY` 或 `YMD`）。这些可以被独立设置或者一起设置。关键字 `Euro` 和 `European` 是 `DMY` 的同义词；关键字 `US`、`NonEuro` 和 `NonEuropean` 是 `MDY` 的同义词。详见第 8.5 节。内置默认值是 `ISO`, `MDY`，但是initdb 将用对应于选中的 `lc_time` 区域设置行为的设置初始化配置文件。

`IntervalStyle` (`enum`)

设置时间间隔值的显示格式。值 `sql_standard` 会生成符合 SQL 标准时间间隔字面量的输出。值 `postgres`（默认值）的输出与 PostgreSQL 8.4 之前版本中 DateStyle 设为 `ISO` 时的输出一致。值 `postgres_verbose` 的输出与 PostgreSQL 8.4 之前版本中 `DateStyle` 设为非 `ISO` 输出时的输出一致。值 `iso_8601` 会生成符合 ISO 8601 第 4.4.3.2 节定义的时间间隔“带标志符格式”的输出。

`IntervalStyle` 参数也会影响对有歧义的时间间隔输入的解释。详见第 8.5.4 节。

`TimeZone` (`string`)

设置用于显示和解释时间戳的时区。内置默认值是 `GMT`，但是它通常会在 `postgresql.conf` 中被覆盖；initdb 将安装一个对应于其系统环境的设置。详见第 8.5.3 节。

`timezone_abbreviations` (`string`)

设置服务器接受的日期时间输入中使用的时区缩写集合。默认值为`'Default'`，这个集合在全世界大多数地方都能工作。也还有`'Australia'` 和`'India'`，以及可能为一种特定安装定义的其他集合。详见第 B.4 节。

extra_float_digits (integer)

此参数调整浮点值显示的位数，包括float4、float8 和几何数据类型。参数值会加到标准位数（按情况使用 FLT_DIG 或 DBL_DIG）上。此值最高可以设置为 3，以包含部分有效的数位；这对于转储需要精确还原的浮点数据特别有用。也可以设置为负数，以抑制不需要的数位。另见第 8.1.3 节。

client_encoding (string)

设置客户端编码（字符集）。默认使用数据库编码。PostgreSQL 服务器所支持的字符集在第 23.3.1 节中描述。

lc_messages (string)

设置消息显示的语言。可接受的值是系统相关的；详见第 23.1 节。如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

在一些系统上，这个区域设置类别并不存在。仍然可以设置这个变量，只是不会有任何效果。同样，所期望语言的翻译消息也可能不存在。在这种情况下，你将仍然继续看到英文消息。

只有超级用户能更改这个设置，因为它会同时影响发送到服务器日志和客户端的消息，设置不当可能降低服务器日志的可读性。

lc_monetary (string)

设置用于格式化货币量的区域设置，例如用to_char 函数族。可接受的值是系统相关的；详见第 23.1 节。如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

lc_numeric (string)

设置用于格式化数字的区域设置，例如用to_char 函数族。可接受的值是系统相关的；详见第 23.1 节。如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

lc_time (string)

设置用于格式化日期和时间的区域设置，例如用to_char 函数族。可接受的值是系统相关的；详见第 23.1 节。如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

default_text_search_config (string)

选择被那些没有显式参数指定配置的文本检索函数变体使用的文本检索配置。详见第 12 章。内置默认值是 pg_catalog.simple，但是如果能够标识一个匹配该区域设置的配置，initdb 将用对应于选中的 lc_ctype 区域设置的值初始化配置文件。

19.11.3. 共享库预载入

为了载入附加的功能或者达到提高性能的目的，可用多个设置来预先载入共享库到服务器中。例如'$libdir/mylib' 设置会使 mylib.so（或者某些平台上的 mylib.sl）从安装的标准库目录被预装载。这些设置之间的区别在于生效的时间以及改变它们所需的权限。

可以用这个方法预装载 PostgreSQL 的过程语言库，通常是使用'$libdir/plXXX' 语法，其中的XXX 是 pgsql、perl、tcl 或 python。

只有特别为与 PostgreSQL 一起使用设计的共享库才能以这种方式载入。每一个 PostgreSQL 支持的库都有一个“魔法块”，它会被检查以保证兼容性。由于这个原因，非 PostgreSQL 库无法以这种方式被载入。你可能可以使用操作系统的工具（如LD_PRELOAD）载入它。

一般来说，请参考特定模块的文档来用推荐的方法载入它。

local_preload_libraries (string)

这个变量指定一个或者多个要在连接开始时预载入的共享库。它包含一个由逗号分隔的库名列表，其中每个名称都会按LOAD 命令的方式解析。项之间的空格会被忽略，如果需要在库名中包含空格或者逗号，请把库名放在双引号内。这个参数值只在连接开始时生效。后续的更改不会有任何效果。如果一个指定的库没有找到，连接尝试将会失败。

任何用户都能设置这个选项。正因为如此，能被这样载入的库被严格限制为出现于安装的标准库目录中plugins 子目录下的共享库（保证只有“安全的”库被安装到这里是数据库管理员的责任）。local_preload_libraries 中的项可以显式指定这个目录，例如$libdir/plugins/mylib，或者只是指定库的名称 – mylib 和 $libdir/plugins/mylib 的效果是相同的。

这个特性的目的是允许非特权用户在特定的会话中载入用于调试或性能测量的库，而无需一个显式的LOAD 命令。为了这个目的，通常通过使用客户端的PGOPTIONS 环境变量或者 ALTER ROLE SET 来设置这个参数。

不过，除非一个模块被特别设计成由非超级用户以这种方式使用，通常不推荐使用这个设置。应该看看 session_preload_libraries。

session_preload_libraries (string)

这个变量指定在连接开始时要预加载的一个或多个共享库。它包含一个逗号分隔的库名称列表，其中每个名称的解释方式与LOAD 命令相同。条目之间的空格会被忽略；如果需要在名称中包含空格或逗号，请用双引号括起库名称。参数值仅在连接开始时生效。后续更改不会生效。如果指定的库未找到，连接尝试将失败。只有超级用户能更改这个设置。

这个特性的意图是允许在特定会话中载入调试用的或者测量性能的库，而不需要显式地给出一个 LOAD 命令。例如，通过用ALTER ROLE SET 设置这个参数可以为一个给定用户名下的所有会话启用 auto_explain。还有，无需重启服务器就能更改这个参数（但是只有新会话启动时才会生效），这样可以以这种方式更容易地增加新模块，即便它们会应用到所有会话。

和 shared_preload_libraries 不同，相对于在库被第一次使用时载入它，在会话开始时载入库并没有明显的性能优势。不过，当使用连接池时这样做还是有一些优势。

shared_preload_libraries (string)

这个变量指定一个或者多个要在服务器启动时预载入的共享库。它包含一个由逗号分隔的库名列表，其中每个名称都会按LOAD 命令的方式解析。项之间的空格会被忽略，如果需要在库名中包含空格或者逗号，请把库名放在双引号内。这个参数只能在服务器启动时设置。如果指定的库没有找到，服务器将无法启动。

有些库需要执行只能在 postmaster 启动时发生的特定操作，例如分配共享内存、保留轻量级锁或者启动后台工作者。这些库必须通过这个参数在服务器启动时载入。每个库的详情请见文档。

其他库也能被预载入。通过预载入一个共享库，当该库被第一次使用时就可以避免库的启动时间。不过，启动每个新服务器进程的时间可能会略有增加，即使该进程从不使用该库。因此，推荐只把这个参数用于那些要在大多数会话中使用的库上。还有，改变

这个参数要求重启服务器，因此对于短期的调试任务来说这不是好的选择，应该转用 session_preload_libraries。

> **注意**
>
> 在 Windows 主机上，在服务器启动时预载入一个库并不会减少启动每个新服务器进程所需的时间；每一个服务器进程将会重新载入所有预载入的库。不过，对于那些要在 postmaster 启动时执行操作的库来说，Windows 主机上的 `shared_preload_libraries` 仍然有用。

`jit_provider` (`string`)

这个变量是要被使用的 JIT 提供者库的名称（见第 32.4.2 节）。默认是 `llvmjit`。这个参数只能在服务器启动时设置。

如果这个变量被设置为一个不存在的库，JIT 将不可用，但是也不会发生错误。这种特性允许在主PostgreSQL 包之外单独安装 JIT 支持。

19.11.4. 其他默认值

`dynamic_library_path` (`string`)

如果需要打开一个可以动态装载的模块并且在 `CREATE FUNCTION` 或 `LOAD` 命令中指定的文件名没有目录部分（即名字中不包含斜线），那么系统将搜索这个路径以查找所需的文件。

参数 `dynamic_library_path` 的值必须是由冒号（Windows 上为分号）分隔的绝对目录路径列表。如果某个列表元素以特殊字符串 `$libdir` 开头，则会使用编译时确定的 PostgreSQL 软件包的库目录来替换 `$libdir`；该目录是标准 PostgreSQL 发行版所提供模块的安装位置。（使用 `pg_config --pkglibdir` 可以找出此目录的名称。）例如：

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

或者，在 Windows 环境中：

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

这个参数的默认值是`'$libdir'`。如果该值被设置为一个空字符串，则关闭自动路径搜索。

这个参数可以由超级用户在运行时更改，但以这种方式进行的设置只会持续到客户端连接结束，因此这种方法应该保留用于开发目的。推荐设置此参数的方法是在`postgresql.conf` 配置文件中。

`gin_fuzzy_search_limit` (`integer`)

GIN 索引扫描返回的集合尺寸的软上限。详见第 66.5 节。

19.12. 锁管理

deadlock_timeout (integer)

指定在检查是否发生死锁之前等待锁的毫秒数。检查死锁相对昂贵，因此服务器不会每次等待锁时都运行它。我们乐观地假设在生产应用程序中死锁并不常见，所以在检查死锁之前只是等待一段时间。增加此值会减少在不必要的死锁检查中浪费的时间，但会减慢实际死锁错误的报告速度。默认值为一秒（1s），这可能是你在实践中想要的最小值。在负载较重的服务器上，你可能希望提高它。理想情况下，设置应超过你的典型事务时间，以提高在等待者决定检查死锁之前释放锁的几率。只有超级用户能更改这个设置。

当 log_lock_waits 被设置时，这个参数也决定发出关于锁等待的日志之前等待的时间量。如果你想调查锁延迟，你可能希望设置一个比正常的deadlock_timeout 小的值。

max_locks_per_transaction (integer)

共享锁表跟踪在 max_locks_per_transaction * (max_connections + max_prepared_transactions) 个对象（如表）上的锁。因此，在任何一个时刻，只有不超过这么多个可区分对象能够被锁住。这个参数控制为每个事务分配的对象锁的平均数量。只要所有事务的锁都能放入锁表中，单个事务就可以锁住更多对象。这不是能被锁住的行数，那个值是没有限制的。默认值 64 已经被历史证明是足够的，但是如果你有需要在一个事务中使用很多不同表的查询（例如查询一个有很多子表的父表），你可能需要提高这个值。这个参数只能在服务器启动时设置。

运行备库时，必须将此参数设置为大于或等于主库上的值。否则，备库上将不允许查询。

max_pred_locks_per_transaction (integer)

共享谓词锁表跟踪在 max_pred_locks_per_transaction * (max_connections + max_prepared_transactions) 个对象（如表）上的锁。因此，在任何一个时刻，只有不超过这么多个可区分对象能够被锁住。这个参数控制为每个事务分配的对象锁的平均数量。只要所有事务的锁都能放入锁表中，单个事务就可以锁住更多对象。这不是能被锁住的行数，那个值是没有限制的。默认值 64 已经在测试中被证明通常是足够的，但是如果你有会在单个可串行化事务中访问许多不同表的客户端，则可能需要提高这个值。这个参数只能在服务器启动时设置。

max_pred_locks_per_relation (integer)

这个参数控制在谓词锁被提升为覆盖整个关系之前，可以对单个关系中的多少页面或元组加谓词锁。大于等于零的值表示一种绝对限制，而负值表示用 max_pred_locks_per_transaction 除以这个设置的绝对值。默认值为-2，它将保持以往版本的PostgreSQL 中的行为。这个参数只能在 postgresql.conf 文件中或者服务器命令行上设置。

max_pred_locks_per_page (integer)

这个参数控制在谓词锁被提升为覆盖整个页面之前，可以对单个页面中的多少行加谓词锁。默认值是 2。这个参数只能在postgresql.conf 文件中或者服务器命令行上设置。

19.13. 版本和平台兼容性

19.13.1. 以前的 PostgreSQL 版本

`array_nulls` (`boolean`)

这个参数控制数组输入解析器是否把未用引号的 `NULL` 识别为一个值为空值的数组元素。默认为 `on`，允许输入包含空值的数组值。但是PostgreSQL 8.2 之前的版本不支持数组中的空值，并且因此将把`NULL` 当作指定一个值为字符串 “NULL” 的正常数组元素。为了向后兼容那些要求旧行为的应用，这个变量可以被设置为`off`。

注意即使这个变量为 `off` 也能够创建包含空值的数组值。

`backslash_quote` (`enum`)

这个参数控制字符串字面量中的单引号是否能够用`\'` 来表示。首选的 SQL 标准的方法是将其双写（`''`），但是PostgreSQL 在历史上也接受`\'`。不过使用`\'` 容易导致安全风险，因为在某些客户端字符集编码中，有多字节字符的最后一个字节在数值上等价于 ASCII 的`\`。如果客户端代码没有做到正确转义，那么就可能遭到 SQL 注入攻击。如果服务器拒绝看起来带有被反斜线转义的单引号的查询，那么就可以避免这种风险。`backslash_quote` 的可用值是`on`（总是允许`\'`）、`off`（总是拒绝）以及 `safe_encoding`（只有客户端编码不允许在多字节字符中存在 ASCII `\` 时允许）。`safe_encoding` 是默认设置。

注意在符合标准的字符串字面量中，`\` 就表示`\`。这个参数只影响不符合标准的字面量的处理，包括转义字符串语法（`E'...'`）。

`default_with_oids` (`boolean`)

此参数控制在既未指定 `WITH OIDS` 也未指定 `WITHOUT OIDS` 时，`CREATE TABLE` 和 `CREATE TABLE AS` 是否在新建表中包含 OID 列。它也决定 `SELECT INTO` 创建的表是否包含 OID。此参数默认为 `off`；在 PostgreSQL 8.0 及更早版本中，默认为`on`。

在用户表中使用 OID 已被弃用，因此大多数安装应让此变量保持禁用状态。需要为特定表使用 OID 的应用应在创建表时指定`WITH OIDS`。可以启用此变量，以兼容不遵循这种做法的旧应用。

`escape_string_warning` (`boolean`)

打开时，如果在普通字符串字面量中（`'...'` 语法）出现了一个反斜线（`\`）并且 `standard_conforming_strings` 为关闭，那么就会发出一个警告。默认值是`on`。

希望使用反斜线作为转义符的应用应该被修改来使用转义字符串语法（`E'...'`），因为按照 SQL 标准，普通字符串现在默认将反斜线视作一个普通字符。这个变量可以被启用来帮助定位需要被更改的代码。

`lo_compat_privileges` (`boolean`)

在 PostgreSQL 9.0 之前的版本中，大对象没有访问权限，因此始终可以被所有用户读取和写入。将此变量设置为`on` 会禁用新的权限检查，以保持与之前版本的兼容性。默认值为`off`。只有超级用户能更改这个设置。

设置此变量不会禁用所有与大对象相关的安全检查 – 只禁用那些在 PostgreSQL 9.0 中默认行为发生变化的检查。

operator_precedence_warning (boolean)

开启时，对于任何可能因操作符优先级的变化而在PostgreSQL 9.4 之后改变含义的构造，解析器都会发出警告。这有助于审查应用，检查优先级变化是否破坏了某些行为；但不应在生产环境中一直开启，因为它也会对一些完全有效、符合标准的 SQL 代码发出警告。默认值为 off。

更多信息参见第 4.1.6 节。

quote_all_identifiers (boolean)

当数据库产生 SQL 时，强制所有标识符被引号包围，即使它们（当前）不是关键字。这将影响EXPLAIN 的输出以及 pg_get_viewdef 等函数的结果。另请参阅 pg_dump 和 pg_dumpall 的 --quote-all-identifiers 选项。

standard_conforming_strings (boolean)

控制普通字符串字面量（'...'）是否按照 SQL 标准把反斜线当普通文本。从 PostgreSQL 9.1 开始，默认值为on（之前的发行中默认值为 off）。应用可以检查这个参数来判断字符串字面量如何被处理。这个参数的存在也可以被当做转义字符串语法（E'...'）被支持的标志。如果一个应用希望反斜线被当做转义字符，应该使用转义字符串语法（第 4.1.2.2 节）。

synchronize_seqscans (boolean)

它允许对大型表的顺序扫描与其他扫描同步，因此并发扫描可以在几乎相同的时刻读取相同的块，这样可以分担 I/O 负载。当启用这个参数时，一个扫描可能会从表的中间开始并且之后“绕回”到开头以覆盖所有的行，这样可以与已在进行中的扫描活动同步。对于没有 ORDER BY 子句的查询，这样的扫描可能会在返回行的顺序中造成不可预料的改变。将这个参数设置为off 以保证 8.3 之前的行为（顺序扫描总是从表的起始处开始）。默认值是 on。

19.13.2. 平台和客户端兼容性

transform_null_equals (boolean)

当打开时，形如*expr* = NULL（或 NULL = *expr*）的表达式将被当做 *expr* IS NULL，也就是说，如果*expr* 计算结果为空值则返回真，否则返回假。正确的 SQL 标准兼容的*expr* = NULL 行为总是返回空值（未知）。因此这个参数默认为 off。

不过，在Microsoft Access 里的过滤表单生成的查询似乎使用 *expr* = NULL 来测试空值，因此，如果你使用这个接口访问数据库，你可能想把这个选项打开。因为*expr* = NULL 形式的表达式总是返回空值（使用 SQL 标准解释），它们不是非常有用并且在普通应用中也不常见，因此这个选项实际上没有什么危害。但是新用户常常对涉及空值的表达式语义感到困惑，因此这个选项默认为关闭。

请注意这个选项只影响= NULL 形式，而不影响其它比较操作符或者其它与一些涉及等值操作符的表达式在计算上等效的其他表达式（例如IN）。因此，这个选项不能普遍修复错误的程序写法。

相关信息请见第 9.2 节。

19.14. 错误处理

exit_on_error (boolean)

如果为 true，任何错误将中止当前会话。默认情况下，这个值被设置为 false，这样只有 FATAL 错误（致命）将中止会话。

restart_after_crash (boolean)

当被设置为 true（默认值），PostgreSQL 将在一次后端崩溃后自动重新初始化。让这个值设置为 true 通常是将数据库可用性最大化的最佳方法。但是在某些环境中，例如 PostgreSQL 被集群软件调用时，禁用重启可能很有用，这样集群软件可以得到控制并且采取它认为适当的行动。

这个参数只能在 postgresql.conf 文件中或通过服务器命令行进行设置。

data_sync_retry (boolean)

如果设置为 false（默认值），PostgreSQL 在将修改后的数据文件刷盘到文件系统失败时，将引发 PANIC 级错误。这样会导致数据库服务器崩溃。这个参数只能在服务器启动时被设置。

在某些操作系统上，回写失败后，内核页面缓存中的数据状态未知。在某些情况下，它可能已被完全遗忘，因此重试不安全；第二次尝试可能报告为成功，而事实上数据已丢失。在此类情形下，避免数据丢失的唯一方法是在报告任何故障后从 WAL 中恢复，最好是在调查了故障的根本原因并更换了任何有故障的硬件之后。

如果设置为 true，PostgreSQL 将报告错误，但会继续运行，以便可以在以后的检查点中重试数据刷盘操作。只有在调查清楚操作系统在回写失败时如何处理缓冲数据之后，才应将其设置为 true。

19.15. 预置选项

下列“参数”是只读的，在编译或安装 PostgreSQL 时确定。因此，它们未列入示例 postgresql.conf 文件。这些选项报告 PostgreSQL 行为的各个方面，某些应用（特别是管理前端）可能对此感兴趣。

block_size (integer)

报告一个磁盘块的大小。它由编译服务器时 BLCKSZ 的值确定。默认值是 8192 字节。有些配置变量的含义（例如 shared_buffers）会被 block_size 影响。详见第 19.4 节。

data_checksums (boolean)

报告对这个集簇是否启用了数据校验和。详见数据校验和。

data_directory_mode (integer)

在 Unix 系统上，这个参数报告服务器启动时数据目录（由data_directory 定义）的权限。（在 Microsoft Windows 上这个参数将总是显示 0700。）更多信息请参考组访问。

`debug_assertions` (`boolean`)

报告 PostgreSQL 是否在构建时启用了断言。如果在构建 PostgreSQL 时定义了宏 `USE_ASSERT_CHECKING`（例如通过 `configure` 选项 `--enable-cassert`），则会报告已启用。默认情况下，PostgreSQL 是在未启用断言的情况下构建的。

`integer_datetimes` (`boolean`)

报告 PostgreSQL 是否在编译时启用了对 64 位整数日期和时间的支持。从 PostgreSQL 10 起，这个值总是`on`。

`lc_collate` (`string`)

报告对文本数据进行排序所使用的区域设置。更多信息参见第 23.1 节。此值在创建数据库时确定。

`lc_ctype` (`string`)

报告决定字符分类的区域设置。更多信息参见第 23.1 节。此值在创建数据库时确定。通常它与 `lc_collate` 相同，但特殊应用可能将它设置为不同的值。

`max_function_args` (`integer`)

报告函数参数的最大数量。它由编译服务器时的 `FUNC_MAX_ARGS` 值决定。默认值是 100 个参数。

`max_identifier_length` (`integer`)

报告标识符的最大长度。它由编译服务器时的 `NAMEDATALEN` 值减一决定。`NAMEDATALEN` 的默认值是 64；因此 `max_identifier_length` 的默认值是 63 字节，在使用多字节编码时，这可能不足 63 个字符。

`max_index_keys` (`integer`)

报告索引键的最大数目。它由编译服务器时的 `INDEX_MAX_KEYS` 值决定。默认值是 32 个键。

`segment_size` (`integer`)

报告一个文件段中可以存储的块（页）的数量。由编译服务器时的 `RELSEG_SIZE` 值决定。一个段文件的最大尺寸（以字节计）等于 `segment_size` 乘以 `block_size`，默认是 1GB。

`server_encoding` (`string`)

报告数据库的编码（字符集）。这是在数据库被创建时决定的。通常，客户端只需要关心 client_encoding 的值。

`server_version` (`string`)

报告服务器的版本号。它是由编译服务器时的 `PG_VERSION` 值决定的。

`server_version_num` (`integer`)

以整数形式报告服务器的版本号。它是由编译服务器时的 `PG_VERSION_NUM` 值决定的。

wal_block_size (integer)

报告一个 WAL 磁盘块的尺寸。由编译服务器时的 XLOG_BLCKSZ 值决定。默认是 8192 字节。

wal_segment_size (integer)

报告 WAL 段文件的大小。默认是 16MB。详见第 30.4 节。

19.16. 自定义选项

这个特性允许附加模块（例如过程语言）向 PostgreSQL 添加系统通常不认识的参数。这样便能以标准方式配置扩展模块。

自定义选项的名称由两部分组成：扩展名称和参数名本身，中间用句点分隔，类似于 SQL 中的限定名。例如plpgsql.variable_conflict。

因为自定义选项可能需要在尚未加载相关扩展模块的进程中设置，PostgreSQL 将接受任何两部分参数名称的设置。这些变量被视为占位符，在定义它们的模块加载之前没有任何功能。当加载扩展模块时，它将添加其变量定义并根据这些定义转换任何占位符值。如果存在以其扩展名称开头的任何未识别的占位符，将发出警告。

19.17. 开发人员选项

下列参数用于 PostgreSQL 源代码的开发工作，在某些情况下也用于辅助恢复严重损坏的数据库。没有理由在生产数据库中使用它们。因此，它们被排除在示例postgresql.conf 文件之外。请注意，许多参数需要特殊的源代码编译标志才能起作用。

allow_in_place_tablespaces (boolean)

当向 CREATE TABLESPACE 命令提供空的位置字符串时，允许将表空间创建为pg_tblspc 内的目录。这旨在允许测试主库和备库在同一台机器上运行的复制场景。这样的目录可能会混淆备份工具，因为这些工具期望在该位置只找到符号链接。只有超级用户能更改这个设置。

allow_system_table_mods (boolean)

允许修改系统表的结构。此参数由 initdb 使用。此参数只能在服务器启动时设置。

ignore_system_indexes (boolean)

读取系统表时忽略系统索引（但是修改系统表时依然同时更新索引）。这在从被破坏的系统索引中恢复数据时有用。这个参数在会话开始之后不能被更改。

post_auth_delay (integer)

如果非零，则在新服务器进程启动并执行认证过程之后，延迟指定的秒数。这旨在给开发者一个机会，用调试器附加到服务器进程上。此参数在会话开始之后不能更改。

pre_auth_delay (integer)

如果非零，则在新服务器进程刚刚派生之后、执行认证过程之前，延迟指定的秒数。这旨在给开发者一个机会，用调试器附加到服务器进程上，跟踪认证过程中的异常行为。此参数只能在postgresql.conf 文件中或在服务器命令行上设置。

trace_notify (boolean)

为 LISTEN 和 NOTIFY 命令生成大量调试输出。client_min_messages 或 log_min_messages 必须是 DEBUG1 或者更低才能把这种输出分别发送到客户端或者服务器日志。

trace_recovery_messages (enum)

启用原本不会记录的恢复相关调试输出。此参数允许用户覆盖 log_min_messages 的常规设置，但仅针对特定消息。它旨在用于调试热备。有效值为DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1 和 LOG。默认值 LOG 完全不影响是否记录消息的决定。其他值会让该优先级或更高优先级的恢复相关调试消息按 LOG 优先级记录；对于 log_min_messages 的常见设置，这会无条件地将它们发送到服务器日志。此参数只能在postgresql.conf 文件中或在服务器命令行上设置。

trace_sort (boolean)

如果开启，输出排序操作中的资源使用信息。只有在编译PostgreSQL 时定义了 TRACE_SORT 宏，这个参数才可用（不过，当前在默认情况下就定义了TRACE_SORT）。

trace_locks (boolean)

如果开启，发出锁使用情况的信息。被转储信息中包括锁操作的类型、锁的类型和被锁或被解锁对象的唯一标识符。同样包括的还有已经授予这个对象的锁类型的位掩码和等待这个对象的锁类型的位掩码。还会转储每种锁类型已授予的锁数、等待的锁数，以及它们的总数。一个日志文件输出的示例如下：

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  CleanUpLock: deleting: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(INVALID)
```

被转储结构体的详细信息可以在 src/include/storage/lock.h 中找到。

只有在编译 PostgreSQL 时定义了 LOCK_DEBUG 宏，这个参数才可用。

trace_lwlocks (boolean)

如果开启，发出轻量级锁的使用信息。轻量级锁主要是为了提供对共享内存数据结构的互斥访问。

只有在编译 PostgreSQL 时定义了 LOCK_DEBUG 宏，这个参数才可用。

trace_userlocks (boolean)

如果开启，发出关于用户锁使用的信息。与trace_locks 的输出一样，但只用于咨询锁。

只有在编译 PostgreSQL 时定义了 LOCK_DEBUG 宏，这个参数才可用。

trace_lock_oidmin (integer)

如果设置，不会跟踪 OID 小于此值的表上的锁（用于避免在系统表上的输出）。

只有在编译 PostgreSQL 时定义了 LOCK_DEBUG 宏，这个参数才可用。

trace_lock_table (integer)

无条件地跟踪此表（OID）上的锁。

只有在编译 PostgreSQL 时定义了 LOCK_DEBUG 宏，这个参数才可用。

debug_deadlocks (boolean)

如果设置，当死锁超时发生时，转储所有当前锁的信息。

只有在编译 PostgreSQL 时定义了 LOCK_DEBUG 宏，这个参数才可用。

log_btree_build_stats (boolean)

如果设置，会记录 B-树操作上的系统资源使用情况统计（内存和 CPU）。

只有在编译 PostgreSQL 时定义了 BTREE_BUILD_STATS 宏，这个参数才可用。

wal_consistency_checking (string)

这个参数被设计用来检查 WAL 重做例程中的缺陷。当这个参数被启用时，与该 WAL 记录相关的所有已修改缓冲区的整页镜像都会被添加到该记录中。如果该记录后来被重放，系统将首先应用每个记录然后测试该记录修改的缓冲区是否符合存储的镜像。在某些情况下（例如提示位），小的变动是可以接受的，并且会被忽略。任何预期之外的差别都将导致致命错误，最后中止恢复。

默认值为空字符串，表示禁用该功能。可以设置为all 以检查所有记录，或者设置为逗号分隔的资源管理器列表，以仅检查来自这些资源管理器的记录。目前支持的资源管理器有 heap、heap2、btree、hash、gin、gist、sequence、spgist、brin 和 generic。只有超级用户才能更改此设置。

wal_debug (boolean)

如果被打开，WAL 相关的调试输出将被发出。只有在编译PostgreSQL 时定义了 WAL_DEBUG 宏的情况下，这个参数才可用。

ignore_checksum_failure (boolean)

只有当数据校验和被启用时才有效。

在读取过程中检测到校验和失败通常会导致 PostgreSQL 报告错误，中止当前事务。将 ignore_checksum_failure 设置为 on 会使系统忽略失败（但仍报告警告），并继续处理。这种行为可能导致崩溃、传播或隐藏损坏，或引发其他严重问题。但是，如果块首部仍然正常，它可能允许你跳过错误，检索表中可能仍然存在的未损坏元组。如果首部损坏，即使启用此选项也会报告错误。默认设置为off。只有超级用户才能更改此设置。

zero_damaged_pages (boolean)

检测到损坏的页面头通常会导致 PostgreSQL 报告错误，中止当前事务。将 zero_damaged_pages 设置为 on 会导致系统报告警告，将内存中的损坏页面清零，并继

续处理。这种行为会破坏数据，即损坏页面上的所有行。但是，它确实允许你跳过错误，并从表中可能存在的未损坏页面中检索行。如果由于硬件或软件错误而发生损坏，这对于恢复数据很有用。通常在放弃从表的损坏页面恢复数据的希望之前，不应将其设置为 on。清零的页面不会强制写入磁盘，因此建议在再次关闭此参数之前重新创建表或索引。默认设置为 off。只有超级用户能更改这个设置。

jit_debugging_support (boolean)

如果 LLVM 具有所需的功能，则向GDB 注册生成的函数。这会让调试更加容易。默认设置为 off。这个参数只能在服务器启动时设置。

jit_dump_bitcode (boolean)

将生成的 LLVM IR 写入文件系统，位于data_directory 内。这仅对开发 JIT 的内部实现有用。默认设置为 off。只有超级用户能更改这个设置。

jit_expressions (boolean)

当 JIT 编译被激活时（见第 32.2 节），确定表达式是否用 JIT 编译。默认值是on。

jit_profiling_support (boolean)

如果 LLVM 具有所需的功能，则输出让perf 对 JIT 生成的函数进行性能分析所需的数据。这会将文件写入 $HOME/.debug/jit/；用户应在需要时自行清理。默认设置为 off。这个参数只能在服务器启动时设置。

jit_tuple_deforming (boolean)

当 JIT 编译被激活时（见第 32.2 节），确定元组拆解是否被 JIT 编译。默认值是on。

19.18. 短选项

为了方便起见，系统中还为一些参数提供了单字母的命令行选项开关。它们在表 19.3 中描述。其中一些选项是由于历史原因而存在，它们以单字母选项的形式存在，并不一定表示鼓励频繁使用这些选项。

表 19.3. 短选项对照

短选项	等效于
-B *x*	shared_buffers = *x*
-d *x*	log_min_messages = DEBUG*x*
-e	datestyle = euro
-fb, -fh, -fi, -fm, -fn, -fo, -fs, -ft	enable_bitmapscan = off, enable_hashjoin = off, enable_indexscan = off, enable_mergejoin = off, enable_nestloop = off, enable_indexonlyscan = off, enable_seqscan = off, enable_tidscan = off
-F	fsync = off
-h *x*	listen_addresses = *x*
-i	listen_addresses = '*'
-k *x*	unix_socket_directories = *x*

短选项	等效于
-l	ssl = on
-N *x*	max_connections = *x*
-O	allow_system_table_mods = on
-p *x*	port = *x*
-P	ignore_system_indexes = on
-s	log_statement_stats = on
-S *x*	work_mem = *x*
-tpa, -tpl, -te	log_parser_stats = on, log_planner_stats = on, log_executor_stats = on
-W *x*	post_auth_delay = *x*

第 20 章 客户端认证

当客户端应用连接到数据库服务器时，它会指定要以哪个 PostgreSQL 数据库用户名连接，这很像以某个特定用户身份登录 Unix 计算机。在 SQL 环境中，当前活动的数据库用户名决定了对数据库对象的访问权限 – 详见第 21 章。因此，必须限制哪些数据库用户能够连接。

> **注意**
>
> 如第 21 章中所述，PostgreSQL 实际上是以“角色”为单位进行权限管理的。在本章中，我们统一使用数据库用户来表示“拥有 `LOGIN` 权限的角色”。

认证是数据库服务器确认客户端身份的过程，并据此决定是否允许该客户端应用（或者运行该客户端应用的用户）以请求的数据库用户名进行连接。

PostgreSQL 提供了多种不同的客户端认证方法。用于认证特定客户端连接的方法可以根据（客户端）主机地址、数据库和用户来选择。

PostgreSQL 数据库用户名在逻辑上独立于服务器所在操作系统中的用户名。如果某台服务器的所有用户在该机器上也都有账号，那么为他们分配与操作系统用户名一致的数据库用户名是有意义的。不过，接受远程连接的服务器可能有许多数据库用户并没有本地操作系统账号，在这种情况下，数据库用户名与操作系统用户名之间就不必存在任何对应关系。

20.1. `pg_hba.conf` 文件

客户端认证由一个配置文件控制，该文件按惯例命名为 `pg_hba.conf`，并存放在数据库集簇的数据目录中。（HBA 代表 host-based authentication，即基于主机的认证。）当数据目录由 `initdb` 初始化时，会安装一个默认的 `pg_hba.conf` 文件。不过，也可以把认证配置文件放在别处；请参见配置参数 hba_file。

`pg_hba.conf` 文件的基本格式是一组记录，每行一条。空行和 `#` 注释字符之后的所有文本都会被忽略。记录不能跨行续写。每条记录由若干字段组成，字段之间用空格和/或制表符分隔。如果字段值用双引号括起来，就可以包含空白。在数据库、用户或地址字段中，将关键字（例如 `all` 或 `replication`）用引号括起来，会使其失去特殊含义，只匹配同名的数据库、用户或主机。

每条认证记录都指定一种连接类型、一个客户端 IP 地址范围（如果该连接类型需要）、一个数据库名、一个用户名，以及对匹配这些参数的连接要使用的认证方法。第一条同时匹配连接类型、客户端地址、请求数据库和用户名的记录会被用来执行认证。这里不存在“继续向后匹配”或“后备”机制：如果选中某条记录而认证失败，就不会再考虑后续记录。如果没有任何记录匹配，则拒绝访问。

记录可以采用以下七种格式之一

```
local      database  user  auth-method  [auth-options]
host       database  user  address  auth-method  [auth-options]
hostssl    database  user  address  auth-method  [auth-options]
hostnossl  database  user  address  auth-method  [auth-options]
host       database  user  IP-address  IP-mask  auth-method  [auth-options]
hostssl    database  user  IP-address  IP-mask  auth-method  [auth-options]
hostnossl  database  user  IP-address  IP-mask  auth-method  [auth-options]
```

各字段的含义如下：

`local`

该记录匹配使用 Unix 域套接字的连接尝试。没有这种类型的记录时，Unix 域套接字连接将被禁止。

`host`

此记录匹配通过 TCP/IP 发起的连接尝试。`host` 记录既匹配 SSL 连接，也匹配非 SSL 连接。

> **注意**
>
> 除非服务器以适当的 listen_addresses 配置参数值启动，否则远程 TCP/IP 连接将不可用，因为默认行为是只在本地回环地址 `localhost` 上监听 TCP/IP 连接。

`hostssl`

该记录匹配使用 TCP/IP 发起的连接尝试，但仅当连接使用 SSL 加密时才匹配。

要使用此选项，服务器必须在构建时启用 SSL 支持。此外，还必须通过设置ssl 配置参数来启用 SSL（有关更多信息，请参见第 18.9 节）。否则，`hostssl` 记录将被忽略，只会记录一条警告，说明它无法匹配任何连接。

`hostnossl`

该记录类型与 `hostssl` 的行为相反；它只匹配通过 TCP/IP 发起且不使用 SSL 的连接尝试。

`database`

指定此记录匹配的数据库名称。`all` 表示匹配所有数据库。`sameuser` 表示请求的数据库与请求的用户同名时才匹配。`samerole` 要求请求的用户是与所请求数据库同名的角色的成员。（`samegroup` 是 `samerole` 已过时但仍被接受的写法。）对于 `samerole`，只有显式地直接或间接属于该角色，超级用户才被视为其成员，仅凭超级用户身份并不算。`replication` 表示请求物理复制连接时匹配此记录（注意，复制连接不指定任何特定数据库）。其他值则表示某个特定的 PostgreSQL 数据库的名称。可以用逗号分隔多个数据库名称。也可以在文件名前加 `@`，指定一个单独存放数据库名称的文件。

`user`

指定此记录匹配的数据库用户名。`all` 表示匹配所有用户。其他值可以是某个特定数据库用户的名称，也可以是前面带 `+` 的组名。（请记住，在 PostgreSQL 中，用户和组并无实质区别；`+` 实际表示“匹配直接或间接属于此角色的任何角色”，而不带 `+` 的名称仅匹配该角色本身。）为此，只有显式地直接或间接属于该角色，超级用户才被视为其成员，仅凭超级用户身份并不算。可以用逗号分隔多个用户名。也可以在文件名前加 `@`，指定一个单独存放用户名的文件。

`address`

指定此记录匹配的客户端机器地址。此字段可以包含主机名、IP 地址范围或下面提到的特殊关键字之一。

IP 地址范围使用标准的数字表示法来指定起始地址，然后是斜杠（/）和一个 CIDR 掩码长度。掩码长度表示客户端 IP 地址必须匹配的高位比特数。给定 IP 地址中右侧的比特应为零。IP 地址、/和 CIDR 掩码长度之间不得有任何空白。

以这种方式指定的 IPv4 地址范围的典型示例包括 `172.20.143.89/32` 用于单个主机，或 `172.20.143.0/24` 用于小型网络，或`10.6.0.0/16` 用于较大的网络。IPv6 地址范围可能看起来像`::1/128` 用于单个主机（在这种情况下是 IPv6 环回地址）或 `fe80::7a31:c1ff:0000:0000/96` 用于小型网络。`0.0.0.0/0` 代表所有 IPv4 地址，`::0/0` 代表所有 IPv6 地址。要指定单个主机，请对 IPv4 使用 32 的掩码长度，对 IPv6 使用 128。在网络地址中，不要省略尾部的零。

以 IPv4 格式给出的条目将仅匹配 IPv4 连接，以 IPv6 格式给出的条目将仅匹配 IPv6 连接，即使所代表的地址在 IPv4-in-IPv6 范围内。请注意，如果系统的 C 库不支持 IPv6 地址，以 IPv6 格式给出的条目将被拒绝。

你也可以写 `all` 来匹配任何 IP 地址，`samehost` 来匹配服务器自己的任何 IP 地址，或 `samenet` 来匹配服务器直接连接到的任何子网中的任何地址。

如果指定了主机名（任何不是 IP 地址范围或特殊关键字的内容都被视为主机名），则将该名称与客户端 IP 地址的反向名称解析结果进行比较（例如，如果使用 DNS，则进行反向 DNS 查找）。主机名比较不区分大小写。如果匹配成功，则对主机名执行正向名称解析（例如，进行正向 DNS 查找），以检查其解析为的任何地址是否等于客户端 IP 地址。如果两个方向都匹配，则将条目视为匹配。（在`pg_hba.conf` 中使用的主机名应该是客户端 IP 地址的地址到名称解析返回的名称，否则该行将不会匹配。一些主机名数据库允许将 IP 地址与多个主机名关联，但操作系统在要求解析 IP 地址时只会返回一个主机名。）

以点（`.`）开头的主机名规范匹配实际主机名的后缀。因此，`.example.com` 将匹配 `foo.example.com`（但不匹配单独的 `example.com`）。

当在 `pg_hba.conf` 中指定主机名时，应确保名称解析相对快速。设置一个本地名称解析缓存可能会有帮助，如`nscd`。此外，还可能希望启用配置参数`log_hostname`，以便在日志中看到客户端的主机名而不是 IP 地址。

此字段仅适用于 `host`、`hostssl` 和 `hostnossl` 记录。

注意

用户有时会想知道为什么主机名以这种看似复杂的方式处理，包括两次名称解析，其中包括对客户端 IP 地址的反向查找。如果客户端的反向 DNS 条目未设置或返回了不符合预期的主机名，则使用该功能会变得复杂。这主要是为了效率：这样，连接尝试最多需要两次解析器查找，一次反向查找和一次正向查找。如果某个地址存在解析器问题，那就只会成为该客户端的问题。一个假设的替代实现只进行正向查找的情况下，在每次连接尝试期间都必须解析`pg_hba.conf` 中提到的每个主机名。如果列出了许多名称，这可能会非常慢。如果其中一个主机名存在解析器问题，那么这将成为所有人的问题。

此外，实现后缀匹配功能需要进行反向查找，因为需要知道实际客户端主机名以便将其与模式进行匹配。

请注意，这种行为与其他流行的基于主机名的访问控制实现一致，例如 Apache HTTP 服务器和 TCP Wrappers。

IP-address
IP-mask

这两个字段可以用作 *IP-address*/*mask-length* 表示法的替代方案。不指定掩码长度，而是在单独一列中指定实际掩码。例如，255.0.0.0 表示 IPv4 的 CIDR 掩码长度为 8，而 255.255.255.255 表示 CIDR 掩码长度为 32。

这些字段仅适用于 host、hostssl 和 hostnossl 记录。

auth-method

指定连接匹配此记录时要使用的认证方法。可选值在此处做了概述；详细说明见第 20.3 节。所有选项都必须使用小写并且区分大小写，因此即使是像 ldap 这样的首字母缩写也必须写成小写。

trust

无条件允许连接。这种方法允许任何能够连接到 PostgreSQL 数据库服务器的人以任意他们希望的 PostgreSQL 用户身份登录，无需密码或任何其他认证。详见第 20.4 节。

reject

无条件拒绝连接。这对于“过滤掉”某些主机很有用，例如一个reject 行可以阻止特定主机连接，而后面的行允许特定网络中的其余主机连接。

scram-sha-256

执行 SCRAM-SHA-256 认证以校验用户密码。详见第 20.5 节。

md5

执行 SCRAM-SHA-256 或 MD5 认证以校验用户密码。详见第 20.5 节。

password

要求客户端提供未加密的密码以完成认证。由于密码会以明文形式通过网络发送，因此不应在不受信任的网络上使用。详见第 20.5 节。

gss

使用 GSSAPI 认证用户。这仅适用于 TCP/IP 连接。详情参见第 20.6 节。

sspi

使用 SSPI 对用户进行认证。这仅适用于 Windows。详见第 20.7 节。

ident

通过联系客户端上的 ident 服务器获取客户端的操作系统用户名，并检查它是否与请求的数据库用户名匹配。Ident 认证只能用于 TCP/IP 连接。当为本地连接指定时，将改为使用 peer 认证。详见第 20.8 节。

peer

从操作系统获取客户端的操作系统用户名，并检查是否与请求的数据库用户名匹配。这仅适用于本地连接。有关详细信息，请参见第 20.9 节。

`ldap`

使用 LDAP 服务器进行认证。详见第 20.10 节。

`radius`

使用 RADIUS 服务器进行认证。详见第 20.11 节。

`cert`

使用 SSL 客户端证书进行认证。详见第 20.12 节。

`pam`

使用操作系统提供的可插拔认证模块（PAM）服务进行认证。详见第 20.13 节。

`bsd`

使用操作系统提供的 BSD 认证服务进行认证。详见第 20.14 节。

`auth-options`

在 *`auth-method`* 字段之后，可以有一个或多个形如 *`name=value`* 的字段，用于指定认证方法的选项。下文会详细说明各认证方法有哪些可用选项。

除下文列出的各方法专用选项外，还有一个与方法无关的认证选项 `clientcert`，可以在任何 `hostssl` 记录中指定。将其设为 `1` 时，除了满足认证方法的其他要求外，客户端还必须提供有效（受信任）的 SSL 证书。

由 `@` 构造引用的文件会被读取为名称列表，其中的名称可以用空白或逗号分隔。注释仍然用 `#` 引入，与 `pg_hba.conf` 中相同，并且允许嵌套的 `@` 构造。除非 `@` 后面的文件名是绝对路径，否则它会被视为相对于引用它的文件所在目录。

由于每次连接尝试都会按顺序检查 `pg_hba.conf` 记录，因此记录的顺序很重要。通常，靠前的记录使用较严格的连接匹配条件和较弱的认证方法，靠后的记录使用较宽松的匹配条件和较强的认证方法。例如，可能希望对本地 TCP/IP 连接使用 `trust` 认证，而要求远程 TCP/IP 连接提供密码。此时，为来自 127.0.0.1 的连接指定 `trust` 认证的记录，应放在为更大范围的允许客户端 IP 地址指定密码认证的记录之前。

在启动时以及主服务器进程收到 SIGHUP 信号时，`pg_hba.conf` 文件会被读取。如果你在运行中的系统上编辑了该文件，就需要通知 postmaster（使用 `pg_ctl reload`、调用 SQL 函数 `pg_reload_conf()`，或者使用 `kill -HUP`）重新读取该文件。

> **注意**
>
> 前面的说明不适用于 Microsoft Windows：在 Windows 上，`pg_hba.conf` 文件中的任何更改会立即被应用到后续的新连接上。

系统视图 `pg_hba_file_rules` 有助于预先测试对 `pg_hba.conf` 文件的修改，也可以在加载该文件没有产生预期效果时用于诊断问题。视图中 `error` 字段非空的行表示文件对应行存在问题。

> **提示**
>
> 要连接到一个特定数据库，一个用户必须不仅要通过pg_hba.conf 检查，还必须要有该数据库上的CONNECT 权限。如果你希望限制哪些用户能够连接到哪些数据库，授予/撤销 CONNECT 权限通常比在 pg_hba.conf 项中设置规则简单。

例 20.1 展示了一些 pg_hba.conf 条目示例。有关不同认证方法的详细信息，参见下一节。

例 20.1. 示例 pg_hba.conf 项

```
# 允许本地系统上的任何用户
# 通过 Unix 域套接字（本地连接的默认方式）以任意
# 数据库用户名连接到任意数据库。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     trust

# 相同的规则，但是使用本地环回 TCP/IP 连接。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             127.0.0.1/32            trust

# 和前一行相同，但是使用了一个独立的掩码列
#
# TYPE  DATABASE        USER            IP-ADDRESS      IP-MASK             METHOD
host    all             all             127.0.0.1       255.255.255.255     trust

# IPv6 上相同的规则
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             ::1/128                 trust

# 使用主机名的相同规则（通常同时覆盖 IPv4 和 IPv6）。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             localhost               trust

# 允许来自任意具有 IP 地址192.168.93.x 的主机上任意
# 用户以 ident 为该连接所报告的相同用户名连接到
# 数据库 "postgres"（通常是操作系统用户名）。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.93.0/24         ident

# 如果用户的密码被正确提供，允许来自主机 192.168.12.10
# 的任意用户连接到数据库 "postgres"。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.12.10/32        scram-sha-256

# 如果用户的密码被正确提供，允许 example.com 域中主机上
# 的任意用户连接到任意数据库。
#
# 为大部分用户要求SCRAM认证，但是用户'mike'是个例外，
# 他使用的是不支持SCRAM认证的旧客户端。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             mike            .example.com            md5
host    all             all             .example.com            scram-sha-256
```

```
# 如果没有前面的 "host" 行，这两行将拒绝所有来自 192.168.54.1 的
# 连接（因为该条目会先被匹配），但允许来自互联网其他任何位置的
# GSSAPI 连接。零掩码表示不考虑主机 IP 地址中的任何位，
# 因而会匹配任意主机。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.54.1/32         reject
host    all             all             0.0.0.0/0               gss

# 允许来自 192.168.x.x 主机的用户连接到任意数据库，如果它们能够
# 通过 ident 检查。例如，假设 ident说用户是 "bryanh" 并且他要求以
# PostgreSQL 用户 "guest1" 连接，如果在 pg_ident.conf 有一个映射
# "omicron" 的条目表明 "bryanh" 被允许以 "guest1" 连接，则该连接将被允许。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.0.0/16          ident map=omicron

# 如果这些是本地连接的唯一三行，它们将允许本地用户只连接到
# 自己的数据库（与其数据库用户名同名的数据库），但管理员和
# 角色 "support" 的成员除外，他们可以连接到所有数据库。
# 文件 $PGDATA/admins 包含管理员名称列表。
# 所有情况下都要求提供密码。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   sameuser        all                                     md5
local   all             @admins                                 md5
local   all             +support                                md5

# 上面的最后两行可以被整合为一行：
local   all             @admins,+support                        md5

# 数据库列也可以用列表和文件名：
local   db1,db2,@demodbs  all                                   md5
```

20.2. 用户名映射

当使用 Ident 或 GSSAPI 之类的外部认证系统时，发起连接的操作系统用户名可能不同于要使用的数据库用户（角色）。在这种情况下，可以通过用户名映射把操作系统用户名映射为数据库用户。要使用用户名映射，需要在 pg_hba.conf 的选项字段中指定 map=*map-name*。此选项适用于所有会接收外部用户名的认证方法。由于不同连接可能需要不同映射，在 pg_hba.conf 中通过 *map-name* 参数指定要使用的映射，以表明每个连接应使用哪个映射。

用户名映射在 ident 映射文件中定义。该文件默认名为 pg_ident.conf，存储在集簇的数据目录中。（也可以将映射文件放在其他位置；参见ident_file 配置参数。）ident 映射文件中各行的基本格式如下：

```
map-name system-username database-username
```

注释和空白的处理方式与 pg_hba.conf 相同。*map-name* 是任意名称，用于在 pg_hba.conf 中引用此映射。另两个字段指定操作系统用户名及与之匹配的数据库用户名。同一个 *map-name* 可以重复使用，以便在一项映射中指定多个用户对应关系。

一个操作系统用户可以对应多少个数据库用户，没有限制，反之亦然。因此，映射中的条目应理解为“此操作系统用户可以作为此数据库用户连接”，而不是说二者等同。只要存在一个映射条目，将从外部认证系统取得的用户名与用户请求连接时使用的数据库用户名配对，就会允许该连接。

如果 *system-username* 字段以斜杠（/）开头，该字段的其余部分就会被视为正则表达式。（参见第 9.7.3.1 节，其中介绍了 PostgreSQL 的正则表达式语法。）正则表达式可以包含一个捕获，即用圆括号括起来的子表达式，随后可以在 *database-username* 字段中使用 \1（反斜杠加数字一）来引用。这样就可以在一行中映射多个用户名，对简单的语法替换尤其有用。例如，以下条目

```
mymap    /^(.*)@mydomain\.com$       \1
mymap    /^(.*)@otherdomain\.com$    guest
```

会去掉系统用户名中的域名部分（如果用户名以 @mydomain.com 结尾），并允许系统用户名以 @otherdomain.com 结尾的任何用户登录为 guest。

> **提示**
>
> 记住在默认情况下，一个正则表达式可以只匹配字符串的一部分。如上例所示，使用^和$来强制匹配整个系统用户名通常是明智的。

在启动时以及主服务器进程收到 SIGHUP 信号时，pg_ident.conf 文件会被读取。如果你在运行中的系统上编辑了该文件，就需要通知 postmaster（使用 pg_ctl reload、调用 SQL 函数 pg_reload_conf()，或者使用 kill -HUP）重新读取该文件。

例 20.2 展示了一个可与例 20.1 中的 pg_hba.conf 文件配合使用的 pg_ident.conf 文件。在这个示例中，任何登录到 192.168 网络中某台机器上的用户，如果其操作系统用户名不是 bryanh、ann 或 robert，都不会被授予访问权限。Unix 用户 robert 只有在尝试以 PostgreSQL 用户 bob 身份连接时才被允许访问，而不能以 robert 或其他身份连接。ann 只能以 ann 身份连接。用户 bryanh 则可以以 bryanh 或 guest1 身份连接。

例 20.2. 示例 pg_ident.conf 文件

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME

omicron         bryanh                  bryanh
omicron         ann                     ann
# bob 在这些机器上有用户名 robert
omicron         robert                  bob
# bryanh 也可以作为 guest1 连接
omicron         bryanh                  guest1
```

20.3. 认证方法

PostgreSQL 提供了多种用户认证方法：

- 信任认证，即简单地相信用户就是其所声称的身份。
- 密码认证，需要用户发送密码。
- GSSAPI 认证，依赖于一个兼容 GSSAPI 的安全库。它通常用于访问 Kerberos 或 Microsoft Active Directory 等认证服务器。
- SSPI 认证，使用一种类似于 GSSAPI 的 Windows 特有协议。

- Ident 认证，依赖客户端机器上的“标识协议”（RFC 1413）服务。（对于本地 Unix 套接字连接，会将其视为 peer 认证。）
- Peer 认证，依赖于操作系统提供的机制来识别本地连接另一端的进程。它不支持远程连接。
- LDAP 认证，依赖于 LDAP 认证服务器。
- RADIUS 认证，依赖于 RADIUS 认证服务器。
- 证书认证，需要 SSL 连接，并通过检查用户发送的 SSL 证书来认证用户。
- PAM 认证，依赖于 PAM（可插拔认证模块）库。
- BSD 认证，依赖于 BSD 认证框架（目前仅在 OpenBSD 上可用）。

通常建议对本地连接使用 Peer 认证，不过在某些情况下，信任认证也可能已经足够。密码认证是远程连接最容易采用的选择。其余所有选项都需要某种外部安全基础设施（通常是认证服务器，或用于签发 SSL 证书的证书颁发机构），或者属于平台特定功能。

以下各节将更详细地介绍这些认证方法。

20.4. 信任认证

当 `trust` 认证被指定时，PostgreSQL 假设任何可以连接到服务器的人都被授权使用他们指定的任何数据库用户名（即使是超级用户）访问数据库。当然，在`database` 和 `user` 列中设置的限制仍然适用。只有当在操作系统层对进入服务器的连接有足够保护时，才应该使用这种方法。

`trust` 认证对于单用户工作站的本地连接是非常合适和方便的。通常它本身不适用于一台多用户机器。不过，只要你利用文件系统权限限制了对服务器的 Unix 域套接字文件的访问，即使在多用户机器上，你也可能可以使用`trust`。要做这些限制，你可以设置第 19.3 节中描述的 `unix_socket_permissions` 配置参数（可能还有 `unix_socket_group`）。或者你可以设置 `unix_socket_directories` 配置参数来把 Unix 域套接字文件放在一个经过恰当限制的目录中。

设置文件系统权限只能有助于 Unix 套接字连接。本地 TCP/IP 连接不会被文件系统权限限制。因此，如果你想利用文件系统权限来控制本地安全，那么从`pg_hba.conf` 中移除 `host ... 127.0.0.1 ...`行，或者把它改为一个非`trust` 认证方法。

只有当你信任由 `pg_hba.conf` 中指定 `trust` 的行所允许连接的每台机器上的每个用户时，`trust` 认证才适合用于 TCP/IP 连接。对来自 localhost（127.0.0.1）以外的任何 TCP/IP 连接使用 `trust`，通常都不合理。

20.5. 密码认证

有几种基于密码的认证方法。这些方法的过程类似，但是区别在于用户密码如何被存放在服务器上以及客户端提供的密码如何被通过连接发送。

`scram-sha-256`

`scram-sha-256` 方法执行 SCRAM-SHA-256 认证，其定义见 RFC 7677[1]。它是一种挑战-响应方案，可防止在不受信任连接上嗅探密码，并支持在服务器上以被认为安全的密码学 hash 形式存储密码。

这是当前提供的方法中最安全的一种，但是旧的客户端库不支持这种方法。

[1] https://tools.ietf.org/html/rfc7677

md5

方法 md5 使用一种自定义的、安全性较低的挑战-响应机制。它能防止密码嗅探，并避免在服务器上以明文存储密码；但如果攻击者设法从服务器上窃取了密码 hash，它就无法提供保护。此外，如今已不再认为 MD5 hash 算法能抵御有决心的攻击者。

md5 方法不能与 db_user_namespace 功能一起使用。

为了简化从 md5 方法到较新的 SCRAM 方法的转变，如果在pg_hba.conf 中指定了 md5 但是用户在服务器上的密码是为 SCRAM（见下文）加密的，则将自动选择基于 SCRAM 的认证。

password

方法 password 以明文形式发送密码，因此它容易受到密码“嗅探”攻击。如果可能应该尽量避免使用它。不过，如果连接被 SSL 加密保护着，那么可以安全地使用password（不过如果依靠 SSL，SSL 证书认证可能是更好的选择）。

PostgreSQL 数据库密码独立于操作系统用户密码。每个数据库用户的密码都存储在系统目录 pg_authid 中。密码可以通过 SQL 命令 CREATE ROLE 和 ALTER ROLE 管理，例如 **CREATE ROLE foo WITH LOGIN PASSWORD 'secret'**，或者通过 psql 的 \password 命令管理。如果某个用户没有设置密码，则其存储密码为 null，该用户的密码认证将始终失败。

不同基于密码的认证方法是否可用，取决于用户在服务器上的密码是如何加密的（更准确地说，是如何进行 hash 运算的）。这由设置密码时的配置参数password_encryption 控制。如果密码是使用 scram-sha-256 设置加密的，那么它可以用于 scram-sha-256 和 password 认证方法（但后一种情况下密码会以明文传输）。如上所述，此时如果认证方法指定为 md5，也会自动切换为使用 scram-sha-256，因此同样能够工作。如果密码是使用 md5 设置加密的，那么它只能用于 md5 和 password 认证方法（同样，后一种情况下密码会以明文传输）。（以前的 PostgreSQL 版本支持在服务器上存储明文密码，但现在已经不再可能。）要检查当前存储的密码 hash，可以查看系统目录 pg_authid。

要把现有安装从 md5 升级到 scram-sha-256，需要先确保所有正在使用的客户端库都足够新并支持 SCRAM，然后在 postgresql.conf 中设置 password_encryption = 'scram-sha-256'，要求所有用户重新设置密码，并将 pg_hba.conf 中的认证方法设置改为 scram-sha-256。

20.6. GSSAPI 认证

GSSAPI 是 RFC 2743 定义的安全认证行业标准协议。PostgreSQL 按照 RFC 1964 支持使用 Kerberos 认证的 GSSAPI。GSSAPI 为支持它的系统提供自动认证（单点登录）。认证过程本身是安全的，但除非使用 SSL，否则数据库连接上传输的数据不会加密。

当编译 PostgreSQL 时，GSSAPI 支持必须被启用，详见第 16 章。

GSSAPI 使用 Kerberos 时，采用格式为 *servicename*/*hostname*@*realm* 的标准主体。PostgreSQL 服务器会接受其所用 keytab 中包含的任何主体，但客户端建立连接时，必须注意通过 krbsrvname 连接参数指定正确的主体信息。（另见第 34.1.2 节。）构建时可以使用 ./configure --with-krb-srvnam=*whatever*，将安装默认值从 postgres 改为其他值。在大多数环境中，无需更改此参数。某些 Kerberos 实现可能要求不同的服务名，例如 Microsoft Active Directory 要求服务名使用大写（POSTGRES）。

hostname 是服务器机器的完全限定主机名。服务主体的 realm 是服务器机器的首选 realm。

可以通过 pg_ident.conf 将客户端主体映射到不同的 PostgreSQL 数据库用户名。例如，可以将 pgusername@realm 映射为 pgusername。也可以不使用任何映射，直接将完整的 username@realm 主体用作 PostgreSQL 中的角色名。

PostgreSQL 还支持一个从主体中去掉 realm 的参数。提供这种方法是为了向后兼容，强烈不建议使用，因为这样就无法区分来自不同 realm 但用户名相同的用户。要启用此行为，将 include_realm 设为 0。对于简单的单 realm 安装环境，如果同时设置 krb_realm 参数（它会检查主体的 realm 是否与 krb_realm 参数值完全一致），这种做法仍是安全的；但与在 pg_ident.conf 中指定显式映射相比，它的能力较弱。

确保 PostgreSQL 服务器账户能够读取服务器的 keytab 文件（最好只能读取，不能写入）。（另见第 18.1 节。）密钥文件的位置由 krb_server_keyfile 配置参数指定。默认位置是 /usr/local/pgsql/etc/krb5.keytab（或者构建时用 sysconfdir 指定的目录）。出于安全考虑，建议为 PostgreSQL 服务器使用专用 keytab，而不是放宽系统 keytab 文件的权限。

keytab 文件由 Kerberos 软件生成；详情参见 Kerberos 文档。以下示例适用于兼容 MIT 的 Kerberos 5 实现：

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

连接数据库时，请确保持有与请求的数据库用户名相匹配的主体票据。例如，数据库用户名为 fred 时，主体 fred@EXAMPLE.COM 可以连接。如果还要允许主体 fred/users.example.com@EXAMPLE.COM，请按第 20.2 节所述使用用户名映射。

以下配置选项适用于 GSSAPI：

include_realm

如果设为 0，则在通过用户名映射（第 20.2 节）之前，会先从已认证用户的主体名中去掉 realm 名称。不建议这样做；它主要是为了向后兼容而保留的，因为在多 realm 环境中这并不安全，除非同时使用了 krb_realm。建议将 include_realm 保持为默认值（1），并在 pg_ident.conf 中提供显式映射，把主体名转换成 PostgreSQL 用户名。

map

允许在系统用户名与数据库用户名之间建立映射。详见第 20.2 节。对于 username@EXAMPLE.COM（或较少见的 username/hostbased@EXAMPLE.COM）这样的 GSSAPI/Kerberos 主体，映射所用的用户名是 username@EXAMPLE.COM（或相应的 username/hostbased@EXAMPLE.COM），除非将 include_realm 设为 0，此时映射所见的系统用户名为 username（或 username/hostbased）。

krb_realm

设置用于匹配用户主体名的 realm。如果设置了该参数，则只接受来自该 realm 的用户；如果未设置，则允许来自任意 realm 的用户连接，但仍受已执行的用户名映射约束。

20.7. SSPI 认证

SSPI 是一种用于安全认证和单点登录的 Windows 技术。PostgreSQL 将在 negotiate 模式下使用 SSPI，该模式会在可能时使用 Kerberos，并在其他情况下自动回退到 NTLM。SSPI 和 GSSAPI 作为客户端和服务器是可互操作的，例如，SSPI 客户端可以向 GSSAPI 服务器完成认证。建议在 Windows 客户端和服务器上使用 SSPI，在非 Windows 平台上使用 GSSAPI。

当使用 Kerberos 认证时，SSPI 和 GSSAPI 的工作方式相同，详见第 20.6 节。

SSPI 支持下列配置选项：

`include_realm`

如果设为 0，则在通过用户名映射（第 20.2 节）之前，会先从已认证用户的主体名中去掉 realm 名称。不建议这样做；它主要是为了向后兼容而保留的，因为在多 realm 环境中这并不安全，除非同时使用了 `krb_realm`。建议将 `include_realm` 保持为默认值（1），并在 `pg_ident.conf` 中提供显式映射，把主体名转换成 PostgreSQL 用户名。

`compat_realm`

如果设为 1，则会在 `include_realm` 选项中使用域的 SAM 兼容名称（也称为 NetBIOS 名称）。这是默认值。如果设为 0，则会使用 Kerberos 用户主体名中的真实 realm 名称。

只有当服务器运行在域账号（包括域成员系统上的虚拟服务账号）下，并且所有通过 SSPI 认证的客户端也都使用域账号时，才可禁用此选项；否则，禁用此选项会导致认证失败。

`upn_username`

如果此选项与 `compat_realm` 一起启用，则认证时会使用 Kerberos UPN 中的用户名。如果禁用它（默认值），则使用 SAM 兼容用户名。默认情况下，对新建用户账号而言，这两个名称是相同的。

注意，如果没有显式指定用户名，libpq 会使用 SAM 兼容名称。如果你使用的是 libpq 或基于它的驱动，应当保持该选项为禁用状态，或者在连接字符串中显式指定用户名。

`map`

允许在系统用户名和数据库用户名之间进行映射。详见第 20.2 节。对于 SSPI/Kerberos 主体，例如 `username@EXAMPLE.COM`（或者较少见的 `username/hostbased@EXAMPLE.COM`），用于映射的用户名分别是 `username@EXAMPLE.COM`（或 `username/hostbased@EXAMPLE.COM`），除非已经将 `include_realm` 设为 0；在那种情况下，映射时视为系统用户名的是 `username`（或 `username/hostbased`）。

`krb_realm`

设置用于匹配用户主体名的 realm。如果设置了该参数，则只接受来自该 realm 的用户；如果未设置，则允许来自任意 realm 的用户连接，但仍受已执行的用户名映射约束。

20.8. Ident 认证

ident 认证方法通过从一个 ident 服务器获得客户端的操作系统用户名并且用它作为被允许的数据库用户名（和可选的用户名映射）来工作。它只在 TCP/IP 连接上支持。

> **注意**
>
> 当为一个本地（非 TCP/IP）连接指定 ident 时，将实际使用 peer 认证（见第 20.9 节）。

ident 支持下列配置选项：

`map`

允许在系统用户名和数据库用户名之间进行映射。详见第 20.2 节。

“标识协议”在 RFC 1413[2] 中有描述。几乎每个类 Unix 的操作系统都默认安装了一个 ident 服务器，监听 TCP 端口 113。ident 服务器的基本功能是回答类似于“哪个用户发起了从你的端口 *X* 到我的端口 *Y* 的连接？”这样的问题。由于 PostgreSQL 在建立物理连接时知道 *X* 和 *Y*，它可以查询连接客户端主机上的 ident 服务器，理论上可以确定任何给定连接的操作系统用户。

这个过程的缺点在于它依赖客户端本身的可信性：如果客户端机器不可信或者已被攻破，攻击者几乎可以在 113 端口上运行任何程序，并返回任意他们选择的用户名。因此，这种认证方法只适用于封闭网络，在这类网络中每台客户端机器都受到严格控制，而且数据库管理员与系统管理员之间保持密切协作。换句话说，你必须信任运行 ident 服务器的那台机器。请注意下面的警告：

> 标识协议的本意不是作为一种授权或访问控制协议。
>
> —RFC 1413

有些 ident 服务器提供了一个非标准选项，会让返回的用户名被加密，而解密所需密钥只有发起连接机器的管理员才知道。将 ident 服务器与 PostgreSQL 配合使用时，绝不能启用这个选项，因为 PostgreSQL 无法解密返回的字符串，也就无法确定实际用户名。

20.9. Peer 认证

Peer 认证通过从内核获取客户端的操作系统用户名，并把它用作被允许的数据库用户名（可结合可选的用户名映射）来工作。这种方法只支持本地连接。

peer 支持下列配置选项：

`map`

允许在系统用户名和数据库用户名之间进行映射。详见第 20.2 节。

Peer 认证只在提供 `getpeereid()` 函数、`SO_PEERCRED` 套接字参数或类似机制的操作系统上可用。目前这包括 Linux、大多数 BSD 变种（包括 macOS）以及 Solaris。

20.10. LDAP 认证

这种认证方法的工作方式与 `password` 类似，只不过它使用 LDAP 作为密码验证方法。LDAP 只用于验证用户名/密码对。因此，在使用 LDAP 进行认证之前，用户必须已经存在于数据库中。

LDAP 认证可以在两种模式下工作。第一种模式称为简单绑定模式，服务器会绑定到按 *prefix* *username* *suffix* 形式构造出的可分辨名称。通常，*prefix* 参数用于指定 `cn=`，或在 Active Directory 环境中指定 *DOMAIN*\。*suffix* 则用于指定非 Active Directory 环境中 DN 的剩余部分。

第二种模式称为搜索+绑定模式，服务器首先使用由 *ldapbinddn* 和 *ldapbindpasswd* 指定的固定用户名和密码绑定到 LDAP 目录，并搜索试图登录数据库的用户。如果没有配置用户名和密码，则会尝试对目录进行匿名绑定。搜索会在 *ldapbasedn* 指定的子树上进行，并尝试对 *ldapsearchattribute* 指定的属性做精确匹配。一旦在搜索中找到了该用户，服务器会断开连接，再作为该用户重新绑定到目录，并使用客户端指定的密码来验证登录是否正确。这种模式

[2] https://datatracker.ietf.org/doc/html/rfc1413

与 Apache `mod_authnz_ldap` 和 `pam_ldap` 等软件中的 LDAP 认证方案相同。这种方法使目录中用户对象的位置更具灵活性，但会与 LDAP 服务器建立两个独立的连接。

以下配置选项在两种模式下都使用：

`ldapserver`

要连接的 LDAP 服务器的名称或 IP 地址。可以指定多个服务器，用空格分隔。

`ldapport`

要连接的 LDAP 服务器的端口号。如果未指定端口，则将使用 LDAP 库的默认端口设置。

`ldapscheme`

设置为 `ldaps` 以使用 LDAPS。这是一种非标准的通过 SSL 使用 LDAP 的方式，受一些 LDAP 服务器实现支持。另请参阅`ldaptls` 选项作为替代。

`ldaptls`

设置为 1 以使 PostgreSQL 和 LDAP 服务器之间的连接使用 TLS 加密。这使用 `StartTLS` 操作，参见RFC 4513[3]。另请参阅 `ldapscheme` 选项作为替代。

注意使用 `ldapscheme` 或 `ldaptls` 仅会加密 PostgreSQL 服务器和 LDAP 服务器之间的通信。PostgreSQL 服务器和 PostgreSQL 客户端之间的连接仍是未加密的，除非也在其上使用 SSL。

下列选项只被用于简单绑定模式：

`ldapprefix`

在进行简单绑定认证时，附加到用户名前面以形成绑定 DN 的字符串。

`ldapsuffix`

在进行简单绑定认证时，附加到用户名后面以形成绑定 DN 的字符串。

以下选项仅在搜索+绑定模式中使用：

`ldapbasedn`

在进行搜索+绑定认证时，用作用户搜索起点的根 DN。

`ldapbinddn`

在进行搜索+绑定认证时，用于绑定到目录并执行搜索的用户 DN。

`ldapbindpasswd`

在进行搜索+绑定认证时，用于绑定到目录并执行搜索的用户密码。

`ldapsearchattribute`

在进行搜索+绑定认证时，用于与用户名匹配的属性。如果未指定属性，则会使用 `uid` 属性。

[3] https://datatracker.ietf.org/doc/html/rfc4513

`ldapsearchfilter`

在进行搜索+绑定认证时使用的搜索过滤器。其中出现的 `$username` 会被替换为用户名。这使得搜索过滤器比 `ldapsearchattribute` 更灵活。

`ldapurl`

符合 RFC 4516 的 LDAP URL。这是另一种指定部分 LDAP 选项的方式，写法更紧凑、更标准。其格式为

```
ldap[s]://host[:port]/basedn[?[attribute][?[scope][?[filter]]]]
```

scope 必须是以下值之一：`base`，`one`，`sub`，通常使用最后一个。（默认值为 `base`，通常不适用于这里的用途。）*attribute* 可以指定单个属性，并用它设置 `ldapsearchattribute`。如果 *attribute* 为空，则可以用 *filter* 设置 `ldapsearchfilter`。

URL 方案 `ldaps` 会选择 LDAPS 方法，通过 SSL 建立 LDAP 连接，等同于使用 `ldapscheme=ldaps`。要通过 `StartTLS` 操作使用加密的 LDAP 连接，请使用普通的 URL 方案 `ldap`，并在 `ldapurl` 之外指定 `ldaptls` 选项。

对于非匿名绑定，必须将`ldapbinddn` 和 `ldapbindpasswd` 指定为单独的选项。

目前只有 OpenLDAP 支持 LDAP URL，Windows 不支持。

将简单绑定模式的配置选项与搜索+绑定模式的配置选项混用是错误的。

在使用搜索+绑定模式时，可以使用由 `ldapsearchattribute` 指定的单个属性执行搜索，也可以使用由 `ldapsearchfilter` 指定的自定义搜索过滤器执行搜索。指定 `ldapsearchattribute=foo` 等价于指定 `ldapsearchfilter="(foo=$username)"`。如果两个选项都未指定，则默认使用 `ldapsearchattribute=uid`。

下面是一个简单绑定 LDAP 配置示例：

```
host ... ldap ldapserver=ldap.example.net ldapprefix="cn=" ldapsuffix=", dc=example, dc=net"
```

当请求以数据库用户 `someuser` 连接数据库服务器时，PostgreSQL 将尝试使用 DN `cn=someuser, dc=example, dc=net` 和客户端提供的密码绑定到 LDAP 服务器。如果该连接成功，数据库访问就会被授予。

下面是搜索加绑定配置的示例：

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net" ldapsearchattribute=
uid
```

当请求以数据库用户 `someuser` 的身份连接数据库服务器时，PostgreSQL 会尝试匿名绑定到 LDAP 服务器（因为没有指定 `ldapbinddn`），并在指定的基础 DN 下搜索 `(uid=someuser)`。如果找到了条目，就会尝试使用找到的信息和客户端提供的密码进行绑定。如果第二次连接成功，就会授予数据库访问权限。

下面是以 URL 形式写出的同一个搜索+绑定配置：

```
host ... ldap ldapurl="ldap://ldap.example.net/dc=example,dc=net?uid?sub"
```

某些支持 LDAP 认证的其他软件也使用相同的 URL 格式，因此共享这类配置会更容易。

这里是一个搜索+绑定配置的示例，它使用 `ldapsearchfilter` 而不是 `ldapsearchattribute` 来允许用用户 ID 或电子邮件地址进行认证：

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net" ldapsearchfilter="(|(
uid=$username)(mail=$username))"
```

> **提示**
>
> 如示例中所示，由于 LDAP 通常使用逗号和空格来分割一个 DN 的不同部分，在配置 LDAP 选项时通常有必要使用双引号包围的参数值。

20.11. RADIUS 认证

这种认证方法的工作方式与 `password` 类似，只不过它使用 RADIUS 作为密码验证方式。RADIUS 只用于验证用户名/密码对。因此，在使用 RADIUS 进行认证之前，用户必须已经存在于数据库中。

使用 RADIUS 认证时，会向配置好的 RADIUS 服务器发送一条 Access Request 消息。该请求的类型为 `Authenticate Only`，并包含 `user name`、`password`（加密的）以及 `NAS Identifier` 参数。该请求会使用与服务器共享的密钥进行加密。RADIUS 服务器会返回 `Access Accept` 或 `Access Reject` 作为响应。PostgreSQL 不支持 RADIUS 记账。

可以指定多个 RADIUS 服务器，这种情况下会按顺序依次尝试。如果某台服务器返回否定响应，则认证失败；如果没有收到响应，则继续尝试列表中的下一台服务器。要指定多台服务器，可用双引号括起整个列表，并用逗号分隔服务器名称。如果指定了多台服务器，其他 RADIUS 选项也可以写成逗号分隔的列表，以便为每台服务器分别提供不同取值；也可以只指定单个值，此时该值会应用到所有服务器。

RADIUS 支持下列配置选项：

`radiusservers`

要连接的 RADIUS 服务器的 DNS 名称或 IP 地址。此参数为必需项。

`radiussecrets`

与 RADIUS 服务器进行安全通信时使用的共享密钥。它在 PostgreSQL 服务器和 RADIUS 服务器上必须完全相同。建议它至少是一个 16 个字符长的字符串。此参数为必需项。

> **注意**
>
> 只有当 PostgreSQL 在构建时启用了 OpenSSL 支持，所使用的加密向量才具有足够的密码学强度。在其他情况下，到 RADIUS 服务器的传输只能被视为经过混淆，而非受到安全保护；如有必要，应额外采取外部安全措施。

`radiusports`

要连接的 RADIUS 服务器端口号。如果未指定端口，则会使用默认的 RADIUS 端口 `1812`。

`radiusidentifiers`

在 RADIUS 请求中用作 `NAS Identifier` 的字符串。举例来说，这个参数可用于标识用户正在尝试连接哪个数据库集簇，从而便于在 RADIUS 服务器上进行策略匹配。如果未指定标识符，则默认使用 `postgresql`。

如果某个 RADIUS 参数值中必须包含逗号或空白，可以通过用双引号括起该值来实现，但这样会比较繁琐，因为此时需要两层双引号。下面是一个在 RADIUS 密钥字符串中包含空白的示例：

```
host ... radius radiusservers="server1,server2" radiussecrets="""secret one"",""secret two"""
```

20.12. 证书认证

这种认证方法使用 SSL 客户端证书来完成认证。因此，它仅适用于 SSL 连接；关于 SSL 的配置说明，见第 18.9.2 节。使用此认证方法时，服务器会要求客户端提供有效且受信任的证书。不会向客户端发送密码提示。证书的 `cn`（通用名称）属性将与请求的数据库用户名进行比较，如果它们匹配，则允许登录。用户名映射可用于允许 `cn` 与数据库用户名不同。

SSL 证书认证支持下列配置选项：

`map`

允许在系统用户名和数据库用户名之间进行映射。详见第 20.2 节。

在指定证书认证的 `pg_hba.conf` 记录中，认证选项 `clientcert` 被视为 `1`，且不能关闭，因为此方法必须使用客户端证书。`cert` 方法在基本的 `clientcert` 证书有效性检查之外，还会检查 `cn` 属性是否与数据库用户名匹配。

20.13. PAM 认证

这种认证方法的工作方式与 `password` 类似，只不过它使用 PAM（可插拔认证模块）作为认证机制。默认的 PAM 服务名是 `postgresql`。PAM 只用于验证用户名/密码对，以及可选地验证已连接远程主机的主机名或 IP 地址。因此，在使用 PAM 进行认证之前，用户必须已经存在于数据库中。关于 PAM 的更多信息，请阅读 Linux-PAM 页面[4]。

PAM 支持下列配置选项：

`pamservice`

PAM 服务名称。

`pam_use_hostname`

决定是通过 `PAM_RHOST` 项向 PAM 模块提供远程 IP 地址还是主机名。默认情况下使用 IP 地址。将此选项设为 1 可改为使用解析出的主机名。主机名解析可能导致登录延迟。（大多数 PAM 配置并不会使用这项信息，因此只有在 PAM 配置被专门设计为利用该信息时，才需要考虑这个设置。）

[4] https://www.kernel.org/pub/linux/libs/pam/

> **注意**
>
> 如果 PAM 被设置为读取 /etc/shadow，认证将会失败，因为 PostgreSQL 服务器是由非 root 用户启动的。不过，当 PAM 被配置为使用 LDAP 或其他认证方法时，这就不是问题。

20.14. BSD 认证

这种认证方法操作起来类似于 password，不过它使用 BSD 认证来验证密码。BSD 认证只被用来验证用户名/密码对。因此，在 BSD 认证可以被用于认证之前，用户的角色必须已经存在于数据库中。BSD 认证框架当前只在 OpenBSD 上可用。

PostgreSQL 中的 BSD 认证使用 auth-postgresql 登录类型，如果login.conf 中定义了 postgresql 登录分类，就会用它来认证。默认情况下这种登录分类不存在，PostgreSQL 将使用默认的登录分类。

> **注意**
>
> 要使用 BSD 认证，PostgreSQL 用户账号（也就是运行服务器的操作系统用户）必须首先被加入到auth 组中。在 OpenBSD 系统上默认存在 auth 组。

20.15. 认证问题

认证失败及相关问题通常会表现为类似下面这样的错误消息：

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym", database "testdb"
```

这种情况通常意味着你已经成功联系到了服务器，但服务器不愿意接受你的连接。正如消息所示，服务器拒绝了该连接请求，因为它没有在自己的 pg_hba.conf 配置文件中找到匹配项。

```
FATAL:  password authentication failed for user "andym"
```

这样的消息表示你已经联系到了服务器，而且服务器也愿意继续处理连接，但前提是你必须先通过 pg_hba.conf 文件中指定的认证方法。请检查你提供的密码；如果错误消息提到了 Kerberos 或 ident 等认证类型，也请检查对应的软件配置。

```
FATAL:  user "andym" does not exist
```

指定的数据库用户不存在。

```
FATAL:  database "testdb" does not exist
```

你试图连接的数据库不存在。注意，如果你没有指定数据库名，默认会使用数据库用户名作为数据库名，但这不一定是所需的数据库名。

提示

服务器日志中可能包含比返回给客户端的更多认证失败信息。如果你不清楚失败原因，请检查服务器日志。

第 21 章 数据库角色

PostgreSQL 使用角色这一概念来管理数据库访问权限。角色可以被看作数据库用户，也可以被看作一组数据库用户，这取决于该角色的设置方式。角色可以拥有数据库对象（例如表和函数），并且可以把这些对象上的权限赋予其他角色，以控制谁能访问哪些对象。此外，还可以把一个角色中的成员资格授予另一个角色，从而允许成员角色使用授予该角色的权限。

角色这一概念涵盖了“用户”和“组”这两个概念。在 PostgreSQL 8.1 之前的版本中，用户和组是两类不同的实体，但现在只有角色。任意角色都可以充当用户、组，或者同时充当两者。

本章描述如何创建和管理角色。有关角色权限对各种数据库对象影响的更多信息，见第 5.6 节。

21.1. 数据库角色

数据库角色在概念上与操作系统用户完全分离。在实践中，维持两者之间的对应关系可能比较方便，但这并非必需。数据库角色在整个数据库集簇安装中是全局的（而非每个数据库各自独立）。要创建角色，请使用CREATE ROLE SQL 命令：

```
CREATE ROLE name;
```

name 遵循 SQL 标识符规则：要么不加修饰且不含特殊字符，要么用双引号括起。（在实践中，通常还需要为该命令添加其他选项，例如 LOGIN。下文会给出更多细节。）要移除现有角色，请使用类似的DROP ROLE 命令：

```
DROP ROLE name;
```

为了方便，提供了createuser 和 dropuser 程序，作为这些 SQL 命令的包装器，可以从 shell 命令行调用：

```
createuser name
dropuser name
```

要确定现有角色的集合，请查看 pg_roles 系统目录，例如

```
SELECT rolname FROM pg_roles;
```

使用 psql 程序的 \du 元命令也可以列出现有角色。

为了完成数据库系统的初始引导，一个刚初始化的系统总会包含一个预定义角色。该角色总是一个“超级用户”，并且除非另行指定了不同名称，否则它的名称与用initdb 初始化数据库集簇的操作系统用户相同。这个角色通常命名为 postgres。要创建更多角色，必须先以这个初始角色建立连接。

每个到数据库服务器的连接都是以某个特定角色名建立的，而这个角色决定在该连接中发出命令时的初始访问权限。某个数据库连接要使用的角色名，由发起连接请求的客户端以应用特定的方式指明。例如，psql 程序使用 -U 命令行选项来指明要以哪个角色连接。很多应用默认假定使用当前操作系统用户的名字（包括 createuser 和 psql）。因此，在角色名和操作系统用户名之间维护对应关系通常很方便。

某个客户端连接能够以哪些数据库角色进行连接，由客户端认证设置决定，如第 20 章中所述。（因此，客户端并不限于只能以与其操作系统用户匹配的角色连接，正如一个人的登录名不必与其真实姓名一致。）由于角色身份决定了已连接客户端可用的权限集合，所以在设置多用户环境时，仔细配置权限非常重要。

21.2. 角色属性

数据库角色可以具有多个属性，这些属性定义其权限，并与客户端认证系统交互。

登录权限

只有具有 LOGIN 属性的角色才能用作数据库连接的初始角色名。具有 LOGIN 属性的角色可以被视为“数据库用户”。要创建具有登录权限的角色，可使用以下任一方式：

```
CREATE ROLE name LOGIN;
CREATE USER name;
```

（CREATE USER 等价于 CREATE ROLE，不同之处只在于CREATE USER 默认包含 LOGIN，而CREATE ROLE 不包含。）

超级用户状态

数据库超级用户会绕过除登录权之外的所有权限检查。这是一种危险的权限，不应草率使用；最好以非超级用户角色完成大部分工作。要创建新的数据库超级用户，使用CREATE ROLE *name* SUPERUSER。这必须由已经是超级用户的角色来执行。

创建数据库

角色必须被明确授予创建数据库的权限（超级用户除外，因为后者会绕过所有权限检查）。要创建这样的角色，使用CREATE ROLE *name* CREATEDB。

创建角色

角色必须被明确授予创建更多角色的权限（超级用户除外，因为后者会绕过所有权限检查）。要创建这样的角色，使用CREATE ROLE *name* CREATEROLE。具有 CREATEROLE 权限的角色，也可以更改和删除其他角色，并且可以授予或撤销其中的成员资格。更改角色包括使用ALTER ROLE 能进行的大多数修改，例如更改密码。它还包括使用COMMENT 和 SECURITY LABEL 命令对角色进行的修改。

然而，CREATEROLE 并不赋予创建 SUPERUSER 角色的能力，也不赋予对已经存在的 SUPERUSER 角色的任何控制权。此外，CREATEROLE 也不赋予创建 REPLICATION 用户的能力，也不能授予或撤销REPLICATION 权限，也不能修改这类用户的角色属性。不过，它允许在REPLICATION 角色上使用 ALTER ROLE ... SET 和 ALTER ROLE ... RENAME，以及使用COMMENT ON ROLE、SECURITY LABEL ON ROLE 和 DROP ROLE。最后，CREATEROLE 也不赋予授予或撤销 BYPASSRLS 权限的能力。

由于 CREATEROLE 权限允许用户即使对某些角色（尚）没有任何访问权，也能授予或撤销其中的成员资格，因此CREATEROLE 用户可以获得系统中每个预定义角色的能力，包括pg_execute_server_program 和 pg_write_server_files 这类高权限角色。

发起复制

角色必须被明确授予发起流复制的权限（超级用户除外，因为后者会绕过所有权限检查）。用于流复制的角色还必须具有LOGIN 权限。要创建这样的角色，使用CREATE ROLE *name* REPLICATION LOGIN。

密码

只有在客户端认证方法要求用户连接数据库时提供密码时，密码才有意义。password 和 md5 认证方法都会使用密码。数据库密码与操作系统密码是分离的。可在创建角色时通过 CREATE ROLE *name* PASSWORD '*string*' 指定密码。

权限继承

默认情况下，角色会继承其所属角色的权限。不过，要创建一个默认不继承权限的角色，使用CREATE ROLE *name* NOINHERIT。

绕过行级安全

角色必须被明确授予绕过每一条行级安全（RLS）策略的权限（超级用户除外，因为后者会绕过所有权限检查）。要创建这样的角色，必须以超级用户身份使用CREATE ROLE *name* BYPASSRLS。

连接限制

连接限制可以指定一个角色能够建立多少个并发连接。-1（默认值）表示无限制。可在创建角色时通过 CREATE ROLE *name* CONNECTION LIMIT '*integer*' 指定连接限制。

角色创建后，可以使用以下命令修改其属性：ALTER ROLE。详情请参见 CREATE ROLE 和 ALTER ROLE 命令的参考页。

角色还可以为第 19 章中描述的许多运行时配置设置指定角色特定默认值。例如，如果出于某种原因你希望每次连接时都禁用索引扫描（提示：这不是个好主意），你可以使用：

```
ALTER ROLE myname SET enable_indexscan TO off;
```

这会保存该设置（但不会立即生效）。在该角色后续建立的连接中，它看起来就像在会话开始之前执行了SET enable_indexscan TO off 一样。你仍然可以在会话期间更改该设置；它只会作为默认值。要移除角色特定默认设置，使用ALTER ROLE *rolename* RESET *varname*。注意，附加到没有LOGIN 权限的角色上的角色特定默认值几乎没有用，因为它们永远不会生效。

21.3. 角色成员资格

为了简化权限管理，把用户分组通常很方便：这样，权限就可以整体授予给一个组，或者从整个组中撤销。在PostgreSQL 中，这通过创建一个表示该组的角色，然后把该组角色中的成员资格授予各个用户角色来实现。

要建立一个组角色，首先创建该角色：

```
CREATE ROLE name;
```

通常，作为组使用的角色不会有LOGIN 属性，不过如果你愿意，也可以设置它。

组角色存在后，就可以使用GRANT 和 REVOKE 命令添加和移除成员：

```
GRANT group_role TO role1, ... ;
REVOKE group_role FROM role1, ... ;
```

也可以把成员资格授予其他组角色（因为组角色与非组角色实际上没有任何区别）。数据库不允许建立循环成员关系。此外，也不允许将角色的成员资格授予 PUBLIC。

组角色的成员可以通过两种方式使用该角色的权限。首先，组的每个成员都可以显式执行 SET ROLE，以暂时“成为”该组角色。在这种状态下，数据库会话可以使用组角色的权限，而不是原始登录角色的权限；创建的任何数据库对象也被视为由组角色而非登录角色拥有。其次，具有 INHERIT 属性的成员角色可以自动使用其所属角色的权限，包括这些角色继承的所有权限。例如，假定我们已经执行了：

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

以角色 joe 连接后，数据库会话立即可以使用直接授予 joe 的权限，以及授予 admin 的所有权限，因为 joe “继承” admin 的权限。不过，授予 wheel 的权限不可用，因为即使 joe 间接属于 wheel，该成员资格也是通过 admin 获得的，而后者具有 NOINHERIT 属性。执行以下命令后：

```
SET ROLE admin;
```

会话将只能使用授予 admin 的权限，而不能使用授予 joe 的权限。执行以下命令后：

```
SET ROLE wheel;
```

会话将只能使用授予 wheel 的权限，而不能使用授予 joe 或 admin 的权限。使用以下任一命令可以恢复原始的权限状态：

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

注意

SET ROLE 命令总是允许切换到原始登录角色直接或间接所属的任何角色。因此，在上例中，没有必要先成为admin 再成为 wheel。

注意

在 SQL 标准中，用户与角色之间有明确区别，并且用户不会自动继承权限，而角色会。这种行为可以在PostgreSQL 中通过让用作 SQL 角色的角色具有 INHERIT 属性，而让用作

> SQL 用户的角色具有NOINHERIT 属性来获得。不过，出于与 8.1 之前版本向后兼容的考虑，PostgreSQL 默认给所有角色都赋予 INHERIT 属性；在那些早期版本中，用户总能使用授予其所属组的权限。

角色属性 LOGIN、SUPERUSER、CREATEDB 和 CREATEROLE 可以被视为特殊权限，但它们从不会像数据库对象上的普通权限那样被继承。要使用其中之一，必须实际执行SET ROLE 切换到拥有该属性的特定角色。继续上面的例子，我们可以选择把CREATEDB 和 CREATEROLE 赋予 admin 角色。这样，一个以joe 角色连接的会话并不会立即拥有这些权限，只有在执行 SET ROLE admin 之后才会拥有。

要删除组角色，请使用 DROP ROLE：

```
DROP ROLE name;
```

该组角色中的所有成员资格都会自动被撤销（但成员角色不会受到其他影响）。

21.4. 删除角色

由于角色可以拥有数据库对象，并且可以拥有访问其他对象的权限，所以删除一个角色通常不只是简单执行一次 DROP ROLE。必须先删除或重新分配该角色拥有的任何对象；并且必须撤销授予该角色的任何权限。

可以使用 ALTER 命令逐个转移对象的所有权，例如：

```
ALTER TABLE bobs_table OWNER TO alice;
```

另一种办法是使用 REASSIGN OWNED 命令，把待删除角色拥有的所有对象的所有权重新分配给另一个角色。由于 REASSIGN OWNED 无法访问其他数据库中的对象，因此必须在包含该角色所拥有对象的每个数据库中运行它。（请注意，首次执行这样的 REASSIGN OWNED 时，会更改待删除角色所拥有的所有跨数据库共享对象（即数据库或表空间）的所有权。）

一旦有价值的对象都已转移给新拥有者，待删除角色所拥有的其余对象即可使用DROP OWNED 命令删除。同样，由于该命令不能访问其他数据库中的对象，因此必须在包含该角色所拥有对象的每个数据库中运行它。另外，DROP OWNED 不会删除整个数据库或表空间，因此如果该角色拥有任何尚未转移给新拥有者的数据库或表空间，就必须手工删除它们。

DROP OWNED 还会负责移除目标角色在其他角色所拥有的对象上获授的所有权限。由于 REASSIGN OWNED 不会处理这类对象，所以通常需要同时运行REASSIGN OWNED 和 DROP OWNED（按这个顺序！），才能完整移除待删除角色的依赖关系。

总之，移除曾用来拥有对象的角色，最通用的做法是：

```
REASSIGN OWNED BY doomed_role TO successor_role;
DROP OWNED BY doomed_role;
-- repeat the above commands in each database of the cluster
DROP ROLE doomed_role;
```

如果并非所有拥有的对象都要转移给同一个后继拥有者，最好先手工处理这些例外情况，然后再执行上述步骤作最后清理。

如果在仍有依赖对象存在时尝试执行 DROP ROLE，它会发出消息指出哪些对象需要被重新分配或删除。

21.5. 默认角色

PostgreSQL 提供一组默认角色，用于访问某些常用的特权功能和信息。管理员可以将这些角色 GRANT 给其环境中的用户和/或其他角色，使这些用户能够访问指定的功能和信息。

默认角色在表 21.1 中介绍。请注意，随着新功能的加入，各默认角色的具体权限可能会在将来发生变化。管理员应关注发行说明中的变更。

表 21.1. 默认角色

角色	允许的访问
pg_read_all_settings	读取所有配置变量，即使是那些通常仅对超级用户可见的变量。
pg_read_all_stats	读取所有 pg_stat_* 视图，并使用各种与统计信息相关的扩展，包括通常仅对超级用户可见的内容。
pg_stat_scan_tables	执行可能在表上持有ACCESS SHARE 锁并且持续较长时间的监控函数。
pg_monitor	读取或执行各种监控视图和函数。该角色是pg_read_all_settings、pg_read_all_stats 和 pg_stat_scan_tables 的成员。
pg_signal_backend	向另一个后端发送信号，以取消查询或终止其会话。
pg_read_server_files	允许使用 COPY 和其他文件访问函数，从服务器上数据库可访问的任何位置读取文件。
pg_write_server_files	允许使用 COPY 和其他文件访问函数，向服务器上数据库可访问的任何位置写入文件。
pg_execute_server_program	允许使用 COPY 和其他允许执行服务器端程序的函数，以数据库运行所用用户的身份在数据库服务器上执行程序。

pg_monitor、pg_read_all_settings、pg_read_all_stats 和 pg_stat_scan_tables 角色旨在让管理员能够方便地为监控数据库服务器配置一个角色。它们授予一组常见权限，使该角色能够读取各种有用的配置设置、统计信息以及其他通常仅限超级用户查看的系统信息。

pg_signal_backend 角色旨在让管理员能够允许受信任但非超级用户的角色向其他后端发送信号。目前这个角色允许发送信号来取消另一个后端上的查询或终止其会话。不过，被授予此角色的用户不能向由超级用户拥有的后端发送信号。见第 9.26.2 节。

pg_read_server_files、pg_write_server_files 和 pg_execute_server_program 角色旨在让管理员能够使受信任但非超级用户的角色能够以数据库运行所用用户的身份访问文件并在数据库服务器上运行程序。由于这些角色能够访问服务器文件系统上的任意文件，它们在直接访问文件时会绕过所有数据库级权限检查，并且可被用来获得超级用户级别的访问能力，因此向用户授予这些角色时必须极其谨慎。

授予这些角色时应当谨慎，确保它们只在需要时使用，并理解这些角色会授予对需要特权才能访问的信息的访问。

管理员可以使用 GRANT 命令将这些角色的访问权限授予用户，例如：

```
GRANT pg_signal_backend TO admin_user;
```

21.6. 函数安全性

函数、触发器和行级安全策略允许用户把代码插入后端服务器，而其他用户可能会无意中执行这些代码。因此，这些机制使用户能够相对容易地对其他用户实施“特洛伊木马”攻击。最强的保护方式是严格控制谁能定义对象。如果这不可行，则应编写只引用拥有者可信的对象的查询，并从 `search_path` 中移除 public 模式以及任何其他允许不受信任用户创建对象的模式。

函数在后端服务器进程中运行，并具有数据库服务器守护进程的操作系统权限。如果函数所用的编程语言允许不受检查的内存访问，就有可能更改服务器的内部数据结构。因此，这类函数可以做很多别的事情，其中就包括绕过任何系统访问控制。允许这种访问的函数语言被认为是“不受信任的”，而PostgreSQL 只允许超级用户创建用这类语言编写的函数。

第 22 章 管理数据库

每个正在运行的 PostgreSQL 服务器实例都管理着一个或多个数据库。因此，数据库是组织SQL对象（“数据库对象”）的最高层级。本章描述数据库的属性，以及如何创建、管理和删除数据库。

22.1. 概述

少量对象，例如角色名、数据库名和表空间名，是在集簇级别定义并存储在pg_global 表空间中的。集簇内部有多个数据库，它们彼此隔离，但可以访问集簇级对象。每个数据库内部又有多个模式，其中包含表、函数等对象。因此，完整的层次结构是：集簇、数据库、模式、表（或者函数等其他类型的对象）。

当连接到数据库服务器时，客户端必须在连接请求中指定数据库名。一个连接无法访问多个数据库。不过，客户端可以对同一个数据库建立多个连接，也可以连接到不同的数据库。数据库级安全由两部分组成：访问控制（见第 20.1 节），在连接级管理；授权控制（见第 5.6 节），通过 GRANT 系统管理。外部数据包装器（见 postgres_fdw）允许一个数据库中的对象充当其他数据库或集簇中对象的代理。较早的 dblink 模块（见 dblink）也提供了类似能力。默认情况下，所有用户都可以使用所有连接方法连接到所有数据库。

如果一个 PostgreSQL 服务器集簇计划容纳彼此无关、在大多数情况下也不需要感知彼此存在的项目或用户，建议将它们放入不同的数据库，并相应地调整授权和访问控制。如果这些项目或用户彼此相关，因此应当能够使用彼此的资源，那么它们应当位于同一个数据库中，但通常应放入不同的模式；这能提供具有名字空间隔离和授权控制的模块化结构。关于管理模式的更多信息见第 5.8 节。

虽然可以在单个集簇中创建多个数据库，但仍建议仔细权衡其收益是否超过风险和限制。尤其要考虑共享 WAL（见第 30 章）对备份和恢复选项的影响。虽然从用户角度看，集簇中的各个数据库彼此隔离，但从数据库管理员的角度看，它们又是紧密绑定的。

数据库使用 CREATE DATABASE 命令创建（见第 22.2 节），使用DROP DATABASE 命令删除（见第 22.5 节）。要确定现有数据库的集合，可以检查系统目录pg_database，例如：

```
SELECT datname FROM pg_database;
```

psql 程序的\l 元命令和 -l 命令行选项也可用于列出现有数据库。

> **注意**
>
> SQL 标准将数据库称为“目录”，但在实践中两者没有区别。

22.2. 创建数据库

要创建数据库，PostgreSQL 服务器必须已经启动并正在运行（见第 18.3 节）。

数据库用 SQL 命令 CREATE DATABASE 创建：

```
CREATE DATABASE name;
```

其中 *name* 遵循 SQL 标识符的一般规则。当前角色会自动成为新数据库的拥有者。数据库拥有者有权在日后删除该数据库（这也会删除其中的全部对象，即使那些对象具有不同的拥有者）。

创建数据库是一项受限操作。关于如何授予该权限，见第 21.2 节。

由于执行 CREATE DATABASE 命令时必须先连接到数据库服务器，因此还会有一个问题：某个站点上的第一个数据库是如何创建的？第一个数据库总是由 initdb 命令在初始化数据存储区时创建的（见第 18.2 节）。这个数据库叫作 postgres。因此，要创建第一个“普通”数据库，可以连接到postgres。

在初始化数据库集簇时，还会创建第二个数据库template1。每当在集簇中创建一个新数据库时，新数据库本质上都是通过克隆template1 得到的。这意味着你在 template1 中所做的任何更改，都会传播到后来创建的所有数据库中。因此，除非你希望这些对象出现在每一个新建数据库中，否则应避免在template1 中创建对象。更多细节见第 22.3 节。

为了方便，还可以从 shell 中执行一个程序来创建新数据库，即createdb。

```
createdb dbname
```

createdb 并没有什么特殊之处。它连接到 postgres 数据库并发出 CREATE DATABASE 命令，和上面描述的完全一样。关于调用细节见createdb 参考页。注意，不带任何参数的 createdb 会创建一个与当前用户名同名的数据库。

> **注意**
>
> 第 20 章包含关于如何限制谁能连接到给定数据库的信息。

有时你会想为其他人创建数据库，并让他们成为新数据库的拥有者，以便他们自行配置和管理它。要实现这一点，可使用下列命令之一：

```
CREATE DATABASE dbname OWNER rolename;
```

这是在 SQL 环境中使用；或者：

```
createdb -O rolename dbname
```

这是在 shell 中使用。只有超级用户才允许为其他人（即为一个你不是其成员的角色）创建数据库。

22.3. 模板数据库

CREATE DATABASE 实际上是通过复制一个现有数据库来创建新数据库。默认情况下，它复制名为template1 的标准系统数据库。因此，该数据库就是创建新数据库时使用的“模板”。如果你向 template1 中添加对象，这些对象也会被复制到以后创建的用户数据库中。这种行为

允许对数据库中的标准对象集合进行站点本地修改。例如，如果你将过程语言PL/Perl 安装到 template1 中，那么以后创建用户数据库时无需额外操作，它就会自动可用。

然而，CREATE DATABASE 不会复制附着在源数据库上的数据库级 GRANT 权限。新数据库将具有默认的数据库级权限。

还有第二个标准系统数据库，名为template0。该数据库包含与 template1 初始内容相同的数据，也就是说，只包含你的PostgreSQL 版本预定义的标准对象。在数据库集簇初始化之后，template0 就不应再被修改。通过指示 CREATE DATABASE 复制 template0 而不是 template1，你可以创建一个“纯净的”用户数据库，它不包含template1 中的任何站点本地附加对象。这在恢复 pg_dump 转储时特别方便：转储脚本应当恢复到一个纯净的数据库中，以确保重建出被转储数据库的正确内容，而不会与后来可能加入template1 的对象发生冲突。

复制 template0 而不是 template1 的另一个常见原因是：复制 template0 时可以指定新的编码和区域设置，而template1 的副本则必须使用与它相同的设置。这是因为 template1 可能包含依赖特定编码或区域设置的数据，而已知template0 不会包含这类数据。

要通过复制 template0 创建数据库，可在 SQL 环境中使用：

```
CREATE DATABASE dbname TEMPLATE template0;
```

或在 shell 中使用：

```
createdb -T template0 dbname
```

可以创建额外的模板数据库，实际上也可以通过将集簇中任意数据库的名称指定为 CREATE DATABASE 的模板来复制它。不过，必须理解的是，这（至少目前）并不是一个通用的“COPY DATABASE”功能。最主要的限制是：复制源数据库时，不能有其他会话连接到该数据库。如果CREATE DATABASE 开始时存在任何其他连接，该命令就会失败；在复制操作期间，也会阻止到源数据库的新连接。

对每个数据库，pg_database 中都有两个有用的标志列：datistemplate 和 datallowconn。datistemplate 可设置为表明某个数据库将作为 CREATE DATABASE 的模板。如果设置了该标志，任何拥有CREATEDB 权限的用户都可以克隆该数据库；如果未设置，则只有超级用户和数据库拥有者才能克隆它。如果datallowconn 为假，则不允许建立到该数据库的新连接（但仅仅把该标志设为假并不会终止已有会话）。为了防止被修改，template0 数据库通常被标记为 datallowconn = false。template0 和 template1 始终都应标记为 datistemplate = true。

> **注意**
>
> template1 和 template0 除了 template1 这个名字是 CREATE DATABASE 默认源数据库名这一点之外，并无任何特殊地位。例如，可以删除template1，然后再从 template0 重新创建它，而不会有任何不良后果。如果有人不小心在template1 里加入了许多杂项对象，这样做也许是可取的。（要删除template1，必须先令其具有pg_database.datistemplate = false。）
>
> 初始化数据库集簇时也会创建 postgres 数据库。这个数据库被用作用户和应用连接时的默认数据库。它只是 template1 的一个副本，因此在必要时也可以删除并重建。

22.4. 数据库配置

正如第 19 章所述，PostgreSQL 服务器提供了大量运行时配置变量。你可以为其中许多设置指定数据库特定的默认值。

例如，如果你因为某种原因想在某个数据库中禁用GEQO 优化器，通常要么必须对所有数据库都禁用它，要么必须确保每个连入的客户端都会执行`SET geqo TO off`。若要让这个设置在某个特定数据库中成为默认值，可以执行如下命令：

```
ALTER DATABASE mydb SET geqo TO off;
```

这会保存该设置（但不会立即生效）。以后连接到该数据库时，它看起来就像是在会话开始前刚执行过`SET geqo TO off;`一样。注意，用户仍然可以在自己的会话中更改该设置；它只会作为默认值。要撤销这样的设置，可使用`ALTER DATABASE` *`dbname`* `RESET` *`varname`*。

22.5. 删除数据库

数据库使用 DROP DATABASE 命令删除：

```
DROP DATABASE name;
```

只有数据库拥有者或超级用户可以删除数据库。删除数据库会移除其中包含的全部对象。数据库的删除无法撤销。

当你连接到目标数据库时，不能执行`DROP DATABASE` 命令。不过，你可以连接到任何其他数据库，包括`template1` 数据库。若要删除某个集簇中的最后一个用户数据库，`template1` 将是唯一的选择。

为了方便，还有一个 shell 程序可用于删除数据库，即 dropdb。

```
dropdb dbname
```

（与 `createdb` 不同，默认操作并不是删除与当前用户名同名的数据库。）

22.6. 表空间

PostgreSQL 中的表空间允许数据库管理员在文件系统中定义存放表示数据库对象的文件的位置。表空间一旦创建，在创建数据库对象时就可以通过名称引用它。

通过使用表空间，管理员可以控制PostgreSQL 安装的磁盘布局。这至少有两方面的用途。首先，如果初始化集簇所在的分区或卷空间耗尽且无法扩展，可以在另一个分区上创建表空间并使用它，直到系统能够重新配置。

其次，表空间允许管理员利用对数据库对象使用模式的了解来优化性能。例如，一个使用非常频繁的索引可以放在速度非常快且高可用的磁盘上，例如昂贵的固态设备。同时，一个存放归档数据且很少使用或对性能并不关键的表，则可以放在成本更低、速度更慢的磁盘系统上。

> **警告**
>
> 即使位于 PostgreSQL 主数据目录之外，表空间仍然是数据库集簇不可分割的一部分，不能被视为一个独立的数据文件集合。它们依赖于主数据目录中的元数据，因此不能附加到不同的数据库集簇，也不能单独备份。同样，如果丢失一个表空间（文件被删除、磁盘故

障等），数据库集簇可能会变得不可读或者无法启动。把表空间放在 RAM 磁盘这类临时文件系统上，会危及整个集簇的可靠性。

要定义一个表空间，使用CREATE TABLESPACE 命令，例如：

```
CREATE TABLESPACE fastspace LOCATION '/ssd1/postgresql/data';
```

该位置必须是一个已经存在、为空并且由PostgreSQL 操作系统用户拥有的目录。随后在这个表空间中创建的所有对象都会存储在该目录下的文件中。该位置不得位于可移动或瞬态存储上，因为如果表空间缺失或丢失，集簇可能无法正常工作。

注意

通常来说，在每个逻辑文件系统上创建多个表空间意义不大，因为你无法控制单个文件在一个逻辑文件系统中的具体位置。不过，PostgreSQL 并不强制这种限制，而且它实际上并不能直接感知你的系统中文件系统的边界。它只是在你指定的目录中存储文件。

表空间本身必须由数据库超级用户创建，但之后你可以允许普通数据库用户使用它。为此，需要向他们授予其上的CREATE 权限。

表、索引以及整个数据库都可以被分配到特定的表空间。为此，对给定表空间拥有CREATE 权限的用户必须把表空间名作为参数传给相应命令。例如，下列命令会在表空间space1 中创建一个表：

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

另一种方式是使用 default_tablespace 参数：

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

当 default_tablespace 被设置为非空字符串时，它会为未显式指定TABLESPACE 子句的 CREATE TABLE 和 CREATE INDEX 命令隐含地补上该子句。

还有一个 temp_tablespaces 参数，用来决定临时表和索引以及为大数据集排序等用途所使用的临时文件应放置在哪里。它可以是一个表空间名列表，而不只是单个表空间，这样与临时对象相关的负载就可以分散到多个表空间上。每次创建临时对象时，都会从该列表中随机挑选一个成员。

与数据库关联的表空间用于存储该数据库的系统目录。此外，如果没有给出TABLESPACE 子句，并且default_tablespace 或 temp_tablespaces（视情况而定）也没有指定其他选择，那么它还是在该数据库中创建的表、索引和临时文件所使用的默认表空间。如果创建数据库时没有为其指定表空间，它将使用其所复制模板数据库的同一个表空间。

初始化数据库集簇时会自动创建两个表空间。pg_global 表空间用于共享系统目录。pg_default 表空间是 template1 和 template0 数据库的默认表空间（因此，除非被 TABLESPACE 子句覆盖，而该子句可在CREATE DATABASE 中指定，否则它也会是其他数据库的默认表空间）。

表空间一旦创建，只要请求的用户具有足够权限，就可以从任何数据库中使用它。这意味着，只有当所有数据库中使用该表空间的对象都已被移除之后，才能删除该表空间。

要移除一个空表空间，使用DROP TABLESPACE 命令。

要确定现有表空间的集合，可以检查pg_tablespace 系统目录，例如：

```
SELECT spcname FROM pg_tablespace;
```

psql 程序的\db 元命令对列出现有表空间也很有用。

PostgreSQL 使用符号链接来简化表空间的实现。这意味着表空间只能在支持符号链接的系统上使用。

$PGDATA/pg_tblspc 目录包含符号链接，它们指向集簇中定义的每一个非内置表空间。虽然不推荐，但也可以通过重新定义这些链接来手工调整表空间布局。无论如何，都不要在服务器运行时执行此操作。注意，在 PostgreSQL 9.1 及更早版本中，还需要用新的位置更新 pg_tablespace 目录。（否则 pg_dump 将继续输出旧的表空间位置。）

第 23 章 本地化

本章从管理员的角度介绍可用的本地化特性。PostgreSQL 支持两类本地化机制：

- 利用操作系统的区域设置（locale）特性，提供与区域设置相关的排序顺序、数字格式、翻译后的消息以及其他方面的支持。详见第 23.1 节和第 23.2 节。
- 提供多种字符集，以支持存储各种语言的文本，并提供客户端与服务器之间的字符集转换。详见第 23.3 节。

23.1. 区域设置支持

区域设置支持是指应用程序在字母表、排序、数字格式等方面尊重文化偏好。PostgreSQL 使用服务器操作系统提供的标准 ISO C 和 POSIX 区域设置机制。更多信息请参考系统文档。

23.1.1. 概述

在使用 `initdb` 创建数据库集簇时，区域设置支持会自动初始化。默认情况下，`initdb` 会使用其执行环境中的区域设置来初始化数据库集簇，因此如果你的系统已经设置为数据库集簇想要使用的区域设置，就无需额外操作。如果想使用不同的区域设置（或者不确定系统当前设置了哪种区域设置），可以通过指定`--locale` 选项，精确告诉`initdb` 使用哪种区域设置。例如：

```
initdb --locale=sv_SE
```

这个 Unix 系统上的示例把区域设置为在瑞典（`SE`）使用的瑞典语（`sv`）。其他可能的值包括 `en_US`（美国英语）和 `fr_CA`（加拿大法语）。如果一个区域设置可以使用多于一种字符集，则说明可以采用 *`language_territory.codeset`* 这样的形式。例如，`fr_BE.UTF-8` 表示在比利时（BE）使用、采用 UTF-8 字符集编码的法语（fr）。

系统上有哪些区域设置可用，以及它们以什么名称出现，取决于操作系统供应商提供了什么以及实际安装了什么。在大多数 Unix 系统上，命令 `locale -a` 会给出所有可用区域设置的列表。Windows 使用更冗长的区域设置名称，例如`German_Germany` 或 `Swedish_Sweden.1252`，但原则是相同的。

有时混合多个区域设置的规则会很有用，例如使用英语排序规则但使用西班牙语消息。为此，存在一组只控制本地化规则某些方面的区域设置子类别：

`LC_COLLATE`	字符串排序顺序
`LC_CTYPE`	字符分类（什么算字母？它的大写等价形式是什么？）
`LC_MESSAGES`	消息的语言
`LC_MONETARY`	货币数量使用的格式
`LC_NUMERIC`	数字的格式
`LC_TIME`	日期和时间的格式

这些类别名会转换成 `initdb` 选项名，用于覆盖特定类别的区域设置选择。例如，要将区域设置设为加拿大法语，但对货币格式采用美国规则，可以使用`initdb --locale=fr_CA --lc-monetary=en_US`。

如果希望系统表现得像没有区域设置支持一样，可使用特殊区域设置名 `C`，或等效的`POSIX`。

某些区域设置类别的值必须在创建数据库时固定下来。你可以为不同数据库使用不同设置，但数据库一旦创建，这些类别的值就不能再更改。LC_COLLATE 和 LC_CTYPE 就属于这类类别。它们会影响索引的排序顺序，因此必须保持固定，否则文本列上的索引会损坏。（不过，可以通过排序规则缓解这一限制，参见第 23.2 节。）这些类别的默认值在运行 initdb 时确定，创建新数据库时会使用这些值，除非在CREATE DATABASE 命令中另行指定。

其他区域设置类别可以在任何时候通过设置与其同名的服务器配置参数来更改（详见第 19.11.2 节）。initdb 选择的值实际上只是写入配置文件 postgresql.conf 中，作为服务器启动时的默认值。如果从postgresql.conf 中删除这些赋值，服务器就会从其执行环境继承相应设置。

请注意，服务器的区域设置行为由服务器看到的环境变量决定，而不是由任何客户端的环境决定。因此，务必在启动服务器前配置好正确的区域设置。由此带来的一个结果是，如果客户端和服务器使用不同的区域设置，消息可能会因其来源不同而显示为不同语言。

> **注意**
>
> 这里所说的从执行环境继承区域设置，在大多数操作系统上是指：对于某个给定的区域设置类别，例如排序规则，会按以下顺序检查环境变量，直到找到一个已设置的变量为止：LC_ALL、LC_COLLATE（或相应类别对应的变量）、LANG。如果这些环境变量都没有设置，则区域设置默认为C。
>
> 某些消息本地化库还会查看环境变量 LANGUAGE，它会覆盖所有其他用于设置消息语言的区域设置。如果有疑问，请参考操作系统文档，尤其是关于gettext 的文档。

若要让消息可以翻译为用户偏好的语言，构建时必须启用NLS（configure --enable-nls）。所有其他区域设置支持都会自动内置。

23.1.2. 行为

区域设置会影响下列 SQL 特性：

- 在对文本数据使用 ORDER BY 或标准比较操作符的查询中的排序顺序
- upper、lower 和 initcap 函数
- 模式匹配操作符（LIKE、SIMILAR TO 以及 POSIX 风格正则表达式）；区域设置既会影响大小写不敏感匹配，也会影响字符类正则表达式中的字符分类
- to_char 函数族
- 对 LIKE 子句使用索引的能力

在 PostgreSQL 中使用 C 或 POSIX 之外的区域设置，缺点是会影响性能。它会减慢字符处理速度，并阻止普通索引用于LIKE。因此，只有在确有需要时才应使用区域设置。

为了让 PostgreSQL 在非 C 区域设置下也能对 LIKE 子句使用索引，存在若干自定义操作符类可供使用。它们允许创建执行严格逐字符比较、忽略区域设置比较规则的索引。详见第 11.10 节。另一种方法是创建使用 C 排序规则的索引，如第 23.2 节所述。

23.1.3. 问题

如果区域设置支持没有像上面说明的那样工作，请检查操作系统中的区域设置支持是否已经正确配置。要检查系统中安装了哪些区域设置，可以使用 `locale -a` 命令（如果操作系统提供该命令）。

请确认 PostgreSQL 实际使用的区域设置就是你预期的区域设置。`LC_COLLATE` 和 `LC_CTYPE` 是在创建数据库时确定的，除非重新创建数据库，否则不能更改。其他区域设置，包括 `LC_MESSAGES` 和 `LC_MONETARY`，最初由启动服务器时的环境决定，但可以在运行中动态修改。你可以使用`SHOW` 命令检查当前生效的区域设置。

源码发布包中的 `src/test/locale` 目录包含 PostgreSQL 区域设置支持的测试套件。

那些通过解析错误消息文本来处理服务器端错误的客户端应用，显然会遇到问题，因为服务器消息可能使用不同语言。这类应用的作者应改用错误代码机制。

维护消息翻译目录需要许多志愿者持续投入，他们希望 PostgreSQL 能够良好地支持他们偏好的语言。如果你所用语言的消息目前还不可用，或尚未完全翻译，我们将非常感谢你的协助。如果你愿意帮忙，请参考第 55 章，或向开发者邮件列表写信。

23.2. 排序规则支持

排序规则特性允许为每一列甚至每一次操作指定数据的排序顺序和字符分类行为。这缓解了数据库的 `LC_COLLATE` 和 `LC_CTYPE` 设置在创建之后无法更改这一限制。

23.2.1. 概念

从概念上讲，每个支持排序规则的数据类型的表达式都有一个排序规则。（内置的支持排序规则的数据类型包括`text`、`varchar` 和 `char`。用户定义的基础类型也可以标记为支持排序规则，当然，建立在支持排序规则的数据类型之上的域也支持排序规则。）如果表达式是列引用，则该表达式的排序规则就是该列定义的排序规则。如果表达式是常量，则其排序规则就是该常量数据类型的默认排序规则。更复杂表达式的排序规则则按下文所述，从其输入表达式的排序规则推导出来。

表达式的排序规则可以是“默认”排序规则，这表示为数据库定义的区域设置。表达式的排序规则也可能是不确定的。在这种情况下，排序操作以及其他需要知道排序规则的操作都会失败。

当数据库系统必须执行排序或字符分类时，它会使用输入表达式的排序规则。这例如发生在 `ORDER BY` 子句以及函数或操作符调用（如 `<`）中。应用于 `ORDER BY` 子句的排序规则就是排序键的排序规则。应用于函数或操作符调用的排序规则则按下文所述，从参数中推导出来。除了比较操作符之外，在大小写之间进行转换的函数（例如`lower`、`upper` 和 `initcap`）、模式匹配操作符，以及`to_char` 及相关函数，也都会考虑排序规则。

对于函数或操作符调用，通过检查参数排序规则推导出的排序规则，会在运行时用于执行指定操作。如果该函数或操作符调用的结果属于支持排序规则的数据类型，那么在解析时它也会被用作该函数或操作符表达式的已定义排序规则，以便在外围表达式需要知道其排序规则时使用。

表达式的排序规则派生可以是显式的，也可以是隐式的。这一区别会影响当一个表达式中出现多个不同排序规则时，系统如何把它们组合起来。使用`COLLATE` 子句时，会发生显式排序规则派生；其他所有排序规则派生都是隐式的。当需要组合多个排序规则时，例如在函数调用中，将使用以下规则：

1. 如果任一输入表达式具有显式排序规则派生，那么输入表达式中所有显式派生的排序规则都必须相同，否则会报错。如果存在显式派生的排序规则，那么排序规则组合的结果就是该排序规则。

2. 否则，所有输入表达式都必须具有相同的隐式排序规则派生，或者为默认排序规则。如果存在任何非默认排序规则，那么排序规则组合的结果就是该排序规则；否则，结果就是默认排序规则。

3. 如果输入表达式之间存在相互冲突的非默认隐式排序规则，则该组合会被视为具有不确定排序规则。除非被调用的特定函数确实需要知道它应当使用哪个排序规则，否则这并不是错误。如果它确实需要，就会在运行时抛出错误。

例如，考虑如下表定义：

```
CREATE TABLE test1 (
    a text COLLATE "de_DE",
    b text COLLATE "es_ES",
    ...
);
```

那么在

```
SELECT a < 'foo' FROM test1;
```

中，<比较会按 de_DE 规则进行，因为该表达式组合了一个隐式派生的排序规则和默认排序规则。但在

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

中，比较会按fr_FR 规则进行，因为显式排序规则派生覆盖了隐式派生。进一步，给定

```
SELECT a < b FROM test1;
```

解析器无法确定应当应用哪个排序规则，因为a 列和 b 列具有冲突的隐式排序规则。由于 <操作符确实需要知道要使用哪个排序规则，因此这会导致一个错误。该错误可以通过给任一输入表达式附加显式排序规则说明符来解决，例如：

```
SELECT a < b COLLATE "de_DE" FROM test1;
```

或者等价地：

```
SELECT a COLLATE "de_DE" < b FROM test1;
```

另一方面，结构相似的情况

```
SELECT a || b FROM test1;
```

不会导致错误，因为||操作符不关心排序规则：无论使用什么排序规则，其结果都相同。

如果函数或操作符返回的是支持排序规则的数据类型，那么分配给该函数或操作符组合输入表达式的排序规则，也被认为适用于其结果。因此，在

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

中，排序将按de_DE 规则进行。但这个查询：

```
SELECT * FROM test1 ORDER BY a || b;
```

会报错，因为即使||操作符本身不需要知道排序规则，ORDER BY 子句仍然需要。与之前一样，可以通过显式排序规则说明符来解决冲突：

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

23.2.2. 管理排序规则

排序规则是一个 SQL 模式对象，它把某个 SQL 名称映射到操作系统中已安装库提供的区域设置。排序规则定义具有一个提供程序，用于指定由哪个库提供区域设置数据。标准提供程序之一是libc，它使用操作系统 C 库提供的区域设置。这些也是大多数操作系统工具所使用的区域设置。另一个提供程序是icu，它使用外部 ICU库。只有在构建 PostgreSQL 时配置了 ICU 支持，才能使用 ICU 区域设置。

由 libc 提供的排序规则对象，对应于 LC_COLLATE 和 LC_CTYPE 设置的组合，其格式与系统库调用setlocale() 所接受的格式相同。（顾名思义，排序规则的主要用途是设置控制排序顺序的 LC_COLLATE。但在实践中，很少有必要让 LC_CTYPE 和 LC_COLLATE 不同，因此把二者归为一个概念，比再建立一套为每个表达式设置LC_CTYPE 的机制更方便。）此外，libc 排序规则还与某种字符集编码绑定（见第 23.3 节）。同一个排序规则名称可能会在不同编码中出现。

由 icu 提供的排序规则对象，对应于 ICU 库提供的具名排序器。ICU 不支持将 “collate” 和 “ctype” 分开设置，因此二者总是相同的。此外，ICU 排序规则与编码无关，因此在一个数据库中，某个给定名称的 ICU 排序规则始终只有一个。

23.2.2.1. 标准排序规则

所有平台都提供名为 default、C 和 POSIX 的排序规则。根据操作系统支持情况，还可能提供其他排序规则。default 排序规则选择创建数据库时指定的 LC_COLLATE 和 LC_CTYPE 值。C 和 POSIX 排序规则都采用 “传统 C” 行为，仅将 ASCII 字母 “A” 到 “Z” 视为字母，并严格按字符编码的字节值排序。

此外，编码 UTF8 还可以使用 SQL 标准排序规则名 ucs_basic。它等价于 C，按 Unicode 码点排序。

23.2.2.2. 预定义排序规则

如果操作系统支持在单个程序中使用多个区域设置（newlocale 及相关函数），或者配置了 ICU 支持，那么在初始化数据库集簇时，initdb 会根据当时在操作系统中发现的所有区域设置，用排序规则填充系统目录pg_collation。

要检查当前可用的区域设置，可使用查询SELECT * FROM pg_collation，或在psql 中使用命令\dOS+。

23.2.2.2.1. libc 排序规则

例如，操作系统可能提供一个名为de_DE.utf8 的区域设置。initdb 随后会为 UTF8 编码创建一个名为 de_DE.utf8 的排序规则，其 LC_COLLATE 和 LC_CTYPE 都设置为 de_DE.utf8。它还会再创建一个从名称中去掉 .utf8 标签的排序规则。因此，你也可以使用 de_DE 这个名称来使用该排序规则，这样写起来更方便，且名称与编码的耦合更小。不过请注意，初始排序规则名称集合仍然取决于平台。

由 libc 提供的默认排序规则，直接映射到操作系统中已安装的区域设置，它们可以用 locale -a 命令列出。如果需要某个 libc 排序规则，而其LC_COLLATE 和 LC_CTYPE 取值不同，或者数据库系统初始化后又在操作系统中安装了新的区域设置，那么可以使用CREATE COLLATION 命令创建新的排序规则。新的操作系统区域设置也可以用 pg_import_system_collations() 函数批量导入。

在任何特定数据库中，只有使用该数据库编码的排序规则才有意义。pg_collation 中的其他条目会被忽略。因此，像 de_DE 这样去掉编码后缀的排序规则名称，在某个给定数据库内可以视为唯一，即使它在全局范围内并不唯一。推荐使用这种去掉后缀的排序规则名称，因为如果你决定改用另一种数据库编码，需要改动的地方会更少。不过要注意，default、C 和 POSIX 排序规则不受数据库编码影响，始终都可以使用。

PostgreSQL 即使面对具有相同属性的不同排序规则对象，也会把它们视为不兼容。例如：

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

即使 C 和 POSIX 排序规则的行为完全相同，这仍然会报错。因此，不建议混用去掉后缀和保留后缀的排序规则名称。

23.2.2.2.2. ICU 排序规则

对于 ICU，枚举所有可能的区域设置名称并不合理。ICU 为区域设置使用特定的命名系统，但给区域设置命名的方法远多于实际存在的不同区域设置。initdb 使用 ICU API 提取一组不同的区域设置，以填充初始排序规则集合。由 ICU 提供的排序规则在 SQL 环境中创建时，其名称采用 BCP 47 语言标签格式，并附加“私有使用”扩展 -x-icu，以便与 libc 区域设置区分开来。

下面是一些可能创建出来的排序规则示例：

de-x-icu

德语排序规则，默认变体

de-AT-x-icu

奥地利德语排序规则，默认变体

（还有例如 de-DE-x-icu 或 de-CH-x-icu，但在撰写本文时，它们与de-x-icu 等价。）

und-x-icu（表示“未定义”）

ICU“根”排序规则。可用它获得一种较为合理、与特定语言无关的排序顺序。

某些（较少使用的）编码不受 ICU 支持。当数据库编码属于其中之一时，pg_collation 中的 ICU 排序规则条目会被忽略。尝试使用这样的排序规则会报出类似“collation "de-x-icu" for encoding "WIN874" does not exist”的错误。

23.2.2.3. 创建新的排序规则对象

如果标准和预定义排序规则不足以满足需求，用户可以使用 SQL 命令CREATE COLLATION 创建自己的排序规则对象。

与所有预定义对象一样，标准和预定义排序规则都位于模式 `pg_catalog` 中。用户定义的排序规则应当创建在用户模式中。这也能确保它们被 `pg_dump` 保存。

23.2.2.3.1. libc 排序规则

新的 libc 排序规则可以这样创建：

```
CREATE COLLATION german (provider = libc, locale = 'de_DE');
```

该命令中 `locale` 子句可接受的确切值取决于操作系统。在类 Unix 系统上，命令`locale -a` 会给出一个列表。

由于预定义的 libc 排序规则已经包含了数据库实例初始化时操作系统中定义的所有排序规则，因此通常不需要手工创建新的排序规则。可能需要这样做的原因包括希望采用不同的命名系统（这种情况下另见第 23.2.2.3.3 节），或操作系统升级后提供了新的区域设置定义（这种情况下另见 `pg_import_system_collations()`）。

23.2.2.3.2. ICU 排序规则

除了 `initdb` 预装载的基本语言与国家组合外，ICU 还允许进一步定制排序规则。建议用户定义自己的排序规则对象，利用这些功能使排序行为满足自己的需求。关于 ICU 区域设置名称的信息，参见https://unicode-org.github.io/icu/userguide/locale/和 https://unicode-org.github.io/icu/userguide/collation/api.html。可接受的名称和属性集合取决于具体的 ICU 版本。

以下是一些示例：

```
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale =
 'de-u-co-phonebk');
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale =
 'de@collation=phonebook');
```

德语排序规则，电话簿排序类型

第一个示例使用 BCP 47 定义的 “语言标签” 来选择 ICU 区域设置。第二个示例使用传统的 ICU 专有区域设置语法。今后应优先采用第一种形式，但较旧的 ICU 版本不支持它。

注意，可以在 SQL 环境中为排序规则对象任意命名。本例遵循预定义排序规则所采用的命名风格，该风格也遵循 BCP 47，但用户定义的排序规则并不要求如此。

```
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale =
 'und-u-co-emoji');
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale =
 '@collation=emoji');
```

根排序规则，表情符号排序类型，依据 Unicode 技术标准 #51

注意，传统的 ICU 区域设置命名系统使用空字符串来选择根区域设置。

```
CREATE COLLATION latinlast (provider = icu, locale = 'en-u-kr-grek-
latn');
CREATE COLLATION latinlast (provider = icu, locale = 'en@colReorder=
grek-latn');
```

将希腊字母排在拉丁字母之前。（默认情况下是拉丁字母排在希腊字母之前。）

```
CREATE COLLATION upperfirst (provider = icu, locale = 'en-u-kf-upper')
;
CREATE COLLATION upperfirst (provider = icu, locale = 'en@colCase
First=upper');
```

将大写字母排在小写字母之前。（默认情况下是小写字母在前。）

```
CREATE COLLATION special (provider = icu, locale = 'en-u-kf-upper-kr-
grek-latn');
CREATE COLLATION special (provider = icu, locale = 'en@colCaseFirst=
upper;colReorder=grek-latn');
```

结合了以上两个选项。

```
CREATE COLLATION numeric (provider = icu, locale = 'en-u-kn-true');
CREATE COLLATION numeric (provider = icu, locale = 'en@colNumeric=
yes');
```

数值排序，按数值对数字序列排序，例如：`A-21` < `A-123`（也称自然排序）。

参见 Unicode 技术标准 #35[1] 和 BCP 47[2] 了解详情。可用排序类型列表（`co` 子标签）可在以下位置找到：CLDR 仓库[3]。

注意，虽然此系统允许创建“忽略大小写”或“忽略重音”等排序规则（使用 `ks` 键），但 PostgreSQL 目前不允许这些排序规则真正以不区分大小写或重音的方式工作。按排序规则比较相等、但字节不相等的字符串，仍会按字节值排序。

> **注意**
>
> ICU 的设计允许接受几乎任何字符串作为区域设置名，并按照其文档描述的回退流程，匹配到它能够提供的最接近区域设置。因此，如果排序规则定义使用了当前 ICU 安装实际不支持的功能，不会得到直接反馈。所以建议创建应用层面的测试用例，以检查排序规则定义是否满足需求。

23.2.2.3.3. 复制排序规则

命令 CREATE COLLATION 也可以用来从现有排序规则创建一个新的排序规则。这有助于在应用程序中使用与操作系统无关的排序规则名称、创建兼容性名称，或者以更易读的名称来使用 ICU 提供的排序规则。例如：

```
CREATE COLLATION german FROM "de_DE";
CREATE COLLATION french FROM "fr-x-icu";
```

[1] http://unicode.org/reports/tr35/tr35-collation.html
[2] https://tools.ietf.org/html/bcp47
[3] https://github.com/unicode-org/cldr/blob/master/common/bcp47/collation.xml

23.3. 字符集支持

PostgreSQL 中的字符集支持允许你以多种字符集（也称为编码）存储文本，包括 ISO 8859 系列之类的单字节字符集，以及 EUC（扩展 Unix 编码）、UTF-8 和 Mule 内部编码等多字节字符集。所有受支持的字符集都可以被客户端透明地使用，但其中少数不支持在服务器内部使用（即不能作为服务器端编码）。默认字符集是在使用 `initdb` 初始化 PostgreSQL 数据库集簇时选定的。创建数据库时可以覆盖该设置，因此你可以拥有多个数据库，并让每个数据库使用不同的字符集。

不过，一个重要限制是，每个数据库的字符集必须与该数据库的 `LC_CTYPE`（字符分类）和 `LC_COLLATE`（字符串排序顺序）区域设置兼容。对于 `C` 或 `POSIX` 区域设置，任何字符集都可以使用；但对于其他由 libc 提供的区域设置，只有一种字符集能够正确工作。（不过在 Windows 上，UTF-8 编码可以与任何区域设置配合使用。）如果配置了 ICU 支持，则 ICU 提供的区域设置可用于大多数（但不是全部）服务器端编码。

23.3.1. 支持的字符集

表 23.1 显示了 PostgreSQL 中可用的字符集。

表 23.1. PostgreSQL 字符集

名称	描述	语言	是否服务器端?	ICU?	字节数/字符	别名
`BIG5`	Big Five	繁体中文	否	否	1-2	`WIN950`, `Windows950`
`EUC_CN`	扩展 Unix 编码-CN	简体中文	是	是	1-3	
`EUC_JP`	扩展 Unix 编码-JP	日语	是	是	1-3	
`EUC_JIS_2004`	扩展 Unix 编码-JP，JIS X 0213	日语	是	否	1-3	
`EUC_KR`	扩展 Unix 编码-KR	韩语	是	是	1-3	
`EUC_TW`	扩展 Unix 编码-TW	繁体中文、台湾语	是	是	1-3	
`GB18030`	国家标准	中文	否	否	1-4	
`GBK`	扩展国家标准	简体中文	否	否	1-2	`WIN936`, `Windows936`
`ISO_8859_5`	ISO 8859-5, ECMA 113	拉丁字母/西里尔字母	是	是	1	
`ISO_8859_6`	ISO 8859-6, ECMA 114	拉丁字母/阿拉伯字母	是	是	1	
`ISO_8859_7`	ISO 8859-7, ECMA 118	拉丁字母/希腊字母	是	是	1	
`ISO_8859_8`	ISO 8859-8, ECMA 121	拉丁字母/希伯来字母	是	是	1	
`JOHAB`	JOHAB	韩语（谚文）	否	否	1-3	
`KOI8R`	KOI8-R	西里尔字母（俄语）	是	是	1	`KOI8`

名称	描述	语言	是否服务器端?	ICU?	字节数/字符	别名
KOI8U	KOI8-U	西里尔字母（乌克兰语）	是	是	1	
LATIN1	ISO 8859-1, ECMA 94	西欧	是	是	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	中欧	是	是	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	南欧	是	是	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	北欧	是	是	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	土耳其语	是	是	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	北欧	是	是	1	ISO885910
LATIN7	ISO 8859-13	波罗的海	是	是	1	ISO885913
LATIN8	ISO 8859-14	凯尔特语	是	是	1	ISO885914
LATIN9	ISO 8859-15	带欧元符号和重音字符的 LATIN1	是	是	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	罗马尼亚语	是	否	1	ISO885916
MULE_INTERNAL	Mule 内部编码	多语种 Emacs	是	否	1-4	
SJIS	Shift JIS	日语	否	否	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	日语	否	否	1-2	
SQL_ASCII	未指定（见正文）	任意	是	否	1	
UHC	统一谚文编码	韩语	否	否	1-2	WIN949, Windows949
UTF8	Unicode，8 位	所有	是	是	1-4	Unicode
WIN866	Windows CP866	西里尔字母	是	是	1	ALT
WIN874	Windows CP874	泰语	是	否	1	
WIN1250	Windows CP1250	中欧	是	是	1	
WIN1251	Windows CP1251	西里尔字母	是	是	1	WIN
WIN1252	Windows CP1252	西欧	是	是	1	

名称	描述	语言	是否服务器端?	ICU?	字节数/字符	别名
WIN1253	Windows CP1253	希腊语	是	是	1	
WIN1254	Windows CP1254	土耳其语	是	是	1	
WIN1255	Windows CP1255	希伯来语	是	是	1	
WIN1256	Windows CP1256	阿拉伯语	是	是	1	
WIN1257	Windows CP1257	波罗的海	是	是	1	
WIN1258	Windows CP1258	越南语	是	是	1	ABC, TCVN, TCVN5712, VSCII

并非所有客户端 API 都支持上表中的全部字符集。例如，PostgreSQL JDBC 驱动就不支持 MULE_INTERNAL、LATIN6、LATIN8 和 LATIN10。

SQL_ASCII 设置的行为与其他设置有很大不同。当服务器字符集为 SQL_ASCII 时，服务器按 ASCII 标准解释字节值 0-127，而将字节值 128-255 视为不作解释的字符。设置为 SQL_ASCII 时，不会进行任何编码转换。因此，这一设置与其说是声明使用某种特定编码，不如说是承认自己不知道编码是什么。在大多数情况下，如果需要处理任何非 ASCII 数据，使用 SQL_ASCII 都是不明智的，因为 PostgreSQL 将无法帮助转换或验证非 ASCII 字符。

23.3.2. 设置字符集

initdb 为 PostgreSQL 集簇定义默认字符集（编码）。例如：

```
initdb -E EUC_JP
```

这会把默认字符集设为 EUC_JP（日语的扩展 Unix 编码）。如果你喜欢更长的选项形式，也可以用--encoding 代替 -E。如果既没有给出 -E 也没有给出 --encoding，initdb 会根据指定的或默认的区域设置，尝试确定应使用的合适编码。

你可以在创建数据库时指定非默认编码，只要该编码与所选区域设置兼容：

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-ctype=ko_KR.euckr korean
```

这将创建一个名为 korean 的数据库，它使用 EUC_KR 字符集和 ko_KR 区域设置。另一种实现方式是使用以下 SQL 命令：

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr' LC_CTYPE='ko_KR.euckr'
 TEMPLATE=template0;
```

请注意，上述命令指定复制template0 数据库。复制其他任何数据库时，新数据库的编码和区域设置都必须与源数据库一致，否则可能导致数据损坏。详见第 22.3 节。

数据库编码存储在系统目录 pg_database 中。可以通过 psql 的 -l 选项或\l 命令查看它。

```
$ psql -l
                                  List of databases
   Name    |  Owner   | Encoding  |  Collation  |    Ctype    |          Access Privileges
-----------+----------+-----------+-------------+-------------+-------------------------------
-----
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/hlinnaka,hlinnaka=CTc/
hlinnaka}
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/hlinnaka,hlinnaka=CTc/
hlinnaka}
(7 rows)
```

> **重要**
>
> 在大多数现代操作系统上，PostgreSQL 能够判断 LC_CTYPE 设置所隐含的字符集，并强制只允许使用与之匹配的数据库编码。在较老的系统上，你需要自行确保所用编码正是所选区域设置期望的编码。这里的错误很可能会导致区域设置相关操作（例如排序）出现奇怪的行为。
>
> 即使 LC_CTYPE 不是 C 或 POSIX，PostgreSQL 仍允许超级用户创建使用 SQL_ASCII 编码的数据库。正如前文所述，SQL_ASCII 并不强制数据库中存储的数据具有任何特定编码，因此这种选择存在区域设置相关误行为的风险。这种设置组合已经被弃用，未来某一天可能会被完全禁止。

23.3.3. 服务器和客户端之间的自动字符集转换

PostgreSQL 支持在服务器和客户端之间，对某些字符集组合进行自动字符集转换。转换信息保存在系统目录 pg_conversion 中。PostgreSQL 提供了一些预定义转换，如表 23.2 所示。可以使用 SQL 命令 CREATE CONVERSION 创建新的转换。

表 23.2. 客户端/服务器字符集转换

服务器字符集	可用的客户端字符集
BIG5	不支持作为服务器编码
EUC_CN	EUC_CN, MULE_INTERNAL, UTF8
EUC_JP	EUC_JP, MULE_INTERNAL, SJIS, UTF8
EUC_JIS_2004	EUC_JIS_2004, SHIFT_JIS_2004, UTF8
EUC_KR	EUC_KR, MULE_INTERNAL, UTF8
EUC_TW	EUC_TW, BIG5, MULE_INTERNAL, UTF8
GB18030	不支持作为服务器编码
GBK	不支持作为服务器编码
ISO_8859_5	ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	ISO_8859_6, UTF8
ISO_8859_7	ISO_8859_7, UTF8
ISO_8859_8	ISO_8859_8, UTF8

服务器字符集	可用的客户端字符集
JOHAB	不支持作为服务器编码
KOI8R	KOI8R, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
KOI8U	KOI8U, UTF8
LATIN1	LATIN1, MULE_INTERNAL, UTF8
LATIN2	LATIN2, MULE_INTERNAL, UTF8, WIN1250
LATIN3	LATIN3, MULE_INTERNAL, UTF8
LATIN4	LATIN4, MULE_INTERNAL, UTF8
LATIN5	LATIN5, UTF8
LATIN6	LATIN6, UTF8
LATIN7	LATIN7, UTF8
LATIN8	LATIN8, UTF8
LATIN9	LATIN9, UTF8
LATIN10	LATIN10, UTF8
MULE_INTERNAL	MULE_INTERNAL, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8R, LATIN1 至 LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	不支持作为服务器编码
SHIFT_JIS_2004	不支持作为服务器编码
SQL_ASCII	任何（不会执行转换）
UHC	不支持作为服务器编码
UTF8	所有受支持的编码
WIN866	WIN866, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN1251
WIN874	WIN874, UTF8
WIN1250	WIN1250, LATIN2, MULE_INTERNAL, UTF8
WIN1251	WIN1251, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866
WIN1252	WIN1252, UTF8
WIN1253	WIN1253, UTF8
WIN1254	WIN1254, UTF8
WIN1255	WIN1255, UTF8
WIN1256	WIN1256, UTF8
WIN1257	WIN1257, UTF8
WIN1258	WIN1258, UTF8

若要启用自动字符集转换，必须告诉PostgreSQL 你希望客户端使用哪种字符集（编码）。有几种方法可以做到这一点：

- 在 psql 中使用\encoding 命令。\encoding 允许动态更改客户端编码。例如，要把编码改成SJIS，可以输入：

```
\encoding SJIS
```

- libpq（第 34.10 节）提供了控制客户端编码的函数。
- 使用 `SET client_encoding TO`。可以使用以下 SQL 命令设置客户端编码：

  ```
  SET CLIENT_ENCODING TO 'value';
  ```

  也可以为此使用标准 SQL 语法 `SET NAMES`：

  ```
  SET NAMES 'value';
  ```

  要查询当前客户端编码：

  ```
  SHOW client_encoding;
  ```

  要返回到默认编码：

  ```
  RESET client_encoding;
  ```

- 使用 `PGCLIENTENCODING`。如果在客户端环境中定义了 `PGCLIENTENCODING` 环境变量，那么连接服务器时会自动选择该客户端编码。（之后仍可以用上面提到的任何其他方法覆盖它。）
- 使用配置变量 client_encoding。如果设置了 `client_encoding` 变量，那么连接服务器时会自动选择该客户端编码。（之后仍可以用上面提到的任何其他方法覆盖它。）

如果某个特定字符无法完成转换，就会报错。例如，假设服务器使用 `EUC_JP` 而客户端使用 `LATIN1`，此时若返回了一些在`LATIN1` 中没有表示形式的日语字符，就会出现错误。

如果客户端字符集定义为 `SQL_ASCII`，则无论服务器字符集是什么，都会禁用编码转换。与服务器一样，除非处理的全部是 ASCII 数据，否则使用 `SQL_ASCII` 是不明智的。

23.3.4. 进一步阅读

以下资料是了解各种编码系统的良好起点。

CJKV Information Processing: Chinese, Japanese, Korean & Vietnamese Computing

其中对 `EUC_JP`、`EUC_CN`、`EUC_KR` 和 `EUC_TW` 做了详细说明。

http://www.unicode.org/

Unicode Consortium 网站。

RFC 3629

这里定义了 UTF-8（8 位 UCS/Unicode 转换格式）。

第 24 章 日常数据库维护任务

和任何数据库软件一样，PostgreSQL 为了获得最佳性能，需要定期执行某些任务。这里讨论的任务是必需的，但它们本质上是重复性的，因此可以很容易地用标准工具实现自动化，例如 cron 脚本或 Windows 的 Task Scheduler。建立合适的脚本并检查其是否成功执行，是数据库管理员的职责。

一个显而易见的维护任务，是按固定计划创建数据的备份副本。没有最近的备份，在灾难（磁盘故障、火灾、误删关键表等）发生后就没有恢复的可能。PostgreSQL 提供的备份和恢复机制在第 25 章中有详细讨论。

另一大类维护任务是定期对数据库进行“清理”。这一活动在第 24.1 节中讨论。与之密切相关的是更新查询规划器将会使用的统计信息，这在第 24.1.3 节中讨论。

另一项可能需要定期关注的任务是日志文件管理。这在第 24.3 节中讨论。

check_postgres[1] 可用于监控数据库健康状况并报告异常情况。check_postgres 能与 Nagios 和 MRTG 集成，但也可以单独运行。

和某些其他数据库管理系统相比，PostgreSQL 的维护工作量较小。不过，适当地关注这些任务，将大大有助于确保你愉快而高效地使用该系统。

24.1. 日常清理

PostgreSQL 数据库需要一种称为清理的周期性维护。对于许多安装，让第 24.1.6 节所述的自动清理守护进程执行清理就足够了。为了在你的场景中获得最佳效果，你可能需要调整其中描述的自动清理参数。有些数据库管理员希望用手工管理的 `VACUUM` 命令来补充甚至取代该守护进程的工作，这类命令通常由 cron 或 Task Scheduler 脚本按计划执行。要正确设置手工管理的清理，理解下面几小节讨论的问题至关重要。依赖自动清理的管理员也不妨略读这一材料，以帮助理解和调整自动清理。

24.1.1. 清理基础

PostgreSQL 的 VACUUM 命令必须定期处理每个表，原因如下：

1. 回收或再利用被更新或删除的行所占用的磁盘空间。
2. 更新 PostgreSQL 查询规划器使用的数据统计信息。
3. 更新可见性映射，以加快仅索引扫描。
4. 防止由于事务 ID 回卷或多事务 ID 回卷而丢失非常旧的数据。

针对上述不同目的，需要以不同的频率和范围执行 `VACUUM` 操作，下面几节会详细说明。

`VACUUM` 有两种变体：标准的 `VACUUM` 和 `VACUUM FULL`。`VACUUM FULL` 可以回收更多磁盘空间，但执行速度要慢得多。此外，标准形式的 `VACUUM` 可以与生产数据库操作并行运行。（`SELECT`、`INSERT`、`UPDATE` 以及 `DELETE` 等命令会继续正常工作，不过在表被清理时，你将不能使用 `ALTER TABLE` 等命令修改该表的定义。）`VACUUM FULL` 需要对正在处理的表持有 `ACCESS EXCLUSIVE` 锁，因此不能与对该表的其他使用并行进行。一般来说，管理员应尽量使用标准 `VACUUM` 并避免 `VACUUM FULL`。

[1] https://bucardo.org/check_postgres/

VACUUM 会产生大量 I/O 流量，这可能导致其他活动会话性能变差。可以调整一些配置参数来降低后台清理对性能的影响，参见第 19.4.4 节。

24.1.2. 回收磁盘空间

在 PostgreSQL 中，对某一行执行 UPDATE 或 DELETE 时，不会立即移除该行的旧版本。这种方法对于获得多版本并发控制（MVCC，见第 13 章）的好处是必需的：当行版本仍可能对其他事务可见时，就不能删除它。但最终，过时或已删除的行版本将不再是任何事务关心的对象。必须回收它占用的空间，以供新行重用，从而避免磁盘空间需求无限增长。这是通过运行 VACUUM 完成的。

标准形式的 VACUUM 会移除表和索引中的死行版本，并将空间标记为可供将来重用。不过，它不会把空间归还给操作系统，除非出现一种特殊情况：表尾的一个或多个页面完全空闲，并且能够轻松获得一个排他表锁。相比之下，VACUUM FULL 会主动压实表，它会写出一个不含死空间的全新表文件版本。这会将表的大小降到最小，但可能耗时很长。在操作完成之前，它还需要额外的磁盘空间来存放表的新副本。

例行清理的通常目标，是足够频繁地执行标准 VACUUM，从而避免需要 VACUUM FULL。自动清理守护进程就是按照这种方式工作的，事实上它永远不会发出 VACUUM FULL。这种方法的思路不是让表始终保持在最小尺寸，而是让磁盘空间的使用维持在稳态：每个表占用的空间相当于其最小尺寸，再加上两次清理之间又被用掉的空间。虽然 VACUUM FULL 可以把表重新缩小到最小尺寸，并把磁盘空间归还给操作系统，但如果该表之后还会再次增长，这样做意义并不大。因此，对于维护被频繁更新的表，与其不常执行 VACUUM FULL，不如适度频繁地执行标准 VACUUM。

有些管理员喜欢自己安排清理，例如在夜间负载较低时完成全部工作。按固定时间表执行清理的难点在于，如果某个表的更新活动出现意外高峰，它可能膨胀到确实需要 VACUUM FULL 才能回收空间的程度。使用自动清理守护进程可以缓解这个问题，因为守护进程会根据更新活动动态安排清理。除非工作负载极其可预测，否则完全禁用守护进程是不明智的。一种可能的折中办法是设置守护进程参数，使其只对异常繁重的更新活动作出反应，从而防止情况失控，而在负载正常时，则期望定期调度的 VACUUM 完成大部分工作。

对于不使用自动清理的人，一个典型的做法是在低使用时段每天安排一次面向整个数据库的 VACUUM，并在必要时更频繁地清理那些更新特别频繁的表。（某些更新率极高的安装，甚至会每隔几分钟就清理一次最繁忙的表。）如果你在一个集簇中有多个数据库，不要忘记对每一个数据库都执行 VACUUM；程序 vacuumdb 可能会有帮助。

> **提示**
>
> 当一个表由于大规模更新或删除活动而包含大量死行版本时，普通的 VACUUM 可能并不能令人满意。如果你有这样一个表，并且需要回收它所占用的多余磁盘空间，就需要使用 VACUUM FULL，或者改用 CLUSTER，又或者使用ALTER TABLE 的某一种重写表变体。这些命令会重写整个表的新副本，并为其构建新的索引。所有这些选项都需要 ACCESS EXCLUSIVE 锁。还要注意，它们会临时额外占用大约等于该表大小的磁盘空间，因为在新表和新索引完成之前，旧的表和索引副本都不能被释放。

> **提示**
>
> 如果你有一个表，其全部内容会被定期删除，考虑使用 TRUNCATE，而不是先 DELETE 再 VACUUM。TRUNCATE 会立即移除表的全部内容，而不需要后续再执行 VACUUM 或 VACUUM FULL 来回收此时未使用的磁盘空间。缺点是它会破坏严格的 MVCC 语义。

24.1.3. 更新规划器统计信息

PostgreSQL 查询规划器依赖于有关表内容的统计信息，以便为查询生成良好的计划。这些统计信息由ANALYZE 命令收集，它既可以单独调用，也可以作为 VACUUM 的一个可选步骤执行。拥有足够准确的统计信息很重要，否则糟糕的计划选择可能会降低数据库性能。

如果启用了自动清理守护进程，它会在表内容发生足够大变化时自动发出 ANALYZE 命令。不过，管理员也可能更愿意依靠手工调度的 ANALYZE 操作，尤其是在已知某个表上的更新活动不会影响“重要”列统计信息的情况下。守护进程严格依据插入或更新的行数来安排 ANALYZE；它并不知道这些变化是否会导致有意义的统计变化。

分区和继承子表中发生变化的元组，不会触发对父表的分析。如果父表为空或很少变化，它可能永远不会被自动清理处理，因此也就不会收集整个继承树的统计信息。为了保持统计信息最新，必须手工在父表上运行 ANALYZE。

就像为了回收空间而进行的清理一样，频繁更新统计信息对于更新频繁的表比对于很少更新的表更有用。但即便是更新频繁的表，如果数据的统计分布变化不大，也未必需要更新统计信息。一个简单的经验法则是考虑表中各列的最小值和最大值变化了多少。例如，一个包含行更新时间的 timestamp 列，会随着行的插入和更新而持续增大其最大值；这样的列可能比例如存放网站页面 URL 的列更需要频繁更新统计信息。URL 列收到更改的频率可能一样高，但其值的统计分布变化大概相对缓慢。

可以对特定表，甚至仅对表中的特定列运行 ANALYZE，因此如果你的应用需要，确实可以比其他统计更频繁地更新某些统计信息。然而在实践中，通常最好直接分析整个数据库，因为这是一项很快的操作。ANALYZE 使用对表行的统计随机抽样，而不是读取每一行。

> **提示**
>
> 虽然按列微调 ANALYZE 的频率未必很有成效，但按列调整 ANALYZE 收集的统计信息详细程度可能是值得的。在 WHERE 子句中被频繁使用且数据分布高度不规则的列，可能需要比其他列更细粒度的数据直方图。参见 ALTER TABLE SET STATISTICS，或者使用配置参数default_statistics_target 更改数据库范围的默认值。
>
> 此外，默认情况下，关于函数选择率的信息很有限。不过，如果创建了使用函数调用的表达式索引，系统就会收集关于该函数的有用统计信息，这可以显著改善使用该表达式索引的查询计划。

> **提示**
>
> 自动清理守护进程不会为外部表发出 ANALYZE 命令，因为它无法判断多高的频率才有用。如果你的查询需要依赖外部表的统计信息来正确规划，一个好办法是按照合适的时间表，在这些表上手工运行 ANALYZE 命令。

> **提示**
>
> 自动清理守护进程不会为分区表发出 ANALYZE 命令。只有在父表本身发生变化时，继承父表才会被分析，子表上的变化不会触发父表的自动分析。如果你的查询需要依赖父表统计信息来正确规划，就必须周期性地对这些表手工运行 ANALYZE，以保持统计信息最新。

24.1.4. 更新可见性映射

清理会为每个表维护一个可见性映射，用来跟踪哪些页面只包含已知对所有活动事务都可见的元组（并且在该页面再次被修改之前，对所有未来事务也都可见）。这样做有两个目的。第一，下一次清理时可以跳过这样的页面，因为其中没有需要清理的内容。

第二，它使 PostgreSQL 能够在不访问底层表的情况下，仅使用索引回答某些查询。由于 PostgreSQL 的索引不包含元组可见性信息，普通索引扫描会为每个匹配的索引条目取回堆元组，以检查当前事务是否应该看到它。相反，仅索引扫描会先检查可见性映射。如果已知该页面上的所有元组都可见，就可以跳过对堆的访问。这在大型数据集中尤其有用，因为可见性映射能够避免磁盘访问。可见性映射比堆小得多，因此即便堆非常大，也很容易缓存。

24.1.5. 防止事务 ID 回卷失败

PostgreSQL 的 MVCC 事务语义依赖于能够比较事务 ID（XID）数字：如果某个行版本的插入 XID 大于当前事务的 XID，那么这个行版本就处于“未来”，当前事务不应看到它。然而，由于事务 ID 的大小有限（32 位），长期运行的集簇（超过 40 亿个事务）会遭遇事务 ID 回卷：XID 计数器回到零，于是原本属于过去的事务会突然看起来像在未来，这意味着它们的结果会变得不可见。简而言之，就是灾难性的数据丢失。（实际上数据仍然在那里，但如果你无法访问，这也没什么安慰。）为了避免这种情况，必须至少每 20 亿个事务对每个数据库中的每个表执行一次清理。

周期性清理之所以能解决这个问题，是因为 VACUUM 会将行标记为冻结，表示这些行由一个很久以前就已提交的事务插入，因此该插入事务的效果肯定对所有当前和未来事务都可见。普通 XID 使用模 2^{32} 算术进行比较。这意味着，对任意普通 XID，都有 20 亿个 XID 比它“更老”，另有 20 亿个比它“更新”；换句话说，普通 XID 空间是一个没有端点的环。因此，一旦某个行版本带着某个普通 XID 创建出来，那么无论当前谈论的是哪个普通 XID，在接下来的 20 亿个事务中，这个行版本都会看起来处于“过去”。如果该行版本在超过 20 亿个事务之后仍然存在，它就会突然看起来处于未来。为防止这种情况，PostgreSQL 保留了一个特殊 XID，FrozenTransactionId，它不遵循普通 XID 的比较规则，并且始终被视为比每一个普通 XID 都更老。被冻结的行版本会被当作其插入 XID 是 FrozenTransactionId，因此无论是否发生回卷问题，它们对于所有普通事务都会显得处于“过去”，从而这类行版本在被删除之前一直有效，不管要持续多久。

> **注意**
>
> 在 9.4 之前的 PostgreSQL 版本中，冻结是通过实际把一行的插入 XID 替换为 FrozenTransactionId 实现的，这一点可以在该行的 xmin 系统列中看到。较新的版本只是设置一个标志位，同时保留行原始的 xmin 以供可能的取证用途。不过，在从 9.4 之前版本通过 pg_upgrade 升级得到的数据库中，仍然可能发现 xmin 等于 FrozenTransactionId (2) 的行。

> 此外，系统目录中可能包含 xmin 等于 BootstrapTransactionId (1) 的行，这表示它们是在 initdb 的第一阶段插入的。和 FrozenTransactionId 一样，这个特殊 XID 也被视为比所有普通 XID 都更老。

vacuum_freeze_min_age 控制某个 XID 值需要达到多老，其对应的行才会被冻结。如果那些本来会被冻结的行很快又会被修改，增大这个设置可以避免不必要的工作；而减小这个设置，则会增加在表必须再次被清理之前可以流逝的事务数量。

VACUUM 使用可见性映射来确定必须扫描表的哪些页面。通常，它会跳过没有死行版本的页面，即使其中可能仍有 XID 值很旧的行版本。因此，普通的 VACUUM 不一定会冻结表中的每一个旧行版本。VACUUM 会定期执行激进清理，只跳过既没有死行，也没有任何未冻结 XID 或 MXID 值的页面。vacuum_freeze_table_age 控制 VACUUM 何时执行这种操作：如果距离上一次此类扫描经过的事务数，大于 vacuum_freeze_table_age 减去 vacuum_freeze_min_age，就会扫描全部可见但未全部冻结的页面。将 vacuum_freeze_table_age 设为 0，会强制 VACUUM 对所有扫描都采用这种更激进的策略。

一个表可以不清理的最长时间，是 20 亿个事务减去上次激进清理时的 vacuum_freeze_min_age 值。超过这个期限仍不清理，就可能丢失数据。为确保不会发生这种情况，对于可能包含未冻结行、且其 XID 年龄大于配置参数autovacuum_freeze_max_age 所指定年龄的任何表，都会调用自动清理。（即使禁用了自动清理，也会如此。）

这意味着，如果一个表本来不会因为其他原因被清理，大约每经过 autovacuum_freeze_max_age 减去 vacuum_freeze_min_age 个事务，就会在该表上触发一次自动清理。对于那些为了回收空间而经常清理的表，这一点并不重要。然而，对于静态表（包括只接收插入、但没有更新或删除的表），并不需要为了回收空间而清理，因此设法尽可能拉大这些非常大的静态表上强制自动清理之间的间隔，可能会很有用。显然，这可以通过增大 autovacuum_freeze_max_age 或减小 vacuum_freeze_min_age 来做到。

vacuum_freeze_table_age 的实际最大值是 0.95 * autovacuum_freeze_max_age；高于该值的设置会被截断到该最大值。设置一个高于 autovacuum_freeze_max_age 的值没有意义，因为无论如何，防回卷自动清理都会在那时被触发，而 0.95 的乘数是为了在那之前留出一些余地以手工运行一次 VACUUM。经验上，vacuum_freeze_table_age 应设置为略低于 autovacuum_freeze_max_age 的值，并留出足够空档，使一次常规调度的 VACUUM 或由正常删除、更新活动触发的自动清理能在这个窗口内运行。若设置得过于接近，即使该表最近已经为了回收空间而清理过，也可能仍会触发防回卷自动清理；而较低的值则会导致更频繁的激进清理。

这意味着，如果一个表本来不会因为其他原因被清理，大约每经过 autovacuum_freeze_max_age 减去 vacuum_freeze_min_age 个事务，就会在该表上触发一次自动清理。对于那些为了回收空间而经常清理的表，这一点并不重要。然而，对于静态表（包括只接收插入、但没有更新或删除的表），并不需要为了回收空间而清理，因此设法尽可能拉大这些非常大的静态表上强制自动清理之间的间隔，可能会很有用。显然，这可以通过增大 autovacuum_freeze_max_age 或减小 vacuum_freeze_min_age 来做到。

减小 vacuum_freeze_min_age 的一个缺点是，它可能导致 VACUUM 做无用功：如果某个行版本随后不久就被修改（因而获得新的 XID），冻结它就是浪费时间。因此，该设置应足够大，使得行在不太可能再次发生变化之前不会被冻结。

为了跟踪数据库中最老未冻结 XID 的年龄，VACUUM 会在系统表 pg_class 和 pg_database 中存储 XID 统计信息。具体来说，某个表在 pg_class 中对应行的 relfrozenxid 列，记录了上一次对该表执行的激进 VACUUM 所使用的冻结截止 XID。所有由 XID 早于此截止值的事务插入的行都保证已被冻结。类似地，某个数据库在 pg_database 中对应行的 datfrozenxid 列，是出现在该数据库中的未冻结 XID 的下界，它就是该数据库内各表 relfrozenxid 值的最小值。查看这些信息的一种方便方式是执行如下查询：

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

age 列度量的是从该截止 XID 到当前事务 XID 之间的事务数。

VACUUM 通常只扫描自上次清理以来被修改过的页面，但只有当表中每一个可能包含未冻结 XID 的页面都被扫描时，relfrozenxid 才会被推进。当 relfrozenxid 的年龄超过 vacuum_freeze_table_age 个事务、使用了 VACUUM 的 FREEZE 选项，或者所有尚未全部冻结的页面碰巧都需要清理以移除死行版本时，就会发生这种情况。当 VACUUM 扫描了表中每个尚未全部冻结的页面时，它应把 age(relfrozenxid) 设为略高于所用 vacuum_freeze_min_age 设置的值（更高的部分等于自 VACUUM 开始以来已启动的事务数）。如果直到达到 autovacuum_freeze_max_age 之前，该表都没有执行一次能推进 relfrozenxid 的 VACUUM，那么很快就会被强制执行一次自动清理。

如果由于某种原因，自动清理未能清除表中的旧 XID，当数据库中最老的 XID 距离回卷点只剩 1100 万个事务时，系统就会开始发出类似以下的警告：

```
WARNING:  database "mydb" must be vacuumed within 10985967 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in that database.
```

（按照提示，手动执行 VACUUM 应能解决问题；但请注意，VACUUM 必须由超级用户执行，否则无法处理系统目录，也就无法推进数据库的 datfrozenxid。）如果忽略这些警告，一旦距离回卷只剩不到 100 万个事务，系统就会关闭并拒绝启动任何新事务：

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database "mydb"
HINT:  Stop the postmaster and vacuum that database in single-user mode.
```

预留这 100 万个事务的安全余量，是为了让管理员可以手动执行必需的 VACUUM 命令，在不丢失数据的情况下恢复。不过，系统一旦进入安全关闭模式，就不会再执行命令，因此唯一的办法是停止服务器，再以单用户模式启动服务器，执行 VACUUM。单用户模式下不会强制进入关闭模式。有关使用单用户模式的详细信息，参见postgres 参考页面。

24.1.5.1. 多事务与回卷

多事务 ID 用于支持多个事务对行加锁。由于元组头中存放锁信息的空间有限，当多个事务同时锁定一行时，这些信息会被编码为一个“多事务 ID”，简称 multixact ID。某个特定多事务 ID 包含哪些事务 ID 的信息，单独存放在 pg_multixact 子目录中，元组头的 xmax 字段中只保存多事务 ID。与事务 ID 一样，多事务 ID 由 32 位计数器和相应的存储实现，都需要谨慎处理老化管理、存储清理和回卷。另有单独的存储区保存每个多事务的成员列表，它同样使用 32 位计数器，也需要管理。

每当 VACUUM 扫描表的任何部分时，它都会把遇到的、年龄超过 vacuum_multixact_freeze_min_age 的任何多事务 ID 替换为另一个值，该值可能是零值、单个事务 ID，或者更新的多事务 ID。对于每个表，pg_class.relminmxid 保存该表任何元组中仍可能出现的最老多事务 ID。如果这个值的年龄超过 vacuum_multixact_freeze_table_age，就会强制执行一次激进清理。正如上一节所述，激进清理意味着只有那些已知为全部冻结的页面才会被跳过。可以对 pg_class.relminmxid 使用 mxid_age() 来查看其年龄。

无论出于何种原因而发生，激进的 VACUUM 扫描都能推进该表的值。最终，随着所有数据库中的所有表都被扫描，并推进其最老的多事务值，较老多事务的磁盘存储就可以被移除。

作为一项安全措施，对于多事务年龄大于autovacuum_multixact_freeze_max_age 的任何表，都会执行激进清理扫描。如果已使用的成员存储空间超过可寻址存储空间的 50%，还会从多事务年龄最大的表开始，逐步对所有表执行激进清理扫描。即使名义上禁用了自动清理，这两类激进扫描也都会发生。

24.1.6. 自动清理守护进程

PostgreSQL 具有一个可选但强烈推荐的特性，称为自动清理，其目的是自动执行 VACUUM 和 ANALYZE 命令。启用后，自动清理会检查那些已经积累了大量插入、更新或删除元组的表。这些检查依赖于统计收集功能；因此，除非track_counts 被设置为 true，否则无法使用自动清理。在默认配置下，自动清理是启用的，并且相关配置参数也设置得比较合适。

所谓“自动清理守护进程”实际上由多个进程组成。其中有一个持久运行的守护进程，称为自动清理启动器，负责为所有数据库启动自动清理工作进程。启动器会把工作分散在时间上，力图在每个数据库中每隔autovacuum_naptime 秒启动一个工作进程。（因此，如果安装中有 *N* 个数据库，就会每隔 autovacuum_naptime/*N* 秒启动一个新的工作进程。）同时最多允许运行 autovacuum_max_workers 个工作进程。如果需要处理的数据库数量多于 autovacuum_max_workers，那么一旦第一个工作进程结束，就会处理下一个数据库。每个工作进程会检查其数据库中的每个表，并根据需要执行 VACUUM 和/或 ANALYZE。可以设置 log_autovacuum_min_duration 来监控自动清理工作进程的活动。

如果多个大型表在很短时间内都变得需要清理，那么所有自动清理工作进程都可能会长期忙于清理这些表。这会导致其他表和数据库在某个工作进程空闲下来之前都得不到清理。单个数据库中的工作进程数量没有上限，但工作进程会尽量避免重复其他工作进程已经完成的工作。注意，正在运行的工作进程数量不计入max_connections 或 superuser_reserved_connections 的限制。

凡是 relfrozenxid 值的年龄超过 autovacuum_freeze_max_age 个事务的表，始终都会被清理（这也适用于通过存储参数修改了最大冻结年龄的表，见下文）。否则，如果自上次 VACUUM 以来失效的元组数超过“清理阈值”，就会清理该表。清理阈值定义为：

```
vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples
```

其中，清理基础阈值为 autovacuum_vacuum_threshold，清理比例因子为 autovacuum_vacuum_scale_factor，元组数为 pg_class.reltuples。失效元组数来自统计收集器；这是一个近似计数，由每次 UPDATE 和 DELETE 操作更新。（之所以只是近似值，是因为高负载下可能丢失部分信息。）如果表的 relfrozenxid 值的年龄超过 vacuum_freeze_table_age 个事务，就会执行激进清理，冻结旧元组并推进 relfrozenxid；否则，只扫描自上次清理以来被修改过的页面。

对于分析操作，也使用了一个类似的条件：其阈值定义如下：

```
analyze threshold = analyze base threshold + analyze scale factor * number of tuples
```

该阈值会与自上次 ANALYZE 以来插入、更新或删除的元组总数进行比较。

分区表不直接存储元组，因此不会被自动清理处理。（自动清理会像处理其他表一样处理表分区。）不幸的是，这意味着自动清理不会对分区表执行 ANALYZE，而这可能导致引用分区表统计信息的查询生成次优计划。你可以在分区表首次填充时手工运行 ANALYZE 来绕过这个问题，并在其各分区中的数据分布发生显著变化时再次运行。

自动清理无法访问临时表。因此，应通过会话 SQL 命令执行适当的清理和分析操作。

默认阈值和尺度因子取自 postgresql.conf，但也可以按表覆盖它们（以及许多其他自动清理控制参数）；详情见存储参数。如果某个设置已经通过表的存储参数修改，那么处理该表时就使用那个值；否则使用全局设置。关于全局设置的更多细节，见第 19.10 节。

当多个工作进程同时运行时，自动清理的代价延迟参数（见第 19.4.4 节）会在所有运行中的工作进程之间“平衡”，这样无论实际运行了多少个工作进程，系统承受的总 I/O 影响都相同。不过，任何正在处理那些为每表存储参数 autovacuum_vacuum_cost_delay 或 autovacuum_vacuum_cost_limit 显式设置过值的表的工作进程，都不会被纳入这个均衡算法。

自动清理工作进程通常不会阻塞其他命令。如果某个进程试图获取与自动清理持有的 SHARE UPDATE EXCLUSIVE 锁冲突的锁，那么该加锁操作会中断自动清理。关于冲突锁模式，见表 13.2。但是，如果自动清理正在运行，其目的是防止事务 ID 回卷（也就是说，在 pg_stat_activity 视图中，自动清理查询名以 (to prevent wraparound) 结尾），则自动清理不会被自动中断。

> **警告**
>
> 定期运行需要获取与 SHARE UPDATE EXCLUSIVE 锁冲突的锁的命令（例如 ANALYZE），实际上可能会使自动清理永远无法完成。

24.2. 日常重建索引

在某些情况下，值得周期性地使用REINDEX 命令或一系列单独的重建步骤，来重建索引。

B-树索引中已经完全变空的索引页会被回收以供重用。不过，仍然可能出现空间利用低效：如果某个页面上的索引键除了少数几个之外都被删除，该页面仍会保留分配。因此，在一种使用模式下，如果每个键范围中的大多数但不是全部键最终都被删除，就会出现糟糕的空间利用。对于这种使用模式，建议定期重建索引。

对于非 B-树索引中的膨胀潜力，人们还没有做过充分研究。在使用任何非 B-树索引类型时，定期监控索引的物理大小是个好主意。

此外，对 B-树索引而言，新近构建的索引比已经多次更新过的索引在访问时稍快一些，因为在新建的索引中，逻辑上相邻的页面通常在物理上也彼此相邻。（这一考虑不适用于非 B-树索引。）即便只是为了提高访问速度，周期性重建索引也可能是值得的。

REINDEX 在所有情况下都能安全、方便地使用。但由于该命令需要表的排他锁，通常更适合通过一系列创建和替换步骤来重建索引。对于支持使用 CONCURRENTLY 选项执行 CREATE INDEX 的索引类型，可以改用这种方式重新创建。如果创建成功且生成的索引有效，就可以结合ALTER INDEX 和 DROP INDEX，用新建索引替换原索引。如果索引用于强制唯一性或其他约束，可能

需要使用 ALTER TABLE，将现有约束替换为由新索引强制执行的约束。使用这种多步骤重建方法前，应仔细评估，因为能够用这种方式重建的索引存在限制，而且必须处理错误。

24.3. 日志文件维护

最好把数据库服务器的日志输出保存到某个地方，而不是简单地把它丢弃到 `/dev/null`。诊断问题时，日志输出极其宝贵。

> **注意**
>
> 服务器日志可能包含敏感信息，因此无论它被如何、在何处存储，或被路由到什么目标，都需要受到保护。例如，某些 DDL 语句可能包含明文密码或其他认证细节。在 `ERROR` 级别记录的语句可能会暴露应用程序的 SQL 源代码，也可能包含部分数据行。记录数据、事件及相关信息，正是该设施的设计目的，因此这不属于信息泄露或程序错误。请确保服务器日志仅对获得适当授权的人可见。

日志输出往往十分庞大（尤其是在较高调试级别下），因此你不会希望无限期保存它。你需要对日志文件进行轮转，以便周期性地启用新的日志文件，并在合理时间后移除旧文件。

如果你只是把 `postgres` 的 stderr 重定向到一个文件，那么虽然也能得到日志输出，但截断该日志文件的唯一办法是停止并重新启动服务器。如果你是在开发环境中使用 PostgreSQL，这也许可以接受，但几乎没有生产服务器会认为这种行为可以接受。

更好的办法是把服务器的 stderr 输出发送给某种日志轮转程序。系统内置了日志轮转功能，你可以在 `postgresql.conf` 中将配置参数 `logging_collector` 设为 `true` 来使用它。该程序的控制参数见第 19.8.1 节。你还可以用这种方式以机器可读的 CSV（逗号分隔值）格式捕获日志数据。

也可以使用外部日志轮转程序，特别是已经在其他服务器软件上使用了某个此类程序时。例如，rotatelogs 工具包含在 Apache 发行版中，也可用于 PostgreSQL。为此，只需通过管道将服务器的 stderr 输出传递给所需程序。如果使用 `pg_ctl` 启动服务器，那么 stderr 已经被重定向到 stdout，因此只需一个管道命令，例如：

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

管理日志输出的另一种生产级方法，是把它发送给 syslog，并让 syslog 负责文件轮转。为此，在 `postgresql.conf` 中把配置参数 `log_destination` 设为 `syslog`（如果只想记录到 syslog）。之后，每当你想强制它开始写入新的日志文件时，都可以向 syslog 守护进程发送一个 `SIGHUP` 信号。如果你想自动化日志轮转，也可以把 logrotate 配置为处理来自 syslog 的日志文件。

不过，在很多系统上，syslog 并不十分可靠，尤其是在日志消息很大时；它可能恰恰在你最需要消息的时候截断或丢弃它们。此外，在 Linux 上，syslog 会把每条消息都刷盘，导致性能较差。（你可以在 syslog 配置文件的文件名开头使用一个“-”来禁用同步。）

请注意，上面描述的所有方案都处理了按可配置间隔启动新日志文件的问题，但它们并不处理删除那些已经没有用处的旧日志文件。你很可能还需要设置一个批处理任务，定期删除旧日志文件。另一种可能是把轮转程序配置为循环覆盖旧日志文件。

pgBadger[2] 是一个进行复杂日志文件分析的外部项目。check_postgres[3] 可以在日志文件中出现重要消息时向 Nagios 发出警报，也能检测许多其他异常情况。

[2] https://pgbadger.darold.net/
[3] https://bucardo.org/check_postgres/

第 25 章 备份和恢复

与任何保存重要数据的系统一样，PostgreSQL 数据库也应定期备份。虽然其过程基本简单，但清楚理解其底层技术和前提假设非常重要。

备份 PostgreSQL 数据主要有三种根本不同的方法：

- SQL 转储
- 文件系统级备份
- 持续归档

每种方法都有各自的优缺点，下面各节将依次讨论。

25.1. SQL 转储

这种转储方法的思路是生成一个包含 SQL 命令的文件；将其重新送回服务器后，服务器就会把数据库重建为转储时的状态。PostgreSQL 为此提供了工具程序 pg_dump。该命令的基本用法是：

```
pg_dump dbname > dumpfile
```

如前所示，pg_dump 把结果写到标准输出。稍后我们会看到这一点有何用处。虽然上述命令会创建一个文本文件，但pg_dump 也可以创建其他格式的文件，从而支持并行处理以及对对象恢复进行更细粒度的控制。

pg_dump 是一个普通的 PostgreSQL 客户端应用程序（尽管它实现得相当巧妙）。这意味着你可以从任何能够访问该数据库的远程主机执行这种备份过程。但请记住，pg_dump 并不会以特殊权限运行。特别是，它必须对你想要备份的所有表具有读权限，因此若要备份整个数据库，几乎总是必须以数据库超级用户身份运行它。（如果你没有足够权限备份整个数据库，仍然可以使用诸如`-n schema` 或 `-t table` 的选项来备份你有权访问的那部分数据库。）

要指定 pg_dump 应连接哪个数据库服务器，请使用命令行选项`-h host` 和 `-p port`。默认主机是本地主机，或者由环境变量`PGHOST` 指定的主机。类似地，默认端口由环境变量`PGPORT` 指定；如果未设置，则采用编译时内置的默认值。（方便的是，服务器通常也会使用同样的内置默认值。）

和其他 PostgreSQL 客户端应用程序一样，pg_dump 默认会以与当前操作系统用户名相同的数据库用户名进行连接。若要覆盖这一点，可以指定`-U` 选项，或者设置环境变量`PGUSER`。请记住，pg_dump 连接同样受常规客户端认证机制约束（见第 20 章）。

与后文介绍的其他备份方法相比，pg_dump 的一个重要优点在于，pg_dump 的输出通常可以重新装入较新版本的 PostgreSQL，而文件级备份和持续归档都高度依赖具体的服务器版本。pg_dump 也是在不同机器体系结构之间迁移数据库时唯一可行的方法，例如从 32 位服务器迁移到 64 位服务器。

由 pg_dump 创建的转储在内部是一致的，也就是说，该转储表示的是pg_dump 开始运行时的数据库快照。pg_dump 在工作期间不会阻塞数据库上的其他操作。（例外是那些需要独占锁的操作，例如大多数形式的`ALTER TABLE`。）

25.1.1. 恢复转储

由 pg_dump 创建的文本文件供 psql 程序读取。恢复转储的通用命令格式是

```
psql dbname < dumpfile
```

其中，*dumpfile* 是由 pg_dump 命令输出的文件。数据库 *dbname* 不会由此命令创建，因此必须自行基于 template0 创建它，再执行 psql（例如使用 createdb -T template0 *dbname*）。psql 支持与 pg_dump 类似的选项，用于指定要连接的数据库服务器及所用的用户名。更多信息参见psql 参考页面。非文本格式的转储使用 pg_restore 工具恢复。

在恢复 SQL 转储之前，所有拥有对象或者在被转储数据库中曾被授予对象权限的用户都必须已经存在。否则，恢复将无法以原有所有权和/或权限重新创建这些对象。（有时这正是你想要的，但通常不是。）

默认情况下，psql 脚本在遇到 SQL 错误后仍会继续执行。你可能希望让 psql 在设置了 ON_ERROR_STOP 变量的情况下运行，以改变这一行为，并让psql 在发生 SQL 错误时以退出状态码 3 退出：

```
psql --set ON_ERROR_STOP=on dbname < dumpfile
```

无论采用上述哪种方式，你最终只会得到一个部分恢复的数据库。另一种做法是指定将整个转储作为单个事务恢复，这样恢复要么全部完成，要么全部回滚。可以通过把-1 或 --single-transaction 命令行选项传给 psql 来启用这种模式。使用这种模式时要注意，即便是一个很小的错误，也可能回滚一个已经运行了许多小时的恢复过程。不过，这仍可能比在部分恢复后手工清理一个复杂数据库更可取。

pg_dump 和 psql 能够写入或读取管道，这使得可以把数据库直接从一台服务器转储到另一台服务器，例如：

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

> **重要**
>
> pg_dump 生成的转储是以 template0 为基准的。这意味着通过 template1 增加的任何语言、过程等也都会被pg_dump 转储。因此，恢复时如果你使用的是定制过的 template1，就必须像上面的例子那样，从template0 创建空数据库。

恢复完备份后，最好对每个数据库运行 ANALYZE，这样查询优化器就能拥有有用的统计信息；更多信息见第 24.1.3 节和第 24.1.6 节。关于如何高效地将大量数据装入 PostgreSQL，请参见第 14.4 节。

25.1.2. 使用 pg_dumpall

pg_dump 一次只转储一个数据库，而且不会转储角色或表空间的信息（因为这些是整个集簇级别的，而不是每个数据库各自的）。为了方便地转储整个数据库集簇的全部内容，提供了 pg_dumpall 程序。pg_dumpall 会备份给定集簇中的每个数据库，并保留角色定义和表空间定义等集簇范围的数据。该命令的基本用法是：

```
pg_dumpall > dumpfile
```

生成的转储可以用 psql 恢复：

```
psql -f dumpfile postgres
```

（实际上，可以指定任意一个现有数据库名作为起点，但如果要装载到一个空集簇中，通常应使用`postgres`。）恢复 pg_dumpall 转储时始终需要数据库超级用户权限，因为恢复角色和表空间信息必须使用该权限。如果使用了表空间，请确保转储中的表空间路径适合新的安装。

pg_dumpall 的工作方式是先发出重新创建角色、表空间和空数据库的命令，然后针对每个数据库调用pg_dump。这意味着每个数据库本身都是内部一致的，但不同数据库的快照并不同步。

集簇范围的数据也可以单独使用 pg_dumpall 的 `--globals-only` 选项进行转储。如果你对各个数据库分别运行 pg_dump 命令，而又希望对整个集簇进行完整备份，这一步就是必需的。

25.1.3. 处理大型数据库

某些操作系统对文件大小有上限，这会在创建大型pg_dump 输出文件时带来问题。幸运的是，pg_dump 可以把输出写到标准输出，因此你可以利用标准 Unix 工具绕过这个潜在问题。有几种可行方法：

使用压缩转储。. 你可以使用自己喜欢的压缩程序，例如gzip：

```
pg_dump dbname | gzip > filename.gz
```

恢复时：

```
gunzip -c filename.gz | psql dbname
```

或者：

```
cat filename.gz | gunzip | psql dbname
```

使用 split。。 split 命令允许你把输出分割成文件系统可接受大小的较小文件。例如，要切分为 2 GB 的块：

```
pg_dump dbname | split -b 2G - filename
```

恢复时：

```
cat filename* | psql dbname
```

如果使用 GNU split，也可以把它和gzip 结合起来：

```
pg_dump dbname | split -b 2G --filter='gzip > $FILE.gz'
```

恢复时可使用 zcat。

使用 pg_dump 的自定义转储格式。。 如果构建 PostgreSQL 所用的系统安装了 zlib 压缩库，自定义转储格式会在向输出文件写入数据时对其进行压缩。这会产生与使用gzip 大致相同大小的转储文件，但额外的优点是其中的表可以有选择地恢复。下面的命令使用自定义转储格式转储数据库：

```
pg_dump -Fc dbname > filename
```

自定义格式的转储不是供 psql 执行的脚本，而必须通过pg_restore 恢复，例如：

```
pg_restore -d dbname filename
```

详细信息见 pg_dump 和 pg_restore 参考页。

对于非常大的数据库，你可能需要把split 与另外两种方法之一结合使用。

使用 pg_dump 的并行转储特性。。 为了加快大型数据库的转储速度，你可以使用pg_dump 的并行模式。这会同时转储多个表。你可以通过-j 参数控制并行度。并行转储只支持“目录”归档格式。

```
pg_dump -j num -F d -f out.dir dbname
```

你可以使用 pg_restore -j 并行恢复转储。这对于任何“自定义”或“目录”模式的归档都有效，无论它是否是用pg_dump -j 创建的。

25.2. 文件系统级备份

另一种备份策略是直接复制 PostgreSQL 用来存储数据库数据的文件；第 18.2 节解释了这些文件的位置。你可以使用自己喜欢的任何方法进行文件系统备份，例如：

```
tar -cf backup.tar /usr/local/pgsql/data
```

不过，这种方法有两个限制，使得它并不实用，至少也不如pg_dump 方法：

1. 为了得到可用的备份，数据库服务器必须关闭。像禁止所有连接这样的折中办法不起作用（部分原因是 `tar` 和类似工具不会对文件系统状态执行原子快照，另一部分原因是服务器内部也存在缓冲）。有关停止服务器的信息见第 18.5 节。不言而喻，恢复数据前也必须关闭服务器。

2. 如果你已经深入了解数据库文件系统布局的细节，可能会想尝试仅通过某些表或数据库各自的文件或目录来备份或恢复它们。这是行不通的，因为这些文件中的信息离开记录所有事务提交状态的提交日志文件`pg_xact/*` 就无法使用。一个表文件只有和这些信息一起才可用。当然，也不可能只恢复一个表以及相关的`pg_xact` 数据，因为那会让数据库集簇中的其他所有表都无法使用。因此，文件系统备份只适用于整个数据库集簇的完整备份与恢复。

另一种文件系统备份方法是对数据目录制作一个“一致快照”，前提是文件系统支持该功能（并且你愿意相信其实现是正确的）。典型流程是：对包含数据库的卷制作一个“冻结快照”，然后从该快照把整个数据目录（不是其中一部分，见上文）复制到备份设备，再释放这个冻结快照。即使数据库服务器正在运行，这种方法也能工作。不过，以这种方式创建的备份会把数据库文件保存为一种仿佛服务器没有被正确关闭的状态；因此，当你基于备份数据启动数据库服务器时，它会认为前一个服务器实例发生了崩溃，并重放 WAL 日志。这不是问题；只要意识到这一点即可（并且务必把 WAL 文件也包含在备份中）。你可以在获取快照之前执行一次`CHECKPOINT`，以缩短恢复时间。

如果数据库分布在多个文件系统上，就可能根本无法对所有卷获得完全同步的冻结快照。例如，如果数据文件和 WAL 日志位于不同磁盘，或者表空间位于不同文件系统上，就可能无法使用快照备份，因为这些快照必须同时获取。在这种情况下，在信任一致快照技术之前，一定要非常仔细地阅读文件系统文档。

如果无法获得同时的快照，一种选择是将数据库服务器关闭足够长的时间，以建立所有冻结快照。另一种选择是执行持续归档基础备份（第 25.3.2 节），因为这种备份不受备份期间文件系统变化的影响。这种方法只需在备份期间启用持续归档；恢复则使用持续归档恢复（第 25.3.4 节）。

另一种选择是使用 rsync 进行文件系统备份。做法是先在数据库服务器运行时执行一次 rsync，然后将数据库服务器关闭足够长的时间，再执行一次`rsync --checksum`。（之所以需要`--checksum`，是因为`rsync` 对文件修改时间的粒度只有 1 秒。）第二次 rsync 会比第一次更快，因为它需要传输的数据相对较少，而最终结果会因为服务器已经关闭而保持一致。这种方法可以在最小停机时间下完成文件系统备份。

注意，文件系统备份通常会比 SQL 转储更大。（例如，pg_dump 不需要转储索引内容，只需转储重建索引的命令。）不过，进行文件系统备份可能更快。

25.3. 持续归档和时间点恢复（PITR）

在任何时候，PostgreSQL 都会维护一个预写式日志（WAL），它位于集簇数据目录的`pg_wal/`子目录中。该日志记录对数据库数据文件所做的每一次修改。这个日志首先是为崩溃安全而存在：如果系统崩溃，可以通过“重放”自上次检查点以来的日志记录，将数据库恢复到一致状态。不过，日志的存在也使第三种数据库备份策略成为可能：我们可以把文件系统级备份与 WAL 文件备份结合起来。如果需要恢复，就先恢复文件系统备份，再重放已备份的 WAL 文件，使系统回到当前状态。与前两种方法相比，这种方法管理起来更复杂，但它有一些显著优点：

- 我们不需要一个完全一致的文件系统备份作为起点。备份中的任何内部不一致性都会通过日志重放纠正（这与崩溃恢复期间发生的事情并没有本质不同）。因此，我们不需要文件系统快照能力，只需要tar 或类似的归档工具。

- 由于我们可以把任意长的一串 WAL 文件串联起来进行重放，只要持续归档 WAL 文件，就可以实现连续备份。这对于大型数据库尤其有价值，因为此时频繁进行完整备份可能并不方便。
- 不必把 WAL 记录一直重放到最后。我们可以在任意位置停止重放，并得到数据库在当时的一致快照。因此，这项技术支持时间点恢复：可以把数据库恢复到自基础备份之后任意时刻的状态。
- 如果我们持续把这一串 WAL 文件输送给另一台已经装载了同一基础备份文件的机器，就得到了一个温备系统：我们随时都可以启用第二台机器，而它将拥有一个几乎是最新的数据库副本。

> **注意**
>
> pg_dump 和 pg_dumpall 不会生成文件系统级备份，因此不能作为持续归档方案的一部分使用。这类转储是逻辑备份，不包含 WAL 重放所需的足够信息。

和普通文件系统备份技术一样，这种方法只能支持整个数据库集簇的恢复，而不支持其中某个子集的恢复。此外，它需要大量归档存储：基础备份可能很庞大，繁忙系统也会产生许多兆字节的、必须归档的 WAL 流量。尽管如此，在很多需要高可靠性的场景中，它仍是首选的备份技术。

要想利用持续归档（许多数据库厂商也称之为“在线备份”）成功恢复，你需要一串连续的已归档 WAL 文件，其时间至少要追溯到备份的开始时刻。因此，入门时应在进行第一次基础备份之前就建立并测试好归档 WAL 文件的流程。下面先讨论归档 WAL 文件的机制。

25.3.1. 设置 WAL 归档

从抽象意义上说，一个运行中的PostgreSQL 系统会产生一串无限长的 WAL 记录序列。系统在物理上把该序列划分为 WAL 段文件，通常每个 16MB（段大小可在initdb 期间修改）。这些段文件会被赋予数字名称，以反映它们在抽象 WAL 序列中的位置。不使用 WAL 归档时，系统通常只创建少量段文件，然后通过把不再需要的段文件重命名为更高的段号来“回收”它们。系统假定内容早于最后一个检查点的段文件已不再有价值，因此可以被回收。

在归档 WAL 数据时，我们需要在每个段文件写满后捕获其内容，并在该段文件被回收重用之前把数据保存到某处。根据应用场景和可用硬件的不同，“把数据保存到某处”可以有很多不同的方法：可以把段文件复制到另一台机器上的 NFS 挂载目录，把它们写到磁带机中（确保你有办法识别每个文件的原始文件名），把它们批量打包后刻录到 CD 上，或者采用完全不同的方式。为了给数据库管理员提供灵活性，PostgreSQL 尽量不对归档方式作任何假设。相反，PostgreSQL 允许管理员指定一个 shell 命令，以便把已完成的段文件复制到它应去的地方。该命令可以只是一次简单的`cp` 调用，也可以调用一个复杂的 shell 脚本，全由你决定。

要启用 WAL 归档，请将wal_level 配置参数设为 replica 或更高值，将archive_mode 设为 on，并在archive_command 配置参数中指定要执行的 shell 命令。实际使用中，这些设置始终放在 postgresql.conf 文件中。对于 archive_command，%p 会被替换为要归档文件的路径名，而 %f 只会被替换为文件名。（路径名相对于当前工作目录，即集簇的数据目录。）使用 %% 可以在命令中嵌入实际的 % 字符。最简单的可用命令类似于：

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'  #
 Unix
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"'  # Windows
```

这会将可归档的 WAL 段复制到目录 /mnt/server/archivedir。（这只是示例，并非推荐做法，而且不一定适用于所有平台。）在替换 %p 和 %f 参数后，实际执行的命令可能如下：

```
test ! -f /mnt/server/archivedir/000000010000000A900000065 && cp pg_wal/
000000010000000A900000065 /mnt/server/archivedir/000000010000000A900000065
```

每个需要归档的新文件都会生成一条类似的命令。

归档命令将以运行 PostgreSQL 服务器的同一用户身份执行。由于归档的一系列 WAL 文件实际上包含数据库中的全部内容，应确保归档数据不会被他人窥视；例如，将其归档到不允许所属组或其他用户读取的目录。

归档命令必须当且仅当成功时才返回退出状态零。收到零状态后，PostgreSQL 会认为该文件已成功归档，并将其删除或回收。非零状态则告诉 PostgreSQL 该文件尚未归档；它会定期重试，直到成功。

通常应将归档命令设计为拒绝覆盖任何已存在的归档文件。这是一项重要的安全保护措施，能在管理员操作失误时（例如将两台不同服务器的输出发送到同一归档目录）维护归档的完整性。

建议测试所拟定的归档命令，确保它确实不会覆盖已有文件，并且在这种情况下返回非零状态。上面给出的 Unix 示例命令通过单独加入一个 test 步骤，确保这两点。在某些 Unix 平台上，cp 提供了诸如 -i 之类的开关，也可以更简洁地实现同样目的，但在你确认它会返回正确的退出状态之前，不应依赖这些开关。（尤其是 GNU cp 在使用 -i 且目标文件已存在时会返回状态零，这不是我们想要的行为。）

在设计归档方案时，请考虑如果归档命令因为某些环节需要操作员干预，或者归档空间耗尽而反复失败，会发生什么。例如，如果你在没有自动换带器的情况下向磁带写入数据，那么磁带满了以后，在更换磁带之前将无法继续归档。你应确保任何错误情况或对人工操作员的请求都能得到适当报告，以便问题能够比较快地解决。在问题解决之前，pg_wal/目录会继续堆积 WAL 段文件。（如果包含 pg_wal/的文件系统被写满，PostgreSQL 将执行 PANIC 关闭。不会丢失已提交的事务，但在释放出一些空间之前，数据库会一直离线。）

归档命令的速度并不重要，只要它能跟上服务器生成 WAL 数据的平均速度即可。即使归档过程稍有滞后，正常操作也会继续。如果归档明显落后，灾难发生时可能丢失的数据量就会增加。这还意味着pg_wal/目录会包含大量尚未归档的段文件，最终可能耗尽可用磁盘空间。建议监控归档过程，确保它按你的预期工作。

在编写归档命令时，应假定待归档文件名最长可达 64 个字符，并且可以包含 ASCII 字母、数字和点号的任意组合。无需保留原始相对路径（%p），但必须保留文件名（%f）。

注意，虽然 WAL 归档允许你恢复对PostgreSQL 数据库中数据所做的任何修改，但它不会恢复对配置文件（即postgresql.conf、pg_hba.conf 和 pg_ident.conf）的修改，因为这些文件是手工编辑的，而不是通过 SQL 操作修改的。你可能希望把配置文件放在常规文件系统备份过程能够覆盖的位置。关于如何重定位配置文件，见第 19.2 节。

归档命令只会针对已完成的 WAL 段调用。因此，如果服务器产生的 WAL 流量很小（或者存在低谷期），事务完成到其被安全写入归档存储之间可能会有很长延迟。为了限制未归档数据可能有多旧，你可以设置 archive_timeout，使服务器强制切换到新 WAL 段文件的间隔不超过这个值。注意，由强制切换而提前归档的文件长度仍与装满的文件相同。因此，把archive_timeout 设得很短并不明智，这会使归档存储膨胀。archive_timeout 设为大约 1 分钟通常是合理的。

此外，如果你希望确保一个刚刚完成的事务尽快被归档，可以使用pg_switch_wal 手工强制一次段切换。其他与 WAL 管理相关的实用函数列在表 9.79 中。

当 wal_level 为 minimal 时，某些 SQL 命令会像第 14.4.7 节中所述那样被优化为避免 WAL 记录。如果在执行这些语句期间启用了归档或流复制，WAL 将不包含归档恢复所需的足够信息。（崩溃恢复不受影响。）因此，wal_level 只能在服务器启动时更改。然而，archive_command 可以通过重新加载配置文件来更改。如果你希望暂时停止归档，一种办法是将archive_command 设置为空字符串（''）。这会导致 WAL 文件在 pg_wal/中累积，直到重新建立可用的archive_command。

25.3.2. 进行基础备份

执行基础备份最简单的方法是使用 pg_basebackup 工具。它可以把基础备份创建为普通文件或 tar 归档。如果需要比 pg_basebackup 提供的更高灵活性，也可以使用低级 API 制作基础备份（见第 25.3.3 节）。

不必过于担心制作基础备份所需的时间。不过，如果你平时在关闭full_page_writes 的情况下运行服务器，可能会注意到备份运行期间性能下降，因为在备份模式下full_page_writes 实际上会被强制开启。

要让该备份可用，你需要保留在文件系统备份期间以及之后生成的所有 WAL 段文件。为帮助完成这件事，基础备份过程会创建一个备份历史文件，并立即将其存入 WAL 归档区域。该文件以文件系统备份所需的第一个 WAL 段文件命名。例如，如果起始 WAL 文件是000000010000123400000055CD，备份历史文件的名称将类似于000000010000123400000055CD.007C9330.backup。（文件名第二部分表示该 WAL 文件中的一个精确位置，通常可以忽略。）一旦你已经安全归档了文件系统备份以及备份期间使用到的 WAL 段文件（如备份历史文件所指定），所有名称在数值上更小的已归档 WAL 段就不再是恢复该文件系统备份所必需的，可以删除。不过，你仍应考虑保留多个备份集，以绝对确保能够恢复数据。

备份历史文件只是一个很小的文本文件。它包含你提供给 pg_basebackup 的标签字符串，以及备份的起止时间和起止 WAL 段。如果你用该标签标识了关联的备份文件，那么已归档的历史文件就足以告诉你应恢复哪个备份文件。

由于你必须保留自上一次基础备份以来的所有已归档 WAL 文件，基础备份之间的间隔通常应根据你愿意为已归档 WAL 文件投入多少存储空间来决定。你还应考虑在确实需要恢复时，你愿意花多长时间进行恢复，因为系统必须重放所有这些 WAL 段；如果距离上次基础备份已经过去很久，这可能会花费一些时间。

25.3.3. 使用低级 API 进行基础备份

使用低级 API 制作基础备份的过程，比pg_basebackup 方法多几个步骤，但相对简单。务必按顺序执行这些步骤，并在继续下一步之前确认当前步骤成功。

低级基础备份可以采用非排他或排他方式。建议使用非排他方式；排他方式已弃用，最终将被移除。

25.3.3.1. 制作非排他低级备份

非排他低级备份允许同时运行其他备份，包括使用同一备份 API 启动的备份和使用 pg_basebackup 启动的备份。

1. 确保 WAL 归档已启用并正常工作。

2. 以有权运行 pg_start_backup 的用户身份（超级用户，或已被授予该函数 EXECUTE 权限的用户）连接到服务器（连接哪个数据库都可以），并执行命令：

   ```
   SELECT pg_start_backup('label', false, false);
   ```

   其中，label 是用于唯一标识此次备份操作的任意字符串。调用 pg_start_backup 的连接必须一直保持到备份结束，否则备份会自动中止。

   默认情况下，pg_start_backup 可能需要很长时间才能完成。这是因为它会执行一次检查点，而检查点所需的 I/O 会分散在较长时间内，默认是检查点间隔的一半（参见配置参数 checkpoint_completion_target）。这通常是理想的行为，因为它尽量减少了对查询处理的影响。如果希望尽快开始备份，请将第二个参数改为 true，这会尽可能利用可用 I/O 立即执行检查点。

   第三个参数为 false，会告知 pg_start_backup 启动非排他基础备份。

3. 使用任何方便的文件系统备份工具执行备份，例如tar 或 cpio（不要使用 pg_dump 或 pg_dumpall）。在此过程中既没有必要，也不希望停止数据库的正常运行。关于执行此备份时需要注意的事项，见第 25.3.3.3 节。

4. 在之前的同一个连接中，执行命令：

   ```
   SELECT * FROM pg_stop_backup(false, true);
   ```

   这会终止备份模式。在主库上，还会自动切换到下一个 WAL 段。在备库上，无法自动切换 WAL 段，因此可以运行 pg_switch_wal，在主库上手动切换。切换的目的是让备份期间写入的最后一个 WAL 段文件准备好归档。

   pg_stop_backup 会返回一行，包含三个值。其中第二个字段应写入备份根目录下名为 backup_label 的文件中。第三个字段除非为空，否则应写入名为tablespace_map 的文件中。这些文件对备份能否正常工作至关重要，必须逐字节原样写入，不能做任何修改，这可能意味着需要以二进制模式打开文件。

5. 一旦备份期间活跃的 WAL 段文件都已归档，备份就完成了。由pg_stop_backup 第一个返回值标识的文件，是形成完整备份文件集所需的最后一个段。在主库上，如果启用了archive_mode 且 wait_for_archive 参数为 true，则pg_stop_backup 在最后一个段归档前不会返回。在备库上，只有当archive_mode 为 always 时，pg_stop_backup 才会等待。由于你已经配置好了 archive_command，这些文件的归档会自动进行。多数情况下这会很快完成，但仍建议监控归档系统，确保没有延迟。如果归档进程因归档命令失败而落后，它会持续重试，直到归档成功并且备份完成。如果你希望对 pg_stop_backup 的执行设置时间限制，请设置合适的statement_timeout 值，但要注意如果pg_stop_backup 因此终止，备份可能无效。

   如果备份过程本身会监控并确保备份所需的所有 WAL 段文件都已成功归档，则可以把 wait_for_archive 参数（默认值为 true）设置为 false，使pg_stop_backup 在停止备份记录写入 WAL 后立即返回。默认情况下，pg_stop_backup 会等待直到所有 WAL 都已

归档，这可能需要一段时间。必须谨慎使用此选项：如果没有正确监控 WAL 归档，备份可能不包含全部 WAL 文件，因此会是不完整且无法恢复的。

25.3.3.2. 制作排他低级备份

排他备份的过程与非排他备份基本相同，但有几个关键步骤不同。这种备份只能在主库上进行，且不允许并发备份。在 PostgreSQL 9.6 之前，这是唯一可用的低级方法，但现在建议所有用户尽可能更新脚本，改用非排他备份。

1. 确保 WAL 归档已启用并正常工作。

2. 以有权运行 pg_start_backup 的用户身份（超级用户，或已被授予该函数 EXECUTE 权限的用户）连接到服务器（连接哪个数据库都可以），并执行命令：

   ```
   SELECT pg_start_backup('label');
   ```

   其中，`label` 是用于唯一标识此次备份操作的任意字符串。`pg_start_backup` 会创建一个备份标签文件，名为 `backup_label`，位于集簇目录中，包含备份信息，例如开始时间和标签字符串。该函数还会创建一个表空间映射文件，名为 `tablespace_map`，位于集簇目录中，包含 `pg_tblspc/` 中表空间符号链接的信息（如果存在一个或多个这样的链接）。如果需要从备份恢复，这两个文件对备份的完整性都至关重要。

   默认情况下，`pg_start_backup` 可能需要很长时间才能完成。这是因为它会执行一次检查点，而检查点所需的 I/O 会分散在较长时间内，默认是检查点间隔的一半（参见配置参数 checkpoint_completion_target）。这通常是理想的行为，因为它尽量减少了对查询处理的影响。如果希望尽快开始备份，请使用：

   ```
   SELECT pg_start_backup('label', true);
   ```

   这会强制尽快完成检查点。

3. 使用任何方便的文件系统备份工具执行备份，例如tar 或 cpio（不要使用 pg_dump 或 pg_dumpall）。在此过程中既没有必要，也不希望停止数据库的正常运行。关于执行此备份时需要注意的事项，见第 25.3.3.3 节。

   注意，如果服务器在备份期间崩溃，可能必须手动删除 `PGDATA` 目录中的 `backup_label` 文件后才能重启。

4. 再次以有权运行 pg_stop_backup 的用户身份（超级用户，或已被授予该函数 EXECUTE 权限的用户）连接到数据库，并执行命令：

   ```
   SELECT pg_stop_backup();
   ```

   该函数会终止备份模式，并自动切换到下一个 WAL 段。切换的目的是让备份期间写入的最后一个 WAL 段准备好归档。

5. 备份期间处于活动状态的 WAL 段文件全部归档后，备份就完成了。pg_stop_backup 返回结果所标识的文件，是构成完整备份文件集所需的最后一个段。如果启用了 archive_mode，pg_stop_backup 会等到最后一个段归档后才返回。由于已经配置了 archive_command，这些文件会自动归档。大多数情况下，归档很快就能完成，但建议监控归档系统，确保没有延迟。如果归档进程因归档命令失败而落后，它会不断重试，直到归档成功、备份完成。如果希望限制 pg_stop_backup 的执行时间，可以设置合适的 statement_timeout 值，但要注意，如果 pg_stop_backup 因此终止，备份可能无效。

25.3.3.3. 备份数据目录

某些文件系统备份工具在复制过程中，如果它们试图复制的文件发生变化，就会发出警告或错误。对活动数据库进行基础备份时，这种情况是正常的，并不表示出错。不过，你需要确保能够把这类提示与真正的错误区分开来。例如，某些版本的rsync 会针对 “vanished source files” 返回单独的退出码，你可以编写一个驱动脚本，把这一退出码视为非错误情况。此外，某些版本的 GNU tar 在文件被 tar 复制时如果发生截断，会返回与致命错误无法区分的错误码。幸运的是，GNU tar 1.16 及之后版本在备份过程中如果文件被更改会以 1 退出，而其他错误则以 2 退出。对于 GNU tar 1.23 及之后版本，可以使用警告选项--warning=no-file-changed --warning=no-file-removed 隐藏相关警告信息。

务必确认你的备份包含数据库集簇目录（例如/usr/local/pgsql/data）下的全部文件。如果你使用的表空间不位于该目录之下，也要记得把它们包括进来（并确保备份把符号链接作为链接归档，否则恢复时会破坏表空间）。

不过，备份中应省略集簇pg_wal/子目录内的文件。这个小调整很值得，因为它能降低恢复时出错的风险。如果pg_wal/是一个指向集簇目录外某处的符号链接，就很容易做到这一点，而出于性能原因，这本来也是一种常见配置。你也可能想排除postmaster.pid 和 postmaster.opts，它们记录的是正在运行的postmaster 的信息，而不是最终使用此备份的 postmaster 的信息。（这些文件可能会让 pg_ctl 感到困惑。）

通常也最好省略集簇 pg_replslot/目录中的文件，以免主库上的复制槽成为备份的一部分。否则，之后用该备份创建备库时，可能导致该备库上的 WAL 文件无限期保留；如果启用了热备反馈，也可能导致主库膨胀，因为使用这些复制槽的客户端仍会连接到并更新主库上的槽，而不是备库上的槽。即使该备份只是用来创建新的主库，复制这些复制槽通常也没有多大意义，因为等新主库上线时，这些槽的内容很可能已经严重过时。

备份可以省略 pg_dynshmem/、pg_notify/、pg_serial/、pg_snapshots/、pg_stat_tmp/ 和 pg_subtrans/ 目录中的内容（但不能省略目录本身），因为它们会在 postmaster 启动时初始化。如果设置了stats_temp_directory 且该目录位于数据目录中，也可以省略其内容。

任何以 pgsql_tmp 开头的文件或目录都可以从备份中省略。这些文件会在 postmaster 启动时被删除，而这些目录也会在需要时重新创建。

只要发现名为 pg_internal.init 的文件，就可以把它从备份中省略。这些文件包含关系缓存数据，而这些数据在恢复时总会被重新构建。

备份标签文件包含你提供给 pg_start_backup 的标签字符串、执行pg_start_backup 的时间以及起始 WAL 文件的名称。因此，在发生混淆时，可以查看备份文件内容，精确确定该备份文件来自哪一次备份会话。表空间映射文件包含目录pg_tblspc/中存在的符号链接名称，以及每个符号链接的完整路径。这些文件不仅仅是给你参考；它们的存在及其内容对于系统恢复过程的正确运行至关重要。

服务器停止时也可以进行备份。在这种情况下，你显然无法使用pg_start_backup 或 pg_stop_backup，因此只能自己追踪各个备份的身份以及相关 WAL 文件最早需要追溯到哪里。通常最好还是遵循上面的持续归档过程。

25.3.4. 使用持续归档备份进行恢复

现在，最坏的情况发生了，你需要通过备份进行恢复。步骤如下：

1. 如果服务器仍在运行，就先停止它。

2. 如果你有足够空间，请把整个集簇数据目录以及所有表空间复制到临时位置，以备后续需要。注意，这一预防措施要求系统有足够空闲空间来保存现有数据库的两份副本。如果空间不够，至少也应保存集簇pg_wal 子目录中的内容，因为其中可能包含系统停机前尚未归档的 WAL 文件。

3. 删除集簇数据目录下以及所有正在使用的表空间根目录下的现有文件和子目录。

4. 从文件系统备份中恢复数据库文件。务必确保它们以正确的所有者（数据库系统用户，而不是root！）和正确的权限恢复。如果使用了表空间，还应验证pg_tblspc/中的符号链接是否已正确恢复。

5. 删除 pg_wal/中现有的所有文件；这些文件来自文件系统备份，因此很可能已经过时而不是最新的。如果你根本没有归档pg_wal/，那么就以正确权限重新创建它，并注意如果它原先是符号链接，就要重新把它设置成符号链接。

6. 如果你手头还有第 2 步中保存下来的未归档 WAL 段文件，请把它们复制到pg_wal/中。（最好复制，而不是移动，这样一旦出问题需要重来时，你手里仍然保留着未修改的原始文件。）

7. 在集簇数据目录中创建恢复命令文件 recovery.conf（参见第 27 章）。也可以临时修改 pg_hba.conf，在确认恢复成功之前阻止普通用户连接。

8. 启动服务器。服务器会进入恢复模式，依次读取所需的已归档 WAL 文件。如果恢复因外部错误而终止，只需重启服务器，它就会继续恢复。恢复过程完成后，服务器会将 recovery.conf 重命名为 recovery.done（防止以后意外重新进入恢复模式），然后开始正常的数据库操作。

9. 检查数据库内容，确认你已经恢复到期望状态。如果不是，就回到第 1 步。如果一切正常，就把pg_hba.conf 恢复为正常设置，让用户重新连接。

整个过程的关键，是设置一个恢复配置文件，说明希望如何恢复，以及恢复到什么位置。可以使用 recovery.conf.sample（通常位于安装目录的 share/ 目录中）作为模板。必须在 recovery.conf 中指定 restore_command，它告诉 PostgreSQL 如何获取已归档的 WAL 段文件。与 archive_command 一样，它是一个 shell 命令字符串，可以包含 %f，该标记会被替换为所需日志文件的名称；还可以包含 %p，该标记会被替换为日志文件要复制到的路径名。（路径名相对于当前工作目录，即集簇的数据目录。）使用 %% 可以在命令中嵌入实际的 % 字符。最简单的可用命令类似于：

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

这会复制先前归档的 WAL 段，来源目录为 /mnt/server/archivedir。当然，也可以使用复杂得多的命令，甚至可以使用要求操作人员挂载相应磁带的 shell 脚本。

重要的是，该命令在失败时必须返回非零退出状态。系统会调用该命令来请求归档中不存在的文件；遇到这种情况时，它必须返回非零值。这不是一种错误情况。例外是，如果该命令被信号终止（用于数据库服务器关闭的SIGTERM 除外），或者因 shell 错误（如命令未找到）而失败，那么恢复将中止，服务器也不会启动。

被请求的文件并不全都是 WAL 段文件；你还应预期会收到对带有 .history 后缀文件的请求。另外请注意，%p 路径的文件名部分会与 %f 不同；不要指望它们可以互换使用。

在归档中找不到的 WAL 段会转而在 pg_wal/中查找；这使得可以使用最近尚未归档的段。不过，凡是归档中可用的段，都会优先于pg_wal/中的文件使用。

通常，恢复会处理完所有可用的 WAL 段，从而把数据库恢复到当前时间点（或者在可用 WAL 段所允许的情况下尽可能接近当前时间点）。因此，一次正常恢复会以一条 “file not found” 消息结束，具体错误文本取决于你选择的restore_command。在恢复开始时，你也可能看到一条针对类似00000001.history 文件的错误消息。这同样是正常的，在简单恢复场景中并不表示有问题；相关讨论见第 25.3.5 节。

如果希望恢复到过去的某个时间点（例如初级 DBA 删除主要事务表之前），只需在 recovery.conf 中指定所需的停止点。这个停止点称为 “恢复目标”，可以通过日期/时间、命名恢复点，或某个特定事务 ID 的完成来指定。撰写本文时，只有日期/时间和命名恢复点这两种方式比较实用，因为还没有工具能帮助你准确确定应使用哪个事务 ID。

> **注意**
>
> 停止点必须晚于基础备份的结束时间，也就是pg_stop_backup 的结束时间。你不能用某次基础备份恢复到该备份仍在进行中的时间点。（若要恢复到这样的时间点，必须回到更早的一次基础备份，再从那里向前重放日志。）

如果恢复过程中发现了损坏的 WAL 数据，恢复会在该点停止，服务器也不会启动。在这种情况下，可以从头重新执行恢复，并指定一个位于损坏点之前的 “恢复目标”，使恢复能够正常完成。如果恢复因外部原因失败，例如系统崩溃或 WAL 归档变得不可访问，那么只需重新启动恢复，它几乎会从上次失败的位置继续。恢复重启的工作方式很像正常运行时的检查点：服务器会周期性地把自身状态强制写盘，然后更新pg_control 文件，表明已处理过的 WAL 数据无需再次扫描。

25.3.5. 时间线

能够将数据库恢复到过去某个时间点，也会带来一些类似科幻故事中时间旅行和平行宇宙的复杂情况。例如，假设在数据库原来的历史中，你在周二下午 5:15 删除了一张重要的表，直到周三中

午才发现错误。你从容地取出备份，将数据库恢复到周二下午 5:14，然后重新投入运行。在数据库宇宙的这段历史中，你从未删除过那张表。但假设你后来发现这样做不太合适，希望回到原来历史中的周三上午某个时刻。如果数据库恢复运行后覆盖了通往该时刻所需的某些 WAL 段文件，就无法回去了。因此，为了避免这种情况，需要区分时间点恢复后产生的一系列 WAL 记录与数据库原来历史中产生的记录。

为解决这一问题，PostgreSQL 引入了时间线的概念。每当归档恢复完成，系统都会创建一条新时间线，用于标识此次恢复之后产生的一系列 WAL 记录。时间线 ID 是 WAL 段文件名的一部分，因此新时间线不会覆盖旧时间线产生的 WAL 数据。实际上，可以归档许多不同的时间线。这个功能看似没什么用，却常常能救急。比如，你不能确定应该恢复到哪个时间点，需要反复尝试时间点恢复，直到找到脱离旧历史的最佳分支点。没有时间线，这个过程很快就会乱得无法管理。有了时间线，就可以恢复到任何先前的状态，包括早先已放弃的时间线分支中的状态。

每当创建一条新的时间线时，PostgreSQL 都会创建一个“时间线历史”文件，记录它是从哪条时间线、在何时分叉出来的。当从包含多条时间线的归档中恢复时，这些历史文件对于系统选取正确的 WAL 段文件是必需的。因此，它们会像 WAL 段文件一样被归档到 WAL 归档区域。历史文件只是很小的文本文件，因此长期保存它们既便宜也合适（而段文件通常很大）。如果你愿意，还可以在历史文件中加入注释，记录创建这条时间线的方式和原因。当你因实验而积累出一批错综复杂的时间线时，这类注释会特别有价值。

默认情况下，恢复会沿着制作基础备份时的当前时间线进行。如果希望恢复到某条子时间线（即返回到一次恢复尝试之后产生的某个状态），需要在 `recovery.conf` 中指定目标时间线 ID。不能恢复到在基础备份之前就已分支出去的时间线。

25.3.6. 建议和示例

这里给出一些配置持续归档的建议。

25.3.6.1. 独立热备份

可以利用 PostgreSQL 的备份设施生成独立热备份。这些备份不能用于时间点恢复，但它们的制作和恢复通常都比pg_dump 转储快得多。（它们也比 pg_dump 转储大得多，因此在某些情况下速度优势可能会被抵消。）

和基础备份一样，生成独立热备份最简单的方法是使用pg_basebackup 工具。如果在调用它时包含`-X` 参数，使用该备份所需的全部预写式日志都会自动包含在备份中，恢复该备份时也不需要额外动作。

如果复制备份文件时需要更大的灵活性，也可以使用更低级的流程制作独立热备份。要准备低级独立热备份，请确保将 `wal_level` 设为 `replica` 或更高值，将 `archive_mode` 设为 `on`，并配置一个 `archive_command`，使其仅在开关文件存在时执行归档。例如：

```
archive_command = 'test ! -f /var/lib/pgsql/backup_in_progress || (test ! -f /var/lib/pgsql/
archive/%f && cp %p /var/lib/pgsql/archive/%f)'
```

当 `/var/lib/pgsql/backup_in_progress` 存在时，这条命令会执行归档；否则会静默地返回零退出状态（允许 PostgreSQL 回收不需要的 WAL 文件）。

做好上述准备后，就可以使用类似下面的脚本进行备份：

```
touch /var/lib/pgsql/backup_in_progress
psql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/pgsql/backup.tar /var/lib/pgsql/data/
psql -c "select pg_stop_backup();"
rm /var/lib/pgsql/backup_in_progress
tar -rf /var/lib/pgsql/backup.tar /var/lib/pgsql/archive/
```

首先创建开关文件 `/var/lib/pgsql/backup_in_progress`，以启用已完成 WAL 文件的归档。备份完成后删除该开关文件。随后将已归档的 WAL 文件加入备份，使基础备份与所有必需的 WAL 文件都包含在同一个 tar 文件中。请记得在备份脚本中加入错误处理。

25.3.6.2. 压缩的归档日志

如果归档存储的大小是个问题，你可以使用gzip 来压缩归档文件：

```
archive_command = 'gzip < %p > /mnt/server/archivedir/%f.gz'
```

那么在恢复时就需要使用 gunzip：

```
restore_command = 'gunzip < /mnt/server/archivedir/%f.gz > %p'
```

25.3.6.3. `archive_command` 脚本

很多人选择使用脚本来定义自己的 `archive_command`，这样`postgresql.conf` 中的配置项就会显得非常简单：

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

只要你希望在归档过程中使用不止一条命令，就建议使用单独的脚本文件。这样一来，所有复杂性都可以在脚本内部管理，而脚本可以使用诸如bash 或 perl 之类的常见脚本语言编写。

可以在脚本中处理的需求示例包括：

- 将数据复制到安全的异地数据存储
- 把 WAL 文件成批处理，使其每三个小时传输一次，而不是每次只传输一个
- 与其他备份和恢复软件对接
- 与监控软件对接以报告错误

> **提示**
>
> 使用 `archive_command` 脚本时，最好启用 logging_collector。脚本写到 stderr 的任何消息都会出现在数据库服务器日志中，这样一来，当复杂配置失败时就更容易诊断。

25.3.7. 注意事项

在撰写本文时，持续归档技术还存在若干局限。这些问题很可能会在未来版本中得到修复：

- 如果在进行基础备份时执行了 CREATE DATABASE 命令，而该`CREATE DATABASE` 所复制的模板数据库又在基础备份尚未结束时被修改，那么恢复时可能会把这些修改也传播到新建数据库中。这当然并不理想。为避免这种风险，最好在进行基础备份时不要修改任何模板数据库。

- CREATE TABLESPACE 命令会以字面绝对路径写入 WAL，因此重放时会按相同的绝对路径创建表空间。如果 WAL 在另一台机器上重放，这可能并不理想。即使 WAL 在同一台机器上、但重放到新的数据目录中，也可能有危险：重放仍会覆盖原表空间的内容。为避免此类潜在陷阱，最佳做法是在创建或删除表空间后重新执行一次基础备份。

还应注意，默认的WAL 格式相当臃肿，因为它包含许多磁盘页面快照。这些页面快照是为支持崩溃恢复而设计的，因为我们可能需要修复部分写入的磁盘页。根据你的系统硬件和软件情况，部分写入的风险可能小到可以忽略；在这种情况下，可以通过full_page_writes 参数关闭页面快照，从而显著减少已归档 WAL 文件的总量。（在这样做之前，请先阅读第 30 章中的说明和警告。）关闭页面快照并不妨碍把 WAL 用于 PITR 操作。未来一个可能的开发方向，是在`full_page_writes` 开启的情况下，通过去除不必要的页面副本来压缩归档 WAL 数据。在此之前，管理员可以考虑尽可能增大检查点间隔相关参数，以减少 WAL 中包含的页面快照数量。

第 26 章 高可用、负载均衡和复制

数据库服务器可以协同工作，从而在主库失效时让第二台服务器快速接管其任务（高可用性），或者让多台计算机提供同一份数据（负载均衡）。理想情况下，数据库服务器应当能够无缝协同工作。提供静态网页服务的 Web 服务器只需把 Web 请求分发到多台机器，就能很容易地组合起来。事实上，只读数据库服务器也相对容易组合起来。不幸的是，大多数数据库服务器同时处理读写请求，而读/写服务器要更难组合。这是因为，只读数据只需要在每台服务器上放置一次，而对任意一台服务器的写入都必须传播到所有服务器，以保证后续对这些服务器的读取请求返回一致的结果。

这种同步问题是服务器协同工作的根本难点。由于不存在一种能够对所有使用场景都消除同步问题影响的单一方案，因此出现了多种不同的解决方案。每一种方案都以不同方式处理这个问题，并且针对特定负载将其影响降到最低。

某些方案通过只允许一台服务器修改数据来处理同步。能够修改数据的服务器称为读/写、主库（master）或主库（primary）服务器。跟踪主库变更的服务器称为备库（standby）或备库（secondary）。只能在被提升为主库之后才能连接的备库称为温备，而能够接受连接并提供只读查询的备库称为热备。

某些方案是同步的，即一个修改数据的事务只有在所有服务器都提交该事务之后才被视为已提交。这保证一次故障切换不会丢失任何数据，并且所有负载均衡的服务器无论查询哪一台都将返回一致的结果。相反，异步方案允许一次提交与其传播到其他服务器之间存在一定延迟，这就带来了切换到备库时丢失某些事务的可能性，也意味着负载均衡的服务器可能返回略微陈旧的结果。当同步通信过慢时，就会使用异步通信。

这些方案还可以按粒度分类。有些方案只能处理整个数据库服务器，而另一些则允许在每个表或者每个数据库级别进行控制。

无论做出哪种选择，都必须考虑性能。功能与性能之间通常存在权衡。例如，在低速网络上采用完全同步的方案，性能可能下降一半以上，而异步方案对性能的影响则可能很小。

本节其余部分将概述多种故障切换、复制和负载均衡方案。

26.1. 不同方案的比较

共享磁盘故障切换

共享磁盘故障切换通过只保留一份数据库拷贝来避免同步开销。它使用一个由多个服务器共享的单一磁盘阵列。如果主库失效，备库就可以挂载并启动数据库，就好像它正在从一次数据库崩溃中恢复一样。这允许快速故障切换而不会丢失数据。

共享硬件功能常见于网络存储设备。使用网络文件系统也是可行的，但必须注意文件系统是否具备完整的 POSIX 行为（见第 18.2.2 节）。这种方法的一大局限是，如果共享磁盘阵列失效或损坏，主库和备库都会无法工作。另一个问题是，在主库运行期间，备库绝不应访问共享存储。

文件系统（块设备）复制

共享硬件功能的一种变体是文件系统复制，其中对一个文件系统的所有更改都会镜像到位于另一台计算机上的文件系统。唯一的限制是，这种镜像必须以能够保证备库拥有该文件系统一致拷贝的方式完成 – 特别是，对备库的写入必须与主库上的写入保持相同顺序。DRBD 是 Linux 上一种流行的文件系统复制方案。

预写式日志传送

温备和热备服务器能够通过读取预写式日志（WAL）记录流来保持最新状态。如果主库失效，备库拥有主库的几乎全部数据，并且能够迅速成为新的主库。这可以是同步的，也可以是异步的，并且只能用于整个数据库服务器。

可以使用基于文件的日志传送（第 26.2 节）、流复制（见第 26.2.5 节）或两者结合的方式来实现备库。有关热备的信息，见第 26.5 节。

逻辑复制

逻辑复制允许数据库服务器向另一台服务器发送数据修改流。PostgreSQL 逻辑复制从 WAL 中构造逻辑数据修改流。逻辑复制允许复制单个表的数据更改，无需将某台特定服务器指定为主库或副本，而是允许数据沿多个方向流动。有关逻辑复制的更多信息，参见第 31 章。第三方扩展也可以通过逻辑解码接口（第 49 章）提供类似功能。

基于触发器的主备复制

主备复制配置会将所有修改数据的查询发送到主库。主库将数据更改异步发送到备库。主库运行期间，备库可以响应只读查询。备库很适合处理数据仓库查询。

Slony-I 是这种复制类型的一个示例。它按表粒度工作，并且支持多个备库。由于它会以批处理方式异步更新备库，因此在故障切换期间可能会发生数据丢失。

基于语句的复制中间件

使用基于语句的复制中间件时，一个程序会截获每条 SQL 查询，并将其发送到一台或所有服务器。每台服务器独立运行。读写查询必须发送到所有服务器，确保每台服务器都收到所有更改。但只读查询可以只发送到一台服务器，从而将读取负载分散到这些服务器上。

如果只是原样广播查询，`random()`、`CURRENT_TIMESTAMP` 等函数以及序列在不同服务器上可能产生不同的值。这是因为每台服务器独立运行，且广播的是 SQL 查询，而不是实际修改后的行。如果无法接受这种情况，中间件或应用就必须从同一台服务器查询这些值，然后在写入查询中使用这些值。另一种选择是将这种复制方式与传统主备配置结合使用，即只向主库发送修改数据的查询，再通过主备复制将其传播到备库，而不是通过复制中间件传播。还必须确保所有事务在所有服务器上要么全部提交，要么全部中止，为此可以使用两阶段提交（PREPARE TRANSACTION 和 COMMIT PREPARED）。Pgpool-II 和 Continuent Tungsten 就是这类复制的例子。

异步多主复制

对于那些并不经常连接或通信链路较慢的服务器，例如笔记本电脑或远程服务器，保持服务器间的数据一致是一个挑战。使用异步多主复制时，每台服务器都独立工作，并定期与其他服务器通信，以识别冲突事务。这些冲突可以由用户或冲突解决规则来解决。Bucardo 是这种复制类型的一个示例。

同步多主复制

在同步多主复制中，每台服务器都能接受写请求，并且在每个事务提交之前，修改过的数据都会从原始服务器传送给其他每台服务器。繁重的写入活动可能导致过多锁定和提交延迟，进而带来较差的性能。读请求可以发送给任意服务器。某些实现使用共享磁盘来减少通信开销。同步多主复制尤其适合以读为主的负载，尽管它的一大优势是任意服务器都能接受写请求 – 无需在主库和备库之间划分负载，并且由于数据更改是从一台服务器传送到另一台服务器，因此不会出现非确定函数（如 `random()`）的问题。

PostgreSQL 本身不提供这种复制，不过可以在应用代码或中间件中利用 PostgreSQL 的两阶段提交（PREPARE TRANSACTION 和 COMMIT PREPARED）来实现这一功能。

商业解决方案

由于 PostgreSQL 是开源的，而且易于扩展，许多公司以 PostgreSQL 为基础，创建了具有独特故障切换、复制和负载均衡功能的商业闭源解决方案。

表 26.1 总结了上述多种方案的能力。

表 26.1. 高可用、负载均衡和复制特性矩阵

特性	共享磁盘故障切换	文件系统复制	预写式日志传送	逻辑复制	基于触发器的主备复制	基于语句的复制中间件	异步多主复制	同步多主复制
最常见的实现	NAS	DRBD	内置流复制	内置逻辑复制、pglogical	Londiste、Slony	pgpool-II	Bucardo	
通信方法	共享磁盘	磁盘块	WAL	逻辑解码	表行	SQL	表行	表行和行锁
不要求特殊硬件		•	•	•	•	•	•	•
允许多个主库				•		•	•	•
主库无额外开销	•		•	•		•		
不等待多个服务器	•		关闭同步时	关闭同步时	•		•	
主库失效时绝不丢失数据	•	•	启用同步时	启用同步时		•		•
副本接受只读查询			启用热备时	•	•	•	•	•
每个表粒度				•	•		•	•
不需要冲突解决	•	•	•		•	•		•

有一些方案不适合上述的类别：

数据分区

数据分区会把表拆分成多个数据集。每个数据集只能由一台服务器修改。例如，可以按办公室划分数据，如伦敦和巴黎，每个办公室各有一台服务器。如果有必要执行组合伦敦和巴黎数据的查询，应用可以同时查询两台服务器，或者可以使用主库/备库复制，在每台服务器上保留其他办公室数据的只读副本。

多服务器并行查询执行

上述许多方案都允许多台服务器处理多个查询，但没有一种允许单个查询同时使用多台服务器来更快完成。这种方案允许多台服务器在同一个查询上并发工作。通常的做法是把数据分散到各台服务器上，让每台服务器执行该查询中属于自己的部分，然后把结果返回给一台中心服务器，由它汇总结果并返回给用户。这种方案也可以使用PL/Proxy 工具集来实现。

26.2. 日志传送备库

持续归档可用于创建一种高可用性（HA）集簇配置，其中有一个或多个备库随时准备在主库失效时接管操作。这种能力通常称为温备或日志传送。

主库和备库协同工作以提供这种能力，不过两者之间只是松耦合。主库运行在持续归档模式下，而每台备库都运行在持续恢复模式下，从主库读取 WAL 文件。启用这种能力不需要修改数据库表，因此与其他一些复制方案相比，它的管理开销较低。这种配置对主库的性能影响也相对较小。

将 WAL 记录直接从一台数据库服务器移动到另一台数据库服务器，通常称为日志传送。PostgreSQL 通过一次传送一个文件（WAL 段）中的 WAL 记录来实现基于文件的日志传送。WAL 文件（16MB）可以轻松且低成本地传送到任意距离，无论是邻近系统、同一站点的另一台系统，还是地球另一端的系统。该技术所需的带宽取决于主库的事务速率。基于记录的日志传送粒度更细，会通过网络连接增量地流式传输 WAL 变更（见第 26.2.5 节）。

需要注意的是，日志传送是异步的，也就是说 WAL 记录是在事务提交之后才被传送的。因此，如果主库发生灾难性故障，就会存在一个数据丢失窗口；尚未传送的事务将会丢失。对于基于文件的日志传送，可以通过使用 `archive_timeout` 参数来限制这个数据丢失窗口，它可以设置为低至数秒。不过，如此低的设置会显著增加文件传送所需的带宽。流复制（见第 26.2.5 节）允许把数据丢失窗口缩得更小。

恢复性能已经足够好，因此一旦备库被激活，通常只需片刻就能达到完全可用状态。因此，这种配置被称为温备配置，它提供了高可用性。从归档的基础备份中恢复服务器并向前重放则需要更长时间，因此这种技术只能用于灾难恢复，而不是高可用性。备库还可以用于只读查询，这种情况下它被称为热备服务器。更多信息见第 26.5 节。

26.2.1. 规划

通常，最好让主库和备库尽可能相似，至少从数据库服务器的角度看应如此。尤其是，与表空间相关的路径名会原样传递，因此如果使用该特性，主库和备库必须为表空间配置完全相同的挂载路径。请记住，如果在主库上执行 CREATE TABLESPACE，则它所需的任何新挂载点都必须在执行该命令之前先在主库和所有备库上创建好。硬件不必完全相同，但经验表明，在应用和系统的整个生命周期内，维护两个相同的系统要比维护两个不同的系统更容易。无论如何，硬件架构必须相同 – 例如，从 32 位系统向 64 位系统传送日志是不可行的。

一般来说，不能在运行不同主版本 PostgreSQL 的服务器之间传送日志。PostgreSQL 全球开发组的策略是在次版本升级期间不改变磁盘格式，因此主库和备库运行不同次版本通常也能正常工作。不过，这方面并没有正式支持，因此仍建议主库和备库尽量保持在相同的发行级别。当升级到新的次版本时，最安全的策略是先升级备库 – 新的次版本更有可能兼容读取前一个次版本生成的 WAL 文件，反过来则未必。

26.2.2. 备库操作

在备库模式下，服务器会持续应用从主库收到的 WAL。备库可以从 WAL 归档读取 WAL（参见 restore_command），也可以通过 TCP 连接直接从主库读取（流复制）。备库还会尝试恢复其集簇 `pg_wal` 目录中找到的所有 WAL。这通常发生在服务器重启后，此时备库会重新重放重启前从主库接收的 WAL；不过，也可以随时手动将文件复制到 `pg_wal` 中，让备库重放它们。

启动时，备库首先调用 `restore_command`，恢复归档位置中所有可用的 WAL。当读到归档中可用 WAL 的末尾，且 `restore_command` 失败后，它会尝试恢复 `pg_wal` 目录中的所有可用 WAL。如果这也失败，且已配置流复制，备库就会尝试连接主库，从归档或 `pg_wal` 中找到的最后一条有效记录开始接收 WAL 流。如果连接失败、未配置流复制，或连接后来断开，备库就会返

回第一步，再次尝试从归档恢复文件。这种依次从归档、pg_wal 和流复制重试的循环，会一直持续到服务器停止，或触发文件触发故障切换。

运行 pg_ctl promote 或发现触发文件（trigger_file）时，服务器会退出备库模式，切换到正常运行。在故障切换前，会恢复归档或 pg_wal 中能立即获取的所有 WAL，但不会尝试连接主库。

26.2.3. 为备库准备主库

如第 25.3 节所述，在主库上设置持续归档，将日志归档到备库可访问的归档目录。即使主库宕机，该归档位置也应该对备库可访问；也就是说，它应位于备库本身或另一台可信服务器上，而不是位于主库上。

如果想使用流复制，就需要在主库上配置认证，以允许来自备库的复制连接；也就是说，创建一个角色，并在 pg_hba.conf 中添加一个或多个把数据库字段设置为 replication 的合适项。还要确保主库配置文件中的 max_wal_senders 被设置为足够大的值。如果要使用复制槽，还应确保 max_replication_slots 也设置得足够高。

如第 25.3.2 节所述，获取一个基础备份来引导备库。

26.2.4. 设置备库

要配置备库，请恢复从主库取得的基础备份（参见第 25.3.4 节）。在备库的集簇数据目录中创建恢复命令文件 recovery.conf，并启用 standby_mode。将 restore_command 设为一个从 WAL 归档复制文件的简单命令。如果计划使用多个备库提供高可用，请将 recovery_target_timeline 设为 latest，使备库能够跟随故障切换到另一个备库时发生的时间线变化。

> **注意**
>
> 不要将 pg_standby 或类似工具与这里介绍的内置备库模式一起使用。如果文件不存在，restore_command 应立即返回；服务器会在必要时重试该命令。有关 pg_standby 等工具的使用，参见第 26.4 节。

如果希望使用流复制，请在 primary_conninfo 中填写 libpq 连接字符串，包含连接主库所需的主机名（或 IP 地址）及其他信息。如果主库使用密码认证，还需要在 primary_conninfo 中指定密码。

如果你是为了高可用目的设置备库，那么也应像主库一样设置 WAL 归档、连接和认证，因为故障切换后该备库将作为主库工作。

如果使用 WAL 归档，可以借助archive_cleanup_command 参数删除备库不再需要的文件，以尽量缩小归档大小。pg_archivecleanup 工具专门设计用于在典型的单备库配置中与 archive_cleanup_command 配合使用，见 pg_archivecleanup。不过请注意，如果你还把归档用于备份目的，即使这些文件对备库已经不再需要，也必须至少保留从最新基础备份恢复所需的那些文件。

一个简单的 recovery.conf 示例如下：

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

备库的数量可以任意多，但如果使用流复制，请确保主库上的 max_wal_senders 设置得足够高，以允许它们同时连接。

26.2.5. 流复制

与基于文件的日志传送相比，流复制可以让备库保持得更接近最新状态。备库连接到主库，主库在生成 WAL 记录时就会把它们流式发送给备库，而不必等待 WAL 文件被写满。

默认情况下流复制是异步的（见第 26.2.8 节），在这种情况下主库上提交一个事务与该变化在备库上变得可见之间存在短暂的延迟。不过这种延迟比基于文件的日志传送方式中要小得多，在备库的能力足以跟得上负载的前提下，延迟通常低于一秒。在流复制中，不需要 archive_timeout 来缩减数据丢失窗口。

如果你使用流复制，但没有启用基于文件的持续归档，服务器可能会在备库收到旧的 WAL 段之前就把它们回收掉。如果发生这种情况，备库就需要重新通过新的基础备份进行初始化。可以通过把 wal_keep_segments 设置得足够大，以确保 WAL 段不会过早被回收，或者为备库配置一个复制槽，从而避免这种情况。如果配置了一个备库可访问的 WAL 归档，就不需要这些方案，因为只要归档保留了足够多的段，备库始终可以利用归档追赶上来。

要使用流复制，请先按第 26.2 节所述配置基于文件的日志传送备库。将这种备库转为流复制备库的关键步骤，是在 recovery.conf 文件中设置 primary_conninfo，使其指向主库。在主库上设置listen_addresses 和认证选项（参见 pg_hba.conf），使备库能够连接主库的 replication 伪数据库（参见第 26.2.5.1 节）。

在支持 keepalive 套接字选项的系统上，设置 tcp_keepalives_idle、tcp_keepalives_interval 和 tcp_keepalives_count 有助于主库迅速注意到断开的连接。

设置来自备库的最大并发连接数（详见 max_wal_senders）。

备库启动后，如果正确设置了 primary_conninfo，备库会在重放归档中所有可用的 WAL 文件后连接主库。如果连接成功，就会在备库上看到 walreceiver 进程，在主库上看到对应的 walsender 进程。

26.2.5.1. 认证

务必正确设置复制访问权限，使只有受信任的用户才能读取 WAL 流，因为很容易从中提取需要权限才能访问的信息。备库必须以超级用户或具有 REPLICATION 权限的账户向主库认证。建议为复制创建专用用户账户，并授予 REPLICATION 和 LOGIN 权限。虽然 REPLICATION 权限赋予的权限很高，但它不允许用户修改主库系统上的任何数据，而 SUPERUSER 权限允许这样做。

复制的客户端认证由 pg_hba.conf 中的一条记录控制，该记录需要把 replication 指定在 *database* 字段中。例如，如果备库运行在主机 IP 192.168.1.100 上，并且用于复制的账户名为 foo，管理员可以在主库上的 pg_hba.conf 文件中加入下列行：

```
# Allow the user "foo" from host 192.168.1.100 to connect to the primary
# as a replication standby if the user's password is correctly supplied.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    replication     foo             192.168.1.100/32        md5
```

主库的主机名、端口号、连接用户名和密码在 recovery.conf 文件中指定。也可以将密码设置在备库的 ~/.pgpass 文件中（将 replication 指定为 *database* 字段的值）。例如，如果主库所在主机的 IP 地址为 192.168.1.50，端口为 5432，用于复制的账户名为 foo，密码为 foopass，管理员可以将以下行添加到 recovery.conf 文件（位于备库）中：

```
# The standby connects to the primary that is running on host 192.168.1.50
# and port 5432 as the user "foo" whose password is "foopass".
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

26.2.5.2. 监控

流复制的一个重要健康指标，是主库上已经生成但尚未在备库上应用的 WAL 记录量。你可以通过比较主库上的当前 WAL 写入位置和备库收到的最后一个 WAL 位置来计算这种滞后。这些位置分别可以用主库上的 pg_current_wal_lsn 和备库上的 pg_last_wal_receive_lsn 取得（详见表 9.79 和表 9.80）。备库上的最后一个 WAL 接收位置也会显示在 WAL 接收进程的进程状态中，即通过 ps 命令显示的状态（详见第 28.1 节）。

可以通过 pg_stat_replication 视图获取 WAL 发送进程列表。pg_current_wal_lsn 与该视图的 sent_lsn 字段相差较大，可能表示主库负载很高；而 sent_lsn 与备库上 pg_last_wal_receive_lsn 之间存在差异，则可能表示网络延迟或备库负载很高。

在热备上，可以通过pg_stat_wal_receiver 视图获取 WAL 接收进程的状态。如果 pg_last_wal_replay_lsn 与该视图的 received_lsn 相差较大，说明接收 WAL 的速度超过了重放速度。

26.2.6. 复制槽

复制槽提供了一种自动化方法，以确保主库在所有备库都收到 WAL 段之前不会删除它们，并且即使备库处于断开状态，主库也不会删除那些一旦删除就可能导致恢复冲突的行。

除了使用复制槽，也可以通过wal_keep_segments 防止旧 WAL 段被删除，或者使用 archive_command 将这些段保存在归档中。不过，这些方法通常会保留多于实际所需的 WAL 段，而复制槽只保留已知必需的段数。这些方法的优点是能限制 pg_wal 的空间需求；目前使用复制槽还无法做到这一点。

类似地，hot_standby_feedback 和 vacuum_defer_cleanup_age 可以保护相关行，避免它们被清理删除，但前者在备库未连接期间无法提供保护，后者则通常需要设为较大的值才能提供充分保护。复制槽克服了这些缺点。

26.2.6.1. 查询和管理复制槽

每个复制槽都有一个名称，该名称可以包含小写字母、数字和下划线字符。

现有复制槽及其状态可以在 pg_replication_slots 视图中查看。

复制槽可以通过流复制协议（见第 53.4 节）或者 SQL 函数（见第 9.26.6 节）创建和删除。

26.2.6.2. 配置示例

可以这样创建复制槽：

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('node_a_slot');
  slot_name  | lsn
-------------+-----
 node_a_slot |

postgres=# SELECT slot_name, slot_type, active FROM pg_replication_slots;
  slot_name  | slot_type | active
-------------+-----------+--------
 node_a_slot | physical  | f
(1 row)
```

要配置备库使用此复制槽，需要将 `primary_slot_name` 配置在备库的 `recovery.conf` 中。下面是一个简单的示例：

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
primary_slot_name = 'node_a_slot'
```

26.2.7. 级联复制

级联复制特性允许一台备库接受复制连接，并像中继器一样把 WAL 记录流式发送给其他备库。这可以用来减少直接连接到主库的连接数，并使站点间的带宽开销最小化。

一台同时扮演接收者和发送者角色的备库称为级联备库。与主库连接更直接（经过更少级联备库）的备库称为上游服务器，而距离更远的备库称为下游服务器。级联复制并不限制下游服务器的数量和拓扑，不过每台备库只连接到一台上游服务器，而这条链路最终都会通向同一台主库。

级联备库不仅发送从主库接收到的 WAL 记录，也会发送那些从归档中恢复的记录。因此，即使某条上游复制连接被中断，只要仍有新的 WAL 记录可用，下游的流复制就会继续。

级联复制目前是异步的。同步复制（见第 26.2.8 节）设置当前对级联复制无影响。

热备反馈会向上传播，无论级联拓扑如何。

如果某个上游备库被提升为新主库，只要 `recovery_target_timeline` 设为 `'latest'`，下游服务器就会继续从新主库接收流。

要使用级联复制，需要把级联备库设置为能够接受复制连接（也就是设置max_wal_senders 和 hot_standby，并配置基于主机的认证）。你还需要把下游备库中的 `primary_conninfo` 设置为指向级联备库。

26.2.8. 同步复制

PostgreSQL 的流复制默认是异步的。如果主库崩溃，则某些已提交的事务可能尚未复制到备库，从而导致数据丢失。数据丢失量与故障切换时的复制延迟成正比。

同步复制能够确认一个事务所做的全部修改已经被传送到一台或多台同步备库。这扩展了事务提交所提供的标准持久性级别。在计算机科学理论中，这种保护级别被称为 2-safe 复制；而当 `synchronous_commit` 被设置为 `remote_write` 时，则称为 group-1-safe（group-safe 和 1-safe）。

请求同步复制时，每个写事务的提交都会等待，直到收到确认，表明该提交已被写入主库和备库磁盘上的预写式日志。数据唯一可能丢失的情况，是主库和备库同时崩溃。这可以提供更高的持久性级别，不过前提是系统管理员必须谨慎地部署和管理这两台服务器。等待确认会增强用户对服务器崩溃时更改不会丢失的信心，但也必然会增加请求事务的响应时间。最短等待时间是主库与备库之间的往返时间。

只读事务和事务回滚不需要等待备库的回应。子事务提交也不需要等待备库响应，只有顶层提交才需要等待。数据装载或索引构建等长时间运行的动作，直到最终提交时才会等待。所有两阶段提交操作都需要等待提交，包括准备和提交两个阶段。

同步备库可以是物理复制备库，也可以是逻辑复制订阅者。它还可以是任何其他物理或逻辑 WAL 复制流的消费者，只要它知道如何发送适当的反馈消息。除了内置的物理和逻辑复制系统之外，还包括pg_receivewal 和 pg_recvlogical 之类的专用程序，以及一些第三方复制系统和定制程序。关于同步复制支持的细节，请查看相应文档。

26.2.8.1. 基本配置

一旦流复制已经配置好，配置同步复制只需要额外一步：必须把synchronous_standby_names 设置为非空值。synchronous_commit 也必须设置为 on，但由于这是默认值，通常无需更改（见第 19.5.1 节和第 19.6.2 节）。这样的配置会导致每次提交都等待确认，以保证备库已经把提交记录写入持久存储。synchronous_commit 可以由单个用户设置，因此既可以在配置文件中配置，也可以针对特定用户或数据库配置，或者由应用动态配置，从而在每事务级别控制持久性保证。

当提交记录已经在主库上写入磁盘之后，WAL 记录就会被发送到备库。每当新的一批 WAL 数据被写入磁盘时，备库就会发送回复消息，除非备库上的wal_receiver_status_interval 被设置为零。如果 synchronous_commit 被设置为 remote_apply，那么备库会在提交记录被重放、该事务变得可见时发送回复消息。如果根据主库上的synchronous_standby_names 设置，该备库被选为同步备库，那么它发出的回复会与其他同步备库的回复一起，用于决定何时释放那些正在等待确认提交记录已被收到的事务。这些参数允许管理员指定哪些备库应作为同步备库。注意，同步复制的配置主要在主库上进行。被命名的备库必须直接连接到主库；主库并不知道使用级联复制的下游备库。

把 synchronous_commit 设置为 remote_write，会使每次提交都等待，直到备库确认已经收到提交记录并把它写入自己的操作系统，但不会等待数据被刷到备库磁盘上。与on 相比，这种设置提供较弱一些的持久性保证：在操作系统崩溃时，备库可能丢失数据，尽管在PostgreSQL 崩溃时不会。不过在实践中，这是一种有用的设置，因为它可以降低事务响应时间。只有当主库和备库都崩溃，并且主库数据库同时发生损坏时，才可能发生数据丢失。

把 synchronous_commit 设置为 remote_apply，会使每次提交都等待，直到当前同步备库报告它们已经重放了该事务，从而使它对用户查询可见。在简单场景下，这可以支持具备因果一致性的负载均衡。

如果请求快速关闭，用户将停止等待。不过，与使用异步复制时一样，在所有尚未传送的 WAL 记录传输到当前已连接的备库之前，服务器不会完全关闭。

26.2.8.2. 多个同步备库

同步复制支持一台或多台同步备库；事务将一直等待，直到所有被视为同步的备库确认已收到其数据。事务需要等待多少台同步备库的回复，由synchronous_standby_names 指定。该参数还指定了一个备库名称列表，以及从中选择同步备库的方法（FIRST 和 ANY）。

FIRST 方法指定基于优先级的同步复制，使事务提交等待，直到它们的 WAL 记录被复制到按优先级选出的、所要求数量的同步备库上。列表中出现得越早的备库优先级越高，并会被视为同步备库。该列表中更靠后的备库则是潜在的同步备库。如果当前某个同步备库因任何原因断开连接，它将立刻由下一个优先级最高的备库替代。

基于优先级的多同步备库的 synchronous_standby_names 示例如下：

```
synchronous_standby_names = 'FIRST 2 (s1, s2, s3)'
```

在这个例子中，如果四台备库s1、s2、s3 和 s4 都在运行，则s1 和 s2 会被选为同步备库，因为它们的名字在备库名称列表中出现得更早。s3 是潜在的同步备库，当s1 或 s2 中的任意一台失效时，它就会接替其角色。由于s4 的名称不在列表中，因此它是异步备库。

ANY 方法指定基于法定人数的同步复制，使事务提交等待，直到它们的 WAL 记录被复制到列表中至少所要求数量的同步备库。

基于法定人数的多同步备库的 synchronous_standby_names 示例如下：

```
synchronous_standby_names = 'ANY 2 (s1, s2, s3)'
```

在这个例子中，如果四台备库s1、s2、s3 和 s4 都在运行，事务提交将等待来自s1、s2 和 s3 中的至少任意两台备库的回复。由于 s4 的名字不在列表中，因此它是异步备库。

备库的同步状态可以通过 pg_stat_replication 视图查看。

26.2.8.3. 性能规划

同步复制通常要求对备库进行仔细规划和部署，才能保证应用具有可接受的性能。等待本身不会占用系统资源，但事务锁会一直保持到传输得到确认。因此，若不谨慎使用同步复制，数据库应用的性能会因为响应时间增加和争用加剧而下降。

PostgreSQL 允许应用开发者通过复制来指定所需的持久性级别。这可以在整个系统级别指定，也可以针对特定用户、特定连接，甚至单个事务指定。

例如，一个应用负载可能由如下部分组成：10% 的变更是重要的客户资料，而 90% 的变更是不太重要、即使丢失业务也较容易承受的数据，例如用户之间的聊天消息。

通过在应用级别（在主库上）指定同步复制选项，我们可以只对最重要的变更提供同步复制，而不会拖慢大部分工作负载。应用级别选项是让高性能应用获得同步复制收益的一种重要而实用的工具。

你应当确保网络带宽高于 WAL 数据的生成速率。

26.2.8.4. 高可用规划

synchronous_standby_names 指定了当 synchronous_commit 设置为 on、remote_apply 或 remote_write 时，事务提交需要等待其响应的同步备库的数量和名称。如果任意一台同步备库崩溃，此类事务提交就可能永远无法完成。

对于高可用而言，最好的办法是确保你始终保有所要求数量的同步备库。这可以通过在 synchronous_standby_names 中命名多个潜在同步备库来实现。

在基于优先级的同步复制中，列表中出现较早的备库将被用作同步备库。列在它们之后的备库，会在当前同步备库失效时接替其角色。

在基于法定人数的同步复制中，列表中的所有备库都会被用作同步备库候选。即使其中一台失效，其他备库仍会继续承担同步备库候选的角色。

当一台备库第一次连接到主库时，它还没有正确同步。这种状态称为catchup 模式。一旦备库与主库之间的滞后第一次变为零，它就会进入实时的streaming 状态。备库刚创建之后，追赶阶

段可能会持续较长时间。如果备库被关闭，则追赶阶段会随着它停机时间的延长而变长。只有在到达streaming 状态后，备库才能成为同步备库。这种状态可以通过pg_stat_replication 视图查看。

如果主库在提交正等待确认时重启，这些等待中的事务会在主库恢复后被标记为已完全提交。无法确定在主库崩溃时，所有备库是否已经收到全部待传送的 WAL 数据。因此，某些事务可能不会在备库上显示为已提交，即使它们在主库上显示为已提交。我们所提供的保证是：只有在确认 WAL 数据已经被所有同步备库安全接收之后，应用才会收到事务成功提交的显式确认。

如果你确实无法维持所要求数量的同步备库，那么就应当在synchronous_standby_names 中减少事务提交需要等待其响应的同步备库数量（或者禁用它），然后在主库上重新加载配置文件。

如果主库与剩余的备库隔离开了，你应当故障切换到那些剩余备库中最佳的候选者。

如果需要在事务等待期间重新创建备库，请确保在 synchronous_commit = off 的会话中运行 pg_start_backup() 和 pg_stop_backup() 命令，否则这些请求会一直等待备库出现。

26.2.9. 在备库中持续归档

当在备库中使用持续 WAL 归档时，有两种不同的场景：WAL 归档可以由主库和备库共享，或者备库可以拥有自己的 WAL 归档。当备库拥有自己的 WAL 归档时，应把archive_mode 设置为 always，这样备库就会为它接收到的每个 WAL 段调用归档命令，无论该 WAL 段是通过从归档恢复得到的，还是通过流复制得到的。共享归档也可以类似处理，但archive_command 必须检查正在归档的文件是否已经存在，以及已有文件的内容是否完全相同。这就要求在 archive_command 中更加小心：既不能用不同内容覆盖现有文件，又要在同一个文件被归档两次且内容完全相同时返回成功。如果两台服务器同时尝试归档同一个文件，还必须确保整个过程不存在竞争条件。

如果 archive_mode 被设置为 on，那么归档器在恢复期间或备库模式下不会启用。如果备库被提升，它会在提升后开始归档，但不会归档任何不是由它自己生成的 WAL 或时间线历史文件。要在归档中获得完整的一系列 WAL 文件，就必须确保所有 WAL 在到达备库之前已经被归档。对于基于文件的日志传送，这天然成立，因为备库只能恢复归档中找到的文件；但在启用流复制时则不是这样。当服务器不处于恢复模式时，on 和 always 模式之间没有区别。

26.3. 故障切换

如果主库失效，备库就应该开始执行故障切换过程。

如果备库失效，则不需要发生故障切换。如果备库能够重新启动，即使是在稍后某个时间点，恢复过程也可以立即重新开始，从而利用可重启恢复的优势。如果备库无法重新启动，则应创建一个全新的备库实例。

如果主库失效，而备库成为新的主库，那么旧主库之后如果重新启动，你必须有一种机制通知它，它已经不再是主库。这有时被称为STONITH（Shoot The Other Node In The Head），它对于避免两个系统都认为自己是主库的情况至关重要，因为那种情况会导致混乱，并最终造成数据丢失。

许多故障切换系统只使用两个系统，即主库和备库，并通过某种心跳机制连接它们，以持续验证两者之间的连通性以及主库的可用性。也可以使用第三个系统（称为见证服务器）来防止某些不恰当的故障切换，但除非设置得足够谨慎并经过严格测试，否则额外增加的复杂性可能并不值得。

PostgreSQL 并不提供用于识别主库故障并通知备库的系统软件。现在已经存在许多这样的工具，并且它们通常能很好地与成功故障切换所需的操作系统设施整合在一起，例如 IP 地址迁移。

一旦故障切换到备库，系统中就只剩下一台服务器在运行。这被称为退化状态。原来的备库现在成为主库，而原来的主库已经停机，并且可能持续停机。要恢复到正常运行状态，就必须重新创建一台备库：要么在原主库恢复后在其上重建，要么在第三台可能是全新的服务器上重建。在大型集簇上，可以使用pg_rewind 工具来加快这一过程。一旦完成，就可以认为主库和备库已经交换了角色。有些人会选择使用第三台服务器，在新的备库重建完成之前为新的主库提供后备支持，但显然这会让系统配置和操作流程更加复杂。

因此，从主库切换到备库可以很快，但重新准备故障切换集簇仍然需要时间。定期在主库与备库之间进行切换是有益的，因为它允许每个系统定期停机维护。这也相当于对故障切换机制进行测试，以确保真正需要它时它能够正常工作。建议编写书面的管理操作规程。

要触发日志传送备库的故障切换，可以运行 `pg_ctl promote`，或按照 `recovery.conf` 中 `trigger_file` 设置指定的文件名和路径创建触发文件。如果计划使用 `pg_ctl promote` 进行故障切换，就不需要设置 `trigger_file`。如果所配置的报表服务器只是用来分担主库的只读查询，而不用于高可用，则不需要提升它。

26.4. 日志传送的另一种方法

除了前几节介绍的内置备库模式，也可以使用一个轮询归档位置的 `restore_command`。在 8.4 及更早版本中，这是唯一可用的方法。采用这种配置时，应关闭 `standby_mode`，因为备库运行所需的轮询由你自己实现。其参考实现参见pg_standby 模块。

注意，在这种模式下，服务器每次应用一整个 WAL 文件。因此，如果使用备库处理查询（参见热备），主库上的操作与备库上能看到该操作的结果之间，会有一段延迟，其长度相当于写满一个 WAL 文件所需的时间。可以用 `archive_timeout` 缩短这一延迟。还要注意，这种方法不能与流复制结合使用。

主库和备库上执行的操作都是普通的持续归档和恢复任务。两台数据库服务器之间唯一的联系，就是它们共享的 WAL 文件归档：主库写入归档，备库从归档读取。必须确保不同主库的 WAL 归档不会混在一起或弄错。如果归档仅用于备库运行，则无需保留很大的归档。

让这两台松散耦合的服务器协同工作的关键，只是备库上的 `restore_command`：当请求下一个 WAL 文件时，它会等待主库提供该文件。`restore_command` 在备库的 `recovery.conf` 文件中指定。正常恢复处理会从 WAL 归档请求文件，如果文件不可用，就报告失败。对备库处理而言，下一个 WAL 文件尚不可用是正常情况，因此备库必须等待它出现。对于以 `.history` 结尾的文件，则无需等待，必须返回非零返回码。可以编写一个自定义脚本，循环检查下一个 WAL 文件是否存在，从而实现会等待的 `restore_command`。还必须提供触发故障切换的方法，用来中断 `restore_command`、跳出循环，并向备库返回文件未找到错误。这会结束恢复，随后备库就会作为普通服务器启动。

一个合适的 `restore_command` 的伪代码如下：

```
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L);         /* wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
        CopyWALFileForRecovery();
```

pg_standby 模块提供了会等待的 `restore_command` 的可用示例。应参考它来正确实现上述逻辑，也可以按需扩展，以支持特定配置和环境。

触发故障切换的方法是规划和设计的重要部分。一个可能的选择是 `restore_command` 命令。每个 WAL 文件都会执行一次该命令，但运行 `restore_command` 的进程针对每个文件单独创建和结束，因此不存在守护进程或服务器进程，也无法使用信号或信号处理程序。所以，`restore_command` 不适合触发故障切换。可以使用简单的超时机制，尤其是在已知主库 `archive_timeout` 设置的情况下配合使用。不过，这容易出错，因为网络问题或主库繁忙就可能导致故障切换。如果能够安排，显式创建触发文件之类的通知机制是理想的方式。

26.4.1. 实现

使用这种替代方法配置备库的简要流程如下。各步骤的完整细节，参见所注明的前面章节。

1. 尽可能将主库和备库系统配置得相同，包括安装两个完全相同、发行版本一致的 PostgreSQL 副本。

2. 配置持续归档，将主库的 WAL 归档到备库上的一个目录。确保在主库上正确设置 archive_mode、archive_command 和 archive_timeout（参见第 25.3.1 节）。

3. 制作主库的基础备份（参见第 25.3.2 节），并将这些数据装载到备库上。

4. 在备库上从本地 WAL 归档开始恢复，在 `recovery.conf` 中指定前面所述的会等待的 `restore_command`（参见第 25.3.4 节）。

恢复过程将 WAL 归档视为只读，因此，WAL 文件一旦复制到备库系统，就可以在备库读取它的同时，将其复制到磁带上。这样，既能运行备库提供高可用，又能保存文件用于更长期的灾难恢复。

为了测试，可以在同一个系统上同时运行主库和备库。这不会对服务器的稳健性带来有意义的提升，也不能称为高可用。

26.4.2. 基于记录的日志传送

也可以使用这种替代方法实现基于记录的日志传送，但需要自行开发，而且只有整个 WAL 文件传送完成后，更改才会对热备上的查询可见。

外部程序可以调用 `pg_walfile_name_offset()` 函数（参见第 9.26 节），取得当前 WAL 末尾所在的文件名及其在文件中的精确字节偏移。随后就可以直接访问 WAL 文件，将上次已知的 WAL 末尾到当前末尾之间的数据复制到备库。采用这种方法，可能丢失数据的时间窗口就是复制程序的轮询周期，可以很短，而且不会因为强制归档未写满的段文件而浪费带宽。注意，备库的 `restore_command` 脚本只能处理完整的 WAL 文件，因此这些增量复制的数据通常不会提供给备库使用。它们只有在主库失效时才有用 – 此时，在允许备库启动前，将最后一个不完整的 WAL 文件交给备库。要正确实现这一过程，`restore_command` 脚本必须与数据复制程序配合。

从 PostgreSQL 9.0 开始，可以使用流复制（参见第 26.2.5 节），更省力地获得同样的收益。

26.5. 热备

热备是用来描述服务器在归档恢复或备库模式下仍然可以接受连接并运行只读查询的能力的术语。这对于复制用途以及把备份以极高精度恢复到目标状态都很有用。术语热备还指服务器能够在用户持续运行查询和/或保持连接不断开的同时，从恢复状态切换到正常运行状态的能力。

在热备模式下运行查询与正常查询操作类似，不过正如下文所述，在使用和管理上存在一些差异。

26.5.1. 用户概览

当备库上的 hot_standby 参数被设置为真时，一旦恢复把系统带到一致状态，它就会开始接受连接。所有这类连接都严格是只读的，甚至不能写入临时表。

备库上的数据需要一些时间才能从主库到达，因此主库和备库之间会有可测量的延迟。因此，在主库和备库上几乎同时运行同一查询，可能会返回不同的结果。我们说备库上的数据与主库是最终一致的。一旦某个事务的提交记录在备库上被重放，该事务所做的修改就会对备库上之后取得的所有新快照可见。快照可以在每个查询开始时取得，也可以在每个事务开始时取得，这取决于当前的事务隔离级别。详见第 13.2 节。

在热备期间启动的事务可以发出下列命令：

- 查询访问：SELECT、COPY TO
- 游标命令：DECLARE、FETCH、CLOSE
- 参数：SHOW、SET、RESET
- 事务管理命令：
  - BEGIN、END、ABORT、START TRANSACTION
  - SAVEPOINT、RELEASE、ROLLBACK TO SAVEPOINT
  - EXCEPTION 块和其他内部子事务
- LOCK TABLE，不过只限于显式指定下列模式之一时：ACCESS SHARE、ROW SHARE 或 ROW EXCLUSIVE
- 计划和资源：PREPARE、EXECUTE、DEALLOCATE、DISCARD
- 插件和扩展：LOAD
- UNLISTEN

在热备期间启动的事务永远不会被分配事务 ID，也不能写入系统预写式日志。因此，下列动作都会产生错误消息：

- 数据操纵语言（DML）：INSERT、UPDATE、DELETE、COPY FROM、TRUNCATE。请注意，恢复期间不存在任何允许执行触发器的动作。这个限制甚至适用于临时表，因为不分配事务 ID 就无法读取或写入表行，而目前热备环境中无法分配事务 ID。
- 数据定义语言（DDL）：CREATE、DROP、ALTER、COMMENT。这个限制甚至适用于临时表，因为执行这些操作需要更新系统目录表。
- SELECT ... FOR SHARE | UPDATE，因为不更新底层数据文件就无法获取行锁。
- 作用在 SELECT 语句上、会生成 DML 命令的规则。
- LOCK，如果它显式请求高于 ROW EXCLUSIVE MODE 的模式。
- 简写默认形式的 LOCK，因为它请求的是 ACCESS EXCLUSIVE MODE。
- 显式设置为非只读状态的事务管理命令：
  - BEGIN READ WRITE、START TRANSACTION READ WRITE

  - SET TRANSACTION READ WRITE、SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE
  - SET transaction_read_only = off

- 两阶段提交命令：PREPARE TRANSACTION、COMMIT PREPARED、ROLLBACK PREPARED，因为即使是只读事务，在准备阶段（两阶段提交的第一阶段）也需要写入 WAL。
- 序列更新：nextval()、setval()
- LISTEN、NOTIFY

在正常运行中，“只读”事务允许使用 LISTEN 和 NOTIFY，因此热备会话受到的限制要比普通只读会话稍微更严格一些。将来的版本中，这些限制中的某些可能会被放宽。

在热备期间，参数transaction_read_only 始终为真，并且不能更改。不过，只要不尝试修改数据库，热备期间的连接行为与其他数据库连接大致相同。如果发生故障切换或计划内切换，数据库将切换到正常处理模式。服务器切换模式时，会话仍会保持连接。一旦热备结束，就可以发起读写事务（即使该会话是在热备期间开始的）。

用户可以执行 SHOW transaction_read_only，判断自己的会话是否只读。此外，还有一组函数（表 9.80）可用于获取备库信息。借助它们，可以编写感知数据库当前状态的程序，用来监控恢复进度，或编写将数据库恢复到特定状态的复杂程序。

26.5.2. 处理查询冲突

主库和备库在许多方面都是松耦合的。主库上的动作会对备库产生影响，因此它们之间可能出现负面交互或冲突。最容易理解的冲突是性能：如果主库上正在执行一次大规模数据装载，那么备库上也会产生类似的 WAL 记录流，因此备库查询可能会争用系统资源，例如 I/O。

热备还可能发生其他类型的冲突。这些冲突属于硬冲突，因为查询可能需要被取消，并且在某些情况下还需要断开会话来解决冲突。系统为用户提供了若干种处理这些冲突的方法。冲突场景包括：

- 主库上获取的 Access Exclusive 锁，包括显式的LOCK 命令和各种 DDL 操作，会与备库查询中的表访问发生冲突。
- 主库上删除表空间，会与备库查询把该表空间用于临时工作文件发生冲突。
- 主库上删除数据库，会与备库上连接到该数据库的会话发生冲突。
- 来自 WAL 的清理记录在应用时，会与那些快照仍然能够“看到”将被删除的行的备库事务发生冲突。
- 来自 WAL 的清理记录在应用时，会与备库上访问目标页面的查询发生冲突，而不管待删除的数据是否可见。

在主库上，这些情况只会导致等待；用户可以选择取消冲突动作中的任意一方。但是在备库上没有这种选择：已经被写入 WAL 的动作早已在主库上发生，因此备库在应用它时不能失败。此外，让 WAL 应用无限期等待通常是很不可取的，因为备库的状态会越来越落后于主库。因此，系统提供了一种机制，用来强制取消那些与即将应用的 WAL 记录发生冲突的备库查询。

一个典型例子是，主库上的管理员在某个表上执行DROP TABLE，而备库上正有查询访问该表。显然，如果DROP TABLE 在备库上被应用，该备库查询就无法继续。如果这种情况发生在主库

上，DROP TABLE 会等待直到其他查询结束。但在主库上执行 DROP TABLE 时，主库并不知道备库上正在运行哪些查询，因此它不会等待这些备库查询。于是，WAL 变更记录会在备库查询仍在运行时到达备库，从而引发冲突。备库要么延迟应用这条 WAL 记录（以及之后的所有记录），要么取消冲突查询，以便应用DROP TABLE。

当冲突查询很短时，通常最好通过稍微延迟 WAL 应用来让它完成；但 WAL 应用长时间延迟通常并不可取。因此，取消机制提供了max_standby_archive_delay 和 max_standby_streaming_delay 这两个参数，用于定义 WAL 应用允许的最大延迟。一旦应用新收到的 WAL 数据所花费的时间超过相应延迟设置，冲突查询就会被取消。之所以有两个参数，是为了分别针对从归档读取 WAL 数据的场景（即从基础备份进行初始恢复，或者“追赶”一台已经远远落后的备库）以及通过流复制读取 WAL 数据的场景指定不同的延迟值。

如果一台备库主要用于高可用性，那么最好把这些延迟参数设置得较短，这样服务器就不会因为备库查询导致的延迟而远远落后于主库。但是，如果该备库的用途是执行长时间运行的查询，那么较高甚至无限的延迟值可能更合适。不过要记住，如果某个长时间运行的查询延迟了 WAL 记录的应用，它也可能使备库上的其他会话看不到主库上的最新变更。

一旦 max_standby_archive_delay 或 max_standby_streaming_delay 指定的延迟被超越，冲突查询将被取消。这通常仅导致一个取消错误，尽管在重放一个DROP DATABASE 的情况下整个冲突会话都将被终止。另外，如果冲突发生在一个被空闲事务持有的锁上，该冲突会话会被终止（这种行为可能在未来被改变）。

被取消的查询可以立即重试（当然，要先开始一个新事务）。由于查询取消取决于正在重放的 WAL 记录的性质，被取消的查询再次执行时完全可能成功。

请记住，延迟参数要与备库收到 WAL 数据之后经过的时间进行比较。因此，留给备库上任何一个查询的宽限期从不会超过延迟参数，并且如果备库已经由于等待之前的查询完成而落后或者因为过重的更新负载而无法跟上主库，宽限期可能会更少。

备库查询与 WAL 重放发生冲突的最常见原因是“过早清理”。正常情况下，当不再有事务需要看到旧行版本来保证符合 MVCC 规则的数据可见性时，PostgreSQL 允许清理这些旧行版本。不过，这条规则只能应用于在主库上执行的事务。因此，主库上的清理有可能移除某个备库事务仍然可见的行版本。

有经验的用户应注意，行版本清理和行版本冻结都有可能与备库查询冲突。手动运行 VACUUM FREEZE 很可能导致冲突，即使表中没有更新过或删除过的行也是如此。

用户应当清楚，主库上经常并且大量更新的表，会很快导致备库上的长时间运行查询被取消。在这种情况下，把max_standby_archive_delay 或 max_standby_streaming_delay 设置为有限值，可以视作类似于设置statement_timeout。

如果备库查询被取消的次数多得令人无法接受，也存在补救办法。第一种选择是设置 hot_standby_feedback 参数，它会阻止VACUUM 移除最近死亡的行，因此不会发生清理冲突。如果这样做，你应当注意这会延迟主库上死行的清理，从而可能导致不希望出现的表膨胀。不过，清理情况不会比直接在主库上运行这些备库查询更糟，而且你仍然可以获得把执行卸载到备库上的好处。如果备库经常连接又断开，你可能还需要进行一些调整，以应对无法提供 hot_standby_feedback 反馈的那段时间。例如，可以考虑增大max_standby_archive_delay，使查询在断开期间不会因为 WAL 归档文件中的冲突而被迅速取消。你也应考虑增大 max_standby_streaming_delay，以避免重新连接后由于新到达的流式 WAL 条目而被快速取消。

另一种选择是增大主库上的 vacuum_defer_cleanup_age，使死行不会像通常那样很快被清理。这样，无需将 max_standby_streaming_delay 设得很大，就能让备库查询在被取消前有更多执行时间。不过，这种方法难以保证具体的执行时间窗口，因为 vacuum_defer_cleanup_age 是按主库上执行的事务数来衡量的。

查询取消的数量及其原因可以通过备库上的 pg_stat_database_conflicts 系统视图查看。pg_stat_database 系统视图也包含汇总信息。

26.5.3. 管理员概览

如果 hot_standby 为 on（默认值，设置在 postgresql.conf 中），且存在 recovery.conf 文件，服务器就会以热备模式运行。不过，可能还要过一段时间才允许热备连接，因为服务器只有完成足够的恢复，达到可供查询运行的一致状态后，才会接受连接。在此期间，尝试连接的客户端会被拒绝，并收到错误消息。要确认服务器已启动，可以在应用中循环尝试连接，或在服务器日志中查找以下消息：

```
LOG:  entering standby mode

... then some time later ...

LOG:  consistent recovery state reached
LOG:  database system is ready to accept read only connections
```

主库每次检查点都会记录一致性信息。如果主库的 wal_level 未设为 replica 或 logical，则读取这一期间写入的 WAL 时不能启用热备。如果同时满足以下两个条件，达到一致状态也可能延迟：

- 某个写事务包含超过 64 个子事务
- 非常长生命周期的写事务

如果运行基于文件的日志传送（“温备”），可能需要等到下一个 WAL 文件到达，最长等待时间可达到主库上的 archive_timeout 设置值。

如果在主库上更改了某些参数，就需要重新配置备库上的相应设置。对于这些参数，备库上的值必须大于或等于主库上的值。因此，要增大这些值，应先修改所有备库，再修改主库。反之，要减小这些值，应先修改主库，再修改所有备库。如果这些参数设置得不够大，备库会拒绝启动。此时可以提高参数值，然后重启服务器，再次开始恢复。这些参数是：

- max_connections
- max_prepared_transactions
- max_locks_per_transaction
- max_worker_processes

管理员为 max_standby_archive_delay 和 max_standby_streaming_delay 选择合适的设置非常重要。最佳选择取决于业务优先级。例如，如果服务器的主要任务是充当高可用服务器，那么你会希望延迟设置较低，甚至可能设为零，尽管这是一个非常激进的设置。如果备库承担的是决策支持查询的附加服务器角色，那么把最大延迟设置为数小时甚至 -1（意味着永远等待查询完成）也可能是可以接受的。

主库上写出的事务状态“提示位”不会被 WAL 记录，因此备库上的数据很可能会再次写出这些提示位。这样一来，即使所有用户都是只读的，备库仍然会执行磁盘写操作；不过数据值本身并不会发生变化。用户仍然会写出大型排序临时文件，并重新生成 relcache 信息文件，因此在热备模式下，数据库没有任何部分是真正只读的。还要注意，使用dblink 模块写入远程数据库，以及借助 PL 函数执行其他数据库外部操作，依然是可能的，即使该事务在本地是只读的。

在恢复模式下，不接受下列类型的管理命令：

- 数据定义语言（DDL）：例如 CREATE INDEX
- 权限和所有权：GRANT、REVOKE、REASSIGN
- 维护命令：ANALYZE、VACUUM、CLUSTER、REINDEX

再次注意，这些命令中的某些在主库上的“只读”事务中实际上是被允许的。

因此，你不能创建只存在于备库上的额外索引，也不能创建只存在于备库上的统计信息。如果需要这些管理命令，应在主库上执行，最终这些变更会传播到备库。

pg_cancel_backend() 和 pg_terminate_backend() 可以作用于用户后端，但不能作用于执行恢复操作的启动进程。pg_stat_activity 不会把正在恢复的事务显示为活动状态。因此，恢复期间pg_prepared_xacts 始终为空。如果你希望解决处于不确定状态的预备事务，请查看主库上的pg_prepared_xacts，并在那里发出命令解决这些事务，或者在恢复结束后再解决它们。

与正常情况一样，pg_locks 会显示后端持有的锁。pg_locks 还会显示一个由启动进程管理的虚拟事务，它拥有所有正在被恢复重放的事务所持有的AccessExclusiveLocks。请注意，启动进程不会为了修改数据库而获取锁，因此除了AccessExclusiveLocks 之外，其他锁不会在启动进程的pg_locks 中显示；它们只是被假定存在。

Nagios 的 check_pgsql 插件可以工作，因为它检查的简单信息是存在的。check_postgres 监控脚本也可以工作，尽管其中某些被报告的值可能会给出不同或令人困惑的结果。例如，最近一次清理时间不会被维护，因为备库上不会发生清理。不过，主库上执行的清理仍然会把其变更发送到备库。

恢复期间，WAL 文件控制命令不可用，例如pg_start_backup、pg_switch_wal 等。

可动态载入的模块可以工作，包括pg_stat_statements。

咨询锁在恢复期间可以正常工作，包括死锁检测。注意，咨询锁从不会被 WAL 记录，因此主库或备库上的咨询锁都不可能与 WAL 重放发生冲突。同样，也不可能在主库上获取一个咨询锁，却在备库上触发一个类似的咨询锁。咨询锁只与获取它们的那台服务器相关。

基于触发器的复制系统，如Slony、Londiste 和 Bucardo，根本无法在备库上运行；不过只要它们的变更不会被发送到备库并在那里应用，它们在主库上就可以正常工作。WAL 重放并不是基于触发器的，因此你不能把备库作为任何需要额外数据库写操作或依赖触发器的系统的中继节点。

不能分配新的 OID，不过某些UUID 生成器仍可工作，只要它们不依赖于向数据库写入新的状态。

目前，在只读事务期间不允许创建临时表，因此某些现有脚本在这种情况下将无法正常运行。这个限制可能会在未来版本中放宽。这既涉及 SQL 标准兼容性问题，也涉及技术问题。

只有在表空间为空时 DROP TABLESPACE 才能成功。某些备库用户可能正在通过他们的 temp_tablespaces 参数使用该表空间。如果该表空间中存在临时文件，所有活动查询都将被取消，以确保临时文件被移除，这样该表空间才能被移除并且 WAL 重放可以继续。

在主库上执行 DROP DATABASE 或 ALTER DATABASE ... SET TABLESPACE 会生成一条 WAL 记录，从而强制断开备库上所有连接到该数据库的用户。这个动作会立即发生，而不受 max_standby_streaming_delay 设置的影响。注意，ALTER DATABASE ... RENAME 不会断开用户，这在大多数情况下不会被注意到，但如果程序依赖某种基于数据库名的机制，在某些情况下可能会导致混乱。

在普通（非恢复）模式下，如果你对一个具有登录能力的角色执行DROP USER 或 DROP ROLE，而该用户仍然处于连接状态，那么已连接用户不会发生任何变化 – 他们会继续保持连接，不过之后不能重新连接。这种行为在恢复期间同样适用，因此在主库上执行一次DROP USER 并不会断开备库上的该用户连接。

统计收集器在恢复期间保持活动状态。所有扫描、读取、块、索引使用情况等，都会在备库上正常记录。重放操作不会重复记录其在主库上产生的统计影响，因此重放一次插入不会增加 pg_stat_user_tables 的 Inserts 列。统计文件会在恢复开始时被删除，因此主库和备库的统计信息不同；这是特性，不是缺陷。

恢复期间自动清理不会运行。它会在恢复结束时正常启动。

检查点进程和后台写入进程在恢复期间是活动的。检查点进程会执行重启点（类似于主库上的检查点），后台写入进程会执行正常的块清理活动。这可能包括更新存储在备库上的提示位信息。恢复期间接受CHECKPOINT 命令，不过它执行的是重启点，而不是新的检查点。

26.5.4. 热备参数参考

多个参数已经在第 26.5.2 节和第 26.5.3 节中提到过。

在主库上，可以使用wal_level 和 vacuum_defer_cleanup_age 参数。max_standby_archive_delay 和 max_standby_streaming_delay 在主库上设置时没有效果。

在备库上，可以使用 hot_standby、max_standby_archive_delay 和 max_standby_streaming_delay 参数。只要服务器仍处于备库模式，vacuum_defer_cleanup_age 就没有效果，不过备库成为主库后，它就会起作用。

26.5.5. 注意事项

热备有若干限制。这些限制在未来的版本中可以、也很可能会被修复：

- 在能够取得快照之前，必须完整了解正在运行的事务。使用大量子事务（目前超过 64 个）的事务，会把只读连接的启动推迟到持续时间最长的写事务完成之后。如果发生这种情况，会向服务器日志发送解释性消息。
- 备库查询的有效起始点是在主库的每个检查点上生成的。如果主库处于关闭状态时备库被关闭，那么在主库再次启动并在 WAL 日志中生成更多起始点之前，备库可能无法重新进入热备状态。不过在最常见的场景中，这通常不是问题。一般来说，如果主库关闭且不再可用，很可能是出现了严重故障，此时无论如何都需要把备库提升为新的主库。而在主库被有意关闭的情况下，协调好备库顺利成为新主库，本来也就是标准流程。
- 在恢复结束时，由预备事务持有的AccessExclusiveLocks 将需要正常数量两倍的锁表项。如果你计划运行大量并发的预备事务，而这些事务通常会持有AccessExclusiveLocks，或者你计划运行一个会持有许多AccessExclusiveLocks 的大事务，那么建议选择更大的max_locks_per_transaction 值，可能要达到主库上该参数值的两倍。如果你的max_prepared_transactions 设置为 0，则完全不需要考虑这一点。
- 热备中尚不支持可串行化事务隔离级别。（详见第 13.2.3 节和第 13.4.1 节。）在热备模式下尝试把事务设置为可串行化隔离级别会产生错误。

第 27 章 恢复配置

本章介绍 recovery.conf 文件中可用的设置。它们只在恢复期间生效。如果之后还要进行恢复，必须重新设置这些参数。恢复开始后，就不能再更改它们。

recovery.conf 中的设置使用 name = 'value' 格式，每行指定一个参数。井号（#）表示该行余下内容为注释。要在参数值中嵌入单引号，请写两个单引号（''）。

安装目录的 share/ 目录中提供了示例文件 share/recovery.conf.sample。

27.1. 归档恢复设置

restore_command (string)

用于获取 WAL 文件系列的一个已归档段的本地 shell 命令。这个参数是归档恢复所必需的，但是对于流复制是可选的。在该字符串中的任何%f 会被替换为从归档中获得的文件的名字，并且任何%p 会被在服务器上的复制目标路径名替换（该路径名是相对于当前工作目录的，即集簇的数据目录）。任何%r 会被包含上一个可用重启点的文件的名字所替换。在那些必须被保留用于使得一次恢复变成可重启的文件中，这个文件是其中最早的一个，因此这个信息可以被用来把归档截断为支持从当前恢复重启所需的最小值。%r 通常只被温备配置（见第 26.2 节）所使用。要嵌入一个真正的% 字符，需要写成%%。

很重要的一点是，该命令只有在成功时才返回一个为零的退出状态。该命令将会被要求获取归档中不存在的文件；遇到这种情况时，它必须返回非零。示例：

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'  # Windows
```

一个例外是如果该命令被一个信号（不是 SIGTERM，它是数据库服务器关闭的一部分）或者一个 shell 错误（例如命令未找到）终止，则恢复将会中止并且服务器将不会启动。

archive_cleanup_command (string)

这个可选参数指定了一个 shell 命令，它将在每一个重启点被执行。archive_cleanup_command 的目的是提供一种清除不再被备库需要的旧的已归档 WAL 文件的机制。任何%r 会被替换为包含最后一个可用重启点的文件的名称。那是使一次恢复变成可重启的所必须被保留的最早的文件，并且因此比%r 更早的所有文件可以被安全地移除。这个信息可以被用来把归档截断为支持从当前恢复重启所需的最小值。对于单一备库配置，pg_archivecleanup 模块常常被用在 archive_cleanup_command 中，例如：

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

但是注意，如果多个备库正在从同一个归档目录中恢复，你将需要保证只有当所有服务器都不再需要这些 WAL 文件时才会删除它们。archive_cleanup_command 通常被用于一种温备配置（见第 26.2 节）中。要在该命令中嵌入一个真正的% 字符，需要写成%%。

如果该命令返回一个非零退出状态，则将会写出一个警告日志消息。一个例外是如果该命令被一个信号或者一个 shell 错误（例如命令未找到）终止，则会抛出一个致命错误。

recovery_end_command (string)

此参数指定一个 shell 命令，在恢复结束时仅执行一次。此参数可选。recovery_end_command 用于提供复制或恢复后的清理机制。与

archive_cleanup_command 一样，所有 %r 都会被替换为包含最后一个有效重启点的文件的名称。

如果该命令返回一个非零退出状态，则一个警告日志消息将被写出并且不管怎样该数据库将继续启动。一个例外是如果该命令被一个信号或者 shell 错误（例如命令未找到）中止，该数据库将不会继续启动。

27.2. 恢复目标设置

默认情况下，恢复会一直进行到 WAL 日志末尾。可以使用以下参数指定更早的停止点。recovery_target、recovery_target_lsn、recovery_target_name、recovery_target_time 和 recovery_target_xid 中最多只能使用一个；如果在配置文件中指定了多个，则使用最后一个条目。

recovery_target = 'immediate'

这个参数指定恢复应该在达到一个一致状态后尽快结束，即尽早结束。在从一个在线备份中恢复时，这意味着备份结束的那个点。

在技术上，这是一个字符串参数，但是'immediate' 是目前唯一允许的值。

recovery_target_name (string)

这个参数指定（pg_create_restore_point() 所创建）的已命名的恢复点，恢复将进行到该恢复点。

recovery_target_time (timestamp)

此参数指定恢复要进行到的时间戳。精确的停止点还受 recovery_target_inclusive 影响。

recovery_target_xid (string)

此参数指定恢复要进行到的事务 ID。请注意，虽然事务 ID 在事务开始时按顺序分配，但事务完成的顺序可能与 ID 的数值顺序不同。恢复的事务是那些在指定事务之前提交的事务，并可选择包括指定事务本身。精确的停止点还受recovery_target_inclusive 影响。

recovery_target_lsn (pg_lsn)

此参数指定恢复要进行到的预写日志位置的 LSN。精确的停止点还受 recovery_target_inclusive 影响。此参数使用系统数据类型 pg_lsn 解析。

下列选项进一步指定恢复目标，并且影响到达目标时会发生什么：

recovery_target_inclusive (boolean)

指定是在所设恢复目标之后立即停止（true），还是在其之前立即停止（false）。在指定了 recovery_target_lsn、recovery_target_time 或 recovery_target_xid 时适用。此设置控制恢复是否包含 WAL 位置（LSN）、提交时间或事务 ID 分别恰好等于目标值的事务。默认值为 true。

recovery_target_timeline (string)

指定恢复到某条特定时间线。默认沿着制作基础备份时的当前时间线恢复。设为 latest 时，会恢复到归档中找到的最新时间线，这对备库很有用。除此之外，只有在复杂的再次恢复场景中才需要设置此参数：需要返回的状态本身就是一次时间点恢复之后达到的状态。相关讨论参见第 25.3.5 节。

recovery_target_action (enum)

指定在达到恢复目标时服务器应该立刻采取的动作。默认动作是 pause，这表示恢复将会被暂停。promote 表示恢复处理将会结束并且服务器将开始接受连接。最后，shutdown 将在达到恢复目标之后停止服务器。

使用 pause 设置的目的是允许对数据库执行查询，以检查这个恢复目标是否为最合适的恢复位置。暂停的状态可以使用pg_wal_replay_resume()（见表 9.81）继续，这会让恢复终结。如果这个恢复目标不是想要的停止点，那么关闭服务器，将恢复目标设置改为一个稍后的目标并且重启以继续恢复。

要让实例在想要的重放点那里准备好，shutdown 设置可以派上用场。该实例将仍能重放更多 WAL 记录（并且事实上，下次启动时必须重新回放自上一个检查点以来的 WAL 记录）。

注意，将 recovery_target_action 设为 shutdown 时，recovery.conf 不会被重命名，因此除非更改配置或手动删除 recovery.conf 文件，否则之后每次启动都会立即关闭。

如果未设置恢复目标，此设置不起作用。如果未启用 hot_standby，则 pause 的行为与 shutdown 相同。

27.3. 备库设置

standby_mode (boolean)

指定是否将 PostgreSQL 服务器作为备库启动。如果此参数为 on，服务器到达已归档 WAL 的末尾时不会停止恢复，而会继续尝试使用 restore_command 获取新的 WAL 段，和/或按照 primary_conninfo 设置连接主库，以继续恢复。

primary_conninfo (string)

指定备库连接主库所用的连接字符串，格式见第 34.1.1 节。如果某个选项未在字符串中指定，就会检查相应的环境变量（参见第 34.14 节）。如果环境变量也未设置，则使用默认值。

连接字符串应指定主库的主机名（或地址）；如果端口号与备库的默认端口不同，也应指定端口号。还应指定一个用户名，对应主库上具有适当权限的角色（参见第 26.2.5.1 节）。如果主库要求密码认证，还需要提供密码。密码可以放在 primary_conninfo 字符串中，也可以放在备库上单独的 ~/.pgpass 文件中（使用 replication 作为数据库名）。不要在 primary_conninfo 字符串中指定数据库名。

如果 standby_mode 为 off，此设置不起作用。

primary_slot_name (string)

可选地指定一个已存在的复制槽，在通过流复制连接主库时使用，以控制上游节点的资源移除（参见第 26.2.6 节）。如果未设置 primary_conninfo，此设置不起作用。

trigger_file (string)

指定一个触发文件，其出现会使备库结束恢复。即使未设置此值，也仍可以使用 pg_ctl promote 提升备库。如果 standby_mode 为 off，此设置不起作用。

`recovery_min_apply_delay`(`integer`)

默认情况下，备库会尽快恢复来自主库的 WAL 记录。保留一份延迟的数据副本可能很有用，因为它提供了纠正数据丢失错误的机会。此参数允许将恢复延迟一段固定时间；如果没有指定单位，则以毫秒计。例如，将此参数设置为 `5min` 时，只有当备库系统时间比主库报告的提交时间至少晚五分钟，备库才会重放各事务的提交。

服务器之间的复制延迟可能超过此参数的值，这种情况下不会增加延迟。注意，延迟根据主库写入的 WAL 时间戳与备库当前时间之差计算。网络延迟或级联复制配置导致的传输延迟，可能显著缩短实际等待时间。如果主库和备库的系统时钟不同步，恢复时可能比预期更早应用记录；但这通常不是主要问题，因为此参数的实用取值远大于服务器间常见的时间偏差。

延迟仅发生在事务提交的 WAL 记录上。其他记录会尽快重放；这不会造成问题，因为 MVCC 可见性规则确保在对应提交记录被应用之前，它们的效果不会可见。

恢复中的数据库达到一致状态后开始延迟，直到备库被提升或触发。此后，备库会结束恢复，不再等待。

此参数旨在用于流复制部署；不过，只要指定了此参数，它就会在所有情况下生效。使用此功能也会延迟 `hot_standby_feedback`，可能导致主库膨胀；同时使用两者时应谨慎。

> **警告**
>
> 当 `synchronous_commit` 设置为 `remote_apply` 时，同步复制会受到此设置影响；每个 `COMMIT` 都必须等待提交被应用。

第 28 章 监控数据库活动

数据库管理员常常会问：“系统现在正在做什么？”本章将讨论如何回答这个问题。

有几种工具可用于监控数据库活动和分析性能。本章的大部分内容将用于描述 PostgreSQL 的统计收集器，但也不应忽略常见的 Unix 监控程序，例如ps、top、iostat 和 vmstat。此外，一旦确定了某个查询性能不佳，可能需要使用PostgreSQL 的 EXPLAIN 命令进行进一步调查。第 14.1 节讨论了 EXPLAIN 以及理解单个查询行为的其他方法。

28.1. 标准 Unix 工具

在大多数 Unix 平台上，PostgreSQL 会修改由 ps 报告的命令标题，以便轻松识别各个服务器进程。以下是一个显示示例：

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00 postgres: background writer
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: checkpointer
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: walwriter
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00 postgres: autovacuum launcher
postgres  15558  0.0  0.0  17512  1068 ?        Ss   18:02   0:00 postgres: stats collector
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00 postgres: joe runbug 127.0.0.
1 idle
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00 postgres: tgl regression [
local] SELECT waiting
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00 postgres: tgl regression [
local] idle in transaction
```

（ps 的适当调用方式随平台而异，显示内容的细节也是如此。此示例来自较新的 Linux 系统。）这里列出的第一个进程是主服务器进程。显示的命令参数与启动该进程时使用的参数相同。接下来的五个进程是主进程自动启动的后台工作进程。（如果系统被设置为不启动统计收集器，则不会出现 “stats collector” 进程；同样，也可以禁用 “autovacuum launcher” 进程。）其余每个进程都是处理一个客户端连接的服务器进程。每个这样的进程都将其命令行显示设置为以下形式：

```
postgres: user database host activity
```

在客户端连接的整个生命周期内，用户、数据库和（客户端）主机这几项保持不变，但活动指示符会改变。活动可以是idle（即等待客户端命令）、idle in transaction（在 BEGIN 块内等待客户端），或者命令类型名称，例如SELECT。此外，服务器进程当前若正在等待另一会话持有的锁，还会附加waiting。在上面的示例中，可以推断进程 15606 正在等待进程 15610 完成事务，从而释放某个锁。（进程 15610 必定是阻塞者，因为不存在其他活动会话。在更复杂的情况下，必须查看pg_locks 系统视图，才能确定谁在阻塞谁。）

如果配置了 cluster_name，则集簇名称也会显示在ps 输出中：

```
$ psql -c 'SHOW cluster_name'
 cluster_name
--------------
 server1
(1 row)

$ ps aux|grep server1
postgres   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00 postgres: server1:
 background writer
...
```

如果您关闭了 update_process_title，则活动指示器不会更新；进程标题仅在新进程启动时设置一次。在某些平台上，这可以节省可观的每命令开销；在其他平台上，则微不足道。

> **提示**
>
> Solaris 需要特殊处理。您必须使用/usr/ucb/ps，而不是/bin/ps。您还必须使用两个 w 标志，而不仅仅是一个。此外，您对postgres 命令的原始调用必须具有比每个服务器进程提供的更短的 ps 状态显示。如果未满足这三个条件，每个服务器进程的ps 输出都会显示原始的 postgres 命令行。

28.2. 统计收集器

PostgreSQL 的统计收集器是一个支持收集和报告服务器活动信息的子系统。目前，对表和索引的访问会按磁盘块和行两种口径计数。每个表中的总行数，以及每个表的清理和分析操作的信息也被计数。它还可以统计对用户定义函数的调用和每个函数中花费的总时间。

PostgreSQL 也支持报告关于系统当前正在发生的情况的动态信息，例如其他服务器进程当前正在执行的确切命令，以及系统中存在哪些其他连接。此功能独立于收集器进程。

28.2.1. 统计信息收集配置

由于收集统计信息会增加查询执行的开销，因此可以配置系统是否收集信息。这由通常在postgresql.conf 中设置的配置参数控制（有关设置配置参数的详细信息，请参阅第 19 章）。

参数 track_activities 启用对任意服务器进程当前执行命令的监控。

参数 track_counts 控制是否收集关于表和索引访问的统计信息。

参数 track_functions 启用对用户定义函数使用的跟踪。

参数 track_io_timing 启用对块读取和写入时间的监控。

通常，这些参数会设置在postgresql.conf 中，以便它们适用于所有服务器进程，但也可以在单个会话中使用SET 命令打开或关闭它们。（为防止普通用户隐藏其活动不被管理员发现，只有超级用户才能使用SET 更改这些参数。）

统计收集器通过临时文件把收集到的信息传递给其他 PostgreSQL 进程。这些文件默认存放在由 stats_temp_directory 参数指定的目录中，即pg_stat_tmp。为了获得更好的性能，可以把 stats_temp_directory 指向基于 RAM 的文件系统，以减少物理 I/O 需求。当服务器正常关闭时，统计信息数据的永久副本会存储在pg_stat 子目录中，以便在服务器重启后保留这些统计信息。当服务器启动时执行恢复时（例如立即关闭后、服务器崩溃后以及进行时间点恢复时），所有统计计数器都会被重置。

28.2.2. 查看统计信息

有几个预定义视图列在表 28.1 中，可用于显示系统的当前状态。另有几个视图列在表 28.2 中，可用于显示统计信息收集的结果。或者，也可以像第 28.2.3 节中所述那样，使用底层的统计函数构建自定义视图。

当使用统计信息来监视收集到的数据时，重要的是要意识到信息不会立即更新。每个单独的服务器进程会在转为空闲之前把新的统计计数发送给收集器，因此仍在进行中的查询或事务不会影响显示的总计。并且，收集器本身最多每隔PGSTAT_STAT_INTERVAL 毫秒才会发出一份新报告一

次（除非在构建服务器时做过修改，否则为 500 毫秒）。因此，显示的信息会滞后于实际活动。然而，由track_activities 收集的当前查询信息始终是最新的。

另一个重要点是，当要求服务器进程显示这些统计信息中的任何一种时，它会先取回收集器进程最近发出的报告，然后在当前事务结束之前，对所有统计视图和函数一直使用这一快照。因此，只要你继续当前事务，统计信息就会显示静态信息。类似地，在事务中第一次请求这类当前查询信息时，就会收集所有会话当前查询的信息，并且在整个事务期间显示的都是同一份信息。这是一个特性，而不是一个错误，因为它允许你对统计信息执行多个查询并对结果做关联，而不必担心数字在你眼前变化。但如果希望每个查询都看到新的结果，请确保在任何事务块之外执行这些查询。或者，也可以调用pg_stat_clear_snapshot()，它会丢弃当前事务的统计快照（如果有）。下一次使用统计信息时，就会获取一个新的快照。

事务还可以在视图 pg_stat_xact_all_tables、pg_stat_xact_sys_tables、pg_stat_xact_user_tables 和 pg_stat_xact_user_functions 中看到自己的统计信息（这些统计信息尚未传送给收集器）。这些数字不像上面所述那样起作用；相反，它们在事务期间持续更新。

表 28.1. 动态统计视图

视图名称	描述
pg_stat_activity	每个服务器进程一行，显示与该进程当前活动有关的信息，例如状态和当前查询。详见 pg_stat_activity。
pg_stat_replication	每个 WAL 发送进程一行，显示向其所连接的备库进行复制的统计信息。详见 pg_stat_replication。
pg_stat_wal_receiver	只有一行，显示 WAL 接收进程从其所连接服务器接收数据的统计信息。详见 pg_stat_wal_receiver。
pg_stat_subscription	每个订阅至少一行，显示订阅工作进程的信息。详见 pg_stat_subscription。
pg_stat_ssl	每个连接（普通连接和复制连接）一行，显示此连接所使用 SSL 的信息。详见 pg_stat_ssl。
pg_stat_progress_vacuum	每个正在运行VACUUM 的后端（包括自动清理工作进程）一行，显示当前进度。请参阅第 28.4.1 节。

表 28.2. 已收集统计信息的视图

视图名称	描述
pg_stat_archiver	只有一行，显示 WAL 归档进程活动的统计信息。详见 pg_stat_archiver。
pg_stat_bgwriter	只有一行，显示后台写入器活动的统计信息。详见 pg_stat_bgwriter。
pg_stat_database	每个数据库一行，显示整个数据库的统计信息。详见 pg_stat_database。
pg_stat_database_conflicts	每个数据库一行，显示备库上因与恢复冲突而取消查询的数据库范围统计信息。详见 pg_stat_database_conflicts。
pg_stat_all_tables	当前数据库中的每个表一行，显示访问该表的统计信息。详见 pg_stat_all_tables。
pg_stat_sys_tables	和pg_stat_all_tables 一样，但只显示系统表。

视图名称	描述
pg_stat_user_tables	和pg_stat_all_tables 一样，但只显示用户表。
pg_stat_xact_all_tables	与 pg_stat_all_tables 相似，但只统计当前事务中截至目前执行的操作（这些操作尚未计入 pg_stat_all_tables 及相关视图）。此视图不包含存活行和死行数量以及清理、分析操作的列。
pg_stat_xact_sys_tables	和pg_stat_xact_all_tables 一样，但只显示系统表。
pg_stat_xact_user_tables	和pg_stat_xact_all_tables 一样，但只显示用户表。
pg_stat_all_indexes	当前数据库中的每个索引一行，显示访问该索引的统计信息。详见 pg_stat_all_indexes。
pg_stat_sys_indexes	和pg_stat_all_indexes 一样，但只显示系统表上的索引。
pg_stat_user_indexes	和pg_stat_all_indexes 一样，但只显示用户表上的索引。
pg_statio_all_tables	当前数据库中的每个表一行，显示该表的 I/O 统计信息。详见 pg_statio_all_tables。
pg_statio_sys_tables	和pg_statio_all_tables 一样，但只显示系统表。
pg_statio_user_tables	和pg_statio_all_tables 一样，但只显示用户表。
pg_statio_all_indexes	当前数据库中的每个索引一行，显示该索引的 I/O 统计信息。详见 pg_statio_all_indexes。
pg_statio_sys_indexes	和pg_statio_all_indexes 一样，但只显示系统表上的索引。
pg_statio_user_indexes	和pg_statio_all_indexes 一样，但只显示用户表上的索引。
pg_statio_all_sequences	当前数据库中的每个序列一行，显示该序列的 I/O 统计信息。详见 pg_statio_all_sequences。
pg_statio_sys_sequences	和pg_statio_all_sequences 一样，但只显示系统序列（目前没有定义系统序列，因此这个视图总是为空）。
pg_statio_user_sequences	和pg_statio_all_sequences 一样，但只显示用户序列。
pg_stat_user_functions	每个被跟踪的函数一行，显示该函数执行的统计信息。详见 pg_stat_user_functions。
pg_stat_xact_user_functions	和pg_stat_user_functions 相似，但是只统计在当前事务期间的调用（还没有被包括在pg_stat_user_functions 中）。

每个索引的统计信息对于判断哪些索引正在被使用以及它们有多有效尤其有用。

pg_statio_ 视图主要用于判断缓冲区缓存的有效性。如果实际磁盘读取次数远少于缓冲区命中次数，缓存就能满足大多数读取请求，而无需调用内核。不过，这些统计信息并不能反映全部情况：由于PostgreSQL 处理磁盘 I/O 的方式，不在PostgreSQL 缓冲区缓存中的数据仍可能驻留在内核的 I/O 缓存中，因而仍可在不执行物理读取的情况下取得。对于希望更详细地了解 PostgreSQL I/O 行为的用户，建议将PostgreSQL 统计收集器与能够深入查看内核 I/O 处理情况的操作系统工具结合使用。

表 28.3. pg_stat_activity 视图

列	类型	描述
datid	oid	这个后端连接到的数据库的 OID
datname	name	这个后端连接到的数据库的名称
pid	integer	这个后端的进程 ID
usesysid	oid	登录到这个后端的用户的 OID
usename	name	登录到此后端的用户名称
application_name	text	连接到此后端的应用名称
client_addr	inet	连接到这个后端的客户端的 IP 地址。如果这个字段为空值，它表示客户端通过服务器机器上的一个 Unix 套接字连接或者这是一个内部进程，如自动清理。
client_hostname	text	已连接的客户端的主机名，由 client_addr 的反向 DNS 查找报告。这个字段将只对 IP 连接非空，并且只有log_hostname 被启用时才会非空。
client_port	integer	客户端用于与此后端通信的 TCP 端口号；如果使用 Unix 套接字，则为-1
backend_start	timestamp with time zone	这个进程被启动的时间。对客户端后端来说，这就是客户端连接到服务器的时间。
xact_start	timestamp with time zone	这个进程的当前事务被启动的时间，如果没有活动事务则为空值。如果当前查询是其事务中的第一个查询，这一列等于query_start 列。
query_start	timestamp with time zone	当前活动查询开始的时间；如果 state 不是active，则为上一个查询的开始时间
state_change	timestamp with time zone	state 上一次发生改变的时间
wait_event_type	text	后端正在等待的事件类型（如果有），否则为空值。可能的值为： • LWLock：后端正在等待轻量级锁。每个这样的锁保护共享内存中的某个特定数据结构。wait_event 将包含标识该轻量级锁用途的名称。（有些锁具有专门的名称，另一些锁属于一组用途相似的锁。） • Lock：后端正在等待重量级锁。重量级锁也称为锁管理器锁，或简称为锁，主要保护 SQL 可见的对象，例如表。不过，它们也用于确保某些内部操作（例如关系扩展）的互斥。wait_event 将标识正在等待的锁类型。

列	类型	描述
		• BufferPin：服务器进程正在等待访问一个数据缓冲区，访问期间不得有其他进程检查该缓冲区。如果另一个进程持有尚未关闭的游标，且该游标最近读取的数据来自此缓冲区，缓冲区钉住等待就可能持续较长时间。 • Activity：服务器进程处于空闲状态。这用于在主处理循环中等待活动的系统进程。wait_event 将标识具体的等待点。 • Extension：服务器进程正在等待扩展模块中的活动。模块可以利用此类别跟踪自定义等待点。 • Client：服务器进程正在等待套接字上来自用户应用程序的某种活动，并且服务器预期发生的事情独立于其内部进程。wait_event 将标识具体的等待点。 • IPC：服务器进程正在等待服务器中另一个进程的某种活动。wait_event 将标识具体的等待点。 • Timeout：服务器进程正在等待超时到期。wait_event 将标识具体的等待点。 • IO：服务器进程正在等待 I/O 完成。wait_event 将标识具体的等待点。
wait_event	text	如果后端当前正在等待，则为等待事件名称，否则为空值。详见表 28.4。

列	类型	描述
state	text	该后端当前的总体状态。可能的值为： • active：后端正在执行一个查询。 • idle：后端正在等待新的客户端命令。 • idle in transaction：后端处于事务中，但当前没有执行查询。 • idle in transaction (aborted)：该状态与idle in transaction 类似，但事务中的某个语句导致了错误。 • fastpath function call：后端正在执行一个快速路径函数。 • disabled：如果在此后端中禁用了 track_activities，就会报告此状态。
backend_xid	xid	这个后端的顶层事务 ID，如果存在。
backend_xmin	xid	当前后端的xmin 视界。
query	text	这个后端最近查询的文本。如果state 为active，这个字段显示当前正在执行的查询。在所有其他状态下，它显示上一个被执行的查询。默认情况下，查询文本会被截断至 1024 个字节，这个值可以通过参数track_activity_query_size更改。
backend_type	text	当前后端的类型。可能的类型为autovacuum launcher、autovacuum worker、logical replication launcher、logical replication worker、parallel worker、background writer、client backend、checkpointer、startup、walreceiver、walsender 和walwriter。此外，由扩展注册的后台工作进程还可能有额外的类型。

pg_stat_activity 视图每个服务器进程将有一行，显示与该进程当前活动相关的信息。

> **注意**
>
> wait_event 和 state 列彼此独立。如果某个后端处于 active 状态，它可能在等待（waiting）某个事件，也可能没有等待。如果状态为active 且 wait_event 非空，就意味着某个查询正在执行，但在系统中的某处被阻塞了。

表 28.4. wait_event 描述

等待事件类型	等待事件名称	描述
LWLock	ShmemIndexLock	等待在共享内存中找到或分配空间。
	OidGenLock	等待分配或指派 OID。
	XidGenLock	等待分配或指派事务 ID。
	ProcArrayLock	等待获取快照，或在事务结束时清除事务 ID。
	SInvalReadLock	等待从共享失效队列中取出或移除消息。
	SInvalWriteLock	等待向共享失效队列添加消息。
	WALBufMappingLock	等待在 WAL 缓冲区中替换一个页面。
	WALWriteLock	等待 WAL 缓冲区写入磁盘。
	ControlFileLock	等待读取或更新控制文件，或创建新的 WAL 文件。
	CheckpointLock	等待执行检查点。
	CLogControlLock	等待读取或更新事务状态。
	SubtransControlLock	等待读取或更新子事务信息。
	MultiXactGenLock	等待读取或更新共享的多事务状态。
	MultiXactOffsetControlLock	等待读取或更新多事务偏移映射。
	MultiXactMemberControlLock	等待读取或更新多事务成员映射。
	RelCacheInitLock	等待读取或写入关系缓存初始化文件。
	CheckpointerCommLock	等待管理 fsync 请求。
	TwoPhaseStateLock	等待读取或更新预备事务的状态。
	TablespaceCreateLock	等待创建或删除表空间。
	BtreeVacuumLock	等待读取或更新 B-树索引的清理相关信息。
	AddinShmemInitLock	等待管理共享内存中的空间分配。
	AutovacuumLock	自动清理工作进程或启动进程正在等待更新或读取自动清理工作进程的当前状态。
	AutovacuumScheduleLock	等待确认已选定要清理的表仍然需要清理。

等待事件类型	等待事件名称	描述
	SyncScanLock	等待获取同步扫描所用的表扫描起始位置。
	RelationMappingLock	等待更新用于存储系统目录到文件节点映射的关系映射文件。
	AsyncCtlLock	等待读取或更新共享通知状态。
	AsyncQueueLock	等待读取或更新通知消息。
	SerializableXactHashLock	等待获取或存储可串行化事务的信息。
	SerializableFinishedListLock	等待访问已完成的可串行化事务列表。
	SerializablePredicateLockListLock	等待对可串行化事务所持有的锁列表执行操作。
	OldSerXidLock	等待读取或记录相互冲突的可串行化事务。
	SyncRepLock	等待读取或更新同步副本的信息。
	BackgroundWorkerLock	等待读取或更新后台工作进程状态。
	DynamicSharedMemoryControlLock	等待读取或更新动态共享内存状态。
	AutoFileLock	等待更新postgresql.auto.conf 文件。
	ReplicationSlotAllocationLock	等待分配或释放复制槽。
	ReplicationSlotControlLock	等待读取或更新复制槽状态。
	CommitTsControlLock	等待读取或更新事务提交时间戳。
	CommitTsLock	等待读取或更新最近设置的事务时间戳值。
	ReplicationOriginLock	等待设置、删除或使用复制源。
	MultiXactTruncationLock	等待读取或截断多事务信息。
	OldSnapshotTimeMapLock	等待读取或更新旧的快照控制信息。
	BackendRandomLock	等待生成随机数。
	LogicalRepWorkerLock	等待对逻辑复制工作进程的操作完成。
	CLogTruncationLock	等待执行txid_status，或更新该函数可用的最早事务 ID。
	WrapLimitsVacuumLock	等待更新事务 ID 和多事务消耗量的限制。
	NotifyQueueTailLock	等待更新通知消息存储的限制。
	clog	等待 clog（事务状态）缓冲区上的 I/O。
	commit_timestamp	等待提交时间戳缓冲区上的 I/O。
	subtrans	等待子事务缓冲区上的 I/O。
	multixact_offset	等待多事务偏移量缓冲区上的 I/O。

等待事件类型	等待事件名称	描述
	multixact_member	等待 multixact_member 缓冲区上的 I/O。
	async	等待 async（通知）缓冲区上的 I/O。
	oldserxid	等待 oldserxid 缓冲区上的 I/O。
	wal_insert	等待将 WAL 插入内存缓冲区。
	buffer_content	等待读取或写入内存中的数据页。
	buffer_io	等待数据页上的 I/O。
	replication_origin	等待读取或更新复制进度。
	replication_slot_io	在复制槽上等待 I/O。
	proc	等待读取或更新快速路径锁信息。
	buffer_mapping	等待将数据块与缓冲池中的缓冲区关联。
	lock_manager	等待添加或检查后端的锁，或者等待加入或退出锁组（用于并行查询）。
	predicate_lock_manager	等待添加或检查谓词锁信息。
	parallel_query_dsa	等待并行查询的动态共享内存分配锁。
	tbm	等待 TBM 共享迭代器锁。
	parallel_append	在 Parallel Append 计划执行期间等待选择下一个子计划。
	parallel_hash_join	在 Parallel Hash 计划执行期间，等待分配或交换内存块，或更新计数器。
Lock	relation	等待获取关系上的锁。
	extend	等待扩展一个关系。
	frozenid	等待更新 pg_database.datfrozenxid 和 pg_database.datminmxid。
	page	等待获取关系页上的锁。
	tuple	等待获取元组上的锁。
	transactionid	等待事务完成。
	virtualxid	等待获取虚拟事务 ID 锁。
	speculative token	等待获取推测插入锁。
	object	等待获取非关系数据库对象上的锁。
	userlock	等待获取用户锁。
	advisory	等待获取用户咨询锁。
BufferPin	BufferPin	等待获取缓冲区上的钉住。
Activity	ArchiverMain	在归档进程的主循环中等待。
	AutoVacuumMain	在自动清理启动进程的主循环中等待。
	BgWriterHibernate	在后台写入器进程中休眠等待。

等待事件类型	等待事件名称	描述
	BgWriterMain	在后台写入器进程（后台工作进程）的主循环中等待。
	CheckpointerMain	在检查点进程的主循环中等待。
	LogicalApplyMain	在逻辑应用进程的主循环中等待。
	LogicalLauncherMain	在逻辑启动进程的主循环中等待。
	PgStatMain	在统计收集器进程的主循环中等待。
	RecoveryWalAll	在恢复期间等待来自流的 WAL。
	RecoveryWalStream	在恢复期间，若所有来源（本地、归档或流）都没有可用的 WAL 数据，则在重新尝试获取 WAL 数据前等待。
	SysLoggerMain	在 syslogger 进程的主循环中等待。
	WalReceiverMain	在 WAL 接收进程的主循环中等待。
	WalSenderMain	在 WAL 发送进程的主循环中等待。
	WalWriterMain	在 WAL 写入进程的主循环中等待。
Client	ClientRead	等待从客户端读取数据。
	ClientWrite	等待写入数据到客户端。
	LibPQWalReceiverConnect	在 WAL 接收进程中等待与远程服务器建立连接。
	LibPQWalReceiverReceive	在 WAL 接收进程中等待从远程服务器接收数据。
	SSLOpenServer	在尝试连接时等待 SSL。
	WalReceiverWaitStart	等待启动进程为流复制发送初始数据。
	WalSenderWaitForWAL	在 WAL 发送进程中等待 WAL 刷盘。
	WalSenderWriteData	在 WAL 发送进程中处理 WAL 接收进程的回复时，等待任何活动。
Extension	Extension	在扩展中等待。
IPC	BgWorkerShutdown	等待后台工作进程关闭。
	BgWorkerStartup	等待后台工作进程启动。
	BtreePage	等待继续并行 B-树扫描所需的页号变得可用。
	ClogGroupUpdate	等待组领导者在事务结束时更新事务状态。
	ExecuteGather	在执行Gather 节点时等待子进程的活动。
	Hash/Batch/Allocating	等待选出的 Parallel Hash 参与者分配 hash 表。
	Hash/Batch/Electing	选出负责分配 hash 表的 Parallel Hash 参与者。

等待事件类型	等待事件名称	描述
	Hash/Batch/Loading	等待其他 Parallel Hash 参与者完成 hash 表的加载。
	Hash/Build/Allocating	等待选出的 Parallel Hash 参与者分配初始 hash 表。
	Hash/Build/Electing	选出负责分配初始 hash 表的 Parallel Hash 参与者。
	Hash/Build/HashingInner	等待其他 Parallel Hash 参与者完成对内层关系的 hash 处理。
	Hash/Build/HashingOuter	等待其他 Parallel Hash 参与者完成对外层关系的分区。
	Hash/GrowBatches/Allocating	等待选出的 Parallel Hash 参与者分配更多批次。
	Hash/GrowBatches/Deciding	选出负责决定后续批次增长的 Parallel Hash 参与者。
	Hash/GrowBatches/Electing	选出负责分配更多批次的 Parallel Hash 参与者。
	Hash/GrowBatches/Finishing	等待选出的 Parallel Hash 参与者决定后续的批次增长。
	Hash/GrowBatches/Repartitioning	等待其他 Parallel Hash 参与者完成重新分区。
	Hash/GrowBuckets/Allocating	等待选出的 Parallel Hash 参与者完成更多桶的分配。
	Hash/GrowBuckets/Electing	选出负责分配更多桶的 Parallel Hash 参与者。
	Hash/GrowBuckets/Reinserting	等待其他 Parallel Hash 参与者完成向新桶插入元组。
	LogicalSyncData	等待逻辑复制远程服务器为初始表同步发送数据。
	LogicalSyncStateChange	等待逻辑复制远程服务器改变状态。
	MessageQueueInternal	等待其他进程附加到共享消息队列。
	MessageQueuePutMessage	等待将协议消息写入共享消息队列。
	MessageQueueReceive	等待从共享消息队列接收字节。
	MessageQueueSend	等待将字节发送到共享消息队列。
	ParallelBitmapScan	等待并行位图扫描完成初始化。
	ParallelCreateIndexScan	等待并行 `CREATE INDEX` 工作进程完成堆扫描。
	ParallelFinish	等待并行工作进程完成计算。
	ProcArrayGroupUpdate	等待组领导者在事务结束时清除事务 ID。
	ReplicationOriginDrop	等待复制源变为非活动状态，以便将其删除。
	ReplicationSlotDrop	等待复制槽变为非活动状态，以便将其删除。

等待事件类型	等待事件名称	描述
	SafeSnapshot	等待READ ONLY DEFERRABLE 事务的快照。
	SyncRep	在同步复制期间等待远程服务器的确认。
Timeout	BaseBackupThrottle	在基础备份期间因限流而等待。
	PgSleep	调用pg_sleep 的进程正在等待。
	RecoveryApplyDelay	在恢复期间，由于应用 WAL 被延迟而等待。
IO	BufFileRead	等待从缓冲文件中读取。
	BufFileWrite	等待对缓冲文件的写入。
	ControlFileRead	等待读取控制文件。
	ControlFileSync	等待控制文件同步到持久存储。
	ControlFileSyncUpdate	等待对控制文件的更新同步到持久存储。
	ControlFileWrite	等待写入控制文件。
	ControlFileWriteUpdate	等待写入以更新控制文件。
	CopyFileRead	在文件复制操作期间等待读取。
	CopyFileWrite	在文件复制操作期间等待写入。
	DataFileExtend	等待关系数据文件被扩展。
	DataFileFlush	等待关系数据文件同步到持久存储。
	DataFileImmediateSync	等待关系数据文件立即同步到持久存储。
	DataFilePrefetch	等待关系数据文件的异步预取。
	DataFileRead	等待对关系数据文件的读取。
	DataFileSync	等待对关系数据文件的更改同步到持久存储。
	DataFileTruncate	等待关系数据文件被截断。
	DataFileWrite	等待对关系数据文件的写入。
	DSMFillZeroWrite	等待向动态共享内存的后备文件写入值为零的字节。
	LockFileAddToDataDirRead	在向数据目录锁文件中添加一行时等待读取。
	LockFileAddToDataDirSync	在向数据目录锁文件中添加一行时，等待数据同步到持久存储。
	LockFileAddToDataDirWrite	在向数据目录锁文件中添加一行时等待写操作。
	LockFileCreateRead	创建数据目录锁文件时等待读取。
	LockFileCreateSync	在创建数据目录锁文件时，等待数据同步到持久存储。
	LockFileCreateWrite	在创建数据目录锁文件时等待写操作。
	LockFileReCheckDataDir Read	在重新检查数据目录锁文件期间等待读取。

等待事件类型	等待事件名称	描述
	LogicalRewriteCheckpointSync	在检查点期间等待逻辑重写映射同步到持久存储。
	LogicalRewriteMappingSync	在逻辑重写期间等待映射数据同步到持久存储。
	LogicalRewriteMappingWrite	在逻辑重写期间等待映射数据的写入。
	LogicalRewriteSync	等待逻辑重写映射同步到持久存储。
	LogicalRewriteTruncate	在逻辑重写期间等待截断映射数据。
	LogicalRewriteWrite	等待逻辑重写映射的写入。
	RelationMapRead	等待关系映射文件的读取。
	RelationMapSync	等待关系映射文件同步到持久存储。
	RelationMapWrite	等待对关系映射文件的写入。
	ReorderBufferRead	在重排序缓冲区管理期间等待读取。
	ReorderBufferWrite	在重排序缓冲区管理期间等待写入。
	ReorderLogicalMappingRead	在重排序缓冲区管理期间等待读取逻辑映射。
	ReplicationSlotRead	等待从复制槽控制文件读取。
	ReplicationSlotRestoreSync	在将复制槽控制文件恢复到内存期间，等待该文件同步到持久存储。
	ReplicationSlotSync	等待复制槽控制文件同步到持久存储。
	ReplicationSlotWrite	等待对复制槽控制文件的写入。
	SLRUFlushSync	在检查点或数据库关闭期间，等待 SLRU 数据同步到持久存储。
	SLRURead	等待读取 SLRU 页面。
	SLRUSync	在页面写入后等待 SLRU 数据同步到持久存储。
	SLRUWrite	等待 SLRU 页面的写入。
	SnapbuildRead	等待读取序列化的历史系统目录快照。
	SnapbuildSync	等待序列化的历史系统目录快照同步到持久存储。
	SnapbuildWrite	等待写入序列化的历史系统目录快照。
	TimelineHistoryFileSync	等待通过流复制接收的时间线历史文件同步到持久存储。
	TimelineHistoryFileWrite	等待通过流复制接收的时间线历史文件的写入。
	TimelineHistoryRead	等待读取时间线历史文件。
	TimelineHistorySync	等待新创建的时间线历史文件同步到持久存储。

等待事件类型	等待事件名称	描述
	TimelineHistoryWrite	等待写入新创建的时间线历史文件。
	TwophaseFileRead	等待读取两阶段状态文件。
	TwophaseFileSync	等待两阶段状态文件同步到持久存储。
	TwophaseFileWrite	等待对两阶段状态文件的写入。
	WALBootstrapSync	在引导过程中等待 WAL 同步到持久存储。
	WALBootstrapWrite	在引导过程中等待 WAL 页面的写入。
	WALCopyRead	通过复制一个已有 WAL 段来创建一个新的 WAL 段时等待读取。
	WALCopySync	等待通过复制现有 WAL 段创建的新 WAL 段同步到持久存储。
	WALCopyWrite	通过复制一个已有 WAL 段来创建一个新的 WAL 段时等待写入。
	WALInitSync	等待新初始化的 WAL 文件同步到持久存储。
	WALInitWrite	在初始化一个新的 WAL 文件时等待写入。
	WALRead	等待 WAL 文件的读取。
	WALSenderTimelineHistoryRead	在 WAL 发送进程执行时间线命令期间，等待读取时间线历史文件。
	WALSyncMethodAssign	在设置 WAL 同步方法时，等待数据同步到持久存储。
	WALWrite	等待写入 WAL 文件。

> **注意**
>
> 对于扩展注册的轻量级锁切片，名称由扩展指定，并显示为wait_event。用户可能只在某个后端中注册了该切片（通过在动态共享内存中分配），此时其他后端没有这项信息，因此在这种情况下显示extension。

以下示例展示如何查看等待事件：

```
SELECT pid, wait_event_type, wait_event FROM pg_stat_activity WHERE wait_event is NOT NULL;
 pid  | wait_event_type |  wait_event
------+-----------------+---------------
 2540 | Lock            | relation
 6644 | LWLock          | ProcArrayLock
(2 rows)
```

表 28.5. pg_stat_replication 视图

列	类型	描述
pid	integer	一个 WAL 发送进程的进程 ID
usesysid	oid	登录到这个 WAL 发送进程的用户的 OID

列	类型	描述
usename	name	登录到这个 WAL 发送进程的用户的名称
application_name	text	连接到这个 WAL 发送进程的应用的名称
client_addr	inet	连接到这个 WAL 发送进程的客户端的 IP 地址。如果这个字段为空值，它表示该客户端通过服务器机器上的一个 Unix 套接字连接。
client_hostname	text	已连接的客户端的主机名，由 client_addr 的反向 DNS 查找报告。这个字段将只对 IP 连接非空，并且只有log_hostname 被启用时才会非空。
client_port	integer	客户端用来与这个 WAL 发送进程通讯的 TCP 端口号，如果使用 Unix 套接字则为-1
backend_start	timestamp with time zone	这个进程开始的时间，即客户端是何时连接到这个 WAL 发送进程的。
backend_xmin	xid	由hot_standby_feedback 报告的该备库的xmin 视界。
state	text	当前的 WAL 发送进程状态。可能的值是： • startup：此 WAL 发送进程正在启动。 • catchup：与此 WAL 发送进程连接的备库正在赶上主库。 • streaming：在其连接的备库赶上主库之后，此 WAL 发送进程正在流式传输更改。 • backup：此 WAL 发送进程正在发送一个备份。 • stopping：此 WAL 发送进程正在停止。
sent_lsn	pg_lsn	在这个连接上发送的最后一个预写式日志的位置
write_lsn	pg_lsn	该备库已写入磁盘的最后一个预写式日志位置
flush_lsn	pg_lsn	该备库已刷入磁盘的最后一个预写式日志位置
replay_lsn	pg_lsn	已在该备库数据库中重放的最后一个预写式日志位置

列	类型	描述
write_lag	interval	从在本地将最近的 WAL 刷盘，到收到该备库已写入该 WAL（但尚未刷盘或应用）的通知之间经过的时间。如果该服务器被配置为同步备库，可用此值衡量在提交时synchronous_commit 级别remote_write 所导致的延迟。
flush_lag	interval	从在本地将最近的 WAL 刷盘，到收到该备库已写入该 WAL 并将其刷盘（但尚未应用）的通知之间经过的时间。如果该服务器被配置为同步备库，可用此值衡量在提交时synchronous_commit 级别on 所导致的延迟。
replay_lag	interval	从在本地将最近的 WAL 刷盘，到收到该备库已写入该 WAL、将其刷盘并应用的通知之间经过的时间。如果该服务器被配置为同步备库，可用此值衡量在提交时synchronous_commit 级别remote_apply 所导致的延迟。
sync_priority	integer	在基于优先级的同步复制中，该备库被选为同步备库的优先级。在基于法定人数的同步复制中，此值不起作用。
sync_state	text	该备库的同步状态。可能的值是： • async：该备库是异步的。 • potential：该备库当前是异步的，但在当前某个同步备库失效时可能变为同步的。 • sync：该备库是同步的。 • quorum：该备库被作为法定人数同步复制候选备库。

pg_stat_replication 视图为每个 WAL 发送进程包含一行，显示有关复制到该发送进程所连接备库的统计信息。这里只列出直接连接的备库；不包含下游备库的信息。

pg_stat_replication 视图中报告的滞后时间，是最近的 WAL 被写入、刷盘和重放，以及发送端获知这些操作完成所花费时间的度量。如果远程服务器被配置为同步备库，这些时间表示各个同步提交级别带来或可能带来的提交延迟。对于异步备库，replay_lag 列是最近事务对查询变得可见所需延迟时间的近似值。如果备库已经完全追上发送服务器且没有新的 WAL 活动，系统会在短时间内继续显示最近测得的滞后时间，随后显示为空值。

对于物理复制，滞后时间会自动测量。逻辑解码插件可以选择发出跟踪消息；如果没有这样做，跟踪机制会将滞后显示为空值。

> **注意**
>
> 报告出的滞后时间并不是根据当前重放速率预测该备库还需多久才能追上发送服务器。在持续生成新 WAL 时，那样的系统可能会显示相近的时间；但当发送端空闲时，就会显示

不同的值。特别是当备库已经完全追上时，pg_stat_replication 显示的是写入、刷盘和重放最近报告的 WAL 位置所花费的时间，而不是某些用户可能预期的零。这样做符合为最近写事务测量同步提交和事务可见性延迟的目的。为减少用户因预期另一种滞后模型而产生的混淆，在一个已完全重放且处于空闲状态的系统上，滞后列会在较短时间后恢复为空值。监控系统应自行决定将这种情况表示为缺失数据、零，还是继续显示最近一次已知值。

表 28.6. pg_stat_wal_receiver 视图

列	类型	描述
pid	integer	WAL 接收进程的进程 ID
status	text	WAL 接收进程的活动状态
receive_start_lsn	pg_lsn	WAL 接收进程启动时使用的第一个预写式日志位置
receive_start_tli	integer	WAL 接收进程启动时使用的第一个时间线编号
received_lsn	pg_lsn	已经接收并刷入到磁盘的最后一个预写式日志位置，该字段的初始值是启动 WAL 接收进程时使用的第一个日志位置
received_tli	integer	接收并刷入到磁盘的最后一个预写式日志位置的时间线编号，该字段的初始值为启动 WAL 接收进程时使用的第一个日志位置的时间线编号
last_msg_send_time	timestamp with time zone	从源 WAL 发送进程收到的最后一条消息的发送时间
last_msg_receipt_time	timestamp with time zone	从源 WAL 发送进程收到的最后一条消息的接收时间
latest_end_lsn	pg_lsn	向源 WAL 发送进程报告的最后一个预写式日志位置
latest_end_time	timestamp with time zone	向源 WAL 发送进程报告最后一个预写式日志位置的时间
slot_name	text	这个 WAL 接收进程使用的复制槽的名称
sender_host	text	这个 WAL 接收进程连接到的 PostgreSQL实例的主机。这可以是主机名、IP 地址，或者目录路径，如果连接是通过 Unix 套接字进行的。（路径的情况可以区分，因为它总是以/开头的绝对路径。）
sender_port	integer	这个 WAL 接收进程连接的PostgreSQL实例的端口号。
conninfo	text	这个 WAL 接收进程使用的连接字符串，对安全敏感的字段进行了模糊处理。

pg_stat_wal_receiver 视图只包含一行，显示 WAL 接收进程从其所连接服务器接收数据的统计信息。

表 28.7. pg_stat_subscription 视图

列	类型	描述
subid	oid	订阅的 OID
subname	text	订阅的名称
pid	integer	订阅工作进程的进程 ID
relid	Oid	该工作进程正在同步的关系的 OID；主应用工作进程则为空值
received_lsn	pg_lsn	接收到的最后一个预写式日志位置，该字段的初始值为 0
last_msg_send_time	timestamp with time zone	从源 WAL 发送进程收到的最后一条消息的发送时间
last_msg_receipt_time	timestamp with time zone	从源 WAL 发送进程收到的最后一条消息的接收时间
latest_end_lsn	pg_lsn	向源 WAL 发送进程报告的最后一个预写式日志位置
latest_end_time	timestamp with time zone	向源 WAL 发送进程报告最后一个预写式日志位置的时间

pg_stat_subscription 视图为每个订阅的主工作进程包含一行（如果该工作进程未运行，则 PID 为空），并为处理订阅表初始数据复制的工作进程包含额外的行。

表 28.8. pg_stat_ssl 视图

列	类型	描述
pid	integer	后端或 WAL 发送进程的进程 ID
ssl	boolean	如果在此连接上使用 SSL，则为真
version	text	使用 SSL 的版本，如果此连接上没有使用 SSL 则为空值
cipher	text	正在使用的 SSL 密码套件的名称，如果此连接上没有使用 SSL 则为空值
bits	integer	使用的加密算法中的位数，如果此连接上没有使用 SSL 则为空值
compression	boolean	使用 SSL 压缩时为真，否则为假；如果此连接未使用 SSL，则为空值
clientdn	text	所用客户端证书中的区别名称（DN，Distinguished Name）字段，如果没有提供客户端证书或在此连接上没有使用 SSL，则为空值。如果 DN 字段长于NAMEDATALEN（标准构建中为 64 个字符），则该字段将被截断。

pg_stat_ssl 视图将为每一个后端或者 WAL 发送进程包含一行，用来显示这个连接上的 SSL 使用情况。可以把它与pg_stat_activity 或者 pg_stat_replication 通过 pid 列连接来得到更多有关该连接的细节。

表 28.9. pg_stat_archiver 视图

列	类型	描述
archived_count	bigint	已成功归档的 WAL 文件数
last_archived_wal	text	最近成功归档的 WAL 文件的名称
last_archived_time	timestamp with time zone	最近成功归档操作的时间
failed_count	bigint	记录 WAL 文件归档失败次数
last_failed_wal	text	最近一次归档操作失败的 WAL 文件的名称
last_failed_time	timestamp with time zone	最近一次归档操作失败的时间
stats_reset	timestamp with time zone	这些统计信息上次被重置的时间

pg_stat_archiver 视图总是有一行，其中包含关于集簇的归档进程的数据。

表 28.10. pg_stat_bgwriter 视图

列	类型	描述
checkpoints_timed	bigint	已执行的计划检查点数
checkpoints_req	bigint	已执行的请求检查点数
checkpoint_write_time	double precision	检查点处理中将文件写入磁盘部分所花费的总时间，单位为毫秒
checkpoint_sync_time	double precision	检查点处理中将文件同步到磁盘部分所花费的总时间，单位为毫秒
buffers_checkpoint	bigint	检查点期间写入的缓冲区数
buffers_clean	bigint	后台写入器写入的缓冲区数量
maxwritten_clean	bigint	后台写入器因写入了过多缓冲区而停止清理扫描的次数
buffers_backend	bigint	后端直接写入的缓冲区数
buffers_backend_fsync	bigint	后端必须自行执行fsync 调用的次数（通常即使后端自行写入，也由后台写入器处理这些调用）
buffers_alloc	bigint	分配的缓冲区数量
stats_reset	timestamp with time zone	这些统计信息上次被重置的时间

pg_stat_bgwriter 视图始终只有一行，包含整个集簇的全局数据。

表 28.11. pg_stat_database 视图

列	类型	描述
datid	oid	数据库的 OID
datname	name	数据库的名称
numbackends	integer	当前连接到此数据库的后端数。这是该视图中唯一返回反映当前状态的值的列；所有其他列返回的都是自上次重置以来累积的值。
xact_commit	bigint	此数据库中已提交的事务数
xact_rollback	bigint	该数据库中已回滚的事务数

列	类型	描述
blks_read	bigint	在该数据库中读取的磁盘块数
blks_hit	bigint	在缓冲区缓存中发现磁盘块、因而无需读取的次数（这里只统计 PostgreSQL 缓冲区缓存中的命中，不包括操作系统文件系统缓存中的命中）
tup_returned	bigint	此数据库中的查询返回的行数
tup_fetched	bigint	此数据库中的查询获取的行数
tup_inserted	bigint	查询在该数据库中插入的行数
tup_updated	bigint	这个数据库中查询更新的行数
tup_deleted	bigint	这个数据库中被查询删除的行数
conflicts	bigint	由于与此数据库中的恢复冲突而被取消的查询数。（冲突只会发生在备库上；请参阅 pg_stat_database_conflicts。）
temp_files	bigint	这个数据库中查询创建的临时文件的数量。所有临时文件都将被计数，而不顾及临时文件为什么被创建（例如，排序或 hash 运算），也不考虑log_temp_files 设置。
temp_bytes	bigint	这个数据库中的查询写入临时文件的数据总量。所有临时文件都将被计数，而不考虑临时文件为什么被创建，也不考虑log_temp_files 设置。
deadlocks	bigint	在此数据库中检测到的死锁数
blk_read_time	double precision	此数据库的后端读取数据文件块所花费的时间，单位为毫秒
blk_write_time	double precision	此数据库的后端写入数据文件块所花费的时间，单位为毫秒
stats_reset	timestamp with time zone	这些统计数据最后一次重置的时间

pg_stat_database 视图为集簇中的每个数据库包含一行，显示整个数据库的统计信息。

表 28.12. pg_stat_database_conflicts 视图

列	类型	描述
datid	oid	数据库的 OID
datname	name	数据库的名称
confl_tablespace	bigint	这个数据库中由于删除表空间而取消的查询的数量
confl_lock	bigint	此数据库中由于锁定超时而被取消的查询数
confl_snapshot	bigint	此数据库中由于旧快照而取消的查询数
confl_bufferpin	bigint	此数据库中由于缓冲区被钉住而被取消的查询数

列	类型	描述
confl_deadlock	bigint	此数据库中由于死锁而被取消的查询数

pg_stat_database_conflicts 视图为每个数据库包含一行，显示备库上因与恢复冲突而取消查询的数据库范围统计信息。由于主库上不会发生这些冲突，该视图只会在备库上包含信息。

表 28.13. pg_stat_all_tables 视图

列	类型	描述
relid	oid	表的 OID
schemaname	name	该表所在的模式的名称
relname	name	这个表的名称
seq_scan	bigint	在此表上启动的顺序扫描数
seq_tup_read	bigint	顺序扫描获取的存活行数
idx_scan	bigint	对这个表发起的索引扫描数
idx_tup_fetch	bigint	索引扫描获取的存活行数
n_tup_ins	bigint	插入的行数
n_tup_upd	bigint	更新的行数（包括HOT 更新的行）
n_tup_del	bigint	删除的行数
n_tup_hot_upd	bigint	HOT 更新的行数（即不需要单独更新索引）
n_live_tup	bigint	活的行的估计数量
n_dead_tup	bigint	死行的估计数量
n_mod_since_analyze	bigint	自上次分析此表以来修改的行的估计数量
last_vacuum	timestamp with time zone	最后一次手动清理这个表的时间（不包括VACUUM FULL）
last_autovacuum	timestamp with time zone	这个表最后一次被自动清理守护进程清理的时间
last_analyze	timestamp with time zone	上一次手动分析这个表的时间
last_autoanalyze	timestamp with time zone	自动清理守护进程最后一次分析这个表的时间
vacuum_count	bigint	这个表被手动清理的次数（VACUUM FULL 不计数）
autovacuum_count	bigint	这个表被自动清理守护进程清理的次数
analyze_count	bigint	手动分析这个表的次数
autoanalyze_count	bigint	这个表被自动清理守护进程分析的次数

pg_stat_all_tables 视图将为当前数据库中的每一个表（包括 TOAST 表）包含一行，该行显示与对该表的访问相关的统计信息。pg_stat_user_tables 和 pg_stat_sys_tables 视图包含相同的信息，但是被过滤得分别只显示用户和系统表。

表 28.14. pg_stat_all_indexes 视图

列	类型	描述
relid	oid	对于此索引的表的 OID
indexrelid	oid	这个索引的 OID
schemaname	name	这个索引所在的模式名称
relname	name	这个索引的表的名称
indexrelname	name	这个索引的名称
idx_scan	bigint	在这个索引上开启的索引扫描的数量
idx_tup_read	bigint	扫描此索引返回的索引项数
idx_tup_fetch	bigint	使用此索引进行简单索引扫描获取的存活表行数

pg_stat_all_indexes 视图将为当前数据库中的每个索引包含一行，该行显示关于对该索引访问的统计信息。pg_stat_user_indexes 和 pg_stat_sys_indexes 视图包含相同的信息，但是被过滤得只分别显示用户和系统索引。

索引可以被简单索引扫描、“位图”索引扫描以及优化器使用。在一次位图扫描中，多个索引的输出可以被通过 AND 或 OR 规则组合，因此当使用一次位图扫描时难以将取得的个体堆行与特定的索引关联起来。因此，一次位图扫描会增加它使用的索引的pg_stat_all_indexes.idx_tup_read 计数，并且为该表增加pg_stat_all_tables.idx_tup_fetch 计数，但是它不影响pg_stat_all_indexes.idx_tup_fetch。如果所提供的常量值不在优化器统计信息记录的范围之内，优化器也会访问索引来检查，因为优化器统计信息可能已经过时。

> **注意**
>
> 即使不用位图扫描，idx_tup_read 和 idx_tup_fetch 计数也可能不同，因为 idx_tup_read 统计从该索引取得的索引项而 idx_tup_fetch 统计从表取得的存活行。如果使用该索引取得了任何死亡行或还未提交的行，或者如果通过一次仅索引扫描的方式避免了任何堆获取，后者将较小。

表 28.15. pg_statio_all_tables 视图

列	类型	描述
relid	oid	表的 OID
schemaname	name	该表所在的模式的名称
relname	name	这个表的名称
heap_blks_read	bigint	从该表中读取的磁盘块的数量
heap_blks_hit	bigint	该表中的缓冲区命中数
idx_blks_read	bigint	从这个表上所有索引读取的磁盘块数
idx_blks_hit	bigint	这个表上所有索引中的缓冲区命中数
toast_blks_read	bigint	从这个表的 TOAST 表中读取的磁盘块的数量（如果有的话）

列	类型	描述
toast_blks_hit	bigint	这个表的 TOAST 表中的缓冲区命中数（如果有的话）
tidx_blks_read	bigint	从这个表的 TOAST 表索引中读取的磁盘块的数量（如果有的话）
tidx_blks_hit	bigint	这个表的 TOAST 表索引中的缓冲区命中数（如果有的话）

pg_statio_all_tables 视图将为当前数据库中的每个表（包括 TOAST 表）包含一行，该行显示指定表上有关 I/O 的统计信息。pg_statio_user_tables 和 pg_statio_sys_tables 视图包含相同的信息，但是被过滤得分别只显示用户表和系统表。

表 28.16. pg_statio_all_indexes 视图

列	类型	描述
relid	oid	对于此索引的表的 OID
indexrelid	oid	这个索引的 OID
schemaname	name	这个索引所在的模式名称
relname	name	这个索引的表的名称
indexrelname	name	这个索引的名称
idx_blks_read	bigint	从此索引中读取的磁盘块的数量
idx_blks_hit	bigint	此索引中的缓冲区命中数

pg_statio_all_indexes 视图将为当前数据库中的每个索引包含一行，该行显示指定索引上有关 I/O 的统计信息。pg_statio_user_indexes 和 pg_statio_sys_indexes 视图包含相同的信息，但是被过滤得分别只显示用户索引和系统索引。

表 28.17. pg_statio_all_sequences 视图

列	类型	描述
relid	oid	序列的 OID
schemaname	name	此序列所在的模式的名称
relname	name	此序列的名称
blks_read	bigint	从这个序列中读取的磁盘块的数量
blks_hit	bigint	在此序列中的缓冲区命中数

pg_statio_all_sequences 视图将为当前数据库中的每个序列包含一行，该行显示在指定序列上有关 I/O 的统计信息。

表 28.18. pg_stat_user_functions 视图

列	类型	描述
funcid	oid	函数的 OID
schemaname	name	这个函数所在的模式的名称
funcname	name	这个函数的名称
calls	bigint	这个函数已经被调用的次数
total_time	double precision	在这个函数以及它所调用的其他函数中花费的总时间，以毫秒计

列	类型	描述
self_time	double precision	在这个函数本身花费的总时间，不包括被它调用的其他函数，以毫秒计

pg_stat_user_functions 视图将为每一个被追踪的函数包含一行，该行显示有关该函数执行的统计信息。track_functions 参数控制到底哪些函数被跟踪。

28.2.3. 统计函数

其他查看统计信息的方法是直接使用查询，这些查询使用上述标准视图用到的底层统计信息访问函数。如要了解如函数名等细节，可参考标准视图的定义（例如，在psql 中你可以发出\d+ pg_stat_activity）。针对每一个数据库统计信息的访问函数把一个数据库 OID 作为参数来标识要报告哪个数据库。而针对每个表和每个索引的函数要求表或索引 OID。针对每个函数统计信息的函数用一个函数 OID。注意只有在当前数据库中的表、索引和函数才能被这些函数看到。

与统计收集器相关的其他功能在表 28.19 中列出。

表 28.19. 附加统计函数

函数	返回类型	描述
pg_backend_pid()	integer	处理当前会话的服务器进程的进程 ID
pg_stat_get_activity(integer)	setof record	返回一条记录，包含指定 PID 的后端的信息；如果指定NULL，则为系统中的每个活动后端返回一条记录。返回字段是pg_stat_activity 视图字段的子集。
pg_stat_get_snapshot_ timestamp()	timestamp with time zone	返回当前统计快照的时间戳。
pg_stat_clear_snapshot()	void	丢弃当前统计信息快照
pg_stat_get_xact_blocks_ fetched(oid)	bigint	返回当前事务中表或索引的块读取请求次数。这个数字减去 pg_stat_get_xact_blocks_hit 给出了内核read() 调用的次数；实际的物理读取次数通常较低，这是由于内核级别的缓冲。
pg_stat_get_xact_blocks_ hit(oid)	bigint	返回在当前事务中在缓存中找到的表或索引的块读取请求次数（不触发内核read() 调用）。
pg_stat_reset()	void	将当前数据库的所有统计计数器重置为零（默认需要超级用户权限，但可以向其他用户授予此函数的 EXECUTE 权限。）

函数	返回类型	描述
pg_stat_reset_shared(text)	void	根据参数将某些集簇范围的统计计数器重置为零（默认需要超级用户权限，但可以向其他用户授予此函数的 EXECUTE 权限）。调用pg_ stat_reset_shared('bgwrit er') 会将pg_stat_bgwriter 视图中显示的所有计数器清零。调用pg_stat_reset_shared('archiver') 会将pg_stat_ archiver 视图中显示的所有计数器清零。
pg_stat_reset_single_ table_counters(oid)	void	将当前数据库中单个表或索引的统计信息重置为零（默认需要超级用户权限，但可以向其他用户授予此函数的 EXECUTE 权限）
pg_stat_reset_single_ function_counters(oid)	void	将当前数据库中单个函数的统计信息重置为零（默认需要超级用户权限，但可以向其他用户授予此函数的 EXECUTE 权限）

警告

使用 pg_stat_reset() 还会重置自动清理使用的计数器，以确定何时触发清理或分析。重置这些计数器可能导致自动清理不执行必要的工作，这可能会导致问题，如表膨胀或过时的表统计信息。建议在统计信息重置后进行全库ANALYZE。

pg_stat_get_activity 是 pg_stat_activity 视图的底层函数，它返回一个行集合，其中包含有关每个后端进程所有可用的信息。有时只获得该信息的一个子集可能会更方便。在那些情况中，可以使用一组更老的针对每个后端的统计访问函数，这些显示在表 28.20 中。这些访问函数使用一个后端 ID 号，范围从 1 到当前活动后端数目。函数pg_stat_get_backend_ idset 提供了一种方便的方法为每个活动后端产生一行来调用这些函数。例如，要显示PID 以及所有后端当前的查询：

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

表 28.20. 按后端统计函数

函数	返回类型	描述
pg_stat_get_backend_ idset()	setof integer	当前活动后端的 ID 编号集合（从 1 到活动后端数）
pg_stat_get_backend_ activity(integer)	text	此后端最近查询的文本
pg_stat_get_backend_ activity_start(integer)	timestamp with time zone	最近查询的开始时间
pg_stat_get_backend_ client_addr(integer)	inet	连接到此后端的客户端的 IP 地址

函数	返回类型	描述
pg_stat_get_backend_client_port(integer)	integer	客户端用于通信的 TCP 端口号
pg_stat_get_backend_dbid(integer)	oid	这个后端连接到的数据库的 OID
pg_stat_get_backend_pid(integer)	integer	这个后端的进程 ID
pg_stat_get_backend_start(integer)	timestamp with time zone	此进程的启动时间
pg_stat_get_backend_userid(integer)	oid	登录到这个后端的用户的 OID
pg_stat_get_backend_wait_event_type(integer)	text	如果后端当前正在等待，则为等待事件类型名称，否则为 NULL。详见表 28.4。
pg_stat_get_backend_wait_event(integer)	text	如果后端当前正在等待，则为等待事件名称，否则为 NULL。详见表 28.4。
pg_stat_get_backend_xact_start(integer)	timestamp with time zone	当前事务的开始时间

28.3. 查看锁

监控数据库活动的另外一个有用的工具是 pg_locks 系统表。这样就允许数据库管理员查看锁管理器中当前存在的锁的信息。例如，这个功能可以被用于：

- 查看当前存在的所有锁、在一个特定数据库中的关系上所有的锁、在一个特定关系上所有的锁，或者由一个特定PostgreSQL 会话持有的所有的锁。
- 判断当前数据库中带有最多未授予锁的关系（它可能是数据库客户端之间的竞争来源）。
- 判断锁竞争给数据库总体性能带来的影响，以及锁竞争随着整个数据库流量的变化范围。

pg_locks 视图的细节在第 52.73 节中。更多有关 PostgreSQL 的锁和管理并发性的信息，请参考第 13 章。

28.4. 进度报告

PostgreSQL 能够在某些命令执行期间报告其进度。目前，支持进度报告的命令只有VACUUM。未来可能会扩展此功能。

28.4.1. VACUUM 进度报告

只要 VACUUM 正在运行，pg_stat_progress_vacuum 视图就会为每个当前正在清理的后端（包括自动清理工作进程）包含一行。下表描述了将报告的信息，并说明如何解释这些信息。目前不支持报告VACUUM FULL 的进度，运行VACUUM FULL 的后端不会列在此视图中。

表 28.21. pg_stat_progress_vacuum 视图

列	类型	描述
pid	integer	后端的进程 ID。
datid	oid	后端连接到的数据库的 OID。
datname	name	后端连接到的数据库的名称。

列	类型	描述
relid	oid	正在清理的表的 OID。
phase	text	清理的当前处理阶段。参见表 28.22。
heap_blks_total	bigint	该表中堆块的总数。这个数字以扫描开始时的数量为准，之后增加的块将不会（并且不需要）被这个 VACUUM 访问。
heap_blks_scanned	bigint	被扫描的堆块数量。由于可见性映射被用来优化扫描，一些块将被跳过而不做检查，被跳过的块会被包括在这个总数中，因此当清理完成时这个数字最终将会等于heap_blks_total。仅当处于scanning heap 阶段时这个计数器才会前进。
heap_blks_vacuumed	bigint	被清理的堆块数量。除非表没有索引，这个计数器仅在处于vacuuming heap 阶段时才会前进。不包含死亡元组的块会被跳过，因此这个计数器可能有时会向前跳跃一个比较大的增量。
index_vacuum_count	bigint	已完成的索引清理周期数。
max_dead_tuples	bigint	在需要执行索引清理周期之前可存储的死亡元组数量，取决于maintenance_work_mem。
num_dead_tuples	bigint	自上一个索引清理周期以来收集到的死亡元组数量。

表 28.22. VACUUM 阶段

阶段	描述
initializing	VACUUM 正在准备开始扫描堆。这个阶段应该很简短。
scanning heap	VACUUM 正在扫描堆。如果需要，它将会对每个页面进行剪枝以及碎片整理，并且可能会执行冻结动作。heap_blks_scanned 列可以用来监控扫描的进度。
vacuuming indexes	VACUUM 当前正在清理索引。如果表有索引，每次清理都会在堆扫描完成后至少执行一次此阶段。如果maintenance_work_mem（或者，对于自动清理，已设置的 autovacuum_work_mem）不足以存储找到的死亡元组数量，则每次清理可能多次执行此阶段。
vacuuming heap	VACUUM 当前正在清理堆。清理堆与扫描堆不是同一个概念，清理堆发生在每次索引清理之后。如果heap_blks_scanned 小于heap_blks_total，系统将在这个阶段完成之后回去扫描堆；否则，系统将在这个阶段完成后开始索引收尾清理。
cleaning up indexes	VACUUM 当前正在进行索引收尾清理。这个阶段发生在堆被完全扫描并且对堆和索引的所有清理都已经完成以后。

阶段	描述
truncating heap	VACUUM 正在截断堆，以便把关系尾部的空页面返还给操作系统。这个阶段发生在索引收尾清理完成之后。
performing final cleanup	VACUUM 正在执行最终清理。在此阶段，VACUUM 将清理空闲空间映射，更新pg_class 中的统计信息，并向统计收集器报告统计信息。当此阶段完成时，VACUUM 将结束。

28.5. 动态追踪

PostgreSQL 提供了功能来支持数据库服务器的动态追踪。这样就允许在代码中的特定点上调用外部工具来追踪执行过程。

一些探针或追踪点已经被插入在源代码中。这些探针的目的是被数据库开发者和管理员使用。默认情况下，探针不被编译到PostgreSQL 中；用户需要显式地告诉配置脚本使得探针可用。

目前，在写本文当时DTrace[1] 已被支持，它在 Solaris、macOS、FreeBSD、NetBSD 和 Oracle Linux 上可用。Linux 的SystemTap[2] 项目提供了一种可用的 DTrace 等价物。支持其他动态追踪工具在理论上可以通过改变 src/include/utils/probes.h 中的宏定义实现。

28.5.1. 为动态追踪编译

默认情况下，探针是不可用的，因此你将需要显式地告诉配置脚本让探针在PostgreSQL 中可用。要包括 DTrace 支持，在配置时指定--enable-dtrace。更多信息请见第 16.4 节。

28.5.2. 内置探针

如表 28.23 所示，源代码中提供了一些标准探针。表 28.24 显示了在探针中使用的类型。当然，可以增加更多探针来增强PostgreSQL 的可观测性。

表 28.23. 内置 DTrace 探针

名称	参数	描述
transaction-start	(LocalTransactionId)	在一个新事务开始时触发的探针。arg0 是事务 ID。
transaction-commit	(LocalTransactionId)	在一个事务成功完成时触发的探针。arg0 是事务 ID。
transaction-abort	(LocalTransactionId)	当一个事务失败结束时触发的探针。arg0 是事务 ID。
query-start	(const char *)	当一个查询的处理开始时触发的探针。arg0 是查询字符串。
query-done	(const char *)	当一个查询的处理完成时触发的探针。arg0 是查询字符串。
query-parse-start	(const char *)	当一个查询的解析开始时触发的探针。arg0 是查询字符串。
query-parse-done	(const char *)	当一个查询的解析完成时触发的探针。arg0 是查询字符串。
query-rewrite-start	(const char *)	当一个查询的重写开始时触发的探针。arg0 是查询字符串。

[1] https://en.wikipedia.org/wiki/DTrace
[2] https://sourceware.org/systemtap/

名称	参数	描述
query-rewrite-done	(const char *)	当一个查询的重写完成时触发的探针。arg0 是查询字符串。
query-plan-start	()	当一个查询的规划开始时触发的探针。
query-plan-done	()	当一个查询的规划完成时触发的探针。
query-execute-start	()	当一个查询的执行开始时触发的探针。
query-execute-done	()	当一个查询的执行完成时触发的探针。
statement-status	(const char *)	任何时候当服务器进程更新它的 pg_stat_activity.status 时触发的探针。arg0 是新的状态字符串。
checkpoint-start	(int)	当一个检查点开始时触发的探针。arg0 传递位标志来区分不同的检查点类型，例如关闭（shutdown）、立即（immediate）或强制（force）。
checkpoint-done	(int, int, int, int, int)	当一个检查点完成时触发的探针（下面列出的探针会在检查点处理过程中依次触发）。arg0 是已写入的缓冲区数量。arg1 是缓冲区的总数。arg2、arg3 和 arg4 分别包含了增加、删除和循环回收的 WAL 文件的数量。
clog-checkpoint-start	(bool)	当一个检查点的 CLOG 部分开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
clog-checkpoint-done	(bool)	当一个检查点的 CLOG 部分完成时触发的探针。arg0 的含义与 clog-checkpoint-start 中相同。
subtrans-checkpoint-start	(bool)	当一个检查点的 SUBTRANS 部分开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
subtrans-checkpoint-done	(bool)	当一个检查点的 SUBTRANS 部分完成时触发的探针。arg0 的含义与subtrans-checkpoint-start 中相同。
multixact-checkpoint-start	(bool)	当一个检查点的 MultiXact 部分开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
multixact-checkpoint-done	(bool)	当一个检查点的 MultiXact 部分完成时触发的探针。arg0 的含义与multixact-checkpoint-start 中相同。

名称	参数	描述
buffer-checkpoint-start	(int)	当一个检查点的写缓冲区部分开始时触发的探针。arg0 传递位标志来区分不同的检查点类型，例如关闭（shutdown）、立即（immediate）或强制（force）。
buffer-sync-start	(int, int)	当我们在检查点期间开始写脏缓冲区时（在标识哪些缓冲区必须被写之后）触发的探针。arg0 是缓冲区总数，arg1 是当前为脏并且需要被写的缓冲区数量。
buffer-sync-written	(int)	在检查点期间当每个缓冲区被写完之后触发的探针。arg0 是缓冲区的 ID。
buffer-sync-done	(int, int, int)	当所有脏缓冲区被写之后触发的探针。arg0 是缓冲区总数。arg1 是检查点进程实际写的缓冲区数量。arg2 是期望写的数目（buffer-sync-start 的 arg1）；arg1 和 arg2 的任何的不同反映在该检查点期间有其他进程刷写了缓冲区。
buffer-checkpoint-sync-start	()	在脏缓冲区被写入到内核之后并且在开始发出 fsync 请求之前触发的探针。
buffer-checkpoint-done	()	当同步缓冲区到磁盘完成时触发的探针。
twophase-checkpoint-start	()	当一个检查点的两阶段部分开始时触发的探针。
twophase-checkpoint-done	()	当一个检查点的两阶段部分完成时触发的探针。
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	当一次缓冲区读开始时触发的探针。arg0 和 arg1 包含该页的分支号和块号（如果这是一次关系扩展请求，arg1 为 -1）。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是 InvalidBackendId（-1）。arg6 为真表示一次关系扩展请求，为假表示正常读。

名称	参数	描述
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)	当一次缓冲区读完成时触发的探针。arg0 和 arg1 包含该页的分支号和块号（如果这是一次关系扩展请求，arg1 现在包含新增加块的块号）。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是 InvalidBackendId（-1）。arg6 为真表示一次关系扩展请求，为假表示正常读。arg7 为真表示在池中找到该缓冲区，为假表示没有找到。
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	在发出对一个共享缓冲区的任意写请求之前触发的探针。arg0 和 arg1 包含该页的分支号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一个写请求完成时触发的探针（注意这只反映传递数据给内核的时间，它通常并没有实际地被写入到磁盘）。参数和buffer-flush-start 的相同。
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一个服务器进程开始写一个脏缓冲区时触发的探针（如果这经常发生，表示shared_buffers 太小，或需要调整后台写入器的控制参数）。arg0 和 arg1 包含该页的分支号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一次脏缓冲区写完成时触发的探针。参数与buffer-write-dirty-start 相同。
wal-buffer-write-dirty-start	()	当一个服务器进程因为没有可用 WAL 缓冲区空间开始写一个脏 WAL 缓冲区时触发的探针（如果这经常发生，表示wal_buffers 太小）。
wal-buffer-write-dirty-done	()	当一次脏 WAL 缓冲区写入完成时触发的探针。
wal-insert	(unsigned char, unsigned char)	当一个 WAL 记录被插入时触发的探针。arg0 是该记录的资源管理器（rmid）。arg1 包含 info 标志。
wal-switch	()	当请求一次 WAL 段切换时触发的探针。

名称	参数	描述
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当开始从一个关系读取一块时触发的探针。arg0 和 arg1 包含该页的分支号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一次块读取完成时触发的探针。arg0 和 arg1 包含该页的分支号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。arg6 是实际读取的字节数，而 arg7 是请求读取的字节数（如果两者不同就意味着麻烦）。
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当开始向一个关系中写入一个块时触发的探针。arg0 和 arg1 包含该页的分支号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。
smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一个块写操作完成时触发的探针。arg0 和 arg1 包含该页的分支号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 来标识该关系。对于一个本地缓冲区，arg5 是创建临时关系的后端 ID；对于一个共享缓冲区，arg5 是InvalidBackendId（-1）。arg6 是实际写的字节数，而 arg7 是要求写的字节数（如果这两者不同，则意味着麻烦）。
sort-start	(int, bool, int, int, bool, int)	当一次排序操作开始时触发的探针。arg0 指示是堆排序、索引排序或 datum 排序。arg1 为真表示强制唯一值。arg2 是键列数。arg3 是允许使用的工作内存大小（以千字节计）。如果要求随机访问排序结果，则 arg4 为真。arg5 为 0 时表示串行，为 1 时表示并行工作进程，为 2 时表示并行领导者。
sort-done	(bool, long)	当一次排序完成时触发的探针。arg0 为真表示外部排序，为假表示内排序。arg1 是用于一次外部排序的磁盘块的数目，或用于一次内排序的以千字节计的内存。

名称	参数	描述
lwlock-acquire	(char *, LWLockMode)	当成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。arg1 是所请求的锁模式，是排他或共享。
lwlock-release	(char *)	当一个 LWLock 被释放时（但是注意还没有唤醒任何一个被释放的等待者）触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。
lwlock-wait-start	(char *, LWLockMode)	当一个 LWLock 不是当即可用并且一个服务器进程因此开始等待该锁变为可用时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。arg1 是所请求的锁模式，是排他或共享。
lwlock-wait-done	(char *, LWLockMode)	当一个进程从对一个 LWLock 的等待中被释放（它实际还没有得到该锁）时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。arg1 是所请求的锁模式，是排他或共享。
lwlock-condacquire	(char *, LWLockMode)	当调用者指定无需等待而成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。arg1 是所请求的锁模式，是排他或共享。
lwlock-condacquire-fail	(char *, LWLockMode)	当调用者指定无需等待而没有成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。arg1 是所请求的锁模式，是排他或共享。
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr 锁）的请求由于锁不可用开始等待时触发的探针。arg0 到 arg3 是标识被锁定对象的标签字段。arg4 指示被锁对象的类型。arg5 表示被请求的锁类型。
lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr 锁）的请求结束等待时（即已经得到锁）触发的探针。参数与lock-wait-start 一样。
deadlock-found	()	当死锁检测器发现死锁时触发的探针。

表 28.24. 探针参数中使用的已定义类型

类型	定义
LocalTransactionId	unsigned int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int

类型	定义
Oid	unsigned int
ForkNumber	int
bool	char

28.5.3. 使用探针

下面的示例展示了一个分析系统中事务计数的 DTrace 脚本，可以用来代替一次性能测试之前和之后的pg_stat_database 快照：

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

当被执行时，该示例 D 脚本给出这样的输出：

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                        2312105013
```

> **注意**
>
> SystemTap 的追踪脚本记法与 DTrace 不同，但底层探针是兼容的。需要注意的一点是，截至本文编写时，SystemTap 脚本必须使用双下划线来代替连字符引用探针名。预计未来的 SystemTap 版本会修复这一点。

你应该记住，DTrace 脚本需要细心地编写和调试，否则被收集的追踪信息可能会毫无意义。在大多数发现问题的情况下，出错的是插桩，而不是底层系统。当讨论使用动态追踪发现的信息时，一定要附上使用的脚本以便其也被检查和讨论。

28.5.4. 定义新探针

开发者可以在代码中任意位置定义新的探针，当然这要重新编译之后才能生效。下面是插入新探针的步骤：

1. 决定探针名称以及要通过探针提供的数据
2. 把该探针定义加入到 src/backend/utils/probes.d
3. 如果 pg_trace.h 尚未被包含该探针点的模块引用，则将它包含进来，并且在源代码中期望的位置插入TRACE_POSTGRESQL 探针宏
4. 重新编译并验证新探针是可用的

示例：. 这里是一个如何增加一个探针来用事务 ID 追踪所有新事务的示例。

1. 决定探针将被命名为 transaction-start 并且需要一个 LocalTransactionId 类型的参数
2. 将该探针定义加入到 src/backend/utils/probes.d：

   ```
   probe transaction__start(LocalTransactionId);
   ```

   注意探针名字中双下划线的使用。在一个使用探针的 DTrace 脚本中，双下划线需要被替换为一个连字符，因此，文档中应向用户说明的名称是transaction-start。
3. 在编译时，transaction__start 被转换成一个名为 TRACE_POSTGRESQL_TRANSACTION_START 的宏（注意这里是单下划线），可以通过包含头文件pg_trace.h 获得。将宏调用加入到源代码中的合适位置。在这种情况下，看起来类似：

   ```
   TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
   ```
4. 在重新编译和运行新的二进制文件之后，通过运行下面的 DTrace 命令来检查新增的探针是否可用。你应该看到类似下面的输出：

   ```
   # dtrace -ln transaction-start
      ID    PROVIDER          MODULE           FUNCTION NAME
   18705 postgresql49878     postgres     StartTransactionCommand transaction-start
   18755 postgresql49877     postgres     StartTransactionCommand transaction-start
   18805 postgresql49876     postgres     StartTransactionCommand transaction-start
   18855 postgresql49875     postgres     StartTransactionCommand transaction-start
   18986 postgresql49873     postgres     StartTransactionCommand transaction-start
   ```

向 C 代码中添加追踪宏时，有一些事情需要注意：

- 需要小心的是，为探针参数指定的数据类型要匹配宏中使用的变量的数据类型，否则会发生编译错误。

- 在大多数平台上，如果用`--enable-dtrace`编译了 PostgreSQL，无论何时当控制经过一个追踪宏时，都会对该宏的参数求值，即使没有进行追踪也会这样做。如果只是报告少数局部变量的值，通常无需担心这一点。但是要注意不要将开销大的函数调用放入参数中。如果你需要这样做，考虑通过检查追踪是否真的被启用来保护该宏：

  ```
  if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
      TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
  ```

  每个追踪宏有一个对应的`ENABLED`宏。

第 29 章 监控磁盘使用情况

本章讨论如何监控 PostgreSQL 数据库系统的磁盘使用情况。

29.1. 确定磁盘使用情况

每个表都有一个主堆磁盘文件，大部分数据都存储在那里。如果表中有任何列可能包含较宽的值，还可能有一个与该表关联的 TOAST 文件，用于存储那些不适合放在主表中的过宽值（参见第 69.2 节）。如果该 TOAST 表存在，则它会有一个有效索引。基础表本身也可能带有索引。每个表和索引都存储在单独的磁盘文件中；如果某个文件会超过 1 GB，则可能拆分为多个文件。关于这些文件的命名规则，请参见第 69.1 节。

你可以通过三种方式监控磁盘空间：使用表 9.84 中列出的 SQL 函数、使用oid2name 模块，或者手工检查系统目录。SQL 函数最容易使用，通常也是推荐的方法。本节余下内容展示如何通过检查系统目录来完成这项工作。

在最近进行过清理或分析的数据库上使用 psql，你可以发出查询来查看任意表的磁盘使用情况：

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

每一页通常是 8 KB。（请记住，`relpages` 只会由 `VACUUM`、`ANALYZE` 以及少数 DDL 命令如 `CREATE INDEX` 更新。）如果你想直接检查该表的磁盘文件，那么文件路径名会很有用。

要显示 TOAST 表使用的空间，可以使用如下查询：

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

也可以轻松显示索引大小：

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname         | relpages
-----------------------+----------
 customer_id_indexdex |       26
```

利用这些信息，你可以很容易地找出最大的表和索引：

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname         | relpages
-----------------------+----------
 bigtable              |     3290
 customer              |     3144
```

29.2. 磁盘已满故障

数据库管理员最重要的磁盘监控任务，是确保磁盘不会被写满。数据磁盘写满不会导致数据损坏，但可能会阻止有用的活动继续进行。如果保存 WAL 文件的磁盘被写满，则数据库服务器可能会 panic 并随后关闭。

如果你无法通过删除其他文件来释放更多磁盘空间，那么可以利用表空间把部分数据库文件移动到其他文件系统上。详见第 22.6 节。

> **提示**
>
> 有些文件系统在接近写满时性能会明显变差，因此不要等到磁盘完全写满才采取措施。

如果你的系统支持按用户设置磁盘配额，那么数据库自然也会受到运行服务器的那个用户所拥有配额的限制。超出配额会带来与磁盘空间完全耗尽相同的不良影响。

第 30 章 可靠性与预写式日志

本章说明如何使用预写式日志，实现高效、可靠的运行。

30.1. 可靠性

可靠性是任何严肃数据库系统的重要属性，而 PostgreSQL 会尽一切可能保证运行可靠。可靠运行的一个方面是，已提交事务所记录的所有数据都应存储在不受断电、操作系统故障和硬件故障影响的非易失性区域中（当然不包括该非易失性区域本身的故障）。通常，只要能成功将数据写入计算机的永久存储设备（磁盘驱动器或等效设备），就满足这一要求。实际上，即使一台计算机遭受致命损坏，只要磁盘驱动器得以幸存，就可以将它们移到另一台硬件相近的计算机上，所有已提交事务仍将完好无损。

虽然周期性地强制将数据写到磁盘盘片似乎是个简单操作，但事实并非如此。由于磁盘驱动器远慢于主存和 CPU，在计算机主存与磁盘盘片之间存在多层缓存。首先是操作系统的缓冲区缓存，它会缓存常用磁盘块并合并磁盘写入。幸运的是，所有操作系统都为应用提供了强制将数据从缓冲区缓存写入磁盘的方法，而 PostgreSQL 会使用这些机制。（关于如何调整这一过程，见参数 wal_sync_method。）

其次，磁盘驱动器控制器中可能还有缓存；这在 RAID 控制卡上尤为常见。其中一些缓存是直写式，也就是说写请求一到达就立刻发送到驱动器。另一些是回写式，也就是说数据会在稍后的某个时间发送到驱动器。这类缓存可能构成可靠性隐患，因为磁盘控制器缓存中的内存是易失性的，断电时会丢失内容。较好的控制卡配有后备电池单元（BBU），也就是卡上带有电池，可在系统断电时维持缓存供电。电力恢复后，数据便会写入磁盘驱动器。

最后，大多数磁盘驱动器自身也带有缓存。有些是直写式，有些是回写式；对于回写式驱动器缓存，同样存在与磁盘控制器缓存相同的数据丢失隐患。消费级 IDE 和 SATA 驱动器尤其可能带有在断电后无法保留内容的回写式缓存。很多固态驱动器（SSD）也带有易失性的回写式缓存。

这些缓存通常都可以禁用；不过，具体方法会因操作系统和驱动器类型而异：

- 在 Linux 上，可以使用 `hdparm -I` 查询 IDE 和 SATA 驱动器；如果 `Write cache` 旁边有一个 `*`，就表示启用了写缓存。可以使用 `hdparm -W 0` 关闭写缓存。SCSI 驱动器可以使用 sdparm[1] 查询。使用 `sdparm --get=WCE` 检查是否启用了写缓存，使用 `sdparm --clear=WCE` 将其禁用。

- 在 FreeBSD 上，可以使用 `atacontrol` 查询 IDE 驱动器，并通过在 `/boot/loader.conf` 中设置 `hw.ata.wc=0` 来关闭写缓存；SCSI 驱动器也可以使用 `camcontrol identify` 查询，并在可用时借助 `sdparm` 查询和修改写缓存。

- 在 Solaris 上，磁盘写缓存由 `format -e` 控制。（Solaris 的 ZFS 文件系统在启用磁盘写缓存时仍然是安全的，因为它会发出自己的磁盘缓存刷新命令。）

- 在 Windows 上，如果 `wal_sync_method` 是 `open_datasync`（默认值），可以通过取消勾选 `My Computer\Open\`*`disk drive`*`\Properties\Hardware\Properties\Policies\Enable write caching on the disk` 来禁用写缓存。另一种办法是将 `wal_sync_method` 设为 `fsync` 或 `fsync_writethrough`，这两者都会阻止写缓存。

- 在 macOS 上，可以通过将 `wal_sync_method` 设为 `fsync_writethrough` 来防止写缓存。

[1] http://sg.danny.cz/sg/sdparm.html

较新的 SATA 驱动器（遵循 ATAPI-6 或更高版本标准的那些）提供了驱动器缓存刷新命令（`FLUSH CACHE EXT`），而 SCSI 驱动器则早已支持类似的 `SYNCHRONIZE CACHE` 命令。这些命令对 PostgreSQL 并不直接可见，但某些文件系统（如 ZFS、ext4）可以利用它们将启用了回写缓存的驱动器上的数据刷新到盘片。遗憾的是，这类文件系统与带后备电池单元（BBU）的磁盘控制器配合时表现并不理想。在这种配置中，同步命令会强制把控制器缓存中的全部数据写到磁盘，削弱 BBU 的大部分好处。可以运行pg_test_fsync 程序查看自己是否受此影响。如果确实受影响，而又能这样做，则可通过关闭文件系统中的写屏障或重新配置磁盘控制器，恢复 BBU 的性能优势。如果关闭了写屏障，一定要确保电池保持正常工作；电池故障有可能导致数据丢失。希望文件系统和磁盘控制器的设计者最终能解决这种次优行为。

当操作系统向存储硬件发出写请求后，它几乎做不了什么来确保数据已经到达真正的非易失性存储区域。相反，管理员有责任确保所有存储组件都能同时保证数据和文件系统元数据的完整性。应避免使用写缓存没有电池后备的磁盘控制器。在驱动器层面，如果驱动器无法保证在关闭前把数据写出，就应禁用回写缓存。如果使用 SSD，要注意其中很多默认并不遵守缓存刷新命令。可以借助 `diskchecker.pl`[2] 测试 I/O 子系统行为是否可靠。

数据丢失的另一种风险来自磁盘盘片写操作本身。磁盘盘片被分为若干扇区，通常每个 512 字节。每一次物理读写都处理整个扇区。当写请求到达驱动器时，其数据量可能是 512 字节的某个倍数（PostgreSQL 通常一次写入 8192 字节，也就是 16 个扇区），而写入过程可能在任何时刻因断电失败，这就意味着某些 512 字节扇区写成功了，另一些却没有。为防范这类故障，PostgreSQL 会在修改磁盘上的实际页面之前，定期将整页镜像写入持久 WAL 存储。这样一来，在崩溃恢复时 PostgreSQL 就能从 WAL 恢复部分写入的页面。如果你使用了能阻止部分页面写入的文件系统软件（如 ZFS），可以通过关闭参数full_page_writes 来禁用这种页面镜像。带后备电池单元（BBU）的磁盘控制器并不能阻止部分页面写入，除非它们保证写入 BBU 的数据总是完整的整页（8kB）。

PostgreSQL 还可防范存储设备上因硬件错误或随时间发生的介质故障而引起的某些数据损坏，例如读写垃圾数据。

- WAL 文件中的每条单独记录都受 CRC-32（32 位）校验保护，因此我们可以判断记录内容是否正确。CRC 值会在写入每条 WAL 记录时设置，并在崩溃恢复、归档恢复和复制过程中进行检查。

- 数据页目前默认不带校验和，但记录在 WAL 记录中的整页镜像会受到保护；关于启用数据页校验和的详情，请参见initdb。

- `pg_xact`、`pg_subtrans`、`pg_multixact`、`pg_serial`、`pg_notify`、`pg_stat`、`pg_snapshots` 等内部数据结构并不直接带校验和，其页面也不受整页写保护。不过，在这些数据结构是持久的场合，会写入 WAL 记录，使最近的更改能够在崩溃恢复时被准确重建，而这些 WAL 记录本身会按前述方式受到保护。

- `pg_twophase` 中各个单独的状态文件受 CRC-32 保护。

- 较大 SQL 查询用于排序、物化和存放中间结果的临时数据文件当前不带校验和，对这些文件的更改也不会写入 WAL 记录。

PostgreSQL 不防范可纠正的内存错误，因此这里假定你使用的是带有业界标准纠错码（ECC）或更强保护机制的 RAM。

30.2. 预写式日志（WAL）

预写式日志（WAL）是确保数据完整性的标准方法。大多数（即使不是全部）事务处理书籍中都能找到详细描述。简而言之，WAL 的核心思想是，对数据文件（即表和索引所在之处）的更改，

[2] https://brad.livejournal.com/2116715.html

必须在这些更改被记入日志之后才能写出；也就是说，必须在描述这些更改的 WAL 记录被刷入持久存储之后，再去写数据文件。如果遵循这个过程，就无需在每次事务提交时都把数据页刷盘，因为我们知道，一旦发生崩溃，就可以利用日志恢复数据库：任何尚未应用到数据页上的更改，都可以根据 WAL 记录重做。（这就是前滚恢复，也称 REDO。）

> **提示**
>
> 由于 WAL 会在崩溃后恢复数据库文件内容，因此要可靠地存储数据文件或 WAL 文件，并不需要日志化文件系统。实际上，日志开销反而可能降低性能，特别是在日志会导致文件系统数据刷盘时。幸运的是，通常可以通过文件系统挂载选项关闭日志期间的数据刷写，例如 Linux ext3 文件系统上的 `data=writeback`。不过，日志化文件系统确实能够提高崩溃后的启动速度。

使用 WAL 会显著减少磁盘写入次数，因为要保证事务已提交，通常只需把 WAL 文件刷入磁盘，而不必把该事务修改过的每个数据文件都刷盘。WAL 文件按顺序写入，因此同步 WAL 的代价远低于刷写数据页。对于处理大量小事务、且这些事务触及数据存储不同部分的服务器，这一点尤其明显。此外，当服务器正在处理大量并发的小事务时，对 WAL 文件执行一次 `fsync` 就可能足以提交多个事务。

WAL 还使在线备份和时间点恢复成为可能，如第 25.3 节所述。通过归档 WAL 数据，我们就能支持回退到可用 WAL 数据覆盖范围内的任意时刻：只需先安装数据库的一个较早物理备份，然后把 WAL 重放到目标时刻即可。更进一步说，这个物理备份不必是数据库状态的瞬时快照 – 即使备份是在一段时间内完成的，重放这段时间内的 WAL 也能修复任何内部不一致。

30.3. 异步提交

异步提交是一项可选功能，它允许事务更快完成，但代价是如果数据库发生崩溃，最近的事务可能会丢失。在许多应用中，这是可以接受的权衡。

如前一节所述，事务提交通常是同步的：服务器要等到该事务的 WAL 记录被刷入持久存储后，才向客户端返回成功指示。因此，客户端可以确信那些已被报告为提交成功的事务一定会被保留，即使服务器紧接着立刻崩溃也是如此。然而，对于短事务而言，这种等待是总事务时间的主要组成部分。选择异步提交模式意味着服务器会在事务从逻辑上完成后就立刻返回成功，而不必等到它生成的 WAL 记录真正到达磁盘。这会显著提升小事务的吞吐量。

异步提交引入了数据丢失风险。在向客户端报告事务完成与事务真正被提交（也就是说，即使服务器崩溃也保证不会丢失）之间，存在一个很短的时间窗口。因此，如果客户端会基于“系统一定会记住这个事务”这一假设采取外部动作，就不应使用异步提交。例如，银行显然不会对记录 ATM 吐钞的事务使用异步提交。但在很多场景中，例如事件日志记录，并不需要这种强保证。

使用异步提交所承担的风险是数据丢失，而不是数据损坏。如果数据库发生崩溃，它会通过重放 WAL 直到最后一条已刷写记录来恢复。因此，数据库会被恢复到自洽状态，但任何尚未刷到磁盘的事务都不会反映在该状态中。其最终效果就是丢失最后几个事务。由于事务按提交顺序重放，所以不会引入任何不一致性 – 例如，如果事务 B 做出的更改依赖于更早的事务 A 的效果，就不可能出现 A 的效果丢失而 B 的效果却被保留的情况。

用户可以为每个事务选择提交模式，因此同步提交事务和异步提交事务可以并发运行。
这使得在性能与事务持久性的确定性之间能够灵活权衡。提交模式由用户可设置的参数 synchronous_commit 控制，而该参数可以用任何设置配置参数的方式来更改。任一事务实际使用的模式取决于事务提交开始时 `synchronous_commit` 的值。

某些实用命令，例如 DROP TABLE，无论 synchronous_commit 如何设置，都会强制同步提交。这是为了保证服务器文件系统与数据库逻辑状态之间的一致性。支持两阶段提交的命令，例如 PREPARE TRANSACTION，也始终是同步的。

如果数据库在异步提交与写出该事务 WAL 记录之间的风险窗口内崩溃，那么该事务所做的更改将丢失。该风险窗口的时长是受限的，因为一个后台进程（“WAL 写入器”）会将尚未写出的 WAL 记录每隔 wal_writer_delay 毫秒刷到磁盘。风险窗口的实际最大时长是 wal_writer_delay 的三倍，因为 WAL 写入器在系统繁忙时被设计为倾向于一次写出整页。

> **小心**
>
> 立即模式关闭等同于服务器崩溃，因此也会造成任何未刷写的异步提交丢失。

异步提交的行为不同于将 fsync 设为 off。fsync 是影响所有事务的全服务器设置。它会禁用 PostgreSQL 内部所有试图同步数据库不同部分写入的逻辑，因此系统崩溃（即硬件或操作系统崩溃，而不是 PostgreSQL 本身故障）可能导致数据库状态遭受任意严重的损坏。在许多场景中，异步提交能带来关闭 fsync 所能获得的大部分性能提升，却不承担数据损坏的风险。

commit_delay 听起来也很像异步提交，但它实际上是一种同步提交方法（事实上，在异步提交期间 commit_delay 会被忽略）。commit_delay 会在事务把 WAL 刷到磁盘之前引入一段延迟，希望某个事务执行的一次刷盘也能服务于其他差不多同时提交的事务。可以把这个设置看成是扩大一个时间窗口，让事务得以加入即将参与同一次刷盘的组，从而在多个事务之间摊销这次刷盘的代价。

30.4. WAL 配置

有若干与 WAL 相关的配置参数会影响数据库性能。本节解释它们的用法。有关设置服务器配置参数的一般信息，请参见第 19 章。

检查点是事务序列中的一些点，在这些点上可以保证堆和索引数据文件已经用检查点之前写入的全部信息进行了更新。执行检查点时，所有脏数据页都会刷盘，并向 WAL 文件写入一条特殊的检查点记录。（更改记录此前已写入 WAL 文件并刷盘。）如果发生崩溃，崩溃恢复过程会查看最新的检查点记录，以确定应该从 WAL 的哪个位置（称为重做记录）开始执行 REDO。该点之前对数据文件所做的任何更改都保证已经在磁盘上。因此，检查点之后，位于包含重做记录的段之前的 WAL 段不再需要，可以被回收或删除。（若正在执行 WAL 归档，则这些 WAL 段必须先归档，随后才能回收或删除。）

检查点要求将所有脏数据页刷盘，这可能造成可观的 I/O 负载。因此，检查点活动会被节流，使 I/O 从检查点开始时就启动，并在下一个检查点预计开始前完成；这样可将检查点期间的性能下降降到最低。

服务器的检查点进程会定期自动执行检查点。每经过 checkpoint_timeout 秒，或者当 max_wal_size 即将超出时，就会开始一个检查点，以先到者为准。默认设置分别是 5 分钟和 1 GB。如果自上一个检查点以来没有写入任何 WAL，那么即使 checkpoint_timeout 已经过去，也会跳过新的检查点。（如果正在使用 WAL 归档，而你希望对文件归档频率设定下限，从而限制潜在数据丢失，那么应当调整参数 archive_timeout，而不是检查点参数。）也可以使用 SQL 命令 CHECKPOINT 强制执行检查点。

减小 checkpoint_timeout 和/或 max_wal_size 会让检查点更频繁地发生。这样可以加快崩溃后恢复，因为需要重做的工作更少。不过，这必须与更频繁地将脏数据页刷盘所增加的成本权衡。如果设置了full_page_writes（默认就是如此），还要考虑另一个因素。为了保证数据页一致性，每个检查点之后对某个数据页的首次修改，都会导致把整页内容写入日志。在这种情况

下，更短的检查点间隔会增加输出到 WAL 的数据量，部分抵消缩短间隔的目的，并且无论如何都会带来更多磁盘 I/O。

检查点的代价相当高，首先是因为它要求写出当前所有脏缓冲区，其次是因为它还会像上文所述那样带来额外的后续 WAL 流量。因此，明智的做法是将检查点参数设得足够高，避免检查点发生得过于频繁。作为对检查点参数的一种简单合理性检查，你可以设置参数checkpoint_warning。如果检查点之间的间隔小于 `checkpoint_warning` 秒，服务器日志就会输出一条消息，建议增大 `max_wal_size`。偶尔出现这样的消息无需惊慌，但如果它频繁出现，就应当增大检查点控制参数。大型 `COPY` 传输之类的批量操作，如果 `max_wal_size` 设得不够高，也可能导致出现多次此类警告。

为了避免大量页面写出在短时间内淹没 I/O 系统，检查点期间对脏缓冲区的写出会被分散到一段时间内。这段时间由checkpoint_completion_target 控制，它表示为检查点间隔的一个分数。系统会调节 I/O 速率，使检查点在给定比例的 `checkpoint_timeout` 秒数经过时完成，或者在 `max_wal_size` 即将超出之前完成，以较早者为准。使用默认值 0.5 时，可以预期 PostgreSQL 会在下一次检查点开始前的这段时间约过一半时完成每个检查点。在正常运行时已非常接近最大 I/O 吞吐量的系统上，你可能希望增大 `checkpoint_completion_target`，以降低检查点带来的 I/O 负载。这样做的缺点是，延长检查点会影响恢复时间，因为需要保留更多 WAL 段以备恢复时使用。虽然 `checkpoint_completion_target` 可以设到 1.0，但最好保持在该值以下（或许最多为 0.9），因为检查点除了写出脏缓冲区之外还包含其他活动。设置为 1.0 很可能导致检查点不能按时完成，从而因为所需 WAL 段数量的意外波动而损失性能。

在 Linux 和 POSIX 平台上，checkpoint_flush_after 允许在检查点写出的操作系统页面达到可配置的字节数后，强制将它们刷盘。否则，这些页面可能会停留在操作系统页缓存中，从而在检查点末尾发出 `fsync` 时引发停顿。这个设置通常有助于降低事务延迟，但也可能对性能产生不利影响，尤其是在工作负载大于 shared_buffers、但小于操作系统页缓存时。

`pg_wal` 目录中 WAL 段文件的数量取决于 `min_wal_size`、`max_wal_size`，以及此前几个检查点周期中产生的 WAL 数量。当旧的 WAL 段文件不再需要时，它们会被删除或回收（也就是重命名为编号序列中的未来段文件）。如果由于 WAL 输出速率短期突增而超过了 `max_wal_size`，那么不再需要的段文件就会被删除，直到系统重新回到该限制之下。在此限制以下，系统会回收足够多的 WAL 文件，以覆盖到下一个检查点之前的预估需求，其余则删除。这个估计基于此前几个检查点周期中所用 WAL 文件数量的移动平均值。如果实际使用量超过估计值，移动平均会立即增大，因而在一定程度上能够适应峰值使用量，而不仅仅是平均使用量。`min_wal_size` 规定了为将来复用而回收的 WAL 文件最小总量；即使系统空闲，且 WAL 使用量估计表明只需要很少 WAL，也始终会回收这么多文件以供未来使用。

无论 `max_wal_size` 如何设置，最近的wal_keep_segments + 1 个 WAL 文件都会始终保留。此外，如果使用了 WAL 归档，那么旧段在完成归档之前不能被删除或回收。如果 WAL 归档跟不上 WAL 产生的速度，或者 `archive_command` 反复失败，那么旧 WAL 文件就会在 `pg_wal` 中不断累积，直到问题解决。使用复制槽的备库如果速度很慢或者已经失败，也会产生同样的效果（见第 26.2.6 节）。

在归档恢复或备库模式下，服务器会周期性地执行重启点，其行为类似于正常运行时的检查点：服务器会强制将其所有状态刷盘，更新 `pg_control` 文件，以表明已经处理过的 WAL 数据无需再次扫描，然后回收 `pg_wal` 目录中的旧 WAL 段文件。重启点的执行频率不会高于主库上的检查点，因为重启点只能在检查点记录处执行。当到达某条检查点记录且距离上次重启点至少已经过去 `checkpoint_timeout` 秒，或者 WAL 大小即将超过 `max_wal_size` 时，就会触发重启点。不过，由于重启点的可执行时机受到限制，恢复期间经常会超出 `max_wal_size`，最多可能超出一个检查点周期对应的 WAL 量。（无论如何，`max_wal_size` 从来都不是硬上限，因此你始终应留出充足余量，以免耗尽磁盘空间。）

有两个常用的内部 WAL 函数：`XLogInsertRecord` 和 `XLogFlush`。`XLogInsertRecord` 用于把新记录放入共享内存中的 WAL 缓冲区。如果新记录没有可用空间，`XLogInsertRecord`

就不得不把若干已满的 WAL 缓冲区写出（移入内核缓存）。这并不理想，因为 XLogInsertRecord 会在每一次数据库底层修改（例如插入行）时被调用，而在调用它时，受影响数据页上正持有排他锁，因此这一操作必须尽可能快。更糟的是，写 WAL 缓冲区还可能迫使系统创建新的 WAL 段，这会花费更多时间。正常情况下，WAL 缓冲区应由 XLogFlush 请求负责写出并刷盘；该请求大多发生在事务提交时，以确保事务记录已刷盘至持久存储。在 WAL 输出量很高的系统上，XLogFlush 请求可能不够频繁，无法避免 XLogInsertRecord 自己去执行写出。在这种系统上，应增加 WAL 缓冲区的数量，具体做法是修改参数 wal_buffers。当设置了 full_page_writes 且系统非常繁忙时，提高 wal_buffers 还有助于平滑每次检查点后紧接着那段时期的响应时间。

参数 commit_delay 定义了组提交的领导者进程在 XLogFlush 内部获取锁之后会睡眠多少微秒，在此期间，组提交的追随者会排队跟在领导者后面。这一延迟允许其他服务器进程把自己的提交记录添加到 WAL 缓冲区中，以便它们都能由领导者最终执行的同步操作一并刷盘。如果没有启用 fsync，或者当前处于活跃事务中的其他会话少于commit_siblings 个，就不会发生睡眠；这样可以避免在其他会话不太可能很快提交时还去睡眠。请注意，在某些平台上，休眠请求的分辨率是 10 毫秒，因此介于 1 到 10000 微秒之间的任何非零 commit_delay 设置，效果都相同。还要注意，在某些平台上，休眠操作持续的时间可能会比该参数所请求的略长。

由于 commit_delay 的目的，是让每次刷盘操作的成本在并发提交的事务之间分摊（代价是可能增加事务延迟），因此在选择该设置之前，有必要先量化这一成本。该成本越高，commit_delay 在提高事务吞吐量方面预计就越有效，当然这种效果也有上限。可以使用 pg_test_fsync 程序来测量单次 WAL 刷盘操作平均需要多少微秒。通常，把该程序报告的“单次 8kB 写操作之后再刷盘”的平均耗时取一半，往往是 commit_delay 最有效的设置，因此建议把这一值作为针对特定工作负载进行优化时的起点。虽然在 WAL 位于高延迟旋转磁盘上时，调优 commit_delay 尤其有用，但即便是在同步时间非常快的存储介质上，例如固态驱动器或带有电池后备写缓存的 RAID 阵列，其收益也可能很可观；不过这无疑应当在有代表性的工作负载上进行测试。在这种场景下，通常应使用更高的 commit_siblings 值，而在高延迟介质上，较小的 commit_siblings 值往往更有帮助。还要注意，commit_delay 如果设得过高，也完全可能由于显著增加事务延迟而导致总体事务吞吐量下降。

当 commit_delay 设为 0（默认值）时，仍然可能出现一种组提交，但每个组只会包含那些恰好在前一次刷盘操作（若有）进行期间，到达“需要将其提交记录刷盘”这一点的会话。客户端数量更高时，往往会出现一种“gangway effect”，以至于即使 commit_delay 为 0，组提交的效果也会变得很显著，因此显式设置 commit_delay 的帮助反而会变小。只有在以下两种条件同时满足时，设置 commit_delay 才可能有帮助：（1）存在一些并发提交的事务；以及（2）吞吐量在某种程度上受限于提交速率。但在旋转延迟较高的介质上，即使只有两个客户端（也就是一个正在提交的客户端，再加上一个并发的兄弟事务），这个设置也能有效提高事务吞吐量。

参数 wal_sync_method 决定 PostgreSQL 将如何请求内核把 WAL 更新强制刷盘。就可靠性而言，所有选项应当都是相同的；例外是 fsync_writethrough，它有时能够在其他选项做不到时强制将磁盘缓存中的数据刷盘。不过，具体哪个选项速度最快则高度依赖平台。可以使用 pg_test_fsync 程序测试不同选项的速度。请注意，如果 fsync 已被关闭，那么这个参数就没有意义。

启用配置参数 wal_debug（前提是 PostgreSQL 编译时支持它）会导致每一次 XLogInsertRecord 和 XLogFlush 的 WAL 调用都被记录到服务器日志中。将来，这个选项可能会被更通用的机制所取代。

30.5. WAL 内部机制

WAL 会自动启用；管理员除了确保满足 WAL 文件的磁盘空间需求，并完成任何必要调优之外，不需要再做其他操作（见第 30.4 节）。

每写入一条新记录，WAL 记录就会追加到 WAL 文件中。插入位置由日志序列号（LSN）描述，它是 WAL 内的一个字节偏移量，会随着每条新记录的写入而单调递增。LSN 值以数据类型 `pg_lsn` 返回。通过比较两个值，可以计算它们之间相隔的 WAL 数据量，因此它们可用于衡量复制和恢复的进度。

WAL 文件保存在数据目录下的 `pg_wal` 目录中，由一组段文件组成，每个段通常为 16 MB（不过可通过修改 `--wal-segsize` initdb 选项来改变大小）。每个段又分为若干页面，通常每页 8 kB（该大小可通过 configure 选项 `--with-wal-blocksize` 更改）。WAL 记录头定义在 `access/xlogrecord.h` 中；记录内容则取决于被记录的事件类型。段文件以不断递增的数字命名，从 `000000010000000000000001` 开始。这些编号不会回绕，但要把可用的编号耗尽还需要非常、非常长的时间。

如果能将 WAL 放在与主数据库文件不同的磁盘上，会更有利。这可以通过把 `pg_wal` 目录移动到其他位置（当然要在服务器关闭时进行），然后在主数据目录中的原位置创建一个指向新位置的符号链接来实现。

WAL 的目标是确保数据库记录在被修改之前，日志已经先被写出；但如果磁盘驱动器实际上只是缓存了数据、尚未将其存储到磁盘上，却向内核谎报写入成功，这一目标就会被破坏。在这种情况下，断电可能导致不可恢复的数据损坏。管理员应尽量确保保存 PostgreSQL WAL 文件的磁盘不会做出这种虚假报告。（见第 30.1 节。）

在完成检查点并刷写 WAL 之后，该检查点的位置会保存在文件 `pg_control` 中。因此，在恢复开始时，服务器会先读取 `pg_control`，再读取检查点记录；然后从检查点记录给出的 WAL 位置向前扫描并执行 REDO。由于检查点之后对数据页的第一次修改，会把整页内容保存在 WAL 中（假设没有禁用 full_page_writes），因而自该检查点以来被修改的所有页面都会恢复到一致状态。

为处理 `pg_control` 损坏的情况，我们本应支持按相反顺序扫描现有 WAL 段 – 也就是从新到旧 –，以便找到最新检查点。这一功能尚未实现。`pg_control` 足够小（不足一个磁盘页），因此不会受部分写问题影响；到目前为止，也没有仅仅因为无法读取 `pg_control` 本身而导致数据库故障的报告。所以，尽管理论上它是个薄弱环节，`pg_control` 在实践中似乎不是问题。

第 31 章 逻辑复制

逻辑复制是一种基于数据对象及其复制标识（通常是主键）来复制数据对象及其变更的方法。这里使用“逻辑”一词，是为了与物理复制相区别；物理复制使用精确的块地址，并按字节逐一复制。PostgreSQL 可以同时支持这两种机制，见第 26 章。逻辑复制允许对数据复制和安全性进行细粒度控制。

逻辑复制采用发布和订阅模型，一个或多个订阅者订阅某个发布者节点上的一个或多个发布。订阅者从其订阅的发布中拉取数据，随后还可以重新发布这些数据，以实现级联复制或更复杂的配置。

表的逻辑复制通常开始于在发布者数据库中为数据建立快照，并将其复制到订阅者。完成后，发布者上的变更会在发生时实时发送给订阅者。订阅者会按与发布者相同的顺序应用这些数据，因此对于单个订阅中的发布，可以保证事务一致性。这种数据复制方法有时也称为事务性复制。

逻辑复制的典型用途包括：

- 当单个数据库或数据库的某个子集发生变更时，将增量变更发送给订阅者。
- 当单个变更到达订阅端时触发触发器。
- 将多个数据库汇总到单个数据库中（例如用于分析）。
- 在 PostgreSQL 的不同主版本之间进行复制。
- 在不同平台上（例如 Linux 到 Windows）的 PostgreSQL 实例之间进行复制。
- 向不同用户组开放对复制数据的访问。
- 在多个数据库间共享数据库的一个子集。

订阅端数据库的行为与任何其他 PostgreSQL 实例相同，并且可以通过定义自己的发布，作为其他数据库的发布者。当应用把订阅端视为只读时，单个订阅不会产生冲突。另一方面，如果应用或其他订阅者对同一组表执行了其他写操作，就可能发生冲突。

31.1. 发布

发布可以定义在任何物理复制主库上。定义发布的节点称为发布者。发布是一组由某个表或一组表产生的变更，也可称为变更集或复制集。每个发布只存在于一个数据库中。

发布与模式不同，不影响表的访问方式。每个表在需要时都可以加入多个发布。发布目前只能包含表。除非创建发布时使用ALL TABLES，否则对象必须显式添加。

发布可以选择将要产生的变更限制为 INSERT、UPDATE、DELETE 和 TRUNCATE 的任意组合，类似于触发器按特定事件类型触发。默认会复制所有操作类型。

已发布的表必须配置了“复制标识”，才能复制UPDATE 和 DELETE 操作，以便在订阅端能够识别要更新或删除的对应行。默认情况下，使用主键（如果有的话）。也可以将另一个唯一索引（有一些额外要求）设置为复制标识。如果表没有任何合适的键，则可以将其复制标识设置为“full”，这意味着整行都成为键。但这样做效率很低，只应在无其他方案时作为后备使用。如果在发布端设置了“full”以外的复制标识，则订阅端也必须设置包含相同或更少列的复制标识。关于如何设置复制标识的详细信息，见 REPLICA IDENTITY。如果一个没有复制标识的表被添加到了可复制 UPDATE 或 DELETE 操作的发布中，后续的UPDATE 或 DELETE 操作将在发布端产生错误。无论复制标识如何，INSERT 操作都可以正常进行。

每个发布都可以有多个订阅者。

发布使用 CREATE PUBLICATION 命令创建，随后可以使用相应命令进行修改或删除。

可以使用 ALTER PUBLICATION 动态添加和移除各个表。`ADD TABLE` 和 `DROP TABLE` 操作都是事务性的；因此一旦事务提交，表将在正确的快照点开始或停止复制。

31.2. 订阅

订阅是逻辑复制的下游端。定义订阅的节点称为订阅者。订阅定义了与另一个数据库的连接以及它想订阅的一组发布（一个或多个）。

订阅者数据库的行为与任何其他 PostgreSQL 实例相同，也可以通过定义自己的发布作为其他数据库的发布者。

订阅者节点可按需拥有多个订阅。也可以在同一对发布者-订阅者之间定义多个订阅，但必须注意确保被订阅的发布对象不重叠。

每个订阅将通过一个复制槽（见第 26.2.6 节）接收变更。对于已有表数据的初始数据同步，可能需要额外的临时复制槽。

逻辑复制订阅可以作为同步复制中的备库（见第 26.2.8 节）。其备库名称默认是订阅名。也可在订阅连接信息中通过 `application_name` 指定替代名称。

如果当前用户是超级用户，`pg_dump` 会转储订阅。否则会写出警告并跳过订阅，因为非超级用户无法从 `pg_subscription` 目录读取全部订阅信息。

订阅使用 CREATE SUBSCRIPTION 添加，可随时使用ALTER SUBSCRIPTION 命令停止/恢复，并使用DROP SUBSCRIPTION 移除。

删除并重建订阅会丢失同步信息，这意味着之后必须重新同步数据。

模式定义不会被复制，已发布的表必须在订阅端存在。复制目标只能是普通表。例如，不能复制到视图。

发布端与订阅端通过表的完全限定名进行匹配。不支持复制到订阅端名称不同的表。

表的列也按名称匹配。订阅端表中列的顺序无需与发布端匹配。列的数据类型也无需匹配，只要数据的文本表示可以转换为目标类型即可。例如，可以从`integer` 类型的列复制到 `bigint` 类型的列。目标表也可以有已发布表未提供的额外列。这些额外列将用目标表定义中指定的默认值填充。

31.2.1. 复制槽管理

如前所述，每个（活动）订阅都从远程（发布）端的复制槽接收变更。通常，在使用 `CREATE SUBSCRIPTION` 创建订阅时，会自动创建远程复制槽；在使用`DROP SUBSCRIPTION` 删除订阅时，也会自动删除该槽。但是，在某些情况下，分别操作订阅及其底层复制槽可能有用或必要。例如：

- 创建订阅时复制槽已经存在。此时可使用 `create_slot = false` 选项创建订阅，以关联到现有的槽。
- 创建订阅时远程主机不可达或状态不明确。此时可使用 `connect = false` 选项创建订阅。这样就完全不会联系远程主机。pg_dump 就是这样使用的。在订阅激活前，必须手动创建远程复制槽。

- 删除订阅时需要保留复制槽。这在将订阅端数据库迁移到另一台主机并从那里激活时很有用。此时应在尝试删除订阅之前，使用ALTER SUBSCRIPTION 解除槽与订阅的关联。

- 删除订阅时远程主机不可达。此时应在尝试删除订阅之前，使用ALTER SUBSCRIPTION 解除槽与订阅的关联。如果远程数据库实例已经不存在了，则无需其他操作。但如果远程数据库实例只是暂时不可达，则应手动删除复制槽，否则它将继续保留 WAL，并可能最终导致磁盘空间耗尽。此类情况应仔细调查。

31.3. 冲突

逻辑复制的行为类似于普通的 DML 操作，即使数据在订阅端本地已被更改，仍会更新数据。如果传入的数据违反了任何约束，复制将停止。这称为冲突。当复制 UPDATE 或 DELETE 操作时，缺失的数据不会产生冲突，这类操作只会被简单跳过。

冲突将产生一个错误并停止复制；它必须由用户手动解决。冲突的详细信息可在订阅端的服务器日志中找到。

解决冲突可以通过更改订阅端的数据使其不与传入的更改冲突，或者跳过与现有数据冲突的事务来完成。可以通过调用pg_replication_origin_advance() 函数来跳过事务，传入与订阅名称对应的*node_name* 和一个位置。复制源的当前位置可在 pg_replication_origin_status 系统视图中查看。

31.4. 限制

逻辑复制目前有以下限制或缺失的功能。这些问题可能会在未来版本中得到解决。

- 数据库模式和 DDL 命令不会被复制。初始模式可以手工使用 pg_dump --schema-only 复制。之后的模式更改需要手工保持同步。（不过请注意，两端的模式其实不需要绝对一致。）当活动数据库中的模式定义发生变化时，逻辑复制仍然能够稳健运行：如果发布端修改了模式，而复制数据开始到达订阅端时与表模式不匹配，复制就会报错，直到模式被更新。在很多情况下，可以通过先在订阅端应用仅添加内容的模式变更来避免间歇性错误。

- 序列数据不会被复制。由序列支撑的 serial 或标识列中的数据当然会作为表的一部分被复制，但序列本身在订阅端仍会显示起始值。如果订阅端被当作只读数据库使用，这通常不是问题。但如果计划把订阅端数据库用作某种切换或故障切换目标，那么就需要把序列更新到最新值，可以从发布端复制当前数据（例如使用 pg_dump），也可以直接从表本身确定一个足够高的值。

- 支持复制 TRUNCATE 命令，但在截断通过外键连接在一起的一组表时必须格外小心。复制截断操作时，订阅端会截断与发布端相同的一组表，这组表可以是显式指定的，也可以是通过 CASCADE 隐式收集到的，然后再减去不属于该订阅的表。如果所有受影响的表都属于同一个订阅，这样可以正确工作。但如果订阅端上将被截断的某些表通过外键链接到了不属于同一个订阅（或根本不属于任何订阅）的表，那么在订阅端应用该截断操作就会失败。

- 大对象（见第 35 章）不会被复制。除了把数据存储在普通表中，没有其他变通办法。

- 只能从基表复制到基表。也就是说，发布端和订阅端的表必须都是普通表，不能是视图、物化视图、分区根表或外部表。因此，对于分区，可以一对一复制分区层次结构，但目前还不能复制到采用不同分区方式的目标。尝试复制基表以外的表会导致错误。

31.5. 架构

逻辑复制首先复制发布端数据库中数据的快照。完成后，发布端的变更会随着发生实时发送给订阅端。订阅端按照发布端提交的顺序应用数据，因此对于单个订阅中的发布，可以保证事务一致性。

逻辑复制采用类似于物理流复制的架构（见第 26.2.5 节）。它由 “walsender” 和 “apply” 进程实现。walsender 进程启动 WAL 的逻辑解码（见第 49 章），并加载标准逻辑解码输出插件（pgoutput）。该插件将从 WAL 读取的变更转换为逻辑复制协议（见第 53.5 节），并根据发布规范过滤数据。然后使用流复制协议将数据持续传输给应用工作进程，该工作进程将数据映射到本地表，并按正确的事务顺序逐一应用每个变更。

订阅端数据库上的应用进程总是在 session_replication_role 设置为 replica 的情况下运行。这意味着默认情况下，触发器和规则不会在订阅端触发。用户可以选择使用 ALTER TABLE 命令和 ENABLE TRIGGER 及 ENABLE RULE 子句在表上启用触发器和规则。

逻辑复制的应用进程目前只触发行触发器，不触发语句触发器。但初始表同步的实现方式类似于 COPY 命令，因此会触发INSERT 的行触发器和语句触发器。

31.5.1. 初始快照

已订阅表中的初始数据由一种特殊的应用进程的并行实例建立快照并复制。该进程将创建自己的临时复制槽并复制现有数据。现有数据复制完成后，工作进程进入同步模式，通过使用标准逻辑复制流式传输初始数据复制期间发生的所有变更，确保该表与主应用进程同步。同步完成后，该表的复制控制权交还给主应用进程，此后复制照常继续。

31.6. 监控

因为逻辑复制基于类似于物理流复制的架构，发布端节点上的监控也类似于物理复制主库的监控（见第 26.2.5.2 节）。

与订阅相关的监控信息可在 pg_stat_subscription 中查看。这个视图为每个订阅工作进程包含一行。一个订阅根据其状态，可能有零个或多个活跃的订阅工作进程。

正常情况下，已启用的订阅会有一个应用进程在运行。被禁用或已崩溃的订阅在此视图中将有零行。如果任何表的初始数据同步正在进行中，将有额外的工作进程用于正在同步的表。

31.7. 安全性

能够修改订阅端表模式的用户可以以超级用户身份执行任意代码。限制此类表的所有权和 TRIGGER 权限，仅授予超级用户信任的角色。此外，如果不受信任的用户可以创建表，则仅使用显式列出表的发布。也就是说，只有在超级用户信任发布端或订阅端上允许创建非临时表的每个用户时，才创建FOR ALL TABLES 的订阅。

用于复制连接的角色必须具有 REPLICATION 属性（或者是超级用户）。如果该角色缺少 SUPERUSER 和 BYPASSRLS，发布端的行安全策略可能会被执行。如果该角色不信任所有表的所有者，可在连接字符串中包含options=-crow_security=off；这样如果某个表所有者添加了行安全策略，该设置将导致复制停止而不是执行该策略。必须在pg_hba.conf 中配置该角色的访问权限，并且该角色必须具有LOGIN 属性。

为了能够复制初始表数据，用于复制连接的角色必须对已发布的表具有 SELECT 权限（或者是超级用户）。

要创建发布，用户必须在数据库中具有 CREATE 权限。

要向发布添加表，用户必须对该表拥有所有权。要创建自动发布所有表的发布，用户必须是超级用户。

要创建订阅，用户必须是超级用户。

订阅应用进程将以超级用户的权限在本地数据库中运行。

权限只在复制连接开始时检查一次。不会在读取发布端的每条变更记录时重新检查，也不会在应用每条变更时重新检查。

31.8. 配置设置

逻辑复制需要设置几个配置选项。

在发布端，wal_level 必须设置为 logical，max_replication_slots 必须至少设置为预期连接的订阅数，加上为表同步预留的一些余量。而max_wal_senders 应至少设置为 max_replication_slots 加上同时连接的物理副本数。

订阅端也必须设置 max_replication_slots，以配置能够跟踪的复制源数量。在这种情况下，它至少应为将要添加到订阅端的订阅数。max_logical_replication_workers 必须至少设置为订阅数，加上为表同步预留的余量。此外，max_worker_processes 可能需要调整以容纳复制工作进程，至少为（max_logical_replication_workers + 1）。注意一些扩展和并行查询也会占用 max_worker_processes 中的工作进程槽。

31.9. 快速设置

首先在 postgresql.conf 中设置配置选项：

```
wal_level = logical
```

对于基础设置来说，其他必需设置的默认值已经足够。

需要调整 pg_hba.conf 以允许复制（这里的值取决于实际网络配置以及要用于连接的用户）：

```
host     all     repuser     0.0.0.0/0     scram-sha-256
```

然后在发布端数据库上：

```
CREATE PUBLICATION mypub FOR TABLE users, departments;
```

并在订阅端数据库上：

```
CREATE SUBSCRIPTION mysub CONNECTION 'dbname=foo host=bar user=repuser' PUBLICATION mypub;
```

上述语句会启动复制过程：先同步 users 和 departments 表的初始内容，然后开始复制这些表上的增量变更。

第 32 章 即时编译（JIT）

本章解释什么是即时编译，以及如何在PostgreSQL 中配置它。

32.1. 什么是 JIT 编译?

即时（Just-In-Time，JIT）编译是在运行时将某种形式的解释式程序求值转换为原生程序的过程。例如，计算某个特定的 SQL 谓词（如 `WHERE a.col = 3`）时，可以生成一个专用于该表达式、由 CPU 原生执行的函数，以取代能够对任意 SQL 表达式求值的通用代码，从而加快执行速度。

当 PostgreSQL 使用 `--with-llvm` 构建时，PostgreSQL 内置了使用 LLVM[1] 执行 JIT 编译的支持。

更多细节参见 `src/backend/jit/README`。

32.1.1. JIT 加速的操作

当前，PostgreSQL 的 JIT 实现支持对表达式求值和元组拆解进行加速。未来还可能加速其他若干操作。

表达式求值用于计算 `WHERE` 子句、目标列表、聚合和投影。它可以通过为每种情况生成专用代码来加速。

元组拆解是将磁盘上的元组（见第 69.6.1 节）转换为其内存表示的过程。它可以通过创建一个针对表布局和要提取的列数的专用函数来加速。

32.1.2. 内联

PostgreSQL 具有很强的可扩展性，允许定义新的数据类型、函数、操作符以及其他数据库对象；见第 38 章。实际上，内置对象几乎也是通过相同的机制实现的。这种可扩展性意味着会有一些额外开销，例如函数调用带来的开销（见第 38.3 节）。为了减少这类开销，JIT 编译可以把小函数的函数体内联到使用它们的表达式中，从而优化掉相当一部分开销。

32.1.3. 优化

LLVM 支持对生成的代码进行优化。有些优化的代价很低，因此每次使用 JIT 时都可以执行；另一些优化则只有在运行时间较长的查询中才有益处。更多有关优化的细节请参见 https://llvm.org/docs/Passes.html#transform-passes。

32.2. 何时使用 JIT?

JIT 编译主要对长时间运行且受 CPU 限制的查询有益。这类查询通常是分析型查询。对于短查询，执行 JIT 编译所增加的额外开销往往会高于它能节省的时间。

为了确定是否应使用 JIT 编译，会使用查询的总估计代价（见第 71 章和第 19.7.2 节）。查询的估计代价会与 jit_above_cost 的设置值进行比较。如果代价更高，就会执行 JIT 编译。之后还需要再做两个决定。第一，如果估计代价高于jit_inline_above_cost 的设置值，查询中使用的短函数和操作符会被内联。第二，如果估计代价高于jit_optimize_above_cost 的设置值，就会应用代价较高的优化来改进生成的代码。这些选项中的每一个都会增加 JIT 编译的开销，但也可能显著减少查询执行时间。

[1] https://llvm.org/

这些基于代价的决定是在规划时而不是执行时作出的。这意味着，当使用预备语句并采用通用计划时（见 PREPARE），控制这些决定的是预备时生效的配置参数值，而不是执行时的设置。

> **注意**
>
> 如果 jit 被设置为 `off`，或者没有可用的 JIT 实现（例如服务器在编译时未使用 `--with-llvm`），那么即使依据上述标准判断这样做会有收益，也不会执行 JIT。将 jit 设为 `off` 会同时影响规划时和执行时。

EXPLAIN 可用于查看是否使用了 JIT。例如，下面这个查询就没有使用 JIT：

```
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
--------------
 Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual time=0.303..0.303 rows=1 loops=1)
   ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4) (actual time=0.017..0.111
 rows=356 loops=1)
 Planning Time: 0.116 ms
 Execution Time: 0.365 ms
(4 rows)
```

考虑到该计划的代价，不使用 JIT 是完全合理的；JIT 的开销会超过可能节省的时间。调整代价阈值后，就会启用 JIT：

```
=# SET jit_above_cost = 10;
SET
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
--------------
 Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual time=6.049..6.049 rows=1 loops=1)
   ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4) (actual time=0.019..0.052
 rows=356 loops=1)
 Planning Time: 0.133 ms
 JIT:
   Functions: 3
   Options: Inlining false, Optimization false, Expressions true, Deforming true
   Timing: Generation 1.259 ms, Inlining 0.000 ms, Optimization 0.797 ms, Emission 5.048 ms,
 Total 7.104 ms
 Execution Time: 7.416 ms
```

从这里可以看出，JIT 已经被使用，但没有启用内联和高代价优化。如果再降低 jit_inline_above_cost 或 jit_optimize_above_cost，情况就会改变。

32.3. 配置

配置变量 jit 决定是否启用 JIT 编译。如果启用了它，那么配置变量 jit_above_cost、jit_inline_above_cost 以及 jit_optimize_above_cost 将决定是否对某个查询执行 JIT 编译，以及在这方面投入多少工作。

jit_provider 决定使用哪一种 JIT 实现。很少需要修改它。见第 32.4.2 节。

为了开发和调试，还存在一些额外的配置参数，如第 19.17 节中所述。

32.4. 可扩展性

32.4.1. 对扩展的内联支持

PostgreSQL 的 JIT 实现可以内联 `C` 和 `internal` 类型函数的函数体，以及基于这类函数的操作符。要对扩展中的函数这样做，需要让这些函数的定义可供使用。当使用PGXS 针对一个编译时启用了 LLVM JIT 支持的服务器构建扩展时，相关文件会自动构建并安装。

相关文件必须安装到 `$pkglibdir/bitcode/$extension/`，并将它们的摘要信息安装到 `$pkglibdir/bitcode/$extension.index.bc`，其中 `$pkglibdir` 是 `pg_config --pkglibdir` 返回的目录，而 `$extension` 是扩展共享库的基名。

> **注意**
>
> 对于 PostgreSQL 自身内置的函数，bitcode 会安装到 `$pkglibdir/bitcode/postgres`。

32.4.2. 可插拔的 JIT 提供者

PostgreSQL 提供了一个基于 LLVM 的 JIT 实现。JIT 提供者接口是可插拔的，因此无需重新编译即可更换提供者（尽管目前构建过程只为 LLVM 提供内联支持数据）。当前活动的提供者通过设置 jit_provider 来选择。

32.4.2.1. JIT 提供者接口

JIT 提供者通过动态装载指定名称的共享库来加载。系统使用通常的库搜索路径定位该库。为了提供所需的 JIT 提供者回调，并表明该库确实是一个 JIT 提供者，它需要提供一个名为 `_PG_jit_provider_init` 的 C 函数。系统会向该函数传入一个结构体，需要用各项动作对应的回调函数指针填充它：

```
struct JitProviderCallbacks
{
    JitProviderResetAfterErrorCB reset_after_error;
    JitProviderReleaseContextCB release_context;
    JitProviderCompileExprCB compile_expr;
};

extern void _PG_jit_provider_init(JitProviderCallbacks *cb);
```

第 33 章 回归测试

回归测试是针对 PostgreSQL 中 SQL 实现的一套全面测试。它们既测试标准 SQL 操作，也测试 PostgreSQL 的扩展能力。

33.1. 运行测试

回归测试既可以针对已经安装并正在运行的服务器执行，也可以在构建树中使用临时安装来执行。此外，运行测试时还有“并行”和“顺序”两种模式。顺序方式一次只运行一个测试脚本，而并行方式会启动多个服务器进程，以并行方式运行一组测试。并行测试能进一步确认进程间通信和锁定机制是否正常工作。

33.1.1. 针对临时安装运行测试

要在构建完成后、安装之前运行并行回归测试，请输入：

```
make check
```

运行位置为顶层目录。（也可以切换到 `src/test/regress` 并在那里运行此命令。）结束时，应该会看到类似下面的结果：

```
=======================
 All 115 tests passed.
=======================
```

否则会显示哪些测试失败的说明。请参阅第 33.2 节中的下文说明，不要过早认定一次“失败”就意味着存在严重问题。

由于这种测试方法会运行一个临时服务器，如果你以 root 用户身份进行构建，它将无法工作，因为服务器不会以 root 身份启动。推荐做法是不要以 root 身份构建；否则应在安装完成后再执行测试。

如果你把 PostgreSQL 配置为安装到某个已经存在旧版 PostgreSQL 安装的位置，并且在安装新版本之前执行 `make check`，可能会发现测试失败，因为新程序会尝试使用已安装的共享库。（典型症状是报出未定义符号。）如果你希望在覆盖旧安装之前运行测试，就需要使用 `configure --disable-rpath` 进行构建。不过，不建议在最终安装时使用这个选项。

并行回归测试会在你的用户 ID 下启动相当多的进程。目前最大并发度为二十个并行测试脚本，也就是四十个进程：每个测试脚本都有一个服务器进程和一个 psql 进程。因此，如果你的系统对每个用户可创建的进程数施加限制，请确保该限制至少有五十个左右，否则并行测试中可能会出现看似随机的失败。如果无法提高该限制，可以通过设置`MAX_CONNECTIONS` 参数来降低并行度。例如：

```
make MAX_CONNECTIONS=10 check
```

这样一次最多只会并发运行十个测试。

33.1.2. 针对现有安装运行测试

要在安装之后运行测试（见第 16 章），请初始化一个数据区域并启动服务器，具体说明见第 18 章，然后输入：

```
make installcheck
```

或者，要并行测试：

```
make installcheck-parallel
```

测试默认通过本地主机和默认端口号连接服务器，除非通过PGHOST 和 PGPORT 环境变量另行指定。测试将在名为 regression 的数据库中运行；任何现有的同名数据库都会被删除。

这些测试还会临时创建一些集簇级对象，例如角色和表空间。这些对象的名称都会以 regress_ 开头。在存在这样命名的真实用户或表空间的安装中，应谨慎使用 installcheck 模式。

33.1.3. 附加测试套件

make check 和 make installcheck 命令只运行“核心”回归测试，这些测试用于检验 PostgreSQL 服务器的内置功能。源码发行版还包含其他测试套件，其中大部分与附加功能有关，例如可选的过程语言。

要运行适用于已选定构建模块的全部测试套件（包括核心测试），请在构建树顶层输入下列命令之一：

```
make check-world
make installcheck-world
```

这些命令分别使用临时服务器或已安装的服务器运行测试，与前面对make check 和 make installcheck 的说明相同。其他注意事项也与前面对各个方法的说明相同。注意，make check-world 会为每个被测试的模块构建一个独立的临时安装树，因此需要多得多的时间和磁盘空间，相比之下耗用较少的是make installcheck-world。

或者，也可以在构建树相应的子目录中执行make check 或 make installcheck 来运行单个测试套件。请记住，make installcheck 假定你已经安装了相关模块，而不仅仅是核心服务器。

可以按这种方式调用的附加测试包括：

- 可选过程语言的回归测试（PL/pgSQL 除外，它由核心测试负责）。这些测试位于 src/pl 下。
- contrib 模块的回归测试，位于 contrib 下。并非所有 contrib 模块都有测试。
- 接口库的回归测试，位于src/interfaces/libpq/test 和 src/interfaces/ecpg/test。
- 针对并发会话行为的压力测试，位于src/test/isolation。
- 客户端程序的测试，位于 src/bin 下。另见第 33.4 节。

使用 installcheck 模式时，这些测试会销毁任何名为 pl_regression、contrib_regression、isolation_regression、ecpg1_regression 或 ecpg2_regression 的现有数据库，以及名为 regression 的现有数据库。

基于 TAP 的测试仅在 PostgreSQL 以 --enable-tap-tests 选项配置时才会运行。对开发而言，建议启用它；但如果没有合适的 Perl 安装，也可以省略。

有些测试套件默认不会运行，原因可能是它们在多用户系统上运行不安全，或者需要特殊软件。你可以通过将make 变量或环境变量 PG_TEST_EXTRA 设置为空白分隔的列表，来决定额外运行哪些测试套件，例如：

```
make check-world PG_TEST_EXTRA='kerberos ldap ssl'
```

目前支持以下取值：

kerberos

运行 src/test/kerberos 下的测试套件。这需要 MIT Kerberos 安装，并会打开 TCP/IP 监听套接字。

ldap

运行 src/test/ldap 下的测试套件。这需要 OpenLDAP 安装，并会打开 TCP/IP 监听套接字。

ssl

运行 src/test/ssl 下的测试套件。这会打开 TCP/IP 监听套接字。

即使在 PG_TEST_EXTRA 中提到了当前构建配置不支持的特性，其测试也不会运行。

33.1.4. 区域设置和编码

默认情况下，使用临时安装的测试会采用当前环境中定义的区域设置，以及由 initdb 决定的相应数据库编码。通过设置适当的环境变量来测试不同的区域设置可能很有用，例如：

```
make check LANG=C
make check LC_COLLATE=en_US.utf8 LC_CTYPE=fr_CA.utf8
```

出于实现原因，设置LC_ALL 不能用于此目的；其他所有与区域设置相关的环境变量都可以。

在针对现有安装测试时，区域设置由现有数据库集簇决定，不能为测试单独设置。

也可以通过设置变量 ENCODING 来显式选择数据库编码，例如：

```
make check LANG=C ENCODING=EUC_JP
```

只有在区域设置为 C 时，这样设置数据库编码通常才有意义；否则编码会从区域设置自动选择，而指定与区域设置不匹配的编码会导致错误。

无论是针对临时安装还是现有安装测试，都可以设置数据库编码；不过在后一种情况下，它必须与该安装的区域设置兼容。

33.1.5. 额外测试

核心回归测试套件包含几个默认不运行的测试文件，因为它们可能依赖于平台，或者运行时间很长。可以通过设置以下变量来运行这些或其他额外的测试文件：`EXTRA_TESTS`。例如，要运行 `numeric_big` 测试：

```
make check EXTRA_TESTS=numeric_big
```

要运行排序规则测试：

```
make check EXTRA_TESTS='collate.linux.utf8 collate.icu.utf8' LANG=en_US.utf8
```

其中，`collate.linux.utf8` 测试仅适用于 Linux/glibc 平台。`collate.icu.utf8` 测试仅在构建时加入 ICU 支持后才能运行。这两个测试都只有在使用 UTF-8 编码的数据库中运行时才会成功。

33.1.6. 测试热备

源代码分发包中还包含针对热备静态行为的回归测试。这些测试需要一个正在运行的主库和一个正在运行的备库，备库正在从主库接收新的 WAL 更改（使用基于文件的日志传送或流复制均可）。这些服务器不会自动创建，复制的设置也不在此处文档说明。请查阅文档中有关所需命令和相关问题的各个章节。

要运行热备测试，首先在主库上创建一个名为`regression` 的数据库：

```
psql -h primary -c "CREATE DATABASE regression"
```

接下来，在主库上的 regression 数据库中运行准备脚本 `src/test/regress/sql/hs_primary_setup.sql`，例如：

```
psql -h primary -f src/test/regress/sql/hs_primary_setup.sql regression
```

等待这些更改传播到备库。

现在将默认数据库连接指向要测试的备库（例如，设置 `PGHOST` 和 `PGPORT` 环境变量）。最后，在回归测试目录中运行 `make standbycheck`：

```
cd src/test/regress
make standbycheck
```

还可以使用脚本 `src/test/regress/sql/hs_primary_extremes.sql` 在主库上生成一些极端行为，以测试备库的行为。

33.2. 测试结果评估

某些安装正确且功能完备的 PostgreSQL 实例，可能会因为平台特有因素而在部分回归测试中“失败”，例如浮点表示或消息措辞不同。目前这些测试是通过把输出与参考系统生成的输出做简单的`diff` 比较来评估的，因此结果会对细微的系统差异很敏感。报告某项测试“失败”时，请务必检查预期结果与实际结果之间的差异；你可能会发现这些差异并不重要。尽管如此，我们仍努力在所有受支持平台上维护准确的参考文件，因此原则上应期待所有测试都能通过。

回归测试的实际输出位于 src/test/regress/results 目录中的文件里。测试脚本使用 diff 将每个输出文件与存放在 src/test/regress/expected 目录中的参考输出进行比较。所有差异都会保存在 src/test/regress/regression.diffs 中供你检查。（运行核心测试以外的测试套件时，这些文件当然会出现在相应的子目录中，而不是src/test/regress。）

如果不喜欢默认使用的 diff 选项，可设置环境变量 PG_REGRESS_DIFF_OPTS，例如 PG_REGRESS_DIFF_OPTS='-u'。（或者如果你愿意，也可以自己运行diff。）

如果由于某种原因，某个特定平台在某个测试上产生了“失败”，但对输出的检查令你相信结果是有效的，那么可以新增一个比较文件，让后续测试运行不再报告该失败。详情见第 33.3 节。

33.2.1. 错误消息差异

某些回归测试包含故意构造的非法输入值。错误消息可能来自 PostgreSQL 代码，也可能来自宿主平台的系统例程。在后一种情况下，消息会因平台而异，但应反映相近的信息。这类消息差异会导致回归测试显示为“失败”，但可以通过检查确认其有效性。

33.2.2. 区域设置差异

如果你针对一个排序区域设置不是 C 的服务器运行测试，就可能因为排序顺序不同而出现差异，并导致后续失败。回归测试套件通过提供备用结果文件来处理这个问题；这些文件已知可以覆盖大量区域设置。

使用临时安装方法时，要在不同区域设置下运行测试，可在 make 命令行上传递合适的区域设置相关环境变量，例如：

```
make check LANG=de_DE.utf8
```

（回归测试驱动器会取消设置 LC_ALL，因此不能用该变量选择区域设置。）如果不使用区域设置，要么取消所有与区域设置相关的环境变量（或将它们设为C），要么使用下面这个特殊调用：

```
make check NO_LOCALE=1
```

针对现有安装运行测试时，区域设置由现有安装决定。要改变它，需要在初始化数据库集簇时向 initdb 传入合适选项，使用另一种区域设置。

一般来说，建议在计划用于生产环境的区域设置下运行回归测试，因为这样可以覆盖实际将在生产中使用的区域设置和编码相关代码部分。根据操作系统环境不同，可能会出现失败，但至少你会知道在真实应用运行时应当预期哪些与区域设置相关的行为。

33.2.3. 日期和时间差异

大多数日期和时间结果依赖于时区环境。参考文件是在时区 PST8PDT（加利福尼亚州伯克利）下生成的，如果测试未在该时区设置下运行，就会出现表面上的失败。回归测试驱动器会将环境变量 PGTZ 设为 PST8PDT，这通常能够确保获得正确结果。

33.2.4. 浮点差异

某些测试涉及根据表列计算 64 位浮点数（double precision）。我们已经观察到涉及 double precision 列数学函数的结果存在差异。float8 和 geometry 测试尤其容易在不同平台之间，甚至在不同编译器优化设置下出现细微差异。要判断这些通常出现在小数点右侧第 10 位附近的差异究竟是否重要，需要人工目测比较。

某些系统把负零显示为-0，而另一些只显示 0。

某些系统对 pow() 和 exp() 发出错误的方式，与当前PostgreSQL 代码所预期的机制不同。

33.2.5. 行顺序差异

你可能会看到这样的差异：同样的行在输出中的顺序与预期文件中的顺序不同。在大多数情况下，严格说来这并不是缺陷。大多数回归测试脚本并没有细致到为每一个SELECT 都使用 ORDER BY，因此按 SQL 规范，它们的结果行顺序并没有明确定义。实际上，由于我们看到的是同一软件在同一数据上执行相同查询，通常在所有平台上都会得到相同的结果顺序，所以缺少 ORDER BY 并不是问题。不过，有些查询确实会表现出跨平台的顺序差异。针对已安装服务器测试时，非 C 区域设置或非默认参数设置，例如自定义的work_mem 值或规划器代价参数，也可能导致顺序差异。

因此，如果你看到顺序差异，一般无需担心，除非查询确实包含 ORDER BY 而你的结果违反了它。不过，仍请报告该问题，这样我们可以为那个特定查询加上ORDER BY，以在后续版本中消除这种虚假的“失败”。

你可能会好奇，为什么我们不显式地为所有回归测试查询排序，从而一劳永逸地解决这个问题。原因在于，那样反而会降低回归测试的价值，因为测试会倾向于覆盖能产生有序结果的查询计划类型，而排除那些不能产生有序结果的计划类型。

33.2.6. 栈深度不足

如果 errors 测试在执行 select infinite_recurse() 命令时导致服务器崩溃，这意味着平台对进程栈大小的限制比max_stack_depth 参数所表明的更小。可通过在更高的栈大小限制下运行服务器来修复（对于 max_stack_depth 的默认值，建议 4MB）。如果做不到，另一种办法是减小max_stack_depth 的值。

在支持 getrlimit() 的平台上，服务器应自动选择 max_stack_depth 的安全值；因此，除非你手工覆盖了该设置，否则这类失败就是一个应报告的缺陷。

33.2.7. “random” 测试

random 测试脚本本来就是要产生随机结果的。在极少数情况下，这会导致该回归测试失败。输入：

```
diff results/random.out expected/random.out
```

只应产生一行或几行差异。除非 random 测试反复失败，否则不必担心。

33.2.8. 配置参数

在针对现有安装运行测试时，某些非默认参数设置可能导致测试失败。例如，改变 enable_seqscan 或 enable_indexscan 等参数，可能导致计划发生变化，进而影响使用 EXPLAIN 的测试结果。

33.3. 变体比较文件

由于某些测试天生会产生依赖环境的结果，我们提供了指定备用“预期”结果文件的方法。每个回归测试都可以有多个比较文件，用来展示不同平台上的可能结果。对于每个测试应使用哪个比较文件，有两种相互独立的判定机制。

第一种机制允许针对特定平台选择比较文件。有一个映射文件 src/test/regress/resultmap，用于定义每个平台应使用哪个比较文件。要为某个特定平台消除虚假的测试“失败”，你需要先选择或创建一个变体结果文件，然后在resultmap 文件中增加一行。

映射文件中的每一行都采用如下形式：

```
testname:output:platformpattern=comparisonfilename
```

测试名就是相应回归测试模块的名称。输出值指明要检查哪个输出文件。对标准回归测试而言，这里始终是out。该值对应于输出文件的扩展名。平台模式是 Unix 工具 expr 风格的模式（也就是在开头隐含^锚点的正则表达式）。它会与 config.guess 打印出的平台名进行匹配。比较文件名则是替代结果比较文件的基名。

例如：某些系统会把极小的浮点数值解释为零，而不是报告下溢错误。这会导致float8 回归测试出现一些差异。因此，我们提供了一个变体比较文件float8-small-is-zero.out，其中包含这些系统上的预期结果。为了消除虚假的“失败”消息，在OpenBSD 平台上，resultmap 包含以下内容：

```
float8:out:i.86-.*-openbsd=float8-small-is-zero.out
```

如果一台机器上的 config.guess 输出匹配 i.86-.*-openbsd，就会触发该规则。resultmap 中的其他行会为其他适用的平台选择变体比较文件。

第二种用于选择变体比较文件的机制更加自动化：它只是在多个提供的比较文件中选取“最佳匹配”。回归测试驱动脚本既考虑某个测试的标准比较文件 *testname*.out，也考虑名为 *testname_digit*.out 的变体文件（其中 *digit* 可以是任一单个数字 0-9）。如果其中任何一个文件完全匹配，就认为测试通过；否则，会使用生成最短 diff 的那个文件来创建失败报告。（如果resultmap 为该特定测试包含一项，那么基础*testname* 就是 resultmap 中给出的替代名称。）

例如，对于char 测试，比较文件 char.out 包含在 C 和 POSIX 区域设置中应当出现的结果，而文件 char_1.out 则包含在许多其他区域设置中出现的排序结果。

最佳匹配机制原本是为了解决依赖区域设置的结果问题，但它也可以用于任何仅凭平台名无法轻易预测测试结果的场景。该机制的一个局限是，测试驱动器无法判断哪一个变体对当前环境才真正“正确”；它只会选择看起来最合适的那个。因此，最安全的做法是只对那些你愿意在所有上下文中都视为同样有效的变体结果使用这种机制。

33.4. TAP 测试

各种测试，尤其是src/bin 下的客户端程序测试，会使用 Perl TAP 工具，并通过 Perl 测试程序 prove 运行。你可以通过设置 make 变量 PROVE_FLAGS 来向 prove 传递命令行选项，例如：

```
make -C src/bin check PROVE_FLAGS='--timer'
```

更多信息请参见 prove 的手册页。

make 变量 PROVE_TESTS 可用于定义一个以空白分隔的路径列表，这些路径相对于 Makefile；该文件调用 prove 来运行指定的测试子集，而不是默认的 t/*.pl。例如：

```
make check PROVE_TESTS='t/001_test1.pl t/003_test3.pl'
```

TAP 测试需要 Perl 模块 IPC::Run。该模块可从 CPAN 或操作系统软件包获得。此外，这些测试还要求 PostgreSQL 在配置时使用 --enable-tap-tests 选项。

33.5. 测试覆盖率检查

可以在编译 PostgreSQL 源代码时加入覆盖率测试插桩，从而检查回归测试或任何其他与该代码一起运行的测试套件覆盖了代码的哪些部分。目前，这只在使用 GCC 编译时受支持，并且需要 `gcov` 和 `lcov` 程序。

一个典型的工作流程如下：

```
./configure --enable-coverage ... OTHER OPTIONS ...
make
make check # or other test suite
make coverage-html
```

然后将 HTML 浏览器指向 `coverage/index.html`。这些 `make` 命令在子目录中也可以使用。

如果没有 `lcov`，或者比起 HTML 报告你更喜欢文本输出，也可以运行

```
make coverage
```

来代替 `make coverage-html`，这样会为与测试相关的每个源文件生成 `.gcov` 输出文件。（`make coverage` 和 `make coverage-html` 会覆盖彼此的文件，因此混用它们可能造成混淆。）

要在各次测试运行之间重置执行计数，请运行：

```
make coverage-clean
```

部分 IV. 客户端接口

本部分描述随 PostgreSQL 一起发布的客户端编程接口。这些章节彼此独立，可以分别阅读。还有许多面向客户端程序的外部编程接口是单独发布的，它们各自也附带文档（附录 H 列出了一些较常用的接口）。本部分的读者应熟悉如何使用 SQL 操作和查询数据库（见第 II 部分），当然还应熟悉相应接口所使用的编程语言。

目录

34. libpq - C 库 781
34.1. 数据库连接控制函数 781
34.1.1. 连接字符串 788
34.1.2. 参数关键词 790
34.2. 连接状态函数 794
34.3. 命令执行函数 801
34.3.1. 主要函数 801
34.3.2. 检索查询结果信息 809
34.3.3. 检索其他结果信息 813
34.3.4. 用于在 SQL 命令中嵌入字符串的转义 814
34.4. 异步命令处理 817
34.5. 逐行检索查询结果 821
34.6. 取消进行中的查询 822
34.7. 快速路径接口 823
34.8. 异步通知 824
34.9. COPY 命令相关的函数 825
34.9.1. 用于发送 COPY 数据的函数 826
34.9.2. 用于接收 COPY 数据的函数 826
34.9.3. 用于 COPY 的过时函数 827
34.10. 控制函数 829
34.11. 杂项函数 831
34.12. 通知处理 834
34.13. 事件系统 835
34.13.1. 事件类型 835
34.13.2. 事件回调过程 837
34.13.3. 事件支持函数 838
34.13.4. 事件示例 839
34.14. 环境变量 841
34.15. 密码文件 843
34.16. 连接服务文件 843
34.17. 连接参数的 LDAP 查找 844
34.18. SSL 支持 845
34.18.1. 客户端对服务器证书的验证 845
34.18.2. 客户端证书 846
34.18.3. 不同模式中提供的保护 846
34.18.4. SSL 客户端文件使用 847
34.18.5. SSL 库初始化 847
34.19. 多线程程序中的行为 848
34.20. 构建 libpq 程序 849
34.21. 示例程序 850
35. 大对象 860
35.1. 简介 860
35.2. 实现特性 860
35.3. 客户端接口 860
35.3.1. 创建一个大对象 861
35.3.2. 导入一个大对象 861
35.3.3. 导出一个大对象 862
35.3.4. 打开一个现有的大对象 862
35.3.5. 向大对象写入数据 862
35.3.6. 从大对象读取数据 863
35.3.7. 在大对象中定位 863

35.3.8. 获取大对象的当前位置 863
35.3.9. 截断一个大对象 864
35.3.10. 关闭一个大对象描述符 864
35.3.11. 移除一个大对象 864
35.4. 服务器端函数 864
35.5. 示例程序 866
36. ECPG - C 中的嵌入式 SQL 871
36.1. 概念 871
36.2. 管理数据库连接 871
36.2.1. 连接到数据库服务器 872
36.2.2. 选择一个连接 873
36.2.3. 关闭一个连接 874
36.3. 运行 SQL 命令 875
36.3.1. 执行 SQL 语句 875
36.3.2. 使用游标 875
36.3.3. 管理事务 876
36.3.4. 预备语句 877
36.4. 使用主变量 877
36.4.1. 概述 878
36.4.2. 声明节 878
36.4.3. 检索查询结果 878
36.4.4. 类型映射 879
36.4.5. 处理非简单 SQL 数据类型 887
36.4.6. 指示符 892
36.5. 动态 SQL 893
36.5.1. 执行没有结果集的语句 893
36.5.2. 执行一个有输入参数的语句 893
36.5.3. 执行一个有结果集的语句 894
36.6. pgtypes 库 895
36.6.1. 字符串 895
36.6.2. numeric 类型 895
36.6.3. date 类型 899
36.6.4. timestamp 类型 903
36.6.5. interval 类型 907
36.6.6. decimal 类型 908
36.6.7. pgtypeslib 的 errno 值 908
36.6.8. pgtypeslib 的特殊常量 909
36.7. 使用描述符区域 909
36.7.1. 命名 SQL 描述符区域 910
36.7.2. SQLDA 描述符区域 912
36.8. 错误处理 924
36.8.1. 设置回调 924
36.8.2. sqlca 927
36.8.3. SQLSTATE 与 SQLCODE 928
36.9. 预处理器指令 932
36.9.1. 包含文件 932
36.9.2. define 和 undef 指令 933
36.9.3. ifdef、ifndef、else、elif 和 endif 指令 933
36.10. 处理嵌入式 SQL 程序 934
36.11. 库函数 935
36.12. 大对象 936
36.13. C++ 应用 937
36.13.1. 主变量的作用域 937

36.13.2. 使用外部 C 模块的 C++ 应用开发 938
36.14. 嵌入式 SQL 命令 941
36.15. Informix 兼容模式 963
36.15.1. 附加类型 964
36.15.2. 附加的/缺少的嵌入式 SQL 语句 964
36.15.3. Informix 兼容的 SQLDA 描述符区域 965
36.15.4. 附加函数 968
36.15.5. 额外的常量 979
36.16. Oracle 兼容模式 980
36.17. 内部 981
37. 信息模式 983
37.1. 模式 983
37.2. 数据类型 983
37.3. information_schema_catalog_name 984
37.4. administrable_role_authorizations 984
37.5. applicable_roles 984
37.6. attributes 985
37.7. character_sets 987
37.8. check_constraint_routine_usage 988
37.9. check_constraints 988
37.10. collations 988
37.11. collation_character_set_applicability 989
37.12. column_domain_usage 989
37.13. column_options 990
37.14. column_privileges 990
37.15. column_udt_usage 990
37.16. columns 991
37.17. constraint_column_usage 994
37.18. constraint_table_usage 994
37.19. data_type_privileges 995
37.20. domain_constraints 995
37.21. domain_udt_usage 996
37.22. domains 996
37.23. element_types 998
37.24. enabled_roles 1000
37.25. foreign_data_wrapper_options 1000
37.26. foreign_data_wrappers 1001
37.27. foreign_server_options 1001
37.28. foreign_servers 1001
37.29. foreign_table_options 1002
37.30. foreign_tables 1002
37.31. key_column_usage 1002
37.32. parameters 1003
37.33. referential_constraints 1004
37.34. role_column_grants 1005
37.35. role_routine_grants 1005
37.36. role_table_grants 1006
37.37. role_udt_grants 1007
37.38. role_usage_grants 1007
37.39. routine_privileges 1007
37.40. routines 1008
37.41. schemata 1012

37.42. sequences 1012
37.43. sql_features 1013
37.44. sql_implementation_info 1013
37.45. sql_languages 1014
37.46. sql_packages 1014
37.47. sql_parts 1015
37.48. sql_sizing 1015
37.49. sql_sizing_profiles 1015
37.50. table_constraints 1016
37.51. table_privileges 1016
37.52. tables 1017
37.53. transforms 1018
37.54. triggered_update_columns 1018
37.55. triggers 1019
37.56. udt_privileges 1020
37.57. usage_privileges 1020
37.58. user_defined_types 1021
37.59. user_mapping_options 1022
37.60. user_mappings 1022
37.61. view_column_usage 1023
37.62. view_routine_usage 1023
37.63. view_table_usage 1024
37.64. views 1024

第 34 章 libpq - C 库

libpq 是 PostgreSQL 的 C 应用程序编程接口。libpq 是一组库函数，客户端程序可用它们向 PostgreSQL 后端服务器发送查询并接收查询结果。

libpq 也是其他几个 PostgreSQL 应用程序接口的底层引擎，包括为 C++、Perl、Python、Tcl 和ECPG 编写的接口。因此，对于这些包的用户，libpq 行为的某些方面也很重要。特别是，第 34.14 节、第 34.15 节和第 34.18 节描述了任何使用 libpq 的应用程序的用户都能观察到的行为。

本章末尾（第 34.21 节）包含一些简短程序，展示如何编写使用libpq 的程序。源代码发行包的 `src/test/examples` 目录中还提供了几个完整的 libpq 应用程序示例。

使用 libpq 的客户端程序必须包含头文件 `libpq-fe.h`，并且必须与libpq 库链接。

34.1. 数据库连接控制函数

以下函数用于建立到 PostgreSQL 后端服务器的连接。应用程序可以同时保持多个后端连接。（这样做的原因之一是访问多个数据库。）每个连接由一个 `PGconn` 对象表示，该对象可以通过以下函数获取：`PQconnectdb`，`PQconnectdbParams`，或`PQsetdbLogin`。注意，这些函数总是返回非空的对象指针，除非内存不足，甚至无法分配`PGconn` 对象。应调用 `PQstatus` 函数检查返回值，确认连接成功后，再通过连接对象发送查询。

> **警告**
>
> 如果不受信任的用户能够访问一个没有采用模式的安全使用方式的数据库，那么每个会话开始时都应从`search_path` 中移除公开可写的模式。可以把参数关键词 `options` 设置为`-csearch_path=`。也可以在连接后发出 `PQexec(`*`conn`*`, "SELECT pg_catalog.set_config('search_path', '', false)")`。这种考虑并非专门针对 libpq；它适用于每一种可执行任意 SQL 命令的接口。

> **警告**
>
> 在 Unix 上，对持有已打开 libpq 连接的进程执行 fork 操作可能导致不可预料的结果，因为父进程和子进程会共享相同的套接字和操作系统资源。出于这个原因，我们不推荐这样的用法，尽管从子进程执行一个`exec` 来载入新的可执行程序是安全的。

`PQconnectdbParams`

开启一个到数据库服务器的新连接。

```
PGconn *PQconnectdbParams(const char * const *keywords,
                          const char * const *values,
                          int expand_dbname);
```

这个函数使用从两个以 `NULL` 结尾的数组中取得的参数打开一个新的数据库连接。第一个数组 `keywords` 是一个字符串数组，其中每个元素都是一个关键词。第二个数组 `values` 给出每个关键词的值。和下面的 `PQsetdbLogin` 不同，参数集合可以在不改变

函数签名的情况下扩展，因此对于新应用，最好使用这个函数（或者相应的非阻塞函数 PQconnectStartParams 和 PQconnectPoll）。

当前能被识别的参数关键词被列举在第 34.1.2 节中。

传入的数组可以为空，以使用所有默认参数，也可以包含一个或多个参数设置。两个数组的长度必须相同。处理会在 keywords 数组的第一个 NULL 元素处停止。如果某个非 NULL 的 keywords 元素所对应的 values 元素为 NULL 或空字符串，则忽略这一项，继续处理下一对数组元素。

当 expand_dbname 非零时，会检查第一个 *dbname* 关键词的值是否为连接字符串。如果是，就将其“展开”为从该字符串中提取的各个连接参数。如果该值包含等号（=），或以 URI 方案标识符开头，就会将其视为连接字符串，而非单纯的数据库名。（连接字符串格式的详细说明见第 34.1.1 节。）只有第一次出现的 *dbname* 会按这种方式处理；后续的 *dbname* 参数都作为普通数据库名处理。

通常会从头到尾处理参数数组。如果某个关键词重复出现，则采用最后一个非 NULL 且非空的值。此规则也适用于连接字符串中的关键词与 keywords 数组中的关键词冲突的情况。因此，程序员可以决定数组元素是覆盖连接字符串中的值，还是被这些值覆盖。出现在要展开的 *dbname* 元素之前的数组元素，可以被连接字符串中的字段覆盖；而这些字段又会被出现在 *dbname* 之后的数组元素覆盖（同样，只有这些元素提供非空值时才会覆盖）。

处理完所有数组元素及展开的连接字符串后，仍未设置的连接参数将填入默认值。如果某个未设置参数对应的环境变量（见第 34.14 节）已经设置，就使用该环境变量的值；否则使用该参数的内置默认值。

PQconnectdb

开启一个到数据库服务器的新连接。

```
PGconn *PQconnectdb(const char *conninfo);
```

这个函数使用从字符串 conninfo 中得到的参数开启一个新的数据库连接。

被传递的字符串可以为空，这样将会使用所有的默认参数。也可以包含由空白分隔的一个或多个参数设置，还可以包含一个URI。详见第 34.1.1 节。

PQsetdbLogin

开启一个到数据库服务器的新连接。

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

这是 PQconnectdb 的前身，使用固定的一组参数。除缺失参数始终采用默认值之外，功能相同。对于要使用默认值的任意固定参数，请传入 NULL 或空字符串。

如果 *dbName* 包含 = 符号，或具有有效的连接 URI 前缀，就会将其当作 *conninfo* 字符串处理，方式与将其传给 PQconnectdb 完全相同，然后按照 PQconnectdbParams 的规则应用其余参数。

PQsetdb

开启一个到数据库服务器的新连接。

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

这是一个调用 PQsetdbLogin 的宏，其中为*login* 和 *pwd* 参数使用空指针。提供它是为了向后兼容非常老的程序。

PQconnectStartParams
PQconnectStart
PQconnectPoll

以非阻塞的方式建立一个到数据库服务器的连接。

```
PGconn *PQconnectStartParams(const char * const *keywords,
                             const char * const *values,
                             int expand_dbname);

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

这三个函数被用来开启一个到数据库服务器的连接，这样你的应用的执行线程不会因为远程的 I/O 而被阻塞。这种方法的要点在于等待 I/O 完成可能在应用的主循环中发生，而不是在 PQconnectdbParams 或 PQconnectdb 中，并且因此应用能够把这种操作和其他动作并行处理。

在 PQconnectStartParams 中，数据库连接使用从keywords 和 values 数组中取得的参数创建，并且被expand_dbname 控制，这和之前描述的PQconnectdbParams 相同。

在 PQconnectStart 中，数据库连接使用从字符串conninfo 中取得的参数创建，这和之前描述的PQconnectdb 相同。

无论是 PQconnectStartParams 还是 PQconnectStart 还是 PQconnectPoll 都不会阻塞，只要满足以下限制：

- 必须正确使用 hostaddr 参数，以避免执行 DNS 查询。详细信息请参见第 34.1.2 节中该参数的说明。
- 如果你调用 PQtrace，确保接收追踪输出的流对象不会阻塞。
- 如后文所述，你必须要确保在调用PQconnectPoll 之前，套接字处于合适的状态。

要开始非阻塞连接请求，可调用PQconnectStart 或者 PQconnectStartParams。如果结果为空指针，则libpq 无法分配一个新的 PGconn 结构体。否则，一个有效的PGconn 指针会被返回（不过还没有表示一个到数据库的有效连接）。接下来调用 PQstatus(conn)。如果结果是 CONNECTION_BAD，则连接尝试已经失败，通常是因为有无效的连接参数。

如果 PQconnectStart 或 PQconnectStartParams 成功，下一个阶段是轮询libpq，这样它能够继续进行连接序列。使用PQsocket(conn) 来获得该数据库连接底层的套接字描述符（警告：不要假定在 PQconnectPoll 调用之间套接字会保持相同）。这样循环：如果 PQconnectPoll(conn) 上一次返回 PGRES_POLLING_READING，等到该套接字准备好读取（按照select()、poll() 或类似的系统函数所指示的）。则再次调用 PQconnectPoll(conn)。反之，如果 PQconnectPoll(conn) 上一次返回 PGRES_POLLING_WRITING，等到该套接字准备好写入，则再次调用PQconnectPoll(conn)。在第一次迭代时，即如果你还没有调用 PQconnectPoll，行为就像是它上次返回了PGRES_POLLING_WRITING。持续这个循环直到 PQconnectPoll(conn) 返回 PGRES_POLLING_FAILED 指示连接过程已经失败，或者返回PGRES_POLLING_OK 指示连接已经被成功地建立。

在连接过程中的任何时刻，都可以通过调用PQstatus 来检查连接状态。如果该调用返回 CONNECTION_BAD，则连接过程已经失败；如果返回CONNECTION_OK，则连接已就绪。通过以下函数的返回值也同样可以检测这两种状态：PQconnectPoll，该函数已在上文介绍。在异步连接过程中（也仅在此过程中）还可能出现其他状态。它们指明连接过程的当前阶段，例如可以用来向用户提供反馈。这些状态如下：

CONNECTION_STARTED

等待连接被建立。

CONNECTION_MADE

连接 OK，等待发送。

CONNECTION_AWAITING_RESPONSE

等待来自服务器的一个回应。

CONNECTION_AUTH_OK

收到认证，等待后端启动结束。

CONNECTION_SSL_STARTUP

协商 SSL 加密。

CONNECTION_SETENV

协商环境驱动的参数设置。

CONNECTION_CHECK_WRITABLE

检查连接是否能够处理写事务。

CONNECTION_CONSUME

消费连接上的任何剩下的响应消息。

注意，虽然这些常量会保留下来以维持兼容性，但应用程序绝不能依赖它们按某种特定顺序出现、必定出现，或状态始终是这些已记录的值之一。应用程序可以采用类似以下的做法：

```
switch(PQstatus(conn))
{
        case CONNECTION_STARTED:
            feedback = "Connecting...";
            break;

        case CONNECTION_MADE:
            feedback = "Connected to server...";
            break;
.
.
.
        default:
            feedback = "Connecting...";
}
```

在使用 PQconnectPoll 时，连接参数connect_timeout 会被忽略：判断是否超时是应用的责任。除此之外，PQconnectStart 后面跟着 PQconnectPoll 循环等效于 PQconnectdb。

注意当 PQconnectStart 或 PQconnectStartParams 返回一个非空的指针时，你必须在用完它之后调用PQfinish 来释放该结构体及其关联的所有内存块。即使连接尝试失败或被放弃时也必须完成这些工作。

PQconndefaults

返回默认连接选项。

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
    char   *keyword;   /* 该选项的关键词 */
    char   *envvar;    /* 后备环境变量名 */
    char   *compiled;  /* 编译时设置的后备默认值 */
    char   *val;       /* 选项的当前值，或者 NULL */
    char   *label;     /* 连接对话框中字段的标签 */
    char   *dispchar;  /* 指示如何在连接对话框中显示此字段。可取值:
                          ""          显示输入的值
                          "*"         密码字段 – 隐藏值
                          "D"         调试选项 – 默认不显示 */
    int     dispsize;  /* 对话框中的字段宽度，以字符计 */
} PQconninfoOption;
```

返回一个连接选项数组。这可以用来确定用于连接服务器的所有可能的 PQconnectdb 选项和它们的当前默认值。返回值指向一个 PQconninfoOption 结构体的数组，该数组以一个包含空keyword 指针的条目结束。如果无法分配内存，则返回空指针。注意当前默认值（val 字段）将依赖于环境变量和其他上下文。一个缺失或者无效的服务文件将会被无声地忽略掉。调用者必须把连接选项当作只读对待。

在处理完选项数组后，把它交给PQconninfoFree 释放。如果没有这么做，每次调用 PQconndefaults 都会导致一小部分内存泄漏。

PQconninfo

返回被一个活动连接使用的连接选项。

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

返回一个连接选项数组。可以用它确定所有可能的 PQconnectdb 选项，以及实际用于连接服务器的值。返回值指向一个 PQconninfoOption 结构体数组，该数组以 keyword 指针为空的条目结束。上文针对 PQconndefaults 的所有注意事项，也适用于 PQconninfo 的结果。

PQconninfoParse

返回从提供的连接字符串中解析到的连接选项。

```
PQconninfoOption *PQconninfoParse(const char *conninfo, char **errmsg);
```

解析一个连接字符串并且将结果选项作为一个数组返回，或者在连接字符串有问题时返回 NULL。这个函数可以用来抽取所提供的连接字符串中的 PQconnectdb 选项。返回值指向一个 PQconninfoOption 结构体的数组，该数组以一个包含空keyword 指针的条目结束。

所有合法选项将出现在结果数组中，但是任何在连接字符串中没有出现的选项的 PQconninfoOption 的 val 会被设置为 NULL，默认值不会被插入。

如果 errmsg 不是 NULL，则成功时将 *errmsg 设为 NULL；失败时将其设为由 malloc 分配的、用于说明问题的错误字符串。（也可能出现 *errmsg 被设为 NULL，同时函数返回 NULL 的情况；这表示内存不足。）

在处理完选项数组后，把它交给PQconninfoFree 释放。如果没有这么做，每次调用 PQconninfoParse 都会导致一小部分内存泄漏。反过来，如果发生一个错误并且errmsg 不是 NULL，确保使用PQfreemem 释放错误字符串。

PQfinish

关闭与服务器的连接。同时释放 PGconn 对象使用的内存。

```
void PQfinish(PGconn *conn);
```

注意，即使与服务器的连接尝试失败（由PQstatus 指示），应用也应当调用PQfinish 来释放 PGconn 对象使用的内存。不能在调用 PQfinish 之后再使用 PGconn 指针。

PQreset

重置与服务器的通信通道。

```
void PQreset(PGconn *conn);
```

此函数将关闭与服务器的连接，并尝试建立一个新连接，用之前使用过的所有参数。这可能有助于在工作连接丢失后的错误恢复。

PQresetStart
PQresetPoll

以非阻塞方式重置与服务器的通信通道。

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

这些函数会关闭与服务器的连接，并尝试使用与之前完全相同的参数建立新连接。如果原本可用的连接丢失，这可以用于错误恢复。它们与上文的 PQreset 不同之处在于采用非阻塞方式。它们受到与 PQconnectStartParams、PQconnectStart 和 PQconnectPoll 相同的限制。

要开始重置连接，请调用 PQresetStart。如果返回 0，表示重置失败。如果返回 1，则使用 PQresetPoll 轮询重置过程，方式与使用 PQconnectPoll 建立连接完全相同。

PQpingParams

PQpingParams 报告服务器状态。它接受的连接参数与上文介绍的 PQconnectdbParams 相同。取得服务器状态不需要提供正确的用户名、密码或数据库名；但如果提供的值不正确，服务器会记录一次失败的连接尝试。

```
PGPing PQpingParams(const char * const *keywords,
                    const char * const *values,
                    int expand_dbname);
```

该函数返回以下值之一：

PQPING_OK

服务器正在运行，并且看起来可以接受连接。

PQPING_REJECT

服务器正在运行，但是处于一种不允许连接的状态（启动、关闭或崩溃恢复）。

PQPING_NO_RESPONSE

无法联系到服务器。这可能表示服务器没有运行，或者给定的连接参数中有些错误（例如，错误的端口号），或者有一个网络连接问题（例如，一个防火墙阻断了连接请求）。

PQPING_NO_ATTEMPT

没有尝试联系服务器，因为提供的参数显然不正确，或者有一些客户端问题（例如，内存用完）。

`PQping`

`PQping` 报告服务器状态。它接受的连接参数与上文介绍的 `PQconnectdb` 相同。取得服务器状态不需要提供正确的用户名、密码或数据库名；但如果提供的值不正确，服务器会记录一次失败的连接尝试。

```
PGPing PQping(const char *conninfo);
```

返回值与 `PQpingParams` 相同。

34.1.1. 连接字符串

几个 libpq 函数解析用户指定的字符串以获取连接参数。这些字符串有两种被接受的格式：普通的关键词/值字符串和 URI。URI 通常遵循 RFC 3986[1]，但也允许使用多主机连接字符串，详见下文。

34.1.1.1. 关键词/值连接字符串

在关键词/值格式中，每一个参数设置的形式都是关键词 = 值，设置之间以空格分隔。设置的等号周围的空格是可选的。要写一个空值或一个包含空格的值，将它用单引号包围，例如`keyword = 'a value'`。值里面的单引号和反斜杠必须用一个反斜杠转义，即`\'`和`\\`。

示例：

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

能被识别的参数关键词在第 34.1.2 节中列出。

34.1.1.2. 连接 URI

一个连接 URI 的一般形式是：

```
postgresql://[userspec@][hostspec][/dbname][?paramspec]
```

其中 *userspec* 为：

```
user[:password]
```

hostspec 为：

```
[host][:port][,...]
```

paramspec 为：

```
name=value[&...]
```

[1] https://tools.ietf.org/html/rfc3986

URI 方案标识符可以是 `postgresql://`或 `postgres://`。每一个剩下的 URI 部分都是可选的。下列示例展示了合法的 URI 语法：

```
postgresql://
postgresql://localhost
postgresql://localhost:5433
postgresql://localhost/mydb
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
postgresql://host1:123,host2:456/somedb?target_session_attrs=any&application_name=myapp
```

通常出现在 URI 的层次部分的值，也能够以命名参数的方式给出。例如：

```
postgresql:///mydb?host=localhost&port=5433
```

所有命名参数都必须与第 34.1.2 节中列出的关键词匹配；唯一的例外是，为兼容 JDBC 连接 URI，会将 `ssl=true` 转换为 `sslmode=require`。

可以在 URI 的任意部分使用百分号编码来包含具有特殊含义的符号，例如将 = 替换为 `%3D`。

主机部分可能是主机名或一个 IP 地址。要指定一个 IPv6 地址，将它封闭在方括号中：

```
postgresql://[2001:db8::1234]/database
```

主机组件会被按照参数 host 对应的描述来解释。特别地，如果主机部分是空或看起来像一个绝对路径名称，将使用一个 Unix 域套接字连接，否则将启动一个 TCP/IP 连接。不过要注意，斜线是 URI 层次部分中的一个保留字符。因此，要指定一个非标准的 Unix 域套接字目录，要么省略 URI 中的主机部分并且指定该主机为一个命名参数，要么在 URI 的主机部分用百分号编码路径：

```
postgresql:///dbname?host=/var/lib/postgresql
postgresql://%2Fvar%2Flib%2Fpostgresql/dbname
```

可以在一个 URI 中指定多个主机，每一个都有一个可选的端口。一个形式为`postgresql://host1:port1,host2:port2,host3:port3/`的 URI 等效于 `host=host1,host2,host3 port=port1,port2,port3` 形式的连接字符串。如下所述，每一个主机都将被依次尝试，直到成功地建立一个连接。

34.1.1.3. 指定多个主机

可以指定多个要连接的主机，这样它们会按给定的顺序被尝试。在关键词/值格式中，`host`、`hostaddr` 和 `port` 选项都接受逗号分隔的值列表。在指定的每一个选项中都必须给出相同数量的元素，这样第一个`hostaddr` 对应于第一个主机名，第二个`hostaddr` 对应于第二个主机名，以此类推。不过，如果仅指定一个`port`，它将被应用于所有的主机。

在连接 URI 格式中，在 URI 的`host` 部分我们可以列出多个由逗号分隔的 `host:port` 对。

不管是哪一种格式，单一的主机名可以被解析成多个网络地址。常见的示例是一个主机同时具有 IPv4 和 IPv6 地址。

当多个主机被指定时或者单个主机名被解析成多个地址时，所有的主机和地址都将按照顺序被尝试，直至遇到一个成功的。如果没有主机可以到达，则连接失败。如果成功地建立一个连接但是认证失败，也不会尝试列表中剩下的主机。

如果使用了密码文件，可以为不同的主机使用不同的密码。所有其他连接选项对列表中的每一台主机都是相同的，例如不能为不同的主机指定不同的用户名。

34.1.2. 参数关键词

当前识别的参数关键字如下：

`host`

要连接的主机名。如果主机名以斜杠开头，则指定的是 Unix 域通信，而非 TCP/IP 通信；此值是存放套接字文件的目录名。当未指定 `host` 或其值为空时，默认连接到 `/tmp`（或构建 PostgreSQL 时指定的套接字目录）中的 Unix 域套接字。在没有 Unix 域套接字的机器上，默认连接到 `localhost`。

也可以接受一个逗号分隔的主机名列表，此时列表中的每个主机名将按顺序尝试；列表中的空项将选择上述默认行为。详细信息请参见第 34.1.1.3 节。

`hostaddr`

要连接的主机的数字 IP 地址。应采用标准的 IPv4 地址格式，例如 `172.28.40.9`。如果机器支持 IPv6，也可以使用 IPv6 地址。只要此参数指定了非空字符串，就始终使用 TCP/IP 通信。

使用 `hostaddr` 代替 `host` 可以避免主机名查找，这对于有时间限制的应用程序可能很重要。但是，采用 GSSAPI 或 SSPI 认证方式，以及进行`verify-full`SSL 证书验证时，都需要主机名。遵循以下规则：

- 如果指定了 `host` 而没有指定 `hostaddr`，则会发生主机名查找。（当使用 `PQconnectPoll` 时，查找发生在`PQconnectPoll` 首次考虑此主机名时，并且可能导致 `PQconnectPoll` 阻塞相当长的时间。）
- 如果指定了 `hostaddr` 而没有指定 `host`，则`hostaddr` 的值给出服务器的网络地址。如果认证方法需要主机名，则连接尝试将失败。
- 如果同时指定了 `host` 和 `hostaddr`，则 `hostaddr` 的值给出服务器的网络地址。只有认证方法需要主机名时，才会将 `host` 的值用作主机名；否则忽略该值。

注意，以下情况很可能导致认证失败：`host` 不是位于网络地址 `hostaddr` 的服务器名称。另外，同时指定`host` 和 `hostaddr` 时，将使用`host` 来标识密码文件中的连接（参见第 34.15 节）。

也可以接受一个逗号分隔的 `hostaddr` 值列表，此时将按顺序尝试列表中的每个主机。列表中的空项会导致使用相应的主机名，如果主机名也为空，则使用默认主机名。详见第 34.1.1.3 节。

如果既没有主机名也没有主机地址，libpq 会使用本地 Unix 域套接字连接；在没有 Unix 域套接字的机器上，则会尝试连接到 `localhost`。

`port`

连接到服务器主机的端口号，或者 Unix 域连接的套接字文件名扩展。如果在 `host` 或 `hostaddr` 参数中给出了多个主机，则此参数可以指定与主机列表长度相同的逗号分隔的端口列表，或者可以指定用于所有主机的单个端口号。空字符串，或逗号分隔列表中的空项，指定了在构建PostgreSQL 时建立的默认端口号。

dbname

数据库名称。默认为与用户名相同。在某些情况下，该值会被检查是否为扩展格式；有关更多详细信息，请参阅第 34.1.1 节。

user

建立连接所用的 PostgreSQL 用户名。默认与运行应用程序的操作系统用户名相同。

password

服务器要求密码认证时所使用的密码。

passfile

指定用于存储密码的文件名（参见第 34.15 节）。默认为~/.pgpass，或在 Microsoft Windows 上为%APPDATA%\postgresql\pgpass.conf。（如果此文件不存在，则不会报告错误。）

connect_timeout

连接时的最长等待时间，以秒为单位（写成十进制整数，例如，10）。零、负值或未指定表示无限等待。最小允许的超时时间为 2 秒，因此 1 的值被解释为 2。此超时时间分别适用于每个主机名或 IP 地址。例如，如果指定了两个主机和connect_timeout 为 5，如果在 5 秒内没有建立连接，每个主机将超时，因此等待连接的总时间可能长达 10 秒。

client_encoding

这将为此连接设置 client_encoding 配置参数。除了对应服务器选项接受的值外，您还可以使用auto 来从客户端的当前区域设置（Unix 系统上的 LC_CTYPE 环境变量）确定正确的编码。

options

指定连接开始时发送到服务器的命令行选项。例如，将其设置为-c geqo=off 会把会话的 geqo 参数值设为 off。此字符串中的空格被视为分隔命令行参数，除非用反斜杠（\）转义；写\\ 表示字面上的反斜杠。有关可用选项的详细讨论，请参阅第 19 章。

application_name

指定 application_name 配置参数的值。

fallback_application_name

指定 application_name 配置参数的后备值。如果没有通过连接参数或 PGAPPNAME 环境变量为 application_name 指定值，则将使用此值。在通用实用程序中指定后备名称很有用，该程序希望设置默认应用程序名称，但允许用户覆盖它。

keepalives

控制是否使用客户端 TCP keepalive。默认值为 1，表示开启；如果不需要 keepalive，可以将其设为 0，表示关闭。对于通过 Unix 域套接字建立的连接，此参数会被忽略。

keepalives_idle

控制在多久没有活动后，TCP 应向服务器发送 keepalive 消息，以秒为单位。值为零时使用系统默认值。对于通过 Unix 域套接字建立的连接，或禁用 keepalive 时，此参数会被忽略。此参数仅在支持 TCP_KEEPIDLE 或等效套接字选项的系统以及 Windows 上受支持；在其他系统上无效。

keepalives_interval

控制未被服务器确认收到的 TCP keepalive 消息在多少秒后应被重传。值为零时使用系统默认值。对于通过 Unix 域套接字建立的连接，或禁用 keepalive 时，此参数会被忽略。此参数仅在支持 TCP_KEEPINTVL 或等效套接字选项的系统以及 Windows 上受支持；在其他系统上无效。

keepalives_count

控制在客户端与服务器之间的连接被视为中断之前，可以丢失多少个 TCP keepalive 消息。值为零时使用系统默认值。对于通过 Unix 域套接字建立的连接，或禁用 keepalive 时，此参数会被忽略。此参数仅在支持 TCP_KEEPCNT 或等效套接字选项的系统上受支持；在其他系统上无效。

tty

忽略此参数（以前用于指定服务器调试输出的发送位置）。

replication

这个选项确定连接是否应该使用复制协议而不是正常协议。这就是 PostgreSQL 复制连接以及诸如 pg_basebackup 这样的工具在内部使用的方式，但也可以被第三方应用程序使用。要了解复制协议的描述，请参考第 53.4 节。

支持以下值（不区分大小写）：

true, on, yes, 1

连接进入物理复制模式。

database

连接进入逻辑复制模式，连接到dbname 参数中指定的数据库。

false, off, no, 0

连接是常规连接，这是默认行为。

在物理或逻辑复制模式下，只能使用简单查询协议。

sslmode

这个选项确定是否以及以何种优先级与服务器协商安全的 SSL TCP/IP 连接。有六种模式：

disable

仅尝试非 SSL 连接

`allow`

首先尝试非 SSL 连接；如果失败，则尝试SSL 连接

`prefer`（默认）

首先尝试 SSL 连接；如果失败，则尝试非SSL 连接

`require`

仅尝试 SSL 连接。如果存在根 CA 文件，则验证证书的方式与指定了`verify-ca` 时相同

`verify-ca`

仅尝试 SSL 连接，并验证服务器证书是否由受信任的证书颁发机构（CA）颁发

`verify-full`

仅尝试 SSL 连接，验证服务器证书是否由受信任的CA 颁发，并且请求的服务器主机名与证书中的匹配

详细了解这些选项如何工作，请参阅第 34.18 节。

在 Unix 域套接字通信中，`sslmode` 会被忽略。如果 PostgreSQL 编译时未启用 SSL 支持，使用选项`require`、`verify-ca` 或 `verify-full` 会导致错误，而选项`allow` 和 `prefer` 将被接受，但libpq 实际上不会尝试建立 SSL 连接。

`requiressl`

此选项已弃用，请改用 `sslmode` 设置。

如果设置为 1，则需要与服务器建立SSL 连接（这相当于 `sslmode require`）。libpq 将拒绝连接，如果服务器不接受 SSL 连接。如果设置为 0（默认值），libpq 将与服务器协商连接类型（相当于 `sslmode prefer`）。此选项仅在 PostgreSQL 编译时启用了 SSL 支持的情况下可用。

`sslcompression`

如果设置为 1，通过 SSL 连接发送的数据将被压缩。如果设置为 0，将禁用压缩。默认值为 0。如果进行非 SSL 连接，则忽略此参数。

SSL 压缩现在被认为不安全，已不再建议使用。OpenSSL 1.1.0 默认禁用压缩，许多操作系统发行版也在更早的版本中禁用了压缩，因此，如果服务器不接受压缩，将此参数设为 on 不会产生效果。另一方面，OpenSSL 1.0.0 之前的版本不支持禁用压缩，因此这些版本会忽略此参数，是否使用压缩取决于服务器。

如果安全性不是主要考虑因素，压缩可以提高吞吐量，如果网络是瓶颈的话。如果 CPU 性能是限制因素，禁用压缩可以缩短响应时间并提高吞吐量。

`sslcert`

这个参数指定客户端 SSL 证书的文件名，替换默认的 `~/.postgresql/postgresql.crt`。如果没有建立 SSL 连接，则此参数将被忽略。

sslkey

这个参数指定了用于客户端证书的密钥的位置。它可以指定一个文件名，该文件名将被用来替代默认的 ~/.postgresql/postgresql.key，或者它可以指定一个从外部“引擎”（引擎是 OpenSSL 可加载模块）获取的密钥。外部引擎的指定形式应包含一个由冒号分隔的引擎名称和一个引擎特定的密钥标识符。如果没有进行 SSL 连接，则此参数将被忽略。

sslrootcert

这个参数指定一个包含 SSL 证书颁发机构（CA）证书的文件名。如果文件存在，服务器的证书将被验证是否由这些机构之一签名。默认值是~/.postgresql/root.crt。

sslcrl

此参数指定 SSL 服务器证书吊销列表（CRL）的文件名。如果该文件存在，在验证服务器证书时，会拒绝其中列出的证书。默认值为 ~/.postgresql/root.crl。

requirepeer

这个参数指定了服务器的操作系统用户名，例如requirepeer=postgres。在建立 Unix 域套接字连接时，如果设置了这个参数，客户端会在连接开始时检查服务器进程是否在指定的用户下运行；如果不是，则连接会因错误而中止。这个参数可用于提供类似于在 TCP/IP 连接上使用 SSL 证书的服务器认证。（请注意，如果 Unix 域套接字位于/tmp 或其他公共可写位置，任何用户都可以在那里启动一个服务器监听。使用这个参数来确保您连接到由受信任用户运行的服务器。）此选项仅在实现了peer 认证方法的平台上受支持；请参见第 20.9 节。

krbsrvname

使用 GSSAPI 认证时所用的 Kerberos 服务名。这必须与服务器配置中指定的 Kerberos 认证服务名称匹配，才能成功进行认证。（另请参见第 20.6 节。）

gsslib

用于 GSSAPI 认证的 GSS 库。目前，除了包含 GSSAPI 和 SSPI 支持的 Windows 构建之外，这将被忽略。在这种情况下，将其设置为gssapi，以使 libpq 使用 GSSAPI 库进行认证，而不是默认的 SSPI。

service

用于额外参数的服务名称。它指定了 pg_service.conf 中保存额外连接参数的服务名称。这允许应用程序只指定一个服务名称，以便可以集中维护连接参数。参见第 34.16 节。

target_session_attrs

如果此参数设为 read-write，只有默认接受读写事务的连接才被视为可接受。每次成功建立连接后都会发送查询 SHOW transaction_read_only；如果返回 on，就关闭该连接。如果连接字符串指定了多个主机，会继续尝试其余服务器，如同连接尝试失败时一样。此参数的默认值 any 将所有连接都视为可接受。

34.2. 连接状态函数

这些函数可用于查询现有数据库连接对象的状态。

> **提示**
>
> 编写 libpq 应用程序时，应注意维护 PGconn 的抽象。请使用下述访问函数获取 PGconn 的内容。不建议通过 libpq-int.h 引用 PGconn 的内部字段，因为这些字段将来可能改变。

以下函数返回建立连接时确定的参数值。这些值在连接存续期间保持不变。如果使用多主机连接字符串，并使用同一个 PGconn 对象建立新连接，则 PQhost、PQport 和 PQpass 的返回值可能改变。其他值在 PGconn 对象的整个生命周期内保持不变。

PQdb

返回该连接的数据库名。

```
char *PQdb(const PGconn *conn);
```

PQuser

返回该连接的用户名。

```
char *PQuser(const PGconn *conn);
```

PQpass

返回该连接的密码。

```
char *PQpass(const PGconn *conn);
```

PQpass 将返回连接参数中指定的密码；如果连接参数中没有密码，并且从密码文件中取得了密码，则返回该密码。在后一种情况下，如果连接参数中指定了多个主机，在连接建立之前不能依赖 PQpass 的结果。连接状态可以用 PQstatus 函数检查。

PQhost

返回活动连接的服务器主机名。可能是主机名、IP 地址或者一个目录路径（如果通过 Unix 套接字连接，路径的情况很容易区分，因为路径总是一个绝对路径，以/开始）。

```
char *PQhost(const PGconn *conn);
```

如果连接参数同时指定了 host 和 hostaddr，则PQhost 将返回 host 信息。如果仅指定了 hostaddr，则返回它。如果在连接参数中指定了多个主机，PQhost 返回实际连接到的主机。

如果 *conn* 参数为 NULL，则 PQhost 返回 NULL。否则，如果在生成主机信息时发生错误（例如连接尚未完全建立，或出现了错误），则返回空字符串。

如果在连接参数中指定了多个主机，则在连接建立之前都不能依赖于PQhost 的结果。连接的状态可以用函数 PQstatus 检查。

PQport

返回活动连接的端口。

```
char *PQport(const PGconn *conn);
```

如果在连接参数中指定了多个端口，PQport 返回实际连接到的端口。

如果 *conn* 参数为 NULL，则 PQport 返回 NULL。否则，如果在生成端口信息时发生错误（例如连接尚未完全建立，或出现了错误），则返回空字符串。

如果在连接参数中指定了多个端口，则在连接建立之前都不能依赖于PQport 的结果。连接的状态可以用函数 PQstatus 检查。

PQtty

返回该连接用于调试的 TTY。（此功能已过时，因为服务器不再使用TTY 设置，但为保持向后兼容仍保留此函数。）

```
char *PQtty(const PGconn *conn);
```

PQoptions

返回连接请求中传递的命令行选项。

```
char *PQoptions(const PGconn *conn);
```

以下函数返回的状态数据，可能随着对 PGconn 对象执行操作而改变。

PQstatus

返回该连接的状态。

```
ConnStatusType PQstatus(const PGconn *conn);
```

该状态可以是一系列值之一。不过，其中只有两个在一个异步连接过程之外可见：CONNECTION_OK 和 CONNECTION_BAD。一个到数据库的完好连接的状态为 CONNECTION_OK。一个失败的连接尝试则由状态 CONNECTION_BAD 表示。通常，一个 OK 状态将一直保持到PQfinish，但是一次通信失败可能导致该状态过早地改变为 CONNECTION_BAD。在那种情况下，该应用可以通过调用PQreset 尝试恢复。

关于其他可能会被返回的状态代码，请见PQconnectStartParams、PQconnectStart 和 PQconnectPoll 的条目。

PQtransactionStatus

返回服务器的当前事务内状态。

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

该状态可能是 PQTRANS_IDLE（当前空闲）、PQTRANS_ACTIVE（一个命令运行中）、PQTRANS_INTRANS（空闲，处于一个有效的事务块中）或者PQTRANS_INERROR（空闲，处于一个失败的事务块中）。如果该连接异常，将会报告PQTRANS_UNKNOWN。只有当一个查询已经被发送给服务器并且还没有完成时，才会报告PQTRANS_ACTIVE。

PQparameterStatus

查找服务器某个参数的当前设置。

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

服务器会在连接启动时，以及某些参数值发生变化时，自动报告这些参数值。PQparameterStatus 可用于查询这些设置。如果已知该参数，则返回其当前值；如果未知，则返回NULL。

当前版本报告的参数包括：server_version、server_encoding、client_encoding、application_name、is_superuser、session_authorization、DateStyle、IntervalStyle、TimeZone、integer_datetimes、standard_conforming_strings。（8.0 之前的版本不报告 server_encoding、TimeZone 和 integer_datetimes；8.1 之前的版本不报告 standard_conforming_strings；8.4 之前的版本不报告 IntervalStyle；9.0 之前的版本不报告 application_name。）注意，server_version、server_encoding 和 integer_datetimes 在启动后不能改变。

使用 3.0 之前协议的服务器不报告参数设置，但 libpq 仍包含获取 server_version 和 client_encoding 值的逻辑。建议应用程序使用 PQparameterStatus，而不是专门编写代码来确定这些值。（但要注意，在使用 3.0 之前协议的连接上，连接启动后通过 SET 改变 client_encoding，不会反映在 PQparameterStatus 的结果中。）对于 server_version，另请参见 PQserverVersion，它以数值形式返回此信息，更易于比较。

如果服务器未报告 standard_conforming_strings 的值，应用程序可以假定其为off，即反斜杠在字符串字面量中被视为转义字符。此外，服务器报告此参数也表明它接受转义字符串语法（E'...'）。

返回的指针虽然被声明为 const，但实际上指向与PGconn 结构体关联的可变存储。不能假定该指针在执行其他查询后仍然有效。

PQprotocolVersion

查询正在使用的前端/后端协议。

```
int PQprotocolVersion(const PGconn *conn);
```

应用程序可以使用此函数判断是否支持某些特性。目前，可能的值为 2（协议 2.0）、3（协议 3.0）或零（连接异常）。连接启动完成后，协议版本不会改变，但理论上可能在连接重置期间改变。通常，与 PostgreSQL 7.4 或更高版本的服务器通信时使用协议 3.0；7.4 之前的服务器仅支持协议 2.0。（协议 1.0 已过时，libpq 不支持它。）

PQserverVersion

返回一个表示服务器版本的整数。

```
int PQserverVersion(const PGconn *conn);
```

应用可能会使用这个函数来判断它们连接到的数据库服务器的版本。结果通过将服务器的主版本号乘以 10000 再加上次版本号形成。例如，版本 10.1 将被返回为 100001，而版本 11.0 将被返回为 110000。如果连接无效则返回零。

在主版本 10 之前，PostgreSQL 采用一种由三个部分组成的版本号，其中前两部分共同表示主版本。对于那些版本，PQserverVersion 为每个部分使用两位数字，例如版本 9.1.5 将被返回为 90105，而版本 9.2.0 将被返回为 90200。

因此，出于判断特性兼容性的目的，应用应该将PQserverVersion 的结果除以 100 而不是 10000 来判断逻辑的主版本号。在所有主版本系列中，各次版本（缺陷修复版本）之间只有最后两位数字不同。

PQerrorMessage

返回连接上的一个操作最近产生的错误消息。

```
char *PQerrorMessage(const PGconn *conn);
```

几乎所有 libpq 函数在失败时都会设置一条供 PQerrorMessage 返回的消息。注意，按照 libpq 的约定，非空的 PQerrorMessage 结果可能包含多行，并以换行符结尾。调用者不应直接释放该结果；当关联的 PGconn 句柄被传给 PQfinish 时，结果会被释放。不能假定在对 PGconn 结构体执行其他操作后，结果字符串仍保持不变。

PQsocket

获取与服务器相连的套接字的文件描述符编号。有效描述符大于或等于 0。结果为 -1 表示当前没有打开服务器连接（在普通操作期间这将不会改变，但是在连接设置或重置期间可能改变）。

```
int PQsocket(const PGconn *conn);
```

PQbackendPID

返回处理这个连接的后端进程的进程 ID（PID）。

```
int PQbackendPID(const PGconn *conn);
```

后端 PID 可用于调试，也可与 NOTIFY 消息进行比较（消息包含发出通知的后端进程的 PID）。注意，该 PID 属于在数据库服务器主机上运行的进程，而非本地主机上的进程！

PQconnectionNeedsPassword

如果连接认证方法要求一个密码但没有可用的密码，返回真（1）。否则返回假（0）。

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

这个函数可以在连接尝试失败后被应用于决定是否向用户提示要求一个密码。

PQconnectionUsedPassword

如果连接认证方法使用一个密码，返回真（1）。否则返回假（0）。

```
int PQconnectionUsedPassword(const PGconn *conn);
```

这个函数能在一次连接尝试失败或成功后用于检测该服务器是否要求一个密码。

以下函数返回与 SSL 相关的信息。这些信息通常在连接建立后不会改变。

PQsslInUse

如果连接使用 SSL，则返回真（1）；否则返回假（0）。

```
int PQsslInUse(const PGconn *conn);
```

PQsslAttribute

返回连接的 SSL 相关信息。

```
const char *PQsslAttribute(const PGconn *conn, const char *attribute_name);
```

可用属性列表因所使用的 SSL 库和连接类型而异。如果某个属性不可用，则返回 NULL。

通常可以获取以下属性：

library

使用的 SSL 实现的名称。（目前只实现了"OpenSSL"）

protocol

使用的 SSL/TLS 版本。常见值为"TLSv1"、"TLSv1.1" 和"TLSv1.2"，但如果使用其他协议，则实现可能返回其他字符串。

key_bits

加密算法使用的密钥位数。

cipher

使用的密码套件的简称，例如"DHE-RSA-DES-CBC3-SHA"。这些名称特定于每个 SSL 实现。

compression

如果使用 SSL 压缩，则返回"on"，否则返回"off"。

PQsslAttributeNames

返回可用的 SSL 属性名称数组。数组以 NULL 指针结尾。

```
const char * const * PQsslAttributeNames(const PGconn *conn);
```

PQsslStruct

返回指向描述此连接的对象的指针，该对象的类型由 SSL 实现决定。

```
void *PQsslStruct(const PGconn *conn, const char *struct_name);
```

可用的结构体取决于所使用的 SSL 实现。对于 OpenSSL，可以通过名称 "OpenSSL" 获取一个结构体，函数返回指向 OpenSSL 的 SSL 结构体的指针。可以使用如下代码调用此函数：

```
#include <libpq-fe.h>
#include <openssl/ssl.h>

...

    SSL *ssl;

    dbconn = PQconnectdb(...);
    ...

    ssl = PQsslStruct(dbconn, "OpenSSL");
    if (ssl)
    {
        /* 使用OpenSSL函数访问ssl */
    }
```

这个结构体可用于验证加密级别，检查服务器证书等。请参考OpenSSL 文档以获取有关此结构体的信息。

PQgetssl

返回在连接中使用的 SSL 结构体，如果未使用 SSL，则返回 NULL。

```
void *PQgetssl(const PGconn *conn);
```

这个函数等同于 PQsslStruct(conn, "OpenSSL")。不应该在新应用程序中使用，因为返回的结构体特定于 OpenSSL，如果使用另一个 SSL 实现，则不可用。要检查连接是否使用 SSL，请调用PQsslInUse，要获取有关连接的更多详细信息，请使用PQsslAttribute。

34.3. 命令执行函数

与数据库服务器的连接成功建立后，此处描述的函数用于执行 SQL 查询和命令。

34.3.1. 主要函数

PQexec

提交一个命令给服务器并且等待结果。

```
PGresult *PQexec(PGconn *conn, const char *command);
```

返回一个 PGresult 指针，也可能返回空指针。除内存不足或无法向服务器发送命令等严重错误外，通常都会返回非空指针。应调用 PQresultStatus 函数检查返回值是否表示错误（也应检查空指针，此时该函数返回 PGRES_FATAL_ERROR）。使用 PQerrorMessage 获取此类错误的更多信息。

命令字符串可以包含多个 SQL 命令（以分号分隔）。在一次 PQexec 调用中发送的多个查询会在同一个事务中处理，除非查询字符串中显式包含BEGIN/COMMIT 命令，将其划分为多个事务。（有关服务器如何处理多查询字符串的详细信息，请参见第 53.2.2.1 节。）但要注意，返回的 PGresult 结构体只描述该字符串中最后执行的命令的结果。如果其中一条命令失败，就会在此处停止处理该字符串，返回的PGresult 则描述该错误。

PQexecParams

提交一个命令给服务器并且等待结果，它可以在 SQL 命令文本之外独立地传递参数。

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

PQexecParams 与 PQexec 相似，但是提供了额外的功能：参数值可以与命令字符串分开指定，并且可以以文本或二进制格式请求查询结果。PQexecParams 仅支持使用协议 3.0 及更高版本的连接；使用协议 2.0 时会失败。

该函数的参数是：

conn

要在其中发送命令的连接对象。

command

要执行的 SQL 命令字符串。如果使用了参数，它们在该命令字符串中被引用为$1、$2 等。

nParams

提供的参数数量。它是数组 *paramTypes[]*、*paramValues[]*、*paramLengths[]* 和 *paramFormats[]* 的长度（当 *nParams* 为零时，数组指针可以是NULL）。

paramTypes[]

通过 OID 指定要赋予给参数符号的数据类型。如果 *paramTypes* 为 NULL 或者该数组中任何特定元素为零，服务器会用对待未指定类型的字符串字面量的方式为参数符号推测一种数据类型。

paramValues[]

指定参数的实际值。数组中的空指针表示对应参数为 null；否则，指针指向以零结尾的文本字符串（文本格式），或采用服务器所要求格式的二进制数据（二进制格式）。

paramLengths[]

指定二进制格式参数的实际数据长度。对于 null 参数和文本格式参数，该值会被忽略。如果没有二进制参数，数组指针可以为空指针。

paramFormats[]

指定参数采用文本格式（在对应数组元素中填入零）还是二进制格式（填入一）。如果数组指针为空指针，则将所有参数视为文本字符串。

以二进制格式传递值时，需要了解后端所要求的内部表示形式。例如，整数必须以网络字节序传递。传递 numeric 值时，需要了解服务器的存储格式，其实现见 src/backend/utils/adt/numeric.c::numeric_send() 和 src/backend/utils/adt/numeric.c::numeric_recv()。

resultFormat

指定零以获取文本格式的结果，指定一以获取二进制格式的结果。（目前无法让不同结果列使用不同格式，尽管底层协议支持这样做。）

PQexecParams 相对于 PQexec 的主要优点是可以将参数值与命令字符串分开，从而避免繁琐且容易出错的加引号和转义工作。

和 PQexec 不同，PQexecParams 至多允许在给定串中出现一个 SQL 命令（其中可以有分号，但是不能有超过一个非空命令）。这是底层协议的一个限制，但是有助于抵抗 SQL 注入攻击。

> **提示**
>
> 通过 OID 指定参数类型较为繁琐，尤其是在不希望将具体 OID 值写死在程序中时。不过，即使服务器无法自行确定参数类型，或者推断出的类型与你所需的不同，也可以避免直接指定 OID。在 SQL 命令文本中，为参数符号添加显式类型转换，即可指定要发送的数据类型。例如：
>
> ```
> SELECT * FROM mytable WHERE x = $1::bigint;
> ```
>
> 这会强制将参数 `$1` 当作 `bigint`，而默认情况下会为它分配与 `x` 相同的类型。以二进制格式发送参数值时，强烈建议采用这种方式，或直接指定类型的数值 OID，来明确决定参数类型。因为二进制格式的冗余比文本格式更少，服务器发现类型不匹配错误的机会也更少。

`PQprepare`

提交一个请求用给定参数创建一个预备语句并且等待完成。

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

`PQprepare` 创建一个预备语句，供随后使用 `PQexecPrepared` 执行。这个特性允许命令被反复执行而无需每次都进行解析和规划，详见 PREPARE。`PQprepare` 仅支持使用协议 3.0 及更高版本的连接；使用协议 2.0 时会失败。

该函数从 *`query`* 串创建一个名为 *`stmtName`* 的预备语句，该串必须包含一个单一 SQL 命令。*`stmtName`* 可以是"" 来创建一个未命名语句，在这种情况下任何已存在的未命名语句将被自动替换。否则，如果语句名称已经在当前会话中被定义，则是一种错误。如果使用了任何参数，它们在查询中以`$1`、`$2` 等引用。*`nParams`* 是在数组 *`paramTypes[]`* 中预先指定了类型的参数数量（当 *`nParams`* 为零时，该数组指针可以是`NULL`）。*`paramTypes[]`* 通过 OID 指定要赋予给参数符号的数据类型。如果 *`paramTypes`* 是 `NULL` 或者该数组中任何特定元素为零，服务器会用对待未指定类型的字符串字面量的方式为参数符号推测一种数据类型。还有，查询能够使用编号高于*`nParams`* 的参数符号，它们的数据类型也会被自动推测（找出推测出的数据类型的方法见`PQdescribePrepared`）。

正如 `PQexec` 一样，结果通常是一个`PGresult` 对象，其内容代表服务器端成功或失败。返回空指针表示内存不足，或者根本无法发送命令。关于错误的更多信息请见 `PQerrorMessage`。

用于 `PQexecPrepared` 的预备语句也可以通过执行 SQLPREPARE 语句来创建。此外，虽然没有 libpq 函数可删除预备语句，但可以使用 SQLDEALLOCATE 语句来完成。

PQexecPrepared

发送一个请求来执行一个带有给定参数的预备语句，并等待结果。

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

PQexecPrepared 类似于 PQexecParams，但它通过已创建的预备语句的名称指定要执行的命令，而非提供查询字符串。此功能使重复使用的命令只需解析和规划一次，而不必在每次执行时都进行这些工作。该语句必须事先在当前会话中创建为预备语句。PQexecPrepared 仅支持使用协议 3.0 及更高版本的连接；使用协议 2.0 时会失败。

参数与 PQexecParams 相同，只是给出了预备语句的名称而不是查询字符串，并且 *paramTypes[]* 参数不存在（因为在创建预备语句时已确定了参数类型）。

PQdescribePrepared

提交请求以获取有关指定预备语句的信息，并等待完成。

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

PQdescribePrepared 允许应用程序获取关于先前创建的预备语句的信息。PQdescribePrepared 仅支持使用协议 3.0 及更高版本的连接；使用协议 2.0 时会失败。

stmtName 可以是"" 或 NULL 来引用未命名的语句，否则必须是现有预备语句的名称。成功时，返回一个状态为PGRES_COMMAND_OK 的 PGresult。函数 PQnparams 和 PQparamtype 可以应用于此 PGresult 以获取有关预备语句参数的信息，函数 PQnfields、PQfname、PQftype 等提供有关语句的结果列（如果有）的信息。

PQdescribePortal

提交请求以获取有关指定 portal 的信息，并等待完成。

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

PQdescribePortal 允许应用程序获取先前创建的 portal 的信息。（libpq 不提供对 portal 的直接访问，但可以用此函数检查通过 DECLARE CURSOR SQL 命令创建的游标的属性。）PQdescribePortal 仅支持使用协议 3.0 及更高版本的连接；使用协议 2.0 时会失败。

portalName 可以是"" 或 NULL 来引用未命名的 portal，否则必须是现有 portal 的名称。成功时，将返回一个带有状态PGRES_COMMAND_OK 的 PGresult。函数 PQnfields、PQfname、PQftype 等可应用于 PGresult，以获取有关 portal 的结果列（如果有）的信息。

PGresult 结构体封装服务器返回的结果。编写 libpq 应用程序时，应注意维护 PGresult 的抽象。请使用下述访问函数获取 PGresult 的内容。避免直接引用 PGresult 结构体的字段，因为这些字段将来可能改变。

PQresultStatus

返回该命令的结果状态。

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus 可以返回以下值之一：

PGRES_EMPTY_QUERY

发送给服务器的字符串为空。

PGRES_COMMAND_OK

一个不返回数据的命令成功完成。

PGRES_TUPLES_OK

一个返回数据的命令（例如 SELECT 或者 SHOW）成功完成。

PGRES_COPY_OUT

从服务器复制出数据的传输开始。

PGRES_COPY_IN

复制数据到服务器的传输开始。

PGRES_BAD_RESPONSE

无法理解服务器的响应。

PGRES_NONFATAL_ERROR

发生了一次非致命错误（一个提示或警告）。

PGRES_FATAL_ERROR

发生了一次致命错误。

PGRES_COPY_BOTH

向服务器复制数据/从服务器复制数据的传输开始。这个特性当前只被用于流复制，因此这个状态应该不会在普通应用中出现。

PGRES_SINGLE_TUPLE

PGresult 包含来自于当前命令的一个单一结果元组。这个状态只在查询选择了单行模式时发生（见第 34.5 节）。

如果结果状态为 PGRES_TUPLES_OK 或 PGRES_SINGLE_TUPLE，可以使用下述函数取得查询返回的行。注意，即使SELECT 命令恰好返回零行，状态仍为PGRES_TUPLES_OK。

PGRES_COMMAND_OK 用于不可能返回行的命令（例如 INSERT 或 UPDATE 不带 RETURNING 子句的命令等）。如果响应为 PGRES_EMPTY_QUERY，可能表示客户端软件中存在缺陷。

一个状态为 PGRES_NONFATAL_ERROR 的结果将不会被 PQexec 或者其他查询执行函数直接返回，这类结果将被传递给通知处理器（见第 34.12 节）。

PQresStatus

将 PQresultStatus 返回的枚举值转换为描述该状态码的字符串常量。调用者不应释放此结果。

```
char *PQresStatus(ExecStatusType status);
```

PQresultErrorMessage

返回与命令关联的错误消息；如果没有错误，则返回空字符串。

```
char *PQresultErrorMessage(const PGresult *res);
```

如果发生了错误，返回的字符串会以换行符结尾。调用者不应直接释放结果；当关联的 PGresult 句柄被传给 PQclear 时，结果会被释放。

紧跟着一个 PQexec 或 PQgetResult 调用，PQerrorMessage（在连接上）将返回与 PQresultErrorMessage 相同的字符串（在结果上）。不过，一个PGresult 将保持它的错误消息直到被销毁，而连接的错误消息将在后续操作被执行时被更改。当你想要知道与一个特定PGresult 相关的状态，使用PQresultErrorMessage。而当你想要知道连接上最后一个操作的状态，使用PQerrorMessage。

PQresultVerboseErrorMessage

返回与 PGresult 对象关联的错误消息的重新格式化版本。

```
char *PQresultVerboseErrorMessage(const PGresult *res,
                                  PGVerbosity verbosity,
                                  PGContextVisibility show_context);
```

有时客户端希望获取先前报告过的错误的更详细版本。PQresultVerboseErrorMessage 可满足这一需求：它按指定的详细程度设置重新生成消息。假如在生成给定 PGresult 时，连接已经采用这些设置，那么 PQresultErrorMessage 返回的就会是这条消息。如果 PGresult 不是错误结果，则改为报告 “PGresult is not an error result” 。返回的字符串以换行符结尾。

和大部分从 PGresult 中提取数据的其他函数不同，这个函数的结果是一个全新分配的字符串。调用者在不需要这个字符串以后，必须使用PQfreemem() 释放它。

如果内存不足，可能会返回 NULL。

PQresultErrorField

返回错误报告中的单个字段。

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

fieldcode 是错误字段标识符，参见下文列出的符号。如果 PGresult 不是错误或警告结果，或者不包含指定字段，则返回 NULL。字段值通常不含末尾换行符。调用者不应直接释放结果；当关联的 PGresult 句柄被传给 PQclear 时，结果会被释放。

可以使用以下字段代码：

PG_DIAG_SEVERITY

严重性。字段的内容是 ERROR、FATAL 或 PANIC（在一个错误消息中）。或者是 WARNING、NOTICE、DEBUG、INFO 或 LOG（在一个提示消息中）。或者是其中之一的一个本地化翻译。总是存在。

PG_DIAG_SEVERITY_NONLOCALIZED

严重性。字段的内容是 ERROR、FATAL 或 PANIC（在一个错误消息中）。或者是 WARNING、NOTICE、DEBUG、INFO 或 LOG（在一个提示消息中）。这和 PG_DIAG_SEVERITY 字段相同，不过内容不会被本地化。只存在于PostgreSQL 9.6 及更高版本产生的报告中。

PG_DIAG_SQLSTATE

用于错误的 SQLSTATE 代码。SQLSTATE 代码标识了已经发生的错误的类型，它可以被前端应用用来执行特定操作（例如错误处理）来响应一个特定数据库错误。一个可能的 SQLSTATE 代码列表可见附录 A。这个字段无法被本地化，并且总是存在。

PG_DIAG_MESSAGE_PRIMARY

主要的人类可读的错误消息（通常是一行）。总是存在。

PG_DIAG_MESSAGE_DETAIL

细节：一个可选的次级错误消息，它携带了关于问题的更多细节。可能有多行。

PG_DIAG_MESSAGE_HINT

提示：一个关于如何处理该问题的可选建议。它与细节的区别在于它提供了建议（可能不合适）而不是确切事实。可能有多行。

PG_DIAG_STATEMENT_POSITION

包含一个十进制整数的字符串，它表示一个错误游标位置，该位置是原始语句字符串的索引。第一个字符的索引是 1，位置以字符计算而不是以字节计算。

PG_DIAG_INTERNAL_POSITION

这被定义为与 PG_DIAG_STATEMENT_POSITION 字段相同，但是它被用在游标位置引用一个内部产生的命令而不是客户端提交的命令时。当这个字段出现时，PG_DIAG_INTERNAL_QUERY 字段将总是出现。

PG_DIAG_INTERNAL_QUERY

一个失败的内部产生的命令的文本。例如，这可能是由一个 PL/pgSQL 函数发出的 SQL 查询。

PG_DIAG_CONTEXT

指示错误发生的上下文。当前这包括活动过程语言函数的调用栈追踪以及内部生成的查询。追踪是每行一项，最近的排在最前面。

PG_DIAG_SCHEMA_NAME

如果错误与某个特定的数据库对象相关，这里是包含该对象的模式名（如果有）。

PG_DIAG_TABLE_NAME

如果错误与某个特定表相关，这里是该表的名字（该表的模式参考模式名字段）。

PG_DIAG_COLUMN_NAME

如果错误与一个特定表列相关，这里是该表列的名字（参考模式和表名字段来标识该表）。

PG_DIAG_DATATYPE_NAME

如果错误与一个特定数据类型相关，这里是该数据类型的名字（该数据类型的模式名参考模式名字段）。

PG_DIAG_CONSTRAINT_NAME

如果错误与一个特定约束相关，这里是该约束的名字。相关的表或域参考上面列出的字段（为了这个目的，索引也被视作约束，即使它们不是用约束语法创建的）。

PG_DIAG_SOURCE_FILE

报告错误的源代码所在的文件名。

PG_DIAG_SOURCE_LINE

报告错误的源代码行号。

PG_DIAG_SOURCE_FUNCTION

报告错误的源代码函数的名字。

注意

用于模式名、表名、列名、数据类型名和约束名的字段只提供给有限的错误类型，见附录 A。不要假定任何这些字段的存在保证另一个字段的存在。核心错误源会遵守上面提到的内在联系，但是用户定义的函数可能以其他方式使用这些字段。同样地，不要假定这些字段代表当前数据库中现存的对象。

客户端负责按自身需要格式化所显示的信息，尤其应在需要时将长行折行。错误消息字段中的换行符应当视为段落分隔，而非行分隔。

libpq 内部产生的错误包含严重性和主要消息，但通常没有其他字段。使用 3.0 之前协议的服务器返回的错误包含严重性和主要消息，有时还包含详细消息，但没有其他字段。

注意，错误字段只对PGresult 对象有效，对PGconn 对象无效。没有 PQerrorField 函数。

PQclear

释放与 PGresult 关联的存储空间。每个命令结果在不再需要时都应通过 PQclear 释放。

```
void PQclear(PGresult *res);
```

你可以在需要时一直保留 PGresult 对象；它不会在你发出新命令时消失，甚至在关闭连接后也不会消失。要销毁它，你必须调用PQclear。否则应用程序会发生内存泄漏。

34.3.2. 检索查询结果信息

这些函数用于从表示成功查询结果的 PGresult 对象中提取信息，即状态为 PGRES_TUPLES_OK 或 PGRES_SINGLE_TUPLE 的对象。它们也可用于提取成功 Describe 操作的结果信息：Describe 结果包含的列信息与实际执行查询时相同，但行数为零。对于其他状态值的对象，这些函数会将结果视为零行、零列。

PQntuples

返回查询结果中的行（元组）数（注意，PGresult 对象被限制为不超过 INT_MAX 行，因此一个int 结果就足够了）。

```
int PQntuples(const PGresult *res);
```

PQnfields

返回查询结果中每一行的列（字段）数。

```
int PQnfields(const PGresult *res);
```

PQfname

返回给定列号对应的列名。列号从 0 开始。调用者不应直接释放结果；当关联的 PGresult 句柄被传给 PQclear 时，结果会被释放。

```
char *PQfname(const PGresult *res,
              int column_number);
```

如果列号超出范围，将返回NULL。

PQfnumber

返回与给定列名相关联的列号。

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

如果给定的名字不匹配任何列，将返回 -1。

给定名称按 SQL 命令中的标识符处理，即除非用双引号引用，否则会转换为小写。例如，对于以下 SQL 命令生成的查询结果：

```
SELECT 1 AS FOO, 2 AS "BAR";
```

会得到以下结果：

```
PQfname(res, 0)              foo
PQfname(res, 1)              BAR
PQfnumber(res, "FOO")        0
PQfnumber(res, "foo")        0
PQfnumber(res, "BAR")        -1
PQfnumber(res, "\"BAR\"")    1
```

PQftable

返回给定列所取自的表的 OID。列号从 0 开始。

```
Oid PQftable(const PGresult *res,
             int column_number);
```

如果列号超出范围、指定的列不是对表列的简单引用，或者使用 3.0 之前的协议，则返回 InvalidOid。可以查询系统表 pg_class，确定所引用的具体表。

包含 libpq 头文件后，将定义类型 Oid 和常量 InvalidOid。它们都属于某种整数类型。

PQftablecol

返回指定查询结果列所对应的表列在表中的列号。查询结果的列号从 0 开始，而表列的编号非零。

```
int PQftablecol(const PGresult *res,
                int column_number);
```

如果列号超出范围、指定的列不是对表列的简单引用，或者使用 3.0 之前的协议，则返回零。

PQfformat

返回表示给定列格式的格式代码。列号从 0 开始。

```
int PQfformat(const PGresult *res,
              int column_number);
```

格式代码零表示文本数据，格式代码一表示二进制数据。（其他代码保留供将来定义。）

PQftype

返回与给定列号相关联的数据类型。被返回的整数是该类型的内部 OID 号。列号从 0 开始。

```
Oid PQftype(const PGresult *res,
            int column_number);
```

可以查询系统表 pg_type，获取各种数据类型的名称和属性。内置数据类型的 OID 定义在安装目录中的 include/server/catalog/pg_type_d.h 文件中。

PQfmod

返回与给定列号相关联的列的类型修饰符。列号从 0 开始。

```
int PQfmod(const PGresult *res,
           int column_number);
```

修饰符值的含义由数据类型决定，通常表示精度或大小限制。值 -1 表示“没有可用信息”。大多数数据类型不使用修饰符，此时该值始终为 -1。

PQfsize

返回给定列号对应列的大小，以字节计。列号从 0 开始。

```
int PQfsize(const PGresult *res,
            int column_number);
```

PQfsize 返回数据库行中为该列分配的空间，即服务器内部表示该数据类型所需的大小。（因此，它对客户端用处不大。）负值表示该数据类型是变长类型。

PQbinaryTuples

如果 PGresult 包含二进制数据，返回 1。如果包含的是文本数据，返回 0。

```
int PQbinaryTuples(const PGresult *res);
```

此函数已弃用（与 COPY 配合使用的情况除外），因为单个 PGresult 可能在部分列中包含文本数据，而在其他列中包含二进制数据。推荐使用 PQfformat。只有结果中的所有列都采用二进制格式（格式 1）时，PQbinaryTuples 才返回 1。

PQgetvalue

返回 PGresult 中某一行的单个字段值。行号和列号从 0 开始。调用者不应直接释放结果；当关联的 PGresult 句柄被传给 PQclear 时，结果会被释放。

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

对于文本格式的数据，PQgetvalue 返回字段值的字符串表示，以零字节结尾。对于二进制格式的数据，返回值采用该数据类型的 typsend 和 typreceive 函数所决定的二进制表

示。（这种情况下，值后面实际上也有一个零字节，但通常没有用处，因为值本身很可能包含零字节。）

如果字段值为 null，则返回空字符串。关于如何区分 null 值与空字符串值，参见 PQgetisnull。

PQgetvalue 返回的指针指向属于 PGresult 结构体的存储空间。不应修改它所指向的数据；如果需要在 PGresult 结构体的生命周期结束后继续使用这些数据，就必须显式地将数据复制到其他存储空间。

PQgetisnull

检查字段是否为 null 值。行号和列号从 0 开始。

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

如果字段为 null，此函数返回 1；如果包含非 null 值，则返回 0。（注意，对于 null 字段，PQgetvalue 返回空字符串，而非空指针。）

PQgetlength

返回字段值的实际长度，以字节计。行号和列号从 0 开始。

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

这是该数据值的实际数据长度，即 PQgetvalue 所指对象的大小。对于文本格式的数据，它与 strlen() 的结果相同。对于二进制格式，这是必不可少的信息。注意，不应依赖 PQfsize 来获取实际数据长度。

PQnparams

返回一个预备语句的参数数量。

```
int PQnparams(const PGresult *res);
```

只有在查看 PQdescribePrepared 的结果时，这个函数才有用。对于其他类型的查询，它将返回零。

PQparamtype

返回所指示的语句参数的数据类型。参数号从 0 开始。

```
Oid PQparamtype(const PGresult *res, int param_number);
```

只有在查看 PQdescribePrepared 的结果时，这个函数才有用。对于其他类型的查询，它将返回零。

PQprint

将所有行输出到指定的输出流，并可选择输出列名。

```
void PQprint(FILE *fout,      /* 输出流 */
             const PGresult *res,
             const PQprintOpt *po);
typedef struct
{
    pqbool  header;      /* 打印输出字段标题和行数 */
    pqbool  align;       /* 填充并对齐字段 */
    pqbool  standard;    /* 旧的简陋格式 */
    pqbool  html3;       /* 输出 HTML 表格 */
    pqbool  expanded;    /* 展开表格 */
    pqbool  pager;       /* 需要时使用分页器输出 */
    char    *fieldSep;   /* 字段分隔符 */
    char    *tableOpt;   /* 用于 HTML 表格元素的属性 */
    char    *caption;    /* HTML 表格标题 */
    char    **fieldName; /* 以空指针结尾的替换字段名数组 */
} PQprintOpt;
```

psql 以前用此函数打印查询结果，现在已不再使用。注意，此函数假定所有数据都是文本格式。

34.3.3. 检索其他结果信息

这些函数被用来从 PGresult 对象中抽取其他信息。

PQcmdStatus

返回来自于产生 PGresult 的 SQL 命令的命令状态标签。

```
char *PQcmdStatus(PGresult *res);
```

通常这就是该命令的名称，但是它可能包括额外数据，例如已被处理的行数。调用者不应该直接释放该结果。它将在相关的PGresult 句柄被传递给 PQclear 之后被释放。

PQcmdTuples

返回受该 SQL 命令影响的行数。

```
char *PQcmdTuples(PGresult *res);
```

此函数返回一个字符串，其中包含产生该 PGresult 的 SQL 语句所影响的行数。此函数只能在执行 SELECT、CREATE TABLE AS、INSERT、UPDATE、DELETE、MOVE、FETCH 或 COPY 语句之后使用，也可以在对包含 INSERT、UPDATE 或 DELETE 语句的预备查询执行 EXECUTE 之后使用。如果产生 PGresult 的是其他命令，PQcmdTuples 将返回空字符串。调用者不应直接释放返回值；当关联的 PGresult 句柄被传给 PQclear 时，返回值会被释放。

PQoidValue

如果该 SQL 命令是一个正好将一行插入到具有 OID 的表的 INSERT，或者是一个包含合适INSERT 语句的预备查询的 EXECUTE，这个函数返回被插入行的 OID。否则，这个函数返回InvalidOid。如果被 INSERT 语句影响的表不包含 OID，这个函数也将返回 InvalidOid。

```
Oid PQoidValue(const PGresult *res);
```

PQoidStatus

此函数不是线程安全的，现已弃用，推荐改用 PQoidValue。它返回包含插入行 OID 的字符串，而 PQoidValue 返回 OID 值。

```
char *PQoidStatus(const PGresult *res);
```

34.3.4. 用于在 SQL 命令中嵌入字符串的转义

PQescapeLiteral

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

为了让一个字符串可用于 SQL 命令，PQescapeLiteral 会对它进行转义。当在 SQL 命令中把数据值作为字符串字面量插入时，这个函数很有用。一些字符（例如引号和反斜杠）必须经过转义，才不会被 SQL 解析器解释成特殊含义。PQescapeLiteral 执行这种操作。

PQescapeLiteral 返回 *str* 参数的转义版本，存放在通过 malloc() 分配的内存中。结果不再需要时，应使用 PQfreemem() 释放该内存。输入不必以零字节结尾，末尾零字节也不应计入 *length*。（如果在处理完 *length* 个字节之前遇到末尾零字节，PQescapeLiteral 会在该字节处停止；这一行为类似于 strncpy。）返回字符串中的所有特殊字符均已替换，以便 PostgreSQL 字符串字面量解析器正确处理。还会添加一个末尾零字节。包围 PostgreSQL 字符串字面量所需的单引号包含在结果字符串中。

发生错误时，PQescapeLiteral 返回 NULL 并且一个合适的消息会被存储在 *conn* 对象中。

> **提示**
>
> 在处理来自不可信来源的字符串时，正确转义尤其重要。否则就会有安全风险：你很容易受到“SQL 注入”攻击，不期望的 SQL 命令可能会被送入数据库。

注意，当一个数据值被作为PQexecParams 或相关例程中的一个独立参数传递时，没有必要做转义而且做转义也不正确。

PQescapeIdentifier

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

PQescapeIdentifier 对字符串进行转义，使其可用作 SQL 标识符，例如表名、列名或函数名。当用户提供的标识符可能包含未经转义就不会被 SQL 解析器视为标识符一部分的特殊字符，或包含需要保留大小写的大写字符时，此函数很有用。

PQescapeIdentifier 返回 *str* 参数作为 SQL 标识符的转义版本，存放在通过 malloc() 分配的内存中。结果不再需要时，必须使用 PQfreemem() 释放该内存。输入不必以零字节结尾，末尾零字节也不应计入 *length*。（如果在处理完 *length* 个字节之前遇到末尾零字节，PQescapeIdentifier 会在该字节处停止；这一行为类似于 strncpy。）返回字符串中的所有特殊字符均已替换，以便正确地作为 SQL 标识符处理。还会添加一个末尾零字节，并用双引号包围返回的字符串。

发生错误时，PQescapeIdentifier 返回 NULL 并且一个合适的消息会被存储在 *conn* 对象中。

> **提示**
>
> 与字符串字面量一样，为防止 SQL 注入攻击，从不可信来源接收到的 SQL 标识符必须经过转义。

PQescapeStringConn

```
size_t PQescapeStringConn(PGconn *conn,
                          char *to, const char *from, size_t length,
                          int *error);
```

PQescapeStringConn 对字符串字面量进行转义，与 PQescapeLiteral 类似。与 PQescapeLiteral 不同，调用者需要提供大小合适的缓冲区。此外，PQescapeStringConn 不会生成包围 PostgreSQL 字符串字面量所需的单引号；应在包含转义结果的 SQL 命令中提供这些引号。*from* 参数指向待转义字符串的首字符，*length* 参数给出该字符串的字节数。输入不必以零字节结尾，末尾零字节也不应计入 *length*。（如果在处理完 *length* 个字节之前遇到末尾零字节，PQescapeStringConn 会在该字节处停止；这一行为类似于 strncpy。）*to* 必须指向一个缓冲区，其容量至少为 *length* 的两倍加一个字节，否则行为未定义。如果 *to* 与 *from* 字符串重叠，行为同样未定义。

如果 *error* 参数不是 NULL，则成功时将 *error 设为零，出错时设为非零。目前唯一可能的错误是源字符串中存在无效的多字节编码。出错时仍会生成输出字符串，但预计服务器会因其格式错误而拒绝它。发生错误时，无论 *error* 是否为 NULL，都会在 *conn* 对象中保存相应消息。

PQescapeStringConn 返回写到 *to* 的字节数，不包括终止的零字节。

PQescapeString

PQescapeString 是 PQescapeStringConn 的旧版本，现已弃用。

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

与 PQescapeStringConn 唯一的区别是，PQescapeString 不接受 PGconn 或 *error* 参数。因此，它无法根据连接属性（例如字符编码）调整行为，可能给出错误的结果。此外，它也无法报告错误情况。

PQescapeString 可以在同一时刻仅使用一个 PostgreSQL 连接的客户端程序中安全使用（在这种情况下，它能够“在内部”取得所需信息）。在其他情形下，它存在安全隐患，应改用 PQescapeStringConn。

PQescapeByteaConn

对二进制数据进行转义，使其能够在 SQL 命令中用作 bytea 类型的值。与 PQescapeStringConn 一样，这仅用于将数据直接插入 SQL 命令字符串的情况。

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

当某些字节值被用作一个 SQL 语句中的 bytea 字面量的一部分时，它们必须被转义。PQescapeByteaConn 使用十六进制编码或反斜杠转义来转义这些字节。详见第 8.4 节。

from 参数指向待转义字符串的首字节，*from_length* 参数给出该二进制字符串的字节数。（末尾零字节既不需要，也不计入长度。）*to_length* 参数指向用于保存转义后字符串长度的变量。该结果字符串长度包含结果末尾的零字节。

PQescapeByteaConn 返回 *from* 参数所指二进制字符串的转义版本，存放在通过 malloc() 分配的内存中。结果不再需要时，应使用 PQfreemem() 释放该内存。返回字符串中的所有特殊字符都已替换，以便 PostgreSQL 字符串字面量解析器和 bytea 输入函数正确处理。还会添加一个末尾零字节。包围 PostgreSQL 字符串字面量所需的单引号不包含在结果字符串中。

在发生错误时，将返回一个空指针，并且一个合适的错误消息被存储在*conn* 对象中。当前，唯一可能的错误是没有足够的内存用于结果串。

PQescapeBytea

PQescapeBytea 是 PQescapeByteaConn 的旧版本，现已弃用。

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

与 PQescapeByteaConn 唯一的区别是，PQescapeBytea 不接受 PGconn 参数。因此，PQescapeBytea 只能在同一时刻仅使用一个 PostgreSQL 连接的客户端程序中安全使用（在这种情况下，它能够“在内部”取得所需信息）。如果在使用多个数据库连接的程序中使用，它可能给出错误的结果（此时应使用 PQescapeByteaConn）。

PQunescapeBytea

将二进制数据的字符串表示转换为二进制数据，这是PQescapeBytea 的逆操作。以文本格式取得 bytea 数据时需要此操作；以二进制格式取得时则不需要。

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

from 参数指向一个字符串，例如对bytea 列调用 PQgetvalue 时返回的字符串。PQunescapeBytea 将这个字符串表示转换为二进制表示。它返回指向通过 malloc() 分配

的缓冲区的指针，出错时返回NULL，并将缓冲区大小存入*to_length*。不再需要结果时，必须使用PQfreemem 释放它。

此转换并不完全是 PQescapeBytea 的逆操作，因为从PQgetvalue 收到的字符串并非经过“转义”的形式。具体而言，这意味着无需考虑字符串引号，因此也不需要PGconn 参数。

34.4. 异步命令处理

该 PQexec 函数足以满足普通同步应用程序提交命令的需要。不过，它有一些缺点，对某些用户可能很重要：

- PQexec 会等待命令完成。该应用可能有其他的工作要做（例如维护用户界面），这时它将不希望阻塞等待回应。
- 因为客户端应用的执行在它等待结果时会被挂起，对于应用来说很难决定要不要尝试取消正在进行的命令（这可以在一个信号处理器中完成，但别无他法）。
- PQexec 只能返回一个 PGresult 结构体。如果提交的命令串包含多个 SQL 命令，除了最后一个PGresult 之外都会被 PQexec 丢弃。
- PQexec 总是收集命令的完整结果，并将其缓存在单个 PGresult 中。这简化了应用程序的错误处理逻辑，但对于包含大量行的结果，可能不切实际。

如果应用程序不希望受到这些限制，可以改用构成 PQexec 的底层函数：PQsendQuery 和 PQgetResult。此外，PQsendQueryParams、PQsendPrepare、PQsendQueryPrepared、PQsendDescribePrepared、PQsendDescribePortal 可以与 PQgetResult 配合使用，分别实现 PQexecParams、PQprepare、PQexecPrepared、PQdescribePrepared、PQdescribePortal 的功能。

PQsendQuery

向服务器提交命令，不等待结果。命令发送成功时返回 1，否则返回 0（此时可使用 PQerrorMessage 获取更多失败信息）。

```
int PQsendQuery(PGconn *conn, const char *command);
```

成功调用 PQsendQuery 后，应调用 PQgetResult 一次或多次来获取结果。在 PQgetResult 返回空指针、表明命令已完成之前，不得在同一连接上再次调用 PQsendQuery。

PQsendQueryParams

向服务器提交命令及独立指定的参数，不等待结果。

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

该函数等价于 PQsendQuery，但查询参数可以与查询字符串分开指定。函数参数的处理方式与 PQexecParams 相同。与 PQexecParams 一样，它不能用于协议 2.0 的连接，并且查询字符串中只允许包含一条命令。

PQsendPrepare

发送按给定参数创建预备语句的请求，不等待完成。

```
int PQsendPrepare(PGconn *conn,
                  const char *stmtName,
                  const char *query,
                  int nParams,
                  const Oid *paramTypes);
```

这是 PQprepare 的异步版本：请求发送成功时返回 1，否则返回 0。调用成功后，再调用 PQgetResult，确定服务器是否成功创建了预备语句。函数参数的处理方式与 PQprepare 相同。与 PQprepare 一样，它不能用于协议 2.0 的连接。

PQsendQueryPrepared

发送使用给定参数执行预备语句的请求，不等待结果。

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

该函数类似于 PQsendQueryParams，但通过已创建的预备语句的名称指定要执行的命令，而非提供查询字符串。函数参数的处理方式与 PQexecPrepared 相同。与 PQexecPrepared 一样，它不能用于协议 2.0 的连接。

PQsendDescribePrepared

提交获取指定预备语句信息的请求，不等待完成。

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

这是 PQdescribePrepared 的异步版本：请求发送成功时返回 1，否则返回 0。调用成功后，再调用PQgetResult 获取结果。函数参数的处理方式与 PQdescribePrepared 相同。与 PQdescribePrepared 一样，它不能用于协议 2.0 的连接。

PQsendDescribePortal

提交获取指定 portal 信息的请求，不等待完成。

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

这是 PQdescribePortal 的异步版本：请求发送成功时返回 1，否则返回 0。调用成功后，再调用PQgetResult 获取结果。函数参数的处理方式与 PQdescribePortal 相同。与 PQdescribePortal 一样，它不能用于协议 2.0 的连接。

PQgetResult

等待先前的 PQsendQuery, PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared, 或PQsendDescribePortal 调用产生的下一个结果，并返回该结果。当命令执行完毕且不再有其他结果时，返回空指针。

```
PGresult *PQgetResult(PGconn *conn);
```

必须反复调用 PQgetResult，直到它返回空指针，表明命令已经完成。（如果当前没有正在执行的命令，调用 PQgetResult 会立即返回空指针。）对于 PQgetResult 返回的非空指针，应使用前文介绍的 PGresult 访问函数处理相应结果。使用完毕后，不要忘记调用 PQclear 释放每个结果对象。注意，只有存在正在执行的命令，且所需响应数据尚未被 PQconsumeInput 读取时，PQgetResult 才会阻塞。

注意

即使 PQresultStatus 指示发生了致命错误，也应该调用PQgetResult 直到它返回一个空指针，以便libpq 完全处理错误信息。

使用 PQsendQuery 和 PQgetResult 可以解决 PQexec 的一个问题：如果命令字符串包含多个 SQL 命令，就能分别获取这些命令的结果。（这也支持一种简单的重叠处理方式：客户端可以处理某条命令的结果，同时服务器继续处理同一命令字符串中后面的查询。）

使用 PQsendQuery 和 PQgetResult 还可以实现另一项常见需求：从大型查询结果中一次读取一行。详见第 34.5 节。

仅仅调用 PQgetResult 仍会使客户端阻塞，直到服务器完成下一条 SQL 命令。可以通过正确使用另外两个函数来避免这种情况：

PQconsumeInput

如果服务器有可读取的输入，则读取这些输入。

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput 通常返回 1，表示“没有错误”；发生问题时则返回 0（此时可查看 PQerrorMessage）。注意，返回值并不说明是否实际读取了输入数据。调用 PQconsumeInput 后，应用程序可以检查 PQisBusy 和/或 PQnotifies，以确定其状态是否发生变化。

即使应用程序尚未准备好处理结果或通知，也可以调用 PQconsumeInput。此函数会读取可用数据并将其保存在缓冲区中，从而清除 select() 的可读就绪指示。因此，应用程序可以用 PQconsumeInput 立即清除 select() 的就绪条件，随后在合适的时候检查结果。

PQisBusy

如果命令仍在忙碌，则返回 1，意味着PQgetResult 会阻塞等待输入。返回 0 则表示可以调用 PQgetResult，并保证不会阻塞。

```
int PQisBusy(PGconn *conn);
```

PQisBusy 本身将不会尝试从服务器读取数据，因此必须先调用PQconsumeInput，否则繁忙状态将永远不会结束。

使用这些函数的典型应用程序会在主循环中通过 select() 或 poll() 等待需要响应的各种条件。其中一个条件是服务器有可读取的输入；对于 select()，这意味着 PQsocket 标识的文件描述符上有可读数据。主循环检测到输入就绪时，应调用 PQconsumeInput 读取输入，然后调用 PQisBusy。如果 PQisBusy 返回假（0），就可以接着调用 PQgetResult。还可以调用 PQnotifies 检测 NOTIFY 消息（见第 34.8 节）。

一个使用 PQsendQuery/PQgetResult 的客户端也可以尝试取消一个正在被服务器处理的命令，见第 34.6 节。但是，不管PQcancel 的返回值是什么，应用都必须继续使用PQgetResult 进行正常的结果读取序列。一次成功的取消只会导致命令比不取消时更快终止。

使用上述函数可以避免在等待数据库服务器输入时阻塞。不过，应用程序仍可能在等待向服务器发送输出时阻塞。这种情况较少见，但发送很长的 SQL 命令或数据值时可能发生。（如果应用程序通过 COPY IN 发送数据，发生的可能性则大得多。）为了防止这种情况，实现完全非阻塞的数据库操作，可以使用以下附加函数。

PQsetnonblocking

设置连接的非阻塞状态。

```
int PQsetnonblocking(PGconn *conn, int arg);
```

如果 *arg* 为 1，则将连接状态设置为非阻塞，如果 *arg* 为 0，则设置为阻塞。如果成功返回 0，出错返回-1。

在非阻塞状态下，对 PQsendQuery、PQputline、PQputnbytes、PQputCopyData 和 PQendcopy 的调用不会阻塞；如果需要再次调用，它们会返回错误。

请注意，PQexec 不遵守非阻塞模式；如果调用它，它将以阻塞方式执行。

PQisnonblocking

返回数据库连接的阻塞状态。

```
int PQisnonblocking(const PGconn *conn);
```

如果连接设置为非阻塞模式，则返回 1，如果为阻塞，则返回 0。

PQflush

尝试将发送队列中的输出数据发送到服务器。成功（或发送队列为空）时返回 0；因某种原因失败时返回 -1；如果尚未能发送队列中的全部数据，则返回 1（这种情况只可能发生在非阻塞连接上）。

```
int PQflush(PGconn *conn);
```

在非阻塞连接上发送命令或数据后，应调用 PQflush。如果返回 1，就等待套接字变为可读或可写。套接字可写时，再次调用 PQflush；可读时，先调用 PQconsumeInput，再调用 PQflush。重复上述步骤，直到 PQflush 返回 0。（必须检查套接字是否可读，并用 PQconsumeInput 读完输入，因为服务器可能在尝试向客户端发送数据时阻塞，例如发送 NOTICE 消息；在客户端读取这些数据之前，服务器不会读取客户端发送的数据。）当 PQflush 返回 0 后，等待套接字变为可读，再按前述方法读取响应。

34.5. 逐行检索查询结果

通常，libpq 会收集一条 SQL 命令的完整结果，并将其作为单个 PGresult 返回给应用程序。对于返回大量行的命令，这种方式可能不可行。在这种情况下，应用程序可以在单行模式下使用 PQsendQuery 和 PQgetResult。此模式在从服务器收到结果行时，每次将一行返回给应用程序。

要进入单行模式，请在成功调用 PQsendQuery（或其同类函数）后，立即调用 PQsetSingleRowMode。此模式选择仅对当前正在执行的查询生效。然后按照第 34.4 节所述，反复调用 PQgetResult，直到返回空指针。如果查询返回了行，每一行都作为独立的 PGresult 对象返回。这些对象看起来与普通查询结果相同，只是状态码为 PGRES_SINGLE_TUPLE，而不是 PGRES_TUPLES_OK。最后一行之后，或者查询返回零行时立即返回一个不含行、状态为 PGRES_TUPLES_OK 的对象，表示不会再有更多行到达。（但注意，仍然必须继续调用 PQgetResult，直到返回空指针。）所有这些 PGresult 对象都包含与该查询普通 PGresult 对象相同的行描述数据（列名、类型等）。每个对象都应像往常一样使用 PQclear 释放。

PQsetSingleRowMode

为当前正在执行的查询选择单行模式。

```
int PQsetSingleRowMode(PGconn *conn);
```

此函数只能在调用 PQsendQuery 或其某个同类函数后立即调用，并且必须在对该连接执行任何其他操作之前，例如 PQconsumeInput 或 PQgetResult。在正确时机调用时，此函数会为当前查询启用单行模式并返回 1；否则，模式保持不变，函数返回 0。无论哪种情况，当前查询完成后都会恢复普通模式。

> **小心**
>
> 处理查询时，服务器可能先返回一些行，然后遇到错误，导致查询中止。通常，libpq 会丢弃这些行，只报告错误。但在单行模式下，这些行已经返回给了应用程序。因此，应用程序会先看到一些 PGRES_SINGLE_TUPLE PGresult 对象，随后看到一个 PGRES_FATAL_ERROR 对象。为了保证正确的事务行为，如果查询最终失败，应用程序必须能够丢弃或撤销此前对这些行所做的全部操作。

34.6. 取消进行中的查询

客户端应用程序可以使用本节介绍的函数，请求取消服务器仍在处理的命令。

PQgetCancel

创建一个数据结构，其中包含取消通过特定数据库连接发出的命令所需的信息。

```
PGcancel *PQgetCancel(PGconn *conn);
```

PQgetCancel 基于一个 PGconn 连接对象创建 PGcancel 对象。如果给定的 *conn* 为 NULL 或无效连接，则返回NULL。PGcancel 是不透明结构体，不应由应用程序直接访问；它只能传给PQcancel 或 PQfreeCancel。

PQfreeCancel

释放由以下函数创建的数据结构：PQgetCancel。

```
void PQfreeCancel(PGcancel *cancel);
```

PQfreeCancel 释放之前由 PQgetCancel 创建的数据对象。

PQcancel

请求服务器停止处理当前命令。

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

返回值为 1 表示取消请求成功发送，为 0 表示未成功发送。如果未成功发送，*errbuf* 将填充解释性错误消息。*errbuf* 必须是大小为 *errbufsize* 的字符数组（推荐大小为 256 字节）。

不过，成功发送请求并不保证请求会产生效果。如果取消生效，当前命令会提前终止并返回错误结果。如果取消失败（例如服务器已经处理完该命令），则不会产生任何可见结果。

如果 *errbuf* 是信号处理程序中的局部变量，则PQcancel 可以安全地从信号处理程序中调用。对于 PQcancel 而言，PGcancel 对象是只读的，因此也可以从与操作PGconn 对象的线程不同的线程中调用。

PQrequestCancel

PQrequestCancel 是以下函数的已弃用变体：PQcancel。

```
int PQrequestCancel(PGconn *conn);
```

请求服务器放弃当前命令的处理。它直接作用于 PGconn 对象，失败时会把错误消息存储到 PGconn 对象中（可通过 PQerrorMessage 获取）。虽然功能相同，但这种方法在多线程程序或信号处理程序中并不安全，因为它可能覆盖PGconn 中的错误消息，从而破坏当前连接上正在进行的操作。

34.7. 快速路径接口

PostgreSQL 提供一种快速路径接口来向服务器发送简单的函数调用。

> **提示**
>
> 此接口已显得有些过时，因为可以通过建立定义函数调用的预备语句，获得相近的性能和更多功能。随后使用二进制格式传输参数和结果来执行该语句，就可以替代快速路径函数调用。

函数 PQfn 请求通过快速路径接口执行服务器函数。

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct
{
    int len;
    int isint;
    union
    {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

fnid 参数是要执行函数的 OID。*args* 和 *nargs* 指定传给函数的参数，必须与函数声明中的参数列表匹配。参数结构体的 *isint* 字段为真时，*u.integer* 值会以指定长度的整数发送到服务器，该长度必须是 2 或 4 字节，并会进行适当的字节序转换。*isint* 为假时，位于 **u.ptr* 的指定数量字节会原样发送；数据必须符合服务器对该函数参数数据类型的二进制传输格式要

求。（将 *u.ptr* 声明为 int * 是历史原因；将其视为 void * 更合适。）*result_buf* 指向用于存放函数返回值的缓冲区。调用者必须事先分配足够空间来保存返回值，这里不会检查！实际结果长度以字节为单位，返回到 *result_len* 指向的整数中。如果预期结果是 2 或 4 字节整数，将 *result_is_int* 设为 1，否则设为 0。将 *result_is_int* 设为 1 后，libpq 会按需转换字节序，使结果成为适合客户端机器的 int 值；注意，无论是哪种允许的结果大小，传入 **result_buf* 的都是 4 字节整数。*result_is_int* 为 0 时，服务器发送的二进制格式字节串会原样返回。（此时，将 *result_buf* 视为 void * 更合适。）

PQfn 总是返回有效的 PGresult 指针：成功时状态为 PGRES_COMMAND_OK，遇到问题时为 PGRES_FATAL_ERROR。使用结果前应检查其状态。不再需要结果时，调用者负责使用 PQclear 释放 PGresult。

要向函数传入 NULL 参数，将该参数结构体的 *len* 字段设为 -1；此时，*isint* 和 *u* 字段便不再相关。（但这仅适用于使用协议 3.0 及更高版本的连接。）

如果函数返回 NULL，则将 **result_len* 设为 -1，而不修改 **result_buf*。（这仅适用于使用协议 3.0 及更高版本的连接；在协议 2.0 中，既不修改 **result_len*，也不修改 **result_buf*。）

注意，使用此接口时无法处理集合值结果。此外，函数必须是普通函数，不能是聚合函数、窗口函数或过程。

34.8. 异步通知

PostgreSQL 通过 LISTEN 和 NOTIFY 命令提供异步通知。客户端会话可使用 LISTEN 命令注册其感兴趣的特定通知通道（也可以用 UNLISTEN 命令停止监听）。当任何会话执行带有该通道名的 NOTIFY 命令时，所有监听该通道的会话都会异步收到通知。还可以传递一个“载荷”字符串，向监听者传递附加数据。

libpq 应用把 LISTEN、UNLISTEN 和 NOTIFY 命令作为普通 SQL 命令提交。随后通过调用 PQnotifies 来检测 NOTIFY 消息的到达。

函数 PQnotifies 从已收到但尚未处理的服务器通知消息列表中返回下一条通知。没有待处理通知时返回空指针。一旦 PQnotifies 返回一条通知，该通知就被视为已处理，并从通知列表中移除。

```
PGnotify *PQnotifies(PGconn *conn);

typedef struct pgNotify
{
    char *relname;              /* 通知通道名称 */
    int  be_pid;                /* 发送通知的服务器进程 ID */
    char *extra;                /* 通知载荷字符串 */
} PGnotify;
```

处理完 PQnotifies 返回的 PGnotify 对象后，务必使用 PQfreemem 释放它。只需释放 PGnotify 指针；relname 和 extra 字段并非独立分配。（这些字段名称是历史遗留的；尤其是，通道名称与关系名称不必有任何关联。）

例 34.2 给出了一个示例程序展示异步通知的使用。

PQnotifies 实际上并不从服务器读取数据；它只是返回之前已被其他 libpq 函数读取的消息。在较早版本的 libpq 中，及时收到NOTIFY 消息的唯一方法是不断提交命令，哪怕是空命令，然

后在每次PQexec 后检查 PQnotifies。虽然这种方法仍然有效，但由于浪费处理能力，现已弃用。

当没有需要执行的实际命令时，一种更好的检查NOTIFY 消息的方法是调用 PQconsumeInput，然后检查PQnotifies。你可以使用 select() 等待服务器数据到达，这样在无事可做时就不会浪费CPU 资源（参见 PQsocket 以获得可传给 select() 的文件描述符）。注意不管是用 PQsendQuery/PQgetResult 提交命令还是简单地使用 PQexec，这种方法都能正常工作。不过，你应该记住在每次PQgetResult 或 PQexec 之后检查 PQnotifies，看看在命令的处理过程中是否有通知到达。

34.9. COPY 命令相关的函数

PostgreSQL 的 COPY 命令提供了选项，可以通过 libpq 使用的网络连接读取或写入数据。本节介绍的函数允许应用程序通过提供或接收复制数据来使用这一能力。

整体流程如下：应用程序先通过 PQexec 或等效函数发出 SQL COPY 命令。如果命令没有错误，响应就是一个 PGresult 对象，其状态码为 PGRES_COPY_OUT 或 PGRES_COPY_IN，取决于指定的复制方向。应用程序随后应使用本节函数接收或发送数据行。数据传输完成后，会返回另一个 PGresult 对象，表示传输成功或失败：成功时状态为 PGRES_COMMAND_OK，出现问题时为 PGRES_FATAL_ERROR。此时可以通过 PQexec 继续发出 SQL 命令。（COPY 操作进行期间，不能在同一连接上执行其他 SQL 命令。）

如果一个 COPY 命令是通过 PQexec 在一个可能包含额外命令的字符串中发出的，那么应用在完成COPY 序列之后必须继续用 PQgetResult 取得结果。只有在 PQgetResult 返回 NULL 时，我们才能确信PQexec 的命令字符串已经处理完毕，并且可以安全地发出更多命令。

只有从 PQexec 或 PQgetResult 获得 PGRES_COPY_OUT 或 PGRES_COPY_IN 结果状态后，才应调用本节函数。

带有上述某个状态值的 PGresult 对象，还会携带关于即将开始的 COPY 操作的附加数据。这些数据可以通过下列函数获取，这些函数也用于查询结果：

PQnfields

返回要复制的列（字段）的数量。

PQbinaryTuples

0 表示整体复制格式为文本（行由换行符分隔，列由分隔符分隔等）。1 表示整体复制格式为二进制。有关更多信息，请参见 COPY。

PQfformat

返回与复制操作的每列关联的格式代码（0 表示文本，1 表示二进制）。当整体复制格式为文本时，每列的格式代码始终为零，但二进制格式可以支持文本和二进制列。（但是，在当前的 COPY 实现中，只有二进制列出现在二进制复制中；因此，每列格式目前始终与整体格式匹配。）

> **注意**
>
> 这些附加数据值仅在使用协议 3.0 时可用。使用协议 2.0 时，这些函数都返回 0。

34.9.1. 用于发送 COPY 数据的函数

这些函数用于在 COPY FROM STDIN 期间发送数据。如果连接不处于 COPY_IN 状态，调用它们会失败。

PQputCopyData

在 COPY_IN 状态中向服务器发送数据。

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

将指定 *buffer* 中长度为 *nbytes* 的 COPY 数据传输到服务器。数据成功加入队列时返回 1；因缓冲区已满而无法加入队列时返回零（仅可能发生在非阻塞模式下）；发生错误时返回 -1。（返回 -1 时，可用 PQerrorMessage 获取详细信息。返回零时，应等待可写就绪后重试。）

应用程序可以将 COPY 数据流分成任意方便大小的数据块，逐块装入缓冲区。发送时，这些数据块的边界没有语义含义。数据流内容必须符合 COPY 命令预期的数据格式；详见 COPY。

PQputCopyEnd

在 COPY_IN 状态中向服务器发送数据结束的指示。

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

如果 *errormsg* 为 NULL，则成功结束 COPY_IN 操作。如果 *errormsg* 不为 NULL，则强制 COPY 失败，并将 *errormsg* 指向的字符串用作错误消息。（但不应假定服务器一定会返回这条完全相同的错误消息，因为服务器可能已经因自身原因使 COPY 失败。还要注意，在使用 3.0 之前协议的连接上，强制失败选项不起作用。）

终止消息已发送时返回 1；在非阻塞模式下，返回 1 也可能仅表示该消息已成功加入发送队列。（在非阻塞模式下，要确认数据已经发送，应接着等待可写就绪并调用 PQflush，反复执行直到返回零。）返回零表示缓冲区已满，无法将终止消息加入队列；这种情况仅可能发生在非阻塞模式下。（此时，应等待可写就绪，再次调用 PQputCopyEnd。）发生严重错误时返回 -1，可用 PQerrorMessage 获取详细信息。

成功调用 PQputCopyEnd 后，调用 PQgetResult 获取 COPY 命令的最终结果状态。可以按通常方式等待该结果就绪，然后恢复正常操作。

34.9.2. 用于接收 COPY 数据的函数

这些函数用于在 COPY TO STDOUT 的过程中接收数据。如果连接不在 COPY_OUT 状态，那么调用它们将会失败。

PQgetCopyData

在 COPY_OUT 状态下从服务器接收数据。

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

在 COPY 期间尝试从服务器获取下一行数据。每次总是返回一个完整数据行；如果只有部分行可用，则不返回。成功返回数据行时，会分配一块内存保存数据。*buffer* 参数必须为非 NULL。**buffer* 会被设置为指向所分配的内存；如果没有返回缓冲区，则设为 NULL。非 NULL 的结果缓冲区在不再需要时应使用 PQfreemem 释放。

成功返回一行时，返回值是该行的数据字节数，始终大于零。返回的字符串总是以零字节结尾，不过这可能仅对文本 COPY 有用。返回零表示 COPY 仍在进行，但尚无可用行（仅在 *async* 为真时可能发生）。返回 -1 表示 COPY 已完成；返回 -2 表示发生了错误（可用 PQerrorMessage 查看原因）。

当 *async* 为真（非零）时，PQgetCopyData 不会阻塞等待输入；如果 COPY 仍在进行，但没有完整行可用，则返回零。（此时，应等待读就绪，先调用 PQconsumeInput，再调用 PQgetCopyData。）当 *async* 为假（零）时，PQgetCopyData 会阻塞，直到数据可用或操作完成。

在 PQgetCopyData 返回 -1 后，调用 PQgetResult 获取 COPY 命令的最终结果状态。可以按通常方式等待该结果就绪，然后恢复正常操作。

34.9.3. 用于 COPY 的过时函数

这些函数使用较旧的方式处理 COPY。虽然仍然可用，但由于错误处理欠佳、检测数据结束的方式不便，而且缺少对二进制或非阻塞传输的支持，已被弃用。

PQgetline

将服务器传来的、以换行符结尾的一行字符读入大小为 *length* 的字符串缓冲区。

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

此函数最多将 *length*-1 个字符复制到缓冲区，并将末尾的换行符转换为零字节。PQgetline 在输入结束时返回 EOF，读完一整行时返回 0，缓冲区已满但尚未读到末尾换行符时返回 1。

注意，应用程序必须检查新读入的一行是否仅由 \. 两个字符组成，这表示服务器已发送完 COPY 命令的结果。如果可能收到长度超过 *length*-1 个字符的行，必须确保正确识别 \. 行，例如不能把长数据行的末尾误当作终止行。

PQgetlineAsync

以非阻塞方式将服务器传来的一行 COPY 数据读入缓冲区。

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

此函数类似于 PQgetline，但可用于必须异步读取 COPY 数据的应用程序，即读取时不阻塞。发出 COPY 命令并收到 PGRES_COPY_OUT 响应后，应用程序应调用 PQconsumeInput 和 PQgetlineAsync，直到检测到数据结束信号。

与 PQgetline 不同，此函数会负责检测数据结束。

每次调用时，如果 libpq 的输入缓冲区中有完整数据行，PQgetlineAsync 就会返回数据；否则，要等该行剩余部分到达后才返回数据。识别到复制数据结束标记时返回 -1，没有可用数据时返回 0，否则返回正数，表示返回的数据字节数。返回 -1 后，调用者必须接着调用 PQendcopy，然后恢复正常处理。

返回的数据不会跨越数据行边界。只要可能，每次就返回一整行；但如果调用者提供的缓冲区太小，容不下服务器发送的一行，则只返回部分行。对于文本数据，可检查最后返回的字节是否为 \n，以判断是否返回了完整行。（对于二进制 COPY，则必须实际解析 COPY 数据格式才能作出相同判断。）返回的字符串不以零字节结尾。（如果要自行添加末尾的零字节，务必将传入的 *bufsize* 设置为比实际可用空间少一字节。）

PQputline

向服务器发送以零字节结尾的字符串。成功时返回 0，无法发送字符串时返回 EOF。

```
int PQputline(PGconn *conn,
              const char *string);
```

连续调用 PQputline 发送的 COPY 数据流，与 PQgetlineAsync 返回的数据格式相同。不过，应用程序不必在每次 PQputline 调用中恰好发送一个数据行；每次发送部分行或多行也可以。

> **注意**
>
> 在 PostgreSQL 协议 3.0 之前，应用程序必须显式发送由 \. 两个字符组成的最后一行，告知服务器应用程序已发送完 COPY 数据。虽然这种方式仍然有效，但已被弃用，\. 的特殊含义预计会在未来版本中移除。发送完实际数据后，调用 PQendcopy 即可。

PQputnbytes

向服务器发送不以零字节结尾的字符串。成功时返回 0，无法发送字符串时返回 EOF。

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

此函数与 PQputline 完全相同，只是直接指定了要发送的字节数，因此数据缓冲区不必以零字节结尾。发送二进制数据时可使用此函数。

PQendcopy

与服务器同步。

```
int PQendcopy(PGconn *conn);
```

此函数会等待服务器完成复制。应在使用 PQputline 向服务器发送最后一个字符串后，或使用 PQgetline 从服务器接收最后一个字符串后调用它。必须调用此函数，否则服务器与客户端会“失去同步”。函数返回后，服务器便准备好接收下一条 SQL 命令。成功完成时返回 0，否则返回非零值。（返回非零值时，可用 PQerrorMessage 获取详细信息。）

使用 PQgetResult 时，收到 PGRES_COPY_OUT 结果后，应用程序应反复调用 PQgetline，并在看到终止行后调用 PQendcopy。随后应回到 PQgetResult 循环，直到 PQgetResult 返回空指针。类似地，收到 PGRES_COPY_IN 结果后，应连续调用 PQputline，再调用 PQendcopy，然后回到 PQgetResult 循环。这样可以保证嵌在一系列 SQL 命令中的 COPY 命令正确执行。

旧的应用很可能会通过 PQexec 提交一个 COPY 命令并且假定事务在 PQendcopy 之后完成。只有在 COPY 是命令字符串中唯一的 SQL 命令时才能正确工作。

34.10. 控制函数

这些函数控制 libpq 行为的各种细节。

PQclientEncoding

返回客户端编码。

```
int PQclientEncoding(const PGconn *conn);
```

注意，返回的是编码 ID，而不是 EUC_JP 这样的符号字符串。失败时返回 -1。要将编码 ID 转换为编码名称，可以使用：

```
char *pg_encoding_to_char(int encoding_id);
```

PQsetClientEncoding

设置客户端编码。

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

conn 是到服务器的连接，而*encoding* 是要使用的编码。如果成功设置编码，函数返回 0，否则返回 -1。此连接的当前编码可以通过以下函数确定：PQclientEncoding。

PQsetErrorVerbosity

设置以下函数所返回消息的详细程度：PQerrorMessage 和 PQresultErrorMessage。

```
typedef enum
{
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

PQsetErrorVerbosity 设置详细程度模式，并返回该连接先前的设置。在TERSE 模式下，返回的消息只包括严重性、主要文本和位置，通常一行就能容纳。默认模式生成的消息除上述字段外，还包括详细信息、提示或上下文字段（这些内容可能跨越多行）。VERBOSE 模式包括所有可用字段。改变详细程度不会影响已有 PGresult 对象中的消息，只影响随后创建的对象。（如果要以不同的详细程度打印先前的错误，请参见PQresultVerbose ErrorMessage。）

PQsetErrorContextVisibility

确定对 CONTEXT 字段的处理方式，这些字段位于以下函数返回的消息中：PQerrorMessage 和 PQresultErrorMessage。

```
typedef enum
{
    PQSHOW_CONTEXT_NEVER,
    PQSHOW_CONTEXT_ERRORS,
    PQSHOW_CONTEXT_ALWAYS
} PGContextVisibility;

PGContextVisibility PQsetErrorContextVisibility(PGconn *conn, PGContextVisibility show_
context);
```

PQsetErrorContextVisibility 设置上下文显示模式，并返回连接的先前设置。此模式控制消息中是否包含CONTEXT 字段（除非详细程度设置为 TERSE，此时CONTEXT 始终不会显示）。NEVER 模式从不包含 CONTEXT，而ALWAYS 在该字段可用时总是包含它。在 ERRORS 模式（默认）下，CONTEXT 字段只包含在错误消息中，不包含在通知和警告中。改变此模式不会影响已有PGresult 对象中的消息，只影响随后创建的对象。（如果要以不同的显示模式打印先前的错误，请参见PQresultVerboseErrorMessage。）

PQtrace

启用客户端/服务器通信跟踪，将跟踪信息写入调试文件流。

```
void PQtrace(PGconn *conn, FILE *stream);
```

注意

在 Windows 上，如果 libpq 库与应用程序使用不同的标志编译，调用此函数会导致应用程序崩溃，因为 FILE 指针的内部表示不同。具体而言，库与所有使用它的应用程序都应使用相同的多线程/单线程、发布/调试和静态/动态标志。

PQuntrace

禁用由 PQtrace 启动的跟踪。

```
void PQuntrace(PGconn *conn);
```

34.11. 杂项函数

一如往常，总有一些函数不适合放在任何其他地方。

PQfreemem

释放 libpq 分配的内存。

```
void PQfreemem(void *ptr);
```

释放 libpq 分配的内存，特别是 PQescapeByteaConn、PQescapeBytea、PQunescapeBytea 和 PQnotifies 分配的内存。在 Microsoft Windows 上，务必使用此函数，而不是 free()。这是因为只有 DLL 与应用程序使用相同的多线程/单线程、发布/调试和静态/动态标志，才能在 DLL 中分配内存并在应用程序中释放它。在 Microsoft Windows 以外的平台上，此函数与标准库函数 free() 相同。

PQconninfoFree

释放 PQconndefaults 或 PQconninfoParse 分配的数据结构。

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

仅调用 PQfreemem 不足以完成此项释放，因为数组还包含指向附属字符串的引用。

PQencryptPasswordConn

准备一个 PostgreSQL 密码的加密形式。

```
char *PQencryptPasswordConn(PGconn *conn, const char *passwd, const char *user, const char
 *algorithm);
```

这个函数旨在用于那些希望发送类似于 ALTER USER joe PASSWORD 'pwd' 命令的客户端应用。不在这样一个命令中发送原始的明文密码是一个好习惯，因为它可能被暴露在命令日志、活动显示等等中。相反，在发送之前使用这个函数可以将密码转换为加密的形式。

passwd 和 *user* 参数是明文密码以及用户的 SQL 名称。*algorithm* 指定用来加密密码的加密算法。当前支持的算法是 md5 和 scram-sha-256（on 和 off 也被接受作为 md5 的别名，用于与较老的服务器版本兼容）。注意，对scram-sha-256 的支持是在 PostgreSQL 版本 10 中引入的，并且在老的服务器版本上无法工作。如果*algorithm* 是 NULL，这个函数将向服务器查询password_encryption 设置的当前值。这一查询可能阻塞，并且当前事务被中止或者连接正忙于执行另一个查询时会失败。如果希望为服务器使用默认的算法但避免阻塞，应在调用PQencryptPasswordConn 之前自行查询 password_encryption，并且将该值作为*algorithm* 传入。

返回值是一个由 malloc 分配的字符串。调用者可以假设该字符串不含有需要转义的任何特殊字符。在处理完它之后，用PQfreemem 释放结果。发生错误时，返回的是NULL，并且适当的消息会被存储在连接对象中。

PQencryptPassword

准备一个 PostgreSQL 密码的 md5 加密形式。

```
char *PQencryptPassword(const char *passwd, const char *user);
```

PQencryptPassword 是 PQencryptPasswordConn 的旧版本，现已弃用。其差别是 PQencryptPassword 不需要连接对象，并且总是用md5 作为加密算法。

PQmakeEmptyPGresult

用给定的状态，构造一个空PGresult 对象。

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

这是 libpq 内部用于分配并初始化一个空 PGresult 对象的函数。如果无法分配内存，此函数返回NULL。将它导出供外部调用，是因为一些应用需要自行生成结果对象，特别是带有错误状态的对象。如果*conn* 不为 null，并且*status* 表示错误，指定连接的当前错误消息会被复制到PGresult 中。此外，如果*conn* 不为 null，连接中注册的所有事件过程也会被复制到PGresult 中。（这些过程不会收到 PGEVT_RESULTCREATE 调用，但可参见PQfireResultCreateEvents。）注意，最终应对该对象调用PQclear，就像处理libpq 自身返回的 PGresult 一样。

PQfireResultCreateEvents

为每一个在 PGresult 对象中注册的事件过程触发一个 PGEVT_RESULTCREATE 事件（见第 34.13 节）。成功时返回非 0，如果任何事件过程失败则返回 0。

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

conn 参数会传递给事件过程，但此函数不会直接使用它。如果事件过程不使用此参数，则可以传入 NULL。

已经接收到这个对象的 PGEVT_RESULTCREATE 或 PGEVT_RESULTCOPY 事件的事件过程不会被再次触发。

此函数与 PQmakeEmptyPGresult 分开的主要原因是，通常适合先创建 PGresult 并填充数据，然后再调用事件过程。

PQcopyResult

创建 PGresult 对象的副本。副本与源结果没有任何关联，不再需要副本时必须调用 PQclear。函数失败时返回 NULL。

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

这不是为了制作一个精确的副本。返回的结果总是放在 PGRES_TUPLES_OK 状态中，并且不复制源中的任何错误消息。（但是会复制命令状态字符串。）*flags* 参数确定要复制的其他内容。它是几个标志的按位或。PG_COPYRES_ATTRS 指定复制源结果的属性（列定义）。PG_COPYRES_TUPLES 指定复制源结果的元组。（这也意味着复制属性。）PG_COPYRES_NOTICEHOOKS 指定复制源结果的通知钩子。PG_COPYRES_EVENTS 指定复制源结果的事件。（但不复制与源相关的任何实例数据。）

PQsetResultAttrs

设置 PGresult 对象的属性。

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc *attDescs);
```

提供的 *attDescs* 被复制到结果中。如果 *attDescs* 指针为 NULL 或 *numAttributes* 小于 1，那么请求将被忽略并且函数成功。如果*res* 已经包含属性，那么函数会失败。如果函数失败，返回值是 0。如果函数成功，返回值是非 0。

PQsetvalue

设置 PGresult 对象中某个元组的字段值。

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char *value, int len);
```

此函数会根据需要自动扩展结果内部的元组数组。不过，*tup_num* 参数必须小于或等于 PQntuples，也就是说，每次只能向元组数组增加一个元组。已有元组的任何字段都可以按任意顺序修改。如果 *field_num* 指定的位置已有值，该值会被覆盖。如果 *len* 为 -1 或 *value* 为 NULL，则将该字段设置为 SQL null 值。*value* 会被复制到结果的私有存储中，因此函数返回后就不再需要它。函数失败时返回零，成功时返回非零值。

PQresultAlloc

为一个 PGresult 对象分配附属存储。

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

使用此函数分配的所有内存都会在清除 *res* 时释放。函数失败时返回 NULL。与 malloc 一样，返回的内存保证满足任意数据类型的对齐要求。

PQlibVersion

返回所使用的 libpq 版本。

```
int PQlibVersion(void);
```

可在运行时根据此函数的结果，判断当前已加载的 libpq 版本是否具有特定功能。例如，可用它判断 PQconnectdb 支持哪些连接选项。

返回值等于库的主版本号乘以 10000 再加上次版本号。例如，版本 10.1 返回 100001，版本 11.0 返回 110000。

在主版本 10 之前，PostgreSQL 使用由三个部分组成的版本号，前两个部分共同表示主版本。对于这些版本，PQlibVersion 用两位数字表示每个部分；例如，版本 9.1.5 返回 90105，版本 9.2.0 返回 90200。

因此，为了判断功能兼容性，应用程序应将 PQlibVersion 的结果除以 100 而非 10000，得到逻辑上的主版本号。在所有版本系列中，次版本（错误修复版本）之间只有最后两位数字不同。

> **注意**
>
> 此函数从 PostgreSQL 9.1 起提供，因而不能用它检测更早版本是否具有所需功能：调用它会建立对 9.1 或更高版本的链接依赖。

34.12. 通知处理

服务器生成的通知和警告消息不由查询执行函数返回，因为它们并不表示查询失败。这些消息会传递给通知处理函数，处理函数返回后，执行正常继续。默认通知处理函数将消息打印到 `stderr`，应用程序也可以提供自己的处理函数来替换这一行为。

由于历史原因，通知处理有两个级别，称为通知接收器和通知处理器。通知接收器的默认行为是格式化通知并且将一个字符串传递给通知处理器来打印。不过，如果一个应用选择提供自己的通知接收器，它通常会忽略通知处理器层并且在通知接收器中完成所有工作。

函数 `PQsetNoticeReceiver` 为一个连接对象设置或者检查当前的通知接收器。相似地，`PQsetNoticeProcessor` 设置或检查当前的通知处理器。

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                    PQnoticeReceiver proc,
                    void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

这些函数中的每一个会返回之前的通知接收器或处理器函数指针，并且设置新值。如果你提供了一个空函数指针，将不会采取任何动作，只会返回当前指针。

从服务器收到通知或警告消息，或者 libpq 内部生成这类消息时，便会调用通知接收器函数。消息以状态为 `PGRES_NONFATAL_ERROR` 的 `PGresult` 对象传给该函数。（接收器可使用 `PQresultErrorField` 提取单个字段，也可使用 `PQresultErrorMessage` 或 `PQresultVerboseErrorMessage` 获取完整的预格式化消息。）传给 `PQsetNoticeReceiver` 的同一个 void 指针也会传入。（必要时，可通过该指针访问应用程序特有的状态。）

默认的通知接收器会简单地抽取消息（使用 `PQresultErrorMessage`）并且将它传递给通知处理器。

通知处理器负责处理以文本形式提供的通知或警告消息。传给它的参数包括消息字符串（含末尾的换行符），以及先前传给 `PQsetNoticeProcessor` 的同一个 void 指针。（必要时，可通过该指针访问应用程序特有的状态。）

默认的通知处理器很简单：

```
static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}
```

设置通知接收器或处理器后，应当预期：只要 PGconn 对象或由它生成的 PGresult 对象仍存在，就可能调用该函数。创建 PGresult 时，会将 PGconn 当前的通知处理指针复制到 PGresult 中，供 PQgetvalue 等函数在需要时使用。

34.13. 事件系统

libpq 的事件系统被设计为通知已注册的事件处理器它感兴趣的 libpq 事件，例如PGconn 以及 PGresult 对象的创建和销毁。一个主要用途是允许应用将自己的数据与一个 PGconn 或者 PGresult 关联在一起，并且确保那些数据在适当的时候被释放。

每个注册的事件处理程序都与两项数据相关联，libpq 仅将其视为不透明的 void * 指针。有一个透传指针，由应用程序在向 PGconn 注册事件处理程序时提供。透传指针在 PGconn 及其生成的所有 PGresult 的生命周期内永远不会更改；因此，如果使用，它必须指向长期存在的数据。此外，还有一个实例数据指针，在每个PGconn 和 PGresult 中一开始都是 NULL。可以使用 PQinstanceData、PQsetInstanceData、PQresultInstanceData 和 PQresultSetInstanceData 函数来操作此指针。请注意，与透传指针不同，PGconn 的实例数据不会自动继承到从中创建的 PGresult。libpq 不知道透传和实例数据指针指向的内容（如果有的话），也永远不会尝试释放它们 – 这是事件处理程序的责任。

34.13.1. 事件类型

枚举 PGEventId 命名了事件系统处理的事件类型。它的所有值的名称都以 PGEVT 开始。对于每一种事件类型，都有一个相应的事件信息结构体用来承载传递给事件处理器的参数。事件类型是：

PGEVT_REGISTER

注册事件在 PQregisterEventProc 被调用时触发。此时最适合初始化事件过程可能需要的 instanceData。每个连接中的每个事件过程只会触发一次注册事件。如果事件过程失败，则中止注册。

```
typedef struct
{
    PGconn *conn;
} PGEventRegister;
```

收到 PGEVT_REGISTER 事件时，应将*evtInfo* 指针强制转换为 PGEventRegister *。此结构体包含一个 PGconn，它应处于CONNECTION_OK 状态；如果调用 PQregisterEventProc 紧接在取得正常的 PGconn 之后，就能保证这一点。返回失败代码时，必须完成全部清理工作，因为不会发送PGEVT_CONNDESTROY 事件。

PGEVT_CONNRESET

连接重置事件会在完成以下调用时触发：PQreset 或 PQresetPoll。在这两种情况下，只有重置成功才会触发该事件。如果事件过程失败，整个连接重置就会失败；PGconn 会被置于 CONNECTION_BAD 状态，并且PQresetPoll 将返回 PGRES_POLLING_FAILED。

```
typedef struct
{
    PGconn *conn;
} PGEventConnReset;
```

收到 PGEVT_CONNRESET 事件时，应将*evtInfo* 指针强制转换为 PGEventConnReset *。虽然其中的 PGconn 刚刚被重置，但所有事件数据都保持不变。应利用此事件重置、重新加载或重新查询相关联的instanceData。注意，即使事件过程未能处理PGEVT_CONNRESET，它仍会在连接关闭时收到PGEVT_CONNDESTROY 事件。

PGEVT_CONNDESTROY

连接销毁事件由以下调用触发：PQfinish。事件过程负责正确清理其事件数据，因为 libpq 无法管理这部分内存。如果不清理，就会造成内存泄漏。

```
typedef struct
{
    PGconn *conn;
} PGEventConnDestroy;
```

收到 PGEVT_CONNDESTROY 事件时，应将*evtInfo* 指针强制转换为 PGEventConnDestroy *。该事件触发于以下函数执行任何其他清理工作之前：PQfinish。事件过程的返回值会被忽略，因为无法通过以下函数报告失败：PQfinish。此外，事件过程失败不应中止清理不再使用的内存的过程。

PGEVT_RESULTCREATE

任何生成结果的查询执行函数都会触发结果创建事件，其中包括PQgetResult。只有成功创建结果后才会触发该事件。

```
typedef struct
{
    PGconn *conn;
    PGresult *result;
} PGEventResultCreate;
```

收到 PGEVT_RESULTCREATE 事件时，应将*evtInfo* 指针转换为 PGEventResultCreate *。其中，*conn* 是用于生成结果的连接。这是初始化需要与结果关联的 instanceData 的理想位置。如果事件过程失败，结果会被清除，失败也会向外传递。事件过程不得自行调用PQclear 来清除结果对象。返回失败代码时，必须完成所有清理工作，因为不会发送PGEVT_RESULTDESTROY 事件。

PGEVT_RESULTCOPY

结果复制事件会在调用 PQcopyResult 时触发。只有复制完成后才会触发该事件。只有为源结果成功处理过 PGEVT_RESULTCREATE 或 PGEVT_RESULTCOPY 事件的事件过程，才会收到PGEVT_RESULTCOPY 事件。

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

收到 PGEVT_RESULTCOPY 事件时，应将*evtInfo* 指针强制转换为 PGEventResultCopy *。其中，*src* 结果是复制源，而*dest* 结果是复制目标。可以利用此事件对 instanceData 进行深复制，因为PQcopyResult 无法完成这项工作。如果事件过程失败，整个复制操作就会失败，并且*dest* 结果将被清除。返回失败代码时，必须完成所有清理工作，因为不会为目标结果发送PGEVT_RESULTDESTROY 事件。

PGEVT_RESULTDESTROY

结果销毁事件由以下调用触发：PQclear。事件过程负责正确清理其事件数据，因为 libpq 无法管理这部分内存。如果不清理，就会造成内存泄漏。

```
typedef struct
{
    PGresult *result;
} PGEventResultDestroy;
```

收到 PGEVT_RESULTDESTROY 事件时，应将*evtInfo* 指针强制转换为 PGEventResultDestroy *。该事件触发于以下函数执行任何其他清理工作之前：PQclear。事件过程的返回值会被忽略，因为无法通过以下函数报告失败：PQclear。此外，事件过程失败不应中止清理不再使用的内存的过程。

34.13.2. 事件回调过程

PGEventProc

PGEventProc 是通过 typedef 定义的事件过程指针类型，也就是接收 libpq 事件的用户回调函数的指针类型。事件过程的签名必须为：

```
int eventproc(PGEventId evtId, void *evtInfo, void *passThrough)
```

evtId 参数指示发生了哪一种 PGEVT 事件。必须将 *evtInfo* 指针强制转换为适当的结构体类型，以获取关于该事件的更多信息。*passThrough* 参数是在注册事件过程时传给 PQregisterEventProc 的指针。函数应在成功时返回非零值，在失败时返回零。

在任何一个 PGconn 中，一个特定事件过程只能被注册一次。这是因为该过程的地址被用作查找键来标识相关的实例数据。

> **小心**
>
> 在 Windows 上，函数可能有两个不同的地址：一个在 DLL 外部可见，另一个在 DLL 内部可见。使用 libpq 的事件过程函数时，务必始终使用其中同一个地址，否则会产生混淆。确保代码正常工作的最简单做法，是将事件过程声明为 `static`。如果需要在过程所在的源文件之外取得其地址，应提供一个单独的函数来返回该地址。

34.13.3. 事件支持函数

`PQregisterEventProc`

为 libpq 注册一个事件回调过程。

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                        const char *name, void *passThrough);
```

对于希望接收其事件的每个 `PGconn`，都必须注册一次事件过程。一个连接可注册的事件过程数量只受内存限制。函数成功时返回非零值，失败时返回零。

当一个 libpq 事件被触发时，*`proc`* 参数将被调用。它的内存地址也被用来查找 `instanceData`。*`name`* 参数被用来在错误消息中引用该事件过程。这个值不能是 `NULL` 或一个零长度串。名字串被复制到 `PGconn` 中，因此传递进来的东西不需要长期存在。当一个事件发生时，*`passThrough`* 指针被传递给 *`proc`*。这个参数可以是 `NULL`。

`PQsetInstanceData`

设置连接 *`conn`* 的用于过程 *`proc`* 的 `instanceData` 为 *`data`*。它在成功时返回非零值，失败时返回零（只有*`proc`* 没有被正确地注册在 *`conn`* 中，才可能会失败）。

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc, void *data);
```

`PQinstanceData`

返回连接 *`conn`* 的与过程 *`proc`* 相关的 `instanceData`，如果没有则返回`NULL`。

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

`PQresultSetInstanceData`

将结果中针对 *`proc`* 的 `instanceData` 设置为 *`data`*。成功时返回非零值，失败时返回零。（只有当 *`proc`* 未在结果中正确注册时，才可能失败。）

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc, void *data);
```

PQresultInstanceData

返回结果的与过程 *proc* 相关的 instanceData，如果没有则返回NULL。

```
void *PQresultInstanceData(const PGresult *res, PGEventProc proc);
```

34.13.4. 事件示例

下面给出一个示例框架，用于管理与 libpq 连接和结果关联的私有数据。

```
/* libpq 事件所需的头文件（注意：其中包含 libpq-fe.h） */
#include <libpq-events.h>

/* instanceData 数据 */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PGconn *conn =
        PQconnectdb("dbname=postgres options=-csearch_path=");

    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    /* 在每个需要接收事件的连接上调用一次。
     * 向 myEventProc 发送 PGEVT_REGISTER 事件。
     */
    if (!PQregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
    {
        fprintf(stderr, "Cannot register PGEventProc\n");
        PQfinish(conn);
        return 1;
    }

    /* 可以取得 conn 的 instanceData */
    data = PQinstanceData(conn, myEventProc);

    /* 向 myEventProc 发送 PGEVT_RESULTCREATE 事件 */
    res = PQexec(conn, "SELECT 1 + 1");

    /* 可以取得结果的 instanceData */
```

```
    data = PQresultInstanceData(res, myEventProc);

    /* 使用 PG_COPYRES_EVENTS 时，向 myEventProc 发送 PGEVT_RESULTCOPY 事件 */
    res_copy = PQcopyResult(res, PG_COPYRES_TUPLES | PG_COPYRES_EVENTS);

    /* 如果调用 PQcopyResult 时使用了 PG_COPYRES_EVENTS，
     * 就可以取得结果的 instanceData。
     */
    data = PQresultInstanceData(res_copy, myEventProc);

    /* 两次清除操作都会向 myEventProc 发送 PGEVT_RESULTDESTROY 事件 */
    PQclear(res);
    PQclear(res_copy);

    /* 向 myEventProc 发送 PGEVT_CONNDESTROY 事件 */
    PQfinish(conn);

    return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);

            /* 将应用程序特有的数据与连接关联 */
            PQsetInstanceData(e->conn, myEventProc, data);
            break;
        }

        case PGEVT_CONNRESET:
        {
            PGEventConnReset *e = (PGEventConnReset *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            if (data)
              memset(data, 0, sizeof(mydata));
            break;
        }

        case PGEVT_CONNDESTROY:
        {
            PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            /* 连接正在销毁，因此释放实例数据 */
            if (data)
              free_mydata(data);
            break;
        }

        case PGEVT_RESULTCREATE:
        {
            PGEventResultCreate *e = (PGEventResultCreate *)evtInfo;
            mydata *conn_data = PQinstanceData(e->conn, myEventProc);
```

```
            mydata *res_data = dup_mydata(conn_data);

            /* 将应用程序特有的数据与结果关联（从 conn 复制） */
            PQresultSetInstanceData(e->result, myEventProc, res_data);
            break;
        }

        case PGEVT_RESULTCOPY:
        {
            PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
            mydata *src_data = PQresultInstanceData(e->src, myEventProc);
            mydata *dest_data = dup_mydata(src_data);

            /* 将应用程序特有的数据与结果关联（从另一个结果复制） */
            PQresultSetInstanceData(e->dest, myEventProc, dest_data);
            break;
        }

        case PGEVT_RESULTDESTROY:
        {
            PGEventResultDestroy *e = (PGEventResultDestroy *)evtInfo;
            mydata *data = PQresultInstanceData(e->result, myEventProc);

            /* 结果正在销毁，因此释放实例数据 */
            if (data)
              free_mydata(data);
            break;
        }

        /* 未知的事件 ID，直接返回 true。 */
        default:
            break;
    }

    return true; /* 事件处理成功 */
}
```

34.14. 环境变量

以下环境变量可用于选择连接参数的默认值，供 PQconnectdb、PQsetdbLogin 和 PQsetdb 在调用代码未直接指定参数值时使用。例如，这样可以避免在简单的客户端应用程序中硬编码数据库连接信息。

- PGHOST 的行为与 host 连接参数相同。
- PGHOSTADDR 的行为与 hostaddr 连接参数相同。这可以替代或者与 PGHOST 一起设置，以避免 DNS 查找开销。
- PGPORT 的行为与 port 连接参数相同。
- PGDATABASE 的行为与 dbname 连接参数相同。
- PGUSER 的行为与 user 连接参数相同。
- PGPASSWORD 的行为与 password 连接参数相同。出于安全原因，不建议使用此环境变量，因为一些操作系统允许非 root 用户通过ps 查看进程环境变量；而应考虑使用密码文件（参见第 34.15 节）。

- PGPASSFILE 的行为与 passfile 连接参数相同。
- PGSERVICE 的行为与 service 连接参数相同。
- PGSERVICEFILE 指定每个用户的连接服务文件的名称（参见第 34.16 节）。默认为~/.pg_service.conf，或者在 Microsoft Windows 上为%APPDATA%\postgresql\.pg_service.conf。
- PGOPTIONS 的行为与 options 连接参数相同。
- PGAPPNAME 的行为与 application_name 连接参数相同。
- PGSSLMODE 的行为与 sslmode 连接参数相同。
- PGREQUIRESSL 的行为与 requiressl 连接参数相同。这个环境变量已被弃用，推荐使用 PGSSLMODE 变量；同时设置这两个变量时，前者不再生效。
- PGSSLCOMPRESSION 的行为与 sslcompression 连接参数相同。
- PGSSLCERT 的行为与 sslcert 连接参数相同。
- PGSSLKEY 的行为与 sslkey 连接参数相同。
- PGSSLROOTCERT 的行为与 sslrootcert 连接参数相同。
- PGSSLCRL 的行为与 sslcrl 连接参数相同。
- PGREQUIREPEER 的行为与 requirepeer 连接参数相同。
- PGKRBSRVNAME 的行为与 krbsrvname 连接参数相同。
- PGGSSLIB 的行为与 gsslib 连接参数相同。
- PGCONNECT_TIMEOUT 的行为与 connect_timeout 连接参数相同。
- PGCLIENTENCODING 的行为与 client_encoding 连接参数相同。
- PGTARGETSESSIONATTRS 的行为与 target_session_attrs 连接参数相同。

以下环境变量可用于指定每个 PostgreSQL 会话的默认行为。（也可参见 ALTER ROLE 和 ALTER DATABASE 命令，了解按用户或按数据库设置默认行为的方法。）

- PGDATESTYLE 设置日期/时间表示的默认风格（等同于 SET datestyle TO ...）。
- PGTZ 设置默认的时区（等同于 SET timezone TO ...）。
- PGGEQO 为遗传查询优化器设置默认模式（等同于 SET geqo TO ...）。

有关这些环境变量的正确取值，请参见 SQL 命令 SET。

下面的环境变量决定 libpq 的内部行为，它们会覆盖编译在程序中的默认值。

- PGSYSCONFDIR 设置包含 pg_service.conf 文件以及未来版本中可能出现的其他系统范围配置文件的目录。
- PGLOCALEDIR 设置包含用于消息本地化的 locale 文件的目录。

34.15. 密码文件

用户主目录中的.pgpass 文件可保存密码，供连接需要密码且尚未通过其他方式指定密码时使用。在 Microsoft Windows 上，文件名为%APPDATA%\postgresql\pgpass.conf（其中 %APPDATA% 指用户配置文件中的应用数据子目录）。也可以通过连接参数 passfile 或环境变量 PGPASSFILE 指定要使用的密码文件。

该文件中的行应采用以下格式：

hostname:*port*:*database*:*username*:*password*

（可以复制上面这一行，并在行首加上#，在文件中加入提示注释。）前四个字段中的每一个都可以是字面值，或者是*，后者可以匹配任何内容。将使用与当前连接参数匹配的第一行中的密码字段。（因此，使用通配符时，应将更具体的条目放在前面。）如果条目需要包含：或\，请使用\ 转义该字符。主机名字段首先匹配已指定的 host 连接参数；否则匹配已指定的 hostaddr 参数。如果两者都未指定，则查找主机名localhost。在以下情况下也会查找主机名 localhost：连接使用 Unix 域套接字，且host 参数与 libpq 的默认套接字目录路径匹配。在备库中，数据库字段为replication 时，匹配连接到主库的流复制连接。除此之外，数据库字段的用途有限，因为同一用户在同一数据库集簇的所有数据库中使用相同的密码。

在 Unix 系统上，密码文件的权限必须禁止所属组及其他用户的任何访问；可以用 chmod 0600 ~/.pgpass 这样的命令实现。如果权限没有这么严格，该文件将被忽略。在 Microsoft Windows 上，假定该文件存储在安全的目录中，因此不进行特殊的权限检查。

34.16. 连接服务文件

连接服务文件允许将 libpq 连接参数与单个服务名称关联。随后可以在 libpq 连接字符串中指定该服务名称，从而使用关联的设置。这样便可以修改连接参数，而无需重新编译使用 libpq 的应用程序。也可以使用 PGSERVICE 环境变量指定服务名称。

服务名称可以在每个用户的服务文件或系统范围的文件中定义。如果同一个服务名称存在于用户文件和系统文件中，则用户文件优先。默认情况下，每个用户的服务文件名为~/.pg_service.conf。在 Microsoft Windows 上，它的名称为%APPDATA%\postgresql\.pg_service.conf（其中 %APPDATA% 指用户配置文件夹中的应用数据子目录）。可以通过设置环境变量 PGSERVICEFILE 来指定不同的文件名。系统范围的文件名为 pg_service.conf。默认情况下，在PostgreSQL 安装的 etc 目录中寻找（使用 pg_config --sysconfdir 来准确识别此目录）。可以通过设置环境变量 PGSYSCONFDIR 来指定另一个目录，但不能指定不同的文件名。

两种服务文件均使用“INI 文件”格式，其中配置段名称是服务名称，参数是连接参数；参数列表见第 34.1.2 节。例如：

```
# 注释
[mydb]
host=somehost
port=5433
user=admin
```

PostgreSQL 安装目录中的 share/pg_service.conf.sample 提供了一个示例文件。

从服务文件中获取的连接参数会与其他来源的参数合并。服务文件中的设置会覆盖相应的环境变量，而连接字符串中直接给出的值又会覆盖服务文件中的设置。例如，使用上述服务文件时，连

接字符串 service=mydb port=5434 将使用主机 somehost、端口 5434、用户 admin，以及由环境变量或内置默认值设置的其他参数。

34.17. 连接参数的 LDAP 查找

如果 libpq 已经在编译时打开了 LDAP 支持（configure 的选项--with-ldap），就可以通过 LDAP 从一个中央服务器检索host 或 dbname 之类的连接参数。这样做的好处是如果一个数据库的连接参数改变，不需要在所有的客户端机器上更新连接信息。

LDAP 连接参数查询使用连接服务文件 pg_service.conf（参见第 34.16 节）。在 pg_service.conf 的配置段中，以ldap://开头的行会被识别为 LDAP URL，并执行 LDAP 查询。结果必须是一个keyword = value 键值对列表，用于设置连接选项。URL 必须符合 RFC 1959，格式如下：

```
ldap://[hostname[:port]]/search_base?attribute?search_scope?filter
```

其中，*hostname* 默认为 localhost 而 *port* 默认为 389。

LDAP 查找成功后就会停止处理 pg_service.conf；如果无法联系 LDAP 服务器，则会继续处理。这使后续指向其他 LDAP 服务器的 LDAP URL 行、常规的 keyword = value 对或默认连接选项能够作为后备。如果希望在这种情况下得到错误消息，可以在 LDAP URL 后添加一个语法不正确的行。

例如，使用以下 LDIF 文件创建的 LDAP 条目：

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:device
cn:mydatabase
description:host=dbserver.mycompany.com
description:port=5439
description:dbname=mydb
description:user=mydb_user
description:sslmode=require
```

可以通过以下 LDAP URL 查询：

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?description?one?(cn=mydatabase)
```

你也可以将常规的服务文件条目和 LDAP 查找混合。pg_service.conf 中一节的完整示例：

```
# 仅 host 和 port 存储在 LDAP 中；显式指定 dbname 和 user
[customerdb]
dbname=customer
user=appuser
ldap://ldap.acme.com/cn=dbserver,cn=hosts?pgconnectinfo?base?(objectclass=*)
```

34.18. SSL 支持

PostgreSQL 原生支持使用 SSL 连接来加密客户端与服务器之间的通信，以提高安全性。有关服务器端 SSL 功能的详细信息，请参见第 18.9 节。

libpq 读取系统范围的 OpenSSL 配置文件。默认情况下，这个文件被命名为`openssl.cnf` 并且位于 `openssl version -d` 所报告的目录中。可以通过设置环境变量 `OPENSSL_CONF` 把这个默认值覆盖为想要的配置文件的名称。

34.18.1. 客户端对服务器证书的验证

默认情况下，PostgreSQL 不会对服务器证书执行任何验证。这意味着可以在客户端不知情的情况下伪造服务器身份，例如修改 DNS 记录或接管服务器的 IP 地址。要防止身份伪造，客户端必须能够通过信任链验证服务器身份。建立信任链的方法是：在一台计算机上放置根证书机构（CA）的自签名证书，在另一台计算机上放置由根证书签发的叶证书。也可以使用由根证书签发、又用于签发叶证书的“中间”证书。

要让客户端验证服务器的身份，请在客户端放置根证书，并在服务器上放置由该根证书签发的叶证书。要让服务器验证客户端的身份，请在服务器上放置根证书，并在客户端放置由该根证书签发的叶证书。也可以使用一个或多个中间证书（通常与叶证书存储在一起），将叶证书链接到根证书。

建立信任链后，客户端可以通过两种方式验证服务器发送的叶证书。如果参数 `sslmode` 设为 `verify-ca`，libpq 会沿证书链检查到存储在客户端上的根证书，以验证服务器是否可信。如果 `sslmode` 设为 `verify-full`，libpq 还会验证服务器主机名是否与服务器证书中存储的名称匹配。如果无法验证服务器证书，SSL 连接将失败。在大多数对安全敏感的环境中，建议使用 `verify-full`。

在 `verify-full` 模式下，会将主机名与证书的主体替代名称属性匹配；如果不存在类型为 `dNSName` 的主体替代名称，则与通用名称属性匹配。如果证书的名称属性以星号（`*`）开头，该星号会被视为通配符，匹配除点（`.`）以外的所有字符。这意味着该证书不会匹配子域。如果使用 IP 地址而不是主机名建立连接，则会匹配该 IP 地址（不执行任何 DNS 查询）。

要允许服务器证书验证，必须将一个或者更多个根证书放置在用户主目录下的 `~/.postgresql/root.crt` 文件中（在 Microsoft Windows 上该文件名为 `%APPDATA%\postgresql\root.crt`）。如果需要把服务器发来的证书链链接到存储在客户端的根证书，还应该将中间证书加到该文件中。

如果文件`~/.postgresql/root.crl` 存在（微软 Windows 上的 `%APPDATA%\postgresql\root.crl`），证书撤销列表（CRL）项也会被检查。

可以通过设置连接参数 `sslrootcert` 和 `sslcrl`，或环境变量 `PGSSLROOTCERT` 和 `PGSSLCRL`，更改根证书文件与 CRL 的位置。

> **注意**
>
> 为与 PostgreSQL 的早期版本向后兼容，如果存在根 CA 文件，`sslmode=require` 的行为将与 `verify-ca` 相同，即根据 CA 验证服务器证书。不建议依赖这种行为；需要证书验证的应用程序应始终使用 `verify-ca` 或 `verify-full`。

34.18.2. 客户端证书

如果服务器请求客户端的叶证书以验证客户端身份，libpq 将发送用户主目录下 `~/.postgresql/postgresql.crt` 文件中存储的证书。这些证书必须通过证书链连接到服务器信任的根证书。还必须存在匹配的私钥文件 `~/.postgresql/postgresql.key`。在 Microsoft Windows 上，这两个文件分别名为 `%APPDATA%\postgresql\postgresql.crt` 和 `%APPDATA%\postgresql\postgresql.key`。可以通过连接参数 `sslcert` 和 `sslkey`，或环境变量 `PGSSLCERT` 和 `PGSSLKEY`，覆盖证书和密钥文件的位置。

在 Unix 系统上，私钥文件的权限必须禁止所属组及其他用户的任何访问；可以用 `chmod 0600 ~/.postgresql/postgresql.key` 这样的命令实现。另一种做法是使文件归 root 所有，并允许组用户读取（即 `0640` 权限）。这种设置适用于由操作系统管理证书和密钥文件的安装环境。此时，应将 libpq 用户加入有权访问这些证书和密钥文件的组。（在 Microsoft Windows 上，不检查文件权限，因为假定 `%APPDATA%\postgresql` 目录是安全的。）

`postgresql.crt` 中的第一个证书必须是客户端的证书，因为它必须匹配客户端的私钥。可以选择将“中间”证书追加到该文件 – 这样做避免了在服务器上存放中间证书的要求（ssl_ca_file）。

有关创建证书的说明，请参见第 18.9.5 节。

34.18.3. 不同模式中提供的保护

`sslmode` 参数的不同值提供不同级别的保护。SSL 可以防范三类攻击：

窃听

如果一个第三方能够检查客户端和服务器之间的网络流量，它能读取连接信息（包括用户名和密码）以及被传递的数据。SSL 使用加密来阻止这种攻击。

中间人（MITM）

如果第三方能修改客户端与服务器之间传输的数据，就可以冒充服务器，进而查看和修改数据，即使数据已经加密。随后，第三方可以将连接信息和数据转发给原来的服务器，使攻击无法被察觉。常见的手段包括 DNS 污染和地址劫持，从而将客户端引向预期之外的服务器。还有其他几种攻击手段可以达到同样的目的。SSL 使用证书验证，让客户端认证服务器身份，以防范这种攻击。

冒充

如果第三方能冒充获授权的客户端，就能直接访问其无权访问的数据。这通常可能由不安全的密码管理导致。SSL 使用客户端证书，确保只有持有有效证书的客户端才能访问服务器，以防范这种攻击。

要确保连接安全，必须在建立连接之前，在客户端和服务器两端配置 SSL。如果仅在服务器上配置，客户端可能在得知服务器要求高安全性之前就已发送敏感信息（例如密码）。在 libpq 中，可以将 `sslmode` 参数设为 `verify-full` 或 `verify-ca`，并向系统提供用于验证的根证书，以确保连接安全。这类似于使用 `https` URL 进行加密的网页浏览。

服务器通过身份认证后，客户端便可以传送敏感数据。这意味着，在此之前，客户端无需知道是否会使用证书进行认证，因此可以安全地仅在服务器配置中指定这一点。

所有 SSL 选项都会产生加密和密钥交换的开销，因此必须在性能与安全性之间作出权衡。表 34.1 说明了不同 `sslmode` 值所能防范的风险，以及它们所表达的对安全性和开销的取舍。

表 34.1. SSL 模式描述

sslmode	窃听保护	MITM 防护	声明
disable	否	否	我不关心安全性，并且我不想为加密增加开销。
allow	可能	否	我不关心安全性，但如果服务器坚持，我将承担加密带来的开销。
prefer	可能	否	我不关心加密，但如果服务器支持，我希望承担加密带来的开销。
require	是	否	我想要对数据加密，并且我接受因此带来的开销。我信任该网络会保证我总是连接到想要连接的服务器。
verify-ca	是	取决于 CA 策略	我想要对数据加密，并且我接受因此带来的开销。我想要确保我连接到的是我信任的服务器。
verify-full	是	是	我想要对数据加密，并且我接受因此带来的开销。我想要确保我连接到的是我信任的服务器，并且就是我指定的那一个。

verify-ca 和 verify-full 之间的区别取决于根 CA 的策略。如果使用了一个公共 CA，verify-ca 允许连接到那些可能已经被其他人注册到该 CA 的服务器。在这种情况下，总是应该使用verify-full。如果使用了一个本地 CA 或者甚至是一个自签名的证书，使用verify-ca 常常就可以提供足够的保护。

sslmode 的默认值是 prefer。如表所示，从安全角度看，这一设置没有意义；它只会在可能时带来性能开销。将其作为默认值仅出于向后兼容的考虑，不建议在有安全要求的部署中使用。

34.18.4. SSL 客户端文件使用

表 34.2 总结了与客户端 SSL 设置相关的文件。

表 34.2. libpq/客户端 SSL 文件用法

文件	内容	效果
~/.postgresql/postgresql.crt	客户端证书	发送到服务器
~/.postgresql/postgresql.key	客户端私钥	证明客户端证书是由拥有者发送；不代表证书拥有者可信
~/.postgresql/root.crt	受信任的证书机构	检查服务器证书是由一个受信任的证书机构签发
~/.postgresql/root.crl	被证书机构撤销的证书	服务器证书不能在这个列表上

34.18.5. SSL 库初始化

如果你的应用程序初始化 libssl 和/或 libcrypto 库，并且libpq 构建时带有 SSL 支持，你应该调用PQinitOpenSSL 告诉 libpq libssl 和/或 libcrypto 库已被你的应用程序初始化，以

便 libpq 不会再初始化这些库。但是，当使用OpenSSL 版本 1.1.0 或更高版本时，无需这样做，因为重复初始化不再成问题。

`PQinitOpenSSL`

允许应用程序选择要初始化的安全库。

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

当 *`do_ssl`* 是非零时，libpq 将在第一次打开数据库连接前初始化 OpenSSL 库。当 *`do_crypto`* 是非零时，`libcrypto` 库将被初始化。默认情况下（如果没有调用 `PQinitOpenSSL`），两个库都会被初始化。当 SSL 支持没有被编译时，这个函数也存在但是什么也不做。

如果你的应用使用并且初始化 OpenSSL 或者它的底层 `libcrypto` 库，你必须在第一次打开数据库连接前调用这个函数，并把相应参数设为零。同时要确保在打开一个数据库连接前已经完成了初始化。

`PQinitSSL`

允许应用程序选择要初始化的安全库。

```
void PQinitSSL(int do_ssl);
```

这个函数等效于 `PQinitOpenSSL(do_ssl, do_ssl)`。这对于要么初始化 OpenSSL 以及 `libcrypto` 要么都不初始化的应用足够用了。

`PQinitSSL` 从 PostgreSQL 8.0 就存在了，而`PQinitOpenSSL` 直到 PostgreSQL 8.4 才被加入，因此`PQinitSSL` 可能对那些需要与旧版本 libpq 一起工作的应用来说更合适。

34.19. 多线程程序中的行为

libpq 默认是可重入且线程安全的。编译应用程序代码时，可能需要使用特殊的编译器命令行选项。有关如何构建支持线程的应用程序，请参阅系统文档，或查看 `src/Makefile.global` 中的 `PTHREAD_CFLAGS` 和 `PTHREAD_LIBS`。以下函数可用于查询 libpq 的线程安全状态：

`PQisthreadsafe`

返回 libpq 库的线程安全状态。

```
int PQisthreadsafe();
```

如果 libpq 是线程安全的则返回 1，否则返回 0。

线程使用的一项限制是：两个线程不能同时操作同一个 `PGconn` 对象。尤其不能通过同一个连接对象从不同线程并发发出命令。（如果需要并发执行命令，请使用多个连接。）

`PGresult` 对象在创建后通常是只读的，因此可以在线程之间自由传递。不过，如果你使用第 34.11 节或第 34.13 节中描述的任何会修改 `PGresult` 的函数，则需要自行避免对同一个 `PGresult` 执行并发操作。

已弃用的函数 `PQrequestCancel` 和 `PQoidStatus` 不是线程安全的，不应在多线程程序中使用。`PQrequestCancel` 可以用 `PQcancel` 替代；`PQoidStatus` 可以用 `PQoidValue` 替代。

如果你在应用程序中使用 Kerberos（除了 libpq 内部之外），则需要在 Kerberos 调用周围加锁，因为 Kerberos 函数不是线程安全的。可参考libpq 源代码中的 PQregisterThreadLock 函数，它提供了一种在libpq 与应用程序之间协作加锁的方法。

34.20. 构建 libpq 程序

要构建（即编译和链接）一个使用 libpq 的程序，需要完成以下所有步骤：

- 包含 libpq-fe.h 头文件：

  ```
  #include <libpq-fe.h>
  ```

  如果没有这样做，编译器通常会给出类似以下内容的错误消息：

  ```
  foo.c: In function `main':
  foo.c:34: `PGconn' undeclared (first use in this function)
  foo.c:35: `PGresult' undeclared (first use in this function)
  foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
  ```

- 向编译器提供 -I*directory* 选项，指定 PostgreSQL 头文件的安装目录。（某些情况下，编译器默认会搜索该目录，此时可以省略这个选项。）例如，编译命令行可以是：

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  如果使用 makefile，请将该选项添加到 CPPFLAGS 变量中：

  ```
  CPPFLAGS += -I/usr/local/pgsql/include
  ```

  如果你的程序可能由其他用户编译，那么你不应该像那样硬编码目录位置。你可以运行工具 pg_config 在本地系统上找出头文件在哪里：

  ```
  $ pg_config --includedir
  /usr/local/include
  ```

  如果你安装了 pkg-config，你可以运行：

  ```
  $ pkg-config --cflags libpq
  -I/usr/local/include
  ```

  注意这将在路径前面包括 -I。

  未向编译器指定正确的选项将导致类似以下内容的错误消息：

  ```
  testlibpq.c:8:22: libpq-fe.h: No such file or directory
  ```

- 链接最终程序时，指定 `-lpq` 选项以链接 libpq 库，同时指定 `-L`*directory* 选项，告知编译器 libpq 库所在的目录。（同样，编译器默认会搜索某些目录。）为获得最大的可移植性，请将 `-L` 选项放在 `-lpq` 选项之前。例如：

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

  你也可以使用 `pg_config` 找出库目录：

  ```
  $ pg_config --libdir
  /usr/local/pgsql/lib
  ```

  或者再次使用 `pkg-config`：

  ```
  $ pkg-config --libs libpq
  -L/usr/local/pgsql/lib -lpq
  ```

  再次提示这会打印出全部的选项，而不仅仅是路径。

  指出这一部分问题的错误消息可能看起来像：

  ```
  testlibpq.o: In function `main':
  testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
  testlibpq.o(.text+0x71): undefined reference to `PQstatus'
  testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
  ```

  这意味着你忘记了 `-lpq`。

  ```
  /usr/bin/ld: cannot find -lpq
  ```

  这意味着你忘记了 `-L` 选项或者没有指定正确的目录。

34.21. 示例程序

这些示例和其他示例可在源代码发行包的 `src/test/examples` 目录中找到。

例 34.1. libpq 示例程序 1

```
/*
 * src/test/examples/testlibpq.c
 *
 *
 * testlibpq.c
 *
 *      测试 PostgreSQL 前端库 libpq 的 C 语言版本。
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
```

```
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int         nFields;
    int         i,
                j;

    /*
     * 如果用户在命令行上提供了一个参数，就将其用作 conninfo 字符串;
     * 否则默认设置 dbname=postgres，并对所有其他连接参数使用环境变量或默认值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* 连接数据库 */
    conn = PQconnectdb(conninfo);

    /* 检查是否已成功建立到后端的连接 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* 设置始终安全的搜索路径，防止恶意用户取得控制权。 */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * 不再需要 PGresult 时，应调用 PQclear 将其释放，以避免内存泄漏。
     */
    PQclear(res);

    /*
     * 这里的测试用例使用游标，因此必须处于事务块内。
     * 只需调用一次 PQexec() 执行 "select * from pg_database" 就能完成全部操作，
     * 但这太简单了，不足以作为一个好的示例。
     */

    /* 开始一个事务块 */
    res = PQexec(conn, "BEGIN");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
```

```
    PQclear(res);

    /*
     * 从数据库系统目录 pg_database 中获取行
     */
    res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in myportal");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /* 首先，打印属性名 */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* 接着，打印各行 */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* 关闭 portal……这里不再检查错误…… */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* 结束事务 */
    res = PQexec(conn, "END");
    PQclear(res);

    /* 关闭数据库连接并清理 */
    PQfinish(conn);

    return 0;
}
```

例 34.2. libpq 示例程序 2

```
/*
 * src/test/examples/testlibpq2.c
 *
 *
 * testlibpq2.c
 *      测试异步通知接口
```

```
 *
 * 启动此程序，然后在另一个窗口中通过 psql 执行
 *   NOTIFY TBL2;
 * 重复四次后，此程序便会退出。
 *
 * 或者，可以尝试更复杂的示例：
 * 使用以下命令填充数据库
 * （这些命令位于 src/test/examples/testlibpq2.sql 中）：
 *
 *   CREATE SCHEMA TESTLIBPQ2;
 *   SET search_path = TESTLIBPQ2;
 *   CREATE TABLE TBL1 (i int4);
 *   CREATE TABLE TBL2 (i int4);
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * 启动此程序，然后通过 psql 将以下命令执行四次：
 *
 *   INSERT INTO TESTLIBPQ2.TBL1 VALUES (10);
 */

#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#ifdef HAVE_SYS_SELECT_H
#include <sys/select.h>
#endif

#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int         nnotifies;

    /*
     * 如果用户在命令行上提供了一个参数，就将其用作 conninfo 字符串；
     * 否则默认设置 dbname=postgres，并对所有其他连接参数使用环境变量或默认值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";
```

```
/* 连接数据库 */
conn = PQconnectdb(conninfo);

/* 检查是否已成功建立到后端的连接 */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
    exit_nicely(conn);
}

/* 设置始终安全的搜索路径，防止恶意用户取得控制权。 */
res = PQexec(conn,
             "SELECT pg_catalog.set_config('search_path', '', false)");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * 不再需要 PGresult 时，应调用 PQclear 将其释放，以避免内存泄漏。
 */
PQclear(res);

/*
 * 发出 LISTEN 命令，以接收规则中的 NOTIFY 发出的通知。
 */
res = PQexec(conn, "LISTEN TBL2");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

/* 收到四次通知后退出。 */
nnotifies = 0;
while (nnotifies < 4)
{
    /*
     * 休眠，直到连接上有事件发生。这里使用 select(2) 等待输入，
     * 也可以使用 poll() 或类似机制。
     */
    int         sock;
    fd_set      input_mask;

    sock = PQsocket(conn);

    if (sock < 0)
        break;              /* 不应发生 */

    FD_ZERO(&input_mask);
    FD_SET(sock, &input_mask);

    if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
    {
```

```
            fprintf(stderr, "select() failed: %s\n", strerror(errno));
            exit_nicely(conn);
        }

        /* 现在检查是否有输入 */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                    "ASYNC NOTIFY of '%s' received from backend PID %d\n",
                    notify->relname, notify->be_pid);
            PQfreemem(notify);
            nnotifies++;
            PQconsumeInput(conn);
        }
    }

    fprintf(stderr, "Done.\n");

    /* 关闭数据库连接并清理 */
    PQfinish(conn);

    return 0;
}
```

例 34.3. libpq 示例程序 3

```
/*
 * src/test/examples/testlibpq3.c
 *
 *
 * testlibpq3.c
 *      测试在 SQL 命令文本之外独立传递的参数和二进制 I/O。
 *
 * 运行前，使用以下命令填充数据库
 * (这些命令位于 src/test/examples/testlibpq3.sql 中)：
 *
 * CREATE SCHEMA testlibpq3;
 * SET search_path = testlibpq3;
 * CREATE TABLE test1 (i int4, t text, b bytea);
 * INSERT INTO test1 values (1, 'joe''s place', '\\000\\001\\002\\003\\004');
 * INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\\000');
 *
 * 预期输出为:
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *  b = (5 bytes) \004\003\002\001\000
 */

#ifdef WIN32
```

```
#include <windows.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* 用于 ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*
 * 此函数打印从上面注释所定义的表中以二进制格式获取的查询结果。
 * 由于 main() 会使用它两次，因此将它单独提取成一个函数。
 */
static void
show_binary_results(PGresult *res)
{
    int         i,
                j;
    int         i_fnum,
                t_fnum,
                b_fnum;

    /* 使用 PQfnumber，以避免对结果中的字段顺序作出假设 */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char       *iptr;
        char       *tptr;
        char       *bptr;
        int         blen;
        int         ival;

        /* 获取字段值（这里忽略它们可能为 null 的情况！） */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*
         * INT4 的二进制表示采用网络字节序，最好将其转换为本机字节序。
         */
        ival = ntohl(*((uint32_t *) iptr));

        /*
         * TEXT 的二进制表示就是文本，而 libpq 会在其后追加一个零字节，
         * 因此可以直接将其用作 C 字符串。
         *
```

```
             * BYTEA 的二进制表示是一组字节，其中可能嵌有零字节，
             * 因此必须注意字段长度。
             */
            blen = PQgetlength(res, i, b_fnum);

            printf("tuple %d: got\n", i);
            printf(" i = (%d bytes) %d\n",
                   PQgetlength(res, i, i_fnum), ival);
            printf(" t = (%d bytes) '%s'\n",
                   PQgetlength(res, i, t_fnum), tptr);
            printf(" b = (%d bytes) ", blen);
            for (j = 0; j < blen; j++)
                printf("\\%03o", bptr[j]);
            printf("\n\n");
        }
    }

    int
    main(int argc, char **argv)
    {
        const char *conninfo;
        PGconn     *conn;
        PGresult   *res;
        const char *paramValues[1];
        int         paramLengths[1];
        int         paramFormats[1];
        uint32_t    binaryIntVal;

        /*
         * 如果用户在命令行上提供了一个参数，就将其用作 conninfo 字符串；
         * 否则默认设置 dbname=postgres，并对所有其他连接参数使用环境变量或默认值。
         */
        if (argc > 1)
            conninfo = argv[1];
        else
            conninfo = "dbname = postgres";

        /* 连接数据库 */
        conn = PQconnectdb(conninfo);

        /* 检查是否已成功建立到后端的连接 */
        if (PQstatus(conn) != CONNECTION_OK)
        {
            fprintf(stderr, "Connection to database failed: %s",
                    PQerrorMessage(conn));
            exit_nicely(conn);
        }

        /* 设置始终安全的搜索路径，防止恶意用户取得控制权。 */
        res = PQexec(conn, "SET search_path = testlibpq3");
        if (PQresultStatus(res) != PGRES_COMMAND_OK)
        {
            fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
            PQclear(res);
            exit_nicely(conn);
        }
        PQclear(res);

        /*
         * 此程序旨在演示如何使用 PQexecParams() 在 SQL 命令文本之外独立传递参数，
         * 以及如何以二进制方式传输数据。
         *
```

```
 * 第一个示例以文本形式传输参数，但以二进制格式接收结果。
 * 通过在 SQL 命令文本之外独立传递参数，即使数据是文本，也可以避免
 * 许多繁琐的引号和转义处理。注意，参数值中的引号无需任何特殊处理。
 */

/* 这是在 SQL 命令文本之外独立传递的参数值 */
paramValues[0] = "joe's place";

res = PQexecParams(conn,
                   "SELECT * FROM test1 WHERE t = $1",
                   1,       /* 一个参数 */
                   NULL,    /* 让后端推断参数类型 */
                   paramValues,
                   NULL,    /* 文本参数无需指定长度 */
                   NULL,    /* 默认所有参数均为文本 */
                   1);      /* 请求二进制结果 */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);

PQclear(res);

/*
 * 第二个示例以二进制形式传输一个整数参数，并同样以二进制形式获取结果。
 *
 * 虽然我们告诉 PQexecParams 让后端推断参数类型，但实际上，
 * 查询文本中的参数符号经过强制类型转换，因此其类型已经确定。
 * 发送二进制参数时，这是一项良好的安全措施。
 */

/* 将整数值 "2" 转换为网络字节序 */
binaryIntVal = htonl((uint32_t) 2);

/* 为 PQexecParams 设置参数数组 */
paramValues[0] = (char *) &binaryIntVal;
paramLengths[0] = sizeof(binaryIntVal);
paramFormats[0] = 1;        /* 二进制 */

res = PQexecParams(conn,
                   "SELECT * FROM test1 WHERE i = $1::int4",
                   1,       /* 一个参数 */
                   NULL,    /* 让后端推断参数类型 */
                   paramValues,
                   paramLengths,
                   paramFormats,
                   1);      /* 请求二进制结果 */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);
```

```
    PQclear(res);

    /* 关闭数据库连接并清理 */
    PQfinish(conn);

    return 0;
}
```

第 35 章 大对象

PostgreSQL 提供一种大对象机制，允许以流式方式访问存储在专用大对象结构中的用户数据。在处理大到无法方便地整体操作的数据值时，这种流式访问非常有用。

本章介绍 PostgreSQL 大对象数据的实现，以及相应的编程接口和查询语言接口。本章示例使用 libpq C 库，但大多数PostgreSQL 原生编程接口都支持等效的功能。其他接口也可能在内部使用大对象接口来为大值提供通用支持，这里不作说明。

35.1. 简介

所有大对象都存储在一个名为 `pg_largeobject` 的系统表中。每个大对象在系统表 `pg_largeobject_metadata` 中也有一个条目。大对象可以使用一种类似标准文件操作的读写 API 来创建、修改和删除。

PostgreSQL 还支持一种名为 “TOAST” 的存储系统，它会自动将大于单个数据库页的值存储到每个表各自的二级存储区域中。这使得大对象机制在一定程度上已经过时。大对象机制仍然保留的一个优势是，它允许的值大小可达 4 TB，而经过TOAST 处理的字段最大只能到 1 GB。此外，可以高效地读取和更新大对象的一部分，而对于经过TOAST 处理的字段，大多数操作都会把整个值作为一个整体读出或写入。

35.2. 实现特性

大对象实现会把大对象拆分成若干 “块”，并将这些块存储在数据库行中。一个 B-树索引保证在执行随机访问读写时，能够快速找到正确的块号。

为一个大对象存储的块不必是连续的。例如，如果一个应用打开一个新的大对象，定位到偏移量 1000000，然后在那里写入几个字节，并不会因此分配 1000000 字节的存储空间；只会分配覆盖实际写入数据字节范围的那些块。不过，对于最后一个现有块之前任何未分配的位置，读取操作读出的字节值都将为零。这对应于Unix 文件系统中 “稀疏分配” 文件的常见行为。

自 PostgreSQL 9.0 起，大对象具有拥有者和一组访问权限，可用GRANT 和 REVOKE 管理。读取大对象需要 `SELECT` 权限，写入或截断大对象需要`UPDATE` 权限。只有大对象的拥有者（或者数据库超级用户）才能删除、注释或更改大对象的拥有者。要为兼容先前版本而调整这种行为，请参见lo_compat_privileges 运行时参数。

35.3. 客户端接口

本节描述 PostgreSQL 的 libpq 客户端接口库为访问大对象所提供的功能。PostgreSQL 的大对象接口是仿照 Unix 文件系统接口设计的，提供了与`open`、`read`、`write`、`lseek` 等相对应的操作。

使用这些函数对大对象进行的所有操作都必须发生在一个 SQL 事务块内，因为大对象文件描述符只在事务持续期间有效。

执行其中任何一个函数时，如果发生错误，函数会返回一个正常情况下不可能出现的值，通常是 0 或 -1。描述错误的消息保存在连接对象中，可以用 `PQerrorMessage` 获取。

使用这些函数的客户端应用应包含头文件 `libpq/libpq-fs.h` 并与 libpq 库链接。

35.3.1. 创建一个大对象

函数

```
Oid lo_creat(PGconn *conn, int mode);
```

创建一个新的大对象。返回值是分配给新大对象的 OID，失败时为`InvalidOid`（零）。自 PostgreSQL 8.1 起，*`mode`* 未被使用且被忽略；不过，为了与更早版本的向后兼容，最好将其设置为`INV_READ`、`INV_WRITE` 或 `INV_READ | INV_WRITE`。（这些符号常量定义在头文件 `libpq/libpq-fs.h` 中。）

例如：

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

函数

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

也会创建一个新的大对象。要分配的 OID 可以由 *`lobjId`* 指定；如果指定了它，而该 OID 已经被某个大对象使用，则会失败。如果*`lobjId`* 是 `InvalidOid`（零），则`lo_create` 会分配一个未使用的 OID（其行为与 `lo_creat` 相同）。返回值是分配给新大对象的 OID，失败时为 `InvalidOid`（零）。

`lo_create` 是 PostgreSQL 8.1 新增的；如果将该函数用于更早版本的服务器，它会失败并返回 `InvalidOid`。

例如：

```
inv_oid = lo_create(conn, desired_oid);
```

35.3.2. 导入一个大对象

要把一个操作系统文件导入为大对象，调用

```
Oid lo_import(PGconn *conn, const char *filename);
```

`filename` 指定要作为大对象导入的操作系统文件名。返回值是分配给新大对象的 OID，失败时为`InvalidOid`（零）。注意，该文件是由客户端接口库读取的，而不是由服务器读取的；因此它必须存在于客户端文件系统中，并且对客户端应用可读。

函数

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid lobjId);
```

也会导入一个新的大对象。要分配的 OID 可以由 *`lobjId`* 指定；如果指定了它，而该 OID 已经被某个大对象使用，则会失败。如果*`lobjId`* 是 `InvalidOid`（零），则 `lo_import_with_oid` 会分配一个未使用的 OID（其行为与 `lo_import` 相同）。返回值是分配给新大对象的 OID，失败时为`InvalidOid`（零）。

lo_import_with_oid 是 PostgreSQL 8.4 新增的，它在内部使用了 8.1 新增的lo_create；如果将该函数用于 8.0 或更早版本的服务器，它会失败并返回InvalidOid。

35.3.3. 导出一个大对象

要把一个大对象导出到操作系统文件中，调用

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

lobjId 参数指定要导出的大对象的 OID，*filename* 参数指定操作系统文件名。注意，该文件是由客户端接口库写入的，而不是由服务器写入的。成功时返回 1，失败时返回 -1。

35.3.4. 打开一个现有的大对象

要打开一个现有的大对象以供读取或写入，调用

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

lobjId 参数指定要打开的大对象的 OID。*mode* 位控制该对象是以读取（INV_READ）、写入（INV_WRITE）还是两者兼有的方式打开。（这些符号常量定义在头文件 libpq/libpq-fs.h 中。）lo_open 返回一个（非负的）大对象描述符，供后续在lo_read、lo_write、lo_lseek、lo_lseek64、lo_tell、lo_tell64、lo_truncate、lo_truncate64 以及 lo_close 中使用。该描述符只在当前事务持续期间有效。失败时返回 -1。

服务器当前不区分 INV_WRITE 和 INV_READ | INV_WRITE 这两种模式：在这两种情况下都允许通过该描述符读取。不过，这两种模式与单独使用INV_READ 存在一个重要区别：使用 INV_READ 时，不能通过该描述符写入，而且从该描述符读取到的数据会反映执行lo_open 时活动事务快照中的大对象内容，而不受本事务或其他事务之后写入的影响。对于以INV_WRITE 打开的描述符，读取返回的数据会反映其他已提交事务的所有写入以及当前事务的写入。这类似于普通 SQL SELECT 命令在 REPEATABLE READ 与 READ COMMITTED 事务模式下的行为差异。

如果对该大对象没有 SELECT 权限，或者指定了INV_WRITE 但没有 UPDATE 权限，则lo_open 会失败。（在 PostgreSQL 11 之前，这些权限检查是在首次使用该描述符执行实际读或写调用时进行的。）这些权限检查可以通过lo_compat_privileges 运行时参数禁用。

例如：

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

35.3.5. 向大对象写入数据

函数

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

将 *buf* 中的 *len* 字节（缓冲区大小必须为 *len*）写入大对象描述符 *fd*。*fd* 参数必须是先前由 lo_open 返回的大对象描述符。返回值是实际写入的字节数（在当前实现中，除非出错，否则它总会等于*len*）。发生错误时，返回值为 -1。

虽然 *len* 参数被声明为 size_t，但该函数会拒绝大于INT_MAX 的长度值。实际上，最好按每块最多几兆字节来传输数据。

35.3.6. 从大对象读取数据

函数

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

从大对象描述符 *fd* 中读取最多 *len* 字节到 *buf* 中（其大小必须为 *len*）。*fd* 参数必须是先前由 lo_open 返回的大对象描述符。返回值是实际读取的字节数；如果先到达大对象末尾，该值就会小于*len*。发生错误时，返回值为 -1。

虽然 *len* 参数被声明为 size_t，但该函数会拒绝大于INT_MAX 的长度值。实际上，最好按每块最多几兆字节来传输数据。

35.3.7. 在大对象中定位

要改变与大对象描述符关联的当前读或写位置，调用

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

该函数将由 *fd* 标识的大对象描述符的当前位置指针移动到由 *offset* 指定的新位置。*whence* 的有效值是 SEEK_SET（从对象起始处定位）、SEEK_CUR（从当前位置定位）以及 SEEK_END（从对象末尾定位）。返回值是新的位置指针，出错时为 -1。

当处理大小可能超过 2 GB 的大对象时，改用

```
pg_int64 lo_lseek64(PGconn *conn, int fd, pg_int64 offset, int whence);
```

该函数的行为与 lo_lseek 相同，但它既可以接受大于 2 GB 的*offset*，也可以返回大于 2 GB 的结果。请注意，如果新位置指针会大于 2 GB，lo_lseek 将失败。

lo_lseek64 是 PostgreSQL 9.3 新增的。如果将该函数用于更早版本的服务器，它会失败并返回 -1。

35.3.8. 获取大对象的当前位置

要取得大对象描述符当前的读或写位置，调用

```
int lo_tell(PGconn *conn, int fd);
```

如果发生错误，返回值是 -1。

当处理大小可能超过 2 GB 的大对象时，改用

```
pg_int64 lo_tell64(PGconn *conn, int fd);
```

该函数的行为与 lo_tell 相同，但它可以返回大于 2 GB 的结果。请注意，如果当前读/写位置大于 2 GB，lo_tell 将失败。

lo_tell64 是 PostgreSQL 9.3 新增的。如果将该函数用于更早版本的服务器，它会失败并返回 -1。

35.3.9. 截断一个大对象

要把一个大对象截断为给定长度，调用

```
int lo_truncate(PGconn *conn, int fd, size_t len);
```

该函数把大对象描述符 *fd* 对应的大对象截断为长度 *len*。*fd* 参数必须是先前由 lo_open 返回的大对象描述符。如果 *len* 大于大对象当前的长度，则会用零字节（'\0'）把该大对象扩展到指定长度。成功时，lo_truncate 返回零；出错时返回值为 -1。

与描述符 *fd* 关联的读/写位置不会改变。

虽然 *len* 参数被声明为 size_t，但lo_truncate 会拒绝大于 INT_MAX 的长度值。

当处理大小可能超过 2 GB 的大对象时，改用

```
int lo_truncate64(PGconn *conn, int fd, pg_int64 len);
```

该函数的行为与 lo_truncate 相同，但它可以接受大于 2 GB 的*len* 值。

lo_truncate 是 PostgreSQL 8.3 新增的；如果将该函数用于更早版本的服务器，它会失败并返回 -1。

lo_truncate64 是 PostgreSQL 9.3 新增的；如果将该函数用于更早版本的服务器，它会失败并返回 -1。

35.3.10. 关闭一个大对象描述符

可以通过调用

```
int lo_close(PGconn *conn, int fd);
```

来关闭一个大对象描述符，其中*fd* 是由 lo_open 返回的大对象描述符。成功时，lo_close 返回零；出错时返回值为 -1。

任何在事务结束时仍保持打开的大对象描述符都会被自动关闭。

35.3.11. 移除一个大对象

要从数据库中移除一个大对象，调用

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

lobjId 参数指定要移除的大对象的 OID。成功时返回 1，失败时返回 -1。

35.4. 服务器端函数

专为通过 SQL 操纵大对象而设计的服务器端函数列在表 35.1 中。

表 35.1. 面向 SQL 的大对象函数

函数	返回类型	描述	示例	结果
lo_from_bytea(*loid* oid, *string* bytea)	oid	创建大对象并在其中存储数据，返回其 OID。传入 0 可让系统选择 OID。	lo_from_bytea(0, '\xfffff f00')	24528
lo_put(*loid* oid, *offset* bigint, *str* bytea)	void	在给定偏移处写入数据。	lo_put(24528, 1, '\xaa')	
lo_get(*loid* oid [, *from* bigint, *for* int])	bytea	提取内容或其中的子串。	lo_get(24528, 0, 3)	\xffaaff

前面介绍的每个客户端函数都有对应的服务器端函数；实际上，大多数客户端函数都只是等价服务器端函数的接口。通过 SQL 命令调用起来同样方便的有lo_creat、lo_create、lo_unlink、lo_import 以及 lo_export。下面是它们的使用示例：

```
CREATE TABLE image (
    name            text,
    raster          oid
);

SELECT lo_creat(-1);        -- 返回新建的空大对象的 OID

SELECT lo_create(43213);    -- 尝试创建 OID 为 43213 的大对象

SELECT lo_unlink(173454);   -- 删除 OID 为 173454 的大对象

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

INSERT INTO image (name, raster)  -- 与上例相同，但指定要使用的 OID
    VALUES ('beautiful image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

服务器端的 lo_import 和 lo_export 函数与其客户端对应函数的行为差异很大。这两个函数在服务器的文件系统中读写文件，使用运行数据库服务器的操作系统用户的权限。因此，默认情况下它们的使用仅限于超级用户。相比之下，客户端导入和导出函数在客户端的文件系统中读写文件，使用客户端程序的权限。客户端函数不需要任何数据库权限，除了读取或写入相关大对象所需的权限。

> **小心**
>
> 可以使用 GRANT 将服务器端 lo_import 和 lo_export 函数的使用权限授予非超级用户，但必须仔细考虑其安全影响。恶意用户如果拥有这类权限，很容易借此升级为超级用户（例如通过重写服务器配置文件），或者无需真正取得数据库超级用户权限就可以攻击服务器文件系统的其他部分。因此，对拥有此类权限的角色的访问必须像对超级用户

> 角色的访问一样谨慎地加以保护。尽管如此，如果某项日常任务确实需要使用服务器端 `lo_import` 或 `lo_export`，那么使用具有此类权限的角色仍然比使用拥有完整超级用户权限的角色更安全，因为这样有助于降低意外错误造成破坏的风险。

`lo_read` 和 `lo_write` 的功能也可以通过服务器端调用获得，但服务器端函数名与客户端接口不同，因为它们不包含下划线。必须将这些函数调用为`loread` 和 `lowrite`。

35.5. 示例程序

例 35.1 是一个示例程序，它展示了如何使用libpq 中的大对象接口。程序中有些部分已经被注释掉，但仍保留在源代码中供读者参考。该程序也可以在源代码发行包的`src/test/examples/testlo.c` 中找到。

例 35.1. 用 libpq 操作大对象的示例程序

```
/*-------------------------------------------------------------------------
 *
 * testlo.c
 *    测试使用 libpq 操作大对象
 *
 * Portions Copyright (c) 1996-2018, PostgreSQL Global Development Group
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 *
 * 标识
 *    src/test/examples/testlo.c
 *
 *-------------------------------------------------------------------------
 */
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE         1024

/*
 * importFile -
 *    将文件 "in_filename" 作为大对象 "lobjOid" 导入数据库
 *
 */
static Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;
```

```
    /*
     * 打开要读取的文件
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                               /* 错误 */
        fprintf(stderr, "cannot open unix file\"%s\"\n", filename);
    }

    /*
     * 创建大对象
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * 从 Unix 文件读取数据并写入大对象
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading \"%s\"", filename);
    }

    close(fd);
    lo_close(conn, lobj_fd);

    return lobjId;
}

static void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = '\0';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
        if (nbytes <= 0)
            break;              /* 没有更多数据？ */
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
```

```
}

static void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = '\0';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
        if (nbytes <= 0)
        {
            fprintf(stderr, "\nWRITE FAILED!\n");
            break;
        }
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}


/*
 * exportFile -
 *    将大对象 "lobjOid" 导出到文件 "out_filename"
 *
 */
static void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * 打开大对象
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    /*
     * 打开要写入的文件
```

```
     */
    fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0666);
    if (fd < 0)
    {                           /* 错误 */
        fprintf(stderr, "cannot open unix file\"%s\"",
                filename);
    }

    /*
     * 从大对象读取数据并写入 Unix 文件
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing \"%s\"",
                    filename);
        }
    }

    lo_close(conn, lobj_fd);
    close(fd);

    return;
}

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * 建立连接
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* 检查是否已成功建立到后端的连接 */
    if (PQstatus(conn) != CONNECTION_OK)
```

```
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* 设置始终安全的搜索路径，防止恶意用户取得控制权。 */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "begin");
    PQclear(res);
    printf("importing file \"%s\" ...\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
    if (lobjOid == 0)
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
    else
    {
        printf("\tas large object %u.\n", lobjOid);

        printf("picking out bytes 1000-2000 of the large object\n");
        pickout(conn, lobjOid, 1000, 1000);

        printf("overwriting bytes 1000-2000 of the large object with X's\n");
        overwrite(conn, lobjOid, 1000, 1000);

        printf("exporting large object to file \"%s\" ...\n", out_filename);
/*      exportFile(conn, lobjOid, out_filename); */
        if (lo_export(conn, lobjOid, out_filename) < 0)
            fprintf(stderr, "%s\n", PQerrorMessage(conn));
    }

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

第 36 章 ECPG - C 中的嵌入式 SQL

这一章描述了用于 PostgreSQL 的嵌入式 SQL 包。它由 Linus Tolke（<linus@epact.se>）和 Michael Meskes（<meskes@postgresql.org>）编写。最初它是为了与 C 一起工作而编写的。它也能与 C++配合，但是它还不识别所有的C++结构。

这份文档还很不完整。不过，由于这一接口是标准化的，因此可以在许多关于 SQL 的资料中找到补充信息。

36.1. 概念

嵌入式 SQL 程序由普通编程语言编写的代码（这里是 C）与位于特殊标记区段中的 SQL 命令混合组成。要构建该程序，首先要将源代码（*.pgc）交给嵌入式 SQL 预处理器，它会把源代码转换成普通 C 程序（*.c），之后就可以由 C 编译器处理。（关于编译和链接的细节见第 36.10 节。）转换后的 ECPG 应用会通过嵌入式 SQL 库（ecpglib）调用 libpq 库中的函数，并使用普通的前端/后端协议与 PostgreSQL 服务器通信。

与其他在 C 代码中处理 SQL 命令的方法相比，嵌入式SQL 有其优势。首先，它能处理与你的C 程序变量之间来回传递信息的繁琐工作。其次，程序中的 SQL 代码会在构建时接受语法正确性检查。第三，C 中的嵌入式SQL 由 SQL 标准规定，并且得到许多其他SQL 数据库系统的支持。PostgreSQL 的实现被设计为尽可能贴合该标准，因此通常可以比较容易地把为其他 SQL 数据库编写的嵌入式SQL 程序移植到 PostgreSQL。

如前所述，为嵌入式 SQL 接口编写的程序，就是插入了特殊代码的普通 C 程序，这些特殊代码用于执行数据库相关操作，其形式始终如下：

```
EXEC SQL ...;
```

这些语句在语法上占据一条 C 语句的位置。根据具体语句的不同，它们可以出现在全局层级，也可以出现在函数内部。嵌入式 SQL 语句遵循普通 SQL 代码的大小写规则，而不是 C 的规则。此外，它们还允许 SQL 标准中规定的嵌套 C 风格注释。但程序中的 C 部分仍遵循 C 标准，不接受嵌套注释。

下列小节解释了所有嵌入式 SQL 语句。

36.2. 管理数据库连接

这一节描述如何打开、关闭以及切换数据库连接。

36.2.1. 连接到数据库服务器

使用以下语句连接数据库：

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

其中，*target* 可以按以下方式指定：

- *dbname*[@*hostname*][:*port*]
- tcp:postgresql://*hostname*[:*port*][/*dbname*][?*options*]
- unix:postgresql://*hostname*[:*port*][/*dbname*][?*options*]
- 包含上述某种形式的 SQL 字符串常量
- 对包含上述某种形式的字符变量的引用（见示例）
- DEFAULT

如果直接写出连接目标（即不通过变量引用），且未用引号括起该值，就会应用普通 SQL 的大小写不敏感规则。此时，也可以按需分别用双引号括起各个参数。实际使用中，采用单引号括起的字符串字面值或变量引用，可能更不容易出错。连接目标 DEFAULT 会以默认用户名发起到默认数据库的连接，此时不能单独指定用户名或连接名。

也有不同的方法来指定用户名：

- *username*
- *username*/*password*
- *username* IDENTIFIED BY *password*
- *username* USING *password*

如上所述，参数*username* 以及 *password* 可以是一个 SQL 标识符、一个 SQL 字符串常量或者一个对字符变量的引用。

如果连接目标包含 *options*，那么它们由若干个*keyword*=*value* 项组成，彼此之间用和号（&）分隔。允许的关键字与 libpq 识别的关键字相同（见第 34.1.2 节）。在任何 *keyword* 或 *value* 之前的空格将被忽略，但其中或之后则不会。请注意，无法将& 写入 *value*。

connection-name 被用来在一个程序中处理多个连接。如果一个程序只使用一个连接，则可以省略它。最近被打开的连接将成为当前连接，当一个 SQL 语句要被执行时，将默认使用它（见这一章稍后的部分）。

如果不受信任的用户可以访问某个尚未采用模式的安全使用方式的数据库，那么每个会话开始时都应先从search_path 中移除公共可写模式。例如，可以把options=-c search_path=加到 *options* 中，或在连接后执行EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);。这一注意事项并非 ECPG 独有；它适用于所有能执行任意 SQL 命令的接口。

以下是一些 CONNECT 语句的示例：

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
const char *passwd = "secret";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :target USER :user USING :passwd;
/* 或者 EXEC SQL CONNECT TO :target USER :user/:passwd; */
```

最后一种形式使用了上面提到的字符变量引用。后续章节会说明，如何在 C 变量前加冒号，以便在 SQL 语句中使用它们。

请注意，连接目标的格式并未在 SQL 标准中规定。因此，如果你想开发可移植的应用，最好采用类似上面最后一个示例的方法，把连接目标字符串封装在某个地方。

36.2.2. 选择一个连接

嵌入式 SQL 程序中的 SQL 语句默认在当前连接上执行，即最近打开的连接。如果应用需要管理多个连接，可以采用两种方式。

第一种方式是显式地为每一个 SQL 语句选择一个连接，例如：

```
EXEC SQL AT connection-name SELECT ...;
```

如果应用需要以混合的顺序使用多个连接，这种方式特别合适。

如果你的应用使用多个线程执行，它们不能并发地共享一个连接。你必须显式地控制对连接的访问（使用互斥量）或者为每个线程使用一个连接。

第二种方式是执行一个语句来切换当前的连接。该语句是：

```
EXEC SQL SET CONNECTION connection-name;
```

如果有很多语句要在同一个连接上执行，这种方式尤其方便。

这里有一个管理多个数据库连接的例子程序：

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
EXEC SQL END DECLARE SECTION;

int
main()
{
    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    /* 这个查询将在最近打开的数据库 "testdb3" 中执行。 */
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb3)\n", dbname);

    /* 使用 "AT" 在 "testdb2" 中运行查询 */
    EXEC SQL AT con2 SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb2)\n", dbname);

    /* 将当前连接切换到 "testdb1"。 */
    EXEC SQL SET CONNECTION con1;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb1)\n", dbname);

    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

这个例子将产生这样的输出：

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
current=testdb1 (should be testdb1)
```

36.2.3. 关闭一个连接

要关闭连接，请使用以下语句：

```
EXEC SQL DISCONNECT [connection];
```

connection 可以通过以下方式指定：

- *connection-name*
- CURRENT
- ALL

如果未指定连接名称，则关闭当前连接。

应用程序始终显式断开其打开的每个连接，是良好的编程习惯。

36.3. 运行 SQL 命令

任何 SQL 命令都可以在一个嵌入式 SQL 应用中被运行。下面是一些在嵌入式 SQL 应用中运行 SQL 命令的例子。

36.3.1. 执行 SQL 语句

创建一个表：

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

插入行：

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

删除行：

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

更新：

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

返回一个单一结果行的 `SELECT` 语句也可以直接使用 `EXEC SQL` 执行。要处理有多行的结果集，一个应用必须使用一个游标，可参考下面的第 36.3.2 节（作为一种特殊情况，一个应用可以一次取出多行到一个数组主变量中，参考第 36.4.4.3.1 节）。

单行选择：

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

还有，一个配置参数可以用`SHOW` 命令检索：

```
EXEC SQL SHOW search_path INTO :var;
```

`:`*something* 形式的词元是主变量，即它们指向 C 程序中的变量。其说明见第 36.4 节。

36.3.2. 使用游标

要读取包含多行的结果集，应用程序必须声明一个游标，并从中逐行取出数据。使用游标的步骤如下：声明游标、打开游标、从中取出一行、重复取行，最后关闭游标。

使用游标选择：

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

有关游标声明的更多细节，参见 DECLARE；有关 `FETCH` 命令的详细信息，参见 FETCH。

> **注意**
>
> ECPG `DECLARE` 命令实际上不会向 PostgreSQL 后端发送语句。在执行 `OPEN` 命令时，游标才会在后端被打开（使用后端的`DECLARE` 命令）。

36.3.3. 管理事务

在默认模式下，只有发出`EXEC SQL COMMIT` 时命令才会提交。嵌入式 SQL 接口也支持事务自动提交（类似于 psql 的默认行为），可以通过`ecpg` 的 `-t` 命令行选项（见 ecpg）或者 `EXEC SQL SET AUTOCOMMIT TO ON` 语句启用。在自动提交模式中，除非位于显式事务块内，每条命令都会自动提交。这种模式也可以通过`EXEC SQL SET AUTOCOMMIT TO OFF` 显式关闭。

可以使用下列事务管理命令：

`EXEC SQL COMMIT`

提交一个进行中的事务。

`EXEC SQL ROLLBACK`

回滚一个进行中的事务。

`EXEC SQL PREPARE TRANSACTION` *`transaction_id`*

为两阶段提交预备当前事务。

`EXEC SQL COMMIT PREPARED` *`transaction_id`*

提交一个处于预备状态的事务。

`EXEC SQL ROLLBACK PREPARED` *`transaction_id`*

回滚一个处于预备状态的事务。

`EXEC SQL SET AUTOCOMMIT TO ON`

启用自动提交模式。

```
EXEC SQL SET AUTOCOMMIT TO OFF
```

禁用自动提交模式。这是默认值。

36.3.4. 预备语句

当要传给 SQL 语句的值在编译时未知，或者同一语句将使用多次时，预备语句就很有用。

语句使用命令 PREPARE 进行预备。对于还未知的值，使用占位符“?”：

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid = ?";
```

如果一个语句返回一个单一行，应用可以在PREPARE 之后调用 EXECUTE 来执行该语句，同时要用一个USING 子句为占位符提供真实的值：

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

如果一个语句返回多行，应用可以使用一个基于该预备语句声明的游标。要绑定输入参数，该游标必须用一个USING 子句打开：

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;

/* 到达结果集末尾时，跳出 while 循环 */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
    ...
}
EXEC SQL CLOSE foo_bar;
```

当你不再需要该预备语句时，你应该释放它：

```
EXEC SQL DEALLOCATE PREPARE name;
```

更多有关 PREPARE 的细节，可参考 PREPARE。关于使用占位符和输入参数的细节，可参考第 36.5 节。

36.4. 使用主变量

在第 36.3 节中，你已经看到如何从嵌入式 SQL 程序中执行 SQL 语句。其中一些语句只使用固定值，既无法把用户提供的值插入语句中，也无法让程序处理查询返回的值。这类语句在实际应用中并不真正有用。本节将详细解释如何使用一种称为主变量的简单机制，在 C 程序与嵌入式 SQL 语句之间传递数据。在嵌入式 SQL 程序中，我们把 SQL 语句视为 C 程序代码中的客体，而 C 是主语言。因此，C 程序中的变量被称为主变量。

另一种在 PostgreSQL 后端和 ECPG 应用之间交换值的方式是使用 SQL 描述符，它在第 36.7 节中介绍。

36.4.1. 概述

在嵌入式 SQL 中，在 C 程序和 SQL 语句之间传递数据特别简单。我们不必让程序把数据拼接进语句中去，那样会带来诸如正确为值加引号等各种复杂问题；只需在 SQL 语句中写出 C 变量的名称，并在前面加上一个冒号即可。例如：

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

该语句引用了两个名为 `v1` 和 `v2` 的 C 变量，同时还使用了一个普通的 SQL 字符串常量，以说明你并不局限于只使用某一种数据。

这种在 SQL 语句中插入 C 变量的方式可以用在 SQL 语句中每一个应该出现值表达式的地方。

36.4.2. 声明节

要把数据从程序传给数据库（例如作为查询参数），或者把数据从数据库传回程序，用来承载这些数据的 C 变量必须声明在经过特殊标记的区段中，这样嵌入式 SQL 预处理器才能识别它们。

这个区段始于：

```
EXEC SQL BEGIN DECLARE SECTION;
```

并结束于：

```
EXEC SQL END DECLARE SECTION;
```

在这两行之间，必须是正常的 C 变量声明，例如：

```
int    x = 4;
char  foo[16], bar[16];
```

如你所见，可以选择为变量指定初始值。变量的作用域由其声明节在程序中的位置决定。你也可以使用下面的语法声明变量，这种写法会隐式创建一个声明节：

```
EXEC SQL int i = 4;
```

一个程序中可以根据需要放置任意多个声明节。

这些声明也会作为普通 C 变量声明写入输出文件，因此无需再次声明它们。不准备在 SQL 命令中使用的变量可以照常在这些特殊声明节之外声明。

结构体或联合的定义也必须写在 `DECLARE` 声明节中。否则预处理器无法处理这些类型，因为它不知道它们的定义。

36.4.3. 检索查询结果

现在你应该能够把程序产生的数据传递到一个 SQL 命令中了。但是怎么检索一个查询的结果呢？为此，嵌入式 SQL 提供了常规命令`SELECT` 和 `FETCH` 的特殊变体。这些命令有一个特殊的 `INTO` 子句，它指定被检索到的值要被存储在哪些主变量中。`SELECT` 被用于只返回单一行的查询，而 `FETCH` 被用于使用一个游标返回多行的查询。

这里是一个例子：

```
/*
 * 假定有下列表：
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

那么 `INTO` 子句出现在选择列表和 `FROM` 子句之间。选择列表中的元素数量必须和 `INTO` 后面列表（也被称为目标列表）的元素数量相等。

这里有一个使用命令 `FETCH` 的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do
{
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

这里 `INTO` 子句出现在所有正常子句的后面。

36.4.4. 类型映射

当 ECPG 应用在 PostgreSQL 服务器和 C 应用之间交换值时（例如从服务器检索查询结果时或者用输入参数执行 SQL 语句时），值需要在 PostgreSQL 数据类型和主语言变量类型（具体来说是 C 语言数据类型）之间转换。ECPG 的要点之一就是它会在大多数情况下自动完成这种转换。

在这方面有两类数据类型：一些简单 PostgreSQL 数据类型（例如 `integer` 和 `text`）可以被应用直接读取和写入。其他 PostgreSQL 数据类型（例如 `timestamp` 和 `numeric`）只能通过特殊库函数访问，见第 36.4.4.2 节。

表 36.1 展示了哪种 PostgreSQL 数据类型对应于哪一种 C 数据类型。当你希望发送或接收一种给定 PostgreSQL 数据类型的值时，你应该在声明节中声明一个具有相应 C 数据类型的 C 变量。

表 36.1. PostgreSQL 数据类型与 C 变量类型的映射

PostgreSQL 数据类型	主变量类型
smallint	short
integer	int
bigint	long long int
decimal	decimal[a]
numeric	numeric[a]
real	float
double precision	double
smallserial	short
serial	int
bigserial	long long int
oid	unsigned int
character(*n*)、varchar(*n*)、text	char[*n*+1]、VARCHAR[*n*+1][b]
name	char[NAMEDATALEN]
timestamp	timestamp[a]
interval	interval[a]
date	date[a]
boolean	bool[c]
bytea	char *

[a] 这种类型只能通过特殊的库函数访问，见第 36.4.4.2 节。

[b] 在 ecpglib.h 中声明

[c] 如果不是原生类型，则在ecpglib.h 中声明

36.4.4.1. 处理字符串

要处理 SQL 字符串数据类型（例如 varchar 以及 text），有两种可能的方式来声明主变量。

一种方式是使用 char[]（一个 char 数组），这是在 C 中处理字符数据最常见的方式。

```
EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;
```

注意你必须自行管理长度。如果你把这个主变量用作一个查询的目标变量并且该查询返回超过 49 个字符的字符串，那么将会发生缓冲区溢出。

另一种方式是使用 VARCHAR 类型，它是 ECPG 提供的一种特殊类型。VARCHAR 数组的定义会为每个变量分别转换成一个具名 struct。这样一个声明：

```
VARCHAR var[180];
```

会被转变成：

```
struct varchar_var { int len; char arr[180]; } var;
```

成员 arr 容纳包含一个终止零字节的字符串。因此，要在一个VARCHAR 主变量中存储一个字符串，该主变量必须被声明为具有包括零字节终止符的长度。成员len 保存存储在 arr 中的字符串的长度，不包括终止零字节。当一个主变量被用作一个查询的输入时，如果strlen(arr) 和 len 不同，将使用短的那一个。

VARCHAR 可以被写成大写或小写形式，但是不能大小写混合。

char 和 VARCHAR 主变量也可以保存其他 SQL 类型的值，它们将被存储为字符串形式。

36.4.4.2. 访问特殊数据类型

ECPG 包含一些特殊类型帮助你容易地与来自 PostgreSQL 服务器的一些特殊数据类型交互。特别地，它已经实现了对于numeric、decimal、date、timestamp 以及 interval 类型的支持。这些数据类型无法有效地被映射到简单的主变量类型（例如 int、long long int 或者 char[]），因为它们有一种复杂的内部结构。应用通过声明特殊类型的主变量以及使用 pgtypes 库中的函数来处理这些类型。pgtypes 库（在第 36.6 节中详细描述）包含了处理这些类型的基本函数，这样你不需要仅仅为了给一个时间戳加上一个时间间隔而发送一个查询给 SQL 服务器。

下面的小节描述了这些特殊数据类型。关于 pgtypes 库函数的更多细节，请参考第 36.6 节。

36.4.4.2.1. timestamp, date

这里有一种在 ECPG 主应用中处理 timestamp 变量的模式。

首先，程序必须包括用于timestamp 类型的头文件：

```
#include <pgtypes_timestamp.h>
```

接着，在声明节中声明一个主变量为类型timestamp：

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

并且在读入一个值到该主变量中之后，使用 pgtypes 库函数处理它。在下面的例子中，timestamp 值被 PGTYPEStimestamp_to_asc() 函数转变成文本（ASCII）形式：

```
EXEC SQL SELECT now()::timestamp INTO :ts;

printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

这个例子将展示像下面形式的一些结果：

```
ts = 2010-06-27 18:03:56.949343
```

另外，DATE 类型可以用相同的方式处理。程序必须包括pgtypes_date.h，声明一个主变量为日期类型并且将一个 DATE 值使用PGTYPESdate_to_asc() 函数转变成一种文本形式。关于 pgtypes 库函数的更多细节，请参考第 36.6 节。

36.4.4.2.2. interval

对 interval 类型的处理也类似于 timestamp 和 date 类型。不过，必须显式为一个 interval 类型分配内存。换句话说，该变量的内存空间必须在堆内存中分配，而不是在栈内存中分配。

这里是一个例子程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_interval.h>

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    interval *in;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    in = PGTYPESinterval_new();
    EXEC SQL SELECT '1 min'::interval INTO :in;
    printf("interval = %s\n", PGTYPESinterval_to_asc(in));
    PGTYPESinterval_free(in);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

36.4.4.2.3. numeric, decimal

numeric 和 decimal 类型的处理类似于 interval 类型：需要定义一个指针、在堆上分配一些内存空间并且使用 pgtypes 库函数访问该变量。关于 pgtypes 库函数的更多细节，请参考第 36.6 节。

pgtypes 库没有特别为 decimal 类型提供函数。一个应用必须使用一个 pgtypes 库函数把它转变成一个 numeric 变量以便进一步处理。

这里是一个处理 numeric 和 decimal 类型变量的例子程序。

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    numeric *num;
    numeric *num2;
    decimal *dec;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    num = PGTYPESnumeric_new();
    dec = PGTYPESdecimal_new();

    EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2) INTO :num, :dec;

    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 0));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 1));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 2));

    /* 将 decimal 转换为 numeric，以显示 decimal 值。 */
    num2 = PGTYPESnumeric_new();
    PGTYPESnumeric_from_decimal(dec, num2);

    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 0));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 1));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 2));

    PGTYPESnumeric_free(num2);
    PGTYPESdecimal_free(dec);
    PGTYPESnumeric_free(num);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

36.4.4.3. 非简单类型的主变量

你也可以把数组、typedef、结构体和指针用作主变量。

36.4.4.3.1. 数组

将数组用作主变量有两种情况。第一种如第 36.4.4.1 节所述，是一种将一些文本字符串存储在 char[] 或 VARCHAR[] 中的方法。第二种是不用一个游标从一个查询结果中检索多行。如果没有一个数组，要处理由多个行组成的查询结果，我们需要使用一个游标以及FETCH 命令。但是使用数组主变量，多个行可以被一次收取。该数组的长度必须被定义成足以容纳所有的行，否则很可能会发生一次缓冲区溢出。

下面的例子扫描 pg_database 系统表并且显示所有可用数据库的 OID 和名称：

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int dbid[8];
    char dbname[8][16];
    int i;
EXEC SQL END DECLARE SECTION;

    memset(dbname, 0, sizeof(char)* 16 * 8);
    memset(dbid, 0, sizeof(int) * 8);

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    /* 一次将多行取入数组。 */
    EXEC SQL SELECT oid,datname INTO :dbid, :dbname FROM pg_database;

    for (i = 0; i < 8; i++)
        printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

这个例子显示下面的结果（确切的值取决于本地环境）。

```
oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=
oid=0, dbname=
oid=0, dbname=
```

36.4.4.3.2. 结构体

一个成员名称匹配查询结果列名的结构体可以被用来一次检索多列。该结构体使得我们能够在一个单一主变量中处理多列值。

下面的例子通过 pg_database 系统表和 pg_database_size() 函数检索可用数据库的 OID、名称和尺寸。在这个例子中，一个成员名匹配SELECT 结果的每一列的结构体变量 dbinfo_t 被用来检索结果行，而不需要把多个主变量放在FETCH 语句中。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
       long long int size;
    } dbinfo_t;

    dbinfo_t dbval;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size FROM
 pg_database;
    EXEC SQL OPEN cur1;

    /* 到达结果集末尾时，跳出 while 循环 */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* 将多列取入一个结构体。 */
        EXEC SQL FETCH FROM cur1 INTO :dbval;

        /* 打印结构体的成员。 */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, dbval.size);
    }

    EXEC SQL CLOSE cur1;
```

这个例子会显示下列结果（确切的值取决于本地环境）。

```
oid=1, datname=template1, size=4324580
oid=11510, datname=template0, size=4243460
oid=11511, datname=postgres, size=4324580
oid=313780, datname=testdb, size=8183012
```

结构体主变量“吸收”的列数与结构体的字段数相同。额外的列可以被分配给其他主变量。例如，上面的程序也可以使用结构体外部的`size`变量改写：

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
    } dbinfo_t;

    dbinfo_t dbval;
    long long int size;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size FROM
 pg_database;
    EXEC SQL OPEN cur1;

    /* 到达结果集末尾时，跳出 while 循环 */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* 将多列取入一个结构体。 */
        EXEC SQL FETCH FROM cur1 INTO :dbval, :size;

        /* 打印结构体的成员。 */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, size);
    }

    EXEC SQL CLOSE cur1;
```

36.4.4.3.3. typedef

使用`typedef`关键字将新类型映射到已有类型。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

注意，也可以使用：

```
EXEC SQL TYPE serial_t IS long;
```

此声明不必放在声明区段内。

36.4.4.3.4. 指针

你可以声明指向最常见类型的指针。但要注意，只有使用自动内存分配时，才能将指针用作查询的目标变量。关于自动内存分配的更多信息，参见第 36.7 节。

```
EXEC SQL BEGIN DECLARE SECTION;
    int   *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;
```

36.4.5. 处理非简单 SQL 数据类型

本节介绍如何在 ECPG 应用中处理非标量以及用户定义的 SQL 层数据类型。注意，这与上一节介绍的非简单类型主变量的处理方式是不同的。

36.4.5.1. 数组

ECPG 不直接支持 SQL 层的多维数组。一维 SQL 数组可以映射为 C 数组主变量，反之亦然。不过，在创建语句时，ecpg 并不知道列的类型，因此无法检查某个 C 数组是否会作为对应 SQL 数组的输入。在处理 SQL 语句输出时，ecpg 则掌握所需信息，因此会检查双方是否都是数组。

如果查询是分别访问数组的元素，那就可以避开在 ECPG 中直接使用数组。此时应使用一种能映射到该元素类型的主变量。例如，如果某列的类型是`integer` 数组，就可以使用`int` 类型的主变量；如果元素类型是 `varchar` 或 `text`，则可以使用`char[]` 或 `VARCHAR[]` 类型的主变量。

下面是一个示例。假定有下列表：

```
CREATE TABLE t3 (
    ii integer[]
);

testdb=> SELECT * FROM t3;
     ii
-------------
 {1,2,3,4,5}
(1 row)
```

下面的示例程序检索该数组的第 4 个元素，并将其存入一个类型为`int` 的主变量中：

```
EXEC SQL BEGIN DECLARE SECTION;
int ii;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii ;
    printf("ii=%d\n", ii);
}

EXEC SQL CLOSE cur1;
```

该示例会显示下列结果：

```
ii=4
```

要把多个数组元素映射到一个数组类型主变量中的多个元素，数组列的每一个元素以及主变量数组的每一个元素都必须被单独管理，例如：

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];
    ...
}
```

注意

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 错误 */
    EXEC SQL FETCH FROM cur1 INTO :ii_a;
    ...
}
```

在这种情况中不会正确工作，因为你无法把一个数组类型列直接映射到一个数组主变量。

另一种变通办法是把数组的外部字符串表示存入 `char[]` 或 `VARCHAR[]` 类型的主变量中。关于这种表示的更多细节见第 8.15.2 节。注意，这意味着在宿主程序中无法自然地把它当作数组访问，除非再做一步解析文本表示的处理。

36.4.5.2. 复合类型

ECPG 不直接支持复合类型，但有一种简单的变通办法。可用方案与上文处理数组时类似：要么分别访问每个属性，要么使用外部字符串表示。

对于下列例子，假定有下面的类型和表：

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'PostgreSQL') );
```

最显而易见的解决方案是单独访问每一个属性。下面的程序通过单独选择类型 comp_t 的每一个属性从例子表中检索数据：

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
EXEC SQL END DECLARE SECTION;

/* 将复合类型列的每个元素放入 SELECT 列表。 */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 将复合类型列的每个元素取入主变量。 */
    EXEC SQL FETCH FROM cur1 INTO :intval, :textval;

    printf("intval=%d, textval=%s\n", intval, textval.arr);
}

EXEC SQL CLOSE cur1;
```

这个示例还可以进一步改进：把在 FETCH 命令中接收值的主变量集中到一个结构体里。关于结构体形式主变量的更多细节，见第 36.4.4.3.2 节。改写成结构体形式后，两个主变量intval 和 textval 就成为 comp_t 结构体的成员，并在FETCH 命令中指定该结构体。

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int intval;
    varchar textval[33];
} comp_t;

comp_t compval;
EXEC SQL END DECLARE SECTION;

/* 将复合类型列的每个元素放入 SELECT 列表。 */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 将 SELECT 列表中的所有值放入一个结构体。 */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}

EXEC SQL CLOSE cur1;
```

尽管在 FETCH 命令中使用了结构体，SELECT 子句中的属性名仍要逐一列出。可以通过使用* 来请求该复合类型值的所有属性，从而进一步改进：

```
...
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 将 SELECT 列表中的所有值放入一个结构体。 */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}
...
```

采用这种方法后，尽管 ECPG 本身并不理解复合类型，复合类型仍然几乎可以无缝映射到结构体上。

最后，也可以把复合类型值的外部字符串表示存入char[] 或 VARCHAR[] 类型的主变量中。不过，如果采用这种方法，就很难在宿主程序中访问该值的各个字段。

36.4.5.3. 用户定义的基础类型

ECPG 并不直接支持新的用户定义的基础类型。你可以使用外部字符串表示以及类型 char[] 或 VARCHAR[] 的主变量，并且这种方案事实上对很多类型都是合适和足够的。

下面的示例使用数据类型 `complex`，它来自第 38.12 节中的示例。该类型的外部字符串表示为 `(%f,%f)`，由函数 `complex_in()` 和 `complex_out()` 定义，见第 38.12 节。以下示例将 complex 类型的值 `(1,1)` 和 `(3,3)` 分别插入列 `a` 和 `b`，然后从表中查询它们。

```
EXEC SQL BEGIN DECLARE SECTION;
    varchar a[64];
    varchar b[64];
EXEC SQL END DECLARE SECTION;

    EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex;
    EXEC SQL OPEN cur1;

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH FROM cur1 INTO :a, :b;
        printf("a=%s, b=%s\n", a.arr, b.arr);
    }

    EXEC SQL CLOSE cur1;
```

该示例的结果如下：

```
a=(1,1), b=(3,3)
```

另一种变通方案是避免在 ECPG 中直接使用用户定义的类型，而是创建一个在用户定义的类型和 ECPG 能处理的简单类型之间转换的函数或者类型转换。不过要注意，在类型系统中引入类型转换（特别是隐式类型转换）要非常小心。

例如，

```
CREATE FUNCTION create_complex(r double, i double) RETURNS complex
LANGUAGE SQL
IMMUTABLE
AS $$ SELECT $1 * complex '(1,0')' + $2 * complex '(0,1)' $$;
```

在这个定义之后，下面的语句

```
EXEC SQL BEGIN DECLARE SECTION;
double a, b, c, d;
EXEC SQL END DECLARE SECTION;

a = 1;
b = 2;
c = 3;
d = 4;

EXEC SQL INSERT INTO test_complex VALUES (create_complex(:a, :b), create_complex(:c, :d));
```

具有和

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2)', '(3,4)');
```

相同的效果。

36.4.6. 指示符

上面的示例不处理空值。实际上，如果从数据库中获取空值，检索示例将引发错误。要把空值传给数据库，或者从数据库中取回空值，就必须在每个承载数据的主变量后面再附加一个第二主变量说明。这个第二主变量称为指示符，其中保存一个标记，用来说明该数据是否为空；如果为空，就会忽略实际主变量中的值。这里是一个正确处理检索空值的示例：

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

如果值不为空，则指示符变量`val_ind` 为零；如果值为空，则为负数。（要启用特定于 Oracle 的行为，见第 36.16 节。）

指示符有另一种功能：如果指示符值为正，它表示值不为空，但是当它被存储在主变量中时已被截断。

如果向预处理器 `ecpg` 传入参数`-r no_indicator`，它就会工作在“无指示符”模式下。在这种模式中，如果没有指定指示符变量，那么对于字符串类型，空值会在输入和输出时表现为空串；对于整数类型，空值会表现为该类型可能的最小值（例如对于`int` 就是 `INT_MIN`）。

36.5. 动态 SQL

在很多情况中，一个应用必须要执行的特定 SQL 语句在编写该应用时就已知。不过在某些情况中，SQL 语句在运行时构造或者由一个外部来源提供。这样你就不能直接把 SQL 语句嵌入到 C 源代码，不过有一种功能允许你调用在一个字符串变量中提供的任意 SQL 语句。

36.5.1. 执行没有结果集的语句

执行一个任意 SQL 语句的最简单方法是使用命令 `EXECUTE IMMEDIATE`。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

`EXECUTE IMMEDIATE` 可以被用于不返回结果集的 SQL 语句（例如 DDL、`INSERT`、`UPDATE`、`DELETE`）。你不能用这种方法执行检索数据的语句（例如 `SELECT`）。下一节将描述如何执行这一种语句。

36.5.2. 执行一个有输入参数的语句

执行任意 SQL 语句的一种更强大的方法是将语句预备一次并且在每次需要时执行该预备语句。也可以预备一个一般化的语句，然后通过替换参数执行它的特定版本。在预备语句时，在你想要稍后替换参数的地方写上问号。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

当你不再需要该预备语句时，你应该释放它：

```
EXEC SQL DEALLOCATE PREPARE name;
```

36.5.3. 执行一个有结果集的语句

要执行一个只有单一结果行的 SQL 语句，可以使用EXECUTE。要保存结果，在其中增加一个 INTO 子句。

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

一个 EXECUTE 命令可以有一个 INTO 子句、一个USING 子句，可以同时有这两个子句，也可以不带这两个子句。

如果一个查询被期望返回多于一个结果行，应该如下列例子所示使用一个游标（关于游标详见第 36.3.2 节）。

```
EXEC SQL BEGIN DECLARE SECTION;
char dbaname[128];
char datname[128];
char *stmt = "SELECT u.usename as dbaname, d.datname "
             "  FROM pg_database d, pg_user u "
             "  WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;
EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

EXEC SQL PREPARE stmt1 FROM :stmt;

EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname,:datname;
    printf("dbaname=%s, datname=%s\n", dbaname, datname);
}

EXEC SQL CLOSE cursor1;

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

36.6. pgtypes 库

pgtypes 库将 PostgreSQL 数据库类型映射到 C 中等价的类型以便在 C 程序中使用。它还提供在 C 中对这些类型进行基本计算的函数，即不依赖PostgreSQL 服务器进行计算。请看下面的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
   date date1;
   timestamp ts1, tsout;
   interval iv1;
   char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
PGTYPESchar_free(out);
```

36.6.1. 字符串

`PGTYPESnumeric_to_asc` 之类的一些函数返回一个新分配的字符串的指针。这些结果应该用 `PGTYPESchar_free` 而不是 `free` 释放（这只在 Windows 上很重要，因为 Windows 上的内存分配和释放有时候需要由同一个库完成）。

36.6.2. numeric 类型

numeric 类型支持任意精度计算。PostgreSQL 服务器中的等效类型见第 8.1 节。为了支持任意精度，这类变量需要能够动态扩展和收缩。因此，numeric 变量只能在堆上创建，并通过 `PGTYPESnumeric_new` 和 `PGTYPESnumeric_free` 函数管理。decimal 类型与之相似，但精度有限，既可以在栈上创建，也可以在堆上创建。

下列函数可用于处理 numeric 类型：

`PGTYPESnumeric_new`

请求一个指向新分配的 numeric 变量的指针。

```
numeric *PGTYPESnumeric_new(void);
```

`PGTYPESnumeric_free`

释放一个 numeric 变量及其占用的全部内存。

```
void PGTYPESnumeric_free(numeric *var);
```

`PGTYPESnumeric_from_asc`

从字符串表示中解析一个 numeric 值。

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

例如，可用的格式是：`-2`、`.794`、`+3.44`、`592.49E07` 或者 `-32.84e-4`。如果值能被成功地解析，将返回一个有效的指针，否则返回 NULL 指针。目前 ECPG 总是解析整个字符串并且因此当前不支持把第一个非法字符的地址存储在`*endptr` 中。你可以安全地把 `endptr` 设置为 NULL。

`PGTYPESnumeric_to_asc`

返回由 `malloc` 分配的字符串的指针，它包含 numeric 类型`num` 的字符串表示。

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

numeric 值将以 `dscale` 位小数输出，必要时进行舍入。结果必须用 `PGTYPESchar_free()` 释放。

`PGTYPESnumeric_add`

把两个 numeric 变量相加放到第三个 numeric 变量中。

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

该函数把变量 `var1` 和 `var2` 相加放到结果变量 `result` 中。成功时该函数返回 0，出错时返回 -1。

`PGTYPESnumeric_sub`

把两个 numeric 变量相减并且把结果返回到第三个 numeric 变量。

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

该函数把变量 `var2` 从变量 `var1` 中减除。该操作的结果被存储在变量 `result` 中。成功时该函数返回 0，出错时返回 -1。

`PGTYPESnumeric_mul`

把两个 numeric 变量相乘并且把结果返回到第三个 numeric 变量。

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

该函数把变量 `var1` 和 `var2` 相乘。该操作的结果被存储在变量 `result` 中。成功时该函数返回 0，出错时返回 -1。

PGTYPESnumeric_div

把两个 numeric 变量相除并且把结果返回到第三个 numeric 变量。

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

该函数用变量 var2 除变量 var1。该操作的结果被存储在变量 result 中。成功时该函数返回 0，出错时返回 -1。

PGTYPESnumeric_cmp

比较两个 numeric 变量。

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

这个函数比较两个 numeric 变量。错误时会返回 INT_MAX。成功时，该函数返回三种可能结果之一：

- var1 大于 var2 则返回 1
- 如果 var1 小于 var2 则返回 -1
- 如果 var1 和 var2 相等则返回 0

PGTYPESnumeric_from_int

将一个 int 变量转换为 numeric 变量。

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

该函数接收一个 signed int 类型的变量，并将其值存入 numeric 变量var 中。成功时返回 0，失败时返回 -1。

PGTYPESnumeric_from_long

将一个 long int 变量转换为 numeric 变量。

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

该函数接收一个 signed long int 类型的变量，并将其值存入 numeric 变量var 中。成功时返回 0，失败时返回 -1。

PGTYPESnumeric_copy

把一个 numeric 变量复制到另一个中。

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

这个函数把 src 指向的变量的值复制到 dst 指向的变量中。成功时该函数返回 0，出错时返回 -1。

PGTYPESnumeric_from_double

将一个 double 类型的变量转换为 numeric 变量。

```
int  PGTYPESnumeric_from_double(double d, numeric *dst);
```

该函数接收一个 double 类型的变量，并将结果存入dst 指向的变量中。成功时返回 0，出错时返回 -1。

PGTYPESnumeric_to_double

将一个 numeric 类型的变量转换为 double。

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

该函数将 nv 指向的变量中的 numeric 值转换为 double，并存入dp 指向的变量中。成功时返回 0，发生错误（包括溢出）时返回 -1。发生溢出时，还会将全局变量errno 设为 PGTYPES_NUM_OVERFLOW。

PGTYPESnumeric_to_int

将一个 numeric 类型的变量转换为 int。

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

该函数将 nv 指向的变量中的 numeric 值转换为 int，并存入ip 指向的变量中。成功时返回 0，发生错误（包括溢出）时返回 -1。发生溢出时，还会将全局变量errno 设为 PGTYPES_NUM_OVERFLOW。

PGTYPESnumeric_to_long

将一个 numeric 类型的变量转换为 long。

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

该函数将 nv 所指变量中的 numeric 值转换为 long，并存入 lp 所指的变量。成功时返回 0，发生错误（包括溢出）时返回 -1。发生溢出时，还会将全局变量 errno 设为 PGTYPES_NUM_OVERFLOW。

PGTYPESnumeric_to_decimal

将一个 numeric 类型的变量转换为 decimal。

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

该函数将 src 指向的变量中的 numeric 值转换成 dst 指向的 decimal 变量。成功时该函数返回 0，出错时返回 -1，包括溢出。发生溢出时，全局变量errno 会被设置为 PGTYPES_NUM_OVERFLOW。

`PGTYPESnumeric_from_decimal`

将一个 decimal 类型的变量转换为 numeric。

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

该函数将 `src` 指向的变量中的 decimal 值转换成 `dst` 指向的 numeric 变量。成功时该函数返回 0，出错时返回 -1。由于 decimal 类型被实现为 numeric 类型的一个受限版本，这种转换不会发生溢出。

36.6.3. date 类型

C 中的 date 类型让程序能够处理 SQL date 类型的数据。PostgreSQL 服务器中的等效类型见第 8.5 节。

下列函数可用于处理 date 类型：

`PGTYPESdate_from_timestamp`

从一个时间戳中抽取日期部分。

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

该函数接收一个时间戳作为它的唯一参数并且从这个时间戳返回抽取的日期部分。

`PGTYPESdate_from_asc`

从日期的文本表示解析一个日期。

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

该函数接收一个 C char* 字符串 `str` 以及一个指向 C char* 字符串的指针 `endptr`。目前 ECPG 始终解析整个字符串，因此尚不支持将第一个无效字符的地址存入`*endptr` 中。可以放心地将 `endptr` 设为 NULL。

注意，该函数始终假定日期采用 MDY 格式，目前 ECPG 中没有变量可以改变这一点。

表 36.2 展示了所有允许的输入格式。

表 36.2. `PGTYPESdate_from_asc` 的合法输入格式

输入	结果
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999

输入	结果
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	年和一年中的第几天
J2451187	儒略日
January 8, 99 BC	公元前 99 年

PGTYPESdate_to_asc

返回一个日期变量的文本表示。

```
char *PGTYPESdate_to_asc(date dDate);
```

该函数接收日期 dDate 作为它的唯一参数。它将以形式 1999-01-18 输出该日期，即以 YYYY-MM-DD 格式输出。结果必须用 PGTYPESchar_free() 释放。

PGTYPESdate_julmdy

从一个 date 类型的变量中提取日、月和年的值。

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

该函数接收日期 d 以及一个指向有 3 个整数值的数组 mdy 的指针。变量名就表明了顺序：mdy[0] 将被设置为包含月份，mdy[1] 将被设置为日的值，而mdy[2] 将包含年。

PGTYPESdate_mdyjul

根据一个包含 3 个整数的数组创建日期值，这些整数指定日期的日、月和年。

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

该函数的第一个参数是包含 3 个整数的数组（mdy），第二个参数是一个指向 date 类型变量的指针，用来保存操作结果。

`PGTYPESdate_dayofweek`

为一个日期值返回表示它是星期几的数字。

```
int PGTYPESdate_dayofweek(date d);
```

这个函数接收日期变量 `d` 作为它唯一的参数并且返回一个整数说明这个日期是星期几。

- 0 - 星期日
- 1 - 星期一
- 2 - 星期二
- 3 - 星期三
- 4 - 星期四
- 5 - 星期五
- 6 - 星期六

`PGTYPESdate_today`

得到当前日期。

```
void PGTYPESdate_today(date *d);
```

该函数接收一个指向 date 变量的指针（`d`），并将其指向的变量设为当前日期。

`PGTYPESdate_fmt_asc`

使用格式掩码将一个 date 类型的变量转换为文本表示。

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

该函数接收要转换的日期（`dDate`）、格式掩码（`fmtstring`）以及将要保存日期的文本表示的字符串（`outbuf`）。

成功时，返回 0；如果发生错误，则返回一个负值。

下列字面量是可用的字段说明符：

- `dd` - 一个月中的第几天。
- `mm` - 一年中的第几个月。
- `yy` - 两位数的年份。
- `yyyy` - 四位数的年份。
- `ddd` - 星期几的名称（简写）。
- `mmm` - 月份的名称（简写）。

所有其他字符会被原封不动地复制到输出字符串中。

表 36.3 给出了一些可用格式，有助于理解此函数的用法。所有输出行都基于同一个日期：1959 年 11 月 23 日。

表 36.3. PGTYPESdate_fmt_asc 的合法输入格式

格式	结果
mmddyy	112359
ddmmyy	231159
yymmdd	591123
yy/mm/dd	59/11/23
yy mm dd	59 11 23
yy.mm.dd	59.11.23
.mm.yyyy.dd.	.11.1959.23.
mmm. dd, yyyy	Nov. 23, 1959
mmm dd yyyy	Nov 23 1959
yyyy dd mm	1959 23 11
ddd, mmm. dd, yyyy	Mon, Nov. 23, 1959
(ddd) mmm. dd, yyyy	(Mon) Nov. 23, 1959

PGTYPESdate_defmt_asc

使用格式掩码把一个 C char* 字符串转换成 date 类型的值。

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

该函数接收一个指向用于保存操作结果的 date 值的指针（d）、用于解析日期的格式掩码（fmt），以及包含日期文本表示的 C char* 字符串（str）。文本表示应与格式掩码匹配，但字符串与格式掩码无需一一对应。该函数只分析先后顺序，并查找表示年份位置的yy 或 yyyy、表示月份位置的mm 以及表示日位置的 dd。

表 36.4 给出了一些可用格式，有助于理解此函数的用法。

表 36.4. rdefmtdate 的合法输入格式

格式	字符串	结果
ddmmyy	21-2-54	1954-02-21
ddmmyy	2-12-54	1954-12-02
ddmmyy	20111954	1954-11-20
ddmmyy	130464	1964-04-13
mmm.dd.yyyy	MAR-12-1967	1967-03-12
yy/mm/dd	1954, February 3rd	1954-02-03
mmm.dd.yyyy	041269	1969-04-12
yy/mm/dd	In the year 2525, in the month of July, mankind will be alive on the 28th day	2525-07-28

格式	字符串	结果
dd-mm-yy	I said on the 28th of July in the year 2525	2525-07-28
mmm.dd.yyyy	9/14/58	1958-09-14
yy/mm/dd	47/03/29	1947-03-29
mmm.dd.yyyy	oct 28 1975	1975-10-28
mmddyy	Nov 14th, 1985	1985-11-14

36.6.4. timestamp 类型

C 中的 timestamp 类型让程序能够处理 SQL timestamp 类型的数据。PostgreSQL 服务器中的等效类型见第 8.5 节。

下列函数可用于处理 timestamp 类型：

`PGTYPEStimestamp_from_asc`

将时间戳的文本表示解析为 timestamp 变量。

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

该函数接收要解析的字符串（`str`）和一个指向 C char* 的指针（`endptr`）。目前 ECPG 始终解析整个字符串，因此尚不支持将第一个无效字符的地址存入`*endptr` 中。可以放心地将 `endptr` 设为 NULL。

该函数在成功时返回解析后的时间戳。出错时返回 `PGTYPESInvalidTimestamp`，并将`errno` 设置为 `PGTYPES_TS_BAD_TIMESTAMP`。关于该值的重要说明，见 `PGTYPESInvalidTimestamp`。

通常，输入字符串可以包含允许的日期规范、空白字符和允许的时间规范的任意组合。请注意，ECPG 不支持时区。它可以解析时区，但不像PostgreSQL 服务器那样应用任何计算。时区标识符会被静默丢弃。

表 36.5 包含了一些输入字符串的示例。

表 36.5. `PGTYPEStimestamp_from_asc` 的合法输入格式

输入	结果
1999-01-08 04:05:06	1999-01-08 04:05:06
January 8 04:05:06 1999 PST	1999-01-08 04:05:06
1999-Jan-08 04:05:06.789-8	1999-01-08 04:05:06.789 (time zone specifier ignored)
J2451187 04:05-08:00	1999-01-08 04:05:00 (time zone specifier ignored)

PGTYPEStimestamp_to_asc

将时间戳转换为 C char* 字符串。

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

该函数接收时间戳 tstamp 作为其唯一参数，并返回一个包含时间戳文本表示的已分配字符串。结果必须使用PGTYPESchar_free() 释放。

PGTYPEStimestamp_current

检索当前时间戳。

```
void PGTYPEStimestamp_current(timestamp *ts);
```

该函数检索当前时间戳，并将其保存到ts 指向的时间戳变量中。

PGTYPEStimestamp_fmt_asc

使用格式掩码将时间戳变量转换为 C char*。

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

该函数接收一个指向要转换的时间戳的指针作为第一个参数（ts），一个指向输出缓冲区的指针（output），已为输出缓冲区分配的最大长度（str_len），以及用于转换的格式掩码（fmtstr）。

成功时返回 0，发生错误时返回负值。

格式掩码可以使用以下格式说明符。它们与 libc 中 strftime 函数使用的格式说明符相同。不是格式说明符的内容都会被复制到输出缓冲区中。

- %A - 替换为完整星期名称的本地化表示。
- %a - 替换为缩写星期名称的本地化表示。
- %B - 替换为完整月份名称的本地化表示。
- %b - 替换为缩写月份名称的本地化表示。
- %C - 替换为年份除以 100 的结果，以十进制数表示；一位数前补零。
- %c - 替换为时间和日期的本地化表示。
- %D - 等同于 %m/%d/%y。
- %d - 替换为一个月中的日，以十进制数表示（01-31）。
- %E* %O* - POSIX 区域设置扩展。序列 %Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy 应该提供替代表示。

  此外，%OB 提供月份名称的替代形式，用于单独显示月份、不提及具体日的场合。
- %e - 替换为一个月中的日，以十进制数表示（1-31）；一位数前补空格。

- %F - 等同于 %Y-%m-%d。
- %G - 替换为包含世纪的年份，以十进制数表示。该年份是这一周中多数天所在的年份（以星期一为每周第一天）。
- %g - 替换为与 %G 相同的年份，但以不带世纪的十进制数表示（00-99）。
- %H - 替换为小时（24 小时制），以十进制数表示（00-23）。
- %h - 与 %b 相同。
- %I - 替换为小时（12 小时制），以十进制数表示（01-12）。
- %j - 替换为一年中的第几天，以十进制数表示（001-366）。
- %k - 替换为小时（24 小时制），以十进制数表示（0-23）；一位数前补空格。
- %l - 替换为小时（12 小时制），以十进制数表示（1-12）；一位数前补空格。
- %M - 替换为分钟，以十进制数表示（00-59）。
- %m - 替换为月份，以十进制数表示（01-12）。
- %n - 被换行符替换。
- %O* - 与 %E* 相同。
- %p - 按需要替换为“上午”或“下午”的本地化表示。
- %R - 等同于 %H:%M。
- %r - 等同于 %I:%M:%S %p。
- %S - 替换为秒，以十进制数表示（00-60）。
- %s - 替换为从 UTC 纪元起经过的秒数。
- %T - 等同于 %H:%M:%S
- %t - 被替换为一个制表符。
- %U - 替换为一年中的周数，以星期日为每周第一天，以十进制数表示（00-53）。
- %u - 替换为星期几，以星期一为每周第一天，以十进制数表示（1-7）。
- %V - 替换为一年中的周数，以星期一为每周第一天，以十进制数表示（01-53）。如果包含 1 月 1 日的那一周有四天或更多天属于新年，该周就是第 1 周；否则，该周是上一年的最后一周，下一周才是第 1 周。
- %v - 等同于 %e-%b-%Y。
- %W - 替换为一年中的周数，以星期一为每周第一天，以十进制数表示（00-53）。
- %w - 替换为星期几，以星期日为每周第一天，以十进制数表示（0-6）。
- %X - 被时间的本地化表示替换。

- %x - 替换为日期的本地化表示。
- %Y - 替换为包含世纪的年份，以十进制数表示。
- %y - 替换为不带世纪的年份，以十进制数表示（00-99）。
- %Z - 被时区名称替换。
- %z - 替换为时区相对于 UTC 的偏移量；前导加号表示 UTC 以东，减号表示 UTC 以西，后接小时和分钟，各占两位数字，中间没有分隔符（RFC 822 日期头的常见形式）。
- %+ - 被日期和时间的本地化表示所替换。
- %-* - GNU libc 扩展。在执行数字输出时不进行任何填充。
- $_* - GNU libc 扩展。显式指定用空格填充。
- %0* - GNU libc 扩展。显式指定用零填充。
- %% - 被% 替换。

PGTYPESTimestamp_sub

从一个时间戳减去另一个时间戳，并将结果保存在 interval 类型变量中。

```
int PGTYPESTimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

该函数从 ts1 指向的时间戳变量中减去 ts2 指向的时间戳变量，并将结果存入iv 指向的 interval 变量中。

成功时返回 0，发生错误时返回负值。

PGTYPESTimestamp_defmt_asc

使用格式掩码从文本表示中解析时间戳值。

```
int PGTYPESTimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

该函数接收变量 str 中的时间戳文本表示，以及变量fmt 中的格式掩码。结果存入 d 指向的变量中。

如果格式掩码 fmt 为 NULL，函数将退回到默认格式掩码，即%Y-%m-%d %H:%M:%S。

这是 PGTYPESTimestamp_fmt_asc 的逆函数。可用的格式掩码项见该函数的说明。

PGTYPESTimestamp_add_interval

向时间戳变量加上一个 interval 变量。

```
int PGTYPESTimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

该函数接收一个指向 timestamp 变量 tin 的指针，以及一个指向 interval 变量span 的指针。它将时间间隔加到时间戳上，并将得到的时间戳存入tout 指向的变量中。

成功时返回 0，发生错误时返回负值。

`PGTYPEStimestamp_sub_interval`

从时间戳变量中减去一个 interval 变量。

```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

该函数从 `tin` 指向的时间戳变量中减去 `span` 指向的时间间隔变量，并将结果存入`tout` 指向的变量中。

成功时返回 0，发生错误时返回负值。

36.6.5. interval 类型

C 中的 interval 类型允许程序处理 SQL interval 类型的数据。PostgreSQL 服务器中的等效类型见第 8.5 节。

下列函数可用于处理 interval 类型：

`PGTYPESinterval_new`

返回一个指向新分配 interval 变量的指针。

```
interval *PGTYPESinterval_new(void);
```

`PGTYPESinterval_free`

释放先前分配的 interval 变量所占用的内存。

```
void PGTYPESinterval_free(interval *intvl);
```

`PGTYPESinterval_from_asc`

从文本表示中解析一个 interval 值。

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

该函数解析输入字符串 `str`，并返回一个已分配 interval 变量的指针。目前 ECPG 总是解析整个字符串，因此暂不支持把第一个非法字符的地址存储在`*endptr` 中。可以安全地把 `endptr` 设为 NULL。

`PGTYPESinterval_to_asc`

将 interval 类型变量转换成文本表示。

```
char *PGTYPESinterval_to_asc(interval *span);
```

该函数把 `span` 指向的 interval 变量转换成 C `char*`。输出形式类似于：`@ 1 day 12 hours 59 mins 10 secs`。结果必须用 `PGTYPESchar_free()` 释放。

`PGTYPESinterval_copy`

复制一个 interval 类型变量。

```
int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
```

该函数把 `intvlsrc` 指向的 interval 变量复制到 `intvldest` 指向的 interval 变量中。注意，需要事先为目标变量分配好内存。

36.6.6. decimal 类型

decimal 类型和 numeric 类型相似。不过，它被限制为最大精度是 30 个有效位。与 numeric 类型只能在堆上创建相反，decimal 类型既可以在栈上也可以在堆上创建（使用函数 `PGTYPESdecimal_new` 和 `PGTYPESdecimal_free`）。在第 36.15 节中描述的 Informix 兼容模式中有很多其它函数可以处理 decimal 类型。

下列函数可用于处理 decimal 类型，并非仅在`libcompat` 库中提供。

`PGTYPESdecimal_new`

要求一个指向新分配的 decimal 变量的指针。

```
decimal *PGTYPESdecimal_new(void);
```

`PGTYPESdecimal_free`

释放一个 decimal 变量及其占用的全部内存。

```
void PGTYPESdecimal_free(decimal *var);
```

36.6.7. pgtypeslib 的 errno 值

`PGTYPES_NUM_BAD_NUMERIC`

一个参数应该包含一个 numeric 变量（或者指向一个 numeric 变量），但是实际上它的内存表示非法。

`PGTYPES_NUM_OVERFLOW`

发生一次溢出。由于 numeric 类型可以处理几乎任何精度，将一个 numeric 变量转换成其他类型可能导致溢出。

`PGTYPES_NUM_UNDERFLOW`

发生一次下溢。由于 numeric 类型可以处理几乎任何精度，将一个 numeric 变量转换成其他类型可能导致下溢。

`PGTYPES_NUM_DIVIDE_ZERO`

尝试了一次除零。

PGTYPES_DATE_BAD_DATE

一个非法的日期字符串被传给了 PGTYPESdate_from_asc 函数。

PGTYPES_DATE_ERR_EARGS

非法参数被传给了 PGTYPESdate_defmt_asc 函数。

PGTYPES_DATE_ERR_ENOSHORTDATE

PGTYPESdate_defmt_asc 函数在输入字符串中发现了一个非法词元。

PGTYPES_INTVL_BAD_INTERVAL

向 PGTYPESinterval_from_asc 函数传入了无效的 interval 字符串，或者向 PGTYPESinterval_to_asc 函数传入了无效的 interval 值。

PGTYPES_DATE_ERR_ENOTDMY

PGTYPESdate_defmt_asc 函数中的日、月、年赋值不匹配。

PGTYPES_DATE_BAD_DAY

PGTYPESdate_defmt_asc 函数发现了月中的一个非法日值。

PGTYPES_DATE_BAD_MONTH

PGTYPESdate_defmt_asc 函数发现了一个非法的月值。

PGTYPES_TS_BAD_TIMESTAMP

一个非法的时间戳字符串被传给了 PGTYPEStimestamp_from_asc 函数，或者一个非法的时间戳值被传给了PGTYPEStimestamp_to_asc 函数。

PGTYPES_TS_ERR_EINFTIME

在一个无法处理无限时间戳值的环境中遇到了这样一个值。

36.6.8. pgtypeslib 的特殊常量

PGTYPESInvalidTimestamp

一个表示无效时间戳的 timestamp 类型值。PGTYPEStimestamp_from_asc 函数在解析出错时返回此值。注意，由于timestamp 数据类型的内部表示，PGTYPESInvalid Timestamp 同时也是一个有效的时间戳，其值为1899-12-31 23:59:59。为了检测错误，应用程序每次调用PGTYPEStimestamp_from_asc 后，不仅要检查 PGTYPESInvalidTimestamp，还要检查errno != 0。

36.7. 使用描述符区域

SQL 描述符区域是一种处理 SELECT、FETCH 或 DESCRIBE 语句结果的高级方法。SQL 描述符区域将一行数据及其元数据项组合到一个数据结构中。在执行动态 SQL 语句时，结果列的性质可能无法提前获知，此时元数据特别有用。PostgreSQL 提供两种使用描述符区域的方式：命名 SQL 描述符区域和 C 结构体 SQLDA。

36.7.1. 命名 SQL 描述符区域

一个命名 SQL 描述符区域由一个头部以及一个或多个条目描述符区域构成，头部包含与整个描述符相关的信息，而条目描述符区域则描述结果行中的每一列。

在使用 SQL 描述符区域之前，需要先分配一个：

```
EXEC SQL ALLOCATE DESCRIPTOR identifier;
```

identifier 会作为该描述符区域的“变量名”。当不再需要该描述符时，应当释放它：

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier;
```

要使用一个描述符区域，把它指定为INTO 子句的存储目标（而不是列出主变量）：

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

如果结果集为空，该描述符区域仍然会包含查询的元数据，即字段的名称。

对于还没有执行的预备查询，DESCRIBE 可以被用来得到其结果集的元数据：

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

在 PostgreSQL 9.0 之前，SQL 关键词是可选的，因此使用DESCRIPTOR 和 SQL DESCRIPTOR 都会产生命名 SQL 描述符区域。现在该关键词是强制性的，省略SQL 关键词会产生 SQLDA 描述符区域（见第 36.7.2 节）。

在 DESCRIBE 和 FETCH 语句中，INTO 和 USING 关键词的用法相似：它们在描述符区域中生成结果集和元数据。

要从描述符区域取得数据，可以将它视为一个包含命名字段的结构。使用以下命令，从头部取得某个字段的值，并存入主变量：

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

目前只定义了一个头部字段：*COUNT*，它表示有多少个条目描述符区域（即结果包含多少列）。主变量必须是整数类型。要从条目描述符区域取得字段，请使用以下命令：

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

num 可以是整数字面值，也可以是包含整数的主变量。可用字段如下：

CARDINALITY（整数）

结果集中的行数

DATA

实际的数据项（因此，这个字段的数据类型取决于查询）

DATETIME_INTERVAL_CODE（整数）

当 TYPE 是 9 时，DATETIME_INTERVAL_CODE 将具有以下值之一：1 表示 DATE，2 表示 TIME，3 表示 TIMESTAMP，4 表示 TIME WITH TIME ZONE，5 表示 TIMESTAMP WITH TIME ZONE。

DATETIME_INTERVAL_PRECISION（整数）

没有实现

INDICATOR（整数）

指示符（表示一个空值或者一个值截断）

KEY_MEMBER（整数）

没有实现

LENGTH（整数）

以字符计的数据长度

NAME（字符串）

列名

NULLABLE（整数）

没有实现

OCTET_LENGTH（整数）

以字节计的数据字符表示的长度

PRECISION（整数）

精度（用于类型 numeric）

RETURNED_LENGTH（整数）

以字符计的数据长度

RETURNED_OCTET_LENGTH（整数）

以字节计的数据字符表示的长度

SCALE（整数）

小数位数（用于类型 numeric）

TYPE（整数）

列的数据类型的数字编码

在 EXECUTE、DECLARE 以及 OPEN 语句中，INTO 和 USING 关键词的效果不同。也可以手工建立一个描述符区域来为一个查询或者游标提供输入参数，并且USING SQL DESCRIPTOR *name* 是用来传递输入参数给参数化查询的方法。建立一个命名 SQL 描述符区域的语句如下：

```
EXEC SQL SET DESCRIPTOR name VALUE num field = :hostvar;
```

PostgreSQL 支持在一条 FETCH 语句中检索多条记录。在这种情况下，要将数据存入主变量，该变量必须是数组。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;

EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

36.7.2. SQLDA 描述符区域

SQLDA 描述符区域是一个 C 语言结构体，它也能被用来得到一个查询的结果集和元数据。一个结构体存储一个来自结果集的记录。

```
EXEC SQL include sqlda.h;
sqlda_t         *mysqlda;

EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

注意，这里省略了SQL 关键词。第 36.7.1 节中关于 INTO 和 USING 关键词用法的说明也适用于这里，并有一项补充规则：在DESCRIBE 语句中，如果使用了INTO 关键词，就可以完全省略 DESCRIPTOR 关键词：

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

使用 SQLDA 的程序的一般流程是：

1. 预备一个查询，并为它声明一个游标。
2. 为结果行声明一个 SQLDA。
3. 为输入参数声明一个 SQLDA，并且初始化它们（内存分配、参数设置）。
4. 用输入 SQLDA 打开一个游标。
5. 从游标中取得行，并且把它们存储到一个输出 SQLDA。
6. 从输出 SQLDA 读取值到主变量中（必要时使用转换）。
7. 关闭游标。
8. 释放为输入 SQLDA 分配的内存区域。

36.7.2.1. SQLDA 数据结构

SQLDA 使用三种数据结构类型：`sqlda_t`、`sqlvar_t` 以及 `struct sqlname`。

> **提示**
>
> PostgreSQL 的 SQLDA 数据结构与 IBM DB2 Universal Database 中的 SQLDA 相似，因此一些有关 DB2 SQLDA 的技术资料有助于更好地理解 PostgreSQL 的 SQLDA。

36.7.2.1.1. sqlda_t 结构体

结构体类型 `sqlda_t` 是实际 SQLDA 的类型。它保存一个记录。并且两个或者更多个 `sqlda_t` 结构体能够以 `desc_next` 字段中的指针连接成一个链表，这样可以表示一个有序的行集合。因此，当两个或多个行被取得时，应用可以通过沿着每一个`sqlda_t` 节点中的 `desc_next` 指针读取它们。

sqlda_t 的定义是：

```
struct sqlda_struct
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqlda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqlda_struct sqlda_t;
```

各字段的含义如下：

sqldaid

它包含一个字符串"SQLDA "。

sqldabc

它包含已分配空间的尺寸（以字节计）。

sqln

当它通过 USING 关键词传递给 OPEN、DECLARE 或 EXECUTE 语句时，它包含参数化查询的输入参数个数。如果它用作SELECT、EXECUTE 或 FETCH 语句的输出，其值与sqld 相同。

sqld

它包含一个结果集中的字段的数量。

desc_next

如果查询返回不止一个记录，会返回多个链接在一起的 SQLDA 结构体，并且desc_next 保存一个指向下一个项的指针。

sqlvar

这是结果集中列的数组。

36.7.2.1.2. sqlvar_t 结构体

结构体类型 `sqlvar_t` 保存一个列值和元数据（例如类型和长度）。该类型的定义是：

```
struct sqlvar_struct
{
    short          sqltype;
    short          sqllen;
    char          *sqldata;
    short         *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;
```

各字段的含义如下：

`sqltype`

包含该字段的类型标识符。值可以参考 `ecpgtype.h` 中的 `enum ECPGttype`。

`sqllen`

包含字段的二进制长度，例如`ECPGt_int` 是 4 字节。

`sqldata`

指向数据。数据的格式在第 36.4.4 节中描述。

`sqlind`

指向空值指示符。0 表示非空，-1 表示空。

`sqlname`

字段的名称。

36.7.2.1.3. struct sqlname 结构体

一个 `struct sqlname` 结构体保存一个列名。它用作 `sqlvar_t` 结构体的成员。该结构体的定义如下：

```
#define NAMEDATALEN 64

struct sqlname
{
        short           length;
        char            data[NAMEDATALEN];
};
```

各字段的含义如下：

`length`

包含字段名称的长度。

`data`

包含实际的字段名称。

36.7.2.2. 使用一个 SQLDA 检索一个结果集

通过一个 SQLDA 检索一个查询结果集的一般步骤是：

1. 声明一个 `sqlda_t` 结构体来接收结果集。
2. 执行 `FETCH/EXECUTE/DESCRIBE` 命令来处理查询，并在命令中指定已声明的 SQLDA。
3. 通过查看 `sqlda_t` 结构体的成员 `sqln` 来检查结果集中记录的数量。
4. 从 `sqlda_t` 结构体的成员 `sqlvar[0]`、`sqlvar[1]` 等中得到每一列的值。
5. 沿着 `sqlda_t` 结构体的成员 `desc_next` 指针到达下一行（`sqlda_t`）。
6. 根据你的需要重复上述步骤。

这里是一个通过 SQLDA 检索结果集的例子。

首先，声明一个`sqlda_t` 结构体来接收结果集。

```
sqlda_t *sqlda1;
```

接下来，在命令中指定 SQLDA。下面是一个`FETCH` 命令的例子。

```
EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

运行一个循环顺着链表来检索行。

```
sqlda_t *cur_sqlda;

for (cur_sqlda = sqlda1;
     cur_sqlda != NULL;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
}
```

在循环内部，运行另一个循环来检索行中每一列的数据（sqlvar_t 结构体）。

```
for (i = 0; i < cur_sqlda->sqld; i++)
{
    sqlvar_t v = cur_sqlda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen  = v.sqllen;
    ...
}
```

要得到一列的值，应检查sqlvar_t 结构体的成员 sqltype 的值。然后，根据列类型切换到一种合适的方法从sqlvar 字段中复制数据到一个主变量。

```
char var_buf[1024];

switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf) - 1 : sqllen));
        break;

    case ECPGt_int: /* integer */
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;

    ...
}
```

36.7.2.3. 使用一个 SQLDA 传递查询参数

使用一个 SQLDA 传递输入参数给一个预备查询的一般步骤是：

1. 创建一个预备查询（预备语句）。
2. 声明一个 sqlda_t 结构体作为输入 SQLDA。
3. 为输入 SQLDA 分配内存区域（作为 sqlda_t 结构体）。
4. 在分配好的内存中设置（复制）输入值。
5. 打开游标，并指定输入 SQLDA。

这里是一个例子。

首先，创建一个预备语句。

```
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d, pg_stat_database s WHERE d.oid = s.
datid AND (d.datname = ? OR d.oid = ?)";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :query;
```

接下来为一个 SQLDA 分配内存，并且在`sqlda_t` 结构体的 `sqln` 成员变量中设置输入参数的数量。当预备查询要求两个或多个输入参数时，应用必须分配额外的内存空间，空间的大小为 (参数数目 - 1) * sizeof(sqlvar_t)。这里的例子展示了为两个输入参数分配内存空间。

```
sqlda_t *sqlda2;

sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

sqlda2->sqln = 2; /* 输入变量的数目 */
```

内存分配之后，把参数值存储到`sqlvar[]` 数组（当 SQLDA 在接收结果集时，这也是用来检索列值的数组）。在这个例子中，输入参数是`"postgres"`（字符串类型）和 `1`（整数类型）。

```
sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqllen  = 8;

int intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *) &intval;
sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

打开游标并指定事先设置好的 SQLDA，即可将输入参数传递给预备语句。

```
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

最后，用完输入 SQLDA 后必须显式地释放已分配的内存空间，这与用于接收查询结果的 SQLDA 不同。

```
free(sqlda2);
```

36.7.2.4. 一个使用 SQLDA 的应用例子

这里是一个例子程序，它描述了如何按照输入参数的指定从系统目录中取得数据库的访问统计。

这个应用在数据库 OID 上连接两个系统表（pg_database 和 pg_stat_database），并且还取得和显示通过两个输入参数（一个数据库`postgres` 和 OID `1`）检索到的数据库统计。

首先，为输入和输出分别声明一个 SQLDA。

```
EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* 一个输出描述符 */
sqlda_t *sqlda2; /* 一个输入描述符  */
```

接下来，连接到数据库，预备一个语句，并为该预备语句声明游标。

```
int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE d.oid=s.
datid AND ( d.datname=? OR d.oid=? )";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

然后，为输入参数在输入 SQLDA 中放入一些值。为输入 SQLDA 分配内存，并且在`sqln` 中设置输入参数的数目。在 `sqlvar` 结构体的 `sqltype`、`sqldata` 和 `sqllen` 中存入类型、值和值长度。

```
    /* 为输入参数创建 SQLDA 结构。 */
    sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* 输入变量的数量 */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *)&intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

设置完输入 SQLDA 之后，用输入 SQLDA 打开一个游标。

```
    /* 用输入参数打开一个游标。 */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

从打开的游标中取得行，并存入输出 SQLDA（通常，需要在循环中反复调用FETCH，才能取出结果集中的所有行）。

```
while (1)
{
    sqlda_t *cur_sqlda;

    /* 分配描述符给游标  */
    EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

接下来，沿着sqlda_t 结构体的链表，从 SQLDA 中读取已取得的记录。

```
for (cur_sqlda = sqlda1 ;
     cur_sqlda != NULL ;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
```

读取第一个记录中的每一列。列的数量被存储在 sqld 中，第一列的实际数据被存储在 sqlvar[0] 中，两者都是sqlda_t 结构体的成员。

```
    /* 打印一行中的每一列。 */
    for (i = 0; i < sqlda1->sqld; i++)
    {
        sqlvar_t v = sqlda1->sqlvar[i];
        char *sqldata = v.sqldata;
        short sqllen  = v.sqllen;

        strncpy(name_buf, v.sqlname.data, v.sqlname.length);
        name_buf[v.sqlname.length] = '\0';
```

现在，列数据已存储在变量v 中。将每个数据项复制到主变量，并查看v.sqltype 来确定列的类型。

```
            switch (v.sqltype) {
                int intval;
                double doubleval;
                unsigned long long int longlongval;

                case ECPGt_char:
                    memset(&var_buf, 0, sizeof(var_buf));
                    memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf)-1 :
 sqllen));
                    break;

                case ECPGt_int: /* 整数 */
                    memcpy(&intval, sqldata, sqllen);
                    snprintf(var_buf, sizeof(var_buf), "%d", intval);
                    break;

                ...

                default:
                    ...
            }

            printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
        }
```

处理所有记录后关闭游标，并且从数据库断开连接。

```
    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;
```

整个程序显示在例 36.1 中。

例 36.1. 示例 SQLDA 程序

```
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* 用于输出的描述符 */
sqlda_t *sqlda2; /* 用于输入的描述符 */

EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE d.oid=s.
datid AND ( d.datname=? OR d.oid=? )";

    int intval;
    unsigned long long int longlongval;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO uptimedb AS con1 USER uptime;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
```

```
    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;

    /* 为一个输入参数创建一个 SQLDA 结构 */
    sqlda2 = (sqlda_t *)malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* 输入变量的数量 */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *) &intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);

    /* 用输入参数打开一个游标。 */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;

    while (1)
    {
        sqlda_t *cur_sqlda;

        /* 给游标分配描述符  */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;

        for (cur_sqlda = sqlda1 ;
             cur_sqlda != NULL ;
             cur_sqlda = cur_sqlda->desc_next)
        {
            int i;
            char name_buf[1024];
            char var_buf[1024];

            /* 打印一行中的每一列。 */
            for (i=0 ; i<cur_sqlda->sqld ; i++)
            {
                sqlvar_t v = cur_sqlda->sqlvar[i];
                char *sqldata = v.sqldata;
                short sqllen  = v.sqllen;

                strncpy(name_buf, v.sqlname.data, v.sqlname.length);
                name_buf[v.sqlname.length] = '\0';

                switch (v.sqltype)
                {
                    case ECPGt_char:
                        memset(&var_buf, 0, sizeof(var_buf));
                        memcpy(&var_buf, sqldata, (sizeof(var_buf)<=sqllen ? sizeof(var_buf)
-1 : sqllen) );
                        break;

                    case ECPGt_int: /* 整数 */
                        memcpy(&intval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%d", intval);
                        break;

                    case ECPGt_long_long: /* 大整数 */
                        memcpy(&longlongval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%lld", longlongval);
                        break;

                    default:
                    {
                        int i;
                        memset(var_buf, 0, sizeof(var_buf));
                        for (i = 0; i < sqllen; i++)
                        {
                            char tmpbuf[16];
                            snprintf(tmpbuf, sizeof(tmpbuf), "%02x ", (unsigned char) sqldata[
i]);
```

```
                            strncat(var_buf, tmpbuf, sizeof(var_buf));
                        }
                    }
                        break;
                }

                printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
            }

            printf("\n");
        }
    }

    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

这个例子的输出应该看起来类似下面的结果（一些数字会变化）。

```
oid = 1 (type: 1)
datname = template1 (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = t (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl = {=c/uptime,uptime=CTc/uptime} (type: 1)
datid = 1 (type: 1)
datname = template1 (type: 1)
numbackends = 0 (type: 5)
xact_commit = 113606 (type: 9)
xact_rollback = 0 (type: 9)
blks_read = 130 (type: 9)
blks_hit = 7341714 (type: 9)
tup_returned = 38262679 (type: 9)
tup_fetched = 1836281 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

oid = 11511 (type: 1)
datname = postgres (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = f (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
```

```
datconfig =  (type: 1)
datacl =  (type: 1)
datid = 11511 (type: 1)
datname = postgres (type: 1)
numbackends = 0 (type: 5)
xact_commit = 221069 (type: 9)
xact_rollback = 18 (type: 9)
blks_read = 1176 (type: 9)
blks_hit = 13943750 (type: 9)
tup_returned = 77410091 (type: 9)
tup_fetched = 3253694 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)
```

36.8. 错误处理

这一节描述如何在嵌入式 SQL 程序中处理异常情况和警告。为此提供了两种可以同时使用的机制。

- 可以使用 `WHENEVER` 命令配置回调来处理警告和错误情况。
- 可以从 `sqlca` 变量中获得错误或警告的详细信息。

36.8.1. 设置回调

捕获错误和警告的一种简单方法是指定一个动作，在特定条件发生时执行。一般形式如下：

```
EXEC SQL WHENEVER condition action;
```

condition 可以是以下值之一：

`SQLERROR`

只要在 SQL 语句执行期间发生一个错误就调用指定的动作。

`SQLWARNING`

只要在 SQL 语句执行期间发生一个警告就调用指定的动作。

`NOT FOUND`

只要一个 SQL 语句检索或者影响零行就调用指定的动作（这种情况不是一个错误，但是你可能需要特别地处理它）。

action 可以是以下值之一：

`CONTINUE`

这实际上表示该情况被忽略。这是默认值。

`GOTO` *label*
`GO TO` *label*

跳到指定的标签（使用一个 C `goto` 语句）。

`SQLPRINT`

把一个消息打印到标准错误。这对简单程序或原型开发很有用。消息的细节无法配置。

`STOP`

调用 `exit(1)` 终止程序。

`DO BREAK`

执行 C 语句 `break`。只应被用在循环或 `switch` 语句中。

`DO CONTINUE`

执行 C 语句 `continue`。它只能用于循环语句，执行后会使控制流返回循环的开头。

`CALL` *`name`* `(`*`args`*`)`
`DO` *`name`* `(`*`args`*`)`

使用指定参数调用指定的 C 函数（这种用法不同于常规 PostgreSQL 语法中 `CALL` 和 `DO` 的含义）。

SQL 标准只规定了 `CONTINUE` 和 `GOTO`（以及 `GO TO`）动作。

这里有一个可能会用在简单程序中的例子。当一个警告发生时它打印一个简单消息，而发生一个错误时它会中止程序：

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

语句 EXEC SQL WHENEVER 是 SQL 预处理器的指令，而不是 C 语句。它设置的错误或警告动作适用于源代码中出现在处理器设置位置之后的所有嵌入式 SQL 语句，除非在第一个 EXEC SQL WHENEVER 与触发该条件的 SQL 语句之间，为同一条件设置了其他动作。这与 C 程序的控制流无关。因此，下面两个 C 程序片段都不会产生预期效果：

```
/*
 * 错误
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}


/*
 * 错误
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

36.8.2. sqlca

为了支持更强大的错误处理，嵌入式 SQL 接口提供了一个名为sqlca（SQL 通信区域）的全局变量，其结构如下：

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

（在一个多线程程序中，每一个线程会自动得到它自己的sqlca 副本。这和对于标准 C 全局变量 errno 的处理相似。）

sqlca 覆盖了警告和错误。如果执行一个语句时发生了多个警告或错误，那么sqlca 将只包含关于最后一个的信息。

如果在上一个 SQL 语句中没有产生错误，sqlca.sqlcode 将为 0 并且 sqlca.sqlstate 将为"00000"。如果发生一个警告或错误，则sqlca.sqlcode 将为负并且 sqlca.sqlstate 将不为"00000"。一个正的 sqlca.sqlcode 表示一种无害的情况，例如上一个查询返回零行。sqlcode 和 sqlstate 是两种不同的错误编码方案，详见下文。

如果上一个 SQL 语句成功，则sqlca.sqlerrd[1] 包含所处理行的 OID（如果适用），sqlca.sqlerrd[2] 包含处理或返回的行数（如果适用于该命令）。

发生错误或警告时，sqlca.sqlerrm.sqlerrmc 包含描述该错误的字符串。字段 sqlca.sqlerrm.sqlerrml 包含存储在 sqlca.sqlerrm.sqlerrmc 中的错误消息的长度（即 strlen() 的结果，对于 C 程序员通常没有多少用处）。注意，有些消息太长，无法存入定长的sqlerrmc 数组，因此会被截断。

发生警告时，sqlca.sqlwarn[2] 被设为 W（其他情况下，它被设为非W 的值）。如果 sqlca.sqlwarn[1] 被设为 W，说明某个值在存入主变量时被截断了。如果其他任一元素被设置为指示警告，sqlca.sqlwarn[0] 就会被设为 W。

字段 sqlcaid、sqlabc、sqlerrp，以及sqlerrd 和 sqlwarn 的其余元素，目前不包含有用信息。

SQL 标准中没有定义 sqlca 结构体，但是在一些其他的 SQL 数据库系统中都有实现。在核心上这些定义都相似，但是如果你想要编写可移植的应用，那么你应该仔细研究不同的实现。

这里有一个整合使用 WHENEVER 和 sqlca 的例子，当一个错误发生时打印出sqlca 的内容。在安装一个更“用户友好”的错误处理器之前，这可能对调试或开发原型应用有用。

```
EXEC SQL WHENEVER SQLERROR CALL print_sqlca();

void
print_sqlca()
{
    fprintf(stderr, "==== sqlca ====\n");
    fprintf(stderr, "sqlcode: %ld\n", sqlca.sqlcode);
    fprintf(stderr, "sqlerrm.sqlerrml: %d\n", sqlca.sqlerrm.sqlerrml);
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\n", sqlca.sqlerrm.sqlerrmc);
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\n", sqlca.sqlerrd[0],sqlca.sqlerrd[1],
sqlca.sqlerrd[2],
                                                          sqlca.sqlerrd[3],sqlca.sqlerrd[4],
sqlca.sqlerrd[5]);
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\n", sqlca.sqlwarn[0], sqlca.sqlwarn[1],
 sqlca.sqlwarn[2],
                                                          sqlca.sqlwarn[3], sqlca.sqlwarn[4],
 sqlca.sqlwarn[5],
                                                          sqlca.sqlwarn[6], sqlca.sqlwarn[7]);
    fprintf(stderr, "sqlstate: %5s\n", sqlca.sqlstate);
    fprintf(stderr, "===============\n");
}
```

结果看起来像（这里的错误是一个拼写错误的表名）：

```
==== sqlca ====
sqlcode: -400
sqlerrm.sqlerrml: 49
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38
sqlerrd: 0 0 0 0 0 0
sqlwarn: 0 0 0 0 0 0 0 0
sqlstate: 42P01
===============
```

36.8.3. SQLSTATE 与 SQLCODE

字段 sqlca.sqlstate 和 sqlca.sqlcode 采用两种不同的错误编码方案。两者都源自 SQL 标准，但SQLCODE 在 SQL-92 标准中已被标记为弃用，并在后续版本中被删除。因此，强烈建议新应用使用SQLSTATE。

SQLSTATE 是一个包含五个字符的数组。这五个字符由数字或大写字母组成，表示各种错误和警告条件的代码。SQLSTATE 采用分层编码方案：前两个字符表示条件所属的大类，后三个字符表示该大类中的子类。代码00000 表示成功状态。大部分 SQLSTATE 代码由 SQL 标准定义。PostgreSQL 服务器原生支持 SQLSTATE 错误码，因此在所有应用中统一使用这种错误编码方案，可以实现较高的一致性。更多信息见附录 A。

已弃用的错误编码方案 SQLCODE 使用一个简单的整数。0 表示成功，正值表示成功并有附加信息，负值表示错误。SQL 标准只定义了正值 +100，表示上一个命令返回或影响了零行，没有定义具体的负值。因此，这种方案的可移植性较差，也不采用分层编码。历史上，PostgreSQL 的嵌入式 SQL 处理器为自身使用分配了一些特定的 SQLCODE 值，下文列出了它们的数值和符号名称。注意，这些值不能移植到其他 SQL 实现。为了便于将应用迁移到SQLSTATE 方案，还列出了对应的SQLSTATE。不过，两种方案之间不存在一对一或一对多的映射关系（实际上是多对多），因此每次都应查阅附录 A 中的完整 SQLSTATE 列表。

以下是已分配的 SQLCODE 值：

0 (ECPG_NO_ERROR)

表示没有错误（SQLSTATE 00000）。

100 (ECPG_NOT_FOUND)

这是一种无害情况，它表示上一个命令检索或者处理了零行，或者你到达了游标的末尾（SQLSTATE 02000）。

在一个循环中处理一个游标时，你可以使用这个代码作为一种方法来检测何时中止该循环，像这样：

```
while (1)
{
    EXEC SQL FETCH ... ;
    if (sqlca.sqlcode == ECPG_NOT_FOUND)
        break;
}
```

但是 WHENEVER NOT FOUND DO BREAK 实际上会在内部这样做，因此显式地把它写出来通常没有什么好处。

-12 (ECPG_OUT_OF_MEMORY)

表示你的虚拟内存已被耗尽。数字值被定义为-ENOMEM（SQLSTATE YE001）。

-200 (ECPG_UNSUPPORTED)

表示预处理器生成了该库无法识别的内容。可能是所用的预处理器与库的版本不兼容（SQLSTATE YE002）。

-201 (ECPG_TOO_MANY_ARGUMENTS)

这表示命令指定了超过该命令预期数量的主变量（SQLSTATE 07001 或 07002）。

-202 (ECPG_TOO_FEW_ARGUMENTS)

这表示命令指定的主变量数量低于该命令的预期（SQLSTATE 07001 或 07002）。

-203 (ECPG_TOO_MANY_MATCHES)

这意味着一个查询已经返回了多个行，但是该语句只准备存储一个结果行（例如，因为指定的变量不是数组）（SQLSTATE 21000）。

-204 (ECPG_INT_FORMAT)

主变量是类型 int 而数据库中的数据是一种不同的类型并且含有一个不能被解释为 int 的值。该库使用 strtol() 进行这种转换（SQLSTATE 42804）。

-205 (ECPG_UINT_FORMAT)

主变量是类型 unsigned int 而数据库中的数据是一种不同的类型并且含有一个不能被解释为 unsigned int 的值。该库使用 strtoul() 进行这种转换（SQLSTATE 42804）。

-206 (ECPG_FLOAT_FORMAT)

主变量是类型 float 而数据库中的数据是另一种类型并且含有一个不能被解释为 float 的值。该库使用 strtod() 进行这种转换（SQLSTATE 42804）。

-207 (ECPG_NUMERIC_FORMAT)

主变量是类型 numeric 而数据库中的数据是另一种类型并且含有一个不能被解释为 numeric 的值（SQLSTATE 42804）。

-208 (ECPG_INTERVAL_FORMAT)

主变量是类型 interval 而数据库中的数据是另一种类型并且含有一个不能被解释为 interval 的值（SQLSTATE 42804）。

-209 (ECPG_DATE_FORMAT)

主变量是类型 date 而数据库中的数据是另一种类型并且含有一个不能被解释为 date 的值（SQLSTATE 42804）。

-210 (ECPG_TIMESTAMP_FORMAT)

主变量是类型 timestamp 而数据库中的数据是另一种类型并且含有一个不能被解释为 timestamp 的值（SQLSTATE 42804）。

-211 (ECPG_CONVERT_BOOL)

这表示主变量是类型 bool 而数据库中的数据既不是't' 也不是'f'（SQLSTATE 42804）。

-212 (ECPG_EMPTY)

发送给 PostgreSQL 服务器的语句是空的（通常在一个嵌入式 SQL 程序中不会发生，因此它可能指向一个内部错误）（SQLSTATE YE002）。

-213 (ECPG_MISSING_INDICATOR)

返回了一个空值，但未提供空值指示符变量（SQLSTATE 22002）。

-214 (ECPG_NO_ARRAY)

在要求一个数组的地方使用了一个普通变量（SQLSTATE 42804）。

-215 (ECPG_DATA_NOT_ARRAY)

在一个要求数组值的地方数据库返回了一个普通变量（SQLSTATE 42804）。

-216 (ECPG_ARRAY_INSERT)

该值不能被插入到数组（SQLSTATE 42804）。

-220 (ECPG_NO_CONN)

程序尝试访问一个不存在的连接（SQLSTATE 08003）。

-221 (ECPG_NOT_CONN)

程序尝试访问一个存在的连接但是它没有打开（这是一个内部错误）（SQLSTATE YE002）。

-230 (ECPG_INVALID_STMT)

尝试使用的语句尚未预备（SQLSTATE 26000）。

-239 (ECPG_INFORMIX_DUPLICATE_KEY)

重复键错误，违背唯一约束（Informix 兼容模式）（SQLSTATE 23505）。

-240 (ECPG_UNKNOWN_DESCRIPTOR)

未找到指定的描述符。尝试使用的语句尚未预备（SQLSTATE 33000）。

-241 (ECPG_INVALID_DESCRIPTOR_INDEX)

指定的描述符索引超出范围（SQLSTATE 07009）。

-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM)

请求了一个无效的描述符条目（这是一个内部错误）（SQLSTATE YE002）。

-243 (ECPG_VAR_NOT_NUMERIC)

执行动态语句时，数据库返回了数值，而主变量不是数值类型（SQLSTATE 07006）。

-244 (ECPG_VAR_NOT_CHAR)

执行动态语句时，数据库返回了非数值的值，而主变量是数值类型（SQLSTATE 07006）。

-284 (ECPG_INFORMIX_SUBSELECT_NOT_ONE)

子查询的结果不是单一行（Informix 兼容模式）（SQLSTATE 21000）。

-400 (ECPG_PGSQL)

PostgreSQL 服务器导致了某个错误。该消息包含来自 PostgreSQL 服务器的错误消息。

-401 (ECPG_TRANS)

PostgreSQL 服务器通知我们不能启动、提交或回滚事务（SQLSTATE 08007）。

-402 (ECPG_CONNECT)

到数据库的连接尝试没有成功（SQLSTATE 08001）。

-403 (ECPG_DUPLICATE_KEY)

重复键错误，违背唯一约束（SQLSTATE 23505）。

-404 (ECPG_SUBSELECT_NOT_ONE)

子查询的结果不是单一行（SQLSTATE 21000）。

-602 (ECPG_WARNING_UNKNOWN_PORTAL)

指定了一个无效的游标名（SQLSTATE 34000）。

-603 (ECPG_WARNING_IN_TRANSACTION)

事务正在进行（SQLSTATE 25001）。

-604 (ECPG_WARNING_NO_TRANSACTION)

没有活动（正在进行）的事务（SQLSTATE 25P01）。

-605 (ECPG_WARNING_PORTAL_EXISTS)

指定了一个现有的游标名（SQLSTATE 42P03）。

36.9. 预处理器指令

一些预处理器指令可以用来改变 ecpg 预处理器解析和处理一个文件的方式。

36.9.1. 包含文件

要在嵌入式 SQL 程序中包含外部文件，可使用：

```
EXEC SQL INCLUDE filename;
EXEC SQL INCLUDE <filename>;
EXEC SQL INCLUDE "filename";
```

嵌入式 SQL 预处理器会查找名为 *filename*.h 的文件，对它进行预处理，并将其包含在生成的 C 输出中。这样，被包含文件中的嵌入式 SQL 语句就能得到正确处理。

ecpg 预处理器将以下列顺序在几个目录中搜索一个文件：

- 当前目录
- /usr/local/include
- PostgreSQL 的头文件目录，在编译时定义（例如/usr/local/pgsql/include）
- /usr/include

但是当使用 EXEC SQL INCLUDE "*filename*" 时，只有当前目录会被搜索。

在每一个目录中，预处理器将首先按给定的文件名搜索，如果没有找到将会追加.h 到文件名并且重试（除非指定的文件名已经具有该后缀）。

注意 EXEC SQL INCLUDE 不同于：

```
#include <filename.h>
```

因为这样包含的文件不会经过 SQL 命令预处理。当然，仍可使用 C 的#include 指令包含其他头文件。

> **注意**
>
> 被包含文件的名称区分大小写，尽管`EXEC SQL INCLUDE` 命令的其他部分遵循通常的 SQL 大小写规则。

36.9.2. define 和 undef 指令

与 C 中我们熟知的指令`#define` 相似，嵌入式 SQL 具有类似的概念：

```
EXEC SQL DEFINE name;
EXEC SQL DEFINE name value;
```

因此你可以定义一个名称：

```
EXEC SQL DEFINE HAVE_FEATURE;
```

并且你也可以定义常量：

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

使用 `undef` 来移除一个之前的定义：

```
EXEC SQL UNDEF MYNUMBER;
```

当然在你的嵌入式 SQL 程序中你可以继续使用 C 版本的`#define` 和`#undef`。区别在于你定义的值会在哪里被计算。如果你使用 `EXEC SQL DEFINE`，那么`ecpg` 预处理器会计算这些定义并且替换值。例如，如果你写：

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

那么 `ecpg` 将已经做过替换并且你的 C 编译器将永远不会看见名为 `MYNUMBER` 的任何名称或标识符。注意你不能把`#define` 用于一个将要在一个嵌入式 SQL 查询中使用的常量，因为在这种情况下嵌入式 SQL 预处理器不能看到这个声明。

36.9.3. ifdef、ifndef、else、elif 和 endif 指令

可以使用以下指令对代码区段进行条件编译：

`EXEC SQL ifdef name;`

检查 *name*，如果已通过 `EXEC SQL define name` 创建了 *name*，就处理后续各行。

`EXEC SQL ifndef name;`

检查 *name*，如果尚未通过 `EXEC SQL define name` 创建 *name*，就处理后续各行。

```
EXEC SQL else;
```

开始处理一个备选区段，对应由 `EXEC SQL ifdef` *name* 或 `EXEC SQL ifndef` *name* 引入的区段。

```
EXEC SQL elif name;
```

检查 *name*，如果已通过 `EXEC SQL define` *name* 创建了 *name*，就开始一个备选区段。

```
EXEC SQL endif;
```

结束备选区段。

示例：

```
EXEC SQL ifndef TZVAR;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL endif;
```

36.10. 处理嵌入式 SQL 程序

现在你已经对如何构造嵌入式 SQL C 程序有所了解了，你可能希望知道如何编译它们。在编译之前，你需要让该文件通过嵌入式SQL C 预处理器，它会把你用到的SQL 转换成特殊的函数调用。在编译之后，你必须链接一个包含所需函数的特殊库。这些函数从参数中取得信息、使用libpq 执行 SQL 命令并且把结果放在指定的参数中用来输出。

该预处理器程序被称作 `ecpg` 并且被包括在一个正常的 PostgreSQL 安装中。嵌入式 SQL 程序通常带有扩展名`.pgc`。如果你有一个程序文件 `prog1.pgc`，你可以调用下面的命令对它进行预处理：

```
ecpg prog1.pgc
```

这将创建一个文件 `prog1.c`。如果你的输入文件不遵循建议的命名模式，你可以用`-o` 选项显式地指定输出文件。

预处理过的文件可以被正常地编译，例如：

```
cc -c prog1.c
```

生成的 C 源文件会包含 PostgreSQL 安装中的头文件。因此，如果PostgreSQL 安装在默认搜索路径之外，就必须在编译命令行中添加类似`-I/usr/local/pgsql/include` 的选项。

要链接嵌入式 SQL 程序，需要链接`libecpg` 库，例如：

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

同样，可能需要在该命令行中添加类似`-L/usr/local/pgsql/lib` 的选项。

你可以使用 `pg_config` 或者 `pkg-config` 加上包名 `libecpg` 来得到你的安装路径。

如果你使用 make 来管理一个大工程的构建过程，把下面的隐式规则包括在你的 makefile 中将会很方便：

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

`ecpg` 命令的完整语法可见 ecpg。

ecpg 库默认是线程安全的。不过，你可能需要使用一些线程命令行选项来编译你的客户端代码。

36.11. 库函数

`libecpg` 库主要包含用于实现嵌入式 SQL 命令功能的“隐藏”函数，也有一些适合直接调用的函数。注意，直接调用这些函数会使代码失去可移植性。

- 调用 `ECPGdebug(int on, FILE *stream)` 时，如果第一个参数非零，就会启用调试日志。日志写入*stream*，包含代入全部输入变量后的所有SQL 语句，以及PostgreSQL 服务器返回的结果。这对于查找 SQL 语句中的错误很有用。

  > **注意**
  >
  > 在 Windows 上，如果ecpg 库和应用是使用不同标志编译的，这个函数调用将会使应用崩溃，因为`FILE` 指针的内部表示不同。特别地，库和使用库的应用应该使用相同的多线程/单线程、发行/调试以及静态/动态标志。

- `ECPGget_PGconn(const char *connection_name)` 返回由给定名称标识的库数据库连接句柄。如果 *connection_name* 被设置为 `NULL`，当前连接句柄将被返回。如果无法定位到连接句柄，该函数返回`NULL`。如果需要，返回的连接句柄可以被用来调用任何其他来自于 libpq 的函数。

  > **注意**
  >
  > 直接使用 libpq 例程来操纵 ecpg 中建立的数据库连接句柄是一种糟糕的做法。

- `ECPGtransactionStatus(const char *connection_name)` 返回由 *connection_name* 标识的连接的当前事务状态。返回状态码的详细信息，参见第 34.2 节和 libpq 的 `PQtransactionStatus()`。

- 如果你连接到了一个数据库，`ECPGstatus(int lineno, const char* connection_name)` 会返回真；否则返回假。如果使用的是一个单一连接，*connection_name* 可以为 `NULL`。

36.12. 大对象

ECPG 并不直接支持大对象，在调用ECPGget_PGconn() 函数获得所需的 PGconn 对象后，ECPG 应用能通过 libpq 大对象函数操纵大对象（不过，对ECPGget_PGconn() 函数的使用以及直接接触 PGconn 对象都必须非常小心，并且最好不要与其他 ECPG 数据库访问调用混合在一起）。

更多关于 ECPGget_PGconn() 的细节可见第 36.11 节。大对象函数接口的相关信息可见第 35 章。

大对象函数必须在一个事务块中被调用，因此当自动提交关闭时，必须显式地发出BEGIN 命令。

例 36.2 给出了一个例子程序，它展示了在一个 ECPG 应用中如何创建、写入和读取一个大对象。

例 36.2. 访问大对象的 ECPG 程序

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <libpq/libpq-fs.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    PGconn     *conn;
    Oid         loid;
    int         fd;
    char        buf[256];
    int         buflen = 256;
    char        buf2[256];
    int         rc;

    memset(buf, 1, buflen);

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);

    /* 创建 */
    loid = lo_create(conn, 0);
    if (loid &lt; 0)
        printf("lo_create() failed: %s", PQerrorMessage(conn));

    printf("loid = %d\n", loid);

    /* 写入测试 */
    fd = lo_open(conn, loid, INV_READ|INV_WRITE);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_write(conn, fd, buf, buflen);
    if (rc &lt; 0)
        printf("lo_write() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));
```

```
    /* 读取测试 */
    fd = lo_open(conn, loid, INV_READ);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_read(conn, fd, buf2, buflen);
    if (rc &lt; 0)
        printf("lo_read() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* 检查 */
    rc = memcmp(buf, buf2, buflen);
    printf("memcmp() = %d\n", rc);

    /* 清理 */
    rc = lo_unlink(conn, loid);
    if (rc &lt; 0)
        printf("lo_unlink() failed: %s", PQerrorMessage(conn));

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

36.13. C++ 应用

ECPG 对 C++ 应用提供有限支持。本节介绍一些注意事项。

ecpg 预处理器接受一个用 C（或类似 C 的语言）和嵌入式 SQL 命令编写的输入文件，将嵌入式 SQL 命令转换为 C 代码片段，最终生成 .c 文件。在 C++ 中使用时，ecpg 生成的 C 代码片段所调用的库函数，其头文件声明会包在 extern "C" { ... } 块中，因此应该能在 C++ 中无缝工作。

不过，通常ecpg 预处理器只理解 C，它无法处理 C++ 语言的特殊语法和保留词。因此，一些写在 C++ 应用代码中的使用了 C++ 特定复杂特性的嵌入式 SQL 代码可能无法被正确地预处理或者无法按预期工作。

使用 C++ 应用中嵌入式 SQL 代码的安全方法是把 ECPG 调用隐藏在一个 C 模块中，C++ 应用代码会调用它来访问数据库，还要把它和剩余的 C++ 代码链接起来。详见第 36.13.2 节。

36.13.1. 主变量的作用域

ecpg 预处理器理解 C 中变量的作用域。在 C 语言中，这比较简单，因为变量的作用域由所在代码块决定。但在 C++ 中，类成员变量可能在与声明位置不同的代码块中被引用，因此ecpg 预处理器无法理解类成员变量的作用域。

例如，在下面的情况中，ecpg 预处理器无法为 test 方法中的变量 dbname 找到任何声明，因此将发生一个错误。

```
class TestCpp
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
}

void Test::test()
{
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

TestCpp::~TestCpp()
{
    EXEC SQL DISCONNECT ALL;
}
```

这段代码将导致一个这样的错误：

```
ecpg test_cpp.pgc
test_cpp.pgc:28: ERROR: variable "dbname" is not declared
```

为避免这个作用域问题，可以修改test 方法，使用局部变量作为中间存储。不过，这只是一个较差的变通办法，因为它使代码变得难看，还会降低性能。

```
void TestCpp::test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char tmp[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :tmp;
    strlcpy(dbname, tmp, sizeof(tmp));

    printf("current_database = %s\n", dbname);
}
```

36.13.2. 使用外部 C 模块的 C++ 应用开发

了解 ecpg 预处理器在 C++ 中的这些技术限制后，可能会认为：在链接阶段将 C 和 C++ 的目标文件链接起来，使 C++ 应用能够使用 ECPG 功能，会比直接在 C++ 代码中编写嵌入式 SQL 命令更好。本节用一个简单示例介绍如何将嵌入式 SQL 命令与 C++ 应用代码分离。在这个示例中，应用用 C++ 实现，连接 PostgreSQL 服务器的部分则使用 C 和 ECPG。

需要创建三种文件：C 文件（*.pgc）、头文件和 C++ 文件：

test_mod.pgc

一个执行嵌入在 C 中的 SQL 命令的子例程模块。它将被预处理器转换成 test_mod.c。

```
#include "test_mod.h"
#include <stdio.h>

void
db_connect()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
}

void
db_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

void
db_disconnect()
{
    EXEC SQL DISCONNECT ALL;
}
```

test_mod.h

包含 C 模块（test_mod.pgc）中函数声明的头文件，由test_cpp.cpp 包含。由于需要从 C++ 模块链接这些函数，声明必须包在extern "C" 块中。

```
#ifdef __cplusplus
extern "C" {
#endif

void db_connect();
void db_test();
void db_disconnect();

#ifdef __cplusplus
}
#endif
```

test_cpp.cpp

应用的主要代码，包括main 例程，在本示例中还包括一个 C++ 类。

```
#include "test_mod.h"

class TestCpp
{
  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    db_connect();
}

void
TestCpp::test()
{
    db_test();
}

TestCpp::~TestCpp()
{
    db_disconnect();
}

int
main(void)
{
    TestCpp *t = new TestCpp();

    t->test();
    return 0;
}
```

要构建该应用，按以下步骤处理。通过运行ecpg 将 test_mod.pgc 转换为 test_mod.c，并且用 C 编译器将test_mod.c 编译成 test_mod.o：

```
ecpg -o test_mod.c test_mod.pgc
cc -c test_mod.c -o test_mod.o
```

接着，用 C++ 编译器把test_cpp.cpp 编译成 test_cpp.o：

```
c++ -c test_cpp.cpp -o test_cpp.o
```

最后，通过 C++ 编译器驱动程序，将`test_cpp.o` 和 `test_mod.o` 这两个目标文件链接成一个可执行文件：

```
c++ test_cpp.o test_mod.o -lecpg -o test_cpp
```

36.14. 嵌入式 SQL 命令

本节介绍专用于嵌入式 SQL 的全部 SQL 命令。另请参见 SQL 命令中列出的 SQL 命令，除非另有说明，它们也可用于嵌入式 SQL。

ALLOCATE DESCRIPTOR

ALLOCATE DESCRIPTOR — 分配一个 SQL 描述符区域

大纲

```
ALLOCATE DESCRIPTOR name
```

描述

`ALLOCATE DESCRIPTOR` 分配一个新的命名 SQL 描述符区域，可用于在 PostgreSQL 服务器与主程序之间交换数据。

描述符区域使用后应通过 `DEALLOCATE DESCRIPTOR` 命令释放。

参数

name

SQL 描述符的名称，大小写敏感。这可以是一个 SQL 标识符或者一个主变量。

示例

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

兼容性

SQL 标准中说明了 `ALLOCATE DESCRIPTOR`。

另见

DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

CONNECT

CONNECT — 建立一个数据库连接

大纲

```
CONNECT TO connection_target [ AS connection_name ] [ USER connection_user ]
CONNECT TO DEFAULT
CONNECT connection_user
DATABASE connection_target
```

描述

CONNECT 命令在客户端和 PostgreSQL 服务器之间建立一个连接。

参数

connection_target

connection_target 以下列形式之一指定连接的目标服务器。

[*database_name*][@*host*][:*port*]

通过 TCP/IP 连接

unix:postgresql://*host* [:*port*] / [*database_name*][?*connection_option*]

通过 Unix 域套接字连接

tcp:postgresql://*host* [:*port*] / [*database_name*][?*connection_option*]

通过 TCP/IP 连接

SQL 字符串常量

包含上述形式之一的一个值

主变量

类型 char[] 或 VARCHAR[] 的主变量，它包含上述形式之一的一个值

connection_name

用于该连接的一个可选标识符，这样可以在其他命令中引用它。这可以是一个 SQL 标识符或者一个主变量。

connection_user

用于数据库连接的用户名。

使用 *user_name*/*password*、*user_name* IDENTIFIED BY *password* 或者 *user_name* USING *password* 之一，这个参数也能指定用户名和密码。

用户名和密码可以是 SQL 标识符、字符串常量或者主变量。

DEFAULT

按 libpq 的定义使用所有默认连接参数。

示例

这里是一些指定连接参数的变体：

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO "unix:postgresql://200.46.204.71/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main USER :user;
EXEC SQL CONNECT TO :db AS :id;
EXEC SQL CONNECT TO :db USER connectuser USING :pw;
EXEC SQL CONNECT TO @localhost AS main USER connectdb;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO AS main USER connectdb;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO connectdb@localhost AS main;
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb;
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY
 connectpw;
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER connectuser IDENTIFIED BY
 connectpw;
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER connectdb;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main USER connectuser;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY
 "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser USING "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?connect_timeout=14 USER connectuser;
```

这里是一个展示使用主变量指定连接参数的例子程序：

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    char *dbname     = "testdb";    /* 数据库名 */
    char *user       = "testuser";  /* 连接用户名 */
    char *connection = "tcp:postgresql://localhost:5432/testdb";
                                    /* 连接字符串 */
    char ver[256];                  /* 存储版本字符串的缓冲区 */
EXEC SQL END DECLARE SECTION;

    ECPGdebug(1, stderr);

    EXEC SQL CONNECT TO :dbname USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    EXEC SQL CONNECT TO :connection USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    return 0;
}
```

兼容性

SQL 标准中说明了 `CONNECT`，但是连接参数的格式是与实现相关的。

另见

DISCONNECT, SET CONNECTION

DEALLOCATE DESCRIPTOR

DEALLOCATE DESCRIPTOR — 释放一个 SQL 描述符区域

大纲

```
DEALLOCATE DESCRIPTOR name
```

描述

`DEALLOCATE DESCRIPTOR` 释放一个命名的 SQL 描述符区域。

参数

name

要被释放的描述符的名称。它是大小写敏感的。这可以是一个 SQL 标识符或者一个主变量。

示例

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

兼容性

SQL 标准说明了 `DEALLOCATE DESCRIPTOR`。

另见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

DECLARE

DECLARE — 定义一个游标

大纲

```
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT }
 HOLD ] FOR prepared_name
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT }
 HOLD ] FOR query
```

描述

DECLARE 声明一个游标用来在一个预备语句的结果集上迭代。这个命令与直接的 SQL 命令 DECLARE 在语义上有一点点区别：后者会执行一个查询并且准备结果集用于检索，而这个嵌入式 SQL 命令仅仅声明一个名称作为“循环变量”用于在一个查询的结果集上迭代，实际的执行在游标被OPEN 命令打开时才发生。

参数

cursor_name

一个游标名称，大小写敏感。这可以是一个 SQL 标识符或者一个主变量。

prepared_name

一个预备查询的名称，可以是一个 SQL 标识符或者一个主变量。

query

一个提供游标要返回的行的 SELECT 或者 VALUES 命令。

游标选项的含义请见 DECLARE。

示例

为一个查询声明一个游标的例子：

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;
EXEC SQL DECLARE cur1 CURSOR FOR SELECT version();
```

为一个预备语句声明一个游标的例子：

```
EXEC SQL PREPARE stmt1 AS SELECT version();
EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

兼容性

SQL 标准中说明了 DECLARE。

另见

OPEN, CLOSE, DECLARE

DESCRIBE

DESCRIBE — 得到有关一个预备语句或结果集的信息

大纲

```
DESCRIBE [ OUTPUT ] prepared_name USING [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO sqlda_name
```

描述

`DESCRIBE` 获取预备语句所含结果列的元数据，而不实际读取任何行。

参数

prepared_name

一个预备语句的名称。这可以是一个 SQL 标识符或者一个主变量。

descriptor_name

一个描述符名称。它是大小写敏感的。它可以是一个 SQL 标识符或者一个主变量。

sqlda_name

一个 SQLDA 变量的名称。

示例

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

兼容性

SQL 标准中说明了 `DESCRIBE`。

另见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

DISCONNECT

DISCONNECT — 终止一个数据库连接

大纲

```
DISCONNECT connection_name
DISCONNECT [ CURRENT ]
DISCONNECT ALL
```

描述

DISCONNECT 关闭一个（或者所有）到数据库的连接。

参数

connection_name

由 CONNECT 命令建立的数据库连接的名称。

CURRENT

关闭“当前的”连接，它可以是最近打开的连接或者是由SET CONNECTION 命令设置的连接。如果没有参数被传给 DISCONNECT 命令，这将是默认值。

ALL

关闭所有打开的连接。

示例

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb AS con3 USER testuser;

    EXEC SQL DISCONNECT CURRENT;  /* 关闭 con3 */
    EXEC SQL DISCONNECT ALL;      /* 关闭 con2 和 con1 */

    return 0;
}
```

兼容性

SQL 标准中说明了 DISCONNECT。

另见

CONNECT, SET CONNECTION

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE — 动态地预备并执行一个语句

大纲

```
EXECUTE IMMEDIATE string
```

描述

`EXECUTE IMMEDIATE` 立刻预备并且执行一个动态指定的 SQL 语句，不检索结果行。

参数

string

C 字符串字面值，或包含待执行 SQL 语句的主变量。

示例

这里是一个用 `EXECUTE IMMEDIATE` 和一个名为 `command` 的主变量执行 `INSERT` 语句的例子：

```
sprintf(command, "INSERT INTO test (name, amount, letter) VALUES ('db: ''r1''', 1, 'f')");
EXEC SQL EXECUTE IMMEDIATE :command;
```

兼容性

SQL 标准中说明了 `EXECUTE IMMEDIATE`。

GET DESCRIPTOR

GET DESCRIPTOR — 从一个 SQL 描述符区域得到信息

大纲

```
GET DESCRIPTOR descriptor_name :cvariable = descriptor_header_item [, ... ]
GET DESCRIPTOR descriptor_name VALUE column_number :cvariable = descriptor_item [, ... ]
```

描述

`GET DESCRIPTOR` 从一个 SQL 描述符区域检索关于一个查询结果集的信息并且把它存储在主变量中。在使用这个命令把信息传输到主语言变量之前，一个描述符区域通常是用`FETCH` 或 `SELECT` 填充的。

此命令有两种形式。第一种取得描述符的“头部”项目，它们适用于整个结果集，例如行数。第二种需要额外提供列号参数，取得特定列的信息，例如列名和实际列值。

参数

descriptor_name

一个描述符名称。

descriptor_header_item

一个标识要检索哪一个头部信息项的词元。当前只支持用于得到结果集中列数的 `COUNT`。

column_number

要检索其信息的列号。计数从 1 开始。

descriptor_item

一个标识要检索哪一个有关一列信息的项的词元。被支持的项可见第 36.7.1 节。

cvariable

接收从描述符区域检索到的数据的主变量。

示例

检索一个结果集中列数的例子：

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

检索第一列中数据长度的例子：

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
```

把第二列的数据体检索成一个字符串的例子：

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

这里是执行 SELECT current_database();并且显示列数、列数据长度和列数据的完整过程的例子：

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int  d_count;
    char d_data[1024];
    int  d_returned_octet_length;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;

    /* 声明、打开游标，并为游标指定描述符 */
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
    EXEC SQL OPEN cur;
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;

    /* 得到全部列的数量 */
    EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
    printf("d_count                 = %d\n", d_count);

    /* 得到一个返回列的长度 */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
    printf("d_returned_octet_length = %d\n", d_returned_octet_length);

    /* 将返回的列取出成一个字符串 */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
    printf("d_data                  = %s\n", d_data);

    /* 关闭 */
    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

当该例子被执行时，结果看起来是：

```
d_count                   = 1
d_returned_octet_length = 6
d_data                    = testdb
```

兼容性

SQL 标准中说明了 GET DESCRIPTOR。

另见

ALLOCATE DESCRIPTOR, SET DESCRIPTOR

OPEN

OPEN — 打开一个动态游标

大纲

```
OPEN cursor_name
OPEN cursor_name USING value [, ... ]
OPEN cursor_name USING SQL DESCRIPTOR descriptor_name
```

描述

OPEN 打开一个游标并且可选地绑定实际值到游标声明中的占位符。该游标必须之前用 DECLARE 命令声明。OPEN 的执行会导致查询开始在服务器上执行。

参数

cursor_name

要被打开的游标的名称。这可以是一个 SQL 标识符或者一个主变量。

value

要被绑定到游标中一个占位符的值。这可以是一个 SQL 常量、一个主变量或者一个带有指示符的主变量。

descriptor_name

包含要绑定到游标中占位符的值的描述符的名称。这可以是一个 SQL 标识符或者一个主变量。

示例

```
EXEC SQL OPEN a;
EXEC SQL OPEN d USING 1, 'test';
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;
EXEC SQL OPEN :curname1;
```

兼容性

SQL 标准中说明了 OPEN。

另见

DECLARE, CLOSE

PREPARE

PREPARE — 预备一个语句以供执行

大纲

```
PREPARE name FROM string
```

描述

PREPARE 预备一个以字符串动态指定的语句，以供执行。这不同于直接的 SQL 语句 PREPARE，后者也可用于嵌入式程序。EXECUTE 命令可用于执行任一种预备语句。

参数

prepared_name

预备查询的一个标识符。

string

C 字符串字面值，或包含可预备语句的主变量，语句可以是 SELECT、INSERT、UPDATE 或 DELETE。

示例

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";

EXEC SQL ALLOCATE DESCRIPTOR outdesc;
EXEC SQL PREPARE foo FROM :stmt;

EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc;
```

兼容性

SQL 标准中说明了 PREPARE。

参见

EXECUTE

SET AUTOCOMMIT

SET AUTOCOMMIT — 设置当前会话的自动提交行为

大纲

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

描述

`SET AUTOCOMMIT` 设置当前数据库会话的自动提交行为。默认情况下，嵌入式 SQL 程序不处于自动提交模式，需要提交时必须显式发出`COMMIT`。此命令可以将会话改为自动提交模式，使每条语句都被隐式提交。

兼容性

`SET AUTOCOMMIT` 是 PostgreSQL ECPG 的扩展。

SET CONNECTION

SET CONNECTION — 选择一个数据库连接

大纲

```
SET CONNECTION [ TO | = ] connection_name
```

描述

`SET CONNECTION` 设置“当前的”数据库连接，除非被覆盖，所有命令都会使用这个连接。

参数

connection_name

由 `CONNECT` 命令建立的数据库连接的名称。

`CURRENT`

将连接设置为当前连接（因此，什么也不会发生）。

示例

```
EXEC SQL SET CONNECTION TO con2;
EXEC SQL SET CONNECTION = con1;
```

兼容性

SQL 标准中说明了 `SET CONNECTION`。

另见

CONNECT, DISCONNECT

SET DESCRIPTOR

SET DESCRIPTOR — 在一个 SQL 描述符区域中设置信息

大纲

```
SET DESCRIPTOR descriptor_name descriptor_header_item = value [, ... ]
SET DESCRIPTOR descriptor_name VALUE number descriptor_item = value [, ...]
```

描述

`SET DESCRIPTOR` 用值填充一个 SQL 描述符区域。然后该描述符区域通常会被用来在一个预备查询执行中绑定参数。

此命令有两种形式：第一种适用于描述符的“头部”，与某个特定数据项无关。第二种为以编号标识的特定数据项赋值。

参数

descriptor_name

一个描述符名称。

descriptor_header_item

一个标识要设置哪个头部信息项的词元。当前只有设置描述符项数量的 `COUNT` 被支持。

number

要设置的描述符项的编号。计数从 1 开始。

descriptor_item

一个标识在描述符中要设置哪个信息项的词元。受支持的项的列表可见第 36.7.1 节。

value

要存入描述符项的值，可以是 SQL 常量或主变量。

示例

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA = 'some string';
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2;
```

兼容性

SQL 标准中说明了 `SET DESCRIPTOR`。

另见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

TYPE

TYPE — 定义一种新数据类型

大纲

```
TYPE type_name IS ctype
```

描述

`TYPE` 命令定义一个新的 C 类型。它等效于把一个 `typedef` 放在声明节中。

只有使用选项 `-c` 运行 `ecpg` 时才能识别这个命令。

参数

type_name

新类型的名称。这必须是一个合法的 C 类型名。

ctype

一个 C 类型说明。

示例

```
EXEC SQL TYPE customer IS
    struct
    {
        varchar name[50];
        int     phone;
    };

EXEC SQL TYPE cust_ind IS
    struct ind
    {
        short   name_ind;
        short   phone_ind;
    };

EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZ];
EXEC SQL TYPE string IS char[11];
```

这里是一个使用 EXEC SQL TYPE 的例子程序：

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;

EXEC SQL TYPE tt IS
    struct
    {
        varchar v[256];
        int     i;
    };

EXEC SQL TYPE tt_ind IS
    struct ind {
        short   v_ind;
        short   i_ind;
    };

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    tt t;
    tt_ind t_ind;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL SELECT current_database(), 256 INTO :t:t_ind LIMIT 1;

    printf("t.v = %s\n", t.v.arr);
    printf("t.i = %d\n", t.i);

    printf("t_ind.v_ind = %d\n", t_ind.v_ind);
    printf("t_ind.i_ind = %d\n", t_ind.i_ind);

    EXEC SQL DISCONNECT con1;

    return 0;
}
```

这个程序的输出看起来像：

```
t.v = testdb
t.i = 256
t_ind.v_ind = 0
t_ind.i_ind = 0
```

兼容性

TYPE 命令是一种 PostgreSQL 扩展。

VAR

VAR — 定义一个变量

大纲

```
VAR varname IS ctype
```

描述

VAR 命令为主变量指定一个新的 C 数据类型。该主变量必须已在声明节中声明。

参数

varname

一个 C 变量名。

ctype

一个 C 类型说明。

示例

```
Exec sql begin declare section;
short a;
exec sql end declare section;
EXEC SQL VAR a IS int;
```

兼容性

VAR 命令是一个 PostgreSQL 扩展。

WHENEVER

WHENEVER — 指定 SQL 语句触发特定类别条件时要采取的动作

大纲

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } action
```

描述

定义 SQL 执行结果出现特殊情况（未找到行、SQL 警告或错误）时采取的行为。

参数

参数描述见第 36.8.1 节。

示例

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER NOT FOUND DO CONTINUE;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR CALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

典型用法是通过 WHENEVER NOT FOUND BREAK 处理遍历结果集的循环：

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256;
    EXEC SQL OPEN cur;

    /* 到达结果集末尾时，跳出 while 循环 */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        ...
    }

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

兼容性

SQL 标准中说明了 WHENEVER，但是大部分动作是 PostgreSQL 扩展。

36.15. Informix 兼容模式

ecpg 可以运行在一种所谓的 Informix 兼容模式中。如果这种模式被激活，它的行为就好像它是一个用于Informix E/SQL 的 Informix 预处理器。一般而言，这将允许你使用美元符号替代 EXEC SQL 来引入嵌入式 SQL 命令：

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

> **注意**
>
> $与后面的预处理器指令（如 include、define、ifdef 等）之间不能有任何空白。否则，预处理器会将该词元解析为主变量。

有两种兼容性模式：INFORMIX、INFORMIX_SE

在链接使用这种兼容性模式的程序时，要记得链接上和 ECPG 一起发布的libcompat。

除了之前解释过的语法糖，Informix 兼容模式从 E/SQL 中移植了一些用于输入、输出和数据转换的函数以及嵌入式 SQL 语句到 ECPG 中。

Informix 兼容模式与 ECPG 的 pgtypeslib 库密切相关。pgtypeslib 将 SQL 数据类型映射到 C 主程序中的数据类型，Informix 兼容模式中的大多数附加函数用于操作这些 C 主程序类型。但兼容范围有限。它并不试图复制 Informix 的行为，而是允许执行大致相同的操作，并提供名称和基本行为相同的函数；如果目前使用的是Informix，则不能用它直接替代。此外，有些数据类型也有差异。例如，PostgreSQL 的日期时间和 interval 类型不支持 `YEAR TO MINUTE` 这样的范围，ECPG 也不支持。

36.15.1. 附加类型

Informix 模式现在支持 Informix 特有的 “string” 伪类型，用于存储去除尾随空格的字符串数据，无需使用`typedef`。事实上，在 Informix 模式中，ECPG 拒绝处理包含`typedef sometype string;`的源文件。

```
EXEC SQL BEGIN DECLARE SECTION;
string userid; /* 此变量将包含去除尾随空格的数据 */
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH MYCUR INTO :userid;
```

36.15.2. 附加的/缺少的嵌入式 SQL 语句

`CLOSE DATABASE`

这个语句关闭当前连接。实际上，这是 ECPG 的`DISCONNECT CURRENT` 的同义词：

```
$CLOSE DATABASE;                /* close the current connection */
EXEC SQL CLOSE DATABASE;
```

`FREE cursor_name`

由于 ECPG 的工作方式与 Informix 的 ESQL/C 有所不同（即，哪些步骤纯粹是语法转换，哪些步骤依赖于底层运行时库），在 ECPG 中没有`FREE cursor_name` 语句。这是因为在 ECPG 中，`DECLARE CURSOR` 不会转换为使用游标名称的运行时库的函数调用。这意味着在 ECPG 运行时库中没有 SQL 游标的运行时记录，只在 PostgreSQL 服务器中有。

`FREE statement_name`

`FREE statement_name` 是 `DEALLOCATE PREPARE statement_name` 的同义词。

36.15.3. Informix 兼容的 SQLDA 描述符区域

Informix 兼容模式支持的结构体与第 36.7.2 节中所述的不同，如下所示：

```
struct sqlvar_compat
{
    short   sqltype;
    int     sqllen;
    char   *sqldata;
    short  *sqlind;
    char   *sqlname;
    char   *sqlformat;
    short   sqlitype;
    short   sqlilen;
    char   *sqlidata;
    int     sqlxid;
    char   *sqltypename;
    short   sqltypelen;
    short   sqlownerlen;
    short   sqlsourcetype;
    char   *sqlownername;
    int     sqlsourceid;
    char   *sqlilongdata;
    int     sqlflags;
    void   *sqlreserved;
};

struct sqlda_compat
{
    short  sqld;
    struct sqlvar_compat *sqlvar;
    char   desc_name[19];
    short  desc_occ;
    struct sqlda_compat *desc_next;
    void  *reserved;
};

typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqlda_compat     sqlda_t;
```

全局属性是：

`sqld`

SQLDA 描述符中的字段数量。

`sqlvar`

指向各字段属性的指针。

`desc_name`

未使用，用零字节填充。

`desc_occ`

已分配结构体的尺寸。

`desc_next`

如果结果集包含多于一个记录，这个字段是下一个 SQLDA 结构体的指针。

`reserved`

未使用的指针，包含 NULL。为 Informix-兼容性而保留。

各字段的属性如下，它们存储在`sqlvar` 数组中：

`sqltype`

字段类型。常量定义在 `sqltypes.h` 中。

`sqllen`

字段数据的长度。

`sqldata`

指向字段数据的指针，类型为`char *`，所指的数据采用二进制格式。例如：

```
int intval;

switch (sqldata->sqlvar[i].sqltype)
{
    case SQLINTEGER:
        intval = *(int *)sqldata->sqlvar[i].sqldata;
        break;
  ...
}
```

`sqlind`

指向 NULL 指示符的指针。如果由 DESCRIBE 或 FETCH 返回，它始终是有效指针。如果作为 `EXECUTE ... USING sqlda;`的输入，则 NULL 指针表示此字段的值不是 NULL。否则，必须正确设置有效指针和`sqlitype`。例如：

```
if (*(int2 *)sqldata->sqlvar[i].sqlind != 0)
    printf("value is NULL\n");
```

`sqlname`

字段名称，是以 0 终止的字符串。

`sqlformat`

在 Informix 中保留；这里是对该字段调用 `PQfformat()` 所得的值。

`sqlitype`

NULL 指示符数据的类型。从服务器返回数据时，它始终是 SQLSMINT。将`SQLDA` 用于参数化查询时，按所设置的类型处理数据。

`sqlilen`

NULL 指示符数据的长度。

`sqlxid`

字段的扩展类型，即 `PQftype()` 的结果。

`sqltypename`
`sqltypelen`
`sqlownerlen`
`sqlsourcetype`
`sqlownername`
`sqlsourceid`
`sqlflags`
`sqlreserved`

未使用。

`sqlilongdata`

如果 `sqllen` 大于 32kB，它等于`sqldata`。

例子：

```
EXEC SQL INCLUDE sqlda.h;

    sqlda_t        *sqlda; /* 这不需要在嵌入式 DECLARE SECTION 下 */

    EXEC SQL BEGIN DECLARE SECTION;
    char *prep_stmt = "select * from table1";
    int i;
    EXEC SQL END DECLARE SECTION;

    ...

    EXEC SQL PREPARE mystmt FROM :prep_stmt;

    EXEC SQL DESCRIBE mystmt INTO sqlda;

    printf("# of fields: %d\n", sqlda->sqld);
    for (i = 0; i < sqlda->sqld; i++)
      printf("field %d: \"%s\"\n", sqlda->sqlvar[i]->sqlname);

    EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
    EXEC SQL OPEN mycursor;
    EXEC SQL WHENEVER NOT FOUND GOTO out;

    while (1)
    {
      EXEC SQL FETCH mycursor USING sqlda;
    }

    EXEC SQL CLOSE mycursor;

    free(sqlda); /* 只需 free() 主结构；sqlda 和 sqlda->sqlvar 位于同一块已分配区域中 */
```

更多信息参见 `sqlda.h` 头文件和 `src/interfaces/ecpg/test/compat_informix/sqlda.pgc` 回归测试。

36.15.4. 附加函数

`decadd`

将两个 decimal 类型的值相加。

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

该函数接收指向第一个 decimal 类型操作数的指针（`arg1`）、指向第二个 decimal 类型操作数的指针（`arg2`），以及指向用于保存和的 decimal 类型值的指针（`sum`）。成功时返回 0，发生溢出时返回`ECPG_INFORMIX_NUM_OVERFLOW`，发生下溢时返回`ECPG_INFORMIX_NUM_UNDERFLOW`。其他失败情况下返回 -1，并将`errno` 设为 pgtypeslib 中对应的 `errno` 值。

deccmp

比较两个 decimal 类型的变量。

```
int deccmp(decimal *arg1, decimal *arg2);
```

该函数接收指向第一个 decimal 值的指针（arg1）和指向第二个 decimal 值的指针（arg2），并返回一个整数值，表示哪个值更大。

- 1，如果arg1 指向的值大于 var2 指向的值
- -1，如果arg1 指向的值小于 arg2 指向的值
- 0，如果arg1 指向的值和 arg2 指向的值相等

deccopy

复制一个 decimal 值。

```
void deccopy(decimal *src, decimal *target);
```

该函数的第一个参数是指向待复制 decimal 值的指针（src），第二个参数是指向目标 decimal 类型结构体的指针（target）。

deccvasc

将一个值的 ASCII 表示转换为 decimal 类型。

```
int deccvasc(char *cp, int len, decimal *np);
```

该函数接收指向待转换数值的字符串表示的指针（cp）及其长度 len。np 是指向用于保存操作结果的 decimal 值的指针。

例如，有效的格式包括：-2、.794、+3.44、592.49E07 或 -32.84e-4。

该函数成功返回 0。如果发生溢出或下溢，则返回ECPG_INFORMIX_NUM_OVERFLOW 或 ECPG_INFORMIX_NUM_UNDERFLOW。如果 ASCII 表示无法解析，则返回ECPG_INFORMIX_BAD_NUMERIC，或者如果在解析指数时出现问题，则返回 ECPG_INFORMIX_BAD_EXPONENT。

deccvdbl

将 double 类型的值转换为 decimal 类型的值。

```
int deccvdbl(double dbl, decimal *np);
```

该函数接收应该被转换的 double 类型变量作为其第一个参数（dbl）。作为第二个参数（np），该函数接收一个指向应该保存操作结果的 decimal 变量的指针。

该函数在成功时返回 0，在转换失败时返回负值。

deccvint

将 int 类型的值转换为 decimal 类型的值。

```
int deccvint(int in, decimal *np);
```

该函数接收应该被转换的 int 类型变量作为其第一个参数（in）。作为第二个参数（np），该函数接收一个指向应该保存操作结果的 decimal 变量的指针。

该函数在成功时返回 0，在转换失败时返回负值。

deccvlong

将类型为 long 的值转换为类型为 decimal 的值。

```
int deccvlong(long lng, decimal *np);
```

该函数接收应该被转换的类型为 long 的变量作为其第一个参数（lng）。作为第二个参数（np），该函数接收一个指向应该保存操作结果的 decimal 变量的指针。

该函数在成功时返回 0，在转换失败时返回负值。

decdiv

将两个 decimal 类型的变量相除。

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

该函数接收指向第一个（n1）和第二个（n2）操作数的变量的指针，并计算n1/n2。result 是应该保存操作结果的变量的指针。

在成功时返回 0，如果除法失败则返回负值。如果发生溢出或下溢，则函数分别返回 ECPG_INFORMIX_NUM_OVERFLOW 或 ECPG_INFORMIX_NUM_UNDERFLOW。如果尝试除以零，则函数返回ECPG_INFORMIX_DIVIDE_ZERO。

decmul

将两个 decimal 值相乘。

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

该函数接收指向第一个（n1）和第二个（n2）操作数的变量的指针，并计算n1*n2。result 是一个指向应该保存操作结果的变量的指针。

在成功时返回 0，如果乘法失败则返回负值。如果发生溢出或下溢，函数分别返回 ECPG_INFORMIX_NUM_OVERFLOW 或 ECPG_INFORMIX_NUM_UNDERFLOW。

decsub

从一个 decimal 值中减去另一个 decimal 值。

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

该函数接收指向第一个（n1）和第二个（n2）操作数的变量的指针，并计算n1-n2。result 是应该保存操作结果的变量的指针。

在成功时返回 0，如果减法失败则返回负值。如果发生溢出或下溢，函数分别返回 ECPG_INFORMIX_NUM_OVERFLOW 或 ECPG_INFORMIX_NUM_UNDERFLOW。

dectoasc

将 decimal 类型变量转换为 C char* 字符串中的 ASCII 表示。

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

该函数接收指向 decimal 类型变量的指针（np），并将该变量转换为文本表示。cp 是用于保存操作结果的缓冲区。参数 right 指定输出中小数点右侧的位数，结果将舍入到这个小数位数。将right 设为 -1 表示输出所有可用的小数位。如果 len 指定的输出缓冲区长度不足以容纳文本表示及末尾的零字节，则只在结果中存入一个* 字符，并返回 -1。

如果缓冲区 cp 太小，函数返回 -1；如果内存耗尽，则返回ECPG_INFORMIX_OUT_OF_MEMORY。

dectodbl

将 decimal 类型的变量转换为 double。

```
int dectodbl(decimal *np, double *dblp);
```

该函数接受指向待转换 decimal 值的指针（np），以及指向用于保存操作结果的 double 变量的指针（dblp）。

当成功时，返回 0；如果转换失败，则返回负值。

dectoint

将 decimal 类型的变量转换为整数。

```
int dectoint(decimal *np, int *ip);
```

该函数接受指向待转换 decimal 值的指针（np），以及指向用于保存操作结果的整数变量的指针（ip）。

当成功时，返回 0；如果转换失败，则返回负值。如果发生溢出，将返回ECPG_INFORMIX_NUM_OVERFLOW。

请注意，ECPG 实现与Informix 实现不同。Informix 将整数限制在-32767 到 32767 的范围内，而 ECPG 实现中的限制取决于架构（INT_MIN .. INT_MAX）。

`dectolong`

将 decimal 类型的变量转换为长整数。

```
int dectolong(decimal *np, long *lngp);
```

该函数接受指向待转换 decimal 值的指针（`np`），以及指向用于保存操作结果的 long 变量的指针（`lngp`）。

当成功时，返回 0；如果转换失败，则返回负值。如果发生溢出，将返回`ECPG_INFORMIX_NUM_OVERFLOW`。

请注意，ECPG 实现与Informix 实现不同。Informix 将长整型限制在-2,147,483,647 到 2,147,483,647 的范围内，而 ECPG 实现中的限制取决于架构（`-LONG_MAX .. LONG_MAX`）。

`rdatestr`

将日期转换成 C char* 字符串。

```
int rdatestr(date d, char *str);
```

该函数接受两个参数，第一个是待转换的日期（`d`），第二个是指向目标字符串的指针。输出格式始终为 `yyyy-mm-dd`，因此必须为字符串分配至少 11 字节（包括值为零的终止字节）。

该函数在成功时返回 0，在错误时返回负值。

请注意，ECPG 的实现与Informix 的实现不同。在 Informix 中，格式可以通过设置环境变量来影响。然而，在 ECPG 中，你无法更改输出格式。

`rstrdate`

解析日期的文本表示。

```
int rstrdate(char *str, date *d);
```

该函数接收要转换的日期的文本表示（`str`）和指向类型为 date 的变量的指针（`d`）。此函数不允许你指定格式掩码。它使用 Informix 的默认格式掩码，即`mm/dd/yyyy`。在内部，此函数通过`rdefmtdate` 实现。因此，`rstrdate` 不会更快，如果可以选择，应选择允许你显式指定格式掩码的 `rdefmtdate`。

这个函数返回与 `rdefmtdate` 相同的值。

`rtoday`

得到当前日期。

```
void rtoday(date *d);
```

该函数接收一个指向 date 变量的指针（`d`），并将其指向的变量设为当前日期。

在内部，此函数使用PGTYPESdate_today 函数。

rjulmdy

从一个 date 类型的变量中提取日、月和年的值。

```
int rjulmdy(date d, short mdy[3]);
```

该函数接收日期 d 和一个指向包含 3 个 short 整数值的数组 mdy 的指针。变量名表明了顺序：mdy[0] 设为月份数，mdy[1] 设为日，mdy[2] 包含年份。

这个函数目前总是返回 0。

在内部，该函数使用PGTYPESdate_julmdy 函数。

rdefmtdate

使用格式掩码将字符串转换为日期类型的值。

```
int rdefmtdate(date *d, char *fmt, char *str);
```

该函数接收一个指向用于保存操作结果的 date 值的指针（d）、用于解析日期的格式掩码（fmt），以及包含日期文本表示的 C char* 字符串（str）。文本表示应与格式掩码匹配，但字符串与格式掩码无需一一对应。该函数只分析先后顺序，并查找表示年份位置的yy 或 yyyy、表示月份位置的mm 以及表示日位置的 dd。

该函数返回以下值：

- 0 - 函数成功终止。
- ECPG_INFORMIX_ENOSHORTDATE - 日期不包含日、月和年之间的分隔符。在这种情况下，输入字符串必须恰好为 6 或 8 个字节长，但实际不是。
- ECPG_INFORMIX_ENOTDMY - 格式字符串未正确指示年、月和日的顺序。
- ECPG_INFORMIX_BAD_DAY - 输入字符串不包含有效的日。
- ECPG_INFORMIX_BAD_MONTH - 输入字符串不包含有效的月。
- ECPG_INFORMIX_BAD_YEAR - 输入字符串不包含有效的年。

在内部，此函数实现为使用PGTYPESdate_defmt_asc 函数。请参阅那里的参考资料，了解示例输入表。

rfmtdate

使用格式掩码将 date 类型的变量转换为文本表示。

```
int rfmtdate(date d, char *fmt, char *str);
```

该函数接收要转换的日期（d）、格式掩码（fmt）和将保存日期文本表示的字符串（str）。

成功时返回 0，发生错误时返回负值。

在内部，此函数使用PGTYPESdate_fmt_asc 函数，有关示例，请参阅那里的参考资料。

rmdyjul

根据一个包含 3 个 short 整数的数组创建日期值，这些整数指定日期的日、月和年。

```
int rmdyjul(short mdy[3], date *d);
```

该函数接收包含 3 个 short 整数的数组（mdy）和一个指向 date 类型变量的指针，用来保存操作结果。

目前该函数始终返回 0。

在内部，该函数实现为使用函数 PGTYPESdate_mdyjul。

rdayofweek

为一个日期值返回表示它是星期几的数字。

```
int rdayofweek(date d);
```

这个函数接收日期变量 d 作为它唯一的参数并且返回一个整数说明这个日期是星期几。

- 0 - 星期日
- 1 - 星期一
- 2 - 星期二
- 3 - 星期三
- 4 - 星期四
- 5 - 星期五
- 6 - 星期六

在内部，该函数被实现为使用函数 PGTYPESdate_dayofweek。

dtcurrent

检索当前时间戳。

```
void dtcurrent(timestamp *ts);
```

该函数检索当前时间戳，并将其保存到ts 指向的时间戳变量中。

`dtcvasc`

将时间戳的文本表示解析为 timestamp 变量。

```
int dtcvasc(char *str, timestamp *ts);
```

该函数接收要解析的字符串（`str`），以及指向用于保存操作结果的 timestamp 变量的指针（`ts`）。

该函数在成功时返回 0，在错误时返回负值。

在内部，此函数使用`PGTYPEStimestamp_from_asc` 函数。请参阅那里的参考资料，了解包含示例输入的表格。

`dtcvfmtasc`

使用格式掩码将时间戳的文本表示解析为 timestamp 变量。

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

该函数接收要解析的字符串（`inbuf`）、使用的格式掩码（`fmtstr`），以及指向用于保存操作结果的 timestamp 变量的指针（`dtvalue`）。

这个函数是通过 `PGTYPEStimestamp_defmt_asc` 函数实现的。请参阅那里的文档，了解可用的格式说明符列表。

该函数在成功时返回 0，在错误时返回负值。

`dtsub`

从一个时间戳中减去另一个时间戳，并返回一个 interval 类型的变量。

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

该函数从 `ts1` 指向的时间戳变量中减去 `ts2` 指向的时间戳变量，并将结果存入`iv` 指向的 interval 变量中。

成功时返回 0，发生错误时返回负值。

`dttoasc`

将 timestamp 变量转换为 C char* 字符串。

```
int dttoasc(timestamp *ts, char *output);
```

该函数接收指向待转换 timestamp 变量的指针（`ts`），以及用于保存操作结果的字符串（`output`）。它将 `ts` 转换为符合 SQL 标准的文本表示，格式为`YYYY-MM-DD HH:MM:SS`。

成功时返回 0，发生错误时返回负值。

dttofmtasc

使用格式掩码将 timestamp 变量转换为 C char*。

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

该函数的第一个参数是指向待转换时间戳的指针（ts），随后是指向输出缓冲区的指针（output）、为输出缓冲区分配的最大长度（str_len），以及转换所用的格式掩码（fmtstr）。

成功时返回 0，发生错误时返回负值。

在内部，此函数使用PGTYPESTimestamp_fmt_asc 函数。请参阅那里的参考资料，了解可以使用哪些格式掩码说明符。

intoasc

将 interval 变量转换为 C char* 字符串。

```
int intoasc(interval *i, char *str);
```

该函数接受指向待转换 interval 变量的指针（i），以及用于保存操作结果的字符串（str）。它将 i 转换为符合 SQL 标准的文本表示，格式为 YYYY-MM-DD HH:MM:SS。

成功时返回 0，发生错误时返回负值。

rfmtlong

将长整型值使用格式掩码转换为其文本表示形式。

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

该函数接收长整型值 lng_val，格式掩码fmt 和指向输出缓冲区 outbuf 的指针。它根据格式掩码将长整型值转换为其文本表示形式。

格式掩码可以由以下格式指定字符组成：

- *（星号）- 如果此位置为空白，用星号填充。
- &（和号）- 如果此位置为空白，用零填充。
- # - 将前导零转换为空格。
- < - 将数字左对齐在字符串中。
- ,（逗号）- 将四位或更多位的数字每三位分为一组，用逗号分隔。
- .（句点）- 此字符将整数部分与小数部分分隔开。
- -（减号）- 如果数字是负值，则显示减号。
- +（加号）- 如果数字是正值，则显示加号。
- （ - 这个字符替换负数前面的减号。减号不会显示。

-) - 这个字符替换减号，并打印在负值后面。
- $ - 货币符号。

rupshift

将字符串转换为大写。

```
void rupshift(char *str);
```

该函数接收一个指向字符串的指针，并将每个小写字符转换为大写。

byleng

返回字符串中字符的数量，不包括末尾的空格。

```
int byleng(char *str, int len);
```

该函数期望一个固定长度的字符串作为第一个参数 (str)，并将其长度作为第二个参数 (len)。它返回有效字符的数量，即不包括末尾空格的字符串长度。

ldchar

将固定长度的字符串复制到以空字符结尾的字符串中。

```
void ldchar(char *src, int len, char *dest);
```

该函数接收要复制的固定长度字符串（src）、其长度（len）和指向目标内存的指针（dest）。请注意，你需要为dest 指向的字符串保留至少 len+1 字节。该函数最多复制 len 字节到新位置（如果源字符串具有尾随空格，则会少一些），并添加空字符终止符。

rgetmsg

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

这个函数存在，但目前尚未实现！

rtypalign

```
int rtypalign(int offset, int type);
```

这个函数存在，但目前尚未实现！

rtypmsize

```
int rtypmsize(int type, int len);
```

这个函数存在，但目前尚未实现！

`rtypwidth`

```
int rtypwidth(int sqltype, int sqllen);
```

这个函数存在，但目前尚未实现！

`rsetnull`

将一个变量设置为 NULL。

```
int rsetnull(int t, char *ptr);
```

该函数接收一个整数，表示变量的类型，以及一个指向变量本身的指针，该指针被转换为 C 中的 char* 指针。

下列类型存在：

- `CCHARTYPE` - 用于 `char` 或 `char*` 类型的变量
- `CSHORTTYPE` - 用于 `short int` 类型的变量
- `CINTTYPE` - 用于 `int` 类型的变量
- `CBOOLTYPE` - 用于 `boolean` 类型的变量
- `CFLOATTYPE` - 用于 `float` 类型的变量
- `CLONGTYPE` - 用于 `long` 类型的变量
- `CDOUBLETYPE` - 用于 `double` 类型的变量
- `CDECIMALTYPE` - 用于 `decimal` 类型的变量
- `CDATETYPE` - 用于 `date` 类型的变量
- `CDTIMETYPE` - 用于 `timestamp` 类型的变量

这是调用此函数的示例：

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

`risnull`

测试变量是否为 NULL。

```
int risnull(int t, char *ptr);
```

该函数接收要测试的变量类型（`t`）以及指向该变量的指针（`ptr`）。请注意，后者需要转换为 char* 类型。查看函数`rsetnull` 以获取可能的变量类型列表。

这是如何使用这个函数的示例：

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

36.15.5. 额外的常量

注意这里所有的常量都描述错误并且它们都被定义为表示负值。在每一种不同常量的描述中，你还可以找到在当前实现中该常量表示的值。不过你不应该依赖于这个数字。但是你可以相信所有的这些常量都是被定义为表示负值。

`ECPG_INFORMIX_NUM_OVERFLOW`

如果在一次计算中发生了溢出，函数会返回这个值。在内部它被定义为 -1200（Informix 定义）。

`ECPG_INFORMIX_NUM_UNDERFLOW`

如果在一次计算中发生了下溢，函数会返回这个值。在内部它被定义为 -1201（Informix 定义）。

`ECPG_INFORMIX_DIVIDE_ZERO`

如果发现尝试除零，函数会返回这个值。在内部它被定义为 -1202（Informix 定义）。

`ECPG_INFORMIX_BAD_YEAR`

如果解析日期时发现无效的年份值，函数返回此值。其内部定义为 -1204（Informix 的定义）。

`ECPG_INFORMIX_BAD_MONTH`

如果解析日期时发现无效的月份值，函数返回此值。其内部定义为 -1205（Informix 的定义）。

`ECPG_INFORMIX_BAD_DAY`

如果解析日期时发现无效的日值，函数返回此值。其内部定义为 -1206（Informix 的定义）。

`ECPG_INFORMIX_ENOSHORTDATE`

如果解析例程需要短日期表示，但未得到长度正确的日期字符串，函数返回此值。其内部定义为 -1209（Informix 的定义）。

`ECPG_INFORMIX_DATE_CONVERT`

如果在日期格式化时产生了一个错误，函数会返回这个值。在内部它被定义为 -1210（Informix 定义）。

`ECPG_INFORMIX_OUT_OF_MEMORY`

如果在操作时内存被耗尽，函数会返回这个值。在内部它被定义为 -1211（Informix 定义）。

`ECPG_INFORMIX_ENOTDMY`

如果一个解析例程被假定为得到一个格式掩码（如 `mmddyy`）但是列出的域并不是全部正确，函数会返回这个值。在内部它被定义为 -1212（Informix 定义）。

`ECPG_INFORMIX_BAD_NUMERIC`

如果解析例程因 numeric 值的文本表示有误而无法解析，或者例程因至少一个 numeric 变量无效而无法完成涉及 numeric 变量的计算，函数返回此值。其内部定义为 -1213（Informix 的定义）。

`ECPG_INFORMIX_BAD_EXPONENT`

如果一个解析例程不能解析一个指数，函数会返回这个值。在内部它被定义为 -1216（Informix 定义）。

`ECPG_INFORMIX_BAD_DATE`

如果一个解析例程不能解析一个日期，函数会返回这个值。在内部它被定义为 -1218（Informix 定义）。

`ECPG_INFORMIX_EXTRA_CHARS`

如果一个解析例程被传递了它不能解析的额外字符，函数会返回这个值。在内部它被定义为 -1264（Informix 定义）。

36.16. Oracle 兼容模式

`ecpg` 可以在所谓的 Oracle 兼容模式下运行。如果此模式激活，它会尝试表现得好像它是 Oracle 的Pro*C。

具体来说，此模式以三种方式更改`ecpg`：

- 在接收字符串类型数据的字符数组末尾填充空格，直至指定长度
- 以零字节终止这些字符数组，并在截断发生时设置指示符变量
- 当字符数组接收空字符串时，将空值指示符设为`-1`

36.17. 内部

这一节解释 ECPG 在内部如何工作。这些信息有时有助于用户理解如何使用 ECPG。

`ecpg` 写入输出的前四行是固定的：其中两行是注释，另外两行是与库接口所必需的包含行。随后预处理器会通读整个文件并写出输出，通常它只是把所有内容原样回显到输出中。

当它看到 `EXEC SQL` 语句时，就会介入并对其进行改写。命令以`EXEC SQL` 开始，以`;`结束。其间的所有内容都会被视为一条 SQL 语句，并进行变量替换解析。

当某个符号以冒号（`:`）开头时，就会发生变量替换。系统会在此前于`EXEC SQL DECLARE` 声明节中声明的变量里查找同名变量。

库中最重要的函数是 `ECPGdo`，负责执行大多数命令。它接收可变数量的参数，参数数量很容易达到 50 个左右；我们希望这在任何平台上都不会造成问题。

参数是：

一个行号

这是原始行的行号，只用于错误消息。

一个字符串

这是将要发出的 SQL 命令。输入变量会对它进行修改，也就是那些在编译时尚未知晓、但要填入命令中的变量。字符串中的`?`表示变量应当放置的位置。

输入变量

每个输入变量都会生成十个参数（见下文）。

`ECPGt_EOIT`

一个表明后面没有更多输入变量的 `enum` 值。

输出变量

每个输出变量都会生成十个参数（见下文）。这些变量将由该函数填充。

`ECPGt_EORT`

一个表明后面没有更多变量的 `enum` 值。

对于每一个作为 SQL 命令一部分的变量，该函数得到十个参数：

1. 用特殊符号表示的类型。
2. 一个值的指针或者一个指针的指针。
3. 如果变量是一个 `char` 或者 `varchar`，这是它的尺寸。
4. 数组中元素的数量（用于数组取数）。
5. 到数组中下一个元素的偏移量（用于数组取数）。
6. 用特殊符号表示的指示符变量类型。

7. 一个指示符变量的指针。

8. 0

9. 指示符数组中的元素数量（用于数组取数）。

10. 到指示符数组中下一个元素的偏移量（用于数组取数）。

注意并非所有 SQL 命令都被以这种方式对待。例如，一个打开游标语句：

```
EXEC SQL OPEN cursor;
```

它不会被复制到输出中。相反，游标的DECLARE 命令会被放在 OPEN 命令的位置上，因为真正打开游标的正是它。

这里有一个完整的例子，它描述了一个文件foo.pgc 的预处理器输出（对预处理器的每一个特定版本细节可能不同）：

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

会被转换成：

```
/* 由 ecpg (2.6.0) 处理 */
/* 这两个头文件由预处理器增加 */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* 声明节开始 */

#line 1 "foo.pgc"

 int index;
 int result;
/* 声明节结束 */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

（这里的缩进是为了可读性而添加的，并非是预处理器做的处理）。

第 37 章 信息模式

信息模式由一组视图组成，其中包含当前数据库中定义的对象的信息。信息模式由 SQL 标准定义，因此可以预期它具有可移植性并保持稳定；而系统目录则不同，它们是 PostgreSQL 特有的，并且是围绕实现方面的考虑建模的。不过，信息模式视图不包含 PostgreSQL 特有功能的信息；要查询这些信息，需要查询系统目录或其他 PostgreSQL 特有视图。

> **注意**
>
> 当在数据库中查询约束信息时，一个期望返回一行的标准兼容的查询可能返回多行。这是因为 SQL 标准要求约束名在一个模式中唯一，但是PostgreSQL 并不强制这种限制。PostgreSQL 自动产生的约束名避免在相同的模式中重复，但是用户能够指定这种重复的名称。
>
> 这个问题可能在查询信息模式视图时出现，例如`check_constraint_routine_usage`、`check_constraints`、`domain_constraints` 和 `referential_constraints`。一些其他视图也有相似的问题，但是它们包含了表名来帮助区分重复行，例如`constraint_column_usage`、`constraint_table_usage`、`table_constraints`。

37.1. 模式

信息模式本身是一个名为 `information_schema` 的模式。这个模式自动存在于所有数据库中。这个模式的拥有者是集簇中的初始数据库用户，并且该用户自然地拥有这个模式上的所有权限，包括删除它的能力（但是这样节省的空间是很小的）。

默认情况下，信息模式不在模式搜索路径中，因此你需要使用限定名访问其中的所有对象。因为信息模式中的某些对象的名称是可能出现在用户应用中的一般名称，如果你想把该信息模式放在路径中，你应该小心。

37.2. 数据类型

信息模式视图的列使用在信息模式中定义的特殊数据类型。这些类型被定义为普通内置类型之上的简单域。不应在信息模式之外使用这些类型，但如果应用要从信息模式中查询数据，就必须能够处理它们。

这些类型是：

`cardinal_number`

一种非负整数。

`character_data`

一种字符串（没有指定最大长度）。

`sql_identifier`

一种字符串。该类型用于 SQL 标识符，其他任何文本数据则使用类型`character_data`。

time_stamp

在类型 timestamp with time zone 之上的一个域。

yes_or_no

一种字符串域，只包含YES 或 NO。它用于在信息模式中表示布尔（真/假）数据。（信息模式是在类型 boolean 被纳入 SQL 标准之前设计出来的，因此需要采用这种约定以保持向后兼容。）

信息模式中的每一列都是这五种类型之一。

37.3. information_schema_catalog_name

information_schema_catalog_name 是一个表，它始终只包含一行一列，用于存放当前数据库的名称（按 SQL 术语称为当前目录）。

表 37.1. information_schema_catalog_name 列

名称	数据类型	描述
catalog_name	sql_identifier	包含这个信息模式的数据库名称

37.4. administrable_role_authorizations

视图 administrable_role_authorizations 标识当前用户对其有管理选项的所有角色。

表 37.2. administrable_role_authorizations 列

名称	数据类型	描述
grantee	sql_identifier	被授予该角色成员资格的角色名称（可以是当前用户，也可以是在嵌套角色成员资格情况下的另一个角色）
role_name	sql_identifier	一个角色的名称
is_grantable	yes_or_no	总是 YES

37.5. applicable_roles

视图 applicable_roles 标识当前用户可以使用其权限的所有角色。这意味着从当前用户到相关角色之间存在某条角色授予链。当前用户本身也是一个适用角色。适用角色的集合通常用于权限检查。

表 37.3. applicable_roles 列

名称	数据类型	描述
grantee	sql_identifier	被授予该角色成员资格的角色名称（可以是当前用户，也可以是在嵌套角色成员资格情况下的另一个角色）
role_name	sql_identifier	一个角色的名称
is_grantable	yes_or_no	如果被授权者在角色上有管理选项则为YES，如果没有则为NO

37.6. attributes

视图 attributes 包含数据库中定义的复合数据类型的属性信息。（注意，该视图不提供表列的信息，而表列在 PostgreSQL 的语境中有时也称为属性。）只显示当前用户能够访问的属性（作为该类型的拥有者或具有该类型上的某种权限）。

表 37.4. attributes 列

名称	数据类型	描述
udt_catalog	sql_identifier	包含该数据类型的数据库名称（总是当前数据库）
udt_schema	sql_identifier	包含该数据类型的模式名称
udt_name	sql_identifier	数据类型名称
attribute_name	sql_identifier	属性名称
ordinal_position	cardinal_number	属性在该数据类型内部的顺序位置（从 1 开始计算）
attribute_default	character_data	该属性的默认表达式
is_nullable	yes_or_no	如果该属性可能为 null，则为 YES；如果已知不能为 null，则为 NO。
data_type	character_data	如果该属性是一个内置类型，此列值为该属性的数据类型；如果该属性是某种数组，此列值为ARRAY（在这种情况下，见视图element_types）；其他情况，此列值为USER-DEFINED（在这种情况下，该类型在attribute_udt_name 和相关列中标识）。
character_maximum_length	cardinal_number	如果data_type 标识字符类型或位串类型，则为声明的最大长度；对于其他所有数据类型，或未声明最大长度时，则为 null。
character_octet_length	cardinal_number	如果data_type 标识字符类型，这里是一个数据值可能的最大字节长度；对于所有其他数据类型则为 null。最大字节长度取决于声明的字符最大长度（见上文）和服务器编码。
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	包含该属性排序规则的数据库名称（总是当前数据库）；如果使用默认排序规则，或该属性的数据类型不支持排序规则，则为 null。
collation_schema	sql_identifier	包含该属性排序规则的模式名称；如果使用默认排序规则，或该属性的数据类型不支持排序规则，则为 null。

名称	数据类型	描述
collation_name	sql_identifier	该属性排序规则的名称；如果使用默认排序规则，或该属性的数据类型不支持排序规则，则为 null。
numeric_precision	cardinal_number	如果data_type 标识数值类型，此列包含该属性数据类型的（声明的或隐式的）精度。精度表示有效位数。它可以按十进制（基数 10）或二进制（基数 2）表示，具体由numeric_precision_radix 列指定。对于其他所有数据类型，此列为 null。
numeric_precision_radix	cardinal_number	如果data_type 标识数值类型，此列指示numeric_precision 和numeric_scale 列中的值采用的进制。值为 2 或 10。对于其他所有数据类型，此列为 null。
numeric_scale	cardinal_number	如果data_type 标识精确数值类型，此列包含该属性数据类型的（声明的或隐式的）小数位数。小数位数表示小数点右侧的有效位数。它可以按十进制（基数 10）或二进制（基数 2）表示，具体由numeric_precision_radix 列指定。对于其他所有数据类型，此列为 null。
datetime_precision	cardinal_number	如果data_type 标识日期、时间、时间戳或时间间隔类型，此列包含该属性数据类型的（声明的或隐式的）小数秒精度，即秒值小数点后保留的十进制位数。对于其他所有数据类型，此列为 null。
interval_type	character_data	如果data_type 标识时间间隔类型，此列说明该属性的时间间隔包含哪些字段，例如YEAR TO MONTH、DAY TO SECOND 等。未指定字段限制时（即时间间隔接受所有字段），或对于其他所有数据类型，此列为 null。
interval_precision	cardinal_number	对应PostgreSQL不支持的特性（关于时间间隔类型属性的小数秒精度，参见datetime_precision）
attribute_udt_catalog	sql_identifier	属性数据类型被定义的数据库名（总是当前数据库）
attribute_udt_schema	sql_identifier	属性数据类型被定义的模式名
attribute_udt_name	sql_identifier	属性数据类型的名称
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
maximum_cardinality	cardinal_number	总是为 null，因为 PostgreSQL 中数组的最大基数始终不受限制

名称	数据类型	描述
dtd_identifier	sql_identifier	该列的数据类型描述符的标识符，在属于同一表的数据类型描述符中是唯一的。它主要用于与其他此类标识符实例进行连接。（标识符的具体格式没有定义，也不保证在未来版本中保持不变。）
is_derived_reference_attribute	yes_or_no	对应 PostgreSQL 不支持的特性

关于某些列的详细信息，也可参见结构相似的视图第 37.16 节中的说明。

37.7. character_sets

视图 character_sets 标识当前数据库中可用的字符集。因为 PostgreSQL 不支持在同一个数据库中有多个字符集，这个视图只显示一个字符集，它就是数据库编码。

请注意 SQL 标准中以下术语的用法：

字符集合

抽象的字符集合，例如UNICODE、UCS 或 LATIN1。它不作为 SQL 对象公开，但可以在此视图中看到。

字符编码形式

某个字符集合的编码方式。大多数较早的字符集合只使用一种编码形式，因此没有为两者分别命名（例如，LATIN1 是适用于 LATIN1 字符集合的编码形式）。但 Unicode 就有 UTF8、UTF16 等编码形式（PostgreSQL 并非全部支持）。编码形式不作为 SQL 对象公开，但可以在此视图中看到。

字符集

一个有名称的 SQL 对象，用来标识字符集合、字符编码和默认排序规则。预定义字符集通常与编码形式同名，但用户也可以定义其他名称。例如，字符集UTF8 通常标识字符集合 UCS、编码形式UTF8 以及某个默认排序规则。

在 PostgreSQL 中，可以把“编码”视为字符集，也可以视为字符编码形式。它们具有相同的名称，并且一个数据库中只能有一个。

表 37.5. character_sets 列

名称	数据类型	描述
character_set_catalog	sql_identifier	当前字符集并未被实现为模式对象，因此这一列为 null。
character_set_schema	sql_identifier	当前字符集并未被实现为模式对象，因此这一列为 null。
character_set_name	sql_identifier	该字符集的名字，当前实现为显示该数据库编码的名字
character_repertoire	sql_identifier	字符集合；如果编码为UTF8 则显示UCS，否则仅显示编码名称
form_of_use	sql_identifier	字符编码形式，与数据库编码相同

名称	数据类型	描述
default_collate_catalog	sql_identifier	包含默认排序规则的数据库名称（只要标识了某个排序规则，就总是当前数据库）
default_collate_schema	sql_identifier	包含该默认排序规则的模式名
default_collate_name	sql_identifier	默认排序规则的名字。该默认排序规则被标识为匹配当前数据库的 COLLATE 和CTYPE 设置的排序规则。如果没有这样的排序规则，则这一列以及相关的模式列和目录列都为 null。

37.8. check_constraint_routine_usage

视图 check_constraint_routine_usage 标识检查约束所使用的例程（函数和过程）。只有当前启用角色拥有的那些例程才会显示出来。

表 37.6. check_constraint_routine_usage 列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含约束的模式的名称
constraint_name	sql_identifier	约束的名称
specific_catalog	sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	函数的“特定名称”。详见第 37.40 节。

37.9. check_constraints

视图 check_constraints 包含所有检查约束，无论它们是定义在表上还是定义在域上，只要它们归当前启用角色所有即可显示（表或域的拥有者就是约束的拥有者）。

表 37.7. check_constraints 列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含约束的模式的名称
constraint_name	sql_identifier	约束的名称
check_clause	character_data	该检查约束的检查表达式

37.10. collations

视图 collations 包含在当前数据库中可用的排序规则。

表 37.8. collations 列

名称	数据类型	描述
collation_catalog	sql_identifier	包含该排序规则的数据库名称（总是当前数据库）
collation_schema	sql_identifier	包含该排序规则的模式名称
collation_name	sql_identifier	默认排序规则的名称
pad_attribute	character_data	总是NO PAD（另一种选择PAD SPACE 没有被 PostgreSQL 支持）

37.11. collation_character_set_applicability

视图 collation_character_set_applicability 标识可用的排序规则适用于哪些字符集。在 PostgreSQL 中，每个数据库中只有一种字符集（解释见第 37.7 节），因此这个视图没有提供很有用的信息。

表 37.9. collation_character_set_applicability 列

名称	数据类型	描述
collation_catalog	sql_identifier	包含该排序规则的数据库名称（总是当前数据库）
collation_schema	sql_identifier	包含该排序规则的模式名称
collation_name	sql_identifier	默认排序规则的名称
character_set_catalog	sql_identifier	字符集当前还未被实现为模式对象，所以这一列为 null
character_set_schema	sql_identifier	字符集当前还未被实现为模式对象，所以这一列为 null
character_set_name	sql_identifier	字符集名称

37.12. column_domain_usage

视图 column_domain_usage 标识所有使用定义在当前数据库中并且被一个当前启用的角色拥有的域的列（表列或视图列）。

表 37.10. column_domain_usage 列

名称	数据类型	描述
domain_catalog	sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名称
domain_name	sql_identifier	域名称
table_catalog	sql_identifier	包含表的数据库的名称（总是当前数据库）
table_schema	sql_identifier	包含表的模式的名称
table_name	sql_identifier	表的名称
column_name	sql_identifier	列名称

37.13. column_options

视图 column_options 包含为当前数据库中外部表列定义的所有选项。只有当前用户能够访问（作为拥有者或具有某些权限）的那些外部表列才被显示。

表 37.11. column_options 列

名称	数据类型	描述
table_catalog	sql_identifier	包含该外部表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该外部表的模式名称
table_name	sql_identifier	外部表名称
column_name	sql_identifier	列名称
option_name	sql_identifier	一个选项名称
option_value	character_data	该选项的值

37.14. column_privileges

视图 column_privileges 标识列上授予给当前已启用角色或由当前已启用角色授出的所有权限。每个列、授予者和被授权者的组合对应一行。

如果一个权限被授予在一整个表上，它在这个视图中被显示为在每一列上授予，但是只有可用于列粒度的权限类型才会这样：SELECT、INSERT、UPDATE、REFERENCES。

表 37.12. column_privileges 列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的角色名称
grantee	sql_identifier	被授予该权限的角色名称
table_catalog	sql_identifier	包含该列的表所在的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该列的表所在的模式名称
table_name	sql_identifier	包含该列的表名称
column_name	sql_identifier	列名称
privilege_type	character_data	权限类型：SELECT、INSERT、UPDATE 或 REFERENCES
is_grantable	yes_or_no	如果该权限可再授予，则为YES，否则为NO

37.15. column_udt_usage

视图 column_udt_usage 标识所有使用被一个当前启用的角色拥有的数据类型的列。注意在 PostgreSQL 中，内置数据类型的行为和用户定义的类型相似，因此它们也被包括在这里。详见第 37.16 节。

表 37.13. column_udt_usage 列

名称	数据类型	描述
udt_catalog	sql_identifier	该列数据类型（如果适用，则为域的底层类型）所在的数据库名称（总是当前数据库）
udt_schema	sql_identifier	该列数据类型（如果适用，则为域的底层类型）所在的模式名称
udt_name	sql_identifier	该列数据类型（如果适用，则为域的底层类型）的名称
table_catalog	sql_identifier	包含表的数据库的名称（总是当前数据库）
table_schema	sql_identifier	包含表的模式的名称
table_name	sql_identifier	表的名称
column_name	sql_identifier	列名称

37.16. columns

视图 columns 包含数据库中有关所有表列（或视图列）的信息。系统列（oid 等）不被包括在内。只有那些当前用户能够访问（作为拥有者或具有某些权限）的列才被显示。

表 37.14. columns 列

名称	数据类型	描述
table_catalog	sql_identifier	包含表的数据库的名称（总是当前数据库）
table_schema	sql_identifier	包含表的模式的名称
table_name	sql_identifier	表的名称
column_name	sql_identifier	列名称
ordinal_position	cardinal_number	该列在表内的顺序位置（从 1 开始计）
column_default	character_data	该列的默认表达式
is_nullable	yes_or_no	如果该列可能为 null，则为YES；如果已知不能为 null，则为NO。非空约束是确定列不能为 null 的一种依据，但也可能有其他依据。
data_type	character_data	如果该列的数据类型是内置类型，则为该列的数据类型；如果是数组，则为ARRAY（此时参见视图element_types）；否则为USER-DEFINED（此时由udt_name 及相关列标识该类型）。如果该列基于一个域，则此列指的是该域的底层类型（该域由domain_name 及相关列标识）。
character_maximum_length	cardinal_number	如果data_type 标识字符类型或位串类型，则为声明的最大长度；对于其他所有数据类型，或未声明最大长度时，则为 null。

名称	数据类型	描述
character_octet_length	cardinal_number	如果data_type 标识字符类型，这里是一个数据值可能的最大字节长度；对于所有其他数据类型则为 null。最大字节长度取决于声明的字符最大长度（见上文）和服务器编码。
numeric_precision	cardinal_number	如果data_type 标识数值类型，此列包含该列数据类型的（声明的或隐式的）精度。精度表示有效位数。它可以按十进制（基数 10）或二进制（基数 2）表示，具体由 numeric_precision_radix 列指定。对于其他所有数据类型，此列为 null。
numeric_precision_radix	cardinal_number	如果data_type 标识数值类型，此列指示numeric_precision 和numeric_scale 列中的值采用的进制。值为 2 或 10。对于其他所有数据类型，此列为 null。
numeric_scale	cardinal_number	如果data_type 标识精确数值类型，此列包含该列数据类型的（声明的或隐式的）小数位数。小数位数表示小数点右侧的有效位数。它可以按十进制（基数 10）或二进制（基数 2）表示，具体由 numeric_precision_radix 列指定。对于其他所有数据类型，此列为 null。
datetime_precision	cardinal_number	如果data_type 标识日期、时间、时间戳或时间间隔类型，此列包含该列数据类型的（声明的或隐式的）小数秒精度，即秒值小数点后保留的十进制位数。对于其他所有数据类型，此列为 null。
interval_type	character_data	如果data_type 标识时间间隔类型，此列说明该列的时间间隔包含哪些字段，例如YEAR TO MONTH、DAY TO SECOND 等。未指定字段限制时（即时间间隔接受所有字段），或对于其他所有数据类型，此列为 null。
interval_precision	cardinal_number	对应PostgreSQL不支持的特性（关于时间间隔类型列的小数秒精度，参见datetime_precision）
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	包含该列排序规则的数据库名称（总是当前数据库）；如果使用默认排序规则，或该列的数据类型不支持排序规则，则为 null。

名称	数据类型	描述
collation_schema	sql_identifier	包含该列排序规则的模式名称；如果使用默认排序规则，或该列的数据类型不支持排序规则，则为 null。
collation_name	sql_identifier	该列排序规则的名称；如果使用默认排序规则，或该列的数据类型不支持排序规则，则为 null。
domain_catalog	sql_identifier	如果该列有一个域类型，这里是该域所在的数据库名（总是当前数据库），否则为 null。
domain_schema	sql_identifier	如果该列有一个域类型，这里是该域所在的模式名，否则为 null。
domain_name	sql_identifier	如果该列有一个域类型，这里是该域的名称，否则为 null。
udt_catalog	sql_identifier	该列数据类型（如果适用，则为域的底层类型）所在的数据库名称（总是当前数据库）
udt_schema	sql_identifier	该列数据类型（如果适用，则为域的底层类型）所在的模式名称
udt_name	sql_identifier	该列数据类型（如果适用，则为域的底层类型）的名称
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
maximum_cardinality	cardinal_number	总是为 null，因为 PostgreSQL 中数组的最大基数始终不受限制
dtd_identifier	sql_identifier	该列的数据类型描述符的标识符，在属于同一表的数据类型描述符中是唯一的。它主要用于与其他此类标识符实例进行连接。（标识符的具体格式没有定义，也不保证在未来版本中保持不变。）
is_self_referencing	yes_or_no	对应 PostgreSQL 不支持的特性
is_identity	yes_or_no	如果该列是一个标识列，则为YES，否则为NO。
identity_generation	character_data	如果该列是一个标识列，则为ALWAYS 或者BY DEFAULT，它反映该列的定义。
identity_start	character_data	如果该列是一个标识列，则是内部序列的起始值，否则为 null。
identity_increment	character_data	如果该列是一个标识列，则是内部序列的增量，否则为 null。
identity_maximum	character_data	如果该列是一个标识列，则是内部序列的最大值，否则为 null。
identity_minimum	character_data	如果该列是一个标识列，则是内部序列的最小值，否则为 null。
identity_cycle	yes_or_no	如果该列是一个标识列，则YES 和NO 分别表示内部序列可循环和不可循环，否则为 null。

名称	数据类型	描述
is_generated	character_data	对应 PostgreSQL 不支持的特性
generation_expression	character_data	对应 PostgreSQL 不支持的特性
is_updatable	yes_or_no	如果该列是可更新的，则为YES，否则为NO（基表中的列总是可更新的，视图中的列则不一定）

因为在 SQL 中有多种方式定义数据类型，而PostgreSQL 还包含额外的方式来定义数据类型，它们在信息模式中的表示可能有点困难。列data_type 应该标识列的底层内置类型。在 PostgreSQL 中，这表示定义在系统目录模式pg_catalog 中的类型。如果应用能够特别地（例如以不同方式格式化数值类型或使用精度列中的数据）处理众所周知的内置类型，这列可能会有用。列udt_name、udt_schema 和 udt_catalog 总是标识列的底层数据类型，即使该列是基于一个域的（因为PostgreSQL 对待内置类型和用户定义类型的方式是一样的，内置类型也出现在这里。这是 SQL 标准的一种扩展）。如果一个应用想要根据该类型以不同的方式处理数据，就应该使用这些列，因为在那种情况下即使该列确实基于一个域也没有关系。如果该列是基于一个域，该域的标识被存储在列domain_name、domain_schema 和 domain_catalog。如果你想要把列和它们相关的数据类型配对并且把域视作单独的类型，你可以写coalesce(domain_name, udt_name) 等等。

37.17. constraint_column_usage

视图 constraint_column_usage 标识当前数据库中被某种约束使用的所有列。只显示那些位于当前启用角色所拥有表中的列。对于检查约束，此视图标识检查表达式中使用的列。对于外键约束，此视图标识外键所引用的列。对于唯一约束或主键约束，此视图标识受该约束限制的列。

表 37.15. constraint_column_usage 列

名称	数据类型	描述
table_catalog	sql_identifier	包含被某个约束使用的列的表所在的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含被某个约束使用的列的表所在的模式名称
table_name	sql_identifier	包含被某个约束使用的列的表名称
column_name	sql_identifier	被某个约束使用的列名称
constraint_catalog	sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名称
constraint_name	sql_identifier	约束的名称

37.18. constraint_table_usage

视图 constraint_table_usage 标识当前数据库中被某个约束使用、且由当前已启用角色拥有的所有表。（这与视图table_constraints 不同，后者标识所有表约束及其所在的表。）对于外键约束，此视图标识外键所引用的表。对于唯一约束或主键约束，此视图仅标识约束所属的表。此视图不包含检查约束和非空约束。

表 37.16. constraint_table_usage 列

名称	数据类型	描述
table_catalog	sql_identifier	包含被某个约束使用的表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含被某个约束使用的表的模式名称
table_name	sql_identifier	被某个约束使用的表名称
constraint_catalog	sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名称
constraint_name	sql_identifier	约束的名称

37.19. data_type_privileges

视图 data_type_privileges 标识当前用户能够访问的所有数据类型描述符，访问权限来自对所描述对象的拥有权或该对象上的某种权限。只要在表列、域或函数的定义中使用一个数据类型（在函数中用作参数类型或返回类型），就会生成一个数据类型描述符，用来存储该数据类型在这次使用中的相关信息（例如声明的最大长度，如果适用）。每个数据类型描述符都会被赋予一个任意的标识符，该标识符在分配给同一对象（表、域或函数）的数据类型描述符标识符中是唯一的。此视图对应用程序可能没有什么用处，但会用于定义信息模式中的其他一些视图。

表 37.17. data_type_privileges 列

名称	数据类型	描述
object_catalog	sql_identifier	包含该被描述对象的数据库名称（总是当前数据库）
object_schema	sql_identifier	包含该被描述对象的模式名称
object_name	sql_identifier	被描述对象的名称
object_type	character_data	被描述对象的类型：TABLE（从属于表的一列的数据类型描述符）、DOMAIN（从属于域的数据类型描述符）、ROUTINE（从属于函数的一个参数或返回数据类型的数据类型描述符）。
dtd_identifier	sql_identifier	数据类型描述符的标识符，它在同一对象的数据类型描述符之间唯一。

37.20. domain_constraints

视图 domain_constraints 包含所有属于当前数据库中定义的域的约束。只有当前用户能访问的那些域才被显示（作为拥有者或具有某些权限）。

表 37.18. domain_constraints 列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名称

名称	数据类型	描述
constraint_name	sql_identifier	约束的名称
domain_catalog	sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名称
domain_name	sql_identifier	域名称
is_deferrable	yes_or_no	如果该约束是可延迟的，则为YES，否则为NO
initially_deferred	yes_or_no	如果该约束是可延迟的且初始就被延迟，则为YES，否则为NO

37.21. domain_udt_usage

视图 domain_udt_usage 标识所有基于被一个当前启用的角色拥有的数据类型的域。注意在 PostgreSQL 中，内置数据类型的行为相似于用户定义的类型，因此它们也被包括在这里。

表 37.19. domain_udt_usage 列

名称	数据类型	描述
udt_catalog	sql_identifier	该域数据类型被定义的数据库名称（总是当前数据库）
udt_schema	sql_identifier	该域数据类型被定义的模式名称
udt_name	sql_identifier	该域数据类型的名称
domain_catalog	sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名称
domain_name	sql_identifier	域名称

37.22. domains

视图 domains 包含当前数据库中定义的所有域。仅显示当前用户有权限访问的域（作为拥有者或具有某种权限）。

表 37.20. domains 列

名称	数据类型	描述
domain_catalog	sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名称
domain_name	sql_identifier	域名称
data_type	character_data	该域的数据类型如果是一种内置类型，这里是该域的数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在udt_name 和相关列中）。

名称	数据类型	描述
character_maximum_length	cardinal_number	如果该域的数据类型是字符类型或位串类型，则为声明的最大长度；对于其他所有数据类型，或未声明最大长度时，则为 null。
character_octet_length	cardinal_number	如果该域有一个字符类型，这里是一个数据的最大可能长度（以字节计）；对其他所有数据类型为 null。最大字节长度取决于声明的字符最大长度（见上文）和服务器编码。
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	包含该域排序规则的数据库名称（总是当前数据库）；如果使用默认排序规则，或该域的数据类型不支持排序规则，则为 null。
collation_schema	sql_identifier	包含该域排序规则的模式名称；如果使用默认排序规则，或该域的数据类型不支持排序规则，则为 null。
collation_name	sql_identifier	该域排序规则的名称；如果使用默认排序规则，或该域的数据类型不支持排序规则，则为 null。
numeric_precision	cardinal_number	如果该域具有数值类型，此列包含该域数据类型的（声明的或隐式的）精度。精度表示有效位数。它可以按十进制（基数 10）或二进制（基数 2）表示，具体由 numeric_precision_radix 列指定。对于其他所有数据类型，此列为 null。
numeric_precision_radix	cardinal_number	如果该域具有数值类型，此列指示 numeric_precision 和numeric_scale 列中的值采用的进制。值为 2 或 10。对于其他所有数据类型，此列为 null。
numeric_scale	cardinal_number	如果该域具有精确数值类型，此列包含该域数据类型的（声明的或隐式的）小数位数。小数位数表示小数点右侧的有效位数。它可以按十进制（基数 10）或二进制（基数 2）表示，具体由numeric_precision_radix 列指定。对于其他所有数据类型，此列为 null。
datetime_precision	cardinal_number	如果data_type 标识日期、时间、时间戳或时间间隔类型，此列包含该域数据类型的（声明的或隐式的）小数秒精度，即秒值小数点后保留的十进制位数。对于其他所有数据类型，此列为 null。

名称	数据类型	描述
interval_type	character_data	如果data_type 标识时间间隔类型，此列说明该域的时间间隔包含哪些字段，例如YEAR TO MONTH、DAY TO SECOND 等。未指定字段限制时（即时间间隔接受所有字段），或对于其他所有数据类型，此列为 null。
interval_precision	cardinal_number	对应PostgreSQL不支持的特性（关于时间间隔类型域的小数秒精度，参见datetime_precision）
domain_default	character_data	该域的默认表达式
udt_catalog	sql_identifier	该域数据类型被定义的数据库名称（总是当前数据库）
udt_schema	sql_identifier	该域数据类型被定义的模式名称
udt_name	sql_identifier	该域数据类型的名称
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
maximum_cardinality	cardinal_number	总是为 null，因为 PostgreSQL 中数组的最大基数始终不受限制
dtd_identifier	sql_identifier	该域的数据类型描述符的标识符，在属于同一域的数据类型描述符中是唯一的（这一点显然成立，因为一个域只包含一个数据类型描述符）。它主要用于与其他此类标识符实例进行连接。（标识符的具体格式没有定义，也不保证在未来版本中保持不变。）

37.23. element_types

视图 element_types 包含数组元素的数据类型描述符。当一个表列、复合类型属性、域、函数参数或函数返回值被定义为一种数组类型时，相应的信息模式视图只在列data_type 中包含 ARRAY。要获得该数组元素类型的信息，你可以连接该相应的视图和这个视图。例如，要显示一个表的列及其数据类型和数组元素类型（如果适用），你可以：

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_type, e.collection_type_
identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

这个视图只包括当前用户能够访问（作为拥有者或具有某些权限）的对象。

表 37.21. element_types 列

名称	数据类型	描述
object_catalog	sql_identifier	包含使用被描述的数组的对象的数据库名称（总是当前数据库）

名称	数据类型	描述
object_schema	sql_identifier	包含使用被描述的数组的对象的模式名称
object_name	sql_identifier	使用所描述数组的对象名称
object_type	character_data	使用被描述的数组的对象的类型：TABLE（被一个表列使用的数组）、USER-DEFINED TYPE（被复合类型的一个属性使用的数组）、DOMAIN（被域使用的数组）、ROUTINE（被函数的一个参数或返回数据类型使用的数组）。
collection_type_identifier	sql_identifier	被描述的数组的数据类型描述符的标识符。使用这个去与其他信息模式视图的dtd_identifier 列连接。
data_type	character_data	如果数组元素的数据类型是内置类型，这里是数组元素的数据类型，否则为USER-DEFINED（在那种情况下，该类型被标识在udt_name 和相关列中）。
character_maximum_length	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的数组元素数据类型
character_octet_length	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的数组元素数据类型
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	包含该元素类型排序规则的数据库名称（总是当前数据库）；如果使用默认排序规则，或该元素类型不支持排序规则，则为 null。
collation_schema	sql_identifier	包含该元素类型排序规则的模式名称；如果使用默认排序规则，或该元素类型不支持排序规则，则为 null。
collation_name	sql_identifier	该元素类型排序规则的名称；如果使用默认排序规则，或该元素类型不支持排序规则，则为 null。
numeric_precision	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的数组元素数据类型
numeric_precision_radix	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的数组元素数据类型
numeric_scale	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的数组元素数据类型
datetime_precision	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的数组元素数据类型

名称	数据类型	描述
interval_type	character_data	总是为 null，因为这种信息不适用于PostgreSQL中的数组元素数据类型
interval_precision	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的数组元素数据类型
domain_default	character_data	尚未实现
udt_catalog	sql_identifier	元素的数据类型所在的数据库名称（总是当前数据库）
udt_schema	sql_identifier	定义元素的数据类型的模式的名称
udt_name	sql_identifier	元素的数据类型名
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
maximum_cardinality	cardinal_number	总是为 null，因为 PostgreSQL 中数组的最大基数始终不受限制
dtd_identifier	sql_identifier	该元素的数据类型描述符的标识符。当前没有用。

37.24. enabled_roles

视图 enabled_roles 标识当前“已启用的角色”。已启用的角色递归地定义为：当前用户，以及所有通过自动继承方式授予给已启用角色的角色。换句话说，这些角色都是当前用户直接或间接自动继承其成员资格的角色。

对于权限检查，使用的是一组“适用角色”，其范围可能比已启用角色的集合更广。因此，一般来说最好使用视图applicable_roles 而不是这个视图；有关 applicable_roles 视图的详细信息，见第 37.5 节。

表 37.22. enabled_roles 列

名称	数据类型	描述
role_name	sql_identifier	一个角色的名称

37.25. foreign_data_wrapper_options

视图 foreign_data_wrapper_options 包含为当前数据库中外部数据包装器定义的所有选项。只有那些当前用户能够访问（作为拥有者或具有某些权限）的外部数据包装器被显示。

表 37.23. foreign_data_wrapper_options 列

名称	数据类型	描述
foreign_data_wrapper_catalog	sql_identifier	该外部数据包装器所在的数据库名称（总是当前数据库）
foreign_data_wrapper_name	sql_identifier	该外部数据包装器的名称
option_name	sql_identifier	一个选项名称
option_value	character_data	该选项的值

37.26. foreign_data_wrappers

视图 foreign_data_wrappers 包含定义在当前数据库中的所有外部数据包装器。只有那些当前用户能够访问（作为拥有者或具有某些权限）的外部数据包装器才会被显示。

表 37.24. foreign_data_wrappers 列

名称	数据类型	描述
foreign_data_wrapper_catalog	sql_identifier	包含该外部数据包装器的数据库名称（总是当前数据库）
foreign_data_wrapper_name	sql_identifier	该外部数据包装器的名称
authorization_identifier	sql_identifier	外部服务器拥有者的名称
library_name	character_data	实现这个外部数据包装器的库文件名称
foreign_data_wrapper_language	character_data	用于实现这个外部数据包装器的语言

37.27. foreign_server_options

视图 foreign_server_options 包含为当前数据库中外部服务器定义的所有选项。只有那些当前用户能够访问（作为拥有者或具有某些权限）的外部服务器才会被显示。

表 37.25. foreign_server_options 列

名称	数据类型	描述
foreign_server_catalog	sql_identifier	该外部服务器所在的数据库名称（总是当前数据库）
foreign_server_name	sql_identifier	该外部服务器的名称
option_name	sql_identifier	一个选项名称
option_value	character_data	该选项的值

37.28. foreign_servers

视图 foreign_servers 包含当前数据库中定义的所有外部服务器。只有那些当前用户能够访问（作为拥有者或具有某些权限）的外部服务器才会被显示。

表 37.26. foreign_servers 列

名称	数据类型	描述
foreign_server_catalog	sql_identifier	该外部服务器所在的数据库名称（总是当前数据库）
foreign_server_name	sql_identifier	该外部服务器的名称
foreign_data_wrapper_catalog	sql_identifier	包含被该外部服务器使用的外部数据包装器的数据库名称（总是当前数据库）
foreign_data_wrapper_name	sql_identifier	被该外部服务器所使用的外部数据包装器的名称
foreign_server_type	character_data	外部服务器类型信息，如果在创建时指定过

名称	数据类型	描述
foreign_server_version	character_data	外部服务器版本信息，如果在创建时指定过
authorization_identifier	sql_identifier	外部服务器拥有者的名称

37.29. foreign_table_options

视图 foreign_table_options 包含为当前数据库中外部表定义的所有选项。只有那些当前用户能够访问（作为拥有者或具有某些权限）的外部表才会被显示。

表 37.27. foreign_table_options 列

名称	数据类型	描述
foreign_table_catalog	sql_identifier	包含该外部表的数据库名称（总是当前数据库）
foreign_table_schema	sql_identifier	包含该外部表的模式名称
foreign_table_name	sql_identifier	外部表的名称
option_name	sql_identifier	一个选项名称
option_value	character_data	该选项的值

37.30. foreign_tables

视图 foreign_tables 包含定义在当前数据库中的所有外部表。只有那些当前用户能够访问（作为拥有者或具有某些权限）的外部表才会被显示。

表 37.28. foreign_tables 列

名称	数据类型	描述
foreign_table_catalog	sql_identifier	该外部表所在的数据库名称（总是当前数据库）
foreign_table_schema	sql_identifier	包含该外部表的模式名称
foreign_table_name	sql_identifier	外部表的名称
foreign_server_catalog	sql_identifier	该外部服务器所在的数据库名称（总是当前数据库）
foreign_server_name	sql_identifier	该外部服务器的名称

37.31. key_column_usage

视图 key_column_usage 标识当前数据库中所有被某种唯一、主键或外键约束限制的列。检查约束不被包括在这个视图中。只有那些当前用户能够访问的列才会被显示（作为拥有者或具有某些权限）。

表 37.29. key_column_usage 列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名称
constraint_name	sql_identifier	约束的名称

名称	数据类型	描述
table_catalog	sql_identifier	包含被这个约束限制的列的表所在的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含被这个约束限制的列的表所在的模式名称
table_name	sql_identifier	包含被这个约束限制的列的表的名称
column_name	sql_identifier	被这个约束限制的列的名称
ordinal_position	cardinal_number	该列在约束键中的顺序位置（从 1 开始计数）
position_in_unique_constraint	cardinal_number	对于外键约束，为被引用列在其唯一约束中的顺序位置（从 1 开始计数）；对于其他约束则为 null

37.32. parameters

视图 parameters 包含当前数据库中所有函数的参数的有关信息。只有那些当前用户能够访问（作为拥有者或具有某些权限）的函数才会被显示。

表 37.30. parameters 列

名称	数据类型	描述
specific_catalog	sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	函数的“特定名称”。详见第 37.40 节。
ordinal_position	cardinal_number	该参数在函数参数列表中的顺序位置（从 1 开始计数）
parameter_mode	character_data	IN 表示输入参数，OUT 表示输出参数，INOUT 表示输入/输出参数。
is_result	yes_or_no	对应 PostgreSQL 不支持的特性
as_locator	yes_or_no	对应 PostgreSQL 不支持的特性
parameter_name	sql_identifier	参数名称，如果参数没有名称则为 null
data_type	character_data	该参数的数据类型如果是一种内置类型，这里是该参数的数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在 udt_name 和相关列中）。
character_maximum_length	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
character_octet_length	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性

名称	数据类型	描述
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
collation_schema	sql_identifier	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
collation_name	sql_identifier	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
numeric_precision	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
numeric_precision_radix	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
numeric_scale	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
datetime_precision	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
interval_type	character_data	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
interval_precision	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的参数数据类型
udt_catalog	sql_identifier	该参数的数据类型所在的数据库名称（总是当前数据库）
udt_schema	sql_identifier	该参数的数据类型所在的模式名称
udt_name	sql_identifier	该参数的数据类型的名称
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
maximum_cardinality	cardinal_number	总是为 null，因为 PostgreSQL 中数组的最大基数始终不受限制
dtd_identifier	sql_identifier	该参数的数据类型描述符的标识符，在属于同一函数的数据类型描述符中是唯一的。它主要用于与其他此类标识符实例进行连接。（标识符的具体格式没有定义，也不保证在未来版本中保持不变。）
parameter_default	character_data	该参数的默认表达式，如果没有或者该函数不被一个当前启用的角色拥有则为 null。

37.33. referential_constraints

视图 referential_constraints 包含当前数据库中的所有引用（外键）约束。只显示当前用户对引用方的表具有写入权限的约束（作为表的拥有者，或具有SELECT 之外的某种权限）。

表 37.31. referential_constraints 列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含约束的数据库名称（总是当前数据库）

名称	数据类型	描述
constraint_schema	sql_identifier	包含约束的模式的名称
constraint_name	sql_identifier	约束的名称
unique_constraint_catalog	sql_identifier	包含该外键约束所引用的唯一或主键约束的数据库名称（总是当前数据库）
unique_constraint_schema	sql_identifier	包含该外键约束所引用的唯一或主键约束的模式名称
unique_constraint_name	sql_identifier	该外键约束所引用的唯一约束或主键约束的名称
match_option	character_data	外键约束的匹配选项：FULL、PARTIAL 或NONE。
update_rule	character_data	外键约束的更新规则：CASCADE、SET NULL、SET DEFAULT、RESTRICT 或 NO ACTION。
delete_rule	character_data	外键约束的删除规则：CASCADE、SET NULL、SET DEFAULT、RESTRICT 或 NO ACTION。

37.34. role_column_grants

视图 role_column_grants 标识所有在列上授予的权限，这些权限的授予者或者被授权者是当前已启用的角色。更多信息可以在column_privileges 中找到。这个视图和 column_privileges 之间的唯一实质性区别是：这个视图忽略那些以授予给 PUBLIC 的方式使当前用户获得其访问权限的列。

表 37.32. role_column_grants 列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的角色名称
grantee	sql_identifier	被授予该权限的角色名称
table_catalog	sql_identifier	包含该列的表所在的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该列的表所在的模式名称
table_name	sql_identifier	包含该列的表名称
column_name	sql_identifier	列名称
privilege_type	character_data	权限类型：SELECT、INSERT、UPDATE 或 REFERENCES
is_grantable	yes_or_no	如果该权限可再授予，则为YES，否则为NO

37.35. role_routine_grants

视图 role_routine_grants 标识所有在函数上授予的权限，这些权限的授予者或者被授权者是当前已启用的角色。更多信息可以在routine_privileges 中找到。这个视图和 routine_privileges 之间的唯一实质性区别是：这个视图忽略那些以授予给 PUBLIC 的方式使当前用户获得其访问权限的函数。

表 37.33. role_routine_grants 列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的角色名称
grantee	sql_identifier	被授予该权限的角色名称
specific_catalog	sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	函数的“特定名称”。详见第 37.40 节。
routine_catalog	sql_identifier	包含该函数的数据库名称（总是当前数据库）
routine_schema	sql_identifier	包含该函数的模式名称
routine_name	sql_identifier	该函数的名称（在重载的情况下可能会重复）
privilege_type	character_data	总是为EXECUTE（函数唯一的权限类型）
is_grantable	yes_or_no	如果该权限可再授予，则为YES，否则为NO

37.36. role_table_grants

视图 role_table_grants 标识所有在表或视图上授予的权限，这些权限的授予者或者被授权者是当前已启用的角色。更多信息可以在table_privileges 中找到。这个视图和 table_privileges 之间的唯一实质性区别是：这个视图忽略那些以授予给 PUBLIC 的方式使当前用户获得其访问权限的表。

表 37.34. role_table_grants 列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的角色名称
grantee	sql_identifier	被授予该权限的角色名称
table_catalog	sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名称
table_name	sql_identifier	表的名称
privilege_type	character_data	该权限的类型：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES 或 TRIGGER
is_grantable	yes_or_no	如果该权限可再授予，则为YES，否则为NO
with_hierarchy	yes_or_no	在 SQL 标准中，WITH HIERARCHY OPTION 是一个独立的（子）权限，它允许在表继承层级上的特定操作。在 PostgreSQL 中，这被包括在SELECT 权限中，因此这一列在权限为SELECT 时显示YES，其他时候显示NO。

37.37. role_udt_grants

视图 role_udt_grants 旨在标识在用户定义类型上授予的 USAGE 权限，这些权限的授予者或者被授权者是当前已启用的角色。更多信息可以在udt_privileges 中找到。这个视图和 udt_privileges 之间的唯一实质性区别是：这个视图忽略那些以授予给 PUBLIC 的方式使当前用户获得其访问权限的对象。因为数据类型在 PostgreSQL 中并没有真正的权限，而是只有一个给PUBLIC 的隐式授予，这个视图为空。

表 37.35. role_udt_grants 列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的角色名称
grantee	sql_identifier	被授予该权限的角色名称
udt_catalog	sql_identifier	包含该类型的数据库名称（总是当前数据库）
udt_schema	sql_identifier	包含该类型的模式名称
udt_name	sql_identifier	类型的名称
privilege_type	character_data	总是TYPE USAGE
is_grantable	yes_or_no	如果该权限可再授予，则为YES，否则为NO

37.38. role_usage_grants

视图 role_usage_grants 标识所有在多种对象上授予的 USAGE 权限，这些权限的授予者或者被授权者是当前已启用的角色。更多信息可以在usage_privileges 中找到。这个视图和 usage_privileges 之间的唯一实质性区别是：这个视图忽略那些以授予给 PUBLIC 的方式使当前用户获得其访问权限的对象。

表 37.36. role_usage_grants 列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的角色名称
grantee	sql_identifier	被授予该权限的角色名称
object_catalog	sql_identifier	包含该对象的数据库名称（总是当前数据库）
object_schema	sql_identifier	如果适用，则为包含该对象的模式名称，否则为一个空字符串
object_name	sql_identifier	对象的名称
object_type	character_data	COLLATION 或DOMAIN 或 FOREIGN DATA WRAPPER 或 FOREIGN SERVER 或SEQUENCE
privilege_type	character_data	总是USAGE
is_grantable	yes_or_no	如果该权限可再授予，则为YES，否则为NO

37.39. routine_privileges

视图 routine_privileges 标识所有在函数上授予的权限，其授予者或被授权者是当前已启用的角色。对于每一种函数、授予者和被授权者的组合，这里都有一行。

表 37.37. routine_privileges 列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的角色名称
grantee	sql_identifier	被授予该权限的角色名称
specific_catalog	sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	函数的“特定名称”。详见第 37.40 节。
routine_catalog	sql_identifier	包含该函数的数据库名称（总是当前数据库）
routine_schema	sql_identifier	包含该函数的模式名称
routine_name	sql_identifier	该函数的名称（在重载的情况下可能会重复）
privilege_type	character_data	总是为EXECUTE（函数唯一的权限类型）
is_grantable	yes_or_no	如果该权限可再授予，则为YES，否则为NO

37.40. routines

视图 routines 包含当前数据库中所有的函数和过程。只有那些当前用户能够访问（作为拥有者或具有某些权限）的函数和过程才会被显示。

表 37.38. routines 列

名称	数据类型	描述
specific_catalog	sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	该函数的“特定名称”。这个名称在模式中唯一标识该函数，即使函数的实际名称被重载也是如此。特定名称的格式没有定义，应当仅用于与其他例程特定名称的实例进行比较。
routine_catalog	sql_identifier	包含该函数的数据库名称（总是当前数据库）
routine_schema	sql_identifier	包含该函数的模式名称
routine_name	sql_identifier	该函数的名称（在重载的情况下可能会重复）
routine_type	character_data	FUNCTION 表示是一个函数，PROCEDURE 表示是一个过程
module_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
module_schema	sql_identifier	对应 PostgreSQL 不支持的特性
module_name	sql_identifier	对应 PostgreSQL 不支持的特性
udt_catalog	sql_identifier	对应 PostgreSQL 不支持的特性

名称	数据类型	描述
udt_schema	sql_identifier	对应 PostgreSQL 不支持的特性
udt_name	sql_identifier	对应 PostgreSQL 不支持的特性
data_type	character_data	该函数的返回数据类型如果是一种内置类型，这里是该数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在type_udt_name 和相关列中）。如果是过程则此项为 null。
character_maximum_length	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
character_octet_length	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
collation_schema	sql_identifier	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
collation_name	sql_identifier	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
numeric_precision	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
numeric_precision_radix	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
numeric_scale	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
datetime_precision	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
interval_type	character_data	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
interval_precision	cardinal_number	总是为 null，因为这种信息不适用于PostgreSQL中的返回数据类型
type_udt_catalog	sql_identifier	该函数的返回数据类型所在的数据库名（总是当前数据库）。如果是过程则此项为 null。
type_udt_schema	sql_identifier	该函数的返回数据类型所在的模式名。如果是过程则此项为 null。
type_udt_name	sql_identifier	该函数的返回数据类型的名字。如果是过程则此项为 null。
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
maximum_cardinality	cardinal_number	总是为 null，因为 PostgreSQL 中数组的最大基数始终不受限制

名称	数据类型	描述
dtd_identifier	sql_identifier	此函数返回数据类型的数据类型描述符的标识符，在属于同一函数的数据类型描述符中是唯一的。它主要用于与其他此类标识符实例进行连接。（标识符的具体格式没有定义，也不保证在未来版本中保持不变。）
routine_body	character_data	如果该函数是一个 SQL 函数，则为SQL，否则为EXTERNAL。
routine_definition	character_data	该函数的源文本（如果该函数不归当前已启用的角色所有，则为 null）（根据 SQL 标准，只有 routine_body 为SQL 时这一列才适用。但是在PostgreSQL中，它将会包含该函数被创建时所指定的任何源文本。）
external_name	character_data	如果这个函数是一个 C 函数，则为该函数的外部名称（链接符号），否则为 null（这会产生和显示在 routine_definition 中相同的值）。
external_language	character_data	该函数所用的语言
parameter_style	character_data	总是GENERAL（SQL 标准定义了其他参数风格，但在PostgreSQL中不可用）
is_deterministic	yes_or_no	如果该函数被声明为不可变（在 SQL 标准中被称为确定性的），则为YES，否则为NO（你不能通过该信息模式查询在PostgreSQL中可用的其他易变性级别）。
sql_data_access	character_data	总是MODIFIES，表示该函数可能修改 SQL 数据。这种信息对PostgreSQL没有用处。
is_null_call	yes_or_no	如果该函数在任一参数为 null 时自动返回 null，则为YES，否则为NO。如果是过程则此项为 null。
sql_path	character_data	对应 PostgreSQL 不支持的特性
schema_level_routine	yes_or_no	总是YES（相反的情况是用户定义类型的方法，而PostgreSQL不支持这种特性）。
max_dynamic_result_sets	cardinal_number	对应 PostgreSQL 不支持的特性
is_user_defined_cast	yes_or_no	对应 PostgreSQL 不支持的特性
is_implicitly_invocable	yes_or_no	对应 PostgreSQL 不支持的特性
security_type	character_data	如果该函数以当前用户的权限运行，则为INVOKER；如果该函数以定义它的用户的权限运行，则为DEFINER。
to_sql_specific_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
to_sql_specific_schema	sql_identifier	对应 PostgreSQL 不支持的特性
to_sql_specific_name	sql_identifier	对应 PostgreSQL 不支持的特性

名称	数据类型	描述
as_locator	yes_or_no	对应 PostgreSQL 不支持的特性
created	time_stamp	对应 PostgreSQL 不支持的特性
last_altered	time_stamp	对应 PostgreSQL 不支持的特性
new_savepoint_level	yes_or_no	对应 PostgreSQL 不支持的特性
is_udt_dependent	yes_or_no	当前总是NO。另一个选项YES 对应 PostgreSQL 不支持的特性。
result_cast_from_data_type	character_data	对应 PostgreSQL 不支持的特性
result_cast_as_locator	yes_or_no	对应 PostgreSQL 不支持的特性
result_cast_char_max_length	cardinal_number	对应 PostgreSQL 不支持的特性
result_cast_char_octet_length	character_data	对应 PostgreSQL 不支持的特性
result_cast_char_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_char_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_char_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_collation_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_collation_schema	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_collation_name	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_numeric_precision	cardinal_number	对应 PostgreSQL 不支持的特性
result_cast_numeric_precision_radix	cardinal_number	对应 PostgreSQL 不支持的特性
result_cast_numeric_scale	cardinal_number	对应 PostgreSQL 不支持的特性
result_cast_datetime_precision	character_data	对应 PostgreSQL 不支持的特性
result_cast_interval_type	character_data	对应 PostgreSQL 不支持的特性
result_cast_interval_precision	cardinal_number	对应 PostgreSQL 不支持的特性
result_cast_type_udt_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_type_udt_schema	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_type_udt_name	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
result_cast_maximum_cardinality	cardinal_number	对应 PostgreSQL 不支持的特性

名称	数据类型	描述
result_cast_dtd_identifier	sql_identifier	对应 PostgreSQL 不支持的特性

37.41. schemata

视图 schemata 包含当前数据库中当前用户能够访问的所有模式（作为拥有者或具有某种权限）。

表 37.39. schemata 列

名称	数据类型	描述
catalog_name	sql_identifier	该模式所在的数据库名称（总是当前数据库）
schema_name	sql_identifier	模式的名称
schema_owner	sql_identifier	模式拥有者的名称
default_character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
default_character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
default_character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
sql_path	character_data	对应 PostgreSQL 不支持的特性

37.42. sequences

视图 sequences 包含所有定义在当前数据库中的序列。只有那些当前用户能够访问（作为拥有者或具有某些权限）的序列才会被显示。

表 37.40. sequences 列

名称	数据类型	描述
sequence_catalog	sql_identifier	包含该序列的数据库名称（总是当前数据库）
sequence_schema	sql_identifier	包含该序列的模式名称
sequence_name	sql_identifier	序列的名称
data_type	character_data	序列的数据类型
numeric_precision	cardinal_number	此列包含该序列数据类型（见上文）的（声明的或隐式的）精度。精度表示有效位数。它可以按十进制（基数 10）或二进制（基数 2）表示，具体由numeric_precision_radix 列指定。
numeric_precision_radix	cardinal_number	此列指示numeric_precision 和numeric_scale 列中的值采用的进制。值为 2 或 10。

名称	数据类型	描述
numeric_scale	cardinal_number	此列包含该序列数据类型（见上文）的（声明的或隐式的）小数位数。小数位数表示小数点右侧的有效位数。它可以按十进制（基数 10）或二进制（基数 2）表示，具体由numeric_precision_radix 列指定。
start_value	character_data	该序列的开始值
minimum_value	character_data	该序列的最小值
maximum_value	character_data	该序列的最大值
increment	character_data	该序列的增量
cycle_option	yes_or_no	如果该序列会循环，则为YES，否则为NO

注意依照 SQL 标准，开始值、最小值、最大值和增量值被作为字符串返回。

37.43. sql_features

表 sql_features 包含 PostgreSQL 支持 SQL 标准中哪些正式特性的信息。这与附录 D 提供的信息相同，那里还有一些额外的背景信息。

表 37.41. sql_features 列

名称	数据类型	描述
feature_id	character_data	特性的标识符字符串
feature_name	character_data	特性的描述性名称
sub_feature_id	character_data	该子特性的标识符字符串，或者如果不是一个子特性则为一个长度为零的字符串
sub_feature_name	character_data	该子特性的描述性名称，或者如果不是一个子特性则为一个长度为零的字符串
is_supported	yes_or_no	如果当前版本的 PostgreSQL 完全支持该特性，则为YES，否则为NO
is_verified_by	character_data	总是为 null，因为PostgreSQL开发组没有对特性的标准符合性进行正式测试
comments	character_data	可能会是关于该特性被支持状态的一段注释

37.44. sql_implementation_info

表 sql_implementation_info 包含 SQL 标准留给具体实现来定义的各个方面的信息。这些信息主要用于 ODBC 接口；使用其他接口的用户可能会觉得用处不大。因此，这里不描述各个实现信息项，其说明可以在 ODBC 接口的文档中找到。

表 37.42. sql_implementation_info 列

名称	数据类型	描述
implementation_info_id	character_data	实现信息项的标识符字符串
implementation_info_name	character_data	实现信息项的描述性名称
integer_value	cardinal_number	实现信息项的整数值；如果该值在列character_value 中给出，则本列为 null
character_value	character_data	实现信息项的字符值；如果该值在列integer_value 中给出，则本列为 null
comments	character_data	可能是从属于该实现信息项的一段注释

37.45. sql_languages

表 sql_languages 为 PostgreSQL 支持的每一种 SQL 语言绑定包含一行。PostgreSQL 支持直接 SQL 和 C 中的嵌入式 SQL；从此表中只能了解到这些信息。

此表已在 SQL:2008 中从 SQL 标准移除，因此没有引用 SQL:2003 之后标准的条目。

表 37.43. sql_languages 列

名称	数据类型	描述
sql_language_source	character_data	语言定义来源的名称；总是ISO 9075，即 SQL 标准
sql_language_year	character_data	sql_language_source 引用的标准获批的年份。
sql_language_conformance	character_data	该语言绑定的标准符合性级别。对于 ISO 9075:2003，此值总是CORE。
sql_language_integrity	character_data	总是为 null（此值与 SQL 标准的较早版本有关。）
sql_language_implementation	character_data	总是为 null
sql_language_binding_style	character_data	语言绑定方式，为DIRECT 或EMBEDDED
sql_language_programming_language	character_data	如果绑定方式为EMBEDDED，则为所用编程语言，否则为 null。PostgreSQL只支持 C 语言。

37.46. sql_packages

表 sql_packages 包含 PostgreSQL 支持 SQL 标准所定义的哪些功能包的信息。功能包的背景信息请参阅附录 D。

表 37.44. sql_packages 列

名称	数据类型	描述
feature_id	character_data	功能包的标识符字符串

名称	数据类型	描述
feature_name	character_data	功能包的描述性名称
is_supported	yes_or_no	如果当前版本的PostgreSQL完全支持此功能包，则为YES，否则为NO
is_verified_by	character_data	总是为 null，因为PostgreSQL开发组没有对特性的标准符合性进行正式测试
comments	character_data	关于此功能包支持状态的可选注释

37.47. sql_parts

表 sql_parts 包含 PostgreSQL 支持 SQL 标准中哪些部分的信息。

表 37.45. sql_parts 列

名称	数据类型	描述
feature_id	character_data	包含该部分编号的一个标识符字符串
feature_name	character_data	该部分的描述性名称
is_supported	yes_or_no	如果当前版本的 PostgreSQL 完全支持该部分，则为YES，否则为NO
is_verified_by	character_data	总是为 null，因为PostgreSQL开发组没有对特性的标准符合性进行正式测试
comments	character_data	可能会是关于该部分被支持状态的一段注释

37.48. sql_sizing

表 sql_sizing 包含 PostgreSQL 中各种尺寸限制和最大值的信息。这些信息主要用于 ODBC 接口；使用其他接口的用户可能会觉得用处不大。因此，这里不描述各个尺寸项，其说明可以在 ODBC 接口的文档中找到。

表 37.46. sql_sizing 列

名称	数据类型	描述
sizing_id	cardinal_number	尺寸项的标识符
sizing_name	character_data	尺寸项的描述性名称
supported_value	cardinal_number	尺寸项的值，如果尺寸是不受限制或不能确定的则为 0，如果尺寸项适用的特性不受支持则为 null
comments	character_data	可能是从属于尺寸项的一段注释

37.49. sql_sizing_profiles

表 sql_sizing_profiles 包含 SQL 标准的各类配置规范所要求的 sql_sizing 值的信息。PostgreSQL 不跟踪任何 SQL 配置规范，因此此表为空。

表 37.47. sql_sizing_profiles 列

名称	数据类型	描述
sizing_id	cardinal_number	尺寸项的标识符
sizing_name	character_data	尺寸项的描述性名称
profile_id	character_data	配置规范的标识符字符串
required_value	cardinal_number	SQL 配置规范对该尺寸限制项要求的值；如果配置规范对此项不设限制，则为 0；如果配置规范不要求此项所适用的任何功能，则为 null
comments	character_data	关于此配置规范中该尺寸限制项的可选注释

37.50. table_constraints

视图 table_constraints 包含属于特定表的所有约束，这些表要满足的条件是：当前用户拥有表或者是当前用户在表上具有某种除SELECT 之外的权限。

表 37.48. table_constraints 列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名称
constraint_name	sql_identifier	约束的名称
table_catalog	sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名称
table_name	sql_identifier	表的名称
constraint_type	character_data	约束类型：CHECK、FOREIGN KEY、PRIMARY KEY 或UNIQUE
is_deferrable	yes_or_no	如果该约束是可延迟的，则为YES，否则为NO
initially_deferred	yes_or_no	如果该约束是可延迟的且初始就被延迟，则为YES，否则为NO
enforced	yes_or_no	适用于PostgreSQL尚不提供的功能（当前总是YES）

37.51. table_privileges

视图 table_privileges 标识在表或视图上所有被授予的权限，这些权限必须是被当前已启用角色授出或者被授予给当前已启用角色。对每一个表、授予者和被授权者的组合都有一行。

表 37.49. table_privileges 列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的角色名称
grantee	sql_identifier	被授予该权限的角色名称

名称	数据类型	描述
table_catalog	sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名称
table_name	sql_identifier	表的名称
privilege_type	character_data	该权限的类型：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES 或 TRIGGER
is_grantable	yes_or_no	如果该权限可再授予，则为YES，否则为NO
with_hierarchy	yes_or_no	在 SQL 标准中，WITH HIERARCHY OPTION 是一个独立的（子）权限，它允许在表继承层级上的特定操作。在 PostgreSQL 中，这被包括在SELECT 权限中，因此这一列在权限为SELECT 时显示YES，其他时候显示NO。

37.52. tables

视图 tables 包含定义在当前数据库中的所有表和视图。只有那些当前用户能够访问（作为拥有者或具有某些权限）的表和视图才会被显示。

表 37.50. tables 列

名称	数据类型	描述
table_catalog	sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名称
table_name	sql_identifier	表的名称
table_type	character_data	该表的类型：BASE TABLE 表示一个持久的基表（常见表类型），VIEW 表示一个视图，FOREIGN 表示一个外部表，LOCAL TEMPORARY 表示一个临时表
self_referencing_column_name	sql_identifier	对应 PostgreSQL 不支持的特性
reference_generation	character_data	对应 PostgreSQL 不支持的特性
user_defined_type_catalog	sql_identifier	如果该表是一个类型化表，则是包含其底层数据类型的数据库名称（总是当前数据库），否则为 null。
user_defined_type_schema	sql_identifier	如果该表是一个类型化表，则是包含其底层数据类型的模式名，否则为 null。
user_defined_type_name	sql_identifier	如果该表是一个类型化表，则是其底层数据类型的名称，否则为 null。

名称	数据类型	描述
is_insertable_into	yes_or_no	如果允许向该表插入数据，则为YES，否则为NO（基表总是允许插入，视图则不一定）。
is_typed	yes_or_no	如果该表是一个类型化表，则为YES，否则为NO
commit_action	character_data	还未被实现

37.53. transforms

视图 transforms 包含定义在当前数据库中的转换的信息。更准确来说，包含在转换中的每一个函数（“FROM SQL”或者“TO SQL”函数）在其中都有一行。

表 37.51. transforms 列

名称	数据类型	描述
udt_catalog	sql_identifier	包含该转换所适用类型的数据库的名称（总是当前数据库）
udt_schema	sql_identifier	包含该转换所适用类型的模式的名称
udt_name	sql_identifier	该转换所适用类型的名称
specific_catalog	sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	函数的“特定名称”。详见第 37.40 节。
group_name	sql_identifier	SQL 标准允许在“组”中定义转换，并且在运行时选择一个组。PostgreSQL 不支持这种做法，转换是与一种语言相关的。作为一种折衷，这个字段包含该转换所适用的语言。
transform_type	character_data	FROM SQL 或者TO SQL

37.54. triggered_update_columns

对于当前数据库中指定一个列列表（如 UPDATE OF column1, column2）的触发器，视图 triggered_update_columns 标识这些列。没有指定一个列列表的触发器不被包括在这个视图中。只有那些当前用户拥有或具有某种除 SELECT 之外权限的列才会被显示。

表 37.52. triggered_update_columns 列

名称	数据类型	描述
trigger_catalog	sql_identifier	包含该触发器的数据库名称（总是当前数据库）
trigger_schema	sql_identifier	包含该触发器的模式名称
trigger_name	sql_identifier	触发器的名称
event_object_catalog	sql_identifier	包含触发器所在的表的数据库名称（总是当前数据库）
event_object_schema	sql_identifier	包含触发器所在的表的模式名称

名称	数据类型	描述
event_object_table	sql_identifier	触发器所在的表的名称
event_object_column	sql_identifier	触发器所在的列的名称

37.55. triggers

视图 triggers 包含当前数据库中定义在表和视图上的所有触发器，前提是当前用户拥有这些表或视图，或具有其上的某种SELECT 之外的权限。

表 37.53. triggers 列

名称	数据类型	描述
trigger_catalog	sql_identifier	包含该触发器的数据库名称（总是当前数据库）
trigger_schema	sql_identifier	包含该触发器的模式名称
trigger_name	sql_identifier	触发器的名称
event_manipulation	character_data	触发该触发器的事件（INSERT、UPDATE 或DELETE）
event_object_catalog	sql_identifier	包含触发器所在的表的数据库名称（总是当前数据库）
event_object_schema	sql_identifier	包含触发器所在的表的模式名称
event_object_table	sql_identifier	触发器所在的表的名称
action_order	cardinal_number	同一个表上具有相同event_manipulation、action_timing 和action_orientation 的触发器之间的触发顺序。在PostgreSQL中，触发器按照名称顺序被触发，因此这一列会反映这种规则。
action_condition	character_data	触发器的WHEN 条件，没有条件时为 null（如果该表不归当前已启用角色所有，也为 null）
action_statement	character_data	触发器执行的语句（当前总是 EXECUTE FUNCTION *function*(...)）
action_orientation	character_data	标识触发器是对每个被处理的行触发一次还是为每个语句触发一次（ROW 或STATEMENT）
action_timing	character_data	触发器在什么时候触发（BEFORE、AFTER 或INSTEAD OF）
action_reference_old_table	sql_identifier	“旧”过渡表的名称，如果没有则为 null
action_reference_new_table	sql_identifier	“新”过渡表的名称，如果没有则为 null
action_reference_old_row	sql_identifier	对应 PostgreSQL 不支持的特性
action_reference_new_row	sql_identifier	对应 PostgreSQL 不支持的特性
created	time_stamp	对应 PostgreSQL 不支持的特性

PostgreSQL 中的触发器有两点与 SQL 标准不兼容，会影响它们在信息模式中的表示。第一，PostgreSQL 中的触发器名称局限于各自的表，触发器并不是模式中的独立对象。因此，同一模式中可以定义名称相同的触发器，只要它们属于不同的表。（trigger_catalog 和 trigger_schema 实际上是触发器所在表对应的数据库和模式值。）第二，PostgreSQL 允许将触发器定义为在多个事件上触发（例如 ON INSERT OR UPDATE），而 SQL 标准只允许一个事件。如果一个触发器被定义为在多个事件上触发，它在信息模式中就会表示为多行，每种事件类型对应一行。由于这两点，视图triggers 的主键实际上是(trigger_catalog, trigger_schema, event_object_table, trigger_name, event_manipulation)，而不是 SQL 标准规定的(trigger_catalog, trigger_schema, trigger_name)。不过，如果以符合 SQL 标准的方式定义触发器（名称在模式中唯一，且每个触发器只有一种事件类型），就不会受到影响。

> **注意**
>
> 在 PostgreSQL 9.1 之前，这个视图的列 action_timing、action_reference_old_table、action_reference_new_table、action_reference_old_row 和 action_reference_new_row 分别被命名为 condition_timing、condition_reference_old_table、condition_reference_new_table、condition_reference_old_row 和 condition_reference_new_row。那也是它们在 SQL:1999 标准中的命名。新的命名遵循 SQL:2003 及其后的版本。

37.56. udt_privileges

视图 udt_privileges 标识所有在用户定义类型上授予的 USAGE 权限，这些权限的授予者或者被授权者是当前已启用的角色。对每一个类型、授予者和被授权者的组合都有一行。这个视图只显示复合类型（原因见下面的第 37.58 节）。域权限见第 37.57 节。

表 37.54. udt_privileges 列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的角色名称
grantee	sql_identifier	被授予该权限的角色名称
udt_catalog	sql_identifier	包含该类型的数据库名称（总是当前数据库）
udt_schema	sql_identifier	包含该类型的模式名称
udt_name	sql_identifier	类型的名称
privilege_type	character_data	总是TYPE USAGE
is_grantable	yes_or_no	如果该权限可再授予，则为YES，否则为NO

37.57. usage_privileges

视图 usage_privileges 标识在多种对象上授予的所有 USAGE 权限，这些权限的授予者或被授权者是当前已启用的角色。在PostgreSQL 中，这目前适用于排序规则、域、外部数据包装器、外部服务器和序列。对于每个对象、授予者和被授权者的组合，这里都有一行。

由于 PostgreSQL 中的排序规则没有实际的权限，此视图对所有排序规则显示由拥有者隐式授予 PUBLIC、且不可再授予的USAGE 权限。其他对象类型则显示实际的权限。

在 PostgreSQL 中，序列也支持除USAGE 之外的 SELECT 和 UPDATE 权限。这些是非标准的并且因此在该信息模式中不可见。

表 37.55. usage_privileges 列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的角色名称
grantee	sql_identifier	被授予该权限的角色名称
object_catalog	sql_identifier	包含该对象的数据库名称（总是当前数据库）
object_schema	sql_identifier	如果适用，则为包含该对象的模式名称，否则为一个空字符串
object_name	sql_identifier	对象的名称
object_type	character_data	COLLATION 或DOMAIN 或 FOREIGN DATA WRAPPER 或 FOREIGN SERVER 或SEQUENCE
privilege_type	character_data	总是USAGE
is_grantable	yes_or_no	如果该权限可再授予，则为YES，否则为NO

37.58. user_defined_types

视图 user_defined_types 目前包含定义在当前数据库中的所有复合类型。只有那些当前用户能够访问（作为拥有者或具有某些权限）的类型才会被显示。

SQL 中有两种用户定义类型：结构类型（在 PostgreSQL 中也称为复合类型）以及独立类型（distinct type，PostgreSQL 尚未实现）。为兼顾将来，请使用列user_defined_type_category 来区分它们。其他用户定义类型，如基础类型和枚举（两者都是PostgreSQL 的扩展），不会在这里显示。域的相关信息见第 37.22 节。

表 37.56. user_defined_types 列

名称	数据类型	描述
user_defined_type_catalog	sql_identifier	包含该类型的数据库名称（总是当前数据库）
user_defined_type_schema	sql_identifier	包含该类型的模式名称
user_defined_type_name	sql_identifier	类型的名称
user_defined_type_category	character_data	当前总是STRUCTURED
is_instantiable	yes_or_no	对应 PostgreSQL 不支持的特性
is_final	yes_or_no	对应 PostgreSQL 不支持的特性
ordering_form	character_data	对应 PostgreSQL 不支持的特性
ordering_category	character_data	对应 PostgreSQL 不支持的特性
ordering_routine_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
ordering_routine_schema	sql_identifier	对应 PostgreSQL 不支持的特性
ordering_routine_name	sql_identifier	对应 PostgreSQL 不支持的特性
reference_type	character_data	对应 PostgreSQL 不支持的特性
data_type	character_data	对应 PostgreSQL 不支持的特性

名称	数据类型	描述
character_maximum_length	cardinal_number	对应 PostgreSQL 不支持的特性
character_octet_length	cardinal_number	对应 PostgreSQL 不支持的特性
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
collation_schema	sql_identifier	对应 PostgreSQL 不支持的特性
collation_name	sql_identifier	对应 PostgreSQL 不支持的特性
numeric_precision	cardinal_number	对应 PostgreSQL 不支持的特性
numeric_precision_radix	cardinal_number	对应 PostgreSQL 不支持的特性
numeric_scale	cardinal_number	对应 PostgreSQL 不支持的特性
datetime_precision	cardinal_number	对应 PostgreSQL 不支持的特性
interval_type	character_data	对应 PostgreSQL 不支持的特性
interval_precision	cardinal_number	对应 PostgreSQL 不支持的特性
source_dtd_identifier	sql_identifier	对应 PostgreSQL 不支持的特性
ref_dtd_identifier	sql_identifier	对应 PostgreSQL 不支持的特性

37.59. user_mapping_options

视图 user_mapping_options 包含在当前数据库中为用户映射定义的所有选项。只有那些当前用户能够访问其相应外部服务器（作为拥有者或具有某些权限）的用户映射才会被显示。

表 37.57. user_mapping_options 列

名称	数据类型	描述
authorization_identifier	sql_identifier	被映射的用户名称，如果映射是公共的则为PUBLIC
foreign_server_catalog	sql_identifier	这个映射所使用的外部服务器所在的数据库名称（总是当前数据库）
foreign_server_name	sql_identifier	这个映射所使用的外部服务器的名称
option_name	sql_identifier	一个选项名称
option_value	character_data	选项的值。除非满足以下任一条件，此列都显示为 null：当前用户是被映射的用户；映射针对 PUBLIC 且当前用户是服务器拥有者；当前用户是超级用户。这样做是为了保护作为用户映射选项存储的密码信息。

37.60. user_mappings

视图 user_mappings 包含定义在当前数据库中的所有用户映射。只有当前用户能够访问其对应外部服务器（作为拥有者或具有某些权限）的用户映射才会被显示。

表 37.58. user_mappings 列

名称	数据类型	描述
authorization_identifier	sql_identifier	被映射的用户名称，如果映射是公共的则为PUBLIC
foreign_server_catalog	sql_identifier	这个映射所使用的外部服务器所在的数据库名称（总是当前数据库）
foreign_server_name	sql_identifier	这个映射所使用的外部服务器的名称

37.61. view_column_usage

视图 view_column_usage 标识视图查询表达式（即定义该视图的 SELECT 语句）中使用的所有列。只有当包含该列的表归当前已启用角色所有时，该列才会出现在这个视图中。

> **注意**
>
> 此视图不包含系统表的列。这个问题有待修复。

表 37.59. view_column_usage 列

名称	数据类型	描述
view_catalog	sql_identifier	包含该视图的数据库名称（总是当前数据库）
view_schema	sql_identifier	包含该视图的模式名称
view_name	sql_identifier	视图的名称
table_catalog	sql_identifier	被视图所使用的列所属表的数据库名称（总是当前数据库）
table_schema	sql_identifier	被视图所使用的列所属表的模式名称
table_name	sql_identifier	被视图所使用的列所属表的名称
column_name	sql_identifier	被该视图所使用的列名称

37.62. view_routine_usage

视图 view_routine_usage 标识视图查询表达式（即定义该视图的 SELECT 语句）中使用的所有例程（函数和过程）。只有归当前已启用角色所有的例程才会出现在这个视图中。

表 37.60. view_routine_usage 列

名称	数据类型	描述
table_catalog	sql_identifier	包含该视图的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该视图的模式名称
table_name	sql_identifier	视图的名称
specific_catalog	sql_identifier	包含该函数的数据库名称（总是当前数据库）

名称	数据类型	描述
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	函数的“特定名称”。详见第 37.40 节。

37.63. view_table_usage

视图 view_table_usage 标识视图查询表达式（即定义该视图的 SELECT 语句）中使用的所有表。只有归当前已启用角色所有的表才会出现在这个视图中。

> **注意**
>
> 此视图不包含系统表。这个问题有待修复。

表 37.61. view_table_usage 列

名称	数据类型	描述
view_catalog	sql_identifier	包含该视图的数据库名称（总是当前数据库）
view_schema	sql_identifier	包含该视图的模式名称
view_name	sql_identifier	视图的名称
table_catalog	sql_identifier	包含被该视图所使用的表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含被该视图所使用的表的模式名称
table_name	sql_identifier	被该视图使用的表名称

37.64. views

视图 views 包含定义在当前数据库中的所有视图。只有当前用户能够访问（作为拥有者或具有某些权限）的视图才会被显示。

表 37.62. views 列

名称	数据类型	描述
table_catalog	sql_identifier	包含该视图的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该视图的模式名称
table_name	sql_identifier	视图的名称
view_definition	character_data	定义视图的查询表达式（如果该视图不被当前已启用角色拥有则为 null）
check_option	character_data	如果视图上定义了CHECK OPTION，则为CASCADED 或 LOCAL；如果没有定义，则为 NONE。
is_updatable	yes_or_no	如果视图可更新（允许UPDATE 和 DELETE），则为YES，否则为NO

名称	数据类型	描述
is_insertable_into	yes_or_no	如果视图可插入（允许INSERT），则为YES，否则为NO
is_trigger_updatable	yes_or_no	如果该视图上定义了 INSTEAD OF UPDATE 触发器，则为YES，否则为NO
is_trigger_deletable	yes_or_no	如果该视图上定义了 INSTEAD OF DELETE 触发器，则为YES，否则为NO
is_trigger_insertable_into	yes_or_no	如果该视图上定义了 INSTEAD OF INSERT 触发器，则为YES，否则为NO

部分 V. 服务器编程

本部分介绍如何使用用户定义函数、数据类型、触发器等扩展服务器功能。这些属于高级主题，通常应在读者已经理解其他有关PostgreSQL 的用户文档之后再来学习。本部分后面的章节介绍 PostgreSQL 发行版中提供的服务器端编程语言，以及与服务器端编程有关的一般问题。在深入学习服务器端编程的内容之前，至少应先阅读第 38 章的前几节（涵盖函数）。

目录

38. 扩展 SQL 1032
38.1. 可扩展性如何运作 1032
38.2. PostgreSQL 类型系统 1032
38.2.1. 基础类型 1032
38.2.2. 容器类型 1032
38.2.3. 域 1033
38.2.4. 伪类型 1033
38.2.5. 多态类型 1033
38.3. 用户定义的函数 1034
38.4. 用户定义的过程 1034
38.5. 查询语言（SQL）函数 1034
38.5.1. SQL 函数的参数 1035
38.5.2. 基础类型上的 SQL 函数 1036
38.5.3. 复合类型上的 SQL 函数 1039
38.5.4. 带有输出参数的 SQL 函数 1042
38.5.5. 带有可变数量参数的 SQL 函数 1043
38.5.6. 带有参数默认值的 SQL 函数 1044
38.5.7. SQL 函数作为表来源 1045
38.5.8. 返回集合的 SQL 函数 1045
38.5.9. 返回 TABLE 的 SQL 函数 1050
38.5.10. 多态 SQL 函数 1051
38.5.11. 带有排序规则的 SQL 函数 1052
38.6. 函数重载 1053
38.7. 函数易变性分类 1053
38.8. 过程语言函数 1055
38.9. 内部函数 1055
38.10. C 语言函数 1055
38.10.1. 动态载入 1055
38.10.2. C 语言函数中的基础类型 1057
38.10.3. 版本 1 的调用约定 1059
38.10.4. 编写代码 1063
38.10.5. 编译和链接动态装载函数 1063
38.10.6. 复合类型参数 1065
38.10.7. 返回行（复合类型） 1067
38.10.8. 返回集合 1068
38.10.9. 多态参数和返回类型 1074
38.10.10. 转换函数 1076
38.10.11. 共享内存与 LWLocks 1076
38.10.12. 把 C++ 用于可扩展性 1077
38.11. 用户定义的聚合 1077
38.11.1. 移动聚合模式 1079
38.11.2. 多态和可变参数聚合 1080
38.11.3. 有序集聚合 1082
38.11.4. 部分聚合 1083
38.11.5. 聚合的支持函数 1084
38.12. 用户定义的类型 1084
38.12.1. TOAST 注意事项 1088
38.13. 用户定义的操作符 1089
38.14. 操作符优化信息 1089
38.14.1. COMMUTATOR 1090
38.14.2. NEGATOR 1090

38.14.3. RESTRICT 1091
38.14.4. JOIN 1091
38.14.5. HASHES 1092
38.14.6. MERGES 1093
38.15. 索引扩展接口 1093
38.15.1. 索引方法和操作符类 1094
38.15.2. 索引方法策略 1094
38.15.3. 索引方法支持例程 1096
38.15.4. 示例 1098
38.15.5. 操作符类和操作符族 1100
38.15.6. 系统对操作符类的依赖 1103
38.15.7. 排序操作符 1105
38.15.8. 操作符类的特殊特性 1105
38.16. 将相关对象打包成扩展 1106
38.16.1. 扩展文件 1107
38.16.2. 扩展的可重定位性 1108
38.16.3. 扩展配置表 1109
38.16.4. 扩展更新 1110
38.16.5. 使用更新脚本安装扩展 1111
38.16.6. 扩展的安全注意事项 1111
38.16.7. 扩展示例 1112
38.17. 扩展构建基础设施 1113
39. 触发器 1118
39.1. 触发器行为概述 1118
39.2. 数据更改的可见性 1120
39.3. 用 C 编写触发器函数 1121
39.4. 一个完整的触发器示例 1124
40. 事件触发器 1128
40.1. 事件触发器行为概述 1128
40.2. 事件触发器触发矩阵 1129
40.3. 用 C 编写事件触发器函数 1133
40.4. 完整的事件触发器示例 1134
40.5. 表重写事件触发器示例 1136
41. 规则系统 1137
41.1. 查询树 1137
41.2. 视图和规则系统 1139
41.2.1. SELECT 规则如何工作 1139
41.2.2. 非 SELECT 语句中的视图规则 1145
41.2.3. PostgreSQL 中视图的威力 1146
41.2.4. 更新视图 1146
41.3. 物化视图 1147
41.4. INSERT、UPDATE 和 DELETE 规则 1150
41.4.1. 更新规则如何工作 1151
41.4.2. 与视图的协作 1156
41.5. 规则和权限 1162
41.6. 规则和命令状态 1164
41.7. 规则与触发器 1165
42. 过程语言 1169
42.1. 安装过程语言 1169
43. PL/pgSQL − SQL 过程语言 1172
43.1. 概述 1172
43.1.1. 使用 PL/pgSQL 的优点 1172
43.1.2. 支持的参数和结果数据类型 1172

43.2. PL/pgSQL 的结构 1173
43.3. 声明 1174
43.3.1. 声明函数参数 1175
43.3.2. ALIAS 1178
43.3.3. 复制类型 1178
43.3.4. 行类型 1178
43.3.5. 记录类型 1179
43.3.6. PL/pgSQL 变量的排序规则 1180
43.4. 表达式 1181
43.5. 基本语句 1181
43.5.1. 赋值 1182
43.5.2. 执行没有结果的命令 1182
43.5.3. 执行返回单行结果的查询 1183
43.5.4. 执行动态命令 1185
43.5.5. 获取结果状态 1188
43.5.6. 什么也不做 1189
43.6. 控制结构 1190
43.6.1. 从函数返回 1190
43.6.2. 从过程返回 1193
43.6.3. 调用过程 1193
43.6.4. 条件语句 1193
43.6.5. 简单循环 1197
43.6.6. 遍历查询结果 1200
43.6.7. 遍历数组 1201
43.6.8. 捕获错误 1202
43.6.9. 获得执行位置信息 1206
43.7. 游标 1206
43.7.1. 声明游标变量 1207
43.7.2. 打开游标 1207
43.7.3. 使用游标 1209
43.7.4. 遍历游标结果 1212
43.8. 事务管理 1212
43.9. 错误和消息 1214
43.9.1. 报告错误和消息 1214
43.9.2. 检查断言 1216
43.10. 触发器函数 1216
43.10.1. 数据更改触发器 1216
43.10.2. 事件触发器 1224
43.11. PL/pgSQL 内部机制 1225
43.11.1. 变量替换 1225
43.11.2. 计划缓存 1227
43.12. PL/pgSQL 开发提示 1229
43.12.1. 引号的处理 1229
43.12.2. 额外的编译时检查 1231
43.13. 从 Oracle PL/SQL 移植 1232
43.13.1. 移植示例 1233
43.13.2. 其他要关注的事项 1239
43.13.3. 附录 1239
44. PL/Tcl - Tcl 过程语言 1242
44.1. 概述 1242
44.2. PL/Tcl 函数和参数 1242
44.3. PL/Tcl 中的数据值 1244
44.4. PL/Tcl 中的全局数据 1245

44.5. 从 PL/Tcl 访问数据库 1245
44.6. PL/Tcl 中的触发器函数 1247
44.7. PL/Tcl 中的事件触发器函数 1249
44.8. PL/Tcl 中的错误处理 1250
44.9. PL/Tcl 中的显式子事务 1251
44.10. 事务管理 1251
44.11. PL/Tcl 配置 1252
44.12. Tcl 过程名 1252
45. PL/Perl - Perl 过程语言 1253
45.1. PL/Perl 函数和参数 1253
45.2. PL/Perl 中的数据值 1258
45.3. 内置函数 1258
45.3.1. 从 PL/Perl 访问数据库 1258
45.3.2. PL/Perl 中的辅助函数 1263
45.4. PL/Perl 中的全局值 1264
45.5. 受信任与不受信任的 PL/Perl 1265
45.6. PL/Perl 触发器 1266
45.7. PL/Perl 事件触发器 1268
45.8. PL/Perl 内部机制 1268
45.8.1. 配置 1268
45.8.2. 限制与缺失特性 1269
46. PL/Python - Python 过程语言 1271
46.1. Python 2 与 Python 3 1271
46.2. PL/Python 函数 1272
46.3. 数据值 1273
46.3.1. 数据类型映射 1274
46.3.2. 空值、None 1275
46.3.3. 数组、列表 1276
46.3.4. 复合类型 1277
46.3.5. 集合返回函数 1279
46.4. 共享数据 1280
46.5. 匿名代码块 1280
46.6. 触发器函数 1280
46.7. 数据库访问 1281
46.7.1. 数据库访问函数 1281
46.7.2. 捕获错误 1284
46.8. 显式子事务 1285
46.8.1. 子事务上下文管理器 1285
46.8.2. 较早的 Python 版本 1286
46.9. 事务管理 1287
46.10. 辅助函数 1287
46.11. 环境变量 1288
47. 服务器编程接口 1290
47.1. 接口函数 1290
47.2. 接口支持函数 1325
47.3. 内存管理 1334
47.4. 事务管理 1344
47.5. 数据更改的可见性 1347
47.6. 示例 1347
48. 后台工作进程 1351
49. 逻辑解码 1354
49.1. 逻辑解码示例 1354
49.2. 逻辑解码概念 1356
49.2.1. 逻辑解码 1356

49.2.2. 复制槽 1356
49.2.3. 输出插件 1356
49.2.4. 导出快照 1356
49.3. 流复制协议接口 1357
49.4. 逻辑解码 SQL 接口 1357
49.5. 与逻辑解码相关的系统目录 1357
49.6. 逻辑解码输出插件 1357
49.6.1. 初始化函数 1358
49.6.2. 能力 1358
49.6.3. 输出模式 1358
49.6.4. 输出插件回调 1358
49.6.5. 产生输出的函数 1361
49.7. 逻辑解码输出写入器 1361
49.8. 逻辑解码的同步复制支持 1361
49.8.1. 概述 1361
49.8.2. 注意事项 1362
50. 复制进度跟踪 1363

第 38 章 扩展 SQL

在接下来的各节中，我们将讨论如何通过增加以下内容来扩展 PostgreSQL SQL 查询语言：

- 函数（从第 38.3 节开始）
- 聚合（从第 38.11 节开始）
- 数据类型（从第 38.12 节开始）
- 操作符（从第 38.13 节开始）
- 索引的操作符类（从第 38.15 节开始）
- 相关对象的包（从第 38.16 节开始）

38.1. 可扩展性如何运作

PostgreSQL 之所以具有可扩展性，是因为其运作由系统目录驱动。如果你熟悉标准的关系数据库系统，就会知道它们把有关数据库、表、列等的信息存储在通常所说的系统目录中（有些系统把这称为数据字典）。这些目录对用户而言看起来就像普通表一样，但 DBMS 会在其中保存自己的内部管理信息。PostgreSQL 与标准关系数据库系统的一个关键差别是，PostgreSQL 在目录中存储的信息要多得多：不仅有关于表和列的信息，还有关于数据类型、函数、访问方法等的信息。这些表可以由用户修改，而 PostgreSQL 又是基于这些表来运行的，这意味着 PostgreSQL 可以由用户扩展。相比之下，传统数据库系统只能通过修改源代码中的硬编码过程，或加载由 DBMS 供应商专门编写的模块来扩展。

此外，PostgreSQL 服务器还能通过动态加载把用户编写的代码纳入自身。也就是说，用户可以指定一个实现了新类型或新函数的目标代码文件（例如共享库），而 PostgreSQL 会在需要时加载它。把用 SQL 编写的代码加入服务器就更为简单了。这种能够“即时”修改自身行为的能力，使 PostgreSQL 特别适合用于新应用和新存储结构的快速原型设计。

38.2. PostgreSQL 类型系统

PostgreSQL 数据类型可以分为基础类型、容器类型、域和伪类型。

38.2.1. 基础类型

基础类型是像 `integer` 这样的类型，它们在 SQL 语言层之下实现（通常使用像 C 这样的底层语言）。它们通常对应于通常所说的抽象数据类型。PostgreSQL 只能通过用户提供的函数来操作这类类型，并且只能在用户所描述的范围内理解这类类型的行为。内置基础类型见第 8 章。

枚举（enum）类型可以看作是基础类型的一个子类。其主要区别在于，它们只需使用 SQL 命令即可创建，而不需要任何底层编程。更多信息参见第 8.7 节。

38.2.2. 容器类型

PostgreSQL 有三种“容器”类型，即包含其他类型多个值的类型。它们是数组、复合类型和范围类型。

数组可以保存多个同一类型的值。每一种基础类型、复合类型、范围类型和域类型都会自动创建一个对应的数组类型。但不存在数组的数组。就类型系统而言，多维数组与一维数组并无区别。更多信息参见第 8.15 节。

复合类型，也称行类型，会在用户创建表时自动产生。也可以使用CREATE TYPE 定义一个没有关联表的“独立”复合类型。复合类型只是带有关联字段名的一组类型。复合类型的值是一行或一条记录，其中包含各字段的值。更多信息参见第 8.16 节。

范围类型可以保存同一类型的两个值，它们分别是该范围的下界和上界。范围类型由用户创建，不过也存在少数内置的范围类型。更多信息参见第 8.17 节。

38.2.3. 域

域基于某一种特定的底层类型，并且在很多场合下都可以与其底层类型互换。不过，域可以带有约束，把其合法值限制在底层类型所允许值的一个子集内。域使用 SQL 命令 CREATE DOMAIN 创建。更多信息参见第 8.18 节。

38.2.4. 伪类型

有少数几种“伪类型”用于特殊目的。伪类型不能作为表的列或容器类型的组成部分出现，但它们可以用来声明函数的参数类型和结果类型。这为在类型系统内部识别特殊类别的函数提供了一种机制。表 8.25 列出了现有的伪类型。

38.2.5. 多态类型

五种特别值得关注的伪类型是 `anyelement`、`anyarray`、`anynonarray`、`anyenum` 和 `anyrange`，它们统称为多态类型。使用这些类型声明的函数称为多态函数。多态函数可以操作多种不同的数据类型，具体类型由某次调用时实际传入的数据类型决定。

多态参数和结果相互关联，在解析调用多态函数的查询时，会被解析为某种具体的数据类型。每个声明为 `anyelement` 的位置（参数或返回值）都可以具有任意具体的实际数据类型，但在同一次调用中，它们必须具有相同的实际类型。每个声明为 `anyarray` 的位置可以具有任意数组数据类型，但同样必须全部具有相同的类型。类似地，声明为 `anyrange` 的位置必须全部具有相同的范围类型。此外，如果某些位置声明为 `anyarray`，另一些位置声明为 `anyelement`，那么 `anyarray` 位置上的实际数组类型，其元素类型必须与 `anyelement` 位置上的类型相同。类似地，如果某些位置声明为 `anyrange`，另一些位置声明为 `anyelement` 或 `anyarray`，那么 `anyrange` 位置上的实际范围类型，其子类型必须与 `anyelement` 位置上的类型相同，并与 `anyarray` 位置上的元素类型相同。`anynonarray` 的处理与 `anyelement` 完全相同，但增加了实际类型不能为数组类型的约束。`anyenum` 的处理与 `anyelement` 完全相同，但增加了实际类型必须为枚举类型的约束。

因此，当多个参数位置被声明为多态类型时，其总体效果就是只允许某些实际参数类型组合。例如，声明为 `equal(anyelement, anyelement)` 的函数可以接受任意两个输入值，只要它们属于同一数据类型。

如果函数的返回值声明为多态类型，那么至少有一个参数位置也必须是多态的，并且实际传入的参数数据类型决定该次调用的实际结果类型。例如，如果尚不存在数组下标机制，可以将实现下标访问的函数定义为 `subscript(anyarray, integer) returns anyelement`。这一声明将实际的第一个参数约束为数组类型，并允许解析器根据第一个参数的实际类型推导正确的结果类型。另一个例子是，声明为 `f(anyarray) returns anyenum` 的函数只接受枚举类型的数组。

在大多数情况下，解析器可以根据属于另一种多态类型的参数，推导出多态结果类型的实际数据类型。例如，可以从 `anyelement` 推导出 `anyarray`，反之亦然。例外是：类型为 `anyrange` 的多态结果要求有一个类型为 `anyrange` 的参数；无法从 `anyarray` 或 `anyelement` 参数推导出来。这是因为多个范围类型可能具有相同的子类型。

注意，`anynonarray` 和 `anyenum` 并不表示独立的类型变量；它们与 `anyelement` 是同一个类型，只是附加了额外约束。例如，把函数声明为 `f(anyelement, anyenum)`，等价于把它声明为 `f(anyenum, anyenum)`：两个实际参数都必须是同一种枚举类型。

可变参数函数（接受可变数量参数的函数，见第 38.5.5 节）可以是多态的：将其最后一个参数声明为 `VARIADIC anyarray` 即可。就参数匹配和确定实际结果类型而言，这样的函数与写出相应数量的 `anynonarray` 参数时行为相同。

38.3. 用户定义的函数

PostgreSQL 提供四种函数：

- 查询语言函数（用 SQL 编写的函数）（第 38.5 节）
- 过程语言函数（例如，用PL/pgSQL 或 PL/Tcl 编写的函数）（第 38.8 节）
- 内部函数（第 38.9 节）
- C 语言函数（第 38.10 节）

每一类函数可以采用基础类型、复合类型或者它们的组合作为参数。此外，每一类函数可以返回一个基础类型或一个复合类型。函数也能被定义成返回基础类型或复合类型值的集合。

很多类函数可以接受或者返回特定的伪类型（例如，多态类型），但是可用的功能因函数类别而异。详情可以参考每一种函数的描述。

定义 SQL 函数最容易，因此我们将从讨论它们开始。大部分SQL 函数的概念也能用到其他类型的函数上。

在这一章中，查看CREATE FUNCTION 命令的参考页有助于更好地理解示例。这章中的一些示例可以在 PostgreSQL 源代码发布的 `src/tutorial` 目录中的 `funcs.sql` 和 `funcs.c` 中找到。

38.4. 用户定义的过程

过程是一个类似于函数的数据库对象。关键差异是：

- 过程是由 CREATE PROCEDURE 命令定义，而不是 `CREATE FUNCTION`。
- 过程不返回函数值；因此 `CREATE PROCEDURE` 缺少 `RETURNS` 子句。不过，过程可以通过输出参数向调用方返回数据。
- 函数可以作为查询或 DML 命令的一部分被调用，而过程则通过CALL 命令单独调用。
- 只要调用 `CALL` 命令不是显式事务块的一部分，过程就可以在执行期间提交或回滚事务（随后会自动开始一个新事务）。函数则不能这样做。
- 某些函数属性（例如严格性）不适用于过程。这些属性控制如何在查询中使用函数，与过程不相关。

后面章节中关于如何定义用户定义的函数的解释同样适用于过程，除了上面指出的那些点。

函数和过程统称为例程。有 ALTER ROUTINE 和 DROP ROUTINE 这样的命令可以操作函数和过程而不需要知道它们是哪一种。不过，要注意没有`CREATE ROUTINE` 命令。

38.5. 查询语言（SQL）函数

SQL 函数执行由任意 SQL 语句组成的一个列表，并返回列表中最后一个查询的结果。在简单（非集合）情况下，将返回最后一个查询结果的第一行。（请记住，除非使用 `ORDER BY`，否则多行结果中的“第一行”并没有明确定义。）如果最后一个查询完全没有返回任何行，则返回空值。

或者，一个 SQL 函数可以通过指定函数的返回类型为SETOF *sometype* 被声明为返回一个集合（也就是多个行），或者等效地声明它为RETURNS TABLE(*columns*)。在这种情况下，最后一个查询的结果的所有行会被返回。下文将给出进一步的细节。

SQL 函数的函数体必须是一个由分号分隔的 SQL 语句列表。最后一条语句后的分号是可选的。除非该函数被声明为返回 void，否则最后一条语句必须是 SELECT，或者是 INSERT、UPDATE 或 DELETE 并带有 RETURNING 子句。

任意一组 SQL 语言命令都可以打包并定义成函数。除 SELECT 查询外，这些命令还可以包含数据修改查询（INSERT，UPDATE、DELETE），以及其他 SQL 命令。（不能使用事务控制命令，例如 COMMIT，SAVEPOINT，以及某些工具命令，例如 VACUUM，来编写 SQL 函数。）不过，最后一条命令必须是 SELECT，或者带有 RETURNING 子句，并返回与函数声明返回类型相符的结果。或者，如果你想定义一个执行动作但没有有用返回值的 SQL 函数，也可以把它定义为返回 void。例如，下面这个函数会删除 emp 表中薪资为负的行：

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------

(1 row)
```

> **注意**
>
> SQL 函数的整个函数体都会先经过解析，然后才开始执行。虽然 SQL 函数可以包含修改系统目录的命令（例如 CREATE TABLE），但在对函数中的后续命令进行解析分析时，这些命令的效果还不可见。因此，例如将 CREATE TABLE foo (...); INSERT INTO foo VALUES(...); 打包到同一个 SQL 函数中，就不会按预期工作，因为 foo 在解析 INSERT 命令时还不存在。在这类情况下，建议使用 PL/pgSQL 而不是 SQL 函数。

CREATE FUNCTION 命令的语法要求把函数体写成一个字符串常量。对于字符串常量，通常使用美元引用最方便（见第 4.1.2.4 节）。如果你选择使用常规的单引号字符串常量语法，那么必须在函数体中将单引号（'）和反斜线（\）写成双份（假定使用转义字符串语法，见第 4.1.2.1 节）。

38.5.1. SQL 函数的参数

一个 SQL 函数的参数可以在函数体中用名称或编号引用。下面会有两种方法的示例。

要使用一个名称，将函数参数声明为带有一个名称，然后在函数体中只写该名称。如果参数名称与函数内当前 SQL 命令中的任意列名相同，列名将优先。如果不想这样，可以用函数本身的名称来限定参数名，也就是*function_name*.*argument_name*（如果这会与一个被限定的列名冲突，照例还是列名赢得优先。你可以通过为 SQL 命令中的表选择一个不同的别名来避免这种混淆）。

在更旧的数字方法中，参数可以用语法$*n* 引用：$1 指的是第一个输入参数，$2 指的是第二个，以此类推。不管特定的参数是否使用名称声明，这种方法都有效。

如果一个参数是一种复合类型，那么点号记法（如 *argname*.*fieldname* 或 $1.*fieldname*）也可以被用来访问该参数的属性。同样，你可能需要用函数的名称来限定参数的名称以避免歧义。

SQL 函数参数只能被用做数据值而不能作为标识符。例如这是合理的：

```
INSERT INTO mytable VALUES ($1);
```

但这样就不行：

```
INSERT INTO $1 VALUES (42);
```

> **注意**
>
> 使用名称来引用 SQL 函数参数的能力是在 PostgreSQL 9.2 中加入的。要在老的服务器中使用的函数必须使用$*n* 记法。

38.5.2. 基础类型上的 SQL 函数

最简单的 SQL 函数没有参数，只是返回一个基础类型，例如 integer：

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;

-- Alternative syntax for string literal:
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

 one
-----
   1
```

注意我们为该函数的结果在函数体内定义了一个列别名（名为 result），但是这个列别名在函数以外是不可见的。因此，结果被标记为one 而不是 result。

定义接受基础类型参数的 SQL 函数也几乎同样容易：

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

或者，我们也可以不用参数名，而改用数字：

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

下面是一个更有用的函数，它可以用来从银行账户中扣款：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT 1;
$$ LANGUAGE SQL;
```

用户可以这样执行该函数，从 17 号账户中扣除 $100.00：

```
SELECT tf1(17, 100.0);
```

在这个示例中，我们为第一个参数选择了名称`accountno`，但是这和表`bank` 中的一个列名相同。在 `UPDATE` 命令中，`accountno` 引用列 `bank.accountno`，因此 `tf1.accountno` 必须被用来引用该参数。我们当然可以通过为该参数使用一个不同的名称来避免这样的问题。

实际上我们可能喜欢从该函数得到一个更有用的结果而不是一个常数 1，因此一个更可能的定义是：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

它会调整余额并且返回新的余额。同样的事情也可以用一个使用 `RETURNING` 的命令实现：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

一个 SQL 函数必须返回与其声明的结果类型完全一致的类型。这可能需要插入显式类型转换。例如，假设我们希望前面的 `add_em` 函数改为返回 `float8` 类型。下面这样是不行的：

```
CREATE FUNCTION add_em(integer, integer) RETURNS float8 AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;
```

即便在其他上下文中，PostgreSQL 愿意插入隐式类型转换，把 `integer` 转换为 `float8`。我们需要将它写成：

```
CREATE FUNCTION add_em(integer, integer) RETURNS float8 AS $$
    SELECT ($1 + $2)::float8;
$$ LANGUAGE SQL;
```

38.5.3. 复合类型上的 SQL 函数

编写接受复合类型参数的函数时，我们不仅必须指定要用哪个参数，还必须指定该参数的哪个属性（字段）。例如，假设 `emp` 是一个包含雇员数据的表，因此它也是该表每一行对应的复合类型名称。下面的函数 `double_salary` 用来计算某人的工资翻倍后会是多少：

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | dream
------+-------
 Bill |  8400
```

注意这里用 `$1.salary` 语法来选取参数行值中的一个字段。还要注意，调用时的 `SELECT` 命令使用 *`table_name`*`.*` 将表的当前整行取作一个复合值。该表行也可以仅用表名来引用：

```
SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

但这种用法已被弃用，因为它很容易让人混淆（关于表行复合值这两种记法的更多细节，见第 8.16.5 节）。

有时候即时构造一个复合参数值会很方便。这可以用 `ROW` 构造器完成。例如，我们可以调整被传递给函数的数据：

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
    FROM emp;
```

也可以构建一个返回复合类型的函数。这是一个返回单一 emp 行的函数示例：

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
        1000.0 AS salary,
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

在这个示例中，我们为每一个属性指定了一个常量值，但是可以用任何计算来替换这些常量。

定义该函数时有两点重要注意事项：

- 查询中的选择列表顺序必须与列在该复合类型所关联的表中出现的顺序完全相同。（系统不考虑像上面这样指定的列名。）
- 我们必须确保每个表达式的类型与复合类型中对应列的类型相匹配，必要时插入类型转换。否则，就会得到这样的错误：

  ```
  ERROR:  function declared to return emp returns varchar instead of text at column 1
  ```

  与基础类型的情况一样，该函数不会自动插入任何类型转换。

定义同一个函数的另一种方法是：

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

这里我们编写了一个 SELECT，它只返回具有正确复合类型的单列。在这种情况下，这样做并没有更好，但在某些情况下它是一种方便的替代方法 – 例如，如果我们需要通过调用另一个返回所需复合值的函数来计算结果。另一个例子是，如果我们要编写的函数返回的是基于复合类型的域，而不是普通复合类型，就总是必须将其写成返回单列的形式，因为没有其他方法可以产生类型恰好为该域类型的值。

我们既可以把这个函数直接当作值表达式调用：

```
SELECT new_emp();

         new_emp
-------------------------
 (None,1000.0,25,"(2,2)")
```

也可以把它当作表函数来调用：

```
SELECT * FROM new_emp();

 name | salary | age | cubicle
------+--------+-----+---------
 None | 1000.0 |  25 | (2,2)
```

第二种方式会在第 38.5.7 节中进一步说明。

当你使用返回复合类型的函数时，可能只需要取其结果中的一个字段（属性）。可以使用下面这样的语法：

```
SELECT (new_emp()).name;

 name
------
 None
```

这里需要额外的括号，以免解析器产生歧义。如果不加括号，结果会是这样：

```
SELECT new_emp().name;
ERROR:  syntax error at or near "."
LINE 1: SELECT new_emp().name;
                        ^
```

另一种选择是使用函数记法来提取属性：

```
SELECT name(new_emp());

 name
------
 None
```

如第 8.16.5 节所述，字段记法和函数记法是等价的。

另一种使用函数返回复合类型的方法是将结果传递给另一个接受正确行类型作为输入的函数：

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

38.5.4. 带有输出参数的 SQL 函数

描述函数结果的另一种方法是使用输出参数，例如：

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
LANGUAGE SQL;

SELECT add_em(3,7);
 add_em
--------
     10
(1 row)
```

这与第 38.5.2 节中展示的 `add_em` 版本本质上并无不同。输出参数的真正价值在于，它们为定义返回多列的函数提供了一种方便的方法。例如：

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

这里本质上发生的事情是：我们为该函数的结果创建了一个匿名复合类型。上面的示例与下面这种写法的最终效果相同：

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

但通常不必额外定义一个独立的复合类型会更方便。注意，附在输出参数上的名称并非只是装饰，它们决定了这个匿名复合类型的列名。（如果省略输出参数名称，系统会自行选择一个名称。）

在从 SQL 调用这样一个函数时，输出参数不会被包括在调用参数列表中。这是因为PostgreSQL只考虑输入参数来定义函数的调用签名。这也意味着在为诸如删除函数等目的引用该函数时只有输入参数有关系。我们可以用下面的命令之一删除上述函数

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

参数可以被标记为 IN（默认）、OUT、INOUT 或者 VARIADIC。一个 INOUT 参数既作为一个输入参数（调用参数列表的一部分）又作为一个输出参数（结果记录类型的一部分）。VARIADIC 参数是输入参数，但被按照下文所述特殊对待。

38.5.5. 带有可变数量参数的 SQL 函数

SQL 函数可以被声明为接受可变数量的参数，只要所有“可选”参数都属于同一种数据类型。可选参数会以数组形式传递给函数。定义这类函数时，需要把最后一个参数标记为 VARIADIC；该参数必须被声明为数组类型。例如：

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
 mleast
--------
     -1
(1 row)
```

实际上，位于 VARIADIC 位置及之后的所有实参都会被收集成一个一维数组，就像你写成了：

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- doesn't work
```

不过你实际上不能这样写，至少它不会匹配这个函数定义。被标记为 VARIADIC 的参数匹配的是其元素类型出现一次或多次，而不是它自身的数组类型。

有时候，能够把一个已经构造好的数组传给可变参数函数会很有用，尤其是当一个可变参数函数想把它的数组参数再传给另一个函数时。此外，在允许不受信任用户创建对象的模式中调用可变参数函数时，这是唯一安全的方式，见第 10.3 节。你可以在调用中指定 VARIADIC 来做到这一点：

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

这样会阻止函数的可变参数按其元素类型展开，从而让数组实参能够按常规方式匹配。VARIADIC 只能附加在函数调用的最后一个实参上。

在调用中指定 VARIADIC 也是向可变参数函数传递空数组的唯一方式，例如：

```
SELECT mleast(VARIADIC ARRAY[]::numeric[]);
```

仅仅写成 SELECT mleast() 是行不通的，因为可变参数必须匹配至少一个实参。（如果你希望允许这种调用，可以再定义一个同名且不带参数的函数mleast。）

从可变参数派生出的数组元素参数会被视为没有自己的名字。这意味着除非你指定了 VARIADIC，否则不能使用命名参数来调用可变参数函数（第 4.3 节）。例如，下面的调用是可行的：

```
SELECT mleast(VARIADIC arr => ARRAY[10, -1, 5, 4.4]);
```

但这些就不行：

```
SELECT mleast(arr => 10);
SELECT mleast(arr => ARRAY[10, -1, 5, 4.4]);
```

38.5.6. 带有参数默认值的 SQL 函数

函数可以被声明为对一些或者所有输入参数具有默认值。只要调用函数时没有给出足够多的实参，就会插入默认值来弥补缺失的实参。由于参数只能从实参列表的尾部开始被省略，在一个有默认值的参数之后的所有参数都不得不也具有默认值（尽管使用命名参数记法可以允许放松这种限制，这种限制仍然会被强制以便位置参数记法能工作）。不管你是否使用它，这种能力都要求在某些用户不信任其他用户的数据库中调用函数时采取一些预防措施，见第 10.3 节。

例如：

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);
 foo
-----
  60
(1 row)

SELECT foo(10, 20);
 foo
-----
  33
(1 row)

SELECT foo(10);
 foo
-----
  15
(1 row)

SELECT foo();  -- fails since there is no default for the first argument
ERROR:  function foo() does not exist
```

也可以用 = 符号代替关键字 DEFAULT。

38.5.7. SQL 函数作为表来源

所有的 SQL 函数都可以被用在查询的 FROM 子句中，但是对于返回复合类型的函数特别有用。如果函数被定义为返回一种基础类型，该表函数会产生一个单列表。如果该函数被定义为返回一种复合类型，该表函数会为该复合类型的每一个属性产生一列。

以下是一个示例：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

如示例所示，我们可以像操作普通表的列一样操作函数结果中的列。

注意我们只从函数得到了一行。这是因为我们没有使用 SETOF。这会在下一节中介绍。

38.5.8. 返回集合的 SQL 函数

当一个 SQL 函数被声明为返回 SETOF *sometype* 时，该函数的最后一个查询会被执行完，并且它输出的每一行都会被作为结果集的一个元素返回。

这一特性通常是在查询的 FROM 子句中调用函数时使用的。在这种情况下，函数返回的每一行都会成为查询所见表中的一行。例如，假设表 foo 与上文相同，我们写：

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

那么会得到：

```
 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

也可以通过输出参数定义的列来返回多行，例如：

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
 sum | product
-----+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

这里的关键点是：你必须写成 `RETURNS SETOF record`，以表明该函数返回的是多行而不是单行。如果只有一个输出参数，则写该参数的类型，而不是 `record`。

通过多次调用集合返回函数来构造查询结果通常很有用，其中每次调用的参数都来自表或子查询的连续行。首选方式是使用 LATERAL 关键字，详见第 7.2.1.5 节。下面是一个使用集合返回函数枚举树结构元素的示例：

```
SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL STABLE;

SELECT * FROM listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;
  name  |   child
--------+-----------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

这个例子并没有做出任何无法用简单连接实现的事，但在更复杂的计算中，把一部分工作放进函数里通常会非常方便。

集合返回函数也可以出现在查询的选择列表中。对于查询本身生成的每一行，集合返回函数都会被调用，并为其结果集中的每一个元素生成一行输出。前面的例子也可以写成这样：

```
SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name   | listchildren
---------+--------------
 Top     | Child1
 Top     | Child2
 Top     | Child3
 Child1  | SubChild1
 Child1  | SubChild2
(5 rows)
```

在最后一个 `SELECT` 中，注意没有为 `Child2`、`Child3` 等显示任何输出行。这是因为 `listchildren` 对这些参数返回的是空集，因此不会生成结果行。这与使用 `LATERAL` 语法把函数结果做内连接时得到的行为相同。

PostgreSQL 对查询选择列表中的集合返回函数的处理方式，几乎与将该函数写在 `LATERAL FROM` 子句项中时完全相同。例如，

```
SELECT x, generate_series(1,5) AS g FROM tab;
```

几乎等价于

```
SELECT x, g FROM tab, LATERAL generate_series(1,5) AS g;
```

两者本应完全相同，但在这个特定示例中，规划器可以选择将 `g` 放在嵌套循环连接的外侧，因为 `g` 实际上并不横向依赖于 `tab`。这会导致输出行的顺序不同。对选择列表中的集合返回函数求值时，总是将其视为位于与 `FROM` 子句其余部分的嵌套循环连接的内侧，因此在考虑 `FROM` 子句的下一行之前，这些函数都会运行到完成。

如果在查询的选择列表中有不止一个集合返回函数，则行为类似于把那些函数放到一个单一的 `LATERAL ROWS FROM( ... ) FROM` 子句项中的行为。对于来自底层查询的每一行，都有一个用到每个函数首个结果的输出行，然后是一个使用每个函数第二个结果的输出行，以此类推。如果某些集合返回函数产生的输出比其他函数少，会用空值代替缺失的数据，因此为一个底层行形成的总行数等于产生最多输出的集合返回函数的输出行数。因此集合返回函数会“步调一致”地运行直到它们的输出被耗尽，然后用下一个底层行继续执行。

集合返回函数可以被嵌套在一个选择列表中，不过在FROM 子句项中不允许这样做。在这种情况下，嵌套的每一层会被单独对待，就像它是一个单独的LATERAL ROWS FROM(...) 项一样。例如，在

```
SELECT srf1(srf2(x), srf3(y)), srf4(srf5(z)) FROM tab;
```

中，集合返回函数srf2、srf3 和 srf5 将为 tab 的每一行步调一致地运行，然后会对较低层的函数产生的每一行以步调一致的形式应用srf1 和 srf4。

在 CASE 或 COALESCE 这样的条件计算结构中，不能使用集合返回函数。例如，考虑

```
SELECT x, CASE WHEN x > 0 THEN generate_series(1, 5) ELSE 0 END FROM tab;
```

看起来这个语句应该产生满足 x > 0 的输入行的五次重复，以及不满足的行的一次重复。但实际上，由于在CASE 表达式被计算前，generate_series(1, 5) 会被运行在一个隐式的LATERAL FROM 项中，它会为每个输入行产生五次重复。为了减少混乱，这类情况会产生一个解析时错误。

> **注意**
>
> 如果一个函数的最后一条命令是 INSERT、UPDATE 或 DELETE 并带有 RETURNING，那么即使该函数没有声明为 SETOF，或者调用它的查询并未取走全部结果行，该命令也总会执行到完成。RETURNING 子句产生的额外结果行会被静默丢弃，但相应的表修改仍然会发生，并且会在函数返回前全部完成。

注意

在 PostgreSQL 10 之前，除非多个集合返回函数始终产生相同数量的行，否则把它们放在同一个选择列表中的行为并不合理。否则，得到的输出行数会是这些集合返回函数各自产生行数的最小公倍数。此外，嵌套的集合返回函数也不像上文所描述的那样工作；相反，一个集合返回函数至多只能有一个集合返回参数，并且每一组嵌套的集合返回函数都会独立运行。再加上过去还允许条件执行（即在 CASE 等结构内部使用集合返回函数），事情就更加复杂了。如果你要编写需要在旧版 PostgreSQL 上运行的查询，建议使用 LATERAL 语法，因为这样在不同版本之间能够得到一致的结果。如果你的查询依赖于集合返回函数的条件执行，或许可以把条件判断移动到自定义集合返回函数内部来修复。例如，

```
SELECT x, CASE WHEN y > 0 THEN generate_series(1, z) ELSE 5 END FROM tab;
```

可以变成

```
CREATE FUNCTION case_generate_series(cond bool, start int, fin int, els int)
  RETURNS SETOF int AS $$
BEGIN
  IF cond THEN
    RETURN QUERY SELECT generate_series(start, fin);
  ELSE
    RETURN QUERY SELECT els;
  END IF;
END$$ LANGUAGE plpgsql;

SELECT x, case_generate_series(y > 0, 1, z, 5) FROM tab;
```

这种写法在所有版本的 PostgreSQL 中的行为都相同。

38.5.9. 返回 TABLE 的 SQL 函数

还有另一种方法可以把函数声明为返回一个集合，即使用 RETURNS TABLE(*columns*) 语法。这等效于使用一个或者多个 OUT 参数外加把函数标记为返回 SETOF record（或者是 SETOF 单个输出参数的类型）。这种写法是在最近的 SQL 标准中指定的，因此可能比使用 SETOF 的移植性更好。

例如，前面的求和并且相乘的示例也可以这样来做：

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

不允许把显式的 OUT 或者 INOUT 参数用于 RETURNS TABLE 记法 – 必须把所有输出列放在 TABLE 列表中。

38.5.10. 多态 SQL 函数

SQL 函数可以被声明为接受并返回这些多态类型：anyelement, anyarray, anynonarray, anyenum、anyrange。关于多态函数的更详细解释，见第 38.2.5 节。下面这个多态函数 make_array 会根据两个任意数据类型的元素构造一个数组：

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

注意类型转换'a'::text 的使用是为了指定该参数的类型是 text。如果该参数只是一个字符串字面量，这就是必须的，因为否则它会被当作unknown 类型，并且 unknown 的数组也不是一种合法的类型。如果没有该类型转换，将得到这样的错误：

```
ERROR:  could not determine polymorphic type because input has type unknown
```

允许存在具有固定返回类型的多态参数，但反过来则不允许。例如：

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A function returning a polymorphic type must have at least one polymorphic argument.
```

多态也可以和带输出参数的函数一起使用。例如：

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
 22 | {22,22}
(1 row)
```

多态也可以用于可变参数函数。例如：

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
 anyleast
----------
       -1
(1 row)

SELECT anyleast('abc'::text, 'def');
 anyleast
----------
 abc
(1 row)

CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat_values('|', 1, 4, 2);
 concat_values
---------------
 1|4|2
(1 row)
```

38.5.11. 带有排序规则的 SQL 函数

当一个 SQL 函数有一个或多个支持排序规则的数据类型参数时，会根据实参所带的排序规则，为每次函数调用确定一个排序规则，详见第 23.2 节。如果能够成功确定（也就是说，参数之间不存在隐式排序规则冲突），那么所有支持排序规则的参数都会被视为隐式带有该排序规则。这会影响函数中对排序规则敏感的操作的行为。例如，使用上文的 `anyleast` 函数时，

```
SELECT anyleast('abc'::text, 'ABC');
```

的结果将取决于数据库的默认排序规则。在 `C` 区域设置下，结果会是 `ABC`，但在许多其他区域设置下则会是 `abc`。可以在任意参数上添加 `COLLATE` 子句来强制指定要使用的排序规则，例如：

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

另外，如果你希望某个函数无论以什么排序规则调用，都始终按特定排序规则工作，可以在函数定义中按需插入 `COLLATE` 子句。下面这个版本的 `anyleast` 将始终使用 `en_US` 区域设置来比较字符串：

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i] COLLATE "en_US") FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

但请注意，如果把它用于不支持排序规则的数据类型，就会抛出错误。

如果无法在实参之间确定共同的排序规则，那么 SQL 函数会把参数视为带有其数据类型的默认排序规则（通常是数据库的默认排序规则，但对域类型参数来说也可能不同）。

这种支持排序规则的参数的行为，可以看作是仅适用于文本数据类型的一种受限多态形式。

38.6. 函数重载

可以用同样的 SQL 名称定义多于一个函数，只要它们的参数不同即可。换句话说，函数名可以被重载。不管你是否使用它，这种能力都要求在某些用户不信任其他用户的数据库中调用函数时采取安全预防措施，见第 10.3 节。当一个查询被执行时，服务器将从数据类型和所提供的参数个数来决定要调用哪个函数。重载也可用来模拟具有可变参数个数（最大个数有限）的函数。

在创建一个重载函数家族时，应该小心不要创建歧义。例如，给定函数：

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

对于 `test(1, 1.5)` 这样的输入就无法立刻清楚地知道应该调用哪个函数。当前实现的解决规则在第 10 章中有描述，但是设计一个依赖于这种行为的系统是不明智的。

具有单个复合类型参数的函数，通常不应与该类型的任何属性（字段）同名。回想一下，*attribute*(*table*) 被视为等价于 *table*.*attribute*。如果在“作用于复合类型的函数”与“复合类型的属性”之间出现歧义，系统总会优先选择属性。可以通过给函数名加上模式限定（即 *schema*.*func*(*table*)）来覆盖这一选择，但更好的做法仍然是避免使用会冲突的名字。

另一类可能的冲突发生在可变参数函数和非可变参数函数之间。例如，可以同时创建 `foo(numeric)` 和 `foo(VARIADIC numeric[])`。这时，对于只提供一个 numeric 参数的调用（例如 `foo(10.1)`），就不清楚该匹配哪个函数。规则是：优先使用在搜索路径中出现较早的那个函数；如果两者位于同一模式中，则优先使用非可变参数函数。

在重载 C 语言函数时有一个额外的约束：重载函数家族中的每一个函数的 C 名称必须与其他所有函数的 C 名称不同，不管是内部的还是动态载入的。如果这条规则被违背，该行为将不可移植。你可能会得到一个运行时链接器错误，或者这些函数之一将被调用（通常是内部的那一个）。SQL `CREATE FUNCTION` 命令的 `AS` 子句的另一种形式可以把 SQL 函数名和 C 源代码中的函数名分离。例如：

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
    LANGUAGE C;
```

这里的 C 函数名称反映了很多种可能的习惯之一。

38.7. 函数易变性分类

每一个函数都有一个易变性分类，可能的类别是 `VOLATILE`，`STABLE` 或 `IMMUTABLE`。`VOLATILE` 会作为默认类别，条件是CREATE FUNCTION 命令没有指定类别。易变性分类是给优化器的关于该函数行为的一种承诺：

- 一个 `VOLATILE` 函数可以做任何事情，包括修改数据库。在使用相同的参数连续调用时，它能返回不同的结果。优化器不会对这类函数的行为做任何假定。使用易变函数的查询会在需要该函数值的每一行上重新对它求值。

- 一个 STABLE 函数不能修改数据库并且被确保对一个语句中的所有行用给定的相同参数返回相同的结果。这种分类允许优化器把该函数的多个调用优化成一个调用。特别是，在一个索引扫描条件中使用包含这样一个函数的表达式是安全的（因为一次索引扫描只会计算一次比较值，而不是为每一行都计算一次，在一个索引扫描条件中不能使用 VOLATILE 函数）。

- 一个 IMMUTABLE 函数不能修改数据库并且被确保用相同的参数永远返回相同的结果。这种分类允许优化器在一个查询用常量参数调用该函数时提前计算该函数。例如，一个 SELECT ... WHERE x = 2 + 2 这样的查询可以被简化为 SELECT ... WHERE x = 4，因为整数加法操作符底层的函数被标记为IMMUTABLE。

为了最好的优化结果，你应该把函数标记为对它们合法的易变性分类中最严格的那种。

任何带有副作用的函数必须被标记为 VOLATILE，这样对它的调用就不能被优化掉。甚至如果一个函数的值在一个查询中会变化，即使它没有副作用也需要被标记为VOLATILE。这样的示例有 random()、currval()、timeofday() 等。

另一种重要的示例是 current_timestamp 家族的函数有资格被标记为 STABLE，因为它们的值在一个事务中不会改变。

在考虑先规划然后立即执行的简单交互式查询时，在STABLE 和 IMMUTABLE 分类间的区别相对较小：一个函数是在规划时只执行一次还是在查询执行开始期间只执行一次没有太大关系。但是如果计划被保存下来然后在后面被重用，区别就大了。如果把一个实际上并非不可变的函数标记为IMMUTABLE，就可能在规划期间过早将它折叠成常量，导致在后续重用该计划时继续使用陈旧的值。在使用预备语句，或使用会缓存执行计划的函数语言（如 PL/pgSQL）时，这会带来严重问题。

对于用 SQL 或者其他任何标准过程语言编写的函数，还有第二种由易变性分类决定的特性，即由调用该函数的 SQL 命令所作的数据修改的可见性。VOLATILE 函数将看到这些更改，STABLE 或者 IMMUTABLE 函数则看不到。这种行为使用 MVCC 的快照行为（见第 13 章）实现：STABLE 和 IMMUTABLE 函数使用一个在调用查询开始时建立的快照，而 VOLATILE 函数在它们执行的每一个查询的开始都获得一个新鲜的快照。

> **注意**
>
> 用 C 编写的函数按照它们自己需要的方式管理快照，但是通常最好让 C 函数也按照上面的方式来。

由于这种快照行为，只包含 SELECT 命令的函数可以安全地标记为 STABLE，即使它查询的表可能正被并发查询修改。PostgreSQL 会使用为调用查询建立的快照来执行 STABLE 函数中的所有命令，因而该函数在整个查询期间看到的都是数据库的固定视图。

对 IMMUTABLE 函数中的 SELECT 也采用同样的快照行为。一般来说，在 IMMUTABLE 函数里查询数据库表并不明智，因为一旦表内容发生变化，就会破坏其不变性。不过，PostgreSQL 并不会强制禁止这样做。

一种常见的错误是当一个函数的结果依赖于一个配置参数时把它标记为 IMMUTABLE。例如，一个操纵时间戳的函数有可能结果依赖于TimeZone 设置。为了安全起见，这类函数应该被标记为 STABLE。

> **注意**
>
> PostgreSQL 要求 STABLE 和 IMMUTABLE 函数中不包含非 SELECT 的 SQL 命令以阻止数据修改（这也不是完全万无一失，因为这类函数还可以调用修改数据库的VOLATILE 函数。如果那样做，你将发现该STABLE 或 IMMUTABLE 函数不会发现由被调用函数所作的数据库改变，因为它们对它的快照不可见）。

38.8. 过程语言函数

PostgreSQL 允许用除 SQL 和 C 之外的语言编写用户定义的函数。这些语言通常被称为过程语言（PL）。过程语言并不内置在 PostgreSQL 服务器中，它们通过可装载模块提供。更多信息请见第 42 章以及接下来的章节。

38.9. 内部函数

内部函数由 C 编写并且已经被静态链接到 PostgreSQL 服务器中。该函数定义的“主体”指定该函数的 C 语言名称，它不必与为 SQL 使用而声明的名称相同。（出于向后兼容的原因，也接受空主体，此时会认为 C 语言函数名与 SQL 函数名相同。）

通常，所有存在于服务器中的内部函数都在数据库集簇的初始化（见第 18.2 节）期间被声明，但是用户可以使用 CREATE FUNCTION 为一个内部函数创建额外的别名。在 CREATE FUNCTION 中用语言名 internal 来声明内部函数。例如，要为 sqrt 函数创建一个别名：

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

（大部分内部函数应该被声明为“严格”）。

> **注意**
>
> 并非所有“预定义”函数在上述意义上都是“内部”函数。有些预定义的函数由 SQL 编写。

38.10. C 语言函数

用户定义的函数可以用 C 编写（或者可以与 C 兼容的语言，例如 C++）。这类函数被编译成动态载入对象（也被称为共享库）并且由服务器在需要时载入。动态载入是把“C 语言”函数和“内部”函数区分开的特性 – 两者真正的编码习惯实际上是一样的（因此，标准内部函数库为用户定义的 C 函数提供了丰富的编码示例）。

当前仅有一种调用约定被用于 C 函数（“版本 1”）。如下文所示，为函数编写一个 PG_FUNCTION_INFO_V1() 宏就能指示对该调用约定的支持。

38.10.1. 动态载入

在一个会话中第一次调用一个特定可载入目标文件中的用户定义函数时，动态载入器会把那个目标文件载入到内存以便该函数被调用。因此用户定义的 C 函数的CREATE FUNCTION 必须为该函

数指定两块信息：可载入目标文件的名称，以及要在该目标文件中调用的特定函数的 C 名称（链接符号）。如果没有显式指定 C 名称，则它被假定为和 SQL 函数名相同。

下面的算法被用来基于 CREATE FUNCTION 命令中给定的名称来定位共享目标文件：

1. 如果名称是一个绝对路径，则载入给定的文件。
2. 如果该名称以字符串 $libdir 开始，那么这一部分会被 PostgreSQL 包的库目录名（在编译时确定）替换。
3. 如果该名称不包含目录部分，会在配置变量dynamic_library_path指定的路径中搜索该文件。
4. 否则（在该路径中没找到该文件，或者它包含一个非绝对目录），动态载入器将尝试接受给定的名称，这很可能会失败（依赖当前工作目录是不可靠的）。

如果这个序列不起作用，会把平台相关的共享库文件名扩展（通常是 .so）追加到给定的名称并且再次尝试上述的过程。如果还是失败，则载入失败。

建议通过相对于 $libdir 的路径，或者通过动态库路径来定位共享库。这样一来，如果新安装位于不同的位置，版本升级会更简单。$libdir 实际代表的目录可以通过命令 pg_config --pkglibdir 查出。

用于运行 PostgreSQL 服务器的用户 ID 必须能够通过要载入文件的路径。常见的错误是把文件或更高层的目录变得对 postgres 用户不可读或者不可执行。

在任何情况下，CREATE FUNCTION 命令中给定的文件名会被原封不动地记录在系统目录中，这样如果需要再次载入该文件则会应用同样的过程。

> **注意**
>
> PostgreSQL 不会自动编译 C 函数。在从 CREATE FUNCTION 命令中引用目标文件之前，它必须先被编译好。更多信息请见第 38.10.5 节。

为确保动态装载的目标文件不会被加载到不兼容的服务器中，PostgreSQL 会检查该文件是否包含内容正确的“魔数块”。这样服务器就能检测出明显的不兼容情况，例如代码是针对另一个 PostgreSQL 主版本编译的。要加入魔数块，请在模块某一个（且只能一个）源文件中，在包含头文件 fmgr.h 之后写入：

```
PG_MODULE_MAGIC;
```

在第一次使用之后，动态载入目标文件会保留在内存中。在同一个会话中，后续对该文件中函数的调用只需付出一次很小的符号表查找开销。如果需要强制重新载入一个目标文件（例如在重新编译之后），就需要开启一个新的会话。

动态装载的文件可以选择包含初始化和终结函数。如果文件包含一个名为 _PG_init 的函数，该函数会在文件装载后立即被调用。该函数不接收参数，并且应返回 void。如果文件包含一个名为 _PG_fini 的函数，该函数会在文件卸载前立即被调用。同样，该函数不接收参数，并且应返回 void。注意，_PG_fini 只会在卸载文件时被调用，而不会在进程终止时被调用。（目前，卸载已被禁用，永远不会发生，但将来可能会改变。）

38.10.2. C 语言函数中的基础类型

要了解如何编写 C 语言函数，你需要了解 PostgreSQL 如何在内部表达基础类型以及如何与函数传递它们。在内部，PostgreSQL 把基础类型视为一块“内存数据块”。你为该类型定义的用户自定义函数，决定了 PostgreSQL 如何操作它。也就是说，PostgreSQL 只负责把数据存到磁盘、再从磁盘取回，而数据的输入、处理和输出则依赖你定义的这些函数。

基础类型可以有三种内部格式之一：

- 传值，定长
- 传引用，定长
- 传引用，变长

传值类型在长度上只能是 1、2 或 4 字节（如果你的机器上 `sizeof(Datum)` 是 8，则还有 8 字节）。你应当小心地定义你的类型以便它们在所有的架构上都是相同的尺寸（字节）。例如，`long` 类型很危险，因为它在某些机器上是 4 字节但在另外一些机器上是 8 字节，而`int` 类型在大部分 Unix 机器上都是 4 字节。在 Unix 机器上 `int4` 类型的一种合理实现可能是：

```
/* 4 字节整数，传值 */
typedef int int4;
```

（实际的 PostgreSQL C 代码会把这种类型称为 `int32`，因为 C 中的习惯是`int`*XX* 表示 *XX* 位。注意因此还有尺寸为 1 字节的 C 类型 `int8`。SQL 类型 `int8` 在 C 中被称为 `int64`。另见表 38.1）。

另一方面，任意尺寸的定长类型都可以通过引用传递。例如，这里有一种 PostgreSQL 类型的实现示例：

```
/* 16 字节结构，传引用 */
typedef struct
{
    double  x, y;
} Point;
```

在 PostgreSQL 函数中传进或传出这种类型时，只能使用指向这种类型的指针。要返回这样一种类型的值，用 `palloc` 分配正确的内存量，然后填充分配好的内存，并且返回一个指向该内存的指针（还有，如果只想返回与具有相同数据类型的一个输入参数相同的值，可以跳过额外的 `palloc` 并且返回指向该输入值的指针）。

最后，所有变长类型必须也以引用的方式传递。所有变长类型必须用一个正好 4 字节的不透明长度字段开始，该字段会由`SET_VARSIZE` 设置，绝不要直接设置该字段！所有要被存储在该类型中的数据必须在内存中接着该长度字段的后面存储。长度字段包含该结构体的总长度，也就是包括长度字段本身的尺寸。

另一个重点是要避免在数据类型值中留下未被初始化的位。例如，要注意把可能存在于结构体中的任何对齐填充字节置零。如果不这样做，你的数据类型的逻辑等价常量可能会被规划器认为是不等的，进而导致低效的（不过还是正确的）计划。

> **警告**
>
> 绝不要修改通过引用传递的输入值的内容。如果这样做很可能会破坏磁盘上的数据，因为给出的指针可能直接指向一个磁盘缓冲区。这条规则唯一的例外在第 38.11 节中有解释。

例如，我们可以这样定义类型text：

```
typedef struct {
    int32 length;
    char data[FLEXIBLE_ARRAY_MEMBER];
} text;
```

[FLEXIBLE_ARRAY_MEMBER] 记号表示数据部分的实际长度不由该声明指定。

在操作变长类型时，我们必须小心分配正确数量的内存，并正确设置长度字段。例如，如果我们想在一个text 结构体中存储 40 字节，我们可以使用这样的代码片段：

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

VARHDRSZ 和 sizeof(int32) 一样，但是用宏VARHDRSZ 来引用变长类型的额外开销的尺寸被认为是比较好的风格。还有，必须使用 SET_VARSIZE 宏来设置长度字段，而不是用简单的赋值来设置。

表 38.1 展示了 PostgreSQL 中许多内置 SQL 数据类型所对应的 C 类型。“定义文件”列给出了需要包含的头文件，以获取类型定义。（实际的定义可能位于所列文件包含的其他文件中。建议用户坚持使用已定义的接口。）请注意，在服务器代码的任何源文件中，都应该始终首先包含 postgres.h，因为它声明了很多你反正都会用到的内容，而且先包含其他头文件可能会带来可移植性问题。

表 38.1. 内置 SQL 类型等效的 C 类型

SQL 类型	C 类型	定义文件
abstime	AbsoluteTime	utils/nabstime.h
boolean	bool	postgres.h（可能是编译器内置）
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	（编译器内置）
character	BpChar*	postgres.h
cid	CommandId	postgres.h

SQL 类型	C 类型	定义文件
date	DateADT	utils/date.h
float4 (real)	float4	postgres.h
float8 (double precision)	float8	postgres.h
int2 (smallint)	int16	postgres.h
int4 (integer)	int32	postgres.h
int8 (bigint)	int64	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
numeric	Numeric	utils/numeric.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	RegProcedure	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp	datatype/timestamp.h
timestamp with time zone	TimestampTz	datatype/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

了解了基础类型所有可能的结构后，就可以看一些实际函数示例。

38.10.3. 版本 1 的调用约定

版本-1 的调用约定依赖于宏来屏蔽传递参数和结果的大部分复杂性。版本-1 函数的 C 声明总是：

```
Datum funcname(PG_FUNCTION_ARGS)
```

此外，宏调用：

```
PG_FUNCTION_INFO_V1(funcname);
```

必须出现在同一个源文件中（按惯例会正好写在该函数本身之前）。这种宏调用不是 internal 语言函数所需要的，因为 PostgreSQL 会假定所有内部函数都使用版本-1 调用约定。不过，对于动态载入函数是必需的。

在版本-1 函数中，每一个实参都使用对应于该参数数据类型的PG_GETARG_*xxx*() 宏取得。（在非严格的函数中，需要使用PG_ARGISNULL() 对参数是否为空值提前做检查；见下文。）结果要用对应于返回类型的 PG_RETURN_*xxx*() 宏返回。PG_GETARG_*xxx*() 的参数是要取得的函数参数的编号，从零开始计。PG_RETURN_*xxx*() 的参数是实际要返回的值。

这里是一些使用版本-1 调用约定的示例：

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

PG_MODULE_MAGIC;

/* by value */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* by reference, fixed length */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* The macros for FLOAT8 hide its pass-by-reference nature. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Here, the pass-by-reference nature of Point is not hidden. */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* by reference, variable length */

PG_FUNCTION_INFO_V1(copytext);

Datum
```

```
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_PP(0);

    /*
     * VARSIZE_ANY_EXHDR is the size of the struct in bytes, minus the
     * VARHDRSZ or VARHDRSZ_SHORT of its header.  Construct the copy with a
     * full-length header.
     */
    text     *new_t = (text *) palloc(VARSIZE_ANY_EXHDR(t) + VARHDRSZ);
    SET_VARSIZE(new_t, VARSIZE_ANY_EXHDR(t) + VARHDRSZ);

    /*
     * VARDATA is a pointer to the data region of the new struct.  The source
     * could be a short datum, so retrieve its data through VARDATA_ANY.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA_ANY(t), /* source */
           VARSIZE_ANY_EXHDR(t));   /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_PP(0);
    text  *arg2 = PG_GETARG_TEXT_PP(1);
    int32 arg1_size = VARSIZE_ANY_EXHDR(arg1);
    int32 arg2_size = VARSIZE_ANY_EXHDR(arg2);
    int32 new_text_size = arg1_size + arg2_size + VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA_ANY(arg1), arg1_size);
    memcpy(VARDATA(new_text) + arg1_size, VARDATA_ANY(arg2), arg2_size);
    PG_RETURN_TEXT_P(new_text);
}
```

假定上述代码已经准备在文件 funcs.c 中并且被编译成一个共享对象，我们可以用这样的命令在PostgreSQL 中定义函数：

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'DIRECTORY/funcs', 'add_one'
     LANGUAGE C STRICT;

-- note overloading of SQL function name "add_one"
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'DIRECTORY/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'DIRECTORY/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'DIRECTORY/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'DIRECTORY/funcs', 'concat_text'
     LANGUAGE C STRICT;
```

这里，*DIRECTORY* 表示共享库文件的目录（例如 PostgreSQL 的教程目录，它包含这一节中用到的示例的代码）。（更好的风格是先把*DIRECTORY* 放入搜索路径，在AS 子句中只使用'funcs'。在任何情况下，我们可以为一个共享库省略系统相关的扩展名，通常是.so）。

注意我们已经把函数指定为“strict”，这意味着如果有任何输入值为空值，系统应该自动假定结果为空值。通过这种做法，我们避免在函数代码中检查空值输入。如果不这样做，我们必须使用 PG_ARGISNULL() 明确地检查空值输入。

宏 PG_ARGISNULL(*n*) 允许一个函数测试是否每一个输入为空值（当然，只需要在没有声明为“strict”的函数中这样做）。和 PG_GETARG_*xxx*() 宏一样，输入参数也是从零开始计数。注意应该在验证了一个参数不是空值之后才执行PG_GETARG_*xxx*()。要返回空值结果，应执行 PG_RETURN_NULL()，它对严格的以及非严格的函数都有用。

乍看之下，与普通的 C 调用约定相比，版本 1 的编码约定似乎只是增加了复杂度。不过，它们确实让我们能够处理可为 NULL 的参数和返回值，以及“经过 TOAST 处理”（压缩或行外）的值。

在版本-1 接口中提供的其他选项是 PG_GETARG_*xxx*() 宏的两个变种。其中的第一种是 PG_GETARG_*xxx*_COPY()，它确保返回的指定参数的拷贝可以被安全地写入（通常的宏有时会返回一个指向表中物理存储值的指针，不能写入该值。使用PG_GETARG_*xxx*_COPY() 宏可以保证得到一个可写的结果）。第二种变种 PG_GETARG_*xxx*_SLICE() 宏有三个参数。第一个是函数参数的编号（如上文）。第二个和第三个是要被返回的段的偏移量和长度。偏移量从零开始计算，而负值的长度则表示要求返回该值的剩余部分。当大型值的存储类型为“external”时，这些宏提供了访问这些大型值的部分内容的更有效方法（列的存储类型可以使用ALTER TABLE *tablename* ALTER COLUMN *colname* SET STORAGE *storagetype* 来指定。*storagetype* 取 plain、external、extended 或者 main）。

最后，版本-1 的函数调用约定可以返回集合结果（第 38.10.8 节）、实现触发器函数（第 39 章）和过程语言调用处理器（第 56 章）。更多细节可见源代码发布中的 src/backend/utils/fmgr/README。

38.10.4. 编写代码

在开始更高级的话题之前，我们应该讨论一下用于 PostgreSQL C 语言函数的编码规则。虽然有可能把不是 C 编写的函数载入到 PostgreSQL 中，但即使能够做到，通常也很困难，因为其他语言（例如 C++、FORTRAN 或者 Pascal）通常不会遵循和 C 相同的调用约定。也就是说，其他语言不会以同样的方式在函数之间传递参数以及返回值。由于这个原因，我们会假定你的 C 语言函数确实是用 C 编写的。

编写和构建 C 语言函数的基本规则如下：

- 使用 `pg_config --includedir-server` 查出 PostgreSQL 服务器头文件在你的系统上（或你的用户将要运行的系统上）安装于何处。
- 为了让你的代码能够被 PostgreSQL 动态装入，编译和链接时总是需要特殊的选项。关于如何在特定操作系统上完成这件事，详见第 38.10.5 节。
- 记得按照第 38.10.1 节中的说明，为你的共享库定义一个“魔数块”。
- 分配内存时，使用 PostgreSQL 提供的 `palloc` 和 `pfree`，而不是相应的 C 库函数 `malloc` 和 `free`。用 `palloc` 分配的内存会在每个事务结束时自动释放，从而避免内存泄漏。
- 总是使用 `memset` 将结构体的所有字节清零（或者一开始就用 `palloc0` 来分配它们）。即使你给结构体的每个字段都赋了值，结构体中仍可能存在包含垃圾值的对齐填充字节（也就是结构体中的空洞）。如果不这么做，就很难支持 hash 索引或哈希连接，因为那时你必须只挑出数据结构中真正有意义的位来计算 hash 值。规划器有时也依赖按位相等来比较常量，因此如果逻辑上等价的值在按位上不相等，就可能得到不理想的规划结果。
- PostgreSQL 的大多数内部类型都在 `postgres.h` 中声明，而函数管理器接口（`PG_FUNCTION_ARGS` 等）位于 `fmgr.h` 中，因此至少需要包含这两个文件。出于可移植性考虑，最好把 `postgres.h` 放在最前面，先于任何其他系统或用户头文件。包含 `postgres.h` 时，也会顺带为你包含 `elog.h` 和 `palloc.h`。
- 目标文件中定义的符号名不能彼此冲突，也不能与 PostgreSQL 服务器可执行文件中定义的符号冲突。如果你收到这类错误消息，就必须重命名相关函数或变量。

38.10.5. 编译和链接动态装载函数

在你能够使用以 C 编写的 PostgreSQL 扩展函数之前，必须以特殊方式对它们进行编译和链接，以生成一个可由服务器动态装载的文件。更准确地说，需要创建一个共享库。

若想了解本节未涵盖的信息，你应阅读操作系统的文档，特别是 C 编译器 `cc` 和链接编辑器 `ld` 的手册页。此外，PostgreSQL 源代码在 `contrib` 目录中包含若干可用的示例。不过，如果你依赖这些示例，就会使你的模块依赖于 PostgreSQL 源代码是否可用。

创建共享库通常与链接可执行文件类似：先把源文件编译为目标文件，再把目标文件链接在一起。目标文件需要以位置无关代码（PIC）形式生成。从概念上讲，这意味着当它们被可执行文件装载时，可以放在内存中的任意位置。（面向可执行文件的目标文件通常不会这样编译。）链接共享库的命令中也包含一些特殊标志，用来把它与链接可执行文件的命令区分开来（至少理论上如此，某些系统上的实际做法要丑陋得多）。

在下面的示例中，我们假定你的源代码位于文件 `foo.c` 中，并将创建共享库 `foo.so`。除非另有说明，中间目标文件名为 `foo.o`。共享库可以包含多个目标文件，但这里我们只使用一个。

FreeBSD

生成 PIC 的编译器选项是 `-fPIC`。创建共享库时使用的编译器选项是 `-shared`。

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

这从 FreeBSD 3.0 版起适用。

HP-UX

生成 PIC 的系统编译器选项是 `+z`。使用 GCC 时则是 `-fPIC`。用于共享库的链接器选项是 `-b`。因此：

```
cc +z -c foo.c
```

或者：

```
gcc -fPIC -c foo.c
```

然后：

```
ld -b -o foo.sl foo.o
```

HP-UX 使用 `.sl` 作为共享库扩展名，这与大多数其他系统不同。

Linux

生成 PIC 的编译器选项是 `-fPIC`。创建共享库的编译器选项是 `-shared`。完整示例如下：

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

macOS

下面是一个示例。它假定开发者工具已经安装。

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

生成 PIC 的编译器选项是 `-fPIC`。对于 ELF 系统，使用带有 `-shared` 选项的编译器来链接共享库。对于较早的非 ELF 系统，则使用 `ld -Bshareable`。

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

生成 PIC 的编译器选项是 `-fPIC`。使用 `ld -Bshareable` 来链接共享库。

```
gcc -fPIC -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

生成 PIC 的编译器选项，对于 Sun 编译器是 `-KPIC`，而 `-fPIC` 则用于 GCC。链接共享库时，两种编译器都可以使用 `-G`，或者也可以改用 `-shared`，但这只适用于 GCC。

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

或者

```
gcc -fPIC -c foo.c
gcc -G -o foo.so foo.o
```

> **提示**
>
> 如果这些内容对你来说过于复杂，可以考虑使用 GNU Libtool[1]，它通过统一接口隐藏了平台差异。

生成的共享库文件随后就可以装载到 PostgreSQL 中。在向 `CREATE FUNCTION` 命令指定文件名时，必须给出共享库文件名，而不是中间目标文件名。请注意，系统标准的共享库扩展名（通常是 `.so` 或 `.sl`）可以在 `CREATE FUNCTION` 命令中省略，并且通常也应省略，以获得最佳可移植性。

关于服务器期望在何处找到共享库文件，请回头参见第 38.10.1 节。

38.10.6. 复合类型参数

复合类型没有像 C 结构体那样的固定布局。复合类型的实例可能包含空值字段。此外，继承层次中的复合类型可能具有和同一继承层次中其他成员不同的字段。因此，PostgreSQL 提供了函数接口以便从 C 访问复合类型的字段。

[1] https://www.gnu.org/software/libtool/

假设我们想要写一个函数来回答查询：

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
    FROM emp
    WHERE name = 'Bill' OR name = 'Sam';
```

如果使用版本-1 的调用约定，我们可以定义 c_overpaid 为：

```
#include "postgres.h"
#include "executor/executor.h"  /* 用于 GetAttributeByName() */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
    /* 另外，我们可能更想对空 salary 用 PG_RETURN_NULL() 。*/

    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

GetAttributeByName 是 PostgreSQL 的一个系统函数，用于从指定行中取出属性。它有三个参数：传入函数的 HeapTupleHeader 类型参数、所需属性的名称，以及一个用于指示该属性是否为 null 的返回参数。GetAttributeByName 返回一个 Datum 值，你可以用适当的 DatumGet*XXX*() 宏把它转换为正确的数据类型。注意，如果 null 标志被设置，那么返回值本身没有意义；在尝试对结果做任何处理之前，务必先检查这个 null 标志。

也有 GetAttributeByNum 函数，它可以用目标属性的列号而不是属性名来选择目标属性。

下面的命令声明 SQL 中的 c_overpaid：

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
    AS 'DIRECTORY/funcs', 'c_overpaid'
    LANGUAGE C STRICT;
```

注意我们用了 STRICT，这样我们不需要检查输入参数是否为 NULL。

38.10.7. 返回行（复合类型）

要从 C 语言函数中返回一行或一个复合类型值，可以使用一套特殊的 API，它通过一组宏和函数隐藏了构造复合数据类型时的大部分复杂性。要使用这套 API，源文件中必须包含：

```
#include "funcapi.h"
```

构造复合数据值（下文简称“元组”）有两种方式：一种是从 Datum 值数组构造，另一种是从 C 字符串数组构造，这些字符串会传给该元组各列数据类型的输入转换函数。无论采用哪种方式，首先都需要获取或构造描述该元组结构的 `TupleDesc`。处理 Datum 时，需要把 `TupleDesc` 传给 `BlessTupleDesc`，然后为每一行调用 `heap_form_tuple`。处理 C 字符串时，则要把 `TupleDesc` 传给 `TupleDescGetAttInMetadata`，然后为每一行调用 `BuildTupleFromCStrings`。对于返回元组集合的函数，这些准备步骤可以在第一次调用函数时一次性完成。

有一些辅助函数可以用来设置所需的 `TupleDesc`。在大多数返回复合值的函数中，推荐的做法是调用：

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

传入调用函数本身收到的同一个 `fcinfo` 结构体（这当然要求使用版本 1 调用约定）。`resultTypeId` 可以指定为 `NULL`，也可以指定为一个本地变量的地址，用于接收函数结果类型的 OID。`resultTupleDesc` 应当是一个本地 `TupleDesc` 变量的地址。检查返回结果是否为 `TYPEFUNC_COMPOSITE`；如果是，`resultTupleDesc` 就会被填入所需的 `TupleDesc`。（如果不是，可以报告一个类似 “function returning record called in context that cannot accept type record” 的错误。）

> **提示**
>
> `get_call_result_type` 能够解析一个多态函数结果的实际类型，因此不仅在返回复合类型的函数中，在返回标量多态结果的函数中它也是非常有用的。`resultTypeId` 输出主要用于返回多态标量的函数。

> **注意**
>
> `get_call_result_type` 有一个兄弟 `get_expr_result_type`，它被用来解析被表示为一棵表达式树的函数调用的输出类型。在尝试从函数自身外部确定结果类型时可以用它。也有一个`get_func_result_type`，当只有函数的 OID 可用时可以用它。不过这些函数无法处理被声明为返回`record` 的函数，并且`get_func_result_type` 无法解析多态类型，因此你应该优先使用`get_call_result_type`。

更早、现在已被弃用的获取TupleDesc 的函数有：

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

它可以为一个指定名称的关系的行类型得到 TupleDesc，还有：

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

可以基于一个类型 OID 得到 TupleDesc。这可以被用来为一种基本或者复合类型获得 TupleDesc。不过，对于返回record 的函数它不起作用，并且它无法解析多态类型。

一旦有了一个 TupleDesc，如果计划处理 Datum，可以调用：

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

如果计划处理 C 字符串，可调用：

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

如果正在编写一个返回集合的函数，你可以把这些函数的结果保存在 FuncCallContext 结构体中 – 分别使用 tuple_desc 或者 attinmeta 字段。

在处理 Datum 时，使用

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

来用 Datum 形式的用户数据构建一个 HeapTuple。

在处理 C 字符串时，使用

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

来用 C 字符串形式的用户数据构建一个 HeapTuple。*values* 是一个 C 字符串数组，每一个元素是返回行的一个属性。每一个 C 字符串应该是该属性数据类型的输入函数所期望的格式。为了对一个属性返回空值，*values* 数组中对应的指针应该被设置为 NULL。对于你返回的每一行都需要再次调用这个函数。

一旦已经构建了一个要从函数中返回的元组，它必须被转换成一个 Datum。使用

```
HeapTupleGetDatum(HeapTuple tuple)
```

可把一个 HeapTuple 转换成合法的 Datum。如果你只想返回一行，那么这个Datum 可以被直接返回，在一个集合返回函数中它也可以被当做当前的返回值。

下一节中会有一个示例。

38.10.8. 返回集合

C 语言函数有两个返回集合（多行）的选项。在一种称为 ValuePerCall 模式的方法中，一个集合返回函数被重复调用（每次传递相同的参数），并在每次调用时返回一个新行，直到没有更多行

要返回并且通过返回 NULL 来表示这一点。因此，集合返回函数 (SRF) 必须在调用之间保存足够的状态以记住它在做什么并在每次调用时返回正确的下一项。在另一种称为 Materialize 模式的方法中，SRF 填充并返回一个包含其整个结果的 tuplestore 对象；这样整个结果只需一次调用即可返回，且不需要跨调用状态。

使用 ValuePerCall 模式时，重要的是要记住查询不能保证运行完成；也就是说，由于诸如LIMIT 之类的选项，执行器可能会在获取所有行之前停止调用集合返回函数。这意味着在最后一次调用中执行清理活动是不安全的，因为这可能永远不会发生。对于需要访问外部资源（例如文件描述符）的函数，建议使用 Materialize 模式。

本节的其余部分记录了一组使用 ValuePerCall 模式的 SRF 常用（尽管不是必须使用）的辅助宏。有关 Materialize 模式的其他详细信息可以在 src/backend/utils/fmgr/README 中找到。此外，PostgreSQL 源代码分发中的 contrib 模块包含许多使用 ValuePerCall 和 Materialize 模式的 SRF 示例。

要使用此处描述的 ValuePerCall 支持宏，请包含funcapi.h。这些宏与结构体 FuncCallContext 一起使用，该结构体包含需要跨调用保存的状态。在使用这些宏的 SRF 中，fcinfo->flinfo->fn_extra 用于在调用之间保存指向 FuncCallContext 的指针。宏在第一次使用时自动填充该字段，并期望在后续使用中找到相同的指针。

```
typedef struct FuncCallContext
{
    /*
     * 本次调用以前已经被调用过多少次
     *
     * SRF_FIRSTCALL_INIT() 会为你把 call_cntr 初始化为 0，
     * 并且在每次调用 SRF_RETURN_NEXT() 时增加。
     */
    uint64 call_cntr;

    /*
     * 可选：最大调用次数
     *
     * 这里的 max_calls 只是为了方便，设置它是可选的。
     * 如果没有设置，你必须提供替代的方法来了解函数什么时候做完。
     */
    uint64 max_calls;

    /*
     * 可选：指向结果槽的指针
     *
     * 此字段已过时，仅为向后兼容而保留，即兼容使用已弃用的
     * TupleDescGetSlot() 的用户定义 SRF。
     */
    TupleTableSlot *slot;

    /*
     * 可选：指向用户提供的上下文信息的指针
     *
     * user_fctx 是一个指向你自己的数据的指针，它可用来在函数的多次
     * 调用之间保存任意的上下文信息。
     */
    void *user_fctx;

    /*
     * 可选：指向包含属性类型输入元数据的结构的指针
```

```
     *
     * attinmeta 被用在返回元组（即复合数据类型）时，在返回基础类型
     * 时不会使用。只有想用BuildTupleFromCStrings()创建返回元组时才需要它。
     */
    AttInMetadata *attinmeta;

    /*
     * 用于保存必须在多次调用间都存在的结构的内存上下文
     *
     * SRF_FIRSTCALL_INIT() 会为你设置 multi_call_memory_ctx，并且由
     * SRF_RETURN_DONE() 来清理。对于任何需要在 SRF 的多次调用间都
     * 存在的内存来说，它是最合适的内存上下文。
     */
    MemoryContext multi_call_memory_ctx;

    /*
     * 可选：指向包含元组描述的结构的指针
     *
     * tuple_desc 被用在返回元组（即复合数据类型）时，并且只有在用
     * heap_form_tuple() 而不是 BuildTupleFromCStrings() 构建元组时才需要它。
     * 注意这里存储的 TupleDesc 指针通常已经被先运行过 BlessTupleDesc()。
     */
    TupleDesc tuple_desc;

} FuncCallContext;
```

使用此基础结构的 SRF 将使用的宏是：

```
SRF_IS_FIRSTCALL()
```

来判断你的函数是否是第一次被调用。在第一次调用时（只能在第一次调用时）使用：

```
SRF_FIRSTCALL_INIT()
```

初始化 `FuncCallContext`。在每次函数调用时，包含第一次，调用：

```
SRF_PERCALL_SETUP()
```

设置使用 `FuncCallContext`。

如果你的函数在当前调用中有数据要返回，请使用：

```
SRF_RETURN_NEXT(funcctx, result)
```

把它返回给调用者（`result` 必须是类型 `Datum`，可以是一个单一值或者按上文所述准备好的元组）。最后，当函数完成了数据返回后，可使用：

```
SRF_RETURN_DONE(funcctx)
```

来清理并且结束 SRF。

调用 SRF 时当前所处的内存上下文是一个瞬时上下文，它会在两次调用之间被清空。这意味着，你不必对用 `palloc` 分配的所有东西都调用 `pfree`，因为它们反正会自动释放。

不过，如果你需要分配在多次调用之间持续存在的数据结构，就必须把它们放到别处。对于任何需要一直存活到 SRF 运行结束的数据，`multi_call_memory_ctx` 所指向的内存上下文就是合适的位置。在大多数情况下，这意味着你应当在做首次调用初始化时切换到 `multi_call_memory_ctx`。可以使用 `funcctx->user_fctx` 保存指向这类跨调用数据结构的指针。（在 `multi_call_memory_ctx` 中分配的数据会在查询结束时自动消失，因此同样无需手工释放。）

> **警告**
>
> 虽然函数的实参在多次调用之间保持不变，但如果在瞬时上下文中反 TOAST 了参数（通常由 `PG_GETARG_`*xxx* 宏完成），那么被反 TOAST 的拷贝将在每次循环中被释放。相应地，如果你把这些值的引用保存在`user_fctx` 中，你也必须在反 TOAST 之后把它们拷贝到 `multi_call_memory_ctx` 中，或者确保你只在那个上下文中反 TOAST 这些值。

一个完整的伪代码示例：

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* 这里是一次性设置代码： */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }

    /* 这里是每一次都要做的设置代码： */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* 这里只是一种测试是否执行完的方法： */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* 这里返回另一个项： */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* 这里已经完成了项的返回，所以只报告事实。 */
        /* （不要将清理代码放在这里。） */
        SRF_RETURN_DONE(funcctx);
    }
}
```

一个返回复合类型的简单 SRF 的完整示例如下：

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;
```

```
/* 仅在函数的第一次调用时执行的操作 */
if (SRF_IS_FIRSTCALL())
{
    MemoryContext   oldcontext;

    /* 为跨调用持久性创建函数上下文 */
    funcctx = SRF_FIRSTCALL_INIT();

    /* 切换到适合多次函数调用的内存上下文 */
    oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

    /* 要返回的元组总数 */
    funcctx->max_calls = PG_GETARG_UINT32(0);

    /* 为我们的结果类型构建一个元组描述符 */
    if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                 errmsg("function returning record called in context "
                        "that cannot accept type record")));

    /*
     * 生成后续从原始C字符串生成元组所需的属性元数据
     */
    attinmeta = TupleDescGetAttInMetadata(tupdesc);
    funcctx->attinmeta = attinmeta;

    MemoryContextSwitchTo(oldcontext);
}

/* 每次函数调用时执行的操作 */
funcctx = SRF_PERCALL_SETUP();

call_cntr = funcctx->call_cntr;
max_calls = funcctx->max_calls;
attinmeta = funcctx->attinmeta;

if (call_cntr < max_calls)    /* 当还有更多要发送时执行 */
{
    char       **values;
    HeapTuple    tuple;
    Datum        result;

    /*
     * 为构建返回的元组准备一个值数组。
     * 这应该是一个由后续类型输入函数处理的C字符串数组。
     */
    values = (char **) palloc(3 * sizeof(char *));
    values[0] = (char *) palloc(16 * sizeof(char));
    values[1] = (char *) palloc(16 * sizeof(char));
    values[2] = (char *) palloc(16 * sizeof(char));

    snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
    snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
    snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

    /* 构建一个元组 */
    tuple = BuildTupleFromCStrings(attinmeta, values);

    /* 将元组转换为 datum */
    result = HeapTupleGetDatum(tuple);

    /* 清理（这实际上并不是必要的） */
```

```
        pfree(values[0]);
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* 当没有更多要发送时执行 */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

在 SQL 中声明此函数的一种方法是：

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

另一种方法是使用 OUT 参数：

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

请注意，采用这种方法时，函数的输出类型在形式上是一个匿名的 `record` 类型。

38.10.9. 多态参数和返回类型

可以声明 C 语言函数来接受和返回多态类型 `anyelement`、`anyarray`、`anynonarray`、`anyenum` 和 `anyrange`。关于多态函数的更详细解释，见第 38.2.5 节。当函数参数或者返回类型被定义为多态类型时，函数的编写者无法提前知道会用什么数据类型调用该函数或者该函数需要返回什么数据类型。在`fmgr.h` 中提供了两种例程来允许版本-1 的 C 函数发现其参数的实际数据类型以及它要返回的类型。这些例程被称为 `get_fn_expr_rettype(FmgrInfo *flinfo)` 和 `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)`。它们返回结果或者参数的类型的 OID，或者当该信息不可用时返回 `InvalidOid`。结构体 `flinfo` 通常被当做 `fcinfo->flinfo` 访问。参数 `argnum` 则是从零开始计。`get_call_result_type` 也可被用作 `get_fn_expr_rettype` 的一种替代品。还有 `get_fn_expr_variadic`，它可以被用来找出可变参数是否已经被合并到了一个数组中。这主要用于 `VARIADIC "any"` 函数，因为对于接收普通数组类型的可变参数函数来说总是会发生这类合并。

例如，假设我们想要写一个接收一个任意类型元素并且返回一个该类型的一维数组的函数：

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* 得到提供的元素，小心它为 NULL 的情况 */
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

    /* 只有一个维度 */
    ndims = 1;
    /* 和一个元素 */
    dims[0] = 1;
    /* 且下界是 1 */
    lbs[0] = 1;

    /* 得到该元素类型所需的信息 */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* 现在构建数组 */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

下面的命令在 SQL 中声明了 `make_array` 函数：

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE C IMMUTABLE;
```

还有一种只对 C 语言函数可用的多态变体：它们可以被声明为接受 "any" 类型的参数。（注意，这个类型名必须用双引号括起来，因为它同时也是 SQL 保留字。）它与 `anyelement` 类似，但不会要求不同的 "any" 参数必须是同一种类型，也不会帮助确定函数的结果类型。C 语言函数还可以把最后一个参数声明为 `VARIADIC "any"`。这可以匹配一个或多个任意类型的实参（不必是同一种类型）。这些参数不会像普通可变参数函数那样被收集成一个数组，而是会单独传给函数。使用这种特性时，必须结合 `PG_NARGS()` 宏以及前面介绍的方法来确定实

参的个数和类型。此外，这种函数的用户也可能希望在函数调用中使用 VARIADIC 关键字，以便让函数把数组元素当作独立参数处理。如果希望支持这种行为，函数本身就必须在使用 get_fn_expr_variadic 检测到实参被标记为 VARIADIC 后自行实现它。

38.10.10. 转换函数

某些函数调用可以在规划期间根据函数特有的属性进行简化。例如，int4mul(n, 1) 可以直接简化为 n。要定义这类针对特定函数的优化，请编写一个转换函数，并将其 OID 放入主函数的 protransform 字段中，该字段位于其 pg_proc 项中。转换函数必须具有 SQL 签名 protransform(internal) RETURNS internal。参数实际上是 FuncExpr *，它是一个表示主函数调用的占位节点。如果转换函数对表达式树的分析证明，可以用简化的表达式树替代它所表示的所有可能的具体调用，就构建并返回该简化表达式。否则，返回一个 NULL 指针（不是 SQL 空值）。

我们不保证 PostgreSQL 在转换函数可以简化的情况下绝不会调用主函数。请确保简化后的表达式与实际调用主函数严格等价。

目前，出于安全考虑，这项功能没有在 SQL 层面开放给用户，因此实际只能用于优化内置函数。

38.10.11. 共享内存与 LWLocks

附加模块可以在服务器启动时预留 LWLock 和一块共享内存。附加模块的共享库必须通过在以下参数中指定它来预装载：shared_preload_libraries。共享内存可以通过下面的调用来预留：

```
void RequestAddinShmemSpace(int size)
```

该调用应放在你的 _PG_init 函数中。

LWLock 可以通过下面的调用来预留：

```
void RequestNamedLWLockTranche(const char *tranche_name, int num_lwlocks)
```

该调用应放在 _PG_init 中。这样可以确保一个包含 num_lwlocks 个 LWLock 的数组可通过以下名称访问：tranche_name。使用 GetNamedLWLockTranche 可以取得指向该数组的指针。

为避免可能的竞争条件，每个后端都应使用 LWLock `AddinShmemInitLock` 来连接和初始化分配给它的共享内存，如下所示：

```
static mystruct *ptr = NULL;

if (!ptr)
{
        bool    found;

        LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
        ptr = ShmemInitStruct("my struct name", size, &found);
        if (!found)
        {
                initialize contents of shmem area;
                acquire any requested LWLocks using:
                ptr->locks = GetNamedLWLockTranche("my tranche name");
        }
        LWLockRelease(AddinShmemInitLock);
}
```

38.10.12. 把 C++ 用于可扩展性

尽管 PostgreSQL 后端是用 C 编写的，只要遵循下面的指导方针也可以用 C++ 编写扩展：

- 所有被后端访问的函数必须对后端呈现一种 C 接口，然后这些 C 函数调用 C++ 函数。例如，被后端访问的函数要求`extern C` 链接。对需要在后端和 C++ 代码之间作为指针传递的任何函数也要这样做。
- 使用合适的释放方法释放内存。例如，大部分后端内存是通过 `palloc()` 分配的，所以应使用 `pfree()` 来释放。在这种情况中使用 C++ 的 `delete` 会失败。
- 防止异常传播到 C 代码中（在所有 `extern C` 函数的顶层使用一个捕捉全部异常的块）。即使 C++ 代码不会显式地抛出任何异常也需要这样做，因为类似内存不足等事件仍会抛出异常。任何异常都必须被捕捉并且用适当的错误传回给 C 接口。如果可能，用 `-fno-exceptions` 来编译 C++ 以完全消灭异常。在这种情况下，你必须在 C++ 代码中检查失败，例如检查`new()` 返回的 NULL。
- 如果从 C++ 代码调用后端函数，确保 C++ 调用栈只包含传统 C 风格的数据结构（POD）。这是必要的，因为后端错误会产生远距离的`longjmp()`，它无法正确地展开具有非 POD 对象的 C++ 调用栈。

总之，最好把 C++ 代码放在与后端交互的`extern C` 函数之后，并且避免异常、内存和调用栈泄漏。

38.11. 用户定义的聚合

PostgreSQL 中的聚合函数是依据状态值和状态转移函数来定义的。也就是说，聚合通过一个状态值来工作，并在处理每个后续输入行时更新该状态值。要定义一个新的聚合函数，需要为状态值选定一种数据类型、为状态选定一个初始值，并指定一个状态转移函数。状态转移函数接受先前的状态值以及当前行的聚合输入值，并返回一个新的状态值。如果聚合的期望结果不同于运行状态值中需要保存的数据，还可以指定一个最终函数。最终函数接受结束时的状态值，并返回所需的聚合结果。原则上，转移函数和最终函数都只是普通函数，因此也可以在聚合上下文之外使用。（实际上，出于性能原因，创建只能在作为聚合一部分被调用时工作的专用转移函数通常更有帮助。）

因此，除了聚合用户所看到的参数和结果数据类型之外，还存在一种内部状态值数据类型，它可能既不同于参数类型，也不同于结果类型。

如果我们定义一个不使用最终函数的聚合，那么得到的就是一个对每一行的列值进行逐步计算的聚合。sum 就是这种聚合的一个示例。sum 从零开始，并始终把当前行的值加到运行总和中。例如，如果我们希望让sum 聚合能用于复数数据类型，那么只需要该数据类型的加法函数。聚合定义如下：

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);
```

它可以这样使用：

```
SELECT sum(a) FROM test_complex;

   sum
-----------
 (34,53.9)
```

（注意，这里依赖了函数重载：名为sum 的聚合不止一个，但PostgreSQL 能够判断对于complex 类型的列该使用哪一种 sum。）

如果没有非空输入值，上述sum 定义将返回零（即初始状态值）。也许我们更希望在这种情况下返回空值 – SQL 标准要求 sum 这样做。实现这一点只需省略 initcond 短语，使初始状态值为空值。通常这意味着sfunc 需要检查输入的状态值是否为空值。但对于 sum 以及 max、min 这类其他简单聚合，只需把第一个非空输入值放入状态变量，然后从第二个非空输入值开始应用转移函数即可。如果初始状态值为空值，并且转移函数被标记为“strict”（即不会对空值输入调用），PostgreSQL 会自动这样做。

“strict” 转移函数的另一项默认行为是，只要遇到空值输入，就会保留先前的状态值不变。因此，空值会被忽略。如果你需要对空值输入采用其他行为，就不要把转移函数声明为 strict，而应在函数代码中自行检测空值输入并完成所需处理。

avg（平均值）是一个更复杂的聚合示例。它需要两项运行状态：输入值之和以及输入值的个数。最终结果通过将这两者相除得到。平均值通常通过使用数组作为状态值来实现。例如，内置的avg(float8) 实现如下：

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

> **注意**
>
> float8_accum 需要一个三元素数组，而不只是两个元素，因为它累积的不仅是输入值的和与计数，还有平方和。这样它除了可用于avg 之外，也可用于其他一些聚合。

SQL 中的聚合函数调用允许使用 DISTINCT 和 ORDER BY 选项，以控制哪些行会传递给聚合的转移函数，以及传递顺序。这些选项由系统在内部实现，并不是聚合支持函数需要关心的事情。

更多细节请参见 CREATE AGGREGATE 命令。

38.11.1. 移动聚合模式

聚合函数可以选择支持移动聚合模式，这样在帧起点会移动的窗口中执行聚合函数时，速度可以显著提高。（关于将聚合函数作为窗口函数使用的信息，见第 3.5 节和第 4.2.8 节。）其基本思想是，除了常规的“前向”状态转移函数外，聚合还提供一个逆向状态转移函数，这样当某些行离开窗口帧时，就可以把它们从聚合的运行状态值中移除。例如，sum 聚合若使用加法作为前向状态转移函数，就会使用减法作为逆向状态转移函数。如果没有逆向状态转移函数，每次帧起点移动时，窗口函数机制都必须从头重新计算聚合，运行时间就与输入行数乘以平均帧长度成正比。有了逆向状态转移函数，运行时间就只与输入行数成正比。

逆向状态转移函数接收当前状态值，以及当前状态中所包含的最早那一行的聚合输入值。它必须重建这样一个状态值：就好像给定的输入行从未参与过聚合，只有其后的各行被聚合一样。这有时要求前向状态转移函数保存比普通聚合模式所需更多的状态。因此，移动聚合模式与普通模式采用完全独立的实现：它有自己的状态数据类型、自己的前向状态转移函数，以及在需要时自己的最终函数。如果不需要额外状态，这些也可以与普通模式的数据类型和函数相同。

作为示例，我们可以把上面给出的sum 聚合扩展为支持移动聚合模式：

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)',
    msfunc = complex_add,
    minvfunc = complex_sub,
    mstype = complex,
    minitcond = '(0,0)'
);
```

名称以 m 开头的参数定义了移动聚合的实现。除了逆向状态转移函数 minvfunc 之外，它们分别对应于不带m 的普通聚合参数。

用于移动聚合模式的前向状态转移函数不允许返回空值作为新的状态值。如果逆向状态转移函数返回空值，就表示该逆向函数无法针对这个特定输入逆转状态计算，因此当前帧起始位置的聚合计算会从头重做。这一约定使得移动聚合模式可用于这样一些场景：在少数不常见的情况下，难以从运行状态值中逆向消除某个输入的影响。遇到这些情况时，逆向状态转移函数可以“放弃处理”，只要它在大多数情况下都能工作，总体上仍然划算。例如，一个处理浮点数的聚合，可能会在必须从运行状态值中移除NaN（非数字）输入时选择放弃处理。

在编写移动聚合支持函数时，务必确保逆向状态转移函数能够精确地重建正确的状态值。否则，是否使用移动聚合模式就可能导致用户可见的结果差异。一个看似很容易添加逆向状态转移函数、但实际上无法满足这一要求的聚合示例，是对float4 或 float8 输入求和的 sum。对 sum(float8) 的一种朴素的声明可能是

```
CREATE AGGREGATE unsafe_sum (float8)
(
    stype = float8,
    sfunc = float8pl,
    mstype = float8,
    msfunc = float8pl,
    minvfunc = float8mi
);
```

然而，这个聚合给出的结果可能与没有逆向状态转移函数时截然不同。例如，考虑

```
SELECT
  unsafe_sum(x) OVER (ORDER BY n ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)
FROM (VALUES (1, 1.0e20::float8),
             (2, 1.0::float8)) AS v (n,x);
```

这个查询把 0 作为第二个结果返回，而预期答案是1。原因在于浮点值的精度有限：把 1 加到 1e20 上，结果仍然是1e20，因此再从中减去1e20 得到的是 0，而不是1。注意，这是浮点算术本身的一般局限，不是PostgreSQL 的限制。

38.11.2. 多态和可变参数聚合

聚合函数可以使用多态的状态转移函数或最终函数，这样同一组函数就能用于实现多个聚合。关于多态函数的解释，请参见第 38.2.5 节。再进一步，聚合函数自身也可以指定为具有多态输入类型和状态类型，从而让同一个聚合定义服务于多种输入数据类型。下面是一个多态聚合的示例：

```
CREATE AGGREGATE array_accum (anycompatible)
(
    sfunc = array_append,
    stype = anycompatiblearray,
    initcond = '{}'
);
```

对于任何给定的聚合调用，其实际状态类型都是一种数组类型，该数组以实际输入类型为元素类型。这个聚合的行为是把所有输入串接成这种类型的一个数组。（注意：内置聚合array_agg 提供了类似的功能，而且性能比这个定义更好。）

下面是以两种不同的实际数据类型作为参数时的输出：

```
SELECT attrelid::regclass, array_accum(attname)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |              array_accum
---------------+---------------------------------------
 pg_tablespace | {spcname,spcowner,spcacl,spcoptions}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid::regtype)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |        array_accum
---------------+---------------------------
 pg_tablespace | {name,oid,aclitem[],text[]}
(1 row)
```

通常，具有多态结果类型的聚合函数也会像上例一样具有多态状态类型。这是必要的，因为否则最终函数就无法以合理方式声明：它会需要多态结果类型，却没有多态参数类型，而 `CREATE FUNCTION` 会以无法从调用中推断结果类型为由拒绝这种声明。不过，有时使用多态状态类型并不方便。最常见的情况是聚合支持函数要用 C 编写，而状态类型应声明为`internal`，因为在 SQL 层面并没有与之对应的类型。为了解决这种情况，可以把最终函数声明为接受额外的 “dummy” 参数，这些参数与聚合的输入参数相匹配。由于调用最终函数时没有可用的具体值，这些假参数总是以 NULL 值传入。它们唯一的作用，是让多态最终函数的结果类型与聚合的输入类型关联起来。例如，内置聚合`array_agg` 的定义等效于：

```
CREATE FUNCTION array_agg_transfn(internal, anynonarray)
  RETURNS internal ...;
CREATE FUNCTION array_agg_finalfn(internal, anynonarray)
  RETURNS anyarray ...;

CREATE AGGREGATE array_agg (anynonarray)
(
    sfunc = array_agg_transfn,
    stype = internal,
    finalfunc = array_agg_finalfn,
    finalfunc_extra
);
```

这里，`finalfunc_extra` 选项指定最终函数除了状态值之外，还会接收与聚合输入参数对应的额外假参数。额外的`anynonarray` 参数使 `array_agg_finalfn` 的声明合法。

与普通函数很类似，可以通过把聚合函数的最后一个参数声明为`VARIADIC` 数组，使其接受可变数量的参数；见第 38.5.5 节。聚合的转移函数也必须将同样的数组类型作为其最后一个参数。这些转移函数通常也会被标记为 `VARIADIC`，但这并非严格要求。

注意

可变参数聚合很容易与 ORDER BY 选项（见第 4.2.7 节）一起被误用，因为在这种组合中，解析器无法判断是否给出了数量错误的实际参数。请记住，ORDER BY 右侧的所有内容都是排序键，而不是聚合的参数。例如，在

```
SELECT myaggregate(a ORDER BY a, b, c) FROM ...
```

中，解析器会把它看作一个聚合函数参数和三个排序键。然而，用户本来可能是想写

```
SELECT myaggregate(a, b, c ORDER BY a) FROM ...
```

如果 myaggregate 是可变参数聚合，那么这两种调用都可能完全合法。

因此，在创建名称相同但常规参数个数不同的聚合函数之前，最好三思。

38.11.3. 有序集聚合

到目前为止，我们所描述的都是“普通”聚合。PostgreSQL 还支持有序集聚合，它与普通聚合有两个关键区别。第一，除了对每个输入行都求值一次的普通聚合参数之外，有序集聚合还可以拥有只在每次聚合操作中求值一次的“直接”参数。第二，普通聚合参数的语法会显式指定它们的排序顺序。有序集聚合通常用于实现依赖特定行顺序的计算，例如排名或百分位点，因此这种排序顺序是任何调用都必须具备的一部分。例如，内置的percentile_disc 定义等效于：

```
CREATE FUNCTION ordered_set_transition(internal, anyelement)
  RETURNS internal ...;
CREATE FUNCTION percentile_disc_final(internal, float8, anyelement)
  RETURNS anyelement ...;

CREATE AGGREGATE percentile_disc (float8 ORDER BY anyelement)
(
    sfunc = ordered_set_transition,
    stype = internal,
    finalfunc = percentile_disc_final,
    finalfunc_extra
);
```

这个聚合接受一个 float8 直接参数（百分位分数）以及一个可以是任意可排序数据类型的聚合输入。它可以这样用来获取家庭收入的中位数：

```
SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_disc
-----------------
           50489
```

这里，0.5 是一个直接参数；如果百分位分数在不同行之间变化，那就没有意义了。

与普通聚合不同，有序集聚合的输入行排序不是由系统在内部完成的，而是由聚合的支持函数负责。典型的实现方法是在聚合的状态值中保存一个“tuplesort”对象的引用，把输入行送入该对象，然后在最终函数中完成排序并读出数据。这样的设计使最终函数可以执行特殊操作，例如向

待排序数据中注入额外的“假想”行。普通聚合通常可以使用由 PL/pgSQL 或其他 PL 语言编写的支持函数来实现，但有序集聚合通常必须用 C 编写，因为它们的状态值无法定义为任何 SQL 数据类型。（在上面的示例中，注意状态值被声明为`internal` 类型，这很典型。）另外，由于最终函数执行了排序，之后就不可能再通过再次执行转移函数来继续加入输入行。这意味着最终函数不是`READ_ONLY`；在 CREATE AGGREGATE 中，必须将其声明为`READ_WRITE`，或者如果额外的最终函数调用能够利用已经排好序的状态，则声明为`SHAREABLE`。

有序集聚合的状态转移函数接收当前状态值以及每一行的聚合输入值，并返回更新后的状态值。这与普通聚合的定义相同，但要注意，不会提供直接参数（如果有的话）。最终函数接收最后的状态值、直接参数的值（如果有），以及（如果指定了`finalfunc_extra`）与聚合输入对应的 NULL 值。与普通聚合一样，只有当聚合是多态的时，`finalfunc_extra` 才真正有用；这时需要这些额外的假参数来把最终函数的结果类型与聚合的输入类型关联起来。

目前，有序集聚合不能用作窗口函数，因此也就无须支持移动聚合模式。

38.11.4. 部分聚合

聚合函数还可以选择支持部分聚合。部分聚合的思想是，在输入数据的不同子集上分别独立运行聚合的状态转移函数，然后把这些子集产生的状态值合并起来，得到与一次性扫描全部输入时相同的状态值。这种模式可用于并行聚合，让不同的工作进程扫描表的不同部分。每个工作进程都会产生一个部分状态值，最后再把这些状态值合并起来，得到最终状态值。（未来，这种模式也可能用于合并本地表和远程表上的聚合结果等用途，但目前尚未实现。）

要支持部分聚合，聚合定义必须提供一个合并函数。它接收该聚合状态类型的两个值（表示分别在两组输入行子集上进行聚合的结果），并生成一个新的同类型值，表示对这两组行合并后进行聚合本应得到的状态。至于这两组输入行的相对顺序，则是未指定的。这意味着，对输入行顺序敏感的聚合通常无法定义出有用的合并函数。

举个简单的例子，`MAX` 和 `MIN` 聚合可以通过把合并函数指定为与其转移函数相同的“两者取较大值”或“两者取较小值”比较函数，来支持部分聚合。`SUM` 聚合则只需要把加法函数用作合并函数。（同样，除非状态值比输入数据类型更宽，否则合并函数与转移函数相同。）

合并函数很像一种转移函数，只不过它的第二个参数接受的是状态类型的值，而不是底层输入类型的值。特别是，处理空值和 strict 函数的规则与转移函数类似。另外，如果聚合定义指定了非空的`initcond`，要记住它不仅会用作每次部分聚合运行的初始状态，也会用作合并函数的初始状态；合并函数会被调用，把每个部分结果都合并到该状态中。

如果聚合的状态类型声明为 `internal`，那么合并函数必须负责确保其结果分配在聚合状态值所使用的正确内存上下文中。这尤其意味着，当第一个输入为`NULL` 时，不能简单返回第二个输入，因为那个值位于错误的上下文中，生命周期也不够长。

当聚合的状态类型声明为 `internal` 时，通常也应在聚合定义中提供序列化函数和反序列化函数，以便能把这种状态值从一个进程复制到另一个进程。如果没有这些函数，就无法进行并行聚合，而未来诸如本地/远程聚合之类的应用也很可能无法工作。

序列化函数必须接受一个 `internal` 类型参数，并返回一个`bytea` 类型结果，用来表示打包成扁平字节块的状态值。反过来，反序列化函数负责逆转这种转换。它必须接受`bytea` 和 `internal` 类型的两个参数，并返回一个`internal` 类型结果。（第二个参数不会被使用，且总是零，但出于类型安全考虑必须保留。）与合并函数的结果不同，反序列化函数的结果不需要长期存在，因此只需在当前内存上下文中分配即可。

还要注意，若要让一个聚合能够并行执行，必须将聚合本身标记为`PARALLEL SAFE`。系统不会参考其支持函数上的并行安全性标记。

38.11.5. 聚合的支持函数

用 C 编写的函数可以通过调用 AggCheckCallContext 来检测自己是否作为聚合支持函数被调用，例如：

```
if (AggCheckCallContext(fcinfo, NULL))
```

进行这种检查的一个原因是：如果结果为真，第一个输入必然是一个临时状态值，因此可以安全地原地修改，而不必分配新的副本。示例可见int8inc()。（虽然聚合转移函数总是允许原地修改状态值，但通常不鼓励聚合最终函数这样做；如果它确实这样做，则必须在创建聚合时声明这种行为。更多细节见 CREATE AGGREGATE。）

AggCheckCallContext 的第二个参数可用于取得保存聚合状态值的内存上下文。这对于希望把“展开”对象（见第 38.12.1 节）用作状态值的转移函数很有用。第一次调用时，转移函数应返回一个展开对象，其内存上下文是聚合状态上下文的子上下文；之后的调用则持续返回同一个展开对象。示例见array_append()。（array_append() 并不是任何内置聚合的转移函数，但它的写法保证了在用作自定义聚合的转移函数时也能高效工作。）

另一个可供用 C 编写的聚合函数使用的支持例程是 AggGetAggref，它返回定义该聚合调用的 Aggref 解析节点。这主要对有序集聚合有用，因为它们可以检查Aggref 节点的子结构，以找出自己应实现的排序顺序。示例可见PostgreSQL 源代码中的 orderedsetaggs.c。

38.12. 用户定义的类型

如第 38.2 节中所述，PostgreSQL 可以扩展以支持新的数据类型。本节描述如何定义新的基础类型，也就是在SQL 语言层之下定义的数据类型。创建新的基础类型需要用底层语言（通常是 C）实现操作该类型的函数。

本节中的示例位于源码分发包的 src/tutorial 目录中的 complex.sql 和 complex.c。关于如何运行这些示例，请参见该目录中的README 文件。

用户定义类型必须始终具有输入函数和输出函数。这些函数决定该类型如何以字符串形式出现（供用户输入和向用户输出），以及该类型在内存中如何组织。输入函数接受一个以空字符结尾的字符串作为参数，并返回该类型的内部（内存中）表示。输出函数接受该类型的内部表示作为参数，并返回一个以空字符结尾的字符串。如果我们希望该类型除了存储之外还能做别的事情，就必须提供额外的函数来实现我们希望该类型支持的各种操作。

假设我们要定义一种表示复数的类型 complex。在内存中表示复数的一种自然方式是下面这个 C 结构体：

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

我们需要把它做成按引用传递的类型，因为它太大了，无法放进单个 Datum 值中。

作为该类型的外部字符串表示，我们选择形如(x,y) 的字符串。

输入函数和输出函数通常都不难编写，尤其是输出函数。但是在定义该类型的外部字符串表示时，要记住，最终你必须为这种表示编写一个完整而健壮的解析器，作为输入函数。例如：

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for type %s: \"%s\"",
                        "complex", str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

输出函数可以简单地写成：

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = psprintf("(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

应当注意让输入函数和输出函数互为逆运算。如果不是这样，当你需要把数据转储到文件中再读回时，就会遇到严重问题。这在涉及浮点数时尤为常见。

可选地，用户定义类型还可以提供二进制输入和输出例程。二进制 I/O 通常比文本 I/O 更快，但可移植性较差。与文本 I/O 一样，外部二进制表示的精确定义完全由你决定。大多数内置数据类型都尽量提供与机器无关的二进制表示。对于`complex`，我们将借助类型`float8`的二进制 I/O 转换器：

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

一旦我们写好了 I/O 函数并将它们编译进共享库，就可以在 SQL 中定义 `complex` 类型。首先将其声明为一种 shell 类型：

```
CREATE TYPE complex;
```

这会建立一个占位符，使我们可以在定义其 I/O 函数时引用该类型。现在我们可以定义这些 I/O 函数：

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;
```

最后，我们可以给出该数据类型的完整定义：

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

当你定义一种新的基础类型时，PostgreSQL 会自动提供对该类型数组的支持。该数组类型通常与基础类型同名，只是在前面加一个下划线字符（`_`）。

一旦该数据类型存在，我们就可以声明额外的函数，为该数据类型提供有用的操作。随后可以在这些函数之上定义操作符；如果需要，还可以创建操作符类以支持该数据类型的索引。这些附加层会在后续各节中讨论。

如果数据类型的内部表示是可变长度的，则这种内部表示必须遵循可变长度数据的标准布局：前四个字节必须是一个从不直接访问的`char[4]` 字段（惯例上命名为 `vl_len_`）。必须使用 `SET_VARSIZE()` 宏在该字段中存储该 datum 的总大小（包括长度字段本身），并使用 `VARSIZE()` 取回它。（这些宏之所以存在，是因为长度字段可能会随平台不同而采用编码形式。）

更多细节见 CREATE TYPE 命令的说明。

38.12.1. TOAST 注意事项

如果你的数据类型的值在内部形式上的大小可变，通常最好让该数据类型支持 TOAST（见第 69.2 节）。即使这些值总是小到不需要压缩或外部存储，也应该这样做，因为 TOAST 还可以通过减少头部开销来为小数据节省空间。

为了支持 TOAST 存储，操作该数据类型的 C 级函数必须始终使用PG_DETOAST_DATUM 小心地对传给它们的任何经过 TOAST 处理的值进行解包。（这一细节通常通过定义特定于该类型的 GETARG_DATATYPE_P 宏来隐藏。）然后，在执行 CREATE TYPE 命令时，把内部长度指定为 variable，并选择某个不同于plain 的合适存储选项。

如果数据对齐并不重要（无论只是对某个特定函数而言，还是因为该数据类型本来就指定了字节对齐），那么就有可能避免 PG_DETOAST_DATUM 的一部分开销。你可以改用 PG_DETOAST_DATUM_PACKED（通常通过定义 GETARG_DATATYPE_PP 宏来隐藏），并使用 VARSIZE_ANY_EXHDR 和 VARDATA_ANY 宏访问一个可能采用打包形式的 datum。再次注意，即使数据类型定义指定了对齐方式，这些宏返回的数据也不是对齐的。如果对齐很重要，就必须使用常规的PG_DETOAST_DATUM 接口。

> **注意**
>
> 较旧的代码常把 vl_len_声明为 int32 字段，而不是char[4] 字段。只要结构体定义中还有其他至少按 int32 对齐的字段，这样做是可以的。但在处理可能未对齐的 datum 时使用这种结构体定义就很危险；编译器可能据此假定该 datum 实际上是对齐的，从而在对齐要求严格的体系结构上导致核心转储。

支持 TOAST 带来的另一个特性，是可以拥有一种比磁盘上存储的格式更便于处理的展开内存数据表示。常规或“扁平”的 varlena 存储格式归根结底只是一块字节数据；例如，它不能包含指针，因为它可能被复制到内存中的其他位置。对于复杂数据类型，处理扁平格式的代价可能相当高，因此PostgreSQL 提供了一种办法，把扁平格式“展开”成更适合计算的表示形式，然后在该数据类型的各个函数之间以这种格式在内存中传递。

要使用展开存储，数据类型必须定义一种遵循 src/include/utils/expandeddatum.h 中规则的展开格式，并提供函数把扁平 varlena 值“展开”为展开格式，再把展开格式“压平”回常规 varlena 表示。然后要确保该数据类型的所有 C 函数都能接受这两种表示，必要时可在收到参数后立即把一种转换成另一种。这并不要求一次性修正该数据类型的全部现有函数，因为标准 PG_DETOAST_DATUM 宏被定义为会把展开输入转换成常规扁平格式。因此，现有那些处理扁平 varlena 格式的函数，即使效率略低一些，也仍能处理展开输入；除非更好的性能很重要，否则不必转换它们。

能够处理展开表示的 C 函数通常分为两类：只能处理展开格式的，以及既能处理展开输入也能处理扁平 varlena 输入的。前者更容易编写，但总体上可能效率较低，因为为了让单个函数使用展开格式而把扁平输入转换成展开形式，其代价可能比在展开格式上操作所节省的还要多。只需处理展开格式时，可以把将扁平输入转换为展开形式的过程隐藏在参数提取宏内部，这样函数看起来不会比处理传统 varlena 输入的函数更复杂。要同时处理这两类输入，可以编写一个参数提取函数，对外部、短头部和压缩的 varlena 输入执行去 TOAST 化，但对展开输入则不做处理。这样的函数可以定义为返回一个指向联合体的指针，该联合体包含扁平 varlena 格式与展开格式。调用者可以使用 VARATT_IS_EXPANDED_HEADER() 宏判断收到的是哪种格式。

TOAST 基础设施不仅允许区分常规 varlena 值和展开值，还能区分指向展开值的“可读写（read-write）”和“只读（read-only）”指针。只需要查看展开值，或者只会以安全且在语

义上不可见的方式修改它的 C 函数，不必关心收到的是哪种指针。那些会生成输入值修改版本的 C 函数，如果收到可读写指针，可以原地修改展开输入值；但如果收到只读指针，则不得修改输入，此时必须先复制该值，生成一个新的可修改值。构造了新展开值的 C 函数应始终返回指向它的可读写指针。另外，原地修改可读写展开值的 C 函数如果在中途失败，应注意让该值保持在合理状态。

有关如何处理展开值的示例，请参见标准数组基础设施，尤其是 `src/backend/utils/adt/array_expanded.c`。

38.13. 用户定义的操作符

每个操作符都是对执行实际工作的底层函数调用的一种“语法糖”；因此，你必须先创建底层函数，才能创建操作符。不过，操作符并不仅仅是语法糖，因为它还携带一些额外信息，可帮助查询规划器优化使用该操作符的查询。下一节将专门解释这些附加信息。

PostgreSQL 支持左一元、右一元和二元操作符。操作符可以被重载；也就是说，同一个操作符名可以用于不同的操作符，而这些操作符具有不同数量和类型的操作数。执行查询时，系统会根据提供的操作数数量和类型确定应调用哪个操作符。

下面是为两个复数相加创建一个操作符的示例。我们假设已经创建了 `complex` 类型的定义（见第 38.12 节）。首先需要一个完成实际工作的函数，然后才能定义该操作符：

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    function = complex_add,
    commutator = +
);
```

现在我们就可以执行下面这样的查询：

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

上面展示了如何创建一个二元操作符。要创建一元操作符，只需省略 `leftarg`（左一元操作符）或 `rightarg`（右一元操作符）之一。`function` 子句和参数子句是 `CREATE OPERATOR` 中唯一必需的项。示例中出现的 `commutator` 子句，则是给查询优化器的一个可选提示。关于 `commutator` 以及其他优化器提示的更多细节，会在下一节说明。

38.14. 操作符优化信息

PostgreSQL 的操作符定义可以包含若干可选子句，用来告诉系统该操作符行为的一些有用信息。只要适用，就应当提供这些子句，因为它们可以显著加快使用该操作符的查询执行速度。但如果你提供了这些信息，就必须确保它们是正确的！错误地使用优化子句可能导致查询变慢、输

出出现隐蔽错误，或者引发其他糟糕后果。如果你拿不准，完全可以省略某个优化子句；唯一的后果，只是查询可能会比本来需要的更慢。

未来版本的 PostgreSQL 可能会加入更多优化子句。这里描述的是 11.22 版本所理解的全部子句。

38.14.1. COMMUTATOR

如果给出 COMMUTATOR 子句，它指定一个与正在定义的操作符互为交换子的操作符。若对所有可能的输入值 x、y，都有 (x A y) 等于 (y B x)，则称操作符 A 是操作符 B 的交换子。注意，B 也同样是 A 的交换子。例如，对某种特定数据类型而言，< 和 > 通常互为交换子，而操作符 + 通常与其自身可交换。但操作符 – 通常并不与任何操作符可交换。

一个可交换操作符的左操作数类型，与其交换子的右操作数类型相同，反之亦然。因此，PostgreSQL 只需知道交换子操作符的名称，就可以查找出该交换子，而这也正是 COMMUTATOR 子句中所需提供的全部内容。

对于将用于索引和连接子句的操作符，提供交换子信息至关重要，因为这样查询优化器才能把这类子句“翻转”成不同计划类型所需的形式。例如，考虑这样一个查询，其 WHERE 子句形如 `tab1.x = tab2.y`，其中 `tab1.x` 和 `tab2.y` 都是某种用户定义类型，并假设 `tab2.y` 上建有索引。除非优化器能够确定如何把该子句翻转成 `tab2.y = tab1.x`，否则它就无法生成索引扫描，因为索引扫描机制要求传给它的操作符左侧必须是已建立索引的列。PostgreSQL 不会仅凭假设就认为这种转换有效；= 操作符的创建者必须通过为该操作符标记交换子信息，明确声明这种转换是有效的。

定义一个与自身可交换的操作符时，直接定义即可。但定义一对可交换操作符时，情况就稍微复杂一些：第一个要定义的操作符如何引用另一个尚未定义的操作符呢？这个问题有两种解决办法：

- 一种办法是在定义第一个操作符时省略 COMMUTATOR 子句，然后在第二个操作符的定义中提供该子句。由于 PostgreSQL 知道可交换操作符是成对出现的，因此它在看到第二个定义时，会自动回头补全第一个定义中缺少的 COMMUTATOR 子句。
- 另一种更直接的办法是在两个定义中都包含 COMMUTATOR 子句。当 PostgreSQL 处理第一个定义并发现 COMMUTATOR 引用了一个不存在的操作符时，系统会在系统目录中为该操作符建立一个占位项。这个占位项只有操作符名称、左右操作数类型和结果类型包含有效数据，因为这就是 PostgreSQL 此时能够推断出的全部信息。第一个操作符的目录项将链接到这个占位项。之后，当你定义第二个操作符时，系统会用第二个定义中的附加信息更新该占位项。如果在占位操作符补全之前尝试使用它，就只会得到一条错误消息。

38.14.2. NEGATOR

如果给出 NEGATOR 子句，它指定一个与正在定义的操作符互为求反器的操作符。若操作符 A 和 B 都返回布尔结果，并且对所有可能的输入 x、y，都有 (x A y) 等于 NOT (x B y)，则称 A 是 B 的求反器。注意，B 也同样是 A 的求反器。例如，对大多数数据类型而言，< 和 >= 构成一对求反器。一个操作符永远不可能合法地成为其自身的求反器。

与交换子不同，一对一元操作符完全可能合法地被标记为彼此的求反器；这意味着对所有 x，都有 (A x) 等于 NOT (B x)，或者对右一元操作符有等价的关系。

一个操作符的求反器必须与待定义操作符具有相同的左操作数类型和/或右操作数类型，因此与 COMMUTATOR 一样，在 NEGATOR 子句中只需给出操作符名称即可。

提供求反器对查询优化器很有帮助，因为它允许把 `NOT (x = y)` 这样的表达式简化为 `x <> y`。这种情况比你想象得更常见，因为 NOT 操作可能会作为其他重排的结果被插入进来。

可以使用上面解释的定义交换子对的相同方法，来定义成对的求反器操作符。

38.14.3. RESTRICT

如果给出 RESTRICT 子句，它指定该操作符的限制选择率估算函数。（注意，这里是函数名，而不是操作符名。）RESTRICT 子句只对返回 boolean 的二元操作符有意义。限制选择率估算器的作用，是针对当前操作符和某个特定常量值，猜测一张表中有多少比例的行会满足如下形式的 WHERE 子句条件：

```
column OP constant
```

这会帮助优化器大致了解具有这种形式的 WHERE 子句将淘汰多少行。（你可能会问：如果常量在左边会怎样？嗯，这正是 COMMUTATOR 的作用之一.....）

编写新的限制选择率估算函数远远超出了本章的范围，不过幸运的是，对于你自己的很多操作符，通常都可以直接使用系统提供的某个标准估算器。标准的限制选择率估算器如下：

eqsel 用于 =
neqsel 用于 <>
scalarltsel 用于 <
scalarlesel 用于 <=
scalargtsel 用于 >
scalargesel 用于 >=

对于选择率非常高或非常低的操作符，即使它们实际上并不是真正的相等或不等比较，你也常常可以勉强使用 eqsel 或 neqsel。例如，几何类型中的近似相等操作符就使用 eqsel，其依据是它们通常只会匹配表中很小一部分项。

对于那些能够以某种合理方式转换为数值标量、从而可进行范围比较的数据类型，你可以使用 scalarltsel、scalarlesel、scalargtsel 和 scalargesel。如果可能的话，请把该数据类型加入函数 convert_to_scalar()（位于 src/backend/utils/adt/selfuncs.c）所理解的范围中。（最终，这个函数应被通过 pg_type 系统目录某一列标识的、按数据类型划分的函数所取代；但这件事目前还没有发生。）如果不这样做，系统仍然可以工作，但优化器的估算效果就不会像本可达到的那样好。

在 src/backend/utils/adt/geo_selfuncs.c 中，还为几何操作符提供了其他选择率估算函数：areasel、positionsel 和 contsel。截至目前，这些函数都还只是桩实现，但你也许仍会想使用它们（或者更好的是，改进它们）。

38.14.4. JOIN

如果给出 JOIN 子句，它指定该操作符的连接选择率估算函数。（注意，这里是函数名，而不是操作符名。）JOIN 子句只对返回 boolean 的二元操作符有意义。连接选择率估算器的作用，是针对当前操作符，猜测两张表中有多少比例的行对会满足如下形式的 WHERE 子句条件：

```
table1.column1 OP table2.column2
```

与 RESTRICT 子句一样，这会极大地帮助优化器判断在若干可能的连接顺序中，哪一种预计工作量最小。

与前面一样，本章不会尝试解释如何编写连接选择率估算函数，而只是建议你在适用时使用某个标准估算器：

`eqjoinsel` 用于 =
`neqjoinsel` 用于 <>
`scalarltjoinsel` 用于 <
`scalarlejoinsel` 用于 <=
`scalargtjoinsel` 用于 >
`scalargejoinsel` 用于 >=
`areajoinsel` 用于基于二维面积的比较
`positionjoinsel` 用于基于二维位置的比较
`contjoinsel` 用于基于二维包含关系的比较

38.14.5. HASHES

如果存在 `HASHES` 子句，它会告知系统：在基于该操作符的连接中，可以使用哈希连接方法。`HASHES` 只对返回 `boolean` 的二元操作符有意义；在实践中，该操作符还必须表示某种数据类型或某对数据类型上的相等关系。

哈希连接背后的假设是：只有当左值和右值经 hash 运算得到同一个 hash 码时，连接操作符才有可能返回 true。如果两个值被放进不同的 hash 桶，连接过程就根本不会去比较它们，这实际上隐含假定连接操作符的结果必定为 false。因此，对于那些并不表示某种相等关系的操作符，指定 `HASHES` 永远没有意义。在大多数情况下，只有对两侧都接受同一数据类型的操作符支持 hash 才是现实可行的。不过，有时也可以为两种或更多数据类型设计兼容的 hash 函数；也就是说，尽管这些值的表示不同，但对于“相等”的值，函数仍会生成相同的 hash 码。例如，对不同位宽的整数做到这一点就相当简单。

若要被标记为 `HASHES`，该连接操作符必须出现在某个 hash 索引操作符族中。创建操作符时并不会强制这一点，因为那时引用它的操作符族当然还不存在。但如果没有这样的操作符族存在，运行时尝试在哈希连接中使用该操作符就会失败。系统需要通过该操作符族来查找与该操作符输入数据类型相对应的 hash 函数。当然，在创建该操作符族之前，你还必须先创建合适的 hash 函数。

准备 hash 函数时应格外小心，因为它有一些依赖机器的方式可能会导致结果不正确。例如，如果你的数据类型是某种结构体，其中可能包含无意义的填充位，那就不能简单地把整个结构体传给 `hash_any`。（除非你编写了其他操作符和函数，以确保这些未使用位始终为零，而这正是推荐的策略。）再比如，在符合 IEEE 浮点标准的机器上，负零和正零是不同的值（位模式不同），但它们被定义为比较相等。如果某个浮点值可能包含负零，就需要采取额外步骤，确保它生成与正零相同的 hash 值。

一个可参与哈希连接的操作符，必须拥有一个交换子（如果两侧操作数数据类型相同，则为它自身；如果不同，则应是相关的相等操作符），并且该交换子也必须出现在同一个操作符族中。否则，在使用该操作符时可能会发生规划器错误。另外，对于支持多种数据类型的 hash 操作符族，最好（虽然并非严格必需）为每一种数据类型组合都提供相等操作符；这样可以获得更好的优化效果。

> **注意**
>
> 作为可哈希连接操作符底层实现的函数，必须标记为 immutable 或 stable。如果它是 volatile，系统就永远不会尝试把该操作符用于哈希连接。

> **注意**
>
> 如果某个可哈希连接操作符的底层函数被标记为 strict，那么该函数还必须是完备的：也就是说，对于任意两个非 null 的输入，它都应返回 true 或 false，而绝不能返回 null。如果不遵循这条规则，对 `IN` 操作进行 hash 优化时可能会产生错误结果。（具体来说，`IN` 可能会在按标准本应返回 null 的地方返回 false；或者抛出一条错误，抱怨它没有为 null 结果做好准备。）

38.14.6. MERGES

如果存在 `MERGES` 子句，它会告知系统：在基于该操作符的连接中，可以使用归并连接方法。`MERGES` 只对返回 `boolean` 的二元操作符有意义；在实践中，该操作符还必须表示某种数据类型或某对数据类型上的相等关系。

归并连接的基本思想，是先把左表和右表分别排序，然后同步扫描它们。因此，两种数据类型都必须能够被完全排序，而连接操作符必须只能在那对值位于排序次序中“同一位置”时才成功。实际效果上，这意味着连接操作符必须表现得像相等比较一样。不过，只要两种不同的数据类型在逻辑上兼容，也完全可以对它们进行归并连接。例如，`smallint` 与 `integer` 之间的相等操作符就是可归并连接的。我们只需要能把两种数据类型都带入逻辑兼容排序序列的排序操作符即可。

若要被标记为 `MERGES`，该连接操作符必须作为相等成员出现在某个 `btree` 索引操作符族中。创建操作符时并不会强制这一点，因为那时引用它的操作符族当然还不存在。但除非能找到匹配的操作符族，否则该操作符实际上不会被用于归并连接。因此，`MERGES` 标记的作用，是向规划器提供一个提示，告诉它值得去查找匹配的操作符族。

一个可参与归并连接的操作符，必须拥有一个交换子（如果两侧操作数数据类型相同，则为它自身；如果不同，则应是相关的相等操作符），并且该交换子也必须出现在同一个操作符族中。否则，在使用该操作符时可能会发生规划器错误。另外，对于支持多种数据类型的 `btree` 操作符族，最好（虽然并非严格必需）为每一种数据类型组合都提供相等操作符；这样可以获得更好的优化效果。

> **注意**
>
> 作为可归并连接操作符底层实现的函数，必须标记为 immutable 或 stable。如果它是 volatile，系统就永远不会尝试把该操作符用于归并连接。

38.15. 索引扩展接口

到目前为止所描述的过程使我们能够定义新类型、新函数以及新操作符。然而，我们还不能在一种新数据类型的列上定义索引。为此，必须为该新数据类型定义一个操作符类。本节稍后将用一个示例说明这一概念：为 B-树索引方法定义一个新的操作符类，以便按绝对值升序存储和排序复数。

操作符类可以分组成操作符族，以展示语义兼容的类之间的关系。只涉及单一数据类型时，一个操作符类就已足够，因此我们先关注这种情况，然后再回到操作符族。

38.15.1. 索引方法和操作符类

pg_am 表为每个索引方法（内部称为访问方法）保存一行。对表进行常规访问的支持内置于 PostgreSQL 中，但所有索引方法都在 pg_am 中描述。可以编写必要的代码，然后在 pg_am 中创建一个条目，从而添加新的索引访问方法 – 但这超出了本章的范围（参见第 61 章）。

索引方法的例程并不直接知道它将处理哪些数据类型。相反，一个操作符类标识了索引方法在处理特定数据类型时需要使用的那组操作。之所以称为操作符类，是因为它指定的一项内容就是可与索引一起使用的 WHERE 子句操作符集合（也就是能被转换成索引扫描条件的操作符）。操作符类还可以指定索引方法内部操作所需的一些支持函数，但这些函数并不直接对应任何可与索引一起使用的 WHERE 子句操作符。

可以为同一种数据类型和索引方法定义多个操作符类。这样就能为一种数据类型定义多套索引语义。例如，一个 B-树索引要求为其处理的每一种数据类型定义一种排序顺序。对于复数数据类型，也许既需要一个按复数绝对值排序的 B-树操作符类，也需要另一个按实部排序的操作符类，等等。通常，其中一个操作符类会被视为最常用，并标记为该数据类型在该索引方法上的默认操作符类。

同一个操作符类名可以用于多个不同的索引方法（例如，B-树和 hash 索引方法都有名为 int4_ops 的操作符类），但每一个这样的类都是独立实体，必须分别定义。

38.15.2. 索引方法策略

与操作符类关联的操作符通过“策略号”来标识，用以表示每个操作符在其操作符类上下文中的语义。例如，B-树对键施加了严格的从小到大的顺序，因此像“小于”和“大于等于”这样的操作符，对 B-树来说就很重要。由于 PostgreSQL 允许用户定义操作符，PostgreSQL 不能仅凭操作符名称（例如 < 或 >=）就判断它是哪一类比较。取而代之的是，索引方法定义了一组“策略”，可以把它们看成是广义的操作符。每个操作符类都会说明，对于某种特定数据类型和某种索引语义解释，每一种策略分别对应哪个实际操作符。

B-树索引方法定义了五种策略，如表 38.2 所示。

表 38.2. B-树策略

操作	策略号
小于	1
小于等于	2
等于	3
大于等于	4
大于	5

Hash 索引只支持等值比较，因此它们只使用一种策略，如表 38.3 所示。

表 38.3. Hash 策略

操作	策略号
等于	1

GiST 索引更加灵活：它们根本没有固定的策略集合。相反，每个特定 GiST 操作符类中负责“一致性”检查的支持例程会按自己的方式解释策略号。举例来说，一些内置的 GiST 索引操作符类会为二维几何对象建立索引，并提供“R 树”策略，如表 38.4 所示。其中四个是真正的二维测试（重叠、相同、包含、被包含），四个只考虑 X 方向，另外四个则在 Y 方向上提供相同测试。

表 38.4. GiST 二维 “R 树” 策略

操作	策略号
严格位于左侧	1
不延伸到右侧	2
重叠	3
不延伸到左侧	4
严格位于右侧	5
相同	6
包含	7
被包含	8
不延伸到上方	9
严格位于下方	10
严格位于上方	11
不延伸到下方	12

SP-GiST 索引在灵活性方面与 GiST 索引类似：它们也没有固定的策略集合。相反，每个操作符类的支持例程会根据该操作符类的定义解释策略号。举例来说，内置点操作符类所使用的策略号如表 38.5 所示。

表 38.5. SP-GiST 点策略

操作	策略号
严格位于左侧	1
严格位于右侧	5
相同	6
被包含	8
严格位于下方	10
严格位于上方	11

GIN 索引与 GiST 和 SP-GiST 索引类似，也没有固定的策略集合。相反，每个操作符类的支持例程会根据该操作符类的定义解释策略号。举例来说，内置数组操作符类所使用的策略号如表 38.6 所示。

表 38.6. GIN 数组策略

操作	策略号
重叠	1
包含	2
被包含	3
等于	4

BRIN 索引也与 GiST、SP-GiST 和 GIN 索引一样，没有固定的策略集合。每个操作符类的支持函数都会根据该操作符类的定义解释策略号。举例来说，内置 `Minmax` 操作符类所使用的策略号如表 38.7 所示。

表 38.7. BRIN Minmax 策略

操作	策略号
小于	1
小于等于	2
等于	3
大于等于	4
大于	5

注意，上面列出的所有操作符都返回布尔值。实际上，所有被定义为索引方法搜索操作符的操作符都必须返回 `boolean`，因为要与索引配合使用，它们必须出现在 `WHERE` 子句的顶层。（某些索引访问方法还支持排序操作符，这类操作符通常不返回布尔值；该特性见第 38.15.7 节。）

38.15.3. 索引方法支持例程

仅靠策略信息通常不足以让系统知道如何使用索引。实际上，索引方法还需要额外的支持例程才能工作。例如，B-树索引方法必须能够比较两个键，并判断其中一个是大于、等于还是小于另一个。类似地，hash 索引方法必须能够为键值计算 hash 码。这些操作并不对应 SQL 命令条件中使用的操作符；它们是索引方法内部使用的管理例程。

与策略一样，操作符类会标识对于给定的数据类型和语义解释，应由哪些具体函数承担这些角色。索引方法定义它需要的函数集合，而操作符类则会通过为函数分配由索引方法规定的“支持函数号”来标识正确的函数。

如表 38.8 所示，B-树要求一个比较支持函数，并允许操作符类作者按需再提供两个支持函数。第 63.3 节进一步解释了这些支持函数的要求。

表 38.8. B-树支持函数

函数	支持号
比较两个键，并返回一个小于零、等于零或大于零的整数，用以表示第一个键是小于、等于还是大于第二个键	1
返回可由 C 调用的排序支持函数地址（可选）	2
将测试值与基准值加上或减去某个偏移量后的结果进行比较，并根据比较结果返回真或假（可选）	3

如表 38.9 所示，Hash 索引要求一个支持函数，并允许操作符类作者按需再提供一个支持函数。

表 38.9. Hash 支持函数

函数	支持号
为一个键计算 32 位 hash 值	1
给定一个 64 位盐值，计算一个键的 64 位 hash 值；如果盐值为 0，结果的低 32 位必须与函数 1 计算出的值一致（可选）	2

GiST 索引有九个支持函数，其中两个是可选的，如表 38.10 所示。（详见第 64 章。）

表 38.10. GiST 支持函数

函数	描述	支持号
consistent	确定键是否满足查询条件	1
union	计算一组键的并集	2
compress	计算将被索引的键或值的压缩表示	3
decompress	计算压缩键的解压表示	4
penalty	计算把新键插入具有给定子树键的子树时的罚值	5
picksplit	确定页面中的哪些项要移到新页面，并计算结果页面的并集键	6
equal	比较两个键，并在它们相等时返回真	7
distance	确定键到查询值的距离（可选）	8
fetch	为仅索引扫描计算压缩键的原始表示（可选）	9

如表 38.11 所示，SP-GiST 索引要求五个支持函数。（详见第 65 章。）

表 38.11. SP-GiST 支持函数

函数	描述	支持号
config	提供有关该操作符类的基本信息	1
choose	确定如何把一个新值插入内部元组	2
picksplit	确定如何划分一组值	3
inner_consistent	确定对某个查询需要搜索哪些子分区	4
leaf_consistent	确定键是否满足查询条件	5

如表 38.12 所示，GIN 索引有六个支持函数，其中三个是可选的（详见第 66 章）。

表 38.12. GIN 支持函数

函数	描述	支持号
compare	比较两个键，并返回一个小于零、等于零或大于零的整数，用以表示第一个键是小于、等于还是大于第二个键	1
extractValue	从待索引值中提取键	2
extractQuery	从查询条件中提取键	3
consistent	确定值是否匹配查询条件（布尔变体；如果支持函数 6 存在则可选）	4
comparePartial	比较查询中的部分键与索引中的键，并返回一个小于零、等于零或大于零的整数，用以指示 GIN 应忽略该索引项、将该项视为匹配，还是停止索引扫描（可选）	5
triConsistent	确定值是否匹配查询条件（三值变体；如果支持函数 4 存在则可选）	6

BRIN 索引有四个基本支持函数，如表 38.13 所示；这些基本函数可能要求提供额外的支持函数。（详见第 67.3 节。）

表 38.13. BRIN 支持函数

函数	描述	支持号
`opcInfo`	返回描述被索引列摘要数据的内部信息	1
`add_value`	向一个现有的摘要索引元组增加一个新值	2
`consistent`	确定值是否匹配查询条件	3
`union`	计算两个摘要元组的并集	4

与搜索操作符不同，支持函数返回的是特定索引方法所期望的数据类型；例如，对 B-树的比较函数来说，就是一个有符号整数。每个支持函数的参数个数和类型也同样取决于索引方法。对于 B-树和 hash，比较支持函数和 hash 支持函数接受的输入数据类型，与操作符类中包含的操作符相同；但对大多数 GiST、SP-GiST、GIN 和 BRIN 支持函数来说并非如此。

38.15.4. 示例

现在我们已经了解了这些基本思想，下面给出先前承诺的创建新操作符类示例。（这个可运行示例位于源码发布包中的`src/tutorial/complex.c` 和 `src/tutorial/complex.sql`。）该操作符类封装了一组按绝对值顺序对复数排序的操作符，因此我们把它命名为 `complex_abs_ops`。首先，我们需要一组操作符。定义操作符的过程已经在第 38.13 节中讨论过。对于 B-树上的操作符类，我们需要如下操作符：

- 绝对值小于（策略 1）
- 绝对值小于等于（策略 2）
- 绝对值等于（策略 3）
- 绝对值大于等于（策略 4）
- 绝对值大于（策略 5）

定义一组相关比较操作符时，最不容易出错的方式是先编写 B-树比较支持函数，再把其他函数写成围绕该支持函数的一行包装器函数。这样可以降低在边界情况下得到不一致结果的概率。按照这种方法，我们首先编写：

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double      amag = Mag(a),
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

现在，小于函数如下所示：

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

其他四个函数的区别只在于它们如何比较内部函数的结果与 0。

接着，我们在 SQL 中声明这些函数以及基于它们的操作符：

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'filename', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

指定正确的交换子和求反器操作符很重要，同样也要指定合适的限制选择率函数和连接选择率函数，否则优化器将无法有效地利用索引。

这里还有几点值得注意：

- 只能有一个名为 = 且两个操作数都为 complex 类型的操作符。在这个例子里，我们并没有任何其他 = 操作符可用于 complex；但如果我们是在构造一种实际使用的数据类型，可能会希望 = 表示复数的普通相等，而不是绝对值相等。在那种情况下，我们就需要为 complex_abs_eq 选用其他操作符名。
- 尽管 PostgreSQL 能处理 SQL 名称相同但参数数据类型不同的函数，C 却只能处理给定名称的一个全局函数。因此，我们不应该把 C 函数简单命名成 abs_eq 之类。通常，在 C 函数名中包含数据类型名称是个好习惯，这样就不会与其他数据类型的函数发生冲突。
- 我们原本也可以把该函数的 SQL 名称取为 abs_eq，并依靠 PostgreSQL 通过参数数据类型把它与其他同名 SQL 函数区分开。为了让示例保持简单，这里我们让 C 层和 SQL 层的函数使用相同的名称。

下一步是注册 B-树要求的支持例程。实现该例程的 C 示例代码与操作符函数位于同一个文件中。该函数的声明如下：

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

现在我们已经有了所需的操作符和支持例程，就可以最终创建操作符类：

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

这样就完成了！现在应该可以在 complex 列上创建并使用 B-树索引。

我们本来也可以把操作符项写得更详细一些，例如：

```
        OPERATOR        1       < (complex, complex) ,
```

但是当操作符接受的数据类型与该操作符类所服务的数据类型相同时，就没有必要这样写。

上述示例假定你希望把这个新操作符类设为 complex 数据类型的默认 B-树操作符类。如果不是这样，只需省去 DEFAULT 这个词。

38.15.5. 操作符类和操作符族

到目前为止，我们一直隐含地假定一个操作符类只处理一种数据类型。虽然某个特定的索引列当然只能有一种数据类型，但对把被索引列与另一种数据类型的值进行比较的操作建立索引往往也很有用。此外，如果某个与操作符类相关的跨数据类型操作符有用，通常另一种数据类型本身也会有一个相关的操作符类。把相关类之间的联系显式表示出来会很有帮助，因为这有助于规划

器优化 SQL 查询（尤其是对 B-树操作符类而言，因为规划器中包含大量有关如何使用它们的知识）。

为了满足这些需求，PostgreSQL 使用操作符族这一概念。一个操作符族包含一个或多个操作符类，还可以包含属于整个族、但不属于族中任何单一类的可索引操作符及其相应的支持函数。我们称这样的操作符和函数在该族中是“松散”的，而不是绑定在某个特定类中。通常，每个操作符类只包含单一数据类型的操作符，而跨数据类型操作符则作为操作符族中的松散成员存在。

一个操作符族中的所有操作符和函数都必须具有兼容的语义，而兼容性的要求由索引方法设定。因此，你也许会疑惑，为什么还要把该族的某些子集单独划成操作符类；事实上，对很多用途而言，类的划分并不重要，真正有意义的分组只有操作符族。之所以定义操作符类，是因为它们规定了支持特定索引所需的操作符族内容。如果某个索引使用了某个操作符类，那么在不删除该索引的情况下就不能删除该操作符类 – 但操作符族中的其他部分，也就是其他操作符类和松散操作符，则可以被删除。因此，一个操作符类应当只包含在特定数据类型上支持索引所合理需要的最小操作符和函数集合，而那些相关但非必需的操作符，则可以作为操作符族的松散成员加入。

例如，PostgreSQL 内置了一个 B-树操作符族 `integer_ops`，其中包含操作符类 `int8_ops`、`int4_ops` 和 `int2_ops`，它们分别用于 `bigint`（`int8`）、`integer`（`int4`）和 `smallint`（`int2`）列上的索引。该操作符族还包含跨数据类型比较操作符，允许比较这三种类型中的任意两种，因此可以用一种类型的比较值搜索另一种类型上的索引。下面这些定义可以复现该操作符族：

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
  -- standard int8 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ,
  FUNCTION 2 btint8sortsupport(internal) ,
  FUNCTION 3 in_range(int8, int8, int8, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ,
  FUNCTION 6 btint8skipsupport(internal) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
  -- standard int4 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint4cmp(int4, int4) ,
  FUNCTION 2 btint4sortsupport(internal) ,
  FUNCTION 3 in_range(int4, int4, int4, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ,
  FUNCTION 6 btint4skipsupport(internal) ;

CREATE OPERATOR CLASS int2_ops
```

```
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
  -- standard int2 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint2cmp(int2, int2) ,
  FUNCTION 2 btint2sortsupport(internal) ,
  FUNCTION 3 in_range(int2, int2, int2, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ,
  FUNCTION 6 btint2skipsupport(internal) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
  -- cross-type comparisons int8 vs int2
  OPERATOR 1 < (int8, int2) ,
  OPERATOR 2 <= (int8, int2) ,
  OPERATOR 3 = (int8, int2) ,
  OPERATOR 4 >= (int8, int2) ,
  OPERATOR 5 > (int8, int2) ,
  FUNCTION 1 btint82cmp(int8, int2) ,

  -- cross-type comparisons int8 vs int4
  OPERATOR 1 < (int8, int4) ,
  OPERATOR 2 <= (int8, int4) ,
  OPERATOR 3 = (int8, int4) ,
  OPERATOR 4 >= (int8, int4) ,
  OPERATOR 5 > (int8, int4) ,
  FUNCTION 1 btint84cmp(int8, int4) ,

  -- cross-type comparisons int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- cross-type comparisons int4 vs int8
  OPERATOR 1 < (int4, int8) ,
  OPERATOR 2 <= (int4, int8) ,
  OPERATOR 3 = (int4, int8) ,
  OPERATOR 4 >= (int4, int8) ,
  OPERATOR 5 > (int4, int8) ,
  FUNCTION 1 btint48cmp(int4, int8) ,

  -- cross-type comparisons int2 vs int8
  OPERATOR 1 < (int2, int8) ,
  OPERATOR 2 <= (int2, int8) ,
  OPERATOR 3 = (int2, int8) ,
  OPERATOR 4 >= (int2, int8) ,
  OPERATOR 5 > (int2, int8) ,
  FUNCTION 1 btint28cmp(int2, int8) ,
```

```
-- cross-type comparisons int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ,

-- cross-type in_range functions
FUNCTION 3 in_range(int4, int4, int8, boolean, boolean) ,
FUNCTION 3 in_range(int4, int4, int2, boolean, boolean) ,
FUNCTION 3 in_range(int2, int2, int8, boolean, boolean) ,
FUNCTION 3 in_range(int2, int2, int4, boolean, boolean) ;
```

注意，这个定义对操作符策略号和支持函数号进行了“重载”：每个编号在该族内都会出现多次。只要某个编号的每次出现都具有不同的输入数据类型，这就是允许的。那些两个输入类型都等于某个操作符类输入类型的实例，就是该操作符类的主要操作符和支持函数；在大多数情况下，它们应当被声明为该操作符类的一部分，而不是该族的松散成员。

如第 63.2 节所详述，在一个 B-树操作符族中，该族中的所有操作符都必须以兼容的方式排序。对族中的每一个操作符，都必须有一个具有相同两个输入数据类型的支持函数。建议让操作符族保持完整，也就是说，对每一种数据类型组合都应包含全部操作符。每个操作符类只应包含其数据类型对应的非跨数据类型操作符和支持函数。

要构建一个多数据类型的 hash 操作符族，必须为该族支持的每一种数据类型创建相互兼容的 hash 支持函数。这里的兼容性是指：对任意两个被该族中的等值操作符视为相等的值，这些函数都保证返回相同的 hash 码，即使这两个值属于不同类型也是如此。当这些类型具有不同的物理表示时，这通常难以实现，但在某些情况下可以做到。此外，将该操作符族中一种数据类型的值通过隐式或二进制强制转换转为该族中另一种数据类型时，不得改变所计算出的 hash 值。注意，每种数据类型只有一个支持函数，而不是每个等值操作符一个。建议让操作符族保持完整，也就是说，对每一种数据类型组合都提供一个等值操作符。每个操作符类只应包含其数据类型对应的非跨数据类型等值操作符和支持函数。

GiST、SP-GiST 和 GIN 索引没有任何显式的跨数据类型操作概念。它们所支持的操作符集合，就是给定操作符类的主要支持函数所能处理的那些操作符。

在 BRIN 中，要求取决于提供操作符类的框架。对于基于 `minmax` 的操作符类，所要求的行为与 B-树操作符族相同：族中的所有操作符都必须以兼容的方式排序，并且类型转换不能改变相关的排序顺序。

> **注意**
>
> 在 PostgreSQL 8.3 之前，并没有操作符族这一概念，因此任何打算与索引一起使用的跨数据类型操作符都必须直接绑定到该索引的操作符类中。虽然这种做法仍然有效，但已被弃用，因为它会使索引的依赖关系过于宽泛，而且当两种数据类型都在同一操作符族中拥有操作符时，规划器能更有效地处理跨数据类型比较。

38.15.6. 系统对操作符类的依赖

PostgreSQL 利用操作符类来从多方面推断操作符的属性，而不仅仅是判断它们能否用于索引。因此，即便你并不打算为自己的数据类型列建立索引，也可能会想创建操作符类。

特别地，ORDER BY 和 DISTINCT 等 SQL 特性要求对值的比较和排序。为了在用户定义的数据类型上实现这些特性，PostgreSQL 会为数据类型查找默认 B-树操作符类。这个操作符类的“相等”成员定义了用于 GROUP BY 和 DISTINCT 的值的等值概念，而该操作符类施加的排序顺序定义了默认的ORDER BY 顺序。

如果一种数据类型没有默认的 B-树操作符类，系统就会查找默认的 hash 操作符类。但由于这类操作符类只提供等值语义，因此它只能支持分组，不能支持排序。

如果某种数据类型没有默认操作符类，而你又试图将这些 SQL 特性用于该数据类型，就会得到类似“无法识别排序操作符”这样的错误。

> **注意**
>
> 在版本 7.4 以前的 PostgreSQL 中，排序和分组操作将隐式地使用名为=、<以及> 的操作符。新的依赖于默认操作符类的行为避免了对具有特定名字的操作符行为作出任何假设。

可以通过在 USING 选项中指定某个非默认 B-树操作符类的小于操作符，按该操作符类进行排序，例如

```
SELECT * FROM mytable ORDER BY somecol USING ~<~;
```

或者，在 USING 中指定该操作符类的大于操作符，就会选择降序排序。

用户定义类型的数组比较也依赖于该类型默认 B-树操作符类所定义的语义。如果没有默认 B-树操作符类，但有默认的 hash 操作符类，则支持数组相等比较，但不支持排序比较。

另一种需要更多数据类型相关知识的 SQL 特性，是窗口函数中的 RANGE *offset* PRECEDING /FOLLOWING 帧选项（见第 4.2.8 节）。对于下面这样的查询

```
SELECT sum(x) OVER (ORDER BY x RANGE BETWEEN 5 PRECEDING AND 10 FOLLOWING)
  FROM mytable;
```

仅仅知道如何按 x 排序还不够；数据库还必须理解如何对当前行的 x 值“减 5”或“加 10”，以标识当前窗口帧的边界。把得到的边界与其他行的 x 值进行比较是可行的，因为可以使用定义 ORDER BY 顺序的 B-树操作符类所提供的比较操作符 – 但加法和减法操作符并不是该操作符类的一部分，那么该用哪些操作符呢？把这种选择硬编码下来并不合适，因为不同的排序顺序（不同的 B-树操作符类）可能需要不同的行为。因此，B-树操作符类可以指定一个in_range 支持函数，用来封装对其排序顺序有意义的加减行为。它甚至可以提供多个 in_range 支持函数，以防有多种数据类型都适合作为 RANGE 子句中的偏移量。如果与窗口 ORDER BY 子句关联的 B-树操作符类没有匹配的 in_range 支持函数，则不支持 RANGE *offset* PRECEDING/FOLLOWING 选项。

另一个重要点在于，出现在 hash 操作符族中的等值操作符，都是哈希连接、hash 聚合以及相关优化的候选对象。这里 hash 操作符族至关重要，因为它标识了应当使用的 hash 函数。

38.15.7. 排序操作符

有些索引访问方法（目前只有 GiST）支持排序操作符的概念。我们到目前为止讨论的是搜索操作符。对于搜索操作符，可以搜索索引以找出满足以下条件的所有行：`WHERE` *`indexed_column operator constant`*。注意，不保证匹配行的返回顺序。排序操作符则不限制可以返回的行集合，而是确定这些行的顺序。对于排序操作符，可以扫描索引，按以下表达式表示的顺序返回行：`ORDER BY` *`indexed_column operator constant`*。这样定义排序操作符，是因为当操作符用于度量距离时，它可以支持最近邻搜索。例如，如下查询：

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

可以找到距离指定目标点最近的十个地点。location 列上的 GiST 索引能够高效完成此操作，因为<-> 是排序操作符。

搜索操作符必须返回布尔结果，而排序操作符通常返回其他类型的结果，例如用于表示距离的 float 或 numeric。这种类型通常不同于被索引的数据类型。为了避免对不同数据类型行为作硬编码假设，在定义排序操作符时，必须指定一个 B-树操作符族，用来说明结果数据类型的排序顺序。正如上一节所述，B-树操作符族定义了 PostgreSQL 的顺序概念，因此这是一种自然的表示方式。由于点的 <-> 操作符返回 `float8`，因此可以在创建操作符类时这样指定它：

```
OPERATOR 15     <-> (point, point) FOR ORDER BY float_ops
```

其中 `float_ops` 是包含针对 `float8` 的操作的内置操作符族。这个声明表明，该索引能够按 <-> 操作符值递增的顺序返回行。

38.15.8. 操作符类的特殊特性

还有两个操作符类的特殊特性我们尚未讨论，主要是因为它们对最常用的索引方法没有用处。

通常，把一个操作符声明为某个操作符类（或操作符族）的成员，意味着该索引方法能够利用该操作符准确检索出满足 `WHERE` 条件的那组行。例如：

```
SELECT * FROM table WHERE integer_column < 4;
```

这个查询可以由整数列上的 B-树索引精确满足。但也有一些情况下，索引只能作为匹配行的不精确指引。例如，如果某个 GiST 索引只存储几何对象的边界框，那么它就无法精确满足测试非矩形对象（如多边形）之间重叠的 `WHERE` 条件。不过，我们仍然可以利用该索引找出边界框与目标对象边界框重叠的对象，然后只对索引找到的那些对象执行精确的重叠测试。如果适用于这种场景，就称该索引对这个操作符是“有损的”。有损索引搜索的实现方式是：当某一行可能满足、也可能不满足查询条件时，由索引方法返回一个recheck 标志。随后，核心系统会在取回的行上重新测试原始查询条件，以判断它是否应当作为合法匹配返回。只要索引能保证返回所有必需的行，外加可能存在的一些额外行，这种方法就是有效的，因为这些额外行可以通过执行原始操作符调用来剔除。支持有损搜索的索引方法（目前有 GiST、SP-GiST 和 GIN）允许个别操作符类的支持函数设置 recheck 标志，因此这本质上也是一种操作符类特性。

再次考虑只在索引中存储多边形等复杂对象的包围盒的情况。此时，在索引条目中存储整个多边形没有多少价值，不如只存储一个更简单的对象，其类型为box。这种情况由 STORAGE 选项表达，该选项位于CREATE OPERATOR CLASS：可以写成如下形式：

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

目前，只有 GiST、GIN 和 BRIN 索引方法支持与列数据类型不同的STORAGE 类型。GiST 的 compress 和 decompress 支持函数在使用 STORAGE 时必须处理数据类型转换。在 GIN 中，STORAGE 类型标识“键”值的类型，通常与被索引列的类型不同。例如，整数数组列的操作符类可以只使用整数作为键。GIN 的extractValue 和 extractQuery 支持函数负责从被索引值中提取键。BRIN 与 GIN 类似：STORAGE 类型标识所存储摘要值的类型，而操作符类的支持函数负责正确解释这些摘要值。

38.16. 将相关对象打包成扩展

一个有用的 PostgreSQL 扩展通常包含多个 SQL 对象；例如，一种新的数据类型将需要新的函数、新的操作符，并且很可能还需要新的索引操作符类。把所有这些对象收集到一个单独的包中，有助于简化数据库管理。PostgreSQL 将这样的包称为扩展。要定义一个扩展，至少需要一个脚本文件，其中包含用于创建扩展对象的 SQL 命令，以及一个控制文件，用来指定扩展本身的一些基本属性。如果扩展包含 C 代码，通常还会有一个共享库文件，C 代码已被构建到其中。一旦准备好这些文件，只需执行一个简单的CREATE EXTENSION 命令，就能把这些对象装入数据库。

使用扩展而不是仅仅运行 SQL 脚本把一堆“松散”对象装入数据库，最主要的优点在于 PostgreSQL 能够理解这些对象是属于同一个扩展的。你可以用一条 DROP EXTENSION 命令删除全部对象（无需维护单独的“卸载”脚本）。更有用的是，pg_dump 知道自己不应转储扩展的各个成员对象 – 它只会在转储中包含一条 CREATE EXTENSION 命令。这极大地简化了迁移到扩展新版本的过程，即使新版本包含比旧版本更多或不同的对象也是如此。不过要注意，在把这样的转储装载到新数据库时，必须能够访问该扩展的控制文件、脚本文件以及其他相关文件。

PostgreSQL 不允许你删除扩展中包含的单个对象，除非删除整个扩展。另外，虽然你可以修改扩展成员对象的定义（例如对函数使用 CREATE OR REPLACE FUNCTION），但要记住，被修改后的定义不会被 pg_dump 转储。这样的修改通常只有在你同时在扩展脚本文件中做出同样修改时才有意义。（不过，对于包含配置数据的表有特殊规定；见第 38.16.3 节。）在生产环境中，通常更好的做法是创建一个扩展更新脚本，用它来完成对扩展成员对象的修改。

扩展脚本可以使用 GRANT 和 REVOKE 语句，为属于扩展的对象设置权限。每个对象的最终权限集合（如果设置了权限）会存储在 pg_init_privs 系统目录中。使用 pg_dump 时，转储中会包含 CREATE EXTENSION 命令，后面再跟上一组必要的 GRANT 和 REVOKE 语句，以把对象权限恢复到生成该转储时的状态。

PostgreSQL 目前不支持扩展脚本发出 CREATE POLICY 或 SECURITY LABEL 语句。它们应当在扩展创建完成之后再设置。扩展对象上的所有 RLS 策略和安全标签都会包含在 pg_dump 创建的转储中。

扩展机制还提供了用于打包修改脚本的支持，以便调整扩展中所含 SQL 对象的定义。例如，如果扩展 1.1 版相比 1.0 版增加了一个函数，并修改了另一个函数的函数体，那么扩展作者可以提供一个更新脚本来只完成这两项修改。随后就可以使用 ALTER EXTENSION UPDATE 命令应用这些修改，并跟踪在某个数据库中实际安装的是该扩展的哪个版本。

哪些 SQL 对象种类可以成为扩展成员，见ALTER EXTENSION 的说明。特别是，数据库集簇范围内的对象，如数据库、角色和表空间，不能成为扩展成员，因为扩展只在单个数据库内可见。（尽管并不禁止扩展脚本创建这类对象，但如果这样做，它们不会作为扩展的一部分受到跟踪。）还要注意，虽然表可以成为扩展成员，但其附属对象（如索引）并不直接被视为扩展成员。另一个重要点是，模式可以属于扩展，但反过来不成立：扩展本身只有一个非限定名，并不“位于”任何模式中。不过，扩展的成员对象会在其对象类型适用的情况下属于某个模式。扩展是否应当拥有其成员对象所在的模式，则要视具体情况而定。

如果扩展脚本创建了任何临时对象（如临时表），这些对象在当前会话的剩余时间里会被视为扩展成员，但会像其他临时对象一样在会话结束时自动删除。这是“扩展成员对象不能在不删除整个扩展的情况下被删除”这一规则的一个例外。

38.16.1. 扩展文件

CREATE EXTENSION 命令依赖于每个扩展都有一个控制文件，其名称必须是扩展名加上 `.control` 后缀，并放在安装目录的 `SHAREDIR/extension` 目录中。还必须至少有一个 SQL 脚本文件，其名称遵循 *`extension--version`*`.sql` 的模式（例如，扩展 `foo` 的 `1.0` 版本使用 `foo--1.0.sql`）。默认情况下，脚本文件也放在 `SHAREDIR/extension` 目录中，但控制文件可以为脚本文件指定其他目录。

扩展控制文件的格式与 `postgresql.conf` 文件相同，即由一组 *`parameter_name`* = *`value`* 赋值组成，每行一条。允许空行和以 `#` 引入的注释。任何不是单个单词或数字的值都要记得加引号。

控制文件可以设置下列参数：

`directory` (`string`)

包含扩展 SQL 脚本文件的目录。除非给出的是绝对路径，否则该名称相对于安装的 `SHAREDIR` 目录。默认行为等价于指定 `directory = 'extension'`。

`default_version` (`string`)

扩展的默认版本（即在 `CREATE EXTENSION` 中未指定版本时将安装的版本）。虽然这个参数可以省略，但那样一来，如果没有给出 `VERSION` 选项，`CREATE EXTENSION` 就会失败，因此一般不希望这样做。

`comment` (`string`)

关于扩展的注释（任意字符串）。该注释会在初次创建扩展时应用，但不会在扩展更新时应用（因为那样可能会覆盖用户后来添加的注释）。另外，也可以在脚本文件中写一个 COMMENT 命令来设置扩展注释。

`encoding` (`string`)

脚本文件所使用的字符集编码。如果脚本文件包含任何非 ASCII 字符，就应指定这个参数。否则将假定这些文件使用数据库编码。

`module_pathname` (`string`)

该参数的值会替换脚本文件中每次出现的 `MODULE_PATHNAME`。如果未设置该参数，则不会进行替换。通常会把它设置为 `$libdir/`*`shared_library_name`*，然后在 C 语言函数的 `CREATE FUNCTION` 命令中使用 `MODULE_PATHNAME`，这样脚本文件就无需把共享库的名字硬编码进去。

`requires` (`string`)

本扩展所依赖的其他扩展名称列表，例如 `requires = 'foo, bar'`。这些被依赖的扩展必须先安装好，本扩展才能安装。

`superuser` (`boolean`)

如果此参数为 `true`（默认值），则只有超级用户可以创建扩展或将其更新到新版本。如果设为 `false`，则只需具有执行安装或更新脚本中命令所需的权限。

`relocatable` (`boolean`)

如果一个扩展在初次创建之后仍然可以把其包含的对象移动到不同的模式中，那么它就是可重定位的。默认值为 `false`，也就是该扩展不可重定位。详见第 38.16.2 节。

`schema` (`string`)

该参数只能为不可重定位扩展设置。它强制扩展装载到指定名称的模式中，而不能装载到其他模式。`schema` 参数只在初次创建扩展时起作用，扩展更新时不会使用。详见第 38.16.2 节。

除主控制文件 *`extension`*`.control` 外，扩展还可以有按如下样式命名的次级控制文件：*`extension--version`*`.control`。如果提供了这些文件，它们必须位于脚本文件目录中。次级控制文件遵循与主控制文件相同的格式。在安装或更新到该扩展的相应版本时，次级控制文件中设置的任何参数都会覆盖主控制文件中的设置。不过，`directory` 和 `default_version` 这两个参数不能在次级控制文件中设置。

扩展的 SQL 脚本文件可以包含任何 SQL 命令，但事务控制命令（`BEGIN`、`COMMIT` 等）和无法在事务块内执行的命令（例如 `VACUUM`）除外。这是因为脚本文件会被隐式地放在事务块中执行。

扩展的 SQL 脚本文件也可以包含以 `\echo` 开头的行，扩展机制会忽略这些行（将其视为注释）。这一约定通常用于在脚本文件被直接交给 psql 而不是通过 `CREATE EXTENSION` 装载时抛出错误（见第 38.16.7 节中的示例脚本）。如果没有这种机制，用户可能会意外地把扩展内容作为“松散”对象装载，而不是作为一个扩展来装载，这种状态恢复起来会有些麻烦。

虽然脚本文件可以包含指定编码允许的任意字符，但控制文件应只包含纯 ASCII 字符，因为 PostgreSQL 无法知道控制文件采用的是什么编码。在实践中，只有当你想在扩展注释中使用非 ASCII 字符时这才会成为问题。对此推荐的做法是不要使用控制文件中的 `comment` 参数，而是在脚本文件中使用 `COMMENT ON EXTENSION` 来设置注释。

38.16.2. 扩展的可重定位性

用户经常希望把扩展中的对象装载到与扩展作者原先设想不同的模式中。对于这种可重定位性，支持三个级别：

- 完全可重定位的扩展可以在任何时候移动到另一个模式中，即使它已经被装载到数据库之后也是如此。这通过 `ALTER EXTENSION SET SCHEMA` 命令完成，该命令会自动把所有成员对象重命名到新模式中。通常，只有当扩展对其任何对象所在模式都没有内部假设时，才有可能做到这一点。此外，扩展的对象一开始必须全部位于同一个模式中（不属于任何模式的对象，如过程语言，不算在内）。要把一个扩展标记为完全可重定位，只需在其控制文件中设置 `relocatable = true`。
- 扩展可能在安装期间可重定位，但安装之后不可重定位。如果扩展脚本文件需要显式引用目标模式，例如为 SQL 函数设置 `search_path` 属性时，通常就是这种情况。对于这样的扩展，应

在控制文件中设置 relocatable = false，并在脚本文件中使用 @extschema@ 来引用目标模式。在脚本执行前，该字符串的每次出现都会被替换为实际目标模式的名称。用户可以通过 CREATE EXTENSION 的 SCHEMA 选项设置目标模式。

- 如果扩展完全不支持重定位，应在控制文件中设置 relocatable = false，并把 schema 设置为预定目标模式的名称。这样将阻止使用 CREATE EXTENSION 的 SCHEMA 选项，除非它指定的正是控制文件中命名的那个模式。如果扩展对模式名称有无法通过 @extschema@ 替换解决的内部假设，通常就需要采用这种方式。在这种情况下，@extschema@ 替换机制仍然可用，只是由于模式名由控制文件决定，其用途比较有限。

在所有情况下，脚本文件执行时，其初始search_path 都会指向目标模式；也就是说，CREATE EXTENSION 做的事情等价于：

```
SET LOCAL search_path TO @extschema@, pg_temp;
```

这使得脚本文件创建的对象能够进入目标模式。脚本文件当然也可以修改 search_path，但通常不建议这样做。CREATE EXTENSION 完成后，search_path 会恢复为先前的设置。

目标模式由控制文件中的 schema 参数决定（如果给出了该参数）；否则由 CREATE EXTENSION 的 SCHEMA 选项决定（如果给出了该选项）；否则由当前默认的对象创建模式决定（也就是调用者的 search_path 中第一个模式）。当使用控制文件中的 schema 参数时，如果目标模式尚不存在，则会自动创建；而在另外两种情况下，目标模式必须已经存在。

如果控制文件的 requires 中列出了任何前置扩展，它们的目标模式会追加到 search_path 的初始设置中，排在新扩展的目标模式之后。这样新扩展的脚本文件就可以看到这些扩展的对象。

出于安全考虑，pg_temp 在所有情况下都会自动追加到 search_path 的末尾。

虽然不可重定位扩展可以包含分布在多个模式中的对象，但通常仍然希望把所有供外部使用的对象放在单个模式中，这个模式会被视为该扩展的目标模式。这种安排与创建依赖扩展时 search_path 的默认设置配合起来会比较方便。

38.16.3. 扩展配置表

有些扩展包含配置表，其中存放的数据可能会在扩展安装后被用户新增或修改。通常，如果表属于某个扩展，那么 pg_dump 既不会转储该表的定义，也不会转储其内容。但这种行为对于配置表来说并不理想；用户所做的数据更改必须包含在转储中，否则在转储并恢复之后，扩展的行为就会发生变化。

为了解决这个问题，扩展脚本文件可以把它创建的某个表或序列标记为配置关系，这样 pg_dump 就会把该表或序列的内容（而不是定义）包含到转储中。做法是在创建表或序列之后调用函数 pg_extension_config_dump(regclass, text)，例如：

```
CREATE TABLE my_config (key text, value text);
CREATE SEQUENCE my_config_seq;

SELECT pg_catalog.pg_extension_config_dump('my_config', '');
SELECT pg_catalog.pg_extension_config_dump('my_config_seq', '');
```

可以用这种方式标记任意数量的表或序列。与 serial 或 bigserial 列关联的序列也可以这样标记。

当 pg_extension_config_dump 的第二个参数是空字符串时，pg_dump 会转储该表的全部内容。通常只有当该表在扩展脚本创建时最初为空时，这样做才是正确的。如果表中混有初始数据和用户提供的数据，那么 pg_extension_config_dump 的第二个参数就提供了一个用于选择要转储数据的 WHERE 条件。例如，你可以这样做：

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT standard_entry');
```

然后确保只有扩展脚本创建的那些行，其 standard_entry 才为真。

对于序列，pg_extension_config_dump 的第二个参数没有作用。

更复杂的情况，例如用户可能会修改最初提供的数据行，可以通过在配置表上创建触发器来处理，以确保被修改的行被正确标记。

你可以再次调用 pg_extension_config_dump 来修改与配置表关联的过滤条件。（这通常在扩展更新脚本中很有用。）要把某个表标记为不再是配置表，唯一的方法是通过 ALTER EXTENSION ... DROP TABLE 将它与扩展解除关联。

注意，这些表之间的外键关系会决定 pg_dump 转储它们的顺序。具体来说，pg_dump 会尝试先转储被引用表，再转储引用表。由于外键关系是在 CREATE EXTENSION 时建立的（那时数据尚未装入这些表中），因此不支持环状依赖。若存在环状依赖，数据仍会被转储出来，但该转储将无法直接恢复，需要用户介入处理。

与 serial 或 bigserial 列关联的序列需要被直接标记，才能转储它们的状态。仅仅标记它们的父关系还不足以达到这一目的。

38.16.4. 扩展更新

扩展机制的一个优点是，它为管理定义扩展对象的 SQL 命令的更新提供了便利方式。这是通过为每个已发布版本的扩展安装脚本关联一个版本名或版本号来实现的。此外，如果你希望用户能够把数据库从一个版本动态更新到下一个版本，就应提供更新脚本，以完成从一个版本切换到下一版本所需的修改。更新脚本的名称遵循如下模式：*extension--old_version--target_version*.sql（例如，foo--1.0--1.1.sql 包含把扩展 foo 的 1.0 版修改为 1.1 版所需的命令）。

在有合适更新脚本可用的前提下，ALTER EXTENSION UPDATE 命令会把已安装的扩展更新到指定的新版本。更新脚本运行在 CREATE EXTENSION 为安装脚本提供的同一环境中：尤其是，search_path 的设置方式完全相同，而且脚本创建的任何新对象都会自动加入扩展中。此外，如果脚本选择删除扩展成员对象，它们也会自动与扩展解除关联。

如果扩展有次级控制文件，那么用于更新脚本的控制参数就是与该脚本目标（新）版本相关联的那些参数。

更新机制可以用于解决一个重要的特殊情况：将“松散”对象集合转换为扩展。在 PostgreSQL 于 9.1 加入扩展机制之前，许多人编写的扩展模块只是创建各种未打包的对象。对于包含这些对象的现有数据库，如何将它们转换为正确打包的扩展？删除它们再执行普通的 CREATE EXTENSION 是一种办法，但如果对象具有依赖关系（例如，某些表列使用扩展创建的数据类型），就不适合这样做。解决方法是先创建一个空扩展，再使用 ALTER EXTENSION ADD 将每个现有对象附加到扩展中，最后创建当前扩展版本中存在、而未打包版本中没有的新对象。CREATE EXTENSION 通过 FROM *old_version* 选项支持这种情况：它不运行目标版本的普通安装脚本，而是运行名为 *extension--old_version--target_version*.sql 的更新脚本。用作 *old_version* 的虚拟版本名由扩展作者选择，不过

通常约定使用 unpackaged。如果需要将多个先前版本更新为扩展形式，应使用不同的虚拟版本名来标识它们。

ALTER EXTENSION 能够执行一系列更新脚本文件来完成请求的更新。例如，如果只有 foo--1.0--1.1.sql 和 foo--1.1--2.0.sql 可用，那么当当前安装的是 1.0，而请求更新到 2.0 时，ALTER EXTENSION 就会按顺序应用这两个脚本。

PostgreSQL 并不对版本名称的属性作任何假设：例如，它并不知道 1.1 是否跟在 1.0 之后。它只是匹配可用的版本名，并选择需要应用更新脚本最少的那条路径。（实际上，版本名可以是任何不包含 --，且不以 - 开头或结尾的字符串。）

有时提供“降级”脚本也是有用的，例如 foo--1.1--1.0.sql 可用于回退与 1.1 版相关的修改。如果你这样做，要注意某个降级脚本可能会因为产生更短的路径而意外被采用。危险情况是，存在一个跨越多个版本的“快速路径”更新脚本，同时又有一个可降级到该快速路径起点的脚本。这时，先降级再走快速路径，可能比按版本逐步前进所需的步数更少。如果降级脚本删除了任何不可替代的对象，就会产生不希望看到的结果。

要检查是否存在意外的更新路径，可使用以下命令：

```
SELECT * FROM pg_extension_update_paths('extension_name');
```

它会显示指定扩展的每一对不同已知版本名，以及从源版本到目标版本将采用的更新路径序列；如果没有可用的更新路径，则显示 NULL。路径会以文本形式显示，并使用 -- 作为分隔符。如果你更喜欢数组形式，可以使用 regexp_split_to_array(path,'--')。

38.16.5. 使用更新脚本安装扩展

一个存在已久的扩展很可能会有多个版本，因此作者需要为其编写更新脚本。例如，如果你发布过 foo 扩展的 1.0、1.1 和 1.2 版本，那么就应当有更新脚本 foo--1.0--1.1.sql 和 foo--1.1--1.2.sql。在 PostgreSQL 10 之前，还必须额外创建新的脚本文件 foo--1.1.sql 和 foo--1.2.sql，用来直接构建更新后的扩展版本；否则新版本无法被直接安装，只能先安装 1.0 再升级。那样既繁琐又重复，但现在已经没有必要了，因为 CREATE EXTENSION 可以自动沿着更新链前进。例如，如果只有 foo--1.0.sql、foo--1.0--1.1.sql 和 foo--1.1--1.2.sql 这几个脚本文件可用，那么请求安装 1.2 版时，就会按顺序运行这三个脚本。处理方式与先安装 1.0 再更新到 1.2 完全相同。（和 ALTER EXTENSION UPDATE 一样，如果有多条路径可用，则优先选择最短的一条。）按这种风格组织扩展的脚本文件，可以减少制作小型更新所需的维护工作量。

如果你对以这种风格维护的扩展使用次级（特定于版本的）控制文件，请记住：即使某个版本没有独立的安装脚本，它也仍然需要一个控制文件，因为该控制文件将决定如何执行到该版本的隐式更新。例如，如果 foo--1.0.control 指定了 requires = 'bar'，而 foo 的其他控制文件没有指定，那么当从 1.0 更新到其他版本时，该扩展对 bar 的依赖就会被去掉。

38.16.6. 扩展的安全注意事项

广泛分发的扩展应尽量少假定其所处数据库的环境。因此，以一种不会被基于搜索路径的攻击破坏的安全风格来编写扩展所提供的函数，是合适的做法。

superuser 属性设为真的扩展，还必须考虑其安装和更新脚本所执行操作的安全风险。恶意用户可以创建木马对象，破坏以后对编写不慎的扩展脚本的执行，从而获得超级用户权限；这并非特别困难。

关于如何安全地编写函数，建议见下面的第 38.16.6.1 节；关于如何安全地编写安装脚本，建议见第 38.16.6.2 节。

38.16.6.1. 扩展函数的安全注意事项

扩展提供的 SQL 语言函数和 PL 语言函数在执行时会面临基于搜索路径的攻击风险，因为这些函数是在执行时而不是创建时解析的。

`CREATE FUNCTION` 参考页中包含了关于如何安全编写 `SECURITY DEFINER` 函数的建议。对于扩展提供的任意函数，都最好应用这些技术，因为该函数可能会被高权限用户调用。

如果你无法把 `search_path` 设置为只包含安全模式，那就应假定每个非限定名都可能被解析为恶意用户定义的对象。要警惕那些会隐式依赖 `search_path` 的构造；例如，`IN` 和 `CASE *expression* WHEN` 总是通过搜索路径来选择操作符。应改用 `OPERATOR(*schema*.=) ANY` 和 `CASE WHEN *expression*`。

通用扩展通常不应假定它被安装到了安全模式中，这意味着即使对其自身对象使用了模式限定引用，也并非完全没有风险。例如，如果扩展定义了函数 `myschema.myfunc(bigint)`，那么像 `myschema.myfunc(42)` 这样的调用，就可能被恶意函数 `myschema.myfunc(integer)` 截获。要注意函数和操作符所传参数的数据类型必须与声明的参数类型精确匹配，必要时请使用显式类型转换。

38.16.6.2. 扩展脚本的安全注意事项

编写扩展安装脚本或更新脚本时，应防范脚本执行过程中发生基于搜索路径的攻击。如果脚本中的对象引用可能被解析为脚本作者原本无意引用的其他对象，那么破坏既可能立刻发生，也可能在稍后使用这个定义错误的扩展对象时才发生。

像 `CREATE FUNCTION` 和 `CREATE OPERATOR CLASS` 这样的 DDL 命令通常是安全的，但要警惕任何把通用表达式作为组成部分的命令。例如，`CREATE VIEW` 就需要仔细审查，`CREATE FUNCTION` 中的 `DEFAULT` 表达式也是如此。

有时扩展脚本可能需要执行通用 SQL，例如为完成 DDL 无法做到的系统目录调整。执行这类命令时，要确保使用安全的 `search_path`；不要相信 `CREATE/ALTER EXTENSION` 提供的路径一定是安全的。最佳做法是临时把 `search_path` 设为 `'pg_catalog, pg_temp'`，并在需要时显式插入对扩展安装模式的引用。（这种做法对创建视图也可能有帮助。）在 PostgreSQL 源代码发行包的 `contrib` 模块中可以找到示例。

跨扩展引用极难做到完全安全，部分原因是无法确定另一个扩展位于哪个模式中。如果两个扩展安装在同一模式，风险会降低，因为此时恶意对象无法在安装时的 `search_path` 中排在被引用扩展之前。不过，目前没有机制强制这一点。

38.16.7. 扩展示例

下面给出一个纯 SQL 扩展的完整示例：一个双元素复合类型，它可以在两个槽位中存储任意类型的值，这两个槽位名为“k”和“v”。非文本值会自动强制转换为文本后再存储。

脚本文件 `pair--1.0.sql` 如下所示：

```
-- complain if script is sourced in psql, rather than via CREATE EXTENSION
\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );

CREATE FUNCTION pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::@extschema@.pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, FUNCTION = pair);

-- "SET search_path" is easy to get right, but qualified names perform better.
CREATE FUNCTION lower(pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW(lower($1.k), lower($1.v))::@extschema@.pair;'
SET search_path = pg_temp;

CREATE FUNCTION pair_concat(pair, pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW($1.k OPERATOR(pg_catalog.||) $2.k,
              $1.v OPERATOR(pg_catalog.||) $2.v)::@extschema@.pair;';
```

控制文件 `pair.control` 如下所示：

```
# pair extension
comment = 'A key/value pair data type'
default_version = '1.0'
# cannot be relocatable because of use of @extschema@
relocatable = false
```

虽然你几乎不需要专门写一个 makefile 来把这两个文件安装到正确的目录中，但你也可以使用如下内容的 `Makefile`：

```
EXTENSION = pair
DATA = pair--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

这个 makefile 依赖于 PGXS，其说明见第 38.17 节。执行 `make install` 命令会把控制文件和脚本文件安装到 pg_config 报告的正确目录中。

安装文件后，使用CREATE EXTENSION 命令将这些对象装入某个具体数据库。

38.17. 扩展构建基础设施

如果你打算分发自己的 PostgreSQL 扩展模块，那么为它们搭建一个可移植的构建系统可能相当困难。因此，PostgreSQL 安装提供了一套称为 PGXS 的扩展构建基础设施，使简单的扩展模块可以针对已安装好的服务器直接构建。PGXS 主要面向包含 C 代码的扩展，不过它也能用于纯 SQL 扩展。注意，PGXS 并不打算成为一个可以用来构建任意与 PostgreSQL 交互软件的通用构建系统框架；它只是把简单服务器扩展模块的常见构建规则自动化。对于更复杂的软件包，你可能还是需要自己编写构建系统。

要为你的扩展使用 PGXS 基础设施，你必须写一个简单的 makefile。在这个 makefile 中，需要设置一些变量并包含全局 PGXS makefile。下面是一个示例，它构建一个名为 isbn_issn 的扩展模块，该模块包含一个装有一些 C 代码的共享库、一个扩展控制文件、一个 SQL 脚本、一个头文件（只有当其他模块可能需要绕过 SQL 直接访问扩展函数时才需要），以及一个文档文本文件：

```
MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn
HEADERS_isbn_issn = isbn_issn.h

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

最后三行始终都应相同。你应在文件前面的部分设置变量或添加自定义的 make 规则。

设定下列三个变量中的一个，以指定要构建的内容：

MODULES

要从同名源文件构建的共享库对象列表（列表中不要包含库后缀）

MODULE_big

要从多个源文件构建的共享库（在 OBJS 中列出目标文件）

PROGRAM

要构建的可执行程序（在 OBJS 中列出目标文件）

还可以设置以下变量：

EXTENSION

扩展名称；对于每个名称，你都必须提供一个 *extension*.control 文件，它将安装到 *prefix*/share/extension

MODULEDIR

prefix/share 下的子目录，用于安装 DATA 和 DOCS 文件（若未设置，则在设置了 EXTENSION 时默认为 extension，否则默认为 contrib）

DATA

要安装到 *prefix*/share/$MODULEDIR 的任意文件

DATA_built

要安装到 *prefix*/share/$MODULEDIR 的任意文件，但它们需要先被构建

DATA_TSEARCH

要安装到 *prefix*/share/tsearch_data 下的任意文件

DOCS

要安装到 *prefix*/doc/$MODULEDIR 下的任意文件

HEADERS
HEADERS_built

要（可选地先构建并）安装到 *prefix*/include/server/$MODULEDIR/$MODULE_big 下的文件。

与 DATA_built 不同，HEADERS_built 中的文件不会被 clean 目标删除；如果你希望删除它们，也应把它们加入 EXTRA_CLEAN，或者自行添加规则来完成删除。

HEADERS_$MODULE
HEADERS_built_$MODULE

要安装的文件（如果指定了则先构建），安装位置为 *prefix*/include/server/$MODULEDIR/$MODULE，其中 $MODULE 必须是 MODULES 或 MODULE_big 中使用的某个模块名。

与 DATA_built 不同，HEADERS_built_$MODULE 中的文件不会被 clean 目标删除；如果你希望删除它们，也应把它们加入 EXTRA_CLEAN，或者自行添加规则来完成删除。

对同一个模块同时使用这两个变量，或以任意方式组合使用，都是合法的；但如果你在 MODULES 列表中有两个模块名只相差一个前缀 built_，就会产生歧义。在这种情况（希望不太可能发生）下，你应只使用 HEADERS_built_$MODULE 变量。

SCRIPTS

要安装到 *prefix*/bin 的脚本文件（不是二进制文件）

SCRIPTS_built

要安装到 *prefix*/bin 的脚本文件（不是二进制文件），但它们需要先被构建

REGRESS

回归测试用例列表（不带后缀），详见下文

REGRESS_OPTS

传递给 pg_regress 的额外开关

NO_INSTALLCHECK

不定义 installcheck 目标，适用于测试需要特殊配置，或者不使用 pg_regress 的情况

EXTRA_CLEAN

在 make clean 中要额外删除的文件

PG_CPPFLAGS

将被添加到 CPPFLAGS 前面

PG_CFLAGS

将被添加到 CFLAGS 后面

PG_CXXFLAGS

将被添加到 CXXFLAGS 后面

PG_LDFLAGS

将被添加到 LDFLAGS 前面

PG_LIBS

将被加入 PROGRAM 的链接命令行

SHLIB_LINK

将被加入 MODULE_big 的链接命令行

PG_CONFIG

要针对其进行构建的 PostgreSQL 安装所对应的 pg_config 程序路径（通常只写 pg_config，表示使用你 PATH 中找到的第一个）

把这个 makefile 命名为 Makefile，并放在保存扩展的目录中。然后你就可以执行 make 进行编译，再执行 make install 安装你的模块。默认情况下，该扩展会针对你 PATH 中找到的第一个 pg_config 所对应的 PostgreSQL 安装进行编译和安装。你也可以使用不同的安装，只需让 PG_CONFIG 指向它的 pg_config 程序，无论是在 makefile 中设置，还是在 make 命令行上设置都可以。

如果你想保持构建目录与源代码目录分离，也可以在扩展源代码树之外的目录中运行 make。这一过程也称为 VPATH 构建。做法如下：

```
mkdir build_dir
cd build_dir
make -f /path/to/extension/source/tree/Makefile
make -f /path/to/extension/source/tree/Makefile install
```

另外，你也可以像核心代码那样为 VPATH 构建准备一个目录。其中一种方法是使用核心脚本 config/prep_buildtree。准备好之后，就可以像下面这样通过设置 make 变量 VPATH 来构建：

```
make VPATH=/path/to/extension/source/tree
make VPATH=/path/to/extension/source/tree install
```

这种方式适用于更多种目录布局。

在 REGRESS 变量中列出的脚本用于对模块做回归测试，可在执行完 make install 之后，通过 make installcheck 来运行。要让它工作，你必须有一个正在运行的 PostgreSQL 服务器。列在 REGRESS 中的脚本文件必须位于扩展目录下名为 sql/ 的子目录中。这些文件必须具有 .sql 扩展名，而该扩展名不能出现在 makefile 的 REGRESS 列表中。对于每个测试，还应在名为 expected/ 的子目录中有一个包含期望输出的文件，其主干名相同，扩展名为 .out。make installcheck 会用 psql 执行每个测试脚本，并把得到的输出与对应的期望文件比较。

任何差异都会以 `diff -c` 格式写入 `regression.diffs` 文件。注意，如果尝试运行一个缺少期望文件的测试，将被报告为“trouble”，所以请确保所有期望文件都已准备好。

> **提示**
>
> 创建期望文件最简单的方法是先建立空文件，然后运行一次测试（当然这会报告差异）。检查 `results/` 目录中的实际结果文件；如果它们与你对测试的预期一致，就把它们复制到 `expected/` 中。

第 39 章 触发器

本章提供有关编写触发器函数的一般信息。触发器函数可以用大多数可用的过程语言编写，包括 PL/pgSQL（第 43 章）、PL/Tcl（第 44 章）、PL/Perl（第 45 章）和 PL/Python（第 46 章）。读完本章后，你还应查阅所用过程语言对应的章节，了解用该语言编写触发器的语言相关细节。

也可以用 C 编写触发器函数，不过大多数人会觉得使用某种过程语言更容易。目前还不能用普通 SQL 函数语言编写触发器函数。

39.1. 触发器行为概述

触发器是一种规定：每当执行某种类型的操作时，数据库就自动执行某个特定函数。触发器可以附加到表（无论是否分区）、视图和外部表上。

在表和外部表上，触发器可以定义为在任何 `INSERT`、`UPDATE` 或 `DELETE` 操作之前或之后执行；既可以对每个被修改的行执行一次，也可以对每个 SQL 语句执行一次。`UPDATE` 触发器还可以设置成只有在 `UPDATE` 语句的 `SET` 子句中提到特定列时才触发。触发器也可以针对 `TRUNCATE` 语句触发。一旦发生触发器事件，就会在适当的时机调用该触发器的函数来处理该事件。

在视图上，触发器可以定义为代替 `INSERT`、`UPDATE` 或 `DELETE` 操作执行。这种 `INSTEAD OF` 触发器会对视图中每个需要修改的行触发一次。触发器函数负责对视图底层的基表执行必要的修改，并在适当时返回该修改后的行，使其在视图中呈现为应有的样子。视图上的触发器也可以定义为在 `INSERT`、`UPDATE` 或 `DELETE` 操作之前或之后，对每个 SQL 语句执行一次。不过，这种触发器只有在该视图上还存在一个 `INSTEAD OF` 触发器时才会触发。否则，任何以该视图为目标的语句都必须被重写成作用于其底层基表的语句，此时触发的将是附加在那些基表上的触发器。

必须先定义触发器函数，之后才能创建触发器本身。触发器函数必须声明为不接受参数并返回类型 `trigger` 的函数。（触发器函数通过特殊传入的 `TriggerData` 结构体接收输入，而不是通过普通函数参数。）

创建出合适的触发器函数后，就可以使用CREATE TRIGGER 创建触发器。同一个触发器函数可以供多个触发器使用。

PostgreSQL 同时提供每行触发器和每语句触发器。对于每行触发器，触发它的语句所影响的每一行都会调用一次触发器函数。相反，每语句触发器只会在执行相应语句时调用一次，不管该语句影响了多少行。特别是，即使某个语句一行也没有影响，仍会执行所有适用的每语句触发器。这两类触发器有时也分别称为行级触发器和语句级触发器。`TRUNCATE` 的触发器只能定义为语句级，不能定义为每行。

触发器还可以按照它们是在操作之前、之后，还是取代该操作而触发来分类。这些分别称为 `BEFORE` 触发器、`AFTER` 触发器和 `INSTEAD OF` 触发器。语句级 `BEFORE` 触发器自然会在语句开始执行任何操作之前触发，而语句级 `AFTER` 触发器则在语句末尾触发。这些类型的触发器可以定义在表、视图或外部表上。行级 `BEFORE` 触发器会在对某一行执行操作之前立即触发，而行级 `AFTER` 触发器会在语句结束时触发（但早于任何语句级 `AFTER` 触发器）。这些类型的触发器只能定义在表和外部表上，不能定义在视图上；`BEFORE` 行级触发器不能定义在分区表上。`INSTEAD OF` 触发器只能定义在视图上，而且只能是行级；当视图中的每一行被识别为需要执行操作时，它们就会立即触发。

如果某条语句以继承层次或分区层次中的父表为目标，它不会导致受影响子表上的语句级触发器触发；只有父表上的语句级触发器会触发。不过，任何受影响子表上的行级触发器仍会触发。

若 INSERT 包含 ON CONFLICT DO UPDATE 子句，则行级 BEFORE INSERT 触发器和行级 BEFORE UPDATE 触发器的效果可能都会体现在更新后行的最终状态中，前提是引用了 EXCLUDED 列。不过，即使没有引用 EXCLUDED 列，这两组行级 BEFORE 触发器也可能都会执行。如果同时存在会更改正在插入/更新的行的 BEFORE INSERT 和 BEFORE UPDATE 行级触发器，就应考虑可能出现令人意外的结果（即使它们的修改大致等效，如果不具有幂等性，也可能出现问题）。注意，语句级 UPDATE 触发器会在指定 ON CONFLICT DO UPDATE 时执行，不管 UPDATE 是否实际影响了行（也不管是否走到了 UPDATE 分支）。INSERT 如果带有 ON CONFLICT DO UPDATE 子句，会先执行语句级 BEFORE INSERT 触发器，再执行语句级 BEFORE UPDATE 触发器，然后执行语句级 AFTER UPDATE 触发器，最后执行语句级 AFTER INSERT 触发器。

如果对分区表执行 UPDATE 导致某行移动到另一个分区，这将表现为从原始分区执行一次 DELETE，然后向新分区执行一次 INSERT。在这种情况下，原始分区上的所有行级 BEFORE UPDATE 触发器和所有行级 BEFORE DELETE 触发器都会触发。随后，目标分区上的所有行级 BEFORE INSERT 触发器都会触发。当这些触发器都会影响被移动的那一行时，应考虑到可能出现令人意外的结果。对于 AFTER ROW 触发器，会执行 AFTER DELETE 和 AFTER INSERT 触发器；但不会执行 AFTER UPDATE 触发器，因为该 UPDATE 已被转换成一次 DELETE 和一次 INSERT。对于语句级触发器而言，即使发生了行移动，也不会触发任何 DELETE 或 INSERT 触发器；只会触发在 UPDATE 语句所用目标表上定义的 UPDATE 触发器。

由每语句触发器调用的触发器函数应始终返回 NULL。由每行触发器调用的触发器函数则可以根据需要向调用它的执行器返回一个表行（即类型 HeapTuple 的值）。在操作之前触发的行级触发器可以作出如下选择：

- 它可以返回 NULL，以跳过对当前行的操作。这会指示执行器不要执行引发该触发器的行级操作（即对某个具体表行的插入、修改或删除）。
- 仅对于行级 INSERT 和 UPDATE 触发器，返回的行会成为将要插入的行，或替换正在更新的行。这使得触发器函数可以修改即将插入或更新的行。

不打算产生上述任何一种行为的行级 BEFORE 触发器，必须小心地返回传入的同一行作为结果（也就是说，INSERT 和 UPDATE 触发器返回 NEW 行，DELETE 触发器返回 OLD 行）。

行级 INSTEAD OF 触发器应当要么返回 NULL，表示它没有修改视图底层基表中的任何数据；要么返回传入的视图行（INSERT 和 UPDATE 操作返回 NEW 行，DELETE 操作返回 OLD 行）。非空返回值用来表明触发器已经在视图中完成了必要的数据修改。这会使该命令所影响的行数计数增加。仅对于 INSERT 和 UPDATE 操作，触发器在返回 NEW 行之前还可以修改它。这会改变 INSERT RETURNING 或 UPDATE RETURNING 返回的数据；当视图显示的数据与提供给它的数据不完全相同时，这很有用。

对于在操作之后触发的行级触发器，其返回值会被忽略，因此它们可以返回 NULL。

如果为同一关系上的同一事件定义了多个触发器，它们会按触发器名称的字母顺序触发。对于 BEFORE 和 INSTEAD OF 触发器，每个触发器返回的、可能已经被修改过的行都会成为下一个触发器的输入。如果任何一个 BEFORE 或 INSTEAD OF 触发器返回 NULL，则会放弃对该行执行该操作，并且后续触发器（针对该行）都不会再触发。

触发器定义还可以指定一个布尔 WHEN 条件，用来测试是否应当触发该触发器。对于行级触发器，WHEN 条件可以检查该行各列的旧值和/或新值。（语句级触发器也可以有 WHEN 条件，不过这个特性对它们用处不大。）在 BEFORE 触发器中，WHEN 条件就在函数即将执行或本会执行之前求值，因此使用 WHEN 与在触发器函数开头测试同样的条件并无实质差别。不过，在 AFTER 触发器中，WHEN 条件会在行更改发生后立即求值，并决定是否要将一个事件排入队列，以便在语句末尾触发该触发器。因此，当 AFTER 触发器的 WHEN 条件不返回真时，就不必排队该事件，也

不必在语句末尾重新取出该行。如果触发器只需要针对少数几行触发，这会让修改大量行的语句显著加快。INSTEAD OF 触发器不支持 WHEN 条件。

通常，行级 BEFORE 触发器用于检查或修改将要插入或更新的数据。例如，BEFORE 触发器可以用来把当前时间写入某个 timestamp 列，或者检查该行的两个元素是否一致。行级 AFTER 触发器最适合用于把更新传播到其他表，或对其他表执行一致性检查。之所以这样分工，是因为 AFTER 触发器可以确定自己看到的是该行的最终值，而 BEFORE 触发器不能；它之后可能还会有其他 BEFORE 触发器触发。如果没有特定理由把触发器做成 BEFORE 或 AFTER，那么 BEFORE 形式更高效，因为关于该操作的信息无需一直保存到语句末尾。

如果触发器函数执行 SQL 命令，那么这些命令可能会再次触发触发器。这就是所谓的级联触发器。对级联层数没有直接限制。级联还有可能导致同一个触发器被递归调用；例如，一个 INSERT 触发器可能执行一条向同一个表再插入一行的命令，从而导致该 INSERT 触发器再次触发。因此，在这种场景下，避免无限递归是触发器编写者自己的责任。

在定义触发器时，可以为它指定参数。在触发器定义中包含参数，是为了让需求相似的不同触发器能够调用同一个函数。举例来说，可以有一个通用触发器函数，它接受两个列名作为参数，把当前用户写入其中一个列，把当前时间戳写入另一个列。只要编写得当，这个触发器函数就应独立于它所作用的具体表。因此，同一个函数可用于任何具有适当列的表上的 INSERT 事件，例如自动跟踪某个事务表中记录的创建。如果把它定义成 UPDATE 触发器，还可以用来跟踪最近更新事件。

每种支持触发器的编程语言都有自己的方法，使触发器输入数据可供触发器函数使用。这些输入数据包括触发器事件的类型（例如 INSERT 或 UPDATE），以及 CREATE TRIGGER 中列出的任何参数。对于行级触发器，输入数据还包括 INSERT 和 UPDATE 触发器的 NEW 行，以及/或者 UPDATE 和 DELETE 触发器的 OLD 行。

默认情况下，语句级触发器无法检查该语句修改的各个行。但是 AFTER STATEMENT 触发器可以请求创建过渡表，从而使受影响的行集合可供该触发器使用。AFTER ROW 触发器也可以请求过渡表，这样它们既能看到表中的整体变化，也能看到当前使其触发的那一行中的变化。如何检查过渡表，同样取决于所使用的编程语言；不过典型做法是让过渡表表现得像只读临时表一样，可以在触发器函数内部发出的 SQL 命令中访问。

39.2. 数据更改的可见性

如果你在触发器函数中执行 SQL 命令，而这些命令又访问该触发器所属的表，就需要了解数据可见性规则，因为这些规则决定了这些 SQL 命令是否能看到引发触发器的数据更改。简而言之：

- 语句级触发器遵循简单的可见性规则：语句所做的任何更改对语句级 BEFORE 触发器都不可见，而所有修改对语句级 AFTER 触发器都可见。
- 导致触发器触发的数据更改（插入、更新或删除）对于在行级 BEFORE 触发器中执行的 SQL 命令自然是不可见的，因为它尚未发生。
- 然而，在行级 BEFORE 触发器中执行的 SQL 命令会看到同一外层命令中先前已处理各行上的数据更改结果。这里必须小心，因为这些更改事件的顺序通常不可预测；一条影响多行的 SQL 命令可能按任意顺序访问这些行。
- 类似地，行级 INSTEAD OF 触发器会看到同一外层命令中此前触发的 INSTEAD OF 触发器所造成的数据更改结果。
- 当行级 AFTER 触发器触发时，外层命令所做的所有数据更改都已经完成，并且对被调用的触发器函数可见。

如果你的触发器函数是用任何一种标准过程语言编写的，那么只有在该函数被声明为 VOLATILE 时，上述说法才成立。被声明为 STABLE 或 IMMUTABLE 的函数在任何情况下都看不到调用命令所做的更改。

有关数据可见性规则的更多信息可见第 47.5 节。第 39.4 节中的示例展示了这些规则。

39.3. 用 C 编写触发器函数

本节说明触发器函数接口的底层细节。这些信息仅在用 C 编写触发器函数时才需要。如果你使用更高层语言，这些细节会由系统代为处理。在多数情况下，在用 C 编写触发器之前，应先考虑使用过程语言。每种过程语言的文档都会说明如何用该语言编写触发器。

触发器函数必须使用“版本 1”函数管理器接口。

当函数由触发器管理器调用时，不会向它传递任何普通参数，但会传递一个指向 TriggerData 结构体的“上下文”指针。C 函数可以通过执行下列宏来检查自己是否由触发器管理器调用：

```
CALLED_AS_TRIGGER(fcinfo)
```

它会展开为：

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

如果该宏返回真，就可以安全地把 fcinfo->context 转换为 TriggerData * 类型，并使用其所指向的 TriggerData 结构体。函数绝不能修改 TriggerData 结构体本身，也不能修改它所指向的任何数据。

struct TriggerData 定义在 commands/trigger.h：

```
typedef struct TriggerData
{
    NodeTag          type;
    TriggerEvent     tg_event;
    Relation         tg_relation;
    HeapTuple        tg_trigtuple;
    HeapTuple        tg_newtuple;
    Trigger         *tg_trigger;
    Buffer           tg_trigtuplebuf;
    Buffer           tg_newtuplebuf;
    Tuplestorestate *tg_oldtable;
    Tuplestorestate *tg_newtable;
} TriggerData;
```

其中各成员的定义如下：

type

始终是 T_TriggerData。

tg_event

描述调用函数的事件。可以使用以下宏检查 tg_event：

TRIGGER_FIRED_BEFORE(tg_event)

如果触发器在操作之前触发，则返回真。

TRIGGER_FIRED_AFTER(tg_event)

如果触发器在操作之后触发，则返回真。

TRIGGER_FIRED_INSTEAD(tg_event)

如果触发器是取代该操作而触发，则返回真。

TRIGGER_FIRED_FOR_ROW(tg_event)

如果触发器是因行级事件触发，则返回真。

TRIGGER_FIRED_FOR_STATEMENT(tg_event)

如果触发器是因语句级事件触发，则返回真。

TRIGGER_FIRED_BY_INSERT(tg_event)

如果触发器是由 INSERT 命令触发，则返回真。

TRIGGER_FIRED_BY_UPDATE(tg_event)

如果触发器是由 UPDATE 命令触发，则返回真。

TRIGGER_FIRED_BY_DELETE(tg_event)

如果触发器是由 DELETE 命令触发，则返回真。

TRIGGER_FIRED_BY_TRUNCATE(tg_event)

如果触发器是由 TRUNCATE 命令触发，则返回真。

tg_relation

指向一个描述该触发器所针对关系的结构体。有关此结构体的细节见 utils/rel.h。其中最值得关注的是 tg_relation->rd_att（关系元组的描述符）和 tg_relation->rd_rel->relname（关系名；其类型不是 char*，而是 NameData；如果需要名称的副本，请使用 SPI_getrelname(tg_relation) 取得一个 char*）。

tg_trigtuple

指向触发该触发器的那一行。这是正在被插入、更新或删除的行。如果该触发器因 INSERT 或 DELETE 而触发，那么如果你不想用另一行替换该行（在 INSERT 的情形下）或跳过该操作，就应从函数中返回它。对于外部表上的触发器，此处系统列的值未指定。

tg_newtuple

如果触发器因 UPDATE 而触发，则指向该行的新版本；如果因 INSERT 或 DELETE 而触发，则为 NULL。如果事件是 UPDATE，而你不想用另一行替换该行或跳过该操作，就必须从函数中返回它。对于外部表上的触发器，此处系统列的值未指定。

tg_trigger

一个指向 Trigger 类型结构体的指针，该结构体定义在utils/reltrigger.h：

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    char        tgenabled;
    bool        tgisinternal;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
    char       *tgqual;
    char       *tgoldtable;
    char       *tgnewtable;
} Trigger;
```

其中 tgname 是触发器名称，tgnargs 是 tgargs 中参数的数量，而tgargs 是一个指针数组，指向CREATE TRIGGER 语句中指定的参数。其他成员仅供内部使用。

tg_trigtuplebuf

包含 tg_trigtuple 的缓冲区；如果没有这样的元组，或它没有存储在磁盘缓冲区中，则为 InvalidBuffer。

tg_newtuplebuf

包含 tg_newtuple 的缓冲区；如果没有这样的元组，或它没有存储在磁盘缓冲区中，则为 InvalidBuffer。

tg_oldtable

指向一个类型为 Tuplestorestate 的结构体，其中包含零行或多行，格式由 tg_relation 指定；如果没有 OLD TABLE 过渡关系，则为 NULL 指针。

tg_newtable

指向一个类型为 Tuplestorestate 的结构体，其中包含零行或多行，格式由 tg_relation 指定；如果没有 NEW TABLE 过渡关系，则为 NULL 指针。

若要允许通过 SPI 发起的查询引用过渡表，请参见 SPI_register_trigger_data。

触发器函数必须返回一个 `HeapTuple` 指针或一个 `NULL` 指针（不是 SQL 空值，也就是说不要把 *`isNull`* 设为真）。如果你不想修改正在处理的行，就要小心地根据情况返回适当的 `tg_trigtuple` 或 `tg_newtuple`。

39.4. 一个完整的触发器示例

这里给出一个非常简单的、用 C 编写的触发器函数示例。（用过程语言编写的触发器示例可见各过程语言的文档。）

函数 `trigf` 会报告表 `ttest` 中的行数，并在命令试图向列 `x` 中插入空值时跳过实际操作。（这样，这个触发器就起到了非空约束的作用，但不会中止事务。）

首先，表定义：

```
CREATE TABLE ttest (
    x integer
);
```

下面是触发器函数的源代码：

```
#include "postgres.h"
#include "fmgr.h"
#include "executor/spi.h"       /* this is what you need to work with SPI */
#include "commands/trigger.h"   /* ... triggers ... */
#include "utils/rel.h"          /* ... and relations */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* make sure it's called as a trigger at all */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* tuple to return to executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* check for null values */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;
```

```
    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connect to SPI manager */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* get number of rows in table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) returns int8, so be careful to convert */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

编译源代码之后（见第 38.10.5 节），声明该函数和这些触发器：

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();
```

现在可以测试该触发器的行为：

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0
```

```
-- Insertion skipped and AFTER trigger is not fired

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                        ^^^^^^^^
                               remember what we said about visibility.
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                        ^^^^^^
                               remember what we said about visibility.
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                        ^^^^^^
                               remember what we said about visibility.
```

```
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

更复杂的示例可见 `src/test/regress/regress.c` 和 spi。

第 40 章 事件触发器

为了补充第 39 章中讨论的触发器机制，PostgreSQL 还提供了事件触发器。与仅附着于单个表并且只捕获 DML 事件的常规触发器不同，事件触发器在某个特定数据库中是全局的，并且能够捕获 DDL 事件。

与常规触发器类似，事件触发器可以用任何支持事件触发器的过程语言编写，也可以用 C 编写，但不能用纯 SQL 编写。

40.1. 事件触发器行为概述

只要与事件触发器关联的事件在其定义所在数据库中发生，事件触发器就会被触发。目前仅支持以下事件：ddl_command_start、ddl_command_end、table_rewrite 和 sql_drop。未来版本可能会增加对更多事件的支持。

ddl_command_start 事件发生在 CREATE、ALTER、DROP、SECURITY LABEL、COMMENT、GRANT 或 REVOKE 命令即将执行之前。在事件触发器触发之前，不会检查受影响对象究竟存在还是不存在。但有一个例外：对于以共享对象 – 数据库、角色和表空间 – 为目标的 DDL 命令或者针对事件触发器本身的命令，不会发生该事件。事件触发器机制不支持这些对象类型。ddl_command_start 也会在 SELECT INTO 命令即将执行之前发生，因为它等价于 CREATE TABLE AS。

ddl_command_end 事件发生在上述同一组命令执行之后。要获取这些 DDL 操作的更多细节，可在 ddl_command_end 事件触发器代码中使用集合返回函数 pg_event_trigger_ddl_commands()（见第 9.28 节）。注意，触发器是在这些动作已发生之后（但在事务提交之前）触发的，因此读取系统目录时，看到的已是变更后的状态。

对于任何删除数据库对象的操作，sql_drop 事件都发生在 ddl_command_end 事件触发器之前。要列出已删除的对象，可在 sql_drop 事件触发器代码中使用集合返回函数 pg_event_trigger_dropped_objects()（见第 9.28 节）。注意，触发器是在这些对象已经从系统目录中删除之后执行的，因此已无法再查找它们。

table_rewrite 事件发生在表即将因 ALTER TABLE 和 ALTER TYPE 命令中的某些操作而被重写之前。虽然还有其他控制语句也可以重写表，例如 CLUSTER 和 VACUUM，但它们不会触发 table_rewrite 事件。

事件触发器（与其他函数一样）不能在已中止的事务中执行。因此，如果 DDL 命令因错误而失败，任何关联的 ddl_command_end 触发器都不会执行。反之，如果 ddl_command_start 触发器因错误而失败，后续事件触发器都不会触发，也不会尝试执行该命令本身。类似地，如果 ddl_command_end 触发器因错误而失败，DDL 语句的效果将被回滚，就像包含该语句的事务在任何其他情况下中止时那样。

有关事件触发器机制所支持命令的完整列表，参见第 40.2 节。

事件触发器通过命令 CREATE EVENT TRIGGER 创建。为了创建事件触发器，必须先创建一个返回类型为 event_trigger 的特殊函数。该函数不需要（也不能）返回值；这个返回类型仅用于表明该函数要作为事件触发器被调用。

如果为某个特定事件定义了多个事件触发器，它们将按触发器名称的字母顺序触发。

触发器定义也可以指定一个 WHEN 条件，这样，例如，ddl_command_start 触发器就可以只针对用户希望拦截的特定命令触发。这类触发器的一个常见用途是限制用户可执行的 DDL 操作范围。

40.2. 事件触发器触发矩阵

表 40.1 列出了事件触发器所支持的所有命令。

表 40.1. 按命令标签分类的事件触发器支持

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	备注
ALTER AGGREGATE	X	X	-	-	
ALTER COLLATION	X	X	-	-	
ALTER CONVERSION	X	X	-	-	
ALTER DOMAIN	X	X	-	-	
ALTER DEFAULT PRIVILEGES	X	X	-	-	
ALTER EXTENSION	X	X	-	-	
ALTER FOREIGN DATA WRAPPER	X	X	-	-	
ALTER FOREIGN TABLE	X	X	X	-	
ALTER FUNCTION	X	X	-	-	
ALTER LANGUAGE	X	X	-	-	
ALTER LARGE OBJECT	X	X	-	-	
ALTER MATERIALIZED VIEW	X	X	-	-	
ALTER OPERATOR	X	X	-	-	
ALTER OPERATOR CLASS	X	X	-	-	
ALTER OPERATOR FAMILY	X	X	-	-	
ALTER POLICY	X	X	-	-	
ALTER PROCEDURE	X	X	-	-	
ALTER PUBLICATION	X	X	-	-	
ALTER ROUTINE	X	X	-	-	
ALTER SCHEMA	X	X	-	-	
ALTER SEQUENCE	X	X	-	-	

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	备注
ALTER SERVER	X	X	-	-	
ALTER STATISTICS	X	X	-	-	
ALTER SUBSCRIPTION	X	X	-	-	
ALTER TABLE	X	X	X	X	
ALTER TEXT SEARCH CONFIGURATION	X	X	-	-	
ALTER TEXT SEARCH DICTIONARY	X	X	-	-	
ALTER TEXT SEARCH PARSER	X	X	-	-	
ALTER TEXT SEARCH TEMPLATE	X	X	-	-	
ALTER TRIGGER	X	X	-	-	
ALTER TYPE	X	X	-	X	
ALTER USER MAPPING	X	X	-	-	
ALTER VIEW	X	X	-	-	
COMMENT	X	X	-	-	仅限本地对象
CREATE ACCESS METHOD	X	X	-	-	
CREATE AGGREGATE	X	X	-	-	
CREATE CAST	X	X	-	-	
CREATE COLLATION	X	X	-	-	
CREATE CONVERSION	X	X	-	-	
CREATE DOMAIN	X	X	-	-	
CREATE EXTENSION	X	X	-	-	
CREATE FOREIGN DATA WRAPPER	X	X	-	-	
CREATE FOREIGN TABLE	X	X	-	-	

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	备注
CREATE FUNCTION	X	X	-	-	
CREATE INDEX	X	X	-	-	
CREATE LANGUAGE	X	X	-	-	
CREATE MATERIALIZED VIEW	X	X	-	-	
CREATE OPERATOR	X	X	-	-	
CREATE OPERATOR CLASS	X	X	-	-	
CREATE OPERATOR FAMILY	X	X	-	-	
CREATE POLICY	X	X	-	-	
CREATE PROCEDURE	X	X	-	-	
CREATE PUBLICATION	X	X	-	-	
CREATE RULE	X	X	-	-	
CREATE SCHEMA	X	X	-	-	
CREATE SEQUENCE	X	X	-	-	
CREATE SERVER	X	X	-	-	
CREATE STATISTICS	X	X	-	-	
CREATE SUBSCRIPTION	X	X	-	-	
CREATE TABLE	X	X	-	-	
CREATE TABLE AS	X	X	-	-	
CREATE TEXT SEARCH CONFIGURATION	X	X	-	-	
CREATE TEXT SEARCH DICTIONARY	X	X	-	-	
CREATE TEXT SEARCH PARSER	X	X	-	-	

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	备注
CREATE TEXT SEARCH TEMPLATE	X	X	-	-	
CREATE TRIGGER	X	X	-	-	
CREATE TYPE	X	X	-	-	
CREATE USER MAPPING	X	X	-	-	
CREATE VIEW	X	X	-	-	
DROP ACCESS METHOD	X	X	X	-	
DROP AGGREGATE	X	X	X	-	
DROP CAST	X	X	X	-	
DROP COLLATION	X	X	X	-	
DROP CONVERSION	X	X	X	-	
DROP DOMAIN	X	X	X	-	
DROP EXTENSION	X	X	X	-	
DROP FOREIGN DATA WRAPPER	X	X	X	-	
DROP FOREIGN TABLE	X	X	X	-	
DROP FUNCTION	X	X	X	-	
DROP INDEX	X	X	X	-	
DROP LANGUAGE	X	X	X	-	
DROP MATERIALIZED VIEW	X	X	X	-	
DROP OPERATOR	X	X	X	-	
DROP OPERATOR CLASS	X	X	X	-	
DROP OPERATOR FAMILY	X	X	X	-	
DROP OWNED	X	X	X	-	
DROP POLICY	X	X	X	-	
DROP PROCEDURE	X	X	X	-	
DROP PUBLICATION	X	X	X	-	
DROP ROUTINE	X	X	X	-	

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	备注
DROP RULE	X	X	X	-	
DROP SCHEMA	X	X	X	-	
DROP SEQUENCE	X	X	X	-	
DROP SERVER	X	X	X	-	
DROP STATISTICS	X	X	X	-	
DROP SUBSCRIPTION	X	X	X	-	
DROP TABLE	X	X	X	-	
DROP TEXT SEARCH CONFIGURATION	X	X	X	-	
DROP TEXT SEARCH DICTIONARY	X	X	X	-	
DROP TEXT SEARCH PARSER	X	X	X	-	
DROP TEXT SEARCH TEMPLATE	X	X	X	-	
DROP TRIGGER	X	X	X	-	
DROP TYPE	X	X	X	-	
DROP USER MAPPING	X	X	X	-	
DROP VIEW	X	X	X	-	
GRANT	X	X	-	-	仅限本地对象
IMPORT FOREIGN SCHEMA	X	X	-	-	
REFRESH MATERIALIZED VIEW	X	X	-	-	
REVOKE	X	X	-	-	仅限本地对象
SECURITY LABEL	X	X	-	-	仅限本地对象
SELECT INTO	X	X	-	-	

40.3. 用 C 编写事件触发器函数

本节描述事件触发器函数接口的底层细节。只有在用 C 编写事件触发器函数时才需要这些信息。如果你使用更高级的语言，那么这些细节会由该语言处理。在大多数情况下，你应先考虑使用过程语言，再决定是否用 C 编写事件触发器。每种过程语言的文档都说明了如何在该语言中编写事件触发器。

事件触发器函数必须使用“版本 1”函数管理器接口。

当事件触发器管理器调用函数时，不会向它传递任何普通参数，而是会传入一个“上下文”指针，指向一个 EventTriggerData 结构体。C 函数可以通过执行下列宏来检查自己是否由事件触发器管理器调用：

```
CALLED_AS_EVENT_TRIGGER(fcinfo)
```

该宏会展开为：

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, EventTriggerData))
```

如果该宏返回真，那么就可以安全地将 fcinfo->context 转换为 EventTriggerData * 类型，并使用它所指向的 EventTriggerData 结构体。该函数不得修改 EventTriggerData 结构体或它所指向的任何数据。

struct EventTriggerData 定义在 commands/event_trigger.h：

```
typedef struct EventTriggerData
{
    NodeTag     type;
    const char *event;      /* event name */
    Node       *parsetree;  /* parse tree */
    const char *tag;        /* command tag */
} EventTriggerData;
```

其成员定义如下：

type

始终为 T_EventTriggerData。

event

描述调用该函数对应的事件，可为 "ddl_command_start"、"ddl_command_end"、"sql_drop"、"table_rewrite" 之一。关于这些事件的含义，参见第 40.1 节。

parsetree

指向该命令语法解析树的指针。详见 PostgreSQL 源代码。语法解析树结构可能在不另行通知的情况下发生变化。

tag

与触发该事件触发器的事件相关联的命令标签，例如 "CREATE FUNCTION"。

事件触发器函数必须返回一个 NULL 指针（不是 SQL 空值，也就是不要将 *isNull* 设为真）。

40.4. 完整的事件触发器示例

下面是一个用 C 编写的非常简单的事件触发器函数示例。（使用过程语言编写的触发器示例可在各过程语言的文档中找到。）

函数 noddl 每次被调用时都会抛出异常。事件触发器定义将该函数与 ddl_command_start 事件关联起来。其效果是阻止所有 DDL 命令运行（不包括第 40.1 节中提到的例外）。

以下是该触发器函数的源代码：

```
#include "postgres.h"

#include "commands/event_trigger.h"
#include "fmgr.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(noddl);

Datum
noddl(PG_FUNCTION_ARGS)
{
    EventTriggerData *trigdata;

    if (!CALLED_AS_EVENT_TRIGGER(fcinfo))  /* internal error */
        elog(ERROR, "not fired by event trigger manager");

    trigdata = (EventTriggerData *) fcinfo->context;

    ereport(ERROR,
            (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
             errmsg("command \"%s\" denied",
                    GetCommandTagName(trigdata->tag))));

    PG_RETURN_NULL();
}
```

编译源代码之后（见第 38.10.5 节），声明该函数和触发器：

```
CREATE FUNCTION noddl() RETURNS event_trigger
    AS 'noddl' LANGUAGE C;

CREATE EVENT TRIGGER noddl ON ddl_command_start
    EXECUTE FUNCTION noddl();
```

现在可以测试该触发器的工作情况：

```
=# \dy
                     List of event triggers
 Name  |       Event       | Owner | Enabled | Function | Tags
-------+-------------------+-------+---------+----------+------
 noddl | ddl_command_start | dim   | enabled | noddl    |
(1 row)

=# CREATE TABLE foo(id serial);
ERROR:  command "CREATE TABLE" denied
```

在这种情况下，为了能在需要时执行某些 DDL 命令，你必须删除该事件触发器或将其禁用。只在一个事务期间禁用该触发器通常更方便：

```
BEGIN;
ALTER EVENT TRIGGER noddl DISABLE;
CREATE TABLE foo (id serial);
ALTER EVENT TRIGGER noddl ENABLE;
COMMIT;
```

（请记住，针对事件触发器本身的 DDL 命令不会受事件触发器影响。）

40.5. 表重写事件触发器示例

借助 `table_rewrite` 事件，可以实现一种表重写策略，从而只允许在维护窗口内执行重写。

以下是实现这种策略的一个示例。

```
CREATE OR REPLACE FUNCTION no_rewrite()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
---
--- Implement local Table Rewriting policy:
---   public.foo is not allowed rewriting, ever
---   other tables are only allowed rewriting between 1am and 6am
---   unless they have more than 100 blocks
---
DECLARE
  table_oid oid := pg_event_trigger_table_rewrite_oid();
  current_hour integer := extract('hour' from current_time);
  pages integer;
  max_pages integer := 100;
BEGIN
  IF pg_event_trigger_table_rewrite_oid() = 'public.foo'::regclass
  THEN
        RAISE EXCEPTION 'you''re not allowed to rewrite the table %',
                        table_oid::regclass;
  END IF;

  SELECT INTO pages relpages FROM pg_class WHERE oid = table_oid;
  IF pages > max_pages
  THEN
        RAISE EXCEPTION 'rewrites only allowed for table with less than % pages',
                        max_pages;
  END IF;

  IF current_hour NOT BETWEEN 1 AND 6
  THEN
        RAISE EXCEPTION 'rewrites only allowed between 1am and 6am';
  END IF;
END;
$$;

CREATE EVENT TRIGGER no_rewrite_allowed
                  ON table_rewrite
   EXECUTE FUNCTION no_rewrite();
```

第 41 章 规则系统

本章讨论 PostgreSQL 中的规则系统。产生式规则系统在概念上很简单，但在实际使用时涉及许多微妙之处。

有些其他数据库系统定义了主动型数据库规则，它们通常表现为存储过程和触发器。在 PostgreSQL 中，这类功能同样可以用函数和触发器实现。

规则系统（更准确地说，查询重写规则系统）与存储过程和触发器完全不同。它会修改查询，使其将规则纳入考虑，然后把修改后的查询交给查询规划器进行规划和执行。它非常强大，可用于查询语言过程、视图和版本等许多用途。关于这一规则系统的理论基础及其能力，另见[ston90b]和[ong90]。

41.1. 查询树

要理解规则系统如何工作，必须先知道它在何时被调用，以及它的输入和输出是什么。

规则系统位于解析器和规划器之间。它接收解析器的输出，即一棵查询树，以及用户定义的重写规则；这些规则本身也是带有一些附加信息的查询树。它会生成零棵或多棵查询树作为结果。因此，它的输入和输出始终都是解析器本身也可能产生的东西，所以它看到的任何内容基本上都可以表示为一个SQL 语句。

那么什么是查询树？它是 SQL 语句的一种内部表示，其中构成语句的各个部分被分别存储。如果你设置了配置参数debug_print_parse、debug_print_rewritten 或 debug_print_plan，这些查询树就可以显示在服务器日志中。规则动作也以查询树的形式存储在系统目录pg_rewrite 中。它们的格式不像日志输出，但包含的却是完全相同的信息。

阅读原始查询树需要一些经验。但由于查询树的 SQL 表示已足以理解规则系统，本章不会讲解如何阅读它们。

在阅读本章中查询树的 SQL 表示时，必须能够识别语句在查询树结构中被拆分成的各个部分。查询树的组成部分有：

命令类型

这是一个简单的值，用来说明是哪一种命令（SELECT、INSERT、UPDATE、DELETE）产生了该查询树。

范围表

范围表是查询中使用的关系列表。在 SELECT 语句中，它们就是关键字FROM 后面给出的关系。

每个范围表项标识一个表或视图，并说明在查询的其他部分以哪个名称来称呼它。在查询树中，范围表项是按编号而不是按名称引用的，因此，名称重复在这里并无影响，而在SQL 语句中却会产生影响。这种情况可能出现在规则的范围表被合并之后。本章中的示例不会涉及这种情形。

结果关系

这是范围表中的一个索引，用来标识查询结果应写入哪个关系。

SELECT 查询没有结果关系。（SELECT INTO 这一特殊情况与 CREATE TABLE 后接 INSERT ... SELECT 几乎相同，这里不再单独讨论。）

对于 INSERT、UPDATE 和 DELETE 命令，结果关系就是要让修改生效的表（或视图！）。

目标列表

目标列表是定义查询结果的表达式列表。对于 SELECT，这些表达式构成查询的最终输出。它们对应于关键字SELECT 和 FROM 之间的表达式。（* 只是某个关系全部列名的缩写。解析器会把它展开为各个独立列，因此规则系统永远不会看到它。）

DELETE 命令不需要普通的目标列表，因为它们不产生任何结果。相反，规划器会向空目标列表加入一个特殊的CTID 项，以便执行器找到要删除的行。（当结果关系是普通表时，会加入CTID；如果结果关系是视图，则规则系统会改为加入一个整行变量，如第 41.2.4 节所述。）

对于 INSERT 命令，目标列表描述的是将要进入结果关系的新行。它由VALUES 子句中的表达式，或INSERT ... SELECT 里 SELECT 子句中的表达式组成。重写过程的第一步会为原始命令未赋值但带有默认值的列补上目标列表项。其余列（既没有给定值也没有默认值）将由规划器填入 null 常量表达式。

对于 UPDATE 命令，目标列表描述要替换旧行的新行。在规则系统中，它只包含命令中SET column = expression 部分的表达式。规划器会通过插入把旧行值复制到新行的表达式来处理缺失列。与 DELETE 一样，还会加入一个CTID 或整行变量，以便执行器标识要更新的旧行。

目标列表中的每一项都包含一个表达式，它可以是常量值、指向范围表中某个关系列的变量、一个参数，或者由函数调用、常量、变量、操作符等构成的表达式树。

条件

查询的条件是一个表达式，与目标列表项中的表达式很相似。该表达式的结果值是布尔值，用来说明是否应对最终结果行执行相应操作（INSERT、UPDATE、DELETE 或 SELECT）。它对应于 SQL 语句的 WHERE 子句。

连接树

查询的连接树显示了 FROM 子句的结构。对于 SELECT ... FROM a, b, c 这样的简单查询，连接树只是FROM 项的一个列表，因为允许按任意顺序连接它们。但当使用JOIN 表达式，特别是外连接时，就必须按连接所显示的顺序进行连接。在这种情况下，连接树展示的是JOIN 表达式的结构。与特定 JOIN 子句相关的限制（来自 ON 或 USING 表达式）会作为条件表达式附着在相应的连接树节点上。把顶层 WHERE 表达式也存储为附着在顶层连接树项上的一个条件，会很方便。因此，连接树实际上同时表示SELECT 的 FROM 和 WHERE 子句。

其他

查询树的其他部分，如ORDER BY 子句，这里不作关注。规则系统在应用规则时会替换其中某些项，但这与规则系统的基本原理关系不大。

41.2. 视图和规则系统

在 PostgreSQL 中，视图通过规则系统实现。实际上，以下命令：

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

与下面这两条命令基本没有区别：

```
CREATE TABLE myview (与 mytab 相同的列列表);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

因为这正是 `CREATE VIEW` 命令在内部所做的事情。这会带来一些副作用。其中之一是，在 PostgreSQL 系统目录中，视图的信息与表的信息完全相同。因此，对解析器而言，表和视图完全没有区别。它们是同一种东西：关系。

41.2.1. SELECT 规则如何工作

`ON SELECT` 规则会在所有查询上作为最后一步应用，即使给出的命令是`INSERT`、`UPDATE` 或 `DELETE` 也一样。它们与其他命令类型上的规则在语义上不同，因为它们是就地修改查询树，而不是创建新的查询树。因此我们先讨论`SELECT` 规则。

目前，一个`ON SELECT` 规则中只能有一个动作，而且它必须是一个无条件、带有`INSTEAD` 的 `SELECT` 动作。之所以有这个限制，是为了让规则足够安全，从而能够向普通用户开放；它也把 `ON SELECT` 规则限制为像视图那样工作。

本章的示例是两个进行一些计算的连接视图，以及另外一些依次使用它们的视图。最初两个视图中的一个会在后面通过为`INSERT`、`UPDATE` 和 `DELETE` 操作添加规则来定制，从而最终得到一个在行为上像真正的表、但又带有某些特殊功能的视图。作为入门示例，这并不算简单，因此会让理解变得更难一些。但与其使用许多可能令人混淆的不同示例，不如用一个示例逐步覆盖这里讨论的全部要点。

在前两节对规则系统的描述中，我们需要用到如下真实表：

```
CREATE TABLE shoe_data (
    shoename   text,          -- 主键
    sh_avail   integer,       -- 可用数量（双）
    slcolor    text,          -- 首选鞋带颜色
    slminlen   real,          -- 最小鞋带长度
    slmaxlen   real,          -- 最大鞋带长度
    slunit     text           -- 长度单位
);

CREATE TABLE shoelace_data (
    sl_name    text,          -- 主键
    sl_avail   integer,       -- 可用数量（双）
    sl_color   text,          -- 鞋带颜色
    sl_len     real,          -- 鞋带长度
    sl_unit    text           -- 长度单位
);

CREATE TABLE unit (
    un_name    text,          -- 主键
    un_fact    real           -- 换算为 cm 的系数
);
```

如你所见，它们表示的是鞋店数据。

这些视图是这样创建的：

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           least(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

创建 shoelace 视图的 CREATE VIEW 命令（这是我们这里最简单的一个例子）会创建一个关系 shoelace，并在pg_rewrite 中创建一项，说明只要查询的范围表中引用了关系shoelace，就必须应用一条重写规则。该规则没有规则条件（稍后在讨论非SELECT 规则时再谈，因为目前 SELECT 规则不能有规则条件），并且它是INSTEAD 规则。注意，规则条件与查询条件不是一回事。这里我们的规则动作带有一个查询条件。规则动作本身是一棵查询树，它是视图创建命令中 SELECT 语句的一个副本。

> **注意**
>
> 你在 pg_rewrite 项中看到的、用于NEW 和 OLD 的两个额外范围表项，在讨论SELECT 规则时暂不关注。

现在我们填充 unit、shoe_data 和 shoelace_data，并在一个视图上运行简单查询：

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

这是你能在这些视图上执行的最简单的 SELECT，因此我们借此机会说明视图规则的基础。SELECT * FROM shoelace 由解析器解释后，会生成如下查询树：

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

然后它会被交给规则系统。规则系统遍历范围表，检查其中的关系是否有相应规则。在处理 shoelace 的范围表项时（到目前为止只有这一个），它会找到带有如下查询树的_RETURN 规则：

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

为了展开该视图，重写器只需创建一个包含规则动作查询树的子查询范围表项，再用这个范围表项替换原先引用该视图的项。重写后的查询树几乎就像你直接写下了下面这条语句：

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

不过有一个区别：子查询的范围表中多了两个项，`shoelace old` 和 `shoelace new`。这些项并不直接参与查询，因为子查询的连接树或目标列表都没有引用它们。重写器利用它们保存原先存在于引用视图的范围表项中的访问权限检查信息。这样，即使在重写后的查询中不再直接使用该视图，执行器仍会检查用户是否具有访问该视图的适当权限。

这就是应用的第一条规则。规则系统接着会检查顶层查询中剩余的范围表项（本例中已经没有了），并递归检查新增子查询中的范围表项，看它们是否引用了视图。（但它不会展开`old` 或 `new`，否则就会出现无限递归！）在这个例子中，`shoelace_data` 和 `unit` 都没有重写规则，因此重写到此结束，上面的结果就是交给规划器的最终结果。

现在我们想写一个查询，找出商店里目前哪些鞋子有颜色和长度都匹配的鞋带，并且完全匹配的总双数大于等于二。

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

这一次解析器的输出是如下查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

首先应用的是用于 shoe_ready 视图的规则，它得到如下查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               least(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

同样地，用于shoe 和 shoelace 的规则也会被替换进子查询的范围表，从而得到一个三层的最终查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               least(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
                       sh.slcolor,
                       sh.slminlen,
                       sh.slminlen * un.un_fact AS slminlen_cm,
                       sh.slmaxlen,
                       sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                       sh.slunit
                  FROM shoe_data sh, unit un
                 WHERE sh.slunit = un.un_name) rsh,
               (SELECT s.sl_name,
                       s.sl_avail,
                       s.sl_color,
                       s.sl_len,
                       s.sl_unit,
                       s.sl_len * u.un_fact AS sl_len_cm
                  FROM shoelace_data s, unit u
                 WHERE s.sl_unit = u.un_name) rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail > 2;
```

这看起来可能效率不高，但规划器会通过“上拉”子查询把它折叠成单层查询树，然后像我们手工写出这些连接一样来规划它们。因此，折叠查询树属于一种优化，重写系统本身无须关心。

41.2.2. 非 SELECT 语句中的视图规则

上文对视图规则的说明没有涉及查询树中的两个细节：命令类型和结果关系。实际上，视图规则并不需要命令类型，但结果关系可能会影响查询重写器的工作方式，因为当结果关系是视图时，必须进行特殊处理。

SELECT 的查询树与其他任何命令的查询树之间只有少数差别。显然，它们的命令类型不同；对于SELECT 之外的命令，结果关系会指向结果应写入的那个范围表项。除此之外，其余部分完全相同。因此，假设有两个表t1 和 t2，都具有列a 和 b，那么下面两条语句的查询树：

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

几乎是一样的。特别是：

- 范围表中包含表 t1 和 t2 的项。
- 目标列表中都包含一个变量，该变量指向表t2 的范围表项中的列 b。
- 条件表达式会比较两个范围表项中的列 a 是否相等。
- 连接树都表示 t1 与 t2 之间的一次简单连接。

结果是，这两个查询树都会产生相似的执行计划：它们都是对这两个表的连接。对于UPDATE，规划器会把t1 中缺失的列补入目标列表，最终查询树会变成：

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

因而，执行器在这个连接上运行时会产生与下面语句完全相同的结果集：

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

但在 UPDATE 中有个小问题：执行器计划中负责连接的那一部分并不关心连接结果将被用于什么。它只是生成一个行结果集。一个是 SELECT 命令，另一个是UPDATE 命令，这一差别是在执行器更高层处理的；在那里，系统知道这是一个UPDATE，也知道结果应写入表t1。但问题在于：其中哪一行应当被新行替换？

为了解决这个问题，UPDATE（以及 DELETE）语句的目标列表中会额外加入一项：当前元组 ID（CTID）。这是一个系统列，包含该行所在的文件块号以及在块中的位置。已知表之后，就可以利用CTID 取回要更新的 t1 原始行。把 CTID 加入目标列表后，查询实际上会变成：

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

现在还要引入 PostgreSQL 的另一个细节。旧表行不会被覆盖，这也是ROLLBACK 之所以很快的原因。在 UPDATE 中，新结果行会被插入表中（去掉CTID 之后），而CTID 所指向的旧行，其行头中的cmax 和 xmax 项会被设置为当前命令计数器和当前事务 ID。于是旧行被隐藏起来，事务提交后，清理器最终就可以移除这条死行。

知道了这些以后，我们就可以用完全相同的方式把视图规则应用到任何命令上。没有区别。

41.2.3. PostgreSQL 中视图的威力

上文演示了规则系统如何把视图定义整合进原始查询树。在第二个示例中，从一个视图发出的简单SELECT 最终生成了一棵四表连接的查询树（unit 以不同名称使用了两次）。

用规则系统实现视图的好处在于：规划器能够在单棵查询树中同时看到哪些表必须被扫描、这些表之间的关系、来自视图的限制条件，以及原始查询自身的条件。即使原始查询本身已经是对若干视图的连接，情况也仍然如此。规划器必须决定执行查询的最佳路径，而它掌握的信息越多，这个决定通常就越好。PostgreSQL 实现的规则系统能够保证，到那个阶段为止，关于该查询的全部可用信息都已经集中在这里。

41.2.4. 更新视图

如果某个视图被指定为 INSERT、UPDATE 或 DELETE 的目标关系，会发生什么？如果按上文描述的方式进行替换，就会得到一棵结果关系指向子查询范围表项的查询树，而这是行不通的。不过，PostgreSQL 仍有几种方法能够支持“更新视图”这种表象。

如果子查询从单个基础关系取数并且足够简单，重写器就可以自动用底层基础关系替换该子查询，使INSERT、UPDATE 或 DELETE 以适当方式作用于基础关系。这种“足够简单”的视图称为自动可更新视图。关于哪些视图可以自动更新的详细信息，请参见 CREATE VIEW。

另一种办法是由视图上的用户自定义 INSTEAD OF 触发器来处理该操作。在这种情况下，重写的工作方式会略有不同。对于INSERT，重写器完全不处理该视图，而是让它继续作为查询的结果关系。对于UPDATE 和 DELETE，仍然需要展开视图查询，以产生命令打算更新或删除的“旧”行。因此，视图会照常展开，但查询中还会再添加一个未展开的范围表项，用来表示该视图作为结果关系时的角色。

此时出现的问题是，如何标识视图中需要更新的行。回想一下，当结果关系是表时，目标列表中会加入一个特殊的CTID 项，用来标识待更新行的物理位置。如果结果关系是视图，这就行不通了，因为视图没有CTID，它的行并没有实际物理位置。取而代之的是，对于UPDATE 或 DELETE 操作，目标列表中会加入一个特殊的wholerow 项，它会展开为该视图的全部列。执行器利用这个值把“旧”行传给 INSTEAD OF 触发器。之后由触发器根据旧行值和新行值决定应当更新什么。

还有一种可能，是由用户定义INSTEAD 规则，为视图上的INSERT、UPDATE 和 DELETE 命令指定替代动作。这些规则会重写命令，通常是把它改写成更新一个或多个表而不是视图的命令。这正是第 41.4 节的主题。

注意，规则总是先被处理，也就是在原始查询被规划和执行之前先完成重写。因此，如果一个视图上同时有INSTEAD OF 触发器以及 INSERT、UPDATE 或 DELETE 规则，那么规则会先执行；根据其结果，触发器甚至可能完全不会被用到。

对简单视图上的 INSERT、UPDATE 或 DELETE 查询，总是最后才尝试自动重写。因此，如果某个视图定义了规则或触发器，它们会覆盖自动可更新视图的默认行为。

如果该视图既没有 INSTEAD 规则，也没有INSTEAD OF 触发器，并且重写器又无法把查询自动改写成对底层基础关系的更新，那么就会抛出错误，因为执行器本身并不能直接更新视图。

41.3. 物化视图

PostgreSQL 中的物化视图和视图一样使用规则系统，但会以类似表的形式持久保存结果。下面两者之间的主要区别是：

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

以及：

```
CREATE TABLE mymatview AS SELECT * FROM mytab;
```

物化视图之后不能被直接更新，并且用于创建物化视图的查询，其存储方式与视图查询的存储方式完全相同，因此可以通过下面的命令为物化视图生成新数据：

```
REFRESH MATERIALIZED VIEW mymatview;
```

在 PostgreSQL 系统目录中，物化视图的信息与表或视图的信息完全相同。因此，对于解析器来说，物化视图也是一种关系，就像表或视图一样。当查询引用物化视图时，数据会像从表中那样直接从物化视图返回；规则只用于填充物化视图。

访问物化视图中存储的数据，往往比直接或通过视图访问底层表快得多，但这些数据并不总是最新的；不过，有时并不需要最新数据。考虑以下记录销售情况的表：

```
CREATE TABLE invoice (
    invoice_no    integer        PRIMARY KEY,
    seller_no     integer,       -- 销售人员 ID
    invoice_date  date,          -- 销售日期
    invoice_amt   numeric(13,2)  -- 销售金额
);
```

如果希望快速绘制历史销售数据的图表，就可能想要汇总数据，也可能不关心当天尚不完整的数据：

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT
      seller_no,
      invoice_date,
      sum(invoice_amt)::numeric(13,2) as sales_amt
    FROM invoice
    WHERE invoice_date < CURRENT_DATE
    GROUP BY
      seller_no,
      invoice_date;

CREATE UNIQUE INDEX sales_summary_seller
  ON sales_summary (seller_no, invoice_date);
```

这个物化视图可能适合在为销售人员创建的仪表板中显示图表。可以调度一个作业，每晚使用以下 SQL 语句更新统计数据：

```
REFRESH MATERIALIZED VIEW sales_summary;
```

物化视图的另一种用途，是让通过外部数据包装器从远程系统获取的数据能够被更快地访问。下面给出一个使用`file_fdw`的简单示例，并附带计时结果；不过由于这里使用的是本地系统缓存，因此与真正访问远程系统相比，性能差异通常会比这里展示的更大。还要注意，我们同时利用了可以在物化视图上建立索引这一能力，而`file_fdw`本身并不支持索引；对其他类型的外部数据访问，这一优势未必适用。

准备工作：

```
CREATE EXTENSION file_fdw;
CREATE SERVER local_file FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE words (word text NOT NULL)
  SERVER local_file
  OPTIONS (filename '/usr/share/dict/words');
CREATE MATERIALIZED VIEW wrd AS SELECT * FROM words;
CREATE UNIQUE INDEX wrd_word ON wrd (word);
CREATE EXTENSION pg_trgm;
CREATE INDEX wrd_trgm ON wrd USING gist (word gist_trgm_ops);
VACUUM ANALYZE wrd;
```

现在来检查一个单词的拼写。直接使用 file_fdw：

```
SELECT count(*) FROM words WHERE word = 'caterpiler';

 count
-------
     0
(1 row)
```

使用 EXPLAIN ANALYZE 可以看到：

```
 Aggregate  (cost=21763.99..21764.00 rows=1 width=0) (actual time=188.180..188.181 rows=1
 loops=1)
   ->  Foreign Scan on words  (cost=0.00..21761.41 rows=1032 width=0) (actual time=188.177..
188.177 rows=0 loops=1)
         Filter: (word = 'caterpiler'::text)
         Rows Removed by Filter: 479829
         Foreign File: /usr/share/dict/words
         Foreign File Size: 4953699
 Planning time: 0.118 ms
 Execution time: 188.273 ms
```

如果改用物化视图，查询会快得多：

```
 Aggregate  (cost=4.44..4.45 rows=1 width=0) (actual time=0.042..0.042 rows=1 loops=1)
   ->  Index Only Scan using wrd_word on wrd  (cost=0.42..4.44 rows=1 width=0) (actual time=0.
039..0.039 rows=0 loops=1)
         Index Cond: (word = 'caterpiler'::text)
         Heap Fetches: 0
 Planning time: 0.164 ms
 Execution time: 0.117 ms
```

无论采用哪种方式，这个单词的拼写都是错误的，因此来找找我们可能想要的单词。再次使用 file_fdw：

```
SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;

     word
---------------
 cater
 caterpillar
 Caterpillar
 caterpillars
 caterpillar's
 Caterpillar's
 caterer
 caterer's
 caters
 catered
(10 rows)
```

```
 Limit  (cost=11583.61..11583.64 rows=10 width=32) (actual time=1431.591..1431.594 rows=10
 loops=1)
   ->  Sort  (cost=11583.61..11804.76 rows=88459 width=32) (actual time=1431.589..1431.591
 rows=10 loops=1)
         Sort Key: ((word <-> 'caterpiler'::text))
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Foreign Scan on words  (cost=0.00..9672.05 rows=88459 width=32) (actual time=0.
057..1286.455 rows=479829 loops=1)
               Foreign File: /usr/share/dict/words
               Foreign File Size: 4953699
 Planning time: 0.128 ms
 Execution time: 1431.679 ms
```

使用物化视图：

```
 Limit  (cost=0.29..1.06 rows=10 width=10) (actual time=187.222..188.257 rows=10 loops=1)
   ->  Index Scan using wrd_trgm on wrd  (cost=0.29..37020.87 rows=479829 width=10) (actual
 time=187.219..188.252 rows=10 loops=1)
         Order By: (word <-> 'caterpiler'::text)
 Planning time: 0.196 ms
 Execution time: 198.640 ms
```

如果能够接受将远程数据定期更新到本地数据库，性能收益可能相当可观。

41.4. INSERT、UPDATE 和 DELETE 规则

定义在 INSERT、UPDATE 和 DELETE 上的规则与上一节描述的视图规则有明显的不同。首先，它们的CREATE RULE 命令允许更多：

- 它们可以没有动作。
- 它们可以有多个动作。
- 它们可以是 INSTEAD 或 ALSO（默认）。
- 伪关系 NEW 和 OLD 可以派上用场。
- 它们可以有规则条件。

第二，它们不是就地修改查询树，而是创建零棵或多棵新查询树，并且可能丢弃原始查询树。

> **小心**
>
> 在很多情况下，使用INSERT/UPDATE/DELETE 规则完成的任务，用触发器会做得更好。触发器在写法上稍微复杂一些，但它们的语义要简单得多。当原始查询包含易变函数时，规则往往会产生出人意料的结果：在执行规则的过程中，易变函数的执行次数可能比预期更多。
>
> 此外，还有一些情况根本无法由这些类型的规则支持，尤其是原始查询中包含WITH 子句，或者UPDATE 查询的 SET 列表中存在多重赋值的子 SELECT。这是因为把这些结构复制到规则查询中会导致子查询被多次求值，这与查询作者的明确意图相违背。

41.4.1. 更新规则如何工作

记住以下语法：

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

在后文中，更新规则是指定义在 INSERT、UPDATE 或 DELETE 上的规则。

当查询树的结果关系和命令类型分别等于 CREATE RULE 命令中给出的对象和事件时，规则系统就会应用更新规则。对于更新规则，规则系统会创建一个查询树列表，初始时该列表为空。动作可以有零个（关键字NOTHING）、一个或多个。为简化说明，我们先看只有一个动作的规则。这个规则可以有条件，也可以没有条件；它还可以是INSTEAD 或 ALSO（默认值）。

什么是规则条件？它是一种限制，用来说明何时执行规则动作、何时不执行。这个条件只能引用伪关系NEW 和/或 OLD，它们基本上表示作为对象给出的那个关系，只是带有特殊含义。

因此，对于只有一个动作的规则，有以下三种情况会产生相应的查询树。

没有条件，有ALSO 或 INSTEAD

规则动作的查询树，再附加原始查询树的条件

给出了条件，有ALSO

规则动作的查询树，再附加规则条件和原始查询树的条件

给出了条件，有INSTEAD

规则动作的查询树，再附加规则条件和原始查询树的条件；以及附加了规则条件取反后的原始查询树

最后，如果规则是 ALSO，就把未经更改的原始查询树加入列表。由于只有带条件的 INSTEAD 规则已经加入了原始查询树，因此，对于只有一个动作的规则，最终会得到一棵或两棵输出查询树。

对于 ON INSERT 规则，原始查询（如果未被INSTEAD 抑制）会先于规则添加的任何动作执行。这样一来，这些动作就能看到被插入的行。但对ON UPDATE 和 ON DELETE 规则，原始查询会在规则添加的动作之后执行。这就保证了这些动作能够看到将被更新或删除的行；否则，动作可能什么也做不了，因为它们找不到符合条件的行。

由规则动作生成的查询树会再次送入重写系统，随后可能还会有更多规则被应用，从而产生更多或更少的查询树。因此，一个规则的动作必须具有不同的命令类型，或者具有与该规则自身不同的结果关系。否则，这种递归过程就会陷入无限循环。（规则的递归展开会被检测出来，并作为错误报告。）

pg_rewrite 系统目录中动作里的查询树只是模板。由于它们可以引用 NEW 和 OLD 的范围表项，因此在使用前必须进行一些替换。对任何NEW 的引用，都会先在原始查询的目标列表中查找相应项。如果找到了，就用该项的表达式替换该引用。否则，NEW 就与 OLD 含义相同（对于 UPDATE），或者被替换为 null（对于INSERT）。任何对 OLD 的引用，都会被替换为对结果关系对应范围表项的引用。

在系统完成应用更新规则后，它再应用视图规则到生成的查询树上。视图无法插入新的更新动作，所以没有必要向视图重写的输出应用更新规则。

41.4.1.1. 第一个规则：逐步分析

假设我们想跟踪 shoelace_data 关系中 sl_avail 列的变化。因此，我们建立一个日志表和一条规则，使其在shoelace_data 上执行 UPDATE 时，有条件地写入一条日志记录。

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- 发生变化的鞋带
    sl_avail   integer,       -- 新的可用数量
    log_who    text,          -- 操作者
    log_when   timestamp      -- 操作时间
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

现在有人执行了：

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

然后我们查看日志表：

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

这正是我们预期的结果。后台发生的事情如下。解析器创建了查询树：

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

此时存在一条 ON UPDATE 规则 log_shoelace，它带有如下规则条件表达式：

```
NEW.sl_avail <> OLD.sl_avail
```

其动作是：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old;
```

（这看起来有点奇怪，因为通常你不能写INSERT ... VALUES ... FROM。这里的 FROM 子句只是为了表明查询树中存在用于 new 和 old 的范围表项。之所以需要这些项，是为了让 INSERT 命令查询树中的变量能够引用它们。）

该规则是一条带条件的 ALSO 规则，因此规则系统必须返回两棵查询树：修改后的规则动作，以及原始查询树。第 1 步中，原始查询的范围表会并入规则动作的查询树，得到：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data;
```

第 2 步将规则条件加进去，因此结果集被限制为sl_avail 发生变化的行：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail;
```

（这看起来更奇怪，因为INSERT ... VALUES 同样没有 WHERE 子句，但规划器和执行器处理它并无困难。反正它们本来也需要为INSERT ... SELECT 支持相同功能。）

第 3 步把原始查询树的条件加进去，把结果集进一步限制为只有那些原始查询会触及的行：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第 4 步把 NEW 引用替换为原始查询树中的目标列表项，或者替换为结果关系中相应的变量引用：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第 5 步把 OLD 引用替换为结果关系引用：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

至此就完成了。由于规则是 ALSO，我们还要输出原始查询树。简而言之，规则系统输出的是一个包含两棵查询树的列表，它们对应于以下语句：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

它们会按这个顺序执行，而这正是该规则想要达到的效果。

上述替换以及附加的条件能够保证：如果原始查询是下面这样，就不会写入任何日志记录：

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

在这种情况下，原始查询树不包含sl_avail 的目标列表项，因此NEW.sl_avail 会被 shoelace_data.sl_avail 替换。于是，规则生成的额外命令是：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

而该条件永远不可能为真。

如果原始查询修改多行，这种机制同样能够正常工作。例如，假设有人发出如下命令：

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

实际上会更新四行（sl1、sl2、sl3 和 sl4）。但 sl3 本来就已经是 sl_avail = 0。在这种情况下，原始查询树的条件不同，因此规则会产生额外的查询树：

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

这棵查询树必然会插入三条新的日志记录。这完全正确。

到这里就能看出，为什么原始查询树最后执行至关重要。如果先执行UPDATE，那么所有行都已经被设为零，记日志的INSERT 就找不到任何满足 0 <> shoelace_data.sl_avail 的行了。

41.4.2. 与视图的协作

为了防止有人像前面提到的那样对视图关系执行 INSERT、UPDATE 或 DELETE，一种简单的办法是让那些查询树直接被丢弃。因此我们可以创建如下规则：

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

如果现在某人尝试对视图关系 shoe 执行这些操作，规则系统就会应用这些规则。由于这些规则没有动作而且是INSTEAD，生成的查询树列表将为空；整个查询也就什么都不会做，因为规则系统处理完后已经没有任何内容可供优化或执行。

另一种更完善的做法，是创建一些规则，把查询树重写成在真实表上执行正确操作的查询树。要在视图shoelace 上做到这一点，我们创建下列规则：

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

如果你想在视图上支持 RETURNING 查询，就需要让规则包含计算视图行的RETURNING 子句。对于单表视图，这通常很简单；但对于像shoelace 这样的连接视图，就会有点烦琐。插入场景的一个示例如下：

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    )
    RETURNING
           shoelace_data.*,
           (SELECT shoelace_data.sl_len * u.un_fact
            FROM unit u WHERE shoelace_data.sl_unit = u.un_name);
```

请注意，这一条规则同时支持该视图上的INSERT 和 INSERT RETURNING 查询；对于普通 INSERT，RETURNING 子句会被简单忽略。

现在假定商店偶尔会收到一包鞋带，随附一份很长的部件清单。但你不想每次都手工更新 shoelace 视图。因此，我们建立两个小表：一个用于插入部件清单中的项目，另一个则采用一个特殊技巧。创建命令如下：

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

现在可以将部件清单中的数据填入 shoelace_arrive 表：

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

快速查看一下当前数据：

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

现在把到货的鞋带入库：

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

然后检查结果：

```
SELECT * FROM shoelace ORDER BY sl_name;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+----------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

从一个 INSERT ... SELECT 到这些结果，中间要经历相当长的一段过程。本章最后再来描述这一查询树转换的细节。首先，看一下解析器的输出：

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

现在应用第一条规则 shoelace_ok_ins，它会把这一输出转换成：

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

同时会丢弃针对 shoelace_ok 的原始 INSERT。这个重写后的查询会再次交给规则系统，而接下来应用的规则shoelace_upd 会生成：

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

这同样是一条 INSTEAD 规则，因此前一个查询树会被丢弃。注意，这个查询仍然使用视图 shoelace。但规则系统尚未完成，因此会继续在其上应用_RETURN 规则，于是得到：

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

最后，规则log_shoelace 被应用，生成额外的查询树：

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u,
       shoelace_data old, shoelace_data new
       shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

到这里，规则系统已经没有更多规则可用，并返回生成的查询树。

因此，我们最终得到两棵查询树，它们等效于以下SQL 语句：

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

结果是：来自一个关系的数据被插入到另一个关系中，这个插入被改写为对第三个关系的更新，再被改写为对第四个关系的更新外加在第五个关系中记录该更新，最后整个过程被化简为两个查询。

这里有个稍显难看的小细节。观察这两个查询会发现，shoelace_data 关系在范围表中出现了两次，而实际上完全可以缩减成一次。规划器不会处理这一点，因此规则系统为INSERT 输出的执行计划会是

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

而省略那个额外的范围表项则会得到

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

这会在日志表中生成完全相同的项。因此，规则系统导致了对shoelace_data 表的一次完全不必要的额外扫描。而在 UPDATE 中，同样的冗余扫描还会再发生一次。不过，能让这一切总体上工作起来，已经是一项相当艰巨的工作。

现在我们来做一个关于 PostgreSQL 规则系统及其威力的最后演示。假设你向数据库中加入了一些颜色特别的鞋带：

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

我们想建立一个视图，用来检查哪些shoelace 项在颜色上与任何鞋子都不匹配。这个视图是：

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

它的输出是：

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

现在希望将没有库存的不匹配鞋带从数据库中删除。为了给 PostgreSQL 增加一点难度，我们不直接删除，而是再创建一个视图：

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

然后这样执行：

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
             WHERE sl_name = shoelace.sl_name);
```

Voilà：

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |        80
 sl2     |        6 | black    |    100 | cm      |       100
 sl7     |        6 | brown    |     60 | cm      |        60
 sl4     |        8 | black    |     40 | inch    |     101.6
 sl3     |       10 | black    |     35 | inch    |      88.9
 sl8     |       21 | brown    |     40 | inch    |     101.6
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
 sl5     |        4 | brown    |      1 | m       |       100
 sl6     |       20 | brown    |    0.9 | m       |        90
(9 rows)
```

作用于一个视图上的 DELETE，其子查询条件总共使用了四个嵌套/连接的视图，其中一个视图自身又带有一个包含视图的子查询条件，并且还用到了计算得到的视图列，最终仍会被重写成单棵查询树，从真正的表中删除所请求的数据。

在现实世界中，大概只有很少的场景会需要这样的构造。但知道它确实能工作，总归让人安心。

41.5. 规则和权限

由于 PostgreSQL 规则系统会重写查询，因此可能访问原始查询中并未使用的其他表或视图。使用更新规则时，这甚至可能包括对表的写访问。

重写规则没有单独的所有者。关系（表或视图）的所有者会自动成为为其定义的重写规则的所有者。PostgreSQL 规则系统改变了默认访问控制系统的行为。由于规则而使用的关系都会根据规则所有者的权限进行检查，而不是调用规则的用户。这意味着用户只需要对其查询中明确命名的表/视图具有所需的权限。

例如，某个用户有一份电话号码列表，其中一部分是私人的，另一部分对办公室助理有用。该用户可以这样构造：

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO assistant;
```

除了该用户本人（以及数据库超级用户）之外，没有人可以访问phone_data 表。但由于 GRANT 的存在，助理可以对phone_number 视图执行 SELECT。规则系统会把对 phone_number 的 SELECT 重写成对 phone_data 的 SELECT。由于该用户是 phone_number 的所有者，因此也是规则的所有者，对phone_data 的读访问会按照该用户的权限进行检查，于是查询被允许。同时，对phone_number 本身的访问检查仍会执行，但这是针对调用用户进行的，因此除了用户本人和助理之外，没有其他人能使用它。

权限是逐条规则检查的。因此，目前助理是唯一能看到公开电话号码的人。但助理还可以再建立一个视图，并把该视图授予公众访问。这样，任何人都可以通过助理的视图看到phone_number 中的数据。助理做不到的是创建一个直接访问 phone_data 的视图。（其实助理可以创建，但它不会起作用，因为每次访问都会在权限检查时被拒绝。）而且，一旦用户发现助理开放了其 phone_number 视图，用户就可以撤销助理的访问权限。那样一来，对助理视图的任何访问都会立即失败。

这种逐条规则的检查看起来似乎像是一个安全漏洞，但实际上并不是。因为即使不这样工作，助理也可以建立一个与phone_number 具有相同列的表，每天把数据复制进去。那样一来，这些就是助理自己的数据，助理同样可以把访问权限授予任何人。GRANT 的含义本来就是“我信任你”。如果某个你信任的人做了上面的事，那么该重新考虑这份信任，并使用REVOKE 了。

需要注意的是，虽然视图可以用前文展示的技术来隐藏某些列的内容，但除非设置了 security_barrier 标志，否则它们不能被用来可靠地隐藏不可见行中的数据。例如，下面这个视图就是不安全的：

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

这个视图看起来似乎很安全，因为规则系统会把对phone_number 的任何 SELECT 重写为对 phone_data 的 SELECT，并附加条件，要求只处理phone 不以 412 开头的项。但是，如果用户可以创建自己的函数，就不难让规划器在NOT LIKE 表达式之前先执行用户定义函数。例如：

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END;
$$ LANGUAGE plpgsql COST 0.0000000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

phone_data 表中的每个人和电话号码都会被打印为一条 NOTICE，因为规划器会选择先执行廉价的tricky 函数，再执行更昂贵的NOT LIKE。即使禁止用户定义新函数，内置函数也可以用于类似攻击。（例如，大多数类型转换函数都会在其错误消息中包含输入值。）

类似的考虑也适用于更新规则。在上一节的示例中，示例数据库中那些表的所有者可以把 shoelace 视图上的 SELECT、INSERT、UPDATE 和 DELETE 权限授予其他用户，但对 shoelace_log 只授予 SELECT 权限。写日志记录的规则动作仍会成功执行，因此其他用户可以看到日志记录。但他们不能伪造记录，也不能操纵或删除已有记录。在这个例子里，不存在通过说服规划器改变操作顺序来破坏规则的可能性，因为唯一引用shoelace_log 的规则是一条无条件的 INSERT。在更复杂的场景中，情况未必如此。

当视图需要提供行级安全时，应为该视图应用 security_barrier 属性。这可以防止在视图完成其工作之前，将行中的值传给恶意选择的函数和操作符。例如，如果上面展示的视图按以下方式创建，就会是安全的：

```
CREATE VIEW phone_number WITH (security_barrier) AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

使用 security_barrier 创建的视图，其性能可能远低于未使用此选项的视图。通常无法避免这一点：如果最快的执行计划可能危及安全，就必须拒绝它。因此，此选项默认不启用。

在处理没有副作用的函数时，查询规划器有更大的灵活性。这类函数称为LEAKPROOF，其中包括许多简单且常用的操作符，例如大量等值操作符。规划器可以安全地允许这类函数在查询执行过程中的任意位置求值，因为即便把它们应用到用户不可见的行上，也不会泄露这些不可见行的信息。此外，不带参数的函数，或者没有从安全屏障视图接收任何参数的函数，即使未被标记为 LEAKPROOF，也可以下推，因为它们根本不会接收到来自视图的数据。相反，那些可能根据参数值抛出错误的函数（例如在发生溢出或除零时抛错的函数）就不具备防泄漏性；如果在安全视图的行过滤之前对其求值，就可能泄露关于不可见行的重要信息。

需要理解的一点是，即使一个视图是用security_barrier 选项创建的，它的安全性也只限于这样一种意义：不可见元组的内容不会被传递给可能不安全的函数。用户仍然可能通过其他方式推断不可见数据；例如，他们可以使用EXPLAIN 查看查询计划，或者测量针对该视图执行查询所需的时间。恶意攻击者或许能够推断出不可见数据的大致数量，甚至获得有关数据分布或高频值的一些信息（因为这些因素可能影响计划运行时间，甚至影响计划的选择，因为它们同样反映在优化器统计信息中）。如果这类“隐通道”攻击值得担心，那么向任何人授予这类数据的访问权限本身就可能是不明智的。

41.6. 规则和命令状态

PostgreSQL 服务器会为收到的每条命令返回一个命令状态字符串，例如INSERT 149592 1。在没有规则参与时，这很简单；但当查询被规则重写时会发生什么呢？

规则对命令状态的影响如下：

- 如果该查询没有无条件的 INSTEAD 规则，那么原始给出的查询就会照常执行，并返回它自身的命令状态。（但请注意，如果存在任何带条件的INSTEAD 规则，那么这些规则条件的取反会被加到原始查询上。这可能会减少它处理的行数，从而影响报告的状态。）

- 如果该查询存在任意无条件的 INSTEAD 规则，那么原始查询将完全不会执行。在这种情况下，服务器会在由INSTEAD 规则（带条件或不带条件）插入、且与原始查询具有相同命令类型（INSERT、UPDATE 或 DELETE）的查询中，返回最后一条查询的命令状态。如果没有任何规则添加出满足这些条件的查询，那么返回的命令状态会显示原始查询类型，并且把行计数和 OID 字段都置零。

在第二种情况下，程序员可以通过为所需的INSTEAD 规则指定一个在活动规则中按字母顺序排在最后的规则名，来确保由它设置命令状态，因为它会最后被应用。

41.7. 规则与触发器

许多能够用触发器完成的事情，同样也可以用PostgreSQL 规则系统实现。不能用规则实现的内容之一是某些约束，尤其是外键。你可以放置一条带条件的规则：当某列的值没有出现在另一张表中时，把命令重写成NOTHING。但那样做会悄无声息地丢弃数据，这并不是好主意。如果需要检查值的有效性，并在值无效时生成错误消息，就必须使用触发器。

在本章中，我们重点讨论了用规则来更新视图。本章中所有更新规则的示例，也都可以用视图上的INSTEAD OF 触发器来实现。编写这类触发器通常比编写规则更容易，尤其是在需要用复杂逻辑执行更新时。

对于两者都能实现的事情，哪一种更好取决于数据库的使用方式。触发器会对每个受影响的行触发一次，而规则会修改查询，或生成额外的查询。因此，如果一条语句会影响很多行，那么发出一条额外命令的规则往往会比触发器更快，因为触发器必须为每一行调用一次，并反复判断该做什么。不过，触发器方法在概念上要比规则方法简单得多，也更容易让新手写对。

下面通过一个例子说明，在一种具体情形中选择规则或触发器会有什么不同。这里有两个表：

```
CREATE TABLE computer (
    hostname        text,    -- 已建索引
    manufacturer    text     -- 已建索引
);

CREATE TABLE software (
    software        text,    -- 已建索引
    hostname        text     -- 已建索引
);
```

两个表都有数千行，并且 hostname 上的索引都是唯一索引。规则或触发器应实现这样一个约束：从 software 中删除引用了已删除计算机的行。触发器会使用以下命令：

```
DELETE FROM software WHERE hostname = $1;
```

由于从 computer 中删除的每一行都会调用触发器，因此触发器可以为这条命令准备并保存执行计划，并通过参数传入 hostname 值。规则则会写成：

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

现在我们来看不同类型的删除。对于下面这种情况：

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

`computer` 表会通过索引扫描（很快），由触发器发出的命令也会使用索引扫描（同样很快）。规则产生的额外查询是：

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

由于已经建立了合适的索引，规划器将创建如下计划：

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

因此，触发器实现和规则实现之间的速度差异不会太大。

在下一个删除场景中，我们想去掉全部 2000 台hostname 以 old 开头的计算机。有两种命令可以做到这件事。其中一种是：

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

规则添加出来的命令将是：

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
                     AND software.hostname = computer.hostname;
```

其计划为：

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

另一个可能的命令是：

```
DELETE FROM computer WHERE hostname ~ '^old';
```

它会让规则所添加的命令得到下面这个执行计划：

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

这说明，当多个条件表达式通过AND 组合在一起时，规划器并没有意识到，对computer 中 hostname 的限制同样也可以用于 software 上的索引扫描，而在该命令的正则表达式版本里它却能做到。触发器会对需要删除的 2000 台旧计算机中的每一台各调用一次，这意味着对 computer 做一次索引扫描，并对software 做 2000 次索引扫描。规则实现则用两条使用索引的命令完成这件事。不过，在顺序扫描这种情况下，规则是否仍然更快，还取决于software 表的总体大小。即使所有索引块很快都会进入缓存，通过 SPI 管理器执行来自触发器的 2000 条命令仍然要耗费一些时间。

我们要看的最后一个命令是：

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

同样，这也可能导致从computer 中删除很多行。因此，触发器同样会通过执行器运行很多条命令。规则生成的命令则是：

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                       AND software.hostname = computer.hostname;
```

这个命令的计划又将是在两个索引扫描上的嵌套循环，只不过使用了computer 上的另一个索引：

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

在上述任何一种情况下，规则系统产生的额外命令都或多或少与该命令影响的行数无关。

总结来说，只有当规则动作导致了规模很大且条件很差的连接，而规划器又无能为力时，规则才会明显慢于触发器。

第 42 章 过程语言

PostgreSQL 允许使用 SQL 和 C 之外的其他语言编写用户定义的函数。这些语言统称为过程语言（PL）。对于用过程语言编写的函数，数据库服务器并不内置关于如何解释函数源文本的知识。相反，这项任务会交给一个了解该语言细节的专门的调用处理器。该调用处理器既可以自行完成解析、语法分析、执行等全部工作，也可以在PostgreSQL 与某种现有编程语言实现之间充当“粘合剂”。与其他任何 C 函数一样，调用处理器本身也是一个被编译进共享对象并按需装载的 C 语言函数。

标准 PostgreSQL 发行版当前提供四种过程语言：PL/pgSQL（第 43 章）、PL/Tcl（第 44 章）、PL/Perl（第 45 章）以及 PL/Python（第 46 章）。另有一些可用的过程语言并未包含在核心发行版中。附录 H 提供了查找它们的信息。此外，用户还可以自行定义其他语言；开发新过程语言的基础知识见第 56 章。

42.1. 安装过程语言

过程语言必须在每个要使用它的数据库中“安装”。不过，安装在数据库`template1` 中的过程语言会自动在随后创建的所有数据库中可用，因为它们在`template1` 中的条目会由`CREATE DATABASE` 复制。因此，数据库管理员可以决定哪些数据库提供哪些语言，并且如果需要，还可以让某些语言默认可用。

对于标准发行版附带的语言，只需执行 `CREATE EXTENSION` *`language_name`*，即可将该语言安装到当前数据库中。下文所述的手工过程只建议用于安装那些尚未打包为扩展的语言。

手工安装过程语言

在数据库中安装过程语言需要五个步骤，且必须由数据库超级用户执行。多数情况下，所需的 SQL 命令都应打包成某个“扩展”的安装脚本，这样就可以用`CREATE EXTENSION` 来执行它们。

1. 该语言调用处理器的共享对象必须先编译好并安装到合适的库目录中。这与构建和安装含有普通用户定义 C 函数的模块的方法相同；见第 38.10.5 节。该语言调用处理器往往还会依赖一个提供实际编程语言引擎的外部库；如果是这样，该库也必须安装。

2. 必须用如下命令声明该调用处理器：

   ```
   CREATE FUNCTION handler_function_name()
       RETURNS language_handler
       AS 'path-to-shared-object'
       LANGUAGE C;
   ```

   特殊返回类型 `language_handler` 会告诉数据库系统，该函数返回的不是某种已定义的 SQL 数据类型，并且不能在 SQL 语句中直接使用。

3. (可选的) 可选地，语言调用处理器可以提供一个“内联”处理器函数，用于执行用该语言编写的匿名代码块（DO 命令）。如果该语言提供了内联处理器函数，可用类似下面的命令声明它：

```
CREATE FUNCTION inline_function_name(internal)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C;
```

4. (可选的) 可选地，语言调用处理器可以提供一个“验证器”函数，用于在不实际执行的情况下检查函数定义是否正确。如果存在验证器函数，CREATE FUNCTION 就会调用它。如果该语言提供了验证器函数，可用类似下面的命令声明它：

```
CREATE FUNCTION validator_function_name(oid)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C STRICT;
```

5. 最后，必须用如下命令声明该 PL：

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language_name
    HANDLER handler_function_name
    [INLINE inline_function_name]
    [VALIDATOR validator_function_name] ;
```

   可选关键字 TRUSTED 表示，该语言不会授予用户原本不具备的数据访问能力。受信任的语言是为普通数据库用户（即没有超级用户权限的用户）设计的，并允许他们安全地创建函数和过程。由于 PL 函数是在数据库服务器内部执行的，因此只有那些不允许访问数据库服务器内部或文件系统的语言才应带有TRUSTED 标记。PL/pgSQL、PL/Tcl 以及 PL/Perl 被认为是受信任的；而 PL/TclU、PL/PerlU 以及 PL/PythonU 旨在提供无限制的功能，因此不应被标记为受信任的。

例 42.1 展示了以 PL/Perl 为例时，手工安装过程是如何进行的。

例 42.1. 手工安装 PL/Perl

下面的命令告诉数据库服务器到哪里查找 PL/Perl 语言调用处理器函数的共享对象：

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
    '$libdir/plperl' LANGUAGE C;
```

PL/Perl 具有内联处理器函数和验证器函数，因此我们也声明它们：

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
    '$libdir/plperl' LANGUAGE C;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
    '$libdir/plperl' LANGUAGE C STRICT;
```

下面的命令：

```
CREATE TRUSTED PROCEDURAL LANGUAGE plperl
    HANDLER plperl_call_handler
    INLINE plperl_inline_handler
    VALIDATOR plperl_validator;
```

随后定义了对于语言属性为 `plperl` 的函数和过程，应调用前面声明的函数。

在默认的 PostgreSQL 安装中，PL/pgSQL 语言的调用处理器会被构建并安装到“库”目录中；此外，PL/pgSQL 语言本身也安装在所有数据库中。如果在构建时配置了 Tcl 支持，那么 PL/Tcl 和 PL/TclU 的调用处理器会被构建并安装到该库目录中，但这两种语言本身默认并不会安装到任何数据库中。同样，如果在构建时配置了 Perl 支持，就会构建并安装 PL/Perl 和 PL/PerlU 的调用处理器；如果在构建时配置了 Python 支持，则会安装 PL/PythonU 的调用处理器，但这些语言默认也不会安装到任何数据库中。

第 43 章 PL/pgSQL – SQL 过程语言

43.1. 概述

PL/pgSQL 是一种用于 PostgreSQL 数据库系统的可载入过程语言。PL/pgSQL 的设计目标，是创建一种具备以下特性的可载入过程语言：

- 可用于创建函数、过程和触发器，
- 为 SQL 语言增加控制结构，
- 可以执行复杂计算，
- 继承所有用户定义的类型、函数、过程和操作符，
- 可定义为受服务器信任的语言，
- 便于使用。

用 PL/pgSQL 创建的函数，可以用于任何可以使用内置函数的地方。例如，可以创建带有复杂条件逻辑的计算函数，随后用它们定义操作符，或者把它们用于索引表达式。

在 PostgreSQL 9.0 及更高版本中，PL/pgSQL 默认安装。不过它仍然是一个可载入模块，因此对安全性特别敏感的管理员也可以选择移除它。

43.1.1. 使用 PL/pgSQL 的优点

SQL 是 PostgreSQL 以及大多数其他关系数据库使用的查询语言。它具有可移植性，也容易学习。但每条 SQL 语句都必须由数据库服务器单独执行。

这意味着客户端应用必须把每个查询发送给数据库服务器，等待其处理完成，接收并处理结果，做一些计算，然后再向服务器发送更多查询。这一切都会产生进程间通信开销；如果客户端与数据库服务器不在同一台机器上，还会额外产生网络开销。

借助 PL/pgSQL，你可以把一段计算逻辑和一系列查询放在数据库服务器内部执行。这样既具备了过程语言的能力，也保留了 SQL 的易用性，同时还能显著减少客户端/服务器通信开销。

- 消除了客户端和服务器之间额外的往返通信
- 客户端不需要的中间结果不必在服务器与客户端之间编组或传输
- 可以避免多轮查询解析

与不使用存储函数的应用相比，这可能带来相当可观的性能提升。

此外，通过 PL/pgSQL 你可以使用 SQL 的全部数据类型、操作符和函数。

43.1.2. 支持的参数和结果数据类型

用 PL/pgSQL 编写的函数，可以接受服务器支持的任意标量或数组数据类型作为参数，也可以返回这些类型中的任意一种。它们还可以接受或返回任何按名称指定的复合类型（行类型）。也可以把 PL/pgSQL 函数声明为接受 `record`，这表示任意复合类型都可作为输入；或者声明为返回 `record`，这表示结果是一种行类型，其列由调用查询中的指定内容决定，如第 7.2.1.4 节所述。

PL/pgSQL 函数还可以通过使用 `VARIADIC` 标记声明为接受可变数量的参数。这与 SQL 函数中的行为完全相同，详见第 38.5.5 节。

PL/pgSQL 函数也可以声明为接受和返回多态类型 anyelement、anyarray、anynonarray、anyenum 和 anyrange。多态函数处理的实际数据类型可以在不同调用之间变化，如第 38.2.5 节所述。示例见第 43.3.1 节。

PL/pgSQL 函数还可以声明为返回任意可单独返回的数据类型的“集合”（或表）。这类函数可以通过对结果集中的每个目标元素执行 RETURN NEXT 生成输出，也可以通过 RETURN QUERY 输出某个查询的结果。

最后，如果某个 PL/pgSQL 函数没有有意义的返回值，可以将其声明为返回 void。（另一种做法是在这种情况下把它写成过程。）

PL/pgSQL 函数也可以声明为使用输出参数，而不是显式指定返回类型。这并没有为该语言增加新的基本能力，但通常会更方便，尤其是在需要返回多个值时。RETURNS TABLE 记法也可以用来代替 RETURNS SETOF。

具体示例见第 43.3.1 节和第 43.6.1 节。

43.2. PL/pgSQL 的结构

通过执行 CREATE FUNCTION 命令，可以把用 PL/pgSQL 编写的函数定义到服务器中。这类命令通常类似于：

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE plpgsql;
```

就 CREATE FUNCTION 而言，函数体只是一个字符串字面量。编写函数体时，通常最好使用美元引用（见第 4.1.2.4 节），而不是普通的单引号语法。如果不使用美元引用，函数体中的任何单引号或反斜线都必须通过双写进行转义。本章几乎所有示例的函数体都使用了美元引用。

PL/pgSQL 是一种块结构语言。函数体的完整文本必须是一个块。块的定义如下：

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

块中的每个声明和每条语句都以分号结束。如上所示，嵌套在其他块中的块，必须在 END 之后写分号；但是结束整个函数体的最后一个 END 不需要分号。

> **提示**
>
> 一个常见错误是在 BEGIN 后面立刻写分号。这是不正确的，并会导致语法错误。

只有在你希望通过 EXIT 语句标识某个块，或者希望用块名限定块内声明的变量名时，才需要 *label*。如果在 END 之后写了标签，它必须与块开始处的标签一致。

所有关键字都不区分大小写。标识符除非用双引号括起，否则会像普通 SQL 命令中那样被隐式转换为小写。

PL/pgSQL 代码中的注释与普通 SQL 相同。双连字符（--）开始一段持续到行尾的注释。/* 开始一段块注释，它会持续到匹配的 */。块注释可以嵌套。

块的语句部分中的任意语句都可以是一个子块。子块可用于逻辑分组，或者把变量局部化到一小组语句中。在子块存续期间，子块中声明的变量会遮蔽外层块中的同名变量。不过，如果使用块标签限定外层变量名，仍然可以访问它们。例如：

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- 创建一个子块
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

> **注意**
>
> 实际上，任何 PL/pgSQL 函数体外面都包着一个隐藏的“外层块”。该块提供函数参数（如果有）的声明，以及一些诸如 FOUND 之类的特殊变量（见第 43.5.5 节）。这个外层块带有函数名标签，这意味着参数和特殊变量可以用函数名来限定。

需要特别注意，不要把 PL/pgSQL 中用于分组语句的 BEGIN/END 与用于事务控制的同名 SQL 命令混淆。PL/pgSQL 中的 BEGIN/END 只用于分组，不会开始或结束事务。关于在 PL/pgSQL 中管理事务的信息，参见第 43.8 节。另外，包含 EXCEPTION 子句的块实际上会形成一个可回滚而不影响外层事务的子事务。详见第 43.6.8 节。

43.3. 声明

在一个块中使用的所有变量，都必须在该块的声明部分声明。（唯一的例外是：在整数范围上迭代的 FOR 循环变量会被自动声明为整数变量；同样，在游标结果上迭代的 FOR 循环变量会被自动声明为记录变量。）

PL/pgSQL 变量可以是任意 SQL 数据类型，例如 integer、varchar 和 char。

这里是变量声明的一些示例：

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

一个变量声明的一般语法是：

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | := | = }
 expression ];
```

如果给定 DEFAULT 子句，它会指定进入该块时赋给该变量的初始值。如果没有给出 DEFAULT 子句，则变量会被初始化为 SQL 空值。CONSTANT 选项会阻止该变量在初始化之后再次被赋值，因此其值在整个块的持续期间保持不变。COLLATE 选项指定该变量使用的排序规则（见第 43.3.6 节）。如果指定了 NOT NULL，给该变量赋空值将导致运行时错误。所有声明为 NOT NULL 的变量都必须指定非空默认值。等号（=）可以代替兼容 PL/SQL 的 :=。

变量的默认值会在每次进入该块时重新计算并赋给该变量，而不是每次函数调用只计算一次。因此，例如把 now() 赋给一个 timestamp 类型变量，会使该变量得到当前函数调用时的时间，而不是函数预编译时的时间。

示例：

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

43.3.1. 声明函数参数

传递给函数的参数使用标识符 $1、$2 等命名。也可以为这些 $*n* 形式的参数名声明别名，以提高可读性。之后既可以使用别名，也可以使用数字标识符来引用参数值。

有两种方式可以创建别名。推荐的方式是在 CREATE FUNCTION 命令中直接为参数命名。例如：

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

另一种方式是使用声明语法显式声明别名。

```
name ALIAS FOR $n;
```

同样的示例，按这种写法如下：

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

> **注意**
>
> 这两个示例并不完全等价。在第一种情况下，subtotal 可以写成 sales_tax.subtotal 来引用；但在第二种情况下则不行。（如果我们给内层块附上一个标签，那么 subtotal 可以用那个标签来限定。）

更多一些示例：

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- 这里是一些使用 v_string 和 index 的计算
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

当 PL/pgSQL 函数声明了输出参数时，输出参数也会像普通输入参数一样获得 `$n` 名称和可选别名。输出参数本质上是一个初始值为 NULL 的变量，应在函数执行期间给它赋值。该参数的最终值就是返回值。例如，销售税的示例也可以这样写：

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

注意这里省略了 `RETURNS real` – 当然也可以写上，但那只是冗余。

当需要返回多个值时，输出参数尤其有用。下面是一个简单示例：

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

如第 38.5.4 节所述，这实际上会为函数结果创建一个匿名记录类型。如果写了 `RETURNS` 子句，它必须是 `RETURNS record`。

声明 PL/pgSQL 函数的另一种方式是使用 `RETURNS TABLE`，例如：

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

这与声明一个或多个 `OUT` 参数并指定 `RETURNS SETOF` *sometype* 完全等效。

如果一个 PL/pgSQL 函数的返回类型被声明为多态类型（`anyelement`, `anyarray`, `anynonarray`, `anyenum` 或 `anyrange`），就会创建特殊参数`$0`。其数据类型是函数的实际返回类型，根据实际输入类型推导得出（见第 38.2.5 节）。这使函数可以访问其实际返回类型，方式见第 43.3.3 节。`$0` 被初始化为 null，函数可以修改它，因此需要时可以用它保存返回值，但这并不是必需的。`$0` 也可以设置别名。例如，下面的函数适用于任何具有+操作符的数据类型：

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

把一个或多个输出参数声明为多态类型，也可以达到同样的效果。在这种情况下，不使用特殊参数 $0，输出参数本身就承担相同作用。例如：

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

43.3.2. ALIAS

newname ALIAS FOR *oldname*;

ALIAS 语法比上一节所展示的更一般化：你可以为任何变量声明别名，而不只是函数参数。它最主要的实际用途，是为那些名称预先固定的变量指定另一个名字，例如触发器函数中的 NEW 或 OLD。

示例：

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

由于 ALIAS 为同一个对象提供了两种命名方式，滥用它会让代码变得混乱。最好只把它用于改写那些预先固定的名称。

43.3.3. 复制类型

variable%TYPE

%TYPE 提供变量或表列的数据类型。可以用它声明将保存数据库值的变量。例如，假设有一个名为user_id 的列，位于users 表中。要声明一个数据类型与 users.user_id 相同的变量，可以写：

```
user_id users.user_id%TYPE;
```

使用 %TYPE 的好处是，你不必知道所引用结构的实际数据类型；更重要的是，如果被引用项的数据类型将来发生变化（例如把 user_id 的类型从 integer 改成 real），你可能就不需要修改函数定义。

%TYPE 在多态函数中特别有价值，因为内部变量所需的数据类型可能在不同调用之间变化。可以把 %TYPE 应用到函数参数或结果占位符上，以创建合适的变量。

43.3.4. 行类型

name table_name%ROWTYPE;
name composite_type_name;

复合类型的变量称为行变量（或行类型变量）。只要查询的列集合与该变量声明的类型相匹配，这种变量就可以保存 SELECT 或 FOR 查询结果中的整行。行值的各个字段可以使用通常的点号记法访问，例如 rowvar.field。

行变量既可以通过 *table_name*%ROWTYPE 记法声明为与现有表或视图的行具有相同类型，也可以通过给出某个复合类型的名称来声明。（由于每个表都有一个同名的关联复合类型，所以在 PostgreSQL 中实际上写不写 %ROWTYPE 并无区别；不过带 %ROWTYPE 的形式可移植性更好。）

函数参数也可以是复合类型（完整的表行）。在这种情况下，相应的标识符 $*n* 就是一个行变量，并且可以从中选取字段，例如 $1.user_id。

下面是一个使用复合类型的示例。table1 和 table2 是已经存在的表，它们至少包含下面提到的字段：

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

43.3.5. 记录类型

```
name RECORD;
```

记录变量与行类型变量类似，但没有预定义结构。它会在 SELECT 或 FOR 命令为其赋值时采用相应行的实际结构。记录变量的内部结构在每次被赋值时都可能变化。其后果是：在记录变量第一次被赋值之前，它没有任何子结构，任何试图访问其中字段的行为都会引发运行时错误。

注意，RECORD 并不是真正的数据类型，它只是一个占位符。还需要认识到，PL/pgSQL 函数被声明为返回 record，与记录变量并不是完全相同的概念，尽管这样的函数可能会用记录变量保存结果。这两种情况下，在编写函数时都不知道实际的行结构；但对于返回 record 的函数，实际结构会在解析调用查询时确定，而记录变量的行结构则可以在运行过程中随时变化。

43.3.6. PL/pgSQL 变量的排序规则

当 PL/pgSQL 函数具有一个或多个支持排序规则的数据类型参数时，每次函数调用都会根据分配给实际参数的排序规则确定出一个排序规则，如第 23.2 节所述。如果该排序规则能成功确定出来（即参数之间的隐式排序规则没有冲突），那么所有支持排序规则的参数都会被视为隐式带有该排序规则。这会影响函数中那些受排序规则影响的操作。例如，考虑

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

第一次调用 `less_than` 时，比较会使用 `text_field_1` 与 `text_field_2` 的共同排序规则；第二次则会使用 `C` 排序规则。

此外，确定出的排序规则也会被视为任何支持排序规则的数据类型局部变量的排序规则。因此，即使把这个函数写成下面这样，其行为也不会有任何不同：

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
$$ LANGUAGE plpgsql;
```

如果函数没有支持排序规则的数据类型的参数，或者无法为它们确定共同排序规则，那么参数和局部变量将使用其数据类型的默认排序规则（通常是数据库默认排序规则，但对于域类型变量也可能不同）。

通过在支持排序规则的数据类型局部变量的声明中加入 `COLLATE` 选项，可以为其指定不同的排序规则，例如

```
DECLARE
    local_a text COLLATE "en_US";
```

这个选项会覆盖按上述规则原本应赋给该变量的排序规则。

当然，如果某个函数希望在特定操作中强制使用特定排序规则，也可以在函数内部显式写出 COLLATE 子句。例如：

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

这会覆盖表达式中表列、参数或局部变量所关联的排序规则，就像在普通 SQL 命令中一样。

43.4. 表达式

在 PL/pgSQL 语句中使用的所有表达式都由服务器的主 SQL 执行器处理。例如，编写如下PL/pgSQL 语句时：

```
IF expression THEN ...
```

PL/pgSQL 会把如下查询交给主 SQL 引擎，以对表达式求值：

```
SELECT expression
```

在形成 SELECT 命令时，所有出现的PL/pgSQL 变量名都会被替换为参数，详见第 43.11.1 节。这样，SELECT 的查询计划只需准备一次，随后就可以在变量值不同时重复使用。因此，首次使用表达式时，实际发生的事情本质上是执行一个PREPARE 命令。例如，如果已经声明了两个整数变量x 和 y，并写下：

```
IF x < y THEN ...
```

背后发生的事情就等同于：

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

随后，对该预备语句调用EXECUTE，以完成每次IF 语句的执行，并把PL/pgSQL 变量的当前值作为参数值提供。通常，这些细节对PL/pgSQL 用户并不重要，但在诊断问题时了解它们很有帮助。更多信息见第 43.11.2 节。

43.5. 基本语句

在本节和后续各节中，我们描述PL/pgSQL 能够明确理解的所有语句类型。凡是不被识别为这些类型之一的内容，都被视为 SQL 命令并发送给主数据库引擎执行，详见第 43.5.2 节和第 43.5.3 节。

43.5.1. 赋值

向 PL/pgSQL 变量赋值的写法为：

```
variable { := | = } expression;
```

如前所述，这种语句中的表达式通过发送给主数据库引擎的 SQLSELECT 命令求值。表达式必须产生单个值（如果变量是行变量或记录变量，也可以是行值）。目标变量可以是简单变量（可以用块名限定）、行变量或记录变量的字段，或者由简单变量或字段表示的数组中的元素。可以使用等号（=）代替符合 PL/SQL 的:=。

如果该表达式的结果数据类型不匹配变量的数据类型，该值将被强制为变量的类型，就好像做了赋值类型转换一样（见第 10.4 节）。如果没有用于所涉及到的数据类型的赋值类型转换可用，PL/pgSQL 解释器将尝试以文本的方式转换结果值，也就是在应用结果类型的输出函数之后再应用变量类型的输入函数。注意如果结果值的字符串形式无法被输入函数所接受，这可能会导致由输入函数产生的运行时错误。

示例：

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

43.5.2. 执行没有结果的命令

对于不返回行的 SQL 命令，例如没有RETURNING 子句的 INSERT，只需写出命令，就可以在PL/pgSQL 函数中执行它。

命令文本中出现的任何 PL/pgSQL 变量名都被视为参数，然后在运行时将变量的当前值作为参数值提供。这与前面描述的表达式处理完全相同；详情见第 43.11.1 节。

以这种方式执行 SQL 命令时，PL/pgSQL 可能会缓存和复用命令的执行计划，详见第 43.11.2 节。

有时，对表达式或SELECT 查询求值但丢弃结果会很有用，例如调用有副作用但没有有用结果值的函数时。要在PL/pgSQL 中这样做，可以使用PERFORM 语句：

```
PERFORM query;
```

这会执行 *query* 并丢弃结果。编写 *query* 的方式与编写 SQLSELECT 命令相同，只需把开头的关键字SELECT 替换为 PERFORM。对于 WITH 查询，使用PERFORM，然后用圆括号括住查询。（此时查询只能返回一行。）PL/pgSQL 变量会被替换到查询中，方式与不返回结果的命令相同，计划也会以相同方式缓存。另外，特殊变量FOUND 在查询至少产生一行时设为真，没有产生行时设为假（见第 43.5.5 节）。

> **注意**
>
> 我们可能期望直接写 SELECT 能实现这个结果，但是当前唯一被接受的方式是PERFORM。一个能返回行的 SQL 命令（例如 SELECT）将被当成一个错误拒绝，除非它像下一节中讨论的有一个INTO 子句。

一个示例：

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

43.5.3. 执行返回单行结果的查询

产生单行（可能有多列）结果的 SQL 命令，其结果可以赋给记录变量、行类型变量或标量变量列表。方法是在基本 SQL 命令中添加INTO 子句。例如：

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

其中，*target* 可以是记录变量、行变量，或逗号分隔的简单变量及记录/行字段列表。PL/pgSQL 变量会被替换到查询的其余部分，计划也会被缓存，正如上面描述的不返回行的命令一样。这适用于SELECT，INSERT/UPDATE/DELETE 带有 RETURNING 的情况，以及返回行集结果的工具命令（例如EXPLAIN）。除了 INTO 子句以外，SQL 命令的写法与在PL/pgSQL 之外的写法相同。

> **提示**
>
> 注意带 INTO 的 SELECT 的这种解释和 PostgreSQL 常规的 SELECT INTO 命令有很大的不同，后者的INTO 目标是一个新创建的表。如果你想要在一个 PL/pgSQL 函数中从一个 SELECT 的结果创建一个表，请使用语法CREATE TABLE ... AS SELECT。

如果使用行变量或变量列表作为目标，查询结果列的数量和数据类型必须与目标结构完全匹配，否则会发生运行时错误。如果目标是记录变量，它会自动将自身配置为查询结果列的行类型。

INTO 子句几乎可以出现在 SQL 命令中的任何位置。通常它被写成刚好在 SELECT 命令中的 *select_expressions* 列表之前或之后，或者在其他命令类型的命令最后。我们推荐你遵循这种惯例，以防PL/pgSQL 的解析器在未来的版本中变得更严格。

如果 STRICT 没有在 INTO 子句中指定，那么*target* 会被设为查询返回的第一行；如果查询没有返回行，则设为空值。（注意，“第一行”并没有明确定义，除非使用了ORDER BY。）第一行之后的所有结果行都会被丢弃。可以检查特殊的 FOUND 变量（见第 43.5.5 节），以确定是否返回了行：

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

如果指定 STRICT 选项，查询必须恰好返回一行，否则会报告运行时错误：NO_DATA_FOUND（没有行）或 TOO_MANY_ROWS（多于一行）。如果希望捕获错误，可以使用异常块，例如：

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

成功执行带有 STRICT 的命令总会将 FOUND 设为真。

对于带有 RETURNING 的 INSERT/UPDATE/DELETE，即使没有指定STRICT，PL/pgSQL 也会针对多于一个返回行的情况报告一个错误。这是因为没有类似于 ORDER BY 的选项可以用来决定应该返回哪个被影响的行。

如果 print_strict_params 已为该函数启用，那么当不满足STRICT 要求而抛出错误时，错误消息的DETAIL 部分将包含传给查询的参数信息。可以为所有函数更改 print_strict_params 设置，方法是设置plpgsql.print_strict_params，不过只会影响随后编译的函数。也可以通过编译器选项逐函数启用，例如：

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END;
$$ LANGUAGE plpgsql;
```

失败时，该函数可能产生如下错误消息：

```
ERROR:  query returned no rows
DETAIL:  parameters: $1 = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_userid(text) line 6 at SQL statement
```

> **注意**
>
> STRICT 选项匹配 Oracle PL/SQL 的 SELECT INTO 和相关语句的行为。

对于需要处理 SQL 查询的多个结果行的情况，请参见第 43.6.6 节。

43.5.4. 执行动态命令

很多时候你将想要在 PL/pgSQL 函数中产生动态命令，也就是每次执行中会涉及到不同表或不同数据类型的命令。PL/pgSQL 通常对于命令所做的缓存计划尝试（如第 43.11.2 节中讨论）在这种情境下无法工作。要处理这一类问题，提供了EXECUTE 语句：

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

其中 *command-string* 是一个表达式，其求值结果（类型为 text）是包含待执行命令的字符串。可选的 *target* 是一个记录变量、一个行变量或者一个逗号分隔的简单变量以及记录/行字段的列表，该命令的结果将存储在其中。可选的USING 表达式提供要被插入到该命令中的值。

在计算得到的命令字符串中，不会做PL/pgSQL 变量的替换。任何所需的变量值必须在命令字符串被构造时被插入其中，或者你可以使用下面描述的参数。

此外，通过 EXECUTE 执行的命令不会缓存计划，而是在每次运行该语句时重新规划。因此，可以在函数中动态构造命令字符串，对不同的表和列执行操作。

INTO 子句指定一个返回行的 SQL 命令的结果应该被赋值到哪里。如果提供了一个行变量或变量列表，它必须完全匹配查询命令结果的结构（当一个记录变量被提供时，它会自动把它自己配置为匹配结果结构）。如果返回多个行，只有第一个行会被赋值给INTO 变量。如果没有返回行，NULL 会被赋值给INTO 变量。如果没有指定 INTO 子句，该查询结果会被抛弃。

如果给出了 STRICT 选项，除非该查询刚好产生一行，否则将会报告一个错误。

命令字符串可以使用参数值，它们在命令中用$1、$2 等引用。这些符号引用在 USING 子句中提供的值。这种方法通常比把数据值作为文本插入命令字符串更可取：它避免了将该值转换为文本以及转换回来的运行时负荷，并且它更不容易被 SQL 注入攻击，因为不需要引用或转义。一个示例是：

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

注意，参数符号只能用于数据值 – 如果要使用动态确定的表名或列名，必须将其作为文本插入命令字符串。例如，如果前面的查询需要针对动态选择的表执行，可以这样写：

```
EXECUTE 'SELECT count(*) FROM '
    || quote_ident(tabname)
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

更清晰的方式是使用 `format()` 的 `%I` 格式说明来处理表名或列名（以换行分隔的字符串会被连接起来）：

```
EXECUTE format('SELECT count(*) FROM %I '
   'WHERE inserted_by = $1 AND inserted <= $2', tabname)
   INTO c
   USING checked_user, checked_date;
```

参数符号的另一个限制是，它们只能用于`SELECT`，`INSERT`，`UPDATE` 以及 `DELETE` 命令。在其他语句类型（统称为工具语句）中，即使只是数据值，也必须以文本形式插入。

在上面第一个示例中，带有一个简单的常量命令字符串和一些`USING` 参数的 `EXECUTE` 命令在功能上等效于直接用 PL/pgSQL 写的命令，并且允许自动发生PL/pgSQL 变量替换。重要的不同之处在于，`EXECUTE` 会在每一次执行时根据当前的参数值重新计划该命令，而PL/pgSQL 则可能创建一个通用计划并且将其缓存以便重用。在最佳计划强依赖于参数值的情况中，使用`EXECUTE` 来明确地保证不会选择一个通用计划是很有帮助的。

`EXECUTE` 目前不支持 `SELECT INTO`。但是可以执行一个纯的 `SELECT` 命令并且指定 `INTO` 作为 `EXECUTE` 本身的一部分。

> **注意**
>
> PL/pgSQL 中的 `EXECUTE` 语句与 EXECUTE 这一由 PostgreSQL 服务器支持的 SQL 语句无关。服务器的 `EXECUTE` 语句不能直接在 PL/pgSQL 函数中使用（并且也没有必要）。

例 43.1. 在动态查询中为值加引号

在使用动态命令时，经常需要处理单引号的转义。我们推荐在函数体中使用美元引用来引用固定文本。（如果你有未使用美元引用的旧代码，请参阅第 43.12.1 节中的概述；在把这类代码转换成更合理的写法时，它会帮你省下一些工夫。）

动态值需要被小心地处理，因为它们可能包含引号字符。一个使用 format() 的示例（这里假定你对函数体使用了美元引用，因此不需要双写引号）：

```
EXECUTE format('UPDATE tbl SET %I = $1 '
   'WHERE key = $2', colname) USING newvalue, keyvalue;
```

还可以直接调用引用函数：

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_literal(newvalue)
        || ' WHERE key = '
        || quote_literal(keyvalue);
```

这个示例展示了 quote_ident 和 quote_literal 函数的用法（见第 9.4 节）。为了安全起见，在插入动态查询之前，包含列名或表名标识符的表达式应先传给quote_ident。在构造出的命令中应作为字符串字面量出现的值，则应传给quote_literal。这两个函数都会采取适当措施，分别返回用双引号或单引号括起来的输入文本，并正确转义其中嵌入的特殊字符。

由于 quote_literal 被标记为 STRICT，因此用 null 参数调用时它总会返回 null。在上面的示例中，如果newvalue 或 keyvalue 为 null，整个动态查询字符串都会变成 null，进而导致EXECUTE 报错。可以通过使用 quote_nullable 函数来避免这个问题；它与 quote_literal 的工作方式相同，只是在用 null 参数调用时会返回字符串NULL。例如：

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_nullable(newvalue)
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

如果正在处理的参数值可能为空值，那么通常应该用quote_nullable 来代替 quote_literal。

通常，必须小心地确保查询中的空值不会产生意料之外的结果。例如如果keyvalue 为空值，下面的WHERE 子句

```
'WHERE key = ' || quote_nullable(keyvalue)
```

永远不会成功，因为在=操作符中使用空值操作数得到的结果总是空值。如果想让空值像普通键值一样工作，你应该将上面的命令重写成

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

（目前，IS NOT DISTINCT FROM 的处理效率不如=，因此只有在非常必要时才这样做。关于空值和IS DISTINCT 的详细信息请见第 9.2 节）。

请注意美元引用只对引用固定文本有用。尝试写出下面这个示例是一个非常糟糕的主意：

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = $$'
        || newvalue
        || '$$ WHERE key = '
        || quote_literal(keyvalue);
```

因为如果 newvalue 的内容碰巧含有$$，那么这段代码就会出问题。同样的问题也适用于你选择的任何其他美元引用定界符。因此，要想安全地引用事先不知道的文本，必须恰当地使用 quote_literal、quote_nullable 或 quote_ident。

动态 SQL 语句也可以使用 format（见第 9.4.1 节）函数来安全地构造。例如：

```
EXECUTE format('UPDATE tbl SET %I = %L '
   'WHERE key = %L', colname, newvalue, keyvalue);
```

%I 等效于 quote_ident 并且 %L 等效于 quote_nullable。format 函数可以和 USING 子句一起使用：

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
   USING newvalue, keyvalue;
```

这种形式更好，因为变量被以它们天然的数据类型格式处理，而不是无条件地把它们转换成文本并且通过%L 引用它们。这也效率更高。

动态命令和 EXECUTE 的一个更大的示例可以在例 43.10 中找到，它会构建并且执行一个 CREATE FUNCTION 命令来定义一个新的函数。

43.5.5. 获取结果状态

有好几种方法可以判断一条命令的效果。第一种方法是使用 GET DIAGNOSTICS 命令，其形式如下：

```
GET [ CURRENT ] DIAGNOSTICS variable { = | := } item [ , ... ];
```

这条命令允许检索系统状态指示符。CURRENT 是一个噪声词（另见第 43.6.8.1 节中的 GET STACKED DIAGNOSTICS）。每个 *item* 是一个关键字，它标识一个要被赋给指定*variable* 的状态值（变量应具有正确的数据类型来接收状态值）。表 43.1 中展示了当前可用的状态项。冒号等号（:=）可以被用来取代 SQL 标准的=符号。例如：

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

表 43.1. 可用的诊断项

名称	类型	描述
ROW_COUNT	bigint	最近的SQL命令处理的行数

名称	类型	描述
RESULT_OID	oid	最近一条SQL命令所插入的最后一行的 OID（仅在对具有 OID 的表执行INSERT 命令后有用）
PG_CONTEXT	text	描述当前调用栈的文本行（见第 43.6.9 节）

第二种确定命令效果的方法是检查名为 FOUND 的特殊变量，类型为boolean。在每次 PL/pgSQL 函数调用中，FOUND 的初始值都是 false。它由以下类型的语句设置：

- SELECT INTO 语句在为目标赋上一行值时将 FOUND 设置为 true，如果没有返回行则设置为 false。
- PERFORM 语句在生成（和丢弃）一个或多个行时将 FOUND 设置为 true，如果没有生成行则设置为 false。
- UPDATE、INSERT 和 DELETE 语句在至少影响一行时将 FOUND 设置为真，如果没有影响行则设置为假。
- FETCH 语句在返回行时将 FOUND 设置为 true，如果没有返回行则设置为 false。
- MOVE 语句在成功重新定位游标时将 FOUND 设置为 true，否则设置为 false。
- FOR 或 FOREACH 语句在迭代一次或多次时将 FOUND 设置为 true，否则设置为 false。当循环退出时，FOUND 会按上述方式设置；在循环执行过程中，FOUND 不会被循环语句修改，尽管它可能会被循环体内的其他语句执行修改。
- RETURN QUERY 和 RETURN QUERY EXECUTE 语句在查询返回至少一行时将 FOUND 设置为 true，如果没有返回行则设置为 false。

其他 PL/pgSQL 语句不会改变 FOUND 的状态。特别注意，EXECUTE 会改变 GET DIAGNOSTICS 的输出，但不会改变FOUND。

FOUND 是每个 PL/pgSQL 函数的局部变量；任何对它的修改只影响当前的函数。

43.5.6. 什么也不做

有时一个什么也不做的占位语句也很有用。例如，它能够指示 if/then/else 链中故意留出的空分支。可以使用NULL 语句达到这个目的：

```
NULL;
```

例如，下面的两段代码是等价的：

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- 忽略错误
END;
```

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN  -- 忽略错误
END;
```

究竟使用哪一种取决于各人的喜好。

> **注意**
>
> 在 Oracle 的 PL/SQL 中，不允许出现空语句列表，并且因此在这种情况下必须使用 NULL 语句。而 PL/pgSQL 允许你什么也不写。

43.6. 控制结构

控制结构可能是 PL/pgSQL 中最有用的（以及最重要）的部分了。利用 PL/pgSQL 的控制结构，你可以以非常灵活而且强大的方法操纵PostgreSQL 的数据。

43.6.1. 从函数返回

有两个命令让我们能够从函数中返回数据：RETURN 和 RETURN NEXT。

43.6.1.1. RETURN

```
RETURN expression;
```

带有一个表达式的 RETURN 用于终止函数并把 *expression* 的值返回给调用者。这种形式被用于不返回集合的 PL/pgSQL 函数。

如果函数返回的是标量类型，表达式结果会按照赋值部分的说明自动转换为函数的返回类型。但如果要返回一个复合（行）值，你必须写出一个恰好生成所需列集合的表达式。这可能需要显式类型转换。

如果你声明带输出参数的函数，那么就只需要写不带表达式的RETURN。输出参数变量的当前值将被返回。

如果你声明函数返回 void，一个RETURN 语句可以被用来提前退出函数；但是不要在 RETURN 后面写一个表达式。

一个函数的返回值不能是未定义。如果控制到达了函数最顶层块的末尾而没有碰到一个 RETURN 语句，那么会发生一个运行时错误。不过，这个限制不适用于带输出参数的函数以及返回void 的函数。在这些情况中，如果顶层的块结束，将自动执行一个RETURN 语句。

一些示例：

```
-- 返回一个标量类型的函数
RETURN 1 + 2;
RETURN scalar_var;

-- 返回一个复合类型的函数
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- 必须把列类型转换成正确的类型
```

43.6.1.2. RETURN NEXT 和 RETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];
```

当 PL/pgSQL 函数被声明为返回 SETOF *sometype* 时，返回过程会略有不同。在这种情况下，要返回的各个项通过一系列 RETURN NEXT 或 RETURN QUERY 命令指定，最后再用一个不带参数的 RETURN 命令表明函数已经执行完毕。RETURN NEXT 可用于标量和复合数据类型；对于复合结果类型，会返回完整的结果“表”。RETURN QUERY 会把查询执行结果追加到函数的结果集中。在同一个集合返回函数中，RETURN NEXT 和 RETURN QUERY 可以自由混用，此时它们的结果会被串接起来。

RETURN NEXT 和 RETURN QUERY 实际上不会从函数中返回 – 它们简单地向函数的结果集中追加零或多行。然后会继续执行 PL/pgSQL 函数中的下一条语句。随着后继的 RETURN NEXT 和 RETURN QUERY 命令的执行，结果集就建立起来了。最后一个RETURN（应该没有参数）会导致控制退出该函数（或者你可以让控制到达函数的结尾）。

RETURN QUERY 有一种变体 RETURN QUERY EXECUTE，它可以动态指定要被执行的查询。可以通过USING 向计算出的查询字符串插入参数表达式，这和在EXECUTE 命令中的方式相同。

如果你声明函数带有输出参数，只需要写不带表达式的RETURN NEXT。在每一次执行时，输出参数变量的当前值将被保存下来用于最终返回为结果的一行。注意为了创建一个带有输出参数的集合返回函数，在有多个输出参数时，你必须声明函数为返回SETOF record；或者如果只有一个类型为 *sometype* 的输出参数时，声明函数为SETOF *sometype*。

下面是一个使用 RETURN NEXT 的函数示例：

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- 这里可以做一些处理
        RETURN NEXT r; -- 返回 SELECT 的当前行
    END LOOP;
    RETURN;
END;
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

这里是一个使用 RETURN QUERY 的函数的示例：

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);

    -- 因为执行还未结束，我们可以检查是否有行被返回
    -- 如果没有就抛出异常。
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No flight at %.', $1;
    END IF;

    RETURN;
 END;
$BODY$
LANGUAGE plpgsql;

-- 返回可用的航班或者在没有可用航班时抛出异常。
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

> **注意**
>
> 如上所述，目前RETURN NEXT 和 RETURN QUERY 的实现在从函数返回之前会把整个结果集都保存起来。这意味着如果一个 PL/pgSQL 函数生成一个非常大的结果集，性能可

能会很差：数据将被写到磁盘上以避免内存耗尽，但是函数本身在整个结果集都生成之前不会退出。将来的PL/pgSQL 版本可能会允许用户定义没有这种限制的集合返回函数。目前，数据开始被写入到磁盘的时机由配置变量work_mem 控制。拥有足够内存来存储大型结果集的管理员可以考虑增大这个参数。

43.6.2. 从过程返回

过程没有返回值。因此，过程的结束可以不用RETURN 语句。如果想用一个 RETURN 语句提前退出代码，只需写一个没有表达式的RETURN。

如果过程有输出参数，那么输出参数最终的值会被返回给调用者。

43.6.3. 调用过程

一个 PL/pgSQL 函数、过程或DO 块可以调用过程，使用的命令是CALL。输出参数的处理方式与 CALL 在普通 SQL 中的行为不同。过程的每个 INOUT 参数都必须对应于 CALL 语句中的一个变量，过程返回后，其返回的值会被赋回该变量。例如：

```
CREATE PROCEDURE triple(INOUT x int)
LANGUAGE plpgsql
AS $$
BEGIN
    x := x * 3;
END;
$$;

DO $$
DECLARE myvar int := 5;
BEGIN
  CALL triple(myvar);
  RAISE NOTICE 'myvar = %', myvar;  -- prints 15
END;
$$;
```

43.6.4. 条件语句

IF 和 CASE 语句让你可以根据条件执行不同的命令。PL/pgSQL 有三种形式的 IF：

- IF ... THEN ... END IF
- IF ... THEN ... ELSE ... END IF
- IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

以及两种形式的 CASE：

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

43.6.4.1. IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

IF-THEN 语句是 IF 的最简单形式。如果条件为真，在THEN 和 END IF 之间的语句将被执行。否则，将忽略它们。

示例：

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

43.6.4.2. IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

IF-THEN-ELSE 语句对 IF-THEN 进行了增加，它让你能够指定一组在条件不为真时应该被执行的语句（注意这也包括条件为 NULL 的情况）。

示例：

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

43.6.4.3. IF-THEN-ELSIF

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...
]
]
[ ELSE
    statements ]
END IF;
```

有时会有多于两种选择。IF-THEN-ELSIF 则提供了一个简便的方法来检查多个条件。IF 条件会被一个接一个测试，直到找到第一个为真的。然后执行相关语句，然后控制会被交给END IF 之后的下一个语句（后续的任何 IF 条件不会被测试）。如果没有一个 IF 条件为真，那么ELSE 块（如果有）将被执行。

这里有一个示例：

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- 嗯，唯一的其他可能性是 number 为 null
    result := 'NULL';
END IF;
```

关键词 ELSIF 也可以被拼写成 ELSEIF。

另一个可以完成相同任务的方法是嵌套 IF-THEN-ELSE 语句，如下例：

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

不过，这种方法需要为每个IF 都写一个匹配的 END IF，因此当有很多选择时，这种方法比使用 ELSIF 要麻烦得多。

43.6.4.4. 简单 CASE

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

CASE 的简单形式提供了基于操作数等值判断的条件执行。*search-expression* 会被计算一次，然后依次与各个 WHEN 子句中的 *expression* 比较。如果找到匹配，就执行相应的 *statements*，随后控制转到 END CASE 之后的下一条语句（后续的 WHEN 表达式不会再被计算）。如果没有找到匹配，则执行 ELSE *statements*；但如果没有 ELSE，则会抛出 CASE_NOT_FOUND 异常。

这里是一个简单的示例：

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

43.6.4.5. 搜索式 CASE

```
CASE
    WHEN boolean-expression THEN
      statements
  [ WHEN boolean-expression THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

CASE 的搜索式形式根据布尔表达式的真假进行条件执行。每个 WHEN 子句的 *boolean-expression* 都会依次求值，直到找到一个结果为true 的表达式。然后执行相应的 *statements*，控制接着转到END CASE 之后的下一条语句。（后续的 WHEN 表达式不会再被求值。）如果没有找到为真的结果，就执行ELSE *statements*；但如果不存在 ELSE，则会抛出 CASE_NOT_FOUND 异常。

这里是一个示例：

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;
```

这种形式的 CASE 与 IF-THEN-ELSIF 完全等价，唯一的区别是：如果执行到被省略的ELSE 子句，就会报错，而不是简单地什么也不做。

43.6.5. 简单循环

使用 LOOP、EXIT、CONTINUE、WHILE、FOR 和 FOREACH 语句，你可以安排PL/pgSQL 函数重复一系列命令。

43.6.5.1. LOOP

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

LOOP 定义一个无条件的循环，它会无限重复直到被EXIT 或 RETURN 语句终止。可选的 *label* 可以被 EXIT 和 CONTINUE 语句用在嵌套循环中指定这些语句引用的是哪一层循环。

43.6.5.2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

如果没有给出 *label*，那么最内层的循环会被终止，然后跟在END LOOP 后面的语句会被执行。如果给出了 *label*，那么它必须是当前或者更高层的嵌套循环或者语句块的标签。然后该命名循环或块就会被终止，并且控制会转移到该循环/块相应的END 之后的语句上。

如果指定了 WHEN，只有*boolean-expression* 为真时才会发生循环退出。否则，控制会转移到EXIT 之后的语句。

EXIT 可以被用在所有类型的循环中，它并不限于在无条件循环中使用。

与 BEGIN 块一起使用时，EXIT 会把控制转交给该块结束后的下一条语句。需要注意的是，为此必须使用标签；未加标签的 EXIT 永远不会被视为匹配某个 BEGIN 块。这与 PostgreSQL 8.4 之前的版本不同，旧版本允许未加标签的 EXIT 匹配 BEGIN 块。

示例：

```
LOOP
    -- 一些计算
    IF count > 0 THEN
        EXIT;  -- 退出循环
    END IF;
END LOOP;

LOOP
    -- 一些计算
    EXIT WHEN count > 0;  -- 和前一个示例相同的结果
END LOOP;

<<ablock>>
BEGIN
    -- 一些计算
    IF stocks > 100000 THEN
        EXIT ablock;  -- 导致从 BEGIN 块中退出
    END IF;
    -- 当stocks > 100000时，这里的计算将被跳过
END;
```

43.6.5.3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

如果没有给出 *label*，最内层循环的下一次迭代会开始。也就是，循环体中剩余的所有语句将被跳过，并且控制会返回到循环控制表达式（如果有）来决定是否需要另一次循环迭代。如果 *label* 存在，它指定应该继续执行的循环的标签。

如果指定了 WHEN，该循环的下一次迭代只有在*boolean-expression* 为真时才会开始。否则，控制会传递给CONTINUE 后面的语句。

CONTINUE 可以被用在所有类型的循环中，它并不限于在无条件循环中使用。

示例：

```
LOOP
    -- 一些计算
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- 一些用于 count IN [50 .. 100] 的计算
END LOOP;
```

43.6.5.4. WHILE

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

只要 *boolean-expression* 被计算为真，WHILE 语句就会重复一个语句序列。在每次进入到循环体之前都会检查该表达式。

例如：

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- 这里是一些计算
END LOOP;

WHILE NOT done LOOP
    -- 这里是一些计算
END LOOP;
```

43.6.5.5. FOR（整型变体）

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

这种形式的 FOR 会创建一个在一个整数范围上迭代的循环。变量 *name* 会自动定义为类型 integer 并且只在循环内存在（任何该变量名的现有定义在此循环内都将被忽略）。给出范围上下界的两个表达式在进入循环的时候计算一次。如果没有指定 BY 子句，迭代步长为 1，否则步长是BY 中指定的值，该值也只在循环进入时计算一次。如果指定了REVERSE，那么在每次迭代后会减去步长，而不是加上步长。

整数 FOR 循环的一些示例：

```
FOR i IN 1..10 LOOP
    -- i 在循环中将取值 1,2,3,4,5,6,7,8,9,10
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- i 在循环中将取值 10,9,8,7,6,5,4,3,2,1
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i 在循环中将取值 10,8,6,4,2
END LOOP;
```

如果下界大于上界（或者在 REVERSE 情况下是小于），循环体根本不会被执行。而且不会抛出任何错误。

如果一个 *label* 被附加到 FOR 循环，那么整数循环变量可以用一个使用那个*label* 的限定名引用。

43.6.6. 遍历查询结果

使用一种不同类型的 FOR 循环，你可以通过一个查询的结果进行迭代并且操纵相应的数据。语法是：

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

target 可以是记录变量、行变量，或者由标量变量组成的逗号分隔列表。*query* 产生的每一行都会依次赋给 *target*，并为每一行执行一次循环体。下面是一个示例：

```
CREATE FUNCTION refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing all materialized views...';

    FOR mviews IN
       SELECT n.nspname AS mv_schema,
              c.relname AS mv_name,
              pg_catalog.pg_get_userbyid(c.relowner) AS owner
         FROM pg_catalog.pg_class c
    LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
        WHERE c.relkind = 'm'
     ORDER BY 1
    LOOP

        -- Now "mviews" has one record with information about the materialized view

        RAISE NOTICE 'Refreshing materialized view %.% (owner: %)...',
                     quote_ident(mviews.mv_schema),
                     quote_ident(mviews.mv_name),
                     quote_ident(mviews.owner);
        EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I', mviews.mv_schema, mviews.mv_name);
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

如果循环被一个 EXIT 语句终止，那么在循环之后你仍然可以访问最后被赋予的行值。

在这类 FOR 语句中使用的 *query*，可以是任何向调用者返回行的 SQL 命令：最常见的是 SELECT，但也可以是带 RETURNING 子句的 INSERT、UPDATE 或 DELETE。某些工具命令，如 EXPLAIN，也可以用于此处。

PL/pgSQL 变量会被替换到查询文本中，并且如第 43.11.1 节和第 43.11.2 节中详细讨论的，查询计划会被缓存以用于可能的重用。

FOR-IN-EXECUTE 语句是在行上迭代的另一种方式：

```
[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
    statements
END LOOP [ label ];
```

这种形式与前面的形式类似，只不过源查询被指定为一个字符串表达式，在每次进入FOR 循环时都会计算它并且重新计划。这允许程序员在一个预先计划好了的命令的速度和一个动态命令的灵活性之间进行选择，就像一个纯EXECUTE 语句那样。与 EXECUTE 一样，可以通过USING 将参数值插入到动态命令中。

另一种指定要对其结果迭代的查询的方式是将它声明为一个游标。这会在第 43.7.4 节中描述。

43.6.7. 遍历数组

这种 FOREACH 循环与 FOR 循环很相似，但它遍历的是数组值中的元素，而不是 SQL 查询返回的行。（一般来说，FOREACH 旨在遍历复合值表达式的组成部分；未来可能会增加用于遍历数组以外复合值的变体。）用于遍历数组的 FOREACH 语句为：

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

如果没有写 SLICE，或者指定的是SLICE 0，循环就会遍历计算*expression* 所得数组的各个独立元素。*target* 变量会依次接收每个元素值，并对每个元素执行一次循环体。下面是一个遍历整数数组元素的示例：

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

元素会按存储顺序访问，而不管数组有多少维。尽管*target* 通常只是单个变量，但在遍历复合值（记录）数组时，它也可以是一个变量列表。在这种情况下，每个数组元素都会按复合值的连续列给这些变量赋值。

当 SLICE 为正值时，FOREACH 遍历的是数组切片，而不是单个元素。SLICE 值必须是一个不大于数组维数的整数常量。*target* 变量必须是数组，并且它会依次接收数组值的各个切片，其中每个切片都具有SLICE 指定的维数。下面是一个遍历一维切片的示例：

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

43.6.8. 捕获错误

默认情况下，PL/pgSQL 函数中发生的任何错误都会中止函数及其外围事务的执行。你可以使用带有 EXCEPTION 子句的 BEGIN 块来捕获错误并从中恢复。其语法是在普通 BEGIN 块语法上的扩展：

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
          handler_statements
      ... ]
END;
```

如果没有发生错误，这种形式的块只是简单地执行所有*statements*，并且接着控制转到END 之后的下一个语句。但是如果在 *statements* 内发生了一个错误，则会放弃对*statements* 的进一步处理，然后控制会转到EXCEPTION 列表。系统会在列表中寻找匹配所发生错误的第一个*condition*。如果找到一个匹配，则执行对应的*handler_statements*，并且接着把控制转到END 之后的下一个语句。如果没有找到匹配，该错误就会传播出去，就好像根本没有EXCEPTION 一样：错误可以被一个带有 EXCEPTION 的外围块捕捉，如果没有这样的块则中止该函数的处理。

condition 的名字可以是附录 A 中显示的任何名字。一个分类名匹配其中所有的错误。特殊的条件名 OTHERS 匹配除了 QUERY_CANCELED 和 ASSERT_FAILURE 之外的所有错误类型（虽然通常并不明智，还是可以用名字捕获这两种错误类型）。条件名是大小写无关的。一个错误条件也可以通过SQLSTATE 代码指定，例如以下是等价的：

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

如果在选中的 *handler_statements* 内发生了新的错误，那么它不能被这个EXCEPTION 子句捕获，而是被传播出去。一个外层的EXCEPTION 子句可以捕获它。

当一个错误被 EXCEPTION 捕获时，PL/pgSQL 函数的局部变量会保持错误发生时的值，但是该块中所有对持久数据库状态的改变都会被回滚。例如，考虑这个片段：

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

当控制到达对 y 赋值的地方时，它会带着一个division_by_zero 错误失败。这个错误将被 EXCEPTION 子句捕获。而在 RETURN 语句中返回的值将是 x 增加过后的值。但是 UPDATE 命令的效果将已经被回滚。不过，在该块之前的INSERT 将不会被回滚，因此最终的结果是数据库包含Tom Jones 但不包含 Joe Jones。

> **提示**
>
> 进入和退出一个包含 EXCEPTION 子句的块要比不包含该子句的块开销大得多。因此，只在必要的时候使用EXCEPTION。

例 43.2. UPDATE/INSERT 的异常

这个示例使用异常处理来酌情执行 UPDATE 或 INSERT。我们推荐应用使用带有 ON CONFLICT DO UPDATE 的 INSERT 而不是真正使用这种模式。下面的示例主要是为了展示 PL/pgSQL 如何控制流程：

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- 先尝试更新该键
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- 该键不存在，因此尝试插入
        -- 如果其他某人并发地插入同一个键，
        -- 就可能发生违反唯一约束的错误
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- 什么也不做，并且循环再次尝试 UPDATE
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

这段代码假定 unique_violation 错误是 INSERT 造成，并且不是由该表上一个触发器函数中的INSERT 导致。如果在该表上有多于一个唯一索引，也可能会发生不正确的行为，因为不管哪个索引导致该错误它都将重试该操作。通过接下来要讨论的特性来检查被捕获的错误是否为所预期的会更安全。

43.6.8.1. 获取错误信息

异常处理器经常需要识别所发生的具体错误。有两种方法可以获取 PL/pgSQL 中当前异常的信息：特殊变量和 GET STACKED DIAGNOSTICS 命令。

在一个异常处理器内，特殊变量SQLSTATE 包含了对应于被抛出异常的错误代码（可能的错误代码列表见表 A.1）。特殊变量 SQLERRM 包含与该异常相关的错误消息。这些变量在异常处理器外是未定义的。

在一个异常处理器内，我们也可以用GET STACKED DIAGNOSTICS 命令检索有关当前异常的信息，该命令的形式为：

```
GET STACKED DIAGNOSTICS variable { = | := } item [ , ... ];
```

每个 *item* 是一个关键词，它标识一个被赋给指定*variable*（应该具有接收该值的正确数据类型）的状态值。表 43.2 中显示了当前可用的状态项。

表 43.2. 错误诊断项

名称	类型	描述
RETURNED_SQLSTATE	text	该异常的 SQLSTATE 错误代码
COLUMN_NAME	text	与异常相关的列名
CONSTRAINT_NAME	text	与异常相关的约束名
PG_DATATYPE_NAME	text	与异常相关的数据类型名
MESSAGE_TEXT	text	该异常的主要消息的文本
TABLE_NAME	text	与异常相关的表名
SCHEMA_NAME	text	与异常相关的模式名
PG_EXCEPTION_DETAIL	text	该异常的详细消息文本（如果有）
PG_EXCEPTION_HINT	text	该异常的提示消息文本（如果有）
PG_EXCEPTION_CONTEXT	text	描述产生异常时调用栈的文本行（见第 43.6.9 节）

如果异常没有为一个项设置值，将返回一个空字符串。

下面是一个示例：

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
BEGIN
  -- 某些可能导致异常的处理
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

43.6.9. 获得执行位置信息

GET DIAGNOSTICS（之前在第 43.5.5 节中描述）命令检索有关当前执行状态的信息（反之上文讨论的 GET STACKED DIAGNOSTICS 命令报告先前发生错误时的执行状态信息）。它的 PG_CONTEXT 状态项可用于标识当前执行位置。状态项 PG_CONTEXT 将返回一个文本字符串，其中包含一行或多行描述该调用栈的文本。第一行会指向当前函数以及当前正在执行的 GET DIAGNOSTICS 命令。第二行及其后的行表示调用栈中更上层的调用函数。例如：

```
CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
BEGIN
  RETURN inner_func();
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
  stack text;
BEGIN
  GET DIAGNOSTICS stack = PG_CONTEXT;
  RAISE NOTICE E'--- Call Stack ---\n%', stack;
  RETURN 1;
END;
$$ LANGUAGE plpgsql;

SELECT outer_func();

NOTICE:  --- Call Stack ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
CONTEXT:  PL/pgSQL function outer_func() line 3 at RETURN
 outer_func
 ------------
           1
(1 row)
```

GET STACKED DIAGNOSTICS ... PG_EXCEPTION_CONTEXT 返回同类的栈跟踪，但是它描述检测到错误的位置而不是当前位置。

43.7. 游标

和一次执行整个查询不同，可以建立一个游标来封装该查询，并且接着一次读取该查询结果的一些行。这样做的原因之一是在结果中包含大量行时避免内存不足（不过，PL/pgSQL 用户通常不需要担心这些，因为FOR 循环在内部会自动使用一个游标来避免内存问题）。一种更有趣的用法是返回一个函数已经创建的游标的引用，允许调用者读取行。这提供了一种有效的方法从函数中返回大型行集。

43.7.1. 声明游标变量

所有在 PL/pgSQL 中对游标的访问都会通过游标变量，它总是特殊的数据类型refcursor。创建游标变量的一种方法是把它声明为一个类型为 refcursor 的变量。另外一种方法是使用游标声明语法，通常是：

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

（为兼容 Oracle，可以用 IS 代替 FOR。）如果指定了 SCROLL，游标就支持向后滚动；如果指定了 NO SCROLL，向后提取会被拒绝；如果两者都未指定，是否允许向后提取则取决于查询本身。如果指定了 *arguments*，它就是一个由 *name datatype* 对组成的逗号分隔列表，这些名字会在给定查询中被参数值替换。实际替换这些名字的值会在打开游标时提供。

一些示例：

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

所有这三个变量都是 refcursor 类型，但是第一个可以用于任何查询，而第二个已经被绑定了一个完全指定的查询，并且最后一个被绑定了一个参数化查询。（游标被打开时，key 将被一个整数参数值替换）。变量 curs1 被称为未绑定，因为它没有被绑定到任何特定查询。

当游标查询使用 FOR UPDATE/SHARE 时，不能使用 SCROLL 选项。此外，对于涉及易失函数的查询，最好使用 NO SCROLL。SCROLL 的实现假定重新读取查询输出会得到一致的结果，而易失函数可能无法满足这一假设。

43.7.2. 打开游标

在能够使用游标检索行之前，必须先将其打开（这等效于 SQL 命令 DECLARE CURSOR）。PL/pgSQL 有三种形式的 OPEN 命令，其中两种用于未绑定游标变量，另一种用于已绑定游标变量。

> **注意**
>
> 可以通过第 43.7.4 节中描述的 FOR 语句在不显式打开游标的情况下使用已绑定的游标变量。

43.7.2.1. OPEN FOR *query*

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

该游标变量会被打开，并被赋予要执行的指定查询。该游标不能已经处于打开状态，并且它必须已被声明为未绑定游标变量（即，一个简单的refcursor 变量）。该查询必须是 SELECT，或其他会返回行的命令（例如EXPLAIN）。该查询会以与 PL/pgSQL 中其他 SQL 命令相同的方式处理：替换 PL/pgSQL 变量名，并缓存查询计划以备后续重用。当一个PL/pgSQL 变量被替换到游标查询中时，被替换的是它在OPEN 时刻所具有的值；之后对该变量的更改不会影响游标的行为。SCROLL 和 NO SCROLL 选项与已绑定游标中的含义相同。

一个示例：

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

43.7.2.2. OPEN FOR EXECUTE

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
                                        [ USING expression [, ... ] ];
```

该游标变量会被打开，并被赋予要执行的指定查询。该游标不能已经处于打开状态，并且必须已被声明为未绑定游标变量（即，一个简单的refcursor 变量）。该查询以字符串表达式的形式给出，这一点与EXECUTE 命令相同。像往常一样，这提供了灵活性，因此查询计划可以在不同执行之间变化（见第 43.11.2 节），同时也意味着不会在该命令字符串上执行变量替换。与EXECUTE 一样，可以通过format() 和 USING 把参数值插入动态命令中。SCROLL 和 NO SCROLL 选项与已绑定游标中的含义相同。

一个示例：

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USING keyvalue;
```

在这个示例中，表名通过format() 插入到查询中。col1 的比较值通过 USING 参数插入，因此无需加引号。

43.7.2.3. 打开已绑定游标

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

这种形式的 OPEN 用于打开一个在声明时就已经绑定查询的游标变量。该游标不能已经处于打开状态。当且仅当该游标被声明为接收参数时，才必须提供实际参数值表达式列表。这些值会被替换到查询中。

已绑定游标的查询计划始终被视为可缓存；在这种情况下没有与 EXECUTE 对应的形式。注意，不能在OPEN 中指定 SCROLL 和 NO SCROLL，因为游标的滚动行为已经确定。

使用位置或命名记法可以传递参数值。在位置记法中，所有参数都必须按照顺序指定。在命名记法中，每一个参数的名字使用:=与参数表达式分隔。类似于第 4.3 节中描述的调用函数，也允许混合位置和命名记法。

示例（这些示例使用上面示例中的游标声明）：

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

因为已绑定游标的查询会进行变量替换，实际上有两种方式把值传给游标：要么向OPEN 传入显式参数，要么在查询中隐式引用PL/pgSQL 变量。不过，只有在声明已绑定游标之前就已声明的变量才会被替换到查询中。在这两种情况下，要传递的值都在OPEN 时确定。例如，获得与上面 curs3 示例相同效果的另一种方式是

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

43.7.3. 使用游标

一旦一个游标已经被打开，那么就可以用这里描述的语句操作它。

这些操作不必发生在最初打开该游标的同一个函数中。你可以从函数中返回一个 refcursor 值，让调用者来操作该游标。（在内部，refcursor 值只是一个所谓 portal 的字符串名称，该 portal 包含了该游标活动查询的状态。这个名称可以被传递、赋给其他refcursor 变量等等，而不会干扰该 portal。）

所有 portal 都会在事务结束时被隐式关闭。因此，refcursor 值只能在事务结束之前用于引用一个打开的游标。

43.7.3.1. FETCH

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

FETCH 从游标中检索下一行到目标中，目标可以是一个行变量、记录变量或者逗号分隔的简单变量列表，就像SELECT INTO 一样。如果没有下一行，目标会被设置为 NULL。与SELECT INTO 一样，可以检查特殊变量FOUND 来看是否获得了一行。

direction 子句可以是 SQL FETCH 命令中允许的任何变体，除了那些能够取得多于一行的。即它可以是 NEXT、PRIOR、FIRST、LAST、ABSOLUTE *count*、RELATIVE *count*、FORWARD 或者 BACKWARD。省略 *direction* 和指定 NEXT 是一样的。在使用 *count* 的形式中，*count* 可以是任意的整数值表达式（与 SQL 命令 FETCH 不一样，后者仅允许整数常量）。除非游标被使用SCROLL 选项声明或打开，否则要求反向移动的*direction* 值很可能会失败。

cursor 必须是一个引用已打开游标 portal 的 refcursor 变量名。

示例：

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

43.7.3.2. MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

MOVE 重新定位一个游标而不检索任何数据。MOVE 的工作方式与 FETCH 完全一样，只是仅重新定位游标，而不返回移动到的行。与SELECT INTO 一样，可以检查特殊变量FOUND 来看是否存在可移动到的下一行。

示例：

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

43.7.3.3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

当游标定位在某个表行上时，可以使用该游标来标识该行，并对其执行更新或删除。游标查询的形式有一些限制（尤其不能包含分组），并且在这类场景中最好对游标使用 FOR UPDATE。详见 DECLARE 参考页。

一个示例：

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

43.7.3.4. CLOSE

```
CLOSE cursor;
```

CLOSE 关闭打开游标所基于的 portal。这样就可以在事务结束之前提前释放资源，或者释放该游标变量以便再次打开。

一个示例：

```
CLOSE curs1;
```

43.7.3.5. 返回游标

PL/pgSQL 函数可以向调用者返回游标。这对于返回多行或多列，特别是非常大的结果集时很有用。要做到这一点，函数需要打开游标，并把游标名返回给调用者（或者直接使用调用者指定或已知的 portal 名称来打开游标）。随后调用者就可以从该游标中提取行。游标既可以由调用者关闭，也会在事务结束时自动关闭。

游标使用的 portal 名称既可以由程序员指定，也可以自动生成。要指定 portal 名称，只需在打开refcursor 变量之前给它赋一个字符串值。OPEN 会把该 refcursor 变量的字符串值用作底层 portal 的名称。不过，如果refcursor 变量为 null，OPEN 就会自动生成一个与任何现有 portal 都不冲突的名称，并把它赋回给refcursor 变量。

> **注意**
>
> 已绑定游标变量会被初始化为表示其名称的字符串值，因此除非程序员在打开游标之前通过赋值覆盖它，否则 portal 名称与游标变量名相同。而未绑定游标变量在初始时默认为空值，因此除非被覆盖，否则它会得到一个自动生成的唯一名称。

下面的示例显示了一个调用者提供游标名字的方法：

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

下面的示例使用了自动游标名生成：

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

-- 需要在一个事务中使用游标。
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

下面的示例展示了从一个函数中返回多个游标的一种方法：

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- 需要在一个事务中使用游标。
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

43.7.4. 遍历游标结果

有一种 FOR 语句的变体，它允许通过游标返回的行进行迭代。语法是：

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ] LOOP
    statements
END LOOP [ label ];
```

该游标变量必须在声明时已经被绑定到某个查询，并且它不能已经被打开。FOR 语句会自动打开游标，并且在退出循环时自动关闭游标。当且仅当游标被声明要使用参数时，才必须出现一个实际参数值表达式的列表。这些值会被替换到查询中，采用OPEN 期间的方式（见第 43.7.2.3 节）。

变量 *recordvar* 会被自动定义为 record 类型，并且只存在于循环内部（循环中该变量名任何已有定义都会被忽略）。每一个由游标返回的行都会被陆续地赋值给这个记录变量并且执行循环体。

43.8. 事务管理

在由 CALL 命令调用的过程中，以及在匿名代码块（DO 命令）中，可以使用COMMIT 和 ROLLBACK 结束事务。事务通过这些命令结束后，会自动开始一个新事务，因此不存在单独的 START TRANSACTION 命令（注意 BEGIN 和 END 在 PL/pgSQL 中含义不同）。

这里是一个简单的示例：

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpgsql
AS $$
BEGIN
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END;
$$;

CALL transaction_test1();
```

只有在从顶层调用的 CALL 或 DO 中才能进行事务控制，在没有任何其他中间命令的嵌套CALL 或 DO 调用中也能进行事务控制。例如，如果调用栈是CALL proc1() → CALL proc2() → CALL proc3()，那么第二个和第三个过程可以执行事务控制动作。但是如果调用栈是 CALL proc1() → SELECT func2() → CALL proc3()，则最后一个过程不能做事务控制，因为中间有SELECT。

游标循环需要特别考虑。来看这个例子：

```
CREATE PROCEDURE transaction_test2()
LANGUAGE plpgsql
AS $$
DECLARE
    r RECORD;
BEGIN
    FOR r IN SELECT * FROM test2 ORDER BY x LOOP
        INSERT INTO test1 (a) VALUES (r.x);
        COMMIT;
    END LOOP;
END;
$$;

CALL transaction_test2();
```

通常，游标会在事务提交时自动关闭。但是，在这样的循环中创建的游标，会自动转换为可保持游标，转换发生于第一次COMMIT 或 ROLLBACK。这意味着，游标将在第一次执行以下命令时被完整求值：COMMIT 或 ROLLBACK，而不是逐行求值。循环结束后游标仍然会被自动移除，因此这些行为对用户大多是不可见的。

由非只读命令（例如 UPDATE ... RETURNING）驱动的游标循环中，不允许使用事务命令。

不能在带有异常处理器的块中结束事务。

43.9. 错误和消息

43.9.1. 报告错误和消息

使用 RAISE 语句报告消息和抛出错误。

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

其中，*level* 选项指定错误的严重程度。允许的级别为 DEBUG，LOG，INFO，NOTICE，WARNING 以及 EXCEPTION，其中EXCEPTION 是默认值。EXCEPTION 会抛出错误（通常会中止当前事务）；其他级别只会生成不同优先级的消息。特定优先级的消息是报告给客户端、写入服务器日志，还是两者都做，由log_min_messages 和 client_min_messages 配置变量控制。更多信息见第 19 章中的说明。

在 *level*（如果有）之后，可以指定一个*format* 字符串（必须是简单的字符串字面量，不能是表达式）。格式字符串指定要报告的错误消息文本。格式字符串之后可以跟上可选的参数表达式，其值将被插入消息中。在格式字符串内，% 会被替换为下一个可选参数值的字符串表示。写成%% 可以输出一个字面的%。参数个数必须与格式字符串中% 占位符的个数匹配，否则会在函数编译期间报错。

在这个示例中，v_job_id 的值将替换字符串中的%：

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

可以为错误报告附加额外信息，方法是写出USING，后面跟上*option* = *expression* 项目。每个 *expression* 都可以是任意字符串值表达式。允许的 *option* 关键字为：

MESSAGE

设置错误消息文本。该选项不能用于在 USING 之前包含格式字符串的 RAISE 形式。

DETAIL

提供错误的详细信息。

HINT

提供一个提示消息。

ERRCODE

指定要报告的错误代码（SQLSTATE），可以用附录 A 中所示的条件名，或者直接作为一个五字符 SQLSTATE 代码。

COLUMN
CONSTRAINT
DATATYPE
TABLE
SCHEMA

提供一个相关对象的名称。

这个例子会中止事务，并给出指定的错误消息和提示：

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
      USING HINT = 'Please check your user ID';
```

这两个示例展示了设置 SQLSTATE 的两种等价的方法：

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

还有第二种 `RAISE` 语法，其中主参数是要报告的条件名或 SQLSTATE，例如：

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

在这种语法中，`USING` 可以用来提供自定义的错误消息、细节或提示。另一种达到前面示例同样效果的方式是：

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

还有另一种变体是写 `RAISE USING` 或 `RAISE` *`level`* `USING`，并把其余内容都放在`USING` 列表里。

`RAISE` 的最后一种变体根本没有参数。这种形式只能被用在一个 `BEGIN` 块的 `EXCEPTION` 子句中，它导致当前正在被处理的错误被重新抛出。

> **注意**
>
> 在 PostgreSQL 9.1 之前，没有参数的`RAISE` 被解释为重新抛出来自包含活动异常处理器的块的错误。因此一个嵌套在那个处理器中的 `EXCEPTION` 子句无法捕捉它，即使`RAISE` 位于嵌套 `EXCEPTION` 子句的块中也是这样。这种行为很奇怪，也并不兼容 Oracle 的 PL/SQL。

如果在 `RAISE EXCEPTION` 命令中没有指定条件名称或 SQLSTATE，则默认使用 `raise_exception` (`P0001`)。如果没有指定消息文本，则默认使用条件名称或 SQLSTATE 作为消息文本。

> **注意**
>
> 当用 SQLSTATE 代码指定错误代码时，你并不受限于预定义错误代码，而是可以选择任何由五位数字和/或大写 ASCII 字母构成的错误代码，唯一不能使用的是 `00000`。我们建议尽量避免抛出以三个零结尾的错误代码，因为这些是类别代码，只能通过捕获整个类别来捕获这类错误。

43.9.2. 检查断言

ASSERT 语句是一种向 PL/pgSQL 函数中插入调试检查的方便方法。

```
ASSERT condition [ , message ];
```

condition 是一个布尔表达式，它被期望总是计算为真。如果确实如此，ASSERT 语句不会再做什么。但如果结果是假或者空值，那么将发生一个ASSERT_FAILURE 异常（如果在计算 *condition* 时发生错误，它会被报告为一个普通错误）。

如果提供了可选的 *message*，它是一个结果（如果不为 NULL）被用来替换默认错误消息文本 “assertion failed” 的表达式（如果 *condition* 失败）。*message* 表达式在断言成功的普通情况下不会被计算。

通过配置参数 plpgsql.check_asserts 可以启用或者禁用断言测试，这个参数接受布尔值且默认为on。如果这个参数为 off，则ASSERT 语句什么也不做。

注意 ASSERT 是为了检测程序的 bug，而不是报告普通的错误情况。如果要报告普通错误，请使用前面介绍的 RAISE 语句。

43.10. 触发器函数

PL/pgSQL 可以被用来在数据更改或者数据库事件上定义触发器函数。触发器函数用 CREATE FUNCTION 命令创建，它被声明为一个没有参数并且返回类型为trigger（对于数据更改触发器）或者 event_trigger（对于数据库事件触发器）的函数。名为 TG_*something* 的特殊局部变量将被自动创建用以描述触发该调用的条件。

43.10.1. 数据更改触发器

一个数据更改触发器被声明为一个没有参数并且返回类型为 trigger 的函数。注意，如下所述，即便该函数准备接收一些在CREATE TRIGGER 中指定的参数 – 这类参数通过 TG_ARGV 传递，也必须把它声明为没有参数。

当一个 PL/pgSQL 函数作为触发器被调用时，会在顶层块中自动创建一些特殊变量。它们是：

NEW

数据类型为 RECORD；该变量保存行级触发器中用于 INSERT/UPDATE 操作的新数据行。在语句级触发器和 DELETE 操作中，该变量为 null。

OLD

数据类型为 RECORD；该变量保存行级触发器中用于 UPDATE/DELETE 操作的旧数据行。在语句级触发器和 INSERT 操作中，该变量为 null。

TG_NAME

数据类型为 name；包含实际触发的触发器名称的变量。

TG_WHEN

数据类型为 text；根据触发器定义，其值为字符串 BEFORE、AFTER 或 INSTEAD OF。

TG_LEVEL

数据类型为 text；根据触发器定义，其值为字符串 ROW 或 STATEMENT。

TG_OP

数据类型为 text；表示触发器对应操作的字符串：INSERT、UPDATE、DELETE 或 TRUNCATE。

TG_RELID

数据类型为 oid；导致触发器调用的表的对象 ID。

TG_RELNAME

数据类型为 name；导致触发器调用的表名。该变量已弃用，未来版本可能移除；请改用 TG_TABLE_NAME。

TG_TABLE_NAME

数据类型为 name；导致触发器调用的表名。

TG_TABLE_SCHEMA

数据类型为 name；导致触发器调用的表所在模式名。

TG_NARGS

数据类型为 integer；CREATE TRIGGER 语句中传给触发器函数的参数个数。

TG_ARGV[]

数据类型为 text 数组；来自 CREATE TRIGGER 语句的参数。索引从 0 开始；非法索引（小于 0 或大于等于 tg_nargs）返回 null。

一个触发器函数必须返回 NULL 或者是一个与触发器为之引发的表结构完全相同的记录/行值。

BEFORE 行级触发器可以返回 null，以通知触发器管理器跳过该行后续的操作（也就是说，不再触发后续触发器，并且不会对该行执行INSERT/UPDATE/DELETE）。如果返回非 null 值，则操作会继续进行，并使用该行值。返回一个不同于原始NEW 的行值会改变即将插入或更新的行。因此，如果触发器函数希望触发动作正常成功而不修改行值，就必须返回NEW（或与之相等的值）。若要修改将被存储的行，可以直接替换NEW 中的单个值并返回修改后的 NEW，或者构造一个完整的新记录/行来返回。对于作用于DELETE 的 before 触发器，返回值本身没有直接效果，但必须为非 null 才能让触发器动作继续。注意，在DELETE 触发器中 NEW 为 null，因此通常没有理由返回它。在DELETE 触发器中，常见写法是返回OLD。

INSTEAD OF 触发器（总是行级触发器，并且只能用于视图）能够返回 null 来表示它们没有执行任何更新，并且对该行剩余的操作应被跳过（即后续的触发器不会被引发，并且该行不会被计入外围INSERT/UPDATE/DELETE 的行影响状态中）。否则应该返回一个非 null 值用以表示该触发器执行了所请求的操作。对于 INSERT 和 UPDATE 操作，返回值应该是NEW，触发器函数可能对它进行了修改来支持INSERT RETURNING 和 UPDATE RETURNING（这也将影响被传递给任何后续触发器的行值，或者被传递给带有ON CONFLICT DO UPDATE 的 INSERT 语句中一个特殊的 EXCLUDED 别名引用）。对于 DELETE 操作，返回值应该是OLD。

一个 AFTER 行级触发器或一个 BEFORE 或 AFTER 语句级触发器的返回值总是会被忽略，因此也可以返回 null。不过，任何这些类型的触发器可能仍会通过抛出一个错误来中止整个操作。

例 43.3 展示了 PL/pgSQL 中一个触发器函数的示例。

例 43.3. 一个 PL/pgSQL 触发器函数

这个示例触发器保证：任何时候一个行在表中被插入或更新时，当前用户名和时间也会被标记在该行中。并且它会检查给出了一个雇员的姓名以及薪水是一个正值。

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- 检查给出了 empname 以及 salary
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- 谁会倒贴钱为我们工作?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- 记住谁在什么时候改变了工资单
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

另一种记录对表的改变的方法涉及到创建一个新表来为每一个发生的插入、更新或删除保持一行。这种方法可以被认为是对一个表的改变的审计。例 43.4 展示了 PL/pgSQL 中一个审计触发器函数的示例。

例 43.4. 一个用于审计的 PL/pgSQL 触发器函数

这个示例触发器保证了在 `emp` 表上的任何插入、更新或删除一行的动作都被记录（即审计）在 `emp_audit` 表中。当前时间和用户名会被记录到行中，还有在其上执行的操作类型。

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- making use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), current_user, OLD.*;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), current_user, NEW.*;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), current_user, NEW.*;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE FUNCTION process_emp_audit();
```

前一个示例的一个变体使用视图把主表和审计表连接起来，以显示每个条目最后一次被修改的时间。这种方法仍然记录了表更改的完整审计轨迹，同时也提供了一个简化视图，只显示从审计轨迹中导出的每个条目的最后修改时间戳。例 43.5 展示了 PL/pgSQL 中一个定义在视图上的审计触发器示例。

例 43.5. 一个用于审计的 PL/pgSQL 视图触发器函数

这个示例在视图上使用了一个触发器让它变得可更新，并且确保视图中一行的任何插入、更新或删除被记录（即审计）在`emp_audit`表中。当前时间和用户名会被与执行的操作类型一起记录，并且该视图会显示每一行的最后修改时间。

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer,
    stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN
        --
        -- Perform the required operation on emp, and create a row in emp_audit
        -- to reflect the change made to emp.
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
            INSERT INTO emp_audit VALUES('D', current_user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', current_user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', current_user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE FUNCTION update_emp_view();
```

触发器的一种用途是维护另一个表的汇总表。得到的汇总结果可以在某些查询中替代原始表使用，而且通常能大幅缩短运行时间。这种技术在数据仓库中很常见，其中用于度量或观察数据的表（称为事实表）可能会非常大。例 43.6 展示了 PL/pgSQL 中一个为数据仓库事实表维护汇总表的触发器函数示例。

例 43.6. 一个用于维护汇总表的 PL/pgSQL 触发器函数

这里详述的模式有一部分是基于 Ralph Kimball 所作的数据仓库工具包中的 Grocery Store 示例。

```
--
-- 主表 - 时间维度和销售事实。
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- 汇总表 - 按时间汇总销售
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- 在 UPDATE、INSERT、DELETE 时修改汇总列的函数和触发器。
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

        -- 算出增量/减量数。
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- 禁止修改 time_key
            -- （可能不会太麻烦，因为大部分的更改是用 DELETE + INSERT 完成的）。
```

```
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                                    OLD.time_key, NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;

        -- 插入或更新带有新值的汇总行。
        <<insert_update>>
        LOOP
            UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                    units_sold = units_sold + delta_units_sold,
                    amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

            EXIT insert_update WHEN found;

            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );

                EXIT insert_update;

            EXCEPTION
                WHEN UNIQUE_VIOLATION THEN
                    -- 什么也不做
            END;
        END LOOP insert_update;

        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE FUNCTION maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
```

```
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

AFTER 触发器也可以利用过渡表来观察触发语句修改的整组行。CREATE TRIGGER 命令会为一个或两个过渡表分配名称，随后函数就可以像引用只读临时表那样引用它们。例 43.7 展示了一个示例。

例 43.7. 使用过渡表审计

这个示例产生和例 43.4 相同的结果，但并未使用一个为每一行都触发的触发器，而是在把相关信息收集到一个过渡表中之后用了一个只为每个语句引发一次的触发器。当调用语句修改了很多行时，这种方法可能显著快于行触发器方法。注意我们必须为每一种事件建立一个单独的触发器声明，因为每种情况的REFERENCING 子句必须不同。但是这并不能阻止我们使用单一的触发器函数（实际上，使用三个单独的函数可能会更好，因为可以避免在TG_OP 上的运行时测试）。

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create rows in emp_audit to reflect the operations performed on emp,
        -- making use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit
                SELECT 'D', now(), current_user, o.* FROM old_table o;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit
                SELECT 'U', now(), current_user, n.* FROM new_table n;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit
                SELECT 'I', now(), current_user, n.* FROM new_table n;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit_ins
    AFTER INSERT ON emp
    REFERENCING NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_upd
    AFTER UPDATE ON emp
    REFERENCING OLD TABLE AS old_table NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_del
    AFTER DELETE ON emp
    REFERENCING OLD TABLE AS old_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
```

43.10.2. 事件触发器

PL/pgSQL 可以被用来定义事件触发器。PostgreSQL 要求一个可以作为事件触发器调用的函数必须被声明为没有参数并且返回类型为 `event_trigger`。

当一个 PL/pgSQL 函数作为事件触发器被调用时，会在顶层块中自动创建一些特殊变量。它们是：

`TG_EVENT`

数据类型为 `text`；表示触发器被触发时对应事件的字符串。

`TG_TAG`

数据类型为 `text`；包含触发该触发器的命令标签的变量。

例 43.8 展示了 PL/pgSQL 中一个事件触发器函数的示例。

例 43.8. 一个 PL/pgSQL 事件触发器函数

这个示例触发器在受支持命令每一次被执行时会简单地抛出一个 NOTICE 消息。

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE FUNCTION snitch();
```

43.11. PL/pgSQL 内部机制

本节讨论一些实现细节，了解这些细节对 PL/pgSQL 用户通常很重要。

43.11.1. 变量替换

在 PL/pgSQL 函数中，SQL 语句和表达式可以引用函数的变量和参数。在内部，PL/pgSQL 会用查询参数替换这些引用。只有在语法上允许参数或列引用的位置才会进行参数替换。作为一个极端例子，来看下面这种不良编程风格：

```
INSERT INTO foo (foo) VALUES (foo);
```

第一次出现的 foo 在语法上必须是表名，所以不会被替换，即使函数有一个变量也叫foo。第二次出现的位置必须是该表的列名，所以也不会被替换。只有第三次出现的位置才可能是对函数变量的引用。

> **注意**
>
> PostgreSQL 9.0 以前的版本会尝试在这三个位置都进行变量替换，从而导致语法错误。

由于变量名在语法上与表列名没有区别，所以在同时引用表的语句中就可能产生歧义：某个给定名称到底是指表列，还是变量？把前面的示例改成下面这样：

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

这里，dest 和 src 必须是表名，col 也必须是 dest 的一列，但 foo 和 bar 既可能是该函数的变量，也可能是 src 的列。

默认情况下，如果 SQL 语句中的名称既可能指变量，也可能指表列，PL/pgSQL 就会报告错误。可以通过重命名变量或列、限定有歧义的引用，或者告诉PL/pgSQL 优先采用哪种解释来解决这类问题。

最简单的解决方案是重命名变量或列。一种常用的编码规则是为 PL/pgSQL 变量使用一种不同于列名的命名习惯。例如，如果你将函数变量统一地命名为v_*something*，而你的列名不会开始于v_，就不会发生冲突。

另外你可以限定有歧义的引用让它们变清晰。在上面的示例中，src.foo 将是对表列的一种无歧义的引用。要创建对一个变量的无歧义引用，在一个被标记的块中声明它并且使用块的标签（见第 43.2 节）。例如

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

这里 block.foo 表示变量，即使在src 中有一个列 foo。函数参数以及诸如 FOUND 的特殊变量，都能通过函数的名称被限定，因为它们被隐式地声明在一个带有该函数名称的外层块中。

有时候在一个大型的 PL/pgSQL 代码体中修复所有的有歧义引用是不现实的。在这种情况下，你可以指定PL/pgSQL 应该将有歧义的引用解析为变量（这与 PL/pgSQL 在 PostgreSQL 9.0 之前的行为兼容）或表列（这与某些其他系统兼容，例如Oracle）。

要在系统范围内改变这种行为，将配置参数plpgsql.variable_conflict 设置为 error、use_variable 或者 use_column（这里 error 是出厂设置）之一。这个参数会影响 PL/pgSQL 函数中语句的后续编译，但是不会影响在当前会话中已经编译过的语句。因为改变这个设置能够导致PL/pgSQL 函数中行为的意想不到的改变，所以只能由一个超级用户来更改它。

你也可以按函数单独设置这一行为，方法是在函数文本开头插入以下特殊命令之一：

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

这些命令只影响它们所属的函数，并且会覆盖plpgsql.variable_conflict 的设置。一个示例是：

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

在 UPDATE 命令中，curtime、comment 以及 id 将引用该函数的变量和参数，不管users 有没有这些名称的列。注意，我们不得不在WHERE 子句中对 users.id 的引用加以限定，以便让它引用表列。但是我们不需要对UPDATE 列表中作为目标的 comment 引用加以限定，因为语法上那必须是users 的一列。我们可以用下面的方式写一个相同的不依赖于variable_conflict 设置的函数：

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

传递给 EXECUTE 及其变体的命令字符串中，不会发生变量替换。如果你需要向这种命令中插入变化的值，应在构造字符串值时完成，或者像第 43.5.4 节所说明的那样使用 USING。

目前，变量替换只在SELECT、INSERT、UPDATE 和 DELETE 命令中生效，因为主 SQL 引擎只允许在这些命令中使用查询参数。若要在其他语句类型（统称为工具语句）中使用非常量名称或值，就必须把该工具语句构造为字符串，再用EXECUTE 执行。

43.11.2. 计划缓存

在函数第一次被调用时（每个会话中都会如此），PL/pgSQL 解释器会解析函数源文本，并生成一棵内部的二进制指令树。该指令树完整表示了PL/pgSQL 语句结构，但函数中使用的各个SQL 表达式和 SQL 命令并不会立即被分析。

当函数中的某个表达式或 SQL 命令第一次执行时，PL/pgSQL 解释器会使用 SPI 管理器的 SPI_prepare 函数，对该命令进行解析和分析，以创建预备语句。之后再次执行该表达式或命令时，就会重用该预备语句。因此，对于那些很少访问的条件分支，函数不会承担分析当前会话中从未执行到的命令的开销。缺点是，某个具体表达式或命令中的错误，只有在执行到函数的那一部分时才能被发现。（简单语法错误会在最初的解析阶段发现，但更深层的问题只有在执行时才会显现。）

PL/pgSQL（更准确地说，是 SPI 管理器）还可以尝试缓存与某个预备语句相关的执行计划。如果没有使用缓存计划，那么每次执行该语句时都会生成新的执行计划，而当前参数值（也就是 PL/pgSQL 变量值）可用于优化所选计划。如果该语句没有参数，或者会被执行很多次，SPI 管理器就会考虑创建一个不依赖具体参数值的通用计划，并将其缓存起来供重复使用。通常只有在执行计划对其中引用的 PL/pgSQL 变量值不太敏感时，才会这样做。如果执行计划对参数值非常敏感，那么每次重新生成计划总体上反而更划算。关于预备语句的行为，详见 PREPARE。

由于 PL/pgSQL 保存预备语句并且有时候以这种方式保存执行计划，直接出现在一个PL/pgSQL 函数中的 SQL 命令必须在每次执行时引用相同的表和列。也就是说，你不能在一个 SQL 命令中把一个参数用作表或列的名字。要绕过这种限制，你可以构建PL/pgSQL `EXECUTE` 使用的动态命令，但是会付出在每次执行时需要执行新解析分析以及构建新执行计划的代价。

记录变量的可变特性在这里还会带来另一个问题。当记录变量的字段被用于表达式或语句中时，这些字段的数据类型不能在函数的不同调用之间发生变化，因为每个表达式都会按照第一次执行到它时所看到的数据类型来分析。必要时，可以用`EXECUTE` 绕过这个问题。

如果同一个函数被用作多个表的触发器，PL/pgSQL 会针对每个这样的表独立地准备并缓存语句。也就是说，缓存是按“触发器函数 + 表”的组合建立的，而不是每个函数只有一个缓存。这缓解了数据类型变化带来的部分问题；例如，即使不同表中名为`key` 的列类型不同，一个触发器函数也仍然能够成功使用它。

同样，具有多态参数类型的函数也会为它们已经被调用的每一种实参类型组合都保留一个独立的缓存，这样数据类型差异不会导致意想不到的失败。

语句缓存有时可能在解释时间敏感的值时产生令人惊讶的效果。例如这两个函数做的事情就有区别：

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

以及：

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

在 `logfunc1` 中，PostgreSQL 的主解析器在分析 `INSERT` 时就知道字符串`'now'` 应该被解释为`timestamp`，因为`logtable` 的目标列是这种类型。因此，在`INSERT` 被分析时`'now'` 将被转换为一个 `timestamp` 常量，并且在该会话的生命周期内被用于所有对`logfunc1` 的调用。不用说，这不是程序员想要的。一个更好的主意是使用`now()` 或 `current_timestamp` 函数。

在 `logfunc2` 中，PostgreSQL 的主解析器不知道应该把`'now'` 解释成什么类型，因此返回一个`text` 类型的数据值，其中包含字符串`now`。在随后给局部变量 `curtime` 赋值时，PL/pgSQL 解释器通过调用用于该转换的 `textout` 和 `timestamp_in` 函数，把这个字符串转换为

timestamp 类型。因此，计算得到的时间戳会按程序员预期在每次执行时更新。虽然这恰好能得到预期结果，但效率并不高，因此使用now() 函数仍然是更好的主意。

43.12. PL/pgSQL 开发提示

在 PL/pgSQL 中进行开发的一种好方法，是用你喜欢的文本编辑器编写函数，并在另一个窗口里使用psql 载入和测试这些函数。如果你采用这种方式，最好使用CREATE OR REPLACE FUNCTION 来编写函数。这样只需重新载入该文件，就可以更新函数定义。例如：

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

在运行 psql 期间，你可以用下面的命令载入或者重载这样一个函数定义文件：

```
\i filename.sql
```

并且接着立即发出 SQL 命令来测试该函数。

另一种使用 PL/pgSQL 进行开发的好方法，是使用有助于过程语言开发的 GUI 数据库访问工具。这类工具的一个例子是pgAdmin，当然也有其他工具。它们通常提供一些方便的特性，例如转义单引号，以及让重新创建和调试函数变得更容易。

43.12.1. 引号的处理

一个 PL/pgSQL 函数的代码在一个 CREATE FUNCTION 中被指定为一个字符串字面量。如果你用通常的方式把该字符串写在单引号中间，那么该函数体中的任何单引号都必须被双写；同样任何反斜线也必须被双写（假定使用了转义字符串语法）。双写引号本身就很繁琐，并且在更复杂的情况中代码会变得完全无法理解，因为你很容易发现你需要半打或者更多相邻的引号。我们推荐你转而把函数体写成一个“美元引用”的字符串（见第 4.1.2.4 节）。在美元引用方法中，你从不需要双写任何引号。但是要注意为你需要的每一层嵌套选择一个不同的美元引用定界符。例如，你可能把CREATE FUNCTION 命令写成：

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
          ....
$PROC$ LANGUAGE plpgsql;
```

在这里面，你可以在 SQL 命令中为简单字符串使用引号并且用$$来界定被你组装成字符串的 SQL 命令片段。如果你需要引用包括$$的文本，你可以使用Q等等。

下表展示了在不使用美元引用时，编写引号必须采取的做法。在把旧式的非美元引用代码改写成更易理解的形式时，它可能会有所帮助。

1 个引号

用来开始和结束函数体，例如：

```
CREATE FUNCTION foo() RETURNS integer AS '
           ....
' LANGUAGE plpgsql;
```

在一个单引号引用的函数体中的任何位置，引号必须成对出现。

2 个引号

用于函数体内的字符串字面量，例如：

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

在美元引用方法中，你只需要写：

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

这恰好就是 PL/pgSQL 解析器在两种情况中会看到的。

4 个引号

当你在函数内的一个字符串常量中需要一个单引号时，例如：

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

实际会被追加到 `a_output` 的值将是： `AND name LIKE 'foobar' AND xyz`。

在美元引用方法中，你可以写：

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

要小心在这周围的任何美元引用定界符都不能是`$$`。

6 个引号

当在函数体内的一个字符串中的一个单引号与该字符串常量末尾相邻，例如：

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

被追加到 `a_output` 的值则将是： `AND name LIKE 'foobar'`。

在美元引用方法中，这会变成：

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 个引号

当字符串常量中需要两个单引号（这需要 8 个引号），而且它们紧邻该字符串常量的末尾（还需 2 个引号）时。通常只有在编写生成其他函数的函数时（如例 43.10 所示），才会需要这种写法。例如：

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

a_output 的值将是：

```
if v_... like ''...'' then return ''...''; end if;
```

在美元引用方法中，这会变成：

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

这里我们假定我们只需要把单引号放在 a_output 中，因为在使用前会对它再次加引号。

43.12.2. 额外的编译时检查

为了辅助用户在一些简单但常见的问题产生危害之前找到它们，PL/pgSQL 提供了额外的检查。当被启用时，根据配置，它们可以在一个函数的编译期间被用来发出 WARNING 或者 ERROR。一个已经收到了 WARNING 的函数可以被继续执行而不会产生进一步的消息，因此建议你在一个单独的开发环境中进行测试。

这些额外检查通过配置变量启用，其中plpgsql.extra_warnings 用于警告，plpgsql.extra_errors 用于错误。两者都可以设为逗号分隔的检查项列表、"none" 或"all"。默认值为"none"。当前可用的检查只有一项：

shadowed_variables

检查声明是否遮蔽了先前定义的变量。

下面的例子展示了此设置的效果：plpgsql.extra_warnings 设为 shadowed_variables：

```
SET plpgsql.extra_warnings TO 'shadowed_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$
DECLARE
f1 int;
BEGIN
RETURN f1;
END;
$$ LANGUAGE plpgsql;
WARNING:  variable "f1" shadows a previously defined variable
LINE 3: f1 int;
        ^
CREATE FUNCTION
```

43.13. 从 Oracle PL/SQL 移植

这一节解释了 PostgreSQL 的 PL/pgSQL 语言和 Oracle 的 PL/SQL 语言之间的差别，用以帮助那些从Oracle®向 PostgreSQL 移植应用的人。

PL/pgSQL 在许多方面都与 PL/SQL 类似。它是一种具有块结构的命令式语言，所有变量都必须声明。赋值、循环和条件语句也都很相似。在从PL/SQL 移植到 PL/pgSQL 时，应当记住以下主要差异：

- 如果 SQL 命令中使用的名字既可能是表的列名，也可能是对函数中变量的引用，PL/SQL 会将其当作列名。这对应于 PL/pgSQL 的 plpgsql.variable_conflict = use_column 行为，这不是默认行为，如第 43.11.1 节所述。通常最好一开始就避免这种歧义，但如果必须移植大量依赖这种行为的代码，设置variable_conflict 可能是最好的解决办法。
- 在 PostgreSQL 中，函数体必须写成字符串字面量。因此你需要使用美元引用或者转义函数体中的单引号（见第 43.12.1 节）。
- 数据类型名称往往需要改写。例如，在 Oracle 中字符串值通常被声明为varchar2 类型，这并不是 SQL 标准类型。在PostgreSQL 中，应改用varchar 或 text 类型。类似地，应将 number 替换为 numeric，或者在适当时替换为其他数值数据类型。
- 应该用模式把函数组织成不同的分组，而不是用包。
- 因为没有包，所以也没有包级变量。这一点有时会带来不便。你可以改为在临时表中保存会话级状态。
- 带有 REVERSE 的整数 FOR 循环的工作方式不同：PL/SQL 中是从第二个数向第一个数倒数，而PL/pgSQL 是从第一个数向第二个数倒数，因此在移植时需要交换循环边界。不幸的是这种不兼容性是不太可能改变的（见第 43.6.5.5 节）。

- 查询上的 FOR 循环（不是游标）的工作方式同样不同：目标变量必须已经被声明，而PL/SQL总是会隐式地声明它们。但是这样做的优点是在退出循环后，变量值仍然可以访问。
- 在使用游标变量方面，存在一些记法差异。

43.13.1. 移植示例

例 43.9 展示了如何从 PL/SQL 移植一个简单的函数到 PL/pgSQL 中。

例 43.9. 从 PL/SQL 移植一个简单的函数到 PL/pgSQL

这里有一个 Oracle PL/SQL 函数：

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar2,
                                                  v_version varchar2)
RETURN varchar2 IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

让我们过一遍这个函数并且看看与 PL/pgSQL 相比有什么样的不同：

- 类型名称 varchar2 被改成了 varchar 或者 text。在这一节的示例中，我们将使用 varchar，但如果不需要特定的字符串长度限制，text 常常是更好的选择。
- 在函数原型中（不是函数体中）的 RETURN 关键字在 PostgreSQL 中变成了 RETURNS。还有，IS 变成了 AS，并且你还需要增加一个LANGUAGE 子句，因为PL/pgSQL 并非唯一可用的函数语言。
- 在 PostgreSQL 中，函数体被认为是一个字符串字面量，所以你需要使用引号或者美元引用定界符包围它。这代替了 Oracle 方法中的用于终止的/。
- 在 PostgreSQL 中没有 show errors 命令，并且也不需要这个命令，因为错误是自动报告的。

这个函数被移植到 PostgreSQL 后看起来会是这样：

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

例 43.10 展示了如何移植一个会创建另一个函数的函数，以及如何处理引号问题。

例 43.10. 从 PL/SQL 移植一个创建另一个函数的函数到 PL/pgSQL

下面的过程从 `SELECT` 语句读取行，并将结果写入 `IF` 语句，从而构造一个大型函数，以提高效率。

这是 Oracle 版本：

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR2,
                 v_domain IN VARCHAR2, v_url IN VARCHAR2) RETURN VARCHAR2 IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ''' || referrer_key.key_string
          || ''' THEN RETURN ''' || referrer_key.referrer_type
          || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

下面是这个函数的最终移植结果，目标数据库为PostgreSQL：

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc() AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
          || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
          || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                        v_domain varchar,
                                                        v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE plpgsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

注意，这里单独构造函数体，然后将其传给quote_literal，使其中的每个引号都变成两个。这种技术是必需的，因为不能安全地使用美元引用来定义新函数：我们无法确定会插入什么字符串，其来源是referrer_key.key_string 字段。（这里假定 referrer_key.kind 可信，其值总是host，domain 或 url，但是referrer_key.key_string 可能是任何内容，尤其可能包含美元符号。）这个函数实际上改进了 Oracle 原版：当referrer_key.key_string 或 referrer_key.referrer_type 中包含引号时，它也不会生成有问题的代码。

例 43.11 展示了如何移植一个带有 OUT 参数和字符串处理的函数。PostgreSQL 没有内置的 instr 函数，但是你可以用其它函数的组合来创建一个。在第 43.13.3 节中有一个 instr 的 PL/pgSQL 实现，你可以用它让你的移植变得更容易。

例 43.11. 从 PL/SQL 移植一个带有字符串操作以及 OUT 参数的过程到 PL/pgSQL

下面的 Oracle PL/SQL 过程被用来解析一个 URL 并且返回一些元素（主机、路径和查询）。

这是 Oracle 版本：

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR2,
    v_host OUT VARCHAR2,  -- 这个值将被返回
    v_path OUT VARCHAR2,  -- 这个也是
    v_query OUT VARCHAR2) -- 还有这个
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

这里给出一种可能的 PL/pgSQL 写法：

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- 这个值将被返回
    v_path OUT VARCHAR,  -- 这个也是
    v_query OUT VARCHAR) -- 还有这个
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

这个函数可以这样使用：

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

例 43.12 展示了如何移植一个使用了多种 Oracle 专属特性的过程。

例 43.12. 从 PL/SQL 移植一个过程到 PL/pgSQL

Oracle 版本：

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- 释放锁
        raise_application_error(-20000,
                'Unable to create a new job: a job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- 如果已经存在也不用担心
    END;
    COMMIT;
END;
/
show errors
```

下面展示了如何将这个过程移植到 PL/pgSQL：

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id integer) AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- 释放锁
        RAISE EXCEPTION 'Unable to create a new job: a job is currently running'; -- 1
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN -- 2
            -- 如果已经存在不要担心
    END;
    COMMIT;
END;
$$ LANGUAGE plpgsql;
```

1 `RAISE` 的语法与 Oracle 的语句相当不同，尽管基本的形式`RAISE` *`exception_name`* 工作起来是相似的。

2 PL/pgSQL 所支持的异常名称不同于 Oracle。内置的异常名称集合要更大（见附录 A）。目前没有办法声明用户定义的异常名称，尽管你能够抛出用户选择的 SQLSTATE 值。

43.13.2. 其他要关注的事项

这一节解释了在移植 Oracle PL/SQL 函数到 PostgreSQL 中时要关注的一些其他问题。

43.13.2.1. 异常后隐式回滚

在 PL/pgSQL 中，当异常被EXCEPTION 子句捕获后，自该块的BEGIN 以来所做的所有数据库更改都会被自动回滚。也就是说，这种行为等效于你在 Oracle 中使用下面的代码所得到的效果：

```
BEGIN
    SAVEPOINT s1;
    ... code here ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
END;
```

如果你正在移植一个以这种方式使用 SAVEPOINT 和 ROLLBACK TO 的 Oracle 过程，工作会比较简单：只要省略SAVEPOINT 和 ROLLBACK TO 即可。如果你的 Oracle 过程以不同方式使用 SAVEPOINT 和 ROLLBACK TO，那就需要认真思考了。

43.13.2.2. EXECUTE

PL/pgSQL 中的 EXECUTE 与 PL/SQL 中的版本工作方式相似，但必须记得按照第 43.5.4 节中的说明使用 quote_literal 和 quote_ident。像 EXECUTE 'SELECT * FROM $1'; 这样的写法如果不借助这些函数，将无法可靠地工作。

43.13.2.3. 优化 PL/pgSQL 函数

PostgreSQL 提供了两种函数创建修饰符来优化执行：“volatility”（对于给定的相同参数，函数是否总是返回相同的结果）以及“strictness”（如果任何参数为空值，函数是否返回空值）。详见CREATE FUNCTION 参考页。

在利用这些优化属性时，你的CREATE FUNCTION 语句可能像这样：

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

43.13.3. 附录

这一节包含了一组 Oracle 兼容的 instr 函数代码，你可以用它来简化你的移植工作。

```
--
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1, string2 [, n [, m]])
-- where [] denotes optional parameters.
--
-- Search string1, beginning at the nth character, for the mth occurrence
-- of string2.  If n is negative, search backwards, starting at the abs(n)'th
```

```
-- character from the end of string1.
-- If n is not passed, assume 1 (search starts at first character).
-- If m is not passed, assume 1 (find first occurrence).
-- Returns starting index of string2 in string1, or 0 if string2 is not found.
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
BEGIN
    RETURN instr($1, $2, 1);
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search_for IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
```

```
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF occur_index <= 0 THEN
        RAISE 'argument ''%'' is out of range', occur_index
          USING ERRCODE = '22003';
    END IF;

    IF beg_index > 0 THEN
        beg := beg_index - 1;
        FOR i IN 1..occur_index LOOP
            temp_str := substring(string FROM beg + 1);
            pos := position(string_to_search_for IN temp_str);
            IF pos = 0 THEN
                RETURN 0;
            END IF;
            beg := beg + pos;
        END LOOP;

        RETURN beg;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                occur_number := occur_number + 1;
                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

第 44 章 PL/Tcl - Tcl 过程语言

PL/Tcl 是一种适用于 PostgreSQL 数据库系统的可加载的过程语言，它允许使用 Tcl 语言[1]来编写 PostgreSQL 函数和过程。

44.1. 概述

PL/Tcl 提供了函数编写者在 C 语言中所具备的大部分能力，但会施加少量限制，并额外提供 Tcl 可用的强大字符串处理库。

其中一个颇具吸引力的良性限制是，所有内容都在 Tcl 解释器的安全上下文中执行。除了安全 Tcl 受限的命令集之外，只提供了少数通过 SPI 访问数据库以及通过`elog()` 发出消息的命令。与 C 函数不同，PL/Tcl 不提供访问数据库服务器内部机制的方法，也不能在PostgreSQL 服务器进程的权限下获得操作系统级访问。因此，可以信任非特权数据库用户使用这种语言；它不会授予他们无限制的权限。

另一项值得注意的实现限制是，Tcl 函数不能用来为新数据类型创建输入/输出函数。

有时需要编写不受安全 Tcl 限制的 Tcl 函数。例如，可能希望某个 Tcl 函数能够发送电子邮件。为处理这类情况，PL/Tcl 提供了一个名为 `PL/TclU` 的变体（其中 U 表示不受信任的 Tcl）。除了使用完整的 Tcl 解释器之外，它与前者是完全相同的语言。如果使用 PL/TclU，就必须将其安装为一种不受信任的过程语言，这样就只有数据库超级用户才能在其中创建函数。PL/TclU 函数的编写者必须确保该函数不会被用来做任何不希望发生的事情，因为它能够执行任何以数据库管理员身份登录的用户所能做的事情。

如果在安装过程的配置步骤中指定了 Tcl 支持，则PL/Tcl 和 PL/TclU 调用处理器的共享对象代码会自动构建并安装到 PostgreSQL 的库目录中。要在某个特定数据库中安装 PL/Tcl 和/或 PL/TclU，请使用`CREATE EXTENSION` 命令，例如`CREATE EXTENSION pltcl` 或 `CREATE EXTENSION pltclu`。

44.2. PL/Tcl 函数和参数

要用 PL/Tcl 语言创建函数，可使用标准的CREATE FUNCTION 语法：

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
    # PL/Tcl 函数体
$$ LANGUAGE pltcl;
```

PL/TclU 的写法相同，只是语言必须指定为`pltclu`。

函数体就是一段 Tcl 脚本。调用函数时，参数值会作为名为`1` ... *n* 的变量传递给 Tcl 脚本。结果按通常方式通过 `return` 语句从 Tcl 代码中返回。对于过程，Tcl 代码的返回值会被忽略。

[1] https://www.tcl.tk/

例如，一个返回两个整数值中较大值的函数可以定义为：

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

注意 `STRICT` 子句，它让我们不必考虑空值输入：如果传入的是空值，函数根本不会被调用，而是会自动返回空值结果。

在非严格函数中，如果某个参数的实际值为空值，对应的*$n* 变量会被设置为空串。要检测某个特定参数是否为空值，可使用函数`argisnull`。例如，假设我们希望在`tcl_max` 的两个参数中一个为空值、一个非空值时返回非空值参数，而不是返回空值：

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

如上所示，要从 PL/Tcl 函数返回空值，请执行`return_null`。无论函数是否为严格函数，都可以这样做。

复合类型参数会作为 Tcl 数组传递给函数。数组元素名就是该复合类型的属性名。如果传入行中的某个属性为空值，它就不会出现在数组中。下面是一个示例：

```
CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

PL/Tcl 函数也可以返回复合类型结果。要做到这一点，Tcl 代码必须返回一个与预期结果类型匹配的“列名/值”对列表。凡是未在列表中出现的列名，都会作为空值返回；如果出现预期之外的列名，则会报错。下面是一个示例：

```
CREATE FUNCTION square_cube(in int, out squared int, out cubed int) AS $$
    return [list squared [expr {$1 * $1}] cubed [expr {$1 * $1 * $1}]]
$$ LANGUAGE pltcl;
```

过程的输出参数也按同样的方式返回，例如：

```
CREATE PROCEDURE tcl_triple(INOUT a integer, INOUT b integer) AS $$
    return [list a [expr {$1 * 3}] b [expr {$2 * 3}]]
$$ LANGUAGE pltcl;

CALL tcl_triple(5, 10);
```

提示

可使用 Tcl 的 `array get` 命令，根据期望元组的数组表示构造结果列表。例如：

```
CREATE FUNCTION raise_pay(employee, delta int) RETURNS employee AS $$
    set 1(salary) [expr {$1(salary) + $2}]
    return [array get 1]
$$ LANGUAGE pltcl;
```

PL/Tcl 函数也可以返回集合。要做到这一点，Tcl 代码应当对每个要返回的行调用一次 `return_next`：返回标量类型时传入适当的值，返回复合类型时传入“列名/值”对的列表。下面是返回标量类型的示例：

```
CREATE FUNCTION sequence(int, int) RETURNS SETOF int AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next $i
    }
$$ LANGUAGE pltcl;
```

下面是返回复合类型的示例：

```
CREATE FUNCTION table_of_squares(int, int) RETURNS TABLE (x int, x2 int) AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next [list x $i x2 [expr {$i * $i}]]
    }
$$ LANGUAGE pltcl;
```

44.3. PL/Tcl 中的数据值

提供给 PL/Tcl 函数代码的参数值，只是将输入参数转换成文本形式（就像用`SELECT` 语句将它们显示出来一样）。反过来，`return` 和 `return_next` 命令会接受任何字符串，只要它是该函数声明结果类型或复合结果类型中指定列的可接受输入格式。

44.4. PL/Tcl 中的全局数据

有时需要在一次函数调用结束后保留某些全局数据，供下一次调用继续使用，或者在不同函数之间共享全局数据。在 PL/Tcl 中这很容易做到，但必须理解其中的一些限制。

出于安全原因，PL/Tcl 会针对每个 SQL 角色，在该角色各自独立的 Tcl 解释器中执行其调用的函数。这样可以防止一个用户无意或恶意地干扰另一用户的 PL/Tcl 函数行为。每个这样的解释器都为任何“global” Tcl 变量维护各自的值。因此，当且仅当两个 PL/Tcl 函数由同一个 SQL 角色执行时，它们才会共享同一组全局变量。在某些应用中，一个会话会在多个 SQL 角色下执行代码（例如通过SECURITY DEFINER 函数、使用SET ROLE 等），这时你可能需要显式采取一些步骤，确保 PL/Tcl 函数能够共享数据。为此，应确保需要互相通信的函数由同一用户拥有，并将它们标记为SECURITY DEFINER。当然，你必须小心，确保这类函数不会被用来执行任何非预期操作。

一个会话中使用的所有 PL/TclU 函数都在同一个 Tcl 解释器中执行，而这个解释器当然不同于 PL/Tcl 函数所使用的解释器。因此，PL/TclU 函数之间会自动共享全局数据。这不被视为安全风险，因为所有 PL/TclU 函数都在相同的信任级别上执行，也就是数据库超级用户的级别。

为帮助防止 PL/Tcl 函数无意间彼此干扰，每个函数都可通过upvar 命令访问一个全局数组。该变量的全局名称是函数的内部名称，局部名称是GD。建议将 GD 用于保存函数的持久私有数据。只有当你明确希望某些值在多个函数之间共享时，才应使用常规的 Tcl 全局变量。（注意GD 数组只在某个特定解释器内部是全局的，因此它们不会绕过上文提到的安全限制。）

下面 spi_execp 的示例展示了 GD 的用法。

44.5. 从 PL/Tcl 访问数据库

在本节中，我们遵循 Tcl 的通常约定，用问号而不是方括号来表示语法摘要中的可选元素。下列命令可用于从 PL/Tcl 函数体中访问数据库：

spi_exec ?-count *n*? ?-array *name*? *command* ?*loop-body*?

执行以字符串形式给出的 SQL 命令。命令出错时会引发错误。否则，spi_exec 的返回值是该命令处理的行数（选出、插入、更新或删除的行），如果命令是工具语句则返回零。此外，如果命令是SELECT 语句，则所选列的值会按下文所述放入 Tcl 变量中。

可选的-count 值告诉 spi_exec，一旦已检索到*n* 行就停止，就像查询中包含了LIMIT 子句一样。如果 *n* 为零，则查询会运行到结束，这与省略-count 时相同。

如果命令是 SELECT 语句，则结果列的值会放入以列名命名的 Tcl 变量中。如果给定了 -array 选项，则列值会存储在指定的关联数组元素中，列名用作数组索引。此外，结果中的当前行号（从零开始计数）将存储在数组元素中，该数组元素的名称为“.tupno”，除非该名称在结果中已被用作列名。

如果命令是 SELECT 语句且未给出 *loop-body* 脚本，则只会把结果的第一行存入 Tcl 变量或数组元素中；其余行如果存在，会被忽略。如果查询没有返回任何行，则不会存储任何内容。（可以通过检查spi_exec 的结果来检测这种情况。）例如：

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

会把 Tcl 变量 $cnt 设置为 pg_proc 系统目录中的行数。

如果给出了可选的 *loop-body* 参数，它是一段 Tcl 脚本，会对查询结果中的每一行执行一次。（如果给定命令不是SELECT，则*loop-body* 会被忽略。）在每次迭代前，当前行各列的值都会被存入 Tcl 变量或数组元素中。例如：

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

会为 pg_class 的每一行打印一条日志消息。这个特性与其他 Tcl 循环结构的工作方式类似；特别是 continue 和 break 在循环体内按通常方式工作。

如果查询结果中的某一列为空值，则对应的目标变量会被“unset”，而不是被设值。

spi_prepare *query typelist*

准备并保存一个查询计划以供后续执行。保存的计划会在当前会话的整个生命周期内保留。

该查询可以使用参数，也就是在实际执行计划时提供值的占位符。在查询字符串中，通过符号$1 ... $*n* 引用参数。如果查询使用了参数，则必须把参数类型名称作为 Tcl 列表给出。（如果不使用参数，则给*typelist* 写一个空列表。）

spi_prepare 的返回值是一个查询 ID，供后续调用spi_execp 时使用。示例参见 spi_execp。

spi_execp ?-count *n*? ?-array *name*? ?-nulls *string*? *queryid* ?*value-list* ? ?*loop-body*?

执行先前用 spi_prepare 准备好的查询。*queryid* 是 spi_prepare 返回的 ID。如果查询引用了参数，则必须提供*value-list*。这是参数实际值构成的 Tcl 列表。该列表的长度必须与先前提供给 spi_prepare 的参数类型列表相同。如果查询没有参数，则省略 *value-list*。

可选的-nulls 值是由空格和'n' 字符组成的字符串，用来告诉spi_execp 哪些参数是空值。如果给出，它的长度必须与*value-list* 完全相同。如果未给出，则所有参数值都视为非空值。

除了指定查询及其参数的方式不同之外，spi_execp 的工作方式与 spi_exec 完全一样。-count、-array 和 *loop-body* 选项都相同，返回值也相同。

下面是使用已准备计划的 PL/Tcl 函数示例：

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # 在首次调用时准备并保存计划
        set GD(plan) [ spi_prepare \
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;
```

我们需要在传给 spi_prepare 的查询字符串中加入反斜杠，以确保$*n* 标记会原样传递给 spi_prepare，而不会被 Tcl 执行变量替换。

`spi_lastoid`

如果最后一次 `spi_exec` 或 `spi_execp` 执行的是单行 `INSERT`，且被修改的表包含 OID，则返回所插入行的 OID。（否则返回零。）

`subtransaction` *`command`*

包含在 *`command`* 中的 Tcl 脚本会在 SQL 子事务内执行。如果脚本返回错误，整个子事务会在将错误返回给外围 Tcl 代码之前回滚。有关更多详细信息和示例，请参见第 44.9 节。

`quote` *`string`*

将给定字符串中的所有单引号和反斜杠字符都加倍。这可用于安全地为那些要插入到传给 `spi_exec` 或 `spi_prepare` 的 SQL 命令中的字符串加引号。例如，考虑如下 SQL 命令字符串：

```
"SELECT '$val' AS ret"
```

其中 Tcl 变量 `val` 的实际内容是 `doesn't`。这会得到最终命令字符串：

```
SELECT 'doesn't' AS ret
```

这会在 `spi_exec` 或 `spi_prepare` 期间导致解析错误。要使其正常工作，提交的命令应当包含：

```
SELECT 'doesn''t' AS ret
```

在 PL/Tcl 中，可以用下面这种方式构造它：

```
"SELECT '[ quote $val ]' AS ret"
```

`spi_execp` 的一个优点是，你不必像这样给参数值加引号，因为这些参数永远不会作为 SQL 命令字符串的一部分被解析。

`elog` *`level msg`*

发出日志或错误消息。可用级别包括 `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `ERROR` 和 `FATAL`。`ERROR` 会引发错误条件；如果外围 Tcl 代码没有捕获它，该错误就会传播到调用查询，导致当前事务或子事务中止。这实际上与 Tcl 的`error` 命令相同。`FATAL` 会中止事务并导致当前会话关闭。（在 PL/Tcl 函数中使用这个错误级别可能并没有什么充分理由，但为了完整性仍提供它。）其他级别只会生成不同优先级的消息。某个特定优先级的消息是报告给客户端、写入服务器日志，还是两者都做，由log_min_messages 和 client_min_messages 配置变量控制。参见第 19 章和第 44.8 节了解更多信息。

44.6. PL/Tcl 中的触发器函数

触发器函数也可以用 PL/Tcl 编写。PostgreSQL 要求，凡是要作为触发器调用的函数，都必须声明为无参数且返回类型为`trigger`。

来自触发器管理器的信息会通过下列变量传递给函数体：

`$TG_name`

`CREATE TRIGGER` 语句中触发器的名称。

`$TG_relid`

导致触发器函数被调用的表的对象 ID。

`$TG_table_name`

触发器函数因其而被调用的表的名称。

`$TG_table_schema`

导致触发器函数被调用的表所在的模式。

`$TG_relatts`

表列名构成的 Tcl 列表，其前面带有一个空列表元素。因此，使用Tcl 的 `lsearch` 命令在该列表中查找列名时，返回的元素编号会从 1 开始表示第一列，这与PostgreSQL 中列的惯常编号方式一致。（空列表元素也会出现在已删除列的位置上，这样其右侧列的属性编号仍然正确。）

`$TG_when`

其值为字符串 `BEFORE`、`AFTER` 或 `INSTEAD OF`，具体取决于触发器事件的类型。

`$TG_level`

其值为字符串 `ROW` 或 `STATEMENT`，取决于触发器事件的类型。

`$TG_op`

其值为字符串 `INSERT`、`UPDATE`、`DELETE` 或 `TRUNCATE`，取决于触发器事件的类型。

`$NEW`

对于 `INSERT` 或 `UPDATE` 动作，这是一个包含新表行值的关联数组；对于`DELETE` 则为空。该数组以列名为索引。值为空值的列不会出现在数组中。对于语句级触发器，不会设置这个变量。

`$OLD`

对于 `UPDATE` 或 `DELETE` 动作，这是一个包含旧表行值的关联数组；对于`INSERT` 则为空。该数组以列名为索引。值为空值的列不会出现在数组中。对于语句级触发器，不会设置这个变量。

`$args`

在 `CREATE TRIGGER` 语句中传给函数的参数组成的 Tcl 列表。在函数体中也可以通过 `$1` ... `$n` 访问这些参数。

触发器函数的返回值可以是字符串 `OK` 或 `SKIP`，也可以是一个“列名/值”对列表。如果返回值是 `OK`，引发该触发器的操作（`INSERT`/`UPDATE`/`DELETE`）会正常继续。`SKIP` 告诉触发器管理器静默抑制针对这一行的该操作。如果返回的是列表，则表示 PL/Tcl 要向触发器管理器返回一个修

改后的行；该修改后行的内容由列表中的列名和值指定。凡是未在列表中提及的列，都会被设为空值。返回修改后的行只对行级BEFORE INSERT 或 UPDATE 触发器有意义；对于这类触发器，插入的将是修改后的行，而不是$NEW 中给出的那一行。返回修改后的行对行级 INSTEAD OF INSERT 或 UPDATE 触发器也有意义，在这种情况下，返回的行将被用作INSERT RETURNING 或 UPDATE RETURNING 子句的源数据。对于行级 BEFORE DELETE 或 INSTEAD OF DELETE 触发器，返回修改后的行与返回OK 效果相同，也就是操作继续。对于所有其他类型的触发器，返回值都会被忽略。

> **提示**
>
> 可使用 Tcl 的 array get 命令，根据修改后元组的数组表示构造结果列表。

下面是一个小型触发器函数示例。它让表中的某个整数值跟踪针对该行执行的更新次数。对于新插入的行，该值初始化为 0，此后每次执行更新操作时都会递增。

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE FUNCTION trigfunc_modcount('modcnt');
```

注意，触发器函数本身并不知道列名；列名是通过触发器参数提供的。这使得该触发器函数可以在不同的表上重用。

44.7. PL/Tcl 中的事件触发器函数

事件触发器函数也可以用 PL/Tcl 编写。PostgreSQL 要求，凡是要作为事件触发器调用的函数，都必须声明为无参数且返回类型为event_trigger。

来自触发器管理器的信息会通过下列变量传递给函数体：

$TG_event

该触发器所针对的事件名称。

`$TG_tag`

该触发器所针对的命令标签。

触发器函数的返回值会被忽略。

下面是一个小型事件触发器函数示例。每当执行受支持的命令时，它都会简单地发出一条NOTICE 消息：

```
CREATE OR REPLACE FUNCTION tclsnitch() RETURNS event_trigger AS $$
  elog NOTICE "tclsnitch: $TG_event $TG_tag"
$$ LANGUAGE pltcl;

CREATE EVENT TRIGGER tcl_a_snitch ON ddl_command_start EXECUTE FUNCTION tclsnitch();
```

44.8. PL/Tcl 中的错误处理

PL/Tcl 函数内的 Tcl 代码，或从 PL/Tcl 函数调用的 Tcl 代码，都可能引发错误：要么是执行了某个非法操作，要么是通过 Tcl 的`error` 命令或 PL/Tcl 的 `elog` 命令主动生成错误。在 Tcl 中，可以使用`catch` 命令捕获这类错误。如果错误没有被捕获，而是一路传播到 PL/Tcl 函数执行的顶层，那么它就会在调用该函数的查询中作为 SQL 错误报告出来。

反过来，在 PL/Tcl 的`spi_exec`、`spi_prepare` 和 `spi_execp` 命令中发生的 SQL 错误会作为 Tcl 错误报告，因此也可以被 Tcl 的`catch` 命令捕获。（这些 PL/Tcl 命令中的每一个都会在子事务中运行其 SQL 操作，该子事务在出错时会回滚，因此任何部分完成的操作都会自动清理掉。）同样地，如果某个错误在未被捕获的情况下传播到顶层，它又会重新变成 SQL 错误。

Tcl 提供了一个 `errorCode` 变量，它以便于 Tcl 程序解释的形式携带关于错误的附加信息。该变量的内容采用 Tcl 列表格式，第一个词标识报告该错误的子系统或库；其后的内容则由相应子系统或库自行定义。对于由 PL/Tcl 命令报告的数据库错误，第一个词是`POSTGRES`，第二个词是 PostgreSQL 版本号，后续内容则是字段名/字段值对，用来提供关于该错误的详细信息。字段`SQLSTATE`、`condition` 和 `message` 总是会提供（前两个分别对应附录 A 中所示的错误代码和条件名称）。可能出现的字段包括 `detail`、`hint`、`context`、`schema`、`table`、`column`、`datatype`、`constraint`、`statement`、`cursor_position`、`filename`、`lineno` 以及 `funcname`。

处理 PL/Tcl 的 `errorCode` 信息时，一种方便的办法是把它载入数组中，这样字段名就变成了数组下标。对应的代码可能如下所示：

```
if {[catch { spi_exec $sql_command }]} {
    if {[lindex $::errorCode 0] == "POSTGRES"} {
        array set errorArray $::errorCode
        if {$errorArray(condition) == "undefined_table"} {
            # 处理表不存在的情况
        } else {
            # 处理其他类型的 SQL 错误
        }
    }
}
```

（双冒号显式指定 `errorCode` 是一个全局变量。）

44.9. PL/Tcl 中的显式子事务

从第 44.8 节所述的数据库访问错误中恢复，可能会导致一种不理想的情况：某些操作在其中一个失败之前已经成功完成，而从该错误恢复之后，数据仍处于不一致状态。PL/Tcl 通过显式子事务为这个问题提供了解决方案。

考虑一个在两个账户之间执行转账的函数：

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
    if [catch {
        spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'"
        spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'"
    } errormsg] {
        set result [format "error transferring funds: %s" $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

如果第二个 UPDATE 语句引发异常，这个函数会记录这次失败，但第一个UPDATE 的结果仍会被提交。换句话说，资金会从 Joe 的账户中扣除，却不会转入 Mary 的账户。这是因为每个 spi_exec 都是一个独立的子事务，而其中只有一个子事务发生了回滚。

要处理这类情况，可以把多个数据库操作包裹在一个显式子事务中，使其作为整体成功完成或整体回滚。PL/Tcl 提供了subtransaction 命令来管理这一点。我们可以把函数改写为：

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
    if [catch {
        subtransaction {
            spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'"
            spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'"
        }
    } errormsg] {
        set result [format "error transferring funds: %s" $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

注意，为达到这个目的，仍然需要使用catch。否则错误会传播到函数顶层，从而阻止对 operations 表执行期望的插入。subtransaction 命令并不会捕获错误，它只保证在其作用域内执行的所有数据库操作，会在报告错误时一起回滚。

显式子事务会在其包含的 Tcl 代码报告任何错误时回滚，而不仅仅是在数据库访问导致错误时才回滚。因此，在subtransaction 命令内部引发的普通 Tcl 异常也会导致子事务回滚。不过，从其中包含的 Tcl 代码中以非错误方式退出（例如由于return）则不会导致回滚。

44.10. 事务管理

在从顶层调用的过程，或者从顶层调用的匿名代码块（DO 命令）中，可以控制事务。要提交当前事务，可调用commit 命令。要回滚当前事务，可调用rollback 命令。（注意，不能通过 spi_exec 或类似命令执行 SQL 命令 COMMIT 或 ROLLBACK；必须使用这些命令来完成。）一个事务结束后，会自动启动新事务，因此没有单独的命令用于此事。

下面是一个示例：

```
CREATE PROCEDURE transaction_test1()
LANGUAGE pltcl
AS $$
for {set i 0} {$i < 10} {incr i} {
    spi_exec "INSERT INTO test1 (a) VALUES ($i)"
    if {$i % 2 == 0} {
        commit
    } else {
        rollback
    }
}
$$;

CALL transaction_test1();
```

当显式子事务处于活动状态时，不能结束事务。

44.11. PL/Tcl 配置

本节列出影响 PL/Tcl 的配置参数。

`pltcl.start_proc`(`string`)

如果该参数被设置为非空字符串，它就指定一个无参数 PL/Tcl 函数的名称（可能带模式限定）。每当为 PL/Tcl 创建新的 Tcl 解释器时，这个函数都会被执行。这样的函数可以执行每个会话的初始化，例如装载额外的 Tcl 代码。当某个 PL/Tcl 函数在数据库会话中第一次执行时，或者由于某个 PL/Tcl 函数是由新的 SQL 角色调用而必须再创建一个解释器时，就会创建新的 Tcl 解释器。

被引用的函数必须使用 `pltcl` 语言编写，并且不得标记为`SECURITY DEFINER`。（这些限制可确保它运行在自己应当初始化的解释器中。）当前用户还必须拥有调用它的权限。

如果该函数因错误而失败，就会中止导致新解释器被创建的那个函数调用，并把错误传播到调用查询，进而导致当前事务或子事务中止。已经在 Tcl 中完成的任何动作都不会被撤销；不过，该解释器之后将不会再被使用。如果再次使用该语言，就会在一个全新的 Tcl 解释器中再次尝试初始化。

只有超级用户才能更改此设置。尽管可以在会话内更改该设置，但这类更改不会影响已经创建好的 Tcl 解释器。

`pltclu.start_proc`(`string`)

这个参数与 `pltcl.start_proc` 完全相同，只不过它适用于 PL/TclU。被引用的函数必须使用`pltclu` 语言编写。

44.12. Tcl 过程名

在 PostgreSQL 中，只要参数个数或参数类型不同，就可以复用同一个函数名。不过，Tcl 要求所有过程名都必须不同。PL/Tcl 处理这一问题的方式是：在内部 Tcl 过程名中包含系统表 `pg_proc` 中的函数对象 ID 作为其名称的一部分。因此，名称相同但参数类型不同的PostgreSQL 函数，也会对应不同的 Tcl 过程。这通常不是 PL/Tcl 程序员需要关心的事情，但在调试时可能会看见。

第 45 章 PL/Perl - Perl 过程语言

PL/Perl 是一种可加载的过程语言，它允许使用 Perl 编程语言[1]编写 PostgreSQL 函数和过程。

使用 PL/Perl 的主要优势在于，它允许在存储函数和过程中使用 Perl 提供的大量“字符串处理”操作符和函数。与 PL/pgSQL 提供的字符串函数和控制结构相比，使用 Perl 解析复杂字符串可能更容易。

要在特定数据库中安装 PL/Perl，可使用 `CREATE EXTENSION plperl`。

> **提示**
>
> 如果把某种语言安装到 `template1` 中，之后创建的所有数据库都会自动安装该语言。

> **注意**
>
> 使用源码包的用户必须在安装过程中专门启用 PL/Perl 的构建（更多信息见第 16 章）。使用二进制包的用户可能会在单独的子包中找到 PL/Perl。

45.1. PL/Perl 函数和参数

要用 PL/Perl 语言创建一个函数，可使用标准的CREATE FUNCTION 语法：

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
    # PL/Perl 函数体
$$ LANGUAGE plperl;
```

函数的主体就是普通的 Perl 代码。事实上，PL/Perl 的粘合代码会把它包裹在一个 Perl 子程序中。PL/Perl 函数在标量上下文中被调用，因此不能返回列表。如下文所述，可以通过返回引用来返回非标量值（数组、记录和集合）。

在 PL/Perl 过程中，Perl 代码返回的任何值都会被忽略。

PL/Perl 也支持用 DO 语句调用的匿名代码块：

```
DO $$
    # PL/Perl 代码
$$ LANGUAGE plperl;
```

匿名代码块不接收参数，它可能返回的任何值都会被丢弃。除此之外，它的行为与函数完全相同。

[1] https://www.perl.org

> **注意**
>
> 在 Perl 中使用命名嵌套子程序是有危险的，特别是当它们引用外层作用域中的词法变量时。因为 PL/Perl 函数被包装成一个子程序，任何放在其中的命名子程序都会成为嵌套子程序。一般来说，创建通过代码引用（coderef）调用的匿名子程序会安全得多。更多信息可见 perldiag 手册页中的 `Variable "%s" will not stay shared` 以及 `Variable "%s" is not available`，或者在互联网上搜索 “perl nested named subroutine” 。

`CREATE FUNCTION` 命令的语法要求函数体写成一个字符串常量。通常对该字符串常量使用美元引用（见第 4.1.2.4 节）最方便。如果选择使用转义字符串语法 `E''`，则必须将函数体中用到的单引号（`'`）和反斜线（`\`）都写成双份（见第 4.1.2.1 节）。

参数和结果的处理与其他 Perl 子程序一样：参数通过 `@_` 传入，结果值通过 `return` 返回，或者作为函数中最后一个被求值的表达式返回。

例如，一个返回两个整数值中较大值的函数可以定义为：

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

> **注意**
>
> 参数会从数据库编码转换为 PL/Perl 中使用的 UTF-8，返回时再从 UTF-8 转回数据库编码。

如果一个 SQL 空值被传给一个函数，在 Perl 中该参数值将呈现为 “未定义” 。上述函数定义对于空值输入的行为并不理想（实际上，它会把它们当作零）。我们可以在函数定义中添加 `STRICT`，让 PostgreSQL 采取更合理的做法：如果传入空值，函数将根本不会被调用，而是自动返回空值结果。另一种方式是在函数体中检查未定义输入。例如，假设我们希望在 `perl_max` 的两个参数中一个为空值、另一个非空值时，返回非空值参数而不是空值：

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($x, $y) = @_;
    if (not defined $x) {
        return undef if not defined $y;
        return $y;
    }
    return $x if not defined $y;
    return $x if $x > $y;
    return $y;
$$ LANGUAGE plperl;
```

如上所示，要从 PL/Perl 函数返回 SQL 空值，就返回一个未定义值。无论函数是否为严格函数，都可以这样做。

函数参数中任何不是引用的值都是字符串，其格式是相应数据类型的标准 PostgreSQL 外部文本表示。对于普通的数值或文本类型，Perl 通常会正确处理，程序员一般不必操心。不过在其他情况下，参数可能需要转换成 Perl 中更易用的形式。例如，可以用 `decode_bytea` 函数把 `bytea` 类型的参数转换成未转义的二进制数据。

同样，返回给 PostgreSQL 的值也必须采用外部文本表示格式。例如，可以用 `encode_bytea` 函数把二进制数据转义成适合作为 `bytea` 类型返回值的形式。

Perl 可以把 PostgreSQL 数组返回为对 Perl 数组的引用。这里有一个示例：

```
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;

select returns_array();
```

Perl 把 PostgreSQL 数组作为一个经 bless 的 `PostgreSQL::InServer::ARRAY` 对象传递。该对象既可以当作数组引用，也可以当作字符串处理，从而可以运行为 9.1 之前版本的 PostgreSQL 编写的 Perl 代码，以保持向后兼容性。例如：

```
CREATE OR REPLACE FUNCTION concat_array_elements(text[]) RETURNS TEXT AS $$
    my $arg = shift;
    my $result = "";
    return undef if (!defined $arg);

    # 作为数组引用
    for (@$arg) {
        $result .= $_;
    }

    # 也可以作为字符串使用
    $result .= $arg;

    return $result;
$$ LANGUAGE plperl;

SELECT concat_array_elements(ARRAY['PL','/','Perl']);
```

> **注意**
>
> 多维数组按 Perl 程序员熟悉的方式表示为指向较低维数组的引用，而这些数组的元素又是引用。

复合类型参数被作为 hash 的引用传递给函数。hash 的键是复合类型的属性名。这里是一个示例：

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT name, empcomp(employee.*) FROM employee;
```

PL/Perl 函数也可以用相同的方法返回复合类型：返回一个具有所需属性的 hash 的引用。例如：

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

在声明的结果数据类型中，凡是 hash 中未出现的列都将作为空值返回。

同样，过程的输出参数也可以作为 hash 引用返回：

```
CREATE PROCEDURE perl_triple(INOUT a integer, INOUT b integer) AS $$
    my ($a, $b) = @_;
    return {a => $a * 3, b => $b * 3};
$$ LANGUAGE plperl;

CALL perl_triple(5, 10);
```

PL/Perl 函数还可以返回标量类型或复合类型的集合。通常会希望逐行返回，这既能缩短启动时间，也能避免把整个结果集排入内存。如下所示，可以用 return_next 实现这一点。注意，在最后一次 return_next 之后，必须写上 return 或者（更好）return undef。

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

对于较小的结果集，可以返回一个数组引用，其中对简单类型、数组类型和复合类型分别包含标量、数组引用或 hash 引用。下面是把整个结果集作为数组引用返回的几个简单示例：

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
    ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

如果希望在代码中使用 strict 编译指示，可以有几种选择。对于临时的全局用法，可以 SET plperl.use_strict 为真。这会影响后续编译的 PL/Perl 函数，但不会影响当前会话中已经编译的函数。对于永久的全局用法，可以在 postgresql.conf 文件中将 plperl.use_strict 设为真。

如果要在特定函数中永久启用它，可以简单地把

```
use strict;
```

放在函数体顶部。

如果 Perl 版本为 5.10.0 或更高，也可以通过 use 使用 feature 编译指示。

45.2. PL/Perl 中的数据值

提供给 PL/Perl 函数代码的参数值，就是转换成文本形式的输入参数（就像它们由 SELECT 语句显示出来一样）。相反，return 和 return_next 命令接受任何可作为该函数声明返回类型输入格式的字符串。

45.3. 内置函数

45.3.1. 从 PL/Perl 访问数据库

可以通过下列函数在 Perl 函数中访问数据库本身：

spi_exec_query(*query* [, *limit*])

spi_exec_query 执行 SQL 命令，并把整个行集作为 hash 引用数组的引用返回。如果指定了 *limit* 且其大于零，则 spi_exec_query 最多只会取回 *limit* 行，就像查询中包含了 LIMIT 子句一样。省略 *limit* 或将其指定为零时，不会限制返回行数。

只有在确定结果集会比较小时，才应使用此命令。下面是一个带可选最大行数的查询（SELECT 命令）示例：

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

这样会从表 my_table 返回最多 5 行。如果 my_table 有一列 my_column，可以像下面这样从结果的第 $i 行取得该值：

```
$foo = $rv->{rows}[$i]->{my_column};
```

可以这样访问 SELECT 查询返回的总行数：

```
$nrows = $rv->{processed}
```

下面是使用另一种命令类型的示例：

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

可以这样访问命令状态（例如 SPI_OK_INSERT）：

```
$res = $rv->{status};
```

要获取受影响的行数，可使用：

```
$nrows = $rv->{processed};
```

这里是一个完整的示例：

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
    foreach my $rn (0 .. $nrows - 1) {
        my $row = $rv->{rows}[$rn];
        $row->{i} += 200 if defined($row->{i});
        $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

```
spi_query(command)
spi_fetchrow(cursor)
spi_cursor_close(cursor)
```

spi_query 和 spi_fetchrow 需要配合使用，适用于结果集可能很大，或者希望在行到达时立即返回的情况。spi_fetchrow 只能与 spi_query 一起使用。下面的示例演示了它们的配合方式：

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
    use Digest::MD5 qw(md5_hex);
    my $file = '/usr/share/dict/words';
    my $t = localtime;
    elog(NOTICE, "opening file $file at $t" );
    open my $fh, '<', $file # 注意，这里访问了文件!
        or elog(ERROR, "cannot open $file for reading: $!");
    my @words = <$fh>;
    close $fh;
    $t = localtime;
    elog(NOTICE, "closed file $file at $t");
    chomp(@words);
    my $row;
    my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
    while (defined ($row = spi_fetchrow($sth))) {
        return_next({
            the_num => $row->{a},
            the_text => md5_hex($words[rand @words])
        });
    }
    return;
$$ LANGUAGE plperlu;

SELECT * from lotsa_md5(500);
```

通常，应重复调用 spi_fetchrow，直到它返回 undef，这表示已经没有更多行可读。当 spi_fetchrow 返回 undef 时，spi_query 返回的游标会被自动释放。如果不打算读取所有行，则应调用 spi_cursor_close 释放游标。否则会导致内存泄漏。

```
spi_prepare(command, argument types)
spi_query_prepared(plan, arguments)
spi_exec_prepared(plan [, attributes], arguments)
spi_freeplan(plan)
```

spi_prepare、spi_query_prepared、spi_exec_prepared 和 spi_freeplan 提供相同的功能，但用于预备查询。spi_prepare 接受一个带编号参数占位符（$1、$2 等）的查询字符串，以及参数类型的字符串列表：

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name = $2',
                                                    'INTEGER', 'TEXT');
```

一旦通过调用 spi_prepare 准备好查询计划，就可以用它代替字符串查询。可以在 spi_exec_prepared 中使用，此时结果与 spi_exec_query 返回的结果相同；也可以在 spi_query_prepared 中使用，它会像 spi_query 一样返回一个游标，随后可传给 spi_fetchrow。spi_exec_prepared 的可选第二个参数是属性 hash 引用；目前唯一支持的属性是 limit，它用于设置查询返回的最大行数。省略 limit 或将其指定为零时，不会限制返回行数。

预备查询的优点在于，一个准备好的计划可以用于多次查询执行。当计划不再需要时，可以用 spi_freeplan 将其释放：

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
        $_SHARED{my_plan} = spi_prepare('SELECT (now() + $1)::date AS now',
                                        'INTERVAL');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0]
        )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();

  add_time  |  add_time  |  add_time
------------+------------+------------
 2005-12-10 | 2005-12-11 | 2005-12-12
```

请注意，spi_prepare 中的参数下标由 $1、$2、$3 等表示，因此应避免用双引号声明查询字符串，以免轻易引入难以察觉的错误。

下面的另一个示例展示了如何在 spi_exec_prepared 中使用可选参数：

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address
                      FROM generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
        $_SHARED{plan} = spi_prepare('SELECT * FROM hosts
                                      WHERE address << $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
        return spi_exec_prepared(
                $_SHARED{plan},
                {limit => 2},
                $_[0]
        )->{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
        spi_freeplan($_SHARED{plan});
        undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

    query_hosts
-----------------
 (1,192.168.1.1)
 (2,192.168.1.2)
(2 rows)
```

spi_commit()
spi_rollback()

提交或回滚当前事务。只能在从顶层调用的过程或匿名代码块（DO 命令）中调用这些函数。请注意，无法通过 spi_exec_query 或类似函数执行 COMMIT 或 ROLLBACK 这类 SQL 命令；必须使用这些函数来完成。在一个事务结束后，会自动启动一个新事务，因此不需要单独的函数来开始新事务。

这里是一个示例：

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plperl
AS $$
foreach my $i (0..9) {
    spi_exec_query("INSERT INTO test1 (a) VALUES ($i)");
    if ($i % 2 == 0) {
        spi_commit();
    } else {
        spi_rollback();
    }
}
$$;

CALL transaction_test1();
```

45.3.2. PL/Perl 中的辅助函数

`elog(`*`level`*`, `*`msg`*`)`

发出日志消息或错误信息。可用级别包括 `DEBUG`、`LOG`、`INFO`、`NOTICE`、`WARNING` 以及 `ERROR`。`ERROR` 会引发错误条件；如果周围的 Perl 代码没有捕获它，错误就会传播到调用查询，导致当前事务或子事务被中止。这实质上等同于 Perl 的 `die` 命令。其他级别只会生成不同优先级的消息。某一优先级的消息是报告给客户端、写入服务器日志，还是两者兼有，由配置变量log_min_messages 和 client_min_messages 控制。详见第 19 章。

`quote_literal(`*`string`*`)`

返回给定字符串适当加引号后的形式，以便把它用作 SQL 语句字符串中的字符串字面量。嵌入的单引号和反斜线会被正确地双写。注意，对于 undef 输入，`quote_literal` 会返回 undef；如果参数可能为 undef，`quote_nullable` 往往更合适。

`quote_nullable(`*`string`*`)`

返回给定字符串适当加引号后的形式，以便把它用作 SQL 语句字符串中的字符串字面量；如果参数为 undef，则返回未加引号的字符串 “NULL” 。嵌入的单引号和反斜线会被正确地双写。

`quote_ident(`*`string`*`)`

返回给定字符串适当加引号后的形式，以便把它用作 SQL 语句字符串中的标识符。只有在必要时才会添加引号，也就是当字符串包含非标识符字符，或者会发生大小写折叠时。嵌入的引号会被正确地双写。

`decode_bytea(`*`string`*`)`

返回由给定字符串的内容表示的未转义二进制数据，该字符串应为 `bytea` 编码形式。

`encode_bytea(`*`string`*`)`

返回给定字符串中二进制数据内容的 `bytea` 编码形式。

`encode_array_literal(`*array*`)`
`encode_array_literal(`*array*`, `*delimiter*`)`

将引用数组的内容以数组字面量格式（见第 8.15.2 节）返回为字符串。若参数不是数组引用，则原样返回该参数值。如果未指定分隔符，或者分隔符为 undef，则数组字面量元素之间默认使用“, ”作为分隔符。

`encode_typed_literal(`*value*`, `*typename*`)`

将一个 Perl 变量转换为第二个参数指定的数据类型的值，并返回该值的字符串表示。它能正确处理嵌套数组和复合类型的值。

`encode_array_constructor(`*array*`)`

将引用数组的内容以数组构造器格式（第 4.2.12 节）返回为字符串。其中每个值都使用 `quote_nullable` 加引号。如果参数不是数组引用，则返回使用 `quote_nullable` 加引号后的参数值。

`looks_like_number(`*string*`)`

如果按照 Perl 的规则看，给定字符串的内容像数字，则返回真值，否则返回假值。如果参数为 undef，则返回 undef。前导和尾随空格会被忽略。`Inf` 和 `Infinity` 被视为数字。

`is_array_ref(`*argument*`)`

如果给定参数可被视为数组引用，则返回真值；也就是该参数的 ref 值为 `ARRAY` 或 `PostgreSQL::InServer::ARRAY`。否则返回假值。

45.4. PL/Perl 中的全局值

可以使用全局 hash `%_SHARED` 在当前会话的整个生命周期内跨函数调用存储数据，包括代码引用。

这是共享数据的一个简单示例：

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
    if ($_SHARED{$_[0]} = $_[1]) {
        return 'ok';
    } else {
        return "cannot set shared variable $_[0] to $_[1]";
    }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
    return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl!  How''s tricks?');
SELECT get_var('sample');
```

下面是一个稍复杂一些的、使用代码引用的示例：

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
    $_SHARED{myquote} = sub {
        my $arg = shift;
        $arg =~ s/(['\\])/\\$1/g;
        return "'$arg'";
    };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* 初始化函数 */

/* 创建一个使用加引号函数的函数 */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
    my $text_to_quote = shift;
    my $qfunc = $_SHARED{myquote};
    return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

（上面的代码也可以替换为单行 `return $_SHARED{myquote}->($_[0]);`，但代价是可读性会变差。）

出于安全原因，PL/Perl 会在每个 SQL 角色各自独立的 Perl 解释器中执行该角色调用的函数。这可以防止一个用户意外或恶意地干扰另一个用户的 PL/Perl 函数行为。每个这样的解释器都有自己的 `%_SHARED` 变量值和其他全局状态。因此，当且仅当两个 PL/Perl 函数都由同一 SQL 角色执行时，它们才会共享 `%_SHARED` 的值。在某些应用中，一个会话可能会在多个 SQL 角色下执行代码（通过 `SECURITY DEFINER` 函数、使用 `SET ROLE` 等），这时可能需要显式采取措施，确保 PL/Perl 函数能够通过 `%_SHARED` 共享数据。为此，要确保需要相互通信的函数由同一用户拥有，并将它们标记为 `SECURITY DEFINER`。当然，必须注意不要让这些函数被用于任何非预期用途。

45.5. 受信任与不受信任的 PL/Perl

通常，PL/Perl 会被安装为一种名为 `plperl` 的“受信任的”编程语言。在这种设置下，为了保持安全性，某些 Perl 操作会被禁用。一般来说，受限制的是那些与环境交互的操作，包括文件句柄操作、`require` 和 `use`（针对外部模块）。它无法像 C 函数那样访问数据库服务器进程的内部，也无法以服务器进程的权限获取操作系统级访问。因此，可以允许任何无特权的数据库用户使用这种语言。

下面示例中的函数将无法工作，因为出于安全原因不允许它做文件操作：

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    my $tmpfile = "/tmp/badfile";
    open my $fh, '>', $tmpfile
        or elog(ERROR, qq{could not open the file "$tmpfile": $!});
    print $fh "Testing writing to a file\n";
    close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
    return 1;
$$ LANGUAGE plperl;
```

这个函数的创建会失败，因为验证器会捕捉到它使用了禁用的操作。

有时需要编写不受这些限制的 Perl 函数。例如，可能需要一个能发送邮件的 Perl 函数。为处理这类情况，也可以把 PL/Perl 安装成一种“不受信任的”语言（通常称为 PL/PerlU）。在这种情况下，完整的 Perl 语言都可用。安装该语言时，使用语言名 `plperlu` 就会选择不受信任的 PL/Perl 变体。

PL/PerlU 函数的编写者必须注意，函数不能被用于任何非预期用途，因为它能够执行以数据库管理员身份登录的用户所能做的任何事情。请注意，数据库系统只允许数据库超级用户用不受信任的语言创建函数。

如果上述函数是由超级用户使用 `plperlu` 语言创建的，执行就会成功。

同样地，如果把语言指定为 `plperlu` 而不是 `plperl`，用 Perl 编写的匿名代码块也可以使用原本受限的操作，但调用者必须是超级用户。

注意

虽然 PL/Perl 函数会为每个 SQL 角色在独立的 Perl 解释器中运行，但给定会话中执行的所有 PL/PerlU 函数都运行在同一个 Perl 解释器中（而且这个解释器并不是任何一个用于 PL/Perl 函数的解释器）。这允许 PL/PerlU 函数自由共享数据，但 PL/Perl 与 PL/PerlU 函数之间不能进行通信。

注意

Perl 无法在单个进程内支持多个解释器，除非构建时使用了适当的标志，即 `usemultiplicity` 或 `useithreads` 之一。（除非确实需要使用线程，否则更推荐 `usemultiplicity`。更多细节见 perlembed 手册页。）如果 PL/Perl 使用的是未按这种方式构建的 Perl 副本，那么每个会话中就只能有一个 Perl 解释器，因此任一会话只能执行 PL/PerlU 函数，或者执行全部由同一 SQL 角色调用的 PL/Perl 函数。

45.6. PL/Perl 触发器

PL/Perl 可用于编写触发器函数。在触发器函数中，hash 引用 `$_TD` 包含有关当前触发器事件的信息。`$_TD` 是一个全局变量，对触发器的每一次调用都会得到一个单独的局部值。`$_TD` hash 引用包含以下字段：

`$_TD->{new}{foo}`

列 `foo` 的 `NEW` 值

`$_TD->{old}{foo}`

列 `foo` 的 `OLD` 值

`$_TD->{name}`

被调用的触发器名称

`$_TD->{event}`

触发器事件：`INSERT`、`UPDATE`、`DELETE`、`TRUNCATE` 或者 `UNKNOWN`

`$_TD->{when}`

触发器被调用的时机：`BEFORE`、`AFTER`、`INSTEAD OF` 或 `UNKNOWN`

`$_TD->{level}`

触发器级别：`ROW`、`STATEMENT` 或者 `UNKNOWN`

`$_TD->{relid}`

触发该触发器的表的 OID

`$_TD->{table_name}`

触发该触发器的表名

`$_TD->{relname}`

触发该触发器的表名。该字段已弃用，并且可能会在未来版本中移除。请改用 $_TD->{table_name}。

`$_TD->{table_schema}`

触发该触发器的表所在模式的名称

`$_TD->{argc}`

触发器函数的参数数目

`@{$_TD->{args}}`

触发器函数的参数。如果 `$_TD->{argc}` 为 0，则该字段不存在

行级触发器可以返回下列之一：

`return;`

执行操作

`"SKIP"`

不执行操作

`"MODIFY"`

指示触发器函数修改了 `NEW` 行

下面是一个触发器函数示例，用于说明上述部分内容：

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
        return "SKIP";    # 跳过 INSERT/UPDATE 命令
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";  # 修改行并执行 INSERT/UPDATE 命令
    } else {
        return;           # 执行 INSERT/UPDATE 命令
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE FUNCTION valid_id();
```

45.7. PL/Perl 事件触发器

PL/Perl 可用于编写事件触发器函数。在事件触发器函数中，hash 引用 `$_TD` 包含有关当前触发器事件的信息。`$_TD` 是一个全局变量，对触发器的每一次调用都会得到一个单独的局部值。`$_TD` hash 引用包含以下字段：

`$_TD->{event}`

该触发器所针对的事件名称。

`$_TD->{tag}`

该触发器所针对的命令标签。

触发器函数的返回值会被忽略。

下面是一个事件触发器函数示例，用于说明上述部分内容：

```
CREATE OR REPLACE FUNCTION perlsnitch() RETURNS event_trigger AS $$
  elog(NOTICE, "perlsnitch: " . $_TD->{event} . " " . $_TD->{tag} . " ");
$$ LANGUAGE plperl;

CREATE EVENT TRIGGER perl_a_snitch
    ON ddl_command_start
    EXECUTE FUNCTION perlsnitch();
```

45.8. PL/Perl 内部机制

45.8.1. 配置

本节列出影响 PL/Perl 的配置参数。

plperl.on_init (string)

指定在 Perl 解释器首次初始化时、在它被专用于 plperl 或 plperlu 之前，要执行的 Perl 代码。执行这段代码时 SPI 函数不可用。如果代码因错误而失败，就会中止解释器初始化，并把错误传播到调用查询，导致当前事务或子事务被中止。

Perl 代码仅限一个字符串。更长的代码可以放在模块中，并由 on_init 字符串加载。示例：

```
plperl.on_init = 'require "plperlinit.pl"'
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

任何通过 plperl.on_init 直接或间接加载的模块，都可供 plperl 使用。这可能带来安全风险。要查看已经加载了哪些模块，可以使用：

```
DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
```

如果 plperl 库被包含在 shared_preload_libraries 中，初始化就会在 postmaster 中发生；在这种情况下，需要额外考虑使 postmaster 不稳定的风险。使用此特性的主要原因是，通过 plperl.on_init 加载的 Perl 模块只需在 postmaster 启动时加载一次，之后在各个数据库会话中都可立即使用，而无需承担加载开销。不过要记住，这种开销只会对数据库会话使用的第一个 Perl 解释器被省掉，也就是 PL/PerlU，或者第一个调用 PL/Perl 函数的 SQL 角色所对应的 PL/Perl。数据库会话中后来创建的任何额外 Perl 解释器，都必须重新执行 plperl.on_init。此外，在 Windows 上预加载完全不会带来节省，因为在 postmaster 进程中创建的 Perl 解释器不会传播到子进程中。

这个参数只能在 postgresql.conf 文件中，或在服务器命令行上设置。

plperl.on_plperl_init (string)
plperl.on_plperlu_init (string)

这些参数分别指定在 Perl 解释器被专用于 plperl 或 plperlu 时要执行的 Perl 代码。这会在数据库会话中首次执行 PL/Perl 或 PL/PerlU 函数时发生；如果因为调用另一种语言，或某个新的 SQL 角色调用了 PL/Perl 函数而需要创建额外解释器时，也会发生。这是在 plperl.on_init 完成的任何初始化之后进行的。执行这段代码时 SPI 函数不可用。plperl.on_plperl_init 中的 Perl 代码是在对解释器进行“锁定”之后执行的，因此只能执行受信任的操作。

如果代码因错误而失败，就会中止初始化，并把错误传播到调用查询，导致当前事务或子事务被中止。在 Perl 中已经完成的任何动作都不会被撤销；不过，该解释器将不再被使用。如果再次使用该语言，就会在一个新的 Perl 解释器中再次尝试初始化。

只有超级用户可以更改这些设置。尽管这些设置可以在会话中修改，但这类更改不会影响已经用于执行函数的 Perl 解释器。

plperl.use_strict (boolean)

如果将其设为真，后续编译的 PL/Perl 函数都会启用 strict 编译指示。该参数不影响当前会话中已经编译的函数。

45.8.2. 限制与缺失特性

PL/Perl 目前仍缺少下列特性，但欢迎为此作出贡献。

- PL/Perl 函数不能直接调用彼此。
- SPI 尚未完全实现。
- 如果使用 `spi_exec_query` 取回非常大的数据集，应注意这些数据都会进入内存。可以像前面所示那样，通过使用 `spi_query`/`spi_fetchrow` 来避免这种情况。

  如果集合返回函数通过 `return` 把一个大型行集返回给 PostgreSQL，也会出现类似问题。前面已经说明过，可以改为对每一行使用 `return_next`，从而避免这个问题。
- 当会话正常结束（而不是由于致命错误结束）时，任何已经定义的 `END` 块都会被执行。目前不会执行其他操作。具体来说，文件句柄不会自动刷写，对象也不会自动销毁。

第 46 章 PL/Python - Python 过程语言

PL/Python 过程语言允许使用 Python 语言[1]编写 PostgreSQL 函数和过程。

要在特定数据库中安装 PL/Python，可使用`CREATE EXTENSION plpythonu`（另见第 46.1 节）。

> **提示**
>
> 如果把某种语言安装到 `template1` 中，之后创建的所有数据库都会自动安装该语言。

PL/Python 只能作为一种“不受信任的”语言使用，这意味着它不提供任何机制来限制用户能在其中做什么，因此其名称为`plpythonu`。如果将来能在 Python 中开发出安全的执行机制，受信任的变体`plpython` 可能会出现。使用不受信任的 PL/Python 编写函数时，函数编写者必须确保该函数不会被用来做任何不希望发生的事情，因为它能够做到任何以数据库管理员身份登录的用户所能做的事情。只有超级用户才能在`plpythonu` 这类不受信任的语言中创建函数。

> **注意**
>
> 源码包用户必须在安装过程中专门启用 PL/Python 的构建（更多信息请参阅安装说明）。二进制包用户则可能会在单独的子包中找到 PL/Python。

46.1. Python 2 与 Python 3

PL/Python 同时支持 Python 2 和 Python 3 语言变体。（PostgreSQL 安装说明中可能包含有关所支持 Python 精确次版本号的更多信息。）由于 Python 2 和 Python 3 语言变体在一些重要方面不兼容，PL/Python 使用以下命名和过渡方案以避免混用：

- PostgreSQL 中名为 `plpython2u` 的语言实现了基于 Python 2 语言变体的 PL/Python。
- PostgreSQL 中名为 `plpython3u` 的语言实现了基于 Python 3 语言变体的 PL/Python。
- 名为 `plpythonu` 的语言实现了基于默认 Python 语言变体的 PL/Python，当前默认为 Python 2。（此默认值独立于任何本地 Python 安装可能认为的其“默认”版本，例如 `/usr/bin/python` 可能是哪个版本。）默认值将来可能会在较远的 PostgreSQL 版本中改为 Python 3，具体取决于 Python 社区向 Python 3 迁移的进展。

该方案类似于 PEP 394[2] 中关于 `python` 命令命名和过渡的建议。

基于 Python 2 或 Python 3 的 PL/Python 是否可用（或者两者都可用），取决于构建配置或已安装的软件包。

[1] https://www.python.org
[2] https://www.python.org/dev/peps/pep-0394/

> **提示**
>
> 所构建的变体取决于安装时找到的 Python 版本，或使用PYTHON 环境变量显式设置的版本；请参见第 16.4 节。要在同一个安装中同时提供两种 PL/Python 变体，需要对源代码树进行两次配置和构建。

由此可以采用以下使用和迁移策略：

- 现有用户和目前不打算使用 Python 3 的用户，可以使用语言名 plpythonu，在可预见的将来无需更改任何内容。建议通过迁移到 Python 2.6/2.7，逐步让代码“适应未来”，以简化最终迁移到 Python 3 的工作。

  实际上，许多 PL/Python 函数只需很少或不做改动即可迁移到 Python 3。

- 如果用户清楚自己的代码严重依赖 Python 2，且不打算更改，可以使用语言名 plpython2u。它在很久以后仍会继续工作，直到 PostgreSQL 可能完全停止支持 Python 2。

- 希望开始使用 Python 3 的用户可以使用语言名 plpython3u，按照目前的标准，它会一直可用。在遥远的未来，当 Python 3 可能成为默认版本时，用户也许会出于美观而去掉名称中的“3”。

- 希望构建仅包含 Python 3 的操作系统环境的大胆用户，可以修改 pg_pltemplate 的内容，使 plpythonu 等同于 plpython3u，但要记住，这会使该安装环境与绝大多数其他环境不兼容。

有关迁移到 Python 3 的更多信息，也可参见文档 Python 3.0 的新变化[3]。

不允许在同一个会话中同时使用基于 Python 2 和基于 Python 3 的 PL/Python，因为动态模块中的符号会冲突，可能导致 PostgreSQL 服务器进程崩溃。系统会检查同一会话是否混用了不同的 Python 大版本，如果发现不匹配，就会中止会话。不过，可以在同一数据库的不同会话中分别使用这两种 PL/Python。

46.2. PL/Python 函数

PL/Python 中的函数通过标准的 CREATE FUNCTION 语法声明：

```
CREATE FUNCTION funcname (argument-list)
  RETURNS return-type
AS $$
  # PL/Python 函数体
$$ LANGUAGE plpythonu;
```

函数体只是一个 Python 脚本。调用函数时，其参数会作为列表args 的元素传入；命名参数还会作为普通变量传给 Python 脚本。使用命名参数通常可读性更好。结果通常通过 return 或 yield（在返回结果集的情况下）从 Python 代码返回。如果不提供返回值，Python 会返回默认值None。PL/Python 会把 Python 的 None 转换成 SQL 空值。对于过程，Python 代码的结果必须是None（通常做法是以不带 return 语句的方式结束过程，或者使用不带参数的return 语句），否则会引发错误。

[3] https://docs.python.org/3/whatsnew/3.0.html

例如，一个返回两个整数中较大值的函数可以定义为：

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

函数定义体中给出的 Python 代码会被转换成一个 Python 函数。例如，上面的定义会变成：

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

这里假定 23456 是由 PostgreSQL 分配给该函数的 OID。

参数会被设置为全局变量。由于 Python 的作用域规则，这会带来一个微妙的结果：除非在代码块中再次将该变量声明为全局变量，否则不能在函数内部把参数变量重新赋值为一个引用该变量名自身的表达式的值。例如，下面这样是行不通的：

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # 错误
  return x
$$ LANGUAGE plpythonu;
```

因为对 `x` 赋值会使 `x` 在整个代码块中都成为局部变量，因此赋值语句右侧的`x` 指向的是尚未赋值的局部变量 `x`，而不是 PL/Python 函数参数。使用`global` 语句后，这样写就能工作：

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # 现在可以正常工作
  return x
$$ LANGUAGE plpythonu;
```

但是，建议不要依赖 PL/Python 的这一实现细节。最好将函数参数视为只读。

46.3. 数据值

一般来说，PL/Python 的目标是在 PostgreSQL 世界和 Python 世界之间提供一种“自然”的映射。下面描述的数据映射规则就是基于这一目标。

46.3.1. 数据类型映射

调用 PL/Python 函数时，其参数会从 PostgreSQL 数据类型转换为相应的 Python 类型：

- PostgreSQL `boolean` 会转换为 Python `bool`。
- PostgreSQL 的 `smallint` 和 `int` 转换为 Python 的 `int`。PostgreSQL 的 `bigint` 和 `oid` 在 Python 2 中转换为 `long`，在 Python 3 中转换为 `int`。
- PostgreSQL `real` 和 `double` 会转换为 Python `float`。
- PostgreSQL `numeric` 会转换为 Python `Decimal`。如果可用，将从 `cdecimal` 包导入这种类型。否则，将使用标准库中的 `decimal.Decimal`。`cdecimal` 明显快于 `decimal`。不过在 Python 3.3 及更高版本中，`cdecimal` 已经以 `decimal` 这一名称并入标准库，因此不再有区别。
- PostgreSQL 的 `bytea` 在 Python 2 中转换为 `str`，在 Python 3 中转换为 `bytes`。在 Python 2 中，应将该字符串视为不带任何字符编码的字节序列。
- 所有其他数据类型，包括 PostgreSQL 字符串类型，都会转换为 Python `str`。在 Python 2 中，该字符串采用 PostgreSQL 服务器编码；在 Python 3 中，它与所有字符串一样，都是 Unicode 字符串。
- 对于非标量数据类型，请参见下文。

当 PL/Python 函数返回时，其返回值会按如下方式转换成该函数声明的 PostgreSQL 返回数据类型：

- 当 PostgreSQL 返回类型为 `boolean` 时，返回值会按照Python 规则进行真值判定。也就是说，0 和空字符串为假，但值得注意的是，`'f'` 为真。
- 如果 PostgreSQL 返回类型是 `bytea`，会先使用相应的 Python 内置函数，将返回值转换为字符串（Python 2）或 bytes（Python 3），再将结果转换为 `bytea`。
- 对于所有其他 PostgreSQL 返回类型，返回值会使用 Python 内置函数`str` 转换为字符串，然后将结果传给 PostgreSQL 数据类型的输入函数。（如果 Python 值是`float`，则会使用`repr` 内置函数而不是 `str` 来转换，以避免精度损失。）

  Python 2 中的字符串传给 PostgreSQL 时，必须采用 PostgreSQL 服务器编码。在当前服务器编码中无效的字符串会引发错误，但并非所有编码不匹配都能被检测到，因此处理不当仍可能产生乱码数据。Unicode 字符串会自动转换为正确的编码，因而使用它们可能更安全、更方便。在 Python 3 中，所有字符串都是 Unicode 字符串。
- 对于非标量数据类型，请参见下文。

请注意，即使声明的 PostgreSQL 返回类型与实际返回对象的 Python 数据类型在逻辑上并不匹配，也不会报错；无论如何该值都会被转换。

46.3.2. 空值、None

如果将 SQL 空值传递给函数，该参数值在 Python 中会表现为 `None`。例如，函数 `pymax` 的定义（见第 46.2 节）对空值输入会返回错误结果。可以在函数定义中添加 `STRICT`，让 PostgreSQL 采取更合理的行为：如果传入空值，就完全不调用函数，而是自动返回空值结果。也可以在函数体中检查空值输入：

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

如上所示，要从 PL/Python 函数返回 SQL 空值，只需返回 `None`。无论函数是否严格，都可以这样做。

46.3.3. 数组、列表

SQL 数组值会作为 Python 列表传入 PL/Python。要从 PL/Python 函数返回 SQL 数组值，请返回一个 Python 列表：

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return [1, 2, 3, 4, 5]
$$ LANGUAGE plpythonu;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

多维数组会作为嵌套的 Python 列表传入 PL/Python。例如，二维数组就是列表的列表。从 PL/Python 函数返回多维 SQL 数组时，每一层中的内部列表都必须大小相同。例如：

```
CREATE FUNCTION test_type_conversion_array_int4(x int4[]) RETURNS int4[] AS $$
plpy.info(x, type(x))
return x
$$ LANGUAGE plpythonu;

SELECT * FROM test_type_conversion_array_int4(ARRAY[[1,2,3],[4,5,6]]);
INFO:  ([[1, 2, 3], [4, 5, 6]], <type 'list'>)
 test_type_conversion_array_int4
---------------------------------
 {{1,2,3},{4,5,6}}
(1 row)
```

其他 Python 序列（如元组）也可被接受，这是为了与 PostgreSQL 9.6 及以下版本保持向后兼容，因为那时尚不支持多维数组。不过，它们始终被视为一维数组，因为它们与复合类型存在歧义。出于同样原因，当在多维数组中使用复合类型时，必须用元组而不是列表来表示。

请注意，在 Python 中，字符串也是序列，这可能会带来一些 Python 程序员熟悉但并不理想的效果：

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

46.3.4. 复合类型

复合类型参数会以 Python 映射的形式传给函数。映射中的元素名就是复合类型的属性名。如果传入行中的某个属性为空值，那么它在映射中的值就是None。例如：

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

有多种方法可以从 Python 函数返回行类型或复合类型。以下示例假定我们有：

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

复合结果可以按以下形式返回：

序列类型（元组或列表，但不能是集合，因为集合不可通过索引访问）

返回的序列对象，其项目数必须与复合结果类型的字段数相同。索引 0 的项目赋给复合类型的第一个字段，索引 1 的项目赋给第二个字段，依此类推。例如：

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return ( name, value )
  # 也可以使用列表： return [ name, value ]
$$ LANGUAGE plpythonu;
```

要为某一列返回 SQL 空值，请将 None 放在对应位置。

当返回复合类型数组时，不能将其表示为列表，因为这样无法区分 Python 列表表示的是复合类型还是另一个数组维度。

映射（字典）

结果类型中每一列的值，都使用列名作为键从映射中取得。例如：

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return { "name": name, "value": value }
$$ LANGUAGE plpythonu;
```

字典中多余的键值对会被忽略，缺少键则会被视为错误。要为某一列返回 SQL 空值，请插入 `None`，并以对应列名为键。

对象（提供方法`__getattr__`的任何对象）

其工作方式与映射相同。例如：

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
  return named_value(name, value)

  # 或者简写为
  class nv: pass
  nv.name = name
  nv.value = value
  return nv
$$ LANGUAGE plpythonu;
```

也支持带 `OUT` 参数的函数。例如：

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple();
```

过程的输出参数也以同样的方式传回。例如：

```
CREATE PROCEDURE python_triple(INOUT a integer, INOUT b integer) AS $$
return (a * 3, b * 3)
$$ LANGUAGE plpythonu;

CALL python_triple(5, 10);
```

46.3.5. 集合返回函数

PL/Python 函数也可以返回标量类型或复合类型的集合。实现方式有多种，因为返回的对象在内部会被转换成一个迭代器。以下示例假定我们有如下复合类型：

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

集合结果可以通过以下对象返回：

序列类型（元组、列表、集合）

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # 返回包含列表的元组，以列表表示复合类型
  # 其他组合方式也都可用
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

迭代器（任何提供 `__iter__` 和 `next` 方法的对象）

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def next (self):
      self.ndx += 1
      if self.ndx == len(self.who):
        raise StopIteration
      return ( self.how, self.who[self.ndx] )

  return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

生成器（yield）

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  for who in [ "World", "PostgreSQL", "PL/Python" ]:
    yield ( how, who )
$$ LANGUAGE plpythonu;
```

也支持带 OUT 参数的集合返回函数（使用 RETURNS SETOF record）。例如：

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer) RETURNS SETOF record
 AS $$
return [(1, 2)] * n
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple_setof(3);
```

46.4. 共享数据

全局字典 SD 可用于在同一函数的多次调用之间保存私有数据。全局字典 GD 则是可供一个会话中的所有 Python 函数使用的公共数据；使用时要小心。

每个函数在 Python 解释器中都有自己的执行环境，因此myfunc 中的全局数据和函数参数对 myfunc2 不可见。例外是前面提到的 GD 字典中的数据。

46.5. 匿名代码块

PL/Python 也支持通过 DO 语句调用的匿名代码块：

```
DO $$
    # PL/Python 代码
$$ LANGUAGE plpythonu;
```

匿名代码块不接受任何参数，而且无论返回什么值都会被丢弃。除此之外，它的行为与函数完全相同。

46.6. 触发器函数

当函数被用作触发器时，字典TD 包含与触发器相关的值：

```
TD["event"]
```

以字符串形式包含事件：INSERT、UPDATE、DELETE 或者 TRUNCATE。

```
TD["when"]
```

包含 BEFORE、AFTER 或者 INSTEAD OF 之一。

```
TD["level"]
```

包含 ROW 或者 STATEMENT。

```
TD["new"]
TD["old"]
```

对于行级触发器，这两个字段中的一个或两个会根据触发器事件包含相应的触发行。

```
TD["name"]
```

包含触发器名称。

```
TD["table_name"]
```

包含触发器所在表的名称。

```
TD["table_schema"]
```

包含触发器所在表的模式。

```
TD["relid"]
```

包含触发器所在表的 OID。

```
TD["args"]
```

如果 `CREATE TRIGGER` 命令包含参数，这些参数可在`TD["args"][0]` 到 `TD["args"][n-1]` 中取得。

如果 `TD["when"]` 是 `BEFORE` 或 `INSTEAD OF`，且`TD["level"]` 是 `ROW`，那么可以从 Python 函数返回`None` 或`"OK"` 来表示该行未被修改，返回`"SKIP"` 来中止该事件；如果 `TD["event"]` 是 `INSERT` 或 `UPDATE`，还可以返回`"MODIFY"` 来表示你已经修改了新行。否则返回值会被忽略。

46.7. 数据库访问

PL/Python 语言模块会自动导入一个名为 `plpy` 的 Python 模块。该模块中的函数和常量在 Python 代码中可以通过 `plpy.`*`foo`* 这样的形式访问。

46.7.1. 数据库访问函数

`plpy` 模块提供若干函数来执行数据库命令：

```
plpy.execute(query [, limit])
```

使用查询字符串和可选的行数限制参数调用 `plpy.execute` 会执行该查询，并把结果作为结果对象返回。

如果指定了 *`limit`* 且其大于零，则`plpy.execute` 最多检索 *`limit`* 行，就像查询中包含了 `LIMIT` 子句一样。省略 *`limit`* 或将其指定为零，则表示不限制行数。

结果对象的行为类似于列表或字典对象。可以通过行号和列名来访问结果对象。例如：

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

会从 my_table 中返回最多 5 行。如果 my_table 有一列名为 my_column，则可以这样访问它：

```
foo = rv[i]["my_column"]
```

返回的行数可以用内置的 len 函数获取。

结果对象还具有以下附加方法：

nrows()

返回该命令处理的行数。注意，这不一定与返回的行数相同。例如，UPDATE 命令会设置这个值，但不会返回任何行（除非使用RETURNING）。

status()

SPI_execute() 的返回值。

colnames()
coltypes()
coltypmods()

分别返回列名列表、列类型 OID 列表以及列的类型相关修饰符列表。

如果在来自不产生结果集的命令的结果对象上调用这些方法，就会引发异常，例如不带 RETURNING 的 UPDATE 或 DROP TABLE。但在包含零行的结果集上使用这些方法是没有问题的。

__str__()

还定义了标准的__str__方法，例如可以使用plpy.debug(rv) 来调试查询执行结果。

结果对象是可修改的。

注意，调用plpy.execute 会把整个结果集读入内存。只有在确信结果集相对较小时才应使用这个函数。获取大型结果时，如果不想承担过高的内存占用风险，应使用plpy.cursor 而不是 plpy.execute。

plpy.prepare(*query* [, *argtypes*])
plpy.execute(*plan* [, *arguments* [, *limit*]])

plpy.prepare 为查询准备执行计划。若查询中有参数引用，则它接受查询字符串和参数类型列表作为参数。例如：

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", ["text"])
```

text 是你要传给 $1 的变量类型。如果查询不需要任何参数，第二个参数可以省略。

准备好语句后，可以使用plpy.execute 函数的另一种调用形式来执行它：

```
rv = plpy.execute(plan, ["name"], 5)
```

将计划对象作为第一个参数传递（而不是查询字符串），并将要代入查询的值列表作为第二个参数传递。如果查询不需要任何参数，第二个参数可以省略。和前面一样，第三个参数仍然是可选的行数限制。

另外，也可以在计划对象上调用execute 方法：

```
rv = plan.execute(["name"], 5)
```

查询参数和结果行字段会按照第 46.3 节中所述，在 PostgreSQL 与 Python 数据类型之间进行转换。

当你使用 PL/Python 模块准备一个计划时，它会被自动保存。关于这意味着什么，请参阅 SPI 文档（第 47 章）。为了在函数调用之间有效利用这一点，需要使用持久存储字典SD 或 GD 之一（见第 46.4 节）。例如：

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if "plan" in SD:
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # 函数的其余部分
$$ LANGUAGE plpythonu;
```

```
plpy.cursor(query)
plpy.cursor(plan [, arguments])
```

plpy.cursor 函数接受与 plpy.execute 相同的参数（只是没有行数限制），并返回一个游标对象，使你可以分块处理大型结果集。与plpy.execute 一样，可以使用查询字符串，也可以使用带参数列表的计划对象；还可以把cursor 函数作为计划对象的方法来调用。

游标对象提供 fetch 方法，它接受一个整数参数并返回结果对象。每次调用fetch，返回对象都会包含下一批行，数量不会超过该参数值。一旦所有行都已取完，fetch 就会开始返回空结果对象。游标对象还提供迭代器接口[4]，每次产生一行，直到取完所有行。以这种方式获取的数据不会作为结果对象返回，而是以字典形式返回，每个字典对应一行结果。

[4] https://docs.python.org/library/stdtypes.html#iterator-types

下面示例展示了处理大表中数据的两种方式：

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
         odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1 <> 0", ["integer"])
rows = list(plpy.cursor(plan, [2]))  # 或者: = list(plan.cursor([2]))

return len(rows)
$$ LANGUAGE plpythonu;
```

游标会被自动释放。但如果想显式释放游标持有的全部资源，可使用`close` 方法。一旦关闭，就不能再从该游标中取数。

> **提示**
>
> 不要把 `plpy.cursor` 创建的对象与 Python 数据库 API 规范[5]定义的 DB-API 游标混淆。除了名字之外，它们毫无共同之处。

46.7.2. 捕获错误

访问数据库的函数可能会遇到错误，这会导致它们中止并抛出异常。`plpy.execute` 和 `plpy.prepare` 都可能抛出 `plpy.SPIError` 某个子类的实例，默认情况下这会终止函数。这个错误可以像其他 Python 异常一样，通过`try/except` 结构来处理。例如：

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:
        return "something went wrong"
    else:
        return "Joe added"
$$ LANGUAGE plpythonu;
```

[5] https://www.python.org/dev/peps/pep-0249/

实际抛出的异常类对应于引起错误的具体条件。可能的条件列表见表 A.1。模块 `plpy.spiexceptions` 为每一种 PostgreSQL 条件都定义了一个异常类，其名称派生自条件名。例如，`division_by_zero` 会变成 `DivisionByZero`，`unique_violation` 会变成 `UniqueViolation`，`fdw_error` 会变成 `FdwError`，等等。每个异常类都继承自 `SPIError`。这种区分让处理特定错误更容易，例如：

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["int", "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
except plpy.SPIError as e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpython3u;
```

请注意，由于`plpy.spiexceptions` 模块中的所有异常都继承自 `SPIError`，处理它的 `except` 子句会捕获任何数据库访问错误。

作为处理不同错误情况的另一种方法，你可以捕获`SPIError` 异常，并在`except` 块中通过查看异常对象的 `sqlstate` 属性来判断具体的错误条件。该属性是一个包含 “SQLSTATE” 错误代码的字符串值。这种方法大体上提供了相同的功能。

46.8. 显式子事务

如第 46.7.2 节中所述，从数据库访问引发的错误中恢复，可能会造成一种不理想的情况：在某个操作失败之前，其他一些操作已经成功，而在从该错误恢复后，数据却处于不一致状态。PL/Python 以显式子事务的形式为这个问题提供了解决方案。

46.8.1. 子事务上下文管理器

考虑以下实现两个账户之间转账的函数：

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

如果第二条 `UPDATE` 语句引发异常，此函数会报告错误，但第一条 `UPDATE` 的结果仍会提交。换句话说，资金会从 Joe 的账户中扣除，却不会转入 Mary 的账户。

为避免此类问题，可以将 `plpy.execute` 调用放在显式子事务中。`plpy` 模块提供了用于管理显式子事务的辅助对象，可通过 `plpy.subtransaction()` 函数创建。此函数创建的对象实现了上下文管理器接口[6]。使用显式子事务后，可以将函数改写为：

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

注意，仍需要使用 `try/catch`。否则，异常会传播到 Python 调用栈顶层，使整个函数因 PostgreSQL 错误而中止，从而不会向 `operations` 表插入任何行。子事务上下文管理器不会捕获错误，只保证在其作用域内执行的所有数据库操作以原子方式提交或回滚。任何异常退出都会使子事务块回滚，并不限于数据库访问错误。显式子事务块中抛出的普通 Python 异常，也会导致该子事务回滚。

46.8.2. 较早的 Python 版本

使用 `with` 关键字的上下文管理器语法，从 Python 2.6 起默认可用。如果 PL/Python 使用更早的 Python 版本，仍然可以使用显式子事务，只是没那么方便。可以通过便捷别名 `enter` 和 `exit`，调用子事务管理器的 `__enter__` 和 `__exit__` 函数。转账示例函数可以写成：

```
CREATE FUNCTION transfer_funds_old() RETURNS void AS $$
try:
    subxact = plpy.subtransaction()
    subxact.enter()
    try:
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
    except:
        import sys
        subxact.exit(*sys.exc_info())
        raise
    else:
        subxact.exit(None, None, None)
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"

plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

> **注意**
>
> 虽然 Python 2.5 已实现上下文管理器，但在该版本中使用 `with` 语法，需要使用 future 语句[7]。不过，由于实现细节的限制，PL/Python 函数中不能使用 future 语句。

[6] https://docs.python.org/library/stdtypes.html#context-manager-types
[7] https://docs.python.org/release/2.5/ref/future.html

46.9. 事务管理

如果过程是从顶层调用的，或者匿名代码块（DO 命令）是从顶层调用的，就可以控制事务。要提交当前事务，调用plpy.commit()。要回滚当前事务，调用plpy.rollback()。（注意，不能通过plpy.execute 或类似函数来执行 SQL 命令 COMMIT 或 ROLLBACK，必须使用这些函数。）事务结束后，会自动启动一个新事务，因此不需要单独提供这样的函数。

下面是一个示例：

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpythonu
AS $$
for i in range(0, 10):
    plpy.execute("INSERT INTO test1 (a) VALUES (%d)" % i)
    if i % 2 == 0:
        plpy.commit()
    else:
        plpy.rollback()
$$;

CALL transaction_test1();
```

当显式子事务处于活动状态时，不能结束事务。

46.10. 辅助函数

plpy 模块还提供以下函数：

```
plpy.debug(msg, **kwargs)
plpy.log(msg, **kwargs)
plpy.info(msg, **kwargs)
plpy.notice(msg, **kwargs)
plpy.warning(msg, **kwargs)
plpy.error(msg, **kwargs)
plpy.fatal(msg, **kwargs)
```

plpy.error 和 plpy.fatal 实际上会引发 Python 异常；如果这些异常未被捕获，就会传播到调用查询，导致当前事务或子事务中止。raise plpy.Error(*msg*) 和 raise plpy.Fatal(*msg*) 分别等价于调用 plpy.error(*msg*) 和 plpy.fatal(*msg*)，不过raise 形式不允许传递关键字参数。其他函数只会生成不同优先级的消息。某一优先级的消息是报告给客户端、写入服务器日志还是两者兼有，由配置变量log_min_messages 和 client_min_messages 控制。更多信息见第 19 章。

msg 参数作为位置参数给出。为了向后兼容，也可以给出多个位置参数。在这种情况下，这些位置参数所构成元组的字符串表示会成为报告给客户端的消息。

可接受以下仅限关键字的参数：

```
detail
hint
sqlstate
schema_name
table_name
column_name
datatype_name
constraint_name
```

作为仅限关键字参数传入的对象，其字符串表示会被用来丰富报告给客户端的消息。例如：

```
CREATE FUNCTION raise_custom_exception() RETURNS void AS $$
plpy.error("custom exception message",
           detail="some info about exception",
           hint="hint for users")
$$ LANGUAGE plpythonu;

=# SELECT raise_custom_exception();
ERROR:  plpy.Error: custom exception message
DETAIL:  some info about exception
HINT:  hint for users
CONTEXT:  Traceback (most recent call last):
  PL/Python function "raise_custom_exception", line 4, in <module>
    hint="hint for users")
PL/Python function "raise_custom_exception"
```

另一组辅助函数是 `plpy.quote_literal(`*`string`*`)`、`plpy.quote_nullable(`*`string`*`)` 以及 `plpy.quote_ident(`*`string`*`)`。它们等价于第 9.4 节中描述的内置加引号函数。在构造临时查询时，这些函数很有用。例 43.1 中动态 SQL 的一个 PL/Python 等价写法如下：

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

46.11. 环境变量

Python 解释器接受的某些环境变量也可用于影响 PL/Python 的行为。这些变量需要在 PostgreSQL 主服务器进程的环境中设置，例如在启动脚本中设置。可用的环境变量取决于 Python 版本；详情请参阅 Python 文档。在撰写本文时，假定 Python 版本足以支持这些变量，下列环境变量会影响 PL/Python：

- `PYTHONHOME`
- `PYTHONPATH`
- `PYTHONY2K`
- `PYTHONOPTIMIZE`
- `PYTHONDEBUG`

- PYTHONVERBOSE
- PYTHONCASEOK
- PYTHONDONTWRITEBYTECODE
- PYTHONIOENCODING
- PYTHONUSERBASE
- PYTHONHASHSEED

（这似乎是 Python 的实现细节，超出了 PL/Python 的控制范围：列在python 手册页中的某些环境变量只在命令行解释器中有效，而在嵌入式 Python 解释器中无效。）

第 47 章 服务器编程接口

服务器编程接口（SPI）使用户定义 C 函数的编写者能够在其函数或过程中运行 SQL 命令。SPI 是一组接口函数，用于简化对解析器、规划器和执行器的访问。SPI 还负责一部分内存管理工作。

> **注意**
>
> 可用的过程语言提供了多种从函数中执行 SQL 命令的方法。这些设施大多基于 SPI，因此本文档对这些语言的用户也会有帮助。

注意，如果通过 SPI 调用的某条命令失败，控制不会返回到你的 C 函数。相反，执行该 C 函数的事务或子事务会被回滚。（考虑到 SPI 函数大多记录了错误返回约定，这一点可能看起来有些意外。不过，这些约定只适用于在 SPI 函数自身内部检测到的错误。）如果在可能失败的 SPI 调用外围建立自己的子事务，则可以在出错后重新取得控制权。

SPI 函数在成功时返回非负结果（要么直接作为整数返回值，要么如后文所述存放在全局变量 `SPI_result` 中）。发生错误时，则返回负值或 `NULL`。

使用 SPI 的源代码文件必须包含头文件 `executor/spi.h`。

47.1. 接口函数

SPI_connect

SPI_connect, SPI_connect_ext — 将一个 C 函数连接到 SPI 管理器

大纲

```
int SPI_connect(void)


int SPI_connect_ext(int options)
```

描述

`SPI_connect` 会为某次 C 函数调用打开到 SPI 管理器的连接。如果要通过 SPI 执行命令，就必须调用此函数。不过，有些 SPI 辅助函数可以在未连接的 C 函数中调用。

`SPI_connect_ext` 执行相同的操作，但多了一个可传递选项标志的参数。目前可用的选项值如下：

`SPI_OPT_NONATOMIC`

将 SPI 连接设为非原子（nonatomic），这意味着允许进行事务控制调用（`SPI_commit`、`SPI_rollback`）。否则，调用这些函数会立即报错。

`SPI_connect()` 等价于 `SPI_connect_ext(0)`。

返回值

`SPI_OK_CONNECT`

成功时

`SPI_ERROR_CONNECT`

发生错误时

SPI_finish

SPI_finish — 将一个 C 函数与 SPI 管理器断开

大纲

```
int SPI_finish(void)
```

描述

`SPI_finish` 关闭到 SPI 管理器的现有连接。在完成当前这次 C 函数调用所需的 SPI 操作后，必须调用此函数。不过，如果通过 `elog(ERROR)` 中止了事务，就不必担心是否显式调用它；这种情况下 SPI 会自行清理。

返回值

`SPI_OK_FINISH`

如果正确地断开连接

`SPI_ERROR_UNCONNECTED`

如果从一个未连接的 C 函数中调用

SPI_execute

SPI_execute — 执行一个命令

大纲

```
int SPI_execute(const char * command, bool read_only, long count)
```

描述

SPI_execute 执行指定的 SQL 命令，并最多检索 *count* 行。如果 *read_only* 为 true，该命令必须是只读的，且执行开销会略有降低。

此函数只能从已连接的 C 函数中调用。

如果 *count* 为零，则该命令会针对其适用的所有行执行。如果 *count* 大于零，则最多检索 *count* 行；达到该计数时就会停止执行，这很像给查询增加了一个 LIMIT 子句。例如：

```
SPI_execute("SELECT * FROM foo", true, 5);
```

最多会从该表中检索 5 行。注意，这种限制只有在命令实际返回行时才有效。例如：

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

会忽略 *count* 参数，把 bar 中的所有行都插入进去。不过：

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

最多只会插入 5 行，因为取到第 5 行 RETURNING 结果后就会停止执行。

你可以在一个字符串中传递多条命令；SPI_execute 返回最后执行的那条命令的结果。*count* 限制会分别作用于每条命令（尽管实际返回的只有最后一条命令的结果）。该限制不适用于规则生成的任何隐藏命令。

当 *read_only* 为 false 时，SPI_execute 会递增命令计数器，并在执行字符串中的每条命令前计算新的快照。如果当前事务隔离级别是 SERIALIZABLE 或 REPEATABLE READ，这个快照实际上不会变化；但在 READ COMMITTED 模式下，更新快照会让每条命令都能看到其他会话中新近提交事务的结果。这对于修改数据库的命令获得一致行为至关重要。

当 *read_only* 为 true 时，SPI_execute 不会更新快照和命令计数器，并且只允许命令字符串中出现普通的 SELECT 命令。这些命令会使用外围查询先前建立的快照来执行。由于消除了每条命令的额外开销，这种执行模式比读写模式略快。它还允许构造真正稳定的函数：由于连续执行都会使用同一个快照，结果也就不会发生变化。

在同一个使用 SPI 的函数中混合只读命令和读写命令通常并不明智，因为只读查询看不到读写查询所做的数据库更新，这可能导致非常令人困惑的行为。

（最后一条）命令实际执行所处理的行数，会通过全局变量 SPI_processed 返回。如果函数返回值是 SPI_OK_SELECT、SPI_OK_INSERT_RETURNING、SPI_OK_DELETE_RETURNING、

SPI_OK_UPDATE_RETURNING，则可以通过全局指针 SPITupleTable *SPI_tuptable 访问结果行。有些工具命令（如 EXPLAIN）也会返回结果行集，此时 SPI_tuptable 同样会保存结果。另一些工具命令（COPY、CREATE TABLE AS）不返回行集，因此 SPI_tuptable 为 NULL，但它们依然会在 SPI_processed 中返回处理的行数。

结构 SPITupleTable 定义如下：

```
typedef struct
{
    MemoryContext tuptabcxt;    /* 结果表的内存上下文 */
    uint64      alloced;        /* 已分配的 vals 数量 */
    uint64      free;           /* 空闲的 vals 数量 */
    TupleDesc   tupdesc;        /* 行描述符 */
    HeapTuple  *vals;           /* 行 */
} SPITupleTable;
```

vals 是一个指向行的指针数组。（有效项数为 SPI_processed。）tupdesc 是一个行描述符，可以传给处理行的 SPI 函数。tuptabcxt，alloced 和 free 是内部字段，不供 SPI 调用者使用。

SPI_finish 会释放当前 C 函数调用期间分配的全部 SPITupleTable。如果某个结果表已经不再需要，也可以提前调用 SPI_freetuptable 释放它。

参数

const char * *command*

包含待执行命令的字符串

bool *read_only*

true 表示只读执行

long *count*

要返回的最大行数，或者用 0 表示不限制

返回值

如果命令执行成功，则返回下列（非负）值之一：

SPI_OK_SELECT

执行了 SELECT（但不是 SELECT INTO）

SPI_OK_SELINTO

执行了 SELECT INTO

SPI_OK_INSERT

执行了 INSERT

SPI_OK_DELETE

执行了 DELETE

SPI_OK_UPDATE

执行了 UPDATE

SPI_OK_INSERT_RETURNING

执行了 INSERT RETURNING

SPI_OK_DELETE_RETURNING

执行了 DELETE RETURNING

SPI_OK_UPDATE_RETURNING

执行了 UPDATE RETURNING

SPI_OK_UTILITY

执行了工具命令（例如 CREATE TABLE）

SPI_OK_REWRITTEN

命令被规则重写成了另一类命令（例如 UPDATE 变成了 INSERT）

出错时，返回以下负值之一：

SPI_ERROR_ARGUMENT

command 为 NULL，或者 *count* 小于 0

SPI_ERROR_COPY

尝试执行了 COPY TO stdout 或 COPY FROM stdin

SPI_ERROR_TRANSACTION

尝试执行了事务控制命令（BEGIN、COMMIT、ROLLBACK、SAVEPOINT、PREPARE TRANSACTION、COMMIT PREPARED、ROLLBACK PREPARED 及其各种变体）

SPI_ERROR_OPUNKNOWN

命令类型未知（理论上不应发生）

SPI_ERROR_UNCONNECTED

如果从一个未连接的 C 函数中调用

注解

所有 SPI 查询执行函数都会设置 SPI_processed 和 SPI_tuptable（只设置指针，而不更改结构体内容）。如果需要在后续调用之后继续访问 SPI_execute 或其他查询执行函数的结果表，请把这两个全局变量保存到 C 函数的局部变量中。

SPI_exec

SPI_exec — 执行一个读/写命令

大纲

```
int SPI_exec(const char * command, long count)
```

描述

`SPI_exec` 和 `SPI_execute` 相同，但后者的 *`read_only`* 参数的值总是取 `false`。

参数

`const char * command`

包含待执行命令的字符串

`long count`

要返回的最大行数，或者用 `0` 表示不限制

返回值

见 `SPI_execute`。

SPI_execute_with_args

SPI_execute_with_args — 执行使用外部参数的命令

大纲

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

描述

SPI_execute_with_args 执行一条可能包含外部提供参数引用的命令。命令文本中的参数引用写作 $*n*，而该调用会为每个这类符号指定数据类型和值。*read_only* 和 *count* 的含义与 SPI_execute 中相同。

相比 SPI_execute，这个例程的主要优点在于可将数据值插入命令而无需繁琐的引用和转义，因此能够显著降低 SQL 注入攻击的风险。

也可以通过先调用 SPI_prepare，再调用 SPI_execute_plan 达到类似效果。不过，使用本函数时，查询计划总会针对所提供的具体参数值进行定制。对于一次性查询执行，应优先选择本函数。如果同一条命令要用许多不同参数重复执行，则两种方式孰快取决于重新规划的代价与定制计划收益之间的权衡。

参数

const char * *command*

命令字符串

int *nargs*

输入参数的数量（$1、$2 等）

Oid * *argtypes*

一个长度为 *nargs* 的数组，包含参数数据类型的 OID

Datum * *values*

一个长度为 *nargs* 的数组，包含实际参数值

const char * *nulls*

一个长度为 *nargs* 的数组，用于描述哪些参数为空值

如果 *nulls* 为 NULL，则 SPI_execute_with_args 会假定没有参数为空值。否则，如果对应参数值非空值，则 *nulls* 数组中的对应项应为 ' '；如果对应参数值为空值，则对应项应为 'n'。（后一种情况下，对应 *values* 项中的实际值无关紧要。）注意，*nulls* 不是文本字符串，而只是一个数组，因此不需要 '\0' 终止符。

bool *read_only*

true 表示只读执行

long *count*

要返回的最大行数，或者用 0 表示不限制

返回值

返回值与 SPI_execute 相同。

成功时，SPI_processed 和 SPI_tuptable 的设置方式与 SPI_execute 相同。

SPI_prepare

SPI_prepare — 准备一个语句，但暂不执行

大纲

```
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

描述

SPI_prepare 为指定命令创建并返回一个预备语句，但并不执行该命令。该预备语句之后可以通过 SPI_execute_plan 重复执行。

当同一条或相似命令需要反复执行时，通常只做一次解析分析是有利的，而重用该命令的执行计划也可能进一步带来收益。SPI_prepare 会把命令字符串转换成一个封装了解析分析结果的预备语句。如果发现为每次执行都生成定制计划并无益处，该预备语句还可用来缓存执行计划。

预备命令可以通过在普通命令原本应写常量的位置写入参数（$1、$2 等）而得到泛化。这些参数的实际值会在调用 SPI_execute_plan 时指定。这样，预备语句就能适用于比无参形式更广泛的场景。

SPI_prepare 返回的语句只能在当前这次 C 函数调用中使用，因为 SPI_finish 会释放为此类语句分配的内存。不过，也可以使用 SPI_keepplan 或 SPI_saveplan 将该语句保存得更久。

参数

`const char * command`

命令字符串

`int nargs`

输入参数的数量（$1、$2 等）

`Oid * argtypes`

一个数组指针，它指向的数组包含参数的数据类型的 OID

返回值

SPI_prepare 返回一个非空指针，指向表示预备语句的不透明结构体 SPIPlan。发生错误时会返回 NULL，并将 SPI_result 设为 SPI_execute 所使用的那些错误码之一；但如果 *command* 为 NULL，或者 *nargs* 小于 0，或者 *nargs* 大于 0 且 *argtypes* 为 NULL，则会将其设置为 SPI_ERROR_ARGUMENT。

注解

如果没有定义参数，则在第一次使用 SPI_execute_plan 时会创建一个通用计划，并在之后的所有执行中继续使用它。如果存在参数，SPI_execute_plan 在最初几次使用时会根据提供的参数值生成定制计划。当同一个预备语句被使用足够多次之后，SPI_execute_plan

会构建一个通用计划；如果它的代价没有比定制计划高出太多，就会开始改用通用计划，而不是每次都重新规划。如果这种默认行为不合适，可以把 CURSOR_OPT_GENERIC_PLAN 或 CURSOR_OPT_CUSTOM_PLAN 标志传给 SPI_prepare_cursor，分别强制使用通用计划或定制计划。

尽管预备语句的主要目的在于避免重复进行解析分析和规划，但只要语句中使用的数据库对象自上次使用该预备语句以来发生了定义性（DDL）变更，PostgreSQL 就会在再次使用前强制重新分析并重新规划该语句。此外，如果search_path 的值在两次使用之间发生变化，该语句也会基于新的 search_path 重新解析。（后一种行为是从 PostgreSQL 9.3 开始引入的。）有关预备语句行为的更多信息，请参见 PREPARE。

此函数只能从已连接的 C 函数中调用。

SPIPlanPtr 在 spi.h 中被声明为指向不透明结构体类型的指针。尝试直接访问其内容并不明智，因为这会让你的代码在 PostgreSQL 后续版本中更容易失效。

SPIPlanPtr 这个名字多少带有历史色彩，因为该数据结构已不再必然包含执行计划。

SPI_prepare_cursor

SPI_prepare_cursor — 准备一个语句，但暂不执行

大纲

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
                              Oid * argtypes, int cursorOptions)
```

描述

SPI_prepare_cursor 与 SPI_prepare 相同，但额外允许指定规划器的“游标选项”参数。该参数是一个位掩码，其可取值对应于 nodes/parsenodes.h 中 DeclareCursorStmt 的 options 字段。SPI_prepare 总是把游标选项设为零。

参数

const char * *command*

命令字符串

int *nargs*

输入参数的数量（$1、$2 等）

Oid * *argtypes*

一个数组指针，它指向的数组包含参数的数据类型的 OID

int *cursorOptions*

整数形式的游标选项位掩码，零会导致默认行为

返回值

SPI_prepare_cursor 的返回约定与 SPI_prepare 相同。

注解

适合在 *cursorOptions* 中设置的位包括 CURSOR_OPT_SCROLL、CURSOR_OPT_NO_SCROLL、CURSOR_OPT_FAST_PLAN、CURSOR_OPT_GENERIC_PLAN 和 CURSOR_OPT_CUSTOM_PLAN。需要特别注意的是，CURSOR_OPT_HOLD 会被忽略。

SPI_prepare_params

SPI_prepare_params — 准备一个语句，但暂不执行

大纲

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

描述

SPI_prepare_params 为指定命令创建并返回一个预备语句，但不执行该命令。它相当于 SPI_prepare_cursor，只是额外允许调用者指定解析器钩子函数，以控制外部参数引用的解析。

参数

const char * *command*

命令字符串

ParserSetupHook *parserSetup*

解析器钩子设置函数

void * *parserSetupArg*

传递给 *parserSetup* 的透传参数

int *cursorOptions*

整数形式的游标选项位掩码，零会导致默认行为

返回值

SPI_prepare_params 的返回约定与 SPI_prepare 相同。

SPI_getargcount

SPI_getargcount — 返回由 SPI_prepare 准备的语句所需参数个数

大纲

```
int SPI_getargcount(SPIPlanPtr plan)
```

描述

SPI_getargcount 返回执行由 SPI_prepare 准备的语句所需的参数个数。

参数

SPIPlanPtr *plan*

预备语句（由 SPI_prepare 返回）

返回值

plan 所需的参数个数。如果 *plan* 为 NULL 或无效，则 SPI_result 会被设置为 SPI_ERROR_ARGUMENT，并返回 -1。

SPI_getargtypeid

SPI_getargtypeid — 返回由 SPI_prepare 准备的语句中某个参数的数据类型 OID

大纲

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

描述

SPI_getargtypeid 返回由 SPI_prepare 准备的语句中第 *argIndex* 个参数类型的 OID。第一个参数的索引为零。

参数

SPIPlanPtr *plan*

预备语句（由 SPI_prepare 返回）

int *argIndex*

参数的索引，从零开始

返回值

给定索引处参数的类型 OID。如果 *plan* 为 NULL 或无效，或者 *argIndex* 小于 0，或者不小于为 *plan* 声明的参数个数，则 SPI_result 会被设置为 SPI_ERROR_ARGUMENT，并返回 InvalidOid。

SPI_is_cursor_plan

SPI_is_cursor_plan — 如果由 SPI_prepare 准备的语句可用于 SPI_cursor_open 则返回 true

大纲

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

描述

如果由 SPI_prepare 准备的语句可以作为参数传给 SPI_cursor_open，则 SPI_is_cursor_plan 返回 true；否则返回 false。判定标准是 *plan* 必须表示单条命令，并且该命令会向调用者返回元组。例如，不包含 INTO 子句的 SELECT 是允许的，而 UPDATE 只有包含 RETURNING 子句时才允许。

参数

SPIPlanPtr *plan*

预备语句（由 SPI_prepare 返回）

返回值

返回 true 或 false，表示 *plan* 能否产生游标，同时将 SPI_result 设为零。如果无法确定答案（例如 *plan* 为 NULL 或无效，或者在未连接到 SPI 时调用），则会将 SPI_result 设为合适的错误码，并返回 false。

SPI_execute_plan

SPI_execute_plan — 执行由 SPI_prepare 准备好的语句

大纲

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                     bool read_only, long count)
```

描述

SPI_execute_plan 执行由 SPI_prepare 或其同类函数准备好的语句。*read_only* 和 *count* 的含义与 SPI_execute 中相同。

参数

SPIPlanPtr *plan*

预备语句（由 SPI_prepare 返回）

Datum * *values*

一个实际参数值的数组。必须和语句的参数数量等长。

const char * *nulls*

一个描述哪些参数为空值的数组。必须和语句的参数数量等长。

如果 *nulls* 为 NULL，则 SPI_execute_plan 会假定没有参数为空值。否则，如果对应参数值非空值，则 *nulls* 数组中的对应项应为 ' '；如果对应参数值为空值，则对应项应为 'n'。（后一种情况下，对应 *values* 项中的实际值无关紧要。）注意，*nulls* 不是文本字符串，而只是一个数组，因此不需要 '\0' 终止符。

bool *read_only*

true 表示只读执行

long *count*

要返回的最大行数，或者用 0 表示不限制

返回值

返回值和 SPI_execute 相同，还有下列额外可能的错误（负值）结果：

SPI_ERROR_ARGUMENT

如果 *plan* 为 NULL 或者非法，或者*count* 小于 0

SPI_ERROR_PARAM

如果 *values* 为 NULL，但 *plan* 在准备时带有参数

成功时，SPI_processed 和 SPI_tuptable 的设置方式与 SPI_execute 相同。

SPI_execute_plan_with_paramlist

SPI_execute_plan_with_paramlist — 执行由 SPI_prepare 准备好的语句

大纲

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

描述

SPI_execute_plan_with_paramlist 执行由 SPI_prepare 准备好的语句。它相当于 SPI_execute_plan，只是向查询传递参数值的方式不同。ParamListInfo 表示形式便于传递已经按这种格式存在的值，也支持通过 ParamListInfo 中指定的钩子函数使用动态参数集。

参数

SPIPlanPtr *plan*

预备语句（由 SPI_prepare 返回）

ParamListInfo *params*

包含参数类型和值的数据结构；没有参数时为 NULL

bool *read_only*

true 表示只读执行

long *count*

要返回的最大行数，或者用 0 表示不限制

返回值

返回值与 SPI_execute_plan 相同。

成功时，SPI_processed 和 SPI_tuptable 的设置方式与 SPI_execute_plan 相同。

SPI_execp

SPI_execp — 以读/写模式执行一个语句

大纲

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)
```

描述

SPI_execp 与 SPI_execute_plan 相同，只不过后者的 *read_only* 参数固定为 false。

参数

SPIPlanPtr *plan*

预备语句（由 SPI_prepare 返回）

Datum * *values*

一个实际参数值的数组。必须和语句的参数数量等长。

const char * *nulls*

一个描述哪些参数为空值的数组。必须和语句的参数数量等长。

如果 *nulls* 为 NULL，则 SPI_execp 会假定没有参数为空值。否则，如果对应参数值非空值，则 *nulls* 数组中的对应项应为 ' '；如果对应参数值为空值，则对应项应为 'n'。（后一种情况下，对应 *values* 项中的实际值无关紧要。）注意，*nulls* 不是文本字符串，而只是一个数组，因此不需要 '\0' 终止符。

long *count*

要返回的最大行数，或者用 0 表示不限制

返回值

见 SPI_execute_plan。

成功时，SPI_processed 和 SPI_tuptable 的设置方式与 SPI_execute 相同。

SPI_cursor_open

SPI_cursor_open — 使用由 SPI_prepare 创建的语句建立一个游标

大纲

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

描述

SPI_cursor_open 建立一个游标（在内部是一个 portal），该游标将执行由SPI_prepare 准备好的一个语句。参数具有和 SPI_execute_plan 的相应参数相同的含义。

使用游标而不是直接执行该语句有两个好处。首先，可以一次只取出少量结果行，从而避免对返回大量行的查询耗尽内存。其次，一个 portal 可以比当前的 C 函数存活更久（实际上，它可以一直存活到当前事务结束）。把 portal 的名称返回给该 C 函数的调用者，就提供了一种把行集作为结果返回的方法。

传入的参数数据会被复制到该游标的 portal 中，因此即使该游标仍然存在，也可以释放这些参数数据。

参数

`const char * name`

portal 的名称，或为NULL 以让系统选择名称

`SPIPlanPtr plan`

预备语句（由 SPI_prepare 返回）

`Datum * values`

一个实际参数值的数组。必须和语句的参数数量等长。

`const char * nulls`

一个描述哪些参数为空值的数组。必须和语句的参数数量等长。

如果 *nulls* 为 NULL，则 SPI_cursor_open 会假定没有参数为空值。否则，如果对应参数值非空值，则 *nulls* 数组中的对应项应为 ' '；如果对应参数值为空值，则对应项应为 'n'。（后一种情况下，对应 *values* 项中的实际值无关紧要。）注意，*nulls* 不是文本字符串，而只是一个数组，因此不需要 '\0' 终止符。

`bool read_only`

true 表示只读执行

返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定，任何错误都将通过`elog` 报告。

SPI_cursor_open_with_args

SPI_cursor_open_with_args — 使用一个查询和参数建立一个游标

大纲

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

描述

SPI_cursor_open_with_args 建立一个将执行指定查询的游标（在内部是一个 portal）。大部分参数具有和 SPI_prepare_cursor 和 SPI_cursor_open 中相应参数相同的含义。

对于一次性查询执行，应优先使用此函数，而不是先调用 SPI_prepare_cursor 再调用 SPI_cursor_open。如果同一条命令要用许多不同参数执行，哪种方法更快取决于重新规划的代价与定制计划收益之间的权衡。

传入的参数数据会被复制到该游标的 portal 中，因此即使该游标仍然存在，也可以释放这些参数数据。

参数

const char * *name*

portal 的名称，或为NULL 以让系统选择名称

const char * *command*

命令字符串

int *nargs*

输入参数的数量（$1、$2 等）

Oid * *argtypes*

一个长度为 *nargs* 的数组，包含参数数据类型的 OID

Datum * *values*

一个长度为 *nargs* 的数组，包含实际参数值

const char * *nulls*

一个长度为 *nargs* 的数组，用于描述哪些参数为空值

如果 *nulls* 为 NULL，则 SPI_cursor_open_with_args 会假定没有参数为空值。否则，如果对应参数值非空值，则 *nulls* 数组中的对应项应为 ' '；如果对应参数值为空

值，则对应项应为 'n'。（后一种情况下，对应 *values* 项中的实际值无关紧要。）注意，*nulls* 不是文本字符串，而只是一个数组，因此不需要 '\0' 终止符。

bool *read_only*

true 表示只读执行

int *cursorOptions*

整数形式的游标选项位掩码，零会导致默认行为

返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定，任何错误都将通过elog 报告。

SPI_cursor_open_with_paramlist

SPI_cursor_open_with_paramlist — 使用参数建立一个游标

大纲

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

描述

SPI_cursor_open_with_paramlist 建立一个游标（内部即一个 portal），用来执行由 SPI_prepare 准备好的语句。它相当于 SPI_cursor_open，只是向查询传递参数值的方式不同。ParamListInfo 表示形式便于传递已经按这种格式存在的值，也支持通过 ParamListInfo 中指定的钩子函数使用动态参数集。

传入的参数数据会被复制到该游标的 portal 中，因此即使该游标仍然存在，也可以释放这些参数数据。

参数

`const char * name`

portal 的名称，或为NULL 以让系统选择名称

`SPIPlanPtr plan`

预备语句（由 SPI_prepare 返回）

`ParamListInfo params`

包含参数类型和值的数据结构；没有参数时为 NULL

`bool read_only`

true 表示只读执行

返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定，任何错误都将通过elog 报告。

SPI_cursor_find

SPI_cursor_find — 用名称查找一个现有的游标

大纲

```
Portal SPI_cursor_find(const char * name)
```

描述

`SPI_cursor_find` 按名称查找现有的 portal。这主要用于解析由其他函数以文本形式返回的游标名。

参数

`const char * name`

该 portal 的名称

返回值

带有指定名称的 portal 的指针，如果没有找到就是 `NULL`

注解

注意，此函数可能返回一个不具备游标属性的 `Portal` 对象；例如，它可能并不返回元组。如果只是把这个 `Portal` 指针传给其他 SPI 函数，它们能够自行防范这类情况；但若要直接检查 `Portal`，则应当格外谨慎。

SPI_cursor_fetch

SPI_cursor_fetch — 从一个游标取出一些行

大纲

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

描述

SPI_cursor_fetch 从一个游标取得一些行。这等效于 SQL 命令 FETCH 的一个子集（更多功能见 SPI_scroll_cursor_fetch）。

参数

Portal *portal*

包含该游标的 portal

bool *forward*

为真表示向前获取，为假表示向后获取

long *count*

要取得的最大行数

返回值

成功时，SPI_processed 和 SPI_tuptable 的设置方式与 SPI_execute 相同。

注解

如果该游标的计划不是用 CURSOR_OPT_SCROLL 选项创建的，向后获取可能失败。

SPI_cursor_move

SPI_cursor_move — 移动一个游标

大纲

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

描述

SPI_cursor_move 跳过一个游标中的一些行。这等效于 SQL 命令 MOVE 的一个子集（更多的功能请见 SPI_scroll_cursor_move）。

参数

Portal *portal*

包含该游标的 portal

bool *forward*

为真表示前移，为假表示后移

long *count*

要移动的最大行数

注解

如果该游标的计划不是用 CURSOR_OPT_SCROLL 选项创建的，向后移动可能失败。

SPI_scroll_cursor_fetch

SPI_scroll_cursor_fetch — 从一个游标取出一些行

大纲

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,
                             long count)
```

描述

SPI_scroll_cursor_fetch 从一个游标中取出一些行。这等效于 SQL 命令 FETCH。

参数

Portal *portal*

包含该游标的 portal

FetchDirection *direction*

FETCH_FORWARD、FETCH_BACKWARD、FETCH_ABSOLUTE 或者 FETCH_RELATIVE 之一

long *count*

FETCH_FORWARD 或 FETCH_BACKWARD 方式中要取出的行数；FETCH_ABSOLUTE 方式中要取出的绝对行号；FETCH_RELATIVE 方式中要取出的相对行号

返回值

成功时，SPI_processed 和 SPI_tuptable 的设置方式与 SPI_execute 相同。

注解

参数 *direction* 和 *count* 的详细解释请见 SQL FETCH 命令。

如果该游标的计划不是用 CURSOR_OPT_SCROLL 选项创建的，除FETCH_FORWARD 之外的方向值可能失败。

SPI_scroll_cursor_move

SPI_scroll_cursor_move — 移动一个游标

大纲

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction,
                            long count)
```

描述

SPI_scroll_cursor_move 在一个游标中跳过一定数量的行。这等效于 SQL 命令 MOVE。

参数

Portal *portal*

包含该游标的 portal

FetchDirection *direction*

FETCH_FORWARD、FETCH_BACKWARD、FETCH_ABSOLUTE 或者 FETCH_RELATIVE 之一

long *count*

FETCH_FORWARD 或者 FETCH_BACKWARD 方式中要移动的行数；FETCH_ABSOLUTE 方式中要移动到的绝对行号；FETCH_RELATIVE 方式中要移动到的相对行号

返回值

成功时，SPI_processed 的设置方式与 SPI_execute 相同。SPI_tuptable 会被设置为 NULL，因为本函数不会返回任何行。

注解

参数 *direction* 和 *count* 的详细解释请见 SQL FETCH 命令。

如果该游标的计划不是用 CURSOR_OPT_SCROLL 选项创建的，除FETCH_FORWARD 之外的方向值可能失败。

SPI_cursor_close

SPI_cursor_close — 关闭一个游标

大纲

```
void SPI_cursor_close(Portal portal)
```

描述

`SPI_cursor_close` 关闭一个之前创建的游标并且释放它的 portal 存储。

所有打开的游标会在事务结束时自动被关闭。只有在希望尽快释放资源时，才需要调用 `SPI_cursor_close`。

参数

`Portal portal`

包含该游标的 portal

SPI_keepplan

SPI_keepplan — 保存一个预备语句

大纲

```
int SPI_keepplan(SPIPlanPtr plan)
```

描述

SPI_keepplan 保存传入的预备语句（由 SPI_prepare 准备），使其不会被 SPI_finish 或事务管理器释放。这让你能够在当前会话后续的 C 函数调用中重用该预备语句。

参数

SPIPlanPtr *plan*

要保存的预备语句

返回值

成功返回 0；如果 *plan* 为 NULL 或者无效则返回 SPI_ERROR_ARGUMENT

注解

这个函数通过调整指针（无需复制数据）的方式，将传入的预备语句重定位到永久存储中。如果之后需要删除它，可以对其使用 SPI_freeplan。

SPI_saveplan

SPI_saveplan — 保存一个预备语句

大纲

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

描述

SPI_saveplan 会把传入的预备语句（由 SPI_prepare 准备）复制到不会被 SPI_finish 或事务管理器释放的内存中，并返回该副本的指针。这让你能够在当前会话后续的 C 函数调用中重用预备语句。

参数

SPIPlanPtr *plan*

要保存的预备语句

返回值

复制后语句的指针；如果未成功则返回 NULL。错误时，SPI_result 会被这样设置：

SPI_ERROR_ARGUMENT

如果 *plan* 为 NULL 或无效

SPI_ERROR_UNCONNECTED

如果从一个未连接的 C 函数中调用

注解

原始传入的预备语句不会被释放，因此你可能需要对其调用 SPI_freeplan，以避免在 SPI_finish 之前发生内存泄露。

在大多数情况下，SPI_keepplan 比此函数更合适，因为它基本能达到相同效果，同时无需实际复制该预备语句的数据结构。

SPI_register_relation

SPI_register_relation — 让临时命名关系可在 SPI 查询中按名称使用

大纲

```
int SPI_register_relation(EphemeralNamedRelation enr)
```

描述

`SPI_register_relation` 使一个带有相关信息的临时命名关系对通过当前 SPI 连接规划和执行的查询可用。

参数

`EphemeralNamedRelation` *enr*

临时命名关系的注册表项

返回值

如果该命令的执行成功，则将返回下列（非负）值：

`SPI_OK_REL_REGISTER`

该关系已成功按名称注册

出错时，返回以下负值之一：

`SPI_ERROR_ARGUMENT`

enr 为 `NULL`，或其 `name` 字段为 `NULL`

`SPI_ERROR_UNCONNECTED`

如果从一个未连接的 C 函数中调用

`SPI_ERROR_REL_DUPLICATE`

enr 的 `name` 字段指定的名称已经为该连接注册

SPI_unregister_relation

SPI_unregister_relation — 从注册表中移除一个临时命名关系

大纲

```
int SPI_unregister_relation(const char * name)
```

描述

`SPI_unregister_relation` 从当前连接的注册表中移除一个临时命名关系。

参数

`const char * name`

该关系的注册表项名称

返回值

如果该命令的执行成功，则将返回下列（非负）值：

`SPI_OK_REL_UNREGISTER`

该 tuplestore 已成功从注册表中移除

出错时，返回以下负值之一：

`SPI_ERROR_ARGUMENT`

如果 *name* 为 `NULL`

`SPI_ERROR_UNCONNECTED`

如果从一个未连接的 C 函数中调用

`SPI_ERROR_REL_NOT_FOUND`

在当前连接的注册表中找不到 *name*

SPI_register_trigger_data

SPI_register_trigger_data — 让触发器捕获的临时数据可在 SPI 查询中使用

大纲

```
int SPI_register_trigger_data(TriggerData *tdata)
```

描述

SPI_register_trigger_data 使触发器捕获的所有临时关系都能在通过当前 SPI 连接规划和执行的查询中使用。目前，这指的是由带有 REFERENCING OLD/NEW TABLE AS ... 子句的 AFTER 触发器捕获的过渡表。PL 触发器处理函数应在建立 SPI 连接后调用此函数。

参数

TriggerData *tdata

作为 fcinfo->context 传给触发器处理函数的 TriggerData 对象

返回值

如果该命令的执行成功，则将返回下列（非负）值：

SPI_OK_TD_REGISTER

捕获到的触发器数据（如果有）已成功注册

出错时，返回以下负值之一：

SPI_ERROR_ARGUMENT

如果 *tdata* 为 NULL

SPI_ERROR_UNCONNECTED

如果从一个未连接的 C 函数中调用

SPI_ERROR_REL_DUPLICATE

任意触发器数据临时关系的名称已经为该连接注册过

47.2. 接口支持函数

这里介绍的函数提供了一个接口，用于从 SPI_execute 及其他 SPI 函数返回的结果集中提取信息。

本节介绍的所有函数都可以在已连接和未连接的 C 函数中使用。

SPI_fname

SPI_fname — 为指定的列号确定列名

大纲

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

描述

`SPI_fname` 返回指定列的列名的一个副本。（不再需要时，可以用 `pfree` 释放它。）

参数

`TupleDesc` *`rowdesc`*

输入行描述

`int` *`colnumber`*

列号（从 1 开始计）

返回值

列名；如果 *`colnumber`* 超出范围则返回 `NULL`。出错时，`SPI_result` 会被设置为 `SPI_ERROR_NOATTRIBUTE`。

SPI_fnumber

SPI_fnumber — 根据指定列名确定列号

大纲

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

描述

`SPI_fnumber` 返回指定列名对应的列号。

如果 *`colname`* 指向的是系统列（例如 `oid`），则会返回相应的负列号。调用者应当通过检查返回值是否恰好等于 `SPI_ERROR_NOATTRIBUTE` 来判断错误；除非本就要拒绝系统列，否则测试结果是否小于等于 0 并不正确。

参数

`TupleDesc` *`rowdesc`*

输入行描述

`const char *` *`colname`*

列名

返回值

列号（用户定义的列从 1 开始计），如果没有找到所提到的列名则返回 `SPI_ERROR_NOATTRIBUTE`。

SPI_getvalue

SPI_getvalue — 返回指定列的字符串值

大纲

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

描述

SPI_getvalue 返回指定列的值的字符串表示。

结果保存在由 palloc 分配的内存中返回。（不再需要时，可以用 pfree 释放该内存。）

参数

HeapTuple *row*

要检查的输入行

TupleDesc *rowdesc*

输入行描述

int *colnumber*

列号（从 1 开始计）

返回值

列值；如果该列为空值、*colnumber* 超出范围（SPI_result 被设置为 SPI_ERROR_NOATTRIBUTE）或者没有输出函数可用（SPI_result 被设置为 SPI_ERROR_NOOUTFUNC）则返回 NULL。

SPI_getbinval

SPI_getbinval — 返回指定列的二进制值

大纲

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
                    bool * isnull)
```

描述

`SPI_getbinval` 以内部形式（即 `Datum` 类型）返回指定列的值。

此函数不会为该 datum 分配新空间。对于传引用的数据类型，返回值将是指向传入行内部数据的指针。

参数

`HeapTuple` *`row`*

要检查的输入行

`TupleDesc` *`rowdesc`*

输入行描述

`int` *`colnumber`*

列号（从 1 开始计）

`bool *` *`isnull`*

列中是否为空值的标志

返回值

返回该列的二进制值。如果该列为空值，则 *`isnull`* 指向的变量会被设为真，否则设为假。

出错时，`SPI_result` 会被设置为 `SPI_ERROR_NOATTRIBUTE`。

SPI_gettype

SPI_gettype — 返回指定列的数据类型名称

大纲

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

描述

`SPI_gettype` 返回该指定列的数据类型名称的副本（当你不再需要该副本后，可以使用`pfree` 释放它）。

参数

`TupleDesc` *`rowdesc`*

输入行描述

`int` *`colnumber`*

列号（从 1 开始计）

返回值

指定列的数据类型名称；出错时返回 `NULL`。出错时，`SPI_result` 会被设置为 `SPI_ERROR_NOATTRIBUTE`。

SPI_gettypeid

SPI_gettypeid — 返回指定列的数据类型的 OID

大纲

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

描述

`SPI_gettypeid` 返回该指定列的数据类型的 OID。

参数

`TupleDesc rowdesc`

输入行描述

`int colnumber`

列号（从 1 开始计）

返回值

指定列数据类型的 OID；出错时返回 `InvalidOid`。出错时，`SPI_result` 会被设置为 `SPI_ERROR_NOATTRIBUTE`。

SPI_getrelname

SPI_getrelname — 返回指定关系的名称

大纲

```
char * SPI_getrelname(Relation rel)
```

描述

`SPI_getrelname` 返回该指定关系的名称的副本（当你不再需要该副本后，可以使用`pfree` 释放它）。

参数

`Relation rel`

输入关系

返回值

指定关系的名称。

SPI_getnspname

SPI_getnspname — 返回指定关系所属的名字空间

大纲

```
char * SPI_getnspname(Relation rel)
```

描述

`SPI_getnspname` 返回指定 `Relation` 所属名字空间名称的一个副本。这等价于该关系的模式名。用完该返回值后，应调用 `pfree` 释放它。

参数

`Relation rel`

输入关系

返回值

指定关系的名字空间的名称。

SPI_result_code_string

SPI_result_code_string — 返回错误码的字符串表示

大纲

```
const char * SPI_result_code_string(int code);
```

描述

SPI_result_code_string 返回各种 SPI 函数返回的，或存放在 SPI_result 中的结果码的字符串表示。

参数

int *code*

结果码

返回值

结果码的字符串表示。

47.3. 内存管理

PostgreSQL 在内存上下文中分配内存。内存上下文为管理那些在许多不同位置创建、且生命周期各不相同的内存分配提供了便捷手段。销毁某个上下文时，会释放其中分配的全部内存。因此，不必为了避免内存泄漏而跟踪每个单独对象；只需管理数量相对较少的上下文即可。palloc 及相关函数都从“当前”上下文中分配内存。

SPI_connect 创建新的内存上下文并将其设为当前上下文。SPI_finish 则恢复先前的当前上下文，并销毁由 SPI_connect 创建的上下文。这些操作可以确保在 C 函数内部所做的临时内存分配会在 C 函数退出时被回收，从而避免内存泄漏。

不过，如果 C 函数需要返回位于已分配内存中的对象（例如传引用数据类型的值），就不能使用 palloc 来分配这块内存，至少在连接到 SPI 的期间不能这么做。否则，该对象会在 SPI_finish 时被释放，C 函数也就无法可靠工作。解决办法是用 SPI_palloc 为返回对象分配内存。SPI_palloc 在“上层执行器上下文”中分配内存，也就是调用 SPI_connect 时的当前内存上下文；这正是从 C 函数返回值最合适的上下文。本节介绍的其他几个辅助函数同样会返回在上层执行器上下文中创建的对象。

调用 SPI_connect 时，会把该 C 函数的私有上下文（由 SPI_connect 创建）设为当前上下文。所有通过 palloc、repalloc 或 SPI 辅助函数分配的内存（本节另有说明的除外）都位于这个上下文中。当 C 函数通过 SPI_finish 与 SPI 管理器断开连接时，当前上下文会恢复为上层执行器上下文，而在该 C 函数内存上下文中分配的所有内存都会被释放，之后就不能再使用。

SPI_palloc

SPI_palloc — 在上层执行器上下文中分配内存

大纲

```
void * SPI_palloc(Size size)
```

描述

`SPI_palloc` 在上层执行器上下文中分配内存。

此函数只能在连接到 SPI 时使用，否则会抛出错误。

参数

`Size size`

要分配的存储空间大小（以字节计）

返回值

指向具有指定大小的新存储空间的指针

SPI_repalloc

SPI_repalloc — 在上层执行器上下文中重分配内存

大纲

```
void * SPI_repalloc(void * pointer, Size size)
```

描述

`SPI_repalloc` 改变之前用 `SPI_palloc` 分配的内存段的大小。

这个函数如今与普通的 `repalloc` 已无区别。保留它只是为了与现有代码保持向后兼容。

参数

`void * pointer`

指向要改变的现有存储空间的指针

`Size size`

要分配的存储空间大小（以字节计）

返回值

指向具有指定大小的新存储空间的指针，现有区域的内容会被复制到其中

SPI_pfree

SPI_pfree — 在上层执行器上下文中释放内存

大纲

```
void SPI_pfree(void * pointer)
```

描述

`SPI_pfree` 释放之前使用 `SPI_palloc` 或者 `SPI_repalloc` 分配的内存。

这个函数如今与普通的 `pfree` 已无区别。保留它只是为了与现有代码保持向后兼容。

参数

`void * pointer`

指向要释放的现有存储空间的指针

SPI_copytuple

SPI_copytuple — 在上层执行器上下文中创建一行的副本

大纲

```
HeapTuple SPI_copytuple(HeapTuple row)
```

描述

`SPI_copytuple` 在上层执行器上下文中创建一行的副本。它通常用于从触发器中返回修改后的行。对于声明为返回复合类型的函数，应改用 `SPI_returntuple`。

此函数只能在连接到 SPI 时使用。否则，它会返回 NULL，并将 `SPI_result` 设为 `SPI_ERROR_UNCONNECTED`。

参数

`HeapTuple` *`row`*

要复制的行

返回值

复制后的行；出错时返回 `NULL`（错误指示见 `SPI_result`）

SPI_returntuple

SPI_returntuple — 准备把一个元组返回为一个 Datum

大纲

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

描述

`SPI_returntuple` 在上层执行器上下文中创建一行的副本，并以行类型 `Datum` 的形式返回。返回的指针只需在返回前用 `PointerGetDatum` 转换为 `Datum` 即可。

此函数只能在连接到 SPI 时使用。否则，它会返回 NULL，并将 `SPI_result` 设为 `SPI_ERROR_UNCONNECTED`。

请注意，它应用于声明为返回复合类型的函数，而不用于触发器；触发器中返回修改后的行应使用 `SPI_copytuple`。

参数

`HeapTuple` *`row`*

要复制的行

`TupleDesc` *`rowdesc`*

行描述符（若要获得最佳缓存效果，每次都传入同一个描述符）

返回值

指向复制后行的 `HeapTupleHeader`；出错时返回 `NULL`（错误指示见 `SPI_result`）

SPI_modifytuple

SPI_modifytuple — 通过替换给定行的选定字段来创建新行

大纲

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

描述

SPI_modifytuple 通过为选定列替换新值、并在其他位置复制原行的列来创建一行新记录。输入行本身不会被修改。新行会在上层执行器上下文中返回。

此函数只能在连接到 SPI 时使用。否则，它会返回 NULL，并将 SPI_result 设为 SPI_ERROR_UNCONNECTED。

参数

Relation *rel*

仅用作该行的行描述符来源。（传递关系而不是行描述符是个设计缺陷。）

HeapTuple *row*

要被修改的行

int *ncols*

要被修改的列数

int * *colnum*

一个长度为 *ncols* 的数组，包含了要被修改的列号（列号从 1 开始）

Datum * *values*

一个长度为 *ncols* 的数组，包含了指定列的新值

const char * *nulls*

一个长度为 *ncols* 的数组，描述哪些新值为空值

如果 *nulls* 为 NULL，则 SPI_modifytuple 会假定没有新值为空值。否则，如果对应新值非空值，则 *nulls* 数组中的对应项应为 ' '；如果对应新值为空值，则对应项应为 'n'。（后一种情况下，对应 *values* 项中的实际值无关紧要。）注意，*nulls* 不是文本字符串，而只是一个数组，因此不需要 '\0' 终止符。

返回值

修改后的新行，分配在上层执行器上下文中；出错时返回 NULL（错误指示见 SPI_result）

出错时，SPI_result 被设置如下：

SPI_ERROR_ARGUMENT

如果 *rel* 为 NULL，或者 *row* 为 NULL，或者*ncols* 小于等于 0，或者*colnum* 为 NULL，或者*values* 为 NULL。

SPI_ERROR_NOATTRIBUTE

如果 *colnum* 包含一个无效的列号（小于等于 0 或者大于 *row* 中的列数）。

SPI_ERROR_UNCONNECTED

SPI 未激活

SPI_freetuple

SPI_freetuple — 释放一个在上层执行器上下文中分配的行

大纲

```
void SPI_freetuple(HeapTuple row)
```

描述

`SPI_freetuple` 释放之前在上层执行器上下文中分配的一行。

这个函数如今与普通的 `heap_freetuple` 已无区别。保留它只是为了与现有代码保持向后兼容。

参数

`HeapTuple` *`row`*

要释放的行

SPI_freetuptable

SPI_freetuptable — 释放由 `SPI_execute` 或类似函数创建的结果行集

大纲

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

描述

`SPI_freetuptable` 释放由先前的 SPI 命令执行函数（例如 `SPI_execute`）创建的结果行集。因此，调用它时经常会直接把全局变量 `SPI_tuptable` 作为参数。

如果某个使用 SPI 的 C 函数需要执行多条命令，并且不想一直保留前面命令的结果，那么这个函数就很有用。注意，任何尚未释放的结果行集都会在 `SPI_finish` 时被释放。此外，如果在使用 SPI 的 C 函数执行过程中开启了一个子事务而后又中止，SPI 会自动释放该子事务运行期间创建的所有结果行集。

从 PostgreSQL 9.3 开始，`SPI_freetuptable` 内置了保护逻辑，以避免对同一个结果行集重复发出删除请求。在更早的版本中，重复删除会导致崩溃。

参数

`SPITupleTable * tuptable`

要释放的结果行集指针，或传入 NULL 表示不执行任何操作

SPI_freeplan

SPI_freeplan — 释放一个之前保存的预备语句

大纲

```
int SPI_freeplan(SPIPlanPtr plan)
```

描述

`SPI_freeplan` 释放一个之前由 `SPI_prepare` 返回的或者由 `SPI_keepplan`、`SPI_saveplan` 保存的预备语句。

参数

`SPIPlanPtr plan`

要释放的语句的指针

返回值

成功返回 0；如果 *plan* 为 `NULL` 或者无效则返回 `SPI_ERROR_ARGUMENT`

47.4. 事务管理

不能通过 `SPI_execute` 这类 SPI 函数运行 `COMMIT`、`ROLLBACK` 之类的事务控制命令。不过，也有单独的接口函数允许通过 SPI 进行事务控制。

如果不考虑调用上下文，那么在任意用户定义的、可从 SQL 调用的函数中开始和结束事务，通常既不安全也不合理。例如，如果某个函数处在某条 SQL 命令的复杂 SQL 表达式中间，那么在该函数内部设置事务边界，很可能会导致隐蔽的内部错误甚至崩溃。这里介绍的接口函数，主要是供过程语言实现使用，以支持由 `CALL` 命令调用的 SQL 层过程进行事务管理，并且会把 `CALL` 调用的上下文考虑在内。用 C 实现、使用 SPI 的过程也可以实现同样的逻辑，但具体细节超出了本文档的范围。

SPI_commit

SPI_commit — 提交当前事务

大纲

```
void SPI_commit(void)
```

描述

`SPI_commit` 提交当前事务。它大致相当于执行 SQL 命令 `COMMIT`。事务提交之后，会自动以默认事务特性启动一个新事务，使调用者能够继续使用 SPI 设施。如果在提交过程中发生失败，则当前事务会改为回滚，并启动一个新事务，随后再按通常方式抛出错误。

只有当对 `SPI_connect_ext` 的调用把 SPI 连接设置为非原子时，此函数才能执行。

SPI_rollback

SPI_rollback — 中止当前事务

大纲

```
void SPI_rollback(void)
```

描述

`SPI_rollback` 回滚当前事务。它大致相当于执行 SQL 命令 `ROLLBACK`。事务回滚后，会自动以默认事务特性启动一个新事务，使调用者能够继续使用 SPI 设施。

只有当对 `SPI_connect_ext` 的调用把 SPI 连接设置为非原子时，此函数才能执行。

SPI_start_transaction

SPI_start_transaction — 废弃函数

大纲

```
void SPI_start_transaction(void)
```

描述

SPI_start_transaction 什么也不做，它存在的唯一目的是兼容较早版本的 PostgreSQL 代码。过去在调用 SPI_commit 或 SPI_rollback 之后需要调用它，而现在这两个函数都会自动开启新事务。

47.5. 数据更改的可见性

下列规则决定了使用 SPI 的函数（以及其他任何 C 函数）中数据更改的可见性：

- 在某条 SQL 命令执行期间，该命令所做的任何数据更改对该命令本身都是不可见的。例如，在

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  中，被插入的行对SELECT 部分不可见。

- 某条命令 C 所做的更改，对所有在 C 之后启动的命令都可见，无论这些命令是在 C 内部（即 C 执行期间）启动，还是在 C 完成之后启动。

- 在由 SQL 命令调用的函数中（无论是普通函数还是触发器）通过 SPI 执行的命令，究竟遵循上述哪条规则，取决于传给 SPI 的读写标志。只读模式下执行的命令遵循第一条规则：它们看不到调用它们的命令所做的更改。读写模式下执行的命令遵循第二条规则：它们能够看到截至当前为止的所有更改。

- 所有标准过程语言都会根据函数的易变性属性设置 SPI 的读写模式。STABLE 和 IMMUTABLE 函数中的命令会以只读模式执行，而 VOLATILE 函数中的命令则以读写模式执行。虽然 C 函数作者能够违背这一约定，但通常并不是好主意。

下一节给出了一个展示这些规则如何应用的示例。

47.6. 示例

本节给出一个非常简单的 SPI 用法示例。C 函数 execq 以一条 SQL 命令作为第一个参数、以行数计数作为第二个参数，使用 SPI_exec 执行该命令，并返回该命令实际处理的行数。更复杂的 SPI 示例可以在源码树的 src/test/regress/regress.c 以及 spi 模块中找到。

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

PG_MODULE_MAGIC;
```

```
PG_FUNCTION_INFO_V1(execq);

Datum
execq(PG_FUNCTION_ARGS)
{
    char *command;
    int cnt;
    int ret;
    uint64 proc;

    /* 把给定的文本对象转换成一个 C 字符串 */
    command = text_to_cstring(PG_GETARG_TEXT_PP(0));
    cnt = PG_GETARG_INT32(1);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;

    /*
     * 如果取出了一些行，通过 elog(INFO) 打印它们。
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        uint64 j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];
            int i;

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    PG_RETURN_INT64(proc);
}
```

将该函数编译成共享库之后（详见第 38.10.5 节），可以这样声明它：

```
CREATE FUNCTION execq(text, integer) RETURNS int8
    AS 'filename'
    LANGUAGE C STRICT;
```

下面是一个示例会话：

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
```

```
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- 由 execq 插入
INFO:  EXECQ:  1    -- 由 execq 返回并被外层 INSERT 插入

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a RETURNING *', 1);
INFO:  EXECQ:  2    -- 0 + 2, 然后执行被 count 停止
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2

 execq
-------
     3              -- 10 只是最大值, 3 才是真正的行数
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 10 FROM a', 1);
 execq
-------
     3              -- 所有行都被处理; count 不会停止它, 因为没有返回任何行
(1 row)

=> SELECT * FROM a;
 x
----
  0
  1
  2
 10
 11
 12
(6 rows)

=> DELETE FROM a;
DELETE 6
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1                  -- 0 (a 中没有行) + 1
```

```
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- 1 (a 中原有一行) + 1
(2 rows)

-- 下面演示数据更改可见性规则。
-- execq 被调用两次，每次看到的行数都不同：

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1    -- 第一次 execq 的结果
INFO:  EXECQ:  2
INFO:  EXECQ:  1    -- 第二次 execq 的结果
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 行 * 1 (第一行中的 x)
 6                  -- 3 行 (前面的 2 行加上刚插入的 1 行) * 2 (第二行中的 x)
(4 rows)
```

第 48 章 后台工作进程

PostgreSQL 可以扩展为在独立进程中运行用户提供的代码。这类进程由 postgres 启动、停止并监控，因此它们的生命周期能够与服务器状态紧密关联。这些进程可以选择附着到PostgreSQL的共享内存区域以及在内部连接到数据库；它们还可以像普通的客户端连接服务器进程一样串行运行多个事务。此外，通过链接libpq，它们还可以连接到服务器，并表现得像普通客户端应用程序一样。

> **警告**
>
> 使用后台工作进程存在相当大的健壮性和安全风险，因为它们是用C 语言编写的，对数据具有不受限制的访问权限。希望启用包含后台工作进程的模块的管理员必须极其谨慎。只应允许经过仔细审计的模块运行后台工作进程。

可以通过在 shared_preload_libraries 中包含模块名，在PostgreSQL 启动时初始化后台工作进程。希望运行后台工作进程的模块可以在其 _PG_init() 函数中调用 RegisterBackgroundWorker(BackgroundWorker *worker) 来注册它。系统启动并运行后，也可以通过调用RegisterDynamicBackgroundWorker(BackgroundWorker *worker, BackgroundWorkerHandle **handle) 来启动后台工作进程。与只能在 postmaster 进程内调用的 RegisterBackgroundWorker 不同，RegisterDynamicBackgroundWorker 必须由普通后端或另一个后台工作进程调用。

BackgroundWorker 结构体定义如下：

```
typedef void (*bgworker_main_type)(Datum main_arg);
typedef struct BackgroundWorker
{
    char        bgw_name[BGW_MAXLEN];
    char        bgw_type[BGW_MAXLEN];
    int         bgw_flags;
    BgWorkerStartTime bgw_start_time;
    int         bgw_restart_time;       /* in seconds, or BGW_NEVER_RESTART */
    char        bgw_library_name[BGW_MAXLEN];
    char        bgw_function_name[BGW_MAXLEN];
    Datum       bgw_main_arg;
    char        bgw_extra[BGW_EXTRALEN];
    int         bgw_notify_pid;
} BackgroundWorker;
```

bgw_name 和 bgw_type 是用于日志消息、进程列表及类似场景的字符串。对于同一类型的所有后台工作进程，bgw_type 都应相同，这样例如就可以在进程列表中将这类工作进程归为一组。另一方面，bgw_name 可以包含该特定进程的额外信息。（通常，bgw_name 中的字符串会以某种方式包含该类型，但这并非严格要求。）

bgw_flags 是一个按位或组合的位掩码，用于指示模块所需的能力。可能的值有：

BGWORKER_SHMEM_ACCESS

请求共享内存访问。没有共享内存访问权的工作进程不能访问 PostgreSQL 的任何共享数据结构，如重量级锁或轻量级锁、共享缓冲区，以及工作进程自身可能希望创建和使用的任何自定义数据结构。

BGWORKER_BACKEND_DATABASE_CONNECTION

请求建立数据库连接的能力，以便之后运行事务和查询。使用BGWORKER_BACKEND_DATABASE_CONNECTION 连接数据库的后台工作进程，还必须使用 BGWORKER_SHMEM_ACCESS 附着到共享内存，否则工作进程启动将失败。

bgw_start_time 表示 postgres 应在服务器的哪个状态下启动该进程；它可以是 BgWorkerStart_PostmasterStart（在 postgres 自身完成初始化后立即启动；请求此值的进程不能建立数据库连接）、BgWorkerStart_ConsistentState（在热备达到一致状态后立即启动，允许进程连接数据库并运行只读查询），或BgWorkerStart_RecoveryFinished（在系统进入正常读写状态后立即启动）。请注意，在非热备服务器中，后两个值是等价的。还要注意，此设置只表示何时启动进程；当到达其他状态时，这些进程不会停止。

bgw_restart_time 是以秒计的时间间隔，表示在进程崩溃时，postgres 在重新启动该进程之前应等待多久。它可以是任意正值，也可以是BGW_NEVER_RESTART，表示在进程崩溃时不重新启动该进程。

bgw_library_name 是一个库的名称，后台工作进程的初始入口点将在该库中查找。该库会由工作进程动态加载，而bgw_function_name 将用于标识要调用的函数。如果加载的是核心代码中的函数，此字段必须设置为 "postgres"。

bgw_function_name 是动态加载库中的一个函数名，用作新后台工作进程的初始入口点。

bgw_main_arg 是传给后台工作进程主函数的 Datum 参数。该主函数应接受一个 Datum 类型的参数，并返回void。bgw_main_arg 会作为该参数传入。此外，全局变量 MyBgworkerEntry 指向注册时传入的 BackgroundWorker 结构体的一份副本；检查该结构体可能会对工作进程有所帮助。

在 Windows（以及任何定义了 EXEC_BACKEND 的地方）上，或者在动态后台工作进程中，以引用方式传递Datum 并不安全，只能按值传递。如果需要传递参数，最安全的方式是传递一个 int32 或其他较小的值，并将其用作共享内存中分配的数组的索引。如果传递的是cstring 或 text 这样的值，那么该指针在新的后台工作进程中将无效。

bgw_extra 可以包含要传递给后台工作进程的额外数据。与 bgw_main_arg 不同，这些数据不会作为参数传递给工作进程的主函数，但可以如上所述通过MyBgworkerEntry 访问。

bgw_notify_pid 是一个 PostgreSQL 后端进程的 PID，当后台工作进程启动或退出时，postmaster 应向该后端进程发送SIGUSR1。对于在 postmaster 启动时注册的工作进程，或者注册该工作进程的后端不希望等待其启动时，它应为 0。否则，它应初始化为MyProcPid。

进程一旦开始运行，就可以通过调用 BackgroundWorkerInitializeConnection(*char *dbname*, *char *username*, *uint32 flags*) 或 BackgroundWorkerInitializeConnectionByOid(*Oid dboid*, *Oid useroid*, *uint32 flags*) 连接到数据库。这使得该进程能够通过 SPI 接口运行事务和查询。如果 dbname 为 NULL，或dboid 为 InvalidOid，则该会话不会连接到任何特定数据库，但仍可访问共享系统目录。如果 username 为 NULL，或useroid 为 InvalidOid，则该进程将以initdb 期间创建的超级用户身份运行。如果将 BGWORKER_BYPASS_ALLOWCONN 指定为 flags，则可以绕过对不允许用户连接的数据库的连接限制。后台工作进程只能调用这两个函数中的一个，而且只能调用一次。不能切换数据库。

当控制流到达后台工作进程的主函数时，信号起初处于阻塞状态，必须由主函数解除阻塞；这样做是为了在必要时允许该进程自定义其信号处理器。在新进程中，可以调用BackgroundWorkerUnblockSignals 解除信号阻塞，并调用BackgroundWorkerBlockSignals 重新阻塞信号。

如果后台工作进程的 bgw_restart_time 配置为 BGW_NEVER_RESTART，或者它以退出码 0 退出，或者被TerminateBackgroundWorker 终止，则 postmaster 会在其退出时自动注销它。否则，它会在bgw_restart_time 配置的时间段之后重新启动；如果 postmaster 因某个后端失败而重新初始化集簇，则会立即重新启动它。只需暂时挂起执行的后端应使用可中断的睡眠而不是退出；这可以通过调用WaitLatch() 来实现。调用该函数时，要确保设置了 WL_POSTMASTER_DEATH 标志，并检查返回码，以便在postgres 自身终止的紧急情况下及时退出。

当使用 RegisterDynamicBackgroundWorker 函数注册后台工作进程时，执行注册的后端可以获取有关该工作进程状态的信息。希望这样做的后端应将BackgroundWorkerHandle * 的地址作为第二个参数传递给 RegisterDynamicBackgroundWorker。如果工作进程成功注册，该指针将初始化为一个不透明句柄，随后可以将其传递给GetBackgroundWorker Pid(*BackgroundWorkerHandle **, *pid_t **) 或 TerminateBackgroundWorker(*BackgroundWorkerHandle **)。GetBackgroundWorkerPid 可用于轮询工作进程的状态：返回值为 BGWH_NOT_YET_STARTED 表示该工作进程尚未由 postmaster 启动；BGWH_STOPPED 表示它已经启动但不再运行；BGWH_STARTED 表示它当前正在运行。在最后一种情况下，PID 也会通过第二个参数返回。TerminateBackgroundWorker 会使 postmaster 在该工作进程运行时向其发送 SIGTERM，并在其不再运行后尽快将其注销。

在某些情况下，注册后台工作进程的进程可能希望等待该工作进程启动。这可以通过将 bgw_notify_pid 初始化为 MyProcPid，然后把注册时获得的BackgroundWorkerHandle * 传递给 WaitForBackgroundWorkerStartup(*BackgroundWorkerHandle *handle*, *pid_t **) 函数来实现。该函数会阻塞，直到 postmaster 已尝试启动该后台工作进程，或者直到 postmaster 死亡。如果后台工作进程正在运行，返回值将是BGWH_STARTED，其 PID 会被写入所提供的地址。否则，返回值将是BGWH_STOPPED 或 BGWH_POSTMASTER_DIED。

进程也可以通过使用 WaitForBackgroundWorkerShutdown(*BackgroundWorkerHandle *handle*) 函数，并传入注册时获得的BackgroundWorkerHandle *，来等待后台工作进程关闭。该函数会阻塞，直到后台工作进程退出，或者 postmaster 死亡。当后台工作进程退出时，返回值为BGWH_STOPPED；如果 postmaster 死亡，则返回BGWH_POSTMASTER_DIED。

如果后台工作进程通过服务器编程接口（SPI）使用 NOTIFY 命令发送异步通知，那么应在提交包含该命令的事务后，显式调用 ProcessCompletedNotifies，以便送达通知。如果后台工作进程通过 SPI 使用 LISTEN 注册接收异步通知，该工作进程会将这些通知写入日志，但无法以编程方式拦截并响应这些通知。

src/test/modules/worker_spi 模块包含一个可工作的示例，展示了一些有用的技巧。

已注册后台工作进程的最大数量受 max_worker_processes 限制。

第 49 章 逻辑解码

PostgreSQL 提供了基础设施，可将通过 SQL 执行的修改流式发送给外部消费者。此功能可用于多种目的，包括复制方案和审计。

更改通过由逻辑复制槽标识的流发送出去。

这些更改以何种格式流式传输，由所使用的输出插件决定。PostgreSQL 发行版提供了一个示例插件。还可以编写额外的插件，在不修改任何核心代码的情况下扩展可用格式的选择。每个输出插件都可以访问由 INSERT 产生的每一个新行，以及由 UPDATE 创建的新行版本。对于 UPDATE 和 DELETE，旧行版本是否可用取决于所配置的复制标识（见 REPLICA IDENTITY）。

更改既可以通过流复制协议消费（见第 53.4 节和第 49.3 节），也可以通过 SQL 调用函数来消费（见第 49.4 节）。还可以在不修改核心代码的情况下编写其他消费复制槽输出的方法（见第 49.7 节）。

49.1. 逻辑解码示例

下面的示例演示如何使用 SQL 接口控制逻辑解码。

在使用逻辑解码之前，必须将wal_level 设置为 logical，并将max_replication_slots 至少设置为 1。然后，应当以超级用户身份连接到目标数据库（下面的示例中为 postgres）。

```
postgres=# -- Create a slot named 'regression_slot' using the output plugin 'test_decoding'
postgres=# SELECT * FROM pg_create_logical_replication_slot('regression_slot', 'test_decoding')
;
    slot_name    |    lsn
-----------------+-----------
 regression_slot | 0/16B1970
(1 row)

postgres=# SELECT slot_name, plugin, slot_type, database, active, restart_lsn, confirmed_flush_
lsn FROM pg_replication_slots;
    slot_name    |    plugin     | slot_type | database | active | restart_lsn | confirmed_
flush_lsn
-----------------+---------------+-----------+----------+--------+-------------+---------------
--
 regression_slot | test_decoding | logical   | postgres | f      | 0/16A4408   | 0/16A4440
(1 row)

postgres=# -- There are no changes to see yet
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

postgres=# CREATE TABLE data(id serial primary key, data text);
CREATE TABLE

postgres=# -- DDL isn't replicated, so all you'll see is the transaction
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |     data
-----------+-------+--------------
 0/BA2DA58 | 10297 | BEGIN 10297
 0/BA5A5A0 | 10297 | COMMIT 10297
(2 rows)

postgres=# -- Once changes are read, they're consumed and not emitted
postgres=# -- in a subsequent call:
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)
```

```
postgres=# BEGIN;
postgres=# INSERT INTO data(data) VALUES('1');
postgres=# INSERT INTO data(data) VALUES('2');
postgres=# COMMIT;

postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+--------------------------------------------------------
 0/BA5A688 | 10298 | BEGIN 10298
 0/BA5A6F0 | 10298 | table public.data: INSERT: id[integer]:1 data[text]:'1'
 0/BA5A7F8 | 10298 | table public.data: INSERT: id[integer]:2 data[text]:'2'
 0/BA5A8A8 | 10298 | COMMIT 10298
(4 rows)

postgres=# INSERT INTO data(data) VALUES('3');

postgres=# -- You can also peek ahead in the change stream without consuming changes
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+--------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- The next call to pg_logical_slot_peek_changes() returns the same changes again
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+--------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- options can be passed to output plugin, to influence the formatting
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL, 'include-
timestamp', 'on');
    lsn    |  xid  |                          data
-----------+-------+--------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299 (at 2017-05-10 12:07:21.272494-04)
(3 rows)

postgres=# -- Remember to destroy a slot you no longer need to stop it consuming
postgres=# -- server resources:
postgres=# SELECT pg_drop_replication_slot('regression_slot');
 pg_drop_replication_slot
----------------------

(1 row)
```

下面的示例展示了如何通过流复制协议控制逻辑解码，所用程序是 PostgreSQL 发行版中自带的 pg_recvlogical。这要求已经配置好客户端认证以允许复制连接（见第 26.2.5.1 节），并且 `max_wal_senders` 的设置足够高，以容纳额外的连接。

```
$ pg_recvlogical -d postgres --slot=test --create-slot
$ pg_recvlogical -d postgres --slot=test --start -f -
Control+Z
$ psql -d postgres -c "INSERT INTO data(data) VALUES('4');"
$ fg
BEGIN 693
table public.data: INSERT: id[integer]:4 data[text]:'4'
COMMIT 693
Control+C
$ pg_recvlogical -d postgres --slot=test --drop-slot
```

49.2. 逻辑解码概念

49.2.1. 逻辑解码

逻辑解码是将数据库表上的所有持久更改提取为一种连贯、易于理解的格式的过程，这种格式无需详细了解数据库内部状态就能解释。

在 PostgreSQL 中，逻辑解码通过解码预写式日志的内容来实现。预写式日志描述的是存储层面的更改，而逻辑解码会将其转换成应用程序特定的形式，例如元组流或 SQL 语句流。

49.2.2. 复制槽

在逻辑复制的上下文中，一个槽表示一个更改流，客户端可以按照这些更改在源服务器上发生的顺序来重放它们。每个槽都从单个数据库流式传输一系列更改。

> **注意**
>
> PostgreSQL 也有流复制槽（见第 26.2.5 节），但它们在那里的使用方式略有不同。

复制槽在整个 PostgreSQL 集簇的所有数据库中都有唯一标识符。槽独立于使用它们的连接而持久存在，并且具备崩溃安全性。

逻辑槽在正常运行时会让每个更改只发送一次。每个槽的当前位置只会在检查点时持久化，因此在发生崩溃时，该槽可能会回退到较早的 LSN，这会导致服务器重启后最近的更改再次被发送。逻辑解码客户端有责任避免多次处理同一消息带来的不良影响。客户端可能希望记录解码时见到的最后一个 LSN，并跳过任何重复数据；或者（在使用复制协议时）请求从该 LSN 开始解码，而不是让服务器自行决定起始点。复制进度跟踪功能正是为此设计的，参见复制源。

对于同一个数据库，可以存在多个彼此独立的槽。每个槽都有自己的状态，允许不同的消费者从数据库更改流中的不同位置接收更改。对于大多数应用，每个消费者都需要一个单独的槽。

逻辑复制槽并不了解接收端的状态。甚至可以让多个不同的接收端在不同时间使用同一个槽；它们只会从前一个接收端停止消费更改的位置之后继续获得更改。但在任意给定时刻，只允许一个接收端从某个槽中消费更改。

> **小心**
>
> 复制槽可跨崩溃持久存在，而且不了解其消费者的状态。即使没有连接使用它们，它们也会阻止所需资源被移除。这会消耗存储，因为只要复制槽仍然需要，`VACUUM` 就不能移除所需的 WAL 和系统目录中的相关行。在极端情况下，这可能会导致数据库关闭，以防止事务 ID 回卷（见第 24.1.5 节）。因此，如果槽已经不再需要，就应当将其删除。

49.2.3. 输出插件

输出插件把预写式日志内部表示的数据转换成复制槽消费者所需的格式。

49.2.4. 导出快照

当使用流复制接口创建新的复制槽时（见 CREATE_REPLICATION_SLOT），会导出一个快照（见第 9.26.5 节），它准确反映了数据库的这样一种状态：从该状态之后开始，所有更改都会被纳入更改流。这可以通过使用 `SET TRANSACTION SNAPSHOT` 读取创建槽那一刻的数据库状

态，据此创建一个新副本。随后，该事务就可以用来转储该时刻的数据库状态，之后再利用该槽的内容进行更新，而不会丢失任何更改。

并非总能创建快照。特别是，连接到热备服务器时，创建快照会失败。不需要导出快照的应用程序可以使用 NOEXPORT_SNAPSHOT 选项禁止导出。

49.3. 流复制协议接口

命令

- CREATE_REPLICATION_SLOT *slot_name* LOGICAL *output_plugin*
- DROP_REPLICATION_SLOT *slot_name* [WAIT]
- START_REPLICATION SLOT *slot_name* LOGICAL ...

分别用于创建复制槽、删除复制槽，以及从复制槽流式传输更改。这些命令只在复制连接上可用，不能通过 SQL 使用。关于这些命令的细节，见第 53.4 节。

命令 pg_recvlogical 可以用来在流复制连接上控制逻辑解码（它在内部使用上述命令）。

49.4. 逻辑解码 SQL 接口

关于与逻辑解码交互的 SQL 层 API 的详细文档，见第 9.26.6 节。

同步复制（见第 26.2.8 节）只支持通过流复制接口使用的复制槽。函数接口以及额外的、非核心的接口都不支持同步复制。

49.5. 与逻辑解码相关的系统目录

pg_replication_slots 视图和 pg_stat_replication 视图分别提供了有关复制槽和流复制连接当前状态的信息。这些视图同时适用于物理复制和逻辑复制。

49.6. 逻辑解码输出插件

PostgreSQL 源码树中的 contrib/test_decoding 子目录里有一个输出插件示例。

49.6.1. 初始化函数

加载输出插件的方式是动态加载共享库，并以输出插件的名称作为库的基本名称。定位该库时使用常规库搜索路径。为了提供所需的输出插件回调，并表明这个库确实是输出插件，库必须提供一个名为 `_PG_output_plugin_init` 的函数。此函数会接收一个结构体，需要在其中填写各项操作的回调函数指针。

```
typedef struct OutputPluginCallbacks
{
    LogicalDecodeStartupCB startup_cb;
    LogicalDecodeBeginCB begin_cb;
    LogicalDecodeChangeCB change_cb;
    LogicalDecodeTruncateCB truncate_cb;
    LogicalDecodeCommitCB commit_cb;
    LogicalDecodeMessageCB message_cb;
    LogicalDecodeFilterByOriginCB filter_by_origin_cb;
    LogicalDecodeShutdownCB shutdown_cb;
} OutputPluginCallbacks;

typedef void (*LogicalOutputPluginInit) (struct OutputPluginCallbacks *cb);
```

其中，`begin_cb`，`change_cb` 和 `commit_cb` 回调是必需的，而`startup_cb`，`filter_by_origin_cb`，`truncate_cb`，以及`shutdown_cb` 是可选的。如果没有设置 `truncate_cb`，但要解码一个`TRUNCATE` 操作，那么该操作会被忽略。

49.6.2. 能力

为了对变更进行解码、格式化和输出，输出插件可以使用后端的大部分常规基础设施，包括调用输出函数。可以对关系进行只读访问，但仅限于以下两种关系：由`initdb` 创建在 `pg_catalog` 模式中的关系，或者使用以下方式标记为用户提供的系统目录表的关系：

```
ALTER TABLE user_catalog_table SET (user_catalog_table = true);
CREATE TABLE another_catalog_table(data text) WITH (user_catalog_table = true);
```

禁止执行任何会导致分配事务 ID 的操作。这包括向表写入数据、执行 DDL 更改以及调用 `txid_current()`。

49.6.3. 输出模式

输出插件回调几乎可以用任意格式向消费者传递数据。对于某些用例，例如通过 SQL 查看更改，把数据返回为能够容纳任意数据的数据类型（例如 `bytea`）会很不方便。如果输出插件只输出采用服务器编码的文本数据，它可以在启动回调中，将 `OutputPluginOptions.output_type` 设置为 `OUTPUT_PLUGIN_TEXTUAL_OUTPUT` 而不是 `OUTPUT_PLUGIN_BINARY_OUTPUT`，以声明这一点。在这种情况下，所有数据都必须采用服务器编码，这样才能装入 `text` datum。断言开启的构建会检查这一点。

49.6.4. 输出插件回调

输出插件通过它所提供的各类回调获知正在发生的更改。

并发事务按提交顺序解码，并且只有属于某个特定事务的更改，才会在 `begin` 和 `commit` 回调之间被解码。显式或隐式回滚的事务永远不会被解码。成功的保存点会按照它们在该事务中执行的顺序，被折叠进包含它们的事务中。

> **注意**
>
> 只有已经安全刷写到磁盘的事务才会被解码。因此，当 synchronous_commit 设置为 off 时，COMMIT 可能不会在紧随其后的 pg_logical_slot_get_changes() 调用中立即被解码。

49.6.4.1. 启动回调

只要创建复制槽，或者要求某个复制槽开始流式传输更改，就会调用可选的 startup_cb 回调，而不管当前是否已经有准备好输出的更改。

```
typedef void (*LogicalDecodeStartupCB) (struct LogicalDecodingContext *ctx,
                                        OutputPluginOptions *options,
                                        bool is_init);
```

当复制槽正在创建时，is_init 参数为真，否则为假。*options* 指向一个结构体，输出插件可以在其中设置选项：

```
typedef struct OutputPluginOptions
{
    OutputPluginOutputType output_type;
    bool        receive_rewrites;
} OutputPluginOptions;
```

output_type 必须设置为 OUTPUT_PLUGIN_TEXTUAL_OUTPUT 或 OUTPUT_PLUGIN_BINARY_OUTPUT。另见第 49.6.3 节。如果 receive_rewrites 为真，则在某些 DDL 操作期间由堆重写产生的更改也会传给输出插件。这对处理 DDL 复制的插件很有用，但需要特殊处理。

启动回调应当验证 ctx->output_plugin_options 中的选项。如果输出插件需要保存状态，可以使用 ctx->output_plugin_private 来存储。

49.6.4.2. 关闭回调

当一个此前处于活动状态的复制槽不再使用时，就会调用可选的 shutdown_cb 回调。它可用于释放输出插件私有的资源。此时只是停止流式传输，并不一定删除该槽。

```
typedef void (*LogicalDecodeShutdownCB) (struct LogicalDecodingContext *ctx);
```

49.6.4.3. 事务开始回调

只要某个已提交事务的开始被解码，就会调用必需的 begin_cb 回调。已中止的事务及其内容永远不会被解码。

```
typedef void (*LogicalDecodeBeginCB) (struct LogicalDecodingContext *ctx,
                                      ReorderBufferTXN *txn);
```

txn 参数包含该事务的元信息，例如它提交时的时间戳以及它的 XID。

49.6.4.4. 事务结束回调

只要事务提交被解码，就会调用必需的 commit_cb 回调。如果有被修改的行，那么在此之前，所有已修改行的 change_cb 回调都已经被调用过。

```
typedef void (*LogicalDecodeCommitCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       XLogRecPtr commit_lsn);
```

49.6.4.5. 更改回调

必需的 change_cb 回调会在事务中的每次行修改时被调用，不论该修改是INSERT，UPDATE 或 DELETE。即使原命令一次修改了多行，也会针对每一行单独调用此回调。

```
typedef void (*LogicalDecodeChangeCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       Relation relation,
                                       ReorderBufferChange *change);
```

参数 *ctx* 和 *txn* 所含的内容与 begin_cb 和 commit_cb 回调中的相同。此外，还会传入关系描述符*relation*（指向该行所属的关系），以及结构体*change*（描述该行的修改）。

> **注意**
>
> 只有用户定义表中的更改才能通过逻辑解码提取，而且这些表不能是不记录 WAL 的表（见 UNLOGGED），也不能是临时表（见 TEMPORARY 或 TEMP）。

49.6.4.6. 截断回调

可选的 truncate_cb 回调会在解码 TRUNCATE 命令时调用。

```
typedef void (*LogicalDecodeTruncateCB) (struct LogicalDecodingContext *ctx,
                                         ReorderBufferTXN *txn,
                                         int nrelations,
                                         Relation relations[],
                                         ReorderBufferChange *change);
```

这些参数与 change_cb 回调类似。不过，由于对通过外键关联的表执行 TRUNCATE 时需要一起执行动作，所以该回调接收的是关系数组，而不是单个关系。详见TRUNCATE 语句的说明。

49.6.4.7. 源过滤回调

可选的 filter_by_origin_cb 回调用于判定，从 *origin_id* 重放而来的数据是否为输出插件所关心的数据。

```
typedef bool (*LogicalDecodeFilterByOriginCB) (struct LogicalDecodingContext *ctx,
                                               RepOriginId origin_id);
```

ctx 参数的内容与其他回调相同。除了源本身之外，没有其他信息可用。如果要表明来自传入节点的更改并不相关，则返回 true，这会使这些更改被过滤掉；否则返回 false。对于被过滤掉的事务和更改，其他回调都不会被调用。

在实现级联复制或多向复制方案时，这个回调很有用。按源过滤可以避免在这类配置中同一更改被来回复制。虽然事务和更改本身也带有源信息，但通过这个回调来过滤会明显更高效。

49.6.4.8. 通用消息回调

可选的 `message_cb` 回调会在每次解码出一条逻辑解码消息时被调用。

```
typedef void (*LogicalDecodeMessageCB) (struct LogicalDecodingContext *ctx,
                                        ReorderBufferTXN *txn,
                                        XLogRecPtr message_lsn,
                                        bool transactional,
                                        const char *prefix,
                                        Size message_size,
                                        const char *message);
```

其中，*`txn`* 参数包含事务的元信息，例如提交时间戳及其 XID。但请注意，如果消息是非事务性的，并且记录该消息的事务尚未分配 XID，这个参数可能为 NULL。*`lsn`* 包含消息的 WAL 位置。*`transactional`* 表示消息是否以事务性方式发送。*`prefix`* 是任意的、以空字符终止的前缀，可用于识别当前插件关注的消息。最后，*`message`* 参数保存实际消息，其大小为 *`message_size`*。

应特别注意，确保输出插件视为有意义的消息前缀具有唯一性。使用扩展的名称或输出插件自身的名称通常是一个不错的选择。

49.6.5. 产生输出的函数

为了真正产生输出，输出插件可以在 `StringInfo` 输出缓冲区 `ctx->out` 中写入数据，此时位于 `begin_cb`、`commit_cb` 或 `change_cb` 回调内部。写入输出缓冲区之前，必须先调用 `OutputPluginPrepareWrite(ctx, last_write)`；写完缓冲区之后，必须调用 `OutputPluginWrite(ctx, last_write)` 来执行写出。*`last_write`* 指示某次写出是否为该回调的最后一次写出。

下面的示例展示了如何把数据输出给输出插件的消费者：

```
OutputPluginPrepareWrite(ctx, true);
appendStringInfo(ctx->out, "BEGIN %u", txn->xid);
OutputPluginWrite(ctx, true);
```

49.7. 逻辑解码输出写入器

逻辑解码还可以增加更多输出方法。详情见 `src/backend/replication/logical/logicalfuncs.c`。本质上，需要提供三个函数：一个读取 WAL，一个准备写出输出，另一个执行写出（见第 49.6.5 节）。

49.8. 逻辑解码的同步复制支持

49.8.1. 概述

逻辑解码可用于构建同步复制方案，其用户接口与流复制的同步复制相同。要做到这一点，必须使用流复制接口（见第 49.3 节）来流式传出数据。客户端必须像流复制客户端一样发送 `Standby status update (F)` 消息（见第 53.4 节）。

> **注意**
>
> 通过逻辑解码接收更改的同步副本，只能在单个数据库范围内工作。而与此不同，*synchronous_standby_names* 当前是整个服务器范围的，这意味着如果有多个数据库在被活跃使用，这种技术将无法正常工作。

49.8.2. 注意事项

在同步复制配置中，如果事务对目录表（包括用户目录表）加了排他锁，就可能发生死锁。参见第 49.6.2 节了解用户目录表。这是因为事务的逻辑解码可能会锁定目录表以访问它们。为避免这种情况，用户必须避免对目录表（包括用户目录表）获取排他锁。以下方式可能导致这种情况：

- 在事务中显式发出 LOCK 命令来锁定 pg_class。
- 在事务中执行 CLUSTER 命令处理 pg_class。
- 在事务中对目录表（包括用户目录表）执行 TRUNCATE。

注意，这些可能造成死锁的命令不仅适用于上面明确指出的系统目录表，也适用于其他任何目录表（包括用户目录表）。

第 50 章 复制进度跟踪

复制源旨在让在逻辑解码之上实现逻辑复制方案变得更容易。它们为两个常见问题提供了解决方案：

- 如何安全地跟踪复制进度
- 如何根据一行数据的来源改变复制行为；例如，在双向复制拓扑中防止形成环路

复制源只有两个属性：名称和 OID。名称是在系统之间引用该复制源时应使用的标识，它是自由形式的 `text` 值。使用它时，应尽量降低不同复制方案创建的复制源之间发生冲突的概率；例如，可以在前面加上复制方案的名称作为前缀。OID 仅用于在空间效率很重要的场合避免存储较长的名称形式。它绝不应在不同系统之间共享。

可以使用函数 `pg_replication_origin_create()` 创建复制源；使用 `pg_replication_origin_drop()` 删除复制源；并可在 `pg_replication_origin` 系统目录中查看它们。

构建复制方案时，一个并不简单的部分是要以安全的方式跟踪重放进度。当应用进程或整个集簇崩溃时，必须能够找出数据已经成功复制到了哪里。对此的一些朴素解决方案，例如每重放一个事务就更新表中的某一行，会带来运行时开销和数据库膨胀等问题。

借助复制源基础设施，可以将一个会话标记为正在重放来自远程节点的数据（使用函数 `pg_replication_origin_session_setup()`）。此外，还可以使用 `pg_replication_origin_xact_setup()` 按事务为每个源事务配置其 LSN 和提交时间戳。这样做之后，复制进度就会以崩溃安全的方式持久保存。所有复制源的重放进度都可以在 `pg_replication_origin_status` 视图中查看。某个单独复制源的进度，例如在恢复复制时，可以通过 `pg_replication_origin_progress()`（适用于任意复制源）或 `pg_replication_origin_session_progress()`（适用于当前会话中配置的复制源）来获取。

在比恰好从一个系统复制到另一个系统更复杂的复制拓扑中，另一个问题在于很难避免再次复制已经重放过的行。这既会导致复制出现环路，也会带来低效。复制源提供了一种可选机制来识别并防止这种情况。当使用前一段提到的函数进行配置后，传递给输出插件回调（见第 49.6 节）的每一项变更和事务，都会带上其生成会话的复制源标记。这使得输出插件可以区别对待它们，例如忽略所有并非源自本地的行。此外，`filter_by_origin_cb` 回调还可用于根据来源过滤逻辑解码变更流。虽然灵活性较低，但通过该回调进行过滤要比在输出插件中自行过滤高效得多。

部分 VI. 参考

本参考中的各条目旨在以适当的篇幅，对各自的主题给出权威、完整且正式的总结。关于如何使用 PostgreSQL 的更多信息，例如叙述性说明、教程或示例，可在本书的其他部分找到。另见各参考页列出的交叉引用。

这些参考条目也提供传统的“man”手册页形式。

目录

I. SQL 命令 1370
ABORT 1374
ALTER AGGREGATE 1375
ALTER COLLATION 1377
ALTER CONVERSION 1379
ALTER DATABASE 1381
ALTER DEFAULT PRIVILEGES 1384
ALTER DOMAIN 1387
ALTER EVENT TRIGGER 1391
ALTER EXTENSION 1392
ALTER FOREIGN DATA WRAPPER 1395
ALTER FOREIGN TABLE 1397
ALTER FUNCTION 1402
ALTER GROUP 1406
ALTER INDEX 1408
ALTER LANGUAGE 1411
ALTER LARGE OBJECT 1412
ALTER MATERIALIZED VIEW 1413
ALTER OPERATOR 1415
ALTER OPERATOR CLASS 1417
ALTER OPERATOR FAMILY 1419
ALTER POLICY 1423
ALTER PROCEDURE 1425
ALTER PUBLICATION 1428
ALTER ROLE 1430
ALTER ROUTINE 1434
ALTER RULE 1436
ALTER SCHEMA 1437
ALTER SEQUENCE 1438
ALTER SERVER 1441
ALTER STATISTICS 1443
ALTER SUBSCRIPTION 1444
ALTER SYSTEM 1446
ALTER TABLE 1448
ALTER TABLESPACE 1463
ALTER TEXT SEARCH CONFIGURATION 1465
ALTER TEXT SEARCH DICTIONARY 1467
ALTER TEXT SEARCH PARSER 1469
ALTER TEXT SEARCH TEMPLATE 1470
ALTER TRIGGER 1471
ALTER TYPE 1473
ALTER USER 1477
ALTER USER MAPPING 1478
ALTER VIEW 1480
ANALYZE 1482
BEGIN 1485
CALL 1487
CHECKPOINT 1488
CLOSE 1489
CLUSTER 1491
COMMENT 1493

COMMIT 1497
COMMIT PREPARED 1498
COPY 1499
CREATE ACCESS METHOD 1509
CREATE AGGREGATE 1510
CREATE CAST 1517
CREATE COLLATION 1522
CREATE CONVERSION 1524
CREATE DATABASE 1526
CREATE DOMAIN 1529
CREATE EVENT TRIGGER 1532
CREATE EXTENSION 1534
CREATE FOREIGN DATA WRAPPER 1537
CREATE FOREIGN TABLE 1539
CREATE FUNCTION 1543
CREATE GROUP 1552
CREATE INDEX 1553
CREATE LANGUAGE 1561
CREATE MATERIALIZED VIEW 1564
CREATE OPERATOR 1566
CREATE OPERATOR CLASS 1569
CREATE OPERATOR FAMILY 1572
CREATE POLICY 1573
CREATE PROCEDURE 1579
CREATE PUBLICATION 1582
CREATE ROLE 1584
CREATE RULE 1589
CREATE SCHEMA 1592
CREATE SEQUENCE 1595
CREATE SERVER 1599
CREATE STATISTICS 1601
CREATE SUBSCRIPTION 1603
CREATE TABLE 1606
CREATE TABLE AS 1626
CREATE TABLESPACE 1629
CREATE TEXT SEARCH CONFIGURATION 1631
CREATE TEXT SEARCH DICTIONARY 1633
CREATE TEXT SEARCH PARSER 1635
CREATE TEXT SEARCH TEMPLATE 1637
CREATE TRANSFORM 1639
CREATE TRIGGER 1641
CREATE TYPE 1648
CREATE USER 1657
CREATE USER MAPPING 1658
CREATE VIEW 1660
DEALLOCATE 1665
DECLARE 1666
DELETE 1670
DISCARD 1673
DO 1675
DROP ACCESS METHOD 1677
DROP AGGREGATE 1678
DROP CAST 1680
DROP COLLATION 1681

DROP CONVERSION 1682
DROP DATABASE 1683
DROP DOMAIN 1684
DROP EVENT TRIGGER 1685
DROP EXTENSION 1686
DROP FOREIGN DATA WRAPPER 1687
DROP FOREIGN TABLE 1688
DROP FUNCTION 1689
DROP GROUP 1691
DROP INDEX 1692
DROP LANGUAGE 1694
DROP MATERIALIZED VIEW 1696
DROP OPERATOR 1697
DROP OPERATOR CLASS 1699
DROP OPERATOR FAMILY 1701
DROP OWNED 1703
DROP POLICY 1704
DROP PROCEDURE 1705
DROP PUBLICATION 1707
DROP ROLE 1708
DROP ROUTINE 1710
DROP RULE 1711
DROP SCHEMA 1712
DROP SEQUENCE 1714
DROP SERVER 1715
DROP STATISTICS 1716
DROP SUBSCRIPTION 1717
DROP TABLE 1719
DROP TABLESPACE 1721
DROP TEXT SEARCH CONFIGURATION 1722
DROP TEXT SEARCH DICTIONARY 1723
DROP TEXT SEARCH PARSER 1724
DROP TEXT SEARCH TEMPLATE 1725
DROP TRANSFORM 1726
DROP TRIGGER 1727
DROP TYPE 1729
DROP USER 1730
DROP USER MAPPING 1731
DROP VIEW 1732
END 1733
EXECUTE 1734
EXPLAIN 1735
FETCH 1740
GRANT 1744
IMPORT FOREIGN SCHEMA 1751
INSERT 1753
LISTEN 1760
LOAD 1762
LOCK 1763
MOVE 1766
NOTIFY 1768
PREPARE 1770
PREPARE TRANSACTION 1773
REASSIGN OWNED 1775

REFRESH MATERIALIZED VIEW 1776
REINDEX 1778
RELEASE SAVEPOINT 1781
RESET 1783
REVOKE 1784
ROLLBACK 1788
ROLLBACK PREPARED 1789
ROLLBACK TO SAVEPOINT 1790
SAVEPOINT 1792
SECURITY LABEL 1794
SELECT 1797
SELECT INTO 1818
SET 1820
SET CONSTRAINTS 1823
SET ROLE 1825
SET SESSION AUTHORIZATION 1827
SET TRANSACTION 1829
SHOW 1832
START TRANSACTION 1834
TRUNCATE 1835
UNLISTEN 1837
UPDATE 1839
VACUUM 1844
VALUES 1847
II. PostgreSQL 客户端应用 1850
clusterdb 1851
createdb 1854
createuser 1857
dropdb 1862
dropuser 1865
ecpg 1868
pg_basebackup 1871
pgbench 1878
pg_config 1896
pg_dump 1899
pg_dumpall 1911
pg_isready 1918
pg_receivewal 1920
pg_recvlogical 1924
pg_restore 1928
psql 1937
reindexdb 1977
vacuumdb 1980
III. PostgreSQL 服务器端应用 1984
initdb 1985
pg_archivecleanup 1989
pg_controldata 1991
pg_ctl 1992
pg_resetwal 1998
pg_rewind 2002
pg_test_fsync 2005
pg_test_timing 2006
pg_upgrade 2010
pg_verify_checksums 2018

pg_waldump 2019
postgres 2022
postmaster 2029

SQL 命令

本部分包含 PostgreSQL 支持的 SQL 命令的参考信息。这里的 “SQL” 指的是一般意义上的该语言；每条命令的标准符合性和兼容性信息可在相应的参考页中找到。

目录

ABORT 1374
ALTER AGGREGATE 1375
ALTER COLLATION 1377
ALTER CONVERSION 1379
ALTER DATABASE 1381
ALTER DEFAULT PRIVILEGES 1384
ALTER DOMAIN 1387
ALTER EVENT TRIGGER 1391
ALTER EXTENSION 1392
ALTER FOREIGN DATA WRAPPER 1395
ALTER FOREIGN TABLE 1397
ALTER FUNCTION 1402
ALTER GROUP 1406
ALTER INDEX 1408
ALTER LANGUAGE 1411
ALTER LARGE OBJECT 1412
ALTER MATERIALIZED VIEW 1413
ALTER OPERATOR 1415
ALTER OPERATOR CLASS 1417
ALTER OPERATOR FAMILY 1419
ALTER POLICY 1423
ALTER PROCEDURE 1425
ALTER PUBLICATION 1428
ALTER ROLE 1430
ALTER ROUTINE 1434
ALTER RULE 1436
ALTER SCHEMA 1437
ALTER SEQUENCE 1438
ALTER SERVER 1441
ALTER STATISTICS 1443
ALTER SUBSCRIPTION 1444
ALTER SYSTEM 1446
ALTER TABLE 1448
ALTER TABLESPACE 1463
ALTER TEXT SEARCH CONFIGURATION 1465
ALTER TEXT SEARCH DICTIONARY 1467
ALTER TEXT SEARCH PARSER 1469
ALTER TEXT SEARCH TEMPLATE 1470
ALTER TRIGGER 1471
ALTER TYPE 1473
ALTER USER 1477
ALTER USER MAPPING 1478
ALTER VIEW 1480
ANALYZE 1482

BEGIN 1485
CALL 1487
CHECKPOINT 1488
CLOSE 1489
CLUSTER 1491
COMMENT 1493
COMMIT 1497
COMMIT PREPARED 1498
COPY 1499
CREATE ACCESS METHOD 1509
CREATE AGGREGATE 1510
CREATE CAST 1517
CREATE COLLATION 1522
CREATE CONVERSION 1524
CREATE DATABASE 1526
CREATE DOMAIN 1529
CREATE EVENT TRIGGER 1532
CREATE EXTENSION 1534
CREATE FOREIGN DATA WRAPPER 1537
CREATE FOREIGN TABLE 1539
CREATE FUNCTION 1543
CREATE GROUP 1552
CREATE INDEX 1553
CREATE LANGUAGE 1561
CREATE MATERIALIZED VIEW 1564
CREATE OPERATOR 1566
CREATE OPERATOR CLASS 1569
CREATE OPERATOR FAMILY 1572
CREATE POLICY 1573
CREATE PROCEDURE 1579
CREATE PUBLICATION 1582
CREATE ROLE 1584
CREATE RULE 1589
CREATE SCHEMA 1592
CREATE SEQUENCE 1595
CREATE SERVER 1599
CREATE STATISTICS 1601
CREATE SUBSCRIPTION 1603
CREATE TABLE 1606
CREATE TABLE AS 1626
CREATE TABLESPACE 1629
CREATE TEXT SEARCH CONFIGURATION 1631
CREATE TEXT SEARCH DICTIONARY 1633
CREATE TEXT SEARCH PARSER 1635
CREATE TEXT SEARCH TEMPLATE 1637
CREATE TRANSFORM 1639
CREATE TRIGGER 1641
CREATE TYPE 1648
CREATE USER 1657
CREATE USER MAPPING 1658
CREATE VIEW 1660
DEALLOCATE 1665
DECLARE 1666
DELETE 1670

DISCARD 1673
DO 1675
DROP ACCESS METHOD 1677
DROP AGGREGATE 1678
DROP CAST 1680
DROP COLLATION 1681
DROP CONVERSION 1682
DROP DATABASE 1683
DROP DOMAIN 1684
DROP EVENT TRIGGER 1685
DROP EXTENSION 1686
DROP FOREIGN DATA WRAPPER 1687
DROP FOREIGN TABLE 1688
DROP FUNCTION 1689
DROP GROUP 1691
DROP INDEX 1692
DROP LANGUAGE 1694
DROP MATERIALIZED VIEW 1696
DROP OPERATOR 1697
DROP OPERATOR CLASS 1699
DROP OPERATOR FAMILY 1701
DROP OWNED 1703
DROP POLICY 1704
DROP PROCEDURE 1705
DROP PUBLICATION 1707
DROP ROLE 1708
DROP ROUTINE 1710
DROP RULE 1711
DROP SCHEMA 1712
DROP SEQUENCE 1714
DROP SERVER 1715
DROP STATISTICS 1716
DROP SUBSCRIPTION 1717
DROP TABLE 1719
DROP TABLESPACE 1721
DROP TEXT SEARCH CONFIGURATION 1722
DROP TEXT SEARCH DICTIONARY 1723
DROP TEXT SEARCH PARSER 1724
DROP TEXT SEARCH TEMPLATE 1725
DROP TRANSFORM 1726
DROP TRIGGER 1727
DROP TYPE 1729
DROP USER 1730
DROP USER MAPPING 1731
DROP VIEW 1732
END 1733
EXECUTE 1734
EXPLAIN 1735
FETCH 1740
GRANT 1744
IMPORT FOREIGN SCHEMA 1751
INSERT 1753
LISTEN 1760
LOAD 1762

LOCK 1763
MOVE 1766
NOTIFY 1768
PREPARE 1770
PREPARE TRANSACTION 1773
REASSIGN OWNED 1775
REFRESH MATERIALIZED VIEW 1776
REINDEX 1778
RELEASE SAVEPOINT 1781
RESET 1783
REVOKE 1784
ROLLBACK 1788
ROLLBACK PREPARED 1789
ROLLBACK TO SAVEPOINT 1790
SAVEPOINT 1792
SECURITY LABEL 1794
SELECT 1797
SELECT INTO 1818
SET 1820
SET CONSTRAINTS 1823
SET ROLE 1825
SET SESSION AUTHORIZATION 1827
SET TRANSACTION 1829
SHOW 1832
START TRANSACTION 1834
TRUNCATE 1835
UNLISTEN 1837
UPDATE 1839
VACUUM 1844
VALUES 1847

ABORT

ABORT — 中止当前事务

大纲

```
ABORT [ WORK | TRANSACTION ]
```

描述

`ABORT` 回滚当前事务，并丢弃该事务所做的全部更新。这个命令在行为上与标准 SQL 命令 ROLLBACK 完全相同，只是出于历史原因而保留。

参数

`WORK`
`TRANSACTION`

可选关键字，没有任何作用。

注解

使用 COMMIT 可成功地终止一个事务。

在事务块之外发出 `ABORT` 会产生一条警告，除此之外没有效果。

示例

要中止所有更改：

```
ABORT;
```

兼容性

这个命令是 PostgreSQL 扩展，只是出于历史原因而保留。`ROLLBACK` 是与之等效的标准 SQL 命令。

另见

BEGIN, COMMIT, ROLLBACK

ALTER AGGREGATE

ALTER AGGREGATE — 更改一个聚合函数的定义

大纲

```
ALTER AGGREGATE name ( aggregate_signature ) RENAME TO new_name
ALTER AGGREGATE name ( aggregate_signature )
                OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER AGGREGATE name ( aggregate_signature ) SET SCHEMA new_schema
```

其中 *aggregate_signature* 为:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ...
 ]
```

描述

ALTER AGGREGATE 更改聚合函数的定义。

要使用 ALTER AGGREGATE，必须拥有该聚合函数。要更改聚合函数的模式，还必须在新模式上拥有CREATE 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在聚合函数所在模式上拥有CREATE 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建聚合函数完成的操作。不过，超级用户无论如何都可以更改任何聚合函数的所有权。）

参数

name

现有聚合函数的名称（可以用模式限定）。

argmode

参数的模式：IN 或 VARIADIC。如果省略，默认为IN。

argname

参数的名称。注意，ALTER AGGREGATE 实际上并不关心参数名，因为确定聚合函数的标识只需要参数的数据类型。

argtype

聚合函数作用于其上的输入数据类型。要引用零参数聚合函数，请在参数说明列表的位置写上*。要引用有序集聚合函数，请在直接参数说明和聚合参数说明之间写上ORDER BY。

new_name

聚合函数的新名称。

new_owner

聚合函数的新拥有者。

new_schema

聚合函数的新模式。

注解

引用有序集聚合的推荐语法，是像CREATE AGGREGATE 中那样，在直接参数说明和聚合参数说明之间写上ORDER BY。不过，省略ORDER BY，直接把两部分参数说明合并为一个列表也同样可行。在这种简写形式中，如果直接参数列表和聚合参数列表中都使用了VARIADIC "any"，则只需写一次VARIADIC "any"。

示例

要把用于类型 integer 的聚合函数 myavg 重命名为 my_average：

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

要把用于类型 integer 的聚合函数 myavg 的拥有者改为 joe：

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

把直接参数类型为 float8、聚合参数类型为integer 的有序集聚合 mypercentile 移动到模式 myschema 中：

```
ALTER AGGREGATE mypercentile(float8 ORDER BY integer) SET SCHEMA myschema;
```

这也同样可行：

```
ALTER AGGREGATE mypercentile(float8, integer) SET SCHEMA myschema;
```

兼容性

在 SQL 标准中没有 ALTER AGGREGATE 语句。

另见

CREATE AGGREGATE, DROP AGGREGATE

ALTER COLLATION

ALTER COLLATION — 更改排序规则的定义

大纲

```
ALTER COLLATION name REFRESH VERSION

ALTER COLLATION name RENAME TO new_name
ALTER COLLATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER COLLATION name SET SCHEMA new_schema
```

描述

`ALTER COLLATION` 更改排序规则的定义。

要使用 `ALTER COLLATION`，必须拥有该排序规则。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在排序规则所在模式上拥有`CREATE` 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建排序规则完成的操作。不过，超级用户无论如何都可以更改任何排序规则的所有权。）

参数

name

一个现有排序规则的名称（可以是模式限定的）。

new_name

排序规则的新名称。

new_owner

排序规则的新拥有者。

new_schema

排序规则的新模式。

`REFRESH VERSION`

刷新排序规则版本。参阅下面的注解。

注解

使用 ICU 库提供的排序规则时，创建排序规则对象会在系统目录中记录 ICU 给出的排序器版本。使用该排序规则时，会将当前版本与记录的版本进行比较；如果二者不匹配，就会发出警告，例如：

```
WARNING:  collation "xx-x-icu" has version mismatch
DETAIL:  The collation in the database was created using version 1.2.3.4, but the operating
 system provides version 2.3.4.5.
HINT:  Rebuild all objects affected by this collation and run ALTER COLLATION pg_catalog."xx-x-
icu" REFRESH VERSION, or build PostgreSQL with the right library version.
```

排序规则定义的变化可能导致索引损坏及其他问题，因为数据库系统依赖已存储对象具有特定的排序顺序。一般来说应避免这种情况，但在某些合理情形下仍可能发生，例如使用 pg_upgrade 升级到链接了较新 ICU 版本的服务器二进制文件时。出现这种情况时，应重建所有依赖该排序规则的对象，例如使用 REINDEX。完成后，可以使用命令 ALTER COLLATION ... REFRESH VERSION 刷新排序规则版本。这样会更新系统目录，使其记录当前的排序器版本，并让该警告消失。请注意，这实际上并不会检查所有受影响对象是否都已被正确重建。

以下查询可用于识别当前数据库中所有需要刷新的排序规则，以及依赖它们的对象：

```
SELECT pg_describe_object(refclassid, refobjid, refobjsubid) AS "Collation",
       pg_describe_object(classid, objid, objsubid) AS "Object"
  FROM pg_depend d JOIN pg_collation c
       ON refclassid = 'pg_collation'::regclass AND refobjid = c.oid
  WHERE c.collversion <> pg_collation_actual_version(c.oid)
  ORDER BY 1, 2;
```

示例

要将排序规则 de_DE 重命名为 german：

```
ALTER COLLATION "de_DE" RENAME TO german;
```

要将排序规则 en_US 的拥有者改为 joe：

```
ALTER COLLATION "en_US" OWNER TO joe;
```

兼容性

SQL 标准中没有 ALTER COLLATION 语句。

另见

CREATE COLLATION, DROP COLLATION

ALTER CONVERSION

ALTER CONVERSION — 更改一个转换的定义

大纲

```
ALTER CONVERSION name RENAME TO new_name
ALTER CONVERSION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER CONVERSION name SET SCHEMA new_schema
```

描述

`ALTER CONVERSION` 更改一个转换的定义。

要使用 `ALTER CONVERSION`，必须拥有该转换。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在转换所在模式上拥有`CREATE` 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建转换完成的操作。不过，超级用户无论如何都可以更改任何转换的所有权。）

参数

name

一个现有转换的名称（可以是模式限定的）。

new_name

转换的新名称。

new_owner

转换的新拥有者。

new_schema

转换的新模式。

示例

要把转换 `iso_8859_1_to_utf8` 重命名为 `latin1_to_unicode`：

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

要把转换 `iso_8859_1_to_utf8` 的拥有者改成 `joe`：

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

兼容性

在 SQL 标准中没有 `ALTER CONVERSION` 语句。

另见

CREATE CONVERSION, DROP CONVERSION

ALTER DATABASE

ALTER DATABASE — 更改一个数据库

大纲

```
ALTER DATABASE name [ [ WITH ] option [ ... ] ]
```

其中 *option* 可以是:

```
    ALLOW_CONNECTIONS allowconn
    CONNECTION LIMIT connlimit
    IS_TEMPLATE istemplate

ALTER DATABASE name RENAME TO new_name

ALTER DATABASE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER DATABASE name SET TABLESPACE new_tablespace

ALTER DATABASE name SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name SET configuration_parameter FROM CURRENT
ALTER DATABASE name RESET configuration_parameter
ALTER DATABASE name RESET ALL
```

描述

`ALTER DATABASE` 更改一个数据库的属性。

第一种形式更改某些数据库级设置（详见下文）。只有数据库拥有者或者超级用户才能更改这些设置。

第二种形式更改数据库的名称。只有数据库拥有者或者超级用户可以重命名数据库；非超级用户拥有者还必须具有 `CREATEDB` 权限。当前连接的数据库不能被重命名（如果需要这样做，请连接到另一个数据库）。

第三种形式更改数据库的所有者。要更改所有者，必须拥有该数据库，并且还必须是新所有者角色的直接或间接成员，并且必须拥有`CREATEDB` 权限。（请注意，超级用户会自动拥有所有这些权限。）

第四种形式更改数据库的默认表空间。只有数据库拥有者或者超级用户可以这样做；你还必须对新表空间具有创建权限。该命令会将数据库旧默认表空间中的所有表和索引在物理上移动到新表空间中。对于该数据库而言，新默认表空间必须为空，并且不能有人连接到该数据库。位于非默认表空间中的表和索引不受影响。

其余形式会更改某个 PostgreSQL 数据库的运行时配置变量的会话默认值。此后每当在该数据库中启动一个新会话时，指定的值就会成为会话默认值。数据库特定的默认值会覆盖 `postgresql.conf` 中的设置，或者从`postgres` 命令行接收到的设置。只有数据库拥有者或者超级用户才能更改该数据库的会话默认值。某些变量不能以这种方式设置，或者只能由超级用户设置。

参数

name

要修改其属性的数据库名称。

allowconn

如果为 false，则任何人都不能连接到该数据库。

connlimit

这个数据库允许多少并发连接。-1 表示没有限制。

istemplate

如果为 true，则任何具有 CREATEDB 权限的用户都可以克隆该数据库；如果为 false，则只有超级用户或该数据库的拥有者可以克隆它。

new_name

数据库的新名称。

new_owner

数据库的新拥有者。

new_tablespace

数据库的新默认表空间。

这种形式的命令不能在事务块内执行。

configuration_parameter
value

将此数据库在指定配置参数上的会话默认值设为给定值。如果 *value* 是 DEFAULT，或者等效地使用了 RESET，则数据库特定设置会被移除，因此新会话将继承系统范围的默认设置。使用 RESET ALL 可清除所有数据库特定设置。SET FROM CURRENT 会把该参数在当前会话中的值保存为数据库特定值。

关于允许的参数名和值的更多信息，见SET 和第 19 章。

注解

也可以把会话默认值绑定到特定角色，而不是数据库；见 ALTER ROLE。如果发生冲突，角色特定设置会覆盖数据库特定设置。

示例

要在数据库 test 中默认禁用索引扫描：

```
ALTER DATABASE test SET enable_indexscan TO off;
```

兼容性

`ALTER DATABASE` 语句是一个 PostgreSQL 扩展。

另见

CREATE DATABASE, DROP DATABASE, SET, CREATE TABLESPACE

ALTER DEFAULT PRIVILEGES

ALTER DEFAULT PRIVILEGES — 定义默认访问权限

大纲

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target_role [, ...] ]
    [ IN SCHEMA schema_name [, ...] ]
    abbreviated_grant_or_revoke
```

其中 *abbreviated_grant_or_revoke* 为以下之一:

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTIONS | ROUTINES }
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | CREATE | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTIONS | ROUTINES }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
```

```
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | CREATE | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

描述

ALTER DEFAULT PRIVILEGES 允许设置将应用于将来创建对象的权限。（它不影响已存在对象的权限。）目前只能更改模式、表（包括视图和外部表）、序列、函数和类型（包括域）的权限。对于此命令，函数包括聚合函数和过程。FUNCTIONS 和 ROUTINES 在此命令中等价。（今后更推荐使用 ROUTINES，作为函数和过程合称时的标准术语。在 PostgreSQL 早期版本中，只允许使用FUNCTIONS 一词。不能分别为函数和过程设置默认权限。）

只能更改将由自己或自己所属的角色创建的对象的默认权限。这些权限可以全局设置（即，适用于当前数据库中创建的所有对象），也可以只针对指定模式中创建的对象设置。

如 GRANT 下所述，任何对象类型的默认权限通常会向对象所有者授予所有可授予的权限，也可能向PUBLIC 授予一些权限。不过，可以通过使用ALTER DEFAULT PRIVILEGES 更改全局默认权限来改变这种行为。

按模式指定的默认权限会叠加到该对象类型的全局默认权限之上。这意味着，如果某项权限是在全局范围内授予的（无论是默认授予，还是根据此前未指定模式的ALTER DEFAULT PRIVILEGES 命令授予的），就不能按模式将其撤销。按模式执行的REVOKE 只适用于抵消此前按模式执行的 GRANT 所产生的效果。

参数

target_role

当前角色所属的现有角色的名称。默认访问权限不会被继承，因此成员角色必须使用 SET ROLE 访问这些权限，或者必须为每个成员角色运行ALTER DEFAULT PRIVILEGES。如果省略 FOR ROLE，则假定为当前角色。

schema_name

一个现有模式的名称。如果指定，将修改以后在该模式中创建的对象的默认权限。如果省略 IN SCHEMA，则修改全局默认权限。在为模式设置权限时，不允许使用IN SCHEMA，因为模式不能嵌套。

role_name

要为其授予或撤销权限的现有角色的名称。此参数以及 *abbreviated_grant_or_revoke* 中的所有其他参数，均按GRANT 或 REVOKE 中的描述工作，只不过这里设置的是整类对象的权限，而不是特定的具名对象。

注解

使用 psql 的\ddp 命令获取现有默认权限分配的信息。权限值的含义与 GRANT 下对\dp 的说明相同。

如果你希望删除一个其默认权限已被修改的角色，则必须先撤销对其默认权限所做的更改，或者使用DROP OWNED BY 去除该角色的默认权限条目。

示例

为你后续在模式 myschema 中创建的所有表（以及视图）向所有人授予 SELECT 权限，并且也允许角色webuser 对它们执行 INSERT：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO webuser;
```

撤销上述操作，使后续创建的表不再拥有任何超出正常情况的权限：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM webuser;
```

移除通常会在函数上授予的公共 EXECUTE 权限，使后续由角色admin 创建的所有函数不再自动具有该权限：

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

但请注意，你不能用仅限单个模式的命令实现这一效果。下面这条命令不会产生作用，除非它是在撤销与之对应的GRANT：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

这是因为按模式设置的默认权限只能向全局设置增加权限，不能移除由全局设置授予的权限。

兼容性

在 SQL 标准中没有 ALTER DEFAULT PRIVILEGES 语句。

另见

GRANT, REVOKE

ALTER DOMAIN

ALTER DOMAIN — 更改一个域的定义

大纲

```
ALTER DOMAIN name
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
    { SET | DROP } NOT NULL
ALTER DOMAIN name
    ADD domain_constraint [ NOT VALID ]
ALTER DOMAIN name
    DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
     RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER DOMAIN name
    VALIDATE CONSTRAINT constraint_name
ALTER DOMAIN name
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER DOMAIN name
    RENAME TO new_name
ALTER DOMAIN name
    SET SCHEMA new_schema
```

描述

`ALTER DOMAIN` 更改一个现有域的定义。有以下几种子形式：

`SET/DROP DEFAULT`

这些形式为域设置或移除默认值。注意，默认值只适用于后续的 `INSERT` 命令；它们不会影响已存在于使用该域的表中的行。

`SET/DROP NOT NULL`

这些形式更改域是被标记为允许 NULL 值还是拒绝 NULL 值。只有当使用该域的列中不包含空值时，才能执行`SET NOT NULL`。

`ADD domain_constraint [ NOT VALID ]`

此形式使用与 CREATE DOMAIN 相同的语法向域添加新约束。向域添加新约束时，所有使用该域的列都会根据新添加的约束进行检查。可以使用`NOT VALID` 选项添加新约束来抑制这些检查；之后可以使用 `ALTER DOMAIN ... VALIDATE CONSTRAINT` 验证该约束。新插入或更新的行始终根据所有约束进行检查，即使这些约束标记为`NOT VALID`。`NOT VALID` 只能用于 `CHECK` 约束。

`DROP CONSTRAINT [ IF EXISTS ]`

这种形式删除域上的约束。如果指定了 `IF EXISTS` 而该约束不存在，则不会抛出错误，而是发出一条提示。

RENAME CONSTRAINT

这种形式更改域上某个约束的名称。

VALIDATE CONSTRAINT

这种形式验证一个先前以 NOT VALID 方式添加的约束，也就是说，它会核实该域类型的表列中的所有值都满足指定约束。

OWNER

这种形式将域的拥有者更改为指定用户。

RENAME

这种形式更改域的名称。

SET SCHEMA

这种形式更改域所在的模式。与该域关联的任何约束也会一并移动到新模式中。

要使用 ALTER DOMAIN，必须拥有该域。要更改域的模式，还必须在新模式上拥有CREATE 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在域所在模式上拥有CREATE 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建域完成的操作。不过，超级用户无论如何都可以更改任何域的所有权。）

参数

name

要修改的现有域的名称（可以是模式限定的）。

domain_constraint

该域的新约束。

constraint_name

要删除或重命名的现有约束名称。

NOT VALID

不验证现有存储的数据是否满足该约束。

CASCADE

自动删除依赖于该约束的对象，以及进一步依赖于这些对象的所有对象（见第 5.13 节）。

RESTRICT

如果存在任何依赖对象，则拒绝删除该约束。这是默认行为。

new_name

域的新名称。

new_constraint_name

约束的新名称。

new_owner

域的新拥有者的用户名。

new_schema

域的新模式。

注解

尽管 ALTER DOMAIN ADD CONSTRAINT 会尝试验证现有存储的数据是否满足新约束，但这种检查并非万无一失，因为该命令无法“看到”那些新插入或已更新但尚未提交的表行。如果存在并发操作可能插入不符合约束的数据的风险，正确做法是使用NOT VALID 选项添加该约束，提交该命令，等待所有在此次提交之前启动的事务结束，然后发出ALTER DOMAIN VALIDATE CONSTRAINT 来查找违反该约束的数据。这种方法是可靠的，因为一旦该约束被提交，所有新事务都保证会对域类型的新值强制执行该约束。

目前，如果指定的域或其任何派生域被数据库中任意表的容器类型列（即复合类型列、数组列或范围列）所使用，那么ALTER DOMAIN ADD CONSTRAINT、ALTER DOMAIN VALIDATE CONSTRAINT 和 ALTER DOMAIN SET NOT NULL 都会失败。将来它们应当会被改进，以便能够验证这类嵌套值上的新约束。

示例

向域添加一个 NOT NULL 约束：

```
ALTER DOMAIN zipcode SET NOT NULL;
```

从域中移除一个 NOT NULL 约束：

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

向域添加一个检查约束：

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

从域中移除一个检查约束：

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

重命名域上的一个检查约束：

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

将域移动到另一个模式中：

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

兼容性

`ALTER DOMAIN` 符合 SQL 标准，但 `OWNER`、`RENAME`、`SET SCHEMA` 和 `VALIDATE CONSTRAINT` 这些变体是 PostgreSQL 扩展；`ADD CONSTRAINT` 变体中的 `NOT VALID` 子句也是 PostgreSQL 扩展。

另见

CREATE DOMAIN, DROP DOMAIN

ALTER EVENT TRIGGER

ALTER EVENT TRIGGER — 更改事件触发器的定义

大纲

```
ALTER EVENT TRIGGER name DISABLE
ALTER EVENT TRIGGER name ENABLE [ REPLICA | ALWAYS ]
ALTER EVENT TRIGGER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER EVENT TRIGGER name RENAME TO new_name
```

描述

`ALTER EVENT TRIGGER` 更改现有事件触发器的属性。

要修改事件触发器，你必须是超级用户。

参数

name

要修改的现有事件触发器的名称。

new_owner

该事件触发器的新所有者的用户名。

new_name

该事件触发器的新名称。

`DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER`

这些形式配置事件触发器的触发行为。被禁用的触发器仍为系统所知，但在其触发事件发生时不会执行。另见 session_replication_role。

兼容性

SQL 标准中没有 `ALTER EVENT TRIGGER` 语句。

另见

CREATE EVENT TRIGGER, DROP EVENT TRIGGER

ALTER EXTENSION

ALTER EXTENSION — 更改扩展的定义

大纲

```
ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object
```

其中 *member_object* 是:

```
  ACCESS METHOD object_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  CONVERSION object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TYPE object_name |
  VIEW object_name
```

其中 *aggregate_signature* 是:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ...
 ]
```

描述

`ALTER EXTENSION` 更改已安装扩展的定义。其子形式如下:

`UPDATE`

这种形式将扩展更新到较新的版本。该扩展必须提供适当的更新脚本（或一系列脚本），能够将当前已安装版本修改为所请求的版本。

`SET SCHEMA`

这种形式将扩展的对象移入另一个模式。要使该命令成功，扩展必须是可重定位的。

ADD *member_object*

这种形式将一个现有对象加入扩展。这主要在扩展更新脚本中有用。该对象随后将被视为扩展的成员；特别是，只能通过删除该扩展来删除它。

DROP *member_object*

这种形式从扩展中移除一个成员对象。这主要在扩展更新脚本中有用。该对象不会被删除，而只是与扩展解除关联。

有关这些操作的更多信息，见第 38.16 节。

要使用 ALTER EXTENSION，你必须拥有该扩展。ADD/DROP 形式还要求你拥有被添加/移除对象的所有权。

参数

name

一个已安装扩展的名称。

new_version

扩展要更新到的新版本。它可以写成标识符或字符串常量。如果未指定，ALTER EXTENSION UPDATE 会尝试更新到该扩展控制文件中标明的默认版本。

new_schema

扩展的新模式。

object_name
aggregate_name
function_name
operator_name
procedure_name
routine_name

要添加到扩展中或从扩展中移除的对象名称。表、聚合、域、外部表、函数、操作符、操作符类、操作符族、过程、例程、序列、文本检索对象、类型和视图的名称可以带模式限定。

source_type

类型转换的源数据类型的名称。

target_type

类型转换的目标数据类型的名称。

argmode

函数、过程或聚合函数参数的模式：IN、OUT、INOUT 或 VARIADIC。如果省略，默认值是 IN。注意 ALTER EXTENSION 实际上并不关心 OUT 参数，因为确定函数身份只需要输入参数。因此，列出IN、INOUT 和 VARIADIC 参数就足够了。

argname

函数、过程或聚合函数参数的名称。注意ALTER EXTENSION 实际上并不关心参数名称，因为确定函数身份只需要参数数据类型。

argtype

函数、过程或聚合函数参数的数据类型。

left_type
right_type

操作符参数的数据类型（可带模式限定）。对于前缀或后缀操作符缺少的参数，写作NONE。

PROCEDURAL

这是一个噪声词。

type_name

该转换的数据类型的名称。

lang_name

该转换的语言的名称。

示例

将 hstore 扩展更新到版本 2.0：

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

将 hstore 扩展的模式更改为 utils：

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

将现有函数添加到 hstore 扩展中：

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

兼容性

ALTER EXTENSION 是一个 PostgreSQL 扩展。

另见

CREATE EXTENSION, DROP EXTENSION

ALTER FOREIGN DATA WRAPPER

ALTER FOREIGN DATA WRAPPER — 更改外部数据包装器的定义

大纲

```
ALTER FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ]) ]
ALTER FOREIGN DATA WRAPPER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER FOREIGN DATA WRAPPER name RENAME TO new_name
```

描述

`ALTER FOREIGN DATA WRAPPER` 更改外部数据包装器的定义。该命令的第一种形式更改外部数据包装器的支持函数或通用选项（至少需要一个子句）。第二种形式更改外部数据包装器的拥有者。

只有超级用户可以更改外部数据包装器。此外，只有超级用户才能成为外部数据包装器的拥有者。

参数

name

一个现有外部数据包装器的名称。

`HANDLER` *handler_function*

为外部数据包装器指定一个新的处理器函数。

`NO HANDLER`

用于指定该外部数据包装器不再具有一个处理器函数。

注意，使用没有处理器函数的外部数据包装器的外部表将无法访问。

`VALIDATOR` *validator_function*

为外部数据包装器指定一个新的验证器函数。

注意，该外部数据包装器本身或者依赖它的服务器、用户映射或外部表中预先存在的选项，根据新的验证器可能是无效的。PostgreSQL 不会检查这一点。用户必须在使用修改后的外部数据包装器之前自行确保这些选项正确。不过，在此`ALTER FOREIGN DATA WRAPPER` 命令中指定的任何选项，都会使用新的验证器进行检查。

`NO VALIDATOR`

用于指定该外部数据包装器不再具有验证器函数。

```
OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

更改外部数据包装器的选项。ADD、SET 和 DROP 指定要执行的动作。如果未显式指定操作，则假定为ADD。选项名称必须唯一；如果存在验证器函数，选项名称和值也会使用该外部数据包装器的验证器函数进行验证。

new_owner

该外部数据包装器的新拥有者的用户名。

new_name

该外部数据包装器的新名称。

示例

更改外部数据包装器 dbi，添加选项foo，删除bar：

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

将外部数据包装器 dbi 的验证器更改为 bob.myvalidator：

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

兼容性

ALTER FOREIGN DATA WRAPPER 符合 ISO/IEC 9075-9 (SQL/MED)，但HANDLER、VALIDATOR、OWNER TO 和 RENAME 子句属于扩展。

另见

CREATE FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER

ALTER FOREIGN TABLE

ALTER FOREIGN TABLE — 更改外部表的定义

大纲

```
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
```

其中 *action* 为以下之一:

```
    ADD [ COLUMN ] column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ALTER [ COLUMN ] column_name OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ])
    ADD table_constraint [ NOT VALID ]
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    SET WITH OIDS
    SET WITHOUT OIDS
    INHERIT parent_table
    NO INHERIT parent_table
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ])
```

描述

`ALTER FOREIGN TABLE` 更改现有外部表的定义。有几种子形式:

`ADD COLUMN`

该形式使用与 CREATE FOREIGN TABLE 相同的语法向外部表添加一个新列。与向普通表添加列的情况不同，底层存储不会发生任何变化；该操作只是声明现在可以通过外部表访问某个新列。

`DROP COLUMN [ IF EXISTS ]`

该形式从外部表中删除一列。如果该表之外的任何对象依赖于该列，例如视图，则需要指定 `CASCADE`。如果指定了 `IF EXISTS` 而该列不存在，则不会抛出错误，而是改为发出一条提示。

SET DATA TYPE

该形式更改外部表中一列的类型。同样，这不会影响任何底层存储：该操作只是更改 PostgreSQL 认为该列具有的类型。

SET/DROP DEFAULT

这些形式为列设置或移除默认值。默认值只会应用于后续的 INSERT 或 UPDATE 命令；它们不会导致表中已有的行发生变化。

SET/DROP NOT NULL

把一列标记为允许或不允许空值。

SET STATISTICS

该形式为后续 ANALYZE 操作设置每列的统计信息收集目标。详见 ALTER TABLE 的类似形式。

SET (*attribute_option* = *value* [, ...])
RESET (*attribute_option* [, ...])

该形式设置或重置每个属性的选项。详见 ALTER TABLE 的类似形式。

SET STORAGE

该形式设置列的存储模式。详见 ALTER TABLE 的类似形式。注意，除非该表的外部数据包装器选择理会该设置，否则存储模式不会产生效果。

ADD *table_constraint* [NOT VALID]

该形式使用与 CREATE FOREIGN TABLE 相同的语法向外部表添加一个新约束。目前只支持 CHECK 约束。

与向普通表添加约束的情况不同，不会采取任何措施来验证约束是否正确；相反，该操作只是声明外部表中的所有行都应被假定满足某个新条件。（参见CREATE FOREIGN TABLE 中的讨论。）如果约束被标记为 NOT VALID，则不会假定其成立，而只会记录下来供将来使用。

VALIDATE CONSTRAINT

该形式把先前被标记为 NOT VALID 的约束标记为有效。不会采取任何措施来验证该约束，但未来的查询会假定它成立。

DROP CONSTRAINT [IF EXISTS]

该形式删除外部表上指定的约束。如果指定了 IF EXISTS 而该约束不存在，则不会抛出错误，而是改为发出一条提示。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

这些形式配置属于外部表的触发器的触发。更多详情参见 ALTER TABLE 的类似形式。

SET WITH OIDS

该形式向表添加一个 oid 系统列（参见第 5.4 节）。如果表已有 OID，则不执行任何操作。除非表的外部数据包装器支持 OID，否则该列只会读作零。

请注意，这并不等同于ADD COLUMN oid oid；后者会添加一个碰巧命名为 oid 的普通列，而不是系统列。

SET WITHOUT OIDS

该形式从表中移除 oid 系统列。这完全等同于 DROP COLUMN oid RESTRICT，但如果表中已经没有oid 列，则不会报错。

INHERIT *parent_table*

该形式将目标外部表添加为指定父表的新子表。更多详情参见 ALTER TABLE 的类似形式。

NO INHERIT *parent_table*

该形式把目标外部表从指定父表的子表列表中移除。

OWNER

该形式把外部表的拥有者更改为指定用户。

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

更改外部表或其某一列的选项。ADD、SET 和 DROP 指定要执行的动作。如果没有显式指定操作，则默认是 ADD。不允许重复的选项名（不过表选项和列选项同名是可以的）。选项名和值也会通过外部数据包装器库进行验证。

RENAME

RENAME 形式更改外部表的名称，或者更改其中某一列的名称。

SET SCHEMA

该形式把外部表移动到另一个模式中。

除了 RENAME 和 SET SCHEMA 之外，所有动作都可以组合在一个包含多项修改的列表中一并应用。例如，可以在一条命令中添加多个列和/或更改多个列的类型。

如果命令写成 ALTER FOREIGN TABLE IF EXISTS ...而外部表不存在，则不会抛出错误。这种情况下会发出一个提示。

要使用 ALTER FOREIGN TABLE，必须拥有该表。要更改外部表的模式，还必须在新模式上拥有CREATE 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在表所在模式上拥有CREATE 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建该表完成的操作。不过，超级用户无论如何都可以更改任何表的所有权。）要添加列或更改列类型，还必须在数据类型上拥有USAGE 权限。

参数

name

要修改的现有外部表的名称（可以带模式限定）。如果在表名前指定了 ONLY，则只修改该表。如果未指定ONLY，则该表及其所有后代表（如果有）都会被修改。也可以在表名后指定 *，显式表示包括后代表。

column_name

新列或现有列的名称。

new_column_name

现有列的新名称。

new_name

表的新名称。

data_type

新列的数据类型，或现有列的新数据类型。

table_constraint

该外部表的新表约束。

constraint_name

要删除的现有约束的名称。

CASCADE

自动删除依赖于被删除列或约束的对象（例如引用该列的视图），以及所有进一步依赖于这些对象的对象（见第 5.13 节）。

RESTRICT

如果存在任何依赖对象，则拒绝删除该列或约束。这是默认行为。

trigger_name

要禁用或启用的单个触发器的名称。

ALL

禁用或启用属于该外部表的所有触发器。（如果其中任何触发器是内部生成的触发器，则需要超级用户权限。核心系统不会向外部表添加这类触发器，但附加代码可能会这么做。）

USER

禁用或启用属于该外部表的所有触发器，但内部生成的触发器除外。

parent_table

要与该外部表关联或解除关联的父表。

new_owner

该表的新拥有者的用户名。

new_schema

该表将被移动到的模式名称。

注解

关键字 COLUMN 不影响语义，可以省略。

使用 ADD COLUMN 或 DROP COLUMN 添加或移除列，添加NOT NULL 或 CHECK 约束，或使用 SET DATA TYPE 更改列类型时，不会检查与外部服务器的一致性。用户有责任确保表定义与远端一致。

关于有效参数的进一步说明，请参见 CREATE FOREIGN TABLE。

示例

要把一列标记为非空：

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

要更改外部表的选项：

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2 'value2', DROP
 opt3);
```

兼容性

ADD、DROP 和 SET DATA TYPE 形式符合 SQL 标准。其他形式是 PostgreSQL 对 SQL 标准的扩展。另外，在单条ALTER FOREIGN TABLE 命令中指定多个修改也是一种扩展。

ALTER FOREIGN TABLE DROP COLUMN 可用于删除外部表的唯一一列，从而留下一个零列的表。这是 SQL 的一种扩展，因为 SQL 不允许零列外部表。

另见

CREATE FOREIGN TABLE, DROP FOREIGN TABLE

ALTER FUNCTION

ALTER FUNCTION — 更改函数的定义

大纲

```
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name
```

其中 *action* 为以下之一:

```
    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE
    [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST execution_cost
    ROWS result_rows
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

描述

`ALTER FUNCTION` 更改函数的定义。

要使用 `ALTER FUNCTION`，必须拥有该函数。要更改函数的模式，还必须在新模式上拥有 `CREATE` 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在函数所在模式上拥有`CREATE` 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建函数完成的操作。不过，超级用户无论如何都可以更改任何函数的所有权。）

参数

name

现有函数的名称（可以用模式限定）。如果没有指定参数列表，这个名称必须在其模式中唯一。

argmode

参数的模式：IN、OUT、INOUT 或 VARIADIC。如果省略，默认为IN。注意 ALTER FUNCTION 实际上并不关心任何 OUT 参数，因为确定函数身份只需要输入参数。因此，只列出IN、INOUT 和 VARIADIC 参数就足够了。

argname

参数的名称。注意 ALTER FUNCTION 实际上并不关心参数名，因为确定函数身份只需要参数数据类型。

argtype

函数参数的数据类型（如果有，可以用模式限定）。

new_name

该函数的新名称。

new_owner

该函数的新拥有者。注意，如果该函数被标记为 SECURITY DEFINER，随后它将以新拥有者的身份执行。

new_schema

该函数的新模式。

extension_name

该函数所依赖的扩展名称。

```
CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT
```

CALLED ON NULL INPUT 将该函数改为在其部分或全部参数为 null 时仍会被调用。RETURNS NULL ON NULL INPUT 或 STRICT 将该函数改为只要任一参数为 null 就不调用，而是自动假定结果为 null。详见 CREATE FUNCTION。

```
IMMUTABLE
STABLE
VOLATILE
```

将该函数的易变性更改为指定的设置。详见 CREATE FUNCTION。

```
[ EXTERNAL ] SECURITY INVOKER
[ EXTERNAL ] SECURITY DEFINER
```

更改该函数是否为安全性定义者。关键字 EXTERNAL 为了符合 SQL 标准而被忽略。关于这一能力的更多信息，参见 CREATE FUNCTION。

```
PARALLEL
```

更改该函数是否被视为对并行执行安全。详见 CREATE FUNCTION。

`LEAKPROOF`

更改该函数是否被视为防泄漏的。关于这一能力的更多信息，参见 CREATE FUNCTION。

`COST` *`execution_cost`*

更改该函数的估计执行代价。详见 CREATE FUNCTION。

`ROWS` *`result_rows`*

更改集合返回函数估计返回的行数。详见 CREATE FUNCTION。

`configuration_parameter`
`value`

添加或更改在调用该函数时要赋给某个配置参数的值。如果 *`value`* 是 `DEFAULT`，或者等效地使用了`RESET`，则会移除函数局部设置，这样该函数就会使用其环境中现有的值执行。使用`RESET ALL` 可以清除所有函数局部设置。`SET FROM CURRENT` 会把执行 `ALTER FUNCTION` 时该参数的当前值保存为进入函数时要应用的值。

关于允许的参数名和值的更多信息，参见SET 和第 19 章。

`RESTRICT`

为符合 SQL 标准而被忽略。

示例

要将类型 `integer` 的函数 `sqrt` 重命名为 `square_root`：

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

要将类型 `integer` 的函数 `sqrt` 的拥有者改为 `joe`：

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

要将类型 `integer` 的函数 `sqrt` 的模式改为 `maths`：

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

要将类型 `integer` 的函数 `sqrt` 标记为依赖于扩展 `mathlib`：

```
ALTER FUNCTION sqrt(integer) DEPENDS ON EXTENSION mathlib;
```

要调整自动为函数设置的搜索路径：

```
ALTER FUNCTION check_password(text) SET search_path = admin, pg_temp;
```

要禁用自动为函数设置的 search_path：

```
ALTER FUNCTION check_password(text) RESET search_path;
```

现在该函数将使用其调用者所使用的搜索路径执行。

兼容性

该语句与 SQL 标准中的 ALTER FUNCTION 语句部分兼容。标准允许修改函数的更多属性，但不提供重命名函数、将函数设为安全性定义者、为函数附加配置参数值，或者更改函数的拥有者、模式或易变性的能力。标准还要求使用RESTRICT 关键字，而它在 PostgreSQL 中是可选的。

另见

CREATE FUNCTION, DROP FUNCTION, ALTER PROCEDURE, ALTER ROUTINE

ALTER GROUP

ALTER GROUP — 更改角色名称或成员资格

大纲

```
ALTER GROUP role_specification ADD USER user_name [, ... ]
ALTER GROUP role_specification DROP USER user_name [, ... ]
```

其中 *role_specification* 可以是：

```
    role_name
  | CURRENT_USER
  | SESSION_USER
```

```
ALTER GROUP group_name RENAME TO new_name
```

描述

`ALTER GROUP` 更改用户组的属性。这是一个已废弃的命令，但出于向后兼容的考虑仍然接受它，因为组（以及用户）已经被更一般的角色概念所取代。

前两个变体将用户加入一个组，或将其从组中移除。（为此目的，任何角色都可以充当“用户”或“组”。）这些变体实际上等效于对名为“组”的角色授予或撤销成员资格，因此更推荐使用 GRANT 或 REVOKE。

第三个变体更改组的名称。这与使用 ALTER ROLE 重命名该角色完全等效。

参数

group_name

要修改的组（角色）的名称。

user_name

要加入该组或从该组中移除的用户（角色）。这些用户必须已经存在；`ALTER GROUP` 不会创建或删除用户。

new_name

该组的新名称。

示例

向组中添加用户：

```
ALTER GROUP staff ADD USER karl, john;
```

从组中移除用户：

```
ALTER GROUP workers DROP USER beth;
```

兼容性

SQL 标准中没有 `ALTER GROUP` 语句。

另见

GRANT, REVOKE, ALTER ROLE

ALTER INDEX

ALTER INDEX — 更改索引的定义

大纲

```
ALTER INDEX [ IF EXISTS ] name RENAME TO new_name
ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name
ALTER INDEX name ATTACH PARTITION index_name
ALTER INDEX name DEPENDS ON EXTENSION extension_name
ALTER INDEX [ IF EXISTS ] name SET ( storage_parameter [= value] [, ... ] )
ALTER INDEX [ IF EXISTS ] name RESET ( storage_parameter [, ... ] )
ALTER INDEX [ IF EXISTS ] name ALTER [ COLUMN ] column_number
    SET STATISTICS integer
ALTER INDEX ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
```

描述

`ALTER INDEX` 更改现有索引的定义。有几种子形式：

`RENAME`

`RENAME` 形式更改该索引的名称。如果该索引与某个表约束（即 `UNIQUE`、`PRIMARY KEY` 或 `EXCLUDE`）相关联，该约束也会一并重命名。这不会影响已存储的数据。

`SET TABLESPACE`

这种形式将该索引的表空间改为指定表空间，并把与该索引关联的数据文件移动到新表空间中。要更改索引的表空间，你必须拥有该索引，并且对新表空间具有 `CREATE` 权限。使用 `ALL IN TABLESPACE` 形式可以移动当前数据库中某个表空间内的全部索引；该形式会锁定所有待移动的索引，然后逐个移动。该形式还支持`OWNED BY`，这时只会移动属于指定角色的索引。如果指定了`NOWAIT` 选项，则在命令无法立即获取所需全部锁时会失败。注意，该命令不会移动系统目录；如果需要这样做，应改用`ALTER DATABASE` 或显式调用 `ALTER INDEX`。另见 CREATE TABLESPACE。

`ATTACH PARTITION`

使指定索引附加到正在更改的索引上。指定索引必须位于包含正在更改索引的表的某个分区上，并且定义等效。附加的索引不能单独删除；如果其父索引被删除，它也会自动删除。

`DEPENDS ON EXTENSION`

该形式将索引标记为依赖扩展，因此扩展被删除时索引也会自动删除。

`SET ( storage_parameter [= value] [, ... ] )`

这种形式更改该索引的一个或多个索引方法相关的存储参数。可用参数的详情请见 CREATE INDEX。请注意，此命令不会立即修改索引内容；根据参数不同，你可能需要使用REINDEX 重建索引，才能获得期望的效果。

`RESET ( storage_parameter [, ... ] )`

这种形式将一个或多个索引方法相关的存储参数重置为默认值。与 `SET` 一样，可能仍需要执行一次`REINDEX` 才能让该索引被完全更新。

`ALTER [ COLUMN ] column_number SET STATISTICS integer`

这种形式为后续的 ANALYZE 操作设置每列的统计信息收集目标，但只能用于定义为表达式的索引列。由于表达式没有唯一名称，我们通过索引列的顺序号来引用它们。该目标可以设置为 0 到 10000 之间的值；或者设为-1，以恢复使用系统默认统计信息目标（default_statistics_target）。关于 PostgreSQL 查询规划器如何使用统计信息的更多信息，请参阅第 14.2 节。

参数

`IF EXISTS`

如果该索引不存在，则不要抛出错误。这种情况下会发出一条提示。

column_number

该顺序号表示该索引列从左到右的位置。

name

要修改的现有索引名称（可以是模式限定的）。

new_name

该索引的新名称。

tablespace_name

该索引将被移动到的表空间。

extension_name

该索引所依赖的扩展的名称。

storage_parameter

一个索引方法相关的存储参数的名称。

value

一个索引方法相关的存储参数的新值。根据具体参数，它可能是数字或一个词。

注解

这些操作同样可以使用 ALTER TABLE 完成。实际上，`ALTER INDEX` 只是 `ALTER TABLE` 中适用于索引的那些形式的别名。

以前曾有一种 `ALTER INDEX OWNER` 变体，但现在会被忽略（同时发出警告）。索引的拥有者不能不同于其所属表的拥有者。更改表的拥有者时，索引的拥有者也会自动更改。

不允许更改系统目录索引的任何部分。

示例

要重命名现有索引：

```
ALTER INDEX distributors RENAME TO suppliers;
```

要将索引移到另一个表空间：

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

要更改索引的填充因子（假设该索引方法支持 fillfactor 参数）：

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

要为表达式索引设置统计信息收集目标：

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS 1000;
```

兼容性

`ALTER INDEX` 是一种 PostgreSQL 扩展。

另见

CREATE INDEX, REINDEX

ALTER LANGUAGE

ALTER LANGUAGE — 更改一种过程语言的定义

大纲

```
ALTER [ PROCEDURAL ] LANGUAGE name RENAME TO new_name
ALTER [ PROCEDURAL ] LANGUAGE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

描述

`ALTER LANGUAGE` 更改一种过程语言的定义。唯一的功能是重命名该语言或为其指定新的所有者。要使用 `ALTER LANGUAGE`，你必须是超级用户或者该语言的所有者。

参数

name

语言的名称

new_name

该语言的新名称

new_owner

该语言的新所有者

兼容性

SQL 标准中没有 `ALTER LANGUAGE` 语句。

另见

CREATE LANGUAGE, DROP LANGUAGE

ALTER LARGE OBJECT

ALTER LARGE OBJECT — 更改一个大对象的定义

大纲

```
ALTER LARGE OBJECT large_object_oid OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

描述

`ALTER LARGE OBJECT` 更改一个大对象的定义。

要使用 `ALTER LARGE OBJECT`，必须拥有该大对象。要更改所有者，还必须是新所有者角色的直接或间接成员。（不过，超级用户无论如何都可以更改任何大对象。）目前唯一的功能是指定新的所有者，因此两项限制始终适用。

参数

large_object_oid

要修改的大对象的 OID

new_owner

该大对象的新拥有者

兼容性

SQL 标准中没有 `ALTER LARGE OBJECT` 语句。

另见

第 35 章

ALTER MATERIALIZED VIEW

ALTER MATERIALIZED VIEW — 更改一个物化视图的定义

大纲

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    action [, ... ]
ALTER MATERIALIZED VIEW name
    DEPENDS ON EXTENSION extension_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME TO new_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER MATERIALIZED VIEW ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
```

其中 *action* 可以是以下之一:

```
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET TABLESPACE new_tablespace
    SET ( storage_parameter [= value] [, ... ] )
    RESET ( storage_parameter [, ... ] )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

描述

`ALTER MATERIALIZED VIEW` 更改一个现有物化视图的各种辅助属性。

要使用 `ALTER MATERIALIZED VIEW`，必须拥有该物化视图。要更改物化视图的模式，还必须在新模式上拥有`CREATE` 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在物化视图所在模式上拥有`CREATE` 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建物化视图完成的操作。不过，超级用户无论如何都可以更改任何视图的所有权。）

`DEPENDS ON EXTENSION` 形式将物化视图标记为依赖扩展，因此扩展被删除时物化视图会自动删除。

`ALTER MATERIALIZED VIEW` 可用的语句子形式和操作是 `ALTER TABLE` 可用项的子集，用于物化视图时含义相同。详情请参见ALTER TABLE 的说明。

参数

name

一个现有物化视图的名称（可以是模式限定的）。

column_name

新列或现有列的名称。

extension_name

物化视图所依赖的扩展名称。

new_column_name

现有列的新名称。

new_owner

该物化视图的新拥有者的用户名。

new_name

该物化视图的新名称。

new_schema

该物化视图的新模式。

示例

把物化视图 `foo` 重命名为 `bar`：

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

兼容性

`ALTER MATERIALIZED VIEW` 是一种 PostgreSQL 扩展。

另见

CREATE MATERIALIZED VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

ALTER OPERATOR

ALTER OPERATOR — 更改一个操作符的定义

大纲

```
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    SET SCHEMA new_schema

ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    SET ( {  RESTRICT = { res_proc | NONE }
           | JOIN = { join_proc | NONE }
         } [, ... ] )
```

描述

`ALTER OPERATOR` 更改一个操作符的定义。

要使用 `ALTER OPERATOR`，必须拥有该操作符。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在操作符所在模式上拥有`CREATE` 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建操作符完成的操作。不过，超级用户无论如何都可以更改任何操作符的所有权。）

参数

name

一个现有操作符的名称（可以是模式限定的）。

left_type

该操作符左操作数的数据类型；如果该操作符没有左操作数，请写 `NONE`。

right_type

操作符右操作数的数据类型；如果操作符没有右操作数，写作`NONE`。

new_owner

该操作符的新拥有者。

new_schema

该操作符的新模式。

res_proc

该操作符的限制选择率估算函数；写成 `NONE` 可移除现有的选择率估算器。

join_proc

该操作符的连接选择率估算函数；写成 NONE 可移除现有的选择率估算器。

示例

更改自定义操作符的拥有者，该操作符为a @@ b，类型为text：

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

更改自定义操作符的限制和连接选择率估计函数，操作符为a && b，类型为int[]：

```
ALTER OPERATOR && (_int4, _int4) SET (RESTRICT = _int_contsel, JOIN = _int_contjoinsel);
```

兼容性

在 SQL 标准中没有 ALTER OPERATOR 语句。

另见

CREATE OPERATOR, DROP OPERATOR

ALTER OPERATOR CLASS

ALTER OPERATOR CLASS — 更改一个操作符类的定义

大纲

```
ALTER OPERATOR CLASS name USING index_method
    RENAME TO new_name

ALTER OPERATOR CLASS name USING index_method
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR CLASS name USING index_method
    SET SCHEMA new_schema
```

描述

`ALTER OPERATOR CLASS` 更改一个操作符类的定义。

要使用 `ALTER OPERATOR CLASS`，必须拥有该操作符类。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在操作符类所在模式上拥有`CREATE` 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建操作符类完成的操作。不过，超级用户无论如何都可以更改任何操作符类的所有权。）

参数

name

一个现有操作符类的名称（可以是模式限定的）。

index_method

该操作符类所对应的索引方法的名称。

new_name

该操作符类的新名称。

new_owner

该操作符类的新拥有者。

new_schema

该操作符类的新模式。

兼容性

SQL 标准中没有 `ALTER OPERATOR CLASS` 语句。

另见

CREATE OPERATOR CLASS, DROP OPERATOR CLASS, ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY — 更改一个操作符族的定义

大纲

```
ALTER OPERATOR FAMILY name USING index_method ADD
  {  OPERATOR strategy_number operator_name ( op_type, op_type )
             [ FOR SEARCH | FOR ORDER BY sort_family_name ]
   | FUNCTION support_number [ ( op_type [ , op_type ] ) ]
             function_name [ ( argument_type [, ...] ) ]
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method DROP
  {  OPERATOR strategy_number ( op_type [ , op_type ] )
   | FUNCTION support_number ( op_type [ , op_type ] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method
    RENAME TO new_name

ALTER OPERATOR FAMILY name USING index_method
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR FAMILY name USING index_method
    SET SCHEMA new_schema
```

描述

`ALTER OPERATOR FAMILY` 更改一个操作符族的定义。你可以向该族添加操作符和支持函数、从该族中移除它们，或者更改该族的名称或拥有者。

使用 `ALTER OPERATOR FAMILY` 向一个族中添加操作符和支持函数时，它们并不属于该族内任何特定的操作符类，而只是作为该族中的“松散”成员存在。这表明这些操作符和函数与该族的语义兼容，但并不是任何特定索引正确工作所必需的。（那些确实必需的操作符和函数应当声明为某个操作符类的一部分；参见 CREATE OPERATOR CLASS。）PostgreSQL 允许随时从一个族中删除松散成员，但如果不删除整个类以及依赖它的任何索引，就不能删除操作符类的成员。通常，单一数据类型的操作符和函数属于操作符类，因为支持该特定数据类型上的索引需要它们，而跨数据类型的操作符和函数则作为该族中的松散成员。

要使用 `ALTER OPERATOR FAMILY`，你必须是超级用户。（之所以这样限制，是因为错误的操作符族定义可能会使服务器混乱，甚至导致崩溃。）

`ALTER OPERATOR FAMILY` 目前不会检查操作符族定义是否包括该索引方法所要求的全部操作符和函数，也不会检查这些操作符和函数是否构成一个自洽的集合。定义一个合法的操作符族是用户的责任。

更多信息请参见第 38.15 节。

参数

name

一个现有操作符族的名称（可以是模式限定的）。

index_method

该操作符族所对应的索引方法名称。

strategy_number

与该操作符族关联的操作符的索引方法策略号。

operator_name

与该操作符族关联的操作符名称（可以是模式限定的）。

op_type

在 OPERATOR 子句中，操作符的操作数数据类型，或使用NONE 表示左一元或右一元操作符。与 CREATE OPERATOR CLASS 中的对应语法不同，必须始终指定操作数数据类型。

在 ADD FUNCTION 子句中，如果函数要支持的操作数数据类型不同于函数的输入数据类型，则指定这些数据类型。对于 B-树比较函数和 hash 函数，不需要指定*op_type*，因为函数的输入数据类型始终是应使用的正确类型。对于 B-树排序支持函数以及 GiST、SP-GiST 和 GIN 操作符类中的所有函数，则必须指定该函数要配合使用的操作数数据类型。

在 DROP FUNCTION 子句中，必须指定该函数意图支持的操作数数据类型。

sort_family_name

一个现有 btree 操作符族的名称（可以是模式限定的），它描述与某个排序操作符关联的排序顺序。

如果既没有指定 FOR SEARCH，也没有指定 FOR ORDER BY，则默认值是FOR SEARCH。

support_number

与该操作符族关联的函数的索引方法支持函数编号。

function_name

作为该操作符族索引方法支持函数的函数名称（可以是模式限定的）。如果没有指定参数列表，则该名称在其模式中必须唯一。

argument_type

该函数的参数数据类型。

new_name

该操作符族的新名称。

new_owner

该操作符族的新拥有者。

new_schema

该操作符族所在的新模式。

OPERATOR 和 FUNCTION 子句可以以任何顺序出现。

注解

注意，DROP 语法只通过策略号或支持函数编号以及输入数据类型来指定操作符族中的“槽位”，不会提及占用该槽位的操作符或函数名称。另外，对于DROP FUNCTION，要指定的类型是该函数意图支持的输入数据类型；对于 GiST、SP-GiST 和 GIN 索引，这些类型可能与该函数实际的输入参数类型毫无关系。

因为索引机制在使用函数之前不会检查函数的访问权限，将一个函数或者操作符包括在一个操作符族中相当于在其上授予公共执行权限。这对操作符族中很有用的这类函数来说通常不成问题。

不应由 SQL 函数定义这些操作符。SQL 函数很可能会被内联到调用查询中，从而阻止优化器识别该查询与索引匹配。

在 PostgreSQL 8.4 之前，OPERATOR 子句可以包含 RECHECK 选项。现在不再支持该选项，因为索引操作符是否“有损”会在运行时动态确定。这样可以高效处理操作符可能有损也可能无损的情况。

示例

下列示例命令向一个已经包含数据类型 int4 和 int2 的 B-树操作符类的操作符族中添加跨数据类型的操作符和支持函数。

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 对 int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 对 int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

要再次移除这些条目：

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 对 int2
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
  FUNCTION 1 (int4, int2) ,

  -- int2 对 int4
  OPERATOR 1 (int2, int4) ,
  OPERATOR 2 (int2, int4) ,
  OPERATOR 3 (int2, int4) ,
  OPERATOR 4 (int2, int4) ,
  OPERATOR 5 (int2, int4) ,
  FUNCTION 1 (int2, int4) ;
```

兼容性

在 SQL 标准中没有 `ALTER OPERATOR FAMILY` 语句。

另见

CREATE OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

ALTER POLICY

ALTER POLICY — 更改行级安全策略的定义

大纲

```
ALTER POLICY name ON table_name RENAME TO new_name

ALTER POLICY name ON table_name
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

描述

`ALTER POLICY` 更改一条现有行级安全策略的定义。请注意，`ALTER POLICY` 只允许修改策略适用的角色集合，以及`USING` 和 `WITH CHECK` 表达式。若要更改策略的其他属性，例如它所适用的命令，或者它是宽松策略还是限制性策略，则必须删除该策略并重新创建。

要使用 `ALTER POLICY`，你必须拥有该策略所适用的表。

在 `ALTER POLICY` 的第二种形式中，角色列表、*`using_expression`* 和 *`check_expression`* 在被指定时会分别独立替换。省略这些子句中的任意一个时，策略中对应的部分保持不变。

参数

`name`

要更改的现有策略名称。

`table_name`

该策略所在表的名称（可选模式限定）。

`new_name`

该策略的新名称。

`role_name`

该策略适用的角色。可以一次指定多个角色。要将该策略应用于所有角色，可使用`PUBLIC`。

`using_expression`

该策略的 `USING` 表达式。详见 CREATE POLICY。

`check_expression`

该策略的 `WITH CHECK` 表达式。详见 CREATE POLICY。

兼容性

`ALTER POLICY` 是 PostgreSQL 扩展。

另见

CREATE POLICY, DROP POLICY

ALTER PROCEDURE

ALTER PROCEDURE — 更改过程的定义

大纲

```
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name
```

其中 *action* 为以下之一:

```
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

描述

`ALTER PROCEDURE` 更改过程的定义。

要使用 `ALTER PROCEDURE`，必须拥有该过程。要更改过程的模式，还必须在新模式上拥有 `CREATE` 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在过程所在模式上拥有`CREATE` 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建过程完成的操作。不过，超级用户无论如何都可以更改任何过程的所有权。）

参数

name

现有过程的名称（可带模式限定）。如果未指定参数列表，该名称在其模式中必须唯一。

argmode

参数的模式：`IN` 或 `VARIADIC`。如果省略，默认值为`IN`。

argname

参数的名称。请注意，`ALTER PROCEDURE` 实际上不会关注参数名称，因为确定过程身份只需要参数数据类型。

argtype

过程参数的数据类型（如果有，可带模式限定）。

new_name

该过程的新名称。

new_owner

该过程的新拥有者。注意，如果该过程被标记为SECURITY DEFINER，随后它将以新拥有者的身份执行。

new_schema

该过程的新模式。

extension_name

该过程所依赖的扩展名称。

```
[ EXTERNAL ] SECURITY INVOKER
[ EXTERNAL ] SECURITY DEFINER
```

更改该过程是否为安全性定义者。关键字 EXTERNAL 为符合 SQL 标准而被忽略。关于这一能力的更多信息，参见 CREATE PROCEDURE。

configuration_parameter
value

添加或更改在调用该过程时要赋给某个配置参数的值。如果 *value* 为 DEFAULT，或者等效地使用了RESET，则会移除过程局部设置，这样该过程就会使用其环境中的值执行。使用 RESET ALL 可以清除所有过程局部设置。SET FROM CURRENT 会将执行 ALTER PROCEDURE 时该参数的当前值保存为进入该过程时要应用的值。

关于允许的参数名和值的更多信息，请参见SET 和第 19 章。

RESTRICT

为符合 SQL 标准而被忽略。

示例

要将具有两个 integer 类型参数的过程 insert_data 重命名为 insert_record：

```
ALTER PROCEDURE insert_data(integer, integer) RENAME TO insert_record;
```

要将具有两个 integer 类型参数的过程 insert_data 的拥有者改为 joe：

```
ALTER PROCEDURE insert_data(integer, integer) OWNER TO joe;
```

要将具有两个 integer 类型参数的过程 insert_data 的模式改为 accounting：

```
ALTER PROCEDURE insert_data(integer, integer) SET SCHEMA accounting;
```

要将过程 insert_data(integer, integer) 标记为依赖于扩展 myext：

```
ALTER PROCEDURE insert_data(integer, integer) DEPENDS ON EXTENSION myext;
```

要调整会为过程自动设置的搜索路径：

```
ALTER PROCEDURE check_password(text) SET search_path = admin, pg_temp;
```

要禁用过程的 search_path 自动设置：

```
ALTER PROCEDURE check_password(text) RESET search_path;
```

现在该过程将使用其调用者所使用的任何搜索路径执行。

兼容性

该语句与 SQL 标准中的 ALTER PROCEDURE 语句部分兼容。标准允许修改过程的更多属性，但不提供重命名过程、将过程设为安全性定义者、为过程附加配置参数值，或者更改过程的拥有者、模式或易变性的能力。标准还要求使用RESTRICT 关键字，而它在PostgreSQL 中是可选的。

另见

CREATE PROCEDURE, DROP PROCEDURE, ALTER FUNCTION, ALTER ROUTINE

ALTER PUBLICATION

ALTER PUBLICATION — 修改发布的定义

大纲

```
ALTER PUBLICATION name ADD TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name SET TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name DROP TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name SET ( publication_parameter [= value] [, ... ] )
ALTER PUBLICATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER PUBLICATION name RENAME TO new_name
```

描述

ALTER PUBLICATION 命令可更改发布的属性。

前三种形式会更改哪些表属于该发布。SET TABLE 子句会用指定列表替换发布中的表列表。ADD TABLE 和 DROP TABLE 子句会向发布添加或从发布移除一个或多个表。请注意，向已被订阅的发布添加表，需要在订阅端执行ALTER SUBSCRIPTION ... REFRESH PUBLICATION 操作后才会生效。

提要中列出的第四种形式可更改 CREATE PUBLICATION 中指定的所有发布属性。命令中未提及的属性会保留其先前的设置。

其余几种形式用于更改发布的所有者和名称。

要使用 ALTER PUBLICATION，必须拥有该发布。向发布添加表还要求拥有该表。要更改所有者，还必须是新所有者角色的直接或间接成员。新所有者必须在数据库上拥有CREATE 权限。此外，FOR ALL TABLES 发布的新所有者必须是超级用户。不过，超级用户无论这些限制如何都可以更改发布的所有权。

参数

name

要更改其定义的现有发布名称。

table_name

现有表的名称。如果在表名之前指定 ONLY，则只影响该表。如果未指定ONLY，则影响该表及其所有后代表（如果有）。也可以在表名后指定*，以明确表示包含后代表。

SET (*publication_parameter* [= *value*] [, ...])

此子句更改最初由 CREATE PUBLICATION 设置的发布参数。详情请参见该命令。

new_owner

新发布所有者的用户名。

new_name

发布的新名称。

示例

将发布修改为只发布删除和更新：

```
ALTER PUBLICATION noinsert SET (publish = 'update, delete');
```

向发布中添加一些表：

```
ALTER PUBLICATION mypublication ADD TABLE users (user_id, firstname), departments;
```

兼容性

`ALTER PUBLICATION` 是 PostgreSQL 的一个扩展。

另见

CREATE PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

ALTER ROLE

ALTER ROLE — 更改数据库角色

大纲

```
ALTER ROLE role_specification [ WITH ] option [ ... ]
```

其中 *option* 可以是:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'

ALTER ROLE name RENAME TO new_name

ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_
parameter { TO | = } { value | DEFAULT }
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_
parameter FROM CURRENT
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] RESET configuration_
parameter
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL
```

其中 *role_specification* 可以是:

```
    role_name
  | CURRENT_USER
  | SESSION_USER
```

描述

ALTER ROLE 更改 PostgreSQL 角色的属性。

语法概要中列出的第一种形式可以更改许多可在 CREATE ROLE 中指定的角色属性。（涵盖了所有可能的属性，但没有用于添加或移除成员资格的选项；为此请使用GRANT 和 REVOKE。）命令中未提及的属性保持原有设置不变。数据库超级用户可以为任何角色更改这些设置。拥有 CREATEROLE 权限的角色可以更改除 SUPERUSER、REPLICATION 和 BYPASSRLS 之外的任何这些设置，但只能针对非超级用户且非复制角色。普通角色只能更改自己的密码。

第二种形式更改角色的名称。数据库超级用户可以重命名任何角色。拥有 CREATEROLE 权限的角色可以重命名非超级用户角色。当前会话用户不能被重命名。（如果需要这样做，请以其他用户连接。）由于MD5 加密的密码使用角色名作为密码学盐，重命名角色会清除其密码（如果密码使用MD5 加密）。

其余形式更改角色针对配置变量的会话默认值，可以针对所有数据库，也可以在指定 IN DATABASE 子句时仅针对指定数据库中的会话。如果指定的是 ALL 而不是角色名，就会更改所有角色的该项设置。将 ALL 与 IN DATABASE 一起使用，实际上与命令 ALTER DATABASE ... SET ...的效果相同。

当该角色随后启动新会话时，指定值将成为会话默认值，覆盖postgresql.conf 中的设置或从 postgres 命令行接收的设置。这只发生在登录时；执行 SET ROLE 或 SET SESSION AUTHORIZATION 不会导致设置新的配置值。为所有数据库设置的设置会被附加到角色的数据库特定设置覆盖。特定数据库或特定角色的设置会覆盖为所有角色设置的设置。

超级用户可以更改任何人的会话默认值。拥有 CREATEROLE 权限的角色可以更改非超级用户角色的默认值。普通角色只能为自己设置默认值。某些配置变量不能以这种方式设置，或者只能由超级用户执行命令来设置。只有超级用户可以更改所有数据库中所有角色的设置。

参数

name

将要修改其属性的角色名称。

CURRENT_USER

修改当前用户，而不是显式标识的角色。

SESSION_USER

修改当前会话用户，而不是显式标识的角色。

SUPERUSER
NOSUPERUSER
CREATEDB
NOCREATEDB
CREATEROLE
NOCREATEROLE
INHERIT
NOINHERIT
LOGIN
NOLOGIN
REPLICATION
NOREPLICATION
BYPASSRLS
NOBYPASSRLS
CONNECTION LIMIT *connlimit*
[ENCRYPTED] PASSWORD '*password*'
PASSWORD NULL
VALID UNTIL '*timestamp*'

这些子句用于更改最初由 CREATE ROLE 设置的属性。欲了解更多信息，请参见 CREATE ROLE 参考页。

new_name

角色的新名称。

database_name

应在其中设置该配置变量的数据库名称。

configuration_parameter
value

将该角色在指定配置参数上的会话默认值设为给定值。如果 *value* 是 DEFAULT，或者等效地使用了RESET，则会移除角色特定变量设置，因此该角色在新会话中将继承系统范围的默认设置。使用 RESET ALL 可清除所有角色特定设置。SET FROM CURRENT 会将该参数在当前会话中的值保存为角色特定值。如果指定了 IN DATABASE，则只会为给定角色和数据库设置或移除该配置参数。

角色特定的变量设置只在登录时生效；SET ROLE 和 SET SESSION AUTHORIZATION 不会处理角色特定的变量设置。

关于允许的参数名称和值详见 SET 和第 19 章。

注解

使用 CREATE ROLE 添加新角色，使用DROP ROLE 删除角色。

ALTER ROLE 无法更改角色的成员资格。请使用 GRANT 和 REVOKE 执行这类操作。

用这个命令指定未加密密码时必须加以注意。密码将以明文形式传输到服务器，并且也可能被记录在客户端命令历史或服务器日志中。psql 包含一个命令\password，可用于更改角色的密码而不暴露明文密码。

也可以把会话默认值绑定到特定数据库，而不是角色；详见 ALTER DATABASE。如果发生冲突，同时针对数据库和角色的设置会覆盖角色特定设置，而角色特定设置又会覆盖数据库特定设置。

示例

更改角色的密码：

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

移除角色的密码：

```
ALTER ROLE davide WITH PASSWORD NULL;
```

更改密码的失效日期，并指定该密码应在比 UTC 快 1 小时的时区中，于 2015 年 5 月 4 日中午失效：

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

使密码永久有效：

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

赋予角色管理其他角色和创建新数据库的能力：

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

为角色指定 maintenance_work_mem 参数的非默认设置：

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

为角色指定 client_min_messages 参数的数据库特定的非默认设置：

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

兼容性

`ALTER ROLE` 语句是 PostgreSQL 的扩展。

另见

CREATE ROLE, DROP ROLE, ALTER DATABASE, SET

ALTER ROUTINE

ALTER ROUTINE — 更改一个例程的定义

大纲

```
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name
```

其中 *action* 为以下之一：

```
    IMMUTABLE | STABLE | VOLATILE
    [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST execution_cost
    ROWS result_rows
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

描述

`ALTER ROUTINE` 更改一个例程的定义。术语“例程”包括聚合函数、普通函数和过程。参数说明、更多示例以及进一步细节，请分别参见 ALTER AGGREGATE、ALTER FUNCTION 和 ALTER PROCEDURE。

示例

将针对 `integer` 类型的例程 `foo` 重命名为 `foobar`：

```
ALTER ROUTINE foo(integer) RENAME TO foobar;
```

不管 `foo` 是聚合、函数还是过程，这条命令都可以工作。

兼容性

该语句与 SQL 标准中的 `ALTER ROUTINE` 语句部分兼容。更多细节请参见 ALTER FUNCTION 和 ALTER PROCEDURE。允许例程名引用聚合函数是 PostgreSQL 的扩展。

另见

ALTER AGGREGATE, ALTER FUNCTION, ALTER PROCEDURE, DROP ROUTINE

注意，没有`CREATE ROUTINE`命令。

ALTER RULE

ALTER RULE — 修改一条重写规则的定义

大纲

```
ALTER RULE name ON table_name RENAME TO new_name
```

描述

`ALTER RULE` 更改一条现有规则的属性。目前唯一可用的操作是更改规则的名称。

要使用 `ALTER RULE`，你必须是该规则适用的表或视图的拥有者。

参数

name

要修改的现有规则的名称。

table_name

该规则适用的表或视图名称（可以是模式限定的）。

new_name

该规则的新名称。

示例

要重命名一条现有规则：

```
ALTER RULE notify_all ON emp RENAME TO notify_me;
```

兼容性

`ALTER RULE` 是 PostgreSQL 的一种语言扩展，整个查询重写系统也是如此。

另见

CREATE RULE, DROP RULE

ALTER SCHEMA

ALTER SCHEMA — 更改一个模式的定义

大纲

```
ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

描述

`ALTER SCHEMA` 更改一个模式的定义。

要使用 `ALTER SCHEMA`，必须拥有该模式。要重命名模式，还必须在数据库上拥有`CREATE` 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且必须在数据库上拥有`CREATE` 权限。（请注意，超级用户会自动拥有所有这些权限。）

参数

name

现有模式的名称。

new_name

该模式的新名称。新名称不能以 `pg_`开头，因为这类名称是为系统模式保留的。

new_owner

该模式的新拥有者。

兼容性

SQL 标准中没有 `ALTER SCHEMA` 语句。

另见

CREATE SCHEMA, DROP SCHEMA

ALTER SEQUENCE

ALTER SEQUENCE — 更改序列发生器的定义

大纲

```
ALTER SEQUENCE [ IF EXISTS ] name
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

描述

ALTER SEQUENCE 更改现有序列发生器的参数。在 ALTER SEQUENCE 命令中未明确设置的参数，会保留其原有设置。

要使用 ALTER SEQUENCE，必须拥有该序列。要更改序列的模式，还必须在新模式上拥有 CREATE 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在序列所在模式上拥有CREATE 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建序列完成的操作。不过，超级用户无论如何都可以更改任何序列的所有权。）

参数

name

要更改的序列名称（可以是模式限定的）。

IF EXISTS

如果该序列不存在，则不要抛出错误。这种情况下会发出一条提示。

data_type

可选子句 AS *data_type* 更改序列的数据类型。有效类型是 smallint、integer 和 bigint。

当且仅当此前的最小值和最大值就是旧数据类型的最小值或最大值时，更改数据类型才会自动更改序列的最小值和最大值（换句话说，如果该序列在创建时隐式或显式使用了 NO MINVALUE 或 NO MAXVALUE）。否则，最小值和最大值会被保留，除非在同一条命令中给出了新值。如果最小值和最大值无法适配新数据类型，则会产生错误。

increment

子句 INCREMENT BY *increment* 是可选的。正值使序列递增，负值使序列递减。如果未指定，则保留原有的增量值。

minvalue
NO MINVALUE

可选子句 MINVALUE *minvalue* 确定序列可生成的最小值。如果指定了 NO MINVALUE，则升序序列和降序序列将分别使用默认值 1 和该数据类型的最小值。如果两种选项都未指定，则保留当前最小值。

maxvalue
NO MAXVALUE

可选子句 MAXVALUE *maxvalue* 确定序列的最大值。如果指定了 NO MAXVALUE，则升序序列和降序序列将分别使用该数据类型的最大值和默认值 -1。如果两种选项都未指定，则保留当前最大值。

start

可选子句 START WITH *start* 更改序列记录的起始值。这不会影响当前序列值；它只是设置未来 ALTER SEQUENCE RESTART 命令将使用的值。

restart

可选子句 RESTART [WITH *restart*] 更改序列的当前值。这类似于以 is_called = false 调用 setval 函数：指定的值会由对 nextval 的下一次调用返回。写成不带 *restart* 值的 RESTART，等价于提供由CREATE SEQUENCE 记录、或最近由ALTER SEQUENCE START WITH 设置的起始值。

与 setval 调用相比，对序列执行RESTART 操作具有事务性，并且会阻止并发事务从同一序列中获取数字。如果这不是所期望的操作模式，则应使用setval。

cache

CACHE *cache* 子句启用预分配序列号并将其存储在内存中，以便更快访问。最小值为 1（一次只能生成一个值，即不缓存）。如果未指定，则保留旧的缓存值。

CYCLE

可选的 CYCLE 关键字可用于启用序列回绕：当递增或递减序列分别达到 *maxvalue* 或 *minvalue* 时回绕。如果达到限制，下一个生成的数将分别为*minvalue* 或 *maxvalue*。

NO CYCLE

如果指定了可选关键字 NO CYCLE，则当序列达到其最大值后，任何对 nextval 的调用都会返回错误。如果既未指定 CYCLE 也未指定 NO CYCLE，则保留原有的循环行为。

OWNED BY *table_name.column_name*
OWNED BY NONE

OWNED BY 选项使序列与特定表列关联；如果该列（或其整个表）被删除，序列也会自动删除。如果指定此选项，该关联会替换序列此前指定的任何关联。指定的表必须与序列拥有相同的所有者，并且位于同一模式中。指定OWNED BY NONE 会移除现有的关联，使序列成为 “独立的” 对象。

new_owner

序列的新所有者的用户名。

new_name

该序列的新拥有者的用户名。

new_schema

该序列的新名称。

注解

ALTER SEQUENCE 不会立即影响除当前后端之外、已经预分配（缓存）了序列值的其他后端中的 nextval 结果。它们会先用完所有缓存值，然后才会注意到序列生成参数已经改变。当前后端则会立即受到影响。

ALTER SEQUENCE 不会影响该序列的 currval 状态。（在 PostgreSQL 8.3 之前，它有时会这样做。）

ALTER SEQUENCE 会阻塞并发的 nextval、currval、lastval 和 setval 调用。

由于历史原因，序列也可以使用ALTER TABLE；但允许用于序列的 ALTER TABLE 变体仅限于与上述形式等价的那些。

示例

将一个名为 serial 的序列重启到 105：

```
ALTER SEQUENCE serial RESTART WITH 105;
```

兼容性

ALTER SEQUENCE 符合 SQL 标准，但 AS、START WITH、OWNED BY、OWNER TO、RENAME TO 和 SET SCHEMA 子句是 PostgreSQL 扩展。

另见

CREATE SEQUENCE, DROP SEQUENCE

ALTER SERVER

ALTER SERVER — 更改外部服务器的定义

大纲

```
ALTER SERVER name [ VERSION 'new_version' ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]
ALTER SERVER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SERVER name RENAME TO new_name
```

描述

`ALTER SERVER` 更改外部服务器的定义。第一种形式更改服务器的版本字符串或其通用选项（至少需要一个子句）。第二种形式更改服务器的拥有者。

要更改服务器，必须是服务器的所有者。此外，要更改所有者，还必须拥有该服务器，是新所有者角色的直接或间接成员，并且必须在服务器的外部数据包装器上拥有`USAGE` 权限。（请注意，超级用户会自动满足所有这些条件。）

参数

name

一个现有服务器的名称。

new_version

新服务器版本。

`OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )`

更改服务器的选项。`ADD`、`SET` 和 `DROP` 指定要执行的动作。如果未显式指定操作，则假定为`ADD`。选项名称必须唯一；名称和值也会使用该服务器的外部数据包装器库进行验证。

new_owner

该外部服务器的新拥有者的用户名。

new_name

该外部服务器的新名称。

示例

更改服务器 `foo`，添加连接选项：

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

更改服务器 `foo`，更改版本并修改 `host` 选项：

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

兼容性

`ALTER SERVER` 符合 ISO/IEC 9075-9 (SQL/MED)。`OWNER TO` 和 `RENAME` 形式是 PostgreSQL 扩展。

另见

CREATE SERVER, DROP SERVER

ALTER STATISTICS

ALTER STATISTICS — 更改扩展统计对象的定义

大纲

```
ALTER STATISTICS name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER STATISTICS name RENAME TO new_name
ALTER STATISTICS name SET SCHEMA new_schema
```

描述

`ALTER STATISTICS` 修改现有扩展统计对象的参数。凡未在 `ALTER STATISTICS` 命令中明确设置的参数，都会保留其原有设置。

要使用 `ALTER STATISTICS`，必须拥有该统计对象。要更改统计对象的模式，还必须在新模式上拥有`CREATE` 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在统计对象所在模式上拥有`CREATE` 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建统计对象完成的操作。不过，超级用户无论如何都可以更改任何统计对象的所有权。）

参数

name

要修改的统计对象的名称（可以用模式限定）。

new_owner

统计对象的新所有者的用户名。

new_name

统计对象的新名称。

new_schema

统计对象的新模式。

兼容性

SQL 标准中没有 `ALTER STATISTICS` 命令。

另见

CREATE STATISTICS, DROP STATISTICS

ALTER SUBSCRIPTION

ALTER SUBSCRIPTION — 修改订阅的定义

大纲

```
ALTER SUBSCRIPTION name CONNECTION 'conninfo'
ALTER SUBSCRIPTION name SET PUBLICATION publication_name [, ...] [ WITH ( set_publication_
option [= value] [, ... ] ) ]
ALTER SUBSCRIPTION name REFRESH PUBLICATION [ WITH ( refresh_option [= value] [, ... ] ) ]
ALTER SUBSCRIPTION name ENABLE
ALTER SUBSCRIPTION name DISABLE
ALTER SUBSCRIPTION name SET ( subscription_parameter [= value] [, ... ] )
ALTER SUBSCRIPTION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SUBSCRIPTION name RENAME TO new_name
```

描述

ALTER SUBSCRIPTION 可以更改大多数可在 CREATE SUBSCRIPTION 中指定的订阅属性。

要使用 ALTER SUBSCRIPTION，必须拥有该订阅。要更改所有者，还必须是新所有者角色的直接或间接成员。新所有者必须是超级用户。（目前，所有订阅所有者都必须是超级用户，因此实际上会绕过所有者检查。但将来可能会改变。）

参数

name

要更改其属性的订阅名称。

CONNECTION '*conninfo*'

该子句更改最初由 CREATE SUBSCRIPTION 设置的连接属性。详情请参见该命令。

SET PUBLICATION *publication_name*

更改已订阅发布的列表。更多信息请参见 CREATE SUBSCRIPTION。默认情况下，此命令还会执行类似REFRESH PUBLICATION 的操作。

set_publication_option 指定此操作的其他选项。支持的选项包括：

refresh (boolean)

为 false 时，命令不会尝试刷新表信息。此时应单独执行REFRESH PUBLICATION。默认值为 true。

此外，还可以指定REFRESH PUBLICATION 中所述的刷新选项。

REFRESH PUBLICATION

从发布者获取缺失的表信息。这将开始复制自上次调用 REFRESH PUBLICATION 或自 CREATE SUBSCRIPTION 以来添加到已订阅发布中的表。

refresh_option 指定刷新操作的其他选项。支持的选项包括：

copy_data (boolean)

指定在复制开始时，是否复制所订阅发布中预先存在的数据。默认值为true。（以前已订阅的表不会再次复制。）

ENABLE

启用先前禁用的订阅，并在事务结束时启动逻辑复制工作进程。

DISABLE

禁用正在运行的订阅，并在事务结束时停止逻辑复制工作进程。

SET (*subscription_parameter* [= *value*] [, ...])

此子句更改最初由 CREATE SUBSCRIPTION 设置的参数。更多信息请参见该命令。允许的选项包括 slot_name 和 synchronous_commit

new_owner

订阅的新所有者的用户名。

new_name

订阅的新名称。

示例

将订阅所订阅的发布更改为 insert_only：

```
ALTER SUBSCRIPTION mysub SET PUBLICATION insert_only;
```

禁用（停止）该订阅：

```
ALTER SUBSCRIPTION mysub DISABLE;
```

兼容性

ALTER SUBSCRIPTION 是 PostgreSQL 的一个扩展。

另见

CREATE SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

ALTER SYSTEM

ALTER SYSTEM — 更改服务器配置参数

大纲

```
ALTER SYSTEM SET configuration_parameter { TO | = } { value [, ...] | DEFAULT }

ALTER SYSTEM RESET configuration_parameter
ALTER SYSTEM RESET ALL
```

描述

`ALTER SYSTEM` 用于更改整个数据库集簇范围内的服务器配置参数。与传统的手工编辑 `postgresql.conf` 文件相比，它可能更方便。`ALTER SYSTEM` 会把给定的参数设置写入 `postgresql.auto.conf` 文件；除读取 `postgresql.conf` 之外，系统还会读取该文件。把参数设置为 `DEFAULT`，或者使用`RESET` 变体，会从`postgresql.auto.conf` 文件中移除相应的配置项。使用 `RESET ALL` 可以移除所有这类配置项。

用 `ALTER SYSTEM` 设置的值，会在下一次重新加载服务器配置后生效；对于那些只能在服务器启动时更改的参数，则会在下一次服务器重启后生效。重新加载服务器配置可以通过调用 SQL 函数 `pg_reload_conf()`、运行 `pg_ctl reload`，或者向主服务器进程发送 SIGHUP 信号来触发。

只有超级用户可以使用 `ALTER SYSTEM`。此外，由于此命令直接作用于文件系统且无法回滚，因此不允许在事务块或函数中使用。

参数

configuration_parameter

一个可设置的配置参数名称。可用参数见第 19 章。

value

参数的新值。根据具体参数的要求，值可以指定为字符串常量、标识符、数字，或这些内容构成的逗号分隔列表。既不是数字也不是有效标识符的值必须加引号。可以写入`DEFAULT`，以指定从 `postgresql.auto.conf` 中删除该参数及其值。

对于某些接受列表的参数，带引号的值会生成带双引号的输出，以保留空白和逗号；对于另一些参数，则必须在单引号字符串内部使用双引号才能达到这种效果。

注解

此命令不能用于设置 data_directory，也不能用于设置`postgresql.conf` 中不允许的参数（例如预设选项）。

关于设置这些参数的其他方法，见第 19.1 节。

示例

设置 wal_level：

```
ALTER SYSTEM SET wal_level = replica;
```

撤销该设置，恢复为postgresql.conf 中生效的设置：

```
ALTER SYSTEM RESET wal_level;
```

兼容性

ALTER SYSTEM 语句是一种 PostgreSQL 扩展。

另见

SET, SHOW

ALTER TABLE

ALTER TABLE — 更改一个表的定义

大纲

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
ALTER TABLE [ IF EXISTS ] name
    ATTACH PARTITION partition_name { FOR VALUES partition_bound_spec | DEFAULT }
ALTER TABLE [ IF EXISTS ] name
    DETACH PARTITION partition_name
```

其中 *action* **为以下之一:**

```
    ADD [ COLUMN ] [ IF NOT EXISTS ] column_name data_type [ COLLATE collation ] [ column_
constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [
 USING expression ]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ (
 sequence_options ) ]
    ALTER [ COLUMN ] column_name { SET GENERATED { ALWAYS | BY DEFAULT } | SET sequence_option
 | RESTART [ [ WITH ] restart ] } [...]
    ALTER [ COLUMN ] column_name DROP IDENTITY [ IF EXISTS ]
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ADD table_constraint [ NOT VALID ]
    ADD table_constraint_using_index
    ALTER CONSTRAINT constraint_name [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED |
 INITIALLY IMMEDIATE ]
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    DISABLE RULE rewrite_rule_name
    ENABLE RULE rewrite_rule_name
    ENABLE REPLICA RULE rewrite_rule_name
    ENABLE ALWAYS RULE rewrite_rule_name
    DISABLE ROW LEVEL SECURITY
    ENABLE ROW LEVEL SECURITY
    FORCE ROW LEVEL SECURITY
    NO FORCE ROW LEVEL SECURITY
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET WITH OIDS
    SET WITHOUT OIDS
    SET TABLESPACE new_tablespace
    SET { LOGGED | UNLOGGED }
    SET ( storage_parameter [= value] [, ... ] )
```

```
    RESET ( storage_parameter [, ... ] )
    INHERIT parent_table
    NO INHERIT parent_table
    OF type_name
    NOT OF
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }
```

而 *partition_bound_spec* 为:

```
IN ( { numeric_literal | string_literal | TRUE | FALSE | NULL } [, ...] ) |
FROM ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE } [, ...] )
  TO ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )
```

而 *column_constraint* 为:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

而 *table_constraint* 为:

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] ) index_parameters [
 WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

而 *table_constraint_using_index* 为:

```
    [ CONSTRAINT constraint_name ]
    { UNIQUE | PRIMARY KEY } USING INDEX index_name
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

UNIQUE、PRIMARY KEY 和 EXCLUDE 约束中的 *index_parameters* 为:

```
[ INCLUDE ( column_name [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

EXCLUDE 约束中的 *exclude_element* 为:

```
{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

描述

ALTER TABLE 更改现有表的定义。下面描述了若干子形式。请注意，各子形式所需的锁级别可能不同。除非另有明确说明，否则将获取ACCESS EXCLUSIVE 锁。给出多个子命令时，获取的锁将是其中任一子命令所需的最严格级别。

ADD COLUMN [IF NOT EXISTS]

该形式使用与 CREATE TABLE 相同的语法向表中添加新列。如果指定了 IF NOT EXISTS，且同名列已经存在，则不会报错。

DROP COLUMN [IF EXISTS]

该形式从表中删除一列。涉及该列的索引和表约束也会自动删除。如果删除该列会使某个引用它的多元统计信息只剩下一列数据，那么该统计信息也会被移除。如果表外有任何对象依赖于该列，例如外键引用或视图，你就需要指定CASCADE。如果指定了 IF EXISTS 而该列不存在，则不会报错；此时会发出一条提示。

SET DATA TYPE

该形式更改表中某一列的类型。涉及该列的索引和简单表约束会通过重新解析最初提供的表达式，自动转换为使用新的列类型。可选的COLLATE 子句为新列指定排序规则；如果省略，则使用新列类型的默认排序规则。可选的USING 子句指定如何根据旧值计算新列的值；如果省略，则默认转换与从旧数据类型到新数据类型的赋值类型转换相同。如果从旧类型到新类型不存在隐式或赋值类型转换，则必须提供USING 子句。

SET/DROP DEFAULT

这些形式为列设置或移除默认值。默认值只会应用于后续的 INSERT 或 UPDATE 命令；它们不会导致表中已有的行发生变化。

SET/DROP NOT NULL

这些形式更改列是否被标记为允许空值，或拒绝空值。只有当列不包含空值时，才能使用 SET NOT NULL。

如果该表是分区，并且某列在父表中标记为NOT NULL，则不能对该列执行 DROP NOT NULL。要从所有分区删除 NOT NULL 约束，请在父表上执行 DROP NOT NULL。即使父表没有 NOT NULL 约束，仍然可以按需将此类约束添加到单个分区；也就是说，即使父表允许空值，子表也可以禁止空值，但反之则不行。

ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
SET GENERATED { ALWAYS | BY DEFAULT }
DROP IDENTITY [IF EXISTS]

这些形式更改列是否为标识列，或更改现有标识列的生成属性。详情请参见 CREATE TABLE。

如果指定了 DROP IDENTITY IF EXISTS，而该列不是标识列，则不会报错；此时会发出一条提示。

SET *sequence_option*
RESTART

这些形式更改支撑现有标识列的序列。*sequence_option* 是 ALTER SEQUENCE 支持的选项，例如INCREMENT BY。

SET STATISTICS

该形式为后续 ANALYZE 操作设置每列的统计信息收集目标。目标可以设置在 0 到 10000 范围内；也可以将其设置为 -1，以恢复使用系统默认的统计目标（default_statistics_target）。有关 PostgreSQL 查询规划器使用统计信息的更多信息，请参见第 14.2 节。

SET STATISTICS 会获取一个 SHARE UPDATE EXCLUSIVE 锁。

```
SET ( attribute_option = value [, ... ] )
RESET ( attribute_option [, ... ] )
```

该形式设置或重置每属性选项。目前定义的每属性选项只有 n_distinct 和 n_distinct_inherited，它们会覆盖后续ANALYZE 操作所作的非重复值数量估计。n_distinct 影响表本身的统计信息，而n_distinct_inherited 影响为表及其继承子表收集的统计信息。当设置为正值时，ANALYZE 会假定该列恰好包含指定数量的非空非重复值。当设置为负值时（该值必须大于或等于 -1），ANALYZE 会假定列中非空非重复值的数量与表大小成线性关系；具体数量通过将估计的表大小乘以给定数值的绝对值计算。例如，-1 表示列中的所有值都不同，而 -0.5 表示平均每个值出现两次。当表大小随时间变化时，这可能很有用，因为只有在查询规划时才会执行与表中行数相乘的操作。指定 0 可恢复正常估计非重复值数量。有关PostgreSQL 查询规划器使用统计信息的更多信息，请参见第 14.2 节。

更改每个属性的选项会获取一个 SHARE UPDATE EXCLUSIVE 锁。

```
SET STORAGE
```

该形式为列设置存储模式。这控制该列是行内保存还是保存在辅助 TOAST 表中，以及是否压缩数据。对于integer 等定长值，必须使用PLAIN，数据以行内且未压缩的形式保存。MAIN 用于行内的可压缩数据。EXTERNAL 用于行外保存的未压缩数据，EXTENDED 用于行外保存的压缩数据。对于支持非 PLAIN 存储的大多数数据类型，EXTENDED 是默认值。使用 EXTERNAL 会使非常大的 text 和 bytea 值上的子字符串操作运行得更快，但会增加存储空间。请注意，SET STORAGE 本身不会更改表中的任何内容，它只会设置今后更新表时采用的策略。更多信息请参见第 69.2 节。

```
ADD table_constraint [ NOT VALID ]
```

该形式使用与 CREATE TABLE 相同的约束语法向表添加新约束，另外还提供NOT VALID 选项；目前该选项只允许用于外键和 CHECK 约束。

通常，该形式会扫描表，以验证表中所有现有行都满足新约束。但如果使用NOT VALID 选项，则会跳过这一可能耗时很长的扫描。后续插入或更新仍会强制执行该约束（也就是说，对于外键，除非被引用表中存在匹配行，否则操作会失败；对于 CHECK 约束，除非新行符合指定检查条件，否则操作会失败）。但是，在使用VALIDATE CONSTRAINT 选项验证约束之前，数据库不会假定该约束对表中的所有行都成立。有关使用NOT VALID 选项的更多信息，请参见下文注解。

尽管大多数形式的 ADD *table_constraint* 需要 ACCESS EXCLUSIVE 锁，ADD FOREIGN KEY 只需要 SHARE ROW EXCLUSIVE 锁。请注意，ADD FOREIGN KEY 除了在声明约束的表上获取锁之外，还会在被引用的表上获取SHARE ROW EXCLUSIVE 锁。

向分区表添加唯一约束或主键约束时还会有其他限制；请参见 CREATE TABLE。此外，目前不能将分区表上的外键约束声明为NOT VALID。

```
ADD table_constraint_using_index
```

该形式基于一个现有唯一索引向表中添加新的 PRIMARY KEY 或 UNIQUE 约束。该约束将包含该索引的全部列。

该索引不能包含表达式列，也不能是部分索引。此外，它必须是使用默认排序顺序的 B-树索引。这些限制保证该索引等价于通过常规ADD PRIMARY KEY 或 ADD UNIQUE 命令构建出来的索引。

如果指定了 PRIMARY KEY，而该索引的列尚未被标记为NOT NULL，则此命令会尝试对每个这样的列执行ALTER COLUMN SET NOT NULL。这需要进行一次全表扫描，以验证这些列不包含空值。在其他情况下，这都是一个快速操作。

如果提供了约束名，则索引会被重命名以匹配该约束名；否则，约束将使用与索引相同的名称。

执行此命令后，该索引会被该约束“拥有”，就像它是通过常规ADD PRIMARY KEY 或 ADD UNIQUE 命令构建出来的一样。特别是，删除该约束也会使该索引一并消失。

该形式目前不支持分区表。

> **注意**
>
> 在需要添加新约束且不希望长时间阻塞表更新的情况下，使用现有索引添加约束可能很有用。为此，请使用CREATE INDEX CONCURRENTLY 创建索引，然后使用此语法将其安装为正式约束。请参见下面的示例。

ALTER CONSTRAINT

该形式更改先前创建的约束的属性。目前只能更改外键约束。

VALIDATE CONSTRAINT

该形式通过扫描表来验证先前以 NOT VALID 创建的外键或 CHECK 约束，确保不存在不满足约束的行。如果约束已经标记为有效，则不执行任何操作。（有关此命令用途的说明，请参见下文注解。）

此命令会获取一个 SHARE UPDATE EXCLUSIVE 锁。

DROP CONSTRAINT [IF EXISTS]

该形式删除表上指定的约束，以及支撑该约束的任何索引。如果指定了IF EXISTS 而该约束不存在，则不会报错；此时会发出一条提示。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

这些形式配置属于表的触发器的触发。已禁用的触发器仍为系统所知，但在触发事件发生时不会执行。对于延迟触发器，会在事件发生时检查启用状态，而不是在实际执行触发器函数时检查。可以按名称指定单个触发器，或指定表上的所有触发器，或仅指定用户触发器，从而禁用或启用触发器（最后一种选项不包括内部生成的约束触发器，例如用于实现外键约束或可延迟唯一性约束和排他约束的触发器）。禁用或启用内部生成的约束触发器需要超级用户权限；应谨慎操作，因为如果不执行触发器，当然无法保证约束的完整性。

触发器的触发机制也受配置变量 session_replication_role 的影响。普通启用的触发器（默认）会在复制角色为 “origin”（默认）或 “local” 时触发。配置为 ENABLE REPLICA 的触发器仅会在会话处于 “replica” 模式时触发，而配置为ENABLE ALWAYS 的触发器无论当前复制角色为何都会触发。

在默认配置下，这种机制的效果是触发器不会在副本上触发。这很有用，因为如果在源端使用触发器在表之间传播数据，那么复制系统也会复制传播的数据；触发器不应在副本上再次触发，否则会导致重复。但是，如果触发器用于其他目的（例如创建外部告警），则可以将其设置为ENABLE ALWAYS，使其也在副本上触发。

此命令会获取一个 SHARE ROW EXCLUSIVE 锁。

DISABLE/ENABLE [REPLICA | ALWAYS] RULE

这些形式配置属于表的重写规则是否应用。禁用的规则仍然为系统所知，但在查询重写期间不会应用。语义与禁用/启用触发器相同。对于ON SELECT 规则，此配置将被忽略，这些规则始终会被应用，以确保视图在当前会话处于非默认复制角色时仍然正常工作。

规则触发机制也受配置变量 session_replication_role 的影响，类似于上面描述的触发器。

DISABLE/ENABLE ROW LEVEL SECURITY

这些形式控制属于表的行安全性策略的应用。如果启用且表不存在任何策略，则应用默认拒绝策略。请注意，即使禁用了表的行级安全，表仍然可以存在策略；在这种情况下，不会应用这些策略，并且会忽略它们。另请参见 CREATE POLICY。

NO FORCE/FORCE ROW LEVEL SECURITY

这些形式控制当用户是表所有者时属于表的行安全性策略的应用。如果启用，当用户是表所有者时会应用行级安全策略。如果禁用（默认设置），当用户是表所有者时不会应用行级安全。另请参见 CREATE POLICY。

CLUSTER ON

该形式为将来的 CLUSTER 操作选择默认索引。它实际上不会对表重新聚簇。

更改聚簇选项会获取一个 SHARE UPDATE EXCLUSIVE 锁。

SET WITHOUT CLUSTER

该形式从表中移除最近使用的 CLUSTER 索引规范。这会影响将来的聚簇操作（这些操作未指定索引）。

更改聚簇选项会获取一个 SHARE UPDATE EXCLUSIVE 锁。

SET WITH OIDS

该形式向表添加一个 oid 系统列（参见第 5.4 节）。如果表已有 OID，则不执行任何操作。

请注意，这并不等同于ADD COLUMN oid oid；后者会添加一个碰巧命名为 oid 的普通列，而不是系统列。

SET WITHOUT OIDS

该形式从表中移除 oid 系统列。这完全等同于 DROP COLUMN oid RESTRICT，但如果表中已经没有oid 列，则不会报错。

SET TABLESPACE

该形式将表的表空间更改为指定的表空间，并将与表关联的数据文件移动到新表空间。表上的索引（如果有）不会移动，但可以通过额外的SET TABLESPACE 命令单独移动。可以使用 ALL IN TABLESPACE 形式移动当前数据库中某个表空间内的所有表；该形式会先锁定所有要移动的表，然后逐个移动。该形式还支持OWNED BY，只移动指定角色所拥有的表。如果指定NOWAIT 选项，而命令无法立即获取所需的全部锁，则会失败。请注意，该命令不会移动系统目录；如有需要，请改用ALTER DATABASE 或显式调用 ALTER TABLE。

information_schema 关系不被视为系统目录的一部分，因此会被移动。另请参见 CREATE TABLESPACE。

SET { LOGGED | UNLOGGED }

该形式把表从不记录 WAL 改为记录 WAL，或反过来（参见 UNLOGGED）。它不能应用于临时表。

SET (*storage_parameter* [= *value*] [, ...])

该形式更改表的一个或多个存储参数。有关可用参数的详情，请参见存储参数。请注意，该命令不会立即修改表内容；根据参数的不同，可能需要重写表才能达到预期效果。可以使用 VACUUM FULL、CLUSTER，或ALTER TABLE 中会强制重写表的某种形式来完成重写。对于与规划器相关的参数，更改会从下次锁定表时生效，因此不会影响当前正在执行的查询。

对于 fillfactor、toast 和 autovacuum 存储参数，以及规划器参数parallel_workers，会获取SHARE UPDATE EXCLUSIVE 锁。

注意

虽然 CREATE TABLE 允许在 WITH (*storage_parameter*) 语法中指定 OIDS，但ALTER TABLE 不会将 OIDS 视为存储参数。应改用 SET WITH OIDS 和 SET WITHOUT OIDS 形式来更改 OID 状态。

RESET (*storage_parameter* [, ...])

该形式把一个或多个存储参数重置为默认值。与 SET 一样，可能仍需要进行表重写才能让整张表完全更新。

INHERIT *parent_table*

该形式将目标表添加为指定父表的新子表。之后，对父表执行的查询将包含目标表的记录。要作为子表添加，目标表必须已经包含与父表相同的所有列（也可以有额外的列）。列必须具有匹配的数据类型；如果父表中的列具有NOT NULL 约束，则子表中的相应列也必须具有 NOT NULL 约束。

父表的所有 CHECK 约束还必须在子表中有匹配的约束，但标记为不可继承的约束除外（即在父表中使用ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT 创建的约束）；这类约束会被忽略。所有匹配的子表约束都不能标记为不可继承。目前不考虑 UNIQUE、PRIMARY KEY 和 FOREIGN KEY 约束，但将来可能会改变。

NO INHERIT *parent_table*

该形式把目标表从指定父表的子表列表中移除。对父表的查询将不再包含来自目标表的记录。

OF *type_name*

该形式将表关联到一个复合类型，就像由CREATE TABLE OF 创建该表一样。表的列名和类型列表必须与复合类型的完全匹配；是否存在 oid 系统列可以不同。该表不能继承任何其他表。这些限制确保 CREATE TABLE OF 允许等效的表定义。

NOT OF

该形式会解除类型化表与其类型之间的关联。

OWNER TO

该形式把表、序列、视图、物化视图或外部表的所有者更改为指定用户。

REPLICA IDENTITY

该形式修改写入预写式日志的信息，以标识被更新或删除的行。在大多数情况下，只有当某列的旧值与新值不同，才记录该列的旧值；不过，如果旧值是外部存储的，则无论它是否改变，都会被记录。除非正在使用逻辑复制，否则该选项没有效果。

DEFAULT

记录主键列（如果有）的旧值。这是非系统表的默认值。

USING INDEX *index_name*

记录由指定索引覆盖的列的旧值。该索引必须是唯一的、非部分的、不可延迟的，并且只包含被标记为NOT NULL 的列。如果该索引被删除，其行为与NOTHING 相同。

FULL

记录该行中所有列的旧值。

NOTHING

不记录关于旧行的任何信息。这是系统表的默认值。

RENAME

RENAME 形式用于更改表（或索引、序列、视图、物化视图或外部表）的名称、表中某个列的名称，或者表上某个约束的名称。重命名一个带有底层索引的约束时，该索引也会一并重命名。已存储的数据不会受到影响。

SET SCHEMA

该形式把表移动到另一个模式。相关索引、约束以及由表列拥有的序列也会一并移动。

ATTACH PARTITION *partition_name* { FOR VALUES *partition_bound_spec* | DEFAULT }

该形式将现有表（该表本身也可以分区）附加为目标表的分区。可以使用 FOR VALUES 将表作为特定值的分区附加，也可以使用DEFAULT 将其作为默认分区附加。对于目标表中的每个索引，附加表都会创建对应的索引；或者，如果已经存在等效索引，则会将其附加到目标表的索引上，就像执行了ALTER INDEX ATTACH PARTITION 一样。请注意，如果现有表是外部表，而目标表上有UNIQUE 索引，目前不允许将该表作为目标表的分区附加。（另请参见 CREATE FOREIGN TABLE。）对于目标表中存在的每个用户定义行级触发器，都会在附加表中创建相应的触发器。

使用 FOR VALUES 的分区使用与 CREATE TABLE 中 *partition_bound_spec* 相同的语法。分区边界规范必须符合目标表的分区策略和分区键。要附加的表必须拥有与目标表完全相同的列，不能多也不能少；此外，列类型也必须匹配。它还必须拥有目标表的所有NOT NULL 和 CHECK 约束。目前不考虑 FOREIGN KEY 约束。父表中的 UNIQUE 和

PRIMARY KEY 约束会在分区中创建（如果尚不存在）。如果要附加的表中的任何 CHECK 约束标记为 NO INHERIT，命令将失败；必须在不带NO INHERIT 子句的情况下重新创建此类约束。

如果新分区是普通表，则会执行一次全表扫描，以检查表中现有行不会违反分区约束。可以在执行此命令之前，先向该表添加一个有效的CHECK 约束，使其只允许满足所需分区约束的行，从而避免这次扫描。系统会利用该CHECK 约束来判断无须扫描该表以验证分区约束。不过，如果分区键中有任何一个是表达式，而该分区又不接受NULL 值，则这一方法无效。如果附加的是一个不接受NULL 值的列表分区，还应向分区键列添加NOT NULL 约束，除非该分区键是表达式。

如果新分区是外部表，则不会执行任何检查来验证该外部表中的所有行都满足分区约束。（关于外部表上的约束，请参见CREATE FOREIGN TABLE 中的讨论。）

当某个表具有默认分区时，定义一个新分区会改变默认分区的分区约束。默认分区不能包含任何本应移动到新分区中的行，因此系统会扫描默认分区以确认不存在此类行。和扫描新分区一样，如果存在适当的CHECK 约束，也可以避免这次扫描；同样地，当默认分区是外部表时，这次扫描总会被跳过。

DETACH PARTITION *partition_name*

该形式从目标表分离指定的分区。分离后的分区继续作为独立表存在，但不再与分离它的表有任何关联。附加到目标表索引的任何索引也会被分离。作为目标表触发器克隆创建的任何触发器都会被移除。

所有作用于单个表的 ALTER TABLE 形式，除RENAME、SET SCHEMA、ATTACH PARTITION 和 DETACH PARTITION 外，都可以组合成一个多项修改列表，一并执行。例如，可以在一条命令中添加多个列并且/或者修改多个列的类型。对于大型表，这一点尤其有用，因为这样只需对表执行一次遍历。

要使用 ALTER TABLE，必须拥有该表。要更改表的模式或表空间，还必须在新模式或表空间上拥有CREATE 权限。要将表作为父表的新子表添加，还必须拥有父表。要将一个表作为另一个表的新分区附加，还必须拥有要附加的表。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在表所在模式上拥有CREATE 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建表完成的操作。不过，超级用户无论如何都可以更改任何表的所有权。）要添加列、更改列类型或使用OF 子句，还必须在数据类型上拥有USAGE 权限。

参数

IF EXISTS

如果表不存在，则不要抛出错误。这种情况下会发出一条提示。

name

要修改的一个现有表的名称（可以是模式限定的）。如果在表名前指定了 ONLY，则只会修改该表。如果没有指定ONLY，该表及其所有后代表（如果有）都会被修改。可选地，在表名后面可以指定 * 用来显式地指示包括后代表。

column_name

一个新列或者现有列的名称。

new_column_name

一个现有列的新名称。

new_name

该表的新名称。

data_type

一个新列的数据类型或者一个现有列的新数据类型。

table_constraint

该表的新的表约束。

constraint_name

一个新约束或者现有约束的名称。

CASCADE

自动删除依赖于被删除列或约束的对象（例如引用该列的视图），并且接着删除依赖于那些对象的所有对象（见第 5.13 节）。

RESTRICT

如果有任何依赖对象时拒绝删除列或者约束。这是默认行为。

trigger_name

要禁用或启用的单个触发器的名称。

ALL

禁用或启用属于表的所有触发器。（如果触发器中有内部生成的约束触发器，例如用于实现外键约束或可延迟唯一性约束和排他约束的触发器，则需要超级用户权限。）

USER

禁用或启用属于表的所有触发器，但不包括内部生成的约束触发器（例如用于实现外键约束或可延迟的唯一约束和排他约束的触发器）。

index_name

一个现有索引的名称。

storage_parameter

一个表存储参数的名称。

value

一个表存储参数的新值。根据该参数，该值可能是一个数字或者一个词。

parent_table

要与这个表关联或者解除关联的父表。

new_owner

该表的新所有者的用户名。

new_tablespace

要把该表移入其中的表空间的名称。

new_schema

要把该表移入其中的模式的名称。

partition_name

要附加为该表的新分区或从该表分离出去的表名。

partition_bound_spec

新分区的分区边界说明。关于该语法的更多细节，请参见 CREATE TABLE。

注解

关键字 COLUMN 不影响语义，可以省略。

使用 ADD COLUMN 添加列且指定了非易失性的 DEFAULT 时，会在语句执行时计算默认值，并将结果存储在表的元数据中。对于所有现有行，该值都将用于该列。如果未指定DEFAULT，则使用NULL。在这两种情况下都不需要重写表。

使用易失性的 DEFAULT 添加列，或更改现有列的类型，都需要重写整个表及其索引。作为更改现有列类型时的例外，如果USING 子句不改变列内容，并且旧类型要么可以二进制强制转换为新类型，要么是新类型之上的无约束域，则不需要重写表；但受影响列上的任何索引仍必须重建。添加或移除系统oid 列也需要重写整个表。对于大型表，重建表和/或索引可能需要相当长的时间，并且会暂时需要最多两倍的磁盘空间。

添加 CHECK 或 NOT NULL 约束需要扫描表，以验证现有行满足约束，但不需要重写表。

类似地，在附加新分区时，也可能会扫描该分区以验证现有行满足分区约束。

在单个 ALTER TABLE 中允许指定多个更改的主要原因，是可以把多次表扫描或重写合并为对表的一次遍历。

扫描大表以验证新的外键或检查约束可能需要很长时间，并且在ALTER TABLE ADD CONSTRAINT 命令提交之前，会阻止对该表的其他更新。NOT VALID 约束选项的主要目的，是减小添加约束对并发更新的影响。使用NOT VALID 时，ADD CONSTRAINT 命令不会扫描表，因此可以立即提交。之后可以发出VALIDATE CONSTRAINT 命令，以验证现有行满足该约束。验证步骤不需要阻止并发更新，因为它知道其他事务会对它们插入或更新的行强制执行该约束；只需检查预先存在的行。因此，验证只会在被修改的表上获取SHARE UPDATE EXCLUSIVE 锁。（如果约束是外键，则被该约束引用的表上还需要ROW SHARE 锁。）除了改善并发性之外，在已知该表包含既有违规数据的情况下，NOT VALID 和 VALIDATE CONSTRAINT 也很有用。一旦约束已经建立，就不能再插入新的违规数据，而现有问题则可以从容修正，直到 VALIDATE CONSTRAINT 最终成功。

DROP COLUMN 形式不会从物理上删除列，而只是使其对 SQL 操作不可见。表后续的插入和更新操作会为该列存储空值。因此，删除列的速度很快，但不会立即减少表在磁盘上的大小，因为被

删除列占用的空间不会被回收。随着现有行被更新，空间会逐渐回收。（删除系统oid 列时不适用这些说明；删除该列会立即重写表。）

若要强制立即回收已删除列所占的空间，可以执行任何一种会导致整表重写的ALTER TABLE 形式。这样会重建每一行，并用空值替换被删除的列。

会重写表的 ALTER TABLE 形式对于 MVCC 来说并不安全。表重写完成后，如果并发事务使用的是在重写发生之前取得的快照，那么该表在这些并发事务看来会像一张空表。详见第 13.5 节。

SET DATA TYPE 的 USING 选项实际上可以指定任何涉及该行旧值的表达式；也就是说，它既可以引用正在转换的列，也可以引用其他列。这使得使用SET DATA TYPE 语法完成非常通用的转换成为可能。正因为这种灵活性，USING 表达式不会应用到列的默认值（如果有）上，因为其结果可能不是默认值所要求的常量表达式。这意味着，当从旧类型到新类型不存在隐式或赋值类型转换时，即便提供了USING 子句，SET DATA TYPE 也可能仍然无法转换默认值。在这种情况下，可以先用DROP DEFAULT 删除默认值，执行ALTER TYPE，然后再用SET DEFAULT 添加一个合适的新默认值。类似的考虑也适用于涉及该列的索引和约束。

如果某个表有任何后代表，那么在不对后代表执行相同操作的情况下，就不允许在父表中添加列、重命名列或更改列类型。这保证了后代表始终拥有与父表相匹配的列。类似地，如果不同时重命名所有后代上的CHECK 约束，就不能只在父表上重命名该检查约束，这样CHECK 约束才能在父表及其后代之间保持匹配。（不过，这一限制不适用于基于索引的约束。）此外，由于查询父表时也会同时查询其后代，父表上的约束除非在这些后代上也被标记为有效，否则就不能被标记为有效。在所有这些情况下，ALTER TABLE ONLY 都会被拒绝。

只有当某个后代表中的列既不是从其他父表继承而来，也从未有过该列的独立定义时，递归的 DROP COLUMN 操作才会移除该后代表中的此列。非递归的 DROP COLUMN（即 ALTER TABLE ONLY ... DROP COLUMN）永远不会移除任何后代列，而只会把它们标记为独立定义，而非继承得到。对于分区表，非递归的DROP COLUMN 命令会失败，因为一张表的所有分区都必须与分区根表拥有相同的列。

标识列的操作（ADD GENERATED、SET 等，以及DROP IDENTITY），以及TRIGGER、CLUSTER、OWNER 和 TABLESPACE 操作，都不会递归到后代表；也就是说，它们的行为始终如同指定了ONLY。添加约束时，只有未标记为NO INHERIT 的 CHECK 约束会递归。

不允许更改系统目录表的任何部分。

有关有效参数的进一步说明，请参见 CREATE TABLE。第 5 章中还有关于继承的更多信息。

示例

要添加类型为 varchar 的列到表中：

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

要从表中删除一列：

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

要在一个操作中更改两个现有列的类型：

```
ALTER TABLE distributors
    ALTER COLUMN address TYPE varchar(80),
    ALTER COLUMN name TYPE varchar(100);
```

要把一个包含 Unix 时间戳的整数列改为 `timestamp with time zone`，并通过`USING` 子句完成转换：

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

如果该列带有一个不能自动转换为新数据类型的默认值表达式，也是同样的做法：

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

要重命名一个现有列：

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

重命名一个现有的表：

```
ALTER TABLE distributors RENAME TO suppliers;
```

重命名一个现有的约束：

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

为一列增加一个非空约束：

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

从一列移除一个非空约束：

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

要向一个表及其所有后代添加一个检查约束：

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

要只向一个表本身添加检查约束，而不添加到其后代：

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5) NO INHERIT;
```

（该检查约束也不会被未来的后代表继承。）

要从一个表及其所有后代移除一个检查约束：

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

只从一个表移除一个检查约束：

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

（该检查约束在所有子表上仍然保留。）

为一个表增加一个外键约束：

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (
address);
```

为一个表增加一个外键约束，并且尽量不要影响其他工作：

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (
address) NOT VALID;
ALTER TABLE distributors VALIDATE CONSTRAINT distfk;
```

为一个表增加一个（多列）唯一约束：

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

为一个表增加一个自动命名的主键约束，注意一个表只能拥有一个主键：

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

把一个表移动到一个不同的表空间：

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

把一个表移动到一个不同的模式：

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

重建一个主键约束，并且在重建索引期间不阻塞更新：

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
    ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

要把一个分区附加到范围分区表上：

```
ALTER TABLE measurement
    ATTACH PARTITION measurement_y2016m07 FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

要把一个分区附加到列表分区表上：

```
ALTER TABLE cities
    ATTACH PARTITION cities_ab FOR VALUES IN ('a', 'b');
```

要把一个分区附加到 hash 分区表上：

```
ALTER TABLE orders
    ATTACH PARTITION orders_p4 FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

要把默认分区附加到分区表上：

```
ALTER TABLE cities
    ATTACH PARTITION cities_partdef DEFAULT;
```

从一个分区表分离一个分区：

```
ALTER TABLE measurement
    DETACH PARTITION measurement_y2015m12;
```

兼容性

以下形式符合 SQL 标准：`ADD`（不带 `USING INDEX`）、`DROP [COLUMN]`、`DROP IDENTITY`、`RESTART`、`SET DEFAULT`、`SET DATA TYPE`（不带 `USING`）、`SET GENERATED` 以及 `SET` *`sequence_option`*。其他形式是 PostgreSQL 对 SQL 标准的扩展。此外，在单条`ALTER TABLE` 命令中指定多个操作的能力也是一种扩展。

`ALTER TABLE DROP COLUMN` 可以被用来删除一个表的唯一的列，从而留下一个零列的表。这是一种 SQL 的扩展，SQL 中不允许零列的表。

另见

CREATE TABLE

ALTER TABLESPACE

ALTER TABLESPACE — 更改一个表空间的定义

大纲

```
ALTER TABLESPACE name RENAME TO new_name
ALTER TABLESPACE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TABLESPACE name SET ( tablespace_option = value [, ... ] )
ALTER TABLESPACE name RESET ( tablespace_option [, ... ] )
```

描述

`ALTER TABLESPACE` 可以用于更改表空间的定义。

要更改表空间的定义，必须拥有该表空间。要更改所有者，还必须是新所有者角色的直接或间接成员。（请注意，超级用户会自动拥有这些权限。）

参数

name

现有表空间的名称。

new_name

该表空间的新名称。新名称不能以 `pg_`开头，因为这类名称保留给系统表空间使用。

new_owner

该表空间的新拥有者。

tablespace_option

要设置或重置的表空间参数。目前唯一可用的参数是 `seq_page_cost`、`random_page_cost` 和 `effective_io_concurrency`。为特定表空间设置其中任一值，会覆盖规划器根据同名配置参数（请参见 seq_page_cost、random_page_cost、effective_io_concurrency）建立的、读取该表空间中表页面的通常代价估计。如果一个表空间位于比 I/O 子系统其余部分更快或更慢的磁盘上，这可能很有用。

示例

将表空间 `index_space` 重命名为 `fast_raid`：

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

更改表空间 `index_space` 的拥有者：

```
ALTER TABLESPACE index_space OWNER TO mary;
```

兼容性

在 SQL 标准中没有 `ALTER TABLESPACE` 语句。

另见

CREATE TABLESPACE, DROP TABLESPACE

ALTER TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH CONFIGURATION — 更改一个文本检索配置的定义

大纲

```
ALTER TEXT SEARCH CONFIGURATION name
    ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ]
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name
ALTER TEXT SEARCH CONFIGURATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema
```

描述

`ALTER TEXT SEARCH CONFIGURATION` 更改一个文本检索配置的定义。你可以修改其从词元类型到词典的映射，或者更改该配置的名称或拥有者。

要使用 `ALTER TEXT SEARCH CONFIGURATION`，你必须是该配置的拥有者。

参数

name

一个现有文本检索配置的名称（可以是模式限定的）。

token_type

由该配置的解析器发出的词元类型的名称。

dictionary_name

用于指定词元类型的文本检索词典名称。如果列出了多个词典，将按指定顺序依次查阅它们。

old_dictionary

在映射中要替换的文本检索词典的名称。

new_dictionary

被用来替代 *old_dictionary* 的文本检索词典的名称。

new_name

该文本检索配置的新名称。

new_owner

该文本检索配置的新拥有者。

new_schema

该文本检索配置的新模式。

ADD MAPPING FOR 形式会为指定词元类型设置要查阅的词典列表；如果任一词元类型已经有映射，则会报错。ALTER MAPPING FOR 形式作用相同，但会先移除这些词元类型的任何现有映射。ALTER MAPPING REPLACE 形式会在 *old_dictionary* 出现的任何位置，用 *new_dictionary* 替换它。如果出现 FOR，则只对指定的词元类型这样做；如果未出现 FOR，则对该配置的所有映射这样做。DROP MAPPING 形式会移除指定词元类型的所有词典，使这些类型的词元被文本检索配置忽略。除非出现IF EXISTS，否则如果这些词元类型没有映射就会报错。

示例

下面的示例将 my_config 中凡是使用 english 词典的地方，都替换为swedish 词典。

```
ALTER TEXT SEARCH CONFIGURATION my_config
  ALTER MAPPING REPLACE english WITH swedish;
```

兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH CONFIGURATION 语句。

另见

CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH DICTIONARY — 更改一个文本检索词典的定义

大纲

```
ALTER TEXT SEARCH DICTIONARY name (
    option [ = value ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name
ALTER TEXT SEARCH DICTIONARY name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema
```

描述

`ALTER TEXT SEARCH DICTIONARY` 更改文本检索词典的定义。你可以更改该词典的模板相关选项，也可以更改该词典的名称或拥有者。

要使用 `ALTER TEXT SEARCH DICTIONARY`，你必须是该词典的拥有者。

参数

name

一个现有文本检索词典的名称（可以是模式限定的）。

option

要为此词典设置的模板相关选项的名称。

value

模板相关选项要使用的新值。如果省略等号和值，则会从该词典中移除该选项之前的设置，从而允许使用默认值。

new_name

该文本检索词典的新名称。

new_owner

该文本检索词典的新拥有者。

new_schema

该文本检索词典的新模式。

模板相关选项可以以任意顺序出现。

示例

下面的示例命令更改了一个基于 Snowball 的词典的停用词列表。其他参数保持不变。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
```

下面的示例命令将语言选项更改为 dutch，并完全移除了停用词选项。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

下面的示例命令“更新”了该词典的定义，但实际上并没有做任何更改。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

（之所以可行，是因为选项移除代码在不存在该选项时也不会报错。）这种技巧在修改该词典的配置文件时很有用：ALTER 会强制现有数据库会话重新读取配置文件，而如果它们先前已经读取过这些文件，本来是不会再次读取的。

兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH DICTIONARY 语句。

另见

CREATE TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH PARSER

ALTER TEXT SEARCH PARSER — 更改一个文本检索解析器的定义

大纲

```
ALTER TEXT SEARCH PARSER name RENAME TO new_name
ALTER TEXT SEARCH PARSER name SET SCHEMA new_schema
```

描述

`ALTER TEXT SEARCH PARSER` 更改文本检索解析器的定义。当前唯一支持的功能是更改该解析器的名称。

要使用 `ALTER TEXT SEARCH PARSER`，你必须是超级用户。

参数

name

一个现有文本检索解析器的名称（可以是模式限定的）。

new_name

该文本检索解析器的新名称。

new_schema

该文本检索解析器的新模式。

兼容性

SQL 标准中没有 `ALTER TEXT SEARCH PARSER` 语句。

另见

CREATE TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

ALTER TEXT SEARCH TEMPLATE

ALTER TEXT SEARCH TEMPLATE — 更改一个文本检索模板的定义

大纲

```
ALTER TEXT SEARCH TEMPLATE name RENAME TO new_name
ALTER TEXT SEARCH TEMPLATE name SET SCHEMA new_schema
```

描述

`ALTER TEXT SEARCH TEMPLATE` 更改文本检索模板的定义。当前唯一支持的功能是更改该模板的名称。

要使用 `ALTER TEXT SEARCH TEMPLATE`，你必须是超级用户。

参数

name

一个现有文本检索模板的名称（可以是模式限定的）。

new_name

该文本检索模板的新名称。

new_schema

该文本检索模板的新模式。

兼容性

SQL 标准中没有 `ALTER TEXT SEARCH TEMPLATE` 语句。

另见

CREATE TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

ALTER TRIGGER

ALTER TRIGGER — 更改触发器的定义

大纲

```
ALTER TRIGGER name ON table_name RENAME TO new_name
ALTER TRIGGER name ON table_name DEPENDS ON EXTENSION extension_name
```

描述

`ALTER TRIGGER` 更改现有触发器的属性。`RENAME` 子句更改给定触发器的名称，但不以其他方式更改触发器定义。`DEPENDS ON EXTENSION` 子句将触发器标记为依赖扩展，因此扩展被删除时触发器也会自动删除。

若要更改触发器的属性，你必须拥有该触发器所作用的表。

参数

name

要修改的现有触发器的名称。

table_name

该触发器所作用的表的名称。

new_name

该触发器的新名称。

extension_name

触发器所依赖的扩展名称。

注解

暂时启用或禁用触发器的功能由 ALTER TABLE 提供，而不是 `ALTER TRIGGER`，因为 `ALTER TRIGGER` 没有方便的方式来表达一次性启用或禁用一个表上全部触发器这一选项。

示例

要重命名一个现有触发器：

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

要将一个触发器标记为依赖于某个扩展：

```
ALTER TRIGGER emp_stamp ON emp DEPENDS ON EXTENSION emplib;
```

兼容性

`ALTER TRIGGER` 是 PostgreSQL 对 SQL 标准的扩展。

另见

ALTER TABLE

ALTER TYPE

ALTER TYPE — 更改类型的定义

大纲

```
ALTER TYPE name action [, ... ]
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE | RESTRICT ]
ALTER TYPE name RENAME TO new_name
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE | AFTER } neighbor_enum_
value ]
ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value
```

其中 *action* 为以下之一:

```
    ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE | RESTRICT ]
    DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
    ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ CASCADE
 | RESTRICT ]
```

描述

`ALTER TYPE` 更改现有类型的定义。有几种子形式:

`ADD ATTRIBUTE`

该形式使用与 CREATE TYPE 相同的语法向复合类型添加新属性。

`DROP ATTRIBUTE [ IF EXISTS ]`

这种形式从复合类型中删除一个属性。如果指定了 `IF EXISTS` 而该属性不存在，则不会抛出错误，而是发出一条提示。

`SET DATA TYPE`

这种形式更改复合类型中某个属性的数据类型。

`OWNER`

这种形式更改类型的拥有者。

`RENAME`

该形式更改类型的名称或复合类型中单个属性的名称。

`SET SCHEMA`

这种形式将类型移动到另一个模式中。

`ADD VALUE [ IF NOT EXISTS ] [ BEFORE | AFTER ]`

这种形式向枚举类型中添加一个新值。可以用 `BEFORE` 或 `AFTER` 指定该新值在枚举排序中的位置，即位于某个现有值之前或之后。否则，新项会被添加到值列表的末尾。

如果指定了 IF NOT EXISTS，而该类型已经包含该新值，则不会报错：系统会发出一条提示，但不会执行任何其他操作。否则，如果该新值已存在，就会报错。

RENAME VALUE

这种形式重命名枚举类型中的一个值。该值在枚举排序中的位置不会受到影响。如果指定的值不存在，或者新名称已经存在，则会报错。

ADD ATTRIBUTE、DROP ATTRIBUTE 和 ALTER ATTRIBUTE 操作可以组合成一个包含多项修改的列表，以便并行应用。例如，可以在一条命令中添加多个属性和/或更改多个属性的数据类型。

要使用 ALTER TYPE，必须拥有该类型。要更改类型的模式，还必须在新模式上拥有CREATE 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在类型所在模式上拥有CREATE 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建类型完成的操作。不过，超级用户无论如何都可以更改任何类型的所有权。）要添加属性或更改属性类型，还必须在该属性的数据类型上拥有USAGE 权限。

参数

name

要修改的现有类型的名称（可以是模式限定的）。

new_name

该类型的新名称。

new_owner

该类型新的拥有者的用户名。

new_schema

该类型的新模式。

attribute_name

要添加、修改或删除的属性名称。

new_attribute_name

要重命名的属性的新名称。

data_type

要添加的属性的数据类型，或者要修改的属性的新类型。

new_enum_value

要添加到枚举类型值列表中的新值，或者要赋给现有值的新名称。与所有枚举字面量一样，它必须加引号。

neighbor_enum_value

一个现有枚举值；在枚举类型的排序顺序中，新值会紧邻该值，被添加到它之前或之后。与所有枚举字面量一样，它必须加引号。

existing_enum_value

应当被重命名的现有枚举值。与所有枚举字面量一样，它必须加引号。

CASCADE

自动将该操作传播到正在修改的类型的类型化表及其后代。

RESTRICT

如果正在修改的类型是某个类型化表的类型，则拒绝该操作。这是默认行为。

注解

ALTER TYPE ... ADD VALUE（向枚举类型添加新值的形式）不能在事务块内执行。

涉及新增枚举值的比较有时会比仅涉及该枚举类型原有成员的比较更慢。这通常只会在使用 BEFORE 或 AFTER 将新值的排序位置设在列表末尾之外时发生。不过，即使新值被添加到末尾，有时也会出现这种情况（如果自最初创建该枚举类型以来 OID 计数器已经“回卷”，就会发生这种情况）。这种变慢通常无关紧要；但如果它确实重要，可以通过删除并重新创建该枚举类型，或者转储并恢复数据库，来恢复最佳性能。

示例

要重命名一个数据类型：

```
ALTER TYPE electronic_mail RENAME TO email;
```

要将类型 email 的拥有者改为 joe：

```
ALTER TYPE email OWNER TO joe;
```

要将类型 email 所在的模式改为 customers：

```
ALTER TYPE email SET SCHEMA customers;
```

向类型添加新属性：

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

要在枚举类型的特定排序位置添加一个新值：

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

要重命名一个枚举值：

```
ALTER TYPE colors RENAME VALUE 'purple' TO 'mauve';
```

兼容性

添加和删除属性的这些变体属于 SQL 标准；其他变体则是 PostgreSQL 扩展。

另见

CREATE TYPE, DROP TYPE

ALTER USER

ALTER USER — 更改数据库角色

大纲

```
ALTER USER role_specification [ WITH ] option [ ... ]
```

其中 *option* 可以是:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'

ALTER USER name RENAME TO new_name

ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_
parameter { TO | = } { value | DEFAULT }
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_
parameter FROM CURRENT
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] RESET configuration_
parameter
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL
```

其中 *role_specification* 可以是:

```
    role_name
  | CURRENT_USER
  | SESSION_USER
```

描述

`ALTER USER` 现为 ALTER ROLE 的别名。

兼容性

`ALTER USER` 语句是 PostgreSQL 的一种扩展。SQL 标准将用户的定义交由具体实现决定。

另见

ALTER ROLE

ALTER USER MAPPING

ALTER USER MAPPING — 更改用户映射的定义

大纲

```
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | SESSION_USER | PUBLIC }
    SERVER server_name
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

描述

`ALTER USER MAPPING` 更改用户映射的定义。

外部服务器的拥有者可以修改任意用户到该服务器的用户映射。此外，如果某个用户已被授予该服务器上的`USAGE` 权限，那么该用户也可以修改其自己用户名对应的用户映射。

参数

user_name

映射的用户名。`CURRENT_USER` 和 `USER` 与当前用户名称匹配。`PUBLIC` 用于匹配系统中当前及未来的所有用户名。

server_name

该用户映射所属服务器的名称。

`OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )`

更改该用户映射的选项。新选项会覆盖先前指定的任何选项。`ADD`、`SET` 和 `DROP` 指定要执行的动作。如果未显式指定操作，则假定为`ADD`。选项名必须唯一；这些选项还会由该服务器的外部数据包装器进行验证。

示例

更改用户映射 `bob`、服务器`foo` 的密码：

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (SET password 'public');
```

兼容性

`ALTER USER MAPPING` 符合 ISO/IEC 9075-9 (SQL/MED)。这里有一个细微的语法问题：该标准省略了 `FOR` 关键字。由于 `CREATE USER MAPPING` 和 `DROP USER MAPPING` 都在类似位置使用 `FOR`，而 IBM DB2（另一个主要的 SQL/MED 实现）也要求在 `ALTER USER MAPPING` 中使用它，因此为了保持一致性和互操作性，PostgreSQL 在这里偏离了标准。

另见

CREATE USER MAPPING, DROP USER MAPPING

ALTER VIEW

ALTER VIEW — 更改视图的定义

大纲

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET DEFAULT expression
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER VIEW [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value] [, ... ] )
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

描述

`ALTER VIEW` 更改视图的各种辅助属性。（如果要修改视图的定义查询，请使用 `CREATE OR REPLACE VIEW`。）

要使用 `ALTER VIEW`，必须拥有该视图。要更改视图的模式，还必须在新模式上拥有`CREATE` 权限。要更改所有者，还必须是新所有者角色的直接或间接成员，并且该角色必须在视图所在模式上拥有`CREATE` 权限。（这些限制确保更改所有者不会执行任何无法通过删除并重新创建视图完成的操作。不过，超级用户无论如何都可以更改任何视图的所有权。）

参数

name

现有视图的名称（可以带模式限定）。

`IF EXISTS`

如果视图不存在，则不抛出错误，而是发出一条提示。

`SET/DROP DEFAULT`

这些形式用于设置或移除列的默认值。对于任何以该视图为目标的 `INSERT` 或 `UPDATE` 命令，视图列的默认值都会在应用该视图上的任何规则或触发器之前被代入。因此，视图的默认值将优先于底层关系中的任何默认值。

new_owner

视图新拥有者的用户名。

new_name

该视图的新名称。

new_schema

该视图的新模式。

```
SET ( view_option_name [= view_option_value] [, ... ] )
RESET ( view_option_name [, ... ] )
```

设置或重置视图选项。目前支持的选项有：

`check_option` (`string`)

更改视图的检查选项。该值必须为 `local` 或 `cascaded`。

`security_barrier` (`boolean`)

更改视图的安全屏障属性。该值必须为布尔值，例如 `true` 或 `false`。

注解

出于历史原因，视图也可以使用 `ALTER TABLE`；但允许用于视图的 `ALTER TABLE` 变体，仅限于与上面所示形式等价的那些。

示例

将视图 `foo` 重命名为 `bar`：

```
ALTER VIEW foo RENAME TO bar;
```

要为一个可更新视图附加默认列值：

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1);  -- ts 将得到 NULL
INSERT INTO a_view(id) VALUES(2);  -- ts 将得到当前时间
```

兼容性

`ALTER VIEW` 是 PostgreSQL 对 SQL 标准的扩展。

另见

CREATE VIEW, DROP VIEW

ANALYZE

ANALYZE — 收集数据库的统计信息

大纲

```
ANALYZE [ ( option [, ...] ) ] [ table_and_columns [, ...] ]
ANALYZE [ VERBOSE ] [ table_and_columns [, ...] ]
```

其中 *option* 为以下之一：

```
    VERBOSE
```

而 *table_and_columns* 为：

```
    table_name [ ( column_name [, ...] ) ]
```

描述

ANALYZE 收集数据库中各表内容的统计信息，并将结果存储到pg_statistic 系统目录中。随后，查询规划器会使用这些统计信息来帮助确定查询的最高效执行计划。

如果没有给出 *table_and_columns* 列表，ANALYZE 会处理当前数据库中当前用户有权分析的每个表和物化视图。如果给出了列表，ANALYZE 则只处理其中列出的表。还可以为某个表给出列名列表，这时只会收集这些列的统计信息。

当选项列表用圆括号括起时，选项可以按任意顺序书写。带圆括号的语法是在PostgreSQL 11 中加入的；不带圆括号的语法已弃用。

参数

VERBOSE

启用进度消息显示。

table_name

要分析的特定表的名称（可带模式限定）。如果省略，则分析当前数据库中的所有普通表、分区表和物化视图（但不包括外部表）。如果指定的表是分区表，则会同时更新整个分区表的继承统计信息和各个分区的统计信息。

column_name

要分析的特定列的名称。默认为所有列。

输出

指定 VERBOSE 时，ANALYZE 会输出进度消息，指示当前正在处理哪个表，同时还会打印这些表的各种统计信息。

注解

要分析一个表，通常调用者必须是该表的拥有者或超级用户。不过，数据库拥有者可以分析其数据库中的所有表，但共享系统目录除外。（对共享系统目录的这一限制意味着，真正意义上的全数据库 ANALYZE 只能由超级用户执行。）ANALYZE 会跳过调用用户无权分析的任何表。

只有在显式选中时才会分析外部表。并非所有外部数据包装器都支持 ANALYZE。如果该表的包装器不支持 ANALYZE，命令会打印一条警告且不执行任何操作。

在默认的 PostgreSQL 配置中，自动清理守护进程（见第 24.1.6 节）会在表首次装载数据时，以及在常规运行过程中数据发生变化时，自动分析这些表。当自动清理被禁用时，最好定期运行 ANALYZE，或者在对表内容做出重大更改之后立即运行。准确的统计信息有助于规划器选择最合适的查询计划，从而提高查询处理速度。对于以读取为主的数据库，一个常见策略是在每天使用率较低的时段运行一次VACUUM 和 ANALYZE。（如果更新活动很频繁，这样做仍然不够。）

ANALYZE 只需获取目标表上的读锁，因此可以与该表上的其他活动并行运行。

ANALYZE 收集的统计信息通常包括每列中某些高频值的列表，以及显示每列大致数据分布的直方图。如果ANALYZE 认为它们没有意义（例如，唯一键列中没有高频值），或者列的数据类型不支持相应的操作符，则可能省略其中一项或两项。更多统计信息见第 24 章。

对于大型表，ANALYZE 会对表内容进行随机采样，而不是检查每一行。这使得即使是很大的表，也能在较短时间内完成分析。不过要注意，这些统计信息只是近似值，并且即使实际表内容没有变化，每次运行ANALYZE 时统计信息也会略有变化。这可能导致 EXPLAIN 中显示的规划器估算代价略有变化。在少数情况下，这种非确定性会导致规划器在运行ANALYZE 之后改用不同的查询计划。要避免这种情况，可以按下文所述提高ANALYZE 收集的统计信息量。

可以通过调整配置变量 default_statistics_target 来控制分析程度，也可以针对单个列使用 ALTER TABLE ... ALTER COLUMN ... SET STATISTICS 设置每列的统计信息目标（参见 ALTER TABLE）。目标值会设置高频值列表中的最大条目数，以及直方图中的最大桶数。默认目标值是 100，但可以把它调高或调低，以在规划器估算精度、ANALYZE 耗费的时间以及 pg_statistic 占用的空间之间作出权衡。特别地，将统计信息目标设置为零会禁用该列的统计信息收集。对于那些从不出现在查询 WHERE、GROUP BY 或 ORDER BY 子句中的列，这样做可能很有用，因为规划器不会用到这些列的统计信息。

被分析列中最大的统计信息目标决定了为了生成统计信息而需要采样的表行数。增加该目标会导致执行 ANALYZE 所需的时间和空间按比例增加。

ANALYZE 估算的值之一是每一列中出现的非重复值数量。由于只检查了部分行，即使使用可能的最大统计信息目标，这种估计有时也可能相当不精确。如果这种不精确导致查询计划不佳，就可以手工确定一个更精确的值，然后用 ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...) 把该值设置进去（参见 ALTER TABLE）。

如果被分析的表有继承子表，ANALYZE 会收集两组统计信息：一组只针对父表中的行，另一组则同时包含父表及其所有子表中的行。规划处理整个继承树的查询时，需要第二组统计信息。在这种情况下，不会分别分析子表本身。不过，自动清理守护进程在决定是否为该表触发自动分析时，只会考虑对父表本身的插入或更新。如果该表很少被插入或更新，那么除非手工运行 ANALYZE，否则继承统计信息就不会保持最新。

对于分区表，ANALYZE 会通过对所有分区中的行进行采样来收集统计信息；此外，它还会递归进入每个分区并更新其统计信息。即使有多级分区，每个叶子分区也只分析一次。不会仅为父表收集统计信息（不含分区数据），因为分区机制保证父表为空。

自动清理守护进程不会处理分区表；如果继承体系中只有子表被修改，它也不会处理继承父表。因此，通常需要定期手工运行ANALYZE，以保持表层次结构的统计信息为最新状态。

如果某些子表或分区是外部表，而其外部数据包装器不支持ANALYZE，那么在收集继承统计信息时会忽略这些表。

如果被分析的表完全为空，ANALYZE 将不会为该表记录新的统计信息。任何现有统计信息都会被保留。

兼容性

SQL 标准中没有 ANALYZE 语句。

另见

VACUUM, vacuumdb, 第 19.4.4 节, 第 24.1.6 节

BEGIN

BEGIN — 开始一个事务块

大纲

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
```

其中 *transaction_mode* 是下列之一:

```
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

BEGIN 启动一个事务块，也就是说，在 BEGIN 命令之后的所有语句都会在同一个事务中执行，直到显式给出COMMIT 或 ROLLBACK 为止。默认情况下（不使用 BEGIN），PostgreSQL 以“自动提交”模式执行事务，也就是说，每条语句都在自己的事务中执行，并在语句结束时隐式提交（如果执行成功，否则回滚）。

在事务块中执行语句通常会更快，因为事务的启动和提交需要大量的 CPU 和磁盘活动。在进行多个相关更改时，把多条语句放在一个事务中执行也有助于保证一致性：其他会话将无法看到那些相关更新尚未全部完成时的中间状态。

如果指定了隔离级别、读写模式或可延迟模式，新事务就会具有这些特征，就像执行了SET TRANSACTION 一样。

参数

WORK
TRANSACTION

可选关键字，没有任何作用。

关于本语句其他参数含义的信息，参见 SET TRANSACTION。

注解

START TRANSACTION 与 BEGIN 具有相同功能。

使用 COMMIT 或 ROLLBACK 可以结束一个事务块。

如果在已经处于事务块中时发出 BEGIN，将会产生一条警告消息。事务状态不会受影响。若要在事务块中实现事务嵌套，请使用保存点（参见 SAVEPOINT）。

出于向后兼容的考虑，连续多个*transaction_modes* 之间的逗号可以省略。

示例

开始一个事务块：

```
BEGIN;
```

兼容性

`BEGIN` 是 PostgreSQL 的一种语言扩展。它等价于 SQL 标准命令 START TRANSACTION，后者的参考页中包含更多兼容性信息。

`DEFERRABLE` *`transaction_mode`* 是 PostgreSQL 的一种语言扩展。

另外，在嵌入式 SQL 中，关键字`BEGIN` 用于不同的目的。在移植数据库应用程序时，应当谨慎处理事务语义。

另见

COMMIT, ROLLBACK, START TRANSACTION, SAVEPOINT

CALL

CALL — 调用一个过程

大纲

```
CALL name ( [ argument ] [, ...] )
```

描述

`CALL` 执行一个过程。

如果该过程有任何输出参数，则会返回一行结果，其中包含这些参数的值。

参数

name

过程的名称（可以是模式限定的）。

argument

过程调用的一个输入参数。有关函数和过程调用语法的完整细节（包括命名参数的使用），请参阅第 4.3 节。

注解

用户必须拥有该过程的 `EXECUTE` 权限，才能调用它。

若要调用函数（而不是过程），请改用`SELECT`。

如果在事务块中执行 `CALL`，则被调用的过程不能执行事务控制语句。只有当`CALL` 作为独立事务执行时，才允许使用事务控制语句。

PL/pgSQL 对 `CALL` 命令中的输出参数采用不同的处理方式；参见第 43.6.3 节。

示例

```
CALL do_db_maintenance();
```

兼容性

`CALL` 符合 SQL 标准。

另见

CREATE PROCEDURE

CHECKPOINT

CHECKPOINT — 强制执行一次预写式日志检查点

大纲

```
CHECKPOINT
```

描述

检查点是预写式日志序列中的一个位置，到这一位置时，所有数据文件都已更新，以反映日志中的信息。所有数据文件都会被刷写到磁盘。有关检查点期间会发生什么的更多细节，参见第 30.4 节。

`CHECKPOINT` 命令会在发出该命令时强制立即执行检查点，而不是等待系统安排的常规检查点（由第 19.5.2 节中的设置控制）。`CHECKPOINT` 并非设计用于正常运行期间。

如果在恢复期间执行 `CHECKPOINT` 命令，它将强制执行一个重启点（见第 30.4 节），而不是写入一个新的检查点。

只有超级用户才能调用 `CHECKPOINT`。

兼容性

`CHECKPOINT` 命令是 PostgreSQL 的一种语言扩展。

CLOSE

CLOSE — 关闭一个游标

大纲

```
CLOSE { name | ALL }
```

描述

CLOSE 释放与一个打开的游标相关联的资源。游标关闭后，不允许再对其执行任何后续操作。当不再需要游标时，应将其关闭。

每个不可保持的打开游标都会在事务通过 COMMIT 或 ROLLBACK 结束时被隐式关闭。可保持游标会在创建它的事务通过 ROLLBACK 中止时被隐式关闭。如果创建它的事务成功提交，则该可保持游标会一直保持打开状态，直到显式执行 CLOSE，或者客户端断开连接。

参数

name

要关闭的打开游标的名称。

ALL

关闭所有打开的游标。

注解

PostgreSQL 没有显式的 OPEN 游标语句；游标在声明时即被视为打开。请使用 DECLARE 语句来声明游标。

可以通过查询 pg_cursors 系统视图查看所有可用游标。

如果在某个保存点之后关闭了游标，而该保存点后来又被回滚，则CLOSE 不会被回滚；也就是说，该游标仍然保持关闭状态。

示例

关闭游标 liahona：

```
CLOSE liahona;
```

兼容性

CLOSE 完全符合 SQL 标准。CLOSE ALL 是 PostgreSQL 的一种扩展。

另见

DECLARE, FETCH, MOVE

CLUSTER

CLUSTER — 按照一个索引对表进行聚簇

大纲

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

描述

`CLUSTER` 指示 PostgreSQL 按照 *`index_name`* 指定的索引，对 *`table_name`* 指定的表进行聚簇。该索引必须已经定义在 *`table_name`* 上。

当一个表被聚簇时，它会根据索引信息在物理上重新排序。聚簇是一次性操作：之后如果表再被更新，这些更改不会再次被聚簇。也就是说，系统不会试图按照索引顺序存储新行或更新后的行。（如果需要，可以定期再次执行该命令来重新聚簇。此外，将表的 `fillfactor` 存储参数设置为小于 100%，有助于在更新期间保持聚簇顺序，因为如果页面上有足够空间，更新后的行会保留在同一页面中。）

当一个表被聚簇时，PostgreSQL 会记住该表是按哪个索引聚簇的。形式 `CLUSTER` *`table_name`* 会使用与之前相同的索引对表重新聚簇。你也可以使用 ALTER TABLE 的 `CLUSTER` 或 `SET WITHOUT CLUSTER` 形式，设置未来聚簇操作要使用的索引，或者清除任何先前的设置。

不带任何参数的 `CLUSTER` 会对当前数据库中所有此前已聚簇且属于调用用户的表重新执行聚簇；如果由超级用户调用，则会对所有此前已聚簇的表重新执行聚簇。这种形式的 `CLUSTER` 不能在事务块内执行。

当对一个表执行聚簇时，会在该表上获取 `ACCESS EXCLUSIVE` 锁。这会阻止任何其他数据库操作（包括读和写）在 `CLUSTER` 完成前访问该表。

参数

`table_name`

表的名称（可能是模式限定的）。

`index_name`

索引的名称。

`VERBOSE`

在每个表被聚簇时打印进度报告。

注解

当你在表中随机访问单行时，数据在表中的实际顺序并不重要。但是，如果访问某些数据比其他数据更频繁，并且存在将这些数据放在一起的索引，使用`CLUSTER` 就会有益。如果要从表中获取

一段索引值范围，或者获取有多行匹配的单个索引值，CLUSTER 会有帮助，因为一旦索引确定第一个匹配行所在的表页，其他匹配行很可能已在同一个表页中，从而减少磁盘访问并加快查询。

CLUSTER 可以通过在指定索引上进行索引扫描，或者（如果该索引是 B-树）先执行顺序扫描再排序，来对表重新排序。它会根据规划器代价参数和可用的统计信息，尝试选择速度更快的方法。

使用索引扫描时，会创建一个表的临时副本，其中表数据按索引顺序排列。还会为表上的每个索引创建临时副本。因此，所需的磁盘空闲空间至少应等于表大小与索引大小之和。

使用顺序扫描加排序时，还会创建临时排序文件，因此临时空间需求峰值可达表大小的两倍再加上索引大小。这种方法通常比索引扫描更快，但如果无法接受其磁盘空间需求，可以临时将 enable_sort 设置为 off 来禁用这种选择。

建议在聚簇之前将 maintenance_work_mem 设置为一个合理的大值（但不超过可专门用于 CLUSTER 操作的内存量）。

因为规划器会记录有关表中数据顺序的统计信息，建议在新近聚簇过的表上运行 ANALYZE。否则，规划器可能会选择很差的查询计划。

因为 CLUSTER 会记住哪些索引已被设为聚簇索引，你可以第一次先手工聚簇需要聚簇的表，然后设置一个定期运行的维护脚本，执行不带任何参数的 CLUSTER，这样这些表就会被周期性地重新聚簇。

示例

按照索引 employees_ind 对表 employees 进行聚簇：

```
CLUSTER employees USING employees_ind;
```

使用之前用过的同一个索引对 employees 表进行聚簇：

```
CLUSTER employees;
```

对数据库中此前已聚簇过的所有表执行聚簇：

```
CLUSTER;
```

兼容性

SQL 标准中没有 CLUSTER 语句。

语法

```
CLUSTER index_name ON table_name
```

也受支持，以兼容 8.3 之前的 PostgreSQL 版本。

另见

clusterdb

COMMENT

COMMENT — 定义或修改对象的注释

大纲

```
COMMENT ON
{
  ACCESS METHOD object_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  COLUMN relation_name.column_name |
  CONSTRAINT constraint_name ON table_name |
  CONSTRAINT constraint_name ON DOMAIN domain_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  EXTENSION object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  POLICY policy_name ON table_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  PUBLICATION object_name |
  ROLE object_name |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  STATISTICS object_name |
  SUBSCRIPTION object_name |
  TABLE object_name |
  TABLESPACE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name
} IS { string_literal | NULL }
```

其中 *aggregate_signature* **是**:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ...
 ]
```

描述

COMMENT 存储数据库对象的注释。

每个对象只存储一个注释字符串，因此要修改注释，请对同一对象发出新的COMMENT 命令。要移除注释，请用NULL 代替文本字符串。对象删除时，其注释也会自动删除。

将在要添加注释的对象上获取一个 SHARE UPDATE EXCLUSIVE 锁。

对于大多数对象类型，只有对象的拥有者才能设置注释。角色没有拥有者，因此 COMMENT ON ROLE 的规则是：要为超级用户角色添加注释，必须是超级用户；要为非超级用户角色添加注释，必须拥有CREATEROLE 权限。同样，访问方法也没有拥有者；要为访问方法添加注释，必须是超级用户。当然，超级用户可以为任何对象添加注释。

可以使用 psql 的\d 系列命令查看这些注释。其他用于检索注释的用户界面，也可以建立在 psql 使用的同一组内置函数之上，即 obj_description、col_description 和 shobj_description（见表 9.68）。

参数

object_name
relation_name.column_name
aggregate_name
constraint_name
function_name
operator_name
policy_name
procedure_name
routine_name
rule_name
trigger_name

要添加注释的对象名称。表、聚合、排序规则、转换、域、外部表、函数、索引、操作符、操作符类、操作符族、过程、例程、序列、统计信息、文本检索对象、类型和视图的名称可以带模式限定。为列添加注释时，*relation_name* 必须引用表、视图、复合类型或外部表。

table_name
domain_name

当为约束、触发器、规则或策略创建注释时，这些参数指定定义该对象的表或域的名称。

source_type

该类型转换的源数据类型的名称。

target_type

该类型转换的目标数据类型的名称。

argmode

函数、过程或聚合函数参数的模式：IN、OUT、INOUT 或 VARIADIC。如果省略，默认值是 IN。注意 COMMENT 实际上并不关心 OUT 参数，因为确定函数标识只需要输入参数。因此，列出IN、INOUT 和 VARIADIC 参数就足够了。

argname

函数、过程或聚合函数参数的名称。注意COMMENT 实际上并不关心参数名称，因为确定函数标识只需要参数数据类型。

argtype

函数、过程或聚合函数参数的数据类型。

large_object_oid

大对象的 OID。

left_type
right_type

操作符参数的数据类型（可带模式限定）。对于前缀或后缀操作符缺失的参数，请写NONE。

PROCEDURAL

这是一个噪声词。

type_name

该转换的数据类型名称。

lang_name

该转换所用语言的名称。

string_literal

新的注释内容，以字符串字面量形式书写。

NULL

写入 NULL 以移除该注释。

注解

目前没有用于查看注释的安全机制：任何连接到某个数据库的用户都可以看到该数据库中所有对象的注释。对于数据库、角色和表空间这类共享对象，注释是全局存储的，因此连接到集簇中任一数据库的任何用户都可以看到共享对象的全部注释。因此，不要在注释中放入与安全密切相关的信息。

示例

为表 mytable 附加一条注释：

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

再将其移除：

```
COMMENT ON TABLE mytable IS NULL;
```

更多示例：

```
COMMENT ON ACCESS METHOD gin IS 'GIN index access method';
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample variance';
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
COMMENT ON COLLATION "fr_CA" IS 'Canadian French';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Constrains column col';
COMMENT ON CONSTRAINT dom_col_constr ON DOMAIN dom IS 'Constrains col of domain';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON EVENT TRIGGER abort_ddl IS 'Aborts all DDL commands';
COMMENT ON EXTENSION hstore IS 'implements the hstore data type';
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'my foreign data wrapper';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Employee Information in other database';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON MATERIALIZED VIEW my_matview IS 'Summary of order history';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, integer) IS 'Unary minus';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators for btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'all integer operators for btrees';
COMMENT ON POLICY my_policy ON mytable IS 'Filter rows by users';
COMMENT ON PROCEDURE my_proc (integer, integer) IS 'Runs a report';
COMMENT ON PUBLICATION alltables IS 'Publishes all operations on all tables';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON ROUTINE my_routine (integer, integer) IS 'Runs a routine (which is a function or
 procedure)';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON SERVER myserver IS 'my foreign server';
COMMENT ON STATISTICS my_statistics IS 'Improves planner row estimations';
COMMENT ON SUBSCRIPTION alltables IS 'Subscription for all operations on all tables';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Special word filtering';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Snowball stemmer for Swedish language';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Splits text into words';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Snowball stemmer';
COMMENT ON TRANSFORM FOR hstore LANGUAGE plpythonu IS 'Transform between hstore and Python
 dict';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

兼容性

SQL 标准中没有 `COMMENT` 命令。

COMMIT

COMMIT — 提交当前事务

大纲

```
COMMIT [ WORK | TRANSACTION ]
```

描述

`COMMIT` 提交当前事务。该事务所做的所有更改都会对其他会话可见，并且如果发生崩溃，也能保证其持久性。

参数

`WORK`
`TRANSACTION`

可选关键字，没有任何作用。

注解

使用 ROLLBACK 可中止一个事务。

在事务之外发出 `COMMIT` 不会造成危害，但会产生一条警告消息。

示例

提交当前事务，并使所有更改永久生效：

```
COMMIT;
```

兼容性

SQL 标准只规定了 `COMMIT` 和 `COMMIT WORK` 这两种形式。在其他方面，此命令完全符合标准。

另见

BEGIN, ROLLBACK

COMMIT PREPARED

COMMIT PREPARED — 提交一个先前为两阶段提交而预备的事务

大纲

```
COMMIT PREPARED transaction_id
```

描述

`COMMIT PREPARED` 提交一个处于预备状态的事务。

参数

transaction_id

要提交的事务的事务标识符。

注解

要提交预备事务，执行者必须是最初执行该事务的同一用户，或者是超级用户；但不必处在执行该事务的同一会话中。

这个命令不能在事务块内执行。该预备事务会被立即提交。

当前所有处于预备状态的事务都列在 `pg_prepared_xacts` 系统视图中。

示例

提交事务标识符为 `foobar` 的事务：

```
COMMIT PREPARED 'foobar';
```

兼容性

`COMMIT PREPARED` 是 PostgreSQL 扩展。它旨在供外部事务管理系统使用，其中有些系统已被标准覆盖（例如 X/Open XA），但这些系统的 SQL 侧并未标准化。

另见

PREPARE TRANSACTION, ROLLBACK PREPARED

COPY

COPY — 在文件和表之间复制数据

大纲

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | PROGRAM 'command' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | PROGRAM 'command' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]
```

其中 *option* 为以下之一:

```
    FORMAT format_name
    OIDS [ boolean ]
    FREEZE [ boolean ]
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] )
    FORCE_NULL ( column_name [, ...] )
    ENCODING 'encoding_name'
```

描述

COPY 在 PostgreSQL 表与标准文件系统文件之间传输数据。COPY TO 将表的内容复制到文件，而COPY FROM 则将数据从文件复制到表中（追加到表中已有的数据之后）。COPY TO 也可以复制 SELECT 查询的结果。

如果指定了列列表，COPY TO 只会将指定列中的数据复制到文件。对于 COPY FROM，文件中的每个字段会按顺序插入到指定列中。未在COPY FROM 列列表中指定的表列将接收其默认值。

带文件名的 COPY 会指示 PostgreSQL 服务器直接从文件读取或向文件写入。该文件必须可由 PostgreSQL 用户（服务器运行时使用的用户 ID）访问，并且其名称必须从服务器的视角指定。当指定 PROGRAM 时，服务器会执行给定的命令，并从该程序的标准输出读取，或者向该程序的标准输入写入。该命令必须从服务器的视角指定，并且必须可由PostgreSQL 用户执行。指定 STDIN 或 STDOUT 时，数据通过客户端与服务器之间的连接传输。

参数

table_name

一个现有表的名称（可以是模式限定的）。

column_name

要复制的可选列列表。如果没有指定列列表，则会复制该表所有列。

query

其结果将被复制的 SELECT、VALUES、INSERT、UPDATE 或 DELETE 命令。注意，查询外层必须带圆括号。

对于 INSERT、UPDATE 和 DELETE 查询，必须提供 RETURNING 子句，并且目标关系不能有条件规则，也不能有 ALSO 规则，也不能有扩展为多个语句的 INSTEAD 规则。

filename

输入或输出文件的路径名。输入文件名可以是绝对路径或相对路径，但输出文件名必须是绝对路径。Windows 用户可能需要使用 E'' 字符串，并将路径名中的任何反斜线写成双反斜线。

PROGRAM

要执行的命令。在 COPY FROM 中，输入从该命令的标准输出读取；而在COPY TO 中，输出会写入该命令的标准输入。

注意，该命令由 shell 调用，因此如果需要向 shell 命令传递来自不可信来源的参数，必须谨慎移除或转义对 shell 可能有特殊意义的字符。出于安全考虑，最好使用固定命令字符串，或至少避免在其中传入任何用户输入。

STDIN

指定输入来自客户端应用。

STDOUT

指定输出发送到客户端应用。

boolean

指定所选选项是否开启。可以写 TRUE、ON 或 1 来启用选项，写FALSE、OFF 或 0 来禁用它。也可以省略 *boolean* 值，此时假定为TRUE。

FORMAT

选择要读取或写入的数据格式：text、csv（逗号分隔值）或 binary。默认为 text。

OIDS

指定复制每一行的 OID。（如果为没有 OID 的表指定了 OIDS，或者复制的是一个 *query*，就会报错。）

FREEZE

请求以行已冻结的状态复制数据，就像运行过VACUUM FREEZE 命令之后一样。这是用于初始数据装载的性能选项。只有当正在装载的表是在当前子事务中创建或截断的、没有打开的游标且该事务不持有更早的快照时，行才会被冻结。目前不能对分区表执行COPY FREEZE。

注意，一旦数据成功装载，所有其他会话将立即能够看到这些数据。这违反了正常的 MVCC 可见性规则，指定此选项的用户应注意可能由此引发的问题。

DELIMITER

指定分隔文件中每一行（记录）内各列的字符。文本格式中默认是一个制表符，而CSV 格式中默认是一个逗号。这必须是一个单一的单字节字符。使用 binary 格式时不允许这个选项。

NULL

指定表示一个空值的字符串。文本格式中默认是 \N（反斜线-N），CSV 格式中默认是一个未加引用的空串。在你不想区分空值和空串的情况下，即使在文本格式中你也可能更喜欢空串。使用binary 格式时不允许这个选项。

> **注意**
>
> 在使用 COPY FROM 时，任何匹配此字符串的数据项都会被存储为空值，因此应确保这里使用的字符串与 COPY TO 时使用的相同。

HEADER

指定文件包含一个标题行，其中列出文件中每一列的名称。输出时，第一行包含表中的列名；输入时，第一行会被忽略。只有使用 CSV 格式时才允许此选项。

QUOTE

指定在对数据值加引号时使用的引用字符。默认是双引号。这必须是一个单一的单字节字符。只有使用 CSV 格式时才允许这个选项。

ESCAPE

指定在与 QUOTE 值匹配的数据字符之前应出现的字符。默认值与 QUOTE 值相同（这样当引用字符出现在数据中时，就会被双写）。这必须是一个单一的单字节字符。只有使用CSV 格式时才允许这个选项。

FORCE_QUOTE

强制对每个指定列中的所有非 NULL 值使用引号。NULL 输出永远不会加引号。如果指定了*，则所有列中的非NULL 值都会加引号。此选项仅允许用于 COPY TO，且只能在使用CSV 格式时使用。

FORCE_NOT_NULL

不要将指定列的值与空值串进行匹配。在空值串就是空串的默认情况下，这意味着空串将被读作长度为零的字符串而不是空值（即使它们没有被引用）。只有在COPY FROM 中使用 CSV 格式时才允许这个选项。

FORCE_NULL

将指定列的值与空值串匹配，即使它已经被加上引号；如果找到匹配，就将该值设为 NULL。在空值串就是空串的默认情况下，这会把一个带引号的空串转换为 NULL。只有在 COPY FROM 中使用 CSV 格式时才允许这个选项。

ENCODING

指定文件采用 *encoding_name* 编码。如果省略此选项，将使用当前客户端编码。详见下文注解。

输出

成功完成时，COPY 命令会返回形如

```
COPY count
```

的命令标签。*count* 为复制的行数。

> **注意**
>
> 只有当命令既不是 COPY ... TO STDOUT，也不是等效的 psql 元命令\copy ... to stdout 时，psql 才会打印这个命令标签。这是为了避免将命令标签与刚刚输出的数据混淆。

注解

COPY TO 只能用于普通表，不能用于视图，且不会复制子表或子分区中的行。例如，COPY *table* TO 复制的行与 SELECT * FROM ONLY *table* 相同。语法 COPY (SELECT * FROM *table*) TO ... 可用于导出继承层次结构、分区表或视图中的所有行。

COPY FROM 可用于普通表、外部表、分区表，以及具有 INSTEAD OF INSERT 触发器的视图。

你必须对 COPY TO 读取其值的表具有 SELECT 权限，并对COPY FROM 插入其值的表具有 INSERT 权限。对于命令中列出的列，具有列级权限即可。

如果对表启用了行级安全，相关的SELECT 策略将应用于 COPY *table* TO 语句。目前，对启用了行级安全的表不支持COPY FROM。请改用等效的 INSERT 语句。

COPY 命令中指定的文件由服务器而非客户端应用直接读取或写入。因此，这些文件必须位于数据库服务器所在机器上，或者可由数据库服务器访问，而不是仅由客户端访问。它们必须可由PostgreSQL 用户（服务器运行时使用的用户 ID）访问，并且对该用户可读或可写。同样，使用PROGRAM 指定的命令也是由服务器而非客户端应用直接执行，因而必须可由 PostgreSQL 用户执行。只有数据库超级用户，或被授予默认角色 pg_read_server_files、pg_write_server_files 或 pg_execute_server_program 之一的用户，才允许使用指定文件名或命令的COPY，因为这允许读取或写入服务器有权访问的任意文件，或者运行服务器有权执行的程序。

不要将 COPY 与 psql 指令 \copy 混淆。\copy 会调用 COPY FROM STDIN 或 COPY TO STDOUT，然后在psql 客户端可访问的文件中读取或存储数据。因此，使用\copy 时，文件的可访问性和访问权限取决于客户端而不是服务器。

建议在 COPY 中使用的文件名始终指定为绝对路径。对于 COPY TO，服务器会强制这一点；但对于 COPY FROM，你仍可选择从使用相对路径指定的文件中读取。该路径将相对于服务器进程的工作目录（通常是集簇的数据目录）而非客户端的工作目录进行解释。

使用 PROGRAM 执行命令可能会受到操作系统的访问控制机制（如 SELinux）的限制。

COPY FROM 将调用目标表上的任何触发器和检查约束。但是它不会调用规则。

对于标识列，COPY FROM 命令总会写入输入数据中提供的列值，其行为类似于INSERT 的 OVERRIDING SYSTEM VALUE 选项。

COPY 的输入和输出会受 DateStyle 影响。为确保数据能移植到其他可能使用非默认 DateStyle 设置的 PostgreSQL 安装中，使用COPY TO 前应将 DateStyle 设置为 ISO。同样也建议避免在 IntervalStyle 设置为 sql_standard 时转储数据，因为负的 interval 值可能会被采用不同 IntervalStyle 设置的服务器误解。

即使数据会被服务器直接从一个文件读取或者写入一个文件而不通过客户端，输入数据也会被根据ENCODING 选项或者当前客户端编码解释，并且输出数据会被根据ENCODING 或者当前客户端编码进行编码。

COPY 会在遇到第一个错误时停止操作。对于 COPY TO，这应当不会导致问题；但对于 COPY FROM，目标表此时已经接收了前面的行。这些行不可见也不可访问，但仍占用磁盘空间。如果在一次大型复制操作已经进行很久后才失败，可能会浪费大量磁盘空间。可以调用 VACUUM 回收这些浪费的空间。

FORCE_NULL 和 FORCE_NOT_NULL 可以同时用于同一列。这会把带引号的空值串转换为空值，并把不带引号的空值串转换为空串。

文件格式

文本格式

在使用 text 格式时，读取或写入的是一个文本文件，其中表中的每一行对应文件中的一行。每行中的列由分隔符字符隔开。列值本身是由各属性数据类型的输出函数生成或可被其输入函数接受的字符串。对于为空值的列，会使用指定的空值串代替。如果输入文件中的任何一行包含的列数多于或少于预期，COPY FROM 就会报错。如果指定了 OIDS，OID 会作为第一列读取或写入，位于用户数据列之前。

数据结束可以表示为只包含反斜线加点号（\.）的一行。从文件读取时，不需要数据结束标记，因为文件结束已足够；只有在使用 3.0 之前版本的客户端协议、向客户端应用程序复制数据或从其复制数据时，才需要该标记。

在 COPY 数据中，可以使用反斜线字符（\）来转义那些原本可能被当作行或列分隔符的数据字符。特别是，如果下列字符作为列值的一部分出现，那么它们前面必须加一个反斜线：反斜线本身、换行、回车以及当前分隔符字符。

COPY TO 输出指定的空值串时不会添加任何反斜线；相反，COPY FROM 会在去除反斜线之前先将输入与空值串进行匹配。因此，像\N 这样的空值串不会与实际的数据值\N 混淆，因为后者会表示为\\N。

COPY FROM 识别下列特殊的反斜线序列：

序列	表示
\b	退格 (ASCII 8)
\f	换页 (ASCII 12)
\n	新行 (ASCII 10)
\r	回车 (ASCII 13)
\t	制表 (ASCII 9)

序列	表示
\v	纵向制表 (ASCII 11)
digits	反斜线后跟一到三个八进制数字表示该数字代码对应的字节
\x*digits*	反斜线加x 后跟一到两个十六进制数字表示该数字代码对应的字节

目前，COPY TO 从不会输出八进制或十六进制数字反斜线序列，但对这些控制字符确实会使用上表列出的其他序列。

上表中未提到的其他字符，在前面加了反斜线后仍表示该字符本身。不过，要注意不要不必要地添加反斜线，因为那可能意外地产生与数据结束标记（\.）或空值串（默认是\N）匹配的字符串。这些字符串会在进行任何其他反斜线处理之前先被识别出来。

强烈建议生成 COPY 数据的应用将数据中的换行和回车分别转换为\n 和\r 序列。目前，仍然可以用反斜线加回车表示数据回车，用反斜线加换行表示数据换行。不过，未来版本可能不再接受这些表示方式。如果COPY 文件在不同机器之间传输（例如从 Unix 到 Windows，或反之），这些表示方式也非常容易被破坏。

所有反斜线序列都在编码转换后进行解释。用八进制和十六进制数字反斜线序列指定的字节必须在数据库编码中形成有效字符。

COPY TO 会用 Unix 风格的换行（“\n”）结束每一行。运行在 Microsoft Windows 上的服务器则会输出回车/换行（“\r\n”），但这只适用于复制到服务器文件的COPY；为保证跨平台一致性，COPY TO STDOUT 总是发送“\n”，与服务器平台无关。COPY FROM 能够处理以换行、回车或回车/换行结束的行。为减少本应是数据的未加反斜线新行或回车带来的风险，如果输入中的行结束符并不一致，COPY FROM 将会报错。

CSV 格式

这种格式选项用于导入和导出许多其他程序（如电子表格）使用的逗号分隔值（CSV）文件格式。不同于 PostgreSQL 标准文本格式使用的转义规则，它会生成并识别通用的 CSV 转义机制。

每条记录中的值由 DELIMITER 字符分隔。如果某个值包含分隔符字符、QUOTE 字符、NULL 字符串、回车或换行字符，那么整个值都会以前后各一个QUOTE 字符包围，并且该值内每次出现 QUOTE 字符或 ESCAPE 字符之前都会加上转义字符。对于指定列中的非 NULL 值输出，还可以使用FORCE_QUOTE 来强制加引号。

CSV 格式没有标准方式区分 NULL 值和空字符串。PostgreSQL 的 COPY 通过引号来处理这一区别。NULL 会按照 NULL 参数字符串输出，且不会被加引号；而与NULL 参数字符串匹配的非 NULL 值会被加引号。例如，在默认设置下，NULL 会写成一个未加引号的空字符串，而空字符串数据值会写成双引号包围的形式（""）。读取值时遵循类似规则。你可以使用 FORCE_NOT_NULL 来阻止对指定列进行 NULL 输入比较。也可以使用 FORCE_NULL 将带引号的空值串数据值转换为 NULL。

因为反斜线在 CSV 格式中不是特殊字符，数据结束标记 \. 也可能作为数据值出现。为避免误解，当 \. 数据值作为一行中的唯一字段出现时，输出时会自动给它加上引号；输入时，如果它带有引号，就不会被解释为数据结束标记。如果正在装载由其他应用程序创建的文件，其中只有一个不带引号的列，而且该列可能包含值 \.，那么可能需要在输入文件中给该值加上引号。

> **注意**
>
> 在 CSV 格式中，所有字符都有意义。被空白字符或 DELIMITER 之外其他字符包围的带引号值，会把这些字符也包含进值中。如果你导入的数据来自某个会用空白把CSV 行填充到固定宽度的系统，这可能导致错误。出现这种情况时，你可能需要在将数据导入 PostgreSQL 之前，先预处理CSV 文件以移除尾随空白。

> **注意**
>
> CSV 格式既能识别也能生成这样的 CSV 文件：其中带引号的值包含内嵌的回车和换行。因此，这类文件不像文本格式文件那样严格地一行对应表中的一行。

> **注意**
>
> 很多程序会生成奇怪、甚至近乎反常的 CSV 文件，因此这种文件格式更像一种约定而非标准。因而你可能会遇到无法用这种机制导入的文件，而COPY 也可能生成其他程序无法处理的文件。

二进制格式

binary 格式选项会使所有数据以二进制格式而不是文本格式存储或读取。它比文本和 CSV 格式稍快一些，但二进制格式文件在不同的机器架构和PostgreSQL 版本之间的可移植性较差。此外，二进制格式与数据类型高度相关。例如，不能从 smallint 列输出二进制数据再读入到 integer 列中，尽管这种做法在文本格式下是可行的。

binary 文件格式由文件头、零个或多个包含行数据的元组以及一个文件尾构成。头部和数据都以网络字节序表示。

> **注意**
>
> 7.4 之前的 PostgreSQL 版本使用一种不同的二进制文件格式。

文件头

文件头由 15 字节的固定字段构成，后面跟着一个变长的头部扩展区。固定字段有：

签名

11 字节序列 PGCOPY\n\377\r\n\0 – 注意，零字节是签名中必不可少的一部分。（该签名的设计目的是便于识别那些在不具备 8 位透明性的传输过程中遭到破坏的文件。行尾转换过滤器、零字节丢失、高位丢失或奇偶校验变化等情况都会改变该签名。）

标志字段

这是一个 32 位整数位掩码，用于表示文件格式的重要属性。位编号从 0（LSB）到 31（MSB）。注意，此字段与文件格式中的所有整数字段一样，采用网络字节序存储（最高

有效字节在前）。位 16-31 保留用于表示文件格式的关键问题；如果读取程序发现这个范围内有非预期的位被置位，就应中止。位 0-15 保留用于表示向后兼容的格式问题；读取程序应直接忽略这个范围内非预期的置位。目前仅定义了一个标志位，其余位必须为零：

位 16

如果为 1，则数据中包含 OID；如果为 0，则不包含。

头部扩展区长度

32 位整数，表示头部剩余部分的长度（以字节计），不包括该字段本身。当前该值为零，因此其后紧接着第一个元组。未来对这种格式的更改可能允许在头部中包含额外数据。如果读取程序不知道如何处理头部扩展区数据，应静默跳过它。

头部扩展区被设想为包含一系列可自我标识的块。标志字段并不用于告诉读取程序扩展区中包含哪些内容。头部扩展内容的具体设计留待后续版本决定。

这种设计既允许向后兼容的头部新增（增加头部扩展块，或设置低位标志位），也允许不向后兼容的更改（设置高位标志位来表明这类更改，并在需要时向扩展区增加支持数据）。

元组

每个元组都以一个 16 位整数计数开头，用于表示该元组中的字段数。（目前，一个表中的所有元组都应有相同的计数，但这未必永远如此。）随后，对元组中的每个字段，都会有一个 32 位长度字，后跟相应字节数的字段数据。（长度字不包括其本身，且可以为零。）特殊情况下，-1 表示一个 NULL 字段值；在 NULL 情况下，后面不会跟随任何值字节。

字段之间没有对齐填充或任何其他额外数据。

当前，二进制格式文件中的所有数据值都假定为二进制格式（格式代码一）。可以预见，未来的扩展可能会增加一个允许为各列分别指定格式代码的头部字段。

要确定实际元组数据应采用的二进制格式，你应该参考 PostgreSQL 源码，特别是各列数据类型对应的`*send`和`*recv`函数（这些函数通常可以在源码分发包的`src/backend/utils/adt/`目录中找到）。

如果文件中包含 OID，则 OID 字段紧跟在字段计数值之后。它是一个普通字段，只是不计入字段数。特别是，它带有一个长度字段 – 这样就能较容易地处理 4 字节或 8 字节的 OID，也允许在将来确有需要时把 OID 表示为空值。

文件尾

文件尾由一个值为 -1 的 16 位整数构成。这很容易与元组的字段计数字区分开来。

如果字段计数字既不是 -1 也不是预期的列数，读取程序应报告错误。这提供了一项额外检查，以防与数据失去同步。

示例

下面的示例使用竖线（|）作为字段分隔符将一个表复制到客户端：

```
COPY country TO STDOUT (DELIMITER '|');
```

要将文件中的数据复制到 country 表中：

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

只把名称以 'A' 开头的国家复制到一个文件中：

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/a_list_
countries.copy';
```

要复制到压缩文件中，可以将输出通过管道送入外部压缩程序：

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

下面给出适合从 STDIN 复制到表中的示例数据：

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

注意每一行中的空白实际上是一个制表符。

下面是用二进制格式输出的相同数据。该数据是用 Unix 工具 od -c 过滤后显示的。该表具有三列，第一列类型是char(2)，第二列类型是text，第三列类型是integer。所有行在第三列都是空值。

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

兼容性

SQL 标准中没有 COPY 语句。

下列语法在 PostgreSQL 9.0 之前的版本中使用，现仍受支持：

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter_character' ]
          [ NULL [ AS ] 'null string' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'quote_character' ]
                [ ESCAPE [ AS ] 'escape_character' ]
                [ FORCE NOT NULL column_name [, ...] ] ] ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter_character' ]
          [ NULL [ AS ] 'null string' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'quote_character' ]
                [ ESCAPE [ AS ] 'escape_character' ]
                [ FORCE QUOTE { column_name [, ...] | * } ] ] ]
```

注意在这种语法中，BINARY 和 CSV 被视为独立的关键字，而不是FORMAT 选项的参数。

下列语法在 PostgreSQL 7.3 之前的版本中使用，现仍受支持：

```
COPY [ BINARY ] table_name [ WITH OIDS ]
    FROM { 'filename' | STDIN }
    [ [USING] DELIMITERS 'delimiter_character' ]
    [ WITH NULL AS 'null_string' ]

COPY [ BINARY ] table_name [ WITH OIDS ]
    TO { 'filename' | STDOUT }
    [ [USING] DELIMITERS 'delimiter_character' ]
    [ WITH NULL AS 'null_string' ]
```

CREATE ACCESS METHOD

CREATE ACCESS METHOD — 定义一种新的访问方法

大纲

```
CREATE ACCESS METHOD name
    TYPE access_method_type
    HANDLER handler_function
```

描述

`CREATE ACCESS METHOD` 创建一种新的访问方法。

访问方法的名称在数据库中必须唯一。

只有超级用户可以定义新的访问方法。

参数

name

要创建的访问方法的名称。

access_method_type

此子句指定要定义的访问方法类型。目前仅支持 `INDEX`。

handler_function

handler_function 是一个先前注册的函数的名称（可能带有模式限定），该函数表示访问方法。处理器函数必须声明为接受一个 `internal` 类型的单一参数，其返回类型取决于访问方法的类型；对于 `INDEX` 访问方法，它必须是 `index_am_handler`。处理器函数必须实现的 C 级 API 会根据访问方法的类型而有所不同。索引访问方法 API 的描述见第 61 章。

示例

创建索引访问方法 `heptree`，其处理器函数为`heptree_handler`：

```
CREATE ACCESS METHOD heptree TYPE INDEX HANDLER heptree_handler;
```

兼容性

`CREATE ACCESS METHOD` 是一种 PostgreSQL 扩展。

另见

DROP ACCESS METHOD, CREATE OPERATOR CLASS, CREATE OPERATOR FAMILY

CREATE AGGREGATE

CREATE AGGREGATE — 定义一个新的聚合函数

大纲

```
CREATE AGGREGATE name ( [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)

CREATE AGGREGATE name ( [ [ argmode ] [ argname ] arg_data_type [ , ... ] ]
                        ORDER BY [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , INITCOND = initial_condition ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
    [ , HYPOTHETICAL ]
)
```

或旧式语法

```
CREATE AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
```

```
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
)
```

描述

CREATE AGGREGATE 定义一个新的聚合函数。系统自带了一些基本且常用的聚合函数；它们在第 9.20 节中有文档说明。如果需要定义新类型或需要一个尚未提供的聚合函数，则可以使用 CREATE AGGREGATE 来实现所需功能。

如果给出了一个模式名（例如 CREATE AGGREGATE myschema.myagg ...），则该聚合函数会在指定模式中创建。否则会在当前模式中创建。

聚合函数由其名称和输入数据类型来标识。如果同一模式中的两个聚合作用于不同的输入类型，它们可以具有相同的名称。聚合的名称和输入数据类型还必须不同于同一模式中每个普通函数的名称和输入数据类型。这种行为与普通函数名的重载完全相同（见 CREATE FUNCTION）。

简单聚合函数由一个或两个普通函数构成：一个状态转移函数 *sfunc*，以及一个可选的最终计算函数 *ffunc*。其用法如下：

```
sfunc( internal-state, next-data-values ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
```

PostgreSQL 会创建一个数据类型为 *stype* 的临时变量，用来保存聚合的当前内部状态。对于每个输入行，都会先计算聚合参数值，然后以当前状态值和新的参数值调用状态转移函数，从而计算新的内部状态值。等到所有行都处理完毕后，再调用一次最终函数来计算聚合的返回值。如果没有最终函数，则原样返回结束时的状态值。

聚合函数可以提供一个初始条件，即内部状态值的初始值。它在数据库中以 text 类型的值指定和存储，但它必须是该状态值数据类型常量的合法外部表示。如果未提供，则状态值初始为空值。

如果状态转移函数被声明为“strict”，则不能用空输入调用它。采用这种转移函数时，聚合执行的行为如下。任何输入值为空的行都会被忽略（不会调用该函数，并保留先前的状态值）。如果初始状态值为空，则在第一个所有输入值都非空的行上，第一个参数值会取代状态值，并在之后每一个所有输入值都非空的行上调用状态转移函数。这对于实现max 之类的聚合很方便。注意，只有当 *state_data_type* 与第一个 *arg_data_type* 相同时，这种行为才可用。当这两种类型不同时，必须提供非空初始条件，或者使用非 strict 状态转移函数。

如果状态转移函数不是 strict，则它会无条件地在每个输入行上被调用，并且必须自行处理空输入和空状态值。这使聚合作者能够完全控制聚合对空值的处理方式。

如果最终函数被声明为“strict”，那么当结束状态值为空时不会调用它；而是自动返回空结果。（这当然只是 strict 函数的正常行为。）无论如何，最终函数都可以选择返回空值。例如，avg 的最终函数在发现输入行数为零时会返回空值。

有时将最终函数声明为不仅接收状态值，还接收与聚合输入值相对应的额外参数会很有用。这样做的主要原因是，如果最终函数是多态的，状态值的数据类型不足以确定结果类型。这些额外参数总是以 NULL 传递（因此在使用 FINALFUNC_EXTRA 选项时，最终函数不能声明为 strict），但它们仍然是有效参数。例如，最终函数可以利用 get_fn_expr_argtype 来识别当前调用中的实际参数类型。

聚合可以选择支持第 38.11.1 节中所述的移动聚合模式。这要求指定 MSFUNC、MINVFUNC 和 MSTYPE 参数，并可选指定 MSSPACE、MFINALFUNC、MFINALFUNC_EXTRA、MFINALFUNC_MODIFY 和 MINITCOND 参数。除 MINVFUNC 外，这些参数的工作方式与对应的不带M 的简单聚合参数相同；它们定义了该聚合的一种包含逆向状态转移函数的独立实现。

参数列表中带有 ORDER BY 的语法会创建一种称为有序集聚合的特殊聚合类型；如果指定了 HYPOTHETICAL，则会创建假想集聚合。这些聚合以依赖顺序的方式对一组已排序的值进行操作，因此指定输入排序顺序是调用中必不可少的一部分。它们还可以有直接参数，即每次聚合只求值一次而不是对每个输入行求值一次的参数。假想集聚合是有序集聚合的一个子类，其中要求某些直接参数在数量和数据类型上与被聚合的参数列匹配。这使得这些直接参数的值可以作为一条额外的“假想”行添加到聚合输入行的集合中。

聚合还可以选择支持第 38.11.4 节中所述的部分聚合。这要求指定 COMBINEFUNC 参数。如果 *state_data_type* 是 internal，通常还应提供SERIALFUNC 和 DESERIALFUNC 参数，以便能够进行并行聚合。注意，要启用并行聚合，该聚合还必须被标记为PARALLEL SAFE。

行为类似于 MIN 或 MAX 的聚合，有时可以通过查阅索引而不是扫描每个输入行来优化。如果该聚合可以这样优化，请通过指定一个排序操作符来表明。基本要求是，该聚合必须返回该操作符所诱导的排序顺序中的第一个元素；换句话说：

```
SELECT agg(col) FROM tab;
```

必须等价于：

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

进一步的假设是，该聚合忽略空输入，并且当且仅当不存在非空输入时返回空结果。通常，某种数据类型的<操作符是 MIN 的合适排序操作符，而> 是 MAX 的合适排序操作符。注意，除非指定的操作符是 B-树索引操作符类中“小于”或“大于”策略成员，否则这种优化实际上永远不会生效。

要能够创建聚合函数，你必须在参数类型、状态类型和返回类型上拥有 USAGE 权限，并在支持函数上拥有 EXECUTE 权限。

参数

name

要创建的聚合函数的名称（可带模式限定）。

argmode

参数的模式：IN 或 VARIADIC。（聚合函数不支持 OUT 参数。）如果省略，默认为 IN。只有最后一个参数可以标记为 VARIADIC。

argname

参数的名称。目前这仅对文档有用。如果省略，则该参数没有名称。

arg_data_type

该聚合函数所操作的输入数据类型。要创建零参数聚合函数，在参数说明列表的位置写*。（此类聚合的一个示例是 count(*)。）

base_type

在 CREATE AGGREGATE 的旧语法中，输入数据类型通过 basetype 参数指定，而不是写在聚合名称旁边。注意，这种语法只允许一个输入参数。要用这种语法定义零参数聚合函数，应将 basetype 指定为"ANY"（不是 *）。有序集聚合不能用旧语法定义。

sfunc

对每个输入行调用的状态转移函数名称。对于普通的 *N* 参数聚合函数，*sfunc* 必须接收 *N*+1 个参数，第一个参数类型为*state_data_type*，其余参数与该聚合声明的输入数据类型匹配。该函数必须返回 *state_data_type* 类型的值。它接收当前状态值和当前输入数据值，并返回下一个状态值。

对于有序集（包括假想集）聚合，状态转移函数只接收当前状态值和聚合参数，不接收直接参数。除此之外它与普通情况相同。

state_data_type

聚合状态值的数据类型。

state_data_size

聚合状态值的大致平均大小（以字节为单位）。如果省略此参数或其值为零，则会基于 *state_data_type* 使用默认估计值。规划器使用此值来估算分组聚合查询所需的内存。规划器仅在估计 hash 表可容纳于 work_mem 时才会考虑使用 hash 聚合；因此，此参数的较大值会抑制 hash 聚合的使用。

ffunc

在遍历完所有输入行后调用、用于计算聚合结果的最终函数名称。对于普通聚合，该函数必须只接收一个类型为*state_data_type* 的参数。聚合的返回数据类型定义为该函数的返回类型。如果未指定 *ffunc*，则结束状态值会被用作聚合结果，返回类型也就是 *state_data_type*。

对于有序集（包括假想集）聚合，最终函数不仅接收最终状态值，还会接收所有直接参数的值。

如果指定了 FINALFUNC_EXTRA，则除了最终状态值和任何直接参数之外，最终函数还会接收额外的 NULL 值，它们对应于该聚合的常规（被聚合的）参数。这主要用于在定义多态聚合时能够正确解析聚合的结果类型。

FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }

此选项指定最终函数是否为不修改其参数的纯函数。READ_ONLY 表示不会修改；另外两个值表示它可能会更改状态值。更多细节见下文注解。默认值是 READ_ONLY，但对于有序集聚合，默认值是 READ_WRITE。

combinefunc

可以选择指定 *combinefunc* 函数，以使聚合函数支持部分聚合。如果提供了它，则 *combinefunc* 必须把两个 *state_data_type* 值合并起来；这两个值各自包含对某个输入值子集聚合得到的结果，并产生一个新的 *state_data_type*，表示同时对这两组输入进行聚合的结果。可以把这个函数看作一种 *sfunc*：不同之处在于，它不是处理单个输入行并把它加入当前聚合状态，而是把另一个聚合状态加入当前状态。

combinefunc 必须声明为接收两个 *state_data_type* 参数，并返回一个 *state_data_type* 值。该函数也可以选择声明为“strict”。在这种情况下，只要任一输入状态为空，就不会调用该函数；另一个状态会被视为正确结果。

对于 *state_data_type* 为 internal 的聚合函数，*combinefunc* 不能声明为 strict。在这种情况下，*combinefunc* 必须确保正确处理空状态，并且返回的状态被正确存储在聚合内存上下文中。

serialfunc

state_data_type 为 internal 的聚合函数，只有在具备一个 *serialfunc* 函数时才能参与并行聚合；该函数必须把聚合状态序列化为一个 bytea 值，以传输给另一个进程。该函数必须接收一个internal 类型参数并返回 bytea 类型结果。还需要相应的 *deserialfunc*。

deserialfunc

将先前序列化的聚合状态反序列化回 *state_data_type*。该函数必须接收 bytea 和 internal 两个参数，并产生一个internal 类型的结果。（注意：第二个 internal 参数未被使用，但出于类型安全的原因必须存在。）

initial_condition

状态值的初始设置。这必须是一个字符串常量，其形式必须能被数据类型 *state_data_type* 接受。如果未指定，状态值初始为空值。

msfunc

在移动聚合模式下，对每个输入行调用的前向状态转移函数名称。它与常规转移函数完全相同，只是其第一个参数和结果都是 *mstate_data_type* 类型，这可能与 *state_data_type* 不同。

minvfunc

在移动聚合模式中使用的逆向状态转移函数名称。该函数的参数和结果类型与 *msfunc* 相同，但它不是把值添加到当前聚合状态中，而是从中移除一个值。逆向状态转移函数必须与前向状态转移函数具有相同的 strict 属性。

mstate_data_type

使用移动聚合模式时，聚合状态值的数据类型。

mstate_data_size

使用移动聚合模式时，聚合状态值的近似平均大小（以字节为单位）。其作用与 *state_data_size* 相同。

mffunc

在使用移动聚合模式时，在遍历完所有输入行后调用、用于计算聚合结果的最终函数名称。其工作方式与 *ffunc* 相同，只是其第一个参数的类型是*mstate_data_type*，而额外的哑参数通过写MFINALFUNC_EXTRA 来指定。由 *mffunc* 或 *mstate_data_type* 决定的聚合结果类型，必须与该聚合常规实现决定的结果类型一致。

MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }

此选项类似于 FINALFUNC_MODIFY，只是它描述了移动聚合最终函数的行为。

minitial_condition

使用移动聚合模式时，状态值的初始设置。其作用与 *initial_condition* 相同。

sort_operator

与类 MIN 或类 MAX 聚合关联的排序操作符。这只是一个操作符名称（可带模式限定）。假定该操作符与该聚合具有相同的输入数据类型（该聚合必须是单参数普通聚合）。

PARALLEL = { SAFE | RESTRICTED | UNSAFE }

PARALLEL SAFE、PARALLEL RESTRICTED 和 PARALLEL UNSAFE 的含义与 CREATE FUNCTION 中的相同。如果聚合函数被标记为 PARALLEL UNSAFE（这是默认值！）或 PARALLEL RESTRICTED，则不会考虑对其进行并行化处理。请注意，规划器不会检查聚合函数支持函数的并行安全性标记，只检查聚合函数本身的标记。

HYPOTHETICAL

仅用于有序集聚合。该标志指定应按假想集聚合的要求处理聚合参数：也就是，最后几个直接参数必须与聚合（WITHIN GROUP）参数的数据类型匹配。HYPOTHETICAL 标志不会影响运行时行为，只影响在解析时对聚合参数的数据类型和排序规则的确定。

CREATE AGGREGATE 的参数可以按任意顺序书写，而不只是按上面展示的顺序。

注解

在指定支持函数名的参数中，如果需要可以写模式名，例如 SFUNC = public.sum。但是不要在那里写参数类型 – 支持函数的参数类型由其他参数决定。

通常，PostgreSQL 函数应当是真正不修改其输入值的函数。然而，聚合的状态转移函数在聚合上下文中使用时，允许取巧地就地修改其状态值参数。与每次都新建一份状态值副本相比，这样可能带来显著的性能收益。

同样，虽然通常也期望聚合最终函数不修改其输入值，但有时无法避免修改状态值参数。这种行为必须通过FINALFUNC_MODIFY 参数声明。READ_WRITE 值表示最终函数会以未指定的方式修改状态值。这个值会阻止将该聚合用作窗口函数，也会阻止对共享相同输入值和状态转移函数的聚合调用合并状态值。SHAREABLE 值表示在调用最终函数之后，不能再应用状态转移函数，但可以对结束时的状态值执行多次最终函数调用。这个值同样会阻止将该聚合用作窗口函数，但允许合并状态值。（也就是说，这里关注的优化不是重复应用同一个最终函数，而是把不同的最终函数应用到同一个结束状态值上。只要这些最终函数都没有标记为READ_WRITE，这就是允许的。）

如果一个聚合支持移动聚合模式，那么当它被用作具有移动帧起点的窗口的窗口函数时（即帧起点模式不是UNBOUNDED PRECEDING），就能提高计算效率。从概念上讲，前向状态转移函数在输入值从底部进入窗口帧时把它们加入聚合状态，逆向状态转移函数则在这些值从顶部离开窗口帧时再次将其移除。因此，移除值时，总是按它们被加入时的相同顺序移除。每当调用逆向状态转移函数时，它收到的就是最早被加入但尚未被移除的参数值。逆向状态转移函数可以假定，在它移除最旧的那一行之后，当前状态中至少还会保留一行。（若非如此，窗口函数机制就会简单地重新开始一次全新的聚合，而不是使用逆向状态转移函数。）

用于移动聚合模式的前向状态转移函数不允许返回 NULL 作为新的状态值。如果逆向状态转移函数返回 NULL，就表示该逆向函数无法针对这个特定输入逆转状态计算，因此会从当前帧起始位置开始重新计算聚合。这一约定使得移动聚合模式可用于这样一些场景：在少数不常见的情况下，难以从当前状态值中逆向消除对应输入的影响。

如果没有提供移动聚合实现，聚合仍然可以与移动帧一起使用，但每当帧起点移动时，PostgreSQL 都会重新计算整个聚合。注意，无论聚合是否支持移动聚合模式，PostgreSQL 都能在不重新计算的情况下处理移动的帧结束位置；做法是继续把新值添加到聚合状态中。这就是为什么将聚合用作窗口函数时要求最终函数必须是只读的：它不能损坏聚合的状态值，这样即使已经针对一组帧边界得到了聚合结果值，聚合也能继续进行。

有序集聚合的语法允许对最后一个直接参数和最后一个聚合（`WITHIN GROUP`）参数都指定 `VARIADIC`。但是，当前实现从两个方面限制了 `VARIADIC` 的使用。第一，有序集聚合只能使用 `VARIADIC "any"`，不能使用其他可变参数数组类型。第二，如果最后一个直接参数是 `VARIADIC "any"`，则只能有一个聚合参数，并且它也必须是`VARIADIC "any"`。（在系统目录使用的表示中，这两个参数会合并为单个 `VARIADIC "any"` 项，因为`pg_proc` 无法表示具有多个 `VARIADIC` 参数的函数。）如果该聚合是假想集聚合，则与`VARIADIC "any"` 参数匹配的直接参数就是假想参数；其前面的任何参数都表示额外的直接参数，它们不受必须与聚合参数匹配的约束。

当前，有序集聚合无需支持移动聚合模式，因为它们不能被用作窗口函数。

部分（包括并行）聚合当前不支持有序集聚合。此外，对于包含 `DISTINCT` 或 `ORDER BY` 子句的聚合调用，也永远不会使用部分聚合，因为在部分聚合期间无法支持这些语义。

示例

见第 38.11 节。

兼容性

`CREATE AGGREGATE` 是 PostgreSQL 的语言扩展。SQL 标准不提供用户定义聚合函数。

另见

ALTER AGGREGATE, DROP AGGREGATE

CREATE CAST

CREATE CAST — 定义一种新的类型转换

大纲

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION function_name [ (argument_type [, ...]) ]
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

描述

`CREATE CAST` 定义一种新的类型转换。类型转换规定如何在两种数据类型之间执行转换。例如，

```
SELECT CAST(42 AS float8);
```

通过调用一个预先指定的函数（此处是 `float8(int4)`）把整型常量 42 转换成 `float8` 类型。（如果没有定义合适的类型转换，转换就会失败。）

两种类型可以是二进制可强制转换的，这意味着无需调用任何函数，就可以“免费”执行转换。这要求对应的值使用相同的内部表示。例如，`text` 和 `varchar` 这两种类型在两个方向上都是二进制可强制转换的。二进制可强制转换性不一定是对称关系。例如，在当前实现中，从`xml` 到 `text` 的类型转换可以免费执行，但反方向则需要一个至少执行语法检查的函数。（双向都二进制可强制转换的两种类型也称为二进制兼容。）

使用 `WITH INOUT` 语法，你可以把一种类型转换定义为基于 I/O 的类型转换。基于 I/O 的类型转换通过调用源数据类型的输出函数，并将得到的字符串传给目标数据类型的输入函数来执行。在许多常见情况下，这项特性避免了为转换单独编写类型转换函数的必要。基于 I/O 的类型转换与常规的基于函数的类型转换行为相同，只是实现方式不同。

默认情况下，只有显式请求类型转换时才会调用一种类型转换，也就是显式使用`CAST(`*`x`* `AS` *`typename`*`)` 或 *`x`*`::`*`typename`* 这种构造。

如果一种类型转换被标记为 AS ASSIGNMENT，那么在把值赋给目标数据类型的列时就可以隐式调用它。例如，假设foo.f1 是一个 text 类型的列，那么如果从integer 到 text 的类型转换被标记为 AS ASSIGNMENT，则：

```
INSERT INTO foo (f1) VALUES (42);
```

就会被允许，否则不会。（我们通常用术语赋值类型转换来描述这种类型转换。）

如果一种类型转换被标记为 AS IMPLICIT，那么无论是在赋值上下文中还是在表达式内部，都可以在任何上下文中隐式调用它。（我们通常用术语隐式类型转换来描述这种类型转换。）例如，考虑这个查询：

```
SELECT 2 + 4.0;
```

解析器最初分别把这两个常量标记为 integer 和 numeric 类型。系统目录中没有 integer + numeric 操作符，但有一个 numeric + numeric 操作符。因此，如果存在一种从integer 到 numeric 的可用类型转换，并且被标记为AS IMPLICIT — 实际上确实如此 — 该查询就会成功。解析器将应用该隐式类型转换，并把该查询解析为如同它被写成了：

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

现在，系统目录还提供了一种从numeric 到 integer 的类型转换。如果该类型转换被标记为 AS IMPLICIT — 实际上并没有 — 那么解析器就必须在上面的解释方式和另一种方案之间作出选择：把 numeric 常量转换成 integer，然后应用 integer + integer 操作符。由于它不知道该偏向哪一种选择，就会放弃并将该查询判定为有歧义。两种类型转换中只有一种是隐式的，正是借此我们让解析器倾向于把混合了 numeric 和 integer 的表达式解析为 numeric；系统对此并没有内置知识。

把类型转换标记为隐式时应当保持保守。过多的隐式类型转换路径可能导致 PostgreSQL 对命令作出令人意外的解释，或者因为存在多种可能的解释而根本无法解析命令。一个好的经验法则是，只有对同一一般类型分类中且能保留信息的类型间转换，才让它可以被隐式调用。例如，从 int2 到 int4 的类型转换可以合理地设为隐式，但从float8 到 int4 的类型转换大概应仅限赋值使用。跨类型分类的类型转换，例如从text 到 int4，最好只允许显式调用。

> **注意**
>
> 有时出于可用性或标准兼容性的原因，有必要在一组类型之间提供多种隐式类型转换，这会带来无法用上述方式避免的歧义。解析器有一种基于类型分类和首选类型的后备启发式规则，在这种情况下有助于提供期望的行为。详见 CREATE TYPE。

要能够创建一种类型转换，你必须拥有源数据类型或目标数据类型之一，并且对另一种类型具有 USAGE 权限。要创建一种二进制强制转换，你必须是超级用户。（之所以有这一限制，是因为错误的二进制强制转换很容易使服务器崩溃。）

参数

source_type

该类型转换的源数据类型的名称。

target_type

该类型转换的目标数据类型的名称。

function_name[(*argument_type* [, ...])]

用于执行该类型转换的函数。函数名可以用模式限定；如果没有，则会在模式搜索路径中查找该函数。该函数的结果数据类型必须与类型转换的目标类型一致。其参数见下文。如果没有指定参数列表，则该函数名在其模式中必须是唯一的。

WITHOUT FUNCTION

表示源类型对目标类型是二进制可强制转换的，因此执行该类型转换不需要函数。

WITH INOUT

表示该类型转换是一种基于 I/O 的类型转换，其执行方式是调用源数据类型的输出函数，并将得到的字符串传给目标数据类型的输入函数。

AS ASSIGNMENT

表示该类型转换可以在赋值上下文中隐式调用。

AS IMPLICIT

表示该类型转换可以在任何上下文中隐式调用。

类型转换实现函数可以有一到三个参数。第一个参数类型必须与源类型相同，或者可以由源类型进行二进制强制转换得到。第二个参数（如果有）必须是 `integer` 类型；它接收与目标类型关联的类型修饰符，如果没有则为`-1`。第三个参数（如果有）必须是 `boolean` 类型；如果该类型转换是显式类型转换，它接收`true`，否则接收`false`。（奇怪的是，SQL 标准在某些情况下要求显式类型转换和隐式类型转换具有不同的行为。这个参数是为必须实现这类类型转换的函数提供的。不建议你把自己的数据类型设计成需要关心这一点。）

类型转换函数的返回类型必须与类型转换的目标类型相同，或者对该目标类型是二进制可强制转换的。

通常，类型转换的源数据类型和目标数据类型必须不同。不过，如果它具有一个接受多个参数的类型转换实现函数，则允许声明源类型和目标类型相同的类型转换。这用于在系统目录中表示特定类型的长度强制函数。所命名的函数用于将该类型的值强制为其第二个参数给定的类型修饰符值。

当一种类型转换的源类型和目标类型不同，且其函数接受多个参数时，它支持在一个步骤中同时完成从一种类型到另一种类型的转换并应用长度强制。如果没有这样的条目，对使用类型修饰符的类型进行强制就需要两个类型转换步骤：先在数据类型之间进行转换，再应用该修饰符。

当前，到域类型或从域类型的类型转换都没有效果。到域或从域的类型转换都会使用与其底层类型关联的类型转换。

注解

使用 DROP CAST 删除用户定义的转换。

请记住，如果你希望能够双向转换类型，就需要在两个方向上分别显式声明类型转换。

通常没有必要在用户定义类型与标准字符串类型（text、varchar 和 char(*n*)，以及被定义为属于字符串分类的用户定义类型）之间创建类型转换。PostgreSQL 会自动为此提供基于 I/O 的类型转换。转换到字符串类型的自动类型转换被视为赋值类型转换，而从字符串类型出发的自动类型转换则只允许显式调用。你可以声明自己的类型转换来替代自动类型转换，从而覆盖这种行为，但通常这样做的唯一原因，是希望该转换比标准的仅赋值或仅显式设置更容易调用。另一种可能的原因是，你希望该转换的行为不同于该类型的 I/O 函数；但这已经足够反常，你应该三思这是不是一个好主意。（确实有少数内置类型在转换行为上有所不同，大多是由于 SQL 标准的要求。）

虽然这不是强制要求，但仍建议你继续遵循这种老惯例，即按目标数据类型为类型转换实现函数命名。许多用户已经习惯于用函数风格的记法进行类型转换，也就是*typename*(*x*)。这种记法实际上无非就是调用类型转换实现函数；它不会被特别当作类型转换处理。如果你的转换函数没有按这种惯例命名，那么用户会感到意外。由于 PostgreSQL 允许同一函数名按不同参数类型重载，因此让来自不同类型的多个转换函数都使用目标类型的名称并不存在困难。

> **注意**
>
> 实际上，前一段说得过于简单了：有两种情况下，即使一个函数调用形式没有匹配到实际存在的函数，也会被当作类型转换请求。如果函数调用 *name*(*x*)不能与任何现有函数精确匹配，但*name* 是一个数据类型名，并且 pg_cast 为从 *x* 的类型到该类型提供了二进制强制转换，那么该调用会被解释为二进制强制转换。作出这一例外，是为了让二进制强制转换即使没有任何函数，也能使用函数语法调用。同样，如果没有pg_cast 项，但该类型转换的目标或源是字符串类型，则该调用会被解释为基于 I/O 的类型转换。这一例外允许基于 I/O 的类型转换使用函数语法调用。

> **注意**
>
> 还有一个例外中的例外：从复合类型到字符串类型的基于 I/O 的类型转换不能使用函数语法调用，而必须写成显式类型转换语法（CAST 或::记法）。增加这一例外，是因为在引入自动提供的基于 I/O 的类型转换之后，如果原意是函数调用或列引用，就太容易意外地触发这种类型转换了。

示例

要使用函数 int4(bigint) 创建一种从类型 bigint 到类型 int4 的赋值类型转换：

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

（在系统中这种类型转换已经被预定义。）

兼容性

CREATE CAST 命令符合 SQL 标准，不过 SQL 没有对二进制可强制转换的类型或实现函数的额外参数作出规定。AS IMPLICIT 也是 PostgreSQL 的扩展。

另见

CREATE FUNCTION, CREATE TYPE, DROP CAST

CREATE COLLATION

CREATE COLLATION — 定义一种新排序规则

大纲

```
CREATE COLLATION [ IF NOT EXISTS ] name (
    [ LOCALE = locale, ]
    [ LC_COLLATE = lc_collate, ]
    [ LC_CTYPE = lc_ctype, ]
    [ PROVIDER = provider, ]
    [ VERSION = version ]
)
CREATE COLLATION [ IF NOT EXISTS ] name FROM existing_collation
```

描述

`CREATE COLLATION` 使用指定的操作系统区域设置，或者通过复制现有排序规则，定义一个新的排序规则。

要创建排序规则，你必须拥有目标模式上的 `CREATE` 权限。

参数

`IF NOT EXISTS`

如果同名排序规则已经存在，则不要抛出错误。在这种情况下会发出一条通知。请注意，不保证现有的排序规则与本应创建出的排序规则有任何相似之处。

`name`

排序规则的名称，可以使用模式限定。若未使用模式限定，则该排序规则会在当前模式中定义。排序规则名称在该模式内必须唯一。（系统目录中可以包含针对其他编码的同名排序规则，但如果数据库编码不匹配，则会忽略它们。）

`locale`

这是同时设置 `LC_COLLATE` 和 `LC_CTYPE` 的快捷方式。如果指定了此选项，则不能指定这两个参数中的任何一个。

`lc_collate`

为 `LC_COLLATE` 区域类别使用指定的操作系统区域设置。

`lc_ctype`

为 `LC_CTYPE` 区域类别使用指定的操作系统区域设置。

provider

指定用于与此排序规则关联的区域设置服务的提供程序。可能的值为：`icu`，`libc`。`libc` 是默认值。可用选项取决于操作系统和构建选项。

version

指定随该排序规则存储的版本字符串。通常应省略该选项，这样版本就会根据操作系统提供的该排序规则实际版本来计算。此选项旨在供 `pg_upgrade` 从现有安装中复制版本时使用。

另见 ALTER COLLATION，了解如何处理排序规则版本不匹配的问题。

existing_collation

要复制的现有排序规则的名称。新的排序规则将具有与现有排序规则相同的属性，但它是一个独立对象。

注解

使用 `DROP COLLATION` 可移除用户定义的排序规则。

关于如何创建排序规则的更多信息，见第 23.2.2.3 节。

使用 `libc` 排序规则提供者时，区域设置必须适用于当前数据库编码。有关精确规则，请参见 CREATE DATABASE。

示例

从操作系统区域设置 `fr_FR.utf8` 创建一种排序规则（假定当前数据库编码是 `UTF8`）：

```
CREATE COLLATION french (locale = 'fr_FR.utf8');
```

使用 ICU 提供程序并采用德语电话簿排序顺序创建一种排序规则：

```
CREATE COLLATION german_phonebook (provider = icu, locale = 'de-u-co-phonebk');
```

从现有排序规则创建排序规则：

```
CREATE COLLATION german FROM "de_DE";
```

这在应用程序中使用与操作系统无关的排序规则名称时非常方便。

兼容性

在 SQL 标准中有一个 `CREATE COLLATION` 语句，但它仅限于复制现有排序规则。创建新排序规则的语法是 PostgreSQL 扩展。

另见

ALTER COLLATION, DROP COLLATION

CREATE CONVERSION

CREATE CONVERSION — 定义一个新的编码转换

大纲

```
CREATE [ DEFAULT ] CONVERSION name
    FOR source_encoding TO dest_encoding FROM function_name
```

描述

`CREATE CONVERSION` 定义字符集编码之间的新转换。此外，被标记为 `DEFAULT` 的转换可用于客户端与服务器之间的自动编码转换。为此目的，必须定义两个转换：从编码 A 到 B 以及从编码 B 到 A。

要创建一个转换，你必须对该函数具有`EXECUTE` 权限，并在目标模式上具有`CREATE` 权限。

参数

`DEFAULT`

`DEFAULT` 子句表示该转换是这一特定源编码到目标编码的默认转换。对于某个编码对，在一个模式中应该只有一个默认转换。

`name`

转换的名称。转换名可以是模式限定的；如果不是，则该转换定义在当前模式中。转换名在一个模式中必须唯一。

`source_encoding`

源编码名称。

`dest_encoding`

目标编码名称。

`function_name`

用于执行该转换的函数。函数名可以是模式限定的；如果不是，则会在搜索路径中查找该函数。

该函数必须具有以下签名：

```
conv_proc(
    integer,  -- source encoding ID
    integer,  -- destination encoding ID
    cstring,  -- source string (null terminated C string)
    internal, -- destination (fill with a null terminated C string)
    integer   -- source string length
) RETURNS void;
```

注解

使用 DROP CONVERSION 可以删除用户定义的转换。

创建转换所要求的权限在未来版本中可能会更改。

示例

使用 myfunc 创建从编码 UTF8 到 LATIN1 的转换：

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

兼容性

CREATE CONVERSION 是 PostgreSQL 的一个扩展。SQL 标准中没有 CREATE CONVERSION 语句，但有一个在目的和语法上都非常相似的 CREATE TRANSLATION 语句。

另见

ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION

CREATE DATABASE

CREATE DATABASE — 创建一个新数据库

大纲

```
CREATE DATABASE name
    [ WITH ] [ OWNER [=] user_name ]
           [ TEMPLATE [=] template ]
           [ ENCODING [=] encoding ]
           [ LC_COLLATE [=] lc_collate ]
           [ LC_CTYPE [=] lc_ctype ]
           [ TABLESPACE [=] tablespace_name ]
           [ ALLOW_CONNECTIONS [=] allowconn ]
           [ CONNECTION LIMIT [=] connlimit ]
           [ IS_TEMPLATE [=] istemplate ]
```

描述

`CREATE DATABASE` 创建一个新的 PostgreSQL 数据库。

要创建数据库，你必须是超级用户，或者拥有特殊的 `CREATEDB` 权限。见 CREATE ROLE。

默认情况下，新数据库通过克隆标准系统数据库 `template1` 来创建。可以通过写成 `TEMPLATE` *name* 指定其他模板。特别是，写成 `TEMPLATE template0` 时，可以创建一个全新的数据库，它只包含你的 PostgreSQL 版本预定义的标准对象。如果你希望避免复制任何可能已添加到 `template1` 中的站点本地附加对象，这会很有用。

参数

name

要创建的数据库名称。

user_name

新数据库所有者的角色名称，或使用 `DEFAULT` 以使用默认值（即执行命令的用户）。要创建由其他角色拥有的数据库，你必须是该角色的直接或间接成员，或为超级用户。

template

用于创建新数据库的模板名称，或者指定 `DEFAULT` 以使用默认模板（`template1`）。

encoding

新数据库要使用的字符集编码。可指定字符串常量（例如 `'SQL_ASCII'`）、整数编码编号，或者指定 `DEFAULT` 以使用默认编码（即模板数据库的编码）。PostgreSQL 服务器支持的字符集见第 23.3.1 节。其他限制见下文。

lc_collate

新数据库中使用的排序规则（LC_COLLATE）。这会影响字符串的排序顺序，例如在带有 ORDER BY 的查询中，以及文本列索引中使用的顺序。默认值是使用模板数据库的排序规则。请参见下文的其他限制。

lc_ctype

新数据库中使用的字符分类（LC_CTYPE）。这会影响字符的分类，例如小写、大写和数字。默认值是使用模板数据库的字符分类。请参见下文的其他限制。

tablespace_name

将与新数据库关联的表空间名称，或者指定 DEFAULT 以使用模板数据库的表空间。该表空间将成为在此数据库中创建对象时使用的默认表空间。详见 CREATE TABLESPACE。

allowconn

如果为 false，则任何人都不能连接到该数据库。默认值为 true，即允许连接（但仍受其他机制限制，例如 GRANT/REVOKE CONNECT）。

connlimit

可对该数据库建立的并发连接数。-1（默认值）表示不受限制。

istemplate

如果为 true，则任何具有 CREATEDB 权限的用户都可以克隆该数据库；如果为 false（默认值），则只有超级用户或该数据库的拥有者可以克隆它。

可选参数可以按任意顺序书写，不一定要按照上面的顺序。

注解

CREATE DATABASE 不能在事务块内执行。

形如 “could not initialize database directory” 的错误，多半与数据目录权限不足、磁盘已满或其他文件系统问题有关。

使用 DROP DATABASE 删除数据库。

程序 createdb 是这个命令的一个包装器程序，为方便使用而提供。

数据库级配置参数（通过 ALTER DATABASE 设置）和数据库级权限（通过 GRANT 设置）不会从模板数据库复制。

尽管可以通过把某个数据库名指定为模板，从而复制它而不是复制 template1，但这（至少目前）并不打算作为一种通用的 “COPY DATABASE” 功能。主要限制是，在复制模板数据库期间，不能有任何其他会话连接到该数据库。CREATE DATABASE 在启动时如果发现存在任何其他连接，就会失败；否则，在 CREATE DATABASE 完成之前，将阻止对模板数据库建立新的连接。详见第 22.3 节。

为新数据库指定的字符集编码必须与所选区域设置（LC_COLLATE 和 LC_CTYPE）兼容。如果区域设置为 C（或等价的 POSIX），则允许所有编码；但对于其他区域设置，只有一种编码能够正常工作。（不过，在 Windows 上，UTF-8 编码可与任何区域设置一起使用。）

CREATE DATABASE 允许超级用户不考虑区域设置而指定 SQL_ASCII 编码，但这种选择已弃用；如果数据库中存储了与该区域设置不兼容编码的数据，字符串函数的行为可能会出错。

编码和区域设置必须与模板数据库的设置一致，除非使用 template0 作为模板。这是因为其他数据库可能包含与指定编码不匹配的数据，或者包含排序顺序会受 LC_COLLATE 和 LC_CTYPE 影响的索引。复制这样的数据会导致数据库在新设置下被视为损坏。不过，已知 template0 不包含任何会受此影响的数据或索引。

CONNECTION LIMIT 选项只是近似地被强制执行；如果两个新会话几乎同时启动，而该数据库只剩下一个连接“槽”，则两者都可能失败。此外，该限制对超级用户或后台工作进程无效。

示例

要创建一个新数据库：

```
CREATE DATABASE lusiadas;
```

要创建一个由用户 salesapp 拥有、默认表空间为 salesspace 的数据库 sales：

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

要创建一个使用不同区域设置的数据库 music：

```
CREATE DATABASE music
    LC_COLLATE 'sv_SE.utf8' LC_CTYPE 'sv_SE.utf8'
    TEMPLATE template0;
```

在这个示例中，如果指定的区域设置与 template1 中的不同，则必须使用 TEMPLATE template0 子句。（如果相同，那么显式指定区域设置就是多余的。）

要创建一个使用不同区域设置和不同字符集编码的数据库 music2：

```
CREATE DATABASE music2
    LC_COLLATE 'sv_SE.iso885915' LC_CTYPE 'sv_SE.iso885915'
    ENCODING LATIN9
    TEMPLATE template0;
```

指定的区域设置和编码设置必须匹配，否则会报错。

请注意，区域设置名称是特定于操作系统的，因此上述命令未必在所有环境中都以相同方式工作。

兼容性

SQL 标准中没有 CREATE DATABASE 语句。数据库相当于目录，而目录的创建由实现定义。

另见

ALTER DATABASE, DROP DATABASE

CREATE DOMAIN

CREATE DOMAIN — 定义一个新域

大纲

```
CREATE DOMAIN name [ AS ] data_type
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ constraint [ ... ] ]
```

其中 *constraint* 为:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

描述

`CREATE DOMAIN` 创建一个新域。域本质上是一种带有可选约束（即对允许值集合的限制）的数据类型。定义域的用户将成为其拥有者。

如果给定了模式名（例如 `CREATE DOMAIN myschema.mydomain ...`），则该域会在指定模式中创建。否则它会在当前模式中创建。域名在其所在模式中的现有类型和域之间必须唯一。

域适合把字段上的常见约束抽象到单一位置进行维护。例如，若有多个表都包含电子邮件地址列，并且都需要同一个 CHECK 约束来验证地址语法，那么定义一个域会比在每个表上分别设置该约束更合适。

要创建域，你必须对其底层类型拥有`USAGE` 权限。

参数

name

要创建的域名（可选地带模式限定）。

data_type

该域的底层数据类型。它可以包含数组说明符。

collation

该域的可选排序规则。如果未指定排序规则，则该域的排序规则行为与其底层数据类型相同。如果指定了`COLLATE`，则底层类型必须是一种支持排序规则的数据类型。

`DEFAULT` *expression*

`DEFAULT` 子句为该域数据类型的列指定默认值。该值可以是任意不含变量的表达式（但不允许子查询）。默认表达式的数据类型必须与该域的数据类型匹配。如果未指定默认值，则默认值为空值。

默认表达式会在任何未为该列指定值的插入操作中使用。如果为某个特定列定义了默认值，它就会覆盖与该域关联的任何默认值。反过来，域默认值又会覆盖与底层数据类型关联的任何默认值。

`CONSTRAINT` *`constraint_name`*

约束的可选名称。如果未指定，系统会生成一个名称。

`NOT NULL`

该域的值不允许为空值（但见下文注解）。

`NULL`

该域的值允许为空值。这是默认行为。

该子句仅用于与非标准 SQL 数据库兼容。不鼓励在新应用中使用它。

`CHECK (`*`expression`*`)`

`CHECK` 子句指定该域的值必须满足的完整性约束或测试。每个约束都必须是一个产生布尔结果的表达式。它应使用关键字 `VALUE` 来引用被测试的值。求值结果为 TRUE 或 UNKNOWN 的表达式会通过检查。如果表达式产生 FALSE 结果，就会报告错误，并且不允许将该值转换成该域类型。

当前，`CHECK` 表达式不能包含子查询，也不能引用 `VALUE` 之外的其他变量。

当一个域有多个 `CHECK` 约束时，会按名称的字母顺序测试它们。（9.5 之前的PostgreSQL 版本并不保证 `CHECK` 约束遵循任何特定的触发顺序。）

注解

域约束，特别是`NOT NULL`，会在把值转换成域类型时进行检查。即便存在这样的约束，一个名义上属于该域类型的列也仍可能读出为空值。例如，在外连接查询中，如果该域列位于外连接中可为空的一侧，就可能发生这种情况。一个更微妙的例子是：

```
INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false));
```

这个空标量子 SELECT 会产生一个空值，该空值被视为域类型的值，因此不会再对其执行进一步的约束检查，插入也会成功。

由于 SQL 普遍假定空值是每种数据类型的合法值，因此很难彻底避免这类问题。因此，最佳实践是把域约束设计为允许空值，然后在需要时对该域类型的列应用列级`NOT NULL` 约束，而不是直接对域类型应用这种约束。

PostgreSQL 假定 `CHECK` 约束的条件是不可变的，也就是说，对于相同的输入值，它们总会给出相同的结果。正是基于这一假设，系统只会在值首次被转换为域类型时检查 `CHECK` 约束，而不会在其他时候检查。（这与表 `CHECK` 约束的处理方式基本相同，如第 5.3.1 节所述。）

打破这一假设的一种常见方式是，在`CHECK` 表达式中引用用户定义函数，然后改变该函数的行为。PostgreSQL 不禁止这样做，但如果此时已有存储的域类型值违反了`CHECK` 约束，系统也不会注意到。这会导致后续数据库转储和恢复失败。推荐的处理方式是先删除该约束（使用 `ALTER DOMAIN`），调整函数定义，然后重新添加约束，从而重新根据已存储数据进行检查。

示例

这个示例创建 us_postal_code 数据类型，然后在一个表定义中使用该类型。这里使用正则表达式测试来验证该值看起来是否为一个合法的美国邮政编码：

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal us_postal_code NOT NULL
);
```

兼容性

命令 CREATE DOMAIN 符合 SQL 标准。

另见

ALTER DOMAIN, DROP DOMAIN

CREATE EVENT TRIGGER

CREATE EVENT TRIGGER — 定义一个新的事件触发器

大纲

```
CREATE EVENT TRIGGER name
    ON event
    [ WHEN filter_variable IN (filter_value [, ... ]) [ AND ... ] ]
    EXECUTE { FUNCTION | PROCEDURE } function_name()
```

描述

`CREATE EVENT TRIGGER` 创建一个新的事件触发器。每当指定的事件发生，并且与该触发器关联的`WHEN` 条件（如果有）得到满足时，就会执行该触发器函数。有关事件触发器的一般性介绍，见第 40 章。创建事件触发器的用户将成为其所有者。

参数

name

要赋予新触发器的名称。该名称在数据库内必须唯一。

event

触发对给定函数调用的事件名称。有关事件名称的更多信息，见第 40.1 节。

filter_variable

用于过滤事件的变量名称。这样可以将触发器限定为只在原本支持的部分情形下触发。当前唯一支持的 *filter_variable* 是 `TAG`。

filter_value

与相关 *filter_variable* 关联的一组值；当其取值属于该列表时，触发器就会被触发。对于 `TAG`，这表示一组命令标签（例如 `'DROP FUNCTION'`）。

function_name

由用户提供的函数，声明为不接受参数并返回类型 `event_trigger`。

在 `CREATE EVENT TRIGGER` 的语法中，关键字 `FUNCTION` 和 `PROCEDURE` 是等效的，但被引用的对象无论如何都必须是函数，而不是过程。这里使用关键字 `PROCEDURE` 属于历史遗留做法，现已弃用。

注解

只有超级用户能创建事件触发器。

事件触发器在单用户模式下被禁用（参见 postgres）。如果错误的事件触发器导致数据库严重失效，以至于无法删除触发器，可以重启到单用户模式，然后就能执行删除操作。

示例

禁止执行任何 DDL 命令：

```
CREATE OR REPLACE FUNCTION abort_any_command()
  RETURNS event_trigger
 LANGUAGE plpgsql
  AS $$
BEGIN
  RAISE EXCEPTION 'command % is disabled', tg_tag;
END;
$$;

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
   EXECUTE FUNCTION abort_any_command();
```

兼容性

在 SQL 标准中没有 `CREATE EVENT TRIGGER` 语句。

另见

ALTER EVENT TRIGGER, DROP EVENT TRIGGER, CREATE FUNCTION

CREATE EXTENSION

CREATE EXTENSION — 安装一个扩展

大纲

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
    [ WITH ] [ SCHEMA schema_name ]
             [ VERSION version ]
             [ FROM old_version ]
             [ CASCADE ]
```

描述

`CREATE EXTENSION` 将一个新扩展装载到当前数据库中。同名扩展不能已经被装载。

装载一个扩展本质上就是运行该扩展的脚本文件。该脚本通常会创建新的 SQL 对象，例如函数、数据类型、操作符以及索引支持方法。`CREATE EXTENSION` 还会记录所有已创建对象的标识，这样在发出`DROP EXTENSION` 时就可以将它们一并删除。

加载扩展需要与创建其组件对象所需的权限相同。对于大多数扩展，这意味着需要超级用户或数据库所有者权限。运行 `CREATE EXTENSION` 的用户将成为扩展的所有者，用于后续的权限检查，以及扩展脚本创建的任何对象的所有者。

参数

`IF NOT EXISTS`

如果已存在同名扩展，则不要抛出错误。这种情况下会发出一条提示。注意，这并不保证现有扩展与根据当前可用脚本文件本应创建出的扩展有任何相似之处。

`extension_name`

要安装的扩展名称。PostgreSQL 将根据 `SHAREDIR/extension/`*`extension_name`*`.control` 文件中的详细信息创建该扩展。

`schema_name`

如果该扩展允许其内容被重定位，则这是安装该扩展所含对象的模式名称。指定的模式必须已经存在。如果未指定，并且扩展控制文件中也未指定模式，则使用当前默认的对象创建模式。

如果扩展在其控制文件中指定了 `schema` 参数，就不能用 `SCHEMA` 子句覆盖该模式。通常，如果给出了 `SCHEMA` 子句，且它与扩展的 `schema` 参数冲突，就会报错。不过，如果同时给出 `CASCADE` 子句，则发生冲突时会忽略 *`schema_name`*。给定的 *`schema_name`* 将用于安装任何必需且其控制文件中未指定 `schema` 的扩展。

请记住，扩展本身不被视为属于任何模式：扩展的名称是非限定名，并且必须在整个数据库范围内唯一。但是，属于该扩展的对象可以位于模式中。

version

要安装的扩展版本。它可以写成标识符，也可以写成字符串常量。默认版本由扩展控制文件指定。

old_version

FROM *old_version* 必须在且仅在尝试安装一个替换“旧式”模块的扩展时指定，该模块仅是一组未打包为扩展的对象集合。此选项会导致 CREATE EXTENSION 运行一个替代安装脚本，将现有对象吸收进扩展中，而不是创建新对象。请确保 SCHEMA 指定的是包含这些现有对象的模式。

old_version 的值由扩展作者决定，如果存在多个可升级为扩展的旧式模块版本，该值可能会有所不同。对于PostgreSQL 9.1 之前版本提供的标准附加模块，将模块更新为扩展形式时应使用unpackaged 作为 *old_version* 的值。

CASCADE

自动安装该扩展所依赖、但尚未安装的任何扩展。它们的依赖也会以同样的方式递归自动安装。如果给出了 SCHEMA 子句，它将适用于所有以这种方式安装的扩展。该语句的其他选项不会应用于自动安装的扩展；特别是，这些扩展总是选择其默认版本。

注解

在使用 CREATE EXTENSION 将扩展装载到数据库之前，必须先安装好该扩展的支持文件。关于安装 PostgreSQL 随附扩展的信息，可见额外提供的模块。

当前可供装载的扩展可从系统视图 pg_available_extensions 或 pg_available_extension_versions 中查出。

> **小心**
>
> 以超级用户身份安装扩展，意味着必须相信扩展作者以安全的方式编写了扩展安装脚本。恶意用户要创建特洛伊木马对象并不算太难；这类对象可能会在之后执行编写粗心的扩展脚本时实施攻击，使该用户获得超级用户权限。不过，只有当特洛伊木马对象在脚本执行期间位于 search_path 中时，它们才会构成危险；这意味着它们位于扩展的安装目标模式中，或位于它所依赖的某个扩展的目标模式中。因此，处理那些脚本尚未经过仔细审查的扩展时，一个经验法则是：只把它们安装到从未向任何不受信任用户授予、而且今后也不会授予 CREATE 权限的模式中。它们所依赖的任何扩展也应如此。
>
> 一般认为，PostgreSQL 随附的扩展能够抵御这类安装时攻击，但少数依赖其他扩展的扩展除外。正如这些扩展的文档所述，它们应当安装到安全模式中，或者安装到与其所依赖扩展相同的模式中，或者同时满足这两点。

关于编写新扩展的信息，见第 38.16 节。

示例

将 hstore 扩展安装到当前数据库中，并把其对象放在 `addons` 模式中：

```
CREATE EXTENSION hstore SCHEMA addons;
```

实现同样效果的另一种方式是：

```
SET search_path = addons;
CREATE EXTENSION hstore;
```

将 PostgreSQL 9.1 之前版本中安装的 `hstore` 更新为扩展形式：

```
CREATE EXTENSION hstore SCHEMA public FROM unpackaged;
```

请注意指定安装现有 `hstore` 对象时所用的模式。

兼容性

`CREATE EXTENSION` 是 PostgreSQL 的扩展。

另见

ALTER EXTENSION, DROP EXTENSION

CREATE FOREIGN DATA WRAPPER

CREATE FOREIGN DATA WRAPPER — 定义一个新的外部数据包装器

大纲

```
CREATE FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

描述

`CREATE FOREIGN DATA WRAPPER` 创建一个新的外部数据包装器。定义该外部数据包装器的用户将成为它的拥有者。

外部数据包装器名称在数据库内必须唯一。

只有超级用户能够创建外部数据包装器。

参数

name

要创建的外部数据包装器的名称。

`HANDLER` *handler_function*

handler_function 是一个预先注册的函数的名称，该函数将被调用以取得外部表所需的执行函数。该处理器函数不能接受任何参数，并且其返回类型必须是 `fdw_handler`。

可以创建没有处理器函数的外部数据包装器，但使用这种包装器的外部表只能被声明，不能被访问。

`VALIDATOR` *validator_function*

validator_function 是一个预先注册的函数的名称，该函数将被调用以检查提供给该外部数据包装器的通用选项，以及用于外部服务器、用户映射和使用该外部数据包装器的外部表的选项。如果未指定验证器函数或者指定了 `NO VALIDATOR`，则在创建时不会检查这些选项。（外部数据包装器可能会在运行时忽略无效的选项声明，或者拒绝它们，这取决于其实现。）验证器函数必须接受两个参数：一个参数的类型为 `text[]`，其中包含系统目录中存储的选项数组；另一个参数的类型为`oid`，其值是包含这些选项的系统目录的 OID。返回类型会被忽略；该函数应当使用 `ereport(ERROR)` 报告无效选项。

`OPTIONS ( option 'value' [, ... ] )`

这个子句为新的外部数据包装器指定选项。允许的选项名和值都依赖于各个外部数据包装器，并通过该外部数据包装器的验证器函数进行验证。选项名必须唯一。

注解

PostgreSQL 的外部数据功能仍在积极开发中。查询优化能力还很初级，而且这项工作在很大程度上也留给了包装器本身。因此，未来在性能方面仍有相当大的改进空间。

示例

创建一个无用的外部数据包装器 `dummy`：

```
CREATE FOREIGN DATA WRAPPER dummy;
```

用处理器函数 `file_fdw_handler` 创建一个外部数据包装器 `file`：

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

用一些选项创建一个外部数据包装器 `mywrapper`：

```
CREATE FOREIGN DATA WRAPPER mywrapper
    OPTIONS (debug 'true');
```

兼容性

`CREATE FOREIGN DATA WRAPPER` 符合 ISO/IEC 9075-9 (SQL/MED)，但`HANDLER` 和 `VALIDATOR` 子句属于扩展，而标准中的 `LIBRARY` 和 `LANGUAGE` 子句在 PostgreSQL 中尚未实现。

不过需要注意，SQL/MED 功能整体上尚未完全符合该标准。

另见

ALTER FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE

CREATE FOREIGN TABLE

CREATE FOREIGN TABLE — 定义一个新外部表

大纲

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ] [ COLLATE collation ] [
 column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
  SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
  PARTITION OF parent_table [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ]
{ FOR VALUES partition_bound_spec | DEFAULT }
  SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]
```

其中 *column_constraint* 为:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr }
```

而 *table_constraint* 为:

```
[ CONSTRAINT constraint_name ]
CHECK ( expression ) [ NO INHERIT ]
```

而 *partition_bound_spec* 为:

```
IN ( { numeric_literal | string_literal | TRUE | FALSE | NULL } [, ...] ) |
FROM ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE } [, ...] )
  TO ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )
```

描述

CREATE FOREIGN TABLE 将在当前数据库中创建一个新的外部表。该表归发出该命令的用户所有。

如果给出了模式名（例如 CREATE FOREIGN TABLE myschema.mytable ...），则表将在指定模式中创建。否则，它将在当前模式中创建。外部表名必须与同一模式中任何其他外部表、表、序列、索引、视图或物化视图的名称不同。

CREATE FOREIGN TABLE 还会自动创建一种数据类型，用以表示与该外部表一行对应的复合类型。因此，外部表名不能与同一模式中任何已有数据类型同名。

如果指定了 PARTITION OF 子句，则该表会按指定边界创建为parent_table 的一个分区。

要创建外部表，必须拥有该外部服务器上的USAGE 权限，以及表中所用所有列类型上的USAGE 权限。

参数

IF NOT EXISTS

已经存在同名关系时不要抛出错误。这种情况下会发出一个提示。注意，已存在的关系不保证与原本将要创建的关系有任何相似之处。

table_name

要创建的表名（可选地带模式限定）。

column_name

要在新表中创建的列名。

data_type

该列的数据类型，可以包含数组说明符。有关PostgreSQL 支持的数据类型的更多信息，参见第 8 章。

COLLATE *collation*

COLLATE 子句为列指定排序规则（该列必须属于支持排序规则的数据类型）。如果未指定，则使用列数据类型的默认排序规则。

INHERITS (*parent_table* [, ...])

可选的 INHERITS 子句指定一个表列表，新创建的外部表将自动继承这些表的所有列。父表可以是普通表或外部表。有关更多详细信息，请参阅CREATE TABLE 的类似形式。

PARTITION OF *parent_table* { FOR VALUES *partition_bound_spec* | DEFAULT }

这种形式可用于将外部表创建为给定父表的一个分区，并为其指定分区边界值。有关更多详细信息，请参阅CREATE TABLE 的类似形式。请注意，如果父表上存在 UNIQUE 索引，则目前不允许将外部表创建为父表的分区。（另请参阅 ALTER TABLE ATTACH PARTITION。）

CONSTRAINT *constraint_name*

列约束或表约束的可选名称。如果约束被违反，错误消息中会包含该约束名，因此诸如 col must be positive 这样的约束名可以向客户端应用传达有用的约束信息。（若约束名中包含空格，则需要用双引号指定。）如果未指定约束名，系统会生成一个。

NOT NULL

该列不允许包含空值。

NULL

该列允许包含空值。这是默认情况。

该子句仅为兼容非标准 SQL 数据库而提供，不建议在新应用中使用。

`CHECK ( expression ) [ NO INHERIT ]`

`CHECK` 子句指定一个产生布尔结果的表达式，外部表中的每一行都应满足该表达式；也就是说，对于外部表中的所有行，该表达式都应产生 TRUE 或 UNKNOWN，而绝不能产生 FALSE。作为列约束指定的检查约束只应引用该列的值，而出现在表约束中的表达式可以引用多个列。

当前，`CHECK` 表达式不能包含子查询，也不能引用当前行的列之外的变量。可以引用系统列 `tableoid`，但不能引用其他系统列。

标记为 `NO INHERIT` 的约束不会传播到子表。

`DEFAULT default_expr`

`DEFAULT` 子句为包含它的列定义所对应的列指定默认数据值。该值可以是任何不含变量的表达式（不允许包含子查询或对当前表中其他列的交叉引用）。默认表达式的数据类型必须与列的数据类型匹配。

默认值表达式会用于任何未为该列指定值的插入操作。如果一列没有默认值，则默认值为 null。

server_name

用于该外部表的现有外部服务器的名称。有关定义服务器的细节，参见 CREATE SERVER。

`OPTIONS ( option 'value' [, ...] )`

要与新外部表或其某一列关联的选项。允许的选项名和值都取决于具体的外部数据包装器，并使用该外部数据包装器的验证器函数进行验证。不允许重复的选项名（不过表选项和列选项同名是允许的）。

注解

核心 PostgreSQL 系统不会强制执行外部表上的约束（例如 `CHECK` 或 `NOT NULL` 子句），而且大多数外部数据包装器也不会尝试强制执行它们；也就是说，这些约束只是被假定为真。因为这种强制执行只会适用于通过外部表插入或更新的行，而不会适用于通过其他方式修改的行，例如直接在远程服务器上修改的行，所以这样做意义不大。相反，附加到外部表上的约束应当表示由远程服务器强制执行的约束。

某些专用的外部数据包装器可能是其所访问数据的唯一访问机制，在这种情况下，由外部数据包装器自身执行约束检查也许是合适的。但除非其文档明确说明，否则不应假定某个包装器会这样做。

尽管 PostgreSQL 不会尝试强制执行外部表上的约束，但出于查询优化的目的，它会假定这些约束是正确的。如果外部表中存在不满足已声明约束的可见行，那么对该表的查询可能会产生错误或不正确的结果。确保约束定义符合实际情况是用户的责任。

> **小心**
>
> 当外部表被用作分区表的一个分区时，会有一个隐含约束，即其内容必须满足分区规则。同样，确保这一点是用户的责任，最好的做法是在远程服务器上安装匹配的约束。

在包含外部表分区的分区表中，如果外部数据包装器支持元组路由，那么更改分区键值的UPDATE 可能导致某一行从本地分区移动到外部表分区。然而，目前还不能将一行从外部表分区移动到另一个分区。需要这样做的UPDATE 会因为分区约束而失败，前提是假定远程服务器已正确强制执行该约束。

示例

创建通过服务器 film_server 访问的外部表 films：

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
)
SERVER film_server;
```

创建通过服务器 server_07 访问的外部表 measurement_y2016m07，并将其作为范围分区表 measurement 的一个分区：

```
CREATE FOREIGN TABLE measurement_y2016m07
    PARTITION OF measurement FOR VALUES FROM ('2016-07-01') TO ('2016-08-01')
    SERVER server_07;
```

兼容性

CREATE FOREIGN TABLE 命令基本符合 SQL 标准；但是，与CREATE TABLE 一样，它允许使用NULL 约束，也允许零列外部表。指定列默认值的能力也是PostgreSQL 的扩展。按 PostgreSQL 定义的形式，表继承是非标准的。

另见

ALTER FOREIGN TABLE, DROP FOREIGN TABLE, CREATE TABLE, CREATE SERVER, IMPORT FOREIGN SCHEMA

CREATE FUNCTION

CREATE FUNCTION — 定义一个新函数

大纲

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
```

描述

CREATE FUNCTION 定义一个新函数。CREATE OR REPLACE FUNCTION 将创建一个新函数，或者替换现有定义。要定义函数，用户必须具有该语言上的USAGE 权限。

如果包含模式名，那么该函数会被创建在指定的模式中。否则，它会被创建在当前模式中。新函数的名称不能匹配同一模式中任何具有相同输入参数类型的现有函数或过程。不过，不同参数类型的函数和过程能够共享一个名字（这被称为重载）。

要替换一个现有函数的当前定义，可以使用CREATE OR REPLACE FUNCTION。但不能用这种方式更改函数的名称或者参数类型（如果尝试这样做，实际上就会创建一个新的不同函数）。此外，CREATE OR REPLACE FUNCTION 也不允许更改现有函数的返回类型。要做到这一点，必须删除该函数并重新创建。（使用OUT 参数时，这意味着除非删除该函数，否则不能更改任何OUT 参数的类型。）

当 CREATE OR REPLACE FUNCTION 被用来替换一个现有函数时，该函数的拥有权和权限不会改变。所有其他的函数属性会按照该命令中指定的或者隐含的值赋值。必须拥有（包括成为拥有角色的成员）该函数才能替换它。

如果删除函数后再重新创建，新函数就不再是旧函数的同一实体；你将必须删除引用旧函数的现有规则、视图、触发器等。使用CREATE OR REPLACE FUNCTION 可以在不破坏引用该函数的对象的情况下更改函数定义。此外，ALTER FUNCTION 还可用于更改现有函数的大多数辅助属性。

创建该函数的用户将成为该函数的拥有者。

要创建一个函数，你必须拥有参数类型和返回类型上的USAGE 权限。

有关编写函数的详细信息，请参阅第 38.3 节。

参数

name

要创建的函数名称（可以被模式限定）。

argmode

参数的模式可以是：IN、OUT、INOUT 或者 VARIADIC。如果省略，则默认为IN。只有 OUT 参数可以跟在 VARIADIC 参数之后。此外，OUT 和 INOUT 参数不能与 RETURNS TABLE 记法一起使用。

argname

参数的名称。某些语言（包括 SQL 和 PL/pgSQL）允许在函数体中使用该名称。对于其他语言，就函数本身而言，输入参数的名称只是额外文档；但你可以在调用函数时使用输入参数名来提高可读性（见第 4.3 节）。无论如何，输出参数的名称很重要，因为它定义了结果行类型中的列名。（如果省略输出参数的名称，系统将选择一个默认列名。）

argtype

该函数参数（如果有）的数据类型（可以是模式限定的）。参数类型可以是基础类型、复合类型或者域类型，也可以引用一个表列的类型。

根据实现语言的不同，也可能允许指定诸如cstring 这样的“伪类型”。伪类型表示实际参数类型要么没有被完整指定，要么不属于普通 SQL 数据类型集合。

写成 *table_name*.*column_name*%TYPE 即可引用一个列的类型。使用这种特性有时有助于让函数独立于表定义的变化。

default_expr

如果未指定该参数，则用作默认值的表达式。该表达式必须能被强制转换为该参数的类型。只有输入参数（包括INOUT）才能有默认值。所有跟在具有默认值参数之后的输入参数也都必须有默认值。

rettype

该函数的返回数据类型（可以是模式限定的）。返回类型可以是基础类型、复合类型或者域类型，也可以引用一个表列的类型。根据实现语言的不同，也可能允许指定诸如cstring 这样的“伪类型”。如果函数不应该返回值，请把返回类型指定为void。

当存在 OUT 或 INOUT 参数时，可以省略RETURNS 子句。如果写出该子句，它必须与输出参数所隐含的结果类型一致：如果有多个输出参数，则为RECORD；如果只有一个输出参数，则为该输出参数的类型。

SETOF 修饰符表示该函数将返回一组项，而不是单个项。

写成 *table_name*.*column_name*%TYPE 即可引用一个列的类型。

column_name

RETURNS TABLE 语法中输出列的名称。这实际上是声明一个具名 OUT 参数的另一种方式，只不过RETURNS TABLE 还隐含了 RETURNS SETOF。

column_type

RETURNS TABLE 语法中的输出列的数据类型。

lang_name

实现该函数所用语言的名称。它可以是 sql、c、internal，也可以是用户定义的过程语言的名称，例如plpgsql。使用单引号括起名称已被弃用，并且要求大小写匹配。

TRANSFORM { FOR TYPE *type_name* } [, ...] }

列出调用该函数时应应用的转换。转换在 SQL 类型和语言相关的数据类型之间进行变换，详见 CREATE TRANSFORM。过程语言实现通常把有关内置类型的知识硬编码在代码中，因此那些不需要列举在这里。如果一种过程语言实现不知道如何处理某种类型且没有提供转换，它将回退到默认的数据类型转换行为，但这取决于具体实现。

WINDOW

WINDOW 表示该函数是窗口函数而不是普通函数。目前这只对用 C 编写的函数有用。在替换现有函数定义时，不能更改WINDOW 属性。

IMMUTABLE
STABLE
VOLATILE

这些属性会告诉查询优化器该函数的行为。最多只能指定其中一个。如果这些属性都没有出现，则默认假定为VOLATILE。

IMMUTABLE 表示该函数不能修改数据库，并且在给定相同参数值时总会返回相同结果；也就是说，它不会执行数据库查找，也不会以其他方式使用未直接出现在其参数列表中的信息。如果给出此选项，任何使用全常量参数对该函数的调用都可以立即替换为该函数值。

STABLE 表示该函数不能修改数据库，并且在一次表扫描内，对于相同参数值会一致地返回相同结果，但其结果可能在不同 SQL 语句之间发生变化。这适用于结果依赖于数据库查找、参数变量（例如当前时区）等的函数。（对于希望查询由当前命令修改过的行的AFTER 触发器，这样做并不合适。）另请注意，current_timestamp 函数族也属于稳定函数，因为它们的值在一个事务内不会变化。

VOLATILE 表示该函数的值即使在一次表扫描内也可能发生变化，因此无法进行任何优化。从这个意义上说，真正不稳定的数据库函数相对较少；一些例子是random()、currval()、timeofday()。但请注意，任何有副作用的函数都必须归类为不稳定，即使其结果相当可预测，也必须如此，以防其调用被优化掉；例如setval()。

更多细节见第 38.7 节。

LEAKPROOF

LEAKPROOF 表示该函数没有副作用。除返回值外，它不会泄露其参数的任何信息。例如，对某些参数值会抛出错误而对另一些不会，或者在错误消息中包含参数值的函数，都不是防泄漏的。这会影响系统如何执行针对使用security_barrier 选项创建的视图或启用了行级安全的表的查询。为了防止数据被无意暴露，系统会先强制执行安全策略和安全屏障视图中的条件，再执行查询本身中包含非防泄漏函数的用户提供条件。被标记为防泄漏的函数和操作符被视为可信，因此可以在安全策略和安全屏障视图的条件之前执行。此外，不接受参数的函数，或者没有从安全屏障视图或表中接收到任何参数的函数，即使未标记为防泄漏，也可以在安全条件之前执行。参见CREATE VIEW 和第 41.5 节。此选项只能由超级用户设置。

```
CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT
```

CALLED ON NULL INPUT（默认）表示当某些参数为空值时，仍会正常调用该函数。如果有需要，则由函数作者负责检查空值并作出适当响应。

RETURNS NULL ON NULL INPUT 或 STRICT 表示只要任一参数为空值，该函数总是返回空值。如果指定了这个选项，那么在参数中出现空值时不会执行该函数，而是自动假定结果为空值。

```
[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER
```

SECURITY INVOKER 表示该函数将以调用它的用户的权限执行。这是默认设置。
SECURITY DEFINER 指定该函数将以其所有者用户的权限执行。

为了符合 SQL，允许使用关键字EXTERNAL。但它是可选的，因为与 SQL 不同，这个特性适用于所有函数，而不仅仅是外部函数。

```
PARALLEL
```

PARALLEL UNSAFE 表示该函数不能在并行模式中执行；SQL 语句中只要出现这类函数就会强制使用串行执行计划。这是默认选项。PARALLEL RESTRICTED 表示该函数可以在并行模式中执行，但只能在并行组领导者进程中执行。PARALLEL SAFE 表示该函数可以在并行模式下不受限制地执行，包括在并行工作进程中执行。

如果函数修改任何数据库状态，或者通过使用子事务等方式改变事务，或者访问序列或尝试对设置进行持久更改（例如使用setval），则应将其标记为并行不安全。如果函数访问临时表、客户端连接状态、游标、预备语句或其他系统无法在并行模式下同步的后端本地状态，则应将其标记为并行受限（例如，setseed 只能由组领导者执行，因为其他进程所做的更改不会反映在领导者中）。一般来说，如果函数在受限或不安全时被标记为安全，或者实际上不安全时被标记为受限，那么在并行查询中使用它可能会抛出错误或产生错误结果。C 语言函数如果被错误标记，理论上可能表现出完全未定义的行为，因为系统无法保护自己免受任意 C 代码的影响，但在最可能的情况下，其结果不会比任何其他函数更糟。如果不确定，函数应标记为UNSAFE，这是默认值。

```
COST execution_cost
```

一个正数，给出该函数的估计执行代价，单位为 cpu_operator_cost。如果该函数返回一个集合，则这是每个返回行的代价。如果未指定代价，则对 C 语言和内部函数假定为 1 个单位，对其他所有语言的函数假定为 100 个单位。较大的值会让规划器尽量避免对该函数进行不必要的频繁求值。

```
ROWS result_rows
```

一个正数，给出规划器应预计该函数返回的行数。只有函数被声明为返回集合时才允许使用此项。默认假定为 1000 行。

```
configuration_parameter
value
```

SET 子句会在进入函数时将指定的配置参数设为给定值，并在函数退出时恢复为先前的值。SET FROM CURRENT 会把执行 CREATE FUNCTION 时该参数的当前值保存下来，作为进入函数时要应用的值。

如果函数附带了 SET 子句，那么在函数内针对同一变量执行的SET LOCAL 命令，其效果会被限制在该函数内部：函数退出时，配置参数先前的值仍会被恢复。不过，普通的SET 命令（不带 LOCAL）会覆盖 SET 子句，就像它会覆盖先前的SET LOCAL 命令一样：这类命令的效果会在函数退出后继续保持，除非当前事务被回滚。

关于允许的参数名和值的更多信息，见SET 和第 19 章。

definition

一个定义该函数的字符串常量，其含义取决于所用语言。它可以是一个内部函数名称、一个对象文件的路径、一个 SQL 命令，或者用一种过程语言编写的文本。

使用美元引用（见第 4.1.2.4 节）来书写函数定义字符串通常会更有帮助，而不是使用普通的单引号语法。如果没有美元引用，函数定义中的任何单引号或者反斜线都必须用双写来转义。

obj_file, *link_symbol*

当 C 语言源代码中的函数名称与 SQL 函数名称不同时，这种形式的AS 子句用于可动态加载的 C 语言函数。字符串 *obj_file* 是包含已编译 C 函数的共享库文件名，其解释方式与 LOAD 命令相同。字符串 *link_symbol* 是函数的链接符号，即 C 语言源代码中函数的名称。如果省略链接符号，则假定它与正在定义的 SQL 函数名称相同。所有函数的 C 名称都必须不同，因此必须为重载的 C 函数指定不同的 C 名称（例如将参数类型作为 C 名称的一部分）。

当重复的 CREATE FUNCTION 调用引用同一个对象文件时，该文件在每个会话中只装载一次。要卸载并重新装载该文件（例如在开发期间），请启动一个新会话。

重载

PostgreSQL 允许函数重载；也就是说，只要输入参数类型不同，同一个名称就可以用于多个不同的函数。无论你是否使用这一能力，在某些用户不信任其他用户的数据库中调用函数时，都需要采取安全预防措施；参见第 10.3 节。

如果两个函数具有相同的名称和输入参数类型，它们被认为相同（不考虑任何OUT 参数）。因此这些声明会冲突：

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

参数类型列表不同的函数在创建时不会被视为冲突，但如果提供了默认值，则在使用时可能发生冲突。例如，考虑下面这些声明：

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

调用 foo(10) 会失败，因为系统无法确定应该调用哪个函数。

注解

允许使用完整的 SQL 类型语法来声明函数参数和返回值。不过，`CREATE FUNCTION` 会丢弃带圆括号的类型修饰符（例如 `numeric` 类型的精度字段）。因此，`CREATE FUNCTION foo ( varchar(10)) ...`与 `CREATE FUNCTION foo (varchar) ...`完全等同。

在用 `CREATE OR REPLACE FUNCTION` 替换现有函数时，更改参数名会受到限制。不能更改已经分配给任何输入参数的名称（但可以给先前没有名称的参数补上名称）。如果输出参数多于一个，也不能更改输出参数的名称，因为那会改变描述函数结果的匿名复合类型的列名。这些限制是为了确保函数被替换时，已有的函数调用不会停止工作。

如果一个函数被声明为带有 `VARIADIC` 参数的 `STRICT` 函数，则严格性检查测试的是可变参数数组作为一个整体是否非空。如果该数组包含空值元素，仍会调用该函数。

示例

使用 SQL 函数将两个整数相加：

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

在 PL/pgSQL 中，使用参数名把一个整数加 1：

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

返回一个包含多个输出参数的记录：

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

你也可以用一个显式命名的复合类型，更详细地表达同样的意思：

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

另一种返回多列的方法是使用 `TABLE` 函数：

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

不过，`TABLE` 函数与前面的示例不同，因为它实际返回的是一组记录，而不只是单条记录。

安全地编写 SECURITY DEFINER 函数

因为 SECURITY DEFINER 函数要以拥有它的用户的权限执行，所以必须小心确保该函数不会被滥用。出于安全考虑，应将search_path 设置为排除任何可被不受信任用户写入的模式。这可以防止恶意用户创建对象（例如表、函数和操作符）来遮蔽该函数原本打算使用的对象。在这方面尤其重要的是临时表模式；默认情况下它最先被搜索，而且通常任何人都可写。一个安全的安排是强制把临时模式放到搜索顺序的最后。要做到这一点，应把pg_temp 写成 search_path 中的最后一项。下面这个函数展示了安全用法：

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        RETURN passed;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER
    -- 设置一个安全的 search_path: 受信的模式，然后是 'pg_temp'。
    SET search_path = admin, pg_temp;
```

这个函数的意图是访问 admin.pwds 表。但如果没有 SET 子句，或者SET 子句只提到 admin，那么该函数就可能因为有人创建一个名为pwds 的临时表而被利用。

在 PostgreSQL 8.3 版本之前，SET 子句尚不可用，因此较旧的函数可能包含相当复杂的逻辑来保存、设置和恢复search_path。使用 SET 子句更容易实现此目的。

另一点需要记住的是，默认情况下，新创建的函数会把执行权限授予PUBLIC（详见 GRANT）。通常你会希望只允许某些用户使用 SECURITY DEFINER 函数。要做到这一点，必须先撤销默认的PUBLIC 权限，然后有选择地授予执行权限。为了避免新函数在一段时间窗口内对所有人都可访问，应在同一个事务中创建该函数并设置权限。例如：

```
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
COMMIT;
```

兼容性

CREATE FUNCTION 命令在 SQL 标准中定义。PostgreSQL 版本与其相似，但并不完全兼容。这些属性和可用的不同语言都不具备可移植性。

为与某些其他数据库系统兼容，*argmode* 可以写在 *argname* 之前或之后，但只有前一种写法符合标准。

对于参数默认值，SQL 标准只规定了带有DEFAULT 关键字的语法。带=的语法用于 T-SQL 和 Firebird。

另见

ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE

CREATE GROUP

CREATE GROUP — 定义一个新的数据库角色

大纲

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]
```

其中 *option* 可以是：

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

描述

`CREATE GROUP` 现在是 CREATE ROLE 的别名。

兼容性

SQL 标准中没有 `CREATE GROUP` 语句。

另见

CREATE ROLE

CREATE INDEX

CREATE INDEX — 定义一个新索引

大纲

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON [ ONLY ] table_name [
 USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS
 { FIRST | LAST } ] [, ...] )
    [ INCLUDE ( column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

描述

`CREATE INDEX` 在指定关系的指定列上构建一个索引，该关系可以是表或物化视图。索引主要用于提升数据库性能（但使用不当也可能导致性能下降）。

索引的键字段指定为列名，或者指定为写在圆括号中的表达式。如果索引方法支持多列索引，则可以指定多个字段。

索引字段可以是根据表行中一个或多个列值计算得到的表达式。该特性可用于根据基础数据的某种变换来快速访问数据。例如，在`upper(col)` 上计算的索引可让子句 `WHERE upper(col) = 'JIM'` 使用索引。

PostgreSQL 提供了索引方法 B-树、hash、GiST、SP-GiST、GIN 以及 BRIN。用户也可以定义自己的索引方法，但这相当复杂。

当 `WHERE` 子句存在时，会创建一个部分索引。部分索引只包含表中一部分行的索引项，通常这一部分比表的其余部分更适合建立索引。例如，如果一个表同时包含已开票和未开票订单，而未开票订单只占整个表的一小部分，但这一部分又经常被访问，就可以只对这部分创建索引来提升性能。另一种可能的应用是将`WHERE` 与 `UNIQUE` 结合使用，以便在表的一个子集上强制唯一性。更多讨论请见第 11.8 节。

`WHERE` 子句中使用的表达式只能引用底层表的列，但它可以使用所有列，而不仅仅是被索引的列。当前，`WHERE` 中也禁止使用子查询和聚合表达式。同样的限制也适用于作为表达式的索引字段。

所有在索引定义中使用的函数和操作符都必须是“不可变的”，也就是说，它们的结果只能依赖其参数，而不能受任何外部因素影响（例如另一个表的内容或当前时间）。这种限制确保索引的行为定义明确。要在索引表达式或`WHERE` 子句中使用用户定义的函数，记得在创建该函数时将其标记为不可变。

参数

`UNIQUE`

使系统在创建索引时（如果数据已经存在）以及每次添加数据时，检查表中的重复值。任何会导致重复项的插入或更新操作都会报错。

当唯一索引用于分区表时，还会有额外限制；参见 CREATE TABLE。

CONCURRENTLY

使用此选项时，PostgreSQL 将在构建索引时不获取任何会阻止对表进行并发插入、更新或删除的锁；而标准索引构建会阻塞对表的写入（但不会阻塞读取），直到构建完成。使用此选项时有几个注意事项需要了解–请参见并发构建索引。

对于临时表，CREATE INDEX 始终是非并发的，因为没有其他会话可以访问它们，而且非并发创建索引的代价更低。

IF NOT EXISTS

如果同名关系已存在，则不抛出错误，而是发出一个提示。注意，现有索引并不保证与本应创建的索引有任何相似之处。指定 IF NOT EXISTS 时，必须提供索引名。

INCLUDE

可选的 INCLUDE 子句指定一个列列表，这些列将作为非键列包含在索引中。非键列不能用作索引扫描的搜索限定条件，在索引强制执行的任何唯一性或排他约束中也会被忽略。不过，非键列的内容可以由仅索引扫描返回，而无需访问索引对应的表，因为它们可直接从索引项中取得。因此，添加非键列可以让原本无法使用仅索引扫描的查询也能使用它。

向索引添加非键列时，谨慎一些是明智的，尤其是宽列。如果索引元组超过该索引类型允许的最大大小，数据插入将失败。无论如何，非键列会复制索引所对应的表中的数据并使索引膨胀，从而可能降低搜索速度。

INCLUDE 子句中列出的列不需要合适的操作符类；该子句可以包含数据类型尚未为给定访问方法定义操作符类的列。

不支持将表达式作为内含列，因为它们不能用于仅索引扫描。

目前，只有 B-树索引访问方法支持此功能。在 B-树索引中，INCLUDE 子句中列出的列值会包含在对应堆元组的叶元组中，但不会包含在用于树导航的上层索引条目中。

name

要创建的索引名称。此处不能包含模式名称；索引始终创建在其父表所在的同一模式中。如果省略名称，PostgreSQL 会根据父表名称和被索引的列名选择合适的名称。

ONLY

表示如果该表是分区表，则不要递归地在各分区上创建索引。默认会递归。

table_name

要建立索引的表名（可以是模式限定名）。

method

要使用的索引方法的名称。选择包括 btree、hash、gist、spgist、gin、brin，或用户安装的访问方法，如 bloom。默认方法是 btree。

column_name

一个表列的名称。

expression

一个基于表中一个或多个列的表达式。通常必须像语法中所示那样写在外围圆括号中。不过，如果该表达式是函数调用形式，则可以省略圆括号。

collation

将用于该索引的排序规则名称。默认情况下，索引使用被索引列声明的排序规则，或者被索引表达式的结果排序规则。对于涉及使用非默认排序规则表达式的查询，使用非默认排序规则的索引可能会很有用。

opclass

一个操作符类的名称。详见下文。

ASC

指定升序排序（默认）。

DESC

指定降序排序。

NULLS FIRST

指定把空值排序在非空值前面。在指定 DESC 时，这是默认行为。

NULLS LAST

指定把空值排序在非空值后面。在没有指定 DESC 时，这是默认行为。

storage_parameter

索引方法专用存储参数的名称。有关详细信息，请参见索引存储参数。

tablespace_name

在其中创建索引的表空间。如果未指定，将查阅 default_tablespace；对于临时表上的索引，则查阅 temp_tablespaces。

predicate

部分索引的约束表达式。

索引存储参数

可选的 WITH 子句为索引指定存储参数。每一种索引方法都有其各自允许的存储参数集合。B-树、hash、GiST 和 SP-GiST 索引方法都接受以下参数：

fillfactor

索引的填充因子是一个百分比，用于确定索引方法将尝试把索引页填充到多满。对于 B-树，在初始构建索引期间，以及向右扩展索引（添加新的最大键值）时，叶页都会填充到该百分比。如果之后页面变得完全填满，就会进行拆分，导致索引效率逐渐下降。B-树使用默认填充因子 90，但可以选择 10 到 100 之间的任意整数值。如果表是静态的，填充因子 100 最适

合将索引的物理大小降至最低；但对于频繁更新的表，较小的填充因子更适合减少页面拆分的需要。其他索引方法以不同但大致类似的方式使用填充因子；默认填充因子因方法而异。

B-树索引还接受以下参数：

`vacuum_cleanup_index_scale_factor`

每个索引的 vacuum_cleanup_index_scale_factor 值。

GiST 索引还接受以下参数：

`buffering`

确定是否使用第 64.4.1 节中描述的缓冲构建技术来构建索引。设置为 `OFF` 时禁用，设置为 `ON` 时启用，设置为`AUTO` 时初始禁用，但一旦索引大小达到 effective_cache_size，就会动态启用。默认值为`AUTO`。

GIN 索引接受不同的参数：

`fastupdate`

此设置控制第 66.4.1 节中描述的快速更新技术的使用。这是一个布尔参数：`ON` 启用快速更新，`OFF` 禁用快速更新。（如第 19.1 节中所述，允许使用`ON` 和 `OFF` 的其他拼写形式。）默认值为 `ON`。

> **注意**
>
> 通过 `ALTER INDEX` 关闭 `fastupdate` 会阻止后续插入进入待处理索引项列表，但这本身不会刷新现有条目。之后可能需要对该表执行`VACUUM`，或调用 `gin_clean_pending_list` 函数，以确保待处理列表被清空。

`gin_pending_list_limit`

自定义 gin_pending_list_limit 参数。该值以千字节为单位。

BRIN 索引接受不同的参数：

`pages_per_range`

定义每个 BRIN 索引项对应的一个块范围由多少个表块组成（详见第 67.1 节）。默认值为 `128`。

`autosummarize`

定义每当在下一页范围检测到插入时，是否为上一页范围执行一次范围摘要操作。

并发构建索引

创建索引可能会干扰数据库的正常运行。通常 PostgreSQL 会锁住要建立索引的表，阻止其写入，并通过一次扫描完成整个索引构建。其他事务仍可读取该表，但如果它们试图在表中插入、更新或删除行，就会阻塞直到索引构建完成。如果系统是在线生产数据库，这可能产生严重影响。对非常大的表建立索引可能需要很多小时，即便是较小的表，索引构建也可能在一段对生产系统而言不可接受的时间内阻止写入者操作。

PostgreSQL 支持在不阻止写入的情况下构建索引。这种方法通过在 CREATE INDEX 中指定 CONCURRENTLY 选项来启用。使用该选项时，PostgreSQL 必须对该表执行两次扫描，此外还必须等待所有现有、可能修改或使用该索引的事务结束。因此，这种方法比标准索引构建需要更多总工作量，完成时间也明显更长。不过，由于它允许在构建索引期间继续进行正常操作，所以这种方法适合在生产环境中新增索引。当然，创建索引带来的额外 CPU 和 I/O 负载也可能拖慢其他操作。

在并发索引构建中，索引实际上会在一个事务中录入系统目录，然后在另外两个事务中执行两次表扫描。每次表扫描之前，索引构建都必须等待已修改该表的现有事务结束。第二次扫描之后，索引构建必须等待所有持有早于第二次扫描的快照（参见第 13 章）的事务结束，其中包括其他表上并发索引构建任一阶段使用的事务。最后，索引才可以标记为可用，CREATE INDEX 命令随之结束。不过即便如此，索引也可能无法立即用于查询：在最坏情况下，只要还存在早于索引构建开始的事务，就不能使用它。

如果在扫描表时出现问题，例如死锁或唯一索引中的唯一性冲突，CREATE INDEX 命令将失败，但会留下一个“无效”索引。由于该索引可能不完整，查询时会忽略它；但是，它仍会带来更新开销。该psql \d 命令会将此类索引报告为 INVALID：

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 col    | integer |           |          |
Indexes:
    "idx" btree (col) INVALID
```

在这种情况下，建议的恢复方法是删除索引，然后重新尝试执行CREATE INDEX CONCURRENTLY。（另一种可能性是重建索引，使用REINDEX。但是，由于REINDEX 不支持并发构建，此选项不太有吸引力。）

并发构建唯一索引时的另一项注意事项是，在第二次表扫描开始时，唯一性约束就已经开始对其他事务生效了。这意味着在该索引可供使用之前，其他查询就可能报告约束违规，甚至在索引构建最终失败的情况下也是如此。另外，如果第二次扫描确实失败了，那个“无效”索引之后仍会继续强制执行其唯一性约束。

也支持并发构建表达式索引和部分索引。计算这些表达式时发生的错误，可能导致与上文所述唯一性约束违规类似的行为。

普通索引构建允许在同一张表上同时进行其他普通索引构建，但一张表在同一时刻只能发生一次并发索引构建。无论哪种情况，在索引构建期间都不允许修改该表的模式。另一个区别是，普通 CREATE INDEX 命令可以在事务块内执行，而CREATE INDEX CONCURRENTLY 不能。

当前不支持对分区表的索引进行并发构建。不过，可以分别在每个分区上并发构建索引，最后再以非并发方式创建分区索引，从而缩短分区表写入被阻止的时间。在这种情况下，构建分区索引只是元数据操作。

注解

关于索引何时能被使用、何时不被使用以及什么情况下它们有用的信息请见第 11 章。

目前，只有 B-树、GiST、GIN 和 BRIN 索引方法支持多列索引。默认最多可指定 32 个字段。（构建PostgreSQL 时可以更改此限制。）目前只有 B-树支持唯一索引。

索引的每一列都可以指定一个操作符类。操作符类标识索引用于该列的操作符。例如，四字节整数上的 B-树索引会使用int4_ops 类；此操作符类包含四字节整数的比较函数。实际上，列数据类型的默认操作符类通常已足够。操作符类存在的主要原因是，某些数据类型可能有多个有意义的排序方式。例如，我们可能希望按绝对值或实部对复数数据类型排序。我们可以为该数据类型定义两个操作符类，并在创建索引时选择合适的类。有关操作符类的更多信息，请参见第 11.10 节和第 38.15 节。

当在分区表上调用 CREATE INDEX 时，默认行为是递归到所有分区，以确保它们都拥有匹配的索引。系统首先检查每个分区是否已经存在等效索引；如果存在，该索引会作为正在创建索引的分区索引附接到其上，而被创建的索引将成为其父索引。若不存在匹配索引，则会创建一个新索引并自动附着；每个分区中新索引的名称会按命令中未指定索引名时的规则确定。如果指定ONLY 选项，则不进行递归，且该索引会被标记为无效。（一旦所有分区都获得匹配索引，ALTER INDEX ... ATTACH PARTITION 会将该索引标记为有效。）不过请注意，将来使用 CREATE TABLE ... PARTITION OF 创建的任何分区都会自动拥有匹配索引，无论是否指定了ONLY。

对于支持有序扫描的索引方法（当前只有 B-树），可以指定可选子句 ASC、DESC、NULLS FIRST 和/或 NULLS LAST 来修改索引的排序顺序。由于有序索引可以向前或向后扫描，因此创建单列DESC 索引通常并无用处 – 常规索引已经提供了这种排序顺序。这些选项的价值在于可以创建与混合排序查询所要求顺序相匹配的多列索引，例如 SELECT ... ORDER BY x ASC, y DESC。如果需要在依靠索引避免排序步骤的查询中支持“空值排在低位”，而不是默认的“空值排在高位”行为，那么NULLS 选项就很有用。

系统定期收集表中所有列的统计信息。新创建的非表达式索引可以立即使用这些统计信息来确定索引的有用性。对于新的表达式索引，需要运行 ANALYZE 或等待自动清理守护进程分析表以生成这些索引的统计信息。

对于大多数索引方法，索引的创建速度取决于maintenance_work_mem 的设置。较大的值将会减少索引创建所需的时间，当然不要把它设置得超过实际可用的内存量（那会迫使机器进行交换）。

PostgreSQL 可以在构建索引时利用多个 CPU，以便更快处理表中的行。这项特性被称为并行索引构建。对于支持并行构建索引的索引方法（当前只有 B-树），maintenance_work_mem 指定每次索引构建操作整体最多可使用的内存量，而不管启动了多少工作者进程。通常，代价模型会自动决定是否需要请求工作者进程，以及请求多少个。

增加 maintenance_work_mem 可能会让并行索引构建受益，而等效的串行索引构建则几乎看不到收益。注意，maintenance_work_mem 可能会影响所请求工作者进程的数量，因为每个并行工作者都必须至少从总的 maintenance_work_mem 预算中分得 32MB。领导者进程也必须保留 32MB。增加 max_parallel_maintenance_workers 可能允许使用更多工作者，从而减少索引创建所需时间，前提是索引构建尚未受到 I/O 限制。当然，还应当有足够的、原本会闲置的 CPU 容量。

通过 ALTER TABLE 为 parallel_workers 设置一个值，可以直接控制 CREATE INDEX 对该表会请求多少并行工作者进程。这会完全绕过代价模型，也使maintenance_work_mem 不再影响所请求的并行工作者数量。通过 ALTER TABLE 将 parallel_workers 设置为 0，将在所有情况下禁用该表上的并行索引构建。

> **提示**
>
> 在把 parallel_workers 作为索引构建调优的一部分进行设置后，可能需要将其重置。这样可以避免无意间改变查询计划，因为 parallel_workers 会影响所有并行表扫描。

虽然带有 CONCURRENTLY 选项的 CREATE INDEX 在没有特殊限制的情况下支持并行构建，但实际上只有第一次表扫描会以并行方式执行。

使用 DROP INDEX 删除索引。

早期版本的 PostgreSQL 还提供过一种 R-tree 索引方法。该方法已经被移除，因为它相对于 GiST 方法并无明显优势。如果指定了USING rtree，CREATE INDEX 会将其解释为 USING gist，以简化旧数据库向 GiST 的转换。

示例

要在表 films 的列 title 上创建一个唯一 B-树索引：

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

要在表 films 的列 title 上创建一个唯一 B-树索引，并将director 和 rating 作为内含列：

```
CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director, rating);
```

要在表达式 lower(title) 上创建一个索引，以便高效执行不区分大小写的搜索：

```
CREATE INDEX ON films ((lower(title)));
```

（在这个示例中，索引名称被省略，因此系统会选择一个名字，通常为films_lower_idx。）

要创建一个具有非默认排序规则的索引：

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

要创建一个具有非默认空值排序顺序的索引：

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

要创建一个具有非默认填充因子的索引：

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

要创建一个禁用快速更新的 GIN 索引：

```
CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate = off);
```

要在表 films 的列 code 上创建一个索引，并让该索引驻留在表空间indexspace 中：

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

要在点属性上创建一个 GiST 索引，以便能够在转换函数的结果上高效地使用 box 操作符：

```
CREATE INDEX pointloc
    ON points USING gist (box(location,location));
SELECT * FROM points
    WHERE box(location,location) && '(0,0),(1,1)'::box;
```

要在不阻止对表执行写操作的情况下创建索引：

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

兼容性

`CREATE INDEX` 是 PostgreSQL 的语言扩展。SQL 标准中没有关于索引的规定。

另见

ALTER INDEX, DROP INDEX

CREATE LANGUAGE

CREATE LANGUAGE — 定义一种新的过程语言

大纲

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
```

描述

CREATE LANGUAGE 在 PostgreSQL 数据库中注册一种新的过程语言。随后，就可以用这种新语言定义函数和过程。

> **注意**
>
> 从 PostgreSQL 9.1 开始，大多数过程语言都已变成“扩展”，因此应使用CREATE EXTENSION 而不是 CREATE LANGUAGE 来安装。现在直接使用 CREATE LANGUAGE 应仅限于扩展安装脚本。如果数据库中有一个“裸”语言（例如升级的结果），可以使用 CREATE EXTENSION *langname* FROM unpackaged 将其转换为扩展。

CREATE LANGUAGE 实际上是将该语言名称与负责执行用这种语言编写的函数的处理器函数关联起来。关于语言处理器的更多信息，参见第 56 章。

CREATE LANGUAGE 命令有两种形式。第一种形式中，用户只提供所需语言的名称，PostgreSQL 服务器会查询 pg_pltemplate 系统目录，以确定正确的参数。第二种形式中，用户在提供语言名称的同时也提供语言参数。第二种形式可用来创建未在pg_pltemplate 中定义的语言，但这种做法被认为已经过时。

当服务器在 pg_pltemplate 目录中找到给定语言名称的条目时，即使命令包含语言参数，也会使用目录中的数据。这种行为简化了旧转储文件的装载，因为它们很可能包含关于语言支持函数的过时信息。

通常，用户必须具有PostgreSQL 超级用户权限才能注册新语言。不过，如果语言列在 pg_pltemplate 目录中，并且标记为允许数据库所有者创建（tmpldbacreate 为真），数据库所有者就可以在该数据库中注册这种新语言。默认情况下，数据库所有者可以创建受信任的语言，但超级用户可以通过修改pg_pltemplate 的内容来调整这一点。语言的创建者成为其所有者，以后可以删除它、重命名它或将它分配给新的所有者。

CREATE OR REPLACE LANGUAGE 会创建新语言，或替换现有定义。如果语言已经存在，则会根据指定值或从pg_pltemplate 获取的值更新其参数，但不会改变语言的所有权和权限设置，并且假定用该语言编写的任何现有函数仍然有效。除了满足创建语言的常规权限要求外，用户还必须是超级用户或现有语言的所有者。REPLACE 主要用于确保语言存在。如果语言具有 pg_pltemplate 条目，则REPLACE 实际上不会改变现有定义的任何内容，除非出现少见的情况：该pg_pltemplate 条目在语言创建后被修改过。

参数

TRUSTED

TRUSTED 指定该语言不会让用户获得其原本无权访问的数据访问能力。如果在注册该语言时省略这个关键字，只有拥有PostgreSQL 超级用户权限的用户才能使用该语言创建新函数。

PROCEDURAL

这是一个噪声词。

name

新过程语言的名称。该名称不能与数据库中其他语言的名称重复。

为了向后兼容，名称可以用单引号括起来。

HANDLER *call_handler*

call_handler 是一个先前注册的函数名称，它会被调用来执行该过程语言中的函数。过程语言的调用处理器必须使用某种编译型语言（例如 C）编写，并采用版本 1 调用约定。它必须在PostgreSQL 中注册为一个不带参数且返回 language_handler 类型的函数。language_handler 是一种占位符类型，仅用于把该函数标识为调用处理器。

INLINE *inline_handler*

inline_handler 是一个先前注册的函数名称，它会被调用来执行该语言中的匿名代码块（DO 命令）。如果没有指定 *inline_handler* 函数，则该语言不支持匿名代码块。该处理器函数必须接受一个 internal 类型的参数，该参数将是 DO 命令的内部表示，并且它通常返回 void。处理器的返回值会被忽略。

VALIDATOR *valfunction*

valfunction 是一个先前注册的函数名称，在创建该语言中的新函数时会调用它来验证该新函数。如果没有指定验证器函数，那么新函数在创建时不会被检查。验证器函数必须接受一个oid 类型的参数，它将是待创建函数的 OID，并且它通常返回void。

验证器函数通常会检查函数体的语法正确性，但它也可以检查函数的其他属性，例如该语言是否支持某些参数类型。要报告错误，验证器函数应使用ereport() 函数。函数的返回值会被忽略。

TRUSTED 选项和支持函数名称会被忽略，前提是服务器在pg_pltemplate 中有指定语言名称的条目。

注解

使用 DROP LANGUAGE 删除过程语言。

系统目录 pg_language（见第 52.29 节）记录当前已安装语言的信息。此外，psql 的\dL 命令也会列出已安装的语言。

要用某种过程语言创建函数，用户必须拥有该语言的 USAGE 权限。默认情况下，对于受信任的语言，USAGE 会授予 PUBLIC（即所有人）。如果需要，也可以撤销这项授权。

过程语言是各个数据库本地的。不过，可以将某种语言安装到 template1 数据库中，这样它就会在之后创建的所有数据库中自动可用。

如果服务器在 pg_pltemplate 中没有该语言的条目，则调用处理器函数、内联处理器函数（如果有）和验证器函数（如果有）必须已经存在。但如果有条目，则这些函数不必事先存在；如果数据库中不存在它们，就会自动定义它们。（如果安装中没有实现该语言的共享库，这可能导致 CREATE LANGUAGE 失败。）

在 7.3 之前的 PostgreSQL 版本中，必须将处理器函数声明为返回占位符类型opaque，而不是language_handler。为了支持装载旧转储文件，CREATE LANGUAGE 会接受声明为返回 opaque 的函数，但会发出提示，并将函数声明的返回类型改为language_handler。

示例

创建任何标准过程语言的首选方式很简单：

```
CREATE LANGUAGE plperl;
```

对于 pg_pltemplate 目录中未知的语言，需要类似以下的命令序列：

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

兼容性

CREATE LANGUAGE 是一种 PostgreSQL 扩展。

另见

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE

CREATE MATERIALIZED VIEW

CREATE MATERIALIZED VIEW — 定义一个新物化视图

大纲

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

描述

`CREATE MATERIALIZED VIEW` 定义一个基于查询的物化视图。在发出该命令时，会执行该查询并用其结果填充该视图（除非使用了 `WITH NO DATA`），之后可以使用 `REFRESH MATERIALIZED VIEW` 来刷新它。

`CREATE MATERIALIZED VIEW` 类似于 `CREATE TABLE AS`，不过它还会记住用于初始化该视图的查询，以便后续按需刷新。物化视图具有很多与表相同的属性，但不支持临时物化视图，也不支持自动生成 OID。

参数

`IF NOT EXISTS`

如果已存在同名物化视图，则不要抛出错误。在这种情况下会发出一条提示。请注意，不能保证现有的物化视图与原本会创建出来的那个物化视图有任何相似之处。

`table_name`

要创建的物化视图名称（可以带模式限定）。

`column_name`

新物化视图中的列名。如果没有提供列名，则采用查询输出列的名称。

`WITH ( storage_parameter [= value] [, ... ] )`

该子句为新物化视图指定可选的存储参数；更多信息见存储参数。凡是 `CREATE TABLE` 支持的参数，`CREATE MATERIALIZED VIEW` 也同样支持，但`OIDS` 除外。更多信息见 CREATE TABLE。

`TABLESPACE tablespace_name`

`tablespace_name` 是在其中创建新物化视图的表空间名称。如果没有指定，将查阅 default_tablespace。

query

一个 SELECT、TABLE 或 VALUES 命令。该查询将在一种安全受限的操作中运行。特别是，调用那些自身会创建临时表的函数将会失败。

`WITH [ NO ] DATA`

该子句指定是否在创建时填充物化视图。如果不填充，则该物化视图会被标记为不可扫描，在使用`REFRESH MATERIALIZED VIEW`之前不能查询。

兼容性

`CREATE MATERIALIZED VIEW`是一种 PostgreSQL 扩展。

另见

ALTER MATERIALIZED VIEW, CREATE TABLE AS, CREATE VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

CREATE OPERATOR

CREATE OPERATOR — 定义一个新的操作符

大纲

```
CREATE OPERATOR name (
    {FUNCTION|PROCEDURE} = function_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
```

描述

`CREATE OPERATOR` 定义一个新的操作符 *name*。定义该操作符的用户将成为其拥有者。如果给出了模式名，该操作符将在指定的模式中创建；否则将在当前模式中创建。

操作符名称最多可以由 `NAMEDATALEN`-1 个字符（默认为 63 个）组成，这些字符必须取自下列列表：

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

名称选择还有一些限制：

- `--`和`/*` 不能出现在操作符名称的任何位置，因为它们会被视为注释的开始。
- 多字符操作符名称不能以`+`或`-` 结尾，除非该名称中还至少包含下列字符之一：

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  例如，`@-`是允许的操作符名称，而 `*-`则不允许。这一限制使 PostgreSQL 能够在不要求词元之间必须有空格的情况下解析符合 SQL 规范的命令。
- 符号`=>` 被 SQL 语法保留，因此不能用作操作符名称。

输入`!=`时会被映射为`<>`，因此这两个名称始终等效。

必须至少定义 `LEFTARG` 和 `RIGHTARG` 中的一个。对于二元操作符，必须同时定义两者。对于右一元操作符，只应定义`LEFTARG`；对于左一元操作符，只应定义`RIGHTARG`。

> **注意**
>
> 右一元操作符（也称后缀操作符）已被弃用，将在PostgreSQL 14 中移除。

function_name 函数必须事先已用 CREATE FUNCTION 定义，并且其定义必须接受正确数量（一个或两个）的指定类型参数。

在 CREATE OPERATOR 的语法中，关键字 FUNCTION 和 PROCEDURE 是等效的，但这里引用的对象无论如何都必须是函数而不是过程。此处使用关键字 PROCEDURE 是历史遗留做法，现已弃用。

其他子句用于指定可选的操作符优化子句。其含义详见第 38.14 节。

要创建操作符，你必须在参数类型和返回类型上具有 USAGE 权限，并在底层函数上具有 EXECUTE 权限。如果指定了交换子或求反器操作符，你还必须拥有这些操作符。

参数

name

要定义的操作符名称。允许使用的字符请见上文。该名称可以带模式限定，例如CREATE OPERATOR myschema.+ (...)。如果不带模式限定，该操作符将在当前模式中创建。同一模式中的两个操作符如果针对不同的数据类型进行操作，可以具有相同的名称。这被称为重载。

function_name

用于实现该操作符的函数。

left_type

该操作符左操作数的数据类型（如果有）。对于左一元操作符，应省略此选项。

right_type

该操作符右操作数的数据类型（如果有）。对于右一元操作符，应省略此选项。

com_op

该操作符的交换子。

neg_op

该操作符的求反器。

res_proc

该操作符的限制选择率估计函数。

join_proc

该操作符的连接选择率估算函数。

HASHES

表示该操作符可以支持哈希连接。

MERGES

表示该操作符可以支持归并连接。

要在 *com_op* 或其他可选参数中给出带模式限定的操作符名称，请使用OPERATOR() 语法，例如：

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

注解

更多信息请参见第 38.13 节。

无法在 CREATE OPERATOR 中指定一个操作符的词法优先级，因为解析器的优先级行为是硬写在代码中的。详见第 4.1.6 节。

废弃的选项 SORT1、SORT2、LTCMP 以及 GTCMP 以前被用来指定与可归并连接的操作符相关联的排序操作符名称。现在已不再需要这样做，因为相关操作符的信息会通过查找 B-树操作符族来获得。如果给出这些选项中的任意一个，除了会隐式将MERGES 设为真之外，其余都会被忽略。

使用 DROP OPERATOR 从数据库中删除用户定义的操作符。使用 ALTER OPERATOR 修改数据库中的操作符。

示例

下面的命令为数据类型 box 定义了一个新的面积相等操作符：

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    FUNCTION = area_equal_function,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_function,
    JOIN = area_join_function,
    HASHES, MERGES
);
```

兼容性

CREATE OPERATOR 是 PostgreSQL 扩展。在 SQL 标准中没有用户定义操作符的规定。

另见

ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR

CREATE OPERATOR CLASS

CREATE OPERATOR CLASS — 定义一个新的操作符类

大纲

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  {  OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ FOR SEARCH | FOR ORDER
 BY sort_family_name ]
   | FUNCTION support_number [ ( op_type [ , op_type ] ) ] function_name ( argument_type [, ...
] )
   | STORAGE storage_type
  } [, ... ]
```

描述

CREATE OPERATOR CLASS 创建一个新的操作符类。操作符类定义特定数据类型如何与索引一起使用。它规定某些操作符在该数据类型和该索引方法中承担特定角色或“策略”。当某个索引列选择了该操作符类时，操作符类还会指定索引方法要使用哪些支持函数。操作符类使用的所有操作符和函数都必须先定义好，然后才能创建该操作符类。

如果给出了一个模式名称，那么该操作符类会被创建在指定模式中。否则，它会被创建在当前模式中。同一模式中的两个操作符类只有在被用于不同的索引方法时才可以具有相同的名称。

定义操作符类的用户将成为其拥有者。目前，创建者必须是超级用户。（之所以有此限制，是因为错误的操作符类定义可能会使服务器混乱，甚至崩溃。）

CREATE OPERATOR CLASS 当前不会检查操作符类定义是否包括该索引方法所要求的所有操作符和函数，也不会检查这些操作符和函数是否构成一个自洽的集合。定义一个合法的操作符类是用户的责任。

相关的操作符类可以分组为操作符族。要把一个新的操作符类加入现有操作符族，可以在CREATE OPERATOR CLASS 中指定 FAMILY 选项。如果没有这个选项，新类会被放入一个与其同名的操作符族中（如果该操作符族尚不存在，就会创建它）。

更多信息请参见第 38.15 节。

参数

name

要创建的操作符类的名称。该名称可以被模式限定。

DEFAULT

如果出现，该操作符类将成为其数据类型的默认操作符类。对于某个特定的数据类型和索引方法，最多只能有一个默认操作符类。

data_type

该操作符类对应的列数据类型。

index_method

该操作符类所对应的索引方法的名称。

family_name

要把该操作符类加入其中的现有操作符族名称。如果没有指定，则使用一个与该操作符类同名的操作符族（如果它还不存在则创建）。

strategy_number

与该操作符类关联的操作符在此索引方法中的策略号。

operator_name

与该操作符类关联的操作符名称（可以是模式限定的）。

op_type

在 OPERATOR 子句中，表示该操作符的操作数数据类型，或者用NONE 表示一个左一元或右一元操作符。在通常情况下，如果操作数数据类型与操作符类的数据类型相同，则可以省略操作数数据类型。

在 FUNCTION 子句中，表示该函数意图支持的操作数数据类型；如果它不同于函数的输入数据类型（对于 B-树比较函数和 hash 函数），或者不同于该类的数据类型（对于 B-树排序支持函数，以及 GiST、SP-GiST、GIN 和 BRIN 操作符类中的所有函数），则需要在这里指定。上述默认值都是正确的，因此无需将*op_type* 指定在 FUNCTION 子句中；唯一的例外是某个 B-树排序支持函数打算支持跨数据类型比较的情形。

sort_family_name

一个现有 btree 操作符族的名称（可以是模式限定的），它描述与某个排序操作符关联的排序顺序。

如果既没有指定 FOR SEARCH，也没有指定 FOR ORDER BY，则默认值是FOR SEARCH。

support_number

与该操作符类关联的函数在此索引方法中的支持函数编号。

function_name

作为该操作符类索引方法支持函数的函数名称（可以是模式限定的）。

argument_type

该函数的参数数据类型。

storage_type

实际存储在索引中的数据类型。通常它与列数据类型相同，但是有些索引方法（目前是 GiST、GIN 和 BRIN）允许它们不同。除非索引方法允许使用不同类型，否则必须省略 STORAGE 子句。如果列的 *data_type* 被指定为 anyarray，则*storage_type* 可以声明为 anyelement，以表明索引条目属于每个特定索引所针对的实际数组类型的元素类型。

OPERATOR、FUNCTION 和 STORAGE 子句可以以任何顺序出现。

注解

因为索引机制在使用函数之前不会检查函数的访问权限，将一个函数或者操作符包括在一个操作符类中，相当于授予其公共执行权限。对适合放入操作符类的那类函数而言，这通常不是问题。

操作符不应该由 SQL 函数定义。SQL 函数很有可能会被内联到调用查询中，这会妨碍优化器识别该查询匹配一个索引。

在 PostgreSQL 8.4 之前，`OPERATOR` 子句可以包含 `RECHECK` 选项。现在不再支持该选项，因为索引操作符是否“有损”会在运行时动态确定。这样可以高效处理操作符可能有损也可能无损的情况。

示例

下面的示例为数据类型`_int4`（`int4` 数组）定义了一个 GiST 索引操作符类。完整示例见 intarray 模块。

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       = (anyarray, anyarray),
        OPERATOR        7       @>,
        OPERATOR        8       <@,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, smallint, oid, internal),
        FUNCTION        2       g_int_union (internal, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

兼容性

`CREATE OPERATOR CLASS` 是一种 PostgreSQL 扩展。在 SQL 标准中没有 `CREATE OPERATOR CLASS` 语句。

另见

ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE OPERATOR FAMILY, ALTER OPERATOR FAMILY

CREATE OPERATOR FAMILY

CREATE OPERATOR FAMILY — 定义一个新的操作符族

大纲

```
CREATE OPERATOR FAMILY name USING index_method
```

描述

`CREATE OPERATOR FAMILY` 创建一个新的操作符族。操作符族定义了一组相关的操作符类，以及可能还有一些与这些操作符类兼容、但对任何单个索引的正常工作都不是必需的额外操作符和支持函数。（那些对索引必不可少的操作符和函数应归入相应的操作符类，而不是作为操作符族中的“松散”成员。通常，单一数据类型的操作符属于操作符类，而跨数据类型的操作符则可以作为操作符族中的松散成员，该操作符族包含这两种数据类型对应的操作符类。）

新的操作符族最初是空的。应当通过随后发出的 `CREATE OPERATOR CLASS` 命令向其中添加所包含的操作符类，并可选择通过 `ALTER OPERATOR FAMILY` 命令添加“松散”操作符及其对应的支持函数。

如果给出了模式名称，该操作符族会被创建在指定的模式中。否则，它会被创建在当前模式中。只有当同一模式中的两个操作符族用于不同的索引方法时，它们才能具有相同的名称。

定义操作符族的用户将成为其拥有者。当前，创建该对象的用户必须是超级用户。（之所以这样限制，是因为错误的操作符族定义可能会使服务器混乱，甚至导致崩溃。）

更多信息请参见第 38.15 节。

参数

name

要创建的操作符族名称。该名称可以是模式限定的。

index_method

该操作符族所对应的索引方法名称。

兼容性

`CREATE OPERATOR FAMILY` 是一种 PostgreSQL 扩展。SQL 标准中没有 `CREATE OPERATOR FAMILY` 语句。

另见

ALTER OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

CREATE POLICY

CREATE POLICY — 为表定义新的行级安全策略

大纲

```
CREATE POLICY name ON table_name
    [ AS { PERMISSIVE | RESTRICTIVE } ]
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

描述

CREATE POLICY 命令为一个表定义一条新的行级安全策略。请注意，必须先在该表上启用行级安全（使用 ALTER TABLE ... ENABLE ROW LEVEL SECURITY），已创建的策略才会被应用。

策略授予选择、插入、更新或删除符合相应策略表达式的行的权限。使用USING 中指定的表达式检查现有表行，而使用WITH CHECK 中指定的表达式检查通过 INSERT 或 UPDATE 创建的新行。当 USING 表达式对给定行返回 true 时，该行对用户可见；如果返回 false 或 null，则该行不可见。当WITH CHECK 表达式对某行返回 true 时，该行会被插入或更新；如果返回 false 或 null，则发生错误。

对于 INSERT 和 UPDATE 语句，会在触发BEFORE 触发器之后、进行任何实际数据修改之前强制执行WITH CHECK 表达式。因此，BEFORE ROW 触发器可能会修改要插入的数据，从而影响安全策略检查的结果。WITH CHECK 表达式会在任何其他约束之前强制执行。

策略名称按表区分。因此，同一个策略名可以用于许多不同的表，并且在每个表上都可以有适合该表的定义。

策略可以针对特定命令或特定角色应用。除非另有指定，新建策略默认适用于所有命令和角色。多个策略可以应用于同一命令；更多细节见下文。表 241 总结了不同类型的策略如何应用于特定命令。

对于既可以具有 USING 又可以具有 WITH CHECK 表达式的策略（ALL 和 UPDATE），如果未定义 WITH CHECK 表达式，那么USING 表达式将同时用于决定哪些行可见（普通 USING 情形）以及允许写入哪些新行（WITH CHECK 情形）。

如果某个表启用了行级安全，但不存在适用的策略，则会假定存在一条“默认拒绝”策略，因此没有任何行可见或可更新。

参数

name

要创建的策略名称。它必须不同于该表上任何其他策略的名称。

table_name

该策略适用的表的名称（可选模式限定）。

PERMISSIVE

指定将该策略创建为宽松策略。适用于给定查询的所有宽松策略都会使用布尔 “OR” 操作符组合在一起。通过创建宽松策略，管理员可以扩大可访问行的集合。策略默认是宽松的。

RESTRICTIVE

指定将该策略创建为限制性策略。适用于给定查询的所有限制性策略都会使用布尔 “AND” 操作符组合在一起。通过创建限制性策略，管理员可以缩小可访问行的集合，因为每一行都必须通过所有限制性策略。

请注意，要让限制性策略能够有效缩小访问范围，必须先至少有一条宽松策略授予对行的访问。如果只存在限制性策略，则没有任何行可访问。当宽松策略和限制性策略混合存在时，只有在至少一条宽松策略通过且所有限制性策略也都通过时，某一行才可访问。

command

该策略适用的命令。有效选项是 ALL、SELECT、INSERT、UPDATE 和 DELETE。ALL 是默认值。有关这些策略如何应用的细节见下文。

role_name

该策略适用的角色（或多个角色）。默认是 PUBLIC，即对所有角色应用该策略。

using_expression

任何 SQL 条件表达式（返回 boolean）。条件表达式不能包含聚合函数或窗口函数。如果启用行级安全，该表达式会添加到引用该表的查询中。表达式返回 true 的行将对用户可见。表达式返回 false 或 null 的任何行对用户都不可见（在SELECT 中），也不能被修改（在 UPDATE 或 DELETE 中）。这类行会被静默抑制，不会报告错误。

check_expression

任何 SQL 条件表达式（返回 boolean）。条件表达式不能包含聚合函数或窗口函数。如果启用行级安全，该表达式会用于针对该表的INSERT 和 UPDATE 查询。只有表达式计算结果为 true 的行才会被允许。如果对于任何插入的记录或更新产生的任何记录，表达式计算结果为 false 或 null，则会抛出错误。请注意，*check_expression* 是针对行预计的新内容计算的，而不是针对原始内容。

针对每种命令的策略

ALL

对策略使用 ALL 意味着无论命令类型如何，该策略都适用于所有命令。如果存在ALL 策略且还存在更具体的策略，则ALL 策略和更具体的策略（或多个策略）都会被应用。此外，ALL 策略会同时应用于查询的选择端和修改端，并在两端使用USING 表达式（如果只定义了 USING 表达式）。

例如，如果发出UPDATE，则ALL 策略既适用于 UPDATE 能够选择哪些行进行更新（应用 USING 表达式），也适用于更新后的行，以检查它们是否允许被添加回表中（如果定义了

WITH CHECK 表达式则应用该表达式，否则应用USING 表达式）。如果 INSERT 或 UPDATE 命令尝试向表中添加不通过 ALL 策略的 WITH CHECK 表达式的行，则整个命令都会中止。

SELECT

对策略使用 SELECT 意味着该策略适用于 SELECT 查询，以及在策略所定义的关系上需要 SELECT 权限的任何时候。因此，在SELECT 查询期间，只会返回关系中通过SELECT 策略的记录；需要 SELECT 权限的查询（例如 UPDATE）也只能看到被 SELECT 策略允许的记录。SELECT 策略不能有 WITH CHECK 表达式，因为它只适用于从关系中检索记录的情况。

INSERT

对策略使用 INSERT，意味着它将适用于 INSERT 命令。插入的行如果未通过该策略，将导致策略违规错误，并且整个INSERT 命令将被中止。INSERT 策略不能带有 USING 表达式，因为它只适用于向关系添加行的情况。

请注意，带有ON CONFLICT DO UPDATE 的 INSERT 只会为通过 INSERT 路径添加到关系中的行检查 INSERT 策略的 WITH CHECK 表达式。

UPDATE

对策略使用 UPDATE，意味着它将适用于 UPDATE、SELECT FOR UPDATE、SELECT FOR SHARE 命令，以及INSERT 命令中辅助的 ON CONFLICT DO UPDATE 子句。由于 UPDATE 命令需要取出现有行并用修改后的新行替换它，因此UPDATE 策略同时接受 USING 表达式和 WITH CHECK 表达式。USING 表达式决定 UPDATE 命令能够看到哪些行来执行操作，而 WITH CHECK 表达式则定义哪些修改后的行允许被写回该关系。

任何更新后的值未通过 WITH CHECK 表达式的行都会导致错误，并且整个命令将被中止。如果只指定了一个 USING 子句，那么该子句将被用于 USING 和 WITH CHECK 两种情况。

通常，UPDATE 命令还需要从待更新关系的列中读取数据（例如在 WHERE 子句、RETURNING 子句，或SET 子句右侧的表达式中）。这种情况下，正在被更新的关系上也需要 SELECT 权限，并且除了 UPDATE 策略外，还会应用适当的 SELECT 或 ALL 策略。这样，用户除了必须通过UPDATE 或 ALL 策略获准更新这些行之外，还必须通过 SELECT 或 ALL 策略访问正在被更新的行。

当 INSERT 命令带有辅助的 ON CONFLICT DO UPDATE 子句时，如果走 UPDATE 路径，则待更新的行会先根据各适用的UPDATE 策略的 USING 表达式进行检查，然后新的更新后行会再根据WITH CHECK 表达式进行检查。但要注意，与独立的UPDATE 命令不同，如果现有行没有通过 USING 表达式检查，就会报错（UPDATE 路径绝不会被静默跳过）。

DELETE

对策略使用 DELETE 意味着该策略适用于 DELETE 命令。DELETE 命令只能看到通过该策略的行。如果某些行不通过 DELETE 策略的 USING 表达式，则它们可能通过SELECT 可见，但不能被删除。

在多数情况下，DELETE 命令也需要从其删除所针对的关系中的列读取数据（例如在 WHERE 子句或 RETURNING 子句中）。这种情况下，该关系上也需要SELECT 权限，并且除了 DELETE 策略外，还会应用适当的 SELECT 或 ALL 策略。这样，用户除了必须通过DELETE 或 ALL 策略获准删除这些行之外，还必须通过 SELECT 或 ALL 策略访问正在被删除的行。

DELETE 策略不能具有 WITH CHECK 表达式，因为它只适用于正在从关系中删除行的情况，所以没有新行需要检查。

表 241. 按命令类型应用的策略

命令	SELECT/ALL策略	INSERT/ALL策略	UPDATE/ALL策略		DELETE/ALL策略
	USING 表达式	WITH CHECK 表达式	USING 表达式	WITH CHECK 表达式	USING 表达式
SELECT	现有行	–	–	–	–
SELECT FOR UPDATE/SHARE	现有行	–	现有行	–	–
INSERT	–	新行	–	–	–
INSERT ... RETURNING	新行[a]	新行	–	–	–
UPDATE	现有 & 新行[a]	–	现有行	新行	–
DELETE	现有行[a]	–	–	–	现有行
ON CONFLICT DO UPDATE	现有 & 新行	–	现有行	新行	–

[a] 如果需要读取现有行或新行（例如引用关系中列的WHERE 或RETURNING 子句）。

多条策略的应用

当不同命令类型的多条策略应用于同一命令时（例如 SELECT 和 UPDATE 策略应用于 UPDATE 命令），用户必须同时具有这两种权限（例如既有从该关系中选取行的权限，也有更新这些行的权限）。因此，一种策略类型的表达式会与另一种策略类型的表达式使用 AND 操作符组合。

当同一命令类型的多条策略应用于同一命令时，必须至少有一条 PERMISSIVE 策略授予对该关系的访问权，并且所有 RESTRICTIVE 策略都必须通过。因此，所有 PERMISSIVE 策略表达式使用 OR 组合，所有RESTRICTIVE 策略表达式使用 AND 组合，然后再将两者的结果使用 AND 组合。如果没有 PERMISSIVE 策略，则访问被拒绝。

请注意，就组合多条策略而言，ALL 策略会被视为与当前正在应用的其他策略同一类型。

例如，对于一个同时需要SELECT 和 UPDATE 权限的 UPDATE 命令，如果这两类策略各自都有多个适用项，它们会按以下方式组合：

```
RESTRICTIVE SELECT/ALL 策略 1 中的 表达式
AND
RESTRICTIVE SELECT/ALL 策略 2 中的 表达式
AND
...
AND
(
  PERMISSIVE SELECT/ALL 策略 1 中的 表达式
  OR
  PERMISSIVE SELECT/ALL 策略 2 中的 表达式
  OR
  ...
)
AND
RESTRICTIVE UPDATE/ALL 策略 1 中的 表达式
AND
RESTRICTIVE UPDATE/ALL 策略 2 中的 表达式
AND
...
AND
(
  PERMISSIVE UPDATE/ALL 策略 1 中的 表达式
  OR
  PERMISSIVE UPDATE/ALL 策略 2 中的 表达式
  OR
  ...
)
```

注解

要为一个表创建或修改策略，你必须是该表的拥有者。

虽然策略会应用于针对数据库中表的显式查询，但当系统执行内部引用完整性检查或验证约束时，并不会应用这些策略。这意味着仍然存在间接判断某个给定值是否存在的方法。一个例子是，尝试向某个主键列或带有唯一约束的列插入重复值。如果插入失败，用户就能推断该值已经存在。（这个例子假定策略允许该用户插入自己无权看见的行。）另一个例子是，用户被允许向一个引用了另一张表的表中插入数据，而被引用的那张表本身对其是隐藏的。用户可以通过向引用表插入值来判断其存在性；插入成功就表示该值存在于被引用表中。要解决这些问题，可以仔细设计策略，防止用户插入、删除或更新那些可能暗示其本来无权看见的值是否存在的行，或者改用生成的值（例如代理键）来代替具有外部含义的键。

通常，为了防止受保护的数据无意间暴露给可能不可信的用户定义函数，系统会在应用用户查询中出现的条件之前，先强制执行安全策略施加的过滤条件。不过，被系统（或系统管理员）标记为 LEAKPROOF 的函数和操作符由于被假定为可信，可以在策略表达式之前求值。

由于策略表达式会被直接添加到用户查询中，因此它们将以执行整个查询的用户权限运行。因此，使用某条策略的用户必须能够访问表达式中引用的所有表或函数，否则在尝试查询启用了行级安全的表时，只会收到权限被拒绝的错误。不过，这并不改变视图的工作方式。与普通查询和

视图一样，被视图引用的表的权限检查和策略将使用视图所有者的权限，以及适用于视图所有者的任何策略。

更多讨论和实际示例见第 5.7 节。

兼容性

`CREATE POLICY` 是一种 PostgreSQL 扩展。

另见

ALTER POLICY, DROP POLICY, ALTER TABLE

CREATE PROCEDURE

CREATE PROCEDURE — 定义一个新过程

大纲

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
```

描述

CREATE PROCEDURE 定义一个新过程。CREATE OR REPLACE PROCEDURE 要么创建一个新过程，要么替换现有定义。要能够定义过程，用户必须对该语言拥有USAGE 权限。

如果包含模式名，则该过程会创建在指定模式中；否则会创建在当前模式中。新过程的名称不能与同一模式中任何输入参数类型相同的现有过程或函数重名。不过，不同参数类型的过程和函数可以共享同一个名称（这称为重载）。

要替换现有过程的当前定义，请使用CREATE OR REPLACE PROCEDURE。不能用这种方式更改过程名称或参数类型（如果这样做，实际上是在创建一个新的不同过程）。

当使用 CREATE OR REPLACE PROCEDURE 替换现有过程时，该过程的所有权和权限不会改变。其他所有过程属性都会被设置为命令中指定或隐含的值。只有拥有该过程才能替换它（这也包括属于拥有者角色的成员）。

创建该过程的用户将成为该过程的拥有者。

要能够创建过程，你必须对参数类型拥有USAGE 权限。

有关编写过程的详细信息，请参阅第 38.4 节。

参数

name

要创建的过程名称（可选模式限定）。

argmode

参数模式：IN、INOUT 或 VARIADIC。如果省略，默认为IN。（目前过程不支持 OUT 参数，请改用INOUT。）

argname

参数的名称。

argtype

过程参数的数据类型（如果有，可选模式限定）。参数类型可以是基础类型、复合类型或域类型，也可以引用表列的类型。

根据实现语言，也可能允许指定诸如cstring 之类的“伪类型”。伪类型表示实际参数类型要么没有被完整指定，要么位于普通 SQL 数据类型集合之外。

可以通过写成 *table_name.column_name*%TYPE 来引用列的类型。使用这一特性有时可以帮助让过程不受表定义变更的影响。

default_expr

在未指定该参数时用作默认值的表达式。该表达式必须能够强制转换为该参数的类型。任何位于带默认值参数之后的输入参数也都必须具有默认值。

lang_name

实现该过程所用语言的名称。可以是 sql、c、internal，也可以是用户定义的过程语言的名称，例如plpgsql。使用单引号括起名称已被弃用，并且要求大小写匹配。

TRANSFORM { FOR TYPE *type_name* } [, ...] }

列出调用该过程时应应用哪些转换。转换负责在 SQL 类型与语言特定数据类型之间进行转换；详见 CREATE TRANSFORM。过程语言实现通常对内置类型有硬编码知识，因此无须在此列出。如果某种过程语言实现不知道如何处理某个类型，且又没有提供转换，则会退回到默认的数据类型转换行为，但具体行为取决于该实现。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER 表示该过程将以调用它的用户的权限执行。这是默认值。
SECURITY DEFINER 指定该过程将以拥有它的用户的权限执行。

关键字 EXTERNAL 是为了符合 SQL 而允许使用的，但它是可选的，因为与 SQL 不同，此特性适用于所有过程，而不只是外部过程。

SECURITY DEFINER 过程不能执行事务控制语句（例如 COMMIT 和 ROLLBACK，具体取决于语言）。

configuration_parameter
value

SET 子句会在进入该过程时把指定的配置参数设置为指定值，并在过程退出时恢复为之前的值。SET FROM CURRENT 会把执行 CREATE PROCEDURE 时该参数的当前值保存为进入过程时要应用的值。

如果为过程附加了 SET 子句，那么在该过程中针对同一变量执行的SET LOCAL 命令，其效果会被限制在该过程内：过程退出时，该配置参数的先前值仍会被恢复。不过，普通的SET 命令（不带 LOCAL）会覆盖这个 SET 子句，就像它会覆盖先前的SET LOCAL 命令一样：这种命令的效果会在过程退出后持续存在，除非当前事务被回滚。

如果为过程附加了 SET 子句，则该过程不能执行事务控制语句（例如COMMIT 和 ROLLBACK，具体取决于语言）。

关于允许的参数名和值的更多信息，见SET 和第 19 章。

definition

定义该过程的字符串常量，其含义取决于所用语言。它可以是内部过程名称、对象文件路径、SQL 命令，或过程语言中的文本。

通常，使用美元引用（见第 4.1.2.4 节）来书写过程定义字符串，会比普通单引号语法更方便。不使用美元引用时，过程定义中的任何单引号或反斜线都必须通过双写进行转义。

obj_file, *link_symbol*

当 C 语言源代码中的过程名称与 SQL 过程名称不同时，这种形式的AS 子句用于可动态加载的 C 语言过程。字符串 *obj_file* 是包含已编译 C 过程的共享库文件名，其解释方式与 LOAD 命令相同。字符串 *link_symbol* 是过程的链接符号，即 C 语言源代码中的过程名称。如果省略链接符号，则假定它与正在定义的 SQL 过程名称相同。

当重复调用的 CREATE PROCEDURE 引用同一个对象文件时，该文件在每个会话中只会被加载一次。要卸载并重新加载该文件（例如在开发期间），需要启动一个新会话。

注解

有关函数创建且同样适用于过程的更多细节，请参见 CREATE FUNCTION。

使用 CALL 执行过程。

示例

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
AS $$
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;

CALL insert_data(1, 2);
```

兼容性

CREATE PROCEDURE 命令在 SQL 标准中定义。PostgreSQL 版本与其相似，但并不完全兼容。详情另见 CREATE FUNCTION。

另见

ALTER PROCEDURE, DROP PROCEDURE, CALL, CREATE FUNCTION

CREATE PUBLICATION

CREATE PUBLICATION — 定义一个新的发布

大纲

```
CREATE PUBLICATION name
    [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
      | FOR ALL TABLES ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]
```

描述

CREATE PUBLICATION 向当前数据库中添加一个新发布。该发布的名称必须不同于当前数据库中任何现有发布的名称。

发布本质上是一组表，其数据更改会通过逻辑复制进行复制。有关发布在逻辑复制配置中如何工作的详细信息，请参见第 31.1 节。

参数

name

新发布的名称。

FOR TABLE

指定要添加到发布中的表列表。如果在表名之前指定 ONLY，则仅将该表添加到发布中。如果未指定ONLY，则将该表及其所有后代表（如果有）添加到发布中。也可以在表名后指定*，以明确表示包含后代表。

只有持久的基表可以成为发布的一部分。临时表、不记录 WAL 的表、外部表、物化视图、普通视图和分区表不能成为发布的一部分。要复制分区表，请将各个分区添加到发布中。

FOR ALL TABLES

将发布标记为复制数据库中所有表（包括将来创建的表）的更改。

WITH (*publication_parameter* [= *value*] [, ...])

此子句指定发布的可选参数。支持以下参数：

publish (string)

此参数确定新发布将向订阅者发布哪些 DML 操作。该值是以逗号分隔的操作列表。允许的操作包括 insert、update、delete 和 truncate。默认发布所有操作，因此此选项的默认值为'insert, update, delete, truncate'。

注解

如果既未指定 FOR TABLE 也未指定 FOR ALL TABLES，则该发布开始时不包含任何表。这对于稍后添加表很有用。

创建发布不会启动复制。它只为未来的订阅者定义分组和过滤逻辑。

要创建发布，调用用户必须具有当前数据库的CREATE 权限。（当然，超级用户会绕过这一检查。）

要向发布中添加表，调用用户必须拥有该表。FOR ALL TABLES 子句要求调用用户是超级用户。

如果某个发布发布 UPDATE 和/或 DELETE 操作，则加入该发布的表必须已经定义 REPLICA IDENTITY。否则，在这些表上将不允许执行这些操作。

对于 INSERT ... ON CONFLICT 命令，发布会发布该命令实际产生的操作。根据执行结果，它可能被发布为INSERT 或 UPDATE，也可能根本不被发布。

COPY ... FROM 命令会作为 INSERT 操作发布。

DDL 操作不会被发布。

示例

创建一个发布，发布两个表中的所有更改：

```
CREATE PUBLICATION mypublication FOR TABLE users, departments;
```

创建一个发布，发布所有表中的所有更改：

```
CREATE PUBLICATION alltables FOR ALL TABLES;
```

创建一个只发布 INSERT 操作、且只针对一个表的发布：

```
CREATE PUBLICATION insert_only FOR TABLE mydata
    WITH (publish = 'insert');
```

兼容性

CREATE PUBLICATION 是 PostgreSQL 的扩展。

另见

ALTER PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

CREATE ROLE

CREATE ROLE — 定义一个新的数据库角色

大纲

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]
```

其中 *option* 可以是：

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

描述

`CREATE ROLE` 向 PostgreSQL 数据库集簇中新增一个角色。角色是一种可以拥有数据库对象并拥有数据库权限的实体；根据其使用方式，它既可以视为“用户”，也可以视为“组”，或者兼具这两种身份。有关用户管理和认证的信息，请参见第 21 章和第 20 章。要使用该命令，你必须拥有 `CREATEROLE` 权限，或者是数据库超级用户。

注意，角色是在数据库集簇级别定义的，因此在该集簇中的所有数据库里都有效。

参数

name

新角色的名称。

`SUPERUSER`
`NOSUPERUSER`

这些子句决定新角色是否为“超级用户”；超级用户可以绕过数据库中的所有访问限制。超级用户身份具有危险性，只有在确有需要时才应使用。要创建新的超级用户，你自己必须先是超级用户。若未指定，默认值是 `NOSUPERUSER`。

CREATEDB
NOCREATEDB

这些子句定义角色创建数据库的能力。如果指定 CREATEDB，则允许正在定义的角色创建新数据库。指定NOCREATEDB 会禁止角色创建数据库。如果未指定，默认值为NOCREATEDB。

CREATEROLE
NOCREATEROLE

这些子句决定某个角色是否被允许创建、更改、删除其他角色，为其添加注释，更改其安全标签，以及授予或撤销其中的成员资格。关于该权限赋予的具体能力，参见创建角色。若未指定，默认值是 NOCREATEROLE。

INHERIT
NOINHERIT

这些子句决定角色是否“继承”其所属角色的权限。具有 INHERIT 属性的角色可以自动使用直接或间接所属的所有角色获得的任何数据库权限。没有 INHERIT 时，成为另一个角色的成员只授予对该角色执行SET ROLE 的能力；另一个角色的权限只有在执行该操作后才可用。如果未指定，默认值为INHERIT。

LOGIN
NOLOGIN

这些子句决定是否允许角色登录；也就是说，在客户端连接期间，是否可以将该角色作为初始会话授权名称。具有LOGIN 属性的角色可以视为用户。没有该属性的角色可用于管理数据库权限，但通常意义上不属于用户。如果未指定，默认值为NOLOGIN，但通过其另一种拼写 CREATE USER 调用 CREATE ROLE 时除外。

REPLICATION
NOREPLICATION

这些子句决定角色是否为复制角色。角色必须具有此属性（或是超级用户），才能以复制模式（物理复制或逻辑复制）连接到服务器，以及创建或删除复制槽。具有REPLICATION 属性的角色权限非常高，因此该属性只应赋予实际用于复制的角色。如果未指定，默认值为 NOREPLICATION。要创建具有 REPLICATION 属性的新角色，必须是超级用户。

BYPASSRLS
NOBYPASSRLS

这些子句决定角色是否绕过每个行级安全（RLS）策略。NOBYPASSRLS 是默认值。要创建具有 BYPASSRLS 属性的新角色，必须是超级用户。

注意，pg_dump 默认会把row_security 设置为 OFF，以确保能够转储表中的全部内容。如果运行 pg_dump 的用户没有适当权限，将返回错误。不过，超级用户和被转储表的拥有者总是会绕过 RLS。

CONNECTION LIMIT *connlimit*

如果该角色可以登录，此项指定该角色可建立多少并发连接。-1（默认值）表示不限制。注意，只有普通连接会计入此限制；预备事务和后台工作进程连接都不会计入此限制。

[ENCRYPTED] PASSWORD '*password*'
PASSWORD NULL

设置角色的密码。（密码只对具有 LOGIN 属性的角色有用，但即使没有该属性，你仍然可以为角色定义密码。）如果你不打算使用密码认证，可以省略此选项。如果未指定密码，密码将被设为 null，并且该用户的密码认证将始终失败。也可以显式写为PASSWORD NULL。

> **注意**
>
> 指定空字符串也会把密码设为 null，但在 PostgreSQL 10 之前并非如此。在更早的版本中，是否可以使用空字符串取决于认证方法和具体版本，而 libpq 在任何情况下都拒绝使用它。为避免歧义，应避免指定空字符串。

密码始终以加密形式存储在系统目录中。ENCRYPTED 关键字没有实际效果，但为了向后兼容仍会被接受。加密方式由配置参数password_encryption 决定。如果给出的密码字符串已经是 MD5 加密或 SCRAM 加密格式，那么无论password_encryption 的值是什么，它都会按原样存储（因为系统无法解密指定的已加密密码字符串，再将其改用其他格式加密）。这样便可在转储/恢复期间重新装载已加密的密码。

VALID UNTIL '*timestamp*'

VALID UNTIL 子句设置一个日期和时间，在此之后角色的密码将不再有效。如果省略此子句，密码将永久有效。

IN ROLE *role_name*

IN ROLE 子句列出一个或多个现有角色，新角色会立即被添加为这些角色的新成员。（注意，没有选项能将新角色添加为管理员；请另行使用GRANT 命令执行此操作。）

IN GROUP *role_name*

IN GROUP 是 IN ROLE 的过时拼写。

ROLE *role_name*

ROLE 子句列出一个或多个现有角色，它们会被自动添加为新角色的成员。（这实际上使新角色成为一个“组”。）

ADMIN *role_name*

ADMIN 子句与 ROLE 类似，但所列角色会以WITH ADMIN OPTION 加入新角色，从而有权向其他人授予此角色的成员资格。

USER *role_name*

USER 子句是 ROLE 子句的过时拼写。

SYSID *uid*

SYSID 子句会被忽略，但为了向后兼容，仍然接受它。

注解

使用 ALTER ROLE 可更改角色属性，使用DROP ROLE 可删除角色。通过 CREATE ROLE 指定的所有属性都可以在后续的 ALTER ROLE 命令中修改。

向作为组使用的角色添加和移除成员，推荐使用GRANT 和 REVOKE。

VALID UNTIL 子句只为密码定义过期时间，而不是为角色本身定义过期时间。特别是，使用非基于密码的认证方式登录时，不会强制执行该过期时间。

INHERIT 属性控制可授予权限的继承（即数据库对象的访问权限和角色成员资格）。它不适用于通过 CREATE ROLE 和 ALTER ROLE 设置的特殊角色属性。例如，成为具有CREATEDB 权限的角色的成员，并不会立即获得创建数据库的能力，即使设置了INHERIT 也是如此；必须先通过 SET ROLE 成为该角色，才能创建数据库。

之所以默认采用 INHERIT 属性，是出于向后兼容的考虑：在较早版本的PostgreSQL 中，用户总能访问其所属组的全部权限。不过，NOINHERIT 更接近 SQL 标准规定的语义。

请谨慎对待 CREATEROLE 权限。对于具有 CREATEROLE 的角色的权限，没有继承的概念。这意味着，即使某个角色不具有某项权限，只要允许它创建其他角色，它就可以轻易创建权限与自身不同的另一个角色（但不能创建具有超级用户权限的角色）。例如，如果角色“user”具有 CREATEROLE 权限而没有 CREATEDB 权限，它仍然可以创建具有CREATEDB 权限的新角色。因此，应将具有CREATEROLE 权限的角色视为近乎超级用户的角色。

PostgreSQL 包含一个程序 createuser，其功能与CREATE ROLE 相同（实际上它就是调用此命令），但它可以从命令 shell 中运行。

CONNECTION LIMIT 选项只是近似地被强制执行；如果两个新会话在几乎同一时刻启动，而此时该角色只剩下一个连接“槽”，则两者都可能失败。此外，该限制从不对超级用户强制执行。

用此命令指定未加密密码时必须谨慎。密码会以明文形式传输到服务器，也可能被记录在客户端命令历史或服务器日志中。而命令createuser 传输的是加密密码。此外，psql 包含 \password 命令，可用于稍后安全地更改密码。

示例

创建一个可登录但没有密码的角色：

```
CREATE ROLE jonathan LOGIN;
```

创建一个带密码的角色：

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

（CREATE USER 与 CREATE ROLE 相同，只是它隐含指定了LOGIN。）

创建一个密码有效期截止到 2004 年底的角色。当 2005 年的第一秒过去后，该密码就不再有效。

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

创建一个可以创建数据库并管理角色的角色：

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

兼容性

SQL 标准包含 `CREATE ROLE` 语句，但标准只要求如下语法：

```
CREATE ROLE name [ WITH ADMIN role_name ]
```

多个初始管理员以及 `CREATE ROLE` 的所有其他选项，都是 PostgreSQL 扩展。

SQL 标准定义了用户和角色这两个概念，但将它们视为彼此不同的概念，并把所有用于定义用户的命令留给各个数据库实现自行规定。在 PostgreSQL 中，我们选择将用户和角色统一为同一类实体。因此，角色拥有的可选属性比标准中更多。

将用户赋予 `NOINHERIT` 属性，而将角色赋予 `INHERIT` 属性，最接近 SQL 标准规定的行为。

另见

SET ROLE, ALTER ROLE, DROP ROLE, GRANT, REVOKE, createuser

CREATE RULE

CREATE RULE — 定义一条新的重写规则

大纲

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table_name [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

其中 *event* 可以是以下之一：

```
    SELECT | INSERT | UPDATE | DELETE
```

描述

`CREATE RULE` 定义一条应用于指定表或视图的新规则。`CREATE OR REPLACE RULE` 将创建一条新规则，或者替换同一个表上同名的现有规则。

PostgreSQL 规则系统允许定义在数据库表上执行插入、更新或删除时的替代动作。粗略地说，当在某个给定表上执行某个命令时，一条规则会导致额外的命令被执行。另一种情况是，`INSTEAD` 规则可以用另一个命令替换给定命令，或者让某个命令完全不执行。规则也用于实现 SQL 视图。必须认识到，规则实际上是一种命令转换机制，或者说是命令宏。这种转换发生在命令开始执行之前。如果你真正需要的是对每个物理行独立触发的操作，那么你很可能应该使用触发器，而不是规则。有关规则系统的更多信息见第 41 章。

目前，`ON SELECT` 规则只能附加到视图上。（将这种规则附加到表上会将该表转换为视图。）这样的规则必须命名为`"_RETURN"`，必须是无条件的`INSTEAD` 规则，并且必须有一个由单个 `SELECT` 命令组成的动作。该命令定义了视图中可见的内容。（视图本身基本上是一个没有存储的虚拟表。）最好把这样的规则视为实现细节。尽管可以通过 `CREATE OR REPLACE RULE "_RETURN" AS ...`重新定义视图，但风格上更好的做法是使用 `CREATE OR REPLACE VIEW`。

可以通过定义 `ON INSERT`、`ON UPDATE` 和 `ON DELETE` 规则（或其中足以满足你需求的任意子集），来营造出可更新视图的效果，从而将视图上的更新动作替换为对其他表的适当更新。如果想支持`INSERT RETURNING` 等功能，那么务必在这些规则中的每一条里都放入合适的 `RETURNING` 子句。

如果你尝试对复杂视图更新使用有条件规则，这里有一个陷阱：对于你希望允许在该视图上执行的每一种动作，必须都有一条无条件的 `INSTEAD` 规则。如果该规则是有条件的，或者不是 `INSTEAD` 规则，那么系统仍会拒绝执行该更新动作，因为它认为在某些情况下最终可能还是会尝试在该视图的虚拟表上执行该动作。如果你想用有条件规则处理所有有用的情况，可以加上一条无条件的 `DO INSTEAD NOTHING` 规则，以确保系统明白它永远不会被要求去更新这个虚拟表。然后，把这些有条件规则改成非 `INSTEAD` 规则；在它们适用的情况下，它们会附加在默认的 `INSTEAD NOTHING` 动作之上。（不过，这种方法当前仍无法支持 `RETURNING` 查询。）

注意

足够简单、可自动更新的视图（见 CREATE VIEW）不需要依靠用户创建的规则来实现更新。虽然你仍然可以显式创建规则，但自动更新转换通常比显式规则性能更好。

另一种值得考虑的办法是使用 INSTEAD OF 触发器（见 CREATE TRIGGER）代替规则。

参数

name

要创建的规则的名称。它必须与同一个表上任何其他规则的名称相区分。同一个表上同一种事件类型的多条规则会按名称的字母顺序应用。

event

事件是 SELECT、INSERT、UPDATE 或者 DELETE 之一。注意，包含ON CONFLICT 子句的 INSERT 不能用于带有 INSERT 或 UPDATE 规则的表。请考虑改用可更新的视图。

table_name

规则适用的表或者视图的名称（可以是模式限定的）。

condition

任意的 SQL 条件表达式（返回 boolean）。条件表达式不能引用 NEW 和 OLD 之外的任何表，并且不能包含聚合函数。

INSTEAD

INSTEAD 表示这些命令应当替代原始命令执行。

ALSO

ALSO 表示这些命令应当在原始命令之外执行。

如果既未指定 ALSO，也未指定 INSTEAD，则默认值为ALSO。

command

组成规则动作的命令。可用的命令有 SELECT、INSERT、UPDATE、DELETE 或者 NOTIFY。

在 *condition* 和 *command* 中，可以使用特殊表名 NEW 和 OLD 来引用目标表中的值。在 ON INSERT 和 ON UPDATE 规则中，NEW 可用于引用待插入或待更新的新行。在 ON UPDATE 和 ON DELETE 规则中，OLD 可用于引用将被更新或删除的现有行。

注解

要在表上创建或者修改规则，必须是表的拥有者。

在视图上的 INSERT、UPDATE 或 DELETE 规则中，可以添加一个输出视图各列的 RETURNING 子句。如果该规则分别由 INSERT RETURNING、UPDATE RETURNING 或 DELETE RETURNING 命令触发，那么这个子句将用于计算输出。当规则由不带RETURNING 的命令触发时，该规则的 RETURNING 子句会被忽略。当前实现只允许无条件的 INSTEAD 规则包含 RETURNING；此外，对于同一事件的所有规则，总共最多只能有一个RETURNING 子句。（这可以确保只有一个候选 RETURNING 子句可用于计算结果。）如果任何可用规则中都没有 RETURNING 子句，那么视图上的RETURNING 查询将被拒绝。

避免循环规则非常重要。例如，尽管下面的两条规则定义都被 PostgreSQL 所接受，SELECT 命令将导致 PostgreSQL 报告错误，因为规则会发生递归展开：

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
        SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
        SELECT * FROM t1;

SELECT * FROM t1;
```

当前，如果一个规则动作包含一个NOTIFY 命令，该NOTIFY 命令将被无条件执行，也就是说，即使没有任何应当应用该规则的行，也会发出NOTIFY。例如：

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

在 UPDATE 期间会发送一个 NOTIFY 事件，无论是否存在匹配条件id = 42 的行。这是一个实现限制，未来版本中可能会修复。

兼容性

CREATE RULE 是一种 PostgreSQL 语言扩展，整个查询重写系统也是如此。

另见

ALTER RULE, DROP RULE

CREATE SCHEMA

CREATE SCHEMA — 定义一个新模式

大纲

```
CREATE SCHEMA schema_name [ AUTHORIZATION role_specification ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION role_specification [ schema_element [ ... ] ]
CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION role_specification ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION role_specification
```

其中 *role_specification* 可以是:

```
    user_name
  | CURRENT_USER
  | SESSION_USER
```

描述

CREATE SCHEMA 在当前数据库中创建一个新模式。该模式名必须与当前数据库中任何现有模式的名称不同。

模式本质上是一个命名空间：它包含具名对象（表、数据类型、函数和操作符），这些对象的名称可以与其他模式中的对象重名。访问具名对象时，可以在其名称前加上模式名作为前缀来“限定”名称，或者设置一个包含所需模式的搜索路径。指定非限定对象名的CREATE 命令会在当前模式中创建该对象（即搜索路径最前面的模式，可通过函数current_schema 确定）。

CREATE SCHEMA 还可以选择包含子命令，以便在新模式中创建对象。这些子命令基本上会被当作在创建模式之后单独发出的命令来处理。不过，如果使用了AUTHORIZATION 子句，则所有创建的对象都将归该用户所有。

参数

schema_name

要创建的模式名称。如果省略，*user_name* 将被用作模式名。该名称不能以 pg_开头，因为这类名称保留给系统模式。

user_name

将拥有新模式的用户的角色名。如果省略，则默认为执行该命令的用户。要创建由另一个角色拥有的模式，你必须是该角色的直接或间接成员，或者是超级用户。

schema_element

定义要在该模式中创建的对象的 SQL 语句。当前，只有CREATE TABLE、CREATE VIEW、CREATE INDEX、CREATE SEQUENCE、CREATE TRIGGER 和 GRANT 可作为 CREATE SCHEMA 中的子句使用。其他类型的对象可以在模式创建之后通过单独的命令创建。

`IF NOT EXISTS`

如果同名模式已经存在，则不执行任何操作（但会发出一条提示）。使用该选项时不能包含 *`schema_element`* 子命令。

注解

要创建一个模式，执行该命令的用户必须拥有当前数据库的`CREATE` 权限（当然，超级用户可以绕过这项检查）。

示例

创建一个模式：

```
CREATE SCHEMA myschema;
```

为用户 `joe` 创建一个模式，该模式也将被命名为 `joe`：

```
CREATE SCHEMA AUTHORIZATION joe;
```

创建一个被用户 `joe` 拥有的名为 `test` 的模式，除非已经有一个名为`test` 的模式（不管 `joe` 是否拥有那个预先存在的模式）。

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

创建一个模式并且在其中创建一个表和视图：

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
```

请注意，各个子命令末尾都不带分号。

下面是实现相同结果的一种等效写法：

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

兼容性

SQL 标准允许在 `CREATE SCHEMA` 中使用 `DEFAULT CHARACTER SET` 子句，并允许使用比 PostgreSQL 当前所接受的更多子命令类型。

SQL 标准规定，`CREATE SCHEMA` 中的子命令可以按任意顺序出现。当前的 PostgreSQL 实现并不能处理子命令中所有前向引用的情况；有时可能需要重新排列子命令的顺序，以避免前向引用。

根据 SQL 标准，模式的拥有者总是拥有其中的所有对象。PostgreSQL 允许模式包含由模式拥有者之外的用户所拥有的对象。只有当模式拥有者把其模式上的 `CREATE` 权限授予其他人，或者超级用户选择在该模式中创建对象时，才会发生这种情况。

`IF NOT EXISTS` 选项是一种 PostgreSQL 扩展。

另见

ALTER SCHEMA, DROP SCHEMA

CREATE SEQUENCE

CREATE SEQUENCE — 定义一个新的序列发生器

大纲

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

描述

CREATE SEQUENCE 创建一个新的序列号发生器。这会创建并初始化一个名为 *name* 的特殊单行表。该发生器归发出该命令的用户所有。

如果给定了模式名，则序列会在指定模式中创建。否则，它会在当前模式中创建。临时序列存在于一个特殊模式中，因此创建临时序列时不能指定模式名。序列名必须不同于同一模式中任何其他关系（表、序列、索引、视图、物化视图或外部表）的名称。

创建序列后，可以使用函数 nextval、currval 以及 setval 来操作该序列。这些函数的文档见第 9.16 节。

虽然不能直接更新序列，但可以使用如下查询：

```
SELECT * FROM name;
```

来查看序列的参数和当前状态。特别是，序列的 last_value 字段会显示任一会话最近分配的值。（当然，如果其他会话正在调用nextval，那么到该值被打印出来时，它可能已经过时。）

参数

TEMPORARY 或 TEMP

如果指定该项，则该序列对象只为当前会话创建，并会在会话退出时自动删除。临时序列存在期间，已有的同名永久序列在本会话中不可见，除非使用模式限定名来引用它们。

IF NOT EXISTS

如果同名关系已经存在，则不会抛出错误，而是发出一条提示。请注意，不能保证现有关系与本来要创建的序列有任何相似之处——它甚至可能根本不是序列。

name

如果同名关系已经存在，则不抛出错误。在这种情况下会发出一个提示。注意，这并不保证现有关系与本应创建出的序列有任何相似之处 – 它甚至可能根本不是序列。

data_type

可选子句 AS *data_type* 指定序列的数据类型。有效类型为 smallint、integer 和 bigint。默认为 bigint。数据类型决定序列的默认最小值和最大值。

increment

可选子句 INCREMENT BY *increment* 指定加到当前序列值上以生成新值的增量。正值使序列递增，负值使序列递减。默认值为 1。

minvalue
NO MINVALUE

可选子句 MINVALUE *minvalue* 确定序列可生成的最小值。如果未提供此子句，或者指定了 NO MINVALUE，则使用默认值。升序序列的默认值为 1。降序序列的默认值为该数据类型的最小值。

maxvalue
NO MAXVALUE

可选子句 MAXVALUE *maxvalue* 确定序列的最大值。如果未提供此子句，或者指定了 NO MAXVALUE，则使用默认值。升序序列的默认值是该数据类型的最大值。降序序列的默认值是 -1。

start

可选子句 START WITH *start* 允许序列从任意值开始。升序序列的默认起始值为 *minvalue*，降序序列则为*maxvalue*。

cache

可选子句 CACHE *cache* 指定要预分配并存储在内存中以加快访问的序列号数量。最小值为 1（一次只能生成一个值，即不缓存），这也是默认值。

CYCLE
NO CYCLE

CYCLE 选项允许序列在升序序列达到 *maxvalue*、或降序序列达到*minvalue* 时分别回绕。如果达到该限制，下一个生成的数字将分别是*minvalue* 或 *maxvalue*。

如果指定 NO CYCLE，则当序列达到其限值后，任何对 nextval 的调用都会返回错误。如果既未指定 CYCLE 也未指定 NO CYCLE，则默认使用NO CYCLE。

OWNED BY *table_name.column_name*
OWNED BY NONE

OWNED BY 选项使序列与特定表列关联，从而在删除该列（或整个表）时，也自动删除该序列。指定表必须与序列具有相同的所有者，并且位于同一模式中。默认值OWNED BY NONE 表示不存在这种关联。

注解

使用 DROP SEQUENCE 删除序列。

序列基于 bigint 算术，因此其范围不能超过八字节整数的范围（-9223372036854775808 到 9223372036854775807）。

由于 nextval 和 setval 调用从不回滚，如果需要对序列号进行“无间隙”分配，就不能使用序列对象。可以通过对包含计数器的表加排他锁来构造无间隙分配；但这种方案比序列对象昂贵得多，尤其是在许多事务需要并发获取序列号时。

如果对一个将由多个会话并发使用的序列对象设置了大于 1 的 *cache*，则可能得到意想不到的结果。每个会话会在一次访问该序列对象时分配并缓存连续的序列值，并相应增加该序列对象的 last_value。随后，该会话中接下来的 *cache*-1 次 nextval 调用会直接返回预分配的值，而不接触序列对象。因此，任何在会话中已分配但未使用的数字都会在会话结束时丢失，从而在序列中造成“空洞”。

此外，虽然多个会话保证会分配到互不相同的序列值，但从所有会话整体来看，这些值的生成顺序可能会乱序。例如，当*cache* 设置为 10 时，会话 A 可能保留值 1..10 并返回nextval=1；然后会话 B 可能保留值 11..20 并返回 nextval=11，而此时会话 A 尚未生成 nextval=2。因此，当*cache* 设置为 1 时，可以安全地假定 nextval 值按顺序生成；当 *cache* 设置大于 1 时，只应假定 nextval 值彼此不同，而不应假定它们严格按顺序生成。此外，last_value 会反映任一会话最近保留的值，不管该值是否已经由nextval 返回。

另一个需要注意的问题是，在这种序列上执行的setval 不会立刻被其他会话察觉，直到它们用完自己缓存的所有预分配值为止。

示例

创建一个名为 serial 的升序序列，从 101 开始：

```
CREATE SEQUENCE serial START 101;
```

从该序列中取出下一个数字：

```
SELECT nextval('serial');

 nextval
---------
     101
```

从该序列中取出下一个数字：

```
SELECT nextval('serial');

 nextval
---------
     102
```

在 INSERT 命令中使用该序列：

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

在一次 COPY FROM 之后更新序列值：

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

兼容性

CREATE SEQUENCE 符合 SQL 标准，但下列情况除外：

- 获取下一个值是使用 nextval() 函数，而不是标准的 NEXT VALUE FOR 表达式完成的。
- OWNED BY 子句是 PostgreSQL 扩展。

另见

ALTER SEQUENCE, DROP SEQUENCE

CREATE SERVER

CREATE SERVER — 定义一个新的外部服务器

大纲

```
CREATE SERVER [ IF NOT EXISTS ] server_name [ TYPE 'server_type' ] [ VERSION 'server_version' ]
    FOREIGN DATA WRAPPER fdw_name
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

描述

`CREATE SERVER` 定义一个新的外部服务器。定义该服务器的用户将成为其拥有者。

外部服务器通常封装外部数据包装器用来访问外部数据资源的连接信息。还可以通过用户映射指定额外的、特定于用户的连接信息。

服务器名称在数据库内必须唯一。

要创建服务器，必须在所使用的外部数据包装器上具有`USAGE` 权限。

参数

`IF NOT EXISTS`

如果同名服务器已存在，则不要抛出错误。在这种情况下会发出一个通知。请注意，不能保证现有服务器与原本将要创建的服务器有任何相似之处。

`server_name`

要创建的外部服务器的名称。

`server_type`

可选的服务器类型，可能对外部数据包装器有用。

`server_version`

可选的服务器版本，可能对外部数据包装器有用。

`fdw_name`

管理该服务器的外部数据包装器的名称。

`OPTIONS ( option 'value' [, ... ] )`

这个子句指定服务器的选项。这些选项通常定义该服务器的连接细节，但实际的名称和值取决于该服务器的外部数据包装器。

注解

在使用 dblink 模块时，可以将外部服务器的名称用作dblink_connect 函数的一个参数，以指示连接参数。要以这种方式使用它，必须在该外部服务器上具有 USAGE 权限。

如果外部服务器支持排序下推，那么它必须与本地服务器具有相同的排序顺序。

示例

创建一个使用外部数据包装器 postgres_fdw 的服务器 myserver：

```
CREATE SERVER myserver FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'foo', dbname 'foodb',
 port '5432');
```

详见 postgres_fdw。

兼容性

CREATE SERVER 符合 ISO/IEC 9075-9 (SQL/MED)。

另见

ALTER SERVER, DROP SERVER, CREATE FOREIGN DATA WRAPPER, CREATE FOREIGN TABLE, CREATE USER MAPPING

CREATE STATISTICS

CREATE STATISTICS — 定义扩展统计信息

大纲

```
CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    [ ( statistics_kind [, ... ] ) ]
    ON column_name, column_name [, ...]
    FROM table_name
```

描述

`CREATE STATISTICS` 创建一个新的扩展统计对象，用于跟踪指定表、外部表或物化视图的数据。该统计对象会在当前数据库中创建，并由发出该命令的用户拥有。

如果给定了模式名（比如，`CREATE STATISTICS myschema.mystat ...`），则在指定的模式中创建该统计对象。否则，在当前模式中创建。该统计对象的名称必须与同一模式中的任何其他统计对象名称不同。

参数

`IF NOT EXISTS`

如果同名统计对象已经存在，则不会抛出错误，而是发出一条提示。注意，这里只考虑统计对象的名称，而不考虑其定义细节。

`statistics_name`

要创建的统计对象名称（可选模式限定）。

`statistics_kind`

要在此统计对象中计算的一种统计信息。目前支持的种类有：`ndistinct`，启用非重复值统计信息；以及`dependencies`，启用函数依赖统计。如果省略此子句，统计对象将包含所有受支持的统计信息种类。更多信息请参见第 14.2.2 节和第 71.2 节。

`column_name`

所计算统计信息要涵盖的表列名称。必须至少给出两个列名。

`table_name`

包含用于计算统计信息的列的表名称（可选模式限定）；关于继承和分区的处理方式，参见 ANALYZE。

注解

必须是表的所有者，才能创建读取该表数据的统计对象。不过，一旦创建，统计对象的所有权就独立于底层表。

扩展统计信息目前尚未被规划器用于表连接的选择率估计。这个限制很可能会在未来版本的 PostgreSQL 中被移除。

示例

创建表 `t1`，其中两列存在函数依赖关系，也就是说，知道第一列中的某个值就足以确定另一列中的值。然后，在这些列上构建函数依赖统计信息：

```
CREATE TABLE t1 (
    a   int,
    b   int
);

INSERT INTO t1 SELECT i/100, i/500
                 FROM generate_series(1,1000000) s(i);

ANALYZE t1;

-- 匹配行的数量将被大大低估：
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;

ANALYZE t1;

-- 现在行计数估计会更准确：
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);
```

如果没有函数依赖统计信息，规划器会认为两个 `WHERE` 条件彼此独立，并将它们的选择率相乘，从而得到远小于实际值的行数估计。有了这类统计信息后，规划器就能识别出 `WHERE` 条件存在冗余，因此不会低估行数。

兼容性

SQL 标准中没有 `CREATE STATISTICS` 命令。

另见

ALTER STATISTICS, DROP STATISTICS

CREATE SUBSCRIPTION

CREATE SUBSCRIPTION — 定义一个新的订阅

大纲

```
CREATE SUBSCRIPTION subscription_name
    CONNECTION 'conninfo'
    PUBLICATION publication_name [, ...]
    [ WITH ( subscription_parameter [= value] [, ... ] ) ]
```

描述

`CREATE SUBSCRIPTION` 为当前数据库添加一个新订阅。订阅名称必须与数据库中任何现有订阅的名称不同。

订阅表示与发布者的复制连接。因此，该命令不仅会在本地系统目录中添加定义，还会在发布者上创建一个复制槽。

在执行该命令所在事务提交时，会启动一个逻辑复制工作进程，为新订阅复制数据。

关于订阅以及逻辑复制整体的更多信息，见第 31.2 节和第 31 章。

参数

subscription_name

新订阅的名称。

CONNECTION '*conninfo*'

连接到发布者的连接字符串。详情请参见第 34.1.1 节。

PUBLICATION *publication_name*

要订阅的发布者上的发布名称。

WITH (*subscription_parameter* [= *value*] [, ...])

此子句指定订阅的可选参数。支持以下参数：

copy_data (boolean)

指定复制开始后，是否复制所订阅发布中已经存在的数据。默认值为`true`。

create_slot (boolean)

指定该命令是否应在发布者上创建复制槽。默认值为 `true`。

enabled (boolean)

指定订阅是应当主动进行复制，还是仅完成设置但暂不启动。默认值为true。

slot_name (string)

要使用的复制槽名称。默认行为是使用订阅名称作为槽名。

当 slot_name 设为 NONE 时，该订阅将不关联任何复制槽。当复制槽将在稍后手工创建时，可以使用这种方式。这类订阅还必须同时将 enabled 和 create_slot 设为 false。

synchronous_commit (enum)

此参数的值会覆盖 synchronous_commit 设置。默认值为 off。

对逻辑复制来说，使用off 是安全的：如果订阅者因缺少同步而丢失了事务，数据会再次从发布者发送过来。

在进行同步逻辑复制时，可能更适合使用不同的设置。逻辑复制工作进程会向发布者报告写入和刷盘位置，而在使用同步复制时，发布者会等待真正的刷盘完成。这意味着，当订阅被用于同步复制时，将订阅者的 synchronous_commit 设为 off 可能会增加发布者上 COMMIT 的延迟。在这种场景下，将 synchronous_commit 设为 local 或更高可能更有利。

connect (boolean)

指定 CREATE SUBSCRIPTION 是否连接到发布者。将其设为 false 会把 enabled、create_slot 和 copy_data 的默认值改为 false。

不允许将 connect 设为 false，同时又把 enabled、create_slot 或 copy_data 设为 true。

由于该选项设为 false 时不会建立连接，因此表不会被订阅，所以即使之后启用了订阅，也不会复制任何内容。必须执行 ALTER SUBSCRIPTION ... REFRESH PUBLICATION 才会订阅这些表。

注意

关于如何在订阅与发布实例之间配置访问控制，详见第 31.7 节。

当创建复制槽时（这是默认行为），CREATE SUBSCRIPTION 不能在事务块内部执行。

创建一个连接到同一数据库集簇的订阅（例如在同一集簇中的不同数据库之间复制，或者在同一数据库内复制）时，只有在复制槽不作为同一命令的一部分创建的情况下才会成功。否则，CREATE SUBSCRIPTION 调用将挂起。要实现这种用法，应分别创建复制槽（使用函数 pg_create_logical_replication_slot 并指定插件名 pgoutput），然后使用参数 create_slot = false 创建订阅。这是一个实现限制，未来版本中可能会解除。

示例

创建一个指向远端服务器的订阅，复制发布 mypublication 和 insert_only 中的表，并在提交时立即开始复制：

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
        PUBLICATION mypublication, insert_only;
```

创建一个指向远端服务器的订阅，复制insert_only 发布中的表，并且要等到稍后启用时才开始复制。

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
        PUBLICATION insert_only
               WITH (enabled = false);
```

兼容性

CREATE SUBSCRIPTION 是 PostgreSQL 扩展。

又见

ALTER SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

CREATE TABLE

CREATE TABLE — 定义一个新表

大纲

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_
name ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) } [ COLLATE collation ]
 [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_
name
    OF type_name [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) } [ COLLATE collation ]
 [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_
name
    PARTITION OF parent_table [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ] { FOR VALUES partition_bound_spec | DEFAULT }
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) } [ COLLATE collation ]
 [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

其中 *column_constraint* 为:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

而 *table_constraint* 为:

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
```

```
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] ) index_parameters [
 WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

而 *like_option* 为:

```
{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS | DEFAULTS | IDENTITY | INDEXES | STATISTICS
 | STORAGE | ALL }
```

而 *partition_bound_spec* 为:

```
IN ( { numeric_literal | string_literal | TRUE | FALSE | NULL } [, ...] ) |
FROM ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE } [, ...] )
  TO ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )
```

UNIQUE、PRIMARY KEY 和 EXCLUDE 约束中的 *index_parameters* 为:

```
[ INCLUDE ( column_name [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

EXCLUDE 约束中的 *exclude_element* 为:

```
{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

描述

CREATE TABLE 将在当前数据库中创建一个新的、初始为空的表。该表归发出该命令的用户所有。

如果给出了模式名（例如 CREATE TABLE myschema.mytable ...），则表将在指定模式中创建。否则，它将在当前模式中创建。临时表存在于一个特殊模式中，因此创建临时表时不能给出模式名。表名必须与同一模式中任何其他表、序列、索引、视图或外部表的名称不同。

CREATE TABLE 还会自动创建一种数据类型，用以表示与该表一行对应的复合类型。因此，表名不能与同一模式中任何已有数据类型同名。

可选的约束子句指定插入或更新要成功时，新行或更新后的行必须满足的约束（测试）。约束是一种 SQL 对象，可用多种方式帮助定义表中允许的值集合。

定义约束有两种方式：表约束和列约束。列约束作为列定义的一部分定义。表约束则不绑定到特定列，并且可以涵盖多个列。每个列约束也都可以写成表约束；当约束只影响一列时，列约束只是一种书写上的方便。

要创建表，必须分别对所有列类型或 OF 子句中的类型拥有 USAGE 权限。

参数

TEMPORARY 或 TEMP

如果指定该选项，表将创建为临时表。临时表会在会话结束时自动删除，或者也可在当前事务结束时删除（见下文 ON COMMIT）。在临时表存在期间，同名的现有永久表对当前会话不可见，除非使用带模式限定的名称引用它们。在临时表上创建的任何索引也都会自动成为临时索引。

自动清理守护进程不能访问并且因此也不能清理或分析临时表。由于这个原因，应该通过会话的 SQL 命令执行合适的清理和分析操作。例如，如果一个临时表将要被用于复杂的查询，最好在把它填充完毕后在其上运行ANALYZE。

可以在 TEMPORARY 或 TEMP 前写 GLOBAL 或 LOCAL。这在当前的 PostgreSQL 中没有区别，而且已弃用；见下文兼容性。

UNLOGGED

如果指定该选项，表将创建为不记录 WAL 的表。写入不记录 WAL 的表的数据不会写入预写式日志（见第 30 章），因此它们比普通表快得多。不过，它们不具备崩溃安全性：在崩溃或非正常关闭后，不记录 WAL 的表会被自动截断。不记录 WAL 的表的内容也不会复制到备库。在不记录 WAL 的表上创建的任何索引也都会自动成为不记录 WAL 的。

IF NOT EXISTS

如果已存在同名关系，则不抛出错误，而是发出一条提示。注意，这并不保证现有关系与本应创建出的关系有任何相似之处。

table_name

要创建的表名（可选地带模式限定）。

OF *type_name*

创建一个类型化表，其结构取自指定的复合类型（名称可以带模式限定）。类型化表与其类型绑定；例如，如果删除该类型（使用DROP TYPE ... CASCADE），该表也会被删除。

创建类型化表时，列的数据类型由底层复合类型决定，不由CREATE TABLE 命令指定。不过，CREATE TABLE 命令可以为表添加默认值和约束，并指定存储参数。

column_name

要在新表中创建的列名。

data_type

列的数据类型。这可以包括数组说明符。有关 PostgreSQL 支持的数据类型的更多信息，请参见第 8 章。

COLLATE *collation*

COLLATE 子句为列指定排序规则（该列必须属于支持排序规则的数据类型）。如果未指定，则使用列数据类型的默认排序规则。

INHERITS (*parent_table* [, ...])

可选的 INHERITS 子句指定一组表，新表将自动从中继承所有列。父表可以是普通表或外部表。

使用 INHERITS 会在新子表与其父表之间建立持久关系。对父表的模式修改通常也会传播到子表，且默认情况下，对父表的扫描会包含子表的数据。

如果同一列名出现在多个父表中，除非这些父表中该列的数据类型全部匹配，否则会报错。如果没有冲突，这些重复列会合并为新表中的单个列。如果新表的列名列表中包含一个同样来自继承的列名，其数据类型也必须与继承列匹配，并且列定义会合并为一个。如果新表显式为该列指定了默认值，该默认值会覆盖继承声明中的任何默认值。否则，任何为该列指定默认值的父表都必须指定相同的默认值，否则会报错。

CHECK 约束基本上也按与列相同的方式合并：如果多个父表和/或新表定义中包含同名的 CHECK 约束，则这些约束必须拥有相同的检查表达式，否则会报错。同名且表达式相同的约束将合并为一份。父表中标记为 NO INHERIT 的约束不会被考虑。注意，新表中未命名的 CHECK 约束永远不会被合并，因为系统总会为它选择一个唯一名称。

列的 STORAGE 设置也会从父表复制过来。

如果父表中的列是标识列，则该属性不会被继承。若需要，可将子表中的列声明为标识列。

```
PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression )
 } [ opclass ] [, ...] )
```

可选的 PARTITION BY 子句指定表的分区策略。这样创建的表称为分区表。括号中的列或表达式列表构成该表的分区键。使用范围分区或 hash 分区时，分区键可以包含多个列或表达式（最多 32 个，但该限制可在构建 PostgreSQL 时调整）；而列表分区的分区键必须由单个列或表达式组成。

范围分区和列表分区需要 B-树操作符类，hash 分区则需要 hash 操作符类。如果未显式指定操作符类，将使用相应类型的默认操作符类；若不存在默认操作符类，则会报错。使用 hash 分区时，所用操作符类必须实现支持函数 2（详见第 38.15.3 节）。

分区表被划分为多个子表（称为分区），它们使用单独的 CREATE TABLE 命令创建。分区表本身为空。插入到该表的数据行会根据分区键中列或表达式的值被路由到相应分区。如果没有现有分区与新行中的值匹配，就会报错。

分区表不支持 EXCLUDE 约束；不过，可以在单个分区上定义这些约束。此外，虽然可以在分区表上定义PRIMARY KEY 约束，但目前不支持创建引用分区表的外键。

有关表分区的更多讨论，请参阅第 5.10 节。

```
PARTITION OF parent_table { FOR VALUES partition_bound_spec |
 DEFAULT }
```

将该表创建为指定父表的分区。可以使用 FOR VALUES 将其创建为特定值的分区，也可以使用 DEFAULT 创建为默认分区。父表上已有的任何索引、约束和用户定义的行级触发器都会克隆到新分区上。

partition_bound_spec 必须对应于父表的分区方法和分区键，并且不能与该父表的任何现有分区重叠。带 IN 的形式用于列表分区，带 FROM 和 TO 的形式用于范围分区，带 WITH 的形式用于 hash 分区。

为 *partition_bound_spec* 指定的每个值都是字面值、NULL、MINVALUE 或 MAXVALUE。每个字面值必须是可转换为相应分区键列类型的数值常量，或者是对该类型有效的字符串字面量。

在创建列表分区时，可以指定 NULL，表示该分区允许分区键列为 NULL。但是，对于给定的父表，这样的列表分区不能多于一个。NULL 不能用于范围分区。

创建范围分区时，由 FROM 指定的下界是包含边界，而由 TO 指定的上界是不包含边界。也就是说，FROM 列表中指定的值是该分区相应分区键列的有效值，而 TO 列表中的值不是。请注意，必须根据按行比较的规则来理解这一点（第 9.23.5 节）。例如，给定 PARTITION BY RANGE (x,y)，分区边界 FROM (1, 2) TO (3, 4) 允许 x=1 且任意 y>=2，x=2 且任意非 NULL 的 y，以及 x=3 且任意 y<4。

在创建范围分区时，可以使用特殊值 MINVALUE 和 MAXVALUE 表示该列值没有下界或上界。例如，使用 FROM (MINVALUE) TO (10) 定义的分区允许任何小于 10 的值，而使用 FROM (10) TO (MAXVALUE) 定义的分区允许任何大于或等于 10 的值。

在创建涉及多列的范围分区时，将 MAXVALUE 用作下界的一部分、将 MINVALUE 用作上界的一部分也可能是有意义的。例如，使用 FROM (0, MAXVALUE) TO (10, MAXVALUE) 定义的分区允许第一个分区键列大于 0 且小于或等于 10 的所有行。类似地，使用 FROM ('a', MINVALUE) TO ('b', MINVALUE) 定义的分区允许第一个分区键列以 “a” 开头的所有行。

请注意，如果 MINVALUE 或 MAXVALUE 用于分区边界中的某一列，则后续所有列都必须使用相同的值。例如，(10, MINVALUE, 0) 不是有效边界；应写成 (10, MINVALUE, MINVALUE)。

还要注意，某些元素类型（如 timestamp）具有 “无穷大” 的概念，那只是另一种可存储的值。这不同于 MINVALUE 和 MAXVALUE，后两者并非可存储的实际值，而只是表示值无界的方式。MAXVALUE 可以视为大于任何其他值，包括 “无穷大”；MINVALUE 可以视为小于任何其他值，包括 “负无穷大”。因此，范围 FROM ('infinity') TO (MAXVALUE) 并不是空范围；它只允许存储一个值 – “无穷大”。

如果指定了 DEFAULT，则该表将创建为父表的默认分区。此选项不适用于 hash 分区表。任何不适合该父表其他分区的分区键值都会被路由到默认分区。

当一个表已有 DEFAULT 分区并且要向其添加新分区时，必须扫描默认分区，以验证其中不包含应属于新分区的任何行。如果默认分区包含大量行，这可能会很慢。如果默认分区是外部表，或者它具有可证明不可能包含应放入新分区之行的约束，则会跳过扫描。

创建 hash 分区时，必须指定模数和余数。模数必须是正整数，余数必须是小于模数的非负整数。通常，在最初设置 hash 分区表时，应选择一个等于分区数的模数，并为每个分区指定相同的模数和不同的余数（见下文示例）。不过，并不要求每个分区都有相同的模数；只要求 hash 分区表中出现的每个模数都是下一个更大模数的因数。这样就可以按增量方式增加分区数量，而不必一次移动全部数据。例如，假设你有一个包含 8 个分区的 hash 分区表，每个分区的模数都是 8，但发现有必要将分区数增加到 16。你可以分离其中一个模数为 8 的分区，再创建两个新的模数为 16 的分区来覆盖同一部分键空间（其中一个的余数等于被分离分区的余数，另一个的余数等于该值加 8），然后重新向它们填充数据。之后可以对每个模数为 8 的分区重复这一过程，也许是在之后某个时间点，直到一个不剩。虽然每一步仍可能涉及大量数据移动，但仍优于创建一张全新的表并一次性移动所有数据。

分区必须具有与其所属分区表相同的列名和类型。如果父表指定了 WITH OIDS，则所有分区都必须有 OID；父表的 OID 列会像其他列一样被所有分区继承。修改分区表的列名或类型，或者添加或删除 OID 列，都会自动传播到所有分区。每个分区都会自动继承CHECK 约束，但单个分区可以指定额外的CHECK 约束；如果额外约束与父表中约束的名称和条件相同，则会与父表约束合并。可以为每个分区分别指定默认值。但请注意，通过分区表插入元组时不会应用分区的默认值。

插入分区表中的行将自动路由到正确的分区。如果不存在合适的分区，则会发生错误。

TRUNCATE 等通常会影响一个表及其所有继承子表的操作，会级联到所有分区，但也可以在单个分区上执行。请注意，使用DROP TABLE 删除分区需要在父表上获取 ACCESS EXCLUSIVE 锁。

LIKE *source_table* [*like_option* ...]

LIKE 子句指定一个表，新表会自动从中复制所有列名、数据类型及其非空约束。

与 INHERITS 不同，新表和原表在创建完成后就完全脱钩了。对原表的修改不会应用到新表，也不可能在扫描原表时包含新表的数据。

只有指定 INCLUDING DEFAULTS 时，才会复制所复制列定义的默认表达式。默认行为是不包含默认表达式，因此新表中所复制列的默认值将为 NULL。请注意，复制调用数据库修改函数（例如nextval）的默认值，可能会在原表和新表之间创建功能性关联。

只有指定 INCLUDING IDENTITY 时，才会复制所复制列定义中的标识规范。新表的每个标识列都会创建一个新序列，与旧表关联的序列分开。

非空约束始终会复制到新表。只有指定 INCLUDING CONSTRAINTS 时，才会复制CHECK 约束。列约束和表约束之间不作区分。

指定 INCLUDING STATISTICS 时，扩展统计信息会复制到新表。

只有指定 INCLUDING INDEXES 时，才会在新表上创建原表的索引、PRIMARY KEY、UNIQUE 和 EXCLUDE 约束。新索引和约束的名称按照默认规则选择，与原名称无关。（此行为可以避免新索引可能发生名称重复错误。）

只有指定 INCLUDING STORAGE 时，才会复制所复制列定义的STORAGE 设置。默认行为是不包含 STORAGE 设置，因此新表中复制的列使用其类型特定的默认设置。有关STORAGE 设置的更多信息，请参见第 69.2 节。

只有指定 INCLUDING COMMENTS 时，才会复制所复制列、约束和索引的注释。默认行为是不包含注释，因此新表中复制的列和约束没有注释。

INCLUDING ALL 是 INCLUDING COMMENTS INCLUDING CONSTRAINTS INCLUDING DEFAULTS INCLUDING IDENTITY INCLUDING INDEXES INCLUDING STATISTICS INCLUDING STORAGE 的简写形式。

请注意，与INHERITS 不同，LIKE 复制的列和约束不会与同名的列和约束合并。如果显式指定了相同的名称，或在另一个LIKE 子句中指定了相同的名称，则会报错。

LIKE 子句也可用于从视图、外部表或复合类型复制列定义。不适用的选项（例如从视图复制 INCLUDING INDEXES）会被忽略。

CONSTRAINT *constraint_name*

列约束或表约束的可选名称。如果约束被违反，错误消息中会包含该约束名，因此诸如 col must be positive 这样的约束名可以向客户端应用传达有用的约束信息。（若约束名中包含空格，则需要用双引号指定。）如果未指定约束名，系统会生成一个。

NOT NULL

该列不允许包含空值。

NULL

该列允许包含空值。这是默认情况。

该子句仅为兼容非标准 SQL 数据库而提供，不建议在新应用中使用。

CHECK (*expression*) [NO INHERIT]

CHECK 子句指定一个产生布尔结果的表达式。要使插入或更新成功，新行或更新后的行必须满足该表达式。计算结果为 TRUE 或 UNKNOWN 的表达式视为成功。如果插入或更新操作中

的任何一行得到 FALSE 结果，就会抛出错误异常，并且插入或更新不会修改数据库。作为列约束指定的检查约束只应引用该列的值，而出现在表约束中的表达式可以引用多个列。

当前，CHECK 表达式不能包含子查询，也不能引用当前行的列之外的变量（参见第 5.3.1 节）。可以引用系统列 tableoid，但不能引用其他系统列。

标记为 NO INHERIT 的约束不会传播到子表。

当一个表有多个 CHECK 约束时，在检查完 NOT NULL 约束之后，会按名称的字母顺序对每一行进行检查。（9.5 之前的 PostgreSQL 版本并不保证 CHECK 约束的特定触发顺序。）

DEFAULT *default_expr*

DEFAULT 子句为其所在列定义的列指定默认数据值。该值可以是任何不含变量的表达式（不允许子查询，也不允许交叉引用当前表中的其他列）。默认表达式的数据类型必须与该列的数据类型匹配。

默认值表达式会用于任何未为该列指定值的插入操作。如果一列没有默认值，则默认值为 null。

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [(*sequence_options*)]

该子句将列创建为标识列。它会隐式附带一个序列，并且在新插入的行中，该列会自动取得分配给它的序列值。这种列会隐式带有 NOT NULL 约束。

ALWAYS 和 BY DEFAULT 子句决定在 INSERT 语句中序列值与用户指定值的优先关系。如果选择了 ALWAYS，则仅当 INSERT 语句指定 OVERRIDING SYSTEM VALUE 时才接受用户指定的值。如果选择了 BY DEFAULT，则用户指定的值优先。有关详细信息，请参阅 INSERT。（在 COPY 命令中，无论此设置如何，始终使用用户指定的值。）

可选的 *sequence_options* 子句可用于覆盖序列的选项。详情见 CREATE SEQUENCE。

UNIQUE（列约束）
UNIQUE (*column_name* [, ...]) [INCLUDE (*column_name* [, ...])]（表约束）

UNIQUE 约束指定表中一列或多列组成的一组只能包含唯一值。表级唯一约束的行为与列级唯一约束相同，只是它还能跨越多列。因此，该约束要求任意两行在这些列中至少有一列不同。

对于唯一约束，空值不被视为相等。

每个唯一约束都应引用一组列，这组列应不同于该表上任何其他唯一约束或主键约束所引用的列集合。（否则，冗余的唯一约束将被丢弃。）

在为多级分区层次结构建立唯一约束时，目标分区表分区键中的所有列，以及其所有后代分区表分区键中的所有列，都必须包含在约束定义中。

添加唯一约束会自动在约束所用的列或列组上创建唯一 B-树索引。

可选的 INCLUDE 子句会向该索引添加一个或多个仅作为“载荷”的列：对这些列不强制唯一性，而且该索引也不能据此进行搜索。不过，它们可以通过仅索引扫描取回。注意，尽管约束不在内含列上强制执行，它仍然依赖于这些列。因此，对这类列执行某些操作（例如 DROP COLUMN）可能导致约束和索引被级联删除。

`PRIMARY KEY`（列约束）
`PRIMARY KEY ( column_name [, ... ] ) [ INCLUDE ( column_name [, ...]) ]`（表约束）

`PRIMARY KEY` 约束指定表的一列或多列只能包含唯一（不重复）且非空的值。无论作为列约束还是表约束，一个表都只能指定一个主键。

主键约束所引用的列集合应不同于同一表上定义的任何唯一约束所引用的列集合。（否则，该唯一约束是冗余的，会被丢弃。）

`PRIMARY KEY` 强制的数据约束与 `UNIQUE` 和 `NOT NULL` 的组合相同。不过，将一组列标识为主键还会为模式设计提供元数据，因为主键意味着其他表可以将这组列作为行的唯一标识符来依赖。

放在分区表上时，`PRIMARY KEY` 约束与前面描述的 `UNIQUE` 约束有相同的限制。

添加 `PRIMARY KEY` 约束会自动在约束所用的列或列组上创建唯一 B-树索引。

可选的 `INCLUDE` 子句会向该索引添加一个或多个仅作为“载荷”的列：对这些列不强制唯一性，而且该索引也不能据此进行搜索。不过，它们可以通过仅索引扫描取回。注意，尽管约束不在内含列上强制执行，它仍然依赖于这些列。因此，对这类列执行某些操作（例如 `DROP COLUMN`）可能导致约束和索引被级联删除。

`EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] ) index_parameters [ WHERE ( predicate ) ]`

`EXCLUDE` 子句定义一个排他约束。它保证如果任意两行在指定列或表达式上使用指定操作符进行比较，这些比较不会全部返回 `TRUE`。如果所有指定操作符都测试相等，这就等价于 `UNIQUE` 约束，尽管普通唯一约束会更快。不过，排他约束可以指定比简单相等更一般的约束。例如，你可以通过使用 `&&` 操作符来指定一个约束，使表中不存在两个包含重叠圆的行（见第 8.8 节）。

排他约束通过索引实现，因此每个指定的操作符都必须与索引访问方法*index_method* 的适当操作符类关联（见第 11.10 节）。这些操作符必须满足交换律。每个 *exclude_element* 都可以选择指定操作符类和/或排序选项；详见 CREATE INDEX。

访问方法必须支持 `amgettuple`（见第 61 章）；目前这意味着不能使用 GIN。虽然允许，但在排他约束上使用 B-树或 hash 索引意义不大，因为它们做不到比普通唯一约束更好的事情。因此，实践中访问方法总是 GiST 或 SP-GiST。

predicate 允许你只在表的一个子集上指定排他约束；在内部，这会创建一个部分索引。注意，谓词周围的圆括号是必需的。

`REFERENCES reftable [ ( refcolumn ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ]`（列约束）
`FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ]`（表约束）

这些子句指定外键约束，要求新表中由一个或多个列组成的列组只能包含与被引用表中某一行的被引用列中的值相匹配的值。如果省略*refcolumn* 列表，则使用*reftable* 的主键。被引用列必须是被引用表中不可延迟的唯一约束或主键约束的列。用户必须拥有被引用表的 `REFERENCES` 权限（可以针对整个表，也可以针对具体的被引用列）。添加外键约束需要在被引用表上获取`SHARE ROW EXCLUSIVE` 锁。请注意，不能在临时表和永久表之间定义外键约束。还要注意，虽然可以在分区表上定义外键，但不能声明引用分区表的外键。

插入到引用列中的值会按照给定的匹配类型，与被引用表及其被引用列中的值进行匹配。共有三种匹配类型：MATCH FULL、MATCH PARTIAL 和 MATCH SIMPLE（默认值）。MATCH FULL 不允许多列外键中的某一列为空，除非所有外键列都为空；如果它们都为空，则不要求该行在被引用表中有匹配行。MATCH SIMPLE 允许任意外键列为空；如果其中任何一列为空，则不要求该行在被引用表中有匹配行。MATCH PARTIAL 目前尚未实现。（当然，可以对引用列应用 NOT NULL 约束，以防止出现这些情况。）

此外，当被引用列中的数据发生变化时，会对本表列中的数据执行某些操作。ON DELETE 子句指定删除被引用表中的被引用行时要执行的操作。同样，ON UPDATE 子句指定将被引用表中的被引用列更新为新值时要执行的操作。如果行被更新，但被引用列实际上没有变化，则不执行任何操作。除NO ACTION 检查以外的引用操作都不能延迟，即使该约束声明为可延迟也是如此。每个子句可以指定以下操作：

NO ACTION

产生错误，指出删除或更新会违反外键约束。如果该约束被延迟，则会在约束检查时仍存在引用行的情况下产生这个错误。这是默认操作。

RESTRICT

产生错误，指出删除或更新会违反外键约束。这与NO ACTION 相同，但检查不能延迟。

CASCADE

分别删除任何引用已删除行的行，或将引用列的值更新为被引用列的新值。

SET NULL

将引用列设置为空值。

SET DEFAULT

将引用列设置为其默认值。（如果默认值不为空，则被引用表中必须存在与这些默认值匹配的行，否则操作会失败。）

如果被引用列经常变化，可以考虑在引用列上添加索引，使与外键约束关联的引用操作能够更高效地执行。

DEFERRABLE
NOT DEFERRABLE

这控制约束是否可以延迟。不可延迟的约束会在每条命令之后立即检查。可延迟约束的检查可以推迟到事务结束（使用 SET CONSTRAINTS 命令）。NOT DEFERRABLE 是默认值。目前，只有UNIQUE、PRIMARY KEY、EXCLUDE 和 REFERENCES（外键）约束接受此子句。NOT NULL 和 CHECK 约束不可延迟。请注意，不能将可延迟约束用作包含 ON CONFLICT DO UPDATE 子句的 INSERT 语句中的冲突仲裁器。

INITIALLY IMMEDIATE
INITIALLY DEFERRED

如果约束可延迟，则此子句指定检查约束的默认时间。如果约束为 INITIALLY IMMEDIATE，则在每条语句之后检查。这是默认值。如果约束为 INITIALLY DEFERRED，则仅在事务结束时检查。可以使用SET CONSTRAINTS 命令更改约束检查时间。

`WITH ( storage_parameter [= value] [, ... ] )`

该子句为表或索引指定可选的存储参数；详情见存储参数。表的 `WITH` 子句还可以包含 `OIDS=TRUE`（或仅写 `OIDS`），以指定为新表的行分配 OID（对象标识符）；也可以包含`OIDS=FALSE`，以指定行不应具有 OID。如果未指定`OIDS`，默认设置取决于 default_with_oids 配置参数。（如果新表继承自任何具有 OID 的表，则会强制使用 `OIDS=TRUE`，即使命令指定了`OIDS=FALSE` 也是如此。）

如果显式或隐式指定了 `OIDS=FALSE`，新表将不存储 OID，也不会为插入其中的行分配 OID。通常认为这样做是值得的，因为它会减少 OID 的消耗，从而推迟 32 位 OID 计数器回卷。一旦计数器回卷，就不能再假定 OID 是唯一的，这会大大降低它们的用途。此外，不在表中包含 OID 可以减少在磁盘上存储该表所需的空间，在大多数机器上每行可减少 4 字节，从而略微提高性能。

要在表创建后移除其 OID，请使用 ALTER TABLE。

`WITH OIDS`
`WITHOUT OIDS`

这些是过时的语法，分别等价于`WITH (OIDS)` 和 `WITH (OIDS=FALSE)`。如果要同时指定 `OIDS` 设置和存储参数，必须使用`WITH ( ... )`语法；见上文。

`ON COMMIT`

可以使用 `ON COMMIT` 控制临时表在事务块结束时的行为。三种选项如下：

`PRESERVE ROWS`

在事务结束时不执行任何特殊操作。这是默认行为。

`DELETE ROWS`

临时表中的所有行都会在每个事务块结束时删除。实际上，每次提交时都会自动执行一次 TRUNCATE。用于分区表时，这个操作不会级联到其分区。

`DROP`

在当前事务块结束时删除临时表。用于分区表时，此操作会删除其分区；用于带有继承子表的表时，则会删除其依赖子表。

`TABLESPACE tablespace_name`

`tablespace_name` 是要在其中创建新表的表空间名称。如果未指定，则查阅 default_tablespace；如果该表是临时表，则查阅 temp_tablespaces。

`USING INDEX TABLESPACE tablespace_name`

该子句允许选择与 `UNIQUE`、`PRIMARY KEY` 或 `EXCLUDE` 约束相关联的索引要创建在哪个表空间中。若未指定，则参考 default_tablespace；如果该表是临时表，则参考 temp_tablespaces。

存储参数

`WITH` 子句可以为表以及与 `UNIQUE`、`PRIMARY KEY` 或 `EXCLUDE` 约束关联的索引指定存储参数。索引的存储参数已在 CREATE INDEX 中介绍。当前可用于表的存储参数列在下文。对于其中许多参数，如下所示，还存在一个同名且带 `toast.` 前缀的附加参数，用于控制该表的二级

TOAST 表（如果有）的行为（更多有关 TOAST 的信息见第 69.2 节）。如果设置了表参数值，而对应的 `toast.` 参数未设置，则 TOAST 表将使用该表的参数值。不支持为分区表指定这些参数，但可以为单独的叶子分区指定。

`fillfactor` (`integer`)

表的填充因子是介于 10 到 100 之间的百分比。100（完全填充）是默认值。当指定较小的填充因子时，`INSERT` 操作只会把表页填充到指定百分比；每页剩余的空间则预留给更新该页上的行。这使得 `UPDATE` 有机会将更新后的行副本放在与原始行相同的页上，这比放在不同页上更高效，也更有可能发生堆内元组更新。对于条目永远不会更新的表，完全填充是最佳选择；但对频繁更新的表来说，较小的填充因子更合适。不能为 TOAST 表设置此参数。

`toast_tuple_target` (`integer`)

toast_tuple_target 指定在尝试压缩和/或将长列值移入 TOAST 表之前所需的最小元组长度，同时也是一旦开始 TOAST 化后试图将长度缩减到其以下的目标长度。这会影响标记为 External（用于移动）、Main（用于压缩）或 Extended（用于两者）的列，并且只适用于新元组。对现有行没有影响。默认情况下，该参数被设置为至少允许每个块容纳 4 个元组，在默认块大小下即为 2040 字节。有效值介于 128 字节与（块大小减去页头）之间，默认上限为 8160 字节。对非常短或非常长的行，修改该值可能没有帮助。注意，默认设置通常已经接近最佳，某些情况下设置该参数可能产生负面影响。不能为 TOAST 表设置此参数。

`parallel_workers` (`integer`)

该参数设置在对该表执行并行扫描时用于辅助扫描的工作进程数量。若未设置，系统会根据关系大小决定一个值。规划器或使用并行扫描的实用命令实际选择的工作进程数量可能更少，例如可能受max_worker_processes 设置所限。

`autovacuum_enabled`, `toast.autovacuum_enabled` (`boolean`)

为特定表启用或禁用自动清理守护进程。如果为真，自动清理守护进程将按照第 24.1.6 节中讨论的规则，在该表上执行自动 `VACUUM` 和/或 `ANALYZE` 操作。如果为假，则该表不会被自动清理，但为了防止事务 ID 回卷，仍可能对其执行自动清理。有关回卷防护的更多信息，见第 24.1.5 节。注意，如果autovacuum 参数为假，则自动清理守护进程根本不会运行（防止事务 ID 回卷的情况除外）；为单独表设置存储参数也不会覆盖这一点。因此，显式将此存储参数设为 `true` 往往意义不大，设为 `false` 才更有用。

`autovacuum_vacuum_threshold`, `toast.autovacuum_vacuum_threshold` (`integer`)

autovacuum_vacuum_threshold 参数的每表取值。

`autovacuum_vacuum_scale_factor`, `toast.autovacuum_vacuum_scale_factor` (`floating point`)

autovacuum_vacuum_scale_factor 参数的每表取值。

`autovacuum_analyze_threshold` (`integer`)

autovacuum_analyze_threshold 参数的每表取值。

`autovacuum_analyze_scale_factor` (`floating point`)

autovacuum_analyze_scale_factor 参数的每表取值。

`autovacuum_vacuum_cost_delay`, `toast.autovacuum_vacuum_cost_delay` (`integer`)

autovacuum_vacuum_cost_delay 参数的每表取值。

`autovacuum_vacuum_cost_limit`, `toast.autovacuum_vacuum_cost_limit` (`integer`)

autovacuum_vacuum_cost_limit 参数的每表取值。

`autovacuum_freeze_min_age`, `toast.autovacuum_freeze_min_age` (`integer`)

vacuum_freeze_min_age 参数的每表取值。注意，自动清理会忽略大于系统范围 autovacuum_freeze_max_age 设置一半的每表 `autovacuum_freeze_min_age` 参数。

`autovacuum_freeze_max_age`, `toast.autovacuum_freeze_max_age` (`integer`)

autovacuum_freeze_max_age 参数的每表取值。注意，自动清理会忽略大于系统范围设置的每表 `autovacuum_freeze_max_age` 参数（它只能设置得更小）。

`autovacuum_freeze_table_age`, `toast.autovacuum_freeze_table_age` (`integer`)

vacuum_freeze_table_age 参数的每表取值。

`autovacuum_multixact_freeze_min_age`, `toast.autovacuum_multixact_freeze_min_age` (`integer`)

vacuum_multixact_freeze_min_age 参数的每表取值。注意，自动清理会忽略大于系统范围autovacuum_multixact_freeze_max_age 设置一半的每表 `autovacuum_multixact_freeze_min_age` 参数。

`autovacuum_multixact_freeze_max_age`, `toast.autovacuum_multixact_freeze_max_age` (`integer`)

autovacuum_multixact_freeze_max_age 参数的每表取值。注意，自动清理会忽略大于系统范围设置的每表 `autovacuum_multixact_freeze_max_age` 参数（它只能设置得更小）。

`autovacuum_multixact_freeze_table_age`, `toast.autovacuum_multixact_freeze_table_age` (`integer`)

vacuum_multixact_freeze_table_age 参数的每表取值。

`log_autovacuum_min_duration`, `toast.log_autovacuum_min_duration` (`integer`)

log_autovacuum_min_duration 参数的每表取值。

`user_catalog_table` (`boolean`)

将该表声明为逻辑复制用途的附加目录表。详见第 49.6.2 节。不能为 TOAST 表设置此参数。

注解

不建议在新应用中使用 OID：在可能的情况下，优先使用标识列或其他序列生成器作为表的主键。不过，如果应用确实使用 OID 来标识表中的特定行，建议在该表的`oid` 列上创建唯一约束，

以确保即使计数器回卷，表中的 OID 也确实能唯一标识行。不要假定 OID 在不同表之间唯一；如果需要数据库范围的唯一标识符，请组合使用`tableoid`和行 OID。

> **提示**
>
> 对于没有主键的表，不建议使用`OIDS=FALSE`，因为既没有 OID，也没有唯一数据键时，很难标识特定行。

PostgreSQL 为每一个唯一约束和主键约束自动创建一个索引来强制唯一性。因此，没有必要显式地为主键列创建一个索引（详见 CREATE INDEX）。

在当前的实现中，唯一约束和主键不会被继承。这使得继承与唯一约束的组合相当不实用。

一个表不能有超过 1600 列（实际上，由于元组长度限制，有效的限制通常更低）。

示例

创建表 `films` 和表 `distributors`：

```
CREATE TABLE films (
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

创建一个带二维数组列的表：

```
CREATE TABLE array_int (
    vector  int[][]
);
```

为表 `films` 定义一个唯一表约束。唯一表约束可以定义在表的一列或多列上：

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

定义一个列检查约束：

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

定义一个表检查约束：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

为表 `films` 定义一个主键表约束：

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

为表 distributors 定义一个主键约束。下面的两个示例是等价的，第一个使用表约束语法，第二个使用列约束语法：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

为列 name 指定一个字面常量默认值，将列did 的默认值设为从某个序列对象中取下一个值，并让modtime 的默认值为插入该行的时间：

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

在表 distributors 上定义两个 NOT NULL 列约束，其中一个显式指定了名称：

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

为 name 列定义一个唯一约束：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

同样的唯一约束用表约束指定：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name)
);
```

创建同样的表，并为该表及其唯一索引都指定 70% 的填充因子：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

创建表 `circles`，并添加一个排他约束以防任意两个圆重叠：

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

在表空间 `diskvol1` 中创建表 `cinemas`：

```
CREATE TABLE cinemas (
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

创建一个复合类型和一个类型化表：

```
CREATE TYPE employee_type AS (name text, salary numeric);

CREATE TABLE employees OF employee_type (
    PRIMARY KEY (name),
    salary WITH OPTIONS DEFAULT 1000
);
```

创建一个范围分区表：

```
CREATE TABLE measurement (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
```

创建一个在分区键中包含多个列的范围分区表：

```
CREATE TABLE measurement_year_month (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (EXTRACT(YEAR FROM logdate), EXTRACT(MONTH FROM logdate));
```

创建列表分区表：

```
CREATE TABLE cities (
    city_id      bigserial not null,
    name         text not null,
    population   bigint
) PARTITION BY LIST (left(lower(name), 1));
```

创建 hash 分区表：

```
CREATE TABLE orders (
    order_id     bigint not null,
    cust_id      bigint not null,
    status       text
) PARTITION BY HASH (order_id);
```

创建范围分区表的分区：

```
CREATE TABLE measurement_y2016m07
    PARTITION OF measurement (
    unitsales DEFAULT 0
) FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

使用分区键中的多个列创建范围分区表的几个分区：

```
CREATE TABLE measurement_ym_older
    PARTITION OF measurement_year_month
    FOR VALUES FROM (MINVALUE, MINVALUE) TO (2016, 11);

CREATE TABLE measurement_ym_y2016m11
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 11) TO (2016, 12);

CREATE TABLE measurement_ym_y2016m12
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 12) TO (2017, 01);

CREATE TABLE measurement_ym_y2017m01
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2017, 01) TO (2017, 02);
```

创建列表分区表的分区：

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b');
```

创建一个本身还要进一步分区的列表分区表分区，然后再向其添加一个分区：

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b') PARTITION BY RANGE (population);

CREATE TABLE cities_ab_10000_to_100000
    PARTITION OF cities_ab FOR VALUES FROM (10000) TO (100000);
```

创建 hash 分区表的分区：

```
CREATE TABLE orders_p1 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE orders_p2 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE orders_p3 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE orders_p4 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

创建默认分区：

```
CREATE TABLE cities_partdef
    PARTITION OF cities DEFAULT;
```

兼容性

`CREATE TABLE` 命令符合 SQL 标准，但有下列例外。

临时表

尽管 `CREATE TEMPORARY TABLE` 的语法看起来类似于 SQL 标准，但其效果并不相同。按标准，临时表只需定义一次，并会自动存在于每个需要它的会话中（内容初始为空）。而 PostgreSQL 要求每个会话都为每个要使用的临时表发出自己的 `CREATE TEMPORARY TABLE` 命令。这使不同会话可以出于不同目的使用相同的临时表名；而标准做法则要求给定临时表名的所有实例都必须具有相同的表结构。

标准对临时表行为的定义在实践中被广泛忽略。PostgreSQL 在这一点上的行为与多种其他 SQL 数据库相似。

SQL 标准还区分全局和局部临时表，其中局部临时表在每个会话内的每个 SQL 模块中都有独立的内容集合，但其定义仍在多个会话之间共享。由于 PostgreSQL 不支持 SQL 模块，这一区别在 PostgreSQL 中没有意义。

出于兼容性考虑，PostgreSQL 接受在临时表声明中使用 `GLOBAL` 和 `LOCAL` 关键字，但它们目前没有效果。不鼓励使用这些关键字，因为未来版本的 PostgreSQL 可能会采用更符合标准的解释。

临时表的 ON COMMIT 子句也与 SQL 标准相似，但存在一些差异。如果省略 ON COMMIT 子句，SQL 规定默认行为是 ON COMMIT DELETE ROWS。然而，PostgreSQL 中的默认行为是 ON COMMIT PRESERVE ROWS。SQL 中不存在 ON COMMIT DROP 选项。

非延迟唯一性约束

当 UNIQUE 或 PRIMARY KEY 约束不可延迟时，只要有行被插入或修改，PostgreSQL 就会立刻检查唯一性。SQL 标准规定应只在语句结束时强制唯一性；例如，当单个命令会更新多个键值时，这两者就会产生差异。若要获得符合标准的行为，应将约束声明为 DEFERRABLE 但不延迟（即 INITIALLY IMMEDIATE）。注意，这可能明显慢于立即检查唯一性。

列检查约束

SQL 标准规定，CHECK 列约束只能引用其所作用的列；只有 CHECK 表约束才能引用多列。PostgreSQL 并不强制这一限制；它对列检查约束和表检查约束一视同仁。

EXCLUDE 约束

EXCLUDE 约束类型是 PostgreSQL 的扩展。

NULL“约束”

NULL“约束”（实际上并不是约束）是 PostgreSQL 对 SQL 标准的扩展；提供它是为了与其他一些数据库系统兼容（以及与 NOT NULL 约束保持对称）。由于它本来就是任意列的默认情况，所以它的存在只是噪声。

约束命名

SQL 标准规定，在包含表或域的模式范围内，表约束和域约束都必须具有唯一名称。PostgreSQL 较为宽松：它只要求约束名称在附加到特定表或域的约束之间唯一。不过，对于基于索引的约束（UNIQUE、PRIMARY KEY 和 EXCLUDE 约束）就没有这种额外自由，因为关联索引与约束同名，而索引名称在同一模式内的所有关系之间必须唯一。

目前，PostgreSQL 根本不会为 NOT NULL 约束记录名称，因此它们不受唯一性限制。这种情况可能会在未来版本中改变。

继承

通过 INHERITS 子句实现的多重继承是 PostgreSQL 的语言扩展。SQL:1999 及后续标准使用不同的语法和语义定义了单继承。PostgreSQL 尚不支持 SQL:1999 风格的继承。

零列的表

PostgreSQL 允许创建没有列的表（例如 CREATE TABLE foo();）。这是对 SQL 标准的扩展，标准不允许零列的表。零列的表本身并不十分有用，但若禁止它们，就会让 ALTER TABLE DROP COLUMN 出现奇怪的特殊情况，因此忽略这一规范限制看起来更整洁。

多个标识列

PostgreSQL 允许一个表拥有多个标识列。该标准指定一个表最多只能有一个标识列。放宽这一限制主要是为了给模式更改或迁移提供更大的灵活性。请注意，INSERT 命令仅支持一个适用于整个语句的覆盖子句，因此对行为不同的多个标识列支持并不好。

`LIKE` 子句

虽然 SQL 标准中存在 `LIKE` 子句，但 PostgreSQL 接受的许多 LIKE 选项并不在标准中，而标准中的某些选项又没有被 PostgreSQL 实现。

`WITH` 子句

`WITH` 子句是 PostgreSQL 的扩展；存储参数和 OID 都不属于标准内容。

表空间

PostgreSQL 的表空间概念不是标准的一部分。因此，`TABLESPACE` 和 `USING INDEX TABLESPACE` 子句都是扩展。

类型化表

类型化表实现了 SQL 标准的一个子集。按照标准，类型化表除了具有与底层复合类型相对应的列之外，还应有一个额外的“自引用列”。PostgreSQL 不显式支持自引用列，但使用 OID 功能可以达到相同的效果。

`PARTITION BY` 子句

`PARTITION BY` 子句是 PostgreSQL 的扩展。

`PARTITION OF` 子句

`PARTITION OF` 子句是 PostgreSQL 的扩展。

另见

ALTER TABLE, DROP TABLE, CREATE TABLE AS, CREATE TABLESPACE, CREATE TYPE

CREATE TABLE AS

CREATE TABLE AS — 根据查询结果定义一个新表

大纲

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_
name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

描述

CREATE TABLE AS 创建一个表，并用 SELECT 命令计算得到的数据填充该表。该表的列具有与 SELECT 输出列对应的名称和数据类型（但可以通过显式给出新的列名列表来覆盖列名）。

CREATE TABLE AS 与创建视图有些相似，但实际上差异很大：它会创建一个新表，并且只执行一次该查询来完成新表的初始填充。这个新表不会跟踪该查询所引用源表的后续变化。相比之下，视图每次被查询时都会重新执行其定义中的 SELECT 语句。

参数

GLOBAL 或 LOCAL

出于兼容性考虑，这些关键字会被忽略。它们的用法已废弃；详见 CREATE TABLE。

TEMPORARY 或 TEMP

如果指定，将把该表创建为临时表。详见 CREATE TABLE。

UNLOGGED

如果指定，该表将创建为不记录 WAL 的表。详见 CREATE TABLE。

IF NOT EXISTS

如果已存在同名关系，则不抛出错误，而是发出一条提示，并保持现有关系不变。

table_name

要创建的表名（可选地带模式限定）。

column_name

新表中某一列的名称。如果未提供列名，则采用查询输出列的列名。

`WITH ( storage_parameter [= value] [, ... ] )`

此子句为新表指定可选存储参数；有关更多信息，请参见存储参数。`WITH` 子句还可以包含 `OIDS=TRUE`（或仅包含 `OIDS`），以指定应为新表的行分配 OID（对象标识符），或者包含 `OIDS=FALSE`，以指定这些行不应有 OID。更多信息请参见 CREATE TABLE。

`WITH OIDS`
`WITHOUT OIDS`

这些是分别等同于 `WITH (OIDS)` 和 `WITH (OIDS=FALSE)` 的过时语法。如果希望同时指定 `OIDS` 设置和存储参数，则必须使用`WITH ( ... )` 语法；见上文。

`ON COMMIT`

临时表在事务块结束时的行为可以使用以下选项控制：`ON COMMIT`。三个选项是：

`PRESERVE ROWS`

在事务结束时不采取特殊动作。这是默认行为。

`DELETE ROWS`

临时表中的所有行将在每个事务块结束时删除。本质上，每次提交时都会执行一次自动的 TRUNCATE。

`DROP`

临时表会在当前事务块结束时被删除。

`TABLESPACE tablespace_name`

`tablespace_name` 是要在其中创建新表的表空间名称。如果未指定，则查阅 default_tablespace；如果该表是临时表，则查阅 temp_tablespaces。

`query`

可以是 SELECT、TABLE 或 VALUES 命令，也可以是运行已预备的`SELECT`、`TABLE` 或 `VALUES` 查询的 EXECUTE 命令。

`WITH [ NO ] DATA`

这个子句指定是否将查询产生的数据复制到新表中。如果不复制，则只复制表结构。默认会复制数据。

注解

这个命令在功能上类似于 SELECT INTO，但更推荐使用它，因为它更不容易与`SELECT INTO` 语法的其他用法相混淆。此外，`CREATE TABLE AS` 提供的功能是 `SELECT INTO` 所提供功能的超集。

`CREATE TABLE AS` 命令允许用户显式指定是否包含 OID。如果未显式指定是否存在 OID，则使用default_with_oids 配置变量。

示例

创建一个新表 films_recent，其中只包含表 films 中的近期条目：

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

要完整复制一个表，也可以使用TABLE 命令的简写形式：

```
CREATE TABLE films2 AS
  TABLE films;
```

创建一个新的临时表 films_recent，其中只包含来自以下表的近期条目：films，使用预备语句。新表具有 OID，并将在提交时删除：

```
PREPARE recentfilms(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent WITH (OIDS) ON COMMIT DROP AS
  EXECUTE recentfilms('2002-01-01');
```

兼容性

CREATE TABLE AS 符合 SQL 标准。以下是非标准扩展：

- 标准要求在子查询子句外围加括号；在 PostgreSQL 中，这些括号是可选的。
- 在标准中，WITH [NO] DATA 子句是必需的；在 PostgreSQL 中它是可选的。
- PostgreSQL 处理临时表的方式与标准有相当大的不同；详见 CREATE TABLE。
- WITH 子句是 PostgreSQL 扩展；标准中既没有存储参数，也没有 OID。
- PostgreSQL 的表空间概念不是标准的一部分。因此，TABLESPACE 子句是一种扩展。

另见

CREATE MATERIALIZED VIEW, CREATE TABLE, EXECUTE, SELECT, SELECT INTO, VALUES

CREATE TABLESPACE

CREATE TABLESPACE — 定义一个新表空间

大纲

```
CREATE TABLESPACE tablespace_name
    [ OWNER { new_owner | CURRENT_USER | SESSION_USER } ]
    LOCATION 'directory'
    [ WITH ( tablespace_option = value [, ... ] ) ]
```

描述

`CREATE TABLESPACE` 注册一个新的集簇级别的表空间。该表空间的名称必须不同于数据库集簇中任何现有表空间的名称。

表空间允许超级用户在文件系统中定义一个替代位置，使数据库对象（如表和索引）的数据文件可以存放在那里。

具有适当权限的用户可以将 *`tablespace_name`* 传递给 `CREATE DATABASE`、`CREATE TABLE`、`CREATE INDEX` 或 `ADD CONSTRAINT`，从而将这些对象的数据文件存储在指定的表空间中。

> **警告**
>
> 表空间不能脱离定义它的集簇单独使用；见第 22.6 节。

参数

`tablespace_name`

要创建的表空间名称。名称不能以 `pg_`开头，因为这类名称保留给系统表空间使用。

`user_name`

将成为该表空间所有者的用户名。若省略，则默认为执行该命令的用户。只有超级用户可以创建表空间，但他们可以将表空间的所有权赋予非超级用户。

`directory`

将用于表空间的目录。该目录应为空，并且必须归PostgreSQL 系统用户所有。必须使用绝对路径名指定该目录。

`tablespace_option`

要设置或重置的表空间参数。目前唯一可用的参数是 `seq_page_cost`、`random_page_cost` 和 `effective_io_concurrency`。为特定表空间设置其中任一值，会覆盖规划器根据同名配置参数（请参见 seq_page_cost、random_page_cost、

effective_io_concurrency）建立的、读取该表空间中表页面的通常代价估计。如果一个表空间位于比 I/O 子系统其余部分更快或更慢的磁盘上，这可能很有用。

注解

表空间仅在支持符号链接的系统上受支持。

`CREATE TABLESPACE` 不能在事务块内执行。

示例

创建一个表空间 `dbspace`，位置为`/data/dbs`：

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

创建一个表空间 `indexspace`，位置为`/data/indexes`，所有者为用户`genevieve`：

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

兼容性

`CREATE TABLESPACE` 是一种 PostgreSQL 扩展。

另见

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE

CREATE TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH CONFIGURATION — 定义一个新的文本检索配置

大纲

```
CREATE TEXT SEARCH CONFIGURATION name (
    PARSER = parser_name |
    COPY = source_config
)
```

描述

`CREATE TEXT SEARCH CONFIGURATION` 创建一个新的文本检索配置。文本检索配置指定一个能把字符串拆分为词元的文本检索解析器，以及一些可用于判断哪些词元对搜索有意义的词典。

如果只指定了解析器，那么新文本检索配置最初没有从词元类型到词典的映射，因而会忽略所有词。要让该配置真正可用，必须使用后续的`ALTER TEXT SEARCH CONFIGURATION` 命令创建映射。另一种方式是复制一个现有的文本检索配置。

如果给出了一个模式名称，则文本检索配置会被创建在指定的模式中。否则它将会被创建在当前模式中。

定义该文本检索配置的用户会成为其拥有者。

更多信息请参见第 12 章。

参数

name

要创建的文本检索配置的名称。该名称可以是模式限定的。

parser_name

此配置要使用的文本检索解析器的名称。

source_config

要复制的现有文本检索配置的名称。

注解

`PARSER` 和 `COPY` 选项是互斥的，因为当一个已有的配置被复制时，它的解析器选择也会一并被复制。

兼容性

在 SQL 标准中没有 `CREATE TEXT SEARCH CONFIGURATION` 语句。

另见

ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH DICTIONARY — 定义一个新的文本检索词典

大纲

```
CREATE TEXT SEARCH DICTIONARY name (
    TEMPLATE = template
    [, option = value [, ... ]]
)
```

描述

`CREATE TEXT SEARCH DICTIONARY` 创建一个新的文本检索词典。文本检索词典指定一种在搜索时识别哪些词值得关注、哪些词不值得关注的方式。词典依赖于文本检索模板，后者规定了实际执行这项工作的函数。通常，词典会提供一些选项，用来控制模板函数的具体行为。

如果给出了一个模式名称，那么该文本检索词典会被创建在指定的模式中。否则它会被创建在当前模式中。

定义文本检索词典的用户将成为其拥有者。

更多信息请参见第 12 章。

参数

name

要创建的文本检索词典的名称。该名称可以被模式限定。

template

用于定义该词典基本行为的文本检索模板名称。

option

要为此词典设置的模板相关选项的名称。

value

模板相关选项要使用的值。如果该值不是简单的标识符或数字，则必须用引号括起（当然，如果你愿意，也始终可以加引号）。

这些选项可以按任意顺序出现。

示例

下面的示例命令创建了一个基于 Snowball 且使用非标准停用词列表的词典。

```
CREATE TEXT SEARCH DICTIONARY my_russian (
    template = snowball,
    language = russian,
    stopwords = myrussian
);
```

兼容性

在 SQL 标准中没有 `CREATE TEXT SEARCH DICTIONARY` 语句。

另见

ALTER TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH PARSER

CREATE TEXT SEARCH PARSER — 定义一个新的文本检索解析器

大纲

```
CREATE TEXT SEARCH PARSER name (
    START = start_function ,
    GETTOKEN = gettoken_function ,
    END = end_function ,
    LEXTYPES = lextypes_function
    [, HEADLINE = headline_function ]
)
```

描述

`CREATE TEXT SEARCH PARSER` 创建一个新的文本检索解析器。文本检索解析器定义了一种方法，用于将文本字符串拆分成词元并为这些词元指定类型（类别）。解析器本身并没有太大用处，必须与一些文本检索词典一起绑定到一个文本检索配置中，才能用于搜索。

如果给出了一个模式名称，那么文本检索解析器将被创建在指定的模式中。否则它会被创建在当前模式中。

要使用 `CREATE TEXT SEARCH PARSER`，你必须是超级用户。之所以有此限制，是因为错误的文本检索解析器定义可能会让服务器陷入混乱，甚至崩溃。

更多信息请参见第 12 章。

参数

name

要创建的文本检索解析器的名称。该名称可以是模式限定的。

start_function

该解析器的启动函数的名称。

gettoken_function

该解析器的获取下一个词元的函数名称。

end_function

该解析器的结束函数的名称。

lextypes_function

该解析器的 lextypes 函数的名称（该函数返回它所产生的词元类型集合的信息）。

headline_function

该解析器的 headline 函数的名称（该函数对一组词元生成摘要）。

如有必要，函数名称可以是模式限定的。这里未给出参数类型，因为每类函数的参数列表都是预先确定的。除 headline 函数外，其余函数都是必需的。

这些参数可以按任意顺序出现，不必与上面展示的顺序一致。

兼容性

在 SQL 标准中没有 `CREATE TEXT SEARCH PARSER` 语句。

另见

ALTER TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

CREATE TEXT SEARCH TEMPLATE

CREATE TEXT SEARCH TEMPLATE — 定义一个新的文本检索模板

大纲

```
CREATE TEXT SEARCH TEMPLATE name (
    [ INIT = init_function , ]
    LEXIZE = lexize_function
)
```

描述

`CREATE TEXT SEARCH TEMPLATE` 创建一个新的文本检索模板。文本检索模板定义实现文本检索词典的函数。模板本身并无直接用途，必须先实例化为词典后才能使用。词典通常会指定要传递给模板函数的参数。

如果给出了模式名称，则文本检索模板会被创建在指定模式中。否则它会被创建在当前模式中。

要使用 `CREATE TEXT SEARCH TEMPLATE`，你必须是超级用户。之所以有此限制，是因为错误的文本检索模板定义可能使服务器陷入混乱，甚至崩溃。将模板与词典分离的原因在于，模板封装了定义词典时那些“不安全”的方面。而在定义词典时可设置的参数，由普通用户设置也是安全的，因此创建词典不必是特权操作。

更多信息请参见第 12 章。

参数

name

要创建的文本检索模板名称。该名称可以是模式限定的。

init_function

该模板的 init 函数的名称。

lexize_function

该模板的 lexize 函数名称。

如有必要，函数名称可以是模式限定的。这里没有给出参数类型，因为每一类函数的参数列表都是预先确定的。lexize 函数是必需的，但 init 函数是可选的。

这些参数可以按任意顺序出现，不必与上面展示的顺序一致。

兼容性

在 SQL 标准中没有 `CREATE TEXT SEARCH TEMPLATE` 语句。

另见

ALTER TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

CREATE TRANSFORM

CREATE TRANSFORM — 定义一个新的转换

大纲

```
CREATE [ OR REPLACE ] TRANSFORM FOR type_name LANGUAGE lang_name (
    FROM SQL WITH FUNCTION from_sql_function_name [ (argument_type [, ...]) ],
    TO SQL WITH FUNCTION to_sql_function_name [ (argument_type [, ...]) ]
);
```

描述

`CREATE TRANSFORM` 定义一种新的转换。`CREATE OR REPLACE TRANSFORM` 会创建一种新的转换，或替换现有定义。

转换指定如何将一种数据类型适配到某种过程语言。例如，在用 PL/Python 编写一个使用 `hstore` 类型的函数时，PL/Python 事先并不知道该如何在 Python 环境中表示`hstore` 值。语言实现通常默认使用文本表示，但这并不方便，例如在更适合使用关联数组或列表时就是如此。

转换指定两个函数：

- 一个 “from SQL” 函数，用于将该类型从 SQL 环境转换到该语言。该函数会作用于用这种语言编写的函数的参数。
- 一个 “to SQL” 函数，用于将该类型从该语言转换到 SQL 环境。该函数会作用于用这种语言编写的函数的返回值。

不必同时提供这两个函数。如果其中一个未指定，则会在需要时使用该语言相关的默认行为。（若要彻底阻止某一方向上的转换，也可以编写一个总是报错的转换函数。）

要创建一种转换，你必须拥有该类型并具有其上的 `USAGE` 权限，具有该语言上的 `USAGE` 权限，并且在指定了 from-SQL 和 to-SQL 函数时，还必须拥有这些函数并具有其上的`EXECUTE` 权限。

参数

type_name

该转换的数据类型名称。

lang_name

该转换所用语言的名称。

from_sql_function_name[(*argument_type* [, ...])]

用于将该类型从 SQL 环境转换到该语言的函数名称。它必须接受一个 `internal` 类型参数，并返回`internal` 类型。实际传入的参数将是该转换对应的类型，因此编写该函数时应按这一点处理。（但不允许声明一个返回`internal`、却没有至少一个 `internal` 类型参数的 SQL 层函数。）实际返回值将是语言实现特有的某个对象。如果未指定参数列表，则函数名在其所在模式中必须唯一。

to_sql_function_name[(*argument_type* [, ...])]

用于将该类型从该语言转换到 SQL 环境的函数名称。它必须接受一个 `internal` 类型参数，并返回该转换对应的类型。实际参数值将是语言实现特有的某个对象。如果未指定参数列表，则函数名在其所在模式中必须唯一。

注解

使用 DROP TRANSFORM 删除转换。

示例

为类型 `hstore` 和语言 `plpythonu` 创建转换时，首先设置该类型和语言：

```
CREATE TYPE hstore ...;

CREATE EXTENSION plpythonu;
```

然后创建所需的函数：

```
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS internal
LANGUAGE C STRICT IMMUTABLE
AS ...;

CREATE FUNCTION plpython_to_hstore(val internal) RETURNS hstore
LANGUAGE C STRICT IMMUTABLE
AS ...;
```

最后创建转换，将它们连接起来：

```
CREATE TRANSFORM FOR hstore LANGUAGE plpythonu (
    FROM SQL WITH FUNCTION hstore_to_plpython(internal),
    TO SQL WITH FUNCTION plpython_to_hstore(internal)
);
```

实际上，这些命令会被封装在扩展中。

`contrib` 部分包含若干提供转换的扩展，可以作为实际示例。

兼容性

这种形式的 `CREATE TRANSFORM` 是 PostgreSQL 的一种扩展。SQL 标准中也有 `CREATE TRANSFORM` 命令，但它用于将数据类型适配到客户端语言。PostgreSQL 不支持那种用法。

另见

CREATE FUNCTION, CREATE LANGUAGE, CREATE TYPE, DROP TRANSFORM

CREATE TRIGGER

CREATE TRIGGER — 定义一个新触发器

大纲

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
```

其中 *event* 为以下之一：

```
    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

描述

CREATE TRIGGER 创建一个新触发器。触发器会关联到指定的表、视图或外部表，并在对该表执行某些操作时执行指定的函数*function_name*。

可以指定触发器在尝试对某一行执行该操作之前引发（即在检查约束以及尝试执行 INSERT、UPDATE 或 DELETE 之前）；也可以在该操作完成之后引发（即在检查约束以及完成 INSERT、UPDATE 或 DELETE 之后）；或者取代该操作执行（用于视图上的插入、更新或删除）。如果触发器在事件之前引发，或者改为取代该事件执行，则它可以跳过对当前行的操作，或者修改待插入的行（仅适用于INSERT 和 UPDATE 操作）。如果触发器在事件之后引发，则所有更改（包括其他触发器的影响）对该触发器都是“可见”的。

被标记为 FOR EACH ROW 的触发器会针对该操作修改的每一行调用一次。例如，一个影响 10 行的DELETE 会导致目标关系上的 ON DELETE 触发器分别被调用 10 次，即每个被删除的行调用一次。相比之下，被标记为FOR EACH STATEMENT 的触发器对于任意给定操作只执行一次，不管该操作修改了多少行（特别是，修改零行的操作仍会导致所有适用的 FOR EACH STATEMENT 触发器执行）。

被指定为取代该触发事件执行的 INSTEAD OF 触发器必须标记为 FOR EACH ROW，并且只能定义在视图上。视图上的 BEFORE 和 AFTER 触发器必须标记为 FOR EACH STATEMENT。

此外，也可以把触发器定义为在 TRUNCATE 时引发，但只能是 FOR EACH STATEMENT。

下表汇总了哪些类型的触发器可用于表、视图和外部表：

时机	事件	行级	语句级
BEFORE	INSERT/ UPDATE/DELETE	表和外部表	表、视图和外部表
	TRUNCATE	–	表
AFTER	INSERT/ UPDATE/DELETE	表和外部表	表、视图和外部表

时机	事件	行级	语句级
	TRUNCATE	–	表
INSTEAD OF	INSERT/ UPDATE/DELETE	视图	–
	TRUNCATE	–	–

此外，触发器定义还可以指定一个布尔型 WHEN 条件，用于测试是否应当引发该触发器。在行级触发器中，WHEN 条件可以检查该行各列的新值和/或旧值。语句级触发器也可以带有 WHEN 条件，不过这一特性对它们的用处不大，因为该条件无法引用表中的任何值。

如果针对同一事件定义了多个同类触发器，它们将按名称的字母顺序引发。

当指定 CONSTRAINT 选项时，此命令会创建一个约束触发器。这与普通触发器相同，只是可以使用SET CONSTRAINTS 调整其触发时机。约束触发器必须是普通表（不是外部表）上的 AFTER ROW 触发器。它们既可以在导致触发事件的语句结束时触发，也可以在包含它的事务结束时触发；后一种情况称为延迟。也可以使用 SET CONSTRAINTS 强制尚待执行的延迟触发器立即触发。约束触发器应当在其所实现的约束遭到违反时抛出异常。

REFERENCING 选项允许收集过渡关系，即包含当前 SQL 语句插入、删除或修改的全部行的行集合。该特性使触发器能够看到该语句所做更改的整体视图，而不是一次只看到一行。该选项仅允许用于不是约束触发器的AFTER 触发器；如果触发器是 UPDATE 触发器，则不得指定 *column_name* 列表。OLD TABLE 最多只能指定一次，并且只能用于可能在UPDATE 或 DELETE 上触发的触发器；它会创建一个过渡关系，其中包含该语句更新或删除的全部行的前映像。同样，NEW TABLE 最多只能指定一次，并且只能用于可能在UPDATE 或 INSERT 上触发的触发器；它会创建一个过渡关系，其中包含该语句更新或插入的全部行的后映像。

SELECT 不会修改任何行，因此无法创建 SELECT 触发器。对于那些看起来需要 SELECT 触发器的问题，规则和视图可能提供可行的解决方案。

参见第 39 章以了解有关触发器的更多信息。

参数

name

新触发器的名称。它必须不同于同一表上的任何其他触发器名称。该名称不能是模式限定名 – 触发器会继承其所属表的模式。对于约束触发器，这也是使用SET CONSTRAINTS 修改触发器行为时所使用的名称。

BEFORE
AFTER
INSTEAD OF

决定该函数是在事件之前、之后调用，还是取代该事件执行。约束触发器只能指定为 AFTER。

event

INSERT、UPDATE、DELETE 或 TRUNCATE 之一；它指定了哪个事件将引发该触发器。除请求过渡关系的情况外，可以使用OR 指定多个事件。

对于 UPDATE 事件，可以使用以下语法指定列列表：

```
UPDATE OF column_name1 [, column_name2 ... ]
```

只有当列出的列中至少有一列被列为 UPDATE 命令的目标时，触发器才会触发。

INSTEAD OF UPDATE 事件不允许指定列列表。请求过渡关系时也不能指定列列表。

table_name

该触发器所作用的表、视图或外部表的名称（可以是模式限定的）。

referenced_table_name

约束所引用的另一张表的名称（可以是模式限定的）。此选项用于外键约束，不建议用于一般用途。它只能为约束触发器指定。

```
DEFERRABLE
NOT DEFERRABLE
INITIALLY IMMEDIATE
INITIALLY DEFERRED
```

触发器的默认时机。有关这些约束选项的细节，请参见 CREATE TABLE。它们只能为约束触发器指定。

```
REFERENCING
```

这个关键字紧接在一个或两个关系名的声明之前，这些关系名可用于访问触发语句的过渡关系。

```
OLD TABLE
NEW TABLE
```

该子句指示后面的关系名是用于前映像过渡关系还是后映像过渡关系。

transition_relation_name

该过渡关系在触发器内部使用的名称（非限定名）。

```
FOR EACH ROW
FOR EACH STATEMENT
```

这指定触发器函数是对受触发事件影响的每一行执行一次，还是每条 SQL 语句仅执行一次。如果两者都未指定，默认值是FOR EACH STATEMENT。约束触发器只能指定为 FOR EACH ROW。

condition

一个布尔表达式，用于决定触发器函数是否会实际执行。如果指定了WHEN，则仅当 *condition* 返回 true 时才会调用该函数。在 FOR EACH ROW 触发器中，WHEN 条件可以分别写成 OLD.*column_name* 或 NEW.*column_name* 来引用旧行值和/或新行值中的列。当然，INSERT 触发器不能引用 OLD，而 DELETE 触发器不能引用 NEW。

INSTEAD OF 触发器不支持 WHEN 条件。

当前，WHEN 表达式不能包含子查询。

请注意，对于约束触发器，WHEN 条件的求值不会被延迟，而是在行操作执行后立即发生。如果该条件求值结果不为真，则该触发器不会被加入延迟执行队列。

function_name

一个由用户提供的函数，声明为不接受参数并返回 trigger 类型；当触发器被触发时，就会执行该函数。

在 CREATE TRIGGER 的语法中，关键字 FUNCTION 和 PROCEDURE 是等价的，但无论如何，被引用的对象都必须是函数而不是过程。这里使用 PROCEDURE 关键字只是出于历史原因，并且已被弃用。

arguments

一个可选的、以逗号分隔的参数列表，在执行触发器时会传给该函数。这些参数是字符串字面常量。这里也可以写简单名称和数字常量，但它们都会被转换成字符串。请查阅该触发器函数所用实现语言的说明，以了解如何在函数内部访问这些参数；这可能与普通函数参数不同。

注解

要在表上创建触发器，用户必须拥有该表的TRIGGER 权限。用户还必须拥有触发器函数的 EXECUTE 权限。

使用 DROP TRIGGER 删除触发器。

列限定触发器（即使用 UPDATE OF *column_name* 语法定义的触发器）会在其任一列被列为 UPDATE 命令的 SET 列表目标时引发。即便触发器没有引发，列值仍有可能发生变化，因为 BEFORE UPDATE 触发器对行内容所作的更改不会被考虑在内。反过来，诸如 UPDATE ... SET x = x ... 这样的命令会引发列 x 上的触发器，即使该列的值实际上并未变化。

有少量内置触发器函数可用于解决常见问题，而无需自己编写触发器代码；参见第 9.27 节。

在 BEFORE 触发器中，WHEN 条件会在函数执行前（或者本应执行前）立即求值，因此使用WHEN 与在触发器函数开头测试相同条件并无实质区别。特别要注意的是，该条件看到的NEW 行是当前值，它可能已经被更早触发的触发器修改过。此外，BEFORE 触发器的 WHEN 条件不允许检查 NEW 行的系统列（例如 oid），因为这些列此时尚未被设置。

在 AFTER 触发器中，WHEN 条件会在行变更发生后立即求值，并决定是否将一个事件排入队列，以便在语句结束时引发该触发器。因此，当 AFTER 触发器的 WHEN 条件不返回真时，就没有必要将事件排队，也不需要在语句结束时重新获取该行。如果触发器只需针对少数几行引发，这能显著提升那些会修改很多行的语句的速度。

在某些情况下，一条 SQL 命令可能会引发不止一种触发器。例如，带有 ON CONFLICT DO UPDATE 子句的 INSERT 可能同时导致插入和更新操作，因此会按需引发这两类触发器。提供给触发器的过渡关系是按事件类型区分的；因此 INSERT 触发器只能看到插入的行，而 UPDATE 触发器只能看到更新的行。

由外键强制执行动作导致的行更新或删除，例如 ON UPDATE CASCADE 或 ON DELETE SET NULL，会被视为导致它们发生的那条 SQL 命令的一部分（注意，这类动作永远不会被延迟）。受影响表上的相关触发器也会被引发，因而这提供了另一种情形，使得一条 SQL 命令可能引发与其类型并不直接匹配的触发器。在简单情况下，请求过渡关系的触发器会把单条原始 SQL 命令在该表中造成的所有更改视为一个过渡关系。不过，在某些情况下，如果存在

一个请求过渡关系的 AFTER ROW 触发器，就会使单条 SQL 命令触发的外键强制执行动作被拆分成多个步骤，每一步都有各自的过渡关系。在这种情况下，任何存在的语句级触发器都会在每组过渡关系创建时引发一次，从而保证这些触发器在过渡关系中看到每个受影响的行时，恰好只看到一次。

只有当视图上的动作由行级 INSTEAD OF 触发器处理时，视图上的语句级触发器才会引发。如果该动作由 INSTEAD 规则处理，那么规则发出的那些语句会代替原先引用该视图的语句执行，因此最终引发的是替代语句所引用表上的触发器。同样，如果该视图是自动可更新的，那么该动作会通过把语句自动重写为作用于视图基表的动作来处理，因而最终引发的是基表上的语句级触发器。

在分区表上创建行级触发器，会在其所有现有分区上创建相同的触发器；以后创建或附加的任何分区也会包含相同的触发器。如果分区从其父表分离，触发器就会被移除。分区表上的触发器只能是AFTER 触发器。

修改分区表或带有继承子表的表时，会触发附着在显式命名表上的语句级触发器，但不会触发其分区或子表上的语句级触发器。相反，行级触发器会在受影响的分区或子表中的行上触发，即使这些分区或子表在查询中没有被显式命名。如果某个语句级触发器通过REFERENCING 子句定义了过渡关系，那么所有受影响分区或子表中的行的前映像和后映像都是可见的。对于继承子表，这些行映像只包含该触发器所附着表中存在的列。目前，带过渡关系的行级触发器不能定义在分区或继承子表上。

在 PostgreSQL 7.3 之前的版本中，必须将触发器函数声明为返回占位类型opaque，而不是 trigger。为了支持加载旧转储文件，CREATE TRIGGER 将接受声明为返回 opaque 的函数，但会发出通知，并将该函数声明的返回类型更改为trigger。

示例

执行函数 check_account_update，每当表 accounts 中的一行即将被更新时调用它：

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

相同，但仅当列 balance 被指定为 UPDATE 命令的目标时才执行该函数：

```
CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

这种写法只有在列 balance 的值确实发生变化时才执行该函数：

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE FUNCTION check_account_update();
```

调用一个函数记录 accounts 的更新，但仅在确有内容发生变化时才调用：

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE FUNCTION log_account_update();
```

执行函数 view_insert_row，针对每一行将行插入视图底层的表中：

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE FUNCTION view_insert_row();
```

执行函数 check_transfer_balances_to_zero，针对每条语句确认 transfer 中这些行相互抵销后的净额为零：

```
CREATE TRIGGER transfer_insert
    AFTER INSERT ON transfer
    REFERENCING NEW TABLE AS inserted
    FOR EACH STATEMENT
    EXECUTE FUNCTION check_transfer_balances_to_zero();
```

执行函数 check_matching_pairs，针对每一行确认对配对项的更改会同时发生（即由同一条语句完成）：

```
CREATE TRIGGER paired_items_update
    AFTER UPDATE ON paired_items
    REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
    FOR EACH ROW
    EXECUTE FUNCTION check_matching_pairs();
```

第 39.4 节包含一个用 C 编写的触发器函数的完整示例。

兼容性

PostgreSQL 中的 CREATE TRIGGER 语句实现了 SQL 标准的一个子集。当前仍缺少以下功能：

- 虽然 AFTER 触发器的过渡表名可以按标准方式用 REFERENCING 子句指定，但 FOR EACH ROW 触发器中使用的行变量不能在 REFERENCING 子句中指定。它们以依赖于触发器函数所用语言的方式提供，但对于任意一种语言而言，这种方式是固定的。有些语言的实际行为就像存在一个含有 OLD ROW AS OLD NEW ROW AS NEW 的 REFERENCING 子句一样。
- 标准允许将过渡表与列限定 UPDATE 触发器一起使用，但这样一来，在过渡表中应当可见的行集合就取决于触发器的列列表。PostgreSQL 当前尚未实现这一点。
- PostgreSQL 只允许通过执行用户定义函数来完成触发动作。标准则允许把许多其他 SQL 命令（例如 CREATE TABLE）作为触发动作执行。通过创建一个执行所需命令的用户定义函数，这一限制并不难绕过。

SQL 规定多个触发器应按创建时间顺序引发。PostgreSQL 则按名称顺序引发，这被认为更方便。

SQL 规定，级联删除上的 `BEFORE DELETE` 触发器应在级联 `DELETE` 完成之后引发。PostgreSQL 的行为则是 `BEFORE DELETE` 总会在删除动作之前引发，即使是级联删除也一样。这被认为更一致。如果 `BEFORE` 触发器在由引用动作引起的更新过程中修改行或阻止更新，也会出现非标准行为。这可能导致约束违反，或者使存储的数据不符合引用约束。

使用 `OR` 为单个触发器指定多个动作的能力，是 PostgreSQL 对 SQL 标准的扩展。

能够在 `TRUNCATE` 上引发触发器，是 PostgreSQL 对 SQL 标准的扩展；能够在视图上定义语句级触发器也是如此。

`CREATE CONSTRAINT TRIGGER` 是 PostgreSQL 对 SQL 标准的扩展。

另见

ALTER TRIGGER, DROP TRIGGER, CREATE FUNCTION, SET CONSTRAINTS

CREATE TYPE

CREATE TYPE — 定义一种新数据类型

大纲

```
CREATE TYPE name AS
    ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
)

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name
```

描述

CREATE TYPE 在当前数据库中注册一种新的数据类型。定义该类型的用户将成为其拥有者。

如果给出了模式名，则该类型会在指定模式中创建；否则会在当前模式中创建。类型名必须不同于同一模式中任何现有类型或域的名称。（由于表也具有关联的数据类型，类型名还必须不同于同一模式中任何现有表的名称。）

如上面的语法概要所示，`CREATE TYPE` 有五种形式。它们分别用于创建复合类型、枚举类型、范围类型、基础类型或 shell 类型。下文将依次讨论前四种。shell 类型只是稍后定义某种类型时使用的占位符；它通过执行除类型名外不带任何参数的 `CREATE TYPE` 来创建。正如相应小节所述，创建范围类型和基础类型时，需要用 shell 类型作为前向引用。

复合类型

第一种形式的 `CREATE TYPE` 创建复合类型。复合类型由属性名和数据类型列表指定。如果某个属性的数据类型支持排序规则，还可以指定该属性的排序规则。复合类型本质上与表的行类型相同，但如果目的只是定义一种类型，使用`CREATE TYPE` 就不必实际创建表。例如，独立的复合类型可用作函数的参数类型或返回类型。

要创建复合类型，必须对所有属性类型都拥有`USAGE` 权限。

枚举类型

如第 8.7 节中所述，第二种形式的 `CREATE TYPE` 创建枚举（enum）类型。枚举类型接受一个带引号标签的列表，其中每个标签的长度都必须小于 `NAMEDATALEN` 字节（在标准 PostgreSQL 构建中为 64 字节）。（也可以创建零标签的枚举类型，但在使用ALTER TYPE 至少添加一个标签之前，这种类型不能用来保存值。）

范围类型

如第 8.17 节中所述，第三种形式的 `CREATE TYPE` 创建范围类型。

范围类型的 *`subtype`* 可以是任何带有关联 B-树操作符类的类型（该操作符类用于确定范围类型值的顺序）。通常使用子类型默认的 B-树操作符类来确定顺序；若要使用非默认操作符类，可用 *`subtype_opclass`* 指定其名称。如果子类型支持排序规则，而你希望在范围排序中使用非默认排序规则，可用 *`collation`* 选项指定所需排序规则。

可选的 *`canonical`* 函数必须接受一个正在定义的范围类型值作为参数，并返回同一类型的值。在适用时，它用于将范围值转换为规范形式。更多信息见第 8.17.8 节。创建 *`canonical`* 函数有些棘手，因为它必须在声明范围类型之前定义。为此，必须先创建一种 shell 类型，它除了名称和拥有者外没有任何属性，只是一个占位符类型。这可通过执行不带任何附加参数的命令 `CREATE TYPE` *`name`* 完成。然后就可以把该 shell 类型用作参数类型和结果类型来声明该函数，最后再用同一名称声明范围类型。这样会自动用有效的范围类型替换 shell 类型条目。

可选的 *`subtype_diff`* 函数必须接受两个 *`subtype`* 类型的值作为参数，并返回一个表示这两个给定值之差的`double precision` 值。虽然这是可选的，但提供该函数可显著提高该范围类型列上 GiST 索引的效率。详见第 8.17.8 节。

基础类型

第四种形式的 `CREATE TYPE` 创建一种新的基础类型（标量类型）。要创建新的基础类型，你必须是超级用户。（这样限制是因为错误的类型定义可能使服务器陷入混乱，甚至导致其崩溃。）

这些参数可以按任意顺序出现，不必局限于上面展示的顺序，而且大多数都是可选的。在定义该类型之前，必须先注册两个或更多函数（使用 `CREATE FUNCTION`）。支持函数 *`input_function`* 和 *`output_function`* 是必需的；*`receive_function`*、

send_function、*type_modifier_input_function*、*type_modifier_output_function*、*analyze_function* 则是可选的。通常这些函数必须用 C 或其他低级语言编写。

input_function 将类型的外部文本表示转换为该类型的操作符和函数所使用的内部表示。*output_function* 执行相反的转换。输入函数可以声明为接受一个 cstring 参数，或者接受三个参数，类型分别为cstring、oid、integer。第一个参数是以 C 字符串表示的输入文本，第二个参数是该类型自身的 OID（数组类型例外，此时传入的是其元素类型的 OID），第三个参数是在已知情况下目标列的 typmod（未知则传入 -1）。输入函数必须返回该数据类型本身的值。通常输入函数应声明为 STRICT；否则，在读取 NULL 输入值时会以 NULL 作为第一个参数调用它。除非函数抛出错误，否则在这种情况下仍必须返回 NULL。（这种情况主要是为了支持可能需要拒绝 NULL 输入的域输入函数。）输出函数必须声明为接受一个新数据类型参数，并且必须返回cstring 类型。对于 NULL 值不会调用输出函数。

可选的 *receive_function* 把类型的外部二进制表示转换为内部表示。如果未提供此函数，该类型就不能参与二进制输入。外部二进制表示应选择为既能低成本转换为内部形式，又具有合理可移植性。（例如，标准整数数据类型把网络字节序用作外部二进制表示，而内部表示则使用机器的本地字节序。）*receive_function* 应进行充分检查以确保值有效。它可以声明为接受一个 internal 参数，或者接受三个参数，类型分别为 internal、oid、integer。第一个参数是指向保存已接收字节串的 StringInfo 缓冲区的指针；其余可选参数与文本输入函数相同。接收函数必须返回该数据类型本身的值。通常，接收函数应声明为 STRICT；否则，在读取 NULL 输入值时会以 NULL 作为第一个参数调用它。除非函数抛出错误，否则在这种情况下仍必须返回 NULL。（这种情况主要是为了支持可能需要拒绝 NULL 输入的域接收函数。）类似地，可选的 *send_function* 把内部表示转换为外部二进制表示。如果未提供此函数，该类型就不能参与二进制输出。发送函数必须声明为接受一个新数据类型参数，并且必须返回 bytea 类型。对于 NULL 值不会调用发送函数。

读到这里，你可能会问：既然新类型本身还没创建，输入和输出函数怎么能声明为返回或接受这个新类型呢？答案是，应先把该类型定义为一种 shell 类型，它除了名称和拥有者外没有任何属性，只是一个占位符类型。这可通过执行不带任何附加参数的命令 CREATE TYPE *name* 完成。然后就可以定义引用该 shell 类型的 C I/O 函数。最后，再用带完整定义的CREATE TYPE 替换该 shell 条目，生成一个完整且有效的类型定义，此后新类型就能正常使用。

如果该类型支持修饰符，也就是附加在类型声明上的可选约束，例如 char(5) 或 numeric(30,2)，就需要可选的*type_modifier_input_function* 和 *type_modifier_output_function*。PostgreSQL 允许用户定义类型接受一个或多个简单常量或标识符作为修饰符。不过，这些信息必须能够打包成单个非负整数值，以便存储在系统目录中。声明的修饰符会以 cstring 数组的形式传递给 *type_modifier_input_function*。它必须检查这些值是否有效（若无效则抛出错误），若有效则返回一个非负integer 值，该值将作为列的“typmod”存储。如果该类型没有 *type_modifier_input_function*，就会拒绝类型修饰符。*type_modifier_output_function* 则把内部整数 typmod 值转换回适合用户显示的正确形式。它必须返回一个 cstring 值，即精确追加到类型名后的字符串；例如，numeric 的该函数可能返回 (30,2)。允许省略 *type_modifier_output_function*；在这种情况下，默认显示格式只是把存储的 typmod 整数值放在圆括号中。

可选的 *analyze_function* 为该数据类型的列执行类型专用的统计信息收集。默认情况下，如果该类型有默认的 B-树操作符类，ANALYZE 将尝试使用该类型的“等于”和“小于”操作符收集统计信息。对于非标量类型，这种行为很可能不合适，因此可以通过指定自定义分析函数来覆盖。分析函数必须声明为接受一个 internal 参数并返回 boolean 结果。分析函数的详细 API 见 src/include/commands/vacuum.h。

虽然新类型内部表示的细节只有 I/O 函数以及你为该类型编写的其他函数才知道，但仍有若干内部表示属性必须向 PostgreSQL 声明。其中最重要的是 *internallength*。基础类型可以是定长的，此时*internallength* 为正整数；也可以是变长的，此时将*internallength* 设为 VARIABLE。（在内部，这通过把 typlen 设为 -1 表示。）所有变长类型的内部表示都必须以一

个 4 字节整数开头，用来给出该类型该值的总长度。（注意，如第 69.2 节中所述，长度字段通常是经过编码的；直接访问它并不明智。）

可选标志 PASSEDBYVALUE 表示该数据类型的值按值传递，而不是按引用传递。按值传递的类型必须是定长的，且其内部表示不能大于Datum 类型的大小（某些机器上为 4 字节，另一些为 8 字节）。

alignment 参数指定该数据类型所需的存储对齐方式。允许的值分别对应按 1、2、4 或 8 字节边界对齐。注意，变长类型的对齐至少必须为 4，因为它们的第一个组成部分必然是一个int4。

storage 参数允许为变长数据类型选择存储策略。（定长类型只允许 plain。）plain 表示该类型数据始终行内存储且不压缩。extended 表示系统会先尝试压缩较长的数据值，如果仍然过长，就把该值移出主表行。external 允许把值移出主表，但系统不会尝试压缩它。main 允许压缩，但不鼓励把值移出主表。（采用这种存储策略的数据项在没有其他办法让一行适配时仍可能被移出主表，但与extended 和 external 数据项相比，它们会被优先保留在主表中。）

如第 69.2 节和第 38.12.1 节所述，除plain 之外所有 *storage* 值都意味着该数据类型的函数能够处理经过 TOAST 处理的值。具体指定哪一种其他值，只是决定可 TOAST 数据类型列的默认 TOAST 存储策略；用户仍可使用ALTER TABLE SET STORAGE 为单个列选择其他策略。

like_type 参数提供了指定数据类型基本表示属性的另一种方法：从某个现有类型复制这些属性。*internallength*、*passedbyvalue*、*alignment* 和 *storage* 的值都从指定类型复制而来。（虽然可以通过同时给出 LIKE 子句和这些选项来覆盖其中某些值，但通常不建议这样做。）当新类型的底层实现以某种方式“借用”现有类型时，以这种方式指定表示属性尤其有用。

category 和 *preferred* 参数可用于在存在歧义时帮助控制应用哪一种隐式类型转换。每种数据类型都属于一个由单个 ASCII 字符命名的类别，并且在其类别内要么是“首选”的，要么不是。当这一规则有助于解析重载函数或操作符时，解析器会优先转换为首选类型（但只会从同一类别中的其他类型转换）。更多细节见第 10 章。对于与任何其他类型之间都没有隐式类型转换的类型，保持这些设置的默认值就足够了。不过，对于一组彼此存在隐式类型转换的相关类型，把它们都标记为属于同一类别，并选择一两个“最通用”的类型作为该类别的首选类型，通常会有帮助。*category* 参数在将用户定义类型加入现有内置类别（例如数值类型或字符串类型）时尤其有用。不过，也可以创建全新的纯用户定义类型类别。为这种类别命名时，可选择任一非大写字母的 ASCII 字符。

如果用户希望该数据类型列的默认值不是空值，可以指定默认值。用 DEFAULT 关键字指定默认值。（该默认值可以被附加到具体列上的显式 DEFAULT 子句覆盖。）

若要表明一种类型是数组类型，可用ELEMENT 关键字指定数组元素的类型。例如，要定义由 4 字节整数（int4）构成的数组，可指定 ELEMENT = int4。更多关于数组类型的细节见下文。

若要指定该类型数组在外部表示中用于分隔各值的分隔符，可把 *delimiter* 设为特定字符。默认分隔符是逗号（,）。注意，这个分隔符关联的是数组元素类型，而不是数组类型本身。

如果可选的布尔参数 *collatable* 为真，则该类型的列定义和表达式可以通过COLLATE 子句携带排序规则信息。是否实际使用这些排序规则信息取决于操作该类型的函数实现；仅仅把类型标记为支持排序规则并不会自动实现这一点。

数组类型

每当创建用户定义类型时，PostgreSQL 都会自动创建一个关联的数组类型，其名称由元素类型名前加一个下划线组成；必要时还会截断，以保持其长度小于 NAMEDATALEN 字节。（如果这样生成的名称与现有类型名冲突，就会重复这一过程，直到找到不冲突的名称。）这种隐式创建的

数组类型是变长的，并使用内置输入/输出函数 `array_in` 和 `array_out`。该数组类型会跟踪其元素类型的拥有者或模式的任何变化，并在元素类型被删除时一并删除。

如果系统会自动创建正确的数组类型，你可能会合理地问，为什么还需要`ELEMENT` 选项。`ELEMENT` 唯一有用的情况是：你正在创建一种定长类型，而它在内部恰好是若干相同元素组成的数组，并且除了为整个类型提供的操作之外，你还希望允许通过下标直接访问这些元素。例如，类型`point` 在内部就表示为两个浮点数，可以用`point[0]` 和 `point[1]` 访问。注意，这种机制只适用于内部形式恰好是一串相同定长字段的定长类型。可用下标访问的变长类型必须采用 `array_in` 和 `array_out` 使用的通用内部表示。由于历史原因（也就是说这显然不对，但现在改已经太晚了），定长数组类型的下标从零开始，而变长数组则从一开始。

参数

name

要创建的类型名称（可选地带有模式限定）。

attribute_name

复合类型的一个属性（列）的名称。

data_type

将成为复合类型一列的现有数据类型名称。

collation

要与复合类型的某一列或范围类型关联的现有排序规则名称。

label

表示枚举类型某个值所关联文本标签的字符串字面量。

subtype

范围类型所表示范围的元素类型名称。

subtype_operator_class

子类型的 B-树操作符类名称。

canonical_function

范围类型规范化函数的名称。

subtype_diff_function

子类型差分函数的名称。

input_function

将数据从类型的外部文本形式转换为内部形式的函数名。

output_function

将数据从类型的内部形式转换为外部文本形式的函数名。

receive_function

将数据从类型的外部二进制形式转换成内部形式的函数名。

send_function

将数据从类型的内部形式转换为外部二进制形式的函数名。

type_modifier_input_function

将类型的修饰符数组转换为内部形式的函数名。

type_modifier_output_function

将类型修饰符的内部形式转换为外部文本形式的函数名。

analyze_function

为该数据类型执行统计分析的函数名。

internallength

一个数字常量，用于指定新类型内部表示的字节长度。默认假定它是变长的。

alignment

该数据类型的存储对齐需求。如果被指定，它必须是 `char`、`int2`、`int4` 或者 `double`。默认是 `int4`。

storage

该数据类型的存储策略。如果被指定，必须是 `plain`、`external`、`extended` 或者 `main`。默认是 `plain`。

like_type

与新类型具有相同表示形式的现有数据类型名称。除非在本 `CREATE TYPE` 命令的其他位置显式覆盖，否则 *internallength*、*passedbyvalue*、*alignment* 和 *storage* 的值都会从该类型复制。

category

该类型的类别码（单个 ASCII 字符）。默认值是表示“用户定义类型”的`'U'`。其他标准类别码见表 52.63。你也可以选择其他 ASCII 字符来创建自定义类别。

preferred

若该类型是其类型类别中的首选类型，则为真，否则为假。默认值为假。在现有类型类别中创建新的首选类型时要格外小心，因为这可能导致出人意料的行为变化。

default

该数据类型的默认值。若省略，默认值为 null。

element

正在创建的类型是数组；该参数指定数组元素类型。

delimiter

在由该类型构成的数组中各值之间使用的分隔符字符。

collatable

如果该类型的操作可以使用排序规则信息，则为真。默认为假。

注解

由于数据类型一旦创建，其使用方式就不再受限制，因此创建基础类型或范围类型，相当于对类型定义中提到的那些函数授予公共执行权限。这对适合用于类型定义的那类函数来说通常不是问题。但如果设计一种类型时，需要在把它转换为外部形式或从外部形式转换回来时使用“秘密”信息，就应当三思。

在 PostgreSQL 8.3 之前，自动生成的数组类型名称总是恰好等于元素类型名称前加一个下划线字符（_）。（因此，类型名称的长度限制比其他名称少一个字符。）虽然现在通常仍是这样，但在名称达到最大长度或与以下划线开头的用户类型名冲突时，数组类型名称可能与此不同。因此，依赖这一约定编写代码的做法已经弃用。请改用 pg_type.typarray 来定位与给定类型关联的数组类型。

建议避免使用以下划线开头的类型名和表名。虽然服务器会改变生成的数组类型名称以避免与用户给定的名称冲突，但仍然存在混淆风险，特别是对旧客户端软件而言，它们可能会假定以下划线开头的类型名总是表示数组。

在 PostgreSQL 8.2 之前，不存在 shell 类型创建语法CREATE TYPE *name*。创建新基础类型的做法是先创建它的输入函数。在这种做法下，PostgreSQL 会首先把新数据类型名视为输入函数的返回类型。此时 shell 类型会被隐式创建，然后就可以在其余 I/O 函数的定义中引用它。这种做法仍然有效，但已弃用，并且可能在未来某个版本中被禁止。另外，为避免由于函数定义中的简单拼写错误而意外使系统目录充满 shell 类型，只有在输入函数用 C 编写时，才会以这种方式创建 shell 类型。

在 PostgreSQL 7.3 以前的版本中，通常完全不创建 shell 类型，而是将函数中对类型名的前向引用替换为占位伪类型opaque。在 7.3 以前，cstring 参数和结果也必须声明为 opaque。为了支持载入旧转储文件，CREATE TYPE 会接受使用 opaque 声明的 I/O 函数，但会发出通知，并修改函数声明以使用正确的类型。

示例

这个示例创建一种复合类型，并在函数定义中使用它：

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

这个示例创建一种枚举类型，并在表定义中使用它：

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

这个示例创建一种范围类型：

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

这个示例创建基础类型 box，然后在表定义中使用它：

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

如果 box 的内部结构是由四个 float4 元素构成的数组，则也可以改为这样写：

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

这样就能通过下标访问 box 值的各个分量。除此之外，该类型的行为与前例相同。

这个示例创建一种大对象类型，并在表定义中使用它：

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

更多示例（包括配套的输入和输出函数）请见第 38.12 节。

兼容性

第一种 `CREATE TYPE` 形式，即创建复合类型的形式，符合SQL 标准。其他形式都是 PostgreSQL 扩展。SQL 标准中的 `CREATE TYPE` 语句还定义了 PostgreSQL 尚未实现的其他形式。

支持创建零属性的复合类型，是 PostgreSQL 对标准的一种特有背离（类似于 `CREATE TABLE` 中的同类情况）。

另见

ALTER TYPE, CREATE DOMAIN, CREATE FUNCTION, DROP TYPE

CREATE USER

CREATE USER — 定义一个新的数据库角色

大纲

```
CREATE USER name [ [ WITH ] option [ ... ] ]
```

其中 *option* 可以是：

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

描述

`CREATE USER` 现为 CREATE ROLE 的别名。唯一的区别在于：当命令写为 `CREATE USER` 时，默认假定为`LOGIN`；而当命令写为 `CREATE ROLE` 时，默认假定为`NOLOGIN`。

兼容性

`CREATE USER` 语句是 PostgreSQL 的一种扩展。SQL 标准将用户的定义留给具体实现决定。

另见

CREATE ROLE

CREATE USER MAPPING

CREATE USER MAPPING — 定义用户到外部服务器的新映射

大纲

```
CREATE USER MAPPING [ IF NOT EXISTS ] FOR { user_name | USER | CURRENT_USER | PUBLIC }
    SERVER server_name
    [ OPTIONS ( option 'value' [ , ... ] ) ]
```

描述

`CREATE USER MAPPING` 定义用户到外部服务器的映射。用户映射通常封装连接信息，外部数据包装器会将其与外部服务器所封装的信息结合起来，以访问外部数据资源。

外部服务器的拥有者可以创建任意用户到该服务器的用户映射。此外，如果某个用户已被授予该服务器上的`USAGE` 权限，那么该用户也可以为自己的用户名创建用户映射。

参数

`IF NOT EXISTS`

如果给定用户到给定外部服务器的映射已经存在，则不要报错。这种情况下会发出一个提示。请注意，不能保证现有用户映射与本应创建的映射有任何相似之处。

`user_name`

被映射到外部服务器的现有用户的名称。`CURRENT_USER` 和 `USER` 都匹配当前用户的名称。当指定 `PUBLIC` 时，会创建一个所谓的公共映射；当没有适用的特定用户映射时，就会使用它。

`server_name`

将为其创建用户映射的现有服务器的名称。

`OPTIONS ( option 'value' [, ... ] )`

该子句指定用户映射的选项。这些选项通常定义该映射实际的用户名和密码。选项名必须唯一。允许的选项名和值取决于该服务器的外部数据包装器。

示例

为用户 `bob`、服务器`foo` 创建一个用户映射：

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'secret');
```

兼容性

`CREATE USER MAPPING` 符合 ISO/IEC 9075-9（SQL/MED）。

另见

ALTER USER MAPPING, DROP USER MAPPING, CREATE FOREIGN DATA WRAPPER, CREATE SERVER

CREATE VIEW

CREATE VIEW — 定义一个新视图

大纲

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

描述

`CREATE VIEW` 定义一个视图，其内容来自一个查询。该视图不会被实际物化。相反，每次在查询中引用该视图时，都会执行定义它的查询。

`CREATE OR REPLACE VIEW` 与之类似，但如果已经存在同名视图，则会用新定义替换它。新查询必须生成与现有视图查询相同的列（即列名、列顺序和数据类型都相同），但可以在列表末尾附加额外的列。生成输出列的计算方式则可以完全不同。

如果给出了模式名称（例如，`CREATE VIEW myschema.myview ...`），则视图将在指定的模式中创建。否则，它将在当前模式中创建。临时视图存在于一个特殊模式中，因此在创建临时视图时不能给出模式名称。视图的名称必须与同一模式中的任何其他视图、表、序列、索引或外部表的名称不同。

参数

`TEMPORARY` 或 `TEMP`

如果指定该选项，视图将被创建为临时视图。临时视图会在当前会话结束时自动删除。当临时视图存在时，除非使用模式限定名称引用，否则具有相同名称的现有永久关系对当前会话不可见。

如果视图引用的任何表是临时表，则该视图也会被创建为临时视图（无论是否指定了 `TEMPORARY`）。

`RECURSIVE`

创建一个递归视图。语法

```
CREATE RECURSIVE VIEW [ schema . ] view_name (column_names) AS SELECT ...;
```

等效于

```
CREATE VIEW [ schema . ] view_name AS WITH RECURSIVE view_name (column_names) AS (SELECT ..
.) SELECT column_names FROM view_name;
```

递归视图必须指定视图列名列表。

name

要创建的视图名称（可以带模式限定）。

column_name

视图各列使用的名称列表，可选。若未给出，则从查询中推断列名。

WITH (*view_option_name* [= *view_option_value*] [, ...])

该子句为视图指定可选参数；支持以下参数：

check_option (string)

该参数可以是 local 或 cascaded，等同于指定WITH [CASCADED | LOCAL] CHECK OPTION（见下文）。可以使用 ALTER VIEW 更改现有视图上的此选项。

security_barrier (boolean)

如果该视图旨在提供行级安全，则应使用此选项。详见第 41.5 节。

query

一个将为该视图提供列和行的 SELECT 或 VALUES 命令。

WITH [CASCADED | LOCAL] CHECK OPTION

该选项控制自动可更新视图的行为。指定该选项时，会检查对该视图执行的 INSERT 和 UPDATE 命令，以确保新行满足视图定义条件（即确保能够通过该视图看到这些新行）。如果不满足条件，该修改将被拒绝。若未指定 CHECK OPTION，则允许在该视图上执行 INSERT 和 UPDATE 命令来创建通过该视图不可见的行。支持以下检查选项：

LOCAL

新行只会根据该视图自身直接定义的条件进行检查。底层基视图上定义的任何条件都不会被检查（除非它们也指定了 CHECK OPTION）。

CASCADED

新行会根据该视图及所有底层基视图的条件进行检查。如果指定了 CHECK OPTION，但既未指定 LOCAL 也未指定 CASCADED，则假定为CASCADED。

CHECK OPTION 不能与 RECURSIVE 视图一起使用。

注意，CHECK OPTION 仅支持自动可更新且不带 INSTEAD OF 触发器或 INSTEAD 规则的视图。如果一个自动可更新视图定义在带有 INSTEAD OF 触发器的基视图之上，则可以使用 LOCAL CHECK OPTION 检查该自动可更新视图的条件，但不会检查带有INSTEAD OF 触发器的基视图上的条件（级联检查选项不会继续向下级联到触发器可更新视图，而直接定义在触发器可更新视图上的任何检查选项也会被忽略）。如果该视图或其任何基关系带有会导致 INSERT 或 UPDATE 命令被重写的 INSTEAD 规则，则在重写后的查询中所有检查选项都会被忽略，包括来自定义在带有INSTEAD 规则的关系之上的自动可更新视图的任何检查选项。

注解

使用 DROP VIEW 语句删除视图。

应留意视图列的名称和类型是否按预期确定。例如：

```
CREATE VIEW vista AS SELECT 'Hello World';
```

这种写法不好，因为列名默认是?column?；而且列数据类型默认是 text，这可能并非所需。若要在视图结果中使用字符串字面量，更好的写法类似于：

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

对视图中引用的表的访问由视图所有者的权限决定。在某些情况下，这可以用于为底层表提供安全但受限的访问。不过，并非所有视图都能防止篡改；详情请参见第 41.5 节。视图调用的函数与直接从使用该视图的查询中调用时的处理方式相同。因此，视图的用户必须拥有调用视图所用全部函数的权限。

对现有视图使用 CREATE OR REPLACE VIEW 时，只会更改视图的定义性 SELECT 规则。其他视图属性（包括所有权、权限和非 SELECT 规则）保持不变。必须拥有该视图才能替换它（包括作为所有者角色的成员）。

可更新视图

简单视图是自动可更新的：系统允许像对普通表一样在视图上使用 INSERT、UPDATE 和 DELETE 语句。满足下列所有条件的视图是自动可更新的：

- 该视图的 FROM 列表中必须恰好只有一项，而且这一项必须是一个表或另一个可更新视图。
- 视图定义的顶层不能包含 WITH、DISTINCT、GROUP BY、HAVING、LIMIT 或者 OFFSET 子句。
- 视图定义的顶层不能包含集合操作（UNION、INTERSECT 或者 EXCEPT）。
- 视图的 SELECT 列表不能包含任何聚合函数、窗口函数或集合返回函数。

自动可更新视图可以同时包含可更新列和不可更新列。如果某列只是简单引用了底层基关系中的一个可更新列，则该列是可更新的；否则，该列为只读，如果INSERT 或 UPDATE 语句试图为其赋值，就会报错。

如果视图是自动可更新的，系统会把视图上的任何INSERT、UPDATE 或 DELETE 语句转换为底层基关系上的对应语句。带有 ON CONFLICT UPDATE 子句的 INSERT 语句也得到完全支持。

如果自动可更新视图包含 WHERE 条件，那么在该视图上执行UPDATE 和 DELETE 语句时，该条件会限制底层基关系中哪些行可被修改。不过，仍允许UPDATE 把某一行改成不再满足 WHERE 条件，从而使其不再能通过该视图看到。类似地，INSERT 命令也可能插入不满足 WHERE 条件的基关系行，因此这些行通过该视图不可见（ON CONFLICT UPDATE 也可能类似地影响现有的、通过该视图不可见的行）。可以使用CHECK OPTION 阻止 INSERT 和 UPDATE 命令创建这类通过该视图不可见的行。

如果自动可更新视图带有 security_barrier 属性，那么视图上的所有WHERE 条件（以及任何使用标记为 LEAKPROOF 的操作符的条件）总会先于视图使用者添加的任何条件求值。详见第 41.5 节。请注意，因此那些最终不会返回的行（因为它们未通过用户的WHERE 条件）仍可能被锁定。可以使用 EXPLAIN 查看哪些条件是在关系级别应用的（因而不会锁定行），哪些则不是。

默认情况下，不满足所有这些条件的更复杂视图是只读的：系统不允许在该视图上执行插入、更新或删除。可以通过在视图上创建INSTEAD OF触发器来实现可更新视图的效果；这些触发器必须把针对该视图的插入等操作转换为在其他表上执行的适当动作。有关更多信息，请参见CREATE TRIGGER。另一种可能性是创建规则（参见 CREATE RULE），但实际上触发器更容易理解和正确使用。

请注意，在视图上执行插入、更新或删除的用户必须拥有该视图上相应的插入、更新或删除权限。此外，视图所有者必须拥有底层基关系上的相关权限，但执行更新的用户不需要拥有底层基关系上的任何权限（参见第 41.5 节）。

示例

创建一个由所有喜剧影片组成的视图：

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

这会创建一个视图，其中包含创建视图时films表中的各列。尽管创建视图时使用了*，但以后添加到该表的列不会成为该视图的一部分。

创建带有 LOCAL CHECK OPTION 的视图：

```
CREATE VIEW universal_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL CHECK OPTION;
```

这会创建一个基于视图 comedies 的视图，只显示 kind = 'Comedy' 且 classification = 'U' 的影片。如果新行不满足 classification = 'U'，则任何对该视图执行 INSERT 或 UPDATE 的尝试都会被拒绝，但不会检查影片的kind。

创建带有 CASCADED CHECK OPTION 的视图：

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

这会创建一个同时检查新行 kind 和 classification 的视图。

创建一个同时包含可更新列和不可更新列的视图：

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS country,
           (SELECT avg(r.rating)
            FROM user_ratings r
            WHERE r.film_id = f.id) AS avg_rating
    FROM films f
    WHERE f.kind = 'Comedy';
```

该视图将支持 `INSERT`、`UPDATE` 和 `DELETE`。所有来自 `films` 表的列都可更新，而计算列 `country` 和 `avg_rating` 为只读。

创建一个由 1 到 100 的数字组成的递归视图：

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

请注意，尽管在这个`CREATE` 命令中递归视图名称带有模式限定，但其内部自引用并没有带模式限定。这是因为隐式创建的 CTE 名称不能带模式限定。

兼容性

`CREATE OR REPLACE VIEW` 是 PostgreSQL 的语言扩展。临时视图的概念也是如此。`WITH ( ... )` 子句同样是扩展。

另见

ALTER VIEW, DROP VIEW, CREATE MATERIALIZED VIEW

DEALLOCATE

DEALLOCATE — 释放一个预备语句

大纲

```
DEALLOCATE [ PREPARE ] { name | ALL }
```

描述

`DEALLOCATE` 用于释放一个先前准备好的 SQL 语句。如果不显式释放预备语句，则它会在会话结束时被释放。

更多关于预备语句的信息，请参见 PREPARE。

参数

`PREPARE`

此关键字会被忽略。

`name`

要释放的预备语句的名称。

`ALL`

释放所有预备语句。

兼容性

SQL 标准中包含一个 `DEALLOCATE` 语句，但它只用于嵌入式 SQL。

另见

EXECUTE, PREPARE

DECLARE

DECLARE — 定义一个游标

大纲

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

描述

DECLARE 允许用户创建游标，游标可用于从较大的查询中一次取出少量行。游标创建后，可使用 FETCH 从中取出行。

> **注意**
>
> 本页面描述的是 SQL 命令层面的游标用法。如果想要在 PL/pgSQL 函数中使用游标，规则会有所不同 – 见第 43.7 节。

参数

name

要创建的游标名称。

BINARY

使游标以二进制格式而不是文本格式返回数据。

INSENSITIVE

表示从游标取出的数据不应受到游标创建后对其底层表所做更新的影响。在 PostgreSQL 中，这是默认行为；因此该关键字没有效果，仅为兼容 SQL 标准而接受它。

SCROLL
NO SCROLL

SCROLL 指定游标可用于以非顺序方式（例如向后）取出行。根据查询执行计划的复杂程度，指定SCROLL 可能会给查询执行带来性能开销。NO SCROLL 指定游标不能以非顺序方式取出行。默认情况下只在某些情形下允许滚动，这与显式指定 SCROLL 并不相同。详见下文注解。

WITH HOLD
WITHOUT HOLD

WITH HOLD 指定在创建该游标的事务成功提交后，仍可继续使用该游标。WITHOUT HOLD 指定该游标不能在创建它的事务之外使用。如果既未指定 WITHOUT HOLD 也未指定 WITH HOLD，默认值是WITHOUT HOLD。

query

一个提供游标要返回的行的 SELECT 或者 VALUES 命令。

关键字 BINARY、INSENSITIVE 和 SCROLL 可以按任意顺序出现。

注解

普通游标以文本格式返回数据，就像SELECT 产生的结果一样。BINARY 选项指定游标应以二进制格式返回数据。这减少了服务器和客户端两端的转换工作量，但代价是程序员需要付出更多精力来处理与平台相关的二进制数据格式。举例来说，如果某个查询从一个整型列返回值 1，那么默认游标会返回字符串 1，而二进制游标则会返回一个 4 字节字段，其中包含该值的内部表示（采用大端字节序）。

应谨慎使用二进制游标。许多应用程序（包括 psql）都没有准备好处理二进制游标，而是期望返回的数据为文本格式。

> **注意**
>
> 当客户端应用使用“扩展查询”协议发出 FETCH 命令时，Bind 协议消息会指定数据应以文本格式还是二进制格式提取。这一选择会覆盖定义游标时所指定的方式。因此，在使用扩展查询协议时，二进制游标这一概念实际上已经过时了 – 任何游标都可以按文本或二进制方式处理。

除非指定了 WITH HOLD，否则该命令创建的游标只能在当前事务中使用。因此，DECLARE 若不带 WITH HOLD，在事务块之外就毫无用处：游标只能存活到该语句执行完成。所以，如果在事务块之外使用这种命令，PostgreSQL 会报错。可使用 BEGIN 和 COMMIT（或 ROLLBACK）来定义事务块。

如果指定了 WITH HOLD，且创建游标的事务成功提交，那么在同一会话中的后续事务里仍可继续访问该游标。（但如果创建事务被中止，游标会被移除。）使用WITH HOLD 创建的游标会在对其发出显式 CLOSE 命令时关闭，或在会话结束时关闭。在当前实现中，这种游标所表示的行会被复制到临时文件或内存区域中，以便它们在后续事务中仍然可用。

当查询包括 FOR UPDATE 或 FOR SHARE 时，不能指定WITH HOLD。

在定义将用于向后取出行的游标时，应指定SCROLL 选项。这是 SQL 标准所要求的。不过，为了兼容早期版本，如果游标的查询计划足够简单，以致支持向后取出不需要额外开销，PostgreSQL 也允许在未指定 SCROLL 的情况下向后取出行。不过，建议应用开发者不要依赖于从未用 SCROLL 创建的游标中向后取出行。如果指定了 NO SCROLL，那么无论如何都不允许向后取出行。

当查询包含 FOR UPDATE 或 FOR SHARE 时，同样不允许向后取出行。因此在这种情况下不能指定SCROLL。

> **小心**
>
> 如果可滚动游标调用了任何易变函数（见第 38.7 节），则可能得到意外结果。当重新取出先前已经取出过的行时，这些函数可能会再次执行，从而导致与第一次不同的结果。对于涉及易变函数的查询，最好指定NO SCROLL。如果这不现实，一个变通办法是将游标声

明为SCROLL WITH HOLD，并在读取其中任何行之前提交事务。这样会强制将游标的整个输出物化到临时存储中，从而使每一行上的易变函数都只执行一次。

如果游标的查询包含 FOR UPDATE 或 FOR SHARE，则返回的行会在首次取出时被锁定，与带有这些选项的普通SELECT 命令相同。此外，返回的行将是最新版本；因此，这些选项提供了相当于 SQL 标准所称的“敏感游标”的行为。（将 INSENSITIVE 与 FOR UPDATE 或 FOR SHARE 一起指定会报错。）

小心

通常都建议使用 FOR UPDATE，如果游标打算与 UPDATE ... WHERE CURRENT OF 或 DELETE ... WHERE CURRENT OF 一起使用。使用 FOR UPDATE 可以防止其他会话在这些行被取出之后、被更新之前更改它们。如果不使用FOR UPDATE，而某一行在游标创建后已经被更改，那么后续的WHERE CURRENT OF 命令将不会有任何效果。

使用 FOR UPDATE 的另一个原因是：如果没有它，而后续的WHERE CURRENT OF 所针对的游标查询不符合 SQL 标准关于“简单可更新”的规则，则该命令可能失败。（特别是，游标必须只引用一个表，且不能使用分组或ORDER BY）。对于并非简单可更新的游标，是否可用取决于计划选择的细节，可能能工作，也可能不能。因此在最坏情况下，应用可能在测试中可用，却会在生产中失败。如果指定了FOR UPDATE，则可保证该游标可更新。

不将 FOR UPDATE 与 WHERE CURRENT OF 一起使用的主要原因，是你需要游标可滚动，或者需要它对后续更新不敏感（也就是说，继续显示旧数据）。如果这是需求，请务必仔细注意上面的警告。

SQL 标准只为嵌入式 SQL 中的游标作出规定。PostgreSQL 服务器没有为游标实现 OPEN 语句；游标在声明时即被视为打开。不过，ECPG 是 PostgreSQL 的嵌入式 SQL 预处理器，它支持标准 SQL 的游标约定，包括涉及DECLARE 和 OPEN 语句的那些约定。

你可以通过查询 pg_cursors 系统视图查看所有可用游标。

示例

声明一个游标：

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

更多游标用法示例见 FETCH。

兼容性

SQL 标准规定，游标默认是否对底层数据的并发更新敏感，由实现决定。在PostgreSQL 中，游标默认不敏感，可以通过指定FOR UPDATE 使其敏感。其他产品的行为可能不同。

SQL 标准只允许在嵌入式 SQL 和模块中使用游标。PostgreSQL 允许以交互方式使用游标。

二进制游标是 PostgreSQL 的一种扩展。

另见

CLOSE, FETCH, MOVE

DELETE

DELETE — 删除表中的行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

描述

DELETE 从指定表中删除满足 WHERE 子句的行。如果省略 WHERE 子句，效果就是删除表中的所有行。结果是一个合法但为空的表。

> **提示**
>
> TRUNCATE 提供了一种更快的机制来移除表中的所有行。

有两种方法可以利用数据库中其他表所包含的信息来删除一个表中的行：使用子选择，或者在 USING 子句中指定附加表。哪种技术更合适取决于具体情况。

可选的 RETURNING 子句使 DELETE 基于每个实际被删除的行计算并返回一个或多个值。可以计算使用该表列和/或 USING 中提到的其他表列的任意表达式。RETURNING 列表的语法与 SELECT 的输出列表相同。

要从一个表中删除行，必须具有该表上的DELETE 权限，以及USING 子句中出现的任何表或者其值会在 *condition* 中读取的任何表上的 SELECT 权限。

参数

with_query

WITH 子句允许你指定一个或多个子查询，这些子查询可在DELETE 查询中按名称引用。详见第 7.8 节和 SELECT。

table_name

要从中删除行的表名（可以是模式限定的）。如果在表名前指定了 ONLY，只会从所提及表中删除匹配行。如果未指定 ONLY，还会删除继承自该表的任何表中的匹配行。可选地，可以在表名后指定*，以显式指示包含后代表。

alias

目标表的替代名称。提供别名时，它会完全隐藏该表的实际名称。例如，给定 DELETE FROM foo AS f，DELETE 语句的其余部分必须将该表称为 f，而不是foo。

from_item

一个表表达式，允许其他表的列出现在WHERE 条件中。这里使用的语法与 FROM 子句相同，即SELECT 语句中的相应子句；例如，可以为表名指定别名。不要把目标表重复写成 *from_item*，除非打算进行自连接（这种情况下，它必须在*from_item* 中带有别名出现）。

condition

一个返回 boolean 类型值的表达式。只有使这个表达式返回 true 的行才会被删除。

cursor_name

要在 WHERE CURRENT OF 条件中使用的游标名。要被删除的行是最近一次从该游标中取出的那一行。该游标必须是针对 DELETE 目标表的非分组查询。注意，WHERE CURRENT OF 不能与布尔条件同时指定。有关将游标用于 WHERE CURRENT OF 的更多信息，请见 DECLARE。

output_expression

在每一行被删除后，由DELETE 命令计算并返回的表达式。该表达式可以使用由 *table_name* 命名的表或 USING 中列出的表的任何列名。写成 * 可返回所有列。

output_name

用于返回列的名称。

输出

在成功完成时，一个DELETE 命令会返回以下形式的命令标签：

```
DELETE count
```

count 是被删除的行数。注意，当删除被BEFORE DELETE 触发器抑制时，这个数量可能小于匹配*condition* 的行数。如果 *count* 为 0，则该查询没有删除任何行（这不被视为错误）。

如果 DELETE 命令包含 RETURNING 子句，其结果将类似于一个SELECT 语句，其中包含 RETURNING 列表中定义的列和值，并在该命令删除的行上进行计算。

注解

PostgreSQL 允许通过在 USING 子句中指定其他表，在 WHERE 条件中引用这些表的列。例如，要删除由给定制片人制作的所有电影，可以这样做：

```
DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name = 'foo';
```

本质上，这里发生的是在films 和 producers 之间进行连接，并将所有成功连接到的films 行标记为删除。这种语法不是标准的。更标准的写法是：

```
DELETE FROM films
  WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

在某些情况下，连接形式比子选择形式更容易书写或者执行更快。

示例

删除所有电影，但音乐剧除外：

```
DELETE FROM films WHERE kind <> 'Musical';
```

清空表 films：

```
DELETE FROM films;
```

删除已完成的任务，并返回被删除行的完整详情：

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

删除游标 c_tasks 当前所定位的 tasks 行：

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

兼容性

这个命令符合 SQL 标准，不过 USING 和 RETURNING 子句是 PostgreSQL 扩展，在 DELETE 中使用 WITH 也是扩展。

另见

TRUNCATE

DISCARD

DISCARD — 丢弃会话状态

大纲

```
DISCARD { ALL | PLANS | SEQUENCES | TEMPORARY | TEMP }
```

描述

`DISCARD` 释放与数据库会话关联的内部资源。该命令可用于部分或完全重置会话状态。它提供了若干子命令，用于释放不同类型的资源；`DISCARD ALL` 变体涵盖了其他所有子命令，并且还会重置额外的状态。

参数

`PLANS`

释放所有缓存的查询计划，从而在下一次使用相关预备语句时强制重新规划。

`SEQUENCES`

丢弃所有缓存的序列相关状态，包括 `currval()`/`lastval()` 信息，以及所有尚未由`nextval()` 返回的预分配序列值。（关于预分配序列值的说明，请参见 CREATE SEQUENCE。）

`TEMPORARY` 或 `TEMP`

删除当前会话中创建的所有临时表。

`ALL`

释放与当前会话相关的所有临时资源，并将会话重置为其初始状态。目前，这与执行下列语句序列的效果相同：

```
CLOSE ALL;
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
DISCARD SEQUENCES;
```

注解

`DISCARD ALL` 不能在事务块内部执行。

兼容性

`DISCARD` 是 PostgreSQL 扩展。

DO

DO — 执行匿名代码块

大纲

```
DO [ LANGUAGE lang_name ] code
```

描述

DO 执行匿名代码块，换言之，就是执行一个以过程语言编写的瞬时匿名函数。

该代码块会被当作一个不带参数、返回void 的函数体。它只会被解析并执行一次。

可选的 LANGUAGE 子句可以写在代码块之前，也可以写在之后。

参数

code

要执行的过程语言代码。与 CREATE FUNCTION 一样，它必须以字符串字面值形式指定。建议使用美元引用的字符串字面值。

lang_name

编写该代码所用的过程语言名称。如果省略，默认值为plpgsql。

注解

要使用的过程语言必须已经通过 CREATE EXTENSION 安装到当前数据库中。plpgsql 默认已安装，而其他语言则不会。

用户必须拥有该过程语言的 USAGE 权限；如果该语言是不受信任的，则必须是超级用户。这与用该语言创建函数时的权限要求相同。

如果在事务块中执行 DO，则该过程语言代码不能执行事务控制语句。只有当DO 在其自身事务中执行时，才允许使用事务控制语句。

示例

将模式 public 中所有视图上的全部权限授予角色 webuser：

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.table_
name) || ' TO webuser';
    END LOOP;
END$$;
```

兼容性

SQL 标准中没有 `DO` 语句。

另见

CREATE LANGUAGE

DROP ACCESS METHOD

DROP ACCESS METHOD — 移除一个访问方法

大纲

```
DROP ACCESS METHOD [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

`DROP ACCESS METHOD` 删除一个现有访问方法。只有超级用户才能删除访问方法。

参数

`IF EXISTS`

如果该访问方法不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

现有访问方法的名称。

`CASCADE`

自动删除依赖于该访问方法的对象（例如操作符类、操作符族和索引），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该访问方法，则拒绝删除它。这是默认值。

示例

删除访问方法 `heptree`：

```
DROP ACCESS METHOD heptree;
```

兼容性

`DROP ACCESS METHOD` 是一种 PostgreSQL 扩展。

另见

CREATE ACCESS METHOD

DROP AGGREGATE

DROP AGGREGATE — 移除一个聚合函数

大纲

```
DROP AGGREGATE [ IF EXISTS ] name ( aggregate_signature ) [, ...] [ CASCADE | RESTRICT ]
```

其中 *aggregate_signature* 是:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ...
 ]
```

描述

`DROP AGGREGATE` 移除一个现有聚合函数。要执行此命令，当前用户必须是该聚合函数的拥有者。

参数

`IF EXISTS`

如果该聚合函数不存在，则不要抛出错误。这种情况下会发出一个提示。

name

现有聚合函数的名称（可以用模式限定）。

argmode

参数的模式：`IN` 或 `VARIADIC`。如果省略，默认为`IN`。

argname

一个参数的名称。注意，`DROP AGGREGATE` 实际上不会关注参数名，因为确定聚合函数标识所需的只有参数数据类型。

argtype

聚合函数作用于其上的输入数据类型。要引用零参数聚合函数，请在参数说明列表的位置写上`*`。要引用有序集聚合函数，请在直接参数说明和聚合参数说明之间写上`ORDER BY`。

`CASCADE`

自动删除依赖于该聚合函数的对象（例如使用它的视图），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该聚合函数，则拒绝删除它。这是默认值。

注解

关于引用有序集聚合的其他语法，参见 ALTER AGGREGATE。

示例

要移除用于类型 `integer` 的聚合函数 `myavg`：

```
DROP AGGREGATE myavg(integer);
```

要移除假想集聚合函数 `myrank`，它接受任意个排序列组成的列表以及与之匹配的直接参数列表：

```
DROP AGGREGATE myrank(VARIADIC "any" ORDER BY VARIADIC "any");
```

在一条命令中移除多个聚合函数：

```
DROP AGGREGATE myavg(integer), myavg(bigint);
```

兼容性

SQL 标准中没有 `DROP AGGREGATE` 语句。

另见

ALTER AGGREGATE, CREATE AGGREGATE

DROP CAST

DROP CAST — 移除一个类型转换

大纲

```
DROP CAST [ IF EXISTS ] (source_type AS target_type) [ CASCADE | RESTRICT ]
```

描述

`DROP CAST` 移除一个先前定义的类型转换。

要删除一种类型转换，你必须拥有其源数据类型或目标数据类型。这与创建该类型转换所需的权限相同。

参数

`IF EXISTS`

如果该类型转换不存在，则不要抛出错误。这种情况下会发出一条提示。

`source_type`

该类型转换的源数据类型的名称。

`target_type`

该类型转换的目标数据类型的名称。

`CASCADE`
`RESTRICT`

这些关键字没有任何作用，因为没有对象依赖于类型转换。

示例

要删除从类型 `text` 到类型 `int` 的类型转换：

```
DROP CAST (text AS int);
```

兼容性

`DROP CAST` 命令符合 SQL 标准。

另见

CREATE CAST

DROP COLLATION

DROP COLLATION — 删除一个排序规则

大纲

```
DROP COLLATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

`DROP COLLATION` 删除一个先前定义的排序规则。要删除一个排序规则，你必须拥有该排序规则。

参数

`IF EXISTS`

如果该排序规则不存在，则不要抛出错误。这种情况下会发出一个提示。

name

排序规则的名称。该名称可以使用模式限定。

`CASCADE`

自动删除依赖于该排序规则的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该排序规则，则拒绝删除该排序规则。这是默认值。

示例

删除名为 `german` 的排序规则：

```
DROP COLLATION german;
```

兼容性

`DROP COLLATION` 命令符合 SQL 标准，但`IF EXISTS` 选项是 PostgreSQL 扩展。

另见

ALTER COLLATION, CREATE COLLATION

DROP CONVERSION

DROP CONVERSION — 移除一个转换

大纲

```
DROP CONVERSION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

`DROP CONVERSION` 移除一个先前定义的转换。要能够删除一个转换，你必须拥有该转换。

参数

`IF EXISTS`

如果该转换不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

转换的名称。转换名可以是模式限定的。

`CASCADE`
`RESTRICT`

这些关键字没有任何作用，因为没有对象依赖于转换。

示例

移除名为 `myname` 的转换：

```
DROP CONVERSION myname;
```

兼容性

SQL 标准中没有 `DROP CONVERSION` 语句，但有一个 `DROP TRANSLATION` 语句，它与 `CREATE TRANSLATION` 语句配套，而后者又类似于 PostgreSQL 中的`CREATE CONVERSION` 语句。

另见

ALTER CONVERSION, CREATE CONVERSION

DROP DATABASE

DROP DATABASE — 移除一个数据库

大纲

```
DROP DATABASE [ IF EXISTS ] name
```

描述

`DROP DATABASE` 删除一个数据库。它会移除数据库的系统目录条目，并删除包含数据的目录。只有数据库拥有者才能执行此命令。此外，当你或任何其他人连接到目标数据库时，都不能执行它。（请连接到`postgres` 或任何其他数据库来发出此命令。）

`DROP DATABASE` 无法撤销。请谨慎使用！

参数

`IF EXISTS`

如果该数据库不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

要移除的数据库名称。

注解

`DROP DATABASE` 不能在事务块内执行。

由于该命令不能在连接到目标数据库时执行，因此改用程序dropdb 可能更方便，它是此命令的一个包装器。

兼容性

SQL 标准中没有 `DROP DATABASE` 语句。

另见

CREATE DATABASE

DROP DOMAIN

DROP DOMAIN — 移除一个域

大纲

```
DROP DOMAIN [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP DOMAIN` 移除一个域。只有域的拥有者才能移除它。

参数

`IF EXISTS`

如果该域不存在，则不会抛出错误，而是发出一条提示。

`name`

要移除的现有域的名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该域的对象（例如表列），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该域，则拒绝删除它。这是默认值。

示例

要移除域 `box`：

```
DROP DOMAIN box;
```

兼容性

此命令符合 SQL 标准，但`IF EXISTS` 选项是 PostgreSQL 扩展。

另见

CREATE DOMAIN, ALTER DOMAIN

DROP EVENT TRIGGER

DROP EVENT TRIGGER — 移除一个事件触发器

大纲

```
DROP EVENT TRIGGER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

`DROP EVENT TRIGGER` 移除一个现有的事件触发器。要执行此命令，当前用户必须是该事件触发器的所有者。

参数

`IF EXISTS`

如果该事件触发器不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

要移除的事件触发器名称。

`CASCADE`

自动删除依赖于该触发器的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该触发器，则拒绝删除它。这是默认值。

示例

销毁事件触发器 `snitch`：

```
DROP EVENT TRIGGER snitch;
```

兼容性

SQL 标准中没有 `DROP EVENT TRIGGER` 语句。

另见

CREATE EVENT TRIGGER, ALTER EVENT TRIGGER

DROP EXTENSION

DROP EXTENSION — 移除一个扩展

大纲

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP EXTENSION` 从数据库中移除扩展。删除扩展也会删除它所包含的对象。

要使用 `DROP EXTENSION`，必须拥有该扩展。

参数

`IF EXISTS`

如果该扩展不存在，则不要抛出错误。这种情况下会发出一个提示。

name

一个已安装扩展的名称。

`CASCADE`

自动删除依赖于该扩展的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于扩展，则拒绝删除它（其自身成员对象以及同一`DROP` 命令中列出的其他扩展除外）。这是默认行为。

示例

要从当前数据库中移除扩展 `hstore`：

```
DROP EXTENSION hstore;
```

如果数据库中使用了 `hstore` 的任何对象，例如某些表具有`hstore` 类型的列，则该命令会失败。加上`CASCADE` 选项可以强制一并移除这些依赖对象。

兼容性

`DROP EXTENSION` 是 PostgreSQL 扩展。

另见

CREATE EXTENSION, ALTER EXTENSION

DROP FOREIGN DATA WRAPPER

DROP FOREIGN DATA WRAPPER — 移除一个外部数据包装器

大纲

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP FOREIGN DATA WRAPPER` 移除一个现有的外部数据包装器。要执行此命令，当前用户必须是该外部数据包装器的拥有者。

参数

`IF EXISTS`

如果该外部数据包装器不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

一个现有外部数据包装器的名称。

`CASCADE`

自动删除依赖于该外部数据包装器的对象（例如外部表和服务器），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该外部数据包装器，则拒绝删除它。这是默认值。

示例

删除外部数据包装器 `dbi`：

```
DROP FOREIGN DATA WRAPPER dbi;
```

兼容性

`DROP FOREIGN DATA WRAPPER` 符合 ISO/IEC 9075-9（SQL/MED）。`IF EXISTS` 子句是 PostgreSQL 扩展。

另见

CREATE FOREIGN DATA WRAPPER, ALTER FOREIGN DATA WRAPPER

DROP FOREIGN TABLE

DROP FOREIGN TABLE — 移除一个外部表

大纲

```
DROP FOREIGN TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP FOREIGN TABLE` 移除一个外部表。只有外部表的拥有者才能移除它。

参数

`IF EXISTS`

如果该外部表不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

要删除的外部表名称（可以被模式限定）。

`CASCADE`

自动删除依赖于该外部表的对象（例如视图），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该外部表，则拒绝删除它。这是默认值。

示例

要销毁两个外部表 `films` 和 `distributors`：

```
DROP FOREIGN TABLE films, distributors;
```

兼容性

此命令符合 ISO/IEC 9075-9（SQL/MED），但标准每条命令只允许删除一个外部表，而 `IF EXISTS` 选项是 PostgreSQL 扩展。

另见

ALTER FOREIGN TABLE, CREATE FOREIGN TABLE

DROP FUNCTION

DROP FUNCTION — 移除一个函数

大纲

```
DROP FUNCTION [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

描述

`DROP FUNCTION` 移除一个现有函数的定义。要执行此命令，用户必须是该函数的拥有者。必须指定该函数的参数类型，因为可能存在多个名称相同但参数列表不同的函数。

参数

`IF EXISTS`

如果该函数不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

现有函数的名称（可以用模式限定）。如果没有指定参数列表，这个名称必须在其模式中唯一。

`argmode`

参数的模式：`IN`、`OUT`、`INOUT` 或 `VARIADIC`。如果省略，默认值为`IN`。注意，`DROP FUNCTION` 实际上并不关注 `OUT` 参数，因为确定函数标识只需要输入参数。因此，只列出 `IN`、`INOUT` 和 `VARIADIC` 参数就足够了。

`argname`

一个参数的名称。注意，`DROP FUNCTION` 实际上并不关注参数名，因为确定函数标识只需要参数数据类型。

`argtype`

函数参数的数据类型（如果有，可以用模式限定）。

`CASCADE`

自动删除依赖于该函数的对象（例如操作符或触发器），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该函数，则拒绝删除它。这是默认值。

示例

此命令删除平方根函数：

```
DROP FUNCTION sqrt(integer);
```

在一条命令中删除多个函数：

```
DROP FUNCTION sqrt(integer), sqrt(bigint);
```

如果函数名在其模式中是唯一的，就可以在不写参数列表的情况下引用它：

```
DROP FUNCTION update_employee_salaries;
```

注意，这不同于

```
DROP FUNCTION update_employee_salaries();
```

后者引用的是一个零参数函数，而前一种写法在名称唯一时，可以引用任意参数个数的函数，包括零参数函数。

兼容性

此命令符合 SQL 标准，但有以下PostgreSQL 扩展：

- 标准每条命令只允许删除一个函数。
- `IF EXISTS` 选项是扩展。
- 允许指定参数模式和参数名是扩展。

另见

CREATE FUNCTION, ALTER FUNCTION, DROP PROCEDURE, DROP ROUTINE

DROP GROUP

DROP GROUP — 移除一个数据库角色

大纲

```
DROP GROUP [ IF EXISTS ] name [, ...]
```

描述

`DROP GROUP` 现在是 DROP ROLE 的别名。

兼容性

SQL 标准中没有 `DROP GROUP` 语句。

另见

DROP ROLE

DROP INDEX

DROP INDEX — 移除一个索引

大纲

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP INDEX` 从数据库系统中删除一个现有索引。要执行此命令，你必须是该索引的拥有者。

参数

`CONCURRENTLY`

删除索引时，不阻塞该索引所在表上的并发查询、插入、更新和删除。普通的`DROP INDEX` 会对该表获取 `ACCESS EXCLUSIVE` 锁，阻塞其他访问，直到索引删除完成。使用该选项时，命令会改为等待相冲突的事务完成。

使用此选项时有若干限制需要注意。只能指定一个索引名，并且不支持 `CASCADE` 选项。（因此，支撑`UNIQUE` 或 `PRIMARY KEY` 约束的索引不能以这种方式删除。）另外，普通的 `DROP INDEX` 可以在事务块内执行，但`DROP INDEX CONCURRENTLY` 不可以。最后，分区表上的索引也不能使用该选项删除。

对于临时表，`DROP INDEX` 始终是非并发的，因为没有其他会话可以访问它们，而且非并发删除索引的代价更低。

`IF EXISTS`

如果该索引不存在，则不要抛出错误。这种情况下会发出一条提示。

`name`

要移除的索引名称（可以被模式限定）。

`CASCADE`

自动删除依赖于该索引的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该索引，则拒绝删除它。这是默认值。

示例

此命令会删除索引 `title_idx`：

```
DROP INDEX title_idx;
```

兼容性

`DROP INDEX` 是 PostgreSQL 的一种语言扩展。SQL 标准中没有关于索引的规定。

另见

CREATE INDEX

DROP LANGUAGE

DROP LANGUAGE — 移除一种过程语言

大纲

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

`DROP LANGUAGE` 移除一种先前注册的过程语言的定义。要使用 `DROP LANGUAGE`，必须是超级用户或该语言的所有者。

> **注意**
>
> 从 PostgreSQL 9.1 起，大多数过程语言都已被实现为“扩展”，因此应当使用DROP EXTENSION 而不是 `DROP LANGUAGE` 来移除它们。

参数

`IF EXISTS`

如果该语言不存在，则不要抛出错误。这种情况下会发出一个提示。

name

现有过程语言的名称。为向后兼容，名称可以用单引号括起。

`CASCADE`

自动删除依赖于该语言的对象（例如以该语言编写的函数），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该语言，则拒绝删除它。这是默认值。

示例

此命令移除过程语言 `plsample`：

```
DROP LANGUAGE plsample;
```

兼容性

SQL 标准中没有 `DROP LANGUAGE` 语句。

另见

ALTER LANGUAGE, CREATE LANGUAGE

DROP MATERIALIZED VIEW

DROP MATERIALIZED VIEW — 移除一个物化视图

大纲

```
DROP MATERIALIZED VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP MATERIALIZED VIEW` 删除一个现有的物化视图。要执行此命令，你必须是该物化视图的拥有者。

参数

`IF EXISTS`

如果该物化视图不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

要移除的物化视图名称（可以被模式限定）。

`CASCADE`

自动删除依赖于该物化视图的对象（例如其他物化视图或普通视图），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该物化视图，则拒绝删除它。这是默认行为。

示例

这个命令将移除名为 `order_summary` 的物化视图：

```
DROP MATERIALIZED VIEW order_summary;
```

兼容性

`DROP MATERIALIZED VIEW` 是一种 PostgreSQL 扩展。

另见

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

DROP OPERATOR

DROP OPERATOR — 移除一个操作符

大纲

```
DROP OPERATOR [ IF EXISTS ] name ( { left_type | NONE } , { right_type | NONE } ) [, ...] [
 CASCADE | RESTRICT ]
```

描述

`DROP OPERATOR` 从数据库系统中删除一个现有操作符。要执行此命令，你必须是该操作符的拥有者。

参数

`IF EXISTS`

如果该操作符不存在，则不要抛出错误。这种情况下会发出一条提示。

`name`

一个现有操作符的名称（可以是模式限定的）。

`left_type`

该操作符左操作数的数据类型；如果该操作符没有左操作数，请写 `NONE`。

`right_type`

操作符右操作数的数据类型；如果操作符没有右操作数，请写`NONE`。

`CASCADE`

自动删除依赖于该操作符的对象（例如使用它的视图），以及依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该操作符，则拒绝删除它。这是默认值。

示例

要移除用于类型 `integer` 的幂操作符 `a^b`：

```
DROP OPERATOR ^ (integer, integer);
```

删除前缀一元按位取反操作符 `~b`，其类型为 `bit`：

```
DROP OPERATOR ~ (none, bit);
```

删除后缀一元阶乘操作符 `x!`，其类型为 `bigint`：

```
DROP OPERATOR ! (bigint, none);
```

在一条命令中删除多个操作符：

```
DROP OPERATOR ~ (none, bit), ! (bigint, none);
```

兼容性

SQL 标准中没有 `DROP OPERATOR` 语句。

另见

CREATE OPERATOR, ALTER OPERATOR

DROP OPERATOR CLASS

DROP OPERATOR CLASS — 移除一个操作符类

大纲

```
DROP OPERATOR CLASS [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

描述

`DROP OPERATOR CLASS` 删除一个现有操作符类。要执行此命令，你必须是该操作符类的拥有者。

`DROP OPERATOR CLASS` 不会删除该类所引用的任何操作符或函数。如果有索引依赖于该操作符类，你将需要指定`CASCADE` 才能完成删除。

参数

`IF EXISTS`

如果该操作符类不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

一个现有操作符类的名称（可以是模式限定的）。

`index_method`

该操作符类所属索引访问方法的名称。

`CASCADE`

自动删除依赖于该操作符类的对象（例如索引），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该操作符类，则拒绝删除它。这是默认值。

注解

`DROP OPERATOR CLASS` 不会删除包含该类的操作符族，即使该族中已经没有任何其他成员（尤其是在该族由`CREATE OPERATOR CLASS` 隐式创建的情况下）。空的操作符族并无害处，但为了整洁起见，你也许会希望用 `DROP OPERATOR FAMILY` 删除该族；或者更好的做法是，一开始就直接使用`DROP OPERATOR FAMILY`。

示例

移除 B-树操作符类 `widget_ops`：

```
DROP OPERATOR CLASS widget_ops USING btree;
```

如果仍有索引使用该操作符类，此命令将不会成功。加上`CASCADE` 可以在删除操作符类的同时删除这类索引。

兼容性

SQL 标准中没有 `DROP OPERATOR CLASS` 语句。

另见

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR FAMILY

DROP OPERATOR FAMILY

DROP OPERATOR FAMILY — 移除一个操作符族

大纲

```
DROP OPERATOR FAMILY [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

描述

`DROP OPERATOR FAMILY` 删除一个现有的操作符族。要执行此命令，你必须是该操作符族的拥有者。

`DROP OPERATOR FAMILY` 还会删除该族中包含的所有操作符类，但不会删除该族引用的任何操作符或函数。如果有任何索引依赖于该族内的操作符类，你需要指定`CASCADE` 才能完成删除。

参数

`IF EXISTS`

如果该操作符族不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

一个现有操作符族的名称（可以是模式限定的）。

`index_method`

该操作符族所对应的索引访问方法名称。

`CASCADE`

自动删除依赖于该操作符族的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该操作符族，则拒绝删除它。这是默认值。

示例

移除 B-树操作符族 `float_ops`：

```
DROP OPERATOR FAMILY float_ops USING btree;
```

如果仍有索引使用该族内的操作符类，此命令将不会成功。加上`CASCADE` 可以在删除操作符族的同时删除这类索引。

兼容性

SQL 标准中没有 `DROP OPERATOR FAMILY` 语句。

另见

ALTER OPERATOR FAMILY, CREATE OPERATOR FAMILY, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

DROP OWNED

DROP OWNED — 移除一个数据库角色所拥有的数据库对象

大纲

```
DROP OWNED BY { name | CURRENT_USER | SESSION_USER } [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP OWNED` 删除当前数据库中由指定角色之一拥有的所有对象。同时，授予给这些角色的、针对当前数据库中的对象或共享对象（数据库、表空间）的任何权限也会被撤销。

参数

name

一个角色的名称；该角色拥有的对象将被删除，并且授予给该角色的权限将被撤销。

`CASCADE`

自动删除依赖于受影响对象的对象，并进一步删除所有依赖于这些对象的对象（参见第 5.13 节）。

`RESTRICT`

如果有其他数据库对象依赖于某个受影响对象，则拒绝删除该角色拥有的对象。这是默认值。

注解

`DROP OWNED` 经常被用来为移除一个或多个角色做准备。因为 `DROP OWNED` 只影响当前数据库中的对象，所以通常需要在包含待移除角色所拥有对象的每个数据库中执行此命令。

使用 `CASCADE` 选项可能会使该命令递归到由其他用户拥有的对象。

REASSIGN OWNED 命令提供了另一种选择，它会重新分配由一个或多个角色拥有的所有数据库对象的拥有关系。不过，`REASSIGN OWNED` 不处理针对其他对象的权限。

由这些角色拥有的数据库和表空间不会被移除。

更多讨论请见第 21.4 节。

兼容性

`DROP OWNED` 命令是 PostgreSQL 扩展。

另见

REASSIGN OWNED, DROP ROLE

DROP POLICY

DROP POLICY — 从一个表中移除一条行级安全策略

大纲

```
DROP POLICY [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

描述

`DROP POLICY` 从表中移除指定的策略。注意，如果移除了某表的最后一条策略，并且该表仍通过`ALTER TABLE` 启用了行级安全，那么将采用默认拒绝策略。无论该表是否存在策略，都可以使用 `ALTER TABLE ... DISABLE ROW LEVEL SECURITY` 来禁用该表的行级安全。

参数

`IF EXISTS`

如果该策略不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

要删除的策略名称。

`table_name`

该策略所在表的名称（可选模式限定）。

`CASCADE`
`RESTRICT`

由于策略上不存在依赖关系，这些关键字不起任何作用。

示例

要删除表 `my_table` 上名为 `p1` 的策略：

```
DROP POLICY p1 ON my_table;
```

兼容性

`DROP POLICY` 是 PostgreSQL 扩展。

另见

CREATE POLICY, ALTER POLICY

DROP PROCEDURE

DROP PROCEDURE — 移除一个过程

大纲

```
DROP PROCEDURE [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

描述

`DROP PROCEDURE` 移除一个现有过程的定义。要执行此命令，用户必须是该过程的拥有者。必须指定过程的参数类型，因为可以存在多个名称相同但参数列表不同的过程。

参数

`IF EXISTS`

如果该过程不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

现有过程的名称（可带模式限定）。如果未指定参数列表，该名称在其模式中必须唯一。

`argmode`

参数的模式：`IN` 或 `VARIADIC`。如果省略，默认为`IN`。

`argname`

参数的名称。注意，`DROP PROCEDURE` 实际上并不关注参数名，因为只需参数的数据类型就能确定过程的身份。

`argtype`

过程参数的数据类型（可带模式限定），如果有参数的话。

`CASCADE`

自动删除依赖于该过程的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该过程，则拒绝删除它。这是默认值。

示例

```
DROP PROCEDURE do_db_maintenance();
```

兼容性

此命令符合 SQL 标准，但有以下 PostgreSQL 扩展：

- 标准每条命令只允许删除一个过程。
- `IF EXISTS` 选项是扩展。
- 允许指定参数模式和参数名是扩展。

另见

CREATE PROCEDURE, ALTER PROCEDURE, DROP FUNCTION, DROP ROUTINE

DROP PUBLICATION

DROP PUBLICATION — 移除一个发布

大纲

```
DROP PUBLICATION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP PUBLICATION` 从数据库中移除一个现有发布。

一个发布只能由其拥有者或超级用户删除。

参数

`IF EXISTS`

如果该发布不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

现有发布的名称。

`CASCADE`
`RESTRICT`

这些关键字没有任何效果，因为发布上不存在依赖关系。

示例

移除一个发布：

```
DROP PUBLICATION mypublication;
```

兼容性

`DROP PUBLICATION` 是 PostgreSQL 扩展。

另见

CREATE PUBLICATION, ALTER PUBLICATION

DROP ROLE

DROP ROLE — 移除一个数据库角色

大纲

```
DROP ROLE [ IF EXISTS ] name [, ...]
```

描述

`DROP ROLE` 移除指定的角色。要删除超级用户角色，你自己也必须是超级用户；要删除非超级用户角色，必须拥有`CREATEROLE` 权限。

如果某角色仍在该集簇的任何数据库中被引用，就不能移除它，否则会报错。在删除角色之前，必须删除它拥有的所有对象（或重新分配其拥有关系），并撤销该角色在其他对象上被授予的任何权限。REASSIGN OWNED 和 DROP OWNED 命令可用于此目的；更多讨论参见第 21.4 节。

不过，不必手工移除与该角色相关的角色成员资格；`DROP ROLE` 会自动撤销目标角色在其他角色中的任何成员资格，以及其他角色在目标角色中的任何成员资格。其他角色不会被删除，也不会受到影响。

参数

`IF EXISTS`

如果该角色不存在，则不要抛出错误。这种情况下会发出一个提示。

name

要移除的角色名称。

注解

PostgreSQL 包含一个程序 dropuser，它与此命令具有相同功能（实际上它就是调用此命令），但可以从命令 shell 中运行。

示例

删除一个角色：

```
DROP ROLE jonathan;
```

兼容性

SQL 标准定义了 `DROP ROLE`，但它一次只允许删除一个角色，并且规定的权限要求也不同于 PostgreSQL。

另见

CREATE ROLE, ALTER ROLE, SET ROLE

DROP ROUTINE

DROP ROUTINE — 移除一个例程

大纲

```
DROP ROUTINE [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

描述

`DROP ROUTINE` 移除现有例程的定义，例程可以是聚合函数、普通函数或过程。有关参数说明、更多示例和进一步细节，请参阅 DROP AGGREGATE、DROP FUNCTION 和 DROP PROCEDURE。

示例

删除参数类型为 `integer` 的例程 `foo`：

```
DROP ROUTINE foo(integer);
```

不管 `foo` 是聚合函数、普通函数还是过程，这条命令都可以工作。

兼容性

此命令符合 SQL 标准，但有以下 PostgreSQL 扩展：

- 标准每条命令只允许删除一个例程。
- `IF EXISTS` 选项是扩展。
- 允许指定参数模式和参数名是扩展。
- 聚合函数是一种扩展。

另见

DROP AGGREGATE, DROP FUNCTION, DROP PROCEDURE, ALTER ROUTINE

注意，没有`CREATE ROUTINE` 命令。

DROP RULE

DROP RULE — 移除一条重写规则

大纲

```
DROP RULE [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

描述

`DROP RULE` 移除一条重写规则。

参数

`IF EXISTS`

如果该规则不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

要移除的规则名称。

`table_name`

该规则适用的表或视图名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该规则的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该规则，则拒绝删除它。这是默认行为。

示例

要移除重写规则 `newrule`：

```
DROP RULE newrule ON mytable;
```

兼容性

`DROP RULE` 是 PostgreSQL 的一种语言扩展，整个查询重写系统也是如此。

另见

CREATE RULE, ALTER RULE

DROP SCHEMA

DROP SCHEMA — 移除一个模式

大纲

```
DROP SCHEMA [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP SCHEMA` 从数据库中移除一个或多个模式。

一个模式只能由其拥有者或超级用户删除。注意，即使该模式的拥有者并不拥有模式内的某些对象，也仍然可以删除该模式（从而删除其中的全部对象）。

参数

`IF EXISTS`

如果该模式不存在，则不要抛出错误。这种情况下会发出一个提示。

name

模式名称。

`CASCADE`

自动删除模式中包含的对象（表、函数等），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果该模式中含有任何对象，则拒绝删除它。这是默认值。

注解

使用 `CASCADE` 选项时，该命令除所命名的模式外，还可能删除其他模式中的对象。

示例

要从数据库中移除模式 `mystuff` 及其包含的全部内容：

```
DROP SCHEMA mystuff CASCADE;
```

兼容性

`DROP SCHEMA` 完全符合 SQL 标准，但标准每条命令只允许删除一个模式，另外`IF EXISTS` 选项是 PostgreSQL 扩展。

另见

ALTER SCHEMA, CREATE SCHEMA

DROP SEQUENCE

DROP SEQUENCE — 移除一个序列

大纲

```
DROP SEQUENCE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP SEQUENCE` 移除序列。只有该序列的拥有者或超级用户才能删除它。

参数

`IF EXISTS`

如果该序列不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

要删除的序列名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该序列的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该序列，则拒绝删除它。这是默认值。

示例

要移除序列 `serial`：

```
DROP SEQUENCE serial;
```

兼容性

`DROP SEQUENCE` 符合 SQL 标准，但标准每条命令只允许删除一个序列，而`IF EXISTS` 选项是 PostgreSQL 扩展。

另见

CREATE SEQUENCE, ALTER SEQUENCE

DROP SERVER

DROP SERVER — 移除一个外部服务器描述符

大纲

```
DROP SERVER [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP SERVER` 移除一个现有外部服务器描述符。要执行此命令，当前用户必须是该服务器的拥有者。

参数

`IF EXISTS`

如果该服务器不存在，则不要抛出错误。这种情况下会发出一个提示。

name

一个现有服务器的名称。

`CASCADE`

自动删除依赖于该服务器的对象（例如用户映射），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该服务器，则拒绝删除它。这是默认值。

示例

如果服务器 `foo` 存在，则删除它：

```
DROP SERVER IF EXISTS foo;
```

兼容性

`DROP SERVER` 符合 ISO/IEC 9075-9（SQL/MED）。`IF EXISTS` 子句是 PostgreSQL 扩展。

另见

CREATE SERVER, ALTER SERVER

DROP STATISTICS

DROP STATISTICS — 删除扩展统计对象

大纲

```
DROP STATISTICS [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP STATISTICS` 从数据库中删除一个或多个统计对象。只有统计对象的拥有者、模式的拥有者或超级用户才能删除统计对象。

参数

`IF EXISTS`

如果统计对象不存在，则不要抛出错误。这种情况下会发出一个提示。

name

要删除的统计对象名称（可选模式限定）。

`CASCADE`
`RESTRICT`

这些关键字没有任何作用，因为没有对象依赖于统计对象。

示例

删除不同模式中的两个统计对象；如果它们不存在也不会失败：

```
DROP STATISTICS IF EXISTS
    accounting.users_uid_creation,
    public.grants_user_role;
```

兼容性

SQL 标准中没有 `DROP STATISTICS` 命令。

另见

ALTER STATISTICS, CREATE STATISTICS

DROP SUBSCRIPTION

DROP SUBSCRIPTION — 移除一个订阅

大纲

```
DROP SUBSCRIPTION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

`DROP SUBSCRIPTION` 从数据库集簇中移除一个订阅。

只有超级用户才能删除订阅。

如果该订阅与复制槽相关联，则不能在事务块内执行 `DROP SUBSCRIPTION`。（可以使用 `ALTER SUBSCRIPTION` 来取消设置该槽。）

参数

name

要移除的订阅名称。

CASCADE
RESTRICT

这些关键字没有任何效果，因为订阅上不存在依赖关系。

注解

删除与远端主机上的复制槽相关联的订阅时（通常如此），`DROP SUBSCRIPTION` 会连接到远端主机，并尝试删除复制槽，作为其操作的一部分。这样做是为了释放远端主机上为订阅分配的资源。如果失败，无论是因为远端主机不可达，还是远端复制槽无法删除、不存在或从未存在，`DROP SUBSCRIPTION` 命令都会失败。要在这种情况下继续，首先执行`ALTER SUBSCRIPTION ... DISABLE` 禁用订阅，再执行`ALTER SUBSCRIPTION ... SET ( slot_name = NONE)`，解除订阅与复制槽的关联。此后，`DROP SUBSCRIPTION` 不再尝试对远端主机执行任何操作。注意，如果远端复制槽仍存在，应手动删除它；否则它将继续保留 WAL，最终可能导致磁盘被占满。另见第 31.2.1 节。

如果订阅与复制槽相关联，则不能在事务块内执行 `DROP SUBSCRIPTION`。

示例

删除一个订阅：

```
DROP SUBSCRIPTION mysub;
```

兼容性

`DROP SUBSCRIPTION` 是 PostgreSQL 扩展。

另见

CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

DROP TABLE

DROP TABLE — 移除一个表

大纲

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP TABLE` 从数据库中移除表。只有表拥有者、模式拥有者以及超级用户才能删除表。若要清空表中的行而不销毁该表，请使用DELETE 或 TRUNCATE。

`DROP TABLE` 总会移除目标表上存在的所有索引、规则、触发器和约束。但是，要删除一个被视图或另一张表的外键约束引用的表，必须指定`CASCADE`。（`CASCADE` 会将依赖于该表的视图整个删除，但在外键场景下，它只会删除外键约束，不会把另一张表整体删除。）

参数

`IF EXISTS`

如果表不存在，则不要抛出错误。这种情况下会发出一条提示。

`name`

要删除的表名称（可选地使用模式限定）。

`CASCADE`

自动删除依赖于该表的对象（例如视图），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该表，则拒绝删除它。这是默认值。

示例

要删除两个表 `films` 和 `distributors`：

```
DROP TABLE films, distributors;
```

兼容性

此命令符合 SQL 标准，只是标准每条命令只允许删除一个表，且`IF EXISTS` 选项是 PostgreSQL 扩展。

另见

ALTER TABLE, CREATE TABLE

DROP TABLESPACE

DROP TABLESPACE — 移除一个表空间

大纲

```
DROP TABLESPACE [ IF EXISTS ] name
```

描述

`DROP TABLESPACE` 从系统中移除一个表空间。

一个表空间只能由其拥有者或超级用户删除。删除前，该表空间中必须不含任何数据库对象。即使当前数据库中没有对象正在使用该表空间，其他数据库中的对象也可能仍然驻留在该表空间中。此外，如果任何活动会话的temp_tablespaces 设置中列出了该表空间，那么由于该表空间中可能仍有临时文件，`DROP` 也可能失败。

参数

`IF EXISTS`

如果该表空间不存在，则不要抛出错误。这种情况下会发出一个提示。

name

表空间的名称。

注解

`DROP TABLESPACE` 不能在事务块内执行。

示例

要从系统中移除表空间 `mystuff`：

```
DROP TABLESPACE mystuff;
```

兼容性

`DROP TABLESPACE` 是一种 PostgreSQL 扩展。

另见

CREATE TABLESPACE, ALTER TABLESPACE

DROP TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH CONFIGURATION — 移除一个文本检索配置

大纲

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

`DROP TEXT SEARCH CONFIGURATION` 删除一个现有文本检索配置。要执行此命令，你必须是该配置的拥有者。

参数

`IF EXISTS`

如果该文本检索配置不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

一个现有文本检索配置的名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该文本检索配置的对象，以及所有进一步依赖于这些对象的对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该文本检索配置，则拒绝删除该配置。这是默认值。

示例

移除文本检索配置 `my_english`：

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

如果已有索引在 `to_tsvector` 调用中引用了该配置，此命令将不会成功。加上`CASCADE` 可以在删除该文本检索配置的同时删除这类索引。

兼容性

SQL 标准中没有 `DROP TEXT SEARCH CONFIGURATION` 语句。

另见

ALTER TEXT SEARCH CONFIGURATION, CREATE TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH DICTIONARY

DROP TEXT SEARCH DICTIONARY — 移除一个文本检索词典

大纲

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

`DROP TEXT SEARCH DICTIONARY` 删除一个现有文本检索词典。要执行此命令，你必须是该词典的拥有者。

参数

`IF EXISTS`

如果该文本检索词典不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

一个现有文本检索词典的名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该文本检索词典的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该文本检索词典，则拒绝删除它。这是默认值。

示例

移除文本检索词典 `english`：

```
DROP TEXT SEARCH DICTIONARY english;
```

如果已有文本检索配置使用该词典，此命令将不会成功。加上`CASCADE` 可以在删除词典的同时删除这类配置。

兼容性

SQL 标准中没有 `DROP TEXT SEARCH DICTIONARY` 语句。

另见

ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY

DROP TEXT SEARCH PARSER

DROP TEXT SEARCH PARSER — 移除一个文本检索解析器

大纲

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

`DROP TEXT SEARCH PARSER` 删除一个现有文本检索解析器。要使用此命令，你必须是超级用户。

参数

`IF EXISTS`

如果该文本检索解析器不存在，则不要抛出错误。这种情况下会发出一个提示。

name

一个现有文本检索解析器的名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该文本检索解析器的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该文本检索解析器，则拒绝删除它。这是默认值。

示例

移除文本检索解析器 `my_parser`：

```
DROP TEXT SEARCH PARSER my_parser;
```

如果已有文本检索配置使用该解析器，此命令将不会成功。加上`CASCADE` 可以在删除解析器的同时删除这类配置。

兼容性

SQL 标准中没有 `DROP TEXT SEARCH PARSER` 语句。

另见

ALTER TEXT SEARCH PARSER, CREATE TEXT SEARCH PARSER

DROP TEXT SEARCH TEMPLATE

DROP TEXT SEARCH TEMPLATE — 移除一个文本检索模板

大纲

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

`DROP TEXT SEARCH TEMPLATE` 删除一个现有文本检索模板。要使用此命令，你必须是超级用户。

参数

`IF EXISTS`

如果该文本检索模板不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

一个现有文本检索模板的名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该文本检索模板的对象，以及所有进一步依赖于这些对象的对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该文本检索模板，则拒绝删除该模板。这是默认值。

示例

移除文本检索模板 `thesaurus`：

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

如果已有文本检索词典使用该模板，此命令将不会成功。加上`CASCADE` 可以在删除该模板的同时删除这类词典。

兼容性

SQL 标准中没有 `DROP TEXT SEARCH TEMPLATE` 语句。

另见

ALTER TEXT SEARCH TEMPLATE, CREATE TEXT SEARCH TEMPLATE

DROP TRANSFORM

DROP TRANSFORM — 移除一个转换

大纲

```
DROP TRANSFORM [ IF EXISTS ] FOR type_name LANGUAGE lang_name [ CASCADE | RESTRICT ]
```

描述

`DROP TRANSFORM` 移除一个先前定义的转换。

要能够删除一种转换，你必须拥有该类型和该语言。这与创建一种转换所需的权限相同。

参数

`IF EXISTS`

如果该转换不存在，则不要抛出错误。这种情况下会发出一个提示。

`type_name`

该转换的数据类型名称。

`lang_name`

该转换所用语言的名称。

`CASCADE`

自动删除依赖于该转换的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该转换，则拒绝删除它。这是默认值。

示例

要删除转换，其类型为 `hstore`，语言为 `plpythonu`：

```
DROP TRANSFORM FOR hstore LANGUAGE plpythonu;
```

兼容性

这种形式的 `DROP TRANSFORM` 是 PostgreSQL 的一种扩展。详见 CREATE TRANSFORM。

另见

CREATE TRANSFORM

DROP TRIGGER

DROP TRIGGER — 移除一个触发器

大纲

```
DROP TRIGGER [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

描述

`DROP TRIGGER` 移除一个现有触发器的定义。要执行此命令，当前用户必须是定义该触发器的表的所有者。

参数

`IF EXISTS`

如果该触发器不存在，则不要抛出错误。这种情况下会发出一个提示。

`name`

要移除的触发器的名称。

`table_name`

定义该触发器的表的名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该触发器的对象，以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该触发器，则拒绝删除它。这是默认值。

示例

删除表 `films` 上的触发器 `if_dist_exists`：

```
DROP TRIGGER if_dist_exists ON films;
```

兼容性

PostgreSQL 中的 `DROP TRIGGER` 语句与 SQL 标准不兼容。在 SQL 标准中，触发器名称不是表局部的，因此该命令仅为 `DROP TRIGGER` *`name`*。

另见

CREATE TRIGGER

DROP TYPE

DROP TYPE — 移除一个数据类型

大纲

```
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP TYPE` 移除一个用户定义的数据类型。只有该类型的拥有者才能移除它。

参数

`IF EXISTS`

如果该类型不存在，则不会抛出错误，而是发出一条提示。

`name`

要移除的数据类型的名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该类型的对象（例如表列、函数和操作符），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该类型，则拒绝删除它。这是默认值。

示例

要移除数据类型 `box`：

```
DROP TYPE box;
```

兼容性

此命令与 SQL 标准中的对应命令相似，但`IF EXISTS` 选项是 PostgreSQL 扩展。不过请注意，`CREATE TYPE` 命令的大部分内容，以及 PostgreSQL 中的数据类型扩展机制，都不同于 SQL 标准。

另见

ALTER TYPE, CREATE TYPE

DROP USER

DROP USER — 移除一个数据库角色

大纲

```
DROP USER [ IF EXISTS ] name [, ...]
```

描述

`DROP USER` 只是 DROP ROLE 的另一种拼写。

兼容性

`DROP USER` 语句是 PostgreSQL 的一种扩展。SQL 标准将用户的定义留给具体实现决定。

另见

DROP ROLE

DROP USER MAPPING

DROP USER MAPPING — 删除外部服务器的用户映射

大纲

```
DROP USER MAPPING [ IF EXISTS ] FOR { user_name | USER | CURRENT_USER | PUBLIC } SERVER server_
name
```

描述

`DROP USER MAPPING` 删除外部服务器上的一个现有用户映射。

外部服务器的拥有者可以删除任意用户到该服务器的用户映射。此外，如果某个用户已被授予该服务器上的`USAGE` 权限，那么该用户也可以删除其自己用户名对应的用户映射。

参数

`IF EXISTS`

如果该用户映射不存在，则不要报错。这种情况下会发出一个提示。

`user_name`

该映射的用户名。`CURRENT_USER` 和 `USER` 都匹配当前用户的名称。`PUBLIC` 用于匹配系统中当前及未来的所有用户名。

`server_name`

该用户映射所属服务器的名称。

示例

如果用户 `bob` 在服务器 `foo` 上的用户映射存在，则将其删除：

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

兼容性

`DROP USER MAPPING` 符合 ISO/IEC 9075-9（SQL/MED）。`IF EXISTS` 子句是 PostgreSQL 扩展。

另见

CREATE USER MAPPING, ALTER USER MAPPING

DROP VIEW

DROP VIEW — 移除一个视图

大纲

```
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP VIEW` 删除一个现有视图。要执行此命令，你必须是该视图的拥有者。

参数

`IF EXISTS`

如果视图不存在，则不抛出错误，而是发出一条提示。

`name`

要移除的视图名称（可选地使用模式限定）。

`CASCADE`

自动删除依赖于该视图的对象（例如其他视图），以及进一步依赖于这些对象的所有对象（参见第 5.13 节）。

`RESTRICT`

如果有任何对象依赖于该视图，则拒绝删除它。这是默认行为。

示例

这个命令将移除名为 `kinds` 的视图：

```
DROP VIEW kinds;
```

兼容性

此命令符合 SQL 标准，只是标准每条命令只允许删除一个视图，且`IF EXISTS` 选项是 PostgreSQL 扩展。

另见

ALTER VIEW, CREATE VIEW

END

END — 提交当前事务

大纲

```
END [ WORK | TRANSACTION ]
```

描述

`END` 提交当前事务。该事务所做的所有更改都会对其他会话可见，并且如果发生崩溃，也能保证其持久性。该命令是PostgreSQL 扩展，等效于 COMMIT。

参数

`WORK`
`TRANSACTION`

可选关键字，没有任何作用。

注解

使用 ROLLBACK 可中止一个事务。

在事务块之外发出 `END` 不会造成任何影响，但会产生一条警告消息。

示例

提交当前事务，并使所有更改永久生效：

```
END;
```

兼容性

命令 `END` 是 PostgreSQL 扩展，提供了与 SQL 标准规定的COMMIT 等效的功能。

另见

BEGIN, COMMIT, ROLLBACK

EXECUTE

EXECUTE — 执行一个预备语句

大纲

```
EXECUTE name [ ( parameter [, ...] ) ]
```

描述

EXECUTE 用于执行一个先前创建的预备语句。由于预备语句只在一个会话期间内存在，因此该预备语句必须由当前会话中较早执行的PREPARE 语句创建。

如果创建该语句的 PREPARE 语句指定了某些参数，那么传递给EXECUTE 语句的参数集合就必须与之兼容，否则将引发错误。注意，预备语句（不同于函数）不会根据其参数的类型或个数进行重载；预备语句的名称在一个数据库会话内必须唯一。

有关预备语句的创建和用法的更多信息，请参见 PREPARE。

参数

name

要执行的预备语句名称。

parameter

预备语句中某个参数的实际值。它必须是一个表达式，并且其求值结果必须与该参数的数据类型兼容，该数据类型在创建预备语句时已确定。

输出

EXECUTE 返回的命令标签是该预备语句的命令标签，而不是EXECUTE。

示例

示例位于示例一节，详见PREPARE 文档。

兼容性

SQL 标准包含 EXECUTE 语句，但它只用于嵌入式 SQL。这里的EXECUTE 语句还使用了稍有不同的语法。

另见

DEALLOCATE, PREPARE

EXPLAIN

EXPLAIN — 显示一个语句的执行计划

大纲

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

其中 *option* 为以下之一：

```
    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    BUFFERS [ boolean ]
    TIMING [ boolean ]
    SUMMARY [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

描述

这个命令显示 PostgreSQL 规划器为给定语句生成的执行计划。执行计划会显示将如何扫描该语句引用的表 – 例如普通顺序扫描、索引扫描等 –，如果引用了多个表，还会显示将使用哪些连接算法来汇集每个输入表中的所需行。

显示结果中最关键的部分是语句执行代价的估计值，它是规划器对运行该语句需要多长时间的猜测（以任意代价单位衡量，但按惯例表示磁盘页面抓取次数）。实际上会显示两个数字：返回第一行之前的启动代价，以及返回全部行的总代价。对大多数查询来说，总代价才是关键；但在某些场景中，例如EXISTS 中的子查询，规划器会选择启动代价最小而不是总代价最小的计划（因为无论如何执行器都会在得到一行后停止）。此外，如果你用LIMIT 子句限制返回的行数，规划器会在端点代价之间作适当插值，以估计哪个计划实际上最便宜。

ANALYZE 选项会让该语句被实际执行，而不仅仅是生成计划。随后会把实际运行统计信息加入显示结果，包括每个计划节点中耗费的总时间（以毫秒计）以及它实际返回的总行数。这有助于判断规划器的估计是否接近实际情况。

> **重要**
>
> 请记住，使用 ANALYZE 选项时，语句会被实际执行。虽然 EXPLAIN 会丢弃 SELECT 返回的任何输出，但语句的其他副作用仍会照常发生。如果希望使用 EXPLAIN ANALYZE 分析 INSERT, UPDATE, DELETE, CREATE TABLE AS 或 EXECUTE 语句而不让命令影响数据，可以采用以下方法：
>
> ```
> BEGIN;
> EXPLAIN ANALYZE ...;
> ROLLBACK;
> ```

不使用括号包围选项列表时，只能指定ANALYZE 和 VERBOSE 选项，而且只能按这个顺序指定。在PostgreSQL 9.0 之前，只支持这种不带括号的语法。预计所有新选项都将仅在带括号的语法中得到支持。

参数

ANALYZE

执行该命令并显示实际运行时间及其他统计信息。此参数默认为 FALSE。

VERBOSE

显示有关计划的附加信息。具体包括：计划树中每个节点的输出列列表、模式限定的表名和函数名、总是用其范围表别名标注表达式中的变量，以及总是打印显示统计信息的每个触发器名称。此参数默认为FALSE。

COSTS

包含每个计划节点的估计启动代价和总代价，以及估计行数和每行的估计宽度。此参数默认为TRUE。

BUFFERS

包含缓冲区使用信息。具体来说，会包括共享块命中、读取、写脏和写出的数量，本地块命中、读取、写脏和写出的数量，以及临时块读取和写出的数量。hit 表示该块在需要时已经在缓存中找到，因此避免了一次读取。共享块包含普通表和索引的数据；本地块包含临时表和索引的数据；临时块则包含排序、hash、Materialize 计划节点等场景使用的短期工作数据。dirtied 块数表示此查询修改的、先前未修改过的块数；written 块数表示在查询处理期间该后端从缓存中逐出的、先前已被写脏的块数。某个上层节点显示的块数包含其所有子节点使用的块数。在文本格式中，只打印非零值。此参数只能在同时启用ANALYZE 时使用。默认为 FALSE。

TIMING

在输出中包含实际启动时间以及每个节点中耗费的时间。反复读取系统时钟的开销在某些系统上可能会显著拖慢查询，因此当只需要实际行数而不需要精确时间时，把此参数设置为 FALSE 可能会有用。即使通过这个选项关闭了节点级计时，整个语句的运行时间也总会被测量。此参数只能在同时启用ANALYZE 时使用。此参数默认为 TRUE。

SUMMARY

在查询计划之后包含摘要信息（例如汇总后的时间信息）。使用 ANALYZE 时默认包含摘要信息；否则默认不包含，但可以使用此选项启用。在EXPLAIN EXECUTE 中，规划时间包括从缓存中取出计划所需的时间，以及在必要时重新规划所需的时间。

FORMAT

指定输出格式，可以是 TEXT、XML、JSON 或 YAML。非文本输出包含与文本输出相同的信息，但更容易被程序解析。此参数默认为TEXT。

boolean

指定所选选项应开启还是关闭。可以写 TRUE、ON 或 1 来启用选项，写FALSE、OFF 或 0 来禁用它。*boolean* 值也可以省略，在这种情况下假定其值为TRUE。

statement

任何你希望查看其执行计划的 SELECT、INSERT、UPDATE、DELETE、VALUES、EXECUTE、DECLARE、CREATE TABLE AS 或 CREATE MATERIALIZED VIEW AS 语句。

输出

该命令的结果是为 *statement* 选择的计划的文本描述，并可选择附带执行统计信息。第 14.1 节描述了所提供的信息。

注解

为了让 PostgreSQL 查询规划器在优化查询时能够做出相当有根据的决策，查询中用到的所有表的pg_statistic 数据都应保持最新。通常自动清理守护进程会自动处理这一点。但如果某个表最近内容发生了大量变化，你可能需要手工执行一次 ANALYZE，而不是等待自动清理跟上这些变化。

为了度量执行计划中每个节点的运行时代价，当前的EXPLAIN ANALYZE 实现会给查询执行增加性能分析开销。因此，对一个查询运行EXPLAIN ANALYZE 有时会比正常执行该查询慢得多。开销大小取决于查询的性质以及所用平台。最坏情况出现在那些自身每次执行耗时极少的计划节点上，以及获取当前时间的操作系统调用相对较慢的机器上。

示例

要显示一个只有单个 integer 列且包含 10000 行的表上的简单查询计划：

```
EXPLAIN SELECT * FROM foo;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

下面是同一查询，但使用 JSON 输出格式：

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
           QUERY PLAN
--------------------------------
 [                              +
   {                            +
     "Plan": {                  +
       "Node Type": "Seq Scan",+
       "Relation Name": "foo", +
       "Alias": "foo",         +
       "Startup Cost": 0.00,   +
       "Total Cost": 155.00,   +
       "Plan Rows": 10000,     +
       "Plan Width": 4         +
     }                          +
   }                            +
 ]
(1 row)
```

如果存在索引，并且我们使用了带有可索引WHERE 条件的查询，EXPLAIN 可能会显示不同的计划：

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                         QUERY PLAN
-------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

下面是同一查询，但采用 YAML 格式：

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
--------------------------------
 - Plan:                       +
     Node Type: "Index Scan"  +
     Scan Direction: "Forward"+
     Index Name: "fi"         +
     Relation Name: "foo"     +
     Alias: "foo"             +
     Startup Cost: 0.00       +
     Total Cost: 5.98         +
     Plan Rows: 1             +
     Plan Width: 4            +
     Index Cond: "(i = 4)"
(1 row)
```

XML 格式留给读者自行练习。

下面是同一计划，但隐藏了代价估计：

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;

        QUERY PLAN
----------------------------
 Index Scan using fi on foo
   Index Cond: (i = 4)
(2 rows)
```

下面是使用聚合函数的查询计划示例：

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                              QUERY PLAN
----------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

下面是使用 `EXPLAIN EXECUTE` 显示预备查询执行计划的示例：

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                     QUERY PLAN

-------------------------------------------------------------------------------------------
-------------------------
 HashAggregate  (cost=9.54..9.54 rows=1 width=8) (actual time=0.156..0.161 rows=11 loops=1)
   Group Key: foo
   ->  Index Scan using test_pkey on test  (cost=0.29..9.29 rows=50 width=8) (actual time=0.
039..0.091 rows=99 loops=1)
         Index Cond: ((id > $1) AND (id < $2))
 Planning time: 0.197 ms
 Execution time: 0.225 ms
(6 rows)
```

当然，这里显示的具体数字取决于所涉及表的实际内容。还要注意，由于规划器的改进，这些数字甚至所选查询策略在PostgreSQL 的不同版本之间都可能发生变化。此外，`ANALYZE` 命令使用随机采样来估计数据统计信息；因此，即使表中数据的实际分布没有变化，在重新运行一次 `ANALYZE` 之后，代价估计也可能发生变化。

兼容性

SQL 标准中没有定义 `EXPLAIN` 语句。

另见

ANALYZE

FETCH

FETCH — 使用游标从查询中检索行

大纲

```
FETCH [ direction ] [ FROM | IN ] cursor_name
```

其中 *direction* 可以是以下之一：

```
NEXT
PRIOR
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
BACKWARD
BACKWARD count
BACKWARD ALL
```

描述

FETCH 使用先前创建的游标检索行。

游标带有一个关联位置，FETCH 会使用该位置。游标位置可以位于查询结果的第一行之前、结果中的任意一行上，或者位于结果的最后一行之后。游标创建时位于第一行之前。提取若干行后，游标会定位在最近提取的那一行上。如果FETCH 越过了可用行的末尾，那么游标会留在最后一行之后；如果是向后提取，则会留在第一行之前。FETCH ALL 或者 FETCH BACKWARD ALL 总是会让游标位于最后一行之后或第一行之前。

NEXT、PRIOR、FIRST、LAST、ABSOLUTE、RELATIVE 这些形式会在适当地移动游标后提取一行。如果不存在这样的行，则返回空结果，并根据情况将游标定位在第一行之前或最后一行之后。

使用 FORWARD 和 BACKWARD 的形式会在向前或向后的方向上提取指定数量的行，并将游标定位在最后返回的那一行上（如果*count* 超过可用行数，则定位在所有行之后或之前）。

RELATIVE 0、FORWARD 0 以及 BACKWARD 0 都会请求提取当前行而不移动游标，也就是重新提取最近一次提取的行。除非游标位于第一行之前或最后一行之后，否则该操作都会成功；在这两种情况下，不会返回任何行。

> **注意**
>
> 本页面描述的是 SQL 命令层面的游标用法。如果想要在 PL/pgSQL 函数中使用游标，规则会有所不同 – 见第 43.7.3 节。

参数

direction

direction 定义提取方向以及要提取的行数。它可以是下列之一：

NEXT

提取下一行。如果省略 *direction*，这将是默认值。

PRIOR

提取前一行。

FIRST

提取查询的第一行（与 ABSOLUTE 1 相同）。

LAST

提取查询的最后一行（与 ABSOLUTE -1 相同）。

ABSOLUTE *count*

提取查询的第 *count* 行；如果 *count* 为负，则提取从末尾算起的第abs(*count*) 行。如果 *count* 超出范围，则定位在第一行之前或最后一行之后。特别地，ABSOLUTE 0 会定位在第一行之前。

RELATIVE *count*

提取后面的第 *count* 行；如果 *count* 为负，则提取前面的第abs(*count*) 行。RELATIVE 0 会重新提取当前行（如果有）。

count

提取接下来的 *count* 行（与 FORWARD *count* 相同）。

ALL

提取所有剩余行（与 FORWARD ALL 相同）。

FORWARD

提取下一行（与 NEXT 相同）。

FORWARD *count*

提取接下来的 *count* 行。FORWARD 0 会重新提取当前行。

FORWARD ALL

提取所有剩余行。

BACKWARD

提取前一行（与 PRIOR 相同）。

BACKWARD *count*

提取前面的 *count* 行（反向扫描）。BACKWARD 0 会重新提取当前行。

BACKWARD ALL

提取之前的所有行（反向扫描）。

count

count 是一个可带符号的整数常量，用于确定要提取的位置或行数。对于 FORWARD 和 BACKWARD 这两种情况，指定负的*count* 等效于颠倒 FORWARD 和 BACKWARD 的方向。

cursor_name

一个已打开游标的名称。

输出

成功完成时，FETCH 命令会返回如下形式的命令标签：

```
FETCH count
```

其中 *count* 是提取到的行数（可能为零）。注意，在psql 中，命令标签实际上不会显示，因为 psql 会改为显示提取到的行。

注解

如果打算使用除 FETCH NEXT 或带正数计数的 FETCH FORWARD 之外的任何 FETCH 变体，则应当在声明游标时使用SCROLL 选项。对于简单查询，PostgreSQL 允许从未使用 SCROLL 声明的游标向后提取，但最好不要依赖这种行为。如果游标声明为 NO SCROLL，则不允许向后提取。

ABSOLUTE 提取并不比通过相对移动到达目标行更快：无论如何，底层实现都必须遍历所有中间行。负值的绝对提取甚至更糟：必须先把查询读到末尾以找到最后一行，然后再从那里反向遍历。不过，回卷到查询起始处（如FETCH ABSOLUTE 0）是很快的。

DECLARE 用于定义游标。使用 MOVE 可以在不检索数据的情况下改变游标位置。

示例

下面的示例展示了如何使用游标遍历一个表：

```
BEGIN WORK;

-- 建立一个游标：
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- 从游标 liahona 中提取前 5 行：
FETCH FORWARD 5 FROM liahona;

 code  |          title           | did | date_prod  |   kind   |  len
-------+--------------------------+-----+------------+----------+-------
 BL101 | The Third Man            | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen        | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme  | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                  | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                   | 103 | 1964-02-03 | Drama    | 02:28

-- 提取前一行：
FETCH PRIOR FROM liahona;

 code  |  title  | did | date_prod  |  kind  |  len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- 关闭游标并结束事务：
CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL 标准只为嵌入式 SQL 定义了 FETCH。这里描述的 FETCH 变体会像 SELECT 结果那样返回数据，而不是把数据放入主变量中。除此之外，FETCH 与 SQL 标准完全向上兼容。

涉及 FORWARD 和 BACKWARD 的 FETCH 形式，以及形式FETCH *count* 和 FETCH ALL（其中隐含了 FORWARD）都是 PostgreSQL 扩展。

SQL 标准只允许在游标名之前使用 FROM；允许使用 IN，或者将两者都完全省略，都是扩展。

另见

CLOSE, DECLARE, MOVE

GRANT

GRANT — 定义访问权限

大纲

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } routine_name [ ( [ [ argmode ] [ arg_name ] arg_
type [, ...] ] ) ] [, ...]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT role_name [, ...] TO role_specification [, ...]
    [ WITH ADMIN OPTION ]
```

```
    [ GRANTED BY role_specification ]
```

其中 *role_specification* 可以是：

```
    [ GROUP ] role_name
  | PUBLIC
  | CURRENT_USER
  | SESSION_USER
```

描述

GRANT 命令有两个基本变体：一种是在数据库对象（表、列、视图、外部表、序列、数据库、外部数据包装器、外部服务器、函数、过程、过程语言、模式或表空间）上授予权限，另一种是授予角色成员资格。这两种变体在许多方面相似，但差异也足够大，因此分别说明。

在数据库对象上 GRANT

这种 GRANT 命令变体将数据库对象上的特定权限授予一个或多个角色。如果此前已经授予过某些权限，新授予的权限会加到现有权限之上。

还可以选择在一个或多个模式中，对同一类型的所有对象授予权限。目前仅对表、序列、函数和过程支持这一功能。ALL TABLES 也会影响视图和外部表，这与针对特定对象的 GRANT 命令相同。ALL FUNCTIONS 也会影响聚合函数，但不包括过程，这同样与针对特定对象的 GRANT 命令一致。

关键字 PUBLIC 表示要把权限授予所有角色，包括以后可能创建的角色。PUBLIC 可以视为一个隐式定义的组，并且始终包含所有角色。任何特定角色实际拥有的权限，是直接授予给它的权限、授予给它当前所属任一角色的权限，以及授予给 PUBLIC 的权限之和。

如果指定了 WITH GRANT OPTION，权限接收者随后可以再把该权限授予其他人。没有授予选项时，接收者不能这样做。授予选项不能授予给 PUBLIC。

没有必要向对象拥有者（通常是创建它的用户）授予权限，因为拥有者默认拥有全部权限。（不过，出于安全考虑，拥有者也可以选择撤销自己的某些权限。）

删除对象或以任何方式更改其定义的权利，不被视为一种可授予的权限；它是拥有者固有的，不能被授予或撤销。（不过，可以通过授予或撤销拥有该对象的角色的成员资格，获得类似效果；见下文。）拥有者还隐式拥有该对象上的全部授予选项。

PostgreSQL 会将某些类型对象上的默认权限授予 PUBLIC。默认情况下，不会在表、表列、序列、外部数据包装器、外部服务器、大对象、模式或表空间上向PUBLIC 授予任何权限。对于其他类型的对象，授予PUBLIC 的默认权限如下：数据库上的 CONNECT 和 TEMPORARY（创建临时表）权限；函数和过程上的 EXECUTE 权限；以及语言和数据类型（包括域）上的 USAGE 权限。当然，对象所有者可以使用REVOKE 撤销默认权限和显式授予的权限。（为了尽可能安全，请在创建对象的同一事务中执行REVOKE，这样其他用户就没有可以使用该对象的时间窗口。）此外，还可以使用ALTER DEFAULT PRIVILEGES 命令更改这些初始默认权限设置。

可用权限如下：

SELECT

> 允许使用 SELECT 读取指定表、视图或序列的任何列，或列出的特定列。还允许使用COPY TO。在 UPDATE 或 DELETE 中引用现有列值时也需要此权限。对于序列，此权限还允许使用 currval 函数。对于大对象，此权限允许读取该对象。

INSERT

允许使用 INSERT 向指定表插入新行。如果列出了特定列，则INSERT 命令只能给这些列赋值（其他列因此会获得默认值）。还允许使用 COPY FROM。

UPDATE

允许使用 UPDATE 更新指定表的任何列，或列出的特定列。（实际上，任何非简单的 UPDATE 命令还需要 SELECT 权限，因为它必须引用表列，以确定要更新的行和/或计算列的新值。）SELECT ... FOR UPDATE 和 SELECT ... FOR SHARE 除了需要 SELECT 权限外，也要求至少在一列上具有此权限。对于序列，此权限允许使用nextval 和 setval 函数。对于大对象，此权限允许写入或截断该对象。

DELETE

允许使用 DELETE 从指定表删除一行。（实际上，任何非简单的DELETE 命令还需要 SELECT 权限，因为它必须引用表列，以确定要删除的行。）

TRUNCATE

允许在指定表上使用 TRUNCATE。

REFERENCES

允许创建引用指定表或表中特定列的外键约束。（参见 CREATE TABLE 语句。）

TRIGGER

允许在指定表上创建触发器。（参见 CREATE TRIGGER 语句。）

CREATE

对于数据库，允许在数据库中创建新的模式和发布。

对于模式，允许在模式中创建新对象。要重命名已有对象，必须拥有该对象并且具有其所在模式的此权限。

对于表空间，允许在其中创建表、索引和临时文件，也允许创建以该表空间为默认表空间的数据库。（注意，撤销此权限不会改变现有对象的存放位置。）

CONNECT

允许用户连接到指定数据库。此权限在连接启动时检查（此外还会检查 pg_hba.conf 施加的任何限制）。

TEMPORARY
TEMP

允许在使用指定数据库时创建临时表。

EXECUTE

允许使用指定的函数或过程，以及基于该函数实现的任何操作符。这是唯一适用于函数和过程的权限类型。FUNCTION 语法也适用于聚合函数。也可以使用 ROUTINE 来引用函数、聚合函数或过程，而不必区分其具体类型。

USAGE

对于过程语言，允许使用指定语言创建以该语言编写的函数。这是唯一适用于过程语言的权限类型。

对于模式，允许访问指定模式中包含的对象（假定也满足对象自身的权限要求）。本质上，这允许被授权者“查找”模式内的对象。没有此权限，仍然可能看到对象名称，例如通过查询系统表。此外，撤销此权限后，现有后端中可能仍有先前已执行过这种查找的语句，因此这并不是阻止对象访问的完全安全的方法。

对于序列，此权限允许使用currval 和 nextval 函数。

对于类型和域，此权限允许在创建表、函数和其他模式对象时使用该类型或域。（注意，它不控制该类型的一般“使用”，例如在查询中出现该类型的值。它仅阻止创建依赖该类型的对象。此权限的主要目的是控制哪些用户可以创建对某个类型的依赖，因为这些依赖可能使所有者以后无法更改该类型。）

对于外部数据包装器，此权限允许使用该外部数据包装器创建新的服务器。

对于服务器，此权限允许使用该服务器创建外部表。被授权者还可以创建、修改或删除与该服务器关联的自己的用户映射。

ALL PRIVILEGES

一次授予所有可用权限。PRIVILEGES 关键字在 PostgreSQL 中是可选的，但在严格 SQL 中是必需的。

其他命令所需的权限列在各自命令的参考页面上。

角色上的 GRANT

这种 GRANT 命令变体把一个角色的成员资格授予一个或多个其他角色。角色成员资格之所以重要，是因为它会把授予该角色的权限传递给其每个成员。

如果指定了 WITH ADMIN OPTION，成员就可以将该角色的成员资格继续授予其他人，也可以撤销该角色的成员资格。没有管理选项时，普通用户不能这样做。一个角色不被认为在其自身上持有WITH ADMIN OPTION，但是在会话用户与该角色相符的数据库会话中，它可以把自身角色的成员资格授予其他角色，或撤销这种资格。数据库超级用户可以向任何人授予或撤销任何角色的成员资格。具有CREATEROLE 权限的角色可以授予或撤销任何非超级用户角色的成员资格。

如果指定了 GRANTED BY，该授权会记录为由指定角色执行。只有数据库超级用户可以使用此选项，除非指定的是执行命令的同一角色。

与权限的情况不同，角色成员资格不能授予给 PUBLIC。还要注意，这种形式的命令不允许在 *role_specification* 中使用噪声词 GROUP。

注解

REVOKE 命令用于撤销访问权限。

从 PostgreSQL 8.1 起，用户和组的概念已统一为一种称为角色的单一实体。因此，不再需要使用关键字 GROUP 来标识被授权者是用户还是组。GROUP 仍可出现在命令中，但它只是一个噪声词。

如果用户对某一列本身，或者对其所在整张表拥有该权限，就可以在该列上执行 SELECT、INSERT 等操作。在表级授予某项权限后，再在单列上撤销该权限，并不会产生人们可能期望的效果：表级授权不会受到列级操作的影响。

当对象的非拥有者试图在该对象上执行 GRANT 时，如果该用户在该对象上完全没有任何权限，命令会立即失败。只要有某项权限可用，命令就会继续执行，但只会授予那些该用户持有授予选项的权限。如果未持有任何授予选项，GRANT ALL PRIVILEGES 形式会发出警告；而其他形式如果命令中特别列出的任一权限未持有其授予选项，也会发出警告。（原则上，这些说明也适用于对象拥有者；但由于拥有者总是被视为持有全部授予选项，这种情况实际上不会发生。）

需要注意，数据库超级用户可以访问所有对象，而不受对象权限设置的影响。这可类比于 Unix 系统中的 root 权限。和 root 一样，除非绝对必要，否则不宜以超级用户身份操作。

如果超级用户选择执行 GRANT 或 REVOKE 命令，该命令会像由受影响对象的拥有者发出一样执行。特别是，通过这种命令授予的权限看起来会像是由对象拥有者授予的。（对于角色成员资格，则看起来像是由被授予成员资格的角色本身授予的。）

GRANT 和 REVOKE 也可以由并非受影响对象拥有者的角色执行，只要该角色是拥有该对象之角色的成员，或者是持有该对象上 WITH GRANT OPTION 权限之角色的成员。在这种情况下，权限会记录为由实际拥有该对象的角色，或者由持有 WITH GRANT OPTION 权限的角色授予。例如，如果表 t1 由角色 g1 拥有，而角色 u1 是 g1 的成员，那么 u1 可以把 t1 上的权限授予给 u2，但这些权限看起来会像是直接由 g1 授予的。角色 g1 的任何其他成员之后都可以撤销这些权限。

如果执行 GRANT 的角色通过多条角色成员资格路径间接持有所需权限，则将哪个上层角色记录为授权者并无规定。在这种情况下，最佳做法是使用 SET ROLE 切换成你希望作为其身份执行 GRANT 的那个具体角色。

在表上授予权限，并不会自动把权限扩展到该表使用的任何序列，包括绑定到 SERIAL 列的序列。序列上的权限必须单独设置。

使用 psql 的 \dp 命令可以获取表和列的现有权限信息。例如：

```
=> \dp mytable
                              Access privileges
 Schema |  Name   | Type  |   Access privileges   | Column access privileges
--------+---------+-------+-----------------------+--------------------------
 public | mytable | table | miriam=arwdDxt/miriam | col1:
                          : =r/miriam             :   miriam_rw=rw/miriam
                          : admin=arw/miriam
(1 row)
```

由 \dp 显示的条目解释如下：

```
rolename=xxxx -- 授予某个角色的权限
        =xxxx -- 授予 PUBLIC 的权限

            r -- SELECT (读取)
            w -- UPDATE (写入)
            a -- INSERT (追加)
            d -- DELETE
            D -- TRUNCATE
            x -- REFERENCES
            t -- TRIGGER
            X -- EXECUTE
            U -- USAGE
            C -- CREATE
            c -- CONNECT
            T -- TEMPORARY
      arwdDxt -- ALL PRIVILEGES (适用于表，其他对象有所不同)
            * -- 前述权限的授予选项

        /yyyy -- 授予此权限的角色
```

上例中的显示结果会出现在用户 miriam 创建表 mytable 并执行以下命令之后：

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

对于非表对象，还有其他\d 命令可以显示其权限。

如果某个对象的 “Access privileges” 列为空，表示该对象具有默认权限（即其权限列为 null）。默认权限始终包括所有者的全部权限，并且可能根据对象类型包含授予PUBLIC 的某些权限，如上所述。在对象上首次执行GRANT 或 REVOKE 时，会先实例化默认权限（例如生成{miriam=arwdDxt/miriam}），然后根据指定请求修改它们。同样，“Column access privileges” 中也只会显示具有非默认权限的列的条目。（注意：此处的 “默认权限” 始终指该对象类型的内置默认权限。权限受 ALTER DEFAULT PRIVILEGES 命令影响的对象，始终会显示显式权限条目，其中包含ALTER 的效果。）

注意所有者的隐式授予选项没有在访问权限显示中标记。仅当授予选项被显式授予给某人时才会出现*。

示例

将表 films 上的插入权限授予所有用户：

```
GRANT INSERT ON films TO PUBLIC;
```

将视图 kinds 上的所有可用权限授予用户 manuel：

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

请注意，如果上述命令由超级用户或 kinds 的拥有者执行，确实会授予所有权限；但如果由其他人执行，则只会授予该执行者持有授予选项的那些权限。

将角色 admins 的成员资格授予用户 joe：

```
GRANT admins TO joe;
```

兼容性

根据 SQL 标准，ALL PRIVILEGES 中的 PRIVILEGES 关键字是必需的。SQL 标准也不支持每条命令对多个对象设置权限。

PostgreSQL 允许对象拥有者撤销自己的普通权限：例如，表拥有者可以通过撤销自己的 INSERT、UPDATE、DELETE 和 TRUNCATE 权限，使该表对自己变成只读。这在 SQL 标准中是不可能的。原因是 PostgreSQL 把拥有者的权限视为拥有者授予给自己的；因此他们也可以撤销这些权限。在 SQL 标准中，拥有者的权限由一个假定实体 “_SYSTEM” 授予。由于拥有者并不是 “_SYSTEM”，因此不能撤销这些权利。

根据 SQL 标准，授予选项可以授予给 PUBLIC；PostgreSQL 只支持将授予选项授予给角色。

SQL 标准允许将 GRANTED BY 选项用于所有形式的 GRANT。PostgreSQL 仅在授予角色成员资格时支持它，即使如此，也只有超级用户可以用它指定其他授权者。

SQL 标准还为其他种类的对象提供 USAGE 权限：字符集、排序规则、翻译。

在 SQL 标准中，序列只有 USAGE 这一项权限，它控制 NEXT VALUE FOR 表达式的使用；该表达式等价于 PostgreSQL 中的 nextval 函数。序列上的 SELECT 和 UPDATE 权限都是 PostgreSQL 扩展。把序列的 USAGE 权限应用到 currval 函数上也是 PostgreSQL 扩展（该函数本身也是扩展）。

数据库、表空间、模式和语言上的权限都是 PostgreSQL 扩展。

另见

REVOKE, ALTER DEFAULT PRIVILEGES

IMPORT FOREIGN SCHEMA

IMPORT FOREIGN SCHEMA — 从一个外部服务器导入表定义

大纲

```
IMPORT FOREIGN SCHEMA remote_schema
    [ { LIMIT TO | EXCEPT } ( table_name [, ...] ) ]
    FROM SERVER server_name
    INTO local_schema
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

描述

`IMPORT FOREIGN SCHEMA` 创建外部表，用来表示存在于外部服务器上的表。新的外部表将归发出该命令的用户所有，并使用与远程表相匹配的正确列定义和选项创建。

默认情况下，外部服务器上某个特定模式中的所有表和视图都会被导入。也可以把待导入表的列表限制为指定的子集，或者排除某些特定表。新的外部表都会创建在目标模式中，该模式必须已经存在。

要使用 `IMPORT FOREIGN SCHEMA`，用户必须在外部服务器上拥有`USAGE` 权限，并在目标模式上拥有`CREATE` 权限。

参数

remote_schema

要从中导入的远程模式。远程模式的具体含义取决于所使用的外部数据包装器。

`LIMIT TO ( table_name [, ...] )`

只导入名称与给定表名之一匹配的外部表。远程模式中的其他表将被忽略。

`EXCEPT ( table_name [, ...] )`

将指定的外部表排除在导入之外。除这里列出的表外，远程模式中的其他表都会被导入。

server_name

要从中导入的外部服务器。

local_schema

将在其中创建所导入外部表的模式。

`OPTIONS ( option 'value' [, ...] )`

导入期间要使用的选项。允许的选项名和值取决于各个外部数据包装器。

示例

从服务器 film_server 上的远程模式 foreign_films 导入表定义，并在本地模式films 中创建这些外部表：

```
IMPORT FOREIGN SCHEMA foreign_films
    FROM SERVER film_server INTO films;
```

与上例相同，但只导入actors 和 directors 这两个表（如果它们存在）：

```
IMPORT FOREIGN SCHEMA foreign_films LIMIT TO (actors, directors)
    FROM SERVER film_server INTO films;
```

兼容性

IMPORT FOREIGN SCHEMA 命令符合 SQL 标准，但OPTIONS 子句是 PostgreSQL 扩展。

另见

CREATE FOREIGN TABLE, CREATE SERVER

INSERT

INSERT — 在表中插入新行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER } VALUE ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
    [ ON CONFLICT [ conflict_target ] conflict_action ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

其中 *conflict_target* 可以是以下之一:

```
    ( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [, ...] )
 [ WHERE index_predicate ]
    ON CONSTRAINT constraint_name
```

其中 *conflict_action* 是以下之一:

```
    DO NOTHING
    DO UPDATE SET { column_name = { expression | DEFAULT } |
                    ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] ) |
                    ( column_name [, ...] ) = ( sub-SELECT )
                  } [, ...]
              [ WHERE condition ]
```

描述

INSERT 将新行插入表中。可以插入由值表达式指定的一行或多行，也可以插入由查询产生的零行或多行。

目标列名可以按任意顺序列出。如果根本未给出列名列表，则默认使用按声明顺序排列的全部列；或者如果VALUES 子句或 *query* 只提供了 *N* 列，则默认使用按声明顺序排列的前*N* 个列名。VALUES 子句或 *query* 提供的值，会按从左到右的顺序与显式或隐式列列表对应起来。

任何未出现在显式或隐式列列表中的列都会填入默认值；如果没有声明默认值，则填入空值。

如果任何列的表达式数据类型不正确，将尝试自动进行类型转换。

对没有唯一索引的表执行 INSERT 不会被并发活动阻塞。若并发会话执行的操作会锁定或修改与待插入唯一索引值匹配的行，则带有唯一索引的表可能发生阻塞；详见第 61.5 节。ON CONFLICT 可用于指定一种替代动作，而不是报出违反唯一约束或排他约束的错误。（见下文 ON CONFLICT 子句。）

可选的 RETURNING 子句使 INSERT 基于每个实际插入的行（若使用了 ON CONFLICT DO UPDATE 子句，则也可能是更新后的行）计算并返回一个或多个值。这主要用于获取由默认值提供的值，例如 serial 序列号。不过，也允许使用任何引用该表列的表达式。RETURNING 列表的语法与 SELECT 的输出列表相同。只有成功插入或更新的行才会被返回。例如，如果某一行被锁定，但由于不满足 ON CONFLICT DO UPDATE ... WHERE 子句中的 *condition* 而未被更新，则该行不会被返回。

要向表中插入行，必须具有该表上的INSERT 权限。如果存在 ON CONFLICT DO UPDATE 子句，还要求具有该表上的UPDATE 权限。

如果指定了列列表，你只需要对所列列具有INSERT 权限。类似地，在指定 ON CONFLICT DO UPDATE 时，你只需要对列出要更新的列具有UPDATE 权限。不过，ON CONFLICT DO UPDATE 还要求具有 SELECT 权限，涵盖ON CONFLICT DO UPDATE 表达式或 *condition* 中读取其值的任何列。

使用 RETURNING 子句要求对 RETURNING 中提到的所有列具有 SELECT 权限。如果使用 *query* 子句将查询结果中的行插入表中，那么当然还需要对查询中用到的任何表或列具有SELECT 权限。

参数

插入

本节介绍仅在插入新行时可用的参数。专门用于 ON CONFLICT 子句的参数将单独说明。

with_query

WITH 子句允许指定一个或多个子查询，这些子查询可以在INSERT 查询中按名称引用。详见第 7.8 节和 SELECT。

query（SELECT 语句）本身也可以包含 WITH 子句。在这种情况下，*query* 中可以引用两组 *with_query*，但由于第二组嵌套得更近，它具有更高的优先级。

table_name

现有表的名称（可选地使用模式限定）。

alias

table_name 的替代名称。提供别名后，它会完全隐藏表的实际名称。当 ON CONFLICT DO UPDATE 的目标是一个名为 excluded 的表时，这一点特别有用，因为否则该名称会被当作表示拟插入行的那个特殊表名。

column_name

名为 *table_name* 的表中某一列的名称。如有需要，列名可以附带子字段名或数组下标。（只向复合列的部分字段插入值时，其余字段将为空值。）在 ON CONFLICT DO UPDATE 中引用列时，不要在目标列的指定中包含表名。例如，INSERT INTO table_name ... ON CONFLICT DO UPDATE SET table_name.col = 1 是无效的（这与 UPDATE 的一般行为一致）。

OVERRIDING SYSTEM VALUE

如果没有这个子句，那么为定义为GENERATED ALWAYS 的标识列指定显式值（除 DEFAULT 外）会报错。这个子句会覆盖该限制。

OVERRIDING USER VALUE

如果指定了此子句，则会忽略为定义为GENERATED BY DEFAULT 的标识列提供的任何值，并应用默认的序列生成值。

例如，在表之间复制值时，这个子句很有用。写成 INSERT INTO tbl2 OVERRIDING USER VALUE SELECT * FROM tbl1 将从 tbl1 复制所有在 tbl2 中不是标识列的列，而 tbl2 中标识列的值将由与 tbl2 关联的序列生成。

`DEFAULT VALUES`

所有列都会填入各自的默认值。（这种形式不允许出现 `OVERRIDING` 子句。）

`expression`

赋给相应列的表达式或值。

`DEFAULT`

相应列将填入其默认值。

`query`

提供要插入行的查询（`SELECT` 语句）。其语法说明请参见 SELECT。

`output_expression`

在每一行插入或更新后由 `INSERT` 命令计算并返回的表达式。该表达式可以使用由 *`table_name`* 指定的表中的任何列。写成`*` 可返回插入或更新后行的全部列。

`output_name`

用于返回列的名称。

ON CONFLICT 子句

可选的 `ON CONFLICT` 子句指定一种替代动作，用来替代抛出唯一约束或排他约束违背错误。对于每一条拟插入的行，要么插入继续进行；要么如果违反了由*`conflict_target`* 指定的某个仲裁约束或索引，就执行替代的 *`conflict_action`*。`ON CONFLICT DO NOTHING` 的替代动作只是跳过该行的插入。`ON CONFLICT DO UPDATE` 的替代动作则是更新与拟插入行冲突的现有行。

`conflict_target` 可以进行唯一索引推断。执行推断时，它由一个或多个 *`index_column_name`* 列和/或 *`index_expression`* 表达式，以及可选的*`index_predicate`* 组成。所有在不考虑顺序的情况下恰好包含 *`conflict_target`* 指定列/表达式的 *`table_name`* 唯一索引，都会被推断（选中）为仲裁索引。如果指定了 *`index_predicate`*，则作为推断的进一步要求，候选仲裁索引还必须满足该谓词。注意，这意味着如果存在某个满足其他所有条件的非部分唯一索引（即没有谓词的唯一索引），那么该索引也会被推断出来（从而被 `ON CONFLICT` 使用）。如果推断尝试失败，则会报错。

`ON CONFLICT DO UPDATE` 保证得到原子的 `INSERT` 或 `UPDATE` 结果；只要没有其他独立错误，即使在高并发下，也能保证结果是这两者之一。这也称为UPSERT – “UPDATE or INSERT”。

`conflict_target`

通过选择仲裁索引来指定 `ON CONFLICT` 对哪些冲突采取替代动作。它要么执行唯一索引推断，要么显式命名一个约束。对于`ON CONFLICT DO NOTHING`，是否指定 *`conflict_target`* 是可选的；省略时，将处理与所有可用约束（以及唯一索引）的冲突。对于 `ON CONFLICT DO UPDATE`，则必须提供 *`conflict_target`*。

conflict_action

conflict_action 指定一个替代的 ON CONFLICT 动作。它可以是 DO NOTHING，也可以是 DO UPDATE 子句，用以精确指定发生冲突时要执行的UPDATE 动作细节。在 ON CONFLICT DO UPDATE 中，SET 和 WHERE 子句既可以通过表名（或别名）访问现有行，也可以通过特殊表 excluded 访问拟插入的行。对于任何被读取的 excluded 列，必须对目标表中的对应列具有SELECT 权限。

注意，所有行级BEFORE INSERT 触发器的效果都会反映在 excluded 值中，因为这些效果可能促成该行被排除在插入之外。

index_column_name

table_name 中某一列的名称。用于推断仲裁索引。遵循 CREATE INDEX 的格式。要求对 *index_column_name* 具有 SELECT 权限。

index_expression

与 *index_column_name* 类似，但用于推断出现在索引定义中的、基于*table_name* 列的表达式（而非简单列）。遵循 CREATE INDEX 的格式。要求对出现在 *index_expression* 中的任何列具有 SELECT 权限。

collation

指定时，要求相应的*index_column_name* 或 *index_expression* 必须使用特定的排序规则，才能在推断时匹配。通常会省略，因为排序规则通常不会影响是否发生约束违背。遵循 CREATE INDEX 格式。

opclass

指定时，要求相应的*index_column_name* 或 *index_expression* 必须使用特定的操作符类，才能在推断时匹配。通常会省略，因为一种类型的各个操作符类在相等性语义上往往是等价的，或者只需相信已定义的唯一索引具有所需的相等性定义即可。遵循 CREATE INDEX 格式。

index_predicate

用于允许推断部分唯一索引。任何满足该谓词的索引（实际上不一定是部分索引）都可以被推断。遵循 CREATE INDEX 格式。要求对出现在 *index_predicate* 中的任何列具有 SELECT 权限。

constraint_name

显式按名称指定一个作为仲裁的约束，而不是通过推断约束或索引。

condition

返回 boolean 值的表达式。只有使该表达式返回 true 的行才会被更新，不过一旦采取 ON CONFLICT DO UPDATE 动作，所有行都会被锁定。请注意，只有在某个冲突已被识别为更新候选后，才会最后计算*condition*。

请注意，排他约束不支持在ON CONFLICT DO UPDATE 中充当仲裁对象。在所有情况下，只有 NOT DEFERRABLE 约束和唯一索引可作为仲裁对象。

带有 ON CONFLICT DO UPDATE 子句的 INSERT 是一种“确定性”语句。这意味着不允许该命令对任何单个现有行产生多于一次的影响；一旦出现这种情况，就会报出基数违背错误。拟插入的行在受仲裁索引或约束限制的属性上不应彼此重复。

请注意，目前尚不支持在针对分区表的INSERT 的 ON CONFLICT DO UPDATE 子句中更新冲突行的分区键，从而需要将该行移动到一个新分区。

> **提示**
>
> 通常更推荐使用唯一索引推断，而不是通过 ON CONFLICT ON CONSTRAINT *constraint_name* 直接命名约束。当底层索引以重叠方式被另一个大致等价的索引替换时，推断仍能继续正确工作，例如在删除待替换索引之前先使用 CREATE UNIQUE INDEX ... CONCURRENTLY。

输出

成功完成时，INSERT 命令返回以下形式的命令标签：

```
INSERT oid count
```

其中，*count* 是插入或更新的行数。如果 *count* 恰好为一，且目标表有 OID，则 *oid* 表示分配给插入行的 OID。该单行必须是插入的，而不是更新的。否则 *oid* 为零。

如果 INSERT 命令包含 RETURNING 子句，则结果将类似于一个SELECT 语句，其中包含 RETURNING 列表定义的列和值，它们是基于该命令插入或更新的行计算出来的。

注解

如果指定的表是分区表，则每一行都会被路由到适当的分区并插入其中。如果指定的表本身就是一个分区，则只要输入行中有任意一行违反该分区约束，就会报错。

示例

向表 films 插入一行：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

在这个示例中，省略了len 列，因此它将取默认值：

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

这个示例对日期列使用 DEFAULT 子句，而不是显式指定值：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

要插入一行，其所有列都由默认值构成：

```
INSERT INTO films DEFAULT VALUES;
```

使用多行 VALUES 语法插入多行：

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

这个示例从与 films 列布局相同的 tmp_films 表中选出一些行并插入到 films 表：

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

这个示例向数组列插入值：

```
-- 为井字棋创建一个空的 3x3 棋盘
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- 上例中的下标实际上并非必需
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

向表 distributors 插入一行，并返回由 DEFAULT 子句生成的序号：

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

为负责 Acme Corporation 账户的销售人员增加销量计数，并把整个更新后的行连同当前时间记录到日志表中：

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

视情况插入或更新新的分销商。假设已经定义了一个唯一索引，用于约束出现在did 列中的值。请注意，特殊的 excluded 表用于引用最初拟插入的值：

```
INSERT INTO distributors (did, dname)
    VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
    ON CONFLICT (did) DO UPDATE SET dname = EXCLUDED.dname;
```

插入一个分销商；如果已存在一条会导致待插入行被排除的现有行（即在行级 BEFORE INSERT 触发器触发后，其受约束列与待插入行匹配的行），则对该待插入行不做任何操作。示例假设已经定义了一个唯一索引，用于约束出现在did 列中的值：

```
INSERT INTO distributors (did, dname) VALUES (7, 'Redline GmbH')
    ON CONFLICT (did) DO NOTHING;
```

视情况插入或更新新的分销商。示例假设已经定义了一个唯一索引，用于约束出现在did 列中的值。这里使用 WHERE 子句来限制实际会被更新的行（不过，任何未更新的现有行仍会被锁定）：

```
-- 不更新位于某个邮政编码区域内的现有分销商
INSERT INTO distributors AS d (did, dname) VALUES (8, 'Anvil Distribution')
    ON CONFLICT (did) DO UPDATE
    SET dname = EXCLUDED.dname || ' (formerly ' || d.dname || ')'
    WHERE d.zipcode <> '21201';

-- 在语句中直接指定约束名称（使用关联的
-- 索引来仲裁是否采取 DO NOTHING 动作）
INSERT INTO distributors (did, dname) VALUES (9, 'Antwerp Design')
    ON CONFLICT ON CONSTRAINT distributors_pkey DO NOTHING;
```

如果可能就插入新的分销商；否则执行 DO NOTHING。示例假设已经定义了一个唯一索引，用于约束这样一个行子集上的did 列值：其中 is_active 布尔列计算结果为 true：

```
-- 这条语句可以推断出 "did" 上以 "WHERE is_active"
-- 为谓词的部分唯一索引，但也可以直接使用
-- "did" 上的普通唯一约束
INSERT INTO distributors (did, dname) VALUES (10, 'Conrad International')
    ON CONFLICT (did) WHERE is_active DO NOTHING;
```

兼容性

INSERT 符合 SQL 标准，但 RETURNING 子句是 PostgreSQL 扩展，在 INSERT 中使用 WITH 的能力以及使用 ON CONFLICT 指定替代动作的能力也都是扩展。此外，标准不允许省略列名列表却又不是所有列都由VALUES 子句或 *query* 填充的情况。

SQL 标准规定，只有当存在一个定义为总是生成值的标识列时，才能指定 OVERRIDING SYSTEM VALUE。而 PostgreSQL 在任何情况下都允许该子句，并在它不适用时忽略它。

query 子句可能存在的限制见 SELECT。

LISTEN

LISTEN — 监听通知

大纲

```
LISTEN channel
```

描述

LISTEN 将当前会话注册为名为 *channel* 的通知通道的监听者。如果当前会话已经在该通知通道上注册为监听者，则不会执行任何操作。

每当命令 NOTIFY *channel* 被执行时，无论是由当前会话还是由另一个连接到同一数据库的会话执行，所有当前正在监听该通知通道的会话都会收到通知，而每个会话随后都会通知其所连接的客户端应用。

会话可以使用 UNLISTEN 命令取消在给定通知通道上的监听注册。会话结束时，其监听注册会被自动清除。

客户端应用必须采用何种方法来检测通知事件，取决于它所使用的 PostgreSQL 应用程序编程接口。使用 libpq 库时，应用程序将LISTEN 作为普通 SQL 命令发出，然后必须定期调用 PQnotifies 函数，以判断是否已经收到任何通知事件。其他接口，例如libpgtcl，则提供了处理通知事件的更高层方法；事实上，在libpgtcl 中，应用程序员甚至不应直接发出LISTEN 或 UNLISTEN。详见所使用接口的相关文档。

NOTIFY 中对 LISTEN 和 NOTIFY 的使用有更详细的讨论。

参数

channel

通知通道的名称（任意标识符）。

注解

LISTEN 会在事务提交时生效。如果在随后被回滚的事务中执行了 LISTEN 或 UNLISTEN，则被监听的通知通道集合不会发生变化。

执行过 LISTEN 的事务不能进入两阶段提交的预备状态。

示例

在 psql 中配置并执行一组 listen/notify 操作：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

兼容性

SQL 标准中没有 `LISTEN` 语句。

另见

NOTIFY, UNLISTEN

LOAD

LOAD — 载入共享库文件

大纲

```
LOAD 'filename'
```

描述

该命令将一个共享库文件载入 PostgreSQL 服务器的地址空间。如果该文件已经载入过，则此命令不执行任何操作。包含 C 函数的共享库文件会在调用其中某个函数时自动载入。因此，显式执行LOAD 通常只在要载入的是通过“钩子”修改服务器行为、而不是提供一组函数的库时才有需要。

库文件名通常只给出文件名本身，系统会在服务器的库搜索路径中查找它（该路径由 dynamic_library_path 设置）。也可以给出完整路径名。无论哪种情况，都可以省略平台标准的共享库文件名扩展。关于此主题的更多信息，见第 38.10.1 节。

非超级用户只能将 LOAD 用于位于 $libdir/plugins/中的库文件，也就是说，指定的 *filename* 必须严格以该字符串开头。（确保那里只安装了“安全”的库，是数据库管理员的责任。）

兼容性

LOAD 是 PostgreSQL 的一种扩展。

另见

CREATE FUNCTION

LOCK

LOCK — 锁定表

大纲

```
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]
```

其中 *lockmode* 可以是以下之一：

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

描述

LOCK TABLE 获取一个表级锁；如有必要，会等待任何冲突锁被释放。如果指定了NOWAIT，LOCK TABLE 就不会等待获取所需的锁：如果无法立即获得，命令将被中止并报错。锁一旦获得，就会一直持有到当前事务结束。（没有UNLOCK TABLE 命令；锁总是在事务结束时释放。）

锁定一个视图时，出现在该视图定义查询中的所有关系也会以相同的锁模式递归地被锁定。

在为引用表的命令自动获取锁时，PostgreSQL 总是尽可能使用限制最少的锁模式。LOCK TABLE 适用于可能需要更严格锁定的场景。例如，假设某个应用在READ COMMITTED 隔离级别下运行事务，并且需要确保某个表中的数据在整个事务期间保持稳定。要做到这一点，可以在查询之前先对该表获取SHARE 锁模式。这样可以阻止并发的数据更改，并确保后续对该表的读取看到已提交数据的稳定视图，因为SHARE 锁模式与写入者获取的 ROW EXCLUSIVE 锁冲突，而LOCK TABLE *name* IN SHARE MODE 语句会一直等待，直到所有并发持有 ROW EXCLUSIVE 模式锁的事务提交或回滚。因此，一旦获得该锁，就不存在尚未提交的写入；而且在释放该锁之前，也不会有新的写入开始。

若要在 REPEATABLE READ 或 SERIALIZABLE 隔离级别的事务中达到类似效果，你必须在执行任何SELECT 或数据修改语句之前执行 LOCK TABLE 语句。REPEATABLE READ 或 SERIALIZABLE 事务的数据视图，会在其第一条SELECT 或数据修改语句开始时冻结。在事务稍后再执行 LOCK TABLE 仍然可以阻止并发写入 – 但它不能保证该事务读取到的是最新已提交的值。

如果这类事务还要修改表中的数据，那么它应使用SHARE ROW EXCLUSIVE 锁模式，而不是SHARE 模式。这样可以确保同一时间只有一个这类事务在运行。否则就可能发生死锁：两个事务都可能先获得 SHARE 模式，然后都无法再获得实际执行更新所需的ROW EXCLUSIVE 模式。（注意，事务自己的锁永远不会互相冲突，因此事务在持有SHARE 模式时仍可获得 ROW EXCLUSIVE 模式，但前提是没有其他人持有SHARE 模式。）为避免死锁，要确保所有事务都按相同顺序对相同对象获取锁；如果同一对象需要多种锁模式，则事务应始终先获取限制最严格的模式。

关于锁模式和锁策略的更多信息，请参见第 13.3 节。

参数

name

要锁定的现有表的名称（可选模式限定）。如果在表名前指定了 ONLY，则只有该表会被锁定。如果未指定ONLY，则该表及其所有后代表（如果有）都会被锁定。也可以在表名后指定 *，以显式表明包含后代表。

命令 LOCK TABLE a, b;等效于 LOCK TABLE a; LOCK TABLE b;。这些表会按 LOCK TABLE 命令中指定的顺序逐个锁定。

lockmode

锁模式指定该锁会与哪些锁冲突。锁模式见第 13.3 节。

如果未指定锁模式，则使用限制最严格的ACCESS EXCLUSIVE 模式。

NOWAIT

指定 LOCK TABLE 不等待任何冲突锁被释放：如果指定的锁无法在不等待的情况下立即获得，事务就会中止。

注解

LOCK TABLE ... IN ACCESS SHARE MODE 要求对目标表具有 SELECT 权限。LOCK TABLE ... IN ROW EXCLUSIVE MODE 要求对目标表具有 INSERT、UPDATE、DELETE 或 TRUNCATE 权限。所有其他形式的 LOCK 都要求具有表级 UPDATE、DELETE 或 TRUNCATE 权限。

在视图上执行锁定操作的用户必须对该视图拥有相应权限。此外，视图所有者必须对底层基关系拥有相关权限，而执行锁定操作的用户不需要对底层基关系拥有任何权限。

LOCK TABLE 在事务块外毫无用处：锁只会一直持有到该语句结束。因此，如果在事务块外使用 LOCK，PostgreSQL 会报告错误。请使用 BEGIN 和 COMMIT（或 ROLLBACK）来定义事务块。

LOCK TABLE 只处理表级锁，因此名称中带有ROW 的模式其实都不准确。这些模式名称通常应理解为：用户打算在被锁定的表中获取行级锁。此外，ROW EXCLUSIVE 模式本身也是一种可共享的表锁。请记住，就LOCK TABLE 而言，所有锁模式的语义完全相同，差别只在于哪些模式彼此冲突。关于如何获取真正的行级锁，请参阅第 13.3.2 节以及锁定子句一节（位于 SELECT 参考文档中）。

示例

在准备向外键表执行插入时，在主键表上获取一个SHARE 锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- 如果未返回记录则执行 ROLLBACK
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

在准备执行删除操作时，在主键表上获取一个SHARE ROW EXCLUSIVE 锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

兼容性

SQL 标准中没有 LOCK TABLE，而是使用SET TRANSACTION 来指定事务的并发级别。PostgreSQL 也支持这一点；详见 SET TRANSACTION。

除 ACCESS SHARE、ACCESS EXCLUSIVE 和 SHARE UPDATE EXCLUSIVE 锁模式外，PostgreSQL 的锁模式和 LOCK TABLE 语法与 Oracle 中的对应语法兼容。

MOVE

MOVE — 定位游标

大纲

```
MOVE [ direction ] [ FROM | IN ] cursor_name
```

其中 *direction* 可以是以下之一：

```
    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
    BACKWARD
    BACKWARD count
    BACKWARD ALL
```

描述

`MOVE` 在不检索任何数据的情况下重新定位游标。`MOVE` 与 `FETCH` 命令的工作方式完全相同，只是它只定位游标而不返回行。

`MOVE` 命令的参数与 `FETCH` 命令完全相同；有关语法和用法的细节，请参阅 FETCH。

输出

成功完成时，`MOVE` 命令会返回如下形式的命令标签：

```
MOVE count
```

其中 *count* 是用相同参数执行 `FETCH` 命令时本应返回的行数（可能为零）。

示例

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- 跳过前 5 行:
MOVE FORWARD 5 IN liahona;
MOVE 5

-- 从游标 liahona 中提取第 6 行:
FETCH 1 FROM liahona;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- 关闭游标 liahona 并结束事务:
CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL 标准中没有 MOVE 语句。

另见

CLOSE, DECLARE, FETCH

NOTIFY

NOTIFY — 发出一个通知

大纲

```
NOTIFY channel [ , payload ]
```

描述

NOTIFY 命令向每个客户端应用发送一个通知事件，并可附带一个可选的“载荷”字符串；这些客户端应用此前都已在当前数据库中针对指定的通道名执行过 LISTEN *channel*。通知对所有用户可见。

NOTIFY 为访问同一个 PostgreSQL 数据库的一组进程提供了一种简单的进程间通信机制。通知中可以附带载荷字符串；如果需要传递结构化数据，也可以借助数据库中的表，将附加数据从通知发送者传递给一个或多个监听者，从而构建用于传递结构化数据的更高层机制。

传递给客户端的通知事件信息包括：通知通道名、发出通知的会话对应的服务器进程PID，以及载荷字符串。如果未指定载荷字符串，则该字符串为空串。

在某个数据库中使用哪些通道名以及各自含义，由数据库设计者自行决定。常见的做法是让通道名与数据库中的某个表同名，而通知事件基本上意味着：“我改动了这张表，去看看有什么新变化”。不过，NOTIFY 和 LISTEN 命令并不会强制这种关联。例如，数据库设计者可以使用多个不同的通道名，来标识同一张表上的不同类型变更；或者也可以利用载荷字符串区分不同情况。

当 NOTIFY 用于表明某张特定表发生变化时，一种很有用的编程技巧是把NOTIFY 放进由表更新触发的语句级触发器中。这样一来，每当表发生变化时就会自动发出通知，应用程序员也不容易忘记这么做。

NOTIFY 会以几种重要方式与 SQL 事务交互。首先，如果在事务内部执行NOTIFY，那么只有在事务提交之后，通知事件才会被递送；如果事务被中止，则其中所有命令都不会生效，NOTIFY 也不例外。这种行为是合理的，但如果你期望通知立即送达，可能会感到意外。其次，如果某个正在监听的会话在事务内部收到了通知信号，那么在该事务结束（提交或中止）之前，通知事件都不会递送给它所连接的客户端。原因同样在于：如果通知在事务内部就已递送，而该事务后来又被中止，我们会希望通知也能被撤销，但服务器一旦把通知发送给客户端，就无法再把它“收回”。因此，通知事件只会在事务之间递送。由此得出的结论是，使用NOTIFY 做实时信号的应用，应尽量让事务保持短小。

如果在同一个事务中，针对同一通道名多次发送完全相同载荷字符串的通知，那么数据库服务器可以决定只递送一个通知。另一方面，载荷字符串不同的通知始终会作为不同通知递送。类似地，来自不同事务的通知也绝不会被折叠为一个通知。除了会丢弃重复通知中较后的那些实例之外，NOTIFY 还保证来自同一事务的通知会按发送顺序递送；来自不同事务的消息，则会按事务提交顺序递送。

执行 NOTIFY 的客户端自己同时也在监听同一通知通道，这是很常见的情形。在这种情况下，它会像其他监听会话一样收到一个通知事件。根据应用逻辑，这可能导致无用功，例如再次读取自己刚刚写入更新的数据库表。要避免这种额外工作，可以检查通知事件消息中提供的发出通知的服务器进程PID，是否与当前会话自身的PID（可通过 libpq 获得）相同。如果二者相同，就说明这是当前会话自己发出的通知回送给自己，可以直接忽略。

参数

channel

要发信号的通知通道名称（任意标识符）。

payload

随通知一起传递的“载荷”字符串。它必须指定为简单的字符串字面值。在默认配置下，它必须短于 8000 字节。（如果需要传递二进制数据或大量信息，最好把它们放入数据库表中，并发送该记录的键。）

注解

系统中有一个队列，用来保存那些已经发送但尚未被所有监听会话处理的通知。如果这个队列被塞满，那么调用NOTIFY 的事务会在提交时失败。该队列非常大（标准安装中为 8GB），几乎足以满足所有用例。不过，如果某个会话执行了LISTEN 后又长时间停留在一个事务里，就无法进行清理。一旦队列占用达到一半，你就会在日志文件中看到警告，指出究竟是哪个会话阻碍了清理。在这种情况下，应确保该会话结束其当前事务，以便清理能够继续进行。

函数 pg_notification_queue_usage 返回当前被待处理通知占用的队列比例。详见第 9.25 节。

执行过 NOTIFY 的事务不能进入两阶段提交的预备状态。

pg_notify

要发送通知，也可以使用函数pg_notify(text, text)。该函数的第一个参数是通道名，第二个参数是载荷。如果需要处理非常量的通道名和载荷，它会比NOTIFY 命令更容易使用。

示例

在 psql 中配置并执行一组 listen/notify 操作：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received from server
 process with PID 8448.

LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server process with PID
 14728.
```

兼容性

SQL 标准中没有 NOTIFY 语句。

另见

LISTEN, UNLISTEN

PREPARE

PREPARE — 预备一个语句以供执行

大纲

```
PREPARE name [ ( data_type [, ...] ) ] AS statement
```

描述

PREPARE 创建一个预备语句。预备语句是一种服务器端对象，可用于优化性能。执行 PREPARE 语句时，指定的语句会被解析、分析并重写。随后发出 EXECUTE 命令时，该预备语句会被规划并执行。这种分工避免了重复的解析分析工作，同时又允许执行计划依赖于所提供的特定参数值。

预备语句可以带参数，也就是在执行时会代入语句中的值。创建预备语句时，可以按位置引用参数，例如 $1、$2 等。也可以选择指定相应的参数数据类型列表。如果某个参数的数据类型未指定，或被声明为 unknown，则其类型会从该参数第一次被引用时所在的上下文中推断出来（如果可能）。执行该语句时，需要在 EXECUTE 语句中为这些参数提供实际值。有关详情，参见 EXECUTE。

预备语句只在当前数据库会话期间存在。会话结束时，预备语句就会被遗忘，因此再次使用前必须重新创建。这也意味着单个预备语句不能由多个并发的数据库客户端共用；不过，每个客户端都可以创建自己要使用的预备语句。预备语句也可以用DEALLOCATE 命令手工释放。

当单个会话要执行大量相似语句时，预备语句可能带来最大的性能优势。如果语句在规划或重写时比较复杂，例如查询涉及许多表的连接，或者需要应用多条规则，那么这种性能差异会特别明显。如果语句相对容易规划和重写，但执行本身代价较高，则预备语句带来的性能优势就不那么明显。

参数

name

为这个特定预备语句指定的任意名称。它在单个会话内必须唯一，随后用于执行或释放先前已准备的语句。

data_type

预备语句中某个参数的数据类型。如果某个参数的数据类型未指定，或被指定为 unknown，则其类型会从该参数第一次被引用时所在的上下文中推断出来。要在预备语句本身中引用参数，可使用 $1、$2 等。

statement

任何 SELECT、INSERT、UPDATE、DELETE 或 VALUES 语句。

注解

预备语句可以使用通用计划，而不必为每组提供的EXECUTE 值重新规划。对于没有参数的预备语句，会立即使用通用计划；否则，只有在执行五次或更多次后，所生成计划的平均估计代价（包括规划开销）高于通用计划的估计代价时，才会使用通用计划。一旦选定通用计划，在该预备语句的剩余生命周期内都会使用它。如果EXECUTE 提供的值在含有大量重复值的列中很少见，生成的自定义计划的代价可能比通用计划低得多，即使加上规划开销也是如此，以至于可能永远不会使用通用计划。

通用计划假定提供给 EXECUTE 的每个值都是该列的非重复值之一，并且列值均匀分布。例如，如果统计信息记录该列有三个非重复值，通用计划就假定对该列进行相等比较会匹配所处理行的33%。列统计信息还使通用计划能够准确计算唯一列的选择率。对分布不均匀的列进行比较以及指定不存在的值，会影响平均计划代价，进而影响是否以及何时选用通用计划。

要检查 PostgreSQL 为预备语句使用的查询计划，请使用 EXPLAIN，例如EXPLAIN EXECUTE。如果使用的是通用计划，其中会包含参数符号*$n*，而自定义计划中会代入所提供的参数值。通用计划中的行数估计反映了为这些参数计算的选择率。

有关查询规划以及 PostgreSQL 为此收集的统计信息的更多内容，请参见ANALYZE 文档。

尽管预备语句的主要目的在于避免对语句重复进行解析分析和规划，但PostgreSQL 会在使用前强制重新分析并重新规划该语句，只要其中使用的数据库对象自上次使用该预备语句以来发生了定义（DDL）变更。此外，如果search_path 的值在两次使用之间发生变化，该语句也会基于新的 search_path 重新解析。（后一种行为是从 PostgreSQL 9.3 开始引入的。）这些规则使得使用预备语句在语义上几乎等同于反复重新提交同一段查询文本，同时在对象定义未改变时仍能获得性能收益，特别是在跨多次使用时最佳计划保持不变的情况下。这种语义等价性并不完全成立。举例来说，如果语句以未限定名引用某个表，随后又在 search_path 中更靠前的某个模式里创建了一个同名新表，那么不会自动触发重新解析，因为语句所使用的对象本身并未改变。不过，如果后来由于其他变更而触发了重新解析，后续使用中引用的就会是这个新表。

可以通过查询 pg_prepared_statements 系统视图，查看当前会话中所有可用的预备语句。

示例

为一个 INSERT 语句创建一个预备语句，然后执行它：

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

为一个 SELECT 语句创建预备语句，然后执行它：

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

注意，第二个参数的数据类型未指定，因此会从 $2 所在的使用上下文中推断出来。

兼容性

SQL 标准包含 `PREPARE` 语句，但它只用于嵌入式 SQL。这里的`PREPARE` 语句在语法上也略有不同。

另见

DEALLOCATE, EXECUTE

PREPARE TRANSACTION

PREPARE TRANSACTION — 为两阶段提交准备当前事务

大纲

```
PREPARE TRANSACTION transaction_id
```

描述

PREPARE TRANSACTION 为两阶段提交准备当前事务。执行此命令后，该事务将不再与当前会话相关联；相反，它的状态会被完整地存储到磁盘上，因此即使在请求提交之前数据库发生崩溃，它也极有可能成功提交。

事务进入预备状态后，稍后可以分别使用COMMIT PREPARED 或 ROLLBACK PREPARED 来提交或回滚。发出这些命令的不必是执行原始事务的那个会话，任何会话都可以这样做。

从发出该命令的会话来看，PREPARE TRANSACTION 颇似 ROLLBACK：执行之后，将不再有活动的当前事务，并且该预备事务的效果也不再可见。（如果该事务随后被提交，这些效果会再次可见。）

如果 PREPARE TRANSACTION 因任何原因失败，它就等同于执行了一次ROLLBACK：当前事务会被取消。

参数

transaction_id

一个任意标识符，后续可通过它在COMMIT PREPARED 或 ROLLBACK PREPARED 中标识该事务。该标识符必须写成字符串字面值，长度必须小于 200 字节，并且不能与当前任何已处于预备状态的事务标识符相同。

注解

PREPARE TRANSACTION 并非供应用程序或交互式会话使用。它的目的是让外部事务管理器能够跨多个数据库或其他事务性资源执行原子性的全局事务。除非是在编写事务管理器，否则通常不应使用PREPARE TRANSACTION。

该命令必须在事务块内部使用。请用 BEGIN 启动事务块。

当前不允许对执行过以下任一操作的事务执行 PREPARE：涉及临时表或会话临时命名空间的操作、创建任何WITH HOLD 游标，或执行过LISTEN、UNLISTEN 或 NOTIFY。这些特性与当前会话绑定得过于紧密，因此在要进入预备状态的事务中没有意义。

如果该事务曾用 SET 修改过任何运行时参数（且未使用 LOCAL 选项），那么这些效果在执行PREPARE TRANSACTION 之后仍会保留，并且不会受到后续任何COMMIT PREPARED 或 ROLLBACK PREPARED 的影响。因此，仅就这一点而言，PREPARE TRANSACTION 的行为更像 COMMIT 而不是 ROLLBACK。

当前所有处于预备状态的事务都列在 pg_prepared_xacts 系统视图中。

> **小心**
>
> 让事务长时间停留在预备状态并不明智。这会妨碍 VACUUM 回收存储空间，在极端情况下甚至可能导致数据库为防止事务 ID 回卷而关闭（见第 24.1.5 节）。还要记住，该事务会继续持有它原本持有的所有锁。此功能的预期用法是：一旦外部事务管理器确认其他数据库也已准备好提交，就尽快提交或回滚该预备事务。
>
> 如果没有设置外部事务管理器来跟踪预备事务并确保它们能被及时结束，最好将 max_prepared_transactions 设为零，从而禁用预备事务功能。这样可以防止意外创建预备事务，而这些事务随后可能被遗忘并最终引发问题。

示例

为两阶段提交准备当前事务，并使用foobar 作为事务标识符：

```
PREPARE TRANSACTION 'foobar';
```

兼容性

PREPARE TRANSACTION 是 PostgreSQL 扩展。它供外部事务管理系统使用，其中有些系统受标准约束（例如 X/Open XA），但这些系统的 SQL 侧并未标准化。

另见

COMMIT PREPARED, ROLLBACK PREPARED

REASSIGN OWNED

REASSIGN OWNED — 更改一个数据库角色所拥有的数据库对象的所有权

大纲

```
REASSIGN OWNED BY { old_role | CURRENT_USER | SESSION_USER } [, ...]
                  TO { new_role | CURRENT_USER | SESSION_USER }
```

描述

`REASSIGN OWNED` 指示系统将任何 *`old_roles`* 所拥有的数据库对象的所有权更改为 *`new_role`*。

参数

`old_role`

一个角色的名称。该角色在当前数据库中拥有的所有对象，以及该角色拥有的所有共享对象（数据库、表空间）的所有权，都将被重新分配给 *`new_role`*。

`new_role`

将成为受影响对象新拥有者的角色名称。

注解

`REASSIGN OWNED` 常用于为移除一个或多个角色做准备。由于 `REASSIGN OWNED` 不会影响其他数据库中的对象，因此通常需要在每个包含待移除角色所拥有对象的数据库中执行此命令。

执行 `REASSIGN OWNED` 要求你同时是源角色和目标角色的成员。

DROP OWNED 命令提供了另一种选择，它会直接删除一个或多个角色所拥有的全部数据库对象。

`REASSIGN OWNED` 命令不会影响授予 *`old_roles`* 的、在非其所拥有的对象上的任何权限。同样，它也不会影响使用`ALTER DEFAULT PRIVILEGES` 创建的默认权限。如需撤销这类权限，请使用`DROP OWNED`。

更多讨论请见第 21.4 节。

兼容性

`REASSIGN OWNED` 命令是 PostgreSQL 扩展。

另见

DROP OWNED, DROP ROLE, ALTER DATABASE

REFRESH MATERIALIZED VIEW

REFRESH MATERIALIZED VIEW — 替换物化视图的内容

大纲

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name
    [ WITH [ NO ] DATA ]
```

描述

`REFRESH MATERIALIZED VIEW` 会完全替换物化视图的内容。要执行此命令，必须是该物化视图的拥有者。旧内容会被丢弃。如果指定了`WITH DATA`（或者默认如此），则会执行其底层查询以生成新数据，并使物化视图处于可扫描状态。如果指定了`WITH NO DATA`，则不会生成新数据，并使物化视图处于不可扫描状态。

不得同时指定 `CONCURRENTLY` 和 `WITH NO DATA`。

参数

`CONCURRENTLY`

刷新物化视图时，不会阻塞针对该物化视图的并发查询。如果不使用此选项，一次影响很多行的刷新通常会占用更少的资源并且更快完成，但可能会阻塞其他试图从该物化视图读取的连接。在只影响少量行的情况下，此选项可能更快。

只有当该物化视图上至少有一个仅使用列名且涵盖所有行的 `UNIQUE` 索引时，才允许使用此选项；也就是说，它不能是表达式索引，也不能带有`WHERE` 子句。

只有在该物化视图已经填充数据时，才能使用此选项。

即使使用了此选项，对于任意一个物化视图，一次也只能运行一个 `REFRESH`。

name

要刷新的物化视图名称（可以带模式限定）。

注解

如果物化视图的定义查询中带有 `ORDER BY` 子句，则该物化视图的原始内容会按该顺序排列；但 `REFRESH MATERIALIZED VIEW` 并不保证保留这种顺序。

示例

这个命令会使用物化视图 `order_summary` 定义中的查询来替换其内容，并使其处于可扫描状态：

```
REFRESH MATERIALIZED VIEW order_summary;
```

这个命令会释放与物化视图 annual_statistics_basis 相关的存储，并使其处于不可扫描状态：

```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH NO DATA;
```

兼容性

REFRESH MATERIALIZED VIEW 是 PostgreSQL 的扩展。

另见

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, DROP MATERIALIZED VIEW

REINDEX

REINDEX — 重建索引

大纲

```
REINDEX [ ( VERBOSE ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } name
```

描述

REINDEX 使用索引所属表中存储的数据重建索引，并替换索引的旧副本。以下几种场景适合使用 REINDEX：

- 一个索引已经损坏，不再包含有效数据。尽管理论上这不该发生，但在实践中索引可能因软件缺陷或硬件故障而损坏。REINDEX 提供了一种恢复方法。
- 一个索引已经“膨胀”，也就是说其中包含许多空页或几乎为空的页。在 PostgreSQL 中，B-树索引在某些不常见的访问模式下可能出现这种情况。REINDEX 可通过写入一个不含死页的新版本索引来减少索引的空间消耗。详见第 24.2 节。
- 你修改了某个索引的存储参数（例如 fillfactor），并希望确保该更改已经完全生效。
- 使用 CONCURRENTLY 选项构建索引失败后，会留下一个“无效”索引。这类索引没有用处，但用 REINDEX 重建它们可能很方便。注意，REINDEX 不会进行并发构建。要在不干扰生产运行的情况下构建索引，应删除该索引并重新执行 CREATE INDEX CONCURRENTLY 命令。

参数

INDEX

重新创建指定的索引。

TABLE

重新创建指定表的所有索引。如果该表有一个辅助“TOAST”表，也会对其重新索引。

SCHEMA

重新创建指定模式中的所有索引。如果该模式中的某个表有一个辅助“TOAST”表，也会对其重新索引。共享系统目录上的索引也会被处理。这种形式的REINDEX 不能在事务块内执行。

DATABASE

重新创建当前数据库中的所有索引。共享系统目录上的索引也会被处理。这种形式的 REINDEX 不能在事务块内执行。

SYSTEM

重新创建当前数据库内系统目录上的所有索引。共享系统目录上的索引也包含在内。用户表上的索引不会被处理。这种形式的 REINDEX 不能在事务块内执行。

name

要重新索引的特定索引、表或数据库的名称。索引名和表名可以带模式限定。目前，REINDEX DATABASE 和 REINDEX SYSTEM 只能对当前数据库重新索引，所以其参数必须与当前数据库名匹配。

VERBOSE

在每个索引被重建时打印进度报告。

注解

如果怀疑某个用户表上的索引已经损坏，可以使用 REINDEX INDEX 或 REINDEX TABLE 直接重建该索引，或者重建该表上的所有索引。

如果需要从系统表上的索引损坏中恢复，情况就更复杂了。在这种情况下，重要的是系统本身没有使用任何可疑索引。（事实上，在这种场景下，你可能会发现服务器进程在启动时立即崩溃，因为它依赖损坏的索引。）要安全恢复，必须用-P 选项启动服务器，该选项会阻止服务器在查找系统目录时使用索引。

一种做法是关闭服务器，并在命令行中包含-P 选项来启动单用户 PostgreSQL 服务器。然后可以根据希望重建的范围，执行REINDEX DATABASE、REINDEX SYSTEM、REINDEX TABLE 或 REINDEX INDEX。如果拿不准，就使用 REINDEX SYSTEM 来重建该数据库中的所有系统索引。然后退出单用户服务器会话并重新启动常规服务器。关于如何与单用户服务器接口交互的更多信息，参见postgres 参考页。

另一种方法是启动一个常规服务器会话，并在其命令行选项中包含 -P。具体做法因客户端而异，但对于所有基于 libpq 的客户端，都可以在启动客户端之前将环境变量PGOPTIONS 设置为-P。注意，尽管这种方法不需要阻止其他客户端，但在修复完成之前，阻止其他用户连接到受损数据库可能仍然更稳妥。

REINDEX 类似于删除并重新创建索引，因为索引内容都是从头重建的。不过，两者在锁方面的考量相当不同。REINDEX 会阻止该索引所属表上的写入，但不阻止读取。它还会对正在处理的特定索引获取ACCESS EXCLUSIVE 锁，从而阻塞试图使用该索引的读取。相比之下，DROP INDEX 会短暂地对父表获取 ACCESS EXCLUSIVE 锁，同时阻塞写入和读取。随后的 CREATE INDEX 会阻止写入但不阻止读取；由于索引不存在，读取不会尝试使用它，因此不会发生阻塞，但读取可能被迫使用代价高昂的顺序扫描。

对单个索引或表重新索引，要求用户是该索引或表的拥有者。对模式或数据库重新索引，则要求用户是该模式或数据库的拥有者。特别要注意，这意味着非超级用户也可能重建其他用户拥有的表的索引。不过，作为一个特殊例外，当非超级用户执行REINDEX DATABASE、REINDEX SCHEMA 或 REINDEX SYSTEM 时，除非该用户拥有共享系统目录（而这通常并非如此），否则会跳过共享系统目录上的索引。当然，超级用户始终可以重新索引任何对象。

不支持对分区表或分区索引重新索引。可以改为分别对每个分区重新索引。

示例

重建单个索引：

```
REINDEX INDEX my_index;
```

重建表 my_table 上的所有索引：

```
REINDEX TABLE my_table;
```

在不假定系统索引已经有效的情况下，重建某个数据库中的所有索引：

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

兼容性

在 SQL 标准中没有 REINDEX 命令。

RELEASE SAVEPOINT

RELEASE SAVEPOINT — 销毁一个先前定义的保存点

大纲

```
RELEASE [ SAVEPOINT ] savepoint_name
```

描述

`RELEASE SAVEPOINT` 销毁先前在当前事务中定义的保存点。

销毁保存点后，就不能再将它用作回滚点，但除此之外没有其他用户可见的行为。它不会撤销保存点建立之后执行的命令的效果。（要这样做，请参见 ROLLBACK TO SAVEPOINT。）在不再需要保存点时将其销毁，可以让系统在事务结束之前回收一些资源。

`RELEASE SAVEPOINT` 还会销毁在指定保存点建立之后建立的所有保存点。

参数

savepoint_name

要销毁的保存点名称。

注解

指定一个此前未定义的保存点名称是一种错误。

当事务处于已中止状态时，不能释放保存点。

如果有多个保存点使用相同名称，则只会释放最近定义且尚未释放的那个。重复执行该命令会逐步释放更早的同名保存点。

示例

要建立保存点，并在稍后销毁它：

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上述事务将同时插入 3 和 4。

兼容性

该命令符合 SQL 标准。标准规定关键字 `SAVEPOINT` 是强制的，但PostgreSQL 允许省略它。

另见

BEGIN, COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT

RESET

RESET — 将一个运行时参数的值恢复为默认值

大纲

```
RESET configuration_parameter
RESET ALL
```

描述

`RESET` 将运行时参数恢复为其默认值。`RESET` 是

```
SET configuration_parameter TO DEFAULT
```

的另一种拼写。详见 SET。

默认值指的是：在当前会话中，如果从未对该参数执行过`SET`，它本应具有的值。该值的实际来源可能是编译时内置的默认值、配置文件、命令行选项，或者针对特定数据库或特定用户的默认设置。这与将其定义为“该参数在会话开始时的值”略有不同，因为如果该值来自配置文件，那么它会被重置为配置文件当前指定的值。详见第 19 章。

`RESET` 的事务行为和 `SET` 相同：它的效果会在事务回滚时被撤销。

参数

configuration_parameter

可设置的运行时参数名称。可用参数见第 19 章以及 SET 参考页。

`ALL`

将所有可设置的运行时参数重置为默认值。

示例

将 `timezone` 配置参数设置为其默认值：

```
RESET timezone;
```

兼容性

`RESET` 是一种 PostgreSQL 扩展。

另见

SET, SHOW

REVOKE

REVOKE — 撤销访问权限

大纲

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } function_name [ ( [ [ argmode ] [ arg_name ] arg_
type [, ...] ] ) ] [, ...]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ ADMIN OPTION FOR ]
    role_name [, ...] FROM role_specification [, ...]
    [ GRANTED BY role_specification ]
    [ CASCADE | RESTRICT ]
```

其中 *role_specification* 可以是:

```
    [ GROUP ] role_name
  | PUBLIC
  | CURRENT_USER
  | SESSION_USER
```

描述

REVOKE 命令从一个或多个角色那里撤销先前授予的权限。关键字 PUBLIC 指的是由所有角色组成的隐式定义组。

有关各种权限类型的含义，请参见GRANT 命令的描述。

请注意，任何特定角色实际拥有的权限，是直接授予给它的权限、授予给它当前所属任一角色的权限，以及授予给 PUBLIC 的权限之和。因此，例如，从 PUBLIC 撤销 SELECT 权限，并不一定意味着所有角色都失去了该对象上的 SELECT 权限：那些被直接授予该权限或通过其他角色获得该权限的角色仍然拥有它。类似地，从某个用户撤销 SELECT 权限，如果 PUBLIC 或其所属的其他角色仍然拥有 SELECT 权限，该用户仍可能使用 SELECT。

如果指定了 GRANT OPTION FOR，则只撤销该权限的授予选项，而不撤销权限本身。否则，权限和授予选项都会被撤销。

如果某个用户持有带授予选项的权限，并且已经将其授予其他用户，那么其他用户持有的这些权限称为依赖权限。若正在撤销第一个用户持有的该权限或其授予选项，并且存在依赖权限，则在指定 CASCADE 时这些依赖权限也会被一并撤销；否则，该撤销操作会失败。此递归撤销只影响那些通过某条可追溯到本 REVOKE 命令目标用户的用户链授予的权限。因此，如果受影响用户还通过其他用户获得了同一权限，那么他们实际上可能仍保有该权限。

在撤销某个表上的权限时，该表每一列上的对应列权限（若有）也会自动被撤销。反过来，如果某个角色已被授予表级权限，那么从单独列上撤销同一权限不会产生任何效果。

在撤销角色成员资格时，GRANT OPTION 改称为 ADMIN OPTION，但行为类似。这种形式的命令还允许使用 GRANTED BY 选项，但目前会忽略该选项（只检查所指定角色是否存在）。另请注意，这种形式的命令不允许把噪声词 GROUP 写在 *role_specification* 中。

注解

使用 psql 的 \dp 命令可以显示现有表和列上授予的权限。有关格式的信息，参见 GRANT。对于非表对象，也有其他 \d 命令可以显示它们的权限。

用户只能撤销由自己直接授予的权限。例如，如果用户 A 已将某项带授予选项的权限授予用户 B，而用户 B 又将其授予用户 C，则用户 A 不能直接从 C 撤销该权限。相反，用户 A 可以从 B 撤销授予选项，并使用 CASCADE 选项，这样该权限就会进一步从 C 撤销。再例如，如果 A 和 B 都将同一权限授予了 C，A 可以撤销自己授予的那份，但不能撤销 B 授予的那份，因此 C 实际上仍会拥有该权限。

当对象的非拥有者尝试对该对象执行 REVOKE 时，如果该用户在该对象上完全没有任何权限，命令将立即失败。只要该用户至少拥有某项权限，命令就会继续执行，但只会撤销那些该用户持有授予选项的权限。如果未持有任何授予选项，REVOKE ALL PRIVILEGES 形式会发出警告；而其他形式如果命令中特别列出的任一权限未持有其授予选项，也会发出警告。（原则上，这些说明也适用于对象拥有者；但由于拥有者始终被视为持有全部授予选项，这种情况实际上不会发生。）

如果超级用户选择执行 GRANT 或 REVOKE 命令，则该命令会像由受影响对象的拥有者发出那样执行。由于所有权限最终都来自对象拥有者（可能经由授予选项链间接传递），超级用户可以撤销所有权限，但如上所述，这可能需要使用 CASCADE。

REVOKE 也可以由并非受影响对象拥有者的角色执行，只要该角色是拥有该对象之角色的成员，或者是持有该对象上 WITH GRANT OPTION 权限之角色的成员。在这种情况下，该命令会视同由实际拥有该对象的角色，或持有该对象上 WITH GRANT OPTION 权限的角色发出。例如，如果表 t1 由角色 g1 拥有，而角色 u1 是 g1 的成员，那么 u1 可以撤销 t1 上那些记录为由 g1 授出的权限。这包括由 u1 以及角色 g1 的其他成员作出的授权。

如果执行 REVOKE 的角色通过多条角色成员资格路径间接持有权限，则使用哪个上层角色来执行该命令并无规定。在这种情况下，最佳做法是使用 SET ROLE 切换成你希望作为其身份执行 REVOKE 的那个具体角色。否则，可能会撤销掉并非你本意要撤销的权限，或者根本没有撤销任何权限。

示例

撤销表 films 上授予所有用户的插入权限：

```
REVOKE INSERT ON films FROM PUBLIC;
```

撤销用户 manuel 在视图 kinds 上的所有权限：

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

请注意，这实际上意味着“撤销所有由我授予的权限”。

撤销用户 joe 在角色 admins 中的成员资格：

```
REVOKE admins FROM joe;
```

兼容性

GRANT 命令的兼容性注解同样适用于 REVOKE。按照标准，关键词 RESTRICT 或 CASCADE 是必需的，但 PostgreSQL 默认假定为 RESTRICT。

另见

GRANT

ROLLBACK

ROLLBACK — 中止当前事务

大纲

```
ROLLBACK [ WORK | TRANSACTION ]
```

描述

`ROLLBACK` 回滚当前事务，并丢弃该事务所做的全部更新。

参数

`WORK`
`TRANSACTION`

可选关键字，没有任何作用。

注解

使用 COMMIT 成功结束事务。

在事务块外执行 `ROLLBACK` 会发出警告，除此之外没有任何效果。

示例

要中止所有更改：

```
ROLLBACK;
```

兼容性

SQL 标准只规定了 `ROLLBACK` 和 `ROLLBACK WORK` 两种形式。除此之外，该命令完全符合标准。

另见

BEGIN, COMMIT, ROLLBACK TO SAVEPOINT

ROLLBACK PREPARED

ROLLBACK PREPARED — 回滚一个先前为两阶段提交而预备的事务

大纲

```
ROLLBACK PREPARED transaction_id
```

描述

`ROLLBACK PREPARED` 回滚一个处于预备状态的事务。

参数

`transaction_id`

要回滚的事务的事务标识符。

注解

要回滚预备事务，执行者必须是最初执行该事务的同一用户，或者是超级用户；但不必处在执行该事务的同一会话中。

这个命令不能在事务块内执行。该预备事务会被立即回滚。

当前所有处于预备状态的事务都列在 `pg_prepared_xacts` 系统视图中。

示例

回滚事务标识符为 `foobar` 的事务：

```
ROLLBACK PREPARED 'foobar';
```

兼容性

`ROLLBACK PREPARED` 是 PostgreSQL 扩展。它旨在供外部事务管理系统使用，其中有些系统已被标准覆盖（例如 X/Open XA），但这些系统的 SQL 侧并未标准化。

另见

PREPARE TRANSACTION, COMMIT PREPARED

ROLLBACK TO SAVEPOINT

ROLLBACK TO SAVEPOINT — 回滚到一个保存点

大纲

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

描述

回滚该保存点建立后执行的所有命令。该保存点仍然有效，如有需要，之后还可以再次回滚到它。

`ROLLBACK TO SAVEPOINT` 会隐式销毁在指定保存点之后建立的所有保存点。

参数

`savepoint_name`

要回滚到的保存点名称。

注解

使用 RELEASE SAVEPOINT 可以销毁一个保存点，而不丢弃在它建立之后执行的命令所产生的效果。

指定一个尚未建立的保存点名称是一种错误。

就保存点而言，游标带有一些非事务性的行为。凡是在某个保存点内打开的游标，在回滚该保存点时都会被关闭。如果先前打开的游标在某个随后又被回滚的保存点内受到了 `FETCH` 或 `MOVE` 命令的影响，那么该游标会保留在`FETCH` 使其指向的位置上（也就是说，由`FETCH` 引起的游标移动不会被回滚）。关闭游标同样不会因回滚而撤销。不过，如果游标查询导致了其他副作用（例如该查询调用的易变函数带来的副作用），且这些副作用发生在后来被回滚的保存点期间，那么它们会被回滚。如果某个游标的执行导致事务中止，该游标会进入不可执行状态，因此即使事务可以通过`ROLLBACK TO SAVEPOINT` 恢复，该游标也不能再使用。

示例

要撤销在 `my_savepoint` 建立后执行的命令的效果：

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

游标位置不受保存点回滚的影响：

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

兼容性

SQL 标准规定关键字 `SAVEPOINT` 是强制的，但 PostgreSQL 和 Oracle 允许省略它。SQL 只允许在 `ROLLBACK` 之后使用 `WORK` 作为噪声词，不允许使用`TRANSACTION`。此外，SQL 还有一个可选子句 `AND [ NO ] CHAIN`，而PostgreSQL 当前尚不支持。除此之外，该命令符合 SQL 标准。

另见

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT

SAVEPOINT

SAVEPOINT — 在当前事务中定义一个新的保存点

大纲

```
SAVEPOINT savepoint_name
```

描述

`SAVEPOINT` 在当前事务中建立一个新的保存点。

保存点是事务中的一种特殊标记，它允许回滚在其建立之后执行的所有命令，从而将事务状态恢复到建立保存点时的状态。

参数

savepoint_name

赋予新保存点的名称。如果已经存在同名保存点，则在较新的同名保存点被释放之前，较旧的保存点将无法访问。

注解

使用 ROLLBACK TO SAVEPOINT 回滚到保存点。使用 RELEASE SAVEPOINT 销毁保存点，同时保留其建立后执行的命令所产生的效果。

保存点只能在事务块内部建立。一个事务中可以定义多个保存点。

示例

要建立一个保存点，并在稍后撤销其建立后执行的所有命令的效果：

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

上述事务将插入值 1 和 3，而不会插入 2。

要建立保存点，并在稍后销毁它：

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上述事务将同时插入 3 和 4。

要重复使用同一个保存点名称：

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);

    -- 回滚到第二个保存点
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;               -- 显示值为 1 和 2 的行

    -- 释放第二个保存点
    RELEASE SAVEPOINT my_savepoint;

    -- 回滚到第一个保存点
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;               -- 仅显示值为 1 的行
COMMIT;
```

上述事务展示了先回滚值为 3 的行，再回滚值为 2 的行。

兼容性

SQL 要求在建立另一个同名保存点时自动销毁原有保存点。在 PostgreSQL 中，旧保存点会被保留，不过在回滚或释放时只会使用最近建立的那个。（如果使用 `RELEASE SAVEPOINT` 释放较新的保存点，较旧的保存点将再次可供 `ROLLBACK TO SAVEPOINT` 和 `RELEASE SAVEPOINT` 使用。）除此之外，`SAVEPOINT` 完全符合 SQL。

另见

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, ROLLBACK TO SAVEPOINT

SECURITY LABEL

SECURITY LABEL — 定义或更改应用于对象的安全标签

大纲

```
SECURITY LABEL [ FOR provider ] ON
{
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  DATABASE object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN TABLE object_name
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  PUBLICATION object_name |
  ROLE object_name |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  SCHEMA object_name |
  SEQUENCE object_name |
  SUBSCRIPTION object_name |
  TABLESPACE object_name |
  TYPE object_name |
  VIEW object_name
} IS { string_literal | NULL }
```

其中 *aggregate_signature* 为:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ...
 ]
```

描述

SECURITY LABEL 为数据库对象设置安全标签。一个给定的数据库对象可以关联任意数量的安全标签，每个标签提供者对应一个。标签提供者是使用函数register_label_provider 注册自身的可加载模块。

> **注意**
>
> register_label_provider 不是一个 SQL 函数；它只能从加载到后端的 C 代码中调用。

标签提供者决定给定标签是否有效，以及是否允许将该标签赋给给定对象。给定标签的含义同样由标签提供者自行决定。PostgreSQL 不限制标签提供者是否解释安全标签，也不限制其如何解释；它仅提供一种存储安全标签的机制。实际上，此功能旨在支持与基于标签的强制访问控制（MAC）系统（例如SELinux）集成。这类系统基于对象标签，而不是基于用户和组等传统的自主访问控制（DAC）概念，做出所有访问控制决策。

参数

object_name
table_name.column_name
aggregate_name
function_name
procedure_name
routine_name

要加上安全标签的对象名称。表、聚合、域、外部表、函数、过程、例程、序列、类型和视图的名称可以带模式限定。

provider

要与该标签关联的标签提供者名称。指定的提供者必须已加载，并且必须同意所提议的标签设置操作。若只加载了一个提供者，则为简洁起见可以省略其名称。

argmode

函数、过程或聚合函数参数的模式：IN、OUT、INOUT 或 VARIADIC。如果省略，默认值是 IN。注意 SECURITY LABEL 实际上并不关心 OUT 参数，因为确定函数身份只需要输入参数。因此，列出 IN、INOUT 和 VARIADIC 参数就足够了。

argname

函数、过程或聚合函数参数的名称。注意SECURITY LABEL 实际上并不关心参数名称，因为确定函数身份只需要参数数据类型。

argtype

函数、过程或聚合函数参数的数据类型。

large_object_oid

大对象的 OID。

PROCEDURAL

这是一个噪声词。

string_literal

安全标签的新值，以字符串字面量形式写出。

NULL

写入 NULL 可删除安全标签。

示例

下面的示例展示了如何设置或更改一个表的安全标签：

```
SECURITY LABEL FOR selinux ON TABLE mytable IS 'system_u:object_r:sepgsql_table_t:s0';
```

要移除该标签：

```
SECURITY LABEL FOR selinux ON TABLE mytable IS NULL;
```

兼容性

在 SQL 标准中没有 `SECURITY LABEL` 命令。

另见

sepgsql, `src/test/modules/dummy_seclabel`

SELECT

SELECT, TABLE, WITH — 从表或视图中检索行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ] [ NOWAIT |
 SKIP LOCKED ] [...] ]
```

其中 *from_item* 为以下之一:

```
    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
                [ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ] ]
    [ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias ( column_definition [, ...]
 )
    [ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    [ LATERAL ] ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS ( column_definition [, .
..] ) ] [, ...] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item join_type from_item { ON join_condition | USING ( join_column [, ...] ) }
    from_item NATURAL join_type from_item
    from_item CROSS JOIN from_item
```

而 *grouping_element* 为以下之一:

```
    ( )
    expression
    ( expression [, ...] )
    ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
    CUBE ( { expression | ( expression [, ...] ) } [, ...] )
    GROUPING SETS ( grouping_element [, ...] )
```

而 *with_query* 为:

```
    with_query_name [ ( column_name [, ...] ) ] AS ( select | values | insert | update | delete
 )

TABLE [ ONLY ] table_name [ * ]
```

描述

SELECT 从零个或多个表中检索行。SELECT 的一般处理流程如下:

1. WITH 列表中的所有查询都会被计算。这些实际上充当临时表，可以在FROM 列表中引用。在 FROM 列表中多次引用的 WITH 查询只会计算一次。（参见下面的 WITH 子句。）

2. 所有 FROM 列表中的元素都会被计算。（FROM 列表中的每个元素都是一个真实或虚拟表。）如果在 FROM 列表中指定了多个元素，则它们会被交叉连接在一起。（参见下面的 FROM 子句。）

3. 如果指定了 WHERE 子句，则不满足条件的所有行将从输出中删除。（请参见下面的 WHERE 子句。）

4. 如果指定了 GROUP BY 子句，或者存在聚合函数调用，输出将被组合成在一个或多个值上匹配的行组，并计算聚合函数的结果。如果存在HAVING 子句，它将消除不满足给定条件的组。（参见GROUP BY 子句和 HAVING 子句。）虽然查询输出列名义上是在下一步计算的，但它们也可以在GROUP BY 子句中被引用（按名称或序号）。

5. 实际输出行是使用每个选定行或行组的 SELECT 输出表达式计算的。（参见下面的 SELECT 列表。）

6. SELECT DISTINCT 消除结果中的重复行。SELECT DISTINCT ON 会消除在所有指定表达式上匹配的行，只保留每组中的第一行。SELECT ALL（默认）将返回所有候选行，包括重复行。（参见下面的 DISTINCT 子句。）

7. 使用操作符 UNION、INTERSECT 和 EXCEPT，可以将多个SELECT 语句的输出合并成一个结果集。UNION 操作符返回在一个或两个结果集中的所有行。INTERSECT 操作符返回同时出现在两个结果集中的所有行。EXCEPT 操作符返回在第一个结果集中但不在第二个结果集中的行。在这三种情况下，除非指定ALL，否则将消除重复行。还可以添加噪声词 DISTINCT，以明确指定去重。请注意，这里的默认行为是DISTINCT，即使SELECT 本身的默认行为是 ALL。（请参见下面的 UNION 子句、INTERSECT 子句和 EXCEPT 子句。）

8. 如果指定了 ORDER BY 子句，则返回的行按指定顺序排序。如果没有给出ORDER BY，则按系统认为最快的顺序返回行。（参见下面的 ORDER BY 子句。）

9. 如果指定了 LIMIT（或 FETCH FIRST）或 OFFSET 子句，SELECT 语句只返回结果行的子集。（参见下面的 LIMIT 子句。）

10. 如果指定了 FOR UPDATE、FOR NO KEY UPDATE、FOR SHARE 或 FOR KEY SHARE，SELECT 语句将选定的行锁定，防止并发更新。（参见下面的锁定子句。）

你必须拥有 SELECT 命令中使用到的每一列上的 SELECT 权限。FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 或者 FOR KEY SHARE 还要求具备 UPDATE 权限（对这样选中的每个表至少一列）。

参数

WITH 子句

WITH 子句允许你指定一个或多个可在主查询中按名称引用的子查询。这些子查询在主查询执行期间实际上充当临时表或视图。每个子查询都可以是 SELECT、TABLE、VALUES、INSERT、UPDATE、DELETE 语句。在 WITH 中编写数据修改语句（INSERT、UPDATE、DELETE）时，通常要包括一个 RETURNING 子句。被主查询读取并构成临时表的是 RETURNING 的输出，而不是该语句所修改的底层表。如果省略 RETURNING，该语句仍会执行，但不会产生输出，因此主查询无法把它当作表来引用。

对于每个 WITH 查询，都必须指定一个名称（不带模式限定）。还可以指定一个列名列表；如果省略，则列名将从子查询中推导出来。

如果指定了 RECURSIVE，则允许一个 SELECT 子查询使用名称引用自身。这样一个子查询的形式必须是

non_recursive_term UNION [ALL | DISTINCT] *recursive_term*

其中递归自引用必须出现在 UNION 的右手边。每个查询中只允许一个递归自引用。不支持递归数据修改语句，但是可以在一个数据修改语句中使用一个递归 SELECT 查询的结果。示例可见第 7.8 节。

RECURSIVE 的另一个效果是 WITH 查询不需要被排序：一个查询可以引用另一个在列表中比它靠后的查询（不过，循环引用或者互递归没有实现）。如果没有RECURSIVE，WITH 查询只能引用在 WITH 列表中位置更前面的兄弟 WITH 查询。

WITH 查询的一个关键特性是，每次执行主查询时，它们都只会求值一次，即使主查询多次引用它们。特别是，无论主查询是否读取了它们的全部输出或任何输出，数据修改语句都保证执行一次且仅执行一次。

当 WITH 子句中有多个查询时，RECURSIVE 应只编写一次，紧跟在WITH 之后。它适用于 WITH 子句中的所有查询，尽管它对不使用递归或前向引用的查询没有影响。

主查询和 WITH 查询（概念上）都在同一时间执行。这意味着，除了读取其RETURNING 输出之外，查询的其他部分都看不到WITH 中数据修改语句的效果。如果两个这样的数据修改语句试图修改同一行，结果并无规定。

更多信息请见第 7.8 节。

FROM 子句

FROM 子句为 SELECT 指定一个或者更多源表。如果指定了多个源表，结果将是所有源表的笛卡尔积（交叉连接）。但是通常会增加限定条件（通过 WHERE）来把返回的行限制为该笛卡尔积的一个小子集。

FROM 子句可以包含以下元素：

table_name

要扫描的现有表或视图的名称（可选模式限定符）。如果在表名之前指定 ONLY，则仅扫描该表。如果未指定ONLY，则扫描该表及其所有后代表（如果有）。可选地，可以在表名后指定 *，以明确指示包括后代表。

alias

包含该别名的 FROM 项的替代名称。别名可用于简写，或者消除自连接（同一张表被扫描多次）中的歧义。提供别名后，它会完全隐藏表或函数的实际名称；例如给定 FROM foo AS f，SELECT 的其余部分必须把这个 FROM 项写成 f 而不是 foo。如果写了别名，还可以写列别名列表，为该表的一个或多个列提供替代名称。

TABLESAMPLE *sampling_method* (*argument* [, ...]) [REPEATABLE (*seed*)]

跟在 *table_name* 之后的 TABLESAMPLE 子句表示，应使用指定的 *sampling_method* 从该表中取回行的一个子集。这种抽样先于任何其他过滤条件（例如 WHERE 子句）的应用。标准 PostgreSQL 发行版包含两种抽样方法，即BERNOULLI 和 SYSTEM；其他抽样方法可以通过扩展安装到数据库中。

BERNOULLI 和 SYSTEM 抽样方法各接受一个 *argument*，表示要抽样的表的比例，以 0 到 100 之间的百分比表示。该参数可以是任何返回real 的表达式。（其他抽样方法可能接受更多或不同的参数。）这两种方法都会返回表的一个随机样本，其中大约包含表中指定百分比的行。BERNOULLI 方法扫描整个表，并以指定概率独立选择或忽略单个行。SYSTEM 方法进行块级抽样，每个块有指定的选择机会；返回每个选定块中的所有行。当指定小的抽样百分比时，SYSTEM 方法比 BERNOULLI 方法快得多，但由于聚类效应，它返回的表样本随机性可能稍差一些。

可选的 REPEATABLE 子句指定一个 *seed* 数字或表达式，用于在抽样方法内部生成随机数。种子值可以是任意非空浮点值。如果两个查询指定了相同的种子和*argument* 值，且该表在此期间未被修改，它们会选出相同的表样本；但不同的种子值通常会产生不同的样本。如果未给出REPEATABLE，则每次查询都会基于系统生成的种子选取新的随机样本。注意，某些附加抽样方法并不接受REPEATABLE，因此每次使用时都会生成新的样本。

select

子 SELECT 可以出现在 FROM 子句中，它的作用就像在这个SELECT 命令的执行期间创建了一个临时表。注意，子SELECT 必须用圆括号括起来，并且必须为其提供别名。这里也可以使用 VALUES 命令。

with_query_name

WITH 查询通过写出它的名称来引用，就像该查询名是表名一样。（事实上，对于主查询而言，WITH 查询会遮蔽任何同名的真实表；如有必要，可以通过模式限定表名来引用该同名真实表。）也可以像对待表一样为它提供别名。

function_name

函数调用可以出现在 FROM 子句中。（这对于返回结果集的函数尤其有用，但任何函数都可以使用。）其效果就像在这条SELECT 命令执行期间，将函数的输出创建成了一张临时表。在函数调用后加上可选的WITH ORDINALITY 子句时，会在函数的所有输出列之后追加一列，为每一行编号。

可以像对表一样提供别名。如果写了别名，还可以写一个列别名列表，为函数复合返回类型中的一个或多个属性提供替代名称，其中也包括ORDINALITY 所添加的列（如果有）。

多个函数调用可以通过用 ROWS FROM(...) 括起来，合并成单个FROM 子句项。这样一个项的输出会先拼接每个函数的第一行，再拼接每个函数的第二行，依此类推。如果某些函数产生的行数少于其他函数，则会用空值替代缺失的数据，以确保返回的总行数始终与产生最多行的那个函数相同。

如果函数被定义为返回 record 数据类型，则必须给出别名或关键字AS，后面跟一个如下形式的列定义列表：(*column_name data_type* [, ...])。列定义列表必须与该函数实际返回的列数和列类型相匹配。

当使用 ROWS FROM(...) 语法时，如果其中某个函数需要列定义列表，最好将该列定义列表放在ROWS FROM(...) 内部、紧跟在函数调用之后。只有在只有一个函数且没有 WITH ORDINALITY 子句时，才能把列定义列表放在 ROWS FROM(...) 结构之后。

要将 ORDINALITY 与列定义列表一起使用，必须使用ROWS FROM(...) 语法，并将列定义列表放在ROWS FROM(...) 内部。

join_type

以下之一：

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN

对于 INNER 和 OUTER 连接类型，必须指定连接条件，即下列三者之一：ON *join_condition*、USING (*join_column* [, ...])，或者NATURAL。请参见下文的含义。

一个 JOIN 子句会把两个 FROM 项组合起来，为方便起见，我们把它们称为“表”，尽管实际上它们可以是任何类型的FROM 项。如有必要，可以使用括号来确定嵌套顺序。在没有括号时，JOIN 按从左到右的顺序嵌套。无论如何，JOIN 的结合都比分隔 FROM 列表项的逗号更紧密。所有 JOIN 选项都只是记法上的便利，因为它们并没有提供任何无法用普通FROM 和 WHERE 做到的能力。

LEFT OUTER JOIN 返回已限定笛卡尔积中的所有行（即，通过其连接条件的所有组合行），外加左侧表中每一行的一个副本；对于这些行，不存在通过连接条件的右侧行。这样的左侧行会通过在右侧列中插入空值而扩展到连接表的完整宽度。注意，在决定哪些行有匹配项时，只考虑JOIN 子句自身的条件；外层条件是在之后应用的。

相反，RIGHT OUTER JOIN 返回所有连接后的行，再加上每个未匹配右侧行对应的一行（左侧用空值扩展）。这只是一种记法上的便利，因为你可以通过交换左右表将其改写成 LEFT OUTER JOIN。

FULL OUTER JOIN 返回所有连接后的行，再加上每个未匹配的左侧行（右侧用空值扩展），以及每个未匹配的右侧行（左侧用空值扩展）。

ON *join_condition*

join_condition 是一个表达式，其结果为boolean 类型的值（类似于 WHERE 子句），指定哪些连接中的行被认为是匹配的。

USING (*join_column* [, ...])

一个形如 USING (a, b, ...) 的子句是 ON left_table.a = right_table.a AND left_table.b = right_table.b ...的简写。此外，USING 意味着只有每对等价列中的一个会包含在连接输出中，而不是两者都包含。

NATURAL

NATURAL 是一个简写，表示一个包含两个表中所有具有相同名称的列的USING 列表。如果没有共同的列名，NATURAL 等同于 ON TRUE。

CROSS JOIN

CROSS JOIN 等同于 INNER JOIN ON (TRUE)，也就是说，没有任何行被条件限制删除。它们产生一个简单的笛卡尔积，与在FROM 的顶层列出两个表时得到的结果相同，但受连接条件（如果有）的限制。

LATERAL

LATERAL 关键字可以在子 SELECT FROM 项之前出现。这允许子 SELECT 引用在 FROM 列表中出现在其前面的 FROM 项的列。（没有 LATERAL，每个子SELECT 都是独立求值的，因此不能交叉引用任何其他FROM 项。）

LATERAL 也可以放在函数调用形式的 FROM 项之前，但在这种情况下它只是一个噪声词，因为函数表达式无论如何都可以引用更早出现的FROM 项。

LATERAL 项既可以出现在 FROM 列表的顶层，也可以出现在JOIN 树中。在后一种情况下，如果它位于某个 JOIN 的右侧，也可以引用该连接左侧的任何项。

当 FROM 项包含 LATERAL 交叉引用时，求值过程如下：对于提供被交叉引用列的FROM 项的每一行，或者对于提供这些列的多个FROM 项的一组行，都会使用该行或行集中的列值来计算 LATERAL 项。得到的行随后照常与生成它们的那些行连接。这个过程会针对列源表中的每一行或每一组行重复执行。

列源表必须通过 INNER 或 LEFT 连接到 LATERAL 项，否则就无法得到一个定义良好的行集合，用来计算该LATERAL 项的每一组结果行。因此，尽管像 *X* RIGHT JOIN LATERAL *Y* 这样的结构在语法上有效，但实际上并不允许*Y* 引用 *X*。

WHERE 子句

可选的 WHERE 子句的形式

```
WHERE condition
```

其中 *condition* 是任一计算得到 boolean 类型结果的表达式。任何不满足这个条件的行都会从输出中被消除。如果用一行的实际值替换其中的变量引用后，该表达式返回真，则该行符合条件。

GROUP BY 子句

可选的 GROUP BY 子句的一般形式为

```
GROUP BY grouping_element [, ...]
```

GROUP BY 会把所有在分组表达式上具有相同值的已选中行压缩成单独一行。用于 *grouping_element* 中的 *expression* 可以是输入列名、输出列（SELECT 列表项）的名称或序号或者由输入列值构成的任意表达式。在出现歧义时，GROUP BY 名称将被解释为输入列名而不是输出列名。

如果 GROUPING SETS、ROLLUP 或 CUBE 中的任何一个作为分组元素出现，那么整个 GROUP BY 子句就定义了若干个相互独立的分组集。其效果等价于在多个子查询之间构造一个 UNION ALL，各个子查询分别以各自的分组集作为它们的GROUP BY 子句。关于分组集处理的更多细节，见第 7.2.4 节。

聚合函数（如果使用）会在组成每一个分组的所有行上进行计算，从而为每一个分组产生一个单独的值（如果有聚合函数但是没有 GROUP BY 子句，则查询会被当成是由所有选中行构成的一个单一分组）。传递给每一个聚合函数的行集合可以通过在聚合函数调用上附加一个FILTER 子句来进一步过滤，详见第 4.2.7 节。当存在一个 FILTER 子句时，只有那些匹配它的行才会被包括在该聚合函数的输入中。

当存在 GROUP BY 子句或者任何聚合函数时，SELECT 列表表达式不能引用非分组列，除非它出现在聚合函数中，或者该非分组列函数依赖于分组列，否则，一个非分组列可能有多个可返回的值。如果分组列（或其子集）构成包含该非分组列的表的主键，则存在函数依赖。

记住所有的聚合函数都是在 HAVING 子句或者 SELECT 列表中的任何“标量”表达式之前被计算。这意味着 CASE 表达式不能被用来跳过聚合函数的计算，见第 4.2.14 节。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 和 FOR KEY SHARE 不能和 GROUP BY 一起指定。

HAVING 子句

可选的 HAVING 子句的形式

```
HAVING condition
```

其中 *condition* 与 WHERE 子句中指定的条件相同。

HAVING 消除不满足该条件的分组行。HAVING 与 WHERE 不同：WHERE 会在应用 GROUP BY 之前过滤个体行，而HAVING 过滤由 GROUP BY 创建的分组行。*condition* 中引用的每一列都必须无歧义地引用某个分组列，除非该引用出现在聚合函数中，或者该非分组列函数依赖于分组列。

即使没有 GROUP BY 子句，HAVING 的存在也会把一个查询转变成一个分组查询。这和查询中包含聚合函数但没有 GROUP BY 子句时的情况相同。所有被选择的行都被认为是一个单一分组，并且SELECT 列表和 HAVING 子句只能从聚合函数内部引用表列。如果该 HAVING 条件为真，这样的查询将输出单独一行；否则不返回行。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 和 FOR KEY SHARE 不能与 HAVING 一起指定。

WINDOW 子句

可选的 WINDOW 子句的一般形式为：

```
WINDOW window_name AS ( window_definition ) [, ...]
```

其中，*window_name* 是一个名称，可从 OVER 子句或后续窗口定义中引用，而 *window_definition* 的定义为：

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

如果指定了一个 *existing_window_name*，它必须引用WINDOW 列表中一个更早出现的项。新窗口将从该项中复制它的分区子句，以及排序子句（如果有）。在这种情况下，新窗口不能指定它自己的PARTITION BY 子句，并且只有在被复制的窗口没有ORDER BY 子句时，才可以指定 ORDER BY。新窗口总是使用自己的帧子句，而被复制的窗口不得指定帧子句。

PARTITION BY 列表元素的解释以 GROUP BY 子句元素的方式进行，不过它们总是简单表达式并且绝不能是输出列的名称或编号。另一个区别是这些表达式可以包含聚合函数调用，而这在常规GROUP BY 子句中是不被允许的。它们被允许的原因是窗口是出现在分组和聚合之后的。

类似地，ORDER BY 列表元素的解释也以语句级 ORDER BY 子句元素的方式进行，不过该表达式总是被当做简单表达式并且绝不会是输出列的名称或编号。

可选的 *frame_clause* 为依赖于帧的窗口函数定义窗口帧（并非所有窗口函数都依赖于帧）。对于查询中的每一行（称为当前行），窗口帧都是一组相关行。*frame_clause* 可以是

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

之一，其中*frame_start* 和 *frame_end* 可以是

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

之一，并且*frame_exclusion* 可以是

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

之一。如果省略 *frame_end*，则默认值为 CURRENT ROW。限制条件是：*frame_start* 不能是 UNBOUNDED FOLLOWING，*frame_end* 不能是 UNBOUNDED PRECEDING，并且在上述 *frame_start* 和 *frame_end* 选项列表中，*frame_end* 的选择不能早于 *frame_start* 的选择；例如 RANGE BETWEEN CURRENT ROW AND *offset* PRECEDING 是不被允许的。

默认的帧选项是 RANGE UNBOUNDED PRECEDING，它等同于 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW；它将帧设置为从分区起始处一直到当前行最后一个同等行的所有行（所谓同等行，是指在该窗口的ORDER BY 子句看来与当前行等价的行；如果没有 ORDER BY，则所有行都互为同等行）。一般来说，UNBOUNDED PRECEDING 表示帧从分区的第一行开始，类似地，UNBOUNDED FOLLOWING 表示帧到分区的最后一行结束，与使用 RANGE、ROWS 还是 GROUPS 模式无关。在 ROWS 模式中，CURRENT ROW 表示帧从当前行开始或在当前行结束；而在 RANGE 或 GROUPS 模式中，它表示帧从当前行在 ORDER BY 排序中的第一个同等行开始，或在最后一个同等行结束。*offset* PRECEDING 和 *offset* FOLLOWING 选项的含义会随帧模式而变化。在 ROWS 模式中，*offset* 是一个整数，表示帧从当前行之前或之后多少行开始或结束。在GROUPS 模式中，*offset* 是一个整数，表示帧从当前行所在同等行组之前或之后多少个同等行组开始或结束，其中同等行组是按窗口 ORDER BY 子句判断等价的一组行。在 RANGE 模式中，使用*offset* 选项要求窗口定义中恰好有一个 ORDER BY 列。此时，帧包含的行满足：其排序列值与当前行的排序列值相比，向小的方向（对于 PRECEDING）或向大的方向（对于 FOLLOWING）的差距不超过 *offset*。在这些情况下，*offset* 表达式的数据类型取决于排序列的数据类型。对于数值排序列，它通常与排序列类型相同；对于日期/时间排序列，则为 interval。在上述所有情况下，*offset* 的值都必须非空且非负。此外，虽然 *offset* 不必是简单常量，但它不能包含变量、聚合函数或窗口函数。

frame_exclusion 选项允许当前行周围的行被排除在帧之外，即便根据帧的开始和结束选项应该把它们包括在帧中。EXCLUDE CURRENT ROW 会把当前行排除在帧之外。EXCLUDE GROUP 会把当前行以及它在顺序上的同等行都排除在帧之外。EXCLUDE TIES 把当前行的任何同等行都从帧中排除，但不排除当前行本身。EXCLUDE NO OTHERS 只是明确地指定不排除当前行或其同等行的这种默认行为。

注意，如果ORDER BY 排序无法把行唯一地排序，则ROWS 模式可能产生不可预测的结果。RANGE 以及 GROUPS 模式的目的是确保在 ORDER BY 顺序中同等行被同样对待：一个给定同等行组中的所有行将在一个帧中或者被从帧中排除。

WINDOW 子句的目的是指定出现在查询的 SELECT 列表或 ORDER BY 子句中的窗口函数的行为。这些函数可以在它们的 OVER 子句中用名称引用 WINDOW 子句项。不过，WINDOW 子句项不是必须被引用。如果在查询中没有用到它，它会被简单地忽略。可以使用根本没有任何 WINDOW 子句的窗口函数，因为窗口函数调用可以直接在其OVER 子句中指定它的窗口定义。不过，当多个窗口函数都需要相同的窗口定义时，WINDOW 子句能够减少输入量。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 和 FOR KEY SHARE 不能和 WINDOW 一起被指定。

窗口函数的详细描述在第 3.5 节、第 4.2.8 节以及第 7.2.5 节中。

SELECT 列表

SELECT 列表（位于关键词 SELECT 和 FROM 之间）指定构成 SELECT 语句输出行的表达式。这些表达式可以（并且通常确实会）引用 FROM 子句中计算得到的列。

正如在表中一样，SELECT 的每一个输出列都有一个名称。在一个简单的 SELECT 中，这个名称只是被用来标记要显示的列，但是当SELECT 是一个大型查询的一个子查询时，大型查询会把该名称看做子查询产生的虚表的列名。要指定用于输出列的名称，在该列的表达式后面写上 AS *output_name*（你可以省略 AS，但只能在期望的输出名称不匹配任何 PostgreSQL 关键词（见附录 C）时省略。为了避免和未来增加的关键词冲突，推荐总是写上AS 或者用双引号引用输出名称）。如果你不指定列名，PostgreSQL 会自动选择一个名称。如果列的表达式是一个简单的列引用，那么被选择的名称就和该列的名称相同。在更复杂的情况下，可能使用函数名或类型名，系统也可能退而使用自动生成的名称，例如?column?。

一个输出列的名称可以被用来在 ORDER BY 以及 GROUP BY 子句中引用该列的值，但是不能用于 WHERE 和 HAVING 子句（在其中必须写出表达式）。

可以在输出列表中写* 来取代表达式，它是被选中行的所有列的一种简写方式。还可以写 *table_name*.*，它是只来自那个表的所有列的简写形式。在这些情况中无法用 AS 指定新的名称，输出列的名称将和表列的名称相同。

按照 SQL 标准，输出列表中的表达式应当在应用DISTINCT、ORDER BY 或 LIMIT 之前计算。对于 DISTINCT 来说，这显然是必要的，否则就不清楚究竟要对哪些值去重。不过，在很多情况下，如果先执行ORDER BY 和 LIMIT 再计算输出表达式会更方便，特别是当输出列表中包含可变函数或代价高昂的函数时。那样一来，函数求值的顺序更符合直觉，也不会去计算那些根本不会出现在输出中的行。只要输出表达式没有被 DISTINCT、ORDER BY 或 GROUP BY 引用，PostgreSQL 实际上就会在排序和限制行数之后再计算它们。（反例是 SELECT f(x) FROM tab ORDER BY 1，它显然必须在排序前计算f(x)。）包含集合返回函数的输出表达式则实际上会在排序之后、限制之前计算，这样LIMIT 才能截断该集合返回函数产生的输出。

> **注意**
>
> 9.6 版本之前的 PostgreSQL 不对执行输出表达式、排序、限制行数的时间顺序做任何保证，那将取决于被选中的查询计划的形式。

DISTINCT 子句

如果指定了 SELECT DISTINCT，所有重复的行会被从结果集中移除（为每一组重复的行保留一行）。SELECT ALL 则指定相反的行为：所有行都会被保留，这也是默认情况。

SELECT DISTINCT ON (*expression* [, ...]) 只保留在给定表达式上计算相等的行集合中的第一行。DISTINCT ON 表达式使用和 ORDER BY 相同的规则（见上文）解释。注意，除非用 ORDER BY 来确保所期望的行出现在第一位，每一个集合的“第一行”是不可预测的。例如：

```
SELECT DISTINCT ON (location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC;
```

为每个地点检索最近的天气报告。但是如果我们不使用 ORDER BY 来强制对每个地点的时间值进行降序排序，我们为每个地点得到的报告的时间可能是无法预测的。

DISTINCT ON 表达式必须匹配最左边的 ORDER BY 表达式。ORDER BY 子句通常将包含额外的表达式，这些额外的表达式用于决定在每一个 DISTINCT ON 分组内行的优先级。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 和 FOR KEY SHARE 不能和 DISTINCT 一起使用。

UNION 子句

UNION 子句的一般形式如下：

```
select_statement UNION [ ALL | DISTINCT ] select_statement
```

select_statement 是任何没有 ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 和 FOR KEY SHARE 子句的 SELECT 语句。（如果某个子表达式被圆括号括起来，ORDER BY 和 LIMIT 可以附加在它上面。如果没有圆括号，这些子句会被视为作用于UNION 的结果，而不是作用于其右侧输入表达式。）

UNION 操作符计算相关 SELECT 语句所返回的行的并集。如果一行至少出现在两个结果集中的一个内，它就会在并集中。作为 UNION 两个操作数的 SELECT 语句必须产生相同数量的列并且对应位置上的列必须具有兼容的数据类型。

UNION 的结果不会包含重复行，除非指定了 ALL 选项。ALL 会阻止消除重复（因此，UNION ALL 通常显著地快于 UNION，尽量使用ALL）。也可以写上 DISTINCT，以显式指定默认的去重行为。

除非用圆括号指定计算顺序，同一个SELECT 语句中的多个 UNION 操作符会从左至右计算。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 和 FOR KEY SHARE 不能用于 UNION 结果或者 UNION 的任何输入。

INTERSECT 子句

INTERSECT 子句的一般形式如下：

select_statement INTERSECT [ALL | DISTINCT] *select_statement*

select_statement 是任何没有 ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 以及 FOR KEY SHARE 子句的 SELECT 语句。

INTERSECT 操作符计算相关 SELECT 语句返回的行的交集。如果一行同时出现在两个结果集中，它就在交集中。

INTERSECT 的结果不会包含重复行，除非指定了 ALL 选项。如果有 ALL，一个在左表中有 *m* 次重复并且在右表中有 *n* 次重复的行将会在结果中出现 min(*m*,*n*) 次。也可以写上 DISTINCT，以显式指定默认的去重行为。

除非用圆括号指定计算顺序，同一个SELECT 语句中的多个 INTERSECT 操作符会从左至右计算。INTERSECT 的优先级比 UNION 更高。也就是说，A UNION B INTERSECT C 将被读成 A UNION (B INTERSECT C)。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 和 FOR KEY SHARE 不能用于 INTERSECT 结果或者 INTERSECT 的任何输入。

EXCEPT 子句

EXCEPT 子句的一般形式如下：

select_statement EXCEPT [ALL | DISTINCT] *select_statement*

select_statement 是任何没有 ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 以及 FOR KEY SHARE 子句的 SELECT 语句。

EXCEPT 操作符计算位于左侧 SELECT 语句的结果中但不在右 SELECT 语句结果中的行集合。

EXCEPT 的结果不会包含重复行，除非指定了 ALL 选项。如果有 ALL，一个在左表中有 *m* 次重复并且在右表中有 *n* 次重复的行将会在结果集中出现 max(*m*-*n*,0) 次。也可以写上 DISTINCT，以显式指定默认的去重行为。

除非用圆括号指定计算顺序，同一个SELECT 语句中的多个 EXCEPT 操作符会从左至右计算。EXCEPT 的优先级与 UNION 相同。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE 和 FOR KEY SHARE 不能用于 EXCEPT 结果或者 EXCEPT 的任何输入。

ORDER BY 子句

可选的 ORDER BY 子句的形式如下：

```
ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...]
```

ORDER BY 子句导致结果行被按照指定的表达式排序。如果两行按照最左边的表达式是相等的，则会根据下一个表达式比较它们，依次类推。如果按照所有指定的表达式它们都是相等的，则它们被返回的顺序取决于实现。

每一个 *expression* 可以是输出列（SELECT 列表项）的名称或者序号，它也可以是由输入列值构成的任意表达式。

序号指的是输出列的顺序（从左至右）位置。这种特性可以为不具有唯一名称的列定义一个顺序。这不是绝对必要的，因为总是可以使用 AS 子句为输出列赋予一个名称。

也可以在 ORDER BY 子句中使用任意表达式，包括没有出现在SELECT 输出列表中的列。因此，下面的语句是合法的：

```
SELECT name FROM distributors ORDER BY code;
```

这种特性的一个限制是一个应用在 UNION、INTERSECT 或 EXCEPT 子句结果上的 ORDER BY 只能指定输出列名称或序号，但不能指定表达式。

如果一个 ORDER BY 表达式是一个既匹配输出列名称又匹配输入列名称的简单名称，ORDER BY 将把它解读成输出列名称。这与在同样情况下 GROUP BY 会做出的选择相反。这种不一致是为了与 SQL 标准兼容。

可以在 ORDER BY 子句中任一表达式之后附加关键字 ASC（升序）或 DESC（降序）。如果没有指定，ASC 被假定为默认值。或者，可以在USING 子句中指定一个特定的排序操作符名称。一个排序操作符必须是某个 B-树操作符族的小于或者大于成员。ASC 通常等价于 USING <而 DESC 通常等价于 USING >（但是一种用户定义数据类型的创建者可以准确地定义默认排序顺序是什么，并且它可能会对应于其他名称的操作符）。

如果指定 NULLS LAST，空值会排在非空值之后；如果指定 NULLS FIRST，空值会排在非空值之前。如果都没有指定，在指定或者隐含ASC 时的默认行为是 NULLS LAST，而指定DESC 时的默认行为是 NULLS FIRST（因此，默认行为是空值大于非空值）。当指定USING 时，默认的空值顺序取决于该操作符是否为小于或者大于操作符。

注意顺序选项只应用到它们所跟随的表达式上。例如 ORDER BY x, y DESC 和 ORDER BY x DESC, y DESC 是不同的。

字符串数据会按适用于被排序列的排序规则来排序。根据需要可以通过在 *expression* 中包括一个 COLLATE 子句来覆盖，例如 ORDER BY mycolumn COLLATE "en_US"。更多信息请见第 4.2.10 节和第 23.2 节。

LIMIT 子句

该 LIMIT 子句由两个独立的子句组成：

```
LIMIT { count | ALL }
OFFSET start
```

count 指定最多返回多少行，而 *start* 指定开始返回行之前要跳过的行数。如果两者都指定，则先跳过 *start* 行，然后开始计数并返回 *count* 行。

如果 *count* 表达式计算结果为 NULL，则视为 LIMIT ALL，即不限制。如果 *start* 计算结果为 NULL，则等同于 OFFSET 0。

SQL:2008 引入了另一种实现相同结果的语法，PostgreSQL 也支持它。其形式如下：

```
OFFSET start { ROW | ROWS }
FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY
```

在这种语法中，标准要求 *start* 或 *count* 的值必须是字面常量、参数或变量名；作为 PostgreSQL 的扩展，也允许其他表达式，但通常需要用圆括号将其括起以避免歧义。如果 *count* 在 FETCH 子句中被省略，则默认为 1。ROW 和 ROWS 以及 FIRST 和 NEXT 都是噪声词，不会影响这些子句的效果。按照标准，OFFSET 子句必须出现在 FETCH 子句之前（如果两者都存在）；而 PostgreSQL 更宽松，允许任意顺序。

在使用 LIMIT 时，用一个ORDER BY 子句把结果行约束到一个唯一顺序是个好办法。否则你将得到该查询结果行的一个不可预测的子集 – 你可能要求从第 10 到第 20 行，但是在什么顺序下的第 10 到第 20 呢？除非指定ORDER BY，你是不知道顺序的。

查询规划器在生成一个查询计划时会考虑 LIMIT，因此根据你使用的LIMIT 和 OFFSET，你很可能得到不同的计划（得到不同的行序）。所以，使用不同的 LIMIT/OFFSET 值来选择一个查询结果的不同子集将会给出不一致的结果，除非你用ORDER BY 强制一种可预测的结果顺序。这不是一个缺陷，它是 SQL 不承诺以任何特定顺序（除非使用 ORDER BY 来约束顺序）给出一个查询结果这一事实造成的必然后果。

如果没有一个 ORDER BY 来强制选择一个确定的子集，重复执行同样的LIMIT 查询甚至可能会返回一个表中行的不同子集。同样，这也不是一种缺陷，在这种情况下也无法保证结果的确定性。

锁定子句

FOR UPDATE、FOR NO KEY UPDATE、FOR SHARE 和 FOR KEY SHARE 是锁定子句，它们影响SELECT 把行从表中取得时如何对它们加锁。

锁定子句的一般形式为：

```
FOR lock_strength [ OF table_name [, ...] ] [ NOWAIT | SKIP LOCKED ]
```

其中，*lock_strength* 可以是以下值之一：

```
UPDATE
NO KEY UPDATE
SHARE
KEY SHARE
```

更多关于每一种行级锁模式的信息可见第 13.3.2 节。

为了防止该操作等待其他事务提交，可使用NOWAIT 或 SKIP LOCKED 选项。使用 NOWAIT 时，如果选中的行不能被立即锁定，该语句会直接报错而不是等待。使用 SKIP LOCKED 时，任何无法立即锁定的已选中行都会被跳过。跳过已锁定行会提供数据的不一致视图，因此不适合一般用途的工作，但可用于避免多个消费者访问类似队列表时的锁竞争。注意，NOWAIT 和 SKIP LOCKED 只适用于行级锁；所需的 ROW SHARE 表级锁仍会按常规方式取得（见第 13 章）。如果想要不等待的表级锁，你可以先使用带NOWAIT 的 LOCK。

如果在一个锁定子句中提到了特定的表，则只有来自于那些表的行会被锁定，任何SELECT 中用到的其他表还是被简单地照常读取。一个没有表列表的锁定子句会影响该语句中用到的所有表。如果一个锁定子句被应用到一个视图或者子查询，它会影响在该视图或子查询中用到的所有表。不过，这些子句不适用于主查询引用的WITH 查询。如果你希望在一个 WITH 查询中发生行锁定，应该在该 WITH 查询内指定一个锁定子句。

如果有必要对不同的表指定不同的锁定行为，可以写多个锁定子句。如果同一个表在多于一个锁定子句中被提到（或者被隐式地影响到），那么会按照所指定的最强的锁定行为来处理它。类似地，如果在任何影响一个表的子句中指定了NOWAIT，就会按照 NOWAIT 的行为来处理该表。否则如果 SKIP LOCKED 在任何影响该表的子句中被指定，该表就会被按照SKIP LOCKED 来处理。

如果返回的行无法清楚地与表中的各个行对应起来，就不能使用锁定子句。例如，锁定子句不能与聚合一起使用。

当一个锁定子句出现在一个 SELECT 查询的顶层时，被锁定的行正好就是该查询返回的行。在连接查询的情况下，被锁定的行是那些对返回的连接行有贡献的行。此外，在该查询的快照时满足查询条件的行将被锁定，如果它们在该快照后被更新并且不再满足查询条件，它们将不会被返回。如果使用了LIMIT，只要已经返回的行数满足了限制，锁定就会停止（但注意被 OFFSET 跳过的行将被锁定）。类似地，如果在一个游标的查询中使用锁定子句，只有被该游标实际取出或者跳过的行才将被锁定。

当锁定子句出现在一个子 SELECT 中时，被锁定行是那些该子查询返回给外层查询的行。这些被锁定的行的数量可能比从子查询自身的角度看到的要少，因为来自外层查询的条件可能会被用来优化子查询的执行。例如：

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

将只锁定具有 col1 = 5 的行（虽然在子查询中并没有写上该条件）。

较早的版本无法保持一个随后在保存点中被升级的锁。例如，下面这段代码：

```
BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;
```

在执行 ROLLBACK TO 之后将无法保持 FOR UPDATE 锁。这个问题已在 9.3 版本中修复。

小心

一个运行在 READ COMMITTED 事务隔离级别并且使用 ORDER BY 和锁定子句的 SELECT 命令有可能返回无序的行。这是因为 ORDER BY 会被首先应用。该命令对结果排序，但是可能接着在尝试获得一行或多行上的锁时阻塞。一旦SELECT 解除阻塞，某些排序列值可能已经被修改，从而导致那些行变成无序的（尽管它们根据原始列值是有序的）。根据需要，可以通过在子查询中放置 FOR UPDATE/SHARE 来解决这一问题，例如

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY column1;
```

注意这将导致锁定 mytable 的所有行，而顶层的 FOR UPDATE 只会锁定实际被返回的行。这可能会导致显著的性能差异，特别是把ORDER BY 与 LIMIT 或者其他限制组合使用时。因此只有在预计会并发更新排序列并且要求严格的排序结果时才推荐使用这种技术。

在 REPEATABLE READ 或 SERIALIZABLE 事务隔离级别下，这将导致串行化失败（ SQLSTATE 为'40001'），因此在这些隔离级别下不可能接收到无序的行。

TABLE 命令

命令

```
TABLE name
```

等价于

```
SELECT * FROM name
```

它可以作为顶层命令使用，也可以作为复杂查询中一种节省空间的语法变体。只有 WITH、UNION、INTERSECT、EXCEPT、ORDER BY、LIMIT、OFFSET、FETCH 以及 FOR 锁定子句可以用于 TABLE。不能使用 WHERE 子句和任何形式的聚合。

示例

要将表 `films` 与表 `distributors` 连接：

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d JOIN films f USING (did);

       title       | did |     name     | date_prod  |   kind
-------------------+-----+--------------+------------+----------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantic
 ...
```

要对所有电影的 `len` 列求和并且用 `kind` 对结果分组：

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;

   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
```

要对所有电影的 `len` 列求和、对结果按照 `kind` 分组并且显示总长小于 5 小时的分组：

```
SELECT kind, sum(len) AS total
    FROM films
    GROUP BY kind
    HAVING sum(len) < interval '5 hours';

   kind   | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
```

下面两个示例都是根据第二列（name）的内容来排序结果：

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

```
 did |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

接下来的示例展示了如何得到表 distributors 和 actors 的并集，把结果限制为那些在每个表中以字母 W 开始的行。这里只需要不重复的行，因此省略了关键词 ALL。

```
distributors:               actors:
 did |     name              id |     name
-----+--------------        ----+----------------
 108 | Westward                1 | Woody Allen
 111 | Walt Disney             2 | Warren Beatty
 112 | Warner Bros.            3 | Walter Matthau
 ...                          ...

SELECT distributors.name
    FROM distributors
    WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM actors
    WHERE actors.name LIKE 'W%';

      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

这个示例展示了如何在 FROM 子句中使用函数，分别使用和不使用列定义列表：

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors(111);
 did |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

下面是为函数结果增加序号列的示例：

```
SELECT * FROM unnest(ARRAY['a','b','c','d','e','f']) WITH ORDINALITY;
 unnest | ordinality
--------+----------
 a      |        1
 b      |        2
 c      |        3
 d      |        4
 e      |        5
 f      |        6
(6 rows)
```

这个例子展示了如何使用一个简单的 WITH 子句：

```
WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM t
UNION ALL
SELECT * FROM t

         x
--------------------
  0.534150459803641
  0.520092216785997
 0.0735620250925422
  0.534150459803641
  0.520092216785997
 0.0735620250925422
```

请注意，WITH 查询只被求值了一次，因此我们得到了两组相同的三个随机值。

这个示例使用 WITH RECURSIVE 从一个只显示直接下属的表中寻找雇员 Mary 的所有下属（直接的或者间接的）以及他们的间接层数：

```
WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
  UNION ALL
    SELECT er.distance + 1, e.employee_name, e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
  )
SELECT distance, employee_name FROM employee_recursive;
```

注意这种递归查询的典型形式：一个初始条件，后面跟着 UNION，然后是查询的递归部分。要确保查询的递归部分最终将不返回任何行，否则该查询将无限循环（更多示例见第 7.8 节）。

这个示例使用 LATERAL 为 manufacturers 表的每一行应用一个集合返回函数 get_product_names()：

```
SELECT m.name AS mname, pname
FROM manufacturers m, LATERAL get_product_names(m.id) pname;
```

当前没有任何产品的制造商不会出现在结果中，因为这是一个内连接。如果我们希望把这类制造商的名称包括在结果中，我们可以：

```
SELECT m.name AS mname, pname
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true;
```

兼容性

当然，SELECT 语句与 SQL 标准兼容。但它也有一些扩展和缺失的特性。

省略的 FROM 子句

PostgreSQL 允许省略 FROM 子句。它可以直接用于计算简单表达式的结果：

```
SELECT 2+2;

 ?column?
----------
        4
```

其他一些 SQL 数据库必须引入一个只有一行的虚拟表，才能从该表执行 SELECT。

请注意，如果未指定 FROM 子句，查询就不能引用任何数据库表。例如，以下查询是无效的：

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

PostgreSQL 在 8.1 之前的版本会接受这种形式的查询，并为查询引用的每个表在该查询的 FROM 子句中添加一个隐式项。现在不再允许这种做法。

空 SELECT 列表

SELECT 之后的输出表达式列表可以为空，这会产生一个零列的结果表。按照 SQL 标准，这不是合法的语法。PostgreSQL 允许这样做，是为了与允许零列的表保持一致。不过在使用 DISTINCT 时不允许空列表。

省略 AS 关键词

在 SQL 标准中，只要新列名是一个合法的列名（就是说与任何保留关键词不同），就可以省略输出列名之前的可选关键词AS。PostgreSQL 要稍微严格些：只要新列名匹配任何关键词（保留或者非保留）就需要 AS。推荐的习惯是使用 AS 或者带双引号的输出列名来防止与未来增加的关键词可能的冲突。

在 FROM 项中，标准和 PostgreSQL 都允许在非保留关键字别名前省略 AS。但是由于语法歧义，这种写法无法用于输出列名。

ONLY 与继承

SQL 标准要求在使用 ONLY 时用圆括号括起表名，例如SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE ...。PostgreSQL 认为这些圆括号是可选的。

PostgreSQL 允许在末尾写上*，显式指定包含子表的非ONLY 行为。标准不允许这种写法。

（这些要点同样适用于所有支持 ONLY 选项的 SQL 命令。）

TABLESAMPLE 子句的限制

目前只有普通表和物化视图可以使用 TABLESAMPLE 子句。按照 SQL 标准，它应当能够应用于任何FROM 项。

FROM 中的函数调用

PostgreSQL 允许把函数调用直接写成 FROM 列表中的一个成员。在 SQL 标准中，必须把这样的函数调用包装在一个子SELECT 中。也就是说，语法 FROM *func*(...) *alias* 近似等价于 FROM LATERAL (SELECT *func*(...)) *alias*。注意该 LATERAL 被认为是隐式的，这是因为标准对于 FROM 中的一个 UNNEST() 项要求 LATERAL 语义。PostgreSQL 会把 UNNEST() 和其他集合返回函数同样对待。

GROUP BY 和 ORDER BY 可用的名字空间

在 SQL-92 标准中，一个ORDER BY 子句只能使用输出列名或者序号，而一个GROUP BY 子句只能使用基于输入列名的表达式。PostgreSQL 扩展了这两种子句以允许它们使用其他的选择（但如果有歧义时还是使用标准的解释）。PostgreSQL 也允许两种子句指定任意表达式。注意出现在一个表达式中的名称将总是被当做输入列名而不是输出列名。

SQL:1999 及其后的标准使用了一种略微不同的定义，它并不完全向后兼容 SQL-92。不过，在大部分的情况下，PostgreSQL 会以与 SQL:1999 相同的方式解释 ORDER BY 或 GROUP BY 表达式。

函数依赖

只有当一个表的主键被包括在 GROUP BY 列表中时，PostgreSQL 才识别函数依赖（允许从 GROUP BY 中省略列）。SQL 标准指定了应该要识别的额外情况。

LIMIT 和 OFFSET

LIMIT 和 OFFSET 子句是 PostgreSQL 特有的语法，MySQL 也使用这种语法。SQL:2008 标准引入了 OFFSET ... FETCH {FIRST|NEXT} ...子句来实现相同功能，如上面的LIMIT 子句所示。IBM DB2 也使用这种语法。（为 Oracle 编写的应用经常采用一种变通办法，通过自动生成的rownum 列实现这些子句的效果，而 PostgreSQL 中没有这一列。）

FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHARE

尽管 SQL 标准中出现了 FOR UPDATE，但标准只允许它作为 DECLARE CURSOR 的一个选项。PostgreSQL 允许它出现在任何 SELECT 查询以及子 SELECT 中，但这是一种扩展。FOR NO KEY UPDATE、FOR SHARE 以及 FOR KEY SHARE 变体以及 NOWAIT 和 SKIP LOCKED 选项没有在标准中出现。

WITH 中的数据修改语句

PostgreSQL 允许将 INSERT、UPDATE 和 DELETE 用作 WITH 查询。SQL 标准中没有这种用法。

非标准子句

DISTINCT ON (...) 是 SQL 标准的扩展。

ROWS FROM(...) 是 SQL 标准的扩展。

SELECT INTO

SELECT INTO — 根据查询结果定义一个新表

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
    INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

描述

SELECT INTO 创建一个新表，并用查询计算得到的数据填充该表。与普通的SELECT 不同，这些数据不会返回给客户端。新表的列具有与SELECT 输出列对应的名称和数据类型。

参数

TEMPORARY 或 TEMP

如果指定，将把该表创建为临时表。详见 CREATE TABLE。

UNLOGGED

如果指定，该表将创建为不记录 WAL 的表。详见 CREATE TABLE。

new_table

要创建的表名（可选地带模式限定）。

所有其他参数都在 SELECT 中有详细说明。

注解

CREATE TABLE AS 在功能上与 SELECT INTO 类似。CREATE TABLE AS 是推荐使用的语法，因为这种形式的SELECT INTO 在 ECPG 或 PL/pgSQL 中不可用，因为它们对 INTO 子句有不同的解释。此外，CREATE TABLE AS 提供的功能是 SELECT INTO 所提供功能的超集。

要为 SELECT INTO 创建的表添加 OID，请启用default_with_oids 配置变量。也可以在 CREATE TABLE AS 中使用 WITH OIDS 子句。

示例

创建一个新表 `films_recent`，它只包含表 `films` 中的最近条目：

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

兼容性

SQL 标准使用 `SELECT INTO` 表示将值选入宿主程序的标量变量，而不是创建一个新表。这确实是 ECPG（见第 36 章）和 PL/pgSQL（见第 43 章）中的用法。PostgreSQL 使用 `SELECT INTO` 表示创建表则是历史遗留用法。对于新代码，最好为此目的使用 `CREATE TABLE AS`。

另见

CREATE TABLE AS

SET

SET — 更改运行时参数

大纲

```
SET [ SESSION | LOCAL ] configuration_parameter { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { value | 'value' | LOCAL | DEFAULT }
```

描述

SET 命令用于更改运行时配置参数。许多在第 19 章中列出的运行时参数都可以通过 SET 即时更改。（有些参数需要超级用户权限才能更改，还有一些参数在服务器启动或会话开始之后便无法再更改。）SET 只影响当前会话所使用的值。

如果在一个随后被中止的事务中发出 SET（或者等效的 SET SESSION），那么在事务回滚时，SET 命令的效果就会消失。一旦该事务提交，这些效果就会持续到会话结束，除非被另一个SET 覆盖。

SET LOCAL 的效果只持续到当前事务结束，无论事务是否提交。一种特殊情况是：在同一个事务中先执行 SET，再执行SET LOCAL。这样 SET LOCAL 设置的值会一直生效到事务结束，但在此之后（如果该事务被提交），SET 设置的值将生效。

SET 或 SET LOCAL 的效果，也会因为回滚到早于该命令的保存点而被取消。

如果在一个针对同一变量带有 SET 选项的函数内使用 SET LOCAL（见 CREATE FUNCTION），那么SET LOCAL 命令的效果会在函数退出时消失；也就是说，无论如何都会恢复函数被调用时生效的值。这使得SET LOCAL 可以在函数内部用于动态或重复地更改某个参数，同时仍能方便地利用 SET 选项保存并恢复调用者的值。不过，普通的 SET 命令会覆盖任何外围函数的 SET 选项；除非回滚，否则它的效果会一直保留。

> **注意**
>
> 在 PostgreSQL 8.0 到 8.2 版本中，SET LOCAL 的效果会因为释放一个更早的保存点，或者成功退出某个PL/pgSQL 异常块而被取消。由于这种行为被认为不够直观，因此后来对其进行了更改。

参数

SESSION

指定该命令对当前会话生效。（如果既没有出现 SESSION，也没有出现LOCAL，这就是默认情况。）

LOCAL

指定该命令只对当前事务生效。在 COMMIT 或 ROLLBACK 之后，会话级设置会再次生效。在事务块外发出该命令会发出警告，但除此之外没有效果。

configuration_parameter

可设置的运行时参数名称。可用参数见第 19 章以及下文。

value

参数的新值。根据具体参数的不同，值可以指定为字符串常量、标识符、数字，或由这些构成的逗号分隔列表。也可以写成DEFAULT，表示把该参数重置为其默认值（即当前会话中如果从未执行过 SET，它本应具有的值）。

除了第 19 章中记载的配置参数外，还有几个参数只能通过SET 命令调整，或者具有特殊语法：

SCHEMA

SET SCHEMA '*value*' 是 SET search_path TO *value* 的别名。使用这种语法时只能指定一个模式。

NAMES

SET NAMES *value* 是 SET client_encoding TO *value* 的别名。

SEED

设置随机数生成器（即函数 random）的内部种子。允许的值是 -1 到 1 之间的浮点数，这些值随后会乘以 2^{31}-1。

种子也可以通过调用函数 setseed 来设置：

```
SELECT setseed(value);
```

TIME ZONE

SET TIME ZONE '*value*' 是以下写法的别名：SET timezone TO '*value*'。语法 SET TIME ZONE 允许使用特殊语法指定时区。下面是一些有效值的示例：

'PST8PDT'

加利福尼亚州伯克利所使用的时区。

'Europe/Rome'

意大利所使用的时区。

-7

比 UTC 向西 7 小时的时区（等同于 PDT）。正值表示位于 UTC 以东。

INTERVAL '-08:00' HOUR TO MINUTE

比 UTC 向西 8 小时的时区（等同于 PST）。

LOCAL
DEFAULT

将时区设置为本地时区（即服务器 timezone 的默认值）。

以数字或时间间隔给出的时区设置在内部会被转换为 POSIX 时区语法。例如，在 SET TIME ZONE -7 之后，SHOW TIME ZONE 会报告<-07>+07。

SET 不支持时区缩写；关于时区的更多信息，见第 8.5.3 节。

注解

函数 set_config 提供了等效功能；见第 9.26 节。此外，也可以对 pg_settings 系统视图执行 UPDATE，以实现与SET 等效的操作。

示例

设置模式搜索路径：

```
SET search_path TO my_schema, public;
```

将日期风格设置为传统 POSTGRES 风格，并采用“日在月之前”的输入惯例：

```
SET datestyle TO postgres, dmy;
```

将时区设置为加利福尼亚州伯克利所使用的时区：

```
SET TIME ZONE 'America/Los_Angeles';
```

将时区设置为意大利所使用的时区：

```
SET TIME ZONE 'Europe/Rome';
```

兼容性

SET TIME ZONE 扩展了 SQL 标准定义的语法。标准只允许使用数值形式的时区偏移量，而 PostgreSQL 允许更灵活的时区指定方式。其他所有 SET 特性都是 PostgreSQL 扩展。

另见

RESET, SHOW

SET CONSTRAINTS

SET CONSTRAINTS — 为当前事务设置约束检查时机

大纲

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

描述

`SET CONSTRAINTS` 设置当前事务中约束检查的行为。`IMMEDIATE` 约束会在每条语句结束时检查。`DEFERRED` 约束要等到事务提交时才检查。每个约束都有各自的 `IMMEDIATE` 或 `DEFERRED` 模式。

创建约束时，它会被赋予以下三种特性之一：`DEFERRABLE INITIALLY DEFERRED`、`DEFERRABLE INITIALLY IMMEDIATE` 或者 `NOT DEFERRABLE`。第三类始终是 `IMMEDIATE`，不会受到 `SET CONSTRAINTS` 命令的影响。前两类在每个事务开始时都处于其指定的模式，但其行为可以在事务中通过 `SET CONSTRAINTS` 修改。

带约束名称列表的 `SET CONSTRAINTS` 只会修改这些约束的模式（它们都必须是可延迟的）。每个约束名称都可以带模式限定。如果未指定模式名称，就会使用当前模式搜索路径查找第一个匹配的名称。`SET CONSTRAINTS ALL` 会修改所有可延迟约束的模式。

当 `SET CONSTRAINTS` 将某个约束的模式从 `DEFERRED` 改为 `IMMEDIATE` 时，新模式具有追溯效力：任何原本会在事务结束时才检查的未决数据修改，都会改为在执行`SET CONSTRAINTS` 命令期间检查。如果违反了任何此类约束，`SET CONSTRAINTS` 就会失败（并且不会改变该约束的模式）。因此，可以利用`SET CONSTRAINTS` 强制在事务中的特定点执行约束检查。

当前，只有`UNIQUE`、`PRIMARY KEY`、`REFERENCES`（外键）以及 `EXCLUDE` 约束受到这个设置的影响。`NOT NULL` 和 `CHECK` 约束总是在一行被插入或修改时立即检查（不是在语句结束时）。未声明为 `DEFERRABLE` 的唯一约束和排他约束也会立即检查。

被声明为“约束触发器”的触发器，其引发也受此设置控制 – 它们会在相关约束应当被检查的同时引发。

注解

因为 PostgreSQL 并不要求约束名称在同一模式内唯一（只要求在每个表内唯一），所以指定的约束名称有可能匹配到多个约束。在这种情况下，`SET CONSTRAINTS` 会作用于所有匹配项。对于未带模式限定的名称，一旦在搜索路径中的某个模式里找到一个或多个匹配项，就不会再搜索路径中更靠后的模式。

这个命令只会改变当前事务中约束的行为。在事务块之外发出该命令会产生一条警告，除此之外不会有任何效果。

兼容性

这个命令符合 SQL 标准定义的行为，但有一个限制：在 PostgreSQL 中，它不会应用在 `NOT NULL` 和 `CHECK` 约束上。此外，PostgreSQL 会立即检查不可延迟的唯一约束，而不是像标准所暗示的那样在语句结束时检查。

SET ROLE

SET ROLE — 设置当前会话的当前用户标识符

大纲

```
SET [ SESSION | LOCAL ] ROLE role_name
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

描述

该命令把当前 SQL 会话的当前用户标识符设置为 *role_name*。角色名可以写成标识符或字符串字面量。在 SET ROLE 之后，SQL 命令的权限检查会视同指定角色是最初登录的角色。

当前会话用户必须是指定的 *role_name* 角色的成员。（如果会话用户是超级用户，则可以选择任意角色。）

SESSION 和 LOCAL 修饰符的作用与常规的 SET 命令相同。

SET ROLE NONE 将当前用户标识符设置为当前会话用户标识符，即session_user 返回的值。如果存在此类设置，RESET ROLE 会将当前用户标识符设置为连接建立时由命令行选项、ALTER ROLE 或 ALTER DATABASE 指定的设置。否则，RESET ROLE 会将当前用户标识符设置为当前会话用户标识符。这些形式可以由任何用户执行。

注解

使用这个命令，既可以增加权限，也可以限制自己的权限。如果会话用户角色具有 INHERIT 属性，它会自动拥有每个可通过 SET ROLE 切换到的角色的全部权限；在这种情况下，SET ROLE 实际上会去掉直接分配给会话用户及其所属其他角色的全部权限，只保留指定角色可用的权限。另一方面，如果会话用户角色具有 NOINHERIT 属性，SET ROLE 会放弃直接分配给会话用户的权限，转而获得指定角色可用的权限。

特别是，当超级用户选择通过 SET ROLE 切换到非超级用户角色时，会失去其超级用户权限。

SET ROLE 的效果与 SET SESSION AUTHORIZATION 相近，但其中涉及的权限检查完全不同。此外，SET SESSION AUTHORIZATION 会决定后续 SET ROLE 命令允许使用哪些角色，而通过 SET ROLE 更改角色并不会改变后续 SET ROLE 允许使用的角色集合。

SET ROLE 不会处理角色的 ALTER ROLE 设置所指定的会话变量；这只会在登录期间发生。

SET ROLE 不能在 SECURITY DEFINER 函数中使用。

示例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

兼容性

PostgreSQL 允许使用标识符语法（"*rolename*"），而 SQL 标准要求将角色名写成字符串字面量。SQL 不允许在事务中执行这个命令；PostgreSQL 不施加这一限制，因为没有理由这样做。`SESSION` 和 `LOCAL` 修饰符，以及 `RESET` 语法，都是PostgreSQL 扩展。

另见

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION — 设置当前会话的会话用户标识符和当前用户标识符

大纲

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION user_name
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

描述

这个命令将当前 SQL 会话的会话用户标识符和当前用户标识符设置为 *user_name*。用户名可以写成一个标识符或者一个字符串字面量。例如，可以使用这个命令临时变为一个非特权用户，之后再切换回超级用户。

会话用户标识符初始时被设置为客户端提供的（可能已认证的）用户名。当前用户标识符通常与会话用户标识符相同，但可能在 SECURITY DEFINER 函数和类似机制的上下文中临时更改；它也可以通过 SET ROLE 更改。当前用户标识符用于权限检查。

只有当初始会话用户（即已认证用户）最初具有超级用户权限时，才能更改会话用户标识符。否则，只有当该命令指定的是已认证用户名时，才会被接受。

SESSION 和 LOCAL 修饰符发挥的作用和常规 SET 命令一样。

DEFAULT 和 RESET 形式会将会话用户标识符和当前用户标识符重置为最初已认证的用户名。这些形式可以由任何用户执行。

注解

SET SESSION AUTHORIZATION 不能在一个 SECURITY DEFINER 函数中使用。

示例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

兼容性

SQL 标准允许在字面值 *user_name* 的位置上使用某些其他表达式，但这些选项在实践中并不重要。PostgreSQL 允许使用标识符语法（"*username*"），而 SQL 标准不允许。SQL 不允许在事务中使用这个命令；PostgreSQL 并不做此限制，因为没有理由这样做。`SESSION` 和 `LOCAL` 修饰符以及 `RESET` 语法都是 PostgreSQL 扩展。

标准把执行这个命令所需的权限留给实现定义。

另见

SET ROLE

SET TRANSACTION

SET TRANSACTION — 设置当前事务的特性

大纲

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

其中 *transaction_mode* 是下列之一：

```
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

`SET TRANSACTION` 命令设置当前事务的特性。它对任何后续事务都没有影响。`SET SESSION CHARACTERISTICS` 设置一个会话中后续事务的默认事务特性。对于单个事务，这些默认值可以用 `SET TRANSACTION` 覆盖。

可用的事务特性是事务隔离级别、事务访问模式（读/写或只读）以及可延迟模式。此外，还可以选择一个快照，不过它只能用于当前事务，不能作为会话默认值。

一个事务的隔离级别决定当其他事务并发运行时该事务能看见什么数据：

`READ COMMITTED`

一条语句只能看到在它开始之前已提交的行。这是默认值。

`REPEATABLE READ`

当前事务中的所有语句都只能看到在该事务中执行第一条查询或数据修改语句之前已提交的行。

`SERIALIZABLE`

当前事务中的所有语句都只能看到在该事务中执行第一条查询或数据修改语句之前已提交的行。如果并发的可串行化事务之间出现某种读写模式，而这种情况不可能在这些事务的任何串行（一次执行一个）执行中发生，那么其中一个事务将以 `serialization_failure` 错误回滚。

SQL 标准还定义了一个额外的级别：`READ UNCOMMITTED`。在 PostgreSQL 中，`READ UNCOMMITTED` 被视为 `READ COMMITTED`。

在事务执行第一条查询或数据修改语句（`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, 或 `COPY`）之后，事务隔离级别就不能再更改。有关事务隔离和并发控制的更多信息，请参见第 13 章。

事务访问模式决定事务是读/写还是只读。读/写是默认值。当事务为只读时，下列 SQL 命令会被禁止：`INSERT`、`UPDATE`、`DELETE` 以及 `COPY FROM`，前提是它们要写入的表不是临时表；所有 `CREATE`、`ALTER` 和 `DROP` 命令；`COMMENT`、`GRANT`、`REVOKE`、`TRUNCATE`；以及

EXPLAIN ANALYZE 和 EXECUTE，前提是它们要执行的命令属于上述列表。这是一个较高层面的只读概念，并不会阻止所有写入磁盘的行为。

只有当事务同时是 SERIALIZABLE 和 READ ONLY 时，DEFERRABLE 事务属性才会生效。当为一个事务同时选择这三个属性时，该事务在首次获取其快照时可能会阻塞；在此之后，它便可以运行，而无需承担普通 SERIALIZABLE 事务的常规开销，也不会有促成串行化失败或因串行化失败而被取消的风险。这种模式非常适合长时间运行的报表或备份。

SET TRANSACTION SNAPSHOT 命令允许一个新事务使用与现有事务相同的快照运行。已有事务必须用 pg_export_snapshot 函数导出其快照（参见第 9.26.5 节）。该函数会返回一个快照标识符，必须将其提供给SET TRANSACTION SNAPSHOT 以指定要导入的快照。该标识符在此命令中必须写成字符串字面量，例如 '00000003-0000001B-1'。SET TRANSACTION SNAPSHOT 只能在事务开始时执行，也就是在事务的第一条查询或数据修改语句（SELECT、INSERT、DELETE、UPDATE、FETCH 或 COPY）之前。此外，事务还必须已经设置为 SERIALIZABLE 或 REPEATABLE READ 隔离级别（否则，快照会被立即丢弃，因为 READ COMMITTED 模式会为每条命令获取一个新快照）。如果导入事务使用 SERIALIZABLE 隔离级别，则导出快照的事务也必须使用该隔离级别。此外，非只读的可串行化事务不能从只读事务导入快照。

注解

如果执行 SET TRANSACTION 之前没有 START TRANSACTION 或者 BEGIN，它会发出一条警告，除此之外没有任何效果。

可以不使用 SET TRANSACTION，而是在 BEGIN 或 START TRANSACTION 中指定所需的 *transaction_modes*。但是，对于SET TRANSACTION SNAPSHOT，这种方式不可用。

会话默认的事务模式也可以通过配置参数 default_transaction_isolation、default_transaction_read_only 和 default_transaction_deferrable 来设置或检查（实际上，SET SESSION CHARACTERISTICS 只是用 SET 设置这些变量的一种更冗长的等价写法）。这意味着可以通过配置文件、ALTER DATABASE 等方式设置默认值。详见第 19 章。

当前事务的模式也可以类似地通过配置参数 transaction_isolation、transaction_read_only 和 transaction_deferrable 来设置或检查。设置其中任一参数的作用都与对应的 SET TRANSACTION 选项相同，并且在可设置时机方面有相同的限制。但是，这些参数不能在配置文件中设置，也不能通过实时执行的 SQL 之外的任何来源来设置。

示例

要以与某个现有事务相同的快照开始一个新事务，先从该现有事务导出快照。这样会返回快照标识符，例如：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
 00000003-0000001B-1
(1 row)
```

然后在新开启事务的开始处，通过`SET TRANSACTION SNAPSHOT`命令给出该快照标识符：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

兼容性

这些命令由 SQL 标准定义，但`DEFERRABLE`事务模式和`SET TRANSACTION SNAPSHOT`这种形式除外，它们是 PostgreSQL 扩展。

`SERIALIZABLE`是标准中的默认事务隔离级别。在 PostgreSQL 中，默认值通常是`READ COMMITTED`，但你可以按上述方式修改。

在 SQL 标准中，还可以用这些命令设置另一项事务特性：诊断区域的大小。这个概念特定于嵌入式 SQL，因此没有在 PostgreSQL 服务器中实现。

SQL 标准要求在连续的 *`transaction_modes`* 之间有逗号，但出于历史原因，PostgreSQL 允许省略逗号。

SHOW

SHOW — 显示一个运行时参数的值

大纲

```
SHOW name
SHOW ALL
```

描述

SHOW 将显示运行时参数的当前设置。这些变量可以使用 SET 语句来设置，也可以通过编辑 postgresql.conf 配置文件、使用 PGOPTIONS 环境变量（在使用 libpq 或基于 libpq 的应用时），或者在启动postgres 服务器时通过命令行标志来设置。详见第 19 章。

参数

name

运行时参数的名称。可用参数的说明见第 19 章以及 SET 参考页。此外，还有几个可以显示但不能设置的参数：

SERVER_VERSION

显示服务器的版本号。

SERVER_ENCODING

显示服务器端字符集编码。目前，该参数可以显示但不能设置，因为编码在数据库创建时就已确定。

LC_COLLATE

显示数据库用于排序规则（文本排序）的区域设置。目前，该参数可以显示但不能设置，因为该设置在数据库创建时就已确定。

LC_CTYPE

显示数据库用于字符分类的区域设置。目前，该参数可以显示但不能设置，因为该设置在数据库创建时就已确定。

IS_SUPERUSER

如果当前角色具有超级用户权限，则为真。

ALL

显示所有配置参数的值及其描述。

注解

函数 `current_setting` 也会产生等效的输出，见第 9.26 节。另外，`pg_settings` 系统视图也会给出相同的信息。

示例

显示参数 `DateStyle` 的当前设置：

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

显示参数 `geqo` 的当前设置：

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

显示所有设置：

```
SHOW ALL;
            name          | setting |                 description
--------------------------+---------+-------------------------------------------------
 allow_system_table_mods | off     | Allows modifications of the structure of ...
    .
    .
    .
 xmloption               | content | Sets whether XML data in implicit parsing ...
 zero_damaged_pages      | off     | Continues processing past damaged page headers.
(196 rows)
```

兼容性

`SHOW` 命令是一种 PostgreSQL 扩展。

另见

SET, RESET

START TRANSACTION

START TRANSACTION — 开始一个事务块

大纲

```
START TRANSACTION [ transaction_mode [, ...] ]
```

其中 *transaction_mode* 是下列之一:

```
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

该命令开始一个新的事务块。如果指定了隔离级别、读写模式或可延迟模式，新事务就会具有这些特性，就像执行了SET TRANSACTION 一样。它与 BEGIN 命令具有相同功能。

参数

关于本语句参数含义的信息，参见 SET TRANSACTION。

兼容性

在 SQL 标准中，并不需要发出START TRANSACTION 来开始一个事务块：任何 SQL 命令都会隐式开始一个事务块。PostgreSQL 的行为可以看作是：对每条不跟在 START TRANSACTION（或 BEGIN）之后的命令，都会在其后隐式发出一个COMMIT，因此这种行为通常被称为“自动提交”。其他关系型数据库系统也可能为了方便而提供自动提交特性。

DEFERRABLE *transaction_mode* 是 PostgreSQL 的一种语言扩展。

SQL 标准要求在连续的 *transaction_modes* 之间有逗号，但出于历史原因，PostgreSQL 允许省略逗号。

另见 SET TRANSACTION 的兼容性小节。

另见

BEGIN, COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION

TRUNCATE

TRUNCATE — 清空一个表或一组表

大纲

```
TRUNCATE [ TABLE ] [ ONLY ] name [ * ] [, ... ]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

描述

TRUNCATE 可以快速移除一组表中的所有行。它的效果与对每个表执行不带条件的 DELETE 相同，但由于它并不实际扫描这些表，因此速度更快。此外，它会立即回收磁盘空间，而不是要求后续再执行一次VACUUM 操作。这一点对于大表尤其有用。

参数

name

要截断的表名（可以是模式限定的）。如果在表名前指定了 ONLY，则只截断该表。如果未指定ONLY，则该表及其所有后代表（如果有）都会被截断。也可以在表名后指定 *，以显式表明包含后代表。

RESTART IDENTITY

自动重新启动由被截断表列拥有的 sequence。

CONTINUE IDENTITY

不更改 sequence 的值。这是默认值。

CASCADE

自动截断所有通过外键引用指定表的表，以及通过外键引用因 CASCADE 而加入截断范围的表的其他表。

RESTRICT

如果任一表具有来自命令中未列出表的外键引用，则拒绝截断。这是默认值。

注解

要截断一个表，你必须拥有该表上的TRUNCATE 权限。

TRUNCATE 会在其操作的每个表上获取 ACCESS EXCLUSIVE 锁，这会阻塞该表上的所有其他并发操作。当指定RESTART IDENTITY 时，任何需要重新启动的 sequence 也会同样被排他锁定。如果需要对某个表进行并发访问，则应改用 DELETE 命令。

TRUNCATE 不能用于被其他表通过外键引用的表，除非所有这类表也在同一条命令中被截断。因为在这种情况下检查有效性将需要扫描表，而该命令的意义恰恰在于避免这样做。CASCADE 选项

可用于自动包含所有依赖表 – 但使用该选项时一定要非常小心，否则你可能会丢失并非有意删除的数据！

`TRUNCATE` 不会引发这些表上可能存在的任何 `ON DELETE` 触发器，但会引发 `ON TRUNCATE` 触发器。如果这些表中的任何一个定义了 `ON TRUNCATE` 触发器，那么所有 `BEFORE TRUNCATE` 触发器都会在任何截断发生之前引发，而所有`AFTER TRUNCATE` 触发器都会在最后一次截断完成且所有 sequence 被重置之后引发。触发器会按照表的处理顺序引发（先是命令中列出的表，然后是由于级联而加入的表）。

`TRUNCATE` 不是多版本并发控制（MVCC）安全的。截断之后，如果并发事务使用的是在截断发生前取得的快照，该表对这些并发事务而言将表现为空。详见第 13.5 节。

就表中数据而言，`TRUNCATE` 是事务安全的：如果外围事务没有提交，截断将被安全地回滚。

当指定 `RESTART IDENTITY` 时，隐含的 `ALTER SEQUENCE RESTART` 操作也会以事务方式执行；也就是说，如果外围事务没有提交，它们也会被回滚。请注意，如果在事务回滚前又对这些重启后的 sequence 执行了额外操作，这些操作对 sequence 本身的影响会被回滚，但对 `currval()` 的影响不会被回滚。也就是说，事务结束后，`currval()` 仍将反映失败事务内取得的最后一个 sequence 值，即使 sequence 本身可能已经不再与之保持一致。这与失败事务之后 `currval()` 的通常行为类似。

目前，`TRUNCATE` 不支持外部表。这意味着，如果指定表的后代表中有外部表，该命令就会失败。

示例

截断表 `bigtable` 和 `fattable`：

```
TRUNCATE bigtable, fattable;
```

同样的操作，并重置所有相关联的 sequence 生成器：

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

截断表 `othertable`，并级联到任何通过外键约束引用`othertable` 的表：

```
TRUNCATE othertable CASCADE;
```

兼容性

SQL:2008 标准包括了一个 `TRUNCATE` 命令，语法是`TRUNCATE TABLE` *`tablename`*。子句 `CONTINUE IDENTITY`/`RESTART IDENTITY` 在该标准中也有出现，但其含义虽相关却略有不同。该命令的某些并发行为在标准中被留作实现定义，因此必要时应结合上述注解并与其他实现进行比较。

另见

DELETE

UNLISTEN

UNLISTEN — 停止监听通知

大纲

```
UNLISTEN { channel | * }
```

描述

UNLISTEN 用于移除现有的 NOTIFY 事件注册。UNLISTEN 会取消当前 PostgreSQL 会话作为名为 *channel* 的通知通道监听者的任何现有注册。特殊通配符* 会取消当前会话的所有监听注册。

NOTIFY 中对 LISTEN 和 NOTIFY 的使用有更详细的讨论。

参数

channel

通知通道的名称（任意标识符）。

*

该会话当前的所有监听注册都会被清除。

注解

对自己并未监听的通道执行 UNLISTEN，不会出现警告或错误。

每个会话结束时，都会自动执行UNLISTEN *。

执行过 UNLISTEN 的事务不能进入两阶段提交的预备状态。

示例

建立一个监听注册：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

执行 UNLISTEN 之后，后续的NOTIFY 消息将被忽略：

```
UNLISTEN virtual;
NOTIFY virtual;
-- no NOTIFY event is received
```

兼容性

SQL 标准中没有 `UNLISTEN` 语句。

另见

LISTEN, NOTIFY

UPDATE

UPDATE — 更新表中的行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] ) |
          ( column_name [, ...] ) = ( sub-SELECT )
        } [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

描述

UPDATE 会更改所有满足条件的行中指定列的值。只需在 SET 子句中提及需要修改的列；未被显式修改的列会保留其原有值。

有两种方法可以利用数据库中其他表所包含的信息来修改一个表：使用子选择，或者在FROM 子句中指定附加表。哪种技术更合适取决于具体情况。

可选的 RETURNING 子句使 UPDATE 基于每个实际更新的行计算并返回一个或多个值。可以计算任何使用该表列和/或 FROM 中提到的其他表列的表达式。使用该表列的新值（更新后值）。RETURNING 列表的语法与 SELECT 的输出列表相同。

必须拥有该表上的 UPDATE 权限，或者至少拥有待更新列的UPDATE 权限。对于其值会在 *expressions* 或者 *condition* 中读取的任何列，还必须拥有SELECT 权限。

参数

with_query

WITH 子句允许你指定一个或多个子查询，这些子查询可在UPDATE 查询中按名称引用。详见第 7.8 节和 SELECT。

table_name

要更新的表名（可以是模式限定的）。如果在表名前指定了 ONLY，只会更新所提及表中的匹配行。如果未指定 ONLY，还会更新继承自该表的任何表中的匹配行。可选地，可以在表名后指定*，以显式指示包含后代表。

alias

目标表的替代名称。提供别名时，它会完全隐藏该表的实际名称。例如，给定 UPDATE foo AS f，UPDATE 语句的其余部分必须将该表称为 f，而不是foo。

column_name

由 *table_name* 命名的表中的列名。如有需要，列名可以使用子字段名或数组下标进行限定。指定目标列时不要包含表名 – 例如，`UPDATE table_name SET table_name.col = 1` 是无效的。

expression

要赋给该列的表达式。该表达式可以使用该表中这一列和其他列的旧值。

`DEFAULT`

把该列设置为其默认值（如果没有为它指定特定的默认表达式，则该值为 NULL）。

sub-SELECT

一个 `SELECT` 子查询，其输出列数必须与它前面圆括号中列列表的列数相同。执行时，该子查询返回的行数不得超过一行。如果返回一行，则其列值会赋给目标列；如果不返回任何行，则把 NULL 值赋给目标列。该子查询可以引用正在更新的表的当前行的旧值。

from_item

一个表表达式，允许其他表的列出现在`WHERE` 条件和更新表达式中。这里使用的语法与 `FROM` 子句相同，后者属于`SELECT` 语句；例如，可以为表名指定别名。不要把目标表重复写成*from_item*，除非你打算进行自连接（这种情况下它必须在*from_item* 中带有别名出现）。

condition

一个返回 `boolean` 类型值的表达式。只有使这个表达式返回 `true` 的行才会被更新。

cursor_name

要在 `WHERE CURRENT OF` 条件中使用的游标名。要被更新的行是最近一次从该游标中取出的那一行。该游标必须是针对 `UPDATE` 目标表的非分组查询。注意，`WHERE CURRENT OF` 不能与布尔条件同时指定。有关将游标用于 `WHERE CURRENT OF` 的更多信息，请见 DECLARE。

output_expression

在每一行被更新后，由`UPDATE` 命令计算并返回的表达式。该表达式可以使用由 *table_name* 命名的表或 `FROM` 中列出的表中的任何列名。写成 `*` 可返回所有列。

output_name

用于返回列的名称。

输出

在成功完成时，一个UPDATE 命令会返回以下形式的命令标签：

```
UPDATE count
```

count 是被更新的行数，包括值没有改变的匹配行。注意，当更新被BEFORE UPDATE 触发器抑制时，这个数量可能小于匹配 *condition* 的行数。如果 *count* 为 0，则该查询没有更新任何行（这不被视为错误）。

如果 UPDATE 命令包含 RETURNING 子句，其结果将类似于一个SELECT 语句，其中包含 RETURNING 列表中定义的列和值，并在该命令更新的行上进行计算。

注解

当存在 FROM 子句时，本质上会将目标表与 *from_item* 列表中提到的表连接起来，而连接的每一条输出行都代表对目标表的一次更新操作。使用 FROM 时，应确保对每一个要修改的目标行，连接至多生成一条输出行。换言之，一条目标行不应与其他表中的多于一行成功连接。如果发生这种情况，则只会使用其中某一条连接行来更新目标行，但具体使用哪一条并不容易预测。

因为存在这种不确定性，所以仅在子选择中引用其他表会更安全，尽管这种写法通常比使用连接更难阅读、也更慢。

对于分区表，更新一行可能会导致它不再满足其所在分区的分区约束。在这种情况下，如果在分区树中存在另一个分区，且该行满足它的分区约束，则该行会被移动到那个分区。如果不存在这样的分区，则会报错。在幕后，行移动实际上是一次DELETE 和一次 INSERT 操作。

在被移动的行上并发执行 UPDATE 或 DELETE 时，有可能收到串行化失败错误。假设会话 1 正在对某个分区键执行UPDATE，与此同时，一个能够看到该行的并发会话 2 对该行执行UPDATE 或 DELETE 操作。在这种情况下，会话 2 的UPDATE 或 DELETE 将检测到行移动，并引发串行化失败错误（其 SQLSTATE 代码始终为'40001'）。如果发生这种情况，应用程序可能需要重试事务。在表未分区或没有发生行移动的通常情况下，会话 2 会识别出新更新的那一行，并在这个新行版本上执行 UPDATE/DELETE。

请注意，虽然行可以从本地分区移动到外部表分区（前提是外部数据包装器支持元组路由），但不能从外部表分区移动到另一个分区。

示例

把表 films 的 kind 列中的单词 Drama 改为 Dramatic：

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

在表 weather 的一行中调整温度项，并将降水量重置为其默认值：

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

执行同样的操作，并返回更新后的各项：

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_lo, temp_hi, prcp;
```

使用另一种列列表语法完成同样的更新：

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

使用 `FROM` 子句语法，将负责 Acme Corporation 账户的销售人员的销量计数加一：

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
  WHERE accounts.name = 'Acme Corporation'
  AND employees.id = accounts.sales_person;
```

在 `WHERE` 子句中使用子选择，执行同样的操作：

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

更新账户表中的联系人姓名，使其与当前分配的销售人员保持一致：

```
UPDATE accounts SET (contact_first_name, contact_last_name) =
    (SELECT first_name, last_name FROM salesmen
     WHERE salesmen.id = accounts.sales_id);
```

使用连接也可以得到类似结果：

```
UPDATE accounts SET contact_first_name = first_name,
                    contact_last_name = last_name
  FROM salesmen WHERE salesmen.id = accounts.sales_id;
```

但是，第二个查询可能会给出意外结果，如果 `salesmen.id` 不是唯一键的话；而第一个查询在存在多个 `id` 匹配时保证会报错。此外，如果某个特定的 `accounts.sales_id` 条目没有匹配项，第一个查询会把相应的姓名字段设为 NULL，而第二个查询则根本不会更新该行。

更新汇总表中的统计数据以匹配当前数据：

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =
    (SELECT sum(x), sum(y), avg(x), avg(y) FROM data d
     WHERE d.group_id = s.group_id);
```

尝试插入一个新库存项及其库存量。如果该项已存在，则改为更新现有项的库存量。要在不致使整个事务失败的情况下做到这一点，请使用保存点：

```
BEGIN;
-- 其他操作
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- 假设上面的语句因唯一键冲突而失败，
-- 那么现在发出这些命令：
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- 继续执行其他操作，最后
COMMIT;
```

更改表 `films` 中游标 `c_films` 当前定位那一行的 `kind` 列：

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

兼容性

这个命令符合 SQL 标准，不过 `FROM` 和 `RETURNING` 子句是 PostgreSQL 扩展，把`WITH` 与 `UPDATE` 一起使用的能力也是扩展。

某些其他数据库系统提供一种 `FROM` 选项，要求在 `FROM` 中再次列出目标表。PostgreSQL 并不是这样解释 `FROM` 的。在移植使用这种扩展的应用时要小心。

根据标准，目标列名的一个圆括号子列表的源值可以是任何能够产生正确列数的行值表达式。PostgreSQL 只允许该源值是一个行构造器或子 `SELECT`。在行构造器的情况下，单个列的更新值可以指定为`DEFAULT`，但在子`SELECT` 中则不能这样做。

VACUUM

VACUUM — 垃圾收集并按需分析数据库

大纲

```
VACUUM [ ( option [, ...] ) ] [ table_and_columns [, ...] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ ANALYZE ] [ table_and_columns [, ...] ]
```

其中 *option* 为以下之一:

```
    FULL
    FREEZE
    VERBOSE
    ANALYZE
    DISABLE_PAGE_SKIPPING
```

而 *table_and_columns* 为:

```
    table_name [ ( column_name [, ...] ) ]
```

描述

VACUUM 回收死元组占用的存储空间。在正常的 PostgreSQL 运行中，被删除或因更新而过时的元组并不会从其表中物理移除；它们会一直保留，直到执行 VACUUM。因此有必要定期执行 VACUUM，尤其是对频繁更新的表。

如果没有给出 *table_and_columns* 列表，VACUUM 会处理当前数据库中当前用户有权清理的每个表和物化视图。如果给出了列表，VACUUM 则只处理其中列出的表。

VACUUM ANALYZE 会对每个选定表先执行 VACUUM，再执行ANALYZE。这种便捷的组合形式很适合例行维护脚本使用。关于其处理细节，参见 ANALYZE。

普通的 VACUUM（不带 FULL）只是回收空间并使其可被重用。这种形式的命令可以与表的正常读写并行运行，因为它不会获得独占锁。不过，在大多数情况下，额外空间不会返还给操作系统；它只是保留在同一张表内供再次使用。VACUUM FULL 会把表的全部内容重写到一个没有额外空闲空间的新磁盘文件中，从而让未使用的空间能够返还给操作系统。这种形式要慢得多，并且在处理每个表时都需要 ACCESS EXCLUSIVE 锁。

当选项列表用圆括号括起来时，选项可以按任意顺序书写。如果不加圆括号，则必须严格按照上面所示的顺序指定选项。带圆括号的语法是在PostgreSQL 9.0 中加入的；不带圆括号的语法已被弃用。

参数

FULL

选择“完全”清理，它可以回收更多空间，但耗时更长，并且会独占锁定该表。这种方法还需要额外的磁盘空间，因为它会写出该表的一个新副本，并且在操作完成之前不会释放旧副本。通常，只有当需要从表内回收大量空间时才应使用这种方法。

FREEZE

选择激进的元组“冻结”。指定 FREEZE 等价于执行一个将 vacuum_freeze_min_age 和 vacuum_freeze_table_age 参数设为零的 VACUUM。在表被重写时总会执行激进冻结，因此指定了 FULL 时，这个选项就是多余的。

VERBOSE

为每个表输出详细的清理活动报告。

ANALYZE

更新规划器用来确定查询最高效执行方式的统计信息。

DISABLE_PAGE_SKIPPING

通常，VACUUM 会根据可见性映射跳过某些页面。已知其中所有元组都已冻结的页面总是可以跳过，而已知其中所有元组都对所有事务可见的页面，也可以跳过，除非正在执行激进清理。此外，除非正在执行激进清理，为了避免等待其他会话结束对页面的使用，也可能会跳过某些页面。此选项会禁用所有跳页行为，只应在怀疑可见性映射内容存在问题时使用；而这种情况应当只会在硬件或软件问题导致数据库损坏时发生。

table_name

要清理的特定表或物化视图的名称（可选地带模式限定）。如果指定的表是分区表，则清理它的所有叶子分区。

column_name

要分析的特定列名。默认会分析所有列。如果指定了列列表，则也必须指定ANALYZE。

输出

指定 VERBOSE 时，VACUUM 会输出进度消息，指示当前正在处理哪个表，同时还会打印这些表的各种统计信息。

注解

要清理一个表，通常调用者必须是该表的拥有者或超级用户。不过，数据库拥有者可以清理其数据库中的所有表，但共享系统目录除外。（对共享系统目录的这一限制意味着，真正意义上的全数据库 VACUUM 只能由超级用户执行。）VACUUM 会跳过调用用户无权清理的任何表。

VACUUM 不能在一个事务块内被执行。

对于带有 GIN 索引的表，VACUUM（任何形式）还会通过把待处理的索引项移动到主 GIN 索引结构中的适当位置，来完成所有挂起的索引插入。详见第 66.4.1 节。

我们建议对活跃的生产数据库频繁执行清理（至少每晚一次），以移除死行。在添加或删除大量行之后，对受影响的表执行VACUUM ANALYZE 命令可能是个好主意。这会用所有近期变更的结果更新系统目录，使PostgreSQL 查询规划器在规划查询时能够做出更好的选择。

FULL 选项不建议在日常场景中使用，但在某些特殊情况下可能很有用。例如，当你删除或更新了表中的绝大多数行，并希望该表在物理上收缩以占用更少磁盘空间、让表扫描更快时，这个选项就比较合适。VACUUM FULL 通常会比普通 VACUUM 更大幅度地收缩表。

VACUUM 会显著增加 I/O 流量，这可能导致其他活动会话性能变差。因此，有时建议使用基于代价的清理延迟特性。详见第 19.4.4 节。

PostgreSQL 提供了一个“自动清理（autovacuum）”机制，可以自动执行常规清理维护。有关自动与手动清理的更多信息，参见第 24.1 节。

示例

清理单个表 onek，对其进行分析以供优化器使用，并打印详细的清理活动报告：

```
VACUUM (VERBOSE, ANALYZE) onek;
```

兼容性

在 SQL 标准中没有 VACUUM 语句。

另见

vacuumdb, 第 19.4.4 节, 第 24.1.6 节

VALUES

VALUES — 计算一组行

大纲

```
VALUES ( expression [, ...] ) [, ...]
    [ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
```

描述

VALUES 计算由值表达式指定的一个行值或一组行值。它最常用于在更大的命令中生成一个“常量表”，但也可以单独使用。

当指定多于一行时，所有行都必须具有相同数量的元素。结果表各列的数据类型由出现在该列中的表达式的显式类型或推断类型组合决定，所用规则与 UNION 相同（见第 10.5 节）。

在更大的命令中，从语法上说，凡是允许出现SELECT 的地方，也都允许出现VALUES。因为在语法中它被当作 SELECT 处理，所以可以对VALUES 命令使用 ORDER BY、LIMIT（或者等效的 FETCH FIRST）以及 OFFSET 子句。

参数

expression

要在结果表（行集合）中相应位置计算并插入的常量或表达式。在出现在 INSERT 顶层的 VALUES 列表中，*expression* 可以用 DEFAULT 替代，以表示应插入目标列的默认值。当 VALUES 出现在其他上下文中时，不能使用 DEFAULT。

sort_expression

一个指示结果行如何排序的表达式或整数常量。该表达式可以将 VALUES 结果的列引用为 column1、column2 等。详细信息参见 ORDER BY 子句。

operator

一个排序操作符。详细信息参见 ORDER BY 子句。

count

返回的最大行数。详细信息参见 LIMIT 子句。

start

开始返回行之前要跳过的行数。详细信息参见 LIMIT 子句。

注解

应避免使用行数非常多的 VALUES 列表，因为这可能导致内存不足错误或性能不佳。出现在 INSERT 中的 VALUES 属于一种特殊情况（因为所需的列类型可以从 INSERT 的目标表得知，并且无须通过扫描 VALUES 列表来推断），因此它能处理比其他上下文中实际可行的更大的列表。

示例

一个独立的 VALUES 命令：

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

这将返回一个两列三行的表。它实际上等效于：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

更常见的是，VALUES 会用在更大的 SQL 命令中。最常见的用法是在 INSERT 中：

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

在 INSERT 的上下文中，VALUES 列表中的项可以写成 DEFAULT，表示此处应使用该列的默认值，而不是显式指定一个值：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

VALUES 也可以用在原本可以编写子 SELECT 的地方，例如在FROM 子句中：

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
  WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
  WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

注意，在FROM 子句中使用 VALUES 时，需要有一个AS 子句，这一点与SELECT 相同。虽然 AS 子句不要求为所有列都指定名称，但这样做是一个好习惯。（在PostgreSQL 中，VALUES 的默认列名是 column1、column2 等，但在其他数据库系统中，这些名称可能不同。）

当在 INSERT 中使用 VALUES 时，所有值都会自动转换为相应目标列的数据类型。当它用于其他上下文时，可能有必要指定正确的数据类型。如果各项都是带引号的字符串常量，只需转换第一项，就足以为所有项确定假定的数据类型：

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'), ('192.168.1.43'));
```

> **提示**
>
> 对于简单的 IN 测试，最好使用IN 的标量列表形式，而不是写一个像上面那样的VALUES 查询。标量列表方法写法更简洁，而且通常效率更高。

兼容性

VALUES 符合 SQL 标准。LIMIT 和 OFFSET 是 PostgreSQL 扩展，另见 SELECT。

另见

INSERT, SELECT

PostgreSQL 客户端应用

本部分包含 PostgreSQL 客户端应用和工具的参考信息。并非所有这些命令都适用于一般用途；有些可能需要特殊权限。这些应用的共同点是，它们都可以在任意主机上运行，而不受数据库服务器所在位置的限制。

在命令行中指定用户名和数据库名时，其大小写会被保留 – 如果其中包含空格或特殊字符，可能需要加引号。除非文档另有说明，表名和其他标识符的大小写不会被保留，而且也可能需要加引号。

目录

clusterdb 1851
createdb 1854
createuser 1857
dropdb 1862
dropuser 1865
ecpg 1868
pg_basebackup 1871
pgbench 1878
pg_config 1896
pg_dump 1899
pg_dumpall 1911
pg_isready 1918
pg_receivewal 1920
pg_recvlogical 1924
pg_restore 1928
psql 1937
reindexdb 1977
vacuumdb 1980

clusterdb

clusterdb — 聚簇一个 PostgreSQL 数据库

大纲

```
clusterdb [connection-option...] [ --verbose | -v ] [ --table | -t table ] ...
[dbname]

clusterdb [connection-option...] [ --verbose | -v ] --all | -a
```

描述

clusterdb 是一个用于重新聚簇 PostgreSQL 数据库中各表的工具。它会找出此前已经聚簇过的表，并按照上次使用的同一索引再次对其进行聚簇。从未聚簇过的表不会受到影响。

clusterdb 是 SQL 命令 CLUSTER 的一个包装器。使用该工具聚簇数据库，与使用其他访问服务器的方法聚簇数据库，并没有实质区别。

选项

clusterdb 接受下列命令行参数：

`-a`
`--all`

聚簇所有数据库。

`[-d] dbname`
`[--dbname=]dbname`

当未使用 `-a`/`--all` 时，指定要聚簇的数据库名。如果未指定，则从环境变量 `PGDATABASE` 读取数据库名。如果该环境变量也未设置，则使用为该连接指定的用户名作为数据库名。*dbname* 可以是一个连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

`-e`
`--echo`

回显 clusterdb 生成并发送到服务器的命令。

`-q`
`--quiet`

不显示进度消息。

`-t table`
`--table=table`

仅对 *table* 执行聚簇。可以通过多次指定 `-t` 选项来聚簇多个表。

-v
--verbose

在处理过程中输出详细信息。

-V
--version

打印 clusterdb 的版本并退出。

-?
--help

显示有关 clusterdb 命令行参数的帮助信息，并退出。

clusterdb 还接受以下用于连接参数的命令行参数：

-h *host*
--host=*host*

指定服务器所在机器的主机名。如果该值以斜杠开头，则它会被用作 Unix 域套接字目录。

-p *port*
--port=*port*

指定服务器监听连接所用的 TCP 端口，或本地 Unix 域套接字文件扩展名。

-U *username*
--username=*username*

连接时使用的用户名。

-w
--no-password

绝不提示输入密码。如果服务器要求密码认证，而又无法通过其他方式获得密码，例如 .pgpass 文件，则连接尝试将失败。此选项可用于无人输入密码的批处理作业和脚本。

-W
--password

强制 clusterdb 在连接到数据库之前提示输入密码。

这个选项并非必不可少，因为如果服务器要求密码认证，clusterdb 会自动提示输入密码。不过，clusterdb 会浪费一次连接尝试，才知道服务器需要密码。在某些情况下，为了避免这次额外的连接尝试，提前指定-W 是值得的。

--maintenance-db=*dbname*

当使用 -a/--all 时，连接到该数据库以收集要聚簇的数据库列表。如果未指定，则使用 postgres 数据库；如果它不存在，则使用template1。这里也可以是一个连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。此外，在连接其他数据库时，除数据库名本身外的连接字符串参数也会被重用。

环境

PGDATABASE
PGHOST
PGPORT
PGUSER

默认连接参数

与大多数其他 PostgreSQL 工具一样，此工具也使用 libpq 支持的环境变量（见第 34.14 节）。

诊断

如果遇到问题，请参见CLUSTER 和 psql 中关于潜在问题和错误消息的讨论。数据库服务器必须在目标主机上运行。此外，libpq 前端库所使用的任何默认连接设置和环境变量都会生效。

示例

要聚簇数据库 `test`：

```
$ clusterdb test
```

要聚簇名为 `xyzzy` 的数据库中的单个表 `foo`：

```
$ clusterdb --table=foo xyzzy
```

另见

CLUSTER

createdb

createdb — 创建新的 PostgreSQL 数据库

大纲

`createdb` [*connection-option*...] [*option*...] [*dbname* [*description*]]

描述

createdb 创建一个新的 PostgreSQL 数据库。

通常，执行此命令的数据库用户会成为新数据库的所有者。不过，如果执行用户具有相应的权限，也可以通过`-O` 选项指定不同的所有者。

createdb 是 SQL 命令 CREATE DATABASE 的一个包装器。通过该工具创建数据库，与通过其他访问服务器的方法创建数据库在实际效果上没有区别。

选项

createdb 接受以下命令行参数：

dbname

指定要创建的数据库名称。该名称必须在此数据库集簇中的所有 PostgreSQL 数据库中唯一。默认情况下，会创建一个与当前系统用户同名的数据库。

description

指定要与新创建的数据库关联的注释。

`-D` *tablespace*
`--tablespace=`*tablespace*

指定数据库的默认表空间。（该名称将按双引号标识符处理。）

`-e`
`--echo`

回显 createdb 生成并发送到服务器的命令。

`-E` *encoding*
`--encoding=`*encoding*

指定该数据库使用的字符集编码。PostgreSQL 服务器支持的字符集见第 23.3.1 节。

`-l` *locale*
`--locale=`*locale*

指定此数据库要使用的区域设置。这相当于同时指定 `--lc-collate` 和 `--lc-ctype`。

`--lc-collate=`*locale*

指定该数据库要使用的 LC_COLLATE 设置。

`--lc-ctype=`*locale*

指定该数据库要使用的 LC_CTYPE 设置。

`-O` *owner*
`--owner=`*owner*

指定将成为新数据库拥有者的数据库用户。（该名称按双引号标识符处理。）

`-T` *template*
`--template=`*template*

指定用于构建该数据库的模板数据库。（该名称按双引号标识符处理。）

`-V`
`--version`

打印 createdb 的版本并退出。

`-?`
`--help`

显示有关 createdb 命令行参数的帮助信息，并退出。

选项 `-D`、`-l`、`-E`、`-O` 和 `-T` 对应于底层 SQL 命令 CREATE DATABASE 的选项；有关这些选项的更多信息请参阅该命令。

createdb 也接受下列与连接参数有关的命令行参数：

`-h` *host*
`--host=`*host*

指定服务器所在机器的主机名。如果该值以斜杠开头，则它将被用作 Unix 域套接字的目录。

`-p` *port*
`--port=`*port*

指定服务器监听连接所用的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U` *username*
`--username=`*username*

以该用户名进行连接。

`-w`
`--no-password`

从不发出密码提示。如果服务器要求密码认证，而密码又无法通过诸如 `.pgpass` 文件等其他方式获得，则连接尝试将失败。该选项适合没有用户在场输入密码的批处理作业和脚本。

-W
--password

强制 createdb 在连接到数据库之前提示输入密码。

这个选项并非必不可少，因为如果服务器要求密码认证，createdb 会自动提示输入密码。不过，createdb 会浪费一次连接尝试，才知道服务器需要密码。在某些情况下，为了避免这次额外的连接尝试，提前指定-W 是值得的。

--maintenance-db=*dbname*

指定在创建新数据库时要连接到的数据库名称。如果未指定，将使用postgres 数据库；若该数据库不存在（或者它正是要创建的新数据库），则改用template1。这也可以是一个连接字符串。如果是这种情况，其中的连接字符串参数将覆盖任何冲突的命令行选项。

环境

PGDATABASE

如果设置了该变量，其值就是要创建的数据库名称，除非被命令行覆盖。

PGHOST
PGPORT
PGUSER

默认连接参数。如果命令行和 PGDATABASE 都没有指定要创建的数据库名称，则PGUSER 也会决定该名称。

与大多数其他 PostgreSQL 工具一样，此工具也使用 libpq 支持的环境变量（见第 34.14 节）。

诊断

如果遇到问题，请参见CREATE DATABASE 和 psql 中关于潜在问题和错误消息的讨论。数据库服务器必须在目标主机上运行。此外，libpq 前端库所使用的任何默认连接设置和环境变量都会生效。

示例

要使用默认数据库服务器创建 demo 数据库：

```
$ createdb demo
```

要使用主机 eden 上、端口为 5000 的服务器，并以template0 作为模板数据库来创建 demo 数据库，可使用以下命令行命令及其底层 SQL 命令：

```
$ createdb -p 5000 -h eden -T template0 -e demo
CREATE DATABASE demo TEMPLATE template0;
```

另见

dropdb, CREATE DATABASE

createuser

createuser — 定义一个新的 PostgreSQL 用户账户

大纲

```
createuser [connection-option...] [option...] [username]
```

描述

createuser 创建一个新的 PostgreSQL 用户（更准确地说，是一个角色）。只有超级用户以及具有CREATEROLE 权限的用户才能创建新用户，因此调用createuser 的用户必须能够以超级用户或具有 CREATEROLE 权限的用户身份连接。

如果要创建一个具有 SUPERUSER、REPLICATION 或 BYPASSRLS 权限的角色，则必须以超级用户身份连接，而不能仅具有CREATEROLE 权限。超级用户意味着能够绕过数据库中的所有访问权限检查，因此不应轻易授予超级用户权限。CREATEROLE 也赋予非常广泛的权限。

createuser 是 SQL 命令 CREATE ROLE 的一个包装器。通过此工具创建用户与通过其他访问服务器的方法创建用户，在效果上没有区别。

选项

createuser 接受以下命令行参数：

username

指定要创建的 PostgreSQL 用户的名称。该名称必须不同于此 PostgreSQL 安装中所有现有角色的名称。

-c *number*
--connection-limit=*number*

为新用户设置最大连接数。默认不设限制。

-d
--createdb

允许新用户创建数据库。

-D
--no-createdb

不允许新用户创建数据库。这是默认设置。

-e
--echo

回显 createuser 生成并发送给服务器的命令。

-E
--encrypted

该选项已废弃，但为了向后兼容仍被接受。

-g *role*
--role=*role*

指定此角色应立即作为新成员加入的现有角色。可以使用多个 -g 开关，指定此角色要加入的多个角色。

-i
--inherit

新角色将自动继承其所属角色的权限。这是默认设置。

-I
--no-inherit

新角色将不会自动继承其所属角色的权限。

--interactive

如果命令行中未指定用户名，则提示输入用户名；此外，还会提示输入命令行中未指定的 -d/-D、-r/-R、-s/-S 选项。（直到 PostgreSQL 9.1 为止，这都是默认行为。）

-l
--login

允许新用户登录（也就是说，该用户名可用作初始会话用户标识符）。这是默认设置。

-L
--no-login

不允许新用户登录。（不具有登录权限的角色仍可用于管理数据库权限。）

-P
--pwprompt

如果给出此选项，createuser 会提示输入新用户的密码。如果不打算使用密码认证，则不需要这样做。

-r
--createrole

允许新用户创建、更改、删除其他角色，对其添加注释，更改其安全标签，以及授予或撤销其他角色的成员资格；也就是说，该用户将具有CREATEROLE 权限。关于该权限赋予哪些能力的更多细节，见创建角色。

-R
--no-createrole

不允许新用户创建新角色。这是默认设置。

`-s`
`--superuser`

新用户将成为超级用户。

`-S`
`--no-superuser`

新用户将不是超级用户。这是默认设置。

`-V`
`--version`

打印 createuser 版本并退出。

`--replication`

新用户将具有 `REPLICATION` 权限，关于该权限的更完整说明见 CREATE ROLE。

`--no-replication`

新用户将不具有 `REPLICATION` 权限，关于该权限的更完整说明见 CREATE ROLE。

`-?`
`--help`

显示有关 createuser 命令行参数的帮助并退出。

createuser 也接受下列用于连接参数的命令行参数：

`-h` *`host`*
`--host=`*`host`*

指定服务器所在主机的主机名。如果该值以斜杠开头，则将其用作 Unix 域套接字所在目录。

`-p` *`port`*
`--port=`*`port`*

指定服务器监听连接所用的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U` *`username`*
`--username=`*`username`*

指定连接时使用的用户名（不是要创建的用户名）。

`-w`
`--no-password`

绝不提示输入密码。如果服务器要求密码认证，而密码又无法通过诸如 `.pgpass` 文件等其他方式获得，则连接尝试将失败。该选项可用于没有用户在场输入密码的批处理作业和脚本。

`-W`
`--password`

强制 createuser 提示输入密码（用于连接到服务器，而不是新用户的密码）。

该选项绝非必需，因为如果服务器要求密码认证，createuser 会自动提示输入密码。不过，createuser 会浪费一次连接尝试来发现服务器需要密码。在某些情况下，输入-W 可以避免这次额外的连接尝试。

环境

PGHOST
PGPORT
PGUSER

默认连接参数

与大多数其他 PostgreSQL 工具一样，此工具也使用 libpq 支持的环境变量（见第 34.14 节）。

诊断

如果遇到问题，请参见CREATE ROLE 和 psql 中关于潜在问题和错误消息的讨论。数据库服务器必须在目标主机上运行。此外，libpq 前端库所使用的任何默认连接设置和环境变量都会生效。

示例

要在默认数据库服务器上创建用户 joe：

```
$ createuser joe
```

要在默认数据库服务器上创建用户 joe，并提示输入一些额外属性：

```
$ createuser --interactive joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

要通过主机 eden、端口 5000 上的服务器创建同一个用户joe，并显式指定各项属性，同时查看底层命令：

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

要把用户 joe 创建为超级用户，并立即为其设置密码：

```
$ createuser -P -s -e joe
Enter password for new role: xyzzy
Enter it again: xyzzy
CREATE ROLE joe PASSWORD 'SCRAM-SHA-256$4096:44560wPMLfjqiAzyPDZ/eQ==$4CA054rZlSFEq8Z3FEh
ToBTa2X6KnWFxFkPwIbKoDe0=:L/nbSZRCjp6RhOhKK56GoR1zibCCSePKshCJ9lnl3yw=' SUPERUSER CREATEDB
 CREATEROLE INHERIT LOGIN NOREPLICATION NOBYPASSRLS;
```

在上面的示例中，输入新密码时实际上不会回显它，但为清楚起见，这里展示了输入的内容。可以看到，该密码在发送给客户端之前就已经被加密了。

另见

dropuser, CREATE ROLE

dropdb

dropdb — 移除一个 PostgreSQL 数据库

大纲

dropdb [*connection-option*...] [*option*...] *dbname*

描述

dropdb 删除一个现有的 PostgreSQL 数据库。执行此命令的用户必须是数据库超级用户或该数据库的拥有者。

dropdb 是 SQL 命令 DROP DATABASE 的一个包装器。通过此工具删除数据库与通过其他访问服务器的方法删除数据库，在效果上没有区别。

选项

dropdb 接受以下命令行参数：

dbname

指定要移除的数据库名称。

-e
--echo

回显 dropdb 生成并发送给服务器的命令。

-i
--interactive

在执行任何破坏性操作之前发出确认提示。

-V
--version

打印 dropdb 版本并退出。

--if-exists

如果数据库不存在，则不要抛出错误。这种情况下会发出一个提示。

-?
--help

显示有关 dropdb 命令行参数的帮助并退出。

dropdb 也接受下列用于连接参数的命令行参数：

`-h` *host*
`--host=`*host*

指定服务器所在机器的主机名。如果该值以斜杠开头，则它将被用作 Unix 域套接字的目录。

`-p` *port*
`--port=`*port*

指定服务器监听连接所用的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U` *username*
`--username=`*username*

指定用于连接的用户名。

`-w`
`--no-password`

从不发出密码提示。如果服务器要求密码认证，而密码又无法通过诸如 `.pgpass` 文件等其他方式获得，则连接尝试将失败。该选项适合没有用户在场输入密码的批处理作业和脚本。

`-W`
`--password`

强制 dropdb 在连接数据库之前提示输入密码。

这个选项并非必不可少，因为如果服务器要求密码认证，dropdb 会自动提示输入密码。不过，dropdb 会浪费一次连接尝试，才知道服务器需要密码。在某些情况下，为了避免这次额外的连接尝试，提前指定`-W` 是值得的。

`--maintenance-db=`*dbname*

指定在删除目标数据库时要连接到的数据库名称。如果未指定，将使用 `postgres` 数据库；若该数据库不存在（或者它正是要被删除的数据库），则改用`template1`。这也可以是一个连接字符串。如果是这种情况，其中的连接字符串参数将覆盖任何冲突的命令行选项。

环境

`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

与大多数其他 PostgreSQL 工具一样，此工具也使用 libpq 支持的环境变量（见第 34.14 节）。

诊断

如果遇到问题，请参见DROP DATABASE 和 psql 中关于潜在问题和错误消息的讨论。数据库服务器必须在目标主机上运行。此外，libpq 前端库所使用的任何默认连接设置和环境变量都会生效。

示例

要在默认数据库服务器上删除数据库 `demo`：

```
$ dropdb demo
```

要使用主机 `eden`、端口 5000 上的服务器删除数据库`demo`，并带确认提示以及查看底层命令：

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE demo;
```

另见

createdb, DROP DATABASE

dropuser

dropuser — 移除一个 PostgreSQL 用户账户

大纲

`dropuser` [*connection-option*...] [*option*...] [*username*]

描述

dropuser 移除现有的 PostgreSQL 用户。只有超级用户和拥有 `CREATEROLE` 权限的用户才能移除 PostgreSQL 用户。（要移除超级用户，你自己也必须是超级用户。）

dropuser 是 SQL 命令 DROP ROLE 的包装器。通过此工具删除用户与通过其他访问服务器的方法删除用户，在效果上并无区别。

选项

dropuser 接受下列命令行参数：

username

指定要移除的 PostgreSQL 用户名。如果命令行中未指定名称，且使用了 `-i`/`--interactive` 选项，则会提示输入名称。

`-e`
`--echo`

回显 dropuser 生成并发送给服务器的命令。

`-i`
`--interactive`

在实际移除该用户之前提示确认；如果命令行中未指定用户名，也会提示输入用户名。

`-V`
`--version`

打印 dropuser 版本并退出。

`--if-exists`

如果该用户不存在，则不要抛出错误。这种情况下会发出一个提示。

`-?`
`--help`

显示有关 dropuser 命令行参数的帮助并退出。

dropuser 也接受下列用于连接参数的命令行参数：

`-h` *host*
`--host=`*host*

指定服务器运行所在主机的主机名。如果该值以斜杠开头，则它将被用作 Unix 域套接字所在目录。

`-p` *port*
`--port=`*port*

指定服务器监听连接所用的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U` *username*
`--username=`*username*

用于连接的用户名（不是要删除的用户名）。

`-w`
`--no-password`

从不发出密码提示。如果服务器要求密码认证，而密码又无法通过诸如 `.pgpass` 文件之类的其他方式获得，则连接尝试将失败。该选项适合没有用户在场输入密码的批处理作业和脚本。

`-W`
`--password`

强制 dropuser 在连接到数据库之前提示输入密码。

这个选项绝非必需，因为如果服务器要求密码认证，dropuser 会自动提示输入密码。不过，dropuser 会浪费一次连接尝试来发现服务器需要密码。在某些情况下，输入`-W` 值得一试，以避免这次额外的连接尝试。

环境

`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

与大多数其他 PostgreSQL 工具一样，此工具也使用 libpq 支持的环境变量（见第 34.14 节）。

诊断

如果遇到问题，请参见DROP ROLE 和 psql 中关于潜在问题和错误消息的讨论。数据库服务器必须在目标主机上运行。此外，libpq 前端库所使用的任何默认连接设置和环境变量都会生效。

示例

要从默认数据库服务器上移除用户 joe：

```
$ dropuser joe
```

要使用主机 eden、端口 5000 上的服务器移除用户joe，并带确认提示以及查看底层命令：

```
$ dropuser -p 5000 -h eden -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE joe;
```

另见

createuser, DROP ROLE

ecpg

ecpg — 嵌入式 SQL C 预处理器

大纲

```
ecpg [option...] file...
```

描述

ecpg 是供 C 程序使用的嵌入式 SQL 预处理器。它会把嵌入了 SQL 语句的 C 程序转换成普通 C 代码，方法是将 SQL 调用替换为特殊的函数调用。生成的输出文件随后可以由任何 C 编译器工具链处理。

ecpg 会将命令行中给出的每个输入文件转换成对应的 C 输出文件。若输入文件名没有任何扩展名，则假定其扩展名为 .pgc。随后会把该文件的扩展名替换为 .c，以构造输出文件名。不过，也可以用 -o 选项覆盖输出文件名。

如果输入文件名只是 -，ecpg 就会从标准输入读取程序（并写到标准输出，除非使用 -o 覆盖该行为）。

本参考页不描述嵌入式 SQL 语言。有关该主题的更多信息，见第 36 章。

选项

ecpg 接受以下命令行参数：

-c

自动从 SQL 代码生成某些 C 代码。目前，这适用于 EXEC SQL TYPE。

-C *mode*

设置兼容模式。*mode* 可以是 INFORMIX、INFORMIX_SE，或 ORACLE。

-D *symbol*

定义一个 C 预处理器符号。

-h

处理头文件。指定该选项时，输出文件扩展名变为 .h 而不是 .c，默认输入文件扩展名也变为 .pgh 而不是 .pgc。此外，还会强制启用 -c 选项。

-i

也解析系统包含文件。

`-I` *directory*

指定一个附加的包含文件搜索路径，用于查找通过`EXEC SQL INCLUDE` 包含的文件。默认路径依次为.（当前目录）、`/usr/local/include`、编译时定义的PostgreSQL 包含文件目录（默认：`/usr/local/pgsql/include`），以及`/usr/include`。

`-o` *filename*

指定 `ecpg` 应将全部输出写入给定的 *filename*。写成`-o -`可将全部输出发送到标准输出。

`-r` *option*

选择运行时行为。*option* 可以是下列之一：

`no_indicator`

不使用指示符，而改用特殊值表示空值。历史上曾有数据库采用这种方式。

`prepare`

在使用前预备所有语句。libecpg 会维护一个已预备语句的缓存；如果某条语句再次执行，就会复用它。若缓存已满，libecpg 会释放使用次数最少的语句。

`questionmarks`

出于兼容性原因，允许使用问号作为占位符。这在很久以前曾是默认行为。

`-t`

打开事务自动提交。在此模式下，除非 SQL 命令位于显式事务块中，否则每条 SQL 命令都会自动提交。在默认模式下，只有发出`EXEC SQL COMMIT` 时才会提交命令。

`-v`

打印附加信息，包括版本和包含文件搜索路径。

`--version`

打印 ecpg 版本并退出。

`-?`
`--help`

显示关于 ecpg 命令行参数的帮助信息，并退出。

注解

在编译预处理后的 C 代码文件时，编译器必须能够在PostgreSQL 的包含文件目录中找到 ECPG 头文件。因此，调用编译器时你可能需要使用`-I` 选项（例如`-I/usr/local/pgsql/include`）。

使用嵌入式 SQL 的 C 程序必须与 `libecpg` 库链接，例如可使用链接器选项`-L/usr/local/pgsql/lib -lecpg`。

当前安装使用的这两个目录可通过 pg_config 查询。

示例

如果你有一个名为 `prog1.pgc` 的嵌入式 SQL C 源文件，就可以使用下列命令序列创建一个可执行程序：

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_basebackup

pg_basebackup — 获取 PostgreSQL 集簇的基础备份

大纲

`pg_basebackup` [*option*...]

描述

pg_basebackup 用于获取正在运行的 PostgreSQL 数据库集簇的基础备份。备份过程不会影响数据库的其他客户端，并且该备份既可用于时间点恢复（见第 25.3 节），也可用作日志传送或流复制备库的起点（见第 26.2 节）。

pg_basebackup 创建数据库集簇文件的二进制副本，同时确保系统自动进入和退出备份模式。备份始终针对整个数据库集簇；无法只备份单个数据库或数据库对象。要备份单个数据库，必须使用pg_dump 之类的工具。

备份通过使用复制协议的普通 PostgreSQL 连接进行。建立连接时必须使用超级用户或具有 `REPLICATION` 权限（参见第 21.2 节）的用户，并且`pg_hba.conf` 必须明确允许复制连接。服务器还必须将 max_wal_senders 设置得足够高，以便至少为备份保留一个可用会话，并在使用 WAL 流式传输时再为其保留一个会话。

可以同时运行多个 `pg_basebackup`，但从性能角度来看，最好只执行一次备份，然后复制其结果。

pg_basebackup 不仅可以从主库获取基础备份，也可以从备库获取。要从备库获取备份，需要将备库配置为能够接受复制连接（即设置`max_wal_senders` 和 hot_standby，并配置基于主机的认证）。还需要在主库上启用 full_page_writes。

注意，从备库获取在线备份时有一些限制：

- 备份历史文件不会在被备份的数据库集簇中创建。
- pg_basebackup 无法强制备库在备份结束时切换到新的 WAL 文件。当使用`-X none` 时，如果主库上的写入活动较少，pg_basebackup 可能需要长时间等待，直到备份所需的最后一个 WAL 文件被切换并归档。在这种情况下，可以在主库上运行`pg_switch_wal`，以立即触发 WAL 文件切换。
- 如果备库在在线备份过程中被提升为主库，则备份会失败。
- 备份所需的所有 WAL 记录都必须包含足够的整页写入，因此必须在主库上启用 `full_page_writes`，并且不能把pg_compresslog 之类的工具用作 `archive_command`，从 WAL 文件中移除整页写入。

选项

以下命令行选项控制输出的位置和格式。

-D *directory*
--pgdata=*directory*

设置写入输出的目标目录。如果该目录不存在，pg_basebackup 会创建它（以及所有缺失的父目录）。如果该目录已经存在，则必须为空。

当备份采用 tar 格式时，目标目录可以指定为-（短横线），从而将 tar 文件写到stdout。

此选项是必需的。

-F *format*
--format=*format*

选择输出格式。*format* 可以是以下值之一：

p
plain

将输出写为普通文件，其布局与源服务器的数据目录和表空间相同。当集簇没有额外表空间时，整个数据库都会放在目标目录中。如果集簇包含额外表空间，则主数据目录会放在目标目录中，而其他所有表空间都会放在与源服务器上相同的绝对路径中。（如需改变这一点，见--tablespace-mapping。）

这是默认格式。

t
tar

将输出写为目标目录中的 tar 文件。主数据目录的内容会写入名为 base.tar 的文件中，而每个其他表空间都会写入一个以该表空间 OID 命名的独立 tar 文件中。

如果目标目录指定为-（短横线），tar 内容将写入标准输出，适合通过管道传给例如 gzip。只有当集簇没有额外表空间且未使用 WAL 流式传输时，才允许这样做。

-r *rate*
--max-rate=*rate*

从服务器传输数据的最大速率。单位为每秒千字节。使用后缀 M 表示每秒兆字节。也接受后缀 k，但它没有效果。有效值介于每秒 32 千字节和每秒 1024 兆字节之间。

目的是限制 pg_basebackup 对运行中的服务器的影响。

此选项始终影响数据目录的传输。只有当收集方法为 fetch 时，WAL 文件的传输才会受到影响。

-R
--write-recovery-conf

在输出目录中（使用 tar 格式时则在基础归档文件中）写入一个最小的 recovery.conf，以便设置备库。recovery.conf 文件会记录 pg_basebackup 所用的连接设置，以及复制槽（如果指定了），以便流复制以后使用相同的设置。

`-T` *olddir*`=`*newdir*
`--tablespace-mapping=`*olddir*`=`*newdir*

在备份期间，将目录*olddir* 中的表空间重定位到 *newdir*。要使此选项生效，*olddir* 必须与该表空间当前定义的路径完全一致。（但如果备份中没有位于 *olddir* 中的表空间，也不算错误。）*olddir* 和 *newdir* 都必须是绝对路径。如果路径中包含=符号，请用反斜线转义。可以多次指定此选项，以处理多个表空间。参见下文示例。

如果以这种方式重定位表空间，主数据目录中的符号链接将被更新为指向新位置。因此，新数据目录已可直接用于启动一个所有表空间都位于更新后位置的新服务器实例。

目前，此选项仅适用于普通文件输出格式；如果选择了 tar 格式，则会被忽略。

`--waldir=`*waldir*

指定预写式日志目录的位置。*waldir* 必须是绝对路径。只有在备份采用普通文件模式时，才能指定预写式日志目录。

`-X` *method*
`--wal-method=`*method*

在备份中包含所需的 WAL（预写式日志）文件。这将包括备份期间生成的所有预写式日志。除非指定方法 `none`，否则就可以在解包后的目录中启动 postmaster，而无需查阅 WAL 归档，从而使输出成为一个完全独立的备份。

支持以下收集预写式日志的方法：

`n`
`none`

不在备份中包含预写式日志。

`f`
`fetch`

在备份结束时收集预写式日志文件。因此，wal_keep_segments 参数必须设置得足够高，以确保在备份结束前不会移除所需的日志数据。如果在传输这些数据之前它们已经被回收，则备份会失败并且无法使用。

当使用 tar 格式时，预写式日志文件会包含在`base.tar` 文件中。

`s`
`stream`

在创建备份的同时流式传输预写式日志。这会打开与服务器的第二个连接，并在执行备份时并行开始流式传输预写式日志。因此，它会占用由max_wal_senders 参数配置的两个连接。只要客户端能够跟上接收到的预写式日志，使用此模式就不需要在主库上额外保留预写式日志。

当使用 tar 格式时，预写式日志文件会写入一个名为`pg_wal.tar` 的独立文件中（如果服务器版本早于 10，则文件名为`pg_xlog.tar`）。

这是默认值。

-z
--gzip

启用 tar 文件输出的 gzip 压缩，使用默认压缩级别。压缩仅在使用 tar 格式时可用，并且所有 tar 文件名都会自动添加 .gz 后缀。

-Z *level*
--compress=*level*

启用 tar 文件输出的 gzip 压缩，并指定压缩级别（0 到 9，其中 0 表示不压缩，9 表示最佳压缩）。压缩仅在使用 tar 格式时可用，并且所有 tar 文件名都会自动添加 .gz 后缀。

以下命令行选项控制备份的生成和程序的运行。

-c *fast|spread*
--checkpoint=*fast|spread*

将检查点模式设置为 fast（立即）或 spread（默认）（见第 25.3.3 节）。

-C
--create-slot

指定在开始备份之前，应创建由--slot 选项指定名称的复制槽。如果该槽已经存在，则会报错。

-l *label*
--label=*label*

设置备份标签。如果未指定，则使用默认值“pg_basebackup base backup”。

-n
--no-clean

默认情况下，当pg_basebackup 因错误中止时，它会删除在发现自己无法完成任务之前可能已创建的所有目录（例如数据目录和预写式日志目录）。此选项会禁止这种清理，因此对调试很有帮助。

请注意，无论是否使用此选项，表空间目录都不会被清理。

-N
--no-sync

默认情况下，pg_basebackup 会等待所有文件都被安全写入磁盘。此选项会使 pg_basebackup 不经等待就直接返回，因此速度更快，但这意味着如果随后操作系统崩溃，基础备份可能会损坏。通常，此选项适合测试，但不应用于创建生产环境安装。

-P
--progress

启用进度报告。打开此选项后，会在备份过程中给出一个近似的进度报告。由于数据库在备份过程中可能发生变化，因此这只是近似值，最终未必恰好结束在100%。特别是当备份中包含 WAL 时，总数据量无法预先估计；在这种情况下，一旦进度超过不含 WAL 时的总估计值，估计目标大小就会继续增加。

启用此选项后，备份会先遍历整个数据库以统计其大小，然后再返回发送实际内容。这可能使备份稍微耗时更长，尤其会使首次发送数据之前的等待时间更长。

-S *slotname*
--slot=*slotname*

此选项只能与-X stream 一起使用。它会使 WAL 流式传输使用指定的复制槽。如果此基础备份打算用作使用复制槽的流复制备库，那么该备库应在recovery.conf 中使用同一个复制槽名称。这样可以确保服务器在基础备份结束与开始流复制之间的这段时间内，不会移除任何必需的 WAL 数据。

指定的复制槽必须已经存在，除非还同时使用了-C。

如果未指定此选项，并且服务器支持临时复制槽（10 及以上版本），则会自动使用临时复制槽进行 WAL 流式传输。

-v
--verbose

启用详细模式。它会在启动和关闭过程中输出一些额外步骤；如果同时启用了进度报告，还会显示当前正在处理的确切文件名。

--no-slot

即使服务器支持，此选项也会阻止在备份期间创建临时复制槽。

默认情况下，如果选择了 WAL 流式传输，但没有使用-S 选项给出槽名，那么会创建一个临时复制槽。

此选项的主要目的是在服务器没有空闲复制槽时，仍然允许执行基础备份。几乎总是首选使用复制槽，因为它能防止服务器在备份期间移除所需的 WAL。

--no-verify-checksums

如果获取基础备份的服务器启用了校验和，则禁用对这些校验和的验证。

默认情况下，会验证校验和；校验和验证失败将导致非零退出状态。不过，在这种情况下，基础备份不会被删除，就像使用了--no-clean 一样。

以下命令行选项控制数据库连接参数。

-d *connstr*
--dbname=*connstr*

以连接字符串的形式指定用于连接服务器的参数；这些参数会覆盖任何相互冲突的命令行选项。

出于与其他客户端应用保持一致的考虑，此选项名为--dbname；但由于 pg_basebackup 并不连接到集簇中的某个特定数据库，连接字符串中的数据库名会被忽略。

-h *host*
--host=*host*

指定服务器运行所在机器的主机名。如果该值以斜线开头，则它会被用作 Unix 域套接字的目录。默认值取自PGHOST 环境变量（如果已设置）；否则会尝试使用 Unix 域套接字连接。

`-p port`
`--port=port`

指定服务器监听连接所使用的 TCP 端口，或本地 Unix 域套接字文件扩展名。默认使用 `PGPORT` 环境变量中的值（如果已设置），否则使用编译时确定的默认值。

`-s interval`
`--status-interval=interval`

指定向服务器回送状态包的时间间隔（秒）。这样更便于从服务器端监视进度。值为零将完全禁用周期性状态更新，不过在服务器请求时仍会发送更新，以避免因超时而断开连接。默认值是 10 秒。

`-U username`
`--username=username`

指定连接时使用的用户名。

`-w`
`--no-password`

禁止发出密码提示。如果服务器要求密码认证，而又无法通过其他方式（例如 `.pgpass` 文件）获得密码，则连接尝试将失败。此选项对于批处理作业和脚本很有用，因为那种场景下通常没有用户在场输入密码。

`-W`
`--password`

强制 pg_basebackup 在连接数据库之前提示输入密码。

此选项绝非必需，因为如果服务器要求密码认证，pg_basebackup 会自动提示输入密码。不过，pg_basebackup 会浪费一次连接尝试来发现服务器需要密码。在某些情况下，输入 `-W` 以避免额外的连接尝试是值得的。

其他选项也可用：

`-V`
`--version`

输出 pg_basebackup 的版本并退出。

`-?`
`--help`

显示 pg_basebackup 命令行参数的帮助并退出。

环境

与大多数其他 PostgreSQL 工具一样，此工具也使用libpq 支持的环境变量（见第 34.14 节）。

注解

在备份开始时，需要在源服务器上执行一次检查点。这可能需要一些时间（尤其是在未使用 `--checkpoint=fast` 选项时）；在此期间，pg_basebackup 看起来会处于空闲状态。

备份将包括数据目录和表空间中的所有文件，包括配置文件以及第三方放在这些目录中的任何额外文件，但由 PostgreSQL 管理的某些临时文件除外。不过，只有普通文件和目录会被复制，但用于表空间的符号链接会被保留。指向 PostgreSQL 已知的某些目录的符号链接会被复制为空目录。其他符号链接和特殊设备文件则会被跳过。具体细节见第 53.4 节。

在普通文件格式中，除非使用了`--tablespace-mapping` 选项，否则表空间会备份到其在源服务器上的相同路径。如果不使用此选项，那么当表空间正在使用时，就无法在与服务器相同的主机上执行普通文件格式的基础备份，因为备份将不得不写入与原始表空间相同的目录位置。

使用 tar 格式时，用户有责任在启动 PostgreSQL 服务器之前解包每个 tar 文件。如果存在额外表空间，则其 tar 文件必须解包到正确的位置。在这种情况下，服务器会根据`tablespace_map` 文件的内容，为这些表空间创建符号链接，该文件包含在`base.tar` 文件中。

pg_basebackup 可以与相同主版本或更低主版本的服务器配合工作，最低支持到 9.1。不过，WAL 流式传输模式（`-X stream`）仅适用于 9.3 及以上版本服务器，当前版本的 tar 格式（`--format=tar`）仅适用于 9.5 及以上版本服务器。

如果源集簇启用了组权限，pg_basebackup 会在 `plain` 和 `tar` 两种格式中保留组权限。

示例

要为服务器 `mydbserver` 创建一个基础备份，并将其存储到本地目录 `/usr/local/pgsql/data` 中：

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

要为本地服务器创建一个备份，为每个表空间各生成一个压缩的 tar 文件，并将其存储在目录 `backup` 中，同时在运行期间显示进度报告：

```
$ pg_basebackup -D backup -Ft -z -P
```

要为一个仅包含单个表空间的本地数据库创建备份，并使用bzip2 进行压缩：

```
$ pg_basebackup -D - -Ft -X fetch | bzip2 > backup.tar.bz2
```

（如果该数据库中有多个表空间，此命令将失败。）

要创建一个本地数据库的备份，并将位于`/opt/ts` 中的表空间重定位到`./backup/ts`：

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

另见

pg_dump

pgbench

pgbench — 运行 PostgreSQL 基准测试

大纲

`pgbench -i` [*option*...] [*dbname*]

`pgbench` [*option*...] [*dbname*]

描述

pgbench 是一个用于对 PostgreSQL 执行基准测试的简单程序。它会反复执行同一组 SQL 命令，也可在多个并发数据库会话中运行，然后计算平均事务速率（每秒事务数）。默认情况下，pgbench 测试的是一个大体上基于 TPC-B 的场景，每个事务包含五条`SELECT`、`UPDATE` 和 `INSERT` 命令。不过，通过编写自己的事务脚本文件，也很容易测试其他场景。

下面是 pgbench 的典型输出：

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 85.184871 (including connections establishing)
tps = 85.296346 (excluding connections establishing)
```

前六行报告了一些最重要的参数设置。下一行报告已完成的事务数和预期的事务数（后者就是客户端数与每个客户端的事务数的乘积）；除非运行在完成前失败，否则这两个数应该相等。（在 `-T` 模式下，只打印实际的事务数。）最后两行报告每秒事务数，分别计入和不计入启动数据库会话的时间。

默认的类 TPC-B 事务测试要求预先建立特定的表。应使用 `-i`（初始化）选项调用 pgbench 来创建并填充这些表。（测试自定义脚本时不需要这一步，但需要自行完成测试所需的准备工作。）初始化命令如下：

```
pgbench -i [ other-options ] dbname
```

其中 *dbname* 是已创建好的、用于执行测试的数据库名称。（可能还需要使用`-h`、`-p` 和/或 `-U` 选项来指定如何连接到数据库服务器。）

> **小心**
>
> `pgbench -i` 会创建四个表 `pgbench_accounts`、`pgbench_branches`、`pgbench_history` 和 `pgbench_tellers`，并销毁任何已存在的同名表。如果数据库中已经存在这些名称的表，请务必改用其他数据库！

在默认的“比例因子”1 下，这些表最初包含如下行数：

```
table                    # of rows
---------------------------------
pgbench_branches         1
pgbench_tellers          10
pgbench_accounts         100000
pgbench_history          0
```

可以使用 `-s`（比例因子）选项来增加行数，而且在大多数场景下通常也应该这样做。此时还可以配合使用`-F`（fillfactor）选项。

完成必要的准备后，就可以使用不带`-i` 的命令运行基准测试，也就是：

```
pgbench [ options ] dbname
```

几乎在所有情况下，都需要附加一些选项才能得到有意义的测试。最重要的选项是`-c`（客户端数）、`-t`（事务数）、`-T`（时间限制）以及 `-f`（指定自定义脚本文件）。完整列表见下文。

选项

以下内容分为三个小节。数据库初始化和运行基准测试时使用不同的选项，但有些选项在这两种情况下都适用。

初始化选项

pgbench 接受以下命令行初始化参数：

`-i`
`--initialize`

进入初始化模式所必需。

`-I` *`init_steps`*
`--init-steps=`*`init_steps`*

仅执行选定的常规初始化步骤。*`init_steps`* 指定要执行的初始化步骤，每个步骤用一个字符表示。各步骤按照指定顺序调用。默认值为 `dtgvp`。可用步骤如下：

`d`（删除）

删除任何现有的 pgbench 表。

t（创建表）

创建标准 pgbench 场景使用的表，即 pgbench_accounts、pgbench_branches、pgbench_history 和 pgbench_tellers。

g（生成数据）

生成数据并将其装载到标准表中，替换其中任何已有数据。

v（清理）

在标准表上调用 VACUUM。

p（创建主键）

在标准表上创建主键索引。

f（创建外键）

在标准表之间创建外键约束。（请注意，默认情况下不执行此步骤。）

-F *fillfactor*
--fillfactor=*fillfactor*

以给定的 fillfactor 创建 pgbench_accounts、pgbench_tellers 和 pgbench_branches 表。默认值为 100。

-n
--no-vacuum

初始化期间不执行任何清理。（即使在 -I 中指定了 v 步骤，此选项也会抑制它。）

-q
--quiet

将日志切换为安静模式，每 5 秒只输出一条进度消息。默认日志每 100000 行打印一条消息，因此通常每秒输出很多行（尤其是在性能良好的硬件上）。

-s *scale_factor*
--scale=*scale_factor*

将生成的行数乘以比例因子。例如，-s 100 会在 pgbench_accounts 表中创建 10,000,000 行。默认值为 1。当比例达到 20,000 或更大时，用于保存账户标识符的列（aid 列）将切换到使用更大的整数（bigint），以便容纳账户标识符的取值范围。

--foreign-keys

在标准表之间创建外键约束。（如果初始化步骤序列中尚未包含 f 步骤，则此选项会把它加入进去。）

--index-tablespace=*index_tablespace*

在指定的表空间中创建索引，而不是默认的表空间。

--tablespace=*tablespace*

在指定的表空间中创建表，而不是默认的表空间。

`--unlogged-tables`

将所有表创建为不记录 WAL 的表，而不是永久表。

基准测试选项

pgbench 接受以下命令行基准测试参数：

`-b` *`scriptname[@weight]`*
`--builtin=`*`scriptname[@weight]`*

将指定的内置脚本添加到待执行脚本列表中。可用的内置脚本包括：`tpcb-like`、`simple-update` 和 `select-only`。也接受内置名称的无歧义前缀。使用特殊名称 `list` 时，会显示内置脚本列表并立即退出。

可选地，可在`@`后写一个整数权重，以调整此脚本相对于其他脚本的选中概率。默认权重为 1。详情请参见下文。

`-c` *`clients`*
`--client=`*`clients`*

模拟的客户端数量，也就是并发数据库会话的数量。默认值为 1。

`-C`
`--connect`

为每个事务建立一个新连接，而不是仅在每个客户端会话中执行一次。这对于测量连接开销很有用。

`-d`
`--debug`

打印调试输出。

`-D` *`varname=value`*
`--define=`*`varname=value`*

定义一个变量，供自定义脚本使用（见下文）。允许使用多个`-D` 选项。

`-f` *`filename[@weight]`*
`--file=`*`filename[@weight]`*

将从 *`filename`* 读取的事务脚本添加到要执行的脚本列表中。

可选地，可在`@`后写一个整数权重，以调整此脚本相对于其他脚本的选中概率。默认权重为 1。（如果脚本文件名本身包含`@`字符，可追加一个权重以消除歧义，例如`filen@me@1`。）详情见下文。

`-j` *`threads`*
`--jobs=`*`threads`*

pgbench 中的工作线程数。在多 CPU 机器上使用多个线程可能会有所帮助。客户端尽可能均匀地分布在可用线程中。默认值为 1。

`-l`
`--log`

将每个事务的信息写入日志文件。详情见下文。

`-L` *`limit`*
`--latency-limit=`*`limit`*

持续时间超过 *`limit`* 毫秒的事务会被单独计数和报告，称为late。

使用限流（`--rate=...`）时，若某个事务落后于计划时间超过*`limit`* 毫秒，从而已经不可能满足延迟限制，则它根本不会被发送到服务器。此类事务会被单独计数并报告为skipped。

`-M` *`querymode`*
`--protocol=`*`querymode`*

用于向服务器提交查询的协议：

- `simple`：使用简单查询协议。
- `extended`：使用扩展查询协议。
- `prepared`：使用带有预备语句的扩展查询协议。

默认为简单查询协议。（详见第 53 章。）

`-n`
`--no-vacuum`

在运行测试前不执行任何清理。如果运行的是不包含标准表 `pgbench_accounts`、`pgbench_branches`、`pgbench_history` 和 `pgbench_tellers` 的自定义测试场景，则此选项是必需的。

`-N`
`--skip-some-updates`

运行内置的 simple-update 脚本。是 `-b simple-update` 的简写。

`-P` *`sec`*
`--progress=`*`sec`*

每 *`sec`* 秒显示一次进度报告。报告包括自运行开始以来的时间、自上次报告以来的 TPS，以及自上次报告以来事务延迟的平均值和标准差。使用限流（`-R`）时，延迟是相对于事务计划开始时间计算的，而不是实际开始时间，因此其中也包含平均计划滞后时间。

`-r`
`--report-latencies`

在基准测试完成后，报告每条命令的平均语句延迟（从客户端视角看到的执行时间）。详情见下文。

`-R` *`rate`*
`--rate=`*`rate`*

以指定速率执行事务，而不是像默认行为那样尽可能快地运行。速率以每秒事务数表示。如果目标速率高于可达到的最大速率，则速率限制不会影响结果。

该速率通过让事务沿着一条符合泊松分布的时间线启动来实现。预期开始时间表是根据客户端首次启动的时间向前推进的，而不是根据前一个事务结束的时间。这意味着，当某些事务超过其原定结束时间时，后续事务仍有可能重新赶上计划。

启用限流后，运行结束时报告的事务延迟是从计划开始时间计算的，因此它包含每个事务等待前一个事务完成的时间。这段等待时间称为计划滞后时间，其平均值和最大值也会单独报告。若要得到相对于事务实际开始时间的延迟，也就是事务在数据库中实际执行所花费的时间，可以用报告中的延迟减去计划滞后时间。

如果同时使用 `--latency-limit` 和 `--rate`，一个事务可能会落后太多，以至于在前一个事务结束时已经超过了延迟限制，因为延迟是从计划开始时间计算的。这样的事务不会发送到服务器，而是完全跳过并单独计数。

较高的计划滞后时间表明，在所选客户端数和线程数下，系统无法以指定速率处理事务。当平均事务执行时间长于事务之间的计划间隔时，后续事务会不断进一步落后，而计划滞后时间也会随着测试持续时间增加。在这种情况下，需要降低指定的事务速率。

`-s` *`scale_factor`*
`--scale=`*`scale_factor`*

在 pgbench 输出中报告指定的比例因子。对于内置测试，这没有必要；系统会通过统计 `pgbench_branches` 表中的行数来检测正确的比例因子。但在只测试自定义基准（`-f` 选项）时，除非使用此选项，否则比例因子会被报告为 1。

`-S`
`--select-only`

运行内置的 select-only 脚本。是 `-b select-only` 的简写。

`-t` *`transactions`*
`--transactions=`*`transactions`*

每个客户端运行的事务数量。默认值为 10。

`-T` *`seconds`*
`--time=`*`seconds`*

让测试运行指定的秒数，而不是让每个客户端执行固定数量的事务。`-t` 和 `-T` 是互斥的。

`-v`
`--vacuum-all`

在运行测试之前，对四个标准表全部执行清理。如果既不使用`-n` 也不使用 `-v`，pgbench 会对 `pgbench_tellers` 和 `pgbench_branches` 表进行清理，并截断 `pgbench_history`。

`--aggregate-interval=`*`seconds`*

聚合间隔的长度（以秒为单位）。只能与 `-l` 选项一起使用。使用此选项时，日志包含每个间隔的摘要数据，如下所述。

`--log-prefix=`*`prefix`*

设置由 `--log` 创建的日志文件的文件名前缀。默认值为 `pgbench_log`。

`--progress-timestamp`

显示进度（选项 `-P`）时，使用时间戳（Unix 纪元）而不是自运行开始以来的秒数。单位为秒，小数点后精确到毫秒。这有助于比较各种工具生成的日志。

`--random-seed=`*`SEED`*

设置随机数生成器种子。它会先为系统随机数生成器设种，再生成一系列初始生成器状态，每个线程一个。*`SEED`* 的取值可以是：`time`（默认值，种子基于当前时间），`rand`（使用强随机源，如果没有可用的则失败），或者无符号十进制整数值。随机生成器既可以在 pgbench 脚本中显式调用（`random...`函数），也可以隐式调用（例如选项 `--rate` 用于调度事务）。显式设置时，实际用于设种的值会显示在终端上。任何允许的*`SEED`* 值也可以通过环境变量 `PGBENCH_RANDOM_SEED` 提供。为确保提供的种子影响所有可能的用途，将此选项放在第一位或使用环境变量。

显式设置种子可以在随机数层面精确复现一次 `pgbench` 运行。由于随机状态按线程管理，这意味着对于相同的调用，如果每个线程只有一个客户端，且不存在外部依赖或数据依赖，那么`pgbench` 运行可以完全一致。从统计角度看，精确复现实验并不是好主意，因为它可能掩盖性能波动，甚至不恰当地提高性能，例如反复命中与前一次运行相同的页面。不过，这对调试也可能非常有帮助，例如重现会触发错误的棘手案例。请谨慎使用。

`--sampling-rate=`*`rate`*

写入日志时使用的采样率，用于减少生成的日志量。如果给定此选项，则只记录指定比例的事务。1.0 表示记录全部事务，0.05 表示只记录 5% 的事务。

处理日志文件时，记得把采样率考虑进去。例如，计算 TPS 值时，需要按采样率对数字进行相应换算（例如采样率为 0.01 时，只能得到实际 TPS 的 1/100）。

公共选项

pgbench 接受以下通用命令行参数：

`-h` *`hostname`*
`--host=`*`hostname`*

数据库服务器的主机名。

`-p` *`port`*
`--port=`*`port`*

数据库服务器的端口号。

`-U` *`login`*
`--username=`*`login`*

连接时使用的用户名。

`-V`
`--version`

打印 pgbench 版本并退出。

-?
--help

显示 pgbench 命令行参数的帮助信息并退出。

注解

在 pgbench 中实际执行的“事务”是什么?

pgbench 会从指定列表中随机选取测试脚本来执行。这些脚本既可以是用 -b 指定的内置脚本，也可以是用-f 指定的用户脚本。每个脚本都可以在其后加上一个以@引出的相对权重，以改变其被选中的概率。默认权重为1。权重为 0 的脚本会被忽略。

默认的内置事务脚本（也可通过 -b tpcb-like 调用）会针对随机选取的 aid、tid、bid 和 delta 在每个事务中发出七条命令。该场景受 TPC-B 基准启发，但并不是真正的 TPC-B，因此才取了这个名字。

1. BEGIN;
2. UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
3. SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
4. UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
5. UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
6. INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
7. END;

如果选择 simple-update 内置脚本（也就是 -N），事务中将不包含第 4 步和第 5 步。这会避免在这些表上发生更新争用，但也会让该测试场景更不像 TPC-B。

如果选择 select-only 内置脚本（也就是 -S），则只执行SELECT。

自定义脚本

pgbench 支持自定义基准测试场景：只需用从文件中读取的事务脚本（-f 选项）替换默认事务脚本（见上文）即可。在这种情况下，一个“事务”就表示脚本文件的一次执行。

脚本文件包含一个或多个以分号结束的 SQL 命令。空行以及以--开头的行会被忽略。脚本文件还可以包含“元命令”，它们由pgbench 自身解释，详见下文。

> **注意**
>
> 在 PostgreSQL 9.6 之前，脚本文件中的 SQL 命令以换行结束，因此不能跨行。现在连续 SQL 命令之间必须用分号分隔（如果 SQL 命令后面跟着一个元命令，则不需要分号）。如

果需要创建一个既能在旧版也能在新版pgbench 下工作的脚本文件，务必将每个 SQL 命令写在单独一行，并以分号结束。

脚本文件提供了简单的变量替换功能。变量名必须由字母（包括非拉丁字母）、数字和下划线组成，且首字符不能是数字。变量可以通过前面介绍的命令行-D 选项设置，也可以通过下面介绍的元命令设置。除了由-D 命令行选项预设的变量外，还有一些自动预设的变量，列于表 242 中。使用 -D 为这些变量指定的值优先于自动预设值。设置变量后，就可以通过书写：*variablename* 将变量值插入 SQL 命令。在运行多个客户端会话时，每个会话都有自己的一组变量。

表 242. 自动变量

变量	简介
client_id	标识客户端会话的唯一编号（从零开始）
default_seed	默认在 hash 函数中使用的种子
random_seed	随机数生成器种子（除非被-D 覆盖）
scale	当前比例因子

脚本文件中的元命令以反斜线（\）开头，通常延伸到行尾，不过也可以通过写反斜线换行继续到后续行。元命令的参数以空白分隔。支持的元命令如下：

\if *expression*
\elif *expression*
\else
\endif

这一组命令实现了可嵌套的条件块，类似于psql 中的\if *expression*。条件表达式与\set 中的表达式相同，非零值会被解释为真。

\set *varname expression*

将变量 *varname* 设置为根据 *expression* 计算出的值。该表达式可以包含 NULL 常量、布尔常量TRUE 和 FALSE、5432 这样的整数常量、3.14159 这样的双精度常量、对变量的引用：*variablename*、操作符（保留其通常的 SQL 优先级和结合性）、函数调用、SQL CASE 通用条件表达式以及括号。

函数和大部分操作符在 NULL 输入上会返回 NULL。

就条件判断而言，非零数值为TRUE，零值和NULL 为 FALSE。

在没有为 CASE 提供最终的 ELSE 子句时，默认值是NULL。

示例：

```
\set ntellers 10 * :scale
\set aid (1021 * random(1, 100000 * :scale)) % \
           (100000 * :scale) + 1
\set divx CASE WHEN :x <> 0 THEN :y/:x ELSE NULL END
```

`\sleep` *number* [us | ms | s]

让脚本执行休眠指定时长，单位可以是微秒（us）、毫秒（ms）或秒（s）。如果省略单位，则默认为秒。*number* 可以是整数常量，也可以是引用了整数值变量的：*variablename*。

示例：

```
\sleep 10 ms
```

`\setshell` *varname command* [*argument* ...]

将变量 *varname* 设置为 shell 命令 *command* 在给定 *argument* 参数下的结果。该命令必须通过标准输出返回一个整数值。

command 和每个 *argument* 都可以是文本常量，也可以是引用某个变量的：*variablename*。如果要使用以冒号开头的 *argument*，请在*argument* 开头再写一个冒号。

示例：

```
\setshell variable_to_be_assigned command literal_argument :variable ::literal_starting_
with_colon
```

`\shell` *command* [*argument* ...]

与`\setshell` 相同，但命令结果会被丢弃。

示例：

```
\shell command literal_argument :variable ::literal_starting_with_colon
```

内置操作符

表 243 中列出的算术、按位、比较和逻辑操作符都内置于pgbench 中，可用于`\set` 中的表达式。

表 243. 按优先级从低到高排列的 pgbench 操作符

操作符	简介	示例	结果
OR	逻辑或	5 or 0	TRUE
AND	逻辑与	3 and 0	FALSE
NOT	逻辑非	not false	TRUE
IS [NOT] (NULL|TRUE|FALSE)	值测试	1 is null	FALSE
ISNULL|NOTNULL	空值测试	1 notnull	TRUE
=	等于	5 = 4	FALSE
<>	不等于	5 <> 4	TRUE
!=	不等于	5 != 5	FALSE
<	小于	5 < 4	FALSE

操作符	简介	示例	结果
<=	小于或等于	5 <= 4	FALSE
>	大于	5 > 4	TRUE
>=	大于或等于	5 >= 4	TRUE
|	整数按位或	1 | 2	3
#	整数按位异或	1 # 3	2
&	整数按位与	1 & 3	1
~	整数按位非	~ 1	-2
<<	整数按位左移	1 << 2	4
>>	整数按位右移	8 >> 2	2
+	加法	5 + 4	9
-	减法	3 - 2.0	1.0
*	乘法	5 * 4	20
/	除法（整数会截断结果）	5 / 3	1
%	取模	3 % 2	1
-	取相反数	- 2.0	-2.0

内置函数

表 244 中列出的函数都内置于 pgbench，可用于\set 中的表达式。

表 244. pgbench 函数

函数	abs(*a*)
返回类型	与*a* 相同
简介	绝对值
示例	abs(-17)
结果	17
函数	debug(*a*)
返回类型	与*a* 相同
简介	将*a* 打印到stderr，并返回*a*
示例	debug(5432.1)
结果	5432.1
函数	double(*i*)
返回类型	double
简介	转换为 double
示例	double(5432)
结果	5432.0
函数	exp(*x*)
返回类型	double

简介	指数函数
示例	`exp(1.0)`
结果	`2.718281828459045`
函数	`greatest(a [, ... ] )`
返回类型	若任一*a* 为 double，则为 double，否则为 integer
简介	参数中的最大值
示例	`greatest(5, 4, 3, 2)`
结果	`5`
函数	`hash(a [, seed ] )`
返回类型	integer
简介	`hash_murmur2()` 的别名
示例	`hash(10, 5432)`
结果	`-5817877081768721676`
函数	`hash_fnv1a(a [, seed ] )`
返回类型	integer
简介	FNV-1a hash[1]
示例	`hash_fnv1a(10, 5432)`
结果	`-7793829335365542153`
函数	`hash_murmur2(a [, seed ] )`
返回类型	integer
简介	MurmurHash2 hash[2]
示例	`hash_murmur2(10, 5432)`
结果	`-5817877081768721676`
函数	`int(x)`
返回类型	integer
简介	转换为 int
示例	`int(5.4 + 3.8)`
结果	`9`
函数	`least(a [, ... ] )`
返回类型	若任一*a* 为 double，则为 double，否则为 integer
简介	参数中的最小值
示例	`least(5, 4, 3, 2.1)`
结果	`2.1`

[1] https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function
[2] https://en.wikipedia.org/wiki/MurmurHash

函数	ln(*x*)
返回类型	double
简介	自然对数
示例	ln(2.718281828459045)
结果	1.0
函数	mod(*i*, *j*)
返回类型	integer
简介	取模
示例	mod(54, 32)
结果	22
函数	pi()
返回类型	double
简介	常量 PI 的值
示例	pi()
结果	3.14159265358979323846
函数	pow(*x*, *y*), power(*x*, *y*)
返回类型	double
简介	求幂
示例	pow(2.0, 10),power(2.0, 10)
结果	1024.0
函数	random(*lb*, *ub*)
返回类型	integer
简介	[lb, ub] 中均匀分布的随机整数
示例	random(1, 10)
结果	1 与10 之间的整数
函数	random_exponential(*lb*, *ub*, *parameter*)
返回类型	integer
简介	[lb, ub] 中服从指数分布的随机整数，见下文
示例	random_exponential(1, 10, 3.0)
结果	1 与10 之间的整数
函数	random_gaussian(*lb*, *ub*, *parameter*)
返回类型	integer
简介	[lb, ub] 中服从高斯分布的随机整数，见下文
示例	random_gaussian(1, 10, 2.5)
结果	1 与10 之间的整数

函数	`random_zipfian(lb, ub, parameter)`
返回类型	integer
简介	`[lb, ub]` 中服从 Zipfian 分布的随机整数，见下文
示例	`random_zipfian(1, 10, 1.5)`
结果	`1` 与`10` 之间的整数
函数	`sqrt(x)`
返回类型	double
简介	平方根
示例	`sqrt(2.0)`
结果	`1.414213562`

`random` 函数使用均匀分布生成值，也就是说，指定范围内的所有值都以相同概率被抽取。`random_exponential`、`random_gaussian` 和 `random_zipfian` 函数则需要额外提供一个 double 参数，用来确定分布的精确形状。

- 对于指数分布，*`parameter`* 通过在 *`parameter`* 处截断一个快速衰减的指数分布，再将其投影到边界之间的整数上，从而控制分布。准确地说，令

  f(x) = exp(-parameter * (x - min) / (max - min + 1)) / (1 - exp(-parameter))

  则 *`min`* 和 *`max`* 之间（含边界）的值 *`i`* 会以 `f(i) - f(i + 1)` 的概率被抽中。

  直观地说，*`parameter`* 越大，越靠近*`min`* 的值越容易被抽到，而越靠近*`max`* 的值越不容易被抽到。*`parameter`* 越接近 0，分布就越平坦（也就越均匀）。对这种分布的一个粗略近似是：范围内出现频率最高的 1% 的值，即最靠近*`min`* 的那些值，大约会占到*`parameter`*% 的抽样次数。*`parameter`* 必须严格大于 0。

- 对于高斯分布，该区间会映射到一个标准正态分布（经典钟形高斯曲线），并在左侧 `-parameter` 和右侧`+parameter` 处截断。区间中部的值更容易被抽到。准确地说，如果 `PHI(x)` 是标准正态分布的累积分布函数，均值`mu` 定义为`(max + min) / 2.0`，则有

  f(x) = PHI(2.0 * parameter * (x - mu) / (max - min + 1)) /
      (2.0 * PHI(parameter) - 1)

  则 *`min`* 和 *`max`*（包含边界）之间的值 *`i`* 被抽中的概率为：`f(i + 0.5) - f(i - 0.5)`。直观地说，*`parameter`* 越大，越靠近区间中间的值被抽到的频率越高，而越靠近*`min`* 和 *`max`* 边界的值被抽到的频率越低。大约 67% 的值会落在区间中部 `1.0 / parameter` 这一段内，也就是均值两侧各占区间长度`0.5 / parameter` 的范围内；约 95% 的值会落在区间中部 `2.0 / parameter` 这一段内，也就是均值两侧各占区间长度`1.0 / parameter` 的范围内。例如，如果*`parameter`* 为 4.0，则 67% 的值会落在区间中间四分之一（1.0 / 4.0）内，也就是从`3.0 / 8.0` 到 `5.0 / 8.0`；95% 的值会落在区间中间一半（`2.0 / 4.0`）内，也就是第二和第三四分位。考虑到 Box-Muller 变换的性能，*`parameter`* 的最小值为 2.0。

- `random_zipfian` 会生成一个近似的有界 Zipfian 分布。对于 (0, 1) 内的 *`parameter`*，采用的近似算法来自 Jim Gray 等人在 SIGMOD 1994 发表的 “Quickly Generating Billion-Record Synthetic Databases” 。对于 (1, 1000) 内的*`parameter`*，采用的拒绝法基于 Luc Devroye 的 “Non-Uniform Random Variate Generation”，Springer 1986，第 550-551 页。当参

数值为 1.0 时，该分布没有定义。当参数值接近且大于 1.0，并且范围较小时，该函数的性能较差。

parameter 定义该分布的倾斜程度。*parameter* 越大，越靠近区间起始处的值被抽到的频率就越高。*parameter* 越接近 0，输出分布就越平坦（越均匀）。这种分布满足：假设范围从 1 开始，抽到*k* 与抽到 *k+1* 的概率之比为 `((`*k*`+1)/`*k*`)**`*parameter*。例如，`random_zipfian(1, ..., 2.5)` 生成值 `1` 的频率大约是生成 `2` 的 `(2/1)**2.5 = 5.66` 倍，而后者的生成频率又大约是生成`3` 的 `(3/2)**2.5 = 2.76` 倍，依此类推。

Hash 函数 `hash`、`hash_murmur2` 和 `hash_fnv1a` 接受一个输入值和一个可选的种子参数。如果未提供种子，则使用 `:default_seed` 的值，该值随机初始化，除非通过命令行 `-D` 选项设置。Hash 函数可以用来打散随机函数的分布，例如 `random_zipfian` 或 `random_exponential`。例如，以下 pgbench 脚本模拟了社交媒体和博客平台中可能出现的典型真实工作负载，其中少数账户产生了过多的负载：

```
\set r random_zipfian(0, 100000000, 1.07)
\set k abs(hash(:r)) % 1000000
```

某些情况下需要几个彼此不相关的不同分布，这时隐式种子参数就能派上用场：

```
\set k1 abs(hash(:r, :default_seed + 123)) % 1000000
\set k2 abs(hash(:r, :default_seed + 321)) % 1000000
```

作为一个示例，内置的类 TPC-B 事务的全部定义是：

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
 CURRENT_TIMESTAMP);
END;
```

该脚本允许事务的每次迭代都引用不同的随机选中行。（这个示例也说明了为什么每个客户端会话都必须拥有自己的变量 – 否则它们就无法彼此独立地访问不同的行。）

逐事务日志记录

使用 `-l` 选项时（但未指定 `--aggregate-interval` 选项），pgbench 会将每个事务的信息写入日志文件。日志文件名为 *prefix*`.`*nnn*，其中*prefix* 默认为 `pgbench_log`，*nnn* 是 pgbench 进程的 PID。可以用 `--log-prefix` 选项修改此前缀。如果 `-j` 选项为 2 或更大，即存在多个工作线程，则每个工作线程都有自己的日志文件。第一个工作线程的日志文件名与标准单工作线程情况相同。其他工作线程的附加日志文件名为*prefix*`.`*nnn*`.`*mmm*，其中*mmm* 是每个工作线程从 1 开始的顺序编号。

日志格式如下：

```
client_id transaction_no time script_no time_epoch time_us [ schedule_lag ]
```

其中，*client_id* 表示运行该事务的客户端会话，*transaction_no* 记录该会话已运行的事务数，*time* 是事务经过的总时间，单位为微秒，*script_no* 标识使用的脚本文件（在通过 -f 或 -b 指定多个脚本时很有用），而 *time_epoch*/*time_us* 分别是 Unix 纪元时间戳和以微秒计的偏移量（适合用来生成带小数秒的 ISO 8601 时间戳），表示事务完成的时间。*schedule_lag* 字段是事务计划开始时间与实际开始时间之间的差值，单位为微秒。它仅在使用 --rate 选项时出现。如果同时使用 --rate 和 --latency-limit，则被跳过事务的 *time* 将报告为 skipped。

这里是在单个客户端运行中生成的一个日志文件的片段：

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

另一个示例使用的是--rate=100 以及--latency-limit=5（注意额外的 *schedule_lag* 列）：

```
0 81 4621 0 1412881037 912698 3005
0 82 6173 0 1412881037 914578 4304
0 83 skipped 0 1412881037 914578 5217
0 83 skipped 0 1412881037 914578 5099
0 83 4722 0 1412881037 916203 3108
0 84 4142 0 1412881037 918023 2333
0 85 2465 0 1412881037 919759 740
```

在这个示例中，事务 82 迟到了，因为它的延迟（6.173 ms）超过了 5 ms 限制。接下来的两个事务被跳过，因为它们在开始之前就已经迟到了。

在能够处理大量事务的硬件上运行长时间测试时，日志文件可能会变得非常大。可以使用 --sampling-rate 选项，仅记录事务的随机样本。

聚合日志记录

使用 --aggregate-interval 选项时，日志文件采用另一种格式：

```
interval_start num_transactions sum_latency sum_latency_2 min_latency max_latency [ sum_
lag sum_lag_2 min_lag max_lag [ skipped ] ]
```

其中，*interval_start* 是时间区间的开始时间（以 Unix 纪元时间戳表示），*num_transactions* 是区间内的事务数，*sum_latency* 是区间内事务延迟的总和，*sum_latency_2* 是区间内事务延迟的平方和，*min_latency* 是区间内的最小延迟，而 *max_latency* 是区间内的最大延迟。接下来的字段 *sum_lag*、*sum_lag_2*、*min_lag* 和 *max_lag* 仅在使用 --rate 选项时出现。它们提供各事务等待前一事务完成的时间统计，即各事务计划开始时间与实际开始时间之间的差值。最后一个字段 *skipped* 仅在还使用 --latency-limit 选项时出现。它记录因开始时间过晚而被跳过的事务数。每个事务都计入其提交时所在的时间区间。

下面是一些输出示例：

```
1345828501 5601 1542744 483552416 61 2573
1345828503 7884 1979812 565806736 60 1479
1345828505 7208 1979422 567277552 59 1391
1345828507 7685 1980268 569784714 60 1398
1345828509 7073 1979779 573489941 236 1411
```

请注意，普通（未聚合）日志格式会显示每个事务所使用的脚本，而聚合格式不会。因此，如果需要按脚本区分的数据，就必须自行聚合。

逐语句延迟

使用 `-r` 选项时，pgbench 会收集每个客户端执行的每条语句所经过的事务时间。基准测试完成后，它会报告这些值的平均值，称为每条语句的延迟。

对于默认脚本，输出与下面类似：

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
latency average = 15.844 ms
latency stddev = 2.715 ms
tps = 618.764555 (including connections establishing)
tps = 622.977698 (excluding connections establishing)
statement latencies in milliseconds:
        0.002  \set aid random(1, 100000 * :scale)
        0.005  \set bid random(1, 1 * :scale)
        0.002  \set tid random(1, 10 * :scale)
        0.001  \set delta random(-5000, 5000)
        0.326  BEGIN;
        0.603  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
        0.454  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
        5.528  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
        7.335  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
        0.371  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :
aid, :delta, CURRENT_TIMESTAMP);
        1.212  END;
```

如果指定了多个脚本文件，则会分别为每个脚本文件报告平均值。

注意，为逐语句延迟计算收集额外的计时信息会带来一定开销。这会拖慢平均执行速度，并降低计算出的 TPS。减速幅度在很大程度上取决于平台和硬件。比较启用和未启用延迟报告时的平均 TPS 值，是判断这一计时开销是否显著的好方法。

良好实践

很容易用 pgbench 得出完全没有意义的数字。下面给出一些有助于获得有用结果的准则。

首先，绝不要相信任何只运行了几秒钟的测试。使用 `-t` 或 `-T` 选项让测试至少持续几分钟，以便平滑掉噪声。在某些情况下，可能需要数小时才能得到可复现的结果。一个好做法是把同一测试运行几次，看看结果是否可复现。

对于默认的类 TPC-B 测试场景，初始化比例因子（`-s`）应至少与计划测试的最大客户端数（`-c`）一样大；否则，测到的主要将是更新争用。`pgbench_branches` 表中只有 `-s` 行，而每个事务都要更新其中一行，因此`-c` 超过 `-s` 时，必然会有大量事务阻塞等待其他事务。

默认测试场景还会对表初始化后的时间长短非常敏感：表中死元组和无效空间的累积会改变结果。要理解这些结果，必须跟踪更新总数以及何时发生清理。如果启用了自动清理，它可能会给测得的性能带来不可预测的变化。

pgbench 的一个局限是：在尝试测试大量客户端会话时，它自己也可能成为瓶颈。可以通过在与数据库服务器不同的机器上运行pgbench 来缓解这一点，不过网络延迟必须足够低。甚至可以在多台客户端机器上同时运行多个pgbench 实例，对同一台数据库服务器施压。

安全性

如果不受信任的用户能够访问尚未采用模式的安全使用方式的数据库，就不要在该数据库中运行 pgbench。pgbench 使用非限定名称，并且不会更改搜索路径。

pg_config

pg_config — 获取已安装的 PostgreSQL 版本的信息

大纲

`pg_config` [*option*...]

描述

pg_config 工具打印当前已安装的 PostgreSQL 版本的配置参数。例如，希望与PostgreSQL 对接的软件包可以使用它，以便找到所需的头文件和库。

选项

要使用 pg_config，请提供下列一个或多个选项：

`--bindir`

打印用户可执行文件所在的位置。例如，可用它来查找`psql` 程序。通常这也是 `pg_config` 程序所在的位置。

`--docdir`

打印文档文件所在的位置。

`--htmldir`

打印 HTML 文档文件所在的位置。

`--includedir`

打印客户端接口的 C 头文件所在的位置。

`--pkgincludedir`

打印其他 C 头文件所在的位置。

`--includedir-server`

打印服务器编程所用 C 头文件所在的位置。

`--libdir`

打印目标代码库所在的位置。

`--pkglibdir`

打印动态可加载模块所在的位置，或服务器将搜索这些模块的位置。（其他与体系结构相关的数据文件也可能安装在该目录中。）

`--localedir`

打印区域设置支持文件所在的位置。（如果在构建 PostgreSQL 时未配置区域设置支持，则这将是一个空字符串。）

`--mandir`

打印手册页所在的位置。

`--sharedir`

打印与体系结构无关的支持文件所在的位置。

`--sysconfdir`

打印系统范围的配置文件所在的位置。

`--pgxs`

打印扩展 makefile 的位置。

`--configure`

打印为构建 PostgreSQL 而运行 `configure` 脚本时给定的选项。这可用于重现完全相同的配置，或者查明某个二进制软件包是用哪些选项构建的。（但请注意，二进制软件包通常包含供应商特定的自定义补丁。）另请参见下面的示例。

`--cc`

打印构建 PostgreSQL 时所用的 `CC` 变量值。这显示所使用的 C 编译器。

`--cppflags`

打印构建 PostgreSQL 时所用的 `CPPFLAGS` 变量值。这显示预处理阶段所需的 C 编译器开关（通常是`-I` 开关）。

`--cflags`

打印构建 PostgreSQL 时所用的 `CFLAGS` 变量值。这显示 C 编译器开关。

`--cflags_sl`

打印构建 PostgreSQL 时所用的 `CFLAGS_SL` 变量值。这显示构建共享库时使用的额外 C 编译器开关。

`--ldflags`

打印构建 PostgreSQL 时所用的 `LDFLAGS` 变量值。这显示链接器开关。

`--ldflags_ex`

打印构建 PostgreSQL 时所用的 `LDFLAGS_EX` 变量值。这显示仅用于构建可执行文件的链接器开关。

`--ldflags_sl`

打印构建 PostgreSQL 时所用的 `LDFLAGS_SL` 变量值。这显示仅用于构建共享库的链接器开关。

`--libs`

打印构建 PostgreSQL 时所用的 `LIBS` 变量值。这通常包含外部库的`-l` 开关，这些外部库会被链接进PostgreSQL。

`--version`

打印 PostgreSQL 的版本。

`-?`
`--help`

显示有关 pg_config 命令行参数的帮助，然后退出。

如果给出了多个选项，信息会按给出的顺序输出，每行一项。如果未给出任何选项，则会输出所有可用信息，并带有标签。

注解

选项 `--docdir`、`--pkgincludedir`、`--localedir`、`--mandir`、`--sharedir`、`--sysconfdir`、`--cc`、`--cppflags`、`--cflags`、`--cflags_sl`、`--ldflags`、`--ldflags_sl` 以及 `--libs` 是在 PostgreSQL 8.1 中加入的。选项 `--htmldir` 是在 PostgreSQL 8.4 中加入的。选项 `--ldflags_ex` 是在 PostgreSQL 9.0 中加入的。

示例

要重现当前 PostgreSQL 安装的构建配置，请运行下列命令：

```
eval ./configure `pg_config --configure`
```

`pg_config --configure` 的输出包含 shell 引号，因此带空格的参数能够被正确表示。因此，要得到正确的结果，需要使用`eval`。

pg_dump

pg_dump — 将一个 PostgreSQL 数据库转储为脚本文件或其他归档文件

大纲

```
pg_dump [connection-option...] [option...] [dbname]
```

描述

pg_dump 是一个用于备份 PostgreSQL 数据库的工具。即使数据库正在被并发使用，它也能生成一致的备份。pg_dump 不会阻塞其他用户访问数据库（无论读还是写）。

pg_dump 只转储单个数据库。要备份整个集簇，或者备份一个集簇中所有数据库共有的全局对象（如角色和表空间），请使用 pg_dumpall。

转储可以输出为脚本格式或归档文件格式。脚本转储是纯文本文件，包含把数据库重建到保存时状态所需的 SQL 命令。要从这样的脚本恢复，只需将其交给 psql。脚本文件甚至可以在其他机器和其他体系结构上用于重建数据库；经过一些修改后，甚至也可以用于其他 SQL 数据库产品。

另一类归档文件格式必须结合 pg_restore 来重建数据库。它们允许 pg_restore 有选择地恢复某些内容，甚至在恢复之前重新排列条目。归档文件格式被设计为可跨体系结构移植。

使用归档文件格式并配合 pg_restore 时，pg_dump 提供了一种灵活的归档和传输机制。pg_dump 可用于备份整个数据库，而 pg_restore 可用于检查归档和/或选择要恢复的数据库部分。最灵活的输出文件格式是 “custom” 格式（`-Fc`）和 “directory” 格式（`-Fd`）。它们允许选择和重新排序所有归档条目，支持并行恢复，并且默认会压缩。只有 “directory” 格式支持并行转储。

运行 pg_dump 时，应检查输出中是否有任何警告（打印到标准错误），尤其要结合下面列出的限制来查看。

选项

以下命令行选项控制输出的内容和格式。

dbname

指定要转储的数据库名称。如果未指定，则使用环境变量`PGDATABASE`。如果未设置该变量，则使用连接指定的用户名。

`-a`
`--data-only`

只转储数据，不转储模式（数据定义）。会转储表数据、大对象和序列值。

此选项类似于指定 `--section=data`，但出于历史原因，两者并不完全相同。

-b
--blobs

在转储中包含大对象。除非指定了 --schema、--table 或 --schema-only，否则这是默认行为。因此，-b 开关只在已经请求了特定模式或表的转储中，需要把大对象加回来时才有用。注意，大对象被视为数据，因此在使用--data-only 时会包含，在使用 --schema-only 时则不会包含。

-B
--no-blobs

在转储中排除大对象。

当同时给出 -b 和 -B 时，行为是在转储数据时输出大对象，请参阅-b 文档。

-c
--clean

在输出创建这些对象的命令之前，先输出用于 DROP 所有被转储数据库对象的命令。当恢复要覆盖现有数据库时，此选项很有用。如果目标数据库中某些对象不存在，则在恢复期间会报告可忽略的错误信息，除非还指定了 --if-exists。

当生成一个归档（非文本）输出文件时，此选项将被忽略。对于归档格式，可以在调用 pg_restore 时指定该选项。

-C
--create

让输出以创建数据库本身并重新连接到新建数据库的命令开始。（使用这种形式的脚本时，在目标安装中先连接到哪个数据库再运行脚本都无关紧要。）如果还指定了 --clean，则脚本会在重新连接到目标数据库之前先删除并重新创建它。

使用 --create 时，输出还会包含该数据库的注释（如果有），以及该数据库特有的任何配置变量设置，也就是提及该数据库的任何 ALTER DATABASE ... SET ... 和 ALTER ROLE ... IN DATABASE ... SET ... 命令。除非指定了 --no-acl，否则数据库本身的访问权限也会被转储。

当生成一个归档（非文本）输出文件时，此选项将被忽略。对于归档格式，可以在调用 pg_restore 时指定该选项。

-E *encoding*
--encoding=*encoding*

在指定的字符集编码中创建转储。默认情况下，转储将以数据库编码创建。（获得相同结果的另一种方法是将PGCLIENTENCODING 环境变量设置为所需的转储编码。）

-f *file*
--file=*file*

将输出发送到指定文件。对于基于文件的输出格式，可以省略此参数，这时使用标准输出。不过，对于目录输出格式，必须给出此参数，因为它指定的是目标目录而不是文件。在这种情况下，该目录由 pg_dump 创建，并且在此之前必须不存在。

`-F format`
`--format=format`

选择输出的格式。*format* 可以是以下之一：

`p`
`plain`

输出纯文本 SQL 脚本文件（默认）。

`c`
`custom`

输出适合供 pg_restore 使用的 custom 格式归档。与目录输出格式一起，这是最灵活的输出格式，因为它允许在恢复时手工选择并重新排序归档条目。这种格式默认也会压缩。

`d`
`directory`

输出适合供 pg_restore 使用的 directory 格式归档。这会创建一个目录，其中每个被转储的表和大对象各有一个文件，外加一个所谓的目录表（Table of Contents）文件，以机器可读格式描述被转储对象，pg_restore 可以读取它。directory 格式归档可以用标准 Unix 工具来操作；例如，未压缩归档中的文件可以使用 gzip 工具进行压缩。该格式默认压缩，并且支持并行转储。

`t`
`tar`

输出 `tar` 格式归档，适合供pg_restore 使用。tar 格式与 directory 格式兼容：解开一个 tar 格式归档就会得到一个有效的 directory 格式归档。不过，tar 格式不支持压缩。另外，使用 tar 格式时，在恢复过程中不能改变表数据项的相对顺序。

`-j njobs`
`--jobs=njobs`

以并行方式运行转储，同时转储 *njobs* 个表。此选项减少执行转储所需的时间，但也会增加数据库服务器的负载。只能在目录输出格式下使用此选项，因为只有这种输出格式允许多个进程同时写入数据。

pg_dump 将打开 *njobs* + 1 个数据库连接，因此请确保max_connections 设置足够高，能够容纳所有连接。

在并行转储运行期间请求数据库对象上的排他锁，可能导致转储失败。原因是 pg_dump 的主进程会对稍后由工作进程转储的对象请求共享锁，以确保在转储运行期间没有人删除这些对象。如果另一个客户端随后请求某个表上的排他锁，该锁不会被授予，而是会排队等待主进程释放共享锁。于是，对该表的任何其他访问也都不会被授予，并会排在该排他锁请求之后，其中包括试图转储该表的工作进程。如果没有任何预防措施，这就会形成一个经典的死锁场景。为检测这种冲突，pg_dump 工作进程会使用 `NOWAIT` 选项再请求一个共享锁。如果工作进程拿不到这个共享锁，就说明这期间已经有人请求了排他锁，而此时已经无法继续转储，因此 pg_dump 只能中止转储。

要获得一致的备份，数据库服务器需要支持同步快照。这一特性是在 PostgreSQL 9.2 中为主库引入的，在 10 中为备库引入。有了这个特性，数据库客户端即使使用不同连接，也能保证看到相同的数据集。`pg_dump -j` 会使用多个数据库连接：它会先由主进程连接数据库一

次，然后每个工作任务再各连一次。若没有同步快照特性，就无法保证各个工作任务在各自连接中看到相同的数据，这可能导致备份不一致。

如果要对 9.2 之前的服务器执行并行转储，必须确保从主进程连接数据库起，到最后一个工作任务连接数据库为止，数据库内容都不发生变化。最简单的方法是在开始备份之前，暂停所有访问数据库并修改数据的进程（DDL 和 DML）。对 9.2 之前的PostgreSQL 服务器运行 `pg_dump -j` 时，还必须指定`--no-synchronized-snapshots` 参数。

`-n` *`schema`*
`--schema=`*`schema`*

只转储匹配 *`schema`* 的模式；这既会选择模式本身，也会选择其中包含的所有对象。未指定此选项时，将转储目标数据库中的所有非系统模式。可以通过写多个 `-n` 开关来选择多个模式。*`schema`* 参数按照 psql 的 `\d` 命令所使用的同样规则进行解释（见模式），因此也可以通过在模式中使用通配符来选择多个模式。使用通配符时，如有需要请小心为模式加引号，以防止 shell 展开通配符；见下文的示例。

> **注意**
>
> 当指定 `-n` 时，pg_dump 不会尝试转储所选模式可能依赖的任何其他数据库对象。因此，不能保证特定模式转储的结果能够单独成功恢复到一个干净的数据库中。

> **注意**
>
> 指定 `-n` 时，不会转储大对象等非模式对象。可以使用 `--blobs` 开关把大对象加回转储中。

`-N` *`schema`*
`--exclude-schema=`*`schema`*

不转储任何名称匹配 *`schema`* 的模式。该匹配条件按照与 `-n` 相同的规则解释。可以多次指定 `-N`，以排除匹配所给条件中任意一个的模式。

当同时给出 `-n` 和 `-N` 时，其行为是只转储至少匹配一个 `-n` 开关但不匹配任何 `-N` 开关的模式。如果出现 `-N` 而没有 `-n`，那么匹配 `-N` 的模式会从原本的正常转储中排除。

`-o`
`--oids`

将对象标识符（OID）作为每个表的数据的一部分进行转储。如果应用程序以某种方式引用 OID 列（例如在外键约束中），请使用此选项。否则，不应使用此选项。

`-O`
`--no-owner`

不输出用于把对象所有权设置成与原始数据库一致的命令。默认情况下，pg_dump 会发出 `ALTER OWNER` 或 `SET SESSION AUTHORIZATION` 语句来设置新建数据库对象的所有权。除非脚本由超级用户（或拥有脚本中所有对象的同一用户）启动，否则这些语句会在运行脚本时报错。若要创建一个可由任意用户恢复、并让该用户拥有所有对象的脚本，请指定 `-O`。

当生成一个归档（非文本）输出文件时，此选项将被忽略。对于归档格式，可以在调用 pg_restore 时指定该选项。

-R
--no-reconnect

这个选项已经过时，但仍然被接受以保持向后兼容性。

-s
--schema-only

只转储对象定义（模式），不转储数据。

此选项与 --data-only 的作用相反。它类似于指定 --section=pre-data --section=post-data，但由于历史原因并不完全相同。

（不要把它与 --schema 选项混淆，后者中的“schema”一词含义不同。）

要只排除数据库中某些表的表数据，请参见 --exclude-table-data。

-S *username*
--superuser=*username*

指定在禁用触发器时要使用的超级用户名。这只在使用 --disable-triggers 时相关。（通常最好省略它，而是以超级用户身份运行生成的脚本。）

-t *table*
--table=*table*

只转储名称匹配 *table* 的表。这里，“表”包括视图、物化视图、序列和外部表。可以通过写多个 -t 开关来选择多个表。*table* 参数按照 psql 的 \d 命令所使用的同样规则进行解释（见模式），因此也可以通过在模式中使用通配符来选择多个表。使用通配符时，如有需要请小心为模式加引号，以防止 shell 展开通配符；见下文的示例。

使用 -t 时，-n 和 -N 开关没有作用，因为由 -t 选中的表无论这些开关如何设置都会被转储，而非表对象则不会被转储。

注意

当指定 -t 时，pg_dump 不会尝试转储所选表可能依赖的任何其他数据库对象。因此，不能保证特定表转储的结果能够单独成功恢复到一个干净的数据库中。

注意

-t 开关的行为与 8.2 之前的 PostgreSQL 版本并不完全向上兼容。以前，写成 -t tab 会转储所有名为 tab 的表，现在则只转储默认搜索路径中可见的那个表。要获得旧行为，可以写成-t '*.tab'。此外，要选择特定模式中的表，必须写成类似 -t sch.tab 的形式，而不是旧的-n sch -t tab 写法。

-T *table*
--exclude-table=*table*

不转储任何匹配 *table* 的表。该模式按照与 -t 相同的规则解释。-T 可以给出多次，以排除匹配所给模式中任意一个的表。

当同时给出 -t 和 -T 时，其行为是只转储至少匹配一个 -t 开关但不匹配任何 -T 开关的表。如果出现 -T 而没有 -t，那么匹配 -T 的表会从原本的正常转储中排除。

-v
--verbose

指定详细模式。这会使 pg_dump 将详细的对象注释、开始/停止时间写入转储文件，并把进度消息写到标准错误。

-V
--version

打印 pg_dump 的版本并退出。

-x
--no-privileges
--no-acl

不转储访问权限（grant/revoke 命令）。

-Z *0..9*
--compress=*0..9*

指定所用的压缩级别。零表示不压缩。对于自定义格式和目录格式归档，该选项指定对各个表数据段的压缩，默认以适中的级别压缩。对于纯文本输出，设置非零压缩级别会压缩整个输出文件，就像将其传给gzip 处理一样；但默认不压缩。tar 归档格式目前完全不支持压缩。

--binary-upgrade

此选项供就地升级实用程序使用。不建议也不支持将其用于其他目的。该选项的行为可能在未来的版本中更改而不另行通知。

--column-inserts
--attribute-inserts

将数据转储为带有显式列名的 INSERT 命令 (INSERT INTO *table* (*column*, ...) VALUES ...)。这会使恢复变得非常缓慢；它主要用于生成可装入非 PostgreSQL 数据库的转储文件。不过，由于此选项为每一行生成单独的命令，恢复某一行时出错只会导致该行丢失，而不会导致整个表的内容丢失。

--disable-dollar-quoting

此选项禁用对函数体的美元引用，并强制用 SQL 标准字符串语法对它们加引号。

--disable-triggers

此选项只在创建仅数据转储时才相关。它指示 pg_dump 在输出中包含一些命令，以便在恢复数据时临时禁用目标表上的触发器。如果这些表上存在不希望在数据恢复期间触发的引用完整性检查或其他触发器，请使用此选项。

目前，为 --disable-triggers 输出的这些命令必须由超级用户执行。因此，还应通过 -S 指定一个超级用户名，或者更好的做法是谨慎地以超级用户身份运行生成的脚本。

当生成一个归档（非文本）输出文件时，此选项将被忽略。对于归档格式，可以在调用 pg_restore 时指定该选项。

--enable-row-security

此选项只在转储启用了行安全的表内容时才相关。默认情况下，pg_dump 会将 row_security 设置为 off，以确保把表中的所有数据都转储出来。如果用户没有足够的权限绕过行安全，则会抛出错误。该参数会指示 pg_dump 改为将 row_security 设置为 on，从而允许用户只转储其有权访问的那部分表内容。

请注意，如果当前使用此选项，通常还会希望让转储采用 INSERT 格式，因为恢复期间的 COPY FROM 不支持行安全。

--exclude-table-data=*table*

不转储任何匹配 *table* 模式的表的数据。该模式按照与 -t 相同的规则解释。可以多次指定 --exclude-table-data，以排除匹配所给模式中任意一个的表。当需要某个表的定义但不需要其中的数据时，此选项很有用。

要排除数据库中所有表的数据，请参阅--schema-only。

--if-exists

使用 DROP ... IF EXISTS 命令在 --clean 模式下删除对象。这会抑制原本可能报告的 “does not exist” 错误信息。除非同时指定了 --clean，否则此选项无效。

--inserts

将数据转储为 INSERT 命令（而不是 COPY）。这会使恢复变得非常缓慢；它主要用于生成可装载到非 PostgreSQL 数据库的转储。不过，由于此选项为每一行生成单独的命令，恢复某一行时出错只会导致该行丢失，而不会导致整个表的内容丢失。注意，如果重新排列了列顺序，恢复可能会完全失败。--column-inserts 选项不受列顺序变化影响，但会更慢。

--load-via-partition-root

当为表分区转储数据时，让 COPY 或 INSERT 语句以包含该分区的分区层次结构根为目标，而不是以该分区本身为目标。这会导致在加载数据时为每一行重新确定适当的分区。当在另一台服务器上恢复数据，而行并不总是落在与原始服务器相同的分区中时，这可能很有用。例如，如果分区列是 text 类型，而两个系统对用于排序该分区列的排序规则定义不同，就可能发生这种情况。

--lock-wait-timeout=*timeout*

在转储开始时，不要无限等待获取共享表锁。如果无法在指定的 *timeout* 内锁定某个表，就让转储失败。超时可以用 SET statement_timeout 接受的任意格式指定。（允许的格式因被转储源服务器的版本而异，但所有版本都接受以毫秒为单位的整数。）

--no-comments

不转储注释。

`--no-publications`

不要转储发布。

`--no-security-labels`

不要转储安全标签。

`--no-subscriptions`

不要转储订阅。

`--no-sync`

默认情况下，`pg_dump` 会等待所有文件都被安全写入磁盘。该选项会使 `pg_dump` 立即返回而不等待，这样更快，但意味着随后如果操作系统崩溃，转储可能会损坏。通常，此选项适用于测试，但不应在从生产安装转储数据时使用。

`--no-synchronized-snapshots`

此选项允许对 9.2 之前的服务器运行 `pg_dump -j`，更多详情请参阅`-j` 参数的文档。

`--no-tablespaces`

不输出用于选择表空间的命令。使用此选项时，所有对象在恢复时都会创建在当时默认的表空间中。

当生成一个归档（非文本）输出文件时，此选项将被忽略。对于归档格式，可以在调用 `pg_restore` 时指定该选项。

`--no-unlogged-table-data`

不转储不记录 WAL 的表的内容。此选项不影响是否转储表定义（模式），只会抑制表数据的转储。从备库转储时，不记录 WAL 的表中的数据始终会被排除。

`--quote-all-identifiers`

强制为所有标识符加引号。当从某个服务器转储数据库，而该服务器的 PostgreSQL 主版本与 pg_dump 不同，或者输出打算装入另一主版本的服务器时，推荐使用此选项。默认情况下，pg_dump 只会为在其自身主版本中属于保留字的标识符加引号。这有时会在处理其他版本服务器时导致兼容性问题，因为这些版本的保留字集合可能略有不同。使用 `--quote-all-identifiers` 可以防止此类问题，但代价是转储脚本更难阅读。

`--section=`*`sectionname`*

仅转储指定的部分。部分名称可以是 `pre-data`、`data` 或 `post-data`。可以多次指定此选项以选择多个部分。默认情况下是转储所有部分。

data 部分包含实际的表数据、大对象内容和序列值。post-data 部分包含索引、触发器、规则，以及除已验证的检查约束之外的其他约束定义。pre-data 部分包含所有其他数据定义项。

`--serializable-deferrable`

使用 `serializable` 事务来执行转储，以确保所用快照与后续数据库状态一致；但这是通过等待事务流到达一个不会出现异常的时点来实现的，从而避免转储失败或导致其他事

务因 serialization_failure 而回滚。有关事务隔离和并发控制的更多信息，请参见第 13 章。

这个选项对于仅用于灾难恢复的转储没有好处。但对于用来加载数据库副本，以供报表或其他只读负载共享，而原始数据库继续更新的转储，它可能有用。如果不使用它，转储可能反映出一种与最终提交事务的任何串行执行都不一致的状态。例如，如果使用批处理技术，转储中可能显示某个批次已经关闭，但批次中的全部条目却并未出现。

如果在启动 pg_dump 时没有活动的读写事务，此选项不会带来任何差别。如果存在活动的读写事务，转储开始时间可能会被延迟一个不确定的时长。一旦开始运行，使用或不使用该开关的性能都是相同的。

--snapshot=*snapshotname*

在制作数据库转储时，使用指定的同步快照（详见表 9.82）。

当需要将转储与逻辑复制槽（参见第 49 章）或与并发会话同步时，此选项很有用。

在并行转储的情况下，将使用此选项定义的快照名称，而不是重新获取一个新快照。

--strict-names

要求每个模式（-n/--schema）和表（-t/--table）限定条件至少匹配要转储的数据库中的一个模式或表。注意，如果所有模式和表限定条件都找不到匹配项，pg_dump 即使没有使用 --strict-names 也会报错。

此选项不影响 -N/--exclude-schema、-T/--exclude-table 或 --exclude-table-data。排除模式未匹配到任何对象不会被视为错误。

--use-set-session-authorization

输出符合 SQL 标准的 SET SESSION AUTHORIZATION 命令，而不是 ALTER OWNER 命令来确定对象所有权。这会让转储更符合标准，但根据转储中对象的历史，可能无法正确恢复。另外，使用 SET SESSION AUTHORIZATION 的转储肯定需要超级用户权限才能正确恢复，而 ALTER OWNER 只需较低权限。

-?
--help

显示关于 pg_dump 命令行参数的帮助信息，并退出。

以下命令行选项控制数据库连接参数。

-d *dbname*
--dbname=*dbname*

指定要连接的数据库名称。这等价于在命令行上把 *dbname* 作为第一个非选项参数指定。*dbname* 可以是连接字符串。如果是这样，连接字符串中的参数将覆盖任何冲突的命令行选项。

-h *host*
--host=*host*

指定服务器所在机器的主机名。如果值以斜杠开头，则被用作 Unix 域套接字的目录。默认值取自 PGHOST 环境变量（如果已设置），否则尝试使用 Unix 域套接字连接。

`-p port`
`--port=port`

指定服务器正在监听连接的 TCP 端口，或本地 Unix 域套接字文件扩展名。默认值取自 `PGPORT` 环境变量（如果已设置），否则使用编译时默认值。

`-U username`
`--username=username`

用于连接的用户名。

`-w`
`--no-password`

绝不发出密码提示。如果服务器要求密码认证，而又无法通过 `.pgpass` 文件等其他方式获得密码，则连接尝试将失败。在没有用户在场输入密码的批处理作业和脚本中，此选项很有用。

`-W`
`--password`

强制 pg_dump 在连接数据库之前提示输入密码。

这个选项绝非必需，因为如果服务器要求密码认证，pg_dump 会自动提示输入密码。不过，pg_dump 会先浪费一次连接尝试来发现服务器需要密码。在某些情况下值得输入 `-W`，因为可以避免这次额外的连接尝试。

`--role=rolename`

指定用于创建转储的角色名称。此选项会让 pg_dump 在连接数据库后发出 `SET ROLE rolename` 命令。当经认证用户（由 `-U` 指定）缺少 pg_dump 所需权限，但可以切换到具有所需权限的角色时，此选项很有用。有些安装环境有禁止直接以超级用户登录的策略，使用此选项就可以在不违反该策略的情况下进行转储。

环境

`PGDATABASE`
`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

默认连接参数。

与大多数其他 PostgreSQL 工具一样，此工具也使用 libpq 支持的环境变量（见第 34.14 节）。

诊断

pg_dump 在内部执行 `SELECT` 语句。如果运行 pg_dump 时遇到问题，请确认能够例如使用 psql 从数据库中查询信息。此外，libpq 前端库所使用的任何默认连接设置和环境变量也都会生效。

pg_dump 的数据库活动通常会被统计收集器收集。如果不希望如此，可以将参数 `track_counts` 通过 `PGOPTIONS` 或 `ALTER USER` 命令设置为 false。

注解

如果数据库集簇在 template1 数据库中有任何本地添加的内容，要小心把 pg_dump 的输出恢复到一个真正空的数据库中；否则很可能因为这些新增对象的重复定义而报错。要创建一个不含任何本地添加的空数据库，应从 template0 而不是 template1 复制，例如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

当选择仅数据转储并使用 --disable-triggers 选项时，pg_dump 会在插入数据前发出命令禁用用户表上的触发器，并在数据插入完成后发出命令重新启用它们。如果恢复在中途停止，系统目录可能会保持在错误状态。

pg_dump 生成的转储文件不包含优化器用于决定查询计划的统计信息。因此，从转储文件恢复后，最好运行ANALYZE 以确保最佳性能；更多信息请参阅第 24.1.3 节和第 24.1.6 节。

由于 pg_dump 常被用来把数据迁移到更新版本的 PostgreSQL，因此通常可以期望 pg_dump 的输出能够装载到 PostgreSQL 服务器中，而这些服务器的版本比 pg_dump 更高。pg_dump 也可以从版本比它自身更旧的 PostgreSQL 服务器上转储数据。（目前支持回溯到 8.0 版的服务器。）然而，pg_dump 不能从主版本高于它自身的 PostgreSQL 服务器上转储；它甚至会拒绝尝试，以免冒生成无效转储的风险。另外，也不能保证 pg_dump 的输出能够装载到更旧主版本的服务器上 – 即使转储正是从该版本服务器上取得的。把转储文件装载到较旧服务器时，可能需要手工编辑转储文件，移除旧服务器无法理解的语法。在跨版本场景中，建议使用 --quote-all-identifiers 选项，因为它可以防止不同 PostgreSQL 版本保留字列表差异带来的问题。

在转储逻辑复制订阅时，pg_dump 会生成 CREATE SUBSCRIPTION 命令，并带上 connect = false 选项，这样恢复订阅时就不会为了创建复制槽或执行初始表复制而建立远程连接。这样一来，无需网络访问远程服务器也能恢复转储。随后由用户以适当方式重新激活订阅。如果涉及的主机已经变化，则可能需要修改连接信息。在开始一次新的完整表复制之前，可能也适合先截断目标表。

示例

要将名为 mydb 的数据库转储到 SQL 脚本文件中：

```
$ pg_dump mydb > db.sql
```

要把这样的脚本重新装入一个（新创建的）名为 newdb 的数据库：

```
$ psql -X -d newdb -f db.sql
```

要把一个数据库转储为 custom 格式归档文件：

```
$ pg_dump -Fc mydb > db.dump
```

要把一个数据库转储为 directory 格式归档：

```
$ pg_dump -Fd mydb -f dumpdir
```

要使用 5 个并行工作任务把一个数据库转储为 directory 格式归档：

```
$ pg_dump -Fd mydb -j 5 -f dumpdir
```

要把一个归档文件重新装入到一个（新创建的）名为 newdb 的数据库：

```
$ pg_restore -d newdb db.dump
```

要把一个归档文件重新装入生成该归档的同一个数据库，并丢弃该数据库的当前内容：

```
$ pg_restore -d postgres --clean --create db.dump
```

要转储一个名为 mytab 的表：

```
$ pg_dump -t mytab mydb > db.sql
```

要转储 detroit 模式中名称以 emp 开头的所有表，但排除名为 employee_log 的表：

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

要转储名称以 east 或 west 开头、并且以 gsm 结尾的所有模式，同时排除名称中包含单词 test 的任何模式：

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

同样，使用正则表达式记法来合并这些开关：

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

要转储除名称以 ts_ 开头的表之外的所有数据库对象：

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

在 -t 及相关开关中，如果要指定一个大写或混合大小写的名称，就需要对该名称加双引号；否则它会被折叠成小写（见“模式”一节）。但双引号对 shell 来说又是特殊字符，因此反过来还必须再被引用。因此，要转储一个具有混合大小写名称的单个表，需要类似下面这样：

```
$ pg_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

另见

pg_dumpall, pg_restore, psql

pg_dumpall

pg_dumpall — 将一个 PostgreSQL 数据库集簇转储为脚本文件

大纲

pg_dumpall [*connection-option*...] [*option*...]

描述

pg_dumpall 是一个实用程序，用于写出（“转储”）一个集簇中的所有 PostgreSQL 数据库到一个脚本文件中。该脚本文件包含 SQL 命令，可作为psql 的输入来恢复这些数据库。它通过对集簇中的每个数据库调用 pg_dump 来实现这一点。pg_dumpall 还会转储所有数据库共有的全局对象，即数据库角色、表空间。（pg_dump 不会保存这些对象。）

由于 pg_dumpall 会读取所有数据库中的表，因此通常必须以数据库超级用户身份连接，才能生成完整的转储。另外，要执行保存的脚本，也需要具备超级用户权限，这样才能添加角色并创建数据库。

SQL 脚本将写入标准输出。使用 -f/--file 选项或 shell 操作符将其重定向到文件。

pg_dumpall 需要多次连接到 PostgreSQL 服务器（每个数据库一次）。如果使用密码认证，它每次都会提示输入密码。在这种情况下准备一个~/.pgpass 文件会比较方便。详见第 34.15 节。

选项

以下命令行选项控制输出的内容和格式。

-a
--data-only

只转储数据，不转储模式（数据定义）。

-c
--clean

在重新创建之前，输出用于DROP 所有已转储数据库、角色和表空间的 SQL 命令。当恢复需要覆盖现有集簇时，此选项很有用。如果目标集簇中不存在某些对象，则恢复期间会报告可忽略的错误消息，除非还指定了--if-exists。

-E *encoding*
--encoding=*encoding*

在指定的字符集编码中创建转储。默认情况下，转储将以数据库编码创建。（获得相同结果的另一种方法是将PGCLIENTENCODING 环境变量设置为所需的转储编码。）

-f *filename*
--file=*filename*

将输出发送到指定文件。如果省略此选项，则使用标准输出。

`-g`
`--globals-only`

只转储全局对象（角色和表空间），不转储数据库。

`-o`
`--oids`

将对象标识符（OID）作为每个表的数据的一部分进行转储。如果应用程序以某种方式引用 OID 列（例如在外键约束中），请使用此选项。否则，不应使用此选项。

`-O`
`--no-owner`

不要输出用于将对象所有权设置为与原始数据库一致的命令。默认情况下，pg_dumpall 会发出 `ALTER OWNER` 或 `SET SESSION AUTHORIZATION` 语句，以设置已创建模式元素的所有权。除非该脚本由超级用户（或拥有脚本中全部对象的同一用户）启动，否则这些语句在运行时会失败。若要创建一个可由任意用户恢复、并让该用户获得所有对象所有权的脚本，请指定`-O`。

`-r`
`--roles-only`

只转储角色，不转储数据库或表空间。

`-s`
`--schema-only`

只转储对象定义（模式），不转储数据。

`-S` *`username`*
`--superuser=`*`username`*

指定在禁用触发器时要使用的超级用户名。这只在使用 `--disable-triggers` 时相关。（通常最好省略它，而是以超级用户身份运行生成的脚本。）

`-t`
`--tablespaces-only`

只转储表空间，不转储数据库或角色。

`-v`
`--verbose`

指定详细模式。这会让 pg_dumpall 把开始/停止时间写入转储文件，并将进度消息输出到标准错误。它还会启用pg_dump 的详细输出。

`-V`
`--version`

打印 pg_dumpall 的版本并退出。

`-x`
`--no-privileges`
`--no-acl`

不转储访问权限（GRANT/REVOKE 命令）。

`--binary-upgrade`

此选项供就地升级实用程序使用。不建议也不支持将其用于其他目的。该选项的行为可能在未来的版本中更改而不另行通知。

`--column-inserts`
`--attribute-inserts`

将数据转储为带有显式列名的 `INSERT` 命令（`INSERT INTO` *`table`* `(`*`column`*`, ...) VALUES ...`）。这会使恢复变得非常缓慢；它主要用于生成可以装入非 PostgreSQL 数据库的转储。

`--disable-dollar-quoting`

此选项禁用对函数体的美元引用，并强制用 SQL 标准字符串语法对它们加引号。

`--disable-triggers`

此选项只在创建仅数据转储时才相关。它指示 pg_dumpall 在输出中包含一些命令，以便在恢复数据时临时禁用目标表上的触发器。如果这些表上存在不希望在数据恢复期间触发的引用完整性检查或其他触发器，请使用此选项。

目前，为 `--disable-triggers` 输出的这些命令必须由超级用户执行。因此，还应通过 `-S` 指定一个超级用户名，或者更好的做法是谨慎地以超级用户身份运行生成的脚本。

`--if-exists`

使用 `DROP ... IF EXISTS` 命令在 `--clean` 模式下删除对象。这会抑制原本可能报告的 “does not exist” 错误信息。除非同时指定了 `--clean`，否则此选项无效。

`--inserts`

将数据转储为 `INSERT` 命令（而不是 `COPY`）。这会使恢复非常缓慢；它主要用于生成可以装入非 PostgreSQL 数据库的转储。注意，如果重新安排了列顺序，恢复可能会彻底失败。`--column-inserts` 选项可以避免列顺序变化带来的问题，但速度更慢。

`--load-via-partition-root`

当为表分区转储数据时，让`COPY` 或 `INSERT` 语句以包含该分区的分区层次结构根为目标，而不是以该分区本身为目标。这会导致在加载数据时为每一行重新确定适当的分区。当在另一台服务器上恢复数据，而行并不总是落在与原始服务器相同的分区中时，这可能很有用。例如，如果分区列是 text 类型，而两个系统对用于排序该分区列的排序规则定义不同，就可能发生这种情况。

`--lock-wait-timeout=`*`timeout`*

在转储开始时，不要无限等待获取共享表锁。如果无法在指定的*`timeout`* 内锁定某个表，就让转储失败。超时可以用`SET statement_timeout` 接受的任意格式指定。允许的值因被转储的服务器版本而异，但从 7.3 起，所有版本都接受以毫秒为单位的整数。对 7.3 之前的服务器进行转储时，此选项会被忽略。

--no-comments

不转储注释。

--no-publications

不要转储发布。

--no-role-passwords

不要转储角色密码。恢复后，角色的密码值将为 null，在设置密码之前，密码认证始终会失败。由于指定此选项时不需要密码值，角色信息将从目录视图pg_roles 而不是 pg_authid 读取。因此，如果某些安全策略限制了对pg_authid 的访问，此选项也会有帮助。

--no-security-labels

不要转储安全标签。

--no-subscriptions

不要转储订阅。

--no-sync

默认情况下，pg_dumpall 会等待所有文件都被安全写入磁盘。该选项会使 pg_dumpall 立即返回而不等待，这样更快，但意味着随后如果操作系统崩溃，转储可能会损坏。通常，此选项适用于测试，但不应在从生产安装转储数据时使用。

--no-tablespaces

不输出创建表空间的命令，也不输出用于为对象选择表空间的命令。使用此选项时，所有对象在恢复时都会创建在当时默认的表空间中。

--no-unlogged-table-data

不要转储不记录 WAL 的表的内容。此选项不影响是否转储表定义（模式）；它只会抑制转储表数据。

--quote-all-identifiers

强制为所有标识符加引号。当从其 PostgreSQL 主版本与 pg_dumpall 不同的服务器转储数据库时，或者当输出打算装入另一主版本服务器中时，建议使用此选项。默认情况下，pg_dumpall 只会给在其自身主版本中属于保留字的标识符加引号。处理其他版本服务器时，这有时会带来兼容性问题，因为它们的保留字集合可能略有不同。使用--quote-all-identifiers 可以避免这类问题，但代价是转储脚本更难阅读。

--use-set-session-authorization

输出符合 SQL 标准的 SET SESSION AUTHORIZATION 命令，而不是用ALTER OWNER 命令来确定对象所有权。这会让转储更符合标准，但根据转储中对象的历史，可能无法正确恢复。

-?
--help

显示关于 pg_dumpall 命令行参数的帮助信息，并退出。

以下命令行选项控制数据库连接参数。

-d *connstr*
--dbname=*connstr*

以连接字符串形式指定用于连接服务器的参数；这些参数将覆盖任何冲突的命令行选项。

该选项名为--dbname，是为了与其他客户端应用程序保持一致；但由于pg_dumpall 需要连接多个数据库，连接字符串中的数据库名将被忽略。请使用-l 选项指定初始连接所用的数据库名，该连接将用于转储全局对象并发现还应转储哪些数据库。

-h *host*
--host=*host*

指定运行数据库服务器的机器的主机名。如果该值以斜杠开头，则将其用作 Unix 域套接字的目录。默认值取自PGHOST 环境变量（如果已设置）；否则会尝试使用 Unix 域套接字连接。

-l *dbname*
--database=*dbname*

指定用于转储全局对象并发现还应转储哪些数据库的连接数据库名。如果未指定，则使用 postgres 数据库；如果该数据库不存在，则使用template1。

-p *port*
--port=*port*

指定服务器正在监听连接的 TCP 端口，或本地 Unix 域套接字文件扩展名。默认值取自 PGPORT 环境变量（如果已设置），否则使用编译时默认值。

-U *username*
--username=*username*

用于连接的用户名。

-w
--no-password

绝不发出密码提示。如果服务器要求密码认证，而又无法通过 .pgpass 文件等其他方式获得密码，则连接尝试将失败。在没有用户在场输入密码的批处理作业和脚本中，此选项很有用。

-W
--password

强制 pg_dumpall 在连接数据库之前提示输入密码。

此选项从来都不是必需的，因为如果服务器要求密码认证，pg_dumpall 会自动提示输入密码。不过，pg_dumpall 会浪费一次连接尝试来发现服务器需要密码。在某些情况下，键入 -W 以避免这次额外的连接尝试是值得的。

注意，对每个要转储的数据库都会再次提示输入密码。通常，最好设置一个~/.pgpass 文件，而不是依赖手工输入密码。

--role=*rolename*

指定一个角色名，用于创建转储。此选项会使pg_dumpall 在连接数据库后发出 SET ROLE *rolename* 命令。当已认证用户（由 -U 指定）缺少 pg_dumpall 所需权限，但可以切换到具备所需权限的角色时，这很有用。有些安装环境不允许直接以超级用户身份登录，而使用此选项可以在不违反该策略的情况下完成转储。

环境

PGHOST
PGOPTIONS
PGPORT
PGUSER

默认连接参数

与大多数其他 PostgreSQL 工具一样，此工具也使用 libpq 支持的环境变量（见第 34.14 节）。

注解

由于 pg_dumpall 在内部调用 pg_dump，因此某些诊断消息会提到pg_dump。

即使目标是将转储脚本恢复到一个全新的集簇中，--clean 选项也可能很有用。使用 --clean 后，脚本可以删除并重新创建内置的postgres 和 template1 数据库，从而确保这些数据库保留与源集簇中相同的属性（例如区域设置和编码）。如果不使用此选项，这些数据库将保留其现有的数据库级属性，以及任何预先存在的内容。

恢复完成后，最好对每个数据库运行ANALYZE，以便优化器获得有用的统计信息。也可以运行 vacuumdb -a -z 来分析所有数据库。

不应指望该转储脚本在执行时完全不出现错误。特别是，由于脚本会为源集簇中存在的每个角色发出CREATE ROLE，因此除非目标集簇使用不同的引导超级用户名初始化，否则一定会遇到针对引导超级用户的 “role already exists” 错误。这个错误是无害的，应当忽略。使用--clean 还可能产生一些关于不存在对象的额外无害错误消息，不过可以通过添加--if-exists 来尽量减少这些消息。

pg_dumpall 要求在恢复之前，所有必需的表空间目录都已经存在；否则，位于非默认位置的数据库在创建时将会失败。

示例

要转储所有数据库：

```
$ pg_dumpall > db.out
```

要从此文件恢复数据库，可以使用：

```
$ psql -f db.out postgres
```

这里连接哪个数据库并不重要，因为 pg_dumpall 创建的脚本文件会包含适当的命令，用于创建并连接到已保存的数据库。例外是，如果指定了 `--clean`，就必须首先连接到 `postgres` 数据库；脚本会立即尝试删除其他数据库，而对于当前连接的数据库，这种操作会失败。

另见

有关可能出现的错误情况，请参见 pg_dump。

pg_isready

pg_isready — 检查 PostgreSQL 服务器的连接状态

大纲

`pg_isready` [*connection-option*...] [*option*...]

描述

pg_isready 是一个用于检查 PostgreSQL 数据库服务器连接状态的工具。其退出状态指示连接检查的结果。

选项

`-d` *dbname*
`--dbname=`*dbname*

指定要连接的数据库名称。*dbname* 也可以是连接字符串。如果是这样，连接字符串中的参数将覆盖任何冲突的命令行选项。

`-h` *hostname*
`--host=`*hostname*

指定服务器所在机器的主机名。如果值以斜杠开头，则被用作 Unix 域套接字的目录。

`-p` *port*
`--port=`*port*

指定服务器正在监听连接的 TCP 端口，或本地 Unix 域套接字文件扩展名。默认值取自 `PGPORT` 环境变量；如果未设置，则取编译时指定的端口，通常为 5432。

`-q`
`--quiet`

不显示状态消息。这在编写脚本时很有用。

`-t` *seconds*
`--timeout=`*seconds*

在尝试连接时，返回服务器无响应之前等待的最大秒数。设为 0 表示禁用。默认值为 3 秒。

`-U` *username*
`--username=`*username*

以用户 *username* 连接到数据库，而不是使用默认用户。

`-V`
`--version`

打印 pg_isready 版本并退出。

-?
--help

显示有关 pg_isready 命令行参数的帮助并退出。

退出状态

如果服务器正常接受连接，pg_isready 会向 shell 返回 0；如果服务器拒绝连接（例如处于启动阶段），则返回1；如果连接尝试未得到响应，则返回 2；如果未发起连接尝试（例如由于参数非法），则返回3。

环境

与大多数其他 PostgreSQL 工具一样，pg_isready 也使用 libpq 支持的环境变量（见第 34.14 节）。

注解

要获得服务器状态，无需提供正确的用户名、密码或数据库名。不过，如果提供了不正确的值，服务器会记录一次失败的连接尝试。

示例

标准用法：

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

使用连接参数检查一个处于启动阶段的 PostgreSQL 集簇：

```
$ pg_isready -h localhost -p 5433
localhost:5433 - rejecting connections
$ echo $?
1
```

使用连接参数检查一个无响应的 PostgreSQL 集簇：

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

pg_receivewal

pg_receivewal — 从 PostgreSQL 服务器流式接收预写式日志

大纲

`pg_receivewal` [*option*...]

描述

pg_receivewal 用于从正在运行的 PostgreSQL 数据库集簇流式接收预写式日志。预写式日志通过流复制协议传输，并写入本地目录中的文件。该目录可作为使用时间点恢复执行恢复时的归档位置（见第 25.3 节）。

pg_receivewal 会在服务器生成预写式日志的同时实时流式接收，而不像archive_command 那样等待 WAL 段完成。因此，使用pg_receivewal 时无需设置 archive_timeout。

与 PostgreSQL 备库的 WAL 接收进程不同，pg_receivewal 默认只会在一个 WAL 文件关闭时才将 WAL 数据刷写到磁盘。要实时刷写 WAL 数据，必须指定`--synchronous` 选项。由于 pg_receivewal 不会应用 WAL，因此当synchronous_commit 等于 `remote_apply` 时，不应让它成为同步备库。如果这样做，它看起来会像一台永远追不上的备库，并导致事务提交阻塞。为避免这种情况，应当为synchronous_standby_names 配置合适的值，或者为 pg_receivewal 指定一个与之不匹配的 `application_name`，或者将`synchronous_commit` 改成 `remote_apply` 以外的值。

预写式日志通过普通的 PostgreSQL 连接进行流式传输，并使用复制协议。建立该连接时，必须使用具有`REPLICATION` 权限（见第 21.2 节）的用户或超级用户，并且`pg_hba.conf` 必须允许复制连接。服务器还必须将 max_wal_senders 设置得足够高，以便至少为该流保留一个可用会话。

预写式日志流传输的起点会在 pg_receivewal 启动时计算：

1. 首先，扫描写入 WAL 段文件的目录，找到最新的已完成段文件，并将下一个 WAL 段文件的开头作为起始位置。
2. 如果无法使用上述方法计算起点，就使用服务器通过`IDENTIFY_SYSTEM` 命令报告的最新 WAL 刷盘位置。

如果因非致命错误导致连接中断或初次连接无法建立，pg_receivewal 将无限期重试该连接，并尽快重新建立流式传输。要避免这种行为，请使用`-n` 参数。

在没有致命错误的情况下，pg_receivewal 会一直运行，直到收到SIGINT（**Control+C**）信号为止。

选项

`-D` *directory*
`--directory=`*directory*

用于写入输出的目录。

这个参数是必需的。

`-E lsn`
`--endpos=lsn`

当接收位置到达指定 LSN 时，自动停止复制并以正常退出状态 0 退出。

如果存在 LSN 恰好等于 *lsn* 的记录，该记录也会被处理。

`--if-not-exists`

当指定 `--create-slot` 且同名槽已存在时，不报错。

`-n`
`--no-loop`

遇到连接错误时不要循环重试，而是立即报错退出。

`--no-sync`

此选项会使 `pg_receivewal` 不强制将 WAL 数据刷写到磁盘。这样会更快，但也意味着如果随后发生操作系统崩溃，可能会留下损坏的 WAL 段。一般来说，此选项适合测试，但不应在生产环境中执行 WAL 归档时使用。

这个选项与`--synchronous` 不兼容。

`-s interval`
`--status-interval=interval`

指定向服务器回送状态包的时间间隔（秒）。这样更便于从服务器端监视进度。值为零将完全禁用周期性状态更新，不过在服务器请求时仍会发送更新，以避免因超时而断开连接。默认值是 10 秒。

`-S slotname`
`--slot=slotname`

要求 pg_receivewal 使用一个现有复制槽（见第 26.2.6 节）。使用此选项时，pg_receivewal 会向服务器报告刷写位置，指示每个段何时已同步到磁盘，以便服务器在该段不再被需要时将其移除。

当服务器将 pg_receivewal 的复制客户端配置为同步备库时，使用复制槽会向服务器报告刷写位置，但只会在一个 WAL 文件关闭时才报告。因此，这种配置会导致主库上的事务长时间等待，实际上无法令人满意地工作。要让这种配置正确工作，还必须另外指定 `--synchronous` 选项（见下文）。

`--synchronous`

在收到 WAL 数据后立即将其刷写到磁盘。无论`--status-interval` 如何，刷写后还会立即向服务器回送状态包。

如果服务器将 pg_receivewal 的复制客户端配置为同步备库，则应指定此选项，以确保及时向服务器发送反馈。

-v
--verbose

启用详细输出模式。

-Z *level*
--compress=*level*

启用预写式日志的 gzip 压缩，并指定压缩级别（0 到 9，0 表示不压缩，9 表示最佳压缩）。所有文件名都会自动添加.gz 后缀。

以下命令行选项控制数据库连接参数：

-d *connstr*
--dbname=*connstr*

以连接字符串的形式指定用于连接到服务器的参数；这些参数会覆盖任何冲突的命令行选项。

为了与其他客户端应用保持一致，该选项被称为--dbname，但由于pg_receivewal 并不连接到集簇中的任何特定数据库，连接字符串中的数据库名会被忽略。

-h *host*
--host=*host*

指定运行服务器的主机名。如果该值以斜杠开头，则被用作 Unix 域套接字的目录。默认值取自PGHOST 环境变量（如果已设置），否则尝试 Unix 域套接字连接。

-p *port*
--port=*port*

指定服务器正在监听连接的 TCP 端口，或本地 Unix 域套接字文件扩展名。默认值取自 PGPORT 环境变量（如果已设置），否则使用编译时的默认值。

-U *username*
--username=*username*

用于连接的用户名。

-w
--no-password

绝不提示输入密码。如果服务器要求密码认证，而又无法通过.pgpass 文件等其他方式获得密码，则连接尝试将会失败。此选项可用于批处理作业和脚本，因为这些场景中没有用户在场输入密码。

-W
--password

强制 pg_receivewal 在建立连接前提示输入密码。

这个选项从来都不是必需的，因为如果服务器要求密码认证，pg_receivewal 会自动提示输入密码。不过，pg_receivewal 需要先浪费一次连接尝试，才能发现服务器需要密码。在某些情况下，输入-W 以避免这次额外的连接尝试是值得的。

为了控制物理复制槽，pg_receivewal 可以执行下列两种操作之一：

`--create-slot`

使用 `--slot` 中指定的名称创建一个新的物理复制槽，然后退出。

`--drop-slot`

删除 `--slot` 中指定名称的复制槽，然后退出。

还可以使用下列其他选项：

`-V`
`--version`

打印 pg_receivewal 的版本并退出。

`-?`
`--help`

显示 pg_receivewal 命令行参数的帮助并退出。

退出状态

当因 SIGINT 信号终止时，pg_receivewal 将以状态 0 退出。（这是其正常结束方式，因此不算错误。）对于致命错误或其他信号，退出状态将为非零值。

环境

与大多数其他 PostgreSQL 工具一样，此工具也使用libpq 支持的环境变量（见第 34.14 节）。

注解

当使用 pg_receivewal 而不是 archive_command 作为主要 WAL 备份方法时，强烈建议使用复制槽。否则，服务器可以在这些预写式日志文件备份完成之前自由回收或删除它们，因为无论是 archive_command 还是复制槽，都不会向服务器提供 WAL 流已经归档到什么位置的信息。不过请注意，如果接收端跟不上获取 WAL 数据的速度，复制槽会占满服务器的磁盘空间。

如果源集簇启用了组权限，pg_receivewal 会在接收到的 WAL 文件上保留这些组权限。

示例

要从位于 `mydbserver` 的服务器流式接收预写式日志，并将其存储到本地目录`/usr/local/pgsql/archive`：

```
$ pg_receivewal -h mydbserver -D /usr/local/pgsql/archive
```

另见

pg_basebackup

pg_recvlogical

pg_recvlogical — 控制 PostgreSQL 逻辑解码流

大纲

`pg_recvlogical` [*option*...]

描述

`pg_recvlogical` 用于控制逻辑解码复制槽，并从这类复制槽流式传输数据。

它会创建复制模式连接，因此除受到与pg_receivewal 相同的约束外，还要满足逻辑复制的相关约束（见第 49 章）。

`pg_recvlogical` 没有与逻辑解码 SQL 接口中 peek 和 get 模式等价的功能。它会在接收数据期间惰性地为这些数据发送回放确认，并在正常退出时发送确认。若要在不消费数据的情况下检查某个槽上的待处理数据，请使用`pg_logical_slot_peek_changes`。

选项

必须至少指定以下选项之一来选择操作：

`--create-slot`

为由 `--dbname` 指定的数据库，使用`--plugin` 指定的输出插件，创建一个名称由`--slot` 指定的新逻辑复制槽。

`--drop-slot`

删除由 `--slot` 指定名称的复制槽，然后退出。

`--start`

开始从由 `--slot` 指定的逻辑复制槽流式传输更改，并持续运行直到被信号终止。如果服务端的更改流因服务器关闭或断开连接而结束，则除非指定了`--no-loop`，否则会循环重试。

流格式由创建该槽时指定的输出插件决定。

该连接必须连到创建该槽时所用的同一个数据库。

`--create-slot` 和 `--start` 可以一同指定。`--drop-slot` 不能与其他操作组合使用。

以下命令行选项控制输出的位置和格式以及其他复制行为：

`-E` *`lsn`*
`--endpos=`*`lsn`*

在 `--start` 模式下，当接收达到指定的 LSN 时，自动停止复制，并以正常退出状态 0 退出。如果在非`--start` 模式下指定，则会报错。

如果存在 LSN 恰好等于 *`lsn`* 的记录，则该记录会被输出。

--endpos 选项不会感知事务边界，因此可能会在事务中途截断输出。任何只输出了一部分的事务都不会被消费，并会在下次从该槽读取时再次回放。单条消息永远不会被截断。

-f *filename*
--file=*filename*

将接收到并解码后的事务数据写入该文件。指定-表示写入 stdout。

-F *interval_seconds*
--fsync-interval=*interval_seconds*

指定 pg_recvlogical 应当以多高的频率发起 fsync() 调用，以确保输出文件安全刷盘。

服务器会偶尔要求客户端执行刷盘，并将刷盘位置报告给服务器。除此之外，此设置还会更频繁地执行刷盘。

将间隔指定为 0 会完全禁用 fsync() 调用，但仍会向服务器报告进度。在这种情况下，发生崩溃时可能会丢失数据。

-I *lsn*
--startpos=*lsn*

在 --start 模式下，从给定的 LSN 开始复制。关于其影响的详细信息，请参见第 49 章和第 53.4 节中的说明。在其他模式下会忽略此选项。

--if-not-exists

当指定 --create-slot 且指定名称的槽已存在时，不报错。

-n
--no-loop

当与服务器的连接丢失时，不要循环重试，直接退出。

-o *name*[=*value*]
--option=*name*[=*value*]

将选项 *name* 传递给输出插件；如果指定了 *value*，则将其用作该选项的值。可用选项及其效果取决于所使用的输出插件。

-P *plugin*
--plugin=*plugin*

创建该槽时，使用指定的逻辑解码输出插件。参见第 49 章。如果该槽已经存在，则此选项不起作用。

-s *interval_seconds*
--status-interval=*interval_seconds*

该选项与 pg_receivewal 中同名选项的效果相同。参见该处说明。

-S *slot_name*
--slot=*slot_name*

在 --start 模式下，使用名为*slot_name* 的现有逻辑复制槽。在 --create-slot 模式下，以此名称创建该槽。在--drop-slot 模式下，删除此名称的槽。

-v
--verbose

启用详细模式。

以下命令行选项控制数据库连接参数：

-d *dbname*
--dbname=*dbname*

要连接的数据库。有关其具体含义，请参阅各操作的说明。*dbname* 可以是一个连接字符串。如果如此，连接字符串参数会覆盖任何与之冲突的命令行选项。默认值为用户名。

-h *hostname-or-ip*
--host=*hostname-or-ip*

指定服务器运行所在机器的主机名。如果该值以斜杠开头，则它会被用作 Unix 域套接字的目录。默认值取自 PGHOST 环境变量（如果已设置），否则将尝试 Unix 域套接字连接。

-p *port*
--port=*port*

指定服务器监听连接所用的 TCP 端口，或本地 Unix 域套接字文件扩展名。默认值取自 PGPORT 环境变量（如果已设置），否则使用编译时的默认值。

-U *user*
--username=*user*

用于连接的用户名。默认为当前操作系统用户名。

-w
--no-password

绝不提示输入密码。如果服务器要求密码认证，而密码又无法通过 .pgpass 文件等其他方式获得，则连接尝试将失败。此选项适用于批处理作业和脚本，因为在这些场景中没有用户在场输入密码。

-W
--password

强制 pg_recvlogical 在连接数据库之前提示输入密码。

这个选项并非必不可少，因为如果服务器要求密码认证，pg_recvlogical 会自动提示输入密码。不过，pg_recvlogical 需要先浪费一次连接尝试，才能发现服务器需要密码。在某些情况下，使用 -W 来避免这次额外的连接尝试是值得的。

还可以使用下列附加选项：

-V
--version

打印 pg_recvlogical 的版本并退出。

-?
--help

显示 pg_recvlogical 命令行参数的帮助并退出。

环境

与大多数其他 PostgreSQL 工具一样，该工具使用libpq 支持的环境变量（见第 34.14 节）。

注解

如果源集簇启用了组权限，pg_recvlogical 会在接收到的 WAL 文件上保留这些组权限。

示例

示例请参见第 49.1 节。

另见

pg_receivewal

pg_restore

pg_restore — 从由 pg_dump 创建的归档文件恢复 PostgreSQL 数据库

大纲

pg_restore [*connection-option*...] [*option*...] [*filename*]

描述

pg_restore 是一个用于从 pg_dump 以非纯文本格式创建的归档中恢复 PostgreSQL 数据库的工具。它会发出必要的命令，将数据库重建为保存时的状态。归档文件还允许 pg_restore 有选择地恢复其中的内容，甚至在恢复前重新排列各项的顺序。归档文件被设计为可跨体系结构移植。

pg_restore 可以以两种模式运行。如果指定了数据库名，pg_restore 就会连接到该数据库，并将归档内容直接恢复到数据库中。否则，它会创建一个脚本，其中包含重建数据库所需的 SQL 命令，并将其写入文件或标准输出。这种脚本输出等价于 pg_dump 的纯文本输出格式。因此，一些控制输出的选项与 pg_dump 的选项相对应。

显然，pg_restore 无法恢复归档文件中不存在的信息。例如，如果归档是使用 “将数据转储为 INSERT 命令” 选项生成的，pg_restore 就不能使用 COPY 语句装载数据。

选项

pg_restore 接受下列命令行参数。

filename

指定要恢复的归档文件所在位置（对于目录格式归档则为目录）。如果未指定，则使用标准输入。

-a
--data-only

仅恢复数据，不恢复模式（数据定义）。如果归档中包含，则会恢复表数据、大对象和序列值。

此选项类似于指定 --section=data，但出于历史原因，两者并不完全相同。

-c
--clean

在恢复数据库对象之前，发出 DROP 命令以删除将要恢复的所有对象。此选项适用于覆盖现有数据库。如果目标数据库中有任何对象不存在，则会报告可忽略的错误消息，除非同时指定了 --if-exists。

-C
--create

在向其中恢复之前先创建数据库。如果同时指定了 --clean，则会在连接到目标数据库之前先删除并重新创建它。

使用 --create 时，pg_restore 还会恢复数据库的注释（如果有）以及该数据库专有的任何配置变量设置，也就是任何提到该数据库的 ALTER DATABASE ... SET ... 和 ALTER ROLE ... IN DATABASE ... SET ... 命令。除非指定了 --no-acl，还会恢复数据库本身的访问权限。

使用此选项时，由 -d 指定的数据库仅用于发出初始的 DROP DATABASE 和 CREATE DATABASE 命令。所有数据都会恢复到以归档中记录的名称命名的数据库。

-d *dbname*
--dbname=*dbname*

连接到数据库 *dbname*，并直接恢复到该数据库中。*dbname* 可以是一个连接字符串。如果是这样，连接字符串参数会覆盖任何冲突的命令行选项。

-e
--exit-on-error

如果在向数据库发送 SQL 命令时遇到错误，则退出。默认行为是继续执行，并在恢复结束时显示错误计数。

-f *filename*
--file=*filename*

指定生成脚本的输出文件，或者在与-l 一起使用时指定列表输出文件。使用-表示标准输出，这也是默认值。

-F *format*
--format=*format*

指定归档格式。不必指定格式，因为 pg_restore 会自动确定格式。如果指定，可以是以下格式之一：

c
custom

归档采用 pg_dump 的自定义格式。

d
directory

归档是目录格式归档。

t
tar

归档是 tar 归档。

-I *index*
--index=*index*

仅恢复指定名称的索引的定义。可以通过多次指定 -I 来恢复多个索引。

-j *number-of-jobs*
--jobs=*number-of-jobs*

使用多个并发作业执行 pg_restore 中最耗时的步骤 – 装载数据、创建索引或创建约束 –。此选项可以显著缩短将大型数据库恢复到运行在多处理器机器上的服务器所需的时间。

每个作业都是一个进程或一个线程，具体取决于操作系统，并使用一个到服务器的独立连接。

此选项的最佳取值取决于服务器、客户端和网络的硬件配置。影响因素包括 CPU 核心数和磁盘配置。一个较好的起点是服务器上的 CPU 核心数，但在很多情况下，更大的值也可能带来更快的恢复速度。当然，取值过高会因为频繁争用而导致性能下降。

只有自定义格式和目录格式归档支持此选项。输入必须是常规文件或目录（例如不能是管道）。当生成脚本而不是直接连接到数据库服务器时，此选项会被忽略。另外，多个作业不能与--single-transaction 选项一起使用。

-l
--list

列出归档的目录内容（table of contents）。此操作的输出可用作 -L 选项的输入。请注意，如果在 -l 中使用 -n 或 -t 等过滤开关，它们会限制被列出的项目。

-L *list-file*
--use-list=*list-file*

仅恢复在 *list-file* 中列出的归档元素，并按它们在文件中出现的顺序进行恢复。请注意，如果在 -L 中使用 -n 或 -t 等过滤开关，它们还会进一步限制被恢复的项目。

list-file 通常是通过编辑先前 -l 操作的输出来创建的。可以移动或删除行，也可以通过在行首放置分号（;）将其注释掉。示例见下文。

-n *schema*
--schema=*schema*

仅恢复位于指定模式中的对象。可以通过多次指定 -n 来指定多个模式。这可以与 -t 选项结合使用，只恢复特定表。

-N *schema*
--exclude-schema=*schema*

不恢复位于指定模式中的对象。可以通过多次指定 -N 来排除多个模式。

当对同一模式名同时给出 -n 和 -N 时，-N 优先，该模式会被排除。

-O
--no-owner

不输出用于设置对象所有权以匹配原始数据库的命令。默认情况下，pg_restore 会发出 ALTER OWNER 或 SET SESSION AUTHORIZATION 语句，为所创建的模式元素设置所有权。除非对数据库的初始连接由超级用户发起（或者由脚本中所有对象的同一所有者发起），否则这些语句会失败。使用 -O 时，初始连接可以使用任意用户名，并且该用户将拥有所有创建的对象。

`-P` *function-name(argtype [, ...])*
`--function=`*function-name(argtype [, ...])*

仅恢复指定函数。请务必按转储文件目录中显示的形式准确拼写函数名和参数。可以通过多次指定 `-P` 来恢复多个函数。

`-R`
`--no-reconnect`

这个选项已经过时，但仍然被接受以保持向后兼容性。

`-s`
`--schema-only`

仅恢复模式（数据定义），不恢复数据，前提是归档中存在模式条目。

此选项与 `--data-only` 的作用相反。它类似于指定 `--section=pre-data --section=post-data`，但由于历史原因并不完全相同。

（不要把它与 `--schema` 选项混淆，后者中的“schema”一词含义不同。）

`-S` *username*
`--superuser=`*username*

指定在禁用触发器时要使用的超级用户用户名。仅当使用 `--disable-triggers` 时才相关。

`-t` *table*
`--table=`*table*

仅恢复指定表的定义和/或数据。这里的“表”包括视图、物化视图、序列和外部表。可以通过多次指定 `-t` 来选择多个表。此选项可以与 `-n` 选项结合使用，以指定特定模式中的表。

注意

当指定 `-t` 时，pg_restore 不会尝试恢复所选表可能依赖的任何其他数据库对象。因此，不能保证将特定表恢复到一个空数据库中一定会成功。

注意

此标志与 pg_dump 的 `-t` 标志并不完全相同。pg_restore 目前不支持通配符匹配，也不能在其 `-t` 中包含模式名。而且，虽然 pg_dump 的 `-t` 标志也会转储所选表的附属对象（例如索引），pg_restore 的 `-t` 标志并不包括这些附属对象。

注意

在 PostgreSQL 9.6 之前的版本中，此标志只匹配表，不匹配其他任何类型的关系。

-T *trigger*
--trigger=*trigger*

仅恢复指定名称的触发器。可以通过多次指定 -T 来恢复多个触发器。

-v
--verbose

指定详细模式。

-V
--version

打印 pg_restore 的版本并退出。

-x
--no-privileges
--no-acl

阻止恢复访问权限（grant/revoke 命令）。

-1
--single-transaction

将恢复作为单个事务执行（也就是把发出的命令包裹在 BEGIN/COMMIT 中）。这可确保要么所有命令都成功完成，要么不应用任何更改。此选项隐含 --exit-on-error。

--disable-triggers

此选项仅在执行仅数据恢复时才相关。它会指示 pg_restore 在恢复数据期间发出命令，临时禁用目标表上的触发器。如果你不希望在恢复数据期间触发表上的引用完整性检查或其他触发器，请使用此选项。

目前，为 --disable-triggers 发出的命令必须以超级用户身份执行。因此，你还应通过 -S 指定超级用户用户名，或者更好的是，直接以 PostgreSQL 超级用户身份运行 pg_restore。

--enable-row-security

此选项仅在恢复启用了行安全性的表内容时才相关。默认情况下，pg_restore 会将 row_security 设置为 off，以确保所有数据都能恢复到表中。如果用户没有足够的权限绕过行安全性，就会抛出错误。此参数会指示 pg_restore 改为将 row_security 设置为 on，从而允许用户尝试在启用行安全性的情况下恢复表内容。如果用户无权将转储中的行插入该表，这仍然可能失败。

请注意，此选项当前还要求转储采用 INSERT 格式，因为 COPY FROM 不支持行安全性。

--if-exists

使用 DROP ... IF EXISTS 命令在 --clean 模式下删除对象。这会抑制原本可能报告的 “does not exist” 错误信息。除非同时指定了 --clean，否则此选项无效。

--no-comments

不输出用于恢复注释的命令，即使归档中包含注释。

`--no-data-for-failed-tables`

默认情况下，即使表的创建命令失败（例如因为表已经存在），表数据仍然会被恢复。使用此选项时，这类表的数据会被跳过。如果目标数据库已经包含所需的表内容，这种行为会很有用。例如，PostgreSQL 扩展（如 PostGIS）的辅助表可能已经装载到目标数据库中；指定此选项可以防止向这些表装载重复或过时的数据。

此选项仅在直接恢复到数据库时有效，而在生成 SQL 脚本输出时无效。

`--no-publications`

不输出用于恢复发布的命令，即使归档中包含它们。

`--no-security-labels`

不输出用于恢复安全标签的命令，即使归档中包含它们。

`--no-subscriptions`

不输出用于恢复订阅的命令，即使归档中包含它们。

`--no-tablespaces`

不输出用于选择表空间的命令。使用此选项时，所有对象在恢复时都会创建在当时默认的表空间中。

`--section=`*`sectionname`*

仅恢复指定部分。部分名称可以是 `pre-data`、`data` 或 `post-data`。可以多次指定此选项来选择多个部分。默认是恢复所有部分。

数据部分包含实际的表数据以及大对象定义。post-data 部分由索引、触发器、规则以及除已验证检查约束之外的约束定义组成。pre-data 部分则由所有其他数据定义项组成。

`--strict-names`

要求每个模式限定符（`-n`/`--schema`）和表限定符（`-t`/`--table`）至少匹配待恢复文件中的一个模式/表。

`--use-set-session-authorization`

输出符合 SQL 标准的 `SET SESSION AUTHORIZATION` 命令，而不是使用 `ALTER OWNER` 命令来确定对象所有权。这样会使转储结果更符合标准，但根据转储中对象的历史，可能无法正确恢复。

`-?`
`--help`

显示关于 pg_restore 命令行参数的帮助信息，并退出。

pg_restore 还接受下列用于连接参数的命令行参数：

`-h` *`host`*
`--host=`*`host`*

指定服务器所在机器的主机名。如果值以斜杠开头，则被用作 Unix 域套接字的目录。默认值取自 `PGHOST` 环境变量（如果已设置），否则尝试使用 Unix 域套接字连接。

`-p port`
`--port=port`

指定服务器正在监听连接的 TCP 端口，或本地 Unix 域套接字文件扩展名。默认值取自 `PGPORT` 环境变量（如果已设置），否则使用编译时默认值。

`-U username`
`--username=username`

用于连接的用户名。

`-w`
`--no-password`

绝不发出密码提示。如果服务器要求密码认证，而又无法通过 `.pgpass` 文件等其他方式获得密码，则连接尝试将失败。在没有用户在场输入密码的批处理作业和脚本中，此选项很有用。

`-W`
`--password`

强制 pg_restore 在连接数据库之前提示输入密码。

此选项实际上并非必需，因为如果服务器要求密码认证，pg_restore 会自动提示输入密码。不过，pg_restore 会浪费一次连接尝试来发现服务器需要密码。在某些情况下，键入 `-W` 以避免这一次额外的连接尝试是值得的。

`--role=rolename`

指定执行恢复时要使用的角色名。此选项会使 pg_restore 在连接数据库后发出 `SET ROLE rolename` 命令。当已认证用户（由 `-U` 指定）缺少 pg_restore 所需的权限，但可以切换到具有所需权限的角色时，此选项会很有用。有些安装环境禁止直接以超级用户登录，而使用此选项可以在不违反该策略的情况下执行恢复。

环境

`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

默认连接参数

与大多数其他 PostgreSQL 工具一样，该工具也使用 libpq 支持的环境变量（见第 34.14 节）。但是，当未提供数据库名时，它不会读取 `PGDATABASE`。

诊断

当使用 `-d` 选项指定直接数据库连接时，pg_restore 会在内部执行 SQL 语句。如果运行 pg_restore 时遇到问题，请确保你能够使用例如psql 从该数据库中查询信息。此外，libpq 前端库使用的任何默认连接设置和环境变量也都会生效。

注解

如果你的安装在 template1 数据库中有任何本地添加内容，请务必将 pg_restore 的输出装载到一个真正空的数据库中；否则很可能会因为这些附加对象的重复定义而报错。要创建一个不带任何本地添加内容的空数据库，应从 template0 而不是 template1 复制，例如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

pg_restore 的局限性详述如下。

- 当把数据恢复到一个预先存在的表中，并且使用了 --disable-triggers 选项时，pg_restore 会在插入数据前发出命令，禁用用户表上的触发器，然后在数据插入后再发出命令重新启用它们。如果恢复在中途停止，系统目录可能会处于错误状态。
- pg_restore 不能有选择地恢复大对象；例如，不能只恢复属于某个特定表的大对象。如果归档中包含大对象，那么要么所有大对象都会被恢复，要么在通过 -L、-t 或其他选项排除它们时一个也不会恢复。

关于 pg_dump 的局限性的细节也可参见 pg_dump 文档。

恢复完成后，最好对每个恢复的表运行ANALYZE，以便优化器获得有用的统计信息；更多信息请参阅第 24.1.3 节和第 24.1.6 节。

示例

假设我们已经把一个名为 mydb 的数据库转储到一个自定义格式转储文件中：

```
$ pg_dump -Fc mydb > db.dump
```

要删除该数据库并从转储中重新创建它：

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

-d 选项中指定的数据库可以是集簇中任何一个已存在的数据库；pg_restore 只用它来为 mydb 发出 CREATE DATABASE 命令。使用 -C 时，数据总是恢复到以转储文件中记录的名称命名的数据库。

要把该转储恢复到一个名为 newdb 的新数据库中：

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

注意，这里我们不使用 -C，而是直接连接到要恢复到的数据库。还要注意，我们是从 template0 而不是 template1 克隆这个新数据库，以确保它一开始是空的。

要重新排列数据库项的顺序，首先需要转储归档的目录：

```
$ pg_restore -l db.dump > db.list
```

列表文件由一个头部和每个项各占一行的内容组成，例如：

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

分号表示一条注释的开始，而行首的数字表示分配给每个项的内部归档 ID。

文件中的行可以被注释掉、删除并重新排序。例如：

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

可以将这样的文件作为 pg_restore 的输入，这样它就只会按该顺序恢复项 10 和 6：

```
$ pg_restore -L db.list db.dump
```

另见

pg_dump, pg_dumpall, psql

psql

psql — PostgreSQL 的交互式终端

大纲

psql [*option*...] [*dbname* [*username*]]

描述

psql 是 PostgreSQL 的一个基于终端的前端。它使你能够交互式地输入查询，将其发送给 PostgreSQL，并查看查询结果。也可以从文件或命令行参数提供输入。此外，psql 还提供了若干元命令和多种类似 shell 的特性，以便于编写脚本和自动化执行各种任务。

选项

`-a`
`--echo-all`

在读入时将所有非空输入行打印到标准输出（不适用于交互式行读取）。这等效于把变量 `ECHO` 设置为 `all`。

`-A`
`--no-align`

切换到非对齐输出模式（默认输出模式是对齐的）。这等效于`\pset format unaligned`。

`-b`
`--echo-errors`

把失败的 SQL 命令打印到标准错误输出。这等效于把变量 `ECHO` 设置为 `errors`。

`-c` *command*
`--command=`*command*

指定 psql 执行一个给定的命令字符串 *command*。这个选项可以重复多次并且以任何顺序与 `-f` 选项组合在一起。当 `-c` 或者 `-f` 被指定时，psql 不会从标准输入读取命令，而是在按顺序处理完所有`-c` 和 `-f` 选项后终止。

command 必须是一个服务器完全可解析的命令字符串（即不包含 psql 专有的特性）或者单个反斜线命令。因此不能在一个 -c 选项中混合 SQL 和 psql 元命令。要那样做，可以使用多个-c 选项或者把字符串用管道输送到 psql 中，例如：

```
psql -c '\x' -c 'SELECT * FROM foo;'
```

或者

```
echo '\x \\ SELECT * FROM foo;' | psql
```

（\\ 是分隔符元命令）。

每个 SQL 命令字符串传递给 -c 都作为一个单独的请求发送到服务器。因此，即使字符串包含多个SQL 命令，服务器也会将其作为单个事务执行，除非字符串中包含显式的BEGIN /COMMIT 命令将其分成多个事务。（有关服务器如何处理多查询字符串的更多详细信息，请参见第 53.2.2.1 节。）此外，psql 只打印字符串中最后一条 SQL 命令的结果。这与从文件读取同一字符串或将其送入 psql 标准输入时的行为不同，因为在这些情况下，psql 会分别发送每条 SQL 命令。

由于这种行为，在单个 -c 字符串中放入多条 SQL 命令常常会产生意外结果。最好重复使用 -c 命令，或将多条命令送入 psql 的标准输入，可以像上面所示使用 echo，也可以使用 shell 的 here-document，例如：

```
psql <<EOF
\x
SELECT * FROM foo;
EOF
```

-d *dbname*
--dbname=*dbname*

指定要连接的数据库的名称。这等效于指定 *dbname* 为命令行上的第一个非选项参数。*dbname* 可以是连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

-e
--echo-queries

也把发送到服务器的所有 SQL 命令复制到标准输出。这等效于把变量 ECHO 设置为 queries。

-E
--echo-hidden

回显\d 以及其他反斜线命令生成的实际查询。可以用它来学习 psql 的内部操作。这等效于把变量 ECHO_HIDDEN 设置为 on。

-f *filename*
--file=*filename*

从文件 *filename* 而不是标准输入中读取命令。这个选项可以重复指定，也可以按任意顺序与-c 选项组合使用。当指定了 -c 或 -f 时，psql 不会从标准输入读取命令，而是在按顺序处理完所有-c 和 -f 选项后终止。除此之外，这个选项在很大程度上等价于元命令\i。

如果 *filename* 是-（连字符），则会读取标准输入，直到遇到 EOF 指示或\q 元命令。这可用于将交互式输入与文件输入交错使用。不过请注意，这种情况下不会使用 Readline（很像指定了 -n 时的情况）。

使用这个选项与写成 psql < *filename* 有细微差别。通常两种形式都会得到你期望的结果，但使用-f 可以启用一些有用的特性，例如带行号的错误消息。使用这个选项也还有一点机会降低启动开销。另一方面，使用 shell 输入重定向的形式在理论上能保证得到与你手工逐行输入时完全相同的输出。

-F *separator*
--field-separator=*separator*

使用 *separator* 作为非对齐输出的字段分隔符。这等效于\pset fieldsep 或者\f。

-h *hostname*
--host=*hostname*

指定运行服务器的机器的主机名。如果该值以斜线开头，则它会被用作 Unix 域套接字所在的目录。

-H
--html

切换到 HTML 表格输出模式。这等效于\pset format html 或者\H 命令。

-l
--list

列出所有可用的数据库，然后退出。其他非连接选项会被忽略。这与元命令\list 类似。

使用该选项时，psql 将连接到数据库 postgres，除非命令行上指定了其他数据库（通过-d 选项或非选项参数，也可能通过服务项，但不能通过环境变量）。

-L *filename*
--log-file=*filename*

除了把所有查询输出写到普通输出目标之外，还写到文件*filename* 中。

-n
--no-readline

不要使用 Readline 进行行编辑，也不要使用命令历史记录。这有助于在剪切和粘贴时关闭 TAB 补全。

-o *filename*
--output=*filename*

把所有查询输出放到文件 *filename* 中。这等效于命令\o。

-p *port*
--port=*port*

指定服务器用于监听连接的 TCP 端口或者本地 Unix 域套接字文件扩展名。默认是 PGPORT 环境变量的值，如果没有设置，则默认为编译时指定的端口号（通常是 5432）。

`-P` *assignment*
`--pset=`*assignment*

以`\pset`的形式指定打印选项。注意，这里你必须用一个等号而不是空格来分隔名称和值。例如，要设置输出格式为LaTeX，应该写成`-P format=latex`。

`-q`
`--quiet`

指定 psql 应该安静地工作。默认情况下，它会打印出欢迎消息和各种提示信息。如果使用了这个选项，以上那些就都不会输出。在使用`-c`选项时，配合这个选项很有用。这等效于设置变量`QUIET`为`on`。

`-R` *separator*
`--record-separator=`*separator*

把 *separator* 用作非对齐输出的记录分隔符。这等效于`\pset recordsep`命令。

`-s`
`--single-step`

运行在单步模式中。这意味着在每个命令被发送给服务器之前都会提示用户，并允许取消执行。使用这个选项可以调试脚本。

`-S`
`--single-line`

运行在单行模式中，其中换行符会终止一个 SQL 命令，就像分号的作用一样。

> **注意**
>
> 这种模式是为坚持使用它的用户提供的，但并不一定值得推荐。特别是，如果在一行中混合了SQL 和元命令，对于没有经验的用户来说，它们的执行顺序未必总是清楚的。

`-t`
`--tuples-only`

关闭打印列名和结果行计数页脚等。这等效于`\t`或者`\pset tuples_only`命令。

`-T` *table_options*
`--table-attr=`*table_options*

指定要放在 HTML `table`标签内的选项。详见`\pset tableattr`。

`-U` *username*
`--username=`*username*

作为用户 *username* 而不是默认用户连接到数据库（当然，你必须具有这样做的权限）。

-v *assignment*
--set=*assignment*
--variable=*assignment*

执行一次变量赋值，和\set 元命令相似。注意你必须在命令行上用等号分隔名字和值（如果有）。要取消变量的设置，去掉等号就行。要把一个变量设为空字符串，使用等号但是去掉值。这些赋值在命令行处理期间被完成，因此反映连接状态的变量将在稍后被覆盖。

-V
--version

打印 psql 版本并且退出。

-w
--no-password

永不发出密码提示。如果服务器要求密码认证，而密码又无法从其他来源获得，例如 .pgpass 文件，则连接尝试将失败。这个选项对于批处理任务和脚本很有用，因为那时通常没有用户在场输入密码。

请注意，这个选项在整个会话期间都会保持生效，因此它不仅影响初始连接尝试，也会影响元命令\connect。

-W
--password

强制 psql 在连接数据库之前提示输入密码，即使该密码实际上不会被使用。

如果服务器要求密码认证，而密码又无法从其他来源获得，例如.pgpass 文件，则psql 无论如何都会提示输入密码。不过，psql 需要先浪费一次连接尝试，才能知道服务器要求密码。在某些情况下，显式指定-W 值得用来避免这次额外的连接尝试。

请注意，这个选项在整个会话期间都会保持生效，因此它不仅影响初始连接尝试，也会影响元命令\connect。

-x
--expanded

打开扩展表格式模式。这等效于\x 或者\pset expanded 命令。

-X,
--no-psqlrc

不读取启动文件（既不读取系统范围的 psqlrc 文件，也不读取用户的~/.psqlrc 文件）。

-z
--field-separator-zero

设置非对齐输出的字段分隔符为零字节。这等效于\pset fieldsep_zero。

-0
--record-separator-zero

设置非对齐输出的记录分隔符为零字节。例如，这有助于与xargs -0 配合使用。这等效于\pset recordsep_zero。

-1
--single-transaction

这个选项只能与一个或多个 -c 和/或 -f 选项结合使用。它会导致 psql 在第一个这样的选项之前发出一个 BEGIN 命令，并在最后一个选项之后发出一个COMMIT 命令，从而将所有命令包装成一个单独的事务。这确保要么所有命令都成功完成，要么不应用任何更改。

如果命令本身包含 BEGIN、COMMIT 或 ROLLBACK，这个选项就不会产生期望的效果。另外，如果某个单独命令不能在事务块中执行，指定这个选项将导致整个事务失败。

-?
--help[=*topic*]

显示有关 psql 的帮助并且退出。可选的 *topic* 参数（默认为 options）选择要解释 psql 的哪一部分：commands 描述 psql 的反斜线命令；options 描述可以被传递给 psql 的命令行选项；而 variables 则显示有关 psql 配置变量的帮助。

退出状态

如果 psql 正常结束，它会向 shell 返回 0；如果它自身发生致命错误（例如内存耗尽、找不到文件），则返回 1；如果到服务器的连接发生故障且该会话不是交互式的，则返回 2；如果脚本中发生错误且变量ON_ERROR_STOP 已设置，则返回 3。

用法

连接到数据库

psql 是常规的 PostgreSQL 客户端应用程序。要连接到数据库，你需要知道目标数据库名称、服务器的主机名和端口号，以及要以哪个数据库用户名连接。psql 可以通过命令行选项 -d、-h、-p 和 -U 分别指定这些参数。如果遇到一个不属于任何选项的参数，它将被解释为数据库名（如果数据库名已经给出，则解释为数据库用户名）。并非所有这些选项都是必需的；它们都有有用的默认值。如果省略主机名，psql 将通过 Unix 域套接字连接到本地主机上的服务器，而在没有 Unix 域套接字的机器上则通过 TCP/IP 连接到 localhost。默认端口号在编译时确定。由于数据库服务器使用相同的默认值，因此在大多数情况下不必指定端口。默认用户名是你的操作系统用户名，默认数据库名也是如此。请注意，你不能随意以任意数据库用户名连接到任意数据库。数据库管理员应当已经告知你拥有的访问权限。

当默认值不完全合适时，可以通过把环境变量PGDATABASE、PGHOST、PGPORT 和 PGUSER 设置为适当的值来少敲一些键盘（额外的环境变量见第 34.14 节）。另外，准备一个~/.pgpass 文件也很方便，这样就不必经常手工输入密码。详见第 34.15 节。

指定连接参数的另一种方法是使用一个 *conninfo* 字符串或一个 URI，它可用来替代数据库名。这种机制可以让你对连接进行非常细致的控制。例如：

```
$ psql "service=myservice sslmode=require"
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

用这种方式，你也可以把LDAP 用于第 34.17 节中描述的连接参数查找。可用连接选项的更多信息请见第 34.1.2 节。

如果由于任何原因（例如权限不足、服务器没有在目标主机上运行等）导致连接无法建立，psql 将返回一个错误并且终止。

如果标准输入和标准输出都是终端，那么psql 会把客户端编码设置为 “auto”，从区域设置中检测合适的客户端编码（在 Unix 系统上是LC_CTYPE 环境变量）。如果结果不符合预期，可以使用环境变量PGCLIENTENCODING 覆盖客户端编码。

输入 SQL 命令

在正常操作时，psql 会提供一个提示符，该提示符是psql 当前连接到的数据库名称后面跟上字符串=>。例如：

```
$ psql testdb
psql (11.22)
Type "help" for help.

testdb=>
```

在提示符下，用户可以输入SQL 命令。通常，当遇到表示命令结束的分号时，输入的内容就会被发送给服务器。行结束并不会终止一条命令，因此为了提高清晰度，命令可以分布在多行上。如果命令被发送并成功执行，其结果就会显示在屏幕上。

如果不受信任的用户能够访问尚未采用模式的安全使用方式的数据库，请在会话开始时从search_path 中移除所有公众可写的模式。可以在连接字符串中加入options=-csearch_path=，或者在执行其他 SQL 命令之前发出SELECT pg_catalog.set_config('search_path', '', false)。这种考虑并非 psql 特有；它适用于任何执行任意 SQL 命令的接口。

每当执行命令时，psql 也会轮询由 LISTEN 和 NOTIFY 生成的异步通知事件。

虽然 C 风格的块注释会被传给服务器处理并移除，但 SQL 标准注释会由psql 自行移除。

元命令

你输入到 psql 中的任何以未加引号的反斜线开始的东西都是一个 psql 元命令，它们由psql 自行处理。这些命令让 psql 对管理和编写脚本更有用。元命令常常被称作斜线或者反斜线命令。

psql 命令的格式是用反斜线后面直接跟上一个命令动词，然后是一些参数。参数与命令动词和其他参数之间用任意多个空白字符分隔开。

要在参数中包含空白，可以用单引号将它括起来。要在参数中包含一个单引号，可以在单引号文本内写两个单引号。单引号中的内容还会接受类似 C 语言的替换：\n（换行）、\t（制表符）、\b（退格）、\r（回车）、\f（换页）、*digits*（八进制）以及\x*digits*（十六进制）。在单引号文本内，反斜线出现在任何其他字符前面时，只是对该单个字符进行转义，不论它是什么字符。

如果在参数中出现一个未加引号的冒号（:），后面紧跟一个psql 变量名，它就会被该变量的值替换，如下面SQL 插值所述。其中介绍的:'*variable_name*' 和:"*variable_name*" 形式也同样适用。:{?*variable_name*}语法允许测试变量是否已定义；它会被 TRUE 或 FALSE 替换。用反斜线转义该冒号可以防止替换发生。

在参数中，用反引号（`）包围的文本会被视为传给 shell 的命令行。该命令的输出（去掉末尾的换行符）会替换反引号中的文本。在反引号包围的文本内部，不会进行特殊的引号处理或其他处理，但出现:*variable_name* 时，如果*variable_name* 是 psql 变量名，就会被替换为该变量的值。此外，:'*variable_name*' 也会被替换为该变量的值，并会适当地加上引号，使其成为单个 shell 命令参数。（后一种形式几乎总是更可取，除非你非常确定变量中包含什么。）由

于无法保证在所有平台上都能对回车和换行字符安全地加引号，当变量值中出现这类字符时，:'*variable_name*' 形式会打印错误消息，并且不会替换变量值。

有些命令把 SQL 标识符（例如表名）作为参数。这些参数遵循 SQL 的语法规则：未加引号的字母会被强制转换为小写，而双引号（"）可以保护字母不发生大小写转换，并允许在标识符中包含空白。在双引号内，成对的双引号会在结果名称中折叠成一个双引号。例如，FOO"BAR"BAZ 会被解释为 fooBARbaz，而"A weird"" name" 会变成 A weird" name。

参数解析会在行尾或遇到另一个未加引号的反斜线时停止。未加引号的反斜线会被视为新元命令的开始。特殊序列\\（两个反斜线）表示参数结束，并继续解析SQL 命令（如果还有）。通过这种方式，SQL 命令和 psql 命令可以自由地混合在同一行中。但无论如何，元命令的参数都不能延续到下一行。

许多元命令作用于当前查询缓冲区。这只是一个保存已输入但尚未发送到服务器执行的 SQL 命令文本的缓冲区。其中既包括先前输入的行，也包括同一行上位于元命令之前的文本。

定义了以下元命令：

\a

如果当前表格输出格式是非对齐，则切换为对齐；否则切换为非对齐。保留此命令是为了向后兼容。更通用的解决方案请参见\pset。

\c 或 \connect [-reuse-previous=*on*|*off*] [*dbname* [*username*] [*host*] [*port*] | *conninfo*]

建立到 PostgreSQL 服务器的新连接。连接参数既可以使用按位置指定的语法（数据库名、用户、主机和端口中的一个或多个），也可以使用*conninfo* 连接字符串，详见第 34.1.1 节。如果没有给出参数，则使用与之前相同的参数建立新连接。

将 *dbname*、*username*、*host* 或 *port* 中的任何一个指定为-，都等价于省略该参数。

新连接可以重用前一个连接的连接参数；不仅包括数据库名称、用户、主机和端口，还包括其他设置，如*sslmode*。默认情况下，参数在位置语法中被重用，但在给定*conninfo* 字符串时不会被重用。传递-reuse-previous=on 或-reuse-previous=off 作为第一个参数将覆盖该默认设置。如果参数被重用，则任何未明确指定为位置参数或在*conninfo* 字符串中的参数将从现有连接的参数中获取。一个例外是，如果使用位置语法更改*host* 设置，使其不同于先前的值，则现有连接参数中存在的任何*hostaddr* 设置将被删除。此外，仅当用户、主机和端口设置未更改时，才会重用现有连接使用的任何密码。当命令既不指定也不重用特定参数时，将使用libpq 的默认值。

如果成功建立了新连接，则关闭先前的连接。如果连接尝试失败（用户名错误、访问被拒绝等），当psql 处于交互模式时，会保留先前的连接。但在执行非交互式脚本时，会立即报错并停止处理。这一区别一方面使用户能够方便地应对输入错误，另一方面也作为安全机制，防止脚本意外地作用于错误的数据库。

例如：

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10 sslmode=disable"
=> \c -reuse-previous=on sslmode=require    -- 仅更改 sslmode
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

\C [*title*]

设置作为查询结果打印的表的标题，或取消此类标题。该命令等价于\pset title *title*。（此命令的名称源自“caption”，因为它过去只用于设置HTML 表的标题。）

\cd [*directory*]

将当前工作目录更改为 *directory*。如果没有参数，则切换到当前用户的主目录。

> **提示**
>
> 要打印当前工作目录，请使用\! pwd。

\conninfo

输出当前数据库连接的信息。

\copy { *table* [(*column_list*)] | (*query*) } { from | to } { *'filename'* | program *'command'* | stdin | stdout | pstdin | pstdout } [[with] (*option* [, ...])]

执行前端（客户端）复制。该操作会运行一个 SQL COPY 命令，但并不是由服务器读取或写入指定文件，而是由psql 读取或写入文件，并在服务器与本地文件系统之间转送数据。这意味着文件可访问性和权限取决于本地用户，而不是服务器，也不需要 SQL 超级用户权限。

当指定 program 时，*command* 由 psql 执行，传给*command* 的数据或从其中读出的数据都会在服务器与客户端之间转送。再次强调，执行权限属于本地用户，而不是服务器，也不需要 SQL 超级用户权限。

对于 \copy ... from stdin，数据行会从发出该命令的同一输入源读取，直到读到仅包含 \. 的一行或流到达 EOF 为止。这个选项适合在 SQL 脚本文件中以内联方式填充表。对于 \copy ... to stdout，输出会发送到与psql 命令输出相同的位置，并且不会打印 COPY *count* 命令状态（因为这可能与数据行混淆）。若要读写 psql 的标准输入或输出，而不受当前命令来源或 \o 选项影响，请写成 from pstdin 或 to pstdout。

该命令的语法与 SQL COPY 命令类似。除数据源或目标之外，其他所有选项都与COPY 中的指定相同。因此，特殊的解析规则适用于\copy 元命令。与大多数其他元命令不同，整个剩余行始终被视为\copy 的参数，参数中不执行变量插值或反引号扩展。

> **提示**
>
> 另一种获得与\copy ... to 相同结果的方法是使用 SQL COPY ... TO STDOUT 命令，并以\g *filename* 或\g |*program* 结束。与\copy 不同，这种方法允许命令跨越多行；此外，可以使用变量插值和反引号扩展。

> **提示**
>
> 这些操作不如以文件或程序作为数据源或目标的 SQL COPY 命令高效，因为所有数据都必须通过客户端/服务器连接传输。对于大量数据，使用SQL 命令可能更合适。另

外，由于采用这种数据转送方式，CSV 模式下的\copy ... from 会错误地把单独占据一行的\.数据值当作输入结束标记。

\copyright

显示 PostgreSQL 的版权和分发条款。

\crosstabview [*colV* [*colH* [*colD* [*sortcolH*]]]]

执行当前查询缓冲区（与\g 类似），并以交叉表网格显示结果。查询必须返回至少三列。由 *colV* 标识的输出列成为纵向表头，由 *colH* 标识的输出列成为横向表头。*colD* 标识要在网格中显示的输出列。*sortcolH* 标识横向表头的可选排序列。

每个列指定都可以是列号（从 1 开始）或列名。通常的 SQL 大小写折叠和加引号规则适用于列名。如果省略，*colV* 取第 1 列，*colH* 取第 2 列。*colH* 必须不同于 *colV*。如果未指定 *colD*，查询结果必须恰好有三列，既不是 *colV* 也不是 *colH* 的那一列被用作 *colD*。

纵向表头显示为最左列，包含 *colV* 列中的值，其顺序与查询结果中相同，但会移除重复值。

横向表头显示为第一行，包含 *colH* 列中的值，并移除重复值。默认情况下，它们按查询结果中的相同顺序显示。但如果给出了可选的 *sortcolH* 参数，它所标识的列的值必须是整数，而 *colH* 中的值会按照对应的 *sortcolH* 值排序后显示在横向表头中。

在交叉表网格中，对于 *colH* 中的每个不同值 x 和 *colV* 中的每个不同值 y，交点 (x,y) 处的单元格包含查询结果中 colD 列的值，该结果行的 *colH* 值为 x，*colV* 值为 y。如果没有这样的行，单元格为空。如果存在多条这样的行，则报错。

\d[S+] [*pattern*]

对于每个匹配 *pattern* 的关系（表、视图、物化视图、索引、序列或外部表）或复合类型，显示所有列及其类型、表空间（如果不是默认表空间），以及NOT NULL 或默认值等特殊属性。还会显示关联的索引、约束、规则和触发器。对于外部表，还会显示关联的外部服务器。（“匹配模式”的定义见下面的模式。）

对于某些关系类型，\d 会为每列显示附加信息：序列的列值、索引的索引表达式，以及外部表的外部数据包装器选项。

\d+形式的命令与之相同，但会显示更多信息：表各列关联的注释、表中是否存在 OID、视图定义（如果关系是视图），以及非默认的复制标识设置。

默认只显示用户创建的对象；提供模式或 S 修饰符可包含系统对象。

注意

如果使用\d 而没有 *pattern* 参数，它等同于\dtvmsE，它将显示所有可见的表、视图、物化视图、序列和外部表的列表。这纯粹是一种便利措施。

\da[S] [*pattern*]

列出聚合函数，以及它们的返回类型和所作用的数据类型。如果指定了 *pattern*，则只显示名称与模式匹配的聚合。默认情况下，只显示用户创建的对象；提供模式或 S 修饰符可包括系统对象。

```
\dA[+] [ pattern ]
```

列出访问方法。如果指定了 *pattern*，则只显示名称与模式匹配的访问方法。如果在命令名后追加 +，则每个访问方法都会连同其关联的处理器函数和描述一起列出。

```
\db[+] [ pattern ]
```

列出表空间。如果指定了 *pattern*，则只显示名称与模式匹配的表空间。如果在命令名后追加 +，则每个表空间都会连同其关联选项、磁盘大小、权限和描述一起列出。

```
\dc[S+] [ pattern ]
```

列出字符集编码之间的转换。如果指定了 *pattern*，则只列出名称与模式匹配的转换。默认情况下，只显示用户创建的对象；提供模式或 S 修饰符可包括系统对象。如果在命令名后追加 +，则每个对象都会连同其相关描述一起列出。

```
\dC[+] [ pattern ]
```

列出类型转换。如果指定了 *pattern*，则只列出源类型或目标类型匹配该模式的类型转换。如果在命令名后附加+，还会列出每个对象关联的描述。

```
\dd[S] [ pattern ]
```

显示类型为 constraint、operator class、operator family、rule 和 trigger 的对象的描述。其他所有注释都可以通过相应对象类型的反斜线命令查看。

\dd 显示匹配 *pattern* 的对象的描述；如果没有给出参数，则显示适当类型的可见对象的描述。但无论哪种情况，都只列出有描述的对象。默认只显示用户创建的对象；提供模式或S 修饰符可包含系统对象。

可以使用 COMMENT SQL 命令创建对象的描述。

```
\dD[S+] [ pattern ]
```

列出域。如果指定了 *pattern*，则只显示名称与模式匹配的域。默认情况下，只显示用户创建的对象；提供模式或 S 修饰符可包括系统对象。如果在命令名后追加 +，则每个对象都会连同其相关权限和描述一起列出。

```
\ddp [ pattern ]
```

列出默认访问权限设置。对于默认权限设置已偏离内置默认值的每个角色（以及模式，如果适用），都会显示一个条目。如果指定了*pattern*，则只列出角色名或模式名匹配该模式的条目。

ALTER DEFAULT PRIVILEGES 命令用于设置默认访问权限。权限显示的含义在 GRANT 中有解释。

```
\dE[S+] [ pattern ]
\di[S+] [ pattern ]
\dm[S+] [ pattern ]
\ds[S+] [ pattern ]
\dt[S+] [ pattern ]
\dv[S+] [ pattern ]
```

在这组命令中，字母E、i、m、s、t 和 v 分别代表外部表、索引、物化视图、序列、表和视图。可以按任意顺序指定这些字母中的任意一个或全部，以获取相应类型的对象列表。例

如，\dit 列出索引和表。如果在命令名后附加+，还会列出每个对象的磁盘上的物理大小以及关联的描述（如果有）。如果指定了*pattern*，则只列出名称匹配该模式的对象。默认只显示用户创建的对象；提供模式或S 修饰符可包含系统对象。

\des[+] [*pattern*]

列出外部服务器（助记词：“external servers”）。如果指定了 *pattern*，则只列出名称匹配该模式的服务器。如果使用\des+形式，则显示每个服务器的完整说明，包括服务器的ACL、类型、版本、选项和描述。

\det[+] [*pattern*]

列出外部表（助记词：“external tables”）。如果指定了 *pattern*，则只列出表名或模式名匹配该模式的条目。如果使用\det+形式，还会显示通用选项和外部表描述。

\deu[+] [*pattern*]

列出用户映射（助记词：“external users”）。如果指定了 *pattern*，则只列出用户名匹配该模式的映射。如果使用\deu+形式，还会显示每个映射的附加信息。

> **小心**
>
> \deu+还可能显示远程用户的用户名和密码，因此应注意不要泄露它们。

\dew[+] [*pattern*]

列出外部数据包装器（助记词：“external wrappers”）。如果指定了 *pattern*，则只列出名称匹配该模式的外部数据包装器。如果使用\dew+形式，还会显示外部数据包装器的ACL、选项和描述。

\df[anptwS+] [*pattern*]

列出函数及其结果数据类型、参数数据类型和函数类型。函数类型分为“agg”（聚合）、“normal”、“procedure”、“trigger”或“window”。要仅显示特定类型的函数，可在命令中添加对应的字母a、n、p、t 或 w。如果指定了 *pattern*，则只显示名称匹配该模式的函数。默认只显示用户创建的对象；提供模式或S 修饰符可包含系统对象。如果使用\df+形式，还会显示每个函数的附加信息，包括易变性、并行安全性、拥有者、安全分类、访问权限、语言、源代码和描述。

> **提示**
>
> 要查找接受特定数据类型参数或返回特定数据类型值的函数，请使用分页器的搜索功能浏览\df 输出。

\dF[+] [*pattern*]

列出全文检索配置。如果指定了 *pattern*，则只显示名称匹配该模式的配置。如果使用\dF+形式，则显示每个配置的完整说明，包括底层全文检索解析器和每种解析器词元类型的词典列表。

\dFd[+] [*pattern*]

列出全文检索词典。如果指定了 *pattern*，则只显示名称匹配该模式的词典。如果使用 \dFd+形式，还会显示每个选中词典的附加信息，包括底层全文检索模板和选项值。

\dFp[+] [*pattern*]

列出全文检索解析器。如果指定了 *pattern*，则只显示名称匹配该模式的解析器。如果使用 \dFp+形式，则显示每个解析器的完整说明，包括底层函数和可识别的词元类型列表。

\dFt[+] [*pattern*]

列出全文检索模板。如果指定了 *pattern*，则只显示名称匹配该模式的模板。如果使用 \dFt+形式，还会显示每个模板的附加信息，包括底层函数名。

\dg[S+] [*pattern*]

列出数据库角色。（由于“用户”和“组”的概念已经统一为“角色”，此命令现在等价于\du。）默认只显示用户创建的角色；提供 S 修饰符可包含系统角色。如果指定了 *pattern*，则只列出名称匹配该模式的角色。如果使用\dg+形式，还会显示每个角色的附加信息；目前会增加每个角色的注释。

\dl

这是\lo_list 的别名，用于显示大对象列表。

\dL[S+] [*pattern*]

列出过程语言。如果指定了 *pattern*，则只列出名称与模式匹配的语言。默认情况下，只显示用户创建的语言；提供 S 修饰符可包括系统对象。如果在命令名后追加 +，则每种语言都会连同其调用处理器、验证器、访问权限以及它是否为系统对象一起列出。

\dn[S+] [*pattern*]

列出模式（命名空间）。如果指定了 *pattern*，则只列出名称与模式匹配的模式。默认情况下，只显示用户创建的对象；提供模式或 S 修饰符可包括系统对象。如果在命令名后追加 +，则每个对象都会连同其相关权限和描述（如果有）一起列出。

\do[S+] [*pattern*]

列出操作符及其操作数类型和结果类型。如果指定了 *pattern*，则只列出名称匹配该模式的操作符。默认只显示用户创建的对象；提供模式或S 修饰符可包含系统对象。如果在命令名后附加+，还会显示每个操作符的附加信息，目前只包括底层函数名。

\dO[S+] [*pattern*]

列出排序规则。如果指定了 *pattern*，则只列出名称与模式匹配的排序规则。默认情况下，只显示用户创建的对象；提供模式或 S 修饰符可包括系统对象。如果在命令名后追加 +，则每个排序规则都会连同其相关描述（如果有）一起列出。请注意，只会显示与当前数据库编码兼容的排序规则，因此在同一安装中的不同数据库里结果可能不同。

\dp [*pattern*]

列出表、视图和序列及其关联的访问权限。如果指定了*pattern*，则只列出名称匹配该模式的表、视图和序列。

GRANT 和 REVOKE 命令用于设置访问权限。权限显示的含义在 GRANT 中有解释。

\drds [*role-pattern* [*database-pattern*]]

列出已定义的配置设置。这些设置可以特定于角色、特定于数据库，或同时特定于两者。*role-pattern* 和 *database-pattern* 分别用于选择要列出的角色和数据库。省略某个模式参数或将其指定为* 时，不会按该参数筛选，还会分别包含不特定于角色或不特定于数据库的设置。

ALTER ROLE 和 ALTER DATABASE 命令用于定义角色专属和数据库专属的配置设置。

\dRp[+] [*pattern*]

列出复制发布。如果指定了 *pattern*，则只列出名称匹配该模式的发布。如果在命令名后附加+，还会显示与每个发布关联的表。

\dRs[+] [*pattern*]

列出逻辑复制订阅。如果指定了 *pattern*，则只列出名称与模式匹配的订阅。如果在命令名后追加 +，则会显示订阅的其他属性。

\dT[S+] [*pattern*]

列出数据类型。如果指定了 *pattern*，则只列出名称与模式匹配的类型。如果在命令名后追加 +，则每个类型都会连同其内部名称、大小以及相关权限一起列出；对于 enum 类型，还会显示其允许值。默认情况下，只显示用户创建的对象；提供模式或 S 修饰符可包括系统对象。

\du[S+] [*pattern*]

列出数据库角色。（由于“用户”和“组”的概念已经统一为“角色”，此命令现在等价于\dg。）默认只显示用户创建的角色；提供 S 修饰符可包含系统角色。如果指定了 *pattern*，则只列出名称匹配该模式的角色。如果使用\du+形式，还会显示每个角色的附加信息；目前会增加每个角色的注释。

\dx[+] [*pattern*]

列出已安装的扩展。如果指定了 *pattern*，则只列出名称匹配该模式的扩展。如果使用 \dx+形式，则列出属于每个匹配扩展的所有对象。

\dy[+] [*pattern*]

列出事件触发器。如果指定了 *pattern*，则只列出名称与模式匹配的事件触发器。如果在命令名后追加 +，则每个对象都会连同其相关描述一起列出。

\e 或 \edit [*filename*] [*line_number*]

如果指定了 *filename*，则编辑该文件；编辑器退出后，将文件内容复制到当前查询缓冲区。如果未给出*filename*，则将当前查询缓冲区复制到临时文件，再以相同方式编辑。如果当前查询缓冲区为空，则将最近执行的查询复制到临时文件，并以相同方式编辑。

随后，按照psql 的正常规则重新解析查询缓冲区的新内容，将整个缓冲区视为一行。任何完整的查询都会立即执行；也就是说，如果查询缓冲区包含分号或以分号结束，就会执行到该位置为止的所有内容。剩余内容会在查询缓冲区中等待；键入分号或\g 发送它，或键入\r 清空查询缓冲区以取消它。将缓冲区视为一行主要影响元命令：缓冲区中位于元命令之后的

所有内容都会被视为该元命令的参数，即使它们跨越多行也是如此。（因此，不能用这种方式编写使用元命令的脚本。应使用\i 来处理这类脚本。）

如果指定了行号，psql 会将光标定位到文件或查询缓冲区中的指定行。请注意，如果只给出一个全由数字组成的参数，psql 会假定它是行号，而不是文件名。

> **提示**
>
> 有关如何配置和定制编辑器，请参见下面的环境。

\echo *text* [...]

将参数打印到标准输出，参数之间用一个空格分隔，末尾跟随换行符。这可用于在脚本输出中插入信息。例如：

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

如果第一个参数是未加引号的-n，则不输出末尾的换行符。

> **提示**
>
> 如果使用\o 命令重定向查询输出，可能会想用\qecho 代替这个命令。

\ef [*function_description* [*line_number*]]

这个命令获取并编辑指定函数或过程的定义，形式为CREATE OR REPLACE FUNCTION 或 CREATE OR REPLACE PROCEDURE 命令。编辑方式与\edit 相同。编辑器退出后，更新后的命令会在查询缓冲区中等待；键入分号或\g 发送它，或用\r 取消。

目标函数可以只用名称指定，也可以同时给出名称和参数，例如foo(integer, text)。如果存在多个同名函数，就必须给出参数类型。

如果未指定函数，则会呈现一个空的CREATE FUNCTION 模板供编辑。

如果指定了行号，psql 会将光标定位到函数体中的指定行。（请注意，函数体通常并不从文件的第一行开始。）

与大多数其他元命令不同，整个行的剩余部分始终被视为\ef 的参数，参数中不进行变量插值或反引号扩展。

> **提示**
>
> 有关如何配置和定制编辑器，请参见下面的环境。

\encoding [*encoding*]

设置客户端字符集编码。没有参数时，此命令显示当前编码。

\errverbose

以最大的详细程度重复最近的服务器错误消息，就好像VERBOSITY 被设置为 verbose，SHOW_CONTEXT 被设置为 always 一样。

\ev [*view_name* [*line_number*]]

这个命令获取并编辑指定视图的定义，形式为CREATE OR REPLACE VIEW 命令。编辑方式与\edit 相同。编辑器退出后，更新后的命令会在查询缓冲区中等待；输入分号或\g 发送它，或用\r 取消。

如果未指定视图，则会呈现一个空的CREATE VIEW 模板供编辑。

如果指定了行号，psql 会将光标定位到视图定义中的指定行。

与大多数其他元命令不同，整个行的剩余部分始终被视为\ev 的参数，参数中不进行变量插值或反引号扩展。

\f [*string*]

设置非对齐查询输出的字段分隔符。默认值是竖线（|）。它等同于\pset fieldsep。

\g [*filename*]
\g [|*command*]

将当前查询缓冲区发送给服务器执行。如果给出参数，查询输出会写入指定文件或通过管道传递给给定的 shell 命令，而不是按通常方式显示。只有查询成功返回零个或更多元组时，才会向文件或命令写入内容；如果查询失败或 SQL 命令不返回数据，则不会写入。

如果当前查询缓冲区为空，则最近发送的查询将被重新执行。除此之外，没有任何参数的\g 基本上等同于一个分号。带有参数的\g 提供了一个“一次性”替代\o 命令的选择。

如果参数以|开头，则该行剩余的全部内容会被视为要执行的*command*，其中不会进行变量插值或反引号扩展。该行剩余的内容只会原样传递给 shell。

\gdesc

显示当前查询缓冲区中查询结果的描述（即列名和数据类型）。查询并不会实际执行；不过，如果其中存在语法错误，会按通常的方式报告。

如果当前查询缓冲区为空，则改为描述最近发送的查询。

\gexec

将当前查询缓冲区发送给服务器，然后把查询输出（如果有）的每一行的每一列视为要执行的 SQL 语句。例如，下面为每一列创建一个索引，目标表是 my_table：

```
=> SELECT format('create index on my_table(%I)', attname)
-> FROM pg_attribute
-> WHERE attrelid = 'my_table'::regclass AND attnum > 0
-> ORDER BY attnum
-> \gexec
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

生成的查询按照返回行的顺序执行；如果有多列，则在每行内从左到右执行。NULL 字段会被忽略。生成的查询按原样发送到服务器进行处理，因此不能是psql 元命令，也不能包含psql 变量引用。如果某个查询失败，仍会继续执行其余查询，除非设置了ON_ERROR_STOP。每个查询的执行都受 ECHO 处理的影响。（通常，在使用\gexec 时，适宜将ECHO 设为 all 或 queries。）查询日志、单步模式、计时及其他查询执行功能也适用于每个生成的查询。

如果当前查询缓冲区为空，则改为重新执行最近发送的查询。

\gset [*prefix*]

将当前查询缓冲区发送给服务器，并将查询输出存入 psql 变量（参见下面的变量）。要执行的查询必须恰好返回一行。该行的每一列分别存入一个变量，变量名与列名相同。例如：

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset
=> \echo :var1 :var2
hello 10
```

如果指定了 *prefix*，则会将该字符串加到查询的列名前面，以构成要使用的变量名：

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset result_
=> \echo :result_var1 :result_var2
hello 10
```

如果某一列的结果为 NULL，则取消设置对应的变量，而不是设置它。

如果查询失败或没有恰好返回一行，则不会更改任何变量。

如果当前查询缓冲区为空，则改为重新执行最近发送的查询。

\gx [*filename*]
\gx [|*command*]

\gx 等价于\g，但会对当前查询强制使用扩展输出模式。参见\x。

\h 或 \help [*command*]

给出指定 SQL 命令的语法帮助。如果未指定 *command*，则psql 将列出所有可用语法帮助的命令。如果 *command* 是星号（*），则显示所有SQL 命令的语法帮助。

与大多数其他元命令不同，整个行的剩余部分始终被视为\help 的参数，参数中不进行变量插值或反引号扩展。

> **注意**
>
> 为了简化输入，由多个单词组成的命令不需要加引号。因此，可以直接输入 **\help alter table**。

\H 或 \html

打开 HTML 查询输出格式。如果 HTML 格式已经打开，则切换回默认的对齐文本格式。此命令是为兼容性和便利性而保留的；设置其他输出选项的方法见\pset。

\i 或 \include *filename*

从文件 *filename* 中读取输入，并像在键盘上输入一样执行它。

如果 *filename* 是-（连字符），则从标准输入读取，直到遇到 EOF 指示或\q 元命令。这可用于将交互式输入与文件输入交错使用。请注意，只有在最外层启用了 Readline，此处才会使用 Readline 功能。

> **注意**
>
> 如果想在屏幕上看到被读入的各行，请将变量ECHO 设置为 all。

\if *expression*
\elif *expression*
\else
\endif

这组命令实现了可嵌套的条件块。条件块必须以\if 开始，以\endif 结束。中间可以包含任意数量的\elif 子句，后面还可以选择跟随一个\else 子句。普通查询和其他类型的反斜线命令可以（通常也会）出现在构成条件块的命令之间。

\if 和\elif 命令读取其参数，并将其作为布尔表达式求值。如果表达式的值为true，则正常继续处理；否则，跳过后续行，直到遇到匹配的\elif、\else 或\endif。一旦\if 或\elif 测试成功，同一块中后续\elif 命令的参数就不再求值，而是被视为假。只有前面所有匹配的\if 和\elif 测试都未成功时，才会处理\else 后面的行。

与其他反斜线命令的参数一样，\if 或\elif 命令的 *expression* 参数会经过变量插值和反引号扩展。随后，按开/关选项变量值的规则对结果求值。因此，有效值是以下值的不区分大小写且无歧义的匹配：true、false、1、0、on、off、yes、no。例如，t、T 和 tR 都会被视为 true。

不能正确求值为真或假的表达式会产生警告，并被视为假。

被跳过的行仍会正常解析，以识别查询和反斜线命令，但查询不会发送给服务器，条件命令（\if、\elif、\else、\endif）以外的反斜线命令会被忽略。对于条件命令，只检查嵌套是否合法。被跳过的行中的变量引用不会展开，也不会执行反引号扩展。

同一个条件块的所有反斜线命令必须出现在同一个源文件中。如果主输入文件或通过\include 引入的文件到达 EOF 时，仍有本地\if 块未关闭，psql 就会报错。

下面是一个示例：

```
-- 检查数据库中是否存在两条不同的记录，并将
-- 结果分别存入不同的 psql 变量
SELECT
    EXISTS(SELECT 1 FROM customer WHERE customer_id = 123) as is_customer,
    EXISTS(SELECT 1 FROM employee WHERE employee_id = 456) as is_employee
\gset
\if :is_customer
    SELECT * FROM customer WHERE customer_id = 123;
\elif :is_employee
    \echo 'is not a customer but is an employee'
    SELECT * FROM employee WHERE employee_id = 456;
\else
    \if yes
        \echo 'not a customer or employee'
    \else
        \echo 'this will never print'
    \endif
\endif
```

\ir 或 \include_relative *filename*

\ir 命令与\i 相似，但解析相对文件名的方式不同。在交互模式下执行时，这两个命令的行为相同。不过，在脚本中调用时，\ir 会相对于脚本所在的目录来解释文件名，而不是相对于当前工作目录。

\l[+] 或 \list[+] [*pattern*]

列出服务器中的数据库，并显示其名称、拥有者、字符集编码和访问权限。如果指定了 *pattern*，则只列出名称匹配该模式的数据库。如果在命令名后附加+，还会显示数据库大小、默认表空间和描述。（大小信息只对当前用户能够连接的数据库可用。）

\lo_export *loid filename*

从数据库中读取具有 OID *loid* 的大对象，并将其写入 *filename*。请注意，这与服务器函数 lo_export 略有不同，后者使用运行数据库服务器的用户的权限，并在服务器的文件系统上操作。

> **提示**
>
> 使用\lo_list 命令来查找大对象的 OID。

\lo_import *filename* [*comment*]

将文件存储到一个 PostgreSQL 大对象中。可选地，它将给定的注释与对象关联起来。例如：

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

响应表明大对象获得了对象 ID 152801，这个 ID 可以用来在将来访问新创建的大对象。为便于阅读，建议始终为每个对象关联一条便于人阅读的注释。查看 OID 和注释时，可以使用 \lo_list 命令。

请注意，此命令与服务器端的lo_import 略有不同，因为它作为本地用户在本地文件系统上操作，而不是服务器的用户和文件系统。

\lo_list

列出当前存储在数据库中的所有 PostgreSQL 大对象，以及为它们提供的注释。

\lo_unlink *loid*

从数据库中删除 OID 为 *loid* 的大对象。

> **提示**
>
> 使用\lo_list 命令来查找大对象的 OID。

\o 或 \out [*filename*]
\o 或 \out [|*command*]

将后续查询结果保存到文件 *filename*，或通过管道传给 shell 命令*command*。如果没有指定参数，查询输出将恢复为标准输出。

如果参数以|开头，则该行剩余的全部内容会被视为要执行的*command*，其中不会进行变量插值或反引号扩展。该行剩余的内容只会原样传递给 shell。

“查询结果”包括从数据库服务器获取的所有表格、命令响应和通知，以及查询数据库的各种反斜杠命令的输出（例如\d）；但不包括错误消息。

> **提示**
>
> 要在查询结果之间插入文本输出，使用\qecho。

\p 或 \print

将当前查询缓冲区打印到标准输出。如果当前查询缓冲区为空，则打印最近执行的查询。

\password [*username*]

更改指定用户（默认为当前用户）的密码。此命令提示输入新密码，对其进行加密，并将其作为ALTER ROLE 命令发送到服务器。这样可以确保新密码不会以明文形式出现在命令历史记录、服务器日志或其他地方。

\prompt [*text*] *name*

提示用户提供文本，将其赋值给变量*name*。可以指定一个可选的提示字符串 *text*。（对于多个单词的提示，用单引号括起文本。）

默认情况下，\prompt 使用终端进行输入和输出。然而，如果使用了 -f 命令行开关，\prompt 将使用标准输入和标准输出。

\pset [*option* [*value*]]

这个命令设置影响查询结果表输出的选项。*option* 指定要设置哪个选项。*value* 的含义取决于所选的选项。对于某些选项，省略*value* 会切换或取消设置该选项，具体见各选项的说明。如果没有提及这类行为，那么省略*value* 只会显示当前设置。

不带任何参数的\pset 会显示所有打印选项的当前状态。

可调整的打印选项如下：

border

value 必须是数字。一般来说，数字越大，表格的边框和分隔线就越多，但细节取决于具体格式。在HTML 格式中，它会直接转换为border=...属性。在大多数其他格式中，只有值 0（无边框）、1（内部分隔线）和 2（表格外框）有意义，大于 2 的值会与 border = 2 作相同处理。latex 和 latex-longtable 格式还允许使用值 3，以在数据行之间添加分隔线。

columns

设置 wrapped 格式的目标宽度，同时也是确定输出是否足够宽以需要分页器或在扩展自动模式下切换到垂直显示的宽度限制。零（默认值）会导致目标宽度由环境变量 COLUMNS 控制，或者如果未设置 COLUMNS 则由检测到的屏幕宽度控制。另外，如果 columns 为零，则wrapped 格式仅影响屏幕输出。如果 columns 为非零，则文件和管道输出也会按该宽度折行。

expanded（或 x）

如果指定了 *value*，它必须是on 或 off（分别启用或禁用扩展模式），或者是auto。如果省略 *value*，该命令会在开启和关闭设置之间切换。启用扩展模式时，查询结果以两列显示，左侧为列名，右侧为数据。如果数据在通常的“横向”模式下无法适应屏幕，这种模式就很有用。在自动设置下，当查询输出包含多列且宽度超过屏幕时，会使用扩展模式；否则使用常规模式。自动设置只在对齐和折行格式中有效。在其他格式中，它的行为始终与关闭扩展模式相同。

fieldsep

指定非对齐输出格式使用的字段分隔符。这样可以创建例如制表符或逗号分隔的输出，这可能更符合其他程序的需要。要将制表符设置为字段分隔符，请输入 \pset fieldsep '\t'。默认字段分隔符是'|'（竖线）。

`fieldsep_zero`

将非对齐输出格式使用的字段分隔符设置为零字节。

`footer`

如果指定了 *`value`*，它必须是`on` 或 `off`，分别启用或禁用表格页脚（`(`*`n`* `rows)` 计数）的显示。如果省略 *`value`*，该命令会切换页脚显示的开关状态。

`format`

将输出格式设置为 `unaligned`、`aligned`、`wrapped`、`html`、`asciidoc`、`latex`（使用 `tabular`）、`latex-longtable` 或 `troff-ms` 之一。允许使用不产生歧义的缩写。

`unaligned` 格式将一行的所有列写在同一行上，以当前生效的字段分隔符分隔。这适合创建供其他程序读取的输出（例如制表符分隔或逗号分隔格式）。

`aligned` 格式是标准的、适合人阅读且排版整齐的文本输出；这是默认格式。

`wrapped` 格式与 `aligned` 相似，但会将较宽的数据值折成多行，使输出适应目标列宽。目标宽度的确定方式见`columns` 选项的说明。请注意，psql 不会尝试对列标题折行；因此，如果列标题所需的总宽度超过目标宽度，`wrapped` 格式的行为就与 `aligned` 相同。

`asciidoc`、`html`、`latex`、`latex-longtable` 和 `troff-ms` 格式生成的表格旨在包含在使用相应标记语言的文档中。它们不是完整的文档！这在 HTML 中可能不是必需的，但在 LaTeX 中，则必须有一个完整文档的外层结构。`latex-longtable` 格式需要 LaTeX 的 `longtable` 和 `booktabs` 包。

`linestyle`

设置边框线绘制样式为 `ascii`、`old-ascii` 或 `unicode` 之一。允许使用唯一缩写。（这意味着一个字母就足够了。）默认设置为 `ascii`。此选项仅影响 `aligned` 和 `wrapped` 输出格式。

`ascii` 样式使用普通的 ASCII 字符。数据中的换行符以右边缘的`+`符号表示。当 `wrapped` 格式在没有换行符的位置把数据折到下一行时，会在第一行的右边缘显示一个点（`.`），并在下一行的左边缘再次显示。

`old-ascii` 样式使用普通的 ASCII 字符，采用PostgreSQL 8.4 及更早版本的格式样式。数据中的换行符以替代左侧列分隔符的`:`符号表示。当数据在没有换行符的位置折到下一行时，则用`;`符号替代左侧列分隔符。

`unicode` 样式使用 Unicode 框线绘制字符。数据中的换行符以右边缘的回车符号表示。当数据在没有换行符的位置折到下一行时，会在第一行的右边缘显示省略号符号，并在下一行的左边缘再次显示。

当 `border` 设置大于零时，`linestyle` 选项还决定用哪些字符绘制边框线。普通的 ASCII 字符在任何环境中都可用，但在支持 Unicode 的显示设备上，Unicode 字符更美观。

null

设置用于代替空值打印的字符串。默认不打印任何内容，这很容易被误认为空字符串。例如，你可能更喜欢使用\pset null '(null)'。

numericlocale

如果指定了 *value*，它必须是on 或 off，分别启用或禁用使用区域设置特定的字符来分隔小数点左侧的数字组。如果省略*value*，该命令会在常规数字输出和区域设置特定的数字输出之间切换。

pager

控制查询和 psql 帮助输出是否使用分页器程序。如果设置了环境变量 PSQL_PAGER 或 PAGER，输出会通过管道传递给指定的程序。否则，使用与平台有关的默认程序（如 more）。

当 pager 选项为 off 时，不使用分页器程序。当pager 选项为 on 时，会在适当时使用分页器，即输出目标为终端且内容无法在屏幕上完整显示时。pager 选项也可以设为 always，这样所有终端输出都会使用分页器，无论内容是否能在屏幕上完整显示。不带 *value* 的\pset pager 会切换分页器的使用状态。

pager_min_lines

如果将 pager_min_lines 设置为大于页面高度的数字，那么只有待显示的输出至少达到这么多行时，才会调用分页器程序。默认设置为 0。

recordsep

指定非对齐输出格式使用的记录（行）分隔符。默认为换行符。

recordsep_zero

将非对齐输出格式使用的记录分隔符设置为零字节。

tableattr（或 T）

在 HTML 格式中，这指定要放在table 标签内的属性，例如cellpadding 或 bgcolor。请注意，你可能不需要在这里指定border，因为\pset border 已经负责处理它。如果没有给出 *value*，则取消设置表格属性。

在 latex-longtable 格式中，这控制每个包含左对齐数据类型的列的宽度比例。它以空白分隔的值列表指定，例如'0.2 0.2 0.6'。未指定的输出列使用最后指定的值。

title（或 C）

设置随后打印的所有表格的标题。这可以为输出提供描述性标签。如果没有给出 *value*，则取消设置标题。

tuples_only（或 t）

如果指定了 *value*，它必须是on 或 off，分别启用或禁用仅元组模式。如果省略 *value*，该命令会在常规输出和仅元组输出之间切换。常规输出包含列标题、表格标题和各种页脚等附加信息。在仅元组模式下，只显示实际的表格数据。

unicode_border_linestyle

将 unicode 线条样式的边框绘制样式设置为 single 或 double。

unicode_column_linestyle

将 unicode 线条样式的列分隔线绘制样式设置为 single 或 double。

unicode_header_linestyle

将 unicode 线条样式的表头分隔线绘制样式设置为 single 或 double。

这些不同格式的外观示例可参见示例一节。

提示

有各种快捷命令用于\pset。请参见\a、\C、\f、\H、\t、\T 和\x。

\q 或 \quit

退出 psql 程序。在脚本文件中，只会终止该脚本的执行。

\qecho *text* [...]

这个命令与\echo 相同，只是输出会写入由\o 设置的查询输出通道。

\r 或 \reset

重置（清空）查询缓冲区。

\s [*filename*]

打印 psql 的命令行历史记录到 *filename*。如果省略 *filename*，历史记录将被写入标准输出（如果适用，将使用分页器）。如果psql 在构建时没有使用 Readline 支持，则此命令不可用。

\set [*name* [*value* [...]]]

将 psql 变量 *name* 设置为 *value*，如果给出多个值，则设置为所有值的串接。如果只给出一个参数，则将变量设置为空字符串值。要取消变量设置，请使用\unset 命令。

\set 没有任何参数时，显示当前设置的所有psql 变量的名称和值。

合法的变量名可以包含字母、数字和下划线。详情见下面的变量。变量名区分大小写。

某些变量是特殊的，它们控制psql 的行为，或者由系统自动设置以反映连接状态。下面的变量介绍了这些变量。

注意

这个命令与 SQL 命令 SET 无关。

\setenv *name* [*value*]

设置环境变量 *name* 为 *value*，或者如果未提供*value*，则取消设置环境变量。示例：

```
testdb=> \setenv PAGER less
testdb=> \setenv LESS -imx4F
```

\sf[+] *function_description*

这个命令获取并显示指定函数或过程的定义，以CREATE OR REPLACE FUNCTION 或 CREATE OR REPLACE PROCEDURE 命令的形式呈现。定义将打印到当前查询输出通道，由 \o 设置。

目标函数可以只用名称指定，也可以同时给出名称和参数，例如foo(integer, text)。如果存在多个同名函数，就必须给出参数类型。

如果在命令名后附加+，输出行会带上行号，函数体的第一行编号为 1。

与大多数其他元命令不同，整个行的剩余部分始终被视为\sf 的参数，参数中不进行变量插值或反引号扩展。

\sv[+] *view_name*

这个命令获取指定视图的定义，并以CREATE OR REPLACE VIEW 命令的形式显示。定义会打印到由\o 设置的当前查询输出通道。

如果在命令名称后添加+，那么输出行将从 1 开始编号。

与大多数其他元命令不同，整个行的剩余部分始终被视为\sv 的参数，参数中不进行变量插值或反引号扩展。

\t

切换输出中的列名标题和行数页脚的显示状态。这个命令等价于\pset tuples_only，提供它是为了使用方便。

\T *table_options*

指定在 HTML 输出格式中放在 table 标签内的属性。这个命令等价于\pset tableattr *table_options*。

\timing [*on* | *off*]

带参数时，开启或关闭每条 SQL 语句执行耗时的显示。不带参数时，切换显示的开关状态。耗时以毫秒显示；超过 1 秒的时间间隔还会以分钟:秒的格式显示，必要时添加小时和天字段。

\unset *name*

取消设置（删除）psql 变量 *name*。

大多数控制 psql 行为的变量不能取消设置；对于这些变量，\unset 命令会被解释为将其设置为默认值。参见下面的变量。

\w 或 \write *filename*
\w 或 \write |*command*

将当前查询缓冲区写入文件 *filename*，或通过管道传递给 shell 命令*command*。如果当前查询缓冲区为空，则改为写入最近执行的查询。

如果参数以|开头，则该行剩余的全部内容会被视为要执行的*command*，其中不会进行变量插值或反引号扩展。该行剩余的内容只会原样传递给 shell。

\watch [*seconds*]

重复执行当前查询缓冲区（如同 \g 一样），直到被中断或查询失败。两次执行之间等待指定的秒数（默认 2 秒）。每次查询结果都会带有一个头部，其中包含 \pset title 字符串（如果有）、查询开始时的时间以及延迟间隔。

如果当前查询缓冲区为空，则重新执行最近发送的查询。

\x [*on* | *off* | *auto*]

设置或切换扩展表格格式模式。它等价于\pset expanded。

\z [*pattern*]

列出表、视图和序列及其关联的访问权限。如果指定了*pattern*，则只列出名称匹配该模式的表、视图和序列。

这是\dp 的别名（“显示权限”）。

\! [*command*]

不带参数时，进入一个子 shell；子 shell 退出后，psql 恢复运行。带参数时，执行 shell 命令*command*。

与大多数其他元命令不同，该行剩余的全部内容始终被视为\!的参数，其中不会进行变量插值或反引号扩展。该行剩余内容会原样传给 shell。

\? [*topic*]

显示帮助信息。可选的 *topic* 参数（默认为 commands）选择要解释的 psql 的哪个部分：commands 描述 psql 的反斜线命令；options 描述可以传递给 psql 的命令行选项；而 variables 显示关于 psql 配置变量的帮助。

\;

反斜线加分号并不是与前述命令相同的元命令；它只是将一个分号添加到查询缓冲区，而不进行任何进一步处理。

通常，psql 一遇到结束命令的分号，就会将 SQL 命令发送给服务器，即使当前行仍有更多输入也是如此。例如，输入

```
select 1; select 2; select 3;
```

会将这三条 SQL 命令分别发送给服务器，每条命令的结果会在继续处理下一条命令之前显示。不过，以以下形式输入的分号：`\;`，不会触发命令处理，因此它前后的命令实际上会合并到一个请求中发送给服务器。例如，输入

```
select 1\; select 2\; select 3;
```

会在遇到未加反斜线的分号时，将这三条 SQL 命令作为一个请求发送给服务器。服务器会将这样的请求作为单个事务执行，除非字符串中包含显式的`BEGIN`/`COMMIT` 命令，将其划分为多个事务。（有关服务器如何处理多查询字符串的更多细节，请参见第 53.2.2.1 节。）psql 对每个请求只打印收到的最后一个查询结果；在这个示例中，虽然三条`SELECT` 命令确实都会执行，但psql 只打印 `3`。

模式

很多`\d` 命令都可以用一个 *`pattern`* 参数来指定要被显示的对象名称。在最简单的情况下，模式正好就是该对象的准确名称。在模式中的字符通常会被变成小写形式（就像在 SQL 名称中那样），例如`\dt FOO` 将会显示名为 `foo` 的表。就像在 SQL 名称中那样，把模式放在双引号中可以阻止它被转换成小写形式。如果需要在一个模式中包括一个真正的双引号字符，则需要在双引号包围的文本内把它写成两个相邻的双引号，这同样是符合 SQL 加引号标识符的规则。例如，`\dt "FOO""BAR"` 将显示名为 `FOO"BAR`（不是 `foo"bar`）的表。和普通的 SQL 名称规则不同，你可以只在模式的一部分周围放上双引号，例如`\dt FOO"FOO"BAR` 将会显示名为 `fooFOObar` 的表。

只要完全省略 *`pattern`* 参数，`\d` 命令就会显示当前模式搜索路径中可见的全部对象 – 这等价于用`*` 作为模式（如果一个对象所在的模式位于搜索路径中，并且在搜索路径中该模式之前没有同类且同名的对象，则该对象就是可见的。这表示可以直接用名称引用该对象，而不需要用模式来限定）。要查看数据库中所有对象而不考虑其可见性，可以把`*.*` 用作模式。

如果放在一个模式中，`*` 将匹配任意字符序列（包括空序列），而`?`会匹配任意的单个字符（这种记号方法就像 Unix shell 的文件名模式一样）。例如，`\dt int*` 会显示名称以 `int` 开始的表。但是如果被放在双引号内，`*` 和`?`就会失去这些特殊含义而变成普通的字符。

包含点号（`.`）的模式会被解释为模式名称的匹配模式，后接对象名称的匹配模式。例如，`\dt foo*.*bar*` 显示模式名以 `foo` 开头且表名包含 `bar` 的所有表。如果没有点号，则该匹配模式只匹配当前模式搜索路径中可见的对象。同样，双引号内的点号会失去其特殊含义，而按字面匹配。

高级用户可以使用字符类等正则表达式记法，如`[0-9]` 可以匹配任意数字。所有的正则表达式特殊字符都按照第 9.7.3 节所说的工作，以下字符除外：`.`会按照上面所说的作为一种分隔符，`*` 会被翻译成正则表达式记号`.*`，`?`会被翻译成`.`，而`$`则按字面意思匹配。根据需要，可以用`?`模拟`.`，用`(`*`R`*`+|)` 模拟 *`R`*`*`，或用`(`*`R`*`|)` 模拟 *`R`*`?`。`$`不需要作为一个正则表达式字符，因为模式必须匹配整个名称，而不是像正则表达式的常规用法那样解释（换句话说，`$`会被自动地追加到模式上）。如果不希望该模式的匹配位置被固定，可以在开头或者结尾写上`*`。注意在双引号内，所有的正则表达式特殊字符会失去其特殊含义并且按照其字面意思进行匹配。还有，在操作符名称模式中（即作为`\do` 的参数），正则表达式特殊字符也按照字面意思进行匹配。

高级特性

变量

psql 提供了和普通 Unix 命令 shell 相似的变量替换特性。变量简单来说就是一对名称/值，其中值可以是任意长度的任意字符串。名称必须由字母（包括非拉丁字母）、数字和下划线构成。

要设置一个变量，可以使用psql 的元命令\set。例如，

```
testdb=> \set foo bar
```

会将 foo 设置为值 bar。要检索该变量的内容，可以在名称前放一个冒号，例如：

```
testdb=> \echo :foo
bar
```

这在常规 SQL 命令和元命令中均有效，下文的SQL 插值中有更多细节。

如果调用\set 时没有第二个参数，该变量会被设置为一个空字符串值。要取消设置（即删除）一个变量，可以使用命令\unset。要显示所有变量的值，在调用\set 时不带任何参数即可。

> **注意**
>
> \set 的参数服从与其他命令相同的替换规则。因此可以构造有趣的引用，例如 \set :foo 'something' 以及分别得到 Perl 或者 PHP 的“软链接”或者“可变变量”。不幸的是（或者幸运的是？），这些构造出来的东西并没有什么用处。在另一方面，\set bar :foo 是一种很好的拷贝变量的方法。

有一些变量会被 psql 特殊对待。它们表示特定的选项设置，运行时这类选项设置可以通过修改该变量的值来改变，或者在某些情况下它们表示psql 的可更改的状态。按照惯例，所有被特殊对待的变量的名称由全部大写形式的 ASCII 字母（还有可能是数字和下划线）组成。为了确保未来最大的兼容性，最好避免把这类变量名用于自己的目的。

控制 psql 行为的变量通常不能被取消设置或者设置为无效值。允许\unset 命令，但它会被解释为将变量设置为它的默认值。没有第二参数的\set 命令会被解释为将变量设置为 on（对于接受该值的控制变量），对不接受该值的变量则会拒绝这个命令。此外，接受值on 和 off 的控制变量也能接受其他常见的布尔值拼写方式，例如true 和 false。

被特殊对待的变量是：

AUTOCOMMIT

在被设置为 on（默认）时，每一个 SQL 命令在成功完成时会被自动提交。在这种模式中要推迟提交，必须输入一个BEGIN 或者 START TRANSACTION SQL 命令。当被设置为 off 或者被取消设置时，在显式发出COMMIT 或者 END 之前，SQL 命令不会被提交。自动提交关闭模式会为你发出一个隐式的BEGIN，这会发生在任何不在一个事务块中且本身既不是BEGIN 及其他事务控制命令且不是无法在事务块中执行的命令（例如 VACUUM）之前。

> **注意**
>
> 在自动提交关闭模式中，必须通过ABORT 或者 ROLLBACK 显式地放弃任何失败的事务。还要记住，如果退出会话时没有提交，则所有的工作都会丢失。

> **注意**
>
> 自动提交打开模式是 PostgreSQL 的传统行为，但是自动提交关闭模式更接近于 SQL 的规范。如果更喜欢自动提交关闭模式，可以在系统级的psqlrc 文件或者个人的~/.psqlrc 文件中设置它。

COMP_KEYWORD_CASE

确定在补全一个 SQL 关键词时要使用的大小写形式。如果被设置为 lower 或者 upper，补全后的词将分别是小写或者大写形式。如果被设置为preserve-lower 或者 preserve-upper（默认），补全后的词将会保持该词已输入部分的大小写形式，但是如果被补全的词还没有被输入，则它会被分别补全成小写或者大写形式。

DBNAME

当前已连接的数据库名称。每次连接到一个数据库时都会设置该变量（包括程序启动时），但是可以被更改或者取消设置。

ECHO

如果被设置为 all，所有非空输入行会在读入时打印到标准输出（不适用于交互式读取的行）。要在程序开始时选择这种行为，可以使用开关-a。如果被设置为 queries，psql 会在发送每个查询给服务器时将它们打印到标准输出。选择这种行为的开关是 -e。如果被设置为 errors，那么只有失败的查询会被显示在标准错误输出上。这种行为的开关是-b。如果被设置为 none（默认值）则不会显示任何查询。

ECHO_HIDDEN

当这个变量被设置为 on 且一个反斜线命令查询数据库时，相应的查询会被先显示。这种特性可以帮助我们学习PostgreSQL 的内部并且在自己的程序中提供类似的功能（要在程序开始时选择这种行为，可以使用开关-E）。如果把这个变量设置为值 noexec，则对应的查询只会被显示而并不真正被发送给服务器执行。默认值是off。

ENCODING

当前的客户端字符集编码。每一次你连接到一个数据库（包括程序启动）时以及当你用\encoding 更改编码时，这个变量都会被设置，但它可以被更改或者取消设置。

ERROR

如果上一个 SQL 查询失败则为 true，如果成功则是false。另见 SQLSTATE。

FETCH_COUNT

如果这个变量被设置为一个大于零的整数值，SELECT 查询的结果会以一组一组的方式取出并且显示（而不是像默认的那样把整个结果集拿到以后再显示），每组包含的行数等于该整

数值。因此，这种方式只会使用有限的内存量，而不管整个结果集的大小。在启用这个特性时，通常会使用 100 到 1000 的设置。记住在使用这种特性时，一个查询可能会在已经显示了一些行之后失败。

> **提示**
>
> 尽管可以把这种特性用于任何的输出格式，但是默认的`aligned` 格式看起来会比较糟糕，因为每一组的`FETCH_COUNT` 行将被单独格式化，这就会导致不同的行组的列宽不同。其他的输出格式会更好。

`HISTCONTROL`

如果这个变量被设置为 `ignorespace`，则以一个空格开始的行不会被放入到历史列表中。如果被设置为值`ignoredups`，则与上一条历史记录相同的行不会被放入。值`ignoreboth` 组合了上述两种值。如果被设置为 `none`（默认值），所有在交互模式中被读入的行都会保存在历史列表中。

> **注意**
>
> 这个特性是可耻地从 Bash 抄袭过来的。

`HISTFILE`

用于存储历史记录列表的文件名。如果未设置，则从环境变量 `PSQL_HISTORY` 中获取文件名。如果该环境变量也未设置，则默认使用 `~/.psql_history`，在 Windows 上则使用 `%APPDATA%\postgresql\psql_history`。例如，将以下内容：

```
\set HISTFILE ~/.psql_history- :DBNAME
```

放入 `~/.psqlrc` 会使 psql 为每个数据库维护单独的历史记录。

> **注意**
>
> 这个特性是可耻地从 Bash 抄袭过来的。

`HISTSIZE`

存储在命令历史中的最大命令数（默认值是 500）。如果被设置为一个负值，则不会应用限制。

> **注意**
>
> 这个特性是可耻地从 Bash 抄袭过来的。

HOST

当前连接到的数据库服务器主机。每次连接到数据库时都会设置该变量（包括程序启动时），但也可以被更改或取消设置。

IGNOREEOF

如果被设置为 1 或者更小，向一个psql 的交互式会话发送一个 EOF 字符（通常是 **Control+D**）将会终止应用。如果设置为一个较大的数字值，则必须连续键入与该数值相等数量的EOF 字符才能让交互式会话终止。如果该变量被设置为一个非数字值，则它会被解释为 10。默认值为 0。

> **注意**
>
> 这个特性是可耻地从 Bash 抄袭过来的。

LASTOID

最后被影响的 OID 的值，这可能会由INSERT 或者\lo_import 命令返回。这个变量只保证在下一个 SQL 命令的结果被显示完之前有效。

LAST_ERROR_MESSAGE
LAST_ERROR_SQLSTATE

当前 psql 会话中最近一个失败查询的主错误消息和相关的 SQLSTATE 代码，如果在当前会话中没有发生错误，则是一个空字符串和00000。

ON_ERROR_ROLLBACK

当被设置为 on 时，如果事务块中的一个语句产生一个错误，该错误会被忽略并且该事务会继续。当被设置为interactive 时，只在交互式会话中忽略这类错误，而读取脚本文件时则不会忽略错误。当被设置为off（默认值）时，事务块中产生错误的一个语句会中止整个事务。错误回滚模式的工作原理是在事务块的每个命令之前都为你发出一个隐式的SAVEPOINT，然后在该命令失败时回滚到该保存点。

ON_ERROR_STOP

默认情况下，发生错误后命令处理会继续。当这个变量被设置为on 时，处理将立即停止。在交互模式下，psql 会返回到命令提示符；否则，psql 会退出，并返回错误代码 3，以便与错误代码 1 所表示的致命错误区分开来。在两种情况下，任何当前正在运行的脚本（顶层脚本以及它可能调用的其他脚本）都会立即中止。如果顶层命令字符串包含多个 SQL 命令，则会在当前命令处停止处理。

PORT

当前连接到的数据库服务器端口。每次连接到数据库时都会设置该变量（包括程序启动时），但也可以被更改或取消设置。

PROMPT1
PROMPT2
PROMPT3

这些变量指定 psql 发出的提示符的模样。见下文的提示符。

QUIET

把这个变量设置为 on 等效于命令行选项 -q。在交互模式下可能用处不大。

ROW_COUNT

上一个 SQL 查询返回的行数或者受影响的行数，如果该查询失败或者没有报告行计数则为 0。

SERVER_VERSION_NAME
SERVER_VERSION_NUM

字符串形式的服务器版本号，例如9.6.2、10.1 或者 11beta1，以及数字形式的服务器版本号，例如90602 或者 100001。每次你连接到一个数据库（包括程序启动）时，这些都会被设置，但可以被改变或者取消设置。

SHOW_CONTEXT

这个变量可以被设置为值 never、errors 或者 always 来控制是否在来自服务器的消息中显示 CONTEXT 字段。默认是 errors（表示在错误消息中显示上下文，但在通知和警告消息中不显示）。当VERBOSITY 被设置为 terse 时，这个设置无效（另见\errverbose，它可以用来得到刚遇到的错误的详细信息）。

SINGLELINE

设置这个变量为 on 等效于命令行选项 -S。

SINGLESTEP

设置这个变量为 on 等效于命令行选项 -s。

SQLSTATE

与上一个 SQL 查询的失败相关的错误代码（见附录 A），如果上一个查询成功则为00000。

USER

当前连接的数据库用户。每次连接到一个数据库时都会设置该变量（包括程序启动时），但是可以被更改或者取消设置。

VERBOSITY

这个变量可以被设置为值 default、verbose 或者 terse 来控制错误报告的详细程度（另见\errverbose，在想得到刚遇到的错误的详细信息时使用）。

VERSION
VERSION_NAME
VERSION_NUM

这些变量在程序启动时被设置以反映 psql 的版本，分别是一个详细的字符串、一个短字符串（例如9.6.2、10.1 或者 11beta1）以及一个数字（例如 90602 或者 100001）。它们可以被更改或取消设置。

SQL 插值

psql 变量的一个关键特性是可以把它们替换（“插值”）到常规 SQL 语句中，也可以把它们替换到元命令的参数中。此外，psql 还提供了功能来确保被用作 SQL 字面量和标识符的变量值会被正确地加引号。不加引号地插值一个值的语法是在变量名前面加上一个冒号（:）。例如，

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

将查询表 my_table。注意这可能会不安全：该变量的值会被按字面拷贝，因此它可能包含不平衡的引号甚至反斜线命令。必须确保把它放在那里是有意义的。

当一个值要用作 SQL 字面量或标识符时，最安全的做法是为它加上引号。要将变量值作为 SQL 字面量加引号，应写一个冒号，后面跟用单引号括起来的变量名。要将变量值作为 SQL 标识符加引号，则在冒号后面用双引号括起变量名。这些写法能正确处理变量值中嵌入的引号和其他特殊字符。前面的示例可用以下更安全的写法：

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :"foo";
```

在加引号的 SQL 字面量和标识符内部，不会执行变量插值。因此，':foo' 这样的写法不能根据变量值生成加引号的字面量（即使能够生效，也不安全，因为它无法正确处理变量值中嵌入的引号）。

使用这种机制的一个示例是把一个文件的内容拷贝到一个表列中。首先把该文件载入到一个变量，然后把该变量的值作为一个加引号的字符串进行插值：

```
testdb=> \set content `cat my_file.txt`
testdb=> INSERT INTO my_table VALUES (:'content');
```

（注意如果 my_file.txt 包含 NUL 字节，这样也不行。psql 不支持在变量值中嵌入 NUL 字节）。

因为冒号可以合法地出现在 SQL 命令中，一次明显的插值尝试（即:name、:'name' 或者:"name"）不会被替换，除非所指的变量当前已设置。在任何情况下，可以用一个反斜线对冒号进行转义以避免它被替换。

:{?*name*}特殊语法会根据该变量是否存在返回 TRUE 或 FALSE，因此总是会被替换，除非冒号被反斜线转义。

变量的冒号语法对嵌入式查询语言（例如 ECPG）来说是标准的 SQL。用于数组切片和类型转换的冒号语法是 PostgreSQL 扩展，它有时可能会与标准用法冲突。把一个变量值转义成 SQL 字面量或者标识符的冒号加引号语法是一种psql 扩展。

提示符

psql 发出的提示符可以按你的喜好进行定制。PROMPT1、PROMPT2 和 PROMPT3 这三个变量包含描述提示符外观的字符串和特殊转义序列。提示符 1 是 psql 请求新命令时发出的常规提示符。提示符 2 会在录入命令期间还需要更多输入时发出，例如命令尚未以分号结束，或者引号尚未闭合时。在执行 SQL COPY FROM STDIN 命令并需要在终端中输入一行值时，会发出提示符 3。

所选提示符变量的值会原样打印，除非遇到百分号（%）。此时会根据下一个字符替换为其他文本。已定义的替换项如下：

%M

数据库服务器的完整主机名（含域名）；如果通过 Unix 域套接字连接，则为[local]；如果 Unix 域套接字不在编译时指定的默认位置，则为[local:*/dir/name*]。

%m

数据库服务器的主机名，在第一个点号处截断；如果通过 Unix 域套接字连接，则为[local]。

%>

数据库服务器监听的端口号。

%n

数据库会话用户名。（在数据库会话期间，SET SESSION AUTHORIZATION 命令可能改变该值的扩展结果。）

%/

当前数据库的名称。

%~

类似 %/，但如果该数据库是你的默认数据库，则输出 ~（波浪号）。

%#

如果会话用户是数据库超级用户，则为#，否则为>。（在数据库会话期间，SET SESSION AUTHORIZATION 命令可能改变该值的扩展结果。）

%p

当前所连接后端的进程 ID。

%R

在提示符 1 中，通常为 =；但如果会话处于条件块的非活动分支中，则为 @；如果处于单行模式，则为 ^；如果会话已与数据库断开连接（这可能发生在 \connect 失败时），则为 !。在提示符 2 中，%R 会被替换为一个字符，该字符取决于 psql 为什么还期待更多输入：如果命令只是尚未终止，则为 -；如果存在未结束的 /* ... */ 注释，则为 *；如果存在未结束的带引号字符串，则为单引号；如果存在未结束的带引号标识符，则为双引号；如果存在未结束的美元引用字符串，则为美元符号；如果存在未匹配的左括号，则为 (。在提示符 3 中，%R 不会产生任何输出。

%x

事务状态：如果当前不在事务块中，则为空字符串；如果处于事务块中，则为 *；如果处于失败的事务块中，则为 !；如果事务状态不确定（例如因为当前没有连接），则为 ?。

`%l`

当前语句中的行号，从 `1` 开始。

`%digits`

替换为指定八进制代码对应的字符。

`%:name:`

psql 变量 *name* 的值。详见变量。

`%`command``

command 的输出，类似普通的“反引号”替换。

`%[ ... %]`

提示符中可以包含终端控制字符，例如改变提示文本的颜色、背景或样式，或者改变终端窗口标题。为了让 Readline 的行编辑功能正常工作，这些不可打印的控制字符必须用 `%[` 和 `%]` 包围起来，以标记为不可见字符。提示符中可以出现多组这样的标记。例如：

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

其效果是在兼容 VT100 且支持颜色的终端上，生成一个粗体（`1;`）的黑底黄字提示符（`33;40`）。

要在提示符中插入百分号，请写为`%%`。默认情况下，提示符 1 和 2 为`'%/%R%# '`，提示符 3 为`'>> '`。

> **注意**
>
> 这个特性是可耻地从 tcsh 抄袭过来的。

命令行编辑

psql 支持 Readline 库，便于编辑和检索输入行。命令历史记录会在 psql 退出时自动保存，并在 psql 启动时重新载入。也支持 Tab 补全，不过其补全逻辑并不声称自己是 SQL 解析器。Tab 补全生成的查询还可能干扰其他 SQL 命令，例如 `SET TRANSACTION ISOLATION LEVEL`。如果出于某种原因你不喜欢 Tab 补全，可以将以下内容放入主目录下名为 `.inputrc` 的文件中，将其关闭：

```
$if psql
set disable-completion on
$endif
```

（这不是 psql 的功能，而是 Readline 的功能。更多细节请阅读其文档。）

环境

COLUMNS

如果\pset columns 为零，这个环境变量控制用于wrapped 格式的宽度以及用来确定是否输出需要用到分页器或者切换到扩展自动模式中的垂直格式的宽度。

PGDATABASE
PGHOST
PGPORT
PGUSER

默认连接参数（见第 34.14 节）。

PSQL_EDITOR
EDITOR
VISUAL

\e、\ef 以及\ev 命令所使用的编辑器。会按照列出的顺序检查这些变量，第一个被设置的将被使用。如果都没有被设置，默认是使用 Unix 系统上的vi 或者 Windows 系统上的notepad.exe。

PSQL_EDITOR_LINENUMBER_ARG

当 \e、\ef 或 \ev 带有行号参数使用时，此变量指定将起始行号传给用户编辑器时所用的命令行参数。对于 Emacs 或 vi 之类的编辑器，这个参数是一个加号。如果选项名和行号之间需要空格，请在变量值中包含末尾空格。例如：

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

在 Unix 系统上默认是+（对应于默认编辑器 vi，且对很多其他常见编辑器可用）。在 Windows 系统上没有默认值。

PSQL_HISTORY

命令历史文件的替代位置。波浪线（~）扩展会被执行。

PSQL_PAGER
PAGER

如果查询结果无法在屏幕上显示完整，它们会通过管道传给此命令。典型的值是more 或 less。可以通过将 PSQL_PAGER 或 PAGER 设置为空字符串，或通过调整\pset 命令的与分页相关的选项来禁用分页器。这些变量按照列出的顺序进行检查；使用按该顺序找到的第一个已设置的变量。如果它们都没有设置，默认情况下在大多数平台上使用more，但在 Cygwin 上使用less。

PSQLRC

用户的.psqlrc 文件的替代位置。波浪线（~）扩展会被执行。

SHELL

被\!命令执行的命令。

TMPDIR

存储临时文件的目录。默认是/tmp。

与大多数其他 PostgreSQL 工具一样，此工具也使用 libpq 支持的环境变量（见第 34.14 节）。

文件

psqlrc 和 ~/.psqlrc

如果没有 -X 选项，在连接到数据库后但在接收正常的命令之前，psql 会尝试依次从系统级的启动文件（psqlrc）和用户的个人启动文件（~/.psqlrc）中读取并且执行命令。这些文件可以被用来设置客户端或者服务器，通常是一些\set 和 SET 命令。

系统范围的启动文件名为 psqlrc。默认情况下，会在安装的“系统配置”目录中查找它，最可靠的识别方式是运行 pg_config --sysconfdir。通常这个目录是相对于包含 PostgreSQL 可执行文件的目录的 ../etc/。也可以通过 PGSYSCONFDIR 环境变量显式指定查找目录。

用户的个人启动文件名为.psqlrc，并且在调用用户的主目录中寻找。Windows 没有主目录这一概念，在 Windows 上，个人启动文件的名称为%APPDATA%\postgresql\psqlrc.conf。在任何情况下，可以通过设置PSQLRC 环境变量来覆盖此默认文件路径。

系统范围的启动文件和用户个人的启动文件都可以通过在文件名后附加连字符和 PostgreSQL 的大版本或小版本号来使其与 psql 版本相关，例如~/.psqlrc-9.2 或~/.psqlrc-9.2.5。最具体版本匹配的文件将优先读取，而不是非特定版本的文件。

.psql_history

命令行历史被存储在文件~/.psql_history 中，或者是 Windows 的文件%APPDATA%\postgresql\psql_history 中。

历史文件的位置可以通过 HISTFILE psql 变量或者 PSQL_HISTORY 环境变量显式设置。

注解

- psql 最适合与相同或较旧大版本的服务器配合使用。如果服务器的版本比 psql 本身更新，反斜线命令特别容易失败。然而，\d 系列的反斜线命令应该可以在最低至 7.4 版本的服务器上运行，但不一定适用于比psql 本身更新的服务器。运行 SQL 命令和显示查询结果的一般功能也应该可以在更新大版本的服务器上运行，但不能保证在所有情况下都能实现。

  如果你想用 psql 连接到多个具有不同大版本的服务器，推荐使用最新版本的psql。或者，你可以为每一个大版本保留一份psql 拷贝，并且针对相应的服务器使用匹配的版本。但实际上，这种额外的麻烦是不必要的。

- 在 PostgreSQL 9.6 之前，-c 选项意味着 -X（--no-psqlrc）；现在已经不是这样了。

- 在 PostgreSQL 8.4 之前，psql 允许一个单字母反斜线命令的第一个参数直接写在该命令后面，中间不需要空白。现在则要求用空白分隔。

给 Windows 用户的注解

psql 是一个 “控制台应用” 。由于 Windows 的控制台窗口使用的是一种和系统中其他应用不同的编码，在psql 中使用 8 位字符时要特别注意。如果 psql 检测到一个有问题的控制台代码页，它将会在启动时警告你。要更改控制台代码页，有两件事是必要的：

- 输入 **cmd.exe /c chcp 1252** 可以设置代码页（1252 是适用于德语的一个代码页，请在这里替换成你的值）。如果正在使用 Cygwin，可以把这个命令放在`/etc/profile` 中。
- 把控制台字体设置为 `Lucida Console`，因为栅格字体无法与 ANSI 代码页一起使用。

示例

第一个示例展示了如何把一条命令分散到多行输入中。注意提示符的变化：

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
testdb-> ;
CREATE TABLE
```

现在再来看一下这个表的定义：

```
testdb=> \d my_table
              Table "public.my_table"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 first  | integer |           | not null | 0
 second | text    |           |          |
```

现在我们把提示符改成更有趣的形式：

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

假设你已经向表中填入数据，并想看一看：

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

你可以使用 \pset 命令以不同方式显示表格：

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format csv
Output format is csv.
peter@localhost testdb=> \pset tuples_only
Tuples only is on.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep '\t'
Field separator is "    ".
peter@localhost testdb=> SELECT second, first FROM my_table;
one     1
two     2
three   3
four    4
```

或者使用短命令：

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

在适合的情况下，可以使用 \crosstabview 命令将查询结果显示为交叉表：

```
testdb=> SELECT first, second, first > 2 AS gt2 FROM my_table;
 first | second | gt2
-------+--------+-----
     1 | one    | f
     2 | two    | f
     3 | three  | t
     4 | four   | t
(4 rows)

testdb=> \crosstabview first second
 first | one | two | three | four
-------+-----+-----+-------+------
     1 | f   |     |       |
     2 |     | f   |       |
     3 |     |     | t     |
     4 |     |     |       | t
(4 rows)
```

第二个示例展示了一个乘法表，其中行按数字倒序排列，而列按独立的升序数字排列。

```
testdb=> SELECT t1.first as "A", t2.first+100 AS "B", t1.first*(t2.first+100) as "AxB",
testdb(> row_number() over(order by t2.first) AS ord
testdb(> FROM my_table t1 CROSS JOIN my_table t2 ORDER BY 1 DESC
testdb(> \crosstabview "A" "B" "AxB" ord
 A | 101 | 102 | 103 | 104
---+-----+-----+-----+-----
 4 | 404 | 408 | 412 | 416
 3 | 303 | 306 | 309 | 312
 2 | 202 | 204 | 206 | 208
 1 | 101 | 102 | 103 | 104
(4 rows)
```

reindexdb

reindexdb — 重建一个 PostgreSQL 数据库中的索引

大纲

reindexdb [*connection-option*...] [*option*...] [-S | --schema *schema*] ... [-t | --table *table*] ... [-i | --index *index*] ... [*dbname*]

reindexdb [*connection-option*...] [*option*...] -a | --all

reindexdb [*connection-option*...] [*option*...] -s | --system [*dbname*]

描述

reindexdb 是用于重建 PostgreSQL 数据库中索引的工具。

reindexdb 是 SQL 命令 REINDEX 的一个包装器。通过该工具重建索引与通过其他方法访问服务器来重建索引，在效果上没有区别。

选项

reindexdb 接受以下命令行参数：

-a
--all

重建所有数据库的索引。

[-d] *dbname*
[--dbname=]*dbname*

在未使用 -a/--all 时，指定要重建索引的数据库名称。如果未指定该选项，则从环境变量PGDATABASE 中读取数据库名称。如果该变量未设置，则使用为连接指定的用户名。*dbname* 可以是一个连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

-e
--echo

回显 reindexdb 生成并发送给服务器的命令。

-i *index*
--index=*index*

只重建 *index*。可以通过写多个 -i 开关来重建多个索引。

-q
--quiet

不显示进度消息。

`-s`
`--system`

仅重建数据库的系统目录上的索引。

`-S` *`schema`*
`--schema=`*`schema`*

只重建 *`schema`* 中的索引。可以通过写多个 `-S` 开关来重建多个模式中的索引。

`-t` *`table`*
`--table=`*`table`*

只重建 *`table`* 上的索引。可以通过写多个 `-t` 开关来重建多个表上的索引。

`-v`
`--verbose`

在处理过程中打印详细信息。

`-V`
`--version`

打印 reindexdb 的版本并退出。

`-?`
`--help`

显示关于 reindexdb 命令行参数的帮助信息，并退出。

reindexdb 还接受以下用于连接参数的命令行参数：

`-h` *`host`*
`--host=`*`host`*

指定服务器所在机器的主机名。如果该值以斜杠开头，则它将被用作 Unix 域套接字的目录。

`-p` *`port`*
`--port=`*`port`*

指定服务器监听连接所用的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U` *`username`*
`--username=`*`username`*

要作为其身份连接的用户名。

`-w`
`--no-password`

永远不发出密码提示。如果服务器要求密码认证，而又无法通过其他方式获取密码，例如通过`.pgpass` 文件，则连接尝试会失败。此选项可用于没有用户可输入密码的批处理作业和脚本。

`-W`
`--password`

强制 reindexdb 在连接数据库前提示输入密码。

该选项并非必需，因为如果服务器要求密码认证，reindexdb 会自动提示输入密码。不过，reindexdb 会先浪费一次连接尝试来确认服务器需要密码。在某些场景下，值得使用 `-W`，以避免这次额外的连接尝试。

`--maintenance-db=`*dbname*

当使用 `-a`/`--all` 时，连接到该数据库以收集要重建索引的数据库列表。如果未指定，则使用`postgres` 数据库；如果它不存在，则使用`template1`。这可以是一个连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。此外，除数据库名称本身以外的连接字符串参数，在连接到其他数据库时也会被重复使用。

环境

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

与大多数其他 PostgreSQL 工具一样，该工具也使用libpq 支持的环境变量（见第 34.14 节）。

诊断

如果遇到困难，请参阅REINDEX 和 psql 中关于潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。此外，libpq 前端库所使用的任何默认连接设置和环境变量也都会生效。

注解

reindexdb 可能需要多次连接到 PostgreSQL 服务器，每次都要求输入密码。在这种情况下，使用 `~/.pgpass` 文件会很方便。更多信息见第 34.15 节。

示例

要重建数据库 `test` 中的索引：

```
$ reindexdb test
```

要重建名为 `abcd` 的数据库中表 `foo` 的索引和索引 `bar`：

```
$ reindexdb --table=foo --index=bar abcd
```

另见

REINDEX

vacuumdb

vacuumdb — 清理并分析一个 PostgreSQL 数据库

大纲

vacuumdb [*connection-option*...] [*option*...] [-t | --table *table* [(*column* [,...])]] ... [*dbname*]

vacuumdb [*connection-option*...] [*option*...] -a | --all

描述

vacuumdb 是用于清理 PostgreSQL 数据库的工具。vacuumdb 还会生成供 PostgreSQL 查询优化器使用的内部统计信息。

vacuumdb 是 SQL 命令 VACUUM 的包装器。通过该工具或通过访问服务器的其他方式来清理和分析数据库，在效果上没有区别。

选项

vacuumdb 接受下列命令行参数：

-a
--all

清理所有数据库。

[-d] *dbname*
[--dbname=]*dbname*

当未使用 -a/--all 时，指定要清理或分析的数据库名称。如果未指定，则从环境变量 PGDATABASE 中读取数据库名。如果该变量未设置，则使用连接时指定的用户名。*dbname* 可以是连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

-e
--echo

回显 vacuumdb 生成并发送给服务器的命令。

-f
--full

执行“完全”清理。

-F
--freeze

激进地“冻结”元组。

`-j` *njobs*
`--jobs=`*njobs*

通过同时运行 *njobs* 条命令，并行执行清理或分析命令。该选项会缩短处理时间，但也会增加数据库服务器上的负载。

vacuumdb 将打开 *njobs* 个到数据库的连接，因此请确保max_connections 的设置足够高，能够容纳所有这些连接。

注意，如果将此模式与 `-f`（`FULL`）选项一起使用，某些系统目录被并行处理时可能会发生死锁失败。

`-q`
`--quiet`

不显示进度消息。

`-t` *table* `[ (`*column* `[,...]) ]`
`--table=`*table* `[ (`*column* `[,...]) ]`

仅清理或分析 *table*。只有与 `--analyze` 或 `--analyze-only` 选项一起使用时，才能指定列名。可以通过多次指定 `-t` 来清理多个表。

> **提示**
>
> 如果指定了列，你很可能需要在 shell 中转义括号。（见下面的示例。）

`-v`
`--verbose`

在处理过程中输出详细信息。

`-V`
`--version`

输出 vacuumdb 的版本并退出。

`-z`
`--analyze`

同时计算供优化器使用的统计信息。

`-Z`
`--analyze-only`

仅计算供优化器使用的统计信息（不执行清理）。

`--analyze-in-stages`

仅计算供优化器使用的统计信息（不执行清理），类似`--analyze-only`。使用不同的配置设置运行多个（目前为三个）分析阶段，以便更快地生成可用统计信息。

此选项适合分析刚通过恢复转储或 `pg_upgrade` 填充的数据库。它会尽快尝试生成一些统计信息，让数据库能够使用，然后在后续阶段生成完整统计信息。

`-?`
`--help`

显示 vacuumdb 命令行参数的帮助并退出。

vacuumdb 还接受下列用于连接参数的命令行参数：

`-h` *host*
`--host=`*host*

指定服务器所在机器的主机名。如果该值以斜杠开头，则它会被用作 Unix 域套接字目录。

`-p` *port*
`--port=`*port*

指定服务器监听连接所用的 TCP 端口，或本地 Unix 域套接字文件扩展名。

`-U` *username*
`--username=`*username*

连接时使用的用户名。

`-w`
`--no-password`

绝不提示输入密码。如果服务器要求密码认证，而又无法通过其他方式获得密码，例如 `.pgpass` 文件，则连接尝试将失败。此选项可用于无人输入密码的批处理作业和脚本。

`-W`
`--password`

强制 vacuumdb 在连接数据库之前提示输入密码。

这个选项绝非必需，因为如果服务器要求密码认证，vacuumdb 会自动提示输入密码。不过，vacuumdb 会先浪费一次连接尝试，才得知服务器需要密码。在某些情况下，输入 `-W` 以避免这次额外的连接尝试是值得的。

`--maintenance-db=`*dbname*

当使用 `-a`/`--all` 时，连接到该数据库以收集要清理的数据库列表。如果未指定，则使用 `postgres` 数据库；若该数据库不存在，则使用 `template1`。这可以是一个连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。此外，除数据库名本身外，连接字符串中的其他参数在连接到其他数据库时也会被复用。

环境

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

与大多数其他 PostgreSQL 工具一样，此工具也使用 libpq 支持的环境变量（见第 34.14 节）。

诊断

如遇困难，请参见VACUUM 和 psql 中关于潜在问题和错误消息的讨论。数据库服务器必须在目标主机上运行。此外，libpq 前端库所使用的任何默认连接设置和环境变量也都会生效。

注解

vacuumdb 可能需要多次连接到 PostgreSQL 服务器，每次都要询问密码。在这种情况下，使用 `~/.pgpass` 文件会比较方便。更多信息见第 34.15 节。

示例

清理数据库 `test`：

```
$ vacuumdb test
```

清理并分析名为 `bigdb` 的数据库，以供优化器使用：

```
$ vacuumdb --analyze bigdb
```

清理名为 `xyzzy` 的数据库中的单个表 `foo`，并为优化器分析该表中的单个列 `bar`：

```
$ vacuumdb --analyze --verbose --table='foo(bar)' xyzzy
```

另见

VACUUM

PostgreSQL 服务器端应用

本部分包含 PostgreSQL 服务器端应用和支持工具的参考信息。这些命令只有在数据库服务器所在的主机上运行时才有意义。其他工具程序列在 PostgreSQL 客户端应用中。

目录

initdb 1985
pg_archivecleanup 1989
pg_controldata 1991
pg_ctl 1992
pg_resetwal 1998
pg_rewind 2002
pg_test_fsync 2005
pg_test_timing 2006
pg_upgrade 2010
pg_verify_checksums 2018
pg_waldump 2019
postgres 2022
postmaster 2029

initdb

initdb — 创建一个新的 PostgreSQL 数据库集簇

大纲

initdb [*option*...] [--pgdata | -D] *directory*

描述

initdb 创建一个新的 PostgreSQL 数据库集簇。数据库集簇是由单个服务器实例管理的一组数据库。

创建数据库集簇包括创建用于存放数据库数据的目录，生成共享系统目录表（属于整个集簇而不是某个特定数据库的表），以及创建template1 和 postgres 数据库。以后创建新数据库时，会复制template1 数据库中的所有内容。（因此，安装在template1 中的任何东西都会自动复制到以后创建的每个数据库中。）postgres 数据库是一个默认数据库，供用户、工具程序和第三方应用程序使用。

虽然 initdb 会尝试创建指定的数据目录，但如果所需数据目录的父目录归 root 所有，它可能没有足够的权限。要在这种环境中初始化，可先由 root 创建一个空的数据目录，然后用chown 将该目录的所有权赋予数据库用户账户，再用 su 切换为该数据库用户来运行 initdb。

initdb 必须以将拥有服务器进程的用户身份运行，因为服务器需要访问 initdb 创建的文件和目录。由于服务器不能以 root 身份运行，因此也绝不能以 root 身份运行initdb。（实际上它会拒绝这样做。）

出于安全原因，initdb 创建的新集簇默认只有集簇拥有者可以访问。--allow-group-access 选项允许与集簇拥有者同组的任何用户读取集簇中的文件。这对由非特权用户执行备份很有用。

initdb 会初始化数据库集簇的默认区域设置和字符集编码。字符集编码、排序顺序（LC_COLLATE）和字符集分类（LC_CTYPE，例如大写、小写、数字）可以在创建数据库时单独设置。initdb 为 template1 数据库确定这些设置，它们将作为所有其他数据库的默认值。

要更改默认排序顺序或字符集分类，请使用--lc-collate 和 --lc-ctype 选项。使用 C 或 POSIX 之外的排序顺序也会带来性能损失。因此，在运行initdb 时选择正确的区域设置很重要。

其余区域设置类别可以在以后启动服务器时更改。还可以使用 --locale 设置所有区域设置类别的默认值，包括排序顺序和字符集分类。所有服务器区域设置值（lc_*）都可以通过 SHOW ALL 显示。更多细节见第 23.1 节。

要修改默认编码，请使用--encoding。更多细节见第 23.3 节。

选项

`-A` *authmethod*
`--auth=`*authmethod*

此选项指定 `pg_hba.conf` 中本地用户使用的默认认证方法（`host` 和 `local` 行）。`initdb` 会使用指定的认证方法，为非复制连接和复制连接预先填充`pg_hba.conf` 条目。

除非信任系统上的所有本地用户，否则不要使用`trust`。为便于安装，默认值为`trust`。

`--auth-host=`*authmethod*

该选项指定 `pg_hba.conf` 中本地用户通过 TCP/IP 连接时（`host` 行）使用的认证方法。

`--auth-local=`*authmethod*

该选项指定 `pg_hba.conf` 中本地用户通过 Unix 域套接字连接时（`local` 行）使用的认证方法。

`-D` *directory*
`--pgdata=`*directory*

该选项指定数据库集簇应存放的目录。这是 `initdb` 所需的唯一信息，但也可以通过设置 `PGDATA` 环境变量来省去显式写出它；这通常更方便，因为数据库服务器（`postgres`）之后也可以通过同一变量找到数据目录。

`-E` *encoding*
`--encoding=`*encoding*

选择模板数据库的编码。这也会成为以后创建的任何数据库的默认编码，除非在创建时覆盖它。默认值由区域设置推导而来，如果无法推导，则使用`SQL_ASCII`。PostgreSQL 服务器支持的字符集在第 23.3.1 节中介绍。

`-g`
`--allow-group-access`

允许与集簇拥有者同组的用户读取由 `initdb` 创建的所有集簇文件。该选项在 Windows 上会被忽略，因为它不支持 POSIX 风格的组权限。

`-k`
`--data-checksums`

在数据页上使用校验和，帮助检测 I/O 系统造成的、否则可能悄无声息的数据损坏。启用校验和可能会带来明显的性能损失。此选项只能在初始化时设置，以后不能更改。如果启用，就会为所有数据库中的所有对象计算校验和。

`--locale=`*locale*

设置数据库集簇的默认区域设置。如果未指定该选项，区域设置将继承自 `initdb` 运行时所在的环境。区域设置支持见第 23.1 节。

--lc-collate=*locale*
--lc-ctype=*locale*
--lc-messages=*locale*
--lc-monetary=*locale*
--lc-numeric=*locale*
--lc-time=*locale*

类似于 --locale，但只在指定的类别中设置区域设置。

--no-locale

等价于 --locale=C。

-N
--no-sync

默认情况下，initdb 会等待所有文件安全写入磁盘。此选项使 initdb 不等待就返回，速度更快，但意味着此后操作系统崩溃可能会造成数据目录损坏。通常，此选项适用于测试，但不应在创建生产安装时使用。

--pwfile=*filename*

使 initdb 从文件中读取数据库超级用户的密码。文件的第一行会被当作密码。

-S
--sync-only

将所有数据库文件安全写入磁盘，然后退出。这不会执行任何常规initdb 操作。

-T *config*
--text-search-config=*config*

设置默认文本检索配置。更多信息见 default_text_search_config。

-U *username*
--username=*username*

选择数据库超级用户的用户名。默认值是运行 initdb 的有效用户的名称。超级用户的名称本身并不重要，不过即使操作系统用户名称不同，也可以选择沿用惯常的名称 postgres。

-W
--pwprompt

使 initdb 提示输入要赋给数据库超级用户的密码。如果不打算使用密码认证，这一点并不重要。否则，在设置密码之前将无法使用密码认证。

-X *directory*
--waldir=*directory*

该选项指定预写式日志（WAL）应存放的目录。

--wal-segsize=*size*

设置 WAL 段大小，单位为兆字节。这是 WAL 日志中每个单独文件的大小。默认大小为 16 兆字节。该值必须是 1 到 1024（兆字节）之间的 2 的幂。此选项只能在初始化期间设置，以后不能更改。

调整该大小可能有助于控制 WAL 日志传送或归档的粒度。此外，在 WAL 量很大的数据库中，每个目录中的 WAL 文件数量可能会成为性能和管理问题。增大 WAL 文件大小会减少 WAL 文件数量。

还可以使用以下较少用到的选项：

`-d`
`--debug`

打印引导后端的调试输出，以及少量普通用户通常不感兴趣的其他消息。引导后端是 `initdb` 用来创建系统目录表的程序。该选项会产生大量极其乏味的输出。

`-L` *directory*

指定 `initdb` 初始化数据库集簇时应到哪里查找其输入文件。通常不需要这样做。若需要显式指定其位置，系统会提示。

`-n`
`--no-clean`

默认情况下，如果`initdb` 发现某个错误使其无法完整创建数据库集簇，就会删除它在发现无法完成任务之前可能已创建的所有文件。该选项会禁止这种清理，因此对调试有用。

其他选项：

`-V`
`--version`

打印 initdb 版本并退出。

`-?`
`--help`

显示有关 initdb 命令行参数的帮助并退出。

环境

`PGDATA`

指定数据库集簇应存放的目录；可使用 `-D` 选项覆盖。

`TZ`

指定所创建数据库集簇的默认时区。该值应为完整的时区名称（见第 8.5.3 节）。

与大多数其他 PostgreSQL 工具程序一样，此工具也使用libpq 支持的环境变量（参见第 34.14 节）。

注解

也可以通过 `pg_ctl initdb` 调用 `initdb`。

另见

pg_ctl, postgres

pg_archivecleanup

pg_archivecleanup — 清理 PostgreSQL WAL 归档文件

大纲

pg_archivecleanup [*option*...] *archivelocation* *oldestkeptwalfile*

描述

pg_archivecleanup 设计为在服务器作为备库运行时用作 archive_cleanup_command，以清理 WAL 文件归档（见第 26.2 节）。pg_archivecleanup 也可作为独立程序使用，以清理 WAL 文件归档。

要配置备库使用 pg_archivecleanup，请将以下内容放入其 recovery.conf 配置文件中：

```
archive_cleanup_command = 'pg_archivecleanup archivelocation %r'
```

其中，*archivelocation* 是应从中移除 WAL 段文件的目录。

当在 archive_cleanup_command 中使用时，逻辑上早于 %r 参数值的所有 WAL 文件都会从 *archivelocation* 中移除。这样既能将需要保留的文件数量降到最低，又能保留崩溃后重启能力。如果 *archivelocation* 只是该特定备库的临时暂存区，那么使用此参数是合适的；但如果 *archivelocation* 打算作为长期 WAL 归档区域，或者有多个备库正从同一归档位置恢复，则不适合使用此参数。

作为独立程序使用时，逻辑上早于 *oldestkeptwalfile* 的所有 WAL 文件都会从 *archivelocation* 中移除。在此模式下，如果指定的是 .partial 或 .backup 文件名，则仅使用其文件名前缀作为 *oldestkeptwalfile*。这样处理 .backup 文件名后，就可以无误地删除某个特定基础备份之前归档的所有 WAL 文件。例如，下例会移除所有早于 WAL 文件名 000000010000003700000010 的文件：

```
pg_archivecleanup -d archive 000000010000003700000010.00000020.backup

pg_archivecleanup:  keep WAL file "archive/000000010000003700000010" and later
pg_archivecleanup:  removing file "archive/00000001000000370000000F"
pg_archivecleanup:  removing file "archive/00000001000000370000000E"
```

pg_archivecleanup 假定 *archivelocation* 是服务器属主用户可读写的目录。

选项

pg_archivecleanup 接受以下命令行参数：

-d

在 stderr 上输出大量调试日志。

`-n`

在 `stdout` 上打印原本会被移除的文件名（执行试运行）。

`-V`
`--version`

输出 pg_archivecleanup 的版本号并退出。

`-x` *`extension`*

指定一个扩展名；在判断文件是否应删除之前，会先从所有文件名中去掉该扩展名。这通常适用于清理由于存储时经过压缩、因而被压缩程序附加了扩展名的归档。例如：`-x .gz`。

`-?`
`--help`

显示有关 pg_archivecleanup 命令行参数的帮助并退出。

注解

pg_archivecleanup 设计为在作为独立工具使用时与 PostgreSQL 8.0 及更高版本配合工作，或在作为归档清理命令使用时与 PostgreSQL 9.0 及更高版本配合工作。

pg_archivecleanup 采用 C 编写，源代码易于修改，并且专门划出了若干区域供你按自身需要修改。

示例

在 Linux 或 Unix 系统上，你可以这样使用：

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r 2>>cleanup.log'
```

其中归档目录实际上位于备库上，因此 `archive_command` 通过 NFS 访问它，但这些文件对备库来说是本地的。这将会：

- 在 `cleanup.log` 中产生调试输出
- 从归档目录中移除不再需要的文件

另见

pg_standby

pg_controldata

pg_controldata — 显示 PostgreSQL 数据库集簇的控制信息

大纲

`pg_controldata` [*option*] [[`-D` | `--pgdata`]*datadir*]

描述

`pg_controldata` 会打印在 `initdb` 期间初始化的信息，例如系统目录版本。它还会显示有关预写式日志和检查点处理的信息。这些信息是整个集簇范围内的，不针对任何单个数据库。

该工具只能由初始化该集簇的用户运行，因为它需要对数据目录具有读取权限。可以在命令行中指定数据目录，也可以使用环境变量`PGDATA`。该工具支持 `-V` 和 `--version` 选项，用于输出 pg_controldata 的版本并退出。它还支持`-?`和 `--help` 选项，用于输出可用参数。

环境

`PGDATA`

默认数据目录位置

pg_ctl

pg_ctl — 初始化、启动、停止或控制PostgreSQL 服务器

大纲

`pg_ctl init[db]` [`-D` *`datadir`*] [`-s`] [`-o` *`initdb-options`*]

`pg_ctl start` [`-D` *`datadir`*] [`-l` *`filename`*] [`-W`] [`-t` *`seconds`*] [`-s`] [`-o` *`options`*] [`-p` *`path`*] [`-c`]

`pg_ctl stop` [`-D` *`datadir`*] [`-m` `s[mart]` | `f[ast]` | `i[mmediate]`] [`-W`] [`-t` *`seconds`*] [`-s`]

`pg_ctl restart` [`-D` *`datadir`*] [`-m` `s[mart]` | `f[ast]` | `i[mmediate]`] [`-W`] [`-t` *`seconds`*] [`-s`] [`-o` *`options`*] [`-c`]

`pg_ctl reload` [`-D` *`datadir`*] [`-s`]

`pg_ctl status` [`-D` *`datadir`*]

`pg_ctl promote` [`-D` *`datadir`*] [`-W`] [`-t` *`seconds`*] [`-s`]

`pg_ctl kill` *`signal_name process_id`*

在 Microsoft Windows 上，还支持：

`pg_ctl register` [`-D` *`datadir`*] [`-N` *`servicename`*] [`-U` *`username`*] [`-P` *`password`*] [`-S` `a[uto]` | `d[emand]`] [`-e` *`source`*] [`-W`] [`-t` *`seconds`*] [`-s`] [`-o` *`options`*]

`pg_ctl unregister` [`-N` *`servicename`*]

描述

pg_ctl 是一个实用工具，用于初始化PostgreSQL 数据库集簇，启动、停止或重启PostgreSQL 数据库服务器（postgres），或者显示正在运行服务器的状态。虽然服务器也可以手工启动，但 pg_ctl 将重定向日志输出、正确地与终端和进程组脱离等任务封装了起来。它还提供了便于实施受控关闭的选项。

`init` 或 `initdb` 模式会创建一个新的 PostgreSQL 数据库集簇，也就是由单个服务器实例管理的一组数据库。该模式会调用`initdb` 命令。详见 initdb。

`start` 模式会启动一个新服务器。服务器在后台启动，其标准输入连接到`/dev/null`（在 Windows 上则是 `nul`）。在类 Unix 系统上，默认情况下，服务器的标准输出和标准错误会被发送到pg_ctl 的标准输出（而不是标准错误）。因此，pg_ctl 的标准输出应被重定向到文件，或者通过管道传给另一个进程，例如rotatelogs 这样的日志轮转程序；否则 `postgres` 会在后台将其输出写入控制终端，并且不会脱离 shell 的进程组。在 Windows 上，默认情况下，服务器的标准输出和标准错误会被发送到终端。使用`-l` 将服务器输出追加到日志文件，可以改变这些默认行为。建议使用`-l` 或输出重定向。

`stop` 模式会关闭在指定数据目录中运行的服务器。可以使用 `-m` 选项选择三种不同的关闭方法。“Smart” 模式不允许新连接，然后等待所有现有客户端断开，以及所有在线备份结束。如果服务器处于热备状态，那么在所有客户端断开后，恢复和流复制都会终止。“Fast” 模式（默

认值）不等待客户端断开，并会终止正在进行的在线备份。所有活动事务都会回滚，客户端会被强制断开，然后服务器关闭。“Immediate”模式会立即中止所有服务器进程，而不执行干净关闭。这样会导致服务器在下次启动时进入一次崩溃恢复周期。

`restart` 模式实际上就是先执行停止再执行启动。这使得可以修改 `postgres` 的命令行选项，或者修改那些不重启服务器就无法更改的配置文件选项。如果服务器启动时在命令行中使用了相对路径，那么除非在与服务器上次启动时相同的当前目录中执行pg_ctl，否则`restart` 可能失败。

`reload` 模式只是向 `postgres` 服务器进程发送一个 SIGHUP 信号，使其重新读取配置文件（`postgresql.conf`、`pg_hba.conf` 等）。这样就可以修改那些无需完全重启服务器即可生效的配置文件选项。

`status` 模式会检查指定数据目录中是否有服务器正在运行。若有，则显示服务器的PID 以及调用它时使用的命令行选项。若服务器未运行，pg_ctl 返回退出状态 3。若未指定一个可访问的数据目录，pg_ctl 返回退出状态 4。

`promote` 模式会指示在指定数据目录中运行的备库结束备库模式，并开始进行读写操作。

`kill` 模式向指定进程发送信号。这在没有内置 kill 命令的 Microsoft Windows 上尤其有用。使用`--help` 可查看受支持的信号名称列表。

`register` 模式会将 PostgreSQL 服务器注册为 Microsoft Windows 上的系统服务。`-S` 选项允许选择服务启动类型，可以是“auto”（系统启动时自动启动服务）或“demand”（按需启动服务）。

`unregister` 模式会在 Microsoft Windows 上注销一个系统服务。这会撤销 `register` 命令的效果。

选项

`-c`
`--core-files`

在支持的平台上，通过解除对核心转储文件施加的任何软资源限制，尝试允许服务器在崩溃时生成核心转储文件。这样就可以从失败的服务器进程中获得栈跟踪，从而有助于调试或诊断问题。

`-D` *`datadir`*
`--pgdata=`*`datadir`*

指定数据库配置文件所在的文件系统位置。若省略此选项，则使用环境变量`PGDATA`。

`-l` *`filename`*
`--log=`*`filename`*

将服务器日志输出追加到 *`filename`*。如果该文件不存在，就会创建它。umask 被设置为 077，因此默认情况下其他用户无法访问该日志文件。

`-m` *`mode`*
`--mode=`*`mode`*

指定关闭模式。*`mode`* 可以是 `smart`、`fast` 或 `immediate`，也可以是这三者之一的首字母。若省略此选项，则默认值为`fast`。

`-o` *options*
`--options=`*options*

指定要直接传递给 `postgres` 命令的选项。`-o` 可以指定多次，给出的所有选项都会被传递过去。

通常应将 *options* 放在单引号或双引号中，以确保它们作为一个整体被传递。

`-o` *initdb-options*
`--options=`*initdb-options*

指定要直接传递给 `initdb` 命令的选项。`-o` 可以指定多次，给出的所有选项都会被传递过去。

通常应将 *initdb-options* 放在单引号或双引号中，以确保它们作为一个整体被传递。

`-p` *path*

指定 `postgres` 可执行程序的位置。默认情况下，`postgres` 可执行程序取自与 `pg_ctl` 相同的目录；如果那里没有，则取自硬编码的安装目录。除非采用了某些非常规方式，并收到找不到`postgres` 可执行程序的错误，否则通常不需要使用此选项。

在 `init` 模式中，此选项同样指定`initdb` 可执行程序的位置。

`-s`
`--silent`

只打印错误，不打印信息性消息。

`-t` *seconds*
`--timeout=`*seconds*

指定等待某个操作完成时最多等待多少秒（见选项 `-w`）。默认值为环境变量 `PGCTLTIMEOUT` 的值；如果未设置该环境变量，则默认为 60 秒。

`-V`
`--version`

打印 pg_ctl 的版本并退出。

`-w`
`--wait`

等待操作完成。`start`、`stop`、`restart`、`promote` 和 `register` 模式支持此选项，而且对这些模式来说它是默认行为。

在等待期间，`pg_ctl` 会反复检查服务器的 PID 文件，并在两次检查之间短暂休眠。当PID 文件表明服务器已准备好接受连接时，启动就被视为完成。当服务器移除PID 文件时，关闭就被视为完成。`pg_ctl` 会根据启动或关闭是否成功返回相应的退出代码。

如果操作未能在超时时间内完成（见选项 `-t`），那么`pg_ctl` 会以非零退出状态退出。不过请注意，该操作仍可能在后台继续进行并最终成功。

-W
--no-wait

不等待操作完成。这与选项 -w 相反。

如果禁用了等待，将触发所请求的操作，但不会反馈其是否成功。在这种情况下，需要借助服务器日志文件或外部监控系统来检查该操作的进展及成败。

在 PostgreSQL 先前的发行版中，除了stop 模式之外，这曾是默认行为。

-?
--help

显示有关 pg_ctl 命令行参数的帮助并退出。

如果指定的某个选项本身有效，但与所选操作模式无关，pg_ctl 会忽略它。

用于 Windows 的选项

-e *source*

指定以 Windows 服务方式运行时，pg_ctl 写入事件日志所使用的事件源名称。默认值为 PostgreSQL。注意，这只控制由pg_ctl 本身发送的消息；服务器一旦启动，就会使用其 event_source 参数指定的事件源。如果服务器在启动的很早阶段、该参数尚未设置之前就失败，它也可能使用默认事件源名称PostgreSQL 记录日志。

-N *servicename*

要注册的系统服务名称。该名称会同时用作服务名和显示名。默认值为 PostgreSQL。

-P *password*

用于运行该服务的用户密码。

-S *start-type*

系统服务的启动类型。*start-type* 可以是 auto、demand，或者两者之一的首字母。若省略此选项，则默认值为auto。

-U *username*

用于运行该服务的用户名。对于域用户，请使用DOMAIN\username 格式。

环境

PGCTLTIMEOUT

等待启动或关闭完成时，默认的等待秒数上限。如果未设置，默认值为 60 秒。

PGDATA

默认数据目录位置。

大多数 pg_ctl 模式都需要知道数据目录的位置，因此除非设置了PGDATA，否则-D 选项就是必需的。

pg_ctl 与大多数其他 PostgreSQL 工具一样，也使用libpq 所支持的环境变量（参见第 34.14 节）。

其他会影响服务器的变量，见 postgres。

文件

postmaster.pid

pg_ctl 会检查数据目录中的这个文件，以确定服务器当前是否正在运行。

postmaster.opts

如果该文件存在于数据目录中，pg_ctl（在 restart 模式下）会将该文件的内容作为选项传递给 postgres，除非被-o 选项覆盖。该文件的内容也会在 status 模式下显示出来。

示例

启动服务器

要启动服务器并等待其开始接受连接：

```
$ pg_ctl start
```

要使用端口 5433 启动服务器，并在未启用fsync 的情况下运行，可使用：

```
$ pg_ctl -o "-F -p 5433" start
```

停止服务器

要停止服务器，可使用：

```
$ pg_ctl stop
```

-m 选项允许控制服务器如何关闭：

```
$ pg_ctl stop -m smart
```

重启服务器

重启服务器几乎等同于先停止服务器再重新启动，只不过默认情况下，pg_ctl 会保存并重用传给前一个运行实例的命令行选项。要使用与之前相同的选项重启服务器，可使用：

```
$ pg_ctl restart
```

但是，如果指定了-o，就会替换先前的所有选项。要使用端口 5433 重启服务器，并在重启时禁用fsync，可使用：

```
$ pg_ctl -o "-F -p 5433" restart
```

显示服务器状态

下面是 pg_ctl 状态输出的示例：

```
$ pg_ctl status

pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"
```

第二行是在重启模式下会调用的命令。

参见

initdb, postgres

pg_resetwal

pg_resetwal — 重置 PostgreSQL 数据库集簇的预写式日志及其他控制信息

大纲

pg_resetwal [-f | --force] [-n | --dry-run] [*option*...] [-D | --pgdata]*datadir*

描述

pg_resetwal 会清除预写式日志（WAL），并可选择性地重置存储在 pg_control 文件中的某些其他控制信息。如果这些文件已经损坏，有时就需要使用此功能。只有当服务器因这类损坏而无法启动时，才应将其作为最后手段使用。

运行此命令后，服务器应该可以启动；但请记住，由于仅部分提交的事务，数据库中可能含有不一致的数据。应立即转储数据，运行 initdb，然后恢复。恢复后，请检查是否存在不一致并按需修复。

此工具只能由安装服务器的用户运行，因为它需要对数据目录的读写访问权限。出于安全原因，必须在命令行上指定数据目录。pg_resetwal 不使用环境变量 PGDATA。

如果 pg_resetwal 报告无法确定 pg_control 的有效数据，则可以通过指定 -f（强制）选项强行继续。在这种情况下，缺失的数据将以看似合理的值代替。大多数字段预计都能匹配，但对于下一个 OID、下一个事务 ID 及其纪元、下一个多事务 ID 及其偏移量，以及 WAL 起始位置字段，可能仍需要人工协助。这些字段可通过下文讨论的选项来设置。如果无法为所有这些字段确定正确值，仍然可以使用 -f，但恢复得到的数据库必须比平时更受怀疑：立即转储并恢复势在必行。切勿在完成转储之前对数据库执行任何修改数据的操作，因为任何此类操作都很可能使损坏进一步恶化。

选项

-f
--force

如上所述，即使pg_resetwal 无法确定 pg_control 的有效数据，也强制其继续执行。

-n
--dry-run

-n/--dry-run 选项会让 pg_resetwal 打印从 pg_control 重建出的值以及即将更改的值，然后在不做任何修改的情况下退出。这主要是调试工具，但在真正允许 pg_resetwal 执行之前，也可用作合理性检查。

-V
--version

显示版本信息，然后退出。

-?
--help

显示帮助，然后退出。

以下选项仅在 pg_resetwal 无法通过读取 pg_control 确定适当值时才需要使用。安全值可按下述方法确定。对于接受数字参数的选项，可使用前缀 0x 指定十六进制值。

-c *xid*,*xid*
--commit-timestamp-ids=*xid*,*xid*

手工设置可检索提交时间的最旧和最新事务 ID。

可检索提交时间的最旧事务 ID 的安全值（第一部分），可以通过查看数据目录下 pg_commit_ts 目录中数值最小的文件名来确定。相反，可检索提交时间的最新事务 ID 的安全值（第二部分），可以通过查看同一目录中数值最大的文件名来确定。文件名均为十六进制。

-e *xid_epoch*
--epoch=*xid_epoch*

手工设置下一个事务 ID 的纪元。

事务 ID 纪元实际上并未存储在数据库中的任何地方，除了由 pg_resetwal 设置的这个字段，因此就数据库本身而言，任何值都可用。可能需要调整该值，以确保 Slony-I 和 Skytools 等复制系统正常工作 – 如果确有需要，应该可以从下游复制数据库的状态中获得合适的值。

-l *walfile*
--next-wal-file=*walfile*

通过指定下一个 WAL 段文件的名称，手工设置 WAL 起始位置。

下一个 WAL 段文件的名称应大于当前数据目录下 pg_wal 目录中现存的任意 WAL 段文件名。这些名称同样采用十六进制，并由三个部分组成。第一部分是“时间线 ID”，通常应保持不变。例如，如果 000000010000003200000004A 是 pg_wal 中最大的条目，则应使用 -l 000000010000003200000004B 或更大的值。

请注意，在使用非默认 WAL 段大小时，WAL 文件名中的数字与系统函数和系统视图报告的 LSN 不同。该选项接受的是 WAL 文件名，而不是 LSN。

注意

pg_resetwal 本身会查看 pg_wal 中的文件，并选择一个超出现有最后一个文件名的默认 -l 设置。因此，只有在明确知道存在当前不在 pg_wal 中的 WAL 段文件（例如离线归档中的条目），或者 pg_wal 的内容已经完全丢失时，才需要手工调整 -l。

-m *mxid*,*mxid*
--multixact-ids=*mxid*,*mxid*

手工设置下一个和最旧的多事务 ID。

下一个多事务 ID 的安全值（第一部分）可以通过查看数据目录下 pg_multixact/offsets 目录中数值最大的文件名，先加一，再乘以 65536 (0x10000) 来确定。相反，最旧的多事务 ID 的安全值（-m 的第二部分）可以通过查看同一目录中数值最小的文件名并乘以 65536 来确定。文件名为十六进制，因此最简单的做法通常是用十六进制指定选项值，并在后面追加四个零。

-o *oid*
--next-oid=*oid*

手工设置下一个 OID。

并没有同样简单的方法来确定一个超出数据库中最大 OID 的下一个 OID，不过幸运的是，下一个 OID 的设置是否精确并不关键。

-O *mxoff*
--multixact-offset=*mxoff*

手工设置下一个多事务偏移量。

安全值可以通过查看数据目录下 pg_multixact/members 目录中数值最大的文件名，先加一，再乘以 52352 (0xCC80) 来确定。文件名是十六进制。这里没有像其他选项那样通过追加零即可得到结果的简单方法。

--wal-segsize=*wal_segment_size*

设置新的 WAL 段大小，单位为兆字节。该值必须是 1 到 1024（兆字节）之间的 2 的幂。更多信息请参阅initdb 的同名选项。

> **注意**
>
> 虽然 pg_resetwal 会把 WAL 起始位置设置到超出现有最新 WAL 段文件的位置，但某些段大小变更可能导致旧 WAL 文件名被重新使用。如果 WAL 文件名重叠会给归档策略带来问题，建议将 -l 与此选项配合使用，手工设置 WAL 起始位置。

-u *xid*
--oldest-transaction-id=*xid*

手工设置最旧的未冻结事务 ID。

安全值可以通过查看数据目录下 pg_xact 目录中数值最小的文件名，再乘以 1048576 (0x100000) 来确定。请注意，文件名是十六进制。通常，选项值也用十六进制指定最为简单。例如，如果 0007 是 pg_xact 中最小的条目，那么 -u 0x700000 就可以使用（五个尾随零提供了正确的乘数）。

-x *xid*
--next-transaction-id=*xid*

手工设置下一个事务 ID。

安全值可以通过查看数据目录下 pg_xact 目录中数值最大的文件名，先加一，再乘以 1048576 (0x100000) 来确定。请注意，文件名是十六进制。通常，选项值也用十六进制指定最为简单。例如，如果 0011 是 pg_xact 中最大的条目，那么 -x 0x1200000 就可以使用（五个尾随零提供了正确的乘数）。

注解

服务器运行期间不得使用此命令。如果在数据目录中发现服务器锁文件，`pg_resetwal` 将拒绝启动。如果服务器曾崩溃，可能遗留锁文件；在这种情况下，可以删除该锁文件以允许 `pg_resetwal` 运行。但在这样做之前，务必再三确认没有任何服务器进程仍在运行。

`pg_resetwal` 仅适用于相同主版本的服务器。

另见

pg_controldata

pg_rewind

pg_rewind — 将一个 PostgreSQL 数据目录与另一个从其分叉而来的数据目录同步

大纲

```
pg_rewind [option...] { -D | --target-pgdata } directory { --source-
pgdata=directory | --source-server=connstr }
```

描述

pg_rewind 是一个工具，用于在同一 PostgreSQL 集簇的两份副本时间线发生分叉后，将其中一份重新同步到另一份。典型场景是故障切换后，让旧主库重新上线，作为跟随新主库的备库。

其结果相当于用源数据目录替换目标数据目录。对于关系文件，只复制发生变化的块；其他所有文件（包括配置文件）都整体复制。pg_rewind 相较于获取新的基础备份或使用 rsync 等工具，其优势在于pg_rewind 无需通读集簇中未发生变化的块。当数据库很大而两个集簇之间只有少量块不同时，这使它快得多。

pg_rewind 会检查源集簇和目标集簇的时间线历史，以确定它们发生分叉的位置，并且要求在目标集簇的`pg_wal` 目录中能够找到一直追溯到该分叉点的 WAL。分叉点可能位于目标时间线、源时间线，或者二者共同的祖先时间线上。在典型的故障切换场景中，目标集簇会在分叉后不久关闭，因此这通常不是问题；但如果目标集簇在分叉后又运行了很长时间，旧的 WAL 文件可能已经不存在。在这种情况下，你可以手动把它们从 WAL 归档复制到`pg_wal` 目录。pg_rewind 的用途并不限于故障切换，例如，备库可以被提升，运行一些写事务，然后再回卷，重新成为备库。

运行 pg_rewind 之后首次启动目标服务器时，它会进入恢复模式，并重放源服务器在分叉点之后生成的全部 WAL。如果运行pg_rewind 时，源服务器上的某些 WAL 已经不可用，因此无法由 pg_rewind 会话复制，那么在启动目标服务器时必须能够获取这些 WAL。这可以通过在目标数据目录中创建`recovery.conf` 文件，并在其中设置合适的`restore_command` 来实现。

pg_rewind 要求目标服务器满足以下条件之一：要么在 `postgresql.conf` 中启用了 wal_log_hints 选项，要么在用initdb 初始化集簇时启用了数据校验和（默认即如此）。此外，full_page_writes 也必须设置为 `on`，不过它默认已启用。

> **警告**
>
> 如果 pg_rewind 在处理过程中失败，目标数据目录很可能已经处于无法恢复的状态。在这种情况下，建议重新获取一份全新的备份。
>
> 如果 pg_rewind 发现某些文件无法直接写入，它就会立即失败。例如，当源服务器和目标服务器对只读 SSL 密钥和证书使用相同的文件映射时，就会发生这种情况。如果目标服务器上存在这类文件，建议在运行pg_rewind 之前将其移除。回卷完成后，其中一些文件可能已经从源复制过来，这种情况下可能需要删除复制过来的数据，并恢复回卷前使用的那组链接。

选项

pg_rewind 接受以下命令行参数：

`-D` *`directory`*
`--target-pgdata=`*`directory`*

此选项指定要与源同步的目标数据目录。在运行 pg_rewind 之前，目标服务器必须已正常关闭。

`--source-pgdata=`*`directory`*

指定源服务器数据目录的文件系统路径，以便将目标与之同步。此选项要求源服务器已正常关闭。

`--source-server=`*`connstr`*

指定一个 libpq 连接字符串，用于连接源PostgreSQL 服务器，以便将目标与之同步。该连接必须是普通连接（非复制连接），所用角色要么拥有足够的权限以执行pg_rewind 在源服务器上使用的函数（详见注解部分），要么是超级用户角色。此选项要求源服务器正在运行且不处于恢复模式。

`-n`
`--dry-run`

执行除实际修改目标目录之外的所有操作。

`-P`
`--progress`

启用进度报告。打开该选项后，在从源集簇复制数据时会给出大致的进度信息。

`--debug`

打印详细的调试输出，这些输出主要对调试pg_rewind 的开发人员有用。

`-V`
`--version`

显示版本信息，然后退出。

`-?`
`--help`

显示帮助，然后退出。

环境

在使用 `--source-server` 选项时，pg_rewind 也会使用 libpq 支持的环境变量（见第 34.14 节）。

注解

当使用在线集簇作为源执行 pg_rewind 时，可以使用一个具备足够权限以执行pg_rewind 在源集簇上所用函数的角色，而无需使用超级用户。下面说明如何创建这样一个角色，这里将其命名为rewind_user：

```
CREATE USER rewind_user LOGIN;
GRANT EXECUTE ON function pg_catalog.pg_ls_dir(text, boolean, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_stat_file(text, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text, bigint, bigint, boolean) TO
 rewind_user;
```

使用最近刚提升的在线集簇作为源来执行 pg_rewind 时，必须在提升后执行CHECKPOINT，使其控制文件反映最新的时间线信息。pg_rewind 会使用这些信息检查能否利用指定的源集簇回卷目标集簇。

工作原理

基本思路是将源集簇中所有文件系统级别的更改复制到目标集簇：

1. 从源集簇的时间线历史与目标集簇分叉这一点之前的最后一个检查点开始，扫描目标集簇的 WAL 日志。对于每条 WAL 记录，记录它所触及的每个数据块。这样就能得到一份列表，其中包含源集簇分叉出去之后，目标集簇中所有发生过更改的数据块。

2. 将所有这些发生过更改的块从源集簇复制到目标集簇，可以使用直接文件系统访问（--source-pgdata）或 SQL（--source-server）。

3. 将所有其他文件，例如pg_xact 和配置文件，从源集簇复制到目标集簇（关系文件除外）。与基础备份类似，目录pg_dynshmem/、pg_notify/、pg_replslot/、pg_serial/、pg_snapshots/、pg_stat_tmp/和 pg_subtrans/中的内容不会从源集簇复制。任何以 pgsql_tmp 开头的文件或目录都会被省略，backup_label、tablespace_map、pg_internal.init、postmaster.opts 和 postmaster.pid 也会被省略。

4. 从故障切换时创建的检查点开始，应用源集簇的 WAL。（严格来说，pg_rewind 并不应用 WAL，它只是创建一个备份标签文件，让PostgreSQL 在启动时重放从该检查点起的全部 WAL。）

pg_test_fsync

pg_test_fsync — 为 PostgreSQL 确定最快的 `wal_sync_method`

大纲

`pg_test_fsync` [*option*...]

描述

pg_test_fsync 旨在让你较为合理地了解，在你的具体系统上哪一种wal_sync_method 最快，并且在识别出 I/O 问题时提供诊断信息。不过，pg_test_fsync 显示出的差异未必会对实际数据库吞吐量产生显著影响，特别是因为许多数据库服务器的速度瓶颈并不在其预写式日志上。pg_test_fsync 会为每种 `wal_sync_method` 报告以微秒计的平均文件同步操作时间，这也可用于为优化commit_delay 的取值提供参考。

选项

pg_test_fsync 接受下列命令行选项：

`-f`
`--filename`

指定用于写入测试数据的文件名。该文件应位于 `pg_wal` 目录所在或将要放置的同一文件系统中。（`pg_wal` 包含 WAL 文件。）默认值是当前目录中的 `pg_test_fsync.out`。

`-s`
`--secs-per-test`

指定每项测试的秒数。每项测试用时越长，测试结果就越精确，但运行完成所需的时间也越长。默认值是 5 秒，这使程序能够在不到 2 分钟内完成。

`-V`
`--version`

打印 pg_test_fsync 版本并退出。

`-?`
`--help`

显示有关 pg_test_fsync 命令行参数的帮助信息并退出。

另见

postgres

pg_test_timing

pg_test_timing — 测量计时开销

大纲

```
pg_test_timing [option...]
```

描述

pg_test_timing 是一个用于测量系统计时开销的工具，并确认系统时间不会倒退。收集计时数据较慢的系统，可能会给出不够精确的 `EXPLAIN ANALYZE` 结果。

选项

pg_test_timing 接受下列命令行选项：

`-d *duration*`
`--duration=*duration*`

指定测试持续时间，以秒为单位。持续时间越长，精度会略有提高，也更可能发现系统时钟倒退的问题。默认测试持续时间为 3 秒。

`-V`
`--version`

打印 pg_test_timing 版本并退出。

`-?`
`--help`

显示有关 pg_test_timing 命令行参数的帮助信息并退出。

用法

结果解读

良好的结果应显示，大多数（>90%）单次计时调用的耗时都小于 1 微秒。每轮循环的平均开销会更低，低于 100 纳秒。下面这个来自一台使用 TSC 时钟源的 Intel i7-860 系统的示例，展示了极佳的性能：

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 ns
Histogram of timing durations:
  < us   % of total      count
     1     96.40465   80435604
     2      3.59518    2999652
     4      0.00015        126
     8      0.00002         13
    16      0.00000          2
```

注意，每轮循环时间与直方图使用的单位不同。循环的解析度可以达到几纳秒（ns），而单次计时调用的解析度只能达到 1 微秒（us）。

测量执行器计时开销

当查询执行器使用 `EXPLAIN ANALYZE` 运行语句时，除了显示汇总信息之外，还会对各个操作分别计时。可以使用psql 程序统计行数，以检查你的系统上的这类开销：

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

在所测的 i7-860 系统上，普通计数查询耗时 9.8 ms，而 `EXPLAIN ANALYZE` 版本耗时 16.6 ms，两者都处理了略多于 100,000 行。这 6.8 ms 的差异意味着每行的计时开销为 68 ns，大约是 pg_test_timing 估计值的两倍。即便是这样相对较小的开销，也会让完整计时的计数语句耗时增加近 70%。对于更复杂的查询，计时开销带来的问题就不会这么明显。

更改时钟源

在某些较新的 Linux 系统上，可以随时更改用于收集计时数据的时钟源。第二个示例展示了在产生上述快速结果的同一台系统上，切换到较慢的 acpi_pm 时钟源可能导致的性能下降：

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 ns
Histogram of timing durations:
  < us   % of total      count
     1     27.84870    1155682
     2     72.05956    2990371
     4      0.07810       3241
     8      0.01357        563
    16      0.00007          3
```

在这种配置下，上面的示例EXPLAIN ANALYZE 需要 115.9 ms。这意味着计时开销达到 1061 ns，同样只是本工具直接测得数值的一个小倍数。如此之大的计时开销说明，实际查询本身只占已统计时间的一小部分，其余大多耗费在开销上。在这种配置下，任何包含大量计时操作的 EXPLAIN ANALYZE 总计值，都会因计时开销而被显著抬高。

FreeBSD 也允许动态更改时钟源，并会记录启动期间所选计时器的信息：

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

其他系统可能只允许在启动时设置时钟源。在较旧的 Linux 系统上，内核设置 “clock” 是进行此类更改的唯一方式。即使在某些较新的系统上，你能看到的时钟源选项也可能只有 “jiffies” 。jiffies 是较早的 Linux 软件时钟实现，如果底层计时硬件足够快，它也能提供良好的解析度，如下例所示：

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
  < us   % of total      count
     1     90.23734   27694571
     2      9.75277    2993204
     4      0.00981       3010
     8      0.00007         22
    16      0.00000          1
    32      0.00000          1
```

时钟硬件和计时准确性

在计算机上收集精确的计时信息，通常依赖精度各异的硬件时钟。对于某些硬件，操作系统几乎可以将系统时钟时间直接传递给程序。系统时钟也可能来自一种只提供计时中断、按已知时间间隔周期性地产生滴答信号的芯片。无论哪种情况，操作系统内核都会提供一个屏蔽这些细节的时钟源。不过，这种时钟源的精度以及返回结果的速度，都取决于底层硬件。

不准确的计时可能导致系统不稳定。对时钟源的任何更改都应十分谨慎地测试。操作系统的默认设置有时会偏向可靠性，而不是追求最佳精度。如果你使用的是虚拟机，还应了解与其兼容的推荐时钟源。虚拟硬件在模拟计时器时会面临额外困难，厂商通常还会针对不同操作系统给出特定的设置建议。

时间戳计数器（TSC）时钟源是当前一代 CPU 上可用的最精确时钟源。当操作系统支持它且 TSC 时钟本身可靠时，它是跟踪系统时间的首选方式。TSC 也可能因多种原因无法提供精确的计时源，从而变得不可靠。较旧的系统中，TSC 时钟可能会随 CPU 温度变化，因此不适合用于计时。在某些较旧的多核 CPU 上使用 TSC，还可能导致不同核心报告的时间彼此不一致。这会引起时间

倒退，而本程序正是用来检查这类问题的。即便在最新的系统上，若节能配置过于激进，也可能无法提供精确的 TSC 计时。

较新的操作系统可能会检测已知的 TSC 问题，并在发现时切换到更慢但更稳定的时钟源。如果你的系统支持 TSC 计时却没有默认使用它，那么禁用它很可能是有充分理由的。某些操作系统也可能无法正确检测所有潜在问题，或者即使已知 TSC 不准确，仍允许继续使用它。

如果系统上有高精度事件计时器（HPET），并且 TSC 不准确，那么 HPET 就是系统更倾向使用的计时器。计时器芯片本身可编程，最高可提供 100 纳秒的解析度，但在你的系统时钟中未必能看到这么高的准确性。

高级配置与电源接口（ACPI）提供了一种电源管理（PM）计时器，Linux 将其称为 acpi_pm。由 acpi_pm 派生的时钟在最佳情况下可提供 300 纳秒的解析度。

较旧 PC 硬件使用的计时器包括 8254 可编程间隔计时器（PIT）、实时时钟（RTC）、高级可编程中断控制器（APIC）计时器以及 Cyclone 计时器。这些计时器的目标解析度是毫秒级。

另见

EXPLAIN

pg_upgrade

pg_upgrade — 升级一个 PostgreSQL 服务器实例

大纲

```
pg_upgrade -b oldbindir -B newbindir -d oldconfigdir -D newconfigdir
[option...]
```

描述

pg_upgrade（以前称为 pg_migrator）允许将存储在 PostgreSQL 数据文件中的数据升级到更新的 PostgreSQL 主版本，而无需执行主版本升级通常所需的数据转储/恢复，例如从 9.5.8 升级到 9.6.4，或从 10.7 升级到 11.2。次版本升级不需要使用它，例如从 9.6.2 升级到 9.6.3 或从 10.1 升级到 10.2。

PostgreSQL 主版本会定期增加新特性，这些特性常常会改变系统表的布局，但内部数据存储格式很少改变。pg_upgrade 利用这一点，通过创建新的系统表并直接重用旧的用户数据文件来快速完成升级。如果未来某个主版本改变了数据存储格式，致使旧数据格式无法读取，那么 pg_upgrade 就不能用于这类升级。（社区会尽量避免这种情况。）

pg_upgrade 会尽最大努力确保新旧集簇在二进制上兼容，例如会检查兼容的编译时设置，包括 32/64 位二进制程序。任何外部模块也同样必须二进制兼容，这一点很重要，但 pg_upgrade 无法检查。

pg_upgrade 支持从 8.4.X 及更高版本升级到当前 PostgreSQL 主版本，包括快照版和 beta 版。

选项

pg_upgrade 接受下列命令行参数：

`-b *bindir*`
`--old-bindir=*bindir*`

旧 PostgreSQL 可执行文件目录；环境变量 `PGBINOLD`

`-B *bindir*`
`--new-bindir=*bindir*`

新 PostgreSQL 可执行文件目录；环境变量 `PGBINNEW`

`-c`
`--check`

仅检查集簇，不更改任何数据

`-d *configdir*`
`--old-datadir=*configdir*`

旧数据库集簇配置目录；环境变量 `PGDATAOLD`

-D *configdir*
--new-datadir=*configdir*

新数据库集簇配置目录；环境变量 PGDATANEW

-j *njobs*
--jobs=*njobs*

要使用的并发进程或线程数

-k
--link

使用硬链接而不是把文件复制到新集簇

-o *options*
--old-options *options*

将直接传递给旧 postgres 命令的选项；多次指定会追加

-O *options*
--new-options *options*

将直接传递给新 postgres 命令的选项；多次指定会追加

-p *port*
--old-port=*port*

旧集簇端口号；环境变量 PGPORTOLD

-P *port*
--new-port=*port*

新集簇端口号；环境变量 PGPORTNEW

-r
--retain

即使成功完成后也保留 SQL 和日志文件

-U *username*
--username=*username*

集簇安装用户名称；环境变量 PGUSER

-v
--verbose

启用详细的内部日志记录

-V
--version

显示版本信息，然后退出

`-?`
`--help`

显示帮助，然后退出

用法

使用 pg_upgrade 执行升级的步骤如下：

1. **移动旧集簇（可选）**

   如果你使用的是带版本号的安装目录，例如 `/opt/PostgreSQL/11`，则不必移动旧集簇。图形化安装程序都使用带版本号的安装目录。

   如果你的安装目录不是带版本号的，例如 `/usr/local/pgsql`，就必须移动当前的 PostgreSQL 安装目录，以免它干扰新的 PostgreSQL 安装。当前的 PostgreSQL 服务器关闭后，就可以安全地重命名 PostgreSQL 安装目录。假设旧目录是 `/usr/local/pgsql`，你可以这样做：

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

   以上命令会重命名该目录。

2. **对于源码安装，编译新版本**

   使用与旧集簇兼容的 `configure` 标志编译新的 PostgreSQL 源码。pg_upgrade 会在开始升级前检查 `pg_controldata`，以确保所有设置都兼容。

3. **安装新的 PostgreSQL 二进制文件**

   安装新服务器的二进制文件和支持文件。默认安装中包含 pg_upgrade。

   对于源码安装，如果你希望把新服务器安装到自定义位置，可以使用 `prefix` 变量：

   ```
   make prefix=/usr/local/pgsql.new install
   ```

4. **初始化新的 PostgreSQL 集簇**

   使用 `initdb` 初始化新集簇。这里同样要使用与旧集簇匹配的兼容 `initdb` 标志。许多预构建的安装程序会自动执行该步骤。无需启动新集簇。

5. **安装扩展共享对象文件**

   许多来自 `contrib` 或其他来源的扩展和自定义模块都会使用共享对象文件（或 DLL），例如 `pgcrypto.so`。如果旧集簇使用了这些模块，则必须在新集簇中安装与新服务器二进制相匹配的共享对象文件，这通常通过操作系统命令完成。不要装载模式定义，例如 `CREATE EXTENSION pgcrypto`，因为这些会从旧集簇复制过来。如果有可用的扩展更新，pg_upgrade 会报告这一点，并创建一个稍后可运行的脚本来更新它们。

6. **复制自定义全文检索文件**

   将所有自定义全文检索文件（词典、同义词、词库、停用词）从旧集簇复制到新集簇。

7. **调整认证**

   pg_upgrade 会多次连接旧服务器和新服务器，因此你可能希望将认证设置为 peer（在 pg_hba.conf 中），或者使用 ~/.pgpass 文件（见第 34.15 节）。

8. **停止两个服务器**

   确保两个数据库服务器都已停止，在 Unix 上例如：

   ```
   pg_ctl -D /opt/PostgreSQL/12 stop
   pg_ctl -D /opt/PostgreSQL/11 stop
   ```

   或者在 Windows 上使用合适的服务名：

   ```
   NET STOP postgresql-12
   NET STOP postgresql-11
   ```

   在此次关闭期间，流复制和日志传送备库必须保持运行，以便接收所有更改。

9. **为备库升级做准备**

   如果你要使用步骤 11 节概述的方法升级备库，请通过对旧主库集簇和备库集簇运行 pg_controldata 来确认旧备库已追上主库。确认所有集簇中的 “Latest checkpoint location” 值一致。另外，请确保 wal_level 没有被设置为 minimal，这一点可在新主库集簇的 postgresql.conf 文件中检查。

10. **运行 pg_upgrade**

    始终运行新服务器的 pg_upgrade 二进制，而不是旧服务器的。pg_upgrade 需要指定新旧集簇的数据目录和可执行文件（bin）目录。你还可以指定用户和端口值，以及是否希望将数据文件链接，而不是采用默认的复制行为。

    如果使用链接模式，升级将快得多（无需复制文件）且占用更少磁盘空间，但一旦在升级后启动新集簇，就无法再访问旧集簇。链接模式还要求新旧集簇的数据目录位于同一文件系统中。（表空间和 pg_wal 可以位于不同文件系统中。）完整选项列表请参阅 pg_upgrade --help。

    --jobs 选项允许使用多个 CPU 核心复制/链接文件，并行转储和恢复数据库模式；一个不错的起始值是 CPU 核心数与表空间数量中的较大值。对于运行在多处理器机器上的多数据库服务器，此选项可以显著减少升级时间。

Windows 用户必须先登录管理员帐号，然后以 postgres 用户身份启动 shell，并设置正确的路径：

```
RUNAS /USER:postgres "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\PostgreSQL\11\bin;
```

然后运行 pg_upgrade，并用引号括起目录，例如：

```
pg_upgrade.exe
        --old-datadir "C:/Program Files/PostgreSQL/9.6/data"
        --new-datadir "C:/Program Files/PostgreSQL/11/data"
        --old-bindir "C:/Program Files/PostgreSQL/9.6/bin"
        --new-bindir "C:/Program Files/PostgreSQL/11/bin"
```

启动后，pg_upgrade 会验证两个集簇是否兼容，然后执行升级。你可以使用 pg_upgrade --check 仅执行检查，即使旧服务器仍在运行。pg_upgrade --check 还会概述升级后需要手工进行的调整。如果打算使用链接模式，应将 --link 选项与 --check 一起使用，以启用链接模式的专用检查。pg_upgrade 需要对当前目录具有写权限。

显然，在升级期间不应有人访问这些集簇。pg_upgrade 默认会在 50432 端口上运行服务器，以避免意外的客户端连接。升级时可以让两个集簇使用相同的端口号，因为新旧集簇不会同时运行。不过，在检查仍在运行的旧服务器时，新旧端口号必须不同。

如果在恢复数据库模式时发生错误，pg_upgrade 将退出，你必须按照下文步骤 17 所述回退到旧集簇。若要再次尝试 pg_upgrade，你需要修改旧集簇，使 pg_upgrade 的模式恢复步骤能够成功完成。如果问题出在某个 contrib 模块，而该模块并未用于存储用户数据，则你可能需要先从旧集簇卸载这个 contrib 模块，并在升级后将其安装到新集簇。

11. **升级流复制和日志传送备库**

    如果你使用了链接模式，并且有流复制（见第 26.2.5 节）或日志传送（见第 26.2 节）备库，可以按照下列步骤快速升级它们。你无需在备库上运行 pg_upgrade，而是要在主库上运行 rsync。现在先不要启动任何服务器。

    如果你没有使用链接模式、没有或不想使用 rsync，或者想要更简单的方案，请跳过本节说明；待 pg_upgrade 完成且新主库启动后，直接重建备库即可。

    a. **在备库上安装新的 PostgreSQL 二进制文件**

       确保所有备库都已安装新的二进制文件和支持文件。

    b. **确保新的备库数据目录不存在**

       确保新的备库数据目录不存在，或者为空。如果运行过 initdb，请删除备库的新数据目录。

    c. **安装扩展共享对象文件**

       在新备库上安装与新主库集簇中相同的扩展共享对象文件。

d. **停止备库**

如果备库仍在运行，请按上述说明立即停止它们。

e. **保存配置文件**

保存旧备库配置目录中需要保留的配置文件，例如 `postgresql.conf`（以及其包含的所有文件）、`postgresql.auto.conf`、`recovery.conf`、`pg_hba.conf`，因为在下一步中它们会被覆盖或删除。

f. **运行 rsync**

在使用链接模式时，可以通过 rsync 快速升级备库。为此，在主库上选择一个位于新旧数据库集簇目录之上的目录，并在主库上针对每个备库运行：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive old_cluster new_
cluster remote_dir
```

其中，`old_cluster` 和 `new_cluster` 是相对于主库当前目录的路径，`remote_dir` 则是备库上位于新旧集簇目录上层的目录。主库和备库在指定目录下的目录结构必须一致。请参阅 rsync 手册页，了解如何指定远程目录，例如：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive /opt/PostgreSQL/
9.5 \
      /opt/PostgreSQL/9.6 standby.example.com:/opt/PostgreSQL
```

你可以使用 rsync 的 `--dry-run` 选项来验证命令将执行什么操作。尽管至少要在主库上为一个备库运行 rsync，但也可以运行 rsync 来升级其他备库，运行位置是某个已升级且尚未启动的备库。

其原理是记录主库上新旧集簇文件之间由 pg_upgrade 链接模式创建的硬链接，然后在备库的旧集簇中找到匹配文件，并在其新集簇中为这些文件创建链接。主库上未被链接的文件会从主库复制到备库（通常都很小）。这使得备库能够快速升级。不幸的是，rsync 会无谓地复制与临时表和不记录 WAL 的表相关的文件，因为这些文件通常不存在于备库上。

如果你有表空间，则需要对每个表空间目录运行类似的 rsync 命令，例如：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive /vol1/pg_tblsp/
PG_9.5_201510051 \
      /vol1/pg_tblsp/PG_9.6_201608131 standby.example.com:/vol1/pg_tblsp
```

如果你把 `pg_wal` 迁移到了数据目录之外，也必须对这些目录运行 rsync。

g. **配置流复制和日志传送备库**

为服务器配置日志传送。（由于备库仍与主库保持同步，因此无需运行 `pg_start_backup()` 和 `pg_stop_backup()`，也无需进行文件系统备份。）复制槽不会被复制，必须重建。

12. **恢复 `pg_hba.conf`**

如果你修改了 `pg_hba.conf`，请恢复其原始设置。也可能需要调整新集簇中的其他配置文件以匹配旧集簇，例如 `postgresql.conf`（以及其包含的所有文件）和 `postgresql.auto.conf`。

13. **启动新服务器**

    现在可以安全地启动新服务器，然后启动所有经 rsync 同步的备库。

14. **升级后处理**

    如果需要任何升级后处理，pg_upgrade 在完成时会发出警告。它还会生成必须由管理员运行的脚本文件。这些脚本会连接到每个需要升级后处理的数据库。每个脚本应通过以下方式运行：

    ```
    psql --username=postgres --file=script.sql postgres
    ```

    这些脚本可以按任意顺序运行，运行完毕后即可删除。

    > **小心**
    >
    > 一般来说，在重建脚本运行完成之前访问其中引用的表是不安全的；这样做可能产生错误结果或较差的性能。未在重建脚本中引用的表则可以立即访问。

15. **统计信息**

    由于优化器统计信息不会由 `pg_upgrade` 传输，升级结束时你会被要求运行命令来重新生成这些信息。你可能需要设置连接参数以匹配新集簇。

16. **删除旧集簇**

    一旦你确认升级结果令人满意，就可以运行 `pg_upgrade` 完成时提到的脚本，删除旧集簇的数据目录。（如果旧数据目录中包含用户定义的表空间，则无法自动删除。）你也可以删除旧的安装目录（例如 `bin`、`share`）。

17. **恢复到旧集簇**

    如果在运行 `pg_upgrade` 之后，你想恢复到旧集簇，可以选择以下几种方法：

    - 如果使用了 `--check` 选项，旧集簇不会被修改；可以重新启动。
    - 如果 `--link` 选项没有被使用，旧集簇就未被修改，可以重新启动。
    - 如果使用了 `--link` 选项，新旧集簇可能共享数据文件：
      - 如果 `pg_upgrade` 在开始建立链接之前就已中止，旧集簇不会被修改；可以重新启动。
      - 如果你没有启动新集簇，则旧集簇没有被修改，只是在开始建立链接时，会把一个 `.old` 后缀附加到 `$PGDATA/global/pg_control`。要重新使用旧集簇，请移除 `.old` 这个附加在 `$PGDATA/global/pg_control` 上的后缀；随后即可重新启动旧集簇。
      - 如果你启动了新集簇，它就已经写入了共享文件，此时再使用旧集簇是不安全的。在这种情况下，旧集簇必须从备份恢复。

注解

pg_upgrade 不支持升级包含使用下列 `reg*` OID 引用系统数据类型的表列的数据库：`regproc`、`regprocedure`、`regoper`、`regoperator`、`regconfig` 和 `regdictionary`。（`regtype` 可以升级。）

凡是会影响你安装环境的失败、重建和重新索引情况，pg_upgrade 都会报告；用于重建表和索引的升级后脚本也会自动生成。如果你试图自动化升级很多集簇，你会发现具有相同数据库模式的集簇在所有升级中需要相同的升级后步骤；这是因为升级后步骤依据的是数据库模式，而不是用户数据。

为了测试部署，请创建旧集簇的纯模式副本，插入虚拟数据，然后对其升级。

如果正在升级 PostgreSQL 9.2 之前的集簇，而且它使用了只包含配置文件的目录，就必须把实际数据目录的位置传给pg_upgrade，并把配置目录的位置传给服务器，例如`-d /real-data-directory -o '-D /configuration-directory'`。

如果旧服务器早于 9.1，而且使用非默认的 Unix 域套接字目录，或者其默认目录与新集簇的默认目录不同，请将`PGHOST` 设置为旧服务器的套接字位置。（这与 Windows 无关。）

如果你想使用链接模式，又不希望在启动新集簇时修改旧集簇，可以先复制旧集簇，再对该副本使用链接模式升级。要创建旧集簇的有效副本，可在服务器运行时使用 `rsync` 创建旧集簇的脏拷贝，然后关闭旧服务器，再运行 `rsync --checksum`，把所有变更同步到该副本，使其达到一致状态。（之所以需要 `--checksum`，是因为 `rsync` 对文件修改时间的粒度只有一秒。）你可能还想排除某些文件，例如 `postmaster.pid`，如第 25.3.3 节所述。如果你的文件系统支持文件系统快照或写时复制文件副本，也可以用这些方式备份旧集簇和表空间，不过快照和副本必须同时创建，或者在数据库服务器关闭时创建。

另见

initdb, pg_ctl, pg_dump, postgres

pg_verify_checksums

pg_verify_checksums — 验证 PostgreSQL 数据库集簇中的数据校验和

大纲

`pg_verify_checksums` [*option*...] [[`-D` | `--pgdata`] *datadir*]

描述

`pg_verify_checksums` 验证 PostgreSQL 集簇中的数据校验和。在运行 pg_verify_checksums 之前，必须干净地关闭服务器。如果没有校验和错误，退出状态为零，否则为非零。

选项

可以使用以下命令行选项：

`-D` *directory*
`--pgdata=`*directory*

指定存储数据库集簇的目录。

`-v`
`--verbose`

启用详细输出，列出所有被检查的文件。

`-r` *relfilenode*

仅验证指定 relfilenode 的关系中的校验和。

`-V`
`--version`

打印 pg_verify_checksums 的版本并退出。

`-?`
`--help`

显示 pg_verify_checksums 命令行参数的帮助并退出。

环境

`PGDATA`

指定存储数据库集簇的目录；可以用 `-D` 选项覆盖。

pg_waldump

pg_waldump — 以人类可读形式显示 PostgreSQL 数据库集簇的预写式日志

大纲

`pg_waldump` [`option`...] [`startseg` [`endseg`]]

描述

`pg_waldump` 显示预写式日志（WAL），主要用于调试或教学。

该工具只能由安装服务器的用户运行，因为它要求对数据目录具有只读访问权限。

选项

以下命令行选项控制输出的位置和格式：

`startseg`

从指定的 WAL 段文件开始读取。这会隐式决定搜索文件时所使用的路径以及要使用的时间线。

`endseg`

读取完指定的 WAL 段文件后停止。

`-b`
`--bkp-details`

输出有关备份块的详细信息。

`-e` *`end`*
`--end=`*`end`*

在指定的 WAL 位置停止读取，而不是一直读取到日志流末尾。

`-f`
`--follow`

在到达有效 WAL 的末尾后，继续每秒轮询一次，等待新的 WAL 出现。

`-n` *`limit`*
`--limit=`*`limit`*

显示指定数量的记录，然后停止。

`-p` *path*
`--path=`*path*

指定一个用于搜索 WAL 段文件的目录，或者指定一个包含 `pg_wal` 子目录且其中包含此类文件的目录。默认会在当前目录、当前目录的`pg_wal` 子目录以及 `PGDATA` 的 `pg_wal` 子目录中搜索。

`-r` *rmgr*
`--rmgr=`*rmgr*

仅显示由指定资源管理器生成的记录。如果传入的名称为 `list`，则打印有效资源管理器名称的列表并退出。

`-s` *start*
`--start=`*start*

从指定的 WAL 位置开始读取。默认会从找到的最早文件中的第一条有效 WAL 记录开始读取。

`-t` *timeline*
`--timeline=`*timeline*

读取日志记录时所使用的时间线。如果指定了 *startseg*，默认使用其中的值；否则默认值为 1。

`-V`
`--version`

打印 pg_waldump 的版本并退出。

`-x` *xid*
`--xid=`*xid*

仅显示标记有给定事务 ID 的记录。

`-z`
`--stats[=record]`

显示汇总统计信息（记录和整页镜像的数量及大小），而不是单条记录。也可以选择按记录类型而不是按资源管理器生成统计信息。

`-?`
`--help`

显示 pg_waldump 命令行参数的帮助信息，并退出。

注解

服务器运行时，可能给出错误的结果。

只显示指定的时间线（如果未指定，则使用默认时间线）。其他时间线中的记录会被忽略。

pg_waldump 不能读取带有`.partial` 后缀的 WAL 文件。如果需要读取这些文件，则必须从文件名中移除`.partial` 后缀。

另见

第 30.5 节

postgres

postgres — PostgreSQL 数据库服务器

大纲

`postgres` [*option*...]

描述

`postgres` 是 PostgreSQL 数据库服务器。客户端应用要访问数据库时，会（通过网络或本地）连接到一个正在运行的`postgres` 实例。该 `postgres` 实例随后会启动一个独立的服务器进程来处理该连接。

一个 `postgres` 实例始终只管理一个数据库集簇的数据。数据库集簇是一组存储在同一文件系统位置（“数据区域”）中的数据库。只要使用不同的数据区域和不同的通信端口（见下文），一个系统上就可以同时运行多个 `postgres` 实例。当 `postgres` 启动时，它需要知道数据区域的位置。该位置必须通过 `-D` 选项或 `PGDATA` 环境变量指定；没有默认值。通常，`-D` 或 `PGDATA` 会直接指向由 initdb 创建的数据区域目录。其他可能的文件布局见第 19.2 节。

默认情况下，`postgres` 在前台启动，并将日志消息打印到标准错误流。在实际应用中，`postgres` 应作为后台进程启动，例如在系统启动时启动。

`postgres` 命令也可以在单用户模式下调用。这种模式的主要用途是在 initdb 执行引导期间使用。有时也会用它来调试或进行灾难恢复；但请注意，运行单用户服务器其实并不适合调试服务器，因为不会发生真实的进程间通信和锁定。从 shell 以单用户模式调用时，用户可以输入查询，结果会打印到屏幕上，但其形式对开发者更有用，而非面向终端用户。在单用户模式下，会话用户会被设置为 ID 为 1 的用户，并向该用户隐式授予超级用户权限。该用户实际上不必存在，因此单用户模式可用于手工恢复系统目录遭受某些意外损坏的情况。

选项

`postgres` 接受以下命令行参数。有关这些选项的详细讨论，请参阅第 19 章。通过设置配置文件，你可以省去键入其中大多数选项的麻烦。某些（安全的）选项还可以由连接的客户端以应用相关的方式设置，并且仅对该会话生效。例如，如果设置了环境变量`PGOPTIONS`，那么基于 libpq 的客户端就会将该字符串传给服务器，服务器会将其解释为 `postgres` 命令行选项。

通用选项

`-B` *`nbuffers`*

设置供服务器进程使用的共享缓冲区数量。该参数的默认值由 initdb 自动选择。指定此选项等效于设置 shared_buffers 配置参数。

`-c` *`name=value`*

设置一个命名的运行时参数。PostgreSQL 支持的配置参数见第 19 章。大多数其他命令行选项实际上都是这种参数赋值的简写形式。`-c` 可以出现多次以设置多个参数。

`-C` *name*

打印指定运行时参数的值并退出。（详情见上面的 `-c` 选项。）这个选项可用于正在运行的服务器，它会返回 `postgresql.conf` 中的值，并应用本次调用中提供的任何参数修改。它不反映集簇启动时提供的参数。

该选项供其他与服务器实例交互的程序（例如 pg_ctl）查询配置参数值之用。面向用户的应用则应使用 SHOW 或 `pg_settings` 视图。

`-d` *debug-level*

设置调试级别。该值设得越高，写入服务器日志的调试输出就越多。取值范围是 1 到 5。还可以针对某个特定会话传入`-d 0`，从而阻止父`postgres` 进程的服务器日志级别传播到该会话。

`-D` *datadir*

指定数据库配置文件在文件系统中的位置。详见第 19.2 节。

`-e`

将默认日期风格设置为“European”，即输入日期字段采用 `DMY` 顺序。这也会在某些日期输出格式中使日期显示为日在月前。更多信息见第 8.5 节。

`-F`

禁用 `fsync` 调用以提高性能，但在系统崩溃时会有数据损坏的风险。指定此选项等效于禁用 fsync 配置参数。使用前请先阅读详细文档！

`-h` *hostname*

指定 `postgres` 用来侦听来自客户端应用的 TCP/IP 连接的 IP 主机名或地址。该值也可以是逗号分隔的地址列表，或者用`*` 指定侦听所有可用接口。空值表示不侦听任何 IP 地址，在这种情况下只能使用 Unix 域套接字连接到服务器。默认只侦听 localhost。指定此选项等效于设置 listen_addresses 配置参数。

`-i`

允许远程客户端通过 TCP/IP（Internet 域）连接。没有此选项时，只接受本地连接。该选项等效于在 `postgresql.conf` 中，或通过`-h` 将 `listen_addresses` 设置为`*`。

该选项已被弃用，因为它无法访问listen_addresses 的全部功能。通常最好直接设置 `listen_addresses`。

`-k` *directory*

指定 `postgres` 用来侦听客户端应用连接的 Unix 域套接字所在目录。该值也可以是逗号分隔的目录列表。空值表示不侦听任何 Unix 域套接字，在这种情况下只能使用 TCP/IP 套接字连接到服务器。默认值通常是 `/tmp`，但可以在编译时更改。指定此选项等效于设置 unix_socket_directories 配置参数。

`-l`

启用使用 SSL 的安全连接。若要使用此选项，PostgreSQL 必须在编译时启用 SSL 支持。关于使用 SSL 的更多信息，请参阅第 18.9 节。

`-N` *`max-connections`*

设置该服务器将接受的最大客户端连接数。该参数的默认值由 initdb 自动选择。指定此选项等效于设置 max_connections 配置参数。

`-o` *`extra-options`*

在 *`extra-options`* 中指定的命令行形式参数，会传给由这个`postgres` 进程启动的所有服务器进程。

`extra-options` 中的空格被视为参数分隔符，除非用反斜线（`\`）转义；写成`\\` 可表示字面的反斜线。也可以多次使用 `-o` 来指定多个参数。

此选项已过时；服务器进程的所有命令行选项都可以直接在 `postgres` 命令行上指定。

`-p` *`port`*

指定 `postgres` 用来侦听客户端应用连接的 TCP/IP 端口或本地 Unix 域套接字文件扩展名。默认值取自 `PGPORT` 环境变量；如果 `PGPORT` 未设置，则默认值就是在编译期间确定的值（通常为 5432）。如果你指定了非默认端口，那么所有客户端应用都必须通过命令行选项或 `PGPORT` 指定相同的端口。

`-s`

在每条命令结束时打印时间信息和其他统计信息。这对基准测试或者调整缓冲区数量很有用。

`-S` *`work-mem`*

指定内部排序和 hash 运算在转而使用临时磁盘文件之前可使用的内存量。有关 `work_mem` 配置参数的说明，请参阅第 19.4.1 节。

`-V`
`--version`

打印 postgres 的版本并退出。

`--`*`name=value`*

设置一个命名的运行时参数；这是 `-c` 的较短形式。

`--describe-config`

该选项会以制表符分隔的 `COPY` 格式导出服务器内部配置变量、说明以及默认值。它主要是为管理工具设计的。

`-?`
`--help`

显示 postgres 命令行参数的帮助并退出。

半内部选项

这里描述的选项主要用于调试，并且在某些情况下可协助恢复严重损坏的数据库。在生产数据库环境中没有理由使用它们。这里列出这些选项，仅供PostgreSQL 系统开发者使用。此外，这些选项在未来版本中可能会更改或被移除，且不另行通知。

`-f { s | i | o | b | t | n | m | h }`

禁止使用某些扫描和连接方法：`s` 和 `i` 分别禁用顺序扫描和索引扫描，`o`、`b` 和 `t` 分别禁用仅索引扫描、位图索引扫描和 TID 扫描，而`n`、`m` 和 `h` 则分别禁用嵌套循环连接、归并连接和哈希连接。

顺序扫描和嵌套循环连接都无法被完全禁用；`-fs` 和 `-fn` 选项只是在优化器有其他选择时，尽量不使用这些计划类型。

`-n`

此选项用于调试导致服务器进程异常终止的问题。在这种情况下，通常的策略是通知所有其他服务器进程必须终止，然后重新初始化共享内存和信号量。这是因为出错的服务器进程可能在终止前破坏了某些共享状态。此选项指定`postgres` 不重新初始化共享数据结构。经验丰富的系统程序员随后可以使用调试器检查共享内存和信号量的状态。

`-O`

允许修改系统表的结构。该选项由 `initdb` 使用。

`-P`

读取系统表时忽略系统索引，但在修改表时仍然更新这些索引。这在从损坏的系统索引中恢复时很有用。

`-t pa[rser] | pl[anner] | e[xecutor]`

打印与各个主要系统模块相关的每个查询的时间统计信息。该选项不能与 `-s` 选项一起使用。

`-T`

此选项用于调试导致服务器进程异常终止的问题。在这种情况下，通常的策略是通知所有其他服务器进程必须终止，然后重新初始化共享内存和信号量。这是因为出错的服务器进程可能在终止前破坏了某些共享状态。此选项指定`postgres` 通过发送 `SIGSTOP` 信号暂停所有其他服务器进程，但不使它们终止。这允许系统程序员手工收集所有服务器进程的核心转储。

`-v` *protocol*

指定特定会话要使用的前端/后端协议版本号。该选项仅供内部使用。

`-W` *seconds*

新服务器进程在完成认证过程后，会延迟这么多秒。这是为了给调试器附着到该服务器进程提供机会。

用于单用户模式的选项

以下选项仅适用于单用户模式（参见单用户模式）。

`--single`

选择单用户模式。这必须是命令行上的第一个参数。

database

指定要访问的数据库名称。这必须是命令行上的最后一个参数。如果省略，则默认为用户名。

-E

在执行前将所有命令回显到标准输出。

-j

使用“分号后跟两个换行符”而不是单个换行符，作为命令输入终止符。

-r *filename*

将所有服务器日志输出发送到 *filename*。只有将此选项作为命令行选项提供时，它才会生效。

环境

PGCLIENTENCODING

客户端使用的默认字符编码。（客户端可以单独覆盖它。）该值也可以在配置文件中设置。

PGDATA

默认数据目录位置

PGDATESTYLE

DateStyle 运行时参数的默认值。（该环境变量的用法已被弃用。）

PGPORT

默认端口号（最好在配置文件中设置）

诊断

提到 semget 或 shmget 的失败消息，很可能表明你需要配置内核，以提供足够的共享内存和信号量。更多讨论见第 18.4 节。你也许可以通过降低 shared_buffers 来减少 PostgreSQL 的共享内存消耗，以及/或通过降低max_connections 来减少信号量消耗，从而推迟重新配置内核。

表明另一个服务器已经在运行的失败消息应仔细检查，例如可根据你的系统使用下面的命令：

```
$ ps ax | grep postgres
```

或

```
$ ps -ef | grep postgres
```

如果你确信没有冲突的服务器在运行，可以删除消息中提到的锁文件，然后重试。

表明无法绑定端口的失败消息，可能意味着该端口已经被某个非PostgreSQL 进程占用。你在终止 postgres 后如果立即使用同一端口重新启动它，也可能会得到这个错误；在这种情况下，你

只需等待几秒钟，直到操作系统关闭该端口后再重试。最后，如果你指定了一个操作系统认为是保留的端口号，也可能会收到这个错误。例如，许多版本的 Unix 认为低于 1024 的端口号是“受信任的”，并且只允许 Unix 超级用户访问它们。

注解

实用命令 pg_ctl 可用于安全、便捷地启动和关闭postgres 服务器。

只要有可能，就不要使用 SIGKILL 杀死主 postgres 服务器。这样会阻止 postgres 在终止前释放其持有的系统资源（例如共享内存和信号量）。这可能会导致新的 postgres 实例在启动时出现问题。

要正常终止 postgres 服务器，可以使用 SIGTERM、SIGINT 或 SIGQUIT 信号。第一个会在退出前等待所有客户端终止，第二个会强制断开所有客户端，而第三个会在不进行正确关闭的情况下立即退出，从而在重启期间导致一次恢复运行。

SIGHUP 信号会重新加载服务器配置文件。也可以向单个服务器进程发送 SIGHUP，但这通常没有意义。

要取消一个正在运行的查询，可以向执行该命令的进程发送SIGINT 信号。要干净地终止一个后端进程，可以向该进程发送SIGTERM。这两种操作在 SQL 中可调用的等效形式，见第 9.26.2 节中的 pg_cancel_backend 和 pg_terminate_backend。

postgres 服务器使用 SIGQUIT 通知从属服务器进程在不进行正常清理的情况下终止。用户不应该使用这个信号。向某个服务器进程发送 SIGKILL 也是不明智的 – 主 postgres 进程会将其解释为一次崩溃，并作为其标准崩溃恢复过程的一部分强制所有同级进程退出。

缺陷

--选项在 FreeBSD 或 OpenBSD 上不起作用。请改用 -c。这是受影响操作系统中的一个缺陷；如果这个问题未被修复，未来版本的PostgreSQL 将提供变通方案。

单用户模式

要启动单用户模式服务器，可以使用类似下面的命令：

postgres --single -D /usr/local/pgsql/data *other-options* my_database

使用 -D 提供正确的数据库目录路径，或者确保已经设置了环境变量PGDATA。还要指定你想要操作的那个数据库的名称。

通常，单用户模式服务器将换行视为命令输入终止符；它不像psql 那样会对分号进行智能处理。要让一个命令跨越多行，必须在除最后一个换行之外的每个换行前输入反斜线。反斜线和紧邻的换行都会从输入命令中删除。注意，即使在字符串字面量或注释中也会如此。

但是，如果你使用-j 命令行开关，单个换行并不会终止命令输入；取而代之的是使用“分号-换行-换行”这一序列。也就是说，输入一个分号，后面立刻跟着一个完全空白的行。在这种模式下，反斜线-换行不会被特殊处理。同样，对于这种序列出现在字符串字面量或注释中的情况，也不会有特殊处理。

在任一种输入模式中，如果你输入的分号既不恰好位于命令输入终止符之前，也不是命令输入终止符的一部分，那么它就会被视为命令分隔符。当你输入命令输入终止符时，已输入的多条语句会作为单个事务执行。

要退出会话，输入EOF（通常是 **Control+D**）。如果自上一个命令输入终止符以来你已经输入了任何文本，那么EOF 会被视为命令输入终止符，你还需要再输入一次EOF 才能退出。

请注意，单用户模式服务器不提供复杂的行编辑功能（例如没有命令历史）。单用户模式也不会执行任何后台处理，例如自动检查点或复制。

示例

要使用默认值在后台启动 `postgres`，输入：

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

要用指定端口启动 `postgres`，例如 1234：

```
$ postgres -p 1234
```

要使用 psql 连接到该服务器，请用`-p` 选项指定这个端口：

```
$ psql -p 1234
```

或者设置环境变量 `PGPORT`：

```
$ export PGPORT=1234
$ psql
```

命名的运行时参数可以采用以下任一种形式设置：

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

两种形式都会覆盖 `postgresql.conf` 中可能存在的 `work_mem` 设置。请注意，参数名中的下划线在命令行中既可以写成下划线，也可以写成连字符。除了短期实验之外，与其依赖命令行开关来设置参数，通常更好的做法是直接编辑 `postgresql.conf` 中的设置。

另见

initdb, pg_ctl

postmaster

postmaster — PostgreSQL 数据库服务器

大纲

`postmaster` [*option*...]

描述

`postmaster` 是 `postgres` 的一个已弃用别名。

另见

postgres

部分 VII. 内部

本部分包含一些可能对 PostgreSQL 开发人员有用的各类信息。

目录

51. PostgreSQL 内部概述 2036
51.1. 查询的路径 2036
51.2. 连接是如何建立的 2036
51.3. 解析器阶段 2037
51.3.1. 解析器 2037
51.3.2. 转换过程 2037
51.4. PostgreSQL 规则系统 2038
51.5. 规划器/优化器 2038
51.5.1. 生成可能的计划 2038
51.6. 执行器 2039
52. 系统目录 2041
52.1. 概述 2041
52.2. pg_aggregate 2043
52.3. pg_am 2044
52.4. pg_amop 2045
52.5. pg_amproc 2046
52.6. pg_attrdef 2046
52.7. pg_attribute 2047
52.8. pg_authid 2049
52.9. pg_auth_members 2050
52.10. pg_cast 2050
52.11. pg_class 2051
52.12. pg_collation 2054
52.13. pg_constraint 2055
52.14. pg_conversion 2057
52.15. pg_database 2058
52.16. pg_db_role_setting 2059
52.17. pg_default_acl 2059
52.18. pg_depend 2060
52.19. pg_description 2061
52.20. pg_enum 2062
52.21. pg_event_trigger 2062
52.22. pg_extension 2063
52.23. pg_foreign_data_wrapper 2063
52.24. pg_foreign_server 2064
52.25. pg_foreign_table 2064
52.26. pg_index 2065
52.27. pg_inherits 2067
52.28. pg_init_privs 2067
52.29. pg_language 2068
52.30. pg_largeobject 2069
52.31. pg_largeobject_metadata 2069
52.32. pg_namespace 2069
52.33. pg_opclass 2070
52.34. pg_operator 2070
52.35. pg_opfamily 2071
52.36. pg_partitioned_table 2071
52.37. pg_pltemplate 2072
52.38. pg_policy 2073
52.39. pg_proc 2074

52.40. pg_publication 2077
52.41. pg_publication_rel 2077
52.42. pg_range 2077
52.43. pg_replication_origin 2078
52.44. pg_rewrite 2078
52.45. pg_seclabel 2079
52.46. pg_sequence 2079
52.47. pg_shdepend 2080
52.48. pg_shdescription 2081
52.49. pg_shseclabel 2081
52.50. pg_statistic 2082
52.51. pg_statistic_ext 2083
52.52. pg_subscription 2084
52.53. pg_subscription_rel 2084
52.54. pg_tablespace 2085
52.55. pg_transform 2085
52.56. pg_trigger 2086
52.57. pg_ts_config 2087
52.58. pg_ts_config_map 2088
52.59. pg_ts_dict 2088
52.60. pg_ts_parser 2089
52.61. pg_ts_template 2089
52.62. pg_type 2090
52.63. pg_user_mapping 2095
52.64. 系统视图 2095
52.65. pg_available_extensions 2096
52.66. pg_available_extension_versions 2097
52.67. pg_config 2097
52.68. pg_cursors 2097
52.69. pg_file_settings 2098
52.70. pg_group 2099
52.71. pg_hba_file_rules 2099
52.72. pg_indexes 2100
52.73. pg_locks 2100
52.74. pg_matviews 2103
52.75. pg_policies 2103
52.76. pg_prepared_statements 2104
52.77. pg_prepared_xacts 2104
52.78. pg_publication_tables 2105
52.79. pg_replication_origin_status 2105
52.80. pg_replication_slots 2105
52.81. pg_roles 2106
52.82. pg_rules 2107
52.83. pg_seclabels 2107
52.84. pg_sequences 2108
52.85. pg_settings 2109
52.86. pg_shadow 2110
52.87. pg_stats 2111
52.88. pg_tables 2113
52.89. pg_timezone_abbrevs 2113
52.90. pg_timezone_names 2114
52.91. pg_user 2114

52.92. pg_user_mappings 2114
52.93. pg_views 2115
53. 前端/后端协议 2116
53.1. 概述 2116
53.1.1. 消息概述 2116
53.1.2. 扩展查询概述 2117
53.1.3. 格式和格式代码 2117
53.2. 消息流 2117
53.2.1. 启动 2117
53.2.2. 简单查询 2120
53.2.3. 扩展查询 2122
53.2.4. 管道化 2125
53.2.5. 函数调用 2125
53.2.6. COPY 操作 2126
53.2.7. 异步操作 2127
53.2.8. 取消正在处理的请求 2128
53.2.9. 终止 2128
53.2.10. SSL 会话加密 2128
53.3. SASL 认证 2129
53.3.1. SCRAM-SHA-256 认证 2129
53.4. 流复制协议 2130
53.5. 逻辑流复制协议 2137
53.5.1. 逻辑流复制参数 2137
53.5.2. 逻辑复制协议消息 2138
53.5.3. 逻辑复制协议消息流 2138
53.6. 消息数据类型 2138
53.7. 消息格式 2139
53.8. 错误和通知消息字段 2157
53.9. 逻辑复制消息格式 2158
53.10. 自协议 2.0 以来的变更总结 2163
54. PostgreSQL 编码约定 2165
54.1. 格式 2165
54.2. 在服务器内部报告错误 2165
54.3. 错误消息风格指南 2168
54.4. 其他编码约定 2172
55. 本地语言支持 2175
55.1. 给翻译者 2175
55.1.1. 要求 2175
55.1.2. 概念 2175
55.1.3. 创建和维护消息目录 2176
55.1.4. 编辑 PO 文件 2177
55.2. 给程序员 2177
55.2.1. 实现机制 2177
55.2.2. 消息编写指南 2179
56. 编写过程语言调用处理器 2181
57. 编写外部数据包装器 2184
57.1. 外部数据包装器函数 2184
57.2. 外部数据包装器回调例程 2184
57.2.1. 扫描外部表的 FDW 例程 2185
57.2.2. 扫描外部连接的 FDW 例程 2187
57.2.3. 规划扫描或连接之后处理的 FDW 例程 2188
57.2.4. 更新外部表的 FDW 例程 2188
57.2.5. 行锁的 FDW 例程 2194

57.2.6. EXPLAIN 的 FDW 例程 2195
57.2.7. ANALYZE 的 FDW 例程 2196
57.2.8. IMPORT FOREIGN SCHEMA 的 FDW 例程 2196
57.2.9. 并行执行的 FDW 例程 2197
57.2.10. 路径重新参数化的 FDW 例程 2198
57.3. 外部数据包装器辅助函数 2198
57.4. 外部数据包装器查询规划 2199
57.5. 外部数据包装器中的行锁定 2201
58. 编写一种表采样方法 2203
58.1. 采样方法支持函数 2203
59. 编写自定义扫描提供者 2206
59.1. 创建自定义扫描路径 2206
59.1.1. 自定义扫描路径回调 2207
59.2. 创建自定义扫描计划 2208
59.2.1. 自定义扫描计划回调 2208
59.3. 执行自定义扫描 2209
59.3.1. 自定义扫描执行回调 2209
60. 遗传查询优化器 2211
60.1. 将查询处理看成是一个复杂的优化问题 2211
60.2. 遗传算法 2211
60.3. PostgreSQL 中的遗传查询优化（GEQO） 2212
60.3.1. 使用 GEQO 生成可能的计划 2213
60.3.2. PostgreSQL GEQO 的未来实现任务 2213
60.4. 进一步阅读 2213
61. 索引访问方法接口定义 2214
61.1. 索引的基本 API 结构 2214
61.2. 索引访问方法函数 2217
61.3. 索引扫描 2222
61.4. 索引锁定注意事项 2223
61.5. 索引唯一性检查 2224
61.6. 索引代价估算函数 2225
62. 通用 WAL 记录 2229
63. B-树索引 2231
63.1. 简介 2231
63.2. B-树操作符类的行为 2231
63.3. B-树支持函数 2232
63.4. 实现 2233
64. GiST 索引 2234
64.1. 简介 2234
64.2. 内置操作符类 2234
64.3. 可扩展性 2235
64.4. 实现 2245
64.4.1. GiST 缓冲构建 2245
64.5. 示例 2245
65. SP-GiST 索引 2246
65.1. 简介 2246
65.2. 内置操作符类 2246
65.3. 可扩展性 2246
65.4. 实现 2255
65.4.1. SP-GiST 限制 2255
65.4.2. 无结点标签的 SP-GiST 2255
65.4.3. “全部相同”的内部元组 2255
65.5. 示例 2256

66. GIN 索引 2257
66.1. 简介 2257
66.2. 内置操作符类 2257
66.3. 可扩展性 2257
66.4. 实现 2260
66.4.1. GIN 快速更新技术 2260
66.4.2. 部分匹配算法 2261
66.5. GIN 提示和技巧 2261
66.6. 限制 2262
66.7. 示例 2262
67. BRIN 索引 2263
67.1. 简介 2263
67.1.1. 索引维护 2263
67.2. 内置操作符类 2263
67.3. 可扩展性 2264
68. Hash 索引 2268
68.1. 概述 2268
68.2. 实现 2268
69. 数据库物理存储 2270
69.1. 数据库文件布局 2270
69.2. TOAST 2272
69.2.1. 行外、磁盘上的 TOAST 存储 2273
69.2.2. 行外、内存中的 TOAST 存储 2274
69.3. 空闲空间映射 2274
69.4. 可见性映射 2274
69.5. 初始化分支 2275
69.6. 数据库页面布局 2275
69.6.1. 表行布局 2276
69.7. 堆内元组（HOT） 2277
70. 系统目录声明与初始内容 2279
70.1. 系统目录声明规则 2279
70.2. 系统目录初始数据 2280
70.2.1. 数据文件格式 2280
70.2.2. OID 分配 2281
70.2.3. OID 引用查找 2282
70.2.4. 编辑数据文件的方法 2282
70.3. BKI 文件格式 2283
70.4. BKI 命令 2284
70.5. 引导 BKI 文件的结构 2285
70.6. BKI 示例 2286
71. 规划器如何使用统计信息 2287
71.1. 行估计示例 2287
71.2. 多元统计信息示例 2293
71.2.1. 函数依赖 2293
71.2.2. 多元非重复值计数 2294
71.3. 规划器统计信息与安全 2294

第 51 章 PostgreSQL 内部概述

作者

本章最初源自[sim98] Stefan Simkovics 在维也纳技术大学完成的硕士论文，该论文由 O. Univ.Prof.Dr. Georg Gottlob 和 Univ.Ass. Mag. Katrin Seyr 指导。

本章概述 PostgreSQL 后端的内部结构。阅读以下各节之后，你应该能够大致了解一个查询是如何被处理的。本章并不试图详细描述 PostgreSQL 的内部运作，因为那样的文档会非常庞大。相反，本章旨在帮助读者理解，从后端收到一个查询开始，到将结果返回给客户端为止，内部通常有哪些操作步骤。

51.1. 查询的路径

这里将简要概述一个查询为得到结果必须经过的各个阶段。

1. 必须先建立从应用程序到 PostgreSQL 服务器的连接。应用程序将查询发送给服务器，并等待接收服务器返回的结果。

2. 解析器阶段检查应用程序传来的查询是否具有正确的语法，并创建一棵查询树。

3. 重写系统接收由解析器阶段创建的查询树，并查找任何可应用于该查询树的规则（存储在系统目录中）。它执行这些规则体中给出的转换。

   重写系统的一个应用是实现视图。每当对一个视图（即虚拟表）发出查询时，重写系统都会将用户的查询重写成一个改为访问视图定义中给出的基表的查询。

4. 规划器/优化器接收（重写后的）查询树，并创建一个查询计划，将作为执行器的输入。

   它首先生成所有能得到同一结果的可能路径。例如，如果待扫描的某个关系上有一个索引，那么该扫描就有两条路径：一种是简单的顺序扫描，另一种是使用该索引。接着会估算执行每条路径的代价，并选择代价最低的路径。代价最低的路径会被展开成一个完整的计划，供执行器使用。

5. 执行器会递归地遍历计划树，并以计划所表示的方式提取行。执行器在扫描关系时会使用存储系统，执行排序和连接，计算限定条件，最后返回得到的行。

在后续各节中，我们将更详细地介绍上述各项内容，以便更好地理解 PostgreSQL 的内部控制和数据结构。

51.2. 连接是如何建立的

PostgreSQL 使用一种简单的“每用户一个进程”客户端/服务器模型。在这种模型中，一个客户端进程恰好连接到一个服务器进程。由于事先不知道会建立多少个连接，因此必须使用一个主进程，在每次收到连接请求时派生一个新的服务器进程。这个主进程称为 `postgres`，在指定的 TCP/IP 端口上监听传入连接。每当检测到连接请求时，`postgres` 进程就会派生一个新的服务器进程。服务器任务之间使用信号量和共享内存通信，以确保并发数据访问期间的数据完整性。

客户端进程可以是任何理解 PostgreSQL 协议（见第 53 章）的程序。很多客户端基于 C 语言库 libpq，但该协议也有若干独立实现，例如 Java 的 JDBC 驱动。

一旦连接建立起来，客户端进程就可以向后端（服务器）发送查询。查询以纯文本方式传输，也就是说，前端（客户端）不进行解析。服务器解析查询，创建一个执行计划，执行该计划，并通过已建立的连接把检索到的行返回给客户端。

51.3. 解析器阶段

解析器阶段由两部分组成：

- 解析器定义在 `gram.y` 和 `scan.l` 中，它使用 Unix 工具 bison 和 flex 构建。
- 转换过程，它会对解析器返回的数据结构做修改和补充。

51.3.1. 解析器

解析器必须检查查询字符串（它以纯文本形式到达）是否具有有效语法。如果语法正确，就会构造出一棵语法解析树并返回；否则会返回错误。解析器和词法分析器是使用著名的 Unix 工具 bison 和 flex 实现的。

词法分析器定义在文件 `scan.l` 中，负责识别标识符、SQL 关键字等。每找到一个关键字或标识符，就会生成一个词元并交给解析器。

解析器定义在文件 `gram.y` 中，由一组语法规则和动作组成，每当某条规则被触发时，对应动作就会执行。动作中的代码（实际上是 C 代码）用于构造语法解析树。

文件 `scan.l` 会被转换成 C 源文件 `scan.c`，所用程序是 flex；而 `gram.y` 则会被转换成 `gram.c`，所用程序是 bison。这些转换完成之后，就可以使用普通的 C 编译器来构建解析器。绝不要修改这些生成出来的 C 文件，因为下一次调用 flex 或 bison 时，它们都会被覆盖。

> **注意**
>
> 上述转换和编译通常都是借助 makefiles 自动完成的，这些文件随 PostgreSQL 源代码发行版一同提供。

对 bison 的详细描述，或者对 `gram.y` 中给出的语法规则的说明，都超出了本文的范围。有许多书籍和文档专门讨论 flex 和 bison。在开始研究之前，你应该先熟悉 bison，然后再去看 `gram.y` 中给出的语法，否则你将无法理解其中发生了什么。

51.3.2. 转换过程

解析器阶段仅依据 SQL 语法结构的固定规则创建一棵语法解析树。它不会在系统目录中做任何查找，因此不可能理解所请求操作的详细语义。解析器完成之后，转换过程会接收解析器返回的树，并进行必要的语义解释，以理解查询引用了哪些表、函数和操作符。为表示这些信息而构建的数据结构称为查询树。

之所以将原始解析与语义分析分开，是因为系统目录查找只能在事务内部进行，而我们不希望在刚收到查询字符串时就立刻启动事务。原始解析阶段已经足以识别事务控制命令（`BEGIN`、`ROLLBACK` 等），因此这些命令可以在无需进一步分析的情况下被正确执行。一旦我们知道当前处理的是实际查询（例如 `SELECT` 或 `UPDATE`），如果尚未处于事务中，就可以启动事务。只有在那之后才能调用转换过程。

转换过程创建的查询树在大多数地方的结构都与原始语法解析树相似，但在细节上有许多不同。例如，语法解析树中的一个 `FuncCall` 节点表示某个在语法上看起来像函数调用的东西。它可能被转换成 `FuncExpr` 节点，也可能被转换成 `Aggref` 节点，这取决于被引用的名称最终被判定为

普通函数还是聚合函数。此外，关于列和表达式结果的实际数据类型的信息也会被加入到查询树中。

51.4. PostgreSQL 规则系统

PostgreSQL 提供了一个强大的规则系统，用于定义视图以及处理有歧义的视图更新。最初，PostgreSQL 的规则系统由两种实现组成：

- 第一种实现使用行级处理，并且深度实现于执行器内部。每当访问到单独一行时，规则系统就会被调用。这种实现在 1995 年被移除，当时 Berkeley Postgres 项目的最后一个官方发行版被转换成了 Postgres 95。
- 规则系统的第二种实现是一种称为查询重写的技术。重写系统是位于解析器阶段与规划器/优化器之间的一个模块。这种技术至今仍在使用。

关于查询重写器，第 41 章中已有相当详细的讨论，因此这里没有必要再展开。我们只指出一点：重写器的输入和输出都是查询树，也就是说，树的表示形式以及语义细节层次都不会发生变化。重写可以被看作某种形式的宏展开。

51.5. 规划器/优化器

规划器/优化器的任务是创建一个最优的执行计划。给定的一个 SQL 查询（因此也就是一棵查询树）实际上可以用多种不同方式执行，而这些方式都会产生同样的结果集。如果在计算上可行，查询优化器将考察这些可能执行计划中的每一种，并最终选择预计运行最快的那个执行计划。

> **注意**
>
> 在某些情况下，考察查询的每一种可能执行方式会耗费过多时间和内存。尤其是在执行涉及大量连接操作的查询时更是如此。为了在合理时间内确定一个合理的（不一定最优的）查询计划，PostgreSQL 会使用遗传查询优化器（见第 60 章），前提是连接数量超过某个阈值（见 geqo_threshold）。

规划器的搜索过程实际上使用一种称为路径的数据结构，它只是计划的简化表示，仅包含规划器做出决策所需的信息。在确定出代价最低的路径之后，会构建一棵完整的计划树传递给执行器。这棵树以足够细致的方式表示了期望的执行计划，使执行器能够运行它。在本节余下部分，我们将忽略路径与计划之间的区别。

51.5.1. 生成可能的计划

规划器/优化器首先为查询中使用的每个单独关系（表）生成扫描计划。可能的计划由每个关系上可用的索引决定。对一个关系总是可以执行顺序扫描，因此一定会生成顺序扫描计划。假设某个关系上定义了一个索引（例如一个 B-树索引），并且查询中包含限制条件 `relation.attribute OPR constant`。如果 `relation.attribute` 恰好匹配该 B-树索引的键，而且 `OPR` 是该索引操作符类中列出的操作符之一，那么就会生成另一个使用该 B-树索引扫描该关系的计划。如果还存在其他索引，并且查询中的限制条件恰好匹配某个索引的键，则还会考虑更多计划。对于那些排序顺序能够匹配查询 `ORDER BY` 子句（如果存在）或者可能有助于进行归并连接（见下文）的索引，也会生成索引扫描计划。

如果查询需要连接两个或更多关系，那么会在找出所有可行的单关系扫描计划之后，再考虑关系连接计划。有以下三种可用的连接策略：

- 嵌套循环连接：左关系中找到的每一行，都会使右关系被扫描一次。这种策略实现起来很容易，但可能非常耗时。（不过，如果右关系可以通过索引扫描，那么这也可能是一种不错的策略。可以把左关系当前行中的值作为右关系索引扫描的键。）
- 归并连接：在连接开始之前，每个关系都会先按照连接属性排序。然后两个关系并行扫描，将匹配的行组合成连接结果行。这种连接方式很有吸引力，因为每个关系只需扫描一次。所需的排序既可以通过显式排序步骤完成，也可以利用连接键上的索引按适当顺序扫描关系来完成。
- 哈希连接：首先扫描右关系，并使用其连接属性作为 hash 键将其装入一个 hash 表。接着扫描左关系，并将找到的每一行中的相应值作为 hash 键，用来在该 hash 表中定位匹配的行。

当查询涉及两个以上的关系时，最终结果必须由一棵连接步骤树构建出来，每个连接步骤都有两个输入。规划器会考察不同的可能连接顺序，以找出代价最低的那个。

如果查询使用的关系少于 geqo_threshold 个，就会执行一次近乎穷举的搜索，以找出最佳连接顺序。对于任意两个关系，只要在 `WHERE` 限定条件中存在相应的连接子句（即存在类似 `where rel1.attr1=rel2.attr2` 这样的限制），规划器就会优先考虑它们之间的连接。没有连接子句的连接对只有在别无选择时才会被考虑，也就是说，某个关系与其他任何关系之间都没有可用的连接子句。对于规划器所考虑的每一对连接，都会生成所有可能的计划，并选择其中估计代价最低的那个。

当超过 `geqo_threshold` 时，被考虑的连接顺序将由启发式方法决定，如第 60 章中所述。除此之外，处理过程与前面相同。

最终完成的计划树由基表的顺序扫描或索引扫描，再加上所需的嵌套循环、归并或哈希连接节点，以及任何需要的辅助步骤（例如排序节点或聚合函数计算节点）组成。这些计划节点类型中的大多数还具有执行选择（丢弃不满足指定布尔条件的行）和投影（根据给定列值计算派生列集，也就是在需要时计算标量表达式）的额外能力。规划器的职责之一，就是把来自 `WHERE` 子句的选择条件以及所需输出表达式的计算附加到计划树中最合适的节点上。

51.6. 执行器

执行器接收由规划器/优化器创建的计划，并递归地处理它，以提取所需的行集合。这本质上是一种按需拉取的流水线机制。每当某个计划节点被调用时，它必须再交付一行，或者报告自己已经完成了行的交付。

为了给出一个具体示例，假设顶层节点是一个 `MergeJoin` 节点。在执行归并之前，必须先取到两行（每个子计划各一行）。因此执行器会递归调用自身去处理这些子计划（从挂接在 `lefttree` 上的子计划开始）。新的顶层节点，也就是左子计划的顶层节点，假设是一个 `Sort` 节点，那么又需要通过递归来取得一个输入行。`Sort` 的子节点可能是一个 `SeqScan` 节点，它表示真正去读取一个表。执行该节点会使执行器从表中取出一行，并把它返回给调用者节点。`Sort` 节点会反复调用它的子节点，以取得所有待排序的行。当输入耗尽时（子节点返回的是 NULL 而不是一行），`Sort` 代码就执行排序，并最终能够返回它的第一条输出行，也就是排序后的第一行。它会把其余行保存起来，以便在后续请求中按排序后的顺序交付这些行。

`MergeJoin` 节点同样会向它的右子计划请求第一行。然后它比较这两行，看它们是否可以连接；如果可以，就向调用者返回一条连接结果行。在下一次调用时，它会前进到其中一个表的下一行（具体取决于比较结果），并再次检查是否匹配；如果当前这对输入无法连接，则会立即执行这一步。最终，其中一个子计划会耗尽，`MergeJoin` 节点就返回 NULL，以表明无法再形成更多连接结果行。

复杂查询可能涉及许多层计划节点，但总体方法是相同的：每个节点在每次被调用时，都会计算并返回它的下一条输出行。每个节点还负责应用规划器分配给它的任何选择或投影表达式。

执行器机制用于求值全部四种基本 SQL 查询类型：SELECT、INSERT、UPDATE 和 DELETE。对于 SELECT，顶层执行器代码只需要把查询计划树返回的每一行发送给客户端。INSERT ... SELECT、UPDATE 和 DELETE 本质上都是 SELECT，位于一个名为 ModifyTable 的特殊顶层计划节点之下。

INSERT ... SELECT 会把各行向上传递给 ModifyTable 以执行插入。对于 UPDATE，规划器会安排每个计算得到的行包含所有被更新列的值，再加上原始目标行的 TID（tuple ID，即元组 ID 或行 ID）；这些数据会向上传递给 ModifyTable 节点，该节点利用这些信息创建一个新的更新后行，并将旧行标记为已删除。对于 DELETE，计划实际返回的唯一列就是 TID，而 ModifyTable 节点只需利用该 TID 访问每个目标行并将其标记为已删除。

一个简单的 INSERT ... VALUES 命令会创建一棵极为简单的计划树，它只包含一个 Result 节点。该节点仅计算一条结果行，并将其向上传递给 ModifyTable 执行插入。

第 52 章 系统目录

系统目录是关系型数据库管理系统存放模式元数据的地方，例如表和列的信息以及内部记账信息。PostgreSQL 的系统目录是普通表。你可以删除并重建这些表、增加列、插入和更新值，并借此把系统严重搞坏。通常，不应手工修改系统目录，通常都有 SQL 命令可用于完成这些操作。（例如，`CREATE DATABASE` 会向 `pg_database` 目录中插入一行 — 并且实际上会在磁盘上创建该数据库。）对于某些特别深奥的操作，确实存在少数例外，但随着时间推移，其中很多也已经可以通过 SQL 命令完成，因此直接操纵系统目录的需求正越来越少。

52.1. 概述

表 52.1 列出了系统目录。每个目录更详细的文档见后文。

大多数系统目录在创建数据库时都会从模板数据库复制过来，因此之后是数据库专有的。少数目录在物理上由一个集簇中的所有数据库共享；这些目录会在各自的说明中注明。

表 52.1. 系统目录

目录名	用途
pg_aggregate	聚合函数
pg_am	索引访问方法
pg_amop	访问方法操作符
pg_amproc	访问方法支持函数
pg_attrdef	列默认值
pg_attribute	表列（“属性”）
pg_authid	授权标识符（角色）
pg_auth_members	授权标识符成员关系
pg_cast	转换（数据类型转换）
pg_class	表、索引、序列、视图（“关系”）
pg_collation	排序规则（区域设置信息）
pg_constraint	检查约束、唯一约束、主键约束、外键约束
pg_conversion	编码转换信息
pg_database	本数据库集簇中的数据库
pg_db_role_setting	每角色和每数据库的设置
pg_default_acl	对象类型的默认权限
pg_depend	数据库对象间的依赖
pg_description	数据库对象上的描述或注释
pg_enum	枚举标签和值定义
pg_event_trigger	事件触发器
pg_extension	已安装扩展
pg_foreign_data_wrapper	外部数据包装器定义
pg_foreign_server	外部服务器定义
pg_foreign_table	外部表的补充信息
pg_index	索引的补充信息

目录名	用途
pg_inherits	表继承层次
pg_init_privs	对象初始权限
pg_language	编写函数的语言
pg_largeobject	大对象的数据页
pg_largeobject_metadata	大对象的元数据
pg_namespace	模式
pg_opclass	访问方法操作符类
pg_operator	操作符
pg_opfamily	访问方法操作符族
pg_partitioned_table	表的分区键的信息
pg_pltemplate	过程语言的模板数据
pg_policy	行安全性策略
pg_proc	函数和过程
pg_publication	用于逻辑复制的发布
pg_publication_rel	关系与发布的映射
pg_range	范围类型的信息
pg_replication_origin	已注册的复制源
pg_rewrite	查询重写规则
pg_seclabel	数据库对象上的安全标签
pg_sequence	有关序列的信息
pg_shdepend	共享对象上的依赖
pg_shdescription	共享对象上的注释
pg_shseclabel	共享数据库对象上的安全标签
pg_statistic	规划器统计信息
pg_statistic_ext	扩展规划器统计信息
pg_subscription	逻辑复制订阅
pg_subscription_rel	订阅的关系状态
pg_tablespace	本数据库集簇内的表空间
pg_transform	转换（将数据类型转换为过程语言需要的形式）
pg_trigger	触发器
pg_ts_config	文本检索配置
pg_ts_config_map	文本检索配置的词元映射
pg_ts_dict	文本检索词典
pg_ts_parser	文本检索解析器
pg_ts_template	文本检索模板
pg_type	数据类型
pg_user_mapping	用户到外部服务器的映射

52.2. pg_aggregate

目录 pg_aggregate 存储关于聚合函数的信息。聚合函数是对一组值（典型的是每个匹配查询条件的行中的同一个列的值）进行操作的函数，它返回从这些值中计算出的单个值。典型的聚合函数是 sum、count 和 max。pg_aggregate 里的每个项都是一个 pg_proc 项的扩展。pg_proc 项记载该聚合的名字、输入和输出数据类型，以及其他一些和普通函数类似的信息。

表 52.2. pg_aggregate 列

名称	类型	引用	描述
aggfnoid	regproc	pg_proc.oid	聚合函数对应的 pg_proc OID
aggkind	char		聚合种类：n 表示“普通”聚合，o 表示“有序集”聚合，或 h 表示“假想集”聚合
aggnumdirectargs	int2		有序集或假想集聚合的直接（非聚合）参数个数，其中一个可变参数数组算作一个参数。如果它等于 pronargs，则该聚合必定是可变参数的，并且该可变参数数组既描述聚合参数也描述末尾的直接参数。对普通聚合总为零。
aggtransfn	regproc	pg_proc.oid	转移函数
aggfinalfn	regproc	pg_proc.oid	最终函数（如果没有就为零）
aggcombinefn	regproc	pg_proc.oid	合并函数（如果没有就为零）
aggserialfn	regproc	pg_proc.oid	序列化函数（如果没有就为零）
aggdeserialfn	regproc	pg_proc.oid	反序列化函数（如果没有就为零）
aggmtransfn	regproc	pg_proc.oid	用于移动聚合模式的前向状态转移函数（如果没有就为零）
aggminvtransfn	regproc	pg_proc.oid	用于移动聚合模式的逆向状态转移函数（如果没有就为零）
aggmfinalfn	regproc	pg_proc.oid	用于移动聚合模式的最终函数（如果没有就为零）
aggfinalextra	bool		若为真，则会向aggfinalfn 传递额外的虚拟参数
aggmfinalextra	bool		若为真，则会向aggmfinalfn 传递额外的虚拟参数

名称	类型	引用	描述
aggfinalmodify	char		aggfinalfn 是否修改转移状态值：r 表示只读，s 表示在aggfinalfn 之后不能再应用aggtransfn，或 w 表示它会写入该值
aggmfinalmodify	char		类似于aggfinalmodify，但用于aggmfinalfn
aggsortop	oid	pg_operator.oid	相关联的排序操作符（如果没有则为 0）
aggtranstype	oid	pg_type.oid	聚合函数的内部转移（状态）数据的数据类型
aggtransspace	int4		转移状态数据的近似平均尺寸（字节），或者为零表示使用一个默认估算值
aggmtranstype	oid	pg_type.oid	聚合函数在移动聚合模式下内部转移（状态）数据的数据类型（如果没有则为零）
aggmtransspace	int4		移动聚合模式的转移状态数据的近似平均尺寸（字节），或者为零表示使用默认估算值
agginitval	text		转移状态的初始值。这是一个文本字段，包含初始值的外部字符串表示。如果此字段为 NULL，则转移状态值初始为 NULL。
aggminitval	text		移动聚合模式下转移状态的初始值。这是一个文本字段，包含初始值的外部字符串表示。如果此字段为 NULL，则转移状态值初始为 NULL。

新的聚合函数可通过 CREATE AGGREGATE 命令注册。更多关于编写聚合函数以及转移函数的含义等信息请参见第 38.11 节。

52.3. pg_am

目录 pg_am 存储关于关系访问方法的信息。系统支持的每种访问方法在这个目录中都有一行。目前只有索引拥有访问方法。索引访问方法的需求在第 61 章中详细讨论。

表 52.3. pg_am 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
amname	name		访问方法的名字

名称	类型	引用	描述
amhandler	regproc	pg_proc.oid	负责提供有关该访问方法信息的处理器函数的 OID
amtype	char		目前总是i，表示索引访问方法；将来可能允许其他值

> **注意**
>
> 在 PostgreSQL 9.6 之前，pg_am 包含很多额外的列以表示索引访问方法的性质。那些数据现在只有在 C 代码级别才是直接可见的。不过，系统中增加了pg_index_column_has_property() 和一些相关函数来允许 SQL 查询检查索引访问方法的性质，请见表 9.63。

52.4. pg_amop

目录 pg_amop 存储关于与访问方法操作符族相关的操作符信息。对于操作符族中的每个成员操作符，此目录中都有一行。一个成员可以是搜索操作符，也可以是排序操作符。一个操作符可以出现在多个族中，但在同一个族中既不能出现在多个搜索位置，也不能出现在多个排序位置（虽然不太可能，但允许一个操作符同时用于搜索和排序）。

表 52.4. pg_amop 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
amopfamily	oid	pg_opfamily.oid	此项所属的操作符族
amoplefttype	oid	pg_type.oid	操作符的左侧输入数据类型
amoprighttype	oid	pg_type.oid	操作符的右侧输入数据类型
amopstrategy	int2		操作符策略号
amoppurpose	char		操作符目的，s 表示搜索，o 表示排序
amopopr	oid	pg_operator.oid	操作符的 OID
amopmethod	oid	pg_am.oid	此操作符族所属的索引访问方法
amopsortfamily	oid	pg_opfamily.oid	如果是排序操作符，则此项按这个 B-树操作符族排序；如果是搜索操作符，则为零

一个“搜索”操作符项表示，该操作符族上的索引可以用来查找满足如下条件的所有行：WHERE *indexed_column operator constant*。显然，这样的操作符必须返回boolean，并且它的左输入类型必须匹配索引列的数据类型。

一个“排序”操作符项表示，该操作符族上的索引可以被扫描，以按如下顺序返回行：ORDER BY *indexed_column operator constant*。这样的操作符可以返回任何可排序的

数据类型，不过它的左输入类型同样必须匹配索引列的数据类型。ORDER BY 的准确语义由 amopsortfamily 列指定，它必须引用一个适用于该操作符结果类型的 B-树操作符族。

> **注意**
>
> 目前，排序操作符的排序顺序被假定为其所引用操作符族的默认值，即 ASC NULLS LAST。将来可能会通过增加额外的列来显式指定排序选项，从而放宽这一假设。

一个项的 amopmethod 必须和其所属操作符族的 opfmethod 相匹配（此处加入 amopmethod 是出于性能原因而对目录结构故意做出的反规范化）。此外，amoplefttype 和 amoprighttype 也必须匹配被引用的 pg_operator 项的 oprleft 和 oprright 字段。

52.5. pg_amproc

目录 pg_amproc 存储与访问方法操作符族相关的支持函数的信息。属于一个操作符族的每一个支持函数在这个目录中都有一行。

表 52.5. pg_amproc 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
amprocfamily	oid	pg_opfamily.oid	此项所属的操作符族
amproclefttype	oid	pg_type.oid	相关操作符的左侧输入数据类型
amprocrighttype	oid	pg_type.oid	相关操作符的右侧输入数据类型
amprocnum	int2		支持函数编号
amproc	regproc	pg_proc.oid	函数的 OID

amproclefttype 和 amprocrighttype 列的通常解释是它们标识了一个特定支持函数所支持的操作符的左右输入类型。对于某些访问方法它们和支持函数本身的输入数据类型相匹配，而对其他的则不会匹配。对于一个索引有一个“默认”支持函数的概念，这些支持函数的 amproclefttype 和 amprocrighttype 都等于索引操作符类的 opcintype。

52.6. pg_attrdef

目录 pg_attrdef 存储列默认值。列的主要信息存储在 pg_attribute 中（见下文）。只有在创建表或添加列时显式指定默认值的列，才会在这里有一项。

表 52.6. pg_attrdef 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
adrelid	oid	pg_class.oid	该列所属的表
adnum	int2	pg_attribute.attnum	列的编号

名称	类型	引用	描述
adbin	pg_node_tree		列默认值的内部表示
adsrc	text		适合人阅读的默认值表示

adsrc 字段是历史遗留，最好不要使用，因为它不会跟踪可能影响默认值表示的外部变化。显示默认值时，更好的方法是反编译adbin 字段（例如使用 pg_get_expr）。

52.7. pg_attribute

目录 pg_attribute 存储关于表列的信息。数据库中每个表的每一列，在pg_attribute 中都恰好有一行。（其中也包括索引的属性项，实际上，凡是在pg_class 中有项的对象，都有对应的属性项。）

术语属性等同于列，这里使用它只是出于历史原因。

表 52.7. pg_attribute 列

名称	类型	引用	描述
attrelid	oid	pg_class.oid	此列所属的表
attname	name		列名称
atttypid	oid	pg_type.oid	此列的数据类型
attstattarget	int4		attstattarget 控制 ANALYZE 为该列收集统计信息时的细节级别。零值表示不应收集统计信息。负值表示使用系统默认统计目标。正值的确切含义依赖于数据类型。对于标量数据类型，attstattarget 既是要收集的“高频值”目标个数，也是要创建的直方图桶目标个数。
attlen	int2		该列数据类型的pg_type.typlen 副本
attnum	int2		列的编号。一般列从 1 开始向上编号。系统列，如 oid，则拥有（任意）负值编号。
attndims	int4		如果该列是数组类型，则为维数；否则为 0。（目前并不会强制检查数组维数，因此任何非零值实际上都只意味着“这是一个数组”。）
attcacheoff	int4		在存储中总是 -1，但加载到内存中的行描述符后，此值可能被更新，用于缓存该属性在行内的偏移量

名称	类型	引用	描述
atttypmod	int4		atttypmod 记录了在表创建时提供的类型相关数据（例如一个varchar 列的最大长度）。它会被传递给类型相关的输入函数和长度强制转换函数。对于那些不需要atttypmod 的类型，这个值通常为 -1。
attbyval	bool		该列类型的pg_type.typbyval 的一个拷贝
attstorage	char		通常是该列类型的pg_type.typstorage 的一个拷贝。对于可 TOAST 的数据类型，这可以在列创建后被修改以控制存储策略。
attalign	char		该列类型的 pg_type.typalign 的一个拷贝
attnotnull	bool		这代表一个非空约束
atthasdef	bool		该列有一个默认值，在此情况下pg_attrdef 目录中会有一个对应项来真正定义该值。
atthasmissing	bool		此列有一个值，供该列在数据行中完全缺失时使用；在数据行创建之后添加带有非易失性 DEFAULT 值的列时，就会出现这种情况。实际使用的值存储在 attmissingval 列中。
attidentity	char		如果是一个零字节（''），则不是一个标识列。否则，a = 总是生成，d = 默认生成。
attisdropped	bool		该列被删除且不再有效。一个删除的列仍然物理存在于表中，但是会被解析器忽略并因此无法通过 SQL 访问。
attislocal	bool		该列是由关系本地定义的。注意一个列可以同时是本地定义和继承的。
attinhcount	int4		该列直接祖先的数量。祖先数量非零的列不能被删除，也不能被重命名。
attcollation	oid	pg_collation.oid	该列定义的排序规则；如果该列的数据类型不支持排序规则，则为零。
attacl	aclitem[]		列级访问权限，如果此列上已有特别授予的权限

名称	类型	引用	描述
attoptions	text[]		属性级选项，以 “keyword=value” 形式的字符串
attfdwoptions	text[]		属性级的外部数据包装器选项，以 “keyword=value” 形式的字符串
attmissingval	anyarray		此列包含一个单元素数组，其中的值供该列在数据行中完全缺失时使用；在数据行创建之后添加带有非易失性 DEFAULT 值的列时，就会出现这种情况。只有 atthasmissing 为真时才使用此值。如果没有这样的值，则此列为 NULL。

在被删除列的 pg_attribute 条目中，atttypid 被重置为零，但 attlen 以及其他从 pg_type 复制的字段仍然有效。这种安排用于应对被删除列的数据类型后来也被删除、因而不再有相应 pg_type 行的情况。attlen 和其他字段可用于解释表中一行的内容。

52.8. pg_authid

目录 pg_authid 包含关于数据库授权标识符（角色）的信息。角色涵盖了 “用户” 和 “组” 这两个概念。用户本质上只是设置了 rolcanlogin 标志的角色。任何角色（无论是否设置了 rolcanlogin）都可以拥有其他角色作为成员；参见 pg_auth_members。

由于这个目录包含密码，不得允许对它进行公开读取。pg_roles 是建立在 pg_authid 之上的公开可读视图，它会隐藏密码字段。

第 21 章包含关于用户和权限管理的详细信息。

由于用户身份是集簇范围的，pg_authid 在一个集簇的所有数据库之间共享：在一个集簇中只有一份 pg_authid 拷贝，而不是每个数据库一份。

表 52.8. pg_authid 列

名称	类型	描述
oid	oid	行标识符（隐藏属性，必须显式选择）
rolname	name	角色名
rolsuper	bool	角色有超级用户权限
rolinherit	bool	该角色是否会自动继承其所属角色的权限
rolcreaterole	bool	角色能创建更多角色
rolcreatedb	bool	角色能创建数据库
rolcanlogin	bool	角色是否能登录。即该角色是否能够作为初始会话授权标识符
rolreplication	bool	角色是一个复制角色。复制角色可以启动复制连接并且创建和删除复制槽。

名称	类型	描述
rolbypassrls	bool	角色是否可以绕过所有的行级安全策略，详见第 5.7 节。
rolconnlimit	int4	对于可以登录的角色，本列设置该角色能够建立的最大并发连接数。-1 表示无限制。
rolpassword	text	密码（可能已加密）；如果未设置则为 NULL。其格式取决于所使用的加密方式。
rolvaliduntil	timestamptz	密码过期时间（仅用于密码认证）；不设过期时间时为 NULL

对于一个 MD5 加密的密码，rolpassword 列将由字符串 md5 后面跟上一个 32 字符的十六进制 MD5 hash 值构成。MD5 hash 值通过在用户密码后拼接用户名再计算得到。例如，如果用户 joe 的密码是 xyzzy，则PostgreSQL 将存储 xyzzyjoe 的 md5 hash。

如果密码使用 SCRAM-SHA-256 加密，其格式为：

SCRAM-SHA-256$*<iteration count>*:*<salt>*$*<StoredKey>*:*<ServerKey>*

其中，*salt*，*StoredKey* 和 *ServerKey* 采用 Base64 编码格式。此格式与 RFC 5803 指定的格式相同。

不符合上述任一种格式的密码将被视为未加密密码。

52.9. pg_auth_members

目录 pg_auth_members 展示了角色之间的成员关系。允许任何无环的关系集合。

由于用户身份是集簇范围的，pg_auth_members 在一个集簇的所有数据库之间共享：在一个集簇中只有一份 pg_auth_members 拷贝，而不是每个数据库一份。

表 52.9. pg_auth_members 列

名称	类型	引用	描述
roleid	oid	pg_authid.oid	拥有成员的角色的 ID
member	oid	pg_authid.oid	roleid 的成员角色的 ID
grantor	oid	pg_authid.oid	授权此成员关系的角色的 ID
admin_option	bool		如果member 能够将 roleid 的成员资格授予其他角色，则为真

52.10. pg_cast

目录 pg_cast 存储数据类型转换路径，包括内置路径和用户定义路径。

需要注意的是，pg_cast 并不表示系统知道如何执行的所有类型转换，它只包括那些不能从某些通用规则推导出的转换。例如，一个域及其基础类型之间的转换并未显式地在pg_cast 中展示。另一个重要的例外是“自动的基于 I/O 的类型转换”，它们通过数据类型自己的 I/O 函数来转换成（或者转换自）text 或其他字符串类型，这些转换也没有显式地在pg_cast 中表示。

表 52.10. pg_cast 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
castsource	oid	pg_type.oid	源数据类型的 OID
casttarget	oid	pg_type.oid	目标数据类型的 OID
castfunc	oid	pg_proc.oid	执行该转换的函数的 OID。如果该转换方法不需要一个函数则存储 0。
castcontext	char		指示可在哪些上下文中调用此类型转换。e 表示只能显式调用（使用 CAST 或 :: 语法）。a 表示可在向目标列赋值时隐式调用，也可显式调用。i 表示可在表达式中隐式调用，也允许前述其他调用方式。
castmethod	char		指示转换如何被执行。f 表明使用castfunc 中指定的函数。i 表明使用输入/输出函数。b 表明该类型是二进制可强制转换的，因此不需要转换。

在 pg_cast 里列出的类型转换函数必须总是以转换的源类型作为它的第一个参数类型，并且返回转换的目标类型作为它的结果类型。一个类型转换函数最多有三个参数。如果出现了第二个参数，必须是integer 类型；它接受与目标类型关联的类型修饰符，如果没有，就是 -1。如果出现了第三个参数，那么必须是boolean 类型；如果该类型转换是一种显式转换，那么它接受true，否则接受false。

在 pg_cast 里创建一条源类型和目标类型相同的记录是合理的，只要相关联的函数接受多过一个参数。这样的记录代表“长度转换函数”，它们把该类型的值转换为对特定类型修饰符值而言合法的值。

如果一个 pg_cast 的项有着不同的源类型和目标类型，并且有一个接收多于一个参数的函数，那么它会在一个步骤中完成从一种类型到另外一种类型的转换并应用一个长度转换。如果没有这样的项，使用一个类型修饰符的转换涉及两个步骤，一个是在数据类型之间转换，另外一个是应用修饰符。

52.11. pg_class

目录 pg_class 描述了表以及其他具有列或在其他方面与表类似的对象。这包括索引（但请参见 pg_index）、序列（但请参见pg_sequence）、视图、物化视图、复合类型和 TOAST 表；请参见relkind。在下面，当我们指的是所有这些类型的对象时，我们称之为“关系（relations）”。并非所有 pg_class 的列对所有关系类型都有意义。

表 52.11. pg_class 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）

名称	类型	引用	描述
relname	name		表、索引、视图等的名字
relnamespace	oid	pg_namespace.oid	包含该关系的名字空间的 OID
reltype	oid	pg_type.oid	与此表的行类型对应的数据类型的 OID（如有）；索引没有pg_type 项，此值为零
reloftype	oid	pg_type.oid	对于类型化表，为底层复合类型的 OID；对于其他所有关系则为零
relowner	oid	pg_authid.oid	关系的拥有者
relam	oid	pg_am.oid	如果这是索引，则为使用的访问方法（B-树、hash 等）
relfilenode	oid		该关系的磁盘文件名；零表示这是一个“映射”关系，其磁盘文件名取决于底层状态
reltablespace	oid	pg_tablespace.oid	存储此关系的表空间。如果为零，则表示使用数据库的默认表空间。（如果关系没有磁盘文件，则没有意义。）
relpages	int4		该表磁盘表示的尺寸，以页面计（页面尺寸为BLCKSZ）。这只是一个由规划器使用的估计值。它被VACUUM、ANALYZE 以及一些 DDL 命令（如CREATE INDEX）所更新。
reltuples	float4		表中的存活行数。这只是一个由规划器使用的估计值。它被VACUUM、ANALYZE 以及一些 DDL 命令（如CREATE INDEX）所更新。
relallvisible	int4		在表的可见性映射中被标记为全可见的页数。这只是一个由规划器使用的估计值。它被VACUUM、ANALYZE 以及一些 DDL 命令，如CREATE INDEX 所更新。
reltoastrelid	oid	pg_class.oid	与该表相关联的 TOAST 表的 OID，如果没有则为零。TOAST 表将大属性“行外”存储在一个二级表中。
relhasindex	bool		如果这是一个表并且其上建有（或最近建有）索引则为真

名称	类型	引用	描述
relisshared	bool		如果该表在集簇中的所有数据库间共享则为真。只有某些系统目录（如pg_database）是共享的。
relpersistence	char		p = 永久表，u = 不记录 WAL 的表，t = 临时表
relkind	char		r = 普通表，i = 索引，S = 序列，t = TOAST 表，v = 视图，m = 物化视图，c = 复合类型，f = 外部表，p = 分区表，I = 分区索引
relnatts	int2		关系中用户列的数量（不计系统列）。在 pg_attribute 中必须有这么多对应项。另请参阅pg_attribute.attnum。
relchecks	int2		表上CHECK 约束的数目，参见pg_constraint 目录
relhasoids	bool		如果为此关系的每一行生成 OID，则为真
relhasrules	bool		如果表有（或曾有）规则则为真，参见pg_rewrite 目录
relhastriggers	bool		如果表有（或曾有）触发器则为真，参见 pg_trigger 目录
relhassubclass	bool		如果表有（或曾经有过）任何继承子表，则为真
relrowsecurity	bool		如果表上启用了行级安全则为真，参见 pg_policy 目录
relforcerowsecurity	bool		如果行级安全（启用时）也适用于表拥有者则为真，参见 pg_policy 目录
relispopulated	bool		如果关系已被填充则为真（对于所有关系该列都为真，但对于某些物化视图却不是）
relreplident	char		用于形成行“复制标识”的列：d = 默认（主键，如果存在），n = 无，f = 所有列，i = 设置了 indisreplident 的索引（如果该索引已被删除，则效果等同于无）
relispartition	bool		如果表或索引是一个分区，则为真

名称	类型	引用	描述
relrewrite	oid	pg_class.oid	对于在需要重写表的 DDL 操作期间写入的新关系，此字段包含原关系的 OID；其他情况下为零。该状态仅在内部可见；对于用户可见的关系，此字段应始终为零。
relfrozenxid	xid		在此之前的所有事务 ID 在表中已经被替换为一个永久的（“冻结的”）事务 ID。这用于跟踪表是否需要被清理，以便阻止事务 ID 回卷或者允许pg_xact 被收缩。如果该关系不是一个表则为 0（InvalidTransactionId）。
relminmxid	xid		在此之前的所有多事务 ID 在表中已经被替换为一个事务 ID。这被用于跟踪表是否需要被清理，以阻止多事务 ID 回卷或者允许pg_multixact 被收缩。如果关系不是一个表则为 0（InvalidMultiXactId）。
relacl	aclitem[]		访问权限，详见GRANT 和REVOKE
reloptions	text[]		访问方法相关的选项，以 “keyword=value” 字符串形式
relpartbound	pg_node_tree		如果表是一个分区（见 relispartition），则为分区边界的内部表示

pg_class 中的一些布尔标志采用延迟维护：当条件成立时，保证它们为真；但当条件不再成立时，可能不会立即将它们重置为假。例如，relhasindex 由 CREATE INDEX 设置，但它从不会被DROP INDEX 清除。作为替代，VACUUM 会在找到无索引表后清除其 relhasindex。这种安排避免了竞争条件并且提高了并发性。

52.12. pg_collation

目录 pg_collation 描述可用的排序规则，其本质上是从 SQL 名字到操作系统区域设置类别的映射。更多信息参见第 23.2 节。

表 52.12. pg_collation 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
collname	name		排序规则名称（在每个名字空间与编码的组合内唯一）

名称	类型	引用	描述
collnamespace	oid	pg_namespace.oid	包含该排序规则的名字空间的 OID
collowner	oid	pg_authid.oid	排序规则的拥有者
collprovider	char		排序规则提供程序：d = 数据库默认值，c = libc，i = icu
collencoding	int4		该排序规则可应用的编码，或以-1 表示它可用于任何编码
collcollate	name		此排序规则对象的 LC_COLLATE
collctype	name		此排序规则对象的 LC_CTYPE
collversion	text		提供程序特有的排序规则版本。它会在创建排序规则时记录下来，并在使用时接受检查，以检测可能导致数据损坏的排序规则定义更改。

注意在这个目录中的唯一键是（collname、collencoding、collnamespace），不仅仅是（collname、collnamespace）。PostgreSQL 通常会忽略所有 collencoding 既不等于当前数据库编码、也不等于 -1 的排序规则，并且禁止创建与 collencoding = -1 的项同名的新项。因此，使用限定的 SQL 名字（*schema*.*name*）来标识一个排序规则已经足够，即使按目录定义它并不唯一。之所以将该目录定义成这样，是因为 initdb 在集簇初始化时会用系统上所有可用的区域设置填充它，因此它必须能够容纳集簇中未来可能使用到的所有编码的项。

在 template0 数据库中，创建与数据库编码不匹配的排序规则可能很有用，因为它们可以匹配之后从template0 克隆出的数据库的编码。目前这必须手工完成。

52.13. pg_constraint

目录 pg_constraint 存储表上的检查约束、主键约束、唯一约束、外键约束和排他约束。（列约束不会被特殊对待。每一个列约束都等价于某种表约束。）非空约束在pg_attribute 目录中表示，而不是在这里。

用户定义的约束触发器（使用 CREATE CONSTRAINT TRIGGER 创建）也会在这个表中产生一项。

域上的检查约束也存储在这里。

表 52.13. pg_constraint 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
conname	name		约束名称（不一定唯一！）
connamespace	oid	pg_namespace.oid	包含此约束的名字空间的 OID

名称	类型	引用	描述
contype	char		c = 检查约束，f = 外键约束，p = 主键约束，u = 唯一约束，t = 约束触发器，x = 排他约束
condeferrable	bool		该约束是否能被延迟？
condeferred	bool		该约束是否默认被延迟？
convalidated	bool		该约束是否已经验证？目前只有外键和 CHECK 约束的此值可以为假
conrelid	oid	pg_class.oid	该约束所在的表，如果不是表约束则为零
contypid	oid	pg_type.oid	该约束所在的域，如果不是域约束则为 0
conindid	oid	pg_class.oid	如果该约束是唯一、主键、外键或排他约束，此列表示支持此约束的索引，否则为零
conparentid	oid	pg_constraint.oid	如果这是一个分区中的约束，则是父分区表中对应的约束；否则为零
confrelid	oid	pg_class.oid	如果此约束是一个外键约束，此列为被引用的表，否则为零
confupdtype	char		外键更新动作代码：a = 无动作，r = 限制，c = 级联，n = 置空，d = 置为默认值
confdeltype	char		外键删除动作代码：a = 无动作，r = 限制，c = 级联，n = 置空，d = 置为默认值
confmatchtype	char		外键匹配类型：f = 完全，p = 部分，s = 简单
conislocal	bool		此约束是定义在关系本地。注意一个约束可以同时是本地定义和继承。
coninhcount	int4		该约束直接继承自多少个祖先。祖先数量非零的约束不能被删除，也不能被重命名。
connoinherit	bool		为真表示此约束被定义在关系本地。它是一个不可继承约束。
conkey	int2[]	pg_attribute.attnum	如果是一个表约束（包括外键但不包括约束触发器），此列是被约束列的列表
confkey	int2[]	pg_attribute.attnum	如果是一个外键，此列是被引用列的列表

名称	类型	引用	描述
conpfeqop	oid[]	pg_operator.oid	如果是一个外键，此列是用于 PK = FK 比较的等值操作符的列表
conppeqop	oid[]	pg_operator.oid	如果是一个外键，此列是用于 PK = PK 比较的等值操作符的列表
conffeqop	oid[]	pg_operator.oid	如果是一个外键，此列是用于 FK = FK 比较的等值操作符的列表
conexclop	oid[]	pg_operator.oid	如果是排他约束，则列出每列的排他操作符
conbin	pg_node_tree		如果是一个检查约束，此列是表达式的一个内部表示
consrc	text		如果是检查约束，则为适合人阅读的表达式表示

对于排他约束，conkey 仅对作为简单列引用的约束元素有用。对于其他情况，conkey 中会出现一个 0，必须查阅关联索引来确定被约束的表达式。（因此，conkey 与该索引的 pg_index.indkey 具有相同的内容。）

注意

被引用对象发生变化时，consrc 不会更新；例如，它不会跟踪列重命名。最好使用 pg_get_constraintdef() 提取检查约束的定义，而不是依赖这个字段。

注意

pg_class.relchecks 需要和每个关系在此目录中的检查约束数量保持一致。

52.14. pg_conversion

目录 pg_conversion 描述编码转换函数。更多信息参见 CREATE CONVERSION。

表 52.14. pg_conversion 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
conname	name		转换的名字（在一个名字空间内唯一）
connamespace	oid	pg_namespace.oid	包含此转换的名字空间的 OID
conowner	oid	pg_authid.oid	转换的拥有者
conforencoding	int4		源编码 ID
contoencoding	int4		目标编码 ID

名称	类型	引用	描述
conproc	regproc	pg_proc.oid	转换函数
condefault	bool		如果这是默认转换则为真

52.15. pg_database

目录 pg_database 存储有关可用数据库的信息。数据库通过 CREATE DATABASE 命令创建。关于其中一些参数的含义的详细信息请查阅第 22 章。

和大部分系统目录不同，pg_database 是在集簇的所有数据库之间共享的：在一个集簇中只有一份 pg_database 拷贝，而不是每个数据库一份。

表 52.15. pg_database 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
datname	name		数据库名称
datdba	oid	pg_authid.oid	数据库的拥有者，通常是创建它的用户
encoding	int4		数据库的字符编码（pg_encoding_to_char() 可以将这个数字翻译为编码名称）
datcollate	name		此数据库的 LC_COLLATE
datctype	name		此数据库的 LC_CTYPE
datistemplate	bool		如果为真，则此数据库可被任何具有CREATEDB 权限的用户克隆；如果为假，则只有超级用户或者该数据库的属主能够克隆它。
datallowconn	bool		如果为假则没有人能连接到这个数据库。这可以用来保护template0 数据库不被修改。
datconnlimit	int4		设置可以连接到此数据库的最大并发连接数。-1 表示没有限制，-2 表示数据库无效。
datlastsysoid	oid		数据库中最后一个系统 OID；对 pg_dump 尤其有用

名称	类型	引用	描述
datfrozenxid	xid		在此事务 ID 之前的所有事务 ID，都已经在该数据库中被替换为永久的（“冻结的”）事务 ID。它用于跟踪数据库是否需要清理，以防止事务 ID 回卷，或者允许pg_xact 收缩。它等于该数据库中所有表的pg_class.relfrozenxid 值的最小值。
datminmxid	xid		在此多事务 ID 之前的所有多事务 ID，都已经在该数据库中被替换为事务 ID。它用于跟踪数据库是否需要清理，以防止多事务 ID 回卷，或者允许pg_multixact 收缩。它等于该数据库中所有表的pg_class.relminmxid 值的最小值。
dattablespace	oid	pg_tablespace.oid	此数据库的默认表空间。在此数据库中，所有pg_class.reltablespace 为 0 的表都将被存储在这个表空间中；尤其是非共享系统目录都会在其中。
datacl	aclitem[]		访问权限，详见GRANT 和REVOKE

52.16. pg_db_role_setting

目录 pg_db_role_setting 为每个角色与数据库组合记录运行时配置变量的默认值。

和大部分系统目录不同，pg_db_role_setting 是在集簇的所有数据库之间共享的：在一个集簇中只有一份 pg_db_role_setting 拷贝，而不是每个数据库一份。

表 52.16. pg_db_role_setting 列

名称	类型	引用	描述
setdatabase	oid	pg_database.oid	此设置适用的数据库 OID；如果不与特定数据库相关，则为零
setrole	oid	pg_authid.oid	此设置适用的角色 OID；如果不与特定角色相关，则为零
setconfig	text[]		运行时配置变量的默认值

52.17. pg_default_acl

目录 pg_default_acl 存储要被分配给新创建对象的初始权限。

表 52.17. pg_default_acl 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
defaclrole	oid	pg_authid.oid	与此项相关的角色的 OID
defaclnamespace	oid	pg_namespace.oid	与此项相关的名字空间的 OID，如果没有则为零
defaclobjtype	char		本条目的对象类型：r = 关系（表、视图），S = 序列，f = 函数，T = 类型，n = 模式
defaclacl	aclitem[]		此类对象在创建时应具有的访问权限

一个 pg_default_acl 项表示要赋给属于指定用户的对象的初始权限。目前有两类项：defaclnamespace = 0 的“全局”项，以及引用特定模式的“按模式”项。如果存在全局项，则它会覆盖该对象类型通常写死的默认权限。如果存在按模式项，则表示要将这些权限附加到全局默认权限或写死的默认权限之上。

注意，当另一个目录中的 ACL 项为 NULL 时，它表示该对象采用写死的默认权限，而不是当前 pg_default_acl 中可能存在的内容。pg_default_acl 只会在对象创建期间被查阅。

52.18. pg_depend

目录 pg_depend 记录数据库对象之间的依赖关系。这些信息允许 DROP 命令查找必须被 DROP CASCADE 删除的其他对象，或者在DROP RESTRICT 情况下阻止删除。

另请参阅 pg_shdepend，它对涉及数据库集簇共享对象的依赖提供了相似的功能。

表 52.18. pg_depend 列

名称	类型	引用	描述
classid	oid	pg_class.oid	依赖对象所在系统目录的 OID
objid	oid	任意 OID 列	特定依赖对象的 OID
objsubid	int4		对于一个表列，这里是列号（objid 和classid 指表本身）。对于所有其他对象类型，此列为 0。
refclassid	oid	pg_class.oid	被引用对象所在的系统目录的 OID
refobjid	oid	任意 OID 列	指定被引用对象的 OID
refobjsubid	int4		对于一个表列，这里是列号（refobjid 和 refclassid 指表本身）。对于所有其他对象类型，此列为 0。
deptype	char		定义此依赖关系语义的一个代码，见下文

在所有情况下，pg_depend 中的条目都表示：如果不同时删除依赖对象，就不能删除被引用对象。不过，还存在若干由 deptype 标识的子类型：

DEPENDENCY_NORMAL (n)

两个独立创建的对象之间的正常关系。依赖对象可以被删除而不影响被引用对象。被引用对象只能通过指定 CASCADE 来删除，这样依赖对象也会被删除。例如：表列对其数据类型有一个正常依赖关系。

DEPENDENCY_AUTO (a)

依赖对象可以与被引用对象分开删除，并且在被引用对象删除时应自动删除（无论使用 RESTRICT 还是 CASCADE 模式）。例如，表上的命名约束会自动依赖于该表，因此删除表时该约束也会消失。

DEPENDENCY_INTERNAL (i)

依赖对象作为被引用对象创建过程的一部分而创建，实际上只是其内部实现的一部分。对依赖对象执行DROP 会被直接禁止（我们会告诉用户改为对被引用对象执行 DROP）。对被引用对象执行 DROP 会传播到依赖对象，使其被删除，无论是否指定CASCADE。例如，为实施外键约束而创建的触发器会在内部依赖于该约束的pg_constraint 项。

DEPENDENCY_INTERNAL_AUTO (I)

依赖对象作为被引用对象创建过程的一部分而创建，实际上只是其内部实现的一部分。对依赖对象执行DROP 会被直接禁止（我们会告诉用户改为对被引用对象执行 DROP）。对于普通内部依赖，只要还有任何这样的依赖关系存在，就不能删除依赖对象；而DEPENDENCY_INTERNAL_AUTO 允许删除，只要沿这些依赖关系中的任意一个能够找到该对象。例如，分区上的索引既以内部自动方式依赖于分区本身，也依赖于父分区表上的索引；因此，无论删除该分区索引所索引的分区，还是删除它所附属的父索引，都会一并删除该分区索引。

DEPENDENCY_EXTENSION (e)

依赖对象是作为被引用对象的扩展的一个成员（参见 pg_extension）。只能通过对被引用对象执行 DROP EXTENSION 来删除依赖对象。在功能上，这种依赖类型与内部依赖相同，但为了清晰和简化pg_dump 而单独区分。

DEPENDENCY_AUTO_EXTENSION (x)

依赖对象不是作为被引用对象的扩展的成员（因此 pg_dump 不应忽略它），但它无法在没有该扩展的情况下运行，所以删除扩展时也应删除它。依赖对象也可以单独删除。

DEPENDENCY_PIN (p)

不存在依赖对象；这种项表示系统本身依赖于被引用对象，因此该对象绝不能删除。这种项只由initdb 创建。用于表示依赖对象的各列包含零。

未来可能需要其他依赖关系的变种。

52.19. pg_description

目录 pg_description 存储对每一个数据库对象可选的描述（注释）。描述可以通过 COMMENT 操作，并可使用psql 的\d 命令查看。在 pg_description 的初始内容中提供了很多内置系统对象的描述。

参见 pg_shdescription，它对在一个数据库集簇中共享的对象的描述提供了相似的功能。

表 52.19. pg_description 列

名称	类型	引用	描述
objoid	oid	任意 OID 列	描述所属对象的 OID
classoid	oid	pg_class.oid	该对象所在的系统目录的 OID
objsubid	int4		对于一个表列上的一个注释，这里是列号（objoid 和classoid 指表本身）。对所有其他对象类型，此列为 0。
description	text		作为该对象描述的任意文本

52.20. pg_enum

pg_enum 目录包含每一个枚举类型的项，其中包括了值和标签。一个给定枚举值的内部表示实际上是它在pg_enum 中的相关行的 OID。

表 52.20. pg_enum 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
enumtypid	oid	pg_type.oid	包含此枚举值的pg_type 项的 OID
enumsortorder	float4		此枚举值在其枚举类型中的排序位置
enumlabel	name		此枚举值的文本标签

pg_enum 行的 OID 值遵循一种特殊的规则：即偶数 OID 值被保证按照其枚举类型的排序顺序进行排序。即如果两个偶数 OID 属于同一枚举类型，较小的 OID 必然具有较小的enumsortorder 值。奇数 OID 值不需要遵循排序顺序。这种规则使得枚举比较例程在很多常见情况下可以避免系统目录查找。创建和修改枚举类型的例程将尝试尽可能地为枚举值分配偶数 OID。

当一个枚举类型被创建后，其成员会被分配排序位置 1..*n*。但后来增加的成员可能会被赋予负值或分数值的 enumsortorder。对这些值的唯一要求是它们必须在每个枚举类型内排序正确且唯一。

52.21. pg_event_trigger

目录 pg_event_trigger 存储事件触发器。更多信息参见第 40 章。

表 52.21. pg_event_trigger 列

名称	类型	引用	描述
evtname	name		触发器名（必须唯一）
evtevent	name		标识会触发此触发器的事件

名称	类型	引用	描述
evtowner	oid	pg_authid.oid	事件触发器的拥有者
evtfoid	oid	pg_proc.oid	将被调用的函数
evtenabled	char		控制事件触发器触发的 session_replication_role 模式。O = 触发器在 “origin” 和 “local” 模式触发，D = 触发器被禁用，R = 触发器在 “replica” 模式触发，A = 触发器总是触发。
evttags	text[]		会触发此触发器的命令标签。如果为 NULL，则此触发器的触发不受命令标签限制。

52.22. pg_extension

目录 pg_extension 存储有关已安装扩展的信息。有关扩展的细节请参见第 38.16 节。

表 52.22. pg_extension 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
extname	name		扩展的名字
extowner	oid	pg_authid.oid	扩展的拥有者
extnamespace	oid	pg_namespace.oid	包含此扩展的导出对象的模式
extrelocatable	bool		如果扩展可被重定位到另一个模式则为真
extversion	text		扩展的版本名字
extconfig	oid[]	pg_class.oid	扩展配置表的 regclass OID 数组；如果没有配置表，则为 NULL
extcondition	text[]		扩展的配置表的WHERE 子句过滤条件的数组，如果没有则为NULL

注意和大部分具有一个 “namespace” 列的系统目录不同，extnamespace 不是用来表示扩展属于该模式。扩展的名字从不用模式进行限定。extnamespace 表明该模式包含了该扩展的大部分或全部对象。如果 extrelocatable 为真，则该模式事实上必须包含属于此扩展的全部可用模式限定的对象。

52.23. pg_foreign_data_wrapper

目录 pg_foreign_data_wrapper 存储外部数据包装器定义。外部数据包装器是一种访问位于外部服务器上数据的机制。

表 52.23. pg_foreign_data_wrapper 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
fdwname	name		外部数据包装器的名字
fdwowner	oid	pg_authid.oid	外部数据包装器的拥有者
fdwhandler	oid	pg_proc.oid	指向一个负责为外部数据包装器提供执行例程的处理函数；如果没有提供处理函数则为零
fdwvalidator	oid	pg_proc.oid	引用一个验证函数，负责检查传给此外部数据包装器的选项，以及使用该包装器的外部服务器和用户映射的选项是否有效。如果未提供验证函数，则为零
fdwacl	aclitem[]		访问权限，详见GRANT和REVOKE
fdwoptions	text[]		外部数据包装器特定选项，以 “keyword=value” 字符串形式

52.24. pg_foreign_server

目录 pg_foreign_server 存储外部服务器定义。外部服务器定义了外部数据的来源，例如一个远程服务器。外部服务器通过外部数据包装器来访问。

表 52.24. pg_foreign_server 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
srvname	name		外部服务器的名字
srvowner	oid	pg_authid.oid	外部服务器的拥有者
srvfdw	oid	pg_foreign_data_wrapper.oid	此外部服务器的外部数据包装器的 OID
srvtype	text		服务器的类型（可选）
srvversion	text		服务器的版本（可选）
srvacl	aclitem[]		访问权限，详见GRANT和REVOKE
srvoptions	text[]		外部服务器特定选项，以 “keyword=value” 字符串形式

52.25. pg_foreign_table

目录 pg_foreign_table 包含关于外部表的辅助信息。一个外部表和普通表一样，主要由一个 pg_class 项表示。它的 pg_foreign_table 项包含外部表所特有的信息。

表 52.25. pg_foreign_table 列

名称	类型	引用	描述
ftrelid	oid	pg_class.oid	外部表的pg_class 项的 OID
ftserver	oid	pg_foreign_server.oid	外部表所在的外部服务器的 OID
ftoptions	text[]		外部表选项，以 “keyword=value” 字符串形式

52.26. pg_index

目录 pg_index 包含关于索引的部分信息。其他信息大部分在 pg_class 中。

表 52.26. pg_index 列

名称	类型	引用	描述
indexrelid	oid	pg_class.oid	此索引的pg_class 项的 OID
indrelid	oid	pg_class.oid	此索引所对应的基表的 pg_class 项的 OID
indnatts	int2		索引中的总列数（与pg_class.relnatts 重复）；此数目包括键属性和内含属性
indnkeyatts	int2		索引中键列的数量，不计内含列；内含列仅被存储，不参与索引语义
indisunique	bool		如为真，这是唯一索引
indisprimary	bool		如为真，表示索引为表的主键（如果此列为真，indisunique 也总是为真）
indisexclusion	bool		如为真，此索引支持一个排他约束
indimmediate	bool		如为真，唯一性检查在插入时立即被执行（如果 indisunique 为假，此列无关）
indisclustered	bool		如果为真，表示表最后以此索引进行了聚簇
indisvalid	bool		如果为真，此索引当前可用于查询。为假表示索引可能不完整：INSERT/UPDATE 操作仍必须修改该索引，但查询不能安全地使用它。如果它是唯一索引，其唯一性也无法保证。

名称	类型	引用	描述
indcheckxmin	bool		如果为真，查询必须等到此 pg_index 行的 xmin 低于这些查询的 TransactionXmin 事件视界后，才能使用该索引，因为表中可能有断裂的HOT链，其中包含这些查询可见的不兼容行
indisready	bool		如果为真，表示此索引当前可以用于插入。为假表示索引必须被INSERT/UPDATE 操作忽略。
indislive	bool		如果为假，索引正处于被删除过程中，并且必须被所有处理忽略（包括 HOT 安全的决策）
indisreplident	bool		如果为真，此索引已通过 ALTER TABLE ... REPLICA IDENTITY USING INDEX ... 被选作“复制标识”
indkey	int2vector	pg_attribute.attnum	这是一个包含 indnatts 个值的数组，指明此索引为哪些表列建立索引。例如，值 1 3 表示表的第一列和第三列组成索引条目。键列位于非键（内含）列之前。数组中的零表示相应的索引属性是基于表列的表达式，而非简单的列引用。
indcollation	oidvector	pg_collation.oid	对于索引键（indnkeyatts 个值）中的每一列，这包含要用于该索引的排序规则的 OID，如果该列的数据类型不支持排序规则则为零。
indclass	oidvector	pg_opclass.oid	对于索引键中的每一列（indnkeyatts 个值），这里包含了要使用的操作符类的 OID。详见pg_opclass。
indoption	int2vector		这是一个包含 indnkeyatts 个值的数组，存储各列的标志位。各位的含义由索引的访问方法定义。

名称	类型	引用	描述
indexprs	pg_node_tree		非简单列引用的索引属性所对应的表达式树（以nodeToString()表示）。这是一个列表，indkey中的每个零值条目在此都有一个对应元素。如果所有索引属性都是简单列引用，则为NULL。
indpred	pg_node_tree		部分索引谓词的表达式树（以nodeToString()形式）。如果不是部分索引，此列为NULL。

52.27. pg_inherits

目录pg_inherits记录有关表和索引的继承层次的信息。数据库中每个表或索引的直接父子关系在这里都有一项（非直接继承可以通过顺着项构成的链来决定）。

表 52.27. pg_inherits 列

名称	类型	引用	描述
inhrelid	oid	pg_class.oid	子表或索引的 OID
inhparent	oid	pg_class.oid	父表或索引的 OID
inhseqno	int4		如果子表有多个直接父表（多重继承），此编号指示应当如何排列继承列的顺序。编号从 1 开始。索引不能多重继承，因为索引只有在使用声明式分区时才能继承。

52.28. pg_init_privs

目录pg_init_privs记录系统中对象的初始权限。数据库中每个具有非默认（非 NULL）初始权限集合的对象都在其中有一个条目。

对象可以在系统初始化（initdb）时获得其初始权限，也可以在CREATE EXTENSION期间创建该对象，并在扩展脚本中用GRANT设置对象的初始权限。注意，系统会自动处理扩展脚本执行期间对权限的记录；扩展作者只需要在脚本中使用GRANT和REVOKE语句，权限就会被记录下来。privtype列表示初始权限是由 initdb 设置，还是在某次CREATE EXTENSION命令期间设置。

对于初始权限由 initdb 设置的对象，其条目中的privtype是'i'；对于初始权限由CREATE EXTENSION设置的对象，其条目中的privtype为'e'。

表 52.28. pg_init_privs 列

名称	类型	引用	描述
objoid	oid	任意 OID 列	指定对象的 OID
classoid	oid	pg_class.oid	对象所在的系统目录的 OID

名称	类型	引用	描述
objsubid	int4		对于一个表列，这里是列编号（objoid 和 classoid 指向表本身）。对于所有其他对象类型，这列为零。
privtype	char		定义此对象初始权限类型的代码，见文字说明
initprivs	aclitem[]		初始访问权限，详见 GRANT 和REVOKE

52.29. pg_language

目录 pg_language 注册了可用于编写函数或存储过程的语言。更多关于语言处理器的信息请参阅 CREATE LANGUAGE 和第 42 章。

表 52.29. pg_language 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
lanname	name		语言的名字
lanowner	oid	pg_authid.oid	语言的拥有者
lanispl	bool		内部语言为假（如 SQL），用户定义语言为真。当前，pg_dump仍然使用这个列来决定要转储哪些语言，但在未来这可能会被一种不同的机制所取代。
lanpltrusted	bool		为真表示这是一种受信任的语言，即认为它不会允许访问正常 SQL 执行环境以外的任何内容。只有超级用户可以用不受信任的语言创建函数。
lanplcallfoid	oid	pg_proc.oid	对于非内部语言，此列引用语言处理器，它是一个特殊函数，负责执行所有用这种语言编写的函数。
laninline	oid	pg_proc.oid	此列引用一个负责执行“内联”匿名代码块的函数（DO 块）。如果不支持内联块则为 0。
lanvalidator	oid	pg_proc.oid	此列引用一个负责在函数创建时对其进行语法和有效性检查的语言验证函数。如果没有提供验证器则为 0。
lanacl	aclitem[]		访问权限，详见GRANT 和REVOKE

52.30. pg_largeobject

目录 pg_largeobject 保存构成“大对象”的数据。一个大对象在被创建时会被分配一个 OID。每个大对象被分解成足够小的段或“页”，以便方便地作为行存储在pg_largeobject 中。每页中的数据量被定义为 LOBLKSIZE（目前是 BLCKSZ/4，通常为 2 kB）。

在 PostgreSQL 9.0 之前，大对象没有相关的权限结构。因此，pg_largeobject 曾是公开可读的，并且可以用来获得系统中所有大对象的 OID（和内容）。但现在已经不是这样了；可使用 pg_largeobject_metadata 来获得大对象 OID 的列表。

表 52.30. pg_largeobject 列

名称	类型	引用	描述
loid	oid	pg_largeobject_metadata.oid	包含此页的大对象的标识符
pageno	int4		此页在它所属大对象中的页号（从 0 开始计）
data	bytea		实际存储在大对象中的数据。它从不会超过 LOBLKSIZE 字节，也可能更少。

pg_largeobject 的每一行保存一个大对象的一个页的数据，从对象内部的字节偏移量（pageno * LOBLKSIZE）开始。现在的实现允许稀疏存储：页面可能不存在，并且可能比 LOBLKSIZE 字节短（即便不是最后一页）。一个大对象中缺失的区域会被读出为 0。

52.31. pg_largeobject_metadata

目录 pg_largeobject_metadata 保持着与大对象有关的元数据。真正的大对象数据被存储在 pg_largeobject 中。

表 52.31. pg_largeobject_metadata 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
lomowner	oid	pg_authid.oid	大对象的拥有者
lomacl	aclitem[]		访问权限，详见GRANT和REVOKE

52.32. pg_namespace

目录 pg_namespace 存储名字空间。名字空间是 SQL 模式之下的结构：每个名字空间拥有一个独立的关系、类型等的集合，且其中没有名字冲突。

表 52.32. pg_namespace 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
nspname	name		名字空间的名字

名称	类型	引用	描述
nspowner	oid	pg_authid.oid	名字空间的拥有者
nspacl	aclitem[]		访问权限，详见GRANT和REVOKE

52.33. pg_opclass

目录 pg_opclass 定义索引访问方法的操作符类。每一个操作符类定义了一种特定数据类型和一种特定索引访问方法的索引列的语义。一个操作符类实际上指定了一个特定的操作符族可以用于一个特定可索引列数据类型。该族中可用于索引列的操作符能够接受该列的数据类型作为它们的左输入。

操作符类详见第 38.15 节。

表 52.33. pg_opclass 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
opcmethod	oid	pg_am.oid	操作符类所属的索引访问方法
opcname	name		操作符类的名称
opcnamespace	oid	pg_namespace.oid	操作符类所属的名字空间
opcowner	oid	pg_authid.oid	操作符类的拥有者
opcfamily	oid	pg_opfamily.oid	包含此操作符类的操作符族
opcintype	oid	pg_type.oid	操作符类索引的数据类型
opcdefault	bool		如果此操作符类是 opcintype 的默认操作符类，则为真
opckeytype	oid	pg_type.oid	存储在索引中的数据的类型，如果值为 0 表示与 opcintype 相同

一个操作符类的 opcmethod 必须匹配包含它的操作符族的 opfmethod。而且，对于任何给定的opcmethod 和 opcintype 组合，只有不超过一个pg_opclass 行的 opcdefault 值为真。

52.34. pg_operator

目录 pg_operator 存储关于操作符的信息。详见 CREATE OPERATOR 和第 38.13 节。

表 52.34. pg_operator 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
oprname	name		操作符的名称
oprnamespace	oid	pg_namespace.oid	操作符所属的名字空间的 OID

名称	类型	引用	描述
oprowner	oid	pg_authid.oid	操作符的拥有者
oprkind	char		b = 中缀（“两侧”），l = 前缀（“左侧”），r = 后缀（“右侧”）
oprcanmerge	bool		该操作符支持归并连接
oprcanhash	bool		该操作符支持哈希连接
oprleft	oid	pg_type.oid	左操作数类型
oprright	oid	pg_type.oid	右操作数类型
oprresult	oid	pg_type.oid	结果类型
oprcom	oid	pg_operator.oid	该操作符的交换子（如有）
oprnegate	oid	pg_operator.oid	该操作符的求反器（如有）
oprcode	regproc	pg_proc.oid	实现该操作符的函数
oprrest	regproc	pg_proc.oid	该操作符的限制选择率估算函数
oprjoin	regproc	pg_proc.oid	该操作符的连接选择率估算函数

未使用的列包含零。例如，前缀操作符的oprleft 为零。

52.35. pg_opfamily

目录 pg_opfamily 定义了操作符族。每一个操作符族是操作符和相关支持例程的集合，这一集合用于实现一个特定索引访问方法的语义。此外，按照访问方法指定的某种方式，一个族内的操作符都是“兼容的”。操作符族概念允许在索引中使用跨数据类型操作符，并可依据访问方法的语义推理这些操作符的行为。

操作符族在第 38.15 节中有详细说明。

表 52.35. pg_opfamily 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
opfmethod	oid	pg_am.oid	操作符族适用的索引访问方法
opfname	name		操作符族的名字
opfnamespace	oid	pg_namespace.oid	操作符族所属的名字空间
opfowner	oid	pg_authid.oid	操作符族的拥有者

定义操作符族的主要信息不在它的 pg_opfamily 行，而是在相关的pg_amop、pg_amproc 和 pg_opclass 行中。

52.36. pg_partitioned_table

目录 pg_partitioned_table 存放有关表如何被分区的信息。

表 52.36. pg_partitioned_table 列

名称	类型	引用	描述
partrelid	oid	pg_class.oid	这个分区表的pg_class 项的 OID
partstrat	char		分区策略；h = hash 分区表，l = 列表分区表，r = 范围分区表
partnatts	int2		分区键中的列数
partdefid	oid	pg_class.oid	这个分区表的默认分区的 pg_class 项的 OID，如果这个分区表没有默认分区则为零。
partattrs	int2vector	pg_attribute.attnum	这是一个包含 partnatts 个值的数组，它指示哪些表列是分区键的组成部分。例如，值1 3 表示第一个和第三个表列组成了分区键。这个数组中的零表示对应的分区键列是一个表达式而不是简单的列引用。
partclass	oidvector	pg_opclass.oid	对于分区键中的每一个列，此字段包含要使用的操作符类的 OID。详见 pg_opclass。
partcollation	oidvector	pg_opclass.oid	对于分区键中的每一个列，此字段包含要用于分区的排序规则的 OID，如果该列的数据类型不支持排序规则则为零。
partexprs	pg_node_tree		非简单列引用的分区键列的表达式树（以nodeToString() 的表达方式）。这是一个列表，partattrs 中每一个零项都有一个元素。如果所有分区键列都是简单列引用，则此字段为 NULL。

52.37. pg_pltemplate

目录 pg_pltemplate 存储过程语言的“模板”信息。有了语言模板，就可以用简单的 CREATE LANGUAGE 命令在特定数据库中创建该语言，无需指定实现细节。

与大多数系统目录不同，pg_pltemplate 由一个集簇中的所有数据库共享：每个集簇只有一份 pg_pltemplate，而不是每个数据库各有一份。这样，每个数据库都可以在需要时访问这些信息。

表 52.37. pg_pltemplate 列

名称	类型	描述
tmplname	name	此模板所对应的语言名称

名称	类型	描述
tmpltrusted	boolean	如果该语言被视为受信任的语言，则为真
tmpldbacreate	boolean	如果数据库拥有者可以创建该语言，则为真
tmplhandler	text	调用处理器函数的名称
tmplinline	text	匿名块处理器函数的名称；如果没有则为 NULL
tmplvalidator	text	验证器函数的名称；如果没有则为 NULL
tmpllibrary	text	实现该语言的共享库的路径
tmplacl	aclitem[]	模板的访问权限（实际上未使用）

目前没有用于操纵过程语言模板的命令；要更改内置信息，超级用户必须使用普通的INSERT、DELETE 或 UPDATE 命令修改此表。

> **注意**
>
> pg_pltemplate 很可能会在 PostgreSQL 的某个未来版本中移除，改为将这些过程语言的信息保存在各自的扩展安装脚本中。

52.38. pg_policy

目录 pg_policy 存储着表的行级安全策略。一个策略包括它适用的命令种类（可能适用于所有命令）、它适用的角色、被作为安全屏障条件增加到包括该表的查询的表达式以及被作为 WITH CHECK 选项增加到尝试向表增加新记录的查询的表达式。

表 52.38. pg_policy 列

名称	类型	引用	描述
polname	name		策略的名称
polrelid	oid	pg_class.oid	策略适用的表
polcmd	char		策略适用的命令类型：r 表示 SELECT，a 表示 INSERT，w 表示 UPDATE，d 表示 DELETE，* 表示所有命令类型
polpermissive	boolean		策略是宽松的还是限制性的？
polroles	oid[]	pg_authid.oid	策略适用的角色
polqual	pg_node_tree		被作为安全屏障条件增加到使用该表的查询的表达式树
polwithcheck	pg_node_tree		被作为 WITH CHECK 条件增加到尝试向表增加行的查询的表达式树

> **注意**
>
> 存储在 pg_policy 中的策略只有在它们所适用的表的 pg_class.relrowsecurity 被设置时才起作用。

52.39. pg_proc

目录 pg_proc 存放有关函数、过程、聚合函数以及窗口函数（统称为例程）的信息。更多信息请参考 CREATE FUNCTION、CREATE PROCEDURE 和第 38.3 节。

如果 prokind 显示该条目用于一个聚合函数，在pg_aggregate 中应该有一个相匹配的行。

表 52.39. pg_proc 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
proname	name		函数的名字
pronamespace	oid	pg_namespace.oid	函数所属的名字空间的 OID
proowner	oid	pg_authid.oid	函数的拥有者
prolang	oid	pg_language.oid	此函数的实现语言或调用接口
procost	float4		估计的执行代价（以 cpu_operator_cost 为单位），如果proretset 为真，这是每行返回的代价
prorows	float4		估计的结果行数量（如果 proretset 为假，该值为 0）
provariadic	oid	pg_type.oid	可变参数数组的元素类型；如果函数没有可变参数，则为零
protransform	regproc	pg_proc.oid	对该函数的调用可以由此处的另一个函数简化（参见第 38.10.10 节）
prokind	char		f 表示普通函数，p 表示过程，a 表示聚合函数，w 表示窗口函数
prosecdef	bool		函数是否为安全定义者函数（即 “setuid” 函数）
proleakproof	bool		该函数没有副作用。除返回值外，不会泄露任何关于参数的信息。任何可能因参数值而报错的函数都不是 leakproof。

名称	类型	引用	描述
proisstrict	bool		任一调用实参为 NULL 时，函数返回 NULL。在这种情况下，实际上根本不会调用该函数。非“严格”函数必须能够处理 NULL 输入。
proretset	bool		函数是否返回一个集合（即，指定数据类型的多个值）
provolatile	char		provolatile 表示函数的结果是否仅依赖输入实参，还是也受外部因素影响。i 表示“不变的”函数，相同输入总是产生相同结果。s 表示“稳定的”函数，对于固定输入，其结果在一次扫描内不变。v 表示“易失的”函数，其结果随时可能变化。（有副作用的函数也应使用 v，以免对它们的调用在优化时被消除。）
proparallel	char		proparallel 表示函数是否可以在并行模式下安全运行。s 表示可以不受限制地在并行模式下安全运行的函数。r 表示可以在并行模式下运行、但仅限并行组领导者执行的函数；并行工作者进程不能调用这些函数。u 表示在并行模式下不安全的函数；此类函数的存在会强制采用串行执行计划。
pronargs	int2		输入参数的个数
pronargdefaults	int2		具有默认值的参数个数
prorettype	oid	pg_type.oid	返回值的数据类型
proargtypes	oidvector	pg_type.oid	一个函数参数的数据类型的数组。这只包括输入参数（含INOUT 和VARIADIC 参数），因此也表现了函数的调用签名。
proallargtypes	oid[]	pg_type.oid	一个函数参数的数据类型的数组。这包括所有参数（含OUT 和INOUT 参数）。但是，如果所有参数都是IN 参数，此字段为 NULL。注意下标是从 1 开始，然而由于历史原因proargtypes 的下标是从 0 开始。

名称	类型	引用	描述
proargmodes	char[]		一个函数参数的模式的数组。这里包括：i 表示IN 参数，o 表示OUT 参数，b 表示INOUT 参数，v 表示VARIADIC 参数，t 表示TABLE 参数。如果所有的参数都是IN 参数，此字段为 NULL。注意这里的下标对应着 proallargtypes 而不是proargtypes 中的位置。
proargnames	text[]		一个函数参数的名字的数组。没有名字的参数在数组中设置为空字符串。如果没有一个参数有名字，此字段为 NULL。注意这里的下标对应着 proallargtypes 而不是proargtypes 中的位置。
proargdefaults	pg_node_tree		默认值的表达式树（以 nodeToString() 表示）。这是一个包含 pronargdefaults 个元素的列表，对应最后 *N* 个输入参数（即最后 *N* 个 proargtypes 位置）。如果没有参数具有默认值，则此字段为 NULL。
protrftypes	oid[]		要应用转换的数据类型 OID。
prosrc	text		此字段告诉函数处理器如何调用该函数。它可以是解释型语言函数的实际源代码、链接符号、文件名或其他内容，具体取决于实现语言和调用约定。
probin	text		关于如何调用函数的附加信息。其解释是与语言相关的。
proconfig	text[]		函数对于运行时配置变量的本地设置值
proacl	aclitem[]		访问权限，详见GRANT 和REVOKE

对于编译型函数，无论是内置函数还是动态加载的函数，prosrc 都包含函数的 C 语言名称（链接符号）。对于目前已知的所有其他语言类型，prosrc 包含函数的源文本。不使用 probin，但动态加载的 C 函数除外：此时它给出包含该函数的共享库文件名。

52.40. pg_publication

目录 pg_publication 包含数据库中创建的所有发布。关于发布的更多信息见第 31.1 节。

表 52.40. pg_publication 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
pubname	name		发布名称
pubowner	oid	pg_authid.oid	发布的拥有者
puballtables	bool		如果为真，则该发布会自动包括数据库中的所有表，包括将来创建的表。
pubinsert	bool		如果为真，则会为发布中的表复制INSERT 操作。
pubupdate	bool		如果为真，则会为发布中的表复制UPDATE 操作。
pubdelete	bool		如果为真，则会为发布中的表复制DELETE 操作。
pubtruncate	bool		如果为真，则会为发布中的表复制TRUNCATE 操作。

52.41. pg_publication_rel

目录 pg_publication_rel 包含数据库中关系与发布之间的映射。这是一种多对多映射。关于这些信息更方便用户查看的视图，参见第 52.78 节。

表 52.41. pg_publication_rel 列

名称	类型	引用	描述
prpubid	oid	pg_publication.oid	对发布的引用
prrelid	oid	pg_class.oid	对关系的引用

52.42. pg_range

目录 pg_range 存储关于范围类型的信息。它是类型在 pg_type 中项的补充。

表 52.42. pg_range 列

名称	类型	引用	描述
rngtypid	oid	pg_type.oid	范围类型的 OID
rngsubtype	oid	pg_type.oid	该范围类型的元素类型（子类型）的 OID
rngcollation	oid	pg_collation.oid	用于范围比较的排序规则的 OID，如果没有则为零
rngsubopc	oid	pg_opclass.oid	用于范围比较的子类型的操作符类的 OID

名称	类型	引用	描述
rngcanonical	regproc	pg_proc.oid	将一个范围值转换为规范形式的函数的 OID，如果没有则为零
rngsubdiff	regproc	pg_proc.oid	返回两个元素值之差并以 double precision 表示的函数的 OID；如果没有则为零

rngsubopc（如果元素类型支持排序规则，则还包括 rngcollation）决定范围类型所用的排序顺序。rngcanonical 用于元素类型为离散类型的情况。rngsubdiff 是可选的，但应当提供它，以提高范围类型上的 GiST 索引性能。

52.43. pg_replication_origin

pg_replication_origin 目录包含所有已创建的复制源。更多复制源的信息请见第 50 章。

和大部分系统目录不同，pg_replication_origin 在一个集簇的所有数据库之间共享：每个集簇只有一份 pg_replication_origin 拷贝，而不是每个数据库一份。

表 52.43. pg_replication_origin 列

名称	类型	引用	描述
roident	Oid		在整个集簇内唯一的复制源标识符。不应在本系统之外使用。
roname	text		外部的由用户定义的复制源名称。

52.44. pg_rewrite

目录 pg_rewrite 存储对于表和视图的重写规则。

表 52.44. pg_rewrite 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
rulename	name		规则名称
ev_class	oid	pg_class.oid	使用该规则的表
ev_type	char		使用该规则的事件类型：1 = SELECT，2 = UPDATE，3 = INSERT，4 = DELETE
ev_enabled	char		控制在哪种session_replication_role 模式中触发该规则。O = 规则在 “origin” 和 “local” 模式触发，D = 规则被禁用，R = 规则在 “replica” 模式触发，A = 规则总是被触发。

名称	类型	引用	描述
is_instead	bool		为真表示是一个INSTEAD 规则
ev_qual	pg_node_tree		规则条件的表达式树（以nodeToString() 表示）
ev_action	pg_node_tree		规则动作的查询树（以nodeToString() 表示）

注意

如果一个表在这个目录中有任何规则，pg_class.relhasrules 必须为真。

52.45. pg_seclabel

目录 pg_seclabel 存储数据库对象上的安全标签。安全标签可以通过 SECURITY LABEL 命令操纵。查看安全标签的更简单方法请见第 52.83 节。

同时请见 pg_shseclabel，它对集簇共享的数据库对象的安全标签执行相似的功能。

表 52.45. pg_seclabel 列

名称	类型	引用	描述
objoid	oid	任意 OID 列	该安全标签依附的对象的 OID
classoid	oid	pg_class.oid	该对象所在的系统目录的 OID
objsubid	int4		对于一个在表列上的安全标签，这将是列号（objoid 和classoid 指表本身）。对于所有其他对象类型，本列为 0。
provider	text		与该标签相关的标签提供者。
label	text		应用于该对象的安全标签。

52.46. pg_sequence

目录 pg_sequence 包含有关序列的信息。一些序列的信息（例如名称和模式）放在 pg_class 中。

表 52.46. pg_sequence 列

名称	类型	引用	描述
seqrelid	oid	pg_class.oid	这个序列的pg_class 项的 OID
seqtypid	oid	pg_type.oid	序列的数据类型

名称	类型	引用	描述
seqstart	int8		序列的起始值
seqincrement	int8		序列的增量值
seqmax	int8		序列的最大值
seqmin	int8		序列的最小值
seqcache	int8		序列的缓存大小
seqcycle	bool		序列是否循环

52.47. pg_shdepend

目录 pg_shdepend 记录数据库对象与共享对象（例如角色）之间的依赖关系。这些信息使得 PostgreSQL 可以确保对象在尝试删除前未被其他对象引用。

另请参阅 pg_depend，它对单个数据库中对象之间的依赖提供了相似的功能。

与大部分其他系统目录不同，pg_shdepend 在整个集簇的所有数据库之间共享：在每一个集簇中只有一个 pg_shdepend 的拷贝，而不是每个数据库一份。

表 52.47. pg_shdepend 列

名称	类型	引用	描述
dbid	oid	pg_database.oid	依赖对象所在数据库的 OID，对于共享对象为零
classid	oid	pg_class.oid	依赖对象所在系统目录的 OID
objid	oid	任意 OID 列	特定依赖对象的 OID
objsubid	int4		对于一个表列，这里是列号（objid 和classid 指表本身）。对于所有其他对象类型，此列为 0。
refclassid	oid	pg_class.oid	被引用对象所在的系统目录的 OID（必须是一个共享的目录）
refobjid	oid	任意 OID 列	指定被引用对象的 OID
deptype	char		定义此依赖关系语义的一个代码，见下文

在所有情况下，pg_shdepend 中的条目都表示：如果不同时删除依赖对象，就不能删除被引用对象。不过，还存在若干由 deptype 标识的子类型：

SHARED_DEPENDENCY_OWNER (o)

被引用对象（必须是一个角色）是依赖对象的拥有者。

SHARED_DEPENDENCY_ACL (a)

被引用对象（必须是角色）出现在依赖对象的 ACL（访问控制列表，即权限列表）中。（不会为对象拥有者创建 SHARED_DEPENDENCY_ACL 项，因为对象拥有者无论如何都会有一个 SHARED_DEPENDENCY_OWNER 项。）

SHARED_DEPENDENCY_POLICY (r)

被引用对象（必须是一个角色）被提及为依赖策略对象的目标。

SHARED_DEPENDENCY_PIN (p)

不存在依赖对象；这种项表示系统本身依赖于被引用对象，因此该对象绝不能删除。这种项只由initdb 创建。用于表示依赖对象的各列包含零。

SHARED_DEPENDENCY_TABLESPACE (t)

被引用对象（必须是一个表空间）被提及为某个不具有存储的关系所使用的表空间。

将来可能还需要其他依赖类型。特别要注意的是，当前定义只支持把角色和表空间作为被引用对象。

52.48. pg_shdescription

目录 pg_shdescription 存储共享数据库对象的可选描述（注释）。这些描述可以通过 COMMENT 命令设置或修改，并且可以使用psql 的\d 命令查看。

另请参阅 pg_description，它对单个数据库内对象的描述提供了相似的功能。

与大部分其他系统目录不同，pg_shdescription 在整个集簇的所有数据库之间共享：在每一个集簇中只有一个 pg_shdescription 的拷贝，而不是每个数据库一份。

表 52.48. pg_shdescription 列

名称	类型	引用	描述
objoid	oid	任意 OID 列	描述所属对象的 OID
classoid	oid	pg_class.oid	该对象所在的系统目录的 OID
description	text		作为该对象描述的任意文本

52.49. pg_shseclabel

目录 pg_shseclabel 存储共享数据库对象上的安全标签。安全标签可以通过 SECURITY LABEL 命令操纵。查看安全标签的更简单方法请见第 52.83 节。

另请参阅 pg_seclabel，它对单个数据库中对象的安全标签提供了相似的功能。

与大部分其他系统目录不同，pg_shseclabel 在整个集簇的所有数据库之间共享：在每一个集簇中只有一个 pg_shseclabel 的拷贝，而不是每个数据库一份。

表 52.49. pg_shseclabel 列

名称	类型	引用	描述
objoid	oid	任意 OID 列	该安全标签依附的对象的 OID
classoid	oid	pg_class.oid	该对象所在的系统目录的 OID

名称	类型	引用	描述
provider	text		与该标签相关的标签提供者。
label	text		应用于该对象的安全标签。

52.50. pg_statistic

目录 pg_statistic 存储有关数据库内容的统计数据。其中的项由 ANALYZE 创建，查询规划器会使用这些数据来进行查询规划。注意所有的统计数据天然就是近似的，即使这些数据是最新的。

通常，每个已分析的表列都有一个条目，其中stainherit = false。如果表具有继承子表，则还会创建第二个条目，其中stainherit = true。此行表示继承树上列的统计信息，即你可以通过SELECT *column* FROM *table** 看到的数据的统计信息，而stainherit = false 行表示 SELECT *column* FROM ONLY *table* 的结果。

pg_statistic 也存储关于索引表达式值的统计数据，就好像它们是真正的数据列，但在这种情况中starelid 指索引。对一个普通非表达式索引列不会创建项，因为它将是底层表列的项的冗余。当前，索引表达式的项都具有stainherit = false。

由于不同种类的数据可能适合不同种类的统计信息，pg_statistic 在设计上尽量不对所存储的统计信息种类作出假定。只有极为通用的统计信息（例如空值情况）才在 pg_statistic 中有专用的列。其余统计信息都存储在“槽位”中。每个槽位都是一组相关的列，其内容由其中一列的代码编号来标识。更多信息见 src/include/catalog/pg_statistic.h。

pg_statistic 不应公开可读，因为即使是表内容的统计信息也可能被认为是敏感的（例如，一个工资列的最大值和最小值就可能非常引人关注）。pg_stats 是建立在 pg_statistic 之上的公开可读视图，它只会显示当前用户可读取的表的信息。

表 52.50. pg_statistic 列

名称	类型	引用	描述
starelid	oid	pg_class.oid	被描述列所属的表或索引
staattnum	int2	pg_attribute.attnum	被描述列的编号
stainherit	bool		如果为真，则统计信息包括子表中的值，而不仅仅是指定关系中的值
stanullfrac	float4		列中空值所占的比例
stawidth	int4		非空项的平均存储宽度，以字节计
stadistinct	float4		列中不同非空数据值的数量。一个大于零的值是不同值的真正数目。一个小于零的值是表中行数的乘数的负值；例如，对于一个约 80% 的值为非空且每个非空值平均出现两次的列，可以表示为stadistinct = -0.4。一个 0 值表示不同值的数目未知。

名称	类型	引用	描述
stakind*N*	int2		一个代码，它表示存储在该pg_statistic 行中第*N* 个“槽位”的统计类型。
staop*N*	oid	pg_operator.oid	一个用于生成这些存储在第*N* 个“槽位”的统计信息的操作符。例如，直方图槽位会使用<操作符，该操作符定义了这些数据的排序顺序。
stanumbers*N*	float4[]		第*N* 个“槽位”中相应种类的数值统计信息；如果该槽位种类不涉及数值，则为 NULL
stavalues*N*	anyarray		第*N* 个“槽位”所需的相应种类的列数据值，如果该槽位类型不存储任何数据值则为 NULL。每个数组的元素值实际上都是指定列的数据类型或者是一个相关类型（如数组元素类型），因此，无法把这些列的类型定义得比anyarray 更具体。

52.51. pg_statistic_ext

目录 pg_statistic_ext 保存扩展的规划器统计信息。这个目录中的每一行对应于一个用 CREATE STATISTICS 创建的统计信息对象。

表 52.51. pg_statistic_ext 列

名称	类型	引用	描述
stxrelid	oid	pg_class.oid	包含这个对象所描述的列的表
stxname	name		统计信息对象的名称
stxnamespace	oid	pg_namespace.oid	包含这个统计信息对象的名字空间的 OID
stxowner	oid	pg_authid.oid	统计信息对象的拥有者
stxkeys	int2vector	pg_attribute.attnum	一个属性编号的数组，表示哪些表列被这个统计信息对象覆盖。例如值1 3 表示第一个和第三个表列被覆盖
stxkind	char[]		包含启用的统计类型代码的数组，可用值有：d 表示非重复值统计信息，f 表示函数依赖统计信息
stxndistinct	pg_ndistinct		非重复值计数，序列化为pg_ndistinct 类型

名称	类型	引用	描述
stxdependencies	pg_dependencies		函数依赖统计信息，序列化为pg_dependencies 类型

stxkind 字段在创建统计信息对象时填充，用于指示所需的统计信息类型。其后的各字段初始为 NULL，只有在ANALYZE 计算出相应统计信息后才会填充。

52.52. pg_subscription

目录 pg_subscription 包含所有现有的逻辑复制订阅。更多有关逻辑复制的信息请见第 31 章。

和大部分系统目录不同，pg_subscription 在集簇的所有数据库之间共享：每个集簇只有一份 pg_subscription 拷贝，而不是每个数据库一份。

对列 subconninfo 的访问被从普通用户那里收回，因为该列可能含有明文密码。

表 52.52. pg_subscription 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
subdbid	oid	pg_database.oid	订阅所在的数据库的 OID
subname	name		订阅的名称
subowner	oid	pg_authid.oid	订阅的拥有者
subenabled	bool		如果为真，则订阅已启用，并且应该正在进行复制。
subsynccommit	text		用于订阅工作者的 synchronous_commit 设置。
subconninfo	text		到上游数据库的连接字符串
subslotname	name		上游数据库中复制槽的名称（也用于本地复制源名称）；NULL 表示 NONE
subpublications	text[]		被订阅的发布名称数组。它们引用的是发布者服务器上的发布。关于发布的更多信息见第 31.1 节。

52.53. pg_subscription_rel

目录 pg_subscription_rel 包含每个订阅中每个被复制关系的状态。这是一种多对多映射。

此目录只包含在运行 CREATE SUBSCRIPTION 或 ALTER SUBSCRIPTION ... REFRESH PUBLICATION 之后，对该订阅已知的表。

表 52.53. pg_subscription_rel 列

名称	类型	引用	描述
srsubid	oid	pg_subscription.oid	对订阅的引用
srrelid	oid	pg_class.oid	对关系的引用
srsubstate	char		状态代码：i = 初始化，d = 数据正在被复制，s = 已同步，r = 就绪（正常复制）
srsublsn	pg_lsn		在s 或r 状态中，用于同步协调的状态更改的远程 LSN，否则为 NULL

52.54. pg_tablespace

目录 pg_tablespace 存储关于可用表空间的信息。表可以被放置在特定表空间中以实现磁盘布局的管理。

与大部分其他系统目录不同，pg_tablespace 在整个集簇的所有数据库之间共享：在每一个集簇中只有一个 pg_tablespace 的拷贝，而不是每个数据库一份。

表 52.54. pg_tablespace 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
spcname	name		表空间名
spcowner	oid	pg_authid.oid	表空间的拥有者，通常是创建它的用户
spcacl	aclitem[]		访问权限，详见GRANT 和REVOKE
spcoptions	text[]		表空间级别的选项，形如 “keyword=value” 的字符串

52.55. pg_transform

目录 pg_transform 存储有关转换的信息，转换是一种让数据类型适应过程语言的机制。详见 CREATE TRANSFORM。

表 52.55. pg_transform 列

名称	类型	引用	描述
trftype	oid	pg_type.oid	这个转换所针对的数据类型的 OID
trflang	oid	pg_language.oid	这个转换所针对的语言的 OID

名称	类型	引用	描述
trffromsql	regproc	pg_proc.oid	用于将数据类型转换为过程语言的输入（例如函数参数）的函数的 OID。如果不支持此操作，则存储零。
trftosql	regproc	pg_proc.oid	用于将过程语言的输出（例如返回值）转换为该数据类型的函数的 OID。如果不支持此操作，则存储零。

52.56. pg_trigger

目录 pg_trigger 存储表和视图上的触发器。详见 CREATE TRIGGER。

表 52.56. pg_trigger 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
tgrelid	oid	pg_class.oid	触发器所在的表
tgname	name		触发器名（在同一个表的触发器中必须唯一）
tgfoid	oid	pg_proc.oid	要被触发器调用的函数
tgtype	int2		标识触发器触发条件的位掩码
tgenabled	char		控制触发器在session_replication_role 模式中的触发。O = 触发器在 “origin” 和 “local” 模式触发，D = 触发器被禁用，R = 触发器在 “replica” 模式触发，A = 触发器总是触发。
tgisinternal	bool		为真表示触发器是内部生成的（通常是为了强制执行由tgconstraint 指定的约束）
tgconstrrelid	oid	pg_class.oid	被一个引用完整性约束引用的表
tgconstrindid	oid	pg_class.oid	支持一个唯一、主键、引用完整性约束或者排他约束的索引
tgconstraint	oid	pg_constraint.oid	与触发器相关的pg_constraint 项（如有）
tgdeferrable	bool		如果约束触发器可延迟则为真
tginitdeferred	bool		如果约束触发器初始处于延迟状态则为真

名称	类型	引用	描述
tgnargs	int2		传递给触发器函数的参数字符串个数
tgattr	int2vector	pg_attribute.attnum	如果触发器是列限定的，这里存放列号；否则这是一个空数组
tgargs	bytea		传递给触发器的参数字符串，每一个都以 NULL 结尾
tgqual	pg_node_tree		触发器WHEN 条件的表达式树（以nodeToString() 表示）；如果没有则为 NULL
tgoldtable	name		OLD TABLE 在 REFERENCING 子句中指定的名称；如果没有，则为 NULL
tgnewtable	name		NEW TABLE 在 REFERENCING 子句中指定的名称；如果没有，则为 NULL

当前，列限定触发器只支持UPDATE 事件，因此tgattr 只用于这种事件类型。tgtype 也可以包含用于其他事件类型的位，但那些事件类型被视为针对全表，而不考虑 tgattr 的内容。

注意

当 tgconstraint 非零时，tgconstrrelid、tgconstrindid、tgdeferrable 和 tginitdeferred 与被引用的 pg_constraint 项有很大的冗余。但是，存在将一个不可延迟触发器关联到一个可延迟约束的可能性：外键约束可以有一些可延迟和一些不可延迟触发器。

注意

如果一个关系在本目录中拥有任何触发器，其pg_class.relhastriggers 必须为真。

52.57. pg_ts_config

pg_ts_config 系统目录包含表示文本检索配置的条目。一个配置指定某个特定的文本检索解析器，以及针对该解析器每种输出词元类型所定义的词典列表。解析器记录在pg_ts_config 条目中，而词元到词典的映射则由pg_ts_config_map 中的辅助项定义。

PostgreSQL 的文本检索特性在第 12 章中有更详尽的描述。

表 52.57. pg_ts_config 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
cfgname	name		文本检索配置名
cfgnamespace	oid	pg_namespace.oid	包含该配置的名字空间的 OID
cfgowner	oid	pg_authid.oid	配置的拥有者
cfgparser	oid	pg_ts_parser.oid	该配置的文本检索解析器的 OID

52.58. pg_ts_config_map

pg_ts_config_map 系统目录中的条目说明了，对于每个文本检索配置所用解析器的每种输出词元类型，应当查询哪些文本检索词典以及查询的顺序。

PostgreSQL 的文本检索特性在第 12 章中有更详尽的描述。

表 52.58. pg_ts_config_map 列

名称	类型	引用	描述
mapcfg	oid	pg_ts_config.oid	拥有该映射项的pg_ts_config 项的 OID
maptokentype	integer		一种由配置的解析器送出的词元类型
mapseqno	integer		查询该项的顺序（mapseqno 值小的优先）
mapdict	oid	pg_ts_dict.oid	要查询的文本检索词典的 OID

52.59. pg_ts_dict

pg_ts_dict 系统目录包含定义文本检索词典的项。一个词典依赖于一个文本检索模板，它指定了所有需要的实现函数，词典本身则为模板支持的用户可设置参数提供值。这种分工允许普通用户创建词典。参数由一个文本串dictinitoption 定义，其格式和意义随着模板而变化。

PostgreSQL 的文本检索特性在第 12 章中有更详尽的描述。

表 52.59. pg_ts_dict 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
dictname	name		文本检索词典名
dictnamespace	oid	pg_namespace.oid	包含该词典的名字空间的 OID
dictowner	oid	pg_authid.oid	词典的拥有者
dicttemplate	oid	pg_ts_template.oid	该词典的文本检索模板的 OID

名称	类型	引用	描述
dictinitoption	text		模板的初始化选项字符串

52.60. pg_ts_parser

pg_ts_parser 系统目录包含定义文本检索解析器的项。一个解析器负责将输入文本分割成词位并为每一个词位分配一个词元类型。由于一个解析器必须用 C 语言级别的函数实现，创建新解析器的工作只限于数据库的超级用户。

PostgreSQL 的文本检索特性在第 12 章中有更详尽的描述。

表 52.60. pg_ts_parser 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
prsname	name		文本检索解析器的名字
prsnamespace	oid	pg_namespace.oid	包含此解析器的名字空间的 OID
prsstart	regproc	pg_proc.oid	解析器启动函数的 OID
prstoken	regproc	pg_proc.oid	解析器的下一词元函数的 OID
prsend	regproc	pg_proc.oid	解析器的关闭函数的 OID
prsheadline	regproc	pg_proc.oid	解析器的 headline 函数的 OID
prslextype	regproc	pg_proc.oid	解析器的 lextype 函数的 OID

52.61. pg_ts_template

pg_ts_template 系统目录包含定义文本检索模板的项。一个模板是一类文本检索词典的实现骨架。由于一个模板必须用 C 语言级别的函数实现，新模板的创建只限于数据库超级用户。

PostgreSQL 的文本检索特性在第 12 章中有更详尽的描述。

表 52.61. pg_ts_template 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
tmplname	name		文本检索模板的名字
tmplnamespace	oid	pg_namespace.oid	包含此模板的名字空间的 OID
tmplinit	regproc	pg_proc.oid	模板的初始化函数的 OID
tmpllexize	regproc	pg_proc.oid	模板的 lexize 函数的 OID

52.62. pg_type

目录 pg_type 存储有关数据类型的信息。基础类型和枚举类型（标量类型）使用 CREATE TYPE 创建，而域使用CREATE DOMAIN 创建。数据库中的每一个表都会有一个自动创建的复合类型，用于表示表的行结构。也可以使用CREATE TYPE AS 创建复合类型。

表 52.62. pg_type 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
typname	name		数据类型的名字
typnamespace	oid	pg_namespace.oid	包含此类型的名字空间的 OID
typowner	oid	pg_authid.oid	类型的拥有者
typlen	int2		对于一个固定尺寸的类型，typlen 是该类型内部表示的字节数。对于一个变长类型，typlen 为负值。-1 表示一个 “varlena” 类型（具有长度字），-2 表示一个以空字符结尾的 C 字符串。
typbyval	bool		typbyval 判断内部例程传递这个类型的值时是通过传值还是传引用。如果typlen 不是 1、2 或 4（或者在 Datum 为 8 字节的机器上为 8），则 typbyval 最好是假。变长类型总是传引用。注意即使长度允许传值，typbyval 也可以为假。
typtype	char		typtype 可以是：b 表示一个基础类型，c 表示一个复合类型（例如一个表的行类型），d 表示一个域，e 表示一个枚举类型，p 表示一个伪类型，r 表示一个范围类型。另请参阅typrelid 和 typbasetype。
typcategory	char		typcategory 是一种任意的数据类型分类，它被解析器用来决定应 “首选” 哪些隐式类型转换。参见表 52.63。
typispreferred	bool		如果此类型在它的 typcategory 中是首选的类型转换目标，此列为真

名称	类型	引用	描述
typisdefined	bool		如果此类型已被定义则为真，如果此类型只是一个表示还未定义类型的占位符则为假。当typisdefined为假，除了类型名字、名字空间和 OID 之外什么都不能被依赖。
typdelim	char		在分析数组输入时，分隔两个此类型值的字符。注意该分隔符是与数组元素数据类型相关联的，而不是和数组的数据类型关联。
typrelid	oid	pg_class.oid	如果这是一个复合类型（见typtype），那么此列指向pg_class中定义对应表的项（对于独立定义的复合类型，pg_class项并不表示一个表，但不管怎样该类型的pg_attribute项需要链接到它）。对非复合类型此列为零。
typelem	oid	pg_type.oid	如果typelem不为 0，则它标识pg_type中的另一行。此时可以像数组一样对当前类型使用下标，得到typelem类型的值。“真正的”数组类型是变长的（typlen = -1），但某些定长类型（typlen > 0）的typelem也非零，例如name和point。如果定长类型具有typelem，则其内部表示必须是若干个typelem数据类型的值，不含其他数据。变长数组类型的首部由数组子例程定义。
typarray	oid	pg_type.oid	如果typarray不是零，则它标识pg_type中的另一行，这一行是一个将此类型作为元素的“真正的”数组类型
typinput	regproc	pg_proc.oid	输入转换函数（文本格式）
typoutput	regproc	pg_proc.oid	输出转换函数（文本格式）
typreceive	regproc	pg_proc.oid	输入转换函数（二进制格式），如果没有则为零
typsend	regproc	pg_proc.oid	输出转换函数（二进制格式），如果没有则为零

名称	类型	引用	描述
typmodin	regproc	pg_proc.oid	类型修饰符输入函数，如果类型不支持修饰符则为零
typmodout	regproc	pg_proc.oid	类型修饰符输出函数，或为零以使用标准格式
typanalyze	regproc	pg_proc.oid	自定义ANALYZE 函数，零表示使用标准函数
typalign	char		typalign 表示存储此类型值时所要求的对齐方式。它适用于磁盘存储，以及该值在 PostgreSQL 内部的大多数表示形式。当多个值连续存放时，例如在磁盘上完整行的表示中，这种类型的数据前会插入填充，使其从指定边界开始。对齐参考点是该序列中第一个数据的起始位置。 可能的值为： • c = char 对齐，即不需要对齐。 • s = short 对齐（在大部分机器上为 2 字节）。 • i = int 对齐（在大部分机器上为 4 字节）。 • d = double 对齐（在很多机器上为 8 字节，但绝不是全部）。 **注意** 对于系统表中使用的类型，关键的是在pg_type 定义的大小和对齐与编译器在表示表行的结构体中布局列的方式一致。

名称	类型	引用	描述
typstorage	char		typstorage 用于指示 varlena 类型（即 typlen = -1）是否支持 TOAST，以及这种类型的属性应采用什么默认策略。可能的值为： • p：值必须始终以普通方式存储。 • e：值可以存储在一个“二级”关系中（如果原关系有这样的关系，参见pg_class.reltoastrelid）。 • m：值可以压缩后存储在行内。 • x：值可以压缩后存储在行内，或存储在“二级”存储中。 注意，m 列也可以被移动到二级存储，但只能作为最后手段（e 和x 列会先被移动）。
typnotnull	bool		typnotnull 表示类型上的一个非空约束。只用于域。
typbasetype	oid	pg_type.oid	如果这是一个域（见 typtype），则typbasetype 标识该域所基于的类型。如果此类型不是域，则为 0。
typtypmod	int4		域使用typtypmod 来记录被应用于它们基础类型的typmod（如果基础类型不使用typmod，则为 -1）。如果此类型不是一个域则为-1。
typndims	int4		对于基于数组的域，typndims 是数组维度数（即，typbasetype 是一个数组类型）。对于不是基于数组的域的类型，此列为零。

名称	类型	引用	描述
typcollation	oid	pg_collation.oid	typcollation 指定此类型的排序规则。如果类型不支持排序规则，此列为零。支持排序规则的基础类型在这里会有 DEFAULT_COLLATION_OID。基于支持排序规则的类型的域，如果为其指定了排序规则 OID，则它的排序规则 OID 可以不同于其基础类型的排序规则 OID。
typdefaultbin	pg_node_tree		如果typdefaultbin 非 NULL，那么它就是该类型默认表达式的nodeToString() 表示形式。此列只用于域。
typdefault	text		如果某类型没有相关默认值，那么typdefault 为 NULL。如果typdefaultbin 不为 NULL，那么typdefault 必须包含由typdefaultbin 表示的默认表达式的人类可读版本。如果typdefaultbin 为 NULL 而typdefault 不为 NULL，则typdefault 是该类型默认值的外部表示形式，它可以被交给该类型的输入转换器来产生一个常量。
typacl	aclitem[]		访问权限，详见GRANT 和REVOKE

表 52.63 列出了 typcategory 的系统定义值。任何未来对此列表的增加都将是大写 ASCII 字母。所有其他 ASCII 字符都保留给用户定义的类别。

表 52.63. typcategory 代码

代码	类别
A	数组类型
B	布尔类型
C	复合类型
D	日期/时间类型
E	枚举类型
G	几何类型
I	网络地址类型
N	数值类型
P	伪类型
R	范围类型

代码	类别
S	字符串类型
T	时间间隔类型
U	用户定义类型
V	位字符串类型
X	unknown 类型

52.63. pg_user_mapping

目录 pg_user_mapping 存储从本地用户到远程用户的映射。对这个目录的访问对普通用户有限制，可使用视图pg_user_mappings 替代。

表 52.64. pg_user_mapping 列

名称	类型	引用	描述
oid	oid		行标识符（隐藏属性，必须显式选择）
umuser	oid	pg_authid.oid	将要被映射的本地角色的 OID，如果用户映射是公共的则为零
umserver	oid	pg_foreign_server.oid	包含此映射的外部服务器的 OID
umoptions	text[]		用户映射特定的选项，以 “keyword=value” 字符串形式

52.64. 系统视图

除了系统目录之外，PostgreSQL 还提供了一些内置视图。一些系统视图提供了对系统目录中一些常用查询的便捷访问。其他视图则提供了对内部服务器状态的访问。

信息模式（第 37 章）提供了一组与系统视图功能重叠的替代视图。由于信息模式是 SQL 标准的一部分，而这里描述的视图是PostgreSQL 特有的，所以如果信息模式提供了所需的全部信息，通常最好使用信息模式。

表 52.65 列出了这里描述的系统视图。每个视图的更详细文档见下文。还有一些额外的视图提供了对统计信息收集器结果的访问；它们在表 28.2 中描述。

除非另有说明，否则这里描述的所有视图都是只读的。

表 52.65. 系统视图

视图名称	用途
pg_available_extensions	可用的扩展
pg_available_extension_versions	扩展的可用版本
pg_config	编译时配置参数
pg_cursors	打开的游标
pg_file_settings	配置文件内容摘要
pg_group	数据库用户组

视图名称	用途
pg_hba_file_rules	客户端认证配置文件内容的摘要
pg_indexes	索引
pg_locks	当前持有或等待的锁
pg_matviews	物化视图
pg_policies	策略
pg_prepared_statements	预备语句
pg_prepared_xacts	预备事务
pg_publication_tables	发布及其关联表的信息
pg_replication_origin_status	有关复制源的信息，包括复制进度
pg_replication_slots	复制槽信息
pg_roles	数据库角色
pg_rules	规则
pg_seclabels	安全标签
pg_sequences	序列
pg_settings	参数设置
pg_shadow	数据库用户
pg_stats	规划器统计信息
pg_tables	表
pg_timezone_abbrevs	时区缩写
pg_timezone_names	时区名称
pg_user	数据库用户
pg_user_mappings	用户映射
pg_views	视图

52.65. pg_available_extensions

pg_available_extensions 视图列出了可供安装的扩展。另请参阅 pg_extension 目录，其中显示当前已安装的扩展。

表 52.66. pg_available_extensions 列

名称	类型	描述
name	name	扩展名
default_version	text	默认版本的名称，如果没有指定则为NULL
installed_version	text	当前已安装的扩展版本，如果没有安装则为NULL
comment	text	来自于扩展的控制文件的注释字符串

pg_available_extensions 视图是只读的。

52.66. pg_available_extension_versions

pg_available_extension_versions 视图列出了可供安装的特定扩展版本。另请参阅 pg_extension 目录，其中显示当前已安装的扩展。

表 52.67. pg_available_extension_versions 列

名称	类型	描述
name	name	扩展名
version	text	版本名
installed	bool	如果此版本的扩展当前已安装则为真
superuser	bool	如果只有超级用户可以安装此扩展，则为真
relocatable	bool	如果扩展能被重定位到另一个模式则为真
schema	name	此扩展必须被安装到的模式名，如果此扩展是部分或者全部可以重定位的，此列为NULL
requires	name[]	所依赖扩展的名称，如果没有则为NULL
comment	text	来自于扩展的控制文件的注释字符串

pg_available_extension_versions 视图是只读的。

52.67. pg_config

视图 pg_config 描述了当前安装版本的 PostgreSQL 的编译时配置参数。例如，它可供希望与 PostgreSQL 交互的软件包使用，以便找到所需的头文件和库。它提供与PostgreSQL 客户端应用程序 pg_config 相同的基本信息。

默认情况下，pg_config 视图只能被超级用户读取。

表 52.68. pg_config 列

名称	类型	描述
name	text	参数名
setting	text	参数值

52.68. pg_cursors

该 pg_cursors 视图列出当前可用的游标。游标可以通过几种方式定义：

- 通过 SQL 中的 DECLARE 语句
- 通过前端/后端协议中的 Bind 消息，如第 53.2.3 节中所述
- 通过服务器编程接口（SPI），如第 47.1 节中所述

该 pg_cursors 视图会显示通过上述任意方式创建的游标。游标仅在定义它的事务持续期间存在，除非将其声明为WITH HOLD。因此，不可保持的游标只会在此视图中保留到创建它的事务结束为止。

> **注意**
>
> 游标在内部用于实现 PostgreSQL 的某些组件，如过程语言。因此，pg_cursors 视图可能包含用户未显式创建的游标。

表 52.69. pg_cursors 列

名称	类型	描述
name	text	游标名
statement	text	用于声明此游标的原始查询字符串
is_holdable	boolean	如果游标是可保持游标（即，在声明该游标的事务提交后仍可访问）则为true，否则为false
is_binary	boolean	如果游标被声明为 BINARY，则为 true；否则为 false
is_scrollable	boolean	如果游标是可滚动的（即，允许以一种非顺序的方式检索行）则为true，否则为false
creation_time	timestamptz	游标被声明的时间

pg_cursors 视图是只读的。

52.69. pg_file_settings

视图 pg_file_settings 提供了服务器配置文件的内容摘要。每个“name = value”条目在文件中出现时，此视图中会出现一行，并附有标注，指示该值是否能够成功应用。还可能出现额外的行，用于表示与“name = value”条目无关的问题，例如文件中的语法错误。

这个视图对于检查计划中的配置文件变更是否有效，或者诊断之前的故障很有帮助。请注意，这个视图报告的是文件的当前内容，而不是服务器上最后应用的内容。（通常可以通过 pg_settings 视图确定最后一次由服务器应用的内容。）

默认情况下，pg_file_settings 视图只能被超级用户读取。

表 52.70. pg_file_settings 列

名称	类型	描述
sourcefile	text	配置文件的完整路径名
sourceline	integer	该项在配置文件中出现的行号
seqno	integer	项被处理的顺序（1..*n*）
name	text	配置参数名
setting	text	将赋给该参数的值
applied	boolean	如果该值能够成功应用则为真

名称	类型	描述
error	text	如果非空，则为说明该条目为何不能应用的错误消息

如果配置文件包含语法错误或无效的参数名称，服务器将不会尝试应用其中的任何设置，因此所有applied 字段将读取为 false。在这种情况下，将会有一个或多个带有非空error 字段的行，指示问题所在。否则，会在可能的情况下应用各个设置。如果无法应用某个单独的设置（例如，无效值，或者在服务器启动后无法更改设置），则error 字段中将有适当的消息。另一种导致条目的 applied = false 的方式是被后续相同参数名称的条目覆盖；这种情况不被视为错误，因此 error 字段中不会出现任何内容。

查看第 19.1 节以获取有关更改运行时参数的各种方法的更多信息。

52.70. pg_group

视图 pg_group 存在是为了向后兼容性：它模拟了在 PostgreSQL 版本 8.1 之前存在的目录。它显示了所有 rolcanlogin 为假的角色的名称和成员，这是对被用作组的角色集合的近似。

表 52.71. pg_group 列

名称	类型	引用	描述
groname	name	pg_authid.rolname	组的名称
grosysid	oid	pg_authid.oid	组 ID
grolist	oid[]	pg_authid.oid	包含此组中角色 ID 的一个数组

52.71. pg_hba_file_rules

视图 pg_hba_file_rules 提供客户端认证配置文件 pg_hba.conf 内容的摘要。文件中每个非空且非注释的行都会在该视图中显示一行，并带有注解，说明该规则是否能成功应用。

该视图有助于检查认证配置文件中的计划变更是否可行，或诊断先前的失败。请注意，该视图报告的是文件的当前内容，而不是服务器上次加载的内容。

默认情况下，pg_hba_file_rules 视图只能由超级用户读取。

表 52.72. pg_hba_file_rules 列

名称	类型	描述
line_number	integer	该规则在 pg_hba.conf 中的行号
type	text	连接类型
database	text[]	这条规则应用的数据库名列表
user_name	text[]	这条规则应用的用户及组名列表
address	text	主机名或 IP 地址，或者all、samehost、samenet 之一，对于本地连接为 NULL
netmask	text	IP 地址掩码，如果不适用则为 NULL
auth_method	text	认证方法

名称	类型	描述
options	text[]	为认证方法指定的选项（如果有）
error	text	如果非空，则为说明该行为何无法处理的错误消息

通常，反映错误条目的行只有 line_number 和 error 字段有值。

关于客户端认证配置的更多信息，参见第 20 章。

52.72. pg_indexes

视图 pg_indexes 提供了有关数据库中每个索引的有用信息。

表 52.73. pg_indexes 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表和索引的模式名
tablename	name	pg_class.relname	此索引的基表名称
indexname	name	pg_class.relname	索引名
tablespace	name	pg_tablespace.spcname	包含索引的表空间名（如使用数据库默认表空间，则为 NULL）
indexdef	text		索引定义（重建出的 CREATE INDEX 命令）

52.73. pg_locks

视图 pg_locks 提供了对数据库服务器中活动进程持有的锁的信息的访问。有关锁定的更多讨论，请参见第 13 章。

pg_locks 为每个活动的可锁定对象、请求的锁模式及相关进程的组合提供一行。因此，如果多个进程正在持有或等待锁定它，同一可锁定对象可能会出现多次。但是，当前没有任何锁定的对象将不会出现。

有几种不同类型的可锁定对象：整个关系（例如，表），关系的单个页面，关系的单个元组，事务 ID（虚拟和永久 ID 均包括），以及一般的数据库对象（由类 OID 和对象 OID 标识，与 pg_description 或 pg_depend 中的方式相同）。此外，扩展关系的权利被表示为单独的可锁定对象，更新 pg_database.datfrozenxid 的权利也是如此。此外，还可以对具有用户定义含义的数字施加“咨询锁”。

表 52.74. pg_locks 列

名称	类型	引用	描述
locktype	text		可锁定对象的类型：relation、extend、frozenid、page、tuple、transactionid、virtualxid、object、userlock 或advisory

名称	类型	引用	描述
database	oid	pg_database.oid	锁目标存在的数据库的 OID，如果目标是一个共享对象则为 0，如果目标是一个事务 ID 则为 NULL
relation	oid	pg_class.oid	作为锁目标的关系的 OID，如果目标既不是关系，也不是关系的一部分则此列为 NULL
page	integer		作为锁目标的页在关系中的页号，如果目标不是一个关系页或元组则此列为 NULL
tuple	smallint		作为锁目标的元组在页中的元组号，如果目标不是一个元组则此列为 NULL
virtualxid	text		锁定目标事务的虚拟 ID，如果目标不是虚拟事务 ID，则为 NULL
transactionid	xid		锁定目标事务的标识，如果目标不是事务 ID，则为 NULL
classid	oid	pg_class.oid	包含锁目标的系统目录的 OID，如果目标不是一个普通数据库对象则此列为 NULL
objid	oid	任意 OID 列	锁目标在它的系统目录中的 OID，如果目标不是一个普通数据库对象则为 NULL
objsubid	smallint		锁的目标列号（classid 和objid 指表本身），如果目标是某种其他普通数据库对象则此列为 0，如果目标不是一个普通数据库对象则此列为 NULL
virtualtransaction	text		持有这个锁或者正在等待这个锁的事务的虚拟 ID
pid	integer		持有这个锁或者正在等待这个锁的服务器进程的 PID，如果此锁被一个预备事务所持有则此列为 NULL
mode	text		此进程已持有或者希望持有的锁模式的名称（参见第 13.3.1 节和第 13.2.3 节）
granted	boolean		如果锁已授予则为真，如果锁被等待则为假
fastpath	boolean		如果锁通过快速路径获得则为真，通过主锁表获得则为假

在表示由指定进程持有的锁的行中，granted 为真。为假表示该进程当前正在等待获取此锁，这意味着至少有一个其他进程正在同一可锁定对象上持有或等待获取冲突的锁模式。等待的进程会休眠，直到另一个锁被释放（或检测到死锁情况）。单个进程一次最多只能等待获取一个锁。

在执行事务期间，服务器进程会对事务的虚拟事务 ID 持有独占锁。如果为事务分配了永久 ID（通常仅在事务改变数据库状态时才会发生），它还会对事务的永久事务 ID 持有独占锁直到事务结束。当一个进程发现有必要等待另一个事务结束时，它会尝试获取另一个事务 ID（根据情况是虚拟 ID 还是永久 ID）的共享锁。只有当另一个事务终止并释放其锁时，这才会成功。

虽然元组是一种可锁定的对象类型，但关于行级锁的信息存储在磁盘上，而不是内存中，因此行级锁通常不会出现在此视图中。如果一个进程正在等待行级锁，它通常会出现在视图中，等待当前行锁持有者的永久事务 ID。

咨询锁可以在由单个 bigint 值或两个整数值组成的键上获取。一个 bigint 键在 classid 列中显示其高位半部分，在objid 列中显示其低位半部分，并且objsubid 等于 1。可以使用表达式 (classid::bigint << 32) | objid::bigint 重新组装原始 bigint 值。整数键在 classid 列中显示第一个键，在objid 列中显示第二个键，并且 objsubid 等于 2。键的实际含义由用户自行决定。咨询锁在每个数据库内都是本地的，因此 database 列对于咨询锁是有意义的。

pg_locks 提供了集簇中所有锁的全局视图，不仅包括与当前数据库相关的锁。虽然它的 relation 列可以与 pg_class.oid 连接来识别被锁定的关系，但这仅对当前数据库中的关系有效（即 database 列为当前数据库的 OID 或零的那些关系）。

pid 列可以与 pg_stat_activity 视图的 pid 列进行连接，以获取有关持有或等待每个锁的会话的更多信息，例如

```
SELECT * FROM pg_locks pl LEFT JOIN pg_stat_activity psa
    ON pl.pid = psa.pid;
```

另外，如果你正在使用预备事务，virtualtransaction 列可以与 pg_prepared_xacts 视图的 transaction 列进行连接，以获取有关持有锁的预备事务的更多信息。（预备事务永远不会等待锁，但它继续持有其在运行时获取的锁。）例如：

```
SELECT * FROM pg_locks pl LEFT JOIN pg_prepared_xacts ppx
    ON pl.virtualtransaction = '-1/' || ppx.transaction;
```

尽管可以通过将 pg_locks 与自身连接来获取关于哪些进程阻塞了哪些其他进程的信息，但在细节上很难做到准确。这样的查询将不得不写明各锁模式之间的冲突规则。更糟糕的是，pg_locks 视图不公开关于哪些进程在锁等待队列中领先于哪些其他进程的信息，也不公开关于哪些进程是代表哪些其他客户端会话运行的并行工作者的信息。最好使用 pg_blocking_pids() 函数（参见表 9.60）来识别等待进程被哪些进程阻塞。

pg_locks 视图显示来自常规锁管理器和谓词锁管理器的数据，这两个是独立的系统；此外，常规锁管理器将其锁分为常规锁和快速路径锁。不能保证这些数据完全一致。当查询该视图时，快速路径锁的数据（具有fastpath = true）从每个后端逐个收集，而不会冻结整个锁管理器的状态，因此在收集信息时可能会发生锁的获取或释放。但请注意，这些锁已知不会与当前持有的任何其他锁发生冲突。在查询所有后端的快速路径锁后，常规锁管理器的剩余部分将被整体锁定，并且所有剩余锁的一致快照将作为一个原子操作收集。解锁常规锁管理器后，谓词锁管理器类似地被锁定，并且所有谓词锁将作为一个原子操作收集。因此，除了快速路径锁外，每个锁管理器将提供一致的结果集，但由于我们不同时锁定两个锁管理器，因此在询问常规锁管理器后和在询问谓词锁管理器前，可能会发生锁的获取或释放。

如果这个视图被非常频繁地访问，锁定常规和（或）谓词锁管理器可能会对数据库性能产生一些影响。锁仅在获取来自锁管理器的数据所需的最短时间内保持，但这并不能完全消除性能影响的可能性。

52.74. pg_matviews

视图 pg_matviews 提供了对数据库中每个物化视图的有用信息的访问。

表 52.75. pg_matviews 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含物化视图的模式名称
matviewname	name	pg_class.relname	物化视图名称
matviewowner	name	pg_authid.rolname	物化视图拥有者名称
tablespace	name	pg_tablespace.spcname	包含物化视图的表空间名（如使用数据库默认表空间则为 NULL）
hasindexes	boolean		如果物化视图有（或者最近有过）任何索引，则此列为真
ispopulated	boolean		如果物化视图当前已被填充，则此列为真
definition	text		物化视图的定义（一个重建的SELECT 查询）

52.75. pg_policies

视图 pg_policies 提供了有关数据库中每个行级安全策略的有用信息。

表 52.76. pg_policies 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含策略所在表的模式的名称
tablename	name	pg_class.relname	策略所在表的名称
policyname	name	pg_policy.polname	策略名称
polpermissive	text		策略是宽松的还是限制性的?
roles	name[]		这个策略适用的角色
cmd	text		这个策略适用的命令类型
qual	text		作为这个策略适用的查询的安全屏障条件增加的表达式
with_check	text		作为尝试向该表增加行的查询的 WITH CHECK 条件增加的表达式

52.76. pg_prepared_statements

pg_prepared_statements 视图显示当前会话中所有可用的预备语句。关于预备语句的更多信息，参见 PREPARE。

pg_prepared_statements 为每个预备语句包含一行。创建新的预备语句时，会向该视图添加一行；释放预备语句时（例如通过DEALLOCATE 命令）则会移除该行。

表 52.77. pg_prepared_statements 列

名称	类型	描述
name	text	预备语句的标识符
statement	text	客户端提交用于创建此预备语句的查询字符串。对于通过 SQL 创建的预备语句，这里是由客户端提交的PREPARE 语句。对于通过前端/后端协议创建的预备语句，这里是预备语句本身的文本。
prepare_time	timestamptz	预备语句被创建的时间
parameter_types	regtype[]	预备语句期望的参数类型，以一个 regtype 数组的形式。这个数组中一个元素所对应的 OID 可通过将 regtype 值转换为oid 获得。
from_sql	boolean	如果预备语句通过 SQL 命令 PREPARE 创建，则为true；如果预备语句通过前端/后端协议创建，则为false

pg_prepared_statements 视图是只读的。

52.77. pg_prepared_xacts

视图 pg_prepared_xacts 显示当前已处于两阶段提交预备状态的事务的信息（详见 PREPARE TRANSACTION）。

pg_prepared_xacts 为每个预备事务包含一行。事务提交或回滚时，对应条目会被移除。

表 52.78. pg_prepared_xacts 列

名称	类型	引用	描述
transaction	xid		预备事务的数字事务 ID
gid	text		分配给该事务的全局事务标识符
prepared	timestamp with time zone		该事务进入预备提交状态的时间
owner	name	pg_authid.rolname	执行此事务的用户名
database	name	pg_database.datname	执行此事务所在数据库的名称

当访问 pg_prepared_xacts 视图时，内部事务管理器数据结构会被暂时锁定，并为视图制作一份副本用以显示。这确保视图生成一致的结果集，同时不会不必要地阻塞正常操作。尽管如此，如果频繁访问该视图，可能会对数据库性能产生一些影响。

52.78. pg_publication_tables

视图 pg_publication_tables 提供发布与其所包含表之间映射关系的信息。与底层目录 pg_publication_rel 不同，该视图会展开定义为 FOR ALL TABLES 的发布，因此对于这类发布，每个符合条件的表都会有一行。

表 52.79. pg_publication_tables 列

名称	类型	引用	描述
pubname	name	pg_publication.pubname	发布名称
schemaname	name	pg_namespace.nspname	包含该表的模式名称
tablename	name	pg_class.relname	表名称

52.79. pg_replication_origin_status

pg_replication_origin_status 视图包含关于某个源的重放进度信息。有关复制源的更多信息，请参见第 50 章。

表 52.80. pg_replication_origin_status 列

名称	类型	引用	描述
local_id	Oid	pg_replication_origin.roident	内部的节点标识符
external_id	text	pg_replication_origin.roname	外部的节点标识符
remote_lsn	pg_lsn		源节点的 LSN，到这个位置的数据都已经被复制。
local_lsn	pg_lsn		本节点上与已复制的 remote_lsn 对应的 LSN。在使用异步提交时，用于在将数据持久化到磁盘之前将提交记录刷盘。

52.80. pg_replication_slots

pg_replication_slots 视图提供当前数据库集簇中所有复制槽及其当前状态的列表。

有关复制槽的更多信息，请参见第 26.2.6 节和第 49 章。

表 52.81. pg_replication_slots 列

名称	类型	引用	描述
slot_name	name		一个集簇范围内唯一的复制槽标识符
plugin	name		包含该逻辑槽所用输出插件的共享对象的基础名称；物理槽为 NULL。

名称	类型	引用	描述
slot_type	text		槽类型--physical 或 logical
datoid	oid	pg_database.oid	与该槽关联的数据库 OID，或为 NULL。只有逻辑槽才有关联数据库。
database	text	pg_database.datname	与该槽关联的数据库名称，或为 NULL。只有逻辑槽才有关联数据库。
temporary	boolean		如果这是一个临时复制槽则为真。临时槽不会保存到磁盘，并会在出错时或会话结束时自动删除。
active	boolean		如果该槽当前正在使用中则为真
active_pid	integer		如果该槽当前正在使用中，则为使用该槽的会话的进程 ID；不活动时为 NULL。
xmin	xid		该槽要求数据库保留的最旧事务。VACUUM 不能移除由任何更晚事务删除的元组。
catalog_xmin	xid		该槽要求数据库保留的、影响系统目录的最旧事务。VACUUM 不能移除由任何更晚事务删除的目录元组。
restart_lsn	pg_lsn		此槽的消费者可能仍然需要的最早 WAL 地址（LSN），因此检查点期间不会自动移除这些 WAL。如果从未为此槽预留过 LSN，则为NULL。
confirmed_flush_lsn	pg_lsn		逻辑槽消费者已确认接收的数据所到达的地址（LSN）。早于此地址的数据将不再可用。物理槽为 NULL。

52.81. pg_roles

视图 pg_roles 提供数据库角色的信息。它本质上是 pg_authid 的公开可读视图，并隐藏密码字段。

此视图显式显示底层表的 OID 列，因为与其他目录连接时需要此列。

表 52.82. pg_roles 列

名称	类型	引用	描述
rolname	name		角色名
rolsuper	bool		角色有超级用户权限

名称	类型	引用	描述
rolinherit	bool		该角色是否会自动继承其所属角色的权限
rolcreaterole	bool		角色能创建更多角色
rolcreatedb	bool		角色能创建数据库
rolcanlogin	bool		角色是否可登录。也就是说，该角色能否被用作初始会话授权标识符
rolreplication	bool		角色是一个复制角色。复制角色可以启动复制连接并且创建和删除复制槽。
rolconnlimit	int4		对于可以登录的角色，本列设置该角色能够建立的最大并发连接数。-1 表示无限制。
rolpassword	text		不是密码（始终显示为 ********）
rolvaliduntil	timestamptz		密码过期时间（仅用于密码认证）；不设过期时间时为 NULL
rolbypassrls	bool		角色是否可以绕过所有的行级安全策略，详见第 5.7 节。
rolconfig	text[]		运行时配置变量的角色特定默认值
oid	oid	pg_authid.oid	角色的 ID

52.82. pg_rules

视图 pg_rules 提供了有关查询重写规则的有用信息。

表 52.83. pg_rules 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式名称
tablename	name	pg_class.relname	规则适用的表名
rulename	name	pg_rewrite.rulename	规则名
definition	text		规则定义（重建的创建命令）

视图 pg_rules 排除了视图和物化视图的 ON SELECT 规则；这些规则可以在 pg_views 和 pg_matviews 中看到。

52.83. pg_seclabels

视图 pg_seclabels 提供有关安全标签的信息。它是 pg_seclabel 目录的更易查询版本。

表 52.84. pg_seclabels 列

名称	类型	引用	描述
objoid	oid	任意 OID 列	该安全标签依附的对象的 OID
classoid	oid	pg_class.oid	该对象所在系统目录的 OID
objsubid	int4		对于一个在表列上的安全标签，这将是列号（objoid 和 classoid 指表本身）。对于所有其他对象类型，本列为 0。
objtype	text		此标签应用的对象类型，以文本方式。
objnamespace	oid	pg_namespace.oid	如果适用，为此对象的名字空间 OID；否则为 NULL。
objname	text		此标签应用的对象名，以文本形式。
provider	text	pg_seclabel.provider	与此标签相关的标签提供者。
label	text	pg_seclabel.label	应用于此对象的安全标签。

52.84. pg_sequences

视图 pg_sequences 提供对数据库中每个序列的有用信息的访问。

表 52.85. pg_sequences 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含该序列的模式名称
sequencename	name	pg_class.relname	序列名称
sequenceowner	name	pg_authid.rolname	序列拥有者名称
data_type	regtype	pg_type.oid	序列的数据类型
start_value	bigint		序列的起始值
min_value	bigint		序列的最小值
max_value	bigint		序列的最大值
increment_by	bigint		序列的增量值
cycle	boolean		该序列是否循环
cache_size	bigint		该序列的缓存大小
last_value	bigint		写入磁盘的最后一个序列值。如果使用缓存，该值可能大于该序列最后发出的值。如果尚未读取过此序列，此值为 NULL。如果当前用户没有该序列的 USAGE 或 SELECT 权限，此值也为 NULL。

52.85. pg_settings

视图 pg_settings 提供对服务器运行时参数的访问。它实质上是 SHOW 和 SET 命令的替代接口。它还提供了一些关于每个参数的事实，这些事实无法直接从SHOW 中获取，例如最小值和最大值。

表 52.86. pg_settings 列

名称	类型	描述
name	text	运行时配置参数名
setting	text	参数的当前值
unit	text	参数的隐含单位
category	text	参数的逻辑组
short_desc	text	参数的简短描述
extra_desc	text	附加的参数的详细描述
context	text	设置参数值所需的上下文（见下文）
vartype	text	参数类型（bool, enum, integer, real, 或 string）
source	text	当前参数值的来源
min_val	text	参数的最小允许值（对非数值参数为 NULL）
max_val	text	参数的最大允许值（对非数值参数为 NULL）
enumvals	text[]	枚举参数的允许值（对非枚举参数为 NULL）
boot_val	text	如果该参数未以其他方式设置，则为服务器启动时假定的参数值
reset_val	text	RESET 在当前会话中会将此参数重置到的值
sourcefile	text	设置当前值的配置文件（对于来自配置文件以外来源的值，或者由既不是超级用户也不是 pg_read_all_settings 成员的用户查看时，为 NULL）；在配置文件中使用 include 指令时这很有帮助
sourceline	integer	设置当前值的配置文件行号（对于来自配置文件以外来源的值，或者查看者既非超级用户、也不是 pg_read_all_settings 成员时，为 NULL）。
pending_restart	boolean	如果配置文件中修改了该值但需要重启，则为true，否则为false

有几种可能的 context 值。按照更改设置的难度递减的顺序，它们是：

internal

这些设置不能直接更改；它们反映了内部确定的值。其中一些可能可通过使用不同的配置选项重新构建服务器，或通过更改提供给initdb 的选项来进行调整。

postmaster

这些设置只能在服务器启动时应用，因此任何更改都需要重启服务器。这些设置的值通常存储在 postgresql.conf 文件中，或者在服务器启动时通过命令行传递。当然，下文列出的任何更易修改的 context 类型的设置也都可以在服务器启动时设置。

sighup

可以在 postgresql.conf 中更改这些设置，而无需重启服务器。向 postmaster 发送 SIGHUP 信号，会使其重新读取 postgresql.conf 并应用这些更改。postmaster 还会将 SIGHUP 信号转发给其子进程，以便它们都采用新值。

superuser-backend

可以在 postgresql.conf 中更改这些设置，而无需重启服务器。也可以在连接请求报文中为特定会话设置这些值（例如，通过 libpq 的 PGOPTIONS 环境变量），但前提是连接用户是超级用户。然而，这些设置在会话启动后永远不会更改。如果在 postgresql.conf 中更改它们，请向 postmaster 发送 SIGHUP 信号，使其重新读取 postgresql.conf。新值只会影响随后启动的会话。

backend

可以在 postgresql.conf 中更改这些设置，而无需重启服务器。也可以在连接请求报文中为特定会话设置这些值（例如，通过 libpq 的 PGOPTIONS 环境变量）；任何用户都可以为其会话进行此类更改。但是，在会话启动后，这些设置永远不会更改。如果在 postgresql.conf 中更改它们，请向 postmaster 发送 SIGHUP 信号，使其重新读取 postgresql.conf。新值只会影响随后启动的会话。

superuser

这些设置可以从 postgresql.conf 中设置，或者通过SET 命令在会话中设置；但只有超级用户可以通过 SET 更改它们。如果没有使用 SET 建立会话本地值，那么在 postgresql.conf 中的更改也会影响到现有会话。

user

这些设置可以从 postgresql.conf 中设置，或通过SET 命令在会话中设置。任何用户都可以更改其会话本地值。如果没有使用 SET 建立会话本地值，那么在postgresql.conf 中的更改也会影响到现有会话。

有关更改这些参数的各种方法，详见第 19.1 节。

pg_settings 视图中的行可以更新，但不能插入或删除。对 pg_settings 的某一行执行 UPDATE，等同于对该命名参数执行SET 命令。更改只影响当前会话所用的值。如果在之后被中止的事务中发出了 UPDATE，那么事务回滚时 UPDATE 命令的效果也会消失。一旦外围事务提交，这些效果会持续到会话结束，除非被另一个 UPDATE 或 SET 覆盖。

52.86. pg_shadow

视图 pg_shadow 存在是为了向后兼容性：它模拟了在 PostgreSQL 8.1 版本之前存在的目录。它显示了在 pg_authid 中 rolcanlogin 为真的所有角色的属性。

这个名称源于该表不应向公众开放读取，因为它包含密码。pg_user 是 pg_shadow 的公开可读视图，并隐藏密码字段。

表 52.87. pg_shadow 列

名称	类型	引用	描述
usename	name	pg_authid.rolname	用户名
usesysid	oid	pg_authid.oid	用户的 ID
usecreatedb	bool		用户可创建数据库
usesuper	bool		用户为一个超级用户
userepl	bool		用户可以发起流复制，并使系统进入或退出备份模式。
usebypassrls	bool		用户能否绕过所有的行级安全策略，详见第 5.7 节。
passwd	text		密码（可能已加密）；如果未设置则为 NULL。关于加密密码的存储方式，详见pg_authid。
valuntil	abstime		密码过期时间（仅用于密码认证）
useconfig	text[]		运行时配置变量的会话默认值

52.87. pg_stats

视图 pg_stats 提供对存储在 pg_statistic 目录中信息的访问。此视图仅允许访问用户具有读取权限的表对应的 pg_statistic 行，因此可以安全地允许对此视图进行公共读取访问。

pg_stats 也旨在以比底层目录更易读的格式呈现信息– 但其模式必须在为 pg_statistic 定义新的槽类型时进行扩展。

表 52.88. pg_stats 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式名称
tablename	name	pg_class.relname	表的名称
attname	name	pg_attribute.attname	被此行描述的列名
inherited	bool		如果为真，则此行包括来自子表的值，而不仅仅是指定表中的值
null_frac	real		列项中为 NULL 的比例
avg_width	integer		列的条目的平均字节宽度

名称	类型	引用	描述
n_distinct	real		如果大于零，表示列中不同值的估计个数。如果小于零，是不同值个数除以行数的负值（当ANALYZE 认为不同值的数量会随着表增长而增加时采用负值的形式，而如果认为列具有固定数量的可能值时采用正值的形式）。例如，-1 表示一个唯一列，即其中不同值的个数等于行数。
most_common_vals	anyarray		列中高频值的一个列表（如果没有任何一个值看起来比其他值更常用，此列为 NULL）
most_common_freqs	real[]		高频值的频率列表，即每一个高频值的出现次数除以总行数（如果 most_common_vals 为 NULL，则此列为 NULL）
histogram_bounds	anyarray		用于将列值划分为若干组的值列表，各组所含值的数量近似相等。如果存在 most_common_vals，其中的值会被直方图计算所忽略（如果列类型没有一个<操作符或者most_common_vals 列表覆盖了全部值，则此列为 NULL）
correlation	real		物理行顺序和列值逻辑顺序之间的统计相关性。其范围从-1 到+1。当值接近-1 或+1 时，在列上的一个索引扫描被估计为比值接近 0 时的代价更低，因为这种情况减少了对磁盘的随机访问（如果列数据类型不具有一个<操作符，则此列为 NULL）
most_common_elems	anyarray		列值内最常出现的非 NULL 元素值的列表（对于标量类型为 NULL）。
most_common_elem_freqs	real[]		高频元素值的频率列表，即至少包含一次给定值的行所占的比例。所有逐元素频率之后还有两个或三个附加值：前述逐元素频率的最小值和最大值，以及可选的 NULL 元素频率（当 most_common_elems 为 NULL 时，此列为 NULL）。

名称	类型	引用	描述
elem_count_histogram	real[]		各列值中不同非 NULL 元素的数量的直方图，后面跟随该数量的平均值（对于标量类型为 NULL）。

数组字段中条目的最大数量可以使用 ALTER TABLE SET STATISTICS 命令逐列控制，或者通过设置default_statistics_target 运行时参数来全局控制。

52.88. pg_tables

视图 pg_tables 提供了对数据库中每个表的有用信息的访问。

表 52.89. pg_tables 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式名称
tablename	name	pg_class.relname	表的名称
tableowner	name	pg_authid.rolname	表拥有者名称
tablespace	name	pg_tablespace.spcname	包含该表的表空间名称（如果使用数据库的默认表空间，此列为 NULL）
hasindexes	boolean	pg_class.relhasindex	如果表有（或最近有过）任何索引，此列为真
hasrules	boolean	pg_class.relhasrules	如果表有（或曾经有过）规则，此列为真
hastriggers	boolean	pg_class.relhastriggers	如果表有（或者曾经有过）触发器，此列为真
rowsecurity	boolean	pg_class.relrowsecurity	如果表上启用了行级安全则为真

52.89. pg_timezone_abbrevs

视图 pg_timezone_abbrevs 提供当前被日期时间输入例程识别的时区缩写列表。当修改 timezone_abbreviations 运行时参数时，该视图的内容会发生变化。

表 52.90. pg_timezone_abbrevs 列

名称	类型	描述
abbrev	text	时区缩写
utc_offset	interval	相对于 UTC 的偏移（正值表示格林威治以东）
is_dst	boolean	如果这是一个夏令时缩写，则为真

虽然大多数时区缩写代表与 UTC 的固定偏移量，但也有一些在历史上变化过（有关更多信息，请参见第 B.4 节）。在这种情况下，此视图呈现它们当前的含义。

52.90. pg_timezone_names

视图 pg_timezone_names 提供了一个由 SET TIMEZONE 识别的时区名称列表，以及它们的相关缩写、UTC 偏移和夏令时状态。（从技术上讲，PostgreSQL 不使用 UTC，因为不处理闰秒。）与在pg_timezone_abbrevs 中显示的缩写不同，这些名称中的许多隐含一组夏令时转换日期规则。因此，相关信息在跨越本地夏令时切换边界时发生变化。显示的信息是基于 CURRENT_TIMESTAMP 的当前值计算的。

表 52.91. pg_timezone_names 列

名称	类型	描述
name	text	时区名
abbrev	text	时区缩写
utc_offset	interval	相对于 UTC 的偏移（正值表示格林威治以东）
is_dst	boolean	如果当前采用夏令时，则为真

52.91. pg_user

视图 pg_user 提供数据库用户的信息。它本质上是 pg_shadow 的公开可读视图，并隐藏密码字段。

表 52.92. pg_user 列

名称	类型	描述
usename	name	用户名
usesysid	oid	用户的 ID
usecreatedb	bool	用户可创建数据库
usesuper	bool	用户为一个超级用户
userepl	bool	用户可以发起流复制，并使系统进入或退出备份模式。
usebypassrls	bool	用户能否绕过所有的行级安全策略，详见第 5.7 节。
passwd	text	不是密码（始终显示为 ********）
valuntil	abstime	密码过期时间（仅用于密码认证）
useconfig	text[]	运行时配置变量的会话默认值

52.92. pg_user_mappings

视图 pg_user_mappings 提供用户映射的信息。它本质上是 pg_user_mapping 的公开可读视图，但如果用户没有使用该用户映射的权限，则会省略选项字段。

表 52.93. pg_user_mappings 列

名称	类型	引用	描述
umid	oid	pg_user_mapping.oid	用户映射的 OID

名称	类型	引用	描述
srvid	oid	pg_foreign_server.oid	包含该映射的外部服务器的 OID
srvname	name	pg_foreign_server.srvname	外部服务器名
umuser	oid	pg_authid.oid	将要被映射的本地角色的 OID，如果用户映射是公共的则为零
usename	name		将被映射的本地用户名
umoptions	text[]		用户映射特定的选项，以 “keyword=value” 字符串形式

为了保护作为用户映射选项存储的密码信息，umoptions 列会显示为空值，除非符合以下条件之一：

- 当前用户是被映射的用户，并拥有服务器或在其上拥有USAGE 权限
- 当前用户是服务器所有者，映射是为PUBLIC 而进行的
- 当前用户是超级用户

52.93. pg_views

视图 pg_views 提供了对数据库中每个视图的有用信息的访问。

表 52.94. pg_views 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含视图的模式名
viewname	name	pg_class.relname	视图名称
viewowner	name	pg_authid.rolname	视图拥有者名称
definition	text		视图定义（一个重建的 SELECT 查询）

第 53 章 前端/后端协议

PostgreSQL 使用一种基于消息的协议在前端和后端（客户端与服务器）之间进行通信。该协议既支持 TCP/IP，也支持 Unix 域套接字。端口号 5432 已在 IANA 注册为支持该协议的服务器的惯用 TCP 端口号，但实际上任何非特权端口号都可以使用。

本文描述协议 3.0 版本，自 PostgreSQL 7.4 起实现。关于更早协议版本的说明，请参阅之前发布的 PostgreSQL 文档。一台服务器可以支持多个协议版本。初始启动请求消息会告知服务器客户端正尝试使用哪个协议版本。如果客户端请求的主版本号不被服务器支持，则连接会被拒绝（例如，如果客户端请求协议版本 4.0，而在本文编写时该版本并不存在，就会出现这种情况）。如果客户端请求的次版本号不被服务器支持（例如客户端请求 3.1，而服务器只支持 3.0），服务器可以拒绝连接，也可以返回一条 NegotiateProtocolVersion 消息，其中包含它所支持的最高次协议版本。客户端随后可以选择使用指定的协议版本继续连接，或者中止连接。

为了高效地为多个客户端提供服务，服务器会为每个客户端启动一个新的“后端”进程。在当前实现中，一旦检测到传入连接，就会立刻创建新的子进程。不过，这一点对协议而言是透明的。就协议而言，术语“后端”和“服务器”可以互换；同样，“前端”和“客户端”也可以互换。

53.1. 概述

协议分为启动和正常操作两个阶段。在启动阶段，前端打开到服务器的连接，并完成服务器所要求的认证。（这可能只涉及一条消息，也可能因所用认证方法不同而需要多条消息。）如果一切顺利，服务器随后会向前端发送状态信息，并最终进入正常操作。除最初的启动请求消息外，协议的这一部分由服务器驱动。

在正常操作中，前端向后端发送查询及其他命令，后端则返回查询结果和其他响应。少数情况下（例如 `NOTIFY`），后端会发送未请求的消息，但会话中的绝大多数交互仍由前端请求驱动。

会话通常由前端选择终止，但在某些情况下也可能由后端强制终止。无论哪种情况，后端关闭连接时，都会在退出前回滚所有打开的（未完成的）事务。

在正常操作中，SQL 命令可以通过两种子协议之一执行。在“简单查询”协议中，前端只需发送文本形式的查询字符串，后端会立即对其进行解析并执行。在“扩展查询”协议中，查询处理被拆分为多个步骤：解析、参数值绑定以及执行。这带来了更高的灵活性和性能收益，但代价是额外的复杂性。

正常操作还包含用于 `COPY` 等特殊操作的额外子协议。

53.1.1. 消息概述

所有通信都通过消息流进行。消息的第一个字节标识消息类型，接下来的四个字节给出消息其余部分的长度（该长度计数包含自身，但不包括消息类型字节）。消息剩余内容由消息类型决定。由于历史原因，客户端发送的第一条消息（启动消息）没有开头的消息类型字节。

为了避免与消息流失去同步，服务器和客户端通常都会先根据字节计数把整条消息读入缓冲区，然后再处理其内容。这样一来，如果在处理内容时检测到错误，就比较容易恢复。在极端情况下（例如没有足够内存缓冲整条消息），接收方也可以利用字节计数判断在恢复读取消息之前需要跳过多少输入。

反过来，服务器和客户端也必须注意绝不能发送不完整的消息。通常的做法是在开始发送之前，先在缓冲区中整理好整条消息。如果在发送或接收消息的途中发生通信故障，唯一合理的做法就是放弃连接，因为几乎不可能重新恢复消息边界的同步。

53.1.2. 扩展查询概述

在扩展查询协议中，SQL 命令的执行被拆分为多个步骤。各步骤之间保留的状态由两类对象表示：预备语句和 portal。预备语句表示对文本查询字符串完成解析和语义分析后的结果。预备语句本身还不能直接执行，因为它可能缺少特定的参数值。portal 表示一条已经可以执行、或已经部分执行过的语句，其中所有缺失的参数值都已补齐。（对于`SELECT` 语句，portal 等价于一个打开的游标；但由于游标不能处理非`SELECT` 语句，这里采用不同术语。）

整个执行周期包括一个解析步骤，它从文本查询字符串创建预备语句；一个绑定步骤，它根据预备语句和所需参数值创建 portal；以及一个执行步骤，用于执行 portal 中的查询。对于返回行的查询（`SELECT`、`SHOW` 等），可以要求执行步骤只取回有限数量的行，因此可能需要多次执行步骤才能完成整个操作。

后端可以跟踪多个预备语句和 portal（但请注意，它们只存在于单个会话内，绝不会在会话之间共享）。已有的预备语句和 portal 都通过创建时赋予的名称来引用。此外，还存在一个“未命名”的预备语句和 portal。虽然它们的行为与有名对象大体相同，但对未命名对象的操作是为“只执行一次然后丢弃”的场景优化的，而对有名对象的操作则是基于会被多次使用的预期进行优化的。

53.1.3. 格式和格式代码

某一特定数据类型的数据可以使用多种不同的格式之一进行传输。自 PostgreSQL 7.4 起，当前只支持“文本”和“二进制”两种格式，但协议为未来扩展留出了空间。任意值所需的格式由格式代码指定。客户端可以为每个传输的参数值以及查询结果的每一列指定格式代码。文本格式的代码为零，二进制格式的代码为一，其他格式代码则保留供将来定义。

值的文本表示是相应数据类型的输入/输出转换函数生成和接受的字符串。在传输形式中，值的末尾没有空字符；前端若要将收到的值作为 C 字符串处理，必须自行添加一个。（文本格式也不允许内嵌空字符。）

整数的二进制表示采用网络字节序（最高有效字节在前）。至于其他数据类型，请查阅文档或源代码了解其二进制表示形式。要注意，复杂数据类型的二进制表示可能会在不同服务器版本之间发生变化；文本格式通常是可移植性更好的选择。

53.2. 消息流

本节描述消息流以及各种消息类型的语义（每种消息的精确格式见第 53.7 节）。根据连接所处的状态不同，存在若干不同的子协议：启动、查询、函数调用、`COPY` 以及终止。对于异步操作（包括通知响应和命令取消）还有专门规定，它们可能在启动阶段结束后的任何时刻发生。

53.2.1. 启动

要开始一个会话，前端会打开到服务器的连接并发送一条启动消息。该消息包含用户名以及用户希望连接的数据库名；它还指明要使用的协议版本。（启动消息也可以选择性地包含运行时参数的附加设置。）随后服务器会结合这些信息以及配置文件（例如 `pg_hba.conf`）的内容，初步判断是否接受该连接，以及需要何种额外认证（如果需要）。

接着服务器会发送适当的认证请求消息，前端必须以适当的认证响应消息进行回应（例如密码）。对于除 GSSAPI、SSPI 和 SASL 之外的认证方法，最多只会有一次请求和一次响应。在某些方法中，前端根本不需要发送响应，因此也不会出现认证请求。对于 GSSAPI、SSPI 和 SASL，则可能需要多轮报文交换才能完成认证。

认证周期要么以服务器拒绝连接（ErrorResponse）结束，要么以 AuthenticationOk 结束。

服务器在此阶段可能发送的消息如下：

ErrorResponse

连接请求被拒绝。然后服务器马上关闭连接。

AuthenticationOk

认证交换成功完成。

AuthenticationKerberosV5

前端现在必须参与与服务器之间的 Kerberos V5 认证对话（此处不再描述，它属于 Kerberos 规范的一部分）。如果对话成功，服务器会响应 AuthenticationOk；否则响应 ErrorResponse。该机制已不再受支持。

AuthenticationCleartextPassword

前端现在必须发送一个以明文形式包含密码的 PasswordMessage。如果密码正确，服务器响应 AuthenticationOk；否则响应 ErrorResponse。

AuthenticationMD5Password

前端现在必须发送一个 PasswordMessage，其中包含密码；该密码先与用户名一起经过 MD5 加密，再使用 AuthenticationMD5Password 消息中指定的 4 字节随机盐重新加密。如果密码正确，服务器响应 AuthenticationOk；否则响应 ErrorResponse。实际的 PasswordMessage 可以用 SQL 计算：`concat('md5', md5(concat(md5(concat(password, username)), random-salt)))`。（请记住 `md5()` 函数返回的是十六进制字符串。）

AuthenticationSCMCredential

此响应仅适用于支持 SCM 凭据消息的平台上的本地 Unix 域连接。前端必须发出一条 SCM 凭据消息，然后发送一个单字节数据。（该数据字节的内容无关紧要；它仅用于确保服务器等待足够长的时间来接收凭据消息。）如果凭据可接受，服务器响应 AuthenticationOk；否则响应 ErrorResponse。（该消息类型仅由 9.1 之前的服务器发出。它最终可能会从协议规范中移除。）

AuthenticationGSS

前端现在必须发起一次 GSSAPI 协商。前端将发送一条带有 GSSAPI 数据流第一部分的 GSSResponse 消息作为响应。如果还需要进一步的消息，服务器会响应 Authentication GSSContinue。

AuthenticationSSPI

前端现在必须发起一次 SSPI 协商。前端将发送一条带有 SSPI 数据流第一部分的 GSSResponse 作为响应。如果还需要进一步的消息，服务器会响应 Authentication GSSContinue。

AuthenticationGSSContinue

该消息包含前一步 GSSAPI 或 SSPI 协商（AuthenticationGSS、AuthenticationSSPI 或前一个 AuthenticationGSSContinue）的响应数据。如果该消息中的 GSSAPI 或 SSPI 数据表明完成认证还需要更多数据，前端必须把所需数据作为另一条 GSSResponse 发送。如果该消

息已完成 GSSAPI 或 SSPI 认证，服务器接下来会发送 AuthenticationOk 表示认证成功，或发送 ErrorResponse 表示认证失败。

AuthenticationSASL

前端现在必须发起一次 SASL 协商，使用该消息中列出的一种 SASL 机制。前端将发送一条 SASLInitialResponse，其中包含所选机制的名称以及 SASL 数据流的第一部分。如果还需要进一步的消息，服务器会响应 AuthenticationSASLContinue。详见第 53.3 节。

AuthenticationSASLContinue

该消息包含 SASL 协商前一步（AuthenticationSASL 或前一个 AuthenticationSASLContinue）的挑战数据。前端必须用一条 SASLResponse 消息响应。

AuthenticationSASLFinal

SASL 认证已完成，并附带了面向客户端的额外机制相关数据。服务器接下来会发送 AuthenticationOk 表示认证成功，或发送 ErrorResponse 表示认证失败。只有当 SASL 机制规定在完成时要从服务器向客户端发送附加数据时，才会发送该消息。

NegotiateProtocolVersion

服务器不支持客户端请求的协议次版本，但支持更早的协议版本；此消息指明其所支持的最高次版本。如果客户端在启动包中请求了不受支持的协议选项（即以 `_pq_.` 开头的选项），也会发送此消息。此消息后面会跟随一条 ErrorResponse 或一条指示认证成功或失败的消息。

如果前端不支持服务器要求的认证方式，那么它应该马上关闭连接。

在收到 AuthenticationOk 消息之后，前端必须继续等待来自服务器的后续消息。在这个阶段，一个后端进程正在启动，而前端只是旁观者。启动尝试仍可能失败（ErrorResponse），服务器也可能拒绝支持所请求的协议次版本（NegotiateProtocolVersion）；但通常情况下，后端会发送一些 ParameterStatus 消息、BackendKeyData，以及最后的 ReadyForQuery。

在这个阶段，后端会尝试应用启动消息中给出的任何额外运行时参数设置。如果成功，这些值就会成为会话的默认值。发生错误则会导致 ErrorResponse 并退出。

这个阶段来自后端的可能消息是：

BackendKeyData

该消息提供密钥数据。如果前端希望稍后发送取消请求，就必须保存这些数据。前端不应响应该消息，而应继续等待 ReadyForQuery 消息。

ParameterStatus

该消息告知前端后端参数的当前（初始）设置，例如client_encoding 或 DateStyle。前端可以忽略这些信息，也可以记录下来供后续使用；详见第 53.2.7 节。前端不应响应该消息，而应继续等待 ReadyForQuery 消息。

ReadyForQuery

启动成功，前端现在可以发出命令。

ErrorResponse

启动失败，在发送完这个消息之后连接被关闭。

NoticeResponse

已发出一条警告消息。前端应显示该消息，但仍应继续等待 ReadyForQuery 或 Error Response。

ReadyForQuery 消息与后端在每个命令周期结束后发出的消息是同一个。前端可根据自身编码需要，将 ReadyForQuery 视为一个命令周期的开始，或视为启动阶段以及后续每个命令周期的结束。

53.2.2. 简单查询

一个简单查询周期由前端向后端发送一条 Query 消息来启动。该消息包含一条或多条以文本字符串表示的 SQL 命令。后端随后会根据查询命令串的内容向前端发送一条或多条响应消息，最后再发送一条 ReadyForQuery 响应消息。ReadyForQuery 通知前端它现在可以安全地发送新的命令。（实际上，前端并不一定非要等到收到 ReadyForQuery 才发送下一条命令，但这样一来，前端就必须自己处理“先前命令失败而后续已发出的命令成功”这种情况。）

后端可能返回的响应消息有：

CommandComplete

一条 SQL 命令正常完成。

CopyInResponse

后端已准备好把数据从前端复制到表中；参见第 53.2.6 节。

CopyOutResponse

后端已准备好把数据从表中复制到前端；参见第 53.2.6 节。

RowDescription

表示即将返回行作为对 `SELECT`、`FETCH` 等查询的响应。此消息的内容描述了行的列布局。该消息之后，每个返回给前端的行都对应一条 DataRow 消息。

DataRow

由 `SELECT`、`FETCH` 等查询返回的一组行中的一个。

EmptyQueryResponse

识别出了一条空查询字符串。

ErrorResponse

发生了一个错误。

ReadyForQuery

查询字符串的处理已完成。发送一个单独的消息来指示这一点，因为查询字符串可能包含多个 SQL 命令。（CommandComplete 标记了一个 SQL 命令的处理结束，而不是整个字符串的结束。）无论处理是成功还是出现错误，都将始终发送 ReadyForQuery。

NoticeResponse

与查询相关的警告消息已发出。通知是其他响应的补充，即后端将继续处理命令。

SELECT 查询（或其他返回行集的查询，如 EXPLAIN 或 SHOW）的响应通常包含 Row Description、零条或多条 DataRow 消息，以及最后的 CommandComplete。在前端与服务器之间执行 COPY 输入或输出时，会使用第 53.2.6 节所述的特殊协议。所有其他类型的查询通常只产生一条 CommandComplete 消息。

由于查询字符串可能包含若干条查询（以分号分隔），因此在后端完成整个查询字符串的处理之前，可能会出现多个这样的响应序列。只有在整个字符串处理完毕且后端已准备好接受新的查询字符串时，才会发出 ReadyForQuery 消息。

如果收到的是一条完全为空的查询字符串（除了空白字符之外没有任何内容），响应就是一条 EmptyQueryResponse，后面跟着 ReadyForQuery。

一旦发生错误，就会发出一条 ErrorResponse 消息，后面跟着 ReadyForQuery。Error Response 会中止该查询字符串中后续所有处理（即使其中还包含其他查询）。请注意，这种情况可能发生在处理单条查询所生成的消息序列中途。

在简单查询模式中，取回值的格式总是文本，除非给出的命令是在使用 BINARY 选项声明的游标上执行 FETCH。在这种情况下，取回的值将采用二进制格式。RowDescription 消息中给出的格式代码会告诉我们使用的是哪种格式。

当前端正在等待其他类型的消息时，也必须准备好接收 ErrorResponse 和 NoticeResponse 消息。参见第 53.2.7 节，了解后端因外部事件而可能生成的消息。

建议以状态机的方式编写前端，使其能够在任何合理的时机接收相应类型的消息，而不把消息确切顺序的假设写死在代码中。

53.2.2.1. 一个简单查询中的多条语句

当一个简单查询消息中包含多于一条 SQL 语句（被分号分隔）时，这些语句会作为单个事务执行，除非其中包括显式事务控制命令来强制不同的行为。例如，如果消息包括

```
INSERT INTO mytable VALUES(1);
SELECT 1/0;
INSERT INTO mytable VALUES(2);
```

则 SELECT 中的除零失败将强制回滚第一个 INSERT。进而，因为该消息的执行在第一个错误时就被放弃，第二个INSERT 根本都不会被尝试。

如果该消息包含的是

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELECT 1/0;
```

那么第一个 INSERT 会被这个显式的 COMMIT 命令提交。第二个 INSERT 以及 SELECT 仍会被当作一个单一事务，这样除零失败将回滚第二个INSERT，但不会回滚第一个。

这种行为的实现方式是：除非已有显式事务块可供这些语句执行，否则就在一个隐式事务块中执行多语句 Query 消息中的语句。隐式事务块与常规事务块的主要区别在于，隐式块会在 Query 消息结束时自动关闭：没有错误时隐式提交，发生错误时隐式回滚。这类似于单独执行一条语句（不在事务块中）时发生的隐式提交或回滚。

如果会话因前面某条消息中的 BEGIN 而已经处于事务块中，那么 Query 消息会继续该事务块，不管该消息包含一个语句还是多个语句。不过，如果该 Query 消息包含一个关闭现有事务块的 COMMIT 或者 ROLLBACK，那么任何接下来的语句都会在一个隐式事务块中被执行。反过来，如果在多语句 Query 消息中出现一个BEGIN，那么它会开始一个常规事务块，这个常规事务块将只能被一个显式的COMMIT 或者 ROLLBACK 终止，不管这两种命令是出现在这个 Query 消息还是后面的一个 Query 消息中。如果BEGIN 跟在一些作为隐式事务块执行的语句后面，那些语句不会被立刻提交。实际上，它们会被包括到新的常规事务块中。

出现在一个隐式事务块中的 COMMIT 或者 ROLLBACK 会被正常执行并且关闭该隐式块。不过，由于没有先前的BEGIN 与之配对的 COMMIT 或 ROLLBACK 可能表示误操作，所以会发出警告。如果后面还有更多语句，将会为它们开始一个新的隐式事务块。

在隐式事务块中不允许保存点，因为它们会与发生错误时自动关闭块的行为发生冲突。

请记住，无论消息中是否包含事务控制命令，Query 消息的执行都会在第一个错误处停止。例如，单条 Query 消息包含以下内容时：

```
BEGIN;
SELECT 1/0;
ROLLBACK;
```

会话会停留在失败的常规事务块内，因为发生除零错误后不会再执行ROLLBACK。需要另发一条 ROLLBACK，才能将会话恢复到可用状态。

另一种要注意的行为是，在执行任何语句之前，会先对整个查询字符串进行词法和语法分析。因此后面的语句中的简单错误（例如拼写错误的关键词）可能会阻止任何语句的执行。这通常对用户是不可见的，因为在当作一个隐式事务块执行时，这些语句不管怎样都会全部被回滚。不过，在尝试于一个多语句 Query 中执行多个事务时，这种现象可能是可见的。例如，如果一个拼写错误把我们之前的示例变成

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELCT 1/0;
```

那么这些语句都不会被运行，导致可见的差别，即第一个INSERT 没有被提交。在语义分析及其后阶段检测到的错误（例如拼错的表名或者列名）不会有这种效果。

53.2.3. 扩展查询

扩展查询协议把上文描述的简单查询协议拆分成多个步骤。准备步骤的结果可以重复使用，从而提高效率。此外，它还提供了额外特性，例如可以把数据值作为独立参数提供，而不必直接插入查询字符串中。

在扩展协议中，前端首先发送一条 Parse 消息，其中包含文本查询字符串、可选的参数占位符数据类型信息，以及目标预备语句对象的名称（空字符串表示未命名预备语句）。响应要么是 ParseComplete，要么是 ErrorResponse。参数数据类型可以用 OID 指定；如果未给出，解析器会像处理无类型字面字符串常量那样尝试推断其数据类型。

> **注意**
>
> 一个参数的数据类型可以通过设为零，或让参数类型 OID 数组短于查询字符串中参数符号（`$n`）的数量来保持未指定。另一个特例是，参数类型可以指定为 `void`（即伪类型 `void` 的 OID）。这样做是为了允许参数符号用于那些实际上是 OUT 参数的函数参数。通常并不存在可使用 `void` 参数的上下文，但如果这样的参数符号出现在函数参数列表中，它实际上会被忽略。例如，像 `foo($1,$2,$3,$4)` 这样的函数调用，如果 `$3` 和 `$4` 被指定为 `void` 类型，就可能匹配一个带有两个 IN 参数和两个 OUT 参数的函数。

> **注意**
>
> Parse 消息中的查询字符串不能包含多于一条 SQL 语句；否则会报告语法错误。这个限制在简单查询协议中并不存在，但在扩展协议中必须如此，因为若允许预备语句或 portal 包含多条命令，会使协议变得过于复杂。

如果成功创建了一个有名的预备语句对象，它会一直持续到当前会话结束，除非被显式销毁。未命名预备语句只会持续到下一条把未命名语句作为目标的 Parse 消息发出为止。（注意，简单 Query 消息也会销毁未命名语句。）有名预备语句在被另一条 Parse 消息重新定义之前必须显式关闭，但未命名语句则无此要求。有名预备语句也可以在 SQL 命令级别通过`PREPARE` 和 `EXECUTE` 来创建和访问。

一旦预备语句存在，就可以用 Bind 消息把它准备为可执行状态。Bind 消息给出源预备语句的名称（空字符串表示未命名预备语句）、目标 portal 的名称（空字符串表示未命名 portal），以及预备语句中所有参数占位符应使用的值。所提供的参数集必须与预备语句所需参数相匹配。（如果你在 Parse 消息中声明了任何 `void` 参数，那么在 Bind 消息中应为它们传递 NULL 值。）Bind 还会指定查询返回数据所使用的格式；格式既可以统一指定，也可以按列指定。响应要么是 BindComplete，要么是 ErrorResponse。

> **注意**
>
> 输出采用文本还是二进制格式，由 Bind 中给出的格式代码决定，而与涉及的 SQL 命令无关。在使用扩展查询协议时，游标声明中的 `BINARY` 属性并不起作用。

通常会在处理 Bind 消息时进行查询规划。如果预备语句没有参数，或者会被重复执行，服务器可能会保存生成的计划，并在后续针对同一预备语句的 Bind 消息中重用它。不过，只有当它发现可以创建一个效率并不比依赖具体参数值的计划差很多的通用计划时，才会这样做。就协议而言，这一切都是透明的。

如果成功创建了一个有名 portal 对象，它会一直持续到当前事务结束，除非被显式销毁。未命名 portal 会在事务结束时销毁，或者在下一条把未命名 portal 作为目标的 Bind 消息发出时立即销毁。（注意，简单 Query 消息也会销毁未命名 portal。）有名 portal 在被另一条 Bind 消息重新定义之前必须显式关闭，而未命名 portal 则无此要求。有名 portal 也可以在 SQL 命令级别通过`DECLARE CURSOR` 和 `FETCH` 来创建和访问。

一旦 portal 存在，就可以使用 Execute 消息执行它。Execute 消息指定 portal 的名称（空字符串表示未命名 portal）以及一个最大的结果行计数（零表示“取回全部行”）。结果行计数只对包含返回行集命令的 portal 有意义；在其他情况下，命令总会执行到完成，而行计数会被忽

略。Execute 的可能响应与通过简单查询协议发出的查询相同，只是 Execute 不会导致后端发送 ReadyForQuery 或 RowDescription。

如果 Execute 在 portal 执行完成之前终止（因为达到了非零的结果行计数），它会发送一条 PortalSuspended 消息；该消息表明前端应当针对同一个 portal 再发出一条 Execute 消息，以完成此次操作。在 portal 执行完成之前，不会发送表示源 SQL 命令结束的 Command Complete 消息。因此，一个 Execute 阶段总会以下列消息中的恰好一条结束：Command Complete、EmptyQueryResponse（如果 portal 是从空查询字符串创建的）、ErrorResponse 或 PortalSuspended。

每一组扩展查询消息完成后，前端都应发送一条 Sync 消息。这条无参数消息会让后端关闭当前事务，如果当前事务并不处在 `BEGIN`/`COMMIT` 事务块内（这里的“关闭”指的是：无错误则提交，有错误则回滚）。随后会发送一条 ReadyForQuery 响应。Sync 的目的是为错误恢复提供一个重新同步点。如果在处理任何扩展查询消息时检测到错误，后端会发出 ErrorResponse，然后持续读取并丢弃消息，直到遇到 Sync，再发送 ReadyForQuery 并回到正常的消息处理流程。（但要注意，如果是在处理 Sync 期间检测到错误，则不会跳过任何消息，这样就能保证每条 Sync 恰好对应一条 ReadyForQuery。）

> **注意**
>
> Sync 不会关闭由 `BEGIN` 打开的事务块。之所以能够检测出这种情况，是因为 ReadyForQuery 消息中包含事务状态信息。

除了这些基本的、必需的操作之外，在扩展查询协议里还有几种可选的操作可以使用。

Describe 消息（portal 变体）指定一个现有 portal 的名称（或者用空字符串表示未命名 portal）。响应要么是一条 RowDescription 消息，用于描述执行该 portal 时将返回的行；要么是一条 NoData 消息，如果该 portal 不包含会返回行的查询；要么是 ErrorResponse，如果不存在这样的 portal。

Describe 消息（语句变体）指定一个现有预备语句的名称（或者用空字符串表示未命名预备语句）。响应是一条 ParameterDescription 消息，用于描述该语句所需的参数，随后是一条 RowDescription 消息，用于描述该语句最终执行时将返回的行（如果该语句不返回行，则为 NoData 消息）。如果不存在这样的预备语句，则返回 ErrorResponse。请注意，由于还没有发出 Bind，后端尚不知道返回列将使用什么格式；因此在这种情况下，RowDescription 消息中的格式代码字段将为零。

> **提示**
>
> 在大多数场景下，前端都应在发出 Execute 之前先发送某一种 Describe 变体，以确保它知道应如何解释接收到的结果。

Close 消息会关闭一个现有的预备语句或 portal，并释放相关资源。针对不存在的语句或 portal 发出 Close 并不算错误。响应通常是 CloseComplete，但如果在释放资源时遇到困难，也可能返回 ErrorResponse。请注意，关闭预备语句会隐式关闭所有基于该语句构造出的打开 portal。

Flush 消息本身不会产生任何特定输出，但会强制后端发送其输出缓冲区中所有尚待发送的数据。如果前端希望在发出更多命令之前检查某条扩展查询命令的结果，那么除了 Sync 之外，任何扩展查询命令之后都必须发送 Flush。如果没有 Flush，后端返回的消息会尽量合并成最少的数据包，以降低网络开销。

> **注意**
>
> 简单 Query 消息大致等价于一串 Parse、Bind、portal Describe、Execute、Close、Sync 操作，它们使用未命名预备语句和未命名 portal 对象，并且不带参数。不同之处在于，简单 Query 会接受包含多条 SQL 语句的查询字符串，并依次自动为每条语句执行绑定、描述和执行序列；另一个区别是它不会返回 ParseComplete、BindComplete、CloseComplete 或 NoData 消息。

53.2.4. 管道化

扩展查询协议允许管道化，即连续发送一系列查询，而不等待先前查询完成。这减少了完成一组操作所需的网络往返次数。不过，用户必须仔细考虑其中某个步骤失败时所需的行为，因为后续查询已经在发往服务器的途中。

处理这个问题的一种方式是让整组查询成为单个事务，即用`BEGIN` ... `COMMIT` 包围它们。不过，如果希望其中某些命令独立于其他命令提交，这种方式就无法满足要求。

扩展查询协议提供了另一种处理方式，即省略相互依赖的步骤之间的 Sync 消息。发生错误后，后端会跳过命令消息，直到遇到 Sync，因此当前面的命令失败时，管道中的后续命令便会自动跳过，而不必由客户端用`BEGIN` 和 `COMMIT` 显式管理。管道中可独立提交的各段可以用 Sync 消息分隔。

如果客户端没有发出显式的 `BEGIN`，那么每个 Sync 通常会在前序步骤成功时引发隐式的 `COMMIT`，失败时引发隐式的`ROLLBACK`。不过，少数 DDL 命令（例如`CREATE DATABASE`）不能在事务块中执行。如果这类命令在管道中执行，除非它是管道中的第一条命令，否则会失败。此外，它成功后会强制立即提交，以保持数据库一致性。因此，紧随这类命令之后的 Sync 除了返回 ReadyForQuery 外，不产生其他效果。

使用这种方式时，必须统计 ReadyForQuery 消息的数量，并等待该数量达到已发送的 Sync 消息数量，才能确定管道已完成。统计命令完成响应并不可靠，因为某些命令可能被跳过，因而不会产生完成消息。

53.2.5. 函数调用

函数调用子协议允许客户端请求直接调用数据库 `pg_proc` 系统目录中的任意函数。客户端必须具有该函数的执行权限。

> **注意**
>
> 函数调用子协议是一个遗留的特性，在新代码里可能最好避免用它。类似的结果可以通过设置一个执行`SELECT function($1, ...)` 的预备语句得到。这样函数调用周期就可以用 Bind/Execute 代替。

函数调用周期由前端向后端发送一条 FunctionCall 消息来启动。后端随后根据函数调用的结果发送一条或多条响应消息，最后发送一条 ReadyForQuery 响应消息。ReadyForQuery 告知前端，可以安全地发送新的查询或函数调用。

来自后端的可能的响应消息是：

ErrorResponse

发生了一个错误。

FunctionCallResponse

函数调用完成并且在消息中返回一个结果（请注意函数调用协议只能处理单个标量结果，不能处理行类型或者结果集合）。

ReadyForQuery

函数调用处理完成。ReadyForQuery 将总是被发送，不管是成功完成处理还是发生一个错误。

NoticeResponse

发出了一条有关该函数调用的警告信息。通知是附加在其他响应上的，也就是说，后端将继续处理该命令。

53.2.6. COPY 操作

`COPY` 命令允许在服务器和客户端之间进行高速大批量数据传输。拷贝入和拷贝出操作每个都把连接切换到一个独立的子协议中，并且持续到操作结束。

拷贝入模式（向服务器传输数据）在后端执行 `COPY FROM STDIN` SQL 语句时启动。后端会向前端发送一条 CopyInResponse 消息。随后前端应发送零条或多条 CopyData 消息，构成一条输入数据流。（消息边界与行边界之间没有任何对应关系要求，尽管让它们对齐通常是合理的选择。）前端可以通过发送 CopyDone 消息来结束拷贝入模式（允许成功结束），也可以发送 CopyFail 消息（这会使`COPY` 语句以错误失败）。然后后端会恢复到 `COPY` 开始之前的命令处理模式，也就是简单查询协议或扩展查询协议。接下来它会发送 CommandComplete（成功时）或 ErrorResponse（失败时）。

如果在拷贝入模式期间后端检测到错误（包括收到 CopyFail 消息），后端会发出一条 Error Response 消息。如果`COPY` 命令是通过扩展查询消息发出的，那么后端会从此开始丢弃前端消息，直到收到一条 Sync 消息，然后发出 ReadyForQuery 并恢复正常处理。如果`COPY` 命令是在简单 Query 消息中发出的，那么该消息的剩余部分会被丢弃，并发送 ReadyForQuery。无论哪种情况，前端随后发出的任何 CopyData、CopyDone 或 CopyFail 消息都会被直接丢弃。

后端会忽略在拷贝入模式期间收到的 Flush 和 Sync 消息。收到任何其他非拷贝类型的消息都会构成错误，并按上述方式中止拷贝入状态。（Flush 和 Sync 的例外是为了方便那些总是在 Execute 消息之后发送 Flush 或 Sync、而不检查待执行命令是否为`COPY FROM STDIN` 的客户端库。）

拷贝出模式（数据从服务器发出）是在后端执行一个 `COPY TO STDOUT` 语句时启动的。后端发出一个 CopyOutResponse 消息给前端，后面跟着零或者多个 CopyData 消息（总是每行一个），然后跟着 CopyDone。然后后端回退到它在`COPY` 开始之前的命令处理模式，然后发送 CommandComplete。前端不能中止传输（除非是关闭连接或者发出一个 Cancel 请求），但是它可以抛弃不需要的 CopyData 和 CopyDone 消息。

在拷贝出模式中，如果后端检测到错误，那么它将发出一个 ErrorResponse 消息并且回到正常的处理。前端应该把收到 ErrorResponse 当作终止拷贝出模式的标志。

NoticeResponse 和 ParameterStatus 消息可能穿插在 CopyData 消息之间；前端必须处理这些情况，并应准备好处理其他异步消息类型（参见第 53.2.7 节）。除此之外，可以将任何除 CopyData 或 CopyDone 以外的消息类型视为拷贝出模式的终止标志。

还有另一种与拷贝相关的模式，称为双向拷贝（copy-both），它允许高速批量地向服务器发送数据以及从服务器接收数据。当处于 WAL 发送进程模式的后端执行 `START_REPLICATION` 语句时，会启动双向拷贝模式。后端会向前端发送一条 CopyBothResponse 消息。此后，前端和后端都可以发送 CopyData 消息，直到任一方发送 CopyDone 消息。客户端发送 CopyDone 后，连接会从双向拷贝模式切换到拷贝出模式，客户端也不得再发送 CopyData。类似地，当服务器发送 CopyDone 后，连接会进入拷贝入模式，服务器也不得再发送 CopyData。当双方都发送完 CopyDone 后，拷贝模式结束，后端恢复到原先的命令处理模式。如果双向拷贝模式期间发生后端检测到的错误，后端会发出 ErrorResponse，丢弃前端消息直到收到 Sync，然后发出 ReadyForQuery 并返回正常处理。前端应将收到 ErrorResponse 视为双向拷贝终止的信号；在这种情况下不应再发送 CopyDone。关于在双向拷贝模式上传输的子协议，见第 53.4 节。

CopyInResponse、CopyOutResponse 和 CopyBothResponse 消息包含一些字段，用于告知前端每行的列数以及每列所使用的格式代码。（在当前实现中，同一次`COPY` 操作的所有列都使用相同格式，但消息设计并不作此假设。）

53.2.7. 异步操作

在若干情况下，后端会发送并非由前端命令流直接触发的消息。前端必须随时准备处理这些消息，即使当前并未处于查询过程中。至少，在开始读取查询响应之前应检查这些情况。

NoticeResponse 消息可能因外部活动而产生；例如，如果数据库管理员发起一次“快速”数据库关闭，后端会在关闭连接之前发送一条 NoticeResponse 说明这一事实。因此，前端应始终准备好接收并显示 NoticeResponse 消息，即使连接表面上处于空闲状态。

只要后端认为前端应当知晓的某个参数的当前有效值发生变化，就会生成 ParameterStatus 消息。最常见的情况是响应前端执行的`SET` 命令，这种情况实际上是同步的；但也可能是管理员修改了配置文件，然后向服务器发送SIGHUP 信号，从而导致参数状态发生变化。同样，如果某条 `SET` 命令被回滚，也会生成适当的 ParameterStatus 消息，用于报告当前生效的值。

目前，会为一组固定的参数生成 ParameterStatus，参数如下：`server_version`、`server_encoding`、`client_encoding`、`application_name`、`is_superuser`、`session_authorization`、`DateStyle`、`IntervalStyle`、`TimeZone`、`integer_datetimes`、`standard_conforming_strings`。（8.0 之前的版本不报告 `server_encoding`、`TimeZone` 和 `integer_datetimes`；8.1 之前的版本不报告 `standard_conforming_strings`；8.4 之前的版本不报告 `IntervalStyle`；9.0 之前的版本不报告 `application_name`。）注意，`server_version`、`server_encoding` 和 `integer_datetimes` 是启动后不能改变的伪参数。这组参数将来可能变化，甚至可能变为可配置。因此，前端应直接忽略其不理解或不关心的参数的 ParameterStatus。

如果前端发出 `LISTEN` 命令，那么每当针对同一通道名执行`NOTIFY` 命令时，后端都会发送一条 NotificationResponse 消息（不要与 NoticeResponse 混淆）。

> **注意**
>
> 目前，NotificationResponse 只能在一个事务外面发送，因此它将不会在一个命令响应序列中间出现，但是它可能正好在 ReadyForQuery 之前出现。不过，在前端逻辑中做上述假设是不明智的。好的做法是在协议的任何点上都可以接受 NotificationResponse。

53.2.8. 取消正在处理的请求

在一条查询正在处理的时候，前端可以请求取消该查询。这种取消请求不是直接通过打开的连接发送给后端的，这么做是因为实现的效率：我们不希望后端在处理查询的过程中不停地检查前端来的输入。取消请求应该相对而言比较少见，所以我们把取消做得稍微笨拙一些，以便不影响正常状况的性能。

要发出取消请求，前端会新建到服务器的连接，并发送 CancelRequest 消息，而不是新连接通常发送的 StartupMessage 消息。服务器处理该请求后便会关闭连接。出于安全原因，服务器不会直接回复取消请求消息。

除非 CancelRequest 消息包含在连接启动过程中传递给前端的相同的密钥数据（PID 和密钥），否则它将被忽略。如果该请求匹配当前运行着的后端的 PID 和密钥，则中止当前查询的处理（目前的实现里采用的方法是向正在处理该查询的后端进程发送一个特殊的信号）。

取消信号可能有效，也可能无效；例如，如果它在后端已经处理完查询之后才到达，就不会起作用。如果取消生效，当前命令就会以一条错误消息提前终止。

这么做是对安全性和效率通盘考虑的结果，前端没有直接的方法获知一个取消请求是否成功。它必须继续等待后端对查询的响应。发出一个取消仅仅是增加了当前查询快些结束的可能性，同时也增加了当前查询会伴随着一条错误消息失败而不是成功执行的可能性。

由于取消请求是通过一条新的连接发送给服务器，而不是通过常规的前端/后端通信链路发送，因此发出取消请求的可以是任意进程，而不一定非要是要取消查询的那个前端。这为构建多进程应用提供了额外的灵活性，同时也带来了安全风险，因为未授权用户可能会尝试取消查询。通过要求在取消请求中提供动态生成的密钥，可以缓解这一安全风险。

53.2.9. 终止

通常优雅的终止过程是前端发送一条 Terminate 消息并且立刻关闭连接。一旦收到消息，后端马上关闭连接并且终止。

在少数情况下（比如一个管理员命令数据库关闭），后端可能在没有任何前端请求的情况下断开连接。在这种情况下，后端将在它断开连接之前尝试发送一个错误或者通知消息给出断开的原因。

其他终止场景来自各种故障，例如任一端发生 core dump、通信链路中断、消息边界同步丢失等。如果前端或后端看到连接意外关闭，就应清理并终止。若前端不想自行终止，也可以重新联系服务器以启动一个新的后端。如果收到无法识别的消息类型，同样建议关闭连接，因为这通常意味着消息边界同步已经丢失。

不管是正常还是不正常的终止，任何打开的事务都会回滚而不是提交。不过，我们应该注意的是如果一个前端在一个非`SELECT`查询正在处理的时候断开，那么后端很可能在发现断开之前先完成查询的处理。如果查询处于任何事务块之外（`BEGIN` ... `COMMIT` 序列），那么其结果可能在发现连接断开之前被提交。

53.2.10. SSL 会话加密

如果编译 PostgreSQL 时启用了 SSL 支持，那么前端/后端通信就可以使用SSL 加密。这为攻击者可能截获会话流量的环境提供了通信安全性。有关使用 SSL 加密 PostgreSQL 会话的更多信息，请参阅第 18.9 节。

要发起 SSL 加密连接，前端首先发送 SSLRequest 消息，而不是 StartupMessage。服务器随后返回包含 `S` 或 `N` 的单个字节，分别表示愿意或不愿意进行 SSL 通信。如果前端不满意此响应，可以在此时关闭连接。要在收到 `S` 后继续，应与服务器执行 SSL 启动握手（这里不作说明，它属于 SSL 规范的一部分）。如果握手成功，则继续发送通常的 StartupMessage。此时，Startup

Message 及其后的所有数据都将经过 SSL 加密。要在收到 `N` 后继续，则发送通常的 Startup Message，并以不加密的方式继续通信。

前端还应准备好处理服务器对 SSLRequest 返回的 ErrorMessage 响应。只有当服务器版本早于 PostgreSQL 引入 SSL 支持时，才会发生这种情况。（这样的服务器已经非常古老，现实中可能已不存在。）此时必须关闭连接，但前端可以选择建立一个新连接，并在不请求 SSL 的情况下继续通信。

当可以执行 SSL 加密时，服务器应仅发送单个 `S` 字节，然后等待前端启动 SSL 握手。如果此时有其他可读取的字节，则很可能意味着中间人正在尝试执行缓冲区填充攻击（CVE-2021-23222[1]）。前端应该编写代码，要么从套接字中恰好读取一个字节，然后将套接字交给所用的 SSL 库，要么在发现已经读取到额外的字节时将其视为协议违规。

如果建立连接是为了发送 CancelRequest 消息，也可以先发送 SSLRequest。

虽然协议本身没有提供让服务器强制使用 SSL 加密的方法，但管理员可以配置服务器，使其在认证检查中拒绝未加密的会话。

53.3. SASL 认证

SASL 是面向连接协议中的认证框架。目前，PostgreSQL 实现了两种 SASL 认证机制：SCRAM-SHA-256 和 SCRAM-SHA-256-PLUS。将来可能增加更多机制。以下步骤说明 SASL 认证的一般流程，下一小节则详细介绍 SCRAM-SHA-256 和 SCRAM-SHA-256-PLUS。

SASL 认证消息流

1. 要开始一个 SASL 认证交换，服务器发送一个 AuthenticationSASL 消息。它包括服务器可以接受的 SASL 认证机制列表，按照服务器的首选顺序排列。

2. 客户端从列表中选择一种受支持的机制，并向服务器发送 SASLInitialResponse 消息。消息包含所选机制的名称；如果该机制使用初始客户端响应，消息还可以包含这一可选响应。

3. 随后会进行一轮或多轮服务器挑战和客户端响应。每次服务器挑战都通过 Authentication SASLContinue 消息发送，随后客户端通过 SASLResponse 消息响应。消息的具体内容取决于所用机制。

4. 最后，认证交换成功完成时，服务器会发送 AuthenticationSASLFinal 消息，紧接着发送 AuthenticationOk 消息。AuthenticationSASLFinal 包含从服务器发给客户端的附加数据，其具体内容取决于所选认证机制。如果该认证机制不使用在完成时发送的附加数据，则不会发送 AuthenticationSASLFinal 消息。

在错误情况下，服务器可以在任何阶段中止认证，并发送一个ErrorMessage。

53.3.1. SCRAM-SHA-256 认证

目前实现的 SASL 机制是 `SCRAM-SHA-256` 及其带通道绑定的变体 `SCRAM-SHA-256-PLUS`。RFC 7677 和 RFC 5802 对它们有详细描述。

当在 PostgreSQL 中使用 SCRAM-SHA-256 时，服务器将忽略客户端在 `client-first-message` 中发送的用户名。而是使用已经在启动消息中发送的用户名。PostgreSQL 支持多种字符编码，而 SCRAM 规定用户名必须使用 UTF-8，因此可能无法用 UTF-8 表示 PostgreSQL 用户名。

[1] https://www.postgresql.org/support/security/CVE-2021-23222/

SCRAM 规范规定密码也必须采用 UTF-8 编码，并通过SASLprep 算法处理。不过，PostgreSQL 并不要求密码使用 UTF-8。设置用户密码时，无论实际采用何种编码，都会将其视作 UTF-8 并用 SASLprep 处理。但是，如果密码不是合法的 UTF-8 字节序列，或者包含 SASLprep 算法禁止的 UTF-8 字节序列，就会直接使用未经 SASLprep 处理的原始密码，而不抛出错误。这样既能对 UTF-8 密码进行规范化，又允许使用非 UTF-8 密码，而且不要求系统知道密码采用何种编码。

通道绑定在支持 SSL 的 PostgreSQL 构建中受支持。带有通道绑定的 SCRAM 的 SASL 机制名称是 `SCRAM-SHA-256-PLUS`。PostgreSQL 使用的通道绑定类型是 `tls-server-end-point`。

在不带通道绑定的 SCRAM 中，服务器会选择一个随机数并发送给客户端，将它与用户提供的密码混合，形成所传输的密码 hash。虽然这样可以防止在后续会话中成功重放该密码 hash，但无法阻止位于真实服务器与客户端之间的伪造服务器转发服务器的随机值并成功通过认证。

带通道绑定的 SCRAM 会将服务器证书的签名混入所传输的密码 hash，从而防止此类中间人攻击。虽然伪造服务器可以转发真实服务器的证书，但它无法取得与证书匹配的私钥，因此无法证明自己是证书所有者，最终导致 SSL 连接失败。

示例

1. 服务器发送一个 AuthenticationSASL 消息。它包括服务器可以接受的 SASL 认证机制列表。如果服务器构建时支持 SSL，这将是`SCRAM-SHA-256-PLUS` 和 `SCRAM-SHA-256`，否则只是后者。

2. 客户端通过发送 SASLInitialResponse 消息做出响应，该消息指示选择的机制，`SCRAM-SHA-256` 或 `SCRAM-SHA-256-PLUS`。（客户端可以自由选择任一机制，但为了更好的安全性，如果支持的话应选择通道绑定变体。）在初始客户端响应字段中，消息包含 SCRAM `client-first-message`。`client-first-message` 还包含客户端选择的通道绑定类型。

3. 服务器发送一个 AuthenticationSASLContinue 消息，其中包含一个 SCRAM `server-first-message` 作为内容。

4. 客户端发送一个 SASLResponse 消息，其中包含 SCRAM `client-final-message` 作为内容。

5. 服务器发送一个 AuthenticationSASLFinal 消息，带有 SCRAM `server-final-message`，紧接着是一个 AuthenticationOk 消息。

53.4. 流复制协议

要启动流复制，前端在启动消息中发送`replication` 参数。布尔值 `true`（或 `on`、`yes`、`1`）告诉后端进入物理复制 WAL 发送进程模式，在此模式下，可以发出下面列出的一小组复制命令，而不是 SQL 语句。

将 `database` 作为 `replication` 参数的值传递给后端，指示其进入逻辑复制 WAL 发送进程模式，并连接到`dbname` 参数中指定的数据库。在逻辑复制 WAL 发送进程模式下，可以发出下文所示的复制命令以及普通 SQL 命令。

无论是物理复制还是逻辑复制的 WAL 发送进程模式，都只能使用简单查询协议。

为了测试复制命令，可以通过 psql 或其他使用 libpq 的工具建立复制连接，连接字符串中应包含 replication 选项，例如：

```
psql "dbname=postgres replication=database" -c "IDENTIFY_SYSTEM;"
```

不过，通常更有用的做法是使用pg_receivewal（用于物理复制）或 pg_recvlogical（用于逻辑复制）。

启用 log_replication_commands 后，复制命令会记录到服务器日志中。

复制模式接受以下命令：

IDENTIFY_SYSTEM

请求服务器标识自身。服务器会返回一个包含四个字段的单行结果集：

systemid (text)

用于标识该集簇的唯一系统标识符。它可用于检查初始化备库所用的基础备份是否来自同一个集簇。

timeline (int4)

当前时间线 ID。也可用于检查备库是否与主库一致。

xlogpos (text)

当前 WAL 刷盘位置。可用于获取预写式日志中一个已知的位置，以便从该处开始流式传输。

dbname (text)

当前连接到的数据库，或为 null。

SHOW *name*

请求服务器发送运行时参数的当前设置。这类似于 SQL 命令 SHOW。

name

一个运行时参数的名称。可用参数在第 19 章中有文档记录。

TIMELINE_HISTORY *tli*

请求服务器发送时间线 *tli* 的时间线历史文件。服务器返回一个只有一行的结果集，包含两个字段。虽然这些字段被标记为 text 和 bytea，它们实际返回的是原始字节，不进行转义或编码转换：

filename (text)

时间线历史文件的文件名，例如，00000002.history。

content (bytea)

时间线历史文件的内容。

```
CREATE_REPLICATION_SLOT slot_name [ TEMPORARY ] { PHYSICAL [ RESERVE_WAL ] |
LOGICAL output_plugin [ EXPORT_SNAPSHOT | NOEXPORT_SNAPSHOT | USE_SNAPSHOT ] }
```

创建一个物理或逻辑复制槽。查看第 26.2.6 节了解更多关于复制槽的信息。

slot_name

要创建的复制槽名称。必须是合法的复制槽名称（参见第 26.2.6.1 节）。

output_plugin

用于逻辑解码的输出插件的名称（参见第 49.6 节）。

TEMPORARY

指定此复制槽为临时槽。临时槽不会保存到磁盘，并且在错误或会话结束时会自动删除。

RESERVE_WAL

指定此物理复制槽立即保留 WAL。否则，只有在流复制客户端连接时才会保留 WAL。

EXPORT_SNAPSHOT
NOEXPORT_SNAPSHOT
USE_SNAPSHOT

决定如何处理初始化逻辑复制槽时创建的快照。默认选项 EXPORT_SNAPSHOT 会导出快照供其他会话使用，此选项不能在事务内使用。USE_SNAPSHOT 会将快照用于执行该命令的当前事务；此选项必须在事务内使用，且 CREATE_REPLICATION_SLOT 必须是该事务中执行的第一条命令。最后，NOEXPORT_SNAPSHOT 只会像平常一样将快照用于逻辑解码，不会对其执行其他操作。

在响应此命令时，服务器将发送一个包含以下字段的单行结果集：

slot_name (text)

新创建的复制槽的名称。

consistent_point (text)

复制槽达到一致状态时的 WAL 位置。这是可从该复制槽开始流式传输的最早位置。

snapshot_name (text)

该命令导出的快照标识符。该快照在此连接上执行下一条命令之前，或复制连接关闭之前有效。如果所创建的是物理复制槽，则为空值。

output_plugin (text)

新建复制槽所使用的输出插件名称。如果所创建的是物理复制槽，则为空值。

```
START_REPLICATION [ SLOT slot_name ] [ PHYSICAL ] XXX/XXX [ TIMELINE tli ]
```

指示服务器从 WAL 位置 *XXX/XXX* 开始流式传输 WAL。如果指定了 TIMELINE 选项，则从时间线 *tli* 开始传输；否则选择服务器当前的时间线。服务器可以返回错误，例如请求的 WAL 部分已被回收时。成功时，服务器返回 CopyBothResponse 消息，然后开始向前端流式传输 WAL。

如果通过 *slot_name* 提供了复制槽名称，该复制槽会随着复制的进行而更新，使服务器知道备库仍需要哪些 WAL 段，以及在启用 hot_standby_feedback 时仍需要哪些事务。

如果客户端请求的时间线不是最新时间线，但属于服务器的历史，服务器会从请求的起点开始，流式传输该时间线上的所有 WAL，直到服务器切换到另一条时间线的位置。如果客户端请求的流式传输起点恰好位于旧时间线的末尾，服务器会立即返回 CommandComplete，而不进入 COPY 模式。

在非最新时间线上流式传输完全部 WAL 后，服务器会通过退出 COPY 模式来结束流式传输。当客户端也通过退出 COPY 模式来确认时，服务器会发送一个包含一行两列的结果集，指示该服务器历史中的下一条时间线。第一列是下一条时间线的 ID（类型为 int8），第二列是发生切换的 WAL 位置（类型为 text）。通常，切换位置就是所流式传输 WAL 的末尾，但也存在一些边界情况，服务器可能会先发送一些自己在提升前尚未重放的旧时间线 WAL。最后，服务器发送 CommandComplete 消息，然后准备接受新的命令。

WAL 数据通过一系列 CopyData 消息发送。（这样可以混合发送其他信息；尤其是服务器在开始流式传输后遇到故障时，可以发送 ErrorResponse 消息。）服务器发给客户端的每条 CopyData 消息，其有效载荷都包含一条具有下列格式之一的消息：

XLogData (B)

Byte1('w')

将该消息标识为 WAL 数据。

Int64

本消息中 WAL 数据的起始位置。

Int64

服务器上当前的 WAL 末尾位置。

Int64

发送消息时服务器的系统时钟，以自 2000-01-01 午夜以来的微秒数表示。

Byte*n*

WAL 数据流的一个片段。

单条 WAL 记录绝不会被拆分到两条 XLogData 消息中。当 WAL 记录跨越 WAL 页边界，因而已经通过续接记录拆分时，可以在页边界处分开发送。换句话说，最初的主 WAL 记录及其续接记录可以在不同的 XLogData 消息中发送。

主库保活消息 (B)

Byte1('k')

将该消息标识为发送端保活消息。

Int64

服务器上当前的 WAL 末尾位置。

Int64

发送消息时服务器的系统时钟，以自 2000-01-01 午夜以来的微秒数表示。

Byte1

1 表示客户端应尽快回复此消息，以避免超时断开连接；否则为 0。

接收进程可以随时使用以下消息格式之一回复发送端（同样放在 CopyData 消息的有效载荷中）：

备库状态更新 (F)

Byte1('r')

将该消息标识为接收端状态更新。

Int64

备库已接收并写入磁盘的最后一个 WAL 字节的位置加 1。

Int64

备库已刷盘的最后一个 WAL 字节的位置加 1。

Int64

备库已应用的最后一个 WAL 字节的位置加 1。

Int64

发送消息时客户端的系统时钟，以自 2000-01-01 午夜以来的微秒数表示。

Byte1

如果为 1，表示客户端请求服务器立即回复此消息。可用它向服务器发送探测请求，以测试连接是否仍然正常。

热备反馈消息 (F)

Byte1('h')

将该消息标识为热备反馈消息。

Int64

发送消息时客户端的系统时钟，以自 2000-01-01 午夜以来的微秒数表示。

Int32

备库当前的全局 xmin，不包括任何复制槽的 catalog_xmin。如果此值及后面的 catalog_xmin 都为 0，则视为通知服务器：此连接将不再发送热备反馈。之后的非零消息可以重新启动反馈机制。

Int32

备库全局 xmin 事务 ID 的纪元。

Int32

备库上所有复制槽中最小的 catalog_xmin。如果备库上不存在 catalog_xmin，或者正在禁用热备反馈，则设为 0。

Int32

备库 catalog_xmin 事务 ID 的纪元。

`START_REPLICATION SLOT` *`slot_name`* `LOGICAL` *`XXX/XXX`* [(*`option_name`* [*`option_value`*] [, ...])]

指示服务器开始为逻辑复制流式传输 WAL，起始 WAL 位置为 *`XXX/XXX`*。服务器可以返回错误，例如请求的 WAL 段已被回收。成功时，服务器会响应一条 CopyBothResponse 消息，然后开始向前端流式传输 WAL。

CopyBothResponse 中承载的消息采用与 `START_REPLICATION ... PHYSICAL` 文档中记载的相同格式。

与所选复制槽关联的输出插件将用于处理流式输出。

`SLOT` *`slot_name`*

要从中流式传输更改的复制槽名称。该参数是必需的，并且必须对应于使用 `CREATE_REPLICATION_SLOT` 在 `LOGICAL` 模式下创建的现有逻辑复制槽。

`XXX/XXX`

开始流式传输的 WAL 位置。

`option_name`

传递给复制槽逻辑解码插件的选项名称。

`option_value`

指定选项相关的可选值，以字符串常量的形式表示。

`DROP_REPLICATION_SLOT` *`slot_name`* [`WAIT`]

删除复制槽，释放任何保留的服务器端资源。如果该槽是逻辑槽，且创建它的数据库与 WAL 发送进程当前连接的数据库不同，则此命令失败。

`slot_name`

要删除的复制槽名称。

`WAIT`

该选项会让命令在复制槽处于活跃状态时等待，直到其变为非活跃，而不是按默认行为报错。

BASE_BACKUP [LABEL *'label'*] [PROGRESS] [FAST] [WAL] [NOWAIT] [MAX_RATE *rate*] [TABLESPACE_MAP] [NOVERIFY_CHECKSUMS]

指示服务器开始流式传输基础备份。在备份开始之前，系统将自动进入备份模式，并在备份完成后退出备份模式。接受以下选项：

LABEL *'label'*

设置备份标签。如果未指定，则使用 base backup 作为备份标签。标签的引号规则与启用 standard_conforming_strings 时的标准 SQL 字符串相同。

PROGRESS

请求生成进度报告所需的信息。这将在每个表空间的首部发送一个近似大小，可用于计算流式传输的进度。这是通过在传输开始之前先枚举所有文件大小来计算的，可能会对性能产生负面影响。特别是，在流式传输数据之前可能需要更长的时间。由于备份期间数据库文件可能会发生变化，因此大小仅为近似值，在估算与实际发送文件之间的这段时间里可能会增长或缩小。

FAST

请求快速检查点。

WAL

在备份中包含必要的 WAL 段。这会把开始备份到停止备份之间的所有文件放入基础目录 tar 文件内的 pg_wal 目录中。

NOWAIT

默认情况下，备份会等待最后一个必需的 WAL 段完成归档；如果未启用日志归档，则发出警告。指定 NOWAIT 会同时禁用等待和警告，由客户端负责确保所需日志可用。

MAX_RATE *rate*

限制（节流）每单位时间从服务器传输到客户端的最大数据量。预期的单位是每秒千字节。如果指定了此选项，则该值必须等于零，或者必须在 32 kB 到 1 GB（含）的范围内。如果传递零或未指定该选项，则不对传输施加任何限制。

TABLESPACE_MAP

在名为 tablespace_map 的文件中包含目录 pg_tblspc 中存在的符号链接的信息。表空间映射文件包括目录 pg_tblspc/ 中每个符号链接的名称及该符号链接的完整路径。

NOVERIFY_CHECKSUMS

默认情况下，如果启用了校验和，基础备份期间就会验证校验和。指定 NOVERIFY_CHECKSUMS 会禁用此项验证。

备份开始时，服务器首先发送两个普通结果集，然后发送一个或多个 CopyResponse 结果。

第一个普通结果集包含备份的起始位置，在一个包含两列的单行中。第一列包含以 XLogRecPtr 格式给出的起始位置，第二列包含相应的时间线 ID。

第二个普通结果集中的每个表空间都有一行。这一行中的字段是：

`spcoid` (`oid`)

表空间的 OID，如果是基础目录则为 null。

`spclocation` (`text`)

表空间目录的完整路径，如果是基础目录则为 null。

`size` (`int8`)

如果请求了进度报告，则为表空间的大致大小；否则为 null。

在第二个普通结果集之后，会发送一个或多个 CopyResponse 结果，其中一个用于主数据目录，其他结果分别用于 `pg_default` 和 `pg_global` 之外的各个附加表空间。CopyResponse 结果中的数据是表空间内容的 tar 格式转储（遵循 POSIX 1003.1-2008 标准中规定的“ustar interchange format”），但省略了标准规定的末尾两个全零块。tar 数据传输完成后，会发送最后一个普通结果集，其中包含备份的 WAL 结束位置，格式与起始位置相同。

数据目录和每个表空间的 tar 归档将包含目录中的所有文件，无论它们是PostgreSQL 文件还是添加到同一目录的其他文件。唯一排除的文件是：

- `postmaster.pid`
- `postmaster.opts`
- `pg_internal.init`（在多个目录中找到）
- PostgreSQL 服务器运行期间创建的各种临时文件和目录，例如以`pgsql_tmp` 开头的任何文件或目录以及临时关系。
- 不记录 WAL 的关系，但不包括恢复时重新创建（空的）不记录 WAL 的关系所需的初始化分支。
- `pg_wal`，包括子目录。如果备份包含 WAL 文件，则将包含`pg_wal` 的合成版本，但它只包含使备份可用所需的文件，而不包含其余内容。
- `pg_dynshmem`，`pg_notify`，`pg_replslot`，`pg_serial`，`pg_snapshots`，`pg_stat_tmp` 和 `pg_subtrans` 将作为空目录复制（即使它们是符号链接）。
- 除了普通文件和目录之外的文件，如符号链接（除了上述列出的目录之外的符号链接）以及特殊设备文件，将被跳过。（`pg_tblspc` 中的符号链接将被保留。）

如果服务器上的底层文件系统支持，将设置所有者、组和文件模式。

53.5. 逻辑流复制协议

本节介绍逻辑复制协议，它是一种以复制命令`START_REPLICATION SLOT` *`slot_name`* `LOGICAL` 开始的消息流。

逻辑流复制协议构建在物理流复制协议的基本机制之上。

53.5.1. 逻辑流复制参数

逻辑复制的 `START_REPLICATION` 命令接受以下参数：

proto_version

协议版本。目前仅支持版本 `1`。

publication_names

要订阅（接收变更）的发布名称列表，以逗号分隔。各个发布名称按标准对象名处理，可按需加引号。

53.5.2. 逻辑复制协议消息

各种协议消息将在后续小节中分别讨论。每种消息的具体格式见第 53.9 节。

所有顶层协议消息都以一个消息类型字节开头。虽然它通常写作字符代码，但本质上它是一个与任何字符编码无关的有符号字节。

由于流复制协议本身提供了消息长度，因此顶层协议消息无需在其头部内嵌长度字段。

53.5.3. 逻辑复制协议消息流

除 START_REPLICATION 命令和重放进度消息之外，所有信息流方向都是从后端到前端。

逻辑复制协议逐个发送事务。这意味着，一对 Begin 和 Commit 消息之间的所有消息都属于同一个事务。

每个发送的事务都包含零条或多条 DML 消息（Insert、Update、Delete）。在级联配置中，还可能包含 Origin 消息。Origin 消息表示该事务源自另一个复制节点。由于逻辑复制协议中的复制节点几乎可以是任何实现，因此唯一的标识符就是复制源的名称。下游负责按需处理这一信息（如果需要处理）。Origin 消息总是在事务中的任何 DML 消息之前发送。

每条 DML 消息都包含一个关系 OID，用于标识被操作的发布者关系。在某个关系 OID 的第一条 DML 消息之前，会先发送一条 Relation 消息，描述该关系的模式。之后，如果该关系的定义自上次发送 Relation 消息以来发生了变化，就会再发送一条新的 Relation 消息。（协议假定客户端能够缓存所需关系的元数据。）

Relation 消息通过 OID 标识列类型。对于内置类型，假定客户端可以在本地查找该类型的 OID，因此无需额外数据。对于非内置类型 OID，会在 Relation 消息之前发送一条 Type 消息，以提供与该 OID 关联的类型名称。因此，需要明确识别关系列类型的客户端应缓存 Type 消息内容，并先检查该缓存中是否已经定义了对应的类型 OID；若没有，再在本地查找该类型 OID。

53.6. 消息数据类型

本节描述了消息中使用的基本数据类型。

Int$n$(i)

以网络字节序（最高有效字节在前）表示的 n 位整数。如果指定了 i，则表示其精确值；否则表示该值可变。例如：Int16、Int32(42)。

Int$n$[k]

由 k 个 n 位整数组成的数组，每个整数都按网络字节序排列。数组长度 k 总是由消息中更早的某个字段确定。例如：Int16[M]。

String(s)

一个以空字符结尾的字符串（C 风格字符串）。字符串没有特定的长度限制。如果指定了 s，则表示其精确值；否则表示该值可变。例如：String、String("user")。

> **注意**
>
> 后端返回的字符串长度没有预定义的限制。前端较好的编码策略是使用可扩展缓冲区，以便接收所有能放进内存的内容。如果做不到这一点，就应读取完整字符串，并丢弃固定大小的缓冲区容纳不下的尾随字符。

Byte*n*(*c*)

精确 *n* 字节。如果字段宽度 *n* 不是常数，它总是可以由消息中更早的字段确定。如果指定了 *c*，则表示其精确值。例如：Byte2、Byte1('\n')。

53.7. 消息格式

本节描述每条消息的详细格式。每条消息都标明它可以由前端（F）、后端（B）或双方（F & B）发送。注意，虽然每条消息开头都有字节计数，但消息格式的定义使得无需参考该计数也能确定消息的结束位置。这有助于检查消息的有效性。（CopyData 消息是例外，因为它构成数据流的一部分，任何单条 CopyData 消息的内容都无法独立解释。）

AuthenticationOk (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度（以字节为单位），包括其自身。

Int32(0)

表示认证成功。

AuthenticationKerberosV5 (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度（以字节为单位），包括其自身。

Int32(2)

表示需要 Kerberos V5 认证。

AuthenticationCleartextPassword (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度（以字节为单位），包括其自身。

Int32(3)

表示需要明文密码。

AuthenticationMD5Password (B)

Byte1('R')

将该消息标识为认证请求。

Int32(12)

消息内容的长度（以字节为单位），包括其自身。

Int32(5)

表示需要经过 MD5 加密的密码。

Byte4

加密密码时使用的盐。

AuthenticationSCMCredential (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度（以字节为单位），包括其自身。

Int32(6)

表示需要 SCM 凭证消息。

AuthenticationGSS (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度（以字节为单位），包括其自身。

Int32(7)

表示需要 GSSAPI 认证。

AuthenticationSSPI (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度（以字节为单位），包括其自身。

Int32(9)

表示需要 SSPI 认证。

AuthenticationGSSContinue (B)

Byte1('R')

将该消息标识为认证请求。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int32(8)

表示此消息包含 GSSAPI 或 SSPI 数据。

Byte*n*

GSSAPI 或 SSPI 认证数据。

AuthenticationSASL (B)

Byte1('R')

将该消息标识为认证请求。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int32(10)

表示需要 SASL 认证。

消息体是按服务器偏好顺序排列的 SASL 认证机制列表。在最后一个认证机制名称之后，必须有一个零字节作为终止符。每个机制包含以下内容：

String

SASL 认证机制的名称。

AuthenticationSASLContinue (B)

Byte1('R')

将该消息标识为认证请求。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int32(11)

表示此消息包含 SASL 挑战。

Byte*n*

SASL 数据，具体内容取决于所使用的 SASL 机制。

AuthenticationSASLFinal (B)

Byte1('R')

将该消息标识为认证请求。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int32(12)

表示 SASL 认证已完成。

Byte*n*

SASL 结果的“附加数据”，具体内容取决于所使用的 SASL 机制。

BackendKeyData (B)

Byte1('K')

将此消息标识为取消请求密钥数据。如果前端希望以后能够发送 CancelRequest 消息，就必须保存这些值。

Int32(12)

消息内容的长度（以字节为单位），包括其自身。

Int32

此后端的进程 ID。

Int32

此后端的密钥。

Bind (F)

Byte1('B')

将该消息标识为 Bind 命令。

Int32

消息内容的长度（以字节为单位），包括其自身。

String

目标 portal 的名称（空字符串选择未命名的 portal）。

String

源预备语句的名称（空字符串选择未命名的预备语句）。

Int16

后续参数格式代码的数量（下文以 C 表示）。可以为零，表示没有参数，或者所有参数都使用默认格式（文本）；也可以为一，此时指定的格式代码应用于所有参数；还可以等于实际参数数量。

Int16[C]

参数格式代码。目前每个格式代码必须为零（文本）或一（二进制）。

Int16

后续参数值的数量（可以为零）。必须与查询所需的参数数量一致。

接下来，每个参数都有以下一对字段：

Int32

参数值的长度，以字节为单位（不包括此长度字段本身）。可以为零。特殊值 -1 表示 NULL 参数值。为 NULL 时，后面不再有值的字节。

Byte$n$

参数值，格式由对应的格式代码指明。n 为上述长度。

最后一个参数之后是以下字段：

Int16

后续结果列格式代码的数量（下文以 R 表示）。可以为零，表示没有结果列，或者所有结果列都应使用默认格式（文本）；也可以为一，此时指定的格式代码应用于所有结果列（如果有）；还可以等于查询实际的结果列数量。

Int16[R]

结果列格式代码。目前每个格式代码必须为零（文本）或一（二进制）。

BindComplete (B)

Byte1('2')

将该消息标识为 Bind 完成指示。

Int32(4)

消息内容的长度（以字节为单位），包括其自身。

CancelRequest (F)

Int32(16)

消息内容的长度（以字节为单位），包括其自身。

Int32(80877102)

取消请求代码。此值的最高 16 位为 `1234`，最低 16 位为 `5678`。（为避免混淆，此代码不能与任何协议版本号相同。）

Int32

目标后端的进程 ID。

Int32

目标后端的密钥。

Close (F)

Byte1('C')

将该消息标识为 Close 命令。

Int32

消息内容的长度（以字节为单位），包括其自身。

Byte1

“`S`”表示关闭预备语句；“`P`”表示关闭 portal。

String

要关闭的预备语句或 portal 的名称（空字符串选择未命名的预备语句或 portal）。

CloseComplete (B)

Byte1('3')

将该消息标识为 Close 完成指示。

Int32(4)

消息内容的长度（以字节为单位），包括其自身。

CommandComplete (B)

Byte1('C')

将该消息标识为命令完成响应。

Int32

消息内容的长度（以字节为单位），包括其自身。

String

命令标签。通常是一个单词，用来标识已完成的 SQL 命令。

对于 `INSERT` 命令，标签是 `INSERT` *`oid rows`*，其中 *`rows`* 是插入的行数。如果 *`rows`* 为 1 且目标表具有 OID，则 *`oid`* 是插入行的对象 ID；否则 *`oid`* 为 0。

对于 DELETE 命令，标签是DELETE *rows*，其中*rows* 表示删除的行数。

对于 UPDATE 命令，标签是UPDATE *rows*，其中*rows* 是更新的行数。

对于 SELECT 或 CREATE TABLE AS 命令，标签是SELECT *rows*，其中*rows* 是检索到的行数。

对于 MOVE 命令，标签是MOVE *rows*，其中*rows* 表示游标位置改变的行数。

对于 FETCH 命令，标签是FETCH *rows*，其中*rows* 是从游标中检索出的行数。

对于 COPY 命令，标签为 COPY *rows*，其中 *rows* 是复制的行数。（注意：行数仅出现在 PostgreSQL 8.2 及更高版本中。）

CopyData (F & B)

Byte1('d')

标识消息为 COPY 数据。

Int32

消息内容的长度（以字节为单位），包括其自身。

Byte*n*

数据是 COPY 数据流的一部分。来自后端的消息始终对应单个数据行，但来自前端的消息可能会任意划分数据流。

CopyDone (F & B)

Byte1('c')

将消息标识为 COPY 完成指示符。

Int32(4)

消息内容的长度（以字节为单位），包括其自身。

CopyFail (F)

Byte1('f')

将消息标识为 COPY 失败指示器。

Int32

消息内容的长度（以字节为单位），包括其自身。

String

作为失败原因报告的错误消息。

CopyInResponse (B)

Byte1('G')

将该消息标识为开始 COPY 输入的响应。前端此时必须发送 COPY 输入数据（如果尚未准备好，应发送 CopyFail 消息）。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int8

0 表示整体 `COPY` 格式是文本的（行由换行符分隔，列由分隔符分隔等）。1 表示整体复制格式是二进制的（类似于 DataRow 格式）。更多信息请参见 COPY。

Int16

要复制的数据中的列数（以下用 N 表示）。

Int16[N]

各列使用的格式代码。目前每个代码必须为零（文本）或一（二进制）。如果整体复制格式为文本，则所有代码都必须为零。

CopyOutResponse (B)

Byte1('H')

将该消息标识为开始 COPY 输出的响应。此消息之后会发送 COPY 输出数据。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int8

0 表示整体 `COPY` 格式是文本的（行由换行符分隔，列由分隔符分隔等）。1 表示整体复制格式是二进制的（类似于 DataRow 格式）。更多信息请参见 COPY。

Int16

要复制的数据中的列数（以下用 N 表示）。

Int16[N]

各列使用的格式代码。目前每个代码必须为零（文本）或一（二进制）。如果整体复制格式为文本，则所有代码都必须为零。

CopyBothResponse (B)

Byte1('W')

将该消息标识为开始双向 COPY 的响应。此消息仅用于流复制。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int8

0 表示整体 COPY 格式是文本的（行由换行符分隔，列由分隔符分隔等）。1 表示整体复制格式是二进制的（类似于 DataRow 格式）。更多信息请参见 COPY。

Int16

要复制的数据中的列数（以下用 *N* 表示）。

Int16[*N*]

各列使用的格式代码。目前每个代码必须为零（文本）或一（二进制）。如果整体复制格式为文本，则所有代码都必须为零。

DataRow (B)

Byte1('D')

标识消息为数据行。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int16

后面跟着的列值的数量（可能为零）。

接下来，每列都有以下两个字段：

Int32

列值的长度，以字节为单位（不包括此长度字段本身）。可以为零。特殊值 -1 表示 NULL 列值。为 NULL 时，后面不再有值的字节。

Byte*n*

列的值，格式由相关的格式代码指示。*n* 是上述长度。

Describe (F)

Byte1('D')

将该消息标识为 Describe 命令。

Int32

消息内容的长度（以字节为单位），包括其自身。

Byte1

'S' 表示描述一个预备语句；或者 'P' 表示描述一个 portal。

String

要描述的预备语句或 portal 的名称（空字符串选择未命名的预备语句或 portal）。

EmptyQueryResponse (B)

Byte1('I')

标识消息为对空查询字符串的响应。（此消息替代 CommandComplete。）

Int32(4)

消息内容的长度（以字节为单位），包括其自身。

ErrorResponse (B)

Byte1('E')

将消息标识为错误。

Int32

消息内容的长度（以字节为单位），包括其自身。

消息体由一个或多个带标识的字段组成，最后以一个零字节终止。字段可以按任意顺序出现。每个字段包含以下内容：

Byte1

一个用于标识字段类型的代码；如果为零，则这是消息终止符，后面没有字符串。目前定义的字段类型列在第 53.8 节中。由于将来可能会添加更多的字段类型，前端应该静默地忽略未识别类型的字段。

String

字段值。

Execute (F)

Byte1('E')

将该消息标识为 Execute 命令。

Int32

消息内容的长度（以字节为单位），包括其自身。

String

要执行的 portal 的名称（空字符串选择未命名的 portal）。

Int32

如果 portal 包含返回行的查询，则这是最多返回的行数（否则忽略此值）。零表示“无限制”。

Flush (F)

Byte1('H')

将该消息标识为 Flush 命令。

Int32(4)

消息内容的长度（以字节为单位），包括其自身。

FunctionCall (F)

Byte1('F')

标识消息为函数调用。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int32

指定要调用的函数的对象 ID。

Int16

后续参数格式代码的数量（以下用 *C* 表示）。可以为零，表示没有参数，或所有参数都采用默认格式（文本）；也可以为一，表示将指定的格式代码用于所有参数；还可以等于实际参数数量。

Int16[*C*]

参数格式代码。每个目前必须是零（文本）或一（二进制）。

Int16

指定传递给函数的参数数量。

接下来，每个参数都有以下两个字段：

Int32

参数值的长度，以字节为单位（不包括此长度字段本身）。可以为零。特殊值 -1 表示 NULL 参数值。为 NULL 时，后面不再有值的字节。

Byte*n*

参数的值，以相关格式代码指示的格式表示。*n* 是上述长度。

最后一个参数之后还有以下字段：

Int16

函数结果的格式代码。目前必须为零（文本）或一（二进制）。

FunctionCallResponse (B)

Byte1('V')

标识消息为函数调用结果。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int32

函数结果值的长度，以字节为单位（不包括此长度字段本身）。可以为零。特殊值 -1 表示 NULL 函数结果。为 NULL 时，后面不再有值的字节。

Byte*n*

函数结果的值，格式由相关的格式代码指示。*n* 是上述长度。

GSSResponse (F)

Byte1('p')

标识消息为 GSSAPI 或 SSPI 响应。请注意，这也用于 SASL 和密码响应消息。可以从上下文中推断出确切的消息类型。

Int32

消息内容的长度（以字节为单位），包括其自身。

Byte*n*

GSSAPI/SSPI 特定的消息数据。

NegotiateProtocolVersion (B)

Byte1('v')

标识消息为协议版本协商消息。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int32

对于客户端请求的协议主版本，服务器所支持的最新协议次版本。

Int32

服务器无法识别的协议选项数量。

接下来，对于服务器无法识别的每个协议选项，都有以下内容：

String

选项名称。

NoData (B)

Byte1('n')

将消息标识为无数据指示器。

Int32(4)

消息内容的长度（以字节为单位），包括其自身。

NoticeResponse (B)

Byte1('N')

将消息标识为通知。

Int32

消息内容的长度（以字节为单位），包括其自身。

消息体由一个或多个带标识的字段组成，最后以一个零字节终止。字段可以按任意顺序出现。每个字段包含以下内容：

Byte1

一个用于标识字段类型的代码；如果为零，则这是消息终止符，后面没有字符串。目前定义的字段类型列在第 53.8 节中。由于将来可能会添加更多的字段类型，前端应该静默地忽略未识别类型的字段。

String

字段值。

NotificationResponse (B)

Byte1('A')

标识消息为通知响应。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int32

发出通知的后端进程的进程 ID。

String

发出该通知的通道名称。

String

通知进程传来的“有效载荷”字符串。

ParameterDescription (B)

Byte1('t')

标识消息为参数描述。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int16

语句使用的参数数量（可以为零）。

接下来，每个参数都有以下内容：

Int32

指定参数数据类型的对象 ID。

ParameterStatus (B)

Byte1('S')

标识消息为运行时参数状态报告。

Int32

消息内容的长度（以字节为单位），包括其自身。

String

所报告的运行时参数的名称。

String

参数的当前值。

Parse (F)

Byte1('P')

将该消息标识为 Parse 命令。

Int32

消息内容的长度（以字节为单位），包括其自身。

String

目标预备语句的名称（空字符串选择未命名的预备语句）。

String

要解析的查询字符串。

Int16

指定的参数数据类型的数量（可以为零）。请注意，这不是查询字符串中可能出现的参数数量的指示，而是前端希望为其预先指定类型的参数数量。

接下来，每个参数都有以下内容：

Int32

指定参数数据类型的对象 ID。此处填零等同于不指定类型。

ParseComplete (B)

Byte1('1')

将该消息标识为 Parse 完成指示。

Int32(4)

消息内容的长度（以字节为单位），包括其自身。

PasswordMessage (F)

Byte1('p')

标识消息为密码响应。请注意，这也用于 GSSAPI、SSPI 和 SASL 响应消息。可以从上下文中推断出确切的消息类型。

Int32

消息内容的长度（以字节为单位），包括其自身。

String

密码（如果需要，已加密）。

PortalSuspended (B)

Byte1('s')

将该消息标识为 portal 挂起指示。注意，仅当达到 Execute 消息指定的行数限制时，才会出现此消息。

Int32(4)

消息内容的长度（以字节为单位），包括其自身。

Query (F)

Byte1('Q')

标识消息为简单查询。

Int32

消息内容的长度（以字节为单位），包括其自身。

String

查询字符串本身。

ReadyForQuery (B)

Byte1('Z')

标识消息类型。ReadyForQuery 在后端准备好进行新的查询周期时发送。

Int32(5)

消息内容的长度（以字节为单位），包括其自身。

Byte1

当前后端事务状态指示器。可能的值为'I'，如果空闲（不在事务块中）；'T'，如果在事务块中；或'E'，如果在失败的事务块中（查询将被拒绝，直到块结束）。

RowDescription (B)

Byte1('T')

标识消息为行描述。

Int32

消息内容的长度（以字节为单位），包括其自身。

Int16

指定一行中的字段数量（可以为零）。

接下来，每个字段都有以下内容：

String

字段名称。

Int32

如果能够确定该字段是某个特定表的列，则为该表的对象 ID；否则为零。

Int16

如果能够确定该字段是某个特定表的列，则为该列的属性编号；否则为零。

Int32

字段数据类型的对象 ID。

Int16

数据类型大小（参见 `pg_type.typlen`）。注意，负值表示可变宽度类型。

Int32

类型修饰符（参见 `pg_attribute.atttypmod`）。修饰符的含义是特定于类型的。

Int16

字段所使用的格式代码。目前为零（文本）或一（二进制）。对于 Describe 针对预备语句的变体所返回的 RowDescription，格式代码尚未确定，始终为零。

SASLInitialResponse (F)

Byte1('p')

标识消息为初始 SASL 响应。请注意，这也用于 GSSAPI、SSPI 和密码响应消息。精确的消息类型是从上下文中推断出来的。

Int32

消息内容的长度（以字节为单位），包括其自身。

String

客户端选择的 SASL 认证机制的名称。

Int32

后续 SASL 机制特有的“客户端初始响应”的长度；如果没有初始响应，则为 -1。

Byte*n*

SASL 机制特定的“初始响应”。

SASLResponse (F)

Byte1('p')

标识消息为 SASL 响应。请注意，这也用于 GSSAPI、SSPI 和密码响应消息。可以从上下文中推断出确切的消息类型。

Int32

消息内容的长度（以字节为单位），包括其自身。

Byte*n*

SASL 机制特定的消息数据。

SSLRequest (F)

Int32(8)

消息内容的长度（以字节为单位），包括其自身。

Int32(80877103)

SSL 请求代码。该值被选择为在最高的 16 位中包含 `1234`，在最低的 16 位中包含 `5679`。（为避免混淆，此代码不得与任何协议版本号相同。）

StartupMessage (F)

Int32

消息内容的长度（以字节为单位），包括其自身。

Int32(196608)

协议版本号。高 16 位为主版本号（此处描述的协议为 3）；低 16 位为次版本号（此处描述的协议为 0）。

协议版本号之后是一个或多个参数名与参数值字符串对。最后一个名称/值对之后必须有一个零字节作为终止符。参数可以按任意顺序出现。其中，`user` 是必需的，其余均为可选。每个参数按以下方式指定：

String

参数名称。目前能够识别的名称如下：

`user`

要连接的数据库用户名称。必填项；没有默认值。

`database`

要连接的数据库。默认为用户名。

`options`

后端的命令行参数。（已弃用，建议设置单独的运行时参数。）此字符串中的空格被视为分隔参数，除非用反斜杠（`\`）转义；写`\\` 表示字面反斜杠。

`replication`

用于以流复制模式连接，可以发出一小组复制命令而不是 SQL 语句。值可以是 `true`、`false` 或 `database`，默认为`false`。详细信息请参见第 53.4 节。

除上述参数外，还可以列出其他参数。以 `_pq_`.开头的参数名称保留用于协议扩展，其余参数则作为运行时参数，在后端启动时设置。这些设置会在后端启动期间应用（在解析命令行参数之后，如果有的话），并作为会话默认值。

String

参数值。

Sync (F)

Byte1('S')

将该消息标识为 Sync 命令。

Int32(4)

消息内容的长度（以字节为单位），包括其自身。

Terminate (F)

Byte1('X')

标识消息为终止。

Int32(4)

消息内容的长度（以字节为单位），包括其自身。

53.8. 错误和通知消息字段

本节描述了 ErrorResponse 和 NoticeResponse 消息中可能出现的字段。每个字段类型都有一个单字节的标识符。请注意，任何给定的字段类型在消息中最多只能出现一次。

`S`

严重性：字段内容为 `ERROR`、`FATAL` 或 `PANIC`（在错误消息中），或 `WARNING`、`NOTICE`、`DEBUG`、`INFO` 或 `LOG`（在通知消息中），或这些内容的本地化翻译。始终存在。

`V`

严重性：字段内容为 `ERROR`、`FATAL` 或 `PANIC`（在错误消息中），或 `WARNING`、`NOTICE`、`DEBUG`、`INFO` 或 `LOG`（在通知消息中）。它与 `S` 字段相同，只是内容从不本地化。仅在 PostgreSQL 9.6 及更高版本生成的消息中存在。

`C`

代码：错误的 SQLSTATE 代码（见附录 A）。不可本地化。始终存在。

`M`

消息：主要的可读错误消息。应准确而简洁（通常为一行）。始终存在。

`D`

详细信息：可选的次要错误消息，提供有关该问题的更多细节。可能占多行。

`H`

提示：可选的建议，说明如何处理该问题。它与详细信息的区别在于，它提供建议（可能并不合适），而非确凿事实。可能占多行。

`P`

位置：字段值是一个十进制 ASCII 整数，表示错误光标位置，作为原始查询字符串的索引。第一个字符的索引为 1，位置以字符而非字节计量。

`p`

内部位置：这与 `P` 字段的定义相同，但在光标位置指向内部生成的命令而不是客户端提交的命令时使用。当此字段出现时，`q` 字段将始终出现。

`q`

内部查询：一个失败的内部生成命令的文本。例如，这可能是由 PL/pgSQL 函数发出的 SQL 查询。

`W`

上下文：说明错误发生时的上下文。目前包括活跃的过程语言函数和内部生成的查询的调用栈回溯。回溯每行一个条目，最近的调用在前。

`s`

模式名：如果错误与某个特定数据库对象有关，则为包含该对象的模式名称（如果有）。

t

表名：如果错误与某个特定表有关，则为该表的名称。（表所在模式的名称见模式名字段。）

c

列名：如果错误与某个特定表列有关，则为该列的名称。（通过模式名和表名字段确定该表。）

d

数据类型名：如果错误与某个特定数据类型有关，则为该数据类型的名称。（数据类型所在模式的名称见模式名字段。）

n

约束名：如果错误与某个特定约束有关，则为该约束的名称。相关的表或域见上述字段。（这里将索引也视为约束，即使它不是用约束语法创建的。）

F

文件：报告错误的源代码位置所在的文件名。

L

行号：报告错误的源代码位置所在的行号。

R

例程：报告错误的源代码例程的名称。

> **注意**
>
> 模式名、表名、列名、数据类型名和约束名字段仅针对有限的几类错误提供，见附录 A。前端不应假设其中任何字段的存在就保证了另一个字段的存在。核心错误源遵循上述相互关系，但用户定义的函数可能以其他方式使用这些字段。同样，客户端不应假设这些字段表示当前数据库中现存的对象。

客户端负责按自身需要格式化显示的信息；尤其应根据需要将长行断开。错误消息字段中的换行符应视为段落分隔符，而非行分隔符。

53.9. 逻辑复制消息格式

本节描述了每个逻辑复制消息的详细格式。这些消息要么由复制槽 SQL 接口返回，要么由 WAL 发送进程发送。对于 WAL 发送进程，它们被封装在复制协议 WAL 消息中，如第 53.4 节所述，并且通常遵循与物理复制相同的消息流程。

Begin

Byte1('B')

将该消息标识为 Begin 消息。

Int64

事务的最终 LSN。

Int64

事务的提交时间戳。其值为自 PostgreSQL 纪元（2000-01-01）起经过的微秒数。

Int32

事务的 Xid。

Commit

Byte1('C')

将该消息标识为 Commit 消息。

Int8

标志；目前未使用（必须为 0）。

Int64

提交的 LSN。

Int64

事务的结束 LSN。

Int64

事务的提交时间戳。其值为自 PostgreSQL 纪元（2000-01-01）起经过的微秒数。

Origin

Byte1('O')

将该消息标识为 Origin 消息。

Int64

源服务器上提交的 LSN。

String

复制源的名称。

注意，单个事务中可能有多条 Origin 消息。

Relation

Byte1('R')

将该消息标识为 Relation 消息。

Int32

关系的 ID。

String

命名空间（对于 pg_catalog 为空字符串）。

String

关系名称。

Int8

关系的复制标识设置（与 pg_class 中的 relreplident 相同）。

Int16

列数。

接下来，每一列都会出现以下消息部分：

Int8

列的标志。目前可以为 0，表示没有标志；或为 1，标记该列是键的一部分。

String

列的名称。

Int32

列的数据类型 ID。

Int32

列的类型修饰符（atttypmod）。

Type

Byte1('Y')

将该消息标识为 Type 消息。

Int32

数据类型的 ID。

String

命名空间（对于 pg_catalog 为空字符串）。

String

数据类型的名称。

Insert

Byte1('I')

将该消息标识为 Insert 消息。

Int32

关系的 ID，与 Relation 消息中的 ID 对应。

Byte1('N')

将后续 TupleData 消息标识为新元组。

TupleData

表示新元组内容的 TupleData 消息部分。

Update

Byte1('U')

将该消息标识为 Update 消息。

Int32

关系的 ID，与 Relation 消息中的 ID 对应。

Byte1('K')

将后续 TupleData 子消息标识为键。此字段可选，仅在更新更改了属于 REPLICA IDENTITY 索引的某一列或多列的数据时出现。

Byte1('O')

将后续 TupleData 子消息标识为旧元组。此字段可选，仅在发生更新的表将 REPLICA IDENTITY 设为 FULL 时出现。

TupleData

表示旧元组或主键内容的 TupleData 消息部分。仅在前面的'O'或'K'部分存在时出现。

Byte1('N')

将后续 TupleData 消息标识为新元组。

TupleData

表示新元组内容的 TupleData 消息部分。

Update 消息可以包含 'K' 消息部分或 'O' 消息部分，也可以两者都不包含，但绝不会同时包含两者。

Delete

Byte1('D')

将该消息标识为 Delete 消息。

Int32

关系的 ID，与 Relation 消息中的 ID 对应。

Byte1('K')

将后续 TupleData 子消息标识为键。如果发生删除的表使用索引作为 REPLICA IDENTITY，则出现此字段。

Byte1('O')

将后续 TupleData 消息标识为旧元组。如果发生删除的表将 REPLICA IDENTITY 设为 FULL，则出现此字段。

TupleData

表示旧元组或主键内容的 TupleData 消息部分，具体取决于前一个字段。

Delete 消息可以包含 'K' 消息部分或 'O' 消息部分，但绝不会同时包含两者。

Truncate

Byte1('T')

将该消息标识为 Truncate 消息。

Int32

关系数量

Int8

选项位用于 `TRUNCATE` 命令：1 表示 `CASCADE`，2 表示`RESTART IDENTITY`

Int32

关系的 ID，与 Relation 消息中的 ID 对应。每个关系都重复出现此字段。

以上消息共享以下消息部分。

TupleData

Int16

列数。

接下来，每一列都会出现以下子消息之一：

Byte1('n')

将该数据标识为 NULL 值。

或者

Byte1('u')

标识经过 TOAST 处理且未更改的值（不发送实际值）。

或者

Byte1('t')

将该数据标识为文本格式的值。

Int32

列值的长度。

Byte*n*

列值，采用文本格式。（未来版本可能支持其他格式。）*n* 为上述长度。

53.10. 自协议 2.0 以来的变更总结

本节提供一份简要的变更清单，供准备将现有客户端库更新到协议 3.0 的开发者参考。

初始启动包采用灵活的字符串列表格式，取代了固定格式。注意，运行时参数的会话默认值现在可以直接在启动包中指定。（实际上，以前也能通过 `options` 字段实现，但由于 `options` 的宽度有限，且无法用引号保护值中的空白字符，这种方法并不稳妥。）

现在，所有消息都在消息类型字节之后紧跟一个长度计数（启动包除外，它没有类型字节）。另请注意，PasswordMessage 现在也有类型字节。

ErrorResponse 和 NoticeResponse（'`E`' 和 '`N`'）消息现在包含多个字段，客户端代码可以利用这些字段组合出所需详细程度的错误消息。注意，各字段通常不会以换行符结束，而旧协议发送的单个字符串总是以换行符结束。

ReadyForQuery（'`Z`'）消息包含一个事务状态指示器。

BinaryRow 与 DataRow 消息类型不再有区别；单一的 DataRow 消息类型用于返回所有格式的数据。注意，DataRow 的布局已经改变，使其更容易解析。另外，二进制值的表示方式也已改变，不再直接取决于服务器内部的表示方式。

新增了“扩展查询”子协议，其中增加了前端消息类型 Parse、Bind、Execute、Describe、Close、Flush 和 Sync，以及后端消息类型 ParseComplete、BindComplete、PortalSuspended、ParameterDescription、NoData 和 CloseComplete。现有客户端不必关注这个子协议，但使用它可能有助于改善性能或功能。

`COPY` 数据现在封装在 CopyData 和 CopyDone 消息中。`COPY` 期间的错误恢复已有明确定义的方法。不再需要特殊的最后一行“`\.`”，`COPY OUT` 期间也不再发送它。（在 `COPY IN` 期间仍会将它识别为终止符，但这种用法已弃用，最终将被移除。）现已支持二进制 `COPY`。CopyInResponse 和 CopyOutResponse 消息包含了表示列数及各列格式的字段。

FunctionCall 和 FunctionCallResponse 消息的布局已经改变。FunctionCall 现在支持向函数传递 NULL 参数，也可以使用文本或二进制格式传递参数和取得结果。由于它不再提供对服务器内部数据表示的直接访问，因此也不再有理由将 FunctionCall 视为潜在的安全漏洞。

在连接启动期间，后端会为其认为客户端库关注的所有参数发送 ParameterStatus（'`S`'）消息。之后，只要这些参数中任何一个的当前值发生变化，就会发送一条 ParameterStatus 消息。

RowDescription（'`T`'）消息针对所描述行中的每一列，增加了表 OID 和列编号字段，并显示各列的格式代码。

后端不再生成 CursorResponse（'`P`'）消息。

NotificationResponse（'A'）消息增加了一个字符串字段，可以携带 NOTIFY 事件发送者传来的“载荷”字符串。

EmptyQueryResponse（'I'）消息以前包含一个空字符串参数，现在已将其移除。

第 54 章 PostgreSQL 编码约定

54.1. 格式

源代码格式使用 4 列制表宽度，并保留制表符（即，不会把制表符展开为空格）。每个逻辑缩进层级都对应再增加一个制表位。

布局规则（花括号位置等）遵循 BSD 约定。特别是，`if`、`while`、`switch` 等控制语句所对应代码块的花括号都单独占一行。

应限制行长度，使代码在 80 列窗口中可读。（这并不是说绝不能超过 80 列。例如，仅仅为了让代码保持在 80 列内而任意拆开一条很长的错误消息字符串，未必会提升可读性。）

不要使用 C++ 风格注释（`//` 注释）。严格的 ANSI C 编译器不接受这种注释。同样，也不要使用在块中途声明新变量这样的 C++ 扩展。

多行注释块的首选样式是

```
/*
 * comment text begins here
 * and continues here
 */
```

注意，从第 1 列开始的注释块会被 pgindent 原样保留，但它会把缩进的注释块当作普通文本重新排版。如果想保留缩进块中的换行，请像下面这样加上横线：

```
    /*----------
     * comment text begins here
     * and continues here
     *----------
     */
```

尽管提交的补丁并不绝对必须遵循这些格式规则，但这样做是个好主意。你的代码会在下一个版本发布前经过 pgindent 处理，因此按另一套格式约定把它写得再漂亮也没有意义。补丁的一个经验法则是“让新代码看起来像周围现有的代码”。

`src/tools` 目录中包含可供 emacs、xemacs 或 vim 编辑器使用的示例设置文件，以帮助确保它们按这些约定格式化代码。

文本浏览工具 more 和 less 可以这样调用：

```
more -x4
less -x4
```

以便正确显示制表符。

54.2. 在服务器内部报告错误

服务器代码内生成的错误、警告和日志消息应使用 `ereport`，或其更老的近亲 `elog` 来创建。这一函数的用法相当复杂，因此需要一些解释。

每条消息都必须包含两个元素：严重性级别（范围从 DEBUG 到 PANIC）以及主消息文本。此外还可以有可选元素，其中最常见的是遵循 SQL 规范 SQLSTATE 约定的错误标识符代码。ereport 本身只是一个包装函数，主要为了语法上的便利，使消息生成在 C 源代码中看起来像一次函数调用。ereport 唯一直接接受的参数是严重性级别。主消息文本以及任何可选消息元素都是通过在 ereport 调用中调用辅助函数（例如 errmsg）来生成的。

对 ereport 的一次典型调用可能如下：

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

这指定了错误严重性级别 ERROR（一种普通错误）。errcode 调用使用定义在 src/include/utils/errcodes.h 中的一个宏指定 SQLSTATE 错误代码。errmsg 调用提供主消息文本。注意辅助函数调用外面多了一层圆括号 – 它们虽然烦人，但在语法上是必需的。

这里有一个更复杂的示例：

```
ereport(ERROR,
        (errcode(ERRCODE_AMBIGUOUS_FUNCTION),
         errmsg("function %s is not unique",
                func_signature_string(funcname, nargs,
                                      NIL, actual_arg_types)),
         errhint("Unable to choose a best candidate function. "
                 "You might need to add explicit typecasts.")));
```

这展示了如何用格式代码把运行时值嵌入消息文本中。此外还提供了一条可选的“提示”消息。

如果严重性级别是 ERROR 或更高，ereport 会中止用户定义函数的执行，并且不会返回给调用者。如果严重性级别低于 ERROR，ereport 会正常返回。

适用于 ereport 的辅助例程有：

- errcode(sqlerrcode) 为该条件指定 SQLSTATE 错误标识符代码。如果不调用这个例程，则默认错误标识符在错误严重性级别为 ERROR 或更高时为 ERRCODE_INTERNAL_ERROR，在错误级别为 WARNING 时为 ERRCODE_WARNING，否则（对于 NOTICE 及以下）为 ERRCODE_SUCCESSFUL_COMPLETION。虽然这些默认值常常很方便，但在省略 errcode() 调用之前，始终要先想想它们是否合适。

- errmsg(const char *msg, ...) 指定主错误消息文本，以及可能要插入其中的运行时值。插入项通过 sprintf 风格的格式代码指定。除了 sprintf 接受的标准格式代码外，还可以使用格式代码 %m 插入 strerror 为 errno 当前值返回的错误消息。[1] %m 不需要在 errmsg 的参数列表中有任何对应项。注意，在处理格式代码之前，消息字符串会先经过 gettext 以便可能进行本地化。

- errmsg_internal(const char *msg, ...) 与 errmsg 相同，只是消息字符串不会被翻译，也不会被收入国际化消息字典。这应当用于那些“不可能发生”、大概不值得为之投入翻译精力的情况。

[1] 也就是说，是到达 ereport 调用点时的那个值；辅助报告例程内部对 errno 的更改不会影响它。如果显式写出 strerror(errno) 作为 errmsg 的参数列表内容，就不是这样了；因此不要这么做。

- `errmsg_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` 类似于 `errmsg`，但支持消息的各种复数形式。*`fmt_singular`* 是英文单数格式，*`fmt_plural`* 是英文复数格式，*`n`* 是决定需要哪种复数形式的整数值，其余参数按所选格式字符串进行格式化。更多信息见第 55.2.2 节。

- `errdetail(const char *msg, ...)` 提供一条可选的“详情”消息；当有额外信息但似乎不适合放在主消息中时，可使用它。消息字符串的处理方式与 `errmsg` 完全相同。

- `errdetail_internal(const char *msg, ...)` 与 `errdetail` 相同，只是消息字符串不会被翻译，也不会被收入国际化消息字典。这应当用于那些不值得投入翻译精力的详情消息，例如它们对大多数用户来说技术性太强而无甚用处。

- `errdetail_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` 类似于 `errdetail`，但支持消息的各种复数形式。更多信息见第 55.2.2 节。

- `errdetail_log(const char *msg, ...)` 与 `errdetail` 相同，只是该字符串只会写入服务器日志，绝不会发送给客户端。如果同时使用 `errdetail`（或其上面的某个等价函数）和 `errdetail_log`，那么一条字符串会发往客户端，另一条会发往日志。对于那些因安全性过于敏感或内容过于庞大而不适合放入发给客户端的报告中的错误细节，这很有用。

- `errdetail_log_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` 类似于 `errdetail_log`，但支持消息的各种复数形式。更多信息见第 55.2.2 节。

- `errhint(const char *msg, ...)` 提供一条可选的“提示”消息；它用于给出如何修复该问题的建议，而不是说明究竟哪里出了错的事实细节。消息字符串的处理方式与 `errmsg` 完全相同。

- `errcontext(const char *msg, ...)` 通常不会直接在 `ereport` 消息处调用；它用于 `error_context_stack` 回调函数中，提供错误发生时的上下文信息，例如 PL 函数中的当前位置。消息字符串的处理方式与 `errmsg` 完全相同。与其他辅助函数不同，它在每次 `ereport` 调用中都可以调用多次；这样提供的各个字符串会以换行分隔并连接起来。

- `errposition(int cursorpos)` 指定错误在查询字符串中的文本位置。目前，它只对在查询处理的词法和语法分析阶段检测到的错误有用。

- `errtable(Relation rel)` 指定一个关系，它的名称和模式名应作为辅助字段包含在错误报告中。

- `errtablecol(Relation rel, int attnum)` 指定一列，它的列名、表名和模式名应作为辅助字段包含在错误报告中。

- `errtableconstraint(Relation rel, const char *conname)` 指定一个表约束，它的名称、表名和模式名应作为辅助字段包含在错误报告中。为此目的，索引也应被视为约束，无论它们是否有关联的 `pg_constraint` 条目。请注意，作为 `rel` 传入的应是底层堆关系，而不是索引本身。

- `errdatatype(Oid datatypeOid)` 指定一个数据类型，它的名称和模式名应作为辅助字段包含在错误报告中。

- `errdomainconstraint(Oid datatypeOid, const char *conname)` 指定一个域约束，它的名称、域名和模式名应作为辅助字段包含在错误报告中。

- errcode_for_file_access() 是一个便捷函数，用于为与文件访问相关的系统调用失败选择合适的 SQLSTATE 错误标识符。它使用保存下来的 errno 来确定要生成哪种错误代码。通常应将它与主错误消息文本中的 %m 结合使用。

- errcode_for_socket_access() 是一个便捷函数，用于为与套接字相关的系统调用失败选择合适的 SQLSTATE 错误标识符。

- 可以调用 errhidestmt(bool hide_stmt) 来指定抑制 postmaster 日志中消息的 STATEMENT: 部分。一般来说，如果消息文本本身已经包含当前语句，这么做就是合适的。

- 可以调用 errhidecontext(bool hide_ctx) 来指定抑制 postmaster 日志中消息的 CONTEXT: 部分。这只应用于详细调试消息，因为在这类消息中反复包含上下文会让日志过于臃肿。

> **注意**
>
> 在一次 ereport 调用中，errtable、errtablecol、errtableconstraint、errdatatype 和 errdomainconstraint 这些函数中最多只能使用一个。这些函数的存在，是为了让应用能够提取与错误条件关联的数据库对象名称，而无需检查可能已经本地化的错误消息文本。这些函数应当用于应用很可能希望自动处理的错误报告。截至 PostgreSQL 9.3，只有 SQLSTATE 类 23（完整性约束违反）中的错误实现了完整覆盖，但未来很可能会扩展。

还有一个较旧的函数 elog，至今仍被大量使用。一个elog 调用：

```
elog(level, "format string", ...);
```

完全等价于：

```
ereport(level, (errmsg_internal("format string", ...)));
```

注意，SQLSTATE 错误代码总会取默认值，而且消息字符串不会被翻译。因此，elog 只应用于内部错误和低层调试日志。凡是普通用户可能感兴趣的消息，都应通过ereport。尽管如此，系统中仍有足够多的内部“不可能发生”错误检查，因此elog 依然被广泛使用；对这类消息来说，由于记法更简洁，它更受青睐。

关于如何编写良好的错误消息，可参见第 54.3 节。

54.3. 错误消息风格指南

提供这份风格指南，是希望在 PostgreSQL 生成的所有消息中保持一致、对用户友好的风格。

各部分写什么

主消息应简短、陈述事实，并避免提及具体函数名等实现细节。“简短”的意思是“在正常情况下应能放在一行里”。如果为了保持主消息简短，或者你觉得有必要提及特定失败的系统调用之类的实现细节，可使用详情消息。主消息和详情消息都应当陈述事实。提示消息则用于给出如何修复问题的建议，尤其是在该建议未必总是适用时。

例如，不要写：

```
IpcMemoryCreate: shmget(key=%d, size=%u, 0%o) failed: %m
(plus a long addendum that is basically a hint)
```

而要写：

```
Primary:    could not create shared memory segment: %m
Detail:     Failed syscall was shmget(key=%d, size=%u, 0%o).
Hint:       the addendum
```

原理：保持主消息简短，有助于使其切中要点，也便于客户端假定一行足以容纳错误消息，从而安排屏幕空间。详情消息和提示消息可以归入详细模式，或者放到弹出的错误详情窗口中。此外，为节省空间，详情消息和提示消息通常会从服务器日志中省略。最好避免提及实现细节，因为用户通常并不了解这些细节。

格式

不要在消息文本中写入任何特定的格式假设。应预期客户端和服务器日志会按照各自需要折行。对于较长的消息，可以使用换行字符（\n）来表示建议的分段。不要让消息以换行结束。不要使用制表符或其他格式控制字符。（在错误上下文显示中，会自动添加换行以分隔函数调用等上下文层次。）

原理：消息不一定显示在终端类设备上。在 GUI 显示或浏览器中，这些格式指令充其量也只是被忽略。

引号

英文文本在适合引用时应使用双引号。其他语言的文本应一致地使用一种引号，这种引号应符合该语言的出版习惯以及其他程序的计算机输出习惯。

原理：选择双引号而不是单引号多少有些武断，但通常更符合首选用法。有人建议按照 SQL 约定，根据对象类型来选择引号的种类（即字符串用单引号，标识符用双引号）。但这是语言内部的技术问题，很多用户甚至并不熟悉；它也无法扩展到其他类型的被引术语，更无法翻译到其他语言中，而且本身也没什么意义。

引号的用法

对文件名、用户提供的标识符以及其他可能包含单词的变量，始终使用引号界定。对于不会包含单词的变量（例如操作符名），则不要加引号。

后端中有些函数会在需要时自行给其输出加上双引号（例如 `format_type_be()`）。不要再对这类函数的输出额外加引号。

原理：对象的名称在嵌入消息时可能造成歧义。对于插入的名称从哪里开始、到哪里结束，要保持一致。但也不要用不必要或重复的引号把消息弄得杂乱无章。

语法和标点

主错误消息与详情/提示消息的规则不同：

主错误消息：首字母不要大写。消息末尾不要加句号。更不要想着在消息末尾加感叹号。

详情消息和提示消息：使用完整句子，并且每条都用句号结束。句子的第一个词要首字母大写。如果后面还有一句，在句号后放两个空格（针对英文文本；对其他语言可能并不合适）。

错误上下文字符串：首字母不要大写，字符串末尾不要加句号。上下文字符串通常不应是完整句子。

原理：避免使用标点，能让客户端应用更容易把消息嵌入各种语法上下文中。主消息往往本来也不是语法完整的句子。（而如果它们长到不止一句，就应当拆分成主消息和详情消息。）不过，详情消息和提示消息更长，也可能需要包含多个句子。为了保持一致，即便只有一个句子，它们也应遵循完整句子的风格。

大写与小写

消息措辞应使用小写，包括主错误消息的首字母。如果消息中出现 SQL 命令或关键字，则使用大写形式。

原理：这样更容易让所有消息看起来一致，因为有些消息是完整句子，而有些则不是。

避免被动语态

使用主动语态。有施动者时，用完整句子（“A could not do B”）。如果施动者就是程序本身，则使用无主语的电报式写法，但不要把程序写成“I”。

原理：程序不是人。不要假装它是。

现在时与过去时

如果一次尝试做某事失败了，但下次仍可能成功（也许在修复某个问题之后），就使用过去时。如果失败显然是永久性的，就使用现在时。

下面两种句式在语义上有明显差别：

```
could not open file "%s": %m
```

和：

```
cannot open file "%s"
```

第一种表示尝试打开文件失败了。消息应给出原因，例如“磁盘已满”或“文件不存在”。过去时更合适，因为下次磁盘可能就不满了，或者所请求的文件可能已经存在。

第二种形式表示，程序中根本不存在打开该命名文件的功能，或者从概念上就不可能。现在时更合适，因为这种情况会无限期持续下去。

原理：诚然，普通用户未必能仅凭消息的时态得出什么重要结论，但既然语言提供了语法，我们就应当正确使用它。

对象类型

在引用对象名称时，要说明它是什么类型的对象。

原理：否则没人会知道“foo.bar.baz”指的是什么。

括号

方括号只应用于：（1）命令概要中表示可选参数，或（2）表示数组下标。

原理：其他任何用法都不符合广为人知的习惯用法，而且会让人困惑。

组装错误消息

当消息中包含其他地方生成的文本时，应按以下方式嵌入：

```
could not open file %s: %m
```

原理：要把这种文本拼进一个流畅的单句中，同时照顾所有可能的错误代码会很困难，因此需要某种标点。也有人建议把嵌入的文本放在圆括号里，但如果嵌入的文本很可能是消息中最重要的部分（而这往往确实如此），那样就显得不自然了。

错误原因

消息总应说明错误发生的原因。例如：

```
BAD:    could not open file %s
BETTER: could not open file %s (I/O failure)
```

如果不知道原因，最好去修复代码。

函数名

不要在错误文本中包含报告该错误的例程名。需要时，我们有其他机制可以找出它，而且对大多数用户来说，这项信息没有帮助。如果去掉函数名后错误文本就不够清楚，请重新措辞。

```
BAD:    pg_atoi: error in "z": cannot parse "z"
BETTER: invalid input syntax for integer: "z"
```

也要避免提及被调用函数的名称；应改为说明代码想要做什么：

```
BAD:    open() failed: %m
BETTER: could not open file %s: %m
```

如果确实有必要，可在详情消息中提及系统调用。（在某些情况下，在详情消息中提供传给该系统调用的实际值也许是合适的信息。）

原理：用户并不知道那些函数都做了什么。

应避免的词语

Unable. “Unable” 几乎就是被动语态。更好的做法是视情况使用 “cannot” 或 “could not” 。

Bad. 像 “bad result” 这样的错误消息，实在很难被合理理解。最好写出结果为什么是 “bad” ，例如 “invalid format” 。

Illegal. “Illegal” 指的是违法，其余情况应使用 “invalid” 。更好的是，直接说明为什么无效。

Unknown. 尽量避免“unknown”。考虑“error: unknown response”。如果你都不知道响应是什么，又如何知道它有错？“Unrecognized”往往是更好的选择。此外，一定要包含被抱怨的值。

```
BAD:    unknown node type
BETTER: unrecognized node type: 42
```

Find vs. Exists. 如果程序为了定位某个资源使用了非平凡算法（例如路径搜索），而该算法失败了，那么说程序没能“find”该资源是公平的。另一方面，如果资源的预期位置已知，但程序无法在那里访问它，那么就应说该资源并不“exist”。在这种情况下使用“find”显得含糊，也会混淆问题。

May、Can 与 Might. “May”暗示许可（例如，"You may borrow my rake."），在文档或错误消息中用途不大。“Can”暗示能力（例如，"I can lift that log."），而“might”暗示可能性（例如，"It might rain today."）。使用恰当的词能澄清含义，也有助于翻译。

缩略形式. 避免使用缩略形式，例如“can't”；应改用“cannot”。

Non-negative. 避免使用“non-negative”，因为它是否接受零有歧义。更好的是使用“greater than zero”或“greater than or equal to zero”。

完整拼写

单词应完整拼写。例如，避免使用：

- spec
- stats
- parens
- auth
- xact

原理：这会提升一致性。

本地化

请记住，错误消息文本需要被翻译成其他语言。请遵循第 55.2.2 节中的指导，避免给翻译者制造麻烦。

54.4. 其他编码约定

C 标准

PostgreSQL 中的代码应只依赖 C89 标准提供的语言特性。这意味着，至少除去少数依赖平台的部分，符合 C89 标准的编译器必须能够编译 postgres。如果提供了回退方案，则可以使用来自 C 标准后续修订的特性或编译器特定特性。

例如，目前使用了 `static inline` 和 `_Static_assert()`，尽管它们来自 C 标准的较新修订。如果这些特性不可用，我们会分别回退为定义不带 inline 的函数，以及使用一种兼容 C89 的替代方案，后者执行相同的检查，但输出的消息比较晦涩。

类函数宏和内联函数

带参数的宏和 `static inline` 函数都可以使用。如果把某段代码写成宏会有多次求值风险，那么后者更可取，例如下面这种情况：

```
#define Max(x, y)       ((x) > (y) ? (x) : (y))
```

或者当宏会变得非常长时也是如此。在其他情况下，只能使用宏，或者至少使用宏更容易。例如，需要向宏传递各种不同类型的表达式时就是如此。

当某个内联函数的定义引用了只在后端可用的符号（即变量、函数）时，该函数在前端代码中被包含时就不应可见。

```
#ifndef FRONTEND
static inline MemoryContext
MemoryContextSwitchTo(MemoryContext context)
{
    MemoryContext old = CurrentMemoryContext;

    CurrentMemoryContext = context;
    return old;
}
#endif   /* FRONTEND */
```

在这个示例中，只在后端中可用的 `CurrentMemoryContext` 被引用了，因此该函数用 `#ifndef FRONTEND` 隐藏起来。这条规则之所以存在，是因为有些编译器即使函数未被使用，也会为内联函数中包含的符号发出引用。

编写信号处理器

要让代码适合在信号处理器中运行，必须非常谨慎地编写。根本问题在于，只要没有被阻塞，信号处理器就能在任何时刻中断代码。如果信号处理器中的代码使用了与外部代码相同的状态，就可能引发混乱。举例来说，想想如果信号处理器试图获取一个已经被中断代码持有的锁，会发生什么。

除非有特殊安排，信号处理器中的代码只能调用异步信号安全函数（按 POSIX 的定义），并且只能访问类型为 `volatile sig_atomic_t` 的变量。`postgres` 中也有少数函数被认为是信号安全的，其中尤其重要的是 `SetLatch()`。

在大多数情况下，信号处理器只应记录信号已经到达，并使用锁存器唤醒在信号处理器之外运行的代码。下面就是这样的一个处理器示例：

```
static void
handle_sighup(SIGNAL_ARGS)
{
    int         save_errno = errno;

    got_SIGHUP = true;
    SetLatch(MyLatch);

    errno = save_errno;
}
```

`errno` 被保存并恢复，是因为`SetLatch()` 可能会改变它。如果不这样做，当前正在检查 `errno` 的被中断代码可能会看到错误的值。

调用函数指针

为了清晰起见，当函数指针是一个简单变量时，调用其指向的函数时最好显式解引用，例如：

```
(*emit_log_hook) (edata);
```

（尽管 `emit_log_hook(edata)` 也能工作）。当函数指针是某个结构体的一部分时，这些额外符号可以，而且通常也应该省略，例如：

```
paramInfo->paramFetch(paramInfo, paramId);
```

第 55 章 本地语言支持

55.1. 给翻译者

PostgreSQL 程序（服务器和客户端）可以用你所偏好的语言发出消息 – 前提是这些消息已经被翻译。创建和维护翻译后的消息集，需要那些精通自己语言并愿意为 PostgreSQL 项目作出贡献的人的帮助。做这件事完全不必是程序员。本节说明如何提供帮助。

55.1.1. 要求

我们不会评判你的语言能力 – 本节讨论的是软件工具。理论上，你只需要一个文本编辑器。但这只有在你不打算试用自己翻译的消息时才成立，而这种情况不太可能发生。配置你的源代码树时，请务必使用 `--enable-nls` 选项。这还会检查 libintl 库和 `msgfmt` 程序，而终端用户无论如何也需要它们。要试用你的工作成果，请遵循安装说明中适用的部分。

如果你想开始一项新的翻译工作，或者想执行一次消息目录合并（后文会描述），则分别需要 GNU 兼容实现的 `xgettext` 和 `msgmerge` 程序。今后我们会尽量做到：如果你使用打包的源码发布版，就不需要 `xgettext` 了。（如果从 Git 工作，你仍然需要它。）目前推荐使用 GNU Gettext 0.10.36 或更高版本。

你本地的 gettext 实现应该自带文档。其中有些内容可能与下文重复，但要了解更多细节，你应当去查阅那些文档。

55.1.2. 概念

原始（英语）消息及其（可能存在的）译文对应项，以成对形式保存在消息目录中；每个程序以及每种目标语言各有一份消息目录（尽管相关程序可以共享同一个消息目录）。消息目录有两种文件格式：第一种是“PO”文件（Portable Object，可移植对象），它是带有特殊语法的纯文本文件，由翻译者编辑。第二种是“MO”文件（Machine Object，机器对象），它是根据相应的 PO 文件生成的二进制文件，在国际化后的程序运行时使用。翻译者不需要处理 MO 文件；事实上几乎没有人需要。

消息目录文件的扩展名，毫不意外地是 `.po` 或 `.mo`。基本名则视情况而定，要么是其所对应的程序名，要么是该文件所对应的语言。这多少有些令人困惑。例子包括 `psql.po`（psql 的 PO 文件）或 `fr.mo`（法语的 MO 文件）。

PO 文件的格式示例如下：

```
# comment

msgid "original string"
msgstr "translated string"

msgid "more original"
msgstr "another translated"
"string can be broken up like this"

...
```

msgid 行是从程序源代码中提取出来的。（并非一定如此，但这是最常见的做法。）msgstr 行起初为空，由翻译者填入有意义的字符串。字符串可以包含 C 风格的转义字符，也可以像上例那样跨多行延续。（下一行必须从行首开始。）

字符引入注释。如果 # 字符后面紧跟空白，那么这就是由翻译者维护的注释。也可能存在自动注释，即 # 后紧跟非空白字符的注释。这些注释由各种处理 PO 文件的工具维护，目的是帮助翻译者。

```
#. automatic comment
#: filename.c:1023
#, flags, flags
```

#. 风格的注释是从使用该消息的源文件中提取出来的。程序员可能已经为翻译者插入了信息，例如期望的对齐方式。#: 注释指出该消息在源代码中使用的精确位置。翻译者不必查看程序源代码，但如果对正确译法有疑问，也可以去看。#, 注释包含以某种方式描述该消息的标志。目前有两种标志：如果某条消息可能因程序源代码的更改而过期，则会设置 `fuzzy`。翻译者随后可以核实这一点，并在可能时移除 fuzzy 标志。注意，fuzzy 消息不会提供给终端用户。另一个标志是 `c-format`，它表示该消息是一个 `printf` 风格的格式模板。这意味着译文也应该是一个格式字符串，并具有相同数量和类型的占位符。有一些工具会据此验证格式字符串，它们就是根据 c-format 标志来检查的。

55.1.3. 创建和维护消息目录

好，那么该如何创建一个“空白”消息目录呢？首先，进入包含你想翻译其消息的程序的目录。如果那里有一个 `nls.mk` 文件，就说明这个程序已经为翻译做好准备了。

如果已经有一些 `.po` 文件，那么就说明已经有人做过部分翻译工作。这些文件命名为 *`language`*`.po`，其中 *`language`* 是 ISO 639-1 两字母语言代码（小写）[1]，例如法语对应 `fr.po`。如果某种语言确实需要进行多套翻译工作，这些文件也可以命名为 *`language_region`*`.po`，其中 *`region`* 是 ISO 3166-1 两字母国家代码（大写）[2]，例如巴西葡萄牙语对应 `pt_BR.po`。如果你找到了自己想要的语言，就可以直接开始编辑那个文件。

如果你需要开始一项新的翻译工作，先运行以下命令：

```
make init-po
```

这会创建一个文件 *`progname`*`.pot`。（`.pot` 用来与已经“投入使用”的 PO 文件区分开来；`T` 代表“template”。）把这个文件复制为 *`language`*`.po` 并进行编辑。为了让系统知道新语言已经可用，还要编辑文件`nls.mk`，并在类似下面的行中添加该语言（或语言加国家）代码：

```
AVAIL_LANGUAGES := de fr
```

（当然也可能有其他语言。）

随着底层程序或库发生变化，程序员可能会修改或增加消息。这种情况下你不需要从头开始。相反，请运行以下命令：

```
make update-po
```

它会创建一个新的空白消息目录文件（也就是你最初使用的那个 pot 文件），并将它与现有的 PO 文件合并。如果合并算法对某条消息拿不准，就会像上文所述那样把它标记为“fuzzy”。新的 PO 文件会以 `.po.new` 作为扩展名保存。

[1] https://www.loc.gov/standards/iso639-2/php/English_list.php
[2] https://www.iso.org/iso-3166-country-codes.html

55.1.4. 编辑 PO 文件

PO 文件可以用普通文本编辑器编辑。翻译者只应修改 msgstr 指令后引号之间的内容、添加注释，以及调整 fuzzy 标志。Emacs 里（不出意外地）有一个 PO 模式，相当有用。

PO 文件不必完全填满。如果没有可用译文（或译文为空），软件会自动回退到原始字符串。把不完整的翻译提交到源码树中也没有问题；这样别人就有机会接手你的工作。不过，完成一次合并之后，建议优先清除 fuzzy 条目。记住，fuzzy 条目不会被安装；它们只用来作为可能正确译文的参考。

编辑译文时要记住以下几点：

- 务必确保如果原文以换行结束，译文也同样以换行结束。制表符等也是如此。
- 如果原文是一个 `printf` 格式字符串，译文也必须如此。译文还需要按相同顺序保留相同的格式说明符。有时语言自身的规则会让这一点变得不可能，或者至少很别扭。在这种情况下，你可以像下面这样修改格式说明符：

  ```
  msgstr "Die Datei %2$s hat %1$u Zeichen."
  ```

  这样一来，第一个占位符实际上会使用参数列表中的第二个参数。*`digits`*`$` 必须紧跟在 % 后面，位于其他任何格式修饰符之前。（这个特性确实存在于 `printf` 函数族中。你以前可能没听说过它，因为在消息国际化之外，它几乎没有用武之地。）
- 如果原始字符串包含语言错误，就报告它（或者你自己在程序源代码中修正它），然后照常翻译。等程序源代码更新之后，修正过的字符串就可以被合并进来。如果原始字符串包含事实性错误，就报告它（或者你自己修正它），并且不要翻译它。相反，你可以在 PO 文件里为该字符串加上一条注释。
- 保持原始字符串的风格和语气。特别是，那些不是完整句子的消息（`cannot open file %s`）通常不应以大写字母开头（如果你的语言区分大小写），也不应以句号结尾（如果你的语言使用标点符号）。阅读第 54.3 节可能会有帮助。
- 如果你不知道某条消息是什么意思，或者它有歧义，就到开发者邮件列表里提问。很可能说英语的终端用户也同样看不懂它，或者会觉得它有歧义，因此最好直接改进那条消息。

55.2. 给程序员

55.2.1. 实现机制

本节描述如何在 PostgreSQL 发行版中的程序或库里实现本地语言支持。目前它只适用于 C 程序。

为程序添加 NLS 支持

1. 把下面的代码插入到该程序的启动序列中：

```
#ifdef ENABLE_NLS
#include <locale.h>
#endif

...

#ifdef ENABLE_NLS
setlocale(LC_ALL, "");
bindtextdomain("progname", LOCALEDIR);
textdomain("progname");
#endif
```

（其中 *progname* 实际上可以自由选择。）

2. 凡是遇到可能需要翻译的消息，都需要插入对 `gettext()` 的调用。例如：

```
fprintf(stderr, "panic level %d\n", lvl);
```

将改成：

```
fprintf(stderr, gettext("panic level %d\n"), lvl);
```

（如果没有配置 NLS 支持，`gettext` 会被定义成一个空操作。）

这样往往会增加很多杂乱内容。一个常用的简写是：

```
#define _(x) gettext(x)
```

如果该程序的大部分通信都是通过一个或少数几个函数完成的，例如后端中的 `ereport()`，另一种可行的解决办法是让这个函数在内部对所有输入字符串调用 `gettext`。

3. 在程序源代码所在目录中添加一个 `nls.mk` 文件。这个文件会作为 makefile 读取。这里需要设置以下变量：

`CATALOG_NAME`

程序名，即 `textdomain()` 调用中提供的名称。

`AVAIL_LANGUAGES`

已提供的翻译列表 – 最初为空。

`GETTEXT_FILES`

包含可翻译字符串的文件列表，也就是那些使用 `gettext` 或其他替代方案标记过的文件。最终，这会包含该程序几乎所有的源文件。如果这个列表太长，可以让第一个“文件”是一个 `+`，第二个词则是一个文件名，该文件每行包含一个文件名。

GETTEXT_TRIGGERS

为翻译者生成消息目录的工具需要知道哪些函数调用包含可翻译字符串。默认只认识 `gettext()` 调用。如果你使用了 `_` 或其他标识符，就需要在这里列出它们。如果可翻译字符串不是第一个参数，条目就需要写成 `func:2` 这样的形式（表示第二个参数）。如果你有一个支持复数形式消息的函数，条目应写成 `func:1,2` 这样（分别标识单数和复数消息参数）。

构建系统会自动处理消息目录的构建和安装。

55.2.2. 消息编写指南

下面列出一些便于翻译的消息编写指南。

- 不要像下面这样在运行时拼接句子：

```
printf("Files were %s.\n", flag ? "copied" : "removed");
```

句子中的词序在其他语言里可能完全不同。另外，即使你记得对每个片段都调用 `gettext()`，这些片段分开来看也可能难以妥善翻译。最好稍微重复一点代码，让每条待翻译消息都成为一个语义完整的整体。只有数字、文件名之类的运行时变量才应该在运行时插入消息文本中。

- 出于类似的原因，下面这样也行不通：

```
printf("copied %d file%s", n, n!=1 ? "s" : "");
```

因为它假定了复数形式的构造方式。如果你以为可以这样解决：

```
if (n==1)
    printf("copied 1 file");
else
    printf("copied %d files", n):
```

那就会失望了。有些语言的复数形式不止两种，而且规则相当特殊。通常最好从设计上彻底避开这个问题，例如这样：

```
printf("number of copied files: %d", n);
```

如果你确实想构造一条正确处理复数的消息，也有相应支持，只是有点别扭。在 `ereport()` 中生成主错误消息或详细错误消息时，可以写成这样：

```
errmsg_plural("copied %d file",
              "copied %d files",
              n,
              n)
```

第一个参数是适用于英语单数形式的格式字符串，第二个参数是适用于英语复数形式的格式字符串，第三个参数是决定使用哪种复数形式的整数控制值。后续参数会像往常一样按照格式字符串进行格式化。（通常，这个复数控制值本身也会是要格式化的值之一，所以必须写两次。）在英语中，只需要区分 *n* 是否为 1；而在其他语言中，可能存在许多不同的复数形式。翻译者会把这两个英语形式视为一组，并有机会提供多个替代字符串，再根据运行时的 *n* 值选择合适的那个。

如果你需要对一条不是直接用于 `errmsg` 或 `errdetail` 报告的消息做复数处理，就必须使用底层函数 `ngettext`。参见 gettext 文档。

- 如果你想向翻译者传达某些信息，例如一条消息应如何与其他输出对齐，就在该字符串出现之前放置一个以 `translator` 开头的注释，例如：

  ```
  /* translator: This message is not what it seems to be. */
  ```

  这些注释会被复制到消息目录文件中，这样翻译者就能看到它们。

第 56 章 编写过程语言调用处理器

凡是不是使用当前针对编译型语言的“版本 1”接口编写的函数，在被调用时都会经过该语言专用的调用处理器函数。这包括用户定义过程语言中的函数，以及用 SQL 编写的函数。调用处理器负责以恰当的方式执行该函数，例如解释所提供的源文本。本章概述如何编写新的过程语言调用处理器。

过程语言的调用处理器是一个“普通”函数，必须使用诸如 C 这样的编译型语言、按照版本 1 接口编写，并在 PostgreSQL 中注册为不接受参数且返回 `language_handler` 类型。这个特殊伪类型会将该函数标识为调用处理器，并阻止它在 SQL 命令中被直接调用。有关 C 语言调用约定和动态装载的更多细节，见第 38.10 节。

调用处理器与其他函数的调用方式相同：它接收一个指向 `FunctionCallInfoData struct` 的指针，其中包含参数值和被调用函数的信息，并应返回一个 `Datum` 结果（如果要返回 SQL 空值，还需要设置 `FunctionCallInfoData` 结构体的 `isnull` 字段）。调用处理器与普通被调用函数的区别在于，`FunctionCallInfoData` 结构体的 `flinfo->fn_oid` 字段包含的是实际要调用的函数的 OID，而不是调用处理器自身的 OID。调用处理器必须使用该字段确定应执行哪个函数。此外，传入的参数列表也是按照目标函数的声明设置的，而不是按照调用处理器的声明。

调用处理器必须从系统目录 `pg_proc` 中取出该函数的条目，并分析被调用函数的参数类型和返回类型。该函数的 `CREATE FUNCTION` 命令中的 `AS` 子句内容，保存在 `pg_proc` 对应行的 `prosrc` 列中。这里通常是一段过程语言源文本，但理论上也可以是其他内容，例如某个文件的路径名，或者任何能详细告诉调用处理器该做什么的信息。

同一个函数在执行一条 SQL 语句期间往往会被调用很多次。调用处理器可以利用 `flinfo->fn_extra` 字段，避免重复查找被调用函数的信息。该字段起初为 `NULL`，但调用处理器可以把它设置为指向与被调用函数有关的信息。在后续调用中，如果 `flinfo->fn_extra` 已经不是 `NULL`，就可以直接使用它并跳过信息查找步骤。调用处理器必须确保 `flinfo->fn_extra` 指向的内存至少能存活到当前查询结束，因为 `FmgrInfo` 数据结构可能会保留这么久。一种做法是在 `flinfo->fn_mcxt` 指定的内存上下文中分配这些额外数据；这类数据通常会与 `FmgrInfo` 本身具有相同的生命周期。不过，处理器也可以选择使用生命周期更长的内存上下文，以便跨查询缓存函数定义信息。

过程语言函数作为触发器调用时，不会按通常方式传入参数，但 `FunctionCallInfoData` 的 `context` 字段会指向一个 `TriggerData` 结构体，而不是像普通函数调用那样为 `NULL`。语言调用处理器应提供让过程语言函数获取触发器信息的机制。

下面是用 C 编写的过程语言处理器模板：

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * 作为触发器函数调用
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        retval = ...
    }
    else
    {
        /*
         * 作为函数调用
         */

        retval = ...
    }

    return retval;
}
```

只需在省略号处添加几千行代码，就能完成这个调用处理器。

将处理器函数编译为可加载模块后（参见第 38.10.5 节），可用以下命令注册示例过程语言：

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'filename'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

虽然仅提供调用处理器就足以创建一种最小可用的过程语言，但还可以额外提供另外两个函数，以便让该语言更易于使用。它们是验证器和内联处理器。可以提供验证器，以便在CREATE FUNCTION 期间执行与该语言相关的检查。也可以提供内联处理器，以便让该语言支持通过DO 命令执行匿名代码块。

如果过程语言提供了验证器，它必须声明为一个接受单个 `oid` 参数的函数。验证器的返回值会被忽略，因此习惯上将其声明为返回 `void`。在 `CREATE FUNCTION` 命令创建或更新了一个用该过程语言编写的函数之后，验证器会在该命令结束时被调用。传入的 OID 是该函数在 `pg_proc` 中对应行的 OID。验证器必须按通常方式取出这一行，并执行适当的检查。首先，应调用 `CheckFunctionValidatorAccess()`，以诊断那些用户无法通过 `CREATE FUNCTION` 实现的对验证器的显式调用。典型检查包括确认该语言支持该函数的参数类型和结果类型，以及确认函数体在该语言中语法正确。如果验证器认为该函数没有问题，就应直接返回；如果发现错误，则应通过常规的 `ereport()` 错误报告机制加以报告。抛出错误会强制事务回滚，从而阻止错误的函数定义被提交。

验证器函数通常应遵循 check_function_bodies 参数：如果它被关闭，就应跳过任何开销高昂或依赖上下文的检查。如果该语言允许在编译时执行代码，验证器还必须抑制会诱发此类执行的检查。特别是，pg_dump 会关闭这个参数，这样它就可以装载过程语言函数，而不必担心副作用或函数体对其他数据库对象的依赖关系。（正因为有这一要求，调用处理器不应假设验证器已经完整检查了函数。提供验证器的目的，并不是让调用处理器省略检查，而是在 `CREATE FUNCTION` 命令中存在明显错误时立即通知用户。）虽然究竟检查哪些内容主要由验证器函数自行决定，但要注意，只有在 `check_function_bodies` 打开时，核心 `CREATE FUNCTION` 代码才会执行函数附带的 `SET` 子句。因此，当 `check_function_bodies` 关闭时，那些结果可能受 GUC 参数影响的检查肯定都应跳过，以避免在恢复转储时出现虚假的失败。

如果过程语言提供了内联处理器，它必须声明为一个接受单个 `internal` 参数的函数。内联处理器的返回值会被忽略，因此习惯上将其声明为返回 `void`。当执行指定了该过程语言的 `DO` 语句时，就会调用内联处理器。实际传入的参数是一个指向 `InlineCodeBlock` 结构体的指针，其中包含 `DO` 语句参数的相关信息，特别是待执行的匿名代码块文本。内联处理器应执行这段代码并返回。

建议你将以上所有函数声明以及 `CREATE LANGUAGE` 命令本身都封装进一个扩展中，这样只需执行一条简单的 `CREATE EXTENSION` 命令就足以安装该语言。关于如何编写扩展，见第 38.16 节。

标准发行版中附带的各过程语言，在尝试编写自己的语言调用处理器时都是很好的参考资料。请查看源码树中的 `src/pl` 子目录。CREATE LANGUAGE 参考页中也包含一些有用的细节。

第 57 章 编写外部数据包装器

所有在一个外部表上的操作都通过它的外部数据包装器来处理，外部数据包装器由一组被核心服务器调用的函数组成。外部数据包装器负责从远程数据源取得数据并把它返回给PostgreSQL 执行器。如果要支持更新外部表，包装器也需要处理更新。本章将介绍如何编写一个新的外部数据包装器。

标准发行版中包含的外部数据包装器，是编写你自己的外部数据包装器时很好的参考。请查看源码树中的`contrib` 子目录。CREATE FOREIGN DATA WRAPPER 参考页中也有一些有用的细节。

> **注意**
>
> SQL 标准规定了一个用于编写外部数据包装器的接口。但是，PostgreSQL 没有实现该 API，因为要让 PostgreSQL 适配它需要大量工作，而且该标准 API 也没有得到广泛采用。

57.1. 外部数据包装器函数

FDW 作者需要实现一个处理器函数，并且可以选择实现一个验证器函数。这两个函数都必须使用 C 之类的编译型语言编写，并采用版本 1 接口。关于 C 语言调用约定和动态加载的细节，请参阅第 38.10 节。

处理器函数只是返回一个结构体，其中包含供规划器、执行器和各种维护命令调用的回调函数指针。编写 FDW 的大部分工作都在于实现这些回调函数。处理器函数必须在PostgreSQL 中注册为不带参数，并返回特殊的伪类型`fdw_handler`。这些回调函数都是普通的 C 函数，在 SQL 层既不可见也不可调用。回调函数见第 57.2 节。

验证器函数负责验证 `CREATE` 和 `ALTER` 命令中指定的选项，这些命令可以针对其外部数据包装器，也可以针对使用该包装器的外部服务器、用户映射和外部表。注册验证器函数时，必须声明它接受两个参数：一个 text 数组，包含待验证的选项；以及一个 OID，表示选项所属对象的类型（使用存放该对象的系统目录的 OID，即 `ForeignDataWrapperRelationId`、`ForeignServerRelationId`、`UserMappingRelationId` 或 `ForeignTableRelationId`）。如果未提供验证器函数，创建或修改对象时就不会检查选项。

57.2. 外部数据包装器回调例程

FDW 处理器函数返回一个通过 palloc 分配的 `FdwRoutine` 结构体，其中包含下文描述的回调函数指针。与扫描相关的函数是必需的，其余则是可选的。

`FdwRoutine` 结构体类型声明在 `src/include/foreign/fdwapi.h` 中，其中还有更多细节。

57.2.1. 扫描外部表的 FDW 例程

```
void
GetForeignRelSize(PlannerInfo *root,
                  RelOptInfo *baserel,
                  Oid foreigntableid);
```

获取外部表的关系大小估计。该函数会在规划扫描外部表的查询之初被调用。root 是规划器关于该查询的全局信息；baserel 是规划器关于该表的信息；foreigntableid 是该外部表在 pg_class 中的 OID。（foreigntableid 也可以从规划器数据结构中取得，但显式传入可以省一些工作。）

此函数应更新 baserel->rows，使其表示考虑限制条件过滤后，表扫描预计返回的行数。baserel->rows 的初始值只是一个固定的默认估计，应尽可能替换。如果能更准确地估计结果行的平均宽度，该函数也可以更新 baserel->width。（初始值基于列的数据类型，以及上次 ANALYZE 测得的列平均宽度。）此外，如果能更准确地估计外部表的总行数，该函数可以更新 baserel->tuples。（初始值来自 pg_class.reltuples，表示上次 ANALYZE 所见的总行数。）

更多信息请见第 57.4 节。

```
void
GetForeignPaths(PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid);
```

为一个外部表上的扫描创建可能的访问路径。这个函数在查询规划过程中被调用。参数和 GetForeignRelSize 相同，后者已经被调用过了。

这个函数必须为外部表上的扫描生成至少一个访问路径（ForeignPath 节点），并且必须调用add_path 把每一个这样的路径加入到 baserel->pathlist 中。推荐使用 create_foreignscan_path 来构造 ForeignPath 节点。该函数可以生成多个访问路径，例如一个具有合法pathkeys 的路径可以表示预排序的结果。每个访问路径都必须包含代价估计，并且可以包含用于标识预期扫描方法的任意 FDW 私有信息。

更多信息请见第 57.4 节。

```
ForeignScan *
GetForeignPlan(PlannerInfo *root,
               RelOptInfo *baserel,
               Oid foreigntableid,
               ForeignPath *best_path,
               List *tlist,
               List *scan_clauses,
               Plan *outer_plan);
```

根据选中的外部访问路径创建一个 ForeignScan 计划节点。这个函数在查询规划结束时被调用。其参数与 GetForeignRelSize 相同，外加所选中的 ForeignPath（此前由 GetForeignPaths、GetForeignJoinPaths 或 GetForeignUpperPaths 生成）、计划节点要输出的目标列表、计划节点要强制执行的限制子句，以及 ForeignScan 的外层子计划，该子计划用于 RecheckForeignScan 执行重检查。（如果该路径针对的是连接而不是基本关系，foreigntableid 将是 InvalidOid。）

这个函数必须创建并返回一个 ForeignScan 计划节点，推荐使用make_foreignscan 来构造该 ForeignScan 节点。

更多信息请见第 57.4 节。

```
void
BeginForeignScan(ForeignScanState *node,
                 int eflags);
```

开始执行外部扫描。这个函数在执行器启动期间被调用。它应执行扫描开始前所需的任何初始化工作，但不应启动实际扫描（那会在第一次调用IterateForeignScan 时进行）。ForeignScanState 节点已经创建，但其fdw_state 字段仍为 NULL。关于待扫描表的信息可通过 ForeignScanState 节点访问，特别是底层的ForeignScan 计划节点，其中包含了由 GetForeignPlan 提供的任何 FDW 私有信息。eflags 包含描述执行器对该计划节点操作模式的标志位。

注意，当(eflags & EXEC_FLAG_EXPLAIN_ONLY) 为真时，这个函数不应执行任何外部可见的动作；它只应做使节点状态对ExplainForeignScan 和 EndForeignScan 有效所需的最少工作。

```
TupleTableSlot *
IterateForeignScan(ForeignScanState *node);
```

从外部数据源取回一行，并将其放入一个元组表槽中返回（应当使用该节点的 ScanTupleSlot 来完成这项工作）。如果没有更多可用的行，则返回 NULL。元组表槽基础设施允许返回物理元组或虚拟元组；在大多数情况下，出于性能考虑，后者更可取。注意，此函数是在一个短生命周期的内存上下文中调用的，该上下文会在两次调用之间被重置。如果需要更长寿命的存储，请在 BeginForeignScan 中创建内存上下文，或者使用该节点的 es_query_cxt（位于 EState 中）。

如果提供了 fdw_scan_tlist 目标列表，则返回的行必须与之匹配；否则，它们必须匹配被扫描外部表的行类型。如果选择优化掉不需要取回的列，则应当在这些列的位置上填入空值，或者生成一个省略这些列的fdw_scan_tlist 列表。

注意 PostgreSQL 的执行器并不在乎被返回的行是否违背了定义在该外部表上的任何约束 – 但是规划器会在乎这一点，并且如果在外部表中有可见行不满足一个约束，规划器可能会错误地优化

查询。如果当用户已经声明一个约束应该为真时它却被违背，产生一个错误可能比较合适（就像在数据类型失配的情况下所作的那样）。

```
void
ReScanForeignScan(ForeignScanState *node);
```

从头开始重启一个扫描。注意扫描所依赖的任何参数可能已经改变了值，因此新扫描不一定会返回完全相同的行。

```
void
EndForeignScan(ForeignScanState *node);
```

结束扫描并释放资源。通常无需释放用 palloc 分配的内存，但例如打开的文件和到远程服务器的连接应当清理。

57.2.2. 扫描外部连接的 FDW 例程

如果一个 FDW 支持远程执行外部表连接（而不是先取回两个表的数据再在本地执行连接），它应当提供这个回调函数：

```
void
GetForeignJoinPaths(PlannerInfo *root,
                    RelOptInfo *joinrel,
                    RelOptInfo *outerrel,
                    RelOptInfo *innerrel,
                    JoinType jointype,
                    JoinPathExtraData *extra);
```

为属于同一外部服务器的两个或多个外部表的连接创建可能的访问路径。此可选函数在查询规划期间调用。与 `GetForeignPaths` 一样，此函数应生成 `ForeignPath` 路径，针对给定的 `joinrel`，并调用 `add_path`，将这些路径加入该连接的候选路径集合。但与 `GetForeignPaths` 不同，此函数不必保证至少创建一条路径，因为始终可以采用本地连接的路径。

注意，对同一个连接关系，会以不同的内外关系组合反复调用这个函数。FDW 应负责尽量减少重复工作。

如果某条 `ForeignPath` 路径被选中用于该连接，它就表示整个连接过程；为组成该连接的各表及其子连接生成的路径将不会再使用。之后对该连接路径的处理，和处理扫描单个外部表的路径大体相同。一个区别是，所得 `ForeignScan` 计划节点的 `scanrelid` 应设为零，因为它不代表某个单一关系；相反，`ForeignScan` 节点的 `fs_relids` 字段表示被连接的关系集合。（后者由核心规划器代码自动设置，FDW 无需填充。）另一个区别是，由于远程连接的列列表无法从系统目录中找到，FDW 必须用适当的 `TargetEntry` 节点列表填充 `fdw_scan_tlist`，表示它在运行时会在返回的元组中提供哪些列。

更多信息请见第 57.4 节。

57.2.3. 规划扫描或连接之后处理的 FDW 例程

如果一个 FDW 支持执行远程的扫描/连接后处理，例如远程聚合，那么它应该提供这个回调函数：

```
void
GetForeignUpperPaths(PlannerInfo *root,
                     UpperRelationKind stage,
                     RelOptInfo *input_rel,
                     RelOptInfo *output_rel,
                     void *extra);
```

创建可能的访问路径，用于上层关系处理；这是规划器对扫描或连接之后的所有查询处理步骤的称呼，包括聚合、窗口函数、排序和表更新。此可选函数在查询规划期间调用。目前，只有查询涉及的所有基础关系都属于同一个 FDW 时，才会调用它。对于 FDW 能在远程执行的扫描或连接之后的处理，此函数应生成 `ForeignPath` 路径，并调用 `add_path`，将这些路径加入指定的上层关系。与 `GetForeignJoinPaths` 一样，此函数不必保证成功创建任何路径，因为始终可以采用本地处理的路径。

`stage` 参数标识当前考虑的是扫描或连接之后的哪个步骤。`output_rel` 是应接收该步骤计算路径的上层关系，`input_rel` 是表示该步骤输入的关系。`extra` 参数提供额外细节，目前只在 `UPPERREL_PARTIAL_GROUP_AGG` 或 `UPPERREL_GROUP_AGG` 时设置，此时它指向 `GroupPathExtraData` 结构。（注意，加入 `output_rel` 的 `ForeignPath` 路径通常不会直接依赖 `input_rel` 的路径，因为这些处理应在外部完成。不过，检查为前一处理步骤生成的路径，有助于避免重复的规划工作。）

更多信息请见第 57.4 节。

57.2.4. 更新外部表的 FDW 例程

如果一个 FDW 支持可写外部表，那么应根据该 FDW 的需要和能力提供以下部分或全部回调函数：

```
void
AddForeignUpdateTargets(Query *parsetree,
                        RangeTblEntry *target_rte,
                        Relation target_relation);
```

`UPDATE` 和 `DELETE` 操作是在先前由表扫描函数取回的行上执行的。FDW 可能需要额外的信息（例如行 ID 或主键列的值）来保证它能够定位要更新或删除的准确行。为支持这一点，这个函数可以向将在 `UPDATE` 或 `DELETE` 期间从外部表取回的列列表中添加额外的隐藏列，也就是 “junk” 目标列。

为此，应向 `parsetree->targetList` 添加 `TargetEntry` 项，其中包含要额外取得的值的表达式。每个这样的项都必须标记为 `resjunk = true`，且必须有不同的 `resname`，以便在执行时识别。避免使用与 `ctid`*N*、`wholerow` 或 `wholerow`*N* 匹配的名称，因为核心系统可能生成这些名称的 junk 列。如果额外表达式比简单的 Var 更复杂，在加入目标列表之前，必须先用 `eval_const_expressions` 处理。

虽然此函数在规划期间调用，但提供的信息与其他规划例程得到的信息略有不同。`parsetree` 是 `UPDATE` 或 `DELETE` 命令的语法解析树，而 `target_rte` 和 `target_relation` 描述目标外部表。

如果 AddForeignUpdateTargets 指针被设置为 NULL，则不会添加额外的目标表达式。（这会使DELETE 操作无法实现，不过如果 FDW 依赖一个不会变化的主键来标识行，UPDATE 仍可能可行。）

```
List *
PlanForeignModify(PlannerInfo *root,
                  ModifyTable *plan,
                  Index resultRelation,
                  int subplan_index);
```

执行外部表上的插入、更新或删除所需的任何附加规划动作。这个函数生成 FDW 私有信息，该信息将被附加到执行该修改操作的ModifyTable 计划节点上。这个私有信息必须采用 List 形式，并会在执行阶段传递给BeginForeignModify。

root 是规划器中关于查询的全局信息。plan 是 ModifyTable 计划节点，除 fdwPrivLists 字段外已经完整。resultRelation 用范围表索引标识目标外部表。subplan_index 标识这是 ModifyTable 计划节点的哪个目标，从零开始计数；如果需要索引 plan->plans 或 plan 节点的其他子结构，可使用此值。

更多信息请见第 57.4 节。

如果 PlanForeignModify 指针被设置为 NULL，就不会执行额外的规划期动作，传递给 BeginForeignModify 的 fdw_private 列表将为 NIL。

```
void
BeginForeignModify(ModifyTableState *mtstate,
                   ResultRelInfo *rinfo,
                   List *fdw_private,
                   int subplan_index,
                   int eflags);
```

开始执行外部表修改操作。执行器启动时会调用此例程，它应完成实际修改表之前所需的初始化。随后，会针对每个待插入、更新或删除的元组，分别调用 ExecForeignInsert，ExecForeignUpdate 或 ExecForeignDelete。

mtstate 是正在执行的 ModifyTable 计划节点的整体状态；可通过该结构体访问计划和执行状态的全局数据。rinfo 是描述目标外部表的 ResultRelInfo 结构体。（ResultRelInfo 的 ri_FdwState 字段可供 FDW 存储本次操作所需的任意私有状态。）fdw_private 包含由 PlanForeignModify 生成的私有数据（如果有的话）。subplan_index 标识这是 ModifyTable 计划节点的哪个目标。eflags 包含描述执行器对该计划节点操作模式的标志位。

注意，当(eflags & EXEC_FLAG_EXPLAIN_ONLY) 为真时，这个函数不应执行任何外部可见的动作；它只需做最少的工作，使节点状态对ExplainForeignModify 和 EndForeignModify 有效。

如果 BeginForeignModify 指针被设置为 NULL，在执行器启动期间将不会采取任何动作。

```
TupleTableSlot *
ExecForeignInsert(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);
```

向外部表中插入一个元组。estate 是该查询的全局执行状态。rinfo 是描述目标外部表的 ResultRelInfo 结构体。slot 包含要插入的元组；它将匹配该外部表定义的行类型。planSlot 包含由 ModifyTable 计划节点的子计划生成的元组；它与 slot 的不同之处在于可能包含额外的 “junk” 列。（对于 INSERT 情况，planSlot 通常没有太大意义，但为了完整性仍然提供它。）

返回值可以是一个包含实际被插入数据的槽（例如，触发器动作可能使其不同于所提供的数据），或者为 NULL，表示实际上没有插入任何行（通常也是触发器导致的）。传入的slot 可重用于这一目的。

只有 INSERT 查询包含 RETURNING 子句，或外部表具有 AFTER ROW 触发器时，才会使用返回槽中的数据。触发器需要所有列，但 FDW 可以根据 RETURNING 子句的内容，选择省略部分或全部列的返回，以进行优化。无论如何，都必须返回某个槽来表示成功，否则查询报告的行数会不正确。

如果 ExecForeignInsert 指针被设置为 NULL，尝试向外部表插入将会失败并报告一个错误消息。

注意，将路由后的元组插入外部表分区，或者在外部表上执行COPY FROM 时，也会调用此函数；在这些情况下，它的调用方式与INSERT 时不同。下文介绍了供 FDW 支持这些操作的回调函数。

```
TupleTableSlot *
ExecForeignUpdate(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);
```

更新外部表中的一个元组。estate 是查询的全局执行状态。rinfo 是描述目标外部表的 ResultRelInfo 结构。slot 包含元组的新数据，与外部表的行类型定义一致。planSlot 包含由 ModifyTable 计划节点的子计划生成的元组；它与 slot 的不同之处在于可能包含额外的 “junk” 列。特别是，可以从此槽中取得由 AddForeignUpdateTargets 请求的所有 junk 列。

返回值可以是一个包含实际被更新数据的槽（例如，触发器动作可能导致它与提供的数据不同），或者为 NULL，表示实际上没有更新任何行（通常也是触发器导致的）。传入的slot 可重用于这一目的。

只有 UPDATE 查询包含 RETURNING 子句，或外部表具有 AFTER ROW 触发器时，才会使用返回槽中的数据。触发器需要所有列，但 FDW 可以根据 RETURNING 子句的内容，选择省略部分或全部列的返回，以进行优化。无论如何，都必须返回某个槽来表示成功，否则查询报告的行数会不正确。

如果 ExecForeignUpdate 指针被设置为 NULL，尝试更新外部表将会失败并报告一个错误消息。

```
TupleTableSlot *
ExecForeignDelete(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);
```

从外部表删除一个元组。estate 是查询的全局执行状态。rinfo 是描述目标外部表的 ResultRelInfo 结构体。slot 在调用时不包含任何有用内容，但可以用来保存返回的元组。planSlot 包含由 ModifyTable 计划节点的子计划生成的元组；特别是，它会携带AddForeignUpdateTargets 所请求的任意 junk 列。这些 junk 列必须用来标识待删除的元组。

返回值可以是一个包含实际被删除行的槽，也可以是 NULL，表示实际上没有删除任何行（通常是触发器导致的）。传入的slot 可被用来保存待返回的元组。

返回槽中的数据只会在以下情况下使用：DELETE 查询带有 RETURNING 子句，或者外部表具有 AFTER ROW 触发器。触发器需要所有列，但 FDW 可以根据 RETURNING 子句的内容，选择优化掉部分或全部返回列。无论如何，都必须返回一个槽以表示成功，否则查询报告的行数会不正确。

如果 ExecForeignDelete 指针被设置为 NULL，尝试从外部表中删除将会失败并报告一个错误消息。

```
void
EndForeignModify(EState *estate,
                 ResultRelInfo *rinfo);
```

结束表更新并释放资源。通常无需释放用 palloc 分配的内存，但例如打开的文件和到远程服务器的连接应当清理。

如果 EndForeignModify 指针被设置为 NULL，在执行器关闭期间不会采取任何动作。

被 INSERT 或者 COPY FROM 插入到分区表中的元组会被路由到分区。如果一个 FDW 支持可路由的外部表分区，它还应该提供下面的回调函数。当在外部表上执行COPY FROM 时，也会调用这些函数。

```
void
BeginForeignInsert(ModifyTableState *mtstate,
                   ResultRelInfo *rinfo);
```

开始在外部表上执行插入操作。无论外部表是元组路由选中的分区，还是 COPY FROM 命令指定的目标，都会在第一个元组插入该外部表之前调用此例程。它应完成实际插入之前所需的初始化。随后，会针对每个待插入外部表的元组调用 ExecForeignInsert。

mtstate 是正在执行的 ModifyTable 计划节点的整体状态；可通过该结构体访问计划和执行状态的全局数据。rinfo 是描述目标外部表的 ResultRelInfo 结构体。（ResultRelInfo 的 ri_FdwState 字段可供 FDW 存储本次操作所需的任意私有状态。）

当此函数由 COPY FROM 命令调用时，不会提供 mtstate 中与计划相关的全局数据；随后对每个插入元组调用 ExecForeignInsert 时，其 planSlot 参数都是 NULL，无论该外部表是作为元组路由选中的分区，还是命令中指定的目标。

如果 BeginForeignInsert 指针被设置为 NULL，则不会执行初始化操作。

请注意，如果 FDW 不支持可路由的外部表分区和/或在外部表上执行 COPY FROM，则这个函数或随后对 ExecForeignInsert 的调用必须在需要时抛出错误。

```
void
EndForeignInsert(EState *estate,
                 ResultRelInfo *rinfo);
```

结束插入操作并且释放资源。通常释放 palloc 的内存并不重要，但是打开的文件和与远程服务器的连接应该被清除。

如果 EndForeignInsert 指针被设置为 NULL，则不会执行终止操作。

```
int
IsForeignRelUpdatable(Relation rel);
```

报告指定的外部表支持哪些更新操作。返回值应该是一个规则事件编号的位掩码，它指示了哪些操作被外部表支持，它使用CmdType 枚举，即：(1 << CMD_UPDATE) = 4 表示 UPDATE、(1 << CMD_INSERT) = 8 表示 INSERT 以及 (1 << CMD_DELETE) = 16 表示 DELETE。

如果 IsForeignRelUpdatable 指针被设置为 NULL，而 FDW 提供了 ExecForeignInsert、ExecForeignUpdate 或 ExecForeignDelete，则外部表分别被假定为可插入、可更新或可删除。只有在 FDW 支持某些表是可更新的而某些不是可更新的时候，才需要这个函数（即便如此，也允许在执行例程中抛出一个错误而不是在这个函数中检查。但是，这个函数被用来决定显示在information_schema 视图中的可更新性）。

可以通过实现另一组接口，优化外部表上的某些插入、更新和删除操作。普通的插入、更新和删除接口会从远程服务器取得行，再逐行修改。在某些情况下，逐行处理是必需的，但效率可能不高。如果外部服务器无需实际取回行就能确定要修改哪些行，而且没有会影响该操作的本地结构（本地行级触发器或来自父视图的 WITH CHECK OPTION 约束），就可以安排整个操作在远程服务器上执行。下面介绍的接口可以实现这一点。

```
bool
PlanDirectModify(PlannerInfo *root,
                 ModifyTable *plan,
                 Index resultRelation,
                 int subplan_index);
```

决定是否可以安全地在远程服务器上执行直接修改。如果可以，就在执行所需的规划动作后返回 true；否则返回 false。这个可选函数在查询规划期间被调用。如果该函数成功，执行阶段将改为调用BeginDirectModify、IterateDirectModify 和 EndDirectModify。否则，对该表的修改将使用上文所述的表更新函数执行。参数与PlanForeignModify 相同。

要在远程服务器上直接执行修改，此函数必须将目标子计划重写为一个 ForeignScan 计划节点，由它在远程服务器上执行直接修改。ForeignScan 的 operation 字段必须设为相应的 CmdType 枚举值：UPDATE 对应 CMD_UPDATE，INSERT 对应 CMD_INSERT，DELETE 对应 CMD_DELETE。

更多信息请见第 57.4 节。

如果 PlanDirectModify 指针被设置为 NULL，不会尝试在远程服务器上执行直接修改。

```
void
BeginDirectModify(ForeignScanState *node,
                  int eflags);
```

准备在远程服务器上执行直接修改。这个函数会在执行器启动时被调用。它应执行直接修改开始前所需的任何初始化工作；实际的直接修改应在第一次调用 IterateDirectModify 时进行。ForeignScanState 节点已经创建，但其fdw_state 字段仍为 NULL。有关待修改表的信息可通过 ForeignScanState 节点访问，特别是底层的ForeignScan 计划节点，其中包含了由 PlanDirectModify 提供的任何 FDW 私有信息。eflags 包含描述执行器对该计划节点操作模式的标志位。

注意，当(eflags & EXEC_FLAG_EXPLAIN_ONLY) 为真时，这个函数不应执行任何外部可见的动作。它只应做使该节点状态对ExplainDirectModify 和 EndDirectModify 有效所需的最少工作。

如果 BeginDirectModify 指针被设置为 NULL，不会尝试在远程服务器上执行直接修改。

```
TupleTableSlot *
IterateDirectModify(ForeignScanState *node);
```

当 INSERT，UPDATE 或 DELETE 查询没有 RETURNING 子句时，在远程服务器上直接执行修改后，返回 NULL 即可。如果查询包含该子句，则应取得一条包含 RETURNING 计算所需数据的结果，并将其放入元组表槽中返回（应使用节点的 ScanTupleSlot）。实际插入、更新或删除的数据，必须存入 es_result_relation_info->ri_projectReturning->pi_exprContext->ecxt_scantuple，该字段属于节点的 EState。如果没有更多行，返回 NULL。注意，此函数在短期内存上下文中调用，该上下文会在各次调用之间重置。如果需要更长久的存储，请在 BeginDirectModify 中创建内存上下文，或者使用 es_query_cxt，该字段属于节点的 EState。

如果提供了 fdw_scan_tlist 目标列表，则被返回的行必须匹配它。否则，被返回的行必须匹配被更新的外部表的行类型。如果选择优化掉RETURNING 计算不需要的列，应该在这些列的位置上插入空值，或者生成一个忽略这些列的fdw_scan_tlist 列表。

无论查询是否带有该子句，查询报告的行数都必须由 FDW 自行递增。当查询不带该子句时，在 EXPLAIN ANALYZE 情况下，FDW 还必须递增 ForeignScanState 节点上的行计数。

如果 IterateDirectModify 指针被设置为 NULL，不会尝试在远程服务器上执行直接修改。

```
void
EndDirectModify(ForeignScanState *node);
```

在远程服务器上直接修改后进行清理。通常无需专门释放 palloc 分配的内存，但应清理打开的文件、到远程服务器的连接等资源。

如果 EndDirectModify 指针被设置为 NULL，不会尝试在远程服务器上执行直接修改。

57.2.5. 行锁的 FDW 例程

如果一个 FDW 希望支持晚期行锁定（如第 57.5 节中所述），它必须提供下列回调函数：

```
RowMarkType
GetForeignRowMarkType(RangeTblEntry *rte,
                      LockClauseStrength strength);
```

报告应当为外部表使用哪一种行标记选项。rte 是该表的 RangeTblEntry 节点，而 strength 描述相关 FOR UPDATE/SHARE 子句（如果有）所请求的锁强度。结果必须是 RowMarkType 枚举类型中的一个成员。

这个函数在查询规划期间会为每一个出现在 UPDATE、DELETE 或者 SELECT FOR UPDATE/SHARE 查询中的外部表调用，并且该外部表不是UPDATE 和 DELETE 的目标。

如果 GetForeignRowMarkType 指针被设置为 NULL，将总是使用ROW_MARK_COPY 选项（这意味着将不会调用 RefetchForeignRow，因此也不必提供它）。

更多信息请见第 57.5 节。

```
HeapTuple
RefetchForeignRow(EState *estate,
                  ExecRowMark *erm,
                  Datum rowid,
                  bool *updated);
```

从外部表重新取得一个元组，必要时先将其锁定。estate 是查询的全局执行状态。erm 是 ExecRowMark 结构，描述目标外部表及需要取得的行锁类型（如有）。rowid 标识要取得的元组。updated 是输出参数。

此函数应返回用 palloc 分配的所取元组的副本；如果无法取得行锁，则返回 NULL。需要取得的行锁类型由 erm->markType 定义，它是先前 GetForeignRowMarkType 返回的值。（ROW_MARK_REFERENCE 表示只重新取得元组，不取得任何锁；此例程不会收到 ROW_MARK_COPY。）

此外，如果取得的是一个更新过的版本而不是之前获得的同一版本，*updated 应被设置为 true（如果 FDW 无法确定这一点，推荐总是返回true）。

注意，默认情况下，无法取得行锁应抛出错误；只有 erm->waitPolicy 指定了 SKIP LOCKED 选项时，才适合返回 NULL。

rowid 是要被重新取得的行之前读到的 ctid 值。尽管 rowid 值被作为 Datum 传递，但是目前它只能是tid 类型的值。选择该函数 API 是希望未来能允许其他的行 ID 数据类型。

如果 RefetchForeignRow 指针被设置为 NULL，重新取得行的尝试将会失败并伴随有一个错误消息。

更多信息请见第 57.5 节。

```
bool
RecheckForeignScan(ForeignScanState *node,
                   TupleTableSlot *slot);
```

重新检查之前返回的元组是否仍然匹配相关的扫描和连接条件，并且可能提供该元组的一个修改版本。对于不执行连接下推的外部数据包装器，通常把这设置为NULL 并且恰当地设置 fdw_recheck_quals 会更方便。不过当外连接被下推时，把与所有基表相关的检查重新应用在结果元组上是不够的，即便所有需要的属性都存在也是如此，因为某个条件不匹配的结果可能是某些属性变成 NULL，而不是完全不返回该元组。RecheckForeignScan 可以重新检查这些条件，并在它们仍然满足时返回真，否则返回假；它也可以把一个替换元组存入所提供的槽中。

要实现连接下推，外部数据包装器通常会构造一个替代性的本地连接计划，它只用于重新检查。这将成为 ForeignScan 的外层子计划。在需要执行重检查时，可以执行这个子计划，并把结果元组存入槽中。该计划不必很高效，因为不会有任何基表返回超过一行。例如，它可以把所有连接都实现为嵌套循环。函数 GetExistingLocalJoinPath 可用于在已有路径中搜索合适的本地连接路径，并将其用作替代性的本地连接计划。GetExistingLocalJoinPath 会在指定连接关系的路径列表中搜索一个非参数化路径（如果找不到这样的路径，它将返回 NULL；在这种情况下，外部数据包装器可以自行构造本地路径，或者选择不为该连接创建访问路径）。

57.2.6. EXPLAIN 的 FDW 例程

```
void
ExplainForeignScan(ForeignScanState *node,
                   ExplainState *es);
```

打印额外的 EXPLAIN 输出，用于说明外部表扫描。此函数可以调用 ExplainPropertyText 及相关函数，向 EXPLAIN 输出添加字段。可以利用 es 中的标志字段确定打印内容；也可以检查 ForeignScanState 节点的状态，以提供运行时统计信息，这适用于 EXPLAIN ANALYZE 的情况。

如果 ExplainForeignScan 指针被设置为 NULL，在EXPLAIN 期间不会打印任何额外的信息。

```
void
ExplainForeignModify(ModifyTableState *mtstate,
                     ResultRelInfo *rinfo,
                     List *fdw_private,
                     int subplan_index,
                     struct ExplainState *es);
```

打印额外的 EXPLAIN 输出，用于说明外部表更新。此函数可以调用 ExplainPropertyText 及相关函数，向 EXPLAIN 输出添加字段。可以利用 es 中的标志字段确定打印内容；也可以检查 ModifyTableState 节点的状态，以提供运行时统计信息，这适用于 EXPLAIN ANALYZE 的情况。前四个参数与以下函数相同：BeginForeignModify。

如果 ExplainForeignModify 指针被设置为 NULL，在EXPLAIN 期间不会打印任何额外的信息。

```
void
ExplainDirectModify(ForeignScanState *node,
                    ExplainState *es);
```

为远程服务器上的直接修改打印额外的 EXPLAIN 输出。这个函数可以调用 ExplainPropertyText 及相关函数来向 EXPLAIN 输出添加字段。es 中的标志字段可用于确定要打印什么，而 ForeignScanState 节点的状态可被检查，以便在 EXPLAIN ANALYZE 情况下提供运行时统计信息。

如果 ExplainDirectModify 指针被设置为 NULL，EXPLAIN 期间不会打印出额外的信息。

57.2.7. ANALYZE 的 FDW 例程

```
bool
AnalyzeForeignTable(Relation relation,
                    AcquireSampleRowsFunc *func,
                    BlockNumber *totalpages);
```

当在外部表上执行 ANALYZE 时会调用这个函数。如果 FDW 能为该外部表收集统计信息，它应返回true，并在 *func* 中提供一个用于从该表采集样本行的函数指针，同时在 *totalpages* 中提供以页面数表示的表大小估计值。否则，返回false。

如果 FDW 不支持为任何表收集统计信息，AnalyzeForeignTable 指针可以被设置为 NULL。

如果提供样本收集函数，其签名必须为

```
int
AcquireSampleRowsFunc(Relation relation,
                      int elevel,
                      HeapTuple *rows,
                      int targrows,
                      double *totalrows,
                      double *totaldeadrows);
```

应从表中随机收集最多 *targrows* 行，并存入调用者提供的 *rows* 数组。必须返回实际收集的行数。此外，还应将表中存活行和死行总数的估计值，分别存入输出参数 *totalrows* 和 *totaldeadrows*。（如果 FDW 没有死行的概念，则将 *totaldeadrows* 设为零。）

57.2.8. IMPORT FOREIGN SCHEMA 的 FDW 例程

```
List *
ImportForeignSchema(ImportForeignSchemaStmt *stmt, Oid serverOid);
```

取得外部表创建命令的列表。执行 IMPORT FOREIGN SCHEMA 时会调用此函数，传入该语句的语法解析树和要使用的外部服务器的 OID。它应返回一个 C 字符串列表，每个字符串必须包含一条CREATE FOREIGN TABLE 命令。这些字符串将由核心服务器解析并执行。

在 ImportForeignSchemaStmt 结构体中，remote_schema 是要从其中导入这些表的远程模式的名称。list_type 标识如何过滤表名：FDW_IMPORT_SCHEMA_ALL 表示该远程模式中的所有表都应该被导入（这种情况下 table_list 为空），FDW_IMPORT_SCHEMA_LIMIT_

TO 表示只包括 table_list 中列出的表，而FDW_IMPORT_SCHEMA_EXCEPT 则表示排除 table_list 中列出的表。options 是一个用于该导入处理的选项列表。选项的含义由 FDW 决定。例如，一个 FDW 可以用一个选项来定义是否应该导入列的NOT NULL 属性。这些选项不需要与那些 FDW 支持的数据库对象选项有什么关系。

FDW 可以忽略 ImportForeignSchemaStmt 的 local_schema 字段，因为核心服务器会自动把该名称插入解析后的 CREATE FOREIGN TABLE 命令中。

FDW 也不必担心实现 list_type 以及 table_list 所指定的过滤，因为核心服务器将自动根据那些选项跳过为被排除的表所返回的命令。不过，从一开始就避免为被排除的表创建命令，通常可以省去这部分工作。函数IsImportableForeignTable() 可能有助于检查给定的外部表名是否能通过该过滤器。

如果 FDW 不支持导入表定义，ImportForeignSchema 指针可以被设置为 NULL。

57.2.9. 并行执行的 FDW 例程

ForeignScan 节点可以选择支持并行执行。一个并行的 ForeignScan 会在多个进程中执行，并且每个元组在这些协作进程中必须只返回一次。为做到这一点，各进程可以通过固定大小的动态共享内存块协作。并不保证这部分共享内存在每个进程中都映射到相同地址，因此其中不能包含指针。下面这些函数都是可选的，但若要支持并行执行，就必须提供其中的大部分。

```
bool
IsForeignScanParallelSafe(PlannerInfo *root, RelOptInfo *rel,
                          RangeTblEntry *rte);
```

测试某个扫描是否可以在并行工作进程中执行。只有当规划器认为可能使用并行计划时才会调用这个函数；如果该扫描在并行工作进程中安全，此函数应返回真。如果远程数据源具有事务语义，通常这并不成立，除非工作进程到该数据源的连接能够以某种方式共享与领导者相同的事务环境。

如果没有定义这个函数，则假定该扫描必须放在并行领导者中。注意，返回真并不意味着该扫描本身可以并行完成，只是说明它可以在并行工作进程中执行。因此，即便不支持并行执行，定义这个方法也可能有用。

```
Size
EstimateDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt);
```

估算并行操作所需的动态共享内存的数量。这可能比实际要用的数量更大，但是绝不能更小。返回值的单位是字节。这个函数是可选的，并且在不需要时可以省略。但是如果它被省略，接下来的三个函数也必须被省略，因为不会为 FDW 分配共享内存。

```
void
InitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
                         void *coordinate);
```

初始化并行操作所需的动态共享内存。coordinate 指向一块共享内存区域，其尺寸等于 EstimateDSMForeignScan 的返回值。这个函数是可选的，并且在不需要时可以省略。

```
void
ReInitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
                           void *coordinate);
```

当外部扫描计划将要被重新扫描时，重新初始化并行操作所要求的动态共享内存。这个函数是可选的，并且在不需要时可以省略。推荐的措施是这个函数只重置共享状态，而 ReScanForeignScan 函数仅重置本地状态。当前，这个函数将在ReScanForeignScan 之前被调用，但是最好不要依赖于这种顺序。

```
void
InitializeWorkerForeignScan(ForeignScanState *node, shm_toc *toc,
                            void *coordinate);
```

基于领导者在 InitializeDSMForeignScan 期间建立的共享状态初始化并行工作者的本地状态。这个函数是可选的，并且在不需要时可以省略。

```
void
ShutdownForeignScan(ForeignScanState *node);
```

在预期该节点不会执行到完成时释放资源。这个函数并不会在所有情况下都被调用；有时可能会在没有先调用它的情况下直接调用 EndForeignScan。由于并行查询使用的 DSM 段会在这个回调被调用后立即销毁，因此希望在 DSM 段消失前执行某些动作的外部数据包装器应当实现这个方法。

57.2.10. 路径重新参数化的 FDW 例程

```
List *
ReparameterizeForeignPathByChild(PlannerInfo *root, List *fdw_private,
                                 RelOptInfo *child_rel);
```

这个函数在把一个按给定子关系 child_rel 的最顶层父关系参数化的路径转换为按该子关系本身参数化时调用。它用于重参数化保存在给定 ForeignPath 的 fdw_private 成员中的任意路径，或者调整其中保存的任意表达式节点。该回调可按需使用 reparameterize_path_by_child、adjust_appendrel_attrs 或 adjust_appendrel_attrs_multilevel。

57.3. 外部数据包装器辅助函数

核心服务器导出了若干辅助函数，使外部数据包装器作者能够方便地访问 FDW 相关对象的属性，例如 FDW 选项。要使用这些函数中的任意一个，需要在源码文件中包含头文件 foreign/foreign.h。该头文件也定义了这些函数返回的结构体类型。

```
ForeignDataWrapper *
GetForeignDataWrapper(Oid fdwid);
```

这个函数为具有给定 OID 的外部数据包装器返回一个 ForeignDataWrapper 对象。该 ForeignDataWrapper 对象包含该 FDW 的属性（详见 foreign/foreign.h）。

```
ForeignServer *
GetForeignServer(Oid serverid);
```

这个函数为一个具有给定 OID 的外部服务器返回 `ForeignServer` 对象。该 `ForeignServer` 对象包含该服务器的属性（详见 `foreign/foreign.h`）。

```
UserMapping *
GetUserMapping(Oid userid, Oid serverid);
```

这个函数为给定角色在给定服务器上的用户映射返回 `UserMapping` 对象（如果指定用户没有映射，则返回`PUBLIC` 的映射；如果也没有，则抛出错误）。该`UserMapping` 对象包含该用户映射的属性（详见 `foreign/foreign.h`）。

```
ForeignTable *
GetForeignTable(Oid relid);
```

该函数为一个具有给定 OID 的外部表返回 `ForeignTable` 对象。该 `ForeignTable` 对象包含该外部表的属性（详见 `foreign/foreign.h`）。

```
List *
GetForeignColumnOptions(Oid relid, AttrNumber attnum);
```

这个函数为具有给定外部表 OID 和属性号的列返回 FDW 选项，形式为一个`DefElem` 列表。如果该列没有选项则返回 NIL。

某些对象类型除了基于 OID 的查找函数之外，还具有基于名称的查找函数：

```
ForeignDataWrapper *
GetForeignDataWrapperByName(const char *name, bool missing_ok);
```

这个函数为一个具有给定名称的外部数据包装器返回 `ForeignDataWrapper` 对象。如果找不到该包装器，则在 missing_ok 为真时返回 NULL，否则抛出错误。

```
ForeignServer *
GetForeignServerByName(const char *name, bool missing_ok);
```

这个函数为一个具有给定名称的外部服务器返回 `ForeignServer` 对象。如果找不到该服务器，则在 missing_ok 为真时返回 NULL，否则抛出错误。

57.4. 外部数据包装器查询规划

FDW 回调函数 `GetForeignRelSize`、`GetForeignPaths`、`GetForeignPlan`、`PlanForeignModify`、`GetForeignJoinPaths`、`GetForeignUpperPaths` 以及 `PlanDirectModify` 必须适应 PostgreSQL 规划器的工作方式。下面给出一些关于它们必须做什么的说明。

`root` 和 `baserel` 中的信息可用于减少必须从外部表获取的信息量（从而降低代价）。`baserel->baserestrictinfo` 特别重要，因为它包含限制条件（`WHERE` 子句），这些条件应当用来过滤待获取的行。（FDW 本身并不一定非要强制这些条件，因为核心执行器也可以检查

它们。）baserel->reltarget->exprs 可用于确定需要取回哪些列；但要注意，它只列出必须由 ForeignScan 计划节点输出的列，而不包括在条件求值中使用但不在查询结果中输出的列。

FDW 规划函数可以利用多个私有字段来保存信息。通常，存储在 FDW 私有字段中的任何内容都应通过 palloc 分配，这样它们会在规划结束时被回收。

baserel->fdw_private 是一个 void 指针，FDW 规划函数可以用它来存储与特定外部表相关的信息。核心规划器除了在创建 RelOptInfo 节点时把它初始化为 NULL 之外，不会碰它。它非常适合把信息从GetForeignRelSize 传给 GetForeignPaths，以及/或者从 GetForeignPaths 传给 GetForeignPlan，从而避免重复计算。

GetForeignPaths 可以通过把私有信息存储在 ForeignPath 节点的 fdw_private 字段中，来标识不同访问路径的含义。fdw_private 被声明为 List 指针，但实际上可以包含任何内容，因为核心规划器不会碰它。不过，最佳实践是使用一种能够被 nodeToString 导出的表示形式，以便利用后端提供的调试支持。

GetForeignPlan 可以检查所选中 ForeignPath 节点的 fdw_private 字段，并生成 fdw_exprs 与 fdw_private 列表，放入 ForeignScan 计划节点中，以供执行期使用。这两个列表都必须采用 copyObject 能够复制的表示形式。fdw_private 列表没有其他限制，核心后端也不会以任何方式解释它。如果 fdw_exprs 列表不是 NIL，则它应包含打算在运行时执行的表达式树。这些树会经过规划器的后处理，变成完全可执行的形式。

在 GetForeignPlan 中，通常可以把传入的目标列表原样复制到计划节点中。传入的 scan_clauses 列表与 baserel->baserestrictinfo 包含相同的子句，但可能会为了更高的执行效率而重新排序。在简单情况下，FDW 可以仅从 scan_clauses 列表中剥离 RestrictInfo 节点（使用 extract_actual_clauses），并把所有子句都放到计划节点的条件列表中，这意味着所有子句都将在运行时由执行器检查。更复杂的 FDW 可能能够在内部检查某些子句，这种情况下可将这些子句从计划节点的条件列表中移除，从而避免执行器重复检查。

举例来说，FDW 可能会识别出某些形如 *foreign_variable* = *sub_expression* 的限制子句，并判断它们可以利用本地计算出的 *sub_expression* 值在远程服务器上执行。这类子句的识别应当在 GetForeignPaths 期间完成，因为它会影响该路径的代价估计。路径的 fdw_private 字段很可能会包含指向已识别子句的 RestrictInfo 节点的指针。然后，GetForeignPlan 会把该子句从 scan_clauses 中移除，但会把 *sub_expression* 加入 fdw_exprs，以确保它被整理成可执行形式。它还很可能会把控制信息放入计划节点的 fdw_private 字段中，用以告诉执行函数在运行时该做什么。传给远程服务器的查询将会包含类似 WHERE *foreign_variable* = $1 这样的条件，其中参数值在运行时通过求值 fdw_exprs 表达式树获得。

任何从计划节点条件列表中移除的子句，都必须改为加入 fdw_recheck_quals，或者由 RecheckForeignScan 重新检查，以确保在 READ COMMITTED 隔离级别下行为正确。当查询涉及的其他某个表发生并发更新时，执行器可能需要验证原先的全部条件对该元组是否仍然成立，甚至可能是在另一组参数值下验证。使用 fdw_recheck_quals 往往比在 RecheckForeignScan 中自行实现检查更容易，但当外连接已经被下推时，这种方法就不够用了，因为在那种情况下，连接元组的某些字段可能会变成 NULL，而不是让整个元组被拒绝。

另一个可由 FDW 填充的 ForeignScan 字段是 fdw_scan_tlist，它描述 FDW 为该计划节点返回的元组。对于简单的外部表扫描，它可以设置为 NIL，表示返回的元组具有该外部表声明的行类型。非 NIL 的值必须是一个目标列表（即 TargetEntry 列表），其中包含表示返回列的 Var 和/或表达式。例如，这可用于表明 FDW 省略了某些它发现对查询并不需要的列。再比如，如果 FDW 能以低于本地计算的代价完成查询所用的表达式，也可以把这些表达式加入 fdw_scan_tlist。注意，连接计划（由GetForeignJoinPaths 生成的路径创建而来）始终必须提供 fdw_scan_tlist，以描述它们将返回的列集合。

FDW 应当始终至少构造一条仅依赖于表限制子句的路径。在连接查询中，它还可能选择构造依赖连接子句的路径，例如 *foreign_variable* = *local_variable*。这类子句不会出现在 baserel->baserestrictinfo 中，而必须在关系的连接列表中查找。使用这类子句的路径被称为“参数化路径”。它必须使用适当的 param_info 值来标识被选中连接子句中使用到的其他关系；可使用 get_baserel_parampathinfo 来计算该值。在 GetForeignPlan 中，连接子句中的 *local_variable* 部分会被加入 fdw_exprs，之后在运行时的处理方式就与普通限制子句相同。

如果某个 FDW 支持远程连接，GetForeignJoinPaths 就应当像 GetForeignPaths 为基本表生成路径那样，为潜在的远程连接生成 ForeignPath。有关预期连接的信息可以用前面所述的相同方式传递给 GetForeignPlan。不过，baserestrictinfo 对连接关系并不适用；相反，特定连接相关的连接子句会作为单独参数（extra->restrictlist）传给 GetForeignJoinPaths。

FDW 还可能进一步支持直接执行一些高于扫描和连接层次的计划动作，例如分组或聚合。为了提供这类能力，FDW 应当生成路径并将其插入适当的上层关系中。例如，表示远程聚合的路径应当使用add_path 插入到 UPPERREL_GROUP_AGG 关系中。该路径会在代价层面与本地聚合进行比较，而本地聚合通过读取外部关系的简单扫描路径来完成（注意，这样的路径也必须提供，否则在规划时会出现错误）。如果远程聚合路径胜出，而这通常会发生，它就会像往常一样通过调用GetForeignPlan 被转换成计划。推荐在 GetForeignUpperPaths 回调函数中生成这类路径；当查询的所有基本关系都来自同一个 FDW 时，这个回调会为每一个上层关系（也就是每个扫描后/连接后处理步骤）调用一次。

PlanForeignModify 以及第 57.2.4 节中描述的其他回调，都是围绕这样一个假设设计的：外部关系会以常规方式被扫描，然后单行更新将由本地的 ModifyTable 计划节点驱动。这种方法对于更新既需要读取本地表又需要读取外部表的一般情况是必需的。不过，如果某个操作可以完全由外部服务器执行，那么 FDW 就可以生成一个表示该操作的路径，并将其插入到 UPPERREL_FINAL 上层关系中，与 ModifyTable 方案竞争。这种方式也可用于实现远程 SELECT FOR UPDATE，而不是使用第 57.2.5 节中描述的行锁定回调。要记住，插入到 UPPERREL_FINAL 中的路径必须负责实现该查询的全部行为。

在规划 UPDATE 或 DELETE 时，PlanForeignModify 和 PlanDirectModify 可以查找外部表的 RelOptInfo 结构体，并利用先前由扫描规划函数创建的 baserel->fdw_private 数据。不过在 INSERT 中，目标表不会被扫描，因此不存在对应的 RelOptInfo。PlanForeignModify 返回的 List 与 ForeignScan 计划节点的 fdw_private 列表有相同的限制，也就是说，它只能包含 copyObject 知道如何复制的结构体。

带有 ON CONFLICT 子句的 INSERT 不支持显式指定冲突目标，因为远程表上的唯一约束或排他约束在本地是不可见的。这又意味着 ON CONFLICT DO UPDATE 也不受支持，因为在那里冲突目标是必需的。

57.5. 外部数据包装器中的行锁定

如果某个 FDW 的底层存储机制具有锁定单行以防止并发更新这些行的概念，那么通常值得让该 FDW 执行行级锁定，以尽可能接近普通 PostgreSQL 表所采用的语义。这里面需要考虑多个方面。

一个关键决策是执行早期锁定还是晚期锁定。在早期锁定中，一行在第一次从底层存储中取回时就会被锁定；而在晚期锁定中，只有在确认它确实需要被锁定时才加锁。（之所以会有这种差异，是因为有些行可能会被本地检查的限制条件或连接条件丢弃。）早期锁定要简单得多，并且避免了与远程存储之间的额外往返，但它可能导致某些本来无需锁定的行也被锁定，从而降低并发性，甚至引发意外的死锁。此外，只有在后续还能唯一重新识别出需要加锁的那一行时，晚期锁定才是可行的。理想情况下，行标识符应能标识该行的某个特定版本，就像 PostgreSQL 的 TID 那样。

默认情况下，PostgreSQL 在与 FDW 交互时不会考虑锁定问题，但 FDW 可以在没有核心代码显式支持的情况下执行早期锁定。第 57.2.5 节中描述的 API 函数是在 PostgreSQL 9.5 中加入的，它们允许 FDW 在需要时使用晚期锁定。

另一个需要考虑的问题是，在 READ COMMITTED 隔离级别下，PostgreSQL 可能需要针对某个目标元组的更新版本重新检查限制条件和连接条件。重新检查连接条件意味着要重新取得之前与目标元组连接过的非目标行副本。对于标准 PostgreSQL 表，这是通过在连接投影出的列列表中包含非目标表的 TID，并在需要时重新取回这些非目标行来完成的。这种方式能让连接数据集保持紧凑，但它要求重新取回元组的代价较低，而且 TID 必须能唯一标识待重新取回的行版本。因此，对于外部表，默认做法是在连接投影出的列列表中包含从外部表取回的整行副本。这不会对 FDW 提出特殊要求，但可能降低归并连接和哈希连接的性能。能够满足重新取回要求的 FDW 可以选择采用前一种方式。

对于外部表上的 UPDATE 或 DELETE，建议目标表上的 ForeignScan 操作对其取回的行执行早期锁定，例如通过等价于 SELECT FOR UPDATE 的方式。FDW 可以在规划时通过将表的 relid 与 root->parse->resultRelation 比较，或者在执行时通过使用 ExecRelation IsTargetRelation()，来检测某张表是否是 UPDATE/DELETE 的目标。另一种可能性是在 ExecForeignUpdate 或 ExecForeignDelete 回调中执行晚期锁定，但系统并未为此提供专门支持。

对于被 SELECT FOR UPDATE/SHARE 命令指定为需要锁定的外部表，ForeignScan 操作同样可以通过以等价于 SELECT FOR UPDATE/SHARE 的方式取回元组来执行早期锁定。若要改为执行晚期锁定，请提供第 57.2.5 节中定义的回调函数。在 GetForeignRowMarkType 中，应根据所请求的锁强度选择 ROW_MARK_EXCLUSIVE、ROW_MARK_NOKEYEXCLUSIVE、ROW_MARK_SHARE 或 ROW_MARK_KEYSHARE 行标记选项。（无论选择这四者中的哪一个，核心代码的行为都是相同的。）在其他地方，可以在规划时使用get_plan_rowmark，或者在执行时使用ExecFindRowMark，来检测某张外部表是否被此类命令指定为需要加锁；你不仅要检查是否返回了非空的行标记结构体，还必须检查其 strength 字段不为 LCS_NONE。

最后，对于那些出现在 UPDATE、DELETE 或 SELECT FOR UPDATE/SHARE 命令中、但并未被指定为行锁定的外部表，你可以覆盖复制整行的默认做法：让 GetForeignRowMarkType 在看到锁强度 LCS_NONE 时选择 ROW_MARK_REFERENCE 选项。这样会导致 RefetchForeignRow 以该值作为 markType 被调用；它随后应在不获取任何新锁的情况下重新取回该行。（如果你实现了 GetForeignRowMarkType，但不希望重新取回未加锁的行，那么在 LCS_NONE 情况下请选择 ROW_MARK_COPY。）

更多信息请参阅 src/include/nodes/lockoptions.h，以及 src/include/nodes/plannodes.h 中关于 RowMarkType 和 PlanRowMark 的注释，还有 src/include/nodes/execnodes.h 中关于 ExecRowMark 的注释。

第 58 章 编写一种表采样方法

PostgreSQL 对 TABLESAMPLE 子句的实现，除了支持 SQL 标准要求的 BERNOULLI 和 SYSTEM 方法之外，还支持自定义表采样方法。采样方法决定了在使用 TABLESAMPLE 子句时会选取表中的哪些行。

在 SQL 层，表采样方法由单个 SQL 函数表示，该函数通常用 C 实现，其签名为

```
method_name(internal) RETURNS tsm_handler
```

该函数的名称就是出现在 TABLESAMPLE 子句中的方法名。internal 参数是一个占位值（其值始终为零），仅用于防止从 SQL 命令直接调用此函数。函数的结果必须是由 palloc 分配的 TsmRoutine 结构体，其中包含指向采样方法支持函数的指针。这些支持函数是普通的 C 函数，在 SQL 层既不可见也不可调用。有关这些支持函数的说明见第 58.1 节。

除了函数指针之外，TsmRoutine 结构体还必须提供以下额外字段：

```
List *parameterTypes
```

这是一个 OID 列表，包含该采样方法在 TABLESAMPLE 子句中可接受参数的数据类型 OID。例如，对于内置方法，该列表只包含一个值为 FLOAT4OID 的项，它表示采样百分比。自定义采样方法可以有更多参数，也可以有不同的参数。

```
bool repeatable_across_queries
```

如果为 true，只要每次都提供相同的参数和 REPEATABLE 种子值，并且表内容未发生变化，该采样方法就能在连续查询之间返回相同的样本。如果该字段为 false，则该采样方法不接受 REPEATABLE 子句。

```
bool repeatable_across_scans
```

如果为 true，该采样方法就能在同一查询中的连续扫描之间返回相同的样本（假定参数、种子值和快照都保持不变）。如果该字段为 false，规划器将不会选择那些需要对被采样表扫描多次的计划，因为那可能导致查询输出不一致。

TsmRoutine 结构体类型声明在 src/include/access/tsmapi.h 中，更多细节见该文件。

标准发行版中包含的表采样方法，是尝试自行编写方法时的良好参考。内置采样方法位于源代码树的 src/backend/access/tablesample 子目录中，附加方法位于 contrib 子目录中。

58.1. 采样方法支持函数

TSM 处理器函数返回一个通过 palloc 分配的 TsmRoutine 结构体，其中包含下文所述支持函数的指针。大多数函数是必需的，但有些是可选的，对应指针可以为 NULL。

```
void
SampleScanGetSampleSize (PlannerInfo *root,
                         RelOptInfo *baserel,
                         List *paramexprs,
                         BlockNumber *pages,
                         double *tuples);
```

该函数在规划阶段调用。它必须估计采样扫描期间将读取的关系页数，以及该扫描将选出的元组数。（例如，可以先估计采样比例，再将 baserel->pages 和 baserel->tuples 的值乘以该比例，并确保结果舍入为整数值。）paramexprs 列表保存 TABLESAMPLE 子句参数对应的表达式。如果出于估算目的需要这些值，建议使用 estimate_expression_value() 尝试将这些表达式化简为常量；但即使无法化简，该函数也必须给出大小估计，而且即便这些值看起来无效也不应失败（别忘了，它们只是对运行时取值的估计）。pages 和 tuples 参数是输出参数。

```
void
InitSampleScan (SampleScanState *node,
                int eflags);
```

为 SampleScan 计划节点的执行进行初始化。该函数在执行器启动期间调用。它应执行开始处理前所需的任何初始化工作。SampleScanState 节点已经创建，但其 tsm_state 字段为 NULL。InitSampleScan 函数可以通过 palloc 分配采样方法所需的任何内部状态数据，并将其指针存入 node->tsm_state。待扫描表的信息可通过 SampleScanState 节点的其他字段访问（但注意 node->ss.ss_currentScanDesc 扫描描述符尚未设置）。eflags 包含描述执行器对此计划节点工作模式的标志位。

当 (eflags & EXEC_FLAG_EXPLAIN_ONLY) 为真时，不会实际执行该扫描，因此这个函数只应做使节点状态对 EXPLAIN 和 EndSampleScan 有效所需的最少工作。

该函数可以省略（将指针设为 NULL），此时 BeginSampleScan 必须执行采样方法所需的全部初始化工作。

```
void
BeginSampleScan (SampleScanState *node,
                 Datum *params,
                 int nparams,
                 uint32 seed);
```

开始执行一次采样扫描。它会在第一次尝试提取一个元组之前调用；如果该扫描需要重启，也可能再次调用。待扫描表的信息可通过 SampleScanState 节点的字段访问（但注意 node->ss.ss_currentScanDesc 扫描描述符尚未设置）。长度为 nparams 的 params 数组包含 TABLESAMPLE 子句中提供的参数值。这些参数的个数和类型与该采样方法的 parameterTypes 列表中指定的一致，并且已确认不为 NULL。seed 包含采样方法内部生成随机数时要使用的种子；如果给定了 REPEATABLE 值，它就是从该值派生出的 hash 值，否则就是 random() 的结果。

该函数可以调整 node->use_bulkread 和 node->use_pagemode 字段。如果 node->use_bulkread 为 true（默认如此），扫描将使用一种鼓励在使用后回收缓冲区的缓冲区访问策略。如果该扫描只会访问该表中的一小部分页，把它设为 false 可能更合理。如果 node->use_pagemode 为 true（默认如此），扫描将对每个访问页上的所有元组以单遍方式执行可见性检查。如果该扫描只会从每个访问页中选出一小部分元组，把它设为 false 可能更合理。这样执行的元组可见性检查会更少，但每次检查的代价会更高，因为需要更多加锁。

如果采样方法被标记为 repeatable_across_scans，那么在重新扫描期间它必须能够像最初那样选出相同的一组元组；也就是说，重新调用 BeginSampleScan 必须像之前一样选出相同的元组（如果 TABLESAMPLE 参数和种子没有变化）。

```
BlockNumber
NextSampleBlock (SampleScanState *node);
```

返回下一个待扫描页的块号；如果没有剩余页可扫描，则返回 InvalidBlockNumber。

该函数可以省略（将指针设为 NULL），此时核心代码将对整个关系执行顺序扫描。这样的扫描可能使用同步扫描，因此采样方法不能假定每次扫描都会按相同顺序访问关系页。

```
OffsetNumber
NextSampleTuple (SampleScanState *node,
                 BlockNumber blockno,
                 OffsetNumber maxoffset);
```

返回指定页上下一个要采样的元组的偏移号；如果没有剩余元组可采样，则返回 InvalidOffsetNumber。maxoffset 是该页上正在使用的最大偏移号。

> **注意**
>
> NextSampleTuple 不会被显式告知在 1 .. maxoffset 范围内哪些偏移号实际上包含有效元组。通常这不是问题，因为核心代码会忽略对缺失或不可见元组的采样请求；这不应在样本中引入任何偏差。不过，如有需要，该函数可以检查 node->ss.ss_currentScanDesc->rs_vistuples[] 来确定哪些元组有效且可见。（这要求 node->use_pagemode 为 true。）

> **注意**
>
> NextSampleTuple 绝不能假定 blockno 与最近一次 NextSampleBlock 调用返回的页号相同。它是由之前某次 NextSampleBlock 调用返回的，但核心代码可以在真正扫描页之前就调用 NextSampleBlock，以支持预取。可以假定的是，一旦开始对某个给定页进行采样，后续连续的 NextSampleTuple 调用在返回 InvalidOffsetNumber 之前都指向同一页。

```
void
EndSampleScan (SampleScanState *node);
```

结束扫描并释放资源。通常不必专门释放通过 palloc 分配的内存，但任何对外可见的资源都应清理掉。在没有此类资源的常见情况下，可以省略这个函数（将指针设为 NULL）。

第 59 章 编写自定义扫描提供者

PostgreSQL 支持一组实验性功能，旨在允许扩展模块向系统中添加新的扫描类型。与只负责了解如何扫描其自身外部表的外部数据包装器不同，自定义扫描提供者可以为系统中的任何关系提供一种替代扫描方法。通常，编写自定义扫描提供者的动机，是为了能够使用某些核心系统尚不支持的优化，例如缓存或某种形式的硬件加速。本章概述如何编写新的自定义扫描提供者。

实现一种新的自定义扫描类型分为三个步骤。首先，在规划期间，需要生成表示使用所提策略进行扫描的访问路径。其次，如果规划器从这些访问路径中选出一条，作为扫描某个特定关系的最优策略，就必须将该访问路径转换为计划。最后，还必须能够执行该计划，并生成与针对同一关系的其他访问路径原本会生成的结果相同的结果。

59.1. 创建自定义扫描路径

自定义扫描提供者通常通过设置下面的钩子，为基本关系添加路径。核心代码为该关系生成所有可生成的访问路径之后，会调用此钩子（Gather 路径除外，它们会在本次调用之后生成，以便使用钩子添加的部分路径）：

```
typedef void (*set_rel_pathlist_hook_type) (PlannerInfo *root,
                                            RelOptInfo *rel,
                                            Index rti,
                                            RangeTblEntry *rte);
extern PGDLLIMPORT set_rel_pathlist_hook_type set_rel_pathlist_hook;
```

虽然此钩子函数可用于检查、修改或删除核心系统生成的路径，但自定义扫描提供者通常只生成 CustomPath 对象，并将它们添加到rel，所用函数是add_path。自定义扫描提供者负责初始化 CustomPath 对象，其声明如下：

```
typedef struct CustomPath
{
    Path      path;
    uint32    flags;
    List     *custom_paths;
    List     *custom_private;
    const CustomPathMethods *methods;
} CustomPath;
```

path 必须像其他任何路径一样完成初始化，包括行数估计、启动代价和总代价，以及该路径提供的排序顺序。flags 是一个位掩码，如果自定义路径支持反向扫描，就应包含 CUSTOMPATH_SUPPORT_BACKWARD_SCAN；如果支持标记和恢复，就应包含 CUSTOMPATH_SUPPORT_MARK_RESTORE。这两种能力都是可选的。可选字段 custom_paths 是该自定义路径节点所使用的 Path 节点列表；规划器会把它们转换成 Plan 节点。custom_private 可用于存储自定义路径的私有数据。私有数据应以 nodeToString 能够处理的形式存储，这样尝试打印该自定义路径的调试例程才能按预期工作。methods 必须指向一个实现了所需自定义路径方法的对象（通常是静态分配的），下文会进一步说明。

自定义扫描提供者也可以提供连接路径。与基本关系一样，这种路径也必须产生与它所替代连接通常会产生的相同输出。为此，连接提供者应设置下列钩子，并在该钩子函数内部为连接关系创建 CustomPath 路径。

```
typedef void (*set_join_pathlist_hook_type) (PlannerInfo *root,
                                             RelOptInfo *joinrel,
                                             RelOptInfo *outerrel,
                                             RelOptInfo *innerrel,
                                             JoinType jointype,
                                             JoinPathExtraData *extra);
extern PGDLLIMPORT set_join_pathlist_hook_type set_join_pathlist_hook;
```

对于同一个连接关系，这个钩子会针对不同的内外关系组合重复调用；如何尽量减少重复工作，是该钩子的职责。

59.1.1. 自定义扫描路径回调

```
Plan *(*PlanCustomPath) (PlannerInfo *root,
                         RelOptInfo *rel,
                         CustomPath *best_path,
                         List *tlist,
                         List *clauses,
                         List *custom_plans);
```

将自定义路径转换为完成的计划。返回值通常是一个 CustomScan 对象，回调函数必须负责分配并初始化该对象。详见第 59.2 节。

```
List *(*ReparameterizeCustomPathByChild) (PlannerInfo *root,
                                          List *custom_private,
                                          RelOptInfo *child_rel);
```

在把由给定子关系 child_rel 的最顶层父关系参数化的路径，转换为由该子关系参数化的路径时，会调用此回调。该回调用于重新参数化保存在 CustomPath 的给定 custom_private 成员中的任何路径，或者转换其中保存的任何表达式节点。该回调可根据需要使用 reparameterize_path_by_child、adjust_appendrel_attrs 或 adjust_appendrel_attrs_multilevel。

59.2. 创建自定义扫描计划

在完成的计划树中，自定义扫描使用下述结构体表示：

```
typedef struct CustomScan
{
    Scan       scan;
    uint32     flags;
    List      *custom_plans;
    List      *custom_exprs;
    List      *custom_private;
    List      *custom_scan_tlist;
    Bitmapset *custom_relids;
    const CustomScanMethods *methods;
} CustomScan;
```

scan 必须像其他任何扫描一样完成初始化，包括估计代价、目标列表、限定条件等。flags 是一个位掩码，其含义与 CustomPath 中相同。custom_plans 可用于存储子 Plan 节点。custom_exprs 应当用于存储那些需要由 setrefs.c 和 subselect.c 修正的表达式树，而 custom_private 则应用于存放仅供自定义扫描提供者自身使用的其他私有数据。扫描基本关系时，custom_scan_tlist 可以为 NIL，表示该自定义扫描返回的扫描元组与该基本关系的行类型一致。否则，它就是描述实际扫描元组的目标列表。对于连接，必须提供 custom_scan_tlist；如果自定义扫描提供者能够计算某些非 Var 表达式，则在扫描场景下也可以提供该字段。custom_relids 由核心代码设置为该扫描节点所处理关系（范围表索引）的集合；除非该扫描是在替代一个连接，否则它将只有一个成员。methods 必须指向一个实现了所需自定义扫描方法的对象（通常是静态分配的），下文会进一步说明。

当 CustomScan 扫描单个关系时，scan.scanrelid 必须是待扫描表的范围表索引。当它替代的是一个连接时，scan.scanrelid 应为零。

计划树必须能够通过 copyObject 复制，因此存放在 “custom” 字段中的所有数据都必须由该函数能够处理的节点组成。此外，自定义扫描提供者不能像对 CustomPath 或 CustomScanState 那样，用一个包含 CustomScan 的更大结构体替代该结构体本身。

59.2.1. 自定义扫描计划回调

```
Node *(*CreateCustomScanState) (CustomScan *cscan);
```

为该 CustomScan 分配一个 CustomScanState。实际分配的空间往往会比普通 CustomScanState 所需的更大，因为很多提供者都希望把它作为第一个字段嵌入到一个更大的结构体中。返回的值必须适当地设置节点标签和 methods，但在这个阶段，其他字段应当保留为零；待 ExecInitCustomScan 完成基本初始化之后，将调用 BeginCustomScan 回调，让自定义扫描提供者有机会执行其他所需操作。

59.3. 执行自定义扫描

在执行 CustomScan 时，其执行状态由 CustomScanState 表示，其声明如下：

```
typedef struct CustomScanState
{
    ScanState ss;
    uint32    flags;
    const CustomExecMethods *methods;
} CustomScanState;
```

ss 的初始化方式与其他任何扫描状态相同，不过如果该扫描针对的是连接而不是基本关系，那么 ss.ss_currentRelation 会保留为 NULL。flags 是一个位掩码，其含义与 CustomPath 和 CustomScan 中相同。methods 必须指向一个实现了所需自定义扫描状态方法的对象（通常是静态分配的），下文会进一步说明。通常，CustomScanState 实际上会是一个更大的结构体，并把上述结构体嵌为其第一个成员；它不需要支持 copyObject。

59.3.1. 自定义扫描执行回调

```
void (*BeginCustomScan) (CustomScanState *node,
                         EState *estate,
                         int eflags);
```

完成所提供 CustomScanState 的初始化。标准字段已经由 ExecInitCustomScan 初始化，但任何私有字段都应在这里初始化。

```
TupleTableSlot *(*ExecCustomScan) (CustomScanState *node);
```

取出下一个扫描元组。如果仍有元组剩余，它应当使用当前扫描方向中的下一个元组填充 ps_ResultTupleSlot，然后返回该元组槽。否则，应返回 NULL 或一个空槽。

```
void (*EndCustomScan) (CustomScanState *node);
```

清理与 CustomScanState 关联的任何私有数据。这个方法是必需的，但如果没有相关数据，或者相关数据会被自动清理，则它不需要做任何事情。

```
void (*ReScanCustomScan) (CustomScanState *node);
```

将当前扫描倒回到开始处，并准备重新扫描该关系。

```
void (*MarkPosCustomScan) (CustomScanState *node);
```

保存当前扫描位置，以便之后可由 RestrPosCustomScan 回调恢复。这个回调是可选的，只需在设置了 CUSTOMPATH_SUPPORT_MARK_RESTORE 标志时提供。

```
void (*RestrPosCustomScan) (CustomScanState *node);
```

恢复由 MarkPosCustomScan 回调保存的先前扫描位置。这个回调是可选的，只需在设置了 CUSTOMPATH_SUPPORT_MARK_RESTORE 标志时提供。

```
Size (*EstimateDSMCustomScan) (CustomScanState *node,
                               ParallelContext *pcxt);
```

估算并行操作所需的动态共享内存数量。这可能会高于实际使用量，但绝不能低于实际使用量。返回值的单位是字节。这个回调是可选的，只需在该自定义扫描提供者支持并行执行时提供。

```
void (*InitializeDSMCustomScan) (CustomScanState *node,
                                 ParallelContext *pcxt,
                                 void *coordinate);
```

初始化并行操作所需的动态共享内存。coordinate 指向一块共享内存区域，其大小等于 EstimateDSMCustomScan 的返回值。这个回调是可选的，只需在该自定义扫描提供者支持并行执行时提供。

```
void (*ReInitializeDSMCustomScan) (CustomScanState *node,
                                   ParallelContext *pcxt,
                                   void *coordinate);
```

当自定义扫描计划节点即将被重新扫描时，重新初始化并行操作所需的动态共享内存。这个回调是可选的，只需在该自定义扫描提供者支持并行执行时提供。推荐的做法是，此回调只重置共享状态，而 ReScanCustomScan 回调只重置本地状态。目前，这个回调会在 ReScanCustomScan 之前调用，但最好不要依赖这种顺序。

```
void (*InitializeWorkerCustomScan) (CustomScanState *node,
                                    shm_toc *toc,
                                    void *coordinate);
```

基于领导者在 InitializeDSMCustomScan 期间建立的共享状态，初始化并行工作者的本地状态。这个回调是可选的，只需在该自定义扫描提供者支持并行执行时提供。

```
void (*ShutdownCustomScan) (CustomScanState *node);
```

在预期该节点不会执行到完成时释放资源。这个回调不会在所有情况下都被调用；有时，可能会在没有先调用它的情况下直接调用 EndCustomScan。由于并行查询使用的 DSM 段会在调用该回调后立即销毁，因此希望在 DSM 段消失前执行某些动作的自定义扫描提供者应实现此方法。

```
void (*ExplainCustomScan) (CustomScanState *node,
                           List *ancestors,
                           ExplainState *es);
```

为自定义扫描计划节点的 EXPLAIN 输出附加信息。这个回调是可选的。即使没有这个回调，存储在 ScanState 中的公共数据，例如目标列表和扫描关系，也会被显示；但该回调允许显示额外的私有状态。

第 60 章 遗传查询优化器

作者

由 Martin Utesch <utesch@aut.tu-freiberg.de> 为德国弗赖贝格矿业和技术大学自动控制研究所编写。

60.1. 将查询处理看成是一个复杂的优化问题

在所有关系操作符中，最难处理和优化的是连接。随着查询中连接数目的增加，可能的查询计划数量会呈指数增长。为了处理单个连接而支持多种连接方法（例如 PostgreSQL 中的嵌套循环、哈希连接和归并连接），以及作为关系访问路径的多种索引（例如 PostgreSQL 中的 B-树、hash、GiST 和 GIN），也进一步增加了优化工作量。

常规的 PostgreSQL 查询优化器会在可选策略空间中执行近似穷举搜索。该算法最早出现在 IBM 的 System R 数据库中，能够产生近乎最优的连接顺序；但当查询中的连接数变得很大时，它可能耗费极大的时间和内存空间。因此，普通的PostgreSQL 查询优化器并不适合需要连接大量表的查询。

德国弗赖贝格矿业和技术大学自动控制研究所在尝试将 PostgreSQL 用作一个用于电网维护的基于知识的决策支持系统后端时遇到了一些问题。该 DBMS 需要为该基于知识系统中的推理机处理大型连接查询。这些查询中的连接数量使得使用普通查询优化器变得不可行。

下文将介绍一种遗传算法的实现，它以适合涉及大量连接的查询的高效方式来解决连接顺序问题。

60.2. 遗传算法

遗传算法（GA）是一种通过随机化搜索进行工作的启发式优化方法。优化问题的可能解集合被视为由若干个体组成的种群。个体对其环境的适应程度由其适应度表示。

一个个体在搜索空间中的坐标由染色体表示，本质上是一组字符串。基因是染色体的一个片段，它编码某个待优化参数的值。基因的典型编码可以是二进制或整数。

通过模拟重组、变异和选择这些进化操作，可以找到平均适应度高于前代的新一代搜索点。

根据 comp.ai.genetic FAQ 中的说法，再怎么强调也不过分：GA 并不是为了求解问题而进行的纯粹随机搜索。GA 会使用随机过程，但其结果显然并非随机的（优于随机）。

图 60.1. 遗传算法的结构图

P(t)	时刻 t 的祖代
P''(t)	时刻 t 的子代

```
+=========================================+
|>>>>>>>>>>>  Algorithm GA  <<<<<<<<<<<<<|
+=========================================+
| INITIALIZE t := 0                       |
+=========================================+
| INITIALIZE P(t)                         |
+=========================================+
| evaluate FITNESS of P(t)                |
+=========================================+
| while not STOPPING CRITERION do         |
|   +-------------------------------------+
|   | P'(t)  := RECOMBINATION{P(t)}       |
|   +-------------------------------------+
|   | P''(t) := MUTATION{P'(t)}           |
|   +-------------------------------------+
|   | P(t+1) := SELECTION{P''(t) + P(t)}  |
|   +-------------------------------------+
|   | evaluate FITNESS of P''(t)          |
|   +-------------------------------------+
|   | t := t + 1                          |
+===+=====================================+
```

60.3. PostgreSQL 中的遗传查询优化（GEQO）

GEQO 模块把查询优化问题视为著名的旅行商问题（TSP）来处理。可能的查询计划被编码为整数串。每个串表示查询中从一个关系到下一个关系的连接顺序。例如，连接树

```
   /\
  /\ 2
 /\ 3
4  1
```

会被编码为整数串 '4-1-3-2'，这意味着先连接关系 '4' 和 '1'，再连接 '3'，最后连接 '2'；其中 1、2、3、4 是PostgreSQL 优化器内部的关系 ID。

PostgreSQL 中 GEQO 实现的具体特征包括：

- 采用稳态 GA（替换种群中适应度最低的个体，而不是整代替换），能够快速收敛到更优的查询计划。这对于在合理时间内处理查询至关重要；
- 采用边重组交叉，它特别适合在利用 GA 求解 TSP 时将边损失保持在较低水平；
- 不使用变异作为遗传操作符，因此无须借助修复机制来生成合法的 TSP 回路。

GEQO 模块的部分内容改编自 D. Whitley 的 Genitor 算法。

GEQO 模块使 PostgreSQL 查询优化器能够通过非穷举搜索有效支持大型连接查询。

60.3.1. 使用 GEQO 生成可能的计划

GEQO 规划过程使用标准规划器代码来生成各个关系扫描的计划，然后再用遗传方法生成连接计划。如上所示，每个候选连接计划都表示为一个连接基本关系的顺序。在初始阶段，GEQO 代码只是随机生成一些可能的连接序列。对于每一个待考察的连接序列，都会调用标准规划器代码来估算按该序列执行查询的代价。（对于连接序列中的每一步，都会考虑全部三种可能的连接策略；并且所有预先确定的关系扫描计划也都可用。估计代价取这些可能性中最低者。）估计代价较低的连接序列会被认为比代价较高的序列“适应度更高”。遗传算法会丢弃适应度最低的候选，然后通过组合适应度更高的候选的基因来生成新的候选，也就是从已知低代价连接序列中随机选取片段，构造新的序列以供考察。这个过程会重复进行，直到已经考察的连接序列数量达到预设值；最后，使用整个搜索过程中找到的最佳序列生成最终计划。

由于初始种群选择以及后续对最佳候选的“变异”都包含随机选择，因此这一过程天生就是非确定性的。为了避免所选计划发生令人意外的变化，每次运行 GEQO 算法时，都会用当前的 geqo_seed 参数设置重新初始化其随机数生成器。只要 `geqo_seed`、其他 GEQO 参数以及统计信息等其他规划器输入保持不变，对于给定查询就会生成相同的计划。若想试验不同的搜索路径，可以尝试修改 `geqo_seed`。

60.3.2. PostgreSQL GEQO 的未来实现任务

为了改进遗传算法的参数设置，仍有一些工作要做。在文件`src/backend/optimizer/geqo/geqo_main.c` 中的例程 `gimme_pool_size` 和 `gimme_number_generations` 里，我们必须为参数设置找到一种折中，以满足两个相互竞争的需求：

- 查询计划的最优性
- 计算时间

在当前实现中，每个候选连接序列的适应度都是通过从头运行标准规划器的连接选择和代价估算代码来估算的。只要不同候选使用了相似的连接子序列，就会有大量工作被重复执行。如果能保留子连接的代价估算，就可以显著加快这一过程。问题在于，必须避免为了保存这种状态而消耗不合理数量的内存。

从更基础的层面看，用一个为 TSP 设计的 GA 算法来解决查询优化问题是否合适，也并不明确。在 TSP 情况下，与任何子串（部分巡回）相关的代价都独立于巡回的其余部分，但对于查询优化显然并非如此。因此，边重组交叉是否是最有效的变异过程，仍然值得怀疑。

60.4. 进一步阅读

下列资源包含关于遗传算法的更多信息：

- 《Hitch-Hiker 演化计算指南》[1]（comp.ai.genetic[2] FAQ）
- 《演化计算及其在艺术和设计中的应用》[3]，Craig Reynolds 著
- [elma04]
- [fong]

[1] http://www.faqs.org/faqs/ai-faq/genetic/part1/
[2] news://comp.ai.genetic
[3] https://www.red3d.com/cwr/evolve.html

第 61 章 索引访问方法接口定义

本章定义 PostgreSQL 核心系统与管理各个索引类型的索引访问方法之间的接口。除这里规定的内容之外，核心系统对索引一无所知，因此可以通过编写附加代码来开发全新的索引类型。

PostgreSQL 中所有索引在技术上都称为二级索引；也就是说，索引与它所描述的表文件在物理上是分离的。每个索引都存储为独立的物理关系，因此在 `pg_class` 目录中都有一个条目。索引的内容完全由其索引访问方法控制。实践中，所有索引访问方法都会把索引划分为标准大小的页面，以便使用常规的存储管理器和缓冲区管理器访问索引内容。（现有的所有索引访问方法还都使用第 69.6 节中描述的标准页面布局，而且大多数还对索引元组头使用相同的格式；但这些决定并不是访问方法必须遵循的。）

索引本质上是从一些数据键值到行版本（元组）在索引父表中的元组标识符（TIDs）的映射。一个 TID 由块号以及该块中的项号组成（见第 69.6 节）。这些信息足以从表中取出某个特定的行版本。索引并不直接知道在 MVCC 之下同一逻辑行可能会存在多个现存版本；对索引来说，每个元组都是一个独立对象，需要它自己的索引条目。因此，对一行的更新总会为该行创建全新的索引条目，即使键值并未改变也是如此。（HOT 元组是这一说法的例外；但索引同样不直接处理它们。）当死元组自身被回收时（通过清理），它们对应的索引条目也会被回收。

61.1. 索引的基本 API 结构

每种索引访问方法都由系统目录 `pg_am` 中的一行描述。`pg_am` 条目指定了该访问方法的名称以及一个处理器函数。这些条目可以通过 CREATE ACCESS METHOD 和 DROP ACCESS METHOD SQL 命令创建和删除。

索引访问方法的处理器函数必须声明为接受一个 `internal` 类型的参数，并返回伪类型 `index_am_handler`。这个参数只是一个占位值，用来防止处理器函数被 SQL 命令直接调用。该函数的结果必须是一个通过 palloc 分配的 `IndexAmRoutine` 结构体，其中包含核心代码使用该索引访问方法所需的全部信息。`IndexAmRoutine` 结构体也称为该访问方法的 API 结构体，其中的字段指定了访问方法的各种固定属性，例如它是否支持多列索引。更重要的是，它包含该访问方法的支持函数指针，而这些支持函数完成了实际访问索引的全部工作。这些支持函数是普通 C 函数，在 SQL 层不可见也不可调用。支持函数将在第 61.2 节中介绍。

`IndexAmRoutine` 结构体定义如下：

```
typedef struct IndexAmRoutine
{
    NodeTag     type;

    /*
     * 可用于遍历或搜索此 AM 的策略（操作符）总数。
     * 如果 AM 没有固定的一组策略分配，则为零。
     */
    uint16      amstrategies;
    /* 此 AM 使用的支持函数总数 */
    uint16      amsupport;
    /* 操作符类选项支持函数编号，或为 0 */
    uint16      amoptsprocnum;
    /* AM 是否支持按被索引列的值进行 ORDER BY？ */
    bool        amcanorder;
    /* AM 是否支持按被索引列上操作符的结果进行 ORDER BY？ */
    bool        amcanorderbyop;
    /* AM 是否支持使用与 hash AM 一致的 API 进行 hash 运算？ */
    bool        amcanhash;
    /* 操作符族内的操作符是否具有一致的相等语义？ */
```

```
bool        amconsistentequality;
/* 操作符族内的操作符是否具有一致的排序语义？ */
bool        amconsistentordering;
/* AM 是否支持反向扫描？ */
bool        amcanbackward;
/* AM 是否支持 UNIQUE 索引？ */
bool        amcanunique;
/* AM 是否支持多列索引？ */
bool        amcanmulticol;
/* AM 是否要求扫描必须约束第一个索引列？ */
bool        amoptionalkey;
/* AM 是否处理 ScalarArrayOpExpr 限定条件？ */
bool        amsearcharray;
/* AM 是否处理 IS NULL/IS NOT NULL 限定条件？ */
bool        amsearchnulls;
/* 索引存储数据类型是否可以不同于列数据类型？ */
bool        amstorage;
/* 是否可以按此类型的索引进行聚簇？ */
bool        amclusterable;
/* AM 是否处理谓词锁？ */
bool        ampredlocks;
/* AM 是否支持并行扫描？ */
bool        amcanparallel;
/* AM 是否支持并行构建？ */
bool        amcanbuildparallel;
/* AM 是否支持 INCLUDE 子句指定的内含列？ */
bool        amcaninclude;
/* AM 是否使用 maintenance_work_mem？ */
bool        amusemaintenanceworkmem;
/* AM 是否对元组进行摘要，且一个摘要至少涵盖
 * 块中的所有元组？ */
bool        amsummarizing;
/* 并行清理标志的 OR 组合 */
uint8       amparallelvacuumoptions;
/* 索引中存储的数据类型；如果可变，则为 InvalidOid */
Oid         amkeytype;

/* 接口函数 */
ambuild_function ambuild;
ambuildempty_function ambuildempty;
aminsert_function aminsert;
aminsertcleanup_function aminsertcleanup;   /* 可以为 NULL */
ambulkdelete_function ambulkdelete;
amvacuumcleanup_function amvacuumcleanup;
amcanreturn_function amcanreturn;   /* 可以为 NULL */
amcostestimate_function amcostestimate;
amgettreeheight_function amgettreeheight;   /* 可以为 NULL */
amoptions_function amoptions;
amproperty_function amproperty;     /* 可以为 NULL */
ambuildphasename_function ambuildphasename;   /* 可以为 NULL */
amvalidate_function amvalidate;
amadjustmembers_function amadjustmembers; /* 可以为 NULL */
ambeginscan_function ambeginscan;
amrescan_function amrescan;
amgettuple_function amgettuple;     /* 可以为 NULL */
amgetbitmap_function amgetbitmap;   /* 可以为 NULL */
amendscan_function amendscan;
ammarkpos_function ammarkpos;       /* 可以为 NULL */
amrestrpos_function amrestrpos;     /* 可以为 NULL */

/* 支持并行索引扫描的接口函数 */
amestimateparallelscan_function amestimateparallelscan;    /* 可以为 NULL */
aminitparallelscan_function aminitparallelscan;    /* 可以为 NULL */
```

```
    amparallelrescan_function amparallelrescan;     /* 可以为 NULL */

    /* 支持规划的接口函数 */
    amtranslate_strategy_function amtranslatestrategy;  /* 可以为 NULL */
    amtranslate_cmptype_function amtranslatecmptype;    /* 可以为 NULL */
} IndexAmRoutine;
```

要使一种索引访问方法真正可用，还必须在 pg_opfamily、pg_opclass、pg_amop 和 pg_amproc 中定义一个或多个操作符族和操作符类。这些条目使规划器能够确定该访问方法的索引可以使用哪些查询限定条件。关于操作符族和操作符类的说明见第 38.15 节；阅读本章前应先掌握这些内容。

单个索引由两个系统目录条目共同定义：一个 pg_class 条目把它描述为一个物理关系，另一个 pg_index 条目给出索引的逻辑内容，也就是它包含哪些索引列，以及由相关操作符类定义的这些列的语义。索引列（键值）既可以是底层表的简单列，也可以是基于表行的表达式。索引访问方法通常并不关心索引键值来自何处（它拿到的总是预先计算好的键值），但会非常关心 pg_index 中的操作符类信息。这两个目录条目都可以作为传递给索引上所有操作的 Relation 数据结构的一部分来访问。

IndexAmRoutine 的某些标志字段还有一些不那么直观的含义。amcanunique 的要求见第 61.5 节。amcanmulticol 标志表明该访问方法支持多键列索引，而 amoptionalkey 表明它允许在第一索引列上没有给出可索引限制子句时进行扫描。当 amcanmulticol 为假时，amoptionalkey 实质上表示该访问方法是否支持没有任何限制子句的全索引扫描。支持多个索引列的访问方法必须支持在第一列之后省略任意一个或全部列限制条件的扫描；但它们可以要求第一索引列必须出现某种限制条件，这通过把 amoptionalkey 设为假来表示。索引 AM 可能把 amoptionalkey 设为假的一个原因，是它不索引 NULL 值。由于大多数可索引操作符都是严格的（strict），因此对 NULL 输入不可能返回 true，所以乍看之下不存储 NULL 值的索引条目似乎很有吸引力：无论如何，这些条目似乎都不可能被索引扫描返回。然而，当某个索引扫描对给定索引列没有限制子句时，这个论证就不成立了。实践中这意味着，若索引把 amoptionalkey 设为真，就必须索引 NULL 值，因为规划器可能决定在完全没有扫描键的情况下使用这种索引。与此相关的另一个限制是，支持多个索引列的索引访问方法必须支持对第一列之后各列中的 NULL 值建立索引，因为规划器会假定该索引可用于不限制这些列的查询。例如，考虑一个在 (a,b) 上的索引以及查询 WHERE a = 4。系统会假定该索引可用于扫描满足 a = 4 的行；如果索引省略了 b 为 NULL 的行，这个假定就是错误的。不过，省略第一索引列为 NULL 的行是可以的。对 NULL 值建立索引的索引访问方法还可以设置 amsearchnulls，表示它支持把 IS NULL 和 IS NOT NULL 子句作为搜索条件。

amcaninclude 标志指示该访问方法是否支持“内含”列，也就是说，它是否能在键列之外额外存储其他列而不对其进行处理。上一段中的要求只适用于键列。特别是，amcanmulticol =false 且 amcaninclude=true 的组合是合理的：这表示只能有一个键列，但仍然可以有内含列。此外，不论 amoptionalkey 的取值如何，内含列都必须允许为 NULL。

61.2. 索引访问方法函数

索引访问方法必须在 IndexAmRoutine 中提供如下索引构建和维护函数：

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

构建新索引。索引关系已在物理上创建，但内容为空。必须填入访问方法所需的固定数据，以及表中所有已有元组对应的条目。通常，ambuild 函数会调用 IndexBuildHeapScan() 扫描表中现有的元组，并计算需要插入索引的键。函数必须返回一个通过 palloc 分配的结构体，其中包含新索引的统计信息。

```
void
ambuildempty (Relation indexRelation);
```

构建一个空索引，并将其写入给定关系的初始化分支（INIT_FORKNUM）。只有不记录 WAL 的索引才会调用此方法；写入初始化分支的空索引会在每次服务器重启时复制到主关系分支上。

```
bool
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
          Relation heapRelation,
          IndexUniqueCheck checkUnique,
          bool indexUnchanged,
          IndexInfo *indexInfo);
```

向现有索引中插入一个新元组。values 和 isnull 数组给出要建立索引的键值，而 heap_tid 是要建立索引的 TID。如果该访问方法支持唯一索引（其 amcanunique 标志为真），那么 checkUnique 指示要执行的唯一性检查类型。这取决于唯一约束是否可延迟；详见第 61.5 节。通常，访问方法只会在执行唯一性检查时需要 heapRelation 参数（因为这时它必须到堆中核实元组是否仍然存活）。

该函数返回的布尔值只有在 checkUnique 为 UNIQUE_CHECK_PARTIAL 时才有意义。在这种情况下，返回真表示新条目已知是唯一的，而返回假表示它可能不是唯一的（因此必须安排一次延迟的唯一性重检）。对于其他情况，建议恒定返回假。

某些索引可能不会为所有元组建立索引。如果某个元组不应被索引，aminsert 应当直接返回而不做任何事。

如果索引 AM 希望在一条 SQL 语句中的连续索引插入之间缓存数据，它可以在 indexInfo->ii_Context 中分配空间，并把数据指针存入 indexInfo->ii_AmCache（该指针初始为 NULL）。

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

从索引中删除元组。这是一个“批量删除”操作，旨在通过扫描整个索引并检查每个条目是否应被删除来实现。必须调用传入的 `callback` 函数，其调用形式为 `callback(`*`TID`*`, callback_state) returns bool`，以确定由其引用的 TID 所标识的某个索引条目是否应删除。该函数必须返回 NULL，或者返回一个通过 palloc 分配的结构体，其中包含此次删除操作影响的统计信息。如果不需要向 `amvacuumcleanup` 传递信息，返回 NULL 也是可以的。

由于 `maintenance_work_mem` 有限，当待删除的元组很多时，`ambulkdelete` 可能需要被调用多次。参数 `stats` 是此前对该索引上一次调用的结果（在一次 `VACUUM` 操作中的第一次调用时它为 NULL）。这使得 AM 可以在整个操作过程中累积统计信息。通常，如果传入的 `stats` 非 NULL，`ambulkdelete` 会修改并返回同一个结构体。

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);
```

在一次 `VACUUM` 操作（零次或多次 `ambulkdelete` 调用）之后执行清理。它不一定要做返回索引统计信息之外的事情，但也可能执行批量清理，例如回收空索引页面。`stats` 是最后一次 `ambulkdelete` 调用返回的结果；如果因为没有需要删除的元组而没有调用 `ambulkdelete`，则为 NULL。如果结果不是 NULL，它必须是一个通过 palloc 分配的结构体。其中的统计信息将用于更新 `pg_class`，并在指定了 `VERBOSE` 时由 `VACUUM` 报告。如果索引在整个 `VACUUM` 操作期间完全没有变化，返回 NULL 也是可以的；否则应返回正确的统计信息。

从 PostgreSQL 8.4 开始，`amvacuumcleanup` 也会在 `ANALYZE` 操作完成时被调用。在这种情况下，`stats` 总是 NULL，而且任何返回值都会被忽略。这种情况可以通过检查 `info->analyze_only` 来区分。我们建议访问方法在这种调用中除了做插入后的清理外什么都不做，而且只在自动清理工作进程中这样做。

```
bool
amcanreturn (Relation indexRelation, int attno);
```

检查索引能否通过返回给定列原始的被索引值，在该列上支持仅索引扫描。属性编号从 1 开始，也就是说第一列的 attno 为 1。若支持则返回真，否则返回假。如果支持内含列，这个函数应始终对内含列返回真，因为不能取回的内含列几乎没有意义。如果访问方法根本不支持仅索引扫描，那么它的 `IndexAmRoutine` 结构体中的 `amcanreturn` 字段可以设为 NULL。

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

估计一次索引扫描的代价。该函数将在后面的第 61.6 节中详细讨论。

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

解析并验证索引的 reloptions 数组。只有当该索引存在非 NULL 的 reloptions 数组时才会调用此函数。*reloptions* 是一个 `text` 数组，其中的条目形如 *name*=*value*。该函数应构造一个 `bytea` 值，并将其复制到索引 relcache 条目的 `rd_options` 字段中。这个 `bytea` 值中的数据内容由访问方法自行定义；大多数标准访问方法使用结构体 `StdRdOptions`。当 *validate* 为真时，若存在未识别选项或无效取值，函数应报告合适的错误消息；当 *validate* 为假时，无效条目应被静默忽略。（当装载已存储在 `pg_catalog` 中的选项时，*validate* 为假；此时只有在访问方法改变了选项规则时才可能发现无效条目，而忽略过时条目是合适的。）如果希望采用默认行为，返回 NULL 也是可以的。

```
bool
amproperty (Oid index_oid, int attno,
            IndexAMProperty prop, const char *propname,
            bool *res, bool *isnull);
```

该 `amproperty` 方法允许索引访问方法覆盖 `pg_index_column_has_property` 及其相关函数的默认行为。如果访问方法对索引属性查询没有任何特殊行为，那么`amproperty` 字段在其 `IndexAmRoutine` 结构体中可以设为 NULL。否则，`amproperty` 方法会收到如下调用参数：*index_oid* 和 *attno* 均为零，对应`pg_indexam_has_property` 调用；或者 *index_oid* 有效且 *attno* 为零，对应`pg_index_has_property` 调用；或者 *index_oid* 有效且 *attno* 大于零，对应`pg_index_column_has_property` 调用。*prop* 是一个枚举值，用来标识当前测试的属性；*propname* 则是原始属性名字符串。如果核心代码不认识该属性名，那么*prop* 为 `AMPROP_UNKNOWN`。访问方法可以通过检查 *propname* 是否匹配来定义自定义属性名（使用 `pg_strcasecmp` 进行匹配，以与核心代码保持一致）；对于核心代码已知的名称，最好检查*prop*。如果 `amproperty` 方法返回 `true`，则表示它已确定属性测试结果：它必须将 `*res` 设为要返回的布尔值，或者把`*isnull` 设为 `true` 以返回 NULL。（所引用的两个变量在调用前都初始化为 `false`。）如果 `amproperty` 方法返回 `false`，核心代码就会按其正常逻辑确定属性测试结果。

支持排序操作符的访问方法应当实现 `AMPROP_DISTANCE_ORDERABLE` 属性测试，因为核心代码不知道如何完成该测试，只会返回 NULL。若完成该测试的代价低于打开索引并调用 `amcanreturn`（这正是核心代码的默认行为），那么实现 `AMPROP_RETURNABLE` 测试也可能是有利的。对于其他所有标准属性，默认行为应当已经足够。

```
bool
amvalidate (Oid opclassoid);
```

在访问方法能够合理验证的范围内，验证指定操作符类的目录条目。例如，可以检查是否提供了所有必需的支持函数。如果操作符类无效，amvalidate 函数必须返回 false。应通过 ereport 消息报告。

索引的目的当然是支持扫描那些匹配可索引 WHERE 条件的元组，这种条件常被称为限定词或扫描键。关于索引扫描的语义，将在下面的第 61.3 节中更详细地说明。一种索引访问方法可以支持“普通”索引扫描、“位图”索引扫描，或者两者都支持。索引访问方法必须或可以提供的扫描相关函数如下：

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             int norderbys);
```

为一次索引扫描做准备。nkeys 和 norderbys 参数表示扫描中将使用的限定条件和排序操作符数量，这些信息可能有助于空间分配。请注意，此时还没有提供扫描键的实际值。结果必须是一个通过 palloc 分配的结构体。出于实现上的原因，索引访问方法必须通过调用 RelationGetIndexScan() 来创建这个结构体。大多数情况下，ambeginscan 除了做这次调用以及也许获取一些锁之外，不会做太多工作；索引扫描启动中真正有意思的部分在 amrescan 中。

```
void
amrescan (IndexScanDesc scan,
          ScanKey keys,
          int nkeys,
          ScanKey orderbys,
          int norderbys);
```

开始或重新开始一次索引扫描，并且可以使用新的扫描键。（若要用之前传入的键重新开始，则为 keys 和/或 orderbys 传递 NULL。）请注意，键的数量或排序操作符的数量都不允许超过传给 ambeginscan 的数量。实践中，这种重启特性通常用于嵌套循环连接选中了新的外层元组，因此需要新的键比较值，但扫描键结构体本身保持不变的场景。

```
bool
amgettuple (IndexScanDesc scan,
            ScanDirection direction);
```

在给定扫描中取出下一个元组，并按给定方向移动（在索引中向前或向后）。若成功取得元组则返回真；若不再有匹配元组则返回假。在返回真时，元组的 TID 会存入 scan 结构体中。请注意，“成功”仅表示索引中存在一个匹配扫描键的条目，并不表示该元组一定仍然存在于堆中，或者一定能通过调用者的快照测试。成功时，amgettuple 还必须把 scan->xs_recheck 设为真或假。假表示可以确定该索引条目匹配扫描键；真表示这一点并不确定，因此在取到堆元组后必须根据扫描键所代表的条件重新检查它。这一机制支持“有损”索引操作符。请注意，重检只针对扫描条件；部分索引谓词（如果有）永远不会由 amgettuple 的调用者重新检查。

如果索引支持仅索引扫描（即其至少一列上 amcanreturn 返回真），那么成功时 AM 还必须检查 scan->xs_want_itup；若其为真，就必须返回该索引条目的原始被索引数据。对于 amcanreturn 返回假的列，可以把其值作为 NULL 返回。数据可以通过以下两

种形式之一返回：一种是把 IndexTuple 指针存入 scan->xs_itup，其元组描述符为 scan->xs_itupdesc；另一种是把 HeapTuple 指针存入 scan->xs_hitup，其元组描述符为 scan->xs_hitupdesc。（当重建出的数据可能装不进 IndexTuple 时，应使用后一种格式。）无论采用哪种形式，指针所引用数据的管理责任都在访问方法一侧。这份数据必须至少保持有效，直到该扫描下一次调用 amgettuple、amrescan 或 amendscan。

只有当访问方法支持“普通”索引扫描时，才需要提供 amgettuple 函数。如果不支持，其 IndexAmRoutine 结构体中的 amgettuple 字段必须设为 NULL。

```
int64
amgetbitmap (IndexScanDesc scan,
             TIDBitmap *tbm);
```

取出给定扫描中的所有元组，并将其加入调用者提供的 TIDBitmap 中（也就是把这组元组 ID 与位图中已有的集合做 OR）。返回值是取得的元组数量（这可能只是近似计数，例如某些 AM 不会检测重复项）。在把元组 ID 插入位图时，amgetbitmap 可以指出某些具体的元组 ID 需要重新检查扫描条件。这类似于 amgettuple 的 xs_recheck 输出参数。注意：在当前实现中，对这一特性的支持与位图自身的有损存储支持混在一起，因此调用者会对需要重检的元组同时重新检查扫描条件和部分索引谓词（如果有）。不过，这并不一定永远如此。amgetbitmap 和 amgettuple 不能在同一次索引扫描中同时使用；使用 amgetbitmap 时还有其他限制，详见第 61.3 节。

只有当访问方法支持“位图”索引扫描时，才需要提供 amgetbitmap 函数。如果不支持，其 IndexAmRoutine 结构体中的 amgetbitmap 字段必须设为 NULL。

```
void
amendscan (IndexScanDesc scan);
```

结束一次扫描并释放资源。scan 结构体本身不应被释放，但必须释放访问方法内部获取的所有锁、解除所有钉住状态，并释放由 ambeginscan 和其他扫描相关函数分配的其他内存。

```
void
ammarkpos (IndexScanDesc scan);
```

标记当前扫描位置。访问方法只需要为每次扫描支持一个被记住的扫描位置。

只有当访问方法支持有序扫描时，才需要提供 ammarkpos 函数。如果不支持，其 IndexAmRoutine 结构体中的 ammarkpos 字段可以设为 NULL。

```
void
amrestrpos (IndexScanDesc scan);
```

将扫描恢复到最近一次标记的位置。

只有当访问方法支持有序扫描时，才需要提供 amrestrpos 函数。如果不支持，其 IndexAmRoutine 结构体中的 amrestrpos 字段可以设为 NULL。

除了支持普通索引扫描外，某些类型的索引还可能希望支持并行索引扫描，这允许多个后端协同执行一次索引扫描。索引访问方法应安排好实现方式，使每个参与协作的进程返回普通非并行索引扫描本应返回的元组集合的一个子集，而且这些子集的并集应等于普通非并行索引扫描的结果

集。此外，虽然并行扫描返回的元组不必具有全局顺序，但每个协作后端内部返回的那个子集的顺序必须匹配请求的顺序。可以实现下列函数来支持并行索引扫描：

```
Size
amestimateparallelscan (void);
```

估算并返回访问方法执行并行扫描所需的动态共享内存字节数。（此数值是额外需要的空间，不包括以下结构体中与访问方法无关的数据所需的空间：`ParallelIndexScanDescData`。）

对于不支持并行扫描，或者所需额外存储字节数为零的访问方法，无需实现此函数。

```
void
aminitparallelscan (void *target);
```

在并行扫描开始时，会调用这个函数来初始化动态共享内存。*`target`* 会指向一块动态共享内存，其大小至少等于此前 `amestimateparallelscan` 返回的字节数，而该函数可以用这部分空间存储它想保存的任何数据。

对于不支持并行扫描，或者虽需要共享内存但无需初始化的情况，无需实现此函数。

```
void
amparallelrescan (IndexScanDesc scan);
```

如果实现了此函数，当并行索引扫描必须重新开始时就会调用它。它应重置由 `aminitparallelscan` 建立的任何共享状态，以便扫描从头开始重新执行。

61.3. 索引扫描

在索引扫描中，索引访问方法负责返回它所知道的、匹配扫描键的所有元组的 TID。访问方法不负责从索引父表中实际取出这些元组，也不负责判断它们是否能通过扫描的可见性测试或其他条件。

扫描键是如下形式的 `WHERE` 子句的内部表示：*`index_key operator constant`*。其中，索引键是该索引的一列，而操作符是与该索引列关联的操作符族成员之一。一次索引扫描可以有零个或多个扫描键，这些扫描键会被隐式地以 AND 连接，也就是说，返回的元组预期应满足所有给定条件。

对于某个特定查询，访问方法可以报告该索引是有损的，或者要求重新检查。这意味着索引扫描会返回所有通过扫描键的条目，并可能额外返回一些未通过扫描键的条目。随后，核心系统的索引扫描机制会再次把索引条件应用到堆元组上，以验证它是否真的应被选中。如果没有指定重检选项，索引扫描就必须精确返回匹配条目的集合。

请注意，确保正确找出通过全部给定扫描键的所有且仅有的条目，这项工作完全由访问方法负责。此外，核心系统只是简单地把所有与索引键和操作符族匹配的 `WHERE` 子句交给访问方法，而不会做任何语义分析来判断它们是否冗余或互相矛盾。例如，给定 `WHERE x > 4 AND x > 14`，其中 `x` 是一个 B-树索引列，那么由 B-树的 `amrescan` 函数负责识别第一个扫描键是冗余的并可被丢弃。`amrescan` 期间需要做多少预处理，取决于索引访问方法需要在多大程度上把扫描键归约成一种“正规化”形式。

某些访问方法会按明确定义的顺序返回索引条目，另一些则不会。实际上，访问方法支持有序输出有两种不同方式：

- 总是按数据自然顺序返回条目的访问方法（例如 btree）应把 amcanorder 设为真。目前，这类访问方法必须对其等值和排序操作符使用与 btree 兼容的策略号。
- 支持排序操作符的访问方法应把 amcanorderbyop 设为真。这表示索引能够按满足 ORDER BY *index_key operator constant* 的顺序返回条目。前面已经提到，这种形式的扫描修饰符可以传给 amrescan。

amgettuple 函数有一个 direction 参数，它可以是 ForwardScanDirection（通常情况）或 BackwardScanDirection。如果 amrescan 之后的第一次调用指定了 BackwardScanDirection，那么匹配条件的索引条目集合就要按从后到前的方向扫描，而不是通常的从前到后，因此 amgettuple 必须返回索引中最后一个匹配元组，而不是通常情况下的第一个。（这只会发生在把 amcanorder 设为真的访问方法上。）第一次调用之后，amgettuple 必须准备好从最近一次返回条目的位置继续按任一方向推进扫描。（但如果 amcanbackward 为假，则后续所有调用的方向都必须与第一次相同。）

支持有序扫描的访问方法必须支持在扫描中“标记”一个位置，并在之后返回到该标记位置。同一个位置可能会被恢复多次。不过，每次扫描只需要记住一个位置；新的 ammarkpos 调用会覆盖先前标记的位置。不支持有序扫描的访问方法无须在 IndexAmRoutine 中提供 ammarkpos 和 amrestrpos 函数；把这些指针设为 NULL 即可。

无论是扫描位置还是标记位置（如果有），在面对索引中的并发插入或删除时都必须保持一致。如果某个新插入的条目没有被一次扫描返回，而假如该条目在扫描开始前就已存在则本应被返回，这也是可以接受的；同样，即使某个条目在第一次扫描通过时未被返回，后续重新扫描或反向扫描时返回它也是可以接受的。类似地，并发删除也可能会或者不会反映在扫描结果中。重要的是，插入或删除本身不能导致扫描漏掉，或者重复返回那些本身并未被插入或删除的条目。

如果索引存储的是原始被索引数据值（而不是它们的某种有损表示），那么它就适合支持仅索引扫描，在这种扫描中索引返回的是实际数据，而不只是堆元组的 TID。只有当可见性映射显示该 TID 位于一个全部可见的页面上时，这才能避免 I/O；否则仍然必须访问堆元组来检查 MVCC 可见性。不过，这并不是访问方法需要关心的事。

除了使用 amgettuple 外，索引扫描也可以使用 amgetbitmap，在一次调用中取出所有元组。与 amgettuple 相比，这可能明显更高效，因为它可以避免访问方法内部的反复加锁和解锁。原则上，amgetbitmap 应具有与重复调用 amgettuple 相同的效果，但为了简化实现，我们施加了若干限制。首先，amgetbitmap 一次性返回全部元组，因此不支持标记或恢复扫描位置。其次，元组是通过一个位图返回的，没有特定顺序，这也是为什么 amgetbitmap 不带 direction 参数的原因。（这种扫描也永远不会提供排序操作符。）此外，使用 amgetbitmap 时也没有仅索引扫描的支持，因为没有办法返回索引元组的内容。最后，amgetbitmap 不保证对返回的元组做任何锁定，其影响见第 61.4 节。

请注意，如果访问方法的内部实现并不适合某一个 API，那么它可以只实现 amgetbitmap 而不实现 amgettuple，反之亦然。

61.4. 索引锁定注意事项

索引访问方法必须处理多个进程并发更新同一个索引的情况。PostgreSQL 核心系统会在索引扫描期间对索引获取 AccessShareLock，并在更新索引时（包括普通的 VACUUM）获取 RowExclusiveLock。由于这些锁类型彼此不冲突，所以访问方法必须自行处理它可能需要的任何细粒度锁定。对整个索引的 ACCESS EXCLUSIVE 锁只会在索引创建、销毁或 REINDEX 期间获取。

构建一种支持并发更新的索引类型，通常需要对所需行为进行大量而细致的分析。对于 B-树和 hash 索引类型，可以阅读 `src/backend/access/nbtree/README` 和 `src/backend/access/hash/README` 中涉及的设计决策。

除了索引自身内部一致性的要求外，并发更新还会带来父表（即堆）与索引之间一致性的问题。由于 PostgreSQL 把对堆的访问和更新与对索引的访问和更新分离开来，因此存在一些窗口期，在这些窗口期内索引可能与堆不一致。我们通过以下规则处理这个问题：

- 先创建新的堆条目，再创建它对应的索引条目。（因此并发索引扫描很可能看不到这个堆条目。这是可以接受的，因为索引读取者本来也不会关心未提交的行。但也请见第 61.5 节。）
- 当某个堆条目将要被删除（由 `VACUUM` 执行）时，必须先删除它的所有索引条目。
- 索引扫描必须在保存 `amgettuple` 最近一次返回条目的索引页面上保持钉住状态，而 `ambulkdelete` 不能从被其他后端钉住的页面中删除条目。下面会解释为什么需要这条规则。

如果没有第三条规则，索引读取者就有可能在某个索引条目被 `VACUUM` 删除之前先看到它，然后在对应的堆条目已被 `VACUUM` 删除之后才到达那里。如果读取者到达时，该项号仍未被重新使用，就不会造成严重问题，因为空项槽位会被 `heap_fetch()` 忽略。但如果第三个后端已经把这个项槽位重新用作别的东西呢？在使用 MVCC 兼容快照时不会有问题，因为该槽位的新占用者一定太新，无法通过快照测试。然而，对于非 MVCC 兼容的快照（例如 `SnapshotAny`），就可能错误地接受并返回一行实际上并不匹配扫描键的数据。我们可以要求在所有情况下都对堆行重新检查扫描键，以防范这种情形，但那样代价太高。于是，我们改为把索引页面的钉住状态作为一种代理，表示读取者从索引条目到匹配堆条目的访问过程可能仍“在进行中”。让 `ambulkdelete` 因这种钉住状态而阻塞，能够保证 `VACUUM` 不会在读取者处理完之前删除对应的堆条目。这种方案运行时开销很小，只有在真正发生冲突的少数情况下才会增加阻塞开销。

这种解决方案要求索引扫描是“同步的”：我们必须在扫描到相应索引条目之后立刻取出对应的堆元组。由于多种原因，这会比较昂贵。相反，“异步”扫描可以先从索引中收集许多 TID，再在稍后某个时刻访问堆元组，这样索引锁定的开销要小得多，也能实现更高效的堆访问模式。根据前面的分析，对于非 MVCC 兼容快照，我们必须采用同步方式；而对于使用 MVCC 快照的查询，异步扫描则是可行的。

在 `amgetbitmap` 索引扫描中，访问方法不会为任何返回的元组保持索引页面的钉住状态。因此，只有把这种扫描与 MVCC 兼容的快照一起使用才是安全的。

当未设置 `ampredlocks` 标志时，在可串行化事务中使用该索引访问方法的任何扫描，都会在整个索引上获取一个非阻塞谓词锁。这会与并发可串行化事务向该索引插入任何元组形成读写冲突。如果在一组并发可串行化事务之间检测到某些特定模式的读写冲突，为了保护数据完整性，其中某个事务可能会被取消。设置该标志则表示该索引访问方法实现了更细粒度的谓词锁，这通常会降低这类事务取消的频率。

61.5. 索引唯一性检查

PostgreSQL 使用唯一索引来强制执行 SQL 唯一性约束；所谓唯一索引，就是不允许存在多个具有相同键值条目的索引。支持这一特性的访问方法会把 `amcanunique` 设为真。（目前只有 B-树支持这一点。）在强制唯一性时，不会考虑 `INCLUDE` 子句中列出的列。

由于 MVCC 的存在，索引中在物理上总是必须允许存在重复条目：这些条目可能指向同一个逻辑行的连续版本。我们真正想强制的行为是，任何 MVCC 快照都不能同时包含两行拥有相同索引键的记录。在向唯一索引插入一行新数据时，必须检查以下几种情况：

- 如果某一条冲突的有效行已被当前事务删除，那么这是允许的。（特别是，因为一次 UPDATE 总是在插入新版本前删除旧版本，这就允许对某一行执行不改变键值的 UPDATE。）

- 如果某条冲突的行是由一个尚未提交的事务插入的，那么当前准备插入的事务必须等待，看看那个事务是否提交。如果它回滚，就不存在冲突；如果它提交并且没有再次删除那条冲突行，就发生了唯一性违背。（实际上，我们只是等待另一个事务结束，然后把可见性检查整个重新做一遍。）

- 类似地，如果某条冲突的有效行是由一个尚未提交的事务删除的，那么当前准备插入的事务必须等待该事务提交或中止，然后重新执行测试。

此外，就在按照上述规则报告唯一性违背之前，访问方法必须重新检查正在插入那一行的存活性。如果它已经是提交后死亡的状态，就不应报告违背。（这种情况不会出现在插入由当前事务刚创建的行这一普通场景中，但在 `CREATE UNIQUE INDEX CONCURRENTLY` 期间却可能发生。）

我们要求索引访问方法自行应用这些测试，这意味着它必须深入堆中检查那些根据索引内容显示为具有重复键的行的提交状态。毫无疑问，这样做既丑陋又不够模块化，但它避免了重复工作：如果我们单独再做一次探测，那么在寻找新行索引条目插入位置时，查找冲突行的索引搜索实际上就会被重复执行。更何况，除非把冲突检查作为插入新索引条目动作的一个组成部分，否则也没有明显的方法可以避免竞争条件。

如果唯一约束是可延迟的，情况会更复杂：我们需要能够为新行插入一个索引条目，但把任何唯一性违背错误延迟到语句结束时甚至更晚才报告。为了避免对索引进行不必要的重复搜索，索引访问方法应在初始插入期间执行一次初步唯一性检查。如果这表明确实不存在冲突的存活元组，那么事情就结束了。否则，我们会安排在真正强制约束时再做一次重检。若在重检时，插入的元组与另外某个具有相同键值的元组都仍然存活，就必须报告错误。（注意，就此用途而言，“存活”实际上是指“索引条目 HOT 链中至少有一个元组是存活的”。）为实现这一点，传给 `aminsert` 函数的 `checkUnique` 参数会取以下值之一：

- `UNIQUE_CHECK_NO` 表示不应执行唯一性检查（这不是唯一索引）。

- `UNIQUE_CHECK_YES` 表示这是一个不可延迟的唯一索引，必须按前述方式立即执行唯一性检查。

- `UNIQUE_CHECK_PARTIAL` 表示该唯一约束是可延迟的。PostgreSQL 会使用此模式为每一行插入索引条目。访问方法必须允许索引中出现重复条目，并通过让 `aminsert` 返回假来报告任何潜在的重复。对于每一行返回假的情况，系统都会安排一次延迟重检。

  访问方法必须识别任何可能违反唯一约束的行，但报告误报并不算错误。这使得检查可以在不等待其他事务结束的情况下完成；此处报告的冲突不会被当作错误处理，而会在之后重新检查，到那时它们可能已不再冲突。

- `UNIQUE_CHECK_EXISTING` 表示这是对某一行的延迟重检，此前该行被报告为可能违反唯一性。虽然这同样是通过调用 `aminsert` 来实现，但访问方法在这种情况下不得插入新的索引条目。该索引条目已经存在。相反，访问方法必须检查是否存在另一个存活的索引条目；如果存在，并且目标行本身也仍然存活，就应报告错误。

  建议在一次 `UNIQUE_CHECK_EXISTING` 调用中，访问方法进一步验证目标行在索引中确实已有现存条目；若没有，就报告错误。这样做是个好主意，因为传给 `aminsert` 的索引元组值会被重新计算。如果索引定义涉及并非真正 immutable 的函数，我们就可能在索引的错误区域上做检查。确认重检时确实找到了目标行，可以验证我们扫描的仍是原始插入时使用的同一组元组值。

61.6. 索引代价估算函数

`amcostestimate` 函数会收到描述某种可能索引扫描方式的信息，其中包括已经确定可用于该索引的 WHERE 子句和 ORDER BY 子句列表。它必须返回访问该索引的代价估算，以及

WHERE 子句选择率的估计值（也就是在索引扫描期间将从父表中检索出的行所占比例）。对于简单情况，代价估算器几乎所有工作都可以通过调用优化器中的标准例程来完成；之所以提供 amcostestimate 函数，是为了让索引访问方法能够提供与索引类型有关的专门知识，以便在可能时改进标准估计。

每个 amcostestimate 函数都必须具有如下签名：

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

前三个参数是输入参数：

root

规划器关于当前正在处理查询的信息。

path

当前正在考虑的索引访问路径。除代价和选择率字段外，其余字段都有效。

loop_count

在代价估算中应计入的索引扫描重复次数。当考虑在嵌套循环连接内部使用参数化扫描时，这个参数通常会大于 1。请注意，代价估算仍应只针对一次扫描；更大的 *loop_count* 只表示可以适当考虑多次扫描之间的一些缓存效应。

最后五个参数是按引用传递的输出参数：

**indexStartupCost*

设为索引启动处理的代价。

**indexTotalCost*

设为索引处理的总代价。

**indexSelectivity*

设为索引选择率。

**indexCorrelation*

设为索引扫描顺序与底层表顺序之间的相关系数。

**indexPages*

设为索引叶子页面数量。

请注意，代价估算函数必须用 C 编写，而不能用 SQL 或任何可用的过程语言，因为它们必须访问规划器/优化器的内部数据结构。

索引访问代价应使用 `src/backend/optimizer/path/costsize.c` 所采用的参数来计算：顺序磁盘块读取的代价为 `seq_page_cost`，非顺序读取的代价为 `random_page_cost`，处理一条索引行的代价通常应取为 `cpu_index_tuple_cost`。此外，在索引处理期间调用的任何比较操作符（特别是对 indexquals 本身的求值）都应计入适当倍数的 `cpu_operator_cost`。

访问代价应包括与扫描索引本身有关的全部磁盘和 CPU 代价，但不包括取出或处理由索引标识出的父表行的代价。

“启动代价”是整个扫描总代价中必须在开始取第一行之前先付出的那一部分。对大多数索引来说，这可以视为零；但启动代价较高的索引类型可能希望把它设为非零。

`indexSelectivity` 应设为在索引扫描期间将从父表中检索出的行的估计比例。对于有损查询，这个值通常会高于实际通过给定限定条件的行比例。

`indexCorrelation` 应设为索引顺序与表顺序之间的相关性（范围从 -1.0 到 1.0）。该值用于调整从父表取行代价的估计。

`indexPages` 应设为叶子页面数量。它会被用来估算并行索引扫描所需工作进程的数量。

当 *`loop_count`* 大于 1 时，返回的数字应是该索引任意一次扫描的期望平均值。

代价估算

一个典型的代价估算器会按如下步骤进行：

1. 基于给定的限定条件，估计并返回将被访问的父表行比例。如果没有任何与索引类型相关的专门知识，可以使用优化器的标准函数 `clauselist_selectivity()`：

```
*indexSelectivity = clauselist_selectivity(root, path->indexquals,
                                           path->indexinfo->rel->relid,
                                           JOIN_INNER, NULL);
```

2. 估计扫描期间将访问的索引行数。对许多索引类型来说，这等于 *`indexSelectivity`* 乘以索引中的行数，但也可能更多。（请注意，索引的页面数和行数可以从 `path->indexinfo` 结构体中取得。）

3. 估计扫描期间将读取的索引页面数。它可能仅仅是 *`indexSelectivity`* 乘以索引总页面数。

4. 计算索引访问代价。一个通用估计器可能会这样做：

```
/*
 * 通用假设是索引页面将按顺序读取，
 * 因此每页代价为 seq_page_cost，而非 random_page_cost。
 * 此外，还要计入在每个索引行上对 indexquals 求值的代价。
 * 假定所有代价都在扫描过程中逐步付出。
 */
cost_qual_eval(&index_qual_cost, path->indexquals, root);
*indexStartupCost = index_qual_cost.startup;
*indexTotalCost = seq_page_cost * numIndexPages +
    (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

   不过，上述做法没有考虑重复索引扫描之间索引读取的摊销效果。

5. 估计索引的相关性。对于单列上的简单有序索引，这个值可以从 pg_statistic 中取得。如果相关性未知，保守估计应为零（即无相关性）。

代价估算器函数的示例可在 `src/backend/utils/adt/selfuncs.c` 中找到。

第 62 章 通用 WAL 记录

虽然所有内置的、会生成 WAL 记录的模块都有各自的 WAL 记录类型，但也有一种通用 WAL 记录类型，可以用通用方式描述对页面的更改。这对于提供自定义访问方法的扩展很有用，因为它们无法注册自己的 WAL 重做例程。

用于构造通用 WAL 记录的 API 定义在 `access/generic_xlog.h` 中，并在`access/transam/generic_xlog.c` 中实现。

要使用通用 WAL 记录机制执行一次需要写入 WAL 的数据更新，请遵循以下步骤：

1. `state = GenericXLogStart(relation)` — 开始为给定关系构造一条通用 WAL 记录。
2. `page = GenericXLogRegisterBuffer(state, buffer, flags)` — 注册一个将在当前通用 WAL 记录中被修改的缓冲区。该函数返回一个指向该缓冲区页面临时副本的指针，应在该副本上进行修改（不要直接修改缓冲区内容）。第三个参数是适用于该操作的标志掩码。目前唯一这样的标志是`GENERIC_XLOG_FULL_IMAGE`，表示应在 WAL 记录中包含整页镜像，而不是增量更新。通常在页面是新的，或者已经被完全重写时，会设置该标志。如果该操作需要修改多个页面，则可以重复调用`GenericXLogRegisterBuffer`。
3. 对上一步得到的页面镜像施加修改。
4. `GenericXLogFinish(state)` — 将更改应用到缓冲区，并发出通用 WAL 记录。

在上述任意步骤之间，都可以调用`GenericXLogAbort(state)` 取消 WAL 记录的构造。这会丢弃对页面镜像副本所做的全部更改。

在使用通用 WAL 记录功能时请注意以下几点：

- 不允许直接修改缓冲区！所有修改都必须在通过 `GenericXLogRegisterBuffer()` 取得的副本上完成。换句话说，生成通用 WAL 记录的代码绝不应自行调用`BufferGetPage()`。不过，在适当的时机对缓冲区执行钉住/解除钉住以及加锁/解锁，仍然是调用者的责任。对于每个目标缓冲区，从调用`GenericXLogRegisterBuffer()` 之前开始直到 `GenericXLogFinish()` 之后，都必须持有排他锁。
- 注册缓冲区（步骤 2）和修改页面镜像（步骤 3）可以自由交错进行，也就是说，这两个步骤可以按任意顺序重复。请记住，注册缓冲区的顺序应当与重放时获取其锁的顺序一致。
- 一个通用 WAL 记录最多能注册 `MAX_GENERIC_XLOG_PAGES` 个缓冲区。如果超过这个限制，就会抛出错误。
- 通用 WAL 假定待修改的页面具有标准布局，尤其是假定`pd_lower` 和 `pd_upper` 之间没有有用数据。
- 由于修改的是缓冲区页面的副本，`GenericXLogStart()` 不会开启临界区。因此，在 `GenericXLogStart()` 和 `GenericXLogFinish()` 之间，可以安全地进行内存分配、抛出错误等操作。唯一真正的临界区位于`GenericXLogFinish()` 内部。此外，在错误退出时也无需调用`GenericXLogAbort()`。
- `GenericXLogFinish()` 会负责将缓冲区标记为脏并设置它们的 LSN。你不需要显式执行这些操作。
- 对于不记录 WAL 的关系，其他行为都完全相同，只是不会真正发出 WAL 记录。因此，通常不需要专门对不记录 WAL 的关系做任何显式检查。

- 通用 WAL 的重做函数会按照缓冲区注册的顺序获取这些缓冲区上的排他锁。在重做完所有更改后，这些锁也会按照同样的顺序释放。
- 如果某个已注册缓冲区未指定 GENERIC_XLOG_FULL_IMAGE，通用 WAL 记录中包含的就是旧页面镜像与新页面镜像之间的差异。该差异基于逐字节比较。对于在页面内移动数据的情况，这种表示方式并不十分紧凑，未来可能会改进。

第 63 章 B-树索引

63.1. 简介

PostgreSQL 包含标准 B-树（btree，即多路平衡树）索引数据结构的一种实现。凡是能够排成定义明确的线性顺序的数据类型，都可以使用 B-树索引。唯一的限制是，索引项在经过 TOAST 压缩（如果适用）后，大小也不能超过一个页面的大约三分之一。

由于每个 B-树操作符类都会为其数据类型施加一种排序顺序，B-树操作符类（更准确地说，是操作符族）已经成为 PostgreSQL 用来统一表示和理解排序语义的方式。因此，它们具备了一些超出单纯支持 B-树索引所需范围的特性，系统中某些与 B-树 AM 相距甚远的部分也会利用它们。

63.2. B-树操作符类的行为

正如表 38.2 所示，B-树操作符类必须提供五个比较操作符，<、<=、=、>= 和 >。人们可能会以为 <> 也应当属于操作符类的一部分，但事实并非如此，因为在索引搜索中使用 <> 的 WHERE 子句几乎毫无用处。（出于某些目的，规划器会把 <> 视为与 B-树操作符类相关联；但规划器是通过 = 操作符的求反器链接找到该操作符，而不是从 `pg_amop` 中找到它。）

当多种数据类型共享几乎相同的排序语义时，可以将它们的操作符类归入同一个操作符族。这样做的好处是，规划器就能够对跨类型比较作出推理。族中的每个操作符类都应当包含对应其输入数据类型的单类型操作符（以及相关支持函数），而跨类型比较操作符及其支持函数则作为该族中的“松散”成员存在。建议在族中包含一套完整的跨类型操作符，从而确保规划器能够表示它根据传递性推导出来的任意比较条件。

B-树操作符族必须满足一些基本假设：

- = 操作符必须是一种等价关系；也就是说，对于该数据类型的所有非 null 值 *A*、*B*、*C*：
  - *A* = *A* 为真（自反律）
  - 如果 *A* = *B*，则 *B* = *A*（对称律）
  - 如果 *A* = *B* 且 *B* = *C*，则 *A* = *C*（传递律）
- < 操作符必须是一种强排序关系；也就是说，对于所有非 null 值 *A*、*B*、*C*：
  - *A* < *A* 为假（非自反律）
  - 如果 *A* < *B* 且 *B* < *C*，则 *A* < *C*（传递律）
- 此外，该顺序还是全序的；也就是说，对于所有非 null 值 *A*、*B*：
  - *A* < *B*、*A* = *B* 以及 *B* < *A* 中恰有一个为真（三分律）
    （三分律当然也正是定义比较支持函数的依据。）

另外三个操作符都可以用 = 和 < 以显而易见的方式定义，并且必须与它们保持一致。

对于支持多种数据类型的操作符族，当 *A*、*B*、*C* 取自该族中的任意数据类型时，上述定律都必须成立。传递律最难保证，因为在跨类型情况下，它要求两种或三种不同操作符的行为彼此一致。举例来说，不能把 `float8` 和 `numeric` 放进同一个操作符族，至少在当前这种语义下不行：会先将 `numeric` 值转换为 `float8`，以便与 `float8` 比较。由于 `float8` 精度有限，这意味着会存在不同的 `numeric` 值都与同一个 `float8` 值比较为相等，于是传递律就会失效。

对于多数据类型操作符族，另一个要求是：在族内数据类型之间定义的任何隐式或二进制强制转换，都不得改变相应的排序顺序。

很容易理解为什么 B-树索引要求这些定律在单一数据类型内部成立：没有这些定律，就不存在可用于排列键的顺序。此外，使用不同数据类型比较键的索引搜索，也要求跨两种数据类型的比较行为合理一致。把这些要求扩展到一个族内的三种或更多数据类型，虽然并非 B-树索引机制本身的严格要求，但规划器会出于优化目的依赖它们。

63.3. B-树支持函数

如表 38.8 所示，B-树定义了一个必需支持函数和两个可选支持函数。

对于 B-树操作符族为其提供比较操作符的每一种数据类型组合，都必须提供一个比较支持函数。该函数在 pg_amproc 中注册，支持函数编号为 1，并且 amproclefttype/amprocrighttype 要等于该比较的左右数据类型（也就是与匹配操作符在 pg_amop 中注册时相同的数据类型）。比较函数必须接受两个非 null 值 *A* 和 *B*，并返回一个 int32 值，其值为 < 0、0 或 > 0，分别对应 *A* < *B*、*A* = *B* 或 *A* > *B*。不允许返回 null：该数据类型的所有值都必须可比较。示例见 src/backend/access/nbtree/nbtcompare.c。

如果参与比较的值属于支持排序规则的数据类型，则会通过标准的 PG_GET_COLLATION() 机制，把适当的排序规则 OID 传递给比较支持函数。

可选地，B-树操作符族可以提供排序支持函数，注册为支持函数编号 2。这些函数允许以比朴素地直接调用比较支持函数更高效的方式，实现用于排序的比较。相关 API 定义在 src/include/utils/sortsupport.h 中。

可选地，B-树操作符族可以提供 in_range 支持函数，注册为支持函数编号 3。这些函数不会在 B-树索引操作期间使用；相反，它们会扩展操作符族的语义，使其能够支持包含 RANGE *offset* PRECEDING 和 RANGE *offset* FOLLOWING 这两种帧边界类型的窗口子句（见第 4.2.8 节）。本质上，它额外提供的信息是：如何以与该族数据排序兼容的方式，对 *offset* 值进行加减。

in_range 函数必须具有如下签名：

```
in_range(val type1, base type1, offset type2, sub bool, less bool)
returns bool
```

val 和 *base* 必须具有相同类型，并且该类型必须是操作符族支持的类型之一（也就是该族为其提供排序的一种类型）。不过，*offset* 可以是另一种类型，而该类型甚至可能不被该族以其他方式支持。例如，内置的 time_ops 族就提供了一个 in_range 函数，其 *offset* 的类型为 interval。一个族可以提供 in_range 函数，分别对应其支持的任意类型以及一个或多个 *offset* 类型。每个 in_range 函数都应在 pg_amproc 中登记，其中 amproclefttype 等于 type1，而 amprocrighttype 等于 type2。

in_range 函数的核心语义取决于这两个布尔标志参数。它应当先对 *base* 与 *offset* 做加法或减法，再把 *val* 与结果比较，具体如下：

- 如果 !*sub* 且 !*less*，返回 *val* >= (*base* + *offset*)
- 如果 !*sub* 且 *less*，返回 *val* <= (*base* + *offset*)
- 如果 *sub* 且 !*less*，返回 *val* >= (*base* - *offset*)
- 如果 *sub* 且 *less*，返回 *val* <= (*base* - *offset*)

在此之前，函数应检查 *offset* 的符号：如果它小于零，就引发错误 ERRCODE_INVALID_PRECEDING_OR_FOLLOWING_SIZE (22013)，错误文本类似于 “invalid preceding or following size in window function” 。（这是 SQL 标准的要求，尽管非标准操作符族或许会选择忽略这一限制，因为从语义上看似乎并没有太大必要。）把这个要求委托给 in_range 函数，是为了让核心代码无需理解对于某个特定数据类型来说 “小于零” 究竟意味着什么。

另外还希望 in_range 函数在可行时避免因为 *base* + *offset* 或 *base* - *offset* 会溢出而抛出错误。即便该值会超出该数据类型的取值范围，仍然可以判定正确的比较结果。注意，如果该数据类型包含诸如 “infinity” 或 “NaN” 之类的概念，就可能需要格外小心，以确保 in_range 的结果与该操作符族的正常排序顺序一致。

in_range 函数的结果必须与操作符族施加的排序顺序保持一致。更精确地说，给定任意固定的 *offset* 和 *sub* 值，则：

- 如果 in_range 在 *less* = true 时，对某个 *val1* 和 *base* 返回真，那么对于每一个 *val2* <= *val1*，只要其 *base* 相同，它也必须返回真。
- 如果 in_range 在 *less* = true 时，对某个 *val1* 和 *base* 返回假，那么对于每一个 *val2* >= *val1*，只要其 *base* 相同，它也必须返回假。
- 如果 in_range 在 *less* = true 时，对某个 *val* 和 *base1* 返回真，那么对于每一个 *base2* >= *base1*，只要其 *val* 相同，它也必须返回真。
- 如果 in_range 在 *less* = true 时，对某个 *val* 和 *base1* 返回假，那么对于每一个 *base2* <= *base1*，只要其 *val* 相同，它也必须返回假。

当 *less* = false 时，也有条件相反的对应陈述成立。

如果被排序的类型（type1）属于支持排序规则的数据类型，则会通过标准的 PG_GET_COLLATION() 机制，把适当的排序规则 OID 传递给 in_range 函数。

in_range 函数不必处理 NULL 输入，并且通常会被标记为 strict。

63.4. 实现

可在 src/backend/access/nbtree/README 中找到对 B-树索引实现的介绍。

第 64 章 GiST 索引

64.1. 简介

GiST 是 Generalized Search Tree（通用搜索树）的缩写。它是一种平衡的树形访问方法，可作为实现任意索引方案的基本模板。B-树、R 树以及许多其他索引方案都可以在GiST 中实现。

GiST 的一个优点是，它使数据类型所属领域的专家而不是数据库专家，能够开发带有适当访问方法的自定义数据类型。

这里的一些内容来自加州大学伯克利分校的 GiST 索引项目网站[1]以及 Marcel Kornacker 的论文 Access Methods for Next-Generation Database Systems[2]。PostgreSQL 中的 GiST 实现主要由 Teodor Sigaev 和 Oleg Bartunov 维护，他们的网站[3]上还有更多信息。

64.2. 内置操作符类

PostgreSQL 核心发行版包含表 64.1 中所示的 GiST 操作符类。（附录 F 中描述的一些可选模块还提供了额外的 GiST 操作符类。）

表 64.1. 内置 GiST 操作符类

名称	被索引数据类型	可索引操作符	排序操作符
box_ops	box	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	
circle_ops	circle	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->
inet_ops	inet, cidr	&& >> >>= > >= <> << <<= < <= =	
point_ops	point	>> >^ << <@ <@ <@ <^ ~=	<->
poly_ops	polygon	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->
range_ops	任意范围类型	&& &> &< >> << <@ -|- = @> @>	
tsquery_ops	tsquery	<@ @>	
tsvector_ops	tsvector	@@	

出于历史原因，inet_ops 操作符类不是类型 inet 和 cidr 的默认类。要使用它，请在 CREATE INDEX 中写出该类名，例如

```
CREATE INDEX ON my_table USING GIST (my_inet_column inet_ops);
```

[1] http://gist.cs.berkeley.edu/
[2] http://www.sai.msu.su/~megera/postgres/gist/papers/concurrency/access-methods-for-next-generation.pdf.gz
[3] http://www.sai.msu.su/~megera/postgres/gist/

64.3. 可扩展性

传统上，实现一种新的索引访问方法意味着大量艰难的工作。必须理解数据库的内部机制，例如锁管理器和预写式日志。GiST 接口具有很高的抽象层次，只要求访问方法实现者实现被访问数据类型的语义。GiST 层本身会处理并发、日志记录以及树结构的搜索。

这种可扩展性不应与其他标准搜索树在可处理数据方面的可扩展性相混淆。例如，PostgreSQL 支持可扩展的 B-树和 hash 索引。这意味着你可以用 PostgreSQL 在任意数据类型上构建 B-树或 hash 索引。但 B-树只支持范围谓词（<、=、>），而 hash 索引只支持等值查询。

因此，如果你用PostgreSQL 的 B-树为一个图像集合建立索引，你只能发出诸如 “imagex 是否等于 imagey” 、“imagex 是否小于 imagey” 以及 “imagex 是否大于 imagey” 之类的查询。取决于你如何在这种上下文中定义 “等于” 、“小于” 和 “大于” ，这可能仍然有用。不过，使用基于GiST 的索引，你就可以构造出能够提出特定领域问题的查询方式，例如 “找出所有马的图片” 或者 “找出所有曝光过度的图片” 。

要让一个 GiST 访问方法运行起来，只需实现几个用户定义的方法，这些方法定义了树中键的行为。当然，要支持复杂查询，这些方法本身也必须足够巧妙；但对于所有标准查询（B-树、R 树等），它们都相对直接。简而言之，GiST 把可扩展性与通用性、代码复用以及清晰的接口结合了起来。

一个 GiST 索引操作符类必须提供五个方法，另外还有四个可选方法。通过正确实现`same`、`consistent` 和 `union` 方法可以保证索引的正确性，而索引的效率（大小与速度）则取决于 `penalty` 和 `picksplit` 方法。两个可选方法是 `compress` 和 `decompress`，它们允许索引的内部树数据使用与其所索引数据不同的类型。叶子必须是被索引数据类型，而其他树节点可以是任意 C 结构体（但这里仍必须遵守PostgreSQL 的数据类型规则，关于变长数据可参见 `varlena`）。如果树的内部数据类型在 SQL 层存在，可以使用`CREATE OPERATOR CLASS` 命令的 `STORAGE` 选项。可选的第八个方法是 `distance`，若操作符类希望支持有序扫描（最近邻搜索），则需要它。可选的第九个方法`fetch` 在操作符类希望支持仅索引扫描时需要，除非省略了`compress` 方法。

`consistent`

给定一个索引项 `p` 和一个查询值 `q`，该函数判断该索引项是否与该查询 “一致” ；也就是说，该索引项所代表的某一行是否可能使谓词 “*`indexed_column indexable_operator` q*” 为真。对于叶子索引项，这等同于测试该可索引条件；而对于内部树节点，这决定是否有必要扫描该树节点所表示的索引子树。当结果为`true` 时，还必须返回一个`recheck` 标志。它表示该谓词是确定为真，还是仅可能为真。如果`recheck` = `false`，则该索引已经精确测试了谓词条件；如果`recheck` = `true`，则该行只是候选匹配。在这种情况下，系统会自动针对实际行值计算*`indexable_operator`*，以判断它是否真的匹配。这种约定使GiST 能够同时支持无损和有损的索引结构。

该函数的 SQL 声明必须如下所示：

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type, smallint, oid, internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

而 C 模块中的对应代码则可以遵循如下框架：

```
PG_FUNCTION_INFO_V1(my_consistent);

Datum
my_consistent(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool       *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type  *key = DatumGetDataType(entry->key);
    bool        retval;

    /*
     * 根据 strategy、key 和 query 确定返回值。
     *
     * 使用 GIST_LEAF(entry) 判断当前调用位于索引树的哪个位置。
     * 例如，支持 = 操作符时这很有用（可以在非叶节点检查
     * union() 是否非空，在叶节点检查是否相等）。
     */

    *recheck = true;         /* 如果检查是精确的，则为 false */

    PG_RETURN_BOOL(retval);
}
```

这里，key 是索引中的一个元素，而query 是在该索引中查找的值。StrategyNumber 参数指示应用的是操作符类中的哪个操作符，它对应于CREATE OPERATOR CLASS 命令中的某个操作符编号。

取决于你在该类中包含了哪些操作符，query 的数据类型可能会随操作符而变化，因为它将是操作符右侧的类型，而这可能不同于左侧出现的被索引数据类型。（上面的代码框架假定只可能有一种类型；如果不是这样，获取query 参数值的方式就必须依赖于具体的操作符。）建议在 consistent 函数的 SQL 声明中，对query 参数使用该操作符类的被索引数据类型，即使实际类型可能因为操作符不同而有所不同。

union

该方法用于汇总树中的信息。给定一组项，该函数生成一个新的索引项，用来表示所有给定项。

该函数的 SQL 声明必须如下所示：

```
CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS storage_type
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

而 C 模块中的对应代码则可以遵循如下框架：

```
PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GISTENTRY  *ent = entryvec->vector;
    data_type  *out,
               *tmp,
               *old;
    int         numranges,
                i = 0;

    numranges = entryvec->n;
    tmp = DatumGetDataType(ent[0].key);
    out = tmp;

    if (numranges == 1)
    {
        out = data_type_deep_copy(tmp);

        PG_RETURN_DATA_TYPE_P(out);
    }

    for (i = 1; i < numranges; i++)
    {
        old = out;
        tmp = DatumGetDataType(ent[i].key);
        out = my_union_implementation(out, tmp);
    }

    PG_RETURN_DATA_TYPE_P(out);
}
```

如你所见，在这个框架里，我们处理的是一种满足`union(X, Y, Z) = union(union(X, Y), Z)`的数据类型。对于不满足这一性质的数据类型，只需在这个GiST 支持方法中实现正确的 union 算法即可。

`union` 函数的结果必须是索引存储类型的值，不管该类型是什么（它可能与被索引列的类型相同，也可能不同）。`union` 函数应返回一个指向新近通过 `palloc()` 分配的内存的指针。即使没有类型变化，也不能原样返回输入值。

如上所示，`union` 函数的第一个 `internal` 参数实际上是一个 `GistEntryVector` 指针。第二个参数是一个指向整数变量的指针，可以忽略。（过去要求`union` 函数把结果值的大小存入该变量，但现在已经不再需要。）

compress

将一个数据项转换成适合在索引页中物理存储的格式。如果省略 compress 方法，数据项将不经修改地存储在索引中。

该函数的 SQL 声明必须如下所示：

```
CREATE OR REPLACE FUNCTION my_compress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

而 C 模块中的对应代码则可以遵循如下框架：

```
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *retval;

    if (entry->leafkey)
    {
        /* 将 entry->key 替换为压缩后的形式 */
        compressed_data_type *compressed_data = palloc(sizeof(compressed_data_type));

        /* 根据 entry->key 填充 *compressed_data ... */

        retval = palloc(sizeof(GISTENTRY));
        gistentryinit(*retval, PointerGetDatum(compressed_data),
                      entry->rel, entry->page, entry->offset, FALSE);
    }
    else
    {
        /* 通常无需对非叶项做任何处理 */
        retval = entry;
    }

    PG_RETURN_POINTER(retval);
}
```

当然，为了压缩叶子节点，你必须把*compressed_data_type* 改成要转换成的具体类型。

decompress

将数据项的存储表示转换成操作符类中其他 GiST 方法能够操作的格式。如果省略 decompress 方法，就假定其他 GiST 方法可以直接处理存储的数据格式。（decompress 不一定是 compress 方法的逆操作；特别是如果 compress 是有损的，那么decompress 就不可能精确重建原始数据。decompress 也不一定等同于 fetch，因为其他 GiST 方法未必需要把数据完全重建出来。）

该函数的 SQL 声明必须如下所示：

```
CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

而 C 模块中的对应代码则可以遵循如下框架：

```
PG_FUNCTION_INFO_V1(my_decompress);

Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}
```

上述框架适用于不需要解压的情况。（当然，在这种情况下，直接完全省略该方法会更简单，而且也推荐这么做。）

`penalty`

返回一个值，指示把新项插入树中特定分支的“代价”。项会沿着树中 `penalty` 最小的路径插入。`penalty` 返回的值应为非负；如果返回负值，它将被按零处理。

该函数的 SQL 声明必须如下所示：

```
CREATE OR REPLACE FUNCTION my_penalty(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;  -- 某些情况下 penalty 函数不必是严格函数
```

而 C 模块中的对应代码则可以遵循如下框架：

```
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY  *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float      *penalty = (float *) PG_GETARG_POINTER(2);
    data_type  *orig = DatumGetDataType(origentry->key);
    data_type  *new = DatumGetDataType(newentry->key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

出于历史原因，`penalty` 函数并不是直接返回一个 `float` 结果；相反，它必须把该值存储到第三个参数指示的位置。返回值本身会被忽略，不过通常会返回该参数所指向的地址。

`penalty` 函数对于索引的良好性能至关重要。它会在插入时用于决定在树中应沿着哪个分支向下，以便选择把新项加到哪里。在查询时，索引越平衡，查找就越快。

`picksplit`

当索引页必须分裂时，该函数决定页面上的哪些项留在旧页中，哪些移到新页中。

该函数的 SQL 声明必须如下所示：

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

而 C 模块中的对应代码则可以遵循如下框架：

```
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    OffsetNumber maxoff = entryvec->n - 1;
    GISTENTRY  *ent = entryvec->vector;
    int         i,
                nbytes;
    OffsetNumber *left,
               *right;
    data_type  *tmp_union;
    data_type  *unionL;
    data_type  *unionR;
    GISTENTRY **raw_entryvec;

    maxoff = entryvec->n - 1;
    nbytes = (maxoff + 1) * sizeof(OffsetNumber);

    v->spl_left = (OffsetNumber *) palloc(nbytes);
    left = v->spl_left;
    v->spl_nleft = 0;

    v->spl_right = (OffsetNumber *) palloc(nbytes);
    right = v->spl_right;
    v->spl_nright = 0;

    unionL = NULL;
    unionR = NULL;

    /* 初始化原始项向量。 */
    raw_entryvec = (GISTENTRY **) malloc(entryvec->n * sizeof(void *));
    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
        raw_entryvec[i] = &(entryvec->vector[i]);

    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
    {
        int         real_index = raw_entryvec[i] - entryvec->vector;

        tmp_union = DatumGetDataType(entryvec->vector[real_index].key);
```

```
            Assert(tmp_union != NULL);

            /*
             * 选择索引项的存放位置，并相应更新 unionL 和 unionR。
             * 将项追加到 v->spl_left 或 v->spl_right，
             * 同时更新计数器。
             */

            if (my_choice_is_left(unionL, curl, unionR, curr))
            {
                if (unionL == NULL)
                    unionL = tmp_union;
                else
                    unionL = my_union_implementation(unionL, tmp_union);

                *left = real_index;
                ++left;
                ++(v->spl_nleft);
            }
            else
            {
                /*
                 * 对右侧执行相同操作
                 */
            }
        }

        v->spl_ldatum = DataTypeGetDatum(unionL);
        v->spl_rdatum = DataTypeGetDatum(unionR);
        PG_RETURN_POINTER(v);
    }
```

注意，picksplit 函数的结果是通过修改传入的 v 结构体来传递的。返回值本身会被忽略，不过通常会返回v 的地址。

和 penalty 一样，picksplit 函数对于索引的良好性能至关重要。设计合适的 penalty 和 picksplit 实现，正是实现高性能GiST 索引的难点所在。

same

如果两个索引项相同则返回真，否则返回假。（“索引项”是索引存储类型的值，不一定是原始被索引列的类型。）

该函数的 SQL 声明必须如下所示：

```
CREATE OR REPLACE FUNCTION my_same(storage_type, storage_type, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

而 C 模块中的对应代码则可以遵循如下框架：

```
PG_FUNCTION_INFO_V1(my_same);

Datum
my_same(PG_FUNCTION_ARGS)
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool       *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}
```

出于历史原因，`same` 函数并不是直接返回一个布尔结果；相反，它必须把该标志存储到第三个参数指示的位置。返回值本身会被忽略，不过通常会返回该参数所指向的地址。

`distance`

给定一个索引项 `p` 和一个查询值 `q`，该函数确定索引项与查询值之间的“距离”。如果操作符类包含任何排序操作符，就必须提供此函数。使用排序操作符的查询会优先返回“距离”值最小的索引项，因此结果必须与该操作符的语义一致。对于叶子索引项，结果仅表示到该索引项的距离；对于内部树节点，结果必须是其任意子项可能具有的最小距离。

该函数的 SQL 声明必须如下所示：

```
CREATE OR REPLACE FUNCTION my_distance(internal, data_type, smallint, oid, internal)
RETURNS float8
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

而 C 模块中的对应代码则可以遵循如下框架：

```
PG_FUNCTION_INFO_V1(my_distance);

Datum
my_distance(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    /* bool *recheck = (bool *) PG_GETARG_POINTER(4); */
    data_type  *key = DatumGetDataType(entry->key);
    double      retval;

    /*
     * 根据 strategy、key 和 query 确定返回值。
     */

    PG_RETURN_FLOAT8(retval);
}
```

distance 函数的参数与 consistent 函数的参数完全相同。

在确定距离时允许有一定近似，只要结果永不大于该项的实际距离即可。因此，例如在几何应用中，到包围盒的距离通常就足够了。对于内部树节点，返回的距离不能大于其任一子节点的距离。如果返回的距离不精确，函数必须将*recheck 设为 true。（对内部树节点则不必这样做；对它们总是假定计算结果不精确。）在这种情况下，执行器会在从堆中取出元组后计算准确距离，并在必要时重新排序这些元组。

如果距离函数对任意一个叶节点返回*recheck = true，原始排序操作符的返回类型必须是float8 或 float4，且距离函数的结果值必须能与原始排序操作符的结果进行比较，因为执行器会同时使用距离函数结果和重新计算得到的排序操作符结果进行排序。否则，距离函数的结果值可以是任意有限的float8 值，只要这些结果值的相对顺序与排序操作符返回的顺序一致即可。（无穷大和负无穷在内部用于处理空值等情况，因此不建议distance 函数返回这些值。）

fetch

为了支持仅索引扫描，将数据项的压缩索引表示转换为原始数据类型。返回的数据必须是最初被索引值的精确、无损副本。

该函数的 SQL 声明必须如下所示：

```
CREATE OR REPLACE FUNCTION my_fetch(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

参数是一个指向 GISTENTRY 结构体的指针。进入该函数时，它的key 字段包含一个压缩形式的非 NULL 叶子 datum。返回值是另一个 GISTENTRY 结构体，其key 字段以原始、未压缩形式包含同一个 datum。如果该操作符类的 compress 函数对叶子项不做任何处理，fetch 方法可以原样返回该参数。或者，如果该操作符类没有 compress 函数，那么fetch 方法也可以省略，因为它必然是空操作。

而 C 模块中的对应代码则可以遵循如下框架：

```
PG_FUNCTION_INFO_V1(my_fetch);

Datum
my_fetch(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    input_data_type *in = DatumGetPointer(entry->key);
    fetched_data_type *fetched_data;
    GISTENTRY  *retval;

    retval = palloc(sizeof(GISTENTRY));
    fetched_data = palloc(sizeof(fetched_data_type));

    /*
     * 将 'fetched_data' 转换为原始数据类型的 Datum。
     */

    /* 根据 fetched_data 填充 *retval。 */
    gistentryinit(*retval, PointerGetDatum(converted_datum),
                  entry->rel, entry->page, entry->offset, FALSE);

    PG_RETURN_POINTER(retval);
}
```

如果 compress 方法对叶子项是有损的，该操作符类就不能支持仅索引扫描，并且不得定义 fetch 函数。

所有 GiST 支持方法通常都在短生命周期的内存上下文中被调用；也就是说，每处理完一个元组，CurrentMemoryContext 都会被重置。因此通常无需过分担心释放所有通过 palloc 分配的内容。不过，在某些情况下，让支持方法在重复调用之间缓存数据是有用的。要做到这一点，可将寿命更长的数据分配在fcinfo->flinfo->fn_mcxt 中，并在fcinfo->flinfo->fn_extra 中保存指向它的指针。这类数据会在一次索引操作期间存活（例如一次 GiST 索引扫描、索引构建或索引元组插入）。在替换fn_extra 值时要注意 pfree 旧值，否则泄漏会在整个操作期间不断累积。

64.4. 实现

64.4.1. GiST 缓冲构建

仅靠把所有元组逐个插入来构建大型 GiST 索引往往很慢，因为如果索引元组分散在整个索引中，而索引又大到无法放入缓存，插入时就需要大量随机 I/O。从版本 9.2 开始，PostgreSQL 支持一种更高效的、基于缓冲的 GiST 索引构建方法，对于非有序数据集，它可以显著减少所需的随机 I/O 次数。对于顺序良好的数据集，收益较小或者根本没有，因为一次只有少量页面会接收新元组，而这些页面即使整个索引放不进缓存，也能够放进缓存。

不过，缓冲索引构建需要更频繁地调用`penalty`函数，这会消耗一些额外的 CPU 资源。此外，缓冲区需要临时磁盘空间，最多可达最终索引的大小。缓冲还可能正面或负面地影响最终索引的质量。这种影响取决于多种因素，例如输入数据的分布以及操作符类的实现。

默认情况下，当索引大小达到effective_cache_size 时，GiST 索引构建会切换到缓冲方法。也可以通过 CREATE INDEX 命令的`buffering`参数手工强制启用或禁止缓冲。默认行为在大多数情况下都不错，但如果输入数据是有序的，关闭缓冲模式可能会略微加快构建速度。

64.5. 示例

PostgreSQL 源代码发行包包含了若干使用 GiST 实现的索引方法示例。核心系统目前提供了文本检索支持（为 `tsvector` 和 `tsquery` 建立索引），并为某些内置几何数据类型提供了与 R 树等价的功能（见`src/backend/access/gist/gistproc.c`）。下列 `contrib` 模块中也包含 GiST 操作符类：

`btree_gist`

为多种数据类型提供与 B-树等价的功能

`cube`

多维立方体的索引

`hstore`

用于存储（键，值）对的模块

`intarray`

一维 int4 值数组的 RD 树

`ltree`

树状结构的索引

`pg_trgm`

基于三字符组匹配的文本相似度

`seg`

“float 范围” 的索引

第 65 章 SP-GiST 索引

65.1. 简介

SP-GiST 是空间分区 GiST 的缩写。SP-GiST 支持分区搜索树，这使得开发多种不同的非平衡数据结构成为可能，例如四叉树、k-d 树以及基数树（trie）。这些结构的共同特征是，它们会反复将搜索空间划分为不必等大的分区。与这种划分规则良好匹配的搜索可以非常快。

这些常见数据结构最初是为内存中使用而开发的。在主存中，它们通常被设计成一组由指针链接的动态分配结点。由于这些指针链可能相当长，直接存储到磁盘上并不合适，因为那会需要过多的磁盘访问。相比之下，基于磁盘的数据结构应当具有较高的扇出，以尽量减少 I/O。SP-GiST 要解决的难题是，如何以这样的方式将搜索树结点映射到磁盘页：即使搜索遍历了许多结点，也只需访问少数几个磁盘页。

像 GiST 一样，SP-GiST 的目标是让数据类型领域专家而非数据库专家，能够针对自定义数据类型开发合适的访问方法。

这里的一些信息来自普渡大学的 SP-GiST 索引项目网站[1]。SP-GiST 在 PostgreSQL 中的实现主要由 Teodor Sigaev 和 Oleg Bartunov 维护，他们的网站[2]上还有更多信息。

65.2. 内置操作符类

PostgreSQL 核心发行版包含了 SP-GiST 操作符类，如表 65.1 所示。

表 65.1. 内置 SP-GiST 操作符类

名称	被索引数据类型	可索引操作符
kd_point_ops	point	<< <@ <^ >> >^ ~=
quad_point_ops	point	<< <@ <^ >> >^ ~=
range_ops	任意范围类型	&& &< &> -|- << <@ = >> @>
box_ops	box	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
poly_ops	polygon	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
text_ops	text	< <= = > >= ~<=~ ~<~ ~>=~ ~>~ ^@
inet_ops	inet, cidr	&& >> >>= > >= <> << <<= < <= =

对于类型 point 的两个操作符类，quad_point_ops 是默认选项。kd_point_ops 支持相同的操作符，但使用不同的索引数据结构，在某些应用中可能提供更好的性能。

65.3. 可扩展性

SP-GiST 提供了一个高度抽象的接口，访问方法开发者只需实现特定数据类型所需的方法。SP-GiST 核心负责将树结构高效映射到磁盘并执行搜索，同时也处理并发与日志记录方面的问题。

[1] https://www.cs.purdue.edu/spgist/
[2] http://www.sai.msu.su/~megera/wiki/spgist_dev

SP-GiST 树的叶子元组包含与被索引列相同数据类型的值。位于根层的叶子元组始终包含原始的被索引数据值，而位于较低层的叶子元组可能只包含压缩表示，例如一个后缀。在这种情况下，操作符类支持函数必须能够利用为到达叶子层而经过的内部元组中累积的信息，重建出原始值。

内部元组更为复杂，因为它们是搜索树中的分支点。每个内部元组都包含一个或多个结点，代表相似叶子值的分组。一个结点包含一个向下链接，它要么指向更低层级的另一个内部元组，要么指向一小组位于同一索引页上的叶子元组。每个结点通常都有一个描述它的标签；例如在基数树中，结点标签可以是字符串值的下一个字符。（或者，如果某个操作符类对所有内部元组都使用固定的一组结点，也可以省略结点标签；参见第 65.4.2 节。）内部元组还可以选择带有一个描述其所有成员的前缀值。在基数树中，这可以是所表示字符串的公共前缀。前缀值不一定真的是前缀，它也可以是操作符类需要的任何数据；例如在四叉树中，它可以存储划分四个象限所依据的中心点。这样，四叉树的内部元组还会包含四个结点，对应于该中心点周围的四个象限。

某些树算法需要知道当前元组所在的层级（或深度），因此 SP-GiST 核心允许操作符类在沿树向下遍历时管理层级计数。它还支持在需要时增量重建所表示的值，并支持在树下降过程中向下传递额外数据（称为遍历值）。

> **注意**
>
> SP-GiST 核心代码负责处理值为 null 的索引项。虽然 SP-GiST 索引会为被索引列中的 null 值存储索引项，但索引操作符类代码看不到这些项：值为 null 的索引项或搜索条件绝不会传给操作符类方法。（这里假定 SP-GiST 操作符是严格的，因此对 null 值不可能返回真。）所以这里不再讨论 null 值。

SP-GiST 的索引操作符类必须提供五个用户定义方法，另外还有一个可选方法。五个必需方法都遵循这样的约定：接受两个 `internal` 参数，第一个参数是指向某个 C 结构体的指针，其中包含该支持方法的输入值；第二个参数也是指向某个 C 结构体的指针，方法必须将输出值写入其中。四个必需方法只返回 `void`，因为它们的全部结果都体现在输出结构体中；但 `leaf_consistent` 还返回一个 `boolean` 结果。这些方法不得修改其输入结构体中的任何字段。在所有情况下，调用用户定义方法之前，输出结构体都会先被清零。可选的第六个方法 `compress` 只接受一个待索引的 datum 参数，并返回适合在叶子元组中物理存储的值。

五个必需的用户定义方法是：

`config`

返回索引实现的静态信息，包括前缀和结点标签数据类型的 OID。

该函数的 SQL 声明必须如下所示：

```
CREATE FUNCTION my_config(internal, internal) RETURNS void ...
```

第一个参数是一个指向 spgConfigIn C 结构体的指针，其中包含该函数的输入数据。第二个参数是一个指向 spgConfigOut C 结构体的指针，函数必须将结果数据填入其中。

```
typedef struct spgConfigIn
{
    Oid         attType;        /* 要被索引的数据类型 */
} spgConfigIn;

typedef struct spgConfigOut
{
    Oid         prefixType;     /* 内部元组前缀的数据类型 */
    Oid         labelType;      /* 内部元组结点标签的数据类型 */
    Oid         leafType;       /* 叶子元组值的数据类型 */
    bool        canReturnData;  /* 操作符类能重建原始数据 */
    bool        longValuesOK;   /* 操作符类能处理大小 > 1 页的值 */
} spgConfigOut;
```

传入 attType 是为了支持多态索引操作符类；对于普通的固定数据类型操作符类，它始终具有相同的值，因此可以忽略。

对于不使用前缀的操作符类，可以将 prefixType 设为 VOIDOID。同样，对于不使用结点标签的操作符类，可以将 labelType 设为 VOIDOID。如果操作符类能够重建最初提供的索引值，则应将 canReturnData 设为真。只有在 attType 是变长类型，并且该操作符类能够通过反复取后缀来切分长值时，才应将 longValuesOK 设为真（参见第 65.4.1 节）。

leafType 通常与 attType 相同。出于向后兼容的考虑，config 方法可以不初始化 leafType；这与将 leafType 设为等于 attType 具有相同效果。当 attType 与 leafType 不同时，必须提供可选的compress 方法。compress 方法负责把待索引的 datum 从 attType 转换为 leafType。注意：两个一致性检查函数都会收到未经改变的 scankeys，不会经过compress 的转换。

choose

为向内部元组插入新值选择一种方法。

该函数的 SQL 声明必须如下所示：

```
CREATE FUNCTION my_choose(internal, internal) RETURNS void ...
```

第一个参数是一个指向 spgChooseIn C 结构体的指针，其中包含该函数的输入数据。第二个参数是一个指向 spgChooseOut C 结构体的指针，函数必须将结果数据填入其中。

```
typedef struct spgChooseIn
{
    Datum       datum;          /* 要被索引的原始 datum */
    Datum       leafDatum;      /* 当前要存储在叶子中的 datum */
    int         level;          /* 当前层级（从零开始计） */
```

```
    /* 来自当前内部元组的数据 */
    bool        allTheSame;     /* 元组被标记为全部相同？ */
    bool        hasPrefix;      /* 元组有前缀？ */
    Datum       prefixDatum;    /* 如果有，前缀值 */
    int         nNodes;         /* 内部元组中的结点数 */
    Datum      *nodeLabels;     /* 结点标签值（如果没有则为 NULL） */
} spgChooseIn;

typedef enum spgChooseResultType
{
    spgMatchNode = 1,           /* 下降到现有结点 */
    spgAddNode,                 /* 向内部元组添加一个结点 */
    spgSplitTuple               /* 拆分内部元组（修改其前缀） */
} spgChooseResultType;

typedef struct spgChooseOut
{
    spgChooseResultType resultType;     /* 动作代码，见上文 */
    union
    {
        struct                  /* spgMatchNode 的结果 */
        {
            int         nodeN;      /* 下降到该结点（索引从 0 开始） */
            int         levelAdd;   /* 层级增加这么多 */
            Datum       restDatum;  /* 新的叶子 datum */
        }           matchNode;
        struct                  /* spgAddNode 的结果 */
        {
            Datum       nodeLabel;  /* 新结点的标签 */
            int         nodeN;      /* 在哪里插入它（索引从 0 开始） */
        }           addNode;
        struct                  /* spgSplitTuple 的结果 */
        {
            /* 构造只有一个子元组的新上层内部元组所需的信息 */
            bool        prefixHasPrefix;    /* 元组应有前缀？ */
            Datum       prefixPrefixDatum;  /* 如果有，前缀值 */
            int         prefixNNodes;       /* 结点数 */
            Datum      *prefixNodeLabels;   /* 它们的标签（或为 NULL 表示
                                             * 没有标签） */
            int         childNodeN;         /* 哪个结点接收子元组 */

            /* 构造包含所有旧结点的新下层内部元组所需的信息 */
            bool        postfixHasPrefix;   /* 元组应有前缀？ */
            Datum       postfixPrefixDatum; /* 如果有，前缀值 */
        }           splitTuple;
    }           result;
} spgChooseOut;
```

datum 是将要插入索引的、类型为 spgConfigIn.attType 的原始 datum。leafDatum 是一个类型为 spgConfigOut.leafType 的值；如果提供了 compress 方法，它最初是将 datum 交给 compress 的结果，否则与 datum 相同。如果 choose 或 picksplit 方法修改了它，那么在树的更低层级中 leafDatum 可能会发生变化。当插入搜索到达叶子页时，leafDatum 的当前值将存储在新创建的叶子元组中。level 是当前内部元组的层级，根层为零。若当前内部元组被标记为包含多个等价结点（参见第 65.4.3 节），则 allTheSame 为真。若当前内部元组包含前缀，则 hasPrefix 为真；若是如此，prefixDatum 就是该前缀值。nNodes 是内部元组中包含的子结点数量，nodeLabels 是它们的标签值数组；如果没有标签，则为 NULL。

choose 函数可以判定：新值要么匹配某个现有子结点，要么必须添加一个新子结点，要么与该元组的前缀不一致，因此必须拆分该内部元组以创建限制性更弱的前缀。

如果新值匹配某个现有子结点，则将 resultType 设为 spgMatchNode。将 nodeN 设为该结点在结点数组中的索引（从零开始）。将 levelAdd 设为通过该结点向下下降所导致的 level 增量；如果操作符类不使用层级，则保持为零。若操作符类不会在层级之间修改 datum，则将 restDatum 设为与 leafDatum 相等；否则，将它设为下一层要用作 leafDatum 的修改后值。

如果必须添加一个新子结点，则将 resultType 设为 spgAddNode。将 nodeLabel 设为新结点要使用的标签，并将 nodeN 设为该结点应插入到结点数组中的位置索引（从零开始）。添加该结点后，choose 函数会使用修改后的内部元组再次被调用；这次调用应返回 spgMatchNode。

如果新值与该元组的前缀不一致，则将 resultType 设为 spgSplitTuple。这个动作会把所有现有结点移动到一个新的较低层内部元组中，并用一个仅含单个向下链接、指向该新下层内部元组的元组替换现有内部元组。将 prefixHasPrefix 设为指示新的上层元组是否应有前缀；若应有，则将 prefixPrefixDatum 设为该前缀值。这个新前缀值必须比原来的限制性更弱，以便能够接受将要索引的新值。将 prefixNNodes 设为新元组所需的结点数，并将 prefixNodeLabels 设为一个通过 palloc 分配的数组，其中保存这些结点的标签；如果不需要结点标签，则设为 NULL。注意，新上层元组的总大小不得超过它所替换的元组的总大小；这限制了新前缀和新标签的长度。将 childNodeN 设为将向下链接到新的较低层内部元组的那个结点的索引（从零开始）。将 postfixHasPrefix 设为指示新的较低层内部元组是否应有前缀；若应有，则将 postfixPrefixDatum 设为该前缀值。这两个前缀与向下链接结点的标签（如果有）的组合，必须与原始前缀具有相同的含义，因为没有机会修改被移动到新下层元组中的结点标签，也不能更改任何子索引项。结点拆分完成后，choose 函数会用替换后的内部元组再次被调用。如果 spgSplitTuple 动作没有创建合适的结点，这次调用可以返回 spgAddNode。最终，choose 必须返回 spgMatchNode，以便插入继续下降到下一层。

picksplit

决定如何在一组叶子元组上创建一个新的内部元组。

该函数的 SQL 声明必须如下所示：

```
CREATE FUNCTION my_picksplit(internal, internal) RETURNS void ...
```

第一个参数是一个指向 spgPickSplitIn C 结构体的指针，其中包含该函数的输入数据。第二个参数是一个指向 spgPickSplitOut C 结构体的指针，函数必须将结果数据填入其中。

```
typedef struct spgPickSplitIn
{
    int          nTuples;        /* 叶子元组的数量 */
    Datum       *datums;         /* 它们的 datum（长度为 nTuples 的数组） */
    int          level;          /* 当前层级（从零开始计） */
} spgPickSplitIn;

typedef struct spgPickSplitOut
{
    bool         hasPrefix;      /* 新内部元组应有前缀？ */
    Datum        prefixDatum;    /* 如果有，前缀值 */

    int          nNodes;         /* 新内部元组的结点数 */
    Datum       *nodeLabels;     /* 它们的标签（或为 NULL 表示无标签） */

    int         *mapTuplesToNodes;   /* 每个叶子元组对应的结点索引 */
    Datum       *leafTupleDatums;    /* 每个新叶子元组中存储的 datum */
} spgPickSplitOut;
```

nTuples 是提供的叶子元组数量。datums 是由它们的 datum 值组成的数组，类型为 spgConfigOut.leafType。level 是这些叶子元组共享的当前层级，它将成为新内部元组的层级。

将 hasPrefix 设为指示新的内部元组是否应有前缀；若应有，则将 prefixDatum 设为该前缀值。将 nNodes 设为新内部元组将包含的结点数，并将 nodeLabels 设为这些结点的标签值数组；如果不需要结点标签，则设为 NULL。将 mapTuplesToNodes 设为一个数组，其中给出每个叶子元组应分配到的结点索引（从零开始）。将 leafTupleDatums 设为要存储在新叶子元组中的值数组（如果操作符类不会在层级之间修改 datum，这些值就与输入的 datums 相同）。注意，picksplit 函数负责为 nodeLabels、mapTuplesToNodes 和 leafTupleDatums 数组执行 palloc。

如果提供了多于一个叶子元组，则期望 picksplit 函数把它们划分到多于一个结点中；否则就无法把叶子元组拆分到多个页上，而这正是此操作的最终目的。因此，如果 picksplit 最终把所有叶子元组都放进同一个结点，SP-GiST 核心代码会覆盖这一决定，生成一个内部元组，并将叶子元组随机分配到多个标签相同的结点上。这样的元组会被标记为 allTheSame，以表明发生了这种情况。choose 和 inner_consistent 函数必须对这种内部元组做出恰当处理。更多信息见第 65.4.3 节。

只有在 config 函数将 longValuesOK 设为真，并且提供了一个大于一页的输入值时，picksplit 才会应用到单个叶子元组。在这种情况下，这个操作的目的是剥离一个前缀，并产生一个新的、更短的叶子 datum 值。该调用会重复进行，直到生成足够短、能够放入一页的叶子 datum。更多信息见第 65.4.1 节。

inner_consistent

在树搜索期间返回需要继续跟随的一组结点（分支）。

该 SQL 声明必须如下所示：

```
CREATE FUNCTION my_inner_consistent(internal, internal) RETURNS void ...
```

第一个参数是一个指向 spgInnerConsistentInC 结构体的指针，其中包含该函数的输入数据。第二个参数是一个指向spgInnerConsistentOutC 结构体的指针，函数必须将结果数据填入其中。

```
typedef struct spgInnerConsistentIn
{
    ScanKey     scankeys;       /* 操作符和比较值的数组 */
    int         nkeys;          /* 数组长度 */

    Datum       reconstructedValue;     /* 在父元组处重建的值 */
    void       *traversalValue; /* 操作符类特定的遍历值 */
    MemoryContext traversalMemoryContext;   /* 将新的遍历值放在这里 */
    int         level;          /* 当前层级（从零开始计） */
    bool        returnData;     /* 必须返回原始数据? */

    /* 来自当前内部元组的数据 */
    bool        allTheSame;     /* 元组被标记为全部相同？ */
    bool        hasPrefix;      /* 元组有前缀? */
    Datum       prefixDatum;    /* 如果有，前缀值 */
    int         nNodes;         /* 内部元组中的结点数 */
    Datum      *nodeLabels;     /* 结点标签值（如果没有则为 NULL） */
} spgInnerConsistentIn;

typedef struct spgInnerConsistentOut
{
    int         nNodes;         /* 需要访问的子结点数 */
    int        *nodeNumbers;    /* 它们在结点数组中的索引 */
    int        *levelAdds;      /* 对每个结点层级增加这么多 */
    Datum      *reconstructedValues;    /* 关联的重建值 */
    void      **traversalValues;        /* 操作符类特定的遍历值 */
} spgInnerConsistentOut;
```

数组 scankeys 的长度为 nkeys，它描述索引搜索条件。这些条件用 AND 组合 — 只有满足全部条件的索引项才是我们关心的。（注意，nkeys= 0 表示所有索引项都满足该查询。）通常一致性检查函数只关心每个数组元素的 sk_strategy 和 sk_argument 字段，它们分别给出可索引操作符和比较值。特别地，无需检查sk_flags 以判断比较值是否为 NULL，因为 SP-GiST 核心代码会过滤掉此类条件。reconstructedValue 是为父元组重建的值；以下情况下它为(Datum) 0：位于根层，或者inner_consistent 函数没有在父层提供该值。reconstructedValue 始终属于 spgConfigOut .leafType 类型。traversalValue 是指向任意遍历数据的指针，这些数据由上一次调用inner_consistent 处理父索引元组时向下传递；在根层时则为 NULL。traversalMemoryContext 是存放输出遍历值（见下文）的内存上下文。level 是当前内部元组的层级，根层为零。returnData 为 true 表示本查询需要重建数据；这要求 config 函数将 canReturnData 设为真。allTheSame 为真，表示当前内部元组被标记为“全部相同”；在这种情况下，所有结点都具有相同的标签（如果有），因此要么全部匹配该查询，要么全部不匹配（参见第 65.4.3 节）。hasPrefix 为真时，表示当前内部元组包含前缀；若是如此，prefixDatum 就是该前缀值。nNodes 是内部元组中包含的子结点数量，而nodeLabels 是它们的标签值数组；如果结点没有标签，则为 NULL。

nNodes 必须设为搜索需要访问的子结点数量，并且 nodeNumbers 必须设为这些结点索引的数组。如果操作符类跟踪层级，则将 levelAdds 设为一个数组，其中给出下降到每个待访问结点时所需增加的层数。（这些增量常常对所有结点都相同，但并非必然如此，所以这里使用数组。）如果需要值重建，则将 reconstructedValues 设为一个数组，其中包含为每个待访问子结点重建的 spgConfigOut.leafType 类型的值；否则，将 reconstructedValues 保持为 NULL。如果希望将额外的带外信息（“遍历值”）向下传递到树搜索的更低层，则将 traversalValues 设为适当遍历值的数组，每个待访问子结点对应一个；否则，将 traversalValues 保持为 NULL。注意，inner_consistent 函数负责在当前内存上下文中为 nodeNumbers、levelAdds、reconstructedValues 和 traversalValues 数组执行 palloc。不过，traversalValues 数组所指向的任何输出遍历值都应在 traversalMemoryContext 中分配。每个遍历值都必须是单独 palloc 的一个块。

leaf_consistent

如果叶子元组满足查询，则返回 true。

该 SQL 声明必须如下所示：

```
CREATE FUNCTION my_leaf_consistent(internal, internal) RETURNS bool ...
```

第一个参数是一个指向 spgLeafConsistentInC 结构体的指针，其中包含该函数的输入数据。第二个参数是一个指向spgLeafConsistentOutC 结构体的指针，函数必须将结果数据填入其中。

```
typedef struct spgLeafConsistentIn
{
    ScanKey     scankeys;       /* 操作符和比较值的数组 */
    int         nkeys;          /* 数组长度 */

    Datum       reconstructedValue;     /* 在父元组处重建的值 */
    void       *traversalValue; /* 操作符类特定的遍历值 */
    int         level;          /* 当前层级（从零开始计） */
    bool        returnData;     /* 必须返回原始数据? */

    Datum       leafDatum;      /* 叶子元组中的 datum */
} spgLeafConsistentIn;

typedef struct spgLeafConsistentOut
{
    Datum       leafValue;      /* 重建出的原始数据（如果有） */
    bool        recheck;        /* 如果必须重新检查操作符则设为真 */
} spgLeafConsistentOut;
```

数组 scankeys 的长度为 nkeys，它描述索引搜索条件。这些条件用 AND 组合 – 只有满足全部条件的索引项才满足该查询。（注意，nkeys= 0 表示所有索引项都满足该查询。）通常一致性检查函数只关心每个数组元素的 sk_strategy 和 sk_argument 字段，它们分别给出可索引操作符和比较值。特别地，无需检查sk_flags 以判断比较值是否为 NULL，因为 SP-GiST 核心代码会过滤掉此类条件。reconstructedValue 是为父元组重建的值；以下情况下它为 (Datum) 0：位于根层，或者inner_consistent 函数没有在父层提供该值。reconstructedValue 始终属于 spgConfigOut.leafType 类型。traversalValue 是指向任意遍历数据的指针，这些数据由上一次调用inner_consistent 处理父索引元组时向下传递；在根层时则为 NULL。level 是当前叶子元组的层级，根层为零。returnData 为 true 表示本查询需要重建数据；这要求 config 函数将 canReturnData 设为真。leafDatum 是当前叶元组中存储的键值，其类型为spgConfigOut.leafType。

如果叶子元组匹配查询，则该函数必须返回 true，否则返回 false。在返回 true 的情况下，如果 returnData 为 true，则必须将 leafValue 设为最初为该叶子元组提供并建立索引的值（类型为 spgConfigIn.attType）。此外，如果匹配结果不确定，因而必须将操作符重新应用到实际的堆元组上以验证匹配，则可以将 recheck 设为 true。

可选的用户定义方法是：

```
Datum compress(Datum in)
```

将数据项转换为一种适合在索引叶子元组中物理存储的格式。它接受一个类型为 spgConfigIn.attType 的值，并返回一个类型为 spgConfigOut.leafType 的值。输出值不应经过 TOAST 处理。

所有 SP-GiST 支持方法通常都在一个短生命周期的内存上下文中调用；也就是说，处理完每个元组后，`CurrentMemoryContext` 都会被重置。因此，通常不必太担心是否 pfree 了你用 palloc 分配的所有内容。（`config` 方法是个例外：它应尽量避免内存泄漏。不过通常 `config` 方法只需把常量赋入传入的参数结构体即可。）

如果被索引列属于支持排序规则的数据类型，则索引排序规则会通过标准的 `PG_GET_COLLATION()` 机制传递给所有支持方法。

65.4. 实现

本节介绍实现细节以及其他一些对 SP-GiST 操作符类实现者有用的技巧。

65.4.1. SP-GiST 限制

单个叶子元组和内部元组都必须能放入单个索引页中（默认 8kB）。因此，在对变长数据类型的值建立索引时，只有像基数树这类方法才能支持长值：树的每一层都包含足够短、能放进一页的前缀，而最终叶子层也包含足够短、能放进一页的后缀。只有当操作符类准备好保证这一点时，才应将 `longValuesOK` 设为 true。否则，SP-GiST 核心会拒绝为过大、无法放入索引页的值建立索引的请求。

同样，确保内部元组不会增长到大得无法放入索引页，也是操作符类的责任；这限制了单个内部元组中可使用的子结点数量，以及前缀值的最大大小。

另一个限制是，当内部元组的某个结点指向一组叶子元组时，这些元组必须全部位于同一个索引页上。（这是一个设计决策，目的是减少寻道，并节省把这类元组链接成链时所需链接占用的空间。）如果一组叶子元组增长到单页无法容纳，就会执行拆分并插入一个中间内部元组。要解决这个问题，新的内部元组必须把叶子值集合划分成多个结点组。如果操作符类的 `picksplit` 函数做不到这一点，SP-GiST 核心就会诉诸第 65.4.3 节中描述的非常规措施。

当 `longValuesOK` 为真时，预期 SP-GiST 树的连续各层会把越来越多的信息吸收到内部元组的前缀和结点标签中，从而使所需的叶子 datum 越来越小，最终能够放入一页。为了防止操作符类中的缺陷导致插入陷入无限循环，如果在连续十次调用 `choose` 方法之后叶子 datum 仍没有变小，SP-GiST 核心就会报错。

65.4.2. 无结点标签的 SP-GiST

某些树算法对每个内部元组都使用固定的一组结点；例如在四叉树中，总是恰好有四个结点，对应于内部元组中心点周围的四个象限。在这种情况下，代码通常按编号处理结点，因此不需要显式的结点标签。为了省略结点标签（从而节省一些空间），`picksplit` 函数可以为 `nodeLabels` 数组返回 NULL，同样在执行 `spgSplitTuple` 动作期间，`choose` 函数也可以为 `prefixNodeLabels` 数组返回 NULL。随后对 `choose` 和 `inner_consistent` 的调用中，`nodeLabels` 也将为 NULL。原则上，同一个索引中可以对某些内部元组使用结点标签，而对另一些省略。

当处理具有无标签结点的内部元组时，`choose` 返回 `spgAddNode` 是错误的，因为在这种情况下结点集合应被视为固定不变。

65.4.3. “全部相同”的内部元组

当 `picksplit` 无法把提供的叶子值划分为至少两个结点类别时，SP-GiST 核心可以覆盖操作符类 `picksplit` 函数的结果。在这种情况下，会创建一个新的内部元组，其中有多个结点，而每个结点都具有相同的标签（如果有）；这个标签就是 `picksplit` 给它唯一使用的那个结点分配的标签。叶子值会被随机分配到这些等价结点中。该内部元组会设置 `allTheSame` 标志，用来提醒 `choose` 和 `inner_consistent` 函数：这个元组的结点集合并不是它们通常会预期的那种。

在处理 allTheSame 元组时，choose 返回 spgMatchNode 表示新值可以分配给任意一个等价结点；核心代码会忽略给出的 nodeN 值，并随机下降到其中一个结点（以保持树的平衡）。choose 返回 spgAddNode 则属于错误，因为那会使结点不再全部等价；如果待插入的值与现有结点不匹配，就必须使用 spgSplitTuple 动作。

在处理 allTheSame 元组时，inner_consistent 函数应当要么把全部结点返回为继续索引搜索的目标，要么一个也不返回，因为它们都是等价的。这是否需要编写特殊情况代码，取决于 inner_consistent 函数平常对这些结点含义做了多大程度的假定。

65.5. 示例

PostgreSQL 源代码发行版中包含了若干 SP-GiST 索引操作符类示例，如表 65.1 所述。要查看代码，请参阅 src/backend/access/spgist/ 和 src/backend/utils/adt/。

第 66 章 GIN 索引

66.1. 简介

GIN 是 Generalized Inverted Index（通用倒排索引）的缩写。GIN 被设计用于处理这样一类情况：要建立索引的项是组合值，而索引需要处理的查询则要搜索出现在这些组合项内部的元素值。例如，这些项可以是文档，而查询可以是搜索包含特定词语的文档。

我们用项来指代要被索引的组合值，用键来指代其中的元素值。GIN 总是存储并搜索键，而不是项值本身。

GIN 索引存储一组（键，倒排列表）对，其中一个倒排列表（posting list）是一个包含出现该键的行 ID 的集合。由于一个项可以包含多个键，同一个行 ID 可以出现在多个倒排列表中。每个键值只会被存储一次，因此在同一个键大量重复出现的场景下，GIN 索引会非常紧凑。

GIN 之所以是通用的，是因为 GIN 访问方法的代码不需要知道它所加速的具体操作。相反，它使用为特定数据类型定义的自定义策略。该策略定义了如何从被索引项和查询条件中提取键，以及如何判断一个包含查询中某些键值的行是否真正满足该查询。

GIN 的一个优点在于，它允许由数据类型所属领域的专家来开发带有合适访问方法的自定义数据类型，而不必依赖数据库专家。这一点与使用 GiST 的优势非常相似。

PostgreSQL 中的 GIN 实现主要由 Teodor Sigaev 和 Oleg Bartunov 维护。关于 GIN 的更多信息，可在他们的网站[1]上找到。

66.2. 内置操作符类

PostgreSQL 核心发行版包含表 66.1 所示的 GIN 操作符类。（在附录 F 中描述的一些可选模块还提供额外的 GIN 操作符类。）

表 66.1. 内置 GIN 操作符类

名称	被索引数据类型	可索引操作符
`array_ops`	`anyarray`	`&& <@ = @>`
`jsonb_ops`	`jsonb`	`? ?& ?| @>`
`jsonb_path_ops`	`jsonb`	`@>`
`tsvector_ops`	`tsvector`	`@@ @@@`

在用于类型 `jsonb` 的两个操作符类中，`jsonb_ops` 是默认项。`jsonb_path_ops` 支持的操作符较少，但对这些操作符提供更好的性能。详见第 8.14.4 节。

66.3. 可扩展性

GIN 接口具有很高的抽象层次，访问方法实现者只需实现所访问数据类型的语义。GIN 层本身会处理并发、日志记录以及树结构的搜索。

要让一个 GIN 访问方法工作起来，只需实现少数几个用户定义的方法，它们定义了树中键的行为，以及键、被索引项和可索引查询之间的关系。简言之，GIN 将可扩展性与通用性、代码重用以及清晰的接口结合在一起。

[1] http://www.sai.msu.su/~megera/wiki/Gin

GIN 操作符类必须提供两个方法：

```
Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)
```

给定一个要建立索引的项，返回一个用 palloc 分配的键数组。返回键的数量必须存入 *nkeys。如果任一键可以为 null，还要另外 palloc 一个包含 *nkeys 个 bool 字段的数组，将其地址存入 *nullFlags，并按需设置这些空值标志。如果所有键都不是 null，*nullFlags 可以保持为 NULL（其初始值）。如果该项不包含任何键，则返回值可以为 NULL。

```
Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool
**pmatch, Pointer **extra_data, bool **nullFlags, int32 *searchMode)
```

给定一个待查询的值，返回一个用 palloc 分配的键数组；也就是说，query 是一个可索引操作符右侧的值，而该操作符左侧是被索引列。n 是该操作符在操作符类中的策略号（见第 38.15.2 节）。通常，extractQuery 需要查看 n，以确定 query 的数据类型，以及应采用何种方法提取键值。返回键的数量必须存入 *nkeys。如果任一键可以为 null，还要另外 palloc 一个包含 *nkeys 个 bool 字段的数组，将其地址存入 *nullFlags，并按需设置这些空值标志。如果所有键都不是 null，*nullFlags 可以保持为 NULL（其初始值）。如果 query 不包含任何键，则返回值可以为 NULL。

searchMode 是一个输出参数，用于让 extractQuery 指定搜索如何执行。若 *searchMode 被设置为 GIN_SEARCH_MODE_DEFAULT（调用前它会被初始化为该值），则只有至少匹配一个返回键的项才会被视为候选匹配。若 *searchMode 被设置为 GIN_SEARCH_MODE_INCLUDE_EMPTY，则除至少包含一个匹配键的项之外，完全不含任何键的项也会被视为候选匹配。（例如，该模式对于实现是子集操作符很有用。）若 *searchMode 被设置为 GIN_SEARCH_MODE_ALL，则索引中所有非 null 项都会被视为候选匹配，无论它们是否匹配任一返回键。（该模式比前两种选择慢得多，因为它基本上需要扫描整个索引；但为了正确处理某些边界情况，可能有此必要。在大多数情况下都需要此模式的操作符，大概并不适合作为 GIN 操作符类的候选。）用于设置该模式的符号定义在 access/gin.h 中。

pmatch 是一个在支持部分匹配时使用的输出参数。要使用它，extractQuery 必须分配一个包含 *nkeys 个 bool 的数组，并将其地址存入 *pmatch。若相应键需要部分匹配，则数组对应元素应设置为 true，否则设置为 false。如果 *pmatch 被设置为 NULL，那么 GIN 认为不需要部分匹配。该变量在调用前会初始化为 NULL，因此不支持部分匹配的操作符类可以直接忽略此参数。

extra_data 是一个输出参数，用于让 extractQuery 向 consistent 和 comparePartial 方法传递额外数据。要使用它，extractQuery 必须分配一个包含 *nkeys 个指针的数组，并将其地址存入 *extra_data，然后把所需内容存入各个指针。该变量在调用前会初始化为 NULL，因此不需要额外数据的操作符类可以直接忽略此参数。如果设置了 *extra_data，则整个数组会传给 consistent 方法，而对应元素会传给 comparePartial 方法。

操作符类还必须提供一个函数，用于检查被索引项是否匹配查询。它有两种形式：布尔型 consistent 函数，以及三值型 triConsistent 函数。triConsistent 覆盖了两者的功能，因此仅提供 triConsistent 就已经足够。不过，如果布尔变体的计算代价明显更低，那么同时提供两者可能更有利。若只提供布尔变体，则一些依赖于在取回所有键之前先排除索引项的优化将被禁用。

```
bool consistent(bool check[], StrategyNumber n, Datum query, int32
nkeys, Pointer extra_data[], bool *recheck, Datum queryKeys[], bool
nullFlags[])
```

如果被索引项满足策略号为 n 的查询操作符，则返回 true；如果同时返回需要复核的指示，那么 true 也可以只表示它可能满足。该函数无法直接访问被索引项的值，因为 GIN 并不显式存储项。它所能利用的是这样一种信息：从查询中提取出的哪些键值出现在给定的被索引项中。check 数组长度为 nkeys，这与先前针对该 query datum 由 extractQuery 返回的键数量相同。如果被索引项包含相应查询键，则 check 数组中的对应元素为 true；也就是说，如果 (check[i] == true)，则 extractQuery 结果数组中的第 i 个键存在于该被索引项中。传入原始 query datum，是为了让 consistent 方法在需要时可以查看它；同样也会传入先前由 extractQuery 返回的 queryKeys[] 和 nullFlags[] 数组。extra_data 则是 extractQuery 返回的额外数据数组，如果没有则为 NULL。

当 extractQuery 在 queryKeys[] 中返回一个 null 键时，若被索引项包含 null 键，则对应的 check[] 元素为 true；也就是说，check[] 的语义类似于 IS NOT DISTINCT FROM。如果 consistent 函数需要区分普通值匹配与 null 匹配，它可以检查对应的 nullFlags[] 元素。

成功时，如果需要根据查询操作符重新检查堆元组，则 *recheck 应设置为 true；如果索引测试是精确的，则设置为 false。也就是说，返回 false 保证该堆元组不匹配查询；返回 true 且 *recheck 为 false 保证该堆元组匹配查询；返回 true 且 *recheck 为 true 则表示该堆元组可能匹配查询，因此需要取出该元组，并直接对最初被索引的项值重新计算查询操作符以完成复核。

```
GinTernaryValue triConsistent(GinTernaryValue check[], StrategyNumber
n, Datum query, int32 nkeys, Pointer extra_data[], Datum queryKeys[],
 bool nullFlags[])
```

triConsistent 与 consistent 类似，不过 check 向量中的元素不是布尔值，而是每个键都有三种可能的值：GIN_TRUE、GIN_FALSE 和 GIN_MAYBE。GIN_FALSE 和 GIN_TRUE 与普通布尔值含义相同，而 GIN_MAYBE 表示该键是否存在尚不确定。存在 GIN_MAYBE 值时，只有当无论索引项是否包含对应查询键，该项都确定匹配时，函数才应返回 GIN_TRUE。同样，只有当无论是否包含 GIN_MAYBE 键，该项都确定不匹配时，函数才能返回 GIN_FALSE。如果结果依赖于 GIN_MAYBE 条目，也就是说，无法根据已知的查询键确认或否定匹配，则函数必须返回 GIN_MAYBE。

当 check 向量中没有 GIN_MAYBE 值时，返回 GIN_MAYBE 等价于在布尔型 consistent 函数中设置 recheck 标志。

此外，GIN 必须有一种方法对存储在索引中的键值进行排序。操作符类可以通过指定一个比较方法来定义这种排序顺序：

```
int compare(Datum a, Datum b)
```

比较两个键（不是被索引项！），并返回一个整数：小于零、等于零或大于零，分别表示第一个键小于、等于或大于第二个键。null 键绝不会被传递给这个函数。

或者，如果操作符类没有提供 compare 方法，GIN 将查找该索引键数据类型的默认 B-树操作符类，并使用其比较函数。建议在仅面向单一数据类型的 GIN 操作符类中显式指定比较函数，因为查找 B-树操作符类会消耗少量处理器周期。不过，多态 GIN 操作符类（例如 array_ops）通常无法指定单一比较函数。

作为可选项，GIN 操作符类可以提供以下方法：

```
int comparePartial(Datum partial_key, Datum key, StrategyNumber n,
 Pointer extra_data)
```

比较部分匹配查询键与索引键。返回一个整数，其符号表示结果：小于零表示索引键不匹配查询，但索引扫描应继续；零表示索引键匹配查询；大于零表示索引扫描应停止，因为后续不可能再有匹配项。产生该部分匹配查询的操作符的策略号 `n` 会被传入，以便在需要其语义来决定何时结束扫描时使用。另外，`extra_data` 是由 `extractQuery` 生成的额外数据数组中的对应元素，如果没有则为 `NULL`。null 键绝不会被传递给这个函数。

要支持“部分匹配”查询，操作符类必须提供 `comparePartial` 方法，并且其 `extractQuery` 方法在遇到部分匹配查询时必须设置 `pmatch` 参数。详见第 66.4.2 节。

上文提到的各种 `Datum` 值，其实际数据类型会因操作符类而异。传给 `extractValue` 的项值始终是该操作符类的输入类型，而所有键值都必须是该类的 `STORAGE` 类型。传给 `extractQuery`、`consistent` 和 `triConsistent` 的 `query` 参数类型，是由该策略号标识的类成员操作符右侧输入类型。只要能够从中提取出正确类型的键值，它就不必与被索引类型相同。不过，建议在这三个支持函数的 SQL 声明中，对 `query` 参数使用该操作符类的被索引数据类型，尽管依据具体操作符，实际类型可能是别的类型。

66.4. 实现

在内部，一个 GIN 索引包含一个基于键构建的 B-树索引，其中每个键都是一个或多个被索引项中的某个元素（例如数组成员），而叶子页中的每个元组要么包含一个指向堆指针 B-树的指针（“倒排树”），要么在列表足够小、能够连同键值一起放入单个索引元组时，直接包含一个简单的堆指针列表（“倒排列表”）。

从 PostgreSQL 9.1 开始，索引中可以包含 null 键值。此外，对于那些自身为 null，或按照 `extractValue` 的定义不包含任何键的被索引项，索引中也会包含占位符 null 值。这样一来，应该找到空项的搜索就能够找到它们。

多列 GIN 索引的实现方式，是在组合值（列号，键值）之上构建一个单独的 B-树。不同列的键值可以具有不同类型。

66.4.1. GIN 快速更新技术

由于倒排索引的内在性质，更新 GIN 索引往往比较慢：插入或更新一个堆行，可能会导致向索引中执行多次插入（从被索引项中提取出的每个键都要插入一次）。从PostgreSQL 8.4 开始，GIN 可以通过把新元组插入一个临时的、未排序的待处理列表，来推迟其中的大部分工作。当表被清理或自动分析时，或者调用 `gin_clean_pending_list` 函数时，又或者待处理列表增长到大于gin_pending_list_limit 时，这些项就会被移入主要的 GIN 数据结构中，所用的是与初始索引创建时相同的批量插入技术。即便把额外的清理开销计算在内，这也会大幅提高 GIN 索引的更新速度。此外，这部分开销工作还可以由后台进程完成，而不是在前台查询处理过程中完成。

这种方法的主要缺点是，搜索除了要查找常规索引之外，还必须扫描待处理列表，因此大型待处理列表会显著拖慢搜索。另一个缺点是，虽然大多数更新都很快，但一旦某次更新使待处理列表变得“过大”，就会立刻触发一次清理周期，因此该次更新会比其他更新慢得多。恰当地使用自动清理可以将这两个问题都尽量减轻。

如果稳定的响应时间比更新速度更重要，可以通过关闭 GIN 索引的 `fastupdate` 存储参数来禁用待处理列表机制。详见 CREATE INDEX。

66.4.2. 部分匹配算法

GIN 可以支持“部分匹配”查询。在这种查询中，查询无法确定一个或多个键的精确匹配，但可能的匹配会落在一个相对狭窄的键值范围内（该范围位于由 `compare` 支持方法确定的键排序顺序中）。`extractQuery` 方法不是返回一个要精确匹配的键值，而是返回待搜索范围的下界键值，并将 `pmatch` 标志设为 true。随后使用 `comparePartial` 方法扫描该键范围。对于匹配的索引键，`comparePartial` 必须返回零；对于虽然不匹配但仍处在待搜索范围内的索引键，返回小于零；如果索引键已经超出了可能匹配的范围，则返回大于零。

66.5. GIN 提示和技巧

创建与插入

向 GIN 索引插入数据可能较慢，因为每个项很可能需要插入许多键。因此，对于向表中执行的批量插入，建议先删除 GIN 索引，待批量插入完成后再重建它。

从 PostgreSQL 8.4 开始，由于采用了延迟索引，这项建议已不那么必要（详见第 66.4.1 节）。但对于非常大的更新，删除并重建索引仍然可能是最佳选择。

maintenance_work_mem

GIN 索引的构建时间对 `maintenance_work_mem` 设置非常敏感；在创建索引时节省工作内存并不划算。

gin_pending_list_limit

在对现有 GIN 索引执行一系列插入且其 `fastupdate` 已启用时，系统会在待处理列表增长到超过 `gin_pending_list_limit` 时清理该列表。为避免观测到的响应时间波动，最好让待处理列表的清理在后台发生（即通过自动清理）。可以通过增大 `gin_pending_list_limit` 或让自动清理更积极，来避免前台清理操作。不过，提高清理触发阈值也意味着，一旦确实发生前台清理，耗时会更长。

可以通过更改存储参数为单个 GIN 索引覆盖 `gin_pending_list_limit`，从而让每个 GIN 索引拥有自己的清理阈值。例如，可以只提高那些更新很频繁的 GIN 索引的阈值，而把其他索引的阈值调低。

gin_fuzzy_search_limit

开发 GIN 索引的主要目标，是为 PostgreSQL 中高度可伸缩的全文检索提供支持，而全文检索返回一个非常大的结果集的情况也很常见。此外，当查询包含非常常见的词时，通常也会出现这种情况，以至于这个大型结果集本身并无多大用处。由于从磁盘读取大量元组并对其排序可能需要很长时间，这在生产环境中是不可接受的。（注意，索引搜索本身是非常快的。）

为了便于受控地执行这类查询，GIN 对返回行数设置了一个可配置的软上限，即配置参数 `gin_fuzzy_search_limit`。默认值为 0（表示无限制）。如果设置了非零限制，那么返回集合将是整个结果集的一个随机子集。

“软”的意思是，实际返回结果的数量可能会与指定的限制略有不同，这取决于查询以及系统随机数生成器的质量。

根据经验，数千量级的值（例如 5000 – 20000）效果不错。

66.6. 限制

GIN 假定可索引操作符是严格的。这意味着，当项值为 null 时，根本不会对其调用 `extractValue`（而是自动创建一个占位符索引项）；当查询值为 null 时，也不会调用 `extractQuery`（而是认为该查询不可满足）。不过要注意，非 null 的组合项或查询值内部包含的 null 键值仍然受支持。

66.7. 示例

PostgreSQL 核心发行版包含前面在表 66.1 中展示过的 GIN 操作符类。下列 `contrib` 模块也包含 GIN 操作符类：

`btree_gin`

为多种数据类型提供与 B-树等价的功能

`hstore`

用于存储（键，值）对的模块

`intarray`

对 `int[]` 的增强支持

`pg_trgm`

基于三字符组匹配的文本相似度

第 67 章 BRIN 索引

67.1. 简介

BRIN 是块范围索引（Block Range Index）的缩写。BRIN 旨在处理非常大的表，其中某些列与它们在表中的物理位置之间存在某种天然的相关性。块范围是表中物理上相邻的一组页；索引会为每个块范围存储一些摘要信息。例如，一个存储商店销售订单的表可能有一个日期列，表示每个订单的下单日期，而大多数情况下较早订单的条目也会较早出现在表中；一个存储邮政编码列的表，则可能会自然地把同一城市的所有邮政编码分组在一起。

BRIN 索引可以通过常规位图索引扫描来满足查询；对于每个范围，如果索引中存储的摘要信息与查询条件相一致，就会返回该范围内所有页上的全部元组。查询执行器负责重新检查这些元组，并丢弃不匹配查询条件的元组 – 换句话说，这些索引是有损的。由于 BRIN 索引非常小，与顺序扫描相比，扫描索引只会带来很小的额外开销，但可以避免扫描那些已知不包含匹配元组的大块表数据。

BRIN 索引所存储的具体数据，以及它能够满足的具体查询，取决于为该索引各列选择的操作符类。例如，具有线性排序顺序的数据类型可以使用在每个块范围内存储最小值和最大值的操作符类；几何类型则可能存储该块范围内所有对象的边界框。

块范围的大小在创建索引时由 `pages_per_range` 存储参数决定。索引项的数量等于该关系的页数除以为 `pages_per_range` 选择的值。因此，该值越小，索引就会越大（因为需要存储更多索引项），但与此同时，存储的摘要数据也会更精确，并且在索引扫描期间可以跳过更多数据块。

67.1.1. 索引维护

在创建索引时，会扫描所有现有堆页，并为每个范围创建一个摘要索引元组，末尾那个可能不完整的范围也包括在内。随着新页被数据填满，已经完成摘要的页范围会利用新元组中的数据更新其摘要信息。当创建了一个不属于最后一个已摘要范围的新页时，该范围不会自动获得摘要元组；这些元组会一直保持未摘要状态，直到稍后调用摘要操作，创建初始摘要。可以使用 `brin_summarize_range(regclass, bigint)` 或 `brin_summarize_new_values(regclass)` 函数手动调用这一过程；也可以在 `VACUUM` 处理该表时自动调用；或者在发生插入时，由 autovacuum 执行自动摘要。（最后一种触发方式默认禁用，可以通过`autosummarize` 参数启用。）反之，可以使用`brin_desummarize_range(regclass, bigint)` 函数对某个范围取消摘要；当现有值已经发生变化，以致索引元组不再是良好的表示时，这会很有用。

当启用自动摘要时，每当一个页范围被填满，就会向 autovacuum 发送请求，要求对该范围执行定向摘要。该请求会在同一数据库中的某个自动清理工作进程下一次结束运行时得到处理。如果请求队列已满，该请求就不会被记录，并会向服务器日志发送一条消息：

```
LOG:  request for BRIN range summarization for index "brin_wi_idx" page 128 was not recorded
```

发生这种情况时，该范围会在对该表执行下一次常规清理时正常完成摘要。

67.2. 内置操作符类

核心 PostgreSQL 发行版包含表 67.1 中所示的 BRIN 操作符类。

minmax 操作符类存储该范围内索引列中出现的最小值和最大值。inclusion 操作符类存储一个能够包含该范围内索引列值的值。

表 67.1. 内置 BRIN 操作符类

名称	索引数据类型	可索引操作符
abstime_minmax_ops	abstime	< <= = >= >
int8_minmax_ops	bigint	< <= = >= >
bit_minmax_ops	bit	< <= = >= >
varbit_minmax_ops	bit varying	< <= = >= >
box_inclusion_ops	box	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
bytea_minmax_ops	bytea	< <= = >= >
bpchar_minmax_ops	character	< <= = >= >
char_minmax_ops	"char"	< <= = >= >
date_minmax_ops	date	< <= = >= >
float8_minmax_ops	double precision	< <= = >= >
inet_minmax_ops	inet	< <= = >= >
network_inclusion_ops	inet	&& >>= <<= = >> <<
int4_minmax_ops	integer	< <= = >= >
interval_minmax_ops	interval	< <= = >= >
macaddr_minmax_ops	macaddr	< <= = >= >
macaddr8_minmax_ops	macaddr8	< <= = >= >
name_minmax_ops	name	< <= = >= >
numeric_minmax_ops	numeric	< <= = >= >
pg_lsn_minmax_ops	pg_lsn	< <= = >= >
oid_minmax_ops	oid	< <= = >= >
range_inclusion_ops	任意范围类型	<< &< && &> >> @> <@ -|- = < <= = > >=
float4_minmax_ops	real	< <= = >= >
reltime_minmax_ops	reltime	< <= = >= >
int2_minmax_ops	smallint	< <= = >= >
text_minmax_ops	text	< <= = >= >
tid_minmax_ops	tid	< <= = >= >
timestamp_minmax_ops	timestamp without time zone	< <= = >= >
timestamptz_minmax_ops	timestamp with time zone	< <= = >= >
time_minmax_ops	time without time zone	< <= = >= >
timetz_minmax_ops	time with time zone	< <= = >= >
uuid_minmax_ops	uuid	< <= = >= >

67.3. 可扩展性

BRIN 接口具有较高层次的抽象，访问方法实现者只需实现被访问数据类型的语义。BRIN 层本身负责并发、日志记录以及搜索索引结构。

要让一种 BRIN 访问方法工作起来，只需实现少数几个用户定义的方法，它们定义索引中存储的摘要值的行为，以及这些摘要值与扫描键之间的交互。简言之，BRIN 把可扩展性与通用性、代码重用以及清晰的接口结合在一起。

在 BRIN 中，操作符类必须提供以下四个方法：

`BrinOpcInfo *opcInfo(Oid type_oid)`

返回被索引列的摘要数据的内部信息。返回值必须指向一个用 palloc 分配的 `BrinOpcInfo`，其定义如下：

```
typedef struct BrinOpcInfo
{
    /* 此操作符类在一个索引列中存储的列数 */
    uint16      oi_nstored;

    /* 供操作符类私有使用的不透明指针 */
    void       *oi_opaque;

    /* 所存储列的类型缓存条目 */
    TypeCacheEntry *oi_typcache[FLEXIBLE_ARRAY_MEMBER];
} BrinOpcInfo;
```

`BrinOpcInfo.oi_opaque` 可供操作符类例程在索引扫描期间于各个支持函数之间传递信息。

`bool consistent(BrinDesc *bdesc, BrinValues *column, ScanKey key)`

返回该 ScanKey 是否与某个范围给定的索引值一致。要使用的属性编号作为扫描键的一部分传入。

`bool addValue(BrinDesc *bdesc, BrinValues *column, Datum newval, bool isnull)`

给定一个索引元组和一个被索引值，修改该元组中指定的属性，使其能够额外表示这个新值。如果对该元组做了任何修改，就返回 `true`。

`bool unionTuples(BrinDesc *bdesc, BrinValues *a, BrinValues *b)`

合并两个索引元组。给定两个索引元组，修改第一个元组中指定的属性，使其能够表示这两个元组。第二个元组不会被修改。

核心发行版包含两种操作符类支持：minmax 和 inclusion。针对核心数据类型，发行版会酌情附带使用它们的操作符类定义。用户也可以为其他数据类型定义等价的操作符类，而无需编写任何源代码；只要声明适当的系统目录项即可。注意，对操作符策略语义的某些假设嵌入在支持函数的源码中。

只要为上文所述的四个主要支持函数编写实现，也可以实现语义完全不同的操作符类。注意，不保证跨主版本的向后兼容性：例如，在后续版本中可能需要附加的支持函数。

要为实现全序集的数据类型编写操作符类，可以按表 67.2 所示，将 minmax 支持函数与相应操作符一起使用。所有操作符类成员（函数和操作符）都是必需的。

表 67.2. Minmax 操作符类的函数和支持编号

操作符类成员	对象
支持函数 1	内部函数`brin_minmax_opcinfo()`
支持函数 2	内部函数`brin_minmax_add_value()`
支持函数 3	内部函数`brin_minmax_consistent()`
支持函数 4	内部函数`brin_minmax_union()`
操作符策略 1	小于操作符
操作符策略 2	小于等于操作符
操作符策略 3	等于操作符
操作符策略 4	大于等于操作符
操作符策略 5	大于操作符

要为一种值能够被包含在另一种类型中的复杂数据类型编写操作符类，可以按表 67.3 所示，将 inclusion 支持函数与相应操作符一起使用。它只需要额外一个函数，而且该函数可以用任何语言编写。还可以定义更多函数以提供附加功能。所有操作符都是可选的。某些操作符依赖其他操作符，表中已列出这些依赖关系。

表 67.3. Inclusion 操作符类的函数和支持编号

操作符类成员	对象	依赖关系
支持函数 1	内部函数`brin_inclusion_opcinfo()`	
支持函数 2	内部函数`brin_inclusion_add_value()`	
支持函数 3	内部函数`brin_inclusion_consistent()`	
支持函数 4	内部函数`brin_inclusion_union()`	
支持函数 11	合并两个元素的函数	
支持函数 12	可选函数，检查两个元素是否可以合并	
支持函数 13	可选函数，检查一个元素是否被包含在另一个中	
支持函数 14	可选函数，用于检查元素是否为空	
操作符策略 1	位于左侧的操作符	操作符策略 4
操作符策略 2	不延伸到右侧的操作符	操作符策略 5
操作符策略 3	重叠操作符	
操作符策略 4	不延伸到左侧的操作符	操作符策略 1
操作符策略 5	位于右侧的操作符	操作符策略 2
操作符策略 6、18	相同或等于操作符	操作符策略 7
操作符策略 7、13、16、24、25	包含或等于操作符	
操作符策略 8、14、26、27	被包含或等于操作符	操作符策略 3
操作符策略 9	不延伸到上方的操作符	操作符策略 11
操作符策略 10	位于下方的操作符	操作符策略 12
操作符策略 11	位于上方的操作符	操作符策略 9

操作符类成员	对象	依赖关系
操作符策略 12	不延伸到下方的操作符	操作符策略 10
操作符策略 20	小于操作符	操作符策略 5
操作符策略 21	小于等于操作符	操作符策略 5
操作符策略 22	大于操作符	操作符策略 1
操作符策略 23	大于等于操作符	操作符策略 1

支持函数编号 1 到 10 保留给 BRIN 内部函数，因此 SQL 层函数从编号 11 开始。支持函数 11 是构建索引所需的主要函数。它应接受两个与操作符类数据类型相同的参数，并返回它们的并集。如果 inclusion 操作符类在定义时使用了 `STORAGE` 参数，则它可以存储具有不同数据类型的并集值。并集函数的返回值应与 `STORAGE` 数据类型匹配。

支持函数 12 和 14 用于处理内置数据类型中的不规则情况。函数 12 用于支持来自不同地址族、无法合并的网络地址。函数 14 用于支持空范围。函数 13 是可选但推荐的；它允许在把新值传给并集函数之前先进行检查。由于 BRIN 框架在并集未发生变化时可以跳过某些操作，使用这个函数可以提升索引性能。

minmax 和 inclusion 操作符类都支持跨数据类型操作符，但这样依赖关系会更加复杂。minmax 操作符类要求定义一整套两侧参数都具有相同数据类型的操作符。它还允许通过定义额外的操作符集合来支持附加数据类型。如表 67.3 所示，inclusion 操作符类的操作符策略依赖于另一种操作符策略，或者依赖于它们自身对应的同名操作符策略。这要求把依赖操作符定义为：左侧参数是 `STORAGE` 数据类型，右侧参数是其他受支持的数据类型。minmax 的示例见 `float4_minmax_ops`，inclusion 的示例见 `box_inclusion_ops`。

第 68 章 Hash 索引

68.1. 概述

PostgreSQL 提供了持久化的磁盘 hash 索引实现，并且完全支持崩溃恢复。任何数据类型都可以使用 hash 索引，包括那些没有明确定义线性顺序的数据类型。Hash 索引只存储被索引数据的 hash 值，因此对被索引数据列的大小没有限制。

Hash 索引仅支持单列索引，也不支持唯一性检查。

Hash 索引仅支持=操作符，因此指定范围操作的 WHERE 子句无法利用 hash 索引。

每个 hash 索引元组只存储 4 字节的 hash 值，而不存储实际的列值。因此，在为 UUID、URL 等较长的数据项建立索引时，hash 索引可能比 B-树小得多。由于没有列值，所有 hash 索引扫描也都是有损的。Hash 索引可以参与位图索引扫描和反向扫描。

Hash 索引最适合于在较大表上执行等值扫描、且以 SELECT 和 UPDATE 为主的工作负载。在 B-树索引中，搜索必须沿树向下直到找到叶页。在拥有数百万行的表中，这种向下遍历会增加访问数据的时间。在 hash 索引中，与叶页对应的页称为桶页。相比之下，hash 索引允许直接访问桶页，因此可能减少大表中的索引访问时间。这种“逻辑 I/O”的减少，在索引/数据大于 shared_buffers/RAM 时会更加明显。

Hash 索引在设计上能够应对 hash 值分布不均的情况。如果 hash 值分布均匀，那么直接访问桶页的效果很好。当插入导致桶页填满时，额外的溢出页就会链接到该特定桶页上，从而在局部扩展用于存放匹配该 hash 值的索引元组的空间。查询期间扫描某个 hash 桶时，需要遍历其全部溢出页。因此，对于某些数据，不平衡的 hash 索引在所需块访问次数方面实际上可能比 B-树更差。

由于会出现这些溢出情况，可以说 hash 索引最适合用于唯一值、近乎唯一值，或者每个 hash 桶中行数较少的数据。避免问题的一种可能办法，是使用部分索引条件把高度非唯一的值排除在索引之外，但这在很多情况下可能并不适用。

与 B-树一样，hash 索引也会执行简单的索引元组删除。这是一种延迟维护操作，用于删除那些已知可以安全删除的索引元组（即其项标识符的 LP_DEAD 位已经被设置的那些元组）。如果插入操作发现页上没有可用空间，我们会尝试通过移除死索引元组来避免创建新的溢出页。如果该页当时已被钉住，就无法执行移除。死索引指针也会在 VACUUM 期间被删除。

如果可能，VACUUM 还会尝试将索引元组集中存放在尽可能少的溢出页上，以最小化溢出链。如果某个溢出页变为空页，该溢出页就可以被回收并在其他桶中重用，不过我们从不将它们返还给操作系统。目前除了使用 REINDEX 重建 hash 索引之外，还没有缩小 hash 索引的方法。同样也没有减少桶数量的方法。

随着被索引行数的增长，Hash 索引可能会扩展桶页数量。hash 键到桶号的映射经过选择，使得索引能够增量扩展。当需要向索引中添加一个新桶时，恰好只需要“拆分”一个现有桶，并根据更新后的键到桶号映射，将其中一部分元组转移到新桶。

这种扩展是在前台进行的，因此可能会增加用户插入操作的执行时间。因此，hash 索引可能不适合行数快速增长的表。

68.2. 实现

hash 索引中有四种页：元页（第零页），其中包含静态分配的控制信息；主桶页；溢出页；以及位图页，后者用于跟踪已被释放且可供重用的溢出页。在寻址时，位图页被视为溢出页的一个子集。

扫描索引和插入元组都需要定位给定元组应当所在的桶。为此，我们需要元页中的桶计数、highmask 和 lowmask；但是，出于性能原因，每次执行这类操作都去锁定并钉住元页并不理想。相反，我们在每个后端的 relcache 条目中保留元页的一个缓存副本。只要目标桶自上次缓存刷新以来没有被拆分，这就会产生正确的桶映射。

主桶页和溢出页是独立分配的，因为任何一个给定索引相对于其桶数，可能需要更多或更少的溢出页。hash 实现代码使用了一组颇有意思的寻址规则，以支持可变数量的溢出页，同时又不必在主桶页创建后再移动它们。

被索引表中的每一行，都由 hash 索引中的单个索引元组表示。Hash 索引元组存储在桶页中，如果存在溢出页，也会存储在溢出页中。为了加快搜索速度，我们将任一索引页中的索引项按 hash 码排序，从而可以在索引页内使用二分查找。但请注意，并不假定同一桶中不同索引页之间的 hash 码具有任何相对顺序。

用于扩展 hash 索引的桶拆分算法过于复杂，这里不再展开说明；但在`src/backend/access/hash/README` 中有更详细的描述。该拆分算法具备崩溃安全性，如果未能成功完成，可以重新启动。

第 69 章 数据库物理存储

本章概述 PostgreSQL 数据库所使用的物理存储格式。

69.1. 数据库文件布局

本节从文件和目录的层次描述存储格式。

传统上，数据库集簇所使用的配置文件和数据文件都存放在集簇的数据目录中，这个目录通常称为 PGDATA（名字来自可用于定义它的环境变量）。PGDATA 的一个常见位置是 /var/lib/pgsql/data。同一台机器上可以存在多个由不同服务器实例管理的集簇。

PGDATA 目录包含若干子目录和控制文件，如表 69.1 所示。除这些必需项之外，集簇配置文件 postgresql.conf、pg_hba.conf 和 pg_ident.conf 传统上也存放在 PGDATA 中，不过也可以把它们放在其他地方。

表 69.1. PGDATA 的内容

项	描述
PG_VERSION	包含 PostgreSQL 主版本号的文件
base	包含各数据库子目录的子目录
current_logfiles	记录日志收集器当前正在写入的日志文件的文件
global	包含集簇范围内表的子目录，例如 pg_database
pg_commit_ts	包含事务提交时间戳数据的子目录
pg_dynshmem	包含动态共享内存子系统所用文件的子目录
pg_logical	包含逻辑解码状态数据的子目录
pg_multixact	包含多事务状态数据的子目录（用于共享行锁）
pg_notify	包含 LISTEN/NOTIFY 状态数据的子目录
pg_replslot	包含复制槽数据的子目录
pg_serial	包含已提交可串行化事务信息的子目录
pg_snapshots	包含导出快照的子目录
pg_stat	包含统计子系统永久文件的子目录
pg_stat_tmp	包含统计子系统临时文件的子目录
pg_subtrans	包含子事务状态数据的子目录
pg_tblspc	包含指向表空间的符号链接的子目录
pg_twophase	包含预备事务状态文件的子目录
pg_wal	包含 WAL（预写式日志）文件的子目录
pg_xact	包含事务提交状态数据的子目录
postgresql.auto.conf	用于存储通过 ALTER SYSTEM 设置的配置参数的文件
postmaster.opts	记录服务器上次启动时所用命令行选项的文件

项	描述
postmaster.pid	记录当前 postmaster 进程 ID（PID）、集簇数据目录路径、postmaster 启动时间戳、端口号、Unix 域套接字目录路径（在 Windows 上为空）、第一个有效的 listen_address（IP 地址或 *，若未监听 TCP 则为空），以及共享内存段 ID 的锁文件（服务器关闭后该文件不存在）

对于集簇中的每个数据库，PGDATA/base 中都有一个子目录，其名称是该数据库在 pg_database 中的 OID。这个子目录是该数据库文件的默认位置；特别是，它的系统目录就存放在这里。

每个表和索引都存储在单独的文件中。对于普通关系，这些文件以表或索引的文件节点编号命名，该编号可在 pg_class.relfilenode 中找到。但对于临时关系，文件名的形式为 t*BBB*_*FFF*，其中 *BBB* 是创建该文件的后端的 ID，*FFF* 是文件节点编号。无论哪种情况，除了主文件（也称主分支）之外，每个表和索引还有一个空闲空间映射（见第 69.3 节），用于存储该关系可用空闲空间的信息。空闲空间映射存放在以文件节点编号加上 _fsm 后缀命名的文件中。表还拥有可见性映射，存放在带有 _vm 后缀的分支中，用于跟踪哪些页面已知不含死元组。可见性映射在第 69.4 节中有进一步说明。不记录 WAL 的表和索引还有第三个分支，称为初始化分支，存放在带有 _init 后缀的分支中（见第 69.5 节）。

> **小心**
>
> 请注意，虽然表的文件节点往往与其 OID 相同，但这并非必然如此；有些操作，例如 TRUNCATE、REINDEX、CLUSTER 以及某些形式的 ALTER TABLE，会在保留 OID 的同时改变文件节点。不要假定文件节点和表 OID 一定相同。此外，对于某些系统目录（包括 pg_class 本身），pg_class.relfilenode 中存放的是零。这些目录的实际文件节点编号保存在更底层的数据结构中，可以使用 pg_relation_filenode() 函数取得。

当表或索引超过 1 GB 时，会被划分成若干个 1 GB 大小的段。第一段的文件名与文件节点相同，后续各段则命名为 filenode.1、filenode.2，依此类推。这样的安排避免了在文件大小受限的平台上出现问题。（实际上，1 GB 只是默认段大小。段大小可以通过配置选项 --with-segsize 在构建 PostgreSQL 时调整。）原则上，空闲空间映射和可见性映射分支也可能需要多个段，不过在实践中这不太可能发生。

如果一个表包含可能具有很大条目的列，它就会有关联的 TOAST 表，用于对那些大到无法直接保存在表行中的字段值进行行外存储。pg_class.reltoastrelid 会把该表关联到其 TOAST 表（如果存在的话）。更多信息见第 69.2 节。

表和索引的内容将在第 69.6 节中进一步讨论。

表空间使情况变得更复杂。每个用户定义的表空间都在 PGDATA/pg_tblspc 目录中有一个符号链接，指向其物理表空间目录（即该表空间的 CREATE TABLESPACE 命令中指定的位置）。该符号链接以表空间的 OID 命名。在物理表空间目录中，有一个名称依赖于 PostgreSQL 服务器版本的子目录，例如 PG_9.0_201008051。（使用这个子目录的原因是，为了让数据库的后续版本可以在不发生冲突的情况下复用同一个 CREATE TABLESPACE 位置值。）在这个版本相关的子目录中，对于每个在该表空间中拥有对象的数据库，都有一个以该数据库 OID 命名的子目录。表和索引就使用文件节点命名方案存储在该目录中。pg_default 表空间不通过 pg_tblspc 访问，而是对应于 PGDATA/base。同样，pg_global 表空间也不通过 pg_tblspc 访问，而是对应于 PGDATA/global。

`pg_relation_filepath()` 函数会显示任意关系的完整路径（相对于 `PGDATA`）。它常可用作记住上述众多规则的替代方法。但请记住，该函数只给出该关系主分支第一段的名字，若要找出与该关系关联的所有文件，你可能还需要追加段号以及/或者 `_fsm`、`_vm` 或 `_init`。

临时文件（用于诸如排序的数据多于内存可容纳量之类的操作）会创建在 `PGDATA /base/pgsql_tmp` 中；或者创建在表空间目录下的 `pgsql_tmp` 子目录中，如果为其指定的是不同于 `pg_default` 的表空间。临时文件的名称形式为 `pgsql_tmp`*`PPP`*`.`*`NNN`*，其中 *`PPP`* 是所属后端的 PID，*`NNN`* 用来区分该后端的不同临时文件。

69.2. TOAST

本节概述超长属性存储技术（TOAST, The Oversized-Attribute Storage Technique）。

PostgreSQL 使用固定页面大小（通常为 8 kB），并且不允许元组跨越多个页面。因此，无法直接存储非常大的字段值。为克服这一限制，大字段值会被压缩并且/或者拆分成多个物理行。这个过程对用户是透明的，而且对大部分后端代码的影响很小。这项技术被亲切地称为 TOAST（或者“自切片面包问世以来最棒的东西”）。TOAST 基础设施也被用于改进内存中大型数据值的处理。

只有某些数据类型支持 TOAST，因为没有必要让那些不可能产生大字段值的数据类型承担这部分额外开销。要支持 TOAST，数据类型必须具有变长（varlena）表示形式，在通常情况下，任何已存储值的第一个四字节字都包含该值以字节计的总长度（包括其自身）。TOAST 对该数据类型表示形式的其余部分没有约束。统称为经过 TOAST 处理的值的特殊表示形式，是通过修改或重新解释这个初始长度字来工作的。因此，支持可 TOAST 数据类型的 C 级函数，必须小心处理可能已经过 TOAST 处理的输入值：在被去 TOAST 化之前，一个输入值未必真正由四字节长度字和内容组成。（通常是在对输入值做任何处理之前调用 `PG_DETOAST_DATUM`，但在某些情况下也可以采用更高效的方法。详见第 38.12.1 节。）

TOAST 会占用 varlena 长度字中的两个位（在大端机器上是最高的两位，在小端机器上是最低的两位），从而把任何可 TOAST 数据类型值的逻辑大小限制为 1 GB（2^{30} - 1 字节）。当这两个位都为零时，该值就是这种数据类型的普通、未经 TOAST 处理的值，长度字的其余位给出该 datum 的总大小（包括长度字），以字节计。当最高位或最低位被置位时，该值使用的不是通常的四字节头部，而是单字节头部；该字节的其余位给出 datum 的总大小（包括长度字节），以字节计。这样既能高效存储短于 127 字节的值，又仍然允许数据类型在需要时增长到 1 GB。带单字节头部的值不按任何特定边界对齐，而带四字节头部的值至少按四字节边界对齐；省去这部分对齐填充后，相对于短值而言，可以显著节省空间。作为特殊情况，如果单字节头部的其余位全为零（对于包含长度字节自身的长度而言，这是不可能的），则该值是一个指向行外数据的指针，它可能有下面将描述的几种形式。这样一种 TOAST 指针的类型和大小，由 datum 第二个字节中存储的编码决定。最后，当最高位或最低位为零、但相邻的一位被置位时，该 datum 的内容已被压缩，使用前必须先解压。在这种情况下，四字节长度字的其余位给出的是压缩后 datum 的总大小，而不是原始数据的大小。请注意，行外数据也可能被压缩，但 varlena 头部不会告诉你是否发生了压缩，这一点要由 TOAST 指针的内容来说明。

如前所述，TOAST 指针 datum 有多种类型。最早、也最常见的一种，是指向存储在 TOAST 表中的行外数据的指针，该表与包含该 TOAST 指针 datum 本身的表相关联，但与之分离存储。当一个要存储到磁盘上的元组过大而无法原样存储时，这些磁盘上的指针 datum 会由 TOAST 管理代码（位于 `access/heap/tuptoaster.c`）创建。更多细节见第 69.2.1 节。另一种情况是，TOAST 指针 datum 可以包含指向内存中其他位置的行外数据的指针。这类 datum 必然是短命的，永远不会出现在磁盘上，但它们对于避免大型数据值的复制和冗余处理非常有用。更多细节见第 69.2.2 节。

无论是行内压缩数据还是行外压缩数据，其所用的压缩技术都是 LZ 压缩技术族中一种相当简单且非常快速的方法。详情见 `src/common/pg_lzcompress.c`。

69.2.1. 行外、磁盘上的 TOAST 存储

如果一个表的任意列是可 TOAST 的，该表就会有关联的 TOAST 表，其 OID 存放在表的 `pg_class.reltoastrelid` 项中。磁盘上的经过 TOAST 处理的值保存在该 TOAST 表中，下文会更详细地描述。

行外值会被划分成最多 `TOAST_MAX_CHUNK_SIZE` 字节的块（如果启用了压缩，则在压缩之后再分块；默认情况下该值的选取方式是让四个块行可以装入一页，因此大约是 2000 字节）。每个块都作为独立的一行，存储在所属表的 TOAST 表中。每个 TOAST 表都有 `chunk_id` 列（标识某个特定经过 TOAST 处理的值的 OID）、`chunk_seq` 列（该块在其值内的序号）以及 `chunk_data` 列（该块的实际数据）。在 `chunk_id` 和 `chunk_seq` 上建立的唯一索引可提供快速检索。因此，一个表示行外、磁盘上的经过 TOAST 处理的值的指针 datum，需要存储要查找的 TOAST 表的 OID，以及该特定值的 OID（即其 `chunk_id`）。为方便起见，指针 datum 还会存储逻辑 datum 大小（原始未压缩数据长度）和物理存储大小（如果应用了压缩，这两者会不同）。再加上 varlena 头部字节，磁盘上的 TOAST 指针 datum 的总大小因此恒为 18 字节，与所表示值的实际大小无关。

TOAST 管理代码只有在要存入表中的行值宽于 `TOAST_TUPLE_THRESHOLD` 字节（通常为 2 kB）时才会被触发。TOAST 代码会压缩并且/或者把字段值移到行外，直到该行值短于 `TOAST_TUPLE_TARGET` 字节（通常也为 2 kB，但可调整），或者已经无法再获得更多收益。在 UPDATE 操作中，未更改字段的值通常会原样保留；因此，如果更新一行时其行外值都没有变化，便不会产生 TOAST 成本。

TOAST 管理代码识别四种在磁盘上存储可 TOAST 的列的不同策略：

- `PLAIN` 既不允许压缩，也不允许行外存储。这是不可 TOAST 数据类型列唯一可能的策略。
- `EXTENDED` 同时允许压缩和行外存储。这是大多数可 TOAST 数据类型的默认策略。系统会先尝试压缩，如果行仍然过大，再使用行外存储。
- `EXTERNAL` 允许行外存储，但不允许压缩。使用 `EXTERNAL` 会让宽 `text` 和 `bytea` 列上的子串操作更快（代价是占用更多存储空间），因为在值未压缩时，这些操作经过优化，只需提取行外值中所需的部分。
- `MAIN` 允许压缩，但不允许行外存储。（实际上，对于这类列，仍然可能进行行外存储，但只有在别无他法、必须这样做才能让行足够小以放入页面时，才会把它作为最后手段。）

每种可 TOAST 数据类型都会为该类型的列指定默认策略，但某个表列的策略可以通过 `ALTER TABLE ... SET STORAGE` 来修改。

`TOAST_TUPLE_TARGET` 也可以针对每个表通过 `ALTER TABLE ... SET (toast_tuple_target = N)` 进行调整。

与允许行值跨页之类更直接的方法相比，这种方案有不少优点。假定查询通常是通过与相对较小的键值比较来限定的，执行器的大部分工作都会只使用主行项完成。经过 TOAST 处理的属性的大值只有在结果集发送给客户端时才会被取出（前提是它们确实被选中了）。因此，主表会小得多，其更多的行能够装入共享缓冲区缓存，这是没有行外存储时做不到的。排序集也会缩小，因此排序更常能够完全在内存中完成。一个小测试表明，一个包含典型 HTML 页面及其 URL 的表，其总存储量（包括 TOAST 表）大约只有原始数据大小的一半，而主表只包含全部数据的大约 10%（URL 以及一些较小的 HTML 页面）。与未进行 TOAST 处理的对照表相比，运行时没有差异；在那个对照表中，所有 HTML 页面都被裁剪到 7 kB 以内以便放得下。

69.2.2. 行外、内存中的 TOAST 存储

TOAST 指针可以指向不在磁盘上、而位于当前服务器进程内存中其他位置的数据。这类指针显然不可能长期存在，但仍然很有用。目前有两个子情况：指向间接数据的指针，以及指向展开数据的指针。

间接 TOAST 指针只是简单地指向某处内存中存放的一个非间接 varlena 值。这一情况最初只是作为概念验证而创建，但目前在逻辑解码期间会用到它，以避免可能不得不创建超过 1 GB 的物理元组（把所有行外字段值都拉入元组就可能导致这种情况）。这种机制的用途有限，因为创建该指针 datum 的一方必须完全负责确保被引用数据在指针可能存在的整个期间都保持有效，而且系统并没有任何基础设施来帮助做到这一点。

展开的 TOAST 指针对那些磁盘表示形式并不特别适合计算用途的复杂数据类型很有用。以标准的 PostgreSQL 数组 varlena 表示为例，它包含维度信息、一个空值位图（如果存在空值元素），然后依次保存所有元素的值。当元素类型本身也是变长时，定位第 *N* 个元素的唯一办法就是扫描它前面的所有元素。这种表示形式因其紧凑性而适合磁盘存储，但对于数组计算而言，一种“展开”或“解构”的表示会更好，因为其中所有元素的起始位置都已识别出来。TOAST 指针机制通过允许按引用传递的 Datum 指向标准 varlena 值（即磁盘表示）或者指向内存中某处展开表示的 TOAST 指针，来满足这一需要。这种展开表示的具体细节由数据类型自行决定，不过它必须具有标准头部，并满足 `src/include/utils/expandeddatum.h` 中给出的其他 API 要求。处理该数据类型的 C 级函数可以选择支持任一种表示。不知道展开表示、但只是对输入应用 `PG_DETOAST_DATUM` 的函数，会自动获得传统的 varlena 表示；因此，对展开表示的支持可以逐步引入，一次增加一个函数即可。

指向展开值的 TOAST 指针还可进一步分为可读写（read-write）和只读（read-only）指针。无论哪种方式，被指向的表示是相同的；但收到可读写指针的函数可以原地修改所引用的值，而收到只读指针的函数则不可以，如果它想生成该值的修改版本，必须先创建一个副本。这种区分以及一些相关约定，使得在查询执行期间可以避免不必要的展开值复制。

对于所有类型的内存中 TOAST 指针，TOAST 管理代码都会确保这类指针 datum 不会意外地被存储到磁盘上。内存中的 TOAST 指针在存储前会自动展开为普通的行内 varlena 值，如果这样会使承载它的元组过大，还可能进一步转换为磁盘上的 TOAST 指针。

69.3. 空闲空间映射

每个堆关系和索引关系（hash 索引除外）都有一个空闲空间映射（FSM），用于跟踪该关系中的可用空间。它与主关系数据并列存放在一个独立的关系分支中，其名称由该关系的文件节点编号加上 `_fsm` 后缀构成。例如，如果某关系的文件节点是 12345，那么 FSM 就存储在名为 `12345_fsm` 的文件中，与主关系文件位于同一目录。

空闲空间映射被组织成一棵由 FSM 页面构成的树。最底层的 FSM 页面存储每个堆页面（或索引页面）上的可用空闲空间，每个这样的页面用一个字节表示。上层页面则聚合来自下层的信息。

每个 FSM 页面内部都有一棵二叉树，以数组形式存储，每个节点占一个字节。每个叶子节点表示一个堆页面或下层 FSM 页面。每个非叶子节点中保存其子节点值中较大的那个，因此叶子节点中的最大值会保存在根节点。

请参见 `src/backend/storage/freespace/README`，了解 FSM 的结构以及如何更新和搜索它的更多细节。pg_freespacemap 模块可用于检查空闲空间映射中存储的信息。

69.4. 可见性映射

每个堆关系都有一个可见性映射（VM），用于跟踪哪些页面只包含已知对所有活动事务都可见的元组；它还会跟踪哪些页面只包含已冻结的元组。它与主关系数据并列存放在一个独立的关系分支中，名称由关系的文件节点编号加上 `_vm` 后缀构成。例如，如果某关系的文件节点是 12345，

那么 VM 会存储在名为 `12345_vm` 的文件中，与主关系文件位于同一目录。请注意，索引没有 VM。

可见性映射为每个堆页面存储两个位。第一位若被置位，表示该页面是全部可见（all-visible），换句话说，该页面不包含任何需要被清理的元组。这些信息还可以被仅索引扫描用来只依据索引元组回答查询。第二位若被置位，则表示页面上的所有元组都已经被冻结。这意味着，即使是防回卷清理也不必重新访问该页面。

这个映射是保守的，也就是说，只要某个位被置位，我们就能确定相应条件确实成立；但如果某个位没有置位，该条件既可能为真，也可能不为真。可见性映射位只会由 VACUUM 设置，但页面上的任何数据修改操作都会把它们清除。

pg_visibility 模块可用于检查可见性映射中存储的信息。

69.5. 初始化分支

每个不记录 WAL 的表以及该表上的每个索引，都有一个初始化分支。初始化分支是一个适当类型的空表或空索引。当不记录 WAL 的表因崩溃而必须重置为空时，初始化分支会被复制覆盖到主分支上，而其他分支都会被擦除（它们会在需要时自动重建）。

69.6. 数据库页面布局

本节概述 PostgreSQL 表和索引内部所使用的页面格式。[1]序列和 TOAST 表的格式与普通表相同。

在下面的说明中，假定一个字节包含 8 个位。另外，术语项指的是存储在页面上的单个数据值。在表中，项是一行；在索引中，项是一条索引条目。

每个表和索引都存储为固定大小的页面数组（通常为 8 kB，不过在编译服务器时可以选择不同的页面大小）。在表中，所有页面在逻辑上都是等价的，因此某个特定项（行）可以存放在任意页面中。在索引中，第一页通常保留为元页，用于保存控制信息；并且根据索引访问方法的不同，索引中还可能存在不同类型的页面。

表 69.2 展示了页面的总体布局。每个页面包含五个部分。

表 69.2. 总体页面布局

项	描述
PageHeaderData	长度为 24 字节。包含页面的一般信息，包括空闲空间指针。
ItemIdData	指向实际项的项标识符数组。每个条目都是一个（偏移量、长度）对。每项 4 字节。
空闲空间	尚未分配的空间。新的项标识符从该区域起始处分配，新的项从末尾处分配。
项	项本身。
特殊空间	索引访问方法特有的数据。不同方法存储不同数据。普通表中为空。

每个页面的前 24 个字节由页头（`PageHeaderData`）组成。它的格式详见表 69.3。第一个字段跟踪与该页面有关的最新 WAL 记录。第二个字段在启用了数据校验和时包含页面校验和。接下来是一个 2 字节字段，包含标志位。随后是三个 2 字节整数字段（`pd_lower`、`pd_upper` 和

[1] 实际上，索引访问方法不要求必须使用这种页面格式。现有的所有索引方法都使用这种基本格式，但索引元页中保存的数据通常并不遵循项布局规则。

pd_special）。它们分别保存从页面起始位置到未分配空间起始位置、未分配空间结束位置以及特殊空间起始位置的字节偏移量。页头接下来的 2 个字节 pd_pagesize_version 同时存储页面大小和版本指示器。从 PostgreSQL 8.3 起，版本号为 4；PostgreSQL 8.1 和 8.2 使用版本号 3；PostgreSQL 8.0 使用版本号 2；PostgreSQL 7.3 和 7.4 使用版本号 1；更早版本使用版本号 0。（这些版本中的大多数，其基本页面布局和页头格式并未改变，但堆行头部的布局发生过变化。）页面大小字段基本上只是用于交叉检查；一个安装中并不支持同时存在多种页面大小。最后一个字段是一个提示，用于显示对页面进行剪枝是否可能有利：它跟踪页面上最老的、尚未剪枝的 XMAX。

表 69.3. PageHeaderData 布局

字段	类型	长度	描述
pd_lsn	PageXLogRecPtr	8 字节	LSN：上次修改此页面的 WAL 记录最后一个字节之后的下一个字节
pd_checksum	uint16	2 字节	页面校验和
pd_flags	uint16	2 字节	标志位
pd_lower	LocationIndex	2 字节	到空闲空间起始位置的偏移量
pd_upper	LocationIndex	2 字节	到空闲空间结束位置的偏移量
pd_special	LocationIndex	2 字节	到特殊空间起始位置的偏移量
pd_pagesize_version	uint16	2 字节	页面大小和布局版本号信息
pd_prune_xid	TransactionId	4 字节	页面上最老的未剪枝 XMAX，若无则为零

所有细节都可以在 src/include/storage/bufpage.h 中找到。

页头之后是项标识符（ItemIdData），每个需要四个字节。一个项标识符包含项起始位置的字节偏移量、其字节长度，以及若干影响解释方式的属性位。新的项标识符会根据需要从未分配空间的起始处分配。当前已有多少个项标识符，可以通过查看 pd_lower 得知；分配新标识符时它会增加。因为一个项标识符在被释放之前永远不会移动，所以即使项本身为了整理空闲空间而在页面内移动，其索引仍然可以被长期用来引用该项。事实上，每个指向项的指针（ItemPointer，也称 CTID）在 PostgreSQL 中都由页号和项标识符索引构成。

项本身存储在从未分配空间末尾开始、向后分配的区域中。其确切结构取决于表要包含什么内容。表和序列都使用一种名为 HeapTupleHeaderData 的结构体，如下所述。

最后一部分是“特殊部分”，其中可以存放访问方法希望保存的任何内容。例如，B-树索引会在这里保存指向页面左、右兄弟的链接，以及其他一些与索引结构相关的数据。普通表完全不使用特殊部分（通过将 pd_special 设为页面大小来表示）。

69.6.1. 表行布局

所有表行的结构都相同。它们都有一个固定大小的头部（在大多数机器上占 23 字节），后面跟着可选的空值位图、可选的对象 ID 字段以及用户数据。头部的详细格式见表 69.4。实际用户数据（行的各列）从 t_hoff 指示的偏移位置开始，它必须始终是该平台 MAXALIGN 对齐单位的整数倍。只有当 HEAP_HASNULL 位在 t_infomask 中被置位时，空值位图才存在。若存在，它紧随固定头部之后，并占用足够多的字节，以便为每个数据列提供一位（也就是说，总共为 t_natts 位）。在这个位图中，1 表示非空，0 表示空值。当位图不存在时，假定所有列都非空。只有当 HEAP_HASOID 位在 t_infomask 中被置位时，对象 ID 才存在。若存在，它位于

t_hoff 边界之前。为了让 t_hoff 成为 MAXALIGN 的整数倍所需的任何填充，都会出现在空值位图与对象 ID 之间。（这又反过来保证了对象 ID 的对齐是合适的。）

表 69.4. HeapTupleHeaderData 布局

字段	类型	长度	描述
t_xmin	TransactionId	4 字节	插入 XID 标记
t_xmax	TransactionId	4 字节	删除 XID 标记
t_cid	CommandId	4 字节	插入和/或删除 CID 标记（与 t_xvac 重叠）
t_xvac	TransactionId	4 字节	VACUUM 操作移动某个行版本时的 XID
t_ctid	ItemPointerData	6 字节	本行版本或较新的行版本的当前 TID
t_infomask2	uint16	2 字节	属性个数以及各种标志位
t_infomask	uint16	2 字节	各种标志位
t_hoff	uint8	1 字节	到用户数据的偏移量

所有细节都可以在 src/include/access/htup_details.h 中找到。

要解释实际数据，必须借助从其他表中获得的信息，其中大部分来自 pg_attribute。识别字段位置所需的关键值是 attlen 和 attalign。除非所有字段都是定宽且没有空值，否则没有办法直接取得某个特定属性。所有这些技巧都封装在 heap_getattr、fastgetattr 和 heap_getsysattr 这些函数中。

读取数据时，需要依次检查每个属性。首先根据空值位图判断该字段是否为 NULL。如果是，就继续下一个。然后确认对齐是否正确。如果字段是定宽字段，那么它的所有字节都是直接摆放的；如果它是变长字段（attlen = -1），情况就会稍复杂一些。所有变长数据类型都共享一个通用头部结构体 struct varlena，其中包含已存储值的总长度以及一些标志位。根据这些标志，数据可能是行内存储的，也可能位于 TOAST 表中；它也可能是经过压缩的（见第 69.2 节）。

69.7. 堆内元组（HOT）

为了支持高并发，PostgreSQL 使用多版本并发控制（MVCC）来存储行。不过，MVCC 对更新查询也有一些缺点。具体来说，更新需要向表中加入新的行版本。这还可能要求为每个被更新的行新增索引条目，而移除旧的行版本及其索引条目可能代价很高。

为了帮助降低更新的开销，PostgreSQL 提供了一种名为堆内元组（HOT）的优化。该优化在以下情况下可行：

- 更新不会修改被该表索引（包括表达式索引和部分索引）引用的任何列。
- 包含旧行的页面上有足够的空闲空间容纳更新后的行。

在这种情况下，堆内元组提供两项优化：

- 不需要新增用于表示更新后行的新索引条目。
- 更新行的旧版本可以在日常操作期间被完全移除，包括在执行 SELECT 时，而不必依赖周期性的清理操作。（这是因为索引并不引用它们的页面项标识符。）

总之，只有在索引使用的列未被更新时，才能产生堆内元组更新。要提高页面拥有足够空间进行 HOT 更新的可能性，你可以降低表的 `fillfactor`。即使不这样做，HOT 更新也仍然会发生，因为新行会自然迁移到新页面，而那些具有足够空闲空间容纳新行版本的现有页面也会被利用。系统视图 pg_stat_all_tables 可以用来监控 HOT 更新和非 HOT 更新的发生情况。

第 70 章 系统目录声明与初始内容

PostgreSQL 使用许多不同的系统目录来跟踪数据库对象（例如表和函数）的存在及其属性。系统目录与普通用户表在物理上没有区别，但后端 C 代码知道每个目录的结构和属性，并且能够在较低层次上直接操作它。因此，例如，试图在运行中修改某个目录的结构并不可取；那会破坏 C 代码中关于目录行布局方式的既有假设。不过，目录的结构可能会在主版本之间发生变化。

目录的结构声明在源码树 `src/include/catalog/` 目录中的、采用特殊格式的 C 头文件里。每个目录都有一个以该目录命名的头文件（例如，`pg_class` 对应 `pg_class.h`），其中定义了该目录具有的列集合，以及 OID 等其他一些基本属性。定义目录结构的其他关键文件包括 `indexing.h`，它定义所有系统目录上的索引，以及 `toasting.h`，它为需要 TOAST 表的目录定义 TOAST 表。

许多目录都有初始数据，这些数据必须在 initdb 的“引导”阶段装入其中，才能让系统达到可以执行 SQL 命令的程度。（例如，`pg_class.h` 必须包含它自身的一条记录，以及每个其他系统目录和索引的一条记录。）这些初始数据以可编辑形式保存在同样位于 `src/include/catalog/` 目录中的数据文件里。例如，`pg_proc.dat` 描述了必须插入到 `pg_proc` 目录中的所有初始行。

为了创建目录文件并将这些初始数据装入其中，以引导模式运行的后端会读取一个包含命令和初始数据的 BKI（Backend Interface，后端接口）文件。在构建 PostgreSQL 发行版时，会由名为 `genbki.pl` 的 Perl 脚本根据上述头文件和数据文件生成该模式所使用的 `postgres.bki` 文件。尽管 `postgres.bki` 针对特定的 PostgreSQL 发行版，但它与平台无关，并会安装到安装树的 `share` 子目录中。

`genbki.pl` 还会为每个目录生成一个派生头文件，例如为 `pg_class` 目录生成 `pg_class_d.h`。该文件包含自动生成的宏定义，也可能包含其他宏、枚举声明等，对于读取特定目录的客户端 C 代码会很有用。

大多数 Postgres 开发者并不需要直接关心 BKI 文件，但后端中几乎任何稍有分量的功能新增，都需要修改目录头文件和/或初始数据文件。本章余下部分将提供一些相关信息，并且为求完整，也会描述 BKI 文件格式。

70.1. 系统目录声明规则

目录头文件的关键部分是一个 C 结构定义，用于描述目录中每一行的布局。它以 `CATALOG` 宏开头；对 C 编译器而言，这只是 `typedef struct FormData_`*catalogname* 的简写。结构中的每个字段对应一个目录列。可以使用 `genbki.h` 中描述的 BKI 属性宏来标注字段，例如定义字段的默认值，或者标记其可空或不可空。还可以使用 `genbki.h` 中描述的其他一些 BKI 属性宏来标注 `CATALOG` 行，定义整个目录的其他属性，例如它是否具有 OID（默认具有）。

系统目录缓存代码（以及一般而言大多数操作目录的代码）都假定所有系统目录元组的定长部分确实存在，因为它会把这个 C 结构体声明映射到这些部分上。因此，所有变长字段和可空字段都必须放在末尾，并且不能作为结构体字段来访问。例如，如果你试图把 `pg_type.typrelid` 设为 NULL，那么当某段代码尝试引用 `typetup->typrelid` 时就会失败；更糟的是，也可能在引用 `typetup->typelem` 时失败，因为后者紧跟在 `typrelid` 之后。这会导致随机错误，甚至段错误。

为了在一定程度上防范这类错误，变长字段或可空字段不应该直接暴露给 C 编译器。实现方法是把它们包在 `#ifdef CATALOG_VARLEN` ... `#endif` 中（其中 `CATALOG_VARLEN` 是一个永远不会被定义的符号）。这样可以防止 C 代码不小心去访问那些可能不存在、或位于其他偏移位置的字段。作为防止创建错误行的另一层保护，我们要求所有本应不可空的列都在 `pg_attribute` 中被标记出来。如果目录列是定宽的，且其前面没有任何可空列，引导代码会自动把它标记

为 NOT NULL。在这一规则不足以覆盖的地方，可以按需使用 BKI_FORCE_NOT_NULL 和 BKI_FORCE_NULL 标注来强制正确的标记。

前端代码不应该包含任何 pg_xxx.h 目录头文件，因为这些文件可能含有在后端之外无法编译的 C 代码。（通常，这是因为这些文件也包含了 src/backend/catalog/ 中文件中的函数声明。）前端代码应改为包含相应生成的 pg_xxx_d.h 头文件，其中会带有 OID #define 以及其他任何可能对客户端有用的数据。如果你希望目录头文件中的宏或其他代码对前端代码可见，可以在那一部分外围加上 #ifdef EXPOSE_TO_CLIENT_CODE ... #endif，以指示 genbki.pl 把该部分复制到 pg_xxx_d.h 头文件中。

有少数目录基础性强到这样的程度，以致它们甚至不能通过大多数目录所使用的 BKI create 命令来创建，因为该命令需要向这些目录中写入信息来描述新目录。这些目录称为引导目录，而定义这样的目录需要额外做很多工作：你必须在 pg_class 和 pg_type 的预装载内容中手工为它们准备合适的条目，并且当该目录的结构后续发生变化时，还需要更新这些条目。（引导目录在 pg_attribute 中也需要预装载条目，不过幸运的是，如今 genbki.pl 会处理这件事。）如果可能的话，应尽量避免让新目录成为引导目录。

70.2. 系统目录初始数据

每个含有手工创建初始数据的目录（有些目录没有）都有一个对应的 .dat 文件，其中以可编辑格式保存该目录的初始数据。

70.2.1. 数据文件格式

每个 .dat 文件都包含 Perl 数据结构字面量，只需直接用 eval 求值，就会得到一个由 hash 引用数组组成的内存数据结构，其中每个数组元素对应一行目录数据。下面摘自 pg_database.dat 的一小段、经过轻微修改的内容，可以展示其关键特性：

```
[

# A comment could appear here.
{ oid => '1', oid_symbol => 'TemplateDbOid',
  descr => 'database\'s default template',
  datname => 'template1', datdba => 'PGUID', encoding => 'ENCODING',
  datcollate => 'LC_COLLATE', datctype => 'LC_CTYPE', datistemplate => 't',
  datallowconn => 't', datconnlimit => '-1', datlastsysoid => '0',
  datfrozenxid => '0', datminmxid => '1', dattablespace => '1663',
  datacl => '_null_' },

]
```

需要注意的几点：

- 整个文件的布局是：一个左方括号、一个或多个花括号集合（每个集合代表一行目录数据），最后是一个右方括号。每个右花括号后面都要写一个逗号。
- 在每个目录行中，写出以逗号分隔的 *key* => *value* 对。允许的 *key* 包括目录列名，以及元数据键 oid、oid_symbol 和 descr。（oid 和 oid_symbol 的用法在下文第 70.2.2 节中说明。descr 提供对象的描述字符串，它会按情况插入 pg_description 或 pg_shdescription。）元数据键是可选的，但目录定义的列必须全部提供，除非目录的 .h 文件为该列指定了默认值。
- 所有值都必须用单引号括起来。值内部出现的单引号要用反斜线转义。作为数据的反斜线可以写成双反斜线，但也不必；这遵循 Perl 对简单单引号字面量的规则。注意，作为数据出现的反斜线也会按照转义字符串常量相同的规则被引导扫描器当作转义处理（见第 4.1.2.2 节）；例如

\t 会转换为一个制表符。如果你确实希望最终值里包含一个反斜线，就需要写四个：Perl 会先去掉两个，留给引导扫描器看到的是 \\。

- NULL 值用 _null_ 表示。（注意，没有办法创建一个值恰好等于这个字符串。）
- 注释以前缀 # 表示，并且必须单独成行。
- 为提高可读性，如果字段值是其他目录条目的 OID，可以用名字代替数值 OID 来表示。下文第 70.2.3 节将对此作出说明。
- 由于 hash 是无序数据结构，字段顺序和行布局在语义上并不重要。不过，为了保持外观一致，我们规定了几条由格式化脚本 reformat_dat_file.pl 应用的规则：
  - 在每对花括号内，元数据字段 oid、oid_symbol 和 descr（如果存在）按此顺序排在最前，然后是按定义顺序排列的目录自身字段。
  - 如有可能，会根据需要在字段之间插入换行，以便把行长限制在 80 个字符以内。元数据字段和普通字段之间也会插入一次换行。
  - 如果目录的 .h 文件为某列指定了默认值，而某条数据又使用了同样的值，reformat_dat_file.pl 就会把该字段从数据文件中省略掉，以保持数据表示紧凑。
  - reformat_dat_file.pl 会原样保留空行和注释行。

  建议在提交目录数据补丁前先运行 reformat_dat_file.pl。为方便起见，你可以直接切换到 src/include/catalog/ 并运行 make reformat-dat-files。
- 如果你想新增一种缩小数据表示的方法，就必须在 reformat_dat_file.pl 中实现它，并且还要让 Catalog::ParseData() 知道如何把数据重新展开为完整表示。

70.2.2. OID 分配

出现在初始数据中的目录行，可以通过写一个 oid => *nnnn* 元数据字段来手工指定 OID。此外，如果已经指定了 OID，还可以通过写一个 oid_symbol => *name* 元数据字段，为该 OID 创建一个 C 宏。

如果其他预装载行通过 OID 引用某个预装载目录行，该行就必须具有预先分配的 OID。如果 C 代码需要引用该行的 OID，也必须预先分配。若这两种情况均不适用，可以省略 oid 元数据字段，此时引导代码会自动分配 OID，或者在没有 OID 的目录中将其保留为零。实践中，即使只有部分行实际被交叉引用，我们通常也会为某个目录中的所有预装载行预先分配 OID，或者一个都不预先分配。

在 C 代码里直接写任何 OID 的实际数值，都被认为是很不好的做法；应始终改用宏。对 pg_proc OID 的直接引用十分常见，因此有一个专门机制可以自动创建所需宏；见 src/backend/utils/Gen_fmgrtab.pl。类似地，pg_type OID 也有一种自动创建宏的方法，只是出于历史原因，做法并不完全相同。因此，这两个目录中并不需要 oid_symbol 项。同样，系统目录和索引的 pg_class OID 宏也是自动建立的。对于其他所有系统目录，你必须通过 oid_symbol 项手工指定所需的宏。

要为新的预装载行找到可用 OID，请运行脚本 src/include/catalog/unused_oids。它会输出未使用 OID 的范围（包含两个端点，例如输出行 “45-900” 表示 OID 45 到 900 尚未分配）。目前，OID 1-9999 保留供手动分配；unused_oids 脚本只是检查目录头文件和 .dat 文件，找出哪些 OID 没有出现。还可以使用 duplicate_oids 脚本检查错误。（genbki.pl 也会在编译时检测重复 OID。）

引导运行开始时，OID 计数器从 10000 开始。如果目录行所在的表需要 OID，而又未通过 oid 字段预先分配 OID，那么该行将获得一个不小于 10000 的 OID。

70.2.3. OID 引用查找

从一个初始目录行到另一个初始目录行的交叉引用，可以直接写出被引用行预先分配的 OID。但这容易出错，也难以理解，因此对于经常被引用的目录，genbki.pl 提供了使用符号引用的机制。目前，访问方法、函数、操作符、操作符类、操作符族和类型的引用可以使用此机制。规则如下：

- 要在某个目录列中启用符号引用，可以在列定义后附加 BKI_LOOKUP(*lookuprule*)，其中 *lookuprule* 为 pg_am、pg_proc、pg_operator、pg_opclass、pg_opfamily 或 pg_type。BKI_LOOKUP 可以附加在类型为 Oid、regproc、oidvector 或 Oid[] 的列上；在后两种情况下，它表示对数组中的每个元素都执行一次查找。
- 在这样的列中，除用 0 表示 InvalidOid 外，所有条目都必须使用符号形式。（如果该列声明为 regproc，还可以选择用 - 代替 0。）genbki.pl 会对无法识别的名字发出警告。
- 访问方法直接用名称表示，类型也是如此。类型名必须与被引用的 pg_type 条目中的 typname 完全一致；不能使用诸如用 integer 代表 int4 这样的别名。
- 如果函数的 proname 在 pg_proc.dat 条目中唯一，那么可以用它来表示函数（这和 regproc 输入的行为类似）。否则，应把它写成 *proname(argtypename,argtypename,...)* 的形式，类似 regprocedure。参数类型名的拼写必须与该 pg_proc.dat 条目中 proargtypes 字段里的写法完全一致。不要插入任何空格。
- 操作符写成 *oprname(lefttype,righttype)* 的形式，其中类型名必须与 pg_operator.dat 条目中 oprleft 和 oprright 字段里的写法完全一致。（对于一元操作符中省略的操作数，写 0。）
- 操作符类和操作符族的名字只在某个访问方法内部唯一，因此它们写成 *access_method_name/object_name* 的形式。
- 在以上所有情形中，都不支持模式限定；引导期间创建的所有对象都应位于 pg_catalog 模式中。

genbki.pl 在运行时会解析所有符号引用，并在输出的 BKI 文件中写入简单的数字 OID。因此，引导后端无需处理符号引用。

70.2.4. 编辑数据文件的方法

下面给出一些建议，说明在更新目录数据文件时，执行常见任务最简便的方法。

向目录中增加一个带默认值的新列：. 在头文件中加入该列，并加上 BKI_DEFAULT(*value*) 标注。数据文件只需在需要非默认值的现有行中补上该字段。

为现有列增加一个原本没有的默认值：. 在头文件中加上 BKI_DEFAULT 标注，然后运行 make reformat-dat-files，移除现在已经冗余的字段项。

移除一列，不论它是否有默认值：. 从头文件中删掉该列，然后运行 make reformat-dat-files，移除现在已经无用的字段项。

修改或移除现有默认值：. 不能只改头文件，因为那样会导致当前数据被错误解释。首先运行 make expand-dat-files，把所有默认值都显式写入数据文件；然后修改或移除 BKI_DEFAULT 标注；再运行 make reformat-dat-files，重新去掉多余字段。

临时性的批量编辑：. reformat_dat_file.pl 可以经过调整来执行多种批量更改。请查找其中说明可以插入一次性代码的位置的块注释。在下面的示例中，我们会将pg_proc 中的两个布尔字段合并为一个 char 字段：

1. 在 pg_proc.h 中加入一个带默认值的新列：

```
+    /* see PROKIND_ categories below */
+    char        prokind BKI_DEFAULT(f);
```

2. 基于 reformat_dat_file.pl 新建一个脚本，在处理过程中插入合适的值：

```
-           # At this point we have the full row in memory as a hash
-           # and can do any operations we want. As written, it only
-           # removes default values, but this script can be adapted to
-           # do one-off bulk-editing.
+           # One-off change to migrate to prokind
+           # Default has already been filled in by now, so change to other
+           # values as appropriate
+           if ($values{proisagg} eq 't')
+           {
+               $values{prokind} = 'a';
+           }
+           elsif ($values{proiswindow} eq 't')
+           {
+               $values{prokind} = 'w';
+           }
```

3. 运行这个新脚本：

```
$ cd src/include/catalog
$ perl  rewrite_dat_with_prokind.pl  pg_proc.dat
```

   此时，pg_proc.dat 会同时具有 prokind、proisagg 和 proiswindow 这三列，不过它们只会出现在取值非默认的那些行里。

4. 从 pg_proc.h 中移除旧列：

```
-    /* is it an aggregate? */
-    bool        proisagg BKI_DEFAULT(f);
-
-    /* is it a window function? */
-    bool        proiswindow BKI_DEFAULT(f);
```

5. 最后，运行 make reformat-dat-files，从 pg_proc.dat 中移除无用的旧项。

更多批量编辑脚本示例，请参见convert_oid2name.pl 和 remove_pg_type_oid_symbols.pl，它们附在以下消息中：https://www.postgresql.org/message-id/CAJVSVGVX8gXnPm+Xa=DxR7kFYprcQ1tNcCT5D0O3ShfnM6jehA@mail.gmail.com

70.3. BKI 文件格式

本节描述 PostgreSQL 后端如何解释 BKI 文件。如果手边有一份 postgres.bki 文件作为示例，这段说明会更容易理解。

BKI 输入由一系列命令组成。根据命令语法的不同，每条命令由若干词元组成。词元通常由空白分隔，但如果不会产生歧义，也可以不分隔。没有专门的命令分隔符；语法上不可能属于前一条命令的下一个词元，就会开始一条新命令。（通常为了清晰，你会把新命令写在新的一行。）词元可以是某些关键字、特殊字符（圆括号、逗号等）、数字，或者双引号字符串。所有内容都区分大小写。

以 # 开头的行会被忽略。

70.4. BKI 命令

create *tablename* *tableoid* [bootstrap] [shared_relation] [without_oids] [rowtype_oid *oid*] (*name1* = *type1* [FORCE NOT NULL | FORCE NULL] [, *name2* = *type2* [FORCE NOT NULL | FORCE NULL], ...])

创建一个名为 *tablename*、OID 为 *tableoid* 的表，其列定义在括号中给出。

bootstrap.c 直接支持以下列类型：bool、bytea、char（1 字节）、name、int2、int4、regproc、regclass、regtype、text、oid、tid、xid、cid、int2vector、oidvector、_int4（数组）、_text（数组）、_oid（数组）、_char（数组）以及 _aclitem（数组）。虽然也可以创建包含其他类型列的表，但这必须等到 pg_type 创建完成并填入合适条目之后才能做到。（实际上，这意味着引导目录中只能使用这些列类型，而非引导目录则可以包含任何内置类型。）

指定 bootstrap 时，表只会在磁盘上创建；不会在 pg_class、pg_attribute 等目录中为它写入任何信息。因此，在通过 insert 命令以较为原始的方式补入这些条目之前，普通 SQL 操作无法访问该表。这个选项用于创建 pg_class 等目录自身。

如果指定了 shared_relation，该表就会作为共享表创建。除非指定 without_oids，否则该表将具有 OID。该表的行类型 OID（即 pg_type 中对应的 OID）还可以通过 rowtype_oid 子句显式指定；如果不指定，就会自动为其生成一个 OID。（如果指定了 bootstrap，那么 rowtype_oid 子句其实没有用，但仍然可以写上，作为文档说明。）

open *tablename*

打开名为 *tablename* 的表，以便插入数据。当前任何已打开的表都会被关闭。

close *tablename*

关闭当前打开的表。必须给出表名，以便进行交叉校验。

insert [OID = *oid_value*] (*value1* *value2* ...)

向当前打开的表插入一行新数据，使用 *value1*、*value2* 等作为各列的值，使用 *oid_value* 作为其 OID。如果 *oid_value* 为零（0）或省略该子句，并且该表具有 OID，则分配下一个可用 OID。

NULL 值可以用特殊关键字 _null_ 表示。凡是不像标识符或纯数字字符串的值，都必须用双引号括起来。

declare [unique] index *indexname* *indexoid* on *tablename* using *amname* (*opclass1* *name1* [, ...])

在名为 *tablename* 的表上，使用 *amname* 访问方法，创建一个名为 *indexname*、OID 为 *indexoid* 的索引。要建立索引的字段分别称为 *name1*、*name2* 等；使用的操作符类分别为

opclass1、*opclass2* 等。该命令会创建索引文件并写入相应的目录条目，但不会初始化索引内容。

`declare toast` *toasttableoid toastindexoid* `on` *tablename*

为名为 *tablename* 的表创建一个 TOAST 表。该 TOAST 表的 OID 设为 *toasttableoid*，其索引的 OID 设为 *toastindexoid*。与 `declare index` 一样，索引填充会被延后。

`build indices`

填充之前已声明的索引。

70.5. 引导 BKI 文件的结构

在 `open` 命令可以使用之前，它所依赖的那些表必须已经存在，并且其中已经有了将要被打开之表对应的条目。（这些最基本的表包括 `pg_class`、`pg_attribute`、`pg_proc` 和 `pg_type`。）为了让这些表本身也能够被填充，带 `bootstrap` 选项的 `create` 命令会隐式打开刚创建的表，以便插入数据。

同样，`declare index` 和 `declare toast` 命令也必须等到它们所需的系统目录已经创建并填充完毕之后才能使用。

因此，`postgres.bki` 文件的结构必须如下：

1. 对某个关键表执行 `create bootstrap`
2. 用 `insert` 插入至少足以描述这些关键表的数据
3. `close`
4. 对其他关键表重复上述步骤。
5. 对某个非关键表执行 `create`（不带 `bootstrap`）
6. `open`
7. 用 `insert` 插入所需数据
8. `close`
9. 对其他非关键表重复上述步骤。
10. 定义索引和 TOAST 表。
11. `build indices`

无疑还存在其他未文档化的顺序依赖。

70.6. BKI 示例

以下命令序列会创建表 `test_table`，其 OID 为 420，包含两列`cola` 和 `colb`，类型分别为 `int4` 和 `text`，并向表中插入两行：

```
create test_table 420 (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

第 71 章 规划器如何使用统计信息

本章建立在第 14.1 节和第 14.2 节所介绍的内容之上，进一步说明规划器如何利用系统统计信息来估计查询各部分可能返回的行数。这是规划过程中的重要组成部分，为代价计算提供了大量原始材料。

本章的目的不是详细说明代码，而是概述其工作方式。这或许能让随后想要阅读代码的人更容易入门。

71.1. 行估计示例

下面的示例使用 PostgreSQL 回归测试数据库中的表。所示输出取自版本 8.3，更早（或更晚）的版本行为可能有所不同。另外，由于 ANALYZE 在生成统计信息时使用随机采样，每次重新执行 ANALYZE 后，结果都会略有变化。

让我们从一个很简单的查询开始：

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

关于规划器如何确定 tenk1 的基数，第 14.2 节中已经介绍过；这里为了完整起见再重复一次。页数和行数是从 pg_class 中查得的：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
----------+-----------
      358 |     10000
```

这些数字反映的是该表最近一次 VACUUM 或 ANALYZE 时的情况。随后，规划器会取得该表当前实际的页数（这是一项开销很小的操作，无需扫描全表）。如果该值与relpages 不同，就会相应地缩放reltuples，从而得到当前的行数估计。在上面的示例中，relpages 的值是最新的，因此行数估计与reltuples 相同。

接着看一个包含范围条件的示例，其条件位于WHERE 子句中：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                 QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

规划器检查 WHERE 子句中的条件，并查找操作符<的选择率函数，所在的系统目录是pg_operator。此函数保存在 oprrest 列中，本例的条目为scalarltsel。scalarltsel 函数取得 unique1 的直方图，来源为pg_statistic。手动查询时，查看更简单的pg_stats 视图更方便：

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                    histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

接下来，计算 “< 1000” 在直方图中所占的比例，这就是选择率。直方图把范围划分为等频率的桶，因此只需找到目标值所在的桶，计入该桶的部分和前面所有桶的全部。值 1000 显然在第二个桶（993-1997）中。假设每个桶内部的值呈线性分布，可以按如下方式计算选择率：

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

即一个完整的桶加上第二个桶中的线性比例，再除以桶数。用选择率乘以表的基数，即可计算估计行数。这里所用的表为tenk1：

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (rounding off)
```

接下来考虑一个在 WHERE 子句中包含等值条件的示例：

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

规划器同样会检查 WHERE 子句中的条件，并查找=对应的选择率函数，也就是eqsel。对于等值估计，直方图并无用处；相反，会使用高频值（MCV）列表来确定选择率。让我们看一下这些MCV，并同时看看后面会用到的几个额外列：

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 676
most_common_vals  | {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

因为 CRAAAA 出现在 MCV 列表中，所以选择率就是高频值频率（MCF）列表中对应的那一项：

```
selectivity = mcf[3]
            = 0.003
```

和前面一样，估计行数就是它与tenk1 基数的乘积：

```
rows = 10000 * 0.003
     = 30
```

现在考虑同一个查询，但其中的常量不在MCV 列表里：

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

这是一个很不同的问题：当该值不在 MCV 列表中时，如何估计选择率。方法是利用它不在列表中的事实，再结合所有MCV 的频率信息：

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

也就是说，先把所有MCV 的频率加总，再从 1 中减去，然后除以其他非重复值的数量。这相当于假设该列中不属于任何 MCV 的那一部分，在所有其他非重复值之间均匀分布。注意这里没有空值，因此不必考虑它们（否则还需要从分子中减去空值比例）。估计行数随后按通常方式计算：

```
rows = 10000 * 0.0014559
     = 15  (rounding off)
```

前面带有 `unique1 < 1000` 的示例，把`scalarltsel` 的实际工作方式过于简化了；现在既然已经看到 MCV 的用法，我们可以补充更多细节。那个示例在其适用范围内是正确的，因为 `unique1` 是唯一列，所以没有 MCV（显然，没有哪个值会比其他值更常见）。对于非唯一列，通常既会有直方图，也会有 MCV 列表，而且直方图不包括该列总体中由 MCV 表示的那一部分。这样做是为了得到更精确的估计。在这种情况下，`scalarltsel` 会把条件（例如 “< 1000” ）直接应用到 MCV 列表中的每个值上，并把满足条件的 MCV 频率加总起来。这就为表中属于 MCV 的那一部分给出了精确的选择率估计。随后，再像上面那样用直方图估算非 MCV 部分的选择率，最后将两者合并，得到整体选择率估计。例如，考虑

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
   Filter: (stringu1 < 'IAAAAA'::name)
```

我们已经看过 `stringu1` 的高频值信息，这里是它的直方图：

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

                              histogram_bounds
-----------------------------------------------------------------------------
 {AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAA,XLAAAA,ZZAAAA}
```

检查 MCV 列表后可知，条件`stringu1 < 'IAAAAA'` 对前 6 项成立，对后 4 项不成立，因此在由 MCV 表示的那部分总体中的选择率是：

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
            = 0.01833333
```

将所有 MCF 相加，还可得知由 MCV 表示的总体比例是 0.03033333，因此由直方图表示的比例就是 0.96966667（同样，这里没有空值，否则还得在此处将它们排除）。可以看出，值`IAAAAA` 几乎落在第三个直方图桶的末尾。规划器基于一些相当粗略的关于不同字符出现频率的假设，得到直方图所表示总体中小于`IAAAAA` 的那一部分估计为 0.298387。然后我们将 MCV 与非 MCV 总体的估计合并：

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
            = 0.01833333 + 0.298387 * 0.96966667
            = 0.307669

rows        = 10000 * 0.307669
            = 3077  (rounding off)
```

在这个特定示例中，来自 MCV 列表的修正相当小，因为该列的分布实际上相当平坦（统计信息显示这些特定值比其他值更常见，主要是抽样误差造成的）。在更典型的场景下，一些值会明显比其他值更常见，这一较为复杂的过程能够有效提高准确性，因为高频值的选择率可以精确求出。

现在来考虑一个 `WHERE` 子句中有多个条件的情形：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1 width=244)
   Recheck Cond: (unique1 < 1000)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

规划器假定这两个条件彼此独立，因此可以将各子句的选择率相乘：

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
            = 0.100697 * 0.0014559
            = 0.0001466

rows        = 10000 * 0.0001466
            = 1  (rounding off)
```

请注意，位图索引扫描估计返回的行数只反映用于该索引的条件；这一点很重要，因为它会影响后续堆获取的代价估计。

最后，来看一个涉及连接的查询：

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17 rows=50 width=244)
         Recheck Cond: (unique1 < 50)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.63 rows=50 width=0)
               Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..6.27 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

对于 tenk1，unique1 < 50 这一限制会在嵌套循环连接之前求值。处理方法与之前的范围查询示例类似。这次，值 50 落在unique1 直方图的第一个桶中：

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows        = 10000 * 0.005035
            = 50  (rounding off)
```

连接的限制条件是 t2.unique2 = t1.unique2。操作符就是熟悉的=，不过选择率函数来自 oprjoin 列，该列属于pg_operator，本例中为eqjoinsel。eqjoinsel 查询两个表的统计信息，这两个表是tenk2 和 tenk1：

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
 tenk1     |         0 |         -1 |
 tenk2     |         0 |         -1 |
```

本例中，没有MCV 信息可用于 unique2，因为所有值看起来都是唯一的。因此使用的算法仅依赖于两个关系的非重复值数量及其空值比例：

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1, 1/num_distinct2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
```

也就是说，对两个关系分别用一减去空值比例，再除以两者非重复值数量中的较大者。预计连接输出的行数，通过将两个输入的笛卡尔积基数乘以选择率来计算：

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (50 * 10000) * 0.0001
     = 50
```

如果这两列存在 MCV 列表，eqjoinsel 就会通过直接比较 MCV 列表，确定由 MCV 表示的那部分列值总体中的连接选择率。剩余部分总体的估计则沿用这里展示的相同方法。

请注意，我们把inner_cardinality 写成 10000，也就是tenk2 未经修改的大小。单看 EXPLAIN 输出，似乎连接行数估计来自 50 * 1，也就是外表行数乘以对tenk2 执行每次内表索引扫描得到的估计行数。但事实并非如此：连接关系的大小是在考虑任何具体连接计划之前估计

的。如果一切正常，这两种连接大小估算方式会得到大致相同的答案，但由于舍入误差和其他因素，它们有时会出现明显偏差。

如果想了解更多细节，表大小（在任何WHERE 子句之前）的估计是在 src/backend/optimizer/util/plancat.c 中完成的。子句选择率的一般逻辑位于 src/backend/optimizer/path/clausesel.c。按操作符区分的选择率函数大多位于 src/backend/utils/adt/selfuncs.c 中。

71.2. 多元统计信息示例

71.2.1. 函数依赖

多元相关性可以通过一个非常简单的数据集来演示 – 一个包含两列的表，两列都含有相同的值：

```
CREATE TABLE t (a INT, b INT);
INSERT INTO t SELECT i % 100, i % 100 FROM generate_series(1, 10000) s(i);
ANALYZE t;
```

如第 14.2 节所述，规划器可以利用从pg_class 得到的页数和行数来确定 t 的基数：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 't';

 relpages | reltuples
----------+-----------
       45 |     10000
```

数据分布非常简单；每一列中都只有 100 个不同的值，并且均匀分布。

下面的示例展示了对 WHERE 条件在 a 列上进行估计的结果：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1;
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..170.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: (a = 1)
   Rows Removed by Filter: 9900
```

规划器会检查该条件，并认定此子句的选择率为 1%。将该估计与实际行数相比，可以看出估计非常准确（实际上是精确的，因为这张表非常小）。把WHERE 条件改为使用 b 列，会生成相同的计划。但请看如果把同一条件同时应用到两列上，并用AND 连接，会发生什么：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

规划器分别估计每个条件的选择率，得到与上面相同的 1% 估计值。然后它假定这些条件彼此独立，于是将它们的选择率相乘，得到最终只有 0.01% 的选择率估计。这是严重的低估，因为实际匹配这些条件的行数（100）要高出两个数量级。

这个问题可以通过创建一个统计信息对象来解决，该对象会指示ANALYZE 在这两列上计算函数依赖的多元统计信息：

```
CREATE STATISTICS stts (dependencies) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

71.2.2. 多元非重复值计数

在估计多列集合的基数时，也会出现类似的问题，例如GROUP BY 子句会生成多少个组。当GROUP BY 只列出一列时，非重复值数量估计（可从 HashAggregate 节点估计返回的行数看出）非常准确：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a;
                                   QUERY PLAN
-------------------------------------------------------------------------------------------
 HashAggregate  (cost=195.00..196.00 rows=100 width=12) (actual rows=100 loops=1)
   Group Key: a
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=4) (actual rows=10000 loops=1)
```

但是如果没有多元统计信息，对于在GROUP BY 中包含两列的查询，组数估计会像下面这样差一个数量级：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                     QUERY PLAN
-----------------------------------------------------------------------------------------------
 HashAggregate  (cost=220.00..230.00 rows=1000 width=16) (actual rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000 loops=1)
```

通过重新定义统计信息对象，使其包含这两列的非重复值计数，估计值就会大幅改进：

```
DROP STATISTICS stts;
CREATE STATISTICS stts (dependencies, ndistinct) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                     QUERY PLAN
-----------------------------------------------------------------------------------------------
 HashAggregate  (cost=220.00..221.00 rows=100 width=16) (actual rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000 loops=1)
```

71.3. 规划器统计信息与安全

对表 pg_statistic 的访问仅限于超级用户，这样普通用户就无法借此了解其他用户表中的内容。有些选择率估算函数会使用用户提供的操作符（查询中出现的操作符或与之相关的操作符）来分析已存储的统计信息。例如，为了判断某个已存储的高频值是否适用，选择率估算器必须运行合适的=操作符，将查询中的常量与已存储的值进行比较。因此，pg_statistic 中的数据有可能被传递给用户定义的操作符。精心构造的操作符可以故意泄露传给它的操作数（例如记录到日志中，或写入另一张表），也可能无意中通过在错误消息中显示其值而泄露它们；无论哪种情况，都可能把pg_statistic 中的数据暴露给本不应看到这些数据的用户。

为了防止这种情况，以下规则适用于所有内置选择率估算函数。在规划查询时，若要使用已存储的统计信息，当前用户必须对该表或相关列具有SELECT 权限，或者所用操作符必须是

LEAKPROOF（更准确地说，是该操作符所基于的函数必须如此）。否则，选择率估算器会像不存在可用统计信息一样工作，规划器则会按默认或后备假设继续。

如果用户对表或列不具备所需权限，那么在许多情况下，查询最终都会收到一个权限被拒绝的错误，因此这一机制在实际中是不可见的。但如果用户是通过安全屏障视图读取数据，规划器可能会想要检查一个用户本来无权访问的底层表的统计信息。在这种情况下，操作符应当是 leak-proof，否则统计信息不会被使用。除了计划可能不够理想之外，对此没有直接反馈。如果怀疑发生了这种情况，可以尝试让权限更高的用户运行该查询，看看是否会得到不同的计划。

此限制只适用于规划器需要对来自 pg_statistic 的一个或多个值执行用户定义操作符的情况。因此，不论访问权限如何，规划器仍可使用通用统计信息，例如空值比例或列中非重复值的数量。

第三方扩展中的选择率估算函数如果可能会用用户定义的操作符处理统计信息，也应遵循同样的安全规则。可参考 PostgreSQL 源代码。

部分 VIII. 附录

目录

A. PostgreSQL 错误代码 2304
B. 日期/时间支持 2312
B.1. 日期/时间输入解释 2312
B.2. 处理无效或有歧义的时间戳 2313
B.3. 日期/时间关键字 2314
B.4. 日期/时间配置文件 2315
B.5. POSIX 时区规范 2316
B.6. 历法的历史 2318
B.7. 儒略日 2319
C. SQL 关键字 2320
D. SQL 符合性 2342
D.1. 支持的特性 2343
D.2. 不支持的特性 2355
D.3. XML 的限制以及对 SQL/XML 的符合性 2365
D.3.1. 查询受限于 XPath 1.0 2365
D.3.2. 实现带来的附带限制 2367
E. 发布说明 2368
E.1. 发布版本 11.22 2368
E.1.1. 迁移到版本 11.22 2368
E.1.2. 变更 2368
E.2. 发布版本 11.21 2371
E.2.1. 迁移到版本 11.21 2371
E.2.2. 变更 2371
E.3. 发布版本 11.20 2374
E.3.1. 迁移到版本 11.20 2374
E.3.2. 变更 2374
E.4. 发布版本 11.19 2378
E.4.1. 迁移到版本 11.19 2378
E.4.2. 变更 2378
E.5. 发布版本 11.18 2381
E.5.1. 迁移到版本 11.18 2381
E.5.2. 变更 2381
E.6. 发布版本 11.17 2384
E.6.1. 迁移到版本 11.17 2384
E.6.2. 变更 2384
E.7. 发布版本 11.16 2388
E.7.1. 迁移到版本 11.16 2388
E.7.2. 变更 2388
E.8. 发布版本 11.15 2391
E.8.1. 迁移到版本 11.15 2391
E.8.2. 变更 2391
E.9. 发布版本 11.14 2393
E.9.1. 迁移到版本 11.14 2393
E.9.2. 变更 2394
E.10. 发布版本 11.13 2399
E.10.1. 迁移到版本 11.13 2399
E.10.2. 变更 2399
E.11. 发布版本 11.12 2404
E.11.1. 迁移到版本 11.12 2404
E.11.2. 变更 2404
E.12. 发布版本 11.11 2407

E.12.1. 迁移到版本 11.11 2407
E.12.2. 变更 2407
E.13. 发布版本 11.10 2412
E.13.1. 迁移到版本 11.10 2412
E.13.2. 变更 2412
E.14. 发布版本 11.9 2415
E.14.1. 迁移到版本 11.9 2415
E.14.2. 变更 2415
E.15. 发布版本 11.8 2419
E.15.1. 迁移到版本 11.8 2420
E.15.2. 变更 2420
E.16. 发布版本 11.7 2423
E.16.1. 迁移到版本 11.7 2423
E.16.2. 变更 2424
E.17. 发布版本 11.6 2428
E.17.1. 迁移到版本 11.6 2428
E.17.2. 变更 2428
E.18. 发布版本 11.5 2433
E.18.1. 迁移到版本 11.5 2433
E.18.2. 变更 2433
E.19. 发布版本 11.4 2436
E.19.1. 迁移到版本 11.4 2436
E.19.2. 变更 2436
E.20. 发布版本 11.3 2439
E.20.1. 迁移到版本 11.3 2439
E.20.2. 变更 2439
E.21. 发布版本 11.2 2444
E.21.1. 迁移到版本 11.2 2444
E.21.2. 变更 2444
E.22. 发布版本 11.1 2449
E.22.1. 迁移到版本 11.1 2449
E.22.2. 变更 2449
E.23. 发布版本 11 2451
E.23.1. 概述 2451
E.23.2. 迁移到版本 11 2451
E.23.3. 变更 2454
E.23.4. 致谢 2464
E.24. 先前版本 2470
F. 额外提供的模块 2471
F.1. adminpack 2473
F.2. amcheck 2474
F.2.1. 函数 2474
F.2.2. 可选的 *heapallindexed* 验证 2475
F.2.3. 有效使用 amcheck 2476
F.2.4. 修复损坏 2476
F.3. auth_delay 2477
F.3.1. 配置参数 2477
F.3.2. 作者 2477
F.4. auto_explain 2478
F.4.1. 配置参数 2478
F.4.2. 示例 2479
F.4.3. 作者 2480
F.5. bloom 2481

F.5.1. 参数 2481
F.5.2. 示例 2481
F.5.3. 操作符类接口 2484
F.5.4. 限制 2484
F.5.5. 作者 2484
F.6. btree_gin 2485
F.6.1. 用法示例 2485
F.6.2. 作者 2485
F.7. btree_gist 2486
F.7.1. 用法示例 2486
F.7.2. 作者 2486
F.8. citext 2488
F.8.1. 原理 2488
F.8.2. 如何使用 2488
F.8.3. 字符串比较行为 2488
F.8.4. 限制 2489
F.8.5. 作者 2490
F.9. cube 2491
F.9.1. 语法 2491
F.9.2. 精度 2491
F.9.3. 用法 2491
F.9.4. 默认规则 2494
F.9.5. 注意 2495
F.9.6. 致谢 2495
F.10. dblink 2496
F.11. dict_int 2527
F.11.1. 配置 2527
F.11.2. 用法 2527
F.12. dict_xsyn 2528
F.12.1. 配置 2528
F.12.2. 用法 2528
F.13. earthdistance 2530
F.13.1. 基于立方体的地球距离 2530
F.13.2. 基于点的地球距离 2531
F.14. file_fdw 2532
F.15. fuzzystrmatch 2535
F.15.1. Soundex 2535
F.15.2. Levenshtein 2535
F.15.3. Metaphone 2536
F.15.4. Double Metaphone 2537
F.16. hstore 2538
F.16.1. `hstore` 外部表示 2538
F.16.2. `hstore` 操作符和函数 2538
F.16.3. 索引 2542
F.16.4. 示例 2542
F.16.5. 统计信息 2543
F.16.6. 兼容性 2544
F.16.7. 转换 2544
F.16.8. 作者 2545
F.17. intagg 2546
F.17.1. 函数 2546
F.17.2. 使用示例 2546
F.18. intarray 2548

F.18.1. intarray 函数和操作符 2548
F.18.2. 索引支持 2549
F.18.3. 示例 2550
F.18.4. 基准测试 2550
F.18.5. 作者 2550
F.19. isn 2551
F.19.1. 数据类型 2551
F.19.2. 类型转换 2552
F.19.3. 函数和操作符 2552
F.19.4. 示例 2553
F.19.5. 参考文献 2553
F.19.6. 作者 2554
F.20. lo 2555
F.20.1. 原理 2555
F.20.2. 如何使用 2555
F.20.3. 限制 2555
F.20.4. 作者 2556
F.21. ltree 2557
F.21.1. 定义 2557
F.21.2. 操作符和函数 2558
F.21.3. 索引 2560
F.21.4. 示例 2561
F.21.5. 转换 2563
F.21.6. 作者 2563
F.22. pageinspect 2564
F.22.1. 通用函数 2564
F.22.2. 堆函数 2565
F.22.3. B-树函数 2566
F.22.4. BRIN 函数 2568
F.22.5. GIN 函数 2569
F.22.6. Hash 函数 2570
F.23. passwordcheck 2572
F.24. pg_buffercache 2573
F.24.1. pg_buffercache 视图 2573
F.24.2. 示例输出 2574
F.24.3. 作者 2574
F.25. pgcrypto 2575
F.25.1. 通用 hash 函数 2575
F.25.2. 密码 hash 函数 2575
F.25.3. PGP 加密函数 2577
F.25.4. 原始加密函数 2582
F.25.5. 随机数据函数 2583
F.25.6. 注意事项 2584
F.25.7. 作者 2585
F.26. pg_freespacemap 2587
F.26.1. 函数 2587
F.26.2. 示例输出 2588
F.26.3. 作者 2588
F.27. pg_prewarm 2589
F.27.1. 函数 2589
F.27.2. 配置参数 2589
F.27.3. 作者 2590
F.28. pgrowlocks 2591

F.28.1. 概述 2591
F.28.2. 示例输出 2592
F.28.3. 作者 2592
F.29. pg_stat_statements 2593
F.29.1. pg_stat_statements 视图 2593
F.29.2. 函数 2595
F.29.3. 配置参数 2595
F.29.4. 示例输出 2596
F.29.5. 作者 2597
F.30. pgstattuple 2598
F.30.1. 函数 2598
F.30.2. 作者 2602
F.31. pg_trgm 2603
F.31.1. 三字符组（Trigram 或 Trigraph）概念 2603
F.31.2. 函数和操作符 2603
F.31.3. GUC 参数 2605
F.31.4. 索引支持 2605
F.31.5. 全文检索集成 2607
F.31.6. 参考 2607
F.31.7. 作者 2608
F.32. pg_visibility 2609
F.32.1. 函数 2609
F.32.2. 作者 2610
F.33. postgres_fdw 2611
F.33.1. postgres_fdw 的 FDW 选项 2611
F.33.2. 连接管理 2614
F.33.3. 事务管理 2614
F.33.4. 远程查询优化 2614
F.33.5. 远程查询执行环境 2615
F.33.6. 跨版本兼容性 2615
F.33.7. 示例 2615
F.33.8. 作者 2616
F.34. seg 2617
F.34.1. 原理 2617
F.34.2. 语法 2617
F.34.3. 精度 2618
F.34.4. 用法 2618
F.34.5. 注意 2619
F.34.6. 致谢 2619
F.35. sepgsql 2620
F.35.1. 概述 2620
F.35.2. 安装 2620
F.35.3. 回归测试 2621
F.35.4. GUC 参数 2622
F.35.5. 特性 2623
F.35.6. sepgsql 函数 2626
F.35.7. 限制 2626
F.35.8. 外部资源 2627
F.35.9. 作者 2627
F.36. spi 2628
F.36.1. refint – 用于实现引用完整性的函数 2628
F.36.2. timetravel – 实现时间旅行的函数 2628
F.36.3. autoinc – 用于字段自动递增的函数 2629

F.36.4. insert_username — 用于跟踪谁修改了表的函数 2629
F.36.5. moddatetime — 用于跟踪最后修改时间的函数 2629
F.37. sslinfo 2630
F.37.1. 提供的函数 2630
F.37.2. 作者 2631
F.38. tablefunc 2632
F.38.1. 提供的函数 2632
F.38.2. 作者 2641
F.39. tcn 2642
F.40. test_decoding 2643
F.41. tsm_system_rows 2644
F.41.1. 示例 2644
F.42. tsm_system_time 2645
F.42.1. 示例 2645
F.43. unaccent 2646
F.43.1. 配置 2646
F.43.2. 用法 2647
F.43.3. 函数 2647
F.44. uuid-ossp 2649
F.44.1. `uuid-ossp` 函数 2649
F.44.2. 构建 `uuid-ossp` 2650
F.44.3. 作者 2650
F.45. xml2 2651
F.45.1. 弃用说明 2651
F.45.2. 函数说明 2651
F.45.3. `xpath_table` 2652
F.45.4. XSLT 函数 2654
F.45.5. 作者 2654
G. 额外提供的程序 2655
G.1. 客户端应用 2655
G.2. 服务器应用 2662
H. 外部项目 2667
H.1. 客户端接口 2667
H.2. 管理工具 2667
H.3. 过程语言 2667
H.4. 扩展 2667
I. 源代码仓库 2668
I.1. 通过 Git 获取源代码 2668
J. 文档 2669
J.1. DocBook 2669
J.2. 工具集 2669
J.2.1. 在 Fedora、RHEL 及其衍生版上安装 2670
J.2.2. 在 FreeBSD 上安装 2670
J.2.3. Debian 软件包 2670
J.2.4. macOS 2670
J.2.5. 通过 `configure` 检测 2671
J.3. 构建文档 2671
J.3.1. HTML 2671
J.3.2. 手册页 2672
J.3.3. PDF 2672
J.3.4. 纯文本文件 2672
J.3.5. 语法检查 2673
J.4. 文档编写 2673

J.4.1. Emacs 2673
J.5. 风格指南 2673
J.5.1. 参考页 2673
K. 缩略词 2675
L. 过时或重命名的功能 2682
L.1. pg_xlogdump 重命名为 pg_waldump 2682
L.2. pg_resetxlog 重命名为 pg_resetwal 2682
L.3. pg_receivexlog 重命名为 pg_receivewal 2682

附录 A. PostgreSQL 错误代码

PostgreSQL 服务器发出的所有消息都被分配了五字符错误代码，遵循 SQL 标准对“SQLSTATE”代码的约定。需要知道发生了哪一种错误条件的应用，通常应测试错误代码，而不是查看错误消息的文本内容。这些错误代码在不同 PostgreSQL 版本之间更不容易发生变化，也不会因错误消息的本地化而改变。请注意，PostgreSQL 产生的错误代码中，只有一部分由 SQL 标准定义；还有一些用于标准未定义情况的附加错误代码，是新设的或从其他数据库借用而来的。

根据标准，错误代码的前两个字符表示错误类别，后三个字符表示该类别中的特定条件。因此，即使应用无法识别某个具体错误代码，仍可能从错误类别推断出应采取的操作。

表 A.1 列出了 PostgreSQL 11.22 中定义的所有错误代码。（其中有些当前实际上并未使用，不过它们由 SQL 标准定义。）其中也显示了错误类别。对于每个错误类别，都有一个后三个字符为 `000` 的“标准”错误代码。该代码只用于那些属于该类别但没有分配更具体代码的错误情况。

“条件名称”列中显示的符号是在 PL/pgSQL 中使用的条件名称。条件名称既可以写成大写，也可以写成小写。（注意，PL/pgSQL 不识别警告条件名称，也就是说不识别 00、01 和 02 这几个类别。）

对于某些类型的错误，服务器会报告与该错误关联的数据库对象名称（例如表、表列、数据类型或约束）；例如，会报告导致 `unique_violation` 错误的唯一约束名称。这些名称会在错误报告消息的独立字段中提供，因此应用不必尝试从可能已经本地化的人类可读消息文本中提取它们。截至 PostgreSQL 9.3，只有 SQLSTATE 类 23（完整性约束违反）中的错误对此特性实现了完整覆盖，但未来很可能会扩展到更多类别。

表 A.1. PostgreSQL 错误代码

错误代码	条件名称
类 00 – 成功完成	
00000	successful_completion
类 01 – 警告	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
类 02 – 无数据（按 SQL 标准，这也是一个警告类别）	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
类 03 – SQL 语句尚未完成	
03000	sql_statement_not_yet_complete
类 08 – 连接异常	
08000	connection_exception

错误代码	条件名称
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
类 09 — 触发动作异常	
09000	triggered_action_exception
类 0A — 不支持的特性	
0A000	feature_not_supported
类 0B — 无效的事务启动	
0B000	invalid_transaction_initiation
类 0F — 定位器异常	
0F000	locator_exception
0F001	invalid_locator_specification
类 0L — 无效的授权者	
0L000	invalid_grantor
0LP01	invalid_grant_operation
类 0P — 无效的角色指定	
0P000	invalid_role_specification
类 0Z — 诊断异常	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
类 20 — 未找到 CASE 分支	
20000	case_not_found
类 21 — 基数违反	
21000	cardinality_violation
类 22 — 数据异常	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm

错误代码	条件名称
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
22013	invalid_preceding_or_following_size
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
2200H	sequence_generator_limit_exceeded
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document

错误代码	条件名称
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
类 23 — 完整性约束违反	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
类 24 — 无效的游标状态	
24000	invalid_cursor_state
类 25 — 无效的事务状态	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
类 26 — 无效的 SQL 语句名称	
26000	invalid_sql_statement_name
类 27 — 触发的数据更改违反规则	
27000	triggered_data_change_violation
类 28 — 无效的授权指定	
28000	invalid_authorization_specification
28P01	invalid_password
类 2B — 依赖的权限描述符仍然存在	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist

错误代码	条件名称
类 2D — 无效的事务终止	
2D000	invalid_transaction_termination
类 2F — SQL 例程异常	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
类 34 — 无效的游标名称	
34000	invalid_cursor_name
类 38 — 外部例程异常	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
类 39 — 外部例程调用异常	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
类 3B — 保存点异常	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
类 3D — 无效的目录名称	
3D000	invalid_catalog_name
类 3F — 无效的模式名称	
3F000	invalid_schema_name
类 40 — 事务回滚	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
类 42 — 语法错误或访问规则违反	
42000	syntax_error_or_access_rule_violation
42601	syntax_error

错误代码	条件名称
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition

错误代码	条件名称
42P16	invalid_table_definition
42P17	invalid_object_definition
类 44 – WITH CHECK OPTION 违反	
44000	with_check_option_violation
类 53 – 资源不足	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
类 54 – 超出程序限制	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
类 55 – 对象未处于所需状态	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
类 57 – 操作员干预	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
类 58 – 系统错误（PostgreSQL自身以外的错误）	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
类 72 – 快照失败	
72000	snapshot_too_old
类 F0 – 配置文件错误	
F0000	config_file_error
F0001	lock_file_exists
类 HV – 外部数据包装器错误（SQL/MED）	
HV000	fdw_error
HV005	fdw_column_name_not_found

错误代码	条件名称
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
类 P0 — PL/pgSQL 错误	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
类 XX — 内部错误	
XX000	internal_error
XX001	data_corrupted
XX002	index_corrupted

附录 B. 日期/时间支持

PostgreSQL 对所有日期/时间输入都使用一个内部的启发式解析器。日期和时间以字符串形式输入，并被拆分为不同的字段，同时初步判断每个字段中可能包含哪类信息。每个字段都会被解释，并被赋予数值、忽略或拒绝。解析器内部为所有文本字段都维护了查找表，包括月份、星期几以及时区。

本附录包含关于这些查找表内容的信息，并描述了解析器解码日期和时间时所使用的步骤。

B.1. 日期/时间输入解释

日期/时间输入字符串按照下面的过程进行解码。

1. 将输入字符串拆分为词元，并把每个词元归类为字符串、时间、时区或数字。

    a. 如果数字词元包含冒号（`:`），则它是一个时间字符串。把其后的所有数字和冒号都包含进来。

    b. 如果数字词元包含连字符（`-`）、斜杠（`/`）或两个以上的点（`.`），则它是一个日期字符串，并且可能带有文本形式的月份。如果已经见过日期词元，则它会被解释为时区名称（例如 `America/New_York`）。

    c. 如果词元仅由数字组成，那么它要么是单个字段，要么是 ISO 8601 的拼接式日期（例如，`19990113` 表示 1999 年 1 月 13 日）或时间（例如，`141516` 表示 14:15:16）。

    d. 如果词元以加号（`+`）或减号（`-`）开头，那么它要么是数字时区，要么是特殊字段。

2. 如果词元是一个纯字母字符串，则将其与可能的字符串匹配：

    a. 先看该词元是否匹配任何已知的时区缩写。这些缩写由第 B.4 节中描述的配置文件提供。

    b. 如果没有找到，则搜索内部表，将该词元识别为特殊字符串（例如 `today`）、星期名（例如 `Thursday`）、月份名（例如 `January`）或噪声词（例如 `at`、`on`）。

    c. 如果仍然没有找到，则抛出错误。

3. 当词元是一个数字或数字字段时：

    a. 如果有八位或六位数字，并且之前还没有读取到其他日期字段，则将其解释为“拼接式日期”（例如 `19990118` 或 `990118`）。其解释方式分别为 `YYYYMMDD` 或 `YYMMDD`。

    b. 如果词元是三位数字，并且已经读取到了年份，则将其解释为一年中的第几天。

    c. 如果是四位或六位数字，并且已经读取到了年份，则将其解释为时间（`HHMM` 或 `HHMMSS`）。

    d. 如果有三位或更多位数字，并且尚未找到任何日期字段，则将其解释为年份（这会强制其余日期字段按 yy-mm-dd 的顺序解释）。

    e. 否则，假定日期字段顺序遵循 `DateStyle` 设置：mm-dd-yy、dd-mm-yy 或 yy-mm-dd。如果发现月份字段或日字段超出范围，则抛出错误。

4. 如果指定了 BC，则将年份取反并加一后再用于内部存储。（格里高利历中没有 0 年，因此从数值上说，公元前 1 年会变成年 0。）

5. 如果没有指定 BC，并且年份字段的长度为两位，则将年份调整为四位。如果该字段小于 70，则加上 2000，否则加上 1900。

> **提示**
>
> 格里高利历公元 1-99 年可以使用带前导零的四位数字输入（例如，`0099` 表示公元 99 年）。

B.2. 处理无效或有歧义的时间戳

通常，如果一个日期/时间字符串在语法上有效，但包含超出范围的字段值，就会抛出错误。例如，指定 2 月 31 日的输入会被拒绝。

在夏令时转换期间，看似有效的时间戳字符串有可能表示一个不存在的时间戳，或者表示一个有歧义的时间戳。这类情况不会被拒绝；系统通过确定应采用哪个 UTC 偏移来消除歧义。例如，假设TimeZone 参数被设置为 `America/New_York`，请看：

```
=> SELECT '2018-03-11 02:30'::timestamptz;
      timestamptz
------------------------
 2018-03-11 03:30:00-04
(1 row)
```

因为那一天在该时区是春季拨快的转换日，所以并不存在民用时间 2:30AM；时钟从 2AM EST 直接跳到 3AM EDT。PostgreSQL 将给定时间按标准时间（UTC-5）来解释，因此显示为 3:30AM EDT（UTC-4）。

相反，考虑秋季回拨转换期间的行为：

```
=> SELECT '2018-11-04 01:30'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-05
(1 row)
```

在那一天，1:30AM 有两种可能的解释：先有一次 1:30AM EDT；一小时后又有一次 1:30AM EST，其间时钟已从 2AM EDT 回拨到 1AM EST。再次，PostgreSQL 将给定时间按标准时间（UTC-5）来解释。我们可以通过显式指定夏令时来强制采用另一种解释：

```
=> SELECT '2018-11-04 01:30 EDT'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-04
(1 row)
```

在这类情况下适用的精确规则是：如果一个无效时间戳看起来落在夏令时前向跳转转换之内，则为它分配该时区在转换前一刻采用的 UTC 偏移；如果一个有歧义的时间戳可能落在回拨转换前后任一侧，则为它分配该时区在转换后一刻采用的 UTC 偏移。在大多数时区，这等价于说“有疑义时，优先按标准时间解释”。

在任何情况下，都可以显式指定与时间戳相关联的 UTC 偏移，方法是使用数字形式的 UTC 偏移，或者使用对应于固定 UTC 偏移的时区缩写。上述规则只在需要为 UTC 偏移会变化的时区推断 UTC 偏移时才适用。

B.3. 日期/时间关键字

表 B.1 展示了哪些词元会被识别为月份名称。

表 B.1. 月份名称

月份	缩写
January	Jan
February	Feb
March	Mar
April	Apr
May	
June	Jun
July	Jul
August	Aug
September	Sep, Sept
October	Oct
November	Nov
December	Dec

表 B.2 展示了哪些词元会被识别为星期名称。

表 B.2. 星期名称

星期	缩写
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

表 B.3 展示了用于各种修饰用途的词元。

表 B.3. 日期/时间字段修饰符

标识符	说明
`AM`	时间早于 12:00
`AT`	忽略
`JULIAN`, `JD`, `J`	下一个字段是儒略日
`ON`	忽略
`PM`	时间为 12:00 或之后

标识符	说明
T	下一个字段是时间

B.4. 日期/时间配置文件

由于时区缩写缺乏良好的标准化，PostgreSQL 提供了一种机制来自定义服务器可接受的缩写集合。timezone_abbreviations 运行时参数决定当前活动的缩写集合。虽然这个参数可以由任何数据库用户修改，但它的可能取值受数据库管理员控制 – 它们实际上是安装目录中 .../share/timezonesets/ 里保存的配置文件名称。通过向该目录添加或修改文件，管理员可以为时区缩写设定本地策略。

如果文件名完全由字母组成，那么 timezone_abbreviations 可以被设置为 .../share/timezonesets/ 中找到的任意文件名。（禁止在 timezone_abbreviations 中使用非字母字符，不仅可以防止读取目标目录之外的文件，也能防止读取编辑器备份文件和其他无关文件。）

时区缩写文件可以包含空行和以 # 开头的注释。非注释行必须具有下列格式之一：

```
zone_abbreviation offset
zone_abbreviation offset D
zone_abbreviation time_zone_name
@INCLUDE file_name
@OVERRIDE
```

zone_abbreviation 就是正在定义的缩写。*offset* 是一个整数，给出相对 UTC 的秒数偏移；相对格林尼治以东为正，以西为负。例如，-18000 表示位于格林尼治以西五小时，也就是北美东海岸标准时间。D 表示该时区名称代表的是本地夏令时，而不是标准时间。

另一种写法是给出 *time_zone_name*，引用 IANA 时区数据库中定义的时区名称。这时会查阅该时区的定义，以判断该缩写是否正在或曾经在该时区使用过；如果是，就采用适当的含义，也就是：在所求时间戳对应时刻实际生效的含义；如果在那个时刻它并不是当时正在使用的含义，则采用紧接其前一个生效的含义；如果它只在那个时刻之后才被使用，则采用最早的含义。对于那些历史上含义曾发生变化的缩写，这种行为至关重要。也允许按一个并不出现该缩写的时区名称来定义某个缩写；这样使用该缩写就等同于直接写出该时区名称。

> **提示**
>
> 如果某个缩写相对 UTC 的偏移从未改变过，那么定义它时最好使用简单的整数 *offset*，因为这类缩写的处理成本远低于那些需要查阅时区定义的缩写。

@INCLUDE 语法允许包含 .../share/timezonesets/ 目录中的另一个文件。包含可以嵌套，但嵌套深度有限。

@OVERRIDE 语法表示文件中的后续条目可以覆盖先前条目（通常是从被包含文件中得到的条目）。如果没有它，同一个时区缩写的冲突定义会被视为错误。

在未经修改的安装中，Default 文件包含世界上大多数地区中所有互不冲突的时区缩写。另外还为澳大利亚和印度这两个地区提供了 Australia 和 India 文件：这些文件会先包含 Default 文件，然后按需添加或修改缩写。

作为参考，标准安装还包含 `Africa.txt`、`America.txt` 等文件，其中包含 IANA 时区数据库中已知正在使用的每个时区缩写的信息。可以按需把这些文件中找到的时区名称定义复制并粘贴到自定义配置文件中。注意，这些文件不能直接作为 `timezone_abbreviations` 设置来引用，因为它们的名称中带有点号。

> **注意**
>
> 如果在读取时区缩写集时发生错误，则不会应用新值，而会保留旧集合。如果该错误发生在数据库启动期间，则数据库启动会失败。

> **小心**
>
> 配置文件中定义的时区缩写会覆盖 PostgreSQL 内置的非时区含义。例如，`Australia` 配置文件定义了 `SAT`（南澳大利亚标准时间）。当该文件处于活动状态时，`SAT` 将不会被识别为 Saturday 的缩写。

> **小心**
>
> 如果你修改了 `.../share/timezonesets/` 中的文件，就需要自行做好备份 – 普通的数据库转储不会包含这个目录。

B.5. POSIX 时区规范

PostgreSQL 可以接受按照 POSIX 标准为 `TZ` 环境变量规定的规则编写的时区规范。POSIX 时区规范不足以处理真实世界时区历史的复杂性，但有时仍有使用它们的理由。

POSIX 时区规范的形式为

```
STD offset [ DST [ dstoffset ] [ , rule ] ]
```

（为便于阅读，这里把字段之间的空格写了出来，但实际使用时不应包含空格。）各字段含义如下：

- *`STD`* 是标准时间所使用的时区缩写。
- *`offset`* 是该时区标准时间相对 UTC 的偏移。
- *`DST`* 是夏令时所使用的时区缩写。如果省略这个字段以及其后的字段，则该时区使用固定的 UTC 偏移，没有夏令时规则。
- *`dstoffset`* 是夏令时相对 UTC 的偏移。这个字段通常会被省略，因为它默认比标准时间的 *`offset`* 少一小时，而这通常正是正确的设置。
- *`rule`* 定义夏令时何时生效的规则，如下所述。

在这种语法中，时区缩写可以是由字母组成的字符串，如 `EST`；也可以是用尖括号括起来的任意字符串，如 `<UTC-05>`。注意，这里给出的时区缩写只用于输出，而且即便如此，也只在某些时间戳输出格式中使用。时间戳输入中可识别的时区缩写按第 B.4 节中说明的方式确定。

偏移字段指定与 UTC 的时差，可以写成小时，并可选地带分钟和秒。其格式为 *hh*[:*mm*[:*ss*]]，并可选带前导符号（+ 或 -）。正号用于格林尼治以西的时区。（注意，这与 PostgreSQL 其他地方使用的 ISO-8601 符号约定正好相反。）*hh* 可以是一位或两位数字；*mm* 和 *ss*（如果使用）必须是两位数字。

夏令时转换 *rule* 的格式为

dstdate [/ *dsttime*] , *stddate* [/ *stdtime*]

（同前，实际使用时不应包含空格。）*dstdate* 和 *dsttime* 字段定义夏令时何时开始，而 *stddate* 和 *stdtime* 定义标准时间何时开始。（在某些情况下，尤其是在赤道以南的时区中，前者在一年中可能晚于后者。）日期字段具有下列格式之一：

n

一个普通整数表示一年中的第几天，从 0 开始计，到 364 为止；闰年则到 365 为止。

J*n*

在这种形式中，*n* 从 1 计到 365，即使存在 2 月 29 日也不计入。（因此，发生在 2 月 29 日的转换无法用这种方式指定。不过，2 月之后的日期无论是否闰年，编号都相同，所以对于固定日期上的转换，这种形式通常比普通整数形式更有用。）

M*m*.*n*.*d*

这种形式指定一个总是发生在同一个月份、同一个星期几的转换。*m* 表示月份，范围是 1 到 12。*n* 指定由 *d* 标识的星期几在该月中第 *n* 次出现。*n* 是 1 到 4 之间的数字，或者是 5，表示该月中该星期几最后一次出现（可能是第四次，也可能是第五次）。*d* 是 0 到 6 之间的数字，其中 0 表示星期日。例如，`M3.2.0` 表示“3 月的第二个星期日”。

> **注意**
>
> `M` 格式足以描述许多常见的夏令时转换规则。但请注意，这些变体都无法处理夏令时法规的变更，因此在实践中，要正确解释过去的时间戳，就需要依赖命名时区中保存的历史数据（位于 IANA 时区数据库中）。

转换规则中的时间字段格式与前面描述的偏移字段相同，只是它们不能包含符号。它们定义了本地当前时间切换到另一种时间的时刻。如果省略，则默认值为 `02:00:00`。

如果给出了夏令时缩写，却省略了转换 *rule* 字段，PostgreSQL 会尝试通过查阅 IANA 时区数据库中的 `posixrules` 文件来确定转换时间。该文件的格式与完整时区条目相同，但这里只使用其转换时间规则，不使用其 UTC 偏移。通常，该文件与 `US/Eastern` 文件内容相同，因此 POSIX 风格的时区说明会遵循美国的夏令时规则。如有需要，可以替换 `posixrules` 文件来调整此行为。

> **注意**
>
> IANA 已弃用查阅 `posixrules` 文件的功能，它可能在未来被移除。此功能有一个缺陷：无法将夏令时规则应用于 2038 年之后的日期；在该功能消失前，这个缺陷也不太可能得到修复。

如果 `posixrules` 文件不存在，则回退为使用规则 `M3.2.0,M11.1.0`。这对应美国截至 2020 年的做法：在三月的第二个星期日向前调时钟，在十一月的第一个星期日向后调时钟，两次转换都发生在当时当地时间凌晨 2 点。

例如，`CET-1CEST,M3.5.0,M10.5.0/3` 描述了巴黎当前（截至 2020 年）的计时做法。这个规范表明标准时间的缩写是 `CET`，并且比 UTC 快 1 小时（即位于其东侧）；夏令时的缩写是 `CEST`，并且隐含地比 UTC 快 2 小时；夏令时从 3 月最后一个星期日的 2AM CET 开始，到 10 月最后一个星期日的 3AM CEST 结束。

四个时区名 `EST5EDT`、`CST6CDT`、`MST7MDT` 和 `PST8PDT` 看起来像 POSIX 时区说明，但实际上会作为命名时区处理，因为 IANA 时区数据库中出于历史原因存在这些名称的文件。这意味着，这些时区名能够给出正确的美国历史夏令时转换，即使普通 POSIX 说明因缺少合适的 `posixrules` 文件而无法做到。

需要注意的是，POSIX 风格时区规范很容易拼错，因为系统不会检查时区缩写是否合理。例如，`SET TIMEZONE TO FOOBAR0` 也能工作，从而使系统实际上使用了一个相当古怪的 UTC 缩写。

B.6. 历法的历史

SQL 标准指出，“在‘日期时间字面量’的定义中，‘日期时间值’受格里高利历中日期和时间自然规则的约束”。PostgreSQL 遵循 SQL 标准的做法，即便对于该历法尚未实际使用之前的年份，也完全按照格里高利历计算日期。这一规则称为外推格里高利历。

儒略历由 Julius Caesar 于公元前 45 年引入。在 1582 年各国开始改用格里高利历之前，它在西方世界一直被广泛使用。在儒略历中，回归年近似为 365 1/4 天 = 365.25 天。这会在大约 128 年里累计 1 天的误差。

不断累积的历法误差促使教皇格里高利十三世按照特伦托会议的指示改革历法。在格里高利历中，回归年近似为 365 + 97 / 400 天 = 365.2425 天。因此，大约要经过 3300 年，回归年与格里高利历之间才会累积出 1 天的偏差。

365+97/400 这一近似值是通过每 400 年设置 97 个闰年来实现的，具体规则如下：

每个能被 4 整除的年份都是闰年。
不过，每个能被 100 整除的年份都不是闰年。
但是，每个能被 400 整除的年份仍然是闰年。

因此，1700、1800、1900、2100 和 2200 都不是闰年，但 1600、2000 和 2400 是闰年。相比之下，较早的儒略历规定所有能被 4 整除的年份都是闰年。

1582 年 2 月发布的教皇诏书规定，应从 1582 年 10 月删去 10 天，使 10 月 15 日紧接在 10 月 4 日之后。这一改革在意大利、波兰、葡萄牙和西班牙得到执行。其他天主教国家不久后也跟进，但新教国家不愿改变，而希腊正教国家直到 20 世纪初才改用格里高利历。大不列颠及其属地（包括今天的美国）在 1752 年实施了这项改革。因此，1752 年 9 月 2 日之后紧接着就是 1752 年 9 月 14 日。这就是为什么提供 `cal` 程序的 Unix 系统会产生如下输出：

```
$ cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

不过，当然，这个历法只对大不列颠及其属地有效，对其他地方并不适用。由于尝试跟踪不同地方在不同时间实际采用的历法既困难又容易引起混淆，PostgreSQL 并不这样做，而是对所有日期都遵循格里高利历规则，尽管这种做法在历史上并不准确。

世界上不同地区发展出了不同的历法，其中许多都早于格里高利体系。例如，中国历法的起源可以追溯到公元前 14 世纪。传说黄帝在公元前 2637 年发明了这种历法。中华人民共和国在民用方面使用格里高利历。中国历法则用于确定节日。

B.7. 儒略日

儒略日系统是一种为天数编号的方法。它与儒略历并无关系，只是名称容易让人误以为二者有关。儒略日系统由法国学者 Joseph Justus Scaliger（1540-1609）发明，其名称大概取自他的父亲、意大利学者 Julius Caesar Scaliger（1484-1558）。

在儒略日系统中，每一天都有一个连续编号，从 JD 0 开始（有时就称为儒略日）。JD 0 在儒略历中对应公元前 4713 年 1 月 1 日，在格里高利历中对应公元前 4714 年 11 月 24 日。天文学家最常使用儒略日来标记其夜间观测，因此一个日期是从一个 UTC 正午到下一个 UTC 正午，而不是从午夜到午夜：JD 0 指的是从公元前 4714 年 11 月 24 日 UTC 正午到公元前 4714 年 11 月 25 日 UTC 正午这 24 小时。

虽然 PostgreSQL 支持在日期输入和输出中使用儒略日记法（并且也会在某些内部日期/时间计算中使用儒略日），但它并不遵循日期从正午算到正午这一约定。PostgreSQL 将儒略日视为从本地午夜到本地午夜，与普通日期相同。

不过，这一定义确实提供了在需要时获得天文学定义的方法：在时区 `UTC+12` 中进行运算。例如，

```
=> SELECT extract(julian from '2021-06-23 7:00:00-04'::timestamptz at time zone 'UTC+12');
     date_part
--------------------
 2459388.9583333335
(1 row)
=> SELECT extract(julian from '2021-06-23 8:00:00-04'::timestamptz at time zone 'UTC+12');
 date_part
-----------
   2459389
(1 row)
=> SELECT extract(julian from date '2021-06-23');
 date_part
-----------
   2459389
(1 row)
```

附录 C. SQL 关键字

表 C.1 列出了在 SQL 标准以及 PostgreSQL 11.22 中被视为关键字的所有词元。背景信息可参见第 4.1.1 节。（由于篇幅所限，这里只列出了 SQL 标准最近两个版本的情况，以及用于历史比较的 SQL-92；这些版本与其他中间标准版本之间的差异很小。）

SQL 区分保留关键字和非保留关键字。根据标准，保留关键字才是真正的关键字，它们绝不允许作为标识符使用。非保留关键字只在特定上下文中具有特殊含义，在其他上下文中则可以作为标识符使用。大多数非保留关键字实际上是 SQL 规定的内置表和内置函数名称。非保留关键字这一概念，本质上只是为了声明某个词在某些上下文中附带了预定义含义。

在 PostgreSQL 解析器中，情况要更复杂一些。这里的词元分属若干不同类别，从绝不能用作标识符的词元，到在解析器中完全没有特殊地位、只是被当作普通标识符处理的词元都有。（后一类通常就是 SQL 规定的函数名。）即使是保留关键字，在PostgreSQL 中也不是绝对保留的，它们仍然可以用作列标签（例如`SELECT 55 AS CHECK`，尽管`CHECK` 是一个保留关键字）。

在表 C.1 中针对 PostgreSQL 的那一列里，我们把解析器明确识别、但又允许作为列名或表名的那些关键字归类为“非保留”。某些原本属于非保留的关键字不能用作函数名或数据类型名，因此会另外加以标记。（这类词大多表示具有特殊语法的内置函数或数据类型。函数或类型本身仍然可用，但用户不能重新定义它们。）不允许作为列名或表名的词元则标为“保留”。有些保留关键字仍允许作为函数名或数据类型名，这一点也会在表中标出。如果没有这样的标记，保留关键字就只允许作为 “AS” 列标签名使用。

一般来说，如果你在某条命令中把这里列出的关键字当作标识符使用，并因此遇到意外的解析器错误，应尝试为该标识符加上引号，看看问题是否消失。

在研读表 C.1 之前，有一点非常重要：某个关键字在PostgreSQL 中不被保留，并不意味着与该词相关的特性尚未实现；反过来，某个关键字的存在也不意味着相应特性一定存在。

表 C.1. SQL 关键字

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
A		非保留	非保留	
ABORT	非保留			
ABS		保留	保留	
ABSENT		非保留	非保留	
ABSOLUTE	非保留	非保留	非保留	保留
ACCESS	非保留			
ACCORDING		非保留	非保留	
ACTION	非保留	非保留	非保留	保留
ADA		非保留	非保留	非保留
ADD	非保留	非保留	非保留	保留
ADMIN	非保留	非保留	非保留	
AFTER	非保留	非保留	非保留	
AGGREGATE	非保留			
ALL	保留	保留	保留	保留
ALLOCATE		保留	保留	保留
ALSO	非保留			

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
ALTER	非保留	保留	保留	保留
ALWAYS	非保留	非保留	非保留	
ANALYSE	保留			
ANALYZE	保留			
AND	保留	保留	保留	保留
ANY	保留	保留	保留	保留
ARE		保留	保留	保留
ARRAY	保留	保留	保留	
ARRAY_AGG		保留	保留	
ARRAY_MAX_CARDINALITY		保留		
AS	保留	保留	保留	保留
ASC	保留	非保留	非保留	保留
ASENSITIVE		保留	保留	
ASSERTION	非保留	非保留	非保留	保留
ASSIGNMENT	非保留	非保留	非保留	
ASYMMETRIC	保留	保留	保留	
AT	非保留	保留	保留	保留
ATOMIC		保留	保留	
ATTACH	非保留			
ATTRIBUTE	非保留	非保留	非保留	
ATTRIBUTES		非保留	非保留	
AUTHORIZATION	保留（可用作函数或类型）	保留	保留	保留
AVG		保留	保留	保留
BACKWARD	非保留			
BASE64		非保留	非保留	
BEFORE	非保留	非保留	非保留	
BEGIN	非保留	保留	保留	保留
BEGIN_FRAME		保留		
BEGIN_PARTITION		保留		
BERNOULLI		非保留	非保留	
BETWEEN	非保留（不能用作函数或类型）	保留	保留	保留
BIGINT	非保留（不能用作函数或类型）	保留	保留	
BINARY	保留（可用作函数或类型）	保留	保留	
BIT	非保留（不能用作函数或类型）			保留
BIT_LENGTH				保留

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
BLOB		保留	保留	
BLOCKED		非保留	非保留	
BOM		非保留	非保留	
BOOLEAN	非保留（不能用作函数或类型）	保留	保留	
BOTH	保留	保留	保留	保留
BREADTH		非保留	非保留	
BY	非保留	保留	保留	保留
C		非保留	非保留	非保留
CACHE	非保留			
CALL	非保留	保留	保留	
CALLED	非保留	保留	保留	
CARDINALITY		保留	保留	
CASCADE	非保留	非保留	非保留	保留
CASCADED	非保留	保留	保留	保留
CASE	保留	保留	保留	保留
CAST	保留	保留	保留	保留
CATALOG	非保留	非保留	非保留	保留
CATALOG_NAME		非保留	非保留	非保留
CEIL		保留	保留	
CEILING		保留	保留	
CHAIN	非保留	非保留	非保留	
CHAR	非保留（不能用作函数或类型）	保留	保留	保留
CHARACTER	非保留（不能用作函数或类型）	保留	保留	保留
CHARACTERISTICS	非保留	非保留	非保留	
CHARACTERS		非保留	非保留	
CHARACTER_LENGTH		保留	保留	保留
CHARACTER_SET_CATALOG		非保留	非保留	非保留
CHARACTER_SET_NAME		非保留	非保留	非保留
CHARACTER_SET_SCHEMA		非保留	非保留	非保留
CHAR_LENGTH		保留	保留	保留
CHECK	保留	保留	保留	保留
CHECKPOINT	非保留			
CLASS	非保留			
CLASS_ORIGIN		非保留	非保留	非保留

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
CLOB		保留	保留	
CLOSE	非保留	保留	保留	保留
CLUSTER	非保留			
COALESCE	非保留（不能用作函数或类型）	保留	保留	保留
COBOL		非保留	非保留	非保留
COLLATE	保留	保留	保留	保留
COLLATION	保留（可用作函数或类型）	非保留	非保留	保留
COLLATION_CATALOG		非保留	非保留	非保留
COLLATION_NAME		非保留	非保留	非保留
COLLATION_SCHEMA		非保留	非保留	非保留
COLLECT		保留	保留	
COLUMN	保留	保留	保留	保留
COLUMNS	非保留	非保留	非保留	
COLUMN_NAME		非保留	非保留	非保留
COMMAND_FUNCTION		非保留	非保留	非保留
COMMAND_FUNCTION_CODE		非保留	非保留	
COMMENT	非保留			
COMMENTS	非保留			
COMMIT	非保留	保留	保留	保留
COMMITTED	非保留	非保留	非保留	非保留
CONCURRENTLY	保留（可用作函数或类型）			
CONDITION		保留	保留	
CONDITION_NUMBER		非保留	非保留	非保留
CONFIGURATION	非保留			
CONFLICT	非保留			
CONNECT		保留	保留	保留
CONNECTION	非保留	非保留	非保留	保留
CONNECTION_NAME		非保留	非保留	非保留
CONSTRAINT	保留	保留	保留	保留
CONSTRAINTS	非保留	非保留	非保留	保留
CONSTRAINT_CATALOG		非保留	非保留	非保留
CONSTRAINT_NAME		非保留	非保留	非保留

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
CONSTRAINT_SCHEMA		非保留	非保留	非保留
CONSTRUCTOR		非保留	非保留	
CONTAINS		保留	非保留	
CONTENT	非保留	非保留	非保留	
CONTINUE	非保留	非保留	非保留	保留
CONTROL		非保留	非保留	
CONVERSION	非保留			
CONVERT		保留	保留	保留
COPY	非保留			
CORR		保留	保留	
CORRESPONDING		保留	保留	保留
COST	非保留			
COUNT		保留	保留	保留
COVAR_POP		保留	保留	
COVAR_SAMP		保留	保留	
CREATE	保留	保留	保留	保留
CROSS	保留（可用作函数或类型）	保留	保留	保留
CSV	非保留			
CUBE	非保留	保留	保留	
CUME_DIST		保留	保留	
CURRENT	非保留	保留	保留	保留
CURRENT_CATALOG	保留	保留	保留	
CURRENT_DATE	保留	保留	保留	保留
CURRENT_DEFAULT_TRANSFORM_GROUP		保留	保留	
CURRENT_PATH		保留	保留	
CURRENT_ROLE	保留	保留	保留	
CURRENT_ROW		保留		
CURRENT_SCHEMA	保留（可用作函数或类型）	保留	保留	
CURRENT_TIME	保留	保留	保留	保留
CURRENT_TIMESTAMP	保留	保留	保留	保留
CURRENT_TRANSFORM_GROUP_FOR_TYPE		保留	保留	
CURRENT_USER	保留	保留	保留	保留
CURSOR	非保留	保留	保留	保留
CURSOR_NAME		非保留	非保留	非保留

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
CYCLE	非保留	保留	保留	
DATA	非保留	非保留	非保留	非保留
DATABASE	非保留			
DATALINK		保留	保留	
DATE		保留	保留	保留
DATETIME_ INTERVAL_CODE		非保留	非保留	非保留
DATETIME_ INTERVAL_ PRECISION		非保留	非保留	非保留
DAY	非保留	保留	保留	保留
DB		非保留	非保留	
DEALLOCATE	非保留	保留	保留	保留
DEC	非保留（不能用作函数或类型）	保留	保留	保留
DECIMAL	非保留（不能用作函数或类型）	保留	保留	保留
DECLARE	非保留	保留	保留	保留
DEFAULT	保留	保留	保留	保留
DEFAULTS	非保留	非保留	非保留	
DEFERRABLE	保留	非保留	非保留	保留
DEFERRED	非保留	非保留	非保留	保留
DEFINED		非保留	非保留	
DEFINER	非保留	非保留	非保留	
DEGREE		非保留	非保留	
DELETE	非保留	保留	保留	保留
DELIMITER	非保留			
DELIMITERS	非保留			
DENSE_RANK		保留	保留	
DEPENDS	非保留			
DEPTH		非保留	非保留	
DEREF		保留	保留	
DERIVED		非保留	非保留	
DESC	保留	非保留	非保留	保留
DESCRIBE		保留	保留	保留
DESCRIPTOR		非保留	非保留	保留
DETACH	非保留			
DETERMINISTIC		保留	保留	
DIAGNOSTICS		非保留	非保留	保留
DICTIONARY	非保留			
DISABLE	非保留			

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
DISCARD	非保留			
DISCONNECT		保留	保留	保留
DISPATCH		非保留	非保留	
DISTINCT	保留	保留	保留	保留
DLNEWCOPY		保留	保留	
DLPREVIOUSCOPY		保留	保留	
DLURLCOMPLETE		保留	保留	
DLURLCOMPLETEONLY		保留	保留	
DLURLCOMPLETEWRITE		保留	保留	
DLURLPATH		保留	保留	
DLURLPATHONLY		保留	保留	
DLURLPATHWRITE		保留	保留	
DLURLSCHEME		保留	保留	
DLURLSERVER		保留	保留	
DLVALUE		保留	保留	
DO	保留			
DOCUMENT	非保留	非保留	非保留	
DOMAIN	非保留	非保留	非保留	保留
DOUBLE	非保留	保留	保留	保留
DROP	非保留	保留	保留	保留
DYNAMIC		保留	保留	
DYNAMIC_FUNCTION		非保留	非保留	非保留
DYNAMIC_FUNCTION_CODE		非保留	非保留	
EACH	非保留	保留	保留	
ELEMENT		保留	保留	
ELSE	保留	保留	保留	保留
EMPTY		非保留	非保留	
ENABLE	非保留			
ENCODING	非保留	非保留	非保留	
ENCRYPTED	非保留			
END	保留	保留	保留	保留
END-EXEC		保留	保留	保留
END_FRAME		保留		
END_PARTITION		保留		
ENFORCED		非保留		
ENUM	非保留			
EQUALS		保留	非保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
ESCAPE	非保留	保留	保留	保留
EVENT	非保留			
EVERY		保留	保留	
EXCEPT	保留	保留	保留	保留
EXCEPTION				保留
EXCLUDE	非保留	非保留	非保留	
EXCLUDING	非保留	非保留	非保留	
EXCLUSIVE	非保留			
EXEC		保留	保留	保留
EXECUTE	非保留	保留	保留	保留
EXISTS	非保留（不能用作函数或类型）	保留	保留	保留
EXP		保留	保留	
EXPLAIN	非保留			
EXPRESSION		非保留		
EXTENSION	非保留			
EXTERNAL	非保留	保留	保留	保留
EXTRACT	非保留（不能用作函数或类型）	保留	保留	保留
FALSE	保留	保留	保留	保留
FAMILY	非保留			
FETCH	保留	保留	保留	保留
FILE		非保留	非保留	
FILTER	非保留	保留	保留	
FINAL		非保留	非保留	
FIRST	非保留	非保留	非保留	保留
FIRST_VALUE		保留	保留	
FLAG		非保留	非保留	
FLOAT	非保留（不能用作函数或类型）	保留	保留	保留
FLOOR		保留	保留	
FOLLOWING	非保留	非保留	非保留	
FOR	保留	保留	保留	保留
FORCE	非保留			
FOREIGN	保留	保留	保留	保留
FORTRAN		非保留	非保留	非保留
FORWARD	非保留			
FOUND		非保留	非保留	保留
FRAME_ROW		保留		
FREE		保留	保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
FREEZE	保留（可用作函数或类型）			
FROM	保留	保留	保留	保留
FS		非保留	非保留	
FULL	保留（可用作函数或类型）	保留	保留	保留
FUNCTION	非保留	保留	保留	
FUNCTIONS	非保留			
FUSION		保留	保留	
G		非保留	非保留	
GENERAL		非保留	非保留	
GENERATED	非保留	非保留	非保留	
GET		保留	保留	保留
GLOBAL	非保留	保留	保留	保留
GO		非保留	非保留	保留
GOTO		非保留	非保留	保留
GRANT	保留	保留	保留	保留
GRANTED	非保留	非保留	非保留	
GREATEST	非保留（不能用作函数或类型）			
GROUP	保留	保留	保留	保留
GROUPING	非保留（不能用作函数或类型）	保留	保留	
GROUPS	非保留	保留		
HANDLER	非保留			
HAVING	保留	保留	保留	保留
HEADER	非保留			
HEX		非保留	非保留	
HIERARCHY		非保留	非保留	
HOLD	非保留	保留	保留	
HOUR	非保留	保留	保留	保留
ID		非保留	非保留	
IDENTITY	非保留	保留	保留	保留
IF	非保留			
IGNORE		非保留	非保留	
ILIKE	保留（可用作函数或类型）			
IMMEDIATE	非保留	非保留	非保留	保留
IMMEDIATELY		非保留		
IMMUTABLE	非保留			
IMPLEMENTATION		非保留	非保留	
IMPLICIT	非保留			

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
IMPORT	非保留	保留	保留	
IN	保留	保留	保留	保留
INCLUDE	非保留			
INCLUDING	非保留	非保留	非保留	
INCREMENT	非保留	非保留	非保留	
INDENT		非保留	非保留	
INDEX	非保留			
INDEXES	非保留			
INDICATOR		保留	保留	保留
INHERIT	非保留			
INHERITS	非保留			
INITIALLY	保留	非保留	非保留	保留
INLINE	非保留			
INNER	保留（可用作函数或类型）	保留	保留	保留
INOUT	非保留（不能用作函数或类型）	保留	保留	
INPUT	非保留	非保留	非保留	保留
INSENSITIVE	非保留	保留	保留	保留
INSERT	非保留	保留	保留	保留
INSTANCE		非保留	非保留	
INSTANTIABLE		非保留	非保留	
INSTEAD	非保留	非保留	非保留	
INT	非保留（不能用作函数或类型）	保留	保留	保留
INTEGER	非保留（不能用作函数或类型）	保留	保留	保留
INTEGRITY		非保留	非保留	
INTERSECT	保留	保留	保留	保留
INTERSECTION		保留	保留	
INTERVAL	非保留（不能用作函数或类型）	保留	保留	保留
INTO	保留	保留	保留	保留
INVOKER	非保留	非保留	非保留	
IS	保留（可用作函数或类型）	保留	保留	保留
ISNULL	保留（可用作函数或类型）			
ISOLATION	非保留	非保留	非保留	保留
JOIN	保留（可用作函数或类型）	保留	保留	保留
K		非保留	非保留	
KEY	非保留	非保留	非保留	保留

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
KEY_MEMBER		非保留	非保留	
KEY_TYPE		非保留	非保留	
LABEL	非保留			
LAG		保留	保留	
LANGUAGE	非保留	保留	保留	保留
LARGE	非保留	保留	保留	
LAST	非保留	非保留	非保留	保留
LAST_VALUE		保留	保留	
LATERAL	保留	保留	保留	
LEAD		保留	保留	
LEADING	保留	保留	保留	保留
LEAKPROOF	非保留			
LEAST	非保留（不能用作函数或类型）			
LEFT	保留（可用作函数或类型）	保留	保留	保留
LENGTH		非保留	非保留	非保留
LEVEL	非保留	非保留	非保留	保留
LIBRARY		非保留	非保留	
LIKE	保留（可用作函数或类型）	保留	保留	保留
LIKE_REGEX		保留	保留	
LIMIT	保留	非保留	非保留	
LINK		非保留	非保留	
LISTEN	非保留			
LN		保留	保留	
LOAD	非保留			
LOCAL	非保留	保留	保留	保留
LOCALTIME	保留	保留	保留	
LOCALTIMESTAMP	保留	保留	保留	
LOCATION	非保留	非保留	非保留	
LOCATOR		非保留	非保留	
LOCK	非保留			
LOCKED	非保留			
LOGGED	非保留			
LOWER		保留	保留	保留
M		非保留	非保留	
MAP		非保留	非保留	
MAPPING	非保留	非保留	非保留	
MATCH	非保留	保留	保留	保留
MATCHED		非保留	非保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
MATERIALIZED	非保留			
MAX		保留	保留	保留
MAXVALUE	非保留	非保留	非保留	
MAX_CARDINALITY			保留	
MEMBER		保留	保留	
MERGE		保留	保留	
MESSAGE_LENGTH		非保留	非保留	非保留
MESSAGE_OCTET_LENGTH		非保留	非保留	非保留
MESSAGE_TEXT		非保留	非保留	非保留
METHOD	非保留	保留	保留	
MIN		保留	保留	保留
MINUTE	非保留	保留	保留	保留
MINVALUE	非保留	非保留	非保留	
MOD		保留	保留	
MODE	非保留			
MODIFIES		保留	保留	
MODULE		保留	保留	保留
MONTH	非保留	保留	保留	保留
MORE		非保留	非保留	非保留
MOVE	非保留			
MULTISET		保留	保留	
MUMPS		非保留	非保留	非保留
NAME	非保留	非保留	非保留	非保留
NAMES	非保留	非保留	非保留	保留
NAMESPACE		非保留	非保留	
NATIONAL	非保留（不能用作函数或类型）	保留	保留	保留
NATURAL	保留（可用作函数或类型）	保留	保留	保留
NCHAR	非保留（不能用作函数或类型）	保留	保留	保留
NCLOB		保留	保留	
NESTING		非保留	非保留	
NEW	非保留	保留	保留	
NEXT	非保留	非保留	非保留	保留
NFC		非保留	非保留	
NFD		非保留	非保留	
NFKC		非保留	非保留	
NFKD		非保留	非保留	
NIL		非保留	非保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
NO	非保留	保留	保留	保留
NONE	非保留（不能用作函数或类型）	保留	保留	
NORMALIZE		保留	保留	
NORMALIZED		非保留	非保留	
NOT	保留	保留	保留	保留
NOTHING	非保留			
NOTIFY	非保留			
NOTNULL	保留（可用作函数或类型）			
NOWAIT	非保留			
NTH_VALUE		保留	保留	
NTILE		保留	保留	
NULL	保留	保留	保留	保留
NULLABLE		非保留	非保留	非保留
NULLIF	非保留（不能用作函数或类型）	保留	保留	保留
NULLS	非保留	非保留	非保留	
NUMBER		非保留	非保留	非保留
NUMERIC	非保留（不能用作函数或类型）	保留	保留	保留
OBJECT	非保留	非保留	非保留	
OCCURRENCES_REGEX		保留	保留	
OCTETS		非保留	非保留	
OCTET_LENGTH		保留	保留	保留
OF	非保留	保留	保留	保留
OFF	非保留	非保留	非保留	
OFFSET	保留	保留	保留	
OIDS	非保留			
OLD	非保留	保留	保留	
ON	保留	保留	保留	保留
ONLY	保留	保留	保留	保留
OPEN		保留	保留	保留
OPERATOR	非保留			
OPTION	非保留	非保留	非保留	保留
OPTIONS	非保留	非保留	非保留	
OR	保留	保留	保留	保留
ORDER	保留	保留	保留	保留
ORDERING		非保留	非保留	
ORDINALITY	非保留	非保留	非保留	
OTHERS	非保留	非保留	非保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
OUT	非保留（不能用作函数或类型）	保留	保留	
OUTER	保留（可用作函数或类型）	保留	保留	保留
OUTPUT		非保留	非保留	保留
OVER	非保留	保留	保留	
OVERLAPS	保留（可用作函数或类型）	保留	保留	保留
OVERLAY	非保留（不能用作函数或类型）	保留	保留	
OVERRIDING	非保留	非保留	非保留	
OWNED	非保留			
OWNER	非保留			
P		非保留	非保留	
PAD		非保留	非保留	保留
PARALLEL	非保留			
PARAMETER		保留	保留	
PARAMETER_MODE		非保留	非保留	
PARAMETER_NAME		非保留	非保留	
PARAMETER_ ORDINAL_POSITI ON		非保留	非保留	
PARAMETER_ SPECIFIC_ CATALOG		非保留	非保留	
PARAMETER_ SPECIFIC_NAME		非保留	非保留	
PARAMETER_ SPECIFIC_ SCHEMA		非保留	非保留	
PARSER	非保留			
PARTIAL	非保留	非保留	非保留	保留
PARTITION	非保留	保留	保留	
PASCAL		非保留	非保留	非保留
PASSING	非保留	非保留	非保留	
PASSTHROUGH		非保留	非保留	
PASSWORD	非保留			
PATH		非保留	非保留	
PERCENT		保留		
PERCENTILE_ CONT		保留	保留	
PERCENTILE_ DISC		保留	保留	
PERCENT_RANK		保留	保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
PERIOD		保留		
PERMISSION		非保留	非保留	
PLACING	保留	非保留	非保留	
PLANS	非保留			
PLI		非保留	非保留	非保留
POLICY	非保留			
PORTION		保留		
POSITION	非保留（不能用作函数或类型）	保留	保留	保留
POSITION_REGEX		保留	保留	
POWER		保留	保留	
PRECEDES		保留		
PRECEDING	非保留	非保留	非保留	
PRECISION	非保留（不能用作函数或类型）	保留	保留	保留
PREPARE	非保留	保留	保留	保留
PREPARED	非保留			
PRESERVE	非保留	非保留	非保留	保留
PRIMARY	保留	保留	保留	保留
PRIOR	非保留	非保留	非保留	保留
PRIVILEGES	非保留	非保留	非保留	保留
PROCEDURAL	非保留			
PROCEDURE	非保留	保留	保留	保留
PROCEDURES	非保留			
PROGRAM	非保留			
PUBLIC		非保留	非保留	保留
PUBLICATION	非保留			
QUOTE	非保留			
RANGE	非保留	保留	保留	
RANK		保留	保留	
READ	非保留	非保留	非保留	保留
READS		保留	保留	
REAL	非保留（不能用作函数或类型）	保留	保留	保留
REASSIGN	非保留			
RECHECK	非保留			
RECOVERY		非保留	非保留	
RECURSIVE	非保留	保留	保留	
REF	非保留	保留	保留	
REFERENCES	保留	保留	保留	保留
REFERENCING	非保留	保留	保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
REFRESH	非保留			
REGR_AVGX		保留	保留	
REGR_AVGY		保留	保留	
REGR_COUNT		保留	保留	
REGR_INTERCEPT		保留	保留	
REGR_R2		保留	保留	
REGR_SLOPE		保留	保留	
REGR_SXX		保留	保留	
REGR_SXY		保留	保留	
REGR_SYY		保留	保留	
REINDEX	非保留			
RELATIVE	非保留	非保留	非保留	保留
RELEASE	非保留	保留	保留	
RENAME	非保留			
REPEATABLE	非保留	非保留	非保留	非保留
REPLACE	非保留			
REPLICA	非保留			
REQUIRING		非保留	非保留	
RESET	非保留			
RESPECT		非保留	非保留	
RESTART	非保留	非保留	非保留	
RESTORE		非保留	非保留	
RESTRICT	非保留	非保留	非保留	保留
RESULT		保留	保留	
RETURN		保留	保留	
RETURNED_ CARDINALITY		非保留	非保留	
RETURNED_ LENGTH		非保留	非保留	非保留
RETURNED_ OCTET_LENGTH		非保留	非保留	非保留
RETURNED_ SQLSTATE		非保留	非保留	非保留
RETURNING	保留	非保留	非保留	
RETURNS	非保留	保留	保留	
REVOKE	非保留	保留	保留	保留
RIGHT	保留（可用作函数或类型）	保留	保留	保留
ROLE	非保留	非保留	非保留	
ROLLBACK	非保留	保留	保留	保留
ROLLUP	非保留	保留	保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
ROUTINE	非保留	非保留	非保留	
ROUTINES	非保留			
ROUTINE_CATALOG		非保留	非保留	
ROUTINE_NAME		非保留	非保留	
ROUTINE_SCHEMA		非保留	非保留	
ROW	非保留（不能用作函数或类型）	保留	保留	
ROWS	非保留	保留	保留	保留
ROW_COUNT		非保留	非保留	非保留
ROW_NUMBER		保留	保留	
RULE	非保留			
SAVEPOINT	非保留	保留	保留	
SCALE		非保留	非保留	非保留
SCHEMA	非保留	非保留	非保留	保留
SCHEMAS	非保留			
SCHEMA_NAME		非保留	非保留	非保留
SCOPE		保留	保留	
SCOPE_CATALOG		非保留	非保留	
SCOPE_NAME		非保留	非保留	
SCOPE_SCHEMA		非保留	非保留	
SCROLL	非保留	保留	保留	保留
SEARCH	非保留	保留	保留	
SECOND	非保留	保留	保留	保留
SECTION		非保留	非保留	保留
SECURITY	非保留	非保留	非保留	
SELECT	保留	保留	保留	保留
SELECTIVE		非保留	非保留	
SELF		非保留	非保留	
SENSITIVE		保留	保留	
SEQUENCE	非保留	非保留	非保留	
SEQUENCES	非保留			
SERIALIZABLE	非保留	非保留	非保留	非保留
SERVER	非保留	非保留	非保留	
SERVER_NAME		非保留	非保留	非保留
SESSION	非保留	非保留	非保留	保留
SESSION_USER	保留	保留	保留	保留
SET	非保留	保留	保留	保留
SETOF	非保留（不能用作函数或类型）			
SETS	非保留	非保留	非保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
SHARE	非保留			
SHOW	非保留			
SIMILAR	保留（可用作函数或类型）	保留	保留	
SIMPLE	非保留	非保留	非保留	
SIZE		非保留	非保留	保留
SKIP	非保留			
SMALLINT	非保留（不能用作函数或类型）	保留	保留	保留
SNAPSHOT	非保留			
SOME	保留	保留	保留	保留
SOURCE		非保留	非保留	
SPACE		非保留	非保留	保留
SPECIFIC		保留	保留	
SPECIFICTYPE		保留	保留	
SPECIFIC_NAME		非保留	非保留	
SQL	非保留	保留	保留	保留
SQLCODE				保留
SQLERROR				保留
SQLEXCEPTION		保留	保留	
SQLSTATE		保留	保留	保留
SQLWARNING		保留	保留	
SQRT		保留	保留	
STABLE	非保留			
STANDALONE	非保留	非保留	非保留	
START	非保留	保留	保留	
STATE		非保留	非保留	
STATEMENT	非保留	非保留	非保留	
STATIC		保留	保留	
STATISTICS	非保留			
STDDEV_POP		保留	保留	
STDDEV_SAMP		保留	保留	
STDIN	非保留			
STDOUT	非保留			
STORAGE	非保留			
STRICT	非保留			
STRIP	非保留	非保留	非保留	
STRUCTURE		非保留	非保留	
STYLE		非保留	非保留	
SUBCLASS_ ORIGIN		非保留	非保留	非保留

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
SUBMULTISET		保留	保留	
SUBSCRIPTION	非保留			
SUBSTRING	非保留（不能用作函数或类型）	保留	保留	保留
SUBSTRING_REGEX		保留	保留	
SUCCEEDS		保留		
SUM		保留	保留	保留
SYMMETRIC	保留	保留	保留	
SYSID	非保留			
SYSTEM	非保留	保留	保留	
SYSTEM_TIME		保留		
SYSTEM_USER		保留	保留	保留
T		非保留	非保留	
TABLE	保留	保留	保留	保留
TABLES	非保留			
TABLESAMPLE	保留（可用作函数或类型）	保留	保留	
TABLESPACE	非保留			
TABLE_NAME		非保留	非保留	非保留
TEMP	非保留			
TEMPLATE	非保留			
TEMPORARY	非保留	非保留	非保留	保留
TEXT	非保留			
THEN	保留	保留	保留	保留
TIES	非保留	非保留	非保留	
TIME	非保留（不能用作函数或类型）	保留	保留	保留
TIMESTAMP	非保留（不能用作函数或类型）	保留	保留	保留
TIMEZONE_HOUR		保留	保留	保留
TIMEZONE_MINUTE		保留	保留	保留
TO	保留	保留	保留	保留
TOKEN		非保留	非保留	
TOP_LEVEL_COUNT		非保留	非保留	
TRAILING	保留	保留	保留	保留
TRANSACTION	非保留	非保留	非保留	保留
TRANSACTIONS_COMMITTED		非保留	非保留	
TRANSACTIONS_ROLLED_BACK		非保留	非保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
TRANSACTION_ACTIVE		非保留	非保留	
TRANSFORM	非保留	非保留	非保留	
TRANSFORMS		非保留	非保留	
TRANSLATE		保留	保留	保留
TRANSLATE_REGEX		保留	保留	
TRANSLATION		保留	保留	保留
TREAT	非保留（不能用作函数或类型）	保留	保留	
TRIGGER	非保留	保留	保留	
TRIGGER_CATALOG		非保留	非保留	
TRIGGER_NAME		非保留	非保留	
TRIGGER_SCHEMA		非保留	非保留	
TRIM	非保留（不能用作函数或类型）	保留	保留	保留
TRIM_ARRAY		保留	保留	
TRUE	保留	保留	保留	保留
TRUNCATE	非保留	保留	保留	
TRUSTED	非保留			
TYPE	非保留	非保留	非保留	非保留
TYPES	非保留			
UESCAPE		保留	保留	
UNBOUNDED	非保留	非保留	非保留	
UNCOMMITTED	非保留	非保留	非保留	非保留
UNDER		非保留	非保留	
UNENCRYPTED	非保留			
UNION	保留	保留	保留	保留
UNIQUE	保留	保留	保留	保留
UNKNOWN	非保留	保留	保留	保留
UNLINK		非保留	非保留	
UNLISTEN	非保留			
UNLOGGED	非保留			
UNNAMED		非保留	非保留	非保留
UNNEST		保留	保留	
UNTIL	非保留			
UNTYPED		非保留	非保留	
UPDATE	非保留	保留	保留	保留
UPPER		保留	保留	保留
URI		非保留	非保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
USAGE		非保留	非保留	保留
USER	保留	保留	保留	保留
USER_DEFINED_TYPE_CATALOG		非保留	非保留	
USER_DEFINED_TYPE_CODE		非保留	非保留	
USER_DEFINED_TYPE_NAME		非保留	非保留	
USER_DEFINED_TYPE_SCHEMA		非保留	非保留	
USING	保留	保留	保留	保留
VACUUM	非保留			
VALID	非保留	非保留	非保留	
VALIDATE	非保留			
VALIDATOR	非保留			
VALUE	非保留	保留	保留	保留
VALUES	非保留（不能用作函数或类型）	保留	保留	保留
VALUE_OF		保留		
VARBINARY		保留	保留	
VARCHAR	非保留（不能用作函数或类型）	保留	保留	保留
VARIADIC	保留			
VARYING	非保留	保留	保留	保留
VAR_POP		保留	保留	
VAR_SAMP		保留	保留	
VERBOSE	保留（可用作函数或类型）			
VERSION	非保留	非保留	非保留	
VERSIONING		保留		
VIEW	非保留	非保留	非保留	保留
VIEWS	非保留			
VOLATILE	非保留			
WHEN	保留	保留	保留	保留
WHENEVER		保留	保留	保留
WHERE	保留	保留	保留	保留
WHITESPACE	非保留	非保留	非保留	
WIDTH_BUCKET		保留	保留	
WINDOW	保留	保留	保留	
WITH	保留	保留	保留	保留
WITHIN	非保留	保留	保留	
WITHOUT	非保留	保留	保留	

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
WORK	非保留	非保留	非保留	保留
WRAPPER	非保留	非保留	非保留	
WRITE	非保留	非保留	非保留	保留
XML	非保留	保留	保留	
XMLAGG		保留	保留	
XMLATTRIBUTES	非保留（不能用作函数或类型）	保留	保留	
XMLBINARY		保留	保留	
XMLCAST		保留	保留	
XMLCOMMENT		保留	保留	
XMLCONCAT	非保留（不能用作函数或类型）	保留	保留	
XMLDECLARATION		非保留	非保留	
XMLDOCUMENT		保留	保留	
XMLELEMENT	非保留（不能用作函数或类型）	保留	保留	
XMLEXISTS	非保留（不能用作函数或类型）	保留	保留	
XMLFOREST	非保留（不能用作函数或类型）	保留	保留	
XMLITERATE		保留	保留	
XMLNAMESPACES	非保留（不能用作函数或类型）	保留	保留	
XMLPARSE	非保留（不能用作函数或类型）	保留	保留	
XMLPI	非保留（不能用作函数或类型）	保留	保留	
XMLQUERY		保留	保留	
XMLROOT	非保留（不能用作函数或类型）			
XMLSCHEMA		非保留	非保留	
XMLSERIALIZE	非保留（不能用作函数或类型）	保留	保留	
XMLTABLE	非保留（不能用作函数或类型）	保留	保留	
XMLTEXT		保留	保留	
XMLVALIDATE		保留	保留	
YEAR	非保留	保留	保留	保留
YES	非保留	非保留	非保留	
ZONE	非保留	非保留	非保留	保留

附录 D. SQL 符合性

本节尝试概述 PostgreSQL 在多大程度上符合当前的 SQL 标准。下文并不是一份完整的符合性声明，但会在合理且对用户有用的范围内，尽可能详细地介绍主要主题。

SQL 标准的正式名称是 ISO/IEC 9075“Database Language SQL”。该标准会不时发布修订版，最近一次更新出现在 2011 年。2011 年版称为 ISO/IEC 9075:2011，也简称 SQL:2011。此前的版本依次为 SQL:2008、SQL:2006、SQL:2003、SQL:1999 和 SQL-92。每个版本都会取代前一版本，因此宣称符合更早版本并无官方意义。PostgreSQL 的开发目标是在不违背传统特性或常识的前提下，尽可能符合标准的最新正式版本。SQL 标准要求的许多特性都得到了支持，虽然有时语法或功能会略有不同。随着时间推移，仍可期待它在符合性方面继续推进。

SQL-92 为符合性定义了三个特性集：入门（Entry）、中级（Intermediate）和完整（Full）。多数宣称符合SQL 标准的数据库管理系统，实际上只符合入门（Entry）级别，因为中级（Intermediate）和完整（Full）级别中的全部特性要么过于庞杂，要么与既有行为相冲突。

从 SQL:1999 开始，SQL 标准不再沿用SQL-92 中那种效果不佳且过于宽泛的三级分类，而是定义了大量独立特性。其中很大一个子集构成了核心（“Core”）特性，也就是每个符合 SQL 标准的实现都必须提供的特性。其余特性则完全是可选的。某些可选特性被组合在一起，形成“包”，SQL 实现可以声明符合这些包，从而表明符合特定的特性组。

从 SQL:2003 开始的各版标准还被拆分为若干部分，每一部分都有一个简称。请注意，这些部分的编号并不是连续的。

- ISO/IEC 9075-1 框架（SQL/Framework）
- ISO/IEC 9075-2 基础（SQL/Foundation）
- ISO/IEC 9075-3 调用级接口（SQL/CLI）
- ISO/IEC 9075-4 持久化存储模块（SQL/PSM）
- ISO/IEC 9075-9 外部数据管理（SQL/MED）
- ISO/IEC 9075-10 对象语言绑定（SQL/OLB）
- ISO/IEC 9075-11 信息与定义模式（SQL/Schemata）
- ISO/IEC 9075-13 使用 Java 语言的例程和类型（SQL/JRT）
- ISO/IEC 9075-14 XML 相关规范（SQL/XML）

PostgreSQL 核心部分涵盖第 1、2、9、11 和 14 部分。第 3 部分由 ODBC 驱动实现，第 13 部分由 PL/Java 插件实现，但目前尚未核验这些组件的确切符合性。PostgreSQL 目前还没有第 4 和 10 部分的实现。

PostgreSQL 支持 SQL:2011 的大多数主要特性。在达到完整“Core”符合性所要求的 179 个强制特性中，PostgreSQL 至少符合其中 160 个。此外，还有一长串已支持的可选特性。值得一提的是，在写作本文时，还没有任何数据库管理系统的现行版本宣称完全符合 SQL:2011 的核心要求。

接下来的两节先列出 PostgreSQL 已支持的特性，再列出SQL:2011 中定义但 PostgreSQL 尚未支持的特性。这两个列表都只是近似描述：某个被列为已支持的特性，可能仍有少量细节并不完全符合标准；反之，一个被列为不支持的特性，其很大一部分实际上也可能已经实现。关于哪些内容可用、哪些内容不可用，文档正文始终是最准确的信息来源。

> **注意**
>
> 带连字符的特性代码表示子特性。因此，如果某个特定子特性不受支持，则即使该主特性的其他子特性受支持，该主特性也会被列为不受支持。

D.1. 支持的特性

标识符	包	描述	注释
B012		嵌入式 C	
B021		直接 SQL	
E011	核心	数值数据类型	
E011-01	核心	INTEGER 和 SMALLINT 数据类型	
E011-02	核心	REAL、DOUBLE PRECISION 和 FLOAT 数据类型	
E011-03	核心	DECIMAL 和 NUMERIC 数据类型	
E011-04	核心	算术操作符	
E011-05	核心	数值比较	
E011-06	核心	数值数据类型之间的隐式转换	
E021	核心	字符数据类型	
E021-01	核心	CHARACTER 数据类型	
E021-02	核心	CHARACTER VARYING 数据类型	
E021-03	核心	字符字面量	
E021-04	核心	CHARACTER_LENGTH 函数	计数前去掉 CHARACTER 值的尾随空格
E021-05	核心	OCTET_LENGTH 函数	
E021-06	核心	SUBSTRING 函数	
E021-07	核心	字符串接	
E021-08	核心	UPPER 和 LOWER 函数	
E021-09	核心	TRIM 函数	
E021-10	核心	字符串类型之间的隐式转换	
E021-11	核心	POSITION 函数	

标识符	包	描述	注释
E021-12	核心	字符比较	
E031	核心	标识符	
E031-01	核心	定界标识符	
E031-02	核心	小写标识符	
E031-03	核心	尾随下划线	
E051	核心	基本查询规范	
E051-01	核心	SELECT DISTINCT	
E051-02	核心	GROUP BY 子句	
E051-04	核心	GROUP BY 可以包含不在 <select list> 中的列	
E051-05	核心	选择列表项可以重命名	
E051-06	核心	HAVING 子句	
E051-07	核心	选择列表中带限定的 *	
E051-08	核心	FROM 子句中的相关名	
E051-09	核心	在 FROM 子句中重命名列	
E061	核心	基本谓词和搜索条件	
E061-01	核心	比较谓词	
E061-02	核心	BETWEEN 谓词	
E061-03	核心	带值列表的 IN 谓词	
E061-04	核心	LIKE 谓词	
E061-05	核心	LIKE 谓词的 ESCAPE 子句	
E061-06	核心	NULL 谓词	
E061-07	核心	量化比较谓词	
E061-08	核心	EXISTS 谓词	
E061-09	核心	比较谓词中的子查询	
E061-11	核心	IN 谓词中的子查询	
E061-12	核心	量化比较谓词中的子查询	
E061-13	核心	相关子查询	
E061-14	核心	搜索条件	
E071	核心	基本查询表达式	
E071-01	核心	UNION DISTINCT 表操作符	
E071-02	核心	UNION ALL 表操作符	
E071-03	核心	EXCEPT DISTINCT 表操作符	
E071-05	核心	通过表操作符组合的列不必具有完全相同的数据类型	
E071-06	核心	子查询中的表操作符	
E081	核心	基本权限	
E081-01	核心	SELECT 权限	
E081-02	核心	DELETE 权限	

标识符	包	描述	注释
E081-03	核心	表级 INSERT 权限	
E081-04	核心	表级 UPDATE 权限	
E081-05	核心	列级 UPDATE 权限	
E081-06	核心	表级 REFERENCES 权限	
E081-07	核心	列级 REFERENCES 权限	
E081-08	核心	WITH GRANT OPTION	
E081-09	核心	USAGE 权限	
E081-10	核心	EXECUTE 权限	
E091	核心	集合函数	
E091-01	核心	AVG	
E091-02	核心	COUNT	
E091-03	核心	MAX	
E091-04	核心	MIN	
E091-05	核心	SUM	
E091-06	核心	ALL 量词	
E091-07	核心	DISTINCT 量词	
E101	核心	基本数据操纵	
E101-01	核心	INSERT 语句	
E101-03	核心	搜索式 UPDATE 语句	
E101-04	核心	搜索式 DELETE 语句	
E111	核心	单行 SELECT 语句	
E121	核心	基本游标支持	
E121-01	核心	DECLARE CURSOR	
E121-02	核心	ORDER BY 列不必出现在选择列表中	
E121-03	核心	ORDER BY 子句中的值表达式	
E121-04	核心	OPEN 语句	
E121-06	核心	定位式 UPDATE 语句	
E121-07	核心	定位式 DELETE 语句	
E121-08	核心	CLOSE 语句	
E121-10	核心	FETCH 语句隐含 NEXT	
E121-17	核心	WITH HOLD 游标	
E131	核心	空值支持（以空值代替值）	
E141	核心	基本完整性约束	
E141-01	核心	NOT NULL 约束	
E141-02	核心	NOT NULL 列的 UNIQUE 约束	
E141-03	核心	PRIMARY KEY 约束	

标识符	包	描述	注释
E141-04	核心	基本 FOREIGN KEY 约束，引用删除动作和引用更新动作均默认为 NO ACTION	
E141-06	核心	CHECK 约束	
E141-07	核心	列默认值	
E141-08	核心	由 PRIMARY KEY 推导出 NOT NULL	
E141-10	核心	外键中的名称可以按任意顺序指定	
E151	核心	事务支持	
E151-01	核心	COMMIT 语句	
E151-02	核心	ROLLBACK 语句	
E152	核心	基本 SET TRANSACTION 语句	
E152-01	核心	SET TRANSACTION 语句：ISOLATION LEVEL SERIALIZABLE 子句	
E152-02	核心	SET TRANSACTION 语句：READ ONLY 和 READ WRITE 子句	
E153	核心	带子查询的可更新查询	
E161	核心	以双减号开头的 SQL 注释	
E171	核心	SQLSTATE 支持	
F021	核心	基本信息模式	
F021-01	核心	COLUMNS 视图	
F021-02	核心	TABLES 视图	
F021-03	核心	VIEWS 视图	
F021-04	核心	TABLE_CONSTRAINTS 视图	
F021-05	核心	REFERENTIAL_CONSTRAINTS 视图	
F021-06	核心	CHECK_CONSTRAINTS 视图	
F031	核心	基本模式操纵	
F031-01	核心	用于创建持久基表的 CREATE TABLE 语句	
F031-02	核心	CREATE VIEW 语句	
F031-03	核心	GRANT 语句	
F031-04	核心	ALTER TABLE 语句：ADD COLUMN 子句	
F031-13	核心	DROP TABLE 语句：RESTRICT 子句	
F031-16	核心	DROP VIEW 语句：RESTRICT 子句	

标识符	包	描述	注释
F031-19	核心	REVOKE 语句：RESTRICT 子句	
F032		CASCADE 删除行为	
F033		ALTER TABLE 语句：DROP COLUMN 子句	
F034		扩展 REVOKE 语句	
F034-01		由模式对象拥有者以外的用户执行 REVOKE 语句	
F034-02		REVOKE 语句：GRANT OPTION FOR 子句	
F034-03		使用 REVOKE 语句撤销被授权者以 WITH GRANT OPTION 获得的权限	
F041	核心	基本连接表	
F041-01	核心	内连接（但不一定包括 INNER 关键字）	
F041-02	核心	INNER 关键字	
F041-03	核心	LEFT OUTER JOIN	
F041-04	核心	RIGHT OUTER JOIN	
F041-05	核心	外连接可以嵌套	
F041-07	核心	左外连接或右外连接的内表也可以用于内连接	
F041-08	核心	支持所有比较操作符（而不仅仅是 =）	
F051	核心	基本日期与时间	
F051-01	核心	DATE 数据类型（包括 DATE 字面量支持）	
F051-02	核心	TIME 数据类型（包括 TIME 字面量支持），小数秒精度至少为 0	
F051-03	核心	TIMESTAMP 数据类型（包括 TIMESTAMP 字面量支持），小数秒精度至少为 0 和 6	
F051-04	核心	DATE、TIME 和 TIMESTAMP 数据类型上的比较谓词	
F051-05	核心	日期时间类型与字符串类型之间的显式 CAST	
F051-06	核心	CURRENT_DATE	
F051-07	核心	LOCALTIME	
F051-08	核心	LOCALTIMESTAMP	
F052	增强的日期时间功能	时间间隔与日期时间运算	
F053		OVERLAPS 谓词	
F081	核心	视图中的 UNION 和 EXCEPT	

标识符	包	描述	注释
F111		SERIALIZABLE 以外的隔离级别	
F111-01		READ UNCOMMITTED 隔离级别	
F111-02		READ COMMITTED 隔离级别	
F111-03		REPEATABLE READ 隔离级别	
F131	核心	分组操作	
F131-01	核心	涉及分组视图的查询支持 WHERE、GROUP BY 和 HAVING 子句	
F131-02	核心	涉及分组视图的查询支持多个表	
F131-03	核心	涉及分组视图的查询支持集合函数	
F131-04	核心	带 GROUP BY、HAVING 子句和分组视图的子查询	
F131-05	核心	带 GROUP BY、HAVING 子句和分组视图的单行 SELECT	
F171		每个用户可有多个模式	
F191	增强的完整性管理	引用删除动作	
F200		TRUNCATE TABLE 语句	
F201	核心	CAST 函数	
F202		TRUNCATE TABLE：标识列重启选项	
F221	核心	显式默认值	
F222		INSERT 语句：DEFAULT VALUES 子句	
F231		权限表	
F231-01		TABLE_PRIVILEGES 视图	
F231-02		COLUMN_PRIVILEGES 视图	
F231-03		USAGE_PRIVILEGES 视图	
F251		域支持	
F261	核心	CASE 表达式	
F261-01	核心	简单 CASE	
F261-02	核心	搜索式 CASE	
F261-03	核心	NULLIF	
F261-04	核心	COALESCE	
F262		扩展 CASE 表达式	
F271		复合字符字面量	
F281		LIKE 增强	
F302		INTERSECT 表操作符	

标识符	包	描述	注释
F302-01		INTERSECT DISTINCT 表操作符	
F302-02		INTERSECT ALL 表操作符	
F304		EXCEPT ALL 表操作符	
F311-01	核心	CREATE SCHEMA	
F311-02	核心	用于持久基表的 CREATE TABLE	
F311-03	核心	CREATE VIEW	
F311-04	核心	CREATE VIEW：WITH CHECK OPTION	
F311-05	核心	GRANT 语句	
F321		用户授权	
F361		子程序支持	
F381		扩展模式操纵	
F381-01		ALTER TABLE 语句：ALTER COLUMN 子句	
F381-02		ALTER TABLE 语句：ADD CONSTRAINT 子句	
F381-03		ALTER TABLE 语句：DROP CONSTRAINT 子句	
F382		更改列的数据类型	
F383		设置列非空子句	
F384		删除标识属性子句	
F386		设置标识列生成方式子句	
F391		长标识符	
F392		标识符中的 Unicode 转义	
F393		字面量中的 Unicode 转义	
F401		扩展连接表	
F401-01		NATURAL JOIN	
F401-02		FULL OUTER JOIN	
F401-04		CROSS JOIN	
F402		LOB、数组和多重集的命名列连接	
F411	增强的日期时间功能	时区规范	字面量解释方面存在差异
F421		国家字符	
F431		只读可滚动游标	
F431-01		带显式 NEXT 的 FETCH	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	

标识符	包	描述	注释
F441		扩展集合函数支持	
F442		集合函数中的混合列引用	
F471	核心	标量子查询值	
F481	核心	扩展 NULL 谓词	
F491	增强的完整性管理	约束管理	
F501	核心	特性和符合性视图	
F501-01	核心	SQL_FEATURES 视图	
F501-02	核心	SQL_SIZING 视图	
F501-03	核心	SQL_LANGUAGES 视图	
F502		增强的文档表	
F502-01		SQL_SIZING_PROFILES 视图	
F502-02		SQL_IMPLEMENTATION_INFO 视图	
F502-03		SQL_PACKAGES 视图	
F531		临时表	
F555	增强的日期时间功能	增强的秒精度	
F561		完整的值表达式	
F571		真值测试	
F591		派生表	
F611		指示符数据类型	
F641		行构造器和表构造器	
F651		目录名称限定符	
F661		简单表	
F672		回溯检查约束	
F690		排序规则支持	但不支持字符集
F692		扩展排序规则支持	
F701	增强的完整性管理	引用更新动作	
F711		ALTER DOMAIN	
F731		INSERT 列权限	
F751		视图 CHECK 增强	
F761		会话管理	
F762		CURRENT_CATALOG	
F763		CURRENT_SCHEMA	
F771		连接管理	
F781		自引用操作	
F791		不敏感游标	
F801		完整集合函数	
F850		<query expression> 中顶层的 <order by clause>	

标识符	包	描述	注释
F851		子查询中的 <order by clause>	
F852		视图中顶层的 <order by clause>	
F855		<query expression> 中嵌套的 <order by clause>	
F856		<query expression> 中嵌套的 <fetch first clause>	
F857		<query expression> 中顶层的 <fetch first clause>	
F858		子查询中的 <fetch first clause>	
F859		视图中顶层的 <fetch first clause>	
F860		<fetch first clause> 中的 <fetch first row count>	
F861		<query expression> 中顶层的 <result offset clause>	
F862		子查询中的 <result offset clause>	
F863		<query expression> 中嵌套的 <result offset clause>	
F864		视图中顶层的 <result offset clause>	
F865		<result offset clause> 中的 <offset row count>	
S071	增强的对象支持	函数名和类型名解析中的 SQL 路径	
S092		用户定义类型的数组	
S095		通过查询构造数组	
S096		可选数组边界	
S098		ARRAY_AGG	
S111	增强的对象支持	查询表达式中的 ONLY	
S201		用于数组的 SQL 调用例程	
S201-01		数组参数	
S201-02		数组作为函数结果类型	
S211	增强的对象支持	用户定义的类型转换函数	
S301		增强的 UNNEST	
T031		BOOLEAN 数据类型	
T071		BIGINT 数据类型	

标识符	包	描述	注释
T121		查询表达式中的 WITH（不含 RECURSIVE）	
T122		子查询中的 WITH（不含 RECURSIVE）	
T131		递归查询	
T132		子查询中的递归查询	
T141		SIMILAR 谓词	
T151		DISTINCT 谓词	
T152		带否定的 DISTINCT 谓词	
T171		表定义中的 LIKE 子句	
T172		表定义中的 AS 子查询子句	
T173		表定义中的扩展 LIKE 子句	
T174		标识列	
T177		sequence 生成器支持：简单重启选项	
T178		标识列：简单重启选项	
T191	增强的完整性管理	引用动作 RESTRICT	
T201	增强的完整性管理	引用约束的可比较数据类型	
T211-01	主动数据库，增强的完整性管理	由对一个基表的 UPDATE、INSERT 或 DELETE 激活的触发器	
T211-02	主动数据库，增强的完整性管理	BEFORE 触发器	
T211-03	主动数据库，增强的完整性管理	AFTER 触发器	
T211-04	主动数据库，增强的完整性管理	FOR EACH ROW 触发器	
T211-05	主动数据库，增强的完整性管理	可以指定调用触发器前必须为真的搜索条件	
T211-07	主动数据库，增强的完整性管理	TRIGGER 权限	
T212	增强的完整性管理	增强的触发器能力	
T213		INSTEAD OF 触发器	
T231		敏感游标	
T241		START TRANSACTION 语句	
T271		保存点	
T281		列粒度的 SELECT 权限	
T285		增强的派生列名称	
T312		OVERLAY 函数	
T321-01	核心	不带重载的用户定义函数	

标识符	包	描述	注释
T321-02	核心	不带重载的用户定义存储过程	
T321-03	核心	函数调用	
T321-04	核心	CALL 语句	
T321-06	核心	ROUTINES 视图	
T321-07	核心	PARAMETERS 视图	
T323		外部例程的显式安全性	
T325		带限定的 SQL 参数引用	
T331		基本角色	
T341		SQL 调用函数和过程的重载	
T351		括号式 SQL 注释（/*...*/ 注释）	
T431	OLAP	扩展分组能力	
T432		嵌套和串接的 GROUPING SETS	
T433		多参数 GROUPING 函数	
T441		ABS 和 MOD 函数	
T461		对称 BETWEEN 谓词	
T491		LATERAL 派生表	
T501		增强的 EXISTS 谓词	
T521		CALL 语句中的命名实参	
T551		默认语法的可选关键字	
T581		正则表达式子串函数	
T591		允许空值的列上的 UNIQUE 约束	
T611	OLAP	基本 OLAP 操作	
T613		采样	
T614		NTILE 函数	
T615		LEAD 和 LAG 函数	
T617		FIRST_VALUE 和 LAST_VALUE 函数	
T620		WINDOW 子句：GROUPS 选项	
T621		增强的数值函数	
T631	核心	只有一个列表元素的 IN 谓词	
T651		SQL 例程中的 SQL 模式语句	
T655		循环依赖的例程	
X010		XML 类型	
X011		XML 类型的数组	
X014		XML 类型的属性	
X016		持久 XML 值	

标识符	包	描述	注释
X020		XMLConcat	
X031		XMLElement	
X032		XMLForest	
X034		XMLAgg	
X035		XMLAgg：ORDER BY 选项	
X036		XMLComment	
X037		XMLPI	
X040		基本表映射	
X041		基本表映射：省略空值	
X042		基本表映射：空值表示为 nil	
X043		基本表映射：表表示为森林	
X044		基本表映射：表表示为元素	
X045		基本表映射：带目标命名空间	
X046		基本表映射：数据映射	
X047		基本表映射：元数据映射	
X048		基本表映射：二进制串的 base64 编码	
X049		基本表映射：二进制串的十六进制编码	
X050		高级表映射	
X051		高级表映射：省略空值	
X052		高级表映射：空值表示为 nil	
X053		高级表映射：表表示为森林	
X054		高级表映射：表表示为元素	
X055		高级表映射：带目标命名空间	
X056		高级表映射：数据映射	
X057		高级表映射：元数据映射	
X058		高级表映射：二进制串的 base64 编码	
X059		高级表映射：二进制串的十六进制编码	
X060		XMLParse：字符串输入和 CONTENT 选项	
X061		XMLParse：字符串输入和 DOCUMENT 选项	
X070		XMLSerialize：字符串序列化和 CONTENT 选项	

标识符	包	描述	注释
X071		XMLSerialize：字符串序列化和 DOCUMENT 选项	
X072		XMLSerialize：字符串序列化	
X090		XML 文档谓词	
X120		SQL 例程中的 XML 参数	
X121		外部例程中的 XML 参数	
X222		XML 传递机制 BY REF	
X301		XMLTable：派生列列表选项	
X302		XMLTable：序号列选项	
X303		XMLTable：列默认值选项	
X304		XMLTable：传递上下文项	必须是 XML DOCUMENT
X400		名称和标识符映射	
X410		更改列的数据类型：XML 类型	

D.2. 不支持的特性

下列在 SQL:2011 中定义的特性，在本版本的PostgreSQL 中尚未实现。在少数情况下，可用等效功能替代。

标识符	包	描述	注释
B011		嵌入式 Ada	
B013		嵌入式 COBOL	
B014		嵌入式 Fortran	
B015		嵌入式 MUMPS	
B016		嵌入式 Pascal	
B017		嵌入式 PL/I	
B031		基本动态 SQL	
B032		扩展动态 SQL	
B032-01		<describe input statement>	
B033		无类型的 SQL 调用函数实参	
B034		动态指定游标属性	
B035		非扩展描述符名称	
B041		嵌入式 SQL 异常声明的扩展	
B051		增强的执行权限	
B111		模块语言 Ada	
B112		模块语言 C	
B113		模块语言 COBOL	

标识符	包	描述	注释
B114		模块语言 Fortran	
B115		模块语言 MUMPS	
B116		模块语言 Pascal	
B117		模块语言 PL/I	
B121		例程语言 Ada	
B122		例程语言 C	
B123		例程语言 COBOL	
B124		例程语言 Fortran	
B125		例程语言 MUMPS	
B126		例程语言 Pascal	
B127		例程语言 PL/I	
B128		例程语言 SQL	
B211		模块语言 Ada：VARCHAR 和 NUMERIC 支持	
B221		例程语言 Ada：VARCHAR 和 NUMERIC 支持	
E182	核心	模块语言	
F054		DATE 类型优先级列表中的 TIMESTAMP	
F121		基本诊断管理	
F121-01		GET DIAGNOSTICS 语句	
F121-02		SET TRANSACTION 语句：DIAGNOSTICS SIZE 子句	
F122		增强的诊断管理	
F123		全部诊断	
F181	核心	多模块支持	
F263		简单 CASE 表达式中以逗号分隔的谓词	
F291		UNIQUE 谓词	
F301		查询表达式中的 CORRESPONDING	
F311	核心	模式定义语句	
F312		MERGE 语句	可考虑 INSERT ... ON CONFLICT DO UPDATE
F313		增强的 MERGE 语句	
F314		带 DELETE 分支的 MERGE 语句	
F341		使用情况表	没有 ROUTINE_*_USAGE 表
F385		删除列生成表达式子句	
F394		可选的规范形式指定	
F403		分区连接表	
F451		字符集定义	

标识符	包	描述	注释
F461		命名字符集	
F492		可选的表约束强制执行	
F521	增强的完整性管理	断言	
F671	增强的完整性管理	CHECK 中的子查询	有意省略
F693		SQL 会话和客户端模块排序规则	
F695		字符转换支持	
F696		附加的字符转换文档	
F721		可延迟约束	仅限外键和唯一键
F741		引用 MATCH 类型	尚不支持部分匹配
F812	核心	基本标记	
F813		扩展标记	
F821		本地表引用	
F831		完整游标更新	
F831-01		可更新的可滚动游标	
F831-02		可更新的有序游标	
F841		LIKE_REGEX 谓词	
F842		OCCURRENCES_REGEX 函数	
F843		POSITION_REGEX 函数	
F844		SUBSTRING_REGEX 函数	
F845		TRANSLATE_REGEX 函数	
F846		正则表达式操作符中的八位组支持	
F847		非常量正则表达式	
F866		FETCH FIRST 子句：PERCENT 选项	
F867		FETCH FIRST 子句：WITH TIES 选项	
S011	核心	独立数据类型	
S011-01	核心	USER_DEFINED_TYPES 视图	
S023	基本对象支持	基本结构化类型	
S024	增强的对象支持	增强的结构化类型	
S025		FINAL 结构化类型	
S026		自引用结构化类型	
S027		按特定方法名创建方法	
S028		可调整顺序的 UDT 选项列表	
S041	基本对象支持	基本引用类型	
S043	增强的对象支持	增强的引用类型	
S051	基本对象支持	创建指定类型的表	部分支持
S081	增强的对象支持	子表	

标识符	包	描述	注释
S091		基本数组支持	部分支持
S091-01		内置数据类型的数组	
S091-02		独立类型的数组	
S091-03		数组表达式	
S094		引用类型的数组	
S097		数组元素赋值	
S151	基本对象支持	类型谓词	
S161	增强的对象支持	子类型处理	
S162		引用的子类型处理	
S202		用于多重集的 SQL 调用例程	
S231	增强的对象支持	结构化类型定位器	
S232		数组定位器	
S233		多重集定位器	
S241		变换函数	
S242		ALTER TRANSFORM 语句	
S251		用户定义的排序	
S261		特定类型方法	
S271		基本多重集支持	
S272		用户定义类型的多重集	
S274		引用类型的多重集	
S275		高级多重集支持	
S281		嵌套集合类型	
S291		整行唯一约束	
S401		基于数组类型的独立类型	
S402		基于独立类型的独立类型	
S403		ARRAY_MAX_CARDINALITY	
S404		TRIM_ARRAY	
T011		信息模式中的时间戳	
T021		BINARY 和 VARBINARY 数据类型	
T022		BINARY 和 VARBINARY 数据类型的高级支持	
T023		复合二进制字面量	
T024		二进制字面量中的空格	
T041	基本对象支持	基本 LOB 数据类型支持	
T041-01	基本对象支持	BLOB 数据类型	
T041-02	基本对象支持	CLOB 数据类型	
T041-03	基本对象支持	LOB 数据类型的 POSITION、LENGTH、LOWER、TRIM、UPPER 和 SUBSTRING 函数	

标识符	包	描述	注释
T041-04	基本对象支持	LOB 数据类型的串接	
T041-05	基本对象支持	LOB 定位器：不可保持	
T042		扩展 LOB 数据类型支持	
T043		倍数后缀 T	
T044		倍数后缀 P	
T051		行类型	
T052		行类型的 MAX 和 MIN	
T053		全字段引用的显式别名	
T061		UCS 支持	
T101		增强的可空性判定	
T111		可更新的连接、并集和列	
T175		生成列	
T176		sequence 生成器支持	
T180		系统版本化表	
T181		应用时间周期表	
T211	主动数据库，增强的完整性管理	基本触发器能力	
T211-06	主动数据库，增强的完整性管理	支持触发器与约束交互的运行时规则	
T211-08	主动数据库，增强的完整性管理	同一事件的多个触发器按在目录中创建的顺序执行	有意省略
T251		SET TRANSACTION 语句：LOCAL 选项	
T261		链式事务	
T272		增强的保存点管理	
T301		函数依赖	部分支持
T321	核心	基本 SQL 调用例程	
T321-05	核心	RETURN 语句	
T322	PSM	声明的数据类型属性	
T324		SQL 例程的显式安全性	
T326		表函数	
T332		扩展角色	大部分支持
T434		GROUP BY DISTINCT	
T471		结果集返回值	
T472		DESCRIBE CURSOR	
T495		组合数据更改与检索	语法不同
T502		时间周期谓词	
T511		事务计数	
T522		SQL 调用过程的 IN 参数默认值	除调用中的 DEFAULT 关键字外均受支持
T561		可保持的定位器	
T571		返回数组的外部 SQL 调用函数	

标识符	包	描述	注释
T572		返回多重集的外部 SQL 调用函数	
T601		本地游标引用	
T612		高级 OLAP 操作	支持部分形式
T616		LEAD 和 LAG 函数的空值处理选项	
T618		NTH_VALUE 函数	函数存在，但缺少部分选项
T619		嵌套窗口函数	
T641		多列赋值	仅支持部分语法变体
T652		SQL 例程中的 SQL 动态语句	
T653		外部例程中的 SQL 模式语句	
T654		外部例程中的 SQL 动态语句	
M001		数据链接	
M002		通过 SQL/CLI 使用数据链接	
M003		通过嵌入式 SQL 使用数据链接	
M004		外部数据支持	部分支持
M005		外部模式支持	
M006		GetSQLString 例程	
M007		TransmitRequest	
M009		GetOpts 和 GetStatistics 例程	
M010		外部数据包装器支持	API 不同
M011		通过 Ada 使用数据链接	
M012		通过 C 使用数据链接	
M013		通过 COBOL 使用数据链接	
M014		通过 Fortran 使用数据链接	
M015		通过 M 使用数据链接	
M016		通过 Pascal 使用数据链接	
M017		通过 PL/I 使用数据链接	
M018		Ada 中的外部数据包装器接口例程	
M019		C 中的外部数据包装器接口例程	API 不同
M020		COBOL 中的外部数据包装器接口例程	
M021		Fortran 中的外部数据包装器接口例程	

标识符	包	描述	注释
M022		MUMPS 中的外部数据包装器接口例程	
M023		Pascal 中的外部数据包装器接口例程	
M024		PL/I 中的外部数据包装器接口例程	
M030		SQL 服务器外部数据支持	
M031		外部数据包装器通用例程	
X012		XML 类型的多重集	
X013		基于 XML 类型的独立类型	
X015		XML 类型的字段	
X025		XMLCast	
X030		XMLDocument	
X038		XMLText	
X065		XMLParse：BLOB 输入和 CONTENT 选项	
X066		XMLParse：BLOB 输入和 DOCUMENT 选项	
X068		XMLSerialize：BOM	
X069		XMLSerialize：INDENT	
X073		XMLSerialize：BLOB 序列化和 CONTENT 选项	
X074		XMLSerialize：BLOB 序列化和 DOCUMENT 选项	
X075		XMLSerialize：BLOB 序列化	
X076		XMLSerialize：VERSION	
X077		XMLSerialize：显式 ENCODING 选项	
X078		XMLSerialize：显式 XML 声明	
X080		XML 发布中的命名空间	
X081		查询级 XML 命名空间声明	
X082		DML 中的 XML 命名空间声明	
X083		DDL 中的 XML 命名空间声明	
X084		复合语句中的 XML 命名空间声明	
X085		预定义命名空间前缀	
X086		XMLTable 中的 XML 命名空间声明	
X091		XML 内容谓词	
X096		XMLExists	仅限 XPath 1.0

标识符	包	描述	注释
X100		宿主语言的 XML 支持：CONTENT 选项	
X101		宿主语言的 XML 支持：DOCUMENT 选项	
X110		宿主语言的 XML 支持：VARCHAR 映射	
X111		宿主语言的 XML 支持：CLOB 映射	
X112		宿主语言的 XML 支持：BLOB 映射	
X113		宿主语言的 XML 支持：STRIP WHITESPACE 选项	
X114		宿主语言的 XML 支持：PRESERVE WHITESPACE 选项	
X131		查询级 XMLBINARY 子句	
X132		DML 中的 XMLBINARY 子句	
X133		DDL 中的 XMLBINARY 子句	
X134		复合语句中的 XMLBINARY 子句	
X135		子查询中的 XMLBINARY 子句	
X141		IS VALID 谓词：由数据决定的情况	
X142		IS VALID 谓词：ACCORDING TO 子句	
X143		IS VALID 谓词：ELEMENT 子句	
X144		IS VALID 谓词：模式位置	
X145		检查约束以外的 IS VALID 谓词	
X151		带 DOCUMENT 选项的 IS VALID 谓词	
X152		带 CONTENT 选项的 IS VALID 谓词	
X153		带 SEQUENCE 选项的 IS VALID 谓词	
X155		IS VALID 谓词：不带 ELEMENT 子句的 NAMESPACE	
X157		IS VALID 谓词：带 ELEMENT 子句的 NO NAMESPACE	
X160		已注册 XML Schema 的基本信息模式	

标识符	包	描述	注释
X161		已注册 XML Schema 的高级信息模式	
X170		XML 空值处理选项	
X171		NIL ON NO CONTENT 选项	
X181		XML(DOCUMENT(UNTYPED)) 类型	
X182		XML(DOCUMENT(ANY)) 类型	
X190		XML(SEQUENCE) 类型	
X191		XML(DOCUMENT(XMLSCHEMA)) 类型	
X192		XML(CONTENT(XMLSCHEMA)) 类型	
X200		XMLQuery	
X201		XMLQuery：RETURNING CONTENT	
X202		XMLQuery：RETURNING SEQUENCE	
X203		XMLQuery：传递上下文项	
X204		XMLQuery：初始化 XQuery 变量	
X205		XMLQuery：EMPTY ON EMPTY 选项	
X206		XMLQuery：NULL ON EMPTY 选项	
X211		XML 1.1 支持	
X221		XML 传递机制 BY VALUE	
X231		XML(CONTENT(UNTYPED)) 类型	
X232		XML(CONTENT(ANY)) 类型	
X241		XML 发布中的 RETURNING CONTENT	
X242		XML 发布中的 RETURNING SEQUENCE	
X251		XML(DOCUMENT(UNTYPED)) 类型的持久 XML 值	
X252		XML(DOCUMENT(ANY)) 类型的持久 XML 值	
X253		XML(CONTENT(UNTYPED)) 类型的持久 XML 值	
X254		XML(CONTENT(ANY)) 类型的持久 XML 值	

标识符	包	描述	注释
X255		XML(SEQUENCE) 类型的持久 XML 值	
X256		XML(DOCUMENT(XMLSCHEMA)) 类型的持久 XML 值	
X257		XML(CONTENT(XMLSCHEMA)) 类型的持久 XML 值	
X260		XML 类型：ELEMENT 子句	
X261		XML 类型：不带 ELEMENT 子句的 NAMESPACE	
X263		XML 类型：带 ELEMENT 子句的 NO NAMESPACE	
X264		XML 类型：模式位置	
X271		XMLValidate：由数据决定的情况	
X272		XMLValidate：ACCORDING TO 子句	
X273		XMLValidate：ELEMENT 子句	
X274		XMLValidate：模式位置	
X281		带 DOCUMENT 选项的 XMLValidate	
X282		带 CONTENT 选项的 XMLValidate	
X283		带 SEQUENCE 选项的 XMLValidate	
X284		XMLValidate：不带 ELEMENT 子句的 NAMESPACE	
X286		XMLValidate：带 ELEMENT 子句的 NO NAMESPACE	
X300		XMLTable	仅限 XPath 1.0
X305		XMLTable：初始化 XQuery 变量	

D.3. XML 的限制以及对 SQL/XML 的符合性

SQL:2006 对 ISO/IEC 9075-14（SQL/XML）中的 XML 相关规范进行了重大修订。PostgreSQL 对 XML 数据类型和相关函数的实现大体遵循较早的 2003 年版，同时借鉴了后续版本的一些内容。具体而言：

- 当现行标准提供一组 XML 数据类型，用于以无类型或 XML Schema 类型化的变体保存“文档”或“内容”，并提供 `XML(SEQUENCE)` 类型来保存任意 XML 内容片段时，PostgreSQL 只提供单一的 `xml` 类型，该类型可以保存“文档”或“内容”。标准中的“sequence”类型在这里没有对应物。

- PostgreSQL 提供了 SQL:2006 引入的两个函数，但采用的是使用 XPath 1.0 语言的变体，而不是标准为它们指定的 XML Query。

本节介绍由此产生的一些差异。

D.3.1. 查询受限于 XPath 1.0

PostgreSQL 特有的函数 `xpath()` 和 `xpath_exists()` 使用 XPath 语言查询 XML 文档。PostgreSQL 还提供了标准函数 `XMLEXISTS` 和 `XMLTABLE` 的仅限 XPath 版本，而这两个标准函数正式使用的是 XQuery 语言。对于所有这些函数，PostgreSQL 都依赖 libxml2 库，而该库只提供 XPath 1.0。

XQuery 与 XPath 2.0 及更高版本之间有很强的关联：凡是在两者中语法有效且都能成功执行的表达式，其结果都相同（一个小例外是包含数字字符引用或预定义实体引用的表达式，XQuery 会将其替换为相应字符，而 XPath 则保持不变）。但这种关联并不适用于这些语言与 XPath 1.0 之间；后者是更早的语言，在许多方面都不同。

需要记住两类限制：一是 SQL 标准所规定函数从 XQuery 缩减到 XPath 的限制，二是标准函数和PostgreSQL 特有函数都只能使用 XPath 1.0 的限制。

D.3.1.1. 将 XQuery 限制为 XPath

XQuery 具有而 XPath 不具备的特性包括：

- XQuery 表达式除了能够产生 XPath 所有可能的值之外，还可以构造并返回新的 XML 节点。XPath 可以创建并返回原子类型的值（数字、字符串等），但它只能返回作为表达式输入提供的文档中本就存在的 XML 节点。

- XQuery 提供了用于迭代、排序和分组的控制结构。

- XQuery 允许声明和使用局部函数。

较新的 XPath 版本开始提供与这些能力部分重叠的功能（例如函数式风格的 `for-each` 和 `sort`、匿名函数，以及用`parse-xml` 从字符串创建节点），但在 XPath 3.0 之前并不具备这些特性。

D.3.1.2. XPath 仅限于 1.0

对熟悉 XQuery 和 XPath 2.0 或更高版本的开发人员来说，XPath 1.0 存在若干必须面对的差异：

- XQuery/XPath 表达式的基础类型 `sequence` 可以包含 XML 节点、原子值或两者兼有，但 XPath 1.0 中不存在这一类型。1.0 表达式只能产生一个节点集（包含零个或多个 XML 节点），或者一个单一的原子值。

- 与可按任意顺序包含任意条目的 XQuery/XPath 序列（sequence）不同，XPath 1.0 的节点集没有保证顺序，并且像任何集合一样，不允许同一条目出现多次。

> **注意**
>
> libxml2 库看起来总是把节点集以其成员在输入文档中的相对顺序返回给 PostgreSQL。但它的文档并未承诺这种行为，而 XPath 1.0 表达式本身也无法控制这一点。

- 虽然 XQuery/XPath 提供了 XML Schema 中定义的全部类型，以及围绕这些类型的大量操作符和函数，但 XPath 1.0 只有节点集以及三种原子类型：`boolean`、`double` 和 `string`。
- XPath 1.0 没有条件操作符。像 `if ( hat ) then hat/@size else "no hat"` 这样的 XQuery/XPath 表达式，在 XPath 1.0 中没有等价写法。
- XPath 1.0 没有用于字符串的排序比较操作符。`"cat" < "dog"` 和`"cat" > "dog"` 都为假，因为它们实际上都是两个`NaN` 之间的数值比较。相比之下，=和!=确实会把字符串按字符串来比较。
- XPath 1.0 模糊了 XQuery/XPath 所定义的值比较与一般比较之间的区别。`sale/@hatsize = 7` 和 `sale/@customer = "alice"` 都属于存在量词比较：只要存在任意一个 `sale` 具有给定属性值，结果就为真。但`sale/@taxable = false()` 则是将整个节点集的有效布尔值进行值比较。只有在没有任何 `sale` 拥有 `taxable` 属性时，它才为真。
- 在 XQuery/XPath 数据模型中，文档节点既可以是文档形式（即恰好有一个顶层元素，且其外部只允许注释和处理指令），也可以是内容形式（放宽这些约束）。它在 XPath 1.0 中对应的根节点只能是文档形式。这也是为什么 `xml` 值在作为上下文项传给任何基于 XPath 的 PostgreSQL 函数时，必须采用文档形式的部分原因。

这里强调的差异并不全面。在 XQuery 以及 XPath 2.0 及更高版本中，存在 XPath 1.0 兼容模式，而 W3C 给出的函数库变化[1]和语言变化[2]列表，在该模式下提供了更完整但仍不穷尽的差异说明。兼容模式也不能让较新的语言与 XPath 1.0 完全等同。

D.3.1.3. SQL 与 XML 数据类型和值之间的映射

在 SQL:2006 及之后的版本中，标准 SQL 数据类型与 XML Schema 类型之间双向转换的规则都被精确定义了。但是，这些规则是用 XQuery/XPath 的类型和语义来表达的，无法直接应用到 XPath 1.0 那套不同的数据模型上。

当 PostgreSQL 把 SQL 数据值映射到 XML（例如 `xmlelement`），或者把 XML 映射到 SQL（例如`xmltable` 的输出列）时，除少数被特殊处理的情况外，PostgreSQL 只是简单地假定：XML 数据类型在 XPath 1.0 中的字符串形式可以作为 SQL 数据类型的文本输入形式，反之亦然。这一规则的优点在于简单，同时对许多数据类型来说，其结果也与标准规定的映射大致相似。在本版本中，如果`xmltable` 的列表达式产生布尔值或双精度值，则需要显式类型转换；参见第 D.3.2 节。

如果需要与其他系统互操作，那么对于某些数据类型，可能需要显式使用数据类型格式化函数（例如第 9.8 节中的函数）来生成标准规定的映射。

[1] https://www.w3.org/TR/2010/REC-xpath-functions-20101214/#xpath1-compatibility
[2] https://www.w3.org/TR/xpath20/#id-backwards-compatibility

D.3.2. 实现带来的附带限制

本节讨论的限制并非 libxml2 库本身固有，而是适用于PostgreSQL 当前实现的限制。

D.3.2.1. 布尔或双精度类型的 `xmltable` 列需要类型转换

如果 `xmltable` 列表达式的计算结果是 XPath 布尔值或数值，就会产生 “unexpected XPath object type” 错误。解决办法是改写列表达式，将其放在 XPath `string` 函数中；这样，PostgreSQL 就能成功地将该字符串值赋给布尔或双精度类型的 SQL 输出列。

D.3.2.2. 列路径结果或 SQL 结果列为 XML 类型

在本版本中，计算结果为 XML 节点集的`xmltable` 列表达式可以赋给 XML 类型的 SQL 结果列，结果是以下内容的串接：对于节点集中的大多数节点类型，是一个包含该节点 XPath 1.0 字符串值的文本节点；但对于元素节点，则是该节点本身的副本。只有当这样的节点集仅含一个节点时，才可以将其赋给非 XML 类型的 SQL 列；此时，大多数节点类型的字符串值会被替换为空字符串，元素节点的字符串值会被替换为仅由其直接文本子节点（不包括后代节点中的文本节点）串接而成的内容，而文本节点或属性节点的字符串值则按 XPath 1.0 的定义处理。赋给 XML 类型结果列的 XPath 字符串值必须可以解析为 XML。

最好不要开发依赖这些行为的代码，因为它们与规范相去甚远，而且在PostgreSQL 12 中已被更改。

D.3.2.3. 仅支持 `BY VALUE` 传递机制

SQL 标准定义了两种传递机制，适用于从 SQL 向 XML 函数传递 XML 参数，或在接收结果时使用：`BY REF` 表示某个特定 XML 值保留其节点身份，`BY VALUE` 表示传递 XML 的内容，但不保留节点身份。该机制既可以写在参数列表之前，作为所有参数的默认机制，也可以写在任一参数之后，以覆盖默认机制。

为了说明其中的区别，如果*x* 是一个 XML 值，那么在 SQL:2006 环境中，下面两个查询将分别产生 true 和 false：

```
SELECT XMLQUERY('$a is $b' PASSING BY REF x AS a, x AS b NULL ON EMPTY);
SELECT XMLQUERY('$a is $b' PASSING BY VALUE x AS a, x AS b NULL ON EMPTY);
```

在本版本中，PostgreSQL 会接受 `BY REF` 出现在 `XMLEXISTS` 或 `XMLTABLE` 构造中的用法，但会忽略它。`xml` 数据类型保存的是序列化后的字符串表示，因而没有需要保留的节点身份，所以传递在实际上始终等同于`BY VALUE`。

D.3.2.4. 无法向查询传递命名参数

基于 XPath 的函数支持传递一个参数作为 XPath 表达式的上下文项，但不支持传递额外的值，供表达式作为命名参数使用。

D.3.2.5. 没有 `XML(SEQUENCE)` 类型

PostgreSQL 的 `xml` 数据类型只能保存 `DOCUMENT` 或 `CONTENT` 形式的值。一个 XQuery/XPath 表达式的上下文项必须是单个 XML 节点或原子值，但 XPath 1.0 又进一步限制它只能是 XML 节点，并且没有允许`CONTENT` 的节点类型。归根结底，良构的`DOCUMENT` 是 PostgreSQL 能够作为 XPath 上下文项提供的唯一 XML 值形式。

附录 E. 发布说明

发布说明包含每个 PostgreSQL 发布中的重要变更，并将主要特性和迁移问题列在最前面。发布说明不包含只影响少数用户的变更，也不包含仅限内部实现、因而对用户不可见的变更。例如，优化器几乎在每次发布中都会得到改进，但用户通常只会把这些改进感知为查询速度更快。

每个发布的完整变更列表都可以通过查看对应版本的 Git 日志获得。`pgsql-committers` 邮件列表[1]也记录了所有源代码变更。另有一个 Web 界面[2]可显示特定文件的变更。

每个条目旁边标出的姓名表示该条目的主要开发者。当然，所有变更都经历了社区讨论和补丁审查，因此每个条目实际上都是社区共同努力的成果。

E.1. 发布版本 11.22

发布日期：. 2023-11-09

本次发布包含来自 11.21 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

这预计将是 PostgreSQL 11.X 系列的最后一个发行版。建议用户尽快更新到更新的发行分支。

E.1.1. 迁移到版本 11.22

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.21 的版本升级，请参见第 E.2 节。

E.1.2. 变更

- 修复 `DISTINCT "any"` 聚合函数中对未知类型参数的处理（Tom Lane）

  此错误导致 `text` 类型的值在运行时被解释为 `unknown` 类型的值（即以零终止的字符串）。这可能导致泄露 `text` 值之后的服务器内存。

  PostgreSQL 项目感谢 Jingzhou Fu 报告此问题。（CVE-2023-5868）

- 检测计算新数组维度时的整数溢出（Tom Lane）

  当向超出当前数组边界的数组下标赋予新元素时，在极端情况下可能发生未检测到的整数溢出。可能利用内存越界写入来执行任意代码，也可能泄露服务器内存。

  PostgreSQL 项目感谢 Pedro Gallegos 报告此问题。（CVE-2023-5869）

- 防止 `pg_signal_backend` 角色向后台工作进程和自动清理进程发送信号（Noah Misch，Jelte Fennema-Nio）

  文档说明 `pg_signal_backend` 不能向超级用户拥有的进程发送信号。但它能够向这些后台进程发送信号，因为它们公布的角色 OID 为零。现在将此视为表示超级用户所有权。取消这些进程类型之一的安全影响就核心代码而言相当小（我们只会启动另一个），但扩展可能添加更容易受影响的后台工作进程。

  同时确保在这些进程中正确设置 `is_superuser` 参数。对于该疏忽，目前没有已知的具体安全后果，但对于某些扩展来说可能是重要的。

[1] https://www.postgresql.org/list/pgsql-committers/
[2] https://git.postgresql.org/gitweb/?p=postgresql.git;a=summary

PostgreSQL 项目感谢 Hemanth Sandrana 和 Mahendrakar Srinivasarao 报告此问题。（CVE-2023-5870）

- 修复具有多个分区键的 hash 分区表的分区步骤生成和运行时分区剪枝（David Rowley）

  某些涉及其中一个分区键上 `IS NULL` 条件的情况可能导致崩溃。

- 修复 B-树标记/恢复处理 ScalarArrayOpExpr 子句时的边界情况（Peter Geoghegan）

  将索引扫描恢复到先前标记的位置时，如果扫描恰好前进到 ScalarArrayOpExpr（即 `indexcol = ANY(ARRAY[])`）子句的匹配末尾，代码可能会遗漏必要的设置步骤。这可能导致漏掉一些应当被获取的行。

- 修复当集合返回函数反复返回零行时的查询内内存泄漏（Tom Lane）

- 修复将 `cursor_to_xmlschema()` 应用于非数据返回 Portal 时的崩溃（Boyu Yang）

- 在各种 SQL 函数中更妥善地处理无效索引（Noah Misch）

  如果将 `pgstatindex()`、`pgstatginindex()`、`pgstathashindex()` 或 `pgstattuple()` 应用于无效索引，现在会报告错误。如果将 `brin_desummarize_range()`、`brin_summarize_new_values()`、`brin_summarize_range()` 或 `gin_clean_pending_list()` 应用于无效索引，则除了报告一条调试级别的消息外不做任何操作。之前这些函数会尝试处理该索引，并可能根据失败的 `CREATE INDEX` 遗留的内容而以奇怪的方式失败。

- 避免在 `to_tsvector()` 处理长输入时过早的内存分配失败（Tom Lane）

- 修复 `tsvectorrecv()` 中构造的 `tsvector` 的过度分配（Denis Erokhin）

  如果传入的向量包含位置数据，二进制接收函数会在完成的 `tsvector` 中留下浪费的空间（大约等于位置数据的大小）。在极端情况下，这可能导致对于发出时在长度限制内的向量产生“maximum total lexeme length exceeded”失败。无论如何都可能导致磁盘空间浪费。

- 修复 `gtsvector_picksplit()` 中的不正确编码（Alexander Lakhin）

  这可能导致 `tsvector` 列上 GiST 索引的页面分裂决策不佳。

- 确保在删除 `ON COMMIT DROP` 临时表时拥有快照（Tom Lane）

  这可以防止当临时表的某些目录条目的字段宽度足够大而需要 toast 时（例如非常复杂的 `CHECK` 条件）可能出现的异常行为。

- 避免在 `system()` 刚刚 fork 的子进程中对关闭信号的不当响应（Nathan Bossart）

  此修复避免了一个竞争条件：由 `system()` fork 出来但尚未 exec 目标子程序的子进程可能接收并处理原本发给父服务器进程的信号。这将导致重复的清理操作，后果不佳。

- 避免在相关 SQL 函数中对 `pg_control` 的撕裂读取（Thomas Munro）

  在读取 `pg_control` 之前获取适当的锁，以确保获得该文件的一致视图。

- 跟踪缓存的 `CALL` 语句的依赖关系，并在需要时重新规划（Tom Lane）

DDL 命令，例如替换已内联到 CALL 参数中的函数，可能需要重新规划已被 PL/pgSQL 缓存的 CALL。之前这不会发生，导致异常行为或诸如 “cache lookup failed” 之类的奇怪错误。

- 在检查来自外层查询级别的 RECORD 类型 Var 时正确跟踪嵌套深度（Richard Guo）

  此疏忽可能导致断言失败、核心转储或 “bogus varno” 错误。

- 避免逆解析包含对复合常量字段引用的视图时出现 “record type has not been registered” 错误（Tom Lane）

- 允许从 RECORD 类型的 ROW() 表达式中提取字段（Tom Lane）

  知道我们将这些字段命名为 f1、f2 等的 SQL 代码，可以使用这些名称从表达式中提取字段。此项修改最初在版本 13 中完成，现在反向移植到更早的分支，以支持一个相关缺陷的测试。

- 修复 RECORD 类型缓存管理中的错误处理缺陷（Thomas Munro）

  在恰好错误的时刻发生内存不足错误可能留下不一致的状态，导致无限循环。

- 修复在同一会话中出错后重试逻辑解码时的断言失败（Hou Zhijie）

- 避免对在解析分析过程中不接受有意义处理的工具语句进行计划缓存重新验证（Tom Lane）

  除了节省一些开销外，这还可以防止在缓存失效后对于不能设置快照的语句（如 SET TRANSACTION ISOLATION LEVEL）发生失败。

- 在使用期间将按引用传递的 attmissingval 值保存在长生命周期的上下文中（Andrew Dunstan）

  这避免了当元组槽的生命周期超过用于构造其值的元组描述符时可能出现的悬挂指针使用。

- 在 ALTER ROLE 之后重新计算 search_path 的有效值（Jeff Davis）

  这确保在重命名角色后，特殊字符串 $user 的含义会被重新确定。

- 修复 GenericXLogFinish 中的操作顺序（Jeff Davis）

  此代码在将更改的缓冲区标记为脏之前就写入了 WAL，违反了崩溃安全所需的条件。核心代码没有使用此函数，但扩展会使用（例如 contrib/bloom）。

- 移除 PL/Python 异常处理中的不正确断言（Alexander Lakhin）

- 修复 pg_restore，使选择性恢复能够同时包含所选表的表级和列级 ACL（Euler Taveira，Tom Lane）

  之前，如果两种类型都存在，只会恢复表级 ACL。

- 避免在 pg_basebackup 中生成无效的临时槽名（Jelte Fennema）

  这仅在服务器连接通过 pgbouncer 时被观察到。

- 在 contrib/amcheck 中，不将被中断的页面删除报告为损坏（Noah Misch）

  此修复防止了 “the first child of leftmost target page is not leftmost of its level”、“block NNNN is not leftmost” 或 “left link/right link pair in index XXXX not

in agreement" 的误报。这些误报出现在 amcheck 在未完成的 B-树索引页面删除之后且 VACUUM 清理之前运行时。

- 修复 contrib/btree_gin 在 interval 列上的索引在执行使用 < 或 <= 操作符的索引扫描时的失败（Dean Rasheed）

  这样的索引扫描无法返回所有应返回的条目。

- 消除在最新 macOS 上的各种编译时警告（Tom Lane）

  Xcode 15（随 macOS Sonoma 发布）改变了链接器的行为方式，在构建 PostgreSQL 时导致许多重复库警告。这些警告是无害的，但很烦人，因此避免两次引用相同的库。同时移除了 -multiply_defined suppress 链接器选项的使用，该选项显然长期以来一直没有实际作用，而现在使用它会引发警告。

- 从默认时区缩写列表中移除 PHOT（凤凰群岛时间）（Tom Lane）

  此缩写在默认列表中的存在可能导致在最新的 Debian 和 Ubuntu 发行版上出现失败，因为它们不再默认安装底层的 tzdb 条目。由于这是一个为总人口约二十几人的区域编造的缩写，不太可能有人会需要它。如果确有需要，可以通过自定义缩写文件将其恢复。

E.2. 发布版本 11.21

发布日期：. 2023-08-10

本次发布包含来自 11.20 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

PostgreSQL 社区将于 2023 年 11 月停止发布 11.X 系列的更新。建议用户尽快升级到较新的发布分支。

E.2.1. 迁移到版本 11.21

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果使用 BRIN 索引，建议在更新后重建这些索引；请参阅下面的第二个变更条目。

此外，如果是从早于 11.14 的版本升级，请参见第 E.9 节。

E.2.2. 变更

- 如果模式或所有者名称包含引号、反斜杠或美元符号，则不允许将其替换到扩展脚本中（Noah Misch）

  此限制可防止受信任的扩展中的 SQL 注入风险。

  PostgreSQL 项目感谢 Micah Gate、Valerie Woolard、Tim Carey-Smith 和 Christoph Berg 报告此问题。（CVE-2023-39417）

- 修复 BRIN 索引中空范围（无行）与全 NULL 范围之间的混淆，以及全 NULL 摘要的不正确合并（Tomas Vondra）

  这些疏忽都可能导致忘记某个 BRIN 索引范围包含 NULL 值，从而可能使后续应当返回 NULL 值的查询遗漏这些值。

  此修复本身不会纠正已有的错误 BRIN 条目。建议对任何可能用于搜索 NULL 值的 BRIN 索引执行 REINDEX。

- 避免在 DROP DATABASE 被中断时留下损坏的数据库（Andres Freund）

  如果 DROP DATABASE 在已开始执行不可逆步骤之后被中断，目标数据库仍然可以访问（因为其 pg_database 行的移除会被回滚），但其内容已损坏。修复方法是在开始执行不可逆操作之前将数据库标记为不可访问。此后的失败将使数据库仍部分存在，但除了再次执行 DROP DATABASE 之外无法对其进行任何操作。

- 确保分区索引在创建时被正确标记为有效或无效（Michael Paquier）

  如果新建的分区索引匹配某个分区上已有但无效的索引，该分区索引可能会被过早地标记为有效。这可能导致后续对分区表的查询出现异常行为或断言失败。

- 在 ALTER TABLE ATTACH PARTITION 将分区索引匹配到子索引时忽略无效的子索引（Michael Paquier）

  这样的索引现在会被忽略，转而创建新的子索引。

- 修复在所有分区都已附加后将分区索引标记为有效时可能出现的失败（Michael Paquier）

  更新索引的 pg_index 条目时可能对其他列使用了过期的数据。报告的一个症状是 “attempted to update invisible tuple” 错误。

- 修复 ALTER EXTENSION SET SCHEMA，使其在扩展包含扩展模式之外的对象时报告错误（Michael Paquier，Heikki Linnakangas）

  当扩展包含多个模式中的对象时报告错误一直是预期行为；但检查逻辑编写有误，未能检测到某些情况，导致令人意外的行为。

- 不要在规划器中使用部分唯一索引进行唯一性证明（David Rowley）

  这可能产生不正确的计划，因为如果相关索引未用于扫描表，则从表中读取的行的假定唯一性可能不成立。

- 避免为具有伪常量连接子句的外部连接生成不正确的计划（Etsuro Fujita）

  规划器目前缺少将伪常量连接子句附加到下推的远程连接的支持，因此在此类情况下禁用远程连接的生成。（更好的解决方案需要对规划器数据结构进行破坏 ABI 的更改，因此必须等到未来的主版本。）

- 在展开规则动作时正确处理 RLS 策略表达式和安全屏障视图中的子 SELECT（Tom Lane）

- 修复 SERIALIZABLE 隔离模式下冲突检测中的竞争条件（Thomas Munro）

  使用位图堆扫描、使用 GIN 索引以及检查初始为空的 B-树索引时可能会遗漏冲突。所有这些情况都可能因不当允许相互冲突的事务提交而破坏可串行性。

- 修复尝试更新复合列的字段时的间歇性失败（Tom Lane）

  如果复合列的整体值宽到需要行外 TOAST 存储，那么在不恰当时机发生的缓存刷新可能导致错误或服务器崩溃。

- 防止非常复杂的文本检索模式导致栈溢出崩溃（Tom Lane）

- 允许 pg_hba.conf 和 pg_ident.conf 中的令牌长度最多为 10240 字节（Tom Lane）

  之前 256 字节的限制已被发现对某些使用场景不足。

- 修复 C++ 内存不足情况的错误处理（Heikki Linnakangas）

  如果正在使用 JIT，C++ `new` 调用中的内存不足会导致 PostgreSQL 的 FATAL 错误，而非预期的 C++ 异常。

- 避免在页面释放导致空闲空间范围合并时丢失对可能有用的共享内存段的跟踪（Dongming Liu）

  确保将该段移动到与其新的空闲空间量相应的 “bin” 中，以便后续搜索能够找到它。

- 允许 `VACUUM` 在检测到某些类型的 B-树索引损坏后继续执行（Peter Geoghegan）

  如果检测到无效的兄弟页面链接，记录该问题并继续执行，而不是像之前那样抛出错误。只有 `REINDEX` 才能修复损坏的索引，但阻止 `VACUUM` 在此之前完成可能会使情况变得更糟。

- 确保在 `VACUUM` 检测到 `pg_database.datfrozenxid` 或 `pg_database.datminmxid` 中的无效数据后释放 `WrapLimitsVacuumLock`（Andres Freund）

  未能释放此锁可能导致后续死锁，尽管如果会话退出或遇到其他错误，该锁会被清理。

- 避免在崩溃恢复期间对预备事务的双重重放（suyu.cmj，Michael Paquier）

  在检查点进行到部分时发生崩溃，且某些两阶段事务状态数据已被该检查点刷新到磁盘后，崩溃恢复可能尝试两次重放预备事务，导致诸如启动进程中 “lock is already held” 之类的致命错误。

- 确保新建但仍为空的表在下一次检查点时执行 `fsync`（Heikki Linnakangas）

  如果没有这一步，在操作系统崩溃导致空文件消失后，对该表的后续操作可能会出现 “could not open file” 错误。

- 确保不记录 WAL 的索引的 init fork 的创建被记录到 WAL（Heikki Linnakangas）

  虽然不记录 WAL 的索引的主数据 fork 不记录 WAL，但其 init fork 应该记录，以确保在崩溃后有一个一致的状态来恢复索引。如果 init fork 不包含数据，则此步骤被遗漏，这是标准索引 AM 不使用的情况；但也许某些扩展会有这种行为。

- 修复延迟检查点结束标志的缺失重新初始化（suyu.cmj）

  这可能导致不必要的检查点延迟，或在启用断言的构建中出现断言失败。

- 避免在已中止的事务中通过扩展查询协议处理空语句时出现断言失败（Tom Lane）

- 修复 `contrib/fuzzystrmatch` 的 Soundex `difference()` 函数以合理处理空输入（Alexander Lakhin，Tom Lane）

  不包含字母字符的输入字符串会产生不可预测的输出。

- 加强 `contrib/hstore` 输入中的空白检查（Evan Jones）

  在某些情况下，字符会被错误地识别为空白从而被丢弃。

- 拒绝 `contrib/intarray` 的 `gist__int_ops` 索引操作符类的超大输入数组（Ankit Kumar Pandey，Alexander Lakhin）

  之前此代码会报告一个 `NOTICE` 但仍然继续执行，创建一个无效的索引条目，在读取索引时可能导致崩溃。

- 避免 `contrib/intarray` 中 GiST 索引条目的无用双重解压（Konstantin Knizhnik，Matthias van de Meent，Tom Lane）
- 确保关系缓存条目中的 `pg_index.indisreplident` 保持最新（Shruthi Gowda）

  此值在某些情况下可能是过期的。核心代码没有依赖关系缓存的此副本，因此就 PostgreSQL 本身而言这只是一个潜在缺陷；但对于某些扩展来说可能是一个实际缺陷。
- 消除使用 OpenSSL 3.0.0 或更新版本编译时的弃用警告（Peter Eisentraut）

E.3. 发布版本 11.20

发布日期：. 2023-05-11

本次发布包含来自 11.19 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

PostgreSQL 社区将于 2023 年 11 月停止发布 11.X 系列的更新。建议用户尽快升级到较新的发布分支。

E.3.1. 迁移到版本 11.20

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.14 的版本升级，请参见第 E.9 节。

E.3.2. 变更

- 防止 `CREATE SCHEMA` 绕过 `search_path` 的更改（Alexander Lakhin）

  在 `CREATE SCHEMA` 命令中，当前 `search_path` 中的对象以及新创建模式中的对象，即使在调用了试图设置安全 `search_path` 的函数或脚本中也是可见的。这可能允许任何有权创建模式的用户劫持安全定义函数或扩展脚本的权限。

  PostgreSQL 项目感谢 Alexander Lakhin 报告此问题。（CVE-2023-2454）
- 在内联集合返回函数后正确执行行级安全策略（Stephen Frost，Tom Lane）

  如果一个集合返回的 SQL 语言函数引用了具有行级安全策略的表，且该函数可以被内联到调用查询中，在某些涉及在不同角色下重用缓存计划的情况下，这些 RLS 策略将不会被正确执行。这可能允许用户查看或修改本应不可见的行。

  PostgreSQL 项目感谢 Wolfgang Walther 报告此问题。（CVE-2023-2455）
- 避免在 `CREATE SCHEMA` 中省略新模式名称时发生崩溃（Michael Paquier）

  SQL 标准允许编写 `CREATE SCHEMA AUTHORIZATION` *`owner_name`*，其中模式名称默认为 *`owner_name`*。然而某些代码路径期望模式名称存在，因此会失败。
- 禁止修改存储在索引中的复合类型（Tom Lane）

  `ALTER TYPE` 不允许对存储在任何表列中的复合类型进行非二进制兼容的修改。（也许将来会允许，但目前还没有实现；重写大量表的锁定影响令人生畏。）我们忽略了索引可能包含不在其所属表中出现的复合类型的情况。
- 确保对启用了 RLS 的父表执行 `COPY TO` 时不会从子表中复制任何行（Antonin Houska）

文档明确说明 `COPY TO` 仅从指定的表中复制行，不包括它可能拥有的任何继承子表。然而，如果该表启用了行级安全，则此行为将不再成立。

- 避免在向 `array_position()` 或 `array_positions()` 传递空数组时可能发生的崩溃（Tom Lane）
- 修复 `to_char()` 中可能的越界读取问题（Tom Lane）

  运气不好的情况下，这可能导致服务器崩溃。
- 避免 `translate()` 函数中的缓冲区越界读取（Daniil Anisimov）

  使用删除功能时，该函数可能会读取输入字符串之后的一个字节，从而产生少量崩溃风险。
- 修复 JSON 字符串字面量中解析错误时的错误光标设置（Tom Lane）

  在 JSON 值中的字符串字面量中检测到语法错误的大多数情况下，未能正确设置错误光标。这至少会导致无用的错误消息（指向字符串之前的标记，而不是实际的问题位置），在 v14 及更高版本中甚至可能导致崩溃。
- 修复解析器在某些不正确嵌套聚合的情况下未能检测到的问题（Tom Lane）

  这一疏忽可能导致执行器对本应作为无效查询被拒绝的查询执行失败。
- 修复解析序列 `SEQUENCE NAME` 选项时的数据结构损坏（David Rowley）

  如果事件触发器捕获了损坏的解析树，可能会导致问题。
- 在将初始计划从一个节点移动到另一个节点时，正确更新计划节点的并行安全标记（Tom Lane）

  这一规划器疏忽可能导致运行时出现 “subplan was not initialized” 错误。
- 当调用包含子查询时，禁用窗口聚合的逆转换优化（David Rowley）

  此优化要求聚合的参数表达式具有可重复的结果，而子查询可能不满足此条件。
- 修复嵌套 `ARRAY[]` 构造在执行中的疏忽（Alexander Lakhin，Tom Lane）

  正确检测结果数组所需总空间的溢出，避免因输出分配不足而可能导致的崩溃。同时确保结果数组中任何尾部填充空间被清零；虽然在大多数情况下留下垃圾数据是无害的，但可能会导致后续出现奇怪的行为。
- 修复布尔列分区剪枝逻辑（David Rowley）

  使用 `boolcol IS NOT TRUE` 条件进行裁剪时处理不正确，可能导致不返回 `boolcol` 为 NULL 的行。此外，按 `NOT boolcol` 进行分区这种不太可能的情况也处理不正确。
- 修复并行哈希连接中每批次清理时的竞态条件（Thomas Munro，Melanie Plageman）

  在时机不巧且 `parallel_leader_participation` = `off`（非默认值）的情况下，可能发生崩溃。
- 当表具有值为零的每表 `vacuum_cost_delay` 设置时，不进行 vacuum 代价延迟均衡（Masahiko Sawada）

延迟均衡应该在 autovacuum 处理具有每表 `vacuum_cost_delay` 设置的表时被禁用，但这仅对正值生效，而不对零值生效。

- 修复在视图末尾添加列时的极端情况崩溃（Tom Lane）
- 修复分区更新中 MULTIEXPR_SUBLINK 子计划的罕见失败（Andres Freund，Tom Lane）

  使用语法 `INSERT ... ON CONFLICT DO UPDATE SET (c1, ...) = (SELECT ...)` 并以分区表作为目标表时，如果任何子表与父表不一致（例如，物理列顺序不同），可能导致失败。这通常表现为执行器中一致性检查的失败；但也可能导致崩溃或不正确的数据更新。
- 修复在具有 `DO ALSO INSERT ... SELECT` 规则的视图上执行多行 `INSERT ... VALUES` 查询时 `DEFAULT` 标记的处理（Dean Rasheed）

  此类情况通常会出现 “unrecognized node type” 错误或断言失败。
- 支持在规则动作的子查询中引用 `OLD` 和 `NEW`（Dean Rasheed，Tom Lane）

  此类引用实际上是横向引用，但如果子查询没有显式标记 `LATERAL`，服务器可能会崩溃。安排在必要时隐式添加该标记。
- 在反编译包含 `WITH` 中 `INSERT`/`UPDATE`/`DELETE` 的规则或 SQL 函数体时，注意打印目标表的正确别名（Tom Lane）
- 修复 `SERIALIZABLE READ ONLY` 优化中的问题（Thomas Munro）

  已标记为 “doomed” 的事务干扰了 `SERIALIZABLE READ ONLY` 事务的安全快照优化。在某些情况下该优化被不必要地跳过。在其他情况下会发生断言失败（但在非断言构建中没有问题）。
- 避免 `pgoutput` 逻辑解码插件中的缓存回调槽泄漏（Shi Yu）

  在单个会话中多次启动和关闭该插件最终会导致 “out of relcache_callback_list slots” 错误。
- 修复在缓冲构建 GiST 索引期间对悬空指针的解引用（Alexander Lakhin）

  此错误在生产构建中通常是无害的，因为获取的值并不关键；但原则上可能导致服务器崩溃。
- 逻辑复制更新或删除操作时忽略已删除列（Onder Kalaci，Shi Yu）

  使用 `REPLICA IDENTITY FULL` 选项进行复制时，如果表包含此类列则会失败。
- 支持使用 SCRAM-SHA-256 通道绑定的 RSA-PSS 证书（Jacob Champion，Heikki Linnakangas）

  此功能需要使用 OpenSSL 1.1.1 或更新版本构建。服务器和 libpq 都受到影响。
- 避免在 Windows 上进程 ID 跟踪的竞态条件（Thomas Munro）

  操作系统可能在 postmaster 观察到子进程已退出之前回收其 PID。这可能导致使用相同的 PID 跟踪多个子进程，从而造成混乱。
- 为 `SPI_result_code_string()` 添加缺失的分支（Dean Rasheed）
- 修复 `AllocSetRealloc()` 中错误的 Valgrind 标记（Karina Litskevich）

在大型（>8kB）palloc 块的大小被减小这种不常见的情况下，启用 Valgrind 的构建会错误标记从块中释放的内存的已定义状态，可能在 Valgrind 测试期间导致不正确的结果。

- 避免在解码事务性逻辑复制消息时发生断言失败（Tomas Vondra）
- 避免在处理正则表达式转义时的区域设置敏感性（Jeff Davis）

  反斜杠后跟非 ASCII 字符有时会导致断言失败，具体取决于当前的区域设置。

- 避免在 `log_newpage_range()` 中当指定范围内最后几个页面为空时尝试写入空 WAL 记录（Matthias van de Meent）

  目前尚不完全清楚此情况在已发布的分支中是否可达，但如果可达则可能发生断言失败。

- 加强将 Perl 列表结构转换为多维 SQL 数组时的数组维度检查（Tom Lane）

  当子列表的嵌套不一致，使得数据不能表示值的矩形数组时，plperl 可能会出现异常行为。此类情况现在会产生错误，但之前可能导致崩溃或产生垃圾输出。

- 加强将 Python 列表结构转换为多维 SQL 数组时的数组维度检查（Tom Lane）

  在处理空子列表，或子列表嵌套不一致使得数据不能表示值的矩形数组时，plpython 可能会出现异常行为。前者应该产生空输出数组，后者应产生错误。但某些情况下会导致崩溃，另一些情况下会产生意外输出。

- 修复 plpython 中异常栈的展开（Xing Guo）

  某些罕见的失败情况可能在未清理 PG_TRY 异常栈的情况下返回，如果在下一个栈层级被展开之前引发另一个错误，则有崩溃风险。

- 修复使用 `-C ORACLE` 选项构建的 ecpg 程序中可能发生的数据损坏（Kyotaro Horiguchi）

  当 `ecpg_get_data()` 被调用时 `varcharsize` 设置为零，它可能会将终止零字符写入前一个字段的最后一个字节，从而截断该字段中的数据。

- 修复 pg_dump，使得在枚举类型列上进行 hash 分区的分区表可以成功恢复（Tom Lane）

  由于枚举值的 hash 码依赖于分配给枚举的 OID，在导出和恢复后它们通常是不同的，这意味着行往往需要进入与原来不同的分区。用户可以通过指定 `--load-via-partition-root` 选项来解决此问题；但由于不使用该选项几乎不可能成功，因此让 pg_dump 对此类表自动应用该选项。

  同时，修复 pg_restore，使其在使用 `--load-via-partition-root` 模式时不再尝试在恢复前 `TRUNCATE` 目标表。这避免了死锁和数据丢失的风险。

- 在 `contrib/hstore_plpython` 中，如果要转换的 Python 值不是映射类型则避免崩溃（Dmitry Dolgov，Tom Lane）

  这应该产生一个错误，但 Python 3 更改了一些 API，导致检查出现异常行为，从而引发崩溃。

- 修复 `contrib/pg_trgm` 在处理不可满足的正则表达式时的异常行为（Tom Lane）

  如 `$foo` 这样的正则表达式是合法的但不可满足；正则表达式编译器会识别到这一点并生成一个空的 NFA 图。尝试将这样的图优化为 pg_trgm GIN 或 GiST 索引限定条件时，会导致访问工作数组末尾之外的内容，可能导致崩溃。

- 使用与 GNU 兼容的 strip 剥离静态库中的符号时，使用 `--strip-unneeded` 选项（Tom Lane）

  之前，`make install-strip` 在这种情况下使用 `-x` 选项。此更改避免了 llvm-strip 的异常行为，同时也能产生稍小的输出。

- 停止推荐自动下载 DTD 文件用于构建文档，并实际禁用该功能（Aleksander Alekseev，Peter Eisentraut，Tom Lane）

  现在似乎不再可能在没有本地安装 DocBook DTD 文件的情况下构建 SGML 文档。之前 xsltproc 可以从 sourceforge.net 即时下载这些文件；但 sourceforge.net 现在只允许 HTTPS 访问，而没有常见版本的 xsltproc 支持该协议。因此，移除了我们文档中建议这种做法是可能或有用的部分，改为在构建配方中添加 xsltproc 的 `--nonet` 选项。

- 在 PGXS 构建中运行 TAP 测试时，为临时 `portlock` 目录使用更合理的位置（Peter Eisentraut）

  将其放置在构建目录下的 `tmp_check` 中。使用之前的代码，PGXS 构建会尝试将其放置在安装目录中，而该目录不一定是可写的。

- 将时区数据文件更新到 tzdata 2023c 版，以反映埃及、格陵兰、摩洛哥和巴勒斯坦的夏令时法规变更。

  在使用莫斯科时间时，Europe/Kirov 和 Europe/Volgograd 现在使用缩写 MSK/MSD 而不是数字缩写，以与其他使用莫斯科时间的时区保持一致。此外，America/Yellowknife 不再与 America/Edmonton 不同；这影响该地区 1948 年之前的一些时间戳。

E.4. 发布版本 11.19

发布日期：. 2023-02-09

本次发布包含来自 11.18 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

PostgreSQL 社区将于 2023 年 11 月停止发布 11.X 系列的更新。建议用户尽快升级到较新的发布分支。

E.4.1. 迁移到版本 11.19

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.14 的版本升级，请参见第 E.9 节。

E.4.2. 变更

- 允许在尚未（完全）有效的索引上设置 `REPLICA IDENTITY`（Tom Lane）

  当 pg_dump 导出一个标记了 `REPLICA IDENTITY` 的分区索引时，它生成的命令序列会在分区索引被标记为有效之前应用 `REPLICA IDENTITY`，导致恢复失败。似乎没有很好的理由禁止按此顺序操作，因此允许这样做。在索引变为有效之前，该标记不会产生任何效果。

- 修复在规则中从多行 `VALUES` 列表执行 `INSERT` 时对 `DEFAULT` 标记的处理（Dean Rasheed）

  在某些情况下，`DEFAULT` 标记不会被替换为正确的默认值表达式，导致 “unrecognized node type” 错误。

- 修复并行哈希连接在边界情况下发生的数据损坏（Dmitry Astapov）

  如果正在写入临时文件的大元组的最后一个块恰好是 32760 字节，由于栅栏柱错误会导致数据损坏。查询通常会在稍后以数据损坏的症状而失败。

- 遵循 `checkpoint_completion_target` 的非默认设置（Bharath Rupireddy）

  在 `checkpoint_completion_target` 更改后，内部状态未被更新，可能导致检查点 I/O 执行得比预期更快或更慢，特别是在动态更改该设置时。

- 在 `recovery_target_xid` 模式下记录正确的结束时间戳（Tom Lane）

  当基于 `recovery_target_xid` 设置且 `recovery_target_inclusive` = `off` 结束恢复时，我们在“recovery stopping before ... transaction”日志消息中打印了一个不正确的时间戳（始终为 2000-01-01）。

- 在扩展查询协议中，如果我们正在运行管道，避免在 `ANALYZE` 之后立即提交（Tom Lane）

  如果没有显式的 `BEGIN TRANSACTION`，`ANALYZE` 会自行提交，但这在管道化的命令序列中不应该发生。

- 拒绝长度错误的取消请求包（Andrey Borodin）

  即使取消请求的长度字段过小，服务器也会处理该请求。这导致读取超出已分配缓冲区的末尾。理论上这可能导致段错误，但在实际中不太可能发生，因为缓冲区需要非常接近内存末尾才会出现。更可能的结果是关于错误后端 PID 或取消码的虚假日志消息。改为对错误的长度发出报告。

- 在子查询提升中增加递归和循环防护（Tom Lane）

  一个精心构造的查询可能导致深度递归和过多的时间用于尝试展平子查询。一个正确的修复对于后移来说过于侵入性，但至少我们可以增加堆栈深度检查和中断检查以允许取消查询。

- 修复按分区连接代码以容忍无法为每个分区生成计划的情况（Tom Lane）

  这可能导致“could not devise a query plan for the given query”错误。

- 限制 `get_actual_variable_range` 执行的清理工作量（Simon Riggs）

  在删除大量出现在索引末尾的元组之后立即进行的规划器运行，可能会花费大量工作来设置那些索引条目的“killed”位。通过在检查 100 个堆页面后放弃此过程来限制任何一次查询中的工作量。所有清理工作最终仍会完成，但不会出现如此大的性能波动。

- 确保在执行短语匹配时可以取消全文检索查询的执行（Tom Lane）

- 在复制连接尝试失败后清理 libpq 连接对象（Andres Freund）

  之前的代码泄漏了连接对象。在后台代码路径中这基本无害，因为调用进程会放弃并退出。但在诸如 `CREATE SUBSCRIPTION` 等命令中，此类失败会导致少量会话生命周期的内存泄漏。

- 在热备服务器中，减少跟踪主库上已知活跃 XID 的处理开销（Simon Riggs，Michail Nikolaev）

  对 KnownAssignedXids 数组清理不够积极可能导致性能下降，特别是在备库上 `max_connections` 设置较大时。

- 修复逻辑解码中未初始化内存的使用（Masahiko Sawada）

在某些情况下，逻辑解码的恢复可能会尝试重用已经释放的 XID 数据，导致不可预测的行为。

- 避免在 WAL 重放 hash 索引页面分裂操作期间出现罕见的 “failed to acquire cleanup lock” 恐慌（Robert Haas）
- 在 WAL 重放期间设置堆页面的 all-visible 位时推进页面的 LSN（Jeff Davis）

  未能这样做会导致备库上的页面可能与主库不同，并违反了关于 LSN 何时更改的其他预期。就 PostgreSQL 本身而言，这似乎只是一个理论上的隐患，但可能影响第三方工具。
- 防止不安全地使用关系缓存条目的 `rd_smgr` 指针（Amul Sul）

  移除各种假设 `rd_smgr` 在一系列操作中始终有效的假定，通过将所有使用包装在一个函数中，在需要时重新计算。这防止了在一系列操作中途发生意外缓存刷新时出现的错误。
- 修复 `WaitEventSet` 逻辑中的潜在缓冲区溢出问题（Thomas Munro）

  基于 `epoll` 和 `kqueue` 的实现，在其内部缓冲区大小与调用者输出缓冲区大小不同时，可能向内核请求过多的事件。该情况在已发布的 PostgreSQL 版本中尚未出现，但此错误对外部模块和未来的错误修复构成隐患。
- 避免在 32 位构建中访问共享内存时出现名义上的未定义行为（Andres Freund）

  clang 的未定义行为检测器对使用对齐不足的指针发出了警告。这在非调试构建中不太可能造成问题，但为了测试目的值得修复。
- 修复 ACL 检查中缓存查找失败消息的复制粘贴错误（Justin Pryzby）

  原则上这些错误应该永远不会触发。但如果触发了，其中一些会报告错误类型的对象。
- 在 pg_dump 中，避免在获取要检查的表的锁之前调用不安全的服务器函数（Tom Lane，Gilles Darold）

  pg_dump 使用某些服务器函数，如果检查的表被并发删除则可能失败。通过确保在深入查询表的属性之前获取访问共享锁，以及不对我们不打算导出的表应用此类函数，来避免此类失败。
- 修复 `ALTER FUNCTION/PROCEDURE/ROUTINE ... SET SCHEMA` 的 Tab 补全（Dean Rasheed）
- 修复 `contrib/seg` 在输入数字超过 127 位时不再崩溃或输出垃圾数据（Tom Lane）
- 在 `contrib/sepgsql` 中，避免较新版本的 libselinux 产生弃用警告（Michael Paquier）
- 修复在使用 Strawberry Perl 时用 MSVC 构建 PL/Perl 的编译失败问题（Andrew Dunstan）
- 修复用 MSVC 构建的 PL/Perl 与用 gcc 构建的 Perl 库之间的不匹配问题（Andrew Dunstan）

  此类组合之前可能导致 “loadable library and perl binaries are mismatched” 错误。
- 抑制来自 Perl 头文件的编译器警告（Andres Freund）

  我们首选的编译器选项会对 Perl 头文件最近版本中出现的构造引发警告。在使用 gcc 时，我们可以用 pragma 来抑制这些警告。

- 修复 pg_waldump 使其在不丢弃未使用的 static-inline 函数的编译器上可以构建（Tom Lane）
- 将时区数据文件更新到 tzdata 2022g 版，以反映格陵兰和墨西哥的夏令时法规变更，并修正加拿大北部、哥伦比亚和新加坡的历史数据。

  值得注意的是，一个新的时区 America/Ciudad_Juarez 从 America/Ojinaga 中分离出来。

E.5. 发布版本 11.18

发布日期：. 2022-11-10

本次发布包含来自 11.17 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.5.1. 迁移到版本 11.18

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.14 的版本升级，请参见第 E.9 节。

E.5.2. 变更

- 修复 `VACUUM`，使其在尝试删除 B-树索引页却找不到父页面中指向该页的下行链接时继续处理（Peter Geoghegan）

  现在只记录此问题并继续处理，不删除该空页面，而不再抛出错误。以前，有缺陷的操作符类或损坏的索引可能使索引清理一直无法完成，最终导致事务回卷问题。
- 修复在可更新视图的 `INSERT` 的多行 `VALUES` 子句中出现的 `DEFAULT` 令牌的处理（Tom Lane）

  此疏忽可能导致 “cache lookup failed for type” 错误，或在较旧的分支中甚至导致崩溃。
- 不允许名为 `_RETURN` 但不是 `ON SELECT` 的规则（Tom Lane）

  这避免了视图的 `ON SELECT` 规则与其可能拥有的任何其他规则之间的混淆。
- 修复继承更新中 MULTIEXPR_SUBLINK 子计划的罕见失败（Tom Lane）

  对继承表或分区表使用 `UPDATE tab SET (c1, ...) = (SELECT ...)` 语法时，如果子表之间的差异足够大，就可能失败。这通常表现为执行器中的一致性检查失败，但也可能导致崩溃或错误的数据更新。
- 修复在创建分区索引时索引表达式和谓词的错误匹配（Richard Guo，Tom Lane）

  在创建分区索引时，我们尝试识别分区上与分区索引匹配的任何现有索引，以便将它们作为子索引吸收而不是构建新的索引。表达式匹配处理不正确，因此可用的子索引可能被忽略，导致创建重复的索引。
- 当外层查询包含分组集时，避免展平没有 `FROM` 的子查询（Tom Lane）

  这一疏漏可能导致断言失败，或 “variable not found in subplan target list” 等规划器错误。
- 防止备库提升后 WAL 损坏（Dilip Kumar，Robert Haas）

当一个执行归档恢复（但未使用备库模式）的 PostgreSQL 实例被提升，且其尝试读取的最后一个 WAL 段以部分记录结尾时，该实例将在新时间线上写入无效的 WAL 段。

- 修复 GIN 索引快速插入路径中 WAL 操作的错误排序（Matthias van de Meent，Zhang Mingli）

  此错误在核心 PostgreSQL 中没有已知的负面后果，但确实导致了一些扩展的问题。

- 修复在重放起始点位于事务开始和其子事务开始之间时逻辑解码中的错误（Masahiko Sawada，Kuroda Hayato）

  这些错误可能导致调试构建中的断言失败，否则会导致内存泄漏。

- 防止在逻辑解码期间使用错误的快照检查系统目录（Masahiko Sawada）

  如果解码从修改系统目录的事务中途开始，解码器可能无法识别这一点，导致未能将该事务视为进行中的事务来进行目录查找。

- 在逻辑解码的更多位置接受中断（Amit Kapila，Masahiko Sawada）

  这改善了复制工作进程缓慢关闭的问题。

- 避免在复制工作进程中函数语法错误后崩溃（Maxim Orlov，Anton Melnikov，Masahiko Sawada，Tom Lane）

  如果在逻辑复制工作进程中执行的 SQL 语言或 PL/pgSQL 语言 `CREATE FUNCTION` 或 `DO` 命令中发生语法错误，工作进程会因空指针解引用或断言失败而崩溃。

- 修复传递给 SQL 函数的读写扩展数据的处理（Tom Lane）

  如果一个非内联 SQL 函数在多个位置使用一个参数，且其中一个函数期望能够就地修改读写数据，则后续使用该参数时会观察到错误的值。（在核心 PostgreSQL 中，扩展数据机制仅用于数组和复合类型值；但扩展可能将其用于其他结构化类型。）

- 在 Snowball 词典中，不尝试对过长的单词进行词干提取（Olly Betts，Tom Lane）

  如果输入单词超过 1000 字节，在大小写折叠后直接返回，而不是尝试通过 Snowball 代码处理。此限制防止了土耳其语词干提取器中已知的递归导致堆栈溢出的问题，并且可以作为对 Snowball 词干提取器中可能存在的其他安全或性能问题的保障。如此长的字符串肯定不是任何人类语言中的单词，因此词干提取器对其也不太可能做出任何有意义的处理。

- 修复字符串比较中的释放后使用隐患（Tom Lane）

  字符串比较函数中的不当内存管理可能导致在已不再分配的缓冲区上写入，可能破坏正在使用该内存的其他内容。这只会在相当长的字符串（超过 1kB）且使用 ICU 排序规则时才会发生。

- 防止共享内存状态损坏时 postmaster 崩溃（Tom Lane）

  postmaster 进程应该在共享内存损坏时存活并启动数据库重启，但有一段代码在这方面不够谨慎。

- 增加更多防止递归至堆栈溢出的防护措施（Richard Guo，Tom Lane）

- 避免自动清理启动器进程中的长期内存泄漏（Reid Thompson）

缺乏用户报告表明此问题在 v15 之前的分支中只是潜在的；但原因不太清楚，因此仍然后移了此修复。

- 改进 PL/pgSQL 处理声明为 `RECORD` 的参数的能力（Tom Lane）

  在会话期间为传递给 `RECORD` 参数的每个具体类型构建单独的函数缓存条目，与我们对多态参数所做的类似。这使一些此前会失败并报告 “type of parameter does not match that when preparing the plan” 的用法正常工作。

- 在 libpq 中为 `NULL` 连接指针增加缺失的保护检查（Daniele Varrazzo，Tom Lane）

  有一个惯例是 libpq 函数应检查 NULL 的 PGconn 参数，并优雅地失败而不是崩溃。`PQflush()` 和 `PQisnonblocking()` 没有遵循这个惯例，因此予以修复。

- 在 ecpg 中，修复在同一声明中声明多个 `varchar` 或 `bytea` 变量时遗漏变量存储类的问题（Andrey Sokolov）

  例如，ecpg 在翻译 `static varchar str1[10], str2[20], str3[30];` 时，只有 `str1` 被标记为 `static`。

- 允许在 pg_basebackup 中进行跨平台表空间重定位（Robert Haas）

  允许 `--tablespace-mapping` 中的远程路径为 Unix 风格或 Windows 风格的绝对路径，因为源服务器可能与本地系统运行在不同的操作系统上。

- 在 pg_stat_statements 中，修复对已释放内存的访问（zhaoqigui）

  当 pg_stat_statements 跟踪通过扩展查询协议发出的 `ROLLBACK` 命令时会出现此问题。在调试构建中，这始终导致断言失败。在生产构建中，通常没有可见的不良影响；但如果释放的内存已经被重用，很可能的结果是存储查询字符串的垃圾数据。

- 在 postgres_fdw 中，确保为 EvalPlanQual 计划构建的目标列表包含所有必需的列（Richard Guo，Etsuro Fujita）

  这避免了在罕见情况下出现 “variable not found in subplan target list” 错误。

- 拒绝平台的 `uuid_create()` 函数产生的意外输出（Nazir Bilal Yavuz）

  uuid-ossp 模块期望 libc 的 `uuid_create()` 产生版本 1 的 UUID，但最近的 NetBSD 版本改为产生版本 4（随机）的 UUID。检查这一点，如果是则抛出错误。删除文档中关于 NetBSD 实现可用于 uuid-ossp 的声明。（如果版本 4 的 UUID 对你的目的来说没问题，你根本不需要 uuid-ossp；只需使用 `gen_random_uuid()` 即可。）

- 将新的 Perl 测试模块包含在标准安装中（Álvaro Herrera）

  在 15 之前的版本分支中将 `PostgreSQL/Test/Cluster.pm` 和 `PostgreSQL/Test/Utils.pm` 添加到标准安装文件集中。这是为了方便那些希望在旧版本分支中使用新编写的测试代码的扩展。

- 在 NetBSD 上，强制在 postmaster 启动时解析动态符号（Andres Freund，Tom Lane）

  这避免了 NetBSD 10 上动态链接器中的死锁风险。

- 修复与 LLVM 15 的不兼容问题（Thomas Munro，Andres Freund）

- 允许在任何机器上使用 `__sync_lock_test_and_set()` 实现自旋锁（Tom Lane）

这简化了向新机器架构的移植，至少在使用支持此 GCC 内置函数的编译器时。

- 将符号 `REF` 重命名为 `REF_P` 以避免在最近的 macOS 上编译失败（Tom Lane）
- 消除 clang 15 及更高版本的各种编译器警告（Tom Lane）
- 将时区数据文件更新到 tzdata 2022f 版，以反映智利、斐济、伊朗、约旦、墨西哥、巴勒斯坦和叙利亚的夏令时法规变更，并修正智利、克里米亚、伊朗和墨西哥的历史数据。

  此外，Europe/Kiev 时区已重命名为 Europe/Kyiv。另外，以下时区已合并到附近的、人口更多的、自 1970 年以来时钟一致的时区中：Antarctica/Vostok、Asia/Brunei、Asia/Kuala_Lumpur、Atlantic/Reykjavik、Europe/Amsterdam、Europe/Copenhagen、Europe/Luxembourg、Europe/Monaco、Europe/Oslo、Europe/Stockholm、Indian/Christmas、Indian/Cocos、Indian/Kerguelen、Indian/Mahe、Indian/Reunion、Pacific/Chuuk、Pacific/Funafuti、Pacific/Majuro、Pacific/Pohnpei、Pacific/Wake 和 Pacific/Wallis。（这间接影响了已经是这些时区别名的时区：Arctic/Longyearbyen、Atlantic/Jan_Mayen、Iceland、Pacific/Ponape、Pacific/Truk 和 Pacific/Yap。）America/Nipigon、America/Rainy_River、America/Thunder_Bay、Europe/Uzhgorod 和 Europe/Zaporozhye 也被合并到附近的时区中，因为发现它们声称的 1970 年后与这些时区的差异似乎是错误的。在所有这些情况下，之前的时区名称仍作为别名保留；但实际数据是所合并到的时区的数据。

  这些时区合并导致被合并时区的 1970 年前时区历史丢失，这对于期望 `timestamptz` 显示一致性的应用程序可能造成困扰。例如，存储的值 `1944-06-01 12:00 UTC` 在选择 Europe/Stockholm 时区时，之前会显示为 `1944-06-01 13:00:00+01`，但现在会显示为 `1944-06-01 14:00:00+02`。

  可以使用选项构建时区数据文件来恢复旧的时区数据，但该选择也会插入大量其他旧的（且通常缺乏充分证据的）时区数据，导致相比接受这些上游更改，总变化更多。PostgreSQL 选择按照推荐方式发布 tzdb 数据，据我们所知，大多数主要操作系统发行版也在这样做。但是，如果这些更改给你的应用程序造成了严重问题，一个可能的解决方案是使用 tzdb 的向后兼容选项安装本地构建的时区数据文件（参见其 `PACKRATDATA` 和 `PACKRATLIST` 选项）。

E.6. 发布版本 11.17

发布日期：. 2022-08-11

本次发布包含来自 11.16 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.6.1. 迁移到版本 11.17

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.14 的版本升级，请参见第 E.9 节。

E.6.2. 变更

- 不允许扩展脚本替换不属于该扩展的已有对象（Tom Lane）

  此更改防止扩展脚本在存在不属于该扩展的现有对象时执行 `CREATE OR REPLACE`。在相同情况下，也防止执行 `CREATE IF NOT EXISTS`。这防止了一种木马攻击形式，即恶意数据库用户可以成为扩展对象的所有者，然后修改它以危害其他用户对该对象的未来使用。作为附带的好处，它还降低了意外替换非预期对象的风险。

PostgreSQL 项目感谢 Sven Klemm 报告此问题。（CVE-2022-2625）

- 修复在备库上重放 `CREATE DATABASE` WAL 记录的问题（Kyotaro Horiguchi，Asim R Praveen，Paul Guo）

  备库在重放数据库创建 WAL 记录时可能遇到缺失的表空间目录。在此补丁之前，备库在这种情况下会恢复失败；然而，此类目录可能是合法缺失的。创建表空间（作为普通目录），然后在重放达到一致状态后检查它是否已被删除。

- 支持“原位”表空间（Thomas Munro，Michael Paquier，Álvaro Herrera）

  通常，PostgreSQL 表空间是指向其他文件系统中某个目录的符号链接。此更改允许表空间直接就是一个普通目录。虽然这对将表分布到不同文件系统上没有帮助，但对测试来说是一种方便的设置。此外，为支持 `CREATE DATABASE` 的重放修复，这一能力是必需的；该修复会将缺失的表空间临时创建为“原位”表空间。

- 修复 `CREATE INDEX` 中的权限检查（Nathan Bossart，Noah Misch）

  CVE-2022-1552 的修复导致 `CREATE INDEX` 在查找操作符类和其他对象时应用表所有者的权限，而之前使用的是调用用户的权限。这破坏了导出/恢复场景，因为 pg_dump 在重新授予权限之前发出 `CREATE INDEX`。

- 在扩展查询协议中，在 `CREATE DATABASE` 和其他不能在事务块中运行的命令之后强制立即提交（Tom Lane）

  如果客户端在此类命令之后没有立即发送 Sync 消息，而是发送了另一个命令，那么该命令中的任何失败都将导致回滚前面的命令，通常会在磁盘上留下不一致的状态（如缺失或多余的数据库目录）。用于防止这种情况的机制在简单查询消息中的多个命令中有效，但在一系列扩展协议消息中无效。为了防止不一致而不破坏当前可行的使用场景，在此类命令之后强制执行隐式提交。

- 修复检查事务可见性时的竞态条件（Simon Riggs）

  `TransactionIdIsInProgress` 可能在目标事务被认为可见之前就报告 `false`，导致各种异常行为。竞态条件窗口通常非常窄，但使用同步复制会使其大幅扩大，因为等待同步副本的过程发生在该窗口中。

- 修复当“整行变量”引用返回复合类型域的函数结果时的查询问题（Tom Lane）

- 修复在提升引用了 `GROUPING` 函数的子 `SELECT` 时出现的“variable not found in subplan target list”规划器错误（Richard Guo）

- 修复 `ALTER TABLE ... ENABLE/DISABLE TRIGGER` 对分区表上的触发器正确处理递归（Álvaro Herrera，Amit Langote）

  在某些情况下，由于命令会尝试调整没有该触发器的子分区上的触发器，会出现“trigger does not exist”错误。

- 防止 `pg_stat_get_subscription()` 可能返回包含垃圾值的额外行（Kuntal Ghosh）

- 确保 `pg_stop_backup()` 正确清理会话状态（Fujii Masao）

  此遗漏可能导致会话后续出现断言失败或崩溃。

- 修复 `FOR [KEY] UPDATE/SHARE` 子句中的连接别名匹配（Dean Rasheed）

在某些边界情况下，可能报告误导性的错误。

- 避免在 XMLTABLE 或 JSON_TABLE 结构上附加了过多列别名时崩溃（Álvaro Herrera）
- 拒绝 FROM 中列数过多的 ROW() 表达式和函数（Tom Lane）

  超过约 1600 列的情况不受支持，且在执行时始终会失败。然而，一些较早的代码在查询超过 32K 列时可能被驱动到断言失败或崩溃。增加一个解析时检查以防止此问题。
- 在反编译视图或规则时，如果 SELECT 输出列的 AS "?column?" 别名子句可能在其他地方被引用，则显示它（Tom Lane）

  之前，这个自动生成的别名总是被隐藏的；但在某些边界情况下，这样做会导致视图或规则定义无法恢复。
- 修复使用 FROM 中返回复合类型的函数的视图的导出问题，当视图创建后复合类型的列已被删除时（Tom Lane）

  此疏忽可能导致导出/恢复或 pg_upgrade 失败，因为导出的视图会有过多的函数列别名。
- 向事件触发器报告隐式创建的操作符族（Masahiko Sawada）

  如果 CREATE OPERATOR CLASS 导致隐式创建了一个操作符族，该对象不会被报告给应当捕获此类事件的事件触发器。
- 修复在备库提升期间有重启点正在运行时的控制文件更新（Kyotaro Horiguchi）

  之前，当重启点完成时，它可能会错误地更新控制文件中的最后检查点字段，如果服务器在下一次正常检查点完成之前崩溃，可能导致 PANIC 和重启失败。
- 防止在逻辑复制大事务期间触发备库的 wal_receiver_timeout（Wang Wei，Amit Kapila）

  如果主库上的一个大事务没有向备库发送数据（可能是因为它修改的表都没有被发布），备库可能会超时。通过确保在这种情况下定期发送保活消息来修复此问题。
- 不允许在逻辑复制 walsender 中执行嵌套备份操作（Fujii Masao）
- 修复逻辑复制订阅者中的内存泄漏（Hou Zhijie）
- 防止在读取无效时区缩写文件时发生打开文件泄漏（Kyotaro Horiguchi）

  这种情况可能导致无害的警告消息。
- 允许自定义服务器参数具有为 NULL 的简短描述（Steve Chavez）

  之前，虽然扩展可以选择创建这样的设置，但某些代码路径在处理它们时会崩溃。
- 修复 WAL 一致性检查逻辑以正确处理 BRIN_EVACUATE_PAGE 标志（Haiyang Wang）
- 修复共享 hash 表管理中的错误断言检查（Thomas Munro）
- 安排在 SPI_commit() 中的提交时错误后进行清理，而不是期望调用者来做（Peter Eisentraut，Tom Lane）

正确的清理很复杂，需要使用底层设施，因此已知的调用者没有一个做对了也不足为奇。这导致当 PL 过程执行 `COMMIT` 但发生失败（如延迟约束检查）时出现异常行为。为了改善情况，将 `SPI_commit()` 重新定义为启动一个新事务，使其等价于 `SPI_commit_and_chain()`，只是获得默认事务特性而不是保留先前事务的特性。为了使此更改在 API 层面具有一定透明性，将 `SPI_start_transaction()` 重新定义为空操作。所有已知的 `SPI_commit()` 调用者都会立即调用 `SPI_start_transaction()`，因此它们不会注意到任何变化。类似的说明适用于 `SPI_rollback()`。

另外修复了 PL/Python，它完全遗漏了对此类错误的任何处理，导致跳出 Python 解释器。据报告这会导致 Python 3.11 崩溃。较旧的 Python 版本会泄漏一些内存，但似乎在其他方面没有问题。

- 移除 libpq 中不恰当的 SSL 密钥文件所有权检查（Tom Lane）

  在之前的小版本中，我们将服务器对 SSL 私钥文件的权限检查规则复制到了 libpq 中。但我们不应该同时复制服务器的文件所有权检查。虽然在正常使用场景下这没有问题，但对于以 root 身份运行的客户端可能会导致意外的失败，在其他情况下也可能如此。

- 确保 ecpg 合理地报告服务器连接丢失（Tom Lane）

  对 libpq 生成的错误结果（例如连接丢失报告）的错误处理，会导致打印 “(null)” 而非有用的错误消息；在更早的版本中则会导致崩溃。

- 避免 ecpglib 在意外的操作顺序下出现核心转储（Tom Lane）

  某些操作如 `EXEC SQL PREPARE` 在建立任何数据库连接之前调用时会崩溃（而不是按预期报告错误）。

- 在 ecpglib 中，避免冗余的 `newlocale()` 调用（Noah Misch）

  在首次连接时每个进程分配一次 C 区域设置对象，而不是每次查询都创建和释放区域设置对象。这缓解了 AIX 上 libc 的内存泄漏问题，并且可能在所有平台上提供一些性能改善。

- 在 psql 的 `\watch` 命令中，使用 control-C 取消后回显换行符（Pavel Stehule）

  这防止 libedit（也可能是 libreadline）对光标所在列的位置产生混乱。

- 修复 `contrib/pg_stat_statements` 以避免在 32 位平台上处理非常大的查询文本文件时出现问题（Tom Lane）

- 确保 `contrib/postgres_fdw` 在发送 `regconfig` 和其他 `reg*` 类型的常量时使用正确的模式限定（Tom Lane）

- 在 Linux 上分配动态共享内存时阻塞信号（Thomas Munro）

  这避免了信号中断 `posix_fallocate()` 时的问题。

- 检测 `shm_open()` 的意外 `EEXIST` 错误（Thomas Munro）

  这避免了在 Solaris 上可能发生的崩溃。

- 调整 PL/Perl 测试用例使其能在 Perl 5.36 下工作（Dagfinn Ilmari Mannsåker）

- 避免在构建 PostgreSQL 时，当存在多个 OpenLDAP 安装时错误地使用过时的 libldap_r 库（Tom Lane）

E.7. 发布版本 11.16

发布日期：. 2022-05-12

本次发布包含来自 11.15 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.7.1. 迁移到版本 11.16

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.14 的版本升级，请参见第 E.9 节。

E.7.2. 变更

- 将更多操作限制在“安全受限操作”沙箱内（Sergey Shinderuk，Noah Misch）

  自动清理、CLUSTER、CREATE INDEX、REINDEX、REFRESH MATERIALIZED VIEW 和 pg_amcheck 激活“安全受限操作”保护机制的时间过晚，甚至在某些代码路径中根本没有激活。在数据库中拥有创建非临时对象权限的用户可以定义一个对象，当自动清理下次处理该对象时，或者当某个超级用户对其运行上述受影响的命令时，该对象会以超级用户权限执行任意 SQL 代码。

  PostgreSQL 项目感谢 Alexander Lakhin 报告此问题。（CVE-2022-1552）

- 停止将查询提供的列别名用于引用普通表的整行变量的列（Tom Lane）

  整行变量（例如在 SELECT 列表顶层以外的上下文中的 tbl.*）产生的元组中的列名，现在始终为关联的命名复合类型的列名（如果有的话）。之前我们曾尝试使其跟踪已应用于该变量所引用的 FROM 条目的任何列别名。但这在语义上是有问题的，因为这样变量的输出实际上根本不是它声称的复合类型。之前处理这种不一致性的尝试产生了糟糕的结果，甚至包括在磁盘上存储不可读的数据，因此放弃了整个想法。

  在需要重新标记这些列的情况下，一种变通方法是引入一个额外的子 SELECT 层级，使整行变量引用子 SELECT 的输出而非普通表。这样变量一开始就是 record 类型，就不存在这个问题了。

- 修复 table_to_xmlschema() 及相关函数中 timestamptz 和 timetz 类型的不正确输出（Renan Soares Lopes）

  这些类型的 xmlschema 输出包含了格式错误的正则表达式。

- 避免解析器在处理零列的 VALUES 子句时发生核心转储（Tom Lane）

- 修复引用外层查询级别的 GROUPING() 构造的规划器错误（Richard Guo，Tom Lane）

- 修复具有可返回列和不可返回列的索引上的仅索引扫描计划生成（Tom Lane）

  之前的代码会尝试读取不可返回列以及可返回列。这基本上是无害的，因为它实际上并没有对错误的值做任何处理，但它违反了最近添加的拒绝此类计划的错误检查。

- 修复正在执行重新排序的 IndexScan 节点中的查询生命周期内存泄漏（Aliaksandr Kalenik）

- 修复 ALTER FUNCTION 使其支持在同一命令中同时更改函数的并行属性及其 SET 变量列表（Tom Lane）

  如果同一命令还更新了函数的 SET 子句，则并行属性的更改会丢失。

- 修复使用前导键为表达式的索引进行 CLUSTER 时表行的错误排序（Peter Geoghegan，Thomas Munro）

  表会使用正确的数据重建，但其顺序与索引顺序关系不大。

- 修复删除分区索引时可能出现的死锁风险（Jimmy Yih，Gaurab Dey，Tom Lane）

  确保以标准顺序（先父后子，先表后索引）获取所需的表锁和索引锁。之前 DROP INDEX 的代码处理方式不同，因此可能与按标准顺序获取这些锁的并发查询发生死锁。

- 修复 DROP TABLESPACE 与检查点之间的竞态条件（Nathan Bossart）

  由 DROP TABLESPACE 强制执行的检查点有时无法从表空间目录中删除所有失效文件，导致虚假的 “tablespace is not empty” 错误。

- 修复与检查点重叠的 TRUNCATE 命令之后崩溃恢复中可能出现的问题（Kyotaro Horiguchi，Heikki Linnakangas，Robert Haas）

  TRUNCATE 必须确保在允许检查点完成之前，表的磁盘文件已被截断。否则，从该检查点开始的重放可能会在本应已移除的页面中发现意外数据，从而可能导致重放失败。

- 修复临时对象清理期间不安全的 toast 数据访问（Andres Freund）

  服务器进程退出期间的临时对象删除可能会失败并显示 “FATAL: cannot fetch toast data without an active snapshot” 。这通常是无害的，因为下次使用该临时模式时会成功清理。

- 修复当存在缺失的 WAL 续接记录时，备库提升期间出现的 “PANIC: xlog flush request is not satisfied” 失败（Sami Imseih）

- 修复热备冲突处理中可能出现的自死锁问题（Andres Freund）

  在不利的时序条件下，WAL 应用进程可能在等待其他进程释放缓冲区锁时卡住。

- 确保逻辑复制应用工作进程即使在达到 max_sync_workers_per_subscription 限制时也能重启（Amit Kapila）

  限制检查的错误编码导致重启的工作进程立即退出，使得工作进程数量少于应有的数量。

- 在更新的 WAL 日志中包含未更改的副本标识键列（如果它们以行外方式存储）（Dilip Kumar，Amit Kapila）

  否则订阅者无法看到这些值，并且将无法复制该更新。

- 改进逻辑复制订阅者针对不支持的关系种类所给出的错误消息（Tom Lane）

  v13 及更新版本的服务器支持发布分区表。更早的服务器版本无法订阅这类表，却给出极具误导性的错误消息：“table XYZ not found on publisher” 。现在改为给出更切合问题的消息。

- 禁止在 PL/Perl 函数编译期间执行 SPI 函数（Tom Lane）

  可以让 Perl 在 PL/Perl 函数编译期间执行用户定义的代码。但是，这类代码不应该尝试通过 SPI 调用 SQL 操作。那样做会导致崩溃，即使不崩溃也是一个安全隐患，因为我们确实不希望在函数验证期间执行代码。增加了一个检查以给出更友好的错误消息。

- 使 libpq 接受 root 拥有的 SSL 私钥文件（David Steele）

此变更使 libpq 对 SSL 密钥文件的安全所有权和权限规则与服务器自 9.6 版本以来使用的规则同步。即，在当前规则的基础上，还允许密钥文件由 root 拥有且权限为 `rw-r-----` 或更严格的情况。这对于密钥文件的系统级管理很有帮助。

- 使 pg_ctl 在等待停止/重启/提升操作时重新检查 postmaster 是否仍然存活（Tom Lane）

  pg_ctl 会在发送停止或提升信号时作为副作用验证 postmaster 是否存活，但之后它只是简单地等待磁盘上的状态变化。如果 postmaster 非正常终止且未删除其 PID 文件或更新控制文件，pg_ctl 将一直等到超时。现在改为每隔一段时间重新检查 postmaster 进程是否仍然存在。

- 修复 pg_waldump 中的错误处理（Kyotaro Horiguchi，Andres Freund）

  在尝试读取 WAL 文件以确定 WAL 段大小时，pg_waldump 会为文件过短的情况报告不正确的错误。此外，此错误消息及相关错误消息中报告的文件名可能是垃圾值。

- 确保 `contrib/pageinspect` 函数能够处理全零页面（Michael Paquier）

  这是一个合理的边界情况，但该模块大多未做准备。安排返回 null 或不返回行，视情况而定；这比抛出错误更有用。

- 在 `contrib/pageinspect` 中，增加对不正确页面“特殊空间”内容的防御，收紧对正确页面大小的检查，并增加一些缺失的索引类型正确性检查（Michael Paquier，Justin Pryzby，Julien Rouhaud）

  这些变更减少了该模块在处理错误数据时崩溃的可能性。

- 在 `contrib/postgres_fdw` 中，在请求远程排序查询之前验证 `ORDER BY` 子句是否可以安全传送，并在必要时包含 `USING` 子句（Ronan Dunklau）

  此修复防止了远程服务器可能按与我们意图不同的顺序排序的情况。虽然有时这只是表面问题，但如果远程数据被用作本地执行的合并连接的输入，则可能产生完全错误的结果。

- 更新 JIT 代码以支持 LLVM 14（Thomas Munro）

- 清理在 clang 的 `-fsanitize=undefined` 检查下出现的各种失败（Tom Lane，Andres Freund，Zhihong Yu）

  这些变更大多只是为了形式上符合 C 和 POSIX 标准的字面要求，不太可能对生产构建产生任何影响。

- 修复 PL/Perl 使其能在不支持嵌套在表达式中的语句的 C 编译器上构建（Tom Lane）

- 修复在 Windows 上不使用 MSVC 构建时 pg_dumpall 可能的构建失败（Andres Freund）

- 在 Windows 构建中，使用 gendef 代替 pexports 来构建 DEF 文件（Andrew Dunstan）

  这使构建过程适应了最近的 MSys 工具链。

- 防止在 MinGW 下构建的程序中对 shell 通配符模式进行额外展开（Andrew Dunstan）

出于某种原因，MinGW 提供的 C 库默认会展开程序命令行参数中的 shell 通配符字符。这令人困惑，尤其是因为在 MSVC 下不会发生这种情况，因此将其关闭。

- 将时区数据文件更新到 tzdata 2022a 版，以反映巴勒斯坦的夏令时法规变更，并修正智利和乌克兰的历史数据。

E.8. 发布版本 11.15

发布日期：. 2022-02-10

本次发布包含来自 11.14 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.8.1. 迁移到版本 11.15

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.14 的版本升级，请参见第 E.9 节。

E.8.2. 变更

- 修复当并非所有索引列都可返回时的仅索引扫描计划（Tom Lane）

  如果一个索引同时具有可返回列和不可返回列，且其中一个不可返回列是使用出现在可返回索引列中的表列的表达式，那么使用该表达式的查询可能会产生一个尝试读取不可返回列的仅索引扫描计划，而不是按预期从可返回列重新计算表达式。不可返回列将读取为 NULL，导致错误的查询结果。

- 确保将类型转换到未指定的 typmod 时生成 RelabelType 节点而非长度强制函数调用（Tom Lane）

  虽然强制函数应该做正确的事情（即什么都不做），但这种转换效率不必要地低下。

- 修复当数据库一致性恰好在 WAL 页面边界处达到时的 WAL 重放失败（Álvaro Herrera）

- 修复物理副本的启动以容忍事务 ID 回卷（Abhijit Menon-Sen，Tomas Vondra）

  如果副本服务器在主库上的活跃事务集跨越回卷边界时启动（即有些较新的事务的 XID 比较旧的事务小），副本将因 “out-of-order XID insertion in KnownAssignedXids” 而失败。副本会重试，但永远无法越过该错误。

- 移除在逻辑复制连接上发出的 SQL 命令的词法限制（Tom Lane）

  如果 SQL 命令包含未加引号的分号，或包含带有奇数个单引号或双引号的美元引用字面量，或以注释开头，walsender 进程就会失败。此外，错误恢复不正确也可能导致后续命令发生意外错误。

- 修复事务的最后一个子事务的提交时间戳可能丢失的问题（Alex Kingsborough，Kyotaro Horiguchi）

- 确保在检查点期间对 `pg_logical/mappings` 子目录执行 `fsync`（Nathan Bossart）

  在某些文件系统上，这一疏忽可能导致在系统崩溃后丢失逻辑重写状态文件。

- 为分区表构建扩展统计信息（Justin Pryzby）

之前的一个错误修复禁用了为旧式继承树构建扩展统计信息，但它也阻止了为分区表构建这些统计信息，这是一个不必要的限制。此变更允许 ANALYZE 计算分区表的统计对象的值。（但请注意，自动清理本身不处理分区表，因此如果需要维护此类统计信息，必须定期对分区表发出手动 ANALYZE。）

- 忽略继承树的扩展统计信息（Justin Pryzby）

  目前，扩展统计信息的值只是针对每个表本地计算的，而不是针对整个继承树。但是，在规划跨继承树的查询时，这些值被错误地参考了，可能导致比默认估计更差的结果。

- 当分区表的行类型在其他地方被用作复合类型时，禁止修改其列的数据类型（Tom Lane）

  这一限制对于普通表早已存在，但由于疏忽，对分区表未做检查。

- 禁止对作为副本标识索引一部分的列执行 ALTER TABLE ... DROP NOT NULL（Haiying Tang，Hou Zhijie）

  对于主键索引已经存在相同的禁止规则。

- 在 ALTER TABLE ADD PRIMARY KEY USING INDEX 期间正确更新缓存的表状态（Hou Zhijie）

  并发会话未能更新其关于表是否具有主键的判断，可能导致不正确的逻辑复制行为。

- 在切换 REPLICA IDENTITY 索引时正确更新缓存的表状态（Tang Haiying，Hou Zhijie）

  并发会话未能更新其关于哪个索引是副本标识索引的判断，可能导致不正确的逻辑复制行为。

- 避免在重新分配大量对象所有权的 REASSIGN OWNED BY 操作期间泄漏内存（Justin Pryzby）

- 修复出现在 INSERT ... VALUES 规则中的整行变量的显示（Tom Lane）

  整行变量会被打印为“var.*”，但这允许在规则重新加载时将其展开为各个列，导致不同的语义。附加显式类型转换以防止这种情况，正如我们在其他地方所做的那样。

- 修复或移除一些不正确的断言（Simon Riggs，Michael Paquier，Alexander Lakhin）

  这些错误应该只影响调试构建，不影响生产环境。

- 修复在多线程使用 libpq 或 ecpglib 早期报告的错误消息本地化失败的竞态条件（Tom Lane）

- 避免从 libpq 的 PQcancel 函数中调用 strerror（Tom Lane）

  PQcancel 应该可以安全地从信号处理程序中调用，但 strerror 不是安全的。这一错误用法仅在向服务器发送取消消息失败的不太可能的情况下才会发生，这也许解释了为什么没有相关报告。

- 使 psql 的 \password 命令默认设置 CURRENT_USER 的密码，而非连接的原始用户名（Tom Lane）

  这与文档中描述的行为一致，并且避免了在会话开始后执行了 SET ROLE 或 SET SESSION AUTHORIZATION 时可能出现的权限失败。为防止混淆，现在密码提示中包含了要操作的角色名称。

- 在 psql 和其他一些客户端程序中，避免在 Ctrl-C 信号处理程序中尝试调用 `gettext()`（Tom Lane）

  虽然没有已报告的失败可追溯到此错误，但这样做几乎不可能是安全的。

- 允许取消 pg_receivewal 和 pg_recvlogical 中的初始密码提示（Tom Lane，Nathan Bossart）

  之前在这些程序提示输入密码时无法通过 Ctrl-C 终止它们。

- 修复 pg_dump 对用户定义类型转换的转储顺序（Tom Lane）

  在极少数情况下，输出脚本可能在用户定义类型转换被创建之前就引用了它。

- 修复 pg_dump 和 pg_basebackup 中可能错误报告错误的问题（Tom Lane）

  之前的代码未能检查某些内核调用的错误，并且在其他情况下可能报告错误的 errno 值。

- 修复 `char(N)` 列上 `contrib/btree_gist` 索引的仅索引扫描结果（Tom Lane）

  仅索引扫描返回的列值去除了尾部空格，这不是预期的行为。这是因为数据就是以这种方式存储在索引中的。此修复更改代码以使用预期的空格填充来存储 `char(N)` 值。除非你执行 `REINDEX`，否则此类索引的行为不会立即改变；在更新过程中，去除空格的值将逐渐被替换。不使用仅索引扫描计划的查询在任何情况下都不受影响。

- 更改 configure 以使用 Python 的 sysconfig 模块代替已弃用的 distutils 模块来确定如何构建 PL/Python（Peter Eisentraut，Tom Lane，Andres Freund）

  对于 Python 3.10，这避免了在 configure 时出现关于 distutils 已弃用并计划在 Python 3.12 中移除的警告。想必一旦 3.12 发布，`configure --with-python` 将会完全失败。这种面向未来的改进确实有一个代价：sysconfig 在 Python 2.7 之前不存在，在 Python 3 分支中在 3.2 之前也不存在，因此不再可能针对早已停止维护的 Python 版本构建 PL/Python。

- 修复在 Windows 上使用 Perl 5.28 及更高版本时的 PL/Perl 编译失败（Victor Wagner）

- 修复在 Python 3.11 及更高版本上的 PL/Python 编译失败（Peter Eisentraut）

- 增加对使用 Visual Studio 2022 构建的支持（Hans Buschmann）

- 允许 MSVC 构建系统中的 `.bat` 包装脚本无需先切换到其所在目录即可调用（Anton Voloshin，Andrew Dunstan）

E.9. 发布版本 11.14

发布日期：. 2021-11-11

本次发布包含来自 11.13 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.9.1. 迁移到版本 11.14

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，请注意使用物理复制的安装应先更新备库再更新主库，如下面第三个变更日志条目所述。

此外，已发现若干可能导致索引损坏的错误，如接下来几个变更日志条目所述。如果存在上述任一情况，建议在更新后重建可能受影响的索引。

此外，如果是从早于 11.11 的版本升级，请参见第 E.12 节。

E.9.2. 变更

- 使服务器在 SSL 或 GSS 加密握手后拒绝多余的数据（Tom Lane）

  能够向 TCP 连接注入数据的中间人攻击者可以在本应受加密保护的数据库会话开始时塞入一些明文数据。这可以被用来向服务器发送伪造的 SQL 命令，不过只有在服务器不要求任何认证数据时才能生效。（然而，依赖 SSL 证书认证的服务器很可能不会要求认证数据。）

  PostgreSQL 项目感谢 Jacob Champion 报告此问题。（CVE-2021-23214）

- 使 libpq 在 SSL 或 GSS 加密握手后拒绝多余的数据（Tom Lane）

  能够向 TCP 连接注入数据的中间人攻击者可以在本应受加密保护的数据库会话开始时塞入一些明文数据。这可能被用来向客户端注入伪造的响应，不过 libpq 行为的其他细节使得这比听起来更难实现。另一种攻击方式是窃取客户端的密码或其他可能在会话早期发送的敏感数据。已证明在受 CVE-2021-23214 漏洞影响的服务器上这是可能的。

  PostgreSQL 项目感谢 Jacob Champion 报告此问题。（CVE-2021-23222）

- 修复在主库发送以不完整 WAL 记录结尾的 WAL 段后崩溃时的物理复制问题（Álvaro Herrera）

  如果主库未能存活足够长的时间来完成写入不完整 WAL 记录的剩余部分，则之前的崩溃恢复逻辑会回退并从不完整 WAL 记录的开头处重新覆盖写入 WAL。这是有问题的，因为备库可能已有该 WAL 段的副本。它们随后会看到不一致的下一个段，将无法在没有人工干预的情况下恢复。为修复此问题，在崩溃后重新启动时不再跨越 WAL 段边界回退。而是在下一个 WAL 段的开头写入一种新类型的 WAL 记录，通知读取器不完整的 WAL 记录将永远不会被完成且必须被忽略。

  应用此更新时，最好先更新备库再更新主库，这样如果主库发生崩溃，备库就已准备好处理这种新类型的 WAL 记录。

- 修复 `CREATE INDEX CONCURRENTLY` 以等待最新的预备事务（Andrey Borodin）

  刚刚预备的事务插入的行可能会从新索引中遗漏，导致依赖该索引的查询错过此类行。之前对此类问题的修复未能考虑到在 `CREATE INDEX CONCURRENTLY` 检查时仍在进行中的 `PREPARE TRANSACTION` 命令。与之前一样，在启用了预备事务（`max_prepared_transactions` > 0）的安装中，建议重建所有并发构建的索引，以防在构建时发生了此问题。

- 避免竞态条件导致后端无法将新行的条目添加到正在并发构建的索引中（Noah Misch，Andrey Borodin）

  虽然此情况在实际环境中似乎很少发生，但它可能影响使用 `CONCURRENTLY` 选项构建或重建的任何索引。建议重建此类索引以确保其正确性。

- 修复 `float4` 和 `float8` 的 hash 函数，使其对 NaN 产生一致的结果（Tom Lane）

  由于 PostgreSQL 的浮点类型将所有 NaN 视为相等，hash 函数必须为所有按 IEEE 754 标准表示 NaN 的位模式生成相同的 hash 值。以前没有做到这一点，因此对于非规范的 NaN 值，hash 索引和基于 hash 的查询计划可能产生错误结果。（在大多数机器上，`'-NaN'::float8` 就是产生此类值的一种方法。）如果浮点列的 hash 索引有任何可能包含此类值，建议重建这些索引。

- 防止在 wal_level = minimal 时，对 CREATE TABLESPACE 进行崩溃恢复导致数据丢失（Noah Misch）

  如果服务器在 CREATE TABLESPACE 与下一次检查点之间崩溃，重放会完全删除新表空间目录中的内容，并依靠后续 WAL 重放来恢复目录中的一切。这与跳过 WAL 写入的优化配合不当（例如向刚创建的表执行 COPY）。此类优化仅在 wal_level 为 minimal 时应用，而这并非 v10 及更新版本的默认设置。

- 确保在将表附加到分区表或从分区表中分离时正确地使关系缓存失效（Amit Langote，Álvaro Herrera）

  此疏忽可能导致直接对该分区进行后续插入/更新时出现错误行为，但仅在当前已存在的会话中。

- 确保创建或删除 FOR ALL TABLES 发布时使关系缓存失效（Hou Zhijie，Vignesh C）

  这一疏漏可能导致不正确的复制行为，直到所有当前已存在的会话都退出。

- 不要丢弃未指定类型修饰符的同类型转换（Tom Lane）

  例如，如果列 f1 的类型为 numeric(18,3)，解析器以前会直接丢弃 f1::numeric 这样的类型转换，理由是它在运行时没有任何作用。这一点没错，但表达式对外呈现的类型仍应视为普通的 numeric，而非 numeric(18,3)。这对于正确确定更大结构的类型很重要，例如递归 UNION。

- 修复复合类型上域的数组中元素字段的更新问题（Tom Lane）

  如果数组的元素是域而不是普通复合类型，则类似 UPDATE tab SET fld[1].subfld = val 这样的命令会失败。

- 如果当前数据库的编码不支持某个 ICU 排序规则，则禁止创建该排序规则（Tom Lane）

  以前允许这样做，但由于排序规则查找的工作方式，之后无法引用该排序规则；既不能使用它，甚至也不能删除它。

- 修复 numeric power() 在边界情况下的精度损失（Dean Rasheed）

  当第一个参数非常接近 1 时，结果可能不准确。

- 避免 {0} 内部存在捕获括号时出现正则表达式错误（Tom Lane）

  (.){0}...\1 这样的正则表达式会产生 “invalid backreference number” 错误。但 Perl 等其他正则表达式引擎不会报错，而且在一些非常相近的情况下，我们的引擎也不会报错。更糟的是，它还可能触发断言失败。现在修复为不抛出错误，而是静默地将该反向引用视为永不匹配。

- 防止正则表达式的反向引用有时在不应匹配的情况下匹配（Tom Lane）

  正则表达式引擎在拒绝一个部分匹配后，没有仔细清除捕获括号的匹配数据。这可能使后续的反向引用在本应因没有已定义的引用对象而失败的位置匹配成功。

- 修复迭代节点内部反向引用的正则表达式性能缺陷（Tom Lane）

  错误的回溯逻辑可能使寻找匹配所花费的时间呈指数级增长。幸运的是，其他优化在大多数情况下掩盖了此问题。

- 修复对 `time with time zone` 值应用 `AT TIME ZONE` 时产生的错误结果（Tom Lane）

  如果目标时区通过动态时区缩写指定（即定义为等价于完整时区名称，而非固定 UTC 偏移量的缩写），结果就会不正确。

- 避免仅使用 MCV 统计信息来估计列的范围（Tom Lane）

  在某些边界情况下，`ANALYZE` 会构建高频值（MCV）列表但不构建直方图，即使 MCV 列表并未包含所有观测值。在这种情况下，防止规划器仅使用 MCV 列表来估计列值的范围。

- 修复子事务内部 Portal 快照的恢复（Bertrand Drouvot）

  如果过程提交或回滚了一个事务，然后其下一个重要操作在新的子事务内部，快照管理会出错，导致悬挂指针和可能的崩溃。PL/pgSQL 中的一个典型示例是 `COMMIT` 紧跟一个执行查询的 `BEGIN ... EXCEPTION` 块。

- 确保在事务导出快照后失败时正确清理（Dilip Kumar）

  此疏忽仅在同一会话再次尝试导出快照时才会导致问题。最可能的场景是创建复制槽（随后回滚）然后再创建另一个复制槽。

- 防止备库上溢出子事务跟踪的回卷（Kyotaro Horiguchi，Alexander Korotkov）

  此疏忽可能导致备库上显著的性能下降（表现为过多的 SubtransSLRU 流量）。

- 确保在备库提升期间正确地将预备事务纳入考虑（Michael Paquier，Andres Freund）

  存在一个很窄的窗口，预备事务可能从并发运行的会话获取的快照中被遗漏。如果该会话随后使用该快照执行数据更新，可能导致错误的结果或数据损坏。

- 如果标记为 `NO SCROLL` 的游标因 `WITH HOLD` 选项而从上一个事务保留下来，则拒绝将其退回起点（Tom Lane）

  我们早已禁止从 `NO SCROLL` 游标向后获取数据，但由于历史原因，这一禁令没有涵盖将整个查询退回起点后再向前重新获取的情况。这一例外会导致不一致，尤其是对于跨事务保留的游标，它们可能没有存储退回起点所需的全部数据。现在禁止不可滚动的保留游标退回起点，以避免最严重的不一致。（v15 将完全移除这一例外。）

- 修复事务结束时保存已读取完毕的 `WITH HOLD` 游标可能发生的失败（Tom Lane）

- 修复对已增长到最大允许长度的关系的检测（Tom Lane）

  虽然尝试将表或索引扩展到超过 2^32-1 个块的限制会被拒绝，但拒绝得不够早，未能防止产生不一致的内部状态。

- 展开 `DO INSTEAD` 规则时，正确跟踪是否存在修改数据的 CTE（Greg Nancarrow，Tom Lane）

  以前未能做到这一点，可能导致不安全地选择并行计划等问题。

- 修复对扩展统计对象权限检查失败的不正确报告（Tomas Vondra）

  代码通常会产生 “cache lookup error”，而非预期的消息。

- 修复并行工作进程中不正确的快照处理（Greg Nancarrow）

  如果事务隔离级别低于 `REPEATABLE READ`，这一疏漏可能导致并行查询行为异常。

- 修复逻辑解码，使其正确忽略短暂表的 TOAST 表变更（Bertrand Drouvot）

  逻辑解码通常会忽略短暂表中的变更，例如 `ALTER TABLE` 堆重写期间创建的表。但这一过滤没有应用于其关联的 TOAST 表（如果存在），因而在重写正在发布的表时可能出错。

- 确保 WAL 接收进程进程在退出前创建所有必需的归档通知文件（Fujii Masao）

  如果 WAL 接收进程恰好在 WAL 段边界处退出，它就不会为最后接收的段创建通知文件，从而延迟该段在备库上的归档。

- 避免尝试锁定使用 `SELECT FOR UPDATE` 的规则中的 `OLD` 和 `NEW` 伪关系（Masahiko Sawada，Tom Lane）

- 修复解析器对聚合 `FILTER` 子句的处理（Tom Lane）

  如果 `FILTER` 表达式是普通的布尔列，聚合的语义层级可能被错误判定，从而产生不符合规范的行为。如果 `FILTER` 表达式本身是返回布尔值的聚合，本应抛出错误却没有，因而很可能在执行时崩溃。

- 避免在 LLVM 内部发生错误后尝试清理 LLVM 状态（Andres Freund，Justin Pryzby）

  这可以防止 LLVM 发生致命错误后，后端退出时很可能出现的崩溃。

- 避免在删除拥有正在被并发删除的对象的角色时出现空指针解引用崩溃（Álvaro Herrera）

- 防止当 `lo_export()` 或相关函数失败时出现 “snapshot reference leak” 警告（Heikki Linnakangas）

- 确保统计视图会计入对 SP-GiST 索引的扫描（Tom Lane）

  SP-GiST 代码遗漏了递增索引扫描次数计数器的操作，虽然逐元组计数器已被正确递增。

- 如果在恢复期间修改 `recovery_min_apply_delay`，则重新计算相关等待间隔（Soumyadeep Chakraborty，Ashwin Agrawal）

- 修复 `simplehash.h` hash 表达到 2^32 个元素时的无限循环（Yura Sokolov）

  此缺陷似乎不太可能在实践中被触发，因为就 `simplehash.h` 的现有用途而言，需要将 `work_mem` 设置为数百 GB 才会出现。

- 减少计算扩展统计信息时的内存消耗（Justin Pryzby，Tomas Vondra）

- 修复 ecpg，使其在建立连接期间 `malloc()` 失败后能够正确恢复（Michael Paquier）

- 修复 PL/pgSQL `CALL` 语句参数中调用的稳定函数被错误求值的问题（Tom Lane）

  这些函数调用时使用了过期的快照，因此无法看到自会话顶层命令开始以来的任何数据库变更。

- 允许使用 `EXIT` 退出 PL/pgSQL 例程的最外层块（Tom Lane）

  如果例程不要求显式 `RETURN`，这种用法应该有效，但以前会被拒绝。

- 移除 pg_ctl 对生成命令总长度的硬编码限制（Phil Krylov）

例如，这移除了对可传递给 postmaster 的命令行选项数量的限制。pg_ctl 处理的单个路径名，例如 postmaster 可执行文件名或数据目录名，在大多数情况下仍限制为 MAXPGPATH 字节。

- 修复 pg_dump 以正确导出非全局默认权限（Neil Chen，Masahiko Sawada）

  如果全局（不受限的）ALTER DEFAULT PRIVILEGES 命令撤销了某个默认存在的权限，例如函数的 EXECUTE，然后受限的 ALTER DEFAULT PRIVILEGES 命令又为选定的角色或模式重新授予了该权限，则 pg_dump 无法正确导出受限的权限授予。

- 使 pg_dump 对要导出的分区表获取共享锁（Tom Lane）

  此疏忽通常是相当无害的，因为一旦 pg_dump 锁定了任何叶分区，就足以阻止对分区表本身的重大 DDL。但是，在导出没有子分区的分区表时可能出现问题，因为不会持有相关的锁。

- 避免逐表查询 RLS 策略和重复调用 format_type()，从而提高 pg_dump 的性能（Tom Lane）

  这些修改在从本地服务器转储时只能带来小幅提升，但从远程服务器转储时，可以因网络往返次数减少而获益显著。

- 修复 pg_dump 在尝试从 8.3 之前版本的服务器导出触发器定义时的崩溃（Tom Lane）

- 修复 pg_restore 关于无效大对象 TOC 文件的错误消息中的不正确文件名（Daniel Gustafsson）

- 修复对 "char" 列（而非 char(*n*) 列）的 contrib/btree_gin 索引使用 < 或 <= 操作符进行索引扫描时的失败（Tom Lane）

  这样的索引扫描无法返回所有应返回的条目。

- 改变 contrib/pg_stat_statements 以最大 1GB 为单位读取其“查询文本”文件（Tom Lane）

  如此大的查询文本文件非常不常见，但如果确实出现，之前的代码在 Windows 64 上会失败（因为它拒绝超过 2GB 的单次读取请求）。

- 修复 contrib/postgres_fdw 尝试报告数据转换错误时的空指针崩溃（Tom Lane）

- 增加对 RISC-V 架构的自旋锁支持（Marek Szuba）

  这对于在该平台上获得合理的性能至关重要。

- 支持 OpenSSL 3.0.0（Peter Eisentraut，Daniel Gustafsson，Michael Paquier）

- 为 PostgreSQL 创建的 OpenSSL BIO（I/O 抽象）对象设置正确的类型标识符（Itamar Gafni）

  这一疏漏可能只对执行审计 OpenSSL 安装等任务的代码有影响。但严格来说，这违反了 OpenSSL API，因此予以修复。

- 使 pg_regexec() 能够稳健地处理超出范围的 *search_start* 参数（Tom Lane）

  当 *search_start* 超过字符串末尾时，返回 REG_NOMATCH，而不再可能崩溃。核心 PostgreSQL 中可能不会出现这种情况，但扩展可能对参数值不够谨慎。

- 确保 `GetSharedSecurityLabel()` 可以在尚未构建关键关系缓存条目的新启动会话中使用（Jeff Davis）

- 使用 CLDR 项目的数据将 Windows 时区名称映射到 IANA 时区（Tom Lane）

  在 Windows 上运行时，initdb 尝试将新集簇的 `timezone` 参数设置为与系统当前时区匹配的 IANA 时区。我们使用的是多年前生成的映射表，并且只是零星地更新；不出所料，它包含许多错误以及遗漏了最近添加的时区。事实证明 CLDR 一直在跟踪最合适的映射，所以开始使用他们的数据。此更改不会影响任何现有安装，仅影响新初始化的集簇。

- 将时区数据文件更新到 tzdata 2021e 版，以反映斐济、约旦、巴勒斯坦和萨摩亚的夏令时法规变更，并修正巴巴多斯、库克群岛、圭亚那、纽埃、葡萄牙和汤加的历史数据。

  此外，Pacific/Enderbury 时区已重命名为 Pacific/Kanton。同时，以下时区已合并到附近的、人口更多的时区中，因为它们的时钟自 1970 年以来一直与之一致：Africa/Accra、America/Atikokan、America/Blanc-Sablon、America/Creston、America/Curacao、America/Nassau、America/Port_of_Spain、Antarctica/DumontDUrville 和 Antarctica/Syowa。在所有这些情况下，之前的时区名称仍作为别名保留。

E.10. 发布版本 11.13

发布日期：. 2021-08-12

本次发布包含来自 11.12 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.10.1. 迁移到版本 11.13

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.11 的版本升级，请参见第 E.12 节。

E.10.2. 变更

- 修复重复应用投影步骤时产生错误计划的问题（Tom Lane）

  当两个 ProjectionPath 相互叠加时，规划器可能生成错误的计划。已知唯一能触发这种情况的方式涉及并行排序操作，但也可能存在其他情形。其结果可能是崩溃或错误的查询结果，也可能泄露服务器内存内容。（CVE-2021-3677）

- 更彻底地禁止 SSL 重新协商（Michael Paquier）

  虽然 SSL 重新协商已被禁用一段时间，但服务器仍会配合客户端发起的重新协商请求。恶意构造的重新协商请求可能导致服务器崩溃（参见 OpenSSL 问题 CVE-2021-3449）。在允许完全禁用此功能的 OpenSSL 版本（即 1.1.0h 及更新版本）上，将其彻底禁用。

- 在过程内部执行 `COMMIT` 或 `ROLLBACK` 后恢复 Portal 级快照（Tom Lane）

  此项修改修复了紧接着 `COMMIT`/`ROLLBACK` 尝试获取经过 TOAST 处理的值时，出现 “no known snapshots” 或 “missing chunk number 0 for toast value” 等错误的情况。

  某些扩展可能尝试在任何 Portal 之外执行 SQL 代码。它们有责任在执行前确保存在外层快照。以前，不提供快照可能行得通，也可能不行；现在则会始终以 “cannot execute SQL without an outer snapshot or portal” 错误失败。

- 避免持久化读取非稳定查询的游标输出时出现异常行为（Tom Lane）

以前，我们总会退回起点并重新读取整个查询结果，可能得到与先前执行不同的结果，从而给后续处理造成很大混乱。对于 NO SCROLL 游标，可以只存储查询输出中尚未读取的部分来修复此问题；由于 NO SCROLL 游标不能后退，这样就足够了。带 SCROLL 选项的游标仍存在风险，但文档早已说明，将此选项用于非稳定查询是不安全的。现在进一步加强这些文档警告。

另外，强制 PL/pgSQL 遍历查询结果的 FOR 循环所使用的隐式游标采用 NO SCROLL 模式，以避免在过程内部提交期间持久化这类游标时出现此类问题。

- 拒绝 `SELECT ... GROUP BY GROUPING SETS (()) FOR UPDATE`（Tom Lane）

  正如不允许将 `FOR UPDATE` 与普通 `GROUP BY` 一起使用一样，这种情况也应被禁止，但相关检查未能正确处理空分组集。最终结果会是执行器中的空指针解引用。

- 拒绝 `WITH` 中的查询被重写为仅包含 `NOTIFY` 的情况（Tom Lane）

  以前，这些情况会导致崩溃。

- 在 `numeric` 乘法中，如果结果的小数点后超过 16383 位，则对结果进行舍入，而不报错（Dean Rasheed）

- 修复对 `numeric` 值计算极大次幂时的边界情况错误和精度损失（Dean Rasheed）

- 修复 `to_char()` 使用 `EEEE` 格式且输入的 `numeric` 值小于 10^(-1001) 时的除零错误（Dean Rasheed）

- 修复 `pg_size_pretty(bigint)`，使其对负值的舍入与对正值的舍入方式一致（也与 `numeric` 版本一致）（Dean Rasheed，David Rowley）

- 使 `pg_filenode_relation(0, 0)` 返回 NULL，而不报错（Justin Pryzby）

- 使 `ALTER EXTENSION` 在添加或移除成员对象时锁定扩展（Tom Lane）

  以前的代码允许 `ALTER EXTENSION ADD/DROP` 与 `DROP EXTENSION` 并发执行，导致崩溃或目录条目损坏。

- 修复 `ALTER SUBSCRIPTION`，使其拒绝空的槽名称（Japin Li）

- 将分区表的触发器克隆到新分区时，确保复制其启用状态（Álvaro Herrera）

- 避免为 `REFRESH MATERIALIZED VIEW CONCURRENTLY` 生成的查询中的别名冲突（Tom Lane，Bharath Rupireddy）

  如果物化视图包含某些名称的列，尤其是 `mv` 和 `newdata`，此命令就会失败。

- 修复 `PREPARE TRANSACTION`，使其正确检查会话生命周期锁与事务生命周期锁之间的冲突（Tom Lane）

  如果一个事务在同一个咨询锁 ID 值上同时持有会话生命周期锁和事务生命周期锁，就不能预备该事务。以前没有完整检查这一限制，可能导致 `PREPARE TRANSACTION` 期间发生 PANIC。

- 修复目标角色在 RLS 策略中被列出多次时 `DROP OWNED BY` 的异常行为（Tom Lane）

- 在 `DROP OWNED BY` 期间从 RLS 策略中移除角色时，跳过不必要的错误检查（Tom Lane）

  值得一提的是，这修复了某些必须是超级用户才能使用 `DROP OWNED BY` 的情况。

- 向外部表添加列时，不存储“快速默认值”（Andrew Dunstan）

  由于此类表没有本地堆存储，快速默认值没有用途，却会干扰后续操作。除了阻止 `ALTER TABLE` 命令创建此类目录条目之外，还调整后续代码，使其能够应对错误存在的此类条目。

- 允许以事务方式更新索引状态标志（Michael Paquier，Andrey Lepikhov）

  这可以避免处理实际上并非不可变的索引谓词时发生失败。虽然这种情况不被视为受支持的用法，但这里最初采用非事务更新的原因早已不复存在，因此不妨进行修改。

- 避免缓存计划中出现 `CREATE DOMAIN` 或 `ALTER DOMAIN` 时损坏计划缓存条目（Tom Lane）

- 使 WAL 发送进程在 `pg_stat_activity` 中显示最新的复制命令（Tom Lane）

  以前，WAL 发送进程会显示其最新的 SQL 命令，如果它此时正在执行复制操作，就会令人困惑。现在将复制协议命令与 SQL 命令同等显示。

- 当 `postgresql.conf` 中的相关条目被移除时，使 `pg_settings.pending_restart` 显示为 true（Álvaro Herrera）

  对于必须重启 postmaster 才能更改的条目，`pending_restart` 能正确显示条目已被修改的情况，却无法显示条目被完全移除的情况。

- 修复新备库在边界情况下无法跟随新主库的问题（Dilip Kumar，Robert Haas）

  在一组少见的条件组合下，备库可能最终尝试跟随错误的 WAL 时间线。

- 当事务中止记录的 WAL 重放导致文件截断时，更新最小恢复点（Fujii Masao）

  文件截断是不可逆的，因此在早于该记录的位置停止恢复已不再安全。事务提交的对应情况多年前就已修复，但这一情况被遗漏了。

- 在 WAL 接收进程中，避免发生错误后尝试查找系统目录（Masahiko Sawada，Bharath Rupireddy）

- 确保备库的启动进程在等待 WAL 到达时及时响应关闭信号（Fujii Masao，Soumyadeep Chakraborty）

- 在未能成为事务提交组成员后正确清除共享状态（Amit Kapila）

  在特定时序下，当后续某个会话复用同一个 PGPROC 对象时，这可能导致断言失败。

- 增加锁定，避免其他进程并发写入时读取到错误的 relmapper 数据（Heikki Linnakangas）

- 改进对违反复制协议的情况的检查（Tom Lane）

  逻辑复制工作进程经常使用断言来检查可能由无效或乱序复制命令触发的情况。这似乎并不明智，因此将这些检查改为常规错误检查。

- 修复多个逻辑复制工作进程尝试截断同一个表时的死锁（Peter Smith，Haiying Tang）

- 修复对推测插入进行逻辑解码时的错误情况和内存泄漏（Dilip Kumar）

- 避免出错后留下无效的记录类型 hash 表条目（Sait Talha Nisanci）

这可能导致后续崩溃或内存泄漏。

- 修复 `CREATE TABLE ... AS EXECUTE` 在某些错误情况下的计划缓存引用泄漏（Tom Lane）
- 修复在并行工作进程之间共享元组描述符的代码中的竞争条件（Thomas Munro）

  在特定时序下，可能发生崩溃。
- 修复释放 BackgroundWorkerSlots 时可能出现的竞争条件（Tom Lane）

  这可能不会修复 Intel 硬件上任何可观察到的缺陷，但内存排序规则较弱的机器可能存在问题。
- 修复排序代码中潜在的崩溃（Ronan Dunklau）

  某个代码路径可能尝试释放空指针。在核心服务器使用排序的方式中，这种情况似乎无法触发，但扩展或许能够触发。
- 防止 SP-GiST 索引插入中的无限循环（Tom Lane）

  如果 INCLUDE 列占用的空间很大，使叶子索引元组无论如何都无法放入一页，text_ops 操作符类就会陷入无限循环，徒劳地尝试让该元组放得下。虽然 v11 之前的版本没有 INCLUDE 列，仍将这一防止循环的修复反向移植到这些版本，因为它似乎能有效防范操作符类中的缺陷。
- 确保 SP-GiST 索引插入可以被查询取消请求终止（Tom Lane，Álvaro Herrera）
- 修复未初始化变量的缺陷，该缺陷可能使 PL/pgSQL 表现得像 `INTO` 子句指定了 `STRICT`，即使实际并未指定（Tom Lane）
- 不要因 libpq 打印函数中的内存不足而中止进程（Tom Lane）
- 在 ecpg 中，允许将 `numeric` 值 INT_MIN（通常为 -2147483648）转换为整数（John Naylor）
- 在 psql 和其他客户端程序中，避免处理编码无效的数据时越过字符串末尾（Tom Lane）

  字符串末尾附近编码不正确的多字节字符，可能导致各种处理循环越过字符串末尾的 NUL。根据随后内存中的内容，其后果可能从没有可检测的问题到程序崩溃不等。这与 CVE-2006-2313 相似，不过这些具体情况似乎不会造成值得关注的安全后果。
- 修复 pg_dump，使其正确处理分区表上启用状态与父触发器不同的触发器（Justin Pryzby，Álvaro Herrera）
- 避免对不同时区创建的归档文件运行 pg_restore 时出现 “invalid creation date in header” 警告（Tom Lane）
- 使 pg_upgrade 沿用旧安装实例的 `oldestXID` 值（Bertrand Drouvot）

  以前，新安装实例的 `oldestXID` 会被设为一个很旧的值，通常足以强制立即进行防回卷自动清理。从性能角度看，这并不理想；更糟的是，使用较大 `autovacuum_freeze_max_age` 值的实例可能在升级后不久就遭遇意外的强制关闭。
- 扩展 pg_upgrade，使其检测应升级的扩展并发出警告（Bruce Momjian）

现在会生成一个脚本文件，其中包含将扩展升级到新安装实例中默认版本所需的 `ALTER EXTENSION UPDATE` 命令。

- 避免 pg_receivewal 在压缩与非压缩 WAL 存储之间切换时出现问题（Michael Paquier）
- 在 `contrib/postgres_fdw` 中，避免发生错误后尝试查找系统目录（Tom Lane）

  虽然这通常能正常工作，但并不十分安全，因为发生的错误可能已经使目录访问无法正常进行。这一修复的副作用是，关于数据转换错误的消息现在会提及查询中的表和列别名（如果使用了别名），而非底层外部表或列的真实名称。
- 改进隔离测试基础设施（Tom Lane，Michael Paquier）

  允许对隔离测试步骤添加注解，说明预期的完成顺序。这使原本存在竞争的测试用例也能获得稳定结果，无需再像以前那样使用长时间延迟来规避竞争条件（而且以前也并未完全成功）。允许使用不带引号的标识符作为隔离测试的会话或步骤名称（以前，所有此类名称都必须加双引号）。检测隔离测试中未使用的步骤并发出警告。改进隔离测试中查询结果的显示。移除 isolationtester 的“试运行”模式。消除 isolationtester 自身的内存泄漏。
- 减少强制破坏缓存测试的开销（Tom Lane）
- 修复 PL/Python 的回归测试，使其在 Python 3.10 下通过（Honza Horak）
- 使 `printf("%s", NULL)` 打印 `(null)`，而不崩溃（Tom Lane）

  这应该能提高服务器在边界情况下的稳健性，并使我们的 `printf` 实现与常见库保持一致。
- 修复时间点恢复停止在 `ROLLBACK PREPARED` 记录处时不正确的日志消息（Simon Riggs）
- 改进 `ALTER TABLE` 对关系种类错误的消息（Kyotaro Horiguchi）
- 使涉及“非负”值的错误消息更清晰（Bharath Rupireddy）
- 修复 configure，使其支持不再提供独立 `libldap_r` 库的 OpenLDAP 2.5（Adrian Ho，Tom Lane）

  如果不存在 `libldap_r` 库，现在会静默假定 `libldap` 是线程安全的。
- 增加新的 make 目标 `world-bin` 和 `install-world-bin`（Andrew Dunstan）

  它们分别与 `world` 和 `install-world` 相同，但不构建或安装文档。
- 修复 TAP 测试的 make 规则（`prove_installcheck`），使其在 PGXS 用法下正常工作（Andrew Dunstan）
- 调整 JIT 代码，为即将到来的 LLVM API 变更做好准备（Thomas Munro，Andres Freund）

  LLVM 13 引入了一项不兼容的 API 变更，会使我们以前的 JIT 编译器崩溃。
- 避免假定 GSSAPI 库返回的字符串以空字符结尾（Tom Lane）

  GSSAPI 规范提供字符串指针和长度。在实践中，字符串后的下一个字节似乎通常为零，因此我们以前的代码没有实际失败；但确实收到了 AddressSanitizer 报错的报告。
- 允许在 MSVC 上构建 GSSAPI 支持（Michael Paquier）

修复与现代 Kerberos 构建的各种不兼容问题。

- 在 MSVC 构建中，如果指定了 `--with-pgport`，则将其包含在 pg_config 报告的 configure 选项中（Andrew Dunstan）

E.11. 发布版本 11.12

发布日期：. 2021-05-13

本次发布包含来自 11.11 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.11.1. 迁移到版本 11.12

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.11 的版本升级，请参见第 E.12 节。

E.11.2. 变更

- 防止数组下标计算中的整数溢出（Tom Lane）

  以前，当数组下界加上长度发生整数溢出时，数组代码不会报错。这会使数组中后面的项无法访问（因为它们的下标无法表示为整数），更重要的是还会干扰后续赋值操作。这可能造成内存覆盖，进而导致崩溃或非预期的数据修改。（CVE-2021-32027）

- 修复 `INSERT ... ON CONFLICT ... UPDATE` 目标列表中对 “junk” 列的错误处理（Tom Lane）

  如果 `UPDATE` 列表包含任何多列子查询（它们除了产生正式结果外，还会产生 junk 列），`UPDATE` 路径最终会存储包含额外 junk 列值的元组。短期内这基本无害，但如果向表中添加新列，这些值就会变得可访问；如果它们与新增列的数据类型不匹配，就可能导致异常。

  此外，在支持跨分区更新的版本中，由此类情况触发的跨分区更新存在相反的问题：junk 列被从目标列表中移除，通常会因多列子查询机制失效而立即崩溃。（CVE-2021-32028）

- 修复涉及连接的跨分区更新中 `UPDATE ... RETURNING` 输出可能计算不正确的问题（Amit Langote，Etsuro Fujita）

  如果对分区表的 `UPDATE` 使某行被移动到另一个行类型在物理上不同的分区（例如，已删除列的集合不同），为该行计算 `RETURNING` 结果时可能报错或得到错误的结果。只有当 `UPDATE` 涉及将其他表连接到目标表时，才会出现此错误。（CVE-2021-32029）

- 修复分区表中约束可延迟属性的调整（Álvaro Herrera）

  对分区表的外键约束执行 `ALTER TABLE ... ALTER CONSTRAINT` 时，未能调整叶子分区的约束和触发器上的 `DEFERRABLE` 或 `INITIALLY DEFERRED` 标记。这导致此类约束产生非预期的行为。更新到此版本后，可以执行新的 `ALTER` 命令设置所需属性，从而修复行为异常的分区表。

  此项修改还禁止直接对叶子分区的约束执行这样的 `ALTER`。唯一受支持的情况是整个分区层次结构具有相同的约束属性，因此此类 `ALTER` 必须在分区根表上执行。

- 禁止将标识列标记为可空（Vik Fearing）

GENERATED ... AS IDENTITY 隐含 NOT NULL，因此不允许将其与显式的 NULL 指定结合使用。

- 允许 ALTER ROLE/DATABASE ... SET 设置 role、session_authorization 和 temp_buffers 参数（Tom Lane）

  以前，过于急切的有效性检查可能拒绝这些命令，即使相关值在之后实际使用时本可以正常工作。这使转储/重载和升级场景中出现了命令顺序方面的隐患。

- 修复将 COLLATE 表达式的结果强制转换为不支持排序规则的类型时的缺陷（Tom Lane）

  这会生成一棵语法解析树，其中 COLLATE 看起来被应用于一个不支持排序规则的值。虽然这通常没有实际影响（因为 COLLATE 在运行时没有作用），但仍有可能构造出在转储/重载期间被拒绝的视图。

- 禁止通过“快速路径”线协议消息调用窗口函数和过程（Tom Lane）

  这里只支持普通函数。虽然尝试调用聚合函数已经会失败，但调用窗口函数会导致崩溃，而调用过程只有在过程不进行事务控制时才能正常工作。

- 扩展 pg_identify_object_as_address()，使其支持事件触发器（Joel Jacobson）

- 修复 to_char() 对负时间间隔使用罗马数字月份格式代码的处理（Julien Rouhaud）

  以前，此类情况通常会导致崩溃。

- 检查 pg_import_system_collations() 的参数是否为有效的模式 OID（Tom Lane）

- 修复解析 BRE 模式正则表达式中的 \{*m*,*n*\} 量词时使用未初始化值的问题（Tom Lane）

  此错误可能使量词表现为非贪婪模式，即表现得像完整正则表达式中的 {*m*,*n*}? 量词。

- 使用扩展统计信息估计分组数量时，不要忽略系统列（Tomas Vondra）

  这会导致 SELECT ... GROUP BY a, b, ctid 等查询得到异常的估计值。

- 避免估计具有很长固定前缀的正则表达式的选择率时出现除零错误（Tom Lane）

  这通常会产生 NaN 选择率值，从而导致断言失败或规划器行为异常。

- 修复 BRIN 索引位图扫描中访问超出表末尾位置的错误（Tomas Vondra）

  如果 BRIN 索引使用的页范围大小不是二的幂，在某些边界情况下，位图扫描可能尝试获取超出表实际末尾的页面，导致“could not open file”错误。

- 避免从 WAL 恢复未提交的两阶段事务时发生不正确的时间线变更（Soumyadeep Chakraborty，Jimmy Yih，Kevin Yeap）

  此错误可能使后续 WAL 记录在错误的时间线 ID 下写入，导致之后出现一致性问题，甚至完全无法重启服务器。

- 确保关闭备库的启动进程时释放锁（Fujii Masao）

  在备库仍处于恢复状态时将其关闭，可能会留下某些仍被持有的锁。这会导致调试构建中的断言失败；尚不清楚在生产构建中是否可能造成严重后果。

- 修复逻辑复制工作进程执行 `ALTER SUBSCRIPTION REFRESH` 时的崩溃（Peter Smith）

  核心代码不会这样做，但副本触发器可能会。

- 确保在较新版本的 FreeBSD 上默认使用 `wal_sync_method` = `fdatasync`（Thomas Munro）

  FreeBSD 13 支持 `open_datasync`，通常它会成为默认选择。但尚不清楚这对 Postgres 是否确有改进，因此暂时保留现有默认值。

- 确保分离 DSM 段期间被中断时仍能完成清理（Thomas Munro）

  此错误可能导致并行查询后未能及时清理临时文件。

- 修复初始化服务器 SSL 参数时的内存泄漏（Michael Paquier）

  这通常影响很小，但如果反复向 postmaster 发送 SIGHUP 信号，泄漏就会随时间累积。

- 修复服务器中多种较小的内存泄漏（Tom Lane，Andres Freund）

- 修复 PL/pgSQL `DO` 块同时使用复合类型变量和事务控制时的失败（Tom Lane）

  以前，此类情况会导致关于元组描述符泄漏的错误。

- 防止 libpq 收到长度损坏的 ParameterDescription 消息时陷入无限循环（Tom Lane）

- 当 initdb 打印如何启动服务器的说明时，使其在 Windows 上显示的 pg_ctl 路径使用反斜杠分隔符（Nitin Jadhav）

- 修复 psql，恢复 `\connect service=`*`something`* 以前的行为（Tom Lane）

  先前的一项缺陷修复导致环境变量（例如 `PGPORT`）在此情况下覆盖服务文件中的条目。现在恢复以前的行为，使优先级反过来。

- 修复 psql 的 `\e` 及相关命令检测文件修改时的竞争条件（Laurenz Albe）

  输入速度非常快的用户可能使代码基于文件时间戳对临时编辑文件是否更改的判断失准。

- 修复 pg_restore 中遗漏的文件版本检查（Tom Lane）

  从不支持定位的来源读取自定义格式归档时，pg_restore 忽略了检查归档版本。如果输入的归档版本比它所支持的版本更新，之后就会以混乱的方式失败。

- 为 pg_upgrade 增加对包含不可升级数据类型的用户表的检查（Tom Lane）

  修复对不可升级的数据类型嵌入容器类型（例如数组或范围）中的某些情况的检测。另外，如果用户表包含系统定义的复合类型的列，则不允许升级，因为这些类型的 OID 在不同版本之间并不稳定。

- 修复 pg_waldump，使其在生成逐记录统计信息时正确统计 `XACT` 记录（Kyotaro Horiguchi）

- 修复 `contrib/amcheck`，使其不再因元组标志 `HEAP_XMAX_LOCK_ONLY` 和 `HEAP_KEYS_UPDATED` 同时设置而报错（Julien Rouhaud）

  这是 `SELECT FOR UPDATE` 之后的一种有效状态。

- 调整 VPATH 构建规则，以支持较新版本的 Oracle Developer Studio 编译器（Noah Misch）
- 修复 Solaris 上针对 Python 3 的 PL/Python 测试（Noah Misch）

E.12. 发布版本 11.11

发布日期：. 2021-02-11

本次发布包含来自 11.10 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.12.1. 迁移到版本 11.11

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，请参阅下方变更记录中的第二项，其中描述了升级后可能适合重建索引的情况。

此外，如果是从早于 11.6 的版本升级，请参见第 E.17 节。

E.12.2. 变更

- 修复违反约束的错误消息中的信息泄漏（Heikki Linnakangas）

  如果 `UPDATE` 命令尝试将某行移动到其他分区，却发现该行违反了新分区上的某个约束，并且该分区中列的物理位置与父表不同，那么错误消息可能泄露用户没有 `SELECT` 权限的列的内容。（CVE-2021-3393）

- 修复 `CREATE INDEX CONCURRENTLY`，使其等待并发的预备事务（Andrey Borodin）

  `CREATE INDEX CONCURRENTLY` 等待所有并发事务完成，以便看到它们插入的行时，也必须出于相同原因等待所有预备事务完成。未能这样做意味着，预备事务插入的行可能被遗漏在新索引之外，导致依赖该索引的查询找不到这些行。对于启用了预备事务（`max_prepared_transactions` > 0）的安装实例，建议重建所有并发构建的索引，以防其构建时发生过此问题。

- 避免通过扩展查询协议执行会回滚事务的 `CALL` 或 `DO` 语句时发生崩溃（Thomas Munro，Tom Lane）

  在 PostgreSQL 13 中，这种情况必然导致空指针解引用。在更早的版本中，此缺陷似乎没有可见症状，但尚不能完全确定它绝不会造成问题。

- 修复分区剪枝逻辑，使其处理不对称的 hash 分区集合（Tom Lane）

  如果 hash 分区表的分区大小不相等（即模数值不同），或缺少某些余数值对应的分区，规划器的剪枝逻辑就可能错误地认定某些分区无需扫描，导致查询无法找到本应找到的行。

- 避免将 `WHERE CURRENT OF` 应用于计划中包含 MergeAppend 节点的游标时产生错误结果（Tom Lane）

  这种情况不受支持（一般来说，不能保证使用 `ORDER BY` 的游标可以简单地用于更新）；但以前的代码没有拒绝它，并可能悄然给出错误匹配。

- 修复将 `WHERE CURRENT OF` 应用于计划中包含自定义扫描节点的游标时的崩溃（David Geier）

- 修复规划器对在某个连接层级计算、且仅在同一层级使用的占位符的处理（Tom Lane）

这一疏漏可能导致 “failed to build any *N*-way joins” 规划器错误。

- 更谨慎地判断索引访问方法是否支持标记/恢复（Andrew Gierth）

  这可以防止在罕见的边界情况下出现关于缺少支持函数的错误。

- 调整设置，降低大量使用并行查询时耗尽 DSM 槽的可能性（Thomas Munro）

- 修复对并行查询所需共享内存量的高估（Takayuki Tsunakawa）

- 修复 `ALTER DEFAULT PRIVILEGES`，使其安全地处理重复参数（Michael Paquier）

  同一命令中重复的角色名或模式名可能导致 “tuple already updated by self” 错误或违反唯一约束。

- 在 `pg_authid` 变更时清除 ACL 相关缓存（Noah Misch）

  此项修改确保与权限相关的判断及时反映 `ALTER ROLE ... [NO] INHERIT` 的结果。

- 防止错误处理有歧义的 `CREATE TABLE LIKE` 子句（Tom Lane）

  初步创建新表后，会重新检查 `LIKE` 子句，以处理索引等对象的导入。这次重新检查有可能找到另一个同名表，从而导致意外行为；例如，新表是一个与 `LIKE` 目标同名的临时表时就会如此。

- 调整 `CREATE TABLE LIKE` 中的操作顺序，在构建外键约束之前克隆索引（Tom Lane）

  这修复了外层 `CREATE TABLE` 中声明的自引用外键约束依赖于来自 `LIKE` 子句的索引的情况。

- 禁止对系统目录执行 `CREATE STATISTICS`（Tomas Vondra）

- 禁止将继承子表转换为视图（Tom Lane）

- 确保为已删除关系分配的磁盘空间在提交时及时释放（Thomas Munro）

  以前，如果删除的关系跨越多个 1GB 段，只有第一个段会立即被截断。其他段只是被解除链接；只要其他后端仍打开着这些文件，内核就不能释放其存储空间。

- 防止删除被分区关系引用、但未用于任何实际存储的表空间（Álvaro Herrera）

  以前允许这样做，但之后对分区关系的操作会失败。

- 修复 `COPY FROM` 对反斜杠转义的多字节字符的处理（Heikki Linnakangas）

  反斜杠后跟多字节字符时未被正确处理。在某些客户端字符编码中，这可能导致多字节字符的一部分被误认为字段分隔符或复制数据结束标记。

- 在不带 `ANALYZE` 的 `EXPLAIN` 中，避免预先分配执行器 hash 表（Alexey Bashtanov）

- 修复最近在 `LISTEN`/`NOTIFY` 队列处理中引入的竞争条件（Tom Lane）

  刚开始侦听的后端可能尝试读取正在被截断的 SLRU 页面，从而可能导致错误。

队列尾指针可能被设为一个与任何后端的队列位置都不相等的值，导致队列截断逻辑实际上失效。继续使用 `NOTIFY` 随后会产生队列填满警告，最终无法再发送任何通知，直到服务器重启。

- 允许 `jsonb` 连接操作符处理所有 JSON 数据类型组合（Tom Lane）

  我们可以连接两个 JSON 对象或两个 JSON 数组。对于其他情况，将非数组输入包装为单元素数组，再进行数组连接。以前，某些输入组合遵循此规则，而其他组合却会无端报错。

- 修复解析 BRE 模式正则表达式中的 `*` 量词时使用未初始化值的问题（Tom Lane）

  此错误可能使量词表现为非贪婪模式，即表现得像完整正则表达式中的 `*?` 量词。

- 修复 numeric `power()` 在指数恰好为 `INT_MIN`（-2147483648）时的行为（Dean Rasheed）

  以前会产生没有有效数字的结果。

- 修复 `substring()` 函数中的整数溢出情况（Tom Lane，Pavel Stehule）

  如果指定的起始索引与长度相加时发生整数溢出，`substring()` 就会行为异常：要么对本应成功的情况抛出虚假的 “negative substring length” 错误，要么未能指出长度为负（而在大多数情况下返回整个字符串）。

- 防止因错误检测 SLRU 日志的回卷点而可能造成的数据丢失（Noah Misch）

  回卷点通常位于页面中间，必须将其取整到页边界，但以前没有正确做到这一点。只有当安装实例距离 SLRU 溢出不到一页时，才可能出现问题，而这在正常运行的系统中不太可能发生。如果确实发生，会表现为后续出现 “apparent wraparound” 或 “could not access status of transaction” 错误。

- 修复 WAL 发送进程进程为逻辑解码发送新快照时的内存泄漏（Amit Kapila）

- 修复 WAL 发送进程，使其在终止复制后接受其他命令（Jeff Davis）

- 确保检测热备后端与启动（WAL 应用）进程之间的死锁（Fujii Masao）

  启动进程没有运行死锁检测代码，因此当启动进程最后加入循环等待时，死锁可能永远不会被识别。

- 确保 postmaster 开始 “smart” 或 “fast” 关闭流程时，清理尚未处理的后台工作进程请求（Tom Lane）

  以前存在一种竞争条件：如果子进程恰好在关闭前请求了后台工作进程，就可能无限期等待，导致关闭无法完成。

- 修复解析 `recovery_target_xid` 值时的可移植性问题（Michael Paquier）

  目标 XID 可能有 64 位宽，但以前使用 `strtoul()` 解析，导致在 `long` 为 32 位的平台（例如 Windows）上行为异常。

- 避免在独立后端中尝试使用并行索引构建（Yulin Pei）

- 允许索引访问方法支持内含列，而不必支持多个键列（Tom Lane）

- 避免对具有非严格反序列化函数的聚合进行并行聚合时发生断言失败（Andrew Gierth）

核心 PostgreSQL 中不存在此类聚合函数，但 PostGIS 等某些扩展提供了此类函数。不过，在未启用断言的构建中，这一错误是无害的。

- 避免 `pg_get_functiondef()` 检查带有 `TRANSFORM` 选项的函数时发生断言失败（Tom Lane）
- 修复 PL/pgSQL `CALL` 语句中错误的数据结构分配（Tom Lane）

  PL/pgSQL 过程中的 `CALL` 调用另一个具有 OUT 参数的过程时，如果被调用过程执行了 `COMMIT` 或 `ROLLBACK`，就会失败。
- 在 psql 中，重新允许在 `\connect` 命令的 *`connection_string`* 参数中包含密码（Tom Lane）

  这以前是可以的，但最近的一项缺陷修复导致密码被忽略（从而提示用户输入密码）。
- 修复 psql 的 `\help` 命令中的多种缺陷（Kyotaro Horiguchi，Tom Lane）

  `\help` 带有两个参数词时，无法仅使用第一个词找到命令说明，例如 `\help reset all` 本应显示 `RESET` 的帮助，但却没有。另外，`\help` 经常未在应该调用分页器时调用它，也会泄漏内存。
- 在 pg_dump 中，确保恢复脚本以发布所有者的身份运行 `ALTER PUBLICATION ADD TABLE` 命令，并同样以分区索引所有者的身份运行 `ALTER INDEX ATTACH PARTITION` 命令（Tom Lane）

  以前，这些命令由启动恢复脚本的角色运行；这通常能够正常工作，但在边界情况下，该角色可能没有足够权限。
- 修复 pg_dump，使其处理扩展初始权限中的 `WITH GRANT OPTION`（Noah Misch）

  如果扩展脚本创建一个对象，并授予其带转授权选项的权限，而用户之后撤销了这些权限，pg_dump 就会生成不正确的 SQL 来重现这种情况。（目前，即使有扩展这样做，数量也很少。）
- 在 pg_rewind 中，确保回退备库时计入所有 WAL（Ian Barwick，Heikki Linnakangas）
- 在 pgbench 中，禁止变量名以数字开头（Fabien Coelho）

  这可以避免尝试在时间戳字面值中替换变量，因为时间戳可能包含 `12:34` 这样的字符串。
- 在某些客户端程序的连接失败错误消息中报告正确的数据库名称（Álvaro Herrera）

  如果数据库名称采用默认值，而非通过命令行给出，pg_dumpall、pgbench、oid2name 和 vacuumlo 就会在连接失败后产生误导性的错误消息。
- 修复 `contrib/auto_explain` 中的内存泄漏（Japin Li）

  生成 `EXPLAIN` 输出时消耗的内存，直到当前事务结束（对于顶层语句）或外层语句结束（对于嵌套语句）才会释放。启用 `log_nested_statements` 时，这尤其容易造成问题。
- 在 `contrib/postgres_fdw` 中，避免删除用户映射或外部服务器对象时泄漏已打开的远程服务器连接（Bharath Rupireddy）

依赖已删除用户映射或外部服务器的打开连接已无法再被引用，但以前仍会将它们保留到本地会话结束。

- 在 contrib/pgcrypto 中，检查 OpenSSL EVP 函数返回的错误（Michael Paquier）

  我们并不真正预期这里会出现错误，但此项修改消除了静态分析工具的警告。

- 使 contrib/pg_prewarm 在预热完成前数据库集簇被关闭时更加稳健（Tom Lane）

  以前，autoprewarm 会只用截至当时已成功加载的块号重写其状态文件，因而可能在很大程度上使下次启动时的预热功能失效。现在改为在初始加载过程完成前，禁止更新状态文件。

- 在 contrib/pg_trgm 的 GiST 索引支持中，避免恰好对两个索引项调用 picksplit 这一罕见情况下的崩溃（Andrew Gierth，Alexander Korotkov）

- 修复 contrib/pg_prewarm 和 contrib/postgres_fdw 中错误的超时计算（Alexey Kondratov，Tom Lane）

  contrib/pg_prewarm 的 autoprewarm 父进程中的主循环，将所需睡眠时间低估为实际值的千分之一，导致 CPU 消耗远高于预期。等待远程服务器返回结果时，contrib/postgres_fdw 则将所需超时时间高估为实际值的 1000 倍（不过，60 秒的上限缓解了这一错误）。

  这两个错误都源于将秒和微秒错误地转换为毫秒。引入新的 API TimestampDifference Milliseconds()，使今后更容易正确完成这一转换。

- 改进 configure 在 macOS 上选择 PG_SYSROOT 的启发式方法（Tom Lane）

  当 Xcode 比底层操作系统更新时，新方法更有可能产生理想结果。选择与操作系统版本不匹配的 sysroot，可能生成无法工作的可执行文件。

- 在 macOS 上构建时，在链接步骤和编译步骤中都指定 -isysroot（James Hilliard）

  这同样可以改善 Xcode 与操作系统版本不同步时的结果。

- 修复 JIT 编译，使其兼容 LLVM 11 和 LLVM 12（Andres Freund）

- 修复 JIT 表达式编译中对布尔变量引用的潜在错误处理（Andres Freund）

  尚未看到可归因于此问题的实际报告，但它似乎很可能在某些架构上造成问题。

- 修复使用 ICU 68 及更新版本时的编译失败（Tom Lane）

- 避免在创建分区索引期间，以 NULL 源指针和零计数调用 memcpy()（Álvaro Herrera）

  虽然尚未发现此类调用本身会造成问题，但某些编译器假定 memcpy() 的参数永不为 NULL，这可能导致对附近代码进行错误的优化。

- 将时区数据文件更新到 tzdata 2021a 版，以反映俄罗斯（伏尔加格勒时区）和南苏丹的夏令时法规变更，并修正澳大利亚、巴哈马、伯利兹、百慕大、加纳、以色列、肯尼亚、尼日利亚、巴勒斯坦、塞舌尔和瓦努阿图的历史数据。

  特别是，Australia/Currie 时区经过修正后，已与 Australia/Hobart 完全相同。

E.13. 发布版本 11.10

发布日期：. 2020-11-12

本次发布包含来自 11.9 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.13.1. 迁移到版本 11.10

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.6 的版本升级，请参见第 E.17 节。

E.13.2. 变更

- 禁止在索引表达式和物化视图查询中执行 `DECLARE CURSOR ... WITH HOLD` 以及触发延迟触发器（Noah Misch）

  这实际上是“安全受限操作”沙箱机制中的一个漏洞。有权创建非临时 SQL 对象的攻击者可以利用此漏洞，以超级用户身份执行任意 SQL 代码。

  PostgreSQL 项目感谢 Etienne Stalmans 报告此问题。（CVE-2020-25695）

- 修复 pg_dump、pg_restore、clusterdb、reindexdb 和 vacuumdb 对复杂连接字符串参数的使用（Tom Lane）

  pg_dump 和 pg_restore 的 `-d` 参数，以及其他上述程序的 `--maintenance-db` 参数，可以是包含多个连接参数的“连接字符串”，而不仅仅是数据库名称。当这些程序需要发起额外连接时，例如进行并行处理或处理多个数据库时，连接字符串会被遗忘，额外连接只会使用基本连接参数（数据库名称、主机、端口和用户名）。如果连接字符串包含其他必要信息，例如非默认的 SSL 或 GSS 参数，这可能导致连接失败。更糟的是，连接可能成功建立，却未按预期加密，或者可能遭受本应由指定连接参数防止的中间人攻击。（CVE-2020-25694）

- 当 psql 的 `\connect` 命令复用连接参数时，确保复用先前连接字符串中所有未被覆盖的参数（Tom Lane）

  这可以避免因遗漏相关参数（例如非默认的 SSL 或 GSS 选项）而导致重新连接失败。更糟的情况是，重新连接可能成功，却未按预期加密，或者可能遭受本应由指定连接参数防止的中间人攻击。这与刚才提到的 pg_dump 等程序的问题基本相同，只是 psql 的行为更复杂，因为用户可能有意覆盖某些连接参数。（CVE-2020-25694）

- 防止 psql 的 `\gset` 命令修改具有特殊处理方式的变量（Noah Misch）

  不带前缀的 `\gset` 会覆盖服务器要求其设置的任何变量。因此，被攻陷的服务器可以设置 `PROMPT1` 等受到特殊处理的变量，从而能够在用户会话中执行任意 shell 代码。

  PostgreSQL 项目感谢 Nick Cleaton 报告此问题。（CVE-2020-25696）

- 防止并发截断 SLRU 日志可能造成的数据丢失（Noah Misch）

  这一罕见问题会表现为后续出现 “apparent wraparound” 或 “could not access status of transaction” 错误。

- 确保在检查点期间对 SLRU 目录正确执行 fsync（Thomas Munro）

  这可以防止后续操作系统崩溃时可能发生的数据丢失。

- 修复对具有 BYPASSRLS 属性的用户执行 ALTER ROLE 的行为（Tom Lane，Stephen Frost）

  只有超级用户才可以修改 BYPASSRLS 属性，但其他 ALTER ROLE 操作（例如修改密码）应该只需通过普通权限检查就可以执行。之前的代码错误地将对此类角色的所有修改都限制为只能由超级用户执行。

- 确保 ALTER TABLE ONLY ... ENABLE/DISABLE TRIGGER 不递归处理子表（Álvaro Herrera）

  以前，ONLY 标志被忽略了。

- 修复结合继承使用的 CREATE TABLE LIKE 中对表达式的处理（Tom Lane）

  如果 CREATE TABLE 命令同时使用 LIKE 和传统继承，来自 LIKE 父表的 CHECK 约束及表达式索引中的列引用往往会被错误编号，导致错误结果或奇怪的错误消息。在具有生成列特性的分支中，GENERATED 表达式也可能出现同样的问题。

- 禁止对分区表执行 DROP INDEX CONCURRENTLY（Álvaro Herrera，Michael Paquier）

  这种情况本来就会失败，但错误消息令人困惑。

- 允许对自引用视图成功执行 LOCK TABLE（Tom Lane）

  以前会抛出关于无限递归的错误，但似乎没有必要禁止这种情况。

- 修复 to_date() 和 to_timestamp() 将负年份转换为公元前日期时相差一年的错误（Dar Alathar-Yemen，Tom Lane）

  另外，使负年份与显式 “BC” 标记相抵消，得到 AD 年份。

- 确保 archive_mode 设为 always 时，备库会归档 WAL 时间线历史文件（Grigory Smolkin，Fujii Masao）

  这一疏漏可能导致后续的 PITR 恢复尝试失败。

- 修复逻辑复制工作进程中的 “cache lookup failed for relation 0” 错误（Tom Lane）

  实际影响很小，因为这种失败不太可能发生；即使发生，工作进程也只会退出并被重启。

- 防止逻辑复制工作进程发送多余的 ping 请求（Tom Lane）

- 在 “smart” 关闭期间，在所有客户端（前台）会话结束之前，不终止后台进程（Tom Lane）

  以前的行为会破坏并行查询处理，因为 postmaster 会终止并行工作进程，并拒绝启动新的并行工作进程。它还会使自动清理停止工作；如果仍存活的客户端会话进行了大量数据变更，就可能造成严重的长期影响。

- 避免 postmaster 处理信号时递归消耗栈空间（Tom Lane）

  已有观察表明，大量使用并行处理时，过多请求创建并行工作进程的并发信号会导致 postmaster 崩溃。

- 避免因 SIGQUIT 退出时运行 atexit 处理函数（Kyotaro Horiguchi，Tom Lane）

  大多数服务器进程早已遵循这一做法，但归档进程被遗漏了。仍在等待客户端启动包的后端也处理有误。

- 避免错误优化引用看似为常量的分组列的子查询限制条件（Tom Lane）

  如果某个“常量”子查询输出列是仅出现在部分分组集中的分组列，它就并非真正的常量。

- 避免 SQL 函数内联改变可能用于 hash 处理的子计划比较表达式的形态时发生失败（Tom Lane）

- 构建或重建索引时，容许因并发更新而出现新的 HOT 链（Anastasia Lubennikova，Álvaro Herrera）

  这一疏漏可能导致“failed to find parent tuple for heap-only tuple”错误。

- 修复索引条件无法满足时并行 B-树索引扫描的失败（James Hunter）

- 确保数据插入 BRIN 索引之前，先将其从 TOAST 表示还原（Tomas Vondra）

  索引条目本不应包含指向行外数据的 TOAST 指针，但 BRIN 没有遵守这一要求。这可能导致“missing chunk number 0 for toast value NNN”等错误。（如果现有索引出现这种错误，执行 `REINDEX` 应该就足以修复。）

- 在 BRIN 索引扫描期间正确处理并发取消汇总（Alexander Lakhin，Álvaro Herrera）

  以前，如果某个页范围恰好在不合适的时刻被取消汇总，索引扫描就可能错误地抛出表示索引损坏的错误。

- 修复多列 GIN 索引扫描中罕见的“lost saved point in index”错误（Tom Lane）

- 修复 `pg_hba_file_rules` 视图中不具可移植性的 `getnameinfo()` 用法（Tom Lane）

  在 FreeBSD 11 以及可能的其他平台上，此错误会使该视图的 `address` 和 `netmask` 列始终为空。

- 避免启动并行工作进程时 `debug_query_string` 为 NULL 所导致的崩溃（Noah Misch）

- 修复事件触发器监控 `ALTER TABLE` 操作时的释放后使用隐患（Jehan-Guillaume de Rorthais）

- 修复关于移动聚合数据类型不一致的不正确错误消息（Jeff Janes）

- 避免并行工作进程报告很长的错误消息时卡死（Vignesh C）

- 避免通过共享内存队列传输很大的数据载荷时发生不必要的失败（Markus Wanner）

- 修复 JIT 代码生成中对模板函数属性的不正确处理（Andres Freund）

  已证实这会在 `s390x` 上导致崩溃，而且其他平台上很可能也存在其他情况。

- 修复 RLS 策略相关的关系缓存内存泄漏（Tom Lane）

- 修复 SIGHUP 处理判定某个 GUC 变量的新值必须重启才能应用时出现的小量内存泄漏（Tom Lane）

- 修复 PL/pgsql 的 `CALL` 处理中的内存泄漏（Pavel Stehule，Tom Lane）

- 使 libpq 支持 `.pgpass` 文件中任意长度的行（Tom Lane）

  这主要用于允许将很长的安全令牌用作密码。

- 在 Windows 版 libpq 中，每个进程只调用一次 `WSAStartup()`，完全不调用 `WSACleanup()`（Tom Lane，Alexander Lakhin）

  以前，libpq 在连接开始时调用 `WSAStartup()`，在连接清理时调用 `WSACleanup()`。但调用 `WSACleanup()` 似乎会干扰程序的其他操作；特别是，我们观察到过无法将预期输出写入 stdout 的罕见失败。省略该调用似乎没有不良影响，因此予以省略。（这还消除了程序执行一连串数据库连接时，反复加载和卸载 DLL 所造成的性能问题。）

- 修复 ecpg 库在 Windows 上的逐线程初始化逻辑（Tom Lane，Alexander Lakhin）

  多线程 ecpg 应用可能因锁定不正确而偶发行为异常。

- 在 Windows 上，使 psql 以文本模式而非二进制模式读取反引号命令的输出（Tom Lane）

  这可以确保正确处理换行符。

- 确保 pg_dump 收集扩展配置表的逐列信息（Fabrízio de Royes Mello，Tom Lane）

  未能这样做会导致指定 `--inserts` 时崩溃，或在使用 `COPY` 重载表数据时生成指定不完整（虽然通常正确）的 `COPY` 命令。

- 使 pg_upgrade 检查目标集群中的表空间目录是否已存在（Bruce Momjian）

- 修复 `contrib/pgcrypto` 中潜在的内存泄漏（Michael Paquier）

- 在 `contrib/pgcrypto` 中增加对一种不太可能发生的失败情况的检查（Daniel Gustafsson）

- 修复最近添加的 `timetz` 测试用例，使其在美国未实行夏令时期间也能正常工作（Tom Lane）

- 将时区数据文件更新到 tzdata 2020d 版，以反映斐济、摩洛哥、巴勒斯坦、加拿大育空地区、麦夸里岛和凯西站（南极洲）的夏令时法规变更，并修正法国、匈牙利、摩纳哥和巴勒斯坦的历史数据。

- 将我们使用的时区库副本同步到 IANA tzcode 2020d 版（Tom Lane）

  这纳入了上游将 zic 默认输出选项从 “fat” 改为 “slim” 的修改。对我们而言，这只是表面上的变化，因为在 v13 之前的分支中，我们仍然选择 “fat” 模式。此项修改还确保 `strftime()` 只在失败时才改变 `errno`。

E.14. 发布版本 11.9

发布日期：. 2020-08-13

本次发布包含来自 11.8 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.14.1. 迁移到版本 11.9

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.6 的版本升级，请参见第 E.17 节。

E.14.2. 变更

- 在逻辑复制 WAL 发送进程和应用工作进程中设置安全的 `search_path`（Noah Misch）

发布者或订阅者数据库中的恶意用户，都可能导致以运行复制的角色身份执行任意 SQL 代码，而该角色通常是超级用户。这里的一部分风险与 CVE-2018-1058 中所述的风险相同，本补丁通过确保复制发送进程和接收进程以空的 `search_path` 设置运行来缓解这些风险。（与 CVE-2018-1058 一样，这一改动可能使被复制表的 DDL 中未充分限定的名称出现问题。）其他风险则是复制不可信角色所属对象所固有的；我们最多只能在文档中说明存在需要考虑的风险。（CVE-2020-14349）

- 提高 contrib 模块安装脚本的安全性（Tom Lane）

  如果攻击者能够在扩展的目标模式或某个前置扩展的模式中创建对象，就可以针对扩展安装脚本实施与 CVE-2018-1058 中所述类似的攻击。由于安装扩展通常需要超级用户权限，这可能成为获取超级用户权限的一条途径。为缓解这一风险，更谨慎地设置运行安装脚本时使用的 `search_path`；在脚本中禁用 `check_function_bodies`；并修正一些 contrib 模块中用于调整系统目录的查询，确保它们安全。同时提供文档，帮助第三方扩展作者编写安全的安装脚本。这并不是完整的解决方案；如果安装时不够谨慎，依赖其他扩展的扩展仍可能面临风险。（CVE-2020-14350）

- 修复分区剪枝中的边界情况（Etsuro Fujita，Dmitry Dolgov）

  存在多个分区键列时，如果某些列没有用于约束它们的 `WHERE` 子句，或具有多个约束子句，剪枝测试的生成就可能行为异常。这可能导致服务器崩溃、错误的查询结果或断言失败。

- 修复逐分区嵌套循环连接内侧的参数化 BitmapAnd 和 BitmapOr 索引扫描的构建（Tom Lane）

  如果计划中的这种扫描需要使用连接外侧的值，通常会在执行时崩溃。

- 在逻辑复制 WAL 发送进程中，修复发送保活消息后未能发送反馈消息的问题（Álvaro Herrera）

  使用内置逻辑复制时，这个问题相对较小，因为内置 WAL 接收进程本来就会相当频繁地发送反馈回复（从而清除错误状态）。但对于 pglogical 等其他复制系统，它会造成显著的性能问题。

- 修复逻辑复制订阅者中特定列 `UPDATE` 触发器的触发（Tom Lane）

  代码忽略了发布者和订阅者的表中列编号可能不同的情况，因此如果列编号确实不同，就可能错误判断应触发哪些触发器。

- 在 `pg_replication_slot_advance()` 期间更新最旧的 xmin 和 LSN 值（Michael Paquier）

  此函数以前未能这样做，可能导致手动推进复制槽后无法清理资源（例如移除不再需要的 WAL 段）。

- 修复 `ts_headline()` 执行缓慢的问题（Tom Lane）

  上一批小版本中添加的短语搜索修复，可能使 `ts_headline()` 处理长文档时耗费极不合理的时间；更糟的是，在出现问题的循环中无法取消查询。

- 确保 `repeat()` 函数可以被查询取消中断（Joe Conway）

- 修复 `pg_current_logfile()`，使其结果在 Windows 上不包含回车符（`\r`）（Tom Lane）

- 确保 pg_read_file() 及相关函数一直读取到 EOF（Joe Conway）

  以前，如果未给出要读取的具体数据长度，这些函数就会在读到 stat() 报告的文件长度时停止。这对管道和其他虚拟文件并不适用。

- 修复对 numeric 类型列进行并行聚合时对 NaN 输入的错误处理（Tom Lane）

  如果某些部分聚合工作进程只找到 NaN，而其他工作进程只找到非 NaN 值，结果就会被错误合并，可能导致整体结果错误（即本应为 NaN 时却不是）。

- 拒绝大于 24 小时的日内时间值（Tom Lane）

  日期时间输入代码的本意是允许 “24:00:00” 或等价的 “23:59:60”，但不允许更大的值。然而，范围检查的代码有误，会接受 “23:59:60.*nnn*”，其中小数秒 *nnn* 可以非零。对于时间戳值，这会使其跨入下一天的第一秒。对于 time 和 timetz 值，存储的值实际上会超过 24 小时，从而导致转储/重载失败，并可能引发其他异常行为。

- 撤销 EXPLAIN 非文本输出格式中为索引名称添加双引号的行为（Tom Lane，Euler Taveira）

- 修复 EXPLAIN 对使用 Gather Merge 节点的计划中并行工作进程的资源使用统计，尤其是缓冲区访问统计（Jehan-Guillaume de Rorthais）

- 修复 ALTER TABLE 中重新验证约束的时机（David Rowley）

  如果 ALTER TABLE 需要完全重写表内容（例如更改列的数据类型），又需要扫描表以重新验证外键或 CHECK 约束，有时会按错误顺序执行，导致 “could not read block 0 in file "base/nnnnn/nnnnn": read only 0 of 8192 bytes” 等奇怪错误。

- 规避 pg_subscription.subslotname 和 pg_subscription_rel.srsublsn 上不正确的非空标记（Tom Lane）

  引导目录数据错误地将这两个目录列标记为始终非空。在现有安装实例中不容易修正这一错误（不过 v13 及更新版本将具有正确标记）。主要依赖该标记正确性的地方是启用 JIT 的元组解构，因此让它显式忽略这两列的标记。另外，调整某些未检查是否为空就访问 srsublsn 的 C 代码；由此引发崩溃的可能性很低，但或许并非不可能。

- 处理附加到 FROM 子句中未扁平化的子 SELECT 上的限制子句里的 LATERAL 引用（Tom Lane）

  这一疏漏可能导致查询执行时断言失败或崩溃。

- 避免认为从未分析过的外部表具有零个元组（Tom Lane）

  这主要影响规划器对 GROUP BY 将产生的分组数量的估计。

- 移除 BRIN 索引取消汇总过程中关于 “leftover placeholder tuple” 的错误警告（Álvaro Herrera）

  取消清理后，这种情况可以正常出现，因此对此发出警告会产生过多噪声。

- 修复 “共享文件集” 临时文件的表空间选择（Magnus Hagander，Tom Lane）

  如果 temp_tablespaces 为空，或显式指定数据库的主表空间，这些文件会被放到 pg_default 表空间，而非预期的数据库主表空间。

- 修复 WAL 一致性检查期间屏蔽 SP-GiST 索引页面内容时的边界情况错误（Alexander Korotkov）

  启用 `wal_consistency_checking` 时，这可能导致错误的失败报告。

- 改进服务器 `buffile` 模块中的错误处理（Thomas Munro）

  修复某些无法区分 I/O 错误与到达 EOF，或完全未报告 I/O 错误的情况。另外，在适当位置增加块号、字节数等详细信息。

- 修复 `SERIALIZABLE` 隔离模式中的冲突检查异常（Peter Geoghegan）

  如果一个并发插入的元组被另一个并发事务更新，且两个元组版本对当前事务的快照都不可见，序列化冲突检查就可能错误判断该元组是否与当前事务的结果有关。这可能允许本应因序列化错误而失败的可序列化事务提交。

- 避免将已失效的 B-树索引条目反复标记为失效（Masahiko Sawada）

  虽然功能上无害，但在启用了校验和或 `wal_log_hints` 时，这会产生无用的 WAL 流量。

- 避免非独占备份期间启用了 JIT 编译时，在备份清理阶段出现问题（Robert Haas）

- 修复某些代码路径修改 `pg_control` 前未获取正确锁的问题（Nathan Bossart，Fujii Masao）

  这一疏漏可能使 `pg_control` 以不一致的校验和写出，导致之后出现问题，包括在下次 `pg_control` 更新之前崩溃时无法重启数据库。

- 修复 `currtid()` 和 `currtid2()` 中的错误（Michael Paquier）

  这些函数没有文档记录，且仅由很早版本的 ODBC 驱动使用。它们包含编码错误，可能导致崩溃，或在应用于没有存储的关系时产生 “could not open file” 等令人困惑的错误消息。

- 避免持有自旋锁时调用 `elog()` 或 `palloc()`（Michael Paquier，Tom Lane）

  复制槽相关逻辑有几处违反了这一编码规则。虽然出现问题的可能性很低，但被调用函数一旦出错，就会导致自旋锁卡住。

- 修复逻辑复制订阅者中的断言，以允许使用 `REPLICA IDENTITY FULL`（Euler Taveira）

  这只是一个不正确的断言，因此不影响标准的生产构建。

- 在 pg_dump 和 pg_basebackup 中正确报告磁盘空间不足错误（Justin Pryzby，Tom Lane，Álvaro Herrera）

  某些代码路径可能产生 “could not write file: Success” 之类不合理的报告。

- 修复同时具有表级权限和逐列权限的表的并行恢复（Tom Lane）

  必须先应用表级权限授予，但并行恢复未能可靠地保证这一顺序，可能导致 “tuple concurrently updated” 错误，或丢失某些逐列权限授予。修复方式是在归档文件中为这些条目添加依赖链接，因此必须使用修正后的 pg_dump 重新转储，才能确保问题不再发生。

- 确保 pg_upgrade 运行时，将目标集群中的 `vacuum_defer_cleanup_age` 设为零（Bruce Momjian）

如果目标集群的配置已将 vacuum_defer_cleanup_age 设为非零值，就会使系统目录的冻结无法正常进行，进而使升级以令人困惑的方式失败。现在确保在升级期间覆盖此类设置。

- 修复 pg_recvlogical，使其在退出前读取完待处理消息（Noah Misch）

  否则，复制发送进程可能检测到发送失败并退出，而未对复制槽的 LSN 位置进行预期的最终更新。这会导致下次连接后重新传输数据。另外，还可能遗漏在 pg_recvlogical 想要消费的最后一条数据之后发送的错误消息。

- 修复 pg_rewind 对源数据目录中刚删除的文件的处理（Justin Pryzby，Michael Paquier）

  使用在线源数据库时，可能发生并发文件删除；但如果删除发生在看到文件目录项与使用 stat() 检查它之间，pg_rewind 就会处理失当。

- 使 pg_test_fsync 在 Windows 上使用二进制 I/O 模式（Michael Paquier）

  以前，它以文本模式写入测试文件，无法准确反映 PostgreSQL 的实际用法。

- 修复 contrib/amcheck，使其不再因已删除的索引页为空而报错（Alexander Korotkov）

  这种状态在 WAL 重放期间是正常的。

- 修复 contrib/dblink 未能正确初始化本地状态的问题（Joe Conway）

  在特定条件组合下，这可能导致 dblink_close() 发出意外的远程 COMMIT。

- 修复 contrib/pgcrypto 对 deflate() 的错误使用（Tom Lane）

  由于未正确遵循 zlib 的 API 要求，pgp_sym_encrypt 函数可能产生不正确的压缩数据。尚未收到使用原版 zlib 时出现此错误的报告，但使用 IBM 的 zlibNX 实现时可以观察到。

- 修复 contrib/pgcrypto 的 pgp_sym_decrypt 函数中解压缩逻辑的边界情况（Kyotaro Horiguchi，Michael Paquier）

  压缩流可以合法地以空包结束，但解压缩器未能处理这种情况，并会报告数据损坏。

- 使用 POSIX 标准的 strsignal() 代替 BSD 风格的 sys_siglist[]（Tom Lane）

  这避免了使用很新版本的 glibc 时构建失败。

- 支持使用 Microsoft Visual Studio 2015 及更新版本构建我们的 NLS 代码（Juan José Santamaría Flecha，Davinder Singh，Amit Kapila）

- 避免源代码树上方数级目录中存在名为 configure 的文件时，MSVC 安装脚本可能发生的失败（Arnold Müller）

  这可能使通过查找 configure 来识别源代码树顶层的某些逻辑产生混淆。

E.15. 发布版本 11.8

发布日期：. 2020-05-14

本次发布包含来自 11.7 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.15.1. 迁移到版本 11.8

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.6 的版本升级，请参见第 E.17 节。

E.15.2. 变更

- 将 `ALTER TABLE ... SET STORAGE` 传播到索引（Peter Eisentraut）

  非表达式索引列在创建时一直会复制对应表列的 `attstorage` 属性。现在执行 `ALTER TABLE ... SET STORAGE` 时也更新它们，以保持一致。

- 保留由 `ALTER TABLE` 重写的索引的 `indisclustered` 设置（Amit Langote，Justin Pryzby）

  以前，`ALTER TABLE` 会丢失哪个索引曾用于 `CLUSTER` 的信息。

- 保留由 `ALTER TABLE` 重写的索引的复制标识属性（Quan Zongliang，Peter Eisentraut）

- 在 `DROP OWNED BY` 期间更早地锁定对象（Álvaro Herrera）

  这可以避免另一个会话同时删除部分相同对象时，因竞争条件而发生的失败。

- 修复 `CREATE ROLE ... IN ROLE` 的错误情况处理（Andrew Gierth）

  某些错误情况会被报告为“unexpected node type”等消息，而非预期的消息。

- 确保分离分区时移除所有从原父表克隆的触发器（Justin Pryzby）

- 确保分区表上的唯一索引与分区键的相等语义一致（Guancheng Luo）

  这只会在索引操作符类采用不寻常的相等定义时成为问题，但理论上仍是错误，因此增加检查。

- 确保 `pg_read_all_stats` 角色的成员可以按预期读取所有统计视图（Magnus Hagander）

  `pg_stat_progress_*` 视图的底层函数以前没有遵守这一要求。

- 修复 `information_schema.triggers` 视图的性能退化（Tom Lane）

  此补丁重新定义了该视图，使约束表名的外层 `WHERE` 子句可以下推到视图内部，从而只针对目标表的触发器进行计算，而非数据库中的所有触发器。在具有大量触发器的数据库中，这会显著提高此类查询的速度。由于 v11 之前就是这样工作的，这是一处潜在的性能退化。受此问题影响的用户可以替换视图定义来修复它（或许也可以直接删除并重新安装整个 `information_schema` 模式）。

- 修复全文检索，使其正确处理短语搜索之上的 NOT（Tom Lane）

  `!(foo<->bar)` 等查询在通过 GiST 或 GIN 索引搜索执行时，无法找到匹配行。

- 修复短语搜索包含同时具有前缀匹配和权重限制的项时的全文检索（Tom Lane）

- 修复 `ts_headline()`，使其处理短语查询时更恰当地选择摘要片段（Tom Lane）

- 修复 gin_fuzzy_search_limit 处理中的缺陷（Adé Heyward，Tom Lane）

  较小的 gin_fuzzy_search_limit 值可能导致同一个索引页被意外重复扫描多次，从而运行异常缓慢。另一个代码路径则完全没有应用预期的过滤，可能返回过多的值。

- 允许 circle 类型的输入按文档所述接受 “(*x*,*y*),*r*” 格式（David Zhang）

- 使 get_bit() 和 set_bit() 函数能够处理长于 256MB 的 bytea 字符串（Movead Li）

  由于位编号参数只有 int4，无法使用这些函数访问长 bytea 值前 256MB 之外的位。我们将在 v13 中将该参数扩为 int8，但在此之前，先允许这些函数处理长 bytea 值的起始子串。

- 在 pg_ls_waldir() 及相关函数中忽略文件不存在错误（Tom Lane）

  这可以防止文件在看到其目录项与尝试对其执行 stat() 之间被移除时，因竞争条件而发生的失败。

- 避免 pg_ls_dir()、pg_timezone_names()、pg_tablespace_databases() 及相关函数可能泄漏目录的打开文件描述符（Justin Pryzby）

- 修复多态函数的类型解析，使其仅给定 anyrange 输入时，也能正确推断 anyarray 输出的实际类型（Tom Lane）

- 避免 hash 子计划的 hash 表在多次执行之间泄漏（Andreas Karlsson，Tom Lane）

  如果查询重复执行 hash 子计划的次数足够多，这一错误可能导致严重的内存膨胀。

- 避免 REINDEX 被会话关闭信号终止时不太可能发生的崩溃（Tom Lane）

- 修复分区表中发生约束违反错误之后低概率出现的崩溃（Andres Freund）

- 防止 EXPLAIN 输出可能不正确的哈希连接表统计信息（Konstantin Knizhnik，Tom Lane，Thomas Munro）

- 修复 VACUUM VERBOSE 对堆截断步骤耗时的报告（Tatsuhito Kasahara）

- 修复 VACUUM VERBOSE 输出中可能低估已删除 B-树索引页数量的问题（Peter Geoghegan）

- 修复 B-树索引中对最早删除页面的错误记录（Peter Geoghegan）

  这可能使有关 VACUUM 何时可以跳过索引清理扫描的判断出现细微错误，不过似乎未必有明显的用户可见影响。

- 确保在所有读写时间线历史文件的代码路径中报告 TimelineHistoryRead 和 TimelineHistoryWrite 等待状态（Masahiro Ikeda）

- 避免在进程的 PS 状态中可能两次显示 “waiting” （Masahiko Sawada）

- 避免自动清理尝试访问刚删除的临时模式时发生失败（Tom Lane）

  只有超级用户手动删除临时模式时才会出现这一隐患；这虽然不是常规做法，但也应该能够正常工作。

- 避免崩溃恢复期间过早回收 WAL 段（Jehan-Guillaume de Rorthais）

  在崩溃恢复期间变为可归档状态的 WAL 段，可能在尚未归档的情况下就被回收。

- 避免归档恢复期间扫描无关的时间线（Kyotaro Horiguchi）

  这可以消除许多从归档存储中获取不存在的 WAL 文件的尝试，在归档访问缓慢时很有帮助。

- 移除逻辑解码中不正确的 “subtransaction logged without previous top-level txn record” 错误检查（Arseny Sher，Amit Kapila）

  在多种场景中都可以正常出现这种情况，因此移除该检查。

- 确保在失败代码路径中释放复制槽的 `io_in_progress_lock`（Pavan Deolasee）

  这可能使 WAL 发送进程之后卡在等待该锁的状态。

- 修复同步备库管理中的竞争条件（Tom Lane）

  更改 `synchronous_standby_names` 设置期间，存在一个时间窗口，可能错误判断是否可以放行正在等待同步提交的事务。如果一个 WAL 发送进程进程退出后立即被另一个替代，也存在作出类似错误判断的隐患。

- 确保备库上的 `nextXid` 不会倒退（Eka Palamadai）

  此竞争条件可能使错误的热备反馈消息被发送回主库，从而可能让主库过早执行 `VACUUM`。

- 为几个错误报告补上缺失的 SQLSTATE 值（Sawada Masahiko）

- 修复 PL/pgSQL，使其可靠地拒绝将事件触发器函数作为普通函数执行（Tom Lane）

- 修复 libpq 使用 `sslmode=verify-full` 时的内存泄漏（Roman Peshkurov）

  连接启动期间的证书验证可能泄漏一些内存。如果客户端进程在其生命周期内打开大量数据库连接，这就会成为问题。

- 修复 ecpg，使其在所有平台上都将仅包含 “-” 的参数解释为 “从 stdin 读取”（Tom Lane）

- 允许对 psql 的 `\gx` 命令的文件名参数进行 Tab 补全（Vik Fearing）

- 为 pg_dump 增加对 `ALTER ... DEPENDS ON EXTENSION` 的支持（Álvaro Herrera）

  pg_dump 以前忽略了以这种方式添加的依赖关系，导致它们在转储/恢复或 pg_upgrade 期间丢失。

- 修复 pg_dump，使其转储 RLS 策略对象上的注释（Tom Lane）

- 在 pg_dump 中，将事件触发器的恢复推迟到最后（Fabrízio de Royes Mello，Hamid Akhtar，Tom Lane）

  这尽可能降低了事件触发器干扰其他对象恢复的风险。

- 使 pg_verify_checksums 跳过属于其他 PostgreSQL 大版本的表空间子目录（Michael Banck，Bernd Helmle）

  这些子目录实际上不属于我们的数据库集簇，因此不得处理它们。

- 在 pg_verify_checksums 及相关程序中忽略 `pg_internal.init` 的临时副本（Michael Paquier）

- 修复 createdb 工具中 `--encoding`、`--lc-ctype` 和 `--lc-collate` 值的引号处理（Michael Paquier）
- 直接调用 `contrib/lo` 的 `lo_manage()` 函数，而非作为触发器调用时，会发生崩溃（Tom Lane）
- 在 `contrib/ltree` 中，防止 `ltree` 和 `lquery` 长度字段溢出（Nikita Glukhov）
- 规避 oldest_xact 值超过 2^31-1 时 `contrib/pageinspect` 的 `bt_metap()` 函数发生的失败（Peter Geoghegan）

  此类 XID 现在会被报告为负整数，这虽然不理想，但总比抛出错误好。v13 将把输出参数扩为 `int8`，以提供更合理的报告。
- 修复 `contrib/sepgsql` 中的缓存引用泄漏（Michael Luo）
- 避免在 Windows 上处理 Unix 风格区域设置名称时发生失败（Juan José Santamaría Flecha）
- 在 configure 期间，如果 pkg-config 可用，则使用它定位 libxml2（Hugh McMaster，Tom Lane，Peter Eisentraut）

  如果 pkg-config 不存在，或不包含 libxml2 的信息，我们仍像以前一样查询 xml2-config。

  某些构建过程通过将非默认版本的 xml2-config 放入 `PATH`，来让 PostgreSQL 使用该版本的 libxml2；此项修改可能使这些构建过程失效。应改为设置 `XML2_CONFIG`，使其指向非默认的 xml2-config。这种方法对较旧和较新的 PostgreSQL 发行版都适用。
- 在 MSVC 构建中，处理 Python 路径名中的空格（Victor Wagner）
- 在 MSVC 构建中，修复 Visual Studio 版本检测，使其适用于更多语言设置（Andrew Dunstan）
- 在 MSVC 构建中，对高于 3.0 的 bison 版本使用 `-Wno-deprecated`，与非 Windows 构建现有行为一致（Andrew Dunstan）
- 将时区数据文件更新到 tzdata 2020a 版，以反映摩洛哥和加拿大育空地区的夏令时法规变更，并修正上海的历史数据。

  America/Godthab 时区已更名为 America/Nuuk，以反映当前英语用法；不过，旧名称仍作为兼容性链接保留。

  另外，更新 initdb 的已知 Windows 时区名称列表，纳入最近新增的名称，提高在该平台上正确转换系统时区设置的可能性。

E.16. 发布版本 11.7

发布日期：. 2020-02-13

本次发布包含来自 11.6 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.16.1. 迁移到版本 11.7

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.6 的版本升级，请参见第 E.17 节。

E.16.2. 变更

- 为 ALTER ... DEPENDS ON EXTENSION 补上缺失的权限检查（Álvaro Herrera）

  将对象标记为依赖某个扩展时，没有进行任何权限检查。这一疏漏使任何用户都可以将例程、触发器、物化视图或索引标记为可由任何有权删除该扩展的用户删除。现在要求调用用户拥有指定对象（因而有权删除它）。（CVE-2020-1720）

- 确保在适当情况下，将分区表上的行触发器正确克隆到子分区（Álvaro Herrera）

  创建或附加分区时，可能遗漏用户定义的触发器（但不会遗漏外键或延迟唯一约束的触发器）。

- 修复逻辑复制订阅者代码，使其在适当情况下执行逐列 UPDATE 触发器（Peter Eisentraut）

- 避免大事务必须溢写到许多独立临时文件时逻辑解码发生失败（Amit Khandekar）

- 修复逻辑复制订阅者处理行更新时可能发生的崩溃或数据损坏（Tom Lane，Tomas Vondra）

  只有当订阅者的表包含未从发布者复制、且数据类型按引用传递的列时，此缺陷才会导致可见问题。

- 修复对已订阅关系进行 DDL 变更后逻辑复制订阅者的崩溃（Jehan-Guillaume de Rorthais，Vignesh C）

- 修复数据库崩溃并重启后逻辑复制发布者的失败（Vignesh C）

- 确保 pg_replication_slot_advance() 对物理复制槽的影响在重启后仍然保留（Alexey Kondratov，Michael Paquier）

- 提高使用 REPLICA IDENTITY FULL 的逻辑复制效率（Konstantin Knizhnik）

  更新或删除操作中查找现有元组时，返回第一个匹配元组，而非最后一个。

- 确保并行计划始终在正确的时机关闭（Kyotaro Horiguchi）

  已知这一疏漏会使多批次并行哈希连接产生 “temporary file leak” 警告。

- 防止过早关闭位于 Limit 节点下方的 Gather 或 GatherMerge 计划节点（Amit Kapila）

  这可以避免此类计划节点需要被扫描多次时发生失败，例如它位于嵌套循环内侧时。

- 提高多核 CPU 上并行哈希连接的效率（Gang Deng，Thomas Munro）

- 避免没有空闲动态共享内存槽时并行 CREATE INDEX 发生崩溃（Thomas Munro）

  改为回退到非并行索引构建。

- 避免没有空闲动态共享内存槽时发生内存泄漏（Thomas Munro）

- 在临时表上创建、删除或重建索引时，忽略 CONCURRENTLY 选项（Michael Paquier，Heikki Linnakangas，Andres Freund）

  这可以避免临时表具有 ON COMMIT 动作时发生奇怪的失败。对临时表使用 CONCURRENTLY 本来就没有益处，因为其他会话无法访问该表，额外处理没有意义。

- 修复重置标记为 `ON COMMIT DELETE ROWS` 的临时表上的表达式索引时可能发生的失败（Tom Lane）
- 修复对 `box`、`range` 和 `inet` 数据类型执行 BRIN 索引操作时可能发生的崩溃（Heikki Linnakangas）
- 修复 GIN 索引中对已删除页面的处理（Alexander Korotkov）

  避免可能发生的死锁、对已删除页面状态的不正确更新，以及无法遍历最近删除页面的问题。
- 修复多行 `VALUES` 列表中包含 SubPlan（子 `SELECT`）时可能发生的崩溃（Tom Lane）
- 修复元组转换期间未能为“缺失”属性插入默认值的问题（Vik Fearing，Andrew Gierth）

  当值来自通过 `ALTER TABLE ADD COLUMN` 添加、且具有常量默认值的列时，这可能导致它们被错误地读取为 NULL。
- 修复 FileClose() 失败后发生的崩溃（Noah Misch）

  只有启用 `data_sync_retry` 时才能观察到此问题，否则 FileClose() 失败会被报告为 PANIC。
- 修复使用按引用传递的聚合转换状态时发生的罕见崩溃（Andres Freund，Teodor Sigaev）
- 改进 `to_date()` 和 `to_timestamp()` 中的错误报告（Tom Lane，Álvaro Herrera）

  报告输入字符串中不正确的月份名或星期名时，可能在多字节字符中间截断输入，导致错误消息的编码不正确，并可能引发后续失败。现在改为在下一个空白处截断。
- 修复对公元前日期计算 `EXTRACT(ISOYEAR FROM` *`timestamp`*`)` 时结果相差一的问题（Tom Lane）
- 避免系统目录中存在自引用视图时 `information_schema` 视图发生栈溢出（Tom Lane）

  自引用视图无法正常工作，它总会导致无限递归。尝试执行视图时，我们已能正确处理这种情况，但在查询它是否可自动更新时却未能正确处理。
- 确保 WAL 发送进程进程在 `pg_stat_activity` 中的事务开始时间始终显示为 NULL（Álvaro Herrera）

  以前，`xact_start` 列有时会显示进程启动时间。
- 提高内侧关系非常大时的哈希连接性能（Thomas Munro）
- 修复 `EXPLAIN` 输出中“Subplans Removed”字段的位置（Daniel Gustafsson，Tom Lane）

  在非文本输出格式中，此字段被输出到“Plans”子组内部，导致输出语法无效。现在按预期将其附加到父级 Append 或 MergeAppend 计划节点。这也会改变该字段在文本输出格式中的位置：如果同一计划节点附加了 InitPlan，“Subplans Removed”现在会出现在它们之前。
- 如果用户能够读取查询中实际指定的表的相应列，则允许规划器对子表统计信息应用可能泄漏信息的测试（Dilip Kumar，Amit Langote）

  此项修改修复了 CVE-2017-7484 的修复在分区表上引入的性能问题。那次安全修复禁止将可能泄露信息的操作符应用于当前用户无权直接读取的列的统计信息。然而，常见的做法是只在

父分区表上授予权限，而不逐一在各个分区上授予权限。在这种情况下，用户可以通过父表读取该列，因此这一安全限制没有意义，只会让规划器的估计不必要地变差。

- 修复为范围操作符 <@ 和 @> 计算选择率时在边界情况下发生的崩溃和估算错误（Michael Paquier，Andrey Borodin，Tom Lane）
- 应用最常见值扩展统计信息时忽略系统列（Tomas Vondra）

  这可避免受影响查询发生“negative bitmapset member not allowed”规划器错误。
- 修复 BRIN 索引逻辑，使其支持假想 BRIN 索引（Julien Rouhaud，Heikki Linnakangas）

  以前，如果“索引建议器”扩展尝试让规划器生成包含假想 BRIN 索引的计划，就会失败，因为 BRIN 代价估算代码始终尝试实际访问索引的元页面。现在会检查索引是否只是假想索引；如果是，则使用索引参数的默认假设。
- 改进对带条件 INSTEAD 规则的视图尝试使用自动更新时的错误报告（Dean Rasheed）

  这种用法从未得到支持，但以前只在执行时才抛出错误，因此它可能被规划器错误掩盖。
- 防止复合类型通过范围类型间接包含自身（Tom Lane，Julien Rouhaud）
- 禁止返回 record 等伪类型的分区键表达式（Tom Lane）
- 修复索引表达式使用禁止的类型时的错误报告（Amit Langote）
- 修复仅包含 VALUES 列表的视图的转储，使其能够处理视图输出列被重命名的情况（Tom Lane）
- 确保计算视图或规则的依赖关系时，将 XMLTABLE 结构中使用的数据类型和排序规则考虑在内（Tom Lane）

  以前，如果某个类型未在视图中的其他地方被引用，删除该类型就可能破坏使用 XMLTABLE 的视图。此修复不会纠正现有视图已记录的依赖关系，仅对新建视图生效。
- 防止 RADIUS 认证参数被意外转换为小写或截断（Marcos David）

  pg_hba.conf 解析器错误地将这些字段当作 SQL 标识符，而它们通常并不是。
- 在发送 ReadyForQuery 之前向客户端传送收到的 NOTIFY 消息，而非在之后发送（Tom Lane）

  此更改确保，对于 libpq 以及行为类似的其他客户端库，当客户端认为事务已完成时，事务期间收到的所有通知都已可用。这在实际应用中可能没有影响（无论如何，应用都需要处理异步通知），但更便于编写行为可重现的测试用例。
- 即使未编译 GSSAPI 代码，也允许 libpq 解析所有与 GSS 有关的连接参数（Tom Lane）

  这使其行为与我们的 SSL 支持类似：我们早已认为始终接受所有相关参数是可取的，即使某个构建缺少相应特性，导致部分参数被忽略或受到限制。
- 修复 ecpg 的 PGTYPEStimestamp_fmt_asc() 函数对格式代码 %b 和 %B 的不正确处理（Tomas Vondra）

  由于索引偏差一的错误，这些代码会输出错误的月份名，或可能导致崩溃。
- 修复并行 pg_dump/pg_restore，使其更妥善地处理无法创建工作进程的情况（Tom Lane）

- 防止尝试通过信号终止并行 pg_dump/pg_restore 运行时可能发生的崩溃或卡死（Tom Lane）
- 在 pg_upgrade 搜索表中不可升级的数据类型时，检查数组和范围的内部类型（Tom Lane）
- 对 createuser 的 `--connection-limit` 选项进行更全面的语法检查（Álvaro Herrera）
- 在更多情况下处理 PL/pgSQL 复合类型变量所引用的具体类型发生变更的情况（Ashutosh Sharma，Tom Lane）

  删除并重新创建 PL/pgSQL 变量所引用的复合类型，可能导致 “could not open relation with OID *NNNN*” 错误。
- 避免 `postgres_fdw` 尝试向远程服务器发送 `UPDATE remote_tab SET (x,y) = ( SELECT ...)` 之类命令时发生崩溃（Tom Lane）
- 在 `contrib/dict_int` 中，拒绝小于一的 `maxlen` 设置（Tomas Vondra）

  这可防止该参数设置不合理时可能发生的崩溃。
- 在 `contrib/tablefunc` 的 `crosstab()` 函数中禁止 NULL 类别值（Joe Conway）

  这种情况从未产生有用的结果，而且在某些平台上会导致崩溃。
- 修复 configure 对 OpenSSL 的 `SSL_clear_options()` 函数的探测，使其适用于 1.1.0 之前的 OpenSSL 版本（Michael Paquier，Daniel Gustafsson）

  使用旧版 OpenSSL 构建 PostgreSQL 时，此问题可能导致无法按预期设置 SSL 压缩选项。
- 将某些超时和统计跟踪 GUC 变量标记为 `PGDLLIMPORT`，以允许扩展在 Windows 上访问它们（Pascal Legrand）

  这适用于 `idle_in_transaction_session_timeout`、`lock_timeout`、`statement_timeout`、`track_activities`、`track_counts` 和 `track_functions`。
- 避免 “slab” 内存上下文健全性检查中的内存泄漏（Tomas Vondra）

  这不会影响生产构建，因为它们通常不会启用内存上下文检查；但在调试构建中，这种泄漏可能相当严重。
- 修复 LWLock 统计机制报告重复统计条目的问题（Fujii Masao）

  由于 hash 表键创建有误，LWLock 统计代码（默认不编译，需要使用 `-DLWLOCK_STATS` 编译）可能为同一个 LWLock 和后端进程报告多个条目。
- 修复导致 Windows 上进程间信号延迟传递的竞争条件（Amit Kapila）

  这使 `NOTIFY` 出现了可见的时序异常，也可能导致其他错误行为。
- 在 Windows 上，发生 `ERROR_ACCESS_DENIED` 文件访问失败后重试几次（Alexander Lakhin，Tom Lane）

  这有助于处理目标文件已被标记为删除、但尚未实际消失，导致打开文件失败的情况。例如，pg_ctl 在探测 postmaster 是否已经关闭时，经常因此类错误而失败。

E.17. 发布版本 11.6

发布日期：. 2019-11-14

本次发布包含来自 11.5 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.17.1. 迁移到版本 11.6

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果使用了带有 GiST 索引的 `contrib/intarray` 扩展，且依赖 `<@` 操作符的索引搜索，请参阅下方相关条目。

此外，如果是从早于 11.1 的版本升级，请参见第 E.22 节。

E.17.2. 变更

- 修复 `ALTER TABLE SET` 使用自定义关系选项时的失败（Michael Paquier）
- 如果没有更改所有父表，则禁止更改多重继承列的类型（Tom Lane）

  以前允许这样做，导致对类型已不同步的父表进行查询时失败。
- 避免在事务块中的 `ANALYZE` 命令中两次指定同一个目标表时发生失败（Tom Lane）
- 防止 `VACUUM` 尝试冻结涉及仍在运行事务的旧 multixact ID（Nathan Bossart，Jeremy Schneider）

  这种情况会导致 `VACUUM` 一直失败，直到旧事务终止。
- `SET CONSTRAINTS ... DEFERRED` 在分区表上执行失败，错误地报告缺少触发器（Álvaro Herrera）
- 修复父分区表包含已删除列时为分区创建索引发生的失败（Michael Paquier）
- 修复规划器对使用 ICU 排序规则的 `ILIKE` 中可进行大小写折叠的字符的测试（Tom Lane）

  此错误导致规划器将模式中太多的部分视为固定前缀，使得从 `ILIKE` 子句生成的索引扫描可能遗漏应当找到的条目。
- 确保处理查询表达式时也处理 `WINDOW` 子句中的偏移量表达式（Andrew Gierth）

  当偏移量是较复杂的表达式时，这一疏漏可能导致各种失败。例如，如果函数被内联，此类表达式中的函数参数引用就会失败。
- 修复 `WITH CHECK OPTION` 表达式和行级安全策略表达式中整行变量的处理（Andres Freund）

  以前，这种用法可能导致虚假的行类型不匹配错误。
- 避免 postmaster 子进程数组已无空闲槽位时，并行查询请求后台工作进程导致 postmaster 失败（Tom Lane）
- 如果 `BEFORE UPDATE` 触发器原样返回旧元组，且它并非最后一个此类触发器，则防止可能发生的重复释放（Thomas Munro）

- 修复 *x* = ANY (*array*) 或相关操作包含值为 NULL 的常量数组时发生的崩溃（Tom Lane）
- 修复查询尝试访问 TOAST 表时发生的 “unexpected relkind” 错误（John Hsu，Michael Paquier，Tom Lane）

  错误应说明权限被拒绝，但代码重构破坏了这种情况的处理。
- 在并行工作进程启动期间设置 GUC 参数发生错误时，提供相关的错误上下文行（Thomas Munro）
- 在可串行化模式下，确保在正确的行版本上获取行级谓词锁（Thomas Munro，Heikki Linnakangas）

  如果行的可见版本经过 HOT 更新，锁可能被加在其现已死亡的前一版本上，导致在不易察觉的情况下无法保证可串行化。
- 确保仅对以读写模式打开的文件应用 fsync()（Andres Freund，Michael Paquier）

  某些代码路径尝试在以只读方式打开文件后执行此操作，但在某些平台上，这会导致 “bad file descriptor” 或类似错误。
- 允许编码转换成功处理比以前更长的字符串（Álvaro Herrera，Tom Lane）

  以前，输入字符串有 0.25GB 的硬性上限；现在，只要转换后的输出不超过 1GB，就能够成功转换。
- 避免堆页面剪枝期间不必要的目录查找（Thomas Munro）

  此处已不再需要检查不记录 WAL 的索引，而该检查在某些工作负载下导致了显著的性能问题。至少在理论上，它也存在发生死锁的可能性。
- 避免为窗口函数创建不必要地庞大的元组存储（Andrew Gierth）

  在某些情况下，元组存储会包含源表的所有列，而非仅包含查询所需的列。
- 修复未能对分组 hash 表的相等比较进行 JIT 编译、导致性能损失的问题（Andres Freund）
- 允许 repalloc() 在大型内存块缩小时归还空间（Tom Lane）
- 确保归档恢复结束时删除临时 WAL 文件和历史文件（Sawada Masahiko）
- 避免启用 recovery_min_apply_delay 时归档恢复发生失败（Fujii Masao）

  recovery_min_apply_delay 通常不用于这种配置，但它应该能够正常工作。
- 修复发布者和订阅者对表的复制标识列认定不一致时逻辑复制发生的失败（Jehan-Guillaume de Rorthais，Peter Eisentraut）

  在订阅者上将发布者根本不存在的列声明为复制标识的一部分，会导致 “negative bitmapset member not allowed” 错误。
- 避免逻辑复制 WAL 发送进程关闭期间出现不必要的延迟（Craig Ringer，Álvaro Herrera）
- 修复逻辑复制 WAL 接收进程进程中的超时处理（Julien Rouhaud）

  错误的逻辑导致 wal_receiver_timeout 在逻辑复制部署中无法生效。

- 为逻辑解码的复制消息正确添加时间戳（Jeff Janes）

  例如，这一疏漏导致 pg_stat_subscription.last_msg_send_time 通常显示为 NULL。

- 在逻辑解码中，确保重建快照时正确计入子事务（Masahiko Sawada）

  此错误会导致断言失败；尚不清楚它在生产构建中是否有不良影响。

- 修复后端进程此前曾等待同步复制时，在退出期间发生的竞争条件（Dongming Liu）

- 修复 ALTER SYSTEM，使其能够处理 postgresql.auto.conf 中的重复条目（Ian Barwick）

  ALTER SYSTEM 本身不会产生这种状态，但修改 postgresql.auto.conf 的外部工具可能会。现在会删除目标变量的重复条目，然后将新设置（如有）追加到末尾。

- 拒绝配置文件中包含空文件名的 include 指令，并更清楚地报告包含文件递归的问题（Ian Barwick，Tom Lane）

- 避免在使用 PAM 认证时记录有关已放弃连接的报错消息（Tom Lane）

  基于 libpq 的客户端在需要密码时通常会尝试连接两次，因为它们直到第一次连接尝试失败后才会提示用户输入密码。因此，服务器在客户端收到密码请求后关闭连接时，不会生成无用的日志消息。然而，PAM 认证代码没有遵循这一约定，会为实际上并未发生的认证失败生成多条消息。

- 修复 time with time zone 输入中某些不完整日期指定未被检测到的情况（Alexander Lakhin）

  如果指定的时区具有随时间变化的 UTC 偏移量，就必须同时指定日期，以确定偏移量。对于某些语法，以前未强制进行此项检查，可能产生错误输出。

- 修复 bitshiftright() 的错误行为（Tom Lane）

  当位串长度不是 8 的倍数时，位串右移操作符未能将结果最后一个字节中的填充空间清零。虽然大多数操作看不到这些填充位，但其中任何非零位都会导致意外的比较行为，因为位串比较预期这些额外的位始终为零，并未特意忽略它们。

  如果由于将 bitshiftright() 的输出保存在表中而导致数据不一致，可以使用类似以下语句进行修复：

  ```
  UPDATE mytab SET bitcol = ~(~bitcol) WHERE bitcol != ~(~bitcol);
  ```

- 恢复 json[b]_populate_record() 和 json[b]_populate_recordset() 从 AS 子句获取类型信息的能力（Tom Lane）

  如果记录参数为 NULL 且没有声明复合类型，则尝试改用 AS 子句。虽然不推荐这种用法，但它以前可以工作，现在又恢复了。

- 避免在 XMLTABLE 中选择命名空间节点时发生崩溃（Chapman Flack）

- 修复对间隔乘法中边界情况下整数溢出的检测（Yuya Watari）

- 修复使用 ICU 排序规则时 lower()、upper() 和 initcap() 函数中的内存泄漏（Konstantin Knizhnik）

- 避免 `ispell` 文本检索词典包含错误的词缀数据时发生崩溃（Arthur Zakirov）
- 修复 GIN 倒排列表的不正确压缩逻辑（Heikki Linnakangas）

  如果相邻被索引 TID 之间的距离超过 16TB，GIN 倒排列表条目可能需要 7 字节。处理逻辑中的某一步未与此保持一致，可能尝试将该值写入 6 字节缓冲区。原则上这可能导致栈越界，但在大多数架构上，下一个字节很可能是未使用的对齐填充，因此此缺陷无害。无论如何，此缺陷都很难触发。
- 修复 KNN-GiST 中对无穷大、NaN 和 NULL 值的处理（Alexander Korotkov）

  如果为非空列值计算出的某些距离为无穷大或 NaN，查询输出顺序可能不正确（与普通排序的结果不同）。
- 修复 KNN-SP-GiST 中对搜索 NULL 的处理（Nikita Glukhov）
- 在 Windows 上，识别 “Norwegian (Bokmål)” 区域设置名称的更多拼写形式（Tom Lane）
- 避免 ECPG 客户端在定义了 `ENABLE_NLS` 的情况下包含 `ecpglib.h` 时编译失败（Tom Lane）

  这种风险由位置不当的声明造成：`ecpg_gettext()` 不应对客户端代码可见。
- 在 psql 中，意外丢失连接并成功重新连接后，重新同步关于服务器的内部状态（Peter Billen，Tom Lane）

  通常这并不必要，因为状态本来就是相同的。但在边界情况下可能有影响，例如连接可能指向多台服务器中的某一台。此更改使 psql 重新输出原本在启动时会输出的所有交互消息，例如是否使用 SSL。
- 避免 psql 中特定平台上的空指针解引用（Quentin Rameau）
- 在 pg_dump 中，确保名称相似的触发器和行级安全策略对象的输出顺序稳定（Benjie Gillam）

  以前，如果不同表上的两个触发器同名，它们会按 OID 排序，这不如按表名排序合适。RLS 策略也存在同样的问题。
- 修复 pg_dump，使其重新能够使用 8.3 之前版本的源服务器（Tom Lane）

  此前的一项修复使 pg_dump 始终尝试查询 `pg_opfamily`，但 8.3 之前的版本中不存在该目录。
- 在 pg_restore 中，将 `-f -` 解释为 “输出到 stdout”（Álvaro Herrera）

  这使 pg_restore 的行为与其他一些应用一致，特别是使 v12 之前分支的行为与版本 12 的 pg_restore 类似，简化了适用于多个 PostgreSQL 版本的转储/恢复脚本的编写。在此更改之前，pg_restore 将此类开关解释为 “输出到名为 - 的文件”，但很少有人需要这种行为。
- 改进 pg_upgrade 对使用了表示形式已变更的数据类型（如 `line`）的检查（Tomas Vondra）

  当所关注的数据类型是某个已存储域类型或复合类型列的底层类型时，以前的代码可能漏检。
- 检测 pg_basebackup 期间的文件读取错误（Jeevan Chalke）
- 在 pg_basebackup 中，直到备份结束才对输出文件执行 fsync（Michael Paquier）

以前的代码在 fsync 较慢时可能导致超时失败。

- 在源集群在线的情况下运行 pg_rewind 时，像 pg_dump 那样禁用超时（Alexander Kukushkin）
- 修复使用 `-s` 选项时，延续的 WAL 记录恰好在页面边界处结束导致 pg_waldump 失败的问题（Andrey Lepikhov）
- 在 pg_waldump 中，将 `newitemoff` 字段包含在 B-树页面分裂记录中（Peter Geoghegan）
- 在使用 `--bkp-details` 选项的 pg_waldump 中，避免为涉及整页写入的 WAL 记录输出多余换行（Andres Freund）
- 修复 pg_waldump 中少量的内存泄漏（Andres Freund）
- 修复设置了较高 `--jobs` 值的 vacuumdb，使其更妥善地处理文件描述符耗尽的情况（Michael Paquier）
- 修复 PL/pgSQL，使其更好地处理复合类型的替换（Tom Lane）

  涵盖在 PL/pgSQL 函数两次执行之间，复合类型被完全删除、随后又创建了同名新类型的情况。复合类型变量现在会更新以匹配新的定义。
- 修复 `contrib/amcheck`，使其在热备期间跳过不记录 WAL 的索引（Andrey Borodin，Peter Geoghegan）

  这种情况下，不记录 WAL 的索引未必包含有效数据，因此不要尝试检查它。
- 修复 `contrib/intarray` 的 GiST 操作符类，使其在空数组上使用 `<@` 时不会失败（Tom Lane）

  `array_column <@ constant_array` 这样的子句被认为可以使用索引，但索引搜索可能找不到空数组值；显然，这类条目应当直接匹配该搜索。

  唯一可行且能回补到旧版本的修复方法，是让 `<@` 索引搜索扫描整个索引，本补丁正是这样做的。遗憾的是，这意味着查询性能很可能比普通顺序扫描还差。

  性能受到此更改不利影响的应用有几种选择。可以切换到没有此缺陷的 GIN 索引，也可以将 *`array_column <@ constant_array`* 替换为 *`array_column <@ constant_array`* `AND` *`array_column && constant_array`*。后者的性能与以前大致相同，并会找出给定常量数组的所有非空子集；这也正是以前能够可靠期望该查询做到的全部事情。
- 修复 configure 对 libperl 是否存在的测试，使其适用于近期的 Red Hat 发行版（Tom Lane）

  以前，如果用户将 `CFLAGS` 设置为 `-O0`，此测试可能失败。
- 确保为 PowerPC 上的自旋锁生成正确的代码（Noah Misch）

  以前的自旋锁代码允许编译器为不接受零号寄存器的汇编指令选择该寄存器，导致构建失败。我们仅见过一份很久以前的报告与此缺陷吻合，但它可能给尝试构建修改过的 PostgreSQL 代码或使用非常规编译器选项的用户带来问题。
- 在 PowerPC 上，避免依赖 xlc 编译器的 `__fetch_and_add()` 函数（Noah Misch）

xlc 13 及更新版本对该函数的解释与我们的用法不兼容，导致构建出的 PostgreSQL 无法使用。现在改用自定义汇编代码来修复此问题。

- 在 AIX 上不再使用编译器选项 `-qsrcmsg`（Noah Misch）

  这避免了 xlc v16.1.0 的内部编译器错误，除了改变编译器错误消息的格式之外，几乎没有其他影响。

- 修复 MSVC 构建过程，使其能够处理 OpenSSL 文件路径中的空格（Andrew Dunstan）

- 将时区数据文件更新到 tzdata 2019c 版，以反映斐济和诺福克岛的夏令时法规变更，并修正艾伯塔、奥地利、比利时、不列颠哥伦比亚、柬埔寨、香港、印第安纳（佩里县）、加里宁格勒、肯塔基、密歇根、诺福克岛、韩国和土耳其的历史数据。

E.18. 发布版本 11.5

发布日期：. 2019-08-08

本次发布包含来自 11.4 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.18.1. 迁移到版本 11.5

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.1 的版本升级，请参见第 E.22 节。

E.18.2. 变更

- 使用函数式类型转换语法转换为临时类型时，要求使用模式限定（Noah Misch）

  我们早已要求调用临时函数时显式指定临时模式，即 `pg_temp.`*`func_name`*`(`*`args`*`)`。现在也要求在使用函数记法转换为临时类型时这样做，例如 `pg_temp.`*`type_name`*`(`*`arg`*`)`。否则，就可能利用临时对象截获函数调用，从而以与我们在 CVE-2007-2138 中阻止的方式基本相同的方式提升权限。（CVE-2019-10208）

- 修复需要跨类型比较的 hash 子计划的执行（Tom Lane，Andreas Seltenreich）

  hash 子计划使用外层查询原有的比较操作符来比较 hash 表中的项。如果该操作符是跨类型的，这样做就是错误的，因为 hash 表中的所有项都会具有子查询的输出类型。对于核心 PostgreSQL 中可进行 hash 运算的跨类型操作符，这一错误在 64 位机器上似乎几乎没有危害，但在 32 位机器上可能导致崩溃，也可能造成未经授权的服务器内存泄露。扩展可能提供风险更大的、可进行 hash 运算的跨类型操作符。（CVE-2019-10209）

- 修复在一条命令中更改多列类型时 `ALTER TABLE ... ALTER COLUMN TYPE` 发生的失败（Tom Lane）

  这修复了最近的小版本引入的回归问题：使用被修改列的索引未得到正确处理，导致 `ALTER TABLE` 期间出现异常失败。

- 如果子分区中存在待处理的触发器事件，则防止删除分区表的触发器（Álvaro Herrera）

  这尤其适用于外键约束，因为外键约束是通过触发器实现的。

- 将触发器定义从分区表复制到其分区时，包含用户指定的触发器参数（Patrick McHardy）

- 建立依赖关系以防止删除分区键列（Tom Lane）

  `ALTER TABLE ... DROP COLUMN` 会拒绝删除分区键列。但间接删除（例如删除键列的数据类型所引发的级联删除）没有这样的检查，因此可能删除键列。这会使分区表严重损坏，既无法访问也无法删除。

  此修复添加了 `pg_depend` 条目，确保如果级联删除迫使键列被移除，将删除整个分区表，而非仅删除键列。不过，这些条目仅在创建分区表时生成，因此此修复无法消除现有分区表的风险。此问题只会发生在分区键列使用非内置数据类型的情况下，所以对大多数用户似乎没有威胁。

- 确保分区表与其默认分区之间的列号得到正确映射（Amit Langote）

  如果映射不是严格的一一对应，例如一个表中有已删除列而另一个表中没有，某些操作会出现错误行为。

- 在分区表上创建索引时，忽略作为外部表的分区（Álvaro Herrera）

  以前，遇到外部表分区时会抛出错误，但这没有帮助，也不能防止任何实际问题。

- 在更多情况下剪枝分区表的默认分区（即避免无用地扫描它）（Yuzuko Hosoya）

- 修复存在多个 `boolean` 类型分区键列时可能无法剪枝分区的问题（David Rowley）

- 当涉及的表是继承父表时，不要通过优化移除 `GROUP BY` 列（David Rowley）

  通常，如果表的主键列包含在 `GROUP BY` 中，就可以安全地移除其他分组列，因为主键列足以使各组唯一。但如果查询还读取继承子表，这条规则就不成立：父表的唯一性并不延伸到子表。

- 避免对半连接查询错误地使用并行哈希连接（Thomas Munro）

  此错误导致某些 `EXISTS` 查询返回重复的结果行。

- 避免在某些带有 `GROUPING SETS` 的查询中使用不必要的排序步骤（Andrew Gierth，Richard Guo）

- 修复规划器的索引端点探测可能发生的失败（Tom Lane）

  使用最近创建的索引确定列的最小值或最大值时，规划器可能选中一个最近死亡、实际并不包含端点值的元组。最坏情况下，该元组可能包含空值，导致可见的 “found unexpected null value in index” 错误；更常见的情况是最终使用错误的值，降低规划估算的质量。

- 修复 `EvalPlanQual` 复检期间无法访问触发器过渡表的问题（Alex Aktsipetrov）

  在存在并发更新时，依赖过渡表的触发器有时会失败。

- 修复重建外键约束时对多列外键的不正确处理（Tom Lane）

  如果外键的各列类型不完全相同，`ALTER TABLE` 可能错误判断是否需要重新验证外键。这种错误似乎总会偏向保守方向，即进行不必要的重新验证。

- 不再为继承树构建扩展统计信息（Tomas Vondra）

  这避免了 `ANALYZE` 期间发生 “tuple already updated by self” 错误。

- 避免升级元组锁时出现虚假的死锁错误（Oleksii Kliukin）

当两个或更多事务正在等待事务 T1 释放元组级锁，而 T1 又将锁升级到更高的级别时，T1 结束时可能会报告这些等待事务之间存在虚假的死锁。

- 修复无法解除涉及多个并行工作进程的死锁的问题（Rui Hai Jiang）

  尚不清楚非人为构造的查询能否触发此缺陷，但如果发生，参与本可解除的死锁的查询会一直阻塞，直到被取消。

- 防止对端点为 `infinity` 的日期范围进行不正确的规范化（Laurenz Albe）

  如果端点为无穷大，通过递增或递减端点值将开区间转换为闭区间（或反过来）是不正确的，因此在这种情况下保持范围原样。

- 修复将非常大的 `money` 值转换为 `numeric` 时小数位丢失的问题（Tom Lane）

- 修复 `BTREE_META_CLEANUP` WAL 记录的输出（Michael Paquier）

- 防止对版本 2 的 B-树元页面处理不当导致断言失败（Peter Geoghegan）

- 修复 MIPS CPU 的自旋锁汇编代码，使其适用于 MIPS r6（YunQiang Su）

- 确保从 PL/pgSQL 函数返回的记录值或行值被标记为函数声明的复合类型（Tom Lane）

  这可避免将结果直接存入表时出现问题。

- 使 libpq 忽略连接服务文件中的回车符（`\r`）（Tom Lane，Michael Paquier）

  在某些边界情况下，包含 Windows 风格换行符的服务文件可能被错误解析，导致连接失败。

- 在 psql 中，避免在 `SET` *`variable`* `=` 之后提供不正确的 Tab 补全选项（Tom Lane）

- 修复 psql 的 `\d` 命令中少量的内存泄漏（Tom Lane）

- 修复 pg_dump，以确保按正确顺序转储自定义操作符类（Tom Lane）

  如果用户定义的操作符类是用户定义范围类型的子类型操作符类，相关对象会按错误顺序转储，产生无法恢复的转储。（底层对操作符类依赖关系的处理缺陷也可能在其他情况下出现，但这是唯一已知的情况。）

- 修复使用 `-R` 选项时 pgbench 可能卡死的问题（Fabien Coelho）

- 提高 `contrib/amcheck` 索引验证的可靠性（Peter Geoghegan）

- 修复 `contrib/jsonb_plperl` 中对 Perl `undef` 值的处理（Ivan Panchenko）

- 修复 `contrib/passwordcheck`，使其能与 `check_password_hook` 的其他使用者共存（Michael Paquier）

- 修复 `contrib/sepgsql` 测试，使其适用于近期的 SELinux 版本（Mike Palmiotto）

- 提高 `src/test/kerberos` 和 `src/test/ldap` 回归测试的稳定性（Thomas Munro，Tom Lane）

- 提高 `src/test/recovery` 回归测试的稳定性（Michael Paquier）

- 减少 pg_upgrade 测试脚本的 stderr 输出（Tom Lane）

- 修复 pgbench 回归测试，使其适用于 Windows（Fabien Coelho）
- 修复构建目录位于非根 msys 挂载点时 TAP 测试在 msys Perl 下的运行（Noah Misch）
- 支持使用 Microsoft Visual Studio 2019 构建 Postgres（Haribabu Kommi）
- 在 Visual Studio 构建中，如果设置了 `WindowsSDKVersion` 环境变量，则遵循其设置（Peifeng Qiu）

  这修复了某些配置下的构建失败。
- 在 Visual Studio 构建中支持 OpenSSL 1.1.0 及更新版本（Juan José Santamaría Flecha，Michael Paquier）
- 在顶层调用非 GNU make 时，允许将 make 选项向下传递给 gmake（Thomas Munro）
- 在 initdb 期间，避免选择 `localtime` 或 `posixrules` 作为 `TimeZone`（Tom Lane）

  在某些情况下，initdb 会选择这两个人为设置的时区名之一，而非“真实”的时区名。现在，只要有其他名称与 C 库的时区行为匹配，就优先选择其他名称。
- 调整 `pg_timezone_names` 视图，使其仅在 `Factory` 时区具有简短缩写时才显示它（Tom Lane）

  历史上，IANA 为这个人为设置的时区指定了类似 `Local time zone must be set--see zic manual page` 的“缩写”。现代版本的 tzdb 数据库改为显示 `-00`，但某些平台会修改数据，显示某个历史短语。现在，仅当此时区使用现代缩写时才显示它。
- 将我们使用的时区库副本同步到 IANA tzcode 2019b 版（Tom Lane）

  这添加了对 zic 的新选项 `-b slim` 的支持，可减小已安装时区文件的大小。我们目前未使用此选项，但未来可能会启用。
- 将时区数据文件更新到 tzdata 2019b 版，以反映巴西的夏令时法规变更，并修正香港、意大利和巴勒斯坦的历史数据。

E.19. 发布版本 11.4

发布日期：. 2019-06-20

本次发布包含来自 11.3 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.19.1. 迁移到版本 11.4

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.1 的版本升级，请参见第 E.22 节。

E.19.2. 变更

- 修复 SCRAM 验证信息解析中的缓冲区溢出隐患（Jonathan Katz，Heikki Linnakangas，Michael Paquier）

  任何已认证用户都可以通过将自己的密码修改为恶意构造的值，引发基于栈的缓冲区溢出。除了使 PostgreSQL 服务器崩溃之外，这还可能足以让攻击者以 PostgreSQL 操作系统账户的身份执行任意代码。

libpq 中也存在类似的溢出隐患，可能允许恶意服务器使客户端崩溃，甚至以客户端的操作系统账户执行任意代码。

PostgreSQL 项目感谢 Alexander Lakhin 报告此问题。（CVE-2019-10164）

- 修复运行时分区剪枝逻辑中的各类错误（Tom Lane，Amit Langote，David Rowley）

  如果用于剪枝的比较值是动态确定的，或者剪枝决策涉及多个范围分区列，或者涉及稳定但非不可变的比较操作符，这些错误可能使分区表查询返回错误结果。

- 修复尝试将触发器定义复制到新分区时可能发生的崩溃（Tom Lane）

- 修复表具有部分排除约束时 `ALTER TABLE ... ALTER COLUMN TYPE` 发生的失败（Tom Lane）

- 修复为域约束添加注释时 `COMMENT` 命令发生的失败（Daniel Gustafsson，Michael Paquier）

- 防止 hash 聚合的 hash 键列表中存在重复列时可能发生的内存破坏（Andrew Gierth）

- 修复对零参数或多参数聚合执行部分聚合时，不正确的参数空值检查（David Rowley，Kyotaro Horiguchi，Andres Freund）

- 修复归并追加计划的错误生成（Tom Lane）

  此错误可能导致 “could not find pathkey item to sort” 错误。

- 修复带有重复连接名称的查询的不正确输出（Philip Dubé）

  这一疏漏导致包含此类查询的视图转储/恢复失败。

- 修复 `json_to_record()` 和 `json_populate_record()` 中将 JSON 字符串字面量转换为 JSON 类型输出列的处理（Tom Lane）

  这些情况下应将字面量作为独立的 JSON 值输出，但如果字面量包含任何需要转义的字符，代码就会出现错误行为。

- 修复对正则表达式中 `{1,1}` 量词的不正确优化（Tom Lane）

  此类量词被视为无操作并被优化掉；但文档规定，它们会为所附着的子表达式施加贪婪性，或在使用非贪婪变体 `{1,1}?` 时施加非贪婪性，而实际并未如此。只有当子表达式包含捕获括号或反向引用时，才会发生这种错误行为。

- 在页面分裂期间处理 INCLUDE 列时发生失败的罕见情况下，避免写入无效的空 B-树索引页（Peter Geoghegan）

  无效页面不会影响正常的索引操作，但可能导致后续 VACUUM 失败。如果某个索引已经发生此问题，可以通过重建该索引恢复。

- 避免初始化新进程的 `pg_stat_activity` 数据时可能发生的失败（Tom Lane）

  某些可能失败的操作，例如将从 SSL 证书提取的字符串转换为数据库编码，原本在临界区内执行。此处的失败会违反共享 `pg_stat_activity` 数据的访问协议，导致整个数据库卡死。

- 修复检查预先存在的共享内存段是否仍被冲突的 postmaster 使用时的竞争条件（Tom Lane）

- 修复 WAL 接收进程信号处理函数中的不安全代码（Tom Lane）

  这避免了 WAL 接收进程进程在收到关闭指令时崩溃或死锁的罕见问题。

- 避免在未连接到特定数据库的进程中，为检查参数而尝试访问数据库（Vignesh C，Andres Freund）

  此错误可能导致 “cannot read pg_class without having selected a database” 等失败。

- 避免使用 SSL 且 OpenSSL 的待处理数据缓冲区恰好包含 256 字节整数倍的数据时，libpq 可能挂起的问题（David Binderman）

- 改进 initdb 对系统时区多个等效名称的处理（Tom Lane，Andrew Gierth）

  如果 `/etc/localtime` 符号链接存在，则让 initdb 检查它，以便在系统时区的多个等效名称之间作出选择。这使 initdb 在存在多个相同时区时，更有可能选择用户预期的时区名。如果 `/etc/localtime` 不是指向时区数据文件的符号链接，或者时区是由 `TZ` 环境变量确定的，则行为不变。

  另外，如果 `TZ` 和 `/etc/localtime` 都未提供线索，则优先选择 `UTC`，而非该时区的其他拼写。这修复了 tzdata 2019a 将 `UCT` 和 `UTC` 时区名设为等效后引入的不便：initdb 之前会优先选择几乎没有人想要的 `UCT`。

- 修复 pg_dump 和 pg_dumpall 为数据库和表空间输出的 `GRANT` 命令的顺序（Nathan Bossart，Michael Paquier）

  如果执行过级联授权，恢复可能因 `GRANT` 命令的顺序未遵循相互依赖关系而失败。

- 让 pg_dump 先使用 `CREATE TABLE`，再使用 `ATTACH PARTITION` 重建表分区，而非在创建命令中包含 `PARTITION OF`（Álvaro Herrera，David Rowley）

  这避免了分区列顺序可能被更改以匹配父表而产生的问题。此外，即使未恢复父表，现在也可将分区从转储中恢复为独立表；`ATTACH` 会失败，但可以忽略该错误。

- 修复 reindexdb 中误导性的错误报告（Julien Rouhaud）

- 确保 vacuumdb 在使用并行作业期间发生错误时返回正确状态（Julien Rouhaud）

- 修复 `contrib/auto_explain`，使其不会在并行查询中引发问题（Tom Lane）

  以前，即使 `auto_explain` 没有记录父查询，并行工作进程也可能尝试记录自己的查询。这有时能够工作，但令人困惑，而且在某些情况下会导致 “could not find key N in shm TOC” 等失败。

  此外，修复了一个相差一的错误，该错误使得即使采样率设置为 1.0，也不一定记录每条查询。

- 在 `contrib/postgres_fdw` 中，考虑本地 `BEFORE ROW UPDATE` 触发器可能进行的数据修改（Shohei Mochizuki）

  如果触发器修改了 `UPDATE` 原本未更改的列，新值就不会传送到远程服务器。

- 在 Windows 上，避免数据库编码设置为 SQL_ASCII 时尝试记录非 ASCII 字符串而发生失败（Noah Misch）

代码原本假定此类字符串必须采用 UTF-8 编码，如果它们看起来不是有效编码就会抛出错误。现在直接将未经转换的字节传送到日志。

- 使 PL/pgSQL 的头文件可安全用于 C++（George Tarasov）

E.20. 发布版本 11.3

发布日期：. 2019-05-09

本次发布包含来自 11.2 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.20.1. 迁移到版本 11.3

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.1 的版本升级，请参见第 E.22 节。

E.20.2. 变更

- 防止通过选择率估计器绕过行级安全策略（Dean Rasheed）

  规划器的某些选择率估计器会将用户定义的操作符应用于 `pg_statistic` 中的值（例如最常见值）。因此，即使调用用户无权读取某列，可能泄露信息的操作符仍能披露该列中的一些项。在 CVE-2017-7484 中，我们增加了限制来防止这种情况，但没有考虑行级安全的影响。某个用户虽然具有读取某列的 SQL 权限，却因 RLS 策略而无法查看某些行，仍可能通过此类操作符获知这些行的部分内容。本补丁进一步收紧了规则，仅在没有相关 RLS 策略时，才允许将可能泄露信息的操作符应用于统计数据。（CVE-2019-10130）

- 避免报告分区路由错误时访问已释放的内存（Michael Paquier）

  这一错误可能导致崩溃，而且原则上可能被用于泄露服务器内存内容。（CVE-2019-10129）

- 避免对分区表执行 `ALTER TABLE` 时，发现分区索引可复用而导致目录损坏（Amit Langote，Tom Lane）

  例如，当 `ALTER COLUMN TYPE` 发现不需要物理重写表时，就会发生这种情况。

- 避免在单语句事务中创建带有 `ON COMMIT DROP` 和标识列的临时表时发生目录损坏（Peter Eisentraut）

  这一隐患被忽略了，因为这种用法实际上没有意义：临时表创建后立即就会被删除。

- 修复分区表包含的已删除列多于其分区时，`ALTER INDEX ... ATTACH PARTITION` 发生的失败（Álvaro Herrera）

- 修复某些情况下未能将分区的现有索引附加到新建分区索引的问题（Amit Langote，Álvaro Herrera）

  这会导致后续使用该分区索引的 DDL 出现 “index ... not found in partition” 等错误。

- 避免对分区查询结果关系执行 EPQ 复检时发生崩溃（Amit Langote）

  当使用 `READ COMMITTED` 隔离级别，且其他会话并发更新了某些目标行时，就会发生这种情况。

- 修复具有已删除属性的多级分区表中的元组路由（Amit Langote，Michael Paquier）
- 修复对分区表采用外键约束初始验证的慢速路径时发生的失败（Hadi Moshayedi，Tom Lane，Andres Freund）

  只有在无法使用快速路径的少见情况下（例如权限问题），此问题才会显现。
- 修复对继承树或分区表执行 `UPDATE` 或 `DELETE` 时，每个表都可被排除的情况下的行为（Amit Langote，Tom Lane）

  在这种情况下，如果存在 `RETURNING` 子句，查询未报告正确的输出列集合；如果存在应当触发的语句级触发器，也未触发它们。
- 直接访问分区且 `constraint_exclusion` 设置为 `on` 时，将分区自身的分区约束与所有 `CHECK` 约束一起用于排除检查（Amit Langote，Tom Lane）

  此更改恢复了 v10 中的行为。
- 避免尝试跨事务提交保存游标查询时发生错误而导致服务器崩溃（Tom Lane）

  如果过程在具有已打开的显式或隐式游标（例如 PL/pgSQL `FOR` 循环查询）时尝试提交，就必须先将游标执行完毕并保存其结果，然后才能提交事务。此类执行期间发生错误曾会导致崩溃。
- 避免存在 `FOR ALL TABLES` 发布时，更新临时表和不记录 WAL 的表抛出不正确的错误（Peter Eisentraut）

  对于发布而言，应当忽略此类表，但部分代码未能做到这一点。
- 目标关系为可更新视图时，修复包含多行 `VALUES` 的 `INSERT ... VALUES` 命令中对显式 `DEFAULT` 项的处理（Amit Langote，Dean Rasheed）

  当可更新视图的列没有默认值，但其底层表有默认值时，单行 `INSERT ... VALUES` 会使用底层表的默认值。然而，多行情况下以前始终使用 NULL。现在修正为与单行情况一致。
- 修复 `CREATE VIEW`，使其允许零列视图（Ashutosh Sharma）

  为与允许零列表的行为保持一致，我们也应该允许这种情况。由于表可以转换为视图，即使存在这一限制，也能创建零列视图，从而导致转储/重载失败。
- 补上对 `CREATE TABLE IF NOT EXISTS ... AS EXECUTE ...` 缺失的支持（Andreas Karlsson）

  `IF NOT EXISTS` 和 `EXECUTE` 的组合本应能够工作，但语法遗漏了它。
- 确保行级安全策略表达式中的子 `SELECT` 以正确用户的权限执行（Dean Rasheed）

  以前，如果通过视图访问带有 RLS 策略的表，此类检查可能以调用视图的用户身份执行，而非按预期以视图所有者的身份执行。
- 按照 SQL:2006 及后续标准的要求，当 `xmloption` 设置为 `content` 时，接受 XML 文档作为 `xml` 类型的有效值（Chapman Flack）

  以前，PostgreSQL 遵循不允许这种用法的 SQL:2003 定义。但这给转储/恢复带来了严重问题：没有任何 `xmloption` 设置能够接受所有有效的 XML 数据。因此，改为采用 2006 版的定义。

同时修改了 pg_dump，使其在恢复数据时输出 `SET xmloption = content`，确保即使当前设置为 `document`，转储/恢复也能正常工作。

- 改进服务器在启动时对预先存在的共享内存段是否仍在使用的检查（Noah Misch）

  即使 `postmaster.pid` 文件已被删除，postmaster 现在也更有可能检测到上一次 postmaster 运行留下的活跃进程。

- 避免 B-树索引清理逻辑中可能发生的除以零（Piotr Stefaniak，Alexander Korotkov）

  这可能导致对是否需要索引清理作出错误判断。

- 避免将并行工作进程的事务计为独立事务（Haribabu Kommi）

- 修复 GIN 索引 WAL 记录的不兼容问题（Alexander Korotkov）

  二月份小版本中应用的一项修复对向后兼容性考虑得不够周全。如果使用该小版本的备库读取由更早小版本的主库生成的 GIN 页面删除 WAL 记录，就会出现问题。

- 修复在复制连接中执行 `SHOW` 命令时可能发生的崩溃（Michael Paquier）

- 避免从 portal 中逐行获取数据时发生服务器内存泄漏（Tom Lane）

- 避免重建分区的关系缓存条目时发生内存泄漏（Amit Langote，Tom Lane）

- 在适当情况下，容忍 `fsync` 和 `sync_file_range` 调用返回 `EINVAL` 和 `ENOSYS` 错误（Thomas Munro，James Sewell）

  此前将文件同步失败改为触发 panic 的做法，在某些失败可预见且实质上表示 “操作不受支持” 的情况下显得过于谨慎。

- 在 BRIN 汇总操作期间，在自动清理的 `pg_stat_activity` 显示中报告正确的关系名（Álvaro Herrera）

- 避免启用 GEQO 时尝试规划逐分区连接而发生崩溃（Tom Lane）

- 修复从 `FULL` 外连接向外引用的横向引用导致的 “failed to build any *N*-way joins” 规划器失败（Tom Lane）

- 修复将返回集合的函数应用于可证明为空的关系时的查询规划错误（Tom Lane，Julien Rouhaud）

  在 v10 中，这一疏漏仅导致计划效率略低，但在 v11 中可能引发 “set-valued function called in context that cannot accept a set” 错误。

- 执行有关是否允许可能泄漏信息的操作符访问 `pg_statistic` 数据的规则时，检查正确用户的权限（Dean Rasheed）

  通过视图访问底层表时，在决定能否将可能泄漏信息的操作符应用于该表的统计数据时，应考虑视图所有者的权限，而非发出查询的用户的权限。这使规划器关于数据可见性的规则与执行器一致，避免不必要的低效计划。

- 修复规划器对分组查询的并行安全性评估（Etsuro Fujita）

  以前，可以并行执行的目标列表求值工作可能未得到并行执行。

- 修复规划器的唯一索引逻辑中对 “内含” 索引列的不正确处理（Tom Lane）

  这可能导致无法识别带有内含列的唯一索引能够证明查询结果唯一，从而产生低效计划。

- 修复对数组强制转换表达式的不正确严格性检查（Tom Lane）

  例如，这可能允许错误地内联严格 SQL 函数，导致未实施严格性条件。

- 在存在大量相等条件和大量可能相关的外键约束时，加快规划速度（David Rowley）

- 避免回滚创建了大量表的事务时发生 O(N^2) 性能问题（Tomas Vondra）

- 修复动态共享内存分配中边界情况下的服务器崩溃（Thomas Munro，Robert Haas）

- 修复动态共享内存管理中的竞争条件（Thomas Munro）

  这些问题可能导致 “dsa_area could not attach to segment” 或 “cannot unpin a segment that is not pinned” 错误。

- 修复热备 postmaster 收到 smart 关闭请求后可能无法关闭的竞争条件（Tom Lane）

- 修复 `pg_identify_object_as_address()` 收到无效输入时可能发生的崩溃（Álvaro Herrera）

- 修复 `txid_status()` 中可能发生的 “could not access status of transaction” 失败（Thomas Munro）

- 修复混用不同 OpenSSL 库版本时尝试使用 SCRAM 认证所导致的认证失败（Michael Paquier，Peter Eisentraut）

  如果 libpq 使用 OpenSSL 1.0.1 或更旧版本，而服务器使用 OpenSSL 1.0.2 或更新版本，则协商所用 SASL 机制的过程会出错，导致令人困惑的 “channel binding not supported by this build” 错误消息。

- 加强对已编码的 SCRAM-SHA-256 和 MD5 密码的验证（Jonathan Katz）

  只要密码字符串以相应字符开头，就可能被误认为已通过 hash 运算正确转换为 SCRAM-SHA-256 或 MD5 格式。这样的密码会被接受，但之后无法使用。

- 修复对隐含编码不同于数据库编码的 `lc_time` 设置的处理（Juan José Santamaría Flecha，Tom Lane）

  在此类区域设置下，包含非 ASCII 字符的本地化月份名或星期名以前会导致意外错误或错误输出。

- 以与服务器数据目录中其他文件相同的权限创建 `current_logfiles` 文件（Haribabu Kommi）

  以前，它使用 `log_file_mode` 指定的权限，但这可能给备份工具带来问题。

- 修复涉及一元负号操作符时不正确的 `operator_precedence_warning` 检查（Rikard Falkeborn）

- 禁止将 `NaN` 用作浮点服务器参数的值（Tom Lane）

- 重新安排 `REINDEX` 的处理，以避免重建 `pg_class` 的单个索引时发生断言失败（Andres Freund，Tom Lane）

- 修复参数化虚拟路径引发的规划器断言失败（Tom Lane）
- 在 `SnapBuildInitialSnapshot()` 的结果中设置正确的测试函数（Antonin Houska）

  核心代码不依赖此项，但某些扩展会依赖。
- 修复 Windows 上间歇出现的 “could not reattach to shared memory” 会话启动失败（Noah Misch）

  此前未识别出的一个失败来源，是为进程的默认线程池创建线程栈。现在安排在不同的内存区域分配此类栈。
- 修复 Windows 上目录扫描的错误检测（Konstantin Knizhnik）

  缺少读取目录的权限等错误未被正确检测或报告；代码反而静默地表现得像目录为空一样。
- 修复 ecpg 中的语法问题（Tom Lane）

  缺少分号导致 ecpg 程序中的 `SET *variable* = DEFAULT` 被错误转换（但 `SET *variable* TO DEFAULT` 不受影响），产生服务器会拒绝的语法无效输出。此外，在列出多个类型名的 `DROP TYPE` 或 `DROP DOMAIN` 命令中，实际只处理了第一个类型名。
- 使 ecpg 的 `CREATE TABLE AS` 语法与服务器一致（Daisuke Higuchi）
- 修复 ecpg 处理包含文件名时可能发生的缓冲区越界（Liu Huailing，Fei Wu）
- 修复未能移除目标数据目录中某些短暂文件而导致的 pg_rewind 失败（Michael Paquier）
- 让 pg_verify_checksums 验证其指定的数据目录属于正确的 PostgreSQL 版本（Michael Paquier）
- 使用远程分组或聚合的查询，其 `SELECT` 列表项为非相关子查询、外层引用或参数符号时，避免 `contrib/postgres_fdw` 发生崩溃（Tom Lane）
- 更改 `contrib/postgres_fdw`：如果被选为路由行插入目标的远程分区，同时也是同一命令中稍后将更新的 `UPDATE` 子计划目标，则报告错误（Amit Langote，Etsuro Fujita）

  以前，此类情况会导致服务器崩溃或 `UPDATE` 结果不正确。让这些情况能够正确工作，留待未来处理。
- 在 `contrib/pg_prewarm` 中，如果预热因某种原因失败，则避免无限地重新启动后台工作进程（Mithun Cy）
- 避免 `lo_unlink()` 调用失败时 `contrib/vacuumlo` 发生崩溃（Tom Lane）
- 将我们使用的时区库副本同步到 IANA tzcode 2019a 版（Tom Lane）

  这修正了 zic 的一个小缺陷，它曾使 `Africa/Casablanca` 时区在 2440 年输出不正确的转换时间；同时添加了对 zic 新选项 `-r` 的支持。
- 将时区数据文件更新到 tzdata 2019a 版，以反映巴勒斯坦和梅特拉卡特拉的夏令时法规变更，并修正以色列的历史数据。

  `Etc/UCT` 现在是指向 `Etc/UTC` 的向后兼容链接，不再是一个会生成缩写 `UCT` 的独立时区；如今，该缩写通常是拼写错误。PostgreSQL 仍会接受 `UCT` 作为输入的时区缩写，但不会输出它。

E.21. 发布版本 11.2

发布日期：. 2019-02-14

本次发布包含来自 11.1 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.21.1. 迁移到版本 11.2

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果是从早于 11.1 的版本升级，请参见第 E.22 节。

E.21.2. 变更

- 默认在 `fsync()` 失败后触发 panic，而不重试，以避免可能的数据损坏（Craig Ringer，Thomas Munro）

  某些常见操作系统在无法写出内核数据缓冲区时会丢弃它们，并将其报告为 `fsync()` 失败。如果重新发出 `fsync()` 请求，它会成功，但实际上数据已丢失，因此继续运行有数据库损坏的风险。改为触发 panic 后，我们可以从 WAL 重放；在这种情况下，WAL 可能包含数据仅存的副本。虽然这种做法显然不够理想且效率低下，但替代方案很少，所幸这种情况极少发生。

  为控制此行为，添加了新的服务器参数 data_sync_retry；如果确信内核不会在此类情况下丢弃脏数据缓冲区，可以将 `data_sync_retry` 设置为 `on`，恢复以前的行为。

- 仅在各大版本分支的文档中包含该分支的发布说明，而非在该分支及所有后续分支中重复包含（Tom Lane）

  此前政策造成的重复内容已难以控制。我们计划在项目网站上提供完整的发布说明归档，但不在每个发行版中重复包含。

- 修复分区表上带有 `INCLUDE` 列的唯一索引的处理（Álvaro Herrera）

  在这种情况下，唯一性条件未得到正确检查。

- 确保分区表的 `NOT NULL` 约束在其分区中得到遵循（Álvaro Herrera，Amit Langote）

- 分离分区时，正确更新分区表约束的目录状态（Amit Langote，Álvaro Herrera）

  以前，此类约束的 `pg_constraint.conislocal` 字段可能被错误地保留为 `false`，使其无法删除。转储/恢复或 pg_upgrade 可以解决此问题，但必要时也可以手动调整该目录字段。

- 向具有外键约束的分区表附加分区或从中分离分区时，正确创建或删除用于实施外键约束的触发器（Amit Langote，Álvaro Herrera）

- 避免在分区表中无用地创建重复外键约束（Álvaro Herrera）

- 使用 `ONLY` 在分区表上创建索引且尚无任何分区时，立即将其标记为有效（Álvaro Herrera）

  否则将无法使其变为有效。

- 分离分区时使用安全的表锁级别（Álvaro Herrera）

  以前的锁级别太弱，可能允许并发对表执行 DDL，产生不良后果。

- 修复将 ON COMMIT DROP 和 ON COMMIT DELETE ROWS 应用于分区表以及具有继承子表的表时出现的问题（Michael Paquier）
- 禁止对分区表执行 COPY FREEZE（David Rowley）

  最终应使这种用法能够工作，但所需补丁可能过于复杂，不适合冒险回补到旧版本。
- 修复索引列具有“快速默认值”时可能发生的索引损坏（即在表已包含一些行之后，通过 ALTER TABLE ADD COLUMN 添加该列，并指定了非 NULL 的常量默认值）（Andres Freund）
- 在 ALTER TABLE ... ALTER COLUMN TYPE 期间正确调整“快速默认值”（Andrew Dunstan）
- 避免获取多个缓冲区锁时可能发生的死锁（Nishant Fnu）
- 避免 GIN 清理与并发索引插入之间的死锁（Alexander Korotkov，Andrey Borodin，Peter Geoghegan）

  此更改部分撤回了版本 10.0 引入的一项性能改进，该改进曾尝试减少删除 GIN 倒排树页面时锁定的索引页数量。现在发现这会导致死锁，因此在进一步分析之前将其移除。
- 避免热备查询与 GIN 索引页面删除重放之间的死锁（Alexander Korotkov）
- 修复使用索引表达式或谓词时逻辑复制中可能发生的崩溃（Peter Eisentraut）
- 避免在表重写期间对 TOAST 数据进行无用且代价高昂的逻辑解码（Tomas Vondra）
- 修复启用同步复制时停止部分 WAL 发送进程的逻辑（Paul Guo，Michael Paquier）
- 避免在元组删除 WAL 记录中写入不正确的复制标识字段（Stas Kelvich）
- 防止向视图或外部表执行 COPY 时不正确地使用跳过 WAL 的优化（Amit Langote，Michael Paquier）
- 让归档进程在选择下一个要归档的文件时，优先选择 WAL 历史文件，而非 WAL 数据文件（David Steele）
- 修复 UPDATE 的多列 SET 子句使用子 SELECT 作为数据源时可能发生的崩溃（Tom Lane）
- 修复向 json[b]_populate_recordset() 或 json[b]_to_recordset() 输入零行时发生的崩溃（Tom Lane）
- 避免 libxml2 返回空错误消息时发生崩溃（Sergio Conde Gómez）
- 修复针对多列表（超过约 800 列）的不正确 JIT 元组解构代码（Andres Freund）
- 修复基于 hash 的分组中的性能和内存泄漏问题（Andres Freund）
- 修复因排序规则分配处理不一致而引发的虚假分组相关解析器错误（Andrew Gierth）

  在某些情况下，如果表达式包含对支持排序规则的数据类型的操作，本应被视为匹配的表达式没有被认定为匹配。
- 修复 CALL 语句参数中对排序规则敏感的表达式的解析（Peter Eisentraut）
- 确保检测到 CALL 语句参数列表中的错误后进行适当清理（Tom Lane）

- 检查 LEAST() 或 GREATEST() 底层的比较函数是否防泄漏，而非直接假定它防泄漏（Tom Lane）

  通常很难使 B-树比较函数实际泄漏信息，但原则上可能发生。

- 修复 Gather 计划节点上下都包含嵌套循环的查询的不正确规划（Tom Lane）

  如果两层嵌套循环都需要将同一个变量传入各自的右侧，就会生成不正确的计划。

- 修复必须在外部表扫描处对横向引用求值的查询的不正确规划（Tom Lane）

- 修复行比较的第一列匹配索引列，但后续列不匹配，且该索引具有内含列（非键列）时发生的规划器失败（Tom Lane）

- 修复边界情况下低估归并连接代价的问题（Tom Lane）

  当外侧键范围远小于内侧键范围时，规划器可能偏好归并连接，即使内侧存在大量重复键，使其成为低效选择。

- 避免查询包含成千上万个可使用索引的子句时，规划时间以 O(N^2) 增长（Tom Lane）

- 提高大型继承或分区表组的规划速度（Amit Langote，Etsuro Fujita）

- 改进 ANALYZE 对并发更新行的处理（Jeff Janes，Tom Lane）

  以前，正在进行的事务所删除的行会被排除在 ANALYZE 样本之外，但现在发现，这比包含它们会带来更多不一致。实际上，样本现在对应于 ANALYZE 开始时的 MVCC 快照。

- 使 TRUNCATE 忽略属于其他会话临时表的继承子表（Amit Langote，Michael Paquier）

  这使 TRUNCATE 的行为与其他命令一致。以前，此类情况通常会以失败告终。

- 修复 TRUNCATE，使其更新正确表的统计计数器（Tom Lane）

  如果被截断的表具有 TOAST 表，以前会重置 TOAST 表的计数器。

- 正确处理 ALTER TABLE ONLY ADD COLUMN IF NOT EXISTS（Greg Stark）

- 允许在热备模式下执行 UNLISTEN（Shay Rojansky）

  这必然是无操作，因为热备模式不允许 LISTEN；但允许这一空操作可以简化客户端的会话状态重置逻辑。

- 修复某些模式和数据类型权限列表中缺失的角色依赖关系（Tom Lane）

  在某些情况下，可以删除已被授予权限的角色。这不会立即引发问题，但随后的转储/重载或升级会失败，表现为尝试向名称完全由数字组成的角色授予权限。

- 防止在两阶段事务中使用会话的临时模式（Michael Paquier）

  很早以前就已禁止在此类事务中访问临时表，但对临时对象的其他操作仍可能引发问题。

- 确保添加或移除外键约束后正确更新关系缓存（Álvaro Herrera）

  这一疏漏可能导致现有会话未实施新建约束，或继续实施已删除的约束。

- 确保重命名约束后正确更新关系缓存（Amit Langote）

- 修复 GiST 索引微清理操作的重放，使并发热备查询不会看到不一致状态（Alexander Korotkov）
- 防止过快回收空 GIN 索引页而导致并发搜索失败（Andrey Borodin，Alexander Korotkov）
- 修复浮点数到整数强制转换在边界情况下的失败（Andrew Gierth，Tom Lane）

  略高于最大有效整数值的值可能未被拒绝，随后发生溢出，反而产生最小有效整数。此外，本应舍入为最小或最大整数值的值可能被错误拒绝。
- 修复对 `pg_hba.conf` 中 LDAP 认证条目的 `ldapserver` 参数中以空格分隔的主机名列表的解析（Thomas Munro）
- 发起 PAM 认证请求时，如果连接通过 Unix 套接字建立，则不设置 `PAM_RHOST` 变量（Thomas Munro）

  以前，该变量会被设置为 `[local]`，但它本应是主机名，因此这样的值至少没有什么帮助。
- 禁止将 `client_min_messages` 设置为高于 `ERROR` 的级别（Jonah Harris，Tom Lane）

  以前，可以将此变量设置为 `FATAL` 或 `PANIC`，从而禁止向客户端传送普通错误消息。但这违背了 PostgreSQL 通信协议规范中的保证，并导致某些客户端行为严重异常。在已发布分支中，修复方法是静默地将此类设置视为 `ERROR`。版本 12 及后续版本将完全拒绝这些选项。
- 修复 ecpglib，使其优先使用 `uselocale()` 或 `_configthreadlocale()`，而非 `setlocale()`（Michael Meskes，Tom Lane）

  由于 `setlocale()` 不局限于线程，甚至可能不是线程安全的，以前的代码会给多线程 ecpg 应用带来问题。
- 修复通过 ecpg SQLDA（SQL 描述符区域）传递的数值数据结果不正确的问题（Daisuke Higuchi）

  带有前导零的值未被正确复制。
- 修复 psql 的 `\g` *`target`* 元命令，使其适用于 `COPY TO STDOUT`（Daniel Vérité）

  以前，*`target`* 选项被忽略，因此复制的数据始终发送到当前查询输出目标。
- 使 psql 的 LaTeX 输出格式正确呈现特殊字符（Tom Lane）

  反斜线和某些其他 ASCII 标点字符未被正确呈现，导致文档语法错误或输出中出现错误字符。
- 指定 `--random-seed=`*`N`* 时，使 pgbench 的随机数生成完全确定且与平台无关（Fabien Coelho，Tom Lane）

  在任何特定平台上，使用某个 *`N`* 值获得的序列，很可能与应用此补丁之前不同。
- 修复 pg_basebackup 和 pg_verify_checksums，使其适当地忽略临时文件（Michael Banck，Michael Paquier）
- 修复 pg_dump 对间接依赖主键的物化视图的处理（Tom Lane）

  这曾导致此类视图的转储归档条目被错误标记，产生无害的 “archive items not in correct section order” 警告；更有影响的是，依赖这些标记的选择性恢复选项（如 `--section`）可能出现错误行为。

- 使 pg_dump 包含 `ALTER INDEX SET STATISTICS` 命令（Michael Paquier）

  加入为索引表达式设置统计目标的能力时，我们忘记让 pg_dump 支持它，因此这些设置会在转储/重载中丢失。

- 修复 pg_dump 对具有 OID 的表的转储（Peter Eisentraut）

  如果 `WITH OIDS` 子句需要应用于第一个被转储的表，它会被遗漏。

- 避免 pg_dump 或 pg_restore 尝试报告错误时，在某些平台上因空指针解引用而崩溃（Tom Lane）

- 防止行内压缩数据导致 `contrib/amcheck` 错误报告索引损坏（Peter Geoghegan）

- 如果 `COPY FROM PROGRAM` 提前停止读取程序输出，则适当地忽略 `SIGPIPE` 错误（Tom Lane）

  实际上，直接使用 `COPY` 不会出现这种情况，但使用 `contrib/file_fdw` 时可能发生。

- 修复 `contrib/hstore`，使其为版本 8.4 或更早版本创建的空 `hstore` 值计算正确的 hash 值（Andrew Gierth）

  以前的代码产生的结果，与较新版本创建的空 `hstore` 值不同，因此可能导致哈希连接或 hash 聚合结果不正确。如果表中可能包含最早在 8.4 时存储、此后从未经过转储/重新加载的数据，建议重建 `hstore` 列上的所有 hash 索引。

- 避免向 `contrib/intarray` 的 `gist__int_ops` 索引支持代码提供大量输入时发生崩溃或运行时间过长（Andrew Gierth）

- 在 configure 中，如果找不到 `python`，则依次查找 `python3` 和 `python2`（Peter Eisentraut）

  这使不再提供无版本号的 `python` 可执行文件的平台，无需显式指定 `PYTHON` 即可配置 PL/Python。

- 将 JIT 相关头文件包含在安装的头文件集合中（Donald Dong）

- 在 pgxs 构建中支持新的 Makefile 变量 `PG_CFLAGS`、`PG_CXXFLAGS` 和 `PG_LDFLAGS`（Christoph Berg）

  这简化了扩展构建过程的自定义。

- 修复以 Perl 编写的构建脚本，使其不再假定搜索路径包含“.”，因为近期 Perl 版本已不再包含它（Andrew Dunstan）

- 修复 OpenBSD 上服务器命令行选项的解析问题（Tom Lane）

- 调整 `set_rel_pathlist_hook` 调用的位置，使扩展可以用它为并行查询提供部分路径（KaiGai Kohei）

  预计这不会影响现有使用场景。

- 将时区数据文件更新到 tzdata 2018i 版，以反映哈萨克斯坦、梅特拉卡特拉及圣多美和普林西比的夏令时法规变更。由于部分地区未更改 UTC 偏移量，哈萨克斯坦的 Qyzylorda 时区被一分为二，创建了新时区 Asia/Qostanay。同时修正了香港及众多太平洋岛屿的历史数据。

E.22. 发布版本 11.1

发布日期：. 2018-11-08

本次发布包含来自 11.0 的多项修复。有关 11 主版本新特性的说明，请参见第 E.23 节。

E.22.1. 迁移到版本 11.1

对于运行 11.X 的用户，不需要执行导出/恢复。

但是，如果使用了 `pg_stat_statements` 扩展，请参阅下方相关变更日志条目。

E.22.2. 变更

- 确保 pg_dump 输出 `CREATE TRIGGER ... REFERENCING` 命令时，正确处理过渡表名称的引号（Tom Lane）

  无特权用户可以利用这一疏漏，在下一次转储/重载或运行 pg_upgrade 时取得超级用户权限。（CVE-2018-16850）

- 创建子索引时，使用为分区索引指定的表空间（Álvaro Herrera）

  以前，子索引始终创建在默认表空间中。

- 修复并行哈希多批次左连接中的 NULL 处理（Andrew Gierth，Thomas Munro）

  hash 键为空值的外侧关系行被遗漏在连接结果之外。

- 修复对具有常量测试表达式的 `CASE` 子句内数组类型强制转换表达式的不正确处理（Tom Lane）

- 修复对缺少最近新增列的元组的不正确展开（Andrew Dunstan，Amit Langote）

  已知这会使最近添加过列的表上的触发器崩溃，也可能出现其他症状。

- 修复 `CALL` 参数列表中命名参数或默认参数的缺陷（Tom Lane，Pavel Stehule）

- 修复带有 `ORDER BY` 列的严格聚合的严格性检查（Andrew Gierth，Andres Freund）

  严格性逻辑错误地忽略了 `ORDER BY` 值为空的行。

- 禁用 `recheck_on_update` 优化（Tom Lane）

  事实证明，这项 v11 新增特性尚未成熟。在能够解决问题之前将其禁用。

- 禁止在附着于父表的触发器中创建该表的分区（Amit Langote）

  理想情况下应允许这样做，但目前必须禁止，以避免崩溃。

- 修复将 `ON COMMIT DELETE ROWS` 应用于分区临时表时出现的问题（Amit Langote）

- 修复字符类检查，使其在 Windows 上不会因高于 U+FFFF 的 Unicode 字符而失败（Tom Lane，Kenji Uno）

  此缺陷影响全文检索操作，以及 `contrib/ltree` 和 `contrib/pg_trgm`。

- 确保服务器在等待客户端输入之前，处理已收到的 NOTIFY 和 SIGTERM 中断（Jeff Janes，Tom Lane）
- 修复重复扫描 SP-GiST 索引时的内存泄漏（Tom Lane）

  目前仅已知在使用 SP-GiST 的排除约束于单条命令中接收大量新索引条目时，这种泄漏才会达到显著程度。
- 如果 wal_level 设置过低，无法支持现有复制槽，则禁止启动服务器（Andres Freund）
- 修复 psql 以及文档示例，使其在每次调用 PQnotifies() 之前调用 PQconsumeInput()（Tom Lane）

  这修复了 psql 直到下一条命令之后才报告收到 NOTIFY 消息的情况。
- 修复 pg_verify_checksums 对哪些文件应检查校验和的判断（Michael Paquier）

  在某些情况下，它会对本来就不应有校验和的文件报错。
- 在 contrib/pg_stat_statements 中，禁止 pg_read_all_stats 角色执行 pg_stat_statements_reset()（Haribabu Kommi）

  pg_read_all_stats 只应授予读取统计信息的权限，而非更改统计信息的权限，因此此前的授权不正确。

  要使此项修改生效，请在每个已安装 pg_stat_statements 的数据库中运行 ALTER EXTENSION pg_stat_statements UPDATE。（在 11.0 中新建的数据库应该不需要这样做，但从旧发行版升级的数据库可能仍包含旧版 pg_stat_statements。如果模块已更新，执行 UPDATE 命令也不会有害。）
- 重命名红黑树支持函数，使用 rbt 前缀代替 rb 前缀（Tom Lane）

  这避免了与 Ruby 函数发生名称冲突，后者曾使 PL/Ruby 失效。希望没有其他扩展受到影响。
- 修复 macOS 10.14（Mojave）上的构建问题（Tom Lane）

  调整 configure，向 CPPFLAGS 添加 -isysroot 开关；否则，PL/Perl 和 PL/Tcl 在 macOS 10.14 上无法配置或构建。可以在 configure 或 make 的参数中设置 PG_SYSROOT 变量，在配置时或构建时覆盖所用的具体 sysroot。

  现在建议 Perl 相关扩展在编译器标志中写入 $(perl_includespec)，而非 -I$(perl_archlibexp)/CORE。后者在大多数平台上仍然有效，但在近期 macOS 上无效。

  此外，在近期 macOS 版本上构建 PL/Tcl 时，应不再需要手动指定 --with-tclconfig。
- 修复 MSVC 构建和回归测试脚本，使其适用于近期 Perl 版本（Andrew Dunstan）

  Perl 默认已不再将当前目录包含在搜索路径中，现在对此进行适配。
- 在 Windows 上，允许使用 Administrator 账户运行回归测试（Andrew Dunstan）

  为安全地做到这一点，pg_regress 现在会在启动时放弃所有此类特权。
- 将时区数据文件更新到 tzdata 2018g 版，以反映智利、斐济、摩洛哥和俄罗斯（伏尔加格勒）的夏令时法规变更，并修正中国、夏威夷、日本、澳门和朝鲜的历史数据。

E.23. 发布版本 11

发布日期：. 2018-10-18

E.23.1. 概述

PostgreSQL 11 的主要增强包括：

- 分区功能改进，包括：
  - 添加按 hash 键分区的支持
  - 添加分区表上的 PRIMARY KEY、FOREIGN KEY、索引和触发器支持
  - 允许创建“默认”分区，用于存储不匹配其他任何分区的数据
  - 更改分区键列的 UPDATE 语句现在会将受影响的行移动到适当的分区
  - 通过增强查询规划和执行期间的分区排除策略，提高 SELECT 性能
- 并行执行改进，包括：
  - CREATE INDEX 现在可以在构建 B-树索引时使用并行处理
  - CREATE TABLE ... AS、CREATE MATERIALIZED VIEW 以及某些使用 UNION 的查询现在可以并行执行
  - 提高并行哈希连接和并行顺序扫描的性能
- 支持内嵌事务的 SQL 存储过程
- 为部分 SQL 代码提供可选的即时（JIT）编译，加快表达式求值
- 窗口函数现在支持 SQL:2011 标准中的所有窗口帧选项，包括 RANGE *distance* PRECEDING/FOLLOWING、GROUPS 模式和帧排除选项
- 现在可以使用 CREATE INDEX 的 INCLUDE 子句创建覆盖索引
- 许多其他有用的性能改进，包括在使用非空列默认值执行 ALTER TABLE ... ADD COLUMN 时避免重写表

以下各节将更详细地说明上述项目。

E.23.2. 迁移到版本 11

对于希望从任何先前版本迁移数据的用户，需要使用pg_dumpall 进行导出/恢复，或使用 pg_upgrade 或逻辑复制。有关迁移到新主版本的一般信息，请参见第 18.6 节。

版本 11 包含许多可能影响与先前版本兼容性的变更。请注意以下不兼容性：

- 使 pg_dump 转储数据库的属性，而不只是内容（Haribabu Kommi）

  以前，数据库本身的属性，例如数据库级 GRANT/REVOKE 权限和 ALTER DATABASE SET 变量设置，只能由pg_dumpall 转储。现在，pg_dump --create 和 pg_restore --create 除恢复数据库内的对象外，也会恢复这些数据库属性。pg_dumpall -g 现在仅转储角色和表空间相关属性。pg_dumpall 的完整输出（不带 -g）保持不变。

  如果没有指定 --create，pg_dump 和 pg_restore 不再转储/恢复数据库级注释和安全标签；这些内容现在被视为数据库的属性。

  pg_dumpall 的输出脚本现在始终使用数据库原有的区域设置和编码创建数据库，因此如果目标系统不认识该区域设置或编码名称，操作就会失败。以前，如果数据库的区域设置和编码与旧集群的默认值一致，输出的 CREATE DATABASE 就不会包含这些指定。

  pg_dumpall --clean 现在会恢复 postgres 和 template1 数据库的原始区域设置和编码，以及用户创建数据库的相应设置。

- 在消除函数引用与列引用之间的歧义时，考虑语法形式（Tom Lane）

  当 *x* 是表名或复合列时，PostgreSQL 传统上将语法形式 *f*(*x*) 和 *x*.*f* 视为等价，从而允许编写函数后，将其当作按需求值的列来使用。然而，如果两种解释都可行，以前总会选择列解释；当用户本意是调用函数时，这会产生意外结果。现在，若存在歧义，会选择与语法形式相符的解释。

- 全面强制表约束名和域约束名的唯一性（Tom Lane）

  PostgreSQL 要求同一表的约束名称各不相同，域的约束名称也是如此。但以前对此没有严格强制检查，存在可创建重复名称的边界情况。

- 使 power(numeric, numeric) 和 power(float8, float8) 按照 POSIX 标准处理 NaN 输入（Tom Lane，Dang Minh Huong）

  POSIX 规定 NaN ^ 0 = 1 且 1 ^ NaN = 1，但其他所有包含 NaN 输入的情况都应返回 NaN。power(numeric, numeric) 以前在所有此类情况下都返回 NaN，现在会遵循这两个例外。只要 C 库符合标准，power(float8, float8) 就会符合标准；但某些旧 Unix 平台的库不符合标准，某些 Windows 版本上也存在问题。

- 当模板分隔符不匹配时，防止 to_number() 消耗字符（Oliver Ford）

  具体而言，SELECT to_number('1234', '9,999') 以前返回 134，现在会返回 1234。L 和 TH 现在只消耗数字、正负号、小数点和逗号以外的字符。

- 修复 to_date()、to_number() 和 to_timestamp()，使其对每个模板字符跳过一个字符（Tom Lane）

  以前，它们对模板字符的每个字节跳过一个字节，如果任一字符串包含多字节字符，就会出现异常行为。

- 调整 to_char()、to_number() 和 to_timestamp() 对模板字符串中双引号内反斜线的处理。

  这种反斜线现在会转义其后的字符，尤其是双引号或另一个反斜线。

- 正确处理 xmltable()、xpath() 及其他 XML 处理函数中的相对路径表达式（Markus Winand）

  按照 SQL 标准，相对路径从 XML 输入文档的文档节点开始，而非像这些函数以前那样从根节点开始。

- 在扩展查询协议中，使 statement_timeout 分别应用于每条 Execute 消息，而非应用于 Sync 之前的所有命令（Tatsuo Ishii，Andres Freund）

- 从系统目录 pg_class 中移除 relhaspkey 列（Peter Eisentraut）

  需要检查主键的应用应查询 pg_index。

- 将系统目录 pg_proc 的 proisagg 和 proiswindow 列替换为 prokind（Peter Eisentraut）

  这个新列更清楚地区分了函数、过程、聚合和窗口函数。

- 修正信息模式列 tables.table_type，使其返回 FOREIGN 而非 FOREIGN TABLE（Peter Eisentraut）

  这种新输出符合 SQL 标准。

- 更改 ps 进程显示中的后台工作进程标签，使其与 pg_stat_activity.backend_type 标签一致（Peter Eisentraut）

- 在通过 lo_open() 打开大对象时检查大对象权限，而非在尝试读写时检查（Tom Lane，Michael Paquier）

  如果请求写入权限但无法获得，现在即使最终从未写入该大对象，也会抛出错误。

- 防止非超级用户重建共享目录的索引（Michael Paquier，Robert Haas）

  以前，数据库所有者也被允许这样做，但现在认为这超出了其权限范围。

- 移除已弃用的 adminpack 函数 pg_file_read()、pg_file_length() 和 pg_logfile_rotate()（Stephen Frost）

  核心后端现在已提供等效功能。现有 adminpack 安装在通过 ALTER EXTENSION ... UPDATE 更新之前，仍可访问这些函数。

- 遵循双引号内命令选项的大小写（Daniel Gustafsson）

  以前，某些 SQL 命令中的选项名即使以双引号输入，也会被强制转换为小写；因此，例如 "FillFactor" 会被接受为索引存储选项，尽管其正确名称应为小写。现在，此类情况将产生错误。

- 移除服务器参数 replacement_sort_tuples（Peter Geoghegan）

  经判断，置换排序已不再有用。

- 移除 CREATE FUNCTION 中的 WITH 子句（Michael Paquier）

PostgreSQL 很早以前就已支持更符合标准的语法来实现此功能。

- 在 PL/pgSQL 触发器函数中，未赋值的 `OLD` 和 `NEW` 变量现在读取为 NULL（Tom Lane）

  以前，对这些变量的引用可以解析，但无法执行。

E.23.3. 变更

以下是 PostgreSQL 11 与上一个主版本之间变更的详细说明。

E.23.3.1. 服务器

E.23.3.1.1. 分区

- 允许根据键列的 hash 值创建分区（Amul Sul）
- 支持分区表上的索引（Álvaro Herrera，Amit Langote）

  分区表上的“索引”并非跨越整个分区表的物理索引，而是用于在表的每个分区上自动创建类似索引的模板。

  如果索引列集合包含分区键，则可以将分区索引声明为 `UNIQUE`。它将表示覆盖整个分区表的有效唯一约束，尽管每个物理索引只在其自身分区内实施唯一性。

  新命令 `ALTER INDEX ATTACH PARTITION` 将分区上的现有索引与其分区表的匹配索引模板关联起来。这为给现有分区表设置新的分区索引提供了灵活性。
- 允许在分区表上定义外键（Álvaro Herrera）
- 允许在分区表上定义 `FOR EACH ROW` 触发器（Álvaro Herrera）

  在分区表上创建触发器，会自动在所有现有和未来分区上创建触发器。这也使分区表能够使用延迟唯一约束。
- 允许分区表具有默认分区（Jeevan Ladhe，Beena Emerson，Ashutosh Bapat，Rahila Syed，Robert Haas）

  默认分区存储不匹配其他任何已定义分区的行，并据此被搜索。
- 更改分区键列的 `UPDATE` 语句现在会将受影响的行移动到适当的分区（Amit Khandekar）
- 允许分区表上的 `INSERT`、`UPDATE` 和 `COPY` 将行正确路由到外部分区（Etsuro Fujita，Amit Langote）

  `postgres_fdw` 外部表支持此功能。由于为此调用 `ExecForeignInsert` 回调函数的方式与以前不同，外部数据包装器必须相应修改以适应此更改。
- 加快查询处理期间的分区排除（Amit Langote，David Rowley，Dilip Kumar）

  这加快了对具有大量分区的分区表的访问。
- 允许在查询执行期间排除分区（David Rowley，Beena Emerson）

  以前，分区排除仅在规划时进行，因此许多连接和预备查询无法使用分区排除。
- 在分区表之间的等值连接中，允许直接连接匹配的分区（Ashutosh Bapat）

此特性默认禁用，但可以通过更改 enable_partitionwise_join 启用。

- 允许对分区表上的聚合函数按分区分别求值，随后合并结果（Jeevan Chalke，Ashutosh Bapat，Robert Haas）

  此特性默认禁用，但可以通过更改 enable_partitionwise_aggregate 启用。

- 允许 postgres_fdw 将聚合下推到作为分区的外部表（Jeevan Chalke）

E.23.3.1.2. 并行查询

- 允许并行构建 B-树索引（Peter Geoghegan，Rushabh Lathia，Heikki Linnakangas）
- 允许使用共享 hash 表并行执行哈希连接（Thomas Munro）
- 如果各条 SELECT 无法分别并行化，则允许 UNION 并行运行各条 SELECT（Amit Khandekar，Robert Haas，Amul Sul）
- 允许分区扫描更高效地使用并行工作进程（Amit Khandekar，Robert Haas，Amul Sul）
- 允许将 LIMIT 传递给并行工作进程（Robert Haas，Tom Lane）

  这使工作进程能够减少返回结果，并使用有针对性的索引扫描。

- 允许单次求值查询（例如 WHERE 子句中的聚合查询）以及目标列表中的函数并行执行（Amit Kapila，Robert Haas）
- 添加服务器参数 parallel_leader_participation，以控制领导者是否也执行子计划（Thomas Munro）

  默认启用，意味着领导者将执行子计划。

- 允许并行执行 CREATE TABLE ... AS、SELECT INTO 和 CREATE MATERIALIZED VIEW 命令（Haribabu Kommi）
- 提高使用大量并行工作进程时顺序扫描的性能（David Rowley）
- 在 EXPLAIN 中添加并行工作进程排序活动的报告（Robert Haas，Tom Lane）

E.23.3.1.3. 索引

- 允许 B-树索引包含不属于搜索键或唯一约束、但可供仅索引扫描读取的列（Anastasia Lubennikova，Alexander Korotkov，Teodor Sigaev）

  这通过 CREATE INDEX 的新 INCLUDE 子句启用。它便于构建优化特定查询类型的“覆盖索引”。即使列的数据类型不支持 B-树，也可以将其包含在内。

- 提高单调递增的索引插入的性能（Pavan Deolasee，Peter Geoghegan）
- 提高 hash 索引扫描的性能（Ashutosh Sharma）
- 为 hash、GiST 和 GIN 索引添加谓词锁定（Shubham Barai）

  这降低了可串行化模式事务发生串行化冲突的可能性。

E.23.3.1.3.1. SP-Gist

- 添加前缀匹配操作符 `text ^@ text`，并由 SP-GiST 支持（Ildus Kurbangaliev）

  这类似于配合 B-树索引使用 *var* `LIKE 'word%'`，但效率更高。

- 允许使用 SP-GiST 对多边形建立索引（Nikita Glukhov，Alexander Korotkov）
- 允许 SP-GiST 对叶子键使用有损表示（Teodor Sigaev，Heikki Linnakangas，Alexander Korotkov，Nikita Glukhov）

E.23.3.1.4. 优化器

- 改进统计信息中高频值的选择（Jeff Janes，Dean Rasheed）

  以前，根据某个值相对于所有列值的频率来识别高频值（MCV）。现在，根据其相对于非 MCV 值的频率选择 MCV。这提高了算法在均匀分布和非均匀分布下的稳健性。

- 改进 >= 和 <= 的选择率估算（Tom Lane）

  以前，除非比较常量为 MCV，这些情况分别采用与 > 和 < 相同的选择率估算。此更改对涉及小范围 `BETWEEN` 的查询尤其有帮助。

- 在等价的情况下，将 *var* = *var* 简化为 *var* `IS NOT NULL`（Tom Lane）

  这会改善选择率估算。

- 改进优化器对 `EXISTS` 和 `NOT EXISTS` 查询的行数估算（Tom Lane）
- 使优化器考虑 `HAVING` 子句的求值代价和选择率（Tom Lane）

E.23.3.1.5. 一般性能

- 为查询计划的某些部分添加即时（JIT）编译，以提高执行速度（Andres Freund）

  此特性要求 LLVM 可用。目前，即使构建支持它，默认也不启用。

- 允许位图扫描在可能时执行仅索引扫描（Alexander Kuzmenkov）
- 在 `VACUUM` 期间更新空闲空间映射（Claudio Freire）

  这使空闲空间能够更快地被复用。

- 允许 `VACUUM` 避免不必要的索引扫描（Masahiko Sawada，Alexander Korotkov）
- 提高提交多个并发事务的性能（Amit Kapila）
- 减少目标列表中使用返回集合函数的查询的内存用量（Andres Freund）
- 提高聚合计算的速度（Andres Freund）
- 允许 `postgres_fdw` 将使用连接的 `UPDATE` 和 `DELETE` 下推到外部服务器（Etsuro Fujita）

以前，只会下推不含连接的 UPDATE 和 DELETE。

- 添加对 Windows 上大页的支持（Takayuki Tsunakawa，Thomas Munro）

  这由 huge_pages 配置参数控制。

E.23.3.1.6. 监控

- 在 log_statement_stats、log_parser_stats、log_planner_stats 和 log_executor_stats 的输出中显示内存用量（Justin Pryzby，Peter Eisentraut）
- 添加 pg_stat_activity.backend_type 列，以显示后台工作进程的类型（Peter Eisentraut）

  该类型在 ps 输出中也可见。

- 使 log_autovacuum_min_duration 记录因正被并发删除而跳过的表（Nathan Bossart）

E.23.3.1.6.1. 信息模式

- 添加与表约束和触发器相关的 information_schema 列（Peter Eisentraut）

  具体而言，triggers.action_order、triggers.action_reference_old_table 和 triggers.action_reference_new_table 现在会填入值，而以前始终为空。此外，table_constraints.enforced 列现在已存在，但尚未填入有用的值。

E.23.3.1.7. Authentication

- 允许服务器在 search+bind 模式下使用更复杂的 LDAP 配置（Thomas Munro）

  具体而言，ldapsearchfilter 允许组合 LDAP 属性进行模式匹配。

- 允许 LDAP 认证使用加密的 LDAP（Thomas Munro）

  以前已可通过 ldaptls=1 支持基于 TLS 的 LDAP。这种用于加密 LDAP 的新 TLS LDAP 方式，通过 ldapscheme=ldaps 或 ldapurl=ldaps:// 启用。

- 改进 LDAP 错误的日志记录（Thomas Munro）

E.23.3.1.8. 权限

- 添加允许文件系统访问的默认角色（Stephen Frost）

  具体来说，新增角色是 pg_read_server_files、pg_write_server_files 和 pg_execute_server_program。这些角色现在还控制谁可以使用服务器端的 COPY 和 file_fdw 扩展。以前，只有超级用户可以使用这些功能，现在默认行为仍然如此。

- 允许通过 GRANT/REVOKE 权限控制对文件系统函数的访问，取代超级用户检查（Stephen Frost）

  具体而言，修改了以下函数：pg_ls_dir()、pg_read_file()、pg_read_binary_file()、pg_stat_file()。

- 使用 GRANT/REVOKE 控制对 lo_import() 和 lo_export() 的访问（Michael Paquier，Tom Lane）

以前，只有超级用户获准访问这些函数。

编译时选项 ALLOW_DANGEROUS_LO_FUNCTIONS 已被移除。

- 在阻止以非密码方式访问 postgres_fdw 表时，使用视图所有者而非会话所有者（Robert Haas）

  PostgreSQL 只允许超级用户在不使用密码的情况下访问 postgres_fdw 表，例如通过 peer 认证。以前，要允许此类访问，会话所有者必须是超级用户；现在改为检查视图所有者。

- 修复对视图执行 SELECT FOR UPDATE 时无效的锁定权限检查（Tom Lane）

E.23.3.1.9. 服务器配置

- 添加服务器设置 ssl_passphrase_command，以允许为 SSL 密钥文件提供口令（Peter Eisentraut）

  同时添加 ssl_passphrase_command_supports_reload，用于指定服务器配置重新加载期间，是否应重新加载 SSL 配置并调用 ssl_passphrase_command。

- 添加存储参数 toast_tuple_target，以控制开始考虑 TOAST 存储的最小元组长度（Simon Riggs）

  默认的 TOAST 阈值没有改变。

- 允许以字节为单位指定与内存和文件大小相关的服务器选项（Beena Emerson）

  新增单位后缀为“B”，与现有的“kB”、“MB”、“GB”和“TB”并用。

E.23.3.1.10. 预写式日志（WAL）

- 允许在 initdb 期间设置 WAL 文件大小（Beena Emerson）

  以前，16MB 的默认值只能在编译时更改。

- 仅保留一个检查点周期的 WAL 数据（Simon Riggs）

  以前，会保留两个检查点周期的 WAL。

- 将被强制切换的 WAL 段文件的未使用部分填零，以提高可压缩性（Chapman Flack）

E.23.3.2. 基础备份与流复制

- 使用逻辑复制时复制 TRUNCATE 操作（Simon Riggs，Marco Nenciarini，Peter Eisentraut）

- 将预备事务信息传递给逻辑复制订阅者（Nikhil Sontakke，Stas Kelvich）

- 从流式基础备份中排除不记录 WAL 的表、临时表和 pg_internal.init 文件（David Steele）

  没有必要复制此类文件。

- 允许在流式基础备份期间验证堆页面的校验和（Michael Banck）

- 允许以编程方式推进复制槽，而无需由订阅者消费（Petr Jelinek）

这使复制槽在内容无需被消费时能够高效推进。相应操作通过 pg_replication_slot_advance() 执行。

- 向 backup_label 文件添加时间线信息（Michael Paquier）

  同时添加检查，确保 WAL 时间线与 backup_label 文件的时间线匹配。

- 向 pg_stat_wal_receiver 系统视图添加主机和端口连接信息（Haribabu Kommi）

E.23.3.3. 工具命令

- 允许 ALTER TABLE 添加具有非空默认值的列，而无需重写表（Andrew Dunstan，Serge Rielau）

  默认值为常量时会启用此优化。

- 允许通过锁定底层表来锁定视图（Yugo Nagata）

- 允许 ALTER INDEX 设置表达式索引的统计收集目标（Alexander Korotkov，Adrien Nayrat）

  在 psql 中，\d+ 现在会显示索引的统计目标。

- 允许在一条 VACUUM 或 ANALYZE 命令中指定多个表（Nathan Bossart）

  此外，如果 VACUUM 中提到的任何表使用了列列表，就必须提供 ANALYZE 关键字；以前，此类情况会隐含执行 ANALYZE。

- 为 ANALYZE 添加括号包围的选项语法（Nathan Bossart）

  这与 VACUUM 支持的语法类似。

- 添加 CREATE AGGREGATE 选项，以指定聚合最终函数的行为（Tom Lane）

  这有助于优化用户定义的聚合函数，并使其能够用作窗口函数。

E.23.3.4. 数据类型

- 允许创建域的数组（Tom Lane）

  这也允许对域使用 array_agg()。

- 支持基于复合类型的域（Tom Lane）

  同时允许 PL/Perl、PL/Python 和 PL/Tcl 处理复合类型域的函数参数和结果，并改进 PL/Python 的域处理。

- 添加从 JSONB 标量到数值和布尔数据类型的类型转换（Anastasia Lubennikova）

E.23.3.5. 函数

- 添加 SQL:2011 规定的所有窗口函数帧选项（Oliver Ford，Tom Lane）

具体而言，允许 RANGE 模式使用 PRECEDING 和 FOLLOWING，选取分组值位于加减指定偏移量范围内的行。添加 GROUPS 模式，以包含前后指定数量的同等行组。同时添加了帧排除语法。

- 添加 SHA-2 系列 hash 函数（Peter Eisentraut）

  具体添加了 sha224()、sha256()、sha384() 和 sha512()。

- 添加对 64 位非密码学 hash 函数的支持（Robert Haas，Amul Sul）

- 允许 to_char() 和 to_timestamp() 以小时和分钟指定时区相对于 UTC 的偏移量（Nikita Glukhov，Andrew Dunstan）

  这通过格式指定符 TZH 和 TZM 完成。

- 添加文本检索函数 websearch_to_tsquery()，支持类似网页搜索引擎所用的查询语法（Victor Drobny，Dmitry Ivanov）

- 添加函数 json(b)_to_tsvector()，以创建用于匹配 JSON/JSONB 值的文本检索查询（Dmitry Dolgov）

E.23.3.6. 服务器端语言

- 添加 SQL 级过程，使其能够开始并提交自己的事务（Peter Eisentraut）

  它们使用新的 CREATE PROCEDURE 命令创建，并通过 CALL 调用。

  新的 ALTER/DROP ROUTINE 命令允许更改/删除所有例程类对象，包括过程、函数和聚合。

  另外，在 CREATE OPERATOR 和 CREATE TRIGGER 中，现在推荐使用 FUNCTION 而非 PROCEDURE，因为所引用的对象必须是函数，而不能是过程。不过，为保持兼容，仍然接受旧语法。

- 为 PL/pgSQL、PL/Perl、PL/Python、PL/Tcl 和 SPI 服务器端语言添加事务控制（Peter Eisentraut）

  事务控制仅可用于事务顶层的过程，以及嵌套的 DO 和 CALL 块；嵌套层次中只能包含其他 DO 和 CALL 块。

- 添加将 PL/pgSQL 复合类型变量定义为非空、常量或带初始值的能力（Tom Lane）

- 允许 PL/pgSQL 处理同一会话中首次和后续函数执行之间发生的复合类型（如 record、row）变更（Tom Lane）

  以前，这类情况会产生错误。

- 添加扩展 jsonb_plpython，以在 JSONB 和 PL/Python 类型之间进行转换（Anthony Bykov）

- 添加扩展 jsonb_plperl，以在 JSONB 和 PL/Perl 类型之间进行转换（Anthony Bykov）

E.23.3.7. 客户端接口

- 更改 libpq，使其默认禁用压缩（Peter Eisentraut）

现代 OpenSSL 版本已禁用压缩，因此 libpq 的这一设置对这些库并无影响。

- 为 ecpg 的 `WHENEVER` 语句添加 `DO CONTINUE` 选项（Vinayak Pokale）

  这会生成 C `continue` 语句，在指定条件发生时返回所在循环的顶部。

- 添加 ecpg 模式，以启用 Oracle Pro*C 风格的字符数组处理。

  此模式通过 `-C` 启用。

E.23.3.8. 客户端应用程序

E.23.3.8.1. psql

- 添加 psql 命令 `\gdesc`，以显示查询结果中各列的名称和类型（Pavel Stehule）
- 添加 psql 变量，以报告查询活动和错误（Fabien Coelho）

  具体新增变量为 `ERROR`、`SQLSTATE`、`ROW_COUNT`、`LAST_ERROR_MESSAGE` 和 `LAST_ERROR_SQLSTATE`。

- 允许 psql 测试变量是否存在（Fabien Coelho）

  具体而言，语法 `:{?variable_name}` 允许在 `\if` 语句中测试变量是否存在。

- 允许环境变量 `PSQL_PAGER` 控制 psql 的分页器（Pavel Stehule）

  这允许用独立的环境变量指定 psql 的默认分页器，将其与其他应用的分页器设置分开。如果未设置 `PSQL_PAGER`，仍会遵循 `PAGER`。

- 使 psql 的 `\d+` 命令始终显示表的分区信息（Amit Langote，Ashutosh Bapat）

  以前，如果分区表没有分区，就不会显示其分区信息。现在还会指出哪些分区本身也是分区表。

- 确保 psql 在提示输入密码时显示正确的用户名（Tom Lane）

  以前，同时使用 `-U` 和嵌入 URI 的用户名会导致显示错误的用户名。另外，在指定 `--password` 时，不再在密码提示前显示用户名。

- 在此前没有输入的情况下，允许 `quit` 和 `exit` 退出 psql（Bruce Momjian）

  当输入缓冲区非空，而 `quit` 和 `exit` 单独出现在一行时，也会输出如何退出的提示。为 `help` 添加类似提示。

- 当 `\q` 单独输入在一行上却被忽略时，让 psql 提示使用 control-D（Bruce Momjian）

  例如，在字符串内输入 `\q` 不会退出。

- 改进 `ALTER INDEX RESET/SET` 的 Tab 补全（Masahiko Sawada）
- 添加基础支持，使 psql 能够根据服务器版本调整 Tab 补全查询（Tom Lane）

  以前，对旧版服务器执行 Tab 补全查询可能失败。

E.23.3.8.2. pgbench

- 为 pgbench 表达式添加对 NULL、布尔值以及某些函数和操作符的支持（Fabien Coelho）
- 为 pgbench 添加 `\if` 条件支持（Fabien Coelho）
- 允许在 pgbench 变量名中使用非 ASCII 字符（Fabien Coelho）
- 添加 pgbench 选项 `--init-steps`，以控制执行哪些初始化步骤（Masahiko Sawada）
- 为 pgbench 添加近似服从 Zipf 分布的随机数生成器（Alik Khilazhev）
- 允许在 pgbench 中设置随机种子（Fabien Coelho）
- 允许 pgbench 使用 `pow()` 和 `power()` 执行幂运算（Raúl Marín Rodríguez）
- 为 pgbench 添加 hash 函数（Ildar Musin）
- 提高使用 `--latency-limit` 和 `--rate` 时 pgbench 统计信息的准确性（Fabien Coelho）

E.23.3.9. 服务器应用程序

- 为 pg_basebackup 添加创建命名复制槽的选项（Michael Banck）

  使用 WAL 流式方法（`--wal-method=stream`）时，`--create-slot` 选项会创建指定名称的复制槽（`--slot`）。

- 允许 initdb 为数据目录设置组读取权限（David Steele）

  这通过新的 initdb 选项 `--allow-group-access` 完成。管理员也可以在运行 initdb 之前，为空数据目录设置组权限。服务器变量 `data_directory_mode` 允许读取数据目录的组权限。

- 添加 pg_verify_checksums 工具，以在离线状态下验证数据库校验和（Magnus Hagander）
- 允许 pg_resetwal 通过 `--wal-segsize` 更改 WAL 段大小（Nathan Bossart）
- 为 pg_resetwal 和 pg_controldata 添加长选项（Nathan Bossart，Peter Eisentraut）
- 为 pg_receivewal 添加 `--no-sync` 选项，用于测试时禁止同步 WAL 写入（Michael Paquier）
- 添加 pg_receivewal 选项 `--endpos`，以指定何时停止接收 WAL（Michael Paquier）
- 允许 pg_ctl 向进程发送 `SIGKILL` 信号（Andres Freund）

  以前因担心可能被误用而不支持此功能。

- 减少 pg_rewind 复制的文件数量（Michael Paquier）
- 禁止以 `root` 身份运行 pg_rewind（Michael Paquier）

E.23.3.9.1. pg_dump, pg_dumpall, pg_restore

- 添加 pg_dumpall 选项 `--encoding`，以控制输出编码（Michael Paquier）

pg_dump 已有此选项。

- 添加 pg_dump 选项 `--load-via-partition-root`，强制将数据加载到分区的根表，而非原分区（Rushabh Lathia）

  当加载目标系统具有不同的排序规则定义或字节序，可能需要将行存储到与以前不同的分区时，此选项很有用。

- 添加禁止转储和恢复数据库对象注释的选项（Robins Tharakan）

  pg_dump、pg_dumpall 和 pg_restore 的新选项为 `--no-comments`。

E.23.3.10. 源代码

- 为 PGXS 添加安装包含文件的支持（Andrew Gierth）

  这支持创建依赖其他模块的扩展模块。以前，依赖模块没有简单的方法找到被引用模块的包含文件。若干定义了数据类型的现有 `contrib` 模块已作调整，以安装相关文件。此外，PL/Perl 和 PL/Python 现在也会安装其包含文件，以支持创建这些语言的转换模块。

- 安装 `errcodes.txt`，使扩展可以访问 PostgreSQL 已知的错误码列表（Thomas Munro）

- 将文档转换为 DocBook XML（Peter Eisentraut，Alexander Lakhin，Jürgen Purtz）

  为与旧分支保持兼容，文件名仍使用 `sgml` 扩展名。

- 在适用的平台（即大多数平台）上，使用 `stdbool.h` 定义 `bool` 类型（Peter Eisentraut）

  这消除了需要包含 `stdbool.h` 的扩展模块的一项编码隐患。

- 全面调整初始系统目录内容的定义方式（John Naylor）

  初始数据现在用 Perl 数据结构表示，更便于机械化处理。

- 禁止扩展创建接受带引号值列表的自定义服务器参数（Tom Lane）

  目前无法支持这种功能，因为在加载扩展之前就必须知道该参数的这一属性。

- 增加在使用 SCRAM 认证时使用通道绑定的能力（Michael Paquier）

  通道绑定旨在防止中间人攻击，但除非能够强制启用通道绑定，否则 SCRAM 无法防止此类攻击。遗憾的是，libpq 中没有办法做到这一点。预计未来版本的 libpq，以及不基于 libpq 构建的接口（例如 JDBC）将支持此功能。

- 允许后台工作进程连接到通常不允许连接的数据库（Magnus Hagander）

- 添加对 ARMv8 上硬件 CRC 计算的支持（Yuqi Gu，Heikki Linnakangas，Thomas Munro）

- 加快按 OID 查找内置函数的速度（Andres Freund）

  以前的二分查找已被查找数组取代。

- 加快查询结果的构建速度（Andres Freund）

- 提高系统缓存访问速度（Andres Freund）

- 添加针对顺序分配/释放优化的分代内存分配器（Tomas Vondra）

  这减少了逻辑解码的内存用量。

- 使 `VACUUM` 对 `pg_class.reltuples` 的计算与 `ANALYZE` 一致（Tomas Vondra）

- 更新为使用 perltidy `20170521` 版（Tom Lane，Peter Eisentraut）

E.23.3.11. 附加模块

- 允许扩展 `pg_prewarm` 在启动时恢复先前的共享缓冲区内容（Mithun Cy，Robert Haas）

  实现方式是让 `pg_prewarm` 在服务器运行期间不时地、并在关闭时，将共享缓冲区的关系和块号数据存储到磁盘。

- 添加 `pg_trgm` 函数 `strict_word_similarity()`，以计算完整单词的相似度（Alexander Korotkov）

  已有的 `word_similarity()` 函数也用于此目的，但它的设计是寻找单词中相似的部分，而 `strict_word_similarity()` 计算的是与完整单词的相似度。

- 允许 `btree_gin` 对 `bool`、`bpchar`、`name` 和 `uuid` 数据类型建立索引（Matheus Oliveira）

- 允许 `cube` 和 `seg` 扩展使用 GiST 索引执行仅索引扫描（Andrey Borodin）

- 允许使用 `~>` 操作符获取 cube 坐标的负值（Alexander Korotkov）

  这对按降序查找坐标的 KNN-GiST 搜索很有用。

- 为 `unaccent` 扩展添加越南语字母处理（Dang Minh Huong，Michael Paquier）

- 增强 `amcheck`，以检查每个堆元组都有对应的索引条目（Peter Geoghegan）

- 让 `adminpack` 使用新的默认文件系统访问角色（Stephen Frost）

  以前，只有超级用户可以调用 `adminpack` 函数；现在改为检查角色权限。

- 将 `pg_stat_statement` 的查询 ID 扩为 64 位（Robert Haas）

  这大幅降低了查询 ID hash 冲突的概率。查询 ID 现在可能显示为负值。

- 移除已不再推荐使用的 `contrib/start-scripts/osx` 脚本（改用 `contrib/start-scripts/macos`）（Tom Lane）

- 移除 `chkpass` 扩展（Peter Eisentraut）

  此扩展不再被视为可用的安全工具或编写扩展的示例。

E.23.4. 致谢

以下人员（按字母顺序排列）作为补丁作者、提交者、审查者、测试者或问题报告者为本版本做出了贡献。

Abhijit Menon-Sen
Adam Bielanski
Adam Brightwell

Adam Brusselback
Aditya Toshniwal
Adrián Escoms
Adrien Nayrat
Akos Vandra
Aleksander Alekseev
Aleksandr Parfenov
Alexander Korotkov
Alexander Kukushkin
Alexander Kuzmenkov
Alexander Lakhin
Alexandre Garcia
Alexey Bashtanov
Alexey Chernyshov
Alexey Kryuchkov
Alik Khilazhev
Álvaro Herrera
Amit Kapila
Amit Khandekar
Amit Langote
Amul Sul
Anastasia Lubennikova
Andreas Joseph Krogh
Andreas Karlsson
Andreas Seltenreich
André Hänsel
Andrei Gorita
Andres Freund
Andrew Dunstan
Andrew Fletcher
Andrew Gierth
Andrew Grossman
Andrew Krasichkov
Andrey Borodin
Andrey Lizenko
Andy Abelisto
Anthony Bykov
Antoine Scemama
Anton Dignös
Antonin Houska
Arseniy Sharoglazov
Arseny Sher
Arthur Zakirov
Ashutosh Bapat
Ashutosh Sharma
Ashwin Agrawal
Asim Praveen
Atsushi Torikoshi
Badrul Chowdhury
Balazs Szilfai
Basil Bourque
Beena Emerson
Ben Chobot
Benjamin Coutu

Bernd Helmle
Blaz Merela
Brad DeJong
Brent Dearth
Brian Cloutier
Bruce Momjian
Catalin Iacob
Chad Trabant
Chapman Flack
Christian Duta
Christian Ullrich
Christoph Berg
Christoph Dreis
Christophe Courtois
Christopher Jones
Claudio Freire
Clayton Salem
Craig Ringer
Dagfinn Ilmari Mannsåker
Dan Vianello
Dan Watson
Dang Minh Huong
Daniel Gustafsson
Daniel Vérité
Daniel Westermann
Daniel Wood
Darafei Praliaskouski
Dave Cramer
Dave Page
David Binderman
David Carlier
David Fetter
David G. Johnston
David Gould
David Hinkle
David Pereiro Lagares
David Rader
David Rowley
David Steele
Davy Machado
Dean Rasheed
Dian Fay
Dilip Kumar
Dmitriy Sarafannikov
Dmitry Dolgov
Dmitry Ivanov
Dmitry Shalashov
Don Seiler
Doug Doole
Doug Rady
Edmund Horner
Eiji Seki
Elvis Pranskevichus
Emre Hasegeli

Erik Rijkers
Erwin Brandstetter
Etsuro Fujita
Euler Taveira
Everaldo Canuto
Fabien Coelho
Fabrízio de Royes Mello
Feike Steenbergen
Frits Jalvingh
Fujii Masao
Gao Zengqi
Gianni Ciolli
Greg Stark
Gunnlaugur Thor Briem
Guo Xiang Tan
Hadi Moshayedi
Hailong Li
Haribabu Kommi
Heath Lord
Heikki Linnakangas
Hugo Mercier
Igor Korot
Igor Neyman
Ildar Musin
Ildus Kurbangaliev
Ioseph Kim
Jacob Champion
Jaime Casanova
Jakob Egger
Jean-Pierre Pelletier
Jeevan Chalke
Jeevan Ladhe
Jeff Davis
Jeff Janes
Jeremy Evans
Jeremy Finzel
Jeremy Schneider
Jesper Pedersen
Jim Nasby
Jimmy Yih
Jing Wang
Jobin Augustine
Joe Conway
John Gorman
John Naylor
Jon Nelson
Jon Wolski
Jonathan Allen
Jonathan S. Katz
Julien Rouhaud
Jürgen Purtz
Justin Pryzby
KaiGai Kohei
Kaiting Chen

Karl Lehenbauer
Keith Fiske
Kevin Bloch
Kha Nguyen
Kim Rose Carlsen
Konstantin Knizhnik
Kuntal Ghosh
Kyle Samson
Kyotaro Horiguchi
Lætitia Avrot
Lars Kanis
Laurenz Albe
Leonardo Cecchi
Liudmila Mantrova
Lixian Zou
Lloyd Albin
Luca Ferrari
Lucas Fairchild
Lukas Eder
Lukas Fittl
Magnus Hagander
Mai Peng
Maksim Milyutin
Maksym Boguk
Mansur Galiev
Marc Dilger
Marco Nenciarini
Marina Polyakova
Mario de Frutos Dieguez
Mark Cave-Ayland
Mark Dilger
Mark Wood
Marko Tiikkaja
Markus Winand
Martín Marqués
Masahiko Sawada
Matheus Oliveira
Matthew Stickney
Metin Doslu
Michael Banck
Michael Meskes
Michael Paquier
Michail Nikolaev
Mike Blackwell
Minh-Quan Tran
Mithun Cy
Morgan Owens
Nathan Bossart
Nathan Wagner
Neil Conway
Nick Barnes
Nicolas Thauvin
Nikhil Sontakke
Nikita Glukhov

Nikolay Shaplov
Noah Misch
Noriyoshi Shinoda
Oleg Bartunov
Oleg Samoilov
Oliver Ford
Pan Bian
Pascal Legrand
Patrick Hemmer
Patrick Krecker
Paul Bonaud
Paul Guo
Paul Ramsey
Pavan Deolasee
Pavan Maddamsetti
Pavel Golub
Pavel Stehule
Peter Eisentraut
Peter Geoghegan
Petr Jelínek
Petru-Florin Mihancea
Phil Florent
Philippe Beaudoin
Pierre Ducroquet
Piotr Stefaniak
Prabhat Sahu
Pu Qun
QL Zhuo
Rafia Sabih
Rahila Syed
Rainer Orth
Rajkumar Raghuwanshi
Raúl Marín Rodríguez
Regina Obe
Richard Yen
Robert Haas
Robins Tharakan
Rod Taylor
Rushabh Lathia
Ryan Murphy
Sahap Asci
Samuel Horwitz
Scott Ure
Sean Johnston
Shao Bret
Shay Rojansky
Shubham Barai
Simon Riggs
Simone Gotti
Sivasubramanian Ramasubramanian
Stas Kelvich
Stefan Kaltenbrunner
Stephen Froehlich
Stephen Frost

Steve Singer
Steven Winfield
Sven Kunze
Taiki Kondo
Takayuki Tsunakawa
Takeshi Ideriha
Tatsuo Ishii
Tatsuro Yamada
Teodor Sigaev
Thom Brown
Thomas Kellerer
Thomas Munro
Thomas Reiss
Tobias Bussmann
Todd A. Cook
Tom Kazimiers
Tom Lane
Tomas Vondra
Tomonari Katsumata
Torsten Grust
Tushar Ahuja
Vaishnavi Prabakaran
Vasundhar Boddapati
Victor Drobny
Victor Wagner
Victor Yegorov
Vik Fearing
Vinayak Pokale
Vincent Lachenal
Vitaliy Garnashevich
Vitaly Burovoy
Vladimir Baranoff
Xin Zhang
Yi Wen Wong
Yorick Peterse
Yugo Nagata
Yuqi Gu
Yura Sokolov
Yves Goergen
Zhou Digoal

E.24. 先前版本

先前各发布分支的发布说明可在 `https://www.postgresql.org/docs/release/`[3]找到。

[3] https://www.postgresql.org/docs/release/

附录 F. 额外提供的模块

本附录与下一附录包含有关 PostgreSQL 发行版中 `contrib` 目录中可以找到的模块的信息。这些模块包括移植工具、分析实用程序，以及不属于 PostgreSQL 核心系统的插件功能，主要是因为它们面向的受众有限，或者实验性太强，不适合作为主源码树的一部分。但这并不影响它们的实用性。

本附录介绍位于 `contrib` 中的扩展以及其他服务器插件模块。附录 G 介绍实用程序。

从源码发行版构建时，除非构建“world”目标（见步骤 2），否则这些组件不会自动构建。可以运行以下命令来构建并安装所有组件：

```
make
make install
```

这些命令应在已配置好的源码树的 `contrib` 目录中运行；若只想构建并安装某个选定的模块，则可在该模块的子目录中执行同样的命令。许多模块都带有回归测试，可以运行以下命令来执行：

```
make check
```

这是在安装前执行的命令；也可以运行

```
make installcheck
```

此时需要已有一个 PostgreSQL 服务器正在运行。

如果你使用的是预打包版本的 PostgreSQL，这些模块通常会作为单独的子包提供，例如 `postgresql-contrib`。

许多模块提供新的用户定义函数、操作符或类型。安装代码后，要使用其中某个模块，就需要在数据库系统中注册新的 SQL 对象。在PostgreSQL 9.1 及后续版本中，这可通过执行CREATE EXTENSION 命令完成。在一个新建的数据库中，你可以直接执行：

```
CREATE EXTENSION module_name;
```

该命令必须由数据库超级用户执行。它只会在当前数据库中注册新的 SQL 对象，因此需要在每个希望使用该模块功能的数据库中运行它。另一种做法是在数据库`template1` 中运行它，这样该扩展默认会被复制到随后创建的数据库中。

许多模块允许你将其对象安装到所选的模式中。要这样做，请在`CREATE EXTENSION` 命令中加入 `SCHEMA schema_name`。默认情况下，这些对象会被放入当前的创建目标模式，而该模式默认是`public`。

如果你的数据库是从 PostgreSQL 9.1 之前的版本通过转储和重新载入升级而来的，并且其中一直在使用该模块的 9.1 之前版本，则应改为执行：

```
CREATE EXTENSION module_name FROM unpackaged;
```

这会将该模块在 9.1 之前版本中的对象更新为一个正规的扩展对象。今后该模块的更新将由 ALTER EXTENSION 管理。有关扩展更新的更多信息，请参见第 38.16 节。

但请注意，其中有些模块并不是这种意义上的“扩展”，而是通过其他方式加载到服务器中，例如借助 shared_preload_libraries。详情见各模块文档。

F.1. adminpack

adminpack 提供了一些支持函数，pgAdmin 和其他管理与维护工具可使用这些函数提供附加功能，例如远程管理服务器日志文件。默认情况下，只有超级用户才允许使用所有这些函数，但也可以使用GRANT 命令允许其他用户使用。

表 F.1 中显示的函数提供对承载服务器的机器上文件的写访问。（另见表 9.88 中的函数，它们提供只读访问。）除非用户是超级用户，或者根据函数的不同被授予 pg_read_server_files 或 pg_write_server_files 角色之一，否则只能访问数据库集簇目录内的文件，但可以使用相对路径或绝对路径。

表 F.1. adminpack 函数

名称	返回类型	描述
pg_catalog.pg_file_write(filename text, data text, append boolean)	bigint	写入文本文件或向其追加内容
pg_catalog.pg_file_ rename(oldname text, newname text [, archiv ename text])	boolean	重命名文件
pg_catalog.pg_file_ unlink(filename text)	boolean	删除文件
pg_catalog.pg_logdir_ls()	setof record	列出log_directory 目录中的日志文件

pg_file_write 将指定的 *data* 写入由 *filename* 指定的文件中。如果 *append* 为 false，则该文件必须尚不存在。如果*append* 为 true，则该文件可以已经存在，并且若已存在则会向其追加内容。返回写入的字节数。

pg_file_rename 重命名文件。如果省略 *archivename* 或其值为空值，则它只是将*oldname* 重命名为 *newname*（后者必须尚不存在）。如果提供了 *archivename*，它会先将*newname* 重命名为 *archivename*（后者必须尚不存在），然后再将*oldname* 重命名为 *newname*。如果第二个重命名步骤失败，它会在报告错误之前尝试将*archivename* 再改回 *newname*。成功时返回 true；如果源文件不存在或不可写，则返回 false；其他情况会抛出错误。

pg_file_unlink 删除指定的文件。成功时返回 true；如果指定的文件不存在，或者 unlink() 调用失败，则返回 false；其他情况会抛出错误。

pg_logdir_ls 返回 log_directory 目录中所有日志文件的起始时间戳和路径名。要使用此函数，log_filename 参数必须保持默认设置（postgresql-%Y-%m-%d_%H%M%S.log）。

F.2. amcheck

amcheck 模块提供了一组函数，可用于验证关系结构的逻辑一致性。如果结构看起来有效，就不会引发错误。

这些函数会验证特定关系表示结构中的多种不变式。索引扫描以及其他重要操作背后的访问方法函数是否正确，依赖于这些不变式始终成立。例如，某些函数除其他事项外，还会验证所有 B-树页面中的项都按“逻辑”顺序排列（例如，对于text 上的 B-树索引，索引元组应当按排序规则定义的词法顺序排列）。如果这种特定不变式由于某种原因不再成立，那么受影响页面上的二分查找就可能错误地引导索引扫描，从而使 SQL 查询返回错误答案。

验证工作使用的过程与索引扫描本身所使用的过程相同，而这些过程可能是用户定义的操作符类代码。例如，B-树索引验证依赖于一个或多个 B-树支持函数 1 例程执行的比较。关于操作符类支持函数的细节，见第 38.15.3 节。

只有超级用户才能使用 amcheck 函数。

F.2.1. 函数

```
bt_index_check(index regclass, heapallindexed boolean) returns void
```

bt_index_check 测试其目标 B-树索引是否满足多种不变式。示例用法如下：

```
test=# SELECT bt_index_check(index => c.oid, heapallindexed => i.indisunique),
              c.relname,
              c.relpages
FROM pg_index i
JOIN pg_opclass op ON i.indclass[0] = op.oid
JOIN pg_am am ON op.opcmethod = am.oid
JOIN pg_class c ON i.indexrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE am.amname = 'btree' AND n.nspname = 'pg_catalog'
-- Don't check temp tables, which may be from another session:
AND c.relpersistence != 't'
-- Function may throw an error when this is omitted:
AND c.relkind = 'i' AND i.indisready AND i.indisvalid
ORDER BY c.relpages DESC LIMIT 10;
 bt_index_check |             relname             | relpages
----------------+---------------------------------+----------
                | pg_depend_reference_index       |       43
                | pg_depend_depender_index        |       40
                | pg_proc_proname_args_nsp_index  |       31
                | pg_description_o_c_o_index      |       21
                | pg_attribute_relid_attnam_index |       14
                | pg_proc_oid_index               |       10
                | pg_attribute_relid_attnum_index |        9
                | pg_amproc_fam_proc_index        |        5
                | pg_amop_opr_fam_index           |        5
                | pg_amop_fam_strat_index         |        5
(10 rows)
```

本例展示了一个会话，它对数据库“test”中最大的 10 个系统目录索引执行验证。对其中属于唯一索引的那一部分，还要求验证所有堆元组在索引中都有对应的索引元组。由于没有引发错误，因此所有受测索引看起来都具有逻辑一致性。当然，也很容易把这个查询改成对数据库中每一个支持验证的索引调用 bt_index_check。

bt_index_check 会在目标索引及其所属的堆关系上获取 AccessShareLock。这种锁模式与简单 SELECT 语句在关系上获取的锁模式相同。bt_index_check 不会验证跨越父子

关系的不变式，但如果*heapallindexed* 为 true，它会验证所有堆元组在索引中都有对应的索引元组。当在线生产环境中需要一种例行的、轻量级的损坏检查时，bt_index_check 通常能在验证彻底程度与对应用性能、可用性的影响之间提供最佳权衡。

bt_index_parent_check(index regclass, heapallindexed boolean) returns void

bt_index_parent_check 测试其目标 B-树索引是否满足多种不变式。如果可选参数 *heapallindexed* 为 true，该函数还会验证索引中是否包含所有本应出现的堆元组，并检查索引结构中不存在缺失的下行链接。bt_index_parent_check 能够执行的检查，是bt_index_check 所能执行检查的超集。可以把 bt_index_parent_check 看作 bt_index_check 更彻底的变体：与 bt_index_check 不同，bt_index_parent_check 还会检查跨越父子关系的不变式。如果发现逻辑不一致或其他问题，bt_index_parent_check 就会按惯例引发错误。

bt_index_parent_check 要求在目标索引上持有 ShareLock（并且在堆关系上也会获取 ShareLock）。这些锁会阻止 INSERT、UPDATE 以及 DELETE 命令并发修改数据。这些锁还会阻止底层关系被并发 VACUUM 处理，以及执行所有其他实用命令。注意，该函数只在运行期间持有这些锁，而不是在整个事务期间持有。

bt_index_parent_check 所做的额外验证，更有可能检测出各种异常情形。这些情形可能涉及被检查索引所使用的 B-树操作符类实现错误，或者假设存在的、底层 B-树索引访问方法代码中尚未发现的缺陷。请注意，与bt_index_check 不同，在启用热备模式时（即在只读物理副本上），不能使用 bt_index_parent_check。

F.2.2. 可选的 *heapallindexed* 验证

当验证函数的 *heapallindexed* 参数为 true 时，会针对与目标索引关系相关联的表执行一个额外的验证阶段。这一阶段由一次“伪” CREATE INDEX 操作构成，它会借助一个临时的、位于内存中的汇总结构，检查所有假想的新索引元组是否存在。这个汇总结构会在基本验证第一阶段中按需构建，并为目标索引中找到的每个元组生成“指纹”。*heapallindexed* 验证背后的总体原则是：一个与现有目标索引等价的新索引，只能包含那些可以在现有结构中找到的项。

额外的 *heapallindexed* 阶段会带来显著开销：验证通常需要耗费数倍于平常的时间。不过，执行*heapallindexed* 验证时，所获取的关系级锁并不会发生变化。

这个汇总结构的大小受 maintenance_work_mem 限制。为了确保对每个本应在索引中有表示的堆元组而言，漏检一处不一致的概率不超过 2%，每个元组大约需要 2 字节内存。为每个元组提供的内存越少，漏掉不一致的概率就会缓慢上升。这种方法显著限制了验证开销，同时只会轻微降低发现问题的概率，对那些将验证视为例行维护任务的部署环境尤其如此。每次重新执行验证时，任何单个缺失或格式错误的元组都会再次获得被检测到的机会。

F.2.3. 有效使用 amcheck

amcheck 能够有效检测出多种数据页校验和始终无法捕捉的故障模式，包括：

- 由操作符类实现错误导致的结构不一致。

  这也包括因操作系统排序规则的比较规则发生变化而引起的问题。像 text 这类支持排序规则的类型的 datum 之间的比较必须是不可变的（正如用于 B-树索引扫描的所有比较都必须不可变一样），这就意味着操作系统排序规则绝不能发生变化。虽然这种情况比较少见，但操作系统排序规则的更新确实可能导致此类问题。更常见的是主库与备库之间的排序顺序不一致，这可能是因为两边使用的操作系统主版本不同。这类不一致通常只会出现在备库上，因此通常也只能在备库上检测到。

  如果出现此类问题，它未必会影响每一个按受影响排序规则排序的索引，因为被索引的值也可能恰好在行为不一致的情况下仍具有相同的绝对顺序。关于 PostgreSQL 如何使用操作系统区域设置和排序规则的更多细节，见第 23.1 节和第 23.2 节。

- 索引与其所索引的堆关系之间的结构不一致（在执行 *heapallindexed* 验证时）。

  在正常运行期间，索引并不会与其堆关系进行交叉核对。堆损坏的症状可能十分隐蔽。

- 假设存在的、底层PostgreSQL 访问方法代码、排序代码或事务管理代码中尚未发现的缺陷所导致的损坏。

  对索引结构完整性的自动验证，在对新的或拟议中的 PostgreSQL 特性进行一般性测试时具有作用，因为这些特性完全可能引入逻辑不一致。对表结构以及相关可见性、事务状态信息的验证，也起着类似作用。一种显而易见的测试策略，是在运行标准回归测试时持续调用 amcheck 函数。关于如何运行测试，见第 33.1 节。

- 在未启用校验和时发生的文件系统或存储子系统故障。

  请注意，如果访问某个块时只是命中了共享缓冲区，那么amcheck 检查的是验证时该页面在某个共享内存缓冲区中的表示。因此，amcheck 并不一定会检查验证时从文件系统读入的数据。另请注意，当启用了校验和时，如果某个损坏块被读入缓冲区，amcheck 可能会因为校验和失败而引发错误。

- 由有缺陷的 RAM 或更广义的内存子系统导致的损坏。

  PostgreSQL 并不防御可纠正的内存错误，并且假定所用 RAM 采用业界标准的纠错码（ECC）或更强的保护机制。然而，ECC 内存通常只对单比特错误有效，不应被视为能对导致内存损坏的故障提供绝对保护。

  执行 *heapallindexed* 验证时，由于会测试严格的二进制相等性，并且还会检查堆中的被索引属性，因而检测出单比特错误的机会通常会大大增加。

一般来说，amcheck 只能证明损坏存在，无法证明损坏不存在。

F.2.4. 修复损坏

amcheck 报告的与损坏有关的错误都不应是误报。amcheck 会在那些按定义绝不应该发生的情况下抛出错误，因此通常需要对amcheck 错误进行仔细分析。

对于 amcheck 检测到的问题，并不存在通用的修复方法。应当查明导致不变式遭到破坏的根本原因。在诊断amcheck 检测到的损坏时，pageinspect 可能会发挥有用作用。REINDEX 未必能够有效修复损坏。

F.3. auth_delay

auth_delay 会使服务器在报告认证失败之前短暂暂停，从而增加对数据库密码进行暴力破解攻击的难度。注意，它并不能阻止拒绝服务攻击，甚至可能加剧此类攻击，因为在报告认证失败前处于等待状态的进程仍会占用连接槽位。

为了使该模块正常工作，必须在postgresql.conf 中通过 shared_preload_libraries 加载它。

F.3.1. 配置参数

auth_delay.milliseconds (int)

在报告认证失败之前等待的毫秒数。默认值为 0。

这些参数必须在 postgresql.conf 中设置。典型的用法如下：

```
# postgresql.conf
shared_preload_libraries = 'auth_delay'

auth_delay.milliseconds = '500'
```

F.3.2. 作者

KaiGai Kohei <kaigai@ak.jp.nec.com>

F.4. auto_explain

auto_explain 模块提供了一种自动记录慢语句执行计划的方法，而无须手工运行 EXPLAIN。这对于在大型应用中追查未优化的查询尤其有帮助。

该模块不提供可通过 SQL 访问的函数。要使用它，只需将其加载到服务器中。可以将其加载到单个会话中：

```
LOAD 'auto_explain';
```

（要这样做，必须是超级用户。）更典型的用法是，在postgresql.conf 中的 session_preload_libraries 或 shared_preload_libraries 里包含 auto_explain，从而将其预加载到部分或全部会话中。这样，无论查询何时意外变慢，都可以追踪到它们。当然，这要付出额外开销。

F.4.1. 配置参数

有若干配置参数可控制 auto_explain 的行为。注意，其默认行为是什么都不做，因此如果希望得到任何结果，至少必须设置auto_explain.log_min_duration。

auto_explain.log_min_duration (integer)

auto_explain.log_min_duration 是会导致记录该语句计划的最小语句执行时间，单位为毫秒。将其设为 0 会记录所有计划。-1（默认值）会禁用计划记录。例如，如果将其设为250ms，则所有运行 250ms 或更久的语句都会被记录。只有超级用户可以更改此设置。

auto_explain.log_analyze (boolean)

auto_explain.log_analyze 使得在记录执行计划时输出 EXPLAIN ANALYZE，而不只是输出EXPLAIN。该参数默认关闭。只有超级用户可以更改此设置。

> **注意**
>
> 当该参数开启时，所有已执行语句都会对每个计划节点计时，无论它们是否实际运行得足够久而最终被记录。这可能对性能造成极其负面的影响。关闭auto_explain.log_timing 可以缓解性能开销，但代价是获得的信息会更少。

auto_explain.log_buffers (boolean)

auto_explain.log_buffers 控制在记录执行计划时是否打印缓冲区使用统计信息；它等效于 EXPLAIN 的 BUFFERS 选项。除非启用 auto_explain.log_analyze，否则该参数没有效果。该参数默认关闭。只有超级用户可以更改此设置。

auto_explain.log_timing (boolean)

auto_explain.log_timing 控制在记录执行计划时是否打印每个计划节点的计时信息；它等效于 EXPLAIN 的 TIMING 选项。在某些系统上，反复读取系统时钟的开销可能会显著拖慢查询，因此如果只需要实际行计数而不需要精确时间，将该参数设为 off 可能会有用。除非启用auto_explain.log_analyze，否则该参数没有效果。该参数默认开启。只有超级用户可以更改此设置。

auto_explain.log_triggers (boolean)

auto_explain.log_triggers 使得在记录执行计划时包含触发器执行统计信息。除非启用 auto_explain.log_analyze，否则该参数没有效果。该参数默认关闭。只有超级用户可以更改此设置。

auto_explain.log_verbose (boolean)

auto_explain.log_verbose 控制在记录执行计划时是否打印详细信息；它等效于 EXPLAIN 的 VERBOSE 选项。该参数默认关闭。只有超级用户可以更改此设置。

auto_explain.log_format (enum)

auto_explain.log_format 选择要使用的 EXPLAIN 输出格式。允许的值有 text、xml、json 和 yaml。默认值为 text。只有超级用户可以更改此设置。

auto_explain.log_nested_statements (boolean)

auto_explain.log_nested_statements 使嵌套语句（在函数内部执行的语句）也会被纳入记录考虑范围。关闭该参数时，只记录顶层查询计划。该参数默认关闭。只有超级用户可以更改此设置。

auto_explain.sample_rate (real)

auto_explain.sample_rate 使 auto_explain 在每个会话中只解释一部分语句。默认值为 1，表示解释所有查询。对于嵌套语句，要么全部解释，要么全部不解释。只有超级用户可以更改此设置。

在通常用法中，这些参数都在postgresql.conf 中设置，尽管超级用户也可以在自己的会话中动态修改它们。典型的用法可能是：

```
# postgresql.conf
session_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '3s'
```

F.4.2. 示例

```
postgres=# LOAD 'auto_explain';
postgres=# SET auto_explain.log_min_duration = 0;
postgres=# SET auto_explain.log_analyze = true;
postgres=# SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
```

这可能产生如下日志输出：

```
LOG:  duration: 3.651 ms  plan:
  Query Text: SELECT count(*)
              FROM pg_class, pg_index
              WHERE oid = indrelid AND indisunique;
  Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual time=3.626..3.627 rows=1 loops=1)
    ->  Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual time=3.349..3.594 rows=92 loops=
1)
          Hash Cond: (pg_class.oid = pg_index.indrelid)
          ->  Seq Scan on pg_class  (cost=0.00..9.55 rows=255 width=4) (actual time=0.016..0.
140 rows=255 loops=1)
          ->  Hash  (cost=3.02..3.02 rows=92 width=4) (actual time=3.238..3.238 rows=92 loops=
1)
                Buckets: 1024  Batches: 1  Memory Usage: 4kB
                ->  Seq Scan on pg_index  (cost=0.00..3.02 rows=92 width=4) (actual time=0.008.
.3.187 rows=92 loops=1)
                      Filter: indisunique
```

F.4.3. 作者

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>

F.5. bloom

`bloom` 提供了一种基于布隆过滤器[1]的索引访问方法。

布隆过滤器是一种空间效率很高的数据结构，用于测试某个元素是否属于某个集合。对于索引访问方法，它允许通过签名快速排除不匹配的元组，而签名的大小则在创建索引时确定。

签名是被索引属性的一种有损表示，因此容易产生误报；也就是说，某个元素实际上并不在集合中，却可能被报告为在集合中。因此，索引搜索结果始终都必须使用堆条目中的实际属性值再次检查。更长的签名可以降低误报概率，从而减少无用的堆访问次数，但当然也会使索引变得更大，因此扫描速度更慢。

当一张表具有很多属性，而查询又会测试这些属性的任意组合时，这类索引最有用。传统 B-树索引比布隆索引更快，但为了支持所有可能的查询，可能需要许多 B-树索引，而布隆索引只需一个。不过要注意，布隆索引只支持等值查询，而 B-树索引还可以执行不等和范围搜索。

F.5.1. 参数

`bloom` 索引在其 `WITH` 子句中接受下列参数：

`length`

每个签名（索引项）的长度，以位为单位。该值会向上圆整为`16` 的倍数。默认值为 `80` 位，最大值为`4096`。

`col1 — col32`

为每个索引列生成的位数。每个参数的名字都对应它所控制的索引列编号。默认值为 `2` 位，最大值为`4095`。对于实际未使用的索引列，其参数会被忽略。

F.5.2. 示例

下面是一个创建布隆索引的示例：

```
CREATE INDEX bloomidx ON tbloom USING bloom (i1,i2,i3)
       WITH (length=80, col1=2, col2=2, col3=4);
```

该索引使用长度为 80 位的签名创建，其中属性 i1 和 i2 各映射为 2 位，属性 i3 映射为 4 位。由于`length`、`col1` 和 `col2` 都采用默认值，因此这些参数的指定也可以省略。

下面给出一个更完整的布隆索引定义和使用示例，同时与等效的 B-树索引进行比较。布隆索引明显比 B-树索引更小，而且性能可能更好。

[1] https://en.wikipedia.org/wiki/Bloom_filter

```
=# CREATE TABLE tbloom AS
    SELECT
      (random() * 1000000)::int as i1,
      (random() * 1000000)::int as i2,
      (random() * 1000000)::int as i3,
      (random() * 1000000)::int as i4,
      (random() * 1000000)::int as i5,
      (random() * 1000000)::int as i6
    FROM
  generate_series(1,10000000);
SELECT 10000000
```

对这个大表进行顺序扫描需要很长时间：

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                  QUERY PLAN

-------------------------------------------------------------------------------------------------------
-------
 Seq Scan on tbloom  (cost=0.00..2137.14 rows=3 width=24) (actual time=18.372..18.373 rows=0
 loops=1)
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 100000
 Planning Time: 0.400 ms
 Execution Time: 18.397 ms
(5 rows)
```

即使定义了 B-树索引，结果仍然是顺序扫描：

```
=# CREATE INDEX btreeidx ON tbloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('btreeidx'));
 pg_size_pretty
----------------
 3992 kB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                  QUERY PLAN

-------------------------------------------------------------------------------------------------------
-------
 Seq Scan on tbloom  (cost=0.00..2137.00 rows=2 width=24) (actual time=11.880..11.881 rows=0
 loops=1)
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 100000
 Planning Time: 0.154 ms
 Execution Time: 11.896 ms
(5 rows)
```

在表上定义 bloom 索引后，处理此类搜索的效果比 B-树更好：

```
=# CREATE INDEX bloomidx ON tbloom USING bloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('bloomidx'));
 pg_size_pretty
----------------
 1584 kB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                      QUERY PLAN

------------------------------------------------------------------------------------------------------
----------------------
 Bitmap Heap Scan on tbloom  (cost=1792.00..1799.69 rows=2 width=24) (actual time=0.388..0.388
 rows=0 loops=1)
   Recheck Cond: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Index Recheck: 25
   Heap Blocks: exact=22
   ->  Bitmap Index Scan on bloomidx  (cost=0.00..1792.00 rows=2 width=0) (actual time=0.358..
0.358 rows=25 loops=1)
         Index Cond: ((i2 = 898732) AND (i5 = 123451))
 Planning Time: 0.118 ms
 Execution Time: 0.412 ms
(8 rows)
```

现在，B-树搜索的主要问题在于，当搜索条件没有约束索引的前导列时，B-树的效率很低。对 B-树而言，更好的策略是在每一列上创建单独的索引。这样，规划器会选择类似下面的计划：

```
=# CREATE INDEX btreeidx1 ON tbloom (i1);
CREATE INDEX
=# CREATE INDEX btreeidx2 ON tbloom (i2);
CREATE INDEX
=# CREATE INDEX btreeidx3 ON tbloom (i3);
CREATE INDEX
=# CREATE INDEX btreeidx4 ON tbloom (i4);
CREATE INDEX
=# CREATE INDEX btreeidx5 ON tbloom (i5);
CREATE INDEX
=# CREATE INDEX btreeidx6 ON tbloom (i6);
CREATE INDEX
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                          QUERY PLAN

------------------------------------------------------------------------------------------------------
----------------------------
 Bitmap Heap Scan on tbloom  (cost=24.34..32.03 rows=2 width=24) (actual time=0.036..0.037
 rows=0 loops=1)
   Recheck Cond: ((i5 = 123451) AND (i2 = 898732))
   ->  BitmapAnd  (cost=24.34..24.34 rows=2 width=0) (actual time=0.033..0.034 rows=0 loops=1)
         ->  Bitmap Index Scan on btreeidx5  (cost=0.00..12.04 rows=500 width=0) (actual time=
0.032..0.032 rows=0 loops=1)
               Index Cond: (i5 = 123451)
         ->  Bitmap Index Scan on btreeidx2  (cost=0.00..12.04 rows=500 width=0) (never
 executed)
               Index Cond: (i2 = 898732)
 Planning Time: 0.531 ms
 Execution Time: 0.072 ms
(9 rows)
```

虽然这个查询比使用任意一个单独索引时都快得多，但代价是索引更大。每个单列 B-树索引占用 2 MB，因此总共需要 12 MB，是 bloom 索引所用空间的八倍。

F.5.3. 操作符类接口

用于布隆索引的操作符类只需要一个针对被索引数据类型的 hash 函数，以及一个用于搜索的等值操作符。下面这个示例展示了 `text` 数据类型的操作符类定义：

```
CREATE OPERATOR CLASS text_ops
DEFAULT FOR TYPE text USING bloom AS
    OPERATOR    1   =(text, text),
    FUNCTION    1   hashtext(text);
```

F.5.4. 限制

- 模块中只包含了 `int4` 和 `text` 的操作符类。
- 搜索只支持=操作符。不过将来有可能增加对带有并集和交集操作的数组的支持。
- `bloom` 访问方法不支持 `UNIQUE` 索引。
- `bloom` 访问方法不支持搜索 `NULL` 值。

F.5.5. 作者

Teodor Sigaev <teodor@postgrespro.ru>，Postgres Professional，俄罗斯莫斯科

Alexander Korotkov <a.korotkov@postgrespro.ru>，Postgres Professional，俄罗斯莫斯科

Oleg Bartunov <obartunov@postgrespro.ru>，Postgres Professional，俄罗斯莫斯科

F.6. btree_gin

btree_gin 提供了示例 GIN 操作符类，为以下数据类型实现 B-树等价行为：int2、int4、int8、float4、float8、timestamp with time zone、timestamp without time zone、time with time zone、time without time zone、date、interval、oid、money、"char"、varchar、text、bytea、bit、varbit、macaddr、macaddr8、inet、cidr、uuid、name、bool、bpchar，以及所有enum 类型。

一般来说，这些操作符类的性能不会优于等价的标准 B-树索引方法，而且它们缺少标准 B-树代码的一项主要特性：强制唯一性的能力。不过，它们对于 GIN 测试以及作为开发其他 GIN 操作符类的基础很有用。此外，对于同时测试一个可由 GIN 建索引的列和一个可由 B-树建索引的列的查询，创建一个使用这些操作符类之一的多列 GIN 索引，可能比创建两个必须通过位图 AND 运算组合的独立索引更高效。

F.6.1. 用法示例

```
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING GIN (a);
-- query
SELECT * FROM test WHERE a < 10;
```

F.6.2. 作者

Teodor Sigaev（<teodor@stack.net>）和 Oleg Bartunov（<oleg@sai.msu.su>）。参阅 http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin。

F.7. btree_gist

btree_gist 提供了 GiST 索引操作符类，可为以下数据类型实现与 B-树等价的行为：int2、int4、int8、float4、float8、numeric、timestamp with time zone、timestamp without time zone、time with time zone、time without time zone、date、interval、oid、money、char、varchar、text、bytea、bit、varbit、macaddr、macaddr8、inet、cidr、uuid，以及所有enum 类型。

一般来说，这些操作符类的性能不会优于对应的标准 B-树索引方法，而且它们缺少标准 B-树实现的一项主要特性：强制唯一性的能力。不过，正如下文所述，它们提供了一些 B-树索引所不具备的其他特性。另外，当需要多列 GiST 索引，而其中某些列的数据类型只能用 GiST 建立索引、其他列只是简单数据类型时，这些操作符类就很有用。最后，这些操作符类对于 GiST 测试以及作为开发其他 GiST 操作符类的基础也很有用。

除典型的 B-树搜索操作符之外，btree_gist 还为<>（“不等于”）提供索引支持。这在与下文描述的排他约束结合使用时可能很有用。

此外，对于那些具有自然距离度量的数据类型，btree_gist 定义了距离操作符<->，并为使用该操作符的最近邻搜索提供 GiST 索引支持。为以下类型提供了距离操作符：int2、int4、int8、float4、float8、timestamp with time zone、timestamp without time zone、time without time zone、date、interval、oid 和 money。

F.7.1. 用法示例

一个使用 btree_gist 代替 btree 的简单示例：

```
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING GIST (a);
-- query
SELECT * FROM test WHERE a < 10;
-- nearest-neighbor search: find the ten entries closest to "42"
SELECT *, a <-> 42 AS dist FROM test ORDER BY a <-> 42 LIMIT 10;
```

使用排他约束来强制执行这样一条规则：动物园中的一个笼子只能容纳一种动物：

```
=> CREATE TABLE zoo (
  cage   INTEGER,
  animal TEXT,
  EXCLUDE USING GIST (cage WITH =, animal WITH <>)
);

=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'lion');
ERROR:  conflicting key value violates exclusion constraint "zoo_cage_animal_excl"
DETAIL:  Key (cage, animal)=(123, lion) conflicts with existing key (cage, animal)=(123, zebra)
.
=> INSERT INTO zoo VALUES(124, 'lion');
INSERT 0 1
```

F.7.2. 作者

Teodor Sigaev（<teodor@stack.net>）、Oleg Bartunov（<oleg@sai.msu.su>）、Janko Richter（<jankorichter@yahoo.de>）和 Paul

Jungwirth（<pj@illuminatedcomputing.com>）。更多信息见 http://www.sai.msu.su/~megera/postgres/gist/。

F.8. citext

citext 模块提供一种大小写不敏感的字符串类型 citext。本质上，它在比较值时会在内部调用 lower。除此之外，它的行为几乎与 text 完全相同。

F.8.1. 原理

在 PostgreSQL 中进行大小写不敏感匹配的标准方法，一直是在比较值时使用lower 函数，例如：

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

这种做法效果尚可，但有几个缺点：

- 它会让 SQL 语句变得冗长，而且你必须时刻记得同时对列和查询值调用 lower。
- 除非你创建一个使用 lower 的函数索引，否则它不会使用索引。
- 如果把列声明为 UNIQUE 或 PRIMARY KEY，隐式生成的索引仍然是大小写敏感的。因此，它既无法用于大小写不敏感搜索，也不能以大小写不敏感的方式强制唯一性。

citext 数据类型允许你在 SQL 查询中省去对 lower 的调用，并允许主键不区分大小写。与 text 一样，citext 也与区域设置相关，这意味着大写字符和小写字符如何匹配取决于数据库 LC_CTYPE 设置的规则。同样，这种行为与在查询中使用 lower 完全一致。但由于这是由数据类型透明地完成的，因此你不必在查询中额外记住任何特殊处理。

F.8.2. 如何使用

下面是一个简单的使用示例：

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT   NOT NULL
);

INSERT INTO users VALUES ( 'larry',  sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Tom',    sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Damian', sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'NEAL',   sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Bjørn',  sha256(random()::text::bytea) );

SELECT * FROM users WHERE nick = 'Larry';
```

这个 SELECT 语句仍会返回一个元组，尽管nick 列中存的是 larry，而查询条件写的是 Larry。

F.8.3. 字符串比较行为

citext 在比较时会先把每个字符串转换为小写（就像调用了 lower 一样），然后再按常规方式比较结果。因此，举例来说，如果两个字符串经过lower 后得到的结果相同，它们就会被视为相等。

为了尽可能贴近大小写不敏感排序规则的行为，一些字符串处理操作符和函数都提供了 citext 专用版本。例如，当应用于citext 时，正则表达式操作符~和~* 表现相同：它们都会以大小写

不敏感的方式匹配。!~和 !~* 也是如此，LIKE 操作符 ~~和~~*、以及 !~~和!~~* 也一样。如果你希望进行大小写敏感匹配，可以把这些操作符的参数转换为text。

同样地，如果这些函数的参数是citext，它们也会以大小写不敏感方式进行匹配：

- regexp_match()
- regexp_matches()
- regexp_replace()
- regexp_split_to_array()
- regexp_split_to_table()
- replace()
- split_part()
- strpos()
- translate()

对于正则表达式函数，如果你想按大小写敏感方式匹配，可以指定 “c” 标志来强制大小写敏感匹配。否则，若要获得大小写敏感行为，就必须在调用这些函数之前先把值转换为text。

F.8.4. 限制

- citext 的大小写折叠行为依赖于数据库的 LC_CTYPE 设置，因此它如何比较值是在创建数据库时确定的。按照 Unicode 标准中的定义，它并不是真正意义上的大小写不敏感。实际上，这意味着只要你对当前排序规则满意，通常也会对 citext 的比较结果满意。但是，如果数据库中存有多种语言的数据，而排序规则只适用于其中某一种语言，那么其他语言的用户可能会发现查询结果并不符合预期。
- 从 PostgreSQL 9.1 起，你可以为 citext 列或数据值附加 COLLATE 说明。当前，citext 操作符在比较已完成大小写折叠的字符串时，会遵从非默认的 COLLATE 说明；但最初转换为小写这一步，始终仍按数据库的 LC_CTYPE 设置执行（也就是说，等同于给出了 COLLATE "default"）。未来的发行版中，这一点可能会改变，从而使这两步都遵循输入的 COLLATE 说明。
- citext 不如 text 高效，因为操作符函数和 B-树比较函数必须复制数据，并将数据转换为小写后才能进行比较。不过，在需要大小写不敏感匹配时，它仍然比使用lower 略高效一些。
- 如果你在某些场景下需要大小写敏感比较，而在另一些场景下又需要大小写不敏感比较，那么 citext 并不会帮上太多忙。标准做法是使用 text 类型，并在需要大小写不敏感比较时手工调用 lower；如果只是偶尔需要这类比较，这种做法是完全可行的。如果你大多数时候都需要大小写不敏感行为，而只是在少数场景下需要大小写敏感比较，那么可以考虑把数据存储为 citext，并在需要大小写敏感比较时显式地把列转换为 text。无论哪种情况，如果你希望这两类搜索都足够快，就都需要建立两个索引。
- 包含 citext 操作符的模式必须位于当前 search_path 中（通常是 public）；如果不在，调用的将是普通的、大小写敏感的text 操作符。

F.8.5. 作者

David E. Wheeler <david@kineticode.com>

灵感来自 Donald Fraser 最初编写的 citext 模块。

F.9. cube

该模块实现了用于表示多维立方体的 cube 数据类型。

F.9.1. 语法

表 F.2 展示了 cube 类型合法的外部表示。*x*、*y* 等表示浮点数。

表 F.2. cube 外部表示

外部语法	含义
x	一个一维点（或者长度为零的一维区间）
(*x*)	同上
x1,*x2*,...,*xn*	n 维空间中的一个点，在内部表示为零体积立方体
(*x1*,*x2*,...,*xn*)	同上
(*x*),(*y*)	一个从*x* 开始、到*y* 结束的一维区间，反过来也可以；顺序无关紧要
[(*x*),(*y*)]	同上
(*x1*,...,*xn*),(*y1*,...,*yn*)	一个 n 维立方体，由一对对角相对的角点表示
[(*x1*,...,*xn*),(*y1*,...,*yn*)]	同上

立方体的两个对角点以何种顺序输入都无关紧要。必要时，cube 函数会自动交换这些值，以创建统一的“左下角 – 右上角”内部表示。当两个角点重合时，cube 只存储其中一个角点，并附带一个“是点”标志，以避免浪费空间。

输入时会忽略空白字符，因此 [(*x*),(*y*)] 与 [(*x*), (*y*)] 相同。

F.9.2. 精度

值在内部以 64 位浮点数存储。这意味着有效数字超过大约 16 位的数值会被截断。

F.9.3. 用法

表 F.3 展示了为 cube 类型提供的操作符。

表 F.3. cube 操作符

操作符	结果	描述
a = b	boolean	立方体 a 和 b 相同。
a && b	boolean	立方体 a 和 b 重叠。
a @> b	boolean	立方体 a 包含立方体 b。
a <@ b	boolean	立方体 a 包含在立方体 b 中。
a < b	boolean	立方体 a 小于立方体 b。
a <= b	boolean	立方体 a 小于或等于立方体 b。
a > b	boolean	立方体 a 大于立方体 b。
a >= b	boolean	立方体 a 大于或等于立方体 b。
a <> b	boolean	立方体 a 不等于立方体 b。
a -> n	float8	获取立方体的第 *n* 个坐标（从 1 开始计数）。

操作符	结果	描述
a ~> n	float8	按以下方式获取立方体的第 *n* 个坐标：n = 2 * k - 1 表示第 *k* 维的下界，n = 2 * k 表示第 *k* 维的上界。*n* 为负时，返回对应正编号所取坐标值的相反数。该操作符专为支持 KNN-GiST 而设计。
a <-> b	float8	a 和 b 之间的欧几里得距离。
a <#> b	float8	a 和 b 之间的出租车距离（L-1 度量）。
a <=> b	float8	a 和 b 之间的切比雪夫距离（L-无穷度量）。

（在 PostgreSQL 8.2 之前，包含操作符 @> 和 <@ 分别称为 @ 和 ~。这些名称仍然可用，但已弃用，最终将被删除。请注意，旧名称与核心几何数据类型以前采用的约定正好相反！）

标量排序操作符（<、>= 等）除了排序之外，在实际用途上没有太大意义。这些操作符首先比较第一个坐标；如果相等，再比较第二个坐标，依此类推。它们主要用于支持cube 的 B-树索引操作符类，例如，当你希望在cube 列上设置 UNIQUE 约束时可能会有用。

cube 模块还为 cube 值提供了一个 GiST 索引操作符类。cube GiST 索引可用于在 WHERE 子句中使用 =、&&、@> 和<@操作符搜索值。

此外，cube GiST 索引还可用于在 ORDER BY 子句中借助度量操作符 <->、<#> 和 <=> 查找最近邻。例如，三维点 (0.5, 0.5, 0.5) 的最近邻可以用下面的查询高效地找到：

```
SELECT c FROM test ORDER BY c <-> cube(array[0.5,0.5,0.5]) LIMIT 1;
```

~> 操作符也可以这样使用，以便高效地检索按选定坐标排序后的前几个值。例如，要获得按第一个坐标（左下角）升序排列的前几个立方体，可以使用下面的查询：

```
SELECT c FROM test ORDER BY c ~> 1 LIMIT 5;
```

要获得按右上角第一个坐标降序排列的二维立方体，可以使用：

```
SELECT c FROM test ORDER BY c ~> 3 DESC LIMIT 5;
```

表 F.4 展示了可用函数。

表 F.4. cube 函数

函数	结果	描述	示例
cube(float8)	cube	创建一个一维立方体，其两个坐标相同。	cube(1) == '(1)'
cube(float8, float8)	cube	创建一个一维立方体。	cube(1,2) == '(1), (2)'
cube(float8[])	cube	根据数组定义的坐标创建一个零体积立方体。	cube(ARRAY[1,2]) == '(1,2)'

函数	结果	描述	示例
cube(float8[], float8[])	cube	创建一个立方体，其右上角和左下角坐标由这两个数组定义，这两个数组必须等长。	cube(ARRAY[1,2], ARRAY[3,4]) == '(1,2),(3,4)'
cube(cube, float8)	cube	通过向现有立方体增加一个维度来创建新的立方体，新坐标的两个端点取相同的值。这可用于根据计算得到的值逐步构建立方体。	cube('(1,2),(3,4)'::cube, 5) == '(1,2,5),(3,4,5)'
cube(cube, float8, float8)	cube	通过向现有立方体增加一个维度来创建新的立方体。这可用于根据计算得到的值逐步构建立方体。	cube('(1,2),(3,4)'::cube, 5, 6) == '(1,2,5),(3,4,6)'
cube_dim(cube)	integer	返回立方体的维数。	cube_dim('(1,2),(3,4)') == '2'
cube_ll_coord(cube, integer)	float8	返回立方体左下角的第 *n* 个坐标值。	cube_ll_coord('(1,2),(3,4)', 2) == '2'
cube_ur_coord(cube, integer)	float8	返回立方体右上角的第 *n* 个坐标值。	cube_ur_coord('(1,2),(3,4)', 2) == '4'
cube_is_point(cube)	boolean	如果立方体是一个点，也就是定义它的两个角相同，则返回真。	
cube_distance(cube, cube)	float8	返回两个立方体之间的距离。如果两个立方体都是点，这就是普通的距离函数。	
cube_subset(cube, integer[])	cube	根据数组中给出的维度索引列表，从现有立方体创建一个新立方体。可用于提取单个维度的端点、删除维度，或者按需要重新排列维度。	cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[2]) == '(3),(7)' cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[3,2,1,1]) == '(5,3,1,1),(8,7,6,6)'
cube_union(cube, cube)	cube	生成两个立方体的并集。	
cube_inter(cube, cube)	cube	生成两个立方体的交集。	

函数	结果	描述	示例
cube_enlarge(cube cube, r double, n integer)	cube	将立方体至少 *n* 个维度的大小增加指定半径 *r*。如果半径为负，则改为缩小立方体。所有已定义的维度都会按半径 *r* 改变。左下角坐标减小 *r*，右上角坐标增大 *r*。如果某个左下角坐标增大到超过对应的右上角坐标（这只可能在 *r* < 0 时发生），则将两个坐标都设为它们的平均值。如果 *n* 大于已定义维度数且立方体正在扩大（*r* > 0），则添加额外维度，使总数达到 *n*；额外坐标的初始值使用 0。此函数对于创建点周围的边界框、以搜索附近点很有用。	cube_enlarge('(1,2),(3,4)', 0.5, 3) == '(0.5,1.5,-0.5),(3.5,4.5,0.5)'

F.9.4. 默认规则

我认为下面这个并集：

```
select cube_union('(0,5,2),(2,3,1)', '0');
cube_union
-------------------
(0, 0, 0),(2, 5, 2)
(1 row)
```

并不违背常识，下面这个交集也是如此：

```
select cube_inter('(0,-1),(1,1)', '(-2),(2)');
cube_inter
-------------
(0, 0),(1, 0)
(1 row)
```

在所有对不同维度立方体执行的二元操作中，我假定维度较低的那个是一个笛卡尔投影，也就是说，在字符串表示中省略的坐标位置上补 0。上面的例子等价于：

```
cube_union('(0,5,2),(2,3,1)','(0,0,0),(0,0,0)');
cube_inter('(0,-1),(1,1)','(-2,0),(2,0)');
```

下面的包含谓词使用的是点语法，但实际上第二个参数在内部表示为一个立方体。这种语法使我们无须单独定义点类型以及用于 (box,point) 谓词的函数。

```
select cube_contains('(0,0),(1,1)', '0.5,0.5');
cube_contains
--------------
t
(1 row)
```

F.9.5. 注意

有关用法示例，请参见回归测试`sql/cube.sql`。

为了避免用户轻易把事情弄坏，立方体的维数上限被设为 100。如果需要更大的值，可在`cubedata.h` 中修改该限制。

F.9.6. 致谢

原作者：Gene Selkov, Jr. <selkovjr@mcs.anl.gov>，阿贡国家实验室数学与计算机科学部。

我首先要感谢 Joe Hellerstein 教授 (https://dsf.berkeley.edu/jmh/)，他为我阐明了 GiST (http://gist.cs.berkeley.edu/) 的要旨；也感谢他曾经的学生 Andy Dong 为 Illustra 编写了示例。我同样感谢过去和现在所有的 Postgres 开发者，他们使我得以创造自己的世界并在其中不受打扰地生活。我还要感谢阿贡实验室以及美国能源部，多年来始终如一地支持我的数据库研究。

Bruno Wolff III <bruno@wolff.to> 在 2002 年 8 月和 9 月对该软件包做了一些小更新，包括将精度从单精度改为双精度，并增加了一些新函数。

Joshua Reich <josh@root.net> 在 2006 年 7 月做了进一步更新。这些更新包括加入 `cube( float8[], float8[])`，并清理代码，使其使用 V1 调用协议而不是已废弃的 V0 协议。

F.10. dblink

`dblink` 是一个支持在数据库会话内连接到其他 PostgreSQL 数据库的模块。

另请参见 postgres_fdw，它使用更现代且更符合标准的基础设施提供了基本相同的功能。

dblink_connect

dblink_connect — 打开到远程数据库的持久连接

大纲

```
dblink_connect(text connstr) returns text
dblink_connect(text connname, text connstr) returns text
```

描述

`dblink_connect()` 建立到远程 PostgreSQL 数据库的连接。要联系的服务器和数据库通过标准的 libpq 连接字符串标识。可以为该连接指定一个名称。可同时打开多个命名连接，但同一时间只允许存在一个未命名连接。该连接会一直保留到被关闭或数据库会话结束。

连接字符串也可以是现有外部服务器的名称。定义该外部服务器时，建议使用外部数据包装器 `dblink_fdw`。请参见下面的示例，以及CREATE SERVER 和 CREATE USER MAPPING。

参数

connname

用于该连接的名称；如果省略，则打开一个未命名连接，并替换现有的未命名连接。

connstr

libpq-风格的连接信息字符串，例如 `hostaddr=127.0.0.1 port=5432 dbname=mydb user=postgres password=mypasswd options=-csearch_path=`。详见第 34.1.1 节。此外，也可以是外部服务器的名称。

返回值

返回状态，总是 `OK`（因为任何错误都会导致该函数抛出错误，而不是返回）。

注解

如果不受信任的用户能够访问一个尚未采用模式的安全使用方式的数据库，则应在每个会话开始时从 `search_path` 中移除公共可写模式。例如，可以把 `options=-csearch_path=` 加到 *connstr* 中。这个注意事项并非 `dblink` 所特有；它适用于每一种执行任意 SQL 命令的接口。

只有超级用户才能使用 `dblink_connect` 创建不使用密码认证的连接。如果非超级用户需要这种能力，请改用 `dblink_connect_u`。

选择包含等号的连接名称并不明智，因为这会导致在其他 `dblink` 函数中与连接信息字符串产生混淆风险。

示例

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_connect('myconn', 'dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

-- FOREIGN DATA WRAPPER 功能
-- 注意：要使本例正常工作，本地连接必须要求密码认证。
--       否则，dblink_connect() 将报告以下错误：
--       ------------------------------------------------------------------------
--       ERROR:  password is required
--       DETAIL:  Non-superuser cannot connect if the server does not request a password.
--       HINT:  Target server's authentication method must be changed.

CREATE SERVER fdtest FOREIGN DATA WRAPPER dblink_fdw OPTIONS (hostaddr '127.0.0.1', dbname
 'contrib_regression');

CREATE USER regress_dblink_user WITH PASSWORD 'secret';
CREATE USER MAPPING FOR regress_dblink_user SERVER fdtest OPTIONS (user 'regress_dblink_user',
 password 'secret');
GRANT USAGE ON FOREIGN SERVER fdtest TO regress_dblink_user;
GRANT SELECT ON TABLE foo TO regress_dblink_user;

\set ORIGINAL_USER :USER
\c - regress_dblink_user
SELECT dblink_connect('myconn', 'fdtest');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'SELECT * FROM foo') AS t(a int, b text, c text[]);
 a  | b |       c
----+---+---------------
  0 | a | {a0,b0,c0}
  1 | b | {a1,b1,c1}
  2 | c | {a2,b2,c2}
  3 | d | {a3,b3,c3}
  4 | e | {a4,b4,c4}
  5 | f | {a5,b5,c5}
  6 | g | {a6,b6,c6}
  7 | h | {a7,b7,c7}
  8 | i | {a8,b8,c8}
  9 | j | {a9,b9,c9}
 10 | k | {a10,b10,c10}
(11 rows)

\c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM regress_dblink_user;
REVOKE SELECT ON TABLE foo FROM regress_dblink_user;
DROP USER MAPPING FOR regress_dblink_user SERVER fdtest;
DROP USER regress_dblink_user;
DROP SERVER fdtest;
```

dblink_connect_u

dblink_connect_u — 不安全地打开到远程数据库的持久连接

大纲

```
dblink_connect_u(text connstr) returns text
dblink_connect_u(text connname, text connstr) returns text
```

描述

`dblink_connect_u()` 与 `dblink_connect()` 相同，不同之处在于它允许非超级用户使用任意认证方法进行连接。

如果远程服务器选择的认证方法不涉及密码，则可能发生身份冒充以及随之而来的权限提升，因为该会话看起来会像是由运行本地 PostgreSQL 服务器的那个用户发起的。此外，即使远程服务器确实要求密码，密码也有可能来自服务器环境，例如属于服务器用户的 `~/.pgpass` 文件。这不仅带来身份冒充的风险，也可能将密码暴露给不可信的远程服务器。因此，`dblink_connect_u()` 在初始安装时会撤销 `PUBLIC` 的全部权限，从而除了超级用户之外无法调用它。在某些情况下，可能适合向被认为可信的特定用户授予 `dblink_connect_u()` 的 `EXECUTE` 权限，但必须谨慎操作。还建议服务器用户的任何 `~/.pgpass` 文件不要包含指定通配主机名的记录。

更多细节请参见 `dblink_connect()`。

dblink_disconnect

dblink_disconnect — 关闭到远程数据库的持久连接

大纲

```
dblink_disconnect() returns text
dblink_disconnect(text connname) returns text
```

描述

`dblink_disconnect()` 关闭先前由 `dblink_connect()` 打开的连接。不带参数的形式会关闭未命名连接。

参数

connname

要关闭的命名连接名称。

返回值

返回状态，总是 `OK`（因为任何错误都会导致该函数抛出错误，而不是返回）。

示例

```
SELECT dblink_disconnect();
 dblink_disconnect
-------------------
 OK
(1 row)

SELECT dblink_disconnect('myconn');
 dblink_disconnect
-------------------
 OK
(1 row)
```

dblink

dblink — 在远程数据库中执行查询

大纲

```
dblink(text connname, text sql [, bool fail_on_error]) returns setof record
dblink(text connstr, text sql [, bool fail_on_error]) returns setof record
dblink(text sql [, bool fail_on_error]) returns setof record
```

描述

`dblink` 在远程数据库中执行查询（通常是 `SELECT`，但也可以是任意返回行的 SQL 语句）。

当提供两个 `text` 参数时，首先会将第一个参数作为持久连接名称查找；如果找到，就在该连接上执行命令。如果没有找到，则把第一个参数视为像 `dblink_connect` 那样的连接信息字符串，并且只在本命令执行期间建立所指明的连接。

参数

connname

要使用的连接名称；省略该参数则使用未命名连接。

connstr

如前面对 `dblink_connect` 所描述的连接信息字符串。

sql

要在远程数据库中执行的 SQL 查询，例如 `select * from foo`。

fail_on_error

如果为真（省略时默认），则连接远程端抛出的错误也会在本地抛出。如果为假，远程端错误会在本地报告为一个 NOTICE，并且该函数不返回任何行。

返回值

该函数返回查询产生的行。由于 dblink 可以与任意查询一起使用，因此它被声明为返回 record，而不是指定某个特定的列集。这意味着必须在调用查询中指定预期的列集 – 否则 PostgreSQL 无法知道应该期待什么。以下是一个示例：

```
SELECT *
    FROM dblink('dbname=mydb options=-csearch_path=',
                'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

FROM 子句中的“别名”部分必须指定该函数将返回的列名和类型。（在别名中指定列名实际上是标准 SQL 语法，但指定列类型则是 PostgreSQL 的扩展。）这样系统才能在尝试执行该函数之前，就知道 * 应展开成什么，以及 WHERE 子句中的 proname 指的是什么。运行时，如果远程数据库返回的实际查询结果与 FROM 子句中列出的列数不同，就会抛出错误。不过，列名不必匹配，dblink 也不要求类型完全一致。只要返回的数据字符串可以作为 FROM 子句中声明的列类型的有效输入，它就会成功。

注解

将预定义查询用于 dblink 的一种方便方式是创建视图。这样就可以把列类型信息隐藏在视图中，而不必在每个查询里都重复写出。例如，

```
CREATE VIEW myremote_pg_proc AS
  SELECT *
    FROM dblink('dbname=postgres options=-csearch_path=',
                'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

示例

```
SELECT * FROM dblink('dbname=postgres options=-csearch_path=',
                     'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)
```

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('myconn', 'dbname=regression options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 bytearecv  | bytearecv
 byteasend  | byteasend
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteain    | byteain
 byteaout   | byteaout
(14 rows)
```

dblink_exec

dblink_exec — 在远程数据库中执行命令

大纲

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
dblink_exec(text sql [, bool fail_on_error]) returns text
```

描述

`dblink_exec` 在远程数据库中执行命令（即任何不返回行的 SQL 语句）。

当提供两个 `text` 参数时，首先会将第一个参数作为持久连接名称查找；如果找到，就在该连接上执行命令。如果没有找到，则把第一个参数视为像 `dblink_connect` 那样的连接信息字符串，并且只在本命令执行期间建立所指明的连接。

参数

connname

要使用的连接名称；省略该参数则使用未命名连接。

connstr

如前面对 `dblink_connect` 所描述的连接信息字符串。

sql

要在远程数据库中执行的 SQL 命令，例如 `insert into foo values(0, 'a', '{"a0","b0","c0"}')`。

fail_on_error

如果为真（省略时默认），则连接远程端抛出的错误也会在本地抛出。如果为假，远程端错误会在本地报告为一个 NOTICE，并且该函数的返回值会被设为 `ERROR`。

返回值

返回状态，可以是命令的状态字符串，也可以是 `ERROR`。

示例

```
SELECT dblink_connect('dbname=dblink_test_standby');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('insert into foo values(21, ''z'', ''{"a0","b0","c0"}'');');
   dblink_exec
-----------------
 INSERT 943366 1
(1 row)

SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('myconn', 'insert into foo values(21, ''z'', ''{"a0","b0","c0"}'');');
   dblink_exec
------------------
 INSERT 6432584 1
(1 row)

SELECT dblink_exec('myconn', 'insert into pg_class values (''foo'')',false);
NOTICE:  sql error
DETAIL:  ERROR:  null value in column "relnamespace" violates not-null constraint

 dblink_exec
-------------
 ERROR
(1 row)
```

dblink_open

dblink_open — 在远程数据库中打开游标

大纲

```
dblink_open(text cursorname, text sql [, bool fail_on_error]) returns text
dblink_open(text connname, text cursorname, text sql [, bool fail_on_error]) returns text
```

描述

`dblink_open()` 在远程数据库中打开游标。随后可以使用 `dblink_fetch()` 和 `dblink_close()` 操作该游标。

参数

connname

要使用的连接名称；省略该参数则使用未命名连接。

cursorname

要分配给该游标的名称。

sql

要在远程数据库中执行的 `SELECT` 语句，例如 `select * from pg_class`。

fail_on_error

如果为真（省略时默认），则连接远程端抛出的错误也会在本地抛出。如果为假，远程端错误会在本地报告为一个 NOTICE，并且该函数的返回值会被设为 `ERROR`。

返回值

返回状态，可以是 `OK` 或 `ERROR`。

注解

由于游标只能在事务中持续存在，如果远程端当前尚未处于事务中，`dblink_open` 会在远程端启动一个显式事务块（`BEGIN`）。当执行匹配的 `dblink_close` 时，该事务会再次关闭。注意，如果在 `dblink_open` 和 `dblink_close` 之间用 `dblink_exec` 修改了数据，随后又发生错误，或者在调用 `dblink_close` 之前使用了 `dblink_disconnect`，所做的更改将会丢失，因为该事务会被中止。

示例

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)
```

dblink_fetch

dblink_fetch — 返回远程数据库中已打开游标的行

大纲

```
dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) returns setof record
dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error]) returns setof
 record
```

描述

dblink_fetch 从先前由 dblink_open 建立的游标中提取行。

参数

connname

要使用的连接名称；省略该参数则使用未命名连接。

cursorname

要从中提取数据的游标名称。

howmany

要检索的最大行数。从当前游标位置开始，向前提取接下来的 *howmany* 行。一旦游标到达末尾，就不会再产生更多行。

fail_on_error

如果为真（省略时默认），则连接远程端抛出的错误也会在本地抛出。如果为假，远程端错误会在本地报告为一个 NOTICE，并且该函数不返回任何行。

返回值

该函数返回从游标提取的行。要使用此函数，需要像前面讨论 dblink 时那样，指定预期的列集。

注解

如果 FROM 子句中指定的返回列数与远程游标实际返回的列数不一致，就会抛出错误。在这种情况下，远程游标仍会像没有发生该错误一样前进相同的行数。远程 FETCH 完成之后，本地查询中发生的任何其他错误也是如此。

示例

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc where proname like ''bytea%''');
 dblink_open
-------------
 OK
(1 row)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname |  source
----------+----------
 byteacat | byteacat
 byteacmp | byteacmp
 byteaeq  | byteaeq
 byteage  | byteage
 byteagt  | byteagt
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname  |  source
-----------+-----------
 byteain   | byteain
 byteale   | byteale
 bytealike | bytealike
 bytealt   | bytealt
 byteane   | byteane
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname  |   source
------------+------------
 byteanlike | byteanlike
 byteaout   | byteaout
(2 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname | source
----------+--------
(0 rows)
```

dblink_close

dblink_close — 关闭远程数据库中的游标

大纲

```
dblink_close(text cursorname [, bool fail_on_error]) returns text
dblink_close(text connname, text cursorname [, bool fail_on_error]) returns text
```

描述

`dblink_close` 关闭先前由 `dblink_open` 打开的游标。

参数

connname

要使用的连接名称；省略该参数则使用未命名连接。

cursorname

要关闭的游标名。

fail_on_error

如果为真（省略时默认），则连接远程端抛出的错误也会在本地抛出。如果为假，远程端错误会在本地报告为一个 NOTICE，并且该函数的返回值会被设为 `ERROR`。

返回值

返回状态，可以是 `OK` 或 `ERROR`。

注解

如果 `dblink_open` 启动了显式事务块，并且这是此连接上最后一个仍然打开的游标，`dblink_close` 将发出对应的 `COMMIT`。

示例

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)

SELECT dblink_close('foo');
 dblink_close
--------------
 OK
(1 row)
```

dblink_get_connections

dblink_get_connections — 返回所有打开的命名 dblink 连接的名称

大纲

```
dblink_get_connections() returns text[]
```

描述

`dblink_get_connections` 返回包含所有已打开命名 `dblink` 连接名称的数组。

返回值

返回包含连接名称的 text 数组；如果没有连接，则返回 NULL。

示例

```
SELECT dblink_get_connections();
```

dblink_error_message

dblink_error_message — 获取命名连接上的最后一条错误消息

大纲

```
dblink_error_message(text connname) returns text
```

描述

`dblink_error_message` 获取给定连接上最近一次的远程错误消息。

参数

connname

要使用的连接名称。

返回值

返回最后一条错误消息；如果该连接没有发生错误，则返回 `OK`。

注解

在由 `dblink_send_query` 发起异步查询时，与该连接关联的错误消息在读取服务器响应消息之前可能不会更新。这通常意味着应该先调用 `dblink_is_busy` 或 `dblink_get_result`，再调用 `dblink_error_message`，以便看到异步查询产生的任何错误。

示例

```
SELECT dblink_error_message('dtest1');
```

dblink_send_query

dblink_send_query — 向远程数据库发送异步查询

大纲

```
dblink_send_query(text connname, text sql) returns int
```

描述

`dblink_send_query` 发送要异步执行的查询，也就是不会立即等待结果。该连接上不能已经有正在进行的异步查询。

异步查询成功发送后，可以用 `dblink_is_busy` 检查完成状态，并最终通过 `dblink_get_result` 收集结果。也可以使用 `dblink_cancel_query` 尝试取消活动中的异步查询。

参数

connname

要使用的连接名称。

sql

要在远程数据库中执行的 SQL 语句，例如`select * from pg_class`。

返回值

如果查询已成功发送，则返回 1；否则返回 0。

示例

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

dblink_is_busy

dblink_is_busy — 检查连接是否正忙于异步查询

大纲

```
dblink_is_busy(text connname) returns int
```

描述

`dblink_is_busy` 测试是否有异步查询正在进行。

参数

connname

要检查的连接名称。

返回值

如果连接正忙则返回 1，否则返回 0。如果此函数返回 0，则可以保证 `dblink_get_result` 不会阻塞。

示例

```
SELECT dblink_is_busy('dtest1');
```

dblink_get_notify

dblink_get_notify — 检索连接上的异步通知

大纲

```
dblink_get_notify() returns setof (notify_name text, be_pid int, extra text)
dblink_get_notify(text connname) returns setof (notify_name text, be_pid int, extra text)
```

描述

`dblink_get_notify` 检索未命名连接上的通知；如果指定了命名连接，则检索该连接上的通知。要通过 dblink 接收通知，必须先使用 `dblink_exec` 执行 `LISTEN`。详见 LISTEN 和 NOTIFY。

参数

connname

要检索通知的命名连接名称。

返回值

返回 `setof (notify_name text, be_pid int, extra text)`；如果没有，则返回空集。

示例

```
SELECT dblink_exec('LISTEN virtual');
 dblink_exec
-------------
 LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
(0 rows)

NOTIFY virtual;
NOTIFY

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
 virtual     |   1229 |
(1 row)
```

dblink_get_result

dblink_get_result — 获取异步查询结果

大纲

```
dblink_get_result(text connname [, bool fail_on_error]) returns setof record
```

描述

dblink_get_result 收集此前用 dblink_send_query 发送的异步查询结果。如果该查询尚未完成，dblink_get_result 将等待直至其完成。

参数

connname

要使用的连接名称。

fail_on_error

如果为真（省略时默认），则连接远程端抛出的错误也会在本地抛出。如果为假，远程端错误会在本地报告为一个 NOTICE，并且该函数不返回任何行。

返回值

对于异步查询（即返回行的 SQL 语句），该函数返回查询产生的行。要使用此函数，需要像前面讨论 dblink 时那样，指定预期的列集。

对于异步命令（即不返回行的 SQL 语句），该函数返回一个只有一个 text 列的单行，其中包含命令的状态字符串。即便如此，仍需要在调用查询的 FROM 子句中指定结果只有一个 text 列。

注解

如果 dblink_send_query 返回 1，则必须调用此函数。每发送一个查询都必须调用一次，并且还要额外再调用一次以获得空集结果，之后该连接才能再次使用。

当使用 dblink_send_query 和 dblink_get_result 时，dblink 会在向本地查询处理器返回任何结果行之前，先获取整个远程查询结果。如果查询返回大量行，这可能导致本地会话暂时出现内存膨胀。对于此类查询，最好用 dblink_open 将其作为游标打开，然后每次提取可管理数量的行。另一种做法是使用普通的 dblink()，它会将大型结果集溢写到磁盘，从而避免内存膨胀。

示例

```
contrib_regression=# SELECT dblink_connect('dtest1', 'dbname=contrib_regression');
 dblink_connect
----------------
 OK
(1 row)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]
);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]
);
 f1 | f2 | f3
----+----+----
(0 rows)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3; select * from
 foo where f1 > 6') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]
);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]
);
 f1 | f2 |      f3
----+----+---------------
  7 | h  | {a7,b7,c7}
  8 | i  | {a8,b8,c8}
  9 | j  | {a9,b9,c9}
 10 | k  | {a10,b10,c10}
(4 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]
);
 f1 | f2 | f3
----+----+----
(0 rows)
```

dblink_cancel_query

dblink_cancel_query — 取消命名连接上的任何活动查询

大纲

```
dblink_cancel_query(text connname) returns text
```

描述

`dblink_cancel_query` 尝试取消命名连接上正在进行的任何查询。注意，这不一定会成功（例如远程查询可能已经结束）。取消请求只会提高该查询很快失败的概率。仍必须完成正常的查询协议，例如调用 `dblink_get_result`。

参数

connname

要使用的连接名称。

返回值

如果取消请求已发送，则返回 `OK`；如果失败，则返回错误消息文本。

示例

```
SELECT dblink_cancel_query('dtest1');
```

dblink_get_pkey

dblink_get_pkey — 返回关系主键字段的位置和字段名

大纲

```
dblink_get_pkey(text relname) returns setof dblink_pkey_results
```

描述

`dblink_get_pkey` 提供有关本地数据库中某个关系主键的信息。这有时有助于生成要发送到远程数据库的查询。

参数

relname

本地关系的名称，例如 `foo` 或 `myschema.mytab`。如果该名称是大小写混合的或包含特殊字符，则应包含双引号，例如 `"FooBar"`；如果没有引号，字符串将被折叠为小写形式。

返回值

为每个主键字段返回一行；如果该关系没有主键，则不返回任何行。结果行类型定义如下：

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

`position` 列只是从 1 到 *N* 递增；它表示该字段在主键中的序号，而不是在表列中的序号。

示例

```
CREATE TABLE foobar (
    f1 int,
    f2 int,
    f3 int,
    PRIMARY KEY (f1, f2, f3)
);
CREATE TABLE

SELECT * FROM dblink_get_pkey('foobar');
 position | colname
----------+---------
        1 | f1
        2 | f2
        3 | f3
(3 rows)
```

dblink_build_sql_insert

dblink_build_sql_insert — 使用本地元组构造 INSERT 语句，并用提供的替代值替换主键字段值

大纲

```
dblink_build_sql_insert(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

描述

`dblink_build_sql_insert` 可用于将本地表有选择地复制到远程数据库。它根据主键从本地表中选取一行，然后构建一条会复制该行的 SQL `INSERT` 命令，但主键值会替换为最后一个参数中的值。（若要精确复制该行，只需为最后两个参数指定相同的值。）

参数

relname

本地关系的名称，例如 `foo` 或 `myschema.mytab`。如果该名称是大小写混合的或包含特殊字符，则应包含双引号，例如 `"FooBar"`；如果没有引号，字符串将被折叠为小写形式。

primary_key_attnums

主键字段的属性编号（从 1 开始），例如 `1 2`。

num_primary_key_atts

主键字段数量。

src_pk_att_vals_array

用于查找本地元组的主键字段值。每个字段都以文本形式表示。如果没有本地行具有这些主键值，则抛出错误。

tgt_pk_att_vals_array

放入结果 `INSERT` 命令中的主键字段值。每个字段都以文本形式表示。

返回值

以文本形式返回所请求的 SQL 语句。

注解

从 PostgreSQL 9.0 起，*primary_key_attnums* 中的属性编号会被解释为逻辑列号，对应于列在 `SELECT * FROM relname` 中的位置。早期版本将这些编号解释为物理列位置。如果在表的生命周期中，指示列左侧的某些列已被删除，两者就会有差异。

示例

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b''a"}');
             dblink_build_sql_insert
-------------------------------------------------
 INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')
(1 row)
```

dblink_build_sql_delete

dblink_build_sql_delete — 使用提供的主键字段值构造 DELETE 语句

大纲

```
dblink_build_sql_delete(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] tgt_pk_att_vals_array) returns text
```

描述

`dblink_build_sql_delete` 可用于将本地表有选择地复制到远程数据库。它会构建一条 SQL `DELETE` 命令，用来删除具有给定主键值的行。

参数

relname

本地关系的名称，例如 `foo` 或 `myschema.mytab`。如果该名称是大小写混合的或包含特殊字符，则应包含双引号，例如 `"FooBar"`；如果没有引号，字符串将被折叠为小写形式。

primary_key_attnums

主键字段的属性编号（从 1 开始），例如 `1 2`。

num_primary_key_atts

主键字段数量。

tgt_pk_att_vals_array

用于结果 `DELETE` 命令中的主键字段值。每个字段都以文本形式表示。

返回值

以文本形式返回所请求的 SQL 语句。

注解

从 PostgreSQL 9.0 起，*primary_key_attnums* 中的属性编号会被解释为逻辑列号，对应于列在 `SELECT * FROM relname` 中的位置。早期版本将这些编号解释为物理列位置。如果在表的生命周期中，指示列左侧的某些列已被删除，两者就会有差异。

示例

```
SELECT dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
           dblink_build_sql_delete
---------------------------------------------
 DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
```

dblink_build_sql_update

dblink_build_sql_update — 使用本地元组构造 UPDATE 语句，并用提供的替代值替换主键字段值

大纲

```
dblink_build_sql_update(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

描述

`dblink_build_sql_update` 可用于将本地表有选择地复制到远程数据库。它根据主键从本地表中选取一行，然后构建一条会复制该行的 SQL `UPDATE` 命令，但主键值会替换为最后一个参数中的值。（若要精确复制该行，只需为最后两个参数指定相同的值。）`UPDATE` 命令总是对该行的所有字段赋值 – 它与 `dblink_build_sql_insert` 的主要区别在于，前者假定目标行已经存在于远程表中。

参数

relname

本地关系的名称，例如 `foo` 或 `myschema.mytab`。如果该名称是大小写混合的或包含特殊字符，则应包含双引号，例如 `"FooBar"`；如果没有引号，字符串将被折叠为小写形式。

primary_key_attnums

主键字段的属性编号（从 1 开始），例如 `1 2`。

num_primary_key_atts

主键字段数量。

src_pk_att_vals_array

用于查找本地元组的主键字段值。每个字段都以文本形式表示。如果没有本地行具有这些主键值，则抛出错误。

tgt_pk_att_vals_array

用于结果 `UPDATE` 命令中的主键字段值。每个字段都以文本形式表示。

返回值

以文本形式返回所请求的 SQL 语句。

注解

从 PostgreSQL 9.0 起，*primary_key_attnums* 中的属性编号会被解释为逻辑列号，对应于列在 SELECT * FROM relname 中的位置。早期版本将这些编号解释为物理列位置。如果在表的生命周期中，指示列左侧的某些列已被删除，两者就会有差异。

示例

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"}');
                   dblink_build_sql_update
-------------------------------------------------------------
 UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

F.11. dict_int

dict_int 是一个全文检索附加词典模板的示例。引入这个示例词典是为了控制整数（有符号和无符号）的索引，使这类数字能够被索引，同时又避免不同词的数量过度增长，因为那会严重影响搜索性能。

F.11.1. 配置

该词典接受两个选项：

- maxlen 参数指定整数词中允许的最大数字位数。默认值为 6。
- rejectlong 参数指定超长整数应被截断还是忽略。如果 rejectlong 为 false（默认值），词典将返回该整数的前maxlen 位数字。如果 rejectlong 为 true，词典会将超长整数视为停用词，因此不会为其建立索引。注意，这也意味着无法搜索这类整数。

F.11.2. 用法

安装 dict_int 扩展后，会创建一个文本检索模板 intdict_template 以及一个基于该模板、使用默认参数的词典 intdict。你可以修改这些参数，例如：

```
mydb# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

或者基于该模板创建新的词典。

要测试该词典，可以尝试：

```
mydb# select ts_lexize('intdict', '12345678');
 ts_lexize
-----------
 {123456}
```

但在实际使用中，通常需要像第 12 章所述那样，将它包含到某个文本检索配置中。可能类似如下：

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR int, uint WITH intdict;
```

F.12. dict_xsyn

dict_xsyn（扩展同义词词典）是全文检索的一个附加词典模板示例。这种词典类型会将词替换为其同义词组，因此可以使用某个词的任一同义词来搜索该词。

F.12.1. 配置

该 dict_xsyn 词典接受以下选项：

- matchorig 控制词典是否接受原词。默认为 true。
- matchsynonyms 控制词典是否接受同义词。默认为 false。
- keeporig 控制词典输出中是否包含原词。默认为 true。
- keepsynonyms 控制词典输出中是否包含同义词。默认为 true。
- rules 是包含同义词列表的文件的基本名。该文件必须存放在 $SHAREDIR/tsearch_data/（其中 $SHAREDIR 表示 PostgreSQL 安装的共享数据目录）中。文件名必须以.rules 结尾（这一后缀不应写入 rules 参数）。

规则文件具有下列格式：

- 每一行表示某个单词的一组同义词，且该单词位于行首。同义词之间用空白分隔，例如：

```
word syn1 syn2 syn3
```

- 井号（#）是注释分隔符。它可以出现在一行中的任意位置，该行余下的内容都会被跳过。

示例可参见安装在 $SHAREDIR/tsearch_data/中的 xsyn_sample.rules。

F.12.2. 用法

安装 dict_xsyn 扩展会使用默认参数创建一个全文检索模板 xsyn_template 和一个基于它的词典 xsyn。你可以修改这些参数，例如：

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

或者基于该模板创建新的词典。

要测试该词典，可以尝试：

```
mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {word,syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false, MATCHSYNONYMS=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true, MATCHORIG=false,
 KEEPSYNONYMS=false);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {word}
```

但实际使用通常需要把它纳入第 12 章中描述的全文检索配置。可能会像这样：

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;
```

F.13. earthdistance

`earthdistance` 模块提供了两种不同的方法，用于计算地球表面上的大圆距离。首先介绍的一种依赖于 `cube` 模块。第二种方法基于内置的 `point` 数据类型，使用经度和纬度作为坐标。

在这个模块中，地球被假定为完全球形的。（如果这种假设对你而言精度不足，可以参见 PostGIS[1] 项目。）

必须先安装 `cube` 模块，才能安装 `earthdistance`（不过，你也可以使用 `CREATE EXTENSION` 的 `CASCADE` 选项，用一条命令同时安装这两个扩展）。

> **小心**
>
> 强烈建议将 `earthdistance` 和 `cube` 安装到同一个模式中，并且该模式过去没有、将来也不会向任何不受信任的用户授予 CREATE 权限。否则，如果 `earthdistance` 所在的模式包含由恶意用户定义的对象，就会在安装时产生安全隐患。此外，在安装后使用 `earthdistance` 的函数时，整个搜索路径都应当只包含受信任的模式。

F.13.1. 基于立方体的地球距离

数据以立方体形式存储，而这些立方体实际上是点（两个角相同），使用 3 个坐标分别表示距地心的 x、y、z 距离。这里还提供了一个基于 `cube` 的域 `earth`，其中包含约束检查，用以确认该值满足这些限制，并且与地球实际表面足够接近。

地球半径取自 `earth()` 函数，其单位为米。不过，只要修改这一个函数，你就可以让该模块改用其他单位，或者使用你认为更合适的半径值。

这个模块在天文数据库中也有用途。天文学家大概会希望修改 `earth()`，使其返回半径 `180/pi()`，这样距离单位就是度。

这里提供了一些函数，用于按经纬度（以度为单位）输入、输出经纬度、计算两点之间的大圆距离，以及方便地指定一个可用于索引搜索的边界框。

所提供的函数见表 F.5。

表 F.5. 基于立方体的地球距离函数

函数	返回值	描述
`earth()`	`float8`	返回假定的地球半径。
`sec_to_gc(float8)`	`float8`	将地球表面两点间的普通直线（割线）距离转换为它们之间的大圆距离。
`gc_to_sec(float8)`	`float8`	将地球表面两点间的大圆距离转换为它们之间的普通直线（割线）距离。
`ll_to_earth(float8, float8)`	`earth`	给定某点以度为单位的纬度（参数 1）和经度（参数 2），返回该点在地球表面上的位置。
`latitude(earth)`	`float8`	返回地球表面上一个点的纬度（以度数形式）。

[1] https://postgis.net/

函数	返回值	描述
`longitude(earth)`	`float8`	返回地球表面上一个点的经度（以度数形式）。
`earth_distance(earth, earth)`	`float8`	返回地球表面上两个点之间的大圆距离。
`earth_box(earth, float8)`	`cube`	返回一个适合使用 cube `@>` 操作符进行索引搜索的框，用于查找距离某个位置在给定大圆距离以内的点。框中的某些点到该位置的距离会超过指定的大圆距离，因此查询中应包含使用`earth_distance` 的二次检查。

F.13.2. 基于点的地球距离

该模块的第二部分依赖于将地球上的位置表示为 `point` 类型的值，其中第一个分量被视为以度为单位的经度，第二个分量被视为以度为单位的纬度。之所以采用 (longitude, latitude) 而不是相反的顺序，是因为经度更接近 x 轴的直观概念，而纬度更接近 y 轴。

这一部分只提供了一个操作符，见表 F.6。

表 F.6. 基于点的地球距离操作符

操作符	返回值	描述
`point <@> point`	`float8`	给出地球表面两点之间的法定英里距离。

注意，与该模块基于 `cube` 的部分不同，这里的单位是固定的：修改 `earth()` 函数不会影响该操作符的结果。

经度/纬度表示的一个缺点是，你需要小心靠近两极以及接近经度 +/- 180 度时的边界情况。基于 `cube` 的表示可以避免这些不连续性。

F.14. file_fdw

file_fdw 模块提供了外部数据包装器 file_fdw，可用于访问服务器文件系统中的数据文件，或者在服务器上执行程序并读取其输出。数据文件或程序输出必须采用 COPY FROM 可读取的格式；详见 COPY。当前对数据文件的访问为只读。

使用此包装器创建的外部表可以具有下列选项：

filename

指定要读取的文件。相对路径以数据目录为基准。必须指定 filename 或 program 之一，但不能同时指定二者。

program

指定要执行的命令。该命令的标准输出会按使用 COPY FROM PROGRAM 时的方式读取。必须指定 program 或 filename 之一，但不能同时指定二者。

format

指定数据格式，与COPY 的 FORMAT 选项相同。

header

指定数据是否包含标题行，与COPY 的 HEADER 选项相同。

delimiter

指定数据的定界符字符，与COPY 的 DELIMITER 选项相同。

quote

指定数据的引用字符，与COPY 的 QUOTE 选项相同。

escape

指定数据的转义字符，与COPY 的 ESCAPE 选项相同。

null

指定数据的空值串，与COPY 的 NULL 选项相同。

encoding

指定数据编码，与COPY 的 ENCODING 选项相同。

请注意，虽然COPY 允许诸如 HEADER 这样的选项在没有对应值的情况下指定，但外部表选项语法要求在所有情况下都必须提供一个值。要启用通常写成不带值的 COPY 选项，可以传入值 TRUE，因为所有这类选项都是布尔值。

使用此包装器创建的外部表的列可以具有下列选项：

force_not_null

这是一个布尔选项。如果为真，它指定该列的值不应与空值串（即表级的 null 选项）匹配。其效果与在 COPY 的 FORCE_NOT_NULL 选项中列出该列相同。

force_null

这是一个布尔选项。如果为真，它指定该列中与空值串匹配的值即使带有引号，也会作为 NULL 返回。没有此选项时，只有未加引号且与空值串匹配的值才会作为NULL 返回。其效果与在 COPY 的 FORCE_NULL 选项中列出该列相同。

file_fdw 当前尚不支持 COPY 的 OIDS 和 FORCE_QUOTE 选项。

这些选项只能在外部表或其列上指定，不能在file_fdw 外部数据包装器的选项中指定，也不能在使用该包装器的服务器或用户映射的选项中指定。

出于安全原因，更改表级选项需要是超级用户，或者拥有默认角色pg_read_server_files（使用文件名时）或默认角色 pg_execute_server_program（使用程序时）的权限：只有特定用户才应能够控制读取哪个文件或运行哪个程序。原则上也可以允许普通用户更改其他选项，但目前尚不支持。

在指定 program 选项时，请记住，该选项字符串会由 shell 执行。如果需要向该命令传递来自不受信任来源的参数，必须谨慎去除或转义任何对 shell 可能具有特殊含义的字符。出于安全原因，最好使用固定的命令字符串，或者至少避免向其中传递任何用户输入。

对于使用 file_fdw 的外部表，EXPLAIN 会显示待读取的文件名或待运行的程序名。对于文件，除非指定COSTS OFF，否则还会显示文件大小（以字节为单位）。

例 F.1. 为 PostgreSQL CSV 日志创建外部表

file_fdw 的一个明显用途，是将 PostgreSQL 活动日志作为表提供以供查询。要做到这一点，首先必须将日志记录到 CSV 文件中，这里将该文件称为 pglog.csv。首先，将file_fdw 安装为扩展：

```
CREATE EXTENSION file_fdw;
```

然后创建一个外部服务器：

```
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;
```

现在就可以创建外部表了。使用 CREATE FOREIGN TABLE 命令时，需要定义该表的列、CSV 文件名及其格式：

```
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text
) SERVER pglog
OPTIONS ( filename '/home/josh/data/log/pglog.csv', format 'csv' );
```

到这里就完成了 – 现在可以直接查询日志了。当然，在生产环境中，还需要定义某种方法来处理日志轮转。

F.15. fuzzystrmatch

`fuzzystrmatch` 模块提供多个函数，用于确定字符串的相似性和距离。

> **小心**
>
> 目前，`soundex`、`metaphone`、`dmetaphone` 和 `dmetaphone_alt` 函数不能很好地处理多字节编码（例如 UTF-8）。

F.15.1. Soundex

Soundex 系统是一种通过将发音相近的名字转换为相同代码来进行匹配的方法。它最初在 1880 年、1900 年和 1910 年的美国人口普查中使用。注意，Soundex 对非英语名字并不是很有用。

`fuzzystrmatch` 模块提供两个用于处理 Soundex 代码的函数：

```
soundex(text) returns text
difference(text, text) returns int
```

`soundex` 函数将字符串转换为其 Soundex 代码。`difference` 函数将两个字符串转换为各自的 Soundex 代码，然后返回代码中相同位置上匹配的个数。由于 Soundex 代码有四个字符，因此结果范围为 0 到 4，其中 0 表示没有匹配，4 表示完全匹配。（因此，这个函数的命名并不贴切，`similarity` 会是更好的名字。）

下面是一些用法示例：

```
SELECT soundex('hello world!');

SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.15.2. Levenshtein

该函数计算两个字符串之间的 Levenshtein 距离：

```
levenshtein(text source, text target, int ins_cost, int del_cost, int sub_cost) returns int
levenshtein(text source, text target) returns int
levenshtein_less_equal(text source, text target, int ins_cost, int del_cost, int sub_cost, int
 max_d) returns int
levenshtein_less_equal(text source, text target, int max_d) returns int
```

source 和 target 都可以是任意非 NULL 字符串，最大长度为 255 个字符。代价参数分别指定字符插入、删除或替换的代价。可以像该函数的第二种形式那样省略代价参数；此时它们都默认为 1。

levenshtein_less_equal 是 Levenshtein 函数的加速版本，用于只关心较小距离的场景。如果实际距离小于等于 max_d，那么 levenshtein_less_equal 会返回正确的距离；否则它会返回某个大于 max_d 的值。如果 max_d 为负，则其行为与 levenshtein 相同。

示例：

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
-------------
           2
(1 row)

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2, 1, 1);
 levenshtein
-------------
           3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive', 2);
 levenshtein_less_equal
------------------------
                      3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive', 4);
 levenshtein_less_equal
------------------------
                      4
(1 row)
```

F.15.3. Metaphone

Metaphone 与 Soundex 一样，基于为输入字符串构造一个代表性代码的思想。如果两个字符串具有相同的代码，则认为它们相似。

该函数计算输入字符串的 Metaphone 代码：

```
metaphone(text source, int max_output_length) returns text
```

source 必须是非 NULL 字符串，最大长度为 255 个字符。max_output_length 设置输出 Metaphone 代码的最大长度；如果超过该长度，输出会被截断为该长度。

示例：

```
test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----------
 KM
(1 row)
```

F.15.4. Double Metaphone

Double Metaphone 系统会为给定输入字符串计算两个“近音”代码，即一个“主”代码和一个“备选”代码。在大多数情况下，它们相同，但尤其对于非英语名字，这两个代码可能会因发音不同而略有差异。这些函数分别计算主代码和备选代码：

```
dmetaphone(text source) returns text
dmetaphone_alt(text source) returns text
```

输入字符串没有长度限制。

示例：

```
test=# SELECT dmetaphone('gumbo');
 dmetaphone
------------
 KMP
(1 row)
```

F.16. hstore

此模块实现了 hstore 数据类型，用于在单个PostgreSQL 值中存储一组键/值对。这在多种场景中可能很有用，例如包含许多很少查看的属性的行，或者半结构化数据。键和值都只是文本字符串。

F.16.1. hstore 外部表示

用于输入和输出的 hstore 文本表示包含零个或多个以逗号分隔的 *key* => *value* 对。一些示例：

```
k => v
foo => bar, baz => whatever
"1-a" => "anything at all"
```

键/值对的顺序并不重要（而且在输出时可能不会按原样重现）。键/值对之间或 => 号周围的空白会被忽略。包含空白、逗号、= 或 > 的键和值必须用双引号括起来。要在键或值中包含双引号或反斜线，请用反斜线转义。

每个 hstore 中的键都是唯一的。如果声明的 hstore 带有重复键，则在该hstore 中只会存储其中一个，而且无法保证保留的是哪一个：

```
SELECT 'a=>1,a=>2'::hstore;
  hstore
----------
 "a"=>"1"
```

值（但键不能）可以是 SQL NULL。例如：

```
key => NULL
```

NULL 关键字不区分大小写。若要将 NULL 视为普通字符串“NULL”，请用双引号括起来。

> **注意**
>
> 请注意，当hstore 文本格式用于输入时，它会在任何必需的加引号或转义之前应用。如果通过参数传递一个 hstore 字面量，则不需要额外处理。但如果将其作为带引号的字面量常量传递，那么其中的单引号字符以及（取决于 standard_conforming_strings 配置参数的设置）反斜线字符都需要被正确转义。关于字符串常量的处理，见第 4.1.2.1 节。

在输出时，即使严格来说并非必需，键和值也总是带有双引号。

F.16.2. hstore 操作符和函数

hstore 模块提供的操作符见表 F.7，函数见表 F.8。

表 F.7. hstore 操作符

操作符	描述	示例	结果
hstore -> text	获取键对应的值（如果不存在则为NULL）	'a=>x, b=>y'::hstore -> 'a'	x
hstore -> text[]	获取各键对应的值（如果不存在则为NULL）	'a=>x, b=>y, c=>z'::hstore -> ARRAY['c','a']	{"z","x"}
hstore || hstore	连接hstore 值	'a=>b, c=>d'::hstore || 'c=>x, d=>q'::hstore	"a"=>"b", "c"=>"x", "d"=>"q"
hstore ? text	hstore 是否包含该键?	'a=>1'::hstore ? 'a'	t
hstore ?& text[]	hstore 是否包含所有指定的键?	'a=>1,b=>2'::hstore ?& ARRAY['a','b']	t
hstore ?| text[]	hstore 是否包含指定键中的任意一个?	'a=>1,b=>2'::hstore ?| ARRAY['b','c']	t
hstore @> hstore	左操作数是否包含右操作数?	'a=>b, b=>1, c=>NULL'::hstore @> 'b=>1'	t
hstore <@ hstore	左操作数是否被包含在右操作数中?	'a=>c'::hstore <@ 'a=>b, b=>1, c=>NULL'	f
hstore - text	从左操作数中删除键	'a=>1, b=>2, c=>3'::hstore - 'b'::text	"a"=>"1", "c"=>"3"
hstore - text[]	从左操作数中删除键	'a=>1, b=>2, c=>3'::hstore - ARRAY['a','b']	"c"=>"3"
hstore - hstore	从左操作数中删除匹配的键值对	'a=>1, b=>2, c=>3'::hstore - 'a=>4, b=>2'::hstore	"a"=>"1", "c"=>"3"
record #= hstore	用hstore 中匹配的值替换record 中的字段	参见示例部分	
%% hstore	将hstore 转换为交替排列的键和值数组	%% 'a=>foo, b=>bar'::hstore	{a,foo,b,bar}
%# hstore	将hstore 转换为二维键/值数组	%# 'a=>foo, b=>bar'::hstore	{{a,foo},{b,bar}}

> **注意**
>
> 在 PostgreSQL 8.2 之前，包含操作符 @> 和 <@ 分别称为 @ 和 ~。这些名称仍然可用，但已弃用，最终将被删除。请注意，旧名称与核心几何数据类型以前采用的约定正好相反！

表 F.8. hstore 函数

函数	返回类型	描述	示例	结果
hstore(record)	hstore	从记录或行构造hstore	hstore(ROW(1, 2))	f1=>1,f2=>2
hstore(text[])	hstore	从数组构造hstore，数组可以是键/值数组，也可以是二维数组	hstore(ARRAY['a','1', 'b','2']) || hstore(ARRAY[['c','3'],['d','4']])	a=>1, b=>2, c=>3, d=>4
hstore(text[], text[])	hstore	从分开的键数组和值数组构造hstore	hstore(ARRAY['a','b'], ARRAY['1','2'])	"a"=>"1","b"=>"2"
hstore(text, text)	hstore	生成只含一个项的hstore	hstore('a', 'b')	"a"=>"b"
akeys(hstore)	text[]	将hstore的键作为数组获取	akeys('a=>1,b=>2')	{a,b}
skeys(hstore)	setof text	将hstore的键作为集合获取	skeys('a=>1,b=>2')	a b
avals(hstore)	text[]	将hstore的值作为数组获取	avals('a=>1,b=>2')	{1,2}
svals(hstore)	setof text	将hstore的值作为集合获取	svals('a=>1,b=>2')	1 2
hstore_to_array(hstore)	text[]	将hstore的键和值作为交替排列的键和值数组获取	hstore_to_array('a=>1,b=>2')	{a,1,b,2}
hstore_to_matrix(hstore)	text[]	将hstore的键和值作为二维数组获取	hstore_to_matrix('a=>1, b=>2')	{{a,1},{b,2}}
hstore_to_json(hstore)	json	将hstore作为json值获取，把所有非空值转换为JSON字符串	hstore_to_json('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}
hstore_to_jsonb(hstore)	jsonb	将hstore作为jsonb值获取，把所有非空值转换为JSON字符串	hstore_to_jsonb('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}

函数	返回类型	描述	示例	结果
hstore_to_json_loose(hstore)	json	将hstore 作为json 值获取，但尝试区分数值和布尔值，使它们在 JSON 中不带引号	hstore_to_json_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}
hstore_to_jsonb_loose(hstore)	jsonb	将hstore 作为jsonb 值获取，但尝试区分数值和布尔值，使它们在 JSON 中不带引号	hstore_to_jsonb_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}
slice(hstore, text[])	hstore	提取hstore 的子集	slice('a=>1, b=>2,c=>3'::hstore, ARRAY['b','c','x'])	"b"=>"2", "c"=>"3"
each(hstore)	setof(key text, value text)	将hstore 的键和值作为集合获取	select * from each('a=>1,b=>2')	key | value -----+------- a | 1 b | 2
exist(hstore, text)	boolean	hstore 是否包含该键?	exist('a=>1','a')	t
defined(hstore,text)	boolean	hstore 是否包含该键的非NULL 值?	defined('a=>NULL','a')	f
delete(hstore, text)	hstore	删除键匹配的键值对	delete('a=>1, b=>2','b')	"a"=>"1"
delete(hstore, text[])	hstore	删除键匹配的键值对	delete('a=>1, b=>2,c=>3', ARRAY['a','b'])	"c"=>"3"
delete(hstore, hstore)	hstore	删除与第二个参数中的键值对匹配的键值对	delete('a=>1, b=>2','a=>4,b=>2'::hstore)	"a"=>"1"
populate_record(record, hstore)	record	用hstore 中匹配的值替换record 中的字段	参见示例部分	

注意

将 hstore 值转换为 json 时使用函数 hstore_to_json。同样，将hstore 值转换为 jsonb 时使用 hstore_to_jsonb。

> **注意**
>
> `populate_record` 函数的第一个参数实际声明为 `anyelement`，而不是`record`，但它会在运行时出错并拒绝非记录类型。

F.16.3. 索引

`hstore` 支持针对`@>`、`?`、`?&` 和`?|`操作符的 GiST 和 GIN 索引。例如：

```
CREATE INDEX hidx ON testhstore USING GIST (h);

CREATE INDEX hidx ON testhstore USING GIN (h);
```

`hstore` 也支持用于`=`操作符的 `btree` 或 `hash` 索引。这允许 `hstore` 列被声明为 `UNIQUE`，或者用于`GROUP BY`、`ORDER BY` 或 `DISTINCT` 表达式。`hstore` 值的排序顺序本身并没有特别实用的意义，但这些索引可能适合用于等值查找。可按如下方式为`=`比较创建索引：

```
CREATE INDEX hidx ON testhstore USING BTREE (h);

CREATE INDEX hidx ON testhstore USING HASH (h);
```

F.16.4. 示例

添加一个键，或用新值更新现有键：

```
UPDATE tab SET h = h || hstore('c', '3');
```

删除一个键：

```
UPDATE tab SET h = delete(h, 'k1');
```

将 `record` 转换为 `hstore`：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT hstore(t) FROM test AS t;
                   hstore
---------------------------------------------
 "col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

将 hstore 转换为预定义的 record 类型：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                              '"col1"=>"456", "col2"=>"zzz"');
 col1 | col2 | col3
------+------+------
  456 | zzz  |
(1 row)
```

使用 hstore 中的值修改现有记录：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
 col1 | col2 | col3
------+------+------
  123 | foo  | baz
(1 row)
```

F.16.5. 统计信息

由于 hstore 类型本身比较宽松，它可能包含大量不同的键。检查键是否合法是应用程序的任务。下面的示例展示了检查键并获取统计信息的几种技术。

简单示例：

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

使用表：

```
SELECT (each(h)).key, (each(h)).value INTO stat FROM testhstore;
```

在线统计信息：

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
  GROUP BY key
  ORDER BY count DESC, key;
    key    | count
-----------+-------
 line      |   883
 query     |   207
 pos       |   203
 node      |   202
 space     |   197
 status    |   195
 public    |   194
 title     |   190
 org       |   189
.................
```

F.16.6. 兼容性

自 PostgreSQL 9.0 起，`hstore` 使用了与更早版本不同的内部表示。这不会妨碍转储/恢复升级，因为文本表示（即转储中使用的表示）没有改变。

在进行二进制升级时，通过让新代码识别旧格式数据，维持了向上兼容性。这会在处理尚未被新代码修改过的数据时带来轻微的性能损失。可以通过执行如下 `UPDATE` 语句，强制升级表列中的所有值：

```
UPDATE tablename SET hstorecol = hstorecol || '';
```

另一种方式是：

```
ALTER TABLE tablename ALTER hstorecol TYPE hstore USING hstorecol || '';
```

使用 `ALTER TABLE` 方法需要对表加 `ACCESS EXCLUSIVE` 锁，但不会因旧行版本而导致表膨胀。

F.16.7. 转换

另外还有一些扩展可用，它们为 PL/Perl 和 PL/Python 语言中的 `hstore` 类型实现了转换。PL/Perl 的扩展分别名为 `hstore_plperl` 和 `hstore_plperlu`，对应受信任的和不受信任的 PL/Perl。如果安装这些转换并在创建函数时指定它们，则`hstore` 值会映射为 Perl hash。PL/Python 的扩展分别名为 `hstore_plpythonu`、`hstore_plpython2u` 和 `hstore_plpython3u`（关于 PL/Python 的命名约定请见第 46.1 节）。如果使用它们，`hstore` 值会映射为 Python 字典。

> **小心**
>
> 强烈建议将转换扩展安装在与 `hstore` 相同的模式中。否则，如果转换扩展所在模式包含由恶意用户定义的对象，在安装时会存在安全隐患。

F.16.8. 作者

Oleg Bartunov <oleg@sai.msu.su>，俄罗斯莫斯科，莫斯科大学

Teodor Sigaev <teodor@sigaev.ru>，俄罗斯莫斯科，Delta-Soft Ltd.

Andrew Gierth <andrew@tao11.riddles.org.uk>，英国，对本模块作了额外增强

F.17. intagg

intagg 模块提供一个整数聚合器和一个枚举器。由于已经有内置函数提供了其能力的超集，所以 intagg 现已过时。不过，该模块仍作为这些内置函数的兼容性包装器提供。

F.17.1. 函数

聚合器是聚合函数 int_array_aggregate(integer)，它会生成一个整数数组，其中恰好包含输入给它的那些整数。这是对array_agg 的包装器，后者对任意数组类型都能做同样的事情。

枚举器是函数 int_array_enum(integer[])，它返回setof integer。本质上，它是聚合器的逆操作：给定一个整数数组，将其展开为一组行。这是对unnest 的包装器，后者对任意数组类型都能做同样的事情。

F.17.2. 使用示例

许多数据库系统有一对多表的概念。这种表通常位于两个带索引的表之间，例如：

```
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT REFERENCES right);
```

它通常按如下方式使用：

```
SELECT right.* from right JOIN one_to_many ON (right.id = one_to_many.right)
  WHERE one_to_many.left = item;
```

这将返回左侧表中某个条目在右侧表中对应的所有项。这是 SQL 中非常常见的一种构造。

不过，如果one_to_many 表中的条目非常多，这种方法就会变得相当繁琐。通常，对于左侧表中的某个特定条目，右侧表中的每个对应项都要执行一次索引扫描并取回一行。如果你的系统非常动态，那就没有太多办法。不过，如果有一部分数据相当静态，可以借助聚合器创建一个汇总表。

```
CREATE TABLE summary AS
  SELECT left, int_array_aggregate(right) AS right
  FROM one_to_many
  GROUP BY left;
```

这样会创建一个表，其中左侧每个条目对应一行，并附带一个由右侧各项组成的数组。不过，如果没有某种使用该数组的方法，它就几乎没有用处；这正是数组枚举器存在的原因。你可以这样做：

```
SELECT left, int_array_enum(right) FROM summary WHERE left = item;
```

上面这个使用 int_array_enum 的查询，产生的结果与下面的查询相同：

```
SELECT left, right FROM one_to_many WHERE left = item;
```

区别在于，针对汇总表的查询只需要从表中取出一行，而直接查询one_to_many 则必须对每个对应项都进行索引扫描并取回一行。

在某个系统上，EXPLAIN 显示，某个查询的代价从 8488 降到了 329。原始查询是一个涉及 one_to_many 表的连接，后来被替换为：

```
SELECT right, count(right) FROM
  ( SELECT left, int_array_enum(right) AS right
    FROM summary JOIN (SELECT left FROM left_table WHERE left = item) AS lefts
         ON (summary.left = lefts.left)
  ) AS list
  GROUP BY right
  ORDER BY count DESC;
```

F.18. intarray

intarray 模块提供了一些有用的函数和操作符，用于操作不含 NULL 的整数数组。它还支持使用其中某些操作符进行索引搜索。

如果所提供的数组中包含任何 NULL 元素，所有这些操作都会抛出错误。

这些操作中有许多只对一维数组才有意义。尽管它们也接受更高维度的输入数组，但数据会被视为按存储顺序排列的线性数组。

F.18.1. intarray 函数和操作符

intarray 模块提供的函数列在表 F.9 中，操作符列在表 F.10 中。

表 F.9. intarray 函数

函数	返回类型	描述	示例	结果
icount(int[])	int	数组中的元素个数	icount('{1,2,3}'::int[])	3
sort(int[], text dir)	int[]	对数组排序 – *dir* 必须为asc 或desc	sort('{1,2,3}'::int[], 'desc')	{3,2,1}
sort(int[])	int[]	按升序排序	sort(array[11,77,44])	{11,44,77}
sort_asc(int[])	int[]	按升序排序		
sort_desc(int[])	int[]	按降序排序		
uniq(int[])	int[]	删除相邻的重复元素	uniq(sort('{1,2,3,2,1}'::int[]))	{1,2,3}
idx(int[], int item)	int	与*item* 匹配的第一个元素的索引（如果没有则为 0）	idx(array[11,22,33,22,11], 22)	2
subarray(int[], int start, int len)	int[]	从位置*start* 开始的数组部分，共 *len* 个元素	subarray('{1,2,3,2,1}'::int[], 2, 3)	{2,3,2}
subarray(int[], int start)	int[]	从位置*start* 开始的数组部分	subarray('{1,2,3,2,1}'::int[], 2)	{2,3,2,1}
intset(int)	int[]	创建一个只含一个元素的数组	intset(42)	{42}

表 F.10. intarray 操作符

操作符	返回值	描述
int[] && int[]	boolean	重叠 – 如果数组至少有一个共同元素，则为true
int[] @> int[]	boolean	包含 – 如果左数组包含右数组，则为true

操作符	返回值	描述
int[] <@ int[]	boolean	包含于 – 如果左数组包含在右数组中，则为true
# int[]	int	数组中的元素个数
int[] # int	int	索引（与idx 函数相同）
int[] + int	int[]	将元素压入数组（添加到数组末尾）
int[] + int[]	int[]	数组连接（右数组添加到左数组末尾）
int[] - int	int[]	从数组中移除与右参数匹配的项
int[] - int[]	int[]	从左数组中移除右数组中的元素
int[] | int	int[]	参数的并集
int[] | int[]	int[]	数组的并集
int[] & int[]	int[]	数组的交集
int[] @@ query_int	boolean	如果数组满足查询（见下文），则为true
query_int ~~ int[]	boolean	如果数组满足查询（@@的交换操作符），则为true

（在 PostgreSQL 8.2 之前，包含操作符 @> 和 <@ 分别称为 @ 和 ~。这些名称仍然可用，但已弃用，最终将被删除。请注意，旧名称与核心几何数据类型以前采用的约定正好相反！）

操作符&&、@> 和 <@等价于 PostgreSQL 内置的同名操作符，不同之处在于它们只适用于不包含 NULL 的整数数组，而内置操作符适用于任何数组类型。这一限制使它们在很多情况下比内置操作符更快。

@@和~~操作符用于测试数组是否满足某个查询，该查询表示为专用数据类型query_int 的一个值。一个查询由若干整数值组成，这些值会与数组元素进行检查，并且可通过操作符& （AND）、| （OR）和! （NOT）组合起来。必要时可以使用括号。例如，查询1&(2|3) 可匹配包含 1 且还包含 2 或 3 之一的数组。

F.18.2. 索引支持

intarray 为&&、@>、<@和@@操作符以及常规数组相等运算提供索引支持。

提供了两个 GiST 索引操作符类：gist__int_ops（默认使用）适用于小型到中型数据集，而 gist__intbig_ops 使用更大的签名，更适合为大型数据集建立索引（即包含大量不同数组值的列）。实现使用带有内置有损压缩的 RD 树数据结构。

另有一个非默认的 GIN 操作符类 gin__int_ops，支持相同的操作符。

在 GiST 和 GIN 索引之间如何选择，取决于二者的相对性能特征，相关讨论见其他部分。

F.18.3. 示例

```
-- a message can be in one or more “sections”
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- create specialized index
CREATE INDEX message_rdtree_idx ON message USING GIST (sections gist__int_ops);

-- select messages in section 1 OR 2 - OVERLAP operator
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- select messages in sections 1 AND 2 - CONTAINS operator
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';

-- the same, using QUERY operator
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;
```

F.18.4. 基准测试

源代码目录 `contrib/intarray/bench` 包含一个基准测试套件，可以针对一个已安装的 PostgreSQL 服务器运行。（还要求已安装 `DBD::Pg`。）运行方式如下：

```
cd .../contrib/intarray/bench
createdb TEST
psql -c "CREATE EXTENSION intarray" TEST
./create_test.pl | psql TEST
./bench.pl
```

`bench.pl` 脚本有很多选项，在不带任何参数运行时会显示这些选项。

F.18.5. 作者

所有工作都由 Teodor Sigaev（`<teodor@sigaev.ru>`）和 Oleg Bartunov（`<oleg@sai.msu.su>`）完成。更多信息请见 http://www.sai.msu.su/~megera/postgres/gist/。Andrey Oktyabrski 在添加新函数和操作方面也做出了很大贡献。

F.19. isn

isn 模块为以下国际产品编号标准提供数据类型：EAN13、UPC、ISBN（图书）、ISMN（音乐）和 ISSN（连续出版物）。输入这些编号时，会依据一份硬编码的前缀列表进行校验；输出时，这份前缀列表也会用于给编号加上连字符。由于新的前缀会不时被分配，这份前缀列表可能已经过时。希望该模块未来版本能从一个或多个表中获取前缀列表，以便用户按需更新；但目前，这份列表只能通过修改源代码并重新编译来更新。或者，该模块未来版本也可能取消前缀校验和连字符支持。

F.19.1. 数据类型

表 F.11 显示了 isn 模块提供的数据类型。

表 F.11. isn 数据类型

数据类型	描述
EAN13	欧洲商品编号，始终以 EAN13 显示格式显示
ISBN13	以新的 EAN13 显示格式显示的国际标准图书编号
ISMN13	以新的 EAN13 显示格式显示的国际标准音乐编号
ISSN13	以新的 EAN13 显示格式显示的国际标准连续出版物编号
ISBN	以旧的短显示格式显示的国际标准图书编号
ISMN	以旧的短显示格式显示的国际标准音乐编号
ISSN	以旧的短显示格式显示的国际标准连续出版物编号
UPC	通用产品代码

几点说明：

1. ISBN13、ISMN13 和 ISSN13 编号都是 EAN13 编号。
2. EAN13 编号并不总是 ISBN13、ISMN13 或 ISSN13（不过其中有些确实是）。
3. 某些 ISBN13 编号可以显示为 ISBN。
4. 某些 ISMN13 编号可以显示为 ISMN。
5. 某些 ISSN13 编号可以显示为 ISSN。
6. UPC 编号是 EAN13 编号的一个子集（它们基本上就是去掉首位 0 的 EAN13 编号）。
7. 所有 UPC、ISBN、ISMN 和 ISSN 编号都可以表示为 EAN13 编号。

在内部，所有这些类型都使用同一种表示形式（一个 64 位整数），并且彼此可以互换。之所以提供多种类型，是为了控制显示格式，并对原本应表示某一特定类型编号的输入执行更严格的有效性检查。

只要可能，ISBN、ISMN 和 ISSN 类型都会显示编号的短版本（ISxN 10）；对于不能用短版本表示的编号，则显示为 ISxN 13 格式。EAN13、ISBN13、ISMN13 和 ISSN13 类型则始终显示 ISxN 的长版本（EAN13）。

F.19.2. 类型转换

isn 模块提供以下几对类型转换：

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

当从 EAN13 转换到其他类型时，会在运行时检查该值是否落在另一类型的取值范围内；如果不在，就会抛出错误。其他类型转换都只是简单地重新标记，因此总会成功。

F.19.3. 函数和操作符

isn 模块提供标准比较操作符，以及对所有这些数据类型的 B-树和 hash 索引支持。此外，它还提供了一些专用函数，如表 F.12 所示。在该表中，isn 表示该模块提供的任意一种数据类型。

表 F.12. isn 函数

函数	返回值	描述
isn_weak(boolean)	boolean	设置弱输入模式（返回新的设置值）
isn_weak()	boolean	获取弱模式的当前状态
make_valid(isn)	isn	使无效编号变为有效（清除无效标记）
is_valid(isn)	boolean	检查是否存在无效标记

弱模式用于允许向表中插入无效数据。这里的“无效”指的是校验位错误，而不是缺少数字。

为什么会需要使用弱模式呢？例如，手头可能有一大批 ISBN 编号，数量多到难免会有一些编号因为某些奇怪的原因而带有错误的校验位（也许这些编号是从印刷清单扫描得到的，而 OCR 把数字识别错了；也许这些编号是人工录入的……谁知道呢）。总之，可能想把这些混乱情况清理干净，但同时仍希望先把所有编号都装入数据库，并借助外部工具在数据库中定位无效编号，以便核对信息并更容易完成校验；例如，可能会想把表中所有无效编号都查询出来。

当在弱模式下向表中插入无效编号时，实际插入的是校验位已更正的编号，但显示时会在末尾附加一个感叹号（!），例如0-11-000322-5!。可以用 is_valid 函数检查这个无效标记，并用 make_valid 函数清除它。

即使未启用弱模式，也可以通过在编号末尾附加!字符来强制插入无效编号。

另一个特殊功能是，在输入时可以用?代替校验位，系统会自动插入正确的校验位。

F.19.4. 示例

```
--Using the types directly:
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');

--Casting types:
-- note that you can only cast from ean13 to another type when the
-- number would be valid in the realm of the target type;
-- thus, the following will NOT work: select isbn(ean13('0220356483481'));
-- but these will:
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));

--Create a table with a single column to hold ISBN numbers:
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');

--Automatically calculate check digits (observe the '?'):
INSERT INTO test VALUES('220500896?');
INSERT INTO test VALUES('978055215372?');

SELECT issn('3251231?');
SELECT ismn('979047213542?');

--Using the weak mode:
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;

SELECT isbn13(id) FROM test;
```

F.19.5. 参考文献

实现该模块所需的信息收集自若干网站，包括：

- https://www.isbn-international.org/
- https://www.issn.org/
- https://www.ismn-international.org/
- https://www.wikipedia.org/

用于进行连字符分隔的前缀还整理自：

- https://www.gs1.org/standards/id-keys

- https://en.wikipedia.org/wiki/List_of_ISBN_identifier_groups
- https://www.isbn-international.org/content/isbn-users-manual
- https://en.wikipedia.org/wiki/International_Standard_Music_Number
- https://www.ismn-international.org/ranges.html

在创建这些算法时已十分谨慎，并依据官方 ISBN、ISMN、ISSN 用户手册中建议的算法进行了细致核验。

F.19.6. 作者

Germán Méndez Bravo (Kronuz), 2004 - 2006

该模块受到了 Garrett A. Wollman 的 `isbn_issn` 代码的启发。

F.20. lo

lo 模块为管理大对象（也称为 LO 或 BLOB）提供支持，其中包括数据类型lo 和触发器 lo_manage。

F.20.1. 原理

JDBC 驱动的一个问题是（ODBC 驱动也受此影响），其规范假定对 BLOB（二进制大对象）的引用存储在表内，并且如果该表项被更改，相关的 BLOB 就会从数据库中删除。

但在目前的 PostgreSQL 中并非如此。大对象被视为独立对象；表中的一个项可以通过 OID 引用某个大对象，但也可能有多个表项引用同一个大对象 OID，因此系统不会仅因你更改或删除了其中一个表项就删除该大对象。

这对于 PostgreSQL 专用应用没有问题，但使用 JDBC 或 ODBC 的标准代码不会删除这些对象，从而产生孤立对象，也就是不再被任何内容引用、只是占用磁盘空间的对象。

lo 模块允许通过把触发器附加到包含 LO 引用列的表上来解决这个问题。该触发器本质上就是在你删除或修改引用大对象的值时调用 lo_unlink。使用这个触发器时，你实际上是假定：凡是出现在受该触发器控制的列中的大对象，在数据库中都只有一个引用！

该模块还提供了数据类型 lo，它实际上只是oid 类型的一个域。这有助于区分保存大对象引用的数据库列与保存其他对象 OID 的列。使用该触发器并不要求必须使用 lo 类型，但用它来标识数据库中哪些列表示由该触发器管理的大对象，可能会更方便。还有传言说，如果 BLOB 列不用 lo，ODBC 驱动会感到困惑。

F.20.2. 如何使用

下面是一个简单的用法示例：

```
CREATE TABLE image (title text, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
    FOR EACH ROW EXECUTE FUNCTION lo_manage(raster);
```

对于每个将保存指向大对象的唯一引用的列，创建一个BEFORE UPDATE OR DELETE 触发器，并将该列名作为唯一的触发器参数。你也可以使用BEFORE UPDATE OF *column_name*，将该触发器限制为仅在更新该列时执行。如果同一张表中需要多个lo 列，就为每一列分别创建一个触发器，并记得为同一张表上的每个触发器指定不同的名称。

F.20.3. 限制

- 删除表时，其中包含的任何对象仍会变成孤立对象，因为这种情况下不会执行触发器。为避免这种情况，可以先执行DELETE FROM *table*，再执行DROP TABLE。

  TRUNCATE 也有同样的风险。

  如果你已经有，或者怀疑有，孤立的大对象，请参见vacuumlo 模块来帮助清理它们。偶尔运行 vacuumlo，作为对lo_manage 触发器的补充保障，是个不错的主意。

- 某些前端可能会创建自己的表，但不会创建相应的触发器。此外，用户也可能不记得（或根本不知道）要创建这些触发器。

F.20.4. 作者

Peter Mount <peter@retep.org.uk>

F.21. ltree

该模块实现了数据类型 `ltree`，用于表示存储在层次化树状结构中的数据标签。它还提供了丰富的标签树搜索能力。

F.21.1. 定义

标签是由字母数字字符和下划线组成的序列（例如，在 C 区域设置下，允许的字符为 `A-Za-z0-9_`）。标签长度必须少于 256 个字符。

示例：`42`、`Personal_Services`

标签路径是由点号分隔的零个或多个标签组成的序列，例如 `L1.L2.L3`，表示从层次树根节点到某个特定节点的一条路径。标签路径的长度不能超过 65535 个标签。

示例：`Top.Countries.Europe.Russia`

`ltree` 模块提供了几种数据类型：

- `ltree` 存储一个标签路径。
- `lquery` 表示一种用于匹配 `ltree` 值的、类似正则表达式的模式。一个简单单词会匹配路径中的相应标签。星号（`*`）匹配零个或多个标签。例如：

`foo`	精确匹配标签路径 *foo*
`*.foo.*`	匹配任何包含标签 *foo* 的标签路径
`*.foo`	匹配最后一个标签为 *foo* 的任意标签路径

  星号还可以带量词，以限制它们能够匹配的标签数量：

`*{`*n*`}`	精确匹配 *n* 个标签
`*{`*n*`,}`	至少匹配 *n* 个标签
`*{`*n*`,`*m*`}`	至少匹配 *n* 个、但不超过 *m* 个标签
`*{,`*m*`}`	至多匹配 *m* 个标签 — 与下式相同： `*{0,`*m*`}`

  在 `lquery` 中，有几个修饰符可以放在非星号标签的末尾，使其不只匹配完全相同的标签：

`@`	不区分大小写地匹配，例如 *a@* 可匹配 *A*
`*`	匹配以此前缀开头的任意标签，例如 *foo** 可匹配 *foobar*
`%`	匹配标签起始处由下划线分隔的单词

  修饰符`%` 的行为稍微复杂一些。它尝试匹配单词，而不是整个标签。例如，`foo_bar%` 可匹配 `foo_bar_baz`，但不能匹配`foo_barbaz`。如果与`*` 组合使用，则前缀匹配会分别作用于每个单词，例如`foo_bar%*` 可匹配 `foo1_bar2_baz`，但不能匹配`foo1_br2_baz`。

  此外，还可以写出多个可能带修饰符的非星号项，并用`|`（OR）分隔，以匹配其中任意一项；也可以在非星号组前加上`!`（NOT），以匹配不符合这些备选项中任意一项的标签。

下面是一个带注释的 lquery 示例：

```
Top.*{0,2}.sport*@.!football|tennis.Russ*|Spain
a.  b.     c.      d.               e.
```

这个查询将匹配满足以下条件的任意标签路径：

a. 以标签 Top 开头

b. 接下来，在下一个条件之前有零到两个标签

c. 然后是一个以前缀 sport 开头的标签，且匹配时不区分大小写

d. 接着有一个不匹配 football 或 tennis 的标签

e. 最后以一个以 Russ 开头的标签，或精确匹配Spain 的标签结束。

- ltxtquery 表示一种用于匹配 ltree 值的、类似全文检索的模式。一个ltxtquery 值包含单词，末尾还可以带有修饰符@、*、%；这些修饰符与它们在 lquery 中的含义相同。单词可以通过&（AND）、|（OR）、!（NOT）以及圆括号组合。它与 lquery 的关键区别在于，ltxtquery 匹配单词时不考虑它们在标签路径中的位置。

  下面是一个 ltxtquery 示例：

  ```
  Europe & Russia*@ & !Transportation
  ```

  它将匹配包含标签 Europe 以及任意以 Russia 开头（不区分大小写）的标签的路径，但不匹配包含标签Transportation 的路径。这些单词在路径中的位置并不重要。另外，当使用% 时，该单词可以匹配标签中任意由下划线分隔的单词，而不考虑其位置。

注意：ltxtquery 允许在符号之间出现空白，而ltree 和 lquery 不允许。

F.21.2. 操作符和函数

类型 ltree 具有常见的比较操作符 =、<>、<、>、<=、>=。比较时采用树遍历顺序，其中节点的子节点按标签文本排序。此外，还提供了表 F.13 中所示的专用操作符。

表 F.13. ltree 操作符

操作符	返回值	描述
ltree @> ltree	boolean	左参数是否为右参数的祖先（或与之相等）？
ltree <@ ltree	boolean	左参数是否为右参数的后代（或与之相等）？
ltree ~ lquery	boolean	ltree 是否匹配lquery？
lquery ~ ltree	boolean	ltree 是否匹配lquery？
ltree ? lquery[]	boolean	ltree 是否匹配数组中的任意 lquery？
lquery[] ? ltree	boolean	ltree 是否匹配数组中的任意 lquery？
ltree @ ltxtquery	boolean	ltree 是否匹配ltxtquery？

操作符	返回值	描述
ltxtquery @ ltree	boolean	ltree 是否匹配ltxtquery?
ltree || ltree	ltree	连接ltree 路径
ltree || text	ltree	将文本转换为ltree 后再连接
text || ltree	ltree	将文本转换为ltree 后再连接
ltree[] @> ltree	boolean	数组是否包含ltree 的某个祖先?
ltree <@ ltree[]	boolean	数组是否包含ltree 的某个祖先?
ltree[] <@ ltree	boolean	数组是否包含ltree 的某个后代?
ltree @> ltree[]	boolean	数组是否包含ltree 的某个后代?
ltree[] ~ lquery	boolean	数组是否包含匹配lquery 的任意路径?
lquery ~ ltree[]	boolean	数组是否包含匹配lquery 的任意路径?
ltree[] ? lquery[]	boolean	ltree 数组是否包含匹配任意lquery 的路径?
lquery[] ? ltree[]	boolean	ltree 数组是否包含匹配任意lquery 的路径?
ltree[] @ ltxtquery	boolean	数组是否包含匹配ltxtquery 的任意路径?
ltxtquery @ ltree[]	boolean	数组是否包含匹配ltxtquery 的任意路径?
ltree[] ?@> ltree	ltree	返回数组中第一个是ltree 祖先的项，如果没有则返回 NULL
ltree[] ?<@ ltree	ltree	返回数组中第一个是ltree 后代的项，如果没有则返回 NULL
ltree[] ?~ lquery	ltree	返回数组中第一个匹配lquery 的项，如果没有则返回 NULL
ltree[] ?@ ltxtquery	ltree	返回数组中第一个匹配ltxtquery 的项，如果没有则返回 NULL

操作符<@、@>、@和~都有对应的 ^<@、^@>、^@、^~变体，它们除了不使用索引之外完全相同。这些变体仅对测试有用。

可用函数见表 F.14。

表 F.14. ltree 函数

函数	返回类型	描述	示例	结果
subltree(ltree, int start, int end)	ltree	从位置*start* 到位置*end*-1 的ltree 子路径（从 0 开始计数）	subltree('Top.Child1. Child2',1,2)	Child1

函数	返回类型	描述	示例	结果
subpath(ltree, int offset, int len)	ltree	从位置*offset*开始、长度为*len*的ltree子路径。如果*offset*为负，则子路径从距路径末尾 -offset 个标签处开始。如果*len*为负，则从路径末尾省去那么多个标签。	subpath('Top.Child1.Child2',0,2)	Top.Child1
subpath(ltree, int offset)	ltree	从位置*offset*开始、一直延伸到路径末尾的ltree子路径。如果*offset*为负，则子路径从距路径末尾 -offset 个标签处开始。	subpath('Top.Child1.Child2',1)	Child1.Child2
nlevel(ltree)	integer	路径中的标签数	nlevel('Top.Child1.Child2')	3
index(ltree a, ltree b)	integer	*a*中首次出现*b*的位置；如果未找到则为 -1	index('0.1.2.3.5.4.5.6.8.5.6.8','5.6')	6
index(ltree a, ltree b, int offset)	integer	从*offset*开始搜索时，*a*中首次出现*b*的位置；负的*offset*表示从路径末尾向前*-offset*个标签处开始	index('0.1.2.3.5.4.5.6.8.5.6.8','5.6',-4)	9
text2ltree(text)	ltree	将text转换为ltree		
ltree2text(ltree)	text	将ltree转换为text		
lca(ltree, ltree, ...)	ltree	路径的最长公共祖先（最多支持 8 个参数）	lca('1.2.3','1.2.3.4.5.6')	1.2
lca(ltree[])	ltree	数组中各路径的最长公共祖先	lca(array['1.2.3'::ltree,'1.2.3.4'])	1.2

F.21.3. 索引

ltree支持几种能够加速所示操作符的索引类型：

- ltree上的 B-树索引：<、<=、=、>=、>
- ltree上的 GiST 索引：<、<=、=、>=、>、@>、<@、@、~、?

创建此类索引的示例：

```
CREATE INDEX path_gist_idx ON test USING GIST (path);
```

- `ltree[]` 上的 GiST 索引：`ltree[] <@ ltree`、`ltree @> ltree[]`、`@`、`~`、`?`

  创建此类索引的示例：

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (array_path);
  ```

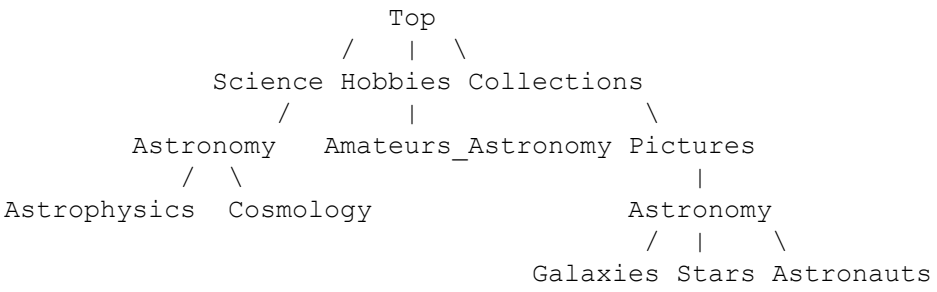
  注意：这种索引类型是有损的。

F.21.4. 示例

本示例使用下列数据（在源代码发行包中的 `contrib/ltree/ltreetest.sql` 文件里也能找到）：

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Stars');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING GIST (path);
CREATE INDEX path_idx ON test USING BTREE (path);
```

现在，我们有一个表`test`，其中的数据描述了下图所示的层次结构：



我们可以做继承查询：

```
ltreetest=> SELECT path FROM test WHERE path <@ 'Top.Science';
                path
------------------------------------
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(4 rows)
```

下面是一些路径匹配的示例：

```
ltreetest=> SELECT path FROM test WHERE path ~ '*.Astronomy.*';
                     path
-----------------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)

ltreetest=> SELECT path FROM test WHERE path ~ '*.!pictures@.*.Astronomy.*';
               path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

下面是一些全文检索示例：

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro*% & !pictures@';
                path
------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Hobbies.Amateurs_Astronomy
(4 rows)

ltreetest=> SELECT path FROM test WHERE path @ 'Astro* & !pictures@';
                path
------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

使用函数构造路径：

```
ltreetest=> SELECT subpath(path,0,2)||'Space'||subpath(path,2) FROM test WHERE path <@ 'Top.
Science.Astronomy';
                 ?column?
------------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

可以通过创建一个 SQL 函数，在路径的指定位置插入标签来简化这一操作：

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
    AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
    LANGUAGE SQL IMMUTABLE;

ltreetest=> SELECT ins_label(path,2,'Space') FROM test WHERE path <@ 'Top.Science.Astronomy';
               ins_label
------------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

F.21.5. 转换

有额外的扩展实现了 PL/Python 的 `ltree` 类型转换。这些扩展分别叫做 `ltree_plpythonu`、`ltree_plpython2u` 和 `ltree_plpython3u`（关于 PL/Python 的命名约定请见第 46.1 节）。如果安装了这些转换扩展，并在创建函数时指定它们，则`ltree` 值会映射为 Python 列表。（不过，目前还不支持反向映射。）

> **小心**
>
> 强烈建议将转换扩展安装在与 `ltree` 相同的模式中。否则，如果转换扩展所在模式包含由恶意用户定义的对象，在安装时会存在安全隐患。

F.21.6. 作者

全部工作均由 Teodor Sigaev（<teodor@stack.net>）和 Oleg Bartunov（<oleg@sai.msu.su>）完成。更多信息见 http://www.sai.msu.su/~megera/postgres/gist/。作者谨感谢 Eugeny Rodichev 的有益讨论。欢迎提出意见和缺陷报告。

F.22. pageinspect

pageinspect 模块提供了一组函数，允许你在底层检查数据库页的内容，这对调试很有帮助。所有这些函数都只能由超级用户使用。

F.22.1. 通用函数

```
get_raw_page(relname text, fork text, blkno int) returns bytea
```

get_raw_page 读取指定关系的指定块，并以bytea 值的形式返回其副本。这样就能获得该块在单一时刻的一致副本。*fork* 应当为'main'（主数据分支）、'fsm'（空闲空间映射）、'vm'（可见性映射）或'init'（初始化分支）。

```
get_raw_page(relname text, blkno int) returns bytea
```

这是 get_raw_page 的简写形式，用于从主分支读取。等价于 get_raw_page(relname, 'main', blkno)

```
page_header(page bytea) returns record
```

page_header 显示 PostgreSQL 所有堆页和索引页共有的字段。

应将用 get_raw_page 获得的页面映像作为参数传入。例如：

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
    lsn    | checksum | flags  | lower | upper | special | pagesize | version | prune_xid
-----------+----------+--------+-------+-------+---------+----------+---------+-----------
 0/24A1B50 |        0 |      1 |   232 |   368 |    8192 |     8192 |       4 |         0
```

返回的列对应于 PageHeaderData 结构体中的字段。详细信息见 src/include/storage/bufpage.h。

checksum 字段是页面中存储的校验和；如果页面以某种方式损坏，它可能不正确。如果本实例未启用数据校验和，则所存储的值没有意义。

```
page_checksum(page bytea, blkno int4) returns smallint
```

page_checksum 计算该页面在位于给定块号时的校验和。

应将用 get_raw_page 获得的页面映像作为参数传入。例如：

```
test=# SELECT page_checksum(get_raw_page('pg_class', 0), 0);
 page_checksum
---------------
         13443
```

注意，校验和依赖于块号，因此应传入匹配的块号（除非是在做某些特殊的调试工作）。

可以将此函数计算出的校验和与 checksum 结果字段进行比较，该字段来自函数 page_header。如果本实例启用了数据校验和，那么这两个值应当相等。

`fsm_page_contents(page bytea) returns text`

`fsm_page_contents` 显示 FSM 页的内部节点结构。例如：

```
test=# SELECT fsm_page_contents(get_raw_page('pg_class', 'fsm', 0));
```

输出是一个多行字符串，页内二叉树中的每个节点对应一行。只打印非零节点。还会打印所谓的 “next” 指针，它指向下一个将从该页返回的槽位。

参见 `src/backend/storage/freespace/README`，了解 FSM 页面结构的更多信息。

F.22.2. 堆函数

`heap_page_items(page bytea) returns setof record`

`heap_page_items` 显示堆页上的所有行指针。对于正在使用的行指针，还会显示元组头和元组原始数据。无论这些元组在复制原始页时是否对某个 MVCC 快照可见，都会显示出来。

应将用 `get_raw_page` 获得的堆页面映像作为参数传入。例如：

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));
```

有关返回字段的说明，参见`src/include/storage/itemid.h` 和 `src/include/access/htup_details.h`。

`tuple_data_split(rel_oid oid, t_data bytea, t_infomask integer, t_infomask2 integer, t_bits text [, do_detoast bool]) returns bytea[]`

`tuple_data_split` 以与后端内部相同的方式将元组数据拆分为属性。

```
test=# SELECT tuple_data_split('pg_class'::regclass, t_data, t_infomask, t_infomask2, t_bits) FROM heap_page_items(get_raw_page('pg_class', 0));
```

调用该函数时，应使用以下函数返回的属性作为参数：`heap_page_items`。

如果 *`do_detoast`* 为 `true`，则会按需对属性执行去 TOAST 化处理。默认值为`false`。

`heap_page_item_attrs(page bytea, rel_oid regclass [, do_detoast bool]) returns setof record`

`heap_page_item_attrs` 与 `heap_page_items` 等效，不同之处在于它将元组原始数据作为属性数组返回，并且可以通过*`do_detoast`* 选择是否对这些属性执行去 TOAST 化处理；该参数默认为 `false`。

应将用 `get_raw_page` 获得的堆页面映像作为参数传入。例如：

```
test=# SELECT * FROM heap_page_item_attrs(get_raw_page('pg_class', 0), 'pg_class'::regclass);
```

F.22.3. B-树函数

`bt_metap(relname text) returns record`

`bt_metap` 返回 B-树索引元页的信息。例如：

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-----------+-------
magic                   | 340322
version                 | 3
root                    | 1
level                   | 0
fastroot                | 1
fastlevel               | 0
oldest_xact             | 582
last_cleanup_num_tuples | 1000
```

`bt_page_stats(relname text, blkno int) returns record`

`bt_page_stats` 返回 B-树索引单个页面的概要信息。例如：

```
test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]-+-----
blkno         | 1
type          | l
live_items    | 256
dead_items    | 0
avg_item_size | 12
page_size     | 8192
free_size     | 4056
btpo_prev     | 0
btpo_next     | 0
btpo          | 0
btpo_flags    | 3
```

```
bt_page_items(relname text, blkno int) returns setof record
```

bt_page_items 返回 B-树索引页中所有项的详细信息。例如：

```
test=# SELECT * FROM bt_page_items('pg_cast_oid_index', 1);
 itemoffset |  ctid   | itemlen | nulls | vars |    data
------------+---------+---------+-------+------+-------------
          1 | (0,1)   |      12 | f     | f    | 23 27 00 00
          2 | (0,2)   |      12 | f     | f    | 24 27 00 00
          3 | (0,3)   |      12 | f     | f    | 25 27 00 00
          4 | (0,4)   |      12 | f     | f    | 26 27 00 00
          5 | (0,5)   |      12 | f     | f    | 27 27 00 00
          6 | (0,6)   |      12 | f     | f    | 28 27 00 00
          7 | (0,7)   |      12 | f     | f    | 29 27 00 00
          8 | (0,8)   |      12 | f     | f    | 2a 27 00 00
```

在 B-树叶页中，ctid 指向一个堆元组。在内部页中，ctid 的块号部分指向索引本身的另一个页面，而偏移量部分（第二个数字）被忽略，通常为 1。

请注意，任何非最右页（即btpo_next 字段值非零的页面）的第一项都是该页的“高键”，这意味着其data 充当该页上所有项的上界，而其ctid 字段没有意义。另外，在非叶子页上，第一个真正的数据项（第一个不是高键的项）是一个“负无穷”项，其data 字段中没有实际值。不过，这样的项在ctid 字段中确实有一个有效的下行链接。

```
bt_page_items(page bytea) returns setof record
```

也可以将页作为 bytea 值传给 bt_page_items。应将用 get_raw_page 获得的页面映像作为参数传入。因此，上一个示例也可以改写成下面这样：

```
test=# SELECT * FROM bt_page_items(get_raw_page('pg_cast_oid_index', 1));
 itemoffset |  ctid   | itemlen | nulls | vars |    data
------------+---------+---------+-------+------+-------------
          1 | (0,1)   |      12 | f     | f    | 23 27 00 00
          2 | (0,2)   |      12 | f     | f    | 24 27 00 00
          3 | (0,3)   |      12 | f     | f    | 25 27 00 00
          4 | (0,4)   |      12 | f     | f    | 26 27 00 00
          5 | (0,5)   |      12 | f     | f    | 27 27 00 00
          6 | (0,6)   |      12 | f     | f    | 28 27 00 00
          7 | (0,7)   |      12 | f     | f    | 29 27 00 00
          8 | (0,8)   |      12 | f     | f    | 2a 27 00 00
```

其余细节与上一项中的说明相同。

F.22.4. BRIN 函数

`brin_page_type(page bytea) returns text`

`brin_page_type` 返回给定 BRIN 索引页的页类型；如果该页不是有效的 BRIN 页，则抛出错误。例如：

```
test=# SELECT brin_page_type(get_raw_page('brinidx', 0));
 brin_page_type
----------------
 meta
```

`brin_metapage_info(page bytea) returns record`

`brin_metapage_info` 返回 BRIN 索引元页的各类信息。例如：

```
test=# SELECT * FROM brin_metapage_info(get_raw_page('brinidx', 0));
   magic    | version | pagesperrange | lastrevmappage
------------+---------+---------------+----------------
 0xA8109CFA |       1 |             4 |              2
```

`brin_revmap_data(page bytea) returns setof tid`

`brin_revmap_data` 返回 BRIN 索引范围映射页中的元组标识符列表。例如：

```
test=# SELECT * FROM brin_revmap_data(get_raw_page('brinidx', 2)) LIMIT 5;
  pages
---------
 (6,137)
 (6,138)
 (6,139)
 (6,140)
 (6,141)
```

`brin_page_items(page bytea, index oid) returns setof record`

`brin_page_items` 返回存储在 BRIN 数据页中的数据。例如：

```
test=# SELECT * FROM brin_page_items(get_raw_page('brinidx', 5),
                                     'brinidx')
       ORDER BY blknum, attnum LIMIT 6;
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |    value
------------+--------+--------+----------+----------+-------------+--------------
        137 |      0 |      1 | t        | f        | f           |
        137 |      0 |      2 | f        | f        | f           | {1 .. 88}
        138 |      4 |      1 | t        | f        | f           |
        138 |      4 |      2 | f        | f        | f           | {89 .. 176}
        139 |      8 |      1 | t        | f        | f           |
        139 |      8 |      2 | f        | f        | f           | {177 .. 264}
```

返回的列对应于 `BrinMemTuple` 和 `BrinValues` 结构体中的字段。详细信息见 `src/include/access/brin_tuple.h`。

F.22.5. GIN 函数

`gin_metapage_info(page bytea) returns record`

`gin_metapage_info` 返回 GIN 索引元页的信息。例如：

```
test=# SELECT * FROM gin_metapage_info(get_raw_page('gin_index', 0));
-[ RECORD 1 ]----+-----------
pending_head     | 4294967295
pending_tail     | 4294967295
tail_free_size   | 0
n_pending_pages  | 0
n_pending_tuples | 0
n_total_pages    | 7
n_entry_pages    | 6
n_data_pages     | 0
n_entries        | 693
version          | 2
```

`gin_page_opaque_info(page bytea) returns record`

`gin_page_opaque_info` 返回 GIN 索引不透明区域的信息，例如页类型。下面是一个示例：

```
test=# SELECT * FROM gin_page_opaque_info(get_raw_page('gin_index', 2));
 rightlink | maxoff |         flags
-----------+--------+-----------------------
         5 |      0 | {data,leaf,compressed}
(1 row)
```

`gin_leafpage_items(page bytea) returns setof record`

`gin_leafpage_items` 返回 GIN 叶页中所存数据的相关信息。例如：

```
test=# SELECT first_tid, nbytes, tids[0:5] AS some_tids
        FROM gin_leafpage_items(get_raw_page('gin_test_idx', 2));
 first_tid | nbytes |                          some_tids
-----------+--------+-------------------------------------------------------------
 (8,41)    |    244 | {"(8,41)","(8,43)","(8,44)","(8,45)","(8,46)"}
 (10,45)   |    248 | {"(10,45)","(10,46)","(10,47)","(10,48)","(10,49)"}
 (12,52)   |    248 | {"(12,52)","(12,53)","(12,54)","(12,55)","(12,56)"}
 (14,59)   |    320 | {"(14,59)","(14,60)","(14,61)","(14,62)","(14,63)"}
 (167,16)  |    376 | {"(167,16)","(167,17)","(167,18)","(167,19)","(167,20)"}
 (170,30)  |    376 | {"(170,30)","(170,31)","(170,32)","(170,33)","(170,34)"}
 (173,44)  |    197 | {"(173,44)","(173,45)","(173,46)","(173,47)","(173,48)"}
(7 rows)
```

F.22.6. Hash 函数

hash_page_type(page bytea) returns text

hash_page_type 返回给定 HASH 索引页的页类型。例如：

```
test=# SELECT hash_page_type(get_raw_page('con_hash_index', 0));
 hash_page_type
----------------
 metapage
```

hash_page_stats(page bytea) returns setof record

hash_page_stats 返回 HASH 索引桶页或溢出页的信息。例如：

```
test=# SELECT * FROM hash_page_stats(get_raw_page('con_hash_index', 1));
-[ RECORD 1 ]---+-----------
live_items      | 407
dead_items      | 0
page_size       | 8192
free_size       | 8
hasho_prevblkno | 4096
hasho_nextblkno | 8474
hasho_bucket    | 0
hasho_flag      | 66
hasho_page_id   | 65408
```

hash_page_items(page bytea) returns setof record

hash_page_items 返回 HASH 索引桶页或溢出页中所存数据的相关信息。例如：

```
test=# SELECT * FROM hash_page_items(get_raw_page('con_hash_index', 1)) LIMIT 5;
 itemoffset |   ctid    |    data
------------+-----------+------------
          1 | (899,77)  | 1053474816
          2 | (897,29)  | 1053474816
          3 | (894,207) | 1053474816
          4 | (892,159) | 1053474816
          5 | (890,111) | 1053474816
```

hash_bitmap_info(index oid, blkno int) returns record

hash_bitmap_info 显示位图页中某个 HASH 索引溢出页对应位的状态。例如：

```
test=# SELECT * FROM hash_bitmap_info('con_hash_index', 2052);
 bitmapblkno | bitmapbit | bitstatus
-------------+-----------+-----------
          65 |         3 | t
```

hash_metapage_info(page bytea) returns record

hash_metapage_info 返回元页中存储的信息，该元页属于一个HASH 索引。例如：

```
test=# SELECT magic, version, ntuples, ffactor, bsize, bmsize, bmshift,
test-#      maxbucket, highmask, lowmask, ovflpoint, firstfree, nmaps, procid,
test-#      regexp_replace(spares::text, '(,0)*}', '}') as spares,
test-#      regexp_replace(mapp::text, '(,0)*}', '}') as mapp
test-# FROM hash_metapage_info(get_raw_page('con_hash_index', 0));
-[ RECORD 1 ]----------------------------------------------------------------------------
-
magic     | 105121344
version   | 4
ntuples   | 500500
ffactor   | 40
bsize     | 8152
bmsize    | 4096
bmshift   | 15
maxbucket | 12512
highmask  | 16383
lowmask   | 8191
ovflpoint | 28
firstfree | 1204
nmaps     | 1
procid    | 450
spares    | {0,0,0,0,0,0,1,1,1,1,1,1,1,3,4,4,4,45,55,58,59,508,567,628,704,1193,1202,
1204}
mapp      | {65}
```

F.23. passwordcheck

每当通过 CREATE ROLE 或 ALTER ROLE 设置用户密码时，`passwordcheck` 模块都会检查密码强度。如果密码被认为过弱，就会被拒绝，该命令也会因错误而终止。

要启用该模块，请将`'$libdir/passwordcheck'` 加入 `postgresql.conf` 中的 shared_preload_libraries，然后重启服务器。

可以通过修改源代码使该模块适应实际需求。例如，可以使用 CrackLib[2] 来检查密码，这只需要在`Makefile` 中取消两行注释并重新构建该模块。（由于许可原因，我们不能默认包含 CrackLib。）如果不使用 CrackLib，该模块会对密码强度施加一些简单规则，而这些规则也可以按需修改或扩展。

> **小心**
>
> 为了防止未加密的密码通过网络传输、被写入服务器日志，或以其他方式被数据库管理员窃取，PostgreSQL 允许用户提供预加密密码。很多客户端程序都会利用这一能力，在将密码发送给服务器之前先行加密。
>
> 这就限制了 `passwordcheck` 模块的作用，因为在这种情况下，它只能尝试猜测密码。因此，如果安全要求很高，并不建议使用 `passwordcheck`。与其依赖数据库内部的密码，不如使用 GSSAPI 之类的外部认证方法（见第 20 章），那样更安全。
>
> 另外，也可以修改`passwordcheck` 以拒绝预加密密码，但强迫用户以明文设置密码本身也会带来安全风险。

[2] https://github.com/cracklib/cracklib

F.24. pg_buffercache

pg_buffercache 模块提供了一种手段，可实时检查共享缓冲区缓存中正在发生的情况。

该模块提供一个 C 函数 pg_buffercache_pages，它返回一组记录；另外还提供一个视图 pg_buffercache，该视图封装此函数以便使用。

默认情况下，仅限超级用户和pg_monitor 角色的成员使用。可以使用 GRANT 将访问权限授予其他角色。

F.24.1. pg_buffercache 视图

该视图公开的列定义见表 F.15。

表 F.15. pg_buffercache 列

名称	类型	引用	描述
bufferid	integer		ID，范围为 1..shared_buffers
relfilenode	oid	pg_class.relfilenode	关系的文件节点编号
reltablespace	oid	pg_tablespace.oid	关系的表空间 OID
reldatabase	oid	pg_database.oid	关系的数据库 OID
relforknumber	smallint		关系中的分支编号；参见include/common/relpath.h
relblocknumber	bigint		关系中的页号
isdirty	boolean		该页是否为脏页？
usagecount	smallint		时钟扫描访问计数
pinning_backends	integer		钉住该缓冲区的后端数量

共享缓存中的每个缓冲区都有一行记录。未使用的缓冲区除了 bufferid 之外，其余字段均显示为 null。共享系统目录显示为属于数据库 0。

由于缓存由所有数据库共享，其中通常会有不属于当前数据库的关系页。这意味着某些行在 pg_class 中可能没有可匹配的行，甚至还可能出现错误的连接结果。如果要与 pg_class 进行连接，最好将连接限制在 reldatabase 等于当前数据库 OID 或 0 的那些行上。

由于复制该视图将要显示的缓冲区状态数据时不会获取缓冲区管理器锁，访问 pg_buffercache 视图对正常缓冲区活动的影响较小，但它不会为所有缓冲区提供一致的结果集。不过，我们会确保每个缓冲区的信息本身是一致的。

F.24.2. 示例输出

```
regression=# SELECT n.nspname, c.relname, count(*) AS buffers
             FROM pg_buffercache b JOIN pg_class c
             ON b.relfilenode = pg_relation_filenode(c.oid) AND
                b.reldatabase IN (0, (SELECT oid FROM pg_database
                                      WHERE datname = current_database()))
             JOIN pg_namespace n ON n.oid = c.relnamespace
             GROUP BY n.nspname, c.relname
             ORDER BY 3 DESC
             LIMIT 10;

  nspname   |        relname          | buffers
------------+-------------------------+---------
 public     | delete_test_table       |     593
 public     | delete_test_table_pkey  |     494
 pg_catalog | pg_attribute            |     472
 public     | quad_poly_tbl           |     353
 public     | tenk2                   |     349
 public     | tenk1                   |     349
 public     | gin_test_idx            |     306
 pg_catalog | pg_largeobject          |     206
 public     | gin_test_tbl            |     188
 public     | spgist_text_tbl         |     182
(10 rows)
```

F.24.3. 作者

Mark Kirkwood <markir@paradise.net.nz>

设计建议：Neil Conway <neilc@samurai.com>

调试建议：Tom Lane <tgl@sss.pgh.pa.us>

F.25. pgcrypto

pgcrypto 模块为 PostgreSQL 提供密码学函数。

F.25.1. 通用 hash 函数

F.25.1.1. digest()

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

计算给定 *data* 的二进制 hash 值。*type* 是要使用的算法。标准算法包括 md5、sha1、sha224、sha256、sha384 和 sha512。如果 pgcrypto 是在有 OpenSSL 的情况下构建的，则还有更多算法可用，详见表 F.19。

如果想把摘要表示为十六进制字符串，可以对结果使用encode()。例如：

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

F.25.1.2. hmac()

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key bytea, type text) returns bytea
```

使用密钥 *key* 为 *data* 计算基于 hash 的消息认证码（HMAC）。*type* 与 digest() 中的相同。

这与 digest() 类似，但只有知道密钥时才能重新计算该 hash。这可以防止有人篡改数据后再同时修改 hash 使之匹配。

如果密钥大于 hash 块大小，则会先对其进行 hash 运算，并将结果用作密钥。

F.25.2. 密码 hash 函数

函数 crypt() 和 gen_salt() 是专门为对密码进行 hash 运算而设计的。crypt() 负责执行 hash 运算，而gen_salt() 负责为其准备算法参数。

crypt() 中的算法在以下方面不同于通常的 MD5 或 SHA1 hash 算法：

1. 它们很慢。由于处理的数据量很小，这是让暴力破解密码变得困难的唯一办法。
2. 它们使用一个称为 salt 的随机盐值，这样使用相同密码的用户也会得到不同的 hash 结果。这也为逆向求解该算法增加了一层额外防护。
3. 它们会在结果中包含算法类型，这样用不同算法进行 hash 运算的密码就能共存。
4. 其中一些是自适应的 – 这意味着当计算机变快时，你可以把算法调得更慢，而不引入与现有密码的不兼容性。

表 F.16 列出了 crypt() 函数支持的算法。

表 F.16. `crypt()` 支持的算法

算法	最大密码长度	是否自适应?	盐值位数	输出长度	描述
bf	72	是	128	60	基于 Blowfish，2a 变体
md5	无限制	否	48	34	基于 MD5 的 crypt
xdes	8	是	24	20	扩展 DES
des	8	否	12	13	原始 UNIX crypt

F.25.2.1. `crypt()`

```
crypt(password text, salt text) returns text
```

计算 *password* 的一个 crypt(3) 风格 hash。存储新密码时，需要使用`gen_salt()` 生成新的 *salt* 值。校验密码时，把已存储的 hash 值作为*salt* 传入，并测试结果是否与已存储值匹配。

设置一个新密码的示例：

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

身份验证示例：

```
SELECT (pswhash = crypt('entered password', pswhash)) AS pswmatch FROM ... ;
```

如果输入的密码正确，这会返回`true`。

F.25.2.2. `gen_salt()`

```
gen_salt(type text [, iter_count integer ]) returns text
```

生成一个供 `crypt()` 使用的新随机盐值字符串。该盐值字符串还会告诉 `crypt()` 应使用哪种算法。

type 参数指定 hash 算法。接受的类型有：`des`、`xdes`、`md5` 和 `bf`。

iter_count 参数允许用户为支持该参数的算法指定迭代次数。次数越高，对密码进行 hash 所需的时间越长，从而破解它所需时间也越长。不过，如果次数过高，计算一个 hash 可能需要数年时间 – 这就不太现实了。若省略*iter_count* 参数，则使用默认迭代次数。允许的 *iter_count* 值取决于算法，如表 F.17 所示。

表 F.17. `crypt()` 的迭代计数

算法	默认值	最小值	最大值
xdes	725	1	16777215
bf	6	4	31

对 `xdes` 算法还有额外的限制：迭代计数必须是一个奇数。

为了选择合适的迭代次数，可以考虑原始 DES crypt 在当时硬件上的设计速度是每秒 4 次 hash。低于每秒 4 次 hash 可能会影响可用性，而高于每秒 100 次 hash 则很可能过快。

表 F.18 概述了不同 hash 算法之间的相对速度差异。该表展示了在 8 字符密码上尝试所有字符组合所需的时间，假定密码只包含小写字母，或者包含大小写字母和数字。在`crypt-bf` 条目中，斜杠后的数字是`gen_salt` 的 *`iter_count`* 参数值。

表 F.18. Hash 算法速度

算法	每秒 hash 次数	针对 `[a-z]`	针对 `[A-Za-z0-9]`	相对于`md5 hash` 的耗时倍数
`crypt-bf/8`	1792	4 年	3927 年	100k
`crypt-bf/7`	3648	2 年	1929 年	50k
`crypt-bf/6`	7168	1 年	982 年	25k
`crypt-bf/5`	13504	188 天	521 年	12.5k
`crypt-md5`	171584	15 天	41 年	1k
`crypt-des`	23221568	157.5 分	108 天	7
`sha1`	37774272	90 分	68 天	4
`md5`(hash)	150085504	22.5 分	17 天	1

注意：

- 所用机器为 Intel Mobile Core i3。
- `crypt-des` 和 `crypt-md5` 算法的数字取自 John the Ripper v1.6.38 的 `-test` 输出。
- `md5 hash` 的数字来自 mdcrack 1.2。
- `sha1` 的数字来自 lcrack-20031130-beta。
- `crypt-bf` 的数字是使用一个简单程序测得的，该程序循环处理 1000 个 8 字符密码。这样我就可以展示不同迭代次数下的速度。作为参考：`john -test` 对 `crypt-bf/5` 给出的结果是 13506 次循环/秒。（结果上的极小差异与这样一个事实一致：`crypt-bf` 在 `pgcrypto` 中的实现与 John the Ripper 使用的是同一套实现。）

请注意，“尝试所有组合”并不是现实中的做法。通常密码破解是借助词典完成的，其中包含常见单词及其各种变体。因此，即使是稍微有点像单词的密码，被破解的速度也可能远快于上表所示；而一个 6 字符、不像单词的密码则可能逃过破解，也可能不会。

F.25.3. PGP 加密函数

这里的函数实现 OpenPGP（RFC 4880）标准的加密部分。同时支持对称密钥加密和公钥加密。

一个加密的 PGP 消息由两个部分，或称两个包组成：

- 包含会话密钥的包 – 该会话密钥要么由对称密钥加密，要么由公钥加密。
- 包含用会话密钥加密的数据的包。

当使用对称密钥（即密码）加密时：

1. 给定密码使用 String2Key (S2K) 算法进行 hash 运算。这与 `crypt()` 算法颇为类似 – 故意设计得较慢，并带有随机盐值 – 但它产生的是一个全长度的二进制密钥。

2. 如果请求单独的会话密钥，则会生成一个新的随机密钥。否则，S2K 密钥将直接用作会话密钥。

3. 如果直接使用 S2K 密钥，那么会话密钥包中只写入 S2K 设置。否则，会话密钥会先用 S2K 密钥加密，再放入会话密钥包。

当使用公钥加密时：

1. 会生成一个新的随机会话密钥。

2. 该密钥会用公钥加密，并放入会话密钥包中。

无论哪种情况，要加密的数据都会按如下步骤处理：

1. 可选的数据处理包括：压缩、转换为 UTF-8 和转换行结束符，三者可任意组合。

2. 数据前面会加上一个随机字节块。这相当于使用随机 IV。

3. 追加对随机前缀和数据计算得到的 SHA-1 hash 值。

4. 然后用会话密钥加密所有这些内容，并放入数据包。

F.25.3.1. pgp_sym_encrypt()

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

使用对称 PGP 密码 *psw* 加密 *data*。*options* 参数可以包含下文所述的选项设置。

F.25.3.2. pgp_sym_decrypt()

```
pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns text
pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ]) returns bytea
```

解密一个经过对称密钥加密的 PGP 消息。

bytea 数据不能用 pgp_sym_decrypt 解密。这是为了避免输出无效字符数据。若原始数据本来是文本，则使用pgp_sym_decrypt_bytea 解密也没有问题。

options 参数可以包含下文所述的选项设置。

F.25.3.3. pgp_pub_encrypt()

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

使用 PGP 公钥 *key* 加密 *data*。向该函数提供私钥会报错。

options 参数可以包含下文所述的选项设置。

F.25.3.4. pgp_pub_decrypt()

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns bytea
```

解密经过公钥加密的消息。*key* 必须是与加密时所用公钥对应的私钥。如果私钥受密码保护，则必须在*psw* 中给出密码。如果没有密码但想指定选项，则需要传入空密码。

bytea 数据不能用 pgp_pub_decrypt 解密。这是为了避免输出无效字符数据。若原始数据本来是文本，则使用pgp_pub_decrypt_bytea 解密也没有问题。

options 参数可以包含下文所述的选项设置。

F.25.3.5. pgp_key_id()

```
pgp_key_id(bytea) returns text
```

pgp_key_id 提取 PGP 公钥或私钥的密钥 ID。如果传入的是加密消息，则返回用于加密该数据的密钥 ID。

它可以返回两个特殊的密钥 ID：

- SYMKEY

  该消息是用对称密钥加密的。

- ANYKEY

  该消息是用公钥加密的，但密钥 ID 已被移除。这意味着你需要尝试自己的所有私钥，看看哪一个能解密它。pgcrypto 本身不会生成这样的消息。

注意，不同的密钥可能具有相同的 ID。这种情况虽然罕见，但属于正常情况。客户端应用此时应该尝试使用每一个密钥解密，以判断哪个匹配 – 就像处理ANYKEY 时一样。

F.25.3.6. armor(), dearmor()

```
armor(data bytea [ , keys text[], values text[] ]) returns text
dearmor(data text) returns bytea
```

这些函数将二进制数据封装/解封装为 PGP ASCII-armor 格式，它本质上就是带 CRC 和附加格式信息的 Base64。

如果指定了 *keys* 和 *values* 数组，则会为每个键/值对添加一个装甲头（armor header）。两个数组都必须是一维的，且长度相同。键和值都不能包含任何非 ASCII 字符。

F.25.3.7. pgp_armor_headers

```
pgp_armor_headers(data text, key out text, value out text) returns setof record
```

pgp_armor_headers() 从 *data* 中提取装甲头。返回值是一个包含两列的行集合，列名为 key 和 value。如果键或值包含任何非 ASCII 字符，则按 UTF-8 处理。

F.25.3.8. PGP 函数的选项

这些选项的命名方式与 GnuPG 类似。选项值应写在等号后面；各选项之间用逗号分隔。例如：

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

除 convert-crlf 外，所有选项都只适用于加密函数。解密函数会从 PGP 数据中获取这些参数。

最值得关注的选项可能是 compress-algo 和 unicode-mode。其余选项应该都具有合理的默认值。

F.25.3.8.1. cipher-algo

使用哪种密码算法。

取值：bf, aes128, aes192, aes256（仅限 OpenSSL：3des、cast5）
默认值：aes128
适用于：pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.2. compress-algo

使用哪种压缩算法。仅当 PostgreSQL 在编译时包含 zlib 时才可用。

取值：
 0 - 无压缩
 1 - ZIP 压缩
 2 - ZLIB 压缩（= ZIP 加元数据和块 CRC）
默认值：0
适用于：pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.3. compress-level

压缩程度。级别越高，压缩后越小，但速度也越慢。0 表示禁用压缩。

取值：0, 1-9
默认值：6
适用于：pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.4. convert-crlf

是否在加密时将\n 转换为\r\n，并在解密时将\r\n 转换为\n。RFC 4880 规定文本数据应使用\r\n 换行符存储。使用此选项可获得完全符合 RFC 的行为。

取值：0, 1
默认值：0
适用于：pgp_sym_encrypt, pgp_pub_encrypt, pgp_sym_decrypt, pgp_pub_decrypt

F.25.3.8.5. disable-mdc

不使用 SHA-1 保护数据。使用此选项唯一合理的理由是兼容早于 RFC 4880 加入 SHA-1 保护包的古老 PGP 产品。较新的 gnupg.org 和 pgp.com 软件都能很好地支持它。

取值：0, 1
默认值：0
适用于：pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.6. sess-key

使用单独的会话密钥。公钥加密总是使用单独的会话密钥；这个选项用于对称密钥加密，因为后者默认直接使用 S2K 密钥。

取值：0, 1
默认值：0
适用于：pgp_sym_encrypt

F.25.3.8.7. s2k-mode

使用哪一种 S2K 算法。

取值：
0 - 不使用盐值。危险！
1 - 使用盐值，但迭代次数固定。
3 - 迭代次数可变。
默认值：3
适用于：pgp_sym_encrypt

F.25.3.8.8. s2k-count

S2K 算法要使用的迭代次数。它必须是一个位于 1024 和 65011712 之间的值，首尾两个值包括在内。

默认值：65536 到 253952 之间的随机值
适用于：pgp_sym_encrypt，仅在 s2k-mode=3 时

F.25.3.8.9. s2k-digest-algo

要在 S2K 计算中使用哪种摘要算法。

取值：md5, sha1
默认值：sha1
适用于：pgp_sym_encrypt

F.25.3.8.10. s2k-cipher-algo

使用哪种密码算法来加密单独的会话密钥。

取值：bf, aes, aes128, aes192, aes256
默认值：使用 cipher-algo
适用于：pgp_sym_encrypt

F.25.3.8.11. unicode-mode

是否将文本数据在数据库内部编码和 UTF-8 之间相互转换。如果数据库已经是 UTF-8，则不会发生转换，但消息会被标记为 UTF-8。如果不启用该选项，则不会进行这种标记。

取值：0, 1
默认值：0
适用于：pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.9. 使用 GnuPG 生成 PGP 密钥

要生成新密钥：

```
gpg --gen-key
```

首选的密钥类型是“DSA 和 Elgamal”。

对于 RSA 加密，你必须先创建一个仅用于签名的 DSA 或 RSA 主密钥，然后使用 `gpg --edit-key` 添加 RSA 加密子密钥。

要列出密钥：

```
gpg --list-secret-keys
```

要以 ASCII-armor 格式导出公钥：

```
gpg -a --export KEYID > public.key
```

要以 ASCII-armor 格式导出私钥：

```
gpg -a --export-secret-keys KEYID > secret.key
```

在把这些密钥交给 PGP 函数之前，需要先用`dearmor()` 处理它们。或者，如果你能处理二进制数据，也可以从命令中去掉`-a`。

更多细节见 `man gpg`、GNU Privacy Handbook[3] 以及 https://www.gnupg.org/上的其他文档。

F.25.3.10. PGP 代码的限制

- 不支持签名。这也意味着不会检查加密子密钥是否属于主密钥。
- 不支持使用加密密钥作为主密钥。由于通常不鼓励这种做法，这不应成为问题。
- 不支持多个子密钥。这看起来可能是个问题，因为这在实践中相当常见。另一方面，你不应将常规的 GPG/PGP 密钥用于`pgcrypto`，而应新建一套密钥，因为其使用场景相当不同。

F.25.4. 原始加密函数

这些函数只是对数据应用密码算法；它们不具备 PGP 加密的任何高级特性。因此存在一些严重问题：

1. 它们直接把用户提供的密钥用作密码算法的密钥。
2. 它们不提供任何完整性检查，无法判断加密数据是否被修改。
3. 它们希望用户自己管理所有加密参数，甚至是 IV。
4. 它们无法处理文本。

因此，在引入 PGP 加密之后，不建议使用原始加密函数。

[3] https://www.gnupg.org/gph/en/manual.html

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

使用 *type* 指定的加密算法对数据进行加密/解密。*type* 字符串的语法是：

```
algorithm [ - mode ] [ /pad: padding ]
```

其中 *algorithm* 可以是：

- bf – Blowfish
- aes – AES（Rijndael-128、-192 或 -256）

而 *mode* 可以是：

- cbc – 下一个块依赖于前一个块（默认）
- ecb – 每个块单独加密（仅用于测试）

而 *padding* 可以是：

- pkcs – 数据长度可以任意（默认）
- none – 数据必须是分组大小的倍数

因此，例如下面两种写法是等价的：

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

在 encrypt_iv 和 decrypt_iv 中，*iv* 参数是 CBC 模式的初始值；对于 ECB，它会被忽略。如果长度不等于块大小，则会被截断或用零填充。在不带该参数的函数中，它默认为全零。

F.25.5. 随机数据函数

```
gen_random_bytes(count integer) returns bytea
```

返回 *count* 个具有密码学安全强度的随机字节。一次最多可提取 1024 个字节，以避免耗尽随机数生成器池。

```
gen_random_uuid() returns uuid
```

返回第 4 版（随机）UUID。

F.25.6. 注意事项

F.25.6.1. 配置

pgcrypto 会根据 PostgreSQL 主 configure 脚本的探测结果自行配置。影响它的选项有--with-zlib 和--with-openssl。

如果编译时包含 zlib，PGP 加密函数就能够在加密前压缩数据。

如果编译时包含 OpenSSL，则会有更多算法可用。同时公钥加密函数也会更快，因为 OpenSSL 具有更加优化的 BIGNUM 函数。

表 F.19. 有无 OpenSSL 时的功能概要

功能	内置	包含 OpenSSL 时
MD5	是	是
SHA1	是	是
SHA224/256/384/512	是	是
其他摘要算法	否	是（注 1）
Blowfish	是	是
AES	是	是
DES/3DES/CAST5	否	是
原始加密	是	是
PGP 对称加密	是	是
PGP 公钥加密	是	是

当针对 OpenSSL 3.0.0 及更高版本编译时，若要使用 DES 或 Blowfish 等旧式密码算法，必须在 openssl.cnf 配置文件中启用 legacy provider（旧版提供程序）。

注意：

1. OpenSSL 支持的任何摘要算法都会被自动识别并纳入支持。但密码算法不能这样做，需要显式支持。

F.25.6.2. NULL 处理

按照 SQL 的标准，只要任一参数为 NULL，所有函数都返回 NULL。这在使用不慎时可能带来安全风险。

F.25.6.3. 安全性限制

所有 pgcrypto 函数都在数据库服务器内部运行。这意味着所有数据和密码都会在 pgcrypto 与客户端应用之间以明文传输。因此，你必须：

1. 使用本地连接或 SSL 连接。

2. 同时信任系统管理员和数据库管理员。

如果做不到，最好在客户端应用内部执行密码学操作。

该实现无法抵御侧信道攻击[4]。例如，对于给定大小的密文，pgcrypto 解密函数完成所需的时间会因具体密文不同而变化。

[4] https://en.wikipedia.org/wiki/Side-channel_attack

F.25.6.4. 有用的阅读材料

- https://www.gnupg.org/gph/en/manual.html

  GNU 隐私手册。

- https://www.openwall.com/crypt/

  介绍 crypt-blowfish 算法。

- https://www.iusmentis.com/security/passphrasefaq/

  如何选择一个好的密码。

- http://world.std.com/~reinhold/diceware.html

  选择密码的有趣思路。

- http://www.interhack.net/people/cmcurtin/snake-oil-faq.html

  介绍好的和不好的密码学。

F.25.6.5. 技术参考资料

- https://tools.ietf.org/html/rfc4880

  OpenPGP 消息格式。

- https://tools.ietf.org/html/rfc1321

  MD5 消息摘要算法。

- https://tools.ietf.org/html/rfc2104

  HMAC：用于消息认证的带密钥的 hash 运算。

- https://www.usenix.org/legacy/events/usenix99/provos.html

  crypt-des、crypt-md5 和 bcrypt 算法的比较。

- https://en.wikipedia.org/wiki/Fortuna_(PRNG)

  Fortuna CSPRNG 的描述。

- https://jlcooke.ca/random/

  Jean-Luc Cooke 基于 Fortuna 的 Linux `/dev/random` 驱动。

F.25.7. 作者

Marko Kreen `<markokr@gmail.com>`

`pgcrypto` 使用了来自以下来源的代码：

算法	作者	源代码来源
DES crypt	David Burren 及其他人	FreeBSD libcrypt

算法	作者	源代码来源
MD5 crypt	Poul-Henning Kamp	FreeBSD libcrypt
Blowfish crypt	Solar Designer	www.openwall.com
Blowfish 密码算法	Simon Tatham	PuTTY
Rijndael 密码算法	Brian Gladman	OpenBSD sys/crypto
MD5 hash 和 SHA1	WIDE Project	KAME kame/sys/crypto
SHA256/384/512	Aaron D. Gifford	OpenBSD sys/crypto
BIGNUM 运算	Michael J. Fromberger	dartmouth.edu/~sting/sw/imath

F.26. pg_freespacemap

pg_freespacemap 模块提供了一种检查空闲空间映射（FSM）的方法。它提供了一个名为 pg_freespace 的函数，更准确地说，是两个重载函数。这些函数显示空闲空间映射中针对给定页面，或关系中所有页面记录的值。

默认情况下，只有超级用户以及pg_stat_scan_tables 角色的成员才能使用。可以使用 GRANT 向其他用户授予访问权限。

F.26.1. 函数

pg_freespace(rel regclass IN, blkno bigint IN) returns int2

返回该关系中由 blkno 指定页面上按 FSM 记录的空闲空间量。

pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)

显示该关系中每个页面上按 FSM 记录的空闲空间量。返回一组(blkno bigint, avail int2) 元组，其中关系中的每个页面对应一个元组。

存储在空闲空间映射中的值并不精确。它们会被舍入到 BLCKSZ 的 1/256 精度（在默认 BLCKSZ 下为 32 字节），并且不会随着元组的插入和更新而完全保持最新。

对于索引，被跟踪的是完全未使用的页面，而不是页面内部的空闲空间。因此，这些值本身并无实际意义，只能表明某个页面是满的还是空的。

> **注意**
>
> 该接口在 8.4 版中进行了更改，以反映同一版本引入的新 FSM 实现。

F.26.2. 示例输出

```
postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |    32
     4 |   704
     5 |   704
     6 |   704
     7 |  1216
     8 |   704
     9 |   704
    10 |   704
    11 |   704
    12 |   704
    13 |   704
    14 |   704
    15 |   704
    16 |   704
    17 |   704
    18 |   704
    19 |  3648
(20 rows)

postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace
--------------
         1216
(1 row)
```

F.26.3. 作者

最初版本由 Mark Kirkwood <markir@paradise.net.nz> 编写。在 8.4 版中，Heikki Linnakangas <heikki@enterprisedb.com> 为适应新的 FSM 实现对其进行了重写。

F.27. pg_prewarm

pg_prewarm 模块提供了一种便捷方式，可将关系数据加载到操作系统缓冲区缓存或 PostgreSQL 缓冲区缓存中。预热既可以使用 pg_prewarm 函数手动执行，也可以通过将 pg_prewarm 加入 shared_preload_libraries 自动执行。在后一种情况下，系统会运行一个后台工作进程，定期将共享缓冲区的内容记录到名为 autoprewarm.blocks 的文件中，并在重启后使用两个后台工作进程重新加载这些块。

F.27.1. 函数

```
pg_prewarm(regclass, mode text default 'buffer', fork text default 'main',
           first_block int8 default null,
           last_block int8 default null) RETURNS int8
```

第一个参数是要预热的关系。第二个参数是要使用的预热方法，下文会进一步讨论；第三个参数是要预热的关系分支，通常为main。第四个参数是要预热的第一个块号（NULL 可作为 0 的同义值）；第五个参数是要预热的最后一个块号（NULL 表示一直预热到该关系中的最后一个块）。返回值是已预热的块数。

可用的预热方法有三种。prefetch 会在支持的情况下向操作系统发出异步预取请求，否则就会报错。read 会读取所请求范围内的块；与 prefetch 不同，它是同步的，并且在所有平台和构建方式上都受支持，但可能较慢。buffer 会将所请求范围内的块读入数据库缓冲区缓存。

请注意，无论采用哪种方法，如果试图预热的块数超过可缓存的数量，那么无论是在使用 prefetch 或 read 时由操作系统缓存，还是在 PostgreSQL 使用 buffer 时缓存，都很可能会在读入更高块号的块时逐出较低块号的块。预热后的数据同样不会得到任何特殊的缓存逐出保护，因此其他系统活动可能会在这些新近预热的块被读入后不久就将其逐出；反过来，预热也可能会把其他数据从缓存中逐出。因此，预热通常在启动时最有用，因为此时缓存大多还是空的。

```
autoprewarm_start_worker() RETURNS void
```

启动主 autoprewarm 工作进程。这通常会自动发生；但如果服务器启动时未配置自动预热，而又希望稍后再启动该工作进程，这个函数就很有用。

```
autoprewarm_dump_now() RETURNS int8
```

立即更新 autoprewarm.blocks。如果 autoprewarm 工作进程当前没有运行，但预计会在下一次重启后运行它，这个函数就很有用。返回值是写入 autoprewarm.blocks 的记录数。

F.27.2. 配置参数

pg_prewarm.autoprewarm (boolean)

控制服务器是否应运行 autoprewarm 工作进程。默认值为 on。该参数只能在服务器启动时设置。

pg_prewarm.autoprewarm_interval (int)

这是更新 autoprewarm.blocks 的时间间隔。默认值为 300 秒。如果设置为 0，则该文件不会按固定间隔转储，而只会在服务器关闭时转储。

这些参数必须在 postgresql.conf 中设置。典型用法如下：

```
# postgresql.conf
shared_preload_libraries = 'pg_prewarm'

pg_prewarm.autoprewarm = true
pg_prewarm.autoprewarm_interval = 300s
```

F.27.3. 作者

Robert Haas <rhaas@postgresql.org>

F.28. pgrowlocks

pgrowlocks 模块提供了一个函数，用于显示指定表的行锁信息。

默认情况下，只有超级用户、pg_stat_scan_tables 角色的成员，以及对该表具有SELECT 权限的用户才能使用它。

F.28.1. 概述

```
pgrowlocks(text) returns setof record
```

参数为表名。结果是一个记录集合，其中表中每个被锁定的行对应结果中的一行。输出列见表 F.20。

表 F.20. pgrowlocks 输出列

名称	类型	描述
locked_row	tid	被锁定行的元组 ID（TID）
locker	xid	持锁者的事务 ID；若为多事务，则为多事务 ID
multi	boolean	如果持锁者为多事务，则为真
xids	xid[]	持锁者的事务 ID（若为多事务，则会有多个）
modes	text[]	持锁者的锁模式（若为多事务，则会有多个），是由 Key Share、Share、For No Key Update、No Key Update、For Update、Update 组成的数组。
pids	integer[]	加锁后端的进程 ID（若为多事务，则会有多个）

pgrowlocks 会对目标表获取 AccessShareLock，并逐行读取以收集行锁信息。对于大表，这样做速度并不快。请注意：

1. 如果该表上存在 ACCESS EXCLUSIVE 锁，pgrowlocks 将被阻塞。
2. pgrowlocks 并不保证给出一个自一致快照。在其执行期间，可能会有新的行锁被获取，也可能有旧的锁被释放。

pgrowlocks 不显示被锁定行的内容。如果你想同时查看这些行的内容，可以这样做：

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
  WHERE p.locked_row = a.ctid;
```

不过要注意，这样的查询效率会很低。

F.28.2. 示例输出

```
=# SELECT * FROM pgrowlocks('t1');
 locked_row | locker | multi | xids  |     modes      |  pids
------------+--------+-------+-------+----------------+--------
 (0,1)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,2)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,3)      |    607 | f     | {607} | {"For Update"} | {3107}
 (0,4)      |    607 | f     | {607} | {"For Update"} | {3107}
(4 rows)
```

F.28.3. 作者

Tatsuo Ishii

F.29. pg_stat_statements

pg_stat_statements 模块提供了一种机制，用于跟踪服务器执行的所有 SQL 语句的执行统计信息。

由于该模块需要额外的共享内存，因此必须通过在postgresql.conf 中将 pg_stat_statements 加入 shared_preload_libraries 来加载该模块。这意味着添加或移除此模块都需要重启服务器。

启用 pg_stat_statements 后，它会跟踪该服务器上所有数据库的统计信息。为了访问和操作这些统计信息，该模块提供了视图pg_stat_statements，以及实用函数pg_stat_statements_reset 和 pg_stat_statements。这些对象并非全局可用，但可以通过 CREATE EXTENSION pg_stat_statements 在特定数据库中启用。

F.29.1. pg_stat_statements 视图

该模块收集的统计信息可通过名为 pg_stat_statements 的视图获取。该视图为每个不同的数据库 ID、用户 ID 和查询 ID 包含一行（最多达到该模块能够跟踪的不同语句数量上限）。视图的列见表 F.21。

表 F.21. pg_stat_statements 列

名称	类型	引用	描述
userid	oid	pg_authid.oid	执行该语句的用户的 OID
dbid	oid	pg_database.oid	执行该语句所在数据库的 OID
queryid	bigint		根据语句的解析树计算的内部 hash 码
query	text		某个代表性语句的文本
calls	bigint		执行次数
total_time	double precision		该语句所花费的总时间，单位为毫秒
min_time	double precision		该语句所花费的最短时间，单位为毫秒
max_time	double precision		该语句所花费的最长时间，单位为毫秒
mean_time	double precision		该语句所花费的平均时间，单位为毫秒
stddev_time	double precision		该语句所花费时间的总体标准差，单位为毫秒
rows	bigint		该语句检索到或影响的总行数
shared_blks_hit	bigint		该语句的共享块缓存命中总数
shared_blks_read	bigint		该语句读取的共享块总数
shared_blks_dirtied	bigint		被该语句弄脏的共享块总数
shared_blks_written	bigint		该语句写入的共享块总数

名称	类型	引用	描述
local_blks_hit	bigint		该语句的本地块缓存命中总数
local_blks_read	bigint		该语句读取的本地块总数
local_blks_dirtied	bigint		被该语句弄脏的本地块总数
local_blks_written	bigint		该语句写入的本地块总数
temp_blks_read	bigint		该语句读取的临时块总数
temp_blks_written	bigint		该语句写入的临时块总数
blk_read_time	double precision		该语句读取块所花费的总时间，单位为毫秒（如果启用了 track_io_timing，否则为零）
blk_write_time	double precision		该语句写入块所花费的总时间，单位为毫秒（如果启用了 track_io_timing，否则为零）

出于安全原因，只有超级用户以及 pg_read_all_stats 角色的成员才允许查看其他用户执行的查询的 SQL 文本和 queryid。不过，只要该视图已经安装在其数据库中，其他用户仍然可以查看统计信息。

可规划的查询（即 SELECT、INSERT、UPDATE 和 DELETE），只要根据内部 hash 计算得出的查询结构相同，就会合并为单条 pg_stat_statements 记录。通常，如果两个查询在语义上等价，仅在查询中出现的字面常量值不同，则会被视为相同。不过，实用命令（即所有其他命令）是严格按照其文本查询字符串进行比较的。

当为了将某个查询与其他查询匹配而忽略常量值时，在 pg_stat_statements 的显示中，该常量会被参数符号（例如 $1）替换。查询文本的其余部分则取自第一个具有该特定 queryid hash 值、并与该 pg_stat_statements 记录相关联的查询。

在某些情况下，文本明显不同的查询也可能被合并到同一条 pg_stat_statements 记录中。通常，这只会发生在语义等价的查询之间，但也存在很小的概率由于 hash 冲突而把不相关的查询合并为同一条记录。（不过，这种情况不会发生在属于不同用户或不同数据库的查询之间。）

由于 queryid hash 值是根据查询经过解析分析后的表示形式计算出来的，相反的情况也可能发生：文本完全相同的查询，如果由于不同的 search_path 设置等因素而具有不同含义，就可能显示为不同的记录。

pg_stat_statements 的使用者可能希望使用 queryid（也许再结合 dbid 和 userid）作为每条记录比查询文本更稳定、更可靠的标识符。不过，必须理解的是，queryid hash 值的稳定性只得到有限保证。由于该标识符源自解析分析后的语法树，它的取值除其他因素外，还取决于该表示形式中出现的内部对象标识符。这会带来一些反直觉的结果。例如，如果两次查询执行之间，所引用的某张表被删除并重新创建，那么 pg_stat_statements 会将两条看似完全相同的查询视为不同。hash 运算过程也对机器架构差异以及平台的其他方面很敏感。此外，也不能安全地假设 queryid 会在 PostgreSQL 的主版本之间保持稳定。

通常可以假定 queryid 值是稳定且可比较的，前提是底层服务器版本和目录元数据细节始终完全相同。参与基于物理 WAL 重放的复制的两个服务器，对于同一查询可以期待具有相同的queryid 值。然而，逻辑复制方案并不承诺在所有相关细节上保持副本完全一致，因此 queryid 不适合作为在一组逻辑副本之间累积开销的标识符。如有疑问，建议直接测试。

在代表性查询文本中，用于替换常量的参数符号从原始查询文本中最高的 $n 参数之后的下一个数字开始，如果原文中没有，则从 $1 开始。值得注意的是，在某些情况下，可能存在会影响这一编号的隐藏参数符号。例如，PL/pgSQL 使用隐藏参数符号将函数局部变量的值插入查询中，因此像 SELECT i + 1 INTO j 这样的 PL/pgSQL 语句，其代表性文本可能是 SELECT i + $2。

代表性查询文本保存在外部磁盘文件中，因此不消耗共享内存。即使非常长的查询文本也可以成功存储。不过，如果积累了很多长查询文本，该外部文件可能会膨胀到难以管理的大小。若发生这种情况，作为一种恢复措施，pg_stat_statements 可能会选择丢弃这些查询文本，这样 pg_stat_statements 视图中的现有记录都会显示 query 字段为 null，但与各个 queryid 相关的统计信息仍会保留。如果发生这种情况，可考虑减小 pg_stat_statements.max 以避免再次发生。

F.29.2. 函数

pg_stat_statements_reset() returns void

pg_stat_statements_reset 会丢弃到目前为止由 pg_stat_statements 收集的所有统计信息。默认情况下，只有超级用户可以执行此函数。

pg_stat_statements(showtext boolean) returns setof record

pg_stat_statements 视图是根据一个同名的 pg_stat_statements 函数定义的。客户端也可以直接调用 pg_stat_statements 函数，并通过指定 showtext := false 来省略查询文本（也就是说，与该视图 query 列对应的 OUT 参数会返回 null）。这一特性旨在支持某些外部工具，这些工具可能希望避免反复获取长度不定的查询文本所带来的开销。这些工具可以自行缓存每条记录第一次观察到的查询文本，因为这正是 pg_stat_statements 本身所做的事情，然后只在需要时再获取查询文本。由于服务器会把查询文本存储在文件中，这种做法在反复检查 pg_stat_statements 数据时可以减少物理 I/O。

F.29.3. 配置参数

pg_stat_statements.max (integer)

pg_stat_statements.max 是该模块跟踪的语句最大数量（即 pg_stat_statements 视图中的最大行数）。如果观察到的不同语句超过该数量，则执行次数最少的语句信息会被丢弃。默认值为 5000。该参数只能在服务器启动时设置。

pg_stat_statements.track (enum)

pg_stat_statements.track 控制该模块统计哪些语句。指定 top 可跟踪顶层语句（即直接由客户端发出的语句），指定 all 还会跟踪嵌套语句（例如在函数内调用的语句），而指定 none 则会禁用语句统计信息收集。默认值为 top。只有超级用户可以更改此设置。

pg_stat_statements.track_utility (boolean)

pg_stat_statements.track_utility 控制该模块是否跟踪实用命令。实用命令是除 SELECT、INSERT、UPDATE 和 DELETE 之外的所有命令。默认值为 on。只有超级用户可以更改此设置。

pg_stat_statements.save (boolean)

pg_stat_statements.save 指定是否在服务器关闭后保留语句统计信息。如果它为 off，则在关闭时不会保存统计信息，并且在服务器启动时也不会重新载入这些统计信息。默认值为 on。该参数只能在 postgresql.conf 文件中或在服务器命令行上设置。

该模块需要与 pg_stat_statements.max 成比例的额外共享内存。注意，只要该模块被载入，就会消耗这部分内存，即使 pg_stat_statements.track 被设置为 none 也是如此。

这些参数必须在 postgresql.conf 中设置。典型用法如下：

```
# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'

pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

F.29.4. 示例输出

```
bench=# SELECT pg_stat_statements_reset();

$ pgbench -i bench
$ pgbench -c10 -t300 bench

bench=# \x
bench=# SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
-[ RECORD 1 ]---------------------------------------------------------------------
query       | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2;
calls       | 3000
total_time  | 9609.00100000002
rows        | 2836
hit_percent | 99.9778970000200936
-[ RECORD 2 ]---------------------------------------------------------------------
query       | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2;
calls       | 3000
total_time  | 8015.156
rows        | 2990
hit_percent | 99.9731126579631345
-[ RECORD 3 ]---------------------------------------------------------------------
query       | copy pgbench_accounts from stdin
calls       | 1
total_time  | 310.624
rows        | 100000
hit_percent | 0.30395136778115501520
-[ RECORD 4 ]---------------------------------------------------------------------
query       | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2;
calls       | 3000
total_time  | 271.741999999997
rows        | 3000
hit_percent | 93.7968855088209426
-[ RECORD 5 ]---------------------------------------------------------------------
query       | alter table pgbench_accounts add primary key (aid)
calls       | 1
total_time  | 81.42
rows        | 0
hit_percent | 34.4947735191637631
```

F.29.5. 作者

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>。查询规范化功能由 Peter Geoghegan <peter@2ndquadrant.com> 添加。

F.30. pgstattuple

pgstattuple 模块提供了多种用于获取元组级统计信息的函数。

由于这些函数会返回详细的页级信息，因此默认会限制访问。默认情况下，只有角色 pg_stat_scan_tables 拥有 EXECUTE 权限。超级用户当然可以绕过这一限制。安装该扩展后，用户可以发出 GRANT 命令来更改这些函数上的权限，以允许其他用户执行它们。不过，更可取的做法可能是将这些用户加入 pg_stat_scan_tables 角色。

F.30.1. 函数

```
pgstattuple(regclass) returns record
```

pgstattuple 返回一个关系的物理长度、“死”元组所占百分比以及其他信息。这可以帮助用户判断是否需要执行清理。参数是目标关系的名称（可选带模式限定）或 OID。例如：

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]------+-------
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

输出列的说明见表 F.22。

表 F.22. pgstattuple 输出列

列	类型	描述
table_len	bigint	关系的物理长度（字节）
tuple_count	bigint	活元组数量
tuple_len	bigint	活元组总长度（字节）
tuple_percent	float8	活元组百分比
dead_tuple_count	bigint	死元组数量
dead_tuple_len	bigint	死元组总长度（字节）
dead_tuple_percent	float8	死元组百分比
free_space	bigint	总空闲空间（字节）
free_percent	float8	空闲空间百分比

注意

table_len 总会大于 tuple_len、dead_tuple_len 和 free_space 之和。差值来自固定的页开销、每页的元组指针表，以及为保证元组正确对齐而产生的填充。

pgstattuple 只会在该关系上获取读锁。因此，结果并不代表一个瞬时快照；并发更新会影响结果。

如果 HeapTupleSatisfiesDirty 返回 false，pgstattuple 就会将该元组判定为“死”。

pgstattuple(text) returns record

这与 pgstattuple(regclass) 相同，只是目标关系是以 TEXT 指定的。该函数目前保留只是出于向后兼容考虑，并将在未来的某个版本中弃用。

pgstatindex(regclass) returns record

pgstatindex 返回一条记录，显示有关 B-树索引的信息。例如：

```
test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 16384
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 54.27
leaf_fragmentation | 0
```

输出列如下：

列	类型	描述
version	integer	B-树版本号
tree_level	integer	根页所在的树层级
index_size	bigint	索引总大小（字节）
root_block_no	bigint	根页的位置（若无则为零）
internal_pages	bigint	“内部”（上层）页的数量
leaf_pages	bigint	叶子页数量
empty_pages	bigint	空页数量
deleted_pages	bigint	已删除页数量
avg_leaf_density	float8	叶子页平均密度
leaf_fragmentation	float8	叶子页碎片化程度

报告的 index_size 通常会比 internal_pages + leaf_pages + empty_pages + deleted_pages 所对应的页数多 1 页，因为它还包含索引的元页。

与 pgstattuple 一样，结果是逐页累积得到的，因此不应期望它表示整个索引的瞬时快照。

`pgstatindex(text) returns record`

这与 `pgstatindex(regclass)` 相同，只是目标索引是以 TEXT 指定的。该函数目前保留只是出于向后兼容考虑，并将在未来的某个版本中弃用。

`pgstatginindex(regclass) returns record`

`pgstatginindex` 返回一条记录，显示有关 GIN 索引的信息。例如：

```
test=> SELECT * FROM pgstatginindex('test_gin_index');
-[ RECORD 1 ]--+--
version        | 1
pending_pages  | 0
pending_tuples | 0
```

输出列如下：

列	类型	描述
`version`	`integer`	GIN 版本号
`pending_pages`	`integer`	待处理列表中的页数
`pending_tuples`	`bigint`	待处理列表中的元组数

`pgstathashindex(regclass) returns record`

`pgstathashindex` 返回一条记录，显示有关 HASH 索引的信息。例如：

```
test=> select * from pgstathashindex('con_hash_index');
-[ RECORD 1 ]--+-----------------
version        | 4
bucket_pages   | 33081
overflow_pages | 0
bitmap_pages   | 1
unused_pages   | 32455
live_items     | 10204006
dead_items     | 0
free_percent   | 61.8005949100872
```

输出列如下：

列	类型	描述
`version`	`integer`	HASH 版本号
`bucket_pages`	`bigint`	桶页数量
`overflow_pages`	`bigint`	溢出页数量
`bitmap_pages`	`bigint`	位图页数量
`unused_pages`	`bigint`	未使用页数量
`live_items`	`bigint`	活元组数量
`dead_items`	`bigint`	死元组数量
`free_percent`	`float`	空闲空间百分比

`pg_relpages(regclass) returns bigint`

`pg_relpages` 返回该关系中的页数。

`pg_relpages(text) returns bigint`

这与 `pg_relpages(regclass)` 相同，只是目标关系是以 TEXT 指定的。该函数目前保留只是出于向后兼容考虑，并将在未来的某个版本中弃用。

`pgstattuple_approx(regclass) returns record`

`pgstattuple_approx` 是 `pgstattuple` 的一种更快替代方案，返回近似结果。参数是目标关系的名称或 OID。例如：

```
test=> SELECT * FROM pgstattuple_approx('pg_catalog.pg_proc'::regclass);
-[ RECORD 1 ]--------+-------
table_len            | 573440
scanned_percent      | 2
approx_tuple_count   | 2740
approx_tuple_len     | 561210
approx_tuple_percent | 97.87
dead_tuple_count     | 0
dead_tuple_len       | 0
dead_tuple_percent   | 0
approx_free_space    | 11996
approx_free_percent  | 2.09
```

输出列的说明见表 F.23。

`pgstattuple` 总是执行全表扫描，并返回活元组和死元组的精确计数及其总长度，以及空闲空间；相比之下，`pgstattuple_approx` 试图避免全表扫描，返回精确的死元组统计信息，以及活元组数量、活元组总长度和空闲空间的近似值。

它通过跳过那些根据可见性映射只包含可见元组的页来做到这一点（如果某页设置了相应的 VM 位，就假定该页不包含死元组）。对于这类页，它从空闲空间映射中得出空闲空间值，并假定页中其余空间都被活元组占用。

对于不能跳过的页，它会扫描每个元组，在相应的计数器中记录其存在和大小，并累加该页上的空闲空间。最后，它根据扫描过的页数和元组数来估算活元组总数（方法与 VACUUM 估算 pg_class.reltuples 时相同）。

表 F.23. `pgstattuple_approx` 输出列

列	类型	描述
`table_len`	`bigint`	关系的物理长度（字节，精确）
`scanned_percent`	`float8`	已扫描表的百分比
`approx_tuple_count`	`bigint`	活元组数量（估计）
`approx_tuple_len`	`bigint`	活元组总长度（字节，估计）
`approx_tuple_percent`	`float8`	活元组百分比
`dead_tuple_count`	`bigint`	死元组数量（精确）
`dead_tuple_len`	`bigint`	死元组总长度（字节，精确）
`dead_tuple_percent`	`float8`	死元组百分比
`approx_free_space`	`bigint`	总空闲空间（字节，估计）

列	类型	描述
approx_free_percent	float8	空闲空间百分比

在上述输出中，空闲空间数值可能与 pgstattuple 的输出不完全一致，因为空闲空间映射给出的数值是确定的，但不保证精确到字节。

F.30.2. 作者

Tatsuo Ishii、Satoshi Nagayasu 和 Abhijit Menon-Sen

F.31. pg_trgm

pg_trgm 模块提供函数和操作符，用于基于三字符组匹配确定字母数字文本的相似度，同时还提供支持快速搜索相似字符串的索引操作符类。

F.31.1. 三字符组（Trigram 或 Trigraph）概念

三字符组是一组从字符串中取出的三个连续字符。我们可以通过统计两个字符串共享的三字符组数量来度量它们的相似度。这个简单的思想在度量许多自然语言中词的相似度时都非常有效。

> **注意**
>
> 从字符串中提取三字符组时，pg_trgm 会忽略非词字符（即非字母数字字符）。在确定字符串所包含的三字符组集合时，认为每个词前面都有两个空格，后面都有一个空格。例如，字符串“cat”的三字符组集合是“ c”、“ ca”、“cat”和“at ”。字符串“foo|bar”的三字符组集合是“ f”、“ fo”、“foo”、“oo ”、“ b”、“ ba”、“bar”和“ar ”。

F.31.2. 函数和操作符

pg_trgm 模块提供的函数列在表 F.24 中，操作符列在表 F.25 中。

表 F.24. pg_trgm 函数

函数	返回值	描述
similarity(text, text)	real	返回一个表示两个参数有多相似的数值。结果范围从 0（表示两个字符串完全不同）到 1（表示两个字符串完全相同）。
show_trgm(text)	text[]	返回由给定字符串中所有三字符组构成的数组。（实际应用中，除了调试之外很少有用。）
word_similarity(text, text)	real	返回一个数值，表示第一个字符串中的三字符组集合与第二个字符串中的有序三字符组集合中任意连续区段之间的最大相似度。详见下文说明。
strict_word_similarity(text, text)	real	与word_similarity(text, text) 相同，但强制范围边界与单词边界相匹配。由于不存在跨单词的三字符组，该函数实际返回第一个字符串与第二个字符串中任意连续单词范围之间的最大相似度。
show_limit()	real	返回% 操作符当前使用的相似度阈值。它设置两个单词被视为足够相似、例如可以看作彼此的拼写错误所需的最小相似度（已弃用）。
set_limit(real)	real	设置% 操作符当前使用的相似度阈值。阈值必须介于 0 和 1 之间（默认值为 0.3）。返回传入的相同值（已弃用）。

考虑以下示例：

```
# SELECT word_similarity('word', 'two words');
 word_similarity
-----------------
             0.8
(1 row)
```

在第一个字符串中，三字符组集合为{" w"," wo","wor","ord","rd "}。在第二个字符串中，有序三字符组集合为{" t"," tw","two","wo "," w"," wo","wor","ord","rds","ds "}。第二个字符串的有序三字符组集合中最相似的连续区段是{" w"," wo","wor","ord"}，相似度为0.8。

这个函数返回的值大致可以理解为第一个字符串与第二个字符串任意子串之间的最大相似度。不过，该函数不会在这个区段的边界处添加填充。因此，除了词边界不匹配的情况外，第二个字符串中额外存在的字符数不会被考虑在内。

同时，strict_word_similarity(text, text) 会在第二个字符串中选择一个由完整单词组成的连续区段。在上面的示例中，strict_word_similarity(text, text) 会选择仅包含单个单词的区段，这个单词是'words'，其三字符组集合为{" w"," wo","wor","ord","rds","ds "}。

```
# SELECT strict_word_similarity('word', 'two words'), similarity('word', 'words');
 strict_word_similarity | similarity
------------------------+------------
               0.571429 |   0.571429
(1 row)
```

因此，strict_word_similarity(text, text) 适合查找与整个词的相似度，而word_similarity(text, text) 更适合查找与词的一部分的相似度。

表 F.25. pg_trgm 操作符

操作符	返回值	描述
text % text	boolean	如果参数之间的相似度大于pg_trgm.similarity_threshold 设置的当前相似度阈值，则返回true。
text <% text	boolean	如果第一个参数中的三字符组集合与第二个参数中的有序三字符组集合某个连续区段之间的相似度大于pg_trgm.word_similarity_threshold 参数设置的当前词相似度阈值，则返回true。
text %> text	boolean	<% 操作符的交换子。
text <<% text	boolean	如果第二个参数中存在一个与词边界一致的有序三字符组集合连续区段，且它与第一个参数三字符组集合的相似度大于 pg_trgm.strict_word_similarity_threshold 参数设置的当前严格词相似度阈值，则返回true。
text %>> text	boolean	<<% 操作符的交换子。

操作符	返回值	描述
text <-> text	real	返回参数之间的“距离”，即 1 减去similarity() 的值。
text <<-> text	real	返回参数之间的“距离”，即 1 减去word_similarity() 的值。
text <->> text	real	<<-> 操作符的交换子。
text <<<-> text	real	返回参数之间的“距离”，即 1 减去strict_word_similarity() 的值。
text <->>> text	real	<<<-> 操作符的交换子。

F.31.3. GUC 参数

pg_trgm.similarity_threshold(real)

设置% 操作符使用的当前相似度阈值。该阈值必须介于 0 和 1 之间（默认值为 0.3）。

pg_trgm.word_similarity_threshold (real)

设置<% 和%> 操作符使用的当前词相似度阈值。该阈值必须介于 0 和 1 之间（默认值为 0.6）。

pg_trgm.strict_word_similarity_threshold (real)

设置<<% 和%>> 操作符使用的当前严格词相似度阈值。该阈值必须介于 0 和 1 之间（默认值为 0.5）。

F.31.4. 索引支持

pg_trgm 模块提供 GiST 和 GIN 索引操作符类，允许你为文本列创建索引，以实现非常快速的相似度搜索。这些索引类型支持上述相似度操作符，还支持对LIKE、ILIKE、~和~* 查询执行基于三字符组的索引搜索。（这些索引不支持等值操作符或简单比较操作符，因此你可能还需要一个常规 B-树索引。）

示例：

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops);
```

或者

```
CREATE INDEX trgm_idx ON test_trgm USING GIN (t gin_trgm_ops);
```

此时，你已经在t 列上建立了一个可用于相似度搜索的索引。典型查询如下：

```
SELECT t, similarity(t, 'word') AS sml
  FROM test_trgm
  WHERE t % 'word'
  ORDER BY sml DESC, t;
```

这会返回文本列中所有与 *word* 足够相似的值，并按从最佳匹配到最差匹配的顺序排序。即使在非常大的数据集上，索引也会让这一操作保持高效。

上述查询的一个变体是

```
SELECT t, t <-> 'word' AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

GiST 索引可以相当高效地实现这一点，但 GIN 索引不能。当只需要少量最接近的匹配项时，它通常会优于第一种写法。

还可以把 t 列上的索引用于词相似度或严格词相似度搜索。典型查询如下：

```
SELECT t, word_similarity('word', t) AS sml
  FROM test_trgm
  WHERE 'word' <% t
  ORDER BY sml DESC, t;
```

以及

```
SELECT t, strict_word_similarity('word', t) AS sml
  FROM test_trgm
  WHERE 'word' <<% t
  ORDER BY sml DESC, t;
```

这会返回文本列中所有满足以下条件的值：在其对应的有序三字符组集合中，存在一个连续区段与*word* 的三字符组集合足够相似。结果按从最佳匹配到最差匹配的顺序排序。即使在非常大的数据集上，索引也会让这一操作保持高效。

上述查询的可能变体还有：

```
SELECT t, 'word' <<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

以及

```
SELECT t, 'word' <<<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

GiST 索引可以相当高效地实现这一点，但 GIN 索引不能。

从 PostgreSQL 9.1 起，这些索引类型还支持LIKE 和 ILIKE 的索引搜索，例如

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

索引搜索的工作方式是从搜索字符串中提取三字符组，然后在索引中查找这些三字符组。搜索字符串中包含的三字符组越多，索引搜索就越有效。与基于 B-树的搜索不同，搜索字符串不需要在左端锚定。

从 PostgreSQL 9.3 起，这些索引类型还支持正则表达式匹配（~和~* 操作符）的索引搜索，例如

```
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

索引搜索的工作方式是从正则表达式中提取三字符组，然后在索引中查找这些三字符组。能从正则表达式中提取出的三字符组越多，索引搜索就越有效。与基于 B-树的搜索不同，搜索字符串不需要在左端锚定。

对于 LIKE 和正则表达式搜索，都要记住：无法提取出三字符组的模式会退化为全索引扫描。

GiST 和 GIN 索引之间如何取舍，取决于二者各自的相对性能特征；相关讨论见其他章节。

F.31.5. 全文检索集成

与全文索引结合使用时，三字符组匹配是非常有用的工具。尤其是，它有助于识别那些因拼写错误而无法被全文检索机制直接匹配的输入词。

第一步是生成一个辅助表，其中包含文档中的所有不重复的词：

```
CREATE TABLE words AS SELECT word FROM
        ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM documents');
```

其中 documents 是一个表，包含我们希望搜索的文本字段bodytext。之所以对 to_tsvector 函数使用 simple 配置，而不是使用特定语言的配置，是因为我们需要原始的（未经词干提取的）词列表。

接下来，在词列上创建一个三字符组索引：

```
CREATE INDEX words_idx ON words USING GIN (word gin_trgm_ops);
```

现在，可以使用与前面示例类似的SELECT 查询，为用户搜索词中拼错的单词提供拼写建议。一个有用的附加测试是要求选出的词长度也与该拼错单词相近。

> **注意**
>
> 由于 words 表是作为一张独立的静态表生成的，因此需要定期重新生成，以便与文档集合保持大致同步。通常没有必要让它始终保持精确同步。

F.31.6. 参考

GiST 开发站点 http://www.sai.msu.su/~megera/postgres/gist/

Tsearch2 开发站点 http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

F.31.7. 作者

Oleg Bartunov <oleg@sai.msu.su>，俄罗斯莫斯科，莫斯科大学

Teodor Sigaev <teodor@sigaev.ru>，俄罗斯莫斯科，Delta-Soft Ltd.

Alexander Korotkov <a.korotkov@postgrespro.ru>，俄罗斯莫斯科，Postgres Professional

文档：Christopher Kings-Lynne

该模块由俄罗斯莫斯科的 Delta-Soft Ltd. 赞助。

F.32. pg_visibility

pg_visibility 模块提供了一种检查表的可见性映射（VM）以及页级可见性信息的手段。它还提供了一些函数，用于检查可见性映射的完整性，并强制重建该映射。

页级可见性信息使用三个不同的位来存储。可见性映射中的全部可见（all-visible）位表示该关系对应页中的每个元组对当前及未来的所有事务均可见。可见性映射中的全部冻结（all-frozen）位表示该页中的每个元组都已被冻结；也就是说，在该页上发生插入、更新、删除或锁定操作之前，未来的清理（VACUUM）都不需要修改该页。页头中的PD_ALL_VISIBLE 位与可见性映射中的 all-visible 位含义相同，但它存储在数据页本身之中，而非单独的数据结构中。这两个位通常应当一致，但在崩溃恢复之后，有时页面上的 all-visible 位可能已被置位，而可见性映射中的对应位却仍然为零。由于在pg_visibility 检查可见性映射之后、检查数据页之前可能发生变更，因此报告的值也可能不一致。任何导致数据损坏的事件也可能导致这些位不一致。

显示 PD_ALL_VISIBLE 位信息的函数比仅查询可见性映射的函数代价高得多，因为它们必须读取关系的数据块，而不只是读取（小得多的）可见性映射。检查关系数据块的函数同样代价高昂。

F.32.1. 函数

```
pg_visibility_map(relation regclass, blkno bigint, all_visible OUT
boolean, all_frozen OUT boolean) returns record
```

返回给定关系中给定块在可见性映射中的 all-visible 位和 all-frozen 位。

```
pg_visibility(relation regclass, blkno bigint, all_visible OUT
boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns
record
```

返回给定关系中给定块在可见性映射中的 all-visible 位和 all-frozen 位，以及该块的 PD_ALL_VISIBLE 位。

```
pg_visibility_map(relation regclass, blkno OUT bigint, all_visible OUT
boolean, all_frozen OUT boolean) returns setof record
```

返回给定关系中每个块在可见性映射中的 all-visible 位和 all-frozen 位。

```
pg_visibility(relation regclass, blkno OUT bigint, all_visible OUT
boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns
setof record
```

返回给定关系中每个块在可见性映射中的 all-visible 位和 all-frozen 位，以及每个块的 PD_ALL_VISIBLE 位。

```
pg_visibility_map_summary(relation regclass, all_visible OUT bigint,
 all_frozen OUT bigint) returns record
```

根据可见性映射，返回关系中 all-visible 页面和 all-frozen 页面的数量。

```
pg_check_frozen(relation regclass, t_ctid OUT tid) returns setof tid
```

返回存储在可见性映射中标记为 all-frozen 的页面里、实际上并未冻结的元组的 TID。如果该函数返回的 TID 集合非空，则说明可见性映射已损坏。

`pg_check_visible(relation regclass, t_ctid OUT tid) returns setof tid`

返回存储在可见性映射中标记为 all-visible 的页面里、实际上并非对所有事务都可见的元组的 TID。如果该函数返回的 TID 集合非空，则说明可见性映射已损坏。

`pg_truncate_visibility_map(relation regclass) returns void`

截断给定关系的可见性映射。如果你认为该关系的可见性映射已损坏并希望强制重建它，此函数就很有用。在执行此函数之后，对该关系执行的第一次`VACUUM`将会扫描关系中的每一页并重建可见性映射。（在此完成之前，查询会将可见性映射视为全部为零。）

默认情况下，这些函数只能由超级用户以及`pg_stat_scan_tables`角色的成员执行；但`pg_truncate_visibility_map(relation regclass)`是个例外，它只能由超级用户执行。

F.32.2. 作者

Robert Haas <rhaas@postgresql.org>

F.33. postgres_fdw

postgres_fdw 模块提供外部数据包装器 postgres_fdw，可用于访问存储在外部 PostgreSQL 服务器中的数据。

本模块提供的功能与较旧的 dblink 模块在很大程度上重叠。但 postgres_fdw 为访问远程表提供了更透明且符合标准的语法，并且在许多情况下性能更好。

要准备通过 postgres_fdw 进行远程访问：

1. 安装 postgres_fdw 扩展，可使用 CREATE EXTENSION。
2. 使用 CREATE SERVER 创建外部服务器对象，用来表示每个要连接的远程数据库。将除 user 和 password 之外的连接信息指定为服务器对象的选项。
3. 对于每个需要获准访问各个外部服务器的数据库用户，使用CREATE USER MAPPING 创建用户映射。将要使用的远程用户名和密码指定为用户映射的 user 和 password 选项。
4. 对于每个要访问的远程表，使用CREATE FOREIGN TABLE 或 IMPORT FOREIGN SCHEMA 创建外部表。外部表的列必须与被引用的远程表匹配。不过，如果在外部表对象的选项中指定正确的远程名称，也可以使用与远程表不同的表名和/或列名。

现在，只需对外部表执行 SELECT，即可访问其底层远程表中存储的数据。也可以使用 INSERT、UPDATE、DELETE 或 COPY 修改远程表。（当然，在用户映射中指定的远程用户必须拥有执行这些操作的权限。）

请注意，postgres_fdw 当前不支持带有 ON CONFLICT DO UPDATE 子句的 INSERT 语句。不过，在省略唯一索引推断规范的前提下，支持 ON CONFLICT DO NOTHING 子句。还要注意，postgres_fdw 支持在分区表上执行的 UPDATE 语句所引发的行移动，但当前尚不能处理这样一种情况：为插入被移动行而选择的远程分区，同时也是之后将被更新的 UPDATE 目标分区。

通常建议将外部表的列声明为与被引用远程表的对应列具有完全相同的数据类型，并在适用时具有相同的排序规则。尽管 postgres_fdw 目前在按需执行数据类型转换方面相当宽容，但当类型或排序规则不匹配时，仍可能出现令人意外的语义异常，因为远程服务器对查询条件的解释可能与本地服务器不同。

请注意，外部表的声明可以比其底层远程表少一些列，或者使用不同的列顺序。与远程表列的匹配是按名称而不是按位置进行的。

F.33.1. postgres_fdw 的 FDW 选项

F.33.1.1. 连接选项

使用 postgres_fdw 外部数据包装器的外部服务器，可以使用与libpq 连接字符串所接受的相同选项，详见第 34.1.2 节，但以下选项不被允许：

- user 和 password（应改为在用户映射中指定）
- client_encoding（会根据本地服务器编码自动设置）
- fallback_application_name（始终设置为 postgres_fdw）

只有超级用户才能不使用密码认证连接到外部服务器，因此应始终为属于非超级用户的用户映射指定 password 选项。

F.33.1.2. 对象名称选项

这些选项可用于控制发送到远程 PostgreSQL 服务器的 SQL 语句中所使用的名称。当创建外部表时所用的名称与其底层远程表的名称不同时，就需要这些选项。

schema_name

该选项可为外部表指定，用于给出在远程服务器上为该外部表使用的模式名。如果省略，则使用外部表自身所在模式的名称。

table_name

该选项可为外部表指定，用于给出在远程服务器上为该外部表使用的表名。如果省略，则使用外部表自身的名称。

column_name

该选项可为外部表的某个列指定，用于给出在远程服务器上为该列使用的列名。如果省略，则使用该列自身的名称。

F.33.1.3. 代价估算选项

postgres_fdw 通过在远程服务器上执行查询来获取远程数据，因此，理想情况下，扫描外部表的估计代价应当等于在远程服务器上完成该操作的代价，再加上一些通信开销。获得这种估算最可靠的方法，是向远程服务器询问，再把开销加上去；但对于简单查询，为了取得代价估算而额外发送一次远程查询，可能并不划算。因此 postgres_fdw 提供以下选项来控制代价估算的方式：

use_remote_estimate

该选项可为外部表或外部服务器指定，用于控制 postgres_fdw 是否发出远程 EXPLAIN 命令来获取代价估算。外部表上的设置会覆盖其所属服务器的设置，但只对该表生效。默认值为 false。

fdw_startup_cost

该选项可为外部服务器指定，是一个数值，会被加到该服务器上任何外部表扫描的估计启动代价中。它表示建立连接、在远程端解析并规划查询等额外开销。默认值为 100。

fdw_tuple_cost

该选项可为外部服务器指定，是一个数值，用作该服务器上外部表扫描的每个元组的额外代价。它表示服务器之间数据传输的额外开销。可以增大或减小该数值，以反映到远程服务器更高或更低的网络延迟。默认值为 0.01。

当 use_remote_estimate 为真时，postgres_fdw 从远程服务器获取行数和代价估算，然后将 fdw_startup_cost 和 fdw_tuple_cost 加到代价估算中。当 use_remote_estimate 为假时，postgres_fdw 在本地执行行数和代价估算，然后再将 fdw_startup_cost 和 fdw_tuple_cost 加到代价估算中。除非有远程表统计信息的本地副本可用，否则这种本地估算不太可能非常准确。更新本地统计信息的方法，是在外部表上运行 ANALYZE；这样会扫描远程表，然后像对待本地表一样计算并存储统计信息。保留本地统计信息可以有效减少远程表每次查询的规划开销；但如果远程表经常更新，本地统计信息很快就会过时。

F.33.1.4. 远程执行选项

默认情况下，只有使用内置操作符和函数的 WHERE 子句才会被考虑在远程服务器上执行。涉及非内置函数的子句会在取回行之后在本地检查。如果这些函数在远程服务器上也可用，并且可以确

信其结果与本地相同，则将这类 WHERE 子句发送到远程端执行可以提高性能。可以使用以下选项控制此行为：

extensions

该选项是一个以逗号分隔的 PostgreSQL 扩展名称列表，这些扩展必须在本地和远程服务器上都已安装且版本兼容。属于列出扩展且不可变的函数和操作符，将被视为可下推到远程服务器执行。该选项只能为外部服务器指定，不能按表指定。

使用 extensions 选项时，确保所列扩展在本地和远程服务器上都存在且行为完全一致，属于用户自己的责任。否则，远程查询可能失败或出现意外行为。

fetch_size

该选项指定 postgres_fdw 在每次取回操作中应获取的行数。它可为外部表或外部服务器指定。表上指定的选项会覆盖服务器上指定的选项。默认值为 100。

F.33.1.5. 可更新性选项

默认情况下，所有使用 postgres_fdw 的外部表都被假定为可更新。这一点可以通过以下选项覆盖：

updatable

该选项控制 postgres_fdw 是否允许使用 INSERT、UPDATE 和 DELETE 命令修改外部表。它可为外部表或外部服务器指定。表级选项会覆盖服务器级选项。默认值为 true。

当然，如果远程表实际上不可更新，最终仍会报错。该选项的主要作用是允许在本地直接抛出错误，而无需查询远程服务器。但请注意，information_schema 视图会根据该选项的设置，将 postgres_fdw 外部表报告为可更新（或不可更新），而不会对远程服务器进行任何检查。

F.33.1.6. 导入选项

postgres_fdw 可以使用 IMPORT FOREIGN SCHEMA 导入外部表定义。该命令会在本地服务器上创建外部表定义，以匹配远程服务器上的表或视图。如果要导入的远程表列使用用户定义数据类型，则本地服务器必须存在同名且兼容的类型。

可使用以下选项（在 IMPORT FOREIGN SCHEMA 命令中给出）自定义导入行为：

import_collate

该选项控制从外部服务器导入的外部表定义中是否包含列的 COLLATE 选项。默认值为 true。如果远程服务器的排序规则名称集合与本地服务器不同，则可能需要关闭此选项；如果远程服务器运行在不同操作系统上，这种情况尤其可能发生。不过，如果这样做，导入表列的排序规则就存在与底层数据不匹配的严重风险，从而导致查询行为异常。

即使将此参数设置为 true，导入排序规则为远程服务器默认值的列仍可能有风险。这些列会以 COLLATE "default" 导入，这将选择本地服务器的默认排序规则，而它可能并不相同。

import_default

该选项控制从外部服务器导入的外部表定义中是否包含列的 DEFAULT 表达式。默认值为 false。如果启用此选项，需要警惕那些在本地服务器上的计算结果可能与远程服务器不同的默认值；nextval() 是常见的问题来源。如果导入的默认值表达式使用了本地不存在的函数或操作符，则整个 IMPORT 将失败。

import_not_null

该选项控制从外部服务器导入的外部表定义中是否包含列的 NOT NULL 约束。默认值为 true。

请注意，除 NOT NULL 之外的约束永远不会从远程表导入。虽然 PostgreSQL 确实支持在外部表上定义 CHECK 约束，但由于约束表达式在本地和远程服务器上可能求值不同，系统不会自动导入它们。此类行为不一致的CHECK 约束，可能导致查询优化中难以发现的错误。因此，如果希望导入CHECK 约束，必须手工完成，并应仔细核实每一个约束的语义。有关外部表上CHECK 约束处理方式的更多细节，请参见 CREATE FOREIGN TABLE。

作为其他表分区的表或外部表会被自动排除。分区表会被导入，除非它本身也是其他表的分区。由于所有数据都可以通过作为分区层次根的分区表访问，因此只导入分区表即可访问全部数据，而无需创建额外对象。

F.33.2. 连接管理

postgres_fdw 在首次执行使用与某个外部服务器关联的外部表的查询时，会建立到该外部服务器的连接。该连接会在同一会话中保留并供后续查询重用。如果使用多个用户标识（用户映射）访问该外部服务器，则会为每个用户映射建立一个连接。

F.33.3. 事务管理

在引用某个外部服务器上任意远程表的查询期间，如果当前本地事务尚未在该远程服务器上打开对应的事务，postgres_fdw 就会在该远程服务器上打开一个事务。本地事务提交或中止时，远程事务也会提交或中止。保存点也会通过创建对应的远程保存点进行类似管理。

当本地事务的隔离级别为 SERIALIZABLE 时，远程事务使用 SERIALIZABLE；否则使用 REPEATABLE READ 隔离级别。这一选择确保如果一个查询在远程服务器上执行多次表扫描，所有扫描都能获得快照一致的结果。其结果是，同一事务中的后续查询会看到来自远程服务器的相同数据，即使远程服务器由于其他活动正在发生并发更新。对于使用 SERIALIZABLE 或 REPEATABLE READ 隔离级别的本地事务，这种行为本来就符合预期；但对于 READ COMMITTED 本地事务，则可能令人意外。未来的 PostgreSQL 版本可能会修改这些规则。

请注意，postgres_fdw 当前不支持为两阶段提交预备远程事务。

F.33.4. 远程查询优化

postgres_fdw 会尽力优化远程查询，以减少从外部服务器传输的数据量。这是通过将查询的 WHERE 子句发送到远程服务器执行，以及不获取当前查询不需要的表列来实现的。为降低查询被错误执行的风险，只有当 WHERE 子句使用的所有数据类型、操作符和函数都是内置的，或属于外部服务器 extensions 选项列出的扩展时，才会将该子句发送到远程服务器。这类子句中的操作符和函数还必须是 IMMUTABLE。对于 UPDATE 或 DELETE 查询，postgres_fdw 会在查询中不存在无法发送到远程服务器的 WHERE 子句、没有本地连接操作、目标表上没有行级本地 BEFORE 或 AFTER 触发器，也没有来自父视图的 CHECK OPTION 约束时，尝试将整个查询发送到远程服务器以优化执行。在 UPDATE 中，为了降低查询被错误执行的风险，赋给目标列的表达式也必须只使用内置数据类型、IMMUTABLE 操作符或 IMMUTABLE 函数。

当 postgres_fdw 遇到同一外部服务器上的外部表之间的连接时，除非由于某种原因它认为分别从各表取回行会更高效，或者相关表引用受不同用户映射约束，否则会将整个连接发送到远程服务器。在发送 JOIN 子句时，它也会采取与前述 WHERE 子句相同的预防措施。

可以使用 EXPLAIN VERBOSE 查看实际发送给远程服务器执行的查询。

F.33.5. 远程查询执行环境

在 postgres_fdw 打开的远程会话中，search_path 参数会被设置为仅包含 pg_catalog，这样无需模式限定就只能看到内置对象。这对 postgres_fdw 自身生成的查询不是问题，因为它总是提供这种限定。然而，这可能会对那些通过远程表上的触发器或规则在远程服务器上执行的函数带来风险。例如，如果远程表实际上是一个视图，则该视图中使用的任何函数都会在受限的搜索路径下执行。建议在这类函数中对所有名称都写成带模式限定的形式，或者为这类函数附加 SET search_path 选项（见 CREATE FUNCTION），以建立其预期的搜索路径环境。

postgres_fdw 还会为远程会话设置以下参数：

- TimeZone 被设置为 UTC
- DateStyle 被设置为 ISO
- IntervalStyle 被设置为 postgres
- 对于 9.0 及以上的远程服务器，extra_float_digits 被设置为 3；对于更早版本，则被设置为 2

这些设置通常不像 search_path 那样容易出问题，但如果有需要，也可以通过函数的 SET 选项处理。

不建议通过修改这些参数的会话级设置来覆盖这种行为；这很可能导致 postgres_fdw 工作异常。

F.33.6. 跨版本兼容性

postgres_fdw 可用于最早追溯到 PostgreSQL 8.3 的远程服务器。只读能力可追溯到 8.1。不过有一个限制是，postgres_fdw 通常假定：如果外部表的 WHERE 子句中出现不可变的内置函数和操作符，那么把它们发送到远程服务器执行是安全的。因此，某个在远程服务器所属发行版本之后才加入的内置函数，可能仍会被发送到该远程服务器执行，从而导致 “function does not exist” 或类似错误。可以通过重写查询绕过这类失败，例如把外部表引用放入一个带 OFFSET 0 的子 SELECT 中，作为优化栅栏，并将有问题的函数或操作符放到子 SELECT 之外。

F.33.7. 示例

下面是使用 postgres_fdw 创建外部表的一个示例。首先安装扩展：

```
CREATE EXTENSION postgres_fdw;
```

然后使用 CREATE SERVER 创建外部服务器。在本示例中，希望连接到一台 PostgreSQL 服务器，它运行在主机 192.83.123.89 上并监听 5432 端口。要连接的数据库在远程服务器上名为 foreign_db：

```
CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host '192.83.123.89', port '5432', dbname 'foreign_db');
```

还需要用 CREATE USER MAPPING 定义一个用户映射，以标识在远程服务器上使用哪个角色：

```
CREATE USER MAPPING FOR local_user
        SERVER foreign_server
        OPTIONS (user 'foreign_user', password 'password');
```

现在可以通过 CREATE FOREIGN TABLE 创建外部表。在本例中，要访问远程服务器上的表 some_schema.some_table，其本地名称为foreign_table：

```
CREATE FOREIGN TABLE foreign_table (
        id integer NOT NULL,
        data text
)
        SERVER foreign_server
        OPTIONS (schema_name 'some_schema', table_name 'some_table');
```

必须确保在 CREATE FOREIGN TABLE 中声明的列的数据类型和其他属性，与实际远程表匹配。列名也必须匹配，除非为各列附加column_name 选项，指明它们在远程表中的名称。在许多情况下，使用IMPORT FOREIGN SCHEMA 优于手工构造外部表定义。

F.33.8. 作者

Shigeru Hanada <shigeru.hanada@gmail.com>

F.34. seg

该模块实现了一个用于表示线段或浮点区间的 `seg` 数据类型。`seg` 可以表示区间端点中的不确定性，因此特别适合表示实验室测量结果。

F.34.1. 原理

测量值的几何形态通常比数值连续体中的一个点更复杂。一次测量通常是该连续体上的一段，其边界多少有些模糊。测量结果之所以会呈现为区间，一方面是由于不确定性和随机性，另一方面也是因为被测值本身可能天然就是表示某种状态的区间，例如蛋白质的稳定温度范围。

凭常识也能看出，把这类数据存储为区间比存储为一对数字更方便。实际上，在大多数应用中，这样做甚至更高效。

再顺着这一常识往下想，边界的模糊性说明，使用传统数值数据类型会造成一定的信息损失。设想一下：你的仪器读数是 6.50，而你把这个读数输入到数据库。取出来时会得到什么？请看：

```
test=> select 6.50 :: float8 as "pH";
 pH
---
6.5
(1 row)
```

在测量领域，6.50 与 6.5 并不相同。有时这种差别至关重要。实验人员通常会记下（并发表）他们认为可靠的那些数位。6.50 实际上是一个模糊区间，它包含在更大、也更模糊的区间 6.5 之中；它们共享的特征（大概）只有中心点。我们当然不希望这类不同的数据项看起来却一样。

结论是什么？最好能有一种专门的数据类型，能够以任意可变的精度记录区间边界。这里所谓“可变”，是指每个数据元素都记录其自身的精度。

看看这个：

```
test=> select '6.25 .. 6.50'::seg as "pH";
          pH
------------
6.25 .. 6.50
(1 row)
```

F.34.2. 语法

区间的外部表示由一个或两个浮点数通过范围操作符（`..` 或 `...`）连接而成。另一种写法是指定中心点再加减一个偏差值。还可以存储可选的确定性指示符（<、> 或 ~）。（不过，所有内置操作符都会忽略这些确定性指示符。）表 F.26 概述了允许的表示形式；表 F.27 给出了一些示例。

在表 F.26 中，*x*、*y* 和 *delta* 表示浮点数。*x* 和 *y* 前面可以带确定性指示符，而*delta* 不可以。

表 F.26. seg 外部表示

x	单个值（零长度区间）
x `..` *y*	从*x* 到*y* 的区间
x `(+-)` *delta*	从*x* - *delta* 到 *x* + *delta* 的区间

x ..	下界为*x*、上界开放的区间
.. *x*	上界为*x*、下界开放的区间

表 F.27. 合法 seg 输入示例

5.0	创建一个零长度线段（如果你愿意，也可以把它看成一个点）
~5.0	创建一个零长度线段，并在数据中记录~。~会被seg 操作忽略，但会作为注释保留下来。
<5.0	在 5.0 处创建一个点。<会被忽略，但会作为注释保留下来。
>5.0	在 5.0 处创建一个点。> 会被忽略，但会作为注释保留下来。
5(+-)0.3	创建区间4.7 .. 5.3。注意，(+-) 记法不会被保留。
50 ..	所有大于或等于 50 的值
.. 0	所有小于或等于 0 的值
1.5e-2 .. 2E-2	创建区间0.015 .. 0.02
1 ... 2	与1...2、1 .. 2 或1..2 相同（范围操作符两侧的空格会被忽略）

由于...操作符在数据源中被广泛使用，因此允许把它作为 ..操作符的另一种写法。不幸的是，这会带来解析歧义：无法确定0...23 中的上界是 23 还是 0.23。解决办法是要求 seg 输入中的所有数字在小数点前至少有一位数字。

作为合理性检查，seg 会拒绝下界大于上界的区间，例如 5 .. 2。

F.34.3. 精度

seg 值在内部存储为一对 32 位浮点数。这意味着有效位数超过 7 位的数字会被截断。

有效位数不超过 7 位的数字会保留其原始精度。也就是说，如果你的查询返回 0.00，你可以确信末尾的零不是格式化造成的假象，而是反映了原始数据的精度。前导零的个数不影响精度：值 0.0067 被认为只有 2 位有效数字。

F.34.4. 用法

seg 模块包含适用于 seg 值的一个 GiST 索引操作符类。该 GiST 操作符类支持的操作符见表 F.28。

表 F.28. seg GiST 操作符

操作符	描述
[a, b] << [c, d]	[a, b] 完全位于 [c, d] 的左侧。也就是说，当 b < c 时，[a, b] << [c, d] 为真，否则为假。
[a, b] >> [c, d]	[a, b] 完全位于 [c, d] 的右侧。也就是说，当 a > d 时，[a, b] >> [c, d] 为真，否则为假。
[a, b] &< [c, d]	重叠或位于左侧 – 也可以更好地理解为“不会延伸到右侧”。当 b <= d 时为真。
[a, b] &> [c, d]	重叠或位于右侧 – 也可以更好地理解为“不会延伸到左侧”。当 a >= c 时为真。

操作符	描述
`[a, b] = [c, d]`	相同 – 线段 [a, b] 和 [c, d] 相同，即 a = c 且 b = d。
`[a, b] && [c, d]`	线段 [a, b] 和 [c, d] 重叠。
`[a, b] @> [c, d]`	线段 [a, b] 包含线段 [c, d]，即 a <= c 且 b >= d。
`[a, b] <@ [c, d]`	线段 [a, b] 包含于 [c, d] 中，即 a >= c 且 b <= d。

（在 PostgreSQL 8.2 之前，包含操作符 `@>` 和 `<@` 分别称为 `@` 和 `~`。这些名称仍然可用，但已弃用，最终将被删除。请注意，旧名称与核心几何数据类型以前采用的约定正好相反！）

还提供标准的 B-树操作符，例如

操作符	描述
`[a, b] < [c, d]`	小于
`[a, b] > [c, d]`	大于

这些操作符除了排序之外，在实际用途上没有太大意义。它们首先比较 (a) 与 (c)，如果相等，再比较 (b) 与 (d)。这在大多数情况下会产生相当好的排序效果；如果希望对这种类型使用 ORDER BY，这会很有用。

F.34.5. 注意

有关用法示例，请参见回归测试`sql/seg.sql`。

把 `(+-)` 转换为常规范围的机制，在确定边界的有效位数时并不完全准确。例如，如果结果区间包含 10 的幂，它会在下边界上多加一位：

```
postgres=> select '10(+-)1'::seg as seg;
      seg
---------
9.0 .. 11             -- should be: 9 .. 11
```

R 树索引的性能很大程度上取决于输入值的初始顺序。按 `seg` 列对输入表排序，可能会很有帮助；示例见脚本 `sort-segments.pl`。

F.34.6. 致谢

原作者：Gene Selkov, Jr. `<selkovjr@mcs.anl.gov>`，阿贡国家实验室数学与计算机科学部。

我首先要感谢 Joe Hellerstein 教授 (https://dsf.berkeley.edu/jmh/)，他为我阐明了 GiST (http://gist.cs.berkeley.edu/) 的要旨。我同样感谢过去和现在所有的 Postgres 开发者，他们使我得以创造自己的世界并在其中不受打扰地生活。我还要感谢阿贡实验室以及美国能源部，多年来始终如一地支持我的数据库研究。

F.35. sepgsql

sepgsql 是一个可加载模块，支持基于 SELinux 安全策略和安全标签的强制访问控制（MAC）。

> **警告**
>
> 当前实现存在重大限制，并不会对所有操作实施强制访问控制。详见第 F.35.7 节。

F.35.1. 概述

该模块与 SELinux 集成，在 PostgreSQL 通常提供的安全检查之外再增加一层安全检查。从 SELinux 的角度看，该模块使 PostgreSQL 能够充当用户空间对象管理器。由 DML 查询发起的每一次表或函数访问，都会依据系统安全策略进行检查。这种检查是对 PostgreSQL 常规 SQL 权限检查的补充。

SELinux 的访问控制决策通过安全标签作出，安全标签以 system_u:object_r:sepgsql_table_t:s0 这样的字符串表示。每个访问控制决策都涉及两个标签：尝试执行操作的主体标签，以及要执行该操作的对象标签。由于这些标签可以应用到任意类型的对象，数据库中存储对象的访问控制决策也可以与文件等其他类型对象一样，遵循同一套一般性标准；使用该模块时也确实如此。这样的设计旨在让集中式安全策略能够保护信息资产，而不受这些资产具体存储方式的影响。

SECURITY LABEL 语句允许为数据库对象分配安全标签。

F.35.2. 安装

sepgsql 只能在启用了 SELinux 的 Linux 2.6.28 或更高版本上使用。它在其他任何平台上都不可用。还需要 libselinux 2.1.10 或更高版本，以及 selinux-policy 3.9.13 或更高版本（尽管某些发行版可能会把所需规则回移植到较旧的策略版本中）。

可以使用 sestatus 命令检查 SELinux 的状态。典型显示如下：

```
$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 24
Policy from config file:        targeted
```

如果 SELinux 被禁用或尚未安装，则必须先安装并配置它，之后才能安装本模块。

要构建此模块，请将选项 --with-selinux 加入 PostgreSQL 的 configure 命令中。还要确保构建时已经安装 libselinux-devel RPM。

要使用此模块，必须将 sepgsql 包含在 shared_preload_libraries 参数中，该参数位于 postgresql.conf 内。如果以其他任何方式加载该模块，它都无法正确工作。模块加载后，应在每个数据库中执行 sepgsql.sql。这会安装安全标签管理所需的函数，并分配初始安全标签。

下面的示例展示了如何初始化一个全新的数据库集簇，并安装 sepgsql 函数和安全标签。请根据实际安装情况调整其中的路径：

```
$ export PGDATA=/path/to/data/directory
$ initdb
$ vi $PGDATA/postgresql.conf
  change
    #shared_preload_libraries = ''                # (change requires restart)
  to
    shared_preload_libraries = 'sepgsql'          # (change requires restart)
$ for DBNAME in template0 template1 postgres; do
    postgres --single -F -c exit_on_error=true $DBNAME \
      </usr/local/pgsql/share/contrib/sepgsql.sql >/dev/null
  done
```

请注意，具体会看到以下通知中的哪些，取决于所使用的 libselinux 和 selinux-policy 版本，可能是部分，也可能是全部：

```
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 33 has invalid object type db_blobs
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 36 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 37 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 38 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 39 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 40 has invalid object type db_language
```

这些消息没有害处，应予忽略。

如果安装过程无错误完成，此时就可以正常启动服务器。

F.35.3. 回归测试

由于 SELinux 的特性，运行 sepgsql 的回归测试需要额外执行若干配置步骤，其中部分步骤必须以 root 身份完成。普通的 make check 或 make installcheck 命令不会运行这些回归测试；必须完成配置，然后手工调用测试脚本。这些测试必须在已配置的 PostgreSQL 构建树的 contrib/sepgsql 目录中运行。虽然需要构建树，但这些测试是为已安装的服务器设计的，也就是说，它们类似于 make installcheck 而不是 make check。

首先，在一个可用数据库中设置 sepgsql，并遵循第 F.35.2 节中的说明。请注意，当前操作系统用户必须能够无需密码认证，以超级用户身份连接到该数据库。

第二，构建并安装回归测试所需的策略包。sepgsql-regtest 策略是一个专用策略包，提供了一组允许在回归测试期间使用的规则。它应从策略源文件 sepgsql-regtest.te 构建，构建过程通过使用 make 和由 SELinux 提供的 Makefile 完成。需要在本机上找到合适的 Makefile；下面展示的路径仅为示例。构建完成后，使用 semodule 命令安装该策略包，该命令会将提供的策略包装载到内核中。如果安装正确，semodule -l 应该把 sepgsql-regtest 列为可用策略包之一：

```
$ cd .../contrib/sepgsql
$ make -f /usr/share/selinux/devel/Makefile
$ sudo semodule -u sepgsql-regtest.pp
$ sudo semodule -l | grep sepgsql
sepgsql-regtest 1.07
```

第三，打开 sepgsql_regression_test_mode。出于安全原因，sepgsql-regtest 中的规则默认并未启用；sepgsql_regression_test_mode 参数会启用启动回归测试所需的规则。可以使用 setsebool 命令将其打开：

```
$ sudo setsebool sepgsql_regression_test_mode on
$ getsebool sepgsql_regression_test_mode
sepgsql_regression_test_mode --> on
```

第四，确认 shell 正在 unconfined_t 域中运行：

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

如有需要，请参阅第 F.35.8 节，了解如何调整工作域。

最后，运行回归测试脚本：

```
$ ./test_sepgsql
```

该脚本会尝试验证是否已正确完成全部配置步骤，然后运行 sepgsql 模块的回归测试。

完成测试后，建议关闭 sepgsql_regression_test_mode 参数：

```
$ sudo setsebool sepgsql_regression_test_mode off
```

也可以选择彻底移除 sepgsql-regtest 策略：

```
$ sudo semodule -r sepgsql-regtest
```

F.35.4. GUC 参数

sepgsql.permissive (boolean)

该参数使 sepgsql 无论系统设置如何都以宽容模式运行。默认值为关闭。该参数只能在 postgresql.conf 文件中或服务器命令行上设置。

当该参数打开时，sepgsql 会以宽容模式运行，即使 SELinux 整体处于强制模式也是如此。该参数主要用于测试。

sepgsql.debug_audit (boolean)

该参数会在不考虑系统策略设置的情况下启用审计消息输出。默认值为关闭，这意味着消息将按系统设置输出。

SELinux 的安全策略本身也有规则来控制是否记录特定的访问。默认情况下，访问违例会被记录，而被允许的访问不会被记录。

该参数会在不考虑系统策略的情况下，强制打开所有可能的日志记录。

F.35.5. 特性

F.35.5.1. 受控对象类

SELinux 的安全模型把全部访问控制规则描述为主体实体（通常是数据库客户端）与对象实体（例如数据库对象）之间的关系，这两者都由安全标签标识。如果尝试访问一个未带标签的对象，则会把它视为被赋予了 `unlabeled_t` 标签。

当前，`sepgsql` 允许为模式、表、列、序列、视图和函数分配安全标签。使用 `sepgsql` 时，安全标签会在受支持的数据库对象创建时自动分配。这个标签称为默认安全标签，其取值由系统安全策略决定；策略的输入包括创建者的标签、新对象父对象的标签，以及待创建对象的名称（如果需要）。

新数据库对象基本上会继承父对象的安全标签，但如果安全策略中存在称为类型转换规则的特殊规则，则可能会应用不同的标签。对于模式，父对象是当前数据库；对于表、序列、视图和函数，父对象是其所在模式；对于列，父对象是其所在表。

F.35.5.2. DML 权限

对于表，会根据语句类型，对所有被引用的目标表检查 `db_table:select`、`db_table:insert`、`db_table:update` 或 `db_table:delete`；此外，对于其列在 `WHERE` 或 `RETURNING` 子句中被引用的所有表，以及作为 `UPDATE` 数据源的所有表等，还会检查 `db_table:select`。

每个被引用的列也都会检查列级权限。`db_column:select` 不仅会对通过 `SELECT` 读取的列进行检查，也会对其他 DML 语句中被引用的列进行检查；对于被 `UPDATE` 或 `INSERT` 修改的列，还会检查 `db_column:update` 或 `db_column:insert`。

例如，考虑：

```
UPDATE t1 SET x = 2, y = func1(y) WHERE z = 100;
```

这里会对 `t1.x` 检查 `db_column:update`，因为它被更新；会对 `t1.y` 检查 `db_column:{select update}`，因为它既被更新也被引用；会对 `t1.z` 检查 `db_column:select`，因为它只是被引用。表级别还会检查 `db_table:{select update}`。

对于序列，当使用 `SELECT` 引用序列对象时，会检查 `db_sequence:get_value`；但请注意，当前不会检查执行相应函数（如 `lastval()`）的权限。

对于视图，会先检查 `db_view:expand`，然后对由该视图展开得到的各个对象分别检查所需的其他权限。

对于函数，当用户试图在查询中执行某个函数，或通过快速路径调用执行某个函数时，会检查 `db_procedure:{execute}`。如果该函数是受信任过程，还会检查 `db_procedure:{entrypoint}` 权限，以判定它是否可以充当受信任过程的入口点。

要访问任何模式对象，都需要在其所在模式上具有 `db_schema:search` 权限。当对象在引用时未带模式限定时，不具备该权限的模式不会被搜索（就像用户在该模式上没有 `USAGE` 权限一样）。如果使用了显式模式限定，而用户在所指定模式上又没有所需权限，则会报错。

客户端必须被允许访问所有被引用的表和列，即使这些表和列最初来自随后被展开的视图也是如此，这样我们才能在不受表内容引用方式影响的情况下，应用一致的访问控制规则。

默认数据库权限系统允许数据库超级用户使用 DML 命令修改系统目录，也允许引用或修改 TOAST 表。启用 `sepgsql` 后，这些操作都会被禁止。

F.35.5.3. DDL 权限

SELinux 为每种对象类型定义了若干权限，用于控制常见操作，例如创建、修改、删除以及重设安全标签。此外，若干对象类型还具有特殊权限，用于控制它们特有的操作，例如在特定模式中添加或删除名称项。

创建新的数据库对象需要 `create` 权限。SELinux 会根据客户端的安全标签以及为新对象拟定的安全标签授予或拒绝该权限。在某些情况下，还需要额外权限：

- CREATE DATABASE 还需要对源数据库或模板数据库具有 `getattr` 权限。
- 创建模式对象还需要在父模式上具有 `add_name` 权限。
- 创建表还需要有权限创建每个单独的表列，就好像每个表列都是独立的顶层对象一样。
- 创建被标记为 `LEAKPROOF` 的函数还需要 `install` 权限。（对现有函数设置 `LEAKPROOF` 时同样会检查该权限。）

执行 `DROP` 命令时，会对将被移除的对象检查 `drop` 权限。通过 `CASCADE` 间接删除的对象同样会检查权限。删除位于特定模式中的对象（表、视图、序列和过程）还需要该模式上的 `remove_name` 权限。

执行 `ALTER` 命令时，会针对被修改的对象检查 `setattr` 权限，但表的索引或触发器等附属对象除外，这类情况改为在父对象上检查权限。在某些情况下，还需要额外权限：

- 将对象移动到新模式还需要在旧模式上具有 `remove_name` 权限，并在新模式上具有 `add_name` 权限。
- 在函数上设置 `LEAKPROOF` 属性需要 `install` 权限。
- 在对象上使用 SECURITY LABEL 还需要同时具备与其旧安全标签组合的对象 `relabelfrom` 权限，以及与其新安全标签组合的对象 `relabelto` 权限。（在安装了多个标签提供者且用户尝试设置一个不由 SELinux 管理的安全标签时，这里本应只检查 `setattr`。由于实现限制，当前尚未这样做。）

F.35.5.4. 受信任过程

受信任过程类似于安全定义器函数或 setuid 命令。SELinux 提供了一项功能，允许受信任代码以不同于客户端的安全标签运行，通常用于以高度受控的方式访问敏感数据（例如可以省略某些

行，或降低已存储值的精度）。函数是否充当受信任过程，由其安全标签和操作系统安全策略控制。例如：

```
postgres=# CREATE TABLE customer (
               cid     int primary key,
               cname   text,
               credit  text
           );
CREATE TABLE
postgres=# SECURITY LABEL ON COLUMN customer.credit
               IS 'system_u:object_r:sepgsql_secret_table_t:s0';
SECURITY LABEL
postgres=# CREATE FUNCTION show_credit(int) RETURNS text
             AS 'SELECT regexp_replace(credit, ''-[0-9]+$'', ''-xxxx'', ''g'')
                         FROM customer WHERE cid = $1'
           LANGUAGE sql;
CREATE FUNCTION
postgres=# SECURITY LABEL ON FUNCTION show_credit(int)
               IS 'system_u:object_r:sepgsql_trusted_proc_exec_t:s0';
SECURITY LABEL
```

上述操作应由管理用户执行。

```
postgres=# SELECT * FROM customer;
ERROR:  SELinux: security policy violation
postgres=# SELECT cid, cname, show_credit(cid) FROM customer;
 cid | cname  |     show_credit
-----+--------+---------------------
   1 | taro   | 1111-2222-3333-xxxx
   2 | hanako | 5555-6666-7777-xxxx
(2 rows)
```

在这种情况下，普通用户不能直接引用 `customer.credit`，但受信任过程 `show_credit` 允许用户打印客户信用卡号，并对其中部分数字做掩码处理。

F.35.5.5. 动态域转换

如果安全策略允许，可以利用 SELinux 的动态域转换功能，将客户端进程（即客户端域）的安全标签切换到新的上下文。客户端域需要 `setcurrent` 权限，以及从旧域切换到新域的 `dyntransition` 权限。

动态域转换需要谨慎考虑，因为它允许用户按自己的意愿切换标签，因而也能切换其权限，而不是像受信任过程那样由系统强制规定。因此，只有当它被用于切换到一个权限集合少于原始域的新域时，`dyntransition` 权限才被认为是安全的。例如：

```
regression=# select sepgsql_getcon();
                  sepgsql_getcon
-----------------------------------------------------
 unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-s0:c1.c4');
 sepgsql_setcon
----------------
 t
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-s0:c1.c1023');
ERROR:  SELinux: security policy violation
```

在上面的例子中，允许从较大的 MCS 范围 `c1.c1023` 切换到较小的范围 `c1.c4`，但不允许切换回去。

动态域转换与受信任过程的组合支持一种很有意思的用例，它契合连接池软件的典型进程生命周期。即使连接池软件本身不允许运行大多数 SQL 命令，也可以让它在受信任过程中调用 `sepgsql_setcon()` 函数来切换客户端的安全标签；该过程应要求某种凭据，以授权切换客户端标签的请求。此后，这个会话就将拥有目标用户而非连接池软件的权限。之后，连接池软件还可以再次在具有适当权限检查的受信任过程中，以 `NULL` 参数调用 `sepgsql_setcon()`，从而恢复这一安全标签变更。这里的关键在于，真正有权修改生效安全标签的只有受信任过程，而且它只会在获得适当凭据时这样做。当然，为了安全运行，凭据存储（表、过程定义或其他载体）必须防止未授权访问。

F.35.5.6. 杂项

我们一律拒绝 LOAD 命令，因为装入任意模块都可能轻易绕过安全策略的强制执行。

F.35.6. sepgsql 函数

表 F.29 展示了可用函数。

表 F.29. sepgsql 函数

`sepgsql_getcon() returns text`	返回客户端域，也就是客户端当前的安全标签。
`sepgsql_setcon(text) returns bool`	如果安全策略允许，将当前会话的客户端域切换到新域。它也接受 `NULL` 输入，表示请求切换到客户端的原始域。
`sepgsql_mcstrans_in(text) returns text`	如果 mcstrans 守护进程正在运行，则把给定的限定格式 MLS/MCS 范围转换为原始格式。
`sepgsql_mcstrans_out(text) returns text`	如果 mcstrans 守护进程正在运行，则把给定的原始 MLS/MCS 范围转换为限定格式。
`sepgsql_restorecon(text) returns bool`	为当前数据库中的所有对象设置初始安全标签。参数可以为 NULL，也可以是一个 specfile 的名称，用来替代系统默认文件。

F.35.7. 限制

数据定义语言（DDL）权限

由于实现限制，某些 DDL 操作不会检查权限。

数据控制语言（DCL）权限

由于实现限制，DCL 操作不会检查权限。

行级访问控制

PostgreSQL 支持行级访问，但 `sepgsql` 不支持。

隐蔽通道

`sepgsql` 不会试图隐藏某个对象的存在，即使用户无权引用它也是如此。例如，即便我们无法取得一个不可见对象的内容，也仍然可以从主键冲突、外键违例等结果中推断出它的存在。绝密表的存在无法隐藏；我们只希望隐藏其内容。

F.35.8. 外部资源

SE-PostgreSQL Introduction[5]

该 wiki 页面提供了简要概述，并介绍了安全设计、体系结构、管理以及未来特性。

SELinux User's and Administrator's Guide[6]

该文档提供了在系统上管理 SELinux 所需的广泛知识。它主要聚焦于 Red Hat 操作系统，但并不限于此。

Fedora SELinux FAQ[7]

该文档回答了关于 SELinux 的常见问题。它主要聚焦于 Fedora，但并不限于 Fedora。

F.35.9. 作者

KaiGai Kohei <kaigai@ak.jp.nec.com>

[5] https://wiki.postgresql.org/wiki/SEPostgreSQL
[6] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/index
[7] https://fedoraproject.org/wiki/SELinux_FAQ

F.36. spi

spi 模块提供了若干使用服务器编程接口（SPI）和触发器的实用示例。虽然这些函数本身也有一定的独立用途，但把它们作为可按自身需求修改的示例来看更有价值。这些函数足够通用，可以用于任何表，但在创建触发器时必须指定表名和字段名（如下所述）。

下文描述的每组函数都作为一个可单独安装的扩展提供。

F.36.1. refint – 用于实现引用完整性的函数

`check_primary_key()` 和 `check_foreign_key()` 用于检查外键约束。（当然，这项功能早已被内置的外键机制取代，不过该模块作为示例仍然有用。）

`check_primary_key()` 检查引用表。使用时，在引用其他表的表上创建一个使用该函数的 `BEFORE INSERT OR UPDATE` 触发器。触发器参数依次为：引用表中构成外键的列名、被引用表名，以及被引用表中构成主键/唯一键的列名。若要处理多个外键，请为每个引用分别创建一个触发器。

`check_foreign_key()` 检查被引用表。使用时，在被其他表引用的表上创建一个使用该函数的 `BEFORE DELETE OR UPDATE` 触发器。触发器参数依次为：该函数需要检查的引用表数量、发现引用键时采取的动作（`cascade` 表示删除引用行，`restrict` 表示若存在引用键则中止事务，`setnull` 表示将引用键字段设为空）、被引用表中构成主键/唯一键的列名，然后是引用表名和列名（按第一个参数指定的引用表数量重复提供）。请注意，主键/唯一键列应标记为 NOT NULL，并具有唯一索引。

`refint.example` 中有示例。

F.36.2. timetravel – 实现时间旅行的函数

很久以前，PostgreSQL 曾内置时间旅行功能，为每个元组保存插入和删除时间。可以使用这些函数模拟这一功能。要使用这些函数，必须为表添加两个`abstime` 类型的列，以存储元组插入的时间（start_date）和更改/删除的时间（stop_date）：

```
CREATE TABLE mytab (
        ...             ...
        start_date      abstime,
        stop_date       abstime
        ...             ...
);
```

这些列可以任意命名，不过在这里我们将它们称为 start_date 和 stop_date。

插入新行时，start_date 通常应设置为当前时间，stop_date 应设置为`infinity`。如果插入的数据在这些列中包含空值，触发器会自动将其替换为上述值。一般来说，只有在重新载入转储数据时，才应向这些列中插入显式的非空数据。

stop_date 等于 `infinity` 的元组“当前有效”，可以修改。stop_date 为有限值的元组则不能再修改，触发器会阻止这种操作。（如果需要这样做，可以按下文所示关闭时间旅行。）

对于可修改的行，更新时只会将被更新元组的 stop_date 改为当前时间，并插入一个包含修改后数据的新元组。新元组的 start_date 将设置为当前时间，stop_date 设置为`infinity`。

删除操作实际上不会删除元组，而只会将其 stop_date 设置为当前时间。

要查询“当前有效”的元组，请在查询的 WHERE 条件中加入`stop_date = 'infinity'`。（你可能希望将其放入视图中。）类似地，通过为 start_date 和 stop_date 设置合适的条件，也可以查询在过去任何时间有效的元组。

`timetravel()` 是支持此行为的通用触发器函数。在每个使用时间旅行的表上，使用此函数创建一个`BEFORE INSERT OR UPDATE OR DELETE` 触发器。指定两个触发器参数：start_date 和 stop_date 列的实际名称。还可以选择指定一到三个额外参数，它们必须引用`text` 类型的列。触发器会在 INSERT 时将当前用户名存入其中的第一列，在 UPDATE 时存入第二列，在 DELETE 时存入第三列。

`set_timetravel()` 允许为表打开或关闭时间旅行。`set_timetravel('mytab', 1)` 会为表 `mytab` 打开 TT。`set_timetravel('mytab', 0)` 会为表 `mytab` 关闭 TT。两种情况都会报告原来的状态。在 TT 关闭时，可以自由修改 start_date 和 stop_date 列。请注意，开关状态仅作用于当前数据库会话；新会话总是从所有表的 TT 都打开的状态开始。

`get_timetravel()` 返回表的 TT 状态，而不改变它。

`timetravel.example` 中有一个示例。

F.36.3. autoinc – 用于字段自动递增的函数

`autoinc()` 是一个将序列的下一个值存入整数字段的触发器。它与内置的“serial 列”功能有一些重叠，但并不相同：`autoinc()` 会覆盖插入时试图为该字段指定其他值的操作，而且还可以选择在更新时也递增该字段。

使用时，创建一个使用该函数的 `BEFORE INSERT`（或者可选的 `BEFORE INSERT OR UPDATE`）触发器。指定两个触发器参数：要修改的整型列名，以及提供这些值的序列对象名。（实际上，如果你希望更新多个自动递增列，也可以指定任意多组这样的名称对。）

`autoinc.example` 中有一个示例。

F.36.4. insert_username – 用于跟踪谁修改了表的函数

`insert_username()` 是一个将当前用户名存入文本字段的触发器。它有助于跟踪表中某一特定行最后一次是由谁修改的。

使用时，创建一个使用该函数的 `BEFORE INSERT` 和/或 `UPDATE` 触发器。指定一个触发器参数：要修改的文本列名。

`insert_username.example` 中有一个示例。

F.36.5. moddatetime – 用于跟踪最后修改时间的函数

`moddatetime()` 是一个将当前时间存入 `timestamp` 字段的触发器。它有助于跟踪表中某一特定行最后一次被修改的时间。

使用时，创建一个使用该函数的 `BEFORE UPDATE` 触发器。指定一个触发器参数：要修改的列名。该列必须是 `timestamp` 或 `timestamp with time zone` 类型。

`moddatetime.example` 中有一个示例。

F.37. sslinfo

`sslinfo` 模块提供关于当前客户端在连接到 PostgreSQL 时所提供 SSL 证书的信息。如果当前连接没有使用 SSL，则该模块没有用处（大多数函数将返回 NULL）。

除非安装时使用`--with-openssl` 配置，否则此扩展根本无法构建。

F.37.1. 提供的函数

`ssl_is_used() returns boolean`

如果当前到服务器的连接使用 SSL，则返回 true，否则返回 false。

`ssl_version() returns text`

返回 SSL 连接使用的协议名称（例如 TLSv1.0、TLSv1.1 或 TLSv1.2）。

`ssl_cipher() returns text`

返回 SSL 连接所用密码套件的名称（例如 DHE-RSA-AES256-SHA）。

`ssl_client_cert_present() returns boolean`

如果当前客户端已向服务器提供了有效的 SSL 客户端证书，则返回 true，否则返回 false。（服务器可能配置为要求客户端证书，也可能没有这样配置。）

`ssl_client_serial() returns numeric`

返回当前客户端证书的序列号。证书序列号与证书颁发者的组合可保证唯一标识一个证书（但不能保证唯一标识其所有者 — 所有者应当定期更换其密钥，并从颁发者处获取新证书）。

因此，如果你运行自己的 CA，并且只允许服务器接受由该 CA 签发的证书，那么序列号就是识别用户的最可靠手段（尽管并不便于记忆）。

`ssl_client_dn() returns text`

返回当前客户端证书的完整主题，并将字符数据转换为当前数据库编码。假定如果你在证书名称中使用了非 ASCII 字符，你的数据库也能表示这些字符。如果你的数据库使用 SQL_ASCII 编码，名称中的非 ASCII 字符将表示为 UTF-8 序列。

结果类似于 `/CN=Somebody /C=Some country/O=Some organization`。

`ssl_issuer_dn() returns text`

返回当前客户端证书的完整颁发者名称，并将字符数据转换为当前数据库编码。编码转换的处理方式与 `ssl_client_dn` 相同。

该函数的返回值与证书序列号的组合可唯一标识该证书。

只有在服务器的证书颁发机构文件中包含多个受信任的 CA 证书，或者该 CA 颁发了一些中间 CA 证书时，此函数才真正有用。

`ssl_client_dn_field(fieldname text) returns text`

该函数返回证书主题中指定字段的值；如果该字段不存在，则返回 NULL。字段名是字符串常量，它们使用 OpenSSL 对象数据库转换为 ASN1 对象标识符。可接受的值如下：

```
commonName (alias CN)
surname (alias SN)
name
givenName (alias GN)
countryName (alias C)
localityName (alias L)
stateOrProvinceName (alias ST)
organizationName (alias O)
organizationalUnitName (alias OU)
title
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress
```

除 `commonName` 外，所有这些字段都是可选的。究竟包含哪些字段、哪些字段不包含，完全取决于你的 CA 策略。不过，这些字段的含义由 X.500 和 X.509 标准严格定义，因此你不能随意赋予它们任意含义。

`ssl_issuer_field(fieldname text) returns text`

与 `ssl_client_dn_field` 相同，只是针对证书颁发者而不是证书主题。

`ssl_extension_info() returns setof record`

提供客户端证书扩展的信息：扩展名、扩展值，以及该扩展是否为关键扩展。

F.37.2. 作者

Victor Wagner <vitus@cryptocom.ru>, Cryptocom LTD

Dmitry Voronin <carriingfate92@yandex.ru>

Cryptocom OpenSSL 开发组的电子邮件：<openssl@cryptocom.ru>

F.38. tablefunc

tablefunc 模块包含多种返回表（即多行结果）的函数。这些函数本身很有用，也可作为如何编写返回多行的 C 函数的示例。

F.38.1. 提供的函数

表 F.30 总结了 tablefunc 模块提供的函数。

表 F.30. tablefunc 函数

函数	返回值	描述
normal_rand(int numvals, float8 mean, float8 stddev)	setof float8	生成一组正态分布的随机值。
crosstab(text sql)	setof record	生成一个“透视表”，其中包含行名以及 *N* 个值列，其中 *N* 由调用查询中指定的行类型决定。
crosstab*N*(text sql)	setof table_crosstab_*N*	生成一个“透视表”，其中包含行名以及 *N* 个值列。crosstab2、crosstab3 和 crosstab4 是预定义的，但也可以按下文所述创建额外的 crosstab*N* 函数。
crosstab(text source_sql, text category_sql)	setof record	生成一个“透视表”，其值列由第二个查询指定。
crosstab(text sql, int N)	setof record	这是 crosstab(text) 的过时版本。参数 *N* 现已被忽略，因为值列的数量始终由调用查询决定。
connectby(text relname, text keyid_fld, text parent_keyid_fld [, text orderby_fld], text start_with, int max_depth [, text branch_delim])	setof record	生成层次树结构的表示形式。

F.38.1.1. normal_rand

```
normal_rand(int numvals, float8 mean, float8 stddev) returns setof float8
```

normal_rand 生成一组正态分布（高斯分布）的随机值。

numvals 是该函数要返回的值的个数。*mean* 是这些值所服从的正态分布的均值，*stddev* 是这些值所服从的正态分布的标准偏差。

例如，下面这个调用请求生成 1000 个值，均值为 5，标准偏差为 3：

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
     normal_rand
----------------------
     1.56556322244898
     9.10040991424657
     5.36957140345079
   -0.369151492880995
    0.283600703686639
       .
       .
       .
     4.82992125404908
     9.71308014517282
     2.49639286969028
(1000 rows)
```

F.38.1.2. `crosstab(text)`

```
crosstab(text sql)
crosstab(text sql, int N)
```

该 `crosstab` 函数用于产生“透视”显示形式，其中数据沿页面横向排列，而不是纵向排列。例如，我们可能有这样的数据

```
row1    val11
row1    val12
row1    val13
...
row2    val21
row2    val22
row2    val23
...
```

并希望将它们显示成这样

```
row1    val11   val12   val13   ...
row2    val21   val22   val23   ...
...
```

该 `crosstab` 函数接受一个文本参数，该参数是一个 SQL 查询，用于产生按第一种方式格式化的原始数据，并生成一个按第二种方式格式化的表。

`sql` 参数是一个产生源数据集的 SQL 语句。该语句必须返回一个 `row_name` 列、一个 `category` 列和一个 `value` 列。*`N`* 是一个过时参数，即使提供也会被忽略（以前它必须与输出值列的个数匹配，但现在这一点由调用查询决定）。

例如，给出的查询可能会产生如下结果：

```
 row_name    cat    value
----------+-------+-------
  row1      cat1    val1
  row1      cat2    val2
  row1      cat3    val3
  row1      cat4    val4
  row2      cat1    val5
  row2      cat2    val6
  row2      cat3    val7
  row2      cat4    val8
```

crosstab 函数被声明为返回 setof record，因此输出列的实际名称和类型必须在调用 SELECT 语句的 FROM 子句中定义，例如：

```
SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2 text);
```

这个示例会产生如下结果：

```
           <== value  columns  ==>
 row_name   category_1   category_2
----------+------------+------------
  row1        val1         val2
  row2        val5         val6
```

FROM 子句必须把输出定义为一个 row_name 列（其数据类型与 SQL 查询的第一列结果相同），其后跟着 N 个 value 列（它们的数据类型都与 SQL 查询的第三列结果相同）。可以按需设置任意数量的输出值列，输出列的名称也可自行决定。

crosstab 函数会为输入行中具有相同 row_name 值的每个连续分组生成一行输出。它会用这些行中的 value 字段从左到右填充输出的 value 列。如果某个分组中的行数少于输出 value 列的个数，多余的输出列会填充为空值；如果行数更多，多出来的输入行会被跳过。

实际上，SQL 查询应始终指定 ORDER BY 1,2，以确保输入行按正确顺序排列，也就是使具有相同 row_name 的值聚在一起，并在行内正确排序。注意，crosstab 本身并不会关注查询结果的第二列；该列只是为了排序而存在，用来控制第三列的值在页面上横向排列的顺序。

下面是一个完整示例：

```
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);

 row_name | category_1 | category_2 | category_3
----------+------------+------------+------------
 test1    | val2       | val3       |
 test2    | val6       | val7       |
(2 rows)
```

可以定义一个在其定义中固定了所需输出行类型的自定义 crosstab 函数，以避免每次都必须写出用于定义输出列的 `FROM` 子句。下一节会介绍这种做法。另一种可能性是在视图定义中嵌入所需的 `FROM` 子句。

> **注意**
>
> 另见 psql 中的 `\crosstabview` 命令，它提供了与 `crosstab()` 类似的功能。

F.38.1.3. `crosstabN(text)`

```
crosstabN(text sql)
```

`crosstabN` 函数展示了如何为通用的 `crosstab` 函数设置自定义包装器，这样就不必在调用的 `SELECT` 查询中写出列名和类型。`tablefunc` 模块包含 `crosstab2`、`crosstab3` 和 `crosstab4`，其输出行类型定义为：

```
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
        .
        .
        .
    category_N TEXT
);
```

因此，当输入查询产生类型为 text 的 row_name 列和 value 列，并且需要 2、3 或 4 个输出值列时，这些函数可以直接使用。除此之外，它们在其他方面的行为与上文描述的通用 crosstab 函数完全相同。

例如，前一节中的示例也可以这样写：

```
SELECT *
FROM crosstab3(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2');
```

这些函数主要用于说明。也可以基于底层的 crosstab() 函数创建自己的返回类型和函数。做法有两种：

- 创建一个描述所需输出列的复合类型，类似于 contrib/tablefunc/tablefunc--1.0.sql 中的示例。然后定义一个具有唯一名称的函数，该函数接受单个 text 参数并返回 setof your_type_name，但底层链接到同一个 crosstab C 函数。例如，如果源数据生成的行名是 text，值是 float8，而希望有 5 个值列：

```
CREATE TYPE my_crosstab_float8_5_cols AS (
    my_row_name text,
    my_category_1 float8,
    my_category_2 float8,
    my_category_3 float8,
    my_category_4 float8,
    my_category_5 float8
);

CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
    RETURNS setof my_crosstab_float8_5_cols
    AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

- 使用 OUT 参数隐式定义返回类型。同一个示例也可以这样实现：

```
CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
    IN text,
    OUT my_row_name text,
    OUT my_category_1 float8,
    OUT my_category_2 float8,
    OUT my_category_3 float8,
    OUT my_category_4 float8,
    OUT my_category_5 float8)
  RETURNS setof record
  AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

F.38.1.4. crosstab(text, text)

```
crosstab(text source_sql, text category_sql)
```

crosstab 单参数形式的主要限制在于，它会把同一组中的所有值一视同仁，并把每个值插入第一个可用列中。如果希望值列对应于特定的数据类别，而某些分组可能没有某些类别的数据，这种形式就不太适用了。crosstab 的双参数形式通过提供一个与输出列对应的显式类别列表来处理这种情况。

source_sql 是一个产生源数据集的 SQL 语句。该语句必须返回一个 row_name 列、一个 category 列和一个 value 列。它也可以有一个或多个“额外”列。row_name 列必须位于第一列。category 列和 value 列必须是最后两列，并且顺序必须如此。位于 row_name 和 category 之间的任何列都被视为“额外”。对于具有相同 row_name 值的所有行，这些“额外”列预期应当相同。

例如，*source_sql* 可能会产生如下结果：

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;

 row_name    extra_col   cat    value
----------+-----------+-----+---------
  row1         extra1    cat1    val1
  row1         extra1    cat2    val2
  row1         extra1    cat4    val4
  row2         extra2    cat1    val5
  row2         extra2    cat2    val6
  row2         extra2    cat3    val7
  row2         extra2    cat4    val8
```

category_sql 是一个生成类别集合的 SQL 语句。该语句必须只返回一列。它必须至少生成一行，否则会产生错误。此外，它不得生成重复值，否则会产生错误。*category_sql* 可以是这样的：

```
SELECT DISTINCT cat FROM foo ORDER BY 1;
    cat
  -------
    cat1
    cat2
    cat3
    cat4
```

crosstab 函数被声明为返回 setof record，因此输出列的实际名称和类型必须在调用 SELECT 语句的 FROM 子句中定义，例如：

```
SELECT * FROM crosstab('...', '...')
    AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4 text);
```

这将产生如下结果：

```
                  <==  value  columns   ==>
row_name   extra   cat1   cat2   cat3   cat4
---------+-------+------+------+------+------
  row1     extra1  val1   val2          val4
  row2     extra2  val5   val6   val7   val8
```

FROM 子句必须定义数量和数据类型都正确的输出列。如果 *source_sql* 查询结果有 *N* 列，那么其中前 *N*-2 列必须与前 *N*-2 个输出列相匹配。其余输出列必须具有 *source_sql* 查询结果最后一列的数据类型，并且这些输出列的数量必须与 *category_sql* 查询结果的行数完全相同。

crosstab 函数会为输入行中具有相同 row_name 值的每个连续分组生成一行输出。输出的 row_name 列以及任何“额外”列都从该分组的第一行复制而来。输出的 value 列则使用那些具有匹配 category 值的行中的 value 字段来填充。如果某一行的 category 与 *category_sql* 查询的任何输出都不匹配，那么它的 value 会被忽略。那些匹配类别未出现在该分组任何输入行中的输出列会填充为空值。

实际上，*source_sql* 查询应始终指定 ORDER BY 1，以确保具有相同 row_name 的值被聚在一起。不过，分组内类别的顺序并不重要。另外，必须确保 *category_sql* 查询输出的顺序与指定的输出列顺序一致。

下面给出两个完整示例：

```
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
  'select year, month, qty from sales order by 1',
  'select m from generate_series(1,12) m'
) as (
  year int,
  "Jan" int,
  "Feb" int,
  "Mar" int,
  "Apr" int,
  "May" int,
  "Jun" int,
  "Jul" int,
  "Aug" int,
  "Sep" int,
  "Oct" int,
  "Nov" int,
  "Dec" int
);
 year | Jan  | Feb  | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov  | Dec
------+------+------+-----+-----+-----+-----+-----+-----+-----+-----+------+------
 2007 | 1000 | 1500 |     |     |     |     | 500 |     |     |     | 1500 | 2000
 2008 | 1000 |      |     |     |     |     |     |     |     |     |      |
(2 rows)
```

```
CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March 2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(
  'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
  'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
AS
(
       rowid text,
       rowdt timestamp,
       temperature int4,
       test_result text,
       test_startdate timestamp,
       volts float8
);
 rowid |          rowdt           | temperature | test_result |      test_startdate      |
 volts
-------+--------------------------+-------------+-------------+--------------------------+-----
---
 test1 | Sat Mar 01 00:00:00 2003 |          42 | PASS        |                          | 2.
6987
 test2 | Sun Mar 02 00:00:00 2003 |          53 | FAIL        | Sat Mar 01 00:00:00 2003 | 3.
1234
(2 rows)
```

可以创建预定义函数，以避免在每个查询中都写出结果列的名称和类型。请参见前一节中的示例。这种形式的 crosstab 所对应的底层 C 函数名为 crosstab_hash。

F.38.1.5. connectby

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
          [, text orderby_fld ], text start_with, int max_depth
          [, text branch_delim ])
```

connectby 函数会显示存储在表中的层次数据。该表必须有一个唯一标识各行的键字段，以及一个引用每一行的父行（如果有）的父键字段。connectby 可以显示从任意一行开始向下展开的子树。

表 F.31 解释了这些参数。

表 F.31. connectby 参数

参数	描述
relname	源关系名称
keyid_fld	键字段名称
parent_keyid_fld	父键字段名称
orderby_fld	用于对同级节点排序的字段名称（可选）
start_with	起始行的键值
max_depth	向下遍历的最大深度，零表示深度不受限制
branch_delim	在分支输出中分隔各键值的字符串（可选）

键字段和父键字段可以是任意数据类型，但它们必须是同一类型。注意，无论键字段的类型是什么，*start_with* 值都必须作为文本字符串输入。

connectby 函数被声明为返回 setof record，因此输出列的实际名称和类型必须在调用 SELECT 语句的 FROM 子句中定义，例如：

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
    AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

前两个输出列用于当前行的键和其父行的键，它们必须与该表键字段的类型匹配。第三个输出列表示树中的深度，必须是 integer 类型。如果给出了 *branch_delim* 参数，下一个输出列就是分支路径显示，必须是 text 类型。最后，如果给出了 *orderby_fld* 参数，最后一个输出列就是一个序列号，必须是 integer 类型。

“branch” 输出列显示了到达当前行所经过的键路径。各键之间用指定的 *branch_delim* 字符串分隔。如果不需要分支显示，则在输出列列表中同时省略 *branch_delim* 参数和 branch 列即可。

如果同一父节点下各同级节点的顺序很重要，可以包含 *orderby_fld* 参数来指定按哪个字段对同级节点排序。该字段可以是任何可排序的数据类型。当且仅当指定了 *orderby_fld* 时，输出列列表才必须包含最后那个整数类型的序列号列。

表名和字段名参数会原样复制到 connectby 在内部生成的 SQL 查询中。因此，如果名称是大小写混合的，或者包含特殊字符，就应包含双引号。还可能需要对表名进行模式限定。

在大表中，除非父键字段上建有索引，否则性能会很差。

重要的是，*branch_delim* 字符串不要出现在任何键值中，否则 connectby 可能会错误地报告无限递归错误。注意，如果没有提供 *branch_delim*，为了递归检测会使用默认值 ~。

下面是一个示例：

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);

INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);

-- with branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text);
 keyid | parent_keyid | level |       branch
-------+--------------+-------+--------------------
 row2  |              |     0 | row2
 row4  | row2         |     1 | row2~row4
 row6  | row4         |     2 | row2~row4~row6
 row8  | row6         |     3 | row2~row4~row6~row8
 row5  | row2         |     1 | row2~row5
 row9  | row5         |     2 | row2~row5~row9
(6 rows)

-- without branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int);
 keyid | parent_keyid | level
```

```
-------+--------------+-------
 row2  |              |     0
 row4  | row2         |     1
 row6  | row4         |     2
 row8  | row6         |     3
 row5  | row2         |     1
 row9  | row5         |     2
(6 rows)

-- with branch, with orderby_fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text, pos int);
 keyid | parent_keyid | level |       branch        | pos
-------+--------------+-------+---------------------+-----
 row2  |              |     0 | row2                |   1
 row5  | row2         |     1 | row2~row5           |   2
 row9  | row5         |     2 | row2~row5~row9      |   3
 row4  | row2         |     1 | row2~row4           |   4
 row6  | row4         |     2 | row2~row4~row6      |   5
 row8  | row6         |     3 | row2~row4~row6~row8 |   6
(6 rows)

-- without branch, with orderby_fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int, pos int);
 keyid | parent_keyid | level | pos
-------+--------------+-------+-----
 row2  |              |     0 |   1
 row5  | row2         |     1 |   2
 row9  | row5         |     2 |   3
 row4  | row2         |     1 |   4
 row6  | row4         |     2 |   5
 row8  | row6         |     3 |   6
(6 rows)
```

F.38.2. 作者

Joe Conway

F.39. tcn

tcn 模块提供一个触发器函数，用于将其所附着的任意表的内容变更通知给监听者。它必须作为 AFTER 触发器，并以FOR EACH ROW 方式使用。

在 CREATE TRIGGER 语句中，至多可以为该函数提供一个参数，而且该参数是可选的。如果提供该参数，它将用作通知的通道名；如果省略，则使用tcn 作为通道名。

通知的载荷由表名、一个用于指示执行了哪种操作的字母，以及主键列的列名/值对组成。各部分之间都以逗号分隔。为了便于用正则表达式解析，表名和列名始终用双引号括起，数据值始终用单引号括起。嵌入的引号会被双写。

下面给出一个使用该扩展的简短示例。

```
test=# create table tcndata
test-#   (
test(#     a int not null,
test(#     b date not null,
test(#     c text,
test(#     primary key (a, b)
test(#   );
CREATE TABLE
test=# create trigger tcndata_tcn_trigger
test-#   after insert or update or delete on tcndata
test-#   for each row execute function triggered_change_notification();
CREATE TRIGGER
test=# listen tcn;
LISTEN
test=# insert into tcndata values (1, date '2012-12-22', 'one'),
test-#                            (1, date '2012-12-23', 'another'),
test-#                            (2, date '2012-12-23', 'two');
INSERT 0 3
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-22'" received
 from server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-23'" received
 from server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='2',"b"='2012-12-23'" received
 from server process with PID 22770.
test=# update tcndata set c = 'uno' where a = 1;
UPDATE 2
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-22'" received
 from server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-23'" received
 from server process with PID 22770.
test=# delete from tcndata where a = 1 and b = date '2012-12-22';
DELETE 1
Asynchronous notification "tcn" with payload ""tcndata",D,"a"='1',"b"='2012-12-22'" received
 from server process with PID 22770.
```

F.40. test_decoding

`test_decoding` 是一个逻辑解码输出插件示例。它本身并没有什么特别有用的功能，但可作为开发自定义输出插件的起点。

`test_decoding` 通过逻辑解码机制接收 WAL，并将其解码为所执行操作的文本表示形式。

通过 SQL 逻辑解码接口使用该插件时，其典型输出可能如下：

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL, NULL, 'include-xids',
 '0');
   lsn     | xid |                       data
-----------+-----+-------------------------------------------------
 0/16D30F8 | 691 | BEGIN
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:2 data[text]:'arg'
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:3 data[text]:'demo'
 0/16D32A0 | 691 | COMMIT
 0/16D32D8 | 692 | BEGIN
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:2
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:3
 0/16D3398 | 692 | COMMIT
(8 rows)
```

F.41. tsm_system_rows

tsm_system_rows 模块提供表采样方法 SYSTEM_ROWS，它可用于SELECT 命令的 TABLESAMPLE 子句中。

这种表采样方法接受一个整数参数，表示最多读取多少行。除非表中没有足够的行，否则生成的样本将始终恰好包含这么多行；如果表中的行数不足，则会选取整个表中的所有行。

与内置的 SYSTEM 采样方法一样，SYSTEM_ROWS 执行块级采样，因此采样并不是完全随机的，可能会受到聚簇效应的影响，特别是在只请求少量行时。

SYSTEM_ROWS 不支持 REPEATABLE 子句。

F.41.1. 示例

下面是一个使用 SYSTEM_ROWS 从表中选取样本的示例。首先安装扩展：

```
CREATE EXTENSION tsm_system_rows;
```

然后就可以在 SELECT 命令中使用它，例如：

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_ROWS(100);
```

这个命令将从表 my_table 中返回一个包含 100 行的样本（除非该表没有 100 个可见行，在这种情况下将返回其中所有行）。

F.42. tsm_system_time

tsm_system_time 模块提供表采样方法 SYSTEM_TIME，它可用于SELECT 命令的 TABLESAMPLE 子句中。

这种表采样方法接受一个浮点参数，表示读取该表时最多可花费多少毫秒。这使你能够直接控制查询耗时，但代价是样本大小会变得难以预测。除非先读完整个表，否则所得样本将包含在指定时间内能够读取到的尽可能多的行。

与内置的 SYSTEM 采样方法一样，SYSTEM_TIME 执行块级采样，因此采样并不是完全随机的，可能会受到聚簇效应的影响，特别是在只选取少量行时。

SYSTEM_TIME 不支持 REPEATABLE 子句。

F.42.1. 示例

下面是一个使用 SYSTEM_TIME 从表中选取样本的示例。首先安装扩展：

```
CREATE EXTENSION tsm_system_time;
```

然后就可以在 SELECT 命令中使用它，例如：

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_TIME(1000);
```

这个命令将返回在 1 秒（1000 毫秒）内能够从 my_table 读取到的尽可能大的样本。当然，如果整个表在不到 1 秒内就能读完，则会返回其中所有行。

F.43. unaccent

unaccent 是一个文本检索词典，它会从词位中移除重音符号（变音符号）。它是一个过滤词典，也就是说，它的输出总会被传递给下一个词典（如果有），这不同于词典的通常行为。这使得全文检索能够以不区分重音的方式处理文本。

unaccent 当前的实现还不能作为 thesaurus 词典的正规化词典使用。

F.43.1. 配置

unaccent 词典接受下列选项：

- RULES 是包含转换规则列表的文件的基名。该文件必须存放在 $SHAREDIR/tsearch_data/ 中（其中 $SHAREDIR 表示 PostgreSQL 安装的共享数据目录）。其名称必须以.rules 结尾，但在RULES 参数中不应包含这个后缀。

规则文件的格式如下：

- 每一行表示一条转换规则，由一个带重音的字符和一个不带重音的字符组成。前者会被转换成后者。例如：

```
À         A
Á         A
Â         A
Ã         A
Ä         A
Å         A
Æ         AE
```

  这两个字符必须以空白分隔，并且每行开头或结尾的空白都会被忽略。

- 另一种情况是，如果某一行只给出一个字符，则该字符的各次出现都会被删除；这对那些用独立字符表示重音的语言很有用。

- 实际上，这里的每个“字符”都可以是不包含空白的任意字符串，因此unaccent 词典除了用于去除变音符号之外，也可用于其他类型的子串替换。

- 与其他 PostgreSQL 文本检索配置文件一样，规则文件必须以 UTF-8 编码存储。加载时，数据会自动转换为当前数据库的编码。任何包含无法转换的字符的行都会被静默忽略，因此规则文件可以包含不适用于当前编码的规则。

一个更完整并且对大多数欧洲语言都可直接使用的示例可见于 unaccent.rules。安装 unaccent 模块时，该文件会被安装到$SHAREDIR/tsearch_data/中。这个规则文件会把带重音的字符转换成对应的不带重音字符，同时还会把连字展开为等价的一串简单字符（例如，将 Æ 转换为 AE）。

F.43.2. 用法

安装 unaccent 扩展会创建一个文本检索模板 unaccent 以及一个基于该模板的词典 unaccent。unaccent 词典的默认参数设置是 RULES='unaccent'，因此它可立即配合标准的unaccent.rules 文件使用。如果愿意，也可以修改这个参数，例如

```
mydb=# ALTER TEXT SEARCH DICTIONARY unaccent (RULES='my_rules');
```

或者基于该模板创建新的词典。

要测试该词典，可以尝试：

```
mydb=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----------
 {Hotel}
(1 row)
```

下面的示例展示了如何将 unaccent 词典插入到文本检索配置中：

```
mydb=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
mydb=# ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word
        WITH unaccent, french_stem;
mydb=# select to_tsvector('fr','Hôtels de la Mer');
    to_tsvector
-------------------
 'hotel':1 'mer':4
(1 row)

mydb=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
 ?column?
----------
 t
(1 row)

mydb=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
      ts_headline
------------------------
 <b>Hôtel</b> de la Mer
(1 row)
```

F.43.3. 函数

unaccent() 函数会从给定字符串中移除重音符号（变音符号）。从本质上说，它是对 unaccent 这一类型词典的一个包装器，但也可以在常规文本检索环境之外使用。

```
unaccent([dictionary regdictionary, ] string text) returns text
```

如果省略 *dictionary* 参数，则会使用与unaccent() 函数本身位于同一模式中、名为 unaccent 的文本检索词典。

例如：

```
SELECT unaccent('unaccent', 'Hôtel');
SELECT unaccent('Hôtel');
```

F.44. uuid-ossp

uuid-ossp 模块提供一些函数，可使用几种标准算法之一生成通用唯一标识符（UUID）。它还提供一些函数，用于生成某些特殊的 UUID 常量。

F.44.1. uuid-ossp 函数

表 F.32 显示了可用于生成 UUID 的函数。相关标准 ITU-T Rec. X.667、ISO/IEC 9834-8:2005 和 RFC 4122 规定了生成 UUID 的四种算法，由版本号 1、3、4 和 5 标识。（不存在版本 2 算法。）每种算法可能适用于不同的一组应用。

表 F.32. 用于生成 UUID 的函数

函数	描述
uuid_generate_v1()	此函数生成第 1 版 UUID。这需要使用计算机的 MAC 地址和时间戳。请注意，这类 UUID 会泄露创建该标识符的计算机身份及创建时间，因此可能不适合某些对安全敏感的应用。
uuid_generate_v1mc()	此函数生成第 1 版 UUID，但使用随机多播 MAC 地址，而不是计算机的真实 MAC 地址。
uuid_generate_v3(namespace uuid, name text)	此函数在给定命名空间中使用指定输入名称生成第 3 版 UUID。命名空间应是表 F.33 中所示uuid_ns_*()函数生成的特殊常量之一。（理论上它可以是任意 UUID。）名称是所选命名空间中的一个标识符。 例如： SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.postgresql.org'); 名称参数会先做 MD5 hash，因此无法从生成的 UUID 中反推出明文内容。通过这种方法生成 UUID 不包含随机因素，也不依赖于环境，因此结果可重现。
uuid_generate_v4()	此函数生成第 4 版 UUID，它完全由随机数派生。
uuid_generate_v5(namespace uuid, name text)	此函数生成第 5 版 UUID，其工作方式类似于第 3 版 UUID，但使用 SHA-1 作为 hash 方法。由于 SHA-1 被认为比 MD5 更安全，应优先使用第 5 版而不是第 3 版。

表 F.33. 返回 UUID 常量的函数

uuid_nil()	一个“nil”UUID 常量，不会作为真实 UUID 出现。
uuid_ns_dns()	指定 UUID 的 DNS 命名空间的常量。
uuid_ns_url()	指定 UUID 的 URL 命名空间的常量。
uuid_ns_oid()	指定 UUID 的 ISO 对象标识符（OID）命名空间的常量。（这涉及 ASN.1 OID，与PostgreSQL使用的 OID 无关。）
uuid_ns_x500()	指定 UUID 的 X.500 可辨识名称（DN）命名空间的常量。

F.44.2. 构建 uuid-ossp

历史上，此模块依赖 OSSP UUID 库，这也是该模块名称的由来。虽然在http://www.ossp.org/pkg/lib/uuid/仍然可以找到 OSSP UUID 库，但它维护得并不好，而且移植到较新的平台上正变得越来越困难。如今，在某些平台上，`uuid-ossp` 已可在不依赖 OSSP 库的情况下构建。在 FreeBSD 和其他一些 BSD 派生平台上，合适的 UUID 生成功能包含在核心 `libc` 库中。在 Linux、macOS 和其他一些平台上，相应功能由`libuuid` 库提供，该库最初来自 `e2fsprogs` 项目（不过在现代 Linux 上，它被视为 `util-linux-ng` 的一部分）。调用 `configure` 时，可指定 `--with-uuid=bsd` 以使用 BSD 函数，或指定`--with-uuid=e2fs` 以使用 `e2fsprogs` 的 `libuuid`，或指定`--with-uuid=ossp` 以使用 OSSP UUID 库。某台机器上可能同时具备多个这样的库，因此`configure` 不会自动选择其一。

> **注意**
>
> 如果只需要随机生成的（第 4 版）UUID，可以考虑改用pgcrypto 模块中的 `gen_random_uuid()` 函数。

F.44.3. 作者

Peter Eisentraut `<peter_e@gmx.net>`

F.45. xml2

xml2 模块提供 XPath 查询与 XSLT 功能。

F.45.1. 弃用说明

自 PostgreSQL 8.3 起，核心服务器就提供了基于 SQL/XML 标准的 XML 相关功能。该功能覆盖了 XML 语法检查和 XPath 查询，也就是本模块所做的事情，而且能力更多；不过，两者的 API 完全不兼容。计划在未来的 PostgreSQL 版本中移除此模块，转而采用较新的标准 API，因此建议你尝试迁移应用程序。如果你发现本模块的某些功能在较新的 API 中还没有以足够完善的形式提供，请向 <pgsql-hackers@lists.postgresql.org> 说明你的问题，以便补齐这一不足。

F.45.2. 函数说明

表 F.34 显示了该模块提供的函数。这些函数提供简单直接的 XML 解析和 XPath 查询功能。所有参数的类型都是 text，为简洁起见，此处不再显示。

表 F.34. 函数

函数	返回值	描述
xml_valid(document)	bool	此函数解析其参数中的文档文本，如果文档是良构 XML，则返回 true。（注意：这是标准 PostgreSQL 函数 xml_is_well_formed() 的别名。由于 XML 中有效性和良构性具有不同含义，函数名 xml_valid() 在技术上并不正确。）
xpath_string(document, query)	text	这些函数计算所提供文档上的 XPath 查询，并将结果转换为指定类型。
xpath_number(document, query)	float4	
xpath_bool(document, query)	bool	
xpath_nodeset(document, query, toptag, itemtag)	text	该函数计算文档上的查询，并将结果封装在 XML 标签中。如果结果有多个值，输出将如下所示： <toptag> <itemtag>Value 1 (可以是 XML 片段) </itemtag> <itemtag>Value 2....</itemtag> </toptag> 如果toptag 或itemtag 为空字符串，就会省略相应的标签。
xpath_nodeset(document, query)	text	与 xpath_nodeset(document, query, toptag, itemtag) 相同，但结果省略这两个标签。
xpath_nodeset(document, query, itemtag)	text	与xpath_nodeset(document, query, toptag, itemtag) 类似，但结果省略toptag。

函数	返回值	描述
`xpath_list(document, query, separator)`	`text`	此函数返回由指定分隔符分隔的多个值，例如，如果分隔符为`,`，则返回 `Value 1,Value 2,Value 3`。
`xpath_list(document, query)`	`text`	这是上一个函数的包装器，使用`,`作为分隔符。

F.45.3. xpath_table

```
xpath_table(text key, text document, text relation, text xpaths, text criteria) returns setof
 record
```

`xpath_table` 是一个表函数，它对一组文档中的每个文档执行一组 XPath 查询，并将结果作为表返回。原始文档表中的主键字段会作为结果的第一列返回，因此结果集可以方便地用于连接。参数说明见表 F.35。

表 F.35. xpath_table 参数

参数	描述
key	“键”字段的名称 – 这只是一个作为输出表第一列使用的字段，也就是说，它用于标识每个输出行来自哪条记录（关于多个值，见下文注释）
document	包含 XML 文档的字段名
relation	包含这些文档的表或视图名
xpaths	一个或多个 XPath 表达式，用 `|` 分隔
criteria	WHERE 子句的内容。这一项不能省略，因此如果你想处理该表或视图中的所有行，请使用 `true` 或 `1=1`

这些参数（XPath 字符串除外）只是直接替换进一个普通的 SQL SELECT 语句中，因此你有一定的灵活性 – 该语句为

```
SELECT <key>, <document> FROM <relation> WHERE <criteria>
```

因此，在这些特定位置上，任何合法内容都可以使用。该 SELECT 的结果必须恰好返回两列（除非你试图为 key 或 document 列出多个字段，否则都会如此）。请注意，这种简化做法要求你验证任何用户提供的值，以避免 SQL 注入攻击。

该函数必须用在 `FROM` 表达式中，并带有一个 `AS` 子句来指定输出列，例如

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
            'date_entered > ''2003-01-01'' ')
AS t(article_id integer, author text, page_count integer, title text);
```

`AS` 子句定义了输出表中各列的名称和类型。第一列是“键”字段，其余各列对应 XPath 查询。如果 XPath 查询多于结果列，多出的查询将被忽略。如果结果列多于 XPath 查询，多出的列将为 NULL。

注意，这个示例把 page_count 结果列定义为整数。该函数内部只处理字符串表示，因此当你在输出中声明整数时，它会取 XPath 结果的字符串表示，并使用 PostgreSQL 输入函数将其转换为整数（或者 AS 子句要求的任何其他类型）。如果做不到这一点 – 例如结果为空 – 就会报错，因此如果你认为数据可能有问题，最好还是把列类型设为 text。

调用该函数的 SELECT 语句不一定非得只是 SELECT * – 它可以按名称引用输出列，或者将它们与其他表连接。该函数会生成一个虚拟表，你可以对它执行任何想要的操作（例如聚合、连接、排序等）。因此，更复杂一点的例子还可以是：

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                 '/article/title|/article/author/@id',
                 'xpath_string(article_xml,''/article/@date'') > ''2003-03-20'' ')
       AS t(article_id integer, title text, author_id integer),
     tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

当然，出于方便，你也可以把这一切封装到一个视图中。

F.45.3.1. 多值结果

xpath_table 函数假定每个 XPath 查询的结果都可能有多个值，因此该函数返回的行数可能不同于输入文档的数量。返回的第一行包含每个查询的第一个结果，第二行包含每个查询的第二个结果。如果某个查询的值少于其他查询，则会在相应位置返回空值。

在某些情况下，用户会知道某个 XPath 查询只会返回单个结果（例如唯一文档标识符）– 如果它与返回多个结果的 XPath 查询一起使用，这个单值结果只会出现在结果的第一行。解决办法是把键字段作为与一个更简单的 XPath 查询结果进行连接的条件之一。例如：

```
CREATE TABLE test (
    id int PRIMARY KEY,
    xml text
);

INSERT INTO test VALUES (1, '<doc num="C1">
<line num="L1"><a>1</a><b>2</b><c>3</c></line>
<line num="L2"><a>11</a><b>22</b><c>33</c></line>
</doc>');

INSERT INTO test VALUES (2, '<doc num="C2">
<line num="L1"><a>111</a><b>222</b><c>333</c></line>
<line num="L2"><a>111</a><b>222</b><c>333</c></line>
</doc>');

SELECT * FROM
  xpath_table('id','xml','test',
              '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
  AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int, val2 int, val3 int)
WHERE id = 1 ORDER BY doc_num, line_num

 id | doc_num | line_num | val1 | val2 | val3
----+---------+----------+------+------+------
  1 | C1      | L1       |    1 |    2 |    3
  1 |         | L2       |   11 |   22 |   33
```

要让 doc_num 在每一行中都出现，解决办法是调用两次 xpath_table 并连接结果：

```
SELECT t.*,i.doc_num FROM
  xpath_table('id', 'xml', 'test',
              '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
    AS t(id int, line_num varchar(10), val1 int, val2 int, val3 int),
  xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
    AS i(id int, doc_num varchar(10))
WHERE i.id=t.id AND i.id=1
ORDER BY doc_num, line_num;

 id | line_num | val1 | val2 | val3 | doc_num
----+----------+------+------+------+---------
  1 | L1       |    1 |    2 |    3 | C1
  1 | L2       |   11 |   22 |   33 | C1
(2 rows)
```

F.45.4. XSLT 函数

如果安装了 libxslt，则可使用下列函数：

F.45.4.1. xslt_process

```
xslt_process(text document, text stylesheet, text paramlist) returns text
```

这个函数将 XSL 样式表应用到文档上，并返回转换后的结果。paramlist 是转换时要使用的参数赋值列表，以 a=1,b=2 的形式指定。注意，参数解析非常简单：参数值中不能包含逗号！

此外还有 xslt_process 的双参数版本，它不会向转换传递任何参数。

F.45.5. 作者

John Gray <jgray@azuli.co.uk>

本模块由 Torchbox Ltd. (www.torchbox.com) 赞助开发。它采用与 PostgreSQL 相同的 BSD 许可证。

附录 G. 额外提供的程序

本附录与前一个附录包含有关 PostgreSQL 发行版中 `contrib` 目录内模块的信息。关于 `contrib` 部分的一般性信息，以及 `contrib` 中服务器扩展和插件的具体信息，请参见附录 F。

本附录介绍位于 `contrib` 中的实用程序。无论是通过源码还是打包系统安装，它们安装后都会出现在 PostgreSQL 安装的 `bin` 目录中，并可像其他任何程序一样使用。

G.1. 客户端应用

本节介绍 `contrib` 中的 PostgreSQL 客户端应用程序。无论数据库服务器位于何处，都可以从任何位置运行它们。另见 PostgreSQL 客户端应用，其中介绍了属于核心 PostgreSQL 发行版一部分的客户端应用程序。

oid2name

oid2name — 解析 PostgreSQL 数据目录中的 OID 和文件节点

大纲

`oid2name` [*option*...]

描述

oid2name 是一个帮助管理员检查 PostgreSQL 所用文件结构的实用程序。要使用它，你需要熟悉数据库文件结构，这在第 69 章中有说明。

> **注意**
>
> “oid2name” 这个名称是历史遗留，实际上颇具误导性，因为大多数时候你真正关心的是表的文件节点编号（也就是数据库目录中可见的文件名）。务必理解表 OID 与表文件节点之间的区别。

oid2name 连接到目标数据库，并提取 OID、文件节点和/或表名信息。你也可以让它显示数据库 OID 或表空间 OID。

选项

oid2name 接受下列命令行参数：

`-f` *`filenode`*

显示文件节点为 *`filenode`* 的表的信息

`-i`

在列表中包含索引和序列

`-o` *`oid`*

显示 OID 为 *`oid`* 的表的信息

`-q`

省略标题（适用于脚本）

`-s`

显示表空间 OID

`-S`

包含系统对象（位于 `information_schema`、`pg_toast` 和 `pg_catalog` 模式中的对象）

`-t` *`tablename_pattern`*

显示与 *`tablename_pattern`* 匹配的表的信息

`-V`
`--version`

输出 oid2name 版本并退出。

`-x`

显示每个对象的更多信息：表空间名、模式名和 OID

`-?`
`--help`

显示 oid2name 命令行参数的帮助并退出。

oid2name 也接受下列用于连接参数的命令行参数：

`-d` *`database`*

要连接的数据库

`-H` *`host`*

数据库服务器的主机

`-p` *`port`*

数据库服务器的端口

`-U` *`username`*

用于连接的用户名

`-P` *`password`*

密码（已弃用 – 将其放在命令行上会带来安全风险）

若要显示特定的表，可使用`-o`、`-f` 和/或 `-t` 来选择要显示哪些表。`-o` 接受一个 OID，`-f` 接受一个文件节点，而`-t` 接受一个表名（实际上是一个 `LIKE` 模式，因此你可以使用诸如`foo%` 之类的形式）。这些选项可以按需重复使用，最终的列表将包含匹配任意一个选项的所有对象。但请注意，这些选项只能显示由`-d` 指定的数据库中的对象。

如果没有给出 `-o`、`-f` 或 `-t`，但给出了`-d`，那么它将列出`-d` 指定数据库中的所有表。在这种模式下，`-S` 和 `-i` 选项控制要列出哪些对象。

如果连 `-d` 也没有给出，它将显示数据库 OID 列表。另外，你也可以给出`-s` 来获取表空间列表。

注解

oid2name 要求数据库服务器正在运行，且系统目录未损坏。因此，它对于从数据库灾难性损坏中恢复的帮助非常有限。

示例

```
$ # what's in this database server, anyway?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # OK, let's look into database alvherre
$ cd $PGDATA/base/17228

$ # get top 10 db objects in the default tablespace, ordered by size
$ ls -lS * | head -10
-rw-------  1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw-------  1 alvherre alvherre  17965056 sep 14 09:51 1155291
-rw-------  1 alvherre alvherre   1204224 sep 14 09:51 16717
-rw-------  1 alvherre alvherre    581632 sep  6 17:51 1255
-rw-------  1 alvherre alvherre    237568 sep 14 09:50 16674
-rw-------  1 alvherre alvherre    212992 sep 14 09:51 1249
-rw-------  1 alvherre alvherre    204800 sep 14 09:51 16684
-rw-------  1 alvherre alvherre    196608 sep 14 09:50 16700
-rw-------  1 alvherre alvherre    163840 sep 14 09:50 16699
-rw-------  1 alvherre alvherre    122880 sep  6 17:51 16751

$ # I wonder what file 155173 is ...
$ oid2name -d alvherre -f 155173
From database "alvherre":
  Filenode  Table Name
----------------------
    155173    accounts

$ # you can ask for more than one object
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":
  Filenode     Table Name
-------------------------
    155173       accounts
   1155291  accounts_pkey

$ # you can mix the options, and get more details with -x
$ oid2name -d alvherre -t accounts -f 1155291 -x
From database "alvherre":
  Filenode     Table Name      Oid  Schema  Tablespace
-----------------------------------------------------
```

```
    155173       accounts   155173  public  pg_default
   1155291  accounts_pkey  1155291  public  pg_default

$ # show disk space for every db object
$ du [0-9]* |
> while read SIZE FILENODE
> do
>   echo "$SIZE       `oid2name -q -d alvherre -i -f $FILENODE`"
> done
16            1155287  branches_pkey
16            1155289  tellers_pkey
17561            1155291  accounts_pkey
...

$ # same, but sort by size
$ du [0-9]* | sort -rn | while read SIZE FN
> do
>   echo "$SIZE   `oid2name -q -d alvherre -f $FN`"
> done
133466             155173    accounts
17561            1155291  accounts_pkey
1177              16717  pg_proc_proname_args_nsp_index
...

$ # If you want to see what's in tablespaces, use the pg_tblspc directory
$ cd $PGDATA/pg_tblspc
$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # what databases have objects in tablespace "fastdisk"?
$ ls -d 155151/*
155151/17228/  155151/PG_VERSION

$ # Oh, what was database 17228 again?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ # Let's see what objects does this database have in the tablespace.
$ cd 155151/17228
$ ls -l
total 0
-rw-------  1 postgres postgres 0 sep 13 23:20 155156

$ # OK, this is a pretty small table ... but which one is it?
$ oid2name -d alvherre -f 155156
From database "alvherre":
```

```
  Filenode  Table Name
----------------------
    155156         foo
```

作者

B. Palmer <bpalmer@crimelabs.net>

vacuumlo

vacuumlo — 从 PostgreSQL 数据库中移除孤立的大对象

大纲

`vacuumlo` [*option*...] *dbname*...

描述

vacuumlo 是一个简单的实用程序，用于从 PostgreSQL 数据库中移除“孤立的”大对象。孤立的大对象（LO）是指其 OID 没有出现在该数据库任何 `oid` 或 `lo` 数据列中的 LO。

如果你使用这个程序，也可能会对lo 模块中的 `lo_manage` 触发器感兴趣。`lo_manage` 有助于尽量避免一开始就创建出孤立 LO。

命令行中给出的所有数据库都会被处理。

选项

vacuumlo 接受下列命令行参数：

`-l` *limit*

每个事务最多移除 *limit* 个大对象（默认值为 1000）。由于服务器会为每个被移除的 LO 获取一个锁，在单个事务中移除过多 LO 有可能超出 max_locks_per_transaction。如果你希望在单个事务中完成全部移除，请把该限制设为 0。

`-n`

不实际移除任何内容，只显示将要执行的操作。

`-v`

输出大量进度消息。

`-V`
`--version`

打印 vacuumlo 版本并退出。

`-?`
`--help`

显示 vacuumlo 命令行参数的帮助并退出。

vacuumlo 还接受下列用于连接参数的命令行参数：

`-h` *hostname*

数据库服务器主机。

`-p` *port*

数据库服务器端口。

`-U` *username*

用于连接的用户名。

`-w`
`--no-password`

绝不发出密码提示。如果服务器要求密码认证，而又无法通过其他方式（例如`.pgpass`文件）获得密码，则连接尝试将失败。这个选项在批处理作业和脚本中很有用，因为这些场景下没有用户可以输入密码。

`-W`

强制 vacuumlo 在连接数据库之前提示输入密码。

这个选项并非必不可少，因为如果服务器要求密码认证，vacuumlo 会自动提示输入密码。不过，vacuumlo 会浪费一次连接尝试，才知道服务器需要密码。在某些情况下，为了避免这次额外的连接尝试，提前指定`-W`是值得的。

注解

vacuumlo 按以下方法工作：首先，vacuumlo 构建一个临时表，其中包含所选数据库中所有大对象的 OID。然后，它会扫描数据库中所有类型为`oid`或`lo`的列，并从临时表中移除匹配的项。（注意：只会考虑名称正好为这两种的类型；特别是，基于这些类型定义的域不会被考虑。）临时表中剩余的项标识出孤立 LO，并会被移除。

作者

Peter Mount `<peter@retep.org.uk>`

G.2. 服务器应用

本节介绍`contrib`中与 PostgreSQL 服务器相关的应用程序。它们通常运行在数据库服务器所在的主机上。另见 PostgreSQL 服务器端应用，其中介绍了作为核心PostgreSQL 发行版一部分的服务器应用程序。

pg_standby

pg_standby — 支持创建 PostgreSQL 温备服务器

大纲

pg_standby [*option*...] *archivelocation nextwalfile walfilepath* [*restartwalfile*]

描述

pg_standby 支持创建“温备”数据库服务器。它既是可用于生产环境的程序，也是可定制的模板，便于在有需要时进行特定修改。

pg_standby 设计为一个会等待的 `restore_command`，将标准归档恢复转变为温备运行需要这样的命令。此外还需要其他配置，服务器的主手册中对此均有介绍（参见第 26.2 节）。

要配置备库使用 pg_standby，请将以下内容放入其 `recovery.conf` 配置文件：

```
restore_command = 'pg_standby archiveDir %f %p %r'
```

其中，*archiveDir* 是恢复 WAL 段文件时所读取的目录。

如果指定了 *restartwalfile*（通常使用 `%r` 宏），就会从 *archivelocation* 中删除逻辑上位于该文件之前的所有 WAL 文件。这样既能尽量减少需要保留的文件数量，又能保留崩溃后重启的能力。如果 *archivelocation* 是专供此备库使用的临时暂存区，就适合使用此参数；但如果 *archivelocation* 用作长期 WAL 归档区，就不应使用。

pg_standby 假定 *archivelocation* 是服务器所属用户可读的目录。如果指定了 *restartwalfile*（或 `-k`），*archivelocation* 目录还必须可写。

主库发生故障时，切换到“温备”数据库服务器有以下两种方式：

智能故障切换

> 在智能故障切换中，服务器会先应用归档中所有可用的 WAL 文件，再正式启动。即使备库已经落后，这也能做到零数据丢失；但如果尚未应用的 WAL 很多，备库可能需要很长时间才能就绪。要触发智能故障切换，请创建包含单词 `smart` 的触发文件，或者直接创建一个空的触发文件。

快速故障切换

> 在快速故障切换中，服务器会立即正式启动。归档中所有尚未应用的 WAL 文件都会被忽略，这些文件中的全部事务都会丢失。要触发快速故障切换，请创建触发文件，并在其中写入单词 `fast`。也可以配置 pg_standby，使其在指定时间间隔内没有出现新 WAL 文件时，自动执行快速故障切换。

选项

pg_standby 接受以下命令行参数：

`-c`

使用 `cp` 或 `copy` 命令从归档恢复 WAL 文件。由于这是唯一受支持的行为，因此该选项没有实际作用。

`-d`

在 `stderr` 上输出大量调试日志。

`-k`

从 *`archivelocation`* 中删除文件，使归档中保留的、位于当前文件之前的 WAL 文件不超过此数量。零（默认值）表示不从 *`archivelocation`* 中删除任何文件。如果指定了 *`restartwalfile`*，就会静默忽略此参数，因为前者能更准确地确定归档的正确截断点。从 PostgreSQL 8.3 起，已弃用此参数；指定 *`restartwalfile`* 参数更安全、更高效。设置过小可能删除备库重启仍需使用的文件，设置过大则会浪费归档空间。

`-r` *`maxretries`*

设置复制命令失败后的最大重试次数（默认为 3）。每次失败后，都会等待 *`sleeptime`* * *`num_retries`*，因此等待时间会逐次增加。默认情况下，会依次等待 5 秒、10 秒、15 秒，然后才向备库报告失败。这会被解释为恢复结束，从而使备库完全启动。

`-s` *`sleeptime`*

设置两次检查之间的等待秒数（最多 60 秒，默认 5 秒），检查的内容是待恢复的 WAL 文件是否已在归档中可用。默认设置不一定就是推荐设置；相关讨论参见第 26.2 节。

`-t` *`triggerfile`*

指定触发文件；一旦该文件出现，就会引发故障切换。建议使用带有明确结构的文件名，例如 `/tmp/pgsql.trigger.5432`，以免同一系统上存在多个服务器时，混淆被触发的是哪个服务器。

`-V`
`--version`

打印 pg_standby 的版本并退出。

`-w` *`maxwaittime`*

设置等待下一个 WAL 文件的最长秒数，超过此时间后执行快速故障切换。零（默认值）表示永远等待。默认设置不一定就是推荐设置；相关讨论参见第 26.2 节。

`-?`
`--help`

显示有关 pg_standby 命令行参数的帮助信息并退出。

说明

pg_standby 设计为与 PostgreSQL 8.2 及更高版本配合使用。

PostgreSQL 8.3 提供了 `%r` 宏，用于让 pg_standby 知道需要保留的最后一个文件。使用 PostgreSQL 8.2 时，如果需要清理归档，就必须使用 `-k` 选项。该选项在 8.3 中仍然可用，但已弃用。

PostgreSQL 8.4 提供了 `recovery_end_command` 选项。如果没有此选项，遗留的触发文件可能带来危险。

pg_standby 用 C 编写，源码易于修改，并专门标出了可根据自身需要进行修改的部分。

示例

在 Linux 或 Unix 系统上，可以使用：

```
archive_command = 'cp %p .../archive/%f'

restore_command = 'pg_standby -d -s 2 -t /tmp/pgsql.trigger.5442 .../archive %f %p %r
 2>>standby.log'

recovery_end_command = 'rm -f /tmp/pgsql.trigger.5442'
```

这里，归档目录实际位于备库上，因此 `archive_command` 通过 NFS 访问该目录，但这些文件对备库而言是本地文件（因此可以使用 `ln`）。此配置会：

- 将调试输出写入 `standby.log`
- 每隔 2 秒检查一次下一个 WAL 文件是否可用
- 仅在名为 `/tmp/pgsql.trigger.5442` 的触发文件出现时停止等待，并根据其内容执行故障切换
- 恢复结束时删除触发文件
- 从归档目录中移除不再需要的文件

在 Windows 上，可以使用：

```
archive_command = 'copy %p ...\\archive\\%f'

restore_command = 'pg_standby -d -s 5 -t C:\pgsql.trigger.5442 ...\archive %f %p %r 2>>standby.
log'

recovery_end_command = 'del C:\pgsql.trigger.5442'
```

注意，需要双写反斜杠的设置是 `archive_command`，而无需双写的设置是 `restore_command` 或 `recovery_end_command`。此配置会：

- 使用 `copy` 命令从归档恢复 WAL 文件
- 将调试输出写入 `standby.log`
- 每隔 5 秒检查一次下一个 WAL 文件是否可用
- 仅在名为 `C:\pgsql.trigger.5442` 的触发文件出现时停止等待，并根据其内容执行故障切换
- 恢复结束时删除触发文件
- 从归档目录中移除不再需要的文件

Windows 上的 `copy` 命令会在文件复制完成之前设置最终文件大小，这通常会让 pg_standby 产生误判。因此，pg_standby 一旦看到正确的文件大小，还会等待 *`sleeptime`* 秒。GNUWin32 的 `cp` 只有在文件复制完成后才设置文件大小。

由于 Windows 示例在两端都使用 `copy`，任一服务器或两个服务器都可能通过网络访问归档目录。

作者

Simon Riggs <simon@2ndquadrant.com>

参见

pg_archivecleanup

附录 H. 外部项目

PostgreSQL 是一个复杂的软件项目，对其进行管理并非易事。我们发现，许多对 PostgreSQL 的增强若脱离核心项目单独开发，往往能更高效地完成。

H.1. 客户端接口

基础 PostgreSQL 发行版中只包含两个客户端接口：

- 之所以包含 libpq，是因为它是主要的 C 语言接口，而且许多其他客户端接口都构建在它之上。
- 之所以包含 ECPG，是因为它依赖服务器端 SQL 语法，因此对 PostgreSQL 自身的变更非常敏感。

其他所有语言接口都属于外部项目，并单独发布。PostgreSQL Wiki 上维护了一份语言接口列表[1]。请注意，其中有些软件包并非按与 PostgreSQL 相同的许可发布。有关各语言接口的更多信息，包括许可条款，请参阅其网站和文档。

https://wiki.postgresql.org/wiki/List_of_drivers

H.2. 管理工具

有若干管理工具可用于 PostgreSQL。其中最流行的是 pgAdmin[2]，另外也有若干商业可用的工具。

H.3. 过程语言

基础 PostgreSQL 发行版中包含若干过程语言：PL/pgSQL、PL/Tcl、PL/Perl 以及 PL/Python。

此外，还有一些过程语言是在核心 PostgreSQL 发行版之外开发和维护的。PostgreSQL Wiki 上维护了一份过程语言列表[3]。请注意，其中有些项目并非按与 PostgreSQL 相同的许可发布。有关各过程语言的更多信息，包括许可信息，请参阅其网站和文档。

https://wiki.postgresql.org/wiki/PL_Matrix

H.4. 扩展

PostgreSQL 的设计目标之一，就是使其易于扩展。因此，载入数据库的扩展可以像内置特性一样工作。随源代码一同提供的 `contrib/` 目录中包含若干扩展，它们在附录 F 中有所描述。其他扩展则是独立开发的，例如 PostGIS[4]。甚至 PostgreSQL 的复制解决方案也可以在外部开发。例如，Slony-I[5] 就是一个流行的主库/备库复制解决方案，它是独立于核心项目开发的。

[1] https://wiki.postgresql.org/wiki/List_of_drivers
[2] https://www.pgadmin.org/
[3] https://wiki.postgresql.org/wiki/PL_Matrix
[4] https://postgis.net/
[5] https://www.slony.info

附录 I. 源代码仓库

PostgreSQL 的源代码使用 Git 版本控制系统进行存储和管理。官方提供了主仓库的公开镜像；主仓库发生任何变更后，该镜像都会在一分钟内更新。

我们的 wiki（https://wiki.postgresql.org/wiki/Working_with_Git）上有一些关于使用 Git 的讨论。

请注意，从源代码仓库构建PostgreSQL 需要版本相当新的 bison、flex 和 Perl。从发行版 tarball 构建则不需要这些工具，因为这些工具生成的文件已经包含在 tarball 中。其他工具要求与第 16.2 节中的说明相同。

I.1. 通过 Git 获取源代码

使用 Git 时，你将在本地机器上创建整个代码仓库的一个副本，因此即使离线也能访问全部历史和分支。这是开发或测试补丁最快且最灵活的方式。

Git

1. 你需要安装 Git，可从https://git-scm.com 获取。许多系统默认已安装了较新版本的 Git，或者可以通过其软件包分发系统获得。

2. 要开始使用 Git 仓库，请克隆官方镜像：

   ```
   git clone https://git.postgresql.org/git/postgresql.git
   ```

   这会将完整的仓库复制到本地机器，因此可能需要一些时间才能完成，尤其是网络连接较慢时。文件将被放置在当前目录下新建的`postgresql` 子目录中。

   也可以通过 Git 协议访问 Git 镜像。只需将 URL 前缀改为 `git`，如下所示：

   ```
   git clone git://git.postgresql.org/git/postgresql.git
   ```

3. 每当想要获取系统的最新更新时，`cd` 进入该仓库目录并运行：

   ```
   git fetch
   ```

Git 能做的远不止获取源代码。要了解更多信息，请参阅Git 的手册页，或者访问https://git-scm.com 网站。

附录 J. 文档

PostgreSQL 有四种主要的文档格式：

- 纯文本，用于安装前信息
- HTML，用于在线浏览和参考
- PDF，用于打印
- 手册页，用于快速参考。

另外，若干纯文本`README` 文件可以在 PostgreSQL 源代码树各处找到，用于说明各种实现方面的问题。

HTML 文档和手册页属于标准发行包的一部分，并会默认安装。PDF 格式的文档可单独下载。

J.1. DocBook

文档源文件使用 DocBook 编写，它是一种由 XML 定义的标记语言。下文中会同时使用 DocBook 和 XML 这两个术语，但严格地说，它们在技术上并不能互换。

DocBook 允许作者指定技术文档的结构和内容，而无需操心展示细节。文档样式定义这些内容如何被渲染成若干最终形式之一。DocBook 由 OASIS group[1] 维护。官方 DocBook 站点[2]提供了很好的入门和参考文档，以及一本可供在线阅读的完整 O'Reilly 图书。NewbieDoc Docbook Guide[3] 对初学者非常有帮助。FreeBSD Documentation Project[4] 也使用 DocBook，并提供了一些很有价值的信息，其中包括若干值得参考的风格指南。

J.2. 工具集

以下工具用于处理文档。其中一些可能是可选的，具体见说明。

DocBook DTD[5]

这是 DocBook 自身的定义。我们当前使用 4.2 版；不能使用更高或更低的版本。你需要的是 DocBook DTD 的 XML 变体，而不是 SGML 变体。

DocBook XSL Stylesheets[6]

它们包含将 DocBook 源文件转换成其他格式（例如 HTML）的处理指令。

当前要求的最低版本是 1.77.0，但建议使用最新可用版本以获得最佳效果。

Libxml2[7]，用于`xmllint`

这个库及其所含的 `xmllint` 工具用于处理 XML。许多开发者可能已经安装了 Libxml2，因为构建 PostgreSQL 代码时也会用到它。不过请注意，`xmllint` 可能需要通过单独的子包安装。

[1] https://www.oasis-open.org
[2] https://www.oasis-open.org/docbook/
[3] http://newbiedoc.sourceforge.net/metadoc/docbook-guide.html
[4] https://www.freebsd.org/docproj/
[5] https://www.oasis-open.org/docbook/
[6] https://github.com/docbook/wiki/wiki/DocBookXslStylesheets
[7] http://xmlsoft.org/

Libxslt[8]，用于`xsltproc`

`xsltproc` 是一个 XSLT 处理器，也就是使用 XSLT 样式表把 XML 转换成其他格式的程序。

FOP[9]

这是一个可将 XML 转换成 PDF 等格式的程序。只有在你希望构建 PDF 格式文档时才需要它。

我们已经记录了多种安装处理文档所需工具的方法，下面将加以介绍。这些工具也可能还有其他打包发行形式。请将软件包状态报告到文档邮件列表，我们会把这些信息补充到这里。

J.2.1. 在 Fedora、RHEL 及其衍生版上安装

要安装所需软件包，请使用：

```
yum install docbook-dtds docbook-style-xsl libxslt fop
```

J.2.2. 在 FreeBSD 上安装

要安装所需软件包，可以使用`pkg`，命令如下：

```
pkg install docbook-xml docbook-xsl libxslt fop
```

从 `doc` 目录构建文档时，你需要使用`gmake`，因为所提供的 makefile 不适用于 FreeBSD 的 `make`。

J.2.3. Debian 软件包

完整的文档工具软件包也适用于 Debian GNU/Linux。只需使用以下命令即可安装：

```
apt-get install docbook-xml docbook-xsl libxml2-utils xsltproc fop
```

J.2.4. macOS

如果使用 MacPorts，以下命令即可完成配置：

```
sudo port install docbook-xml docbook-xsl-nons libxslt fop
```

如果使用 Homebrew，请运行：

```
brew install docbook docbook-xsl libxslt fop
```

[8] http://xmlsoft.org/XSLT/
[9] https://xmlgraphics.apache.org/fop/

Homebrew 提供的程序要求设置下列环境变量。对于基于 Intel 的机器，请使用：

```
export XML_CATALOG_FILES=/usr/local/etc/xml/catalog
```

对于基于 Apple Silicon 的机器，请使用：

```
export XML_CATALOG_FILES=/opt/homebrew/etc/xml/catalog
```

否则，`xsltproc` 会抛出如下错误：

```
I/O error : Attempt to load network entity http://www.oasis-open.org/docbook/xml/4.5/docbookx.
dtd
postgres.sgml:21: warning: failed to load external entity "http://www.oasis-open.org/docbook/
xml/4.5/docbookx.dtd"
...
```

虽然可以使用 Apple 提供的 `xmllint` 和 `xsltproc`，而不使用 MacPorts 或 Homebrew 提供的版本，但你仍然需要安装 DocBook DTD 和样式表，并设置一个指向它们的 catalog 文件。

J.2.5. 通过 `configure` 检测

在构建文档之前，你需要运行`configure` 脚本，就像构建 PostgreSQL 程序本身时那样。检查运行接近结尾处的输出；它应该大致如下：

```
checking for xmllint... xmllint
checking for xsltproc... xsltproc
checking for fop... fop
checking for dbtoepub... dbtoepub
```

如果找不到 `xmllint` 或 `xsltproc`，你就无法构建任何文档。`fop` 仅在构建 PDF 格式文档时需要。`dbtoepub` 仅在构建 EPUB 格式文档时需要。

如有必要，你可以告诉`configure` 到哪里查找这些程序，例如：

```
./configure ... XMLLINT=/opt/local/bin/xmllint ...
```

J.3. 构建文档

一切设置妥当后，切换到`doc/src/sgml` 目录，并运行后续各小节中介绍的某个命令来构建文档。（记得使用 GNU make。）

J.3.1. HTML

要构建文档的 HTML 版本：

```
doc/src/sgml$ make html
```

这也是默认目标。输出位于子目录 `html` 中。

若要使用 postgresql.org[10] 上所使用的样式表，而不是默认的简单样式来生成 HTML 文档，请使用：

```
doc/src/sgml$ make STYLE=website html
```

J.3.2. 手册页

我们使用 DocBook XSL 样式表将 DocBook `refentry` 页面转换为适合手册页的 *roff 输出。要创建手册页，请使用以下命令：

```
doc/src/sgml$ make man
```

J.3.3. PDF

要使用 FOP 生成文档的 PDF 版本，可根据所偏好的纸张格式使用下列命令之一：

- 对于 A4 格式：

  ```
  doc/src/sgml$ make postgres-A4.pdf
  ```

- 对于美国信纸格式：

  ```
  doc/src/sgml$ make postgres-US.pdf
  ```

由于 PostgreSQL 文档相当庞大，FOP 需要占用相当多的内存。因此，在某些系统上，构建会因内存相关错误而失败。这通常可以通过在配置文件 `~/.foprc` 中配置 Java 堆设置来解决，例如：

```
# FOP 二进制发行版
FOP_OPTS='-Xmx1500m'
# Debian
JAVA_ARGS='-Xmx1500m'
# Red Hat
ADDITIONAL_FLAGS='-Xmx1500m'
```

所需内存存在一个最低门槛，而且在一定程度上，内存越多似乎会让构建稍快一些。对于内存很少（小于 1 GB）的系统，构建要么会因交换而非常缓慢，要么根本无法工作。

也可以手工使用其他 XSL-FO 处理器，但自动化构建过程只支持 FOP。

J.3.4. 纯文本文件

安装说明也以纯文本形式分发，以便在没有更好的阅读工具时使用。`INSTALL` 文件对应第 16 章，并针对不同语境作了少量调整。要重新生成该文件，请切换到`doc/src/sgml` 目录，然后输入**`make INSTALL`**。

过去，发行说明和回归测试说明也曾以纯文本形式分发，但现已停止这种做法。

[10] https://www.postgresql.org/docs/current/

J.3.5. 语法检查

构建文档可能非常耗时。但有一种方法可以只检查文档文件的语法是否正确，这只需要几秒钟：

```
doc/src/sgml$ make check
```

J.4. 文档编写

修改文档源文件最方便的方式，是使用带有 XML 编辑模式的编辑器；如果它还能了解一些 XML 模式语言，从而能够专门识别DocBook 语法，那就更好了。

请注意，出于历史原因，文档源文件现在虽然已经是 XML 文件，但文件扩展名仍然使用`.sgml`。因此你可能需要调整编辑器配置，以设置正确的模式。

J.4.1. Emacs

nXML Mode 随 Emacs 一起提供，是编辑XML 文档时在 Emacs 中最常见的模式。它允许你使用 Emacs 插入标签并检查标记一致性，而且开箱即支持DocBook。详细文档请参阅 nXML manual[11]。

`src/tools/editors/emacs.samples` 包含了这种模式的推荐设置。

J.5. 风格指南

J.5.1. 参考页

参考页应遵循标准布局。这能让用户更快找到所需信息，也能促使作者记录命令的所有相关方面。不仅希望PostgreSQL 参考页彼此保持一致，也希望它们与操作系统及其他软件包提供的参考页保持一致。因此制定了下列准则。它们在很大程度上与多个操作系统所建立的类似准则保持一致。

描述可执行命令的参考页应按以下顺序包含这些小节。不适用的小节可以省略。只有在特殊情况下才应增加顶层小节；这些信息通常应放在“用法”小节中。

名称

本节自动生成。其中包含命令名以及一句简短的功能摘要。

概要

本节包含命令的语法图。概要通常不应列出每一个命令行选项；这些内容会在下文说明。相反，它应列出命令行的主要组成部分，例如输入文件和输出文件的位置。

描述

用若干段解释该命令的作用。

选项

用列表描述每个命令行选项。如果选项很多，可以使用小节。

[11] https://www.gnu.org/software/emacs/manual/html_mono/nxml-mode.html

退出状态

如果程序以 0 表示成功、以非零表示失败，那么你通常不需要记录这一点。如果不同的非零退出代码有各自的含义，就在这里列出它们。

用法

描述程序的任何子语言或运行时接口。如果程序不是交互式的，通常可以省略本节。否则，本节就是用于描述运行时特性的总括性小节。适当时可以使用小节。

环境

列出程序可能使用的所有环境变量。尽量完整；即便像 `SHELL` 这样看似微不足道的变量，也可能让用户感兴趣。

文件

列出程序可能会隐式访问的任何文件。也就是说，不要列出在命令行上指定的输入和输出文件，而应列出配置文件等内容。

诊断

解释程序可能产生的任何不寻常输出。不要去列出所有可能的错误消息。这项工作量很大，而在实践中用处不大。不过，例如如果错误消息有一种用户可以解析的标准格式，那这里就是解释它的地方。

注解

放不进其他地方的任何内容，尤其包括错误、实现缺陷、安全性考虑和兼容性问题。

示例

示例

历史

如果该程序的历史上有一些重要里程碑，可以在这里列出。通常本节可以省略。

作者

作者（仅用于 contrib 部分）

另见

交叉引用按以下顺序列出：其他 PostgreSQL 命令参考页、PostgreSQL SQL 命令参考页、对PostgreSQL 手册的引用、其他参考页（例如操作系统或其他软件包）、其他文档。同一组中的条目按字母顺序排列。

描述 SQL 命令的参考页应包含下列小节：名称、概要、描述、参数、输出、注解、示例、兼容性、历史、另见。参数小节类似于选项小节，但在决定列出命令的哪些子句时有更大的自由度。只有当命令返回的内容不是默认的命令完成标签时，才需要输出小节。兼容性小节应解释该命令在多大程度上符合 SQL 标准，或者它与哪种其他数据库系统兼容。SQL 命令的另见小节应先列出 SQL 命令，再列出对程序的交叉引用。

附录 K. 缩略词

这是在 PostgreSQL 文档和有关 PostgreSQL 的讨论中常用的缩略词列表。

ANSI

美国国家标准协会 [1]

API

应用程序编程接口 [2]

ASCII

美国信息交换标准代码 [3]

BKI

后端接口

CA

证书颁发机构 [4]

CIDR

无类域间路由 [5]

CPAN

Perl 综合档案网络 [6]

CRL

证书吊销列表 [7]

CSV

逗号分隔值 [8]

CTE

公共表表达式

CVE

常见漏洞与暴露 [9]

[1] https://en.wikipedia.org/wiki/American_National_Standards_Institute
[2] https://en.wikipedia.org/wiki/API
[3] https://en.wikipedia.org/wiki/Ascii
[4] https://en.wikipedia.org/wiki/Certificate_authority
[5] https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing
[6] https://www.cpan.org/
[7] https://en.wikipedia.org/wiki/Certificate_revocation_list
[8] https://en.wikipedia.org/wiki/Comma-separated_values
[9] https://cve.mitre.org/

DBA

数据库管理员 [10]

DBI

数据库接口（Perl）[11]

DBMS

数据库管理系统 [12]

DDL

数据定义语言[13]，例如 SQL 命令`CREATE TABLE`、`ALTER USER`

DML

数据操纵语言[14]，例如 SQL 命令`INSERT`、`UPDATE`、`DELETE`

DST

夏令时 [15]

ECPG

用于 PostgreSQL 的嵌入式 C

ESQL

嵌入式 SQL[16]

FAQ

常见问题 [17]

FSM

空闲空间映射

GEQO

遗传查询优化器

GIN

通用倒排索引

[10] https://en.wikipedia.org/wiki/Database_administrator
[11] https://dbi.perl.org/
[12] https://en.wikipedia.org/wiki/Dbms
[13] https://en.wikipedia.org/wiki/Data_Definition_Language
[14] https://en.wikipedia.org/wiki/Data_Manipulation_Language
[15] https://en.wikipedia.org/wiki/Daylight_saving_time
[16] https://en.wikipedia.org/wiki/Embedded_SQL
[17] https://en.wikipedia.org/wiki/FAQ

GiST

通用搜索树

Git

Git[18]

GMT

格林尼治标准时间 [19]

GSSAPI

通用安全服务应用程序接口 [20]

GUC

Grand Unified Configuration，即负责处理服务器配置的PostgreSQL 子系统

HBA

基于主机的认证

HOT

堆内元组

IEC

国际电工委员会 [21]

IEEE

电气和电子工程师学会 [22]

IPC

进程间通信 [23]

ISO

国际标准化组织 [24]

ISSN

国际标准连续出版物编号 [25]

[18] https://en.wikipedia.org/wiki/Git_(software)
[19] https://en.wikipedia.org/wiki/GMT
[20] https://en.wikipedia.org/wiki/Generic_Security_Services_Application_Program_Interface
[21] https://en.wikipedia.org/wiki/International_Electrotechnical_Commission
[22] https://standards.ieee.org/
[23] https://en.wikipedia.org/wiki/Inter-process_communication
[24] https://www.iso.org/home.html
[25] https://en.wikipedia.org/wiki/Issn

JDBC

Java 数据库连接 [26]

JIT

即时编译 [27]

JSON

JavaScript 对象表示法 [28]

LDAP

轻量级目录访问协议 [29]

LSN

日志序列号，参见 `pg_lsn` 和 WAL 内部机制。

MSVC

Microsoft Visual C[30]

MVCC

多版本并发控制

NLS

本地语言支持 [31]

ODBC

开放数据库连接 [32]

OID

对象标识符

OLAP

联机分析处理 [33]

OLTP

联机事务处理 [34]

[26] https://en.wikipedia.org/wiki/Java_Database_Connectivity
[27] https://en.wikipedia.org/wiki/Just-in-time_compilation
[28] https://www.json.org
[29] https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol
[30] https://en.wikipedia.org/wiki/Visual_C++
[31] https://en.wikipedia.org/wiki/Internationalization_and_localization
[32] https://en.wikipedia.org/wiki/Open_Database_Connectivity
[33] https://en.wikipedia.org/wiki/Olap
[34] https://en.wikipedia.org/wiki/OLTP

ORDBMS

对象关系数据库管理系统 [35]

PAM

可插拔认证模块 [36]

PGSQL

PostgreSQL

PGXS

PostgreSQL 扩展系统

PID

进程标识符 [37]

PITR

时间点恢复（持续归档）

PL

过程语言（服务器端）

POSIX

可移植操作系统接口 [38]

RDBMS

关系数据库管理系统 [39]

RFC

征求意见稿 [40]

SGML

标准通用标记语言 [41]

SPI

服务器编程接口

SP-GiST

空间分区通用搜索树

[35] https://en.wikipedia.org/wiki/ORDBMS
[36] https://en.wikipedia.org/wiki/Pluggable_Authentication_Modules
[37] https://en.wikipedia.org/wiki/Process_identifier
[38] https://en.wikipedia.org/wiki/POSIX
[39] https://en.wikipedia.org/wiki/Relational_database_management_system
[40] https://en.wikipedia.org/wiki/Request_for_Comments
[41] https://en.wikipedia.org/wiki/SGML

SQL

结构化查询语言 [42]

SRF

集合返回函数

SSH

安全外壳协议 [43]

SSL

安全套接字层 [44]

SSPI

安全支持提供程序接口 [45]

SYSV

Unix System V[46]

TCP/IP

传输控制协议（TCP）/ 互联网协议（IP）[47]

TID

元组标识符

TLS

传输层安全 [48]

TOAST

超长属性存储技术

TPC

事务处理性能委员会 [49]

URL

统一资源定位符 [50]

[42] https://en.wikipedia.org/wiki/SQL
[43] https://en.wikipedia.org/wiki/Secure_Shell
[44] https://en.wikipedia.org/wiki/Secure_Sockets_Layer
[45] https://msdn.microsoft.com/en-us/library/aa380493%28VS.85%29.aspx
[46] https://en.wikipedia.org/wiki/System_V
[47] https://en.wikipedia.org/wiki/Transmission_Control_Protocol
[48] https://en.wikipedia.org/wiki/Transport_Layer_Security
[49] http://www.tpc.org/
[50] https://en.wikipedia.org/wiki/URL

UTC

协调世界时 [51]

UTF

Unicode 转换格式 [52]

UTF8

8 位 Unicode 转换格式 [53]

UUID

通用唯一标识符

WAL

预写式日志

XID

事务 ID

XML

可扩展标记语言[54]

[51] https://en.wikipedia.org/wiki/Coordinated_Universal_Time
[52] http://www.unicode.org/
[53] https://en.wikipedia.org/wiki/Utf8
[54] https://en.wikipedia.org/wiki/XML

附录 L. 过时或重命名的功能

有时，某些功能会从 PostgreSQL 中移除，某些特性、设置和文件名会发生变化，文档也可能移到别处。本节会将来自旧版本文档或外部链接的用户引导到适当的新位置，以便找到他们所需的信息。

L.1. pg_xlogdump 重命名为 pg_waldump

PostgreSQL 9.6 及以下版本提供了名为 `pg_xlogdump` 的命令，用于读取预写式日志（WAL）文件。该命令已重命名为`pg_waldump`；关于 `pg_waldump` 的文档，请参见 pg_waldump；关于这一变更的详情，请参见 PostgreSQL 10 的发布说明。

L.2. pg_resetxlog 重命名为 pg_resetwal

PostgreSQL 9.6 及以下版本提供了名为 `pg_resetxlog` 的命令，用于重置预写式日志（WAL）文件。该命令已重命名为`pg_resetwal`；有关 `pg_resetwal` 的文档，请参见 pg_resetwal；有关这一变更的详情，请参见PostgreSQL 10 的发布说明。

L.3. pg_receivexlog 重命名为 pg_receivewal

PostgreSQL 9.6 及以下版本提供了名为 `pg_receivexlog` 的命令，用于获取预写式日志（WAL）文件。该命令已重命名为`pg_receivewal`；关于 `pg_receivewal` 的文档，请参见 pg_receivewal；关于这一变更的详情，请参见PostgreSQL 10 的发布说明。

参考书目

这里列出了一些关于 SQL 和 PostgreSQL 的参考文献与读物。

来自最初的 POSTGRES 开发团队的一些白皮书和技术报告，可在加州大学伯克利分校计算机系的网站[1]上获取。

SQL 参考书

[bowman01] The Practical SQL Handbook. Using SQL Variants. Fourth Edition. Judith Bowman、Sandra Emerson 和 Marcy Darnovsky. ISBN 0-201-70309-2. Addison-Wesley Professional. 2001.

[date97] A Guide to the SQL Standard. A user's guide to the standard database language SQL. Fourth Edition. C. J. Date 和 Hugh Darwen. ISBN 0-201-96426-0. Addison-Wesley. 1997.

[date04] An Introduction to Database Systems. Eighth Edition. C. J. Date. ISBN 0-321-19784-4. Addison-Wesley. 2003.

[elma04] Fundamentals of Database Systems. Fourth Edition. Ramez Elmasri 和 Shamkant Navathe. ISBN 0-321-12226-7. Addison-Wesley. 2003.

[melt93] Understanding the New SQL. A complete guide. Jim Melton 和 Alan R. Simon. ISBN 1-55860-245-3. Morgan Kaufmann. 1993.

[ull88] Principles of Database and Knowledge-Base Systems. Classical Database Systems. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.

PostgreSQL 相关文档

[sim98] Enhancement of the ANSI SQL Implementation of PostgreSQL. Stefan Simkovics.

该文讨论了 SQL 的历史和语法，并描述了将`INTERSECT` 和 `EXCEPT` 构造加入 PostgreSQL 的过程。它是在维也纳技术大学 O. Univ. Prof. Dr. Georg Gottlob 和 Univ. Ass. Mag. Katrin Seyr 的支持下撰写的硕士学位论文。

Department of Information Systems, Vienna University of Technology. Vienna, Austria. November 29, 1998.

[yu95] The Postgres 95 User Manual. A. Yu 和 J. Chen. University of California. Berkeley, California. Sept. 5, 1995.

[fong] The design and implementation of the POSTGRES query optimizer[2]. Zelaine Fong. University of California, Berkeley, Computer Science Department.

[1] https://dsf.berkeley.edu/papers/
[2] https://dsf.berkeley.edu/papers/UCB-MS-zfong.pdf

会议论文和期刊文章

[ports12] “Serializable Snapshot Isolation in PostgreSQL[3]” . D. Ports 和 K. Grittner. VLDB Conference, August 2012.

[berenson95] “A Critique of ANSI SQL Isolation Levels[4]” . H. Berenson、P. Bernstein、J. Gray、J. Melton、E. O'Neil 和 P. O'Neil. ACM-SIGMOD Conference on Management of Data, June 1995.

[olson93] Partial indexing in POSTGRES: research project. Nels Olson. UCB Engin T7.49.1993 O676. University of California. Berkeley, California. 1993.

[ong90] “A Unified Framework for Version Modeling Using Production Rules in a Database System” . L. Ong 和 J. Goh. ERL Technical Memorandum M90/33. University of California. Berkeley, California. April, 1990.

[rowe87] “The POSTGRES data model[5]” . L. Rowe 和 M. Stonebraker. VLDB Conference, Sept. 1987.

[seshadri95] “Generalized Partial Indexes[6]” . P. Seshadri 和 A. Swami. Eleventh International Conference on Data Engineering, 1995 年 3 月 6-10 日. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, California. 1995. 420-7.

[ston86] “The design of POSTGRES[7]” . M. Stonebraker 和 L. Rowe. ACM-SIGMOD Conference on Management of Data, May 1986.

[ston87a] “The design of the POSTGRES. rules system” . M. Stonebraker、E. Hanson 和 C. H. Hong. IEEE Conference on Data Engineering, Feb. 1987.

[ston87b] “The design of the POSTGRES storage system[8]” . M. Stonebraker. VLDB Conference, Sept. 1987.

[ston89] “A commentary on the POSTGRES rules system[9]” . M. Stonebraker、M. Hearst 和 S. Potamianos. SIGMOD Record 18(3). Sept. 1989.

[ston89b] “The case for partial indexes[10]” . M. Stonebraker. SIGMOD Record 18(4). Dec. 1989. 4-11.

[ston90a] “The implementation of POSTGRES[11]” . M. Stonebraker、L. A. Rowe 和 M. Hirohama. Transactions on Knowledge and Data Engineering 2(1). IEEE. March 1990.

[ston90b] “On Rules, Procedures, Caching and Views in Database Systems[12]” . M. Stonebraker、A. Jhingran、J. Goh 和 S. Potamianos. ACM-SIGMOD Conference on Management of Data, June 1990.

[3] https://arxiv.org/pdf/1208.4179
[4] https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf
[5] https://dsf.berkeley.edu/papers/ERL-M87-13.pdf
[6] https://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.5740
[7] https://dsf.berkeley.edu/papers/ERL-M85-95.pdf
[8] https://dsf.berkeley.edu/papers/ERL-M87-06.pdf
[9] https://dsf.berkeley.edu/papers/ERL-M89-82.pdf
[10] https://dsf.berkeley.edu/papers/ERL-M89-17.pdf
[11] https://dsf.berkeley.edu/papers/ERL-M90-34.pdf
[12] https://dsf.berkeley.edu/papers/ERL-M90-36.pdf

索引

符号

$，45
$libdir，1056
$libdir/plugins，607，1762
*，128
.pgpass，843
.pg_service.conf，843
::，52
_PG_fini，1055
_PG_init，1055
_PG_output_plugin_init，1358
下标，45
不可重复读，452
中位数，49
  (参见百分位点)
串 (见字符串)
串行化异常，452，455
临时命名关系
  从 SPI 注销，1324
  向 SPI 注册，1323，1325
主机名，790
主键，69
事件日志
  事件日志，552
事件触发器，1128
  在 C 中，1133
  在 PL/Tcl 中，1249
事务，19
事务 ID
  回卷，668
事务日志 (见 WAL)
事务隔离，452
事务隔离级别，452，602
  可串行化，455
  可重复读，454
  设置，1829
  设置默认值，602
  读已提交，453
二进制串
  串接，236
  长度，237
二进制数据，149
  函数，235
交叉编译，506
交叉连接，113
仅索引扫描，403
任意精度数值，143
休眠，277
众数
  统计，336
位串
  函数，239
  常量，40
  数据类型，169
位图扫描，398，580
例程，1034
保存点
  回滚，1790
  定义，1792
  释放，1781
信号
  后端进程，364
信号量，533
信息模式，983
值表达式，44
假想集聚合
  内置，337
儒略日，2319
全局数据
  在 PL/Python 中，1280
  在 PL/Tcl 中，1245
全文检索，409
  函数和操作符，170
  数据类型，170
公共表表达式 (见 WITH)
共享内存，533
共享库，512，1063
关系，7
关系数据库，7
关键字
  列表，2320
关键词
  语法，35
关闭，542
内存上下文
  在 SPI 中，1334
内存过量分配，540
准备一个查询
  在 PL/pgSQL 中，1227
准备查询
  在 PL/Python 中，1282
  在 PL/Tcl 中，1246
函数，211
  RETURNS TABLE，1050
  位置记法，60
  使用 SETOF，1045
  内部，1055
  参数的默认值，1044
  可变参数，1043
  命名参数，1035
  命名记法，60
  在 FROM 子句中，118
  多态，1033
  混合记法，61

用户定义的，1034
在 C 中，1055
在 SQL 中，1034
调用，46
调用中的类型解析，386
输出参数，1042
函数依赖，124
分区，92
分区剪枝，102
分区表，92
分组，122
切片面包 (见 TOAST)
列，7，62
增加，74
移除，75
系统列，73
重命名，77
列引用，45
列的数据类型
更改，76
初始化分支，2275
删除，110
别名
在 FROM 子句中，117
在选择列表中，129
查询中的表名，13
加密，546
针对特定列，2575
动态装载，608
动态载入，1055
区域设置，529，651
匿名代码块，1675
升级，543
协方差
总体，333
样本，333
协议
前端-后端，2116
单用户模式，2025
即时编译 (见 JIT)
历史
PostgreSQL 的，xxxiv
参数
语法，45
反斜线转义，37
取消
SQL 命令，822
受信任的
PL/Perl，1265
只读事务，602
设置，1829
设置默认值，602
可串行化，455
可串行化快照隔离，452
可变参数函数，1043
可延迟事务，603
设置，1829
设置默认值，602
可更新视图，1662
可见性映射，2274
可重复读，454
右连接，114
合取，211
同步复制，691
同步提交，755
名称
语法，35
限定，85
非限定，86
后台工作进程，1351
否定，211
咨询锁，462
唯一约束，68
回卷
事务 ID 的，668
多事务 ID 的，670
回归截距，334
回归斜率，334
回归测试，511，768
圆括号，44
域，206
基础类型，1032
备份，365，675
备库，691
复制，691
复制槽
流复制，697
逻辑复制，1356
复制源，1363
复制进度跟踪，1363
复合类型，194，1032
常量，196
构造器，56
比较，342
外连接，114
外部数据，105
外部数据包装器
处理器，2184
外部表，105
外键，18，70
自引用，71
多态函数，1033
多态类型，1033
多版本并发控制，452
大对象，860
大小写敏感性
SQL 命令，36
子事务

在 PL/Tcl 中，1251
子查询，14，53，118，340
字段
计算，200
字段选择，46
字符串
串接，220
反斜线引号，610
常量，37
数据类型，147
符合标准，611
转义警告，610
长度，221
字符集，606，613，659
存储参数，1615
安装，498
在 Windows 上，522
客户端认证，619
期间超时，561
容器类型，1032
密码，640
认证，627
超级用户的，529
密码文件，843
对象标识符
数据类型，207
层次数据库，7
嵌入式 SQL
在 C 中，871
左连接，114
已启用角色，1000
布尔
操作符 (见操作符，逻辑)
数据类型，161
常量，37
平均值，330
并发，452
并行查询，486
幻读，452
序列，317
与 serial 类型，146
序号，348
库初始化函数，1055
库终结函数，1055
延迟，277
异常
在 PL/pgSQL 中，1202
在 PL/Tcl 中，1250
异步提交，755
引号
与标识符，36
转义，37
引导
期间启动服务器，530
引用完整性，18，70
循环
在 PL/pgSQL 中，1197
快速路径，823
性能，466
所有者，77
扩展，1106
外部维护，2667
扩展 SQL，1032
报告错误
在 PL/pgSQL 中，1214
持续归档，675
在备库中，701
排他约束，73
排序，131
排序操作符，1103
排序规则，653
在 PL/pgSQL 中，1180
在 SQL 函数中，1052
接口
外部维护，2667
控制文件，1107
插入，108
搜索路径，86
对象可见性，354
当前，349
操作符，211
优先级，42
用户定义的，1089
语法，41
调用，46
调用中的类型解析，382
逻辑，211
操作符族，406，1101
操作符类，406，1094
支持函数
in_range，2232
故障切换，691
数字
常量，40
数据分区，691
数据区域 (见数据库集簇)
数据库，645
创建，3
创建权限，639
数据库活动
监控，714
数据库集簇，7，528
数据类型，141
内部组织，1057
分类，382
域，206
基础，1032
复合，1032

多态，1033
容器，1032
常量，41
数字，142
枚举（enum），162
用户定义的，1084
类型转换，52
转换，381
数组，183
I/O，193
修改，189
声明，184
常量，185
搜索，192
构造器，54
用户定义类型的，1087
访问，186
文件系统挂载点，529
文本检索，409
函数和操作符，170
数据类型，170
索引，446
文档
文本检索，410
断言
在 PL/pgSQL 中，1216
方差
总体，335
样本，336
日常维护，665
日志传送，691
日志写到哪里，586
日志记录
current_logfiles 文件和 pg_current_logfile 函数，350
pg_current_logfile 函数，350
日期
当前，275
时区，157，605
POSIX 风格规范，2316
转换，274
输入缩写，2315
时区名称，605
时区数据，506
时间
当前，275
时间点恢复，675
时间线，675
时间跨度，151
易变性
函数，1053
更新，109
有序集聚合，47
内置，336
有效数字，606
服务器日志，586
日志文件维护，673
服务器欺骗，546
权限，77
与规则，1162
模式，87
与视图，1162
查询，351
权限（permission）(见权限)
条件表达式，319
析取，211
枚举类型，162
查询，9，112
查询树，1137
查询计划，466
标准差，335
总体，335
样本，335
标签 (见别名)
标识符
语法，35
长度，35
标量 (见表达式)
格式化，255
格里高利历，2318
检查点，756
检查约束，64
模式，84，645
public，86
创建，85
在 psql 和 pg_dump 中，1963
当前，86，349
移除，85
模式匹配，240
正则表达式，241，242
(参见模式匹配)
与区域设置，652
死锁，461
期间超时，609
比较
复合类型，342
子查询结果行，340
操作符，211
行构造器，342
注释
关于数据库对象，360
在 SQL 中，42
流复制，691
测试，768
浮点，144
浮点数
显示，606
清理，665

温备，691
游标
  CLOSE，1489
  DECLARE，1666
  FETCH，1740
  MOVE，1766
  在 PL/pgSQL 中，1206
  显示查询计划，1735
热备，691
版本
  兼容性，543
物化视图，2103
  通过规则实现，1147
环境变量，841
用户，638
  当前，349
用户名映射，625
用户映射，105
登录权限，639
百分位数
  离散，336
百分位点
  连续，336
监控
  数据库活动，714
目标列表，1138
相关性，333
  查询规划器中的，478
矩形，165
磁盘使用情况，751
磁盘空间，666
磁盘驱动器，759
移动聚合模式，1079
空值
  用于检查约束，66
  在 DISTINCT 中，129
  在 PL/Perl 中，1254
  在 PL/Python 中，1275
  用于唯一约束，69
  比较，213
  默认值，63
空闲空间映射，2274
窗口函数，21
  内置，338
  执行顺序，128
  调用，50
端口，790
策略，78
管理工具
  外部维护，2667
管道化
  协议规范，2125
类型 (见数据类型)
类型转换，40，52
  I/O 转换，1517
系统目录
  模式，87
索引，394
  B-树，395，2231
  BRIN，396，2263
  基于表达式，399
  GIN，396，2257
    文本检索，446
  GiST，395，2234
    文本检索，446
  hash，395
  Hash，2268
  SP-GiST，396，2246
  与 ORDER BY，397
  仅索引扫描，403
  唯一，399
  多列，397
  并发构建，1556
  检查使用情况，408
  用于用户定义的数据类型，1093
  组合多个索引，398
  覆盖，403
  部分，400
  锁，465
索引扫描，580
约束，64
  NOT NULL，67
  主键，69
  名称，65
  唯一，68
  增加，75
  外键，70
  排他，73
  检查，64
  移除，76
约束排除，104，584
级联复制，691
线性回归，333
线段，165
线程
  用于 libpq，848
统计，333
统计信息，715
  规划器的，477，478，667
继承，24，89
维护，665
编译
  libpq 应用，849
网络
  数据类型，167
网络文件系统，529
网络附加存储（NAS）(见网络文件系统)
美元引用，39

聚合函数，13
  内置，329
  可变参数，1080
  多态，1080
  支持函数，1084
  有序集，1082
  用户定义的，1077
  移动聚合，1079
  调用，47
  部分聚合，1083
脏读，452
自动提交
  psql，1964
  批量装载数据，482
自动清理
  一般信息，671
  配置参数，598
自动递增 (见 serial)
自定义扫描提供者
  处理器，2206
自然连接，114
范围类型，201
  排他约束，206
  索引，205
范围表，1137
行，7，62
行估计
  多元，2293
  规划器，2287
行比较，342
行类型，194
  构造器，56
行级安全，78
表，7，62
  修改，74
  分区，92
  创建，62
  移除，63
  继承，89
  重命名，77
表函数，118
  XMLTABLE，300
表空间，648
  临时，601
  默认，601
表表达式，112
表达式
  计算的顺序，57
  语法，44
表采样方法，2203
覆盖索引，403
见证服务器，691
规则，1137
  用于 DELETE，1150
  用于 INSERT，1150
  用于 SELECT，1139
  与触发器比较，1165
  用于 UPDATE，1150
  和物化视图，1147
  和视图，1139
视图，18
  更新，1156
  物化，1147
  通过规则实现，1139
角色，638，643
  创建权限，639
  发起复制的权限，640
  已启用，1000
  成员资格，640
  绕过权限，640
  继承权限，640
  连接限制，640
  适用，984
触发器，1118
  与规则比较，1165
  在 C 中，1121
  在 PL/pgSQL 中，1216
  在 PL/Python 中，1280
  在 PL/Tcl 中，1247
  用于更新一个派生的 tsvector 列，426
  触发器函数的参数，1120
计算字段，200
记法
  函数，59
证书，635
词元，35
语法
  SQL，35
读已提交，453
谓词锁，455
负载均衡，691
超时
  客户端认证，561
  死锁，609
超级用户，5，639
路径
  用于模式，600
转义字符串
  libpq 中的，814
转义字符串语法，37
输入函数，1084
输出函数，1084
过渡表，1641
  (参见临时命名关系)
  从 C 触发器中引用，1121
  在 PL 中实现，1325
过程
  用户定义的，1034

过程语言，1169
  外部维护，2667
  调用处理器，2181
过量分配，540
连接，11，113
  交叉，113
  右，114
  外连接，12，114
  左，114
  控制顺序，480
  自然，114
  自连接，13
连接服务文件，843
适用角色，984
逆分布，336
透明巨型页，564
通知处理
  在 libpq 中，834
通知处理器，834
通知接收器，834
逻辑解码，1354，1356
遗传查询优化，583
配置
  服务器
    函数，364
  恢复
    备库，710
  服务器端，554
重启点，757
重复，11，129
重建索引，672
重载
  函数，1053
  操作符，1089
锁，457
  咨询，462
  监控，740
错误代码
  列表，2304
错误消息，798
错误码
  libpq，807
长度
  二进制串 (见二进制串，长度)
  字符串 (见字符串，长度)
限定名，85
集合交，130
集合差，130
集合并，130
集合操作，130
集合返回函数
  函数，345
集簇，691
  数据库 (见数据库集簇)
非持久，485
非数
  double precision，145
  numeric（数据类型），144
非空约束，67
非阻塞连接，783，817
非限定名，86
面向对象数据库，7
顺序扫描，581
预备语句
  创建，1770
  执行，1734
  显示查询计划，1735
  释放，1665
高可用，691
魔数块，1055
默认值，63
  更改，76

A

abbrev，282
ABORT，1374
abs，215
acos，219
acosd，219
adminpack，2473
age，263
AIX
  IPC 配置，535
  在其上安装，514
akeys，2540
ALL，340，342
allow_in_place_tablespaces 配置参数，614
allow_system_table_mods 配置参数，614
ALTER AGGREGATE，1375
ALTER COLLATION，1377
ALTER CONVERSION，1379
ALTER DATABASE，1381
ALTER DEFAULT PRIVILEGES，1384
ALTER DOMAIN，1387
ALTER EVENT TRIGGER，1391
ALTER EXTENSION，1392
ALTER FOREIGN DATA WRAPPER，1395
ALTER FOREIGN TABLE，1397
ALTER FUNCTION，1402
ALTER GROUP，1406
ALTER INDEX，1408
ALTER LANGUAGE，1411
ALTER LARGE OBJECT，1412
ALTER MATERIALIZED VIEW，1413
ALTER OPERATOR，1415
ALTER OPERATOR CLASS，1417
ALTER OPERATOR FAMILY，1419

ALTER POLICY，1423
ALTER PROCEDURE，1425
ALTER PUBLICATION，1428
ALTER ROLE，640，1430
ALTER ROUTINE，1434
ALTER RULE，1436
ALTER SCHEMA，1437
ALTER SEQUENCE，1438
ALTER SERVER，1441
ALTER STATISTICS，1443
ALTER SUBSCRIPTION，1444
ALTER SYSTEM，1446
ALTER TABLE，1448
ALTER TABLESPACE，1463
ALTER TEXT SEARCH CONFIGURATION，1465
ALTER TEXT SEARCH DICTIONARY，1467
ALTER TEXT SEARCH PARSER，1469
ALTER TEXT SEARCH TEMPLATE，1470
ALTER TRIGGER，1471
ALTER TYPE，1473
ALTER USER，1477
ALTER USER MAPPING，1478
ALTER VIEW，1480
amcheck，2474
ANALYZE，667，1482
AND（操作符），211
any，209
ANY，332，340，342
anyarray，209
anyelement，209
anyenum，209
anynonarray，209
anyrange，209
application_name 配置参数，591
archive_cleanup_command 恢复参数，710
archive_command 配置参数，575
archive_mode 配置参数，575
archive_timeout 配置参数，575
area，280
armor，2579
ARRAY，54
  结果类型的确定，391
array_agg，329，2546
array_append，323
array_cat，323
array_dims，323
array_fill，323
array_length，323
array_lower，323
array_ndims，323
array_nulls 配置参数，610
array_position，323
array_positions，323
array_prepend，323
array_remove，323
array_replace，323
array_to_json，310
array_to_string，323
array_to_tsvector，287
array_upper，323
ascii，222
asin，219
asind，219
ASSERT
  在 PL/pgSQL 中，1216
AT TIME ZONE，274
atan，220
atan2，220
atan2d，220
atand，220
authentication_timeout 配置参数，561
auth_delay，2477
auth_delay.milliseconds 配置参数，2477
autovacuum_analyze_scale_factor 配置参数，599
autovacuum_analyze_threshold 配置参数，599
autovacuum_freeze_max_age 配置参数，599
autovacuum_max_workers 配置参数，599
autovacuum_multixact_freeze_max_age 配置参数，600
autovacuum_naptime 配置参数，599
autovacuum_vacuum_cost_delay 配置参数，600
autovacuum_vacuum_cost_limit 配置参数，600
autovacuum_vacuum_scale_factor 配置参数，599
autovacuum_vacuum_threshold 配置参数，599
autovacuum_work_mem 配置参数，565
autovacuum 配置参数，598
auto_explain，2478
auto_explain.log_analyze 配置参数，2478
auto_explain.log_buffers 配置参数，2478
auto_explain.log_format 配置参数，2479
auto_explain.log_min_duration 配置参数，2478
auto_explain.log_nested_statements 配置参数，2479
auto_explain.log_timing 配置参数，2478
auto_explain.log_triggers 配置参数，2479
auto_explain.log_verbose 配置参数，2479
auto_explain.sample_rate 配置参数，2479
avals，2540
avg，330

B

B-树 (见索引)
backend_flush_after 配置参数，569
backslash_quote 配置参数，610
BASE_BACKUP，2136

BEGIN，1485
BETWEEN，213
BETWEEN SYMMETRIC，213
BGWORKER_BACKEND_DATABASE_
CONNECTION，1352
BGWORKER_SHMEM_ACCESS，1351
bgwriter_delay 配置参数，567
bgwriter_flush_after 配置参数，568
bgwriter_lru_maxpages 配置参数，568
bgwriter_lru_multiplier 配置参数，568
bigint，40，143
bigserial，146
bison，499
bit_and，330
bit_length，221
bit_or，330
BLOB (见大对象)
block_size 配置参数，612
bloom，2481
bonjour_name 配置参数，560
bonjour 配置参数，560
bool_and，330
bool_or，330
box，280
box（数据类型），165
BRIN (见索引)
brin_desummarize_range，374
brin_metapage_info，2568
brin_page_items，2568
brin_page_type，2568
brin_revmap_data，2568
brin_summarize_new_values，374
brin_summarize_range，374
broadcast，283
BSD 认证，636
btree_gin，2485
btree_gist，2486
btrim，223，237
bt_index_check，2474
bt_index_parent_check，2475
bt_metap，2566
bt_page_items，2567，2567
bt_page_stats，2566
bytea，149
bytea_output 配置参数，604

C

C，781，871
C++，1077
CALL，1487
cardinality，323
CASCADE
  用于 DROP，106
  外键动作，72
CASE，319
  结果类型的确定，391
cbrt，215
ceil，215
ceiling，216
center，280
char，147
character，147
character varying，147
char_length，221
CHECK OPTION，1660
CHECKPOINT，1488
checkpoint_completion_target 配置参数，574
checkpoint_flush_after 配置参数，574
checkpoint_timeout 配置参数，574
checkpoint_warning 配置参数，574
check_function_bodies 配置参数，602
chr，223
cid，207
cidr，167
circle，166，280
citext，2488
client_encoding 配置参数，606
client_min_messages 配置参数，600
clock_timestamp，263
CLOSE，1489
CLUSTER，1491
clusterdb，1851
cluster_name 配置参数，597
cmax，73
cmin，73
COALESCE，321
COLLATE，53
collation for，355
col_description，360
COMMENT，1493
COMMIT，1497
COMMIT PREPARED，1498
commit_delay 配置参数，573
commit_siblings 配置参数，574
concat，223
concat_ws，223
configure，500
config_file 配置参数，558
connectby，2632，2639
conninfo，788
constraint_exclusion 配置参数，584
CONTINUE
  在 PL/pgSQL 中，1198
convert，223
convert_from，223
convert_to，224
COPY，9，1499

使用 libpq，825
corr，333
cos，220
cosd，220
cot，220
cotd，220
count，330
covar_pop，333
covar_samp，333
cpu_index_tuple_cost 配置参数，582
cpu_operator_cost 配置参数，582
cpu_tuple_cost 配置参数，582
CREATE ACCESS METHOD，1509
CREATE AGGREGATE，1510
CREATE CAST，1517
CREATE COLLATION，1522
CREATE CONVERSION，1524
CREATE DATABASE，645，1526
CREATE DOMAIN，1529
CREATE EVENT TRIGGER，1532
CREATE EXTENSION，1534
CREATE FOREIGN DATA WRAPPER，1537
CREATE FOREIGN TABLE，1539
CREATE FUNCTION，1543
CREATE GROUP，1552
CREATE INDEX，1553
CREATE LANGUAGE，1561
CREATE MATERIALIZED VIEW，1564
CREATE OPERATOR，1566
CREATE OPERATOR CLASS，1569
CREATE OPERATOR FAMILY，1572
CREATE POLICY，1573
CREATE PROCEDURE，1579
CREATE PUBLICATION，1582
CREATE ROLE，638，1584
CREATE RULE，1589
CREATE SCHEMA，1592
CREATE SEQUENCE，1595
CREATE SERVER，1599
CREATE STATISTICS，1601
CREATE SUBSCRIPTION，1603
CREATE TABLE，8，1606
CREATE TABLE AS，1626
CREATE TABLESPACE，649，1629
CREATE TEXT SEARCH CONFIGURATION，1631
CREATE TEXT SEARCH DICTIONARY，1633
CREATE TEXT SEARCH PARSER，1635
CREATE TEXT SEARCH TEMPLATE，1637
CREATE TRANSFORM，1639
CREATE TRIGGER，1641
CREATE TYPE，1648
CREATE USER，1657
CREATE USER MAPPING，1658
CREATE VIEW，1660
createdb，3，646，1854
createuser，638，1857
CREATE_REPLICATION_SLOT，2132
crosstab，2633，2635，2636
crypt，2576
cstring，209
ctid，74
CTID，1145
CUBE，125
cube（扩展），2491
cume_dist，338
  假设行，337
current_catalog，349
current_database，349
current_date，263
current_logfiles
  和 log_destination 配置参数，586
  和 pg_current_logfile 函数，350
current_query，349
current_role，349
current_schema，349
current_schemas，349
current_setting，364
current_time，264
current_timestamp，264
current_user，349
currval，317
cursor_tuple_fraction 配置参数，585
Cygwin
  在其上安装，517

D

data_checksums 配置参数，612
data_directory_mode 配置参数，612
data_directory 配置参数，558
data_sync_retry 配置参数，612
date，151，153
  常量，155
  输出格式，156
    (参见格式化)
DateStyle 配置参数，605
date_part，264，269
date_trunc，264，273
dblink，2496，2501
dblink_build_sql_delete，2523
dblink_build_sql_insert，2521
dblink_build_sql_update，2525
dblink_cancel_query，2519
dblink_close，2510
dblink_connect，2497
dblink_connect_u，2499
dblink_disconnect，2500
dblink_error_message，2513

dblink_exec，2504
dblink_fetch，2508
dblink_get_connections，2512
dblink_get_notify，2516
dblink_get_pkey，2520
dblink_get_result，2517
dblink_is_busy，2515
dblink_open，2506
dblink_send_query，2514
db_user_namespace 配置参数，561
deadlock_timeout 配置参数，609
DEALLOCATE，1665
dearmor，2579
debug_assertions 配置参数，613
debug_deadlocks 配置参数，616
debug_pretty_print 配置参数，591
debug_print_parse 配置参数，591
debug_print_plan 配置参数，591
debug_print_rewritten 配置参数，591
decimal (见 numeric)
DECLARE，1666
decode，224，237
decode_bytea
  在 PL/Perl 中，1263
decrypt，2583
decrypt_iv，2583
default_statistics_target 配置参数，584
default_tablespace 配置参数，601
default_text_search_config 配置参数，606
default_transaction_deferrable 配置参数，602
default_transaction_isolation 配置参数，602
default_transaction_read_only 配置参数，602
default_with_oids 配置参数，610
defined，2541
degrees，216
DELETE，17，110，1670
  RETURNING，110
delete，2541
dense_rank，338
  假设行，337
diameter，280
dict_int，2527
dict_xsyn，2528
difference，2535
digest，2575
DISCARD，1673
DISTINCT，11，129
div，216
dmetaphone，2537
dmetaphone_alt，2537
DO，1675
double precision，144
DROP ACCESS METHOD，1677
DROP AGGREGATE，1678
DROP CAST，1680
DROP COLLATION，1681
DROP CONVERSION，1682
DROP DATABASE，648，1683
DROP DOMAIN，1684
DROP EVENT TRIGGER，1685
DROP EXTENSION，1686
DROP FOREIGN DATA WRAPPER，1687
DROP FOREIGN TABLE，1688
DROP FUNCTION，1689
DROP GROUP，1691
DROP INDEX，1692
DROP LANGUAGE，1694
DROP MATERIALIZED VIEW，1696
DROP OPERATOR，1697
DROP OPERATOR CLASS，1699
DROP OPERATOR FAMILY，1701
DROP OWNED，1703
DROP POLICY，1704
DROP PROCEDURE，1705
DROP PUBLICATION，1707
DROP ROLE，638，1708
DROP ROUTINE，1710
DROP RULE，1711
DROP SCHEMA，1712
DROP SEQUENCE，1714
DROP SERVER，1715
DROP STATISTICS，1716
DROP SUBSCRIPTION，1717
DROP TABLE，8，1719
DROP TABLESPACE，1721
DROP TEXT SEARCH CONFIGURATION，1722
DROP TEXT SEARCH DICTIONARY，1723
DROP TEXT SEARCH PARSER，1724
DROP TEXT SEARCH TEMPLATE，1725
DROP TRANSFORM，1726
DROP TRIGGER，1727
DROP TYPE，1729
DROP USER，1730
DROP USER MAPPING，1731
DROP VIEW，1732
dropdb，648，1862
dropuser，638，1865
DROP_REPLICATION_SLOT，2135
DTD，175
DTrace，507，742
dynamic_library_path，1056
dynamic_library_path 配置参数，608
dynamic_shared_memory_type 配置参数，566

E

each，2541
earth，2530

earthdistance，2530
earth_box，2531
earth_distance，2531
ECPG，871
ecpg，1868
effective_cache_size 配置参数，583
effective_io_concurrency 配置参数，568
elog，2165
  在 PL/Perl 中，1263
  在 PL/Python 中，1287
  在 PL/Tcl 中，1247
enable_bitmapscan 配置参数，580
enable_gathermerge 配置参数，580
enable_hashagg 配置参数，580
enable_hashjoin 配置参数，580
enable_indexonlyscan 配置参数，580
enable_indexscan 配置参数，580
enable_material 配置参数，580
enable_mergejoin 配置参数，580
enable_nestloop 配置参数，580
enable_parallel_append 配置参数，580
enable_parallel_hash 配置参数，581
enable_partitionwise_aggregate 配置参数，581
enable_partitionwise_join 配置参数，581
enable_partition_pruning 配置参数，581
enable_seqscan 配置参数，581
enable_sort 配置参数，581
enable_tidscan 配置参数，581
encode，224，237
encode_array_constructor
  在 PL/Perl 中，1264
encode_array_literal
  在 PL/Perl 中，1264
encode_bytea
  在 PL/Perl 中，1263
encode_typed_literal
  在 PL/Perl 中，1264
encrypt，2583
encrypt_iv，2583
END，1733
enum_first，277
enum_last，277
enum_range，277
ereport，2165
escape_string_warning 配置参数，610
event_source 配置参数，589
event_trigger，209
every，331
EXCEPT，130
EXECUTE，1734
exist，2541
EXISTS，340
EXIT
  在 PL/pgSQL 中，1197
exit_on_error 配置参数，612
exp，216
EXPLAIN，466，1735
external_pid_file 配置参数，558
extract，264，269
extra_float_digits 配置参数，606

F

factorial，216
false，161
family，283
fdw_handler，209
FETCH，1740
file_fdw，2532
FILTER，47
first_value，339
flex，499
float4 (见 real)
float8 (见 double precision)
floor，216
force_parallel_mode 配置参数，586
format，224，233
  在 PL/pgSQL 中使用，1186
format_type，355
FreeBSD
  IPC 配置，535
  共享库，1064
  启动脚本，530
from_collapse_limit 配置参数，585
FSM (见空闲空间映射)
fsm_page_contents，2565
fsync 配置参数，570
full_page_writes 配置参数，572
fuzzystrmatch，2535

G

gc_to_sec，2530
generate_series，345
generate_subscripts，347
gen_random_bytes，2583
gen_random_uuid，2583
gen_salt，2576
GEQO (见遗传查询优化)
geqo_effort 配置参数，584
geqo_generations 配置参数，584
geqo_pool_size 配置参数，584
geqo_seed 配置参数，584
geqo_selection_bias 配置参数，584
geqo_threshold 配置参数，583
geqo 配置参数，583
get_bit，237
get_byte，237
get_current_ts_config，287

get_raw_page，2564
GIN (见索引)
gin_clean_pending_list，374
gin_fuzzy_search_limit 配置参数，608
gin_leafpage_items，2569
gin_metapage_info，2569
gin_page_opaque_info，2569
gin_pending_list_limit 配置参数，605
GiST (见索引)
GRANT，77，1744
GREATEST，322
  结果类型的确定，391
GROUP BY，15，122
GROUPING，337
GROUPING SETS，125
GSSAPI，628
GUID，173

H

hash (见索引)
hash_bitmap_info，2570
hash_metapage_info，2571
hash_page_items，2570
hash_page_stats，2570
hash_page_type，2570
has_any_column_privilege，352
has_column_privilege，352
has_database_privilege，352
has_foreign_data_wrapper_privilege，352
has_function_privilege，352
has_language_privilege，352
has_schema_privilege，352
has_sequence_privilege，352
has_server_privilege，352
has_tablespace_privilege，352
has_table_privilege，352
has_type_privilege，352
HAVING，15，124
hba_file 配置参数，558
heap_page_items，2565
heap_page_item_attrs，2565
height，280
hmac，2575
host，283
hostmask，283
hot_standby_feedback 配置参数，579
hot_standby 配置参数，578
HP-UX
  IPC 配置，536
  共享库，1064
  在其上安装，518
hstore，2538，2540
hstore_to_array，2540
hstore_to_json，2540
hstore_to_jsonb，2540
hstore_to_jsonb_loose，2541
hstore_to_json_loose，2541
hstore_to_matrix，2540
huge_pages 配置参数，564

I

icount，2548
ICU，503，655，1523
ident，630
IDENTIFY_SYSTEM，2131
ident_file 配置参数，558
idle_in_transaction_session_timeout 配置参数，603
idx，2548
IFNULL，321
ignore_checksum_failure 配置参数，616
ignore_system_indexes 配置参数，614
IMMUTABLE，1053
IMPORT FOREIGN SCHEMA，1751
IN，340，342
INCLUDE
  在索引定义中，404
include
  配置文件中的，556
include_dir
  配置文件中的，557
include_if_exists
  配置文件中的，556
index，2560
index_am_handler，209
inet_client_addr，350
inet_client_port，350
inet_merge，284
inet_same_family，284
inet_server_addr，350
inet_server_port，350
inet（数据类型），167
initcap，224
initdb，528，1985
INSERT，8，108，1753
  RETURNING，110
instr 函数，1239
int2 (见 smallint)
int4 (见 integer)
int8 (见 bigint)
intagg，2546
intarray，2548
integer，40，143
integer_datetimes 配置参数，613
internal，209
INTERSECT，130

interval，151，158
  输出格式，160
    (参见格式化)
IntervalStyle 配置参数，605
intset，2548
int_array_aggregate，2546
int_array_enum，2546
in_range 支持函数，2232
IS DISTINCT FROM，213，342
IS DOCUMENT，298
IS FALSE，214
IS NOT DISTINCT FROM，213，342
IS NOT DOCUMENT，298
IS NOT FALSE，214
IS NOT NULL，213
IS NOT TRUE，214
IS NOT UNKNOWN，214
IS NULL，213，611
IS TRUE，214
IS UNKNOWN，214
isclosed，280
isempty，328
isfinite，265
isn，2551
ISNULL，213
isn_weak，2552
isopen，280
is_array_ref
  在 PL/Perl 中，1264
is_valid，2552

J

JIT，765
jit_above_cost 配置参数，583
jit_debugging_support 配置参数，617
jit_dump_bitcode 配置参数，617
jit_expressions 配置参数，617
jit_inline_above_cost 配置参数，583
jit_optimize_above_cost 配置参数，583
jit_profiling_support 配置参数，617
jit_provider 配置参数，608
jit_tuple_deforming 配置参数，617
jit 配置参数，585
join_collapse_limit 配置参数，585
JSON，176
  函数和操作符，307
JSONB，176
jsonb
  包含，179
  存在，179
  索引，180
jsonb_agg，331
jsonb_array_elements，311
jsonb_array_elements_text，311
jsonb_array_length，311
jsonb_build_array，310
jsonb_build_object，310
jsonb_each，311
jsonb_each_text，311
jsonb_extract_path，311
jsonb_extract_path_text，311
jsonb_insert，311
jsonb_object，310
jsonb_object_agg，331
jsonb_object_keys，311
jsonb_populate_record，311
jsonb_populate_recordset，311
jsonb_pretty，311
jsonb_set，311
jsonb_strip_nulls，311
jsonb_to_record，311
jsonb_to_recordset，311
jsonb_typeof，311
json_agg，331
json_array_elements，311
json_array_elements_text，311
json_array_length，311
json_build_array，310
json_build_object，310
json_each，311
json_each_text，311
json_extract_path，311
json_extract_path_text，311
json_object，310
json_object_agg，331
json_object_keys，311
json_populate_record，311
json_populate_recordset，311
json_strip_nulls，311
json_to_record，311
json_to_recordset，311
json_typeof，311
justify_days，265
justify_hours，265
justify_interval，265

K

krb_caseins_users 配置参数，561
krb_server_keyfile 配置参数，561

L

lag，339
language_handler，209
lastval，317
last_value，339
LATERAL

在 FROM 子句中，120
latitude，2530
lca，2560
lc_collate 配置参数，613
lc_ctype 配置参数，613
lc_messages 配置参数，606
lc_monetary 配置参数，606
lc_numeric 配置参数，606
lc_time 配置参数，606
LDAP，504，631
LDAP 连接参数查找，844
ldconfig，513
lead，339
LEAST，322
结果类型的确定，391
left，224
length，225，237，280，287
length(tsvector)，423
levenshtein，2535
levenshtein_less_equal，2535
lex，499
libedit，498
libperl，499
libpq，781
单行模式，821
libpq-fe.h，781，795
libpq-int.h，795
libpython，499
LIKE，240
与区域设置，652
LIMIT，132
line，165
Linux
IPC 配置，536
共享库，1064
启动脚本，531
LISTEN，1760
listen_addresses 配置参数，558
llvm-config，503
ll_to_earth，2530
ln，216
lo，2555
LOAD，1762
localtime，266
localtimestamp，266
local_preload_libraries 配置参数，607
LOCK，458，1763
lock_timeout 配置参数，603
log，217
logging_collector 配置参数，587
log_autovacuum_min_duration 配置参数，599
log_btree_build_stats 配置参数，616
log_checkpoints 配置参数，591
log_connections 配置参数，591
log_destination 配置参数，586
log_directory 配置参数，587
log_disconnections 配置参数，592
log_duration 配置参数，592
log_error_verbosity 配置参数，592
log_executor_stats 配置参数，598
log_filename 配置参数，587
log_file_mode 配置参数，588
log_hostname 配置参数，592
log_line_prefix 配置参数，593
log_lock_waits 配置参数，594
log_min_duration_statement 配置参数，590
log_min_error_statement 配置参数，590
log_min_messages 配置参数，589
log_parser_stats 配置参数，598
log_planner_stats 配置参数，598
log_replication_commands 配置参数，595
log_rotation_age 配置参数，588
log_rotation_size 配置参数，588
log_statement_stats 配置参数，598
log_statement 配置参数，594
log_temp_files 配置参数，595
log_timezone 配置参数，595
log_truncate_on_rotation 配置参数，588
longitude，2531
looks_like_number
在 PL/Perl 中，1264
lower，221，328
与区域设置，652
lower_inc，328
lower_inf，328
lo_close，864
lo_compat_privileges 配置参数，610
lo_creat，861，865
lo_create，861
lo_export，862，865
lo_from_bytea，865
lo_get，865
lo_import，861，865
lo_import_with_oid，861
lo_lseek，863
lo_lseek64，863
lo_open，862
lo_put，865
lo_read，863
lo_tell，863
lo_tell64，863
lo_truncate，864
lo_truncate64，864
lo_unlink，864，865
lo_write，862
lpad，225
lseg，165，281
LSN，758

ltree，2557
ltree2text，2560
ltrim，225

M

MAC 地址 (见 macaddr)
MAC 地址（EUI-64 格式）(见 macaddr)
macaddr8_set7bit，285
macaddr8（数据类型），169
macaddr（数据类型），168
macOS
  IPC 配置，537
  共享库，1064
  在其上安装，518
maintenance_work_mem 配置参数，565
make，498
make_date，266
make_interval，266
make_time，266
make_timestamp，266
make_timestamptz，267
make_valid，2552
MANPATH，513
masklen，283
max，331
max_connections 配置参数，559
max_files_per_process 配置参数，566
max_function_args 配置参数，613
max_identifier_length 配置参数，613
max_index_keys 配置参数，613
max_locks_per_transaction 配置参数，609
max_logical_replication_workers 配置参数，579
max_parallel_maintenance_workers 配置参数，569
max_parallel_workers_per_gather 配置参数，569
max_parallel_workers 配置参数，569
max_pred_locks_per_page 配置参数，609
max_pred_locks_per_relation 配置参数，609
max_pred_locks_per_transaction 配置参数，609
max_prepared_transactions 配置参数，565
max_replication_slots 配置参数，576
max_stack_depth 配置参数，565
max_standby_archive_delay 配置参数，578
max_standby_streaming_delay 配置参数，578
max_sync_workers_per_subscription 配置参数，579
max_wal_senders 配置参数，576
max_wal_size 配置参数，574
max_worker_processes 配置参数，568
md5，225，238
MD5，627
metaphone，2536
min，331
MinGW
  在其上安装，519
min_parallel_index_scan_size 配置参数，583
min_parallel_table_scan_size 配置参数，583
min_wal_size 配置参数，574
mod，217
MOVE，1766
MultiXactId，670
MVCC，452

N

NaN (见非数)
NetBSD
  IPC 配置，535
  共享库，1064
  启动脚本，531
netmask，283
network，284
nextval，317
NFS (见网络文件系统)
nlevel，2560
normal_rand，2632
NOT IN，340，342
NOTIFY，1768
  在 libpq 中，824
NOTNULL，213
NOT（操作符），211
now，267
npoints，280
nth_value，339
ntile，338
null 值
  libpq 中的，812
NULLIF，321
numeric，40
numeric（数据类型），143
numnode，287，424
num_nonnulls，214
num_nulls，214
NVL，321

O

obj_description，360
octet_length，221，236
OFFSET，132
OID
  libpq 中的，814
  列，73
oid，207
oid2name，2656
old_snapshot_threshold 配置参数，569
ON CONFLICT，1753

ONLY，113
OOM，540
opaque，209
OpenBSD
  IPC 配置，536
  共享库，1065
  启动脚本，531
OpenSSL，504
  (参见 SSL)
operator_precedence_warning 配置参数，611
Oracle
  从 PL/SQL 移植到 PL/pgSQL，1232
ORDER BY，10，131
  与区域设置，652
OR（操作符），211
OVER 子句，50
OVERLAPS，268
overlay，221，236

P

pageinspect，2564
page_checksum，2564
page_header，2564
palloc，1063
PAM，504，635
parallel_leader_participation 配置参数，585
parallel_setup_cost 配置参数，582
parallel_tuple_cost 配置参数，582
parse_ident，225
passwordcheck，2572
password_encryption 配置参数，561
path，281
PATH，513
path（数据类型），166
pclose，280
peer，631
percent_rank，338
  假设行，337
perl，499
Perl，1253
pfree，1063
PGAPPNAME，842
pgbench，1878
PGcancel，822
PGCLIENTENCODING，842
PGconn，781
PGCONNECT_TIMEOUT，842
pgcrypto，2575
PGDATA，528
PGDATABASE，841
PGDATESTYLE，842
PGEventProc，837
PGGEQO，842
PGGSSLIB，842
PGHOST，841
PGHOSTADDR，841
PGKRBSRVNAME，842
PGLOCALEDIR，842
PGOPTIONS，842
PGPASSFILE，842
PGPASSWORD，841
PGPORT，841
pgp_armor_headers，2579
pgp_key_id，2579
pgp_pub_decrypt，2578
pgp_pub_decrypt_bytea，2578
pgp_pub_encrypt，2578
pgp_pub_encrypt_bytea，2578
pgp_sym_decrypt，2578
pgp_sym_decrypt_bytea，2578
pgp_sym_encrypt，2578
pgp_sym_encrypt_bytea，2578
PGREQUIREPEER，842
PGREQUIRESSL，842
PGresult，805
pgrowlocks，2591，2591
PGSERVICE，842
PGSERVICEFILE，842
PGSSLCERT，842
PGSSLCOMPRESSION，842
PGSSLCRL，842
PGSSLKEY，842
PGSSLMODE，842
PGSSLROOTCERT，842
pgstatginindex，2600
pgstathashindex，2600
pgstatindex，2599
pgstattuple，2598，2598
pgstattuple_approx，2601
PGSYSCONFDIR，842
PGTARGETSESSIONATTRS，842
PGTZ，842
PGUSER，841
pgxs，1113
pg_advisory_lock，377
pg_advisory_lock_shared，377
pg_advisory_unlock，377
pg_advisory_unlock_all，377
pg_advisory_unlock_shared，377
pg_advisory_xact_lock，377
pg_advisory_xact_lock_shared，378
pg_aggregate，2043
pg_am，2044
pg_amop，2045
pg_amproc，2046
pg_archivecleanup，1989
pg_attrdef，2046

pg_attribute，2047
pg_authid，2049
pg_auth_members，2050
pg_available_extensions，2096
pg_available_extension_versions，2097
pg_backend_pid，349
pg_backup_start_time，365
pg_basebackup，1871
pg_blocking_pids，350
pg_buffercache，2573
pg_buffercache_pages，2573
pg_cancel_backend，364
pg_cast，2050
pg_class，2051
pg_client_encoding，225
pg_collation，2054
pg_collation_actual_version，373
pg_collation_is_visible，354
pg_column_size，371
pg_config，1896，2097
  与 ecpg 配合使用，935
  用于 libpq，849
  用于用户定义的 C 函数，1063
pg_conf_load_time，350
pg_constraint，2055
pg_controldata，1991
pg_control_checkpoint，362
pg_control_init，362
pg_control_recovery，362
pg_control_system，362
pg_conversion，2057
pg_conversion_is_visible，354
pg_create_logical_replication_slot，369
pg_create_physical_replication_slot，369
pg_create_restore_point，365
pg_ctl，528，530，1992
pg_current_logfile，350
pg_current_wal_flush_lsn，365
pg_current_wal_insert_lsn，365
pg_current_wal_lsn，365
pg_cursors，2097
pg_database，647，2058
pg_database_size，371
pg_db_role_setting，2059
pg_ddl_command，209
pg_default_acl，2059
pg_depend，2060
pg_describe_object，359
pg_description，2061
pg_drop_replication_slot，369
pg_dump，1899
pg_dumpall，1911
  在升级期间使用，544
pg_enum，2062
pg_event_trigger，2062
pg_event_trigger_ddl_commands，378
pg_event_trigger_dropped_objects，379
pg_event_trigger_table_rewrite_oid，380
pg_event_trigger_table_rewrite_reason，380
pg_export_snapshot，368
pg_extension，2063
pg_extension_config_dump，1109
pg_filenode_relation，373
pg_file_rename，2473
pg_file_settings，2098
pg_file_unlink，2473
pg_file_write，2473
pg_foreign_data_wrapper，2063
pg_foreign_server，2064
pg_foreign_table，2064
pg_freespace，2587
pg_freespacemap，2587
pg_function_is_visible，354
pg_get_constraintdef，355
pg_get_expr，355
pg_get_functiondef，355
pg_get_function_arguments，355
pg_get_function_identity_arguments，355
pg_get_function_result，355
pg_get_indexdef，355
pg_get_keywords，355
pg_get_object_address，359
pg_get_ruledef，355
pg_get_serial_sequence，355
pg_get_statisticsobjdef，355
pg_get_triggerdef，355
pg_get_userbyid，355
pg_get_viewdef，355
pg_group，2099
pg_has_role，352
pg_hba.conf，619
pg_hba_file_rules，2099
pg_ident.conf，625
pg_identify_object，359
pg_identify_object_as_address，359
pg_import_system_collations，373
pg_index，2065
pg_indexam_has_property，355
pg_indexes，2100
pg_indexes_size，371
pg_index_column_has_property，355
pg_index_has_property，355
pg_inherits，2067
pg_init_privs，2067
pg_isready，1918
pg_is_in_backup，365
pg_is_in_recovery，367
pg_is_other_temp_schema，350

pg_is_wal_replay_paused，368
pg_language，2068
pg_largeobject，2069
pg_largeobject_metadata，2069
pg_last_committed_xact，362
pg_last_wal_receive_lsn，367
pg_last_wal_replay_lsn，367
pg_last_xact_replay_timestamp，367
pg_listening_channels，350
pg_locks，2100
pg_logdir_ls，2473
pg_logical_emit_message，371
pg_logical_slot_get_binary_changes，370
pg_logical_slot_get_changes，370
pg_logical_slot_peek_binary_changes，370
pg_logical_slot_peek_changes，370
pg_lsn，208
pg_ls_dir，375
pg_ls_logdir，375
pg_ls_waldir，375
pg_matviews，2103
pg_my_temp_schema，350
pg_namespace，2069
pg_notification_queue_usage，350
pg_notify，1769
pg_opclass，2070
pg_opclass_is_visible，354
pg_operator，2070
pg_operator_is_visible，354
pg_opfamily，2071
pg_opfamily_is_visible，354
pg_options_to_table，355
pg_partitioned_table，2071
pg_pltemplate，2072
pg_policies，2103
pg_policy，2073
pg_postmaster_start_time，351
pg_prepared_statements，2104
pg_prepared_xacts，2104
pg_prewarm，2589
pg_prewarm.autoprewarm 配置参数，2589
pg_prewarm.autoprewarm_interval 配置参数，2589
pg_proc，2074
pg_publication，2077
pg_publication_rel，2077
pg_publication_tables，2105
pg_range，2077
pg_read_binary_file，376
pg_read_file，375
pg_receivewal，1920
pg_receivexlog，2682(见 pg_receivewal)
pg_recvlogical，1924
pg_relation_filenode，373
pg_relation_filepath，373
pg_relation_size，371
pg_reload_conf，364
pg_relpages，2601
pg_replication_origin，2078
pg_replication_origin_advance，371
pg_replication_origin_create，370
pg_replication_origin_drop，370
pg_replication_origin_oid，370
pg_replication_origin_progress，371
pg_replication_origin_session_is_setup，370
pg_replication_origin_session_progress，371
pg_replication_origin_session_reset，370
pg_replication_origin_session_setup，370
pg_replication_origin_status，2105
pg_replication_origin_xact_reset，371
pg_replication_origin_xact_setup，371
pg_replication_slots，2105
pg_replication_slot_advance，370
pg_resetwal，1998
pg_resetxlog，2682(见 pg_resetwal)
pg_restore，1928
pg_rewind，2002
pg_rewrite，2078
pg_roles，2106
pg_rotate_logfile，364
pg_rules，2107
pg_safe_snapshot_blocking_pids，351
pg_seclabel，2079
pg_seclabels，2107
pg_sequence，2079
pg_sequences，2108
pg_service.conf，843
pg_settings，2109
pg_shadow，2110
pg_shdepend，2080
pg_shdescription，2081
pg_shseclabel，2081
pg_size_bytes，371
pg_size_pretty，371
pg_sleep，277
pg_sleep_for，277
pg_sleep_until，277
pg_standby，2663
pg_start_backup，365
pg_statio_all_indexes，717
pg_statio_all_sequences，717
pg_statio_all_tables，717
pg_statio_sys_indexes，717
pg_statio_sys_sequences，717
pg_statio_sys_tables，717
pg_statio_user_indexes，717
pg_statio_user_sequences，717
pg_statio_user_tables，717

pg_statistic，477，2082
pg_statistics_obj_is_visible，354
pg_statistic_ext，478，2083
pg_stats，478，2111
pg_stat_activity，716
pg_stat_all_indexes，717
pg_stat_all_tables，716
pg_stat_archiver，716
pg_stat_bgwriter，716
pg_stat_clear_snapshot，738
pg_stat_database，716
pg_stat_database_conflicts，716
pg_stat_file，376
pg_stat_get_activity，738
pg_stat_get_snapshot_timestamp，738
pg_stat_get_xact_blocks_fetched，738
pg_stat_get_xact_blocks_hit，738
pg_stat_progress_vacuum，716
pg_stat_replication，716
pg_stat_reset，738
pg_stat_reset_shared，739
pg_stat_reset_single_function_counters，739
pg_stat_reset_single_table_counters，739
pg_stat_ssl，716
pg_stat_statements，2593
  函数，2595
pg_stat_statements_reset，2595
pg_stat_subscription，716
pg_stat_sys_indexes，717
pg_stat_sys_tables，716
pg_stat_user_functions，717
pg_stat_user_indexes，717
pg_stat_user_tables，717
pg_stat_wal_receiver，716
pg_stat_xact_all_tables，717
pg_stat_xact_sys_tables，717
pg_stat_xact_user_functions，717
pg_stat_xact_user_tables，717
pg_stop_backup，365
pg_subscription，2084
pg_subscription_rel，2084
pg_switch_wal，365
pg_tables，2113
pg_tablespace，2085
pg_tablespace_databases，355
pg_tablespace_location，355
pg_tablespace_size，371
pg_table_is_visible，354
pg_table_size，371
pg_temp，601
  用于保护函数，1550
pg_terminate_backend，364
pg_test_fsync，2005
pg_test_timing，2006
pg_timezone_abbrevs，2113
pg_timezone_names，2114
pg_total_relation_size，371
pg_transform，2085
pg_trgm，2603
pg_trgm.similarity_threshold 配置参数，2605
pg_trgm.strict_word_similarity_threshold 配置参数，2605
pg_trgm.word_similarity_threshold 配置参数，2605
pg_trigger，2086
pg_try_advisory_lock，377
pg_try_advisory_lock_shared，377
pg_try_advisory_xact_lock，378
pg_try_advisory_xact_lock_shared，378
pg_ts_config，2087
pg_ts_config_is_visible，354
pg_ts_config_map，2088
pg_ts_dict，2088
pg_ts_dict_is_visible，354
pg_ts_parser，2089
pg_ts_parser_is_visible，354
pg_ts_template，2089
pg_ts_template_is_visible，354
pg_type，2090
pg_typeof，355
pg_type_is_visible，354
pg_upgrade，2010
pg_user，2114
pg_user_mapping，2095
pg_user_mappings，2114
pg_verify_checksums，2018
pg_views，2115
pg_visibility，2609
pg_waldump，2019
pg_walfile_name，365
pg_walfile_name_offset，365
pg_wal_lsn_diff，365
pg_wal_replay_pause，368
pg_wal_replay_resume，368
pg_xact_commit_timestamp，362
pg_xlogdump，2682(见 pg_waldump)
phraseto_tsquery，288，418
pi，217
PIC，1063
PID
  确定服务器进程的 PID
    在 libpq 中，799
PITR，675
PITR 备库，691
pkg-config，504
  与 ecpg 配合使用，935
  用于 libpq，849
PL/Perl，1253

PL/PerlU，1266
PL/pgSQL，1172
PL/Python，1271
PL/SQL (Oracle)
  移植到 PL/pgSQL，1232
PL/Tcl，1242
plainto_tsquery，288，417
plperl.on_init 配置参数，1269
plperl.on_plperlu_init 配置参数，1269
plperl.on_plperl_init 配置参数，1269
plperl.use_strict 配置参数，1269
plpgsql.check_asserts 配置参数，1216
plpgsql.variable_conflict 配置参数，1226
pltcl.start_proc 配置参数，1252
pltclu.start_proc 配置参数，1252
point，165，281
polygon，166，281
popen，280
populate_record，2541
port 配置参数，559
position，221，236
POSTGRES，xxxiv
postgres，3，530，646，2022
postgres 用户，528
Postgres 95，xxxiv
postgresql.auto.conf，555
postgresql.conf，554
postgres_fdw，2611
postmaster，2029
post_auth_delay 配置参数，614
power，217
PQbackendPID，799
PQbinaryTuples，811
  用于 COPY，825
PQcancel，822
PQclear，809
PQclientEncoding，829
PQcmdStatus，813
PQcmdTuples，813
PQconndefaults，785
PQconnectdb，782
PQconnectdbParams，781
PQconnectionNeedsPassword，799
PQconnectionUsedPassword，799
PQconnectPoll，783
PQconnectStart，783
PQconnectStartParams，783
PQconninfo，786
PQconninfoFree，831
PQconninfoParse，786
PQconsumeInput，820
PQcopyResult，832
PQdb，795
PQdescribePortal，804
PQdescribePrepared，804
PQencryptPassword，832
PQencryptPasswordConn，831
PQendcopy，829
PQerrorMessage，798
PQescapeBytea，816
PQescapeByteaConn，816
PQescapeIdentifier，814
PQescapeLiteral，814
PQescapeString，815
PQescapeStringConn，815
PQexec，801
PQexecParams，801
PQexecPrepared，804
PQfformat，810
  用于 COPY，825
PQfinish，786
PQfireResultCreateEvents，832
PQflush，821
PQfmod，811
PQfn，823
PQfname，809
PQfnumber，809
PQfreeCancel，822
PQfreemem，831
PQfsize，811
PQftable，810
PQftablecol，810
PQftype，811
PQgetCancel，822
PQgetCopyData，827
PQgetisnull，812
PQgetlength，812
PQgetline，827
PQgetlineAsync，828
PQgetResult，819
PQgetssl，800
PQgetvalue，811
PQhost，795
PQinitOpenSSL，848
PQinitSSL，848
PQinstanceData，838
PQisBusy，820
PQisnonblocking，821
PQisthreadsafe，848
PQlibVersion，833
  (参见 PQserverVersion)
PQmakeEmptyPGresult，832
PQnfields，809
  用于 COPY，825
PQnotifies，824
PQnparams，812
PQntuples，809
PQoidStatus，814

PQoidValue，814
PQoptions，796
PQparameterStatus，797
PQparamtype，812
PQpass，795
PQping，788
PQpingParams，787
PQport，796
PQprepare，803
PQprint，813
PQprotocolVersion，798
PQputCopyData，826
PQputCopyEnd，826
PQputline，828
PQputnbytes，828
PQregisterEventProc，838
PQrequestCancel，823
PQreset，786
PQresetPoll，787
PQresetStart，787
PQresStatus，806
PQresultAlloc，833
PQresultErrorField，807
PQresultErrorMessage，806
PQresultInstanceData，839
PQresultSetInstanceData，838
PQresultStatus，805
PQresultVerboseErrorMessage，806
PQsendDescribePortal，819
PQsendDescribePrepared，819
PQsendPrepare，818
PQsendQuery，817
PQsendQueryParams，818
PQsendQueryPrepared，818
PQserverVersion，798
PQsetClientEncoding，829
PQsetdb，783
PQsetdbLogin，782
PQsetErrorContextVisibility，830
PQsetErrorVerbosity，830
PQsetInstanceData，838
PQsetnonblocking，820
PQsetNoticeProcessor，834
PQsetNoticeReceiver，834
PQsetResultAttrs，833
PQsetSingleRowMode，821
PQsetvalue，833
PQsocket，798
PQsslAttribute，799
PQsslAttributeNames，800
PQsslInUse，799
PQsslStruct，800
PQstatus，796
PQtrace，830
PQtransactionStatus，797
PQtty，796
PQunescapeBytea，816
PQuntrace，831
PQuser，795
PREPARE，1770
PREPARE TRANSACTION，1773
pre_auth_delay 配置参数，614
primary_conninfo 恢复参数，712
primary_slot_name 恢复参数，712
ps
  监控活动，714
psql，5，1937
Python，1271

Q

querytree，288，424
quote_all_identifiers 配置参数，611
quote_ident，226
  在 PL/Perl 中，1263
  在 PL/pgSQL 中使用，1186
quote_literal，226
  在 PL/Perl 中，1263
  在 PL/pgSQL 中使用，1186
quote_nullable，226
  在 PL/Perl 中，1263
  在 PL/pgSQL 中使用，1186

R

radians，217
radius，280
RADIUS，634
RAISE
  在 PL/pgSQL 中，1214
random，219
random_page_cost 配置参数，582
rank，338
  假设行，337
readline，498
real，144
REASSIGN OWNED，1775
record，209
recovery.conf，710
recovery_end_command 恢复参数，710
recovery_min_apply_delay 恢复参数，713
recovery_target 恢复参数，711
recovery_target_action 恢复参数，712
recovery_target_inclusive 恢复参数，711
recovery_target_lsn 恢复参数，711
recovery_target_name 恢复参数，711
recovery_target_time 恢复参数，711
recovery_target_timeline 恢复参数，711
recovery_target_xid 恢复参数，711

RECURSIVE
  在公共表表达式中，135
  在视图中，1660
REFRESH MATERIALIZED VIEW，1776
regclass，207
regconfig，207
regdictionary，207
regexp_match，226，242
regexp_matches，227，242
regexp_replace，227，242
regexp_split_to_array，227，242
regexp_split_to_table，227，242
regoper，207
regoperator，207
regproc，207
regprocedure，207
regr_avgx，334
regr_avgy，334
regr_count，334
regr_intercept，334
regr_r2，334
regr_slope，334
regr_sxx，334
regr_sxy，335
regr_syy，335
regtype，207
REINDEX，1778
reindexdb，1977
RELEASE SAVEPOINT，1781
repeat，227
replace，228
RESET，1783
restart_after_crash 配置参数，612
restore_command 恢复参数，710
RESTRICT
  用于 DROP，106
  外键动作，72
RETURN NEXT
  在 PL/pgSQL 中，1191
RETURN QUERY
  在 PL/pgSQL 中，1191
RETURNING，110
RETURNING INTO
  在 PL/pgSQL 中，1183
reverse，228
REVOKE，77，1784
right，228
ROLLBACK，1788
rollback
  psql，1967
ROLLBACK PREPARED，1789
ROLLBACK TO SAVEPOINT，1790
ROLLUP，125
round，218
ROW，56
row_number，338
row_security 配置参数，601
row_security_active，352
row_to_json，310
rpad，228
rtrim，228

S

SAVEPOINT，1792
scale，218
SCRAM，627
search_path 配置参数
  用于保护函数，1550
search_path 配置参数，86，600
SECURITY LABEL，1794
sec_to_gc，2530
seg，2617
segment_size 配置参数，613
SELECT，9，112，1797
  结果类型的确定，393
  选择列表，128
SELECT INTO，1818
  在 PL/pgSQL 中，1183
sepgsql，2620
sepgsql.debug_audit 配置参数，2622
sepgsql.permissive 配置参数，2622
seq_page_cost 配置参数，582
serial，146
serial2，146
serial4，146
serial8，146
server_encoding 配置参数，613
server_version_num 配置参数，613
server_version 配置参数，613
session_preload_libraries 配置参数，607
session_replication_role 配置参数，603
session_user，349
SET，364，1820
SET CONSTRAINTS，1823
SET ROLE，1825
SET SESSION AUTHORIZATION，1827
SET TRANSACTION，1829
SET XML OPTION，605
setseed，219
setval，317
setweight，288，423
  为指定词位设置权重，288
set_bit，238
set_byte，238
set_config，364
set_limit，2603
set_masklen，284

sha224，238
sha256，238
sha384，238
sha512，239
shared_buffers 配置参数，564
shared_preload_libraries，1076
shared_preload_libraries 配置参数，607
shobj_description，360
SHOW，364，1832，2131
show_limit，2603
show_trgm，2603
SIGHUP，555，623，626
SIGINT，543
sign，218
SIGQUIT，543
SIGTERM，542
SIMILAR TO，241
similarity，2603
sin，220
sind，220
skeys，2540
slice，2541
smallint，143
smallserial，146
Solaris
  IPC 配置，538
  共享库，1065
  启动脚本，531
  在其上安装，520
SOME，332，340，342
sort，2548
sort_asc，2548
sort_desc，2548
soundex，2535
SP-GiST (见索引)
SPI，1290
  示例，2628
spi_commit
  在 PL/Perl 中，1262
SPI_commit，1345
SPI_connect，1290
SPI_connect_ext，1290
SPI_copytuple，1338
spi_cursor_close
  在 PL/Perl 中，1260
SPI_cursor_close，1320
SPI_cursor_fetch，1316
SPI_cursor_find，1315
SPI_cursor_move，1317
SPI_cursor_open，1310
SPI_cursor_open_with_args，1312
SPI_cursor_open_with_paramlist，1314
SPI_exec，1296
SPI_execp，1309
SPI_execute，1293
SPI_execute_plan，1306
SPI_execute_plan_with_paramlist，1308
SPI_execute_with_args，1297
spi_exec_prepared
  在 PL/Perl 中，1261
spi_exec_query
  在 PL/Perl 中，1258
spi_fetchrow
  在 PL/Perl 中，1260
SPI_finish，1292
SPI_fname，1326
SPI_fnumber，1327
spi_freeplan
  在 PL/Perl 中，1261
SPI_freeplan，1344
SPI_freetuple，1342
SPI_freetuptable，1343
SPI_getargcount，1303
SPI_getargtypeid，1304
SPI_getbinval，1329
SPI_getnspname，1333
SPI_getrelname，1332
SPI_gettype，1330
SPI_gettypeid，1331
SPI_getvalue，1328
SPI_is_cursor_plan，1305
SPI_keepplan，1321
spi_lastoid
  在 PL/Tcl 中，1247
SPI_modifytuple，1340
SPI_palloc，1335
SPI_pfree，1337
spi_prepare
  在 PL/Perl 中，1261
SPI_prepare，1299
SPI_prepare_cursor，1301
SPI_prepare_params，1302
spi_query
  在 PL/Perl 中，1260
spi_query_prepared
  在 PL/Perl 中，1261
SPI_register_relation，1323
SPI_register_trigger_data，1325
SPI_repalloc，1336
SPI_result_code_string，1334
SPI_returntuple，1339
spi_rollback
  在 PL/Perl 中，1262
SPI_rollback，1346
SPI_saveplan，1322
SPI_scroll_cursor_fetch，1318
SPI_scroll_cursor_move，1319
SPI_start_transaction，1347

SPI_unregister_relation，1324
split_part，228
SQL/CLI，2342
SQL/Foundation，2342
SQL/Framework，2342
SQL/JRT，2342
SQL/MED，2342
SQL/OLB，2342
SQL/PSM，2342
SQL/Schemata，2342
SQL/XML，2342
  限制和符合性，2365
sqrt，218
ssh，551
SSI，452
SSL，547，845
  在 libpq 中，793
  在 libpq 中，800
sslinfo，2630
ssl_ca_file 配置参数，562
ssl_cert_file 配置参数，562
ssl_cipher，2630
ssl_ciphers 配置参数，562
ssl_client_cert_present，2630
ssl_client_dn，2630
ssl_client_dn_field，2631
ssl_client_serial，2630
ssl_crl_file 配置参数，562
ssl_dh_params_file 配置参数，563
ssl_ecdh_curve 配置参数，563
ssl_extension_info，2631
ssl_issuer_dn，2630
ssl_issuer_field，2631
ssl_is_used，2630
ssl_key_file 配置参数，562
ssl_passphrase_command_supports_reload 配置参数，564
ssl_passphrase_command 配置参数，563
ssl_prefer_server_ciphers 配置参数，563
ssl_version，2630
ssl 配置参数，562
SSPI，629
STABLE，1053
standard_conforming_strings 配置参数，611
standby_mode 恢复参数，712
START TRANSACTION，1834
starts_with，229
START_REPLICATION，2132
statement_timeout 配置参数，603
statement_timestamp，267
stats_temp_directory 配置参数，598
stddev，335
stddev_pop，335
stddev_samp，335
STONITH，691
strict_word_similarity，2603
string_agg，332
string_to_array，323
strip，288，423
strpos，228
subarray，2548
subltree，2559
subpath，2560
substr，229
substring，221，236，241，242
sum，332
superuser_reserved_connections 配置参数，559
suppress_redundant_updates_trigger，378
svals，2540
synchronize_seqscans 配置参数，611
synchronous_commit 配置参数，571
synchronous_standby_names 配置参数，576
syslog_facility 配置参数，589
syslog_ident 配置参数，589
syslog_sequence_numbers 配置参数，589
syslog_split_messages 配置参数，589
systemd，504，531
  RemoveIPC，538

T

TABLE 命令，1797
tablefunc，2632
tableoid，73
TABLESAMPLE 方法，2203
tan，220
tand，220
Tcl，1242
tcn，2642
tcp_keepalives_count 配置参数，561
tcp_keepalives_idle 配置参数，560
tcp_keepalives_interval 配置参数，560
template0，647
template1，646，646
temp_buffers 配置参数，565
temp_file_limit 配置参数，566
temp_tablespaces 配置参数，601
test_decoding，2643
text，147，284
text2ltree，2560
tid，207
time，151，153
  常量，155
  输出格式，156
    (参见格式化)
time with time zone，151，153
time without time zone，151，153
TIMELINE_HISTORY，2131

timeofday，267
timestamp，151，155
timestamp with time zone，151，155
timestamp without time zone，151，155
timestamptz，151
timezone_abbreviations 配置参数，605
TimeZone 配置参数，605
TOAST，2272
  与大对象对比，860
  与用户定义类型，1088
  每列存储设置，1451
to_ascii，229
to_char，255
  与区域设置，652
to_date，255
to_hex，229
to_json，310
to_jsonb，310
to_number，255
to_regclass，355
to_regnamespace，355
to_regoper，355
to_regoperator，355
to_regproc，355
to_regprocedure，355
to_regrole，355
to_regtype，355
to_timestamp，255，267
to_tsquery，289，416
to_tsvector，289，415
trace_locks 配置参数，615
trace_lock_oidmin 配置参数，616
trace_lock_table 配置参数，616
trace_lwlocks 配置参数，615
trace_notify 配置参数，615
trace_recovery_messages 配置参数，615
trace_sort 配置参数，615
trace_userlocks 配置参数，615
track_activities 配置参数，597
track_activity_query_size 配置参数，597
track_commit_timestamp 配置参数，576
track_counts 配置参数，597
track_functions 配置参数，598
track_io_timing 配置参数，598
transaction_deferrable 配置参数，603
transaction_isolation 配置参数，602
transaction_read_only 配置参数，602
transaction_timestamp，267
transform_null_equals 配置参数，611
translate，229
trigger，209
triggered_change_notification，2642
trigger_file 恢复参数，712
trim，222，236
true，161
trunc，218，285，285
TRUNCATE，1835
tsm_handler，209
tsm_system_rows，2644
tsm_system_time，2645
tsquery_phrase，291，424
tsquery（数据类型），172
tsvector 连接，423
tsvector_to_array，291
tsvector_update_trigger，291
tsvector_update_trigger_column，291
tsvector（数据类型），171
ts_debug，292，442
ts_delete，289
ts_filter，290
ts_headline，290，421
ts_lexize，292，445
ts_parse，292，444
ts_rank，290，419
ts_rank_cd，290，420
ts_rewrite，290，425
ts_stat，293，428
ts_token_type，293，444
tuple_data_split，2565
txid_current，360
txid_current_if_assigned，360
txid_current_snapshot，360
txid_snapshot_xip，360
txid_snapshot_xmax，360
txid_snapshot_xmin，360
txid_status，360
txid_visible_in_snapshot，360

U

UESCAPE，36，39
unaccent，2646，2647
Unicode 转义
  在字符串常量中，38
  在标识符中，36
UNION，130
  结果类型的确定，391
uniq，2548
Unix 域套接字，790
unix_socket_directories 配置参数，559
unix_socket_group 配置参数，559
unix_socket_permissions 配置参数，560
unknown，209
UNLISTEN，1837
unnest，323
  用于 tsvector，291
UPDATE，16，109，1839
  RETURNING，110

update_process_title 配置参数，597
upper，222，328
  与区域设置，652
upper_inc，328
upper_inf，328
UPSERT，1753
URI，788
user，349
UUID，173，504
uuid-ossp，2649
uuid_generate_v1，2649
uuid_generate_v1mc，2649
uuid_generate_v3，2649

V

VACUUM，1844
vacuumdb，1980
vacuumlo，2661
vacuum_cleanup_index_scale_factor 配置参数，604
vacuum_cost_delay 配置参数，567
vacuum_cost_limit 配置参数，567
vacuum_cost_page_dirty 配置参数，567
vacuum_cost_page_hit 配置参数，567
vacuum_cost_page_miss 配置参数，567
vacuum_defer_cleanup_age 配置参数，578
vacuum_freeze_min_age 配置参数，604
vacuum_freeze_table_age 配置参数，603
vacuum_multixact_freeze_min_age 配置参数，604
vacuum_multixact_freeze_table_age 配置参数，604
VALUES，133，1847
  结果类型的确定，391
varchar，147
variance，335
var_pop，335
var_samp，336
version，6，351
VM (见可见性映射)
void，209
VOLATILE，1053
VPATH，500，1116

W

WAL，753
wal_block_size 配置参数，614
wal_buffers 配置参数，573
wal_compression 配置参数，573
wal_consistency_checking 配置参数，616
wal_debug 配置参数，616
wal_keep_segments 配置参数，576
wal_level 配置参数，570
wal_log_hints 配置参数，572
wal_receiver_status_interval 配置参数，578
wal_receiver_timeout 配置参数，579
wal_retrieve_retry_interval 配置参数，579
wal_segment_size 配置参数，614
wal_sender_timeout 配置参数，576
wal_sync_method 配置参数，572
wal_writer_delay 配置参数，573
wal_writer_flush_after 配置参数，573
websearch_to_tsquery，288
WHERE，121
WHILE
  在 PL/pgSQL 中，1198
width，280
width_bucket，219
WITH
  在 SELECT 中，134，1797
WITH CHECK OPTION，1660
WITHIN GROUP，47
word_similarity，2603
work_mem 配置参数，565

X

xid，207
xmax，73
xmin，73
XML，174
XML 函数，293
XML 导出，303
XML 选项，175，605
xml2，2651
xmlagg，297，332
xmlbinary 配置参数，604
xmlcomment，293
xmlconcat，294
xmlelement，294
XMLEXISTS，298
xmlforest，296
xmloption 配置参数，605
xmlparse，175
xmlpi，296
xmlroot，297
xmlserialize，175
xmltable，300
xml_is_well_formed，298
xml_is_well_formed_content，298
xml_is_well_formed_document，298
XPath，299
xpath_exists，300
xpath_table，2652
xslt_process，2654

Y

yacc，499

Z

zero_damaged_pages 配置参数，616
zlib，498，506