PostgreSQL 7.4.30 手册

PostgreSQL 全球开发组

翻译：冯若航[1]，Pigsty 项目组[2]

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

PostgreSQL 7.4.30 手册

PostgreSQL 全球开发组
翻译：冯若航[1]，Pigsty 项目组[2]
版权 © 1996-2003 PostgreSQL 全球开发组

法律声明

PostgreSQL 版权所有 © 1996-2002，归 PostgreSQL 全球开发组所有，并按照下文所述加利福尼亚大学的许可条款分发。

Postgres95 版权所有 © 1994-5，归加利福尼亚大学董事会所有。

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

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

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

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

目录

前言 xxii
1. 什么是PostgreSQL? xxii
2. PostgreSQL简史 xxiii
2.1. 伯克利 POSTGRES 项目 xxiii
2.2. Postgres95 xxiii
2.3. PostgreSQL xxiv
3. 约定 xxiv
4. 更多信息 xxiv
5. 缺陷报告指南 xxv
5.1. 识别缺陷 xxv
5.2. 报告哪些内容 xxvi
5.3. 到哪里报告缺陷 xxvii
I. 教程 1
1. 从头开始 3
1.1. 安装 3
1.2. 架构基础 3
1.3. 创建数据库 3
1.4. 访问数据库 5
2. SQL语言 7
2.1. 简介 7
2.2. 概念 7
2.3. 创建新表 7
2.4. 向表中填充行 8
2.5. 查询表 9
2.6. 表之间的连接 10
2.7. 聚合函数 12
2.8. 更新 13
2.9. 删除 13
3. 高级特性 15
3.1. 简介 15
3.2. 视图 15
3.3. 外键 15
3.4. 事务 16
3.5. 继承 17
3.6. 小结 18
II. SQL 语言 19
4. SQL 语法 24
4.1. 词法结构 24
4.2. 值表达式 29
5. 数据定义 36
5.1. 表基础 36
5.2. 系统列 37
5.3. 默认值 38
5.4. 约束 38
5.5. 继承 44
5.6. 修改表 46
5.7. 权限 48
5.8. 模式 49
5.9. 其他数据库对象 52
5.10. 依赖跟踪 53
6. 数据操纵 54

6.1. 插入数据 54
6.2. 更新数据 55
6.3. 删除数据 55
7. 查询 57
7.1. 概述 57
7.2. 表表达式 57
7.3. 选择列表 66
7.4. 组合查询 67
7.5. 行排序 68
7.6. LIMIT 和 OFFSET 69
8. 数据类型 70
8.1. 数字类型 71
8.2. 货币类型 74
8.3. 字符类型 74
8.4. 二进制数据类型 76
8.5. 日期/时间类型 77
8.6. 布尔类型 83
8.7. 几何类型 84
8.8. 网络地址类型 86
8.9. 位串类型 88
8.10. 数组 88
8.11. 对象标识符类型 95
8.12. 伪类型 96
9. 函数和操作符 98
9.1. 逻辑操作符 98
9.2. 比较操作符 98
9.3. 数学函数和操作符 100
9.4. 字符串函数和操作符 104
9.5. 二进制串函数和操作符 113
9.6. 模式匹配 115
9.7. 数据类型格式化函数 125
9.8. 日期/时间函数和操作符 130
9.9. 几何函数和操作符 139
9.10. 网络地址类型函数 142
9.11. 序列操作函数 144
9.12. 条件表达式 145
9.13. 杂项函数 147
9.14. 数组函数和操作符 152
9.15. 聚合函数 153
9.16. 子查询表达式 155
9.17. 行和数组比较 157
10. 类型转换 160
10.1. 概述 160
10.2. 操作符 161
10.3. 函数 164
10.4. 值存储 166
10.5. UNION、CASE和ARRAY结构 166
11. 索引 168
11.1. 简介 168
11.2. 索引类型 168
11.3. 多列索引 169
11.4. 唯一索引 170
11.5. 表达式索引 170

11.6. 操作符类 171
11.7. 部分索引 172
11.8. 检查索引使用情况 174
12. 并发控制 176
12.1. 简介 176
12.2. 事务隔离 176
12.3. 显式锁定 178
12.4. 应用级别的数据一致性检查 181
12.5. 锁与索引 181
13. 性能提示 183
13.1. 使用EXPLAIN 183
13.2. 规划器使用的统计信息 186
13.3. 用显式JOIN子句控制规划器 188
13.4. 填充一个数据库 189
III. 服务器管理 191
14. 安装说明 195
14.1. 简短版本 195
14.2. 需求 195
14.3. 获取源代码 197
14.4. 如果你要升级 197
14.5. 安装过程 198
14.6. 安装后设置 203
14.7. 受支持的平台 205
15. 在 Windows 上安装 209
16. 服务器运行时环境 210
16.1. PostgreSQL 用户账户 210
16.2. 创建一个数据库集簇 210
16.3. 启动数据库服务器 211
16.4. 运行时配置 213
16.5. 管理内核资源 230
16.6. 关闭服务器 235
16.7. 使用 SSL 的安全 TCP/IP 连接 236
16.8. 使用 SSH 隧道的安全 TCP/IP 连接 236
17. 数据库用户和权限 238
17.1. 数据库用户 238
17.2. 用户属性 238
17.3. 组 239
17.4. 权限 239
17.5. 函数和触发器 240
18. 管理数据库 241
18.1. 概述 241
18.2. 创建数据库 241
18.3. 模板数据库 242
18.4. 数据库配置 243
18.5. 备选位置 243
18.6. 删除数据库 244
19. 客户端认证 246
19.1. pg_hba.conf 文件 246
19.2. 认证方法 251
19.3. 认证问题 254
20. 本地化 255
20.1. 区域支持 255
20.2. 字符集支持 257

21. 日常数据库维护任务 262
21.1. 例行清理 262
21.2. 例行重索引 265
21.3. 日志文件维护 265
22. 备份与恢复 266
22.1. SQL 转储 266
22.2. 文件系统级别的备份 268
22.3. 版本间的迁移 269
23. 监控数据库活动 271
23.1. 标准 Unix 工具 271
23.2. 统计收集器 272
23.3. 查看锁 275
24. 监控磁盘使用情况 276
24.1. 确定磁盘使用情况 276
24.2. 磁盘已满故障 277
25. 预写式日志（WAL） 278
25.1. WAL 的优势 278
25.2. 未来的收益 278
25.3. WAL 配置 279
25.4. 内部机制 280
26. 回归测试 281
26.1. 运行测试 281
26.2. 测试结果评估 282
26.3. 平台相关的比较文件 284
IV. 客户端接口 285
27. libpq - C 库 289
27.1. 数据库连接控制函数 289
27.2. 连接状态函数 294
27.3. 命令执行函数 297
27.4. 异步命令处理 307
27.5. 快速路径接口 310
27.6. 异步通知 311
27.7. 与`COPY`命令相关的函数 312
27.8. 控制函数 316
27.9. 通知处理 316
27.10. 环境变量 317
27.11. 密码文件 318
27.12. 在线程化程序中的行为 318
27.13. 构建libpq程序 319
27.14. 示例程序 320
28. 大对象 330
28.1. 历史 330
28.2. 实现特性 330
28.3. 客户端接口 330
28.4. 服务器端函数 332
28.5. 示例程序 333
29. pgtcl - Tcl 绑定库 338
29.1. 概述 338
29.2. 将 pgtcl 载入应用程序 338
29.3. pgtcl 命令参考 339
29.4. 示例程序 361
30. ECPG - C 中的嵌入式 SQL 363
30.1. 概念 363

30.2. 连接到数据库服务器 363
30.3. 关闭一个连接 364
30.4. 运行 SQL 命令 365
30.5. 选择一个连接 365
30.6. 使用主变量 366
30.7. 动态 SQL 368
30.8. 使用描述符区域 369
30.9. 错误处理 371
30.10. 包含文件 375
30.11. 处理嵌入式 SQL 程序 376
30.12. 库函数 377
30.13. 内部 377
31. JDBC 接口 380
31.1. 设置 JDBC 驱动 380
31.2. 初始化驱动 381
31.3. 发出查询并处理结果 382
31.4. 执行更新 384
31.5. 调用存储函数 384
31.6. 创建和修改数据库对象 386
31.7. 存储二进制数据 386
31.8. PostgreSQL 对 JDBC API 的扩展 389
31.9. 在多线程或 Servlet 环境中使用驱动 409
31.10. 连接池和数据源 409
31.11. 延伸阅读 413
32. 信息模式 414
32.1. 模式 414
32.2. 数据类型 414
32.3. information_schema_catalog_name 414
32.4. applicable_roles 415
32.5. check_constraints 415
32.6. column_domain_usage 415
32.7. column_privileges 416
32.8. column_udt_usage 416
32.9. columns 417
32.10. constraint_column_usage 419
32.11. constraint_table_usage 419
32.12. data_type_privileges 420
32.13. domain_constraints 420
32.14. domain_udt_usage 421
32.15. domains 421
32.16. element_types 423
32.17. enabled_roles 425
32.18. key_column_usage 425
32.19. parameters 425
32.20. referential_constraints 427
32.21. role_column_grants 427
32.22. role_routine_grants 428
32.23. role_table_grants 428
32.24. role_usage_grants 429
32.25. routine_privileges 429
32.26. routines 430
32.27. schemata 432
32.28. sql_features 433

32.29. sql_implementation_info 433
32.30. sql_languages 434
32.31. sql_packages 434
32.32. sql_sizing 434
32.33. sql_sizing_profiles 435
32.34. table_constraints 435
32.35. table_privileges 436
32.36. tables 436
32.37. triggers 437
32.38. usage_privileges 438
32.39. view_column_usage 438
32.40. view_table_usage 439
32.41. views 439
V. 服务器编程 440
33. 扩展 SQL 444
33.1. 可扩展性如何运作 444
33.2. PostgreSQL 类型系统 444
33.3. 用户定义的函数 445
33.4. 查询语言（SQL）函数 446
33.5. 过程语言函数 452
33.6. 内部函数 452
33.7. C 语言函数 452
33.8. 函数重载 472
33.9. 用户自定义聚合 472
33.10. 用户定义的类型 474
33.11. 用户自定义操作符 477
33.12. 操作符优化信息 477
33.13. 索引扩展接口 481
34. 规则系统 488
34.1. 查询树 488
34.2. 视图和规则系统 489
34.3. INSERT、UPDATE 和 DELETE 上的规则 496
34.4. 规则和权限 505
34.5. 规则和命令状态 506
34.6. 规则与触发器 507
35. 触发器 510
35.1. 触发器行为概述 510
35.2. 数据更改的可见性 510
35.3. 用 C 编写触发器函数 511
35.4. 一个完整的示例 513
36. 过程语言 517
36.1. 安装过程语言 517
37. PL/pgSQL - SQL 过程语言 519
37.1. 概述 519
37.2. PL/pgSQL 开发技巧 520
37.3. PL/pgSQL 的结构 522
37.4. 声明 523
37.5. 表达式 526
37.6. 基本语句 527
37.7. 控制结构 530
37.8. 游标 536
37.9. 错误和消息 539
37.10. 触发器过程 539

37.11. 从 Oracle PL/SQL 移植 541
38. PL/Tcl - Tcl 过程语言 551
38.1. 概述 551
38.2. PL/Tcl 函数和参数 551
38.3. PL/Tcl 中的数据值 552
38.4. PL/Tcl 中的全局数据 552
38.5. 从 PL/Tcl 访问数据库 553
38.6. PL/Tcl 中的触发器过程 555
38.7. 模块和 unknown 命令 556
38.8. Tcl 过程名 557
39. PL/Perl - Perl 过程语言 558
39.1. PL/Perl 函数和参数 558
39.2. PL/Perl 中的数据值 559
39.3. 从 PL/Perl 访问数据库 559
39.4. 受信任与不受信任的 PL/Perl 560
39.5. 缺失的特性 560
40. PL/Python - Python 过程语言 562
40.1. PL/Python 函数 562
40.2. 触发器函数 563
40.3. 数据库访问 563
41. 服务器编程接口 565
41.1. 接口函数 565
41.2. 接口支持函数 577
41.3. 内存管理 584
41.4. 数据更改的可见性 594
41.5. 示例 594
VI. 参考手册 598
I. SQL 命令 602
ABORT 604
ALTER AGGREGATE 605
ALTER CONVERSION 606
ALTER DATABASE 607
ALTER DOMAIN 609
ALTER FUNCTION 611
ALTER GROUP 612
ALTER LANGUAGE 613
ALTER OPERATOR CLASS 614
ALTER SCHEMA 615
ALTER SEQUENCE 616
ALTER TABLE 618
ALTER TRIGGER 623
ALTER USER 624
ANALYZE 627
BEGIN 629
CHECKPOINT 630
CLOSE 631
CLUSTER 632
COMMENT 634
COMMIT 636
COPY 637
CREATE AGGREGATE 643
CREATE CAST 645
CREATE CONSTRAINT TRIGGER 648

CREATE CONVERSION 649
CREATE DATABASE 651
CREATE DOMAIN 653
CREATE FUNCTION 655
CREATE GROUP 659
CREATE INDEX 660
CREATE LANGUAGE 662
CREATE OPERATOR 664
CREATE OPERATOR CLASS 667
CREATE RULE 670
CREATE SCHEMA 672
CREATE SEQUENCE 674
CREATE TABLE 677
CREATE TABLE AS 685
CREATE TRIGGER 686
CREATE TYPE 688
CREATE USER 693
CREATE VIEW 695
DEALLOCATE 697
DECLARE 698
DELETE 701
DROP AGGREGATE 703
DROP CAST 704
DROP CONVERSION 705
DROP DATABASE 706
DROP DOMAIN 707
DROP FUNCTION 708
DROP GROUP 709
DROP INDEX 710
DROP LANGUAGE 711
DROP OPERATOR 712
DROP OPERATOR CLASS 714
DROP RULE 715
DROP SCHEMA 716
DROP SEQUENCE 717
DROP TABLE 718
DROP TRIGGER 719
DROP TYPE 720
DROP USER 721
DROP VIEW 722
END 723
EXECUTE 724
EXPLAIN 725
FETCH 728
GRANT 732
INSERT 736
LISTEN 738
LOAD 739
LOCK 740
MOVE 742
NOTIFY 743
PREPARE 745
REINDEX 747
RESET 749

REVOKE 750
ROLLBACK 752
SELECT 753
SELECT INTO 763
SET 764
SET CONSTRAINTS 767
SET SESSION AUTHORIZATION 768
SET TRANSACTION 770
SHOW 772
START TRANSACTION 774
TRUNCATE 775
UNLISTEN 776
UPDATE 777
VACUUM 779
II. PostgreSQL 客户端应用 782
clusterdb 783
createdb 785
createlang 788
createuser 790
dropdb 793
droplang 795
dropuser 797
ecpg 799
pg_config 801
pg_dump 803
pg_dumpall 809
pg_restore 812
pgtclsh 818
pgtksh 819
psql 820
vacuumdb 842
III. PostgreSQL 服务器应用 845
initdb 846
initlocation 848
ipcclean 849
pg_controldata 850
pg_ctl 851
pg_resetxlog 855
postgres 856
postmaster 860
VII. 内部 865
42. PostgreSQL 内部概述 868
42.1. 查询的路径 868
42.2. 连接是如何建立的 868
42.3. 解析器阶段 869
42.4. PostgreSQL 规则系统 870
42.5. 规划器/优化器 870
42.6. 执行器 871
43. 系统目录 872
43.1. 概述 872
43.2. `pg_aggregate` 873
43.3. `pg_am` 873
43.4. `pg_amop` 874

43.5. pg_amproc 874
43.6. pg_attrdef 875
43.7. pg_attribute 875
43.8. pg_cast 877
43.9. pg_class 877
43.10. pg_constraint 879
43.11. pg_conversion 880
43.12. pg_database 880
43.13. pg_depend 881
43.14. pg_description 882
43.15. pg_group 883
43.16. pg_index 883
43.17. pg_inherits 884
43.18. pg_language 884
43.19. pg_largeobject 885
43.20. pg_listener 885
43.21. pg_namespace 886
43.22. pg_opclass 886
43.23. pg_operator 887
43.24. pg_proc 888
43.25. pg_rewrite 889
43.26. pg_shadow 889
43.27. pg_statistic 890
43.28. pg_trigger 891
43.29. pg_type 892
43.30. 系统视图 895
43.31. pg_indexes 896
43.32. pg_locks 896
43.33. pg_rules 897
43.34. pg_settings 897
43.35. pg_stats 898
43.36. pg_tables 899
43.37. pg_user 900
43.38. pg_views 900
44. 前端/后端协议 901
44.1. 概述 901
44.2. 消息流 902
44.3. 消息数据类型 911
44.4. 消息格式 911
44.5. 错误和通知消息字段 925
44.6. 自协议 2.0 以来的更改汇总 926
45. PostgreSQL 编码约定 928
45.1. 格式 928
45.2. 在服务器内部报告错误 928
45.3. 错误消息风格指南 930
46. 本地语言支持 935
46.1. 给翻译者 935
46.2. 给程序员 937
47. 编写过程语言调用处理器 940
48. 遗传查询优化器 942
48.1. 将查询处理看成是一个复杂的优化问题 942
48.2. 遗传算法 942

48.3. PostgreSQL 中的遗传查询优化（GEQO） 943
48.4. 进一步阅读 944
49. 索引代价估计函数 945
50. GiST 索引 948
50.1. 介绍 948
50.2. 可扩展性 948
50.3. 实现 948
50.4. 限制 949
50.5. 示例 949
51. 页文件 951
52. BKI 后端接口 954
52.1. BKI 文件格式 954
52.2. BKI 命令 954
52.3. 示例 955
VIII. 附录 956
A. PostgreSQL 错误代码 963
B. 日期/时间支持 969
B.1. 日期/时间输入解释 969
B.2. 日期/时间关键字 970
B.3. 历法的历史 974
C. SQL 关键字 976
D. SQL 符合性 991
D.1. 支持的特性 991
D.2. 不支持的特性 1001
E. 发布说明 1006
E.1. 版本 7.4.30 1006
E.2. 版本 7.4.29 1007
E.3. 版本 7.4.28 1008
E.4. 版本 7.4.27 1009
E.5. 版本 7.4.26 1010
E.6. 版本 7.4.25 1011
E.7. 版本 7.4.24 1011
E.8. 版本 7.4.23 1012
E.9. 版本 7.4.22 1013
E.10. 版本 7.4.21 1013
E.11. 版本 7.4.20 1014
E.12. 版本 7.4.19 1015
E.13. 版本 7.4.18 1016
E.14. 版本 7.4.17 1016
E.15. 版本 7.4.16 1017
E.16. 版本 7.4.15 1018
E.17. 版本 7.4.14 1018
E.18. 版本 7.4.13 1019
E.19. 版本 7.4.12 1020
E.20. 版本 7.4.11 1021
E.21. 版本 7.4.10 1021
E.22. 版本 7.4.9 1022
E.23. 版本 7.4.8 1023
E.24. 版本 7.4.7 1025
E.25. 版本 7.4.6 1026
E.26. 版本 7.4.5 1027
E.27. 版本 7.4.4 1027
E.28. 版本 7.4.3 1028
E.29. 版本 7.4.2 1029

E.30. 版本 7.4.1 1031
E.31. 版本 7.4 1033
E.32. 版本 7.3.21 1048
E.33. 版本 7.3.20 1049
E.34. 版本 7.3.19 1049
E.35. 版本 7.3.18 1050
E.36. 版本 7.3.17 1050
E.37. 版本 7.3.16 1051
E.38. 版本 7.3.15 1051
E.39. 版本 7.3.14 1052
E.40. 版本 7.3.13 1053
E.41. 版本 7.3.12 1054
E.42. 版本 7.3.11 1054
E.43. 版本 7.3.10 1055
E.44. 版本 7.3.9 1056
E.45. 版本 7.3.8 1057
E.46. 版本 7.3.7 1058
E.47. 版本 7.3.6 1058
E.48. 版本 7.3.5 1059
E.49. 版本 7.3.4 1060
E.50. 版本 7.3.3 1061
E.51. 版本 7.3.2 1063
E.52. 版本 7.3.1 1065
E.53. 版本 7.3 1066
E.54. 版本 7.2.8 1078
E.55. 版本 7.2.7 1078
E.56. 版本 7.2.6 1079
E.57. 版本 7.2.5 1080
E.58. 版本 7.2.4 1080
E.59. 版本 7.2.3 1081
E.60. 版本 7.2.2 1081
E.61. 版本 7.2.1 1082
E.62. 版本 7.2 1083
E.63. 版本 7.1.3 1093
E.64. 版本 7.1.2 1094
E.65. 版本 7.1.1 1094
E.66. 版本 7.1 1095
E.67. 版本 7.0.3 1099
E.68. 版本 7.0.2 1100
E.69. 版本 7.0.1 1100
E.70. 版本 7.0 1101
E.71. 版本 6.5.3 1108
E.72. 版本 6.5.2 1109
E.73. 版本 6.5.1 1109
E.74. 版本 6.5 1110
E.75. 版本 6.4.2 1115
E.76. 版本 6.4.1 1115
E.77. 版本 6.4 1116
E.78. 版本 6.3.2 1120
E.79. 版本 6.3.1 1121
E.80. 版本 6.3 1122
E.81. 版本 6.2.1 1126
E.82. 版本 6.2 1127
E.83. 版本 6.1.1 1130

E.84. 版本 6.1 1130
E.85. 版本 6.0 1133
E.86. 版本 1.09 1135
E.87. 版本 1.02 1135
E.88. 版本 1.01 1137
E.89. 版本 1.0 1139
E.90. Postgres95 Release 0.03 1140
E.91. Postgres95 Release 0.02 1142
E.92. Postgres95 Release 0.01 1142
F. 源代码仓库 1143
F.1. 通过 Git 获取源代码 1143
G. 文档 1144
G.1. DocBook 1144
G.2. 工具集 1144
G.3. 构建文档 1148
G.4. 文档编写 1151
G.5. 风格指南 1152
参考书目 1154
索引 1156

插图清单

48.1. 遗传算法的结构图 943

表格清单

4.1. 操作符优先级（递减） 28
8.1. 数据类型 70
8.2. 数字类型 71
8.3. 货币类型 74
8.4. 字符类型 74
8.5. 特殊字符类型 76
8.6. 二进制数据类型 76
8.7. `bytea`输入转义字节 76
8.8. `bytea`输出转义字节 77
8.9. 日期/时间类型 77
8.10. 日期输入 79
8.11. 时间输入 79
8.12. 时区输入 80
8.13. 特殊日期/时间输入 81
8.14. 日期/时间输出风格 81
8.15. 日期顺序习惯 82
8.16. 几何类型 84
8.17. 网络地址类型 86
8.18. `cidr` 类型输入示例 87
8.19. 对象标识符类型 96
8.20. 伪类型 96
9.1. 比较操作符 98
9.2. 数学操作符 100
9.3. 位串按位操作符 100
9.4. 数学函数 101
9.5. 三角函数 104
9.6. SQL字符串函数和操作符 104
9.7. 其他字符串函数 106
9.8. 内置转换 110
9.9. SQL二进制串函数和操作符 113
9.10. 其他二进制串函数 115
9.11. 正则表达式匹配操作符 117
9.12. 正则表达式原子 119
9.13. 正则表达式量词 119
9.14. 正则表达式约束 120
9.15. 正则表达式字符输入转义 121
9.16. 正则表达式字符类简写转义 122
9.17. 正则表达式约束转义 122
9.18. 正则表达式反向引用 122
9.19. ARE 嵌入选项字母 123
9.20. 格式化函数 125
9.21. 用于日期/时间格式化的模板模式 126
9.22. 用于日期/时间格式化的模板模式修饰符 127
9.23. 用于数值格式化的模板模式 128
9.24. `to_char`示例 129
9.25. 日期/时间操作符 130
9.26. 日期/时间函数 131
9.27. `AT TIME ZONE` 变体 137
9.28. 几何操作符 139
9.29. 几何函数 140
9.30. 几何类型转换函数 141

9.31. cidr 和 inet 操作符 142
9.32. cidr 和 inet 函数 142
9.33. macaddr 函数 144
9.34. 序列函数 144
9.35. 会话信息函数 147
9.36. 配置设定函数 148
9.37. 访问权限查询函数 149
9.38. 模式可见性查询函数 150
9.39. 系统目录信息函数 151
9.40. 注释信息函数 151
9.41. array 操作符 152
9.42. array 函数 152
9.43. 聚合函数 154
12.1. SQL 事务隔离级别 176
16.1. 短选项对照 229
16.2. System V IPC参数 230
20.1. 服务器字符集 257
20.2. 客户端/服务器字符集转换 259
23.1. 标准统计视图 272
23.2. 统计访问函数 274
29.1. pgtcl 命令 338
31.1. ConnectionPoolDataSource 实现 410
31.2. ConnectionPoolDataSource 配置属性 410
31.3. DataSource 实现 411
31.4. DataSource 配置属性 411
31.5. 池化 DataSource 的额外配置属性 411
32.1. information_schema_catalog_name 的列 415
32.2. applicable_roles 的列 415
32.3. check_constraints 的列 415
32.4. column_domain_usage 的列 415
32.5. column_privileges 的列 416
32.6. column_udt_usage 的列 416
32.7. columns 的列 417
32.8. constraint_column_usage 的列 419
32.9. constraint_table_usage 的列 420
32.10. data_type_privileges 的列 420
32.11. domain_constraints 的列 420
32.12. domain_udt_usage 的列 421
32.13. domains 的列 421
32.14. element_types 的列 423
32.15. enabled_roles 的列 425
32.16. key_column_usage 的列 425
32.17. parameters 的列 425
32.18. referential_constraints 的列 427
32.19. role_column_grants 的列 427
32.20. role_routine_grants 的列 428
32.21. role_table_grants 的列 428
32.22. role_usage_grants 的列 429
32.23. routine_privileges 的列 429
32.24. routines 的列 430
32.25. schemata 的列 432
32.26. sql_features 的列 433

32.27. sql_implementation_info 的列 433
32.28. sql_languages 的列 434
32.29. sql_packages 的列 434
32.30. sql_sizing 的列 435
32.31. sql_sizing_profiles 的列 435
32.32. table_constraints 的列 435
32.33. table_privileges 的列 436
32.34. tables 的列 436
32.35. triggers 的列 437
32.36. usage_privileges 的列 438
32.37. view_column_usage 的列 438
32.38. view_table_usage 的列 439
32.39. views 的列 439
33.1. 内置 SQL 类型的等价 C 类型 455
33.2. B-树策略 482
33.3. 哈希策略 483
33.4. R-树策略 483
33.5. B-树支持函数 483
33.6. 哈希支持函数 484
33.7. R-树支持函数 484
33.8. GiST 支持函数 484
43.1. 系统目录 872
43.2. pg_aggregate 列 873
43.3. pg_am 列 873
43.4. pg_amop 列 874
43.5. pg_amproc 列 875
43.6. pg_attrdef 列 875
43.7. pg_attribute 列 875
43.8. pg_cast 列 877
43.9. pg_class 列 877
43.10. pg_constraint 列 879
43.11. pg_conversion 列 880
43.12. pg_database 列 880
43.13. pg_depend 列 881
43.14. pg_description 列 882
43.15. pg_group 列 883
43.16. pg_index 列 883
43.17. pg_inherits 列 884
43.18. pg_language 列 884
43.19. pg_largeobject 列 885
43.20. pg_listener 列 886
43.21. pg_namespace 列 886
43.22. pg_opclass 列 886
43.23. pg_operator 列 887
43.24. pg_proc 列 888
43.25. pg_rewrite 列 889
43.26. pg_shadow 列 890
43.27. pg_statistic 列 890
43.28. pg_trigger 列 891
43.29. pg_type 列 892
43.30. 系统视图 896
43.31. pg_indexes 列 896

43.32. pg_locks 列 896
43.33. pg_rules 列 897
43.34. pg_settings 列 898
43.35. pg_stats 列 898
43.36. pg_tables 列 899
43.37. pg_user 列 900
43.38. pg_views 列 900
51.1. 页面布局 951
51.2. PageHeaderData 布局 951
51.3. HeapTupleHeaderData 布局 952
A.1. PostgreSQL Error Codes 963
B.1. 月份缩写 970
B.2. 星期缩写 970
B.3. 日期/时间字段修饰符 971
B.4. 时区缩写 971
B.5. 澳大利亚时区缩写 974
C.1. SQL 关键字 976

范例清单

8.1. 使用字符类型 75
8.2. 使用boolean类型 83
8.3. 使用位串类型 88
10.1. 指数操作符类型解析 162
10.2. 字符串连接操作符类型解析 162
10.3. 绝对值与阶乘操作符的类型解析 163
10.4. round 函数参数类型解析 165
10.5. substr 函数类型解析 165
10.6. character 的存储类型转换 166
10.7. 未充分指定类型的 UNION 中的类型解析 167
10.8. 简单 UNION 中的类型解析 167
10.9. 次序对调的 UNION 中的类型解析 167
11.1. 建立一个部分索引以排除常见值 172
11.2. 建立一个部分索引以排除不感兴趣的值 173
11.3. 建立一个部分唯一索引 173
19.1. 一个 pg_hba.conf 文件示例 249
19.2. 一个 pg_ident.conf 文件示例 253
27.1. libpq示例程序 1 320
27.2. libpq示例程序 2 323
27.3. libpq Example Program 3 325
28.1. 用 libpq 操作大对象的示例程序 333
29.1. pgtcl 示例程序 361
31.1. 在 JDBC 中处理一个简单查询 382
31.2. 设置提取数量来打开和关闭游标。 383
31.3. 在 JDBC 中删除行 384
31.4. 调用一个内置存储函数 384
31.5. 从函数获取 refcursor 值 385
31.6. 把 refcursor 当作一个独立的类型对待 385
31.7. 在 JDBC 中删除一个表 386
31.8. 在 JDBC 中处理二进制数据 387
31.9. DataSource 代码示例 412
31.10. DataSource JNDI 代码示例 412
36.1. PL/pgSQL 的手工安装 518
37.1. 一个 PL/pgSQL 触发器过程 541
37.2. 把简单函数从 PL/SQL 移植到 PL/pgSQL 542
37.3. 把创建另一函数的函数从 PL/SQL 移植到 PL/pgSQL 543
37.4. 把带字符串处理和 OUT 参数的过程从 PL/SQL 移植到 PL/pgSQL 544
37.5. 把过程从 PL/SQL 移植到 PL/pgSQL 545

前言

本书是PostgreSQL的官方文档。它由 PostgreSQL的开发者和其他志愿者与 PostgreSQL软件的开发并行编写。它描述当前版本的PostgreSQL正式支持的所有功能。

为了使关于PostgreSQL的大量信息便于管理，本书被组织成几个部分。
每一部分面向不同类别的用户，或处于PostgreSQL使用经验不同阶段的用户：

- 第 I 部分是新用户的非正式入门介绍。
- 第 II 部分文档化SQL查询语言环境，包括数据类型和函数，以及用户级的性能调整。每个PostgreSQL用户都应阅读这一部分。
- 第 III 部分描述服务器的安装和管理。每个运行 PostgreSQL服务器的人，无论是自用还是供他人使用，都应阅读这一部分。
- 第 IV 部分描述 PostgreSQL客户端程序的编程接口。
- 第 V 部分包含供高级用户使用的关于服务器可扩展能力的信息。主题例如用户定义的数据类型和函数。
- 第 VI 部分包含 SQL 命令、客户端和服务器程序的语法信息。这一部分以命令或程序为序提供结构化的信息，支持其他各部分。
- 第 VII 部分包含可能对 PostgreSQL开发者有用的各类信息。

1. 什么是PostgreSQL?

PostgreSQL是一个对象-关系数据库管理系统（ORDBMS），
它基于加州大学伯克利分校计算机科学系开发的 POSTGRES, Version 4.2[1]。POSTGRES 开创的许多概念直到很久以后才在一些商业数据库系统中出现。

PostgreSQL是这一原始伯克利代码的开源后代。它支持 SQL92 和 SQL99，并提供许多现代特性：

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

此外，PostgreSQL可以在许多方面由用户扩展，例如通过添加新的

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

而且由于宽松的许可证，任何人都可以免费使用、修改和分发 PostgreSQL，用于任何目的，无论是私人的、商业的还是学术的。

[1] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

2. PostgreSQL简史

如今名为PostgreSQL的对象关系数据库管理系统，源自加州大学伯克利分校编写的POSTGRES软件包。经过十多年的发展，PostgreSQL如今已成为当今最先进的开源数据库。

2.1. 伯克利 POSTGRES 项目

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

自那以后，POSTGRES 经历了几次重大版本发布。第一个“演示版”系统于 1987 年开始运行，并在 1988 年的 ACM-SIGMOD 大会上进行了展示。版本 1 在[ston90a]中有描述，并于 1989 年 6 月发布给少数外部用户。针对第一代规则系统所受到的批评（[ston89]），规则系统被重新设计（[ston90b]），版本 2 连同新的规则系统于 1990 年 6 月发布。版本 3 于 1991 年问世，增加了对多个存储管理器的支持、改进后的查询执行器以及重写后的规则系统。此后直到 Postgres95（见下文）之前的后续版本，主要聚焦于可移植性和可靠性。

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

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

2.2. Postgres95

1994 年，Andrew Yu 和 Jolly Chen 为 POSTGRES 添加了一个 SQL 语言解释器。随后，它以 Postgres95 这一新名称发布到网络上，作为最初的 POSTGRES 伯克利代码的开源后继者自行发展。

Postgres95的代码完全采用 ANSI C 编写，并将体积缩减了 25%。许多内部更改提升了性能和可维护性。Postgres951.0.x 版本在 Wisconsin Benchmark 上快了大约 30-50%，比较对象是 POSTGRES4.2 版。除了错误修复之外，主要增强还包括：

- 查询语言 PostQUEL 被 SQL 所取代（在服务器中实现）。直到 PostgreSQL（见下文）才支持子查询，但在 Postgres95 中可以用用户定义的 SQL 函数来模拟。聚合函数也得到了重新实现，并增加了对 `GROUP BY` 查询子句的支持。
- 除了 monitor 程序之外，还提供了一个用于交互式 SQL 查询的新程序（psql），它使用了 GNU Readline。
- 新增的前端库 `libpgtcl` 支持基于 Tcl 的客户端。示例 shell `pgtclsh` 提供了新的 Tcl 命令，用于让 Tcl 程序与 Postgres95 服务器交互。
- 大对象接口经过了全面改造。反转大对象成为存储大对象的唯一机制。（反转文件系统已被移除。）

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

- 实例级规则系统被移除。规则仍然可以作为重写规则使用。
- 源代码中附带了一份简短教程，介绍常规 SQL 特性以及 Postgres95 的特性。
- 构建时使用的是 GNU make（而不是 BSD make）。此外，Postgres95 可以用未经修改的 GCC 编译（双精度浮点数的数据对齐问题已修复）。

2.3. PostgreSQL

到了 1996 年，很明显“Postgres95”这个名字经不起时间的考验。我们选择了一个新名字 PostgreSQL，以体现最初的 POSTGRES 与后来具备 SQL 能力的版本之间的关系。同时，我们将版本号从 6.0 开始，使编号重新回到最初由伯克利 POSTGRES 项目开启的序列中。

在开发 Postgres95 期间，重点是识别并理解服务器代码中已有的问题。到了 PostgreSQL 阶段，重点则转向增强特性和能力，不过各个方面的工作仍在继续。

自那以后 PostgreSQL 发生了哪些变化，可见附录 E。

3. 约定

本书使用以下排版约定来标记某些文本：新术语、外来语和其他重要段落以斜体强调。所有表示计算机输入或输出的内容，特别是命令、程序代码和屏幕输出，均以等宽字体（示例）显示。在这类内容中，斜体（*`example`*）表示占位符，你必须用实际值来替换占位符。有时，程序代码中相对前一个示例新增或修改的部分会以粗体（**`示例`**）强调。

以下约定用于命令概要：方括号（`[`和`]`）表示可选部分。（在 Tcl 命令的概要中，改用问号（`?`）表示，这是 Tcl 的惯例。）大括号（`{`和`}`）和竖线（`|`）表示必须选择其中一个替代项。点（`...`）表示前面的元素可以重复。

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

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

4. 更多信息

除了本文档，也就是本书之外，还有其他关于PostgreSQL的资源：

FAQ

FAQ 列表持续更新地收录了对常见问题的解答。

README

大多数附加软件包都带有`README`文件。

网站

PostgreSQL 网站[5]提供最新版本的详细信息，以及其他能让你在工作中或日常使用 PostgreSQL时更加高效的信息。

[5] http://www.postgresql.org

邮件列表

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

你自己！

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

5. 缺陷报告指南

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

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

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

5.1. 识别缺陷

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

- 程序因致命信号而终止，或者给出一条表明程序本身存在问题的操作系统错误消息。（反例可能是“磁盘满”之类的消息，因为那得由你自己解决。）
- 程序对给定输入产生了错误的输出。
- 程序拒绝接受有效输入（按文档定义）。
- 程序接受了无效输入，却没有给出提示或错误消息。但请记住，你认为的无效输入，可能正是我们认为的扩展功能，或者是对传统做法的兼容。
- 在受支持的平台上，PostgreSQL无法按照说明完成编译、构建或安装。

这里的“程序”指的是任何可执行程序，不只是后端服务器。

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

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

5.2. 报告哪些内容

关于缺陷报告，最重要的一点是陈述全部事实，而且只陈述事实。
不要猜测你觉得哪里出了问题、“它看起来像是在做什么”，或者程序的哪一部分有错。
如果你不熟悉实现，十有八九会猜错，对我们毫无帮助。即使你熟悉实现，
有根据的解释也只能是事实的有益补充，而不能替代事实。如果我们要修复这个缺陷，
仍然必须先亲自重现它。陈述这些基本事实并不难做（你大概可以直接从屏幕上复制粘贴），
但太多时候，重要细节会被遗漏，因为有人觉得它们不重要，或者认为即使不说清楚，
报告也一样能被理解。

以下各项应包含在每一个缺陷报告中：

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

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

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

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

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

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

- 你期望得到的输出，也非常重要。如果你只是写“这个命令给了我那个输出。”
  或“这不是我期望的。”，我们可能会自己运行一遍，扫一眼输出，然后觉得它看起来没问题，
  也正是我们所期望的。我们不应该花时间去揣摩你的命令背后的确切语义。
  尤其不要只是说“这不是 SQL 规定的/Oracle 所做的。”从SQL标准中找出正确行为并不是一件

轻松的事，而且我们也不都了解其他所有关系数据库的行为。（如果你的问题是程序崩溃，显然可以省略这一项。）

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

- 任何偏离安装说明的做法。

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

  如果你的版本早于 7.4.30，我们几乎肯定会建议你升级。每个新版本都包含大量的缺陷修复，这正是我们发布新版本的原因。

- 平台信息。这包括内核名称和版本、C 库、处理器、内存信息等。在大多数情况下，报告供应商和版本就足够了，但不要想当然地认为所有人都知道“Debian”具体包含什么，或者所有人都运行在 i386 上。如果你遇到的是安装问题，那么你机器上的工具链信息（编译器、make 等）也是必需的。

不要担心你的缺陷报告会变得很长；这很正常。第一次就把所有内容都报告出来，总比让我们反复追问细节要好。另一方面，如果你的输入文件非常大，先问问是否有人愿意看一看，也是合理的。这里有一篇文章[6]，概述了更多关于报告缺陷的建议。

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

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

5.3. 到哪里报告缺陷

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

另一种方法是填写项目网站[7]上提供的缺陷报告网页表单。通过这种方式提交的缺陷报告，也会被发送到 `<pgsql-bugs@postgresql.org>`邮件列表。

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

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

[6] http://www.chiark.greenend.org.uk/~sgtatham/bugs.html
[7] http://www.postgresql.org/

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

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

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

注意

由于垃圾邮件泛滥，上述所有电子邮件地址都是封闭邮件列表。也就是说，你必须订阅某个列表才能向它发帖。（不过，使用缺陷报告 Web 表单无需订阅。）如果你希望发邮件但不想接收列表邮件，可以订阅并把订阅选项设置为nomail。要了解更多信息，请向 <majordomo@postgresql.org>发送邮件，正文只写一个词 help。

部分 I. 教程

欢迎阅读PostgreSQL教程。后面这几章旨在为对 PostgreSQL、关系数据库概念或 SQL 语言中任何一个方面感到陌生的读者，提供关于它们的简单介绍。我们只假定读者具备一些使用计算机的常识。不要求特定的 Unix 或编程经验。这一部分主要是让你获得 PostgreSQL 系统的一些重要方面的动手经验。它并不试图对所涵盖的主题做出完整或彻底的论述。

在完成本教程的学习之后，你可能想继续阅读第 II 部分以获得对 SQL 语言更正式的了解，或者阅读第 IV 部分了解开发 PostgreSQL 应用程序的信息。那些自己搭建和管理服务器的人还应当阅读第 III 部分。

目录

1. 从头开始 3
1.1. 安装 3
1.2. 架构基础 3
1.3. 创建数据库 3
1.4. 访问数据库 5
2. SQL语言 7
2.1. 简介 7
2.2. 概念 7
2.3. 创建新表 7
2.4. 向表中填充行 8
2.5. 查询表 9
2.6. 表之间的连接 10
2.7. 聚合函数 12
2.8. 更新 13
2.9. 删除 13
3. 高级特性 15
3.1. 简介 15
3.2. 视图 15
3.3. 外键 15
3.4. 事务 16
3.5. 继承 17
3.6. 小结 18

第 1 章 从头开始

1.1. 安装

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

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

如果你要自己安装PostgreSQL，请参阅第 14 章中的安装说明，待安装完成后再回到本指南。务必仔细遵循关于设置适当环境变量的小节。

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

1.2. 架构基础

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

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

- 一个服务器进程，负责管理数据库文件、接受客户端应用到数据库的连接，
  并代表客户端执行数据库操作。数据库服务器程序叫做`postmaster`。

- 用户用来执行数据库操作的客户端（前端）应用。
  客户端应用的形式可以非常多样：可以是一个文本工具，也可以是一个图形应用程序，
  或者是一个通过访问数据库来显示网页的 Web 服务器，还可以是专门的数据库维护工具。有些客户端应用随PostgreSQL发行版一起提供；大多数则由用户开发。

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

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

1.3. 创建数据库

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

你的站点管理员可能已经为你创建了一个可用的数据库。
他应该已经告诉你你的数据库名是什么。如果是这样，你就可以省略这一步，直接跳到下一节。

要创建一个新数据库（本例中命名为 `mydb`），请使用以下命令：

```
$createdb mydb
```

它应当产生如下响应：

```
CREATE DATABASE
```

如果是这样，这一步就成功了，你可以跳过本节余下的内容。

如果你看到类似下面这样的信息：

```
createdb: command not found
```

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

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

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

另一种响应可能是这样：

```
createdb: could not connect to database template1: could not connect
 to server:
No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

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

如果你没有创建数据库所需的权限，那么你会看到下面的信息：

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

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

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

```
$createdb
```

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

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

```
$dropdb mydb
```

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

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

1.4. 访问数据库

一旦你创建了数据库，你就可以通过以下方式访问它：

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

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

```
$psql mydb
```

如果你不指定数据库名，那么它默认就是你的用户帐号名。你在上一节已经见识过这种机制了。

在psql中，你将看到下面的欢迎信息：

```
Welcome to psql 7.4.30, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

mydb=>
```

最后一行也可能是：

```
mydb=#
```

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

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

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

```
mydb=>SELECT version();
                            version
-----------------------------------------------------------------
 PostgreSQL 7.4.30 on i586-pc-linux-gnu, compiled by GCC 2.96
```

```
(1 row)

mydb=> SELECT current_date;
    date
------------
 2002-08-31
(1 row)

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

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

```
mydb=>\h
```

要退出psql，输入：

```
mydb=>\q
```

psql将退出并返回到命令 shell。（要了解更多内部命令，可输入\?，在psql提示符下执行。）psql的完整功能见psql。如果PostgreSQL 安装正确，你还可以在操作系统 shell 提示符下输入man psql来查看文档。在本教程中，我们不会明确使用这些功能，但在有帮助时你可以自行使用它们。

第 2 章 SQL语言

2.1. 简介

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

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

本手册中的例子也可以在 PostgreSQL 源码发行包的 src/tutorial/ 目录中找到。如何使用它们请参考该目录下的 README 文件。要开始教程，请执行以下操作：

```
$cd ..../src/tutorial$psql -s mydb...mydb=>\i basics.sql
```

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

2.2. 概念

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

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

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

2.3. 创建新表

你可以通过指定表名以及各列的列名和类型来创建一个新表：

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);
```

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

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

varchar(80) 声明了一个能存储任意字符串、长度上限为 80 个字符的数据类型。int 是普通的整数类型。real 是用于存储单精度浮点数的类型。date 顾名思义。（是的，类型为 date 的列也正好名叫 date。这可能是方便也可能是困扰——由你选择。）

PostgreSQL 支持常见的 SQL 类型 int、smallint、real、double precision、char(*N*)、varchar(*N*)、date、time、timestamp 和 interval，还有其他一些通用类型以及一套丰富的几何类型。PostgreSQL 可以用任意数量的用户自定义数据类型来定制。因此，类型名不是语法上的关键字，除非为了支持 SQL 标准中的特殊情形所必需。

第二个示例将存储城市及其关联的地理位置：

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

point类型就是PostgreSQL特有数据类型的一个例子。

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

```
DROP TABLE tablename;
```

2.4. 向表中填充行

INSERT语句用于向表中填充行：

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

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

point类型要求一个坐标对作为输入，如下所示：

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

到目前为止所用的语法要求你记住列的顺序。另一种语法允许你显式列出列：

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

如果愿意，你可以按不同顺序列出列，甚至省略某些列，例如降水量未知时：

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

许多开发者认为，与依赖隐含顺序相比，显式列出列是一种更好的风格。

请把上面展示的所有命令都输入一遍，这样你在下面各节中才有一些数据可供操作。

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

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

其中，源文件必须可以在后端服务器机器上访问，而不是客户端机器，
因为后端服务器直接读取该文件。有关COPY命令的更多信息，请参见COPY。

2.5. 查询表

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

```
SELECT * FROM weather;
```

（这里 * 表示“所有列”）输出应为：

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

你可以在选择列表中指定任意表达式。例如：

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

这样应该得到：

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

注意 AS 子句是如何用来重新标注输出列的。（它是可选的。）

查询的限定条件中允许使用任意布尔运算符（AND、OR 和 NOT）。例如，下面的查询检索旧金山在下雨天的天气：

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

结果为：

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

最后提一句，你可以要求查询结果按排序顺序返回，或者去掉重复的行：

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

     city
```

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

当然，DISTINCT 和 ORDER BY 也可以分开使用。

2.6. 表之间的连接

到目前为止，我们的查询每次只访问一个表。查询可以同时访问多个表，
也可以访问同一个表并同时处理其中的多行。一次访问同一个表或不同表中多行的查询称为
连接查询。例如，假设要列出所有天气记录及其对应城市的位置。为此，
需要把weather表每一行的city列与cities表所有行的name列进行比较，
并选出这些值匹配的行对。

> **注意**
>
> 这只是一个概念模型。实际的连接可能以更高效的方式执行，但这对用户是不可见的。

这可以用下面的查询来完成：

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

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

注意结果集的两点：

- 没有与城市 Hayward 对应的结果行。这是因为cities表中没有与 Hayward 匹配的项，所以连接会忽略weather表中的不匹配行。我们很快就会看到如何修正这一点。
- 有两列包含城市名称。这是正确的，因为来自weather和cities表的列列表被拼接在一起。不过，实际中通常不希望这样，因此可能更希望显式列出输出列，而不使用*：

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

练习：. 试着确定省略WHERE子句时，这条查询的含义。

由于所有列的名称各不相同，解析器自动确定了它们所属的表。
但最好养成在连接查询中完全限定列名的良好风格：

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
```

```
    FROM weather, cities
    WHERE cities.name = weather.city;
```

目前所见的连接查询，也可以采用下面这种形式编写：

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

这种语法不如上面的语法常用，不过这里展示它是为了帮助理解后续主题。

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

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

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

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

练习：. 还有右外连接和全外连接。试着找出它们能做什么。

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

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

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

这里我们把 weather 表重命名为 W1 和 W2，以便区分连接的左侧和右侧。你也可以在其他查询中使用这类别名来少打些字，例如：

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

你会经常遇到这种缩写方式。

2.7. 聚合函数

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

例如，我们可以用下面的语句找出所有记录中最低温度读数的最高值：

```
SELECT max(temp_lo) FROM weather;

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

如果我们想知道该读数出现在哪个城市（或哪些城市），可能会尝试：

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

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

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

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

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

聚合与GROUP BY子句结合使用也很有用。例如，可以通过下面的查询，得到每个城市所观测到的最低温度的最大值：

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

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

这会为每个城市输出一行。每个聚合结果都是针对与该城市匹配的表行计算的。
这些分组行可以用以下子句过滤：HAVING：

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

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

这样就只会为所有temp_lo值均小于 40 的城市返回相同的结果。最后，如果只关心名称以“S”开头的城市，可以这样做：

```
SELECT city, max(temp_lo)
    FROM weather
    WHERE city LIKE 'S%'1
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

1 LIKE操作符执行模式匹配，详细说明见第 9.6 节。

理解聚合与 SQL 的 WHERE 和 HAVING 子句之间的交互很重要。WHERE 与 HAVING 的根本区别在于：WHERE 在分组和聚合计算之前选择输入行（因此它控制哪些行进入聚合计算），而 HAVING 在分组和聚合计算之后选择分组行。因此，WHERE 子句不得包含聚合函数；试图用聚合来决定哪些行作为聚合的输入是没有意义的。另一方面，HAVING 子句总是包含聚合函数。（严格说来，你也可以写不使用聚合的 HAVING 子句，但那是浪费：同样的条件放在 WHERE 阶段会更高效。）

注意，我们可以把城市名的限制条件放在 WHERE 中，因为它不需要聚合。这比把该限制加到 HAVING 更高效，因为我们避免了对所有未通过 WHERE 检查的行做分组和聚合计算。

2.8. 更新

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

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

看看数据的新状态：

```
SELECT * FROM weather;

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

2.9. 删除

假设你不再关心 Hayward 的天气。那么你可以执行以下操作从表中删除这些行。删除是用 DELETE 命令执行的：

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

所有属于 Hayward 的天气记录都会被删除。

```
SELECT * FROM weather;

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

对如下形式的语句必须保持警惕：

```
DELETE FROM tablename;
```

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

第 3 章 高级特性

3.1. 简介

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

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

3.2. 视图

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

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

SELECT * FROM myview;
```

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

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

3.3. 外键

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

新的表声明如下：

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

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

```
 date      date
);
```

现在试着插入一条无效记录：

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

ERROR:  insert or update on table "weather" violates foreign key
 constraint "$1"
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

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

3.4. 事务

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

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

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

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

我们还希望保证，一旦事务完成并得到数据库系统确认，它确实已经被永久记录下来，
即便紧接着发生崩溃也不会丢失。例如，如果我们正在记录Bob的一次现金提款，
我们当然不希望他刚走出银行大门，对他账户的扣款就在崩溃后消失。事务型数据库保证，
在把事务报告为完成之前，会先将该事务所做的全部更新记入永久存储（即磁盘）。

事务型数据库的另一个重要性质与原子更新的概念密切相关：当多个事务并发运行时，
每个事务都不应该看到其他事务尚未完成的更改。例如，
如果某个事务正在统计所有营业网点的余额，就不能让它只算进Alice所在网点的扣款，
却没有算进Bob所在网点的入账，反过来也不行。因此，
事务不仅在对数据库的持久影响上必须是全有或全无的，

在其发生过程中的可见性上也必须如此。一个未结束事务到目前为止所做的更新，对其他事务不可见；等到该事务完成时，所有更新会同时变得可见。

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

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- etc etc
COMMIT;
```

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

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

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

3.5. 继承

继承是面向对象数据库中的一个概念。它为数据库设计打开了一些有趣的新可能性。

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

```
CREATE TABLE capitals (
  name       text,
  population real,
  altitude   int,    -- (in ft)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude  int     -- (in ft)
);

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

就查询而言，这样做还行，但一旦需要更新多行，它就会变得很麻烦。

更好的解决办法是：

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int     -- (in ft)
);

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

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

例如，下面的查询会找出所有海拔超过500英尺的城市名称，其中也包括州首府：

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

返回：

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

另一方面，下面的查询会找出所有海拔500英尺或更高且不是州首府的城市：

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

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

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

3.6. 小结

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

如果你觉得还需要更多入门材料，请访问 PostgreSQL 官方网站[1]，其中有指向更多资源的链接。

[1] http://www.postgresql.org

部分 II. SQL 语言

本部分描述SQL语言在PostgreSQL中的用法。我们首先介绍SQL的一般语法，
然后说明如何创建保存数据的结构、如何填充数据库以及如何查询它。中间部分列出可在
SQL数据命令中使用的数据类型和函数。其余部分讨论为获得最佳性能而调优数据库时重要的几个方面。

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

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

目录

4. SQL 语法 24
4.1. 词法结构 24
4.1.1. 标识符和关键词 24
4.1.2. 常量 25
4.1.3. 操作符 27
4.1.4. 特殊字符 27
4.1.5. 注释 28
4.1.6. 词法优先级 28
4.2. 值表达式 29
4.2.1. 列引用 30
4.2.2. 位置参数 30
4.2.3. 下标 30
4.2.4. 字段选择 31
4.2.5. 操作符调用 31
4.2.6. 函数调用 31
4.2.7. 聚合表达式 31
4.2.8. 类型转换 32
4.2.9. 标量子查询 33
4.2.10. 数组构造器 33
4.2.11. 表达式计算规则 34
5. 数据定义 36
5.1. 表基础 36
5.2. 系统列 37
5.3. 默认值 38
5.4. 约束 38
5.4.1. 检查约束 39
5.4.2. 非空约束 40
5.4.3. 唯一约束 41
5.4.4. 主键 41
5.4.5. 外键 42
5.5. 继承 44
5.6. 修改表 46
5.6.1. 增加列 46
5.6.2. 移除列 47
5.6.3. 增加约束 47
5.6.4. 移除约束 47
5.6.5. 更改默认值 47
5.6.6. 重命名列 48
5.6.7. 重命名表 48
5.7. 权限 48
5.8. 模式 49
5.8.1. 创建模式 49
5.8.2. 公共模式 50
5.8.3. 模式搜索路径 50
5.8.4. 模式和权限 51
5.8.5. 系统目录模式 52
5.8.6. 使用方式 52
5.8.7. 可移植性 52
5.9. 其他数据库对象 52
5.10. 依赖跟踪 53
6. 数据操纵 54

6.1. 插入数据 54
6.2. 更新数据 55
6.3. 删除数据 55
7. 查询 57
7.1. 概述 57
7.2. 表表达式 57
7.2.1. FROM 子句 57
7.2.2. WHERE 子句 63
7.2.3. GROUP BY 和 HAVING 子句 64
7.3. 选择列表 66
7.3.1. 选择列表项 66
7.3.2. 列标签 66
7.3.3. DISTINCT 67
7.4. 组合查询 67
7.5. 行排序 68
7.6. LIMIT 和 OFFSET 69
8. 数据类型 70
8.1. 数字类型 71
8.1.1. 整数类型 71
8.1.2. 任意精度数字 72
8.1.3. 浮点类型 73
8.1.4. serial 类型 73
8.2. 货币类型 74
8.3. 字符类型 74
8.4. 二进制数据类型 76
8.5. 日期/时间类型 77
8.5.1. 日期/时间输入 78
8.5.2. 日期/时间输出 81
8.5.3. 时区 82
8.5.4. 内部实现 83
8.6. 布尔类型 83
8.7. 几何类型 84
8.7.1. 点 84
8.7.2. 线段 85
8.7.3. 方框 85
8.7.4. 路径 85
8.7.5. 多边形 85
8.7.6. 圆 86
8.8. 网络地址类型 86
8.8.1. inet 86
8.8.2. cidr 86
8.8.3. inet vs. cidr 87
8.8.4. macaddr 87
8.9. 位串类型 88
8.10. 数组 88
8.10.1. 数组类型的声明 88
8.10.2. 数组值输入 89
8.10.3. 访问数组 91
8.10.4. 修改数组 92
8.10.5. 在数组中搜索 94
8.10.6. 数组输入和输出语法 94
8.11. 对象标识符类型 95
8.12. 伪类型 96

9. 函数和操作符 98
9.1. 逻辑操作符 98
9.2. 比较操作符 98
9.3. 数学函数和操作符 100
9.4. 字符串函数和操作符 104
9.5. 二进制串函数和操作符 113
9.6. 模式匹配 115
9.6.1. LIKE 116
9.6.2. SIMILAR TO 和 SQL99 正则表达式 116
9.6.3. POSIX正则表达式 117
9.7. 数据类型格式化函数 125
9.8. 日期/时间函数和操作符 130
9.8.1. EXTRACT, date_part 133
9.8.2. date_trunc 136
9.8.3. AT TIME ZONE 137
9.8.4. 当前日期/时间 138
9.9. 几何函数和操作符 139
9.10. 网络地址类型函数 142
9.11. 序列操作函数 144
9.12. 条件表达式 145
9.12.1. CASE 145
9.12.2. COALESCE 147
9.12.3. NULLIF 147
9.13. 杂项函数 147
9.14. 数组函数和操作符 152
9.15. 聚合函数 153
9.16. 子查询表达式 155
9.16.1. EXISTS 155
9.16.2. IN 155
9.16.3. NOT IN 156
9.16.4. ANY/SOME 156
9.16.5. ALL 157
9.16.6. 按行比较 157
9.17. 行和数组比较 157
9.17.1. IN 158
9.17.2. NOT IN 158
9.17.3. ANY/SOME（数组） 158
9.17.4. ALL（数组） 159
9.17.5. 按行比较 159
10. 类型转换 160
10.1. 概述 160
10.2. 操作符 161
10.3. 函数 164
10.4. 值存储 166
10.5. UNION、CASE和ARRAY结构 166
11. 索引 168
11.1. 简介 168
11.2. 索引类型 168
11.3. 多列索引 169
11.4. 唯一索引 170
11.5. 表达式索引 170
11.6. 操作符类 171
11.7. 部分索引 172

11.8. 检查索引使用情况 174
12. 并发控制 176
12.1. 简介 176
12.2. 事务隔离 176
12.2.1. 读已提交隔离级别 177
12.2.2. 可串行化隔离级别 177
12.3. 显式锁定 178
12.3.1. 表级锁 178
12.3.2. 行级锁 180
12.3.3. 死锁 180
12.4. 应用级别的数据一致性检查 181
12.5. 锁与索引 181
13. 性能提示 183
13.1. 使用EXPLAIN 183
13.2. 规划器使用的统计信息 186
13.3. 用显式JOIN子句控制规划器 188
13.4. 填充一个数据库 189
13.4.1. 禁用自动提交 189
13.4.2. 使用COPY FROM 189
13.4.3. 移除索引 190
13.4.4. 增大sort_mem 190
13.4.5. 之后运行ANALYZE 190

第 4 章 SQL 语法

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

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

4.1. 词法结构

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

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

此外，注释也可以出现在 SQL 输入中。它们不是词元，实际上等价于空白。

例如，下面是一个（语法上）合法的SQL输入：

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

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

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

4.1.1. 标识符和关键词

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

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

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

关键词和未加引号的标识符不区分大小写。因此：

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

也可以等价地写成：

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

一种常见约定是关键词使用大写、名称使用小写，例如：

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

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

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

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

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

4.1.2. 常量

在 PostgreSQL 中有三种隐式类型常量：字符串、位串和数字。也可以为常量指定显式类型，这样系统就能更准确地表示并更高效地处理它们。隐式常量将在下面介绍；显式常量在随后讨论。

4.1.2.1. 字符串常量

SQL 中的字符串常量是由单引号（`'`）括起的任意字符序列，例如 `'This is a string'`。SQL 允许在字符串中嵌入单引号，方法是写两个相邻的单引号，例如 `'Dianne''s horse'`。在 PostgreSQL 中，也可以用反斜线（`\`）对单引号转义，例如 `'Dianne\'s horse'`。

也可以使用 C 风格的反斜线转义：`\b` 是退格，`\f` 是换页，`\n` 是换行，`\r` 是回车，`\t` 是制表符，而 `\`*xxx*（其中 *xxx* 是一个八进制数）是具有相应代码的一个字节。（你有责任确保所创建的字节序列构成服务器字符集编码中的有效字符。）
跟随在反斜线后面的任何其他字符都按其字面意思对待。因此，
要在字符串常量中包含一个反斜线，请写两个反斜线。

编码为零的字符不能出现在字符串常量中。

两个只由空白及至少一个新行分隔的字符串常量会被连接在一起，
并且实际上被当作该字符串写成一个常量来对待。例如：

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

等同于

```
SELECT 'foobar';
```

但是

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

不是合法的语法。（这一略显古怪的行为是 SQL 标准规定的；PostgreSQL 遵循该标准。）

4.1.2.2. 位串常量

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

或者，也可以用十六进制记法指定位串常量，即在开头使用 `X`（大写或小写形式），例如 `X'1FF'`。这种记法等价于每个十六进制位对应四个二进制位的位串常量。

两种形式的位串常量都可以以与常规字符串常量相同的方式跨行继续。

4.1.2.3. 数字常量

数字常量接受下列一般形式：

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

其中 *digits* 是一个或多个十进制数字（0 到 9）。如果使用小数点，小数点前后必须至少有一位数字。如果存在指数标记（`e`），其后必须至少跟随一位数字。
常量中不能嵌入任何空格或其他字符。注意，
任何前导的加号或减号实际上不被视为常量的一部分；它是应用于该常量的操作符。

这些是合法数字常量的一些示例：

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

一个既不包含小数点也不包含指数的数字常量，如果其值适合 `integer` 类型（32 位），最初被假定为 `integer` 类型；否则如果其值适合 `bigint` 类型（64 位），被假定为 `bigint` 类型；再否则被当作 `numeric` 类型。包含小数点和/或指数的常量总是最初被假定为 `numeric` 类型。

为数字常量最初指定的数据类型，只是类型解析算法的起点。在大多数情况下，
常量会根据上下文被自动强制转换成最合适的类型。必要时，
你可以通过类型转换强制把一个数值解释为特定的数据类型。例如，你可以通过写成

```
REAL '1.23'  -- string style
1.23::REAL   -- PostgreSQL (historical) style
```

强制把一个数值当作 `real`（`float4`）类型对待。

4.1.2.4. 其他类型的常量

一个任意类型的常量可以用下列记法之一输入：

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

字符串的文本被传递给名为 *type* 的类型的输入转换例程。结果就是所指示类型的一个常量。如果常量必须是哪个类型没有歧义（例如，当它作为参数传递给一个未重载的函数时），显式类型转换可以省略，此时它会被自动强制转换。

也可以用类似函数调用的语法来指定类型强制转换：

typename ('*string*')

但并非所有类型名都可以这样使用；详见第 4.2.8 节。

`::`、`CAST()` 以及函数调用语法也可以用来指定任意表达式的运行时类型转换，如第 4.2.8 节中所述。但 *type* '*string*' 形式只能用来指定字面常量的类型。*type* '*string*' 的另一个限制是它不适用于数组类型；要用 `::` 或 `CAST()` 来指定数组常量的类型。

4.1.3. 操作符

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

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

不过，操作符名称也有一些限制：

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

  ~ ! @ # % ^ & | ` ?

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

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

4.1.4. 特殊字符

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

- 美元符号（`$`）后跟数字时，用来表示函数定义体或预备语句中的位置参数。在其他上下文中，美元符号可以是标识符的一部分。
- 圆括号（`()`）具有它们通常的含义，用来对表达式分组并确定运算优先级。在某些情况中，圆括号被要求作为一个特定 SQL 命令的固定语法的一部分。
- 方括号（`[]`）被用来选择一个数组中的元素。更多关于数组的信息见第 8.10 节。
- 逗号（`,`）被用在某些语法结构中来分割一个列表的元素。
- 分号（`;`）结束一个 SQL 命令。它不能出现在一个命令中间的任何位置，除了在一个字符串常量中或者一个带引号的标识符中。
- 冒号（`:`）被用来从数组中选择“切片”（见第 8.10 节）。在某些 SQL 的“方言”（例如嵌入式 SQL）中，冒号被用来作为变量名的前缀。

- 星号（`*`）在`SELECT`命令中或与`COUNT`聚合函数一起使用时有特殊含义。
- 句点（`.`）被用在数字常量中，并且被用来分割模式、表和列名。

4.1.5. 注释

注释是一串以双连字符开始并延伸到行尾的字符，例如：

```
-- 这是一条标准 SQL 注释
```

另外，也可以使用 C 风格注释块：

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

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

在进一步进行语法分析之前，注释会从输入流中移除，并实际被替换为空白。

4.1.6. 词法优先级

表 4.1展示了 PostgreSQL 中操作符的优先级和结合性。大多数操作符具有相同的优先级，并且是左结合的。操作符的优先级和结合性是固定写在解析器中的。这可能导致不直观的行为；例如，布尔操作符 `<` 和 `>` 与布尔操作符 `<=` 和 `>=` 的优先级不同。此外，在组合使用二元和一元操作符时，有时需要添加圆括号。例如：

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

将被解析为：

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

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

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

这是为可扩展性付出的代价。

表 4.1. 操作符优先级（递减）

操作符/元素	结合性	描述
`.`	左	表/列名分隔符
`::`	左	PostgreSQL-风格的类型转换
`[ ]`	左	数组元素选择
`-`	右	一元负号
`^`	左	求幂
`* / %`	左	乘、除、取模
`+ -`	左	加、减
`IS`		`IS TRUE`、`IS FALSE`、`IS UNKNOWN`、`IS NULL`

操作符/元素	结合性	描述
ISNULL		空值测试
NOTNULL		非空测试
（任意其他）	左	所有其他内建和用户定义的操作符
IN		集合成员关系
BETWEEN		范围包含
OVERLAPS		时间区间重叠
LIKE ILIKE SIMILAR		字符串模式匹配
< >		小于、大于
=	右	相等、赋值
NOT	右	逻辑否定
AND	左	逻辑合取
OR	左	逻辑析取

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

当一个模式限定的操作符名被用在OPERATOR语法中时，如下面的示例：

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

OPERATOR 结构在优先级上会被视为表 4.1中为“任意其他”操作符所示的优先级。无论OPERATOR()中指定的是哪个具体操作符，这一点都成立。

4.2. 值表达式

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

一个值表达式是下列之一：

- 一个常量或字面值
- 一个列引用
- 在一个函数定义体或预备语句中的一个位置参数引用
- 一个下标表达式
- 一个字段选择表达式
- 一个操作符调用
- 一个函数调用
- 一个聚合表达式
- 一个类型转换

- 一个标量子查询
- 一个数组构造器
- 另一个圆括号中的值表达式（用于对子表达式分组并覆盖优先级）

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

我们已经在第 4.1.2 节中讨论过常量。下面的小节会讨论剩下的选项。

4.2.1. 列引用

一个列可以以下面的形式被引用：

correlation.columnname

*correlation*是一个表的名字（可能带有模式名限定），或者是在`FROM`子句中为某个表定义的别名，或者是关键词`NEW`或`OLD`之一。（`NEW`和`OLD`只能出现在重写规则中，而其他关联名称可以用在任何 SQL 语句中。）如果该列名在当前查询所用的所有表中都是唯一的，则关联名称和分隔用的句点都可以省略（另见第 7 章）。

4.2.2. 位置参数

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

$*number*

例如，考虑一个函数`dept`的定义：

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

这里`$1`引用函数被调用时第一个函数参数的值。

4.2.3. 下标

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

expression[*subscript*]

或者抽取出多个相邻元素（一个“数组切片”）：

expression[*lower_subscript*:*upper_subscript*]

（这里，方括号`[ ]`表示其字面意思）。每一个下标自身是一个表达式，它必须得到一个整数值。

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

```
mytable.arraycolumn[4]
```

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

最后一个示例中的圆括号是必需的。详见第 8.10 节。

4.2.4. 字段选择

如果一个表达式得到一个复合类型（行类型）的值，那么可以抽取该行的指定字段：

expression.*fieldname*

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

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

（因此，限定列引用实际上只是字段选择语法的一种特例。）

4.2.5. 操作符调用

对于一次操作符调用，有三种可能的语法：

expressionoperatorexpression（二元中缀操作符）
operatorexpression（一元前缀操作符）
expressionoperator（一元后缀操作符）

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

OPERATOR(*schema*.*operatorname*)

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

4.2.6. 函数调用

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

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

例如，下面会计算 2 的平方根：

```
sqrt(2)
```

内置函数的列表在第 9 章中。其他函数可以由用户增加。

4.2.7. 聚合表达式

一个聚合表达式表示将聚合函数应用于查询选中的各行。
聚合函数将多个输入归约为单个输出值，例如输入的和或平均值。
聚合表达式的语法可以是以下形式之一：

```
aggregate_name (expression)
aggregate_name (ALL expression)
aggregate_name (DISTINCT expression)
aggregate_name ( * )
```

其中，*aggregate_name* 是先前定义的聚合（可以带模式名限定），而*expression* 是任何本身不包含聚合表达式的值表达式。

第一种形式的聚合表达式为所有使给定表达式产生非空值的输入行调用聚合。（实际上，是否忽略空值取决于聚合函数本身——但所有标准聚合都会忽略空值。）
第二种形式和第一种相同，因为ALL是默认选项。
第三种形式为输入行中表达式的所有非空可区分值调用聚合。
最后一种形式为每一个输入行调用一次聚合而不管值是空还是非空；
因为没有指定特定的输入值，它通常只对count()聚合函数有用。

例如，count(*)得到输入行的总数。count(f1)得到输入行中f1为非空的数量；而count(distinct f1)得到f1的非空可区分值的数量。

预定义的聚合函数在第 9.15 节中描述。其他聚合函数可以由用户增加。

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

当一个聚合表达式出现在一个子查询中（见第 4.2.9 节和第 9.16 节），聚合通常在该子查询的行上被计算。
但是如果该聚合的参数只包含外层变量则会产生一个异常：该聚合则属于最近的那个外层，
并且会在那个查询的行上被计算。
该聚合表达式从整体上则是对其所出现于的子查询的一种外层引用，
并且在那个子查询的任意一次计算中都作为一个常量。只出现在结果列表或HAVING
子句的限制适用于该聚合所属的查询层次。

4.2.8. 类型转换

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

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

CAST语法遵从 SQL，而用::的语法是PostgreSQL的历史用法。

当类型转换被应用于一种已知类型的值表达式时，它表示一次运行时类型转换。
只有在已经定义了合适的类型转换操作时，该类型转换才会成功。注意，
这与常量上的类型转换（如第 4.1.2.4 节中所示）略有不同。应用于未修饰字符串字面量的类型转换，表示为一个字面常量值赋予初始类型，
因此它对任意类型都能成功（前提是该字符串字面量的内容符合该数据类型的输入语法）。

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

还可以用类似函数调用的语法来指定类型转换：

typename (*expression*)

不过，这只对那些名称本身也可作为函数名使用的类型有效。例如，double precision 不能用这种方式，但等效的 float8 可以。另外，由于语法冲突，名称 interval、time 和 timestamp 只有在使用双引号引用时才能采用这种写法。因此，函数风格的类型转换语法会带来不一致性，通常应尽量避免。（函数风格的语法实际上只是一个函数调用。
当两种标准转换语法之一被用于执行运行时转换时，
它将在内部调用一个已注册的函数来执行转换。按照惯例，
这些转换函数与它们的输出类型同名，因此“函数风格的语法”
不过是对底层转换函数的一次直接调用。显然，
这不是一个可移植的应用程序应该依赖的东西。）

4.2.9. 标量子查询

标量子查询是一个放在圆括号中的普通 SELECT 查询，它恰好返回一行一列（关于如何编写查询，见第 7 章）。该 SELECT 查询会被执行，其返回的单个值会用在外围值表达式中。把一个返回多于一行或多于一列的查询当作标量子查询来用是错误的。
（但如果在某次具体执行中该子查询没有返回任何行，则不算错误；
其标量结果会被视为空值。）该子查询可以引用外围查询中的变量，
这些变量在该子查询的每一次求值期间都会作为常量。有关其他涉及子查询的表达式，另见第 9.16 节。

例如，下列语句会寻找每个州中最大的城市人口：

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

4.2.10. 数组构造器

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

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

数组元素的类型是成员表达式的公共类型，其确定规则与UNION或CASE构造相同（见第 10.5 节）。

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

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

SELECT ARRAY[[1,2],[3,4]];
     array
```

```
---------------
 {{1,2},{3,4}}
(1 row)
```

由于多维数组必须是长方形的，因此同一级别的内部构造器必须生成维度完全相同的子数组。

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

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

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

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

也可以根据子查询的结果构造数组。在这种形式中，数组构造器写为关键字 ARRAY 后跟一个用圆括号（而不是方括号）括起的子查询。例如：

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

子查询必须返回单列。生成的数组为子查询结果中的每一行包含一个元素，
元素类型与子查询输出列的类型相匹配。

用 ARRAY 构造出来的数组值，其下标总是从 1 开始。关于数组的更多信息，见第 8.10 节。

4.2.11. 表达式计算规则

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

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

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

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

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

注意这和一些编程语言中布尔操作符从左至右的“短路”不同。

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

当有必要强制计算顺序时，可以使用一个CASE结构（见第 9.12 节）。例如，在一个WHERE子句
中使用下面的方法尝试避免除零是不可靠的：

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

但是这是安全的：

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

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

第 5 章 数据定义

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

5.1. 表基础

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

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

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

要创建一个表，可以使用名字很贴切的 `CREATE TABLE` 命令。在这条命令中你至少要指定新表的名称、列的名称以及每列的数据类型。例如：

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

这会创建一个名为 `my_first_table` 的表，它有两列。第一列名为 `first_column`，数据类型为 `text`；第二列名为 `second_column`，类型为 `integer`。表名和列名遵循第 4.1.1 节中解释的标识符语法。类型名通常也是标识符，但有一些例外。注意，列的列表以逗号分隔并放在圆括号中。

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

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

（`numeric`类型可以存储小数部分，这对金额很常见。）

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

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

如果不再需要某个表，可以使用`DROP TABLE`命令删除它。例如：

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

尝试删除不存在的表会报错。不过，在 SQL 脚本文件中，人们常常在创建每个表之前无条件地尝试删除它，并忽略错误消息。

如果我们需要修改一个已经存在的表，请参考本章稍后的第 5.6 节。

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

5.2. 系统列

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

`oid`

行的对象标识符（对象 ID）。这是PostgreSQL自动为所有表行添加的一个序列号（除非表是使用`WITHOUT OIDS`创建的，这种情况下该列不存在）。该列的类型是`oid`（与列同名）；关于该类型的更多信息，参见第 8.11 节。

`tableoid`

包含该行的表的 OID。这个列对于从继承层次中选择的查询特别有用，因为如果没有它，很难判断一行来自哪个具体的表。可以把 `tableoid` 与 `pg_class` 的 `oid` 列连接来获得表名。

`xmin`

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

`cmin`

插入事务中的命令标识符（从0开始）。

`xmax`

删除事务的标识（事务 ID），对未删除的行版本为零。在可见的行版本中这个列也可能非零：这通常表示删除事务尚未提交，或者一次删除尝试已被回滚。

`cmax`

删除事务中的命令标识符，或者为0。

`ctid`

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

OID 是 32 位的量，从一个集群范围内的单一计数器分配。在一个大的或长期使用的数据库中，该计数器有可能回绕。因此，除非采取措施确保 OID 唯一，否则假定 OID 唯一是不好的做法。在用 OID 做行标识时，推荐的做法是为每个要使用 OID 的表在其 OID 列上创建唯一约束。绝不要假定 OID 在表之间唯一；如果需要数据库范围的标识符，请使用 `tableoid` 与行 OID 的组合。（PostgreSQL 的未来版本可能为每个表使用单独的 OID 计数器，因此要得到全局唯一的标识符必须包含 `tableoid`。）

事务 ID也是 32 位的量。在一个长期运行的数据库中，事务 ID 可能会回卷。只要采取适当的维护措施，这并不是致命问题，详见第 21 章。不过，从长期来看（超过十亿个事务）依赖事务 ID 的唯一性是不明智的。

命令标识符也是 32 位的量。这对单个事务内的 SQL 命令数造成了 2^{32}（40 亿）的硬限制。实践中这个限制不是问题——注意该限制针对的是 SQL 命令的数量，而不是处理的行数。

5.3. 默认值

可以为列指定默认值。当创建新行且某些列没有指定值时，这些列将被填入各自的默认值。数据操作命令也可以在不知道默认值是什么的情况下，显式地要求把某个列设为它的默认值。（数据操作命令的细节见第 6 章。）

如果没有显式声明默认值，空值就是默认值。这通常是合理的，因为空值可以被认为表示未知的数据。

在表定义中，默认值写在列的数据类型之后。例如：

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

默认值可以是标量表达式，每当插入默认值时它都会被求值（而不是在创建表时）。

5.4. 约束

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

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

5.4.1. 检查约束

检查约束是最通用的约束类型。它允许你指定某个列中的值必须满足一个任意的表达式。例如，要求产品价格为正，可以这样写：

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

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

你还可以给约束一个单独的名字。这能让错误消息更清晰，并且在需要更改约束时可以引用它。
语法是：

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

因此，要指定一个命名的约束，使用关键字 CONSTRAINT，后跟一个标识符，再后跟约束定义。

检查约束也可以引用多个列。假设你存储了常规价格和折扣价格，
并且你想要确保折扣价格低于常规价格。

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

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

我们说前两个约束是列约束，而第三个是表约束，因为它与列定义分开书写。
列约束也可以写成表约束，反之则不一定可行。上面的例子也可以写成

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

```
);
```

or even

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

这只是口味问题。

应当注意，如果检查表达式的求值结果为真或空值，检查约束就得到满足。
由于大多数表达式在有一个操作数为空时都会求值为空值，它们不会阻止受约束列中的空值。
要确保列不包含空值，应当使用下一节描述的非空约束。

5.4.2. 非空约束

非空约束只是指定某列不能取空值。语法示例如下：

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

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

当然，一个列可以有多个约束。只需把约束一个接一个地写下来：

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

顺序无关紧要。它并不一定决定约束被检查的顺序。

`NOT NULL` 约束有一个反面：`NULL` 约束。这并不意味着列必须为空值——那肯定毫无用处。相反，它只是定义了列可以为空值的默认行为。`NULL` 约束没有在 SQL 标准中定义，不应在可移植的应用中使用。（它被加入 PostgreSQL 只是为了与某些其他数据库系统兼容。）不过有些用户喜欢它，因为它便于在脚本文件中切换约束。例如，你可以从

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

开始，然后在需要的地方插入 `NOT` 关键字。

> **提示**
>
> 在大多数数据库设计中，多数列都应标记为非空。

5.4.3. 唯一约束

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

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

写成表约束时则是：

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

如果唯一约束引用一组列，这些列以逗号分隔列出：

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

也可以给唯一约束指定名称：

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

一般而言，当表中（至少）有两行在作为约束一部分的各对应列上的值都相等时，
唯一约束就被违反。但在这种判断中空值不被视为相等。这意味着即使存在唯一约束，
也可以存储不限数量的至少一个受约束列包含空值的行。这种行为符合 SQL 标准，但我们听说其他 SQL 数据库可能不遵循这条规则。所以在开发打算移植的应用时要小心。

5.4.4. 主键

从技术上讲，主键约束只是唯一约束和非空约束的组合。因此，
下面的两个表定义接受相同的数据：

```
CREATE TABLE products (
```

```
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);

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

主键也可以包含多于一个列，其语法和唯一约束相似：

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

主键表示一个列或一组列可以用作表中行的唯一标识符。（这是主键定义的直接结果。注意，唯一约束本身并不能提供唯一标识符，因为它不排除空值。）
这对于文档目的和客户端应用都很有用。例如，一个允许修改行值的 GUI 应用可能需要知道一个表的主键，以便能唯一地标识行。

一个表最多只能有一个主键（但它可以有多个唯一和非空约束）。
关系数据库理论要求每一个表都要有一个主键。PostgreSQL 并未强制要求这一点，但通常最好遵循它。

5.4.5. 外键

一个外键约束指定一列（或一组列）中的值必须匹配出现在另一个表中某些行的值。
我们说这维持了两个关联表之间的引用完整性。

假设你有我们已经用过多次的产品表：

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

再假设你有一个存储这些产品订单的表。我们想要确保订单表只包含实际存在的产品的订单。
于是我们在订单表中定义一个引用 products 表的外键约束：

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

现在已经不可能创建 `product_no` 条目不出现在 products 表中的订单了。

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

你也可以把上面的命令简写成

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

因为在没有列列表时，被引用表的主键被用作被引用的列。

外键也可以约束和引用一组列。与往常一样，此时它需要写成表约束的形式。
下面是一个人为的语法示例：

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

当然，被约束列的数量和类型需要与被引用列的数量和类型匹配。

一个表可以包含多个外键约束。这用来实现表之间的多对多关系。
假设你有关于产品和订单的表，
但现在你想要允许一个订单包含可能多个产品（上面的结构不允许这样做）。
你可以使用这样的表结构：

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

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

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

还要注意最后一个表中的主键与外键重叠。

我们知道外键禁止创建与任何产品无关的订单。
但如果在创建引用某个产品的订单之后该产品被删除了会怎样？SQL 也允许你指定这种情况。直观上，我们有几种选择：

- 不允许删除被引用的产品
- 同时删除订单

- 其他处理方式？

为了说明这一点，
让我们在上面的多对多关系示例上实现下面的策略：当有人想要删除一个仍被订单（经由 order_items）引用的产品时，我们不允许。如果有人删除一个订单，订单项也随之删除。

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

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

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

限制和级联删除是两个最常见的选项。RESTRICT 也可以写成 NO ACTION，而且在你什么都没有指定时它也是默认行为。对于主键被删除时外键列应当发生什么，还有另外两个选项：SET NULL 和 SET DEFAULT。注意这些并不能免除你遵守约束的义务。例如，如果某个动作指定了 SET DEFAULT 而默认值不满足外键，主键的删除将失败。

与 ON DELETE 类似，还有 ON UPDATE，它在主键被更改（更新）时被调用。可能的动作是相同的。

更多关于更新和删除数据的信息请见第 6 章。

最后，我们要提到外键必须引用是主键或构成唯一约束的列。如果外键引用唯一约束，
关于空值如何匹配有一些额外的可能性。这些在CREATE TABLE的参考文档中有说明。

5.5. 继承

让我们创建两个表。capitals 表包含也是城市的州首府。自然，capitals 表应当继承自 cities。

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- (in ft)
);

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

在这种情况下，capitals 的一行从其父表 cities 继承所有属性（name、population 和 altitude）。属性 name 的类型是 text，这是 PostgreSQL 的原生类型，用于变长字符串。属性

population 的类型是 `float`，这是 PostgreSQL 的原生类型，用于双精度浮点数。州首府有一个额外的属性 state，表示它所在的州。在 PostgreSQL 中，一个表可以从零个或多个其他表继承，而查询既可以只引用一个表的所有行，也可以引用一个表及其所有后代的所有行。

> **注意**
>
> 继承层次实际上是一个有向无环图。

例如，下面的查询将查找所有位于海拔500英尺以上的城市的名称，包括州首府：

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

它将返回：

```
   name    | altitude
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
```

另一方面，下面的查询将找到所有高度超过 500 尺且不是州首府的城市：

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

   name    | altitude
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
```

这里cities之前的 “ONLY” 指示查询只应作用于cities本身，
而不作用于继承层次中位于cities之下的表。很多我们已经讨论过的命令 -- `SELECT`、`UPDATE` 和`DELETE` -- 都支持这种 “ONLY” 记法。

在某些情况下，你可能想知道某个特定行来自哪个表。每个表中有一个叫 `TABLEOID` 的系统列，它可以告诉你起源的表：

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

它返回：

```
 tableoid |   name    | altitude
----------+-----------+----------
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

（如果你试图重现这个例子，你可能会得到不同的数字 OID。）通过与 `pg_class` 做连接，你可以看到实际的表名：

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 and c.tableoid = p.oid;
```

它返回：

```
 relname  |   name    | altitude
----------+-----------+----------
 cities   | Las Vegas |     2174
 cities   | Mariposa  |     1953
 capitals | Madison   |      845
```

已弃用

在以前版本的 PostgreSQL 中，默认行为是不在查询中包含子表。这被发现容易出错，而且也违反 SQL99 标准。在旧语法下，要得到子表，你可以在表名后面附加 `*`。例如

```
SELECT * from cities*;
```

你仍然可以通过附加 `*` 显式指定扫描子表，也可以通过写“ONLY”显式指定不扫描子表。但从 7.1 版开始，不带修饰的表名的默认行为是也扫描其子表，而以前默认不这样做。要得到旧的默认行为，把配置选项 `SQL_Inheritance` 设为 off，例如，

```
SET SQL_Inheritance TO OFF;
```

or add a line in your `postgresql.conf` file.

继承特性的一个限制是索引（包括唯一约束）和外键约束只适用于单个表，
不适用于其继承子表。因此在上面的例子中，指定另一个表的列 `REFERENCES cities(name)` 将允许另一个表包含城市名但不能包含首府名。这个缺陷可能在将来的某个版本中修复。

5.6. 修改表

当你创建了一个表后发现犯了错误，或者应用的需求变了，可以删除该表再重新创建。
但如果表中已经装满数据，或者表被其他数据库对象（例如外键约束）引用，
这就不是一个方便的选择。因此 PostgreSQL 提供了一族命令来修改现有的表。

你可以

- 增加列
- 移除列
- 增加约束
- 移除约束
- 修改默认值
- 重命名列
- 重命名表

All these actions are performed using the `ALTER TABLE` command.

5.6.1. 增加列

要添加一个列，使用这条命令：

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

新列在表的现有行中最初将填充为空值。

你也可以同时在该列上定义约束，使用通常的语法：

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

新列不能有非空约束，因为该列最初必须包含空值。但你可以稍后添加非空约束。此外，你不能在新列上定义默认值。按照 SQL 标准，这将必须用默认值填充现有行中的新列，这尚未实现。但你可以稍后调整列默认值。

5.6.2. 移除列

要删除一个列，使用这条命令：

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

5.6.3. 增加约束

添加约束时使用表约束语法。例如：

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

非空约束不能写成表约束，添加时应使用以下语法：

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

系统会立即检查约束，因此只有表中的数据满足约束，才能将其添加。

5.6.4. 移除约束

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

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

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

除非空约束之外，所有类型的约束都可以用相同方式删除。要删除非空约束，使用：

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

（请记住，非空约束没有名称。）

5.6.5. 更改默认值

要为列设置新的默认值，使用像这样的命令：

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

要删除任何默认值，使用

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

这等价于把默认值设为空值，至少在 PostgreSQL 中是这样。因此，删除一个从未定义过的默认值并不是错误，因为默认值隐式地就是空值。

5.6.6. 重命名列

要重命名一个列：

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

5.6.7. 重命名表

要重命名一个表：

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

5.7. 权限

当你创建一个数据库对象时，你就成为它的所有者。默认情况下，
只有对象的所有者才能对对象做任何操作。为了让其他用户能够使用它，必须授予权限。
（也有一些用户拥有超级用户权限。这些用户总是可以访问任何对象。）

> **注意**
>
> 要更改表、索引、序列或视图的所有者，请使用ALTER TABLE命令。

有多种不同的权限：SELECT、INSERT、UPDATE、DELETE、RULE、REFERENCES、TRIGGER、CREATE、TEMPORARY、EXECUTE、USAGE 和 ALL PRIVILEGES。关于 PostgreSQL 支持的各种权限类型的完整信息，请参阅GRANT参考页。后面的各节和各章还会向你展示这些权限是如何使用的。

修改或销毁对象的权利始终只属于该对象的拥有者。

要分配权限，使用 GRANT 命令。因此，如果 joe 是一个已存在的用户，而 accounts 是一个已存在的表，更新该表的权限可以这样授予：

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

执行这条命令的用户必须是表的拥有者。要把权限授予一个组，使用

```
GRANT SELECT ON accounts TO GROUP staff;
```

特殊的“用户”名 PUBLIC 可用来把权限授予系统中的每个用户。在具体权限的位置写 ALL 表示将授予所有权限。

要收回权限，使用名字同样贴切的 REVOKE 命令：

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

表拥有者的特殊权限（即执行 DROP、GRANT、REVOKE 等的权利）总是隐含在拥有者身份中，不能被授予或收回。但表拥有者可以选择收回自己的普通权限，
例如让自己和别人都只能对表进行只读访问。

5.8. 模式

一个PostgreSQL数据库集簇中包含一个或更多命名的数据库。用户和用户组被整个集簇共享，
但数据库之间不共享其他任何数据。
任何一条到服务器的客户端连接都只能访问单个数据库中的数据，
也就是连接请求中指定的那个数据库。

> **注意**
>
> 一个集簇的用户并不必拥有访问集簇中每一个数据库的权限。
> 用户名的共享意味着不可能在同一个集簇中出现重名的不同用户，
> 例如两个数据库中都有叫joe的用户。但系统可以被配置为只允许joe访问某些数据库。

一个数据库包含一个或多个命名的模式，模式中又包含表。模式还包含其他类型的命名对象，
包括数据类型、函数和操作符。在同一个模式中，同一类型的两个对象不能有相同的名称。
此外，表、序列、索引、视图、物化视图和外部表共享同一个名字空间，
因此例如当它们位于同一模式中时，索引和表就必须具有不同的名称。
相同的对象名可以在不同模式中重复使用而不发生冲突；例如，schema1和myschema
都可以包含名为mytable的表。与数据库不同，模式并不是被严格隔离的：只要拥有相应权限，
用户就可以访问其所连接数据库中任意模式里的对象。

下面是一些使用模式的原因：

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

模式类似于操作系统层的目录，但是模式不能嵌套。

5.8.1. 创建模式

要创建一个独立的模式，使用命令 CREATE SCHEMA。给模式起一个你选择的名字。例如：

```
CREATE SCHEMA myschema;
```

要在模式中创建或访问对象，写一个限定名，它由模式名和表名组成并以点分隔：

schema.table

实际上，更一般的语法

database.schema.table

也可以使用，但目前这只是形式上与 SQL 标准兼容；如果你写了数据库名，它必须与你所连接的数据库相同。

因此要在新模式中创建一个表，使用

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

这在需要表名的任何地方都有效，包括后面各章讨论的表修改命令和数据访问命令。

要删除一个为空的模式（其中的所有对象已经被删除），可用：

```
DROP SCHEMA myschema;
```

要删除一个模式以及其中包含的所有对象，可用：

```
DROP SCHEMA myschema CASCADE;
```

有关于此的更一般的机制请参见第 5.10 节。

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

```
CREATE SCHEMA schemaname AUTHORIZATION username;
```

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

以`pg_`开头的模式名被保留用于系统目的，所以不能被用户所创建。

5.8.2. 公共模式

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

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

以及：

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

5.8.3. 模式搜索路径

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

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

要显示当前搜索路径，可使用下面的命令：

```
SHOW search_path;
```

在默认设置下这将返回：

```
 search_path
--------------
```

```
 $user,public
```

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

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

要把新模式放在搜索路径中，我们可以使用：

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

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

```
DROP TABLE mytable;
```

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

我们也可以这样写：

```
SET search_path TO myschema;
```

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

访问模式搜索路径的其他方法另见第 9.13 节。

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

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

这是为了避免句法歧义。例如：

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

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

5.8.4. 模式和权限

默认情况下，用户无法访问他们不拥有的模式中的任何对象。要允许这样做，
模式的所有者必须授予该模式上的`USAGE`权限。要允许用户使用模式中的对象，
可能需要授予其他权限，适用于该对象。

也可以允许用户在别人的模式中创建对象。为此，需要授予该模式上的 `CREATE` 权限。注意默认情况下，每个人都拥有模式 `public`。这使得所有能连接到某个给定数据库的用户都可以在该数据库的 `public` 模式中创建对象。如果你不希望这样，可以收回该权限：

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

（第一个“public”是模式名，第二个“public”指“每个用户”。前者是标识符，
后者是保留字，因此大小写不同；回想第 4.1.1 节中的指导原则。）

5.8.5. 系统目录模式

除public和用户创建的模式之外，每一个数据库还包括一个pg_catalog模式，它包含了系统表和所有内置的数据类型、函数以及操作符。pg_catalog总是搜索路径的一个有效部分。如果没有在路径中显式地包括该模式，它将在路径中的模式之前被搜索。这保证了内置的名称总是能被找到。然而，如果我们希望用用户定义的名称重载内置的名称，可以显式的将pg_catalog放在搜索路径的末尾。

在PostgreSQL 7.3 之前的版本中，以pg_开头的表名是被保留的。现在不再如此：如果你愿意，可以在任何非系统模式中创建这样名字的表。不过，最好还是继续避免使用这样的名称，以确保将来某个版本定义出与你的表同名的系统表时，你不会遇到冲突。系统表会继续遵循以pg_开头的约定，因此只要用户避免使用pg_前缀，它们就不会与未经限定的用户表名发生冲突。

5.8.6. 使用方式

模式可以用多种方式组织你的数据。有几种被推荐且很容易被支持的使用模式：默认配置：

- 如果你不创建任何模式，那么所有用户都将隐式地访问 public 模式。这模拟了根本没有模式可用的情况。这种设置主要推荐用于数据库中只有单个用户或少数几个相互合作的用户的场景。这种设置也允许从不支持模式的环境平滑过渡。
- 你可以为每个用户创建一个与其同名的模式。请记住，默认搜索路径以$user开头，它会解析为用户名。因此，如果每个用户都有单独的模式，他们默认访问的就是自己的模式。

  如果采用这种设置，你可能还想撤销对 public 模式的访问（或者干脆删除它），使用户真正被限制在他们自己的模式中。
- 要安装共享的应用（供所有人使用的表、第三方提供的额外函数等），请把它们放到单独的模式中。记住授予适当的权限以允许其他用户访问它们。用户之后可以通过用模式名限定名称来引用这些额外的对象，也可以按自己的选择把这些额外的模式放进自己的路径。

5.8.7. 可移植性

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

同样，SQL 标准中也没有public模式这一概念。为了尽可能符合标准，你不应使用（甚至可以移除）public模式。

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

5.9. 其他数据库对象

表是关系数据库结构中的核心对象，因为它们承载着数据。但它们并不是数据库中唯一存在的对象。还可以创建许多其他种类的对象，以便让数据的使用和管理更加高效或方便。本章不会讨论这些对象，但这里给出一个列表，让你了解有哪些可能性：

- 视图
- 函数、操作符、数据类型、域
- 触发器和重写规则

这些主题的详细信息请见第 V 部分。

5.10. 依赖跟踪

当我们创建一个涉及到很多具有外键约束、视图、触发器、函数等的表的复杂数据库结构时，我们隐式地创建了一张对象之间的依赖关系网。例如，具有一个外键约束的表依赖于它所引用的表。

为了确保整个数据库结构的完整性，PostgreSQL 确保你不能删除仍被其他对象依赖的对象。例如，试图删除我们在第 5.4.5 节中考虑过的 products 表（orders 表依赖它），会得到像这样的错误消息：

```
DROP TABLE products;

NOTICE:  constraint $1 on table orders depends on table products
ERROR:  cannot drop table products because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

错误消息包含一个有用的提示：如果你不想逐个删除所有依赖对象，可以运行

```
DROP TABLE products CASCADE;
```

，所有依赖对象都会被删除。在这种情况下，它不会删除 orders 表，只会删除外键约束。(If you want to check what `DROP ... CASCADE` will do, run `DROP` without `CASCADE` and read the `NOTICE` messages.)

PostgreSQL中所有 DROP 命令都支持指定`CASCADE`。当然，可能出现的依赖关系形态会随着对象类型不同而变化。你也可以写`RESTRICT`来代替`CASCADE`，从而得到默认行为，也就是阻止删除任何被其他对象依赖的对象。

> **注意**
>
> 根据 SQL 标准，必须指定 `RESTRICT` 或 `CASCADE` 之一。实际上没有数据库系统按这种方式实现，但默认行为是 `RESTRICT` 还是 `CASCADE` 因系统而异。

> **注意**
>
> PostgreSQL 7.3 之前版本的外键约束依赖和 serial 列依赖在升级过程中不会被维护或创建。所有其他依赖类型在升级期间都会被正确创建。

第 6 章 数据操纵

上一章讨论了如何创建表以及其他用来保存数据的结构。现在该向表中填充数据了。
本章介绍如何插入、更新和删除表中的数据。
我们还介绍在特定事件发生时实现自动数据更改的方法：触发器和重写规则。
下一章将最终解释如何把你久违的数据从数据库中提取出来。

6.1. 插入数据

表创建时不包含任何数据。在数据库能派上用场之前，首先要做的事情就是插入数据。
从概念上讲，数据是一次一行地插入的。当然你也可以一次插入多行，
但没有办法一次插入不到一行的数据。即使你只知道某些列的值，也必须创建一个完整的行。

要创建一个新行，使用`INSERT`命令。该命令需要表名以及表中每一列的一个值。例如，考虑
第 5 章中的 products 表：

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

插入一行的一个示例命令是：

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

数据值按各列在表中出现的顺序列出，用逗号分隔。通常，数据值是字面量（常量），
但也允许标量表达式。

上面的语法有一个缺点：你需要知道表中各列的顺序。为避免这一点，你也可以显式列出列。
例如，下面两条命令与上面的命令效果相同：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.
99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99,
 1);
```

许多用户认为始终列出列名是一种好习惯。

如果你没有所有列的值，可以省略其中一些。这种情况下，被省略的列将用它们的默认值填充。
例如，

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

第二种形式是PostgreSQL的扩展。它从左边开始用给定的值填充各列，其余列将使用默认值。

为了清楚起见，你也可以显式地请求默认值，可以对单个列，也可以对整行：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese',
 DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

> **提示**
>
> 要做“批量装载”，也就是插入大量数据，可以看看COPY命令。它不如INSERT 命令灵活，但效率更高。

6.2. 更新数据

对数据库中已有数据的修改称为更新。你可以更新单独的行、
表中的所有行或所有行的一个子集。每一列都可以单独更新；其他列不受影响。

要执行一次更新，你需要三条信息：

1. 要更新的表和列的名称，
2. 列的新值，
3. 要更新哪些行。

回想第 5 章中所说的，SQL 一般不为行提供唯一的标识符。因此不一定能直接指定要更新哪一行。相反，你指定的是行必须满足什么条件才会被更新。
只有当表中有主键时（无论你是否声明了它），
你才能通过选择一个与主键匹配的条件来可靠地定位单独的行。
图形化的数据库访问工具正是依靠这一点来让你逐行更新的。

例如，这条命令把所有价格为 5 的产品的价格更新为 10：

```
UPDATE products SET price = 10 WHERE price = 5;
```

这可能导致零行、一行或多行被更新。尝试一次不匹配任何行的更新并不是错误。

让我们详细看看这条命令：首先是关键字UPDATE，后跟表名。与通常一样，
表名可以是模式限定的，否则将在路径中查找。接下来是关键字SET，后跟列名、
一个等号和列的新值。新的列值可以是任何标量表达式，而不仅仅是常量。例如，
如果你想把所有产品的价格提高 10%，可以使用：

```
UPDATE products SET price = price * 1.10;
```

如你所见，新值的表达式也可以引用旧值。我们还省略了 WHERE子句。如果省略它，就意味着表中的所有行都被更新。如果给出它，则只有满足WHERE 之后条件的那些行被更新。注意，SET子句中的等号是赋值，而WHERE子句中的等号是比较，但这不会产生任何歧义。当然，
条件不必是相等测试。还有许多其他操作符可用（见第 9 章）。但表达式必须求值为布尔结果。

你也可以在一条UPDATE命令中更新多列，方法是在SET子句中列出多个赋值。例如：

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. 删除数据

到目前为止我们已经解释了如何向表中添加数据以及如何更改数据。
剩下的是讨论如何删除不再需要的数据。正如添加数据只能整行进行一样，
你也只能从表中删除整行。上一节我们讨论过，SQL 不提供直接定位单独行的办法。因此，删除行只能通过指定要删除的行必须满足的条件来完成。如果表中有主键，
那么你可以指定精确的行。但你也可以删除匹配某个条件的成组的行，
或者一次删除表中的所有行。

你使用`DELETE`命令删除行；其语法与 `UPDATE`命令非常相似。例如，要从 products 表中删除所有价格为 10 的行，使用

```
DELETE FROM products WHERE price = 10;
```

如果你直接写

```
DELETE FROM products;
```

那么表中的所有行都将被删除！程序员当心。

第 7 章 查询

前面的章节解释了如何创建表、如何用数据填充它们，以及如何操纵这些数据。
现在我们终于可以讨论如何从数据库中检索数据了。

7.1. 概述

从数据库检索数据的过程或用于检索数据的命令称为查询。在 SQL 中，SELECT命令用于指定查询。SELECT命令的一般语法为：

```
SELECT select_list FROM table_expression [sort_specification]
```

以下各节将详细描述选择列表、表表达式和排序说明。

一种简单查询的形式如下：

```
SELECT * FROM table1;
```

假设存在一个名为table1的表，则这条命令会从table1中检索所有行以及所有列。
（具体的检索方式取决于客户端应用。例如，psql程序会在屏幕上显示一个 ASCII 形式的表格，而客户端库则会提供从查询结果中提取单个值的函数。）选择列表说明*
表示表表达式所提供的全部列。选择列表也可以只选择部分可用列，或者利用这些列进行计算。
例如，如果table1有名为a、b和c的列（还可能有其他列），那么可以写出下面的查询：

```
SELECT a, b + c FROM table1;
```

（假设b和c都是数值数据类型）。有关更多细节请参见第 7.3 节。

FROM table1是一种简单的表表达式：它只读取一个表。通常，表表达式可以是由基本表、连接和子查询组成的复杂结构。不过，你也可以完全省略表表达式，把SELECT
命令当成计算器来使用：

```
SELECT 3 * 4;
```

如果选择列表中的表达式会返回变化的结果，这种用法就更有用了。例如，
你可以这样调用函数：

```
SELECT random();
```

7.2. 表表达式

表表达式用于计算出一个表。表表达式包含一个FROM子句，后面可以根据需要跟上WHERE、GROUP BY和HAVING子句。最简单的表表达式只是引用磁盘上的一个表，即所谓的基本表；但也可以使用更复杂的表达式，以多种方式修改或组合基本表。

表表达式中可选的WHERE、GROUP BY和HAVING子句指定了一个连续转换的流水线，这些转换作用于由FROM子句派生出的表。所有这些转换都会生成一个虚拟表，
该表提供的各行会传递给选择列表，以计算查询的输出行。

7.2.1. FROM 子句

FROM子句从逗号分隔的表引用列表所给出的一个或多个其他表推导出一个表。

```
FROM table_reference [, table_reference [, ...]]
```

表引用可以是表名（可能带模式限定），也可以是推导表，例如子查询、表连接，
或它们的复杂组合。如果FROM子句中列出了多个表引用，这些表会进行交叉连接（见下文），
形成一个中间虚拟表，随后该虚拟表可以被WHERE、GROUP BY和HAVING子句变换，并最终成为
整个表表达式的结果。

当表引用命名的是表继承层次中的父表时，除非在表名前加上关键字ONLY，
否则该表引用不仅会产生该表中的行，还会产生其所有子表后代的行。不过，
这种引用只会产生所命名表中出现的列——子表中新增的列会被忽略。

7.2.1.1. 连接表

一个连接表是根据特定的连接类型的规则从两个其它表（真实表或推导表）中派生的表。
目前可以使用内连接、外连接和交叉连接。

连接类型

Cross join

```
T1 CROSS JOIN T2
```

对来自于*T1*和*T2*的行的每一种组合，派生表将包含这样一行：它由所有*T1*
中的列后面跟着所有*T2*中的列构成。如果两个表分别有 N 和 M 行，连接表将有 N * M 行。

FROM *T1* CROSS JOIN *T2*等效于FROM *T1*, *T2*。它也等效于 FROM *T1* INNER JOIN *T2* ON TRUE（见下文）。

Qualified joins

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING (
 join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

INNER和OUTER在所有形式中都是可选的。INNER是默认值；LEFT、RIGHT和FULL
表示外连接。

连接条件在ON或USING子句中指定，或者用关键字NATURAL隐含地指定。
连接条件决定来自两个源表中的哪些行被认为是“匹配”的，这些我们将在下文详细解释。

ON子句是最一般的连接条件形式：它接收一个和WHERE子句里用的一样的布尔值表达式。
如果两个分别来自*T1*和*T2*的行在ON表达式上运算的结果为真，那么它们就算是匹配的行。

USING是个缩写符号：它接受一个逗号分隔的列名列表，
这些列名是被连接的两个表必须共有的，
并为其中每一对列构造一个指定等值比较的连接条件。此外，JOIN USING的输出为每一对相等的输入列产生一个列，后面跟上各个表的所有其余列。因此，USING (a, b, c)等效于ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)，不同之处在于：如果使用ON，结果中会有两个分别名为a、b和c的列，而使用USING时每列只有一个。

最后，NATURAL是USING的一种缩写形式：它会形成一个USING列表，
其中包含的正是两个输入表中共同出现的那些列名。和USING一样，
这些列在输出表中只出现一次。

限定连接有以下几种类型：

`INNER JOIN`

对于 T1 的每一行 R1，生成的连接表都有一行对应 T2 中的每一个满足和 R1 的连接条件的行。

`LEFT OUTER JOIN`

首先，执行一次内连接。然后，为 T1 中每一个无法在连接条件上匹配 T2 里任何一行的行返回一个连接行，该连接行中 T2 的列用空值补齐。因此，生成的连接表里为来自 T1 的每一行都至少包含一行。

`RIGHT OUTER JOIN`

首先，执行一次内连接。然后，为 T2 中每一个无法在连接条件上匹配 T1 里任何一行的行返回一个连接行，该连接行中 T1 的列用空值补齐。这是左连接的反面：生成的连接表里为来自 T2 的每一行都无条件地包含一行。

`FULL OUTER JOIN`

首先，执行一次内连接。然后，为 T1 中每一个无法在连接条件上匹配 T2 里任何一行的行返回一个连接行，该连接行中 T2 的列用空值补齐。同样，为 T2 中每一个无法在连接条件上匹配 T1 里任何一行的行返回一个连接行，该连接行中 T1 的列用空值补齐。

所有类型的连接都可以被链在一起或者嵌套：*T1*和*T2*都可以是连接表。在`JOIN`子句周围可以使用圆括号来控制连接顺序。如果不使用圆括号，`JOIN`子句会从左至右嵌套。

综合起来看，假设我们有表`t1`

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

和`t2`

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

那么各种连接会得到下面的结果：

```
=>SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)
```

```
=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

用ON指定的连接条件也可以包含与连接不直接相关的条件。这种功能可能对某些查询很有用，但是需要仔细想清楚。例如：

```
=>SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value =
 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

7.2.1.2. 表和列别名

你可以给表以及复杂的表引用指定一个临时名字，以便在后续处理中引用该派生表。这被称为表别名。

要创建一个表别名，我们可以写：

FROM *table_reference* AS *alias*

或者

FROM *table_reference alias*

AS关键字只是可选的语法噪音。*alias*可以是任意标识符。

表别名的典型应用是给长表名赋予比较短的标识符，好让连接子句更易读。例如：

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_
name a ON s.id = a.num;
```

对于当前查询而言，别名成为该表引用的新名称——不再能通过原名称引用该表。因此

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;
```

不是合法的 SQL 语法。实际会发生的是（这是PostgreSQL对标准的一个扩展）：一个隐式的表引用会被加入FROM子句，因此该查询会被当作像下面这样写的一样来处理

```
SELECT * FROM my_table AS m, my_table AS my_table WHERE my_table.a >
 5;
```

这将产生一个交叉连接，而这通常不是你想要的。

表别名主要用于简化符号，但是当把一个表连接到它自身时必须使用别名，例如：

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
```

此外，当表引用是子查询时也必须使用别名（参见第 7.2.1.3 节）。

圆括弧用于解决歧义。下面的语句会把别名b赋给连接的结果，这与前一个例子不同：

```
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

另外一种表别名形式还会给表的列赋予临时名字：

FROM *table_reference* [AS] *alias* (*column1* [, *column2* [, ...]])

如果指定的列别名比表里实际的列少，那么剩下的列就没有被重命名。
这种语法对于自连接或子查询特别有用。

如果用这些形式中的任何一种给一个JOIN子句的输出附加了一个别名，那么该别名就在JOIN的作用下隐去了其原始的名字。例如：

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

是合法 SQL，但是：

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

是不合法的：表别名a在别名c外面是看不到的。

7.2.1.3. 子查询

指定派生表的子查询必须用圆括号括起来，并且必须赋予一个表别名。（参见第 7.2.1.2 节。）例如：

```
FROM (SELECT * FROM table1) AS alias_name
```

这个示例等效于FROM table1 AS alias_name。更有趣的情况是在子查询里面有分组或聚合的时候，这时子查询不能被简化为一个简单的连接。

7.2.1.4. 表函数

表函数是那些生成行集合的函数，这些行可以由基础数据类型（标量类型）组成，
也可以由复合数据类型（表行）组成。它们在查询的FROM子句中的用法类似于表、
视图或子查询。表函数返回的列可以像表、视图或子查询的列一样，以相同的方式出现在SELECT、JOIN或WHERE子句中。

如果一个表函数返回基础数据类型，唯一的结果列名与该函数名相同。
如果该函数返回一个复合类型，结果列会得到与该类型的各个属性相同的名称。

表函数可以在FROM子句中被赋予别名，但也可以不使用别名。如果一个函数在FROM子句中使用时没有别名，该函数名将被用作结果表的表名。

一些例子：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS '
    SELECT * FROM foo WHERE fooid = $1;
' LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (select foosubid from getfoo(foo.fooid) z
                           where z.fooid = foo.fooid);

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);
SELECT * FROM vw_getfoo;
```

有时候，定义一个能够根据调用方式返回不同列集合的表函数很有用。为支持这一点，
表函数可以被声明为返回伪类型record。当此类函数在查询中使用时，

必须在查询本身中指定预期的行结构，这样系统才能知道如何分析和规划该查询。
考虑下面的示例：

```
SELECT *
    FROM dblink('dbname=mydb', 'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

dblink函数执行一个远程查询（见contrib/dblink）。它被声明为返回record，
因为它可能用于任意类型的查询。实际的列集必须在调用它的查询中指定，这样分析器才知道像*
这样的写法应当扩展成什么。

7.2.2. WHERE 子句

WHERE子句的语法是：

WHERE *search_condition*

其中，*search_condition*是任意的值表达式（如第 4.2 节中所定义），它返回一个boolean
类型的值。

在完成FROM子句的处理之后，生成的虚拟表中的每一行都会根据搜索条件进行检查。
如果条件结果为真，该行就会保留在输出表中；否则（也就是说，如果结果为假或空）
就会被丢弃。搜索条件通常至少会引用FROM子句生成的表中的某一列；这并非必须，
但若不引用这些列，WHERE子句通常就没什么用了。

> **注意**
>
> 在JOIN语法实现之前，内连接的连接条件必须放在WHERE子句中。例如，
> 这些表表达式是等效的：
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```
>
> 和：
>
> ```
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
> ```
>
> 或者可能还有：
>
> ```
> FROM a NATURAL JOIN b WHERE b.val > 5
> ```
>
> 选择其中哪一种主要只是风格问题。FROM子句中的JOIN语法移植到其他 SQL 数据库管理
> 系统时可能没有那么容易。对于外连接在任何情况下都没有选择的余地：它们必须在FROM
> 子句中完成。外连接的ON或USING子句与WHERE条件并不等价，
> 因为它除了会在最终结果中移除行之外，还会为不匹配的输入行增加结果行。

下面是一些WHERE子句的例子：

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 +
 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 =
 fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

fdt是由FROM子句推导得到的表。不满足WHERE子句搜索条件的行会从fdt中消除。注意，这里将标量子查询用作值表达式。与其他查询一样，子查询也可以使用复杂的表表达式。还要注意子查询中对fdt的引用方式。将c1限定为fdt.c1仅在c1也是子查询推导输入表中的列名时才有必要。不过，即使不必限定，限定列名仍能使含义更清晰。本例说明了外层查询的列命名作用域如何扩展到其内部查询。

7.2.3. GROUP BY 和 HAVING 子句

通过WHERE过滤之后，派生出来的输入表还可以使用GROUP BY子句进行分组，并通过HAVING子句排除某些分组行。

```
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference [, grouping_column_reference].
..
```

GROUP BY子句用于把表中所有列出列的值均相同的行分到同一组。列的列出顺序没有影响。其效果是把具有相同值的每一组行合并为一个分组行，由它代表组内的所有行。这样可以消除输出中的冗余，或者计算应用于各组的聚合。例如：

```
=>SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

在第二个查询里，我们不能写成SELECT * FROM test1 GROUP BY x，因为列y没有一个单一值可以与每个分组对应。被分组的列可以在选择列表中引用是因为它们在每个组都有单一的值。

通常，如果一个表被分了组，那么没有参与分组的列都不能被引用，除非是在聚合表达式中。一个用聚合表达式的示例是：

```
=>SELECT x, sum(y) FROM test1 GROUP BY x;
```

```
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

这里的sum是一个聚合函数，它在整个组上计算出一个单一值。
有关可用聚合函数的更多信息可以在第 9.15 节中找到。

> **提示**
>
> 不带聚合表达式的分组实际上是在计算某一列的非重复值集合。这也可以用DISTINCT子句实现（参阅第 7.3.3 节）。

这里是另外一个示例：它计算每种产品的总销售额（而不是所有产品的总销售额）：

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

在这个示例里，列product_id、p.name和p.price必须在GROUP BY子句里，因为它们都在查询的选择列表里被引用到。（根据 products 表的具体设计，name 和 price 可能完全依赖于产品 ID，因此理论上这些额外的分组并非必要，不过这一点尚未实现。）列s.units不必在GROUP BY列表里，因为它只是在一个聚合表达式（sum(...)）里使用，它代表一种产品的销售额。对于每种产品，这个查询都返回一个关于该产品所有销售额的汇总行。

在严格的 SQL 里，GROUP BY只能对源表的列进行分组，但PostgreSQL把这个扩展为也允许GROUP BY去根据选择列表中的列分组。也允许对值表达式进行分组，而不仅是简单的列名。

如果一个表已经用GROUP BY子句分了组，而你只对其中某些组感兴趣，那么就可以使用HAVING子句。它很像WHERE子句，用于从结果中删除某些组。其语法是：

SELECT *select_list* FROM ... [WHERE ...] GROUP BY ... HAVING *boolean_expression*

在HAVING
子句中的表达式可以引用分组的表达式和未分组的表达式（后者必须涉及一个聚合函数）。

示例：

```
=>SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
```

```
 b |   5
(2 rows)
```

再次，一个更现实的示例：

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS
 profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

在上面的示例里，`WHERE`
子句通过一个未分组的列来选择数据行（该表达式仅对最近四周内发生的销售为真），而`HAVING`子句则把输出限制为总销售收入超过 5000 的组。请注意，聚合表达式不必在查询的所有部分都完全相同。

7.3. 选择列表

如前面几节所示，`SELECT`命令中的表表达式会构造出一个中间虚拟表，
其中可能涉及组合表和视图、消除行、分组等操作。这个表随后交由选择列表处理。
选择列表决定中间表中的哪些列会实际输出。

7.3.1. 选择列表项

最简单的选择列表是`*`，它会输出表表达式生成的所有列。否则，
选择列表就是一个逗号分隔的值表达式列表（定义见第 4.2 节）。例如，它可以是一个列名列表：

```
SELECT a, b, c FROM ...
```

列名`a`、`b`和`c`要么是`FROM`子句中所引用表的实际列名，要么是像第 7.2.1.2 节所述为它们指定的别名。选择列表中可用的名字空间与`WHERE`子句相同；如果使用了分组，则与`HAVING`
子句相同。

如果超过一个表有同样的列名，那么你还必须给出表名字，如：

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

在使用多个表时，要求一个特定表的所有列也是有用的：

```
SELECT tbl1.*, tbl2.a FROM ...
```

（另请参阅第 7.2.2 节。）

如果在选择列表中使用任意值表达式，那么从概念上说，它会在返回表中增加一个新的虚拟列。
该值表达式会对每个结果行计算一次，并用该行中的值替换其中的列引用。不过，
选择列表中的表达式并不一定要引用`FROM`子句表表达式中的列；例如，
它也可以是任意常量算术表达式。

7.3.2. 列标签

选择列表中的项可以被赋予名字，以供后续处理使用。这里的“后续处理”
指的是可选的排序说明和客户端应用（例如用于显示的列标题）。例如：

```
SELECT a AS value, b + c AS sum FROM ...
```

如果没有使用AS指定输出列名，系统就会分配一个默认列名。对于简单的列引用，它就是被引用列的名字；对于函数调用，它就是函数的名字；对于复杂表达式，系统则会生成一个通用名称。

> **注意**
>
> 输出列的命名和在FROM子句里的命名是不一样的（参阅第 7.2.1.2 节）。它实际上允许你对同一个列命名两次，但是在选择列表中分配的名字是要传递下去的名字。

7.3.3. DISTINCT

在处理完选择列表之后，结果表还可以选择性地删除重复行。我们可以直接在SELECT后面写上DISTINCT关键字来指定：

```
SELECT DISTINCT select_list ...
```

（如果不用DISTINCT，则可以用ALL关键字来显式指定保留所有行这一默认行为。）

显然，如果两行至少在一个列值上不同，就认为它们是不同的。空值在这种比较中被认为是相等的。

另外，我们还可以用任意表达式来判断什么行可以被认为是可区分的：

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

这里的*expression*是一个对所有行求值的任意值表达式。如果某一组行在这些表达式上的值都相等，那么它们就会被视为重复行，因此只保留该组中的第一行。请注意，某一组里的“第一行”是不可预测的，除非查询在足够多的列上进行了排序，以保证到达DISTINCT过滤器的行顺序是唯一的。（DISTINCT ON的处理发生在ORDER BY排序之后。）

DISTINCT ON子句不是 SQL 标准的一部分，有时会被认为风格不佳，因为它的结果可能具有不确定性。通过审慎使用GROUP BY以及FROM中的子查询，可以避免使用这种结构，但它往往是最方便的替代方案。

7.4. 组合查询

两个查询的结果可以用集合操作并、交、差进行组合。语法是

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

*query1*和*query2*都是可以使用以上所有特性的查询。集合操作也可以嵌套和链式使用，例如

```
query1 UNION query2 UNION query3
```

它会被当作：

```
(query1 UNION query2) UNION query3
```

来执行。

UNION有效地把*query2*的结果附加到*query1*
的结果上（不过我们不能保证这就是这些行实际被返回的顺序）。此外，
它将删除结果中所有重复的行，就像DISTINCT做的那样，除非你使用了UNION ALL。

INTERSECT返回那些同时存在于*query1*和*query2*的结果中的行，除非声明了INTERSECT ALL，否则所有重复行都被消除。

EXCEPT返回所有在*query1*结果中但不在*query2*结果中的行。
（这有时被称为两个查询的差。）同样，除非使用了EXCEPT ALL，否则重复行都会被消除。

为了计算两个查询的并、交、差，这两个查询必须是“并操作兼容的”，
也就意味着它们都返回同样数量的列，并且对应的列有兼容的数据类型，如第 10.5 节中描述的那样。

7.5. 行排序

当一个查询生成输出表之后（也就是处理完选择列表之后），还可以选择对其排序。
如果没有选择排序，行将按随机顺序返回。
这时的实际顺序取决于扫描和连接计划类型以及磁盘上的物理顺序，但绝不能依赖这些因素。
只有显式选择排序步骤，才能保证特定的输出顺序。

ORDER BY子句指定排序顺序：

```
SELECT select_list
    FROM table_expression
    ORDER BY column1 [ASC | DESC] [, column2 [ASC | DESC] ...]
```

*column1*等指的是选择列表的列。它们既可以是列的输出名（见第 7.3.2 节），也可以是列的编号。一些例子：

```
SELECT a, b FROM table1 ORDER BY a;
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, sum(b) FROM table1 GROUP BY a ORDER BY 1;
```

作为对 SQL 标准的扩展，PostgreSQL还允许按任意表达式排序：

```
SELECT a, b FROM table1 ORDER BY a + b;
```

在选择列表中被重命名的FROM子句列名的引用也是允许的：

```
SELECT a AS b FROM table1 ORDER BY a;
```

但这些扩展在涉及UNION、INTERSECT或EXCEPT的查询中不起作用，并且不能移植到其他 SQL 数据库。

每个列说明后面都可以选择性地写上ASC或DESC关键字，以指定升序或降序排序方向。ASC 是默认值。升序会把较小的值排在前面，而“较小”是由<操作符定义的。类似地，降序则由> 操作符定义。[1]

如果指定了多个排序列，
较后的项将用于对那些在前面排序列所决定的顺序下相等的行继续排序。

[1] 事实上，PostgreSQL会使用该列数据类型的默认 B-树操作符类来决定ASC和DESC的排序顺序。按惯例，数据类型会被设置成由<和> 操作符来对应这种排序顺序，不过用户定义数据类型的设计者也可以选择不同的做法。

7.6. LIMIT 和 OFFSET

LIMIT和OFFSET允许仅检索查询其余部分所生成行的一部分：

```
SELECT select_list
    FROM table_expression
    [LIMIT { number | ALL }] [OFFSET number]
```

如果给出了一个限制计数，那么会返回数量不超过该限制的行（但可能更少些，因为查询本身可能生成的行数就比较少）。LIMIT ALL的效果和省略LIMIT子句一样。

OFFSET说明在开始返回结果行之前要跳过多少行。OFFSET 0的效果与省略OFFSET子句相同。如果OFFSET和LIMIT都出现，那么在开始返回LIMIT行之前，会先跳过OFFSET行。

如果使用LIMIT，那么用一个ORDER BY子句把结果行约束成一个唯一的顺序是很重要的。否则你就会拿到一个不可预料的该查询的行的子集。你要的可能是第十到第二十行，但以什么顺序的第十到第二十？除非你指定了ORDER BY，否则顺序是不知道的。

查询优化器在生成查询计划时会考虑LIMIT，因此如果你给定LIMIT和OFFSET，那么你很可能收到不同的规划（产生不同的行顺序）。因此，使用不同的LIMIT/OFFSET值选择查询结果的不同子集将生成不一致的结果，除非你用ORDER BY强制一个可预测的顺序。这并非 bug，这是一个很自然的结果，因为 SQL 没有许诺把查询的结果按照任何特定的顺序发出，除非用了ORDER BY来约束顺序。

第 8 章 数据类型

PostgreSQL 提供了丰富的原生数据类型。用户可以使用CREATE TYPE命令向 PostgreSQL添加新类型。

表 8.1列出了所有内建的通用数据类型。“别名”列中列出的大多数替代名称，是 PostgreSQL 出于历史原因在内部使用的名称。此外还有一些内部使用或已废弃的类型可用，但这里没有列出。

表 8.1. 数据类型

名字	别名	描述
bigint	int8	有符号的8字节整数
bigserial	serial8	自动递增的8字节整数
bit		定长位串
bit varying(*n*)	varbit(*n*)	变长位串
boolean	bool	逻辑布尔值（真/假）
box		平面上的矩形框
bytea		二进制数据
character varying(*n*)	varchar(*n*)	变长字符串
character(*n*)	char(*n*)	定长字符串
cidr		IPv4或IPv6网络地址
circle		平面上的圆
date		日历日期（年、月、日）
double precision	float8	双精度浮点数
inet		IPv4或IPv6主机地址
integer	int, int4	有符号4字节整数
interval(*p*)		时间段
line		平面上的无限直线（未完全实现）
lseg		平面上的线段
macaddr		MAC 地址
money		货币数量
numeric [(*p*, *s*)]	decimal [(*p*, *s*)]	精度可选择的精确数值
path		平面上的开放和闭合几何路径
point		平面上的几何点
polygon		平面上的封闭几何路径
real	float4	单精度浮点数
smallint	int2	有符号2字节整数
serial	serial4	自动递增的4字节整数
text		变长字符串
time [(*p*)] [without time zone]		一天中的时间
time [(*p*)] with time zone	timetz	一天中的时间，包括时区

名字	别名	描述
timestamp [(*p*)] [without time zone]	timestamp	日期和时间
timestamp [(*p*)] with time zone	timestamptz	日期和时间，包括时区

兼容性

下列类型（或它们的这些拼写形式）由SQL规定：bit、bit varying、boolean、char、character varying、character、varchar、date、double precision、integer、interval、numeric、decimal、real、smallint、time（有时区或无时区）、timestamp（有时区或无时区）。

每种数据类型都有由其输入和输出函数决定的外部表示。
许多内建类型都有显而易见的外部格式。但也有一些类型要么为 PostgreSQL 所独有（例如开放和闭合路径），要么有多种可选格式（例如日期和时间类型）。
有些输入和输出函数是不可逆的。也就是说，输出函数的结果与原始输入相比可能损失精度。

为了提高执行速度，某些运算符和函数（例如加法和乘法）不做运行时错误检查。
例如在某些系统上，某些数据类型的数值运算符可能静默地导致下溢或上溢。

8.1. 数字类型

数值类型包括二字节、四字节和八字节整数，四字节和八字节浮点数，以及固定精度的小数。
表 8.2列出了可用的类型。

表 8.2. 数字类型

名字	存储尺寸	描述	范围
smallint	2字节	小范围整数	-32768 到 +32767
integer	4字节	整数的常用选择	-2147483648 到 +2147483647
bigint	8字节	大范围整数	-9223372036854775808 到 9223372036854775807
decimal	可变	用户指定精度，精确	无限制
numeric	可变	用户指定精度，精确	无限制
real	4字节	可变精度，不精确	6位十进制精度
double precision	8字节	可变精度，不精确	15位十进制精度
serial	4字节	自动递增的整数	1到2147483647
bigserial	8字节	自动递增的大整数	1到9223372036854775807

数字类型常量的语法在第 4.1.2 节里描述。数字类型有一整套对应的算术操作符和函数。相关信息请参考第 9 章。下面的几节详细描述这些类型。

8.1.1. 整数类型

类型 smallint、integer 和 bigint 用于存储不同范围的整数，也就是没有小数部分的数。试图存储超出允许范围的值会导致错误。

常用的类型是integer，因为它提供了在范围、存储空间和性能之间的最佳平衡。
一般只有在磁盘空间紧张的时候才使用smallint类型。而bigint类型只有在integer
的范围不够用时才应使用，因为后者（integer）无疑更快。

bigint类型并非在所有平台上都能正确工作，因为它依赖于编译器对八字节整数的支持。
在没有这种支持的机器上，bigint的行为与integer相同（但仍占用八个字节的存储空间）。
不过，我们不知道有任何合理的平台属于这种情况。

SQL只规定了整数类型integer（或int）和smallint。类型bigint以及类型名int2、int4
和int8都是扩展，也在各种其他SQL数据库系统中使用。

注意

如果你有一个带索引的 smallint 或 bigint 类型的列，可能会遇到系统不使用该索引的问题。例如，如下形式的子句

```
... WHERE smallint_column = 42
```

不会使用索引，因为系统把类型 integer 指派给常量 42，而 PostgreSQL 目前在涉及两种不同数据类型时无法使用索引。一个变通办法是给常量加单引号，即：

```
... WHERE smallint_column = '42'
```

这会使系统推迟类型解析，从而为常量指派正确的类型。

8.1.2. 任意精度数字

numeric 类型最多可存储 1000 位精度的数字并精确执行计算。特别推荐用它存储货币金额和其他要求精确的数量。但与下一节描述的浮点类型相比，numeric 类型非常慢。

下面使用如下术语：numeric的小数位数是小数部分中十进制数字的数量，
即小数点右侧的位数。numeric的精度是整个数中有效数字的总数，
即小数点两侧的数字位数之和。因此，数值 23.5141 的精度为 6，小数位数为 4。整数可以视为小数位数为零。

numeric 类型的精度和小数位数都可以配置。要声明一个 numeric 类型的列，可使用如下语法

NUMERIC(*precision*, *scale*)

精度必须为正，小数位数为零或正。也可以

NUMERIC(*precision*)

选择小数位数 0。指定

NUMERIC

而不带任何精度和小数位数，则会创建一个可以存储任意精度和小数位数的数值的列，
精度上限为实现限制。这种列不会把输入值强制转换为任何特定的小数位数，
而声明了小数位数的 numeric 列会把输入值强制转换为该小数位数。（SQL 标准要求默认小数位数为 0，即强制转换为整数精度。我们觉得这没什么用处。如果你关心可移植性，就总是显式指定精度和小数位数。）

如果一个值的精度或小数位数大于列所声明的精度或小数位数，系统会尝试对该值四舍五入。
如果无法通过四舍五入满足声明的限制，则会引发一个错误。

类型decimal和numeric是等效的。两种类型都是SQL标准的一部分。

8.1.3. 浮点类型

数据类型real和double precision是不精确的、变精度的数字类型。实际上，在底层处理器、操作系统和编译器支持的范围内，这两种类型通常实现了IEEE 754 二进制浮点算术标准（分别对应单精度和双精度）。

所谓不精确，是指某些值无法被精确转换为内部格式，只能以近似值存储，
因此存储并再取出一个值时，可能会看到轻微差异。
如何处理这类误差以及它们在计算中如何传播，是数学和计算机科学中的一个独立领域，
这里不再展开，只强调以下几点：

- 如果你要求准确的存储和计算（例如计算货币金额），应使用numeric类型。
- 如果你想用这些类型进行任何重要的复杂计算，尤其是依赖边界情形（无穷大、下溢）特定行为的计算，那么应该仔细评估其实现。
- 比较两个浮点值是否相等，未必总能得到符合预期的结果。

在大多数平台上，real类型的范围至少为 1E-37 到 1E+37，精度至少为 6 位十进制数字。double precision类型的范围通常约为 1E-307 到 1E+308，精度至少为 15 位数字。过大或过小的值会导致错误。如果输入数字的精度过高，可能会发生舍入。过于接近零且无法表示为与零不同的数值，会导致下溢错误。

PostgreSQL 也支持 SQL 标准记法 `float` 和 `float(p)` 来指定不精确数值类型。这里，*p* 指定可接受的最小二进制精度位数。PostgreSQL 将 `float(1)` 到 `float(24)` 视为选择 `real` 类型，而 `float(25)` 到 `float(53)` 则选择 `double precision`。*p* 超出允许范围会报错。未指定精度的 `float` 视为 `double precision`。

> **注意**
>
> 在PostgreSQL 7.4 之前，`float(p)` 中的精度被理解为这么多位十进制数字。现在这一点已被纠正，以符合 SQL 标准，即精度以二进制位来衡量。假定real和double precision的尾数分别恰好有 24 位和 53 位，对于 IEEE 标准浮点实现而言是正确的。在非 IEEE 平台上可能会有少许偏差，但为简单起见，所有平台都使用相同的*p*取值范围。

8.1.4. serial 类型

serial和bigserial并不是真正的数据类型，
它们只是为了创建唯一标识符列而提供的记法便利（类似于其他一些数据库支持的AUTO_INCREMENT属性）。在当前实现中，下列语句：

```
CREATE TABLE tablename (
    colname SERIAL
);
```

等效于指定：

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
    colname integer DEFAULT nextval('tablename_colname_seq') NOT NULL
);
```

这样，我们就创建了一个整型列，并安排它的默认值由一个序列产生器分配。这里应用了NOT NULL约束，以确保不会插入空值。在大多数情况下，你可能还会想附加一个UNIQUE或PRIMARY KEY约束，以防止意外插入重复值，但这不是自动的。

> **注意**
>
> 在PostgreSQL 7.3 之前，serial隐含UNIQUE。现在这不再是自动的。如果希望一个serial 列具有唯一约束或作为主键，现在必须像对待任何其他数据类型一样显式指定。

要向 serial 列插入序列中的下一个值，应指定让serial列使用其默认值。这既可以通过在INSERT 语句的列表中省略该列来实现，也可以通过使用 DEFAULT 关键字来实现。

类型名serial和serial4是等价的：二者都会创建integer列。类型名bigserial和serial8的工作方式相同，只是它们创建的是bigint列。
如果预计表在其生命周期内会使用超过 2^{31} 个标识符，就应使用bigserial。

为 serial 列创建的序列会在所属列被删除时自动删除，且不能以其他方式删除。（在PostgreSQL 7.3 之前的版本中并非如此。注意，对于通过重新加载 7.3 之前数据库的转储创建的序列，这种自动删除链接不会发生；转储文件不含建立依赖链接所需的信息。）此外，
序列与列之间的这种依赖只针对 serial 列本身建立；如果有其他列引用该序列（例如通过手动调用 nextval 函数），在序列被删除时它们将被破坏。以这种方式使用 serial 列的序列被认为是不好的做法；如果你想让多个列共用同一个序列生成器，请把序列创建为独立对象。

8.2. 货币类型

> **注意**
>
> money类型已被弃用。请改用numeric或decimal类型，并结合to_char函数。

money类型以固定的小数精度存储货币金额；参见表 8.3。输入支持多种格式，包括整数字面量和浮点数字面量，以及“典型的”货币格式，例如'$1,000.00'。输出通常也采用后一种形式，但会受到区域设置影响。

表 8.3. 货币类型

名字	存储尺寸	描述	范围
money	4字节	货币数量	-21474836.48 到 +21474836.47

8.3. 字符类型

表 8.4. 字符类型

名字	描述
character varying(*n*),varchar(*n*)	有长度限制的变长
character(*n*),char(*n*)	定长，空白填充
text	无限长度的变长

表 8.4显示了在PostgreSQL里可用的一般用途的字符类型。

SQL 定义了两种主要字符类型：`character varying(n)` 和 `character(n)`，其中 *n* 是正整数。这两种类型都可以存储长度最多为 *n* 个字符的字符串。试图把更长的字符串存入这些类型的列时会报错，除非超出的字符全部都是空格，在这种情况下字符串会被截断到最大长度。（这一多少有些奇怪的例外是 SQL 标准要求的。）如果要存储的字符串短于声明长度，`character` 类型的值会用空格填充；`character varying` 类型则只是简单地存储较短的字符串。

如果显式把一个值转换为 `character varying(n)` 或 `character(n)`，那么超长的值会被截断为 *n* 个字符而不会报错。（这同样是 SQL 标准要求的。）

> **注意**
>
> 在 PostgreSQL 7.2 之前，无论显式还是隐式转换的上下文中，过长的字符串总是被截断而不报错。

记法 `varchar(n)` 和 `char(n)` 分别是 `character varying(n)` 和 `character(n)` 的别名。不带长度说明符的 `character` 等价于 `character(1)`；不带长度说明符的 `character varying` 则接受任意大小的字符串。后者是 PostgreSQL 的一个扩展。

此外，PostgreSQL 还提供 `text` 类型，用于存储任意长度的字符串。虽然 `text` 类型不在 SQL 标准中，但其他若干 SQL 数据库管理系统也提供了它。

这些类型的数据的存储需求是 4 个字节加上实际的字符串本身，对 `character` 而言，还要加上填充的空格。长字符串会被系统自动压缩，因此在磁盘上的实际占用可能更小。非常长的值还会被存储在后台表中，以免影响对较短列值的快速访问。无论如何，可存储的最长字符串大约为 1 GB。（数据类型声明中 *n* 允许的最大值比这还小。更改这个限制并没有意义，因为在多字节字符编码下，字符数和字节数可能差异很大。如果你想存储没有明确上限的长字符串，应使用 `text` 或未指定长度的 `character varying`，而不是随意给出一个长度上限。）

> **提示**
>
> 除了使用空格填充的类型会增加存储尺寸之外，这三种类型没有性能差异。

关于字符串字面量的语法，参见第 4.1.2.1 节；关于可用的运算符和函数，参见第 9 章。

例 8.1. 使用字符类型

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1
  a   | char_length
------+-------------
 ok   |           4

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
```

```
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- explicit
 truncation
SELECT b, char_length(b) FROM test2;
   b    | char_length
-------+-------------
 ok    |           2
 good  |           5
 too l |           5
```

1 函数char_length在第 9.4 节中讨论。

PostgreSQL 中还有两种固定长度字符类型，如表 8.5所示。name 类型仅用于在内部系统目录中存储标识符，并非供一般用户使用。它的长度目前定义为 64 字节（63 个可用字符加结束符），但应使用常量 NAMEDATALEN 来引用。这个长度是在编译时设定的（因此可以针对特殊用途调整）；默认最大长度在未来版本中可能会变化。类型 "char"（注意带引号）不同于 char(1)，因为它只使用 1 个字节存储。它在系统目录中被用作一种简单的枚举类型。

表 8.5. 特殊字符类型

名字	存储尺寸	描述
"char"	1字节	单字节内部类型
name	64字节	用于对象名的内部类型

8.4. 二进制数据类型

bytea数据类型允许存储二进制串，参见表 8.6。

表 8.6. 二进制数据类型

名字	存储尺寸	描述
bytea	4字节外加实际的二进制串	变长二进制串

二进制串是字节（八位组）的序列。二进制串与字符串的区别在于两个特性：第一，二进制串明确允许存储值为零的字节和其他“不可打印”字节（定义为 32 到 126 范围之外的字节）。第二，对二进制串的操作处理实际的字节，而字符串的编码和处理依赖于区域设置。

输入 bytea 值时，在 SQL 语句中作为字符串字面量的一部分使用时，某些值的字节必须转义（但所有字节值都可以转义）。一般而言，转义一个字节时，把它转换为等于其十进制字节值的三位八进制数，并在前面加两个反斜杠。表 8.7列出了必须转义的字符，并在适用处给出了替代的转义序列。

表 8.7. bytea输入转义字节

十进制字节值	描述	转义输入表示	示例	输出表示
0	0字节	'\\000'	SELECT '\\000'::bytea;	\000
39	单引号	'\''或'\\047'	SELECT '\''::bytea;	'
92	反斜线	'\\\\'或'\\134'	SELECT '\\\\'::bytea;	\\

十进制字节值	描述	转义输入表示	示例	输出表示
0 到 31 以及 127 到 255	“不可打印”字节	`'\\xxx'`（八进制值）	`SELECT '\\001'::bytea;`	`\001`

是否必须转义这些“不可打印”字节，会因区域设置不同而有所差异。在某些情况下，你可以不转义它们。请注意，表 8.7中每个示例的结果都恰好是一个字节长，尽管零字节和反斜线的输出表示多于一个字符。

表 8.7中之所以需要多个反斜线，是因为写成字符串常量的输入串在PostgreSQL服务器中必须经过两个解析阶段。每一对中的第一个反斜线会被字符串常量解析器解释为转义字符并因此被消耗，留下该对中的第二个反斜线。剩下的反斜线随后被`bytea`输入函数识别为三位八进制值的开始，或者对另一个反斜线的转义。例如，以`'\\001'`形式传给服务器的字符串常量，在经过字符串常量解析器之后变成`\001`。然后`\001`被送到`bytea`输入函数，在那里它被转换为十进制值为 1 的单个字节。注意，撇号字符不会被`bytea`特殊对待，因此它遵循字符串常量的正常规则。（另见第 4.1.2.1 节。）

`Bytea`字节在输出时也会被转义。通常，每个“不可打印”字节都会被转换为等价的三位八进制值并冠以一个反斜线。多数“可打印”字节按其在客户端字符集中的标准表示输出。十进制值为 92（反斜线）的字节有一种特殊的替代输出表示。细节见表 8.8。

表 8.8. bytea输出转义字节

十进制字节值	描述	转义输出表示	示例	输出结果
92	反斜线	`\\`	`SELECT '\\134'::bytea;`	`\\`
0 到 31 以及 127 到 255	“不可打印”字节	`\xxx`（八进制值）	`SELECT '\\001'::bytea;`	`\001`
32 到 126	“可打印”字节	客户端字符集表示	`SELECT '\\176'::bytea;`	~

取决于你所使用的 PostgreSQL 前端，在转义和反转义 `bytea` 串时可能还需要做额外工作。例如，如果你的接口会自动转换换行和回车，那么你可能还需要对它们进行转义。

SQL 标准定义了另一种二进制串类型，称为 `BLOB` 或 `BINARY LARGE OBJECT`。其输入格式与 `bytea` 不同，但提供的函数和运算符大体相同。

8.5. 日期/时间类型

PostgreSQL 支持全套 SQL 日期和时间类型，如表 8.9所示。

表 8.9. 日期/时间类型

名字	存储尺寸	描述	低值	高值	精度
`timestamp [ (p) ] [ without time zone ]`	8字节	日期和时间	4713 BC	5874897 AD	1微秒 / 14位
`timestamp [ (p) ] with time zone`	8字节	包括日期和时间，有时区	4713 BC	5874897 AD	1微秒 / 14位

名字	存储尺寸	描述	低值	高值	精度
interval [(*p*)]	12字节	时间间隔	-178000000年	178000000年	1微秒
date	4字节	仅日期	4713 BC	32767 AD	1日
time [(*p*)] [without time zone]	8字节	仅一天中的时间	00:00:00.00	23:59:59.99	1微秒
time [(*p*)] with time zone	12字节	仅一天中的时间，带时区	00:00:00.00+12	23:59:59.99-12	1微秒

注意

在 PostgreSQL 7.3 之前的版本中，仅写timestamp等效于timestamp with time zone。为了符合 SQL 标准，这一行为已被改变。

time、timestamp 和 interval 都接受一个可选精度值 *p*，用来指定在秒字段中保留多少位小数。默认情况下，对精度没有显式上界。对于 timestamp 和 interval 类型，*p* 的允许范围是 0 到 6。

注意

当 timestamp 值以双精度浮点数存储（目前的默认方式）时，精度的有效上限可能小于 6。timestamp 值存储为相对 2000-01-01 午夜之前或之后的秒数。对 2000-01-01 前后几年的日期可以达到微秒精度，但对更远的日期，精度会进一步降低。当 timestamp 值以八字节整数存储（一种编译期选项）时，全部取值范围内都有微秒精度。但八字节整数时间戳的日期范围比上表所示的更有限：从公元前 4713 年到公元 294276 年。

对于 time 类型，采用八字节整数存储时，*p* 的允许范围是 0 到 6；采用浮点存储时，允许范围是 0 到 10。

time with time zone 类型由 SQL 标准定义，但其定义具有一些会让人怀疑其实用性的特性。在大多数情况下，date、time、timestamp without time zone 和 timestamp with time zone 的组合，就足以提供任何应用所需的完整日期/时间功能。

类型abstime和reltime是内部使用的低精度类型。不推荐在应用中使用这些类型；
这些内部类型可能在将来的版本中消失。

8.5.1. 日期/时间输入

日期和时间输入几乎可以采用任何合理的格式，包括 ISO 8601、SQL 兼容格式、传统 POSTGRES 格式等。对于某些格式，日期输入中月、日、年的顺序可能有歧义，因此支持指定所期望的这些字段的顺序。将 datestyle 参数设为 MDY 可选择月-日-年解释，设为 DMY 可选择日-月-年解释，设为 YMD 可选择年-月-日解释。

PostgreSQL 在处理日期/时间输入方面，比 SQL 标准要求的更灵活。有关日期/时间输入的精确解析规则，以及可识别的文本字段（包括月份、星期几和时区），请参见附录 B。

请记住，任何日期或时间字面值输入都必须像文本字符串一样用单引号括起来。
更多信息请参见第 4.1.2.4 节。SQL 要求使用下列语法：

type [(*p*)] '*value*'

其中 *p* 是可选的精度说明，给出秒字段中保留的小数位数。精度可用于 time、timestamp 和 interval 类型，允许的取值范围见上文。如果在常量声明中没有指定精度，则默认采用该字面值本身的精度。

8.5.1.1. 日期

表 8.10显示了date类型可能的输入方式。

表 8.10. 日期输入

示例	描述
January 8, 1999	在任何datestyle输入模式下都无歧义
1999-01-08	ISO 8601；任何模式下的1月8日（推荐格式）
1/8/1999	MDY模式中的1月8日；DMY模式中的8月1日
1/18/1999	MDY模式中的1月18日；在其他模式中被拒绝
01/02/03	MDY模式中的2003年1月2日；DMY模式中的2003年2月1日；YMD模式中的2001年2月3日
1999-Jan-08	任何模式下的1月8日
Jan-08-1999	任何模式下的1月8日
08-Jan-1999	任何模式下的1月8日
99-Jan-08	YMD模式中的1月8日，否则错误
08-Jan-99	1月8日，除了在YMD模式中错误
Jan-08-99	1月8日，除了在YMD模式中错误
19990108	ISO 8601；任何模式中的1999年1月8日
990108	ISO 8601；任何模式中的1999年1月8日
1999.008	年和一年中的日子
J2451187	儒略日
January 8, 99 BC	公元前99年

8.5.1.2. 时间

一天中的时间类型包括 time [(*p*)] without time zone 和 time [(*p*)] with time zone。单独写 time 等效于 time without time zone。

这些类型的有效输入由一个一天中的时间，加上一个可选时区组成（见表 8.11和表 8.12）。如果在 time without time zone 的输入中指定了时区，它会被静默忽略。

表 8.11. 时间输入

示例	描述
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	和04:05一样，AM并不影响值
04:05 PM	和16:05一样；输入的小时必须 <= 12

示例	描述
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	名称指定的时区

表 8.12. 时区输入

示例	描述
PST	太平洋标准时间
-8:00	PST的ISO-8601偏移
-800	PST的ISO-8601偏移
-8	PST的ISO-8601偏移
zulu	UTC的军方缩写
z	zulu的缩写形式

8.5.1.3. 时间戳

时间戳类型的有效输入由一个日期和一个时间连接而成，其后可跟可选的 AD 或 BC，再后可跟可选的时区。因此

```
1999-01-08 04:05:06
```

和

```
1999-01-08 04:05:06 -8:00
```

都是有效的值，它们遵循 ISO 8601 标准。此外，还支持广泛使用的格式

```
January 8 04:05:06 1999 PST
```

也被支持。

对于 timestamp [without time zone]，输入中指定的任何显式时区都会被静默忽略。也就是说，得到的日期/时间值由输入值中显式的日期/时间字段导出，不随时区调整。

对于 timestamp with time zone，内部存储的值总是 UTC（Universal Coordinated Time，传统上称为格林尼治标准时间 GMT）。指定了显式时区的输入值会按该时区的相应偏移转换为 UTC。如果输入串中没有给出时区，则假定它位于系统 timezone 参数指示的时区，并按 timezone 时区的偏移转换为 UTC。

当输出一个 timestamp with time zone 值时，它总会从 UTC 转换到当前 timezone 时区，并显示为该时区的本地时间。若要查看其他时区的时间，可以修改 timezone，或者使用 AT TIME ZONE 构造（见第 9.8.3 节）。

在 timestamp without time zone 和 timestamp with time zone 之间转换时，通常假定 timestamp without time zone 值应被解释为，或输出为，timezone 本地时间。要为该转换指定不同的时区，可以使用 AT TIME ZONE。

8.5.1.4. 间隔

interval值可以用下列语法书写：

[@] *quantity unit* [*quantity unit*...] [*direction*]

其中：*quantity*是一个数字（可带符号）；*unit*是second、minute、hour、day、week、month、year、decade、century、millennium，或者这些单位的缩写或复数形式；*direction*可以是ago或为空。at 符号（@）是可选的噪音。不同单位的数量会带着适当的符号被隐式地累加起来。

天、小时、分钟和秒的数量可以不带显式的单位标记来指定。例如，'1 12:59:10'的读法与'1 day 12 hours 59 min 10 sec'相同。

可选的精度*p*应当在 0 到 6 之间，默认为输入字面量的精度。

8.5.1.5. 特殊值

下面的函数与 SQL 兼容，可用作相应数据类型的日期或时间值：CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME、LOCALTIMESTAMP。
后四个函数接受可选的精度说明。（另见第 9.8.4 节。）

PostgreSQL 还为方便使用支持几种特殊日期/时间输入值，如表 8.13所示。值 infinity 和 -infinity 在系统内部有特殊的表示并以相同方式显示；其余的只是记法上的简写，读取时会被转换为普通的日期/时间值。所有这些值都被当作普通常量处理，需要写成单引号括起来的形式。

表 8.13. 特殊日期/时间输入

输入字符串	有效类型	描述
epoch	date, timestamp	1970-01-01 00:00:00+00（Unix系统时间0）
infinity	timestamp	晚于所有其他时间戳
-infinity	timestamp	早于所有其他时间戳
now	date, time, timestamp	当前事务的开始时间
today	date, timestamp	今天午夜
tomorrow	date, timestamp	明天午夜
yesterday	date, timestamp	昨天午夜
allballs	time	00:00:00.00 UTC

8.5.2. 日期/时间输出

日期/时间类型的输出格式可以设为四种样式之一：ISO 8601、SQL（Ingres）、传统的POSTGRES 和 German，这通过命令SET datestyle设置。默认是ISO格式。（SQL标准要求使用 ISO 8601 格式。之所以存在名为 “SQL” 的输出格式，只是历史原因。）表 8.14展示了各种输出样式的示例。date和time类型的输出当然只包含给定示例中相应的日期部分或时间部分。

表 8.14. 日期/时间输出风格

样式说明	描述	示例
ISO	ISO 8601/SQL 标准	1997-12-17 07:37:16-08
SQL	传统样式	12/17/1997 07:37:16.00 PST
POSTGRES	原始样式	Wed Dec 17 07:37:16 1997 PST

样式说明	描述	示例
German	地区样式	17.12.1997 07:37:16.00 PST

SQL和POSTGRES风格中，如果指定了 DMY 字段顺序，“日”将出现在“月”之前，否则“月”出现在“日”之前（有关该设置如何影响输入值的解释，请参考第 8.5.1 节）。表 8.15给出了示例。

表 8.15. 日期顺序习惯

datestyle 设置	输入顺序	输出示例
SQL, DMY	*day*/*month*/*year*	17/12/1997 15:37:16.00 CET
SQL, MDY	*month*/*day*/*year*	12/17/1997 07:37:16.00 PST
Postgres, DMY	*day*/*month*/*year*	Wed 17 Dec 07:37:16 1997 PST

interval 的输出与输入格式相似，区别在于像 century 或 wek 这样的单位会被转换为年和天，并且 ago 会被转换为适当的符号。在 ISO 模式下输出形如

[*quantity unit* [...]] [*days*] [*hours*:*minutes*:*sekunden*]

用户可以用 SET datestyle 命令、postgresql.conf 配置文件中的 datestyle 参数，或服务器/客户端上的 PGDATESTYLE 环境变量来选择日期/时间风格。格式化函数 to_char（见第 9.7 节）也可作为一种更灵活的日期/时间输出格式化手段。

8.5.3. 时区

时区及时区惯例受政治决策的影响，而不只是地球几何。世界各地的时区在 20 世纪期间变得多少标准化了一些，但仍容易发生任意的变更。PostgreSQL 使用操作系统的底层特性来提供输出时区支持，而这些系统通常只包含 1902 到 2038 年期间的信息（对应传统 Unix 系统时间的全部范围）。timestamp with time zone 和 time with time zone 只在该年份范围内使用时区信息，并假定范围外的时间位于 UTC。不过由于时区支持源自底层操作系统的时区能力，它可以处理夏令时和其他特殊行为。

PostgreSQL 力求在典型用法上与 SQL 标准定义兼容。但 SQL 标准对日期和时间类型和能力方面有一种奇怪的组合。两个明显的问题是：

- 尽管 date 类型没有关联的时区，time 类型却可以有关联的时区。现实世界中的时区若不同时关联日期和时间就可能没有意义，因为偏移量在一年中可能随夏令时边界而变化。
- 默认时区被指定为相对 UTC 的固定数字偏移。在跨 DST 边界进行日期/时间运算时无法适应夏令时。

为解决这些困难，我们建议在使用时区时选用同时包含日期和时间的日期/时间类型。我们建议不要使用 time with time zone 类型（尽管为了旧应用以及与其他 SQL 实现的兼容，PostgreSQL 支持它）。对于只包含日期或只包含时间的类型，PostgreSQL 假定使用你的本地时区。

所有日期和时间在内部都以 UTC 存储。时间在发送给客户端之前会被转换为数据库服务器上的本地时间，因此默认情况下采用服务器时区。

有几种方法可以选择服务器使用的时区：

- 如果没有指定其他时区，服务器主机上的 TZ 环境变量将被服务器用作默认时区。
- timezone 配置参数可以在文件 postgresql.conf 中设置。

- libpq客户端使用`PGTZ`环境变量在连接时向服务器发送`SET TIME ZONE`命令。
- SQL 命令 `SET TIME ZONE` 设置会话的时区。

> **注意**
>
> 如果指定了无效的时区，时区会变成 UTC（至少在大多数系统上如此）。

有关可用时区的列表，请参见附录 B。

8.5.4. 内部实现

PostgreSQL在所有日期/时间计算中使用儒略日。这带来一个有用的特性：在假定一年长度为 365.2425 天的前提下，能够正确计算从公元前 4713 年到遥远未来的日期。

19　世纪之前的日期约定读起来很有意思，但它们并不足够一致，不值得写进日期/时间处理器中。

8.6. 布尔类型

PostgreSQL提供标准的SQL类型`boolean`。`boolean`类型只能处于两种状态之一：“true” 或“false”。第三种状态“unknown”由SQL空值表示。

“真”状态的有效字面值有：

```
TRUE
't'
'true'
'y'
'yes'
'1'
```

对于“假”状态，可以使用下列值：

```
FALSE
'f'
'false'
'n'
'no'
'0'
```

使用关键字 `TRUE` 和 `FALSE` 是首选的（符合 SQL）用法。

例 8.2. 使用boolean类型

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
```

```
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

例 8.2表明，`boolean`值在输出时使用字母`t`和`f`。

> **提示**
>
> `boolean` 类型的值不能直接转换为其他类型（例如 `CAST(boolval AS integer)`）不可行。这可以用 `CASE` 表达式完成：CASE WHEN *boolval* THEN '为真时的值' ELSE '为假时的值' END。另见第 9.12 节。

`boolean`使用 1 字节存储。

8.7. 几何类型

几何数据类型表示二维的空间物体。表 8.16展示了PostgreSQL中可以用的几何类型。最基础的类型是点，它是所有其他类型的基础。

表 8.16. 几何类型

名字	存储尺寸	表示	描述
`point`	16字节	平面上的点	(x,y)
`line`	32字节	无限直线	((x1,y1),(x2,y2))
`lseg`	32字节	有限线段	((x1,y1),(x2,y2))
`box`	32字节	矩形框	((x1,y1),(x2,y2))
`path`	16+16n字节	封闭路径（类似于多边形）	((x1,y1),...)
`path`	16+16n字节	开放路径	[(x1,y1),...]
`polygon`	40+16n字节	多边形（类似于封闭路径）	((x1,y1),...)
`circle`	24字节	圆	<(x,y),r>（圆心和半径）

我们提供了丰富的函数和操作符来进行各种几何操作，例如缩放、平移、旋转以及求交等，详见第 9.9 节。

8.7.1. 点

点是几何类型的基本二维构造块。用下面任一种语法描述`point`类型的值：

```
( x , y )
  x , y
```

其中*x*和*y*分别是坐标，都是浮点数。

点使用第一种语法输出。

8.7.2. 线段

线段（lseg）用成对的点表示。lseg类型的值用下面任一种语法指定：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中 (*x1*,*y1*) 和 (*x2*,*y2*) 是线段的两个端点。

线段使用第一种语法输出。

8.7.3. 方框

方框由作为其两个对角的点对表示。box 类型的值用下列任一种语法指定：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中的 (*x1*,*y1*) 和 (*x2*,*y2*) 是方框的两个对角。

方框使用第二种语法输出。

在输入时可以提供任意两个对角，但是值将根据需要被重新排列，
并依次存储为右上角和左下角。

8.7.4. 路径

路径由相连的点集表示。路径可以是开放的（集合中第一个点和最后一个点不相连），也可以是封闭的（第一个点和最后一个点相连）。函数 popen(*p*) 和 pclose(*p*) 可用于强制路径开放或封闭，而函数 isopen(*p*) 和 isclosed(*p*) 可用于在表达式中测试这两种类型。

path类型的值用下面任一种语法指定：

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

其中的点是组成路径的线段的端点。方括弧（[]）表示一个开放的路径，圆括弧（()）表示一个封闭的路径。如第三种到第五种语法所示，当最外面的圆括号被省略时，
路径将被假定为封闭。

路径使用第一种语法输出。

8.7.5. 多边形

多边形由点集表示。多边形或许应被视为等价于封闭路径，但其存储方式不同，
并且有自己的一套支持例程。

polygon类型的值用下列任一种语法指定：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

其中的点是组成多边形边界的线段的端点。

多边形的输出使用第一种语法。

8.7.6. 圆

圆由一个圆心和一个半径代表。`circle`类型的值用下面任一种语法指定：

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

其中`(x,y)`是圆心，而`r`是圆的半径。

圆的输出用第一种语法。

8.8. 网络地址类型

PostgreSQL 提供存储 IPv4、IPv6 和 MAC 地址的数据类型，如表 8.17所示。最好使用这些类型而不是普通文本类型，因为这些类型提供输入错误检查和若干专用的运算符与函数。

表 8.17. 网络地址类型

名字	存储尺寸	描述
`cidr`	12或24字节	IPv4 或 IPv6 网络
`inet`	12或24字节	IPv4和IPv6主机以及网络
`macaddr`	6字节	MAC地址

在对 `inet` 或 `cidr` 数据类型排序时，IPv4 地址总是排在 IPv6 地址之前，即使后者是封装或映射 IPv4 地址所得的 IPv6 地址，例如 ::10.2.3.4 或 ::ffff:10.4.3.2。

8.8.1. `inet`

`inet` 类型在单个字段中保存一个 IPv4 或 IPv6 主机地址，以及可选的其所在子网的标识。子网标识通过说明主机地址中有多少位表示网络地址（即“网络掩码”）来表达。如果网络掩码是 32 且地址是 IPv4，该值就不表示子网而只表示单个主机。在 IPv6 中，地址长 128 位，因此 128 位将指定唯一的主机地址。注意，如果你只想接受网络，应使用 `cidr` 类型而不是 `inet`。

此类型的输入格式是 *address/y*，其中 *address* 是一个 IPv4 或 IPv6 地址，*y* 是网络掩码的位数。如果省略了 */y* 部分，则 IPv4 的网络掩码为 32、IPv6 的为 128，该值只表示单个主机。显示时，如果网络掩码指定的是单个主机，则会省略 */y* 部分。

8.8.2. `cidr`

`cidr` 类型保存 IPv4 或 IPv6 网络说明。输入和输出格式遵循无类别域间路由（CIDR）约定。指定网络的格式为 *address/y*，其中 *address* 是网络的 IPv4 或 IPv6 表示，而 *y* 是网络掩码中的位

数。如果省略 *y*，则会按照旧式的分类网络编号系统假设来计算，但至少会大到足以覆盖输入中写出的所有八位组。若指定的网络地址在所给网络掩码右侧仍有置位比特，则会报错。

表 8.18展示了一些示例。

表 8.18. cidr 类型输入示例

cidr输入	cidr输出	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.8.3. inet vs. cidr

inet 和 cidr 两种数据类型的本质区别在于：inet 接受网络掩码右侧仍含有非零位的值，而 cidr 不接受。

> **提示**
>
> 如果你不喜欢 inet 或 cidr 值的输出格式，可以试试 host、text 和 abbrev 函数。

8.8.4. macaddr

macaddr 类型存储 MAC 地址，即以太网卡硬件地址（尽管 MAC 地址也用于其他目的）。输入接受多种惯用格式，包括

```
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08-00-2b-01-02-03'
'08:00:2b:01:02:03'
```

它们都表示同一个地址。数字 a 到 f 大小写均可接受。输出总是采用所示形式中的最后一种。

PostgreSQL 源码分发中的 `contrib/mac` 目录包含一些工具，可以用来把 MAC 地址映射为硬件制造商名称。

8.9. 位串类型

位串是由 1 和 0 组成的字符串。它们可用于存储或展示位掩码。SQL 中有两种位类型：`bit(n)` 和 `bit varying(n)`，其中 *n* 是正整数。

`bit` 类型的数据长度必须与 *n* 完全一致；试图存储更短或更长的位串都会报错。`bit varying` 数据则是长度上限为 *n* 的变长类型，更长的串会被拒绝。不带长度的 `bit` 等效于 `bit(1)`，而不带长度说明的 `bit varying` 表示长度不受限。

> **注意**
>
> 如果显式地将一个位串值转换为 `bit(n)`，它会在右侧被截断或补零，直到长度恰好为 *n* 位，且不会报错。类似地，如果显式地将一个位串值转换为 `bit varying(n)`，而其长度超过 *n* 位，则会在右侧被截断。

> **注意**
>
> 在 PostgreSQL 7.2 之前，`bit` 数据总是被静默地在右侧截断或补零，无论是否显式转换。为了遵循 SQL 标准，这一行为已被改变。

关于位串常量的语法，参见第 4.1.2.2 节。可用的还有位逻辑运算符和字符串操作函数；见第 9 章。

例 8.3. 使用位串类型

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
ERROR:  bit string length 2 does not match type bit(3)
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;
  a  |  b
-----+-----
 101 | 00
 100 | 101
```

8.10. 数组

PostgreSQL允许把表的列定义为变长的多维数组。
可以创建任何内建类型或用户定义类型的数组。

8.10.1. 数组类型的声明

为了说明数组类型的使用，我们创建这个表：

```
CREATE TABLE sal_emp (
```

```
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

如上所示，数组数据类型通过在数组元素的数据类型名称后面追加方括号（[]）来命名。上面的命令将创建一个名为 sal_emp的表，它有一个text 类型的列（name）、一个 integer 类型的一维数组（pay_by_quarter，表示员工的季度薪水）以及一个text 类型的二维数组（schedule，表示员工的周计划）。

CREATE TABLE的语法允许指定数组的确切大小，例如：

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

然而，当前的实现并不强制数组的大小限制——其行为与未指定长度的数组相同。

实际上，当前的实现也不强制声明的维数。特定元素类型的数组无论大小或维数如何，都被视为同一类型。因此，在 CREATE TABLE中声明维数或大小只是文档性的，并不影响运行时行为。

对于一维数组，也可以使用另一种 SQL99 标准语法。pay_by_quarter本可以定义为：

```
    pay_by_quarter  integer ARRAY[4],
```

这种语法要求用一个整数常量表示数组的大小。但和前面一样，PostgreSQL并不强制该大小限制。

8.10.2. 数组值输入

要把一个数组值写成文字常量，需要把元素值用花括号括起来并用逗号分隔。（如果你了解 C，这很像 C 中初始化结构的语法。）任何元素值都可以加双引号，而且当它包含逗号或花括号时必须加双引号。（更多细节见下文。）因此，数组常量的一般格式如下：

```
'{ val1 delim val2 delim ... }'
```

其中*delim*是该类型的分隔符字符，记录在它的pg_type项中。（对所有内建类型，这个字符都是逗号“,”。）每个 *val*要么是数组元素类型的一个常量，要么是一个子数组。数组常量的一个例子是

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

这个常量是一个二维的 3×3 数组，由三个整数子数组组成。

（这类数组常量实际上只是第 4.1.2.4 节中讨论的通用类型常量的一个特例。该常量最初被当作字符串处理并传递给数组的输入转换例程。可能需要显式的类型说明。）

现在我们可以展示一些INSERT语句。

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');
```

```
INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

当前数组实现的一个限制是：数组的单个元素不能是 SQL 空值。整个数组可以设置为空，但不能让数组的一部分元素为空而另一部分非空。

这可能导致令人惊讶的结果。例如，前面两次插入的结果如下：

```
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |      schedule
-------+---------------------------+--------------------
 Bill  | {10000,10000,10000,10000} | {{meeting},{""}}
 Carol | {20000,25000,25000,25000} | {{talk},{meeting}}
(2 rows)
```

由于`schedule`的`[2][2]`元素在每条`INSERT`语句中都缺失，`[1][2]`元素被丢弃了。

> **注意**
>
> 修复这个问题已列入待办事项。

也可以使用`ARRAY`表达式语法：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['','']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['talk', 'consult'], ['meeting', '']]);
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |           schedule
-------+---------------------------+-------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{"",""}}
 Carol | {20000,25000,25000,25000} | {{talk,consult},{meeting,""}}
(2 rows)
```

注意在这种语法下，多维数组的每个维度必须有匹配的长度。不匹配会导致报错，而不是像前面的情况那样静默地丢弃值。例如：

```
INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['talk', 'consult'], ['meeting']]);
ERROR:  multidimensional arrays must have array expressions with
 matching dimensions
```

还要注意数组元素是普通的 SQL 常量或表达式；例如，字符串字面量用单引号，而不是像数组文字中那样用双引号。`ARRAY`表达式语法在第 4.2.10 节中有更详细的讨论。

8.10.3. 访问数组

现在，我们可以在这个表上运行一些查询。首先，我们展示如何一次访问数组的一个元素。下面的查询检索第二季度薪水发生变化的员工的姓名：

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

数组的下标编号写在方括号内。默认情况下，PostgreSQL对数组使用从 1 开始的编号约定，也就是说，一个有*n*个元素的数组从 `array[1]`开始，以`array[n]`结束。

这个查询检索所有员工的第三季度薪水：

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

我们也可以访问数组的任意矩形切片或子数组。数组切片通过为一个或多个数组维度写 *lower-bound*:*upper-bound* 来表示。例如，下面的查询检索 Bill 周计划前两天的第一项：

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

      schedule
--------------------
 {{meeting},{""}}
(1 row)
```

我们也可以写成

```
SELECT schedule[1:2][1] FROM sal_emp WHERE name = 'Bill';
```

结果相同。只要任何一个下标写成了 *lower*:*upper* 的形式，数组下标操作就总是表示一个数组切片。对于只指定了一个值的下标，会假定其下界为 1，例如：

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';
         schedule
---------------------------
 {{meeting,lunch},{"",""}}
(1 row)
```

任何数组值的当前维度都可以用`array_dims` 函数检索：

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:1]
(1 row)
```

array_dims产生一个text类型的结果，便于人阅读，但对程序来说可能不太方便。
维度也可以用 array_upper和array_lower 检索，它们分别返回指定数组维度的上界和下界。

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

8.10.4. 修改数组

可以完整地替换一个数组值：

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

或者使用ARRAY表达式语法：

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

也可以在单个元素上更新数组：

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

或者在切片上更新：

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

通过向与已有元素相邻的元素赋值，或向与已有数据相邻或重叠的切片赋值，
可以扩大已存储的数组值。例如，如果数组myarray当前有 4 个元素，那么向myarray[5] 赋值的更新完成后它将有 5 个元素。目前，只允许以这种方式扩大一维数组，不允许扩大多维数组。

数组切片赋值允许创建不使用从 1 开始下标的数组。例如，可以向myarray[-2:7]赋值来创建一个下标值从 -2 到 7 的数组。

也可以使用连接操作符||构造新的数组值：

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

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

连接操作符允许把单个元素压入一维数组的开头或末尾。它也接受两个*N*维数组，或者一个*N*维数组和一个*N+1*维数组。

当单个元素被压入一维数组的开头时，结果数组的下界等于右操作数的下界减一。
当单个元素被压入一维数组的末尾时，结果数组保留左操作数的下界。例如：

```
SELECT array_dims(1 || ARRAY[2,3]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

当两个维数相同的数组连接时，结果保留左操作数最外维度的下界。
结果数组由左操作数的每个元素后跟右操作数的每个元素组成。例如：

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

当一个*N*维数组被压入一个*N+1*维数组的开头或末尾时，结果与上面的元素-数组情形类似。每个 *N* 维子数组本质上是*N+1*维数组最外层维度的一个元素。例如：

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [0:2][1:2]
(1 row)
```

还可以使用函数array_prepend、array_append或array_cat 构造数组。前两个只支持一维数组，但array_cat 支持多维数组。注意，相对于直接使用这些函数，更推荐使用上面讨论的连接操作符。实际上，这些函数主要用于实现连接操作符。不过，
在创建用户自定义聚合时它们可能直接有用。一些例子：

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

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

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

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

8.10.5. 在数组中搜索

要在数组中搜索一个值，必须检查数组的每个值。如果你知道数组的大小，可以手工完成。例如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

然而，对于大型数组这很快就会变得繁琐，而且在数组大小不确定时也帮不上忙。第 9.17 节中描述了一种替代方法。上面的查询可以替换为：

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

此外，你还可以找出数组中所有值都等于 10000 的行：

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

提示

数组不是集合；搜索特定数组元素可能表明数据库设计不当。考虑使用一个独立的表，其中每个原本会是数组元素的项对应一行。这样更容易搜索，而且在元素数量很大时可能有更好的可伸缩性。

8.10.6. 数组输入和输出语法

数组值的外部文本表示由若干项构成，这些项按照数组元素类型的 I/O 转换规则解释，外加指示数组结构的装饰。装饰由环绕数组值的花括号（`{`和`}`）以及相邻项之间的分隔符字符组成。分隔符通常是逗号（`,`），但也可以是别的：它由数组元素类型的`typdelim`设置决定。（在PostgreSQL发行版提供的标准数据类型中，`box`类型使用分号（`;`），其他所有类型都使用逗号。）在多维数组中，每个维度（行、平面、立方体等）都有自己的一层花括号，并且在同层的、由花括号括起的相邻各项之间必须写上分隔符。你可以在左花括号之前、右花括号之后或任何单个项字符串之前写空白。

但项后面的空白不会被忽略：跳过前导空白后，
直到下一个右花括号或分隔符之前的所有内容都被当作项的值。

如前所述，写数组值时可以给任何单个数组元素加双引号。
如果元素值会导致数组值解析器混淆，你必须这样做。例如，包含花括号、
逗号（或任何分隔符字符）、双引号、反斜线或前导空白的元素必须加双引号。
要在加了引号的数组元素值中放入双引号或反斜线，需要在它前面加一个反斜线。另外，
你也可以使用反斜线转义来保护所有会被当作数组语法或可忽略空白的数据字符。

如果元素值是空字符串或者包含花括号、分隔符字符、双引号、反斜线或空白，
数组输出例程会给它们加上双引号。元素值中内嵌的双引号和反斜线将被反斜线转义。
对于数字数据类型，可以放心地假设永远不会出现双引号，但对于文本数据类型，
应当准备同时应对有无引号两种情况。（这是与 7.2 之前的 PostgreSQL版本相比的一个行为变化。）

注意

请记住，你在 SQL 命令中写的内容会先被解释为字符串字面量，然后再被解释为数组。这会使你需要的反斜线数量加倍。例如，要插入一个包含一个反斜线和一个双引号的`text`数组值，你需要写

```
INSERT ... VALUES ('{"\\\\","\\""}');
```

字符串字面量处理器会去掉一层反斜线，因此到达数组值解析器的内容看起来像`{"\\","\""}`。反过来，送给`text` 数据类型输入例程的字符串分别变成`\`和`"`。（如果我们使用的是其输入例程也特殊处理反斜线的数据类型，例如`bytea`，
那么要在存储的数组元素中放入一个反斜线，命令中可能需要多达八个反斜线。）

提示

在 SQL 命令中写数组值时，`ARRAY`构造器语法往往比数组文字语法更容易使用。在`ARRAY`中，单个元素值的写法与它们不是数组成员时的写法相同。

8.11. 对象标识符类型

对象标识符（OID）在 PostgreSQL 内部被用作各种系统表的主键。此外，用户创建的表也会加上一个 OID 系统列（除非建表时指定了 `WITHOUT OIDS`）。类型 `oid` 表示对象标识符。`oid` 还有几个别名类型：`regproc`、`regprocedure`、`regoper`、`regoperator`、`regclass` 和 `regtype`。表 8.19给出了一个概览。

`oid` 类型目前实现为无符号 4 字节整数。因此，在大型数据库中，它不足以提供数据库范围内的唯一性，甚至在大型的单个表中也无法保证唯一性。因此，不推荐把用户创建表的 OID 列用作主键。OID 最好仅用于引用系统表。

`oid` 类型本身除比较之外几乎没有其他操作。不过，它可以转换为整数，然后再用标准整数操作符进行处理。（这样做时要注意有符号与无符号可能造成的混淆。）

OID 别名类型除了专门的输入和输出例程外没有自己的运算。这些例程能够接受并显示系统对象的符号名称，而不是类型 `oid` 会使用的原始数值。别名类型可以简化对象 OID 值的查找：例如，可以写 `'mytable'::regclass` 来获得表 `mytable` 的 OID，而不必写 `SELECT oid FROM pg_`

class WHERE relname = 'mytable'。（实际上，当不同模式中有多个名为 mytable 的表时，要选出正确的 OID 需要复杂得多的 SELECT。）

表 8.19. 对象标识符类型

名字	引用	描述	值示例
oid	任意	数字形式的对象标识符	564182
regproc	pg_proc	函数名称	sum
regprocedure	pg_proc	函数与参数类型	sum(int4)
regoper	pg_operator	操作符名称	+
regoperator	pg_operator	带参数类型的操作符	*(integer,integer) 或-(NONE,integer)
regclass	pg_class	关系名称	pg_type
regtype	pg_type	数据类型名称	integer

所有 OID 别名类型都接受模式限定名称；如果不加限定就无法在当前搜索路径中找到对象，输出时也会显示模式限定名称。regproc和regoper别名类型只接受唯一的（未重载的）输入名称，因此用途有限；对于大多数用途，regprocedure或regoperator更合适。
对于regoperator，通过将未使用的操作数写为NONE来标识一元操作符。

系统使用的另一种标识符类型是xid，即事务（缩写为xact）标识符。这是系统列xmin和xmax的数据类型。事务标识符是 32 位的量。

系统使用的第三种标识符类型是 cid，也就是命令标识符。这是系统列 cmin 和 cmax 的数据类型。命令标识符同样是 32 位量。

系统使用的最后一种标识符类型是 tid，也就是元组标识符（行标识符）。这是系统列 ctid 的数据类型。一个元组 ID 是一对值（块号、块内元组索引），用来标识该行在其所属表中的物理位置。

（系统列的更多说明见第 5.2 节。）

8.12. 伪类型

PostgreSQL 的类型系统中包含一些特殊用途的条目，统称为伪类型。伪类型不能作为列数据类型使用，但可以用于声明函数的参数类型或结果类型。
每一种伪类型都适用于这样的场景：函数的行为并不对应于简单地接受或返回某个特定 SQL 数据类型的值。表 8.20列出了现有的伪类型。

表 8.20. 伪类型

名字	描述
any	表示一个函数可以接受任意输入数据类型。
anyarray	表示一个函数可以接受任意数组数据类型（参见第 33.2.5 节）。
anyelement	表示一个函数可以接受任意数据类型（参见第 33.2.5 节）。
cstring	表示一个函数接受或返回一个以空字符结尾的 C 字符串。
internal	表示一个函数接受或返回一个服务器内部数据类型。

名字	描述
language_handler	表示过程语言调用处理器被声明为返回 language_handler。
record	标识一个返回未指定行类型的函数。
trigger	触发器函数被声明为返回trigger.
void	表示一个函数不返回值。
opaque	一个已废弃的类型名称，以前用于上述所有用途。

用 C 编写的函数（无论是内置的还是动态装载的）都可以声明为接受或返回这些伪类型中的任意一种。函数作者必须自行确保，当伪类型被用作参数类型时，该函数的行为仍然安全。

用过程语言编写的函数，只有在其实现语言允许的情况下才能使用伪类型。目前，各过程语言都禁止把伪类型用作参数类型，而只允许把void和record用作结果类型（如果函数被用作触发器，则也允许trigger）。有些语言还支持使用anyarray和anyelement类型实现多态函数。

internal 伪类型用于声明那些只应由数据库系统内部调用、而不应被 SQL 查询直接调用的函数。如果一个函数至少有一个 internal 类型参数，那么它就不能从 SQL 中调用。为了保持这一限制的类型安全性，必须遵守这样一条编码规则：除非函数至少有一个 internal 类型参数，否则不要声明任何返回 internal 的函数。

第 9 章 函数和操作符

PostgreSQL为内置数据类型提供了大量函数和操作符。用户也可以按第 V 部分中的说明定义自己的函数和操作符。psql命令`\df`和`\do`可分别用于列出所有可用的函数和操作符。

如果你关心可移植性，
请注意本章描述的大多数函数和操作符（最简单的算术和比较操作符以及一些明确标注的函数除外）都不是 SQL 标准定义的。部分扩展功能在其他 SQL 数据库管理系统中也存在，而且在很多情况下这些功能在各种实现之间是兼容且一致的。

9.1. 逻辑操作符

常用的逻辑操作符有：

```
AND
OR
NOT
```

SQL 使用三值布尔逻辑，其中空值表示“未知”。请观察下面的真值表：

a	*b*	*a* **AND** *b*	*a* **OR** *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	**NOT** *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

操作符`AND`和`OR`是可交换的，也就是说，你可以交换左右操作数而不影响结果。但请参见第 4.2.11 节获取有关子表达式计算顺序的更多信息。

9.2. 比较操作符

常见的比较操作符都可用，如表 9.1所示。

表 9.1. 比较操作符

操作符	描述
<	小于
>	大于
<=	小于或等于
>=	大于或等于
=	等于

操作符	描述
<> or !=	不等于

> **注意**
>
> != 操作符在解析器阶段会转换为 <>。不可能实现行为不同的 != 和 <> 操作符。

比较操作符适用于所有相关的数据类型。所有比较操作符都是返回 `boolean` 类型值的二元操作符；类似 `1 < 2 < 3` 的表达式是无效的（因为没有用于比较布尔值和 `3` 的 `<` 操作符）。

除了比较操作符之外，还可以使用特殊的 `BETWEEN` 结构。

```
a BETWEEN x AND y
```

等价于

```
a >= x AND a <= y
```

类似地，

```
a NOT BETWEEN x AND y
```

等价于

```
a < x OR a > y
```

除了把第一种形式在内部改写为第二种形式所需的 CPU 周期外，这两种形式没有任何区别。

要检查一个值是否为空值，可以使用结构

```
expression IS NULL
expression IS NOT NULL
```

或者使用等价但非标准的结构

```
expression ISNULL
expression NOTNULL
```

不要写*expression* `= NULL`，因为`NULL`并不“等于”`NULL`。（null 值表示未知值，而我们并不知道两个未知值是否相等。）

一些应用可能（错误地）要求 *expression* `= NULL` 在 *expression* 求值为空值时返回真。为了支持这类应用，可以打开运行时选项 `transform_null_equals`（例如 `SET transform_null_equals TO ON;`）。这样 PostgreSQL 会把 `x = NULL` 子句转换为 `x IS NULL`。在 6.5 到 7.1 版本中这是默认行为。

还可以使用下面的结构测试布尔值

```
expression IS TRUE
expression IS NOT TRUE
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

这些结构类似于 IS NULL，它们总是返回真或假，即使操作数为空也绝不会返回空值。空输入被视为逻辑值“未知”。

9.3. 数学函数和操作符

PostgreSQL为很多类型提供了数学操作符。对于那些没有标准数学惯例的类型（如日期/时间类型），我们将在后续小节中描述实际的行为。

表 9.2列出了可用的数学操作符。

表 9.2. 数学操作符

操作符	描述	示例	结果
+	加法	2 + 3	5
-	减法	2 - 3	-1
*	乘法	2 * 3	6
/	除法（整数除法会截断结果）	4 / 2	2
%	取模（余数）	5 % 4	1
^	求幂	2.0 ^ 3.0	8
|/	平方根	|/ 25.0	5
||/	立方根	||/ 27.0	3
!	阶乘	5 !	120
!!	阶乘（前缀操作符）	!! 5	120
@	绝对值	@ -5.0	5
&	按位与	91 & 15	11
|	按位或	32 | 3	35
#	按位异或	17 # 5	20
~	按位非	~1	-2
<<	按位左移	1 << 4	16
>>	按位右移	8 >> 2	2

位串类型 bit 和 bit varying 也可以使用位操作符，如表 9.3所示。&、| 和 # 的位串操作数必须等长。进行移位时，字符串的原始长度保持不变，如表中所示。

表 9.3. 位串按位操作符

示例	结果
B'10001' & B'01101'	00001
B'10001' | B'01101'	11101
B'10001' # B'01101'	11100
~ B'10001'	01110
B'10001' << 3	01000
B'10001' >> 2	00100

表 9.4列出了可用的数学函数。表中的 dp 表示 double precision。许多函数提供了参数类型不同的多种形式。除非另有说明，函数的每种形式都返回与其参数相同的数据类型。处理

`double precision` 数据的函数大多基于主机系统的 C 库实现；因此，其精度和边界情况下的行为可能因主机系统而异。

表 9.4. 数学函数

函数	`abs(x)`
返回类型	（与*x*相同）
描述	绝对值
示例	`abs(-17.4)`
结果	`17.4`
函数	`cbrt(dp)`
返回类型	`dp`
描述	立方根
示例	`cbrt(27.0)`
结果	`3`
函数	`ceil(dp or numeric)`
返回类型	（与输入相同）
描述	不小于参数的最小整数
示例	`ceil(-42.8)`
结果	`-42`
函数	`degrees(dp)`
返回类型	`dp`
描述	弧度转换为角度
示例	`degrees(0.5)`
结果	`28.6478897565412`
函数	`exp(dp or numeric)`
返回类型	（与输入相同）
描述	指数
示例	`exp(1.0)`
结果	`2.71828182845905`
函数	`floor(dp or numeric)`
返回类型	（与输入相同）
描述	不大于参数的最大整数
示例	`floor(-42.8)`
结果	`-43`
函数	`ln(dp or numeric)`
返回类型	（与输入相同）

描述	自然对数
示例	`ln(2.0)`
结果	`0.693147180559945`
函数	`log(dp or numeric)`
返回类型	（与输入相同）
描述	以 10 为底的对数
示例	`log(100.0)`
结果	`2`
函数	`log(b numeric, x numeric)`
返回类型	`numeric`
描述	以 *b* 为底的对数
示例	`log(2.0, 64.0)`
结果	`6.0000000000`
函数	`mod(y, x)`
返回类型	（与参数类型相同）
描述	*y*/*x*的余数
示例	`mod(9,4)`
结果	`1`
函数	`pi()`
返回类型	`dp`
描述	“π” 常量
示例	`pi()`
结果	`3.14159265358979`
函数	`pow(a dp, b dp)`
返回类型	`dp`
描述	*a*的*b*次幂
示例	`pow(9.0, 3.0)`
结果	`729`
函数	`pow(a numeric, b numeric)`
返回类型	`numeric`
描述	*a*的*b*次幂
示例	`pow(9.0, 3.0)`
结果	`729`

函数	`radians(dp)`
返回类型	`dp`
描述	角度转换为弧度
示例	`radians(45.0)`
结果	`0.785398163397448`
函数	`random()`
返回类型	`dp`
描述	介于 0.0 和 1.0 之间的随机值
示例	`random()`
结果	
函数	`round(dp or numeric)`
返回类型	（与输入相同）
描述	舍入到最接近的整数
示例	`round(42.4)`
结果	`42`
函数	`round(v numeric, s int)`
返回类型	`numeric`
描述	舍入到 *s* 位小数
示例	`round(42.4382, 2)`
结果	`42.44`
函数	`setseed(dp)`
返回类型	`int32`
描述	为后续`random()`调用设置种子
示例	`setseed(0.54823)`
结果	`1177314959`
函数	`sign(dp or numeric)`
返回类型	（与输入相同）
描述	参数的符号（-1、0、+1）
示例	`sign(-8.4)`
结果	`-1`
函数	`sqrt(dp or numeric)`
返回类型	（与输入相同）
描述	平方根
示例	`sqrt(2.0)`
结果	`1.4142135623731`

函数	`trunc(dp or numeric)`
返回类型	（与输入相同）
描述	向零截断
示例	`trunc(42.8)`
结果	`42`
函数	`trunc(v numeric, s int)`
返回类型	`numeric`
描述	截断到 *s* 位小数
示例	`trunc(42.4382, 2)`
结果	`42.43`

最后，表 9.5列出了可用的三角函数。所有三角函数都接受 `double precision` 类型的参数并返回该类型的值。

表 9.5. 三角函数

函数	描述
`acos(x)`	反余弦
`asin(x)`	反正弦
`atan(x)`	反正切
`atan2(x, y)`	*x*/*y*的反正切
`cos(x)`	余弦
`cot(x)`	余切
`sin(x)`	正弦
`tan(x)`	正切

9.4. 字符串函数和操作符

本节描述用于检查和操作字符串值的函数和操作符。这里的字符串包括 `character`、`character varying` 和 `text` 类型的值。除非另有说明，下面列出的所有函数都适用于这些类型，但使用 `character` 类型时，应注意自动填充空格可能带来的影响。通常，这里描述的函数也能用于非字符串类型的数据，方法是先把该数据转换为字符串表示。
一些函数也为位串类型提供了原生实现。

SQL定义了一些字符串函数，它们使用特殊语法，其中用某些关键字而不是逗号来分隔参数。详情见表 9.6。这些函数也使用常规的函数调用语法实现（见表 9.7）。

表 9.6. SQL字符串函数和操作符

函数	*string* `||` *string*
返回类型	`text`
描述	字符串串接
示例	`'Post' || 'greSQL'`
结果	`PostgreSQL`

函数	`bit_length(string)`
返回类型	`integer`
描述	串中的位数
示例	`bit_length('jose')`
结果	`32`
函数	`char_length(string)` or `character_length(string)`
返回类型	`integer`
描述	字符串中的字符数
示例	`char_length('jose')`
结果	`4`
函数	`convert(string using conversion_name)`
返回类型	`text`
描述	使用指定的转换名更改编码。可以用`CREATE CONVERSION`定义转换，也有一些预定义的转换名。可用的转换名见表 9.8。
示例	`convert('PostgreSQL' using iso_8859_1_to_utf_8)`
结果	Unicode（UTF-8）编码下的`'PostgreSQL'`
函数	`lower(string)`
返回类型	`text`
描述	转换字符串为小写
示例	`lower('TOM')`
结果	`tom`
函数	`octet_length(string)`
返回类型	`integer`
描述	串中的字节数
示例	`octet_length('jose')`
结果	`4`
函数	`overlay(string placing string from integer [for integer])`
返回类型	`text`
描述	替换子串
示例	`overlay('Txxxxas' placing 'hom' from 2 for 4)`
结果	`Thomas`
函数	`position(substring in string)`
返回类型	`integer`
描述	指定子串的位置
示例	`position('om' in 'Thomas')`

结果	3
函数	substring(*string* [from integer] [for integer])
返回类型	text
描述	提取子串
示例	substring('Thomas' from 2 for 3)
结果	hom
函数	substring(*string* from *pattern*)
返回类型	text
描述	提取匹配 POSIX 正则表达式的子字符串
示例	substring('Thomas' from '...$')
结果	mas
函数	substring(*string* from *pattern* for *escape*)
返回类型	text
描述	提取匹配SQL正则表达式的子字符串
示例	substring('Thomas' from '%#"o_a#"_' for '#')
结果	oma
函数	trim([leading | trailing | both] [*characters*] from *string*)
返回类型	text
描述	移除*string*开始端/结束端/两端（默认为两端）仅含*characters*（默认为空格）的最长字符串。
示例	trim(both 'x' from 'xTomxx')
结果	Tom
函数	upper(*string*)
返回类型	text
描述	转换字符串为大写
示例	upper('tom')
结果	TOM

还有一些其他的字符串处理函数可用，列于表 9.7。其中一些在内部用于实现表 9.6中列出的 SQL 标准字符串函数。

表 9.7. 其他字符串函数

函数	ascii(text)
返回类型	integer
描述	参数第一个字符的ASCII码
示例	ascii('x')
结果	120

函数	btrim(*string* text, *characters* text)
返回类型	text
描述	移除*string*开头和结尾处仅由*characters*中的字符组成的最长字符串
示例	btrim('xyxtrimyyx', 'xy')
结果	trim
函数	chr(integer)
返回类型	text
描述	具有给定ASCII代码的字符
示例	chr(65)
结果	A
函数	convert(*string* text, [*src_encoding* name,] *dest_encoding* name)
返回类型	text
描述	把字符串转换为*dest_encoding*编码。原始编码由*src_encoding*指定。如果省略*src_encoding*，则假定使用数据库编码。
示例	convert('text_in_unicode', 'UNICODE', 'LATIN1')
结果	ISO 8859-1 编码表示的text_in_unicode
函数	decode(*string* text, *type* text)
返回类型	bytea
描述	解码*string*中先前用encode编码的二进制数据。参数类型与encode中的相同。
示例	decode('MTIzAAE=', 'base64')
结果	123\000\001
函数	encode(*data* bytea, *type* text)
返回类型	text
描述	把二进制数据编码为仅含ASCII的表示。支持的类型有：base64、hex、escape。
示例	encode('123\\000\\001', 'base64')
结果	MTIzAAE=
函数	initcap(text)
返回类型	text
描述	把每个单词（以空白分隔）的首字母转换为大写
示例	initcap('hi thomas')
结果	Hi Thomas
函数	length(*string*)
返回类型	integer
描述	字符串中的字符数
示例	length('jose')

结果	4
函数	lpad(*string* text, *length* integer [, *fill* text])
返回类型	text
描述	通过在前面重复字符*fill*（默认为空格）把*string*填充到长度*length*。如果*string*已经比*length*长，则将其截断（在右侧）。
示例	lpad('hi', 5, 'xy')
结果	xyxhi
函数	ltrim(*string* text, *characters* text)
返回类型	text
描述	移除字符串开头处仅由*characters*中的字符组成的最长字符串。
示例	ltrim('zzzytrim', 'xyz')
结果	trim
函数	md5(*string* text)
返回类型	text
描述	计算给定字符串的 MD5 哈希，以十六进制返回结果。
示例	md5('abc')
结果	900150983cd24fb0 d6963f7d28e17f72
函数	pg_client_encoding()
返回类型	name
描述	当前客户端编码的名称
示例	pg_client_encoding()
结果	SQL_ASCII
函数	quote_ident(*string* text)
返回类型	text
描述	返回给定字符串的适当加引号形式，使其可以在SQL语句字符串中用作标识符。只在必要时才加引号（即，当字符串包含非标识符字符或会被大小写折叠时）。内嵌的引号会被适当地双写。
示例	quote_ident('Foo')
结果	"Foo"
函数	quote_literal(*string* text)
返回类型	text
描述	返回给定字符串的适当加引号形式，使其可以在SQL语句字符串中用作字符串字面量。内嵌的引号和反斜线会被适当地双写。
示例	quote_literal('O\'Reilly')
结果	'O''Reilly'

函数	`repeat(text, integer)`
返回类型	`text`
描述	将文本重复指定次数
示例	`repeat('Pg', 4)`
结果	`PgPgPgPg`
函数	`replace(string text, from text, to text)`
返回类型	`text`
描述	把*string*中子字符串*from*的所有出现替换为子字符串*to*。
示例	`replace('abcdefabcdef', 'cd', 'XX')`
结果	`abXXefabXXef`
函数	`rpad(string text, length integer [, fill text])`
返回类型	`text`
描述	通过在后面重复字符 *fill*（默认为空格）把 *string* 填充到长度 *length*。如果 *string* 已经比 *length* 长，则将其截断。
示例	`rpad('hi', 5, 'xy')`
结果	`hixyx`
函数	`rtrim(string text, characters text)`
返回类型	`text`
描述	移除字符串结尾处仅由*characters*中的字符组成的最长字符串。
示例	`rtrim('trimxxxx', 'x')`
结果	`trim`
函数	`split_part(string text, delimiter text, field integer)`
返回类型	`text`
描述	按*delimiter*分隔*string*，并返回给定字段（从 1 开始计数）
示例	`split_part( 'abc~@~def~@~ghi', '~@~', 2)`
结果	`def`
函数	`strpos(string, substring)`
返回类型	`int`
描述	指定子串的位置（与 `position(substring in string)` 相同，但注意参数顺序相反）
示例	`strpos('high', 'ig')`
结果	`2`
函数	`substr(string, from [, count])`
返回类型	`text`
描述	提取子串（与 `substring(string from from for count)` 相同）

示例	substr('alphabet', 3, 2)
结果	ph
函数	to_ascii(text [, *encoding*])
返回类型	text
描述	把文本从其他编码转换为 ASCII [a]
示例	to_ascii('Karel')
结果	Karel
函数	to_hex(*number* integer or bigint)
返回类型	text
描述	把数字转换为等价的十六进制表示
示例	to_hex(2147483647)
结果	7fffffff
函数	translate(*string* text, *from* text, *to* text)
返回类型	text
描述	*string*中任何与*from*集合中字符匹配的字符，都会被替换成*to*集合中的相应字符。
示例	translate('12345', '14', 'ax')
结果	a23x5

[a] to_ascii 函数只支持从 LATIN1、LATIN2 和 WIN1250 转换。

表 9.8. 内置转换

Conversion Name [a]	源编码	目标编码
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf_8	SQL_ASCII	UNICODE
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf_8	BIG5	UNICODE
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf_8	EUC_CN	UNICODE
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf_8	EUC_JP	UNICODE
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf_8	EUC_KR	UNICODE
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf_8	EUC_TW	UNICODE
gb18030_to_utf_8	GB18030	UNICODE
gbk_to_utf_8	GBK	UNICODE

Conversion Name [a]	源编码	目标编码
iso_8859_10_to_utf_8	LATIN6	UNICODE
iso_8859_13_to_utf_8	LATIN7	UNICODE
iso_8859_14_to_utf_8	LATIN8	UNICODE
iso_8859_15_to_utf_8	LATIN9	UNICODE
iso_8859_16_to_utf_8	LATIN10	UNICODE
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf_8	LATIN1	UNICODE
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf_8	LATIN2	UNICODE
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf_8	LATIN3	UNICODE
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf_8	LATIN4	UNICODE
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf_8	ISO_8859_5	UNICODE
iso_8859_5_to_windows_1251	ISO_8859_5	WIN
iso_8859_5_to_windows_866	ISO_8859_5	ALT
iso_8859_6_to_utf_8	ISO_8859_6	UNICODE
iso_8859_7_to_utf_8	ISO_8859_7	UNICODE
iso_8859_8_to_utf_8	ISO_8859_8	UNICODE
iso_8859_9_to_utf_8	LATIN5	UNICODE
johab_to_utf_8	JOHAB	UNICODE
koi8_r_to_iso_8859_5	KOI8	ISO_8859_5
koi8_r_to_mic	KOI8	MULE_INTERNAL
koi8_r_to_utf_8	KOI8	UNICODE
koi8_r_to_windows_1251	KOI8	WIN
koi8_r_to_windows_866	KOI8	ALT
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4

Conversion Name [a]	源编码	目标编码
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN
mic_to_windows_866	MULE_INTERNAL	ALT
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf_8	SJIS	UNICODE
tcvn_to_utf_8	TCVN	UNICODE
uhc_to_utf_8	UHC	UNICODE
utf_8_to_ascii	UNICODE	SQL_ASCII
utf_8_to_big5	UNICODE	BIG5
utf_8_to_euc_cn	UNICODE	EUC_CN
utf_8_to_euc_jp	UNICODE	EUC_JP
utf_8_to_euc_kr	UNICODE	EUC_KR
utf_8_to_euc_tw	UNICODE	EUC_TW
utf_8_to_gb18030	UNICODE	GB18030
utf_8_to_gbk	UNICODE	GBK
utf_8_to_iso_8859_1	UNICODE	LATIN1
utf_8_to_iso_8859_10	UNICODE	LATIN6
utf_8_to_iso_8859_13	UNICODE	LATIN7
utf_8_to_iso_8859_14	UNICODE	LATIN8
utf_8_to_iso_8859_15	UNICODE	LATIN9
utf_8_to_iso_8859_16	UNICODE	LATIN10
utf_8_to_iso_8859_2	UNICODE	LATIN2
utf_8_to_iso_8859_3	UNICODE	LATIN3
utf_8_to_iso_8859_4	UNICODE	LATIN4
utf_8_to_iso_8859_5	UNICODE	ISO_8859_5
utf_8_to_iso_8859_6	UNICODE	ISO_8859_6
utf_8_to_iso_8859_7	UNICODE	ISO_8859_7
utf_8_to_iso_8859_8	UNICODE	ISO_8859_8
utf_8_to_iso_8859_9	UNICODE	LATIN5
utf_8_to_johab	UNICODE	JOHAB
utf_8_to_koi8_r	UNICODE	KOI8
utf_8_to_sjis	UNICODE	SJIS
utf_8_to_tcvn	UNICODE	TCVN
utf_8_to_uhc	UNICODE	UHC
utf_8_to_windows_1250	UNICODE	WIN1250
utf_8_to_windows_1251	UNICODE	WIN

Conversion Name [a]	源编码	目标编码
utf_8_to_windows_1256	UNICODE	WIN1256
utf_8_to_windows_866	UNICODE	ALT
utf_8_to_windows_874	UNICODE	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf_8	WIN1250	UNICODE
windows_1251_to_iso_8859_5	WIN	ISO_8859_5
windows_1251_to_koi8_r	WIN	KOI8
windows_1251_to_mic	WIN	MULE_INTERNAL
windows_1251_to_utf_8	WIN	UNICODE
windows_1251_to_windows_866	WIN	ALT
windows_1256_to_utf_8	WIN1256	UNICODE
windows_866_to_iso_8859_5	ALT	ISO_8859_5
windows_866_to_koi8_r	ALT	KOI8
windows_866_to_mic	ALT	MULE_INTERNAL
windows_866_to_utf_8	ALT	UNICODE
windows_866_to_windows_1251	ALT	WIN
windows_874_to_utf_8	WIN874	UNICODE

[a] 转换名称遵循标准命名规则：将源编码正式名称中的所有非字母数字字符替换为下划线，再接上 _to_，然后接上按相同方式处理的目标编码名称。因此，这些名称可能与惯用的编码名称不同。

9.5. 二进制串函数和操作符

本节描述用于检查和操作 bytea 类型值的函数和操作符。

SQL定义了一些具有特殊语法的二进制串函数，其中使用某些关键字而不是逗号来分隔参数。详情见表 9.9。其中一些函数也使用常规的函数调用语法实现（见表 9.10）。

表 9.9. SQL二进制串函数和操作符

函数	*string* || *string*
返回类型	bytea
描述	字符串串接
示例	'\\\\Post'::bytea || '\\047gres\\000'::bytea
结果	\\Post'gres\000
函数	octet_length(*string*)
返回类型	integer
描述	二进制串中的字节数
示例	octet_length('jo\\000se'::bytea)

结果	5
函数	position(*substring* in *string*)
返回类型	integer
描述	指定子串的位置
示例	position('\\000om'::bytea in 'Th\\000omas'::bytea)
结果	3
函数	substring(*string* [from integer] [for integer])
返回类型	bytea
描述	提取子串
示例	substring('Th\\000omas'::bytea from 2 for 3)
结果	h\000o
函数	trim([both] *bytes* from *string*)
返回类型	bytea
描述	移除*string*开始端和结束端仅含*bytes*中的字节（默认为空格）组成的最长字符串
示例	trim('\\000'::bytea from '\\000Tom\\000'::bytea)
结果	Tom
函数	get_byte(*string*, *offset*)
返回类型	integer
描述	从字符串中抽取字节。
示例	get_byte('Th\\000omas'::bytea, 4)
结果	109
函数	set_byte(*string*, *offset*, *newvalue*)
返回类型	bytea
描述	在字符串中设置字节。
示例	set_byte('Th\\000omas'::bytea, 4, 64)
结果	Th\000o@as
函数	get_bit(*string*, *offset*)
返回类型	integer
描述	从字符串中抽取位。
示例	get_bit('Th\\000omas'::bytea, 45)
结果	1
函数	set_bit(*string*, *offset*, *newvalue*)
返回类型	bytea
描述	在字符串中设置位。

示例	`set_bit('Th\\000omas'::bytea, 45, 0)`
结果	`Th\000omAs`

还有一些其他的二进制串处理函数可用，列于表 9.10。其中一些在内部用于实现表 9.9中列出的 SQL 标准字符串函数。

表 9.10. 其他二进制串函数

函数	`btrim(`*`string`* `bytea` *`bytes`* `bytea)`
返回类型	`bytea`
描述	移除*`string`*开始端和结束端仅由*`bytes`*中的字节组成的最长字符串。
示例	`btrim('\\000trim\\000'::bytea, '\\000'::bytea)`
结果	`trim`
函数	`length(`*`string`*`)`
返回类型	`integer`
描述	二进制串的长度
示例	`length('jo\\000se'::bytea)`
结果	`5`
函数	`decode(`*`string`* `text,` *`type`* `text)`
返回类型	`bytea`
描述	解码*`string`*中先前用`encode`编码的二进制串。参数类型与`encode`中的相同。
示例	`decode('123\\000456', 'escape')`
结果	`123\000456`
函数	`encode(`*`string`* `bytea,` *`type`* `text)`
返回类型	`text`
描述	把二进制串编码为仅含ASCII的表示。支持的类型有：`base64`、`hex`、`escape`。
示例	`encode('123\\000456'::bytea, 'escape')`
结果	`123\000456`

9.6. 模式匹配

PostgreSQL 提供了三种独立的模式匹配方法：传统的 SQL `LIKE` 操作符、较新的 SQL99 `SIMILAR TO` 操作符，以及 POSIX 风格的正则表达式。此外还有一个模式匹配函数 `substring` 可用，它可以使用 SQL99 风格或 POSIX 风格的正则表达式。

> **提示**
>
> 如果你的模式匹配的要求超出了这些，请考虑用 Perl 或 Tcl 写一个用户定义的函数。

9.6.1. LIKE

string LIKE *pattern* [ESCAPE *escape-character*]
string NOT LIKE *pattern* [ESCAPE *escape-character*]

每个 *pattern*（模式）都定义了一个字符串集合。如果 *string*（字符串）包含在 *pattern* 所表示的字符串集合中，LIKE 表达式就返回真。（与预期的一样，如果 LIKE 返回真，NOT LIKE 表达式就返回假，反之亦然。等价的表达式是 NOT (*string* LIKE *pattern*)。）

如果 *pattern* 不包含百分号或下划线，那么该模式只表示字符串本身；此时 LIKE 的行为与等于操作符相同。*pattern* 中的下划线（_）代表（匹配）任意单个字符；百分号（%）匹配任意零个或多个字符的字符串。

一些示例：

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

LIKE 模式匹配总是覆盖整个字符串。要匹配字符串中任意位置的模式，模式必须以百分号开头并以百分号结尾。

要匹配字面量下划线或百分号而不是把它们当作通配符，*pattern* 中相应的字符前面必须带有转义字符。默认的转义字符是反斜线，但也可以使用ESCAPE 子句选择其他转义字符。要匹配转义字符本身，请写两个转义字符。

注意，反斜线在字符串常量中本来就有特殊含义，因此要写一个包含反斜线的模式常量，必须在 SQL 语句中写两个反斜线。这样，要写一个实际匹配字面量反斜线的模式，就意味着在语句中写四个反斜线。可以通过ESCAPE选择不同的转义字符来避免这一点；这样反斜线对LIKE 就不再是特殊字符。（但反斜线对字符串常量解析器仍然是特殊字符，因此要匹配一个反斜线仍需写两个。）

也可以通过写 ESCAPE '' 来选择不使用转义字符。这会禁用转义机制，从而无法关闭模式中下划线和百分号的特殊含义。

可以使用关键字 ILIKE 代替 LIKE，按照当前的区域设置使匹配不区分大小写。这不是 SQL 标准的内容，而是 PostgreSQL 的扩展。

操作符~~等效于LIKE，而~~*对应ILIKE。还有 !~~和!~~*操作符分别代表NOT LIKE和NOT ILIKE。所有这些操作符都是PostgreSQL特有的。

9.6.2. SIMILAR TO 和 SQL99 正则表达式

string SIMILAR TO *pattern* [ESCAPE *escape-character*]
string NOT SIMILAR TO *pattern* [ESCAPE *escape-character*]

SIMILAR TO 操作符根据其模式是否匹配给定字符串返回真或假。它与 LIKE 非常相似，区别在于它使用 SQL99 对正则表达式的定义来解释模式。SQL99 的正则表达式是 LIKE 记法和常用正则表达式记法的一种奇特交叉。

与LIKE类似，SIMILAR TO操作符只有在其模式匹配整个字符串时才算成功；这一点不同于普通正则表达式，后者可以匹配字符串的任意部分。与LIKE相同，SIMILAR TO也使用_和%作为通配符，分别匹配任意单个字符和任意字符串（分别类似于 POSIX 正则表达式中的.和.*）。

除了这些从LIKE借用的功能之外，SIMILAR TO支持下面这些从 POSIX 正则表达式借用的模式匹配元字符：

- |表示选择（两个候选之一）。
- *表示重复前面的项零次或更多次。
- +表示重复前面的项一次或更多次。
- 可以使用圆括号()把多个项组合成一个逻辑项。
- 一个方括号表达式[...]声明一个字符类，就像 POSIX 正则表达式一样。

注意，有界重复（?和{...}）没有提供，尽管它们存在于 POSIX 中。另外，点号（.）不是 SIMILAR TO 的一个元字符。

与LIKE一样，反斜杠会禁用这些元字符的特殊含义；也可以用 ESCAPE 指定其他转义字符。

下面是一些示例：

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

带三个参数的 substring 函数 substring(*string* from *pattern* for *escape-character*) 提取与 SQL99 正则表达式模式匹配的子串。与 SIMILAR TO 一样，指定的模式必须匹配整个数据字符串，否则函数失败并返回空值。要指明模式中成功时应当返回的部分，模式必须包含两处后面跟着双引号（"）的转义字符。
返回的文本是与这两个标记之间的模式部分匹配的文本。

下面是一些示例：

```
substring('foobar' from '%#"o_b#"%' for '#')   oob
substring('foobar' from '#"o_b#"%' for '#')    NULL
```

9.6.3. POSIX正则表达式

表 9.11列出了所有可用于 POSIX 正则表达式模式匹配的操作符。

表 9.11. 正则表达式匹配操作符

操作符	描述	示例
~	匹配正则表达式，区分大小写	'thomas' ~ '.*thomas.*'
~*	匹配正则表达式，不区分大小写	'thomas' ~* '.*Thomas.*'
!~	不匹配正则表达式，区分大小写	'thomas' !~ '.*Thomas.*'
!~*	不匹配正则表达式，不区分大小写	'thomas' !~* '.*vadim.*'

POSIX正则表达式提供了比LIKE和SIMILAR TO操作符更强大的模式匹配方式。许多 Unix 工具，例如egrep、sed或awk，都使用与这里描述的模式匹配语言相似的语言。

正则表达式是一个字符序列，它是一个字符串集合（正则集）的缩写定义。
如果一个字符串是正则表达式所描述的正则集中的成员，就说该字符串匹配这个正则表达式。与 LIKE 一样，模式字符与字符串字符精确匹配，除非它们是正则表达式语言中的特殊字符-- 但正

则表达式使用的特殊字符与 `LIKE` 不同。与 `LIKE` 模式不同的是，除非正则表达式被显式地锚定到字符串的开头或结尾，否则正则表达式可以匹配字符串中任意位置的内容。

下面是一些示例：

```
'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^(b|c)' false
```

带两个参数的 `substring` 函数 `substring(string from pattern)` 提取与 POSIX 正则表达式模式匹配的子串。如果没有匹配则返回空值，否则返回与模式匹配的文本部分。但是，如果模式包含任何圆括号，
则返回与第一个带圆括号的子表达式（左圆括号最先出现的那个）匹配的文本部分。
如果想在表达式中使用圆括号而又不触发这一例外，总可以把整个表达式用圆括号括起来。
另请参阅下文描述的非捕获圆括号。

下面是一些示例：

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

PostgreSQL的正则表达式是使用 Henry Spencer 写的一个包来实现的。下面的正则表达式的大部分描述都是从他的手册页中逐字拷贝过来的。

9.6.3.1. 正则表达式细节

正则表达式（RE），在POSIX 1003.2 中定义，它有两种形式：扩展的RE或者是ERE（大概地说就是那些在`egrep`里的），基本的RE或者是BRE（大概地说就是那些在`ed`里的）。
PostgreSQL支持两种形式，并且还实现了一些POSIX标准中没有但是在类似 Perl 或者 Tcl 这样的语言中得到广泛应用的一些扩展。使用了那些非POSIX扩展的RE叫高级RE，或者本文档里说的ARE。ARE 几乎完全是 ERE 的超集，但是 BRE 有几个符号上的不兼容（以及更多的限制）。我们首先描述 ARE 和 ERE 形式，描述那些只适用于 ARE 的特性，然后描述 BRE 的区别是什么。

> **注意**
>
> PostgreSQL 接受哪种形式的正则表达式，可以通过设置运行时参数 `regex_flavor`（在第 16.4 节中描述）来选择。通常的设置是 `advanced`，但为了与 PostgreSQL 7.4 之前的版本最大程度地向后兼容，也可以选择 `extended`。

一个正则表达式被定义为一个或更多分支，它们之间被 `|` 分隔。
只要能匹配其中一个分支的东西都能匹配正则表达式。

一个分支是零个或多个量化原子或约束，连接在一起。它匹配第一个的匹配项，
然后是第二个的匹配项，依此类推；一个空分支匹配空字符串。

一个量化原子是一个原子，后面可以跟一个量词。没有量词时，匹配一次原子所匹配的内容；
有量词时，按量词指定的次数匹配原子所匹配的内容。原子可以是表 9.12列出的任何一种形式。
可用量词及其含义见表 9.13。

一个约束匹配一个空串，但只是在满足特定条件下才匹配。
约束可以在能够使用原子的地方使用，只是它不能跟着量词。简单的约束在表 9.14里显示；更多的约束稍后描述。

表 9.12. 正则表达式原子

原子	描述
(*re*)	（其中*re*是任意正则表达式）匹配*re*所匹配的内容，并记录该匹配，以备输出结果
(?: *re*)	同上，但匹配不被记录以供报告（一组“不捕获的”圆括号）（仅 ARE）
.	匹配任意单个字符
[*chars*]	一个方括号表达式，匹配*chars*中的任意一个（详见第 9.6.3.2 节）
\ *k*	（其中*k*既不是字母也不是数字）把该字符视为普通字符并匹配它，例如，\\匹配反斜线字符
\ *c*	其中*c*是字母或数字（后面可能还有其他字符），这是一个转义，参见第 9.6.3.3 节（仅适用于 ARE；在 ERE 和 BRE 中，它匹配*c*）
{	如果后面跟着非数字字符，则匹配左花括号{；如果后面跟着数字，则是*bound*的开头（见下文）
x	其中*x*是一个没有其它意义的单个字符，则匹配该字符

RE 不能以反斜线（\）结尾。

> **注意**
>
> 请记住，反斜线（\）在PostgreSQL字符串常量中本来就有特殊含义。
> 要写一个包含反斜线的模式常量，必须在语句中写两个反斜线。

表 9.13. 正则表达式量词

量词	匹配
*	一个由原子的 0 次或更多次匹配组成的序列
+	一个由原子的 1 次或更多次匹配组成的序列
?	一个由原子的 0 次或 1 次匹配组成的序列
{ *m* }	一个由原子的正好*m*次匹配组成的序列
{ *m* , }	一个由原子的*m*次或更多次匹配组成的序列
{ *m* , *n* }	一个由原子的从*m*次到*n*次（包括）匹配组成的序列；*m*不能超过*n*
*?	*的非贪婪版本
+?	+的非贪婪版本
??	?的非贪婪版本
{ *m* }?	{*m*}的非贪婪版本
{ *m* , }?	{*m*, }的非贪婪版本
{ *m* , *n* }?	{*m*, *n*}的非贪婪版本

使用 { . . . } 的形式被称为界限。界限中的数字 *m* 和 *n* 是无符号十进制整数，允许的值从 0 到 255（含）。

非贪婪量词（仅适用于 ARE）与对应的普通（贪婪）量词匹配相同的可能内容，但优先选择最少的匹配次数，而不是最多的匹配次数。详见第 9.6.3.5 节。

> **注意**
>
> 一个量词不能紧跟在另外一个量词后面。量词不能作为表达式或者子表达式的开头，也不能跟在^或者|后面。

表 9.14. 正则表达式约束

约束	描述
^	在字符串开头匹配
$	在字符串末尾匹配
(?= *re*)	正向前瞻在匹配 *re* 的子串开始的任何位置匹配（仅限 ARE）
(?! *re*)	负向前瞻在没有子串匹配 *re* 开始的任何位置匹配（仅限 ARE）

先行约束不能包含反向引用（参见第 9.6.3.3 节），其中的所有圆括号都视为非捕获圆括号。

9.6.3.2. 方括号表达式

方括号表达式是一个由 [] 括起来的字符列表。它通常匹配列表中的任意单个字符（但见下文）。如果列表以 ^ 开头，则匹配任意一个不在列表其余部分中的单个字符。如果列表中的两个字符由 - 分隔，这是按整理序列位于这两个字符之间（含）的完整字符范围的简写形式，例如在 ASCII 中 [0-9] 匹配任意十进制数字。两个范围共享一个端点是非法的，例如 a-c-e。范围非常依赖于整理序列，因此可移植的程序应当避免依赖它们。

想在列表中包含字面字符]，可以让它做列表的首字符（如果使用了^，需要放在其后）。
想在列表中包含字面字符-，可以让它做列表的首字符或者尾字符，或者一个范围的第二个端点。
想在列表中把字面字符-当做范围的起点，把它用[.和.]包围起来，
这样它就成为一个排序元素（见下文）。除了这些字符本身、一些用[的组合（见下段）
以及转义（只在 ARE 中有效）以外，所有其它特殊字符在方括号表达式里都失去它们的特殊含义。特别是，在 ERE 和 BRE 规则下\不是特殊的，但在 ARE 里，它是特殊的（引入一个转义）。

在一个方括号表达式里，一个排序元素（一个字符、
一个被当做一个单一字符排序的多字符序列或者一个表示上面两种情况的排序序列名称）包含在[.和.]里面的时候表示该排序元素的字符序列。该序列被当做该方括号列表的一个单一元素。
这允许一个包含多字符排序元素的方括号表达式去匹配多于一个字符，例如，
如果排序序列包含一个ch排序元素，那么 RE [[.ch.]]*c匹配chchcc的头五个字符。

> **注意**
>
> PostgreSQL当前不支持多字符排序元素。这些信息描述了将来可能有的行为。

在方括号表达式里，包围在[=和=]里的排序元素是一个等价类，
代表等效于那一个的所有排序元素的字符序列，包括它本身（如果没有其它等效排序元素，
那么就好像封装定界符是[.和 .]）。例如，如果o和^是一个等价类的成员，那么[[=o=]]、[[=^=]]和[o^]都是同义的。一个等价类不能是一个范围的端点。

在方括号表达式中，用 `[:` 和 `:]` 括起的字符类名表示属于该类的所有字符的列表。标准字符类名包括：`alnum`、`alpha`、`blank`、`cntrl`、`digit`、`graph`、`lower`、`print`、`punct`、`space`、`upper`、`xdigit`。它们表示 ctype 中定义的字符类。区域设置可以提供其他字符类。字符类不能用作范围的端点。

方括号表达式有两个特例：`[[:<:]]` 和 `[[:>:]]` 是约束，分别匹配单词开头和结尾的空字符串。单词定义为一个前后都没有单词字符的单词字符序列。单词字符是 `alnum` 字符（由 ctype 定义）或下划线。这是与 POSIX 1003.2 兼容但未由其规定的扩展，在打算移植到其他系统的软件中应谨慎使用。通常更适合使用下文介绍的约束转义；它们并不更标准，但更容易键入。

9.6.3.3. 正则表达式转义

转义是以\开头，后面跟着一个字母或数字字符的特殊序列。转义有好几种变体：字符输入转义、字符类简写转义、约束转义以及反向引用。在 ARE 里，如果一个\后面跟着一个字母或数字，但是并未组成一个合法的转义，那么它是非法的。在 ERE 中没有转义：在方括号表达式之外，一个后面跟着字母或数字字符的\只是表示该字符是一个普通的字符，而且在一个方括号表达式里，\是一个普通的字符（后者是 ERE 与 ARE 之间唯一的实际不兼容之处）。

字符输入转义便于在 RE 中指定不可打印或其他不便输入的字符，见表 9.15。

字符类简写转义用来提供一些常用的字符类简写。它们显示在表 9.16中。

约束转义是以转义形式书写的约束，在满足特定条件时匹配空字符串，见表 9.17。

反向引用（`\n`）匹配数字*n*指定的被前面的圆括号子表达式匹配的同一个串（参阅表 9.18）。例如，`([bc])\1`匹配`bb`或者`cc`，但是不匹配`bc`或者`cb`。RE 中子表达式必须完全在反向引用前面。子表达式以它们的左圆括号的顺序编号。非捕获圆括号并不定义子表达式。

> **注意**
>
> 请记住，把模式作为 SQL 字符串常量输入时，转义的前导 \ 需要写成两个。

表 9.15. 正则表达式字符输入转义

转义	描述
`\a`	警告（响铃）字符，和 C 中一样
`\b`	退格，和 C 中一样
`\B`	反斜线（\）的同义词，用来减少双写反斜线
`\c` *X*	（其中*X*是任意字符）低序5位和*X*相同的字符，它的其他位都是零
`\e`	整理序列名为 `ESC` 的字符，若没有该名称则取八进制值 033 的字符
`\f`	换页，和 C 中一样
`\n`	换行符，与 C 中相同
`\r`	回车，和 C 中一样
`\t`	水平制表符，和 C 中一样
`\u` *wxyz*	（其中*wxyz*正好是四个十六进制位）本地字节序中的 Unicode 字符`U+`*wxyz*
`\U` *stuvwxyz*	（其中*stuvwxyz*正好是八个十六进制位）为假设性的 32 位 Unicode 扩展保留
`\v`	垂直制表符，和 C 中一样

转义	描述
\x *hhh*	（其中*hhh*是十六进制位的任意序列）十六进制值为0x*hhh*的字符（一个单一字符，不管用了多少个十六进制位）
\0	值为0（空字节）的字符
\ *xy*	（其中*xy*正好是两个八进制位，并且不是一个反向引用）八进制值为0*xy*的字符
\ *xyz*	（其中*xyz*正好是三个八进制位，并且不是一个反向引用）八进制值为0*xyz*的字符

十六进制位是0-9、a-f和A-F。八进制位是0-7。

字符输入转义总是被当作普通字符。例如，\135是 ASCII 中的]，但\135并不终止一个方括号表达式。

表 9.16. 正则表达式字符类简写转义

转义	描述
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_]（注意包括下划线）
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]_]（注意包括下划线）

在方括号表达式中，\d、\s 和 \w 会失去其外层方括号，而 \D、\S 和 \W 则是非法的。（因此，例如 [a-c\d] 等同于 [a-c[:digit:]]。此外，等同于 [a-c^[:digit:]] 的 [a-c\D] 是非法的。）

表 9.17. 正则表达式约束转义

转义	描述
\A	只在字符串的开头匹配（它与 ^ 的区别见第 9.6.3.5 节）
\m	只在一个词的开头匹配
\M	只在一个词的末尾匹配
\y	只在一个词的开头或末尾匹配
\Y	只在不属于单词开头或末尾的位置匹配
\Z	只在字符串的结尾匹配（它与 $ 的区别见第 9.6.3.5 节）

单词的定义与上文[[:<:]]和[[:>:]]的说明相同。方括号表达式中不允许使用约束转义。

表 9.18. 正则表达式反向引用

转义	描述
\ *m*	（其中 *m* 是一个非零数字）对第*m*个子表达式的反向引用
\ *mnn*	（其中*m*是一个非零数字，并且*nn*是后续的若干数字，并且十进制值*mnn*不大于此前已出现的捕获右圆括号数）一个到第*mnn*个子表达式的反向引用

> **注意**
>
> 八进制字符输入转义与反向引用之间存在固有歧义，
> 按上文提到的启发式规则解决：前导零始终表示八进制转义。单个非零数字，
> 如果后面没有其他数字，始终视为反向引用。不以零开头的多位数字序列，
> 如果前面已有相应的子表达式（即该数字在反向引用的合法范围内），则视为反向引用，
> 否则视为八进制转义。

9.6.3.4. 正则表达式元语法

除了上面描述的主要语法之外，还有几种特殊形式和杂项语法。

通常所使用的 RE 风味由 `regex_flavor` 决定。但这可以被一个导向前缀覆盖。如果任何风味的 RE 以 `***:` 开头，RE 的其余部分被当作 ARE。如果任何风味的 RE 以 `***=` 开头，RE 的其余部分被当作字面字符串，其中所有字符都被视为普通字符。

一个 ARE 可以以嵌入选项开头：一个序列 `(?xyz)`（这里的*xyz*是一个或多个字母字符）声明影响剩余 RE 的选项。这些选项覆盖任何先前确定的选项（包括 RE 的语法种类和大小写敏感性）。可用的选项字母在表 9.19中显示。

表 9.19. ARE 嵌入选项字母

选项	描述
`b`	RE的剩余部分是一个BRE
`c`	大小写敏感的匹配（覆盖操作符类型）
`e`	RE的剩余部分是一个ERE
`i`	大小写不敏感的匹配（见第 9.6.3.5 节）（覆盖操作符类型）
`m`	`n`的历史原因的同义词
`n`	换行敏感的匹配（见第 9.6.3.5 节）
`p`	部分换行敏感的匹配（见第 9.6.3.5 节）
`q`	RE 的剩余部分按字面（“加引号”）解释，所有字符都视为普通字符
`s`	非换行敏感的匹配（默认）
`t`	紧凑语法（默认；见下文）
`w`	逆部分换行敏感（“怪异”）的匹配（见第 9.6.3.5 节）
`x`	扩展语法（见下文）

嵌入选项在终止该序列的 `)` 处生效。它们只能在 ARE 的开头使用，不能在其后面的位置使用。

除了常用的（紧凑）RE 语法（其中所有字符都是有效的）之外，还有一种扩展语法，可以通过指定嵌入选项 `x` 来使用。在扩展语法中，RE 中的空白字符被忽略，`#` 和其后换行符（或 RE 结尾）之间的所有字符也被忽略。这样就可以对复杂的 RE 进行分段和注释。这一基本规则有三个例外：

- 前面有 `\` 的空白字符或 `#` 会被保留
- 方括号表达式内的空白或 `#` 会被保留

- 空白和注释在多字符符号中是非法的，例如 ARE 的 `(?:` 或 BRE 的 `\(`

扩展语法的空白字符是空格、制表符、换行符，以及任何属于 *space* 字符类的字符。

最后，在 ARE 里，方括号表达式外面，序列 `(?#ttt)`（其中 *ttt* 是任意不包含一个 `)` 的文本）是一个注释，它被完全忽略。同样，这样的东西是不允许出现在多字符符号的字符中间的，例如 `(?:`。这种注释更像是一种历史产物而不是一种有用的设施，并且它们的使用已经被废弃；请使用扩展语法来替代。

如果开头的 `***=` 导向已指定把用户的输入当作字面字符串而不是 RE 处理，那么这些元语法扩展一个都不可用。

9.6.3.5. 正则表达式匹配规则

当一个 RE 可以匹配给定字符串的多个子串时，RE 匹配在字符串中最先开始的那个。如果 RE 可以匹配从该点开始的多个子串，其选择由它的偏好决定：要么是最长子串，要么是最短子串。

大多数原子以及所有约束都没有偏好。带圆括号的 RE 与该 RE 具有相同的偏好（可能没有）。带量词 `{m}` 或 `{m}?` 的量化原子与原子本身具有相同的偏好（可能没有）。带其他普通量词（包括 *m* 等于 *n* 的 `{m,n}`）的量化原子偏好最长匹配。带其他非贪婪量词（包括 *m* 等于 *n* 的 `{m,n}?`）的量化原子偏好最短匹配。一个分支与其第一个有偏好的量化原子具有相同的偏好。由 `|` 操作符连接的两个或更多分支组成的 RE 偏好最长匹配。

在匹配整个 RE 的规则所施加的约束之下，子表达式也根据它们的偏好匹配最长或最短的可能子串，在 RE 中靠前的子表达式比靠后的子表达式优先。注意，外层子表达式因而比其组成部分的子表达式优先。

量词 `{1,1}` 和 `{1,1}?` 可分别用于对一个子表达式或整个 RE 强制最长和最短偏好。

在决定更长或者更短的匹配时，匹配长度是以字符衡量的，而不是排序元素。
一个空串会被认为比什么都不匹配长。例如：`bb*` 匹配 `abbbc` 的中间三个字符；`(week|wee)(night|knights)` 匹配 `weeknights` 的所有十个字符；而 `(.*).*` 匹配 `abc` 的时候，圆括号包围的子表达式匹配所有三个字符；当 `(a*)*` 被拿来匹配 `bc` 时，整个 RE 和圆括号子表达式都匹配一个空串。

如果指定不区分大小写的匹配，其效果近似于字母表中的所有大小写差别都消失了。
当存在大小写形式的字母作为普通字符出现在方括号表达式之外时，
实际上会转换为包含其大小写形式的方括号表达式，例如 `x` 变成 `[xX]`。
当它出现在方括号表达式内部时，其所有大小写形式都会加入该表达式，例如 `[x]` 变成 `[xX]`，`[^x]` 变成 `[^xX]`。

如果指定了换行敏感的匹配，`.` 和使用 `^` 的方括号表达式将永远不会匹配换行字符（这样，
匹配就不会跨越行，除非 RE 显式安排了跨行匹配）并且 `^` 和 `$` 除了分别匹配串开头和结尾之外，
还将分别匹配换行后面和前面的空串。但是 ARE 转义 `\A` 和 `\Z` 仍然只匹配串的开头和结尾。

如果指定了部分换行敏感的匹配，那么它影响 `.` 和方括号表达式，
这个时候和换行敏感的匹配一样，但是不影响 `^` 和 `$`。

如果指定了逆部分换行敏感匹配，那么它影响 `^` 和 `$`，其作用和在换行敏感的匹配里一样，
但是不影响 `.` 和方括号表达式。这个并不是很有用，只是为了满足对称性而提供的。

9.6.3.6. 限制和兼容性

在这个实现里，对 RE 的长度没有特别的限制。但是，那些希望高移植性的程序应该避免使用长度超过 256 字节的 RE，因为 POSIX 兼容的实现可以拒绝接受这样的 RE。

ARE 与 POSIX ERE 实际不兼容的唯一特性是：\在方括号表达式中不会失去特殊含义。其他所有 ARE 特性所使用的语法，在 POSIX ERE 中都是非法的，或其效果未定义或未指定；引导前缀的 ***语法同样不属于 POSIX 的 BRE 或 ERE 语法。

许多 ARE 扩展借鉴自 Perl，但其中一些经过了整理和修改，也未实现少数 Perl 扩展。需要注意的不兼容之处包括\b、\B、不对末尾换行符作特殊处理、
取反的方括号表达式也受换行敏感匹配影响、先行约束中对圆括号及反向引用的限制，
以及采用最长或最短匹配而非首次匹配的语义。

ARE 与 PostgreSQL 7.4 之前版本识别的 ERE 语法之间存在两个显著的不兼容之处：

- 在 ARE 中，\ 后跟字母数字字符要么是转义要么是错误，而在以前的版本中，这只是书写该字母数字字符的另一种方式。这应该不会造成太大问题，
  因为在早期版本中没有理由书写这样的序列。
- 在 ARE 中，\ 在 [] 内仍然是特殊字符，因此方括号表达式中的字面 \ 必须写成 \\。

虽然这些差异不太可能给大多数应用带来问题，但必要时可以通过把 regex_flavor 设置为 extended 来避免它们。

9.6.3.7. 基本正则表达式

BRE 在几个方面和 ERE 不太一样。在 BRE 中，|、+和?都是普通字符并且没有与它们功能等价的东西。范围的定界符是\{和\}，而{和}本身是普通字符。嵌套的子表达式的圆括号是\(和\)，而(和)自身是普通字符。除非在 RE 开头或者是圆括号子表达式开头，^都是一个普通字符。除非在 RE 结尾或者是圆括号子表达式的结尾，$是一个普通字符。如果*出现在 RE 开头或者是圆括号封装的子表达式开头（前面可能有^），那么它是个普通字符。最后，可以用单数字的反向引用，\<和\>分别是[[:<:]]和[[:>:]]的同义词；在 BRE 中没有其它可用的转义。

9.7. 数据类型格式化函数

PostgreSQL格式化函数提供一套强大的工具用于把各种数据类型（日期/时间、整数、浮点数、数值）转换成格式化的字符串以及反过来从格式化的字符串转换成指定的数据类型。表 9.20列出了这些函数。这些函数都遵循一个公共的调用规范：第一个参数是待格式化的值，
而第二个是一个定义输出或输入格式的模板。

表 9.20. 格式化函数

函数	返回类型	描述	示例
to_char(timestamp, text)	text	将时间戳转换为字符串	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	将时间间隔转换为字符串	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	将整数转换为字符串	to_char(125, '999')
to_char(double precision, text)	text	将实数/double precision 转换为字符串	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	将 numeric 转换为字符串	to_char(-125.8, '999D99S')

函数	返回类型	描述	示例
to_date(text, text)	date	将字符串转换为日期	to_date('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(text, text)	timestamp	将字符串转换为时间戳	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	将字符串转换为 numeric	to_number('12,454.8-', '99G999D9S')

警告：to_char(interval, text)已被废弃，不应在新编写的代码中使用。它将在下一个版本中被移除。

在一个输出模板串（对to_char）中，一些特定的模式会被识别，
并被替换成来自待格式化值的、格式恰当的数据。
任何不属于模板模式的文本都简单地照字面拷贝。同样，在一个输入模板串里（对除to_char以外的函数），模板模式标识输入数据串中要查看的部分以及要在其中找到的值。

表 9.21展示了可以用于格式化日期和时间值的模板模式。

表 9.21. 用于日期/时间格式化的模板模式

模式	描述
HH	一天中的小时（01-12）
HH12	一天中的小时（01-12）
HH24	一天中的小时（00-23）
MI	指定位置的负号（如果数字 < 0）
SS	秒（00-59）
MS	毫秒（000-999）
US	微秒（000000-999999）
SSSS	自午夜起的秒数（0-86399）
AM, A.M., PM 或 P.M.	大写形式的上午/下午指示符
am, a.m., pm 或 p.m.	小写形式的上午/下午指示符
Y,YYY	带逗号的年（4 位或者更多位）
YYYY	年（4 位或者更多位）
YYY	年的最后 3 位数字
YY	年的最后 2 位数字
Y	年的最后 1 位数字
BC, B.C., AD 或 A.D.	大写形式的纪元指示符
bc, b.c., ad 或 a.d.	小写形式的纪元指示符
MONTH	全大写的月名（空格填充到 9 个字符）
Month	首字母大写的月份全称（空格补齐到 9 字符）
month	全小写的月名（空格填充到 9 个字符）
MON	缩写的全大写月名（3 个字符）
Mon	简写的首字母大写形式的月名（3 个字符）
mon	缩写的全小写月名（3 个字符）
MM	月份编号（01-12）

模式	描述
DAY	全大写的星期名（空格填充到 9 个字符）
Day	首字母大写的星期全称（空格补齐到 9 字符）
day	全小写的星期名（空格填充到 9 个字符）
DY	缩写的全大写星期名（3 个字符）
Dy	首字母大写的星期简称（3 个字符）
dy	缩写的全小写星期名（3 个字符）
DDD	一年中的第几天（001-366）
DD	月内日序数（01-31）
D	一周中的日（1-7；周日是 1）
W	月内周序数（1-5）（第一周从该月第一天开始）
WW	一年中的第几周（1-53）（第一周从该年第一天开始）
IW	ISO 一年中的第几周（该年的第一个星期四位于第 1 周）
CC	世纪（2 位数）
J	儒略日（自公元前 4712 年 1 月 1 日以来的天数）
Q	季度
RM	以大写罗马数字表示的月份（I-XII；I=一月）
rm	以小写罗马数字表示的月份（i-xii；i=一月）
TZ	大写形式的时区名称
tz	小写形式的时区名称

修饰符可以被应用于模板模式来修改它们的行为。例如，FMMonth就是带着FM修饰符的Month模式。表 9.22展示了可用于日期/时间格式化的修饰符模式。

表 9.22. 用于日期/时间格式化的模板模式修饰符

修饰符	描述	示例
FM 前缀	填充模式（抑制填充的空格和零）	FMMonth
TH 后缀	大写序数后缀	DDTH
th 后缀	小写序数后缀	DDth
FX 前缀	固定格式的全局选项（见用法说明）	FX Month DD Day
SP 后缀	拼写模式（尚未实现）	DDSP

日期/时间格式化的用法说明：

- FM 抑制前导零和尾随空白，否则它们会被添加进来使模式的输出具有固定宽度。
- 如果不使用 FX 选项，to_timestamp 和 to_date 会跳过输入字符串中的多个连续空格。FX 必须指定为模板中的第一项。例如 to_timestamp('2000 JUN', 'YYYY MON') 是正确的，但 to_timestamp('2000 JUN', 'FXYYYY MON') 返回一个错误，因为 to_timestamp 只要求一个空格。
- to_char 模板中允许有普通文本，它们会按字面输出。可以把一个子串放在双引号中，强制把它解释为字面文本，即使它包含模式关键字。例如，在 '"Hello Year "YYYY' 中，YYYY 会被年份数据替换，但 Year 中单个的 Y 不会被替换。

- 如果想在输出中出现双引号，必须在它前面加一个反斜杠，例如 `'\\"YYYY Month\\"'`。（需要两个反斜杠，因为反斜杠在字符串常量中已经具有特殊含义。）
- 如果使用多于 4 位数字的年份，从字符串到 `timestamp` 或 `date` 的 `YYYY` 转换有一个限制：必须在 `YYYY` 之后使用某个非数字字符或模板，否则年份总是被解释为 4 位数字。例如（年份为 20000 时）：`to_date('200001131', 'YYYYMMDD')` 会被解释为 4 位年份；应在年份后使用非数字分隔符，例如 `to_date('20000-1131', 'YYYY-MMDD')` 或 `to_date('20000Nov31', 'YYYYMonDD')`。
- 在从字符串到 `timestamp` 的转换中，毫秒（`MS`）和微秒（`US`）值被用作小数点后秒的一部分。例如 `to_timestamp('12:3', 'SS:MS')` 不是 3 毫秒，而是 300，因为转换把它算作 12 + 0.3 秒。这意味着对于 `SS:MS` 格式，输入值 `12:3`、`12:30` 和 `12:300` 指定的毫秒数相同。要得到三毫秒，必须使用 `12:003`，转换把它算作 12 + 0.003 = 12.003 秒。

  下面是一个更复杂的例子：`to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US')` 是 15 小时 12 分 2 秒 + 20 毫秒 + 1230 微秒 = 2.021230 秒。

表 9.23展示了可以用于格式化数值的模板模式。

表 9.23. 用于数值格式化的模板模式

模式	描述
`9`	具有指定位数的值
`0`	带前导零的值
`.`（句点）	小数点
`,`（逗号）	分组（千位）分隔符
`PR`	尖括号内的负值
`S`	紧贴数值的正负号（使用区域设置）
`L`	货币符号（使用区域设置）
`D`	小数点（使用区域设置）
`G`	分组分隔符（使用区域设置）
`MI`	分钟（00-59）
`PL`	在指定位置的正号（如果数字 > 0）
`SG`	在指定位置的正/负号
`RN`	罗马数字（输入在 1 和 3999 之间）
`TH` or `th`	序数后缀
`V`	移动指定位数（参阅注解）
`EEEE`	科学记数法（未实现）

数字格式化的用法说明：

- 使用 `SG`、`PL` 或 `MI` 格式化的符号不锚定到数字；例如 `to_char(-12, 'S9999')` 产生 `'-12'`，而 `to_char(-12, 'MI9999')` 产生 `'- 12'`。Oracle 实现不允许在 `9` 前面使用 `MI`，而是要求 `9` 在 `MI` 之前。
- `9` 产生一个位数与 `9` 的个数相同的值。如果没有可用数字，则输出一个空格。
- `TH` 不转换小于零的值，也不转换小数。
- `PL`、`SG` 和 `TH` 是 PostgreSQL 扩展。

- V 实际上把输入值乘以 10^*n*，其中 *n* 是 V 后面的数字个数。to_char 不支持 V 与小数点组合使用。（例如，不允许 99.9V99。）

表 9.24展示了一些使用to_char函数的示例。

表 9.24. to_char示例

表达式	结果
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485'
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'

表达式	结果
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'

9.8. 日期/时间函数和操作符

表 9.26展示了可用于处理日期/时间值的函数，其细节在随后的小节中描述。表 9.25演示了基本算术操作符（+、*等）的行为。而与格式化相关的函数，可以参考第 9.7 节。你应当熟悉第 8.5 节中的日期/时间数据类型的背景知识。

下文描述的所有接受 time 或 timestamp 输入的函数和操作符实际上都有两个变体：一个接受 time with time zone 或 timestamp with time zone，另一个接受 time without time zone 或 timestamp without time zone。为简洁起见，这些变体没有单独列出。

表 9.25. 日期/时间操作符

操作符	示例	结果
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00'
+	time '03:00' + date '2001-09-28'	timestamp '2001-09-28 03:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00'
+	time '01:00' + interval '3 hours'	time '04:00'
+	interval '3 hours' + time '01:00'	time '04:00'
-	- interval '23 hours'	interval '-23:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3'
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00'
-	time '05:00' - time '03:00'	interval '02:00'
-	time '05:00' - interval '2 hours'	time '03:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00'
-	interval '1 day' - interval '1 hour'	interval '23:00'

操作符	示例	结果
-	interval '2 hours' - time '05:00'	time '03:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00'
*	double precision '3.5' * interval '1 hour'	interval '03:30'
*	interval '1 hour' * double precision '3.5'	interval '03:30'
/	interval '1 hour' / double precision '1.5'	interval '00:40'

表 9.26. 日期/时间函数

函数	age(timestamp)
返回类型	interval
描述	从今天减去
示例	age(timestamp '1957-06-13')
结果	43 years 8 mons 3 days
函数	age(timestamp, timestamp)
返回类型	interval
描述	参数相减
示例	age('2001-04-10', timestamp '1957-06-13')
结果	43 years 9 mons 27 days
函数	current_date
返回类型	date
描述	今天的日期；见第 9.8.4 节
示例	
结果	
函数	current_time
返回类型	time with time zone
描述	当日的时刻；见第 9.8.4 节
示例	
结果	
函数	current_timestamp
返回类型	timestamp with time zone
描述	日期和时间；见第 9.8.4 节
示例	
结果	

函数	`date_part(text, timestamp)`
返回类型	`double precision`
描述	获取子字段（等价于 `extract`）；参见第 9.8.1 节
示例	`date_part('hour', timestamp '2001-02-16 20:38:40')`
结果	`20`
函数	`date_part(text, interval)`
返回类型	`double precision`
描述	获取子字段（等价于 `extract`）；参见第 9.8.1 节
示例	`date_part('month', interval '2 years 3 months')`
结果	`3`
函数	`date_trunc(text, timestamp)`
返回类型	`timestamp`
描述	截断到指定精度；另见第 9.8.2 节
示例	`date_trunc('hour', timestamp '2001-02-16 20:38:40')`
结果	`2001-02-16 20:00:00`
函数	`extract(`*`field`*` from timestamp)`
返回类型	`double precision`
描述	获取子字段；参见第 9.8.1 节
示例	`extract(hour from timestamp '2001-02-16 20:38:40')`
结果	`20`
函数	`extract(`*`field`*` from interval)`
返回类型	`double precision`
描述	获取子字段；参见第 9.8.1 节
示例	`extract(month from interval '2 years 3 months')`
结果	`3`
函数	`isfinite(timestamp)`
返回类型	`boolean`
描述	测试时间戳是否有限（不等于无穷）
示例	`isfinite(timestamp '2001-02-16 21:28:30')`
结果	`true`
函数	`isfinite(interval)`
返回类型	`boolean`
描述	测试时间间隔是否有限
示例	`isfinite(interval '4 hours')`

结果	`true`
函数	`localtime`
返回类型	`time`
描述	当日的时刻；见第 9.8.4 节
示例	
结果	
函数	`localtimestamp`
返回类型	`timestamp`
描述	日期和时间；见第 9.8.4 节
示例	
结果	
函数	`now()`
返回类型	`timestamp with time zone`
描述	当前的日期和时间（等效于`current_timestamp`）；见第 9.8.4 节
示例	
结果	
函数	`timeofday()`
返回类型	`text`
描述	当前的日期和时间；见第 9.8.4 节
示例	
结果	

除了这些函数之外，还支持 SQL 的 `OVERLAPS` 操作符：

```
( start1, end1 ) OVERLAPS ( start2, end2 )
( start1, length1 ) OVERLAPS ( start2, length2 )
```

当两个时间段（由其端点定义）重叠时，该表达式产生真；不重叠时产生假。
端点可以指定为一对日期、时间或时间戳；也可以是一个日期、
时间或时间戳后跟一个时间间隔。

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
结果: true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
结果: false
```

9.8.1. EXTRACT, date_part

```
EXTRACT (field FROM source)
```

`extract` 函数从日期/时间值中检索诸如年或小时这样的子字段。*`source`* 是一个求值结果为 `timestamp` 或 `interval` 类型的值表达式。（`date` 或 `time` 类型的表达式会被转换为 `timestamp`，因此也可以使用。）*`field`* 是一个标识符或字符串，它选择从源值中提取哪个字段。`extract` 函数返回 `double precision` 类型的值。下面是有效的字段名：

`century`

年份字段除以 100

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
```

结果：`20`

注意，century 字段的结果只是年份字段除以 100，而不是把 1900 年代的大部分年份归入二十世纪的传统定义。

`day`

（月中的）日字段（1 - 31）

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
```

结果：`16`

`decade`

年份字段除以 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
```

结果：`200`

`dow`

星期几（0 - 6；星期日为 0）（仅用于 `timestamp` 值）

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
```

结果：`5`

`doy`

一年中的第几天（1 - 365/366）（仅用于 `timestamp` 值）

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
```

结果：`47`

`epoch`

对于 `date` 和 `timestamp` 值，是自 1970-01-01 00:00:00-00 以来的秒数（可以为负）；对于 `interval` 值，是该间隔中的总秒数

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:
38:40-08');
```

结果：`982384720`

```
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
```

结果：`442800`

`hour`

小时字段（0 - 23）

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
```

结果：20

microseconds

秒字段（包括小数部分）乘以 1 000 000。注意这包括完整的秒数。

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
```
结果：28500000

millennium

年份字段除以 1000

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：2

注意，millennium 字段的结果只是年份字段除以 1000，而不是把 1900 年代的年份归入第二个千年的传统定义。

milliseconds

秒字段（包括小数部分）乘以 1000。注意这包括完整的秒数。

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
```
结果：28500

minute

分钟字段（0 - 59）

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：38

month

对于 timestamp 值，是一年中的月份数（1 - 12）；对于 interval 值，是月数模 12（0 - 11）

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：2

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
```
结果：3

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
```
结果：1

quarter

该日所在的一年中的季度（1 - 4）（仅用于 timestamp 值）

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：1

second

秒字段（包括小数部分）（0 - 59[1]）

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
```

[1]如果操作系统实现了闰秒则为 60

结果：40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
结果：28.5

timezone

相对 UTC 的时区偏移量，以秒为单位。正值对应 UTC 以东的时区，负值对应 UTC 以西的时区。

timezone_hour

时区偏移量的小时部分

timezone_minute

时区偏移量的分钟部分

week

该日所在一年中的周数。根据定义（ISO 8601），一年的第一周包含该年的 1 月 4 日。（ISO-8601 的周从星期一开始。）换句话说，一年的第一个星期四在该年的第 1 周。（仅用于 timestamp 值）

SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
结果：7

year

年份字段

SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
结果：2001

extract函数主要的用途是做计算性处理。对于用于显示的日期/时间值格式化，参阅第 9.7 节。

date_part函数仿照传统的Ingres实现，后者对应SQL标准的extract函数：

date_part('*field*', *source*)

注意，此处的*field*参数必须是字符串值，而不能是名称。date_part
的有效字段名与extract相同。

SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
结果：16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
结果：4

9.8.2. date_trunc

date_trunc函数在概念上和用于数字的trunc函数类似。

date_trunc('*field*', *source*)

*source*是timestamp或interval类型的值表达式。（类型为date和time
的值会自动转换为timestamp或interval，分别对应这两种输入类型。）*field*
选择输入值的截断精度。返回值的类型是timestamp或interval，
所有小于所选精度的字段都设为零（日和月则设为一）。

field 的有效值有：

```
microseconds
milliseconds
second
minute
hour
day
month
quarter
year
decade
century
millennium
```

示例：

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
```
结果：`2001-02-16 20:00:00`

```
SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
```
结果：`2001-01-01 00:00:00`

9.8.3. AT TIME ZONE

`AT TIME ZONE`结构允许把时间戳转换到不同的时区。表 9.27展示了它的各种变体。

表 9.27. AT TIME ZONE 变体

表达式	返回类型	描述
`timestamp without time zone AT TIME ZONE` *zone*	`timestamp with time zone`	把给定时区中的本地时间转换为 UTC
`timestamp with time zone AT TIME ZONE` *zone*	`timestamp without time zone`	把 UTC 转换为给定时区中的本地时间
`time with time zone AT TIME ZONE` *zone*	`time with time zone`	跨时区转换本地时间

在这些表达式里，所需的时区 *zone* 既可以指定为文本字符串（例如 `'PST'`），也可以指定为一个间隔（例如 `INTERVAL '-08:00'`）。

示例（假定本地时区为`PST8PDT`）：

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
```
结果：`2001-02-16 19:38:40-08`

```
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE
 'MST';
```
结果：`2001-02-16 18:38:40`

第一个示例取一个无时区的时间戳，把它解释为 MST 时间（UTC-7）以产生一个 UTC 时间戳，然后旋转到 PST（UTC-8）显示。第二个示例取一个以 EST（UTC-5）指定的时间戳，把它转换成 MST（UTC-7）的本地时间。

函数`timezone(`*zone*`, `*timestamp*`)`等效于符合 SQL 标准的结构*timestamp* `AT TIME ZONE` *zone*。

9.8.4. 当前日期/时间

下列函数可用于获取当前日期和/或时间：

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME (precision)
CURRENT_TIMESTAMP (precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME (precision)
LOCALTIMESTAMP (precision)
```

CURRENT_TIME和CURRENT_TIMESTAMP返回带时区的值；LOCALTIME和LOCALTIMESTAMP返回不带时区的值。

CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME和 LOCALTIMESTAMP可以有选择地接受一个精度参数，该精度会使结果的秒字段舍入到指定的小数位数。如果没有精度参数，结果将给出可用的全部精度。

> **注意**
>
> 在PostgreSQL 7.2 之前，精度参数是没有实现的，结果总是以整数秒给出。

下面是一些示例：

```
SELECT CURRENT_TIME;
结果: 14:39:53.662522-05
SELECT CURRENT_DATE;
结果: 2001-12-23
SELECT CURRENT_TIMESTAMP;
结果: 2001-12-23 14:39:53.662522-05
SELECT CURRENT_TIMESTAMP(2);
结果: 2001-12-23 14:39:53.66-05
SELECT LOCALTIMESTAMP;
结果: 2001-12-23 14:39:53.662522
```

now()是PostgreSQL中与CURRENT_TIMESTAMP等价的传统函数。

还有函数timeofday()，出于历史原因，它返回一个text字符串而不是timestamp值：

```
SELECT timeofday();
结果: Sat Feb 17 19:07:32.000126 2001 EST
```

重要的是要知道CURRENT_TIMESTAMP及相关函数返回的是当前事务的开始时间；它们的值在事务运行期间不会改变。这被认为是一个特性：目的是为了允许一个事务在“当前”时间上有一致的概念，这样在同一个事务里的多个修改可以保持同样的时间戳。timeofday()返回墙上时钟时间，在事务运行期间会推进。

> **注意**
>
> 其他数据库系统可能会更频繁地推进这些值。

所有日期/时间数据类型也都接受特殊字面值now来指定当前日期和时间。因此，下面三种写法都返回相同的结果：

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';
```

> **注意**
>
> 在创建表时指定DEFAULT子句的情况下，你不希望使用第三种形式。系统将在分析这个常量的时候把now转换为一个timestamp，这样需要默认值时就会得到创建表的时间！而前两种形式要到实际使用默认值的时候才被计算，因为它们是函数调用。因此它们可以给出每次插入行的时刻。

9.9. 几何函数和操作符

几何类型point、box、lseg、line、path、polygon和circle有大量内置支持函数和操作符，如表 9.28、表 9.29和表 9.30中所示。

表 9.28. 几何操作符

操作符	描述	示例
+	平移	box '((0,0),(1,1))' + point '(2.0,0)'
-	平移	box '((0,0),(1,1))' - point '(2.0,0)'
*	缩放/旋转	box '((0,0),(1,1))' * point '(2.0,0)'
/	缩放/旋转	box '((0,0),(2,2))' / point '(2.0,0)'
#	相交的点或矩形框	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	路径或多边形中的点数	# '((1,0),(0,1),(-1,0))'
@-@	长度或周长	@-@ path '((0,0),(1,0))'
@@	中心	@@ circle '((0,0),10)'
##	第二个操作数上距离第一个操作数最近的点	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	两者之间的距离	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	是否重叠？（有一个公共点即为真。）	box '((0,0),(1,1))' && box '((0,0),(2,2))'
&<	重叠或在左边？	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	重叠或在右边？	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<	是否位于左侧？	circle '((0,0),1)' << circle '((5,0),1)'

操作符	描述	示例
`>>`	是否位于右侧?	`circle '((5,0),1)' >> circle '((0,0),1)'`
`<^`	是否位于下方?	`circle '((0,0),1)' <^ circle '((0,5),1)'`
`>^`	是否位于上方?	`circle '((0,5),1)' >^ circle '((0,0),1)'`
`?#`	是否相交?	`lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'`
`?-`	是否水平?	`?- lseg '((-1,0),(1,0))'`
`?-`	是否水平对齐?	`point '(1,0)' ?- point '(0,0)'`
`?|`	是否竖直?	`?| lseg '((-1,0),(1,0))'`
`?|`	是否竖直对齐?	`point '(0,1)' ?| point '(0,0)'`
`?-|`	是否互相垂直?	`lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'`
`?||`	是否平行?	`lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'`
`~`	是否包含?	`circle '((0,0),2)' ~ point '(1,1)'`
`@`	是否位于内部或边界上?	`point '(1,1)' @ circle '((0,0),2)'`
`~=`	是否相同?	`polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'`

表 9.29. 几何函数

函数	返回类型	描述	示例
`area(`*object*`)`	`double precision`	面积	`area(box '((0,0),(1,1))')`
`box_intersect(box, box)`	`box`	交集矩形框	`box_intersect(box '((0,0),(1,1))', box '((0.5,0.5),(2,2))')`
`center(`*object*`)`	`point`	中心	`center(box '((0,0),(1,2))')`
`diameter(circle)`	`double precision`	圆的直径	`diameter(circle '((0,0),2.0)')`
`height(box)`	`double precision`	矩形框的竖直尺寸	`height(box '((0,0),(1,1))')`
`isclosed(path)`	`boolean`	是否为闭合路径?	`isclosed(path '((0,0),(1,1),(2,0))')`
`isopen(path)`	`boolean`	是否为开放路径?	`isopen(path '[(0,0),(1,1),(2,0)]')`
`length(`*object*`)`	`double precision`	长度	`length(path '((-1,0),(1,0))')`

函数	返回类型	描述	示例
npoints(path)	integer	点数	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	integer	点数	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	将路径转换为闭合路径	popen(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	将路径转换为开放路径	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	圆的半径	radius(circle '((0,0),2.0)')
width(box)	double precision	矩形框的水平尺寸	width(box '((0,0),(1,1))')

表 9.30. 几何类型转换函数

函数	返回类型	描述	示例
box(circle)	box	将圆转换为矩形框	box(circle '((0,0),2.0)')
box(point, point)	box	将点转换为矩形框	box(point '(0,0)', point '(1,1)')
box(polygon)	box	将多边形转换为矩形框	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	将矩形框转换为圆	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	把点和半径转换为圆	circle(point '(0,0)', 2.0)
lseg(box)	lseg	矩形框对角线转换为线段	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	由点构造线段	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	point	将多边形转换为路径	path(polygon '((0,0),(1,1),(2,0))')
point(circle)	point	圆的圆心	point(circle '((0,0),2.0)')
point(lseg, lseg)	point	交集	point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')
point(polygon)	point	多边形的中心	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	将矩形框转换为 4 个点的多边形	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	将圆转换为 12 个顶点的多边形	polygon(circle '((0,0),2.0)')
polygon(*npts*, circle)	polygon	圆转换为*npts*个点的多边形	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	将路径转换为多边形	polygon(path '((0,0),(1,1),(2,0))')

可以把一个point当作下标为 0 和 1 的数组，访问它的两个数值分量。例如，如果t.p是一个point列，那么SELECT p[0] FROM t检索 X 坐标而 UPDATE t SET p[1] = ...改变 Y 坐标。同样，box或者lseg类型的值可以当作两个point值组成的数组看待。

9.10. 网络地址类型函数

表 9.31列出了可用于 cidr 和 inet 类型的操作符。操作符<<、<<=、>>和>>=测试子网包含关系。它们只考虑两个地址的网络部分（忽略主机部分），
并判断一个网络是否与另一个网络相同，或是另一个网络的子网。

表 9.31. cidr 和 inet 操作符

操作符	描述	示例
<	小于	inet '192.168.1.5' < inet '192.168.1.6'
<=	小于或等于	inet '192.168.1.5' <= inet '192.168.1.5'
=	等于	inet '192.168.1.5' = inet '192.168.1.5'
>=	大于或等于	inet '192.168.1.5' >= inet '192.168.1.5'
>	大于	inet '192.168.1.5' > inet '192.168.1.4'
<>	不等于	inet '192.168.1.5' <> inet '192.168.1.4'
<<	被包含于	inet '192.168.1.5' << inet '192.168.1/24'
<<=	被包含于或等于	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	包含	inet'192.168.1/24' >> inet '192.168.1.5'
>>=	包含或等于	inet '192.168.1/24' >>= inet '192.168.1/24'

表 9.32列出了可用于 cidr 和 inet 类型的函数。abbrev、host 和 text 函数主要用于提供其他显示格式。你可以用普通的类型转换语法把一个文本值转换为inet：inet(*expression*)或*colname*::inet。

表 9.32. cidr 和 inet 函数

函数	broadcast(inet)
返回类型	inet
描述	网络的广播地址
示例	broadcast('192.168.1.5/24')
结果	192.168.1.255/24
函数	host(inet)
返回类型	text
描述	以 text 形式抽取 IP 地址

示例	`host('192.168.1.5/24')`
结果	`192.168.1.5`
函数	`masklen(inet)`
返回类型	`integer`
描述	抽取网络掩码长度
示例	`masklen('192.168.1.5/24')`
结果	`24`
函数	`set_masklen(inet, integer)`
返回类型	`inet`
描述	设置 `inet` 值的网络掩码长度
示例	`set_masklen('192.168.1.5/24', 16)`
结果	`192.168.1.5/16`
函数	`netmask(inet)`
返回类型	`inet`
描述	构造网络的网络掩码
示例	`netmask('192.168.1.5/24')`
结果	`255.255.255.0`
函数	`hostmask(inet)`
返回类型	`inet`
描述	构造网络的主机掩码
示例	`hostmask('192.168.23.20/30')`
结果	`0.0.0.3`
函数	`network(inet)`
返回类型	`cidr`
描述	抽取地址的网络部分
示例	`network('192.168.1.5/24')`
结果	`192.168.1.0/24`
函数	`text(inet)`
返回类型	`text`
描述	以 text 形式抽取 IP 地址和网络掩码长度
示例	`text(inet '192.168.1.5')`
结果	`192.168.1.5/32`
函数	`abbrev(inet)`
返回类型	`text`

描述	以文本表示的缩略显示格式
示例	`abbrev(cidr '10.1.0.0/16')`
结果	`10.1/16`

表 9.33列出了可用于 `macaddr` 类型的函数。函数 `trunc(macaddr)` 返回将最后 3 个字节置零的 MAC 地址。可以用余下的前缀确定制造商。源码发行版的`contrib/mac`目录中包含一些用于创建和维护这种关联表的工具。

表 9.33. macaddr 函数

函数	`trunc(macaddr)`
返回类型	`macaddr`
描述	把最后 3 个字节设置为零
示例	`trunc(macaddr '12:34:56:78:90:ab')`
结果	`12:34:56:00:00:00`

`macaddr`类型还支持用于字典序排序的标准关系操作符（>、<=等）。

9.11. 序列操作函数

本节描述PostgreSQL中用于操作序列对象（也称为序列生成器，或简称序列）的函数。
序列对象是用`CREATE SEQUENCE`创建的特殊单行表。序列对象通常用于为表中的行生成惟一标识符。在表 9.34中列出的序列函数，提供了简单的、多用户安全方法，用于从序列对象中获取连续的序列值。

表 9.34. 序列函数

函数	返回类型	描述
`nextval(text)`	`bigint`	推进序列并返回新值
`currval(text)`	`bigint`	返回最近一次用`nextval`获取的值
`setval(text, bigint)`	`bigint`	设置序列的当前值
`setval(text, bigint, boolean)`	`bigint`	设置序列的当前值和 `is_called` 标志

主要是由于历史原因，序列函数调用所要操作的序列是通过一个文本字符串参数指定的。
为了与普通 SQL 名称的处理方式保持一定的兼容性，除非字符串用双引号括起，否则序列函数会把参数转换为小写。因此

```
nextval('foo')      操作序列 foo
nextval('FOO')      操作序列 foo
nextval('"Foo"')    操作序列 Foo
```

必要时序列名称可以带模式限定：

```
nextval('myschema.foo')     操作 myschema.foo
nextval('"myschema".foo')   与上面相同
nextval('foo')              在搜索路径中搜索 foo
```

当然，文本参数也可以是表达式的结果，不仅仅可以是简单的文字常量，这偶尔会很有用。

可用的序列函数有：

`nextval`

把序列对象推进到它的下一个值并返回该值。
这是以原子方式完成的：即使多个会话并发执行 `nextval`，每个会话也会安全地得到一个不同的序列值。

`currval`

返回当前会话中此序列最近一次由 `nextval` 获取的值。（如果本会话中从未对这个序列调用过 `nextval`，则会报告一个错误。）注意，由于返回的是会话局部的值，即使其他会话同时在执行 `nextval`，它也能给出可预测的答案。

`setval`

重置序列对象的计数器值。双参数形式把序列的 `last_value` 字段设置为指定的值，并把其 `is_called` 字段设置为 `true`，这意味着下一次 `nextval` 将先推进序列再返回一个值。在三参数形式中，`is_called` 可以设置为 `true` 或 `false`。如果把它设置为 `false`，那么下一次 `nextval` 将准确返回指定的值，序列的推进从再下一次 `nextval` 开始。例如，

```
SELECT setval('foo', 42);           下一次 nextval 将返回 43
SELECT setval('foo', 42, true);     与上面相同
SELECT setval('foo', 42, false);    下一次 nextval 将返回 42
```

`setval` 返回的结果只是其第二个参数的值。

> **重要**
>
> 为了避免阻塞从同一序列获取编号的并发事务，`nextval`操作永远不会被回滚；
> 也就是说，一旦取出了一个值，它就被视为已使用，即使执行`nextval`
> 的事务随后中止也是如此。
> 这意味着中止的事务可能会在已分配的值序列中留下未使用的“空洞”。`setval`
> 操作也永远不会被回滚。

如果一个序列对象是用默认参数创建的，那么连续调用`nextval`将返回从 1 开始的连续值。通过在CREATE SEQUENCE命令中使用特殊参数，可以得到其他行为；更多信息见其命令参考页。

9.12. 条件表达式

本节描述 PostgreSQL 中可用的 SQL 兼容的条件表达式。

> **提示**
>
> 如果你的需求超过这些条件表达式的能力，
> 你可能会希望用一种更富表现力的编程语言写一个存储过程。

9.12.1. CASE

SQL 的 `CASE` 表达式是一种通用的条件表达式，类似于其他语言中的 if/else 语句：

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

CASE 子句可以用于任何允许表达式的地方。*condition* 是一个返回 boolean 结果的表达式。如果结果为真，CASE 表达式的值就是该条件后面的 *result*。如果结果为假，则以相同的方式搜索后续的 WHEN 子句。如果没有 *condition* 为真的 WHEN 子句，则 case 表达式的值是 ELSE 子句中的 *result*。如果省略了 ELSE 子句且没有任何条件匹配，结果为空值。

示例：

```
SELECT * FROM test;

 a
---
 1
 2
 3

SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

所有*result*表达式的数据类型都必须可以转换成单一的输出类型。参阅第 10.5 节获取细节。

下面的“简单”CASE 表达式是上面一般形式的一个特化变体：

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

先计算 *expression*，然后将其与 WHEN 子句中的所有 *value* 说明逐个比较，直到找到一个相等的。如果没有找到匹配，则返回 ELSE 子句中的 *result*（或空值）。这类似于 C 中的 switch 语句。

上面的示例可以用简单CASE语法来写：

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;
```

```
 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

CASE表达式并不计算任何无助于判断结果的子表达式。例如，
下面是一个可以避免被零除错误的方法：

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

9.12.2. COALESCE

```
COALESCE(value [, ...])
```

COALESCE函数返回参数中第一个不为 null 的值。只有所有参数都为 null 时，才返回 null。它常用于在检索数据以供显示时，用默认值替换 null 值。例如：

```
SELECT COALESCE(description, short_description, '(none)') ...
```

和CASE表达式一样，COALESCE只计算确定结果所需的参数；也就是说，在第一个不为 null 的参数右边的参数不会被计算。

9.12.3. NULLIF

```
NULLIF(value1, value2)
```

当且仅当 *value1* 和 *value2* 相等时，NULLIF 函数返回空值。否则它返回 *value1*。它可用于执行与上面给出的 COALESCE 例子相反的操作：

```
SELECT NULLIF(value, '(none)') ...
```

9.13. 杂项函数

表 9.35列出了多个用于提取会话和系统信息的函数。

表 9.35. 会话信息函数

名称	返回类型	描述
current_database()	name	当前数据库名称
current_schema()	name	当前模式名称
current_schemas(boolean)	name[]	搜索路径中的模式名称，可选地包含隐式模式
current_user	name	当前执行上下文的用户名
session_user	name	会话用户名
user	name	等价于 current_user
version()	text	PostgreSQL 版本信息

session_user 是发起数据库连接的用户；它在该连接期间是固定的。current_user 是适用于权限检查的用户标识符。通常它等于会话用户，但在执行带有 SECURITY DEFINER 属性的函数期间会发生变化。用 Unix 的说法，会话用户是“真实用户”，当前用户是“有效用户”。

> **注意**
>
> current_user、session_user和user
> 在SQL中具有特殊的语法地位：调用它们时不得带尾随的圆括号。

current_schema 返回搜索路径中第一个模式的名称（如果搜索路径为空，则返回空值）。在创建表或其他命名对象时，如果未指定目标模式，就会使用该模式。current_schemas (boolean) 返回当前搜索路径中所有模式名称的数组。布尔选项决定是否在返回的搜索路径中包含 pg_catalog 等隐式包含的系统模式。

> **注意**
>
> 可以在运行时修改搜索路径，命令如下：
>
> SET search_path TO *schema* [, *schema*, ...]

version()返回一个描述PostgreSQL服务器版本的字符串。

表 9.36展示了那些可以用于查询以及修改运行时配置参数的函数。

表 9.36. 配置设定函数

名称	返回类型	描述
current_setting(*setting_name*)	text	获取设置的当前值
set_config(*setting_name*, *new_value*, *is_local*)	text	设置参数并返回新值

函数current_setting返回设置*setting_name*的当前值。它对应于SQL命令SHOW。例如：

```
SELECT current_setting('datestyle');

 current_setting
-----------------
 ISO, MDY
(1 row)
```

set_config将参数*setting_name*设置为*new_value*。如果*is_local*为true，新值仅适用于当前事务。如果要让新值适用于当前会话，请改用false。该函数对应于 SQL 命令SET。例如：

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
 off
```

```
(1 row)
```

表 9.37列出了允许用户以编程方式查询对象访问权限的函数。关于权限的更多信息，请参见第 5.7 节。

表 9.37. 访问权限查询函数

名称	返回类型	描述
has_table_privilege(*user*, *table*, *privilege*)	boolean	用户是否具有表权限
has_table_privilege(*table*, *privilege*)	boolean	用户是否具有表权限
has_database_privilege(*user*, *database*, *privilege*)	boolean	用户是否具有数据库权限
has_database_privilege(*database*, *privilege*)	boolean	用户是否具有数据库权限
has_function_privilege(*user*, *function*, *privilege*)	boolean	用户是否具有函数权限
has_function_privilege(*function*, *privilege*)	boolean	用户是否具有函数权限
has_language_privilege(*user*, *language*, *privilege*)	boolean	用户是否具有语言权限
has_language_privilege(*language*, *privilege*)	boolean	用户是否具有语言权限
has_schema_privilege(*user*, *schema*, *privilege*)	boolean	用户是否具有模式权限
has_schema_privilege(*schema*, *privilege*)	boolean	用户是否具有模式权限

has_table_privilege检查用户是否可以以某种方式访问表。用户可以通过名称或 ID（pg_user.usesysid）指定；如果省略该参数，则假定使用current_user。表可以通过名称或 OID 指定。（因此，has_table_privilege实际上有六种变体，可以通过参数的数量和类型来区分。）通过名称指定时，如有需要，可以用模式限定名称。
所需的访问权限类型由文本字符串指定，其值必须为SELECT、INSERT、UPDATE、DELETE、RULE、REFERENCES或TRIGGER之一。（不过字符串的大小写无关紧要。）一个例子是：

```
SELECT has_table_privilege('myschema.mytable', 'select');
```

has_database_privilege 检查用户是否可以以某种方式访问数据库。其参数形式与 has_table_privilege 类似。所需访问权限类型必须为 CREATE、TEMPORARY 或 TEMP（等同于 TEMPORARY）的某种组合。

has_function_privilege 检查一个用户能否以某种特定方式访问一个函数。其参数的可能形式与 has_table_privilege 类似。当通过文本字符串而不是 OID 指定函数时，允许的输入与 regprocedure 数据类型的相同。所需的访问权限类型当前必须求值为 EXECUTE。

has_language_privilege 检查一个用户能否以某种特定方式访问一个过程语言。其参数的可能形式与 has_table_privilege 类似。所需的访问权限类型当前必须求值为 USAGE。

has_schema_privilege 检查一个用户能否以某种特定方式访问一个模式。其参数的可能形式与 has_table_privilege 类似。所需的访问权限类型必须求值为 CREATE 或 USAGE。

要判断一个用户是否持有该权限的授权选项，可以在权限关键字后面追加 WITH GRANT OPTION；例如 'UPDATE WITH GRANT OPTION'。

表 9.38列出了判断某个特定对象是否可见的函数，其判断依据是当前模式搜索路径。例如，如果表所在的模式位于搜索路径中，并且在搜索路径的更前面没有同名表，就称该表可见。这等价于说，可以只通过表名引用该表，而不必显式地用模式限定。要列出所有可见表的名称：

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

表 9.38. 模式可见性查询函数

名称	返回类型	描述
pg_table_is_visible(*table_oid*)	boolean	表是否在搜索路径中可见
pg_type_is_visible(*type_oid*)	boolean	类型（或域）是否在搜索路径中可见
pg_function_is_visible(*function_oid*)	boolean	函数是否在搜索路径中可见
pg_operator_is_visible(*operator_oid*)	boolean	操作符是否在搜索路径中可见
pg_opclass_is_visible(*opclass_oid*)	boolean	操作符类是否在搜索路径中可见
pg_conversion_is_visible(*conversion_oid*)	boolean	转换是否在搜索路径中可见

pg_table_is_visible对表（或视图，或任何其他类型的pg_class条目）执行检查。pg_type_is_visible、pg_function_is_visible、pg_operator_is_visible、pg_opclass_is_visible和pg_conversion_is_visible分别对类型（和域）、函数、操作符、操作符类和转换执行同类可见性检查。对于函数和操作符，如果搜索路径中的一个对象在路径更早的位置没有同参数数据类型的同名对象，则该对象在搜索路径中可见。对于操作符类，要考虑名字和相关的索引访问方法。

所有这些函数都需要对象的 OID 来标识要检查的对象。如果想按名称测试一个对象，使用 OID 别名类型会很方便（regclass、regtype、regprocedure 或 regoperator），例如

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

注意，用这种方式测试一个未限定的名称没有多大意义—— 如果该名称根本能被识别出来，那它必然是可见的。

表 9.39列出了从系统目录中提取信息的函数。pg_get_viewdef、pg_get_ruledef、pg_get_indexdef、pg_get_triggerdef 和 pg_get_constraintdef 分别重构造视图、规则、索引、触发器或约束的创建命令。（注意，这是反编译的重构造，不是该命令的原始文本。）这些函数大多有两种变体，其中一种可以选择对结果进行“美化打印”。美化打印的格式更易读，但默认格式更有可能被 PostgreSQL 的未来版本以相同的方式解释；出于转储目的请避免使用美化打印的输出。把美化打印参数设为 false 得到的结果与完全没有该参数的变体相同。pg_get_expr 反编译单个表达式的内部形式，例如列的默认值。在检查系统目录的内容时它可能很有用。pg_get_userbyid 根据用户 ID 号提取用户名。

表 9.39. 系统目录信息函数

名称	返回类型	描述
pg_get_viewdef(*view_name*)	text	获取视图的 CREATE VIEW 命令（已废弃）
pg_get_viewdef(*view_name*, *pretty_bool*)	text	获取视图的 CREATE VIEW 命令（已废弃）
pg_get_viewdef(*view_oid*)	text	获取视图的 CREATE VIEW 命令
pg_get_viewdef(*view_oid*, *pretty_bool*)	text	获取视图的 CREATE VIEW 命令
pg_get_ruledef(*rule_oid*)	text	获取规则的 CREATE RULE 命令
pg_get_ruledef(*rule_oid*, *pretty_bool*)	text	获取规则的 CREATE RULE 命令
pg_get_indexdef(*index_oid*)	text	获取索引的 CREATE INDEX 命令
pg_get_indexdef(*index_oid*, *column_no*, *pretty_bool*)	text	获取索引的CREATE INDEX命令，或在*column_no*不为零时仅获取一个索引列的定义
pg_get_triggerdef(*trigger_oid*)	text	获取触发器的 CREATE [CONSTRAINT] TRIGGER 命令
pg_get_constraintdef(*constraint_oid*)	text	获取约束定义
pg_get_constraintdef(*constraint_oid*, *pretty_bool*)	text	获取约束定义
pg_get_expr(*expr_text*, *relation_oid*)	text	反编译表达式的内部形式，假定其中所有 Var 节点都引用第二个参数指定的关系
pg_get_expr(*expr_text*, *relation_oid*, *pretty_bool*)	text	反编译表达式的内部形式，假定其中所有 Var 节点都引用第二个参数指定的关系
pg_get_userbyid(*userid*)	name	获取具有给定 ID 的用户名

表 9.40中所示的函数提取先前用 COMMENT 命令存储的注释。如果找不到与指定参数匹配的注释，则返回空值。

表 9.40. 注释信息函数

名称	返回类型	描述
obj_description(*object_oid*, *catalog_name*)	text	获取数据库对象的注释
obj_description(*object_oid*)	text	获取数据库对象的注释（已废弃）
col_description(*table_oid*, *column_number*)	text	获取表列的注释

obj_description 的双参数形式返回数据库对象的注释，对象由其 OID 和所在系统目录的名称指定。例如，obj_description(123456,'pg_class') 会获取 OID 为 123456 的表的注释。obj_description 的单参数形式只需要对象 OID。该形式已被弃用，因为无法保证 OID 在不同系统目录之间唯一，因而可能返回错误的注释。

col_description返回表列的注释，列由所属表的 OID 和列号指定。由于列本身没有各自的 OID，obj_description不能用于表列。

9.14. 数组函数和操作符

表 9.41列出了可用于 array 类型的操作符。

表 9.41. array 操作符

操作符	描述	示例	结果
=	等于	ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]	t
<>	不等于	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	小于	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	大于	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	小于等于	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	大于等于	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
||	数组到数组串接	ARRAY[1,2,3] || ARRAY[4,5,6]	{1,2,3,4,5,6}
||	数组到数组串接	ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
||	元素到数组串接	3 || ARRAY[4,5,6]	{3,4,5,6}
||	数组到元素串接	ARRAY[4,5,6] || 7	{4,5,6,7}

参阅第 8.10 节获取有关数组操作符行为的更多细节。

表 9.42列出了可用于数组类型的函数。关于这些函数的更多讨论和例子，参见第 8.10 节。

表 9.42. array 函数

函数	array_cat(anyarray, anyarray)
返回类型	anyarray
描述	连接两个数组，NULL 输入返回 NULL
示例	array_cat(ARRAY[1,2,3], ARRAY[4,5])
结果	{1,2,3,4,5}
函数	array_append(anyarray, anyelement)
返回类型	anyarray
描述	在数组末尾追加一个元素，NULL 输入返回 NULL
示例	array_append(ARRAY[1,2], 3)

结果	{1,2,3}
函数	array_prepend(anyelement, anyarray)
返回类型	anyarray
描述	在数组开头追加一个元素，NULL 输入返回 NULL
示例	array_prepend(1, ARRAY[2,3])
结果	{1,2,3}
函数	array_dims(anyarray)
返回类型	text
描述	返回数组各维下界和上界的文本表示，NULL 输入产生一个错误
示例	array_dims(array[[1,2,3], [4,5,6]])
结果	[1:2][1:3]
函数	array_lower (anyarray, integer)
返回类型	integer
描述	返回所请求数组维度的下界，NULL 输入返回 NULL
示例	array_lower(array_prepend(0, ARRAY[1,2,3]), 1)
结果	0
函数	array_upper (anyarray, integer)
返回类型	integer
描述	返回所请求数组维度的上界，NULL 输入返回 NULL
示例	array_upper(ARRAY[1,2,3,4], 1)
结果	4
函数	array_to_string(anyarray, text)
返回类型	text
描述	使用指定的分隔符连接数组元素，NULL 输入返回 NULL
示例	array_to_string(array[1, 2, 3], '~^~')
结果	1~^~2~^~3
函数	string_to_array(text, text)
返回类型	text[]
描述	使用指定的分隔符把字符串拆分为数组元素，NULL 输入返回 NULL
示例	string_to_array('xx~^~yy~^~zz', '~^~')
结果	{xx,yy,zz}

9.15. 聚合函数

聚合函数从一组输入值计算单个结果。表 9.43展示了内置的聚合函数。聚合函数的特殊语法注意事项在第 4.2.7 节中说明。更多入门信息请参见第 2.7 节。

表 9.43. 聚合函数

函数	参数类型	返回类型	描述
avg(*expression*)	smallint, integer, bigint, real, double precision, numeric, or interval	整数类型参数为numeric，浮点参数为double precision，否则与参数数据类型相同	所有输入值的平均值（算术平均）
count(*)		bigint	输入值的数目
count(*expression*)	任意类型	bigint	*expression* 的值不为 null 的输入行数
max(*expression*)	任意数字、字符串或日期/时间类型	与参数类型相同	表达式在所有输入值中的最大值
min(*expression*)	任意数字、字符串或日期/时间类型	与参数类型相同	表达式在所有输入值中的最小值
stddev(*expression*)	smallint, integer, bigint, real, double precision, or numeric	浮点参数为 double precision，否则为 numeric	输入值的样本标准差
sum(*expression*)	smallint, integer, bigint, real, double precision, numeric, or interval	smallint或integer参数为bigint，bigint参数为numeric，浮点参数为double precision，否则与参数数据类型相同	表达式在所有输入值上的总和
variance(*expression*)	smallint, integer, bigint, real, double precision, or numeric	浮点参数为 double precision，否则为 numeric	输入值的样本方差（样本标准差的平方）

应当注意，除了 count 之外，这些函数在没有选中任何行时都返回空值。特别地，没有行时 sum 返回空值，而不是人们可能预期的零。必要时可以使用 coalesce 函数把空值替换为零。

注意

习惯于其他 SQL 数据库管理系统的用户可能会对 PostgreSQL 中某些聚合函数在聚合应用于整个表（换句话说，没有指定 WHERE 子句）时的性能特征感到惊讶。特别地，像

```
SELECT min(col) FROM sometable;
```

这样的查询会被 PostgreSQL 用整个表的顺序扫描来执行。其他数据库系统可能会把这种形式的查询优化为使用该列上的索引（如果可用的话）。类似地，在 PostgreSQL 中，聚合函数 max() 和 count() 应用于整个表时也总是需要顺序扫描。

PostgreSQL无法轻易实现这种优化，因为它还允许用户定义的聚合查询。由于min()、max()和count()是用聚合函数的通用 API 定义的，因此没有为在某些情况下特殊处理这些函数的执行提供机制。

幸运的是，min() 和 max() 有一个简单的变通方法。下面显示的查询与上面的查询等价，只是如果所讨论的列上存在 B-tree 索引，它可以利用该索引。

```
SELECT col FROM sometable ORDER BY col ASC LIMIT 1;
```

类似的查询（把上面查询中的 ASC 替换为 DESC）可以用于代替 max()）。

> 遗憾的是，没有类似的简单查询可以用来提高count()应用于整个表时的性能。

9.16. 子查询表达式

本节描述 PostgreSQL 中可用的 SQL 兼容的子查询表达式。本节中记录的所有表达式形式都返回布尔值（真/假）结果。

9.16.1. EXISTS

```
EXISTS ( subquery )
```

EXISTS的参数是一个任意的SELECT语句，或者说子查询。
系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，那么
EXISTS的结果就为“真”；如果子查询没有返回行，那么EXISTS的结果是“假”。

子查询可以引用外层查询的变量，这些变量在该子查询的任何一次计算中都起常量的作用。

这个子查询通常只是运行到能判断它是否可以返回至少一行为止，而不是等到全部结束。
在这里写任何有副作用的子查询都是不明智的（例如调用序列函数）；
这些副作用是否发生是很难判断的。

因为结果只取决于是否会返回行，而不取决于这些行的内容，
所以这个子查询的输出列表通常是无关紧要的。一个常用的编码习惯是用EXISTS(SELECT 1 WHERE ...)的形式写所有的EXISTS测试。不过这条规则有例外，例如那些使用INTERSECT的子查询。

下面这个简单的示例类似在col2上的一次内连接，但是它为每个 tab1的行最多生成一行输出，即使存在多个匹配tab2的行也如此：

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.16.2. IN

```
expression IN (subquery)
```

右手边是一个圆括号括起来的子查询，它必须恰好返回一列。左手边表达式会被求值，
并与子查询结果逐行比较。如果找到任意相等的子查询行，那么IN的结果就是“真”。
如果没有找到相等行，那么结果就是“假”（包括子查询不返回任何行的情况）。

请注意，如果左侧表达式得到空值，或者右侧没有相等的值且至少有一行得到空值，则
IN结构的结果将是空值，而不是假。这符合 SQL 对空值布尔组合的一般规则。

和EXISTS一样，假定子查询一定会完整运行并不明智。

```
(expression [, expression ...]) IN (subquery)
```

这种形式的 IN 的右边是一个带圆括号的子查询，它必须返回与左边列表中表达式数量完全相同的列。左边的表达式被求值并与子查询结果的每一行逐行比较。如果找到任何相等的子查询行，IN 的结果为“真”。如果没有找到相等的行（包括子查询不返回行的特殊情况），结果为“假”。

通常，表达式或者子查询行里的空值是按照 SQL 布尔表达式的一般规则进行组合的。如果两个行对应的成员都非空并且相等，那么认为这两行相等；如果任意对应成员为非空且不等，那么这两行不等；否则这样的行比较的结果是未知（空值）。如果所有行的结果要么是不等，要么是空值，并且至少有一个空值，那么IN的结果是空值。

9.16.3. NOT IN

expression NOT IN (*subquery*)

右手边是一个圆括号括起来的子查询，它必须恰好返回一列。左手边表达式会被求值，并与子查询结果逐行比较。如果只找到不相等的子查询行（包括子查询不返回任何行的情况），那么NOT IN的结果是“真”。如果找到任何相等行，则结果为“假”。

请注意，如果左侧表达式得到空值，或者右侧没有相等的值且至少有一行得到空值，则NOT IN结构的结果将是空值，而不是真。这符合 SQL 对空值布尔组合的一般规则。

和EXISTS一样，假定子查询一定会完整运行并不明智。

(*expression* [, *expression* ...]) NOT IN (*subquery*)

这种形式的 NOT IN 的右边是一个带圆括号的子查询，它必须返回与左边列表中表达式数量完全相同的列。左边的表达式被求值并与子查询结果的每一行逐行比较。
如果只找到不相等的子查询行（包括子查询不返回行的特殊情况），NOT IN 的结果为“真”。如果找到任何相等的行，结果为“假”。

通常，表达式或者子查询行里的空值是按照 SQL 布尔表达式的一般规则进行组合的。如果两个行对应的成员都非空并且相等，那么认为这两行相等；如果任意对应成员为非空且不等，那么这两行不等；否则这样的行比较的结果是未知（空值）。如果所有行的结果要么是不等，要么是空值，并且至少有一个空值，那么NOT IN的结果是空值。

9.16.4. ANY/SOME

expressionoperator ANY (*subquery*)
expression operator SOME (*subquery*)

右侧是一个用圆括号括起来的子查询，必须恰好返回一列。左侧表达式会被求值，并使用给定的*operator*与子查询结果的每一行进行比较。该操作符必须产生布尔结果。
如果得到任何真值结果，则ANY的结果为“真”。
如果没有找到真值结果（包括子查询不返回任何行的情况），则结果为“假”。

SOME是ANY的同义词。IN等价于= ANY。

注意，如果没有任何成功的比较，并且至少有一个右侧行的操作符结果为 null，则ANY结构的结果将是 null 而不是假。这符合 SQL 对空值布尔组合的正常规则。

和EXISTS一样，假定子查询一定会完整运行并不明智。

(*expression* [, *expression* ...]) *operator* ANY (*subquery*)
(*expression* [, *expression* ...]) *operator* SOME (*subquery*)

这种形式的 ANY 的右边是一个带圆括号的子查询，它必须返回与左边列表中表达式数量完全相同的列。左边的表达式使用给定的 *operator* 与子查询结果的每一行逐行比较。目前，逐行 ANY 结构中只允许使用 = 和 <> 操作符。相应地，如果找到相等或不等的行，ANY 的结果为“真”。如果没有找到这样的行（包括子查询不返回行的特殊情况），结果为“假”。

和通常一样，行中的空值按照 SQL 布尔表达式的正常规则进行组合。如果两个行的所有对应成员都非空且相等，则认为两行相等；如果任何对应成员非空且不相等，则两行不相等；否则该行比较的结果为未知（null）。如果至少有一个行的结果为 null，则ANY的结果不可能是假；它将是真或 null。

9.16.5. ALL

*expression*operator ALL (*subquery*)

右侧是一个用圆括号括起来的子查询，必须恰好返回一列。左侧表达式会被求值，并使用给定的 *operator*与子查询结果的每一行进行比较。该操作符必须产生布尔结果。
如果所有行都得到真（包括子查询不返回任何行的情况），则ALL的结果为“真”。
如果得到任何假值结果，则结果为“假”。

NOT IN等价于<> ALL。

注意，如果没有任何失败，但至少有一个右侧行的操作符结果为 null，则ALL结构的结果将是 null 而不是真。这符合 SQL 对空值布尔组合的正常规则。

和EXISTS一样，假定子查询一定会完整运行并不明智。

(*expression* [, *expression* ...]) *operator* ALL (*subquery*)

这种形式的 ALL 的右边是一个带圆括号的子查询，它必须返回与左边列表中表达式数量完全相同的列。左边的表达式使用给定的 *operator* 与子查询结果的每一行逐行比较。目前，逐行 ALL 查询中只允许使用 = 和 <> 操作符。相应地，如果所有子查询行都相等或都不相等（包括子查询不返回行的特殊情况），ALL 的结果为“真”。相应地，如果发现任何一行不相等或相等，结果为“假”。

和通常一样，行中的空值按照 SQL 布尔表达式的正常规则进行组合。如果两个行的所有对应成员都非空且相等，则认为两行相等；如果任何对应成员非空且不相等，则两行不相等；否则该行比较的结果为未知（null）。如果至少有一个行的结果为 null，则ALL的结果不可能是真；它将是假或 null。

9.16.6. 按行比较

(*expression* [, *expression* ...]) *operator* (*subquery*)

左边是标量表达式的列表。右边是一个带圆括号的子查询，
它必须返回与左边表达式数量完全相同的列。此外，子查询不能返回多于一行。
（如果它返回零行，结果视为空值。）左边被求值并与子查询结果的唯一一行逐行比较。目前，逐行比较中只允许使用 = 和 <> 操作符。相应地，如果这两行相等或不相等，结果为“真”。

和通常一样，行中的空值按照 SQL 布尔表达式的正常规则进行组合。如果两个行的所有对应成员都非空且相等，则认为两行相等；如果任何对应成员非空且不相等，则两行不相等；否则该行比较的结果为未知（null）。

9.17. 行和数组比较

本节描述几个特殊的结构，用于在值的组之间进行多重比较。
这些形式语法上和前面一节的子查询形式相关，但是不涉及子查询。涉及数组子表达式的形式是PostgreSQL的扩展；其余形式是SQL兼容的。
本节介绍的所有表达式形式都返回布尔（Boolean）结果（真/假）。

9.17.1. IN

```
expression IN (value[, ...])
```

右侧是一个用圆括号括起来的标量表达式列表。
如果左侧表达式的结果等于任何一个右侧表达式的结果，则结果为“真”。这是

```
expression = value1
OR
expression = value2
OR
...
```

的简写形式。

注意，如果左侧表达式产生 null，或者不存在相等的右侧值且至少有一个右侧表达式产生 null，则`IN`结构的结果将是 null 而不是假。这符合 SQL 对空值布尔组合的正常规则。

9.17.2. NOT IN

```
expression NOT IN (value[, ...])
```

右侧是一个用圆括号括起来的标量表达式列表。
如果左侧表达式的结果与所有右侧表达式的结果都不相等，则结果为“真”。这是

```
expression <> value1
AND
expression <> value2
AND
...
```

的简写形式。

注意，如果左侧表达式产生 null，或者不存在相等的右侧值且至少有一个右侧表达式产生 null，则`NOT IN`结构的结果将是 null，而不是人们可能天真期望的真。这符合 SQL 对空值布尔组合的正常规则。

> **提示**
>
> `x NOT IN y`在任何情况下都等价于`NOT (x IN y)`。不过，对新手来说，使用`NOT IN`时 null 值比使用`IN`时更容易造成困扰。最好尽可能用肯定的形式表达你的条件。

9.17.3. ANY/SOME（数组）

```
expressionoperator ANY (array expression)
expression operator SOME (array expression)
```

右侧是一个用圆括号括起来的表达式，它必须产生一个数组值。左侧表达式会被求值，
并使用给定的*operator*与数组的每个元素进行比较。该操作符必须产生布尔结果。
如果得到任何真值结果，则`ANY`的结果为“真”。
如果没有找到真值结果（包括数组有零个元素的情况），则结果为“假”。

SOME是ANY的同义词。

9.17.4. ALL（数组）

expressionoperator ALL (*array expression*)

右侧是一个用圆括号括起来的表达式，它必须产生一个数组值。左侧表达式会被求值，并使用给定的*operator*与数组的每个元素进行比较。该操作符必须产生布尔结果。如果所有比较都得到真（包括数组有零个元素的情况），则ALL的结果为“真”。如果找到任何假值结果，则结果为“假”。

9.17.5. 按行比较

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

每一边都是标量表达式的列表；两个列表的长度必须相同。对两边求值后逐行比较。目前，逐行比较中只允许使用 = 和 <> 操作符。相应地，如果这两行相等或不相等，结果为“真”。

和通常一样，行中的空值按照 SQL 布尔表达式的正常规则进行组合。如果两个行的所有对应成员都非空且相等，则认为两行相等；如果任何对应成员非空且不相等，则两行不相等；否则该行比较的结果为未知（null）。

第 10 章 类型转换

SQL语句可能有意或无意地要求在同一表达式中混用不同的数据类型。
PostgreSQL提供了丰富的机制来求值这类混合类型表达式。

在很多情况下，用户不需要了解类型转换机制的细节。但 PostgreSQL 所做的隐式转换会影响查询的结果。必要时，用户或程序员可以用显式类型转换来定制这些结果。

本章介绍PostgreSQL的类型转换机制和约定。
有关特定数据类型以及允许使用的函数和操作符的更多信息，请参阅第 8 章和第 9 章中的相关章节。

10.1. 概述

SQL 是一种强类型语言。也就是说，每个数据项都有一个相关联的数据类型，它决定了该数据项的行为和允许的用法。PostgreSQL 拥有一个可扩展的类型系统，比其他 SQL 实现更加通用和灵活。因此，PostgreSQL 中大部分类型转换行为应当由一般规则而不是ad hoc（特定）启发式规则支配，以使混合类型表达式即使在有用户定义类型时也有意义。

PostgreSQL 的扫描器/解析器只把词法元素解码成五种基本类别：整数、浮点数、字符串、名字和关键字。大多数非数值类型的常量最初被归类为字符串。SQL 语言定义允许用字符串指定类型名，这一机制可以用来让解析器走上正确的路径。例如，查询

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

中的两个文字常量分别是 `text` 类型和 `point` 类型。如果没有为字符串文字指定类型，最初会先赋予占位类型 `unknown`，并在后续阶段按下面描述的方式解析。

在 PostgreSQL 解析器中有四种基本的 SQL 结构需要专门的类型转换规则：

操作符

PostgreSQL 允许带前缀和后缀一元（单参数）操作符的表达式，也允许二元（双参数）操作符的表达式。

函数调用

PostgreSQL 类型系统的很大一部分是围绕一套丰富的函数构建的。函数调用可以有一个或多个参数。由于 PostgreSQL 允许函数重载，仅凭函数名并不能唯一标识要调用的函数；解析器必须根据所提供参数的数据类型选择正确的函数。

值存储

SQL的`INSERT`和`UPDATE`语句会把表达式的结果放入表中。
语句中的表达式必须与目标列的类型匹配，并且在必要时转换为目标列类型。

`UNION`、`CASE`和`ARRAY`结构

由于联合起来的 `SELECT` 语句的所有查询结果必须出现在一组列中，每个 `SELECT` 子句结果的类型必须匹配并转换成统一的集合。类似地，`CASE` 结构的各分支表达式必须转换成公共类型，使 `CASE` 表达式整体有已知的输出类型。`ARRAY` 结构也是如此。

系统目录保存了哪些数据类型之间存在哪些类型转换，以及如何执行这些转换的信息。
用户还可以使用CREATE CAST命令添加额外的类型转换。（这通常与定义新数据类型一并完成。
内置类型之间的类型转换集合经过精心设计，最好不要改动。）

解析器还提供了一个附加的启发式规则，以便对SQL标准类型做出更好的猜测。
系统定义了若干基本的类型分类：boolean、numeric、string、bitstring、datetime、timespan、geometric、network以及用户定义类型。除用户定义分类外，
每个分类都有一个或多个首选类型，存在歧义时会优先选择它们。在用户定义分类中，
每个类型自己的首选类型就是它自己。因此，当存在多个可能的内置类型时，
有歧义的表达式（即存在多个候选解析方案的表达式）常常仍能被解析；
但当存在多个用户定义类型选择时，则会报错。

所有类型转换规则都以以下几个原则为基础：

- 隐式转换绝不应产生令人意外或无法预测的结果。
- 解析器没有关于用户定义类型的先验知识，这类类型在类型层次结构中应当处于“更高”的位置。在混合类型的表达式中，原生类型应当总是被转换为用户定义类型（当然，仅在确有必要转换时）。
- 用户定义类型彼此之间没有关联。目前，PostgreSQL除了对内置类型的硬编码启发式规则，以及基于可用函数和类型转换的隐式关系外，并没有关于类型之间关系的信息可用。
- 如果查询不需要隐式类型转换，解析器和执行器就不应承担额外开销。也就是说，如果查询本身书写正确且类型已经匹配，那么执行时不应让解析器额外耗时，也不应在查询中引入不必要的隐式类型转换调用。

  此外，如果某个查询通常需要先对参数做隐式类型转换才能调用某个函数，而此后用户又定义了一个参数类型正确的新函数，那么解析器应改为使用这个新函数，而不再通过隐式转换去调用旧函数。

10.2. 操作符

操作符表达式所引用的具体操作符按照下述过程确定。注意，
该过程还会间接受到相关操作符优先级的影响。详见第 4.1.6 节。

操作符类型解析

1. 从系统目录pg_operator中选取要考虑的操作符。
   如果使用的是未带模式限定的操作符名（通常如此），
   则考虑当前搜索路径中可见且名称和参数个数匹配的操作符（见第 5.8.3 节）。如果给出了限定名，则只考虑指定模式中的操作符。

   - (可选的) 如果搜索路径中找到多个参数类型完全相同的操作符，只考虑路径中最早出现的那一个。参数类型不同的操作符则不受搜索路径位置影响，处于同等地位。

2. 检查是否存在一个恰好接受输入参数类型的操作符。
   如果存在（在被考虑的操作符集合中至多只有一个精确匹配），就使用它。

   - (可选的) 如果二元操作符调用的一个参数是unknown类型，则在本次检查中假定它与另一个参数类型相同。涉及unknown的其他情况在这一步都不会找到匹配。

3. 寻找最佳匹配。

a. 丢弃那些输入类型不匹配，且也无法通过隐式类型转换实现匹配的候选操作符。为此，假定unknown字面量可以转换为任何类型。如果只剩一个候选操作符，就使用它；否则继续下一步。

b. 遍历所有候选操作符，保留那些在输入类型上精确匹配数最多的候选项。（在这一步中，域被视为与其基础类型相同。）如果没有任何候选项存在精确匹配，则保留全部候选项。如果只剩一个候选操作符，就使用它；否则继续下一步。

c. 遍历所有候选操作符，在需要类型转换的位置上，保留接受首选类型的位置数最多的候选项。如果没有候选项接受首选类型，则保留全部候选项。如果只剩一个候选操作符，就使用它；否则继续下一步。

d. 如果有任何输入参数是unknown，就检查剩余候选操作符在这些参数位置上接受哪些类型分类。在每个位置上，如果有任何候选项接受string分类，就选择该分类。（这种偏向字符串的做法是合适的，因为unknown类型的字面量看起来像字符串。）否则，如果所有剩余候选项都接受同一种类型分类，就选择该分类；否则失败，因为在缺少更多线索的情况下无法推断出正确选择。然后丢弃不接受所选类型分类的候选项。此外，如果有任何候选项接受该分类中的首选类型，就丢弃在该参数上接受非首选类型的候选项。

e. 如果只剩一个候选操作符，就使用它。如果没有候选操作符或者不止一个候选操作符剩余，则失败。

下面是一些示例。

例 10.1. 指数操作符类型解析

目录中只定义了一个指数操作符，它接受类型为 double precision 的参数。扫描器为以下查询表达式的两个参数赋予初始类型 integer：

```
SELECT 2 ^ 3 AS "exp";

 exp
-----
   8
(1 row)
```

因此，解析器会对两个操作数进行类型转换，该查询等价于：

```
SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS
 "exp";
```

例 10.2. 字符串连接操作符类型解析

处理字符串类型以及复杂的扩展类型时，常常会使用类似字符串的语法。未指定类型的字符串会与可能的操作符候选项进行匹配。

一个参数未指定类型的例子：

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
```

```
------------------
 abcdef
(1 row)
```

在这种情况下，解析器会查看是否存在一个两边参数都接受text的操作符。既然存在，它就会假定第二个参数应解释为text类型。

下面是对未指定类型进行连接的例子：

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

这里对使用哪种类型没有初始提示，因为查询中没有指定任何类型。因此，解析器会查找所有候选操作符，并发现既有接受字符串分类输入的候选项，也有接受位串分类输入的候选项。由于在可用时优先选择字符串分类，所以会选中该分类，再使用字符串的首选类型text作为解析这些unknown类型字面量的具体类型。

例 10.3. 绝对值与阶乘操作符的类型解析

The PostgreSQL 运算符目录中前缀运算符 @ 有若干条目，它们为各种数值数据类型实现绝对值操作。其中一个条目面向 float8 类型，它是数值类别中的首选类型。因此，PostgreSQL 遇到非数值输入时会使用那个条目：

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

这里，系统在应用所选操作符之前，先执行了一次从 text 到 float8 的隐式转换。我们可以验证用的是 float8 而不是其他类型：

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

另一方面，后缀运算符 ！（阶乘）只为整数数据类型定义，不为 float8 定义。因此，如果我们用！尝试类似的情形，会得到：

```
SELECT '20' ! AS "factorial";

ERROR:  operator is not unique: "unknown" !
HINT:  Could not choose a best candidate operator. You may need to add
 explicit
type casts.
```

这是因为系统无法决定几个可能的！运算符中应该首选哪个。我们可以用显式转换来帮它解决：

```
SELECT CAST('20' AS int8) ! AS "factorial";

      factorial
```

```
---------------------
 2432902008176640000
(1 row)
```

10.3. 函数

函数调用所引用的具体函数按照下述过程确定。

函数类型解析

1. 从系统目录pg_proc中选取要考虑的函数。如果使用的是未带模式限定的函数名，则考虑当前搜索路径中可见且名称和参数个数匹配的函数（见第 5.8.3 节）。如果给出了限定名，则只考虑指定模式中的函数。

   - (可选的) 如果搜索路径中找到多个参数类型完全相同的函数，只考虑路径中最早出现的那一个。参数类型不同的函数则不受搜索路径位置影响，处于同等地位。

2. 检查是否存在一个恰好接受输入参数类型的函数。
   如果存在（在被考虑的函数集合中至多只有一个精确匹配），就使用它。（涉及unknown的情况在这一步永远找不到匹配。）

3. 如果没有找到精确匹配，就检查该函数调用是否看起来像一种简单的类型转换请求。
   这种情况发生在：函数调用只有一个参数，且函数名与某个数据类型的（内部）名称相同。
   此外，函数参数必须是unknown类型字面量，
   或者是一种与该命名数据类型二进制兼容的类型。满足这些条件时，
   函数参数会被直接转换为该命名数据类型，而不会发生实际的函数调用。

4. 寻找最佳匹配。

   a. 丢弃那些输入类型不匹配，且也无法通过隐式类型转换实现匹配的候选函数。为此，
      假定unknown字面量可以转换为任何类型。如果只剩一个候选函数，就使用它；
      否则继续下一步。

   b. 遍历所有候选操作符，保留那些在输入类型上精确匹配数最多的候选项。
      （在这一步中，域被视为与其基础类型相同。）如果没有任何候选项存在精确匹配，
      则保留全部候选项。如果只剩一个候选操作符，就使用它；否则继续下一步。

   c. 遍历所有候选操作符，在需要类型转换的位置上，
      保留接受首选类型的位置数最多的候选项。如果没有候选项接受首选类型，
      则保留全部候选项。如果只剩一个候选操作符，就使用它；否则继续下一步。

   d. 如果有任何输入参数是unknown，
      就检查剩余候选操作符在这些参数位置上接受哪些类型分类。在每个位置上，
      如果有任何候选项接受string分类，就选择该分类。
      （这种偏向字符串的做法是合适的，因为unknown类型的字面量看起来像字符串。）
      否则，如果所有剩余候选项都接受同一种类型分类，就选择该分类；否则失败，
      因为在缺少更多线索的情况下无法推断出正确选择。
      然后丢弃不接受所选类型分类的候选项。此外，
      如果有任何候选项接受该分类中的首选类型，
      就丢弃在该参数上接受非首选类型的候选项。

   e. 如果只剩一个候选操作符，就使用它。
      如果没有候选操作符或者不止一个候选操作符剩余，则失败。

注意，操作符类型解析和函数类型解析的“最佳匹配”规则是完全相同的。下面是一些示例。

例 10.4. round 函数参数类型解析

只有一个接受两个参数的round函数；它的第一个参数类型是numeric，第二个参数类型是integer。因此下面的查询会自动把类型为integer的第一个参数转换为numeric：

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

该查询实际上会被解析器改写成：

```
SELECT round(CAST (4 AS numeric), 4);
```

由于带小数点的数字常量最初会被赋予numeric类型，下面的查询不需要类型转换，因此可能稍微更高效一些：

```
SELECT round(4.0, 4);
```

例 10.5. substr 函数类型解析

存在多个substr函数，其中一个接受text和integer。如果以未指定类型的字符串常量调用，系统会选择接受首选类型分类string参数的候选函数（也就是text类型的那个）。

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

如果字符串被声明为varchar类型，例如它来自表中的某一列，那么解析器会尝试把它转换为text：

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

解析器会把它改写成如下形式：

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **注意**
>
> 解析器从pg_cast目录得知text与varchar二进制兼容，这意味着把其中一种类型传给接受另一种类型的函数时，无需做任何物理转换。因此，这种情况下实际上不会插入类型转换调用。

如果用integer类型的参数调用该函数，解析器会尝试把它转换为 text：

```
SELECT substr(1234, 3);

 substr
--------
     34
(1 row)
```

这实际上会执行为

```
SELECT substr(CAST (1234 AS text), 3);
```

这种自动转换之所以能成功，是因为存在一个可隐式调用的从integer 到text的类型转换。

10.4. 值存储

要插入表中的值会按照以下步骤转换为目标列的数据类型。

值存储的类型转换

1. 检查是否与目标类型精确匹配。
2. 否则，尝试把表达式转换为目标类型。如果这两种类型之间存在已登记的类型转换，这就会成功。如果表达式是unknown类型字面量，则会把字符串字面量的内容送入目标类型的输入转换例程。
3. 如果目标是定长类型（例如声明了长度的 char 或 varchar），则尝试为目标类型寻找一个尺寸调整函数。尺寸调整函数是一个与类型同名的函数，接受两个参数，第一个是该类型，第二个是 integer 类型，并返回同一类型。如果找到了这样的函数，就应用它，把列的声明长度作为第二个参数传递。

例 10.6. character 的存储类型转换

对于声明为character(20)的目标列，下面的语句表明存储值会被调整到正确的长度：

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
 abcdef               |           20
(1 row)
```

这里实际发生的是，两个unknown类型字面量默认都被解析为text，从而使||操作符被解析为text的连接操作。然后，操作符得到的text结果被转换为bpchar（“用空格填充的字符”，blank-padded char，即character数据类型的内部名称），以匹配目标列类型。（由于text和bpchar二进制兼容，这一转换不会插入任何真实的函数调用。）最后，系统从系统目录中找到尺寸调整函数bpchar(bpchar, integer)，并把它应用到操作符结果以及存储列长度上。这个特定于类型的函数会执行所需的长度检查并补齐填充空格。

10.5. UNION、CASE和ARRAY结构

SQL 的UNION结构必须让可能不同的类型彼此匹配，以形成单一结果集。该解析算法会分别应用到联合查询的每个输出列。INTERSECT和EXCEPT结构以与UNION相同的方式解析不同类型。CASE和ARRAY构造使用完全相同的算法来匹配其组成表达式并选择结果数据类型。

UNION、CASE和ARRAY结构的类型解析

1. 如果所有输入都是unknown类型，则解析为text类型（字符串分类的首选类型）。否则，忽略unknown输入。
2. 如果非unknown输入并非全都属于同一类型分类，则失败。
3. 选择第一个满足如下条件的非unknown输入类型：它是该分类中的首选类型，或者所有非unknown输入都可以隐式转换为它。
4. 把所有输入都转换为所选类型。

下面是一些示例。

例 10.7. 未充分指定类型的 UNION 中的类型解析

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

这里，unknown类型字面量'b'会被解析为text类型。

例 10.8. 简单 UNION 中的类型解析

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

字面量1.2的类型是numeric，而integer值1可以隐式转换为numeric，因此使用该类型。

例 10.9. 次序对调的 UNION 中的类型解析

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

这里，由于real类型不能隐式转换为integer，而integer可以隐式转换为real，因此UNION结果类型被解析为real。

第 11 章 索引

索引是增强数据库性能的一种常用手段。
索引让数据库服务器能够比没有索引时快得多地找到并检索特定的行。
但索引也会给整个数据库系统带来开销，因此应该合理地使用它们。

11.1. 简介

假设我们有一个类似这样的表：

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

而应用程序需要执行大量如下形式的查询

```
SELECT content FROM test1 WHERE id = constant;
```

如果事先没有任何准备，系统就必须逐行扫描整个test1 表才能找出所有匹配的项。如果test1 中有大量行，而这种查询只会返回很少几行（可能只有零行或一行），那么这显然是一种低效的方法。但如果系统被指示在 id 列上维护一个索引，它就可以使用一种更高效的方法来定位匹配的行。例如，它可能只需深入搜索树中的少数几层。

大多数非小说类书籍都采用类似的做法：读者经常查阅的术语和概念被收集在书末的字母顺序索引中。有兴趣的读者可以相对快速地扫描索引并翻到相应的页，
而不必为了找到感兴趣的内容通读整本书。
正如预见读者最可能查阅哪些条目是作者的任务一样，
预见哪些索引会带来好处也是数据库程序员的任务。

如上所述，要在 id 列上创建索引，可以使用下面的命令：

```
CREATE INDEX test1_id_index ON test1 (id);
```

名称 test1_id_index 可以自由选择，但你应该选一个能让你事后记起这个索引用途的名字。

要删除一个索引，使用 DROP INDEX 命令。索引可以在任何时候添加到表或从表中删除。

索引一旦创建，就不再需要进一步的干预：系统会在表被修改时更新索引，
并且当它认为使用索引比顺序表扫描更高效时，就会在查询中使用索引。但你可能需要定期运行 ANALYZE 命令来更新统计信息，使查询规划器能够做出有根据的决策。关于如何查明索引是否被使用，以及规划器在何时、为何可能选择不使用索引，请参见第 13 章。

索引也可以让带有搜索条件的 UPDATE 和DELETE 命令受益。此外，索引还可以用于连接查询。因此，在作为连接条件一部分的列上定义的索引，可以显著加快带连接的查询。

创建索引后，系统必须让索引与表保持同步。这会给数据操作操作带来开销。因此，
非必需或根本不会被用到的索引应该被删除。注意，
一条查询或数据操作命令对每个表最多只能使用一个索引。

11.2. 索引类型

PostgreSQL 提供几种索引类型：B-树、R-树、GiST 和 Hash。每种索引类型使用不同的算法，分别最适合不同类型的查询。默认情况下，CREATE INDEX 命令会创建B-树索引，它适合最常见

的情况。B-树可以处理对能排成某种顺序的数据的等值和范围查询。特别地，
只要被索引列参与的比较使用下列操作符之一，PostgreSQL 查询规划器就会考虑使用 B-树索引：<, <=, =, >=, >

如果模式被锚定在字符串的开头，例如 `col LIKE 'foo%'` 或 `col ~ '^foo'`，而不是`col LIKE '%bar'`，那么优化器也可以对涉及模式匹配操作符 `LIKE`、`ILIKE`、`~` 和 `~*` 的查询使用B-树索引。但是，如果你的服务器不使用 C 区域，就需要用特殊的操作符类来创建索引。参见下文的第 11.6 节。

R-树索引适合对空间数据的查询。要创建 R-树索引，使用如下形式的命令

```
CREATE INDEX name ON table USING RTREE (column);
```

只要被索引列参与的比较使用下列操作符之一，PostgreSQL 查询规划器就会考虑使用R-树索引：`<<`, `&<`, `&>`, `>>`, `@`, `~=`, `&&`（这些操作符的含义请参阅第 9.9 节。）

Hash 索引只能处理简单的等值比较。只要被索引列参与的比较使用 = 操作符，查询规划器就会考虑使用hash 索引。下面的命令用于创建 hash 索引：

```
CREATE INDEX name ON table USING HASH (column);
```

> **注意**
>
> 测试表明 PostgreSQL 的 hash 索引的性能并不比 B-树索引好，而且 hash 索引的索引尺寸和构建时间都要糟糕得多。由于这些原因，目前不鼓励使用 hash 索引。

B-树索引方法是 Lehman-Yao 高并发 B-树的一种实现。R-树索引方法使用 Guttman 的二次分裂算法实现标准 R-树。hash 索引方法是 Litwin 线性散列的一种实现。我们提到这些算法只是为了说明所有这些索引方法都是完全动态的，不需要定期优化（例如，
与静态散列方法的情况不同）。

11.3. 多列索引

索引可以定义在多个列上。例如，如果你有一个这样形式的表：

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

（比如说，你把你的 `/dev` 目录保存在数据库里.....）而且你经常做这样的查询

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

那么在 `major` 和`minor` 列上一起定义一个索引可能是合适的，例如

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

当前，只有 B-树和 GiST 实现支持多列索引。最多可以指定 32 个列。（在构建 PostgreSQL 时可以改变这个限制；参见文件 `pg_config_manual.h`。）

查询规划器可以把多列索引用于涉及索引定义中最左列加上它右侧任意多个连续列（不能有空档）的查询。例如，一个定义在`(a, b, c)` 上的索引可以用于涉及`a`、`b`、`c` 全部的查询，或涉及

a 和 b 两者的查询，或只涉及 a 的查询，但不能用于其他组合。（在涉及 a 和 c 的查询中，规划器可以选择把索引用于 a，而把c 当作普通的未索引列。）当然，每个列都必须与适合该索引类型的操作符一起使用；涉及其他操作符的子句不会被考虑。

只有当涉及被索引列的子句是用 AND 连接时，才能使用多列索引。例如，

```
SELECT name FROM test2 WHERE major = constant OR minor = constant;
```

不能使用上面定义的索引 `test2_mm_idx` 来同时查找两个列。（不过，它可以用来只查找 major 列。）

多列索引应该节制使用。大多数情况下，单个列上的索引就足够了，而且节省空间和时间。
除非表的使用方式极端特定，超过三个列的索引不太可能有帮助。

11.4. 唯一索引

索引还可以用于强制列值唯一，或者强制多个列组合值唯一。

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

当前，只有 B-树索引可以声明为唯一。

当一个索引被声明为唯一时，就不允许存在多个具有相同索引值的表行。空值不被视为相等。
多列唯一索引只会拒绝那种在多行中所有索引列都相等的情况。

当为表定义唯一约束或主键时，PostgreSQL会自动创建一个唯一索引。
该索引覆盖构成主键或唯一约束的列（必要时是多列索引），并且正是它在强制执行该约束。

> **注意**
>
> 向表中添加唯一约束的首选方式是ALTER TABLE ... ADD CONSTRAINT。用索引来强制唯一约束可以被认为是一种不应被直接访问的实现细节。不过应当知道，
> 无需手工在唯一列上创建索引；那样做只会重复自动创建的索引。

11.5. 表达式索引

索引列不一定非得是底层表中的某一列，
也可以是一个由表中一列或多列计算得到的函数或标量表达式。
这个特性有助于基于计算结果来快速访问表。

例如，进行大小写不敏感比较的一种常见方式，是使用lower函数：

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

如果为lower(col1)函数的结果定义了索引，这个查询就可以使用该索引：

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

如果我们把这个索引声明为UNIQUE，那么它不仅会阻止创建那些col1
值仅在大小写上不同的行，也会阻止创建col1值完全相同的行。因此，
表达式索引可用于强制那些无法定义为简单唯一约束的约束。

再例如，如果你经常执行如下查询：

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John
 Smith';
```

那么创建一个如下索引可能是值得的：

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name)
);
```

正如第二个示例所示，`CREATE INDEX`命令的语法通常要求对索引表达式写上圆括号。当表达式只是一个函数调用时，则可以像第一个示例那样省略这些圆括号。

维护索引表达式的代价相对较高，因为在插入每一行时以及每次更新时都必须计算派生表达式。因此，只有当能使用该索引的查询非常频繁时，才应该使用表达式索引。

11.6. 操作符类

索引定义可以为索引的每一列指定一个操作符类。

```
CREATE INDEX name ON table (column opclass [, ...]);
```

操作符类确定索引针对该列使用哪些操作符。例如，建立在类型`int4`上的 B-树索引会使用`int4_ops`类；该操作符类包含针对`int4`类型值的比较函数。实际中，列数据类型的默认操作符类通常就足够了。之所以需要操作符类，主要是因为对于某些数据类型，可能存在不止一种有意义的索引行为。例如，可能希望按绝对值或实部对复数数据类型排序。可以为该数据类型定义两个操作符类，并在创建索引时选择合适的类。

除了默认操作符类之外，还有一些内置操作符类：

- 操作符类`text_pattern_ops`、`varchar_pattern_ops`、`bpchar_pattern_ops`和`name_pattern_ops`分别支持类型`text`、`varchar`、`char`和`name`上的 B-树索引。它们与普通操作符类的区别在于，值是严格按字符逐个比较，而不是按照区域设置相关的排序规则比较。因此，当服务器未使用标准“C”区域设置时，这些操作符类适用于涉及模式匹配表达式（`LIKE`或 POSIX 正则表达式）的查询。例如，你可以像这样为一个`varchar`列建立索引：

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  如果你使用的是 C 区域设置，则可以改为创建一个使用默认操作符类的索引，它对于模式匹配查询仍然有用。另外请注意，如果你希望涉及普通比较的查询能够使用索引，就应该创建一个使用默认操作符类的索引。这类查询无法使用*xxx*`_pattern_ops`操作符类。同一列上允许创建多个使用不同操作符类的索引。

下面的查询会显示所有已定义的操作符类：

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcamid = am.oid
    ORDER BY index_method, opclass_name;
```

它可以扩展为显示每个类中包含的所有操作符：

```
SELECT am.amname AS index_method,
```

```
       opc.opcname AS opclass_name,
       opr.oprname AS opclass_operator
    FROM pg_am am, pg_opclass opc, pg_amop amop, pg_operator opr
    WHERE opc.opcamid = am.oid AND
          amop.amopclaid = opc.oid AND
          amop.amopopr = opr.oid
    ORDER BY index_method, opclass_name, opclass_operator;
```

11.7. 部分索引

部分索引是建立在表的一个子集上的索引；
这个子集由一个条件表达式定义（称为部分索引的谓词）。
索引中只包含满足该谓词的表行的项。

使用部分索引的一个主要原因是避免索引常见值。
由于搜索常见值（即占全部表行百分之几以上的值）的查询反正也不会使用索引，
因此完全没有必要把这些行保留在索引里。这会减小索引尺寸，
从而加快那些确实会使用该索引的查询。它也会加快很多表更新操作，
因为索引并不需要在所有情况下都更新。例 11.1展示了这种思路的一种可能应用。

例 11.1. 建立一个部分索引以排除常见值

假设你把 Web 服务器访问日志存储在数据库中。大多数访问来自你所在组织的 IP 地址范围，但也有一些来自其他地方（例如使用拨号连接的员工）。如果你按 IP 搜索时主要关心外部访问，那么你大概没有必要为对应于组织内子网的 IP 范围建索引。

假设有一个如下表：

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

要创建一个适合这个例子的部分索引，可使用如下命令：

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND client_ip < inet
 '192.168.100.255');
```

一个能够使用此索引的典型查询是：

```
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip =
 inet '212.78.10.32';
```

一个无法使用此索引的查询是：

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

可以看到，这类部分索引要求常见值事先可知。如果值的分布是内在的（由应用的本质决定）
且静态的（不随时间变化），这并不困难；但如果常见值只是由巧合的数据装载造成的，
这可能就需要大量的维护工作。

部分索引的另一种可能用途，是把典型查询工作负载不感兴趣的值排除在索引之外，如例 11.2所示。这样会得到与上面相同的好处，但也意味着这些“不感兴趣”的值无法通过该索引访问，

即使在那种情况下索引扫描可能是有利的。显然，
为这种场景建立部分索引需要大量的谨慎和实验。

例 11.2. 建立一个部分索引以排除不感兴趣的值

如果你有一张表，其中同时包含已开账单和未开账单的订单，
而未开账单订单只占整张表的一小部分，却是最常被访问的那些行，
那么你可以通过只为未开账单的行创建索引来提升性能。创建该索引的命令如下：

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

一个可能会使用该索引的查询是：

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

不过，这个索引也可以用于那些完全不涉及`order_nr`的查询，例如：

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

这不像在`amount`列上建立部分索引那样高效，因为系统必须扫描整个索引。不过，
如果未开账单的订单相对较少，那么即使只是用这个部分索引来找出未开账单订单，
也可能是值得的。

请注意，下面这个查询不能使用该索引：

```
SELECT * FROM orders WHERE order_nr = 3501;
```

因为订单 3501 可能属于已开账单订单，也可能属于未开账单订单。

例 11.2也说明了：索引列和谓词中使用的列不必一致。PostgreSQL支持带任意谓词的部分索引，只要其中只涉及正在建立索引的那张表的列。不过要记住，
谓词必须与那些希望从该索引受益的查询中使用的条件相匹配。更准确地说，
只有当系统能够识别出查询的`WHERE`条件在数学上蕴含该索引的谓词时，
部分索引才能用于该查询。PostgreSQL并没有一个复杂的定理证明器，
来识别那些写法不同但数学上等价的表达式。（不仅构建这样一个通用定理证明器极其困难，
而且它很可能也会慢到失去实际用途。）系统可以识别简单的不等式蕴含，例如“x < 1”蕴含“x < 2”；否则，谓词条件必须与查询`WHERE`条件的某一部分完全匹配，否则索引不会被识别为可用。

部分索引的第三种可能用途，甚至不要求索引被查询使用。这里的思路是像例 11.3那样，在表的一个子集上创建唯一索引。这样就能在满足索引谓词的那些行之间强制唯一性，
而不会约束不满足谓词的行。

例 11.3. 建立一个部分唯一索引

假设我们有一张描述测试结果的表。我们希望确保对于给定的测试对象和目标组合，
只有一条“成功”记录，但可以有任意多条“失败”记录。实现方法之一如下：

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);
```

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject,
 target)
    WHERE success;
```

当成功测试很少而失败测试很多时，这是一种特别高效的方法。

最后，部分索引还可以用来影响系统的查询计划选择。同样，
分布异常的数据集可能导致系统在实际上不应该使用索引的时候选择使用它。在这种情况下，
可以把索引建成对那个有问题的查询不可用。通常，
PostgreSQL会对索引使用作出合理选择（例如，它会在检索常见值时避免使用索引，
因此前面的例子实际上只是节省索引空间，而不是为了避免使用索引所必需的），
如果出现明显错误的计划选择，那就应当提交 bug 报告。

请记住，建立部分索引意味着你至少与查询规划器一样了解情况，
尤其是你知道索引何时可能是有利的。形成这种认识需要经验，以及对
PostgreSQL中索引工作方式的理解。在大多数情况下，部分索引相比普通索引的优势都很小。

关于部分索引的更多信息可参见[ston89b]、[olson93]和[seshadri95]。

11.8. 检查索引使用情况

尽管PostgreSQL中的索引不需要维护或调优，
但检查真实查询工作负载到底实际使用了哪些索引，仍然很重要。
检查某个单独查询的索引使用情况，可以使用EXPLAIN命令；它在这方面的应用见第 13.1 节。也可以像第 23.2 节中描述的那样，在运行中的服务器上收集索引使用的总体统计信息。

很难给出一个确定应该创建哪些索引的通用流程。前面各节的示例已经展示了若干典型情形。
通常需要做大量实验。下面这部分会给出一些相关提示：

- 一定要先运行ANALYZE。这个命令会收集表中值分布的统计信息。估计查询会返回多少行，
  需要这些信息；而规划器要为每种可能的查询计划分配现实的代价，也需要这些信息。
  如果没有真实统计信息，系统就会假定一些默认值，而这些默认值几乎肯定并不准确。因此，
  在没有运行`ANALYZE`的情况下去检查应用的索引使用情况，基本上是徒劳的。

- 用真实数据做实验。使用测试数据来设置索引，只会告诉你测试数据需要什么索引，
  仅此而已。

  特别要避免使用按比例缩减的测试数据集。虽然在 100000 行里选 1000 行可能适合用索引，但在 100 行里选 1 行几乎不适合，因为这 100 行大概都能放进一个磁盘页，而没有任何计划能比顺序取一个磁盘页更快。

  另外，在编造测试数据时也要小心；如果应用尚未投入生产，这往往无法避免。非常相似、
  完全随机、或者按排序顺序插入的值，都会让统计信息偏离真实数据应有的分布。

- 当索引没有被使用时，强制使用它们对测试会很有帮助。
  有一些运行时参数可以关闭不同的计划类型（见第 16.4 节）。例如，关闭顺序扫描（`enable_seqscan`）和嵌套循环连接（`enable_nestloop`）这两种最基本的计划，
  就会迫使系统使用不同的计划。如果系统仍然选择顺序扫描或嵌套循环连接，
  那么索引未被使用很可能有更根本的原因，例如查询条件并不匹配该索引。
  （前面各节已经解释了什么样的查询可以使用什么样的索引。）

- 如果强制使用索引后确实使用了索引，那么就有两种可能：要么系统是对的，
  使用索引确实不合适；要么查询计划的代价估计并未反映现实。因此，
  你应该分别在有索引和无索引的情况下对查询计时。`EXPLAIN ANALYZE`命令在这里会很有帮助。

- 如果发现代价估计是错的，那么又有两种可能。
  总代价是根据每个计划节点的逐行代价乘以该计划节点的选择率估计计算出来的。
  计划节点的估计代价可以通过运行时参数来调整（见第 16.4 节）。而选择率估计不准确，则是因为统计信息不足。可以尝试通过调整统计信息收集参数来改进这一点（见ALTER TABLE）。

  如果你无法把这些代价调得更合适，那么可能就不得不退而求其次，显式强制使用索引。
  你也可能希望联系PostgreSQL开发者来研究这个问题。

第 12 章 并发控制

本章描述 PostgreSQL 数据库系统在两个或更多会话试图同时访问同一数据时的行为。在这种情况下，目标是在保持严格数据完整性的同时，为所有会话提供高效访问。
每一个数据库应用开发者都应该熟悉本章涉及的主题。

12.1. 简介

与使用锁进行并发控制的传统数据库系统不同，
PostgreSQL通过使用多版本模型（多版本并发控制，MVCC）来维护数据一致性。这意味着，
在查询数据库时，每个事务看到的都是某个较早时刻的数据快照（即一个数据库版本），
而不受底层数据当前状态的影响。这样可以保护事务，使其看不到由（其他）
并发事务在同一数据行上执行更新所产生的不一致数据，从而为每个数据库会话提供事务隔离。

使用MVCC并发控制模型而不是加锁的主要优点在于，在MVCC中，为查询（读取）
数据而获得的锁不会与为写入数据而获得的锁冲突，因此读取不会阻塞写入，
写入也不会阻塞读取。

对于无法轻易适应MVCC行为的应用，PostgreSQL也提供表级和行级锁设施。不过，正确使用
MVCC通常会比加锁提供更好的性能。

12.2. 事务隔离

SQL标准根据并发事务之间必须防止的三个现象，定义了四个事务隔离级别。
这些不受欢迎的现象是：

dirty read

一个事务读取了由另一个并发未提交事务写入的数据。

nonrepeatable read

一个事务重新读取它之前读过的数据，
发现这些数据已经被另一个事务修改（且该事务已在初次读取之后提交）。

phantom read

一个事务重新执行一个返回满足某个搜索条件的行集合的查询，
发现满足该条件的行集合由于另一个最近提交的事务而发生了变化。

四种事务隔离级别及其对应行为见表 12.1。

表 12.1. SQL 事务隔离级别

隔离级别	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

PostgreSQL 提供读已提交和可串行化两个隔离级别。

12.2.1. 读已提交隔离级别

读已提交是PostgreSQL中默认的隔离级别。当事务使用这一隔离级别时，SELECT 查询只能看到查询开始之前已经提交的数据；它既看不到未提交的数据，也看不到查询执行期间并发事务提交的更改。实际上，SELECT 查询看到的是该查询开始运行瞬间的数据库快照。不过，SELECT 能看到其自身事务中先前执行的更新效果，即使这些更新尚未提交。还要注意，即使两个连续的 SELECT命令位于同一个事务中，如果其他事务在第一个SELECT执行期间提交了更改，这两个命令也可能看到不同的数据。

UPDATE、DELETE、SELECT FOR UPDATE 命令在搜索目标行时的行为与 SELECT 相同：它们只会找到在命令开始时已经提交的目标行。不过，当找到这样的目标行时，它可能已经被其他并发事务更新（或删除或标记为更新）。在这种情况下，即将执行更新的事务会等待第一个更新事务提交或回滚（如果它仍在进行中）。如果第一个更新事务回滚，那么它的作用将被撤销，第二个更新事务就可以继续更新最初找到的行。如果第一个更新事务提交了，那么若该行已被第一个更新者删除，第二个更新事务就会忽略该行；否则，第二个更新者将尝试在该行的已更新版本上应用自己的操作。该命令的搜索条件（WHERE 子句）会被重新计算，以判断该行的已更新版本是否仍然符合搜索条件。如果符合，则第二个更新者会基于该行的已更新版本继续执行操作。

由于上述规则，更新命令可能看到一个不一致的快照：它能够看到并发更新命令在它试图更新的同一行上的效果，却看不到这些命令对数据库中其他行的影响。这种行为使得读已提交模式不适合涉及复杂搜索条件的命令；不过，对于较简单的场景它恰到好处。例如，考虑以如下事务更新银行余额：

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

如果两个这样的事务并发地尝试更改账户 12345 的余额，我们显然希望第二个事务从该账户行的已更新版本开始。因为每个命令只影响一个预先确定的行，让它看到该行的已更新版本不会造成任何麻烦的不一致。

因为在读已提交模式中，每个新命令都从一个新快照开始，而该快照包含截至当时已提交的所有事务，所以同一事务中的后续命令无论如何都会看到并发事务已提交的效果。这里讨论的问题在于，在单个命令内部，我们是否能看到数据库的一个绝对一致视图。

读已提交模式所提供的部分事务隔离对于许多应用来说已经足够，而且这种模式既快速又容易使用。不过，它并不能满足所有场景。执行复杂查询和更新的应用，可能需要比读已提交模式所提供的更严格一致的数据库视图。

12.2.2. 可串行化隔离级别

可串行化隔离级别提供最严格的事务隔离。该级别模拟串行事务执行，就像事务是一个接一个串行执行而不是并发执行一样。不过，使用该级别的应用必须准备好在发生串行化失败时重试事务。

当事务使用可串行化级别时，SELECT查询只能看到事务开始之前已经提交的数据；它既看不到未提交的数据，也看不到事务执行期间并发事务提交的更改。（不过，SELECT

能看到其自身事务中先前执行的更新效果，即使这些更新尚未提交。）这一点与读已提交不同：SELECT看到的是事务开始时刻的快照，而不是事务中当前查询开始时刻的快照。因此，单个事务中连续的SELECT命令看到相同的数据，也就是说，它们看不到在本事务启动之后由其他事务提交的更改。（这种行为对于报表类应用来说可能非常理想。）

UPDATE、DELETE和SELECT FOR UPDATE命令在搜索目标行方面的行为与SELECT相同：它们只会找到事务开始时刻已提交的目标行。然而，这样的目标行在被找到时，可能已经被另一个并发事务更新（或删除或标记为更新）。在这种情况下，可串行化事务将等待第一个更新事务提交或回滚（如果仍在进行）。如果第一个更新事务回滚，那么它的作用将被撤销，可串行化事务就可以继续更新最初找到的行。但如果第一个更新事务提交了（并且确实更新或删除了该行，而不仅仅是选取它准备更新），那么可串行化事务将被回滚，并收到以下消息：

```
ERROR:  could not serialize access due to concurrent update
```

这是因为可串行化事务不能修改或锁定在它开始之后被其他事务更改过的行。

当应用收到这条错误消息时，应当中止当前事务，并从头重试整个事务。第二次执行时，该事务会把先前已提交的更改视为其初始数据库视图的一部分，因此以该行的新版本作为新事务更新的起点时，就不存在逻辑冲突。

注意，只有更新事务才可能需要重试；只读事务永远不会发生串行化冲突。

可串行化模式严格保证每个事务看到的都是完全一致的数据库视图。然而，当并发更新使得串行执行的假象无法维持时，应用必须准备好重试事务。由于重做复杂事务的代价可能相当可观，只有当更新事务包含的逻辑复杂到在读已提交模式下可能给出错误结果时，才推荐使用可串行化模式。最常见的情况是，当一个事务要执行多条必须看到相同数据库视图的连续命令时，就需要使用可串行化模式。

12.3. 显式锁定

PostgreSQL提供多种锁模式来控制对表中数据的并发访问。当MVCC无法给出期望的行为时，这些模式可用于由应用控制的锁定。此外，大多数PostgreSQL命令会自动获取适当模式的锁，以确保被引用的表在命令执行期间不会以不兼容的方式被删除或修改。（例如，ALTER TABLE不能与同一张表上的其他操作并发执行。）

要查看数据库服务器中当前尚未释放的锁列表，可以使用 pg_locks 系统视图（第 43.32 节）。有关监控锁管理器子系统状态的更多信息，请参见第 23 章。

12.3.1. 表级锁

下面的列表展示了可用的锁模式，以及 PostgreSQL 自动使用它们的场景。请记住，所有这些锁模式都是表级锁，即使名称中包含“row”（行）一词也是如此；这些锁模式的名称是历史遗留的。名称在某种程度上反映了每种锁模式的典型用法——但语义都是一样的。一种锁模式与另一种的唯一实质区别，是各自与之冲突的锁模式集合。两个事务不能同时在同一张表上持有相互冲突模式的锁。（不过，事务永远不会与自身冲突。例如，一个事务可以先取得 ACCESS EXCLUSIVE 锁，之后再在同一张表上取得 ACCESS SHARE 锁。）不冲突的锁模式可以由多个事务同时持有。特别要注意，有些锁模式是自冲突的（例如，ACCESS EXCLUSIVE 锁同一时刻只能被一个事务持有），而另一些不是自冲突的（例如，ACCESS SHARE 锁可以被多个事务持有）。锁一旦取得，就会保持到事务结束。

表级锁模式

ACCESS SHARE

只与 ACCESS EXCLUSIVE 锁模式冲突。

SELECT 和 ANALYZE 命令会在被引用的表上获取这种模式的锁。通常，任何只读取表而不修改它的查询都会获取这种锁模式。

ROW SHARE

与 EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。

SELECT FOR UPDATE 命令会在目标表上获取这种模式的锁（此外，对于引用到但没有用 FOR UPDATE 选取的其他表，还会获取 ACCESS SHARE 锁）。

ROW EXCLUSIVE

与SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。

UPDATE、DELETE 和 INSERT 命令会在目标表上获取这种锁模式（此外，对任何其他被引用的表还会获取 ACCESS SHARE 锁）。通常，任何修改表中数据的命令都会获取这种锁模式。

SHARE UPDATE EXCLUSIVE

与 SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。这种模式用于保护表不受并发模式更改和 VACUUM 运行的影响。

由VACUUM（不带FULL）获取。

SHARE

与 ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。这种模式用于保护表不受并发数据更改的影响。

由 CREATE INDEX 获取。

SHARE ROW EXCLUSIVE

与ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。

这种锁模式不会被任何 PostgreSQL 命令自动获取。

EXCLUSIVE

与 ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。这种模式只允许并发的 ACCESS SHARE 锁，也就是说，只有对该表的读操作可以与持有这种锁模式的事务并行执行。

这种锁模式不会被任何 PostgreSQL 命令自动获取。

ACCESS EXCLUSIVE

与所有模式的锁冲突（ACCESS SHARE、ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE）。这种模式保证持有者是以任何方式访问该表的唯一事务。

由ALTER TABLE、DROP TABLE、REINDEX、CLUSTER和VACUUM FULL命令获取。这也是未显式指定模式的LOCK TABLE语句的默认锁模式。

提示

只有 ACCESS EXCLUSIVE 锁才会阻塞 SELECT（不带 FOR UPDATE）语句。

12.3.2. 行级锁

除了表级锁之外，还有行级锁。当某一行被更新（或删除或标记为更新）时，
会自动获得该特定行上的行级锁。这种锁一直保持到事务提交或回滚为止。
行级锁不影响数据查询；它们只会阻塞对同一行的写入者。
要在不实际修改某行的情况下获取该行上的行级锁，可以使用SELECT FOR UPDATE选择该行。
注意，一旦获得了特定的行级锁，事务就可以多次更新该行而无须担心冲突。

PostgreSQL 不会在内存中保存任何关于已修改行的信息，因此一次加锁的行数没有限制。不过，锁定一行可能会导致一次磁盘写；例如，SELECT FOR UPDATE 会修改被选中的行以标记它们，因此会产生磁盘写入。

除了表级锁和行级锁之外，还使用页级共享/排他锁来控制对共享缓冲池中表页面的读写访问。
这些锁会在取出或更新某一行后立即释放。应用开发者通常不需要关心页级锁，
这里提到它们只是为了完整性。

12.3.3. 死锁

使用显式锁可能会增加死锁的发生概率。所谓死锁，是指两个（或更多）
事务各自持有对方想要的锁。例如，如果事务 1 在表 A 上获得了排他锁，然后试图再获取表 B 上的排他锁，而事务 2 已经持有表 B 上的排他锁，现在又想获取表 A 上的排他锁，那么两者都无法继续进行。PostgreSQL 会自动检测死锁，并通过中止其中一个事务来解决这个问题，使其他事务得以继续完成。（具体会中止哪个事务很难预测，也不应依赖这种预测。）

还要注意，死锁也可能由于行级锁而发生（因此，即使没有使用显式锁，它们也可能出现）。
考虑下面这种情况：两个并发事务都在修改一个表。第一个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

这会在指定账号的那一行上获取一个行级锁。然后第二个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

第一条 UPDATE 语句成功地在指定行上获取了一个行级锁，因此它成功更新了那一行。然而，第二条 UPDATE 语句发现它试图更新的行已经被锁住了，于是它等待持有该锁的事务结束。此时，事务二正在等待事务一结束之后才能继续执行。现在，事务一执行：

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

事务一试图在指定行上获取一个行级锁，但它做不到，因为事务二已经持有了这个锁。
于是它必须等待事务二完成。这样一来，事务一被事务二阻塞，
而事务二又被事务一阻塞：死锁条件形成了。PostgreSQL 会检测到这种情况，并中止其中一个事务。

防范死锁的最佳办法通常是通过确保所有使用数据库的应用以一致的顺序在多个对象上获取锁来避免死锁。前面死锁例子的原因正在于此：如果两个事务以相同的顺序更新行，
就不会发生死锁。还应确保事务在一个对象上首先获取的锁是该对象所需的最高模式。
如果无法提前验证这一点，那么可以在运行时重试因死锁而中止的事务来处理死锁。

只要没有检测到死锁，寻求表级锁或行级锁的事务就会无限期等待冲突锁被释放。
这意味着让应用长时间保持事务打开并不是好主意（例如等待用户输入时）。

12.4. 应用级别的数据一致性检查

由于PostgreSQL中的读操作不加锁（无论事务隔离级别如何），
一个事务读取的数据仍可能被另一个并发事务覆盖。换句话说，某行被`SELECT`返回，
并不意味着该行在返回的那一刻（即当前查询开始之后的某个时刻）仍然是当前的。
该行可能已经被一个在本事务开始之后提交的事务修改或删除。即使该行“现在”仍然有效，
它也可能在当前事务提交或回滚之前被更改或删除。

换个角度想，每个事务看到的都是数据库内容的一个快照，
而并发执行的事务完全可能看到不同的快照。因此“现在”这个概念本身就很可疑。
如果客户端应用彼此隔离，这通常不是什么大问题；
但如果客户端可以通过数据库之外的渠道通信，就可能引起严重的混乱。

要确保某一行当前仍然有效，并保护它不受并发更新影响，就必须使用`SELECT FOR UPDATE`或适当的`LOCK TABLE`语句。（`SELECT FOR UPDATE`只锁定返回的行以防止并发更新，而`LOCK TABLE`会锁住整张表。）从其他环境向PostgreSQL迁移应用时，应当考虑这一点。（在 6.5 版之前，PostgreSQL使用的是读锁，因此从早于 6.5 的PostgreSQL版本升级时，上述考虑同样相关。）

在 MVCC 之下，全局有效性检查需要格外用心。例如，一个银行应用可能想检查一张表中所有贷方金额之和等于另一张表中借方金额之和，而两张表都在被活跃地更新。在读已提交模式下，
比较两条先后执行的 `SELECT sum(...)` 命令的结果并不可靠，因为第二条查询很可能把第一条查询未计入的事务的结果包含进来。在单个可串行化事务中做两次求和，
可以准确反映该可串行化事务开始之前已提交事务的效果——但人们完全有理由怀疑，
等到结果交付时它是否仍然有效。
如果可串行化事务在尝试做一致性检查之前自己已经做了某些更改，
这个检查的有用性就更值得商榷了，因为此时它包含的是事务开始后一部分而非全部更改。
在这种情况下，谨慎的人可能希望把检查所需的所有表都锁住，
以获得关于当前现实无可争议的图景。`SHARE` 模式（或更高模式）的锁保证被锁表中没有未提交的更改，当前事务自己的更改除外。

还要注意，如果依靠显式锁来防止并发更改，就应当使用读已提交模式；
或者在可串行化模式下小心确保在执行查询之前先取得锁。
在可串行化事务中取得的显式锁保证没有其他正在修改该表的事务还在运行，
但如果事务看到的快照早于取得锁的时刻，它也可能早于表中一些现已提交的更改。
可串行化事务的快照实际上是在其第一条查询或数据修改命令（`SELECT`、`INSERT`、`UPDATE` 或 `DELETE`）开始时冻结的，因此完全可以在快照冻结之前先取得显式锁。

12.5. 锁与索引

尽管PostgreSQL对表数据提供了非阻塞的读写访问，但对于
PostgreSQL中实现的每一种索引访问方法，目前并非都能提供非阻塞的读写访问。
各种索引类型的处理方式如下：

B-树索引

读/写访问使用短期的共享/排他页级锁。锁在每次索引行被取出或插入之后立即释放。
B-树索引在不产生死锁条件的情况下提供了最高的并发度。

GiST和 R 树索引

读/写访问使用共享/排他的索引级锁。锁在命令完成之后释放。

Hash 索引

读/写访问使用共享/排他的页级锁。锁在页面处理完成之后释放。
页级锁比索引级锁提供更好的并发性，但容易发生死锁。

总之，B-树索引为并发应用提供了最佳性能；由于它们还比 Hash 索引拥有更多特性，因此对于需要为标量数据建立索引的并发应用，推荐使用 B-树索引类型。在处理非标量数据时，B-树显然无法使用；在这种情况下，应用开发者应当了解 GiST 和 R 树索引相对较差的并发性能。

第 13 章 性能提示

查询性能可能受许多因素影响。其中一些因素可以由用户控制，
另一些则属于系统底层设计的基本特性。本章提供一些帮助理解和调优PostgreSQL性能的提示。

13.1. 使用EXPLAIN

PostgreSQL会为收到的每个查询制定一个查询计划。
选择与查询结构和数据特性相匹配的正确计划，对获得良好的性能至关重要。可以使用EXPLAIN命令查看系统为任意查询生成的查询计划。读懂计划是一门值得专门教程详述的技艺，
本节并非那样的教程，但这里会介绍一些基础知识。

EXPLAIN当前引用的这些数字是：

- 估计启动代价（在输出扫描开始之前消耗的时间，例如排序节点中完成排序所需的时间）
- 估计总代价（如果所有行都被检索；不过它们也可能不被检索：例如带LIMIT子句的查询会在付出全部代价之前停止）
- 该计划节点输出的估计行数（同样，仅在完全执行的情况下）
- 该计划节点输出行的估计平均宽度（以字节计）

这些开销以磁盘页读取的次数为单位来衡量。（CPU 工作量估计会用一些相当任意的经验系数换算成磁盘页单位。如果你想试验这些系数，可以看第 16.4.2 节中的运行时配置参数列表。）

需要理解的一点是，上层节点的开销包含了其所有子节点的开销。还要注意，
这个开销只反映规划器/优化器关心的内容。特别是，
它没有考虑将结果行传输给前端所消耗的时间，而这在实际耗时中可能是重要因素；
但规划器会忽略这些代价，因为它无法通过改变计划来影响它们。
（我们相信每个正确的计划都会输出相同的行集。）

rows 输出值有些容易误解，因为它不是查询处理或扫描的行数。这个数字通常更小，反映的是在该节点上应用的任何WHERE子句条件的估计选择率。理想情况下，
顶层的行数估计应当接近查询实际返回、更新或删除的行数。

这里是一些例子（使用执行过VACUUM ANALYZE的回归测试数据库，以及 7.3 开发版源码）：

```
EXPLAIN SELECT * FROM tenk1;

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

这大概是最简单的情况了。如果你执行

```
SELECT * FROM pg_class WHERE relname = 'tenk1';
```

就会发现tenk1有 233 个磁盘页和 10000 行。因此代价估计为 233 次页面读取（每次定义为 1.0），加上 10000 * cpu_tuple_cost（当前为 0.01；可以试试SHOW cpu_tuple_cost）。

现在修改查询，添加一个WHERE条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
```

```
                          QUERY PLAN
----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..358.00 rows=1033 width=148)
   Filter: (unique1 < 1000)
```

由于有了WHERE子句，估计输出行数减少了。但扫描仍需访问全部 10000 行，因此代价没有下降；实际上还略有上升，以反映检查WHERE条件所消耗的额外 CPU 时间。

这条查询实际选出的行数是 1000，但估计值只是近似。如果重复这个实验，很可能会得到略有不同的估计值；此外，由于ANALYZE生成的统计信息来自该表的随机采样，
这个估计值可能在每次执行ANALYZE命令之后发生变化。

进一步收紧查询条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50;

                                  QUERY PLAN
----------------------------------------------------------------------
---------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..179.33 rows=49
 width=148)
   Index Cond: (unique1 < 50)
```

可以看到，如果WHERE条件的选择性足够强，规划器最终会判定索引扫描比顺序扫描更便宜。
得益于索引，这个计划只需访问 50 行，因此尽管每次单独读取比顺序读取一个完整磁盘页更昂贵，它仍然是赢家。

再向WHERE子句添加另一个条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50 AND stringu1 = 'xxx';

                                 QUERY PLAN
----------------------------------------------------------------------
---------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..179.45 rows=1
 width=148)
   Index Cond: (unique1 < 50)
   Filter: (stringu1 = 'xxx'::name)
```

新增的条件stringu1 = 'xxx'降低了估计输出行数，却没有降低代价，因为仍然必须访问同一组行。注意，stringu1子句不能用作索引条件（因为该索引只建立在unique1列上）。
它只能作为过滤条件，应用到通过索引取出的行上。因此，代价实际上略有增加，
以反映这项额外检查。

使用之前讨论的列，尝试连接两个表：

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.
unique2 = t2.unique2;

                               QUERY PLAN
----------------------------------------------------------------------
-------
 Nested Loop  (cost=0.00..327.02 rows=49 width=296)
   ->  Index Scan using tenk1_unique1 on tenk1 t1
                                      (cost=0.00..179.33 rows=49
 width=148)
```

```
         Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2
                                     (cost=0.00..3.01 rows=1 width=
148)
         Index Cond: ("outer".unique2 = t2.unique2)
```

在这个嵌套循环连接中，外层扫描就是前前一个例子中的那个索引扫描，
其代价和行计数也相同，因为WHERE子句unique1 < 50正是在该节点上应用的。t1.unique2 = t2.unique2子句此时还无关，因此不会影响外层扫描的行计数。对于内层扫描，当前外层扫描行的unique2值被代入内层索引扫描，产生一个形如t2.unique2 = *constant*的索引条件。因此我们得到的内层扫描计划和代价，与执行EXPLAIN SELECT * FROM tenk2 WHERE unique2 = 42之类查询所得到的相同。循环节点的代价则建立在外层扫描代价之上，再加上每个外层行都要执行一次内层扫描的代价（这里是 49 * 3.01），以及少量连接处理的 CPU 时间。

本例中，连接的输出行数等于两个扫描行数的乘积，但并非所有情况都如此，
因为可能还有同时涉及两个表的WHERE子句，因此只能在连接处应用，
不能应用到任一输入扫描。例如，如果我们加上WHERE ... AND t1.hundred < t2.hundred，它会减少连接节点的输出行数，但不会改变任何一个输入扫描。

查看不同计划的一种办法，是使用针对每种计划类型的启用/禁用开关，
强制规划器放弃它认为的最优策略。（这是一种粗糙的工具，但很有用。另见第 13.3 节。）

```
SET enable_nestloop = off;
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.
unique2 = t2.unique2;

                              QUERY PLAN
----------------------------------------------------------------------
-----
 Hash Join  (cost=179.45..563.06 rows=49 width=296)
   Hash Cond: ("outer".unique2 = "inner".unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..333.00 rows=10000 width=148)
   ->  Hash  (cost=179.33..179.33 rows=49 width=148)
         ->  Index Scan using tenk1_unique1 on tenk1 t1
                                    (cost=0.00..179.33 rows=49 width=
148)
               Index Cond: (unique1 < 50)
```

这个计划打算用与之前相同的索引扫描取出tenk1中那 50 行感兴趣的行，把它们存入一个内存中的哈希表，然后对tenk2做顺序扫描，对每一行tenk2在哈希表中探测t1.unique2 = t2.unique2的可能匹配。读取tenk1并建立哈希表的代价完全是哈希连接的启动代价，因为在开始读取tenk2之前不会有任何输出。连接的总时间估计中还包含一笔不小的 CPU 时间开销，用于对哈希表探测 10000 次。但注意，我们不是按 10000 乘以 179.33 来计费的；在这种计划类型中，哈希表的建立只做一次。

可以使用EXPLAIN ANALYZE检查规划器代价估计的准确性。这个命令会实际执行查询，然后显示每个计划节点内累计的真实运行时间，以及普通EXPLAIN所显示的相同估计代价。例如，
我们可能得到这样的结果：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50
 AND t1.unique2 = t2.unique2;

                                   QUERY PLAN
----------------------------------------------------------------------
---------
```

```
 Nested Loop  (cost=0.00..327.02 rows=49 width=296)
                                 (actual time=1.181..29.822 rows=50
 loops=1)
   ->  Index Scan using tenk1_unique1 on tenk1 t1
                  (cost=0.00..179.33 rows=49 width=148)
                                 (actual time=0.630..8.917 rows=50
 loops=1)
         Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2
                  (cost=0.00..3.01 rows=1 width=148)
                                 (actual time=0.295..0.324 rows=1
 loops=50)
         Index Cond: ("outer".unique2 = t2.unique2)
 Total runtime: 31.604 ms
```

注意，“actual time”值以真实时间的毫秒为单位，而“cost”估计则以任意磁盘读取单位表示，因此两者不太可能相互吻合。需要关注的是实际时间与估计代价之间的比值是否一致。

在某些查询计划中，一个子计划节点可能会执行多次。例如，
上面那个嵌套循环计划中的内层索引扫描会对外层的每一行执行一次。在这种情况下，“loops”值报告的是该节点的总执行次数，而显示的 actual time 和 rows 值是每次执行的平均值。这样做是为了让这些数字更容易与代价估计的展示方式相比较。将它们乘以“loops”值，
就能得到该节点实际消耗的总时间。

`EXPLAIN ANALYZE`显示的`Total runtime`包括执行器启动和关闭所需的时间，以及处理结果行所花费的时间。它不包括解析、重写和规划时间。对于`SELECT`查询，
总运行时间通常只比顶层计划节点报告的总时间略大。对于`INSERT`、`UPDATE`和`DELETE`命令，
总运行时间可能大得多，因为它包括了处理结果行所花费的时间。在这些命令中，
顶层计划节点的时间本质上就是计算新行和/或定位旧行所花费的时间，
但不包括实施更改所花费的时间。

值得注意的是，不应把`EXPLAIN`的结果外推到实际测试场景之外的情况；例如，
不能假定在很小的表上得到的结果也适用于大型表。规划器的代价估计不是线性的，
因此它可能会为更大或更小的表选择不同的计划。
一个极端示例是：对于只占用一个磁盘页的表，无论索引是否可用，
几乎总会得到顺序扫描计划。规划器认识到，无论如何处理该表都需要一次磁盘页读取，
因此再额外读取页面去查看索引并没有价值。

13.2. 规划器使用的统计信息

如上一节所见，查询规划器需要估计查询将检索多少行，才能对查询计划做出良好选择。
本节简要介绍系统用于这些估计的统计信息。

统计信息的一部分是各个表和索引的条目总数，以及各个表和索引占用的磁盘块数。
这些信息保存在表`pg_class`的`reltuples`和`relpages`列中。
可以使用类似下面的查询来查看：

```
SELECT relname, relkind, reltuples, relpages FROM pg_class WHERE
 relname LIKE 'tenk1%';

    relname    | relkind | reltuples | relpages
---------------+---------+-----------+----------
 tenk1         | r       |     10000 |      233
 tenk1_hundred | i       |     10000 |       30
```

```
 tenk1_unique1 | i       |     10000 |       30
 tenk1_unique2 | i       |     10000 |       30
(4 rows)
```

这里可以看到，tenk1包含 10000 行，它的索引也一样，但索引比表小得多（这并不意外）。

出于效率考虑，reltuples和relpages不会实时更新，
因此它们通常只包含近似值（这对规划器的用途来说已经足够）。创建表时，
它们被初始化为哑值（目前分别为 1000 和 10）。某些命令会更新它们，目前是VACUUM、ANALYZE和CREATE INDEX。独立的ANALYZE（即不作为VACUUM一部分执行的那种）由于不会读取表的每一行，生成的reltuples值是近似的。

大多数查询只会检索表中一部分行，因为它们通过WHERE子句限制了需要检查的行。因此，
规划器需要估算WHERE子句的选择率，也就是满足WHERE子句中各个条件的行所占比例。
完成这项任务所需的信息存储在pg_statistic系统目录中。pg_statistic
中的条目由ANALYZE和VACUUM ANALYZE命令更新，而且即使刚更新完，也始终只是近似值。

手动检查统计信息时，与其直接查看pg_statistic，不如查看它的视图pg_stats。pg_stats旨在让统计信息更易读。此外，pg_stats所有用户都可以读取，而pg_statistic
只有超级用户可以读取。
（这可以防止没有相应权限的用户通过统计信息获知其他人表中的内容。pg_stats
视图被限制为只显示当前用户有权读取的表的相关行。）例如，可以执行：

```
SELECT attname, n_distinct, most_common_vals FROM pg_stats WHERE
 tablename = 'road';

 attname | n_distinct |

                                                                most_
common_vals


---------+------------+-----------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------
 name    |  -0.467008 | {"I- 580                        Ramp","I- 880
                        Ramp","Sp Railroad                        ",
"I- 580                             ","I- 680
 Ramp","I- 80                        Ramp","14th
     St  ","5th                          St  ","Mission
         Blvd","I- 880                           "}
 thepath |         20 | {"[(-122.089,37.71),(-122.0886,37.711)]"}
(2 rows)
```

pg_stats的详细描述见第 43.35 节。

pg_statistic中存储的信息量，特别是每列most_common_vals和histogram_bounds
数组中的最大项数，可以使用ALTER TABLE SET STATISTICS命令按列设置，也可以通过设置
运行时参数default_statistics_target进行全局设置。目前默认上限是 10 项。提高这一
上限可能让规划器做出更准确的估计，尤其是对于数据分布不规则的列；代价则是pg_

statistic占用更多空间，并且计算估计值所需时间也会略有增加。相反，
对于数据分布较简单的列，较低的上限可能已经足够。

13.3. 用显式JOIN子句控制规划器

可以在一定程度上用显式JOIN语法控制查询规划器。要理解这一点为何重要，
先需要一些背景知识。

在一个简单的连接查询中，例如

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

规划器可以自由地按任意顺序连接这些表。例如，它可以生成一个查询计划，先利用WHERE条件a.id = b.id将 A 连接到 B，然后再利用另一个WHERE条件把 C 连接到这个结果上。也可以先连接 B 和 C，再把 A 连接到所得结果上。甚至还可以先连接 A 和 C，再与 B 连接，但这样效率会很低，因为必须先形成 A 和 C 的完整笛卡尔积，而WHERE子句中并没有可用于优化这一连接的条件。（PostgreSQL执行器中的所有连接都发生在两个输入表之间，因此结果必须以这些形式之一逐步构造出来。）
关键在于，这些不同的连接可能性在语义上是等价的，但执行代价可能相差极大。因此，
规划器会探索它们，力图找出最高效的查询计划。

当查询只涉及两个或三个表时，需要考虑的连接顺序并不多。
但可能的连接顺序数量会随着表数增加而呈指数增长。输入表超过十个左右之后，
对所有可能性做穷举搜索实际上就不再可行，甚至六七个表也可能让规划耗时长得令人厌烦。
输入表过多时，PostgreSQL规划器会从穷举搜索切换到一种遗传概率搜索，
只考虑有限数量的可能性。（切换阈值由运行时参数geqo_threshold控制。）
遗传搜索耗时更少，但并不一定能找到最优计划。

当查询涉及外连接时，规划器比处理普通（内）连接时拥有更小的自由度。例如，考虑

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.
id);
```

尽管这个查询的约束表面上与前一个非常相似，但它们的语义不同，因为如果 A 中有某一行无法匹配 B 和 C 连接结果中的任何行，该行仍然必须被输出。因此这里规划器对连接顺序没有选择：它必须先连接 B 和 C，再把 A 连接到该结果上。相应地，这个查询比前一个查询需要更少的规划时间。

显式内连接语法（INNER JOIN、CROSS JOIN或无修饰的JOIN）在语义上和FROM中列出输入关系是一样的，因此它不需要约束连接顺序。但仍然可以指示
PostgreSQL查询规划器将显式内JOIN当作带有连接顺序约束来处理。例如，
下面三个查询在逻辑上是等价的：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref
 = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

但如果告诉规划器遵循JOIN顺序，那么第二个和第三个查询在规划上会比第一个花费更少时间。
对于只有三个表的连接来说，这种效果微不足道；但当表很多时，它可能非常关键。

要强制规划器遵循内连接的JOIN顺序，可以将运行时参数join_collapse_limit设置为 1。
（其他可能值见下文讨论。）

不必为了缩短搜索时间而完全约束连接顺序，因为可以在普通FROM列表的某一项中使用JOIN
操作符。例如，考虑

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

如果设置join_collapse_limit = 1，就会强制规划器先将 A 连接到 B，然后再与其他表连接，但不会进一步约束它的选择。在这个示例中，可能的连接顺序数量减少了 5 倍。

以这种方式约束规划器的搜索，是一种既可减少规划时间、
又可引导规划器生成更好查询计划的实用技巧。如果规划器默认选择了糟糕的连接顺序，
可以通过JOIN语法强制它采用更好的顺序，前提当然是确实知道哪个顺序更好。建议进行实验。

与此密切相关、同样会影响规划时间的另一个问题，是将子查询折叠进父查询。例如，考虑

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

这种情况可能出现在使用包含连接的视图时；该视图的SELECT规则会被插入到引用视图的位置，
从而得到一个与上面非常相似的查询。通常，规划器会尝试把子查询折叠进父查询，得到

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

这通常会生成比单独规划子查询更好的计划。（例如，外层WHERE条件可能会先把 X 连接到 A，
从而消除 A 中的大量行，也就避免了形成子查询完整逻辑输出的需要。）但与此同时，规划时间
也增加了；这里把两个彼此独立的三路连接问题，替换成了一个五路连接问题。
由于可能性数量呈指数增长，这种差别可能非常大。为了避免陷入巨大的连接搜索问题，
如果折叠子查询会使父查询产生超过from_collapse_limit个FROM项，
规划器就不会折叠该子查询。可以通过调高或调低这个运行时参数，
在规划时间和计划质量之间做权衡。

from_collapse_limit和join_collapse_limit名称相似，
因为它们做的几乎是同一件事：一个控制规划器何时将子查询“平面化”，
另一个控制何时将显式内连接平面化。通常，要么把join_collapse_limit设置为与from_collapse_limit相同，这样显式连接与子查询的行为类似；要么把join_collapse_limit设置为 1，如果想使用显式连接控制连接顺序。不过，也可以把它们设置为不同的值，以更细致地调节规划时间与运行时间之间的平衡。

13.4. 填充一个数据库

初次填充数据库时，可能需要做大量的表插入。
本节给出一些让这一过程尽可能高效的建议和技巧。

13.4.1. 禁用自动提交

关闭自动提交，只在最后提交一次。（在普通 SQL 中，这意味着开始时发出BEGIN，结束时发出COMMIT。某些客户端库可能会替调用方完成这件事，
在这种情况下需要确认它们确实会在所需的时候这样做。）如果允许每次插入都单独提交，
PostgreSQL就必须为每一行的加入执行大量工作。
把所有插入都放在一个事务中的另一个好处是：如果其中某一行插入失败，
那么此前插入的所有行都会被回滚，这样就不会留下部分装载的数据。

13.4.2. 使用COPY FROM

使用COPY FROM STDIN在一条命令中装载所有记录，而不是使用一系列INSERT命令。这会大幅减少解析、规划等方面的开销。采用这种方法时无须关闭自动提交，因为反正它只是一条命令。

13.4.3. 移除索引

如果正在装载一个新创建的表，最快的方法是先创建表，用COPY批量装载数据，然后再创建该表所需的索引。在已有数据的表上创建索引，要比在每行装载时对索引做增量更新更快。

如果正在向现有表加入大量数据，可以删除索引、装载数据、再重建索引。当然，在索引缺失期间，其他数据库用户的性能可能会受到不利影响。删除唯一索引之前也应慎重考虑，因为唯一约束提供的错误检查会在索引缺失期间丧失。

13.4.4. 增大sort_mem

在还原大量数据时，临时增大sort_mem配置变量可以提升性能。这是因为从头创建 B-树索引时，需要对表的现有内容排序。允许归并排序使用更多缓冲页，意味着需要的归并趟数更少。

13.4.5. 之后运行ANALYZE

每当添加或更新了大量数据之后——包括刚完成表的初次装载之后——运行ANALYZE或VACUUM ANALYZE都是个好主意。这可以确保规划器掌握该表的最新统计信息。如果没有统计信息，或者统计信息已经过时，规划器在生成查询计划时就可能做出糟糕决定，从而导致使用该表的查询性能不佳。

部分 III. 服务器管理

本部分涵盖PostgreSQL数据库管理员关心的主题。包括软件的安装、服务器的设置和配置、用户和数据库的管理以及维护任务。任何运行PostgreSQL服务器的人，无论是供个人使用，尤其是在生产环境中，都应熟悉本部分涵盖的主题。

本部分的内容大体按照新用户应当阅读的顺序来安排。但各章自成体系，可按需单独阅读。相关内容按主题单元以叙述方式展开。希望获得某个命令完整描述的读者，建议查阅第 VI 部分。

开头几章写得无需预备知识即可理解，因此需要自行搭建服务器的新用户可以从这一部分开始探索。本部分其余关于调优和管理的内容假定读者已经熟悉PostgreSQL数据库系统的一般用法。建议读者再参阅第 I 部分和第 II 部分以获取更多信息。

目录

14. 安装说明 195
14.1. 简短版本 195
14.2. 需求 195
14.3. 获取源代码 197
14.4. 如果你要升级 197
14.5. 安装过程 198
14.6. 安装后设置 203
14.6.1. 共享库 203
14.6.2. 环境变量 204
14.7. 受支持的平台 205
15. 在 Windows 上安装 209
16. 服务器运行时环境 210
16.1. PostgreSQL 用户账户 210
16.2. 创建一个数据库集簇 210
16.3. 启动数据库服务器 211
16.3.1. 服务器启动失败 212
16.3.2. 客户端连接问题 213
16.4. 运行时配置 213
16.4.1. 连接和认证 214
16.4.2. 资源消耗 216
16.4.3. 预写式日志 218
16.4.4. 查询规划 219
16.4.5. 错误报告和日志 221
16.4.6. 运行时统计 224
16.4.7. 客户端连接默认值 225
16.4.8. 锁管理 227
16.4.9. 版本和平台兼容性 228
16.4.10. 开发者选项 229
16.4.11. 短选项 229
16.5. 管理内核资源 230
16.5.1. 共享内存和信号量 230
16.5.2. 资源限制 234
16.5.3. Linux 内存过量分配 235
16.6. 关闭服务器 235
16.7. 使用 SSL 的安全 TCP/IP 连接 236
16.8. 使用 SSH 隧道的安全 TCP/IP 连接 236
17. 数据库用户和权限 238
17.1. 数据库用户 238
17.2. 用户属性 238
17.3. 组 239
17.4. 权限 239
17.5. 函数和触发器 240
18. 管理数据库 241
18.1. 概述 241
18.2. 创建数据库 241
18.3. 模板数据库 242
18.4. 数据库配置 243
18.5. 备选位置 243
18.6. 删除数据库 244
19. 客户端认证 246
19.1. `pg_hba.conf` 文件 246

19.2. 认证方法 251
19.2.1. trust 认证 251
19.2.2. 密码认证 251
19.2.3. Kerberos 认证 252
19.2.4. 基于 ident 的认证 252
19.2.5. PAM 认证 254
19.3. 认证问题 254
20. 本地化 255
20.1. 区域支持 255
20.1.1. 概述 255
20.1.2. 优点 256
20.1.3. 问题 256
20.2. 字符集支持 257
20.2.1. 支持的字符集 257
20.2.2. 设置字符集 258
20.2.3. 服务器和客户端之间的自动字符集转换 259
20.2.4. 进一步阅读 260
21. 日常数据库维护任务 262
21.1. 例行清理 262
21.1.1. 回收磁盘空间 262
21.1.2. 更新规划器统计信息 263
21.1.3. 防止事务 ID 回卷故障 263
21.2. 例行重索引 265
21.3. 日志文件维护 265
22. 备份与恢复 266
22.1. SQL 转储 266
22.1.1. 恢复转储 266
22.1.2. 使用`pg_dumpall` 267
22.1.3. 大型数据库 267
22.1.4. 注意事项 268
22.2. 文件系统级别的备份 268
22.3. 版本间的迁移 269
23. 监控数据库活动 271
23.1. 标准 Unix 工具 271
23.2. 统计收集器 272
23.2.1. 统计信息收集配置 272
23.2.2. 查看收集到的统计信息 272
23.3. 查看锁 275
24. 监控磁盘使用情况 276
24.1. 确定磁盘使用情况 276
24.2. 磁盘已满故障 277
25. 预写式日志（WAL） 278
25.1. WAL 的优势 278
25.2. 未来的收益 278
25.3. WAL 配置 279
25.4. 内部机制 280
26. 回归测试 281
26.1. 运行测试 281
26.2. 测试结果评估 282
26.2.1. 错误消息差异 282
26.2.2. Locale 差异 282
26.2.3. 日期和时间差异 282
26.2.4. 浮点差异 283

26.2.5. 行顺序差异 283
26.2.6. “random” 测试 284
26.3. 平台相关的比较文件 284

第 14 章 安装说明

本章描述如何从源码发行包安装 PostgreSQL。

14.1. 简短版本

```
./configure
gmake
su
gmake install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data >logfile 2>&1
 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

长版本就是本章的其余部分。

14.2. 需求

一般来说，现代的 Unix 兼容平台应该能够运行 PostgreSQL。发布时经过具体测试的平台列于下方第 14.7 节。发行包的 `doc` 子目录中有若干平台特定的 FAQ 文档，遇到问题时可以查阅。

构建PostgreSQL需要下列软件包：

- 需要 GNU make；其他make程序不能用。GNU make通常安装在`gmake`名下；本文档一律用该名称称呼它。（在某些系统上GNU make就是默认的`make`工具。）要测试是否为GNU make，可输入

  `gmake --version`

  建议使用 3.76.1 或更高版本。

- 需要一个ISO/ANSI C 编译器。推荐使用较新版本的GCC，但已知PostgreSQL可以用各厂商的多种编译器构建。

- 首先需要gzip来解开发布包。

- 默认将使用 GNU Readline 库（用于舒适的行编辑和命令历史检索）。如果你不想使用它，就必须为 `configure` 指定 `--without-readline` 选项。（在 NetBSD 上，`libedit` 库与 Readline 兼容，找不到 `libreadline` 时会使用它。）

- 要在 Windows NT 或 Windows 2000 上构建，需要 Cygwin 和 cygipc 软件包。详情见 `doc/FAQ_MSWIN` 文件。

下面的软件包是可选的。默认配置中不需要它们，但启用某些构建选项时需要，如下所述。

- 要构建服务器编程语言 PL/Perl，你需要完整的 Perl 安装，包括 `libperl` 库和头文件。由于 PL/Perl 将是一个共享库，在大多数平台上 `libperl` 库也必须是共享库。较新的

Perl 版本中这似乎是默认设置，但早期版本并非如此，而且无论如何这都是在你的站点上安装 Perl 的人的选择。如果你打算不只是偶尔使用 PL/Perl，应确保 Perl 安装在构建时启用了 `usemultiplicity` 选项（`perl -V` 会显示是否如此）。

如果你没有共享库但需要一个，构建期间会出现类似下面的消息指出这一事实：

```
*** Cannot build PL/Perl because libperl is not a shared library.
*** You might have to rebuild your Perl installation.  Refer to
*** the documentation for details.
```

（如果你没有留意屏幕输出，只会注意到 PL/Perl 库对象（`plperl.so` 或类似名称）没有被安装。）如果看到这条消息，你就必须手动重新构建并安装 Perl，才能构建 PL/Perl。在 Perl 的配置过程中，请求共享库。

- 要构建 PL/Python 服务器编程语言，你需要一个 Python 安装，包括头文件。由于 PL/Python 将是一个共享库，在大多数平台上 `libpython` 库也必须是共享库。默认的 Python 安装并非如此。

  如果构建和安装之后你有一个名为 `plpython.so`（扩展名可能不同）的文件，那就一切顺利。否则你应当看到过类似下面一闪而过的通知：

  ```
  *** Cannot build PL/Python because libpython is not a shared
   library.
  *** You might have to rebuild your Python installation.  Refer to
  *** the documentation for details.
  ```

  这意味着你必须重新构建（部分）你的 Python 安装来提供这个共享库。

  问题在于 Python 发行版或 Python 维护者没有提供任何直接的做法。我们能提供的最接近的信息是 Python FAQ 3.30[1]。在某些操作系统上你其实不必构建共享库，但那时你得让 PostgreSQL 构建系统相信这一点。细节请查阅 `src/pl/plpython` 目录中的 `Makefile`。

- 如果你想构建 Tcl 或 Tk 组件（客户端和 PL/Tcl 语言），你当然需要一个 Tcl 安装。

- 要构建 JDBC 驱动，你需要 Ant 1.5 或更高版本和一个 JDK。Ant 是一个用于构建基于 Java 的软件包的特殊工具。可以从 Ant 的网站[2]下载。

  如果你安装了多个 Java 编译器，使用哪一个取决于 Ant 的配置。预编译的 Ant 发行版通常被设置为读取当前用户主目录中的 `.antrc` 文件作为配置。例如，要使用与默认不同的 JDK，这样做也许可行：

  ```
  JAVA_HOME=/usr/local/sun-jdk1.3
  JAVACMD=$JAVA_HOME/bin/java
  ```

  > **注意**
  >
  > 不要试图通过直接调用 `ant` 甚或 `javac` 来构建驱动。那样不行。按下面所述正常运行 `gmake` 即可。

- 要启用本机语言支持（NLS），即以英语以外的语言显示程序消息的能力，你需要一个 Gettext API 的实现。某些操作系统内置了它（例如 Linux、NetBSD、Solaris），其他系统可以从这里

[1] http://www.python.org/doc/FAQ.html#3.30
[2] http://jakarta.apache.org/ant/index.html

下载附加软件包：http://developer.postgresql.org/~petere/bsd-gettext/。
如果你使用的是 GNU C 库中的 Gettext 实现，那么你还需要 GNU Gettext 软件包提供的一些实用程序。对其他任何实现都不需要它。

- 以及 Kerberos、OpenSSL 或 PAM，如果你想支持使用这些服务的认证的话。

如果你要从Git树而不是正式源码包构建，或者要做服务器开发，还需要下列软件包：

- 从 Git 检出构建时需要 GNU Flex和Bison，修改了实际的扫描器和解析器定义文件时也需要。如果需要它们，务必使用Flex 2.5.4 或更高版本以及Bison 1.875 或更高版本。其他yacc程序有时也能使用，但这样做需要额外的努力，不予推荐。其他lex程序则肯定不能用。

如果需要获取 GNU 软件包，可以从本地的 GNU 镜像站点获取（站点列表见http://www.gnu.org/order/ftp.html），或者从ftp://ftp.gnu.org/gnu/获取。

还要检查你是否有足够的磁盘空间。编译期间源码树大约需要 65 MB，安装目录大约需要 15 MB。空的数据库集簇约占 25 MB，数据库所占空间约为容纳相同数据的纯文本文件的五倍。如果你打算运行回归测试，临时最多还需要额外的 90 MB。用 `df` 命令检查磁盘空间。

14.3. 获取源代码

PostgreSQL 7.4.30 的源码可以通过匿名 FTP 从ftp://ftp.postgresql.org/pub/source/v7.4.30/postgresql-7.4.30.tar.gz获取。如果可能请使用镜像。获取文件后，解开源码包：

```
gunzip postgresql-7.4.30.tar.gztar xf postgresql-7.4.30.tar
```

这会创建目录 `postgresql-7.4.30`（位于当前目录之下），其中是PostgreSQL的源码。在其余步骤之前请切换到该目录以继续安装流程。

14.4. 如果你要升级

PostgreSQL的内部数据存储格式会随新版本而变化。因此，
如果你要升级的现有安装的版本号不是“7.4.x”，就必须按照这里的方法备份并恢复数据。
这些说明假定你现有的安装在`/usr/local/pgsql`目录下，数据区域在`/usr/local/pgsql/data`中。请酌情替换为你实际的路径。

1. 确保在备份期间和备份之后数据库没有被更新。这不影响备份的完整性，
   但已更改的数据当然不会包含在内。必要时，可以修改文件`/usr/local/pgsql/data/pg_hba.conf`（或等价文件）中的权限，禁止除你之外的所有人访问。

2. 要备份你的数据库安装，输入：

   ```
   pg_dumpall > outputfile
   ```

   如果你需要保留 OID（例如把它们用作外键时），运行 `pg_dumpall` 时使用 `-o` 选项。

   `pg_dumpall` 不保存大对象。如果你需要保存大对象，请查阅第 22.1.4 节。

   做备份时，你可以使用当前运行版本中的 `pg_dumpall` 命令。但为获得最佳结果，请尽量使用 PostgreSQL 7.4.30 的 `pg_dumpall` 命令，因为该版本相对旧版本包含错误修复和改进。
   虽然你尚未安装新版本时这条建议显得有些特立独行，
   但如果你打算把新版本与旧版本并行安装，遵循它是明智的。那样你就可以正常完成安装，
   稍后再传输数据。这还会减少停机时间。

3. 如果你把新版本安装在与旧版本相同的位置，那么最迟在安装新文件之前关闭旧服务器：

```
kill -INT `cat /usr/local/pgsql/data/postmaster.pid | sed 1q`
```

7.0 之前的版本没有这个 `postmaster.pid` 文件。如果你使用的是这样的版本，就必须自己找出服务器的进程 ID（例如输入 **ps ax | grep postmaster**），并把它提供给 `kill` 命令。

在开机时启动 PostgreSQL 的系统上，很可能有一个能完成同样事情的启动文件。例如，在 Red Hat Linux 系统上可能会发现

```
/etc/rc.d/init.d/postgresql stop
```

有效。另一种可能是 **pg_ctl stop**。

4. 如果你安装在老版本所在的同一位置，最好也把旧安装移开，以防遇到问题需要回退。可以使用类似下面的命令：

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

安装好PostgreSQL 7.4.30 之后，创建新的数据库目录并启动新服务器。记住，这些命令必须以特殊的数据库用户账户登录执行（如果你是在升级，该账户已经存在）。

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data/usr/local/pgsql/
bin/postmaster -D /usr/local/pgsql/data
```

最后，用

```
/usr/local/pgsql/bin/psql -d template1 -f outputfile
```

恢复你的数据，这里使用新的psql。

这些主题在第 22.3 节中有详细讨论，无论如何都建议你阅读。

14.5. 安装过程

1. **配置**

   安装流程的第一步是为你的系统配置源码树并选择所需选项。这通过运行`configure`脚本来完成。默认安装只需输入

   ```
   ./configure
   ```

   该脚本会运行一系列测试来猜测各种系统相关变量的值并检测操作系统的某些特性，最后会在构建树中创建若干文件记录其发现。（如果希望把构建目录与源码树分开，也可以在源码树之外的目录中运行`configure`。）

   默认配置将构建服务器和实用程序，以及所有只需要 C 编译器的客户端应用程序和接口。默认情况下，所有文件都会安装到 `/usr/local/pgsql` 之下。

   可以通过向`configure`提供下列一个或多个命令行选项来自定义构建和安装过程：

   `--prefix=`*`PREFIX`*

   把所有文件安装在目录*`PREFIX`*而不是`/usr/local/pgsql`之下。实际文件会安装到各个子目录中；任何文件都不会直接安装到*`PREFIX`*目录本身。

   如果你有特殊需求，还可以用下面的选项自定义各个子目录。

`--exec-prefix=EXEC-PREFIX`

可以把体系结构相关的文件安装在不同于*PREFIX*所设置的另一个前缀*EXEC-PREFIX*之下。这在多台主机共享体系结构无关文件时很有用。如果省略此选项，*EXEC-PREFIX*将被设置为等于*PREFIX*，体系结构相关和无关文件将安装到同一个目录树下，这多半正是你想要的。

`--bindir=DIRECTORY`

指定可执行程序所在的目录。默认值为*EXEC-PREFIX*/bin，通常即/usr/local/pgsql/bin。

`--datadir=DIRECTORY`

已安装程序使用的只读数据文件所在的目录。默认值为*PREFIX*/share。注意这与数据库文件放在哪里无关。

`--sysconfdir=DIRECTORY`

各种配置文件所在的目录，默认为*PREFIX*/etc。

`--libdir=DIRECTORY`

安装库和动态可装载模块的位置。默认值为*EXEC-PREFIX*/lib。

`--includedir=DIRECTORY`

安装 C 和 C++ 头文件的目录。默认值为*PREFIX*/include。

`--docdir=DIRECTORY`

文档文件（“man”页除外）将安装到此目录。默认为 *PREFIX*/doc。

`--mandir=DIRECTORY`

PostgreSQL附带的手册页将安装在该目录下各自的man*x*子目录中。默认值为*PREFIX*/man。

注意

为了能把 PostgreSQL 安装到共享安装位置（例如 /usr/local/include），同时又不干扰系统其他部分的名字空间，我们做了特别处理。首先，除非完整展开后的目录名已经包含字符串“postgres”或“pgsql”，否则会自动把字符串“/postgresql”追加到 datadir、sysconfdir 和 docdir 上。例如，如果你选择 /usr/local 作为前缀，那么文档会安装到 /usr/local/doc/postgresql；但如果前缀是 /opt/postgres，那么它会安装到 /opt/postgres/doc。客户端接口的公共 C 头文件会安装到 includedir 中，并且不会污染名字空间。内部头文件和服务器头文件则会安装到 includedir 下的私有目录中。关于如何访问这些头文件，请参见各接口自己的文档。最后，如果有需要，也会在 libdir 下创建私有子目录，用于存放动态可加载模块。

`--with-includes=DIRECTORIES`

*DIRECTORIES*是一个冒号分隔的目录列表，这些目录将被加入编译器搜索头文件的路径列表。如果把可选软件包（如 GNU Readline）安装在非标准位置，就必须使用此选项，并且很可能还要使用相应的`--with-libraries`选项。

例如：`--with-includes=/opt/gnu/include:/usr/sup/include`。

`--with-libraries=DIRECTORIES`

*DIRECTORIES*是一个冒号分隔的搜索库的目录列表。如果软件包安装在非标准位置，你很可能需要使用此选项（以及相应的`--with-includes`选项）。

例如：`--with-libraries=/opt/gnu/lib:/usr/sup/lib`。

`--enable-nls[=LANGUAGES]`

启用本机语言支持（NLS），即以英语以外的语言显示程序消息的能力。*LANGUAGES* 是一个用空格分隔的语言代码列表，列出你想要支持的语言，例如 `--enable-nls='de fr'`。（你的列表与实际提供的翻译集合的交集会自动计算。）如果你不指定列表，则安装所有可用的翻译。

要使用此选项，需要一个 Gettext API 实现；参见上文。

`--with-pgport=NUMBER`

将 *NUMBER* 设为服务器和客户端的默认端口号。默认值是 5432。端口号以后始终都可以修改，但如果在这里指定，那么服务器和客户端都会编译进同一个默认值，这可能很方便。通常选择非默认值的唯一合理原因，是你打算在同一台机器上运行多个 PostgreSQL 服务器。

`--with-perl`

构建 PL/Perl 服务器端语言。

`--with-python`

构建 PL/Python 服务器端语言。

`--with-tcl`

构建需要 Tcl/Tk 的组件，即 libpgtcl、pgtclsh、pgtksh 和 PL/Tcl。但关于 `--without-tk` 请见下文。

`--without-tk`

如果你同时指定 `--with-tcl` 和此选项，那么需要 Tk 的程序（pgtksh）将被排除。

`--with-tclconfig=DIRECTORY`
`--with-tkconfig=DIRECTORY`

Tcl/Tk 会安装文件 `tclConfig.sh` 和 `tkConfig.sh`，其中包含构建与 Tcl 或 Tk 接口的模块所需的配置信息。这些文件通常会在其知名位置被自动找到，但如果你想使用不同版本的 Tcl 或 Tk，可以指定查找它们的目录。

`--with-java`

构建 JDBC 驱动及相关的 Java 软件包。

`--with-krb4[=DIRECTORY]`
`--with-krb5[=DIRECTORY]`

构建时加入 Kerberos 认证支持。你可以使用 Kerberos 版本 4 或 5，但不能同时使用。*DIRECTORY* 参数指定 Kerberos 安装的根目录；默认假定为 `/usr/athena`。如果相关的头文件和库不在一个共同的父目录下，你必须在此选项之外再使用 `--with-includes` 和 `--with-libraries` 选项。另一方面，如果所需文件位于默认会被搜索的位置（例如 `/usr/lib`），则可以省略该参数。

`configure` 会检查所需的头文件和库，以确保你的 Kerberos 安装满足要求，然后才继续。

`--with-krb-srvnam=NAME`

Kerberos 服务主体的名称。默认值为 `postgres`。可能没有理由更改它。

`--with-openssl[=DIRECTORY]`

构建时加入对 SSL（加密）连接的支持。这要求已安装 OpenSSL 软件包。*DIRECTORY* 参数指定 OpenSSL 安装的根目录；默认为 `/usr/local/ssl`。

`configure` 会检查所需的头文件和库，以确保你的 OpenSSL 安装满足要求，然后才继续。

`--with-pam`

构建时支持 PAM（可插拔认证模块）。

`--without-readline`

阻止使用 Readline 库。这将禁用 psql 中的命令行编辑和历史功能，因此不推荐。

`--with-rendezvous`

构建时加入 Rendezvous 支持。

`--disable-spinlocks`

即使 PostgreSQL 对该平台没有 CPU 自旋锁支持，也允许构建成功。缺乏自旋锁支持会导致性能低下；因此，只有在构建中止并告知你该平台缺乏自旋锁支持时，
才应使用此选项。

`--enable-thread-safety`

使客户端库成为线程安全的。这允许 libpq 和 ECPG 程序中的并发线程安全地控制它们各自的私有连接句柄。

`--without-zlib`

阻止使用 Zlib 库。这将禁用 pg_dump 中的压缩支持。此选项仅为那些没有该库的罕见系统准备。

`--enable-debug`

将所有程序和库编译为带调试符号的版本。这意味着你可以在调试器中运行程序，以分析问题。这会显著增大安装后的可执行文件大小，而且在非 GCC 编译器上，通常还会禁用编译器优化，从而导致变慢。不过，保留这些符号对于处理可能出现的各种问题极其有帮助。目前，只有在你使用 GCC 的情况下，才建议在生产安装中使用该选项。但如果你在做开发工作或运行测试版，就应始终启用它。

`--enable-cassert`

在服务器中启用断言检查，用于测试许多“不可能发生”的条件。这对代码开发非常有价值，但这些测试会使速度略有下降。此外，启用这些测试并不一定会增强服务器稳定性！断言检查并未按严重程度分类，因此即使某个 bug 相对无害，只要触发了断言失败，仍会导致服务器重启。目前，该选项不建议用于生产环境，但如果你在做开发工作或运行测试版，就应当启用它。

`--enable-depend`

启用自动依赖跟踪。启用后，makefile 会在任何头文件被修改时，重新构建所有受影响的目标文件。如果你在做开发工作，这很有用；但如果你只是打算编译一次并安装，这只是额外的开销。目前该选项只在 GCC 下有效。

如果你更喜欢用 `configure` 所选之外的 C 编译器，可以把环境变量 `CC` 设为你选择的程序。默认情况下，`configure` 会选择 `gcc`，除非它不适用于该平台。类似地，你可以用 `CFLAGS` 变量覆盖默认的编译器标志。

可以在 `configure` 命令行上指定环境变量，例如：

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

2. **构建**

   要开始构建，输入

   ```
   gmake
   ```

   （记住要使用 GNU make。）构建过程需要 5 分钟到半小时不等的时间，具体取决于你的硬件。最后显示的一行应该是

   ```
   All of PostgreSQL is successfully made. Ready to install.
   ```

3. **回归测试**

   如果你想在安装之前测试新构建的服务器，可以在此时运行回归测试。回归测试是一套测试用例，用于验证 PostgreSQL 在你的机器上按开发者预期的方式运行。输入

   ```
   gmake check
   ```

   （以 root 身份无法运行；请以非特权用户身份执行。）第 26 章包含关于解释测试结果的详细信息。你可以在之后任何时候通过执行相同的命令来重复此测试。

4. **安装文件**

> **注意**
>
> 如果你是在升级现有系统并打算把新文件安装在旧文件之上，
> 那么现在你应当已经按上面第 14.4 节的说明备份了数据并关闭了旧服务器。

要安装 PostgreSQL，输入

```
gmake install
```

这会把文件安装到步骤 1中指定的目录中。请确保你有写入该区域的适当权限。通常你需要以 root 身份执行这一步。或者，你也可以提前创建目标目录，并安排授予适当的权限。

你可以用 `gmake install-strip` 代替 `gmake install`，以便在安装时剥离可执行文件和库中的符号。这会节省一些空间。如果你构建时启用了调试支持，
剥离操作会实际移除调试支持，因此只应在不再需要调试时进行。`install-strip` 会尽力节省空间，但它并不能完全掌握如何从可执行文件中剥离每一个多余的字节，
所以如果想尽可能节省所有磁盘空间，你还得做一些手工工作。

标准安装只提供客户端应用开发所需的头文件。
如果你打算做任何服务器端程序开发（例如用 C 编写的自定义函数或数据类型），那么你可能想把整个 PostgreSQL include 树安装到你的目标 include 目录。为此，输入

```
gmake install-all-headers
```

这会给安装增加一两 MB 的占用，而且只有当你不打算保留整个源码树供参考时才有用。（如果保留，构建服务器端软件时直接使用源码的 include 目录即可。）

仅客户端安装：. 如果你只想安装客户端应用和接口库，可以使用这些命令：

```
gmake -C src/bin installgmake -C src/include installgmake -C src/
interfaces installgmake -C doc install
```

卸载：. 要撤销安装，可使用命令 `gmake uninstall`。但是，这不会删除任何已创建的目录。

清理：. 安装之后，你可以用命令 `gmake clean` 从源码树中删除构建出来的文件以腾出空间。这会保留 `configure` 程序生成的文件，这样你以后还能用 `gmake` 重新构建一切。要把源码树重置为分发时的状态，使用 `gmake distclean`。如果你打算从同一个源码树为多个平台构建，就必须为每次构建执行此操作并重新配置。

如果你执行了构建，之后发现 `configure` 选项有误，或者你更改了 `configure` 所考察的任何内容（例如软件升级），那么最好在重新配置和重新构建之前先执行 `gmake distclean`。否则，你在配置选择上的更改可能无法传播到所有需要的地方。

14.6. 安装后设置

14.6.1. 共享库

在一些有共享库的系统上（大多数系统都有），你需要告诉系统如何找到新安装的共享库。
不需要这样做的系统包括 BSD/OS、FreeBSD、HP-UX、IRIX、Linux、NetBSD、OpenBSD、Tru64 UNIX（原 Digital UNIX）和 Solaris。

设置共享库搜索路径的方法因平台而异，但最常见的方法是设置环境变量 `LD_LIBRARY_PATH`，方法如下。在 Bourne shell（`sh`，`ksh`，`bash`，`zsh`）：

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

或者在 `csh` 或 `tcsh`：

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

请将 `/usr/local/pgsql/lib` 替换为你为 `--libdir` 指定的值，设置过程见步骤 1。应将这些命令放入 shell 启动文件，例如 `/etc/profile` 或 `~/.bash_profile`。关于此方法相关注意事项的一些有用信息，参见http://www.visi.com/~barr/ldpath.html。

在某些系统上，更好的做法可能是在构建之前设置环境变量 `LD_RUN_PATH`。

在 Cygwin 上，请把库目录加入 `PATH`，或者把 `.dll` 文件移到 `bin` 目录中。

如果不确定，请参考你系统的手册页（可能是 `ld.so` 或 `rld`）。如果你随后收到类似下面这样的消息

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or
 directory
```

那就说明这一步确实是必需的。到那时再处理即可。

如果你使用的是 BSD/OS、Linux 或 SunOS 4，并且拥有 root 权限，可以在安装后运行：

```
/sbin/ldconfig /usr/local/pgsql/lib
```

（或等价目录），使运行时链接器能更快地找到共享库。更多信息参见 `ldconfig` 的手册页。在 FreeBSD、NetBSD 和 OpenBSD 上，命令则是：

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

其他系统据知没有等价的命令。

14.6.2. 环境变量

如果你安装到了 `/usr/local/pgsql` 或其他默认不搜索程序的位置，就应该把 `/usr/local/pgsql/bin`（或你在步骤 1中为 `--bindir` 设置的值）加入 `PATH`。严格来说，这不是必需的，但这会让使用 PostgreSQL 方便得多。

为此，请把下面内容加到你的 shell 启动文件中，例如 `~/.bash_profile`（如果你希望影响所有用户，则用 `/etc/profile`）：

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

如果你使用的是 `csh` 或 `tcsh`，则使用这条命令：

```
set path = ( /usr/local/pgsql/bin $path )
```

要让系统能找到 man 文档，你需要把如下所示的行加入 shell 启动文件，除非你安装到了默认会搜索的位置：

```
MANPATH=/usr/local/pgsql/man:$MANPATH
export MANPATH
```

环境变量 PGHOST 和 PGPORT 会告诉客户端应用数据库服务器的主机和端口，从而覆盖编译时内置的默认值。如果你准备从远端运行客户端应用，那么让每个打算使用数据库的用户都设置 PGHOST 会比较方便。不过这不是必需的；对于大多数客户端程序，也可以通过命令行选项传递这些设置。

14.7. 受支持的平台

开发者社区已验证 PostgreSQL 能在下面列出的平台上运行。受支持的平台通常意味着 PostgreSQL 能按这些说明构建和安装，并且回归测试通过。

> **注意**
>
> 如果你在受支持的平台上安装时遇到问题，请写信给 <pgsql-bugs@postgresql.org>，而不是写信给这里列出的人。

OS	处理器	版本	报告	备注
AIX	RS6000	7.4	2003-10-25, Hans-Jürgen Schönig (<hs@cybertec.at>)	参见 doc/FAQ_AIX
BSD/OS	x86	7.4	2003-10-24, Bruce Momjian (<pgman@candle.pha.pa.us>)	4.3
FreeBSD	Alpha	7.4	2003-10-25, Peter Eisentraut (<peter_e@gmx.net>)	4.8
FreeBSD	x86	7.4	2003-10-24, Peter Eisentraut (<peter_e@gmx.net>)	4.9
HP-UX	PA-RISC	7.4	2003-10-31, 10.20, Tom Lane (<tgl@sss.pgh.pa.us>); 2003-11-04, 11.00, Peter Eisentraut (<peter_e@gmx.net>)	gcc 和 cc；另请参阅 doc/FAQ_HPUX
IRIX	MIPS	7.4	2003-11-12, Robert E. Bruccoleri (<bruc@stone.congenomics.com>)	6.5.20, 仅 cc
Linux	Alpha	7.4	2003-10-25, Noèl Köthe (<noel@debian.org>)	2.4
Linux	arm41	7.4	2003-10-25, Noèl Köthe (<noel@debian.org>)	2.4

OS	处理器	版本	报告	备注
Linux	Itanium	7.4	2003-10-25, Noèl Köthe (<noel@debian.org>)	2.4
Linux	m68k	7.4	2003-10-25, Noèl Köthe (<noel@debian.org>)	2.4
Linux	MIPS	7.4	2003-10-25, Noèl Köthe (<noel@debian.org>)	2.4
Linux	Opteron	7.4	2003-11-01, Jani Averbach (<jaa@cc.jyu.fi>)	2.6
Linux	PPC	7.4	2003-10-25, Noèl Köthe (<noel@debian.org>)	
Linux	S/390	7.4	2003-10-25, Noèl Köthe (<noel@debian.org>)	2.4
Linux	Sparc	7.4	2003-10-24, Peter Eisentraut (<peter_e@gmx.net>)	2.4, 32-bit
Linux	x86	7.4	2003-10-24, Peter Eisentraut (<peter_e@gmx.net>)	2.4
Mac OS X	PPC	7.4	2003-10-24, 10.2.8, Adam Witney (<awitney@sghms.ac.uk>), 10.3, Marko Karppinen (<marko@karppinen.fi>)	
NetBSD	arm32	7.4	2003-11-12, Patrick Welche (<prlw1@newn.cam.ac.uk>)	1.6ZE/acorn32
NetBSD	Sparc	7.4.1	2003-11-26, Peter Eisentraut (<peter_e@gmx.net>)	1.6.1, 32-bit
NetBSD	x86	7.4	2003-10-24, Peter Eisentraut (<peter_e@gmx.net>)	1.6
OpenBSD	Sparc	7.4	2003-11-01, Peter Eisentraut (<peter_e@gmx.net>)	3.4
OpenBSD	x86	7.4	2003-10-24, Peter Eisentraut (	3.2

OS	处理器	版本	报告	备注
			<peter_e@gmx.net>)	
Solaris	Sparc	7.4	2003-10-26, Christopher Browne (<cbbrowne@libertyrms.info>)	2.8；另请参阅 doc/FAQ_Solaris
Solaris	x86	7.4	2003-10-26, Kurt Roeckx (<Q@ping.be>)	2.6；另请参阅 doc/FAQ_Solaris
Tru64 UNIX	Alpha	7.4	2003-10-25, 5.1b, Peter Eisentraut (<peter_e@gmx.net>); 2003-10-29, 4.0g, Alessio Bragadini (<alessio@albourne.com>)	
UnixWare	x86	7.4	2003-11-03, Larry Rosenman (<ler@lerctr.org>)	7.1.3；连接测试可能失败，另请参阅 doc/FAQ_SCO
Windows with Cygwin	x86	7.4	2003-10-24, Peter Eisentraut (<peter_e@gmx.net>)	参见 doc/FAQ_MSWIN
Windows	x86	7.4	2003-10-27, Dave Page (<dpage@vale-housing.co.uk>)	本机方式只有客户端，见第 15 章

不支持的平台：. 下面的平台要么已知无法工作，要么曾在以前的版本中可以工作，但编制此列表时我们没有收到在 7.4 版本上测试成功的明确确认。我们把它们列在这里，是让你知道这些平台只要稍加关注就可能受到支持。

OS	处理器	版本	报告	备注
BeOS	x86	7.2	2001-11-29, Cyril Velter (<cyril.velter@libertysurf.fr>)	需要更新信号量代码
Linux	PlayStation 2	7.4	2003-11-02, Peter Eisentraut (<peter_e@gmx.net>)	需要新的 config.guess、--disable-spinlocks、#undef HAS_TEST_AND_SET、禁用 tas_dummy()
Linux	PA-RISC	7.4	2003-10-25, Noèl Köthe (<noel@debian.org>)	需要 --disable-spinlocks，其余正常
NetBSD	Alpha	7.2	2001-11-20, Thomas Thai (<tom@minnesota.com>)	1.5W

OS	处理器	版本	报告	备注
NetBSD	MIPS	7.2.1	2002-06-13, Warwick Hunter (<whunter@agile.tv>)	1.5.3
NetBSD	PPC	7.2	2001-11-28, Bill Studenmund (<wrstuden@netbsd.org>)	1.5
NetBSD	VAX	7.1	2001-03-30, Tom I. Helbekkmo (<tih@kpnQwest.no>)	1.5
QNX 4 RTOS	x86	7.2	2001-12-10, Bernd Tegge (<tegge@repas-aeg.de>)	需要更新信号量代码; 参见 doc/FAQ_QNX4
QNX RTOS v6	x86	7.2	2001-11-20, Igor Kovalenko (<Igor.Kovalenko@motorola.com>)	补丁在邮件列表存档中可用，但未赶上 7.2
SCO OpenServer	x86	7.3.1	2002-12-11, Shibashish Satpathy (<shib@postmark.net>)	5.0.4, gcc; 参见 doc/FAQ_SCO
SunOS 4	Sparc	7.2	2001-12-04, Tatsuo Ishii (<t-ishii@sra.co.jp>)	

第 15 章 在 Windows 上安装

虽然 PostgreSQL 是为类 Unix 操作系统编写的，但 C 客户端库（libpq）和交互式终端（psql）可以在 Windows 下原生编译。源码发行版中包含的 makefile 是为 Microsoft Visual C++ 编写的，可能在其他系统上不能用。在其他情况下应当可以手动编译这些库。

> **提示**
>
> 如果你使用 Windows 98 或更新版本，可以先安装 Cygwin 工具包，然后“以 Unix 的方式”构建并使用完整的 PostgreSQL。这种情况下请参阅第 14 章。

要构建在 Windows 上能构建的一切，进入 `src` 目录并输入命令

```
nmake /f win32.mak
```

这假设你的路径中有 Visual C++。

将构建出下列文件：

`interfaces\libpq\Release\libpq.dll`

可动态链接的前端库

`interfaces\libpq\Release\libpqdll.lib`

把你的程序链接到 `libpq.dll` 的导入库

`interfaces\libpq\Release\libpq.lib`

前端库的静态库版本

`bin\psql\Release\psql.exe`

PostgreSQL 交互式终端

真正需要安装的只有 `libpq.dll` 库。在大多数情况下，这个文件应当放在 `WINNT\SYSTEM32` 目录中（在 Windows 95/98/ME 系统上则放在 `WINDOWS\SYSTEM` 中）。如果这个文件是用安装程序安装的，应当使用文件中包含的 `VERSIONINFO` 资源带版本检查地安装，以确保不会覆盖较新版本的库。

如果你计划在这台机器上使用 libpq 进行开发，你必须把源码树的 `src\include` 和 `src\interfaces\libpq` 子目录加到你的编译器设置的 include 路径中。

要使用这个库，你必须把 `libpqdll.lib` 文件加到你的项目中。（在 Visual C++ 中，只需右键点击项目并选择添加它。）

psql 被编译为一个“控制台应用程序”。由于 Windows 控制台窗口使用的编码与系统其余部分不同，在 psql 提示符下使用 8 位字符时必须特别小心。当 psql 检测到有问题的控制台代码页时，它会在启动时警告你。要更改控制台代码页，需要做两件事：

- 输入 **`cmd.exe /c chcp 1252`** 设置代码页。（1252 是适合德语的代码页；用你的值替换它。）如果你使用 Cygwin，可以把这个命令放在 `/etc/profile` 中。
- 把控制台字体设置为“Lucida Console”，因为点阵字体不能与 ANSI 代码页一起工作。

第 16 章 服务器运行时环境

本章讨论如何设置和运行数据库服务器，以及它与操作系统的交互。

16.1. PostgreSQL 用户账户

与任何可从外部访问的服务器守护进程一样，建议使用单独的用户账户运行 PostgreSQL。该用户账户应该只拥有服务器管理的数据，不应与其他守护进程共用。（例如，使用 nobody 用户就不是好主意。）不建议将可执行文件安装为由该用户所有，因为系统一旦被攻破，
就可能修改自己的二进制文件。

要在系统中添加一个 Unix 用户账户，请查找useradd或 adduser命令。用户名postgres 很常用，但并非必须。

16.2. 创建一个数据库集簇

在做任何事情之前，必须先在磁盘上初始化一个数据库存储区域。我们把它称为数据库集簇。（SQL 标准使用术语目录集簇（catalog cluster）。）数据库集簇是由单个运行中的数据库服务器实例访问的一组数据库。初始化之后，数据库集簇中会包含一个名为template1的数据库。顾名思义，它会作为后续新建数据库的模板；不应将它用于实际工作。
（关于创建数据库的信息，见第 18 章。）

从文件系统角度看，数据库集簇就是一个目录，所有数据都存储在其下。我们将它称为
数据目录或数据区域。数据存放位置完全由你决定，没有默认值，不过诸如/usr/local/pgsql/data或/var/lib/pgsql/data这样的位置比较常用。要初始化数据库集簇，
请使用随 PostgreSQL 一起安装的命令initdb。数据库系统在文件系统中的目标位置通过 -D 选项指定，例如：

```
$initdb -D /usr/local/pgsql/data
```

注意，必须以 PostgreSQL 用户账户登录后再执行此命令，该账户已在上一节介绍。

> **提示**
>
> 除了使用 -D 选项，也可以设置环境变量 PGDATA。

如果指定的目录尚不存在，initdb 会尝试创建它。它很可能没有这样做的权限（如果你听从了我们的建议，创建的是非特权账户）。这种情况下，你应该（以 root 身份）自行创建该目录，并将其所有者改为 PostgreSQL 用户。过程可能如下所示：

```
root# mkdir /usr/local/pgsql/data
root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

如果数据目录看起来已经被初始化过，initdb 将拒绝运行。

由于数据目录包含存储在数据库中的全部数据，必须保护它免遭未经授权的访问。因此，initdb 会撤销除 PostgreSQL 用户以外所有人的访问权限。

不过，尽管目录内容是安全的，默认的客户端认证设置却允许任何本地用户连接到数据库，甚至成为数据库超级用户。如果你不信任其他本地用户，我们建议使用 `initdb` 的 `-W` 或 `--pwprompt` 选项为数据库超级用户指定口令。运行 `initdb` 之后，在首次启动服务器之前修改 `pg_hba.conf` 文件，把 `trust` 认证改为 `md5` 或 `password`。（其他方法包括使用 `ident` 认证或文件系统权限来限制连接。更多信息见第 19 章。）

`initdb` 还会初始化数据库集簇的默认区域设置。通常，它会直接采用环境中的区域设置，并将其应用于初始化的数据库。可以为数据库指定不同的区域设置；更多信息见第 20.1 节。特定数据库集簇中使用的排序顺序由 `initdb` 设置，事后无法更改，除非转储全部数据、重新运行 `initdb` 再重新装载数据。因此，一开始就作出正确选择很重要。

16.3. 启动数据库服务器

在任何人访问数据库之前，必须先启动数据库服务器。数据库服务器程序名为 `postmaster`。`postmaster` 必须知道应到哪里查找它要使用的数据。这通过 `-D` 选项指定。因此，启动服务器最简单的方法是：

```
$ postmaster -D /usr/local/pgsql/data
```

这会让服务器在前台运行。必须以 PostgreSQL 用户账户登录后再执行此操作。如果不使用 `-D`，服务器会尝试使用环境变量 `PGDATA` 所指定的数据目录。如果两者都未提供，则会失败。

要在后台启动 `postmaster`，可使用常见的 shell 语法：

```
$ postmaster -D /usr/local/pgsql/data >logfile 2>&1 &
```

如上所示，把服务器的stdout和 stderr输出保存到某个地方非常重要。这有助于审计，也有助于诊断问题。（关于日志文件处理的更深入讨论，见第 21.3 节。）

`postmaster` 还接受许多其他命令行选项。更多信息请见参考页以及下面的第 16.4 节。特别是，要让服务器接受 TCP/IP 连接（而不只是 Unix 域套接字连接），必须指定 `-i` 选项。

这些 shell 语法很快就会让人觉得繁琐。因此提供了 shell 脚本包装器 `pg_ctl` 来简化一些任务。例如：

```
pg_ctl start -l logfile
```

会在后台启动服务器，并把输出写入指定的日志文件。这里的 `-D` 选项含义与在 `postmaster` 中相同。`pg_ctl` 还可用于停止服务器。

通常，你会希望在计算机启动时就启动数据库服务器。自动启动脚本依赖于操作系统。PostgreSQL 在 `contrib/start-scripts` 目录中附带了一些示例脚本。安装这些脚本可能需要 root 权限。

不同系统在引导时启动守护进程的惯例各不相同。许多系统有 `/etc/rc.local`或`/etc/rc.d/rc.local` 文件，其他系统则使用`rc.d`目录。无论采用哪种方式，服务器都必须由PostgreSQL 用户账户而不是 root或其他用户来启动。因此，你大概应该在命令中使用 `su -c '...' postgres` 这种形式。例如：

```
su -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog' postgres
```

下面是一些与操作系统相关的补充建议。（务必把它们替换为正确的安装目录和用户名。）

- 在FreeBSD上，请查看PostgreSQL 源码发布包中的`contrib/start-scripts/freebsd`文件。

- 在OpenBSD上，把以下内容加入 /etc/rc.local文件：

  ```
  if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/
  postmaster ]; then
      su - -c '/usr/local/pgsql/bin/pg_ctl start -l /var/postgresql/
  log -s' postgres
      echo -n ' postgresql'
  fi
  ```

- 在Linux系统上，可以把

  ```
  /usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/
  data
  ```

  加入/etc/rc.d/rc.local，或者查看 PostgreSQL源码发布包中的 contrib/start-scripts/linux文件。

- 在NetBSD上，可以根据喜好使用 FreeBSD或Linux 的启动脚本。

- 在Solaris上，创建一个名为 /etc/init.d/postgresql的文件，其中包含以下行：

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /
  usr/local/pgsql/data"
  ```

  然后在/etc/rc3.d中创建一个指向它的符号链接 S99postgresql。

当 postmaster 运行时，其PID会保存在数据目录中的postmaster.pid文件里。这可以防止多个 postmaster 进程在同一个数据目录上运行，也可以用来关闭 postmaster 进程。

16.3.1. 服务器启动失败

服务器启动失败有几种常见原因。请检查服务器日志文件，
或者手工启动服务器（不要重定向标准输出和标准错误），看看出现了什么错误消息。
下面将更详细地解释一些最常见的错误消息。

```
LOG:  could not bind IPv4 socket: Address already in use
HINT:  Is another postmaster already running on port 5432? If not,
 wait a few seconds and retry.
FATAL:  could not create TCP/IP listen socket
```

这通常就是字面上的意思：你试图在一个已有 postmaster 运行的端口上再启动另一个 postmaster。不过，如果内核错误消息不是Address already in use 或类似变体，也可能是别的问题。例如，试图在一个保留端口号上启动 postmaster，可能会得到类似下面的消息：

```
$ postmaster -i -p 666
LOG:  could not bind IPv4 socket: Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait
 a few seconds and retry.
FATAL:  could not create TCP/IP listen socket
```

像下面这样的消息：

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640,
 03600).
```

可能表示内核对共享内存大小的限制小于 PostgreSQL 尝试创建的工作区（本例中为 4011376640 字节）。也可能表示内核中根本没有配置 System V 风格的共享内存支持。作为临时解决办法，可以尝试以少于通常数量的缓冲区启动服务器（`-B` 开关）。最终还是需要重新配置内核，增大允许的共享内存大小。如果尝试在同一台机器上启动多个服务器，而它们请求的总空间超出内核限制，也可能看到这条消息。

像下面这样的错误：

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

并不意味着你已经用光了磁盘空间。它表示你的内核对System V信号量数量的限制，小于PostgreSQL想要创建的数量。和上面一样，你也许可以通过减少允许的连接数（`-N` 开关）来暂时绕过这个问题，但最终还是应该提高内核限制。

如果收到“illegal system call”错误，很可能是内核根本不支持共享内存或信号量。在这种情况下，唯一的办法就是重新配置内核以启用这些功能。

关于配置System V IPC功能的细节请见第 16.5.1 节。

16.3.2. 客户端连接问题

客户端一侧可能出现的错误种类很多，并且依赖于具体应用，但其中有一些可能直接与服务器的启动方式有关。除下面列出的几种情况外，其他问题应查阅相应客户端应用的文档。

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

这是常见的“我找不到可通信的服务器”失败。尝试 TCP/IP 通信时，它就会显示为上述形式。一个常见错误是忘记配置服务器以允许 TCP/IP 连接。

另一种情况是，尝试通过 Unix 域套接字与本地服务器通信时，会得到以下消息：

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

最后一行有助于确认客户端是否正尝试连接到正确的位置。如果那里实际上没有运行服务器，内核错误消息通常会像示例那样是 `Connection refused` 或 `No such file or directory`。（要注意，在这里 `Connection refused` 并不表示服务器收到了你的连接请求并拒绝了它；那种情况会产生另一条消息，如第 19.3 节所示。）其他错误消息，例如 `Connection timed out`，可能表示更底层的问题，例如网络不通。

16.4. 运行时配置

有大量配置参数会影响数据库系统的行为。在本小节中，我们描述如何设置配置参数；随后的小节将逐一详细讨论每个参数。

所有参数名都不区分大小写。每个参数取以下四种类型之一的值：布尔、整数、浮点和字符串。布尔值可以是 `ON`、`OFF`、`TRUE`、`FALSE`、`YES`、`NO`、`1`、`0`（不区分大小写）或其中任何一个无歧义的前缀。

设置这些参数的一种方法是编辑数据目录中的 `postgresql.conf` 文件。（安装时会在此处放置一个默认文件。）该文件的内容可能类似于：

```
# This is a comment
log_connections = yes
syslog = 2
search_path = '$user, public'
```

每行指定一个参数。名称和值之间的等号是可选的。空白无关紧要，空行会被忽略。井号（#）可在任何位置引入注释。不是简单标识符或数字的参数值应当用单引号引起来。

每当 postmaster 进程收到 SIGHUP 信号时（用 pg_ctl reload 发送最为简单），配置文件就会被重新读取。postmaster 还会把该信号传播给所有当前运行的服务器进程，使现有会话也获得新值。另外，你也可以直接把信号发送给单个服务器进程。

设置这些配置参数的第二种方法是把它们作为命令行选项传给 postmaster，例如：

```
postmaster -c log_connections=yes -c syslog=2
```

命令行选项覆盖 postgresql.conf 中任何与之冲突的设置。

有时，只给某个特定会话指定命令行选项也很有用。为此可以在客户端一侧使用环境变量 PGOPTIONS：

```
env PGOPTIONS='-c geqo=off' psql
```

（这适用于任何基于 libpq 的客户端应用程序，不只是 psql。）注意，对那些在服务器启动时已固定的参数（例如端口号），这种方式不起作用。

此外，还可以把一组选项设置指派给某个用户或某个数据库。每当会话启动时，会加载相关用户和数据库的默认设置。配置这些设置分别使用 ALTER DATABASE 和 ALTER USER 命令。每数据库设置覆盖从 postmaster 命令行或配置文件获得的任何设置，并依次被每用户设置覆盖；两者都会被每会话选项覆盖。

有些参数可以在单个 SQL 会话中用SET命令更改，例如：

```
SET ENABLE_SEQSCAN TO OFF;
```

如果允许 SET，它将覆盖该参数所有其他来源的值。超级用户被允许 SET 的值比普通用户更多。

SHOW命令允许查看所有参数的当前值。

虚拟表 pg_settings（见第 43.34 节）也允许显示和更新会话运行时参数。它等价于 SHOW 和 SET，但用起来可能更方便，因为它可以与其他表连接，或用任何想要的选择条件进行查询。

16.4.1. 连接和认证

16.4.1.1. 连接设置

tcpip_socket (boolean)

如果为真，服务器将接受 TCP/IP 连接。否则只接受本地 Unix 域套接字连接。默认是关闭的。这个选项只能在服务器启动时设置。

max_connections (integer)

决定数据库服务器允许的最大并发连接数。默认值通常是 100，但如果内核设置不支持（在 initdb 期间确定），则可能更少。这个参数只能在服务器启动时设置。

增加这个参数可能会导致PostgreSQL请求比操作系统的默认配置所允许的更多的System V共享内存或信号量。如有必要，关于如何调整这些参数的信息见第 16.5.1 节。

`superuser_reserved_connections` (`integer`)

决定为 PostgreSQL 超级用户的连接保留多少个“连接槽”。同时活跃的连接数最多始终只能达到 `max_connections`。每当活跃并发连接数至少达到 `max_connections` 减去 `superuser_reserved_connections` 时，新连接将只接受超级用户。

默认值是 2。该值必须小于 `max_connections` 的值。这个参数只能在服务器启动时设置。

`port` (`integer`)

服务器监听的 TCP 端口，默认是 5432。这个选项只能在服务器启动时设置。

`unix_socket_directory` (`string`)

指定服务器用于监听来自客户端应用连接的 Unix 域套接字所在的目录。默认值通常是 `/tmp`，但可以在构建时更改。

`unix_socket_group` (`string`)

设置 Unix 域套接字的所属组。（套接字的所属用户总是启动服务器的用户。）与选项 `unix_socket_permissions` 组合，可以作为这种套接字类型的一种附加访问控制机制。默认是一个空字符串，表示使用当前用户的默认组。这个选项只能在服务器启动时设置。

`unix_socket_permissions` (`integer`)

设置 Unix 域套接字的访问权限。Unix 域套接字使用通常的 Unix 文件系统权限集。选项值应是以 `chmod` 和 `umask` 系统调用所接受格式指定的数字权限模式。（要使用惯用的八进制格式，数字必须以 `0`（零）开头。）

默认权限是 `0777`，表示任何人都可以连接。合理的其他取值包括 `0770`（仅用户和组，另见 `unix_socket_group`）和 `0700`（仅用户）。（注意，对 Unix 域套接字而言，实际上只有写权限起作用，设置或撤销读权限和执行权限没有意义。）

此访问控制机制独立于第 19 章中描述的机制。

这个选项只能在服务器启动时设置。

`virtual_host` (`string`)

指定服务器用于监听来自客户端应用连接的主机名或 IP 地址。默认是监听所有已配置的地址（包括 localhost）。

`rendezvous_name` (`string`)

指定 Rendezvous 广播名。默认使用计算机名，即指定为 ''。

16.4.1.2. 安全和认证

`authentication_timeout` (`integer`)

完成客户端认证的最长时间，以秒为单位。
如果一个潜在客户端没有在这段时间里完成认证协议，服务器将断开连接。
这防止了挂起的客户端无限期占用连接。这个选项只能在服务器启动时或在 `postgresql.conf` 文件中设置。默认值是 60。

`ssl` (`boolean`)

启用SSL连接。使用前请阅读第 16.7 节。默认是关闭的。

`ssl_renegotiation_limit` (`integer`)

指定在一个SSL加密连接上可以流过多少数据之后将进行会话的重新协商。
在发送大量数据时，重新协商能降低被密码分析的机会，但它也会带来很大的性能损失。
该限制以发送和接收流量的总和来检查。如果这个参数被设置为 0，则禁用重新协商。默认值为 `512MB`。

> **注意**
>
> 2009 年 11 月之前的 SSL 库在使用 SSL 重新协商时不安全，原因是 SSL 协议中的一个漏洞。作为针对该漏洞的权宜修复，一些厂商发布的 SSL 库不具备重新协商能力。如果在客户端或服务器上正在使用这类库，应当禁用 SSL 重新协商。

`password_encryption` (`boolean`)

在 `CREATE USER` 或 `ALTER USER` 中指定口令，且既没有写 `ENCRYPTED` 也没有写 `UNENCRYPTED` 时，这个选项决定口令是否被加密。默认是打开的（加密口令）。

`krb_server_keyfile` (`string`)

设置 Kerberos 服务器密钥文件的位置。详情见第 19.2.3 节。

`db_user_namespace` (`boolean`)

此参数允许各数据库独立的用户名。默认是关闭的。

如果启用此参数，你应以`username@dbname`的形式创建用户。
连接的客户端传入`username`时，会在用户名后附加`@`和数据库名，
然后由服务器查找此数据库专属的用户名。请注意，在 SQL 环境中创建名称包含 `@`的用户时，需要用引号括起用户名。

启用此参数后，仍然可以创建普通的全局用户。只需在客户端指定用户名时附加`@`。
服务器查找用户名之前会去掉 `@`。

> **注意**
>
> 此特性旨在作为找到完整解决方案之前的临时措施。届时将移除此选项。

16.4.2. 资源消耗

16.4.2.1. 内存

`shared_buffers` (`integer`)

设置数据库服务器使用的共享内存缓冲区数量。默认值通常是 1000，但如果内核设置不支持（在 initdb 期间确定），则可能更少。除非在构建服务器时选择了不同的 `BLCKSZ` 值，

每个缓冲区为 8192 字节。此设置必须至少为 16，并且至少是 max_connections 值的两倍；不过，要获得良好的性能，通常需要显著高于最小值的设置。对于生产安装，建议取几千的值。这个选项只能在服务器启动时设置。

增加这个参数可能会导致PostgreSQL请求比操作系统的默认配置所允许的更多的System V共享内存。如有必要，关于如何调整这些参数的信息见第 16.5.1 节。

sort_mem (integer)

指定内部排序操作和哈希表在切换到临时磁盘文件之前可使用的内存量。
值以千字节为单位指定，默认为 1024 千字节（1 MB）。注意，对于复杂查询，可能同时运行多个排序或哈希操作；每一个在被允许开始把数据放入临时文件之前，
都可以使用多达该值所指定的内存。此外，多个运行中的会话可能同时进行排序操作。
因此，使用的总内存可能是 sort_mem 值的许多倍。排序操作由 ORDER BY、归并连接和 CREATE INDEX 使用。哈希表用于哈希连接、基于哈希的聚合以及基于哈希的 IN 子查询处理。由于恢复数据库时会使用 CREATE INDEX，在进行大型恢复操作之前增大 sort_mem 可以提升性能。

vacuum_mem (integer)

指定 VACUUM 用于跟踪待回收行所用的最大内存量。值以千字节为单位指定，默认为 8192 kB。对包含大量已删除行的大表进行清理时，更大的设置可能提升清理速度。

16.4.2.2. 空闲空间映射

max_fsm_pages (integer)

设置共享空闲空间映射中为其跟踪空闲空间的磁盘页最大数量。每个页槽占用 6 字节共享内存。此设置必须大于 16 * max_fsm_relations。默认值是 20000。这个选项只能在服务器启动时设置。

max_fsm_relations (integer)

设置共享空闲空间映射中为其跟踪空闲空间的关系（表和索引）的最大数量。
每个槽大约占用 50 字节共享内存。默认值是 1000。这个选项只能在服务器启动时设置。

16.4.2.3. 内核资源使用

max_files_per_process (integer)

设置每个服务器子进程允许同时打开的最大文件数。默认值是 1000。如果内核正在强制执行一个安全的每进程限制，则无需担心此设置。但在某些平台上（特别是大多数 BSD 系统），当大量进程都试图打开那么多文件时，
内核会允许单个进程打开远超系统真正能够支持的文件数。如果你发现出现“Too many open files”失败，可尝试调低此设置。这个选项只能在服务器启动时设置。

preload_libraries (string)

此变量指定服务器启动时预装载的一个或多个共享库。
可以选择为每个库调用一个无参数的初始化函数。要指定它，
在库名后加冒号和初始化函数名。例如 '$libdir/mylib:mylib_init' 会导致预装载 mylib 并执行 mylib_init。如果要装载多个库，用逗号分隔它们的名称。

如果找不到 mylib 或 mylib_init，服务器将无法启动。

PostgreSQL 过程语言库可以用这种方式预装载，通常使用语法 '$libdir/plXXX:plXXX_init'，其中 XXX 是 pgsql、perl、tcl 或 python。

通过预装载共享库（并在适用时初始化它），可以避免首次使用该库时的库启动时间。不过，每个新服务器进程的启动时间可能增加，即使该进程从未使用过这个库。

16.4.3. 预写式日志

关于 WAL 调优的细节另见第 25.3 节。

16.4.3.1. 设置

fsync (boolean)

如果此选项打开，PostgreSQL 服务器会在多处使用 fsync() 系统调用，确保更新被物理写入磁盘。这保证数据库集簇在操作系统或硬件崩溃之后能恢复到一致状态。（数据库服务器自身的崩溃与此无关。）

不过，使用 fsync() 会带来性能损失：当事务提交时，PostgreSQL 必须等待操作系统把预写日志刷到磁盘。禁用 fsync 后，操作系统被允许尽其所能地对写操作进行缓冲、排序和延迟。这可以显著提升性能。但是，如果系统崩溃，最近若干已提交事务的结果可能部分或全部丢失。最坏情况下，可能发生不可恢复的数据损坏。

由于存在这些风险，fsync 没有普适正确的设置。有些管理员总是禁用 fsync，另一些管理员只在批量装载时关闭它（批量装载出问题时存在明确的重启点），还有一些管理员始终启用 fsync。默认是启用 fsync，以获得最大可靠性。如果你信任你的操作系统、硬件和电力公司（或你的电池备份），可以考虑禁用 fsync。

这个选项只能在服务器启动时或在 postgresql.conf 文件中设置。

wal_sync_method (string)

用于强制把 WAL 更新刷到磁盘的方法。可能的取值有 fsync（每次提交时调用 fsync()）、fdatasync（每次提交时调用 fdatasync()）、open_sync（以 open() 选项 O_SYNC 写 WAL 文件）和 open_datasync（以 open() 选项 O_DSYNC 写 WAL 文件）。并非所有平台上这些选择都可用。这个选项只能在服务器启动时或在 postgresql.conf 文件中设置。

wal_buffers (integer)

共享内存中用于 WAL 日志的磁盘页缓冲区数量。默认值是 8。这个选项只能在服务器启动时设置。

16.4.3.2. 检查点

checkpoint_segments (integer)

自动 WAL 检查点之间的最大距离，以日志文件段为单位（每段通常为 16 兆字节）。默认值是三。这个选项只能在服务器启动时或在 postgresql.conf 文件中设置。

checkpoint_timeout (integer)

自动 WAL 检查点之间的最长时间，以秒为单位。默认值是 300 秒。这个选项只能在服务器启动时或在 postgresql.conf 文件中设置。

checkpoint_warning (integer)

如果检查点段文件写满所导致的检查点发生得比这个秒数更频繁，就向服务器日志写一条消息。默认值是 30 秒。零会关闭该警告。

commit_delay (integer)

向 WAL 缓冲区写入提交记录与把缓冲区刷到磁盘之间的时间延迟，以微秒为单位。非零的延迟允许多个事务只通过一次 fsync() 系统调用提交，前提是系统负载足够高，在给定的时间间隔内可能有额外的事务准备好提交。但如果没有其他事务准备好提交，该延迟就白白浪费了。因此，只有当服务器进程写入其提交记录的瞬间，至少有 commit_siblings 个其他事务处于活跃状态时，才执行该延迟。默认值是零（无延迟）。

commit_siblings (integer)

执行 commit_delay 延迟之前所要求的最少并发打开事务数。更大的值使在延迟间隔内至少有另一个事务准备好提交更有可能。默认值是五。

16.4.4. 查询规划

16.4.4.1. 规划器方法配置

注意

这些配置参数提供了一种影响查询优化器所选查询计划的粗略方法。如果优化器为某个查询选择的默认计划不是最优的，可以临时用其中一个配置参数强制优化器选择更好的计划。改进优化器所选计划质量的其他方法包括配置规划器代价常量、更频繁地运行 ANALYZE，以及使用 ALTER TABLE SET STATISTICS 增大为特定列收集的统计量。

enable_hashagg (boolean)

启用或禁用查询规划器对哈希聚合计划类型的使用。默认是打开的。这用于调试查询规划器。

enable_hashjoin (boolean)

启用或禁用查询规划器对哈希连接计划类型的使用。默认是打开的。这用于调试查询规划器。

enable_indexscan (boolean)

启用或禁用查询规划器对索引扫描计划类型的使用。默认是打开的。这用于调试查询规划器。

enable_mergejoin (boolean)

启用或禁用查询规划器对归并连接计划类型的使用。默认是打开的。这用于调试查询规划器。

enable_nestloop (boolean)

启用或禁用查询规划器对嵌套循环连接计划的使用。不可能完全压制嵌套循环连接，但关闭此变量会使规划器在有其他方法可用时不去使用它。默认是打开的。这用于调试查询规划器。

`enable_seqscan` (`boolean`)

启用或禁用查询规划器对顺序扫描计划类型的使用。不可能完全压制顺序扫描，但关闭此变量会使规划器在有其他方法可用时不去使用它。默认是打开的。这用于调试查询规划器。

`enable_sort` (`boolean`)

启用或禁用查询规划器对显式排序步骤的使用。不可能完全压制显式排序，但关闭此变量会使规划器在有其他方法可用时不去使用它。默认是打开的。这用于调试查询规划器。

`enable_tidscan` (`boolean`)

启用或禁用查询规划器对 TID 扫描计划类型的使用。默认是打开的。这用于调试查询规划器。

16.4.4.2. 规划器代价常量

注意

遗憾的是，没有定义良好的方法来确定下面这组“代价”变量的理想值。鼓励你进行实验并分享你的发现。

`effective_cache_size` (`floating point`)

设置规划器关于磁盘缓存有效大小的假设（即内核磁盘缓存中将被用于 PostgreSQL 数据文件的部分）。它以磁盘页为单位计量，每页通常为 8192 字节。默认值是 1000。

`random_page_cost` (`floating point`)

设置查询规划器对非顺序获取一个磁盘页的代价的估计。它以顺序获取一个页面的代价的倍数来计量。较高的值使顺序扫描更可能被使用，较低的值使索引扫描更可能被使用。默认值是四。

`cpu_tuple_cost` (`floating point`)

设置查询规划器对查询期间处理每一行的代价的估计。它以顺序获取一个页面的代价的一个比例来计量。默认值是 0.01。

`cpu_index_tuple_cost` (`floating point`)

设置查询规划器对索引扫描期间处理每个索引行的代价的估计。它以顺序获取一个页面的代价的一个比例来计量。默认值是 0.001。

`cpu_operator_cost` (`floating point`)

设置规划器对处理 `WHERE` 子句中每个操作符的代价的估计。它以顺序获取一个页面的代价的一个比例来计量。默认值是 0.0025。

16.4.4.3. 遗传查询优化器

`geqo` (`boolean`)

启用或禁用遗传查询优化，它是一种试图不做穷举搜索就完成查询规划的算法。默认是打开的。另见其他各种 `geqo_` 设置。

`geqo_threshold`(`integer`)

当查询涉及的 `FROM` 项至少达到此数量时，使用遗传查询优化进行规划。（注意，一个外层 `JOIN` 结构只算一个 `FROM` 项。）默认值是 11。对较简单的查询，通常最好使用确定性的穷举规划器，但对有很多表的查询，确定性规划器耗时太长。

`geqo_effort`(`integer`)
`geqo_generations`(`integer`)
`geqo_pool_size`(`integer`)
`geqo_selection_bias`(`floating point`)

遗传查询优化算法的各种调整参数：池大小（pool size）是一个种群中个体的数量。有效值在 128 到 1024 之间。如果设置为 0（默认值），则取池大小为 2^(QS+1)，其中 QS 是查询中 `FROM` 项的数量。effort 用于计算 generations 的默认值。有效值在 1 到 80 之间，默认为 40。generations 指定算法中迭代的次数。该数必须是正整数。如果指定为 0，则使用 `Effort * Log2(PoolSize)`。算法的运行时间大致与池大小和 generations 之和成正比。选择偏差（selection bias）是种群内的选择压力。取值可以从 1.50 到 2.00；后者是默认值。

16.4.4.4. 其他规划器选项

`default_statistics_target`(`integer`)

为没有通过 `ALTER TABLE SET STATISTICS` 设置列专属目标的表列设置默认统计目标。较大的值会增加 `ANALYZE` 所需的时间，但可能改进规划器估计的质量。默认值是 10。

`from_collapse_limit`(`integer`)

如果合并后得到的 FROM 列表包含的项不多于此数，规划器就会把子查询合并到上层查询中。较小的值减少规划时间，但可能得到较差的查询计划。默认值是 8。通常明智的做法是让它小于 `geqo_threshold`。

`join_collapse_limit`(`integer`)

每当得到的列表包含的项不多于此数时，规划器就会把显式的内层 `JOIN` 结构展平为 `FROM` 项列表。通常把它设置得与 `from_collapse_limit` 相同。把它设置为 1 可以阻止内层 `JOIN` 的任何展平，从而可以用显式 `JOIN` 语法来控制连接顺序。中间值可用于在规划时间与计划质量之间权衡。

16.4.5. 错误报告和日志

16.4.5.1. Syslog

`syslog`(`integer`)

PostgreSQL 允许使用 syslog 记录日志。如果此选项设置为 1，消息会同时发送到 syslog 和标准输出。设置为 2 则只把输出发送到 syslog。（有些消息仍会发送到标准输出/错误。）默认值是 0，表示 syslog 关闭。此选项必须在服务器启动时设置。

`syslog_facility`(`string`)

此选项决定启用通过 syslog 记录日志时所使用的 syslog “设施”。可以从 `LOCAL0`、`LOCAL1`、`LOCAL2`、`LOCAL3`、`LOCAL4`、`LOCAL5`、`LOCAL6`、`LOCAL7` 中选择；默认值是 `LOCAL0`。另见你的系统的 syslog 文档。

syslog_ident (string)

如果启用了记录到 syslog，此选项决定在 syslog 日志消息中用于标识 PostgreSQL 消息的程序名。默认值是 postgres。

16.4.5.2. 何时记录日志

client_min_messages (string)

控制发送到客户端的消息级别。有效值是 DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、LOG、NOTICE、WARNING 和 ERROR。每个级别包括排在它之后的全部级别。级别越靠后，发送的消息越少。默认值是 NOTICE。注意，LOG 在这里的级别排序与在 log_min_messages 中不同。

log_min_messages (string)

控制写入服务器日志的消息级别。有效值是 DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、ERROR、LOG、FATAL 和 PANIC。每个级别包括排在它之后的全部级别。级别越靠后，发送到日志的消息越少。默认值是 NOTICE。注意，LOG 在这里的级别排序与在 client_min_messages 中不同。只有超级用户能调高此选项。

log_error_verbosity (string)

控制服务器日志中为每条被记录消息写入的细节量。有效值是 TERSE、DEFAULT 和 VERBOSE，每个都比前一个向显示的消息中增加更多字段。

log_min_error_statement (string)

控制导致错误条件的 SQL 语句是否也记录到服务器日志中。所有导致指定级别或更高级别错误的 SQL 语句都会被记录。默认值是 PANIC（对正常使用而言实际上关闭了此特性）。有效值是 DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、ERROR、FATAL 和 PANIC。例如，如果把它设置为 ERROR，那么所有导致错误、致命错误或恐慌的 SQL 语句都将被记录。启用此选项有助于追踪服务器日志中出现的任何错误的来源。只有超级用户能调高此选项。

log_min_duration_statement (integer)

设置语句被记录的最短执行时间（以毫秒为单位）。所有运行时间达到或超过指定时间的 SQL 语句都将连同其时长被记录。设置为零会打印所有查询及其时长。负一（默认值）禁用此功能。例如，如果把它设置为 250，那么所有运行 250ms 或更长时间的 SQL 语句都将被记录。启用此选项有助于追踪应用程序中未经优化的查询。如果此选项是由管理员设置的，则只有超级用户能调高它或把它设置为负一。

silent_mode (boolean)

让服务器静默运行。如果设置了此选项，服务器将自动在后台运行，并脱离任何控制终端。这样，不会有任何消息写到标准输出或标准错误（效果与 postmaster 的 -S 选项相同）。除非启用了 syslog 日志，否则不建议使用此选项，因为它会使你无法看到错误消息。

下面是这些设置中使用的各种消息严重级别的列表：

DEBUG[1-5]

提供供开发者使用的信息。

INFO

提供用户隐式请求的信息，例如在 VACUUM VERBOSE 期间。

NOTICE

提供可能对用户有帮助的信息，例如长标识符被截断以及作为主键一部分创建索引。

WARNING

向用户提供警告，例如在事务块之外执行 COMMIT。

ERROR

报告导致当前事务中止的错误。

LOG

报告管理员可能关心的信息，例如检查点活动。

FATAL

报告导致当前会话中止的错误。

PANIC

报告导致所有会话中止的错误。

16.4.5.3. 记录什么内容

debug_print_parse (boolean)
debug_print_rewritten (boolean)
debug_print_plan (boolean)
debug_pretty_print (boolean)

这些选项启用向客户端或服务器日志发送各种调试输出。对每个执行的查询，它们会打印所得到的语法解析树、查询重写器输出或执行计划。debug_pretty_print 会对这些显示进行缩进，产生更可读但也长得多的输出格式。
要把输出发送到客户端或服务器日志，client_min_messages 或 log_min_messages 必须是 DEBUG1 或更低。这些选项默认是关闭的。

log_connections (boolean)

向服务器日志输出一行，详述每个成功的连接。默认是关闭的，尽管它可能非常有用。
此选项只能在服务器启动时或在 postgresql.conf 配置文件中设置。

log_duration (boolean)

使每条已完成语句的时长都被记录。要使用此选项，请启用 log_statement 和 log_pid，以便能用进程 ID 把语句和时长关联起来。默认是关闭的。如果此选项是由管理员启用的，则只有超级用户能把它关闭。

log_pid (boolean)

在服务器日志文件的每条消息前加上服务器进程的进程 ID。这有助于分清哪些消息属于哪个连接。默认是关闭的。此参数不影响通过 syslog 记录的消息，那些消息总是包含进程 ID。

`log_statement` (`boolean`)

使每条 SQL 语句都被记录。默认是关闭的。如果此选项是由管理员启用的，则只有超级用户能把它关闭。

`log_timestamp` (`boolean`)

在服务器日志的每条消息前加上时间戳。默认是关闭的。

`log_hostname` (`boolean`)

默认情况下，连接日志只显示连接主机 IP 地址。如果想让它显示主机名，可以打开此选项，但取决于你的主机名解析配置，这可能带来不可忽视的性能损失。
此选项只能在服务器启动时设置。

`log_source_port` (`boolean`)

在连接日志消息中显示连接主机的源端口号。你可以追溯端口号，
找出是哪个用户发起的连接。除此之外它没什么用处，因此默认是关闭的。
此选项只能在服务器启动时设置。

16.4.6. 运行时统计

16.4.6.1. 统计监控

`log_statement_stats` (`boolean`)
`log_parser_stats` (`boolean`)
`log_planner_stats` (`boolean`)
`log_executor_stats` (`boolean`)

对每个查询，把相应模块的性能统计写入服务器日志。这是一种粗略的性能剖析工具。
所有这些选项默认都是禁用的。如果这些选项是由管理员启用的，
则只有超级用户能把它们关闭。

16.4.6.2. 查询和索引统计收集器

`stats_start_collector` (`boolean`)

控制服务器是否应启动统计收集子进程。默认是打开的，
但如果你确定对收集统计信息没有兴趣，可以把它关闭。此选项只能在服务器启动时设置。

`stats_command_string` (`boolean`)

启用对每个会话当前正在执行的命令及其开始执行时间的统计收集。此选项默认是关闭的。
注意，即使启用了它，这些信息也不是对所有用户可见，
只有超级用户和被报告会话的属主才能看到；因此它不会构成安全风险。这些数据可以通过 `pg_stat_activity` 系统视图访问；更多信息参见第 23 章。

`stats_block_level` (`boolean`)
`stats_row_level` (`boolean`)

分别启用对数据库活动的块级和行级统计的收集。这些选项默认是关闭的。
这些数据可以通过 `pg_stat` 和 `pg_statio` 系列系统视图访问；更多信息参见第 23 章。

stats_reset_on_server_start (boolean)

如果打开，收集到的统计信息在服务器每次重启时清零。如果关闭，统计信息跨服务器重启累计。默认是打开的。此选项只能在服务器启动时设置。

16.4.7. 客户端连接默认值

16.4.7.1. 语句行为

search_path (string)

此变量指定当对象（表、数据类型、函数等）以不带模式部分的简单名称被引用时，搜索模式的顺序。当不同模式中存在同名的对象时，使用搜索路径中最先找到的那个。不在搜索路径任何模式中的对象，只能通过用限定（带点号）名称指定其所属模式来引用。

search_path 的值必须是一个逗号分隔的模式名列表。如果列表项之一是特殊值 $user，则替换为名为 SESSION_USER 返回值的模式（如果存在这样的模式）。（如果不存在，$user 被忽略。）

系统目录模式 pg_catalog 总是被搜索，无论路径中是否提到它。如果路径中提到了它，则按指定的顺序搜索。如果 pg_catalog 不在路径中，则会在搜索任何路径项之前搜索它。

同样，当前会话的临时表模式 pg_temp_*nnn*（如果存在）也总是被搜索。可以使用别名 pg_temp 把它显式列在路径中。如果它没有列在路径中，则会最先被搜索（甚至在 pg_catalog 之前）。不过，只有关系（表、视图、序列等）和数据类型名称才搜索临时模式，函数和操作符名称永远不会搜索它。

当创建对象而没有指定特定的目标模式时，对象会被放在搜索路径中列出的第一个模式中。如果搜索路径为空，则会报告错误。

此参数的默认值是 '$user, public'（其中第二部分在没有名为 public 的模式时被忽略）。这支持共享使用一个数据库（其中没有用户拥有私有模式，所有人共享使用 public）、每用户私有模式以及两者的组合。通过全局或逐用户地更改默认搜索路径设置，可以获得其他效果。

搜索路径的当前有效值可以通过 SQL 函数 current_schemas() 查看。这与查看 search_path 的值并不完全相同，因为 current_schemas() 显示的是 search_path 中出现的请求是如何被解析的。

关于模式处理的更多信息，见第 5.8 节。

check_function_bodies (boolean)

此参数通常为真。设置为假时，它会禁用对 CREATE FUNCTION 中函数体字符串的验证。在从转储恢复函数定义时，禁用验证偶尔有用，可以避免前向引用之类的问题。

default_transaction_isolation (string)

每个 SQL 事务都有一个隔离级别，可以是“读已提交”或“可串行化”。此参数控制每个新事务的默认隔离级别。默认是“读已提交”。

更多信息参见第 12 章和SET TRANSACTION。

default_transaction_read_only (boolean)

只读 SQL 事务不能修改非临时表。此参数控制每个新事务的默认只读状态。默认是假（读/写）。

更多信息参见SET TRANSACTION。

`statement_timeout` (`integer`)

中止任何耗时超过指定毫秒数的语句。值为零时关闭计时器，这也是默认值。

16.4.7.2. 区域和格式化

`datestyle` (`string`)

设置日期和时间值的显示格式，以及解释有歧义的日期输入值的规则。由于历史原因，此变量包含两个独立的组成部分：输出格式规范（`ISO`、`Postgres`、`SQL` 或 `German`）和日期字段顺序规范（`DMY`、`MDY` 或 `YMD`）。它们可以分别设置，也可以一起设置。关键字 `Euro` 和 `European` 是 `DMY` 的同义词；关键字 `US`、`NonEuro` 和 `NonEuropean` 是 `MDY` 的同义词。更多信息见第 8.5 节。默认值是 `ISO, MDY`。

`timezone` (`string`)

设置用于显示和解释时间戳的时区。默认使用系统环境指定的时区。更多信息见第 8.5 节。

`australian_timezones` (`boolean`)

如果设置为真，`ACST`、`CST`、`EST` 和 `SAT` 会被解释为澳大利亚时区，而不是北美/ 南美时区和星期六。默认是假。

`extra_float_digits` (`integer`)

此参数调整浮点值（包括 `float4`、`float8` 和几何数据类型）显示的位数。参数值被加到标准位数（`FLT_DIG` 或 `DBL_DIG`，视情况而定）上。该值最高可以设置为 2，以包含部分有效的数字；这在转储需要精确恢复的浮点数据时特别有用。也可以设置为负值来抑制不需要的数字。

`client_encoding` (`string`)

设置客户端编码（字符集）。默认使用数据库编码。

`lc_messages` (`string`)

设置显示消息所用的语言。可接受的值依赖于系统；更多信息见第 20.1 节。如果此变量设置为空字符串（默认值），则以依赖于系统的方式从服务器的执行环境继承该值。

在某些系统上，此区域设置类别不存在。设置此变量仍然有效，但不会有任何作用。另外，所需语言的已翻译消息也可能不存在。那种情况下你将继续看到英文消息。

`lc_monetary` (`string`)

设置用于格式化货币金额的区域设置，例如供 `to_char` 函数族使用。可接受的值依赖于系统；更多信息见第 20.1 节。如果此变量设置为空字符串（默认值），则以依赖于系统的方式从服务器的执行环境继承该值。

`lc_numeric` (`string`)

设置用于格式化数字的区域设置，例如供 `to_char()` 函数族使用。可接受的值依赖于系统；更多信息见第 20.1 节。如果此变量设置为空字符串（默认值），则以依赖于系统的方式从服务器的执行环境继承该值。

`lc_time` (`string`)

设置用于格式化日期和时间值的区域设置。（目前此设置不起作用，但将来可能会有。）可接受的值依赖于系统；更多信息见第 20.1 节。如果此变量设置为空字符串（默认值），则以依赖于系统的方式从服务器的执行环境继承该值。

16.4.7.3. 其他默认值

`explain_pretty_print` (`boolean`)

决定 `EXPLAIN VERBOSE` 显示详细的查询树转储时使用缩进格式还是非缩进格式。默认是打开的。

`dynamic_library_path` (`string`)

当需要打开一个动态可加载模块而指定的名称不含目录部分（即名称中不含斜杠）时，系统会在此路径中搜索指定的文件。（所用的名称是 `CREATE FUNCTION` 或 `LOAD` 命令中指定的名称。）

`dynamic_library_path` 的值必须是一个冒号分隔的绝对目录名列表。如果目录名以特殊值 `$libdir` 开头，则替换为编译时确定的 PostgreSQL 包库目录。PostgreSQL 发行版提供的模块就安装在那里。（使用 `pg_config --pkglibdir` 可以打印此目录的名称。）例如：

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/
lib:$libdir'
```

此参数的默认值是 `'$libdir'`。如果值设置为空字符串，则关闭自动路径搜索。

超级用户可以在运行时更改此参数，但以那种方式做的设置只会保持到客户端连接结束，因此这种方法应仅保留用于开发目的。设置此参数的推荐方式是在 `postgresql.conf` 配置文件中。

`max_expr_depth` (`integer`)

设置解析器的最大表达式嵌套深度。默认值 10000 对任何正常查询都足够高，但如有需要可以提高它。（但如果提得太高，就有因栈溢出而导致服务器崩溃的风险。）

16.4.8. 锁管理

`deadlock_timeout` (`integer`)

这是在等待锁多长时间（以毫秒计）之后才检查是否存在死锁条件。死锁检查相对较慢，所以服务器不会在每次等待锁时都运行它。我们（乐观地？）假设生产应用中死锁并不常见，先在锁上等待一会儿，然后才开始检查死锁。增大此值可以减少无谓的死锁检查所浪费的时间，但会减慢真实死锁错误的报告。默认值是 1000（即一秒），这大概是你实践中想要的最小值。在负载很重的服务器上，你可能需要提高它。理想情况下，该设置应当超过你的典型事务时间，以便提高在等待者决定检查死锁之前锁就被释放的机会。

`max_locks_per_transaction` (`integer`)

共享锁表的大小基于这样的假设：任意时刻最多有 `max_locks_per_transaction` * `max_connections` 个不同的对象需要加锁。默认值 64 历史上已被证明是足够的，但如果你的客

户端在单个事务中会接触许多不同的表，可能需要提高此值。
此选项只能在服务器启动时设置。

16.4.9. 版本和平台兼容性

16.4.9.1. 以前的 PostgreSQL 版本

`add_missing_from` (`boolean`)

当为 `true` 时，查询引用的表如果尚未出现在 `FROM` 子句中，会被自动加入其中。默认值是 `true`，以兼容以前的 PostgreSQL 版本。但此行为不符合 SQL 标准，许多人也不喜欢它，因为它可能掩盖错误。设置为 `false` 可获得 SQL 标准行为：拒绝引用未在 `FROM` 中列出的表。

`backslash_quote` (`string`)

此参数控制字符串字面量中是否可以用 `\'` 表示引号。表示引号的首选 SQL 标准方式是把它加倍（`''`），但 PostgreSQL 历史上也接受 `\'`。然而，使用 `\'` 会带来安全风险，因为在某些客户端字符集编码中，存在最后一个字节在数值上等于 ASCII `\` 的多字节字符。如果客户端代码的转义不正确，就可能发生 SQL 注入攻击。让服务器拒绝引号似乎被反斜杠转义的查询，可以防止这种风险。`backslash_quote` 的允许值是 `on`（总是允许 `\'`）、`off`（总是拒绝）和 `safe_encoding`（仅当客户端编码不允许在多字节字符中出现 ASCII `\` 时才允许）。默认设置是 `safe_encoding`。

`regex_flavor` (`string`)

正则表达式的“风格”可以设置为 `advanced`、`extended` 或 `basic`。默认是 `advanced`。`extended` 设置可用于与 7.4 之前的 PostgreSQL 版本精确向后兼容。

`sql_inheritance` (`boolean`)

此参数控制继承语义，特别是各种命令是否默认包含子表。7.1 之前的版本不包含子表。如果你需要旧行为，可以把此变量设置为关闭，但长远来看，建议你修改应用程序，用 `ONLY` 关键字排除子表。关于继承的更多信息见第 5.5 节。

16.4.9.2. 平台和客户端兼容性

`transform_null_equals` (`boolean`)

打开时，形如 *`expr`* `= NULL`（或 `NULL =` *`expr`*）的表达式会被当作 *`expr`* `IS NULL` 处理，即当 *`expr`* 求值为空值时返回真，否则返回假。*`expr`* `= NULL` 的正确行为是总是返回空值（未知）。因此此选项默认是关闭的。

不过，Microsoft Access 中的筛选形式生成的查询看起来是用 *`expr`* `= NULL` 来测试空值，因此如果你用该界面访问数据库，可能需要打开此选项。由于形如 *`expr`* `= NULL` 的表达式（按正确解释）总是返回空值，它们不是很有用，在正常应用中也不常出现，所以此选项实际上没什么害处。但新用户经常对涉及空值的表达式的语义感到困惑，
所以此选项默认不打开。

注意，此选项只影响字面上的 `=` 操作符，不影响其他比较操作符，也不影响在计算上等价于某个涉及等号操作符表达的其他表达式（例如 `IN`）。因此，此选项不是对不良编程的通用修复。

相关信息参见第 9.2 节。

16.4.10. 开发者选项

以下选项用于 PostgreSQL 源代码方面的工作，某些情况下也用于协助恢复严重损坏的数据库。在生产数据库环境中没有理由使用它们。因此，它们没有被列入示例 postgresql.conf 文件。注意，其中许多选项需要特殊的源码编译标志才能工作。

debug_assertions (boolean)

打开各种断言检查。这是一种调试辅助。如果你遇到奇怪的问题或崩溃，可能需要打开它，因为它可能暴露编程错误。要使用此选项，构建 PostgreSQL 时必须定义宏 USE_ASSERT_CHECKING（通过 configure 选项 --enable-cassert 完成）。注意，如果 PostgreSQL 构建时启用了断言，DEBUG_ASSERTIONS 默认是打开的。

pre_auth_delay (integer)

如果非零，新服务器进程被 fork 之后、进行认证过程之前，会发生这么多秒的延迟。这是为了给用调试器附到服务器进程上、追踪认证中的错误行为提供机会。

trace_notify (boolean)

为 LISTEN 和 NOTIFY 命令生成大量调试输出。要分别把此输出发送到客户端或服务器日志，client_min_messages 或 log_min_messages 必须是 DEBUG1 或更低。

trace_locks (boolean)
trace_lwlocks (boolean)
trace_userlocks (boolean)
trace_lock_oidmin (boolean)
trace_lock_table (boolean)
debug_deadlocks (boolean)
log_btree_build_stats (boolean)

各种其他代码跟踪和调试选项。

wal_debug (integer)

如果非零，打开与 WAL 相关的调试输出。

zero_damaged_pages (boolean)

检测到损坏的页头通常会使 PostgreSQL 报告错误并中止当前事务。把 zero_damaged_pages 设置为真会使系统改为报告警告，将损坏页清零并继续处理。此行为会破坏数据，即损坏页上的所有行。但它允许你跳过该错误，从表中可能存在的未损坏页中取回行。因此，当因硬件或软件错误发生损坏时，它对恢复数据很有用。通常，在对从表的损坏页恢复数据不抱希望之前，不应把它设置为真。默认设置是关闭的，且只有超级用户能更改它。

16.4.11. 短选项

为方便起见，有些参数还有单字母的命令行选项开关。它们在表 16.1中描述。

表 16.1. 短选项对照

短选项	等价形式
-B *x*	shared_buffers = *x*
-d *x*	log_min_messages = DEBUG*x*

短选项	等价形式
-F	fsync = off
-h *x*	virtual_host = *x*
-i	tcpip_socket = on
-k *x*	unix_socket_directory = *x*
-l	ssl = on
-N *x*	max_connections = *x*
-p *x*	port = *x*
-fi, -fh, -fm, -fn, -fs, -ft[a]	enable_indexscan=off, enable_hashjoin=off, enable_mergejoin=off, enable_nestloop=off, enable_seqscan=off, enable_tidscan=off
-s [a]	log_statement_stats = on
-S *x* [a]	sort_mem = *x*
-tpa, -tpl, -te[a]	log_parser_stats=on, log_planner_stats=on, log_executor_stats=on

[a] 由于历史原因，这些选项必须通过 postmaster 的 -o 选项传给单个服务器进程，例如：

```
$ postmaster -o '-S 1024 -s'
```

或者从客户端一侧通过 PGOPTIONS 传递，如上文所述。

16.5. 管理内核资源

一个大型PostgreSQL安装会很快耗尽各种操作系统资源限制。（在某些系统上，出厂默认值非常低，你甚至不需要真正“大型”的安装就会遇到问题。）如果你遇到过这类问题，请继续阅读。

16.5.1. 共享内存和信号量

共享内存和信号量统称为“System V IPC”（连同消息队列，但消息队列与 PostgreSQL无关）。几乎所有现代操作系统都提供这些特性，但并非所有系统都默认启用了它们或将其设置得足够大，尤其是有 BSD 血统的系统。（对于 QNX 和 BeOS 移植，PostgreSQL 为这些设施提供了自己的替代实现。）

完全缺少这些功能时，通常会在服务器启动时出现 Illegal system call 错误。在这种情况下，除了重新配置内核别无选择。没有它们，PostgreSQL 无法工作。

当PostgreSQL超出各种硬性IPC限制之一时，服务器会拒绝启动，并应留下带有指导性的错误消息，说明所遇到的问题以及应如何处理（另见第 16.3.1 节）。相关内核参数在不同系统上的命名基本一致，表 16.2给出了概览。不过，设置它们的方法却各不相同。下面给出一些平台上的建议。请注意，更改这些设置通常需要重启机器，甚至可能需要重新编译内核。

表 16.2. System V IPC参数

名称	描述	合理的值
SHMMAX	共享内存段的最大尺寸（字节）	250 kB + 8.2 kB * shared_buffers + 14.2 kB * max_connections，直至无穷大
SHMMIN	共享内存段的最小尺寸（字节）	1

名称	描述	合理的值
SHMALL	可用共享内存的总量（字节或页面）	如果是字节，同SHMMAX；如果是页面，为ceil(SHMMAX/PAGE_SIZE)
SHMSEG	每个进程的最大共享内存段数目	只需要 1 段，但是默认值高很多
SHMMNI	系统范围内的最大共享内存段数目	像SHMSEG外加其他应用的空间
SEMMNI	信号量标识符（即，集合）的最大数目	至少 ceil(max_connections / 16)
SEMMNS	系统范围内的最大信号量数目	ceil(max_connections / 16) * 17，并为其他应用保留空间
SEMMSL	每个集合中信号量的最大数目	至少 17
SEMMAP	信号量映射中的项数	见文本
SEMVMX	信号量的最大值	至少 1000（默认值常常是 32767，如非必要不要更改）

最重要的共享内存参数是 SHMMAX，即以字节计的共享内存段的最大尺寸。如果 shmget返回像Invalid argument这样的错误消息，可能是超出了这个限制。所需共享内存段的尺寸随请求的缓冲区数量（-B 选项）和允许的连接数量（-N 选项）而变化，尽管前者的影响最大。（作为临时解决办法，你可以调低这些设置来消除失败。）粗略估计，
你可以用缓冲区数量乘以块大小（默认 8 kB）再加上充足的额外开销（至少半兆字节）来估算所需的段尺寸。你可能得到的任何错误消息都会包含失败分配请求的尺寸。

不太可能出问题的是共享内存段的最小尺寸（SHMMIN），对PostgreSQL来说应该最多大约是 256 kB（通常只是 1）。系统范围（SHMMNI）或每进程（SHMSEG）的最大共享内存段数目不会导致问题，除非你的系统把它们设成零。有些系统还对系统内共享内存的总量有限制；
参见下面的平台特定说明。

PostgreSQL为每个允许的连接（-N 选项）使用一个信号量，每 16 个组成一组。每组还包含第 17 个信号量，其中存放一个“魔数”，用来检测与其他应用程序使用的信号量集的冲突。系统中的信号量最大数量由SEMMNS设置，因此它必须至少等于max_connections再加上每 16 个允许连接一个的额外信号量（见表 16.2中的公式）。参数SEMMNI决定系统中同时可以存在的信号量集的数量上限，因此它必须至少为ceil(max_connections / 16)。降低允许的连接数是失败时的一种临时变通办法，这种失败通常表现为 semget函数返回措辞令人困惑的No space left on device。

在某些情况下，可能还需要增大SEMMAP，使其至少与 SEMMNS处于同一数量级。这个参数定义信号量资源映射的大小，映射中每个连续的可用信号量块都需要占用一项。
每当一个信号量集合被释放时，它要么会并入与该释放块相邻的现有项，
要么会登记为一个新的映射项。如果映射已满，被释放的信号量就会丢失（直到重启）。
信号量空间的碎片化随时间推移可能导致可用信号量少于应有数量。

SEMMSL参数决定一个信号量集中可以包含多少个信号量，对于PostgreSQL，它必须至少为 17。

与“信号量撤销”有关的其他各种设置，如SEMMNU 和SEMUME，对PostgreSQL无关紧要。

BSD/OS

共享内存. 默认只支持 4 MB 共享内存。请记住，共享内存是不可换页的，它被锁定在 RAM 中。要增加系统支持的共享内存量，可以在内核配置文件中加入下面的内容。SHMALL 值 1024 表示 4 MB 共享内存。以下设置把最大共享内存区域增加到 32 MB：

```
options "SHMALL=8192"
```

```
options "SHMMAX=\(SHMALL*PAGE_SIZE\)"
```

对于运行 4.3 或更高版本的用户，可能还需要把 KERNEL_VIRTUAL_MB增加到默认值248 之上。所有修改完成后，重新编译内核并重启。

对于运行 4.0 及更早版本的用户，可用bpatch在当前内核中找到sysptsize值。该值是在引导时动态计算的。

```
$ bpatch -r sysptsize
0x9 = 9
```

然后，在内核配置文件中把SYSPTSIZE加为硬编码值。把你用bpatch找到的值增大。每想要 4 MB 额外共享内存就加 1。

```
options "SYSPTSIZE=16"
```

sysptsize无法通过sysctl更改。

信号量. 你可能需要增加信号量的数量。默认情况下，PostgreSQL分配 34 个信号量，这超过了系统默认总数 60 的一半。请在内核配置文件中设置你想要的值，例如：

```
options "SEMMNI=40"
options "SEMMNS=240"
```

FreeBSD
NetBSD
OpenBSD

编译内核时需要启用选项SYSVSHM和SYSVSEM（默认已启用）。共享内存的最大尺寸由选项SHMMAXPGS（以页计）决定。下面展示了一个如何设置各参数的例子：

```
options         SYSVSHM
options         SHMMAXPGS=4096
options         SHMSEG=256

options         SYSVSEM
options         SEMMNI=256
options         SEMMNS=512
options         SEMMNU=256
options         SEMMAP=256
```

（在NetBSD和 OpenBSD上，关键字实际上是单数形式的option。）

你可能还希望配置内核，把共享内存锁定在 RAM 中，防止其被换出到交换区。这可以通过 sysctl 设置 kern.ipc.shm_use_phys 来实现。

HP-UX

默认设置通常足以满足普通安装的需求。在 HP-UX 10 上，SEMMNS 的出厂默认值为 128，对较大的数据库站点来说可能过低。

可以在 System Administration Manager（SAM）的 Kernel Configuration→Configurable Parameters 中设置 IPC 参数。完成后选择 Create A New Kernel。

Linux

在 2.2 内核中，默认的共享内存限制（SHMMAX和SHMALL都是）是 32 MB，但可以在 proc 文件系统中更改（无需重启）。例如，要允许 128 MB：

$echo 134217728 >/proc/sys/kernel/shmall$echo 134217728 >/proc/sys/kernel/shmmax

你可以把这些命令放进引导时运行的脚本中。

另外，如果可用的话，也可以用sysctl控制这些参数。找到名为/etc/sysctl.conf的文件，向其中加入如下行：

```
kernel.shmall = 134217728
kernel.shmmax = 134217728
```

此文件通常在引导时被处理，但之后也可以显式调用 sysctl。

其他参数对任何应用来说尺寸都已足够。如果你想亲自查看，可以看 /usr/src/linux/include/asm-*xxx*/shmpara m.h和/usr/src/linux/include/linux/sem.h。

MacOS X

在 OS X 10.2 及更早版本中，编辑文件 /System/Library/StartupItems/SystemTuning/SystemTuning 并更改以下命令中的值：

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

在 OS X 10.3 中，这些命令已被移到/etc/rc中，必须在那里编辑。

SCO OpenServer

在默认配置中，每个段只允许 512 kB 共享内存，大约只够 -B 24 -N 12使用。要提高此设置，首先切换到目录 /etc/conf/cf.d。要显示SHMMAX的当前值，运行

```
./configure -y SHMMAX
```

要为SHMMAX设置新值，运行

```
./configure SHMMAX=value
```

其中*value*是你要使用的新值（以字节计）。设置 SHMMAX之后，重建内核：

```
./link_unix
```

然后重启。

Solaris

至少在 2.6 版中，共享内存段的默认最大尺寸对 PostgreSQL来说太低。相关设置可以在 /etc/system中更改，例如：

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
```

```
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

需要重启才能使更改生效。

关于 Solaris 下共享内存的信息，另见http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html。

UnixWare

在 UnixWare 7 上，默认配置中共享内存段的最大尺寸是 512 kB。这大约够 `-B 24 -N 12`使用。要显示 `SHMMAX`的当前值，运行

```
/etc/conf/bin/idtune -g SHMMAX
```

它会显示当前值、默认值、最小值和最大值。要为 `SHMMAX`设置新值，运行

```
/etc/conf/bin/idtune SHMMAX value
```

其中*`value`*是你要使用的新值（以字节计）。设置 `SHMMAX`之后，重建内核：

```
/etc/conf/bin/idbuild -B
```

然后重启。

16.5.2. 资源限制

类 Unix 操作系统强制执行多种资源限制，它们可能干扰 PostgreSQL 服务器的运行。特别重要的是每用户进程数、每进程打开文件数以及每个进程可用内存量的限制。这些限制都各有“硬”限制和“软”限制。实际生效的是软限制，但用户可以把它提高到硬限制为止。硬限制只能由 root 用户更改。系统调用`setrlimit`负责设置这些参数。从命令行控制资源限制使用 shell 内建命令`ulimit`（Bourne shell）或`limit`（csh）。在 BSD 衍生的系统上，文件`/etc/login.conf`控制登录时设置的各种资源限制。细节参见操作系统文档。相关参数是 `maxproc`、`openfiles`和 `datasize`。例如：

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

（`-cur`是软限制。加`-max`后缀可设置硬限制。）

内核对某些资源也可能有系统范围的限制。

- 在 Linux 上，`/proc/sys/fs/file-max`决定内核支持的最大打开文件数。可以通过向该文件写入不同的数字或在 `/etc/sysctl.conf`中加入赋值来更改它。每进程的最大文件数上限在内核编译时固定；更多信息见 `/usr/src/linux/Documentation/proc.txt`。

PostgreSQL服务器每个连接使用一个进程，因此除了系统其余部分所需的进程外，你应当为允许的连接数准备至少同样多的进程。这通常不是问题，但如果在一台机器上运行多个服务器，就可能变得紧张。

打开文件数的出厂默认限制通常被设置为“兼顾大家”的值，允许多个用户共存于一台机器而不占用过多的系统资源。如果你在一台机器上运行许多服务器，这可能正合你意，但在专用服务器上，你可能希望提高此限制。

另一方面，有些系统允许单个进程打开大量文件；如果不止几个进程这样做，
系统范围的限制就很容易被超出。如果你发现这种情况发生，而又不想更改系统范围的限制，
可以设置PostgreSQL 的max_files_per_process配置参数来限制打开文件的消耗。

16.5.3. Linux 内存过量分配

在 Linux 2.4 及更高版本中，默认的虚拟内存行为对 PostgreSQL来说不是最优的。由于内核实现内存过量分配的方式，如果其他进程的内存需求导致系统虚拟内存耗尽，内核可能终止PostgreSQL服务器（即 postmaster进程）。

如果发生这种情况，
你会看到类似这样的内核消息（该去哪里找这种消息请查阅你的系统文档和配置）：

```
Out of Memory: Killed process 12345 (postmaster).
```

这表明postmaster进程因内存压力而被终止。虽然现有的数据库连接会继续正常工作，
但不会接受新连接。要恢复，需要重启PostgreSQL。

避免此问题的一种方法是把PostgreSQL 运行在这样一台机器上：你能确保其他进程不会把机器的内存耗尽。

在 Linux 2.6 及更高版本中，更好的解决方案是修改内核的行为，使其不“过量分配”内存。这可以通过 sysctl 选择严格的过量分配模式来完成：

```
sysctl -w vm.overcommit_memory=2
```

或者在/etc/sysctl.conf中加入等价的条目。你可能还想修改相关的设置vm.overcommit_ratio。细节见内核文档文件 Documentation/vm/overcommit-accounting。

据报告，一些厂商的 Linux 2.4 内核带有 2.6 过量分配 sysctl 的早期版本。然而，在没有相关代码的内核上把vm.overcommit_memory 设置为 2 会使情况变得更糟而不是更好。建议你在 2.4 安装上尝试此操作之前，检查实际的内核源代码（见文件mm/mmap.c中的函数vm_enough_memory）来验证你的副本支持什么。存在 overcommit-accounting文档文件并不能作为该特性存在的证据。如有疑问，请咨询内核专家或你的内核供应商。

16.6. 关闭服务器

关闭数据库服务器有几种方式。通过向postmaster进程发送不同的信号来控制关闭的类型。

SIGTERM

收到 SIGTERM 后，服务器不再接受新连接，但让现有会话正常结束其工作。只有在所有会话都正常终止之后，它才会关闭。这是智能关闭。

SIGINT

服务器不再接受新连接，并向所有现有的服务器进程发送 SIGTERM，使它们中止当前事务并迅速退出。然后它等待服务器进程退出，最终关闭。这是快速关闭。

SIGQUIT

这是立即关闭，它会使postmaster 进程向所有子进程发送 SIGQUIT 并立即退出（不做自身的妥善关闭）。子进程收到 SIGQUIT 后同样立即退出。这会导致下次启动时进行恢复（通过重放 WAL 日志）。建议只在紧急情况下使用。

> **重要**
>
> 最好不要用 SIGKILL 来关闭服务器。这会阻止服务器释放共享内存和信号量，之后可能必须手动完成。

postmaster 进程的 PID 可以用 ps 程序找到，也可以从数据目录中的 postmaster.pid 文件得到。例如，要做快速关闭：

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

pg_ctl 程序是一个 shell 脚本，为关闭服务器提供了更方便的接口。

16.7. 使用 SSL 的安全 TCP/IP 连接

PostgreSQL 原生支持使用 SSL 连接来加密客户端/服务器通信，以提高安全性。这要求客户端和服务器系统上都安装了 OpenSSL，并且构建时启用了 PostgreSQL 中的相应支持（见第 14 章）。

编译了 SSL 支持后，在 postgresql.conf 中把参数 ssl 设置为 on，即可启动启用了 SSL 的 PostgreSQL 服务器。以 SSL 模式启动时，服务器会在数据目录中寻找 server.key 和 server.crt 文件，它们分别应包含服务器私钥和证书。这些文件必须正确设置，启用了 SSL 的服务器才能启动。如果私钥受口令保护，服务器会提示输入口令，输入之前不会启动。

服务器会在同一个 TCP 端口上同时监听标准连接和 SSL 连接，并与每个连接的客户端协商是否使用 SSL。关于如何强制服务器对某些连接要求使用 SSL，见第 19 章。

关于如何创建服务器私钥和证书的细节，请参考 OpenSSL 文档。可以使用简单的自签名证书来开始测试，但在生产环境中应使用由证书颁发机构（CA）（全球性 CA 或本地 CA 之一）签名的证书，以便客户端能验证服务器的身份。要快速创建一个自签名证书，使用以下 OpenSSL 命令：

```
openssl req -new -text -out server.req
```

填写 openssl 询问的信息。确保把本地主机名输入为“Common Name”；challenge password 可以留空。该程序会生成一个受口令保护的密钥；它不接受少于四个字符的口令。要去除口令（如果想让服务器自动启动就必须去除），运行以下命令

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

输入旧口令以解锁现有密钥。然后执行

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
chmod og-rwx server.key
```

把证书转换为自签名证书，并把密钥和证书复制到服务器将要寻找它们的位置。

16.8. 使用 SSH 隧道的安全 TCP/IP 连接

可以用 SSH 加密客户端与 PostgreSQL 服务器之间的网络连接。做得正确的话，这能提供足够安全的网络连接。

首先确保 SSH 服务器在 PostgreSQL 服务器所在的同一台机器上正常运行，并且你能用 ssh 以某个用户登录。然后，你可以从客户端机器上用这样的命令建立安全隧道：

```
ssh -L 3333:foo.com:5432 joe@foo.com
```

`-L` 参数中的第一个数字 `3333` 是隧道你这一端的端口号，可以自由选择。第二个数字 5432 是隧道的远端：你的服务器正在使用的端口号。两个端口号之间的名称或地址是你要连接的数据库服务器所在的主机。要通过此隧道连接数据库服务器，你连接本机的 3333 端口：

```
psql -h localhost -p 3333 template1
```

对数据库服务器来说，就像是你是用户 `joe@foo.com` 一样，服务器会使用为该用户设置的任何认证过程。为使隧道设置成功，你必须被允许像尝试用 `ssh` 建立终端会话那样，以 `joe@foo.com` 身份通过 `ssh` 连接。

提示

还有其他一些应用程序可以用与刚才描述的概念类似的步骤提供安全隧道。

第 17 章 数据库用户和权限

每个数据库集簇都包含一组数据库用户。
这些用户与运行服务器的操作系统所管理的用户是相互独立的。
用户拥有数据库对象（例如表），并且可以把这些对象上的权限赋予其他用户，
以控制谁能访问哪些对象。

本章描述如何创建和管理用户，并介绍权限系统。
关于各种数据库对象类型和权限效果的更多信息，见第 5 章。

17.1. 数据库用户

数据库用户在概念上与操作系统用户完全分离。实践中维持对应关系可能方便，
但这不是必需的。
数据库用户名在整个数据库集簇安装中是全局的（而不是每个数据库单独的）。要创建用户，
使用 `CREATE USER` SQL 命令：

```
CREATE USER name;
```

name 遵循 SQL 标识符的规则：要么是不带特殊字符的普通形式，要么用双引号括起。要移除现有的用户，使用类似的 `DROP USER` 命令：

```
DROP USER name;
```

为方便起见，提供了 `createuser` 和 `dropuser` 程序作为这些 SQL 命令的包装，可以从 shell 命令行调用：

```
createuser name
dropuser name
```

为了完成数据库系统的初始引导，一个刚初始化的系统总会包含一个预定义用户。
该用户将拥有固定的 ID 1，并且默认情况下（除非在运行`initdb`时另行指定）它与初始化数据库集簇的操作系统用户同名。这个用户通常命名为`postgres`。要创建更多用户，
必须先以这个初始用户建立连接。

每个到数据库服务器的连接中恰好只有一个用户身份处于活动状态。
某个数据库连接要使用的用户名，由发起连接请求的客户端以应用特定的方式指明。例如，`psql` 程序使用`-U`命令行选项来指明要以哪个用户连接。
很多应用默认假定使用当前操作系统用户的名字（包括`createuser`和`psql`）。因此，
在这两组用户之间维护命名对应关系通常比较方便。

给定的客户端连接可以以哪些数据库用户身份连接，由客户端认证设置决定，如第 19 章所述。
（因此，客户端不一定只能以与其操作系统用户同名的用户连接，
就像一个人的登录名不受其真实姓名约束一样。）
由于用户身份决定了已连接客户端可用的权限集，在建立多用户环境时仔细配置这一点很重要。

17.2. 用户属性

数据库用户可以具有一些定义其权限并与客户端认证系统交互的属性。

超级用户

数据库超级用户绕过所有权限检查。此外，只有超级用户才能创建新用户。
要创建数据库超级用户，使用`CREATE USER name CREATEUSER`。

创建数据库

必须显式授予用户创建数据库的权限（超级用户除外，因为超级用户绕过所有权限检查）。要创建这样的用户，使用`CREATE USER name CREATEDB`。

密码

只有当客户端认证方法要求用户在连接数据库时提供密码时，密码才有意义。`password`、`md5`和`crypt`认证方法使用密码。数据库密码与操作系统密码是分开的。
在创建用户时用`CREATE USER name PASSWORD 'string'`指定密码。

A user's attributes can be modified after creation with `ALTER USER`. See the reference pages for `CREATE USER` and `ALTER USER` for details.

用户还可以为第 16.4 节中描述的许多运行时配置设置设置个人默认值。例如，如果出于某种原因你想在每次连接时禁用索引扫描（提示：这不是好主意），可以使用

```
ALTER USER myname SET enable_indexscan TO off;
```

这会保存该设置（但不立即设置它），在后续连接中，效果就如同在会话开始之前刚调用过 `SET enable_indexscan TO off;` 一样。在会话期间你仍然可以更改这个设置；保存的只是默认值。要撤销这类设置，使用 `ALTER USER username RESET varname;`。

17.3. 组

与 Unix 中一样，组是一种对用户进行逻辑分组以便于管理权限的方法：权限可以授予组或从组收回（把组当作一个整体）。要创建组，使用

```
CREATE GROUP name;
```

要向组中添加用户或从组中删除用户，使用

```
ALTER GROUP name ADD USER uname1, ... ;
ALTER GROUP name DROP USER uname1, ... ;
```

17.4. 权限

创建数据库对象时，它会被指派一个所有者。所有者是执行创建语句的用户。要更改表、索引、序列或视图的所有者，请使用 `ALTER TABLE` 命令。默认情况下，只有所有者（或超级用户）才能对对象做任何操作。为了让其他用户能使用它，必须授予权限。

有多种不同的权限：`SELECT`，`INSERT`，`UPDATE`，`DELETE`，`RULE`，`REFERENCES`，`TRIGGER`，`CREATE`，`TEMPORARY`，`EXECUTE`，`USAGE` 和 `ALL PRIVILEGES`。关于 PostgreSQL 支持的各种权限的更多信息，见GRANT参考页。修改或销毁对象的权利永远只属于所有者。要授予权限，使用 `GRANT` 命令。因此，如果 `joe` 是一个现有用户，`accounts` 是一个现有的表，更新表的权限可以用下面的命令授予：

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

执行这条命令的用户必须是表的所有者。要向组授予权限，使用

```
GRANT SELECT ON accounts TO GROUP staff;
```

特殊“用户”名 `PUBLIC` 可用来向系统中的每个用户授予权限。用 `ALL` 代替具体的权限就表示授予所有权限。

要收回权限，使用名字同样贴切的 REVOKE 命令：

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

表拥有者的特殊权限（即执行 DROP、GRANT、REVOKE 等的权利）总是隐含在拥有者身份中，不能被授予或收回。但表拥有者可以选择收回自己的普通权限，
例如让自己和别人都只能对表进行只读访问。

17.5. 函数和触发器

函数和触发器允许用户把代码插入到后端服务器中，其他用户可能在不知情的情况下执行它。
因此，这两种机制都允许用户相对不受惩罚地“特洛伊木马”他人。
唯一真正的保护是严格控制谁可以定义函数。

用 SQL 之外的语言编写的函数在后端服务器进程内运行，拥有数据库服务器守护进程的操作系统权限。从受信任函数内部就有可能更改服务器的内部数据结构。因此，除了许多其他事情之外，这样的函数可以绕过任何系统访问控制。这是用户定义 C 函数的固有问题。

第 18 章 管理数据库

每个正在运行的PostgreSQL服务器实例都管理着一个或多个数据库。因此，数据库是组织SQL对象（“数据库对象”）的最高层级。本章描述数据库的属性，以及如何创建、管理和删除数据库。

18.1. 概述

数据库是一个命名的SQL对象（“数据库对象”）集合。通常，每个数据库对象（表、函数等）都属于且只属于一个数据库。（不过也有少量系统目录，例如`pg_database`，属于整个集簇，可以从集簇中的每个数据库访问。）更准确地说，数据库是模式的集合，而模式中包含表、函数等。因此完整的层次结构是：服务器、数据库、模式、表（或其他类型的对象，如函数）。

当连接到数据库服务器时，客户端必须在连接请求中指定要连接的数据库名。一个连接无法访问多个数据库。不过，一个应用可以不受限制地打开到同一数据库或其他数据库的多个连接。但是，也可以通过同一个连接访问多个模式。数据库在物理上是相互隔离的，访问控制在连接级别管理。因此，如果一个PostgreSQL服务器实例要容纳彼此应当分离、并且在大多数情况下互不知晓的项目或用户，建议把它们放到不同的数据库中。如果项目或用户是相互关联的，并且应该能够使用彼此的资源，则应当把它们放在同一个数据库中，但可能使用不同的模式。模式是纯逻辑结构，谁能访问什么由权限系统管理。更多关于管理模式的信息见第 5.8 节。

> **注意**
>
> SQL 把数据库称为“目录”，但在实践中没有区别。

18.2. 创建数据库

要创建数据库，PostgreSQL服务器必须已经启动并正在运行（见第 16.3 节）。

数据库使用 SQL 命令 `CREATE DATABASE` 创建：

```
CREATE DATABASE name;
```

其中*name*遵循SQL标识符的通常规则。当前用户自动成为新数据库的所有者。数据库的所有者有权在以后删除它（这也会删除其中的所有对象，即使它们有不同的所有者）。

创建数据库是一项受限操作。关于如何授予该权限，见第 17.2 节。

由于执行`CREATE DATABASE`命令时必须先连接到数据库服务器，因此还会有一个问题：某个站点上的第一个数据库是如何创建的？第一个数据库总是由`initdb`命令在初始化数据存储区时创建的（见第 16.2 节）。这个数据库叫做 `template1`。因此，要创建第一个“真正”的数据库，你可以连接到 `template1`。

`template1` 这个名字并非偶然：每当创建一个新数据库时，模板数据库都会被实质性地克隆。这意味着你在 `template1` 中所做的任何更改，都会传播到后来创建的所有数据库中。这暗示你不应把模板数据库用于实际工作，但如果使用得当，这一特性可以带来便利。更多细节见第 18.3 节。

为了方便，还可以从 shell 中执行一个程序来创建新数据库，即`createdb`。

```
createdb dbname
```

`createdb`并没有什么特殊之处。它连接到 `template1` 数据库并发出`CREATE DATABASE`命令，和上面描述的完全一样。`createdb` 的参考页包含调用细节。注意，不带任何参数的`createdb`会创建一个与当前用户名同名的数据库。

> **注意**
>
> 第 19 章包含关于如何限制谁能连接到给定数据库的信息。

有时你会想为其他人创建数据库，并让他们成为新数据库的拥有者，以便他们自行配置和管理它。要实现这一点，可使用下列命令之一：

```
CREATE DATABASE dbname OWNER username;
```

在 SQL 环境中使用，或者

```
createdb -O username dbname
```

只有超级用户才允许为其他人创建数据库。

18.3. 模板数据库

`CREATE DATABASE`实际上是通过复制一个现有数据库来创建新数据库。默认情况下，它复制名为 `template1` 的标准系统数据库。因此，该数据库就是创建新数据库时使用的“模板”。如果你向 `template1` 中添加对象，这些对象也会被复制到以后创建的用户数据库中。这种行为允许对数据库中的标准对象集合进行站点本地修改。例如，如果你将过程语言 PL/pgSQL 安装到 `template1` 中，那么以后创建用户数据库时无需额外操作，它就会自动可用。

还有第二个标准系统数据库，名为`template0`。该数据库包含与`template1`初始内容相同的数据，也就是说，只包含你的PostgreSQL版本预定义的标准对象。在 `initdb` 之后绝不应当更改 `template0`。通过指示`CREATE DATABASE`复制`template0`而不是`template1`，你可以创建一个“纯净的”用户数据库，它不包含`template1`中的任何站点本地附加对象。这在恢复`pg_dump`转储时特别方便：转储脚本应当恢复到一个纯净的数据库中，以确保重建出被转储数据库的正确内容，而不会与现在可能存在于 `template1` 中的附加对象发生冲突。

要创建一个通过复制 `template0` 的数据库，可以使用

```
CREATE DATABASE dbname TEMPLATE template0;
```

（在 SQL 环境中），或者

```
createdb -T template0 dbname
```

可以创建额外的模板数据库，实际上也可以通过将集簇中任意数据库的名称指定为 `CREATE DATABASE` 的模板来复制它。但重要的是要理解，这（至少目前）并不是一个通用的“`COPY DATABASE`”功能。特别地，在复制操作期间，源数据库必须是空闲的（没有正在进行的更改数据的事务），这一点至关重要。`CREATE DATABASE` 会检查操作开始时没有其他会话（它自己除外）连接到源数据库，但这并不能保证复

制进行期间不会发生更改，那样会导致复制出的数据库不一致。因此，
我们建议把用作模板的数据库当作只读的。

对每个数据库，pg_database中都有两个有用的标志列：datistemplate
和datallowconn。datistemplate可设置为表明某个数据库将作为CREATE DATABASE的模板。如果设置了该标志，任何拥有CREATEDB权限的用户都可以克隆该数据库；如果未设置，
则只有超级用户和数据库拥有者才能克隆它。如果datallowconn为假，
则不允许建立到该数据库的新连接（但仅仅把该标志设为假并不会终止已有会话）。
为了防止被修改，template0数据库通常被标记为datallowconn = false。template0
和template1始终都应标记为datistemplate = true。

准备好模板数据库或对其做任何更改之后，最好在该数据库中执行一次 VACUUM FREEZE 或 VACUUM FULL FREEZE。如果在同一数据库中没有其他打开的事务时执行，就可以保证该数据库中的所有行都已被“冻结”，不会出现事务 ID 回卷问题。这对于将把 datallowconn 设为假的数据库尤为重要，因为在这样的数据库中将无法进行例行的 VACUUM 维护。更多信息见第 21.1.3 节。

注意

template1 和 template0 除了以下事实之外没有任何特殊地位：template1 这个名字是 CREATE DATABASE 的默认源数据库名，也是 createdb 等多种程序的默认连接数据库。例如，可以删除 template1，然后再从 template0 重新创建它，而不会有任何不良后果。如果有人不小心在 template1 里加入了许多杂项对象，这样做也许是可取的。

18.4. 数据库配置

正如第 16.4 节所述，PostgreSQL服务器提供了大量运行时配置变量。你可以为其中许多设置指定数据库特定的默认值。

例如，如果你因为某种原因想在某个数据库中禁用GEQO优化器，
通常要么必须对所有数据库都禁用它，要么必须确保每个连入的客户端都会执行SET geqo TO off。要使这个设置成为默认值，可以执行命令

```
ALTER DATABASE mydb SET geqo TO off;
```

这会保存该设置（但不会立即生效）。以后连接到该数据库时，
它看起来就像是在会话开始前刚执行过SET geqo TO off;一样。注意，用户仍然可以在自己的会话中更改该设置；它只会作为默认值。要撤销这样的设置，可使用ALTER DATABASE *dbname* RESET *varname*。

18.5. 备选位置

可以在安装的默认位置之外的地方创建数据库。但请记住，
所有数据库访问都是通过数据库服务器进行的，
因此所指定的任何位置都必须是服务器可以访问的。

备选数据库位置通过一个环境变量来引用，该变量给出预期存储位置的绝对路径。
这个环境变量必须存在于服务器的环境中，因此它必须在服务器启动之前已被定义。（因此，
可用备选位置的集合由站点管理员控制；普通用户不能更改它。）
任何有效的环境变量名都可以用来引用备选位置，不过建议使用以 PGDATA 为前缀的变量名，以避免与其他变量混淆和冲突。

要在服务器进程的环境中创建变量，你必须先关闭服务器、定义变量、初始化数据区域，最后重新启动服务器。（另见第 16.6 节和第 16.3 节。）要设置环境变量，在 Bourne shell 中输入

```
PGDATA2=/home/postgres/data
export PGDATA2
```

或者在 `csh` 或 `tcsh` 中输入

```
setenv PGDATA2 /home/postgres/data
```

你必须确保这个环境变量始终在服务器环境中有定义，否则你将无法访问那个数据库。因此你大概会想在某个 shell 启动文件或服务器启动脚本中设置它。

要在 `PGDATA2` 中创建一个数据存储区域，请确保包含它的目录（这里是 `/home/postgres`）已经存在，并且运行服务器的用户账号可以写入它（见第 16.1 节）。然后在命令行输入

```
initlocation PGDATA2
```

（而不是 `initlocation $PGDATA2`）。然后就可以重新启动服务器了。

要在新位置中创建数据库，使用命令

```
CREATE DATABASE name WITH LOCATION 'location';
```

其中 *`location`* 是你使用的环境变量，本例中即 `PGDATA2`。`createdb` 命令有 `-D` 选项用于此目的。

在备选位置创建的数据库可以像任何其他数据库一样被访问和删除。

> **注意**
>
> 也可以直接向 `CREATE DATABASE` 命令指定绝对路径而不定义环境变量。默认情况下这是不允许的，因为它有安全风险。要允许这样做，你必须在编译 PostgreSQL 时定义 C 预处理器宏 `ALLOW_ABSOLUTE_DBPATHS`。一种做法是这样运行编译步骤：
>
> ```
> gmake CPPFLAGS=-DALLOW_ABSOLUTE_DBPATHS all
> ```

18.6. 删除数据库

数据库使用命令 `DROP DATABASE` 删除：

```
DROP DATABASE name;
```

只有数据库的拥有者（即创建它的用户）或超级用户才能删除数据库。删除数据库会移除其中包含的全部对象。数据库的销毁无法撤销。

当连接到目标数据库时，不能执行`DROP DATABASE`命令。不过，你可以连接到任何其他数据库，包括 `template1` 数据库。`template1` 将是删除某个给定集簇中最后一个用户数据库时的唯一选择。

为了方便，还有一个 shell 程序可以删除数据库：

```
dropdb dbname
```

（与 `createdb` 不同，删除与当前用户名同名的数据库并不是默认操作。）

第 19 章 客户端认证

当客户端应用连接到数据库服务器时，它会指定要以哪个 PostgreSQL 数据库用户名连接，这很像以某个特定用户身份登录 Unix 计算机。在 SQL 环境中，当前活动的数据库用户名决定了对数据库对象的访问权限——详见第 17 章。因此，必须限制哪些数据库用户能够连接。

认证是数据库服务器确认客户端身份的过程，
并据此决定是否允许该客户端应用（或者运行该客户端应用的用户）
以请求的数据库用户名进行连接。

PostgreSQL 提供了多种不同的客户端认证方法。用于认证特定客户端连接的方法可以根据（客户端）主机地址、数据库和用户来选择。

PostgreSQL 数据库用户名在逻辑上独立于服务器所在操作系统中的用户名。如果某台服务器的所有用户在该机器上也都有账号，
那么为他们分配与操作系统用户名一致的数据库用户名是有意义的。不过，
接受远程连接的服务器可能有许多数据库用户并没有本地操作系统账号，在这种情况下，
数据库用户名与操作系统用户名之间就不必存在任何对应关系。

19.1. pg_hba.conf 文件

客户端认证由数据目录中的文件 `pg_hba.conf` 控制，例如 `/usr/local/pgsql/data/pg_hba.conf`。（HBA 代表 host-based authentication，即基于主机的认证。）当数据目录由 `initdb` 初始化时，会安装一个默认的 `pg_hba.conf` 文件。

`pg_hba.conf` 文件的基本格式是一组记录，每行一条。空行会被忽略，# 注释字符之后的文本同样如此。每条记录由若干字段组成，字段之间用空格和/或制表符分隔。
如果字段值用引号括起来，就可以包含空白。记录不能跨行续写。

每条记录都指定一种连接类型、一个客户端 IP 地址范围（如果该连接类型需要）、一个数据库名、一个用户名，以及对匹配这些参数的连接要使用的认证方法。第一条同时匹配连接类型、
客户端地址、请求数据库和用户名的记录会被用来执行认证。这里不存在“继续向后匹配”
或“后备”机制：如果选中某条记录而认证失败，就不会再考虑后续记录。
如果没有任何记录匹配，则拒绝访问。

记录可以采用以下七种格式之一

```
local      database  user  authentication-method  [authentication-
option]
host       database  user  IP-address  IP-mask  authentication-method
 [authentication-option]
hostssl    database  user  IP-address  IP-mask  authentication-method
 [authentication-option]
hostnossl  database  user  IP-address  IP-mask  authentication-method
 [authentication-option]
host       database  user  IP-address/IP-masklen  authentication-
method  [authentication-option]
hostssl    database  user  IP-address/IP-masklen  authentication-
method  [authentication-option]
hostnossl  database  user  IP-address/IP-masklen  authentication-
method  [authentication-option]
```

各字段的含义如下：

`local`

该记录匹配使用 Unix 域套接字的连接尝试。没有这种类型的记录时，Unix 域套接字连接将被禁止。

`host`

该记录匹配使用 TCP/IP 网络进行的连接尝试。注意，除非服务器启动时带了 `-i` 选项，或者启用了 `tcpip_socket` 配置参数，否则 TCP/IP 连接是被禁用的。

`hostssl`

该记录匹配使用 TCP/IP 上的 SSL 进行的连接尝试。`host` 记录既匹配 SSL 连接尝试，也匹配非 SSL 连接尝试，但 `hostssl` 记录要求 SSL 连接。

要使用此选项，服务器编译时必须带有 SSL 支持启用。此外，还必须通过启用 `ssl` 配置参数来启用 SSL（更多信息参见第 16.4 节）。

`hostnossl`

这种记录类型与 `hostssl` 类似，但逻辑相反：它只匹配不使用 SSL 的常规连接尝试。

`database`

指定此记录匹配哪些数据库。值 `all` 表示匹配所有数据库。值 `sameuser` 表示当请求的数据库与请求的用户同名时记录匹配。值 `samegroup` 表示请求的用户必须是与请求数据库同名的组的成员。否则，这就是某个具体 PostgreSQL 数据库的名称。多个数据库名可以用逗号分隔提供。还可以在文件名前加 `@` 来指定一个包含数据库名的文件。该文件必须与 `pg_hba.conf` 位于同一目录中。

`user`

指定此记录匹配哪些 PostgreSQL 用户。值 `all` 表示匹配所有用户。否则，这就是某个具体 PostgreSQL 用户的名称。多个用户名可以用逗号分隔提供。组名可以在组名前加 `+` 来指定。还可以在文件名前加 `@` 来指定一个包含用户名的文件。该文件必须与 `pg_hba.conf` 位于同一目录中。

`IP-address`
`IP-mask`

这两个字段包含标准点分十进制表示的 IP 地址和掩码值。（IP 地址只能以数值形式指定，不能用域名或主机名。）二者合起来指定此记录匹配的客户端机器 IP 地址。精确的逻辑是

(*actual-IP-address* xor *IP-address-field*) and *IP-mask-field*

必须为零，记录才匹配。

以 IPv4 格式给出的 IP 地址会匹配具有相应地址的 IPv6 连接，例如 `127.0.0.1` 会匹配 IPv6 地址 `::ffff:127.0.0.1`。以 IPv6 格式给出的条目只匹配 IPv6 连接，即使所表示的地址位于 IPv4-in-IPv6 范围内也是如此。注意，如果系统的 C 库不支持 IPv6 地址，以 IPv6 格式给出的条目将被拒绝。

这些字段只适用于 `host`、`hostssl` 和 `hostnossl` 记录。

`IP-masklen`

此字段可用作 *`IP-mask`* 表示法的替代。它是一个整数，指定掩码中要置位的高位位数。该数字必须在 0 到 32（IPv4 地址的情形）或 128（IPv6 地址的情形）之间（含）。0 匹配任何

地址，而 32（或相应的 128）只匹配指定的那个确切主机。使用的匹配逻辑与点分表示法的 *IP-mask* 相同。

IP-address 与 / 之间以及 / 与 *IP-masklen* 之间不能有任何空白，否则文件将不能被正确解析。

此字段只适用于 `host`、`hostssl` 和 `hostnossl` 记录。

authentication-method

指定当连接匹配此记录时所使用的认证方法。可选方法在此概述，细节见第 19.2 节。

`trust`

无条件地允许连接。此方法允许任何能连接到 PostgreSQL 数据库服务器的人以他们希望的任何 PostgreSQL 用户身份登录，无需密码。详见第 19.2.1 节。

`reject`

无条件地拒绝连接。这有助于从一组主机中“过滤掉”某些主机。

`md5`

要求客户端提供经 MD5 加密的密码进行认证。这是唯一允许把加密密码存储在 `pg_shadow` 中的方法。详见第 19.2.2 节。

`crypt`

类似于 `md5` 方法，但使用较旧的 `crypt()` 加密，pre-7.2 客户端需要它。对于 7.2 及以后的客户端，首选 `md5`。详见第 19.2.2 节。

`password`

与 `md5` 相同，但密码以明文形式在网络上发送。不应在不可信网络上使用此方法。详见第 19.2.2 节。

`krb4`

使用 Kerberos V4 认证用户。这只适用于 TCP/IP 连接。详见第 19.2.3 节。

`krb5`

使用 Kerberos V5 认证用户。这只适用于 TCP/IP 连接。详见第 19.2.3 节。

`ident`

获取客户端的操作系统用户名（对 TCP/IP 连接，通过联系客户端上的 ident 服务器获取；对本地连接，从操作系统获取），并通过查询 `ident` 关键字之后指定的映射，检查该用户是否被允许以请求的数据库用户身份连接。

如果使用映射 `sameuser`，则要求用户名相同。否则，会在与 `pg_hba.conf` 位于同一目录的文件 `pg_ident.conf` 中查找映射名。如果该文件中包含此映射名下操作系统用户名与所请求的 PostgreSQL 用户名的一条条目，则接受连接。

对本地连接，这只在支持 Unix 域套接字凭据的机器上有效（目前有 Linux、FreeBSD、NetBSD、OpenBSD 和 BSD/OS）。

详细信息见下面的第 19.2.4 节。

pam

使用操作系统提供的可插拔认证模块（PAM）服务进行认证。详见第 19.2.5 节。

authentication-option

这个可选字段的含义取决于所选的认证方法，将在下一节中描述。

由于对每次连接尝试都是顺序检查 pg_hba.conf 记录，记录的顺序是重要的。通常，靠前的记录有较紧的连接匹配参数和较弱的认证方法，
而靠后的记录有较松的匹配参数和较强的认证方法。例如，你可能希望对本地 TCP/IP 连接使用 trust 认证，但对远程 TCP/IP 连接要求密码。这种情况下，为来自 127.0.0.1 的连接指定 trust 认证的记录应当出现在为更宽的允许客户端 IP 地址范围指定密码认证的记录之前。

重要

不要阻止超级用户访问 template1 数据库。各种工具命令需要访问 template1。

pg_hba.conf 文件在启动时以及主服务器进程（postmaster）收到 SIGHUP 信号时读取。如果在运行的系统上编辑了该文件，你需要（使用 pg_ctl reload 或 kill -HUP）向 postmaster 发信号，让它重新读取该文件。

一个 pg_hba.conf 文件的例子见例 19.1。各种认证方法的细节见下一节。

例 19.1. 一个 pg_hba.conf 文件示例

```
# Allow any user on the local system to connect to any database under
# any user name using Unix-domain sockets (the default for local
# connections).
#
# TYPE  DATABASE    USER        IP-ADDRESS        IP-MASK
 METHOD
local   all         all
 trust

# The same using local loopback TCP/IP connections.
#
# TYPE  DATABASE    USER        IP-ADDRESS        IP-MASK
 METHOD
host    all         all         127.0.0.1         255.255.255.255
 trust

# The same as the last line but using a CIDR mask
#
# TYPE  DATABASE    USER        IP-ADDRESS/CIDR-mask  METHOD
host    all         all         127.0.0.1/32          trust

# Allow any user from any host with IP address 192.168.93.x to connect
# to database "template1" as the same user name that ident reports for
# the connection (typically the Unix user name).
```

```
#
# TYPE  DATABASE    USER        IP-ADDRESS        IP-MASK
 METHOD
host    template1   all         192.168.93.0      255.255.255.0
 ident sameuser

# The same as the last line but using a CIDR mask
#
# TYPE  DATABASE    USER        IP-ADDRESS/CIDR-mask  METHOD
host    template1   all         192.168.93.0/24       ident sameuser

# Allow a user from host 192.168.12.10 to connect to database
# "template1" if the user's password is correctly supplied.
#
# TYPE  DATABASE    USER        IP-ADDRESS        IP-MASK
 METHOD
host    template1   all         192.168.12.10     255.255.255.255
 md5

# In the absence of preceding "host" lines, these two lines will
# reject all connection from 192.168.54.1 (since that entry will be
# matched first), but allow Kerberos V connections from anywhere else
# on the Internet.  The zero mask means that no bits of the host IP
# address are considered so it matches any host.
#
# TYPE  DATABASE    USER        IP-ADDRESS        IP-MASK
 METHOD
host    all         all         192.168.54.1      255.255.255.255
 reject
host    all         all         0.0.0.0           0.0.0.0
 krb5

# Allow users from 192.168.x.x hosts to connect to any database, if
# they pass the ident check.  If, for example, ident says the user is
# "bryanh" and he requests to connect as PostgreSQL user "guest1", the
# connection is allowed if there is an entry in pg_ident.conf for map
# "omicron" that says "bryanh" is allowed to connect as "guest1".
#
# TYPE  DATABASE    USER        IP-ADDRESS        IP-MASK
 METHOD
host    all         all         192.168.0.0       255.255.0.0
 ident omicron

# If these are the only three lines for local connections, they will
# allow local users to connect only to their own databases (databases
# with the same name as their user name) except for administrators and
# members of group "support" who may connect to all databases.  The
 file
# $PGDATA/admins contains a list of user names.  Passwords are
 required in
# all cases.
#
# TYPE  DATABASE    USER        IP-ADDRESS        IP-MASK
 METHOD
```

```
local   sameuser    all
 md5
local   all         @admins
 md5
local   all         +support
 md5

# The last two lines above can be combined into a single line:
local   all         @admins,+support
 md5

# The database column can also use lists and file names, but not
 groups:
local   db1,db2,@demodbs  all
 md5
```

19.2. 认证方法

下面详细描述这些认证方法。

19.2.1. trust 认证

当trust认证被指定时，PostgreSQL假设任何可以连接到服务器的人都被授权使用他们指定的任何数据库用户名（包括数据库超级用户）访问数据库。只有当在操作系统层对进入服务器的连接有足够保护时，才应该使用这种方法。

trust 认证对单用户工作站上的本地连接来说是合适且非常方便的。在多用户机器上，通常单用它并不合适。不过，如果用文件系统权限限制对服务器 Unix 域套接字文件的访问，即使在多用户机器上也可以使用 trust。为此，可按第 16.4.1 节中的说明设置 unix_socket_permissions（可能还有 unix_socket_group）配置参数。或者，也可以设置 unix_socket_directory 配置参数，把套接字文件放在一个经过适当限制的目录中。

设置文件系统权限只能有助于 Unix 套接字连接。本地 TCP/IP 连接不会被它限制。因此，如果你想利用文件系统权限来控制本地安全，那么从pg_hba.conf中移除host ... 127.0.0.1 ...行，或者把它改为一个非trust认证方法。

只有当你信任由 pg_hba.conf 中指定 trust 的行所允许连接的每台机器上的每个用户时，trust 认证才适合用于 TCP/IP 连接。对来自 localhost（127.0.0.1）以外的任何 TCP/IP 连接使用 trust，通常都不合理。

19.2.2. 密码认证

基于密码的认证方法有 md5、crypt 和 password。除了密码通过连接发送的方式之外，这些方法的操作是相似的。如果你对密码“嗅探”攻击有任何顾虑，应首选 md5；如果必须支持 pre-7.2 客户端，则次选 crypt。对通过开放 Internet 的连接（除非你在连接周围使用 SSL、SSH 或其他通信安全封装），尤其应避免使用明文 password。

PostgreSQL 数据库密码与操作系统用户密码是分开的。每个数据库用户的密码存储在 pg_shadow 系统目录表中。密码可以用 SQL 命令 CREATE USER 和 ALTER USER 管理，例如 **CREATE USER foo WITH PASSWORD 'secret';**。默认情况下，即如果没有设置密码，存储的密码为空，该用户的密码认证将总是失败。

要限制允许连接到某些数据库的用户集合，可在 `pg_hba.conf` 的 *`user`* 列中列出这些用户，如前一节所解释的那样。

19.2.3. Kerberos 认证

Kerberos 是一种行业标准的、适合在公共网络上进行分布式计算的安全认证系统。对 Kerberos 系统的描述远超出本文档的范围；就一般性而言，它可能相当复杂（然而强大）。Kerberos FAQ[1] 或 MIT Project Athena[2] 可以作为入门的起点。存在若干 Kerberos 发行版来源。

虽然 PostgreSQL 同时支持 Kerberos 4 和 Kerberos 5，但只推荐 Kerberos 5。Kerberos 4 被认为不安全，不再推荐一般使用。

要使用 Kerberos，必须在构建时启用对它的支持。更多信息见第 14 章。Kerberos 4 和 5 都受支持，但任何一个构建中只能支持一个版本。

PostgreSQL 像一个正常的 Kerberos 服务那样运作。服务主体的名称是 *`servicename/hostname@realm`*，其中 *`servicename`* 是 `postgres`（除非在 configure 时用 `./configure --with-krb-srvnam=whatever` 选择了不同的服务名）。*`hostname`* 是服务器机器的完全限定主机名。服务主体的 realm 是服务器机器的首选 realm。

客户端主体必须以它们的 PostgreSQL 用户名作为第一个组成部分，例如 `pgusername/otherstuff@realm`。目前 PostgreSQL 不检查客户端的 realm；因此，如果你启用了跨 realm 认证，那么任何能与你的 realm 通信的任何 realm 中的任何主体都会被接受。

确保你的服务器密钥文件可被 PostgreSQL 服务器账号读取（而且最好只被它读取）。（另见第 16.1 节。）密钥文件的位置由运行时配置参数 `krb_server_keyfile` 指定。（另见第 16.4 节。）默认值：使用 Kerberos 4 时为 `/etc/srvtab`，使用 Kerberos 5 时为 `FILE:/usr/local/pgsql/etc/krb5.keytab`（或构建时指定为 `sysconfdir` 的那个目录）。

要生成 keytab 文件，例如可以（用版本 5）使用

```
kadmin% ank -randkey postgres/server.my.domain.orgkadmin% ktadd -k
 krb5.keytab postgres/server.my.domain.org
```

细节请阅读 Kerberos 文档。

连接到数据库时，确保你持有与所请求的数据库用户名匹配的主体的票据。
一个例子：对于数据库用户名 `fred`，主体 `fred@EXAMPLE.COM` 和 `fred/users.example.com@EXAMPLE.COM` 都可以用来向数据库服务器认证。

如果你在 Apache Web 服务器上使用来自 http://modauthkerb.sf.net 的 mod_auth_kerb 和 mod_perl，你可以在 mod_perl 脚本中使用 `AuthType KerberosV5SaveCredentials`。这能通过 Web 安全地访问数据库，无需额外的密码。

19.2.4. 基于 ident 的认证

ident 认证方法的工作方式是获取客户端的操作系统用户名，并用一个列出允许的对应用户名对的映射文件来确定允许的数据库用户名。确定客户端用户名是安全上的关键点，
其工作方式因连接类型而异。

19.2.4.1. TCP/IP 上的 Ident 认证

“Identification Protocol”（识别协议）在 RFC 1413 中描述。几乎每个类 Unix 操作系统都附带一个 ident 服务器，默认监听 TCP 端口 113。ident 服务器的基本功能是回答诸如“哪个用

[1] http://www.nrl.navy.mil/CCS/people/kenh/kerberos-faq.html
[2] ftp://athena-dist.mit.edu

户发起了从你的端口 *X* 出来并连接到我的端口 *Y* 的连接？”这样的问题。由于在建立物理连接时 PostgreSQL 同时知道 *X* 和 *Y*，它可以查询连接客户端主机上的 ident 服务器，并由此在理论上确定任何给定连接的操作系统用户。

这一过程的缺点是它依赖于客户端的完整性：如果客户端机器不可信或已被攻破，
攻击者可以在端口 113 上运行几乎任何程序并返回他选择的任何用户名。因此这种认证方法只适用于每台客户端机器都处于严格控制之下、且数据库与系统管理员保持密切联系的封闭网络。
换句话说，你必须信任运行 ident 服务器的那台机器。请留意这个警告：

> 标识协议（Identification Protocol）并不打算用作授权或访问控制协议。
>
> —RFC 1413

19.2.4.2. 本地套接字上的 Ident 认证

在支持对 Unix 域套接字的 `SO_PEERCRED` 请求的系统上（目前有 Linux、FreeBSD、NetBSD、OpenBSD 和 BSD/OS），ident 认证也可以应用于本地连接。这种情况下，使用 ident 认证不会增加安全风险；实际上，在这样的系统上它是本地连接的更可取选择。

在不支持 `SO_PEERCRED` 请求的系统上，ident 认证只适用于 TCP/IP 连接。作为一种变通办法，可以指定 localhost 地址 127.0.0.1 并连接到该地址。

19.2.4.3. Ident 映射

使用基于 ident 的认证时，在确定了发起连接的操作系统用户的名称之后，PostgreSQL会检查该用户是否被允许以他请求连接的数据库用户身份连接。这由 `pg_hba.conf` 文件中跟在 `ident` 关键字之后的 ident 映射参数控制。有一个预定义的 ident 映射 `sameuser`，它允许任何操作系统用户以同名的数据库用户身份连接（如果后者存在）。其他映射必须手工创建。

`sameuser` 之外的 ident 映射定义在数据目录的文件 `pg_ident.conf` 中，该文件包含如下一般形式的行：

map-nameident-usernamedatabase-username

注释和空白的处理与通常一样。*map-name* 是一个任意名称，将用于在 `pg_hba.conf` 中引用这个映射。另外两个字段指定哪个操作系统用户被允许以哪个数据库用户身份连接。同一个 *map-name* 可以重复使用，以在单个映射中指定更多的用户映射。一个给定的操作系统用户可以对应多少个数据库用户、反之一个数据库用户可以对应多少个操作系统用户，都没有限制。

`pg_ident.conf` 文件在启动时以及主服务器进程（`postmaster`）收到 SIGHUP 信号时读取。如果在运行的系统上编辑了该文件，你需要（使用 `pg_ctl reload` 或 `kill -HUP`）向 `postmaster` 发信号，让它重新读取该文件。

一个可与例 19.1中的 `pg_hba.conf` 文件配合使用的 `pg_ident.conf` 文件见例 19.2。在这个示例设置中，任何登录到 192.168 网络某台机器上、但 Unix 用户名不是 `bryanh`、`ann` 或 `robert` 的人都不会被授予访问权限。Unix 用户 `robert` 只有在他尝试以 PostgreSQL 用户 `bob` 而不是 `robert` 或任何其他人身份连接时才被允许访问。`ann` 只被允许以 `ann` 身份连接。用户 `bryanh` 被允许以 `bryanh` 本人或 `guest1` 身份连接。

例 19.2. 一个 `pg_ident.conf` 文件示例

```
# MAPNAME     IDENT-USERNAME    PG-USERNAME

omicron       bryanh            bryanh
omicron       ann               ann
```

```
# bob has user name robert on these machines
omicron       robert            bob
# bryanh can also connect as guest1
omicron       bryanh            guest1
```

19.2.5. PAM 认证

这种认证方法的工作方式与 `password` 类似，只不过它使用 PAM（可插拔认证模块）作为认证机制。默认的 PAM 服务名为 `postgresql`。你可以在文件 `pg_hba.conf` 中 `pam` 关键字之后可选地提供你自己的服务名。关于 PAM 的更多信息，请阅读 Linux-PAM 页面[3]和 Solaris PAM 页面[4]。

19.3. 认证问题

认证失败及相关问题通常会表现为类似下面这样的错误消息：

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym",
 database "testdb"
```

如果你成功联系到了服务器、但服务器不想与你交谈，最可能得到的就是这个。正如消息所示，服务器拒绝了该连接请求，因为它没有在自己的 `pg_hba.conf` 配置文件中找到匹配项。

```
FATAL:  Password authentication failed for user "andym"
```

这样的消息表示你已经联系到了服务器，而且服务器也愿意继续处理连接，
但前提是你必须先通过 `pg_hba.conf` 文件中指定的认证方法。请检查你提供的密码；如果错误消息提到了 Kerberos 或 ident 等认证类型，也请检查对应的软件配置。

```
FATAL:  user "andym" does not exist
```

指定的数据库用户不存在。

```
FATAL:  database "testdb" does not exist
```

你试图连接的数据库不存在。注意，如果你没有指定数据库名，
默认会使用数据库用户名作为数据库名，但这不一定是所需的数据库名。

> **提示**
>
> 服务器日志中可能包含比返回给客户端的更多认证失败信息。如果你不清楚失败原因，请检查服务器日志。

[3]http://www.kernel.org/pub/linux/libs/pam/
[4]http://www.sun.com/software/solaris/pam/

第 20 章 本地化

本章从管理员的角度描述可用的本地化特性。PostgreSQL通过两种方式支持本地化：

- 使用操作系统的区域特性来提供与区域相关的排序顺序、数字格式、翻译消息以及其他方面。
- 提供在PostgreSQL服务器中定义的多种不同字符集，包括多字节字符集，以支持存储各种语言的文本，并提供客户端与服务器之间的字符集转换。

20.1. 区域支持

区域设置支持是指应用程序在字母表、排序、数字格式等方面尊重文化偏好。PostgreSQL使用服务器操作系统提供的标准 ISO C 和POSIX区域设置机制。更多信息请参考系统文档。

20.1.1. 概述

在使用`initdb`创建数据库集簇时，区域设置支持会自动初始化。`initdb`会使用其执行环境中的区域设置来初始化数据库集簇，因此如果你的系统已经设置为数据库集簇想要使用的区域设置，就无需额外操作。如果想使用不同的区域设置（或者不确定系统当前设置了哪种区域设置），可以通过指定`--locale`选项，精确告诉`initdb`使用哪种区域设置。例如：

```
initdb --locale=sv_SE
```

这个例子把区域设置为瑞典（`SE`）所讲的瑞典语（`sv`）。其他可能的设置有`en_US`（美国英语）和`fr_CA`（加拿大法语）。如果一个区域可以使用多种字符集，那么规范可以采用`cs_CZ.ISO8859-2`这样的形式。系统上有哪些区域设置可用，以及它们以什么名称出现，取决于操作系统供应商提供了什么以及实际安装了什么。

有时混合多个区域设置的规则会很有用，例如使用英语排序规则但使用西班牙语消息。为此，存在一组只控制本地化规则某些方面的区域设置子类别：

`LC_COLLATE`	字符串排序顺序
`LC_CTYPE`	字符分类（什么算字母？它的大写等价形式是什么？）
`LC_MESSAGES`	消息的语言
`LC_MONETARY`	货币数量的格式
`LC_NUMERIC`	数字的格式
`LC_TIME`	日期和时间的格式

这些类别名会转换成`initdb`选项名，用于覆盖特定类别的区域设置选择。例如，要将区域设置设为加拿大法语，但对货币格式采用美国规则，可以使用`initdb --locale=fr_CA --lc-monetary=en_US`。

如果希望系统的行为如同没有区域支持一样，可以使用特殊区域`C`或`POSIX`。

有些区域类别的性质决定了其值必须在数据库集簇的整个生命周期内保持固定。也就是说，一旦`initdb`运行过，就无法再更改它们。`LC_COLLATE`和`LC_CTYPE`就属于这类类别。它们影响索引的排序顺序，因此必须保持固定，否则文本列上的索引会损坏。

PostgreSQL通过记录initdb所见的LC_COLLATE和LC_CTYPE值来强制这一点。
服务器启动时会自动采用这两个值。

其他区域类别可以在服务器运行期间随时通过设置与区域类别同名的运行时配置变量来更改（细节见第 16.4 节）。initdb所选择的默认值实际上只是写入了配置文件postgresql.conf，作为服务器启动时的默认值。如果你从postgresql.conf中删除这些赋值，
服务器将从其执行环境继承设置。

请注意，服务器的区域设置行为由服务器看到的环境变量决定，
而不是由任何客户端的环境决定。因此，务必在启动服务器前配置好正确的区域设置。
由此带来的一个结果是，如果客户端和服务器使用不同的区域设置，
消息可能会因其来源不同而显示为不同语言。

> **注意**
>
> 这里所说的从执行环境继承区域设置，
> 在大多数操作系统上是指：对于某个给定的区域设置类别，例如排序规则，
> 会按以下顺序检查环境变量，直到找到一个已设置的变量为止：LC_ALL、LC_COLLATE（或相应类别对应的变量）、LANG。如果这些环境变量都没有设置，则区域设置默认为C。
>
> 某些消息本地化库还会查看环境变量LANGUAGE，
> 它会覆盖所有其他用于设置消息语言的区域设置。如果有疑问，请参考操作系统的文档，
> 尤其是关于gettext的文档。

若要让消息可以翻译为用户偏好的语言，构建时必须启用NLS。
这一选择独立于其他区域设置支持。

20.1.2. 优点

区域支持尤其会影响下列特性：

- 使用ORDER BY的查询中的排序顺序
- to_char函数族

在PostgreSQL中使用区域支持的唯一严重缺点是速度。因此，
只有在确实需要时才使用区域设置。

20.1.3. 问题

如果区域设置支持没有像上面说明的那样工作，
请检查操作系统中的区域设置支持是否已经正确配置。要检查系统中安装了哪些区域设置，
可以使用 locale -a命令（如果操作系统提供该命令）。

请确认PostgreSQL实际使用的区域设置就是你预期的区域设置。LC_COLLATE和LC_CTYPE是在initdb时确定的，不重新运行initdb就无法更改。其他区域设置（包括LC_MESSAGES和LC_MONETARY）最初由启动服务器的环境决定。你可以使用实用程序pg_controldata检查数据库的LC_COLLATE和LC_CTYPE设置。

源码发布包中的src/test/locale目录包含 PostgreSQL区域设置支持的测试套件。

那些通过解析错误消息文本来处理服务器端错误的客户端应用，显然会遇到问题，因为服务器消息可能使用不同语言。这类应用的作者应改用错误代码机制。

维护消息翻译目录需要许多志愿者持续投入，他们希望 PostgreSQL能够良好地支持他们偏好的语言。如果你所用语言的消息目前还不可用，或尚未完全翻译，我们将非常感谢你的协助。如果你愿意帮忙，请参考第 46 章，或向开发者邮件列表写信。

20.2. 字符集支持

PostgreSQL中的字符集支持允许你以多种字符集存储文本，包括 ISO 8859 系列之类的单字节字符集，以及 EUC（扩展 Unix 编码）、Unicode 和 Mule 内部编码等多字节字符集。所有字符集都可以在整个服务器中透明地使用。（如果你使用来自其他来源的扩展函数，则取决于它们的代码编写得是否正确。）默认字符集是在使用 `initdb`初始化PostgreSQL 数据库集簇时选定的。创建数据库时可以使用`createdb` 或 SQL 命令`CREATE DATABASE`覆盖该设置。因此你可以拥有多个数据库，并让每个数据库使用不同的字符集。

20.2.1. 支持的字符集

表 20.1显示了服务器中可用的字符集。

表 20.1. 服务器字符集

名称	描述
`SQL_ASCII`	ASCII
`EUC_JP`	日语 EUC
`EUC_CN`	中文 EUC
`EUC_KR`	韩语 EUC
`JOHAB`	韩语 EUC（基于韩文字母）
`EUC_TW`	台湾 EUC
`UNICODE`	Unicode（UTF-8）
`MULE_INTERNAL`	Mule 内部编码
`LATIN1`	ISO 8859-1/ECMA 94（拉丁字母 no.1）
`LATIN2`	ISO 8859-2/ECMA 94（拉丁字母 no.2）
`LATIN3`	ISO 8859-3/ECMA 94（拉丁字母 no.3）
`LATIN4`	ISO 8859-4/ECMA 94（拉丁字母 no.4）
`LATIN5`	ISO 8859-9/ECMA 128（拉丁字母 no.5）
`LATIN6`	ISO 8859-10/ECMA 144（拉丁字母 no.6）
`LATIN7`	ISO 8859-13（拉丁字母 no.7）
`LATIN8`	ISO 8859-14（拉丁字母 no.8）
`LATIN9`	ISO 8859-15（拉丁字母 no.9）
`LATIN10`	ISO 8859-16/ASRO SR 14111（拉丁字母 no.10）
`ISO_8859_5`	ISO 8859-5/ECMA 113（拉丁/西里尔）
`ISO_8859_6`	ISO 8859-6/ECMA 114（拉丁/阿拉伯）
`ISO_8859_7`	ISO 8859-7/ECMA 118（拉丁/希腊）
`ISO_8859_8`	ISO 8859-8/ECMA 121（拉丁/希伯来）
`KOI8`	KOI8-R(U)

名称	描述
WIN	Windows CP1251
ALT	Windows CP866
WIN1256	Windows CP1256（阿拉伯语）
TCVN	TCVN-5712/Windows CP1258（越南语）
WIN874	Windows CP874（泰语）

重要

在PostgreSQL 7.2 之前，LATIN5错误地表示 ISO 8859-5。从 7.2 开始，LATIN5表示 ISO 8859-9。如果你有一个在 7.1 或更早版本上创建的LATIN5数据库，并且想迁移到 7.2 或更高版本，就应该小心这个变化。

并非所有API都支持所有列出的字符集。例如，PostgreSQL的 JDBC 驱动不支持MULE_INTERNAL、LATIN6、LATIN8和LATIN10。

20.2.2. 设置字符集

initdb为PostgreSQL集簇定义默认字符集（编码）。例如：

```
initdb -E EUC_JP
```

这会把默认字符集（编码）设为EUC_JP（日语的扩展 Unix 编码）。如果你喜欢更长的选项形式，也可以用--encoding代替-E。如果既没有给出-E也没有给出 --encoding，就使用SQL_ASCII。

你可以创建使用不同字符集的数据库：

```
createdb -E EUC_KR korean
```

这将创建一个名为korean的数据库，它使用 EUC_KR字符集。另一种实现方式是使用以下 SQL 命令：

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR';
```

数据库的编码存储在系统目录pg_database中。可以通过 psql的-l选项或\l 命令查看它。

```
$ psql -l
            List of databases
   Database    |  Owner   |   Encoding
---------------+---------+---------------
 euc_cn        | t-ishii | EUC_CN
 euc_jp        | t-ishii | EUC_JP
 euc_kr        | t-ishii | EUC_KR
 euc_tw        | t-ishii | EUC_TW
 mule_internal | t-ishii | MULE_INTERNAL
 regression    | t-ishii | SQL_ASCII
 template1     | t-ishii | EUC_JP
 test          | t-ishii | EUC_JP
 unicode       | t-ishii | UNICODE
(9 rows)
```

20.2.3. 服务器和客户端之间的自动字符集转换

对于某些字符集，PostgreSQL支持在服务器和客户端之间进行自动字符集转换。
转换信息保存在pg_conversion系统目录中。可以使用 SQL 命令CREATE CONVERSION创建新的转换。PostgreSQL自带一些预定义转换，它们列在表 20.2中。

表 20.2. 客户端/服务器字符集转换

服务器字符集	可用的客户端字符集
SQL_ASCII	SQL_ASCII, UNICODE, MULE_INTERNAL
EUC_JP	EUC_JP, SJIS, UNICODE, MULE_INTERNAL
EUC_CN	EUC_CN, UNICODE, MULE_INTERNAL
EUC_KR	EUC_KR, UNICODE, MULE_INTERNAL
JOHAB	JOHAB, UNICODE
EUC_TW	EUC_TW, BIG5, UNICODE, MULE_INTERNAL
LATIN1	LATIN1, UNICODE MULE_INTERNAL
LATIN2	LATIN2, WIN1250, UNICODE, MULE_INTERNAL
LATIN3	LATIN3, UNICODE, MULE_INTERNAL
LATIN4	LATIN4, UNICODE, MULE_INTERNAL
LATIN5	LATIN5, UNICODE
LATIN6	LATIN6, UNICODE, MULE_INTERNAL
LATIN7	LATIN7, UNICODE, MULE_INTERNAL
LATIN8	LATIN8, UNICODE, MULE_INTERNAL
LATIN9	LATIN9, UNICODE, MULE_INTERNAL
LATIN10	LATIN10, UNICODE, MULE_INTERNAL
ISO_8859_5	ISO_8859_5, UNICODE, MULE_INTERNAL, WIN, ALT, KOI8
ISO_8859_6	ISO_8859_6, UNICODE
ISO_8859_7	ISO_8859_7, UNICODE
ISO_8859_8	ISO_8859_8, UNICODE
UNICODE	EUC_JP, SJIS, EUC_KR, UHC, JOHAB, EUC_CN, GBK, EUC_TW, BIG5, LATIN1 to LATIN10, ISO_8859_5, ISO_8859_6, ISO_8859_7, ISO_8859_8, WIN, ALT, KOI8, WIN1256, TCVN, WIN874, GB18030, WIN1250
MULE_INTERNAL	EUC_JP, SJIS, EUC_KR, EUC_CN, EUC_TW, BIG5 , LATIN1 to LATIN5, WIN, ALT, WIN1250, BIG5, ISO_8859_5, KOI8
KOI8	ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
WIN	ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
ALT	ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
WIN1256	WIN1256, UNICODE
TCVN	TCVN, UNICODE

服务器字符集	可用的客户端字符集
WIN874	WIN874,UNICODE

若要启用自动字符集转换，必须告诉PostgreSQL 你希望客户端使用哪种字符集（编码）。有几种方法可以做到这一点：

- 在psql中使用\encoding 命令。\encoding允许动态更改客户端编码。例如，要把编码改成SJIS，可以输入：

  ```
  \encoding SJIS
  ```

- 使用libpq函数。\encoding实际上就是为此目的调用 PQsetClientEncoding()的。

  ```
  int PQsetClientEncoding(PGconn *conn, const char *encoding);
  ```

  其中*conn*是到服务器的连接，*encoding*是你想使用的编码。如果函数成功设置了编码，它返回 0，否则返回 -1。此连接的当前编码可以通过下面的函数确定：

  ```
  int PQclientEncoding(const PGconn *conn);
  ```

  注意它返回的是编码 ID，而不是像EUC_JP这样的符号字符串。要把编码 ID 转换为编码名，可以使用：

  ```
  char *pg_encoding_to_char(int encoding_id);
  ```

- 使用SET client_encoding TO。可以使用以下 SQL 命令设置客户端编码：

  ```
  SET CLIENT_ENCODING TO 'value';
  ```

  也可以为此使用更标准的 SQL 语法SET NAMES：

  ```
  SET NAMES 'value';
  ```

  要查询当前客户端编码：

  ```
  SHOW client_encoding;
  ```

  要返回到默认编码：

  ```
  RESET client_encoding;
  ```

- 使用PGCLIENTENCODING。如果在客户端环境中定义了PGCLIENTENCODING环境变量，那么连接服务器时会自动选择该客户端编码。
  （之后仍可以用上面提到的任何其他方法覆盖它。）

- 使用配置变量client_encoding。如果设置了postgresql.conf中的client_encoding变量，那么连接服务器时会自动选择该客户端编码。
  （之后仍可以用上面提到的任何其他方法覆盖它。）

如果某个特定字符无法转换——例如你为服务器选择了EUC_JP，为客户端选择了LATIN1，那么某些日语字符无法转换为LATIN1——该字符会被转换成括在圆括号中的十六进制字节值，例如(826C)。

20.2.4. 进一步阅读

这些都是开始了解各种编码系统的良好资料。

ftp://ftp.ora.com/pub/examples/nutshell/ujip/doc/cjk.inf

对`EUC_JP`、`EUC_CN`、`EUC_KR`、`EUC_TW`的详细说明见 3.2 节。

http://www.unicode.org/

Unicode Consortium 的网站

RFC 2044

UTF-8 在这里定义。

第 21 章 日常数据库维护任务

为了保持PostgreSQL服务器平稳运行，有一些必须定期执行的例行维护事务。
这里讨论的任务本质上是重复性的，使用诸如cron脚本之类的标准 Unix 工具就可以轻松自动化。
但是，建立合适的脚本并检查它们是否成功执行是数据库管理员的职责。

一项显而易见的维护任务是按固定的计划定期创建数据的备份副本。如果没有较新的备份，
在灾难（磁盘故障、火灾、误删除关键表等）发生之后你就没有任何恢复的机会。
PostgreSQL中可用的备份和恢复机制在第 22 章中有详细讨论。

另一类主要的维护任务是对数据库进行周期性的“清理”（vacuuming）。这项活动在第 21.1 节中讨论。

还有一件可能需要周期性关注的事情是日志文件管理。这在第 21.3 节中讨论。

与其他一些数据库管理系统相比，PostgreSQL 的维护需求较低。尽管如此，对这些任务给予适当的关注，将大大有助于确保你愉快而高效地使用这个系统。

21.1. 例行清理

PostgreSQL的VACUUM命令必须定期运行，原因有以下几点：

1. 回收被更新或删除的行占用的磁盘空间。
2. 更新PostgreSQL查询规划器使用的数据统计信息。
3. 防止因事务 ID 回卷（transaction ID wraparound）而丢失非常旧的数据。

出于这些原因而执行的VACUUM操作的频率和范围，因各站点的需求而异。因此，
数据库管理员必须理解这些问题并制定合适的维护策略。本节集中解释高层面的问题；
关于命令语法等细节，请参阅VACUUM命令参考页。

从PostgreSQL 7.2 开始，标准形式的 VACUUM可以与正常的数据库操作（选择、插入、更新、删除，但不包括对表定义的更改）并行运行。因此，例行清理不像以前的版本那样具有侵入性，
也就不必那么刻意地安排在一天中使用率低的时段进行。

21.1.1. 回收磁盘空间

在PostgreSQL的正常操作中，对一行执行UPDATE或DELETE并不会立即删除该行的旧版本。
为了获得多版本并发控制的好处（见第 12 章），这种做法是必要的：在其他事务仍可能看到某个行版本时，不能删除它。但最终，一个过期或已删除的行版本不再被任何事务关注。
它占用的空间必须被回收以供新行重用，以避免磁盘空间需求无限增长。这项工作通过运行VACUUM完成。

显然，频繁更新或删除的表需要比很少更新的表更经常地清理。设置只清理选定表的周期性
cron任务可能有用，跳过已知不常变化的表。
只有当你同时有大型的频繁更新表和大型的不常更新表时，
这才可能有帮助——清理一个小表的额外开销不值得担心。

标准形式的VACUUM最适合以维持磁盘空间的平稳占用为目标。
标准形式会找到旧的行版本并使其空间可在表内重用，
但它并不十分努力地缩短表文件并把磁盘空间归还给操作系统。
如果你需要把磁盘空间归还给操作系统，可以使用 VACUUM FULL——但释放出的磁盘空间如果很

快又要重新分配，这样做又有什么意义呢？对于维护频繁更新的表，较频繁的标准VACUUM 运行比不频繁的 VACUUM FULL运行是更好的做法。

对大多数站点，推荐的做法是每天在使用率低的时段安排一次数据库范围的VACUUM，必要时辅以对频繁更新表的更频繁清理。（如果集群中有多个数据库，别忘了对每个数据库都进行清理；vacuumdb程序可能会有帮助。）例行的空间回收清理应使用普通的VACUUM，而不是 VACUUM FULL。

在你确知已删除了一个表中大部分行的情况下，推荐使用 VACUUM FULL，这样表的稳态尺寸可以通过 VACUUM FULL更激进的方式大幅收缩。

如果你有一个表的内容会周期性地被完全删除，可以考虑用 TRUNCATE来完成，而不是DELETE 后跟 VACUUM。

21.1.2. 更新规划器统计信息

PostgreSQL查询规划器依靠关于表内容的统计信息来为查询生成好的计划。这些统计信息由 ANALYZE命令收集，该命令可以单独调用，也可以作为 VACUUM的一个可选步骤调用。拥有相当准确的统计信息很重要，否则糟糕的计划选择可能降低数据库性能。

与为回收空间而清理一样，频繁更新统计信息对频繁更新的表比对不常更新的表更有用。但即使是频繁更新的表，如果数据的统计分布变化不大，也可能不需要更新统计信息。一个简单的经验法则想一想表中各列的最小值和最大值变化有多大。例如，一个包含行更新时间的timestamp列，随着行的添加和更新，其最大值会不断增加；这样的列可能比一个包含网站页面 URL 的列需要更频繁的统计更新。URL 列可能同样经常变化，但其值的统计分布可能变化相对较慢。

可以在特定的表甚至表的特定列上运行ANALYZE，因此如果你的应用需要，可以灵活地比其他统计信息更频繁地更新某些统计信息。然而在实践中，这个特性的用处值得怀疑。从PostgreSQL 7.2 开始，ANALYZE即使在大表上也是相当快的操作，因为它使用对表行的统计随机抽样而不是读取每一行。所以，隔一段时间就对整个数据库运行一次它，可能要简单得多。

> **提示**
>
> 虽然按列调整ANALYZE的频率可能收效不大，但你很可能发现按列调整ANALYZE 收集的统计信息的详细程度是值得的。在WHERE 子句中大量使用且数据分布高度不规则的列，可能需要比其他列更细粒度的数据直方图。见ALTER TABLE SET STATISTICS。

对大多数站点，推荐的做法是每天在使用率低的时段安排一次数据库范围的ANALYZE；这可以有效地与每夜的 VACUUM结合进行。不过，表统计信息变化相对缓慢的站点可能会发现这样做过了头，不那么频繁的ANALYZE 运行就足够了。

21.1.3. 防止事务 ID 回卷故障

PostgreSQL的 MVCC 事务语义依赖于能够比较事务 ID（XID）号：一个插入 XID 大于当前事务 XID 的行版本处于“未来”，对当前事务不可见。但由于事务 ID 的长度有限（撰写本文时为 32 位），一个运行很长时间（超过 40 亿个事务）的集群将遭遇事务 ID 回卷：XID 计数器回卷到零，突然之间，过去的事务看起来处于未来——这意味着它们的输出变得不可见。简而言之，灾难性的数据丢失。（实际上数据还在那里，但如果你无法访问它，这也于事无补。）

在PostgreSQL 7.2 之前，防御 XID 回卷的唯一方法是至少每 40 亿个事务重新initdb一次。这对高流量站点当然不太令人满意，因此设计出了更好的解决方案。
新方法允许服务器无限期保持运行，无需initdb或任何形式的重启。
代价是这条维护要求：数据库中的每个表必须至少每 10 亿个事务被清理一次。

在实践中这并不是一个苛刻的要求，
但由于不满足它的后果可能是完全的数据丢失（而不只是浪费磁盘空间或性能下降），
因此专门做了一些规定来帮助数据库管理员跟踪自上次VACUUM 以来的时间。本节余下部分给出细节。

新的 XID 比较方法区分出两个特殊的 XID，即编号 1 和 2（BootstrapXID和FrozenXID）。这两个 XID 总是被认为比每个普通 XID 都旧。普通 XID（大于 2 的那些）使用模 2^{31}算术进行比较。这意味着对每个普通 XID 来说，有 20 亿个“更旧”的 XID 和 20 亿个“更新”的 XID；换一种说法，普通 XID 空间是循环的，没有终点。因此，一旦一个行版本以某个普通 XID 创建，无论我们谈论的是哪个普通 XID，该行版本在接下来的 20 亿个事务中都会显得处于“过去”。如果该行版本在超过 20 亿个事务后仍然存在，它将突然显得处于未来。为了防止数据丢失，必须在旧的行版本达到 20 亿个事务这一时限之前的某个时刻，把它们的 XID 重新指定为FrozenXID。一旦被指定了这个特殊的 XID，无论回卷问题如何，它们对所有普通事务都会显得处于“过去”，因此这样的行版本在被删除之前一直有效，不管时间多长。这种 XID 的重新指定由VACUUM处理。

VACUUM的常规策略是把FrozenXID重新指定给任何普通 XID 已在过去中超过 10 亿个事务的行版本。这个策略保留了原始的插入 XID，直到它不再可能被关注。（事实上，大多数行版本可能终生都不会被“冻结”。）按照这个策略，任何表上两次VACUUM运行之间的最大安全间隔正好是 10 亿个事务：如果你等得更久，某个上次还不够旧、未被重新指定的行版本现在可能已超过 20 亿个事务并回卷到了未来——也就是说，对你而言丢失了。（当然，再过 20 亿个事务后它会重新出现，但那无济于事。）

由于前述原因本来就需要周期性地运行VACUUM，任何一个表长达 10 亿个事务未被清理的情况不太可能发生。但为了帮助管理员确保满足这个约束，VACUUM把事务 ID 统计信息存储在系统表pg_database中。特别地，在完成任何数据库范围的清理操作（即不指定具体表的VACUUM）时，会更新数据库的pg_database行的 datfrozenxid列。该字段中存储的值是那次 VACUUM 命令使用的冻结截止 XID。所有比该截止 XID 旧的普通 XID 都保证在该数据库中已被替换为FrozenXID。查看这些信息的一个便捷方法是执行查询

```
SELECT datname, age(datfrozenxid) FROM pg_database;
```

age列度量从截止 XID 到当前事务 XID 的事务数。

按照标准的冻结策略，一个刚清理过的数据库的age列将从 10 亿开始。当age接近 20 亿时，必须再次清理该数据库，以避免回卷故障的风险。推荐的做法是至少每 5 亿个事务清理每个数据库一次，以提供充足的安全余量。为了帮助满足这条规则，每次数据库范围的VACUUM都会在存在pg_database项显示age超过 15 亿个事务时自动发出警告，例如：

```
play=# VACUUM;
WARNING:  some databases have not been vacuumed in 1613770184
 transactions
HINT:  Better vacuum them within 533713463 transactions, or you may
 have a wraparound failure.
VACUUM
```

带FREEZE选项的VACUUM使用更激进的冻结策略：足够旧、
可被所有打开的事务视为有效的行版本都会被冻结。特别地，在一个完全空闲的数据库中执行 VACUUM FREEZE，可以保证该数据库中的所有行版本都被冻结。因此，只要该数据库不以任何方式被修改，就不需要后续的清理来避免事务 ID 回卷问题。这项技术被 initdb用来准备template0数据库。它也应当被用来准备任何要在pg_database中标记为 datallowconn

= false的用户创建数据库，因为你无法连接的数据库没有便捷的清理方法。注意，VACUUM关于未清理数据库的自动警告消息将忽略datallowconn = false的 pg_database项，以避免对这些数据库发出虚假警告；因此，确保这样的数据库被正确冻结是你的责任。

21.2. 例行重索引

在某些情况下，值得用REINDEX命令周期性地重建索引。（还有contrib/reindexdb 可以重建整个数据库的索引。）不过，与以前的版本相比，PostgreSQL 7.4 已大幅减少了对这项活动的需求。

21.3. 日志文件维护

把数据库服务器的日志输出保存到某个地方，而不是直接把它送到 /dev/null，是个好主意。到了诊断问题的时候，日志输出是无比珍贵的。然而，
日志输出往往很庞大（在较高的调试级别下尤其如此），你不会想无限期地保存它。
你需要“轮转”日志文件，以便不时地启动新的日志文件并丢弃旧的。

如果你只是简单地把postmaster的 stderr重定向到一个文件，那么截断日志文件的唯一方法就是停止并重启postmaster。这对开发环境可能没问题，但你不会想让生产服务器这样运行。

管理日志输出的最简单的生产级方法是把它们全部发送到 syslog，让syslog处理文件轮转。为此，在postgresql.conf中把配置参数 syslog设置为 2（只记录到syslog）。然后，每当你想强制syslog守护进程开始写入新的日志文件时，可以向它发送SIGHUP信号。
如果想自动化日志轮转，可以配置logrotate 程序来处理syslog的日志文件。

然而，在许多系统上，syslog并不很可靠，尤其是对于大的日志消息；
它可能恰在你最需要的时候截断或丢弃消息。你可能会发现，把postmaster的stderr 通过管道送给某种日志轮转程序更有用。如果你用 pg_ctl启动服务器，那么postmaster 的stderr已经被重定向到stdout了，所以你只需要一条管道命令：

```
pg_ctl start | logrotate
```

PostgreSQL发行版没有包含合适的日志轮转程序，但 Internet 上有很多可用的；例如，Apache 发行版中就包含一个。

第 22 章 备份与恢复

与任何包含有价值数据的事物一样，PostgreSQL 数据库也应定期备份。虽然这个过程基本上很简单，但理解其底层技术和前提假设的基本概念非常重要。

备份PostgreSQL数据有两种根本不同的方法：

- SQL 转储
- 文件系统级别的备份

22.1. SQL 转储

SQL 转储方法的思路是生成一个包含 SQL 命令的文本文件，当把这些命令送回服务器时，服务器会把数据库重建为转储时的状态。PostgreSQL为此提供了工具程序 pg_dump。该命令的基本用法是：

```
pg_dump dbname > outfile
```

如你所见，pg_dump把结果写到标准输出。我们将在下面看到这样做的用处。

pg_dump是一个常规的PostgreSQL 客户端应用（尽管是一个特别聪明的应用）。这意味着你可以从任何能访问数据库的远程主机上执行这个备份过程。但请记住，pg_dump并不会以特殊的权限运行。特别是，你必须对所有想要备份的表拥有读权限，因此在实践中你几乎总是必须是数据库超级用户。

要指定pg_dump应当连接哪个数据库服务器，可以使用命令行选项`-h host`和`-p port`。默认主机是本地主机，或者由`PGHOST`环境变量指定的主机。类似地，默认端口由`PGPORT`环境变量指示，若未设置，则使用编译时的默认值。（方便的是，服务器通常也有相同的编译时默认值。）

与任何其他PostgreSQL客户端应用一样，pg_dump默认以与当前操作系统用户名相同的数据库用户名连接。要覆盖这一行为，可以指定`-U`选项或设置环境变量`PGUSER`。请记住，pg_dump 的连接同样受常规的客户端认证机制约束（这些机制在第 19 章中描述）。

pg_dump创建的转储在内部是一致的，也就是说，pg_dump运行期间对数据库的更新不会出现在转储中。pg_dump工作时不会阻塞数据库上的其他操作。（例外是那些需要以排他锁运行的操作，例如`VACUUM FULL`。）

> **重要**
>
> 当你的数据库模式依赖于 OID（例如用作外键）时，你必须指示 pg_dump把 OID 也一并转储。为此，可以使用 `-o`命令行选项。“大对象”默认也不会被转储。如果你使用了大对象，请参阅pg_dump的命令参考页。

22.1.1. 恢复转储

pg_dump创建的文本文件应由psql 程序读入。恢复转储的一般命令形式是：

```
psql dbname < infile
```

其中*infile*就是你在pg_dump命令中用作*outfile* 的文件。数据库*dbname*不会由此命令创建，你必须在执行psql之前自己从template0创建它（例如用createdb -T template0 *dbname*）。psql支持与pg_dump类似的用于控制数据库服务器位置和用户名的选项。更多信息见其参考页。

如果原始数据库中的对象属于不同的用户，那么转储会指示 psql依次以每个受影响的用户连接，然后创建相应的对象。这样便保留了原来的属主关系。然而，这也意味着所有这些用户必须已经存在，而且你必须被允许以每个用户的身份连接。因此，可能有必要临时放宽客户端认证设置。

恢复完成后，在每个数据库上运行ANALYZE是明智的，这样优化器就能获得有用的统计信息。你也可以运行 vacuumdb -a -z来对所有数据库执行ANALYZE。

pg_dump和psql能够写入或读取管道，这使得把一个数据库从一台服务器直接转储到另一台服务器成为可能；例如：

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

重要

pg_dump产生的转储是相对于template0 的。这意味着任何添加到template1中的语言、过程等也会被pg_dump转储。因此，恢复时如果你使用了定制的template1，就必须像上面的例子那样从 template0创建空数据库。

提示

通过增大配置参数sort_mem可以提高恢复性能（见第 16.4.2.1 节）。

22.1.2. 使用pg_dumpall

上述机制在备份整个数据库集群时既繁琐又不合适。为此提供了 pg_dumpall程序。pg_dumpall 备份给定集群中的每个数据库，同时保留集群范围的数据，例如用户和组。pg_dumpall的调用序列很简单：

```
pg_dumpall > outfile
```

得到的转储可以用psql恢复：

```
psql template1 < infile
```

（实际上，你可以指定任何现有的数据库名作为起点，但如果你要在一个空集群中重新装载，那么template1 是唯一可用的选择。）恢复pg_dumpall 转储时总是需要数据库超级用户权限，因为恢复用户和组信息需要这样的权限。

22.1.3. 大型数据库

由于PostgreSQL允许表大于系统上的最大文件尺寸，把这样一个表转储到一个文件可能会成问题，因为得到的文件很可能大于系统允许的最大尺寸。由于 pg_dump可以写到标准输出，你可以直接使用标准 Unix 工具来绕过这个潜在问题。

使用压缩转储。. 你可以使用自己喜欢的压缩程序，例如 gzip。

```
pg_dump dbname | gzip > filename.gz
```

恢复时用：

```
createdb dbname
gunzip -c filename.gz | psql dbname
```

或者：

```
cat filename.gz | gunzip | psql dbname
```

使用split。. split命令允许你把输出分割成底层文件系统可接受大小的块。例如，要切分为 1 MB 的块：

```
pg_dump dbname | split -b 1m - filename
```

恢复时用：

```
createdb dbname
cat filename* | psql dbname
```

使用定制转储格式。. 如果PostgreSQL是在安装了zlib 压缩库的系统上构建的，定制转储格式会在把数据写入输出文件时压缩数据。对于大型数据库，这会产生与使用gzip 相当的转储尺寸，而且还有一个额外的优点：可以选择性地恢复部分表。
下面的命令用定制转储格式转储一个数据库：

```
pg_dump -Fc dbname > filename
```

详情见pg_dump和pg_restore的参考页。

22.1.4. 注意事项

pg_dump（相应地也包括pg_dumpall）有一些限制，
这些限制源于从系统目录中重建某些信息的困难。

具体来说，pg_dump写对象的顺序并不十分精巧。例如，当函数被用作列默认值时，
这可能导致问题。唯一的解决办法是手动对转储重新排序。如果你在模式中创建了循环依赖，
那么你要做的工作会更多。

出于向后兼容的原因，pg_dump默认不转储大对象。要转储大对象，你必须使用定制输出格式或 TAR 输出格式，并在 pg_dump中使用-b选项。详情见参考页。PostgreSQL源码树的 contrib/pg_dumplo目录中也包含一个可以转储大对象的程序。

请熟悉pg_dump参考页。

22.2. 文件系统级别的备份

另一种备份策略是直接复制PostgreSQL用来存储数据库中数据的文件。这些文件的位置在第 16.2 节中有说明，但如果你对这个方法感兴趣，可能早就已经找到它们了。你可以使用自己喜欢的任何常规文件系统备份方法，例如：

```
tar -cf backup.tar /usr/local/pgsql/data
```

然而，有两个限制使这种方法不实用，或者至少不如pg_dump 方法：

1. 为了得到可用的备份，数据库服务器必须被关闭。
   诸如禁止所有连接之类的折中办法是行不通的，因为总会有一些缓冲在进行。因此，
   也不建议信任那些声称支持“一致快照”的文件系统。关于停止服务器的信息可以在第 16. 6 节中找到。不用说，恢复数据之前你也需要关闭服务器。

2. 如果你深入研究过数据的文件系统布局的细节，
   你可能会想尝试只从各自的文件或目录中备份或恢复某些单独的表或数据库。这行不通，
   因为这些文件中包含的信息只是真相的一半。另一半在提交日志文件`pg_clog/*` 中，其中包含所有事务的提交状态。一个表文件只有配合这些信息才可用。当然，
   只恢复一个表及相关的`pg_clog` 数据也是不可能的，因为那会使数据库集群中的所有其他表都变得不可用。

另一种文件系统备份方法是：如果文件系统支持该功能，可以对数据目录制作“一致快照”。
这样的快照保存的数据库文件处于数据库服务器未被正常关闭的状态；因此，
当你在这个备份目录上启动数据库服务器时，它会认为服务器曾崩溃并重放 WAL 日志。这不是问题，只是要知道这一点。

注意，文件系统备份不一定比 SQL 转储小。相反，它很可能会更大。（例如，pg_dump不需要转储索引的内容，只需要转储重建它们的命令。）

22.3. 版本间的迁移

作为一般规则，PostgreSQL的主版本（第一个点号后的数字变化）
之间内部数据存储格式可能发生变化。
这不适用于同一主版本下的不同次版本（第二个点号后的数字变化）；
它们总是具有兼容的存储格式。例如，版本 7.0.1、7.1.2 和 7.2 互不兼容，而 7.1.1 和 7. 1.2 是兼容的。在兼容的版本之间升级时，你可以直接让新的可执行文件复用磁盘上的数据目录。
否则，你需要“备份”你的数据并把它“恢复”到新服务器上，使用 pg_dump。（有些检查可以防止你做错事，因此混淆这些事情不会造成损害。）具体的安装过程不是本节的主题；
这些细节见第 14 章。

把新服务器安装到另一个目录中，并让旧服务器和新服务器在不同的端口上并行运行，
可以实现最短的停机时间。然后你可以用类似下面的命令：

```
pg_dumpall -p 5432 | psql -d template1 -p 6543
```

来传输你的数据。如果愿意，也可以使用中间文件。然后你可以关闭旧服务器，
并在旧服务器运行的端口上启动新服务器。你应当确保在运行pg_
dumpall之后数据库没有被更新，否则你显然会丢失那些数据。关于如何禁止访问，见第 19 章。
在实践中，切换前你可能想先在新的配置上测试你的客户端应用。

如果无法或不想并行运行两个服务器，可以在安装新版本之前先执行备份步骤，
然后关闭旧服务器、把旧版本移开、安装新版本、启动新服务器并恢复数据。例如：

```
pg_dumpall > backup
pg_ctl stop
mv /usr/local/pgsql /usr/local/pgsql.old
cd ~/postgresql-7.4.30
gmake install
initdb -D /usr/local/pgsql/data
postmaster -D /usr/local/pgsql/data
psql template1 < backup
```

有关启动和停止服务器的方式及其他细节，见第 16 章。安装说明会提示执行这些步骤的合适位置。

注意

当你“把旧安装移开”时，它就不再完全可用了。
安装的某些部分包含其他部分所在位置的信息。这通常不是大问题，
但如果你计划让两个安装并行使用一段时间，
应当在构建时就给它们分配不同的安装目录。

第 23 章 监控数据库活动

数据库管理员常常会问："系统现在正在做什么？"本章将讨论如何回答这个问题。

有多种工具可用于监控数据库活动和分析性能。本章的大部分内容用于描述 PostgreSQL 的统计收集器，但不应忽视常规的 Unix 监控程序，如 `ps` 和 `top`。此外，一旦识别出性能不佳的查询，可能还需要用 PostgreSQL 的 `EXPLAIN` 命令做进一步调查。第 13.1 节讨论了 `EXPLAIN` 以及理解单个查询行为的其他方法。

23.1. 标准 Unix 工具

在大多数平台上，PostgreSQL会修改由`ps`报告的命令标题，以便轻松识别各个服务器进程。
以下是一个显示示例：

```
$ ps auxww | grep ^postgres
postgres   960  0.0  1.1  6104 1480 pts/1    SN   13:17   0:00
 postgres -i
postgres   963  0.0  1.1  7084 1472 pts/1    SN   13:17   0:00
 postgres: stats buffer process
postgres   965  0.0  1.1  6152 1512 pts/1    SN   13:17   0:00
 postgres: stats collector process
postgres   998  0.0  2.3  6532 2992 pts/1    SN   13:18   0:00
 postgres: tgl runbug 127.0.0.1 idle
postgres  1003  0.0  2.4  6532 3128 pts/1    SN   13:19   0:00
 postgres: tgl regression [local] SELECT waiting
postgres  1016  0.1  2.4  6532 3080 pts/1    SN   13:19   0:00
 postgres: tgl regression [local] idle in transaction
```

（`ps`的适当调用方式随平台而异，显示内容的细节也是如此。此示例来自较新的 Linux 系统。）
这里列出的第一个进程是postmaster，即主服务器进程。
显示的命令参数与启动该进程时使用的参数相同。接下来的两个进程实现统计收集器，
下一节将详细描述。（如果你已把系统设置为不启动统计收集器，这两个进程将不存在。）
其余每个进程都是处理一个客户端连接的服务器进程。
每个这样的进程都将其命令行显示设置为以下形式：

```
postgres: user database host activity
```

在客户端连接的整个生命周期内，用户、数据库和连接来源主机这几项保持不变，
但活动指示符会改变。活动可以是`idle`（即等待客户端命令）、`idle in transaction`（在`BEGIN`块内等待客户端），或者命令类型名称，例如`SELECT`。此外，
服务器进程当前若正在等待另一服务器进程持有的锁，会附上`waiting`。在上面的示例中，
可以推断进程 1003 正在等待进程 1016 完成事务，从而释放某个锁。

> **提示**
>
> Solaris需要特殊处理。您必须使用`/usr/ucb/ps`，而不是`/bin/ps`。
> 您还必须使用两个`w`标志，而不仅仅是一个。此外，您对`postmaster`
> 命令的原始调用必须具有比每个服务器进程提供的更短的`ps`状态显示。
> 如果未满足这三个条件，每个服务器进程的`ps`输出都会显示原始的`postmaster`命令行。

23.2. 统计收集器

PostgreSQL的统计收集器是一个支持收集和报告服务器活动信息的子系统。目前，收集器可以按磁盘块和单行两种方式统计对表和索引的访问。它还支持确定其他服务器进程当前正在执行的确切命令。

23.2.1. 统计信息收集配置

由于收集统计信息会增加查询执行的开销，因此可以配置系统是否收集信息。这由通常在 postgresql.conf中设置的配置参数控制（有关设置配置参数的详细信息，请参阅第 16.4 节）。

要让统计收集器能够启动，必须把参数 stats_start_collector 设为 true。这是默认且被推荐的设置，但如果你对统计信息不感兴趣并想榨干每一点开销，也可以把它关掉。（不过节省的开销可能很小。）注意这个选项在服务器运行期间不能更改。

参数 stats_command_string、stats_block_level 和 stats_row_level 控制实际发送给收集器的信息量，从而决定运行时开销的大小。它们分别决定服务器进程是否把自己的当前命令串、磁盘块级访问统计和行级访问统计发送给收集器。通常这些参数在 postgresql.conf 中设置，从而作用于所有服务器进程，但也可以在单个会话中用 SET 命令打开或关闭它们。（为防止普通用户向管理员隐藏自己的活动，只有超级用户才被允许用 SET 更改这些参数。）

> **注意**
>
> 由于参数stats_command_string、stats_block_level和 stats_row_level默认为false，默认配置下收集的统计信息很少。启用这些配置变量中的一个或多个，都会显著增加统计收集器产生的有用数据量，但代价是额外的运行时开销。

23.2.2. 查看收集到的统计信息

有若干预定义视图可用来显示统计收集的结果，见表 23.1。另外，也可以用底层的统计函数构建自定义视图。

用统计信息监控当前活动时，重要的一点是要认识到这些信息不会即时更新。每个单独的服务器进程在等待下一条客户端命令之前传输新的访问计数；因此仍在进行中的查询不会影响显示的总量。此外，收集器本身每最多 pgstat_stat_interval 毫秒（默认 500）才发出一次新的总量。因此显示的总量落后于实际活动。

另一个要点是，当服务器进程被要求显示这些统计信息中的任何一种时，它会先取回收集器进程发出的最新总量，然后在当前事务结束之前继续使用这一快照用于所有统计视图和函数。因此，只要继续当前事务，统计信息看起来就不会变化。这是一个特性而不是缺陷，因为它允许你对统计信息执行多个查询并关联结果，而不必担心数字在你手下变化。但如果想让每个查询都看到新结果，请务必在任何事务块之外执行查询。

表 23.1. 标准统计视图

视图名	描述
pg_stat_activity	每个服务器进程一行，显示进程ID、数据库、用户、当前查询，以及当前查询开始执行的时间。报告当

视图名	描述
	前查询数据的各列只有在参数stats_command_string被打开时才可用。此外，除非查看该视图的用户是超级用户或与被报告进程的属主是同一用户，这些列读取为空值。（注意，由于收集器的报告延迟，当前查询只对长时间运行的查询是最新的。）
pg_stat_database	每个数据库一行，显示活动后端服务器进程数、该数据库中提交和回滚的事务总数、读取的磁盘块总数，以及缓冲区命中总数（即因在缓冲缓存中找到块而避免的块读请求数）。
pg_stat_all_tables	当前数据库中的每个表一行，显示顺序扫描和索引扫描的总次数、每种扫描返回的行总数，以及行插入、更新和删除的总数。
pg_stat_sys_tables	与pg_stat_all_tables相同，但只显示系统表。
pg_stat_user_tables	与pg_stat_all_tables相同，但只显示用户表。
pg_stat_all_indexes	当前数据库中的每个索引一行，显示使用该索引的索引扫描总次数、读取的索引行数，以及成功取出的堆行数。（当存在指向过期堆行的索引条目时，后者可能会更少。）
pg_stat_sys_indexes	和pg_stat_all_indexes一样，但只显示系统表上的索引。
pg_stat_user_indexes	和pg_stat_all_indexes一样，但只显示用户表上的索引。
pg_statio_all_tables	当前数据库中的每个表一行，显示从该表读取的磁盘块总数、缓冲区命中数、该表所有索引的磁盘块读取数和缓冲区命中数、该表辅助 TOAST 表（若有）的磁盘块读取数和缓冲区命中数，以及 TOAST 表索引的磁盘块读取数和缓冲区命中数。
pg_statio_sys_tables	与pg_statio_all_tables相同，但只显示系统表。
pg_statio_user_tables	与pg_statio_all_tables相同，但只显示用户表。
pg_statio_all_indexes	当前数据库中的每个索引一行，显示该索引中的磁盘块读取数和缓冲区命中数。
pg_statio_sys_indexes	和pg_statio_all_indexes一样，但只显示系统表上的索引。
pg_statio_user_indexes	和pg_statio_all_indexes一样，但只显示用户表上的索引。
pg_statio_all_sequences	当前数据库中的每个序列对象一行，显示该序列中的磁盘块读取数和缓冲区命中数。
pg_statio_sys_sequences	和pg_statio_all_sequences一样，但只显示系统序列。（目前没有定义任何系统序列，因此该视图总是空的。）
pg_statio_user_sequences	和pg_statio_all_sequences一样，但只显示用户序列。

每个索引的统计信息对于判断哪些索引正在被使用以及它们有多有效尤其有用。

pg_statio_视图主要用于判断缓冲区缓存的有效性。
如果实际磁盘读取次数远少于缓冲区命中次数，缓存就能满足大多数读取请求，
而无需调用内核。不过，这些统计信息并不能反映全部情况：由于PostgreSQL处理磁盘 I/O 的方式，不在PostgreSQL缓冲区缓存中的数据仍可能驻留在内核的 I/O 缓存中，因而仍可在不执行物

理读取的情况下取得。对于希望更详细地了解PostgreSQL I/O 行为的用户，建议将PostgreSQL统计收集器与能够深入查看内核 I/O 处理情况的操作系统工具结合使用。

也可以通过编写查询来建立查看统计信息的其他方式，
这些查询使用与标准视图相同的底层统计访问函数。这些函数列于表 23.2中。每数据库访问函数以数据库 OID 为参数来标识要报告的数据库。每表和每索引函数以表或索引 OID 为参数。（注意，用这些函数只能看到当前数据库中的表和索引。）每后端进程访问函数以一个后端进程 ID 为参数，其范围从 1 到当前活动的后端进程数。

表 23.2. 统计访问函数

函数	返回类型	描述
pg_stat_get_db_numbackends(oid)	integer	数据库的活动后端进程数
pg_stat_get_db_xact_commit(oid)	bigint	数据库中提交的事务
pg_stat_get_db_xact_rollback(oid)	bigint	数据库中回滚的事务
pg_stat_get_db_blocks_fetched(oid)	bigint	数据库的磁盘块读取请求数
pg_stat_get_db_blocks_hit(oid)	bigint	数据库在缓存中找到的磁盘块读取请求数
pg_stat_get_numscans(oid)	bigint	参数为表时是完成的顺序扫描数，参数为索引时是完成的索引扫描数
pg_stat_get_tuples_returned(oid)	bigint	参数为表时是顺序扫描读取的行数，参数为索引时是读取的索引行数
pg_stat_get_tuples_fetched(oid)	bigint	参数为表时是顺序扫描取出的有效（未过期）表行数，参数为索引时是使用该索引的索引扫描取出的表行数
pg_stat_get_tuples_inserted(oid)	bigint	插入表中的行数
pg_stat_get_tuples_updated(oid)	bigint	表中更新的行数
pg_stat_get_tuples_deleted(oid)	bigint	从表中删除的行数
pg_stat_get_blocks_fetched(oid)	bigint	表或索引的磁盘块读取请求数
pg_stat_get_blocks_hit(oid)	bigint	表或索引在缓存中找到的磁盘块请求数
pg_stat_get_backend_idset()	set of integer	当前活动的后端进程 ID 的集合（从 1 到活动的后端进程数）。用法示例见正文。
pg_backend_pid()	integer	附属于当前会话的后端进程的进程 ID
pg_stat_get_backend_pid(integer)	integer	给定后端进程的进程 ID
pg_stat_get_backend_dbid(integer)	oid	给定后端进程的数据库 ID

函数	返回类型	描述
pg_stat_get_backend_userid(integer)	oid	给定后端进程的用户 ID
pg_stat_get_backend_activity(integer)	text	给定后端进程的活动命令（如果当前用户既不是超级用户也不是被查询会话的用户，或者stats_command_string未打开，则为空值）
pg_stat_get_backend_activity_start(integer)	timestamp with time zone（带时区的时间戳）	给定后端进程当前执行的查询的启动时间（如果当前用户既不是超级用户也不是被查询会话的用户，或者stats_command_string未打开，则为空值）
pg_stat_reset()	boolean	重置当前收集到的所有统计信息

注意

pg_stat_get_db_blocks_fetched减去pg_stat_get_db_blocks_hit
得到的是为该表、索引或数据库发出的内核read()调用数；
但实际物理读取数通常由于内核级缓冲而更低。

函数pg_stat_get_backend_idset提供了一种为每个活动后端进程生成一行的便捷方法。
例如，要显示所有后端进程的PID和当前查询：

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

23.3. 查看锁

监控数据库活动的另外一个有用的工具是pg_locks系统表。
这样就允许数据库管理员查看锁管理器中当前存在的锁的信息。例如，这个功能可以被用于：

- 查看当前存在的所有锁、在一个特定数据库中的关系上所有的锁、
  在一个特定关系上所有的锁，或者由一个特定PostgreSQL会话持有的所有的锁。
- 判断当前数据库中带有最多未授予锁的关系（它可能是数据库客户端之间的竞争来源）。
- 判断锁竞争给数据库总体性能带来的影响，以及锁竞争随着整个数据库流量的变化范围。

pg_locks视图的细节在第 43.32 节中。更多有关PostgreSQL的锁和管理并发性的信息，请参考第 12 章。

第 24 章 监控磁盘使用情况

本章讨论如何监控 PostgreSQL 数据库系统的磁盘使用。在当前版本中，数据库管理员对磁盘上的存储布局没有太多控制权，因此本章大多是介绍性的，
可以给你一些用操作系统工具管理磁盘使用的思路。

24.1. 确定磁盘使用情况

每个表有一个主堆磁盘文件，大部分数据存储在其中。为了存储较长的列值，
还有一个与表相关联的 TOAST 文件，基于表的 OID 命名（实际上是 `pg_class.relfilenode`），以及 TOAST 表上的一个索引。也可能有与基础表相关联的索引。

你可以从三个地方监控磁盘空间：在 psql 中使用 `VACUUM` 信息、在 psql 中使用 `contrib/dbsize` 中的工具，以及在命令行上使用 `contrib/oid2name` 中的工具。在一个刚做过 VACUUM 或 ANALYZE 的数据库上使用 psql，你可以发出查询来查看任何表的磁盘使用：

```
SELECT relfilenode, relpages FROM pg_class WHERE relname = 'customer';

 relfilenode | relpages
-------------+----------
       16806 |       60
(1 row)
```

每页通常是 8 KB。（记住，`relpages` 只由 `VACUUM` 和 `ANALYZE` 更新。）

要显示 TOAST 表使用的空间，使用下面这样的查询，把堆的 `relfilenode` 编号（由上面的查询确定）代入：

```
SELECT relname, relpages
    FROM pg_class
    WHERE relname = 'pg_toast_16806' OR relname = 'pg_toast_16806_
index'
    ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

也可以轻松显示索引大小：

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname        | relpages
----------------------+----------
 customer_id_indexdex |       26
```

利用这些信息，你可以很容易地找出最大的表和索引：

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

contrib/dbsize 向你的数据库装载一些函数，使你能够在 psql 内部查出表或数据库的大小，而无须 VACUUM 或 ANALYZE。

你还可以使用 contrib/oid2name 来显示磁盘使用情况。示例见该目录中的README.oid2name。其中包含一个显示每个数据库磁盘使用情况的脚本。

24.2. 磁盘已满故障

数据库管理员最重要的磁盘监控任务是确保磁盘不会被填满。
被填满的数据磁盘可能导致数据库索引随后损坏，但不会损坏表本身。如果 WAL 文件在同一个磁盘上（默认配置就是如此），那么数据库初始化期间磁盘被填满可能导致 WAL 文件损坏或不完整。这种故障条件会被检测到，数据库服务器将拒绝启动。

如果无法通过删除其他东西在磁盘上腾出更多空间，
你可以把一些数据库文件移动到其他文件系统并在原位置创建一个符号链接。但注意 pg_dump 无法保存这种安排的位置布局信息；恢复时会把所有东西放回同一个地方。
为了避免磁盘空间耗尽，你可以在创建 WAL 文件或单个数据库时把它们放在其他位置。更多信息见 initdb 文档和第 18.5 节。

> **提示**
>
> 有些文件系统在几乎满时性能很差，所以不要等到磁盘满了才采取行动。

第 25 章 预写式日志（WAL）

预写式日志（WAL）是一种标准的事务日志方法。它的详细描述可以在大多数（即使不是全部）关于事务处理的书中找到。简言之，WAL 的核心概念是：对数据文件（表和索引所在之处）的更改必须只在这些更改已被记录之后写入，也就是在日志记录已刷写到永久存储之后。
如果遵循这一过程，就不需要在每次事务提交时把数据页刷写到磁盘，
因为我们知道一旦发生崩溃，
可以用日志恢复数据库：尚未应用到数据页的任何更改将先从日志记录中重做（这是前滚恢复，也称 REDO），然后由未提交事务所做的更改将从数据页中移除（后滚恢复，UNDO）。

25.1. WAL 的优势

使用 WAL 的第一个明显好处是大幅减少磁盘写入次数，因为在事务提交时只需要把日志文件刷写到磁盘；在多用户环境中，许多事务的提交可以通过对日志文件的一次 `fsync()` 完成。此外，日志文件是顺序写入的，因此同步日志的代价远小于刷写数据页的代价。

下一个好处是数据页的一致性。事实是，在 WAL 之前，PostgreSQL 从不能保证崩溃后的一致性。在 WAL 之前，写入期间的任何崩溃都可能导致：

1. 索引行指向不存在的表行

2. 索引行在分裂操作中丢失

3. 由于数据页部分写入，表或索引页内容完全损坏

索引的问题（问题 1 和 2）或许可以通过额外的 `fsync()` 调用修复，但如果没有 WAL，如何处理最后一种情况并不明显；WAL 在需要时会把整个数据页内容保存在日志中，以确保崩溃后恢复的页面一致性。

25.2. 未来的收益

UNDO 操作尚未实现。这意味着由中止的事务所做的更改仍会占用磁盘空间，而且仍需要一个保存事务状态的永久 `pg_clog` 文件，因为事务标识符不能重用。一旦实现了 UNDO，`pg_clog` 就不再需要是永久的；可以在关机时删除 `pg_clog`。（不过，自从对 `pg_clog` 采用分段存储方法以来，这个问题的紧迫性已大大降低：不再需要永远保留旧的 `pg_clog` 条目。）

有了 UNDO，还可以实现保存点，允许部分回滚无效的事务操作（命令输错导致的解析错误、插入重复的主键/唯一键等），同时能够继续或提交事务在出错之前所做的有效操作。目前，
任何错误都会使整个事务无效并要求中止事务。

WAL 为数据库在线备份和恢复（BAR）的新方法提供了机会。要使用这种方法，需要定期把数据文件保存到另一块磁盘、磁带或另一台主机，并归档 WAL 日志文件。数据库文件副本和归档的日志文件可以像崩溃后恢复那样用于恢复。每次制作新的数据库文件副本后，
旧的日志文件就可以删除。实现这一功能需要记录数据文件和索引的创建与删除；
还需要开发一种复制数据文件的方法（操作系统的复制命令并不合适）。

实现这些收益的一个障碍是它们需要在相当长的时间内保存 WAL 条目（例如，如果想要事务 UNDO，则需要保存最长可能事务那么久）。目前的 WAL 格式极其庞大，因为它包含许多磁盘页快照。目前这不算严重问题，因为条目只需保存一两个检查点间隔；但要获得这些未来的收益，将需要某种压缩的 WAL 格式。

25.3. WAL 配置

有若干影响数据库性能的与 WAL 相关的配置参数。本节说明它们的用法。设置配置参数的详情请查阅第 16.4 节。

检查点是事务序列中保证数据文件已用检查点之前记录的所有信息更新过的点。在检查点时刻，所有脏数据页被刷写到磁盘，并向日志文件写入一条特殊的检查点记录。结果是，
一旦发生崩溃，恢复者就知道应从日志中的哪条记录（称为重做记录）开始 REDO 操作，因为在该记录之前对数据文件所做的任何更改都已在磁盘上。做完检查点之后，
重做记录之前写入的任何日志段都不再需要并可被回收或删除。（当实现了基于 WAL 的 BAR 时，日志段在回收或删除之前会被归档。）

服务器每隔一段时间派生一个专门的进程来创建下一个检查点。每 `checkpoint_segments` 个日志段或每 `checkpoint_timeout` 秒（以先到者为准）创建一个检查点。默认设置分别是 3 段和 300 秒。也可以用 SQL 命令 `CHECKPOINT` 强制检查点。

减小 `checkpoint_segments` 和/或 `checkpoint_timeout` 会使检查点更频繁。这可以让崩溃后恢复更快（因为需要重做的工作更少）。
但必须把这与更频繁地刷写脏数据页的增加的成本相权衡。此外，为确保数据页一致性，
每个检查点后对数据页的第一次修改会导致记录整个页面内容。
因此更小的检查点间隔会增加日志输出量，部分抵消了使用更小间隔的目的，
而且无论如何都会造成更多磁盘 I/O。

至少会有一个 16 MB 的段文件，通常不会超过 2 * `checkpoint_segments` + 1 个文件。可以用这个来估算 WAL 的空间需求。通常，当旧的日志段文件不再需要时，它们会被回收（改名为编号序列中的下几个段）。如果由于日志输出率的短期峰值，段文件超过了 2 * `checkpoint_segments` + 1 个，不需要的段文件将被删除而不是回收，直到系统回到该限制之内。

有两个常用的 WAL 函数：`LogInsert` 和 `LogFlush`。`LogInsert` 用于把一条新记录放入共享内存中的 WAL 缓冲区。如果没有空间放新记录，`LogInsert` 就不得不写出（移入内核缓存）几个已满的 WAL 缓冲区。这并不理想，因为 `LogInsert` 在每次数据库低层修改（例如行插入）时都会被使用，而那时受影响的数据页上正持有排他锁，因此该操作需要尽可能快。更糟的是，写出 WAL 缓冲区还可能迫使创建新的日志段，这需要更多时间。正常情况下，WAL 缓冲区应当由 `LogFlush` 请求写出并刷写，该请求大部分在事务提交时发出，以确保事务记录被刷写到永久存储。在日志输出很高的系统上，`LogFlush` 请求可能不够频繁，无法阻止 WAL 缓冲区被 `LogInsert` 写出。在这样的系统上，应当通过修改配置参数 `wal_buffers` 来增加 WAL 缓冲区的数量。WAL 缓冲区的默认数量是 8。增大这个值将相应地增加共享内存的使用。

检查点的代价相当高，因为它用操作系统的 `sync()` 调用强制把所有脏内核缓冲区写回磁盘。繁忙的服务器可能过快地填满检查点段文件，导致过于频繁的检查点。
如果这种被迫的检查点发生得比 `checkpoint_warning` 秒更频繁，就会向服务器日志输出一条消息，建议增大 `checkpoint_segments`。

`commit_delay` 参数定义服务器进程在用 `LogInsert` 把提交记录写入日志之后、执行 `LogFlush` 之前休眠多少微秒。这个延迟允许其他服务器进程把它们的提交记录加入日志，从而让所有这些记录用一次日志同步刷写。如果未启用 `fsync`，或者当前处于活跃事务中的其他会话少于 `commit_siblings` 个，就不会休眠；这避免了在不太可能有其他会话很快提交时休眠。注意在大多数平台上，休眠请求的分辨率是十毫秒，因此 1 到 10000 微秒之间的任何非零 `commit_delay` 设置效果都相同。这些参数的理想值尚不清楚；鼓励进行实验。

`wal_sync_method` 参数决定 PostgreSQL 如何要求内核把 WAL 更新强制写到磁盘。就可靠性而言所有选项都相同，但哪一个最快则非常依赖于平台。注意如果关闭了 `fsync`，这个参数就无关紧要。

把 wal_debug 参数设为任何非零值将导致每次 LogInsert 和 LogFlush 的 WAL 调用都被记录到服务器日志。目前，非零值具体是什么没有区别。这个选项将来可能被更通用的机制取代。

25.4. 内部机制

WAL 是自动启用的；管理员无需任何操作，只需确保满足 WAL 日志的额外磁盘空间需求，并完成必要的调优（见第 25.3 节）。

WAL 日志存储在数据目录下的 pg_xlog 目录中，作为一组段文件，每个大小为 16 MB。每个段划分为 8 kB 的页。日志记录头在 access/xlog.h 中描述；记录内容取决于所记录事件的类型。段文件以递增的数字命名，从 0000000000000000 开始。目前数字不会回绕，但用尽可用数字也需要非常长的时间。

WAL 缓冲区和控制结构位于共享内存中，由服务器子进程处理；它们受轻量级锁保护。对共享内存的需求取决于缓冲区的数量。WAL 缓冲区的默认大小是 8 个缓冲区、每个 8 kB，共 64 kB。

如果能将 WAL 放在与主数据库文件不同的磁盘上，会更有利。这可以通过把 pg_xlog 目录移动到其他位置（当然要在服务器关闭时进行），然后在主数据目录中的原位置创建一个指向新位置的符号链接来实现。

WAL 的目标——确保日志在数据库记录被更改之前写入——可能被向内核虚假报告写入成功的磁盘驱动器破坏，而实际上它们只是缓存了数据而尚未存储到磁盘上。
这种情况下断电仍可能导致不可恢复的数据损坏。管理员应尽量确保存放 PostgreSQL WAL 日志文件的磁盘不做这样的虚假报告。

在完成检查点并刷写 WAL 之后，该检查点的位置会保存在文件 pg_control 中。因此，在恢复开始时，服务器会先读取 pg_control，再读取检查点记录；然后从检查点记录给出的 WAL 位置向前扫描并执行 REDO。由于检查点之后对数据页的第一次修改，会把整页内容保存在 WAL 中，因而自该检查点以来被修改的所有页面都会恢复到一致状态。

用 pg_control 获取检查点位置可以加快恢复过程，但为了处理 pg_control 可能的损坏，我们其实应该实现以相反顺序——从最新到最旧——读取现有日志段来找到最后一个检查点。这一点尚未实现。

第 26 章 回归测试

回归测试是针对PostgreSQL中 SQL 实现的一套全面测试。它们既测试标准 SQL 操作，也测试PostgreSQL的扩展能力。从 PostgreSQL 6.1 起，回归测试对每个正式发行版都是最新的。

26.1. 运行测试

回归测试既可以针对一个已安装并正在运行的服务器运行，
也可以使用构建树中的一个临时安装来运行。此外，还有“并行”和“顺序”
两种运行测试的模式。顺序方法依次运行每个测试脚本，
而并行方法会启动多个服务器进程来并行运行成组的测试。
并行测试可以让人确信进程间通信和锁机制工作正常。由于历史原因，
顺序测试通常针对现有安装运行，而并行方法针对临时安装运行，但这并没有技术上的原因。

要在构建之后、安装之前运行回归测试，在顶层目录中键入

```
gmake check
```

（或者你可以切换到 `src/test/regress` 并在那里运行该命令。）这会先构建若干辅助文件，例如一些示例用户定义触发器函数，然后运行测试驱动脚本。最后你应当看到类似

```
=======================
 All 93 tests passed.
=======================
```

的输出，或者一条关于哪些测试失败的说明。更多信息见下文第 26.2 节。

由于这种测试方法运行的是一个临时服务器，当你以 root 用户身份操作时它将无法工作（因为服务器不会以 root 启动）。如果你已经以 root 完成了构建，不必从头再来。只需让回归测试目录可被其他用户写入，以该用户登录，然后重新启动测试。例如

```
root# chmod -R a+w src/test/regressroot# chmod -R a+w contrib/
spiroot# su - joeuserjoeuser$ cd top-level build directoryjoeuser
$ gmake check
```

（这里唯一可能的“安全风险”是其他用户可能背着你篡改回归测试结果。
管理用户权限时请运用常识。）

或者，在安装之后运行测试。

并行回归测试会以你的用户 ID 启动相当多的进程。目前最大并发度是二十个并行测试脚本，这意味着六十个进程：每个测试脚本有一个服务器进程、一个 psql，并且通常还有一个 psql 的 shell 父进程。因此，如果你的系统对每个用户的进程数有限制，请确保该限制至少在七十五左右，否则并行测试中可能出现看似随机的失败。如果你无法提高该限制，可以通过设置 `MAX_CONNECTIONS` 参数来降低并行度。例如

```
gmake MAX_CONNECTIONS=10 check
```

最多同时运行十个测试。

在某些系统上，默认的 Bourne 兼容 shell（`/bin/sh`）在要并行管理太多子进程时会出问题。这可能导致并行测试运行锁死或失败。在这种情况下，在命令行上指定一个不同的 Bourne 兼容 shell，例如：

```
gmake SHELL=/bin/ksh check
```

如果没有可用的无缺陷 shell，你也可以像上面那样通过限制连接数来规避该问题。

要在安装之后运行测试（见第 14 章），初始化一个数据区域并启动服务器，如第 16 章所述，然后键入

```
gmake installcheck
```

除非 `PGHOST` 和 `PGPORT` 环境变量另有指示，测试将期望在本地主机和默认端口号上连接服务器。

26.2. 测试结果评估

某些安装正确且功能完备的PostgreSQL实例，
可能会因为平台特有因素而在部分回归测试中“失败”，例如浮点表示和时区支持的差异。
目前这些测试是通过把输出与参考系统生成的输出做简单的`diff`比较来评估的，
因此结果会对细微的系统差异很敏感。报告某项测试“失败”时，
请务必检查预期结果与实际结果之间的差异；你可能会发现这些差异并不重要。尽管如此，
我们仍努力在所有受支持平台上维护准确的参考文件，因此原则上应期待所有测试都能通过。

回归测试的实际输出位于`src/test/regress/results`目录中的文件里。测试脚本使用`diff`把每个输出文件与存储在`src/test/regress/expected`目录中的参考输出进行比较。
任何差异都会保存到`src/test/regress/regression.diffs`中供你检查。（如果愿意，
也可以自己运行`diff`。）

26.2.1. 错误消息差异

某些回归测试包含故意构造的非法输入值。错误消息可能来自 PostgreSQL代码，也可能来自宿主平台的系统例程。在后一种情况下，消息会因平台而异，但应反映相近的信息。
这类消息差异会导致回归测试显示为“失败”，但可以通过检查确认其有效性。

26.2.2. Locale 差异

如果你在一个使用 C 之外的排序规则 locale 初始化的已安装服务器上运行测试，那么可能会因排序顺序而出现差异以及后续失败。回归测试套件通过提供备选结果文件来处理这个问题，
这些文件合起来已知能处理大量 locale。例如，对 `char` 测试，期望文件 `char.out` 处理 `C` 和 `POSIX` locale，而文件 `char_1.out` 处理许多其他 locale。回归测试驱动程序会在检查成功和计算失败差异时自动挑选最匹配的文件。（这意味着回归测试无法判断结果是否适合所配置的 locale。测试只是简单地挑选一个效果最好的结果文件。）

如果由于某种原因现有的期望文件没有覆盖某个 locale，你可以添加一个新文件。命名方案是 *`testname_digit`*`.out`。具体的数字并不重要。请记住，回归测试驱动程序会认为所有这样的文件都是同等有效的测试结果。如果测试结果是平台相关的，应当改用第 26.3 节中描述的技术。

26.2.3. 日期和时间差异

如果你在一个夏令时切换日或其次日运行 `horology` 测试，其中少数查询会失败。这些查询期望昨天午夜、今天午夜和明天午夜之间的间隔正好是二十四小时——如果其间夏令时生效或失效，
这就是错误的。

> **注意**
>
> 由于使用的是美国的夏令时规则，无论你所在地的夏令时何时生效，这个问题总是出现在 4 月的第一个星期日、10 月的最后一个星期日及其后的星期一。还要注意，问题是在太平洋时间（UTC-7 或 UTC-8）的午夜出现或消失，而不是你本地时间的午夜。因此，根据你所在的位置，失败可能出现在星期六深夜或一直持续到星期二的大部分时间。

大多数日期和时间结果依赖于时区环境。参考文件是为时区 `PST8PDT`（加利福尼亚州伯克利）生成的，如果测试不使用该时区设置运行，就会出现明显的失败。回归测试驱动程序会把环境变量 `PGTZ` 设置为 `PST8PDT`，这通常能确保正确的结果。但是，你的操作系统必须提供对 `PST8PDT` 时区的支持，否则依赖时区的测试将会失败。要验证你的机器确实支持这一点，请键入：

```
env TZ=PST8PDT date
```

上面的命令应当以 `PST8PDT` 时区返回当前系统时间。如果 `PST8PDT` 时区不可用，你的系统可能返回的是 UTC 时间。如果缺少 `PST8PDT` 时区，你可以显式设置时区规则：

```
PGTZ='PST8PDT7,M04.01.0,M10.05.03'; export PGTZ
```

有些系统似乎不接受显式设置本地时区规则的推荐语法；在这类机器上你可能需要使用不同的 `PGTZ` 设置。

一些使用较老时区库的系统无法对 1970 年之前的日期应用夏令时修正，导致 1970 年前的 PDT 时间显示为 PST。这会导致测试结果中出现局部差异。

26.2.4. 浮点差异

某些测试涉及从表列计算 64 位浮点数（`double precision`）。已经观察到涉及`double precision`列数学函数的结果存在差异。`float8`和`geometry`测试尤其容易在不同平台之间、甚至在不同编译器优化设置下出现微小差异。
需要人工目测比较来确定这些差异的真实意义，它们通常出现在小数点右侧第 10 位。

某些系统把负零显示为`-0`，而另一些只显示 `0`。

某些系统对`pow()`和`exp()` 发出错误的方式，与当前PostgreSQL代码所预期的机制不同。

26.2.5. 行顺序差异

你可能会看到这样的差异：同样的行在输出中的顺序与预期文件中的顺序不同。
在大多数情况下，严格说来这并不是缺陷。大多数回归测试脚本并没有细致到为每一个`SELECT`都使用`ORDER BY`，因此按 SQL 规范，它们的结果行顺序并没有良好定义。实际上，由于我们看到的是同一软件在同一数据上执行相同查询，通常在所有平台上都会得到相同的结果顺序，
所以缺少`ORDER BY`并不是问题。不过，有些查询确实会表现出跨平台的顺序差异。（非 C 区域设置也可能触发顺序差异。）

因此，如果你看到顺序差异，一般无需担心，除非查询确实包含 `ORDER BY`而你的结果违反了它。不过，仍请报告该问题，这样我们可以为那个特定查询加上`ORDER BY`，以在后续版本中消除这种虚假的“失败”。

你可能会好奇，为什么我们不显式地为所有回归测试查询排序，从而一劳永逸地解决这个问题。
原因在于，那样反而会降低回归测试的价值，
因为测试会倾向于覆盖能产生有序结果的查询计划类型，
而排除那些不能产生有序结果的计划类型。

26.2.6. “random” 测试

random 测试脚本中至少有一种情况旨在产生随机结果。这会使 random 回归测试偶尔（也许每五到十次尝试一次）失败。键入

```
diff results/random.out expected/random.out
```

应当只产生一行或几行差异。除非随机测试在反复尝试中总是失败，否则不必担心。
（另一方面，如果在回归测试的多次尝试中随机测试从未被报告失败，你可能确实应该担心。）

26.3. 平台相关的比较文件

由于某些测试固有的会产生与平台相关的结果，
我们提供了一种提供平台相关结果比较文件的方法。通常，同一种变化适用于多个平台；
与其为每个平台提供一个单独的比较文件，不如用一个映射文件来定义使用哪个比较文件。
因此，要消除特定平台上虚假的测试“失败”，你必须选择或制作一个变体结果文件，
然后向映射文件 src/test/regress/resultmap 中添加一行。

映射文件中的每一行的形式是

```
testname/platformpattern=comparisonfilename
```

测试名就是特定回归测试模块的名称。平台模式是一个 Unix 工具 expr 风格的模式（即一个在开头隐式带有 ^ 锚点的正则表达式）。它是与 config.guess 输出的平台名后跟 :gcc 或 :cc 进行匹配（取决于你使用的是 GNU 编译器还是系统原生编译器，在有差异的系统上）。比较文件名是替代结果比较文件的名称。

例如：一些使用较老时区库的系统无法对 1970 年之前的日期应用夏令时修正，导致 1970 年前的 PDT 时间显示为 PST。这在 horology 回归测试中造成了一些差异。因此，我们提供了一个变体比较文件 horology-no-DST-before-1970.out，其中包含这些系统上应期望的结果。为了在 HPUX 平台上消除虚假的“失败”消息，resultmap 中包含了

```
horology/.*-hpux=horology-no-DST-before-1970
```

它会在任何 config.guess 输出包含 -hpux 的机器上触发。resultmap 中的其他行为其他合适的平台选择变体比较文件。

部分 IV. 客户端接口

本部分描述随PostgreSQL一起发布的客户端编程接口。这些章节彼此独立，可以分别阅读。注意，还有许多面向客户端程序的其他编程接口是单独发布的，它们各自附带自己的文档。
本部分的读者应熟悉使用SQL命令来操作和查询数据库（见第 II 部分），当然还应熟悉接口所使用的编程语言。

目录

27. libpq - C 库 289
27.1. 数据库连接控制函数 289
27.2. 连接状态函数 294
27.3. 命令执行函数 297
27.3.1. 主要函数 297
27.3.2. 检索查询结果信息 301
27.3.3. 检索其他结果信息 304
27.3.4. 用于在 SQL 命令中嵌入字符串的转义 305
27.3.5. 用于在 SQL 命令中嵌入二进制串的转义 306
27.4. 异步命令处理 307
27.5. 快速路径接口 310
27.6. 异步通知 311
27.7. 与COPY命令相关的函数 312
27.7.1. 发送COPY数据的函数 313
27.7.2. 接收COPY数据的函数 313
27.7.3. COPY的已废弃函数 314
27.8. 控制函数 316
27.9. 通知处理 316
27.10. 环境变量 317
27.11. 密码文件 318
27.12. 在线程化程序中的行为 318
27.13. 构建libpq程序 319
27.14. 示例程序 320
28. 大对象 330
28.1. 历史 330
28.2. 实现特性 330
28.3. 客户端接口 330
28.3.1. 创建一个大对象 330
28.3.2. 导入一个大对象 331
28.3.3. 导出一个大对象 331
28.3.4. 打开一个现有的大对象 331
28.3.5. 向大对象写入数据 331
28.3.6. 从大对象读取数据 331
28.3.7. 在大对象上定位 331
28.3.8. 获取大对象的当前位置 332
28.3.9. 关闭一个大对象描述符 332
28.3.10. 移除一个大对象 332
28.4. 服务器端函数 332
28.5. 示例程序 333
29. pgtcl - Tcl 绑定库 338
29.1. 概述 338
29.2. 将 pgtcl 载入应用程序 338
29.3. pgtcl 命令参考 339
29.4. 示例程序 361
30. ECPG - C 中的嵌入式 SQL 363
30.1. 概念 363
30.2. 连接到数据库服务器 363
30.3. 关闭一个连接 364
30.4. 运行 SQL 命令 365
30.5. 选择一个连接 365
30.6. 使用主变量 366

30.6.1. 概述 366
30.6.2. 声明节 366
30.6.3. SELECT INTO 和 FETCH INTO 367
30.6.4. 指示符 368
30.7. 动态 SQL 368
30.8. 使用描述符区域 369
30.9. 错误处理 371
30.9.1. 设置回调 371
30.9.2. sqlca 372
30.9.3. SQLSTATE 与 SQLCODE 373
30.10. 包含文件 375
30.11. 处理嵌入式 SQL 程序 376
30.12. 库函数 377
30.13. 内部 377
31. JDBC 接口 380
31.1. 设置 JDBC 驱动 380
31.1.1. 获取驱动 380
31.1.2. 设置类路径 380
31.1.3. 为 JDBC 准备数据库服务器 380
31.2. 初始化驱动 381
31.2.1. 导入 JDBC 381
31.2.2. 载入驱动 381
31.2.3. 连接到数据库 381
31.2.4. 关闭连接 382
31.3. 发出查询并处理结果 382
31.3.1. 基于游标获取结果 383
31.3.2. 使用 Statement 或 PreparedStatement 接口 383
31.3.3. 使用 ResultSet 接口 384
31.4. 执行更新 384
31.5. 调用存储函数 384
31.5.1. 使用 CallableStatement 接口 385
31.5.2. 从存储函数获取 ResultSet 385
31.6. 创建和修改数据库对象 386
31.7. 存储二进制数据 386
31.8. PostgreSQL 对 JDBC API 的扩展 389
31.8.1. 访问扩展 389
31.8.2. 几何数据类型 393
31.8.3. 大对象 406
31.9. 在多线程或 Servlet 环境中使用驱动 409
31.10. 连接池和数据源 409
31.10.1. 概述 409
31.10.2. 应用服务器：ConnectionPoolDataSource 410
31.10.3. 应用程序：DataSource 411
31.10.4. 数据源和 JNDI 412
31.11. 延伸阅读 413
32. 信息模式 414
32.1. 模式 414
32.2. 数据类型 414
32.3. information_schema_catalog_name 414
32.4. applicable_roles 415
32.5. check_constraints 415
32.6. column_domain_usage 415
32.7. column_privileges 416

32.8. column_udt_usage 416
32.9. columns 417
32.10. constraint_column_usage 419
32.11. constraint_table_usage 419
32.12. data_type_privileges 420
32.13. domain_constraints 420
32.14. domain_udt_usage 421
32.15. domains 421
32.16. element_types 423
32.17. enabled_roles 425
32.18. key_column_usage 425
32.19. parameters 425
32.20. referential_constraints 427
32.21. role_column_grants 427
32.22. role_routine_grants 428
32.23. role_table_grants 428
32.24. role_usage_grants 429
32.25. routine_privileges 429
32.26. routines 430
32.27. schemata 432
32.28. sql_features 433
32.29. sql_implementation_info 433
32.30. sql_languages 434
32.31. sql_packages 434
32.32. sql_sizing 434
32.33. sql_sizing_profiles 435
32.34. table_constraints 435
32.35. table_privileges 436
32.36. tables 436
32.37. triggers 437
32.38. usage_privileges 438
32.39. view_column_usage 438
32.40. view_table_usage 439
32.41. views 439

第 27 章 libpq - C 库

libpq 是 PostgreSQL 的 C 应用程序员接口。libpq 是一组库函数，客户端程序通过它们可以向 PostgreSQL 后端服务器传送查询并接收这些查询的结果。libpq 也是其他几个 PostgreSQL 应用接口的底层引擎，包括 libpq++（C++）、libpgtcl（Tcl）、Perl 和 ECPG。因此，即使你使用的是上述某个软件包，libpq行为的某些方面对你来说也很重要。

本章末尾（第 27.14 节）包含一些简短程序，展示如何编写使用libpq的程序。源代码发行包的`src/test/examples`目录中还提供了几个完整的libpq应用程序示例。

使用libpq的客户端程序必须包含头文件`libpq-fe.h`，并且必须与libpq库链接。

27.1. 数据库连接控制函数

下面的函数用于建立与 PostgreSQL 后端服务器的连接。一个应用程序可以同时打开多个后端连接（原因之一是需要访问多个数据库）。每个连接由一个 `PGconn` 对象表示，该对象通过 `PQconnectdb` 或 `PQsetdbLogin` 函数获得。注意，除非内存太少以至于连 `PGconn` 对象都无法分配，这些函数总是返回一个非空的对象指针。在通过连接对象发送查询之前，应当调用 `PQstatus` 函数检查连接是否成功建立。

`PQconnectdb`

与数据库服务器建立一个新连接。

```
PGconn *PQconnectdb(const char *conninfo);
```

此函数使用从字符串 `conninfo` 中取得的参数打开一个新的数据库连接。与下面的 `PQsetdbLogin` 不同，该函数的参数集可以在不改变函数签名的情况下扩展，因此新的应用程序编程首选使用此函数（或其非阻塞版本 `PQconnectStart` 和 `PQconnectPoll`）。

传入的字符串可以为空以使用全部默认参数，
也可以包含一个或多个用空白分隔的参数设置。每个参数设置的形式为 `keyword = value`。（要写入空值或包含空格的值，请用单引号包围，例如 `keyword = 'a value'`。值中的单引号和反斜杠必须用反斜杠转义，即 `\'` 和 `\\`。）等号两边的空格是可选的。

目前可识别的参数关键字有：

`host`

要连接的主机名。如果它以斜杠开头，则表示使用 Unix 域通信而非 TCP/IP 通信；该值是存放套接字文件的目录名。默认连接到 `/tmp` 中的 Unix 域套接字。

`hostaddr`

要连接的主机的数字 IP 地址。其格式应为标准 IPv4 地址格式，例如 `172.28.40.9`。如果你的机器支持 IPv6，也可以使用 IPv6 地址。为此参数指定非空字符串时总是使用 TCP/IP 通信。

使用 `hostaddr` 代替 `host` 允许应用程序避免主机名查找，这对于有时间约束的应用可能很重要。但 Kerberos 认证需要主机名。因此适用以下规则：如果只指定了 `host` 而没有指定 `hostaddr`，则进行主机名查找。如果只指定了 `hostaddr` 而没有指定 `host`，则 `hostaddr` 的值给出远程地址。使用 Kerberos 时，会进行反向名称查询以获得 Kerberos 所需的主机名。如果 `host` 和 `hostaddr` 都被指定，`hostaddr` 的值给出远程地址；`host` 的值被忽略；但使用 Kerberos 时例外，此时该值用于 Kerberos 认

证。（注意，如果传给 libpq 的主机名不是位于 `hostaddr` 的机器的名称，认证很可能失败。）此外，在 `$HOME/.pgpass` 中标识连接使用的是 `host` 而不是 `hostaddr`。

既没有主机名也没有主机地址时，libpq 将使用本地 Unix 域套接字连接。

`port`

服务器主机上要连接的端口号，或 Unix 域连接的套接字文件名扩展。

`dbname`

数据库名。默认与用户名相同。

`user`

要以哪个 PostgreSQL 用户名连接。

`password`

如果服务器要求密码认证则使用的密码。

`connect_timeout`

连接的最大等待时间，以秒为单位（写成十进制整数字符串）。
零或未指定表示无限等待。不建议使用小于 2 秒的超时。

`options`

要发送给服务器的命令行选项。

`tty`

忽略（以前此参数指定服务器调试输出发送到哪里）。

`sslmode`

此选项决定是否或以何种优先级与服务器协商 SSL 连接。共有四种模式：`disable` 只尝试非 SSL 加密连接；`allow` 协商时先尝试非 SSL 连接，失败后再尝试 SSL 连接；`prefer`（默认值）协商时先尝试 SSL 连接，失败后再尝试常规非 SSL 连接；`require` 只尝试 SSL 连接。

如果 PostgreSQL 编译时未包含 SSL 支持，使用选项 `require` 将导致错误，选项 `allow` 和 `prefer` 可以被接受，但 libpq 无法协商建立 SSL 连接。

`requiressl`

此选项已废弃，由 `sslmode` 设置取代。

设为 1 时，要求与服务器的 SSL 连接（等价于 `sslmode require`）。如果服务器不接受 SSL 连接，libpq 将拒绝连接。设为 0（默认值）时，libpq 将与服务器协商连接类型（等价于 `sslmode prefer`）。仅当 PostgreSQL 编译时包含 SSL 支持时此选项才可用。

`service`

用于附加参数的服务名。它指定 `pg_service.conf` 中一个持有附加连接参数的服务名。这样应用只需指定一个服务名，连接参数便可以集中维护。有关如何设置该文件，参见 *PREFIX*`/share/pg_service.conf.sample`。

如果任何参数未指定，则检查相应的环境变量（见第 27.10 节）。如果环境变量也未设置，则使用内建默认值。

`PQsetdbLogin`

与数据库服务器建立一个新连接。

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

这是 `PQconnectdb` 的前身，带有一组固定参数。除缺失的参数总是取默认值外，功能相同。对任何一个要取默认值的固定参数，可写 `NULL` 或空字符串。

`PQsetdb`

与数据库服务器建立一个新连接。

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

这是一个宏，以空指针作为 *`login`* 和 *`pwd`* 参数调用 `PQsetdbLogin`。它只是为了与很老的程序保持向后兼容而保留。

`PQconnectStart`
`PQconnectPoll`

以非阻塞方式建立与数据库服务器的连接。

```
PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

这两个函数用于打开到数据库服务器的连接，同时应用程序的执行线程不会在远程 I/O 上阻塞。这种做法的意义在于，对 I/O 完成的等待可以发生在应用的主循环中，而不是在 `PQconnectdb` 内部，因此应用可以将此操作与其他活动并行管理。

数据库连接使用传给 `PQconnectStart` 的字符串 `conninfo` 中的参数建立。该字符串的格式与上面 `PQconnectdb` 中描述的相同。

只要满足若干限制条件，`PQconnectStart` 和 `PQconnectPoll` 都不会阻塞：

- 恰当地使用 `hostaddr` 和 `host` 参数，确保不进行（正向）名称和反向名称查询。详情见上面 `PQconnectdb` 下对这些参数的说明。
- 如果调用 `PQtrace`，确保要写入跟踪信息的流对象不会阻塞。
- 在调用 `PQconnectPoll` 之前，确保套接字处于适当的状态，如下所述。

要开始一个非阻塞连接请求，调用 `conn = PQconnectStart("`*`connection_info_string`*`")`。如果 `conn` 为空，说明 libpq 无法分配新的 `PGconn` 结构。否则返回一个有效的

PGconn 指针（尽管它尚未表示一个有效的数据库连接）。从 PQconnectStart 返回后，调用 status = PQstatus(conn)。如果 status 等于 CONNECTION_BAD，则 PQconnectStart 失败。

如果 PQconnectStart 成功，下一阶段是轮询 libpq，使其继续进行连接序列。使用 PQsocket(conn) 获取数据库连接底层套接字的描述符。循环如下：如果 PQconnectPoll(conn) 上次返回 PGRES_POLLING_READING，则等待套接字可读（由 select()、poll() 或类似的系统函数指示），然后再次调用 PQconnectPoll(conn)。反之，如果 PQconnectPoll(conn) 上次返回 PGRES_POLLING_WRITING，则等待套接字可写，然后再次调用 PQconnectPoll(conn)。如果尚未调用过 PQconnectPoll（即刚调用完 PQconnectStart 之后），则视同它上次返回的是 PGRES_POLLING_WRITING。持续此循环，直到 PQconnectPoll(conn) 返回 PGRES_POLLING_FAILED（表示连接过程失败）或 PGRES_POLLING_OK（表示连接成功建立）为止。

在连接期间的任何时刻，都可以调用 PQstatus 检查连接状态。如果返回 CONNECTION_BAD，则连接过程已失败；如果返回 CONNECTION_OK，则连接已就绪。这两种状态同样可以从上面描述的 PQconnectPoll 的返回值中检测到。其他状态只在（且仅会在）异步连接过程中出现，它们指示连接过程的当前阶段，例如可用于向用户提供反馈。这些状态是：

CONNECTION_STARTED

等待建立连接。

CONNECTION_MADE

连接成功；等待发送。

CONNECTION_AWAITING_RESPONSE

等待来自服务器的响应。

CONNECTION_AUTH_OK

已通过认证；等待后端启动完成。

CONNECTION_SSL_STARTUP

正在协商 SSL 加密。

CONNECTION_SETENV

正在协商环境驱动的参数设置。

注意，尽管这些常量会保留（为了维护兼容性），
应用程序绝不应依赖它们按特定顺序出现、依赖它们全都出现，
或依赖状态总是这些已记录的值之一。应用程序可以这样做：

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Connecting...";
        break;

    case CONNECTION_MADE:
        feedback = "Connected to server...";
        break;
```

```
.
.
.
    default:
        feedback = "Connecting...";
}
```

使用 PQconnectPoll 时会忽略 connect_timeout 连接参数；由应用负责判断是否已经耗时过长。除此之外，PQconnectStart 加上 PQconnectPoll 循环等价于 PQconnectdb。

注意，如果 PQconnectStart 返回非空指针，使用完毕后必须调用 PQfinish，以释放该结构和所有关联的内存块。即使连接尝试失败或被放弃，也必须这样做。

PQconndefaults

返回默认的连接选项。

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
    char   *keyword;   /* The keyword of the option */
    char   *envvar;    /* Fallback environment variable name */
    char   *compiled;  /* Fallback compiled in default value */
    char   *val;       /* Option's current value, or NULL */
    char   *label;     /* Label for field in connect dialog */
    char   *dispchar;  /* Character to display for this field
                          in a connect dialog. Values are:
                          ""        Display entered value as is
                          "*"       Password field - hide value
                          "D"       Debug option - don't show by
 default */
    int     dispsize;  /* Field size in characters for dialog */
} PQconninfoOption;
```

返回一个连接选项数组。它可用来确定 PQconnectdb 的所有可用选项及其当前默认值。返回值指向一个 PQconninfoOption 结构数组，该数组以一个 keyword 指针为空的条目结尾。注意，当前默认值（val 字段）依赖于环境变量和其他上下文。调用者必须将连接选项数据视为只读。

处理完选项数组后，将它传递给 PQconninfoFree 释放。如果不这样做，每次调用 PQconndefaults 都会泄漏少量内存。

PQfinish

关闭与服务器的连接，同时释放 PGconn 对象使用的内存。

```
void PQfinish(PGconn *conn);
```

注意，即使与服务器的连接尝试失败（由 PQstatus 指示），应用也应调用 PQfinish 释放 PGconn 对象使用的内存。在调用过 PQfinish 之后，不得再使用该 PGconn 指针。

PQreset

重置与服务器的通信通道。

```
void PQreset(PGconn *conn);
```

此函数将关闭与服务器的连接，
并尝试使用先前使用的全部相同参数与同一服务器重新建立新连接。
如果工作中的连接丢失，这可用于错误恢复。

`PQresetStart`
`PQresetPoll`

以非阻塞方式重置与服务器的通信通道。

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

这些函数将关闭与服务器的连接，
并尝试使用先前使用的全部相同参数与同一服务器重新建立新连接。
如果工作中的连接丢失，这可用于错误恢复。它们与上面的 `PQreset` 的不同之处在于以非阻塞方式运作。这些函数受到与 `PQconnectStart` 和 `PQconnectPoll` 相同的限制。

要发起连接重置，调用 `PQresetStart`。如果返回 0，重置失败。如果返回 1，用与使用 `PQconnectPoll` 建立连接完全相同的方式，使用 `PQresetPoll` 对重置进行轮询。

27.2. 连接状态函数

这些函数可用于查询现有数据库连接对象的状态。

> **提示**
>
> 编写 libpq 应用程序时，应注意维护 `PGconn` 的抽象。请使用下述访问函数获取 `PGconn` 的内容。避免直接引用 `PGconn` 结构的字段，因为它们将来可能改变。（从 PostgreSQL 6.4 版开始，`libpq-fe.h` 中甚至不再提供 `PGconn` 背后的 `struct` 定义。如果你有直接访问 `PGconn` 字段的旧代码，可以通过同时包含 `libpq-int.h` 继续使用它，但我们建议你尽快修改这些代码。）

下面的函数返回在建立连接时确定的参数值。这些值在 `PGconn` 对象的整个生命周期内保持不变。

`PQdb`

返回连接的数据库名。

```
char *PQdb(const PGconn *conn);
```

`PQuser`

返回连接的用户名。

```
char *PQuser(const PGconn *conn);
```

`PQpass`

返回连接的密码。

```
char *PQpass(const PGconn *conn);
```

PQhost

返回连接的服务器主机名。

```
char *PQhost(const PGconn *conn);
```

PQport

返回连接的端口。

```
char *PQport(const PGconn *conn);
```

PQtty

返回连接的调试 TTY。（此功能已过时，因为服务器不再关注 TTY 设置，但为了向后兼容保留了该函数。）

```
char *PQtty(const PGconn *conn);
```

PQoptions

返回连接请求中传入的命令行选项。

```
char *PQoptions(const PGconn *conn);
```

下面的函数返回随着在 PGconn 对象上执行操作而可能变化的状态数据。

PQstatus

返回连接的状态。

```
ConnStatusType PQstatus(const PGconn *conn);
```

状态可以是若干值之一。但在异步连接过程之外只能见到其中两个：CONNECTION_OK 和 CONNECTION_BAD。正常的数据库连接状态为 CONNECTION_OK。失败的连接尝试由状态 CONNECTION_BAD 表示。通常，OK 状态会一直保持到调用 PQfinish 为止，但通信故障可能导致状态提前变为 CONNECTION_BAD。此时应用可以尝试调用 PQreset 来恢复。

关于可能见到的其他状态码，参见 PQconnectStart 和 PQconnectPoll 的条目。

PQtransactionStatus

返回服务器当前的事务内状态。

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

状态可以是 PQTRANS_IDLE（当前空闲）、PQTRANS_ACTIVE（一个命令正在执行）、PQTRANS_INTRANS（空闲，位于一个有效的事务块中）或 PQTRANS_INERROR（空闲，位于一个失败的事务块中）。连接失效时报告 PQTRANS_UNKNOWN。只有当查询已发送到服务器且尚未完成时才报告 PQTRANS_ACTIVE。

小心

当连接的是参数 autocommit 设置为 off 的 PostgreSQL 7.3 服务器时，PQtransactionStatus 会给出不正确的结果。服务器端的 autocommit 特性已被废弃，在后续服务器版本中不存在。

PQparameterStatus

查找服务器的一个当前参数设置。

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

某些参数值会在连接启动时以及每当其值变化时由服务器自动报告。可以用 PQparameterStatus 查询这些设置。它在参数已知时返回该参数的当前值，参数未知时返回 NULL。

截至当前版本会报告的参数包括 server_version（启动后不能变化）、client_encoding、is_superuser、session_authorization 和 DateStyle。

3.0 之前协议的服务器不报告参数设置，但 libpq 内含获取 server_version 和 client_encoding 值的逻辑。鼓励应用使用 PQparameterStatus 而不是自行编写代码来确定这些值。（但要注意，在 3.0 之前的连接上，连接启动后通过 SET 更改 client_encoding 不会反映到 PQparameterStatus 中。）

PQprotocolVersion

查询所使用的前端/后端协议。

```
int PQprotocolVersion(const PGconn *conn);
```

应用可能希望用它判断是否支持某些特性。目前可能的取值为 2（2.0 协议）、3（3.0 协议）或零（连接失效）。连接启动完成后此值不会变化，但在连接重置期间理论上可能变化。与 PostgreSQL 7.4 或更高版本的服务器通信时通常使用 3.0 协议；7.4 之前的服务器只支持 2.0 协议。（1.0 协议已过时，libpq 不支持。）

PQerrorMessage

返回连接上最近一次操作所产生的错误消息。

```
char *PQerrorMessage(const PGconn* conn);
```

几乎所有 libpq 函数在失败时都会为 PQerrorMessage 设置一条消息。注意，按照 libpq 的惯例，非空的 PQerrorMessage 结果会包含一个末尾换行符。

PQsocket

获取到服务器的连接套接字的文件描述符号。有效的描述符大于等于 0；结果为 -1 表示当前没有打开的服务器连接。（正常操作期间此值不会变化，
但在连接建立或重置期间可能变化。）

```
int PQsocket(const PGconn *conn);
```

PQbackendPID

返回处理此连接的后端服务器进程的进程 ID（PID）。

```
int PQbackendPID(const PGconn *conn);
```

后端 PID 可用于调试，也可用于与 NOTIFY 消息（其中包含发出通知的后端进程的 PID）进行比较。注意，该 PID 属于数据库服务器主机上执行的进程，而不是本地主机上的进程！

PQgetssl

返回连接中使用的 SSL 结构；若未使用 SSL 则返回空。

```
SSL *PQgetssl(const PGconn *conn);
```

此结构可用于核实加密级别、检查服务器证书等。有关此结构的信息，请参阅 OpenSSL 文档。

必须定义 `USE_SSL` 才能获得此函数的原型。这样做还会自动包含来自 OpenSSL 的 `ssl.h`。

27.3. 命令执行函数

与数据库服务器的连接成功建立后，此处描述的函数用于执行 SQL 查询和命令。

27.3.1. 主要函数

`PQexec`

向服务器提交一个命令并等待结果。

```
PGresult *PQexec(PGconn *conn, const char *command);
```

返回一个 `PGresult` 指针，也可能返回空指针。除内存耗尽或无法将命令发送到服务器等严重错误外，一般会返回非空指针。如果返回了空指针，应将其视同 `PGRES_FATAL_ERROR` 结果。使用 `PQerrorMessage` 获取该错误的更多信息。

命令字符串中允许包含多条 SQL 命令（用分号分隔）。在单次 `PQexec` 调用中发送的多个查询会在单个事务中处理，除非查询字符串中包含显式的 `BEGIN/COMMIT` 命令将其划分成多个事务。但要注意，返回的 `PGresult` 结构只描述字符串中最后一条被执行命令的结果。如果其中一条命令失败，对该字符串的处理即停止，返回的 `PGresult` 描述该错误情况。

`PQexecParams`

提交一个命令给服务器并且等待结果，它可以在 SQL 命令文本之外独立地传递参数。

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

`PQexecParams`与`PQexec`相似，
但是提供了额外的功能：参数值可以与命令字符串分开指定，
并且可以以文本或二进制格式请求查询结果。`PQexecParams` 仅支持使用协议 3.0 及更高版本的连接；使用协议 2.0 时会失败。

如果使用了参数，它们在命令字符串中被引用为`$1`、`$2`等。*`nParams`*是提供的参数数量；它是数组*`paramTypes[]`*、*`paramValues[]`*、*`paramLengths[]`*和*`paramFormats[]`*的长度（当*`nParams`*为零时，数组指针可以是`NULL`）。*`paramTypes[]`*通过 OID 指定要赋予给参数符号的数据类型。如果*`paramTypes`*为`NULL`或者该数组中任何特定元素为零，
服务器会用对待未指定类型的字符串字面量的相同方式为参数符号指派一种数据类型。
*`paramValues[]`*指定参数的实际值。数组中的空指针表示对应参数为 null；否则，指针指向以零结尾的文本字符串（文本格式），
或采用服务器所要求格式的二进制数据（二进制格式）。*`paramLengths[]`*

指定二进制格式参数的实际数据长度。对于 null 参数和文本格式参数，该值会被忽略。如果没有二进制参数，数组指针可以为空指针。*paramFormats[]*
指定参数采用文本格式（在数组中填入零）还是二进制格式（填入一）。
如果数组指针为空指针，则将所有参数视为文本。*resultFormat*
为零时获取文本格式的结果，为一时获取二进制格式的结果。
（目前无法让不同结果列使用不同格式，尽管底层协议支持这样做。）

PQexecParams 相对于 PQexec 的主要优点是可以将参数值与命令字符串分开，从而避免繁琐且容易出错的加引号和转义工作。和PQexec不同，PQexecParams至多允许在给定串中出现一个 SQL 命令（其中可以有分号，但是不能有超过一个非空命令）。这是底层协议的一个限制，但是有助于抵抗 SQL 注入攻击。

PQexecPrepared

发送一个以给定参数执行预备语句的请求，并等待结果。

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

PQexecPrepared 与 PQexecParams 相似，但要执行的命令通过命名一个先前已准备好的语句来指定，而不是给出查询字符串。此特性让将反复使用的命令只需解析和规划一次，而不是每次执行时都进行。PQexecPrepared 只在协议 3.0 及更高版本的连接中受支持；使用协议 2.0 时会失败。

参数与 PQexecParams 相同，只是用预备语句的名称代替查询字符串，且没有 *paramTypes[]* 参数（不需要它，因为预备语句的参数类型在创建时就已确定）。

目前，供 PQexecPrepared 使用的预备语句必须通过执行 SQL PREPARE 命令建立，该命令通常用 PQexec 发送（不过 libpq 的任何查询提交函数都可以使用）。未来的版本可能会提供更底层的语句准备接口。

PGresult 结构封装了服务器返回的结果。libpq 应用程序员应注意维护 PGresult 的抽象。请使用下面的访问函数获取 PGresult 的内容。避免直接引用 PGresult 结构的字段，因为它们将来可能改变。

PQresultStatus

返回命令的结果状态。

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus 可以返回下列值之一：

PGRES_EMPTY_QUERY

发送给服务器的字符串为空。

PGRES_COMMAND_OK

不返回数据的命令成功完成。

PGRES_TUPLES_OK

返回数据的命令（例如 SELECT 或 SHOW）成功完成。

PGRES_COPY_OUT

Copy Out（从服务器传出）数据传输已开始。

PGRES_COPY_IN

Copy In（向服务器传入）数据传输已开始。

PGRES_BAD_RESPONSE

无法理解服务器的响应。

PGRES_NONFATAL_ERROR

发生了一个非致命错误（一个通知或警告）。

PGRES_FATAL_ERROR

发生了一个致命错误。

如果结果状态为 PGRES_TUPLES_OK，就可以使用下面描述的函数检索查询返回的行。注意，碰巧检索到零行的 SELECT 命令仍然显示 PGRES_TUPLES_OK。PGRES_COMMAND_OK 用于永远不会返回行的命令（INSERT、UPDATE 等）。PGRES_EMPTY_QUERY 响应可能表明客户端软件有 bug。

状态为 PGRES_NONFATAL_ERROR 的结果永远不会由 PQexec 或其他查询执行函数直接返回；这类结果会被传递给通知处理器（见第 27.9 节）。

PQresStatus

把 PQresultStatus 返回的枚举类型转换为描述该状态码的字符串常量。

```
char *PQresStatus(ExecStatusType status);
```

PQresultErrorMessage

返回与命令相关联的错误消息；如果没有错误则返回空字符串。

```
char *PQresultErrorMessage(const PGresult *res);
```

如果发生过错误，返回的字符串将包含一个末尾换行符。

在 PQexec 或 PQgetResult 调用之后紧接着，（连接上的）PQerrorMessage 会返回与（结果上的）PQresultErrorMessage 相同的字符串。但 PGresult 会保留其错误消息直到被销毁，而连接的错误消息会随后续操作而变化。想知道与某个特定 PGresult 相关联的状态时使用 PQresultErrorMessage；想知道连接上最新操作的状态时使用 PQerrorMessage。

PQresultErrorField

返回错误报告的单个字段。

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

fieldcode 是错误字段标识符；见下面列出的符号。如果 PGresult 不是错误或警告结果，或不包含指定字段，则返回 NULL。字段值通常不包含末尾换行符。

可用的字段代码如下：

PG_DIAG_SEVERITY

严重级别；字段内容为 ERROR、FATAL 或 PANIC（错误消息中），或 WARNING、NOTICE、DEBUG、INFO 或 LOG（通知消息中），或它们的本地化翻译。总是存在。

PG_DIAG_SQLSTATE

错误的 SQLSTATE 代码（见附录 A）。不可本地化。总是存在。

PG_DIAG_MESSAGE_PRIMARY

主要的人类可读错误消息（通常一行）。总是存在。

PG_DIAG_MESSAGE_DETAIL

细节：可选的次要错误消息，携带有关该问题的更多细节。可能有多行。

PG_DIAG_MESSAGE_HINT

提示：关于如何处理该问题的可选建议。
它与细节的区别在于它提供的是建议（可能并不恰当）而非确凿事实。可能有多行。

PG_DIAG_STATEMENT_POSITION

一个包含十进制整数的字符串，以原始语句字符串中的下标指示错误光标位置。
第一个字符的下标为 1，位置按字符而非字节度量。

PG_DIAG_CONTEXT

错误发生所在上下文的指示。目前这包括活跃的 PL 函数的调用栈回溯。回溯每行一个条目，最近的在最前面。

PG_DIAG_SOURCE_FILE

报告错误的源代码位置的文件名。

PG_DIAG_SOURCE_LINE

报告错误的源代码位置的行号。

PG_DIAG_SOURCE_FUNCTION

报告错误的源代码函数的名称。

客户端负责按自身需要格式化要显示的信息；特别是应当在需要时折行。
错误消息字段中出现的换行符应被视为段落分隔而不是换行。

libpq 内部产生的错误会有严重级别和主要消息，但通常没有其他字段。3.0 之前协议的服务器返回的错误会包含严重级别和主要消息，有时还有一条细节消息，但没有其他字段。

注意，错误字段只能从 PGresult 对象获取，不能从 PGconn 对象获取；不存在 PQerrorField 函数。

`PQclear`

释放与 `PGresult` 关联的存储。每个命令结果在不再需要时都应通过 `PQclear` 释放。

```
void PQclear(PQresult *res);
```

`PGresult` 对象需要保留多久就可以保留多久；发出新命令时它不会消失，即使关闭连接也不会。要清除它，必须调用 `PQclear`。不这样做将导致应用内存泄漏。

`PQmakeEmptyPGresult`

以给定的状态构造一个空的 `PGresult` 对象。

```
PGresult* PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

这是 libpq 分配并初始化空 `PGresult` 对象的内部函数。导出它是因为某些应用发现自己生成结果对象（特别是带有错误状态的对象）很有用。如果 *conn* 非空且 *status* 指示一个错误，指定连接的当前错误消息会被复制到 `PGresult` 中。注意，最终也应对该对象调用 `PQclear`，就像对待 libpq 本身返回的 `PGresult` 一样。

27.3.2. 检索查询结果信息

这些函数用于从表示成功查询结果的`PGresult`对象（即状态为`PGRES_TUPLES_OK`的对象）中提取信息。对于其他状态值的对象，这些函数会将结果视为零行、零列。

`PQntuples`

返回查询结果中的行（元组）数量。

```
int PQntuples(const PGresult *res);
```

`PQnfields`

返回查询结果中每一行的列（字段）数。

```
int PQnfields(const PGresult *res);
```

`PQfname`

返回与给定列号相关联的列名。列号从 0 开始。

```
char *PQfname(const PGresult *res,
              int column_number);
```

如果列号超出范围，将返回`NULL`。

`PQfnumber`

返回与给定列名相关联的列号。

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

如果给定的名字不匹配任何列，将返回 -1。

给定的名称会被当作 SQL 命令中的标识符来处理，也就是说，除非用双引号括起来，否则会转换为小写。例如，对于由 SQL 命令

```
select 1 as FOO, 2 as "BAR";
```

生成的查询结果，将有如下结果：

```
PQfname(res, 0)              foo
PQfname(res, 1)              BAR
PQfnumber(res, "FOO")        0
PQfnumber(res, "foo")        0
PQfnumber(res, "BAR")        -1
PQfnumber(res, "\"BAR\"")    1
```

`PQftable`

返回给定列所取自的表的 OID。列号从 0 开始。

```
Oid PQftable(const PGresult *res,
             int column_number);
```

如果列号超出范围、指定的列不是对表列的简单引用，或者使用 3.0 之前的协议，则返回 `InvalidOid`。可以查询系统表 `pg_class`，确定所引用的具体表。

包含 libpq 头文件后，将定义类型 `Oid` 和常量 `InvalidOid`。它们都属于某种整数类型。

`PQftablecol`

返回构成指定查询结果列的列在其所属表中的列号。结果列号从 0 开始。

```
int PQftablecol(const PGresult *res,
                int column_number);
```

如果列号超出范围、指定的列不是对表列的简单引用，或者使用 3.0 之前的协议，则返回零。

`PQfformat`

返回表示给定列格式的格式代码。列号从 0 开始。

```
int PQfformat(const PGresult *res,
              int column_number);
```

格式代码零表示文本数据，格式代码一表示二进制数据。（其他代码保留供将来定义。）

`PQftype`

返回与给定列号相关联的数据类型。被返回的整数是该类型的内部 OID 号。列号从 0 开始。

```
Oid PQftype(const PGresult *res,
            int column_number);
```

可以查询系统表 `pg_type` 来获取各种数据类型的名称和属性。内置数据类型的 OID 定义在源码树的 `src/include/catalog/pg_type.h` 文件中。

`PQfmod`

返回与给定列号相关联的列的类型修饰符。列号从 0 开始。

```
int PQfmod(const PGresult *res,
           int column_number);
```

修饰符值的含义由数据类型决定，通常表示精度或大小限制。值 -1 表示“没有可用信息”。大多数数据类型不使用修饰符，此时该值始终为 -1。

PQfsize

返回给定列号对应列的大小，以字节计。列号从 0 开始。

```
int PQfsize(const PGresult *res,
            int column_number);
```

PQfsize 返回数据库行中为该列分配的空间，即服务器内部表示该数据类型所需的大小。（因此，它对客户端用处不大。）负值表示该数据类型是变长类型。

PQbinaryTuples

如果PGresult包含二进制数据，返回 1。如果包含的是文本数据，返回 0。

```
int PQbinaryTuples(const PGresult *res);
```

此函数已弃用（与 COPY 配合使用的情况除外），因为单个 PGresult 可能在部分列中包含文本数据，而在其他列中包含二进制数据。推荐使用 PQfformat。只有结果中的所有列都采用二进制格式（格式 1）时，PQbinaryTuples 才返回 1。

PQgetvalue

返回 PGresult 中某一行的一个字段值。行号和列号都从 0 开始。

```
char* PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

对于文本格式的数据，PQgetvalue 返回字段值的字符串表示，以零字节结尾。对于二进制格式的数据，返回值采用该数据类型的 typsend 和 typreceive 函数所决定的二进制表示。（这种情况下，值后面实际上也有一个零字节，但通常没有用处，
因为值本身很可能包含零字节。）

如果字段值为 null，则返回空字符串。关于如何区分 null 值与空字符串值，参见 PQgetisnull。

PQgetvalue 返回的指针指向属于 PGresult 结构体的存储空间。不应修改它所指向的数据；如果需要在 PGresult 结构体的生命周期结束后继续使用这些数据，就必须显式地将数据复制到其他存储空间。

PQgetisnull

测试一个字段是否为 null 值。行号和列号从 0 开始。

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

如果字段为 null，此函数返回 1；如果包含非 null 值，则返回 0。（注意，对于 null 字段，PQgetvalue 返回空字符串，而非空指针。）

PQgetlength

返回字段值的实际长度，以字节计。行号和列号从 0 开始。

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

这是该数据值的实际数据长度，即 `PQgetvalue` 所指对象的大小。对于文本格式的数据，它与 `strlen()` 的结果相同。对于二进制格式，这是必不可少的信息。注意，不应依赖 `PQfsize` 来获取实际数据长度。

`PQprint`

打印出所有的行以及（可选的）列名到指定的输出流。

```
void PQprint(FILE* fout,      /* output stream */
             const PGresult *res,
             const PQprintOpt *po);

typedef struct {
    pqbool  header;      /* print output field headings and row
 count */
    pqbool  align;       /* fill align the fields */
    pqbool  standard;    /* old brain dead format */
    pqbool  html3;       /* output HTML tables */
    pqbool  expanded;    /* expand tables */
    pqbool  pager;       /* use pager for output if needed */
    char    *fieldSep;   /* field separator */
    char    *tableOpt;   /* attributes for HTML table element */
    char    *caption;    /* HTML table caption */
    char    **fieldName; /* null-terminated array of replacement
 field names */
} PQprintOpt;
```

psql 以前用此函数打印查询结果，现在已不再使用。注意，此函数假定所有数据都是文本格式。

27.3.3. 检索其他结果信息

这些函数用于从并非 `SELECT` 结果的 `PGresult` 对象中提取信息。

`PQcmdStatus`

返回生成该 `PGresult` 的 SQL 命令的命令状态标签。

```
char * PQcmdStatus(PGresult *res);
```

通常这只是命令的名称，但可能包含附加数据，例如处理的行数。

`PQcmdTuples`

返回受 SQL 命令影响的行数。

```
char * PQcmdTuples(PGresult *res);
```

如果生成该 `PGresult` 的 SQL 命令是 `INSERT`、`UPDATE`、`DELETE`、`MOVE` 或 `FETCH`，此函数返回一个包含受影响行数的字符串。如果是其他命令，返回空字符串。

PQoidValue

如果 SQL 命令是一个向带 OID 的表中恰好插入一行的 INSERT，返回所插入行的 OID。否则返回 InvalidOid。

```
Oid PQoidValue(const PGresult *res);
```

PQoidStatus

如果 SQL 命令是一个 INSERT，返回一个含有所插入行 OID 的字符串。（如果该 INSERT 并非恰好插入一行，或目标表不带 OID，字符串为 0。）如果命令不是 INSERT，返回空字符串。

```
char * PQoidStatus(const PGresult *res);
```

此函数已被弃用，推荐使用PQoidValue。它不是线程安全的。

27.3.4. 用于在 SQL 命令中嵌入字符串的转义

PQescapeStringConn为了让一个字符串可用于 SQL 命令而对它进行转义。当在 SQL 命令中把数据值作为字符串字面量插入时，这很有用。一些字符（例如引号和反斜杠）必须经过转义，才不会被 SQL 解析器解释成特殊含义。PQescapeStringConn执行这种操作。

> **提示**
>
> 在处理来自不可信来源的字符串时，正确转义尤其重要。
> 否则就会有安全风险：你很容易受到“SQL 注入”攻击，不期望的 SQL 命令可能会被送入数据库。

注意，当一个数据值被作为PQexecParams或相关例程中的一个独立参数传递时，没有必要做转义而且做转义也不正确。

```
size_t PQescapeStringConn(PGconn *conn,
                          char *to, const char *from, size_t length,
                          int *error);
```

PQescapeStringConn把*from*字符串的转义版本写入*to*缓冲区，对特殊字符进行转义使它们不会造成任何危害，并添加一个终止的零字节。包围PostgreSQL字符串字面量所需的单引号不包含在结果字符串中；它们应在插入转义结果的 SQL 命令中提供。*from*参数指向待转义字符串的首字符，*length*参数给出该字符串的字节数。输入不必以零字节结尾，末尾零字节也不应计入*length*。（如果在处理完*length*个字节之前遇到末尾零字节，PQescapeStringConn会在该字节处停止；这一行为类似于strncpy。）*to*必须指向一个缓冲区，其容量至少为*length*的两倍加一个字节，否则行为未定义。如果*to*与*from*字符串重叠，行为同样未定义。

如果*error*参数不是NULL，则成功时将*error设为零，出错时设为非零。目前唯一可能的错误是源字符串中存在无效的多字节编码。出错时仍会生成输出字符串，但预计服务器会因其格式错误而拒绝它。发生错误时，无论*error*是否为NULL，一条合适的错误消息都会存储在*conn*对象中。

PQescapeStringConn返回写到*to*的字节数，不包括终止的零字节。

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

PQescapeString是PQescapeStringConn的一个较老的、已弃用的版本；区别在于它不接受*conn*或*error*参数。因此，它无法根据连接属性（例如字符编码）调整行为，可能给出错误的结果。此外，它也无法报告错误情况。

PQescapeString
可以在同一时刻仅使用一个PostgreSQL连接的单线程客户端程序中安全使用（在这种情况下，它能够“在内部”取得所需信息）。在其他情形下，它存在安全隐患，应改用PQescapeStringConn。

27.3.5. 用于在 SQL 命令中嵌入二进制串的转义

PQescapeByteaConn

对二进制数据进行转义，使其能够在 SQL 命令中用作 bytea 类型的值。与 PQescapeStringConn 一样，这仅用于将数据直接插入 SQL 命令字符串的情况。

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

当某些字节值被用作一个 SQL 语句中的 bytea 字面量的一部分时，它们必须被转义（但所有字节值都可以被转义）。一般而言，要转义一个字节，需要把它转换为与该字节值相等的三位八进制数，并在前面加上一个或两个反斜线。单引号（'）和反斜线（\）字符有特殊的替代转义序列。更多信息见第 8.4 节。PQescapeByteaConn 执行这种操作，只转义最少必需的字节。

from 参数指向待转义字符串的首字节，*from_length* 参数给出该二进制字符串的字节数。（末尾零字节既不需要，也不计入长度。）*to_length* 参数指向用于保存转义后字符串长度的变量。该结果字符串长度包含结果末尾的零字节。

PQescapeByteaConn 返回 *from*参数所指二进制字符串的转义版本，存放在通过 malloc() 分配的内存中。结果不再需要时，应使用 PQfreemem() 释放该内存。返回字符串中的所有特殊字符都已替换，以便 PostgreSQL 字符串字面量解析器和 bytea 输入函数正确处理。还会添加一个末尾零字节。包围 PostgreSQL 字符串字面量所需的单引号不包含在结果字符串中。

在发生错误时，将返回一个空指针，并且一个合适的错误消息被存储在*conn*对象中。当前，唯一可能的错误是没有足够的内存用于结果串。

PQescapeBytea

PQescapeBytea 是 PQescapeByteaConn 的旧版本，现已弃用。

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

与 PQescapeByteaConn 唯一的区别是，PQescapeBytea 不接受 PGconn 参数。因此，它无法根据连接属性（尤其是是否启用了标准符合字符串）调整行为，可能给出错误的结果。此外，它也无法在失败时返回错误消息。

PQescapeBytea 可以在同一时刻仅使用一个 PostgreSQL 连接的单线程客户端程序中安全使用（在这种情况下，它能够“在内部”取得所需信息）。在其他情形下，它存在安全隐患，应改用 PQescapeByteaConn。

PQunescapeBytea

把二进制数据的字符串表示转换为二进制数据——即 PQescapeBytea 的逆操作。以文本格式检索 bytea 数据时需要此函数；以二进制格式检索时则不需要。

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t
 *to_length);
```

*from*参数指向一个字符串，例如对bytea列调用PQgetvalue时返回的字符串。PQunescapeBytea将这个字符串表示转换为二进制表示。它返回指向通过malloc()分配的缓冲区的指针，出错时返回NULL，并将缓冲区大小存入*to_length*。不再需要结果时，必须使用PQfreemem释放它。

此转换并不完全是PQescapeBytea的逆操作，因为从PQgetvalue收到的字符串并非经过“转义”的形式。具体而言，这意味着无需考虑字符串引号，因此也不需要PGconn参数。

PQfreemem

释放libpq分配的内存。

```
void PQfreemem(void *ptr);
```

释放 libpq 分配的内存，特别是 PQescapeByteaConn、PQescapeBytea、PQunescapeBytea 和 PQnotifies 分配的内存。Microsoft Windows 需要此函数，因为它无法跨 DLL 释放内存，除非使用多线程 DLL（VC6 中的/MD）。在其他平台上，此函数与标准库函数 free()相同。

27.4. 异步命令处理

PQexec函数足以满足普通同步应用程序提交命令的需要。不过，它有两个缺点，对某些用户可能很重要：

- PQexec会等待命令完成。应用可能有其他工作要做（例如维护一个用户界面），这种情况下它不会想要阻塞等待响应。
- 因为客户端应用的执行在它等待结果时会被挂起，对于应用来说很难决定要不要尝试取消正在进行的命令（这可以在一个信号处理器中完成，但别无他法）。
- PQexec只能返回一个PGresult结构体。如果提交的命令串包含多个SQL命令，除了最后一个PGresult之外都会被PQexec丢弃。

不喜欢这些限制的应用可以改用构成 PQexec 的底层函数：PQsendQuery 和 PQgetResult。还有 PQsendQueryParams 和 PQsendQueryPrepared，它们与 PQgetResult 配合使用，可分别实现 PQexecParams 和 PQexecPrepared 的功能。

PQsendQuery

向服务器提交一个命令而不等待结果。命令成功发出返回 1，否则返回 0（此时用 PQerrorMessage 获取失败的更多信息）。

```
int PQsendQuery(PGconn *conn, const char *command);
```

成功调用 PQsendQuery 后，调用一次或多次 PQgetResult 获取结果。在 PQgetResult 返回空指针（表示命令已完成）之前，不得（在同一连接上）再次调用 PQsendQuery。

PQsendQueryParams

向服务器提交一个命令和独立的参数而不等待结果。

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

此函数等价于 PQsendQuery，区别在于查询参数可以与查询字符串分开指定。其函数参数的处理方式与 PQexecParams 完全相同。与 PQexecParams 一样，它不能用于 2.0 协议的连接，且查询字符串中只允许一个命令。

PQsendQueryPrepared

发送一个以给定参数执行预备语句的请求，而不等待结果。

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

此函数类似于 PQsendQueryParams，但要执行的命令通过命名一个先前已准备好的语句来指定，而不是给出查询字符串。其函数参数的处理方式与 PQexecPrepared 完全相同。与 PQexecPrepared 一样，它不能用于 2.0 协议的连接。

PQgetResult

等待先前一次 PQsendQuery、PQsendQueryParams 或 PQsendQueryPrepared 调用的下一个结果并将其返回。命令完成且不再有结果时返回空指针。

```
PGresult *PQgetResult(PGconn *conn);
```

必须反复调用 PQgetResult 直到它返回空指针，表示命令已完成。（如果在没有活跃命令时调用，PQgetResult 会立即返回空指针。）对 PQgetResult 返回的每个非空结果，应使用前文所述的同一批 PGresult 访问函数处理。用完每个结果对象后别忘了用 PQclear 释放。注意，只有当命令处于活跃状态且必要的响应数据尚未被 PQconsumeInput 读取时，PQgetResult 才会阻塞。

使用PQsendQuery和PQgetResult解决了PQexec
的一个问题：如果命令字符串包含多个SQL命令，就能分别获取这些命令的结果。
（这也支持一种简单的重叠处理方式：客户端可以处理某条命令的结果，
同时服务器继续处理同一命令字符串中后面的查询。）然而，调用PQgetResult
仍会导致客户端阻塞直到服务器完成下一条SQL命令。
通过恰当使用另外两个函数可以避免这一点：

PQconsumeInput

如果服务器的输入可用，消费它。

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput 通常返回 1，表示“没有错误”；发生问题时则返回 0（此时可查看 PQerrorMessage）。注意，返回值并不说明是否实际读取了输入数据。调用 PQconsumeInput 后，应用程序可以检查 PQisBusy 和/或 PQnotifies，以确定其状态是否发生变化。

即使应用程序尚未准备好处理结果或通知，也可以调用 PQconsumeInput。此函数会读取可用数据并将其保存在缓冲区中，从而清除 select() 的可读就绪指示。因此，应用程序可以用 PQconsumeInput 立即清除 select() 的就绪条件，随后在合适的时候检查结果。

PQisBusy

如果命令仍在忙碌，则返回 1，意味着PQgetResult会阻塞等待输入。返回 0 则表示可以调用PQgetResult，并保证不会阻塞。

```
int PQisBusy(PGconn *conn);
```

PQisBusy本身将不会尝试从服务器读取数据，因此必须先调用PQconsumeInput，否则繁忙状态将永远不会结束。

使用这些函数的典型应用程序会在主循环中通过 select() 或 poll() 等待需要响应的各种条件。其中一个条件是服务器有可读取的输入；对于 select()，这意味着 PQsocket 标识的文件描述符上有可读数据。主循环检测到输入就绪时，应调用 PQconsumeInput 读取输入，然后调用 PQisBusy。如果 PQisBusy 返回假（0），就可以接着调用 PQgetResult。还可以调用 PQnotifies 检测 NOTIFY 消息（见第 27.6 节）。

使用 PQsendQuery/PQgetResult 的客户端还可以尝试取消一个仍在服务器中处理的命令。

PQrequestCancel

请求服务器放弃对当前命令的处理。

```
int PQrequestCancel(PGconn *conn);
```

取消请求成功发出则返回 1，否则返回 0。（若为 0，PQerrorMessage 会说明原因。）但成功发出并不能保证请求会生效。无论 PQrequestCancel 的返回值如何，应用都必须继续用 PQgetResult 按正常顺序读取结果。如果取消生效，当前命令会提前终止并返回一个错误结果。如果取消失败（例如因为服务器已完成该命令的处理），
则根本不会有可见的结果。

注意，如果当前命令是事务块的一部分，取消将中止整个事务。

PQrequestCancel 可以在信号处理器中安全地调用。因此，如果取消的决定可以在信号处理器中做出，也可以将它与普通的 PQexec 配合使用。例如，psql 从 SIGINT 信号处理器中调用 PQrequestCancel，从而允许交互式地取消它通过 PQexec 发出的命令。

使用上述函数可以避免在等待数据库服务器输入时阻塞。不过，
应用程序仍可能在等待向服务器发送输出时阻塞。这种情况较少见，但发送很长的 SQL 命令或数据值时可能发生。（如果应用程序通过 COPY IN 发送数据，发生的可能性则大得多。）为了防止这种情况，实现完全非阻塞的数据库操作，可以使用以下附加函数。

PQsetnonblocking

设置连接的非阻塞状态。

```
int PQsetnonblocking(PGconn *conn, int arg);
```

如果*arg*为1，则将连接状态设置为非阻塞，如果 *arg*为0，则设置为阻塞。如果成功返回0，出错返回-1。

在非阻塞状态下，对 PQsendQuery、PQputline、PQputnbytes 和 PQendcopy 的调用不会阻塞；如果需要再次调用，它们会返回错误。

请注意，PQexec不遵守非阻塞模式；如果调用它，它将以阻塞方式执行。

PQisnonblocking

返回数据库连接的阻塞状态。

```
int PQisnonblocking(const PGconn *conn);
```

如果连接设置为非阻塞模式，则返回1，如果为阻塞，则返回0。

PQflush

尝试将发送队列中的输出数据发送到服务器。成功（或发送队列为空）时返回 0；因某种原因失败时返回 -1；如果尚未能发送队列中的全部数据，则返回 1（这种情况只可能发生在非阻塞连接上）。

```
int PQflush(PGconn *conn);
```

在非阻塞连接上发送任何命令或数据之后，调用 PQflush。如果返回 1，等待套接字变为可写后再次调用它；重复直到返回 0。一旦 PQflush 返回 0，等待套接字变为可读，然后按前述方法读取响应。

27.5. 快速路径接口

PostgreSQL提供一种快速路径接口来向服务器发送简单的函数调用。

> **提示**
>
> 此接口已显得有些过时，因为可以通过建立定义函数调用的预备语句，
> 获得相近的性能和更多功能。随后使用二进制格式传输参数和结果来执行该语句，
> 就可以替代快速路径函数调用。

函数 PQfn 通过快速路径接口请求执行一个服务器函数：

```
PGresult* PQfn(PGconn* conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct {
    int len;
    int isint;
    union {
```

```
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

fnid 参数是要执行函数的 OID。*args* 和 *nargs* 指定传给函数的参数，必须与函数声明中的参数列表匹配。参数结构体的 *isint* 字段为真时，*u.integer* 值会以指定长度的整数发送到服务器，该长度必须是 2 或 4 字节，并会进行适当的字节序转换。*isint* 为假时，位于 **u.ptr* 的指定数量字节会原样发送；数据必须符合服务器对该函数参数数据类型的二进制传输格式要求。（将 u.ptr 声明为 int * 是历史原因；将其视为 void * 更合适。）*result_buf* 指向用于存放函数返回值的缓冲区。调用者必须事先分配足够空间来保存返回值，这里不会检查！实际结果长度以字节为单位，返回到 *result_len* 指向的整数中。如果预期结果是 2 或 4 字节整数，将 *result_is_int* 设为 1，否则设为 0。将 *result_is_int* 设为 1 后，libpq 会按需转换字节序，使结果成为适合客户端机器的 `int` 值；注意，无论是哪种允许的结果大小，传入 *result_buf 的都是 4 字节整数。*result_is_int* 为 0 时，服务器发送的二进制格式字节串会原样返回。（此时，将 result_buf 视为 void * 更合适。）

`PQfn` 总是返回有效的 `PGresult` 指针。使用结果前应检查其状态。不再需要结果时，调用者负责使用 `PQclear` 释放 `PGresult`。

注意，使用此接口时无法处理 NULL 参数、NULL 结果以及集合值结果。

27.6. 异步通知

PostgreSQL通过`LISTEN`和`NOTIFY`命令提供异步通知。客户端会话可使用`LISTEN`命令注册其感兴趣的特定通知通道（也可以用`UNLISTEN`命令停止监听）。当任何会话执行带有该通道名的`NOTIFY`命令时，所有监听该通道的会话都会异步收到通知。

libpq应用把`LISTEN`、`UNLISTEN`和`NOTIFY`命令作为普通 SQL 命令提交。随后通过调用`PQnotifies`来检测 NOTIFY 消息的到达。

函数 `PQnotifies` 从服务器发来的尚未处理的通知消息列表中返回下一条通知。如果没有待处理的通知则返回空指针。一条通知被 `PQnotifies` 返回后即视为已处理，并会从通知列表中移除。

```
PGnotify* PQnotifies(PGconn *conn);

typedef struct pgNotify {
    char *relname;              /* notification condition name */
    int  be_pid;                /* process ID of server process */
    char *extra;                /* notification parameter */
} PGnotify;
```

处理完 `PQnotifies` 返回的 `PGnotify` 对象后，务必用 `PQfreemem` 释放它。只需释放 `PGnotify` 指针本身即可；`relname` 和 `extra` 字段并不是独立分配的。（目前 `extra` 字段未使用，总是指向一个空字符串。）

> **注意**
>
> 在PostgreSQL 6.4 及之后的版本中，`be_pid`是发送通知的服务器进程的 PID，而在较早的版本中它总是你自己的服务器进程的PID。

例 27.2给出了一个示例程序展示异步通知的使用。

PQnotifies实际上并不从服务器读取数据；它只是返回之前已被其他libpq函数读取的消息。在较早版本的libpq中，及时收到NOTIFY消息的唯一方法是不断提交命令，哪怕是空命令，然后在每次PQexec后检查PQnotifies。虽然这种方法仍然有效，但由于浪费处理能力，现已弃用。

当没有需要执行的实际命令时，一种更好的检查NOTIFY消息的方法是调用PQconsumeInput，然后检查PQnotifies。你可以使用select()等待服务器数据到达，这样在无事可做时就不会浪费CPU资源（参见PQsocket以获得可传给select()的文件描述符）。注意不管是用PQsendQuery/PQgetResult提交命令还是简单地使用PQexec，这种方法都能正常工作。不过，你应该记住在每次PQgetResult 或 PQexec之后检查PQnotifies，看看在命令的处理过程中是否有通知到达。

27.7. 与COPY命令相关的函数

PostgreSQL 的 COPY 命令提供了选项，可以通过 libpq 使用的网络连接读取或写入数据。本节介绍的函数允许应用程序通过提供或接收复制数据来使用这一能力。

整体流程如下：应用程序先通过 PQexec 或等效函数发出 SQL COPY 命令。如果命令没有错误，响应就是一个 PGresult 对象，其状态码为 PGRES_COPY_OUT 或 PGRES_COPY_IN，取决于指定的复制方向。应用程序随后应使用本节函数接收或发送数据行。数据传输完成后，会返回另一个 PGresult 对象，表示传输成功或失败：成功时状态为 PGRES_COMMAND_OK，出现问题时为 PGRES_FATAL_ERROR。此时可以通过 PQexec 继续发出 SQL 命令。（COPY 操作进行期间，不能在同一连接上执行其他 SQL 命令。）

如果一个COPY命令是通过PQexec在一个可能包含额外命令的字符串中发出的，那么应用在完成COPY序列之后必须继续用PQgetResult取得结果。只有在PQgetResult返回NULL时，我们才能确信PQexec的命令字符串已经处理完毕，并且可以安全地发出更多命令。

只有从 PQexec 或 PQgetResult 获得 PGRES_COPY_OUT 或 PGRES_COPY_IN 结果状态后，才应调用本节函数。

带有上述某个状态值的 PGresult 对象，还会携带关于即将开始的 COPY 操作的附加数据。这些数据可以通过下列函数获取，这些函数也用于查询结果：

PQnfields

返回要复制的列（字段）的数量。

PQbinaryTuples

0表示整体复制格式为文本（行由换行符分隔，列由分隔符分隔等）。
1表示整体复制格式为二进制。有关更多信息，请参见COPY。

PQfformat

返回与复制操作的每列关联的格式代码（0表示文本，1表示二进制）。当整体复制格式为文本时，每列的格式代码始终为零，但二进制格式可以支持文本和二进制列。（但是，在当前的 COPY 实现中，只有二进制列出现在二进制复制中；因此，每列格式目前始终与整体格式匹配。）

注意

这些附加数据值仅在使用协议 3.0 时可用。使用协议 2.0 时，这些函数都返回 0。

27.7.1. 发送COPY数据的函数

这些函数用于在 COPY FROM STDIN 期间发送数据。如果连接不处于 COPY_IN 状态，调用它们会失败。

PQputCopyData

在COPY_IN状态中向服务器发送数据。

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

将指定 *buffer* 中长度为 *nbytes* 的 COPY 数据传输到服务器。数据发送成功时返回 1；因尝试会阻塞而未发送时返回零（仅可能发生在连接处于非阻塞模式时）；发生错误时返回 -1。（返回 -1 时，可用 PQerrorMessage 获取详细信息。返回零时，应等待可写就绪后重试。）

应用程序可以将 COPY 数据流分成任意方便大小的数据块，逐块装入缓冲区。发送时，这些数据块的边界没有语义含义。数据流内容必须符合 COPY 命令预期的数据格式；详见COPY。

PQputCopyEnd

在COPY_IN状态中向服务器发送数据结束的指示。

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

如果 *errormsg* 为 NULL，则成功结束 COPY_IN 操作。如果 *errormsg* 不为 NULL，则强制 COPY 失败，并将 *errormsg* 指向的字符串用作错误消息。（但不应假定服务器一定会返回这条完全相同的错误消息，因为服务器可能已经因自身原因使 COPY 失败。还要注意，在使用 3.0 之前协议的连接上，强制失败选项不起作用。）

终止数据发送成功时返回 1；因尝试会阻塞而未发送时返回零（仅可能发生在连接处于非阻塞模式时）；发生错误时返回 -1。（返回 -1 时，可用 PQerrorMessage 获取详细信息。返回零时，应等待可写就绪后重试。）

成功调用 PQputCopyEnd 后，调用 PQgetResult 获取 COPY 命令的最终结果状态。可以按通常方式等待该结果就绪，然后恢复正常操作。

27.7.2. 接收COPY数据的函数

这些函数用于在COPY TO STDOUT的过程中接收数据。如果连接不在COPY_OUT状态，那么调用它们将会失败。

PQgetCopyData

在COPY_OUT状态下从服务器接收数据。

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

在 COPY 期间尝试从服务器获取下一行数据。每次总是返回一个完整数据行；如果只有部分行可用，则不返回。成功返回数据行时，会分配一块内存保存数据。*buffer* 参数必须为非

NULL。*buffer* 会被设置为指向所分配的内存；如果没有返回缓冲区，则设为 NULL。非 NULL 的结果缓冲区在不再需要时应使用 PQfreemem 释放。

成功返回一行时，返回值是该行的数据字节数，始终大于零。
返回的字符串总是以零字节结尾，不过这可能仅对文本 COPY 有用。返回零表示 COPY 仍在进行，但尚无可用行（仅在 *async* 为真时可能发生）。返回 -1 表示 COPY 已完成；返回 -2 表示发生了错误（可用 PQerrorMessage 查看原因）。

当 *async* 为真（非零）时，PQgetCopyData 不会阻塞等待输入；如果 COPY 仍在进行，但没有完整行可用，则返回零。（此时，应等待读就绪，先调用 PQconsumeInput，再调用 PQgetCopyData。）当 *async* 为假（零）时，PQgetCopyData 会阻塞，直到数据可用或操作完成。

在 PQgetCopyData 返回 -1 后，调用 PQgetResult 获取 COPY 命令的最终结果状态。可以按通常方式等待该结果就绪，然后恢复正常操作。

27.7.3. COPY的已废弃函数

这些函数使用较旧的方式处理 COPY。虽然仍然可用，但由于错误处理欠佳、检测数据结束的方式不便，而且缺少对二进制或非阻塞传输的支持，已被弃用。

PQgetline

将服务器传来的、以换行符结尾的一行字符读入大小为 *length* 的字符串缓冲区。

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

此函数最多将 *length*-1 个字符复制到缓冲区，并将末尾的换行符转换为零字节。PQgetline 在输入结束时返回 EOF，读完一整行时返回 0，缓冲区已满但尚未读到末尾换行符时返回 1。

注意，应用程序必须检查新读入的一行是否仅由 \. 两个字符组成，这表示服务器已发送完 COPY 命令的结果。如果可能收到长度超过 *length*-1 个字符的行，必须确保正确识别 \. 行，例如不能把长数据行的末尾误当作终止行。

PQgetlineAsync

以非阻塞方式将服务器传来的一行 COPY 数据读入缓冲区。

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

此函数类似于 PQgetline，但可用于必须异步读取 COPY 数据的应用程序，即读取时不阻塞。发出 COPY 命令并收到 PGRES_COPY_OUT 响应后，应用程序应调用 PQconsumeInput 和 PQgetlineAsync，直到检测到数据结束信号。

与 PQgetline 不同，此函数会负责检测数据结束。

每次调用时，如果 libpq 的输入缓冲区中有完整数据行，PQgetlineAsync 就会返回数据；否则，要等该行剩余部分到达后才返回数据。识别到复制数据结束标记时返回 -1，没有可用数据时返回 0，否则返回正数，表示返回的数据字节数。返回 -1 后，调用者必须接着调用 PQendcopy，然后恢复正常处理。

返回的数据不会跨越数据行边界。只要可能，每次就返回一整行；
但如果调用者提供的缓冲区太小，容不下服务器发送的一行，则只返回部分行。
对于文本数据，可检查最后返回的字节是否为 `\n`，以判断是否返回了完整行。（对于二进制 `COPY`，则必须实际解析 `COPY` 数据格式才能作出相同判断。）返回的字符串不以零字节结尾。（如果要自行添加末尾的零字节，务必将传入的 *bufsize* 设置为比实际可用空间少一字节。）

`PQputline`

向服务器发送以零字节结尾的字符串。成功时返回 0，无法发送字符串时返回 `EOF`。

```
int PQputline(PGconn *conn,
              const char *string);
```

连续调用 `PQputline` 发送的 `COPY` 数据流，与 `PQgetlineAsync` 返回的数据格式相同。不过，应用程序不必在每次 `PQputline` 调用中恰好发送一个数据行；每次发送部分行或多行也可以。

> **注意**
>
> 在 PostgreSQL 协议 3.0 之前，应用程序必须显式发送由 `\.` 两个字符组成的最后一行，告知服务器应用程序已发送完 `COPY` 数据。虽然这种方式仍然有效，但已被弃用，`\.` 的特殊含义预计会在未来版本中移除。发送完实际数据后，调用 `PQendcopy` 即可。

`PQputnbytes`

向服务器发送不以零字节结尾的字符串。成功时返回 0，无法发送字符串时返回 `EOF`。

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

此函数与 `PQputline` 完全相同，只是直接指定了要发送的字节数，因此数据缓冲区不必以零字节结尾。发送二进制数据时可使用此函数。

`PQendcopy`

与服务器同步。

```
int PQendcopy(PGconn *conn);
```

此函数会等待服务器完成复制。应在使用 `PQputline` 向服务器发送最后一个字符串后，或使用 `PQgetline` 从服务器接收最后一个字符串后调用它。必须调用此函数，否则服务器与客户端会“失去同步”。函数返回后，服务器便准备好接收下一条 SQL 命令。成功完成时返回 0，否则返回非零值。（返回非零值时，可用 `PQerrorMessage` 获取详细信息。）

使用 `PQgetResult` 时，收到 `PGRES_COPY_OUT` 结果后，应用程序应反复调用 `PQgetline`，并在看到终止行后调用 `PQendcopy`。随后应回到 `PQgetResult` 循环，直到 `PQgetResult` 返回空指针。类似地，收到 `PGRES_COPY_IN` 结果后，应连续调用 `PQputline`，再调用 `PQendcopy`，然后回到 `PQgetResult` 循环。这样可以保证嵌在一系列 SQL 命令中的 `COPY` 命令正确执行。

旧的应用很可能会通过`PQexec`提交一个`COPY`命令并且假定事务在`PQendcopy`之后完成。只有在`COPY`是命令字符串中唯一的SQL命令时才能正确工作。

27.8. 控制函数

这些函数控制 libpq 行为的各种细节。

`PQsetErrorVerbosity`

确定 `PQerrorMessage` 和 `PQresultErrorMessage` 返回的消息的详细程度。

```
typedef enum {
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity
 verbosity);
```

`PQsetErrorVerbosity` 设置冗长模式并返回连接先前的设置。在terse（简短）模式下，返回的消息只包含严重级别、主要文本和位置；这通常能放在一行内。
默认模式产生的消息除上述内容外还包含任何细节、
提示或上下文字段（这些可能占多行）。VERBOSE（详细）模式包含所有可用字段。
更改冗长程度不影响已存在的 `PGresult` 对象中可获取的消息，只影响之后创建的对象。

`PQtrace`

启用客户端/服务器通信跟踪，将跟踪信息写入调试文件流。

```
void PQtrace(PGconn *conn, FILE *stream);
```

`PQuntrace`

禁用由 `PQtrace` 启动的跟踪。

```
void PQuntrace(PGconn *conn);
```

27.9. 通知处理

服务器生成的通知和警告消息不由查询执行函数返回，因为它们并不表示查询失败。
这些消息会传递给通知处理函数，处理函数返回后，执行正常继续。
默认通知处理函数将消息打印到 `stderr`，应用程序也可以提供自己的处理函数来替换这一行为。

由于历史原因，通知处理有两个级别，称为通知接收器和通知处理器。
通知接收器的默认行为是格式化通知并且将一个字符串传递给通知处理器来打印。不过，
如果一个应用选择提供自己的通知接收器，
它通常会忽略通知处理器层并且在通知接收器中完成所有工作。

函数`PQsetNoticeReceiver` 为一个连接对象设置或者检查当前的通知接收器。相似地，`PQsetNoticeProcessor` 设置或检查当前的通知处理器。

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
```

```
                    PQnoticeReceiver proc,
                    void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

这些函数中的每一个会返回之前的通知接收器或处理器函数指针，并且设置新值。
如果你提供了一个空函数指针，将不会采取任何动作，只会返回当前指针。

从服务器收到通知或警告消息，或者 libpq 内部生成这类消息时，便会调用通知接收器函数。消息以状态为 PGRES_NONFATAL_ERROR 的 PGresult 对象传给该函数。（接收器可使用 PQresultErrorField 提取单个字段，或使用 PQresultErrorMessage 获取完整的预格式化消息。）传给 PQsetNoticeReceiver 的同一个 void 指针也会传入。（必要时，可通过该指针访问应用程序特有的状态。）

默认的通知接收器会简单地抽取消息（使用PQresultErrorMessage）
并且将它传递给通知处理器。

通知处理器负责处理以文本形式提供的通知或警告消息。
传给它的参数包括消息字符串（含末尾的换行符），以及先前传给 PQsetNoticeProcessor 的同一个 void 指针。（必要时，可通过该指针访问应用程序特有的状态。）

默认的通知处理器只是

```
static void
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

设置通知接收器或处理器后，应当预期：只要 PGconn 对象或由它生成的 PGresult 对象仍存在，就可能调用该函数。创建 PGresult 时，会将 PGconn 当前的通知处理指针复制到 PGresult 中，供 PQgetvalue 等函数在需要时使用。

27.10. 环境变量

下面的环境变量可用于选择默认的连接参数值；当调用代码没有直接指定值时，PQconnectdb、PQsetdbLogin 和 PQsetdb 将使用这些值。例如，它们有助于避免把数据库连接信息硬编码进简单的客户端应用。

- PGHOST 设置数据库服务器名。如果以斜杠开头，则表示使用 Unix 域通信而非 TCP/IP 通信；该值是存放套接字文件的目录名（默认 /tmp）。
- PGHOSTADDR 指定数据库服务器的数字 IP 地址。可以设置它来代替 PGHOST，以避免 DNS 查找开销。关于这些参数的相互作用，参见上面 PQconnectdb 下对这些参数的说明。
- PGPORT 设置与 PostgreSQL 服务器通信所用的 TCP 端口号或 Unix 域套接字文件扩展名。
- PGDATABASE 设置 PostgreSQL 数据库名。
- PGUSER 设置连接数据库所用的用户名。

- PGPASSWORD 设置当服务器要求密码认证时使用的密码。出于安全原因，此环境变量已废弃；请考虑改用 $HOME/.pgpass 文件（见第 27.11 节）。
- PGSERVICE 设置要在 pg_service.conf 中查找的服务名。这提供了一种设置全部参数的简写方式。
- PGREALM 设置与 PostgreSQL 一起使用的 Kerberos realm（当它与本地 realm 不同时）。如果设置了 PGREALM，libpq 应用将尝试对该 realm 中的服务器进行认证，并使用单独的 ticket 文件以避免与本地 ticket 文件冲突。仅当服务器选择了 Kerberos 认证时才会使用此环境变量。
- PGOPTIONS 设置 PostgreSQL 服务器的附加运行时选项。
- PGSSLMODE 决定是否以及以何种优先级与服务器协商 SSL 连接。共有四种模式：disable 只尝试非加密的 SSL 连接；allow 协商时先尝试非 SSL 连接，失败后再尝试 SSL 连接；prefer（默认值）协商时先尝试 SSL 连接，失败后再尝试常规的非 SSL 连接；require 只尝试 SSL 连接。如果 PostgreSQL 编译时未包含 SSL 支持，使用选项 require 将导致错误，选项 allow 和 prefer 可以被接受，但 libpq 无法协商建立 SSL 连接。
- PGREQUIRESSL 设置是否必须通过 SSL 建立连接。若设为“1”，当服务器不接受 SSL 连接时，libpq 将拒绝连接（等价于 sslmode prefer）。此选项已废弃，由 sslmode 设置取代；仅当 PostgreSQL 编译时包含 SSL 支持时才可用。
- PGCONNECT_TIMEOUT 设置 libpq 尝试连接 PostgreSQL 服务器时等待的最大秒数。若未设置或设为零，libpq 将无限等待。不建议将超时设为小于 2 秒。

下面的环境变量可用于为每个 PostgreSQL 会话指定默认行为。（关于按用户或按数据库设置默认行为的方法，另见 ALTER USER 和 ALTER DATABASE 命令。）

- PGDATESTYLE 设置日期/时间表示的默认风格。（等价于 SET datestyle TO ...。）
- PGTZ 设置默认时区。（等价于 SET timezone TO ...。）
- PGCLIENTENCODING 设置默认的客户端字符集编码。（等价于 SET client_encoding TO ...。）
- PGGEQO 设置遗传查询优化器的默认模式。（等价于 SET geqo TO ...。）

关于这些环境变量的正确取值，请参阅 SQL 命令 SET。

27.11. 密码文件

用户主目录中的 .pgpass 文件可以包含在连接需要密码（且未通过其他方式指定密码）时使用的密码。此文件的每一行应采用如下格式：

hostname:*port*:*database*:*username*:*password*

前四个字段中的每一个都可以是字面值，或者 * （匹配任意值）。将与当前连接参数匹配的第一行中的密码字段投入使用。（因此，使用通配符时应把更具体的条目放在前面。）
如果某个条目需要包含 : 或 \，请用 \ 转义该字符。

.pgpass 的权限必须禁止任何组或其他用户的访问；可以用命令 chmod 0600 ~/.pgpass 实现。如果权限比这宽松，该文件将被忽略。

27.12. 在线程化程序中的行为

如果在构建 PostgreSQL 发行版时使用了 `configure` 命令行选项 `--enable-thread-safety`，那么 libpq 是可重入且线程安全的。此外，编译应用程序代码时可能还需要使用附加的编译器命令行选项。有关如何构建支持线程的应用，请参阅你的系统文档。

一个限制是，不能有两个线程同时尝试操作同一个 `PGconn` 对象。特别是，不能通过同一个连接对象从不同线程发出并发的命令。（如果需要运行并发命令，请建立多个连接。）

`PGresult`对象在创建后是只读的，因此可以在线程之间自由传递。

已废弃的函数 `PQoidStatus` 和 `fe_setauthsvc` 不是线程安全的，不应在多线程程序中使用。`PQoidStatus` 可以由 `PQoidValue` 代替。而调用 `fe_setauthsvc` 则根本没有正当理由。

使用`crypt`认证方法的libpq应用程序依赖操作系统的`crypt()`函数，
而它常常不是线程安全的。最好使用`md5`方法，它在所有平台上都是线程安全的。

27.13. 构建libpq程序

要构建（即编译和链接）你的 libpq 程序，需要完成以下所有事项：

- 包含 `libpq-fe.h` 头文件：

  ```
  #include <libpq-fe.h>
  ```

  如果没有这样做，编译器通常会报出类似下面的错误消息：

  ```
  foo.c: In function `main':
  foo.c:34: `PGconn' undeclared (first use in this function)
  foo.c:35: `PGresult' undeclared (first use in this function)
  foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
  ```

- 通过给编译器提供 `-I`*`directory`* 选项，让编译器找到 PostgreSQL 头文件的安装目录。（某些情况下编译器默认就会搜索该目录，因此可以省略此选项。）例如，编译命令行可以是：

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  如果使用 makefile，则把该选项加到 `CPPFLAGS` 变量中：

  ```
  CPPFLAGS += -I/usr/local/pgsql/include
  ```

  如果程序有可能被其他用户编译，就不要像这样把目录位置硬编码。可以运行工具 `pg_config` 来查明头文件在本地系统中的位置：

  ```
  $ pg_config --includedir
  /usr/local/include
  ```

  未给编译器指定正确的选项将导致类似下面的错误消息：

  ```
  testlibpq.c:8:22: libpq-fe.h: No such file or directory
  ```

- 链接最终程序时，指定 `-lpq` 选项以引入 libpq 库，同时指定 `-L`*`directory`* 选项让编译器找到 libpq 库所在的目录。（编译器同样会默认搜索某些目录。）为获得最大的可移植性，应把 `-L` 选项放在 `-lpq` 选项之前。例如：

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

也可以用 pg_config 查明库目录：

```
$ pg_config --libdir
/usr/local/pgsql/lib
```

指向这方面问题的错误消息可能如下所示。

```
testlibpq.o: In function `main':
testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
testlibpq.o(.text+0x71): undefined reference to `PQstatus'
testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
```

这表示你忘记了 -lpq。

```
/usr/bin/ld: cannot find -lpq
```

这表示你忘记了 -L 选项，或者没有指定正确的目录。

如果你的代码引用了头文件libpq-int.h，而你拒绝修改代码使其不再使用它，那么从 PostgreSQL 7.2 开始，这个文件位于*includedir*/postgresql/internal/libpq-int.h，因此你需要在编译器命令行中加入适当的-I选项。

27.14. 示例程序

这些示例和其他示例可在源代码发行包的src/test/examples目录中找到。

例 27.1. libpq示例程序 1

```
/*
 * testlibpq.c
 *
 *              Test the C version of LIBPQ, the POSTGRES frontend
 library.
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
        PQfinish(conn);
        exit(1);
}

int
main(int argc, char **argv)
{
        const char *conninfo;
        PGconn     *conn;
        PGresult   *res;
        int                     nFields;
        int                     i,
```

```
                                j;

        /*
         * If the user supplies a parameter on the command line, use
 it as
         * the conninfo string; otherwise default to setting dbname=
template1
         * and using environment variables or defaults for all other
 connection
         * parameters.
         */
        if (argc > 1)
                conninfo = argv[1];
        else
                conninfo = "dbname = template1";

        /* Make a connection to the database */
        conn = PQconnectdb(conninfo);

        /* Check to see that the backend connection was successfully
 made */
        if (PQstatus(conn) != CONNECTION_OK)
        {
                fprintf(stderr, "Connection to database '%s' failed.
\n", PQdb(conn));
                fprintf(stderr, "%s", PQerrorMessage(conn));
                exit_nicely(conn);
        }

        /*
         * Our test case here involves using a cursor, for which we
 must be
         * inside a transaction block.  We could do the whole thing
 with a
         * single PQexec() of "select * from pg_database", but that's
 too
         * trivial to make a good example.
         */

        /* Start a transaction block */
        res = PQexec(conn, "BEGIN");
        if (PQresultStatus(res) != PGRES_COMMAND_OK)
        {
                fprintf(stderr, "BEGIN command failed: %s", PQerror
Message(conn));
                PQclear(res);
                exit_nicely(conn);
        }

        /*
         * Should PQclear PGresult whenever it is no longer needed to
 avoid
         * memory leaks
         */
```

```
        PQclear(res);

        /*
         * Fetch rows from pg_database, the system catalog of
 databases
         */
        res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from
 pg_database");
        if (PQresultStatus(res) != PGRES_COMMAND_OK)
        {
                fprintf(stderr, "DECLARE CURSOR failed: %s", PQerror
Message(conn));
                PQclear(res);
                exit_nicely(conn);
        }
        PQclear(res);

        res = PQexec(conn, "FETCH ALL in myportal");
        if (PQresultStatus(res) != PGRES_TUPLES_OK)
        {
                fprintf(stderr, "FETCH ALL failed: %s", PQerror
Message(conn));
                PQclear(res);
                exit_nicely(conn);
        }

        /* first, print out the attribute names */
        nFields = PQnfields(res);
        for (i = 0; i < nFields; i++)
                printf("%-15s", PQfname(res, i));
        printf("\n\n");

        /* next, print out the rows */
        for (i = 0; i < PQntuples(res); i++)
        {
                for (j = 0; j < nFields; j++)
                        printf("%-15s", PQgetvalue(res, i, j));
                printf("\n");
        }

        PQclear(res);

        /* close the portal ... we don't bother to check for errors .
.. */
        res = PQexec(conn, "CLOSE myportal");
        PQclear(res);

        /* end the transaction */
        res = PQexec(conn, "END");
        PQclear(res);

        /* close the connection to the database and cleanup */
        PQfinish(conn);
```

```
        return 0;
}
```

例 27.2. libpq示例程序 2

```
/*
 * testlibpq2.c
 *              Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 * Repeat four times to get this program to exit.
 *
 * Or, if you want to get fancy, try this:
 * populate a database with the following commands
 * (provided in src/test/examples/testlibpq2.sql):
 *
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * and do this four times:
 *
 *   INSERT INTO TBL1 VALUES (10);
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
        PQfinish(conn);
        exit(1);
}

int
main(int argc, char **argv)
{
        const char *conninfo;
        PGconn     *conn;
        PGresult   *res;
        PGnotify   *notify;
        int                     nnotifies;

        /*
         * If the user supplies a parameter on the command line, use
 it as
```

```
         * the conninfo string; otherwise default to setting dbname=
template1
         * and using environment variables or defaults for all other
 connection
         * parameters.
         */
        if (argc > 1)
                conninfo = argv[1];
        else
                conninfo = "dbname = template1";

        /* Make a connection to the database */
        conn = PQconnectdb(conninfo);

        /* Check to see that the backend connection was successfully
 made */
        if (PQstatus(conn) != CONNECTION_OK)
        {
                fprintf(stderr, "Connection to database '%s' failed.
\n", PQdb(conn));
                fprintf(stderr, "%s", PQerrorMessage(conn));
                exit_nicely(conn);
        }

        /*
         * Issue LISTEN command to enable notifications from the
 rule's NOTIFY.
         */
        res = PQexec(conn, "LISTEN TBL2");
        if (PQresultStatus(res) != PGRES_COMMAND_OK)
        {
                fprintf(stderr, "LISTEN command failed: %s", PQerror
Message(conn));
                PQclear(res);
                exit_nicely(conn);
        }

        /*
         * should PQclear PGresult whenever it is no longer needed to
 avoid
         * memory leaks
         */
        PQclear(res);

        /* Quit after four notifies are received. */
        nnotifies = 0;
        while (nnotifies < 4)
        {
                /*
                 * Sleep until something happens on the connection.
 We use select(2)
                 * to wait for input, but you could also use poll() or
 similar
                 * facilities.
```

```
                 */
                int                     sock;
                fd_set          input_mask;

                sock = PQsocket(conn);

                if (sock < 0)
                        break;                          /* shouldn't
 happen */

                FD_ZERO(&input_mask);
                FD_SET(sock, &input_mask);

                if (select(sock + 1, &input_mask, NULL, NULL, NULL) <
 0)
                {
                        fprintf(stderr, "select() failed: %s\n",
 strerror(errno));
                        exit_nicely(conn);
                }

                /* Now check for input */
                PQconsumeInput(conn);
                while ((notify = PQnotifies(conn)) != NULL)
                {
                        fprintf(stderr,
                                        "ASYNC NOTIFY of '%s' received
 from backend pid %d\n",
                                        notify->relname, notify->be_
pid);
                        PQfreemem(notify);
                        nnotifies++;
                }
        }

        fprintf(stderr, "Done.\n");

        /* close the connection to the database and cleanup */
        PQfinish(conn);

        return 0;
}
```

例 27.3. libpq Example Program 3

```
/*
 * testlibpq3.c
 *              Test out-of-line parameters and binary I/O.
 *
 * Before running this, populate a database with the following
 commands
 * (provided in src/test/examples/testlibpq3.sql):
 *
 * CREATE TABLE test1 (i int4, t text, b bytea);
```

```
 *
 * INSERT INTO test1 values (1, 'joe''s place', '\\000\\001\\002\\003\
\004');
 * INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\
\000');
 *
 * The expected output is:
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(PGconn *conn)
{
        PQfinish(conn);
        exit(1);
}

int
main(int argc, char **argv)
{
        const char *conninfo;
        PGconn     *conn;
        PGresult   *res;
        const char *paramValues[1];
        int                     i,
                                j;
        int                     i_fnum,
                                t_fnum,
                                b_fnum;

        /*
         * If the user supplies a parameter on the command line, use
 it as
         * the conninfo string; otherwise default to setting dbname=
template1
         * and using environment variables or defaults for all other
 connection
         * parameters.
         */
```

```
        if (argc > 1)
                conninfo = argv[1];
        else
                conninfo = "dbname = template1";

        /* Make a connection to the database */
        conn = PQconnectdb(conninfo);

        /* Check to see that the backend connection was successfully
 made */
        if (PQstatus(conn) != CONNECTION_OK)
        {
                fprintf(stderr, "Connection to database '%s' failed.
\n", PQdb(conn));
                fprintf(stderr, "%s", PQerrorMessage(conn));
                exit_nicely(conn);
        }

        /*
         * The point of this program is to illustrate use of PQexec
Params()
         * with out-of-line parameters, as well as binary transmission
 of
         * results.  By using out-of-line parameters we can avoid a
 lot of
         * tedious mucking about with quoting and escaping.  Notice
 how we
         * don't have to do anything special with the quote mark in
 the
         * parameter value.
         */

        /* Here is our out-of-line parameter value */
        paramValues[0] = "joe's place";

        res = PQexecParams(conn,
                                                "SELECT * FROM test1 WHERE
 t = $1",
                                                1,           /* one param
 */
                                                NULL,        /* let the
 backend deduce param type */
                                                paramValues,
                                                NULL,        /* don't need
 param lengths since text */
                                                NULL,        /* default to
 all text params */
                                                1);          /* ask for
 binary results */

        if (PQresultStatus(res) != PGRES_TUPLES_OK)
        {
                fprintf(stderr, "SELECT failed: %s", PQerrorMessage(
conn));
```

```
                PQclear(res);
                exit_nicely(conn);
        }

        /* Use PQfnumber to avoid assumptions about field order in
result */
        i_fnum = PQfnumber(res, "i");
        t_fnum = PQfnumber(res, "t");
        b_fnum = PQfnumber(res, "b");

        for (i = 0; i < PQntuples(res); i++)
        {
                char       *iptr;
                char       *tptr;
                char       *bptr;
                int                     blen;
                int                     ival;

                /* Get the field values (we ignore possibility they
are null!) */
                iptr = PQgetvalue(res, i, i_fnum);
                tptr = PQgetvalue(res, i, t_fnum);
                bptr = PQgetvalue(res, i, b_fnum);

                /*
                 * The binary representation of INT4 is in network
byte order,
                 * which we'd better coerce to the local byte order.
                 */
                ival = ntohl(*((uint32_t *) iptr));

                /*
                 * The binary representation of TEXT is, well, text,
and since
                 * libpq was nice enough to append a zero byte to it,
it'll work
                 * just fine as a C string.
                 *
                 * The binary representation of BYTEA is a bunch of
bytes, which
                 * could include embedded nulls so we have to pay
attention to
                 * field length.
                 */
                blen = PQgetlength(res, i, b_fnum);

                printf("tuple %d: got\n", i);
                printf(" i = (%d bytes) %d\n",
                           PQgetlength(res, i, i_fnum), ival);
                printf(" t = (%d bytes) '%s'\n",
                           PQgetlength(res, i, t_fnum), tptr);
                printf(" b = (%d bytes) ", blen);
                for (j = 0; j < blen; j++)
                        printf("\\%03o", bptr[j]);
```

```
                printf("\n\n");
        }

        PQclear(res);

        /* close the connection to the database and cleanup */
        PQfinish(conn);

        return 0;
}
```

第 28 章 大对象

在 7.1 之前的 PostgreSQL 版本中，数据库中任何一行的大小都不能超过一个数据页的大小。由于数据页的大小是 8192 字节（默认值，最高可以提高到 32768），数据值大小的上限相对较低。为了支持更大的原子值的存储，PostgreSQL 过去提供了、而且现在仍然提供大对象接口。该接口以面向文件的方式访问存储在特殊的大对象结构中的用户数据。

本章介绍PostgreSQL大对象数据的实现，以及相应的编程接口和查询语言接口。本章示例使用 libpq C 库，但大多数PostgreSQL原生编程接口都支持等效的功能。其他接口也可能在内部使用大对象接口来为大值提供通用支持，这里不作说明。

28.1. 历史

PostgreSQL的间接前身 POSTGRES 4.2支持三种标准的大对象实现：作为POSTGRES服务器外部的文件、作为由 POSTGRES服务器管理的外部文件，以及作为存储在POSTGRES数据库内部的数据。这在用户中造成了相当大的混乱。因此，PostgreSQL 中只保留了把大对象作为数据库内部存储数据的支持。虽然这样访问速度较慢，但它提供了更严格的数据完整性。由于历史原因，这种存储方案被称为Inversion 大对象。（你会看到 Inversion 一词偶尔也被用来表示大对象的同义词。）从 PostgreSQL 7.1开始，所有大对象都存放在一个名为`pg_largeobject`的系统表中。

PostgreSQL 7.1 引入了一种机制（绰号“TOAST”），它允许数据行远大于单个数据页。这使大对象接口在一定程度上过时了。大对象接口剩下的一个优点是它允许最大 2 GB 的值，而 TOAST 只能处理 1 GB。

28.2. 实现特性

大对象实现会把大对象拆分成若干“块”，并将这些块存储在数据库行中。一个 B-树索引保证在执行随机访问读写时，能够快速找到正确的块号。

28.3. 客户端接口

本节描述 PostgreSQL 客户端接口库为访问大对象所提供的功能。使用这些函数对大对象进行的所有操作必须发生在一个 SQL 事务块内。（从 PostgreSQL 6.5 开始，这一要求被严格强制执行；不过在更早的版本中它一直是隐含的要求，如果忽视就会导致错误行为。）PostgreSQL 的大对象接口是仿照 Unix 文件系统接口设计的，提供了与 `open`、`read`、`write`、`lseek` 等相对应的操作。

在 libpq 中使用大对象接口的客户端应用应包含头文件 `libpq/libpq-fs.h` 并与 libpq 库链接。

28.3.1. 创建一个大对象

函数

```
Oid lo_creat(PGconn *conn, int mode);
```

创建一个新的大对象。*mode* 是一个位掩码，描述新大对象的若干不同属性。这里列出的符号常量定义在头文件 `libpq/libpq-fs.h` 中。访问类型（读、写或读写）由把 `INV_READ` 和 `INV_WRITE` 位"或"在一起来控制。历史上，掩码的低十六位在伯克利被用来指定大对象应驻留的存储管理器编号。现在这些位应当始终为零。返回值就是分配给新大对象的 OID。

例如：

```
inv_oid = lo_creat(INV_READ|INV_WRITE);
```

28.3.2. 导入一个大对象

要把一个操作系统文件导入为大对象，调用

```
Oid lo_import(PGconn *conn, const char *filename);
```

filename 指定要导入为大对象的文件的操作系统名称。返回值就是分配给新大对象的 OID。

28.3.3. 导出一个大对象

要把一个大对象导出到操作系统文件，调用

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

lobjId 参数指定要导出的大对象的 OID，*filename* 参数指定文件的操作系统名称。

28.3.4. 打开一个现有的大对象

要打开一个现有的大对象，调用

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

lobjId 参数指定要打开的大对象的 OID。*mode* 位控制对象是以读（`INV_READ`）、写（`INV_WRITE`）还是读写方式打开。大对象在创建之前无法打开。`lo_open` 返回一个大对象描述符，供稍后在 `lo_read`、`lo_write`、`lo_lseek`、`lo_tell` 和 `lo_close` 中使用。该描述符只在当前事务期间有效。

28.3.5. 向大对象写入数据

函数

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

把 *len* 字节从 *buf* 写入大对象 *fd*。*fd* 参数必须是先前某个 `lo_open` 返回的。返回值是实际写入的字节数。发生错误时，返回值为负。

28.3.6. 从大对象读取数据

函数

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

从大对象 *fd* 中读取 *len* 字节到 *buf*。*fd* 参数必须是先前某个 `lo_open` 返回的。返回值是实际读取的字节数。发生错误时，返回值为负。

28.3.7. 在大对象上定位

要更改大对象上当前的读或写位置，调用

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

此函数把 *fd* 所描述大对象的当前位置指针移动到 *offset* 指定的新位置。*whence* 的有效值是 SEEK_SET（从对象开头定位）、SEEK_CUR（从当前位置定位）和 SEEK_END（从对象末尾定位）。返回值是新的位置指针。

28.3.8. 获取大对象的当前位置

要获取大对象当前的读或写位置，调用

```
int lo_tell(PGconn *conn, int fd);
```

发生错误时，返回值为负。

28.3.9. 关闭一个大对象描述符

可以调用

```
int lo_close(PGconn *conn, int fd);
```

来关闭一个大对象描述符，其中 *fd* 是由 lo_open 返回的大对象描述符。成功时，lo_close 返回零。出错时，返回值为负。

任何在事务结束时仍保持打开的大对象描述符都会被自动关闭。

28.3.10. 移除一个大对象

要从数据库中移除一个大对象，调用

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

lobjId 参数指定要删除的大对象的 OID。发生错误时，返回值为负。

28.4. 服务器端函数

有两个内置的服务器端函数，lo_import 和 lo_export，用于大对象访问，它们可以在 SQL 命令中使用。下面是一个用法的例子：

```
CREATE TABLE image (
    name            text,
    raster          oid
);

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

这些函数使用数据库属主用户的权限读写服务器文件系统中的文件。因此，
它们的使用仅限于超级用户。（相比之下，
客户端的导入导出函数使用客户端程序的权限读写客户端文件系统中的文件。
它们的使用不受限制。）

28.5. 示例程序

例 28.1是一个示例程序，它展示了如何使用libpq中的大对象接口。程序中有些部分已经被注释掉，但仍保留在源代码中供读者参考。该程序也可以在源代码发行包的`src/test/examples/testlo.c`中找到。

例 28.1. 用 libpq 操作大对象的示例程序

```
/*--------------------------------------------------------------
 *
 * testlo.c--
 *    test using large objects with libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *--------------------------------------------------------------
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile
 *    import file "in_filename" into database as large object "lobj
Oid"
 *
 */
Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n", filename);
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
```

```
        fprintf(stderr, "can't create large object\n");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading large object\n");
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = ' ';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
```

```
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len -
 nwritten);
        nwritten += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile *    export large object "lobjOid" to file "out_
filename"
 *
 */
void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * create an inversion "object"
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
```

```
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT | O_WRONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n",
                filename);
    }

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing %s\n",
                    filename);
        }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_
filename\n",
                argv[0]);
        exit(1);
```

```
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made
 */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n",
 database);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);

    printf("importing file %s\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
/*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X's
\n");
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file %s\n", out_filename);
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

第 29 章 pgtcl - Tcl 绑定库

pgtcl 是一个 Tcl 包，供客户端程序与 PostgreSQL 服务器交互。它使 Tcl 脚本可以使用 libpq 的大部分功能。

29.1. 概述

表 29.1概述了 pgtcl 中可用的命令。这些命令将在后续页面中进一步描述。

表 29.1. pgtcl 命令

命令	描述
pg_connect	打开一个到服务器的连接
pg_disconnect	关闭一个到服务器的连接
pg_conndefaults	获取连接选项及其默认值
pg_exec	向服务器发送一条命令
pg_result	获取关于命令结果的信息
pg_select	循环遍历查询的结果
pg_execute	发送一条查询并可选地遍历其结果
pg_listen	设置或更改异步通知消息的回调
pg_on_connection_loss	设置或更改意外断开连接的回调
pg_lo_creat	创建一个大对象
pg_lo_open	打开一个大对象
pg_lo_close	关闭一个大对象
pg_lo_read	从大对象读取
pg_lo_write	写入大对象
pg_lo_lseek	在大对象中定位到某个位置
pg_lo_tell	返回大对象的当前定位位置
pg_lo_unlink	删除一个大对象
pg_lo_import	从文件导入一个大对象
pg_lo_export	将大对象导出到文件

pg_lo_* 命令是 PostgreSQL 大对象功能的接口。这些函数的设计模仿标准 Unix 文件系统接口中类似的文件系统函数。pg_lo_* 命令应当在一个 BEGIN/COMMIT 事务块中使用，因为 pg_lo_open 返回的描述符只在当前事务中有效。pg_lo_import 和 pg_lo_export 必须在一个 BEGIN/COMMIT 事务块中使用。

29.2. 将 pgtcl 载入应用程序

在使用 pgtcl 命令之前，你必须把 libpgtcl 库载入你的 Tcl 应用程序。这通常用 Tcl 的 load 命令完成。下面是一个例子：

```
load libpgtcl[info sharedlibextension]
```

推荐使用 info sharedlibextension，而不是在程序中硬编码 .so 或 .sl。

除非系统的动态载入器知道到哪里寻找 `libpgtcl` 共享库文件，否则 `load` 命令会失败。你可能需要使用 `ldconfig`，或者设置环境变量 `LD_LIBRARY_PATH`，或者使用你的平台上的一些等价设施才能让它工作。更多信息请参阅 PostgreSQL 的安装说明。

`libpgtcl` 又依赖于 `libpq`，因此动态载入器还必须能够找到 `libpq` 共享库。在实践中这很少成为问题，因为这两个共享库通常存放在同一个目录中，但在某些配置下它可能成为一个绊脚石。

如果你的应用程序使用自定义的可执行文件，你可以选择把 `libpgtcl` 静态绑定到可执行文件中，从而避免 `load` 命令以及动态链接的潜在问题。一个例子参见 pgtclsh 的源代码。

29.3. pgtcl 命令参考

pg_connect

pg_connect — 打开一个到服务器的连接

大纲

```
pg_connect -conninfo connectOptions
pg_connect dbName ?-host hostName? ?-port portNumber? ?-tty tty? ?-
options serverOptions?
```

描述

`pg_connect` 打开一个到 PostgreSQL 服务器的连接。

有两种可用的语法。在较老的一种中，每个可能的选项在 `pg_connect` 命令中都有一个单独的选项开关。在较新的一种中，提供的是单个选项字符串，其中可以包含多个选项值。可以用 `pg_conndefaults` 获取关于较新语法中可用选项的信息。

参数

新风格

connectOptions

一个连接选项字符串，每个选项都写成 `keyword = value` 的形式。有效选项的列表可以在 libpq 函数 `PQconnectdb` 的描述中找到。

旧风格

dbName

要连接的数据库名。

-host *hostName*

要连接的数据库服务器的主机名。

-port *portNumber*

要连接的数据库服务器的 TCP 端口号。

-tty *tty*

用于服务器可选调试输出的文件或 TTY。

-options *serverOptions*

要传递给服务器的额外配置选项。

返回值

如果成功，返回一个数据库连接的句柄。句柄以 `pgsql` 前缀开头。

pg_disconnect

pg_disconnect — 关闭一个到服务器的连接

大纲

`pg_disconnect` *`conn`*

描述

`pg_disconnect` 关闭一个到 PostgreSQL 服务器的连接。

参数

`conn`

要关闭的连接的句柄。

返回值

无

pg_conndefaults

pg_conndefaults — 获取连接选项及其默认值

大纲

```
pg_conndefaults
```

描述

`pg_conndefaults` 返回关于 `pg_connect -conninfo` 中可用的连接选项的信息以及每个选项的当前默认值。

参数

无

返回值

结果是一个列表，描述可能的连接选项及其当前默认值。
列表中的每个条目都是一个如下格式的子列表：

```
{optname label dispchar dispsize value}
```

其中 *optname* 可用作 `pg_connect -conninfo` 的一个选项。

pg_exec

pg_exec — 向服务器发送一条命令

大纲

`pg_exec` *`conn commandString`*

描述

`pg_exec` 向 PostgreSQL 服务器提交一条命令并返回一个结果。命令结果句柄以连接句柄开头，再加上一个句点和一个结果编号。

注意，没有发生 Tcl 错误并不能证明命令成功了！服务器返回的错误消息会被作为带有失败状态的命令结果来处理，而不是让 `pg_exec` 产生一个 Tcl 错误。

参数

`conn`

要在其上执行命令的连接的句柄。

`commandString`

要执行的 SQL 命令。

返回值

一个结果句柄。如果 pgtcl 无法获得服务器响应，将返回一个 Tcl 错误。否则，会创建一个命令结果对象并返回它的一个句柄。这个句柄可以传递给 `pg_result` 以获取该命令的结果。

pg_result

pg_result — 获取关于命令结果的信息

大纲

`pg_result` *`resultHandle resultOption`*

描述

`pg_result` 返回关于先前 `pg_exec` 所创建命令结果的信息。

你可以根据需要把一个命令结果保留任意长的时间，但用完之后，务必通过执行 `pg_result -clear` 来释放它。否则就会发生内存泄漏，pgtcl 最终会开始抱怨你创建了太多命令结果对象。

参数

`resultHandle`

命令结果的句柄。

`resultOption`

下列选项之一，指定要返回哪部分结果信息：

`-status`

结果的状态。

`-error`

错误消息，如果状态表明是一个错误，否则为空字符串。

`-conn`

产生该结果的连接。

`-oid`

如果命令是一条 `INSERT`，则为被插入行的 OID，否则为 0。

`-numTuples`

查询返回的行（元组）数。

`-cmdTuples`

被该命令影响的行（元组）数。

`-numAttrs`

每行中的列（属性）数。

`-assign` *`arrayName`*

把结果赋给一个数组，使用形如 `(rowNumber, columnName)` 的下标。

`-assignbyidx` *`arrayName`* `?`*`appendstr`*`?`

使用第一列的值和其余列的名称作为键，把结果赋给一个数组。如果给定了 *`appendstr`*，则把它追加到每个键之后。简而言之，除了每行的第一列之外的所有列都会被存入数组，使用形如 `(firstColumnValue, columnNameAppendStr)` 的下标。

`-getTuple` *`rowNumber`*

以列表形式返回指定行的各列。行号从零开始。

`-tupleArray` *`rowNumber arrayName`*

把该行的各列存入数组 *`arrayName`*，以列名作为索引。行号从零开始。

`-attributes`

返回结果中各列名称的列表。

`-lAttributes`

返回一个子列表的列表，每列对应 `{name typeOid typeSize}`。

`-clear`

清除命令结果对象。

返回值

结果取决于所选的选项，如上所述。

pg_select

pg_select — 循环遍历查询的结果

大纲

`pg_select` *`conn commandString arrayVar procedure`*

描述

`pg_select` 向 PostgreSQL 服务器提交一条查询（`SELECT` 语句），并对结果中的每一行执行给定的代码块。*`commandString`* 必须是一条 `SELECT` 语句；任何其他内容都会返回一个错误。*`arrayVar`* 变量是循环中使用的一个数组名。对于每一行，*`arrayVar`* 都会用该行的值填充，以列名作为数组下标。然后执行 *`procedure`*。

除了列值之外，数组中还会生成下列特殊条目：

`.headers`

查询返回的列名的列表。

`.numcols`

查询返回的列数。

`.tupno`

当前行号，从零开始，循环体每迭代一次递增一次。

参数

`conn`

要在其上执行查询的连接的句柄。

`commandString`

要执行的 SQL 查询。

`arrayVar`

用于返回行的数组变量。

`procedure`

为每个返回的行运行的过程。

返回值

无

示例

这个例子假定表 `table1` 具有列 `control` 和 `name`（还可能有其他列）：

```
pg_select $pgconn "SELECT * FROM table1;" array {
    puts [format "%5d %s" $array(control) $array(name)]
}
```

pg_execute

pg_execute — 发送一条查询并可选地遍历其结果

大纲

```
pg_execute ?-array arrayVar? ?-oid oidVar? conn commandString
 ?procedure?
```

描述

`pg_execute` 向 PostgreSQL 服务器提交一条命令。

如果该命令不是 `SELECT` 语句，则返回被该命令影响的行数。如果该命令是一条 `INSERT` 语句且只插入了一行，那么在提供了可选的 *-oid* 参数时，被插入行的 OID 会被存入变量 *oidVar*。

如果该命令是一条 `SELECT` 语句，那么对于结果中的每一行，如果提供了 *arrayVar* 变量，行值就会被存入该变量，并以列名作为数组下标，否则存入以列名命名的变量中，然后如果提供了可选的 *procedure*，则执行它。（只有当查询将只返回一行时，省略 *procedure* 才大概是合理的。）返回被选中的行数。

procedure 可以使用 Tcl 命令 `break`、`continue` 和 `return`，其行为符合预期。注意，如果 *procedure* 执行了 `return`，那么 `pg_execute` 不会返回受影响的行数。

`pg_execute` 是一个较新的函数，它提供了 `pg_select` 功能的一个超集，并且在许多不需要访问结果句柄的场合可以取代 `pg_exec`。

对于由服务器处理的错误，`pg_execute` 会抛出一个 Tcl 错误并返回一个双元素列表。第一个元素是一个错误码，例如 `PGRES_FATAL_ERROR`，第二个元素是服务器的错误文本。对于更严重的错误，例如无法与服务器通信，`pg_execute` 会抛出一个 Tcl 错误并只返回错误消息文本。

参数

`-array` *arrayVar*

指定一个数组变量的名称，结果行将被存入其中，以列名作为索引。如果 *commandString* 不是 `SELECT` 语句，则忽略此参数。

`-oid` *oidVar*

指定一个变量的名称，`INSERT` 语句产生的 OID 将被存入该变量。

conn

要在其上执行命令的连接的句柄。

commandString

要执行的 SQL 命令。

procedure

可选的过程，为 `SELECT` 语句的每个结果行执行。

返回值

被该命令影响或返回的行数。

示例

在下面的例子中，为了清晰起见，省略了用 `catch` 进行的错误检查。

插入一行并把 OID 保存到 `result_oid` 中：

```
pg_execute -oid result_oid $pgconn "INSERT INTO mytable VALUES (1);"
```

打印每一行中的列 `item` 和 `value`：

```
pg_execute -array d $pgconn "SELECT item, value FROM mytable;" {
    puts "Item=$d(item) Value=$d(value)"
}
```

找出最大值和最小值，并把它们存入 `$s(max)` 和 `$s(min)`：

```
pg_execute -array s $pgconn "SELECT max(value) AS max, min(value) AS
 min FROM mytable;"
```

找出最大值和最小值，并把它们存入 `$max` 和 `$min`：

```
pg_execute $pgconn "SELECT max(value) AS max, min(value) AS min FROM
 mytable;"
```

pg_listen

pg_listen — 设置或更改异步通知消息的回调

大纲

`pg_listen` *`conn notifyName`* ?*`callbackCommand`*?

描述

`pg_listen` 创建、更改或取消一个监听来自 PostgreSQL 服务器的异步通知消息的请求。带有 *`callbackCommand`* 参数时，会建立该请求，或者替换一个已存在请求的命令字符串。不带 *`callbackCommand`* 参数时，则取消先前的请求。

在 `pg_listen` 请求建立之后，每当从服务器收到带有给定名称的通知消息时，就会执行指定的命令字符串。这发生在任何 PostgreSQL 客户端应用发出一条引用该名称的 `NOTIFY` 命令时。命令字符串从 Tcl 空闲循环中执行。那是用 Tk 编写的应用程序的正常空闲状态。在非 Tk 的 Tcl shell 中，你可以执行 `update` 或 `vwait` 来进入空闲循环。

在使用 `pg_listen` 时，你不应直接调用 SQL 语句 `LISTEN` 或 `UNLISTEN`。pgtcl 会替你负责发出这些语句。但如果你想自己发送一条通知消息，可以用 `pg_exec` 调用 SQL `NOTIFY` 语句。

参数

`conn`

要在其上监听通知的连接的句柄。

`notifyName`

要开始或停止监听的通知条件的名称。

`callbackCommand`

如果提供，给出在收到匹配的通知时执行的命令字符串。

返回值

无

pg_on_connection_loss

pg_on_connection_loss — 设置或更改意外断开连接的回调

大纲

`pg_on_connection_loss` *`conn`* ?*`callbackCommand`*?

描述

`pg_on_connection_loss` 创建、更改或取消一个请求：在发生到数据库的连接意外丢失时执行一个回调命令。带有 *`callbackCommand`* 参数时，会建立该请求，或者替换一个已存在请求的命令字符串。不带 *`callbackCommand`* 参数时，则取消先前的请求。

回调命令字符串从 Tcl 空闲循环中执行。那是用 Tk 编写的应用程序的正常空闲状态。在非 Tk 的 Tcl shell 中，你可以执行 `update` 或 `vwait` 来进入空闲循环。

参数

`conn`

要监视其连接丢失的句柄。

`callbackCommand`

如果提供，给出在检测到连接丢失时执行的命令字符串。

返回值

无

pg_lo_creat

pg_lo_creat — 创建一个大对象

大纲

`pg_lo_creat` *`conn mode`*

描述

`pg_lo_creat` 创建一个大对象。

参数

`conn`

到要在其中创建大对象的数据库的连接句柄。

`mode`

大对象的访问模式。它可以是 `INV_READ` 和 `INV_WRITE` 的任意按位或组合。“或”操作符是 `|`。例如：

```
[pg_lo_creat $conn "INV_READ|INV_WRITE"]
```

返回值

所创建的大对象的 OID。

pg_lo_open

pg_lo_open — 打开一个大对象

大纲

`pg_lo_open` *`conn loid mode`*

描述

`pg_lo_open` 打开一个大对象。

参数

`conn`

到大对象所在的数据库的连接句柄。

`loid`

大对象的 OID。

`mode`

指定大对象的访问模式。模式可以是 `r`、`w` 或 `rw`。

返回值

一个供后续大对象命令使用的描述符。

pg_lo_close

pg_lo_close — 关闭一个大对象

大纲

`pg_lo_close` *`conn descriptor`*

描述

`pg_lo_close` 关闭一个大对象。

参数

`conn`

到大对象所在的数据库的连接句柄。

`descriptor`

来自 `pg_lo_open` 的大对象描述符。

返回值

无

pg_lo_read

pg_lo_read — 从大对象读取

大纲

`pg_lo_read` *`conn descriptor bufVar len`*

描述

`pg_lo_read` 从大对象中最多读取 *`len`* 个字节，存入名为 *`bufVar`* 的变量。

参数

`conn`

到大对象所在的数据库的连接句柄。

`descriptor`

来自 `pg_lo_open` 的大对象描述符。

`bufVar`

用于容纳大对象片段的缓冲区变量的名称。

`len`

要读取的最大字节数。

返回值

返回实际读取的字节数；如果先到达了大对象的末尾，它可能小于请求的数量。发生错误时，返回值为负数。

pg_lo_write

pg_lo_write — 写入大对象

大纲

`pg_lo_write` *`conn descriptor buf len`*

描述

`pg_lo_write` 从变量 *`buf`* 中最多向大对象写入 *`len`* 个字节。

参数

`conn`

到大对象所在的数据库的连接句柄。

`descriptor`

来自 `pg_lo_open` 的大对象描述符。

`buf`

要写入大对象的字符串（不是变量名，而是值本身）。

`len`

要写入的最大字节数。写入的数量将是该值与字符串长度中较小的那个。

返回值

返回实际写入的字节数；通常与请求的数量相同。发生错误时，返回值为负数。

pg_lo_lseek

pg_lo_lseek — 在大对象中定位到某个位置

大纲

`pg_lo_lseek` *`conn descriptor offset whence`*

描述

`pg_lo_lseek` 把当前读/写位置移动到从 *`whence`* 指定的位置起 *`offset`* 字节处。

参数

`conn`

到大对象所在的数据库的连接句柄。

`descriptor`

来自 `pg_lo_open` 的大对象描述符。

`offset`

新的定位位置，以字节为单位。

`whence`

指定从哪里开始计算新的定位位置：`SEEK_CUR`（从当前位置）、`SEEK_END`（从末尾）或 `SEEK_SET`（从开头）。

返回值

无

pg_lo_tell

pg_lo_tell — 返回大对象的当前定位位置

大纲

`pg_lo_tell` *`conn descriptor`*

描述

`pg_lo_tell` 返回从大对象开头起的当前读/写位置（以字节为单位）。

参数

`conn`

到大对象所在的数据库的连接句柄。

`descriptor`

来自 `pg_lo_open` 的大对象描述符。

返回值

一个从零开始的字节偏移量，适合作为 `pg_lo_lseek` 的输入。

pg_lo_unlink

pg_lo_unlink — 删除一个大对象

大纲

`pg_lo_unlink` *`conn loid`*

描述

`pg_lo_unlink` 删除指定的大对象。

参数

`conn`

到大对象所在的数据库的连接句柄。

`loid`

大对象的 OID。

返回值

无

pg_lo_import

pg_lo_import — 从文件导入一个大对象

大纲

`pg_lo_import` *`conn filename`*

描述

`pg_lo_import` 读取指定的文件并把内容放入一个新的大对象。

参数

`conn`

到要在其中创建大对象的数据库的连接句柄。

`filename`

指定要从中导入数据的文件。

返回值

所创建的大对象的 OID。

说明

必须在一个 `BEGIN/COMMIT` 事务块内调用 `pg_lo_import`。

pg_lo_export

pg_lo_export — 将大对象导出到文件

大纲

`pg_lo_export` *`conn loid filename`*

描述

`pg_lo_export` 把指定的大对象写入一个文件。

参数

`conn`

到大对象所在的数据库的连接句柄。

`loid`

大对象的 OID。

`filename`

指定要把数据导出到其中的文件。

返回值

无

说明

必须在一个 `BEGIN`/`COMMIT` 事务块内调用 `pg_lo_export`。

29.4. 示例程序

例 29.1展示了一个如何使用 pgtcl 命令的小例子。

例 29.1. pgtcl 示例程序

```
# getDBs :
#   get the names of all the databases at a given host and port number
#   with the defaults being the localhost and port 5432
#   return them in alphabetical order
proc getDBs { {host "localhost"} {port "5432"} } {
    # datnames is the list to be result
    set conn [pg_connect template1 -host $host -port $port]
    set res [pg_exec $conn "SELECT datname FROM pg_database ORDER BY
 datname;"]
    set ntups [pg_result $res -numTuples]
    for {set i 0} {$i < $ntups} {incr i} {
 lappend datnames [pg_result $res -getTuple $i]
    }
```

```
    pg_result $res -clear
    pg_disconnect $conn
    return $datnames
}
```

第 30 章 ECPG - C 中的嵌入式 SQL

本章介绍 PostgreSQL 的嵌入式 SQL 软件包。它可用于 C 和 C++。它由 Linus Tolke（<linus@epact.se>）和 Michael Meskes（<meskes@postgresql.org>）编写。

诚然，本文档相当不完整。但由于这个接口是标准化的，更多相关信息可以在许多关于 SQL 的资料中找到。

30.1. 概念

嵌入式 SQL 程序由用普通编程语言（这里是 C）编写的代码与特殊标记区中的 SQL 命令混合组成。要构建程序，源代码首先被交给嵌入式 SQL 预处理器，由它把代码转换成普通的 C 程序，之后就可以用 C 编译工具链处理。

与其他在 C 代码中处理SQL命令的方法相比，嵌入式SQL有其优势。首先，它能处理与你的C程序变量之间来回传递信息的繁琐工作。其次，程序中的 SQL 代码会在构建时接受语法正确性检查。第三，C 中的嵌入式SQL由SQL标准规定，并且得到许多其他SQL数据库系统的支持。PostgreSQL 的实现被设计为尽可能贴合该标准，因此通常可以比较容易地把为其他 SQL 数据库编写的嵌入式SQL程序移植到PostgreSQL。

如前所述，为嵌入式 SQL 接口编写的程序是插入了执行数据库相关动作的特殊代码的普通 C 程序。这种特殊代码总是具有这样的形式

```
EXEC SQL ...;
```

这些语句在语法上取代一条 C 语句的位置。根据具体语句的不同，它们可以出现在全局上下文中或函数内。嵌入式 SQL 语句遵循普通 SQL 代码而不是 C 的大小写敏感性规则。

下列小节解释了所有嵌入式 SQL 语句。

30.2. 连接到数据库服务器

使用以下语句连接数据库：

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

其中，*target* 可以按以下方式指定：

- *dbname*[@*hostname*][:*port*]
- tcp:postgresql://*hostname*[:*port*][/*dbname*][?*options*]
- unix:postgresql://*hostname*[:*port*][/*dbname*][?*options*]
- 包含上述某种形式的 SQL 字符串常量
- 对包含上述某种形式的字符变量的引用（见示例）
- DEFAULT

如果直接写出连接目标（即不通过变量引用），且未用引号括起该值，就会应用普通 SQL 的大小写不敏感规则。此时，也可以按需分别用双引号括起各个参数。实际使用中，
采用单引号括起的字符串字面值或变量引用，可能更不容易出错。连接目标 DEFAULT 会以默认用户名发起到默认数据库的连接，此时不能单独指定用户名或连接名。

也有不同的方法来指定用户名：

- *username*
- *username/password*
- *username* IDENTIFIED BY *password*
- *username* USING *password*

如上所述，参数*username*以及*password*可以是一个 SQL 标识符、一个 SQL 字符串常量或者一个对字符变量的引用。

*connection-name*被用来在一个程序中处理多个连接。如果一个程序只使用一个连接，则可以省略它。最近被打开的连接将成为当前连接，当一个 SQL 语句要被执行时，将默认使用它（见这一章稍后的部分）。

以下是一些 CONNECT 语句的示例：

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS
 myconnection USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :target USER :user;
```

最后一种形式使用了上面提到的字符变量引用。后续章节会说明，如何在 C 变量前加冒号，以便在 SQL 语句中使用它们。

请注意，连接目标的格式并未在 SQL 标准中规定。因此，如果你想开发可移植的应用，最好采用类似上面最后一个示例的方法，把连接目标字符串封装在某个地方。

30.3. 关闭一个连接

要关闭连接，请使用以下语句：

EXEC SQL DISCONNECT [*connection*];

其中，*connection* 可以按以下方式指定：

- *connection-name*
- DEFAULT
- CURRENT
- ALL

如果未指定连接名，则关闭当前连接。

应用程序始终显式断开其打开的每个连接，是良好的编程习惯。

30.4. 运行 SQL 命令

任何 SQL 命令都可以在一个嵌入式 SQL 应用中被运行。下面是一些在嵌入式 SQL 应用中运行 SQL 命令的例子。

创建一个表：

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

插入行：

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

删除行：

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

单行选择：

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

使用游标选择：

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

更新：

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

`:`*`something`*形式的词元是主变量，即它们指向 C 程序中的变量。其说明见第 30.6 节。

在默认模式下，只有在发出 `EXEC SQL COMMIT` 时语句才被提交。嵌入式 SQL 接口还支持事务自动提交（就像其他接口中那样），可以通过 `ecpg` 的 `-t` 命令行选项（见下文）或 `EXEC SQL SET AUTOCOMMIT TO ON` 语句启用。在自动提交模式下，除非处于显式的事务块中，每条命令都会自动提交。这个模式可以用 `EXEC SQL SET AUTOCOMMIT TO OFF` 显式关闭。

30.5. 选择一个连接

上一节中展示的 SQL 语句在当前连接（即最近打开的连接）上执行。如果一个应用需要管理多个连接，有两种处理方式。

第一种方式是显式地为每一个 SQL 语句选择一个连接，例如：

```
EXEC SQL AT connection-name SELECT ...;
```

如果应用需要以混合的顺序使用多个连接，这种方式特别合适。

第二种方式是执行一个语句来切换当前的连接。该语句是：

```
EXEC SQL SET CONNECTION connection-name;
```

如果有很多语句要在同一个连接上执行，这种方式尤其方便。

30.6. 使用主变量

在第 30.4 节中，你已经看到如何从嵌入式 SQL 程序中执行 SQL 语句。其中一些语句只使用固定值，既无法把用户提供的值插入语句中，也无法让程序处理查询返回的值。
这类语句在实际应用中并不真正有用。本节将详细解释如何使用一种称为主变量的简单机制，在 C 程序与嵌入式 SQL 语句之间传递数据。

30.6.1. 概述

在嵌入式 SQL 中，在 C 程序和 SQL 语句之间传递数据特别简单。我们不必让程序把数据拼接进语句中去，那样会带来诸如正确为值加引号等各种复杂问题；只需在 SQL 语句中写出 C 变量的名称，并在前面加上一个冒号即可。例如：

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

该语句引用了两个名为`v1`和`v2`的 C 变量，同时还使用了一个普通的 SQL 字符串常量，以说明你并不局限于只使用某一种数据。

这种在 SQL 语句中插入 C 变量的方式可以用在 SQL 语句中每一个应该出现值表达式的地方。在 SQL 环境中，我们把对 C 变量的引用称为主变量。

30.6.2. 声明节

要把数据从程序传给数据库（例如作为查询参数），或者把数据从数据库传回程序，
用来承载这些数据的 C 变量必须声明在经过特殊标记的区段中，这样嵌入式 SQL 预处理器才能识别它们。

这个区段始于：

```
EXEC SQL BEGIN DECLARE SECTION;
```

并结束于：

```
EXEC SQL END DECLARE SECTION;
```

在这两行之间，必须是正常的 C 变量声明，例如：

```
int   x;
char  foo[16], bar[16];
```

一个程序中可以根据需要放置任意多个声明节。

这些声明也会作为普通的 C 变量回显到输出文件中，所以不需要再次声明它们。不打算与 SQL 命令一起使用的变量可以在这些特殊区之外正常声明。

结构体或联合的定义也必须写在DECLARE声明节中。否则预处理器无法处理这些类型，因为它不知道它们的定义。

特殊类型VARCHAR会被转换成每个变量的一个命名的struct。像这样的声明：

```
VARCHAR var[180];
```

会被转换为：

```
struct varchar_var { int len; char arr[180]; } var;
```

这个结构适合与varchar类型的 SQL 数据交互。

30.6.3. SELECT INTO 和 FETCH INTO

现在你应该能够把程序产生的数据传递到一个 SQL 命令中了。但是怎么检索一个查询的结果呢？为此，嵌入式 SQL 提供了常规命令SELECT和FETCH的特殊变体。这些命令有一个特殊的INTO子句，它指定被检索到的值要被存储在哪些主变量中。

这里是一个例子：

```
/*
 * assume this table:
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

那么INTO子句出现在选择列表和FROM子句之间。选择列表中的元素数量必须和INTO后面列表（也被称为目标列表）的元素数量相等。

这里有一个使用命令FETCH的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do
{
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
```

```
    ...
} while (...);
```

这里INTO子句出现在所有正常子句的后面。

上面这两种方法每次都只能检索单行数据。如果需要处理可能包含多于一行的结果集，就需要使用游标，如第二个示例所示。

30.6.4. 指示符

上面的例子没有处理空值。事实上，检索例子如果从数据库中取出空值就会报错。为了能够向数据库传递空值或从数据库检索空值，你需要为每个包含数据的宿主变量追加第二个宿主变量说明。这第二个宿主变量称为指示符，它包含一个标志，指出数据是否为空，若为空则忽略真实宿主变量的值。下面是一个正确处理空值检索的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

如果值不为空，指示符变量 val_ind 将为零，如果值为空则它为负数。

指示符有另一种功能：如果指示符值为正，它表示值不为空，但是当它被存储在主变量中时已被截断。

30.7. 动态 SQL

在很多情况中，一个应用必须要执行的特定 SQL 语句在编写该应用时就已知。不过在某些情况中，SQL 语句在运行时构造或者由一个外部来源提供。这样你就不能直接把 SQL 语句嵌入到 C 源代码，不过有一种功能允许你调用在一个字符串变量中提供的任意 SQL 语句。

执行一个任意 SQL 语句的最简单方法是使用命令EXECUTE IMMEDIATE。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

你不能用这种方法执行检索数据的语句（例如SELECT）。

执行任意 SQL 语句的一种更强大的方法是一次预备它们，然后随意多次执行预备好的语句。也可以预备某个语句的通用版本，然后通过替换参数执行它的特定版本。预备语句时，在以后想替换参数的位置写上问号。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
```

```
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

If the statement you are executing returns values, then add an `INTO` clause:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3;
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO v1, v2, v3 USING 37;
```

An `EXECUTE` command may have an `INTO` clause, a `USING` clause, both, or neither.

当你不再需要该预备语句时，你应该释放它：

```
EXEC SQL DEALLOCATE PREPARE name;
```

30.8. 使用描述符区域

SQL 描述符区域是一种处理`SELECT`、`FETCH`或 DESCRIBE 语句结果的高级方法。SQL 描述符区域将一行数据及其元数据项组合到一个数据结构中。在执行动态 SQL 语句时，结果列的性质可能无法提前获知，此时元数据特别有用。PostgreSQL 提供两种使用描述符区域的方式：命名 SQL 描述符区域和 C 结构体 SQLDA。

一个命名 SQL 描述符区域由一个头部以及一个或多个条目描述符区域构成，头部包含与整个描述符相关的信息，而条目描述符区域则描述结果行中的每一列。

在使用 SQL 描述符区之前，你需要先分配一个：

```
EXEC SQL ALLOCATE DESCRIPTOR identifier;
```

该标识符充当描述符区的“变量名”。不再需要描述符时，应当释放它：

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier;
```

要使用一个描述符区域，把它指定为`INTO`子句的存储目标（而不是列出主变量）：

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

如果结果集为空，该描述符区域仍然会包含查询的元数据，即字段的名称。

那么如何从描述符区域中取得数据呢？你可以把描述符区域当作一个包含命名字段的结构。
要从头部取得某个字段的值并存入一个主变量，可使用以下命令：

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

目前只定义了一个头部字段：*`COUNT`*，
它表示存在多少个条目描述符区域（即结果中包含多少列）。主变量需要是整数类型。
要从条目描述符区域取得一个字段，可使用以下命令：

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

*`num`*可以是一个整数字面量或者一个包含整数的宿主变量。可能的字段有：

`CARDINALITY` (integer)

结果集中的行数

`DATA`

实际的数据项（因此，这个字段的数据类型取决于查询）

`DATETIME_INTERVAL_CODE` (integer)

?

`DATETIME_INTERVAL_PRECISION` (integer)

没有实现

`INDICATOR` (integer)

指示符（表示一个空值或者一个值截断）

`KEY_MEMBER` (integer)

没有实现

`LENGTH` (integer)

以字符计的数据长度

`NAME` (string)

列名

`NULLABLE` (integer)

没有实现

`OCTET_LENGTH` (integer)

以字节计的数据字符表示的长度

`PRECISION` (integer)

精度（用于类型`numeric`）

`RETURNED_LENGTH` (integer)

以字符计的数据长度

`RETURNED_OCTET_LENGTH` (integer)

以字节计的数据字符表示的长度

`SCALE` (integer)

小数位数（用于类型 `numeric`）

`TYPE` (integer)

列的数据类型的数字编码

30.9. 错误处理

本节描述如何在嵌入式 SQL 程序中处理异常情况和警告。为此有几种可以同时使用的机制。

30.9.1. 设置回调

捕获错误和警告的一种简单方法是指定一个动作，在特定条件发生时执行。一般形式如下：

`EXEC SQL WHENEVER condition action;`

*condition*可以是以下值之一：

`SQLERROR`

只要在 SQL 语句执行期间发生一个错误就调用指定的动作。

`SQLWARNING`

只要在 SQL 语句执行期间发生一个警告就调用指定的动作。

`NOT FOUND`

只要一个 SQL 语句检索或者影响零行就调用指定的动作（这种情况不是一个错误，但是你可能需要特别地处理它）。

*action*可以是以下值之一：

`CONTINUE`

这实际上表示该情况被忽略。这是默认值。

`GOTO label`
`GO TO label`

跳到指定的标签（使用一个 C `goto`语句）。

`SQLPRINT`

把一个消息打印到标准错误。这对简单程序或原型开发很有用。消息的细节无法配置。

`STOP`

调用`exit(1)`终止程序。

`BREAK`

执行 C 语句`break`。只应被用在循环或`switch`语句中。

`CALL name (args)`
`DO name (args)`

使用指定参数调用指定的 C 函数。

SQL 标准只规定了以下动作：`CONTINUE` 和 `GOTO`（以及 `GO TO`）。

这里有一个可能会用在简单程序中的例子。当一个警告发生时它打印一个简单消息，
而发生一个错误时它会中止程序：

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

语句`EXEC SQL WHENEVER`是 SQL 预处理器的指令，而不是 C 语句。它设置的错误或警告动作适用于源代码中出现在处理器设置位置之后的所有嵌入式 SQL 语句，除非在第一个`EXEC SQL WHENEVER`与触发该条件的 SQL 语句之间，为同一条件设置了其他动作。这与 C 程序的控制流无关。因此，下面两个 C 程序片段都不会产生预期效果：

```
/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}
```

```
/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

30.9.2. sqlca

对于更强大的错误处理，嵌入式 SQL 接口提供了一个名为 `sqlca` 的全局变量，它具有下面的结构：

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[70];
    } sqlerrm;
```

```
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

（在多线程程序中，每个线程自动获得自己的 sqlca 副本。这与标准 C 全局变量 errno 的处理方式类似。）

sqlca覆盖了警告和错误。如果执行一个语句时发生了多个警告或错误，那么sqlca将只包含关于最后一个的信息。

如果在上一个SQL语句中没有产生错误，sqlca.sqlcode将为 0 并且sqlca.sqlstate将为"00000"。如果发生一个警告或错误，则sqlca.sqlcode将为负并且sqlca.sqlstate将不为"00000"。一个正的sqlca.sqlcode表示一种无害的情况，例如上一个查询返回零行。sqlcode和sqlstate是两种不同的错误编码方案，详见下文。

如果上一个 SQL 语句成功，则sqlca.sqlerrd[1]包含所处理行的 OID（如果适用），sqlca.sqlerrd[2]包含处理或返回的行数（如果适用于该命令）。

出现错误或警告时，sqlca.sqlerrm.sqlerrmc 将包含一个描述错误的字符串。字段 sqlca.sqlerrm.sqlerrml 包含存储在 sqlca.sqlerrm.sqlerrmc 中的错误消息的长度（即 strlen() 的结果，对 C 程序员来说没什么意思）。

发生警告时，sqlca.sqlwarn[2]被设为W（其他情况下，它被设为非W的值）。如果sqlca.sqlwarn[1]被设为W，说明某个值在存入主变量时被截断了。
如果其他任一元素被设置为指示警告，sqlca.sqlwarn[0]就会被设为W。

字段sqlcaid、sqlabc、sqlerrp，以及sqlerrd和sqlwarn的其余元素，目前不包含有用信息。

结构 sqlca 未在 SQL 标准中定义，但其他几个 SQL 数据库系统也实现了它。这些定义的核心部分相似，但如果想编写可移植的应用，就应当仔细研究各个不同的实现。

30.9.3. SQLSTATE 与 SQLCODE

字段sqlca.sqlstate和sqlca.sqlcode采用两种不同的错误编码方案。两者都在 SQL 标准中定义，但SQLCODE在标准的 1992 版本中已被标记为弃用，并在 1999 版本中被删除。因此，强烈建议新应用使用SQLSTATE。

SQLSTATE是一个包含五个字符的数组。这五个字符由数字或大写字母组成，表示各种错误和警告条件的代码。SQLSTATE采用分层编码方案：前两个字符表示条件所属的大类，后三个字符表示该大类中的子类。代码00000表示成功状态。大部分SQLSTATE代码由 SQL 标准定义。PostgreSQL服务器原生支持SQLSTATE错误码，因此在所有应用中统一使用这种错误编码方案，可以实现较高的一致性。更多信息见附录 A。

SQLCODE 是已废弃的错误代码方案，它是一个简单的整数。值 0 表示成功，正值表示成功并带有附加信息，负值表示错误。SQL 标准只定义了正值 +100（表示上一条命令返回或影响了零行），没有定义具体的负值。因此这个方案的可移植性很差，也没有层次化的代码分配。历史上，PostgreSQL 的嵌入式 SQL 处理器为它自己指定了一些 SQLCODE 值，下面列出了这些值的数值和符号名。请记住它们不能移植到其他 SQL 实现。为了便于把应用移植到 SQLSTATE 方案，还列出了对应的 SQLSTATE。但这两个方案之间没有一对一或一对多映射（实际上是多对多），因此每种情况都应当查阅附录 A中的全局 SQLSTATE 列表。

下面是这些被分配的 `SQLCODE` 值：

-12 (`ECPG_OUT_OF_MEMORY`)

表示你的虚拟内存已被耗尽。（SQLSTATE YE001）

-200 (`ECPG_UNSUPPORTED`)

表示预处理器生成了该库无法识别的内容。
可能是所用的预处理器与库的版本不兼容（SQLSTATE YE002）。

-201 (`ECPG_TOO_MANY_ARGUMENTS`)

这表示命令指定了超过该命令预期数量的主变量（SQLSTATE 07001 或 07002）。

-202 (`ECPG_TOO_FEW_ARGUMENTS`)

这表示命令指定的主变量数量低于该命令的预期（SQLSTATE 07001 或 07002）。

-203 (`ECPG_TOO_MANY_MATCHES`)

这意味着一个查询已经返回了多个行，但是该语句只准备存储一个结果行（例如，
因为指定的变量不是数组）（SQLSTATE 21000）。

-204 (`ECPG_INT_FORMAT`)

主变量是类型`int`而数据库中的数据是一种不同的类型并且含有一个不能被解释为`int`的值。
该库使用`strtol()`进行这种转换（SQLSTATE 42804）。

-205 (`ECPG_UINT_FORMAT`)

主变量是类型`unsigned int`而数据库中的数据是一种不同的类型并且含有一个不能被解释为`unsigned int`的值。该库使用`strtoul()`进行这种转换（SQLSTATE 42804）。

-206 (`ECPG_FLOAT_FORMAT`)

主变量是类型`float`而数据库中的数据是另一种类型并且含有一个不能被解释为`float`
的值。该库使用`strtod()`进行这种转换（SQLSTATE 42804）。

-211 (`ECPG_CONVERT_BOOL`)

这表示主变量是类型`bool`而数据库中的数据既不是`'t'`也不是`'f'`（SQLSTATE 42804）。

-212 (`ECPG_EMPTY`)

发送给PostgreSQL服务器的语句是空的（通常在一个嵌入式 SQL 程序中不会发生，因此它可
能指向一个内部错误）（SQLSTATE YE002）。

-213 (`ECPG_MISSING_INDICATOR`)

返回了一个空值，但未提供空值指示符变量（SQLSTATE 22002）。

-214 (`ECPG_NO_ARRAY`)

在要求一个数组的地方使用了一个普通变量（SQLSTATE 42804）。

-215 (`ECPG_DATA_NOT_ARRAY`)

在一个要求数组值的地方数据库返回了一个普通变量（SQLSTATE 42804）。

-220 (`ECPG_NO_CONN`)

程序尝试访问一个不存在的连接（SQLSTATE 08003）。

-221 (`ECPG_NOT_CONN`)

程序尝试访问一个存在的连接但是它没有打开（这是一个内部错误）（SQLSTATE YE002）。

-230 (`ECPG_INVALID_STMT`)

尝试使用的语句尚未预备（SQLSTATE 26000）。

-240 (`ECPG_UNKNOWN_DESCRIPTOR`)

未找到指定的描述符。尝试使用的语句尚未预备（SQLSTATE 33000）。

-241 (`ECPG_INVALID_DESCRIPTOR_INDEX`)

指定的描述符超出范围（SQLSTATE 07009）。

-242 (`ECPG_UNKNOWN_DESCRIPTOR_ITEM`)

请求了一个无效的描述符条目（这是一个内部错误）（SQLSTATE YE002）。

-243 (`ECPG_VAR_NOT_NUMERIC`)

执行动态语句时，数据库返回了数值，而主变量不是数值类型（SQLSTATE 07006）。

-244 (`ECPG_VAR_NOT_CHAR`)

执行动态语句时，数据库返回了非数值的值，而主变量是数值类型（SQLSTATE 07006）。

-400 (`ECPG_PGSQL`)

PostgreSQL服务器导致了某个错误。该消息包含来自PostgreSQL服务器的错误消息。

-401 (`ECPG_TRANS`)

PostgreSQL服务器通知我们不能启动、提交或回滚事务（SQLSTATE 08007）。

-402 (`ECPG_CONNECT`)

到数据库的连接尝试没有成功（SQLSTATE 08001）。

100 (`ECPG_NOT_FOUND`)

这是一种无害情况，它表示上一个命令检索或者处理了零行，
或者你到达了游标的末尾（SQLSTATE 02000）。

30.10. 包含文件

要把一个外部文件包含到嵌入式 SQL 程序中，可以使用：

```
EXEC SQL INCLUDE filename;
```

嵌入式 SQL 预处理器会查找名为*filename*.h的文件，对它进行预处理，并把它包含到生成的 C 输出中。这样，被包含文件中的嵌入式 SQL 语句会被正确处理。

注意这并不等同于

```
#include <filename.h>
```

因为这个文件不会经过 SQL 命令预处理。当然，你可以继续使用 C 的 `#include` 指令来包含其他头文件。

> **注意**
>
> 包含文件名是区分大小写的，尽管`EXEC SQL INCLUDE`命令的其余部分遵循 SQL 通常的大小写敏感性规则。

30.11. 处理嵌入式 SQL 程序

现在你已经知道如何编写嵌入式 SQL C 程序了，可能还想知道如何编译它们。编译之前，要先用嵌入式 SQL C 预处理器处理文件，它把你使用的 SQL 语句转换成特殊的函数调用。编译之后，必须与一个包含所需函数的特殊库链接。这些函数从参数中获取信息，用 libpq 接口执行 SQL 命令，并把结果放到指定用于输出的参数中。

该预处理器程序被称作`ecpg`并且被包括在一个正常的PostgreSQL安装中。嵌入式 SQL 程序通常带有扩展名`.pgc`。如果你有一个程序文件`prog1.pgc`，
你可以调用下面的命令对它进行预处理：

```
ecpg prog1.pgc
```

这将创建一个文件`prog1.c`。如果你的输入文件不遵循建议的命名模式，你可以用
`-o`选项显式地指定输出文件。

预处理后的文件可以正常编译，例如：

```
cc -c prog1.c
```

生成的 C 源文件包含来自 PostgreSQL 安装的码头文件，因此如果你把 PostgreSQL 安装在默认不搜索的位置，就必须在编译命令行中加上类似 `-I/usr/local/pgsql/include` 的选项。

要链接嵌入式 SQL 程序，需要链接`libecpg`库，例如：

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

同样，可能需要在该命令行中添加类似`-L/usr/local/pgsql/lib`的选项。

如果你使用make来管理一个大工程的构建过程，把下面的隐式规则包括在你的 makefile 中将会很方便：

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

`ecpg`命令的完整语法可见ecpg。

如果 ecpg 是用 `configure` 的 `--enable-thread-safety` 命令行选项编译的，它就是线程安全的。（编译客户端代码时可能需要使用其他与线程相关的命令行选项。）

30.12. 库函数

libecpg库主要包含用于实现嵌入式 SQL 命令功能的“隐藏”函数，也有一些适合直接调用的函数。注意，直接调用这些函数会使代码失去可移植性。

- 调用ECPGdebug(int *on*, FILE **stream*)时，如果第一个参数非零，就会启用调试日志。日志写入*stream*，包含代入全部输入变量后的所有SQL语句，以及PostgreSQL服务器返回的结果。这对于查找SQL语句中的错误很有用。
- 如果已连接到数据库，ECPGstatus()返回真，否则返回假。

30.13. 内部

本节解释 ECPG 的内部工作原理。这些信息偶尔能帮助用户理解如何使用 ECPG。

ecpg写入输出的前四行是固定的：其中两行是注释，另外两行是与库接口所必需的包含行。随后预处理器会通读整个文件并写出输出，通常它只是把所有内容原样回显到输出中。

当它看到EXEC SQL语句时，就会介入并对其进行改写。命令以EXEC SQL开始，以;结束。其间的所有内容都会被视为一条SQL语句，并进行变量替换解析。

当某个符号以冒号（:）开头时，就会发生变量替换。系统会在此前于EXEC SQL DECLARE声明节中声明的变量里查找同名变量。

库中最重要的函数是ECPGdo，负责执行大多数命令。它接收可变数量的参数，参数数量很容易达到 50 个左右；我们希望这在任何平台上都不会造成问题。

参数是：

一个行号

这是原始行的行号，只用于错误消息。

一个字符串

这是将要发出的SQL命令。输入变量会对它进行修改，也就是那些在编译时尚未知晓、但要填入命令中的变量。字符串中的?表示变量应当放置的位置。

输入变量

每个输入变量都会生成十个参数（见下文）。

ECPGt_EOIT

一个表明后面没有更多输入变量的enum值。

输出变量

每个输出变量都会生成十个参数（见下文）。这些变量将由该函数填充。

ECPGt_EORT

一个表明后面没有更多变量的enum值。

对于每一个作为SQL命令一部分的变量，该函数得到十个参数：

1. 用特殊符号表示的类型。
2. 一个值的指针或者一个指针的指针。
3. 如果变量是一个`char`或者`varchar`，这是它的尺寸。
4. 数组中元素的数量（用于数组获取）。
5. 数组中下一个元素的偏移量（用于数组获取）。
6. 作为一个特殊符号的指示符变量的类型。
7. 一个指示符变量的指针。
8. 0
9. 指示符数组中的元素数量（用于数组取数）。
10. 到指示符数组中下一个元素的偏移量（用于数组取数）。

注意并非所有 SQL 命令都被以这种方式对待。例如，一个打开游标语句：

```
EXEC SQL OPEN cursor;
```

它不会被复制到输出中。相反，游标的`DECLARE`命令会被放在`OPEN`命令的位置上，因为真正打开游标的正是它。

这里有一个完整的例子，它描述了一个文件`foo.pgc`的预处理器输出（对预处理器的每一个特定版本细节可能不同）：

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

会被转换成：

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* declare section start */

#line 1 "foo.pgc"

 int index;
 int result;
/* declare section end */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
```

```
#line 147 "foo.pgc"
```

（这里的缩进是为了可读性而添加的，并非是预处理器做的处理）。

第 31 章 JDBC 接口

JDBC 是 Java 1.1 及更高版本的核心 API。它为兼容 SQL 的数据库提供了一组标准接口。

PostgreSQL 提供一个 4 类（type 4）JDBC 驱动。4 类（type 4）表示该驱动用纯 Java 编写，并使用数据库系统自己的网络协议通信。因此，该驱动是平台无关的；一旦编译完成，就可以在任何系统上使用。

本章并不是一份完整的 JDBC 编程指南，但应该能帮助你入门。更多信息请参阅标准的 JDBC API 文档。另外，也可以看看源码中附带的例子。

31.1. 设置 JDBC 驱动

本节描述在编写或运行使用 JDBC 接口的程序之前需要做的准备工作。

31.1.1. 获取驱动

可以从 PostgreSQL JDBC 网站[1]下载预编译版本的驱动。

你也可以从源码构建驱动，但只有在你修改源代码时才需要这样做。详情请参阅 PostgreSQL 安装说明。安装之后，驱动应该位于 *PREFIX*/share/java/postgresql.jar。构建出的驱动将针对你正在使用的 Java 版本构建。如果你用 1.1 版 JDK 构建，会得到一个支持 JDBC 1 规范的版本；如果用 1.2 或 1.3 版 JDK 构建，会得到一个支持 JDBC 2 规范的版本；最后，如果用 1.4 版 JDK 构建，会得到一个支持 JDBC 3 规范的版本。

31.1.2. 设置类路径

要使用该驱动，需要把 JAR 归档（如果你是从源码构建的，它名为 postgresql.jar，否则它可能名为 pg7.4jdbc1.jar、pg7.4jdbc2.jar 或 pg7.4jdbc3.jar，分别对应 JDBC 1、JDBC 2 和 JDBC 3 版本）包含在类路径中，可以把它放入 CLASSPATH 环境变量，或者在 java 命令行上使用相应的标志。

例如，假设我们有一个应用程序使用 JDBC 驱动访问数据库，并且该应用程序安装在 /usr/local/lib/myapp.jar。PostgreSQL JDBC 驱动安装为 /usr/local/pgsql/share/java/postgresql.jar。要运行该应用程序，我们可以使用：

```
export CLASSPATH=/usr/local/lib/myapp.jar:/usr/local/pgsql/share/java/
postgresql.jar:.
java MyApp
```

在应用程序内载入驱动的内容在第 31.2 节中介绍。

31.1.3. 为 JDBC 准备数据库服务器

由于 Java 只使用 TCP/IP 连接，必须把 PostgreSQL 服务器配置为接受 TCP/IP 连接。这可以通过在 postgresql.conf 文件中设置 tcpip_socket = true，或者在启动 postmaster 时提供 -i 选项标志来完成。

另外，可能还需要配置 pg_hba.conf 文件中的客户端认证设置。详情请参阅第 19 章。JDBC 驱动支持 trust、ident、password、md5 和 crypt 认证方法。

[1] http://jdbc.postgresql.org

31.2. 初始化驱动

本节描述如何在你的程序中载入并初始化 JDBC 驱动。

31.2.1. 导入 JDBC

任何使用 JDBC 的源文件都需要导入 `java.sql` 包，使用：

```
import java.sql.*;
```

> **注意**
>
> 不要导入 `org.postgresql` 包。如果你这样做，你的源代码将无法编译，因为 `javac` 会产生混淆。

31.2.2. 载入驱动

在连接数据库之前，你需要载入驱动。有两种可用的方法，哪一种最好取决于你的代码。

第一种方法中，你的代码使用 `Class.forName()` 方法隐式地载入驱动。对于 PostgreSQL，你可以使用：

```
Class.forName("org.postgresql.Driver");
```

这会载入驱动，并且在载入时，驱动会自动向 JDBC 注册自身。

> **注意**
>
> 如果驱动不可用，`forName()` 方法可能抛出 `ClassNotFoundException`。

这是最常用的方法，但它把你的代码限制为只使用 PostgreSQL。如果你的代码将来可能访问其他数据库系统，并且你不使用任何 PostgreSQL 特有的扩展，那么建议使用第二种方法。

第二种方法在 JVM 启动时使用 `-D` 参数把驱动作为参数传递给它。例如：

```
java -Djdbc.drivers=org.postgresql.Driver example.ImageViewer
```

在这个例子中，JVM 会尝试把载入驱动作为其初始化的一部分。完成后，就会启动 `ImageViewer`。

现在，这种方法是更好的一种，
因为它允许你的代码在不重新编译的情况下与其他数据库包一起使用。唯一还需要改变的是连接 URL，这是接下来要介绍的内容。

最后一点：当你的代码随后尝试打开 `Connection` 时，如果抛出了 No driver available `SQLException`，那么这可能是驱动不在类路径中，或者参数中的值不正确造成的。

31.2.3. 连接到数据库

在 JDBC 中，数据库由一个 URL（统一资源定位符）表示。对于 PostgreSQL，它采用下列形式之一：

- jdbc:postgresql:*database*
- jdbc:postgresql://*host*/*database*
- jdbc:postgresql://*host*:*port*/*database*

这些参数的含义如下：

host

服务器的主机名。默认为 localhost。要指定一个 IPv6 地址，你必须用方括号把 *host* 参数括起来，例如：

```
jdbc:postgresql://[::1]:5740/accounting
```

port

服务器监听的端口号。默认为 PostgreSQL 的标准端口号（5432）。

database

数据库名。

要建立连接，你需要从 JDBC 获得一个 Connection 实例。为此，你可以使用 DriverManager.getConnection() 方法：

```
Connection db = DriverManager.getConnection(url, username, password);
```

31.2.4. 关闭连接

要关闭数据库连接，只需对 Connection 调用 close() 方法：

```
db.close();
```

31.3. 发出查询并处理结果

每当你想向数据库发出 SQL 语句时，你都需要一个 Statement 或 PreparedStatement 实例。一旦你有了一个 Statement 或 PreparedStatement，就可以用它发出查询。这将返回一个 ResultSet 实例，其中包含全部结果（如何改变这一行为参见第 31.3.1 节）。例 31.1演示了这个过程。

例 31.1. 在 JDBC 中处理一个简单查询

这个例子将使用 Statement 发出一个简单的查询，并打印出每一行的第一列。

```
Statement st = db.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM mytable WHERE columnfoo
 = 500");
while (rs.next()) {
    System.out.print("Column 1 returned ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

这个例子发出与之前相同的查询，但使用 PreparedStatement 和查询中的一个绑定值。

```
int foovalue = 500;
PreparedStatement st = db.prepareStatement("SELECT * FROM mytable
 WHERE columnfoo = ?");
st.setInt(1, foovalue);
ResultSet rs = st.executeQuery();
while (rs.next()) {
    System.out.print("Column 1 returned ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

31.3.1. 基于游标获取结果

默认情况下，驱动会一次性收集该查询的所有结果。这对大型数据集来说可能不方便，因此 JDBC 驱动提供了一种把 ResultSet 建立在数据库游标上、并且每次只取少量行的手段。

少量行会被缓存在连接的客户端一侧，当缓存用尽时，通过重新定位游标取回下一批行。

例 31.2. 设置提取数量来打开和关闭游标。

把代码改成游标模式非常简单，只需把 Statement 的提取数量（fetch size）设置为适当的大小即可。把提取数量设回 0 将导致缓存所有行（默认行为）。

```
Statement st = db.createStatement();
// Turn use of the cursor on.
st.setFetchSize(50);
ResultSet rs = st.executeQuery("SELECT * FROM mytable");
while (rs.next()) {
   System.out.print("a row was returned.");
}
rs.close();
// Turn the cursor off.
st.setFetchSize(0);
ResultSet rs = st.executeQuery("SELECT * FROM mytable");
while (rs.next()) {
   System.out.print("many rows were returned.");
}
rs.close();
// Close the statement.
st.close();
```

31.3.2. 使用 Statement 或 PreparedStatement 接口

使用 Statement 或 PreparedStatement 接口时必须考虑以下几点：

- 你可以按需多次使用同一个 Statement 实例。你可以在打开连接后立刻创建一个，并在该连接的整个生命周期内使用它。但你必须记住，每个 Statement 或 PreparedStatement 在同一时刻只能存在一个 ResultSet。
- 如果你在处理一个 ResultSet 的同时还需要执行另一个查询，只需创建并使用另一个 Statement。

- 如果你使用线程，并且有多个线程在使用数据库，你必须为每个线程使用一个单独的 `Statement`。如果你正在考虑使用线程，请参阅第 31.9 节，其中涵盖了一些要点。
- 用完 `Statement` 或 `PreparedStatement` 之后，你应该关闭它。

31.3.3. 使用 ResultSet 接口

使用 `ResultSet` 接口时必须考虑以下几点：

- 在读取任何值之前，你必须调用 `next()`。如果还有结果它就返回 true，但更重要的是，它会让该行准备好被处理。
- 按照 JDBC 规范，你应该只访问一个字段一次。遵守这条规则是最安全的，尽管目前 PostgreSQL 驱动允许你按需多次访问一个字段。
- 一旦用完一个 `ResultSet`，你必须通过调用 `close()` 来关闭它。
- 一旦你用创建某个 `ResultSet` 的 `Statement` 发起另一个查询，当前打开的 `ResultSet` 实例会被自动关闭。

31.4. 执行更新

要更改数据（执行 `INSERT`、`UPDATE` 或 `DELETE`），你使用 `executeUpdate()` 方法。这个方法类似于用于发出 `SELECT` 语句的 `executeQuery()` 方法，但它不返回 `ResultSet`；而是返回被该 `INSERT`、`UPDATE` 或 `DELETE` 语句影响的行数。例 31.3演示了这种用法。

例 31.3. 在 JDBC 中删除行

这个例子将发出一条简单的 `DELETE` 语句并打印出被删除的行数。

```
int foovalue = 500;
PreparedStatement st = db.prepareStatement("DELETE FROM mytable WHERE
 columnfoo = ?");
st.setInt(1, foovalue);
int rowsDeleted = st.executeUpdate();
System.out.println(rowsDeleted + " rows deleted");
st.close();
```

31.5. 调用存储函数

PostgreSQL 的 JDBC 驱动完全支持调用 PostgreSQL 存储函数。

例 31.4. 调用一个内置存储函数

这个例子展示如何调用一个 PostgreSQL 内置函数 `upper`，它只是把提供的字符串参数转换为大写。

```
// Turn transactions off.
con.setAutoCommit(false);
// Procedure call.
CallableStatement upperProc = con.prepareCall("{ ? = call upper( ? ) }
");
```

```
upperProc.registerOutParameter(1, Types.VARCHAR);
upperProc.setString(2, "lowercase to uppercase");
upperProc.execute();
String upperCased = upperProc.getString(1);
upperProc.close();
```

31.5.1. 使用 CallableStatement 接口

适用于 Statement 和 PreparedStatement 的所有注意事项同样适用于 CallableStatement，但此外你还必须考虑一个额外的限制：

- 只能在一个事务内部调用存储函数。

31.5.2. 从存储函数获取 ResultSet

PostgreSQL 的存储函数可以通过 refcursor 值返回结果。一个 refcursor。

作为对 JDBC 的扩展，PostgreSQL JDBC 驱动可以把 refcursor 值作为 ResultSet 值返回。

例 31.5. 从函数获取 refcursor 值

在调用一个返回 refcursor 的函数时，你必须把 getObject 的返回类型强制转换为 ResultSet

```
// Turn transactions off.
con.setAutoCommit(false);
// Procedure call.
CallableStatement proc = con.prepareCall("{ ? = call doquery ( ? ) }")
;
proc.registerOutParameter(1, Types.Other);
proc.setInt(2, -1);
proc.execute();
ResultSet results = (ResultSet) proc.getObject(1);
while (results.next()) {
  // do something with the results...
}
results.close();
proc.close();
```

也可以把 refcursor 返回值本身当作一个独立的类型对待。JDBC 驱动为此提供了 org.postgresql.PGRefCursorResultSet 类。

例 31.6. 把 refcursor 当作一个独立的类型对待

```
con.setAutoCommit(false);
CallableStatement proc = con.prepareCall("{ ? = call doquery ( ? ) }")
;
proc.registerOutParameter(1, Types.Other);
proc.setInt(2, 0);
org.postgresql.PGRefCursorResultSet refcurs
    = (PGRefCursorResultSet) con.getObject(1);
String cursorName = refcurs.getRefCursor();
proc.close();
```

31.6. 创建和修改数据库对象

要创建、修改或删除一个数据库对象（例如表或视图），你使用 `execute()` 方法。这个方法类似于 `executeQuery()` 方法，但它不返回结果。例 31.7演示了这种用法。

例 31.7. 在 JDBC 中删除一个表

这个例子将删除一个表。

```
Statement st = db.createStatement();
st.execute("DROP TABLE mytable");
st.close();
```

31.7. 存储二进制数据

PostgreSQL 提供两种不同的存储二进制数据的方式。二进制数据可以使用 `bytea` 数据类型存储在一个表中，或者使用大对象特性——它把二进制数据以一种特殊格式存储在一个单独的表中，并通过在你的表中存储一个 `oid` 类型的值来引用该表。

为了确定哪种方法合适，你需要理解每种方法的限制。`bytea` 数据类型并不适合存储非常大量的二进制数据。虽然 `bytea` 类型的列最多可以容纳 1 GB 的二进制数据，但处理这样大的值需要大量的内存。用大对象方法存储二进制数据更适合存储非常大的值，但它也有自己的限制。具体来说，删除一个包含大对象引用的行并不会删除该大对象。删除大对象是一个需要单独执行的操作。大对象还有一些安全性问题，因为任何连接到数据库的人都可以查看和/或修改任何大对象，即使他们没有查看/更新包含该大对象引用的行的权限。

7.2 版是第一个支持 `bytea` 数据类型的 JDBC 驱动版本。与之前的版本相比，7.2 中引入的这一功能带来了一个行为上的变化。从 7.2 开始，`getBytes()`、`setBytes()`、`getBinaryStream()` 和 `setBinaryStream()` 方法操作 `bytea` 数据类型。而在 7.1 及更早的版本中，这些方法操作与大对象相关联的 `oid` 数据类型。通过把 `Connection` 对象上的 `compatible` 属性设置为 `7.1`，可以把驱动恢复为旧的 7.1 行为。

要使用 `bytea` 数据类型，你只需使用 `getBytes()`、`setBytes()`、`getBinaryStream()` 或 `setBinaryStream()` 方法。

要使用大对象功能，你可以使用 PostgreSQL JDBC 驱动提供的 `LargeObject` 类，或者使用 `getBLOB()` 和 `setBLOB()` 方法。

重要

你必须在一个 SQL 事务块内访问大对象。你可以通过调用 `setAutoCommit(false)` 来启动一个事务块。

注意

在 JDBC 驱动的一个未来版本中，`getBLOB()` 和 `setBLOB()` 方法可能不再与大对象交互，而是改为处理 `bytea` 数据类型。因此，如果你打算使用大对象，建议使用 `LargeObject` API。

例 31.8包含一些关于如何用 PostgreSQL JDBC 驱动处理二进制数据的例子。

例 31.8. 在 JDBC 中处理二进制数据

例如，假设你有一个表存储图像的文件名，并且你还想把图像本身存储在一个 `bytea` 列中：

```
CREATE TABLE images (imgname text, img bytea);
```

要插入一幅图像，你可以使用：

```
File file = new File("myimage.gif");
FileInputStream fis = new FileInputStream(file);
PreparedStatement ps = conn.prepareStatement("INSERT INTO images
 VALUES (?, ?)");
ps.setString(1, file.getName());
ps.setBinaryStream(2, fis, file.length());
ps.executeUpdate();
ps.close();
fis.close();
```

这里，`setBinaryStream()` 从一个流向 `bytea` 类型的列传输指定数量的字节。如果图像的内容已经在一个 `byte[]` 中，这也可以用 `setBytes()` 方法来完成。

检索一幅图像甚至更容易。（我们在这里使用 `PreparedStatement`，但同样可以使用 `Statement` 类。）

```
PreparedStatement ps = con.prepareStatement("SELECT img FROM images
 WHERE imgname = ?");
ps.setString(1, "myimage.gif");
ResultSet rs = ps.executeQuery();
if (rs != null) {
    while (rs.next()) {
        byte[] imgBytes = rs.getBytes(1);
        // use the data in some way here
    }
    rs.close();
}
ps.close();
```

这里二进制数据是作为 `byte[]` 检索的。你也可以改用一个 `InputStream` 对象。

又或者，你要存储一个非常大的文件，并想使用 `LargeObject` API 来存储该文件：

```
CREATE TABLE imageslo (imgname text, imgoid oid);
```

要插入一幅图像，你可以使用：

```
// All LargeObject API calls must be within a transaction block
conn.setAutoCommit(false);

// Get the Large Object Manager to perform operations with
LargeObjectManager lobj = ((org.postgresql.PGConnection)conn).getLarge
ObjectAPI();

// Create a new large object
int oid = lobj.create(LargeObjectManager.READ | LargeObjectManager.
WRITE);
```

```
// Open the large object for writing
LargeObject obj = lobj.open(oid, LargeObjectManager.WRITE);

// Now open the file
File file = new File("myimage.gif");
FileInputStream fis = new FileInputStream(file);

// Copy the data from the file to the large object
byte buf[] = new byte[2048];
int s, tl = 0;
while ((s = fis.read(buf, 0, 2048)) > 0) {
    obj.write(buf, 0, s);
    tl += s;
}

// Close the large object
obj.close();

// Now insert the row into imageslo
PreparedStatement ps = conn.prepareStatement("INSERT INTO imageslo
 VALUES (?, ?)");
ps.setString(1, file.getName());
ps.setInt(2, oid);
ps.executeUpdate();
ps.close();
fis.close();
```

从大对象中检索该图像:

```
// All LargeObject API calls must be within a transaction block
conn.setAutoCommit(false);

// Get the Large Object Manager to perform operations with
LargeObjectManager lobj = ((org.postgresql.PGConnection)conn).getLarge
ObjectAPI();

PreparedStatement ps = con.prepareStatement("SELECT imgoid FROM
 imageslo WHERE imgname = ?");
ps.setString(1, "myimage.gif");
ResultSet rs = ps.executeQuery();
if (rs != null) {
    while (rs.next()) {
        // Open the large object for reading
        int oid = rs.getInt(1);
        LargeObject obj = lobj.open(oid, LargeObjectManager.READ);

        // Read the data
        byte buf[] = new byte[obj.size()];
        obj.read(buf, 0, obj.size());
        // Do something with the data read here

        // Close the object
        obj.close();
```

```
    }
    rs.close();
}
ps.close();
```

31.8. PostgreSQL 对 JDBC API 的扩展

PostgreSQL 是一个可扩展的数据库系统。你可以向服务器添加自己的函数，然后从查询中调用它们，甚至添加你自己的数据类型。由于这些是 PostgreSQL 独有的功能，我们用一组扩展 API 从 Java 中支持它们。标准驱动核心中的一些功能实际上就是用这些扩展来实现大对象等特性的。

31.8.1. 访问扩展

要访问其中一些扩展，你需要使用 `org.postgresql.PGConnection` 类中的一些额外方法。在这种情况下，你需要对 `Driver.getConnection()` 的返回值进行强制转换。例如：

```
Connection db = Driver.getConnection(url, username, password);
// ...
// later on
Fastpath fp = ((org.postgresql.PGConnection)db).getFastpathAPI();
```

31.8.1.1. `org.postgresql.PGConnection` 类

```
public class PGConnection
```

这些是用于访问 PostgreSQL 扩展的额外方法。

31.8.1.1.1. 方法

- `public Fastpath getFastpathAPI() throws SQLException`

  这将返回当前连接的 fast-path API。它主要供大对象 API 使用。

  使用它的最佳方式如下：

  ```
  import org.postgresql.fastpath.*;
  ...
  Fastpath fp = ((org.postgresql.PGConnection)myconn).getFastpathAPI()
  ;
  ```

  其中 `myconn` 是一个到 PostgreSQL 的已打开的 `Connection`。

  返回：. 一个允许访问 PostgreSQL 服务器上函数的 `Fastpath` 对象。

  抛出：. 首次初始化时由 `Fastpath` 抛出的 `SQLException`

- `public LargeObjectManager getLargeObjectAPI() throws SQLException`

  这将返回当前连接的大对象 API。

  使用它的最佳方式如下：

  ```
  import org.postgresql.largeobject.*;
  ...
  ```

```
LargeObjectManager lo = ((org.postgresql.PGConnection)myconn).get
LargeObjectAPI();
```

其中 `myconn` 是一个到 PostgreSQL 的已打开的 `Connection`。

返回：. 一个实现了该 API 的 `LargeObject` 对象

抛出：. 首次初始化时由 `LargeObject` 抛出的 `SQLException`

- `public void addDataType(String type, String name)`

  这允许客户端代码为 PostgreSQL 较独特的数据类型之一添加一个处理器。通常，驱动不认识的数据类型会被 `ResultSet.getObject()` 作为 `PGobject` 实例返回。这个方法允许你编写一个扩展 `PGobject` 的类，并告诉驱动要使用的类型名和类名。它的缺点是，每次建立连接时你都必须调用这个方法。

  使用它的最佳方式如下：

```
 ...
((org.postgresql.PGConnection)myconn).addDataType("mytype","my.
class.name");
 ...
```

  其中 `myconn` 是一个到 PostgreSQL 的已打开的 `Connection`。处理类必须扩展 `org.postgresql.util.PGobject`。

31.8.1.2. org.postgresql.Fastpath 类

```
public class Fastpath extends Object

java.lang.Object
   |
   +----org.postgresql.fastpath.Fastpath
```

`Fastpath` 是一个存在于 libpq C 接口中的 API，它允许一台客户端机器在数据库服务器上执行一个函数。大多数客户端代码不需要使用这个方法，但之所以提供它，是因为大对象 API 用到它。

要使用它，你需要用下面这一行导入 `org.postgresql.fastpath` 包：

```
import org.postgresql.fastpath.*;
```

然后，在你的代码中，你需要获取一个 `FastPath` 对象：

```
Fastpath fp = ((org.postgresql.PGConnection)conn).getFastpathAPI();
```

这将返回一个与你可用它来发出命令的数据库连接相关联的实例。必须把 `Connection` 强制转换为 `org.postgresql.PGConnection`，因为 `getFastpathAPI()` 是一个扩展方法，不属于 JDBC。一旦你有了一个 `Fastpath` 实例，就可以使用 `fastpath()` 方法来执行一个服务器函数。

参见：. `FastpathFastpathArg`、`LargeObject`

31.8.1.2.1. 方法

- `public Object fastpath(int fnid,`
  `boolean resulttype,`
  `FastpathArg args[]) throws SQLException`

向 PostgreSQL 服务器发送一次函数调用。

参数：. *fnid* - 函数 id *resulttype* - 如果结果是一个整数则为 true，对于其他结果为 false *args* - 要传递给 fast-path 调用的 `FastpathArguments`

返回：. 无数据时为 null，整数结果时为 Integer，否则为 byte[]

- `public Object fastpath(String name, boolean resulttype, FastpathArg args[]) throws SQLException`

  按名称向 PostgreSQL 服务器发送一次函数调用。

  > **注意**
  >
  > 过程名到函数 id 的映射必须已存在，通常是先前调用过 `addfunction()`。这是首选的调用方法，因为函数 id 在服务器的不同版本之间可以/可能会改变。关于它如何工作的例子，请参阅 org.postgresql.LargeObject

  参数：. *name* - 函数名 *resulttype* - 如果结果是一个整数则为 true，对于其他结果为 false *args* - 要传递给 fast-path 调用的 `FastpathArguments`

  返回：. 无数据时为 null，整数结果时为 Integer，否则为 byte[]

  参见：. `LargeObject`

- `public int getInteger(String name, FastpathArg args[]) throws SQLException`

  这个便捷方法假定返回值是一个 Integer

  参数：. *name* - 函数名 *args* - 函数参数

  返回：. 整数结果

  抛出：. 发生数据库访问错误或没有结果时的 `SQLException`

- `public byte[] getData(String name, FastpathArg args[]) throws SQLException`

  这个便捷方法假定返回值是二进制数据。

  参数：. *name* - 函数名 *args* - 函数参数

  返回：. 包含结果的 byte[] 数组

  抛出：. 发生数据库访问错误或没有结果时的 `SQLException`

- `public void addFunction(String name, int fnid)`

  这会向我们的查找表添加一个函数。用户代码应当使用基于查询的 `addFunctions` 方法，而不是硬编码 OID。一个函数的 OID 不保证保持不变，即使在同一版本的不同服务器上也是如此。

- `public void addFunctions(ResultSet rs) throws SQLException`

  这需要一个包含两列的 `ResultSet`。第 1 列包含函数名，第 2 列包含 OID。它会读取整个 `ResultSet`，把这些值装入函数表。

> **重要**
>
> 记得在调用它之后对 `ResultSet` 调用 `close()`！

> **关于函数名查找的实现说明**
>
> PostgreSQL 把函数 id 及其对应的名称存储在 `pg_proc` 表中。为了在本地加快速度，不是在需要时从该表中逐个查询函数，而是使用一个 `Hashtable`。而且，只有所需的函数才会被输入到这个表中，以使连接时间尽可能快。
>
> `org.postgresql.LargeObject` 类在启动时执行一次查询，并把返回的 `ResultSet` 传递给这里的 `addFunctions()` 方法。完成之后，大对象 API 就可以按名称引用这些函数。
>
> 不要以为手动把它们转换成 OID 就能工作。好吧，目前它们可以，但在开发过程中它们可能改变（V7.0 曾有过一些关于此的讨论），因此这样实现是为了避免将来出现不必要的麻烦。

  参见：. `LargeObjectManager`

- `public int getID(String name) throws SQLException`

  这将返回与名称相关联的函数 id。如果没有为这个名称调用过 `addFunction()` 或 `addFunctions()`，那么就会抛出 `SQLException`。

31.8.1.3. `org.postgresql.fastpath.FastpathArg` 类

```
public class FastpathArg extends Object
```

```
java.lang.Object
   |
   +----org.postgresql.fastpath.FastpathArg
```

每次 fast-path 调用都需要一个参数数组，其数量和类型取决于被调用的函数。这个类实现了提供这一能力所需的方法。

关于如何使用它的例子，请参阅 `org.postgresql.LargeObject` 包。

参见：. `Fastpath`、`LargeObjectManager`、`LargeObject`

31.8.1.3.1. 构造器

- `public FastpathArg(int value)`

  构造一个由一个整数值组成的参数

  参数：. value - 要设置的 int 值

- `public FastpathArg(byte bytes[])`

  构造一个由一个字节数组组成的参数

  参数：. bytes - 要存储的数组

- `public FastpathArg(byte buf[],`
  `int off,`
  `int len)`

  构造一个由字节数组的一部分组成的参数

  参数：.

  buf

  源数组

  off

  数组内的偏移量

  len

  要包含的数据长度

- `public FastpathArg(String s)`

  构造一个由一个 String 组成的参数。

31.8.2. 几何数据类型

PostgreSQL 有一组可以把几何特性存储到表中的数据类型。包括点、线和多边形。我们用 org.postgresql.geometric 包在 Java 中支持这些类型。它包含一些扩展 org.postgresql.util.PGobject 类的类。关于如何实现你自己的数据类型处理器的详情，请参阅那个类。

```
Class org.postgresql.geometric.PGbox

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGbox

   public class PGbox extends PGobject implements Serializable,
Cloneable
```

它表示 `PostgreSQL` 中的 `box` 数据类型。

变量

```
 public PGpoint point[]
```

这是该矩形的两个角点。

构造器

```
public PGbox(double x1,
             double y1,
             double x2,
             double y2)
```

参数:
x1 - 第一个 x 坐标
y1 - 第一个 y 坐标
x2 - 第二个 x 坐标
y2 - 第二个 y 坐标

```
public PGbox(PGpoint p1,
             PGpoint p2)
```

参数:
p1 - 第一个点
p2 - 第二个点

```
public PGbox(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的矩形定义

抛出: SQLException
如果定义无效

```
public PGbox()
```

必需的构造器

方法

```
public void setValue(String value) throws SQLException
```

此方法设置该对象的值。子类应当覆盖它，但仍应调用它。

参数:
value - 对象值的字符串表示

抛出: SQLException
如果值对该类型无效则抛出

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

参数:
obj - 要比较的对象

返回值:
如果两个矩形相同则为 true

Overrides:

```
                equals in class PGobject

 public Object clone()

          必须覆盖此方法才能克隆该对象

        Overrides:
                clone in class PGobject

 public String getValue()

        返回值:
                按 PostgreSQL 所期望语法表示的 PGbox

        Overrides:
                getValue in class PGobject

Class org.postgresql.geometric.PGcircle

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGcircle

   public class PGcircle extends PGobject implements Serializable,
Cloneable

   它表示 PostgreSQL 的 circle 数据类型，由一个点和一个半径组成

变量

 public PGpoint center

          这是圆心点

 double radius

          这是半径

构造器

 public PGcircle(double x,
                 double y,
                 double r)

        参数:
               x - 圆心的坐标
                y - 圆心的坐标
                r - 圆的半径

 public PGcircle(PGpoint c,
                 double r)
```

参数:
c - 描述圆心的 PGpoint
r - 圆的半径

public PGcircle(String s) throws SQLException

参数:
s - PostgreSQL 语法的圆定义。

抛出: SQLException
转换失败时

public PGcircle()

此构造器由驱动使用。

方法

public void setValue(String s) throws SQLException

参数:
s - PostgreSQL 语法的圆定义。

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

public boolean equals(Object obj)

参数:
obj - 要比较的对象

返回值:
如果两个圆相同则为 true

Overrides:
equals in class PGobject

public Object clone()

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

public String getValue()

返回值:
按 PostgreSQL 所期望语法表示的 PGcircle

Overrides:
getValue in class PGobject

```
Class org.postgresql.geometric.PGline

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGline

   public class PGline extends PGobject implements Serializable,
Cloneable
```

它实现由两点构成的 line。服务器目前尚未实现 line 类型，但这个类保证了在实现之后我们就能直接使用它。

变量

```
 public PGpoint point[]
```

这就是这两个点。

构造器

```
 public PGline(double x1,
               double y1,
               double x2,
               double y2)
```

参数:
x1 - 第一个点的坐标
y1 - 第一个点的坐标
x2 - 第二个点的坐标
y2 - 第二个点的坐标

```
 public PGline(PGpoint p1,
               PGpoint p2)
```

参数:
p1 - 第一个点
p2 - 第二个点

```
 public PGline(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的直线定义。

抛出: SQLException
转换失败时

```
 public PGline()
```

驱动所需

方法

```
public void setValue(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的线段定义

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

参数:
obj - 要比较的对象

返回值:
如果两条直线相同则为 true

Overrides:
equals in class PGobject

```
public Object clone()
```

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

```
public String getValue()
```

返回值:
按 PostgreSQL 所期望语法表示的 PGline

Overrides:
getValue in class PGobject

Class org.postgresql.geometric.PGlseg

```
java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGlseg
```

public class PGlseg extends PGobject implements Serializable, Cloneable

它实现由两点构成的 lseg (线段)

变量

```
public PGpoint point[]
```

这就是这两个点。

构造器

```
public PGlseg(double x1,
              double y1,
              double x2,
              double y2)
```

参数:

x1 - 第一个点的坐标
y1 - 第一个点的坐标
x2 - 第二个点的坐标
y2 - 第二个点的坐标

```
public PGlseg(PGpoint p1,
              PGpoint p2)
```

参数:
p1 - 第一个点
p2 - 第二个点

```
public PGlseg(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的线段定义。

抛出: SQLException
转换失败时

```
public PGlseg()
```

驱动所需

方法

```
public void setValue(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的线段定义

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

参数:
obj - 要比较的对象

返回值:
如果两条线段相同则为 true

Overrides:
equals in class PGobject

public Object clone()

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

public String getValue()

返回值:
按 PostgreSQL 所期望语法表示的 PGlseg

Overrides:
getValue in class PGobject

Class org.postgresql.geometric.PGpath

```
java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGpath
```

public class PGpath extends PGobject implements Serializable, Cloneable

它实现 path (一条多段线，可以是闭合的)

变量

public boolean open

如果路径是开放的则为 true，闭合则为 false

public PGpoint points[]

定义该路径的各个点

构造器

public PGpath(PGpoint points[],
boolean open)

参数:
points - 定义该路径的 PGpoint 数组
open - 如果路径是开放的则为 true，闭合则为 false

public PGpath()

驱动所需

```
public PGpath(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的路径定义。

抛出: SQLException
转换失败时

方法

```
public void setValue(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的路径定义

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

参数:
obj - 要比较的对象

返回值:
如果两条路径相同则为 true

Overrides:
equals in class PGobject

```
public Object clone()
```

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

```
public String getValue()
```

返回按 PostgreSQL 所期望语法表示的该路径

Overrides:
getValue in class PGobject

```
public boolean isOpen()
```

如果路径是开放的则返回 true

```
public boolean isClosed()
```

如果路径是闭合的则返回 true

```
public void closePath()
```

把路径标记为闭合

```
public void openPath()
```

把路径标记为开放

```
Class org.postgresql.geometric.PGpoint

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGpoint

   public class PGpoint extends PGobject implements Serializable,
Cloneable
```

它实现 java.awt.Point 的一个版本，只是用 double 表示坐标。

它映射到 PostgreSQL 的 point 数据类型。

变量

```
public double x
```

该点的 X 坐标

```
public double y
```

该点的 Y 坐标

构造器

```
public PGpoint(double x,
               double y)
```

参数:
x - 坐标
y - 坐标

```
public PGpoint(String value) throws SQLException
```

当某点嵌入其他几何类型的定义中时，主要由那些类型调用此构造器。

参数:
value - PostgreSQL 语法的该点定义

```
public PGpoint()
```

驱动所需

方法

```
public void setValue(String s) throws SQLException
```

参数:
s - PostgreSQL 语法的该点定义

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

```
public boolean equals(Object obj)
```

参数:
obj - 要比较的对象

返回值:
如果两个点相同则为 true

Overrides:
equals in class PGobject

```
public Object clone()
```

必须覆盖此方法才能克隆该对象

Overrides:
clone in class PGobject

```
public String getValue()
```

返回值:
按 PostgreSQL 所期望语法表示的 PGpoint

Overrides:
getValue in class PGobject

```
public void translate(int x,
                      int y)
```

按给定的量平移该点。

参数:
x - 在 x 轴上要加的整数增量
y - 在 y 轴上要加的整数增量

```
public void translate(double x,
                      double y)
```

按给定的量平移该点。

参数:
x - 在 x 轴上要加的 double 增量
y - 在 y 轴上要加的 double 增量

```
public void move(int x,
                 int y)
```

把该点移动到给定的坐标。

参数:
x - 整数坐标
y - 整数坐标

```
public void move(double x,
                 double y)
```

把该点移动到给定的坐标。

参数:
x - double 坐标
y - double 坐标

```
public void setLocation(int x,
                        int y)
```

把该点移动到给定的坐标。相关描述参见 java.awt.Point

参数:
x - 整数坐标
y - 整数坐标

参见:
Point

```
public void setLocation(Point p)
```

把该点移动到给定的 java.awt.Point。相关描述参见 java.awt.Point

参数:
p - 要移动到的点

参见:
Point

Class org.postgresql.geometric.PGpolygon

```
java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGpolygon
```

```
public class PGpolygon extends PGobject implements Serializable,
```

Cloneable

它实现 PostgreSQL 中的 polygon 数据类型。

变量

public PGpoint points[]

定义该多边形的各个点

构造器

public PGpolygon(PGpoint points[])

用一个 PGpoint 数组创建多边形

参数:
points - 定义该多边形的各个点

public PGpolygon(String s) throws SQLException

参数:
s - PostgreSQL 语法的多边形定义。

抛出: SQLException
转换失败时

public PGpolygon()

驱动所需

方法

public void setValue(String s) throws SQLException

参数:
s - PostgreSQL 语法的多边形定义

抛出: SQLException
转换失败时

Overrides:
setValue in class PGobject

public boolean equals(Object obj)

参数:
obj - 要比较的对象

返回值:
如果两个多边形相同则为 true

Overrides:
equals in class PGobject

```
public Object clone()

          必须覆盖此方法才能克隆该对象

       Overrides:
               clone in class PGobject

public String getValue()

       返回值:
               按 PostgreSQL 所期望语法表示的 PGpolygon

       Overrides:
               getValue in class PGobject
```

31.8.3. 大对象

大对象在标准的 JDBC 规范中受支持。但是，那个接口是有限的，而 PostgreSQL 提供的 API 允许对对象内容进行随机访问，就像它是一个本地文件一样。

org.postgresql.largeobject 包向 Java 提供了 libpq C 接口的大对象 API。它由两个类组成：`LargeObjectManager` 处理大对象的创建、打开和删除，而 `LargeObject` 处理单个对象。

31.8.3.1. `org.postgresql.largeobject.LargeObject` 类

```
public class LargeObject extends Object

java.lang.Object
   |
   +----org.postgresql.largeobject.LargeObject
```

这个类实现了到 PostgreSQL 的大对象接口。

它提供了运行该接口所需的基本方法，外加为这个对象提供 `InputStream` 和 `OutputStream` 类的一对方法。

通常，客户端代码会使用 `BLOB` 中的方法来访问大对象。

然而，有时需要对大对象的更低层访问，这是 JDBC 规范不支持的。

关于如何获得一个大对象的访问权，或者如何创建一个，请参阅 org.postgresql.largeobject.LargeObjectManager。

参见：. `LargeObjectManager`

31.8.3.1.1. 变量

public static final int SEEK_SET

表示从文件开头定位

public static final int SEEK_CUR

表示从当前位置定位

public static final int SEEK_END

表示从文件末尾定位

31.8.3.1.2. 方法

- `public int getOID()`

  返回这个 `LargeObject` 的 OID

- `public void close() throws SQLException`

  这个方法关闭该对象。在调用它之后，你不应再调用这个对象中的方法。

- `public byte[] read(int len) throws SQLException`

  从该对象读取一些数据，并以 byte[] 数组返回

- ```
  public int read(byte buf[],
                  int off,
                  int len) throws SQLException
  ```

  从该对象读取一些数据到现有的数组中

  参数：.

  buf

  目标数组

  off

  数组内的偏移量

  len

  要读取的字节数

- `public void write(byte buf[]) throws SQLException`

  向该对象写入一个数组

- ```
  public void write(byte buf[],
                    int off,
                    int len) throws SQLException
  ```

  从一个数组向该对象写入一些数据

  参数：.

  buf

  目标数组

  off

  数组内的偏移量

len

要写入的字节数

31.8.3.2. org.postgresql.largeobject.LargeObjectManager 类

```
public class LargeObjectManager extends Object

java.lang.Object
   |
   +----org.postgresql.largeobject.LargeObjectManager
```

这个类实现了到 PostgreSQL 的大对象接口。它提供的方法允许客户端代码从数据库中创建、打开和删除大对象。打开一个对象时，会返回一个 `org.postgresql.largeobject.LargeObject` 的实例，然后它的方法就允许访问该对象。

这个类只能由 org.postgresql.PGConnection 创建。要获得这个类的访问权，使用下面这段代码：

```
import org.postgresql.largeobject.*;
Connection  conn;
LargeObjectManager lobj;
// ... code that opens a connection ...
lobj = ((org.postgresql.PGConnection)myconn).getLargeObjectAPI();
```

通常，客户端代码会使用 `BLOB` 方法来访问大对象。但是，有时需要对大对象的更低层访问，这是 JDBC 规范不支持的。

关于如何操纵大对象的内容，请参阅 org.postgresql.largeobject.LargeObject。

31.8.3.2.1. 变量

`public static final int WRITE`

这个模式表示我们想写入一个对象。

`public static final int READ`

这个模式表示我们想读取一个对象。

`public static final int READWRITE`

这个模式是默认值。它表示我们想对一个大对象进行读写访问。

31.8.3.2.2. 方法

- `public LargeObject open(int oid) throws SQLException`

  这将根据 OID 打开一个现有的大对象。这个方法假定需要 `READ` 和 `WRITE` 访问（默认值）。

- `public LargeObject open(int oid, int mode) throws SQLException`

  这将根据 OID 打开一个现有的大对象，并允许设置访问模式。

- `public int create() throws SQLException`

  这将创建一个大对象并返回其 OID。新对象的属性默认为 `READWRITE`。

- `public int create(int mode) throws SQLException`

  这将创建一个大对象，返回其 OID，并设置访问模式。

- `public void delete(int oid) throws SQLException`

  这将删除一个大对象。

- `public void unlink(int oid) throws SQLException`

  这将删除一个大对象。它与 delete 方法完全相同，提供它是因为 C API 使用“unlink”。

31.9. 在多线程或 Servlet 环境中使用驱动

许多 JDBC 驱动的一个问题是，同一时刻只能有一个线程使用一个 `Connection` —— 否则一个线程可能在另一个线程接收结果时发送查询，这可能造成严重的混乱。

PostgreSQL JDBC 驱动是线程安全的。因此，如果你的应用程序使用多个线程，你不必担心用复杂的算法来确保同一时刻只有一个线程使用数据库。

如果一个线程在另一个线程正在使用连接时尝试使用它，它会等到另一个线程完成其当前操作。如果该操作是一条常规的 SQL 语句，那么该操作由发送语句和检索任何 `ResultSet`（完整地）组成。如果它是一次 fast-path 调用（例如从一个大对象读取一个块），那么它由发送和检索相应的数据组成。

这对应用程序和 applet 来说没有问题，但对 servlet 可能造成性能问题。如果你有多个线程执行查询，那么除一个之外每个线程都会暂停。为了解决这个问题，建议你创建一个连接池。
每当一个线程需要使用数据库时，它向一个管理器类请求一个 `Connection` 对象。管理器把一个空闲的连接交给该线程并把它标记为忙。如果没有空闲连接可用，它就打开一个。
一旦该线程用完连接，就把它归还给管理器，管理器随后可以关闭它或者把它加入池中。
管理器还会检查连接是否仍然存活，如果已经死掉就把它从池中移除。
连接池的缺点是它增加了服务器的负载，因为每个 `Connection` 对象都会创建一个新会话。这取决于你和你的应用程序的需求。

31.10. 连接池和数据源

JDBC 2 在一个称为 JDBC 2.0 Optional Package（也称为 JDBC 2.0 Standard Extension）的附加 API 中引入了标准的连接池特性。这些特性后来被包含在核心 JDBC 3 API 中。PostgreSQL JDBC 驱动在配合 JDBC 2.0 Optional Package（JDBC 2）与 JDK 1.3.x 一起编译，或者与 JDK 1.4 或更高版本（JDBC 3）一起编译时支持这些特性。大多数应用服务器都包含 JDBC 2.0 Optional Package，它也可以从 Sun 的 JDBC 下载站点[2]单独获得。

31.10.1. 概述

JDBC API 为连接池提供了一个客户端接口和一个服务器端接口。客户端接口是 `javax.sql.DataSource`，应用代码通常用它来获取一个池化的数据库连接。服务器端接口是 `javax.sql.ConnectionPoolDataSource`，大多数应用服务器将通过它与 PostgreSQL JDBC 驱动对接。

[2] http://java.sun.com/products/jdbc/download.html#spec

在应用服务器环境中，应用服务器的配置通常引用 PostgreSQL 的 ConnectionPoolDataSource 实现，而应用组件代码通常获取由应用服务器（而不是 PostgreSQL）提供的 DataSource 实现。

对于没有应用服务器的环境，PostgreSQL 提供了应用程序可以直接使用的两种 DataSource 实现。一种实现执行连接池化，而另一种只是通过 DataSource 接口提供对数据库连接的访问，不做任何池化。另外，这些实现不应在应用服务器环境中使用，除非应用服务器不支持 ConnectionPoolDataSource 接口。

31.10.2. 应用服务器：ConnectionPoolDataSource

PostgreSQL 为 JDBC 2 和 JDBC 3 各包含一个 ConnectionPoolDataSource 的实现，如表 31.1所示。

表 31.1. ConnectionPoolDataSource 实现

JDBC	实现类
2	org.postgresql.jdbc2.optional.ConnectionPool
3	org.postgresql.jdbc3.Jdbc3ConnectionPool

两种实现使用相同的配置方案。JDBC 要求 ConnectionPoolDataSource 通过 JavaBean 属性配置，如表 31.2所示，因此每种属性都有相应的 get 和 set 方法。

表 31.2. ConnectionPoolDataSource 配置属性

属性	类型	描述
serverName	String	PostgreSQL 数据库服务器的主机名
databaseName	String	PostgreSQL 数据库名
portNumber	int	PostgreSQL 数据库服务器监听的 TCP 端口（或者用 0 表示使用默认端口）
user	String	用来建立数据库连接的用户
password	String	用来建立数据库连接的口令
defaultAutoCommit	boolean	连接被交给调用方时，其自动提交是启用还是禁用。默认值为 false，即禁用自动提交。

许多应用服务器使用属性风格的语法来配置这些属性，因此把属性作为一块文本输入并不罕见。如果应用服务器提供单一区域来输入所有属性，它们可能像这样列出：

```
serverName=localhost
databaseName=test
user=testuser
password=testpassword
```

或者，如果用分号而不是换行符作为分隔符，它可能像这样：

```
serverName=localhost;databaseName=test;user=testuser;password=
testpassword
```

31.10.3. 应用程序：DataSource

PostgreSQL 为 JDBC 2 包含两种 `DataSource` 实现，为 JDBC 3 也包含两种，如表 31.3所示。池化实现在客户端调用 `close` 方法时实际上并不关闭连接，而是把连接归还到一个可供其他客户端使用的空闲连接池中。这避免了反复打开和关闭连接的任何开销，
并允许大量客户端共享少量数据库连接。

这里提供的池化数据源实现并不是世界上功能最丰富的。
举几个方面：连接直到池本身关闭之前永远不会被关闭；没有办法收缩池。而且，
为默认配置用户之外的用户请求的连接不会被池化。许多应用服务器提供更高级的池化特性，
并改用 `ConnectionPoolDataSource` 实现。

表 31.3. DataSource 实现

JDBC	是否池化	实现类
2	否	`org.postgresql.jdbc2.optional.SimpleDataSource`
2	是	`org.postgresql.jdbc2.optional.PoolingDataSource`
3	否	`org.postgresql.jdbc3.Jdbc3SimpleDataSource`
3	是	`org.postgresql.jdbc3.Jdbc3PoolingDataSource`

所有实现都使用相同的配置方案。JDBC 要求 `DataSource` 通过 JavaBean 属性配置，如表 31.4所示，因此每种属性都有相应的 get 和 set 方法。

表 31.4. DataSource 配置属性

属性	类型	描述
`serverName`	`String`	PostgreSQL 数据库服务器的主机名
`databaseName`	`String`	PostgreSQL 数据库名
`portNumber`	`int`	PostgreSQL 数据库服务器监听的 TCP 端口（或者用 0 表示使用默认端口）
`user`	`String`	用来建立数据库连接的用户
`password`	`String`	用来建立数据库连接的口令

池化实现需要一些额外的配置属性，它们显示在表 31.5中。

表 31.5. 池化 DataSource 的额外配置属性

属性	类型	描述
`dataSourceName`	`String`	每个池化 `DataSource` 都必须有一个唯一的名称。
`initialConnections`	`int`	池初始化时要创建的数据库连接数量。
`maxConnections`	`int`	允许的打开数据库连接的最大数量。当请求更多连接时，调用方将挂起，直到有连接被归还到池中。

例 31.9展示了使用池化 DataSource 的典型应用代码的一个例子。

例 31.9. DataSource 代码示例

初始化一个池化 DataSource 的代码可能像这样：

```
Jdbc3PoolingDataSource source = new Jdbc3PoolingDataSource();
source.setDataSourceName("A Data Source");
source.setServerName("localhost");
source.setDatabaseName("test");
source.setUser("testuser");
source.setPassword("testpassword");
source.setMaxConnections(10);
```

然后使用池中一个连接的代码可能像这样。注意，这些连接最终被关闭是至关重要的。否则池会“泄漏”连接，并最终把所有客户端都锁在门外。

```
Connection con = null;
try {
    con = source.getConnection();
    // use connection
} catch (SQLException e) {
    // log error
} finally {
    if (con != null) {
        try { con.close(); } catch (SQLException e) {}
    }
}
```

31.10.4. 数据源和 JNDI

所有 ConnectionPoolDataSource 和 DataSource 实现都可以存储在 JNDI 中。对于非池化实现，每次从 JNDI 检索该对象时都会创建一个新实例，其设置与被存储的实例相同。对于池化实现，只要它可用（例如，不是从 JNDI 检索该池的另一个不同 JVM），就会检索到同一个实例，否则会用相同的设置创建一个新实例。

在应用服务器环境中，通常存储在 JNDI 中的是应用服务器的 DataSource 实例，而不是 PostgreSQL 的 ConnectionPoolDataSource 实现。

在应用程序环境中，应用程序可以把 DataSource 存储在 JNDI 中，这样它就不必把对 DataSource 的引用提供给它可能需要使用它的所有应用组件。例 31.10展示了这样的一个例子。

例 31.10. DataSource JNDI 代码示例

初始化一个池化 DataSource 并把它添加到 JNDI 的应用代码可能像这样：

```
Jdbc3PoolingDataSource source = new Jdbc3PoolingDataSource();
source.setDataSourceName("A Data Source");
source.setServerName("localhost");
source.setDatabaseName("test");
source.setUser("testuser");
source.setPassword("testpassword");
source.setMaxConnections(10);
new InitialContext().rebind("DataSource", source);
```

然后使用池中一个连接的代码可能像这样：

```
Connection con = null;
try {
    DataSource source = (DataSource)new InitialContext().lookup("Data
Source");
    con = source.getConnection();
    // use connection
} catch (SQLException e) {
    // log error
} catch (NamingException e) {
    // DataSource wasn't found in JNDI
} finally {
    if (con != null) {
        try { con.close(); } catch (SQLException e) {}
    }
}
```

31.11. 延伸阅读

如果你还没有读过，建议你阅读 JDBC API 文档（随 Sun 的 JDK 提供）和 JDBC 规范。两者都可以从http://java.sun.com/products/jdbc/index.html获得。

http://jdbc.postgresql.org 包含本章未收录的最新信息，并且还提供预编译的驱动。

第 32 章 信息模式

信息模式由一组视图组成，这些视图包含当前数据库中所定义对象的信息。信息模式由 SQL 标准定义，因此可以期望它是可移植且稳定的——不像系统目录那样是 PostgreSQL 特有的、且依照实现考虑来建模的。不过，信息模式视图并不包含 PostgreSQL 特有功能的信息；要查询这些，你需要查询系统目录或其他 PostgreSQL 特有的视图。

32.1. 模式

信息模式本身是一个名为`information_schema`的模式。这个模式自动存在于所有数据库中。这个模式的拥有者是集簇中的初始数据库用户，并且该用户自然地拥有这个模式上的所有权限，包括删除它的能力（但是这样节省的空间是很小的）。

默认情况下，信息模式不在模式搜索路径中，因此你需要使用限定名访问其中的所有对象。因为信息模式中的某些对象的名称是可能出现在用户应用中的一般名称，如果你想把该信息模式放在路径中，你应该小心。

32.2. 数据类型

信息模式视图的列使用在信息模式中定义的特殊数据类型。这些类型被定义为普通内置类型之上的简单域。不应在信息模式之外使用这些类型，但如果应用要从信息模式中查询数据，就必须能够处理它们。

这些类型为：

`cardinal_number`

一种非负整数。

`character_data`

一种字符串（没有指定最大长度）。

`sql_identifier`

一种字符串。该类型用于 SQL 标识符，其他任何文本数据则使用类型`character_data`。

`time_stamp`

在类型`timestamp`之上的一个域。

信息模式中的每一列都具有这四种类型之一。

布尔（真/假）数据在信息模式中由一个`character_data`类型的列表示，该列只包含`YES`或`NO`。（信息模式是在类型`boolean`被纳入 SQL 标准之前设计出来的，因此需要采用这种约定以保持向后兼容。）

32.3. information_schema_catalog_name

`information_schema_catalog_name` 是一个表，它始终只包含一行一列，用于存放当前数据库的名称（按 SQL 术语称为当前目录）。

表 32.1. information_schema_catalog_name 的列

名称	数据类型	描述
catalog_name	sql_identifier	包含这个信息模式的数据库名称

32.4. applicable_roles

视图applicable_roles标识当前用户作为成员的所有组。（角色和组是同一回事。）
一般来说，最好使用视图enabled_roles而不是这个视图；另见那里。

表 32.2. applicable_roles 的列

名称	数据类型	描述
grantee	sql_identifier	总是当前用户的名称
role_name	sql_identifier	一个组的名称
is_grantable	character_data	对应 PostgreSQL 不支持的特性

32.5. check_constraints

视图check_constraints包含所有检查约束，无论它们是定义在表上还是定义在域上，
只要它们归当前用户所有（表或域的拥有者就是约束的拥有者）。

表 32.3. check_constraints 的列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含约束的数据库名称(总是当前数据库)
constraint_schema	sql_identifier	包含约束的模式的名称
constraint_name	sql_identifier	约束的名称
check_clause	character_data	该检查约束的检查表达式

32.6. column_domain_usage

视图column_domain_usage
标识所有使用定义在当前数据库中并且被当前用户拥有的域的列（表列或视图列）。

表 32.4. column_domain_usage 的列

名称	数据类型	描述
domain_catalog	sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名称
domain_name	sql_identifier	域名称
table_catalog	sql_identifier	包含表的数据库的名称(总是当前数据库)
table_schema	sql_identifier	包含表的模式的名称
table_name	sql_identifier	表的名称
column_name	sql_identifier	列名称

32.7. column_privileges

视图column_privileges标识所有授予给当前用户或者由当前用户授予的列上的权限。对每一个列、授予者、被授予者的组合只有一行。授予给组的权限在视图role_column_grants中标识。

在PostgreSQL中，你只能对整表授权，而不能对单个列授权。因此，这个视图包含和table_privileges相同的信息，只是通过每个适当表中每一列一行的方式来表示，但它只覆盖可能有列粒度的权限类型：SELECT、INSERT、UPDATE、REFERENCES。如果你想让你的应用为将来可能的发展做好准备，在涉及这些权限类型之一时，使用这个视图而不是table_privileges通常是正确的选择。

表 32.5. column_privileges 的列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的用户名称
grantee	sql_identifier	权限被授予到的用户或组的名称
table_catalog	sql_identifier	包含该列的表所在的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该列的表所在的模式名称
table_name	sql_identifier	包含该列的表名称
column_name	sql_identifier	列的名称
privilege_type	character_data	权限的类型：SELECT、INSERT、UPDATE或REFERENCES
is_grantable	character_data	如果该权限是可授予的，则为YES，否则为NO

注意，grantee 列不区分用户和组。如果有同名的用户和组，很遗憾没有办法区分它们。PostgreSQL 的未来版本可能会禁止创建同名的用户和组。

32.8. column_udt_usage

视图column_udt_usage标识所有使用被当前用户拥有的数据类型的列。注意在PostgreSQL中，内置数据类型的行为和用户定义的类型相似，因此它们也被包括在这里。详见第 32.9 节。

表 32.6. column_udt_usage 的列

名称	数据类型	描述
udt_catalog	sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的数据库名称（总是当前数据库）
udt_schema	sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的模式名称
udt_name	sql_identifier	该列数据类型（如果适用，底层的域类型）的名称
table_catalog	sql_identifier	包含表的数据库的名称(总是当前数据库)
table_schema	sql_identifier	包含表的模式的名称

名称	数据类型	描述
table_name	sql_identifier	表的名称
column_name	sql_identifier	列名称

32.9. columns

视图columns包含数据库中有关所有表列（或视图列）的信息。系统列（oid等）不被包括在内。只有那些当前用户能够访问（作为拥有者或具有某些权限）的列才被显示。

表 32.7. columns 的列

名称	数据类型	描述
table_catalog	sql_identifier	包含表的数据库的名称(总是当前数据库)
table_schema	sql_identifier	包含表的模式的名称
table_name	sql_identifier	表的名称
column_name	sql_identifier	列名称
ordinal_position	cardinal_number	该列在表内的顺序位置（从 1 开始计）
column_default	character_data	该列的默认表达式（如果当前用户不是包含该列的表的拥有者，则为空）
is_nullable	character_data	如果该列可以为空，则为YES，否则为NO。一个非空约束是让一列成为不能为空的方法，但还有其他方法。
data_type	character_data	如果该列的数据类型是一种内置类型，则为该列的数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况下该类型被标识在udt_name和相关列中）。如果该列基于一个域，这一列引用该域底层的类型（该列被标识在domain_name和相关列中）。
character_maximum_length	cardinal_number	如果data_type标识字符类型或位串类型，这里是声明的最大长度；如果未声明最大长度，或者对于所有其他数据类型，则为空。
character_octet_length	cardinal_number	如果data_type标识字符类型，这里是一个数据值的最大可能字节长度（这一点PostgreSQL用户不必关心）；对于所有其他数据类型则为空。
numeric_precision	cardinal_number	如果data_type标识一种数字类型，这列包含该列类型的（声明的或隐式的）精度。精度指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制

名称	数据类型	描述
		（基于 2）。对于所有其他数据类型，这一列为空。
numeric_precision_radix	cardinal_number	如果data_type标识一种数字类型，这一列指示numeric_precision和numeric_scale列中的值是基于什么来表示。该值为 2 或 10。对于所有其他数据类型，这一列为空。
numeric_scale	cardinal_number	如果data_type标识一种准确数字类型，这列包含该列类型的（声明的或隐式的）标度。标度表示小数点右侧的有效数字位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
datetime_precision	cardinal_number	如果data_type标识一种日期、时间或时间间隔类型，这一列包含声明的精度；对于所有其他数据类型或未声明精度时则为空。
interval_type	character_data	尚未实现
interval_precision	character_data	还未被实现
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
collation_schema	sql_identifier	对应 PostgreSQL 不支持的特性
collation_name	sql_identifier	对应 PostgreSQL 不支持的特性
domain_catalog	sql_identifier	如果该列有一个域类型，这里是该域所在的数据库名（总是当前数据库），否则为空。
domain_schema	sql_identifier	如果该列有一个域类型，这里是该域所在的模式名，否则为空。
domain_name	sql_identifier	如果该列有一个域类型，这里是该域的名称，否则为空。
udt_catalog	sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的数据库名称（总是当前数据库）
udt_schema	sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的模式名称
udt_name	sql_identifier	该列数据类型（如果适用，底层的域类型）的名称
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
maximum_cardinality	cardinal_number	总是为空，因为 PostgreSQL 中数组的最大基数始终不受限制

名称	数据类型	描述
dtd_identifier	sql_identifier	该列的数据类型描述符的一个标识符，在从属于该表的数据类型描述符之中唯一。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。
is_self_referencing	character_data	对应 PostgreSQL 不支持的特性

因为在 SQL 中有多种方式定义数据类型，而PostgreSQL还包含额外的方式来定义数据类型，它们在信息模式中的表示可能有点困难。列data_type应该标识列的底层内置类型。在PostgreSQL中，这表示定义在系统目录模式pg_catalog中的类型。如果应用能够特别地（例如以不同方式格式化数字类型或使用精度列中的数据）处理众所周知的内置类型，这列可能会有用。列udt_name、udt_schema和udt_catalog总是标识列的底层数据类型，即使该列是基于一个域的（因为PostgreSQL对待内置类型和用户定义类型的方式是一样的，内置类型也出现在这里。这是 SQL 标准的一种扩展）。如果一个应用想要根据该类型以不同的方式处理数据，就应该使用这些列，因为在那种情况下即使该列真地基于一个域也没有关系。如果该列是基于一个域，该域的标识被存储在列domain_name、domain_schema和domain_catalog。如果你想要把列和它们相关的数据类型配对并且把域视作单独的类型，你可以写coalesce(domain_name, udt_name)等等。

32.10. constraint_column_usage

视图constraint_column_usage标识当前数据库中被某种约束使用的所有列。只显示那些位于当前用户拥有的表中的列。对于检查约束，此视图标识检查表达式中使用的列。对于外键约束，此视图标识外键所引用的列。对于唯一约束或主键约束，此视图标识受该约束限制的列。

表 32.8. constraint_column_usage 的列

名称	数据类型	描述
table_catalog	sql_identifier	包含被某个约束使用的列的表所在的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含被某个约束使用的列的表所在的模式名称
table_name	sql_identifier	包含被某个约束使用的列的表名称
column_name	sql_identifier	被某个约束使用的列的名称
constraint_catalog	sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名称
constraint_name	sql_identifier	约束的名称

32.11. constraint_table_usage

视图constraint_table_usage标识在当前数据库中被某个约束使用的所有表（这与视图table_constraints不同，它标识哪些表约束定义在哪些表上）。对于一个外键约束，这个视图标识该外键引用的表。对于一个唯一或主键约束，这个视图仅标识该约束属于的表。检查约束和非空约束不被包括在这个视图中。

表 32.9. constraint_table_usage 的列

名称	数据类型	描述
table_catalog	sql_identifier	包含被某个约束使用的表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含被某个约束使用的表的模式名称
table_name	sql_identifier	被某个约束使用的表的名称
constraint_catalog	sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名称
constraint_name	sql_identifier	约束的名称

32.12. data_type_privileges

视图data_type_privileges
标识当前用户能够访问（作为被描述对象的拥有者或者具有其上的某种权限）
的所有数据类型描述符。只要一个数据类型被用在一个表列、
一个域或一个函数（作为参数或返回类型）
就会生成一个数据类型描述符并且在那个实例中存储一些有关该数据类型如何被使用的信息（例如，声明的最大长度，如果适用）。每一个数据类型描述符被赋予一个任意的标识符，
它在被赋予给一个对象（表、域、函数）的数据类型描述符中唯一。
这个视图对于应用可能没什么用，但是它被用于定义信息模式中的一些其他视图。

表 32.10. data_type_privileges 的列

名称	数据类型	描述
object_catalog	sql_identifier	包含该被描述对象的数据库名称（总是当前数据库）
object_schema	sql_identifier	包含被描述对象的模式名称
object_name	sql_identifier	被描述对象的名称
object_type	character_data	被描述对象的类型：TABLE（数据类型描述符从属于该表的一个列）、DOMAIN（数据类型描述符从属于该域）、ROUTINE（数据类型描述符从属于该函数的一个参数或返回数据类型）之一。
dtd_identifier	sql_identifier	数据类型描述符的标识符，在同一对象的各数据类型描述符之间唯一。

32.13. domain_constraints

视图domain_constraints包含所有属于当前用户拥有的域的约束。

表 32.11. domain_constraints 的列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名称（总是当前数据库）

名称	数据类型	描述
constraint_schema	sql_identifier	包含该约束的模式名称
constraint_name	sql_identifier	约束的名称
domain_catalog	sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名称
domain_name	sql_identifier	域的名称
is_deferrable	character_data	如果该约束是可延迟的，则为YES，否则为NO
initially_deferred	character_data	如果该约束是可延迟的且初始就被延迟，则为YES，否则为NO

32.14. domain_udt_usage

视图domain_udt_usage标识所有使用被当前用户拥有的数据类型的列。
注意在PostgreSQL中，内置数据类型的行为相似于用户定义的类型，因此它们也被包括在这里。

表 32.12. domain_udt_usage 的列

名称	数据类型	描述
udt_catalog	sql_identifier	该域数据类型被定义的数据库名称（总是当前数据库）
udt_schema	sql_identifier	该域数据类型被定义的模式名称
udt_name	sql_identifier	该域数据类型的名称
domain_catalog	sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名称
domain_name	sql_identifier	域的名称

32.15. domains

视图domains包含当前数据库中定义的所有域。

表 32.13. domains 的列

名称	数据类型	描述
domain_catalog	sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema	sql_identifier	包含该域的模式名称
domain_name	sql_identifier	域的名称
data_type	character_data	该域的数据类型如果是一种内置类型，这里是该域的数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在udt_name和相关列中）。
character_maximum_length	cardinal_number	如果该域有一个字符或位串类型，这里是声明的最大长度；如果没有

名称	数据类型	描述
		声明最大长度，则对于所有其他数据类型为空。
character_octet_length	cardinal_number	如果该域有一个字符类型，如果该域有一个字符类型，这里是一个数据值的最大可能字节长度（这一点PostgreSQL用户不必关心）；对于所有其他数据类型则为空。
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
collation_schema	sql_identifier	对应 PostgreSQL 不支持的特性
collation_name	sql_identifier	对应 PostgreSQL 不支持的特性
numeric_precision	cardinal_number	如果该域有一种数字类型，这列包含这个域类型的（声明的或隐式的）精度。精度指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
numeric_precision_radix	cardinal_number	如果该域有一种数字类型，这一列指示numeric_precision和numeric_scale列中的值是基于什么来表示。该值为 2 或 10。对于所有其他数据类型，这一列为空。
numeric_scale	cardinal_number	如果该域有一种准确数字类型，这列包含这个域类型的（声明的或隐式的）比例。标度表示小数点右侧的有效数字位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
datetime_precision	cardinal_number	如果该域有一种日期、时间或时间间隔类型，这一列包含声明的精度；对于所有其他数据类型或未声明精度时则为空。
interval_type	character_data	尚未实现
interval_precision	character_data	还未被实现
domain_default	character_data	该域的默认表达式
udt_catalog	sql_identifier	该域数据类型被定义的数据库名称（总是当前数据库）
udt_schema	sql_identifier	该域数据类型被定义的模式名称
udt_name	sql_identifier	该域数据类型的名称
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性

名称	数据类型	描述
maximum_cardinality	cardinal_number	总是为空，因为 PostgreSQL 中数组的最大基数始终不受限制
dtd_identifier	sql_identifier	该域的数据类型描述符的一个标识符，在从属于该域的数据类型描述符之中唯一（这不重要，因为一个域只包含一个数据类型描述符）。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。

32.16. element_types

视图element_types包含数组元素的数据类型描述符。当一个表列、域、函数参数或函数返回值被定义为一种数组类型时，相应的信息模式视图只在列data_type中包含ARRAY。要获得该数组元素类型的信息，你可以连接该相应的视图和这个视图。例如，要显示一个表的列及其数据类型和数组元素类型（如果适用），你可以：

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.
element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.
dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_
type, e.array_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

这个视图只包括当前用户能够访问（作为拥有者或具有某些权限）的对象。

表 32.14. element_types 的列

名称	数据类型	描述
object_catalog	sql_identifier	包含使用被描述的数组的对象的数据库名称（总是当前数据库）
object_schema	sql_identifier	包含使用被描述数组的对象的模式名称
object_name	sql_identifier	使用被描述数组的对象的名称
object_type	character_data	使用被描述数组的对象的类型：TABLE（该数组被该表的一个列使用）、DOMAIN（该数组被该域使用）、ROUTINE（该数组被该函数的一个参数或返回数据类型使用）之一。
array_type_identifier	sql_identifier	被描述数组的数据类型描述符的标识符。用它与其他信息模式视图的dtd_identifier列进行连接。
data_type	character_data	如果数组元素的数据类型是内置类型，这里是数组元素的数据类型，否则为USER-DEFINED（在那种情况下，该类型被标识在udt_name和相关列中）。

名称	数据类型	描述
character_maximum_length	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的数组元素数据类型
character_octet_length	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的数组元素数据类型
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
collation_schema	sql_identifier	对应 PostgreSQL 不支持的特性
collation_name	sql_identifier	对应 PostgreSQL 不支持的特性
numeric_precision	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的数组元素数据类型
numeric_precision_radix	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的数组元素数据类型
numeric_scale	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的数组元素数据类型
datetime_precision	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的数组元素数据类型
interval_type	character_data	总是为空，因为这种信息不适用于 PostgreSQL 中的数组元素数据类型
interval_precision	character_data	总是为空，因为这种信息不适用于 PostgreSQL 中的数组元素数据类型
domain_default	character_data	尚未实现
udt_catalog	sql_identifier	元素的数据类型所在的数据库名称（总是当前数据库）
udt_schema	sql_identifier	定义元素的数据类型的模式的名称
udt_name	sql_identifier	元素的数据类型名
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
maximum_cardinality	cardinal_number	总是为空，因为 PostgreSQL 中数组的最大基数始终不受限制
dtd_identifier	sql_identifier	该元素的数据类型描述符的标识符。当前没有用。

32.17. enabled_roles

视图enabled_roles标识当前用户所属的所有组。（角色和组是同一回事。）这个视图与 applicable_roles 的区别在于，将来可能会有一种在会话中启用和禁用组的机制。在那种情况下，这个视图标识的是当前已启用的那些组。

表 32.15. enabled_roles 的列

名称	数据类型	描述
role_name	sql_identifier	一个组的名称

32.18. key_column_usage

视图key_column_usage标识当前数据库中所有被某种唯一、主键或外键约束限制的列。
检查约束不被包括在这个视图中。
只有那些当前用户能够访问的列才会被显示（作为拥有者或具有某些权限）。

表 32.16. key_column_usage 的列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名称
constraint_name	sql_identifier	约束的名称
table_catalog	sql_identifier	包含被这个约束限制的列的表所在的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该列的表所在的模式名称 that is restricted by some constraint
table_name	sql_identifier	包含该列的表名称 that is restri cted by some constraint
column_name	sql_identifier	列的名称 that is restricted by some constraint
ordinal_position	cardinal_number	该列在约束键中的顺序位置（从 1 开始计）

32.19. parameters

视图parameters包含当前数据库中所有函数的参数的有关信息。
只有那些当前用户能够访问（作为拥有者或具有某些权限）的函数才会被显示。

表 32.17. parameters 的列

名称	数据类型	描述
specific_catalog	sql_identifier	包含该函数的数据库名称(总是当前数据库)
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	函数的“特定名称”。详见第 32. 26 节。
ordinal_position	cardinal_number	该参数在函数参数列表中的顺序位置（从 1 开始计数）

名称	数据类型	描述
parameter_mode	character_data	总是IN，表示输入参数（将来可能会有其他参数模式。）
is_result	character_data	对应 PostgreSQL 不支持的特性
as_locator	character_data	对应 PostgreSQL 不支持的特性
parameter_name	sql_identifier	总是为空，因为 PostgreSQL 不支持命名参数
data_type	character_data	该参数的数据类型如果是一种内置类型，这里是该参数的数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在udt_name和相关列中）。
character_maximum_length	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的参数数据类型
character_octet_length	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的参数数据类型
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
collation_schema	sql_identifier	对应 PostgreSQL 不支持的特性
collation_name	sql_identifier	对应 PostgreSQL 不支持的特性
numeric_precision	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的参数数据类型
numeric_precision_radix	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的参数数据类型
numeric_scale	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的参数数据类型
datetime_precision	cardinal_number	总是为空，因为这种信息不适用于 PostgreSQL 中的参数数据类型
interval_type	character_data	总是为空，因为这种信息不适用于 PostgreSQL 中的参数数据类型
interval_precision	character_data	总是为空，因为这种信息不适用于 PostgreSQL 中的参数数据类型
udt_catalog	sql_identifier	该参数的数据类型所在的数据库名称（总是当前数据库）
udt_schema	sql_identifier	该参数的数据类型所在的模式名称
udt_name	sql_identifier	该参数的数据类型的名称
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
maximum_cardinality	cardinal_number	总是为空，因为 PostgreSQL 中数组的最大基数始终不受限制
dtd_identifier	sql_identifier	该参数的数据类型描述符的一个标识符，在从属于该函数的数据类型描述符之中唯一。这主要用于与这

名称	数据类型	描述
		类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。

32.20. referential_constraints

视图referential_constraints
包含当前数据库中属于当前用户拥有的表的所有引用（外键）约束。

表 32.18. referential_constraints 的列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含约束的数据库名称(总是当前数据库)
constraint_schema	sql_identifier	包含约束的模式的名称
constraint_name	sql_identifier	约束的名称
unique_constraint_catalog	sql_identifier	包含该外键约束所引用的唯一或主键约束的数据库名称（总是当前数据库）
unique_constraint_schema	sql_identifier	包含该外键约束所引用的唯一或主键约束的模式名称
unique_constraint_name	sql_identifier	包含该外键约束所引用的唯一或主键约束的名称
match_option	character_data	外键约束的匹配选项：FULL、PARTIAL或NONE。
update_rule	character_data	外键约束的更新规则：CASCADE、SET NULL、SET DEFAULT、RESTRICT或 NO ACTION。
delete_rule	character_data	外键约束的删除规则：CASCADE、SET NULL、SET DEFAULT、RESTRICT或 NO ACTION。

32.21. role_column_grants

视图role_column_grants标识所有在列上授予给当前用户作为成员的组的权限。
更多信息可以在column_privileges中找到。

表 32.19. role_column_grants 的列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的用户名称
grantee	sql_identifier	权限被授予到的组的名称
table_catalog	sql_identifier	包含该列的表所在的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该列的表所在的模式名称
table_name	sql_identifier	包含该列的表名称

名称	数据类型	描述
column_name	sql_identifier	列的名称
privilege_type	character_data	权限的类型：SELECT、INSERT、UPDATE或REFERENCES
is_grantable	character_data	如果该权限是可授予的，则为YES，否则为NO

32.22. role_routine_grants

视图role_routine_grants标识所有在函数上授予给当前用户作为成员的组的权限。更多信息可以在routine_privileges中找到。

表 32.20. role_routine_grants 的列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的用户名称
grantee	sql_identifier	权限被授予到的组的名称
specific_catalog	sql_identifier	包含该函数的数据库名称(总是当前数据库)
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	函数的“特定名称”。详见第 32.26 节。
routine_catalog	sql_identifier	包含该函数的数据库名称(总是当前数据库)
routine_schema	sql_identifier	包含函数的模式的名称
routine_name	sql_identifier	该函数的名称（在重载的情况下可能会重复）
privilege_type	character_data	总是EXECUTE（函数唯一的权限类型）
is_grantable	character_data	如果该权限是可授予的，则为YES，否则为NO

32.23. role_table_grants

视图role_table_grants标识所有在表或视图上授予给当前用户作为成员的组的权限。更多信息可以在table_privileges中找到。

表 32.21. role_table_grants 的列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的用户名称
grantee	sql_identifier	权限被授予到的组的名称
table_catalog	sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名称
table_name	sql_identifier	表的名称

名称	数据类型	描述
privilege_type	character_data	权限的类型：SELECT、DELETE、INSERT、UPDATE、REFERENCES、RULE或TRIGGER
is_grantable	character_data	如果该权限是可授予的，则为YES，否则为NO
with_hierarchy	character_data	对应 PostgreSQL 不支持的特性

32.24. role_usage_grants

视图role_usage_grants旨在标识在多种对象上授予给当前用户作为成员的组的USAGE权限。在PostgreSQL中，这目前只适用于域，而且由于域在PostgreSQL中没有真正的权限，这个视图为空。更多信息可以在usage_privileges中找到。将来，这个视图可能会包含更多有用的信息。

表 32.22. role_usage_grants 的列

名称	数据类型	描述
grantor	sql_identifier	将来：授予该权限的用户名称
grantee	sql_identifier	将来：权限被授予到的组的名称
object_catalog	sql_identifier	包含该对象的数据库名称（总是当前数据库）
object_schema	sql_identifier	包含该对象的模式名称
object_name	sql_identifier	对象的名称
object_type	character_data	将来：对象的类型
privilege_type	character_data	总是USAGE
is_grantable	character_data	如果该权限是可授予的，则为YES，否则为NO

32.25. routine_privileges

视图routine_privileges标识所有授予给当前用户或者由当前用户授予的函数上的权限。对于每一种函数、授予者和被授予者的组合，这里都有一行。授予给组的权限在视图role_routine_grants中标识。

表 32.23. routine_privileges 的列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的用户名称
grantee	sql_identifier	权限被授予到的用户或组的名称
specific_catalog	sql_identifier	包含该函数的数据库名称(总是当前数据库)
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	函数的“特定名称”。详见第 32.26 节。
routine_catalog	sql_identifier	包含该函数的数据库名称(总是当前数据库)
routine_schema	sql_identifier	包含函数的模式的名称

名称	数据类型	描述
routine_name	sql_identifier	该函数的名称（在重载的情况下可能会重复）
privilege_type	character_data	总是EXECUTE（函数唯一的权限类型）
is_grantable	character_data	如果该权限是可授予的，则为YES，否则为NO

注意，grantee 列不区分用户和组。如果有同名的用户和组，很遗憾没有办法区分它们。PostgreSQL 的未来版本可能会禁止创建同名的用户和组。

32.26. routines

视图routines包含当前数据库中所有的函数。
只有那些当前用户能够访问（作为拥有者或具有某些权限）的函数才会被显示。

表 32.24. routines 的列

名称	数据类型	描述
specific_catalog	sql_identifier	包含该函数的数据库名称(总是当前数据库)
specific_schema	sql_identifier	包含函数的模式的名称
specific_name	sql_identifier	该函数的“专用名”。这是一个在模式中唯一标识该函数的名称，即使该函数真正的名称已经被重载。专用名的格式尚未被定义，它应当仅被用来与指定例程名称的其他实例进行比较。
routine_catalog	sql_identifier	包含该函数的数据库名称(总是当前数据库)
routine_schema	sql_identifier	包含函数的模式的名称
routine_name	sql_identifier	该函数的名称（在重载的情况下可能会重复）
routine_type	character_data	总是FUNCTION（将来可能会有其他类型的例程。）
module_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
module_schema	sql_identifier	对应 PostgreSQL 不支持的特性
module_name	sql_identifier	对应 PostgreSQL 不支持的特性
udt_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
udt_schema	sql_identifier	对应 PostgreSQL 不支持的特性
udt_name	sql_identifier	对应 PostgreSQL 不支持的特性
data_type	character_data	该函数的返回数据类型如果是一种内置类型，这里是该数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在type_udt_name和相关列中）。
character_maximum_length	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL 中的返回数据类型

名称	数据类型	描述
character_octet_length	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL 中的返回数据类型
character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性
character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
collation_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
collation_schema	sql_identifier	对应 PostgreSQL 不支持的特性
collation_name	sql_identifier	对应 PostgreSQL 不支持的特性
numeric_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL 中的返回数据类型
numeric_precision_radix	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL 中的返回数据类型
numeric_scale	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL 中的返回数据类型
datetime_precision	cardinal_number	总是为空，因为这种信息不适用于PostgreSQL 中的返回数据类型
interval_type	character_data	总是为空，因为这种信息不适用于PostgreSQL 中的返回数据类型
interval_precision	character_data	总是为空，因为这种信息不适用于PostgreSQL 中的返回数据类型
type_udt_catalog	sql_identifier	该函数的返回数据类型所在的数据库名（总是当前数据库）。
type_udt_schema	sql_identifier	该函数的返回数据类型所在的模式名。
type_udt_name	sql_identifier	该函数的返回数据类型的名字。
scope_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
scope_schema	sql_identifier	对应 PostgreSQL 不支持的特性
scope_name	sql_identifier	对应 PostgreSQL 不支持的特性
maximum_cardinality	cardinal_number	总是为空，因为 PostgreSQL 中数组的最大基数始终不受限制
dtd_identifier	sql_identifier	该函数返回数据类型的数据类型描述符的一个标识符，在从属于该函数的数据类型描述符之中唯一。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。
routine_body	character_data	如果该函数是一个 SQL 函数，则为SQL，否则为EXTERNAL。
routine_definition	character_data	该函数的源文本（如果当前用户不是该函数的拥有者，则为空）（根据 SQL 标准，只有routine_body为SQL时这一列才适用。但是在PostgreSQL中，它将会包含该函数被创建时所指定的任何源文本。）
external_name	character_data	如果这个函数是一个 C 函数，则为该函数的外部名称（链接符

名称	数据类型	描述
		号），否则为空（这会产生和显示在routine_definition中相同的值）。
external_language	character_data	该函数所用的语言
parameter_style	character_data	总是GENERAL（SQL 标准定义了其他参数风格，但在 PostgreSQL 中不可用）
is_deterministic	character_data	如果该函数被声明为不变（在 SQL 标准中被称为确定性的），则为YES，否则为NO（你不能通过该信息模式查询在 PostgreSQL 中可用的其他易变性级别）。
sql_data_access	character_data	总是MODIFIES，表示该函数可能修改 SQL 数据。这种信息对 PostgreSQL 没有用处。
is_null_call	character_data	如果该函数在任一参数为空时自动返回空值，则为YES，否则为NO。
sql_path	character_data	对应 PostgreSQL 不支持的特性
schema_level_routine	character_data	总是YES（反例是一个用户定义类型的方法，这是 PostgreSQL 不可用的一种特性）。
max_dynamic_result_sets	cardinal_number	对应 PostgreSQL 不支持的特性
is_user_defined_cast	character_data	对应 PostgreSQL 不支持的特性
is_implicitly_invocable	character_data	对应 PostgreSQL 不支持的特性
security_type	character_data	如果该函数以当前用户的权限运行，则为INVOKER；如果该函数以定义它的用户的权限运行，则为DEFINER。
to_sql_specific_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
to_sql_specific_schema	sql_identifier	对应 PostgreSQL 不支持的特性
to_sql_specific_name	sql_identifier	对应 PostgreSQL 不支持的特性
as_locator	character_data	对应 PostgreSQL 不支持的特性

32.27. schemata

视图schemata包含当前数据库中当前用户拥有的所有模式。

表 32.25. schemata 的列

名称	数据类型	描述
catalog_name	sql_identifier	该模式所在的数据库名称（总是当前数据库）
schema_name	sql_identifier	模式的名称
schema_owner	sql_identifier	模式拥有者的名称
default_character_set_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
default_character_set_schema	sql_identifier	对应 PostgreSQL 不支持的特性

名称	数据类型	描述
default_character_set_name	sql_identifier	对应 PostgreSQL 不支持的特性
sql_path	character_data	对应 PostgreSQL 不支持的特性

32.28. sql_features

表sql_features包含的信息指示了哪些 SQL 标准中定义的正式特性被PostgreSQL所支持。这和附录 D中的信息一样。这里你也能找到一些额外的背景信息。

表 32.26. sql_features 的列

名称	数据类型	描述
feature_id	character_data	特性的标识符字符串
feature_name	character_data	特性的描述性名称
sub_feature_id	character_data	该子特性的标识符字符串，或者如果不是一个子特性则为一个长度为零的字符串
sub_feature_name	character_data	该子特性的描述性名称，或者如果不是一个子特性则为一个长度为零的字符串
is_supported	character_data	如果当前版本的 PostgreSQL 完全支持该特性，则为YES，否则为NO
is_verified_by	character_data	总是为空，因为 PostgreSQL 开发组没有对特性的一致性执行正式的测试
comments	character_data	可能会是关于该特性被支持状态的一段注释

32.29. sql_implementation_info

表sql_implementation_info包含的信息指示剩下的由 SQL 标准实现定义的多个方面。这类信息主要用来在 ODBC 接口的情境中使用；其它接口的用户可能将发现这类信息用处不大。由于这个原因，个体实现信息项没有在这里描述，你将会在 ODBC 接口的描述中找到它们。

表 32.27. sql_implementation_info 的列

名称	数据类型	描述
implementation_info_id	character_data	实现信息项的标识符字符串
implementation_info_name	character_data	实现信息项的描述性名称
integer_value	cardinal_number	实现信息项的整数值；如果该值在列character_value中给出，则本列为空
character_value	character_data	实现信息项的字符值；如果该值在列integer_value中给出，则本列为空
comments	character_data	可能是从属于该实现信息项的一段注释

32.30. sql_languages

表sql_languages为PostgreSQL支持的每一种 SQL 语言绑定包含一行。PostgreSQL支持直接 SQL 和 C 中的嵌入式 SQL；从此表中只能了解到这些信息。

表 32.28. sql_languages 的列

名称	数据类型	描述
sql_language_source	character_data	语言定义来源的名称；总是ISO 9075，即 SQL 标准
sql_language_year	character_data	sql_language_source引用的标准获批的年份；当前为1999
sql_language_comformance	character_data	语言绑定的标准一致性级别。对于 ISO 9075:1999，这总是CORE。
sql_language_integrity	character_data	总是为空（此值与 SQL 标准的较早版本有关。）
sql_language_implementation	character_data	总是为空
sql_language_binding_style	character_data	语言绑定方式，为DIRECT或EMBEDDED
sql_language_programming_language	character_data	如果绑定方式为EMBEDDED，则为所用编程语言，否则为空。PostgreSQL 只支持 C 语言。

32.31. sql_packages

表sql_packages包含PostgreSQL支持 SQL 标准所定义的哪些功能包的信息。功能包的背景信息请参阅附录 D。

表 32.29. sql_packages 的列

名称	数据类型	描述
feature_id	character_data	功能包的标识符字符串
feature_name	character_data	功能包的描述性名称
is_supported	character_data	如果当前版本的 PostgreSQL 完全支持此功能包，则为YES，否则为NO
is_verified_by	character_data	总是为空，因为 PostgreSQL 开发组没有对特性的一致性执行正式的测试
comments	character_data	关于此功能包支持状态的可选注释

32.32. sql_sizing

表sql_sizing包含有关PostgreSQL中多种尺寸限制和最大值的信息。这类信息主要用来在 ODBC 接口的情境中使用；其它接口的用户可能将发现这类信息用处不大。由于这个原因，单个尺寸项没有在这里描述，你将会在 ODBC 接口的描述中找到它们。

表 32.30. sql_sizing 的列

名称	数据类型	描述
sizing_id	cardinal_number	尺寸项的标识符
sizing_name	character_data	尺寸项的描述性名称
supported_value	cardinal_number	尺寸项的值，如果尺寸是不受限制或不能确定的则为 0，如果尺寸项适用的特性不受支持则为空
comments	character_data	可能是从属于尺寸项的一段注释

32.33. sql_sizing_profiles

表sql_sizing_profiles包含 SQL 标准的各类配置规范所要求的sql_sizing值的信息。PostgreSQL 不跟踪任何 SQL 配置规范，因此此表为空。

表 32.31. sql_sizing_profiles 的列

名称	数据类型	描述
sizing_id	cardinal_number	尺寸项的标识符
sizing_name	character_data	尺寸项的描述性名称
profile_id	character_data	配置规范的标识符字符串
required_value	cardinal_number	SQL 配置规范对该尺寸限制项要求的值；如果配置规范对此项不设限制，则为 0；如果配置规范不要求此项所适用的任何功能，则为空
comments	character_data	关于此配置规范中该尺寸限制项的可选注释

32.34. table_constraints

视图table_constraints包含属于当前用户拥有的表的所有约束。

表 32.32. table_constraints 的列

名称	数据类型	描述
constraint_catalog	sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema	sql_identifier	包含该约束的模式名称
constraint_name	sql_identifier	约束的名称
table_catalog	sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名称
table_name	sql_identifier	表的名称
constraint_type	character_data	约束的类型：CHECK、FOREIGN KEY、PRIMARY KEY或UNIQUE
is_deferrable	character_data	如果该约束是可延迟的，则为YES，否则为NO

名称	数据类型	描述
initially_deferred	character_data	如果该约束是可延迟的且初始就被延迟，则为YES，否则为NO

32.35. table_privileges

视图table_privileges标识所有授予给当前用户或者由当前用户授予的表或视图上的权限。对每一个表、授予者和被授予者的组合都有一行。授予给组的权限在视图role_table_grants中标识。

表 32.33. table_privileges 的列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的用户名称
grantee	sql_identifier	权限被授予到的用户或组的名称
table_catalog	sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名称
table_name	sql_identifier	表的名称
privilege_type	character_data	权限的类型：SELECT、DELETE、INSERT、UPDATE、REFERENCES、RULE或TRIGGER
is_grantable	character_data	如果该权限是可授予的，则为YES，否则为NO
with_hierarchy	character_data	对应 PostgreSQL 不支持的特性

注意，grantee 列不区分用户和组。如果有同名的用户和组，很遗憾没有办法区分它们。PostgreSQL 的未来版本可能会禁止创建同名的用户和组。

32.36. tables

视图tables包含定义在当前数据库中的所有表和视图。
只有那些当前用户能够访问（作为拥有者或具有某些权限）的表和视图才会被显示。

表 32.34. tables 的列

名称	数据类型	描述
table_catalog	sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该表的模式名称
table_name	sql_identifier	表的名称
table_type	character_data	表的类型：持久基本表（普通表类型）为BASE TABLE，视图为VIEW，临时表为LOCAL TEMPORARY
self_referencing_column_name	sql_identifier	对应 PostgreSQL 不支持的特性

名称	数据类型	描述
reference_generation	character_data	对应 PostgreSQL 不支持的特性
user_defined_type_catalog	sql_identifier	对应 PostgreSQL 不支持的特性
user_defined_type_schema	sql_identifier	对应 PostgreSQL 不支持的特性
user_defined_type_name	sql_identifier	对应 PostgreSQL 不支持的特性

32.37. triggers

视图triggers包含当前数据库中定义的、当前用户拥有的所有触发器。（表的拥有者就是触发器的拥有者。）

表 32.35. triggers 的列

名称	数据类型	描述
trigger_catalog	sql_identifier	包含该触发器的数据库名称（总是当前数据库）
trigger_schema	sql_identifier	包含该触发器的模式名称
trigger_name	sql_identifier	触发器的名称
event_manipulation	character_data	触发触发器的事件（INSERT、UPDATE或DELETE）
event_object_catalog	sql_identifier	包含触发器所在的表的数据库名称（总是当前数据库）
event_object_schema	sql_identifier	定义触发器的表所在的模式名称
event_object_table	sql_identifier	定义触发器的表名称
action_order	cardinal_number	尚未实现
action_condition	character_data	对应 PostgreSQL 不支持的特性
action_statement	character_data	触发器执行的语句（当前总是EXECUTE PROCEDURE *function*(...)）
action_orientation	character_data	标识触发器是对每个被处理的行触发一次还是对每条语句触发一次（ROW或STATEMENT）
condition_timing	character_data	触发器触发的时间（BEFORE或AFTER）
condition_reference_old_table	sql_identifier	对应 PostgreSQL 不支持的特性
condition_reference_new_table	sql_identifier	对应 PostgreSQL 不支持的特性

PostgreSQL中的触发器有两点与 SQL 标准不兼容，这会影响在该信息模式中的表示。第一，在PostgreSQL中触发器的名字是局限于每个表的，而不是独立于模式对象。因此可能在一个模式中会有重复的触发器名称，只要它们属于不同的表（trigger_catalog和trigger_schema才真正标识了触发器被定义在哪个表上）。第二，在PostgreSQL中触发器可以被定义为在多个事件上触发（例如ON INSERT OR UPDATE），而在SQL 标准中只允许一个。如果一个触发器被定义为在多个事件上触发，它在信息模式中被表示为多行，每一行对应于一类事件。作为这两个问题的结果，视图triggers的主键实际上是(trigger_catalog, trigger_schema, trigger_name, event_object_table, event_manipulation)，而不是(trigger_catalog, trigger_schema, trigger_

name)（这是 SQL 标准指定的）。尽管如此，如果你以符合 SQL 标准（在模式中触发器名称唯一并且每个触发器只能有一种事件类型）的方式定义你的触发器，这将不会影响你。

32.38. usage_privileges

视图usage_privileges旨在标识在多种对象上授予给当前用户或由当前用户授予的 USAGE 权限。在 PostgreSQL 中，这目前只适用于域，而且由于域在 PostgreSQL 中没有真正的权限，这个视图显示的是为所有域隐式授予给 PUBLIC 的 USAGE 权限。将来，这个视图可能会包含更多有用的信息。

表 32.36. usage_privileges 的列

名称	数据类型	描述
grantor	sql_identifier	授予该权限的用户名称
grantee	sql_identifier	当前总是PUBLIC
object_catalog	sql_identifier	包含该对象的数据库名称（总是当前数据库）
object_schema	sql_identifier	包含该对象的模式名称
object_name	sql_identifier	对象的名称
object_type	character_data	当前总是 DOMAIN
privilege_type	character_data	总是USAGE
is_grantable	character_data	当前总是NO

32.39. view_column_usage

视图view_column_usage标识视图查询表达式（即定义该视图的 SELECT 语句）中使用的所有列。只有当包含该列的表归当前用户所有时，该列才会出现在这个视图中。

> **注意**
>
> 系统表列不被包括。在某个时候这应该会被修复。

表 32.37. view_column_usage 的列

名称	数据类型	描述
view_catalog	sql_identifier	包含该视图的数据库名称（总是当前数据库）
view_schema	sql_identifier	包含该视图的模式名称
view_name	sql_identifier	视图的名称
table_catalog	sql_identifier	被视图所使用的列所属表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含被视图使用的列的表所在的模式名称
table_name	sql_identifier	包含被视图使用的列的表名称
column_name	sql_identifier	被视图使用的列的名称

32.40. view_table_usage

视图view_table_usage标识视图查询表达式（即定义该视图的 SELECT 语句）中使用的所有表。只有归当前用户所有的表才会出现在这个视图中。

> **注意**
>
> 系统表没有被包括。这应当会在某个时候被修复。

表 32.38. view_table_usage 的列

名称	数据类型	描述
view_catalog	sql_identifier	包含该视图的数据库名称（总是当前数据库）
view_schema	sql_identifier	包含该视图的模式名称
view_name	sql_identifier	视图的名称
table_catalog	sql_identifier	包含被该视图所使用的表的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含被视图使用的表的模式名称
table_name	sql_identifier	被视图使用的表的名称

32.41. views

视图views包含定义在当前数据库中的所有视图。
只有当前用户能够访问（作为拥有者或具有某些权限）的视图才会被显示。

表 32.39. views 的列

名称	数据类型	描述
table_catalog	sql_identifier	包含该视图的数据库名称（总是当前数据库）
table_schema	sql_identifier	包含该视图的模式名称
table_name	sql_identifier	视图的名称
view_definition	character_data	定义该视图的查询表达式（如果当前用户不是该视图的拥有者，则为空）
check_option	character_data	对应 PostgreSQL 不支持的特性
is_updatable	character_data	尚未实现
is_insertable_into	character_data	尚未实现

部分 V. 服务器编程

本部分介绍如何使用用户定义函数、数据类型、触发器等扩展服务器功能。这些属于高级主题，
大概应当在理解了所有其他有关PostgreSQL的用户文档之后再来学习。本部分还描述 PostgreSQL发行版中可用的服务器端编程语言，以及有关服务器端编程语言的一般问题。
这些信息只对至少读过本部分前几章的读者有用。

目录

33. 扩展 SQL 444
33.1. 可扩展性如何运作 444
33.2. PostgreSQL 类型系统 444
33.2.1. 基础类型 444
33.2.2. 复合类型 444
33.2.3. 域 444
33.2.4. 伪类型 445
33.2.5. 多态类型 445
33.3. 用户定义的函数 445
33.4. 查询语言（SQL）函数 446
33.4.1. 基本类型上的 SQL 函数 446
33.4.2. 复合类型上的 SQL 函数 447
33.4.3. 用作表源的 SQL 函数 449
33.4.4. 返回集合的 SQL 函数 450
33.4.5. 多态 SQL 函数 451
33.5. 过程语言函数 452
33.6. 内部函数 452
33.7. C 语言函数 452
33.7.1. 动态载入 452
33.7.2. C 语言函数中的基础类型 453
33.7.3. C 语言函数的调用约定版本 0 456
33.7.4. C 语言函数的调用约定版本 1 458
33.7.5. 编写代码 460
33.7.6. 编译和链接动态装载的函数 461
33.7.7. C 语言函数中的复合类型参数 463
33.7.8. 从 C 语言函数中返回行（复合类型） 464
33.7.9. 从 C 语言函数中返回集合 466
33.7.10. 多态参数和返回类型 470
33.8. 函数重载 472
33.9. 用户自定义聚合 472
33.10. 用户定义的类型 474
33.11. 用户自定义操作符 477
33.12. 操作符优化信息 477
33.12.1. COMMUTATOR 478
33.12.2. NEGATOR 478
33.12.3. RESTRICT 479
33.12.4. JOIN 479
33.12.5. HASHES 480
33.12.6. MERGES (SORT1, SORT2, LTCMP, GTCMP) 480
33.13. 索引扩展接口 481
33.13.1. 索引方法和操作符类 482
33.13.2. 索引方法策略 482
33.13.3. 索引方法支持例程 483
33.13.4. 示例 484
33.13.5. 系统对操作符类的依赖 486
33.13.6. 操作符类的特殊特性 487
34. 规则系统 488
34.1. 查询树 488
34.2. 视图和规则系统 489
34.2.1. SELECT 规则如何工作 490
34.2.2. 非 SELECT 语句中的视图规则 494

34.2.3. PostgreSQL 中视图的能力 495
34.2.4. 更新视图 496
34.3. INSERT、UPDATE 和 DELETE 上的规则 496
34.3.1. 更新规则如何工作 496
34.3.2. 与视图的协作 500
34.4. 规则和权限 505
34.5. 规则和命令状态 506
34.6. 规则与触发器 507
35. 触发器 510
35.1. 触发器行为概述 510
35.2. 数据更改的可见性 510
35.3. 用 C 编写触发器函数 511
35.4. 一个完整的示例 513
36. 过程语言 517
36.1. 安装过程语言 517
37. PL/pgSQL - SQL 过程语言 519
37.1. 概述 519
37.1.1. 使用 PL/pgSQL 的优点 520
37.1.2. 支持的参数和结果数据类型 520
37.2. PL/pgSQL 开发技巧 520
37.2.1. 引号的处理 521
37.3. PL/pgSQL 的结构 522
37.4. 声明 523
37.4.1. 函数参数的别名 523
37.4.2. 复制类型 524
37.4.3. 行类型 525
37.4.4. 记录类型 525
37.4.5. RENAME 525
37.5. 表达式 526
37.6. 基本语句 527
37.6.1. 赋值 527
37.6.2. SELECT INTO 527
37.6.3. 执行无结果的表达式或查询 528
37.6.4. 执行动态命令 528
37.6.5. 获取结果状态 530
37.7. 控制结构 530
37.7.1. 从函数返回 530
37.7.2. 条件 531
37.7.3. 简单循环 533
37.7.4. 遍历查询结果 535
37.8. 游标 536
37.8.1. 声明游标变量 536
37.8.2. 打开游标 536
37.8.3. 使用游标 537
37.9. 错误和消息 539
37.10. 触发器过程 539
37.11. 从 Oracle PL/SQL 移植 541
37.11.1. 移植示例 542
37.11.2. 其他需要注意的事项 547
37.11.3. 附录 547
38. PL/Tcl - Tcl 过程语言 551
38.1. 概述 551
38.2. PL/Tcl 函数和参数 551

38.3. PL/Tcl 中的数据值 552
38.4. PL/Tcl 中的全局数据 552
38.5. 从 PL/Tcl 访问数据库 553
38.6. PL/Tcl 中的触发器过程 555
38.7. 模块和 unknown 命令 556
38.8. Tcl 过程名 557
39. PL/Perl - Perl 过程语言 558
39.1. PL/Perl 函数和参数 558
39.2. PL/Perl 中的数据值 559
39.3. 从 PL/Perl 访问数据库 559
39.4. 受信任与不受信任的 PL/Perl 560
39.5. 缺失的特性 560
40. PL/Python - Python 过程语言 562
40.1. PL/Python 函数 562
40.2. 触发器函数 563
40.3. 数据库访问 563
41. 服务器编程接口 565
41.1. 接口函数 565
41.2. 接口支持函数 577
41.3. 内存管理 584
41.4. 数据更改的可见性 594
41.5. 示例 594

第 33 章 扩展 SQL

在接下来的各节中，我们将讨论如何通过增加以下内容来扩展 PostgreSQL SQL 查询语言：

- 函数（从第 33.3 节开始）
- 聚合（从第 33.9 节开始）
- 数据类型（从第 33.10 节开始）
- 操作符（从第 33.11 节开始）
- 用于索引的操作符类（从第 33.13 节开始）

33.1. 可扩展性如何运作

PostgreSQL 之所以具有可扩展性，是因为其运作由系统目录驱动。如果你熟悉标准的关系数据库系统，就会知道它们把有关数据库、表、
列等的信息存储在通常所说的系统目录中（有些系统把这称为数据字典）。
这些目录对用户而言看起来就像普通表一样，但 DBMS 会在其中保存自己的内部管理信息。
PostgreSQL 与标准关系数据库系统的一个关键差别是，PostgreSQL 在目录中存储的信息要多得多：不仅有关于表和列的信息，还有关于数据类型、函数、访问方法等的信息。
这些表可以由用户修改，而 PostgreSQL 又是基于这些表来运行的，这意味着 PostgreSQL 可以由用户扩展。相比之下，传统数据库系统只能通过修改源代码中的硬编码过程，或加载由 DBMS 供应商专门编写的模块来扩展。

此外，PostgreSQL 服务器还能通过动态加载把用户编写的代码纳入自身。也就是说，用户可以指定一个实现了新类型或新函数的目标代码文件（例如共享库），而 PostgreSQL 会在需要时加载它。把用 SQL 编写的代码加入服务器就更为简单了。这种能够“即时”修改自身行为的能力，使 PostgreSQL 特别适合用于新应用和新存储结构的快速原型设计。

33.2. PostgreSQL 类型系统

PostgreSQL 数据类型可以分为基础类型、复合类型、域和伪类型。

33.2.1. 基础类型

基本类型是像 `int4` 这样在 SQL 语言层级之下实现（通常用 C 之类的低级语言）的类型。它们大体对应于通常所说的抽象数据类型。PostgreSQL 只能通过用户提供的函数操作这类类型，并且只在用户描述它们的程度上理解这类类型的行为。基本类型进一步细分为标量类型和数组类型。
对每个标量类型，都会自动创建一个对应的数组类型，
它可以保存该标量类型的大小可变的数组。

33.2.2. 复合类型

用户每次创建表时，都会创建复合类型（也称行类型）。还可以定义不关联任何表的“独立”
复合类型。复合类型就是带有对应字段名的基类型列表。
复合类型的值是由字段值组成的一行或一条记录。用户可以在 SQL 查询中访问各个组成字段。

33.2.3. 域

域基于某一种特定的基础类型，并且在很多场合下都可以与其基础类型互换。不过，
域可以带有约束，把其合法值限制在底层基础类型所允许值的一个子集内。

域可以用 SQL 命令 `CREATE DOMAIN` 创建。它们的创建和使用不在本章讨论之列。

33.2.4. 伪类型

有少数几种“伪类型”用于特殊目的。伪类型不能作为表的列或复合类型的属性出现，但它们可以用来声明函数的参数类型和结果类型。这为在类型系统内部识别特殊类别的函数提供了一种机制。表 8.20列出了现有的伪类型。

33.2.5. 多态类型

两种特别值得关注的伪类型是 `anyelement` 和 `anyarray`，它们统称为多态类型。使用这些类型声明的函数称为多态函数。多态函数可以操作多种不同的数据类型，具体类型由某次调用时实际传入的数据类型决定。

多态参数和结果相互关联，在解析调用多态函数的查询时，会被解析为某种具体的数据类型。每个声明为 `anyelement` 的位置（参数或返回值）都可以具有任意具体的实际数据类型，但在同一次调用中，它们必须具有相同的实际类型。每个声明为 `anyarray` 的位置可以具有任意数组数据类型，但同样必须全部具有相同的类型。此外，如果某些位置声明为 `anyarray`，另一些位置声明为 `anyelement`，那么 `anyarray` 位置上的实际数组类型，其元素类型必须与 `anyelement` 位置上的类型相同。

因此，当多于一个参数位置声明为多态类型时，净效果是只允许实际参数类型的某些组合。例如，声明为 `foo(anyelement, anyelement)` 的函数接受任意两个输入值，只要它们具有相同的数据类型。

如果函数的返回值声明为多态类型，那么至少有一个参数位置也必须是多态的，并且实际传入的参数数据类型决定该次调用的实际结果类型。例如，如果尚不存在数组下标机制，可以将实现下标访问的函数定义为 `subscript(anyarray, integer) returns anyelement`。这一声明将实际的第一个参数约束为数组类型，并允许解析器根据第一个参数的实际类型推导正确的结果类型。

33.3. 用户定义的函数

PostgreSQL提供四种函数：

- 查询语言函数（用SQL编写的函数）（第 33.4 节）
- 过程语言函数（例如用 PL/Tcl 或 PL/pgSQL 编写的函数）（第 33.5 节）
- 内部函数（第 33.6 节）
- C 语言函数（第 33.7 节）

每种函数都可以接受基本类型、复合类型或它们的组合作为参数。此外，每种函数都可以返回基本类型或复合类型。

很多类函数可以接受或者返回特定的伪类型（例如，多态类型），但是可用的功能因函数类别而异。详情可以参考每一种函数的描述。

定义SQL函数最容易，因此我们将从讨论它们开始。大部分SQL函数的概念也能用到其他类型的函数上。

在本章各处，查阅 `CREATE FUNCTION` 命令的参考页有助于更好地理解示例。本章的一些示例可以在 PostgreSQL 源代码发行版的 `src/tutorial` 目录中的 `funcs.sql` 和 `funcs.c` 中找到。

33.4. 查询语言（SQL）函数

SQL 函数执行由任意 SQL 语句组成的一个列表，并返回列表中最后一个查询的结果。在简单（非集合）情况下，将返回最后一个查询结果的第一行。（请记住，除非使用 `ORDER BY`，否则多行结果中的“第一行”并没有明确定义。）如果最后一个查询完全没有返回任何行，则返回空值。

或者，一个 SQL 函数可以通过指定函数的返回类型为`SETOF` *`sometype`*被声明为返回一个集合。在这种情况下，最后一个查询的结果的所有行会被返回。下文将给出进一步的细节。

SQL 函数体应是一个或多个用分号分隔的 SQL 语句的列表。注意，由于 `CREATE FUNCTION` 命令的语法要求函数体用单引号括起，因此函数体中使用的单引号（`'`）必须转义：在需要引号的地方写两个单引号（`''`）或一个反斜杠（`\'`）。

SQL 函数的参数可以在函数体中用语法 `$n` 引用：`$1` 指第一个参数，`$2` 指第二个参数，依此类推。如果参数是复合类型，则可以使用点记法（例如 `$1.name`）访问参数的属性。

33.4.1. 基本类型上的 SQL 函数

最简单的 SQL 函数没有参数，只是返回一个基本类型，例如 `integer`：

```
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

 one
-----
   1
```

注意我们为该函数的结果在函数体内定义了一个列别名（名为`result`），
但是这个列别名在函数以外是不可见的。因此，结果被标记为`one`而不是`result`。

定义以基本类型为参数的 SQL 函数几乎同样容易。在下面的示例中，注意我们如何在函数内把参数引用为 `$1` 和 `$2`。

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS '
    SELECT $1 + $2;
' LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

下面是一个更有用的函数，它可以用来给银行账户扣款：

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS integer AS '
    UPDATE bank
        SET balance = balance - $2
        WHERE accountno = $1;
    SELECT 1;
```

```
' LANGUAGE SQL;
```

用户可以执行如下命令给账户 17 扣款 $100.00：

```
SELECT tf1(17, 100.0);
```

实践中人们可能希望函数返回比常量更有用的结果，所以更可能的定义是

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS '
    UPDATE bank
        SET balance = balance - $2
        WHERE accountno = $1;
    SELECT balance FROM bank WHERE accountno = $1;
' LANGUAGE SQL;
```

它调整余额并返回新余额。

SQL 语言中任何命令的集合都可以打包到一起并定义为函数。除 `SELECT` 查询外，这些命令还可以包括数据修改（即 `INSERT`、`UPDATE` 和 `DELETE`）。但最后一条命令必须是一个 `SELECT`，它返回函数返回类型所指定的内容。或者，如果你想定义一个执行操作但没有有用值可返回的 SQL 函数，可以把它定义为返回 `void`。此时函数体不能以 `SELECT` 结尾。例如：

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary <= 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------

(1 row)
```

33.4.2. 复合类型上的 SQL 函数

指定带复合类型参数的函数时，我们不仅要指明想要哪个参数（如上面用 `$1` 和 `$2`），还要指明该参数的属性。例如，设 `emp` 是一个包含员工数据的表，因而也是该表每行的复合类型的名称。下面的函数 `double_salary` 计算某人薪资翻倍后的值：

```
CREATE TABLE emp (
    name        text,
    salary      integer,
    age         integer,
    cubicle     point
);

CREATE FUNCTION double_salary(emp) RETURNS integer AS '
    SELECT $1.salary * 2 AS salary;
' LANGUAGE SQL;

SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

```
 name | dream
------+-------
 Sam  |  2400
```

注意使用语法 `$1.salary` 选择参数行值的一个字段。还要注意调用它的 `SELECT` 命令如何用表名把该表的整个当前行表示为一个复合值。表行也可以这样引用：

```
SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

这强调了它的行本质。

也可以构建返回复合类型的函数。下面是一个返回单个 `emp` 行的函数示例：

```
CREATE FUNCTION new_emp() RETURNS emp AS '
    SELECT text ''None'' AS name,
        1000 AS salary,
        25 AS age,
        point ''(2,2)'' AS cubicle;
' LANGUAGE SQL;
```

在本例中，我们为每个属性都指定了常量值，但这些常量可以换成任何计算。

定义该函数时有两点重要注意事项：

- 查询中的选择列表顺序必须与列在该复合类型所关联的表中出现的顺序完全相同。（系统不考虑像上面这样指定的列名。）
- 你必须对表达式进行类型转换，使之与复合类型的定义匹配，否则会得到这样的错误：

  ```
  ERROR:  function declared to return emp returns varchar instead of
   text at column 1
  ```

返回行（复合类型）的函数可以用作表函数，如下文所述。它也可以在 SQL 表达式的上下文中调用，但仅当从行中提取单个属性，或把整行传递给另一个接受相同复合类型的函数时。

这是从行类型中提取一个属性的示例：

```
SELECT (new_emp()).name;

 name
------
 None
```

我们需要额外的圆括号来避免解析器混淆：

```
SELECT new_emp().name;
ERROR:  syntax error at or near "." at character 17
```

另一种选择是使用函数记法提取属性。简单的解释是：`attribute(table)` 和 `table.attribute` 这两种记法可以互换使用。

```
SELECT name(new_emp());
```

```
 name
------
 None

-- This is the same as:
-- SELECT emp.name AS youngster FROM emp WHERE emp.age < 30

SELECT name(emp) AS youngster
    FROM emp
    WHERE age(emp) < 30;

 youngster
-----------
 Sam
```

使用返回行的函数结果的另一种方式是声明一个接受行类型参数的第二个函数，
并把第一个函数的结果传给它：

```
CREATE FUNCTION getname(emp) RETURNS text AS '
    SELECT $1.name;
' LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

33.4.3. 用作表源的 SQL 函数

所有的 SQL 函数都可以被用在查询的`FROM`子句中，但是对于返回复合类型的函数特别有用。如果函数被定义为返回一种基础类型，该表函数会产生一个单列表。
如果该函数被定义为返回一种复合类型，该表函数会为该复合类型的每一个属性产生一列。

下面是一个例子：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS '
    SELECT * FROM foo WHERE fooid = $1;
' LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(2 rows)
```

如示例所示，我们可以像使用普通表的列一样使用函数结果的各列。

注意我们只从函数得到了一行。这是因为我们没有使用 SETOF。这将在下一节描述。

33.4.4. 返回集合的 SQL 函数

当一个 SQL 函数被声明为返回SETOF *sometype*时，该函数的最后一个 SELECT 查询会被执行完，并且它输出的每一行都会被作为结果集的一个元素返回。

此特性通常用于在 FROM 子句中调用函数。此时函数返回的每一行都成为查询所见表的一行。例如，设表 foo 的内容同上，我们执行：

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS '
    SELECT * FROM foo WHERE fooid = $1;
' LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

那么我们会得到：

```
 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

目前，返回集合的函数也可以在查询的选择列表中调用。对查询自身生成的每一行，都会调用一次返回集合的函数，并为函数结果集的每个元素生成一个输出行。但要注意，此能力已被弃用，在未来的版本中可能被移除。下面是一个从选择列表返回集合的函数示例：

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS
'SELECT name FROM nodes WHERE parent = $1'
LANGUAGE SQL;

SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name  | listchildren
--------+--------------
```

```
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

在最后一个 SELECT 中，注意 Child2、Child3 等没有输出行。这是因为 listchildren 对这些参数返回空集，因此不会生成结果行。

33.4.5. 多态 SQL 函数

SQL 函数可以声明为接受和返回多态类型 anyelement 和 anyarray。关于多态函数更详细的解释见第 33.2.5 节。下面是一个多态函数 make_array，它由两个任意数据类型的元素构建一个数组：

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS
 '
    SELECT ARRAY[$1, $2];
' LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS
 textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

注意类型转换 'a'::text 的使用是为了指定该参数的类型是 text。如果该参数只是一个字符串字面量，就必须这样做，否则它会被当作 unknown 类型，并且 unknown 的数组也不是一种合法的类型。如果没有该类型转换，将得到这样的错误：

```
ERROR:  could not determine "anyarray"/"anyelement" type because input
 has type "unknown"
```

允许有带确定性返回类型的多态参数，但反过来不行。例如：

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS
 '
    SELECT $1 > $2;
' LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS '
    SELECT 1;
' LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A function returning "anyarray" or "anyelement" must have at
 least one argument of either type.
```

33.5. 过程语言函数

过程语言并不内建于 PostgreSQL 服务器；它们由可装载模块提供。关于各语言的语法以及函数体如何被解释的细节，请参阅相应过程语言的文档。

目前标准的 PostgreSQL 发行版中有四种过程语言可用：PL/pgSQL、PL/Tcl、PL/Perl 和 PL/Python。更多信息请参见第 36 章。其他语言可以由用户定义。开发新过程语言的基础见第 47 章。

33.6. 内部函数

内部函数由 C 编写并且已经被静态链接到 PostgreSQL 服务器中。该函数定义的“主体”指定该函数的 C 语言名称，它不必与为 SQL 使用而声明的名称相同。（出于向后兼容的原因，也接受空主体，此时会认为 C 语言函数名与 SQL 函数名相同。）

通常，所有存在于服务器中的内部函数都在数据库集簇的初始化（`initdb`）期间被声明，但是用户可以使用 `CREATE FUNCTION`为一个内部函数创建额外的别名。在`CREATE FUNCTION`中用语言名`internal`来声明内部函数。例如，要为 `sqrt`函数创建一个别名：

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

（大部分内部函数应该被声明为“严格”）。

> **注意**
>
> 上述意义上，并非所有“预定义”的函数都是“内部”函数。有些预定义的函数由 SQL 编写。

33.7. C 语言函数

用户自定义函数可以用 C（或可与 C 兼容的语言，如 C++）编写。这样的函数被编译成动态可装载对象（也称共享库），由服务器按需装载。动态装载特性正是“C 语言”函数与“internal”函数的区别所在——两者的实际编码约定基本相同。（因此，标准的内部函数库是用户自定义 C 函数编码示例的丰富来源。）

目前 C 函数使用两种不同的调用约定。较新的“版本 1”调用约定通过为函数编写一个`PG_FUNCTION_INFO_V1()`宏来指示，如下文所示。没有这样的宏则表示这是一个旧式（“版本 0”）函数。无论哪种情况，`CREATE FUNCTION`中指定的语言名称都是`C`。旧式函数由于可移植性问题和功能缺乏，现在已被弃用，但出于兼容性原因仍然支持。

33.7.1. 动态载入

在一个会话中第一次调用一个特定可载入目标文件中的用户定义函数时，动态载入器会把那个目标文件载入到内存以便该函数被调用。因此用户定义的 C 函数的`CREATE FUNCTION`必须为该函数指定两块信息：可载入目标文件的名称，以及要在该目标文件

中调用的特定函数的 C 名称（链接符号）。如果没有显式指定 C 名称，则它被假定为和 SQL 函数名相同。

下面的算法被用来基于CREATE FUNCTION 命令中给定的名称来定位共享目标文件：

1. 如果名称是一个绝对路径，则载入给定的文件。

2. 如果该名称以字符串$libdir开始，那么这一部分会被 PostgreSQL包的库目录名（在编译时确定）替换。

3. 如果名称不含目录部分，则在配置变量 dynamic_library_path 指定的路径中搜索该文件。

4. 否则（在该路径中没找到该文件，或者它包含一个非绝对目录），
   动态载入器将尝试接受给定的名称，这很可能会失败（依赖当前工作目录是不可靠的）。

如果这一序列不成功，就会把平台特定的共享库文件名扩展（通常是 .so）追加到给定名称后并再试这一序列。如果仍然失败，装载就会失败。

用于运行PostgreSQL服务器的用户 ID 必须能够通过要载入文件的路径。常见的错误是把文件或更高层的目录变得对postgres用户不可读或者不可执行。

在任何情况下，CREATE FUNCTION命令中给定的文件名会被原封不动地记录在系统目录中，这样如果需要再次载入该文件则会应用同样的过程。

> **注意**
>
> PostgreSQL不会自动编译 C 函数。在从CREATE FUNCTION命令中引用目标文件之前，它必须先被编译好。更多信息请见第 33.7.6 节。

在第一次使用之后，动态载入目标文件会保留在内存中。在同一个会话中，
后续对该文件中函数的调用只需付出一次很小的符号表查找开销。
如果需要强制重新载入一个目标文件（例如在重新编译之后），可以使用 LOAD命令或者开启一个新的会话。

建议通过相对于 $libdir 的路径，或者通过动态库路径来定位共享库。这样一来，如果新安装位于不同的位置，版本升级会更简单。$libdir 实际代表的目录可以通过命令 pg_config --pkglibdir 查出。

在PostgreSQL 7.2 版本之前，在 CREATE FUNCTION中只能指定目标文件的精确绝对路径。这种做法现在已被弃用，因为它会使函数定义变得不必要地不可移植。
最好只指定不带路径和扩展名的共享库名，而让搜索机制来提供这些信息。

33.7.2. C 语言函数中的基础类型

要了解如何编写 C 语言函数，你需要了解 PostgreSQL如何在内部表达基础类型以及如何与函数传递它们。在内部，PostgreSQL把基础类型视为一块“内存数据块”。
你为该类型定义的用户自定义函数，决定了 PostgreSQL 如何操作它。也就是说，PostgreSQL 只负责把数据存到磁盘、再从磁盘取回，而数据的输入、处理和输出则依赖你定义的这些函数。

基础类型可以有三种内部格式之一：

- 传值，定长
- 传引用，定长
- 传引用，变长

传值类型在长度上只能是 1、2 或 4 字节（如果你的机器上 sizeof(Datum) 是 8，则还有 8 字节）。你应当小心地定义你的类型，使其在所有架构上都具有相同的尺寸（字节）。例如，long 类型很危险，因为它在某些机器上是 4 字节但在另一些机器上是 8 字节；而 int 类型在大多数 Unix 机器上都是 4 字节。在 Unix 机器上 int4 类型的一种合理实现可以是：

```
/* 4-byte integer, passed by value */
typedef int int4;
```

另一方面，任意尺寸的定长类型都可以通过引用传递。例如，这里有一种 PostgreSQL类型的实现示例：

```
/* 16 字节结构，传引用 */
typedef struct
{
    double  x, y;
} Point;
```

在PostgreSQL函数中传进或传出这种类型时，只能使用指向这种类型的指针。
要返回这样一种类型的值，用 palloc分配正确的内存量，然后填充分配好的内存，并且返回一个指向该内存的指针。
（你也可以通过返回指向某个输入值的指针来直接返回一个与返回值类型相同的输入值。但绝不要修改一个按引用传递的输入值的内容。）

最后，所有变长类型也必须按引用传递。所有变长类型都必须以一个正好 4 字节的长度字段开始，而要存储在该类型中的所有数据都必须位于紧随该长度字段之后的内存中。
长度字段包含该结构的总长度，也就是说，它包含长度字段本身的尺寸。

作为一个例子，我们可以把类型 text 定义为如下形式：

```
typedef struct {
    int4 length;
    char data[1];
} text;
```

显然，这里声明的 data 字段不足以容纳所有可能的字符串。由于在 C 中无法声明变长结构体，我们依赖于这样的知识：C 编译器不会对数组下标做范围检查。我们只需分配必要数量的空间，然后就像数组声明为正确长度那样去访问它。（这是一个常见的技巧，你可以在许多关于 C 的教科书中读到。）

在操作变长类型时，我们必须小心分配正确数量的内存，并正确设置长度字段。例如，
如果我们想在一个 text 结构体中存储 40 字节，我们可以使用这样的代码片段：

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
destination->length = VARHDRSZ + 40;
```

```
memcpy(destination->data, buffer, 40);
...
```

VARHDRSZ 和 sizeof(int4) 一样，但是用宏 VARHDRSZ 来引用变长类型的额外开销的尺寸被认为是比较好的风格。

表 33.1说明了在编写使用 PostgreSQL 内置类型的 C 语言函数时，哪种 C 类型对应哪种 SQL 类型。“定义文件”列给出了需要包含的头文件，以获取类型定义。（实际的定义可能位于所列文件包含的其他文件中。建议用户坚持使用已定义的接口。）请注意，
在任何源文件中都应该始终首先包含postgres.h，因为它声明了很多你反正都会用到的内容。

表 33.1. 内置 SQL 类型的等价 C 类型

SQL 类型	C 类型	定义文件
abstime	AbsoluteTime	utils/nabstime.h
boolean	bool	postgres.h（可能是编译器内置）
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	（编译器内置）
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int2 或 int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int4 或 int32	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h
interval	Interval*	utils/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp*	utils/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

了解了基础类型所有可能的结构后，就可以看一些实际函数示例。

33.7.3. C 语言函数的调用约定版本 0

我们先介绍“旧风格”调用约定——虽然此方法现已弃用，但初学者更容易上手。在版本 0 方法中，C 函数的参数和结果就按普通 C 风格声明，只是要小心使用上所示的每种 SQL 数据类型的 C 表示。

下面是一些例子：

```
#include "postgres.h"
#include <string.h>

/* by value */

int
add_one(int arg)
{
    return arg + 1;
}

/* by reference, fixed length */

float8 *
add_one_float8(float8 *arg)
{
    float8    *result = (float8 *) palloc(sizeof(float8));

    *result = *arg + 1.0;

    return result;
}

Point *
makepoint(Point *pointx, Point *pointy)
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* by reference, variable length */

text *
copytext(text *t)
{
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(new_t) = VARSIZE(t);
```

```
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t)-VARHDRSZ);    /* how many bytes */
    return new_t;
}

text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    VARATT_SIZEP(new_text) = new_text_size;
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    return new_text;
}
```

假定上述代码已经写在文件`funcs.c`中并编译为共享对象，我们可以用类似下面的命令向PostgreSQL定义这些函数：

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'DIRECTORY/funcs', 'add_one'
     LANGUAGE C STRICT;

-- note overloading of SQL function name "add_one"
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'DIRECTORY/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'DIRECTORY/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'DIRECTORY/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'DIRECTORY/funcs', 'concat_text',
     LANGUAGE C STRICT;
```

这里*DIRECTORY*代表共享库文件所在的目录（例如 PostgreSQL的教程目录，其中包含本节示例所用的代码）。（更好的风格是在`AS`子句中只写`'funcs'`，前提是已把 *DIRECTORY*加入搜索路径。无论哪种情况，都可以省略共享库的系统特定扩展名，通常是 `.so`或`.sl`。）

注意我们把函数声明为“strict”（严格），意思是如果任何输入值为空，系统会自动假定结果为空。这样做可以避免在函数代码中检查空输入。否则，我们就必须显式检查空值，即检查每个传引用参数是否为空指针。（对于传值参数，我们甚至没有办法检查！）

尽管这种调用约定使用简单，但可移植性不好；在某些体系结构上，以这种方式传递小于`int`的数据类型会有问题。另外，也没有返回空结果的简单方法，除了把函数声明为严格函数之外，也没有其他处理空参数的简单方法。下面介绍的版本 1 约定克服了这些缺点。

33.7.4. C 语言函数的调用约定版本 1

版本-1 的调用约定依赖于宏来屏蔽传递参数和结果的大部分复杂性。版本-1 函数的 C 声明总是：

```
Datum funcname(PG_FUNCTION_ARGS)
```

此外，宏调用：

```
PG_FUNCTION_INFO_V1(funcname);
```

必须出现在同一个源文件中（按惯例会正好写在该函数本身之前）。这种宏调用不是`internal`语言函数所需要的，因为 PostgreSQL会假定所有内部函数都使用版本-1 调用约定。不过，对于动态载入函数是必需的。

在版本-1 函数中，每一个实参都使用对应于该参数数据类型的`PG_GETARG_`*`xxx`*`()`宏取得，结果要用对应于返回类型的`PG_RETURN_`*`xxx`*`()`宏返回。`PG_GETARG_`*`xxx`*`()`的参数是要取得的函数参数的编号，从零开始计。`PG_RETURN_`*`xxx`*`()`的参数是实际要返回的值。

下面我们展示与上面相同的函数，以版本 1 风格编码：

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"

/* by value */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* b reference, fixed length */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* The macros for FLOAT8 hide its pass-by-reference nature. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}
```

```
PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Here, the pass-by-reference nature of Point is not hidden. */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* by reference, variable length */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text     *new_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(new_t) = VARSIZE(t);
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t)-VARHDRSZ);    /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_P(0);
    text  *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    VARATT_SIZEP(new_text) = new_text_size;
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

这些函数的CREATE FUNCTION命令与版本 0 的等价形式相同。

乍看之下，版本 1 的编码约定似乎只是无谓的故弄玄虚。不过，它们确实提供了许多改进，因为宏可以隐藏不必要的细节。例如，在编码add_one_float8时，我们不再需要知道float8是传引用类型。另一个例子是，变长类型的GETARG宏允许更有效地取得“toasted”（压缩或外置）值。

版本-1 函数的一大改进是对空输入和结果的更好处理。宏PG_ARGISNULL(*n*)允许一个函数测试是否每一个输入为空值（当然，只需要在没有声明为“strict”的函数中这样做）。和PG_GETARG_*xxx*()宏一样，输入参数也是从零开始计数。
注意应该在验证了一个参数不是空值之后才执行PG_GETARG_*xxx*()。要返回空值结果，应执行PG_RETURN_NULL()，它对严格的以及非严格的函数都有用。

新式接口中提供的其他选项是PG_GETARG_*xxx*()宏的两个变种。其中的第一种是PG_GETARG_*xxx*_COPY()，
它确保返回的指定参数的拷贝可以被安全地写入（通常的宏有时会返回一个指向表中物理存储值的指针，不能写入该值。使用PG_GETARG_*xxx*_COPY()宏可以保证得到一个可写的结果）。
第二种变种PG_GETARG_*xxx*_SLICE()宏有三个参数。第一个是函数参数的编号（如上文）。
第二个和第三个是要被返回的段的偏移量和长度。偏移量从零开始计算，
而负值的长度则表示要求返回该值的剩余部分。当大型值的存储类型为“external”时，
这些宏提供了访问这些大型值的部分内容的更有效方法（列的存储类型可以使用ALTER TABLE *tablename* ALTER COLUMN *colname* SET STORAGE *storagetype*来指定。*storagetype*取plain、external、extended或者main）。

最后，版本-1 的函数调用约定可以返回集合结果（第 33.7.9 节）、实现触发器函数（第 35 章）和过程语言调用处理器（第 47 章）。版本-1 代码也比版本-0 更可移植，因为它不违反 C 标准中关于函数调用协议的限制。更多细节可见源代码发布中的src/backend/utils/fmgr/README。

33.7.5. 编写代码

在开始更高级的话题之前，我们应该讨论一下用于 PostgreSQL C 语言函数的编码规则。虽然有可能把不是 C 编写的函数载入到 PostgreSQL中，但即使能够做到，通常也很困难，因为其他语言（例如 C++、FORTRAN 或者 Pascal）通常不会遵循和 C 相同的调用约定。也就是说，其他语言不会以同样的方式在函数之间传递参数以及返回值。由于这个原因，我们会假定你的 C 语言函数确实是用 C 编写的。

编写和构建 C 语言函数的基本规则如下：

- 使用 pg_config --includedir-server 查出 PostgreSQL 服务器头文件在你的系统上（或你的用户将要运行的系统上）安装于何处。这个选项是 PostgreSQL 7.2 新增的。对于 PostgreSQL 7.1，你应当使用选项 --includedir。（pg_config 遇到未知选项时会以非零状态退出。）对于 7.1 之前的版本，你只能去猜，但由于那是在现行调用约定引入之前，你不太可能还想要支持那些版本。

- 为了让你的代码能够被 PostgreSQL 动态装入，编译和链接时总是需要特殊的选项。关于如何在特定操作系统上完成这件事，详见第 33.7.6 节。

- 分配内存时，使用 PostgreSQL 提供的 palloc 和 pfree，而不是相应的 C 库函数 malloc 和 free。用 palloc 分配的内存会在每个事务结束时自动释放，从而避免内存泄漏。

- 总是使用 memset 将结构体的所有字节清零。如果不这么做，就很难支持哈希索引或哈希连接，因为那时你必须只挑出数据结构中真正有意义的位来计算哈希值。
  即使你初始化了结构体的所有字段，
  结构体中仍可能存在包含垃圾值的对齐填充字节（也就是结构体中的空洞）。

- PostgreSQL 的大多数内部类型都在 postgres.h 中声明，而函数管理器接口（PG_FUNCTION_ARGS 等）位于 fmgr.h 中，因此至少需要包含这两个文件。出于可移植性考虑，最好把 postgres.h 放在最前面，先于任何其他系统或用户头文件。包含 postgres.h 时，也会顺带为你包含 elog.h 和 palloc.h。
- 目标文件中定义的符号名不能彼此冲突，也不能与 PostgreSQL 服务器可执行文件中定义的符号冲突。如果你收到这类错误消息，就必须重命名相关函数或变量。

33.7.6. 编译和链接动态装载的函数

在你能够使用以 C 编写的 PostgreSQL 扩展函数之前，必须以特殊方式对它们进行编译和链接，以生成一个可由服务器动态装载的文件。更准确地说，需要创建一个共享库。

若想了解本节未涵盖的信息，你应阅读操作系统的文档，特别是 C 编译器 cc 和链接编辑器 ld 的手册页。此外，PostgreSQL 源代码在 contrib 目录中包含若干可用的示例。不过，如果你依赖这些示例，就会使你的模块依赖于 PostgreSQL 源代码是否可用。

创建共享库通常与链接可执行文件类似：先把源文件编译为目标文件，
再把目标文件链接在一起。目标文件需要以位置无关代码（PIC），形式生成。从概念上讲，
这意味着当它们被可执行文件装载时，可以放在内存中的任意位置。
（面向可执行文件的目标文件通常不会这样编译。）链接共享库的命令中也包含一些特殊标志，
用来把它与链接可执行文件的命令区分开来（至少理论上如此，
某些系统上的实际做法要丑陋得多）。

在下面的示例中，我们假定你的源代码位于文件 foo.c 中，并将创建共享库 foo.so。除非另有说明，中间目标文件名为 foo.o。共享库可以包含多个目标文件，但这里我们只使用一个。

BSD/OS

生成 PIC 的编译器选项是 -fpic。创建共享库时使用的链接器选项是 -shared。

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

这从 BSD/OS 4.0 版起适用。

FreeBSD

生成 PIC 的编译器选项是 -fpic。创建共享库时使用的编译器选项是 -shared。

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

这从 FreeBSD 3.0 版起适用。

HP-UX

生成 PIC 的系统编译器选项是 +z。使用 GCC 时则是 -fpic。用于共享库的链接器选项是 -b。因此：

```
cc +z -c foo.c
```

or:

```
gcc -fpic -c foo.c
```

and then:

```
ld -b -o foo.sl foo.o
```

与大多数其他系统不同，HP-UX 使用 `.sl` 作为共享库扩展名。

IRIX

PIC 是默认行为，无须特殊的编译器选项。生成共享库时使用的链接器选项是 `-shared`。

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

Linux

生成 PIC 的编译器选项是 `-fpic`。在某些平台的某些情况下，如果 `-fpic` 不起作用，则必须使用 `-fPIC`。更多信息请参阅 GCC 手册。创建共享库的编译器选项是 `-shared`。完整示例如下：

```
cc -fpic -c foo.c
cc -shared -o foo.so foo.o
```

MacOS X

下面是一个示例。它假定开发者工具已经安装。

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

生成 PIC 的编译器选项是 `-fpic`。对于 ELF 系统，使用带 `-shared` 选项的编译器来链接共享库。在较旧的非 ELF 系统上，则使用 `ld -Bshareable`。

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

生成 PIC 的编译器选项是 `-fpic`。链接共享库使用 `ld -Bshareable`。

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

使用 Sun 编译器时，生成 PIC 的编译器选项是 `-KPIC`；使用 GCC 时则是 `-fpic`。要链接共享库，两种编译器都使用编译器选项 `-G`，或者在使用 GCC 时改用 `-shared`。

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

Tru64 UNIX

PIC 是默认值，因此编译命令就是常规的那条。链接时需要使用带特殊选项的`ld`。

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

使用 GCC 代替系统编译器时过程相同；不需要特殊选项。

UnixWare

生成 PIC 的编译器选项，对于 SCO 编译器是 `-K PIC`，而 `-fpic` 则用于 GCC。链接共享库时，编译器选项对于 SCO 编译器是 `-G`，对于 GCC 则是 `-shared`。

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

或者

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

提示

如果这些内容对你来说过于复杂，可以考虑使用 GNU Libtool[1]，它通过统一接口隐藏了平台差异。

生成的共享库文件随后就可以装载到 PostgreSQL 中。在向 `CREATE FUNCTION` 命令指定文件名时，必须给出共享库文件名，而不是中间目标文件名。请注意，系统标准的共享库扩展名（通常是 `.so` 或 `.sl`）可以在 `CREATE FUNCTION` 命令中省略，并且通常也应省略，以获得最佳可移植性。

关于服务器期望在何处找到共享库文件，请回头参见第 33.7.1 节。

33.7.7. C 语言函数中的复合类型参数

复合类型没有像 C 结构体那样的固定布局。复合类型的实例可能包含空值字段。此外，继承层次中的复合类型可能具有和同一继承层次中其他成员不同的字段。因此，PostgreSQL提供了函数接口以便从 C 访问复合类型的字段。

假设我们想编写一个函数来回答查询

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
    FROM emp
    WHERE name = 'Bill' OR name = 'Sam';
```

使用版本 0 调用约定，我们可以把 `c_overpaid` 定义为：

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

bool
c_overpaid(TupleTableSlot *t, /* the current row of emp */
           int32 limit)
{
    bool isnull;
```

[1] http://www.gnu.org/software/libtool/

```
    int32 salary;

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
    if (isnull)
        return false;
    return salary > limit;
}
```

以版本 1 编码时，上面的函数形如：

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    TupleTableSlot  *t = (TupleTableSlot *) PG_GETARG_POINTER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    int32 salary;

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
    if (isnull)
        PG_RETURN_BOOL(false);
    /* Alternatively, we might prefer to do PG_RETURN_NULL() for null
 salary. */

    PG_RETURN_BOOL(salary > limit);
}
```

`GetAttributeByName` 是 PostgreSQL 的系统函数，返回指定行中的属性。它有三个参数：传给函数的 `TupleTableSlot*` 类型参数、所需属性的名称，以及一个返回参数（告知该属性是否为空）。`GetAttributeByName` 返回一个 `Datum` 值，你可以用适当的 `DatumGet`*`XXX`*`()` 宏把它转换为正确的数据类型。

下面的命令在 SQL 中声明函数 `c_overpaid`：

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
    AS 'DIRECTORY/funcs', 'c_overpaid'
    LANGUAGE C;
```

33.7.8. 从 C 语言函数中返回行（复合类型）

要从 C 语言函数中返回一行或一个复合类型值，可以使用一套特殊的 API，它通过一组宏和函数隐藏了构造复合数据类型时的大部分复杂性。要使用这套 API，源文件中必须包含：

```
#include "funcapi.h"
```

对返回复合数据类型（即行）的支持始于 `AttInMetadata` 结构。该结构持有从原始 C 字符串创建一行所需的各属性信息的数组。结构中包含的信息派生自 `TupleDesc` 结构，但存储它是为了避免在每次调用返回集合的函数时做冗余计算（见下一节）。对于返回集合的函数，`AttIn Metadata` 结构应在第一次调用时计算一次并保存，供后续调用复用。`AttInMetadata` 还保存指向原始 `TupleDesc` 的指针。

```
typedef struct AttInMetadata
{
    /* full TupleDesc */
    TupleDesc       tupdesc;

    /* array of attribute type input function finfo */
    FmgrInfo       *attinfuncs;

    /* array of attribute type typelem */
    Oid            *attelems;

    /* array of attribute typmod */
    int32        *atttypmods;
} AttInMetadata;
```

为帮助你填充此结构，有若干函数和一个宏可用。使用

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

获取具名关系的 `TupleDesc`，或者

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

基于类型 OID 获取 `TupleDesc`。后者可用于获取基本类型或复合类型的 `TupleDesc`。然后

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

将返回一个指向 `AttInMetadata` 的指针，它基于给定的 `TupleDesc` 初始化。`AttInMetadata` 可与 C 字符串一起使用，产生格式正确的行值（内部称为元组）。

要返回元组，你必须基于 `TupleDesc` 创建一个元组槽。你可以使用

```
TupleTableSlot *TupleDescGetSlot(TupleDesc tupdesc)
```

初始化此元组槽，或通过其他（用户提供的）手段获得一个。需要元组槽来创建供函数返回的 `Datum`。同一槽位可以（而且应该）在每次调用时复用。

在构造好 `AttInMetadata` 结构之后，

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char
 **values)
```

可用于从 C 字符串形式的用户数据构建 `HeapTuple`。`values` 是一个 C 字符串数组，返回行的每个属性对应一个。每个 C 字符串都应具有该属性数据类型的输入函数所期望的形式。要为某个属性返回空值，应把 *`values`* 数组中相应的指针设为 `NULL`。对返回的每一行，都需要再次调用此函数。

只有当你的函数自然地以文本字符串计算要返回的值时，通过 `TupleDescGetAttInMetadata` 和 `BuildTupleFromCStrings` 构建元组才是方便的。如果你的代码自然地把值计算为一组 `Datum` 值，就应改用底层函数 `heap_formtuple` 把 `Datum` 值直接转换为元组。你仍然需要 `TupleDesc` 和 `TupleTableSlot`，但不需要 `AttInMetadata`。

在函数中构建好要返回的元组后，必须把它转换为 `Datum`。使用

```
TupleGetDatum(TupleTableSlot *slot, HeapTuple tuple)
```

从元组和槽得到 `Datum`。如果你只打算返回单行，这个 `Datum` 可以直接返回；在返回集合的函数中，它可以用作当前的返回值。

下一节中会有一个示例。

33.7.9. 从 C 语言函数中返回集合

还有一种特殊的 API，为从 C 语言函数返回集合（多行）提供支持。集合返回函数必须遵循版本 1 调用约定。此外，与前述一样，源文件必须包含`funcapi.h`。

返回集合的函数（SRF）对它返回的每个项各调用一次。因此 SRF 必须保存足够的状态来记住自己正在做什么，并在每次调用时返回下一个项。结构 `FuncCallContext` 就是用来帮助控制这一过程的。在函数内，`fcinfo->flinfo->fn_extra` 用于跨调用保存指向 `FuncCallContext` 的指针。

```
typedef struct
{
    /*
     * Number of times we've been called before
     *
     * call_cntr is initialized to 0 for you by SRF_FIRSTCALL_INIT(),
 and
     * incremented for you every time SRF_RETURN_NEXT() is called.
     */
    uint32 call_cntr;

    /*
     * OPTIONAL maximum number of calls
     *
     * max_calls is here for convenience only and setting it is
 optional.
     * If not set, you must provide alternative means to know when the
     * function is done.
     */
    uint32 max_calls;

    /*
     * OPTIONAL pointer to result slot
     *
     * slot is for use when returning tuples (i.e., composite data
 types)
     * and is not needed when returning base data types.
     */
    TupleTableSlot *slot;

    /*
     * OPTIONAL pointer to miscellaneous user-provided context
 information
     *
     * user_fctx is for use as a pointer to your own data to retain
     * arbitrary context information between calls of your function.
     */
    void *user_fctx;

    /*
     * OPTIONAL pointer to struct containing attribute type input
 metadata
```

```
     *
     * attinmeta is for use when returning tuples (i.e., composite
 data types)
     * and is not needed when returning base data types. It
     * is only needed if you intend to use BuildTupleFromCStrings() to
 create
     * the return tuple.
     */
    AttInMetadata *attinmeta;

    /*
     * memory context used for structures that must live for multiple
 calls
     *
     * multi_call_memory_ctx is set by SRF_FIRSTCALL_INIT() for you,
 and used
     * by SRF_RETURN_DONE() for cleanup. It is the most appropriate
 memory
     * context for any memory that is to be reused across multiple
 calls
     * of the SRF.
     */
    MemoryContext multi_call_memory_ctx;
} FuncCallContext;
```

SRF 会使用若干自动操纵 `FuncCallContext` 结构体（并期望通过 `fn_extra` 找到它）的函数和宏。使用：

```
SRF_IS_FIRSTCALL()
```

来判断你的函数是第一次被调用还是后续调用。在第一次调用时（只在第一次调用时）使用：

```
SRF_FIRSTCALL_INIT()
```

来初始化 `FuncCallContext`。在每次函数调用时，包括第一次，使用：

```
SRF_PERCALL_SETUP()
```

来正确设置对 `FuncCallContext` 的使用，并清除上一次处理遗留的任何已返回数据。

如果你的函数有数据要返回，使用：

```
SRF_RETURN_NEXT(funcctx, result)
```

把它返回给调用者（`result`必须是类型`Datum`，
可以是一个单一值或者按上文所述准备好的元组）。最后，当函数完成了数据返回后，可使用：

```
SRF_RETURN_DONE(funcctx)
```

来清理并且结束SRF。

调用 SRF 时当前所处的内存上下文是一个瞬时上下文，它会在两次调用之间被清空。这意味着，你不必对用 `palloc` 分配的所有东西都调用 `pfree`，因为它们反正会自动释放。不过，如果你需要分配在多次调用之间持续存在的数据结构，就必须把它们放到别处。对于任何需要一直存活到 SRF 运行结束的数据，`multi_call_memory_ctx` 所指向的内存上下文就是合适的位置。在大多数情况下，这意味着你应当在做首次调用初始化时切换到 `multi_call_memory_ctx`。

一个完整的伪代码示例如下：

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    MemoryContext     oldcontext;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_
ctx);
        /* One-time setup code appears here: */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
            obtain slot
            funcctx->slot = slot;
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }

    /* Each-time setup code appears here: */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* this is just one way we might test whether we are done: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* Here we want to return another item: */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* Here we are done returning items and just need to clean up:
 */
        user code
        SRF_RETURN_DONE(funcctx);
    }
}
```

一个返回复合类型的简单 SRF 的完整示例如下：

```
PG_FUNCTION_INFO_V1(testpassbyval);

Datum
testpassbyval(PG_FUNCTION_ARGS)
```

```
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    TupleTableSlot      *slot;
    AttInMetadata       *attinmeta;

     /* stuff done only on the first call of the function */
     if (SRF_IS_FIRSTCALL())
     {
        MemoryContext oldcontext;

        /* create a function context for cross-call persistence */
        funcctx = SRF_FIRSTCALL_INIT();

        /* switch to memory context appropriate for multiple function
 calls */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_
ctx);

        /* total number of tuples to be returned */
        funcctx->max_calls = PG_GETARG_UINT32(0);

        /* Build a tuple description for a __testpassbyval tuple */
        tupdesc = RelationNameGetTupleDesc("__testpassbyval");

        /* allocate a slot for a tuple with this tupdesc */
        slot = TupleDescGetSlot(tupdesc);

        /* assign slot to function context */
        funcctx->slot = slot;

        /*
         * generate attribute metadata needed later to produce tuples
 from raw
         * C strings
         */
        attinmeta = TupleDescGetAttInMetadata(tupdesc);
        funcctx->attinmeta = attinmeta;

        MemoryContextSwitchTo(oldcontext);
    }

    /* stuff done on every call of the function */
    funcctx = SRF_PERCALL_SETUP();

    call_cntr = funcctx->call_cntr;
    max_calls = funcctx->max_calls;
    slot = funcctx->slot;
    attinmeta = funcctx->attinmeta;

    if (call_cntr < max_calls)    /* do when there is more left to
 send */
```

```
    {
        char       **values;
        HeapTuple    tuple;
        Datum        result;

        /*
         * Prepare a values array for storage in our slot.
         * This should be an array of C strings which will
         * be processed later by the type input functions.
         */
        values = (char **) palloc(3 * sizeof(char *));
        values[0] = (char *) palloc(16 * sizeof(char));
        values[1] = (char *) palloc(16 * sizeof(char));
        values[2] = (char *) palloc(16 * sizeof(char));

        snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
        snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
        snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

        /* build a tuple */
        tuple = BuildTupleFromCStrings(attinmeta, values);

        /* make the tuple into a datum */
        result = TupleGetDatum(slot, tuple);

        /* clean up (this is not really necessary) */
        pfree(values[0]);
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* do when there is no more left */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

声明此函数的 SQL 代码为：

```
CREATE TYPE __testpassbyval AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION testpassbyval(integer, integer) RETURNS
 SETOF __testpassbyval
    AS 'filename', 'testpassbyval'
    LANGUAGE C IMMUTABLE STRICT;
```

源码发布中的 `contrib/tablefunc` 目录包含更多返回集合函数的示例。

33.7.10. 多态参数和返回类型

C 语言函数可以声明为接受和返回多态类型 `anyelement` 和 `anyarray`。关于多态函数的更详细解释，见第 33.2.5 节。当函数参数或者返回类型被定义为多态类型时，函数的编写者无法提前

知道会用什么数据类型调用该函数或者该函数需要返回什么数据类型。fmgr.h 中提供了两种例程，允许版本-1 的 C 函数发现其参数的实际数据类型以及它要返回的类型。这些例程被称为 get_fn_expr_rettype(FmgrInfo *flinfo) 和 get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)。它们返回结果或者参数的类型的 OID，或者当该信息不可用时返回 InvalidOid。结构体 flinfo 通常被当做 fcinfo->flinfo 访问。参数 argnum 则是从零开始计。

例如，假设我们想要写一个接收一个任意类型元素并且返回一个该类型的一维数组的函数：

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* 得到所提供的元素 */
    element = PG_GETARG_DATUM(0);

    /* 只有一个维度 */
    ndims = 1;
    /* 和一个元素 */
    dims[0] = 1;
    /* 且下界是 1 */
    lbs[0] = 1;

    /* 得到该元素类型所需的信息 */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* 现在构建数组 */
    result = construct_md_array(&element, ndims, dims, lbs,
                                element_type, typlen, typbyval,
 typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

下面的命令在 SQL 中声明了 make_array 函数：

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE C STRICT;
```

注意这里使用了STRICT；这一点至关重要，因为代码并没有费心去测试空值输入。

33.8. 函数重载

可以用同一个 SQL 名称定义多个函数，只要它们接受的参数不同即可。换句话说，函数名可以被重载。当一个查询被执行时，
服务器将根据所提供参数的数据类型和数量来决定要调用哪个函数。
重载也可用来模拟具有可变参数个数（最大个数有限）的函数。

函数也可以与属性同名。（回忆一下，`attribute(table)` 等价于 `table.attribute`。）当复合类型上的函数与该复合类型的属性存在歧义时，总是使用属性。

在创建一个重载函数家族时，应该小心不要创建歧义。例如，给定函数：

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

对于`test(1, 1.5)`这样的输入就无法立刻清楚地知道应该调用哪个函数。当前实现的解决规则在第 10 章中有描述，但是设计一个依赖于这种行为的系统是不明智的。

重载 C 语言函数时还有一个附加约束：重载函数族中每个函数的 C 名称必须不同于所有其他函数（无论是内部的还是动态装载的）的 C 名称。违反此规则时，行为不可移植。你可能遇到运行时链接器错误，或者其中一个函数被调用（通常是内部那个）。SQL `CREATE FUNCTION` 命令的 `AS` 子句的另一种形式可以把 SQL 函数名与 C 源代码中的函数名解耦。例如，

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
    LANGUAGE C;
```

这里 C 函数的名称反映了多种可能约定中的一种。

33.9. 用户自定义聚合

PostgreSQL 中的聚合函数是依据状态值和状态转移函数来定义的。也就是说，聚合通过一个状态值来工作，并在处理每个后续输入行时更新该状态值。要定义一个新的聚合函数，
需要为状态值选定一种数据类型、为状态选定一个初始值，并指定一个状态转移函数。
状态转移函数只是普通函数，也可以在聚合的上下文之外使用。
如果聚合的期望结果不同于运行状态值中需要保存的数据，还可以指定一个最终函数。

因此，除了聚合用户所看到的参数和结果数据类型之外，还存在一种内部状态值数据类型，
它可能既不同于参数类型，也不同于结果类型。

如果我们定义一个不使用最终函数的聚合，
那么得到的就是一个对每一行的列值进行逐步计算的聚合。`sum` 就是这种聚合的一个示例。`sum` 从零开始，并始终把当前行的值加到运行总和中。例如，如果我们希望让 `sum` 聚合能用于复数数据类型，那么只需要该数据类型的加法函数。聚合定义如下：

```
CREATE AGGREGATE complex_sum (
    sfunc = complex_add,
    basetype = complex,
    stype = complex,
    initcond = '(0,0)'
);
```

```
SELECT complex_sum(a) FROM test_complex;

 complex_sum
-------------
 (34,53.9)
```

（实际上，我们只需把聚合命名为 sum，并依靠 PostgreSQL 自己判断对 complex 类型的列应用哪种 sum。）

如果没有非空的输入值，上面 sum 的定义将返回零（初始状态条件）。也许我们希望在这种情形下改为返回 null——SQL 标准期望 sum 表现出那样的行为。要做到这一点，只需省略 initcond 子句，使初始状态条件为 null。通常这意味着 sfunc 需要检查 null 的状态条件输入，但对于 sum 以及 max、min 等一些简单聚合，只需把第一个非空输入值插入状态变量，然后从第二个非空输入值开始应用转换函数就足够了。当初始条件为 null 且转换函数被标记为“strict”（即不对 null 输入调用）时，PostgreSQL 会自动完成这一工作。

“strict”转移函数的另一项默认行为是，只要遇到空值输入，就会保留先前的状态值不变。
因此，空值会被忽略。如果你需要对空值输入采用其他行为，就不要把转移函数声明为 strict，而应在函数代码中自行检测空值输入并完成所需处理。

avg（平均值）是更复杂的聚合示例。它需要两个运行状态：输入的总和与输入的个数。
最终结果由二者相除得到。平均值通常用两元素数组作为状态值来实现。例如，avg(float8) 的内置实现如下所示：

```
CREATE AGGREGATE avg (
    sfunc = float8_accum,
    basetype = float8,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0}'
);
```

聚合函数可以使用多态的状态转移函数或最终函数，这样同一组函数就能用于实现多个聚合。
关于多态函数的解释，请参见第 33.2.5 节。再进一步，聚合函数自身也可以指定为具有多态输入类型和状态类型，从而让同一个聚合定义服务于多种输入数据类型。
下面是一个多态聚合的示例：

```
CREATE AGGREGATE array_accum (
    sfunc = array_append,
    basetype = anyelement,
    stype = anyarray,
    initcond = '{}'
);
```

对于任何给定的聚合调用，其实际状态类型都是一种数组类型，
该数组以实际输入类型为元素类型。

下面是以两种不同的实际数据类型作为参数时的输出：

```
SELECT attrelid::regclass, array_accum(attname)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_user'::regclass
    GROUP BY attrelid;
```

```
 attrelid |                                array_accum
----------+-------------------------------------------------------------
------------------
 pg_user  | {usename,usesysid,usecreatedb,usesuper,usecatupd,passwd,
valuntil,useconfig}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_user'::regclass
    GROUP BY attrelid;

 attrelid |         array_accum
----------+-----------------------------
 pg_user  | {19,23,16,16,16,25,702,1009}
(1 row)
```

更多细节请参见CREATE AGGREGATE命令。

33.10. 用户定义的类型

如第 33.2 节所述，PostgreSQL可以扩展以支持新的数据类型。本节描述如何定义新的基本类型，即在SQL 语言层级之下定义的数据类型。创建一个新的基本类型需要用低级语言（通常是C）实现操作该类型的函数。

本节中的示例可以在源码发行版的`src/tutorial`目录下的 `complex.sql`和`complex.c`中找到。关于运行这些示例的说明，请参阅该目录中的 `README`文件。

用户定义的类型必须始终具有输入和输出函数。
这些函数决定该类型在字符串中如何表示（供用户输入和输出给用户）以及在内存中如何组织。
输入函数接受一个以空字符结尾的字符串作为参数，并返回该类型的内部（内存中）表示。
输出函数接受该类型的内部表示作为参数，并返回一个以空字符结尾的字符串。
如果我们想对该类型做的不仅仅是存储它，
就必须提供额外的函数来实现我们希望该类型拥有的任何操作。

假设我们想定义一个表示复数的类型`complex`。在内存中表示复数的一种自然方式是下面的 C 结构：

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

我们需要把它做成传引用的类型，因为它太大，无法放入单个 `Datum`值中。

作为该类型的外部字符串表示，我们选择形如`(x,y)` 的字符串。

输入和输出函数通常不难编写，输出函数尤其如此。但在定义该类型的外部字符串表示时，
请记住：你最终必须为该表示编写一个完整而健壮的解析器作为输入函数。例如：

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
```

```
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

输出函数可以简单地写成：

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

你应当小心地让输入和输出函数互为逆函数。否则，在你需要把数据转储到文件中再读回来时，就会出现严重的问题。当涉及浮点数时，这是一个特别常见的问题。

可选地，用户定义的类型可以提供二进制输入和输出例程。二进制 I/O 通常比文本 I/O 更快但可移植性更差。与文本 I/O 一样，外部二进制表示究竟是什么由你自己定义。大多数内建数据类型都试图提供与机器无关的二进制表示。对于 `complex`，我们将借用`float8`类型的二进制 I/O 转换器：

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);
```

```
Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

要定义complex类型，我们需要在创建类型之前先创建用户定义的 I/O 函数：

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;
```

注意，输入和输出函数的声明必须引用尚未定义的类型。这是允许的，
但会产生一些可以忽略的警告消息。输入函数必须出现在前面。

最后，我们可以声明该数据类型：

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

当你定义一个新的基本类型时，PostgreSQL 会自动提供对该类型数组的支持。由于历史原因，
数组类型的名称与基本类型相同，只是在前面加上下划线字符（_）。

一旦该数据类型存在，我们就可以声明额外的函数来为它提供有用的操作。
然后可以在这些函数之上定义操作符，如果需要，还可以创建操作符类来支持该类型的索引。
这些额外的层次将在后续各节中讨论。

如果你的数据类型的值（内部形式）大小可能超过几百字节，就应当让该数据类型支持 TOAST。为此，内部表示必须遵循变长数据的标准布局：前四个字节必须是一个`int32`，包含该 datum 的总字节长度（包括其自身）。操作该数据类型的 C 函数必须小心地解开交给它们的任何已 toast 的值（这个细节通常可以隐藏在`GETARG`宏中）。然后，在运行 `CREATE TYPE`命令时，把内部长度指定为 `variable`并选择合适的存储选项。

更多细节请参阅CREATE TYPE命令的描述。

33.11. 用户自定义操作符

每个操作符都是对某个完成实际工作的底层函数调用的“语法糖”；因此你必须先创建底层函数，然后才能创建操作符。然而，操作符不仅仅是语法糖，因为它还携带额外信息，帮助查询规划器优化使用该操作符的查询。下一节将专门解释这些额外信息。

PostgreSQL 支持左一元、右一元和二元操作符。操作符可以重载；也就是说，同一个操作符名可以用于不同的操作符，而这些操作符具有不同数量和类型的操作数。执行查询时，系统会根据提供的操作数数量和类型确定应调用哪个操作符。

下面是创建一个把两个复数相加的操作符的例子。我们假设已经创建过 `complex` 类型的定义（见第 33.10 节）。首先需要一个完成工作的函数，然后就可以定义操作符：

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

现在我们可以执行这样的查询：

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

我们在这里展示了如何创建二元操作符。要创建一元操作符，只需省略 `leftarg`（左一元）或 `rightarg`（右一元）之一即可。`procedure` 子句和参数子句是 `CREATE OPERATOR` 中仅有的必选项。例子中展示的 `commutator` 子句是给查询优化器的一个可选提示。关于 `commutator` 和其他优化器提示的更多细节见下一节。

33.12. 操作符优化信息

PostgreSQL 的操作符定义可以包含若干可选子句，告诉系统关于操作符行为的有用信息。只要合适就应当提供这些子句，因为它们能给使用该操作符的查询的执行带来可观的加速。但如果你提供了它们，就必须确保它们是对的！错误地使用优化子句可能导致服务器进程崩溃、微妙错误的输出或其他糟糕后果。如果不确定，你随时可以省略某个优化子句；唯一的后果是查询可能比所需的更慢。

未来的 PostgreSQL 版本可能会添加更多优化子句。这里描述的是 7.4.30 版本理解的所有子句。

33.12.1. COMMUTATOR

如果给出 COMMUTATOR 子句，它指定一个与正在定义的操作符互为交换子的操作符。若对所有可能的输入值 x、y，都有 (x A y) 等于 (y B x)，则称操作符 A 是操作符 B 的交换子。注意，B 也同样是 A 的交换子。例如，对某种特定数据类型而言，< 和 > 通常互为交换子，而操作符 + 通常与其自身可交换。但操作符 – 通常并不与任何操作符可交换。

可交换操作符的左操作数类型与其交换子的右操作数类型相同，反之亦然。因此，只要给出交换子操作符的名称，PostgreSQL 就足以查找到交换子，这也正是 COMMUTATOR 子句需要提供的全部内容。

为将用于索引和连接子句的操作符提供交换子信息至关重要，因为这使查询优化器能够把这样的子句“翻转”成不同计划类型所需的形式。例如，考虑一个带有形如 tab1.x = tab2.y 的 WHERE 子句的查询，其中 tab1.x 和 tab2.y 是某种用户自定义类型，并假设 tab2.y 上有索引。除非优化器能确定如何把该子句翻转成 tab2.y = tab1.x，否则无法生成索引扫描，因为索引扫描机制期望看到索引列位于给定操作符的左侧。PostgreSQL 不会简单地假定这是一个有效的变换——= 操作符的创建者必须通过给操作符标记交换子信息来指明该变换有效。

定义一个与自身可交换的操作符时，直接定义即可。但定义一对可交换操作符时，情况就稍微复杂一些：第一个要定义的操作符如何引用另一个尚未定义的操作符呢？这个问题有两种解决办法：

- 一种办法是在你定义的第一个操作符中省略 COMMUTATOR 子句，然后在第二个操作符的定义中给出它。由于 PostgreSQL 知道可交换操作符成对出现，当它看到第二个定义时，会自动回过头来补填第一个定义中缺失的 COMMUTATOR 子句。

- 另一种更直接的办法是在两个定义中都包含 COMMUTATOR 子句。当 PostgreSQL 处理第一个定义并发现 COMMUTATOR 引用了一个尚不存在的操作符时，系统会在系统目录中为该操作符创建一个占位条目。这个占位条目只在操作符名称、左右操作数类型和结果类型上拥有有效数据，因为那是 PostgreSQL 此时所能推断出的全部。第一个操作符的目录条目会链接到这个占位条目。之后当你定义第二个操作符时，系统会用第二个定义中的额外信息更新该占位条目。如果你在占位条目被填充之前就尝试使用该占位操作符，只会得到一条错误消息。

33.12.2. NEGATOR

如果给出 NEGATOR 子句，它指定一个与正在定义的操作符互为求反器的操作符。若操作符 A 和 B 都返回布尔结果，并且对所有可能的输入 x、y，都有 (x A y) 等于 NOT (x B y)，则称 A 是 B 的求反器。注意，B 也同样是 A 的求反器。例如，对大多数数据类型而言，< 和 >= 构成一对求反器。一个操作符永远不可能合法地成为其自身的求反器。

与交换子不同，一对一元操作符完全可能合法地被标记为彼此的求反器；这意味着对所有 x，都有 (A x) 等于 NOT (B x)，或者对右一元操作符有等价的关系。

一个操作符的求反器必须与待定义操作符具有相同的左操作数类型和/或右操作数类型，因此与 COMMUTATOR 一样，在 NEGATOR 子句中只需给出操作符名称即可。

提供求反器对查询优化器很有帮助，因为它允许把 NOT (x = y) 这样的表达式简化为 x <> y。这种情况比你想象得更常见，因为 NOT 操作可能会作为其他重排的结果被插入进来。

可以使用上面解释的定义交换子对的相同方法，来定义成对的求反器操作符。

33.12.3. RESTRICT

如果给出 RESTRICT 子句，它指定该操作符的限制选择率估算函数。（注意，这里是函数名，而不是操作符名。）RESTRICT 子句只对返回 boolean 的二元操作符有意义。限制选择率估算器的作用，是猜测一张表中有多少比例的行会满足如下形式的 WHERE 子句条件：

```
column OP constant
```

（即针对当前操作符和某个特定常量值）。这会帮助优化器大致了解这种形式的 WHERE 子句会消掉多少行。（你可能在想：如果常量在左边会怎样？嗯，这正是 COMMUTATOR 的用途之一.....）

编写新的限制选择率估算函数远远超出了本章的范围，不过幸运的是，
对于你自己的很多操作符，通常都可以直接使用系统提供的某个标准估算器。
标准的限制选择率估算器如下：

eqsel 用于 =
neqsel 用于 <>
scalarltsel 用于 < 或 <=
scalargtsel 用于 > 或 >=

这些分类可能看起来有点奇怪，但仔细想想就会发现它们是有道理的。= 通常只会接受表中很小一部分行；<> 通常只会拒绝很小一部分行。< 接受的比例取决于给定常量落在该表列的值范围中的什么位置（恰好，这正是 ANALYZE 收集并提供给选择率估算器的信息）。对于相同的比较常量，<= 接受的比例会比 < 略大，但二者足够接近，不值得区分，尤其是无论如何我们很可能也只能做出粗略猜测。类似的说法也适用于 > 和 >=。

对于选择率非常高或非常低的操作符，即使它们实际上并不是真正的相等或不等比较，
你也常常可以勉强使用 eqsel 或 neqsel。例如，几何类型中的近似相等操作符就使用 eqsel，其依据是它们通常只会匹配表中很小一部分项。

对于那些能够以某种合理方式转换为数值标量、从而可进行范围比较的数据类型，你可以使用 scalarltsel 和 scalargtsel。如果可能的话，请把该数据类型加入函数 convert_to_scalar()（位于 src/backend/utils/adt/selfuncs.c）所理解的范围中。（最终，这个函数应被通过 pg_type 系统目录某一列标识的、按数据类型划分的函数所取代；但这件事目前还没有发生。）如果不这样做，系统仍然可以工作，
但优化器的估算效果就不会像本可达到的那样好。

在 src/backend/utils/adt/geo_selfuncs.c 中，还为几何操作符提供了其他选择率估算函数：areasel、positionsel 和 contsel。截至目前，这些函数都还只是桩实现，但你也许仍会想使用它们（或者更好的是，改进它们）。

33.12.4. JOIN

如果给出 JOIN 子句，它指定该操作符的连接选择率估算函数。（注意，这里是函数名，而不是操作符名。）JOIN 子句只对返回 boolean 的二元操作符有意义。连接选择率估算器的作用，是猜测两张表中有多少比例的行对会满足如下形式的 WHERE 子句条件：

```
table1.column1 OP table2.column2
```

（即针对当前操作符）。与 RESTRICT 子句一样，这也能极大地帮助优化器判断若干可能的连接顺序中哪一个可能耗时最少。

与前面一样，本章不会尝试解释如何编写连接选择率估算函数，
而只是建议你在适用时使用某个标准估算器：

eqjoinsel 用于 =
neqjoinsel 用于 <>
scalarltjoinsel 用于 < 或 <=
scalargtjoinsel 用于 > 或 >=
areajoinsel 用于基于二维面积的比较
positionjoinsel 用于基于二维位置的比较
contjoinsel 用于基于二维包含关系的比较

33.12.5. HASHES

如果存在 HASHES 子句，它会告知系统：在基于该操作符的连接中，可以使用哈希连接方法。HASHES 只对返回 boolean 的二元操作符有意义；在实践中，该操作符最好是某种数据类型上的相等关系。

哈希连接背后的假设是：只有当左值和右值被哈希到同一个哈希码时，连接操作符才有可能返回 true。如果两个值被放进不同的哈希桶，连接过程就根本不会去比较它们，这实际上隐含假定连接操作符的结果必定为 false。因此，对于那些并不表示相等关系的操作符，指定 HASHES 永远没有意义。

若要被标记为 HASHES，该连接操作符必须出现在某个哈希索引操作符类中。创建操作符时并不会强制这一点，因为那时引用它的操作符类当然还不存在。但如果没有这样的操作符类存在，运行时尝试在哈希连接中使用该操作符就会失败。系统需要通过该操作符类来查找与该操作符输入数据类型相对应的、特定于数据类型的哈希函数。当然，在创建该操作符类之前，你还必须先提供合适的哈希函数。

准备哈希函数时应格外小心，因为它有一些依赖机器的方式可能会导致结果不正确。例如，如果你的数据类型是某种结构体，其中可能包含无意义的填充位，那就不能简单地把整个结构体传给 hash_any。（除非你编写了其他操作符和函数，以确保这些未使用位始终为零，而这正是推荐的策略。）再比如，在符合 IEEE 浮点标准的机器上，负零和正零是不同的值（位模式不同），但它们被定义为比较相等。如果某个浮点值可能包含负零，就需要采取额外步骤，确保它生成与正零相同的哈希值。

> **注意**
>
> 作为可哈希连接操作符底层实现的函数，必须标记为 immutable 或 stable。如果它是 volatile，系统就永远不会尝试把该操作符用于哈希连接。

> **注意**
>
> 如果某个可哈希连接操作符的底层函数被标记为 strict，那么该函数还必须是完备的：也就是说，对于任意两个非空输入，它都应返回 true 或 false，而绝不能返回 null。如果不遵循这条规则，对 IN 操作进行哈希优化时可能会产生错误结果。（具体来说，IN 可能会在按标准本应返回 null 的地方返回 false；或者抛出一条错误，抱怨它没有为 null 结果做好准备。）

33.12.6. MERGES (SORT1, SORT2, LTCMP, GTCMP)

如果存在 MERGES 子句，它会告知系统：在基于该操作符的连接中，可以使用归并连接方法。MERGES 只对返回 boolean 的二元操作符有意义；在实践中，该操作符还必须表示某种数据类型或某对数据类型上的相等关系。

归并连接的基本思想，是先把左表和右表分别排序，然后同步扫描它们。因此，两种数据类型都必须能够被完全排序，而连接操作符必须只能在那对值位于排序次序中“同一位置”时才成功。实际效果上，这意味着连接操作符必须表现得像相等比较一样。但与哈希连接不同（哈希连接要求左右数据类型最好相同，或者至少按位等价），只要两种不同的数据类型在逻辑上兼容，就可以对它们进行归并连接。例如，smallint 与 integer 之间的相等操作符就是可归并连接的。我们只需要能把两种数据类型都带入逻辑兼容排序序列的排序操作符即可。

归并连接的执行要求系统能够识别出与该归并连接相等操作符相关的四个操作符：左操作数数据类型的小于比较、右操作数数据类型的小于比较、两种数据类型之间的小于比较，以及两种数据类型之间的大于比较。（如果可归并连接的操作符具有两个不同的操作数数据类型，这实际上是四个不同的操作符；但当两个操作数类型相同时，三个小于操作符是同一个操作符。）可以通过名称分别指定这些操作符，即分别使用 SORT1、SORT2、LTCMP 和 GTCMP 选项。在指定 MERGES 时，如果省略了其中任何一个，系统将分别填入默认名称 <、<、<、>。此外，如果这四个操作符选项中任何一个出现，就会认为隐含了 MERGES，因此可以只指定其中一部分，而让系统补全其余部分。

这四个比较操作符的操作数数据类型可以从可归并连接操作符的操作数类型推断出来，因此与 COMMUTATOR 一样，这些子句中只需给出操作符名称即可。除非你选用的操作符名称比较特殊，否则只需写 MERGES 并让系统补全细节就足够了。（与 COMMUTATOR 和 NEGATOR 一样，如果你碰巧先定义了相等操作符，系统也能够建立占位操作符条目。）

对于你标记为可归并连接的操作符，还有一些附加限制。这些限制目前不会被 CREATE OPERATOR 检查，但如果其中任何一条不成立，在使用该操作符时可能会出错：

- 一个可归并连接的相等操作符必须拥有一个可归并连接的交换子（如果两个操作数数据类型相同，则为它自身；如果不同，则应是相关的相等操作符）。
- 如果存在一个可归并连接的操作符关联某两个数据类型 A 和 B，又存在另一个可归并连接的操作符关联 B 和任何第三种数据类型 C，那么 A 和 C 也必须有一个可归并连接的操作符；换句话说，可归并连接操作符的存在必须是可传递的。
- 如果你指定的四个比较操作符没有以兼容的方式对数据值排序，运行时会出现奇怪的结果。

注意

作为可归并连接操作符底层实现的函数，必须标记为 immutable 或 stable。如果它是 volatile，系统就永远不会尝试把该操作符用于归并连接。

注意

在 7.3 之前的 PostgreSQL 版本中，没有 MERGES 简写：要创建可归并连接的操作符，必须显式写出 SORT1 和 SORT2 两者。此外，LTCMP 和 GTCMP 选项也不存在；这些操作符的名称被硬编码为 < 和 >。

33.13. 索引扩展接口

到目前为止所描述的过程使我们能够定义新类型、新函数以及新操作符。然而，我们还不能在一种新数据类型的列上定义索引。为此，必须为该新数据类型定义一个操作符类。

本节稍后将用一个示例说明这一概念：为 B-树索引方法定义一个新的操作符类，以便按绝对值升序存储和排序复数。

> **注意**
>
> 在 PostgreSQL 7.3 版本之前，要创建用户定义的操作符类，必须手工向系统目录 pg_amop、pg_amproc 和 pg_opclass 中添加内容。这种方式现已废弃，改为使用 CREATE OPERATOR CLASS，它是创建所需目录项的一种简单得多、也不易出错的方式。

33.13.1. 索引方法和操作符类

pg_am 表为每个索引方法（内部称为访问方法）保存一行。对表进行常规访问的支持内置于 PostgreSQL 中，但所有索引方法都在 pg_am 中描述。可以定义所需的接口例程，然后在 pg_am 中创建一行，从而添加新的索引方法 – 但这远超出了本章的范围。

索引方法的例程并不直接知道它将处理哪些数据类型。相反，一个
操作符类标识了索引方法在处理特定数据类型时需要使用的那组操作。之所以称为操作符类，
是因为它指定的一项内容就是可与索引一起使用的 WHERE 子句操作符集合（也就是能被转换成索引扫描条件的操作符）。操作符类还可以指定索引方法内部操作所需的某些支持过程，
但这些过程并不直接对应任何可与索引一起使用的 WHERE 子句操作符。

可以为同一种数据类型和索引方法定义多个操作符类。
这样就能为一种数据类型定义多套索引语义。例如，一个 B-树索引要求为其处理的每一种数据类型定义一种排序顺序。对于复数数据类型，也许既需要一个按复数绝对值排序的 B-树操作符类，
也需要另一个按实部排序的操作符类，等等。通常，其中一个操作符类会被视为最常用，
并标记为该数据类型在该索引方法上的默认操作符类。

同一个操作符类名可以用于多个不同的索引方法（例如，B-树和哈希索引方法都有名为 oid_ops 的操作符类），但每一个这样的类都是独立实体，必须分别定义。

33.13.2. 索引方法策略

与操作符类关联的操作符通过“策略号”来标识，
用以表示每个操作符在其操作符类上下文中的语义。例如，B-
树对键施加了严格的从小到大的顺序，因此像“小于”和“大于等于”这样的操作符，对 B-树来说就很重要。由于 PostgreSQL 允许用户定义操作符，PostgreSQL 不能仅凭操作符名称（例如 < 或 >=）就判断它是哪一类比较。取而代之的是，索引方法定义了一组“策略”，可以把它们看成是广义的操作符。每个操作符类都会说明，对于某种特定数据类型和某种索引语义解释，
每一种策略分别对应哪个实际操作符。

B-树索引方法定义了五种策略，如表 33.2所示。

表 33.2. B-树策略

操作	策略号
小于	1
小于等于	2
等于	3
大于等于	4
大于	5

哈希索引只支持等值比较，因此它们只使用一种策略，如表 33.3所示。

表 33.3. 哈希策略

操作	策略号
等于	1

R-树索引表达矩形包含关系。它们使用八种策略，如表 33.4所示。

表 33.4. R-树策略

操作	策略号
位于左侧	1
位于左侧或重叠	2
重叠	3
位于右侧或重叠	4
位于右侧	5
相同	6
包含	7
被包含	8

GiST 索引更加灵活：它们根本没有固定的策略集合。相反，每个特定 GiST 操作符类的“一致性”支持例程会按自己的方式解释策略号。

注意，所有策略操作符都返回布尔值。实际上，所有被定义为索引方法策略的操作符都必须返回 `boolean` 类型，因为要与索引配合使用，它们必须出现在 `WHERE` 子句的顶层。

顺便说一下，`pg_am` 中的 `amorderstrategy` 列表明索引方法是否支持有序扫描。零表示不支持；如果支持，`amorderstrategy` 就是与排序操作符对应的策略号。例如，B-树的 `amorderstrategy` = 1，即它的“小于”策略号。

33.13.3. 索引方法支持例程

仅靠策略信息通常不足以让系统知道如何使用索引。实际上，
索引方法还需要额外的支持例程才能工作。例如，B-树索引方法必须能够比较两个键，
并判断其中一个是大于、等于还是小于另一个。类似地，R-
树索引方法必须能够计算矩形的交集、并集和大小。这些操作并不对应 SQL 命令条件中使用的操作符；它们是索引方法内部使用的管理例程。

与策略一样，操作符类会标识对于给定的数据类型和语义解释，
应由哪些具体函数承担这些角色。索引方法定义它需要的函数集合，
而操作符类则会通过为函数分配由索引方法规定的“支持函数号”来标识正确的函数。

B-树要求一个支持函数，如表 33.5所示。

表 33.5. B-树支持函数

函数	支持号
比较两个键，并返回一个小于零、等于零或大于零的整数，用以表示第一个键是小于、等于还是大于第二个键	1

哈希索引同样只需要一个支持函数，如表 33.6所示。

表 33.6. 哈希支持函数

函数	支持号
计算一个键的哈希值	1

R-树索引要求三个支持函数，如表 33.7所示。

表 33.7. R-树支持函数

函数	支持号
union	1
intersection	2
size	3

GiST 索引要求七个支持函数，如表 33.8所示。

表 33.8. GiST 支持函数

函数	支持号
consistent	1
union	2
compress	3
decompress	4
penalty	5
picksplit	6
equal	7

与策略操作符不同，支持函数返回的是特定索引方法所期望的数据类型；例如，对 B-树的比较函数来说，就是一个有符号整数。

33.13.4. 示例

现在我们已经了解了这些基本思想，下面给出先前承诺的创建新操作符类示例。
（这个可运行示例位于源码发布包中的`src/tutorial/complex.c`和`src/tutorial/complex.sql`。）该操作符类封装了一组按绝对值顺序对复数排序的操作符，
因此我们把它命名为`complex_abs_ops`。首先，我们需要一组操作符。
定义操作符的过程已经在第 33.11 节中讨论过。对于 B-树上的操作符类，我们需要如下操作符：

- 绝对值小于（策略 1）
- 绝对值小于等于（策略 2）
- 绝对值等于（策略 3）
- 绝对值大于等于（策略 4）
- 绝对值大于（策略 5）

定义一组相关比较操作符时，最不容易出错的方式是先编写 B-树比较支持函数，再把其他函数写成围绕该支持函数的一行包装器函数。这样可以降低在边界情况下得到不一致结果的概率。
按照这种方法，我们首先编写：

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
```

```
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double      amag = Mag(a),
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

现在，小于函数如下所示：

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

其他四个函数的区别只在于它们如何比较内部函数的结果与 0。

接下来，在 SQL 中声明这些函数，以及基于这些函数的操作符：

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'filename', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

必须指定正确的交换子和求反器操作符，以及合适的限制选择率与连接选择率函数，
否则优化器无法有效使用索引。注意，小于、等于和大于这几种情况应使用不同的选择率函数。

这里还有几点值得注意：

- 只能有一个名为 = 且两个操作数都为 `complex` 类型的操作符。在这个例子里，我们并没有任何其他 = 操作符可用于 `complex`；但如果我们是在构造一种实际使用的数据类型，可能会希望 = 表示复数的普通相等，而不是绝对值相等。在那种情况下，我们就需要为 `complex_abs_eq` 选用其他操作符名。
- 尽管 PostgreSQL 能处理 SQL 名称相同但参数数据类型不同的函数，C 却只能处理给定名称的一个全局函数。因此，我们不应该把 C 函数简单命名成 `abs_eq` 之类。通常，在 C 函数名中包含数据类型名称是个好习惯，这样就不会与其他数据类型的函数发生冲突。
- 我们原本也可以把该函数的 PostgreSQL 名称取为 `abs_eq`，并依靠 PostgreSQL 通过参数数据类型把它与任何其他同名的 PostgreSQL 函数区分开。为了让示例保持简单，这里我们让 C 层和 PostgreSQL 层的函数使用相同的名称。

下一步是注册 B-树要求的支持例程。实现该例程的 C 示例代码与操作符函数位于同一个文件中。该函数的声明如下：

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

现在我们已经有了所需的操作符和支持例程，就可以最终创建操作符类：

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

这样就完成了！现在应该可以在 `complex` 列上创建并使用 B-树索引。

我们本来也可以把操作符项写得更详细一些，例如：

```
        OPERATOR        1       < (complex, complex) ,
```

但是当操作符接受的数据类型与该操作符类所服务的数据类型相同时，就没有必要这样写。

上述示例假定你希望把这个新操作符类设为 `complex` 数据类型的默认 B-树操作符类。如果不是这样，只需省去 `DEFAULT` 这个词。

33.13.5. 系统对操作符类的依赖

PostgreSQL利用操作符类来从多方面推断操作符的属性，而不仅仅是判断它们能否用于索引。因此，即便你并不打算为自己的数据类型列建立索引，也可能会想创建操作符类。

特别地，`ORDER BY`和`DISTINCT`等 SQL 特性要求对值的比较和排序。为了在用户定义的数据类型上实现这些特性，PostgreSQL会为数据类型查找默认 B-树操作符类。这个操作符类的“相等”成员定义了用于`GROUP BY`和`DISTINCT`的值的等值概念，而该操作符类施加的排序顺序定义了默认的`ORDER BY`顺序。

用户定义类型的数组比较也依赖于该类型默认 B-树操作符类所定义的语义。

如果一种数据类型没有默认的 B-树操作符类，系统就会查找默认的哈希操作符类。但由于这类操作符类只提供等值语义，因此在实践中它只足以支持数组相等比较。

如果某种数据类型没有默认操作符类，而你又试图将这些 SQL 特性用于该数据类型，就会得到类似“无法识别排序操作符”这样的错误。

> **注意**
>
> 在版本 7.4 以前的PostgreSQL中，排序和分组操作将隐式地使用名为=、<以及>的操作符。新的依赖于默认操作符类的行为避免了对具有特定名字的操作符行为作出任何假设。

33.13.6. 操作符类的特殊特性

还有两个操作符类的特殊特性我们尚未讨论，主要是因为它们对最常用的索引方法没有用处。

通常，把一个操作符声明为操作符类的成员，
意味着索引方法能够正好检索出满足使用该操作符的 WHERE 条件的行集。例如：

```
SELECT * FROM table WHERE integer_column < 4;
```

可以由整数列上的 B-tree 索引精确满足。但也存在这样的情况：索引只是查找匹配行的不精确向导。例如，如果一个 R-树索引只存储对象的包围盒，那么它就无法精确满足测试多边形等非矩形对象之间重叠的 WHERE 条件。不过，我们可以用索引找出包围盒与目标对象的包围盒重叠的对象，然后只对索引找到的对象做精确的重叠测试。如果适用这种情形，
就称该索引对该操作符是“有损”的，此时我们在 CREATE OPERATOR CLASS 命令的 OPERATOR 子句中加上 RECHECK。如果索引保证返回所有需要的行（可能还外加一些额外的行，它们可以通过执行原始操作符调用来排除），RECHECK 就是合法的。

再次考虑只在索引中存储多边形等复杂对象的包围盒的情况。此时，
在索引条目中存储整个多边形没有多少价值，不如只存储一个更简单的对象，其类型为box。
这种情况由STORAGE选项表达，该选项位于CREATE OPERATOR CLASS：可以写成如下形式：

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

目前，只有 GiST 索引方法支持与列数据类型不同的STORAGE类型。GiST 的compress和decompress支持例程在使用STORAGE时必须处理数据类型转换。

第 34 章 规则系统

本章讨论PostgreSQL中的规则系统。产生式规则系统在概念上很简单，
但在实际使用时涉及许多微妙之处。

有些其他数据库系统定义了主动型数据库规则，它们通常表现为存储过程和触发器。在PostgreSQL中，这类功能同样可以用函数和触发器实现。

规则系统（更准确地说，查询重写规则系统）与存储过程和触发器完全不同。它会修改查询，
使其将规则纳入考虑，然后把修改后的查询交给查询规划器进行规划和执行。它非常强大，
可用于查询语言过程、视图和版本等许多用途。关于这一规则系统的理论基础及其能力，另见[ston90b]和[ong90]。

34.1. 查询树

要理解规则系统如何工作，必须先知道它在何时被调用，以及它的输入和输出是什么。

规则系统位于解析器和规划器之间。它接收解析器的输出，即一棵查询树，
以及用户定义的重写规则；这些规则本身也是带有一些附加信息的查询树。
它会生成零棵或多棵查询树作为结果。因此，
它的输入和输出始终都是解析器本身也可能产生的东西，
所以它看到的任何内容基本上都可以表示为一个SQL语句。

那么什么是查询树？它是SQL语句的一种内部表示，其中构成语句的各个部分被分别存储。
如果你设置了配置参数`debug_print_parse`、`debug_print_rewritten`或`debug_print_plan`，这些查询树就可以显示在服务器日志中。规则动作也以查询树的形式存储在系统目录`pg_rewrite`中。它们的格式不像日志输出，但包含的却是完全相同的信息。

阅读原始查询树需要一些经验。但由于查询树的SQL表示已足以理解规则系统，
本章不会讲解如何阅读它们。

在阅读本章中查询树的 SQL 表示时，需要能够辨认语句在查询树结构中被拆分成了哪些部分。查询树的组成部分是

命令类型

这是一个简单的值，用来说明是哪一种命令（`SELECT`、`INSERT`、`UPDATE`、`DELETE`）
产生了该查询树。

范围表

范围表是查询中使用的关系列表。在 `SELECT` 语句中，它们就是关键字 `FROM` 后面给出的关系。

每个范围表项标识一个表或视图，并说明在查询的其他部分以哪个名称来称呼它。
在查询树中，范围表项是按编号而不是按名称引用的，因此即使像 SQL 语句中那样出现重名，这里也无关紧要。这种情况可能出现在规则的范围表被合并之后。
本章中的示例不会涉及这种情形。

结果关系

这是范围表中的一个索引，用来标识查询结果应写入哪个关系。

`SELECT` 查询通常没有结果关系。`SELECT INTO` 这一特殊情况与 `CREATE TABLE` 后接 `INSERT ... SELECT` 几乎相同，这里不再单独讨论。

对于 INSERT、UPDATE 和 DELETE 命令，结果关系就是要让修改生效的表（或视图！）。

目标列表

目标列表是定义查询结果的表达式列表。对于 SELECT，这些表达式构成查询的最终输出。它们对应于关键字 SELECT 和 FROM 之间的表达式。（* 只是某个关系全部列名的缩写。解析器会把它展开为各个独立列，因此规则系统永远不会看到它。）

DELETE 命令不需要目标列表，因为它们不产生任何结果。实际上，规划器会向空目标列表加入一个特殊的 CTID 项，但这发生在规则系统之后，将在后面讨论；对规则系统而言，目标列表是空的。

对于 INSERT 命令，目标列表描述的是将要进入结果关系的新行。它由 VALUES 子句中的表达式，或 INSERT ... SELECT 里 SELECT 子句中的表达式组成。重写过程的第一步会为原始命令未赋值但带有默认值的列补上目标列表项。其余列（既没有给定值也没有默认值）将由规划器填入常量空值表达式。

对于 UPDATE 命令，目标列表描述要替换旧行的新行。在规则系统中，它只包含命令中 SET column = expression 部分的表达式。规划器会通过插入把旧行值复制到新行的表达式来处理缺失列。与 DELETE 一样，它也会加入一个特殊的 CTID 项。

目标列表中的每一项都包含一个表达式，它可以是常量值、
指向范围表中某个关系列的变量、一个参数，或者由函数调用、常量、变量、
操作符等构成的表达式树。

条件

查询的条件是一个表达式，与目标列表项中的表达式很相似。该表达式的结果值是布尔值，用来说明是否应对最终结果行执行相应操作（INSERT、UPDATE、DELETE 或 SELECT）。它对应于 SQL 语句的 WHERE 子句。

连接树

查询的连接树显示了 FROM 子句的结构。对于 SELECT ... FROM a, b, c 这样的简单查询，连接树只是 FROM 项的一个列表，因为允许按任意顺序连接它们。但当使用 JOIN 表达式，特别是外连接时，就必须按连接所显示的顺序进行连接。在这种情况下，连接树展示的是 JOIN 表达式的结构。与特定 JOIN 子句相关的限制（来自 ON 或 USING 表达式）会作为条件表达式附着在相应的连接树节点上。把顶层 WHERE 表达式也存储为附着在顶层连接树项上的一个条件，会很方便。因此，连接树实际上同时表示 SELECT 的 FROM 和 WHERE 子句。

其他

查询树的其他部分，如 ORDER BY 子句，这里不作关注。规则系统在应用规则时会替换其中某些项，但这与规则系统的基本原理关系不大。

34.2. 视图和规则系统

在 PostgreSQL 中，视图通过规则系统实现。实际上，以下命令：

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

与下面这两条命令基本没有区别：

```
CREATE TABLE myview (same column list as mytab);
```

```
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

因为这正是 CREATE VIEW 命令在内部所做的事情。这会带来一些副作用。其中之一是，在 PostgreSQL 系统目录中，视图的信息与表的信息完全相同。因此，对解析器而言，表和视图完全没有区别。它们是同一种东西：关系。

34.2.1. SELECT 规则如何工作

ON SELECT规则会在所有查询上作为最后一步应用，即使给出的命令是INSERT、UPDATE或DELETE也一样。它们与其他命令类型上的规则在语义上不同，因为它们是就地修改查询树，而不是创建新的查询树。因此我们先讨论SELECT规则。

目前，一个ON SELECT规则中只能有一个动作，而且它必须是一个无条件、带有INSTEAD的SELECT动作。之所以有这个限制，是为了让规则足够安全，从而能够向普通用户开放；它也把ON SELECT规则限制为真正的视图规则。

本章的示例是两个进行一些计算的连接视图，以及另外一些依次使用它们的视图。最初两个视图中的一个会在后面通过为INSERT、UPDATE和DELETE操作添加规则来定制，从而最终得到一个在行为上像真正的表、但又带有某些特殊功能的视图。作为入门示例，这并不算简单，因此会让理解变得更难一些。但与其使用许多可能令人混淆的不同示例，不如用一个示例逐步覆盖这里讨论的全部要点。

对于这个例子，我们需要一个返回两个整数值中较小者的 min 函数。我们这样创建它：

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS '
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
' LANGUAGE SQL STRICT;
```

在前两节对规则系统的描述中，我们需要用到如下真实表：

```
CREATE TABLE shoe_data (
    shoename   text,          -- primary key
    sh_avail   integer,       -- available number of pairs
    slcolor    text,          -- preferred shoelace color
    slminlen   real,          -- minimum shoelace length
    slmaxlen   real,          -- maximum shoelace length
    slunit     text           -- length unit
);

CREATE TABLE shoelace_data (
    sl_name    text,          -- primary key
    sl_avail   integer,       -- available number of pairs
    sl_color   text,          -- shoelace color
    sl_len     real,          -- shoelace length
    sl_unit    text           -- length unit
);

CREATE TABLE unit (
    un_name    text,          -- primary key
    un_fact    real           -- factor to transform to cm
);
```

如你所见，它们表示的是鞋店数据。

这些视图的创建方式如下：

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

创建 shoelace 视图的 CREATE VIEW 命令（这是我们这里最简单的一个例子）会创建一个关系 shoelace，并在 pg_rewrite 中创建一项，说明只要查询的范围表中引用了关系 shoelace，就必须应用一条重写规则。该规则没有规则条件（稍后在讨论非 SELECT 规则时再谈，因为目前 SELECT 规则不能有规则条件），并且它是 INSTEAD 规则。注意，规则条件与查询条件不是一回事。这里我们的规则动作带有一个查询条件。规则动作本身是一棵查询树，它是视图创建命令中 SELECT 语句的一个副本。

> **注意**
>
> 你在pg_rewrite项中看到的、用于NEW和OLD的两个额外范围表项（在打印出来的查询树中，它们出于历史原因被命名为*NEW*和*OLD*），与SELECT规则无关。

现在我们填充 unit，shoe_data 和 shoelace_data，然后对视图运行一个简单查询：

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
```

```
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

这是你能在这些视图上执行的最简单的SELECT，因此我们借此机会说明视图规则的基础。SELECT * FROM shoelace由解析器解释后，会生成如下查询树：

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

然后它会被交给规则系统。规则系统遍历范围表，检查其中的关系是否有相应规则。在处理shoelace的范围表项时（到目前为止只有这一个），它会找到带有如下查询树的_RETURN规则：

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

为了展开视图，重写器只需创建一个子查询范围表条目，其中包含规则的动作查询树，然后用这个范围表条目替换原先引用视图的条目。
所得的重写后查询树几乎等同于输入以下语句的结果：

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
```

```
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

不过有一点不同：子查询的范围表中有两个额外条目，shoelace *OLD* 和 shoelace *NEW*。这些条目不直接参与查询，因为子查询的连接树或目标列表并未引用它们。重写器用它们保存原先引用视图的范围表条目中的访问权限检查信息。这样，即使重写后的查询没有直接使用视图，执行器仍会检查用户是否具有访问该视图所需的权限。

这就是应用的第一条规则。规则系统接着会检查顶层查询中剩余的范围表项（本例中已经没有了），并递归检查新增子查询中的范围表项，看它们是否引用了视图。（但它不会展开*OLD*或*NEW*，否则就会出现无限递归！）在这个例子中，shoelace_data和unit都没有重写规则，因此重写到此结束，上面的结果就是交给规划器的最终结果。

现在我们想写一个查询，找出商店里目前哪些鞋子有颜色和长度都匹配的鞋带，并且完全匹配的总双数大于等于二。

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

这一次解析器的输出是如下查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

首先应用的是用于 shoe_ready 视图的规则，它得到如下查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

类似地，shoe 和 shoelace 的规则会被替换进子查询的范围表中，最终得到一棵三层查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
                       sh.slcolor,
                       sh.slminlen,
                       sh.slminlen * un.un_fact AS slminlen_cm,
                       sh.slmaxlen,
                       sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                       sh.slunit
                  FROM shoe_data sh, unit un
                 WHERE sh.slunit = un.un_name) rsh,
               (SELECT s.sl_name,
                       s.sl_avail,
                       s.sl_color,
                       s.sl_len,
                       s.sl_unit,
                       s.sl_len * u.un_fact AS sl_len_cm
                  FROM shoelace_data s, unit u
                 WHERE s.sl_unit = u.un_name) rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail > 2;
```

事实证明，规划器会把这棵树折叠成两层的查询树：最底层的 SELECT 命令会被“上拉”到中间的 SELECT 中，因为不需要单独处理它们。但中间的 SELECT 会保持与顶层分离，因为它包含聚合函数。如果把那些也上拉，就会改变顶层 SELECT 的行为，而这不是我们想要的。不过，折叠查询树属于一种优化，重写系统本身无须关心。

注意

规则系统目前对视图规则没有递归终止机制（对其他类型的规则才有）。这并不会造成太大问题，因为要把处理推入无限循环（使服务器进程不断膨胀，直至达到内存上限）的唯一方式，是先创建一些表，然后手工用 CREATE RULE 把视图规则设置成一个从另一个选择、而另一个又从这个选择的形式。如果使用 CREATE VIEW，这种情况绝不会发生，因为在执行第一个 CREATE VIEW 时，第二个关系尚不存在，因此第一个视图无法从第二个视图选择。

34.2.2. 非 SELECT 语句中的视图规则

上面关于视图规则的描述没有涉及查询树的两个细节：命令类型和结果关系。事实上，视图规则并不需要这些信息。

SELECT的查询树与其他任何命令的查询树之间只有少数差别。显然，它们的命令类型不同；对于SELECT之外的命令，结果关系会指向结果应写入的那个范围表项。除此之外，
其余部分完全相同。因此，假设有两个表t1和t2，都具有列a和b，那么下面两条语句的查询树：

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b WHERE t1.a = t2.a;
```

几乎是一样的。特别是：

- 范围表中包含表t1和t2的项。
- 目标列表中都包含一个变量，该变量指向表t2的范围表项中的列b。
- 条件表达式会比较两个范围表项中的列a是否相等。
- 连接树都表示t1与t2之间的一次简单连接。

结果是，这两个查询树都会产生相似的执行计划：它们都是对这两个表的连接。对于 UPDATE，规划器会把 t1 中缺失的列补入目标列表，最终查询树会变成：

```
UPDATE t1 SET a = t1.a, b = t2.b WHERE t1.a = t2.a;
```

因而，执行器在这个连接上运行时会产生与下面语句完全相同的结果集：

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

但在 UPDATE 中有个小问题：执行器并不关心它所做的连接的结果将被用于什么。它只是生成一个行结果集。一个是 SELECT 命令，另一个是 UPDATE 命令，这一差别是在执行器的调用者中处理的。调用者仍然知道（通过查看查询树）这是一个 UPDATE，也知道这个结果应写入表 t1。但问题在于：其中哪一行应当被新行替换？

为了解决这个问题，UPDATE（以及 DELETE）语句的目标列表中会额外加入一项：当前元组 ID（CTID）。这是一个系统列，包含该行所在的文件块号以及在块中的位置。已知表之后，
就可以利用 CTID 取回要更新的 t1 原始行。把 CTID 加入目标列表后，查询实际上会变成：

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

现在还要引入 PostgreSQL 的另一个细节。旧表行不会被覆盖，这也是 ROLLBACK 之所以很快的原因。在 UPDATE 中，新结果行会被插入表中（去掉 CTID 之后），而 CTID 所指向的旧行，其行头中的 cmax 和 xmax 项会被设置为当前命令计数器和当前事务 ID。于是旧行被隐藏起来，事务提交后，清理器（vacuum）最终就可以真正移除它。

知道了这些以后，我们就可以用完全相同的方式把视图规则应用到任何命令上。没有区别。

34.2.3. PostgreSQL 中视图的能力

上文演示了规则系统如何把视图定义整合进原始查询树。在第二个示例中，
从一个视图发出的简单SELECT最终生成了一棵四表连接的查询树（unit
以不同名称使用了两次）。

用规则系统实现视图的好处在于：规划器能够在单棵查询树中同时看到哪些表必须被扫描、
这些表之间的关系、来自视图的限制条件，以及原始查询自身的条件。
即使原始查询本身已经是对若干视图的连接，情况也仍然如此。
规划器必须决定执行查询的最佳路径，而它掌握的信息越多，这个决定通常就越好。
PostgreSQL实现的规则系统能够保证，到那个阶段为止，
关于该查询的全部可用信息都已经集中在这里。

34.2.4. 更新视图

如果把一个视图指定为 INSERT、UPDATE 或 DELETE 的目标关系，会发生什么？在完成上述替换之后，我们得到的查询树中，结果关系会指向一个子查询范围表项。这是行不通的，因此重写器一旦发现自己产生了这样的结果，就会抛出一个错误。

要改变这种情况，我们可以定义规则来修改这类命令的行为。这是下一节的主题。

34.3. INSERT、UPDATE 和 DELETE 上的规则

定义在INSERT、UPDATE和DELETE上的规则与前几节描述的视图规则有明显的不同。首先，它们的CREATE RULE命令允许更多：

- 它们可以没有动作。
- 它们可以有多个动作。
- 它们可以是INSTEAD，也可以不是。
- 伪关系NEW和OLD可以派上用场。
- 它们可以有规则条件。

第二，它们不是就地修改查询树，而是创建零棵或多棵新查询树，并且可能丢弃原始查询树。

34.3.1. 更新规则如何工作

记住以下语法：

```
CREATE RULE rule_name AS ON event
    TO object [WHERE rule_qualification]
    DO [INSTEAD] [action | (actions) | NOTHING];
```

在后文中，更新规则是指定义在INSERT、UPDATE或DELETE上的规则。

当查询树的结果关系和命令类型分别等于CREATE RULE命令中给出的对象和事件时，规则系统就会应用更新规则。对于更新规则，规则系统会创建一个查询树列表，初始时该列表为空。动作可以有零个（关键字NOTHING）、一个或多个。为简化说明，我们先看只有一个动作的规则。这个规则可以有条件，也可以没有条件；它可以是INSTEAD，也可以不是。

什么是规则条件？它是一种限制，用来说明何时执行规则动作、何时不执行。这个条件只能引用伪关系NEW和/或OLD，它们基本上表示作为对象给出的那个关系，只是带有特殊含义。

因此，对于只有一个动作的规则，有以下四种情况会产生相应的查询树。

没有条件，也不是INSTEAD

规则动作的查询树，再附加原始查询树的条件

没有条件，但带有INSTEAD

规则动作的查询树，再附加原始查询树的条件

给出了条件，不是INSTEAD

规则动作的查询树，再附加规则条件和原始查询树的条件

给出了条件，有INSTEAD

规则动作的查询树，再附加规则条件和原始查询树的条件；
以及附加了规则条件取反后的原始查询树

最后，如果规则不是 INSTEAD，就把未经更改的原始查询树加入列表。由于只有带条件的INSTEAD 规则已经加入了原始查询树，因此，对于只有一个动作的规则，最终会得到一棵或两棵输出查询树。

对于ON INSERT规则，原始查询（如果未被INSTEAD抑制）会先于规则添加的任何动作执行。这样一来，这些动作就能看到被插入的行。但对ON UPDATE和ON DELETE规则，原始查询会在规则添加的动作之后执行。这就保证了这些动作能够看到将被更新或删除的行；否则，
动作可能什么也做不了，因为它们找不到符合条件的行。

由规则动作生成的查询树会再次送入重写系统，随后可能还会有更多规则被应用，
从而产生更多或更少的查询树。因此，一个规则的动作必须具有不同的命令类型，
或者具有与该规则自身不同的结果关系。否则，这种递归过程就会陷入无限循环。
（规则的递归展开会被检测出来，并作为错误报告。）

pg_rewrite系统目录中动作里的查询树只是模板。由于它们可以引用NEW和OLD的范围表项，
因此在使用前必须进行一些替换。对任何NEW的引用，
都会先在原始查询的目标列表中查找相应项。如果找到了，就用该项的表达式替换该引用。
否则，NEW就与OLD含义相同（对于UPDATE），或者被替换为一个空值（对于INSERT）。
任何对OLD的引用，都会被替换为对结果关系对应范围表项的引用。

在系统完成应用更新规则后，它再应用视图规则到生成的查询树上。
视图无法插入新的更新动作，所以没有必要向视图重写的输出应用更新规则。

34.3.1.1. 第一个规则：逐步分析

假设我们想跟踪shoelace_data关系中sl_avail列的变化。因此，
我们建立一个日志表和一条规则，使其在shoelace_data上执行UPDATE时，
有条件地写入一条日志记录。

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    text,          -- who did it
    log_when   timestamp      -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

现在有人执行了：

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

然后我们查看日志表：

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

这正是我们预期的结果。后台发生的事情如下。解析器创建了查询树：

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

此时存在一条ON UPDATE规则log_shoelace，它带有如下规则条件表达式：

```
NEW.sl_avail <> OLD.sl_avail
```

其动作是：

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*;
```

（这看起来有点奇怪，因为通常你不能写INSERT ... VALUES ... FROM。这里的FROM子句只是为了表明查询树中存在用于*NEW*和*OLD*的范围表项。之所以需要这些项，是为了让INSERT命令查询树中的变量能够引用它们。）

该规则是一条带条件的非INSTEAD规则，
因此规则系统必须返回两棵查询树：修改后的规则动作，以及原始查询树。第 1 步中，原始查询的范围表会并入规则动作的查询树，得到：

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data;
```

第 2 步将规则条件加进去，因此结果集被限制为sl_avail发生变化的行：

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE *NEW*.sl_avail <> *OLD*.sl_avail;
```

（这看起来更奇怪，因为INSERT ... VALUES同样没有WHERE子句，但规划器和执行器处理它并无困难。反正它们本来也需要为INSERT ... SELECT支持相同功能。）

第 3 步把原始查询树的条件加进去，把结果集进一步限制为只有那些原始查询会触及的行：

```
INSERT INTO shoelace_log VALUES (
```

```
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE *NEW*.sl_avail <> *OLD*.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第 4 步把NEW引用替换为原始查询树中的目标列表项，或者替换为结果关系中相应的变量引用：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE 6 <> *OLD*.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第 5 步把OLD引用替换为结果关系引用：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

至此就完成了。由于规则不是INSTEAD，我们还要输出原始查询树。简而言之，
规则系统输出的是一个包含两棵查询树的列表，它们对应于以下语句：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

它们会按这个顺序执行，而这正是该规则想要达到的效果。

上述替换以及附加的条件能够保证：如果原始查询是下面这样，就不会写入任何日志记录：

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

在这种情况下，原始查询树不包含sl_avail的目标列表项，因此NEW.sl_avail
会被shoelace_data.sl_avail替换。于是，规则生成的额外命令是：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

而该条件永远不可能为真。

如果原始查询修改多行，这种机制同样能够正常工作。例如，假设有人发出如下命令：

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

实际上会更新四行（sl1、sl2、sl3和sl4）。但sl3本来就已经是sl_avail = 0。在这种情况下，原始查询树的条件不同，因此规则会产生额外的查询树：

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

这棵查询树必然会插入三条新的日志记录。这完全正确。

到这里就能看出，为什么原始查询树最后执行至关重要。如果先执行UPDATE，那么所有行都已经被设为零，记日志的INSERT就找不到任何满足0 <> shoelace_data.sl_avail的行了。

34.3.2. 与视图的协作

为了防止有人像前面提到的那样对视图关系执行INSERT、UPDATE或DELETE，一种简单的办法是让那些查询树直接被丢弃。因此我们可以创建如下规则：

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

如果现在某人尝试对视图关系shoe执行这些操作，规则系统就会应用这些规则。由于这些规则没有动作而且是INSTEAD，生成的查询树列表将为空；整个查询也就什么都不会做，因为规则系统处理完后已经没有任何内容可供优化或执行。

另一种更完善的做法，是创建一些规则，把查询树重写成在真实表上执行正确操作的查询树。要在视图shoelace上做到这一点，我们创建下列规则：

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data
```

```
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

现在假定商店每隔一段时间就会收到一包鞋带，随附一份很长的部件清单。
但你不想每次都手工更新 shoelace 视图。因此，我们建立两个小表：一个用于插入部件清单中的项目，另一个则采用一个特殊技巧。创建命令如下：

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

现在可以将部件清单中的数据填入 shoelace_arrive 表：

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

快速查看一下当前数据：

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
```

```
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

现在把到货的鞋带入库：

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

然后检查结果：

```
SELECT * FROM shoelace ORDER BY sl_name;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+----------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

从一个`INSERT ... SELECT`到这些结果，中间要经历相当长的一段过程。本章最后再来描述这一查询树转换的细节。首先，看一下解析器的输出：

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

现在应用第一条规则`shoelace_ok_ins`，它会把这一输出转换成：

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

同时会丢弃针对`shoelace_ok`的原始`INSERT`。这个重写后的查询会再次交给规则系统，而接下来应用的规则`shoelace_upd`会生成：

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
```

```
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

这同样是一条INSTEAD规则，因此前一个查询树会被丢弃。注意，这个查询仍然使用视图shoelace。但规则系统尚未完成，因此会继续在其上应用_RETURN规则，于是得到：

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

最后，规则log_shoelace被应用，生成额外的查询树：

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u,
       shoelace_data *OLD*, shoelace_data *NEW*
       shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

到这里，规则系统已经没有更多规则可用，并返回生成的查询树。

因此，我们最终得到两棵查询树，它们等效于以下SQL语句：

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
```

```
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

结果是：来自一个关系的数据被插入到另一个关系中，这个插入被改写为对第三个关系的更新，再被改写为对第四个关系的更新外加在第五个关系中记录该更新，
最后整个过程被化简为两个查询。

这里有个稍显难看的小细节。观察这两个查询会发现，shoelace_data
关系在范围表中出现了两次，而实际上完全可以缩减成一次。规划器不会处理这一点，
因此规则系统为INSERT输出的执行计划会是

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

而省略那个额外的范围表项则会得到

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

这会在日志表中生成完全相同的项。因此，规则系统导致了对shoelace_data
表的一次完全不必要的额外扫描。而在UPDATE中，同样的冗余扫描还会再发生一次。不过，
能让这一切总体上工作起来，已经是一项相当艰巨的工作。

现在我们来做一个关于PostgreSQL规则系统及其威力的最后演示。
假设你向数据库中加入了一些颜色特别的鞋带：

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.
0);
```

我们想建立一个视图，用来检查哪些shoelace项在颜色上与任何鞋子都不匹配。这个视图是：

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

它的输出是：

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

现在希望将没有库存的不匹配鞋带从数据库中删除。为了给 PostgreSQL 增加一点难度，我们不直接删除，而是再创建一个视图：

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

然后这样执行：

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
             WHERE sl_name = shoelace.sl_name);
```

Voilà：

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |        80
 sl2     |        6 | black    |    100 | cm      |       100
 sl7     |        6 | brown    |     60 | cm      |        60
 sl4     |        8 | black    |     40 | inch    |     101.6
 sl3     |       10 | black    |     35 | inch    |      88.9
 sl8     |       21 | brown    |     40 | inch    |     101.6
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
 sl5     |        4 | brown    |      1 | m       |       100
 sl6     |       20 | brown    |    0.9 | m       |        90
(9 rows)
```

作用于一个视图上的`DELETE`，其子查询条件总共使用了四个嵌套/连接的视图，
其中一个视图自身又带有一个包含视图的子查询条件，并且还用到了计算得到的视图列，
最终仍会被重写成单棵查询树，从真正的表中删除所请求的数据。

在现实世界中，大概只有很少的场景会需要这样的构造。但知道它确实能工作，总归让人安心。

34.4. 规则和权限

由于PostgreSQL规则系统会重写查询，因此可能访问原始查询中并未使用的其他表或视图。
使用更新规则时，这甚至可能包括对表的写访问。

重写规则没有单独的所有者。关系（表或视图）
的所有者会自动成为为其定义的重写规则的所有者。
PostgreSQL规则系统改变了默认访问控制系统的行为。

由于规则而使用的关系都会根据规则所有者的权限进行检查，而不是调用规则的用户。这意味着用户只需要对其查询中明确命名的表/视图具有所需的权限。

例如，某个用户有一份电话号码列表，其中一部分是私人的，另一部分对办公室秘书有用。该用户可以这样构造：

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE NOT private;
GRANT SELECT ON phone_number TO secretary;
```

除了该用户本人（以及数据库超级用户）之外，没有人可以访问phone_data表。但由于GRANT的存在，秘书可以对phone_number视图执行SELECT。规则系统会把对phone_number的SELECT重写成对phone_data的SELECT，并附加只选取private为假的项的条件。由于该用户是phone_number的所有者，因此也是规则的所有者，对phone_data的读访问会按照该用户的权限进行检查，于是查询被允许。同时，对phone_number本身的访问检查仍会执行，但这是针对调用用户进行的，因此除了用户本人和秘书之外，没有其他人能使用它。

权限是逐条规则检查的。因此，目前秘书是唯一能看到公开电话号码的人。但秘书还可以再建立一个视图，并把该视图授予公众访问。这样，任何人都可以通过秘书的视图看到phone_number中的数据。秘书做不到的是创建一个直接访问phone_data的视图。（其实秘书可以创建，但它不会起作用，因为每次访问都会在权限检查时被拒绝。）而且，一旦用户发现秘书开放了其phone_number视图，用户就可以撤销秘书的访问权限。那样一来，对秘书视图的任何访问都会立即失败。

这种逐条规则的检查看起来似乎像是一个安全漏洞，但实际上并不是。因为即使不这样工作，秘书也可以建立一个与phone_number具有相同列的表，每天把数据复制进去。那样一来，这些就是秘书自己的数据，秘书同样可以把访问权限授予任何人。GRANT的含义本来就是“我信任你”。如果某个你信任的人做了上面的事，那么该重新考虑这份信任，并使用REVOKE了。

类似的考虑也适用于更新规则。在上一节的示例中，示例数据库中那些表的所有者可以把shoelace视图上的SELECT、INSERT、UPDATE和DELETE权限授予其他用户，但对shoelace_log只授予SELECT权限。写日志记录的规则动作仍会成功执行，因此其他用户可以看到日志记录。但他们不能伪造记录，也不能操纵或删除已有记录。

34.5. 规则和命令状态

PostgreSQL服务器会为收到的每条命令返回一个命令状态字符串，例如INSERT 149592 1。在没有规则参与时，这很简单；但当查询被规则重写时会发生什么呢？

规则对命令状态的影响如下：

- 如果该查询没有无条件的 INSTEAD 规则，那么原始给出的查询就会照常执行，并返回它自身的命令状态。（但请注意，如果存在任何带条件的 INSTEAD 规则，那么这些规则条件的取反会被加到原始查询上。这可能会减少它处理的行数，从而影响报告的状态。）
- 如果该查询存在任意无条件的 INSTEAD 规则，那么原始查询将完全不会执行。在这种情况下，服务器会返回由 INSTEAD 规则（带条件或不带条件）插入的、最后一条且与原始查询具有相同命令类型（INSERT、UPDATE 或 DELETE）的查询的命令状态。如果没有任何规则添加出满足这些条件的查询，那么返回的命令状态会显示原始查询类型，并且把行计数和 OID 字段都置零。

（这一系统是在 PostgreSQL 7.3 中确立的。在更早的版本中，存在规则时命令状态可能会显示不同的结果。）

在第二种情况下，程序员可以通过为所需的INSTEAD
规则指定一个在活动规则中按字母顺序排在最后的规则名，来确保由它设置命令状态，
因为它会最后被应用。

34.6. 规则与触发器

许多能够用触发器完成的事情，同样也可以用PostgreSQL规则系统实现。
不能用规则实现的内容之一是某些约束，尤其是外键。
你可以放置一条带条件的规则：当某列的值没有出现在另一张表中时，把命令重写成NOTHING。
但那样做会悄无声息地丢弃数据，这并不是好主意。如果需要检查值的有效性，
并在值无效时生成错误消息，就必须使用触发器。

另一方面，在视图上因 INSERT 而触发的触发器可以做到与规则相同的事情：把数据放到别处，并抑制向视图本身的插入。但在 UPDATE 或 DELETE 上它就无能为力了，因为视图关系中没有被扫描的真实数据，触发器因而永远不会被调用。这时只有规则才能解决问题。

对于两者都能实现的事情，哪一种更好取决于数据库的使用方式。
触发器会对每个受影响的行触发一次，而规则会修改查询，或生成额外的查询。因此，
如果一条语句会影响很多行，那么发出一条额外命令的规则往往会比触发器更快，
因为触发器必须为每一行调用一次，并反复执行其操作。

下面通过一个例子说明，在一种具体情形中选择规则或触发器会有什么不同。这里有两个表：

```
CREATE TABLE computer (
    hostname        text,    -- indexed
    manufacturer    text     -- indexed
);

CREATE TABLE software (
    software        text,    -- indexed
    hostname        text     -- indexed
);
```

两个表都有数千行，并且 hostname 上的索引都是唯一索引。规则或触发器应实现这样一个约束：从 software 中删除引用了已删除计算机的行。触发器会使用以下命令：

```
DELETE FROM software WHERE hostname = $1;
```

由于从 computer 中删除的每一行都会调用触发器，因此触发器可以为这条命令准备并保存执行计划，并通过参数传入 hostname 值。规则则会写成：

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

现在我们来看不同类型的删除。对于下面这种情况：

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

computer表会通过索引扫描（很快），由触发器发出的命令也会使用索引扫描（同样很快）。
规则产生的额外查询是：

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
```

```
                        AND software.hostname = computer.hostname;
```

由于已经建立了合适的索引，规划器将创建一个计划

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

因此，触发器实现和规则实现之间的速度差异不会太大。

在下一个删除场景中，我们想去掉全部 2000 台hostname以old开头的计算机。有两种命令可以做到这件事。其中一种是：

```
DELETE FROM computer WHERE hostname >= 'old'
                        AND hostname <  'ole'
```

规则添加出来的命令将是：

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.
hostname < 'ole'
                        AND software.hostname = computer.hostname;
```

其计划为：

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

另一个可能的命令是：

```
DELETE FROM computer WHERE hostname ~ '^old';
```

它会让规则所添加的命令得到下面这个执行计划：

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

这说明，当多个条件表达式通过AND组合在一起时，规划器并没有意识到，对computer中hostname的限制同样也可以用于software上的索引扫描，
而在该命令的正则表达式版本里它却能做到。触发器会对需要删除的 2000 台旧计算机中的每一台各调用一次，这意味着对computer做一次索引扫描，并对software做 2000 次索引扫描。规则实现则用两条使用索引的命令完成这件事。不过，在顺序扫描这种情况下，
规则是否仍然更快，还取决于software表的总体大小。即使所有索引块很快都会进入缓存，
通过 SPI 管理器执行来自触发器的 2000 条命令仍然要耗费一些时间。

我们要看的最后一个命令是：

```
DELETE FROM computer WHERE manufacurer = 'bim';
```

同样，这也可能导致从computer中删除很多行。因此，
触发器同样会通过执行器运行很多条命令。规则生成的命令则是：

```
DELETE FROM software WHERE computer.manufacurer = 'bim'
                        AND software.hostname = computer.hostname;
```

这个命令的计划又将是在两个索引扫描上的嵌套循环，只不过使用了computer
上的另一个索引：

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

在上述任何一种情况下，规则系统产生的额外命令都或多或少与该命令影响的行数无关。

总结来说，只有当规则动作导致了规模很大且条件很差的连接，而规划器又无能为力时，
规则才会明显慢于触发器。

第 35 章 触发器

本章介绍如何编写触发器函数。触发器函数可以用 C 或某些可用的过程语言编写。目前还无法用 SQL 语言编写触发器函数。

35.1. 触发器行为概述

触发器可以定义为在 INSERT、UPDATE 或 DELETE 操作之前或之后执行，执行方式既可以是每个被修改的行一次，也可以是每条 SQL 语句一次。一旦触发器事件发生，触发器的函数就会在恰当的时机被调用以处理该事件。

必须先定义触发器函数，之后才能创建触发器本身。
触发器函数必须声明为不接受参数并返回类型 trigger 的函数。（触发器函数通过特殊传入的 TriggerData 结构体接收输入，而不是以普通函数参数的形式。）

创建出合适的触发器函数后，就可以使用CREATE TRIGGER建立触发器。同一个触发器函数可以供多个触发器使用。

触发器函数向调用它的执行器返回一个表行（即类型 HeapTuple 的值）。在操作之前触发的触发器可以作出如下选择：

- 它可以返回一个 NULL 指针，以跳过对当前行的操作（这样该行就不会被插入/更新/删除）。
- 仅对于 INSERT 和 UPDATE 触发器，返回的行会成为将要插入的行，或替换正在更新的行。这使得触发器函数可以修改即将插入或更新的行。

不打算引发上述任何一种行为的 BEFORE 触发器，必须注意把传入的行原样作为结果返回（也就是说，对于 INSERT 和 UPDATE 触发器返回 NEW 行，对于 DELETE 触发器返回 OLD 行）。

对于在操作之后触发的触发器，其返回值会被忽略，因此它们可以返回 NULL。

如果为同一关系上的同一事件定义了多个触发器，它们会按触发器名称的字母顺序触发。对于 BEFORE 触发器，每个触发器返回的、可能已经被修改过的行都会成为下一个触发器的输入。如果任何 BEFORE 触发器返回一个 NULL 指针，则会放弃该操作，并且后续触发器不会再触发。

如果触发器函数执行 SQL 命令，那么这些命令可能会再次触发触发器。这就是所谓的级联触发器。对级联层数没有直接限制。级联还有可能导致同一个触发器被递归调用；例如，一个 INSERT 触发器可能执行一条向同一个表再插入一行的命令，从而导致该 INSERT 触发器再次触发。因此，在这种场景下，避免无限递归是触发器编写者自己的责任。

在定义触发器时，可以为它指定参数。在触发器定义中包含参数，
是为了让需求相似的不同触发器能够调用同一个函数。举例来说，可以有一个通用触发器函数，
它接受两个列名作为参数，把当前用户写入其中一个列，把当前时间戳写入另一个列。
只要编写得当，这个触发器函数就应独立于它所作用的具体表。因此，
同一个函数可用于任何具有适当列的表上的 INSERT 事件，例如自动跟踪某个事务表中记录的创建。如果把它定义成 UPDATE 触发器，还可以用来跟踪最近更新事件。

35.2. 数据更改的可见性

如果你在触发器函数中执行 SQL 命令，而这些命令访问的正是该触发器所针对的表，那么你需要了解数据可见性规则，因为它们决定这些 SQL 命令能否看到引发触发器的数据更改。简而言之：

- 引发触发器的数据更改（插入、更新或删除）自然对 BEFORE 触发器中执行的 SQL 命令不可见，因为它还没有发生。
- 然而，BEFORE 触发器中执行的 SQL 命令会看到同一外层命令中先前处理过的行所做的数据更改的影响。这需要小心，因为这些更改事件的发生顺序一般是不可预测的；一条影响多行的 SQL 命令可能以任意顺序访问各行。
- 当 AFTER 触发器触发时，外层命令所做的所有数据更改都已完成，并对执行的 SQL 命令可见。

关于数据可见性规则的更多信息可以参见第 41.4 节。第 35.4 节中的示例包含对这些规则的一个演示。

35.3. 用 C 编写触发器函数

本节描述触发器函数接口的底层细节。只有用 C 编写触发器函数时才需要这些信息。如果你使用的是更高层次的语言，那么这些细节会替你处理。
每种过程语言的文档说明了如何用该语言编写触发器。

触发器函数必须使用“版本 1”函数管理器接口。

当函数被触发器管理器调用时，它不会被传递任何普通参数，而是被传递一个指向 `TriggerData` 结构体的“上下文”指针。C 函数可以通过执行宏

```
CALLED_AS_TRIGGER(fcinfo)
```

来检查自己是否是被触发器管理器调用的，该宏展开为

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

如果它返回真，那么就可以安全地把 `fcinfo->context` 转换为 `TriggerData *` 类型并使用所指向的 `TriggerData` 结构体。该函数不得修改 `TriggerData` 结构体或它指向的任何数据。

`struct TriggerData` 定义在 `commands/trigger.h` 中：

```
typedef struct TriggerData
{
    NodeTag       type;
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
} TriggerData;
```

其成员的含义如下：

`type`

总是 `T_TriggerData`。

`tg_event`

描述引发该函数调用的事件。你可以使用下列宏来检查 `tg_event`：

TRIGGER_FIRED_BEFORE(tg_event)

如果触发器在操作之前触发则返回真。

TRIGGER_FIRED_AFTER(tg_event)

如果触发器在操作之后触发则返回真。

TRIGGER_FIRED_FOR_ROW(tg_event)

如果触发器针对行级事件触发则返回真。

TRIGGER_FIRED_FOR_STATEMENT(tg_event)

如果触发器针对语句级事件触发则返回真。

TRIGGER_FIRED_BY_INSERT(tg_event)

如果触发器由一条 INSERT 命令引发则返回真。

TRIGGER_FIRED_BY_UPDATE(tg_event)

如果触发器由一条 UPDATE 命令引发则返回真。

TRIGGER_FIRED_BY_DELETE(tg_event)

如果触发器由一条 DELETE 命令引发则返回真。

tg_relation

一个指针，指向描述该触发器所针对的关系的结构体。关于这个结构体的细节请参看 utils/rel.h。其中最令人感兴趣的是 tg_relation->rd_att（关系元组的描述符）和 tg_relation->rd_rel->relname（关系名；其类型不是 char* 而是 NameData；如果你需要该名称的一个副本，可以使用 SPI_getrelname(tg_relation) 来获得一个 char*）。

tg_trigtuple

一个指针，指向引发该触发器的行。这就是正在被插入、更新或删除的行。
如果这个触发器是针对 INSERT 或 DELETE 触发的，那么当你不想（在 INSERT 的情形下）用另一个行替换该行或跳过该操作时，就应当从函数返回它。

tg_newtuple

一个指针，指向该行的新版本（如果触发器是针对 UPDATE 触发的）；如果是针对 INSERT 或 DELETE，则它为 NULL。当事件是一次 UPDATE 而你不想用另一个行替换该行或跳过该操作时，就必须从函数返回它。

tg_trigger

一个指针，指向 Trigger 类型的结构体，它定义在 utils/rel.h 中：

```
typedef struct Trigger
{
    Oid         tgoid;
```

```
        char       *tgname;
        Oid         tgfoid;
        int16       tgtype;
        bool        tgenabled;
        bool        tgisconstraint;
        Oid         tgconstrrelid;
        bool        tgdeferrable;
        bool        tginitdeferred;
        int16       tgnargs;
        int16       tgattr[FUNC_MAX_ARGS];
        char      **tgargs;
    } Trigger;
```

其中 tgname 是触发器的名称，tgnargs 是 tgargs 中参数的数量，tgargs 是一个指针数组，指向 CREATE TRIGGER 语句中指定的参数。其他成员仅供内部使用。

触发器函数必须返回 NULL 或者一个 HeapTuple 指针。如果你不想修改正在被操作的行，请注意根据情况返回 tg_trigtuple 或 tg_newtuple。

35.4. 一个完整的示例

下面是一个非常简单的用 C 编写的触发器函数示例。（用过程语言编写的触发器示例可以在各过程语言的文档中找到。）

函数 trigf 报告表 ttest 中的行数，并且当命令试图向列 x 插入空值时跳过实际的操作。（因此该触发器起到一个非空约束的作用，但不会中止事务。）

首先，表定义：

```
CREATE TABLE ttest (
    x integer
);
```

这是触发器函数的源代码：

```
#include "postgres.h"
#include "executor/spi.h"       /* this is what you need to work with
 SPI */
#include "commands/trigger.h"   /* ... and triggers */

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;
```

```
    /* make sure it's called as a trigger at all */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* tuple to return to executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* check for null values */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connect to SPI manager */
    if ((ret = SPI_connect()) < 0)
        elog(INFO, "trigf (fired %s): SPI_connect returned %d", when,
ret);

    /* get number of rows in table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(NOTICE, "trigf (fired %s): SPI_exec returned %d", when,
ret);

    /* count(*) returns int8, so be careful to convert */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when,
i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
```

```
}
```

编译完源代码后，声明该函数和触发器：

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();
```

现在你可以测试触发器的行为：

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                       ^^^^^^
                             remember what we said about visibility.
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
```

```
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                       ^^^^^^
                             remember what we said about visibility.
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

更多更复杂的示例见 `src/test/regress/regress.c` 和 `contrib/spi`。

第 36 章 过程语言

PostgreSQL 允许用户添加可用于编写函数和过程的新编程语言。这些语言统称为过程语言（PL）。对于用过程语言编写的函数或触发器过程，数据库服务器并不内置关于如何解释函数源文本的知识。相反，这项任务会交给一个了解该语言细节的专门的调用处理器。该调用处理器既可以自行完成解析、语法分析、执行等全部工作，也可以在PostgreSQL与某种现有编程语言实现之间充当“粘合剂”。调用处理器本身是一个特殊的 C 语言函数，被编译进共享对象并按需装载。

编写新过程语言的调用处理器在第 47 章中描述。标准 PostgreSQL 发行版提供了几种过程语言，可用作示例。

36.1. 安装过程语言

过程语言必须在每个要使用它的数据库中“安装”。不过，安装在数据库`template1`中的过程语言会自动在随后创建的所有数据库中可用。因此，数据库管理员可以决定哪些数据库提供哪些语言，并且如果愿意，也可以让某些语言默认可用。

对于标准发行版自带的语言，可以使用程序 `createlang` 来安装语言，而不必手工完成各个细节。例如，要把语言 PL/pgSQL 安装到数据库 `template1`，可以使用

```
createlang plpgsql template1
```

下文所述的手工过程只建议用于安装 `createlang` 不认识的自定义语言。

手工安装过程语言

在数据库中安装过程语言需要三个步骤，且必须由数据库超级用户执行。`createlang` 程序会自动完成步骤 2和步骤 3。

1. 该语言调用处理器的共享对象必须先编译好并安装到合适的库目录中。这与构建和安装含有普通用户定义 C 函数的模块的方法相同；见第 33.7.6 节。

2. 必须用如下命令声明该调用处理器：

   ```
   CREATE FUNCTION handler_function_name()
       RETURNS language_handler
       AS 'path-to-shared-object'
       LANGUAGE C;
   ```

   特殊返回类型`language_handler`会告诉数据库系统，该函数返回的不是某种已定义的SQL数据类型，并且不能在 SQL语句中直接使用。

3. 该 PL 必须用如下命令声明：

   ```
   CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language-name
       HANDLER handler_function_name;
   ```

   可选关键字 `TRUSTED` 指定允许没有超级用户权限的普通数据库用户使用该语言创建函数和触发器过程。由于 PL 函数是在数据库服务器内部执行的，`TRUSTED` 标记只应赋予那些不允许访问数据库服务器内部或文件系统的语言。语言 PL/pgSQL、PL/Tcl 和 PL/Perl 被认为是受信任的；而语言 PL/TclU、PL/PerlU 和 PL/PythonU 旨在提供无限制的功能，因此不应被标记为受信任的。

例 36.1展示了以 PL/pgSQL为例时，手工安装过程是如何进行的。

例 36.1. PL/pgSQL 的手工安装

下面的命令告诉数据库服务器到哪里查找 PL/pgSQL语言调用处理器函数的共享对象：

```
CREATE FUNCTION plpgsql_call_handler() RETURNS language_handler AS
    '$libdir/plpgsql' LANGUAGE C;
```

而命令

```
CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
    HANDLER plpgsql_call_handler;
```

则定义对于语言属性为 `plpgsql` 的函数和触发器过程，将调用先前声明的调用处理器函数。

在默认的 PostgreSQL 安装中，PL/pgSQL 语言的调用处理器会被构建并安装到“库”目录中。如果配置了 Tcl/Tk 支持，PL/Tcl 和 PL/TclU 的调用处理器也会被构建并安装到同一位置。类似地，如果配置了 Perl 支持，PL/Perl 和 PL/PerlU 的调用处理器也会被构建和安装；如果配置了 Python 支持，则会安装 PL/PythonU。

第 37 章 PL/pgSQL - SQL 过程语言

PL/pgSQL 是一种用于 PostgreSQL 数据库系统的可载入过程语言。PL/pgSQL 的设计目标是创建一种具备以下特性的可载入过程语言：

- 可以用于创建函数和触发器过程，
- 为 SQL 语言增加控制结构，
- 可以执行复杂计算，
- 继承所有用户定义的类型、函数和操作符，
- 可以定义为被服务器信任，
- 易于使用。

37.1. 概述

PL/pgSQL 调用处理器在函数第一次被调用时（在每个会话内）解析函数的源文本，并生成一棵内部二进制指令树。这棵指令树完整地翻译了 PL/pgSQL 的语句结构，但函数中使用的各个 SQL 表达式和 SQL 命令并不会立即翻译。

当每个表达式和 SQL 命令在函数中第一次被使用时，PL/pgSQL 解释器会创建一个预备执行计划（使用 SPI 管理器的 `SPI_prepare` 和 `SPI_saveplan` 函数）。之后对该表达式或命令的访问会重用这个预备计划。这样，一个包含条件代码、其中含有许多可能需要执行计划的语句的函数，
只会预备并保存那些在数据库连接的生命周期内真正用到的计划。这可以大大减少为一个 PL/pgSQL 函数中的语句解析和生成执行计划所需的总时间。一个缺点是，某个特定表达式或命令中的错误可能要到执行到函数的那一部分时才会被发现。

一旦 PL/pgSQL 为函数中的某个特定命令生成了执行计划，它就会在数据库连接的生命周期内一直重用该计划。这通常对性能有利，但如果你动态地更改数据库模式，也可能引发一些问题。
例如：

```
CREATE FUNCTION populate() RETURNS integer AS '
DECLARE
    -- declarations
BEGIN
    PERFORM my_function();
END;
' LANGUAGE plpgsql;
```

如果你执行上面的函数，它会在为 `PERFORM` 语句生成的执行计划中引用 `my_function()` 的 OID。之后，如果你删除并重新创建 `my_function()`，那么 `populate()` 将再也找不到 `my_function()` 了。此时你必须重新创建 `populate()`，或者至少启动一个新的数据库会话，使它得以重新编译。避免这一问题的另一种方法是，在更新 `my_function()` 的定义时使用 `CREATE OR REPLACE FUNCTION`（函数被“替换”时，其 OID 不会改变）。

由于 PL/pgSQL 以这种方式保存执行计划，直接出现在 PL/pgSQL 函数中的 SQL 命令在每次执行时都必须引用相同的表和列；也就是说，你不能在 SQL 命令中把参数用作表名或列名。要绕过这一限制，可以用 PL/pgSQL 的 `EXECUTE` 语句构造动态命令——代价是每次执行都要构造一个新的执行计划。

> **注意**
>
> PL/pgSQL 的 `EXECUTE` 语句与 PostgreSQL 服务器支持的 `EXECUTE` 语句无关。服务器的 `EXECUTE` 语句不能在 PL/pgSQL 函数中使用（也不需要）。

除了用户定义类型的输入/输出转换和计算函数之外，凡是可以用 C 语言函数定义的事情，也都可以用 PL/pgSQL 完成。例如，可以创建复杂条件计算函数，之后用它们定义操作符或在索引表达式中使用。

37.1.1. 使用 PL/pgSQL 的优点

SQL 是 PostgreSQL（以及大多数其他关系数据库）用作查询语言的语言。它可移植且易学。但每条 SQL 语句都必须由数据库服务器逐条执行。

这意味着你的客户端应用程序必须把每个查询发送到数据库服务器，等待其处理，接收结果，做一些计算，然后再向服务器发送其他查询。所有这些都会产生进程间通信开销；如果你的客户端与数据库服务器不在同一台机器上，还会产生网络开销。

使用 PL/pgSQL，你可以把一块计算和一系列查询组合在数据库服务器内部，从而既拥有过程语言的能力，又拥有 SQL 的易用性，同时因为省去了整个客户端/服务器通信开销而节省大量时间。这可以带来可观的性能提升。

此外，使用 PL/pgSQL 可以使用 SQL 的所有数据类型、操作符和函数。

37.1.2. 支持的参数和结果数据类型

用 PL/pgSQL 编写的函数可以接受服务器支持的任何标量或数组数据类型作为参数，也可以返回这些类型中任何一种的结果。它们还可以接受或返回任何按名称指定的复合类型（行类型）。也可以把 PL/pgSQL 函数声明为返回 `record`，这表示结果是一个其列由调用查询中的规范确定的行类型，如第 7.2.1.4 节所述。

PL/pgSQL 函数还可以声明为接受和返回多态类型 `anyelement` 和 `anyarray`。多态函数实际处理的数据类型可以随调用而变化，如第 33.2.5 节所述。示例见第 37.4.1 节。

PL/pgSQL 函数还可以声明为返回它们能返回单个实例的任何数据类型的“集合”（即表）。这样的函数通过对结果集中每个想要的元素执行 `RETURN NEXT` 来生成输出。

最后，如果 PL/pgSQL 函数没有有用的返回值，可以声明为返回 `void`。

37.2. PL/pgSQL 开发技巧

开发 PL/pgSQL 的一个好方法是，用你选定的文本编辑器创建函数，同时在另一个窗口用 psql 装载并测试这些函数。如果你这样做，最好用 `CREATE OR REPLACE FUNCTION` 编写函数。这样只需重新装载文件即可更新函数定义。例如：

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS '
          ....
end;
' LANGUAGE plpgsql;
```

在运行 psql 期间，可以用

```
\i filename.sql
```

装载或重新装载这样的函数定义文件，然后立即发出 SQL 命令来测试该函数。

开发 PL/pgSQL 的另一个好方法是使用便于过程语言开发的 GUI 数据库访问工具。PgAccess 就是这样一个工具的例子，尽管还有其他工具。这类工具通常提供便捷特性，例如转义单引号、让重新创建和调试函数变得更容易。

37.2.1. 引号的处理

由于 PL/pgSQL 函数的代码在 `CREATE FUNCTION` 中是作为字符串字面量指定的，函数体内部的单引号必须通过加倍来转义。这有时会导致相当复杂的代码，
尤其是当你在编写一个生成其他函数的函数时，例如第 37.6.4 节中的例子。下表可作为各种情况下所需引号数目的一个概要，或许有用。

1 个引号

用于开始和结束函数体，例如：

```
CREATE FUNCTION foo() RETURNS integer AS '...'
    LANGUAGE plpgsql;
```

在函数体内的任何位置，引号都必须成对出现。

2 个引号

用于函数体内部的字符串字面量，例如：

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

PL/pgSQL 会把第二行看作

```
SELECT * FROM users WHERE f_name='foobar';
```

4 个引号

当你在函数体内部的字符串常量中需要一个单引号时，例如：

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

实际追加到 `a_output` 的值将是：`AND name LIKE 'foobar' AND xyz`。

6 个引号

当函数体内部字符串中的一个单引号与该字符串常量的末尾相邻时，例如：

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

此时追加到 `a_output` 的值将是：`AND name LIKE 'foobar'`。

10 个引号

当你想在字符串常量中得到两个单引号（这需要 8 个引号），并且它们与该字符串常量的末尾相邻（再加 2 个）时。只有当你在编写生成其他函数的函数时才可能需要这样做。例如：

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like '''''''''
    || referrer_keys.key_string || '''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

此时 a_output 的值将是：

```
if v_... like ''...'' then return ''...''; end if;
```

另一种方法是用反斜杠而不是加倍来转义函数体中的引号。用这种方法，你会发现自己在写 \'\' 而不是 '''' 这样的东西。有些人觉得这样更容易跟踪，有些人则不这么认为。

37.3. PL/pgSQL 的结构

PL/pgSQL 是一种块结构语言。函数定义的完整文本必须是一个块。块的定义为：

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END;
```

块中的每个声明和每条语句都以分号结束。

所有关键字和标识符都可以用大小写混合书写。除非加双引号，标识符会被隐式转换为小写。

PL/pgSQL 中有两类注释。双划线（--）开始一个延续到行尾的注释。/* 开始一个延续到下一个 */ 的块注释。块注释不能嵌套，但双划线注释可以包含在块注释中，双划线也可以隐藏块注释定界符 /* 和 */。

块的语句部分中的任何语句都可以是一个子块。子块可用于逻辑分组，
或把变量局部化到一小群语句。

块前导声明区中声明的变量，在每次进入该块时都会被初始化为它们的默认值，
而不是每次函数调用只初始化一次。例如：

```
CREATE FUNCTION somefunc() RETURNS integer AS '
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE ''Quantity here is %'', quantity;  -- Quantity here
 is 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE ''Quantity here is %'', quantity;  -- Quantity
 here is 80
    END;

    RAISE NOTICE ''Quantity here is %'', quantity;  -- Quantity here
 is 50

    RETURN quantity;
END;
```

```
' LANGUAGE plpgsql;
```

重要的是，不要把 PL/pgSQL 中用于语句分组的 BEGIN/END 与事务控制的数据库命令混淆。PL/pgSQL 的 BEGIN/END 只用于分组；它们不开始或结束事务。函数和触发器过程总是在由外层查询建立的事务中执行——它们不能开始或提交事务，因为 PostgreSQL 没有嵌套事务。

37.4. 声明

块中使用的所有变量都必须在该块的声明区中声明。（唯一的例外是，在整数值范围上迭代的 FOR 循环的循环变量会被自动声明为 integer 变量。）

PL/pgSQL 变量可以有任何 SQL 数据类型，例如 integer、varchar 和 char。

下面是一些变量声明的例子：

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

变量声明的一般语法为：

name [CONSTANT] *type* [NOT NULL] [{ DEFAULT | := } *expression*];

如果给出了 DEFAULT 子句，它指定进入块时赋给变量的初始值。如果没有给出 DEFAULT 子句，变量被初始化为 SQL 空值。CONSTANT 选项阻止对变量赋值，使其值在块的持续期间保持不变。如果指定了 NOT NULL，赋予空值会导致运行时错误。所有声明为 NOT NULL 的变量都必须指定一个非空默认值。

默认值在每次进入块时求值。因此，例如把 'now' 赋给一个 timestamp 类型的变量，会使该变量具有当前函数调用的时间，而不是函数预编译时的时间。

例子：

```
quantity integer DEFAULT 32;
url varchar := ''http://mysite.com'';
user_id CONSTANT integer := 10;
```

37.4.1. 函数参数的别名

name ALIAS FOR $*n*;

传递给函数的参数以标识符 $1、$2 等命名。可以选择为 $*n* 参数名声明别名以提高可读性。之后既可以用别名、也可以用数字标识符来引用参数值。一些例子：

```
CREATE FUNCTION sales_tax(real) RETURNS real AS '
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
' LANGUAGE plpgsql;
```

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS '
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- some computations here
END;
' LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(tablename) RETURNS text AS '
DECLARE
    in_t ALIAS FOR $1;
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
' LANGUAGE plpgsql;
```

当 PL/pgSQL 函数的返回类型声明为多态类型（`anyelement` 或 `anyarray`）时，会创建一个特殊参数 `$0`。其数据类型是由实际输入类型推断出的函数实际返回类型（见第 33.2.5 节）。这使函数可以访问自己的实际返回类型，如第 37.4.2 节所示。`$0` 初始化为空值，可以被函数修改，因此如果愿意，可以用它来保存返回值，尽管这不是必需的。`$0` 也可以有别名。例如，下面的函数适用于任何有 + 操作符的数据类型：

```
CREATE FUNCTION add_three_values(anyelement, anyelement, anyelement)
RETURNS anyelement AS '
DECLARE
    result ALIAS FOR $0;
    first ALIAS FOR $1;
    second ALIAS FOR $2;
    third ALIAS FOR $3;
BEGIN
    result := first + second + third;
    RETURN result;
END;
' LANGUAGE plpgsql;
```

37.4.2. 复制类型

variable`%TYPE`

`%TYPE` 提供一个变量或表列的数据类型。可以用它来声明将保存数据库值的变量。例如，假设你的 `users` 表中有一个名为 `user_id` 的列。要声明一个与 `users.user_id` 数据类型相同的变量，可以这样写：

```
user_id users.user_id%TYPE;
```

使用 `%TYPE`，你不需要知道所引用结构的数据类型，而且最重要的是，如果所引用项的数据类型将来发生变化（例如你把 `user_id` 的类型从 `integer` 改为 `real`），可能不需要更改你的函数定义。

`%TYPE` 在多态函数中特别有价值，因为内部变量所需的数据类型可能随每次调用而变化。把 `%TYPE` 应用于函数的参数或结果占位符，即可创建合适的变量。

37.4.3. 行类型

nametable_name%ROWTYPE;
name composite_type_name;

复合类型的变量称为行变量（或行类型变量）。这样的变量可以保存 SELECT 或 FOR 查询结果的一整行，只要该查询的列集与变量声明的类型相匹配。行值的各个字段用通常的点表示法访问，例如 rowvar.field。

通过使用 *table_name*%ROWTYPE 表示法，可以声明一个与现有表或视图的行具有相同类型的行变量；也可以通过给出复合类型的名称来声明。（由于每个表都有一个同名关联复合类型，在 PostgreSQL 中写不写 %ROWTYPE 实际上没有区别。但带 %ROWTYPE 的形式可移植性更好。）

函数的参数可以是复合类型（完整的表行）。这种情况下，对应的标识符 $*n* 将是一个行变量，可以从中选择字段，例如 $1.user_id。

行类型变量中只能访问表行的用户定义列，不能访问 OID 或其他系统列（因为该行可能来自视图）。行类型的字段继承表对于 char(*n*) 等数据类型的字段大小或精度。

下面是使用复合类型的一个例子：

```
CREATE FUNCTION use_two_tables(tablename) RETURNS text AS '
DECLARE
    in_t ALIAS FOR $1;
    use_t table2name%ROWTYPE;
BEGIN
    SELECT * INTO use_t FROM table2name WHERE ... ;
    RETURN in_t.f1 || use_t.f3 || in_t.f5 || use_t.f7;
END;
' LANGUAGE plpgsql;
```

37.4.4. 记录类型

name RECORD;

记录变量类似于行类型变量，但它们没有预定义的结构。它们在 SELECT 或 FOR 命令期间被赋值时，采用被赋行的实际行结构。记录变量的子结构在每次被赋值时都可以改变。这样做的后果是，记录变量在第一次被赋值之前没有子结构，
任何访问其字段的尝试都会引发运行时错误。

注意 RECORD 不是真正的数据类型，只是一个占位符。还应当认识到，当 PL/pgSQL 函数声明为返回 record 类型时，这与记录变量并不完全是同一个概念，尽管这样的函数很可能使用记录变量来保存其结果。两种情况下，编写函数时实际的行结构都未知，但对于返回 record 的函数，实际结构在解析调用查询时确定，而记录变量的行结构可以即时改变。

37.4.5. RENAME

RENAME *oldname* TO *newname*;

使用 RENAME 声明可以更改变量、记录或行的名称。这主要用于在触发器过程中用另一个名字引用 NEW 或 OLD。另见 ALIAS。

例子：

```
RENAME id TO user_id;
```

```
RENAME this_var TO that_var;
```

> **注意**
>
> RENAME 在 PostgreSQL 7.3 中似乎是坏的。修复它的优先级很低，因为 ALIAS 已覆盖 RENAME 的大多数实际用途。

37.5. 表达式

PL/pgSQL 语句中使用的所有表达式都由服务器常规的 SQL 执行器处理。看起来只包含常量的表达式实际上可能需要运行时求值（例如 timestamp 类型的 'now'），因此 PL/pgSQL 解析器无法识别 NULL 关键字之外的真实常量值。所有表达式都通过使用 SPI 管理器执行查询

```
SELECT expression
```

在内部求值。求值时，PL/pgSQL 变量标识符的出现被替换为参数，变量的实际值通过参数数组传递给执行器。这使 SELECT 的查询计划只需预备一次，之后即可重用于后续求值。

PostgreSQL 主解析器所做的求值对常量值的解释有一些副作用。具体来说，下面这两个函数的行为是有差异的：

```
CREATE FUNCTION logfunc1(text) RETURNS timestamp AS '
    DECLARE
        logtxt ALIAS FOR $1;
    BEGIN
        INSERT INTO logtable VALUES (logtxt, ''now'');
        RETURN ''now'';
    END;
' LANGUAGE plpgsql;
```

和

```
CREATE FUNCTION logfunc2(text) RETURNS timestamp AS '
    DECLARE
        logtxt ALIAS FOR $1;
        curtime timestamp;
    BEGIN
        curtime := ''now'';
        INSERT INTO logtable VALUES (logtxt, curtime);
        RETURN curtime;
    END;
' LANGUAGE plpgsql;
```

在 logfunc1 的情况下，PostgreSQL 主解析器在为 INSERT 预备计划时，知道字符串 'now' 应被解释为 timestamp，因为 logtable 的目标列就是该类型。于是它会在此时把它变成一个常量，而这个常量值在该会话的生命周期内 logfunc1 的所有调用中都会被使用。不用说，这并不是程序员想要的。

在 logfunc2 的情况下，PostgreSQL 主解析器不知道 'now' 应变成什么类型，因此它返回一个包含字符串 now 的 text 类型数据值。在随后向局部变量 curtime 赋值期间，PL/pgSQL 解释器通过调用 text_out 和 timestamp_in 函数把这个字符串转换为 timestamp 类型。这样，计算出的时间戳就会如程序员所期望的那样在每次执行时更新。

记录变量的可变性质在这方面带来一个问题。当记录变量的字段被用于表达式或语句中时，
各字段的数据类型在同一表达式的多次调用之间不能改变，
因为表达式将使用首次到达该表达式时出现的数据类型来规划。
在编写处理多个表的事件的触发器过程时要记住这一点。（必要时可以用 `EXECUTE` 绕过这个问题。）

37.6. 基本语句

在本节及随后几节中，我们描述 PL/pgSQL 显式理解的所有语句类型。凡是没有被识别为这些语句类型之一的内容，都被推定为 SQL 命令，并被发送到主数据库引擎执行（在此之后，语句中使用的任何 PL/pgSQL 变量都会被替换）。因此，例如 SQL 命令 `INSERT`、`UPDATE` 和 `DELETE` 可以被视为 PL/pgSQL 的语句，但这里不专门列出。

37.6.1. 赋值

向变量或行/记录字段赋值写作：

```
identifier := expression;
```

如上文所述，这种语句中的表达式通过向主数据库引擎发送 SQL `SELECT` 命令来求值。表达式必须产生单个值。

如果表达式的结果数据类型与变量的数据类型不匹配，或者变量有特定的大小/精度（比如 `char(20)`），结果值将由 PL/pgSQL 解释器使用结果类型的输出函数和变量类型的输入函数隐式转换。注意，如果结果值的字符串形式不被输入函数接受，
这可能潜在地导致输入函数产生运行时错误。

例子：

```
user_id := 20;
tax := subtotal * 0.06;
```

37.6.2. SELECT INTO

产生多列（但只有一行）的 `SELECT` 命令的结果可以赋给一个记录变量、行类型变量或标量变量列表。写法是：

```
SELECT INTO target select_expressions FROM ...;
```

其中 *target* 可以是记录变量、行变量，或由简单变量和记录/行字段组成的逗号分隔列表。*select_expressions* 和命令的其余部分与普通 SQL 中相同。

注意，这与 PostgreSQL 对 `SELECT INTO` 的正常解释大不相同——后者的 `INTO` 目标是一张新建的表。如果你想在 PL/pgSQL 函数内部从 `SELECT` 结果创建表，请使用语法 `CREATE TABLE ... AS SELECT`。

如果用一行或一个变量列表作为目标，选出的值必须与目标的结构完全匹配，
否则会发生运行时错误。当目标是记录变量时，它会自动把自己配置为查询结果列的行类型。

除了 `INTO` 子句之外，`SELECT` 语句与普通 SQL `SELECT` 命令相同，可以使用其全部能力。

如果查询返回零行，会向目标赋予空值。如果查询返回多行，第一行被赋给目标，
其余的都被丢弃。（注意，除非你使用了 `ORDER BY`，否则“第一行”没有明确的定义。）

目前，INTO 子句几乎可以出现在 SELECT 语句的任何位置，但建议像上面描述的那样把它紧放在 SELECT 关键字之后。PL/pgSQL 的未来版本可能对 INTO 子句的位置不那么宽容。

你可以在 SELECT INTO 语句之后立即使用 FOUND 来判断赋值是否成功（即查询是否至少返回了一行）。例如：

```
SELECT INTO myrec * FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION ''employee % not found'', myname;
END IF;
```

要测试记录/行结果是否为空，可以使用 IS NULL 条件。但没有办法知道是否有额外的行被丢弃。下面是一个处理未返回行情况的例子：

```
DECLARE
    users_rec RECORD;
    full_name varchar;
BEGIN
    SELECT INTO users_rec * FROM users WHERE user_id=3;

    IF users_rec.homepage IS NULL THEN
        -- user entered no homepage, return "http://"
        RETURN ''http://'';
    END IF;
END;
```

37.6.3. 执行无结果的表达式或查询

有时人们希望求一个表达式或查询的值但丢弃其结果（通常是因为正在调用一个有有用副作用但没有有用结果值的函数）。在 PL/pgSQL 中要做到这一点，使用 PERFORM 语句：

```
PERFORM query;
```

这会执行 *query*（它必须是一条 SELECT 语句）并丢弃结果。PL/pgSQL 变量照常在查询中被替换。此外，如果查询产生了至少一行，特殊变量 FOUND 会被设置为真；如果没有产生行，则为假。

> **注意**
>
> 人们可能以为不带 INTO 子句的 SELECT 能达到这个效果，但目前唯一被接受的方式是 PERFORM。

一个例子：

```
PERFORM create_mv(''cs_session_page_requests_mv'', my_query);
```

37.6.4. 执行动态命令

很多时候你会想在 PL/pgSQL 函数内部生成动态命令，也就是每次执行时会涉及不同的表或不同数据类型的命令。PL/pgSQL 为命令缓存计划的常规做法在这种场景下行不通。为处理这类问题，提供了 EXECUTE 语句：

```
EXECUTE command-string;
```

其中 *command-string* 是一个产生字符串（类型为 text）的表达式，该字符串包含要执行的命令。这个字符串被按字面送给 SQL 引擎。

特别注意，命令字符串不做 PL/pgSQL 变量替换。变量的值必须在构造命令字符串时插入其中。

处理动态命令时，你将不得不面对 PL/pgSQL 中单引号的转义问题。请参阅第 37.2.1 节中的概述，它可以为你省些力气。

与 PL/pgSQL 中的所有其他命令不同，由 EXECUTE 语句运行的命令不会在会话生命周期内只预备和保存一次。相反，该命令在语句每次运行时都会重新预备。
命令字符串可以在函数内动态创建，以对可变的表和列执行操作。

SELECT 命令的结果会被 EXECUTE 丢弃，而且 SELECT INTO 目前不能在 EXECUTE 中使用。有两种方法可以从动态创建的 SELECT 中提取结果：一是使用第 37.7.4 节中描述的 FOR-IN-EXECUTE 循环形式，二是使用带 OPEN-FOR-EXECUTE 的游标，如第 37.8.2 节中所述。

一个例子：

```
EXECUTE ''UPDATE tbl SET ''
        || quote_ident(colname)
        || '' = ''
        || quote_literal(newvalue)
        || '' WHERE ...'';
```

这个例子展示了函数 quote_ident(text) 和 quote_literal(text) 的使用。为安全起见，包含列和表标识符的变量应传递给函数 quote_ident。包含在构造的命令中应作为字符串字面量出现的值的变量应传递给 quote_literal。两者都会采取适当的步骤，分别返回以双引号或单引号括起的输入文本，并正确转义任何嵌入的特殊字符。

下面是一个大得多的动态命令与 EXECUTE 的例子：

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS integer AS '
DECLARE
    referrer_keys RECORD;  -- declare a generic record to be used in a
 FOR
    a_output varchar(4000);
BEGIN
    a_output := ''CREATE FUNCTION cs_find_referrer_type(varchar,
 varchar, varchar)
                  RETURNS varchar AS ''''
                     DECLARE
                         v_host ALIAS FOR $1;
                         v_domain ALIAS FOR $2;
                         v_url ALIAS FOR $3;
                     BEGIN '';

    -- Notice how we scan through the results of a query in a FOR loop
    -- using the FOR <record> construct.

    FOR referrer_keys IN SELECT * FROM cs_referrer_keys ORDER BY try_
order LOOP
        a_output := a_output || '' IF v_'' || referrer_keys.kind || ''
 LIKE '''''''''
```

```
                || referrer_keys.key_string || '''''''''' THEN RETURN
 ''''''
                || referrer_keys.referrer_type || ''''''; END IF;'';
    END LOOP;

    a_output := a_output || '' RETURN NULL; END; '''' LANGUAGE
 plpgsql;'';

    EXECUTE a_output;
END;
' LANGUAGE plpgsql;
```

37.6.5. 获取结果状态

有几种方法可以确定命令的效果。第一种方法是使用 `GET DIAGNOSTICS` 命令，其形式为：

```
GET DIAGNOSTICS variable = item [ , ... ] ;
```

这个命令允许检索系统状态指示器。每个 *item* 是一个关键字，标识要赋给指定变量的状态值（该变量应具有接收它的正确数据类型）。当前可用的状态项有 `ROW_COUNT`（最近发送到 SQL 引擎的 SQL 命令所处理的行数）和 `RESULT_OID`（最近一条 SQL 命令插入的最后一行的 OID）。注意 `RESULT_OID` 只在 `INSERT` 命令之后有用。

一个例子：

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

确定命令效果的第二种方法是检查名为 `FOUND` 的特殊变量，其类型为 `boolean`。`FOUND` 在每个 PL/pgSQL 函数调用开始时为假。它被下列各类语句设置：

- `SELECT INTO` 语句在返回行时把 `FOUND` 设为真，未返回行时设为假。
- `PERFORM` 语句在产生（并丢弃）行时把 `FOUND` 设为真，未产生行时设为假。
- `UPDATE`、`INSERT` 和 `DELETE` 语句在至少影响一行时把 `FOUND` 设为真，未影响行时设为假。
- `FETCH` 语句在返回行时把 `FOUND` 设为真，未返回行时设为假。
- `FOR` 语句在迭代一次或多次时把 `FOUND` 设为真，否则设为假。这适用于 `FOR` 语句的所有三种变体（整数 `FOR` 循环、记录集 `FOR` 循环和动态记录集 `FOR` 循环）。`FOUND` 只在 `FOR` 循环退出时设置：在循环体执行期间，`FOUND` 不会被 `FOR` 语句修改，尽管循环体内其他语句的执行可能会改变它。

`FOUND` 是一个局部变量；对它的任何更改都只影响当前的 PL/pgSQL 函数。

37.7. 控制结构

控制结构也许是 PL/pgSQL 最有用（也最重要）的部分。使用 PL/pgSQL 的控制结构，你可以非常灵活而强大地操纵 PostgreSQL 数据。

37.7.1. 从函数返回

有两个命令可用于从函数返回数据：`RETURN` 和 `RETURN NEXT`。

37.7.1.1. RETURN

```
RETURN expression;
```

带表达式的 RETURN 终止函数并把 *expression* 的值返回给调用者。这种形式用于不返回集合的 PL/pgSQL 函数。

返回标量类型时，可以使用任何表达式。
表达式的结果会像赋值所描述的那样自动转换为函数的返回类型。要返回复合（行）值，必须把一个记录或行变量写作 *expression*。

函数的返回值不能没有定义。如果控制到达函数顶层块的末尾而没有遇到 RETURN 语句，将发生运行时错误。

如果你已把函数声明为返回 void，仍必须指定 RETURN 语句；但这种情况下 RETURN 后面的表达式是可选的，如果存在也会被忽略。

37.7.1.2. RETURN NEXT

```
RETURN NEXT expression;
```

当 PL/pgSQL 函数被声明为返回 SETOF *sometype* 时，要遵循的过程略有不同。这种情况下，要返回的各个项在 RETURN NEXT 命令中指定，然后用一条不带参数的最终 RETURN 命令表示函数已执行完毕。RETURN NEXT 既可用于标量数据类型，也可用于复合数据类型；后者将返回整张“表”的结果。

使用 RETURN NEXT 的函数应按以下方式调用：

```
SELECT * FROM some_func();
```

也就是说，函数在 FROM 子句中用作表源。

RETURN NEXT 实际上并不从函数返回；它只是保存表达式（或记录或行变量，视返回的数据类型而定）的值。然后执行继续到 PL/pgSQL 函数中的下一条语句。随着相继的 RETURN NEXT 命令被执行，结果集逐渐建立。最后的 RETURN（不应带参数）使控制退出函数。

> **注意**
>
> PL/pgSQL 当前对 RETURN NEXT 的实现是在从函数返回之前保存整个结果集，如上所述。这意味着，如果 PL/pgSQL 函数产生非常大的结果集，性能可能不佳：为避免耗尽内存，数据会被写入磁盘，但函数本身要等到整个结果集生成之后才会返回。PL/pgSQL 的未来版本可能允许用户定义没有这一限制的返回集合的函数。目前，
> 数据开始写入磁盘的时机由 sort_mem 配置变量控制。有足够内存把更大结果集保存在内存中的管理员应考虑增大此参数。

37.7.2. 条件

IF 语句让你根据某些条件执行命令。PL/pgSQL 有四种 IF 形式：

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF

- IF ... THEN ... ELSIF ... THEN ... ELSE

37.7.2.1. IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

IF-THEN 语句是 IF 最简单的形式。条件为真时，THEN 和 END IF 之间的语句将被执行。否则，它们被跳过。

例子：

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

37.7.2.2. IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

IF-THEN-ELSE 语句在 IF-THEN 的基础上增加了对条件求值为假时应执行的另一组语句的指定。

例子：

```
IF parentid IS NULL OR parentid = ''''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || ''/'' || fullname;
END IF;

IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN ''t'';
ELSE
    RETURN ''f'';
END IF;
```

37.7.2.3. IF-THEN-ELSE IF

IF 语句可以嵌套，如下例所示：

```
IF demo_row.sex = ''m'' THEN
    pretty_sex := ''man'';
ELSE
    IF demo_row.sex = ''f'' THEN
        pretty_sex := ''woman'';
    END IF;
END IF;
```

使用这种形式时，你实际上是在外层 IF 语句的 ELSE 部分中嵌套了一个 IF 语句。因此，每个嵌套的 IF 需要一个 END IF 语句，父级 IF-ELSE 也需要一个。这可行，但当要检查的备选很多时就变得乏味。因此有下一种形式。

37.7.2.4. IF-THEN-ELSIF-ELSE

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

IF-THEN-ELSIF-ELSE 提供了在一条语句中检查许多备选的更方便的方法。形式上它等价于嵌套的 IF-THEN-ELSE-IF-THEN 命令，但只需要一个 END IF。

下面是一个例子：

```
IF number = 0 THEN
    result := ''zero'';
ELSIF number > 0 THEN
    result := ''positive'';
ELSIF number < 0 THEN
    result := ''negative'';
ELSE
    -- hmm, the only other possibility is that number is null
    result := ''NULL'';
END IF;
```

37.7.3. 简单循环

使用 LOOP、EXIT、WHILE 和 FOR 语句，你可以让 PL/pgSQL 函数重复执行一系列命令。

37.7.3.1. LOOP

```
[<<label>>]
LOOP
    statements
END LOOP;
```

LOOP 定义一个无限重复的无条件循环，直到被 EXIT 或 RETURN 语句终止。可选的标签可供嵌套循环中的 EXIT 语句用来指定应终止哪一层嵌套。

37.7.3.2. EXIT

```
EXIT [ label ] [ WHEN expression ];
```

如果没有给出 *label*，最内层的循环被终止，接下来执行 END LOOP 之后的语句。如果给出了 *label*，它必须是当前层或某个外层嵌套循环或块的标签。然后命名的循环或块被终止，控制继续到该循环/块对应的 END 之后的语句。

如果存在 WHEN，只有当指定条件为真时才退出循环，否则控制传递到 EXIT 之后的语句。

例子：

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;

LOOP
    -- some computations
    EXIT WHEN count > 0;  -- same result as previous example
END LOOP;

BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT;  -- invalid; cannot use EXIT outside of LOOP
    END IF;
END;
```

37.7.3.3. WHILE

```
[<<label>>]
WHILE expression LOOP
    statements
END LOOP;
```

WHILE 语句在条件表达式求值为真期间重复一个语句序列。条件在每次进入循环体之前检查。

例如：

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
END LOOP;

WHILE NOT boolean_expression LOOP
    -- some computations here
END LOOP;
```

37.7.3.4. FOR（整数变体）

```
[<<label>>]
FOR name IN [ REVERSE ] expression .. expression LOOP
    statements
END LOOP;
```

这种形式的 FOR 创建一个在整数值范围上迭代的循环。变量 *name* 自动定义为 integer 类型，且只存在于循环内部。给出范围下界和上界的两个表达式在进入循环时求值一次。迭代步长通常为 1，指定 REVERSE 时为 -1。

整数 FOR 循环的一些例子：

```
FOR i IN 1..10 LOOP
    -- some computations here
    RAISE NOTICE ''i is %'', i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- some computations here
END LOOP;
```

如果下界大于上界（REVERSE 情况下则是小于），循环体完全不执行。不会引发错误。

37.7.4. 遍历查询结果

使用另一种类型的 FOR 循环，你可以遍历查询的结果并相应地操纵那些数据。语法为：

```
[<<label>>]
FOR record_or_row IN query LOOP
    statements
END LOOP;
```

记录或行变量被相继赋以查询（SELECT 命令）产生的每一行，并对每一行执行循环体。下面是一个例子：

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS '
DECLARE
    mviews RECORD;
BEGIN
    PERFORM cs_log(''Refreshing materialized views...'');

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_
key LOOP

        -- Now "mviews" has one record from cs_materialized_views

        PERFORM cs_log(''Refreshing materialized view '' || quote_
ident(mviews.mv_name) || ''...'');
        EXECUTE ''TRUNCATE TABLE  '' || quote_ident(mviews.mv_name);
        EXECUTE ''INSERT INTO '' || quote_ident(mviews.mv_name) || ''
 '' || mviews.mv_query;
    END LOOP;

    PERFORM cs_log(''Done refreshing materialized views.'');
    RETURN 1;
END;
' LANGUAGE plpgsql;
```

如果循环被 EXIT 语句终止，最后赋的行值在循环之后仍可访问。

FOR-IN-EXECUTE 语句是遍历记录的另一种方式：

```
[<<label>>]
FOR record_or_row IN EXECUTE text_expression LOOP
    statements
END LOOP;
```

这与前面的形式类似，不同之处在于源 SELECT 语句是作为字符串表达式指定的，它在每次进入 FOR 循环时被求值并重新规划。这使程序员可以在预规划查询的速度和动态查询的灵活性之间做出选择，就像普通的 EXECUTE 语句一样。

> **注意**
>
> PL/pgSQL 解析器目前通过检查 FOR 之后提到的目标变量是否被声明为记录或行变量来区分两类 FOR 循环（整数或查询结果）。如果不是，就推定为整数 FOR 循环。当真正的问题比如说只是 FOR 后面的变量名拼错时，这可能导致相当不直观的错误消息。通常报的错类似于 missing ".." at end of SQL expression。

37.8. 游标

不是一次执行整个查询，而是可以建立一个封装该查询的游标，
然后每次读取查询结果的少数几行。这样做的一个原因是避免结果包含大量行时的内存溢出。
（不过，PL/pgSQL 用户通常不需要担心这一点，因为 FOR 循环内部自动使用游标来避免内存问题。）更有趣的用法是返回函数所创建游标的引用，让调用者读取这些行。
这提供了一种从函数返回大型行集的高效方式。

37.8.1. 声明游标变量

PL/pgSQL 中对游标的所有访问都通过游标变量进行，游标变量总是属于特殊数据类型 refcursor。创建游标变量的一种方法是直接把它声明为 refcursor 类型的变量。另一种方法是使用游标声明语法，其一般形式为：

```
name CURSOR [ ( arguments ) ] FOR query ;
```

（为兼容 Oracle，FOR 可以换成 IS。）如果指定了 *arguments*，它是一个由 *namedatatype* 对组成的逗号分隔列表，这些名称在给定查询中将被参数值替换。替换这些名称的实际值将在游标打开时指定。

一些例子：

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 =
 key;
```

这三个变量的数据类型都是 refcursor，但第一个可用于任何查询，第二个已有完全指定的查询绑定于其上，最后一个有一个参数化查询绑定于其上。（打开游标时，key 将被替换为一个整数参数值。）变量 curs1 被称为未绑定的，因为它没有绑定到任何特定查询。

37.8.2. 打开游标

游标可以用来检索行之前，必须先被打开。（这等价于 SQL 命令 DECLARE CURSOR。）PL/pgSQL 有三种形式的 OPEN 语句，其中两种使用未绑定游标变量，第三种使用绑定游标变量。

37.8.2.1. OPEN FOR SELECT

```
OPEN unbound-cursor FOR SELECT ...;
```

游标变量被打开并被给定要执行的指定查询。游标不能已经处于打开状态，
而且它必须被声明为未绑定游标（即简单的 `refcursor` 变量）。`SELECT` 查询的处理方式与 PL/pgSQL 中其他 `SELECT` 语句相同：替换 PL/pgSQL 变量名，并缓存查询计划供可能的重用。

一个例子：

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

37.8.2.2. OPEN FOR EXECUTE

```
OPEN unbound-cursor FOR EXECUTE query-string;
```

游标变量被打开并被给定要执行的指定查询。游标不能已经处于打开状态，
而且它必须被声明为未绑定游标（即简单的 `refcursor` 变量）。查询以与 `EXECUTE` 命令中相同的方式指定为字符串表达式。与通常一样，这提供了灵活性，使查询可以随每次运行而变化。

一个例子：

```
OPEN curs1 FOR EXECUTE ''SELECT * FROM '' || quote_ident($1);
```

37.8.2.3. 打开绑定游标

```
OPEN bound-cursor [ ( argument_values ) ];
```

这种形式的 `OPEN` 用于打开其查询在声明时就绑定于其上的游标变量。游标不能已经处于打开状态。当且仅当游标被声明为接受参数时，必须出现实参值表达式列表。
这些值将在查询中被替换。绑定游标的查询计划总是被视为可缓存的；这种情况没有 `EXECUTE` 的等价物。

例子：

```
OPEN curs2;
OPEN curs3(42);
```

37.8.3. 使用游标

游标一旦被打开，就可以用这里描述的语句操纵它。

这些操纵不必发生在最初打开游标的同一个函数中。你可以从函数中返回 `refcursor` 值，让调用者操作该游标。（在内部，`refcursor` 值只是包含该游标活动查询的所谓门廊的字符串名称。这个名称可以被传递、赋给其他 `refcursor` 变量等等，而不会干扰门廊。）

所有门廊都在事务结束时隐式关闭。因此 `refcursor` 值只在事务结束之前可用于引用打开的游标。

37.8.3.1. FETCH

```
FETCH cursor INTO target;
```

`FETCH` 从游标检索下一行到目标中，目标可以是行变量、记录变量或简单变量的逗号分隔列表，就像 `SELECT INTO` 一样。与 `SELECT INTO` 一样，可以检查特殊变量 `FOUND` 来看是否取得了行。

一个例子：

```
FETCH curs1 INTO rowvar;
```

```
FETCH curs2 INTO foo, bar, baz;
```

37.8.3.2. CLOSE

```
CLOSE cursor;
```

`CLOSE` 关闭打开游标底层的门廊。这可用于在事务结束之前释放资源，或释放游标变量以便再次打开。

一个例子：

```
CLOSE curs1;
```

37.8.3.3. 返回游标

PL/pgSQL 函数可以向调用者返回游标。这对返回多行或多列很有用，尤其是对于非常大的结果集。为此，
函数打开游标并把游标名返回给调用者（或者只是用调用者指定的或以其他方式为调用者所知的门廊名打开游标）。然后调用者可以从游标中获取行。游标可以由调用者关闭，
也可以在事务关闭时自动关闭。

游标使用的门廊名可以由程序员指定，也可以自动生成。要指定门廊名，
只需在打开游标之前把一个字符串赋给 `refcursor` 变量。`OPEN` 会把 `refcursor` 变量的字符串值用作底层门廊的名称。但如果 `refcursor` 变量为空，`OPEN` 会自动生成一个不与任何现有门廊冲突的名称，并把它赋给该 `refcursor` 变量。

> **注意**
>
> 绑定游标变量被初始化为表示其名称的字符串值，因此门廊名与游标变量名相同，
> 除非程序员在打开游标之前通过赋值覆盖它。但未绑定游标变量初始默认为空值，
> 因此除非被覆盖，它会得到一个自动生成的唯一名称。

下面的例子展示了由调用者提供游标名的一种方式：

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

下面的例子使用自动游标名生成：

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
```

```
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

37.9. 错误和消息

使用 RAISE 语句报告消息和引发错误。

```
RAISE level 'format' [, variable [, ...]];
```

可能的级别有 DEBUG、LOG、INFO、NOTICE、WARNING 和 EXCEPTION。EXCEPTION 引发错误并中止当前事务；其他级别只产生不同优先级别的消息。特定优先级的消息是被报告给客户端、写入服务器日志还是两者都做，由 log_min_messages 和 client_min_messages 配置变量控制。更多信息参见第 16.4 节。

在格式字符串内部，% 被下一个可选参数的字符串表示替换。写 %% 可以输出字面的 %。注意，可选参数目前必须是简单变量而不是表达式，格式必须是简单的字符串字面量。

在这个例子中，v_job_id 的值将替换字符串中的 %：

```
RAISE NOTICE ''Calling cs_create_job(%)'', v_job_id;
```

这个例子将以给定的错误消息中止事务：

```
RAISE EXCEPTION ''Inexistent ID --> %'', user_id;
```

PostgreSQL 没有一个非常聪明的异常处理模型。每当解析器、规划器/优化器或执行器断定一条语句无法再处理下去时，整个事务都会被中止，
系统跳回主循环以从客户端应用程序获取下一条命令。

可以挂钩到错误机制中来注意到这种情况的发生。
但目前无法知道到底是什么导致了中止（数据类型格式错误、浮点错误、解析错误等）。
而且此时数据库服务器有可能处于不一致状态，
因此返回上层执行器或发出更多命令可能会损坏整个数据库。

因此，PL/pgSQL 目前在函数或触发器过程执行期间遇到中止时所做的唯一事情，就是向消息添加一些字段，说明错误发生在哪个函数中以及哪里（行号和语句类型）。
该错误总是会停止函数的执行。

37.10. 触发器过程

PL/pgSQL 可用于定义触发器过程。触发器过程用 CREATE FUNCTION 命令创建，声明为无参数、返回类型为 trigger 的函数。注意，即使函数期望接收 CREATE TRIGGER 中指定的参数，也必须声明为无参数—— 触发器参数通过 TG_ARGV 传递，如下所述。

当 PL/pgSQL 函数作为触发器被调用时，顶层块中会自动创建若干特殊变量。它们是：

NEW

数据类型 RECORD；在行级触发器中保存 INSERT/UPDATE 操作的新数据库行的变量。此变量在语句级触发器中为空。

OLD

数据类型 RECORD；在行级触发器中保存 UPDATE/DELETE 操作的旧数据库行的变量。此变量在语句级触发器中为空。

TG_NAME

数据类型 name；包含实际触发的触发器名称的变量。

TG_WHEN

数据类型 text；依据触发器定义为 BEFORE 或 AFTER 的字符串。

TG_LEVEL

数据类型 text；依据触发器定义为 ROW 或 STATEMENT 的字符串。

TG_OP

数据类型 text；取值为 INSERT、UPDATE 或 DELETE 的字符串，说明触发器是针对哪个操作触发的。

TG_RELID

数据类型 oid；导致触发器调用的表的对象 ID。

TG_RELNAME

数据类型 name；导致触发器调用的表的名称。

TG_NARGS

数据类型 integer；CREATE TRIGGER 语句中给触发器过程的参数个数。

TG_ARGV[]

数据类型为 text 的数组；来自 CREATE TRIGGER 语句的参数。索引从 0 计数。无效索引（小于 0 或大于等于 tg_nargs）得到空值。

触发器函数必须返回空值，或返回与触发该触发器的表的结构完全相同的记录/行值。

BEFORE 行级触发器可以返回空值，向触发器管理器示意跳过这一行的其余操作（即不触发后续触发器，并且不对这一行执行 INSERT/UPDATE/DELETE）。如果返回非空值，操作将以该行值继续进行。返回与 NEW 原值不同的行值会改变将被插入或更新的行（但在 DELETE 情况下没有直

接效果）。要更改将被存储的行，可以直接替换 NEW 中的单个值并返回修改后的 NEW，也可以构造一个完整的新记录/ 行来返回。

BEFORE 或 AFTER 语句级触发器以及 AFTER 行级触发器的返回值总是被忽略；不妨返回空值。但这些类型的触发器中的任何一种都仍可以通过引发错误中止整个操作。

例 37.1展示了一个用 PL/pgSQL 编写的触发器过程的例子。

例 37.1. 一个 PL/pgSQL 触发器过程

这个示例触发器确保每当表中有行被插入或更新时，当前用户名和时间都被记入该行。
它还检查给出了雇员姓名，并且薪金为正值。

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS '
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION ''empname cannot be null'';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION ''% cannot have null salary'', NEW.
empname;
        END IF;

        -- Who works for us when she must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION ''% cannot have a negative salary'', NEW.
empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := ''now'';
        NEW.last_user := current_user;
        RETURN NEW;
    END;
' LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

37.11. 从 Oracle PL/SQL 移植

本节解释 PostgreSQL 的 PL/pgSQL 语言与 Oracle 的 PL/SQL 语言之间的差异，以帮助把应用程序从 Oracle 移植到 PostgreSQL 的开发者。

PL/pgSQL 在许多方面与 PL/SQL 相似。它是一种块结构的命令式语言，所有变量都必须声明。赋值、循环、条件都相似。从 PL/SQL 移植到 PL/pgSQL 时要记住的主要差异是：

- PostgreSQL 中参数没有默认值。
- 在 PostgreSQL 中可以重载函数名。这常被用来绕开缺少默认参数的问题。
- PL/pgSQL 中不需要游标，把查询放进 `FOR` 语句即可。（见例 37.3。）
- 在 PostgreSQL 中，需要转义函数体中的单引号。见第 37.2.1 节。
- 不要用包（package），改用模式（schema）来把函数组织成组。

37.11.1. 移植示例

例 37.2展示如何把一个简单函数从 PL/SQL 移植到 PL/pgSQL。

例 37.2. 把简单函数从 PL/SQL 移植到 PL/pgSQL

下面是一个 Oracle PL/SQL 函数：

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name IN varchar,
 v_version IN varchar)
RETURN varchar IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

让我们过一遍这个函数，看看它与 PL/pgSQL 的差异：

- PostgreSQL 没有命名参数。你必须在函数内部显式地为它们建立别名。
- Oracle 的函数可以有 `IN`、`OUT` 和 `INOUT` 参数。例如 `INOUT` 表示参数将接收一个值并返回另一个值。PostgreSQL 只有 `IN` 参数。
- 函数原型（不是函数体）中的 `RETURN` 关键字在 PostgreSQL 中变为 `RETURNS`。
- 在 PostgreSQL 中，函数是用单引号作为函数体的定界符创建的，因此必须转义函数体内部的引号。
- PostgreSQL 中不存在 `/show errors` 命令。

这个函数移植到 PostgreSQL 后是这个样子：

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(varchar, varchar)
RETURNS varchar AS '
DECLARE
    v_name ALIAS FOR $1;
    v_version ALIAS FOR $2;
BEGIN
    IF v_version IS NULL THEN
        return v_name;
    END IF;
    RETURN v_name || ''/'' || v_version;
END;
```

```
' LANGUAGE plpgsql;
```

例 37.3展示如何移植一个创建另一个函数的函数，以及如何处理随之而来的引号问题。

例 37.3. 把创建另一函数的函数从 PL/SQL 移植到 PL/pgSQL

下面的过程从 `SELECT` 语句抓取行，并把结果用 `IF` 语句构造成一个大函数，以求高效。请特别注意游标和 `FOR` 循环中的差异，

这是 Oracle 版本：

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;

    a_output VARCHAR(4000);
BEGIN
    a_output := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_
host IN VARCHAR, v_domain IN VARCHAR,
v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        a_output := a_output || ' IF v_' || referrer_key.kind || '
 LIKE ''' ||
referrer_key.key_string || ''' THEN RETURN ''' || referrer_key.
referrer_type ||
'''; END IF;';
    END LOOP;

    a_output := a_output || ' RETURN NULL; END;';
    EXECUTE IMMEDIATE a_output;
END;
/
show errors;
```

下面是这个函数在 PostgreSQL 中的最终样子：

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS integer AS '
DECLARE
    referrer_keys RECORD;  -- Declare a generic record to be used in a
 FOR
    a_output varchar(4000);
BEGIN
    a_output := ''CREATE FUNCTION cs_find_referrer_type(varchar,
 varchar, varchar)
                  RETURNS varchar AS ''''
                     DECLARE
                         v_host ALIAS FOR $1;
                         v_domain ALIAS FOR $2;
                         v_url ALIAS FOR $3;
                     BEGIN '';

    -- Notice how we scan through the results of a query in a FOR loop
    -- using the FOR <record> construct.
```

```
    FOR referrer_keys IN SELECT * FROM cs_referrer_keys ORDER BY try_
order LOOP
        a_output := a_output || '' IF v_'' || referrer_keys.kind || ''
 LIKE '''''''''
                || referrer_keys.key_string || ''''''''' THEN RETURN
 ''''''
                || referrer_keys.referrer_type || ''''''; END IF;'';
    END LOOP;

    a_output := a_output || '' RETURN NULL; END; '''' LANGUAGE
 plpgsql;'';

    -- EXECUTE will work because we are not substituting any
 variables.
    -- Otherwise it would fail.  Look at PERFORM for another way to
 run functions.

    EXECUTE a_output;
END;
' LANGUAGE plpgsql;
```

例 37.4展示如何移植一个带 `OUT` 参数和字符串处理的过程。PostgreSQL 没有 `instr` 函数，但可以用其他函数组合来绕过。第 37.11.3 节中有一个 PL/pgSQL 实现的 `instr`，可以用来简化你的移植工作。

例 37.4. 把带字符串处理和 OUT 参数的过程从 PL/SQL 移植到 PL/pgSQL

下面的 Oracle PL/SQL 过程用于解析 URL 并返回若干元素（主机、路径和查询）。PL/pgSQL 函数只能返回一个值。在 PostgreSQL 中，绕过这个问题的一种办法是把过程拆成三个不同的函数：一个返回主机，一个返回路径，一个返回查询。

这是 Oracle 版本：

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- This will be passed back
    v_path OUT VARCHAR,  -- This one too
    v_query OUT VARCHAR) -- And this one
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
```

```
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

下面是返回主机部分的 PL/pgSQL 函数可能的样子：

```
CREATE OR REPLACE FUNCTION cs_parse_url_host(varchar) RETURNS varchar
 AS '
DECLARE
    v_url ALIAS FOR $1;
    v_host varchar;
    v_path varchar;
    a_pos1 integer;
    a_pos2 integer;
    a_pos3 integer;
BEGIN
    v_host := NULL;
    a_pos1 := instr(v_url, ''//'');

    IF a_pos1 = 0 THEN
        RETURN '''';  -- Return a blank
    END IF;

    a_pos2 := instr(v_url,''/'',a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := ''/'';
        RETURN v_host;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2 );
    RETURN v_host;
END;
' LANGUAGE plpgsql;
```

例 37.5展示如何移植一个使用许多 Oracle 特有特性的过程。

例 37.5. 把过程从 PL/SQL 移植到 PL/pgSQL

Oracle 版本：

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;❶
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;❷

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_
stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock❸
        raise_application_error(-20000, 'Unable to create a new job: a
 job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id,
 sysdate);
        EXCEPTION WHEN dup_val_on_index THEN NULL; -- don't worry if
 it already exists❹
    END;
    COMMIT;
END;
/
show errors
```

这样的过程可以很容易地转换为返回 `integer` 的 PostgreSQL 函数。这个过程特别有意思，因为它能教给我们一些东西：

❶ PostgreSQL 中没有 `PRAGMA` 语句。
❷ 如果你在 PL/pgSQL 中执行 `LOCK TABLE`，该锁在调用事务结束之前不会被释放。
❸ PL/pgSQL 函数中也不能有事务。整个函数（以及其中调用的其他函数）在一个事务中执行，如果出了问题，PostgreSQL 会回滚该事务。
❹ exception when 需要换成 `IF` 语句。

下面是我们把这个过程移植到 PL/pgSQL 的样子：

```
CREATE OR REPLACE FUNCTION cs_create_job(integer) RETURNS integer AS '
DECLARE
    v_job_id ALIAS FOR $1;
    a_running_job_count integer;
    a_num integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;
    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_
stamp IS NULL;

    IF a_running_job_count > 0
    THEN
        RAISE EXCEPTION ''Unable to create a new job: a job is
 currently running.'';
    END IF;
```

```
    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    SELECT count(*) INTO a_num FROM cs_jobs WHERE job_id=v_job_id;
    IF NOT FOUND THEN  -- If nothing was returned in the last query
        -- This job is not in the table so lets insert it.
        INSERT INTO cs_jobs(job_id, start_stamp) VALUES (v_job_id,
 current_timestamp);
        RETURN 1;
    ELSE
        RAISE NOTICE ''Job already running.'';1
    END IF;

    RETURN 0;
END;
' LANGUAGE plpgsql;
```

1 注意你可以在 PL/pgSQL 中引发通知（或错误）。

37.11.2. 其他需要注意的事项

本节说明把 Oracle PL/SQL 函数移植到 PostgreSQL 时还有一些要注意的事项。

37.11.2.1. EXECUTE

PL/pgSQL 版的 `EXECUTE` 与 PL/SQL 版工作方式类似，但必须记得按第 37.6.4 节所述使用 `quote_literal(text)` 和 `quote_string(text)`。`EXECUTE ''SELECT * FROM $1'';` 这类构造除非使用这些函数，否则不会工作。

37.11.2.2. 优化 PL/pgSQL 函数

PostgreSQL 提供两个函数创建修饰符用于优化执行：易变性（函数在给定相同参数时是否总是返回相同结果）和“严格性”（函数在有参数为空时是否返回空）。细节请查阅 `CREATE FUNCTION` 的描述。

要利用这些优化属性，你的 `CREATE FUNCTION` 语句可以是下面这个样子：

```
CREATE FUNCTION foo(...) RETURNS integer AS '
...
' LANGUAGE plpgsql STRICT IMMUTABLE;
```

37.11.3. 附录

本节包含一个 Oracle 兼容的 `instr` 函数的代码，你可以用它来简化移植工作。

```
--
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1, string2, [n], [m]) where [] denotes optional
 parameters.
--
-- Searches string1 beginning at the nth character for the mth
 occurrence
```

```
-- of string2.  If n is negative, search backwards.  If m is not
 passed,
-- assume 1 (search starts at first character).
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS '
DECLARE
    pos integer;
BEGIN
    pos:= instr($1, $2, 1);
    RETURN pos;
END;
' LANGUAGE plpgsql;


CREATE FUNCTION instr(varchar, varchar, varchar) RETURNS integer AS '
DECLARE
    string ALIAS FOR $1;
    string_to_search ALIAS FOR $2;
    beg_index ALIAS FOR $3;
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
```

```
' LANGUAGE plpgsql;


CREATE FUNCTION instr(varchar, varchar, integer, integer) RETURNS
 integer AS '
DECLARE
    string ALIAS FOR $1;
    string_to_search ALIAS FOR $2;
    beg_index ALIAS FOR $3;
    occur_index ALIAS FOR $4;
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                occur_number := occur_number + 1;

                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
```

```
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
' LANGUAGE plpgsql;
```

第 38 章 PL/Tcl - Tcl 过程语言

PL/Tcl 是 PostgreSQL 数据库系统的一种可加载过程语言，可以使用 Tcl 语言编写函数和触发器过程。

38.1. 概述

除某些限制外，PL/Tcl 提供了函数编写者在 C 语言中所具备的大部分能力。

一个良性的限制是，所有内容都在一个安全的 Tcl 解释器中执行。除了安全 Tcl 受限的命令集之外，只提供了少数通过 SPI 访问数据库以及通过`elog()`发出消息的命令。没有办法访问数据库服务器的内部，也无法像 C 函数那样在 PostgreSQL 服务器进程的权限下获得操作系统级访问。因此，任何非特权数据库用户都可以被允许使用这种语言。

另一个实现上的限制是，Tcl 函数不能用来为新数据类型创建输入/输出函数。

有时需要编写不受安全 Tcl 限制的 Tcl 函数。例如，可能希望某个 Tcl 函数能够发送电子邮件。为处理这类情况，PL/Tcl 有一个叫做 `PL/TclU` 的变体（表示不受信任的 Tcl）。除了使用完整的 Tcl 解释器之外，它与前者是完全相同的语言。如果使用 PL/TclU，就必须把它安装为一种不受信任的过程语言，这样就只有数据库超级用户才能在其中创建函数。PL/TclU 函数的编写者必须注意确保该函数不会被用来做任何不希望发生的事情，因为它能够执行任何以数据库管理员身份登录的用户所能做的事情。

如果在安装过程的配置步骤中指定了 Tcl/Tk 支持，则 PL/Tcl 和 PL/TclU 调用处理器的共享对象代码会自动构建并安装到 PostgreSQL 的库目录中。要在某个特定数据库中安装 PL/Tcl 和/或 PL/TclU，请使用 `createlang` 程序，例如 `createlang pltcl` *`dbname`* 或 `createlang pltclu` *`dbname`*。

38.2. PL/Tcl 函数和参数

要用 PL/Tcl 语言创建函数，可使用标准语法：

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS '
    # PL/Tcl function body
' LANGUAGE pltcl;
```

PL/TclU的写法相同，只是语言必须指定为`pltclu`。

函数体就是一段 Tcl 脚本。调用函数时，参数值以变量 `$1` ... `$n` 的形式传入 Tcl 脚本。Tcl 代码按通常方式使用 `return` 语句返回结果。

例如，一个返回两个整数值中较大值的函数可以定义为：

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS '
    if {$1 > $2} {return $1}
    return $2
' LANGUAGE pltcl STRICT;
```

注意`STRICT`子句，它让我们不必考虑空值输入：如果传入的是空值，函数根本不会被调用，而是会自动返回空值结果。

在非严格函数中，如果某个参数的实际值为空值，对应的`$n`变量会被设置为空串。
要检测某个特定参数是否为空值，可使用函数`argisnull`。例如，假设我们希望在`tcl_max`的两个参数中一个为空值、一个非空值时返回非空值参数，而不是返回空值：

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS '
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
' LANGUAGE pltcl;
```

如上所示，要从 PL/Tcl 函数返回空值，请执行`return_null`。无论函数是否为严格函数，都可以这样做。

复合类型参数会作为 Tcl 数组传递给函数。数组元素名就是该复合类型的属性名。如果传入行中的某个属性为空值，它就不会出现在数组中。下面是一个示例：

```
CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS '
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
' LANGUAGE pltcl;
```

目前不支持返回复合类型的结果值。

38.3. PL/Tcl 中的数据值

提供给 PL/Tcl 函数代码的参数值，只是将输入参数转换成文本形式（就像用`SELECT`语句将它们显示出来一样）。反过来，`return`命令会接受任何字符串，
只要它是该函数声明返回类型的可接受输入格式。因此，在 PL/Tcl 函数内部，所有值都只是文本字符串。

38.4. PL/Tcl 中的全局数据

有时需要在一次函数调用结束后保留某些全局数据，供下一次调用继续使用，
或者在不同函数之间共享全局数据。在 PL/Tcl 中这很容易做到，但必须理解其中的一些限制。

出于安全原因，PL/Tcl 会针对每个 SQL 角色，在该角色各自独立的 Tcl 解释器中执行其调用的函数。这样可以防止一个用户无意或恶意地干扰另一用户的 PL/Tcl 函数行为。每个这样的解释器都为任何 “global” Tcl 变量维护各自的值。因此，当且仅当两个 PL/Tcl 函数由同一个 SQL 角色执行时，它们才会共享同一组全局变量。在某些应用中，一个会话会在多个 SQL 角色下执行代码（例如通过`SECURITY DEFINER`函数、使用`SET ROLE`等），这时你可能需要显式采取一些步骤，确保 PL/Tcl 函数能够共享数据。为此，应确保需要互相通信的函数由同一用户拥有，并将它们标记为`SECURITY DEFINER`。当然，你必须小心，确保这类函数不会被用来执行任何非预期操作。

一个会话中使用的所有 PL/TclU 函数都在同一个 Tcl 解释器中执行，而这个解释器当然不同于 PL/Tcl 函数所使用的解释器。因此，PL/TclU 函数之间会自动共享全局数据。这不被视为安全风险，因为所有 PL/TclU 函数都在相同的信任级别上执行，也就是数据库超级用户的级别。

为帮助防止 PL/Tcl 函数无意间彼此干扰，每个函数都可通过`upvar`命令访问一个全局数组。该变量的全局名称是函数的内部名称，局部名称是`GD`。建议将`GD`用于保存函数的持久私有数据。只有当你明确希望某些值在多个函数之间共享时，才应使用常规的 Tcl 全局变量。（注意`GD`数组只在某个特定解释器内部是全局的，因此它们不会绕过上文提到的安全限制。）

下面`spi_execp`的示例展示了`GD`的用法。

38.5. 从 PL/Tcl 访问数据库

在 PL/Tcl 函数体中可以使用下列命令来访问数据库：

`spi_exec ?-count` *`n`*`? ?-array` *`name`*`?` *`command`* `?`*`loop-body`*`?`

执行以字符串形式给出的 SQL 命令。命令出错时会引发错误。否则，`spi_exec` 的返回值是该命令处理的行数（选出、插入、更新或删除的行），如果命令是工具语句则返回零。此外，如果命令是 `SELECT` 语句，则所选列的值会按下文所述放入 Tcl 变量中。

可选的 `-count` 值告诉 `spi_exec` 此命令最多处理多少行，其效果类似于将查询设为游标后执行 `FETCH` *`n`*。

如果命令是 `SELECT` 语句，则结果列的值会放入以列名命名的 Tcl 变量中。如果给定了 `-array` 选项，则列值会存储在指定的关联数组元素中，列名用作数组索引。

如果命令是 `SELECT` 语句且未给出 *`loop-body`* 脚本，则只会把结果的第一行存入 Tcl 变量中；其余行如果存在，会被忽略。如果查询没有返回任何行，则不会进行任何存储。（这种情况可以通过检查 `spi_exec` 的结果来发现。）例如：

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

会把 Tcl 变量 `$cnt` 设置为 `pg_proc` 系统目录中的行数。

如果给出了可选的 *`loop-body`* 参数，它就是一段 Tcl 脚本，对查询结果中的每一行执行一次。（如果给定的命令不是 `SELECT`，则忽略 *`loop-body`*。）在每次迭代之前，当前行各列的值会被存入 Tcl 变量。例如：

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

会为 `pg_class` 的每一行打印一条日志消息。这一特性的工作方式类似于其他 Tcl 循环构造；特别是 `continue` 和 `break` 在循环体中按通常方式工作。

如果查询结果的某一列为空值，则其目标变量会被“unset”（取消设置），而不是被设置。

`spi_prepare`*`querytypelist`*

准备并保存一个查询计划以供后续执行。保存的计划会在当前会话的整个生命周期内保留。

查询可以使用参数，也就是在实际执行该计划时要提供的值的占位符。在查询字符串中，用符号 `$1` ... `$`*`n`* 引用参数。如果查询使用了参数，则必须以 Tcl 列表的形式给出各参数类型的名称。（如果未使用参数，则为 *`typelist`* 写一个空列表。）目前，参数类型必须用系统表 `pg_type` 中所示的内部类型名标识；例如用 `int4` 而不是 `integer`。

spi_prepare 的返回值是一个查询 ID，供后续调用 spi_execp 时使用。示例参见 spi_execp。

spi_execp ?-count *n*? ?-array *name*? ?-nulls *string*? *queryid* ?*value-list*? ?*loop-body*?

执行先前用 spi_prepare 准备好的查询。*queryid* 是 spi_prepare 返回的 ID。如果查询引用了参数，则必须提供 *value-list*。这是参数实际值构成的 Tcl 列表。该列表的长度必须与先前提供给 spi_prepare 的参数类型列表相同。如果查询没有参数，则省略 *value-list*。

可选的 -nulls 值是由空格和 'n' 字符组成的字符串，用来告诉 spi_execp 哪些参数是空值。如果给出，它的长度必须与 *value-list* 完全相同。如果未给出，则所有参数值都视为非空值。

除了指定查询及其参数的方式之外，spi_execp 的工作方式与 spi_exec 完全相同。-count、-array 和 *loop-body* 选项的含义相同，结果值也相同。

下面是使用已准备计划的 PL/Tcl 函数示例：

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS '
    if {![ info exists GD(plan) ]} {
        # prepare the saved plan on the first call
        set GD(plan) [ spi_prepare \\
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \\$1
 AND num <= \\$2" \\
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
' LANGUAGE pltcl;
```

注意，在输入函数时，Tcl 应看到的每个反斜杠都必须写成双份，因为主解析器在 CREATE FUNCTION 中也会处理反斜杠。我们需要在传给 spi_prepare 的查询字符串中加入反斜杠，以确保 $*n* 标记会原样传递给 spi_prepare，而不会被 Tcl 执行变量替换。

spi_lastoid

如果最后一次 spi_exec 或 spi_execp 执行的是单行 INSERT，则返回所插入行的 OID。（否则返回零。）

quote*string*

将给定字符串中的所有单引号和反斜杠字符都加倍。这可用于安全地为那些要插入到传给 spi_exec 或 spi_prepare 的 SQL 命令中的字符串加引号。例如，考虑如下 SQL 命令字符串：

```
"SELECT '$val' AS ret"
```

其中 Tcl 变量 val 的实际内容是 doesn't。这会得到最终命令字符串：

```
SELECT 'doesn't' AS ret
```

这会在 spi_exec 或 spi_prepare 期间导致解析错误。要使其正常工作，提交的命令应当包含：

```
SELECT 'doesn''t' AS ret
```

它在 PL/Tcl 中可以这样构成：

```
"SELECT '[ quote $val ]' AS ret"
```

`spi_execp` 的一个优点是，你不必像这样为参数值加引号，因为参数永远不会被当作 SQL 命令字符串的一部分来解析。

`elog` *`levelmsg`*

发出日志或错误消息。可用级别包括 `DEBUG`、`LOG`、`INFO`、`NOTICE`、`WARNING`、`ERROR` 和 `FATAL`。大多数级别只是像 `elog` C 函数那样发出给定的消息。`ERROR` 会引发错误条件：放弃函数的进一步执行，并中止当前事务。`FATAL` 会中止事务并导致当前会话关闭。（在 PL/Tcl 函数中使用这个错误级别可能并没有什么充分理由，但为了完整性仍提供它。）

38.6. PL/Tcl 中的触发器过程

可以用 PL/Tcl 编写触发器过程。PostgreSQL 要求作为触发器调用的函数必须声明为无参数、返回类型为 `trigger` 的函数。

触发器管理器的信息通过下列变量传入过程体：

`$TG_name`

`CREATE TRIGGER` 语句中触发器的名称。

`$TG_relid`

导致触发器过程被调用的表的对象 ID。

`$TG_relatts`

表列名构成的 Tcl 列表，其前面带有一个空列表元素。因此，使用 Tcl 的 `lsearch` 命令在该列表中查找列名时，返回的元素编号会从 1 开始表示第一列，这与 PostgreSQL 中列的惯常编号方式一致。（空列表元素也会出现在已删除列的位置上，这样其右侧列的属性编号仍然正确。）

`$TG_when`

字符串 `BEFORE` 或 `AFTER`，取决于触发调用的类型。

`$TG_level`

字符串 `ROW` 或 `STATEMENT`，取决于触发调用的类型。

`$TG_op`

字符串 `INSERT`、`UPDATE` 或 `DELETE`，取决于触发调用的类型。

`$NEW`

对于 `INSERT` 或 `UPDATE` 动作，这是一个包含新表行值的关联数组；对于 `DELETE` 则为空。该数组以列名为索引。值为空值的列不会出现在数组中。对于语句级触发器，不会设置这个变量。

$OLD

对于 UPDATE 或 DELETE 动作，这是一个包含旧表行值的关联数组；对于 INSERT 则为空。该数组以列名为索引。值为空值的列不会出现在数组中。对于语句级触发器，不会设置这个变量。

$args

一个 Tcl 列表，包含 CREATE TRIGGER 语句中给出的过程参数。这些参数也可以在过程体中以 $1 ... $*n* 的形式访问。

触发器过程的返回值可以是字符串 OK 或 SKIP，也可以是 array get Tcl 命令返回的列表。如果返回值为 OK，触发该触发的操作（INSERT/UPDATE/DELETE）将正常进行。SKIP 告诉触发器管理器静默地取消针对该行的操作。如果返回的是列表，则告诉 PL/Tcl 向触发器管理器返回一个修改后的行，用该行代替 $NEW 中给出的行被插入。（这只对 INSERT 和 UPDATE 有效。）不用说，所有这些只在触发器是 BEFORE 且为 FOR EACH ROW 时才有意义；否则返回值会被忽略。

下面是一个简单的触发器过程示例，它强制用表中的一个整数值记录对该行执行的更新次数。对于新插入的行，该值被初始化为 0，之后每次更新操作都将其递增。

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS '
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
' LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

注意，触发器过程本身并不知道列名；列名由触发器参数提供。
这使得该触发器过程可以在不同的表中复用。

38.7. 模块和 unknown 命令

PL/Tcl 支持在使用时自动装载 Tcl 代码。它会识别一张特殊的表 pltcl_modules，该表被假定包含若干 Tcl 代码模块。如果这张表存在，模块unknown会从该表取出，并在数据库会话中第一次执行 PL/Tcl 函数之前立即装载到 Tcl 解释器中。（如果一个会话中使用了多个 Tcl 解释器，则对每个解释器分别执行这一过程；参见第 38.4 节。）

虽然unknown模块实际上可以包含你需要的任何初始化脚本，但它通常会定义一个 Tcl 的unknown过程，每当 Tcl 无法识别被调用的过程名时，就会调用该过程。PL/Tcl的这一过程的标准版本会尝试在pltcl_modules中找到一个能够定义所需过程的模块。如果找到了，

就把它装载到解释器中，然后允许继续执行最初尝试的过程调用。辅助表pltcl_modfuncs 提供了哪个模块定义了哪些函数的索引，使查找相当快速。

PostgreSQL发行版中包含用于维护这些表的支持脚本：pltcl_loadmod、pltcl_listmod、pltcl_delmod，以及标准unknown模块的源代码，位于share/unknown.pltcl。要支持自动装载机制，最初必须把这个模块装载到每个数据库中。

表pltcl_modules和pltcl_modfuncs 必须对所有用户可读，但最好只让数据库管理员拥有并可以写入它们。作为一种安全预防措施，除非pltcl_modules由超级用户拥有，否则 PL/Tcl 会忽略它（因而不尝试装载unknown模块）。不过，如果你足够信任其他用户，可以把这张表的 UPDATE 权限授予他们。

38.8. Tcl 过程名

在PostgreSQL中，只要参数个数或参数类型不同，就可以复用同一个函数名。不过，Tcl 要求所有过程名都必须不同。PL/Tcl 处理这一问题的方式是：在内部 Tcl 过程名中包含系统表 pg_proc 中的函数对象 ID 作为其名称的一部分。因此，名称相同但参数类型不同的PostgreSQL函数，也会对应不同的 Tcl 过程。这通常不是 PL/Tcl 程序员需要关心的事情，但在调试时可能会看见。

第 39 章 PL/Perl - Perl 过程语言

PL/Perl 是一种可装载的过程语言，使你能够用 Perl[1] 编程语言编写 PostgreSQL 函数。

要在某个特定数据库中安装 PL/Perl，使用 `createlang plperl dbname`。

> **提示**
>
> 如果把某种语言安装到`template1`中，之后创建的所有数据库都会自动安装该语言。

> **注意**
>
> 源码包的用户必须在安装过程中专门启用 PL/Perl 的构建。（更多信息请参阅安装说明。）二进制包的用户可能会在单独的子包中找到 PL/Perl。

39.1. PL/Perl 函数和参数

要用 PL/Perl 语言创建函数，使用标准的语法：

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS '
    # PL/Perl function body
' LANGUAGE plperl;
```

函数体是普通的 Perl 代码。

参数和结果的处理与任何其他 Perl 子例程一样：参数通过 `@_` 传入，结果值用 `return` 或作为函数中最后求值的表达式返回。

例如，一个返回两个整数值中较大者的函数可以这样定义：

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS '
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
' LANGUAGE plperl;
```

如果 SQL 空值被传给函数，参数值在 Perl 中将显示为“未定义”。上面的函数定义对空输入表现不太好（事实上，它会把它们当作零来处理）。我们可以在函数定义中加上 `STRICT`，让 PostgreSQL 做更合理的事情：如果传入空值，函数根本不会被调用，而是自动返回空结果。另一种做法是在函数体内检查未定义的输入。例如，假设我们希望 `perl_max` 在一个参数为空、另一个非空时返回非空的那个参数，而不是返回空值：

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS '
    my ($a,$b) = @_;
    if (! defined $a) {
        if (! defined $b) { return undef; }
        return $b;
    }
    if (! defined $b) { return $a; }
```

[1] http://www.perl.com

```
    if ($a > $b) { return $a; }
    return $b;
' LANGUAGE plperl;
```

如上所示，要让 PL/Perl 函数返回 SQL 空值，就返回一个未定义的值。无论函数是否为 strict，都可以这样做。

复合类型的参数以哈希的引用形式传给函数。哈希的键是该复合类型的属性名。
下面是一个例子：

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS '
    my ($emp) = @_;
    return $emp->{''basesalary''} + $emp->{''bonus''};
' LANGUAGE plperl;

SELECT name, empcomp(employee) FROM employee;
```

目前不支持返回复合类型的结果值。

> **提示**
>
> 由于函数体是作为 SQL 字符串字面量传给 `CREATE FUNCTION` 的，你必须转义 Perl 源码中的单引号和反斜线，通常像上例那样将它们加倍。另一种可行的方法是使用 Perl 的扩展引号操作符（`q[]`、`qq[]`、`qw[]`）来避免写单引号。

39.2. PL/Perl 中的数据值

提供给 PL/Perl 函数代码的参数值只是转换为文本形式的输入参数（就像它们被 `SELECT` 语句显示出来一样）。反过来，`return` 命令接受任何作为函数声明的返回类型可接受输入格式的字符串。因此，PL/Perl 程序员可以像操作纯文本一样操作数据值。

39.3. 从 PL/Perl 访问数据库

从 Perl 函数访问数据库本身可以通过实验性模块 `DBD::PgSPI`[2]（也可在 CPAN 镜像站点[3]获取）完成。该模块提供了一个名为 `$pg_dbh` 的符合 DBI 的数据库句柄，可以用正常的 DBI 语法执行查询。

PL/Perl 本身目前只提供一个额外的 Perl 命令：

elog*level*,*msg*

发出一条日志或错误消息。可能的级别有 `DEBUG`、`LOG`、`INFO`、`NOTICE`、`WARNING` 和 `ERROR`。`ERROR` 引发一个错误条件：函数的进一步执行被放弃，当前事务被中止。

[2]http://www.cpan.org/modules/by-module/DBD/APILOS/
[3]http://www.cpan.org/SITES.html

39.4. 受信任与不受信任的 PL/Perl

通常，PL/Perl 作为一个名为 plperl 的“受信任”编程语言安装。在这种设置下，某些 Perl 操作被禁用以保障安全。一般来说，受限制的是那些与环境交互的操作。这包括文件句柄操作、require 和 use（用于外部模块）。没有办法访问数据库服务器进程的内部，也无法以服务器进程的权限获得 OS 级访问——而 C 函数可以做到。因此，任何非特权数据库用户都可以被允许使用这种语言。

下面是一个由于安全原因不允许文件系统操作而不能工作的函数例子：

```
CREATE FUNCTION badfunc() RETURNS integer AS '
    open(TEMP, ">/tmp/badfile");
    print TEMP "Gotcha!\n";
    return 1;
' LANGUAGE plperl;
```

这个函数的创建会成功，但执行它不会成功。

有时需要编写不受这些限制的 Perl 函数。例如，可能需要一个能发送邮件的 Perl 函数。为处理这类情况，也可以把 PL/Perl 安装成一种“不受信任的”语言（通常称为 PL/PerlU）。在这种情况下，完整的 Perl 语言都可用。如果用 createlang 程序安装该语言，则语言名 plperlu 会选择不受信任的 PL/Perl 变体。

PL/PerlU 函数的编写者必须注意，函数不能被用于任何非预期用途，因为它能够执行以数据库管理员身份登录的用户所能做的任何事情。请注意，
数据库系统只允许数据库超级用户用不受信任的语言创建函数。

如果上述函数是由超级用户使用 plperlu 语言创建的，执行就会成功。

> **注意**
>
> 虽然 PL/Perl 函数会为每个 SQL 角色在独立的 Perl 解释器中运行，但给定会话中执行的所有 PL/PerlU 函数都运行在同一个 Perl 解释器中（而且这个解释器并不是任何一个用于 PL/Perl 函数的解释器）。这允许 PL/PerlU 函数自由共享数据，但 PL/Perl 与 PL/PerlU 函数之间不能进行通信。

> **注意**
>
> Perl 无法在单个进程内支持多个解释器，除非构建时使用了适当的标志，即 usemultiplicity 或 useithreads 之一。（除非确实需要使用线程，否则更推荐 usemultiplicity。更多细节见 perlembed 手册页。）如果 PL/Perl 使用的是未按这种方式构建的 Perl 副本，那么每个会话中就只能有一个 Perl 解释器，因此任一会话只能执行 PL/PerlU 函数，或者执行全部由同一 SQL 角色调用的 PL/Perl 函数。

39.5. 缺失的特性

PL/Perl 目前缺少下列特性，但欢迎为它们做出贡献。

- PL/Perl 函数不能直接相互调用（因为它们是 Perl 内部的匿名子例程）。目前也没有办法让它们共享全局变量。

- PL/Perl 不能用于编写触发器函数。
- DBD::PgSPI 或类似的能力应当被集成进标准 PostgreSQL 发行版。

出于安全原因，PL/Perl 会在每个 SQL 角色各自独立的 Perl 解释器中执行该角色调用的函数。这可以防止一个用户意外或恶意地干扰另一个用户的 PL/Perl 函数行为。每个这样的解释器都有自己的 `%_SHARED` 变量值和其他全局状态。因此，当且仅当两个 PL/Perl 函数都由同一 SQL 角色执行时，它们才会共享 `%_SHARED` 的值。在某些应用中，一个会话可能会在多个 SQL 角色下执行代码（通过 `SECURITY DEFINER` 函数、使用 `SET ROLE` 等），这时可能需要显式采取措施，确保 PL/Perl 函数能够通过 `%_SHARED` 共享数据。为此，要确保需要相互通信的函数由同一用户拥有，并将它们标记为 `SECURITY DEFINER`。当然，必须注意不要让这些函数被用于任何非预期用途。

第 40 章 PL/Python - Python 过程语言

PL/Python 过程语言允许用 Python[1] 语言编写 PostgreSQL 函数。

要在某个特定数据库中安装 PL/Python，使用 `createlang plpythonu dbname`。

> **提示**
>
> 如果把某种语言安装到`template1`中，随后创建的所有数据库都会自动安装该语言。

从 PostgreSQL 7.4 起，PL/Python 只能作为一种“不受信任的”（untrusted）语言使用（这意味着它不提供任何限制用户能在其中做什么的方法）。因此它已被改名为 `plpythonu`。如果将来在 Python 中开发出新的安全执行机制，受信任的变体 `plpython` 可能会重新出现。

> **注意**
>
> 源码包的用户必须在安装过程中特别启用 PL/Python 的构建。（更多信息请参阅安装说明。）二进制包的用户可能会在一个单独的子包中找到 PL/Python。

40.1. PL/Python 函数

你编写的 Python 代码会被转换成一个 Python 函数。例如，

```
CREATE FUNCTION myfunc(text) RETURNS text
    AS 'return args[0]'
    LANGUAGE plpythonu;
```

会被转换成

```
def __plpython_procedure_myfunc_23456():
        return args[0]
```

这里假设 23456 是该函数的 OID。

如果你不提供返回值，Python 返回默认的 `None`。语言模块把 Python 的 `None` 转换成 SQL 空值。

PostgreSQL 的函数参数可以在全局列表 `args` 中得到。在 `myfunc` 的例子中，`args[0]` 包含作为 text 参数传进来的任何内容。对于 `myfunc2(text, integer)`，`args[0]` 将包含 `text` 参数，`args[1]` 包含 `integer` 参数。

全局字典 `SD` 可用于在函数调用之间存储数据。这个变量是私有静态数据。全局字典 `GD` 是公共数据，对一个会话中的所有 Python 函数可用。请小心使用。

每个函数在 Python 解释器中得到自己的执行环境，因此来自 `myfunc` 的全局数据和函数参数对 `myfunc2` 不可用。例外是 `GD` 字典中的数据，如上所述。

[1] http://www.python.org

40.2. 触发器函数

当一个函数被用于触发器时，字典 `TD` 包含与触发器相关的值。触发器的行在 `TD["new"]` 和/或 `TD["old"]` 中，取决于触发器事件。`TD["event"]` 包含作为字符串的事件（`INSERT`、`UPDATE`、`DELETE` 或 `UNKNOWN`）。`TD["when"]` 包含 `BEFORE`、`AFTER` 和 `UNKNOWN` 之一。`TD["level"]` 包含 `ROW`、`STATEMENT` 和 `UNKNOWN` 之一。`TD["name"]` 包含触发器的名字，`TD["relid"]` 包含触发器发生的表的 OID。如果触发器是带参数调用的，这些参数可以在 `TD["args"][0]` 到 `TD["args"][(n-1)]` 中得到。

如果 `TD["when"]` 是 `BEFORE`，你可以从 Python 函数返回 `None` 或 `"OK"` 来表示该行未被修改，返回 `"SKIP"` 中止该事件，或者返回 `"MODIFY"` 表示你修改了该行。

40.3. 数据库访问

PL/Python 语言模块自动导入一个名为 `plpy` 的 Python 模块。你可以在 Python 代码中以 `plpy.`*`foo`* 的形式使用这个模块中的函数和常量。目前 `plpy` 实现了函数 `plpy.debug("msg")`、`plpy.log("msg")`、`plpy.info("msg")`、`plpy.notice("msg")`、`plpy.warning("msg")`、`plpy.error("msg")` 和 `plpy.fatal("msg")`。它们大体上等价于从 C 代码中调用 `elog(`*`LEVEL`*`, "msg")`。`plpy.error` 和 `plpy.fatal` 实际上会引发一个 Python 异常，如果未被捕获，该异常会使 PL/Python 模块在函数处理器从 Python 解释器返回时调用 `elog(ERROR, msg)`。从 Python 解释器中长跳转（long-jump）出来可能不太好。`raise plpy.ERROR("msg")` 和 `raise plpy.FATAL("msg")` 分别等价于调用 `plpy.error` 和 `plpy.fatal`。

此外，`plpy` 模块提供两个名为 `execute` 和 `prepare` 的函数。带一个查询字符串和一个可选的 limit 参数调用 `plpy.execute` 会运行该查询并把结果作为一个结果对象返回。结果对象模拟列表或字典对象。结果对象可以按行号和列名访问。它有这些额外的方法：`nrows` 返回查询返回的行数，`status` 是 `SPI_exec()` 的返回值。结果对象可以被修改。

例如，

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

从 `my_table` 返回最多 5 行。如果 `my_table` 有一个列 `my_column`，可以这样访问它：

```
foo = rv[i]["my_column"]
```

第二个函数 `plpy.prepare` 为一个查询准备执行计划。如果查询中有参数引用，就用一个查询字符串和一个参数类型列表调用它。例如：

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name =
 $1", [ "text" ])
```

`text` 是你将为 `$1` 传递的变量的类型。准备好一条语句之后，你用函数 `plpy.execute` 运行它：

```
rv = plpy.execute(plan, [ "name" ], 5)
```

第三个参数是 limit，它是可选的。

在当前版本中，运行 PL/Python 函数时遇到的任何数据库错误都会导致服务器立即终止该函数；不可能用 Python 的 `try ... catch` 构造捕获错误条件。例如，传递给 `plpy.execute` 调用的 SQL 语句中的一个语法错误将终止该函数。这一行为可能在未来的版本中改变。

当你用 PL/Python 模块准备一个计划时，它会被自动保存。阅读 SPI 文档（第 41 章）了解这意味着什么。为了在函数调用之间有效地利用这一点，需要使用持久存储字典 `SD` 或 `GD` 之一（见第 40.1 节）。例如：

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS '
    if SD.has_key("plan"):
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # rest of function
' LANGUAGE plpythonu;
```

第 41 章 服务器编程接口

服务器编程接口（SPI）使用户定义 C 函数的编写者能够在其函数中运行 SQL 命令。SPI 是一组接口函数，用于简化对解析器、规划器/优化器和执行器的访问。SPI 还负责一部分内存管理工作。

> **注意**
>
> 可用的过程语言提供了多种从过程中执行 SQL 命令的方法。这些方法中有一些基于 SPI 或以 SPI 为蓝本，因此本文档对这些语言的用户也会有帮助。

为避免误解，本文用“函数”指SPI接口函数，而用“过程”指使用SPI的用户定义 C 函数。

注意，如果在过程执行期间，事务因某条命令中的错误而中止，控制将不会返回到你的过程。
相反，所有工作都会被回滚，服务器将等待客户端的下一条命令。
一个相关的限制是无法在过程内部执行 `BEGIN`、`COMMIT` 和 `ROLLBACK`（事务控制语句）。这两个限制将来都可能会被改变。

SPI 函数在成功时返回非负结果（要么直接作为整数返回值，要么如后文所述存放在全局变量 `SPI_result` 中）。发生错误时，则返回负值或 `NULL`。

使用 SPI 的源代码文件必须包含头文件 `executor/spi.h`。

41.1. 接口函数

SPI_connect

SPI_connect — 将一个过程连接到 SPI 管理器

大纲

```
int SPI_connect(void)
```

描述

`SPI_connect` 会为某次过程调用打开到 SPI 管理器的连接。如果要通过 SPI 执行命令，就必须调用此函数。不过，有些 SPI 辅助函数可以在未连接的过程中调用。

如果你的过程已经处于连接状态，`SPI_connect`将返回错误代码→ `SPI_ERROR_CONNECT`。当某个直接调用了`SPI_connect`的过程又直接调用另一个调用`SPI_connect`的过程时，就会发生这种情况。虽然当通过 SPI 调用的 SQL 命令调用另一个使用 SPI 的函数时，允许对 SPI 管理器的递归调用，但直接嵌套调用`SPI_connect`和`SPI_finish`是被禁止的。

返回值

`SPI_OK_CONNECT`

成功时

`SPI_ERROR_CONNECT`

发生错误时

SPI_finish

SPI_finish — 将一个过程与 SPI 管理器断开

大纲

```
int SPI_finish(void)
```

描述

`SPI_finish` 关闭到 SPI 管理器的现有连接。在完成当前这次过程调用所需的 SPI 操作后，必须调用此函数。不过，如果通过 `elog(ERROR)` 中止了事务，就不必担心是否显式调用它；这种情况下 SPI 会自行清理。

如果在没有有效连接的情况下调用`SPI_finish`，它将返回`SPI_ERROR_UNCONNECTED`。这并不会带来根本性问题；它只表示 SPI 管理器无事可做。

返回值

`SPI_OK_FINISH`

如果正确地断开连接

`SPI_ERROR_UNCONNECTED`

如果从未连接的过程中调用

SPI_exec

SPI_exec — 执行一条命令

大纲

```
int SPI_exec(const char * command, int count)
```

描述

`SPI_exec` 执行指定的 SQL 命令，并最多检索 *count* 行。

此函数只应从已连接的过程中调用。如果 *count* 为零，则该命令会针对其适用的所有行执行。如果 *count* 大于 0，则该命令要执行的行数会受到限制（很像一个 `LIMIT` 子句）。例如：

```
SPI_exec("INSERT INTO tab SELECT * FROM tab", 5);
```

最多只允许向该表中插入 5 行。

你可以在一个字符串中传递多条命令，并且命令可能会被规则改写。`SPI_exec` 返回最后执行的那条命令的结果。

（最后一条）命令实际执行所处理的行数，会通过全局变量 `SPI_processed` 返回（除非函数的返回值是 `SPI_OK_UTILITY`）。如果函数返回值是 `SPI_OK_SELECT`，则可以通过全局指针 `SPITupleTable *SPI_tuptable` 访问结果行。

结构 `SPITupleTable` 定义如下：

```
typedef struct
{
    MemoryContext tuptabcxt;    /* 结果表的内存上下文 */
    uint32      alloced;        /* 已分配的 vals 数量 */
    uint32      free;           /* 空闲的 vals 数量 */
    TupleDesc   tupdesc;        /* 行描述符 */
    HeapTuple  *vals;           /* 行 */
} SPITupleTable;
```

`vals` 是一个指向行的指针数组。（有效项数为 `SPI_processed`。）`tupdesc` 是一个行描述符，可以传给处理行的 SPI 函数。`tuptabcxt`，`alloced` 和 `free` 是内部字段，不供 SPI 调用者使用。

`SPI_finish` 会释放当前过程调用期间分配的全部 `SPITupleTable`。如果某个结果表已经不再需要，也可以提前调用 `SPI_freetuptable` 释放它。

参数

`const char * command`

包含待执行命令的字符串

`int count`

要处理或返回的最大行数

返回值

如果命令执行成功，则返回下列（非负）值之一：

SPI_OK_SELECT

执行了 SELECT（但不是 SELECT INTO）

SPI_OK_SELINTO

执行了 SELECT INTO

SPI_OK_DELETE

执行了 DELETE

SPI_OK_INSERT

执行了 INSERT

SPI_OK_UPDATE

执行了 UPDATE

SPI_OK_UTILITY

执行了工具命令（例如 CREATE TABLE）

出错时，返回以下负值之一：

SPI_ERROR_ARGUMENT

command 为 NULL，或者 *count* 小于 0

SPI_ERROR_COPY

尝试执行了 COPY TO stdout 或 COPY FROM stdin

SPI_ERROR_CURSOR

尝试执行了 DECLARE、CLOSE 或 FETCH

SPI_ERROR_TRANSACTION

尝试执行了 BEGIN、COMMIT 或 ROLLBACK

SPI_ERROR_OPUNKNOWN

命令类型未知（理论上不应发生）

SPI_ERROR_UNCONNECTED

如果从未连接的过程中调用

注解

函数 SPI_exec、SPI_execp 和 SPI_prepare 都会更改 SPI_processed 和 SPI_tuptable（只更改指针，而不更改结构体内容）。如果需要在后续调用之后继续访问 SPI_exec 或 SPI_execp 的结果，请把这两个全局变量保存到过程局部变量中。

SPI_prepare

SPI_prepare — 为命令准备一个计划，但暂不执行

大纲

```
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid
 * argtypes)
```

描述

SPI_prepare 为指定命令创建并返回一个执行计划，但不执行该命令。该函数只能从已连接的过程调用。

当同一条或相似命令需要反复执行时，只做一次规划可能是有利的。SPI_prepare 把命令字符串转换成一个执行计划，之后可以使用 SPI_execp 反复执行。

预备命令可以通过在普通命令原本应写常量的位置写入参数（$1、$2 等）而得到泛化。这些参数的实际值会在调用 SPI_execp 时指定。这样，预备语句就能适用于比无参形式更广泛的场景。

SPI_prepare 返回的计划只能在当前这次过程调用中使用，因为 SPI_finish 会释放为计划分配的内存。但可以使用函数 SPI_saveplan 把计划保存得更久。

参数

const char * *command*

命令字符串

int *nargs*

输入参数的数量（$1、$2 等）

Oid * *argtypes*

一个数组指针，它指向的数组包含参数的数据类型的 OID

返回值

SPI_prepare 返回一个指向执行计划的非空指针。发生错误时会返回 NULL。无论成功还是出错，SPI_result 都会被设成与 SPI_exec 所返回值类似的值；但如果 *command* 为 NULL，或者 *nargs* 小于 0，或者 *nargs* 大于 0 且 *argtypes* 为 NULL，则会将其设置为 SPI_ERROR_ARGUMENT。

注解

使用参数有一个不利之处：由于规划器不知道将为参数提供什么值，
它做出的规划选择可能比在所有常量都可见的普通命令下更差。

SPI_execp

SPI_execp — 执行由 SPI_prepare 准备好的计划

大纲

```
int SPI_execp(void * plan, Datum *values, const char * nulls,
 int count)
```

描述

SPI_execp 执行由 SPI_prepare 准备的计划。其 *count* 参数的含义与 SPI_exec 中相同。

参数

void * *plan*

执行计划（由 SPI_prepare 返回）

Datum **values*

实际参数值

const char * *nulls*

一个描述哪些参数为空值的数组。n 表示空值（*values* 中的对应项会被忽略）；空格表示非空值（*values* 中的对应项有效）。

如果 *nulls* 为 NULL，则 SPI_execp 假定没有参数为空值。

int *count*

该计划将要执行的行数

返回值

返回值与 SPI_exec 的相同，或者是下列值之一：

SPI_ERROR_ARGUMENT

如果*plan*为NULL 或者*count*小于 0

SPI_ERROR_PARAM

如果*values*为NULL，而 *plan*在准备时带有参数

成功时，SPI_processed 和 SPI_tuptable 的设置方式与 SPI_exec 相同。

注解

如果预备计划所引用的对象（表、函数等）在会话期间被删除，那么对该计划执行 SPI_execp 的结果将是不可预知的。

SPI_cursor_open

SPI_cursor_open — 使用由 `SPI_prepare` 创建的计划建立一个游标

大纲

```
Portal SPI_cursor_open(const char * name, void * plan, Datum * values,
 const char * nulls)
```

描述

`SPI_cursor_open` 建立一个游标（在内部是一个 portal），该游标将执行由 `SPI_prepare` 准备好的一个计划。

使用游标而不是直接执行该计划有两个好处。首先，可以一次只取出少量结果行，从而避免对返回大量行的查询耗尽内存。其次，一个 portal 可以比当前的过程存活更久（实际上，它可以一直存活到当前事务结束）。把 portal 的名称返回给该过程的调用者，就提供了一种把行集作为结果返回的方法。

参数

`const char * name`

portal 的名称，或为`NULL`以让系统选择名称

`void * plan`

执行计划（由 `SPI_prepare` 返回）

`Datum * values`

实际参数值

`const char *nulls`

一个描述哪些参数为空值的数组。`n` 表示空值（*values* 中的对应项会被忽略）；空格表示非空值（*values* 中的对应项有效）。如果 *nulls* 为 `NULL`，则 `SPI_cursor_open` 假定没有参数为空值。

返回值

指向包含该游标的 portal 的指针，出错时为 `NULL`

SPI_cursor_find

SPI_cursor_find — 用名称查找一个现有的游标

大纲

`Portal SPI_cursor_find(const char * name)`

描述

`SPI_cursor_find` 按名称查找现有的 portal。这主要用于解析由其他函数以文本形式返回的游标名。

参数

`const char * name`

该 portal 的名称

返回值

带有指定名称的 portal 的指针，如果没有找到就是 `NULL`

SPI_cursor_fetch

SPI_cursor_fetch — 从一个游标取出一些行

大纲

`void SPI_cursor_fetch(Portal portal, bool forward, int count)`

描述

`SPI_cursor_fetch`从一个游标取得一些行。这等效于 SQL 命令`FETCH`。

参数

`Portal portal`

包含该游标的 portal

`bool forward`

为真表示向前获取，为假表示向后获取

`int count`

要取得的最大行数

返回值

成功时，`SPI_processed` 和 `SPI_tuptable` 的设置方式与 `SPI_exec` 相同。

SPI_cursor_move

SPI_cursor_move — 移动一个游标

大纲

```
void SPI_cursor_move(Portal portal, bool forward, int count)
```

描述

`SPI_cursor_move`跳过一个游标中的一些行。这等效于 SQL 命令`MOVE`。

参数

`Portal` *`portal`*

包含该游标的 portal

`bool` *`forward`*

为真表示前移，为假表示后移

`int` *`count`*

要移动的最大行数

SPI_cursor_close

SPI_cursor_close — 关闭一个游标

大纲

`void SPI_cursor_close(Portal portal)`

描述

`SPI_cursor_close`关闭一个之前创建的游标并且释放它的 portal 存储。

所有打开的游标会在事务结束时自动被关闭。只有在希望尽快释放资源时，才需要调用 `SPI_cursor_close`。

参数

`Portal portal`

包含该游标的 portal

SPI_saveplan

SPI_saveplan — 保存一个计划

大纲

`void * SPI_saveplan(void * plan)`

描述

`SPI_saveplan` 会把传入的计划（由 `SPI_prepare` 准备）保存到受保护、不会被 `SPI_finish` 和事务管理器释放的内存中，并返回指向所保存计划的指针。这让你能够在当前会话中当前过程的后续调用里重用预备计划。你可以把返回的指针保存到一个局部变量中。
无论是准备计划时，还是在 `SPI_execp` 中使用一个已准备好的计划时，都应始终检查该指针是否为 `NULL`。

参数

`void * plan`

要保存的计划

返回值

指向所保存计划的指针；如果不成功则为 `NULL`。出错时，`SPI_result` 会被设置为：

`SPI_ERROR_ARGUMENT`

如果 *plan* 为 `NULL`

`SPI_ERROR_UNCONNECTED`

如果从未连接的过程中调用

注解

如果预备计划所引用的对象（表、函数等）之一在该会话期间被删除，那么对该计划执行 `SPI_execp` 的结果将是不可预测的。

41.2. 接口支持函数

这里介绍的函数提供了一个接口，用于从 `SPI_exec` 及其他 SPI 函数返回的结果集中提取信息。

本节介绍的所有函数都可以在已连接和未连接的过程中使用。

SPI_fname

SPI_fname — 为指定的列号确定列名

大纲

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

描述

`SPI_fname` 返回指定列的列名。（不再需要时，可以用 `pfree` 释放该名称的副本。）

参数

`TupleDesc` *`rowdesc`*

输入行描述

`int` *`colnumber`*

列号（从 1 开始计）

返回值

列名；如果 *`colnumber`* 超出范围则返回 `NULL`。出错时，`SPI_result` 会被设置为 `SPI_ERROR_NOATTRIBUTE`。

SPI_fnumber

SPI_fnumber — 根据指定列名确定列号

大纲

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

描述

`SPI_fnumber` 返回指定列名对应的列号。

如果 *colname* 指向的是系统列（例如 `oid`），则会返回相应的负列号。调用者应当通过检查返回值是否恰好等于 `SPI_ERROR_NOATTRIBUTE` 来判断错误；除非本就要拒绝系统列，否则测试结果是否小于等于 0 并不正确。

参数

`TupleDesc` *rowdesc*

输入行描述

`const char *` *colname*

列名

返回值

列号（从 1 开始计），如果没有找到所提到的列名则返回 `SPI_ERROR_NOATTRIBUTE`。

SPI_getvalue

SPI_getvalue — 返回指定列的字符串值

大纲

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

描述

`SPI_getvalue`返回指定列的值的字符串表示。

结果保存在由 `palloc` 分配的内存中返回。（不再需要时，可以用 `pfree` 释放该内存。）

参数

`HeapTuple` *`row`*

要检查的输入行

`TupleDesc` *`rowdesc`*

输入行描述

`int` *`colnumber`*

列号（从 1 开始计）

返回值

列值；如果该列为空值、*`colnumber`* 超出范围（`SPI_result` 被设置为 `SPI_ERROR_NOATTRIBUTE`）或者没有输出函数可用（`SPI_result` 被设置为 `SPI_ERROR_NOOUTFUNC`）则返回 `NULL`。

SPI_getbinval

SPI_getbinval — 返回指定列的二进制值

大纲

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
                    bool * isnull)
```

描述

`SPI_getbinval` 以内部形式（即 `Datum` 类型）返回指定列的值。

此函数不会为该 datum 分配新空间。对于传引用的数据类型，返回值将是指向传入行内部数据的指针。

参数

`HeapTuple` *`row`*

要检查的输入行

`TupleDesc` *`rowdesc`*

输入行描述

`int` *`colnumber`*

列号（从 1 开始计）

`bool *` *`isnull`*

列中是否为空值的标志

返回值

返回该列的二进制值。如果该列为空值，则 *`isnull`* 指向的变量会被设为真，否则设为假。

出错时，`SPI_result` 会被设置为 `SPI_ERROR_NOATTRIBUTE`。

SPI_gettype

SPI_gettype — 返回指定列的数据类型名称

大纲

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

描述

`SPI_gettype`返回指定列的数据类型名称。（当你不再需要该名称的副本后，可以使用`pfree`释放它。）

参数

`TupleDesc` *`rowdesc`*

输入行描述

`int` *`colnumber`*

列号（从 1 开始计）

返回值

指定列的数据类型名称；出错时返回 `NULL`。出错时，`SPI_result` 会被设置为 `SPI_ERROR_NOATTRIBUTE`。

SPI_gettypeid

SPI_gettypeid — 返回指定列的数据类型 OID

大纲

`Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)`

描述

`SPI_gettypeid`返回该指定列的数据类型的 OID。

参数

`TupleDesc rowdesc`

输入行描述

`int colnumber`

列号（从 1 开始计）

返回值

指定列数据类型的 OID；出错时返回 `InvalidOid`。出错时，`SPI_result` 会被设置为 `SPI_ERROR_NOATTRIBUTE`。

SPI_getrelname

SPI_getrelname — 返回指定关系的名称

大纲

```
char * SPI_getrelname(Relation rel)
```

描述

SPI_getrelname返回指定关系的名称。（当你不再需要该名称的副本后，可以使用pfree 释放它。）

参数

Relation *rel*

输入关系

返回值

指定关系的名称。

41.3. 内存管理

PostgreSQL 在内存上下文中分配内存。内存上下文为管理那些在许多不同位置创建、且生命周期各不相同的内存分配提供了便捷手段。销毁某个上下文时，会释放其中分配的全部内存。因此，不必为了避免内存泄漏而跟踪每个单独对象；只需管理数量相对较少的上下文即可。palloc 及相关函数都从“当前”上下文中分配内存。

SPI_connect 创建新的内存上下文并将其设为当前上下文。SPI_finish 则恢复先前的当前上下文，并销毁由 SPI_connect 创建的上下文。这些操作可以确保在过程内部所做的临时内存分配会在过程退出时被回收，从而避免内存泄漏。

不过，如果过程需要返回位于已分配内存中的对象（例如传引用数据类型的值），就不能使用 palloc 来分配这块内存，至少在连接到 SPI 的期间不能这么做。否则，该对象会在 SPI_finish 时被释放，过程也就无法可靠工作。解决办法是用 SPI_palloc 为返回对象分配内存。SPI_palloc 在“上层执行器上下文”中分配内存，也就是调用 SPI_connect 时的当前内存上下文；这正是从过程返回值最合适的上下文。

如果在过程尚未连接到 SPI 时调用SPI_palloc，它的行为与普通的palloc相同。在过程连接到 SPI 管理器之前，当前内存上下文就是上层执行器上下文，因此过程通过palloc或 SPI 辅助函数所做的所有分配都位于这个上下文中。

调用 SPI_connect 时，会把该过程的私有上下文（由 SPI_connect 创建）设为当前上下文。所有通过 palloc、repalloc 或 SPI 辅助函数分配的内存（但SPI_copytuple、SPI_copytupledesc、SPI_copytupleintoslot、SPI_modifytuple 和SPI_palloc除外）都位于这个上下文中。当过程通过 SPI_finish 与 SPI 管理器断开连接时，当前上下文会恢复为上层执行器上下文，而在该过程内存上下文中分配的所有内存都会被释放，之后就不能再使用。

本节介绍的所有函数既可由已连接的过程使用，也可由未连接的过程使用。在未连接的过程中，它们的行为与底层普通服务器函数（palloc等）相同。

SPI_palloc

SPI_palloc — 在上层执行器上下文中分配内存

大纲

```
void * SPI_palloc(Size size)
```

描述

`SPI_palloc` 在上层执行器上下文中分配内存。

参数

`Size size`

要分配的存储空间大小（以字节计）

返回值

指向具有指定大小的新存储空间的指针

SPI_repalloc

SPI_repalloc — 在上层执行器上下文中重分配内存

大纲

```
void * SPI_repalloc(void * pointer, Size size)
```

描述

`SPI_repalloc`改变之前用`SPI_palloc` 分配的内存段的大小。

这个函数如今与普通的`repalloc`已无区别。保留它只是为了与现有代码保持向后兼容。

参数

`void * pointer`

指向要改变的现有存储空间的指针

`Size size`

要分配的存储空间大小（以字节计）

返回值

指向具有指定大小的新存储空间的指针，现有区域的内容会被复制到其中

SPI_pfree

SPI_pfree — 在上层执行器上下文中释放内存

大纲

`void SPI_pfree(void * pointer)`

描述

`SPI_pfree`释放之前使用 `SPI_palloc`或者 `SPI_repalloc`分配的内存。

这个函数如今与普通的`pfree`已无区别。保留它只是为了与现有代码保持向后兼容。

参数

`void * pointer`

指向要释放的现有存储空间的指针

SPI_copytuple

SPI_copytuple — 在上层执行器上下文中创建一行的副本

大纲

`HeapTuple SPI_copytuple(HeapTuple row)`

描述

`SPI_copytuple` 在上层执行器上下文中创建一行的副本。

参数

`HeapTuple row`

要复制的行

返回值

复制后的行；仅当*tuple*为 `NULL` 时才返回 `NULL`

SPI_copytupledesc

SPI_copytupledesc — 在上层执行器上下文中创建一个行描述符的副本

大纲

`TupleDesc SPI_copytupledesc(TupleDesc tupdesc)`

描述

`SPI_copytupledesc` 在上层执行器上下文中创建一个行描述符的副本。

参数

`TupleDesc tupdesc`

要复制的行描述符

返回值

复制得到的行描述符；仅当 *tupdesc* 为 `NULL` 时才返回 `NULL`

SPI_copytupleintoslot

SPI_copytupleintoslot — 在上层执行器上下文中创建一个行及其描述符的副本

大纲

```
TupleTableSlot * SPI_copytupleintoslot(HeapTuple row, Tuple
Desc rowdesc)
```

描述

`SPI_copytupleintoslot` 在上层执行器上下文中创建一个行的副本，并以填充好的 `Tuple TableSlot` 结构的形式返回它。

参数

`HeapTuple` *`row`*

要复制的行

`TupleDesc` *`rowdesc`*

要复制的行描述符

返回值

包含复制得到的行和描述符的 `TupleTableSlot`；仅当 *`row`* 或 *`rowdesc`* 为 `NULL` 时才返回 `NULL`

SPI_modifytuple

SPI_modifytuple — 通过替换给定行中选定的字段来创建一个行

大纲

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, ncols, colnum,
 Datum * values, const char * nulls)
```

描述

`SPI_modifytuple` 通过为选定列替换新值、并在其他位置复制原行的列来创建一行新记录。输入行本身不会被修改。

参数

`Relation rel`

仅用作该行的行描述符来源。（传递关系而不是行描述符是个设计缺陷。）

`HeapTuple row`

要被修改的行

`int ncols`

数组 *colnum* 中列号的数量

`int * colnum`

一个数组，包含要被修改的列的编号（列号从 1 开始）

`Datum * values`

指定列的新值

`const char * Nulls`

哪些新值为空值（如果有的话；格式参见 `SPI_execp`）

返回值

修改后的新行，分配在上层执行器上下文中；仅当*row* 为 `NULL` 时才返回 `NULL`

出错时，`SPI_result`被设置如下：

`SPI_ERROR_ARGUMENT`

如果*rel*为`NULL`，或者 *row*为`NULL`，或者*ncols* 小于等于 0，或者*colnum*为`NULL`，或者*values*为`NULL`。

`SPI_ERROR_NOATTRIBUTE`

如果*colnum*包含一个无效的列号（小于等于 0 或者大于 *row*中的列数）。

SPI_freetuple

SPI_freetuple — 释放在上层执行器上下文中分配的一个行

大纲

```
void SPI_freetuple(HeapTuple row)
```

描述

`SPI_freetuple`释放之前在上层执行器上下文中分配的一行。

这个函数如今与普通的`heap_freetuple`已无区别。保留它只是为了与现有代码保持向后兼容。

参数

`HeapTuple` *`row`*

要释放的行

SPI_freetuptable

SPI_freetuptable — 释放由 `SPI_exec` 或类似函数创建的结果行集

大纲

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

描述

`SPI_freetuptable` 释放由先前的 SPI 命令执行函数（例如 `SPI_exec`）创建的结果行集。因此，调用它时经常会直接把全局变量 `SPI_tupletable` 作为参数。

如果某个使用 SPI 的过程需要执行多条命令，并且不想将前面命令的结果一直保留到过程结束，那么这个函数就很有用。注意，任何尚未释放的结果行集都会在 `SPI_finish` 时被释放。

参数

`SPITupleTable * tuptable`

要释放的行集的指针

SPI_freeplan

SPI_freeplan — 释放一个先前保存的计划

大纲

`int SPI_freeplan(void * plan)`

描述

`SPI_freeplan` 释放先前由 `SPI_prepare` 返回或由 `SPI_saveplan` 保存的命令执行计划。

参数

`void * plan`

要释放的计划的指针

返回值

如果 *plan* 为 `NULL`，返回 `SPI_ERROR_ARGUMENT`

41.4. 数据更改的可见性

下列两条规则决定了使用 SPI 的函数（以及其他任何 C 函数）中数据更改的可见性：

- 在某条 SQL 命令执行期间，该命令（或由该命令调用的函数，包括触发器函数）所做的任何数据更改对该命令本身都是不可见的。例如，在命令

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  中，被插入的行对`SELECT`部分不可见。

- 某条命令 C 所做的更改，对所有在 C 之后启动的命令都可见，无论这些命令是在 C 内部（即 C 执行期间）启动，还是在 C 完成之后启动。

下一节给出了一个展示这些规则如何应用的示例。

41.5. 示例

本节给出一个非常简单的 SPI 用法示例。过程 `execq` 以一条 SQL 命令作为第一个参数、以行数计数作为第二个参数，使用 `SPI_exec` 执行该命令，并返回该命令实际处理的行数。更复杂的 SPI 示例可以在源码树的 `src/test/regress/regress.c` 以及 `contrib/spi` 中找到。

```
#include "executor/spi.h"

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
    char *command;
```

```
    int ret;
    int proc;

    /* Convert given text object to a C string */
    command = DatumGetCString(DirectFunctionCall1(textout,
                                                  PointerGetDatum(sql)
));

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;
    /*
     * If this is a SELECT and some rows were fetched,
     * then the rows are printed via elog(INFO).
     */
    if (ret == SPI_OK_SELECT && SPI_processed > 0)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i, j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen (buf), sizeof(buf) - strlen(buf)
, " %s%s",
                         SPI_getvalue(tuple, tupdesc, i),
                         (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    return (proc);
}
```

（此函数使用版本 0 调用约定，以使示例更易于理解。在实际应用中，你应该使用新的版本 1 接口。）

将该函数编译成共享库之后，可以这样声明它：

```
CREATE FUNCTION execq(text, integer) RETURNS integer
    AS 'filename'
    LANGUAGE C;
```

以下是一个示例会话：

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
```

```
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 167631 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- 由 execq 插入
INFO:  EXECQ:  1    -- 由 execq 返回并被外层 INSERT 插入

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2    -- 0 + 2, 按指定只插入一行

 execq
-------
     3              -- 10 只是最大值, 3 才是真正的行数
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 167712 1
=> SELECT * FROM a;
 x
---
 1                  -- a 中没有行 (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  0
INSERT 167713 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- a 中原有一行 + 1
(2 rows)

-- 下面演示数据更改可见性规则:
```

```
=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 行 * 1 (第一行中的 x)
 6                  -- 3 行 (前面的 2 行加上刚插入的 1 行) * 2 (第二行中的
 x)
(4 rows)               ^^^^^^
                       不同调用中 execq() 可见的行
```

部分 VI. 参考手册

本参考手册中的条目旨在以合理的篇幅提供关于其各自主题的权威、完整而正式的摘要。关于PostgreSQL用法的更多信息——叙述式、教程式或示例式的——可以在本书的其他部分找到。请参阅每个参考页上列出的交叉引用。

这些参考条目也可以以传统的“man”页的形式获得。

目录

I. SQL 命令 602
ABORT 604
ALTER AGGREGATE 605
ALTER CONVERSION 606
ALTER DATABASE 607
ALTER DOMAIN 609
ALTER FUNCTION 611
ALTER GROUP 612
ALTER LANGUAGE 613
ALTER OPERATOR CLASS 614
ALTER SCHEMA 615
ALTER SEQUENCE 616
ALTER TABLE 618
ALTER TRIGGER 623
ALTER USER 624
ANALYZE 627
BEGIN 629
CHECKPOINT 630
CLOSE 631
CLUSTER 632
COMMENT 634
COMMIT 636
COPY 637
CREATE AGGREGATE 643
CREATE CAST 645
CREATE CONSTRAINT TRIGGER 648
CREATE CONVERSION 649
CREATE DATABASE 651
CREATE DOMAIN 653
CREATE FUNCTION 655
CREATE GROUP 659
CREATE INDEX 660
CREATE LANGUAGE 662
CREATE OPERATOR 664
CREATE OPERATOR CLASS 667
CREATE RULE 670
CREATE SCHEMA 672
CREATE SEQUENCE 674
CREATE TABLE 677
CREATE TABLE AS 685
CREATE TRIGGER 686
CREATE TYPE 688
CREATE USER 693
CREATE VIEW 695
DEALLOCATE 697
DECLARE 698
DELETE 701
DROP AGGREGATE 703
DROP CAST 704
DROP CONVERSION 705
DROP DATABASE 706

DROP DOMAIN 707
DROP FUNCTION 708
DROP GROUP 709
DROP INDEX 710
DROP LANGUAGE 711
DROP OPERATOR 712
DROP OPERATOR CLASS 714
DROP RULE 715
DROP SCHEMA 716
DROP SEQUENCE 717
DROP TABLE 718
DROP TRIGGER 719
DROP TYPE 720
DROP USER 721
DROP VIEW 722
END 723
EXECUTE 724
EXPLAIN 725
FETCH 728
GRANT 732
INSERT 736
LISTEN 738
LOAD 739
LOCK 740
MOVE 742
NOTIFY 743
PREPARE 745
REINDEX 747
RESET 749
REVOKE 750
ROLLBACK 752
SELECT 753
SELECT INTO 763
SET 764
SET CONSTRAINTS 767
SET SESSION AUTHORIZATION 768
SET TRANSACTION 770
SHOW 772
START TRANSACTION 774
TRUNCATE 775
UNLISTEN 776
UPDATE 777
VACUUM 779
II. PostgreSQL 客户端应用 782
clusterdb 783
createdb 785
createlang 788
createuser 790
dropdb 793
droplang 795
dropuser 797
ecpg 799
pg_config 801
pg_dump 803

pg_dumpall 809
pg_restore 812
pgtclsh 818
pgtksh 819
psql 820
vacuumdb 842
III. PostgreSQL 服务器应用 845
initdb 846
initlocation 848
ipcclean 849
pg_controldata 850
pg_ctl 851
pg_resetxlog 855
postgres 856
postmaster 860

SQL 命令

这一部分包含PostgreSQL所支持的 SQL命令的参考信息。这里的 “SQL” 指的是这门语言本身；每条命令的标准符合性和兼容性信息可以在相应的参考页上找到。

目录

ABORT 604
ALTER AGGREGATE 605
ALTER CONVERSION 606
ALTER DATABASE 607
ALTER DOMAIN 609
ALTER FUNCTION 611
ALTER GROUP 612
ALTER LANGUAGE 613
ALTER OPERATOR CLASS 614
ALTER SCHEMA 615
ALTER SEQUENCE 616
ALTER TABLE 618
ALTER TRIGGER 623
ALTER USER 624
ANALYZE 627
BEGIN 629
CHECKPOINT 630
CLOSE 631
CLUSTER 632
COMMENT 634
COMMIT 636
COPY 637
CREATE AGGREGATE 643
CREATE CAST 645
CREATE CONSTRAINT TRIGGER 648
CREATE CONVERSION 649
CREATE DATABASE 651
CREATE DOMAIN 653
CREATE FUNCTION 655
CREATE GROUP 659
CREATE INDEX 660
CREATE LANGUAGE 662
CREATE OPERATOR 664
CREATE OPERATOR CLASS 667
CREATE RULE 670
CREATE SCHEMA 672
CREATE SEQUENCE 674
CREATE TABLE 677
CREATE TABLE AS 685
CREATE TRIGGER 686
CREATE TYPE 688
CREATE USER 693
CREATE VIEW 695
DEALLOCATE 697

DECLARE 698
DELETE 701
DROP AGGREGATE 703
DROP CAST 704
DROP CONVERSION 705
DROP DATABASE 706
DROP DOMAIN 707
DROP FUNCTION 708
DROP GROUP 709
DROP INDEX 710
DROP LANGUAGE 711
DROP OPERATOR 712
DROP OPERATOR CLASS 714
DROP RULE 715
DROP SCHEMA 716
DROP SEQUENCE 717
DROP TABLE 718
DROP TRIGGER 719
DROP TYPE 720
DROP USER 721
DROP VIEW 722
END 723
EXECUTE 724
EXPLAIN 725
FETCH 728
GRANT 732
INSERT 736
LISTEN 738
LOAD 739
LOCK 740
MOVE 742
NOTIFY 743
PREPARE 745
REINDEX 747
RESET 749
REVOKE 750
ROLLBACK 752
SELECT 753
SELECT INTO 763
SET 764
SET CONSTRAINTS 767
SET SESSION AUTHORIZATION 768
SET TRANSACTION 770
SHOW 772
START TRANSACTION 774
TRUNCATE 775
UNLISTEN 776
UPDATE 777
VACUUM 779

ABORT

ABORT — 中止当前事务

大纲

```
ABORT [ WORK | TRANSACTION ]
```

描述

`ABORT`回卷当前事务并使该事务所做的所有更新被丢弃。
这条命令的行为与标准SQL命令ROLLBACK完全相同，其存在仅出于历史原因。

参数

`WORK`
`TRANSACTION`

可选的关键字。它们没有任何效果。

注解

使用COMMIT来成功终止一个事务。

在不处于事务内部时发出`ABORT`没有害处，但会引出一条警告消息。

示例

中止所有更改：

```
ABORT;
```

兼容性

这条命令是一个因历史原因而保留的 PostgreSQL扩展。`ROLLBACK` 是等价的标准 SQL 命令。

另见

BEGIN, COMMIT, ROLLBACK

ALTER AGGREGATE

ALTER AGGREGATE — 更改一个聚合函数的定义

大纲

```
ALTER AGGREGATE name ( type ) RENAME TO newname
```

描述

`ALTER AGGREGATE` 更改一个聚合函数的定义。目前唯一可用的功能是重命名该聚合函数。

参数

name

一个现有聚合函数的名称（可带模式限定）。

type

该聚合函数的参数数据类型，如果函数接受任意数据类型则为 `*`。

newname

聚合函数的新名称。

示例

要把类型 `integer` 的聚合函数 `myavg` 重命名为 `my_average`：

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

兼容性

SQL 标准中没有 `ALTER AGGREGATE` 语句。

另见

CREATE AGGREGATE, DROP AGGREGATE

ALTER CONVERSION

ALTER CONVERSION — 更改一个转换的定义

大纲

`ALTER CONVERSION` *`name`* `RENAME TO` *`newname`*

描述

`ALTER CONVERSION` 更改一个转换的定义。目前唯一可用的功能是重命名转换。

参数

`name`

一个现有转换的名称（可以是模式限定的）。

`newname`

转换的新名称。

示例

要把转换 `iso_8859_1_to_utf_8` 重命名为 `latin1_to_unicode`：

```
ALTER CONVERSION iso_8859_1_to_utf_8 RENAME TO latin1_to_unicode;
```

兼容性

在 SQL 标准中没有`ALTER CONVERSION`语句。

另见

CREATE CONVERSION, DROP CONVERSION

ALTER DATABASE

ALTER DATABASE — 更改一个数据库

大纲

```
ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name RESET parameter

ALTER DATABASE name RENAME TO newname
```

描述

`ALTER DATABASE`用于更改一个数据库的属性。

前两种形式更改一个PostgreSQL数据库的某个运行时配置变量的会话默认值。
此后每当在该数据库中启动新的会话时，指定的值就会成为会话默认值。
数据库特定的默认值会覆盖`postgresql.conf`中的设置或从 `postmaster`命令行接收到的设置。只有数据库拥有者或超级用户才能更改数据库的会话默认值。

第三种形式更改数据库的名字。只有数据库拥有者才能重命名数据库，而且他还必须拥有`CREATEDB`权限。当前数据库不能被重命名。（如果你需要这样做，
请连接到一个不同的数据库。）

参数

name

要更改其会话默认值的数据库名字。

parameter
value

把该数据库指定配置参数的会话默认值设置为给定值。如果 *value*为`DEFAULT`，或者等效地使用`RESET`，则会移除数据库特定的变量设置，新会话将继承系统范围的默认设置。使用`RESET ALL`可清除所有设置。

有关允许的参数名和取值的更多信息，请参阅SET和第 16.4 节。

newname

数据库的新名字。

注意

使用ALTER USER，也可以把会话默认值绑定到一个特定的用户而不是数据库。如果发生冲突，用户特定的设置会覆盖数据库特定的设置。

示例

要在数据库`test`中默认禁用索引扫描：

```
ALTER DATABASE test SET enable_indexscan TO off;
```

兼容性

`ALTER DATABASE`语句是一个 PostgreSQL扩展。

另见

ALTER USER, CREATE DATABASE, DROP DATABASE, SET

ALTER DOMAIN

ALTER DOMAIN — 更改一个域的定义

大纲

```
ALTER DOMAIN name
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
    { SET | DROP } NOT NULL
ALTER DOMAIN name
    ADD domain_constraint
ALTER DOMAIN name
    DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
    OWNER TO new_owner
```

描述

`ALTER DOMAIN`更改一个现有域的定义。有以下几种子形式：

SET/DROP DEFAULT

这些形式为域设置或移除默认值。注意，默认值只适用于后续的 `INSERT`命令；它们不会影响已存在于使用该域的表中的行。

SET/DROP NOT NULL

这些形式更改域是被标记为允许 NULL 值还是拒绝 NULL 值。只有当使用该域的列中不包含空值时，才能执行`SET NOT NULL`。

ADD *domain_constraint*

这种形式使用与CREATE DOMAIN相同的语法向域添加一个新约束。只有当所有使用该域的列都满足新约束时，这一操作才会成功。

DROP CONSTRAINT

这种形式删除域上的约束。

OWNER

这种形式将域的拥有者更改为指定用户。

要使用 `ALTER DOMAIN`，你必须拥有该域；但 `ALTER DOMAIN OWNER` 例外，它只能由超级用户执行。

参数

name

要修改的现有域的名称（可以是模式限定的）。

domain_constraint

该域的新约束。

constraint_name

要删除的现有约束的名称。

CASCADE

自动删除依赖于该约束的对象。

RESTRICT

如果存在任何依赖对象，则拒绝删除该约束。这是默认行为。

new_owner

域的新拥有者的用户名。

示例

向域添加一个 NOT NULL 约束：

```
ALTER DOMAIN zipcode SET NOT NULL;
```

从域中移除一个 NOT NULL 约束：

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

向域添加一个检查约束：

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) =
 5);
```

从域中移除一个检查约束：

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

兼容性

`ALTER DOMAIN` 语句与 SQL99 兼容，但 `OWNER` 变体是 PostgreSQL 的扩展。

ALTER FUNCTION

ALTER FUNCTION — 更改一个函数的定义

大纲

```
ALTER FUNCTION name ( [ type [, ...] ] ) RENAME TO newname
```

描述

`ALTER FUNCTION`更改一个函数的定义。它唯一的功能是重命名该函数。

参数

name

一个现有函数的名称（可以是模式限定的）。

type

该函数的一个参数的数据类型。

newname

该函数的新名称。

示例

要把`sqrt`函数（用于类型`integer`）重命名为`square_root`：

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

兼容性

SQL 标准中有`ALTER FUNCTION`语句，但它不提供重命名函数的选项。

另请参阅

CREATE FUNCTION, DROP FUNCTION

ALTER GROUP

ALTER GROUP — 更改一个用户组

大纲

```
ALTER GROUP groupname ADD USER username [, ... ]
ALTER GROUP groupname DROP USER username [, ... ]

ALTER GROUP groupname RENAME TO newname
```

描述

`ALTER GROUP`用于更改一个用户组。前两种变体向组中添加或从组中移除用户。只有数据库超级用户才能使用这条命令。把一个用户添加到组中并不会创建该用户。类似地，
从组中移除一个用户也不会删除该用户本身。

第三种变体更改组的名称。只有数据库超级用户才能重命名组。

参数

groupname

要修改的组的名称。

username

要添加到组中或从组中移除的用户。这些用户必须已经存在。

newname

组的新名称。

示例

向组中添加用户：

```
ALTER GROUP staff ADD USER karl, john;
```

从组中移除一个用户：

```
ALTER GROUP workers DROP USER beth;
```

兼容性

SQL 标准中没有`ALTER GROUP`语句。角色的概念与之相似。

另见

CREATE GROUP, DROP GROUP

ALTER LANGUAGE

ALTER LANGUAGE — 更改一个过程语言的定义

大纲

`ALTER LANGUAGE` *`name`* `RENAME TO` *`newname`*

描述

`ALTER LANGUAGE`更改一个语言的定义。它唯一的功能是重命名该语言。只有超级用户才能重命名语言。

参数

`name`

一个语言的名称

`newname`

该语言的新名称

兼容性

SQL 标准中没有`ALTER LANGUAGE`语句。

另请参阅

CREATE LANGUAGE, DROP LANGUAGE

ALTER OPERATOR CLASS

ALTER OPERATOR CLASS — 更改一个操作符类的定义

大纲

`ALTER OPERATOR CLASS` *`name`* `USING` *`index_method`* `RENAME TO` *`newname`*

描述

`ALTER OPERATOR CLASS`更改一个操作符类的定义。它唯一的功能就是重命名操作符类。

参数

`name`

一个已有操作符类的名字（可以用模式限定）。

`index_method`

该操作符类所针对的索引方法的名字。

`newname`

操作符类的新名字。

兼容性

SQL 标准中没有`ALTER OPERATOR CLASS` 语句。

另见

CREATE OPERATOR CLASS, DROP OPERATOR CLASS

ALTER SCHEMA

ALTER SCHEMA — 更改一个模式的定义

大纲

`ALTER SCHEMA` *`name`* `RENAME TO` *`newname`*

描述

`ALTER SCHEMA` 更改一个模式的定义。唯一的功能是重命名该模式。要重命名一个模式，你必须拥有该模式并且拥有数据库的 `CREATE` 权限。

参数

`name`

一个模式的名称

`newname`

该模式的新名称

兼容性

SQL 标准中没有 `ALTER SCHEMA` 语句。

另见

CREATE SCHEMA, DROP SCHEMA

ALTER SEQUENCE

ALTER SEQUENCE — 更改序列发生器的定义

大纲

```
ALTER SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO
 MAXVALUE ]
    [ RESTART [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

描述

`ALTER SEQUENCE` 更改现有序列发生器的参数。在 `ALTER SEQUENCE` 命令中未明确设置的参数，会保留其原有设置。

参数

name

要更改的序列名称（可以是模式限定的）。

increment

子句`INCREMENT BY` *increment*是可选的。正值会创建升序序列，负值会创建降序序列。如果未指定，则保留原有的增量值。

minvalue
`NO MINVALUE`

可选子句`MINVALUE` *minvalue*确定序列可生成的最小值。如果指定了`NO MINVALUE`，则升序序列和降序序列将分别使用默认值 1 和 $-2^{63}-1$。如果两种选项都未指定，则保留当前最小值。

maxvalue
`NO MAXVALUE`

可选子句`MAXVALUE` *maxvalue*确定序列的最大值。如果指定了`NO MAXVALUE`，则升序序列和降序序列将分别使用默认值 $2^{63}-1$ 和 -1。如果两种选项都未指定，则保留当前最大值。

start

可选子句`RESTART WITH` *start*更改序列的当前值。

cache

子句`CACHE` *cache*启用预分配序列号并将其存储在内存中，以便更快访问。最小值为 1（一次只能生成一个值，即不缓存）。如果未指定，则保留旧的缓存值。

CYCLE

可选的`CYCLE`关键字可用于启用序列回绕：当递增或递减序列分别达到 *maxvalue*或 *minvalue*时回绕。如果达到限制，下一个生成的数将分别为 *minvalue*或 *maxvalue*。

NO CYCLE

如果指定了可选关键字NO CYCLE，则当序列达到其最大值后，任何对 nextval的调用都会返回错误。如果既未指定CYCLE也未指定 NO CYCLE，则保留原有的循环行为。

示例

从 105 开始，重新启动一个名为 serial 的序列：

```
ALTER SEQUENCE serial RESTART WITH 105;
```

注解

为避免阻塞从同一序列获取号码的并发事务，ALTER SEQUENCE 从不回滚；更改立即生效且不可逆。

ALTER SEQUENCE 不会立即影响当前后端之外的其他后端中已预分配（缓存）了序列值的 nextval结果。这些后端会在注意到序列参数更改之前用完所有缓存的值。当前后端会立即受到影响。

兼容性

SQL99

ALTER SEQUENCE 是 PostgreSQL 的一种语言扩展。SQL99 中没有 ALTER SEQUENCE 语句。

ALTER TABLE

ALTER TABLE — 更改一个表的定义

大纲

```
ALTER TABLE [ ONLY ] name [ * ]
    ADD [ COLUMN ] column type [ column_constraint [ ... ] ]
ALTER TABLE [ ONLY ] name [ * ]
    DROP [ COLUMN ] column [ RESTRICT | CASCADE ]
ALTER TABLE [ ONLY ] name [ * ]
    ALTER [ COLUMN ] column { SET DEFAULT expression | DROP DEFAULT }
ALTER TABLE [ ONLY ] name [ * ]
    ALTER [ COLUMN ] column { SET | DROP } NOT NULL
ALTER TABLE [ ONLY ] name [ * ]
    ALTER [ COLUMN ] column SET STATISTICS integer
ALTER TABLE [ ONLY ] name [ * ]
    ALTER [ COLUMN ] column SET STORAGE { PLAIN | EXTERNAL | EXTENDED
 | MAIN }
ALTER TABLE [ ONLY ] name [ * ]
    SET WITHOUT OIDS
ALTER TABLE [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column TO new_column
ALTER TABLE name
    RENAME TO new_name
ALTER TABLE [ ONLY ] name [ * ]
    ADD table_constraint
ALTER TABLE [ ONLY ] name [ * ]
    DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
ALTER TABLE name
    OWNER TO new_owner
ALTER TABLE name
    CLUSTER ON index_name
```

描述

`ALTER TABLE` 更改一个现有表的定义。它有若干子形式：

`ADD COLUMN`

这种形式使用与CREATE TABLE相同的语法向表添加一个新列。

`DROP COLUMN`

这种形式从表中删除一个列。涉及该列的索引和表约束也将被自动删除。
如果表外的任何对象依赖于该列（例如外键引用或视图），则需要指定 `CASCADE`。

`SET/DROP DEFAULT`

这些形式为列设置或移除默认值。默认值只会应用于后续的 `INSERT` 命令；它们不会导致表中已有的行发生变化。也可以为视图创建默认值，这种情况下，在应用视图的 `ON INSERT` 规则之前，这些默认值会被插入到视图上的 `INSERT` 语句中。

SET/DROP NOT NULL

这些形式更改列是否被标记为允许空值，或拒绝空值。只有当列不包含空值时，才能使用SET NOT NULL。

SET STATISTICS

该形式为后续ANALYZE操作设置每列的统计信息收集目标。目标可以设置在 0 到 1000 范围内；也可以将其设置为 -1，以恢复使用系统默认的统计目标。

SET STORAGE

该形式为列设置存储模式。这控制该列是内联保存还是保存在辅助表中，
以及是否压缩数据。对于integer等定长值，必须使用PLAIN，数据以内联、
未压缩方式保存。MAIN用于内联、可压缩的数据。EXTERNAL用于外部、未压缩的数据，而EXTENDED用于外部、压缩的数据。对于支持它的所有数据类型，EXTENDED是默认值。
使用EXTERNAL（例如）会使对text列的子串操作更快，但代价是占用更多的存储空间。

SET WITHOUT OIDS

这种形式从表中移除 oid 列。从表中移除 OID 并不会立即发生。OID 使用的空间会在行被更新时回收。如果不更新行，空间和 OID 的值都会无限期保留。这在语义上类似于 DROP COLUMN 过程。

RENAME

RENAME 形式更改表（或索引、序列或视图）的名称，或者更改表中某一列的名称。存储的数据不受影响。

ADD *table_constraint*

这种形式使用与CREATE TABLE相同的语法向一个表增加一个新约束。

DROP CONSTRAINT

该形式删除表上的约束。目前，并不要求表上的约束具有唯一的名称，
因此可能有多个约束与指定的名称匹配。所有这样的约束都将被删除。

OWNER

该形式把表、索引、序列或视图的所有者更改为指定用户。

CLUSTER

该形式为将来的CLUSTER操作标记表。

要使用 ALTER TABLE，你必须拥有该表；只有 ALTER TABLE OWNER 例外，它只能由超级用户执行。

参数

name

要更改的现有表的名称（可带模式限定）。如果在表名之前指定ONLY，则只更改该表。
如果未指定ONLY，则更改该表及其所有后代表（如果有）。也可以在表名后指定*
来指示要更改后代表，但在当前版本中这是默认行为。（在 7.1 之前的版本中，ONLY 是默认行为。默认行为可以通过更改配置参数 sql_inheritance 来改变。）

column

新列或现有列的名称。

type

新列的数据类型。

new_column

现有列的新名称。

new_name

表的新名称。

table_constraint

表的新表约束。

constraint_name

要删除的现有约束的名称。

new_owner

表的新所有者的用户名。

index_name

表将为之标记聚簇的索引名称。

CASCADE

自动删除依赖于被删除列或约束的对象（例如引用该列的视图）。

RESTRICT

如果存在任何依赖对象，则拒绝删除列或约束。这是默认行为。

注意

关键字COLUMN只是噪声，可以省略。

在当前 ADD COLUMN 的实现中，不支持新列的默认值和 NOT NULL 子句。新列总是以所有值为空的状态产生。可以在之后使用 ALTER TABLE 的 SET DEFAULT 形式设置默认值。（你可能还想用UPDATE把已有的行更新为新默认值。）如果想要把列标记为非空，可以先为该列在所有行中输入非空值，然后使用 SET NOT NULL 形式。

DROP COLUMN形式不会从物理上删除列，而只是使其对 SQL 操作不可见。表后续的插入和更新操作会为该列存储空值。因此，删除列的速度很快，但不会立即减少表在磁盘上的大小，因为被删除列占用的空间不会被回收。随着现有行被更新，空间会逐渐回收。要立即回收空间，可以对所有行做一次虚拟的 UPDATE 然后清理（vacuum），例如：

```
UPDATE table SET col = col;
VACUUM FULL table;
```

如果表有任何后代表，则不允许只在父表中添加或重命名列，而不对后代表执行相同操作。也就是说，ALTER TABLE ONLY会被拒绝。这确保后代表始终拥有与父表匹配的列。

只有当某个后代表中的列既不是从其他任何父表继承而来，也从未有过该列的独立定义时，递归的DROP COLUMN操作才会移除该后代表中的此列。非递归的DROP COLUMN（即ALTER TABLE ONLY ... DROP COLUMN）永远不会移除任何后代列，而只会把它们标记为独立定义，而非继承得到。

不允许更改系统目录表的任何部分。

有关有效参数的进一步说明，请参见CREATE TABLE。第 5 章中还有关于继承的更多信息。

示例

要添加类型为varchar的列到表中：

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

要从表中删除一列：

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

要重命名一个现有列：

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

重命名一个现有的表：

```
ALTER TABLE distributors RENAME TO suppliers;
```

为一列增加一个非空约束：

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

从一列移除一个非空约束：

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

要向一个表增加一个检查约束：

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(
zipcode) = 5);
```

要从一个表及其所有后代移除一个检查约束：

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

为一个表增加一个外键约束：

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address)
 REFERENCES addresses (address) MATCH FULL;
```

为一个表增加一个（多列）唯一约束：

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (
dist_id, zipcode);
```

为一个表增加一个自动命名的主键约束，注意一个表只能拥有一个主键：

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

兼容性

`ADD COLUMN` 形式符合 SQL 标准，但不支持默认值和非空约束，如上所述。`ALTER COLUMN` 形式完全符合标准。

重命名表、列、索引、视图和序列的子句是 PostgreSQL 对 SQL 标准的扩展。

`ALTER TABLE DROP COLUMN`可以被用来删除一个表的唯一的列，从而留下一个零列的表。这是一种对 SQL 的扩展，SQL 中不允许零列的表。

ALTER TRIGGER

ALTER TRIGGER — 更改一个触发器的定义

大纲

```
ALTER TRIGGER name ON table RENAME TO newname
```

描述

`ALTER TRIGGER`更改一个现有触发器的属性。`RENAME`子句更改给定触发器的名称，而不改变触发器的其他定义。

你必须拥有触发器所作用的表才能更改它的属性。

参数

name

要更改的现有触发器的名称。

table

该触发器所作用的表的名称。

newname

触发器的新名称。

示例

重命名一个现有的触发器：

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

兼容性

`ALTER TRIGGER`是PostgreSQL 对 SQL 标准的扩展。

ALTER USER

ALTER USER — 更改一个数据库用户帐号

大纲

```
ALTER USER name [ [ WITH ] option [ ... ] ]

where option can be:

    [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | CREATEDB | NOCREATEDB
    | CREATEUSER | NOCREATEUSER
    | VALID UNTIL 'abstime'

ALTER USER name RENAME TO newname

ALTER USER name SET parameter { TO | = } { value | DEFAULT }
ALTER USER name RESET parameter
```

描述

`ALTER USER` 用于更改 PostgreSQL 用户账户的属性。命令中未提及的属性保持其原有设置。

语法中的第一种变体更改某些全局用户权限和认证设置。（细节见下文。）
只有数据库超级用户才能用此命令更改这些权限和口令过期。普通用户只能更改自己的口令。

第二种变体更改用户的名称。只有数据库超级用户才能重命名用户账户。
会话用户不能被重命名。（如果你需要这样做，请以其他用户连接。）

第三种和第四种变体更改用户对指定配置变量的会话默认值。之后每当该用户启动新会话时，指定的值就成为会话默认值，覆盖 `postgresql.conf` 中存在的或从 `postmaster` 命令行接收到的任何设置。普通用户可以更改自己的会话默认值。超级用户可以更改任何人的会话默认值。

参数

name

要更改其属性的用户名。

password

此帐号要使用的新密码。

ENCRYPTED
UNENCRYPTED

这些关键字控制密码是否以加密形式存储在`pg_shadow`中。（有关这一选择的更多信息，参见CREATE USER。）

CREATEDB
NOCREATEDB

这些子句定义用户创建数据库的能力。如果指定了 `CREATEDB`，该用户将被允许创建自己的数据库。使用 `NOCREATEDB` 将拒绝用户创建数据库的能力。

`CREATEUSER`
`NOCREATEUSER`

这些子句决定是否允许用户自己创建新用户。
此选项还会把该用户变为可以越过所有访问限制的超级用户。

abstime

该用户密码将过期失效的日期（以及可选的时间）。要让密码永不过期，请使用 `'infinity'`。

newname

该用户的新名称。

parameter
value

把该用户对指定配置参数的会话默认值设为给定值。如果 *value* 为 `DEFAULT`，或等价地使用了 `RESET`，则移除该用户专有的变量设置，用户在新会话中将继承系统范围的默认设置。使用 `RESET ALL` 可清除所有设置。

有关允许的参数名和值的更多信息，请参见SET和第 16.4 节。

注意

使用CREATE USER添加新用户，使用DROP USER删除用户。

`ALTER USER`不能更改用户的组成员资格。要做到这一点，请使用ALTER GROUP。

使用ALTER DATABASE，也可以把会话默认值绑定到特定数据库而不是用户。

示例

更改用户口令：

```
ALTER USER davide WITH PASSWORD 'hu8jmn3';
```

更改用户的有效期限（valid until）日期：

```
ALTER USER manuel VALID UNTIL 'Jan 31 2030';
```

更改用户的有效期限日期，指定他的授权应在 2005 年 5 月 4 日正午、使用比 UTC 早一小时的时区过期：

```
ALTER USER chris VALID UNTIL 'May 4 12:00:00 2005 +1';
```

让用户永久有效：

```
ALTER USER fred VALID UNTIL 'infinity';
```

授予用户创建其他用户和新数据库的能力：

```
ALTER USER miriam CREATEUSER CREATEDB;
```

兼容性

`ALTER USER`语句是 PostgreSQL的一种扩展。SQL 标准将用户的定义交由具体实现决定。

另请参阅

CREATE USER, DROP USER, SET

ANALYZE

ANALYZE — 收集关于数据库的统计信息

大纲

ANALYZE [VERBOSE] [*table* [(*column* [, ...])]]

描述

ANALYZE收集数据库中表内容的统计信息，并把结果存储在系统表pg_statistic中。随后，查询规划器使用这些统计信息来帮助确定查询最有效的执行计划。

不带参数时，ANALYZE检查当前数据库中的每个表。带参数时，ANALYZE只检查该表。还可以给出一个列名列表，这种情况下只收集这些列的统计信息。

参数

VERBOSE

启用进度消息的显示。

table

要分析的一个指定表的名称（可以是模式限定的）。默认为当前数据库中的所有表。

column

要分析的一个指定列的名称。默认为所有列。

输出

指定VERBOSE时，ANALYZE会发出进度消息，指明当前正在处理哪个表。还会打印关于表的各种统计信息。

注解

定期运行ANALYZE，或者在一个表的内容发生重大变化之后紧接着运行它，是个好主意。准确的统计信息能帮助规划器选择最合适的查询计划，从而提高查询处理的速度。常见的策略是每天在使用率低的时段运行一次VACUUM和ANALYZE。

与VACUUM FULL不同，ANALYZE 只需要对目标表的读锁，因此它可以与该表上的其他活动并行运行。

ANALYZE收集的统计信息通常包括每个列中一些最常见值的列表，以及显示每个列中近似数据分布的直方图。如果 ANALYZE认为它们没有意义（例如在唯一键列中没有常见值），或者列数据类型不支持适当的操作符，则可以省略其中之一或两者。关于统计信息的更多信息见第 21 章。

对于大型表，ANALYZE对表内容抽取随机样本，而不是检查每一行。这使得即使非常大的表也能在很短时间内完成分析。但要注意，统计信息只是近似的，

即使表的实际内容没有变化，每次运行ANALYZE的结果也会略有不同。这可能导致EXPLAIN显示的规划器估计代价发生小的变化。在罕见的情况下，这种非确定性会导致查询优化器在两次ANALYZE运行之间选择不同的查询计划。要避免这一点，可以增大ANALYZE收集的统计信息量，方法如下所述。

分析的深度可以通过调整DEFAULT_STATISTICS_TARGET 参数变量来控制，也可以逐列地用ALTER TABLE ... ALTER COLUMN ... SET STATISTICS设置每列的统计目标来控制（见ALTER TABLE）。目标值设定了最常见值列表中的最大条目数和直方图中的最大桶数。默认的目标值是 10，可以调高或调低，以便在规划器估计的准确性与ANALYZE所花费的时间以及pg_statistic中占用的空间之间做出权衡。特别地，
把统计目标设置为零将禁用该列的统计信息收集。对于从不用作查询的WHERE、GROUP BY或ORDER BY子句部分的列，这样做可能有用，因为规划器不会用到这类列的统计信息。

被分析各列中最大的统计目标决定了为准备统计信息而采样的表行数。
提高目标会使执行ANALYZE所需的时间和空间成比例地增加。

兼容性

SQL 标准中没有ANALYZE语句。

BEGIN

BEGIN — 开始一个事务块

大纲

```
BEGIN [ WORK | TRANSACTION ]
```

描述

`BEGIN` 开启一个事务块，也就是说，`BEGIN` 命令之后的所有语句都将在单个事务中执行，直到给出显式的COMMIT或ROLLBACK为止。默认情况下（不带 `BEGIN`），PostgreSQL 以“autocommit”（自动提交）模式执行事务，也就是说，每条语句都在它自己的事务中执行，并且在语句结束时隐式地执行一次提交（如果执行成功，否则执行一次回滚）。

在事务块中语句执行得更快，因为事务的启动/提交需要可观的 CPU 和磁盘活动。在一个事务中执行多条语句也有助于在做多个相关更改时确保一致性：其他会话将无法看到相关更新尚未全部完成的中间状态。

参数

`WORK`
`TRANSACTION`

可选的关键字。它们没有效果。

注意

START TRANSACTION具有与 `BEGIN` 相同的功能。

使用COMMIT或ROLLBACK终止一个事务块。

在已经处于一个事务块中时发出 `BEGIN` 会引发一条警告消息。事务的状态不受影响。

示例

开始一个事务块：

```
BEGIN;
```

兼容性

`BEGIN` 是一种 PostgreSQL 语言扩展。SQL 标准中没有显式的 `BEGIN` 命令；事务的开启总是隐式的，它以 `COMMIT` 或 `ROLLBACK` 语句终止。

其他关系数据库系统可能出于便利提供自动提交特性。

顺便一提，`BEGIN` 关键字在嵌入式 SQL 中用于不同的目的。在移植数据库应用程序时，建议注意事务语义。

另见

COMMIT, ROLLBACK

CHECKPOINT

CHECKPOINT — 强制执行一次事务日志检查点

大纲

```
CHECKPOINT
```

描述

预写式日志（WAL）会每隔一段时间在事务日志中设置一个检查点。（要调整自动检查点间隔，请参见运行时配置选项 `checkpoint_segments` 和 `checkpoint_timeout`。）`CHECKPOINT` 命令会在发出该命令时强制立即执行检查点，而不等待预定的检查点。

检查点是事务日志序列中的一个位置，到这一位置时，所有数据文件都已更新，以反映日志中的信息。所有数据文件都会被刷出到磁盘。有关 WAL 系统的更多信息，参见第 25 章。

只有超级用户才能调用 `CHECKPOINT`。该命令并非设计用于正常运行期间。

兼容性

`CHECKPOINT`命令是 PostgreSQL的一种语言扩展。

CLOSE

CLOSE — 关闭一个游标

大纲

```
CLOSE name
```

描述

CLOSE释放与一个打开的游标相关联的资源。游标关闭后，不允许再对其执行任何后续操作。当不再需要游标时，应将其关闭。

每个不可保持的打开游标都会在事务通过COMMIT或 ROLLBACK结束时被隐式关闭。可保持游标会在创建它的事务通过ROLLBACK中止时被隐式关闭。如果创建它的事务成功提交，则该可保持游标会一直保持打开状态，直到显式执行 CLOSE，或者客户端断开连接。

参数

name

要关闭的打开游标的名称。

注意

PostgreSQL 没有显式的 OPEN 游标语句；游标在声明时即被视为打开。请使用 DECLARE 语句声明游标。

示例

关闭游标liahona：

```
CLOSE liahona;
```

兼容性

CLOSE 完全符合 SQL 标准。

CLUSTER

CLUSTER — 根据一个索引对某个表进行聚簇

大纲

```
CLUSTER indexname ON tablename
CLUSTER tablename
CLUSTER
```

描述

CLUSTER指示 PostgreSQL基于 *indexname*所指定的索引，对 *tablename*所指定的表进行聚簇。该索引必须已经定义在 *tablename*上。

当一个表被聚簇时，它会基于索引信息被物理地重新排序。
聚簇是一次性操作：当表随后被更新时，这些更改不会被聚簇。也就是说，
系统不会尝试按照索引顺序存储新的或更新过的行。如果需要，
可以通过再次发出该命令来周期性地重新聚簇。

当一个表被聚簇时，PostgreSQL会记住它是基于哪个索引聚簇的。CLUSTER *tablename*形式会按照该表先前聚簇所用的同一个索引重新聚簇该表。

不带任何参数的CLUSTER
会重新聚簇当前数据库中调用用户所拥有的所有之前被聚簇过的表（如果是超级用户调用，
则是所有表）。（从未聚簇过的表不包括在内。）这种形式的CLUSTER
不能在事务或函数内部调用。

当一个表正在被聚簇时，会在其上获取ACCESS EXCLUSIVE锁。这将阻止任何其他数据库操作（读和写）在该表上执行，直到CLUSTER完成。

参数

indexname

一个索引的名称。

tablename

一个表的名称（可以有模式限定）。

注意

CLUSTER会丢失元组的所有可见性信息，这使得该表在任何在CLUSTER
命令完成之前获取的快照看来是空的。这使得CLUSTER不适合在访问被聚簇表的事务与CLUSTER
并发运行的应用中使用。这在可串行化事务中最为明显，因为它们只在事务开始时取一个快照，
但读已提交事务也受影响。

如果你在表中随机访问单个行，数据在表中的实际顺序并不重要。
但如果你倾向于比其他数据更常访问某些数据，并且有一个索引把它们聚集在一起，使用
CLUSTER会带来好处。如果你从一个表中请求某个索引值范围，
或者一个有多行匹配的单个索引值，CLUSTER会有帮助，

因为一旦索引识别出第一行匹配所在的堆页，所有其他匹配的行很可能已经在同一个堆页上，这样就节省了磁盘访问并加快了查询。

在聚簇操作期间，会创建表的一个临时副本，其中按索引顺序包含表数据。表上每个索引的临时副本也会被创建。因此，你需要的空闲磁盘空间至少等于表大小与索引大小之和。

因为`CLUSTER`会记住聚簇信息，可以先在第一次手动聚簇想要聚簇的表，然后设置一个类似`VACUUM`的定时事件，让这些表被周期性地重新聚簇。

因为规划器会记录关于表排序的统计信息，建议在新聚簇的表上运行`ANALYZE`。否则，规划器可能会做出糟糕的查询计划选择。

还有另一种聚簇数据的方法。`CLUSTER`命令使用你指定的索引的顺序对原表重新排序。在大表上这可能很慢，因为行是按索引顺序从堆中取出的，如果堆表是无序的，各项就散布在随机页面上，因此每移动一行就要检索一个磁盘页。（PostgreSQL有缓存，但大表的大部分放不进缓存。）聚簇一个表的另一种方法是使用

```
CREATE TABLE newtable AS
    SELECT columnlist FROM table ORDER BY columnlist;
```

它在`ORDER BY`子句中使用 PostgreSQL的排序代码来产生想要的顺序；对于无序数据，这通常比索引扫描快得多。然后你删除旧表，使用 `ALTER TABLE ... RENAME`把 *`newtable`*改名为旧名称，并重建该表的索引。不过，这种方法不保留 OID、约束、外键关系、已授予的权限以及表的其他附属属性——所有这些项都必须手动重建。

示例

基于表 `employees` 的索引 `emp_ind` 对该表进行聚簇：

```
CLUSTER emp_ind ON emp;
```

使用与之前相同的索引聚簇 `employees` 关系：

```
CLUSTER emp;
```

聚簇数据库中所有之前被聚簇过的表：

```
CLUSTER;
```

兼容性

SQL 标准中没有`CLUSTER`语句。

另见

clusterdb

COMMENT

COMMENT — 定义或修改对象的注释

大纲

```
COMMENT ON
{
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type) |
  CONSTRAINT constraint_name ON table_name |
  DATABASE object_name |
  DOMAIN object_name |
  FUNCTION func_name (arg1_type, arg2_type, ...) |
  INDEX object_name |
  OPERATOR op (leftoperand_type, rightoperand_type) |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name
} IS 'text'
```

描述

`COMMENT` 存储关于某个数据库对象的注释。注释可以用 psql 的命令 `\dd`、`\d+` 和 `\l+` 轻松检索。其他检索注释的用户界面可以构建在 psql 所用的相同内置函数之上，即 `obj_description` 和 `col_description`。

要修改一个注释，可以对同一对象发出一条新的`COMMENT`命令。
每个对象只存储一个注释字符串。要移除一个注释，用`NULL`代替文本字符串。对象被删除时，
其注释也会被自动删除。

参数

object_name
table_name.column_name
aggname
constraint_name
func_name
op
rule_name
trigger_name

要加注释的对象的名称。表、聚合、域、函数、索引、操作符、序列、
类型和视图的名称可以带模式限定。

text

新的注释。

注意

目前注释没有任何安全机制：任何连接到数据库的用户都能看到该数据库中对象的全部注释（不过只有超级用户才能修改他们不拥有的对象的注释）。因此，
不要把安全攸关的信息放在注释里。

示例

为表mytable附加一条注释：

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

再将其移除：

```
COMMENT ON TABLE mytable IS NULL;
```

更多示例：

```
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes
 sample variance';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman
 Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two
 texts';
COMMENT ON OPERATOR ^ (NONE, text) IS 'This is a prefix operator on
 text';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee
 records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

兼容性

SQL 标准中没有COMMENT命令。

COMMIT

COMMIT — 提交当前事务

大纲

```
COMMIT [ WORK | TRANSACTION ]
```

描述

`COMMIT`提交当前事务。该事务所做的所有更改都会对其他会话可见，并且如果发生崩溃，也能保证其持久性。

参数

`WORK`
`TRANSACTION`

可选关键字，没有任何作用。

注解

使用ROLLBACK中止一个事务。

在事务之外发出 `COMMIT` 不会造成危害，但会产生一条警告消息。

示例

提交当前事务，并使所有更改永久生效：

```
COMMIT;
```

兼容性

SQL 标准只规定了 `COMMIT` 和 `COMMIT WORK` 这两种形式。在其他方面，此命令完全符合标准。

另见

BEGIN, ROLLBACK

COPY

COPY — 在文件和表之间复制数据

大纲

```
COPY tablename [ ( column [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ] ]

COPY tablename [ ( column [, ...] ) ]
    TO { 'filename' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ] ]
```

描述

`COPY`在 PostgreSQL表与标准文件系统文件之间传输数据。`COPY TO`将表的内容复制到文件，而`COPY FROM`则将数据从文件复制到表中（追加到表中已有的数据之后）。

如果指定了列列表，`COPY`将只在指定列与文件之间复制数据。如果表中存在未列入列列表的列，`COPY FROM`将为这些列插入默认值。

带文件名的`COPY`会指示 PostgreSQL 服务器直接从文件读取或向文件写入。该文件必须可由服务器访问，并且其名称必须从服务器的视角指定。当指定`STDIN`或`STDOUT`时，数据通过客户端与服务器之间的连接传输。

参数

tablename

一个现有表的名称（可以是模式限定的）。

column

要复制的可选列列表。如果没有指定列列表，则会使用所有列。

filename

输入或输出文件的绝对路径名。

`STDIN`

指定输入来自客户端应用程序。

STDOUT

指定输出发送到客户端应用程序。

BINARY

使所有数据以二进制而不是文本格式存储或读取。在二进制模式下不能指定 DELIMITER 或 NULL 选项。

OIDS

指定复制每一行的 OID。（如果为没有 OID 的表指定了 OIDS，就会报错。）

delimiter

分隔文件每行（一行数据）中各列的单个字符。默认是制表符。

null string

表示空值的字符串。默认是 \N（反斜线-N）。例如你可能更喜欢空字符串。

> **注意**
>
> 在 COPY FROM 中，任何匹配此字符串的数据项都会被存储为空值，所以应确保使用与 COPY TO 相同的字符串。

注解

COPY只能用于普通表，不能用于视图。

BINARY关键字使所有数据以二进制格式而不是文本格式存储/读取。它比普通文本模式略快，但二进制格式的文件在跨机器体系结构和PostgreSQL版本之间的可移植性较差。

你必须对COPY TO读取其值的表具有 SELECT权限，并对COPY FROM 插入其值的表具有INSERT权限。

COPY 命令中命名的文件由服务器直接读取或写入，而不是由客户端应用进行。因此，这些文件必须位于数据库服务器机器上或可由其访问，而不是客户端上。它们必须可由 PostgreSQL 用户（服务器运行时使用的用户 ID）访问并可读或可写，而不是由客户端访问。带有文件名的 COPY 只允许数据库超级用户使用，因为它允许读取或写入服务器有权访问的任何文件。

不要把 COPY 与 psql 的指令 \copy 混淆。\copy 会调用 COPY FROM STDIN 或 COPY TO STDOUT，然后在 psql 客户端可访问的文件中获取/存储数据。因此，使用 \copy 时，文件的可访问性和访问权限取决于客户端而不是服务器。

建议在COPY中使用的文件名始终指定为绝对路径。对于COPY TO，服务器会强制这一点；但对于COPY FROM，你仍可选择从使用相对路径指定的文件中读取。该路径将相对于服务器进程的工作目录（位于数据目录下某处）而非客户端的工作目录进行解释。

COPY FROM将调用目标表上的任何触发器和检查约束。但是它不会调用规则。

COPY 会在遇到第一个错误时停止操作。对于 COPY TO，这应当不会导致问题；但对于COPY FROM，目标表此时已经接收了前面的行。这些行不可见也不可访问，但仍占用磁盘空间。

如果在一次大型复制操作已经进行很久后才失败，可能会浪费大量磁盘空间。可以调用 VACUUM 回收这些浪费的空间。

文件格式

文本格式

当 COPY 不带 BINARY 选项使用时，读取或写入的数据是一个文本文件，每个表行占一行。一行中的各列由分隔字符分隔。
列值本身是由各属性数据类型的输出函数生成（或可被输入函数接受）的字符串。
指定的空字符串用于代替为空的列。如果输入文件的任何一行包含多于或少于预期的列数，COPY FROM 将报错。如果指定了 OIDS，OID 将作为第一列读取或写入，位于用户数据列之前。

数据结束可以表示为只包含反斜线加点号（\.）的一行。从文件读取时，不需要数据结束标记，因为文件结束已足够；只有在使用 3.0 之前版本的客户端协议、向客户端应用程序复制数据或从其复制数据时，才需要该标记。

反斜线字符（\）可用于 COPY 数据中引用否则会被当作行或列分隔符的数据字符。特别是，以下字符如果作为列值的一部分出现，前面必须有反斜线：反斜线自身、换行符、
回车符和当前的分隔字符。

COPY TO输出指定的空值串时不会添加任何反斜线；相反，COPY FROM会在去除反斜线之前先将输入与空值串进行匹配。因此，像\N这样的空值串不会与实际的数据值\N混淆，
因为后者会表示为\\N。

COPY FROM识别下列特殊的反斜线序列：

序列	表示
\b	退格 (ASCII 8)
\f	换页 (ASCII 12)
\n	新行 (ASCII 10)
\r	回车 (ASCII 13)
\t	制表 (ASCII 9)
\v	纵向制表 (ASCII 11)
digits	反斜线后跟一到三个八进制数字表示该数字代码对应的字节

目前，COPY TO从不会输出八进制数字反斜线序列，但对这些控制字符确实会使用上表列出的其他序列。

任何上表中未提到的其他反斜线字符都将表示其自身。不过，要注意不要不必要地添加反斜线，因为那可能意外地产生与数据结束标记（\.）或空值串（默认是\N）匹配的字符串。
这些字符串会在进行任何其他反斜线处理之前先被识别出来。

强烈建议生成 COPY 数据的应用程序把数据中的换行符和回车符分别转换为 \n 和 \r 序列。目前可以用反斜线加回车符表示数据回车符，用反斜线加换行符表示数据换行符。但这些表示在未来的版本中可能不被接受。如果 COPY 文件在不同机器之间传输（例如从 Unix 到 Windows 或反之），它们也非常容易损坏。

COPY TO 将以 Unix 风格的换行符（“\n”）结束每一行。运行在 MS Windows 上的服务器则输出回车符/换行符（“\r\n”），但仅当 COPY 到服务器文件时；为了跨平台一致

性，无论服务器平台如何，COPY TO STDOUT 总是发送“\n”。COPY FROM 可以处理以换行符、回车符或回车符/换行符结尾的行。为了降低本应作为数据的未反斜线转义的换行符或回车符导致错误的风险，如果输入中的行结尾不全部相同，COPY FROM 将提出警告。

二进制格式

COPY BINARY所使用的文件格式在 PostgreSQL 7.4 中发生了变化。新格式由一个文件头、零个或多个包含行数据的元组以及一个文件尾组成。头部和数据现在都采用网络字节序。

文件头

文件头由 15 字节的固定字段组成，其后跟着变长的头部扩展区。固定字段有：

签名

11 字节序列 PGCOPY\n\377\r\n\0 --- 注意，零字节是签名的必要组成部分。（签名的设计目的是便于识别被非 8 位干净传输破坏的文件。行尾翻译过滤器、丢弃的零字节、丢弃的高位或奇偶校验位的改变都会改变这个签名。）

标志域

这是一个 32 位整数位掩码，用于表示文件格式的重要属性。位编号从 0（LSB）到 31（MSB）。注意，此字段与文件格式中的所有整数字段一样，采用网络字节序存储（最高有效字节在前）。位 16-31 保留用于表示文件格式的关键问题；如果读取程序发现这个范围内有非预期的位被置位，就应中止。位 0-15 保留用于表示向后兼容的格式问题；读取程序应直接忽略这个范围内非预期的置位。目前仅定义了一个标志位，其余位必须为零：

位 16

如果为 1，则数据中包含 OID；如果为 0，则不包含。

头部扩展区长度

32 位整数，表示头部剩余部分的长度（以字节计），不包括该字段本身。当前该值为零，因此其后紧接着第一个元组。未来对这种格式的更改可能允许在头部中包含额外数据。
如果读取程序不知道如何处理头部扩展区数据，应静默跳过它。

头部扩展区被设想为包含一系列可自我标识的块。
标志域并不用于告诉读取程序扩展区中包含哪些内容。
头部扩展内容的具体设计留待后续版本决定。

这种设计既允许向后兼容的头部新增（增加头部扩展块，或设置低位标志位），
也允许不向后兼容的更改（设置高位标志位来表明这类更改，
并在需要时向扩展区增加支持数据）。

元组

每个元组都以一个 16 位整数计数开头，用于表示该元组中的字段数。（目前，一个表中的所有元组都应有相同的计数，但这未必永远如此。）随后，对元组中的每个字段，都会有一个 32 位长度字，后跟该字段数据的相应字节数。（长度字不包括其本身，且可以为零。）特殊情况下，-1 表示一个 NULL 字段值；在 NULL 情况下，后面不会跟随任何值字节。

字段之间没有对齐填充或任何其他额外数据。

当前，COPY BINARY文件中的所有数据值都假定为二进制格式（格式代码一）。可以预见，未来的扩展可能会增加一个允许为各列分别指定格式代码的头部字段。

要确定实际元组数据应采用的二进制格式，你应该参考 PostgreSQL源码，特别是各列数据类型对应的`*send`和`*recv`函数（这些函数通常可以在源码分发包的`src/backend/utils/adt/`目录中找到）。

如果文件中包含 OID，OID 字段紧跟在字段计数字之后。它是一个正常字段，只是不计入字段计数。特别地，它有一个长度字——这使处理 4 字节与 8 字节 OID 不至于太痛苦，并且如果将来需要的话还允许 OID 显示为空。

文件尾

文件尾由一个值为 -1 的 16 位整数构成。这很容易与元组的字段计数字区分开来。

如果字段计数字既不是 -1 也不是预期的列数，读取程序应报告错误。这提供了一项额外检查，以防与数据失去同步。

示例

下面的示例使用竖线（`|`）作为字段分隔符将一个表复制到客户端：

```
COPY country TO STDOUT WITH DELIMITER '|';
```

要将文件中的数据复制到`country`表中：

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

下面给出适合从`STDIN`复制到表中的示例数据：

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

注意每一行中的空白实际上是一个制表符。

下面是用二进制格式输出的相同数据。该数据是用 Unix 工具 `od -c`过滤后显示的。该表具有三列，第一列类型是`char(2)`，第二列类型是`text`，第三列类型是`integer`。
所有行在第三列都是空值。

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0
 \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013
 A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0
 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N
 I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0
 \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0
 \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377
 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z
 I
```

```
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

兼容性

SQL 标准中没有COPY语句。

以下语法在 PostgreSQL 7.3 版之前使用，现在仍受支持：

```
COPY [ BINARY ] tablename [ WITH OIDS ]
    FROM { 'filename' | STDIN }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]

COPY [ BINARY ] tablename [ WITH OIDS ]
    TO { 'filename' | STDOUT }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
```

CREATE AGGREGATE

CREATE AGGREGATE — 定义一个新的聚合函数

大纲

```
CREATE AGGREGATE name (
    BASETYPE = input_data_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ]
)
```

描述

`CREATE AGGREGATE` 定义一个新的聚合函数。标准发行版中已经提供了一些针对基本类型的聚合函数，例如`min(integer)`和`avg(double precision)`。如果定义了新的类型，或者需要一个尚未提供的聚合函数，就可以使用`CREATE AGGREGATE`来提供所需的功能。

如果给出了一个模式名（例如`CREATE AGGREGATE myschema.myagg ...`），则该聚合函数会在指定模式中创建。否则会在当前模式中创建。

聚合函数由其名称和输入数据类型标识。同一模式中的两个聚合如果操作于不同的输入类型，则可以有相同的名称。
聚合的名称和输入数据类型还必须与同一模式中每个普通函数的名称和输入数据类型都不同。

一个聚合函数由一个或两个普通函数构成：一个状态转移函数*sfunc*，
以及一个可选的最终计算函数*ffunc*。它们的用法如下：

```
sfunc( internal-state, next-data-item ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
```

PostgreSQL 会创建一个数据类型为 *stype* 的临时变量，用来保存聚合的当前内部状态。对于每个输入数据项，都会调用状态转移函数来计算新的内部状态值。等到所有数据都处理完毕后，再调用一次最终函数来计算聚合的返回值。如果没有最终函数，则原样返回结束时的状态值。

聚合函数可以提供一个初始条件，即内部状态值的初始值。它在数据库中以 `text`类型的列的形式指定和存储，但它必须是状态值数据类型常量的合法外部表示。如果未提供，
则状态值初始为空值。

如果状态转移函数被声明为“strict”的，那么它不能以空值输入被调用。使用这种转移函数时，聚合的执行行为如下。空值输入会被忽略（不调用函数并保留先前的状态值）。
如果初始状态值为空，那么第一个非空输入值会取代状态值，
而转移函数从第二个非空输入值开始被调用。这对于实现`max`这类聚合很方便。注意，只有当*state_data_type*与*input_data_type*相同时，此行为才可用。当这两种类型不同时，必须提供非空的初始条件或者使用非严格的转移函数。

如果状态转移函数不是严格的，那么它会在每个输入值上被无条件调用，
并且必须自行处理空值输入和空值转移值。这使聚合作者能完全控制聚合对空值的处理。

如果最终函数被声明为“strict”，那么当结束状态值为空时不会调用它；而是自动返回空结果。（这当然只是 strict 函数的正常行为。）无论如何，最终函数都可以选择返回空值。例如，`avg`的最终函数在发现输入行数为零时会返回空值。

参数

name

要创建的聚合函数的名称（可带模式限定）。

input_data_type

此聚合函数所操作的输入数据类型。对于不检查其输入值的聚合，可以将其指定为"ANY"（这种聚合的一个例子是count(*)）。

sfunc

要为每个输入数据值调用的状态转移函数的名称。它通常是一个双参数函数，
第一个参数的类型为*state_data_type*，第二个参数的类型为*input_data_type*。
或者，对于不检查其输入值的聚合，该函数只接受一个类型为*state_data_type*的参数。
无论哪种情况，该函数都必须返回*state_data_type*类型的值。
此函数接受当前状态值和当前输入数据项，并返回下一个状态值。

state_data_type

聚合状态值的数据类型。

ffunc

在遍历完所有输入数据之后调用的、用于计算聚合结果的最终函数的名称。
该函数必须接受单个类型为*state_data_type*的参数。
聚合的返回数据类型被定义为该函数的返回类型。如果没有指定*ffunc*，
则结束状态值被用作聚合的结果，返回类型为*state_data_type*。

initial_condition

状态值的初始设置。这必须是*state_data_type*数据类型所接受形式的字符串常量。
如果未指定，状态值将从空值开始。

CREATE AGGREGATE的参数可以按任意顺序书写，而不只是按上面展示的顺序。

示例

见第 33.9 节。

兼容性

CREATE AGGREGATE是 PostgreSQL的语言扩展。SQL 标准不提供用户定义的聚合函数。

另见

ALTER AGGREGATE, DROP AGGREGATE

CREATE CAST

CREATE CAST — 定义一个新的转换

大纲

```
CREATE CAST (sourcetype AS targettype)
    WITH FUNCTION funcname (argtype)
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (sourcetype AS targettype)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

描述

`CREATE CAST` 定义一个新的转换。转换指定如何执行两种数据类型之间的转换。例如，

```
SELECT CAST(42 AS text);
```

通过调用一个先前指定的函数（这里是 `text(int4)`）把整数常量 42 转换为 `text` 类型。（如果没有定义合适的转换，该转换失败。）

两种类型可以是二进制兼容的，这意味着它们可以“免费”地互相转换而无需调用任何函数。这要求相应的值使用相同的内部表示。例如，`text` 和 `varchar` 类型是二进制兼容的。

默认情况下，一个转换只能在显式转换请求中被调用，也就是显式的 `CAST(x AS typename)`、`x::typename` 或 `typename(x)` 构造。

如果转换被标记为 `AS ASSIGNMENT`，那么在向目标数据类型的列赋值时它可以被隐式调用。例如，假设 `foo.f1` 是一个 `text` 类型的列，那么

```
INSERT INTO foo (f1) VALUES (42);
```

只有在从 `integer` 类型到 `text` 类型的转换被标记为 `AS ASSIGNMENT` 时才是允许的，否则不允许。（我们通常用赋值转换这个术语来描述这种转换。）

如果转换被标记为 `AS IMPLICIT`，那么它在任何上下文中都可以被隐式调用，无论是赋值还是表达式内部。例如，由于 `||` 接受 `text` 操作数，

```
SELECT 'The time is ' || now();
```

只有在从 `timestamp` 到 `text` 的转换被标记为 `AS IMPLICIT` 时才允许。否则就必须显式写出转换，例如

```
SELECT 'The time is ' || CAST(now() AS text);
```

（我们通常用隐式转换这个术语来描述这种转换。）

在把转换标记为隐式方面明智的做法是保守。过多的隐式转换路径可能导致 PostgreSQL 对命令做出令人惊讶的解释，或者因为存在多种可能的解释而完全无法解析命令。
一个好的经验法则是在同一通用类型类别中的类型之间，只让保持信息的转换可以隐式调用。例如，从 `int2` 到 `int4` 的转换可以合理地隐式，但从 `float8` 到 `int4` 的转换或许应该只用于赋值。跨类型类别的转换，例如从 `text` 到 `int4`，最好只用于显式转换。

要能创建一个转换，你必须拥有源数据类型或目标数据类型。要创建一个二进制兼容的转换，你必须是超级用户。（这个限制是因为一个错误的二进制兼容转换很容易使服务器崩溃。）

参数

sourcetype

转换的源数据类型的名称。

targettype

转换的目标数据类型的名称。

funcname(*argtype*)

用来执行该转换的函数。函数名可以带模式限定。如果不带，该函数将在路径中查找。参数类型必须与源类型相同，结果数据类型必须与该转换的目标类型匹配。

`WITHOUT FUNCTION`

表示源类型和目标类型是二进制兼容的，因此不需要函数来执行该转换。

`AS ASSIGNMENT`

表示该转换可以在赋值上下文中被隐式调用。

`AS IMPLICIT`

表示该转换可以在任何上下文中被隐式调用。

注意

使用 `DROP CAST` 删除用户定义的转换。

记住，如果你希望能够双向转换类型，需要显式地在两个方向上声明转换。

在 PostgreSQL 7.3 之前，每个与某个数据类型同名、返回该数据类型并且接受一个不同类型参数的函数都自动是一个转换函数。面对模式（schema）的引入以及为了能在系统目录中表示二进制兼容的转换，这一约定已被放弃。（内建的转换函数仍然遵循这一命名方案，但现在它们必须作为转换显示在系统目录 `pg_cast` 中。）

示例

要用函数 `int4(text)` 创建一个从 `text` 类型到 `int4` 类型的转换：

```
CREATE CAST (text AS int4) WITH FUNCTION int4(text);
```

（这个转换已经在系统中预定义。）

兼容性

`CREATE CAST` 命令符合 SQL99，只是 SQL99 没有为二进制兼容类型作规定。`AS IMPLICIT` 也是一种 PostgreSQL 扩展。

另见

CREATE FUNCTION, CREATE TYPE, DROP CAST

CREATE CONSTRAINT TRIGGER

CREATE CONSTRAINT TRIGGER — 定义一个新的约束触发器

大纲

```
CREATE CONSTRAINT TRIGGER name
    AFTER events ON
    tablename constraint attributes
    FOR EACH ROW EXECUTE PROCEDURE funcname ( args )
```

描述

`CREATE CONSTRAINT TRIGGER` 被 `CREATE TABLE/ALTER TABLE` 和 pg_dump 用来为参照完整性创建特殊的触发器。它不打算用于一般用途。

参数

name

约束触发器的名称。

events

该触发器应针对其触发的事件类别。

tablename

触发事件发生的表的名称（可带模式限定）。

constraint

实际的约束说明。

attributes

约束属性。

funcname(*args*)

作为触发器处理的一部分调用的函数。

CREATE CONVERSION

CREATE CONVERSION — 定义一个新的编码转换

大纲

```
CREATE [DEFAULT] CONVERSION name
    FOR source_encoding TO dest_encoding FROM funcname
```

描述

CREATE CONVERSION定义字符集编码之间的一个新转换。转换名可以用在convert函数中来指定一个特定的编码转换。此外，标记为 DEFAULT的转换可用于客户端和服务器之间的自动编码转换。为此目的，必须定义两个转换：从编码 A 到 B 以及从编码 B 到 A。

要能够创建转换，你必须对该函数拥有EXECUTE 权限，并对目标模式拥有CREATE权限。

参数

DEFAULT

DEFAULT子句表示这个转换是这一特定的源编码到目标编码的默认转换。在一个模式中，对于一对编码只应有一个默认转换。

name

转换的名称。转换名可以是模式限定的。如果不是，转换定义在当前模式中。转换名在一个模式内必须唯一。

source_encoding

源编码名称。

dest_encoding

目标编码名称。

funcname

执行转换所用的函数。函数名可以是模式限定的。如果不是，函数将在路径中查找。

该函数必须具有下面的签名：

```
conv_proc(
    integer,  -- source encoding ID
    integer,  -- destination encoding ID
    cstring,  -- source string (null terminated C string)
    internal, -- destination (fill with a null terminated C string)
    integer   -- source string length
) RETURNS void;
```

注解

使用DROP CONVERSION删除用户定义的转换。

创建转换所需的权限在将来的版本中可能会更改。

示例

要创建一个从编码`UNICODE`到 `LATIN1`的转换，使用`myfunc`：

```
CREATE CONVERSION myconv FOR 'UNICODE' TO 'LATIN1' FROM myfunc;
```

兼容性

`CREATE CONVERSION` 是PostgreSQL的一个扩展。SQL 标准中没有`CREATE CONVERSION`语句。

另见

ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION

CREATE DATABASE

CREATE DATABASE — 创建一个新数据库

大纲

```
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] dbowner ]
           [ LOCATION [=] 'dbpath' ]
           [ TEMPLATE [=] template ]
           [ ENCODING [=] encoding ] ]
```

描述

`CREATE DATABASE` 创建一个新的 PostgreSQL 数据库。

要创建数据库，你必须是超级用户，或者拥有特殊的 `CREATEDB` 权限。见CREATE USER。

通常，创建者成为新数据库的拥有者。超级用户可以使用 `OWNER` 子句创建由其他用户拥有的数据库。他们甚至可以创建由没有特殊权限的用户拥有的数据库。拥有 `CREATEDB` 权限的非超级用户只能创建由他们自己拥有的数据库。

可以指定一个备选位置，例如，把数据库存储在另一块磁盘上。该路径必须已经用initlocation命令准备好。

如果路径名不包含斜杠，它会被解释为一个环境变量名，服务器进程必须知道这个变量。这样数据库管理员就可以控制可以在哪些位置创建数据库。（惯例的选择例如 `PGDATA2`。）如果服务器编译时带 `ALLOW_ABSOLUTE_DBPATHS`（默认不带），那么以前导斜杠标识的绝对路径名（例如 `/usr/local/pgsql/data`）也是允许的。无论哪种情况，最终路径名必须是绝对路径，并且不能包含单引号。

默认情况下，新数据库将通过克隆标准系统数据库 `template1` 来创建。可以通过写成 `TEMPLATE name` 指定其他模板。特别是，写成 `TEMPLATE template0` 时，可以创建一个全新的数据库，它只包含你的 PostgreSQL 版本预定义的标准对象。如果你希望避免复制任何可能已添加到 `template1` 中的安装本地对象，这会很有用。

可选的编码参数允许选择数据库编码。未指定时，它默认为所选模板数据库使用的编码。

参数

name

要创建的数据库的名称。

dbowner

将拥有新数据库的数据库用户的名称，或者 `DEFAULT` 表示使用默认值（即执行该命令的用户）。

dbpath

存储新数据库的备选文件系统位置，以字符串字面量的形式指定；或者 `DEFAULT` 表示使用默认位置。

template

创建新数据库所用模板的名称，或者 `DEFAULT` 表示使用默认模板（`template1`）。

encoding

在新数据库中使用的字符集编码。指定一个字符串常量（例如 `'SQL_ASCII'`）、一个整数编码号，或者 `DEFAULT` 表示使用默认编码。

可选参数可以按任意顺序书写，而不只是按上面展示的顺序。

注意

`CREATE DATABASE` 不能在事务块内执行。

类似“could not initialize database directory”（无法初始化数据库目录）这样的错误最有可能与数据目录上的权限不足、磁盘已满或其他文件系统问题有关。使用备选位置时，运行数据库服务器的用户必须能访问该位置。

使用DROP DATABASE删除一个数据库。

程序createdb是围绕这个命令的包装程序，为方便起见提供。

使用以绝对路径名指定的备选数据库位置会涉及安全问题；这就是该特性默认未启用的原因。见第 18.5 节了解更多信息。

虽然可以通过把某个数据库的名字指定为模板来复制 `template1` 以外的数据库，但这（目前）并不是要作为一种通用的“`COPY DATABASE`”设施。我们建议把用作模板的数据库当作只读的。见第 18.3 节了解更多信息。

示例

创建一个新的数据库：

```
CREATE DATABASE lusiadas;
```

要在备选区域 `~/private_db` 中创建一个新数据库，从 shell 执行下列命令：

```
mkdir private_db
initlocation ~/private_db
```

然后在 psql 会话中执行下列命令：

```
CREATE DATABASE elsewhere WITH LOCATION '/home/olly/private_db';
```

兼容性

SQL 标准中没有 `CREATE DATABASE` 语句。数据库等价于目录，其创建是实现定义的。

CREATE DOMAIN

CREATE DOMAIN — 定义一个新的域

大纲

```
CREATE DOMAIN name [AS] data_type
    [ DEFAULT expression ]
    [ constraint [ ... ] ]
```

其中 *constraint* 是:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

描述

`CREATE DOMAIN`创建一个新的数据域。定义域的用户将成为它的拥有者。

如果给出了一个模式名（例如`CREATE DOMAIN myschema.mydomain ...`），那么该域会被创建在指定的模式中。否则它会被创建在当前模式中。
域名在其模式中已有的类型和域之间必须唯一。

域适合用于把多个表共用的公共字段抽象到一个单一位置进行维护。例如，
一个电子邮件地址列可能在多个表中使用，而它们具有相同的属性。定义一个域并使用它，
而不是为每个表单独设置约束。

参数

name

要创建的域的名字（可以用模式限定）。

data_type

域的底层数据类型。这可以包括数组说明符。

`DEFAULT` *expression*

`DEFAULT`子句为域数据类型的列指定一个默认值。
该值可以是任何不含变量的表达式（但不允许子查询）。
默认表达式的数据类型必须与域的数据类型匹配。如果没有指定默认值，
那么默认值为空值。

在任何未为列指定值的插入操作中都会使用默认表达式。
如果为某个特定的列定义了默认值，它会覆盖与域相关联的任何默认值。反过来，
域默认值又会覆盖与底层数据类型相关联的任何默认值。

`CONSTRAINT` *constraint_name*

一个约束的可选名称。如果没有指定，系统会生成一个名字。

`NOT NULL`

此域的值不允许为空。

`NULL`

此域的值允许为空。这是默认值。

此子句只是为了与非标准 SQL 数据库兼容而设。不鼓励在新的应用中使用它。

`CHECK (expression)`

`CHECK`子句指定该域的值必须满足的完整性约束或测试。
每个约束都必须是一个产生布尔结果的表达式。它应当使用名字`VALUE`来引用被测试的值。

当前，`CHECK`表达式不能包含子查询，也不能引用`VALUE`之外的变量。

示例

这个例子创建`country_code`数据类型，然后在表定义中使用该类型：

```
CREATE DOMAIN country_code char(2) NOT NULL;
CREATE TABLE countrylist (id integer, country country_code);
```

兼容性

`CREATE DOMAIN`命令符合 SQL 标准。

另见

DROP DOMAIN

CREATE FUNCTION

CREATE FUNCTION — 定义一个新函数

大纲

```
CREATE [ OR REPLACE ] FUNCTION name ( [ argtype [, ...] ] )
    RETURNS rettype
  { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

描述

`CREATE FUNCTION`定义一个新函数。`CREATE OR REPLACE FUNCTION`将创建一个新函数，或者替换现有定义。

如果包含模式名，则函数将在指定的模式中创建。否则将在当前模式中创建。
新函数的名称不能与同一模式中具有相同参数类型的任何现有函数匹配。然而，
不同参数类型的函数可以共享一个名称（这称为重载）。

要更新现有函数的定义，可以使用`CREATE OR REPLACE FUNCTION`。但不能用这种方式更改函数的名称或者参数类型（如果尝试这样做，实际上就会创建一个新的不同函数）。此外，`CREATE OR REPLACE FUNCTION`也不允许更改现有函数的返回类型。要做到这一点，必须删除该函数并重新创建。

如果删除函数后再重新创建，新函数就不再是旧函数的同一实体；
你将破坏引用旧函数的现有规则、视图、触发器等。使用`CREATE OR REPLACE FUNCTION`可以在不破坏引用该函数的对象的情况下更改函数定义。

创建该函数的用户将成为该函数的拥有者。

参数

name

要创建的函数的名称。

argtype

函数参数的数据类型（可以有模式限定），如果有的话。参数类型可以是基本类型、
复合类型或域，也可以复制一个现有列的类型。

引用一个列的类型时写作 *tablename.columnname*`%TYPE`；有时使用这种写法可以帮助让函数独立于表定义的变化。

取决于实现语言，也可能允许指定`cstring`之类的“伪类型”。
伪类型表示实际参数类型要么没有完整指定，要么在普通 SQL 数据类型的范围之外。

rettype

返回数据类型（可以有模式限定）。返回类型可以是基本类型、复合类型或域，也可以指定为复制一个现有列的类型。关于如何引用现有列的类型，见上面argtype的描述。

取决于实现语言，也可能允许指定cstring之类的“伪类型”。SETOF 修饰符表示该函数将返回一组结果项，而不是单个项。

langname

函数实现所用的语言名称。可以是SQL、C、internal，或者一种用户定义的过程语言的名称。（另见createlang。）为了向后兼容，该名称可以用单引号包围。

IMMUTABLE
STABLE
VOLATILE

这些属性告知系统，出于运行时优化的目的，用单次求值替换对函数的多次求值是否安全。最多只能指定其中之一。如果没有出现这些选项，默认假定是VOLATILE。

IMMUTABLE表示该函数在给定相同参数值时总是返回相同结果；也就是说，它不做数据库查找，也不使用不直接出现在其参数列表中的信息。如果给出了这个选项，任何用全常量参数对该函数的调用都可以立即替换为函数值。

STABLE表示在一次表扫描中，该函数对相同参数值会一致地返回相同结果，但其结果可能跨 SQL 语句变化。对于结果依赖数据库查找、参数变量（例如当前时区）等的函数，这是合适的选择。还要注意，current_timestamp函数族属于稳定（stable），因为它们的值在一个事务内不会改变。

VOLATILE表示即使在一次表扫描中函数值也可能改变，因此无法做任何优化。相对较少的数据库函数在这个意义上是易变（volatile）的；一些例子是random()、currval()、timeofday()。注意，任何有副作用的函数都必须归类为 volatile，即使其结果相当可预测，以防止调用被优化掉；一个例子是 setval()。

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT（默认值）表示当某些参数为空时，函数仍会被正常调用。此时如有必要，由函数作者负责检查空值并做出适当响应。

RETURNS NULL ON NULL INPUT或 STRICT表示只要任一参数为空，函数就总是返回空。如果指定了这个选项，当存在空参数时函数不会被执行；而是自动假定结果为空。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER表示函数将以调用它的用户的权限执行。这是默认值。SECURITY DEFINER 指定函数将以创建它的用户的权限执行。

关键字EXTERNAL是为了 SQL 一致性而存在的，但它是可选的，因为与 SQL 中不同，这个特性并不只适用于外部函数。

definition

定义函数的字符串；其含义取决于语言。它可以是内部函数名、对象文件的路径、SQL 命令，或者过程语言中的文本。

obj_file, *link_symbol*

当 C 语言源代码中的函数名与 SQL 函数名不同时，动态载入的 C 语言函数使用这种形式的AS子句。字符串*obj_file*是包含可动态载入对象的文件名，而 *link_symbol*是函数的链接符号，也就是 C 语言源代码中函数的名称。如果省略链接符号，则假定它与被定义的 SQL 函数同名。

attribute

指定函数可选信息的传统方式。这里可以出现下列属性：

isStrict

等价于STRICT或RETURNS NULL ON NULL INPUT

isCachable

isCachable是IMMUTABLE的已废弃等价物；出于向后兼容的原因它仍被接受。

属性名不区分大小写。

注意

关于编写函数的更多信息，请参考第 33.3 节。

输入参数和返回值允许使用完整的SQL类型语法。但是，类型规范的某些细节（例如numeric类型的精度字段）由底层函数实现负责，CREATE FUNCTION命令会静默地忽略它们（即不识别也不强制执行）。

PostgreSQL允许函数重载；也就是说，只要几个不同函数的参数类型不同，它们就可以使用同一个名称。但是，所有函数的 C 名称必须不同，因此你必须给重载的 C 函数起不同的 C 名（例如，把参数类型用作 C 名的一部分）。

当多次CREATE FUNCTION调用引用同一个对象文件时，该文件只会被装载一次。要卸载并重新装载该文件（也许是在开发期间），可以使用LOAD命令。

使用DROP FUNCTION 删除用户定义的函数。

函数定义中的任何单引号或反斜杠都必须通过双写来转义。

要能够定义函数，用户必须具有该语言上的USAGE权限。

示例

这里有一个帮助你入门的简单示例。更多信息和示例见第 33.3 节。

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

安全地编写SECURITY DEFINER函数

由于SECURITY DEFINER函数以创建它的用户的权限执行，需要小心确保函数不被滥用。出于安全考虑，应把search_path设置为排除任何可被不受信任用户写入的模式。
这可以防止恶意用户创建遮蔽函数所用对象的对象。在这方面特别重要的是临时表模式，它默认最先被搜索，而且通常任何人都可以写。
可以通过强制临时模式最后被搜索来获得安全的安排。为此，把pg_temp写成search_path中的最后一项。下面的函数展示了安全用法：

```
CREATE FUNCTION check_password(TEXT, TEXT)
RETURNS BOOLEAN AS '
DECLARE passed BOOLEAN;
        old_path TEXT;
BEGIN
        -- Save old search_path; notice we must qualify current_
setting
        -- to ensure we invoke the right function
        old_path := pg_catalog.current_setting(''search_path'');

        -- Set a secure search_path: trusted schemas, then ''pg_
temp''.
        -- We set is_local = true so that the old value will be
 restored
        -- in event of an error before we reach the function end.
        PERFORM pg_catalog.set_config(''search_path'', ''admin, pg_
temp'', true);

        -- Do whatever secure work we came for.
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        -- Restore caller''s search_path
        PERFORM pg_catalog.set_config(''search_path'', old_path, true)
;

        RETURN passed;
END;
' LANGUAGE plpgsql SECURITY DEFINER;
```

兼容性

CREATE FUNCTION命令在 SQL99 中定义。PostgreSQL的版本与之类似但不完全兼容。属性不可移植，各种可用语言也一样。

另见

ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE, createlang

CREATE GROUP

CREATE GROUP — 定义一个新的用户组

大纲

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]

where option can be:

     SYSID gid
   | USER  username [, ...]
```

描述

`CREATE GROUP` 将在数据库集群中创建一个新组。你必须是数据库超级用户才能使用这条命令。

使用ALTER GROUP更改组的成员，使用DROP GROUP删除组。

参数

name

组的名称。

gid

`SYSID` 子句可以用来选择新组的 PostgreSQL 组 ID。不过并不一定要这样做。

如果没有指定，默认将使用已分配的最大组 ID 加一（从 1 开始）。

username

要包含在该组中的用户列表。这些用户必须已经存在。

示例

创建一个空组：

```
CREATE GROUP staff;
```

创建一个带成员的组：

```
CREATE GROUP marketing WITH USER jonathan, david;
```

兼容性

SQL 标准中没有`CREATE GROUP`语句。角色在概念上与组类似。

CREATE INDEX

CREATE INDEX — define a new index

大纲

```
CREATE [ UNIQUE ] INDEX name ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [, ...] )
    [ WHERE predicate ]
```

描述

`CREATE INDEX` 在指定的表上构建一个索引 *index_name*。索引主要用于提升数据库性能（但使用不当也可能导致性能下降）。

索引的键字段指定为列名，或者指定为写在圆括号中的表达式。如果索引方法支持多列索引，则可以指定多个字段。

索引字段可以是根据表行中一个或多个列值计算得到的表达式。该特性可用于根据基础数据的某种变换来快速访问数据。例如，在`upper(col)`上计算的索引可让子句 `WHERE upper(col) = 'JIM'`使用索引。

PostgreSQL提供的索引方法包括 B-树、R-tree、hash 和 GiST。B-树索引方法是 Lehman-Yao 高并发 B-树的一个实现。R-tree 索引方法使用 Guttman 的二次分裂算法实现标准 R-tree。hash 索引方法是 Litwin 线性散列的一个实现。用户也可以定义自己的索引方法，但这相当复杂。

当`WHERE`子句存在时，会创建一个部分索引。部分索引只包含表中一部分行的索引项，通常这一部分比表的其余部分更适合建立索引。例如，如果一个表同时包含已开票和未开票订单，而未开票订单只占整个表的一小部分，但这一部分又经常被访问，就可以只对这部分创建索引来提升性能。另一种可能的应用是将`WHERE`与`UNIQUE`结合使用，以便在表的一个子集上强制唯一性。

`WHERE`子句中使用的表达式只能引用底层表的列，但它可以使用所有列，而不仅仅是被索引的列。当前，`WHERE`中也禁止使用子查询和聚合表达式。同样的限制也适用于作为表达式的索引字段。

所有在索引定义中使用的函数和操作符都必须是“不可变的”，也就是说，它们的结果只能依赖其参数，而不能受任何外部因素影响（例如另一个表的内容或当前时间）。这种限制确保索引的行为定义明确。要在索引表达式或`WHERE`子句中使用用户定义的函数，记得在创建该函数时将其标记为不可变。

参数

`UNIQUE`

使系统在创建索引时（如果数据已经存在）以及每次添加数据时，检查表中的重复值。任何会导致重复项的插入或更新操作都会报错。

name

要创建的索引名称。此处不能包含模式名称；索引始终创建在其父表所在的同一模式中。

table

要建立索引的表名（可以是模式限定名）。

method

要使用的索引方法的名称。选择包括 `btree`、`hash`、`rtree` 和 `gist`。默认方法是 `btree`。

column

一个表列的名称。

expression

一个基于表中一个或多个列的表达式。通常必须像语法中所示那样写在外围圆括号中。不过，如果该表达式是函数调用形式，则可以省略圆括号。

opclass

一个操作符类的名称。详见下文。

predicate

部分索引的约束表达式。

注解

关于索引何时能被使用、何时不被使用以及什么情况下它们有用的信息请见第 11 章。

目前，只有 B-树和 GiST 索引方法支持多列索引。默认最多可指定 32 个字段。（构建PostgreSQL时可以更改此限制。）目前只有 B-树支持唯一索引。

索引的每一列都可以指定一个操作符类。操作符类标识索引用于该列的操作符。例如，四字节整数上的 B-树索引会使用`int4_ops`类；此操作符类包含四字节整数的比较函数。实际上，列数据类型的默认操作符类通常已足够。操作符类存在的主要原因是，某些数据类型可能有多个有意义的排序方式。例如，我们可能希望按绝对值或实部对复数数据类型排序。我们可以为该数据类型定义两个操作符类，并在创建索引时选择合适的类。有关操作符类的更多信息，请参见第 11.6 节和第 33.13 节。

使用DROP INDEX删除索引。

示例

创建 B-树索引，列为`title`，所在表为`films`：

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

兼容性

`CREATE INDEX`是 PostgreSQL的语言扩展。SQL 标准中没有关于索引的规定。

CREATE LANGUAGE

CREATE LANGUAGE — 定义一种新的过程语言

大纲

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ VALIDATOR valfunction ]
```

描述

使用CREATE LANGUAGE，PostgreSQL用户可以在 PostgreSQL数据库中注册一种新的过程语言。随后，就可以用这种新语言定义函数和触发器过程。注册新语言的用户必须拥有PostgreSQL超级用户权限。

CREATE LANGUAGE实际上把语言名称与一个负责执行以该语言编写的函数的调用处理器关联起来。关于语言调用处理器的更多信息，请参考第 33.3 节。

注意，过程语言是每个数据库局部的。要让一种语言默认在所有数据库中可用，应该把它安装进template1数据库。

参数

TRUSTED

TRUSTED指定该语言的调用处理器是安全的，也就是说，它不会为无特权用户提供任何绕过访问限制的功能。如果注册语言时省略这个关键字，则只有拥有 PostgreSQL超级用户权限的用户能用这种语言创建新函数。

PROCEDURAL

这是一个噪声词。

name

新过程语言的名称。语言名不区分大小写。该名称在数据库中的语言之间必须唯一。

为了向后兼容，该名称可以用单引号包围。

HANDLER *call_handler*

*call_handler*是一个先前已注册函数的名称，该函数将被调用来执行过程语言函数。过程语言的调用处理器必须用 C 之类的编译语言以版本 1 调用约定编写，并作为不带参数、返回 language_handler类型的函数注册到 PostgreSQL中，这个占位类型只是用来把该函数标识为调用处理器。

VALIDATOR *valfunction*

*valfunction*是一个先前已注册函数的名称，当用该语言创建新函数时会调用它来验证新函数。如果没有指定验证器函数，那么新函数在创建时将不被检查。验证器函数必须接受一个 oid类型的参数，它是要创建的函数的 OID，并且通常返回void。

验证器函数通常会检查函数体的语法正确性，但它也可以检查函数的其他属性，例如该语言是否不能处理某些参数类型。要发出错误信号，验证器函数应该使用`ereport()`函数。函数的返回值会被忽略。

注意

通常用户不应直接执行这条命令。对于 PostgreSQL发行版本中提供的过程语言，应该使用createlang程序，它还会安装正确的调用处理器。（`createlang`会在内部调用`CREATE LANGUAGE`。）

在PostgreSQL 7.3 之前的版本中，必须把处理器函数声明为返回占位类型 `opaque`而不是`language_handler`。为了支持装载旧的转储文件，`CREATE LANGUAGE`会接受声明为返回`opaque`的函数，但会发出一条提示，并把该函数声明的返回类型改为`language_handler`。

使用CREATE FUNCTION命令创建新函数。

使用DROP LANGUAGE，或者更好的是droplang程序，来删除过程语言。

系统目录`pg_language`（见第 43.18 节）记录当前已安装语言的信息。此外`createlang`也有一个列出已安装语言的选项。

过程语言的定义一旦创建就不能更改，权限除外。

要使用一种过程语言，用户必须被授予`USAGE`权限。如果已知语言是可信的，`createlang`程序会自动向所有人授予权限。

示例

依次执行下面两条命令将注册一种新的过程语言及相关的调用处理器。

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

兼容性

`CREATE LANGUAGE`是 PostgreSQL扩展。

另见

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE, createlang, droplang

CREATE OPERATOR

CREATE OPERATOR — 定义一个新的操作符

大纲

```
CREATE OPERATOR name (
    PROCEDURE = funcname
    [, LEFTARG = lefttype ] [, RIGHTARG = righttype ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
    [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op ]
    [, LTCMP = less_than_op ] [, GTCMP = greater_than_op ]
)
```

描述

`CREATE OPERATOR`定义一个新的操作符 *name*。定义该操作符的用户将成为其拥有者。如果给出了模式名，该操作符将在指定的模式中创建；否则将在当前模式中创建。

操作符名称是一个字符序列，最多包含 `NAMEDATALEN`-1 个字符（默认为 63 个），字符取自以下列表：

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

名称的选择有以下几项限制：

- `--`和`/*`不能出现在操作符名称的任何位置，因为它们会被视为注释的开始。
- 多字符操作符名称不能以`+`或`-` 结尾，除非该名称中还至少包含下列字符之一：

  ~ ! @ # % ^ & | ` ?

  例如，`@-`是允许的操作符名称，而 `*-`则不允许。这一限制使 PostgreSQL能够在不要求记号之间必须有空格的情况下解析符合 SQL 规范的命令。

输入`!=`时会被映射为`<>`，因此这两个名称始终等效。

必须至少定义`LEFTARG`和`RIGHTARG`中的一个。对于二元操作符，必须同时定义两者。
对于右一元操作符，只应定义`LEFTARG`；对于左一元操作符，只应定义`RIGHTARG`。

*funcname*过程必须事先已用`CREATE FUNCTION`定义，并且其定义必须接受正确数量（一个或两个）的指定类型参数。

其余子句指定可选的操作符优化子句。其含义详见第 33.11 节。

参数

name

要定义的操作符名称。允许使用的字符请见上文。该名称可以带模式限定，例如`CREATE OPERATOR myschema.+ (...)`。如果不带模式限定，该操作符将在当前模式中创建。同一

模式中的两个操作符如果针对不同的数据类型进行操作，可以具有相同的名称。这被称为重载。

funcname

用于实现该操作符的函数。

lefttype

操作符左参数的类型（如果有的话）。对于左一元操作符，这个选项要省略。

righttype

操作符右参数的类型（如果有的话）。对于右一元操作符，这个选项要省略。

com_op

该操作符的交换子。

neg_op

该操作符的求反器。

res_proc

该操作符的限制选择度估计函数。

join_proc

该操作符的连接选择度估算函数。

HASHES

表示该操作符可以支持哈希连接。

MERGES

表示该操作符可以支持归并连接。

left_sort_op

如果该操作符能支持归并连接，则为对该操作符左手数据类型排序的小于操作符。

right_sort_op

如果该操作符能支持归并连接，则为对该操作符右手数据类型排序的小于操作符。

less_than_op

如果该操作符能支持归并连接，则为比较该操作符输入数据类型的小于操作符。

greater_than_op

如果该操作符能支持归并连接，则为比较该操作符输入数据类型的大于操作符。

要在*com_op*或其他可选参数中给出带模式限定的操作符名称，请使用OPERATOR()语法，例如：

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

注意

更多信息请参见第 33.11 节。

使用 `DROP OPERATOR` 从数据库中删除用户定义的操作符。

示例

以下命令定义一个新的面积相等操作符，适用的数据类型为 `box`：

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES,
    SORT1 = <<<,
    SORT2 = <<<
    -- Since sort operators were given, MERGES is implied.
    -- LTCMP and GTCMP are assumed to be < and > respectively
);
```

兼容性

`CREATE OPERATOR` 是 PostgreSQL 的扩展。SQL 标准中没有用户自定义操作符的规定。

CREATE OPERATOR CLASS

CREATE OPERATOR CLASS — 定义一个新的操作符类

大纲

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type USING index_
method AS
  {  OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [
 RECHECK ]
   | FUNCTION support_number funcname ( argument_type [, ...] )
   | STORAGE storage_type
  } [, ... ]
```

描述

CREATE OPERATOR CLASS 创建一个新的操作符类。操作符类定义一种特定的数据类型如何与索引一起使用。
操作符类指定某些操作符将为这种数据类型和这种索引方法填充特定的角色或“策略”。
操作符类还指定当为一个索引列选择该操作符类时索引方法要使用的支持过程。
操作符类使用的所有操作符和函数都必须在创建操作符类之前定义。

如果给出了模式名，则操作符类在指定的模式中创建。否则它在当前模式中创建。
同一模式中的两个操作符类只有在用于不同索引方法时才能同名。

定义操作符类的用户成为它的拥有者。目前，创建者必须是超级用户。
（这个限制是因为一个错误的操作符类定义可能混淆甚至使服务器崩溃。）

CREATE OPERATOR CLASS 目前不检查操作符类定义是否包含该索引方法所需的全部操作符和函数。定义一个有效的操作符类是用户的责任。

进一步的信息请参阅第 33.13 节。

参数

name

要创建的操作符类的名称。该名称可以带模式限定。

DEFAULT

如果出现，该操作符类将成为其数据类型的默认操作符类。
对于一种特定的数据类型和索引方法，最多只有一个操作符类可以是默认的。

data_type

该操作符类所针对的列数据类型。

index_method

该操作符类所针对的索引方法的名称。

strategy_number

与操作符类关联的操作符的索引方法策略号。

operator_name

与操作符类关联的操作符的名称（可带模式限定）。

op_type

操作符的操作数数据类型，或 NONE 表示左一元或右一元操作符。在操作数数据类型与操作符类的数据类型相同的正常情况下，可以省略操作数数据类型。

RECHECK

如果出现，索引对该操作符是“有损的”（lossy），
因此使用该索引检索出的行必须重新检查，以验证它们确实满足涉及该操作符的限定子句。

support_number

与操作符类关联的函数的索引方法支持过程号。

funcname

作为该操作符类的索引方法支持过程的函数的名称（可带模式限定）。

argument_types

函数的参数数据类型。

storage_type

实际存储在索引中的数据类型。通常它与列数据类型相同，
但某些索引方法（本文撰写时只有 GiST）允许它不同。除非索引方法允许使用不同的类型，
否则必须省略 STORAGE 子句。

OPERATOR、FUNCTION 和 STORAGE 子句可以按任意顺序出现。

示例

下面的示例命令为数据类型 _int4（int4 的数组）定义一个 GiST 索引操作符类。完整的示例见 contrib/intarray/。

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       =       RECHECK,
        OPERATOR        7       @,
        OPERATOR        8       ~,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4,
 int4),
        FUNCTION        2       g_int_union (bytea, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal,
 internal),
```

```
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

兼容性

`CREATE OPERATOR CLASS` 是一种 PostgreSQL 扩展。SQL 标准中没有 `CREATE OPERATOR CLASS` 语句。

另见

ALTER OPERATOR CLASS, DROP OPERATOR CLASS

CREATE RULE

CREATE RULE — 定义一条新的重写规则

大纲

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

描述

CREATE RULE定义一条应用于指定表或视图的新规则。CREATE OR REPLACE RULE将创建一条新规则，或者替换同一个表上同名的现有规则。

PostgreSQL规则系统允许定义在数据库表上执行插入、更新或删除时的替代动作。粗略地说，当在某个给定表上执行某个命令时，一条规则会导致额外的命令被执行。另一种情况是，一条规则可以用另一个命令替换给定命令，或者让某个命令完全不执行。
规则也用于实现表视图。必须认识到，规则实际上是一种命令转换机制，或者说是命令宏。这种转换发生在命令开始执行之前。如果你真正需要的是对每个物理行独立触发的操作，那么你很可能应该使用触发器，而不是规则。有关规则系统的更多信息见第 34 章。

目前，ON SELECT规则必须是无条件的INSTEAD规则，其动作必须由单个SELECT命令组成。因此，ON SELECT规则实际上会把表变为视图，其可见内容是该规则的SELECT命令返回的行，而不是表中原来存储的内容（如果有）。直接编写CREATE VIEW命令，被认为比创建真实表并为其定义ON SELECT规则的风格更好。

可以通过定义ON INSERT、ON UPDATE 和ON DELETE规则（或其中足以满足你需求的任意子集），来营造出可更新视图的效果，从而将视图上的更新动作替换为对其他表的适当更新。

如果你尝试对视图更新使用有条件规则，
这里有一个陷阱：对于你希望允许在该视图上执行的每一种动作，必须都有一条无条件的INSTEAD规则。如果该规则是有条件的，或者不是 INSTEAD规则，那么系统仍会拒绝执行该更新动作，因为它认为在某些情况下最终可能还是会尝试在该视图的虚拟表上执行该动作。
如果你想用有条件规则处理所有有用的情况，可以加上一条无条件的 DO INSTEAD NOTHING规则，以确保系统明白它永远不会被要求去更新这个虚拟表。然后，把这些有条件规则改成非INSTEAD规则；在它们适用的情况下，它们会附加在默认的 INSTEAD NOTHING动作之上。

参数

name

要创建的规则的名称。它必须与同一个表上任何其他规则的名称相区分。
同一个表上同一种事件类型的多条规则会按名称的字母顺序应用。

event

事件是SELECT、INSERT、UPDATE或者 DELETE之一。

table

规则适用的表或者视图的名称（可以是模式限定的）。

condition

任意的SQL条件表达式（返回 boolean）。条件表达式不能引用NEW和 OLD之外的任何表，并且不能包含聚合函数。

command

组成规则动作的命令。可用的命令有SELECT、INSERT、UPDATE、DELETE或者NOTIFY。

在*condition*和 *command*中，可以使用特殊表名 NEW和OLD来引用目标表中的值。在ON INSERT和ON UPDATE规则中，NEW可用于引用待插入或待更新的新行。在 ON UPDATE和ON DELETE规则中，OLD可用于引用将被更新或删除的现有行。

注解

要在一个表上定义规则，你必须拥有该表上的RULE权限。

避免循环规则非常重要。例如，尽管下面的两条规则定义都被 PostgreSQL所接受，SELECT命令将导致 PostgreSQL报告错误，因为规则会发生递归展开：

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
 SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
 SELECT * FROM t1;

SELECT * FROM t1;
```

当前，如果一个规则动作包含一个NOTIFY命令，该NOTIFY命令将被无条件执行，也就是说，即使没有任何应当应用该规则的行，也会发出NOTIFY。例如：

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

在UPDATE期间会发送一个NOTIFY事件，无论是否存在匹配条件id = 42的行。这是一个实现限制，未来版本中可能会修复。

兼容性

CREATE RULE是一种 PostgreSQL语言扩展，整个查询重写系统也是如此。

CREATE SCHEMA

CREATE SCHEMA — 定义一个新模式

大纲

```
CREATE SCHEMA schemaname [ AUTHORIZATION username ] [ schema_element [
 ... ] ]
CREATE SCHEMA AUTHORIZATION username [ schema_element [ ... ] ]
```

描述

`CREATE SCHEMA` 将把一个新模式登记到当前数据库中。模式名必须与当前数据库中任何现有模式的名称不同。

模式本质上是一个名字空间：它包含一些有名对象（表、数据类型、函数和操作符），
它们的名字可以与其他模式中现有对象的名字重复。
访问有名对象既可以通过用模式名作前缀来“限定”其名字，
也可以通过设置一个包含所需模式的搜索路径。
非限定对象创建在当前模式中（即搜索路径最前端的那个模式，可以用函数 `current_schema` 确定）。

`CREATE SCHEMA`还可以选择包含子命令，以便在新模式中创建对象。这些子命令基本上会被当作在创建模式之后单独发出的命令来处理。不过，如果使用了`AUTHORIZATION`子句，
则所有创建的对象都将归该用户所有。

参数

schemaname

要创建的模式的名称。如果省略，则用用户名作为模式名。

username

将拥有新模式的用户的角色名。如果省略，则默认为执行该命令的用户。
只有超级用户才能创建由其他用户拥有的模式。

schema_element

一条定义要在模式内创建的对象的 SQL 语句。目前，只有 `CREATE TABLE`、`CREATE VIEW` 和 `GRANT` 被接受为 `CREATE SCHEMA` 中的子句。其他类型的对象可以在模式创建之后用单独的命令创建。

注解

要创建模式，调用用户必须拥有当前数据库上的 `CREATE` 权限。（当然，超级用户会绕过这个检查。）

示例

创建一个模式：

```
CREATE SCHEMA myschema;
```

为用户joe创建一个模式，该模式也将被命名为 joe：

```
CREATE SCHEMA AUTHORIZATION joe;
```

创建一个模式并且在其中创建一个表和视图：

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
```

请注意，各个子命令末尾都不带分号。

下面是实现相同结果的一种等效写法：

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]
);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT
 NULL;
```

兼容性

SQL 标准允许在 CREATE SCHEMA 中使用 DEFAULT CHARACTER SET 子句，也允许比 PostgreSQL 目前所接受的更多的子命令类型。

SQL 标准规定 CREATE SCHEMA 中的子命令可以以任意顺序出现。目前的 PostgreSQL 实现并不处理子命令中前向引用的所有情形；有时可能需要重新排列子命令的顺序以避免前向引用。

根据 SQL 标准，模式的拥有者总是拥有其中的所有对象。PostgreSQL允许模式包含由模式拥有者之外的用户所拥有的对象。只有当模式拥有者把其模式上的CREATE
权限授予其他人时才会发生。

另见

ALTER SCHEMA, DROP SCHEMA

CREATE SEQUENCE

CREATE SEQUENCE — 定义一个新的序列发生器

大纲

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name [ INCREMENT [ BY ] increment
 ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO
 MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

描述

`CREATE SEQUENCE`创建一个新的序列号发生器。这会创建并初始化一个名为*name*的特殊单行表。该发生器归发出该命令的用户所有。

如果给定了模式名，则序列会在指定模式中创建。否则，它会在当前模式中创建。
临时序列存在于一个特殊模式中，因此创建临时序列时不能指定模式名。
序列名必须不同于同一模式中任何其他关系（表、序列、索引、视图、物化视图或外部表）的名称。

创建序列后，可以使用函数 `nextval`、`currval`以及 `setval`来操作该序列。这些函数的文档见第 9.11 节。

虽然不能直接更新序列，但可以使用如下查询：

```
SELECT * FROM name;
```

来查看序列的参数和当前状态。特别是，序列的 `last_value`字段会显示任一会话最近分配的值。（当然，如果其他会话正在调用`nextval`，那么到该值被打印出来时，它可能已经过时。）

参数

`TEMPORARY` or `TEMP`

如果指定该项，则该序列对象只为当前会话创建，并会在会话退出时自动删除。
临时序列存在期间，已有的同名永久序列在本会话中不可见，
除非使用模式限定名来引用它们。

name

要创建的序列的名称（可以是模式限定的）。

increment

可选子句`INCREMENT BY` *increment*指定加到当前序列值上以生成新值的增量。正值生成升序序列，负值生成降序序列。默认值为 1。

minvalue
`NO MINVALUE`

可选子句`MINVALUE` *minvalue*确定序列可生成的最小值。如果未提供此子句，或者指定了`NO MINVALUE`，则使用默认值。升序序列和降序序列的默认值分别为 1 和 $-2^{63}-1$。

maxvalue
NO MAXVALUE

可选子句MAXVALUE *maxvalue*确定序列的最大值。如果未提供此子句，或者指定了 NO MAXVALUE，则使用默认值。升序序列和降序序列的默认值分别为 2^{63}-1 和 -1。

start

可选子句START WITH *start* 允许序列从任意值开始。升序序列的默认起始值为*minvalue*，降序序列则为*maxvalue*。

cache

可选子句CACHE *cache*指定要预分配并存储在内存中以加快访问的序列号数量。最小值为 1（一次只能生成一个值，即不缓存），这也是默认值。

CYCLE
NO CYCLE

CYCLE选项允许序列在升序序列达到*maxvalue*、或降序序列达到*minvalue*时分别回绕。如果达到该限制，下一个生成的数字将分别是*minvalue*或*maxvalue*。

如果指定NO CYCLE，则当序列达到其限值后，任何对 nextval的调用都会返回错误。如果既未指定 CYCLE也未指定NO CYCLE，则默认使用NO CYCLE。

注解

使用DROP SEQUENCE删除序列。

序列基于bigint算术，因此其范围不能超过八字节整数的范围（-9223372036854775808 到 9223372036854775807）。在一些较老的平台上，编译器可能不支持八字节整数，此时序列将使用常规的integer算术（范围为 -2147483648 到 +2147483647）。

如果对一个将由多个会话并发使用的序列对象设置了大于 1 的*cache*，则可能得到意想不到的结果。每个会话会在一次访问该序列对象时分配并缓存连续的序列值，并相应增加该序列对象的 last_value。随后，该会话中接下来的 *cache*-1 次nextval 调用会直接返回预分配的值，而不接触序列对象。因此，任何在会话中已分配但未使用的数字都会在会话结束时丢失，从而在序列中造成“空洞”。

此外，虽然多个会话保证会分配到互不相同的序列值，但从所有会话整体来看，这些值的生成顺序可能会乱序。例如，当*cache*设置为 10 时，会话 A 可能保留值 1..10 并返回nextval=1；然后会话 B 可能保留值 11..20 并返回nextval=11，而此时会话 A 尚未生成 nextval=2。因此，当*cache*设置为 1 时，可以安全地假定 nextval值按顺序生成；当*cache*设置大于 1 时，只应假定 nextval值彼此不同，而不应假定它们严格按顺序生成。此外，last_value会反映任一会话最近保留的值，不管该值是否已经由nextval返回。

另一个需要注意的问题是，在这种序列上执行的setval 不会立刻被其他会话察觉，直到它们用完自己缓存的所有预分配值为止。

示例

创建一个名为serial的升序序列，从 101 开始：

```
CREATE SEQUENCE serial START 101;
```

从该序列中取出下一个数字：

```
SELECT nextval('serial');

 nextval
---------
     114
```

在INSERT命令中使用该序列：

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

在一次COPY FROM之后更新序列值：

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

兼容性

CREATE SEQUENCE 是一种 PostgreSQL 语言扩展。SQL 标准中没有 CREATE SEQUENCE 语句。

CREATE TABLE

CREATE TABLE — 定义一个新表

大纲

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name (
  { column_name data_type [ DEFAULT default_expr ] [ column_constraint
 [ ... ] ]
    | table_constraint
    | LIKE parent_table [ { INCLUDING | EXCLUDING } DEFAULTS ] }  [, .
.. ]
)
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY |
  CHECK (expression) |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL |
 MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
 IMMEDIATE ]

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) |
  PRIMARY KEY ( column_name [, ... ] ) |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ (
 refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ]
 [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
 IMMEDIATE ]
```

描述

`CREATE TABLE` 将在当前数据库中创建一个新的、初始为空的表。该表归发出该命令的用户所有。

如果给出了模式名（例如 `CREATE TABLE myschema.mytable ...`），则表将在指定模式中创建。否则，它将在当前模式中创建。临时表存在于一个特殊模式中，
因此创建临时表时不能给出模式名。表名必须与同一模式中任何其他表、序列、索引、
视图的名称不同。

`CREATE TABLE` 还会自动创建一种数据类型，用以表示与该表一行对应的复合类型。因此，表名不能与同一模式中任何已有数据类型同名。

一个表不能有超过 1600 列。（实际上，由于元组长度限制，有效的限制通常更低。）

可选的约束子句指定插入或更新要成功时，新行或更新后的行必须满足的约束（测试）。
约束是一种 SQL 对象，可用多种方式帮助定义表中允许的值集合。

定义约束有两种方式：表约束和列约束。列约束作为列定义的一部分定义。
表约束则不绑定到特定列，并且可以涵盖多个列。每个列约束也都可以写成表约束；
当约束只影响一列时，列约束只是一种书写上的方便。

参数

`TEMPORARY` or `TEMP`

如果指定该选项，表将创建为临时表。临时表会在会话结束时自动删除，
或者也可在当前事务结束时删除（见下文 ON COMMIT）。在临时表存在期间，同名的现有永久表对当前会话不可见，除非使用带模式限定的名称引用它们。
在临时表上创建的任何索引也都会自动成为临时索引。

可以选择在 `TEMPORARY` 或 `TEMP` 前写上 `GLOBAL` 或 `LOCAL`。这在 PostgreSQL 中没有任何区别，但参见兼容性。

`table_name`

要创建的表名（可选地带模式限定）。

`column_name`

要在新表中创建的列名。

`data_type`

列的数据类型。这可以包括数组说明符。

`DEFAULT` *`default_expr`*

`DEFAULT`子句为其所在列定义的列指定默认数据值。
该值可以是任何不含变量的表达式（不允许子查询，
也不允许交叉引用当前表中的其他列）。
默认表达式的数据类型必须与该列的数据类型匹配。

默认值表达式会用于任何未为该列指定值的插入操作。如果一列没有默认值，则默认值为 null。

`LIKE` *`parent_table`* `[ { INCLUDING | EXCLUDING } DEFAULTS ]`

`LIKE` 子句指定一个表，新表会自动从中继承所有列名、它们的数据类型以及非空约束。

与 `INHERITS` 不同，新表与被继承表在创建完成之后完全脱钩。向新表中插入的数据不会反映到父表中。

只有指定 `INCLUDING DEFAULTS` 时，才会包含被复制列定义的默认表达式。默认行为是不包含默认表达式。

`INHERITS ( parent_table [, ... ] )`

可选的 `INHERITS` 子句指定一组表，新表将自动从中继承所有列。如果同一列名出现在多个父表中，除非这些父表中该列的数据类型全部匹配，否则会报错。如果没有冲突，这些重复列会合并为新表中的单个列。
如果新表的列名列表中包含一个同时也是继承来的列，其数据类型也必须与继承列匹配，并且列定义会合并为一个。不过，
同名的继承列声明和新列声明不必指定相同的约束：来自任何声明的所有约束都会合并在一起，并全部应用于新表。如果新表为该列显式指定了默认值，
该默认值将覆盖列的继承声明中的任何默认值。否则，
为该列指定默认值的所有父表必须指定相同的默认值，否则将报错。

`WITH OIDS`
`WITHOUT OIDS`

这个可选子句指定新表的行是否应被分配 OID（对象标识符）。默认是有 OID。（如果新表继承自任何有 OID 的表，那么即使命令写的是 `WITHOUT OIDS`，也会强制 `WITH OIDS`。）

指定 `WITHOUT OIDS` 允许用户抑制表行 OID 的生成。对于大表这可能值得一做，因为它可以减少 OID 消耗，从而推迟 32 位 OID 计数器的回卷。一旦计数器回卷，OID 的唯一性就无法再保证，这会大大降低其用处。指定 `WITHOUT OIDS` 还会使表的磁盘存储每行减少 4 字节，从而提高性能。

`CONSTRAINT constraint_name`

列约束或表约束的可选名称。如果未指定，系统会生成一个名称。

`NOT NULL`

该列不允许包含空值。

`NULL`

该列允许包含空值。这是默认情况。

该子句仅为兼容非标准 SQL 数据库而提供，不建议在新应用中使用。

`UNIQUE` (column constraint)
`UNIQUE ( column_name [, ... ] )` (table constraint)

`UNIQUE` 约束指定表中一列或多列组成的一组只能包含唯一值。表级唯一约束的行为与列级唯一约束相同，只是它还能跨越多列。

对于唯一约束，空值不被视为相等。

每个唯一表约束必须命名一组列，
这组列须不同于为该表定义的任何其他唯一约束或主键约束所命名的列集合。
（否则它就只是把同一个约束列出了两次。）

`PRIMARY KEY` (column constraint)
`PRIMARY KEY ( column_name [, ... ] )` (table constraint)

主键约束指定表的一个或多个列只能包含唯一（不重复）的非空值。从技术上讲，`PRIMARY KEY`只是`UNIQUE`和`NOT NULL`的组合，但把一组列标识为主键还为模式设计提供了元数据，因为主键意味着其他表可以依赖这组列作为行的唯一标识符。

一个表只能指定一个主键，无论是作为列约束还是表约束。

主键约束所命名的列集合应当不同于为同一表定义的任何唯一约束所命名的其他列集合。

`CHECK (expression)`

`CHECK` 子句指定一个产生布尔结果的表达式，要使插入或更新操作成功，新行或更新后的行必须满足该表达式。作为列约束指定的检查约束只应引用该列的值，
而出现在表约束中的表达式可以引用多个列。

当前，`CHECK` 表达式不能包含子查询，也不能引用当前行的列之外的变量。

`REFERENCES reftable [ ( refcolumn ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ]` (column constraint)
`FOREIGN KEY ( column [, ... ] ) REFERENCES reftable [ ( refcolumn [ , ... ] ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ]` (table constraint)

这些子句指定外键约束，要求新表的一个或多个列组成的列组只能包含与被引用表 *reftable* 中被引用列 *refcolumn* 的值相匹配的值。如果省略*refcolumn*列表，则使用*reftable*的主键。被引用列必须是被引用表中某个唯一约束或主键约束的列。

插入到引用列中的值会按照给定的匹配类型，与被引用表及其被引用列中的值进行匹配。
共有三种匹配类型：`MATCH FULL`、`MATCH PARTIAL` 和 `MATCH SIMPLE`（默认值）。`MATCH FULL` 不允许多列外键中的某一列为空，除非所有外键列都为空。`MATCH SIMPLE` 允许某些外键列为空而外键的其他部分不为空。`MATCH PARTIAL` 目前尚未实现。

此外，当被引用列中的数据发生变化时，会对本表列中的数据执行某些操作。`ON DELETE`子句指定删除被引用表中的被引用行时要执行的操作。同样，`ON UPDATE`子句指定将被引用表中的被引用列更新为新值时要执行的操作。如果行被更新，但被引用列实际上没有变化，
则不执行任何操作。每个子句可以指定以下操作：

`NO ACTION`

产生错误，指出删除或更新会违反外键约束。这是默认操作。

`RESTRICT`

与 `NO ACTION` 相同，但即使该约束的其余部分是可延迟的且已被延迟，此操作也不会被延迟。

`CASCADE`

分别删除任何引用已删除行的行，或将引用列的值更新为被引用列的新值。

`SET NULL`

将引用列设置为空值。

`SET DEFAULT`

将引用列设置为其默认值。

如果主键列经常更新，可以考虑在外键列上添加索引，使与该外键列关联的 `NO ACTION` 和 `CASCADE` 操作能够更高效地执行。

`DEFERRABLE`
`NOT DEFERRABLE`

这控制约束是否可以延迟。不可延迟的约束会在每条命令之后立即检查。可延迟约束的检查可以推迟到事务结束（使用SET CONSTRAINTS命令）。`NOT DEFERRABLE` 是默认值。目前只有外键约束接受此子句。所有其他类型的约束都不可延迟。

`INITIALLY IMMEDIATE`
`INITIALLY DEFERRED`

如果约束可延迟，则此子句指定检查约束的默认时间。如果约束为`INITIALLY IMMEDIATE`，则在每条语句之后检查。这是默认值。如果约束为`INITIALLY DEFERRED`，则仅在事务结束时检查。可以使用SET CONSTRAINTS命令更改约束检查时间。

`ON COMMIT`

可以使用 `ON COMMIT` 控制临时表在事务块结束时的行为。三种选项如下：

`PRESERVE ROWS`

在事务结束时不执行任何特殊操作。这是默认行为。

`DELETE ROWS`

临时表中的所有行都会在每个事务块结束时删除。实际上，每次提交时都会自动执行一次TRUNCATE。

`DROP`

在当前事务块结束时删除临时表。

注解

- 每当应用使用 OID 来标识表中的特定行时，建议在该表的 `oid` 列上创建唯一约束，以确保即使计数器回卷之后，表中的 OID 也确实能唯一标识行。不要假定 OID 在不同表之间是唯一的；如果需要数据库范围的唯一标识符，请组合使用 `tableoid` 和行 OID 来达到目的。（未来的 PostgreSQL 版本很可能会为每个表使用单独的 OID 计数器，因此把 `tableoid` 包含进来以获得数据库范围的唯一标识符，将成为必要而非可选。）

> **提示**
>
> 对于没有主键的表，不建议使用 `WITHOUT OIDS`，因为既没有 OID 也没有唯一数据键时，很难标识特定行。

- PostgreSQL 会为每一个唯一约束和主键约束自动创建一个索引来强制唯一性。因此，没有必要显式地为主键列创建索引（更多信息见CREATE INDEX）。
- 在当前的实现中，唯一约束和主键不会被继承。这使得继承与唯一约束的组合相当不实用。

示例

创建表 `films` 和表 `distributors`：

```
CREATE TABLE films (
```

```
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY DEFAULT nextval('serial'),
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

创建一个带二维数组列的表：

```
CREATE TABLE array (
    vector  int[][]
);
```

为表`films`定义一个唯一表约束。唯一表约束可以定义在表的一列或多列上：

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

定义一个列检查约束：

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

定义一个表检查约束：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

为表`films`定义一个主键表约束：

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT code_title PRIMARY KEY(code,title)
```

```
);
```

为表distributors定义一个主键约束。下面两个示例是等价的，第一个使用表约束语法，第二个使用列约束语法：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

为列name指定一个字面常量默认值，将列did的默认值设为从某个序列对象中取下一个值，并让modtime的默认值为插入该行的时间：

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

在表distributors上定义两个NOT NULL列约束，其中一个显式指定了名称：

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

为name列定义一个唯一约束：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

同样的唯一约束用表约束指定：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name)
);
```

兼容性

CREATE TABLE 命令符合 SQL92 以及 SQL99 的一个子集，但有以下例外。

临时表

尽管 CREATE TEMPORARY TABLE 的语法看起来类似于 SQL 标准，但其效果并不相同。按标准，临时表只需定义一次，并会自动存在于每个需要它的会话中（内容初始为空）。而

PostgreSQL 要求每个会话都为每个要使用的临时表发出自己的 `CREATE TEMPORARY TABLE` 命令。这使不同会话可以出于不同目的使用相同的临时表名；而标准做法则要求给定临时表名的所有实例都必须具有相同的表结构。

标准对临时表行为的定义在实践中被广泛忽略。PostgreSQL 在这一点上的行为与多种其他 SQL 数据库相似。

标准中全局临时表和局部临时表的区分在 PostgreSQL 中并不存在，因为这一区分依赖于模块的概念，而 PostgreSQL 没有这一概念。出于兼容性考虑，PostgreSQL 会接受在临时表声明中使用 `GLOBAL` 和 `LOCAL` 关键字，但它们没有效果。

临时表的 `ON COMMIT` 子句也与 SQL 标准相似，但存在一些差异。如果省略 `ON COMMIT` 子句，SQL 规定默认行为是 `ON COMMIT DELETE ROWS`。然而，PostgreSQL 中的默认行为是 `ON COMMIT PRESERVE ROWS`。SQL 中不存在 `ON COMMIT DROP` 选项。

列检查约束

SQL 标准规定，`CHECK` 列约束只能引用其所作用的列；只有 `CHECK` 表约束才能引用多列。PostgreSQL 并不强制这一限制；它对列检查约束和表检查约束一视同仁。

NULL“约束”

`NULL`“约束”（实际上并不是约束）是 PostgreSQL 对 SQL 标准的扩展；提供它是为了与其他一些数据库系统兼容（以及与 `NOT NULL` 约束保持对称）。由于它本来就是任意列的默认情况，所以它的存在只是噪声。

继承

通过 `INHERITS` 子句实现的多重继承是 PostgreSQL 的语言扩展。SQL99（但 SQL92 不）使用不同的语法和语义定义了单继承。PostgreSQL 尚不支持 SQL99 风格的继承。

对象 ID

PostgreSQL 的 OID 概念不是标准的一部分。

零列表

PostgreSQL 允许创建没有列的表（例如 `CREATE TABLE foo();`）。这是对 SQL 标准的扩展，标准不允许零列的表。零列的表本身并不十分有用，但若禁止它们，就会让 `ALTER TABLE DROP COLUMN` 出现奇怪的特殊情况，因此忽略这一规范限制看起来更整洁。

另见

ALTER TABLE, DROP TABLE

CREATE TABLE AS

CREATE TABLE AS — 根据查询的结果创建一个新表

大纲

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name [
 (column_name [, ...] ) ]
    AS query
```

描述

CREATE TABLE AS创建一个表并用一条 SELECT命令或一条运行已预备的 SELECT命令的EXECUTE所计算出的数据填充它。表列具有与SELECT的输出列相关联的名称和数据类型（不过你可以通过给出新列名的显式列表来覆盖列名）。

CREATE TABLE AS与创建视图有些相似，但两者实际上完全不同：它创建一个新表并且只对查询求值一次来初始地填充新表。新表不会跟踪查询源表的后续变化。相反，
视图在每次被查询时都会重新求值其定义的SELECT语句。

参数

TEMPORARY or TEMP

如果指定，该表将创建为临时表。细节请参阅CREATE TABLE。

table_name

要创建的表的名称（可以是模式限定的）。

column_name

新表中一列的名称。如果没有提供列名，则从查询的输出列名中取得。如果表是由一条EXECUTE命令创建的，目前无法指定列名列表。

query

一条查询语句（即一条SELECT命令或一条运行已预备的SELECT命令的 EXECUTE命令）。允许语法的描述分别参阅SELECT或EXECUTE。

注解

这条命令在功能上等价于SELECT INTO，但它更受推荐，因为它不太容易与SELECT ... INTO语法的其他用法混淆。

兼容性

这条命令仿效Oracle的一个特性。SQL 标准中没有功能等价的命令。不过，把CREATE TABLE和INSERT ... SELECT组合起来只需稍多功夫就能完成同样的事情。

另见

CREATE TABLE, CREATE VIEW, EXECUTE, SELECT, SELECT INTO

CREATE TRIGGER

CREATE TRIGGER — 定义一个新的触发器

大纲

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    EXECUTE PROCEDURE funcname ( arguments )
```

描述

CREATE TRIGGER创建一个新触发器。触发器会关联到指定的表，并在发生某些事件时执行指定的函数*funcname*。

可以指定触发器在尝试对某一行执行该操作之前引发（即在检查约束以及尝试执行INSERT、UPDATE或DELETE之前）；也可以在该操作完成之后引发（即在检查约束以及完成 INSERT、UPDATE或 DELETE之后）。如果触发器在事件之前引发，则它可以跳过对当前行的操作，或者修改待插入的行（仅适用于INSERT和UPDATE操作）。如果触发器在事件之后引发，则所有更改（包括最后一次插入、更新或删除）对该触发器都是“可见”的。

被标记为FOR EACH ROW的触发器会针对该操作修改的每一行调用一次。例如，一个影响 10 行的DELETE会导致目标关系上的 ON DELETE触发器分别被调用 10 次，即每个被删除的行调用一次。相比之下，被标记为FOR EACH STATEMENT的触发器对于任意给定操作只执行一次，不管该操作修改了多少行（特别是，修改零行的操作仍会导致所有适用的 FOR EACH STATEMENT触发器执行）。

如果针对同一事件定义了多个同类触发器，它们将按名称的字母顺序引发。

SELECT 不会修改任何行，因此无法创建 SELECT 触发器。对于这类情形，规则和视图更为合适。

参见第 35 章以了解有关触发器的更多信息。

参数

name

要赋予新触发器的名称。它必须不同于同一表上任何其他触发器的名称。

BEFORE
AFTER

决定该函数是在事件之前还是之后被调用。

event

INSERT、UPDATE或DELETE之一；它指定将引发该触发器的事件。可以使用OR指定多个事件。

table

该触发器所作用的表的名称（可以是模式限定的）。

FOR EACH ROW
FOR EACH STATEMENT

这指定触发器过程是针对受触发事件影响的每一行引发一次，还是针对每条 SQL 语句只引发一次。如果两者都未指定，则默认为FOR EACH STATEMENT。

funcname

一个由用户提供的函数，声明为不接受参数并返回 trigger 类型；当触发器被触发时，就会执行该函数。

arguments

一个可选的、以逗号分隔的参数列表，在执行触发器时会传给该函数。
这些参数是字符串字面常量。这里也可以写简单名称和数字常量，
但它们都会被转换成字符串。请查阅该触发器函数所用实现语言的说明，
以了解如何在函数内部访问这些参数；这可能与普通函数参数不同。

注意

要在表上创建触发器，用户必须拥有该表的TRIGGER权限。

在PostgreSQL 7.3 之前的版本中，必须将触发器函数声明为返回占位类型opaque，而不是trigger。为了支持加载旧转储文件，CREATE TRIGGER将接受声明为返回opaque的函数，但会发出通知，并将该函数声明的返回类型更改为trigger。

用DROP TRIGGER移除一个触发器。

示例

第 35.4 节包含一个完整示例。

兼容性

PostgreSQL 中的 CREATE TRIGGER 语句实现了 SQL99 标准的一个子集。（SQL92 中没有关于触发器的规定。）缺少以下功能：

- SQL99 允许触发器在对特定列的更新时引发（例如 AFTER UPDATE OF col1, col2）。
- SQL99 允许为“旧”行或表和“新”行或表定义别名，供被触发动作的定义使用（例如 CREATE TRIGGER ... ON tablename REFERENCING OLD ROW AS somename NEW ROW AS othername ...）。由于 PostgreSQL 允许用任意多种用户自定义语言编写触发器过程，对数据的访问以各语言特定的方式处理。
- PostgreSQL 只允许被触发动作执行一个用户自定义函数。SQL99 允许执行若干其他 SQL 命令（例如 CREATE TABLE）作为被触发动作。通过创建一个执行所需命令的用户自定义函数，这一限制不难绕过。

SQL99 规定多个触发器应按创建时间的先后顺序引发。PostgreSQL 使用名称顺序，这被判定为使用起来更方便。

使用 OR 为单个触发器指定多个动作的能力，是 PostgreSQL 对 SQL 标准的扩展。

另请参阅

CREATE FUNCTION, ALTER TRIGGER, DROP TRIGGER

CREATE TYPE

CREATE TYPE — 定义一种新的数据类型

大纲

```
CREATE TYPE name AS
    ( attribute_name data_type [, ... ] )

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
)
```

描述

`CREATE TYPE` 为当前数据库注册一种新的数据类型。定义类型的用户将成为其所有者。

如果给出了模式名，则该类型会在指定模式中创建；否则会在当前模式中创建。
类型名必须不同于同一模式中任何现有类型或域的名称。（由于表也具有关联的数据类型，
类型名还必须不同于同一模式中任何现有表的名称。）

复合类型

`CREATE TYPE` 的第一种形式创建一个复合类型。复合类型由属性名称和数据类型的列表指定。它本质上与表的行类型相同，但当只想定义一个类型时，使用 `CREATE TYPE` 可以避免创建实际的表。独立的复合类型适合作为函数的返回类型。

基本类型

`CREATE TYPE` 的第二种形式创建新的基本类型（标量类型）。参数可以按任意顺序出现，不必是上面展示的顺序，而且大多数是可选的。在定义类型之前，你必须（用 `CREATE FUNCTION`）先注册两个或更多函数。支持函数 *input_function* 和 *output_function* 是必需的，而函数 *receive_function* 和 *send_function* 是可选的。一般来说，这些函数必须用 C 或其他低级语言编写。

input_function 把该类型的外部文本表示转换为为该类型定义的操作符和函数所使用的内部表示。*output_function* 执行反向的转换。输入函数可以声明为接受一个 `cstring` 类型的参数，或者接受 `cstring`、`oid`、`integer` 三个类型的参数。第一个参数是作为 C 字符串的输入文本，第二个参数是数组类型情况下的元素类型，第三个是目标列的 `typmod`（如果已知）。输入函数应当返回该数据类型本身的值。输出函数可以声明为接受一个新数据类型的参数，或者接受两个参数且第二个参数为 `oid` 类型。第二个参数同样是数组类型时的数组元素类型。输出函数应当返回 `cstring` 类型。

可选的 *receive_function* 把该类型的外部二进制表示转换为内部表示。如果不提供该函数，该类型就无法参与二进制输入。二进制表示应当选择转换成内部形式代价低廉、同时具有合理可移植性的形式。（例如，标准整数数据类型使用网络字节序作为外部二进制表示，而内部表示是机器的本机字节序。）接收函数应当执行充分的检查以确保值是有效的。接收函数可以声明为接受一个 `internal` 类型的参数，或者 `internal` 和 `oid` 两个类型的参数。它必须返回该数据类型本身的值。（第一个参数是指向持有收到的字节串的 `StringInfo` 缓冲区的指针；可选的第二个参数是数组类型情况下的元素类型。）类似地，可选的 *send_function* 把内部表示转换为外部二进制表示。如果不提供该函数，该类型就无法参与二进制输出。发送函数可以声明为接受一个新数据类型的参数，或者接受两个参数且第二个参数为 `oid` 类型。第二个参数同样是数组类型时的数组元素类型。发送函数必须返回 `bytea` 类型。

读到这里，你可能会问：既然新类型本身还没创建，输入和输出函数怎么能声明为返回或接受这个新类型呢？答案是，必须先创建输入函数，然后是输出函数（以及需要的二进制 I/O 函数），最后才是数据类型本身。PostgreSQL会首先把新数据类型的名称看作输入函数的返回类型。此时它会创建一个 “shell” 类型，它只是系统目录中的一个占位条目，并把输入函数的定义链接到该 shell 类型。类似地，其他函数也会链接到（此时已经存在的）shell 类型。最后，`CREATE TYPE`用完整的类型定义替换该 shell 条目，此后新类型就可以使用了。

虽然新类型内部表示的细节只有 I/O 函数以及你为该类型编写的其他函数才知道，但仍有若干内部表示属性必须向 PostgreSQL声明。其中最重要的是 *internallength*。基础类型可以是定长的，此时*internallength*为正整数；也可以是变长的，此时将*internallength*设为 `VARIABLE`。（在内部，这通过把 `typlen`设为 -1 表示。）所有变长类型的内部表示都必须以一个 4 字节整数开头，用来给出该类型该值的总长度。

可选标志`PASSEDBYVALUE`表示该数据类型的值按值传递，而不是按引用传递。内部表示大于 `Datum`类型大小（多数机器上为 4 字节，少数为 8 字节）的类型不能按值传递。

*alignment*参数指定该数据类型所需的存储对齐方式。允许的值分别对应按 1、2、4 或 8 字节边界对齐。注意，变长类型的对齐至少必须为 4，因为它们的第一个组成部分必然是一个`int4`。

*storage*参数允许为变长数据类型选择存储策略。（定长类型只允许 `plain`。）`plain`表示该类型数据始终内联存储且不压缩。`extended`表示系统会先尝试压缩较长的数据值，如果仍然过长，就把该值移出主表行。`external`允许把值移出主表，但系统不会尝试压缩它。`main`允许压缩，但不鼓励把值移出主表。（采用这种存储策略的数据项在没有其他办法让一行适配时仍可能被移出主表，但与`extended`和`external` 数据项相比，它们会被优先保留在主表中。）

如果用户希望该数据类型列的默认值不是空值，可以指定默认值。用 `DEFAULT`关键字指定默认值。（该默认值可以被附加到具体列上的显式`DEFAULT`子句覆盖。）

若要表明一种类型是数组类型，可用`ELEMENT` 关键字指定数组元素的类型。例如，要定义由 4 字节整数（`int4`）构成的数组，可指定 `ELEMENT = int4`。更多关于数组类型的细节见下文。

若要指定该类型数组在外部表示中用于分隔各值的分隔符，可把 *delimiter*设为特定字符。默认分隔符是逗号（`,`）。注意，这个分隔符关联的是数组元素类型，而不是数组类型本身。

数组类型

每当创建用户定义的基础数据类型时，PostgreSQL 都会自动创建一个关联的数组类型，其名称由基础类型名前加一个下划线组成。解析器理解这一命名约定，会把对`foo[]`类型列的请求转换为对`_foo`类型的请求。这种隐式创建的数组类型是变长的，并使用内置的输入/输出函数 `array_in`和`array_out`。

如果系统会自动创建正确的数组类型，你可能会合理地问，为什么还需要ELEMENT选项。ELEMENT唯一有用的情况是：你正在创建一种定长类型，而它在内部恰好是若干相同元素组成的数组，并且除了为整个类型提供的操作之外，你还希望允许通过下标直接访问这些元素。例如，类型name就允许用这种方式访问其组成元素char。一个二维的 point类型可以允许像point[0]和point[1]这样访问它的两个分量数字。注意，这种机制只适用于内部形式恰好是一串相同定长字段的定长类型。可用下标访问的变长类型必须采用array_in和array_out使用的通用内部表示。由于历史原因（也就是说这显然不对，但现在改已经太晚了），定长数组类型的下标从零开始，而变长数组则从一开始。

参数

name

要创建的类型名称（可选地带有模式限定）。

attribute_name

复合类型的一个属性（列）的名称。

data_type

将成为复合类型一列的现有数据类型名称。

input_function

将数据从类型的外部文本形式转换为内部形式的函数名。

output_function

将数据从类型的内部形式转换为外部文本形式的函数名。

receive_function

将数据从类型的外部二进制形式转换成内部形式的函数名。

send_function

将数据从类型的内部形式转换为外部二进制形式的函数名。

internallength

一个数字常量，用于指定新类型内部表示的字节长度。默认假定它是变长的。

alignment

该数据类型的存储对齐需求。如果被指定，它必须是 char、int2、int4或者double。默认是 int4。

storage

该数据类型的存储策略。如果被指定，必须是 plain、external、extended或者main。默认是plain。

default

该数据类型的默认值。若省略，默认值为 null。

element

正在创建的类型是数组；该参数指定数组元素类型。

delimiter

在由该类型构成的数组中各值之间使用的分隔符字符。

注意

用户定义的类型名不能以下划线字符（_）开头，并且长度最多只能是 62 个字符（通常来说是 NAMEDATALEN - 2，而不是其他名称允许的 NAMEDATALEN - 1 个字符）。以下划线开头的类型名保留给内部创建的数组类型名使用。

在PostgreSQL 7.3 以前的版本中，通常完全不创建 shell 类型，而是将函数中对类型名的前向引用替换为占位伪类型 opaque。在 7.3 以前，cstring参数和结果也必须声明为opaque。为了支持载入旧转储文件，CREATE TYPE会接受使用opaque声明的 I/O 函数，但会发出通知，并修改函数声明以使用正确的类型。

示例

这个示例创建一种复合类型，并在函数定义中使用它：

```
CREATE TYPE compfoo AS (f1 int, f2 text);
CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS
  'SELECT fooid, fooname FROM foo' LANGUAGE SQL;
```

这个示例创建基础类型box，然后在表定义中使用它：

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

如果box的内部结构是由四个 float4元素构成的数组，则也可以改为这样写：

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

这样就能通过下标访问 box 值的各个分量。除此之外，该类型的行为与前例相同。

这个示例创建一种大对象类型，并在表定义中使用它：

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
```

```
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

更多例子（包括合适的输入和输出函数）见第 33 章。

兼容性

这个 `CREATE TYPE` 命令是 PostgreSQL 的扩展。SQL99 中也有一个 `CREATE TYPE` 语句，但在细节上大不相同。

另请参阅

CREATE FUNCTION, DROP TYPE

CREATE USER

CREATE USER — 定义一个新的数据库用户账号

大纲

```
CREATE USER name [ [ WITH ] option [ ... ] ]

where option can be:

      SYSID uid
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | CREATEDB | NOCREATEDB
    | CREATEUSER | NOCREATEUSER
    | IN GROUP groupname [, ...]
    | VALID UNTIL 'abstime'
```

描述

`CREATE USER` 向一个 PostgreSQL 数据库集簇中添加一个新用户。关于管理用户和认证的信息，请参阅第 17 章和第 19 章。你必须是数据库超级用户才能使用这个命令。

参数

name

用户的名称。

uid

`SYSID` 子句可以用来选择正被创建的用户的 PostgreSQL 用户 ID。通常这并不需要，但在你需要重新创建某个孤立对象的拥有者时可能有用。

如果没有指定，将默认使用已分配的最大用户 ID 加一（最小为 100）。

password

设置用户的口令。如果你不打算使用口令认证，可以省略这个选项，但那样的话，
如果你以后决定切换到口令认证，该用户将无法连接。口令可以稍后设置或更改，使用ALTER USER。

`ENCRYPTED`
`UNENCRYPTED`

这些关键字控制口令在系统目录中是否以加密形式存储。（如果两者都未指定，
默认行为由配置参数 `PASSWORD_ENCRYPTION` 决定。）如果给出的口令字符串已经是 MD5 加密格式，那么不管指定的是 `ENCRYPTED` 还是 `UNENCRYPTED`，它都会原样加密存储（因为系统无法解密所指定的加密口令字符串）。这允许在转储/恢复期间重新装载加密的口令。

注意，较老的客户端可能不支持处理以加密形式存储的口令所需的 MD5 认证机制。

`CREATEDB`
`NOCREATEDB`

这些子句定义用户创建数据库的能力。如果指定了 `CREATEDB`，将被定义的用户将被允许创建他自己的数据库。使用 `NOCREATEDB` 将拒绝用户创建数据库的能力。如果省略这个子句，默认使用 `NOCREATEDB`。

`CREATEUSER`
`NOCREATEUSER`

这些子句决定是否允许用户自己创建新用户。
这个选项还会让该用户成为可以越过所有访问限制的超级用户。
省略这个子句将把该用户的这个属性值设置为 `NOCREATEUSER`。

groupname

把该用户作为新成员插入其中的组的名称。可以列出多个组名。

abstime

`VALID UNTIL` 子句设置一个绝对时间，超过该时间后用户的口令不再有效。如果省略这个子句，登录将永久有效。

注意

使用ALTER USER更改用户的属性，使用DROP USER删除一个用户。使用ALTER GROUP把用户添加到组中或从组中移除用户。

PostgreSQL 包含一个程序createuser，它具有与 `CREATE USER` 相同的功能（事实上，它调用这个命令），但可以从命令 shell 运行。

示例

创建一个没有口令的用户：

```
CREATE USER jonathan;
```

创建一个带口令的用户：

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

创建一个带口令的用户，口令有效到 2004 年底。2005 年到来一秒之后，口令不再有效。

```
CREATE USER miriam WITH PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

创建一个用户可以创建数据库的账号：

```
CREATE USER manuel WITH PASSWORD 'jw8s0F4' CREATEDB;
```

兼容性

`CREATE USER` 语句是一种 PostgreSQL 扩展。SQL 标准把用户的定义留给实现。

另见

ALTER USER, DROP USER, createuser

CREATE VIEW

CREATE VIEW — 定义一个新视图

大纲

```
CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ] AS query
```

描述

`CREATE VIEW`定义一个视图，其内容来自一个查询。该视图不会被实际物化。相反，每次在查询中引用该视图时，都会执行定义它的查询。

`CREATE OR REPLACE VIEW`与之类似，但如果已经存在同名视图，则会替换它。只能用一个生成相同列集合（即相同的列名和数据类型）的新查询来替换视图。

如果给出了模式名称（例如，`CREATE VIEW myschema.myview ...`），则视图将在指定的模式中创建。否则，它将在当前模式中创建。视图的名称必须与同一模式中的任何其他视图、表、序列或索引的名称不同。

参数

name

要创建的视图名称（可以带模式限定）。

column_name

视图各列使用的名称列表，可选。若未给出，则从查询中推断列名。

query

一个查询（即`SELECT`语句），它将提供该视图的列和行。

有关有效查询的更多信息，请参见SELECT。

注意

当前，视图是只读的：系统不允许在视图上执行插入、更新或删除。你可以通过创建规则，把对视图的插入等操作重写为对其他表的适当动作，来获得可更新视图的效果。
更多信息见CREATE RULE。

使用`DROP VIEW`语句删除视图。

应留意视图列的名称和类型是否按预期确定。例如：

```
CREATE VIEW vista AS SELECT 'Hello World';
```

这种写法在两方面都不好：列名默认是`?column?`，而且列数据类型默认为`unknown`。
如果想要在视图结果中使用字符串字面量，可使用类似下面的写法：

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

对视图中引用的表的访问由视图所有者的权限决定。不过，
视图中调用的函数与直接从使用该视图的查询中调用时的处理方式相同。因此，
视图的使用者必须拥有调用视图所用全部函数的权限。

示例

创建一个由所有喜剧电影组成的视图：

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

兼容性

SQL 标准为`CREATE VIEW`语句指定了一些额外的能力：

```
CREATE VIEW name [ ( column [, ...] ) ]
    AS query
    [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

完整 SQL 命令的可选子句有：

`CHECK OPTION`

此选项与可更新视图有关。视图上的所有 `INSERT` 和 `UPDATE` 命令都会被检查，以确保数据满足定义视图的条件（也就是说，新数据可以通过该视图看到）。如果不满足，
更新将被拒绝。

`LOCAL`

在该视图上检查完整性。

`CASCADE`

在该视图以及任何依赖视图上检查完整性。如果既未指定`CASCADE`也未指定`LOCAL`，
则假定使用`CASCADE`。

`CREATE OR REPLACE VIEW` 是一种 PostgreSQL 语言扩展。

DEALLOCATE

DEALLOCATE — 释放一个预备语句

大纲

```
DEALLOCATE [ PREPARE ] plan_name
```

描述

`DEALLOCATE`用于释放一个先前预备的 SQL 语句。如果你不显式地释放一个预备语句，它会在会话结束时被释放。

关于预备语句的更多信息，见PREPARE。

参数

`PREPARE`

这个关键字被忽略。

`plan_name`

要释放的预备语句的名称。

兼容性

SQL 标准包含一条`DEALLOCATE` 语句，但它只用于嵌入式 SQL。

DECLARE

DECLARE — 定义一个游标

大纲

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] } ]
```

描述

DECLARE允许用户创建游标，游标可用于从较大的查询中一次取出少量行。游标可以使用FETCH以文本或二进制格式返回数据。

普通游标以文本格式返回数据，就像SELECT所产生的结果一样。
由于数据在内部是以二进制格式存储的，系统必须做一次转换才能产生文本格式。
而信息以文本形式返回后，客户端应用又可能需要把它转换回二进制格式才能处理。此外，
文本格式的数据通常比二进制格式占用的空间更大。二进制游标以二进制表示返回数据，
处理起来可能更方便。不过，如果你本来就打算把数据显示为文本，
那么以文本形式检索会省去客户端的一些工作量。

举例来说，如果某个查询从一个整型列返回值 1，那么默认游标会返回字符串 1，而二进制游标则会返回一个 4 字节字段，其中包含该值的内部表示（采用大端字节序）。

应谨慎使用二进制游标。许多应用程序（包括 psql）都没有准备好处理二进制游标，而是期望返回的数据为文本格式。

> **注意**
>
> 当客户端应用使用“扩展查询”协议发出 FETCH 命令时，Bind 协议消息指定以文本还是二进制格式检索数据。这个选择覆盖游标定义的方式。因此在使用扩展查询协议时，
> 二进制游标这一概念本身已经过时—— 任何游标都可以按文本或二进制对待。

参数

name

要创建的游标名称。

BINARY

使游标以二进制格式而不是文本格式返回数据。

INSENSITIVE

表示从游标取出的数据不应受到游标存在期间对其底层表所做更新的影响。
在PostgreSQL中，所有游标都是不敏感的；该关键字目前没有任何效果，只是为兼容 SQL 标准而存在。

SCROLL
NO SCROLL

SCROLL指定游标可用于以非顺序方式（例如向后）取出行。根据查询执行计划的复杂程度，指定SCROLL可能会给查询执行带来性能开销。NO SCROLL指定游标不能以非顺序方式取出行。

WITH HOLD
WITHOUT HOLD

WITH HOLD指定在创建该游标的事务成功提交后，仍可继续使用该游标。WITHOUT HOLD指定该游标不能在创建它的事务之外使用。如果既未指定WITHOUT HOLD也未指定WITH HOLD，默认值是WITHOUT HOLD。

query

一个将为游标提供要返回的行的SELECT命令。有关有效查询的更多信息，请参阅SELECT。

FOR READ ONLY
FOR UPDATE

FOR READ ONLY表示该游标将以只读方式使用。FOR UPDATE表示该游标将用于更新表。由于 PostgreSQL目前不支持游标更新，指定 FOR UPDATE会导致一条错误消息，而指定 FOR READ ONLY则没有任何效果。

column

将被游标更新的列。由于PostgreSQL目前不支持游标更新，FOR UPDATE子句会引发一条错误消息。

关键字BINARY、INSENSITIVE和SCROLL可以按任意顺序出现。

注解

除非指定了WITH HOLD，否则该命令创建的游标只能在当前事务中使用。因此，DECLARE若不带WITH HOLD，在事务块之外就毫无用处：游标只能存活到该语句执行完成。所以，如果在事务块之外使用这种命令，PostgreSQL会报错。可使用BEGIN、COMMIT和ROLLBACK来定义事务块。

如果指定了WITH HOLD，且创建游标的事务成功提交，那么在同一会话中的后续事务里仍可继续访问该游标。（但如果创建事务被中止，游标会被移除。）使用WITH HOLD创建的游标会在对其发出显式CLOSE命令时关闭，或在会话结束时关闭。在当前实现中，这种游标所表示的行会被复制到临时文件或内存区域中，以便它们在后续事务中仍然可用。

在定义将用于向后取出行的游标时，应指定SCROLL选项。这是 SQL 标准所要求的。不过，为了兼容早期版本，如果游标的查询计划足够简单，以致支持向后取出不需要额外开销，PostgreSQL也允许在未指定SCROLL的情况下向后取出行。不过，建议应用开发者不要依赖于从未用SCROLL创建的游标中向后取出行。如果指定了NO SCROLL，那么无论如何都不允许向后取出行。

SQL 标准只为嵌入式SQL中的游标作出规定。PostgreSQL服务器没有为游标实现OPEN语句；游标在声明时即被视为打开。不过，ECPG是PostgreSQL的嵌入式 SQL 预处理器，它支持标准 SQL 的游标约定，包括涉及DECLARE和OPEN语句的那些约定。

示例

声明一个游标：

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

更多游标用法示例见FETCH。

兼容性

SQL 标准只允许在嵌入式SQL和模块中使用游标。PostgreSQL允许以交互方式使用游标。

SQL 标准允许游标更新表数据。所有 PostgreSQL游标都是只读的。

二进制游标是PostgreSQL的一种扩展。

DELETE

DELETE — 删除一个表中的行

大纲

```
DELETE FROM [ ONLY ] table [ WHERE condition ]
```

描述

DELETE从指定表中删除满足WHERE子句的行。如果省略 WHERE子句，则删除表中的所有行。结果是一个有效但为空的表。

> **提示**
>
> TRUNCATE是一个 PostgreSQL扩展，它提供了一种从表中删除所有行的更快机制。

默认情况下，DELETE将删除指定表及其所有子表中的行。如果只想从提到的特定表中删除，必须使用 ONLY子句。

要从一个表中删除行，你必须拥有该表上的DELETE权限，同时对*condition*中读取其值的任何表拥有SELECT权限。

参数

table

一个已存在表的名称（可以有模式限定）。

condition

一个返回 boolean 类型值的值表达式，它决定要删除哪些行。

输出

成功完成时，DELETE命令返回如下形式的命令标签

```
DELETE count
```

其中 *count* 是删除的行数。如果 *count* 为 0，表示没有行满足 *condition*（这不被视为错误）。

示例

删除除音乐剧外的所有电影：

```
DELETE FROM films WHERE kind <> 'Musical';
```

清空表 films：

```
DELETE FROM films;
```

兼容性

这条命令符合 SQL 标准。

DROP AGGREGATE

DROP AGGREGATE — 删除一个聚集函数

大纲

```
DROP AGGREGATE name ( type ) [ CASCADE | RESTRICT ]
```

描述

`DROP AGGREGATE`将删除一个已存在的聚集函数。要执行这条命令，当前用户必须是该聚集函数的拥有者。

参数

name

一个已存在聚集函数的名称（可以有模式限定）。

type

聚集函数的参数数据类型，如果函数接受任意数据类型则为 `*`。

`CASCADE`

自动删除依赖于该聚集函数的对象。

`RESTRICT`

如果有任何对象依赖于该聚集函数，则拒绝删除它。这是默认值。

示例

删除类型 `integer` 的聚集函数 `myavg`：

```
DROP AGGREGATE myavg(integer);
```

兼容性

SQL 标准中没有 `DROP AGGREGATE` 语句。

另见

ALTER AGGREGATE, CREATE AGGREGATE

DROP CAST

DROP CAST — 删除一个类型转换

大纲

```
DROP CAST (sourcetype AS targettype) [ CASCADE | RESTRICT ]
```

描述

`DROP CAST`移除一个先前定义的类型转换。

要删除一种类型转换，你必须拥有其源数据类型或目标数据类型。
这与创建该类型转换所需的权限相同。

参数

sourcetype

该类型转换的源数据类型的名称。

targettype

该类型转换的目标数据类型的名称。

`CASCADE`
`RESTRICT`

这些关键字没有任何作用，因为没有对象依赖于类型转换。

示例

要删除从类型`text`到类型`int`的类型转换：

```
DROP CAST (text AS int);
```

兼容性

`DROP CAST`命令符合 SQL 标准。

另见

CREATE CAST

DROP CONVERSION

DROP CONVERSION — 移除一个转换

大纲

```
DROP CONVERSION name [ CASCADE | RESTRICT ]
```

描述

`DROP CONVERSION` 移除一个先前定义的转换。要能删除一个转换，你必须拥有该转换。

参数

name

转换的名称。转换名可以带模式限定。

`CASCADE`
`RESTRICT`

这些关键字没有任何效果，因为转换上没有依赖。

示例

要删除名为 `myname` 的转换：

```
DROP CONVERSION myname;
```

兼容性

SQL 标准中没有 `DROP CONVERSION` 语句。

另见

ALTER CONVERSION, CREATE CONVERSION

DROP DATABASE

DROP DATABASE — 删除一个数据库

大纲

`DROP DATABASE name`

描述

`DROP DATABASE`删除一个数据库。它删除该数据库的目录项并删除包含数据的目录。它只能由数据库所有者执行。而且，当你或任何其他人正连接到目标数据库时，不能执行它。（连接到`template1`或任何其他数据库来发出这条命令。）

`DROP DATABASE`无法撤销。请小心使用！

参数

name

要删除的数据库名。

注解

`DROP DATABASE`不能在事务块内部执行。

当连接到目标数据库时无法执行这条命令。因此，改用程序dropdb可能更方便，它是这条命令的一个包装。

兼容性

SQL 标准中没有 `DROP DATABASE` 语句。

另见

CREATE DATABASE

DROP DOMAIN

DROP DOMAIN — 移除一个域

大纲

```
DROP DOMAIN name [, ...]  [ CASCADE | RESTRICT ]
```

描述

`DROP DOMAIN`移除一个域。只有域的拥有者才能移除它。

参数

name

要移除的现有域的名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该域的对象（例如表列）。

`RESTRICT`

如果有任何对象依赖于该域，则拒绝删除它。这是默认值。

示例

要移除域`box`：

```
DROP DOMAIN box;
```

兼容性

此命令符合 SQL 标准。

另见

CREATE DOMAIN

DROP FUNCTION

DROP FUNCTION — 移除一个函数

大纲

```
DROP FUNCTION name ( [ type [, ...] ] ) [ CASCADE | RESTRICT ]
```

描述

`DROP FUNCTION`移除一个现有函数的定义。要执行此命令，用户必须是该函数的拥有者。必须指定该函数的参数类型，因为可能存在多个名称相同但参数列表不同的函数。

参数

name

现有函数的名称（可以被模式限定）。

type

函数某个参数的数据类型。

`CASCADE`

自动删除依赖于该函数的对象（例如操作符或触发器）。

`RESTRICT`

如果有任何对象依赖于该函数，则拒绝删除它。这是默认值。

示例

此命令删除平方根函数：

```
DROP FUNCTION sqrt(integer);
```

兼容性

`DROP FUNCTION`语句在 SQL 标准中有定义，但与该命令不兼容。

另见

CREATE FUNCTION, ALTER FUNCTION

DROP GROUP

DROP GROUP — 移除一个用户组

大纲

```
DROP GROUP name
```

描述

`DROP GROUP` 移除指定的组。组中的用户不会被删除。

参数

name

现有组的名称。

示例

删除一个组：

```
DROP GROUP staff;
```

兼容性

SQL 标准中没有 `DROP GROUP` 语句。

另见

ALTER GROUP, CREATE GROUP

DROP INDEX

DROP INDEX — 移除一个索引

大纲

```
DROP INDEX name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP INDEX` 从数据库系统中删除一个现有的索引。要执行这个命令，你必须是指定索引的拥有者。

参数

name

要移除的索引的名称（可带模式限定）。

`CASCADE`

自动删除依赖于该索引的对象。

`RESTRICT`

如果有任何对象依赖于该索引，则拒绝删除它。这是默认行为。

示例

这条命令将移除索引 `title_idx`：

```
DROP INDEX title_idx;
```

兼容性

`DROP INDEX` 是一种 PostgreSQL 语言扩展。SQL 标准中没有关于索引的规定。

另见

CREATE INDEX

DROP LANGUAGE

DROP LANGUAGE — 移除一个过程语言

大纲

```
DROP [ PROCEDURAL ] LANGUAGE name [ CASCADE | RESTRICT ]
```

描述

`DROP LANGUAGE`将删除先前注册的名为 *name*的过程语言的定义。

参数

name

现有过程语言的名称。为向后兼容，名称可以用单引号括起。

`CASCADE`

自动删除依赖于该语言的对象（例如以该语言编写的函数）。

`RESTRICT`

如果有任何对象依赖于该语言，则拒绝删除它。这是默认值。

示例

此命令移除过程语言`plsample`：

```
DROP LANGUAGE plsample;
```

兼容性

SQL 标准中没有`DROP LANGUAGE`语句。

另见

ALTER LANGUAGE, CREATE LANGUAGE, droplang

DROP OPERATOR

DROP OPERATOR — 移除一个操作符

大纲

```
DROP OPERATOR name ( lefttype | NONE , righttype | NONE ) [ CASCADE |
 RESTRICT ]
```

描述

DROP OPERATOR从数据库系统中删除一个现有操作符。要执行此命令，你必须是该操作符的拥有者。

参数

name

一个现有操作符的名称（可以是模式限定的）。

lefttype

操作符左操作数的数据类型；如果该操作符没有左操作数，请写 NONE。

righttype

操作符右操作数的数据类型；如果操作符没有右操作数，请写NONE。

CASCADE

自动删除依赖于该操作符的对象。

RESTRICT

如果有任何对象依赖于该操作符，则拒绝删除它。这是默认值。

示例

删除幂操作符a^b，其类型为 integer：

```
DROP OPERATOR ^ (integer, integer);
```

删除前缀一元按位取反操作符 ~b，其类型为 bit：

```
DROP OPERATOR ~ (none, bit);
```

移除 integer 类型的右一元阶乘操作符 x!：

```
DROP OPERATOR ! (integer, none);
```

兼容性

SQL 标准中没有DROP OPERATOR语句。

另请参阅

CREATE OPERATOR

DROP OPERATOR CLASS

DROP OPERATOR CLASS — 删除一个操作符类

大纲

```
DROP OPERATOR CLASS name USING index_method [ CASCADE | RESTRICT ]
```

描述

`DROP OPERATOR CLASS`删除一个现有操作符类。要执行此命令，你必须是该操作符类的拥有者。

参数

name

一个现有操作符类的名称（可以被模式限定）。

index_method

该操作符类所属索引访问方法的名称。

`CASCADE`

自动删除依赖于该操作符类的对象。

`RESTRICT`

如果有任何对象依赖于该操作符类，则拒绝删除它。这是默认值。

示例

移除 B-树操作符类`widget_ops`：

```
DROP OPERATOR CLASS widget_ops USING btree;
```

如果仍有索引使用该操作符类，此命令将不会成功。加上`CASCADE`可以在删除操作符类的同时删除这类索引。

兼容性

SQL 标准中没有`DROP OPERATOR CLASS`语句。

另见

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS

DROP RULE

DROP RULE — 删除一个重写规则

大纲

```
DROP RULE name ON relation [ CASCADE | RESTRICT ]
```

描述

`DROP RULE`删除一个重写规则。

参数

name

要删除的规则的名称。

relation

该规则所适用的表或视图的名称（可以有模式限定）。

`CASCADE`

自动删除依赖于该规则的对象。

`RESTRICT`

如果有任何对象依赖于该规则，则拒绝删除它。这是默认值。

示例

删除重写规则 `newrule`：

```
DROP RULE newrule ON mytable;
```

兼容性

SQL 标准中没有 `DROP RULE` 语句。

另见

CREATE RULE

DROP SCHEMA

DROP SCHEMA — 移除一个模式

大纲

```
DROP SCHEMA name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP SCHEMA` 从数据库中移除模式。

一个模式只能由其拥有者或超级用户删除。注意，即使拥有者并不拥有模式中的某些对象，他也可以删除该模式（以及其中包含的所有对象）。

参数

name

一个模式的名称。

`CASCADE`

自动删除模式中包含的对象（表、函数等）。

`RESTRICT`

如果模式包含任何对象，则拒绝删除它。这是默认行为。

示例

要从数据库中移除模式 `mystuff` 及其包含的所有内容：

```
DROP SCHEMA mystuff CASCADE;
```

兼容性

`DROP SCHEMA` 完全符合 SQL 标准，只是标准只允许每条命令删除一个模式。

另见

ALTER SCHEMA, CREATE SCHEMA

DROP SEQUENCE

DROP SEQUENCE — 移除一个序列

大纲

```
DROP SEQUENCE name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP SEQUENCE` 移除序列号生成器。

参数

name

序列的名称（可带模式限定）。

`CASCADE`

自动删除依赖于该序列的对象。

`RESTRICT`

如果有任何对象依赖于该序列，则拒绝删除它。这是默认行为。

示例

要移除序列 `serial`：

```
DROP SEQUENCE serial;
```

兼容性

SQL 标准中没有 `DROP SEQUENCE` 语句。

另见

CREATE SEQUENCE

DROP TABLE

DROP TABLE — 删除一个表

大纲

```
DROP TABLE name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP TABLE`从数据库中删除表。只有表的拥有者才能删除它。要清空一个表中的行而不删除表本身，可使用 `DELETE`。

`DROP TABLE`总会删除目标表上存在的所有索引、规则、触发器和约束。但是，要删除一个被另一个表的外键约束引用的表，必须指定 `CASCADE`。（`CASCADE`会删除该外键约束，而不是整个另一个表。）

参数

name

要删除的表的名称（可以有模式限定）。

`CASCADE`

自动删除依赖于该表的对象（例如视图）。

`RESTRICT`

如果有任何对象依赖于该表，则拒绝删除它。这是默认值。

示例

删除两个表 `films` 和 `distributors`：

```
DROP TABLE films, distributors;
```

兼容性

这条命令符合 SQL 标准。

另见

ALTER TABLE, CREATE TABLE

DROP TRIGGER

DROP TRIGGER — 移除一个触发器

大纲

```
DROP TRIGGER name ON table [ CASCADE | RESTRICT ]
```

描述

`DROP TRIGGER` 将移除一个现有的触发器定义。要执行这个命令，当前用户必须是定义该触发器的表的拥有者。

参数

name

要移除的触发器的名称。

table

定义该触发器的表的名称（可带模式限定）。

`CASCADE`

自动删除依赖于该触发器的对象。

`RESTRICT`

如果有任何对象依赖于该触发器，则拒绝删除它。这是默认行为。

示例

销毁表 `films` 上的触发器 `if_dist_exists`：

```
DROP TRIGGER if_dist_exists ON films;
```

兼容性

PostgreSQL 中的 `DROP TRIGGER` 语句与 SQL 标准不兼容。在 SQL 标准中，触发器名字不是表局部的，因此该命令就是 `DROP TRIGGER name`。

另见

CREATE TRIGGER

DROP TYPE

DROP TYPE — 移除一个数据类型

大纲

```
DROP TYPE name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP TYPE`移除一个用户定义的数据类型。只有该类型的拥有者才能移除它。

参数

name

要移除的数据类型的名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该类型的对象（例如表列、函数和操作符）。

`RESTRICT`

如果有任何对象依赖于该类型，则拒绝删除它。这是默认值。

示例

要移除数据类型`box`：

```
DROP TYPE box;
```

兼容性

此命令与 SQL 标准中的对应命令相似。但请注意，`CREATE TYPE`命令和 PostgreSQL中的数据类型扩展机制都不同于 SQL 标准。

另请参阅

CREATE TYPE

DROP USER

DROP USER — 删除一个数据库用户账户

大纲

```
DROP USER name
```

描述

`DROP USER`删除指定的用户。它不会删除该用户拥有的表、视图或其他对象。如果该用户拥有任何数据库，则会引发错误。

参数

name

要删除的用户的名称。

注解

PostgreSQL包含一个程序dropuser，
它与这条命令功能相同（实际上它调用的就是这条命令），但可以在命令 shell 中运行。

要删除一个拥有数据库的用户，请先删除该数据库或更改其所有权。

示例

删除一个用户账户：

```
DROP USER jonathan;
```

兼容性

`DROP USER`语句是 PostgreSQL的一种扩展。SQL 标准将用户的定义留给具体实现决定。

另见

ALTER USER, CREATE USER

DROP VIEW

DROP VIEW — 删除一个视图

大纲

```
DROP VIEW name [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP VIEW`删除一个现有视图。要执行这条命令，你必须是该视图的所有者。

参数

name

要删除的视图的名称（可以是模式限定的）。

`CASCADE`

自动删除依赖于该视图的对象（例如其他视图）。

`RESTRICT`

如果有任何对象依赖于该视图，则拒绝删除它。这是默认行为。

示例

这条命令将删除名为`kinds`的视图：

```
DROP VIEW kinds;
```

兼容性

这条命令符合 SQL 标准。

另见

CREATE VIEW

END

END — 提交当前事务

大纲

```
END [ WORK | TRANSACTION ]
```

描述

`END` 提交当前事务。该事务所做的所有更改对其他人可见，并且保证在发生崩溃时是持久的。这个命令是一种 PostgreSQL 扩展，等价于COMMIT。

参数

`WORK`
`TRANSACTION`

可选的关键字。它们没有效果。

注意

使用ROLLBACK中止一个事务。

在不处于事务中时发出 `END` 没有害处，但它会引发一条警告消息。

示例

要提交当前事务并使所有更改永久化：

```
END;
```

兼容性

`END` 是一种提供与COMMIT（它在 SQL 标准中指定）等价功能的 PostgreSQL 扩展。

另见

BEGIN, COMMIT, ROLLBACK

EXECUTE

EXECUTE — 执行一个预备语句

大纲

```
EXECUTE plan_name [ (parameter [, ...] ) ]
```

描述

EXECUTE用于执行一个之前预备好的语句。由于预备语句只在会话期间存在，
该预备语句必须是由当前会话中更早执行的PREPARE语句所创建的。

如果创建该语句的PREPARE语句指定了一些参数，那么必须向EXECUTE
语句传递一组兼容的参数，否则会引发错误。注意（与函数不同）
预备语句不会根据参数的类型或数量被重载；预备语句的名字在数据库会话中必须唯一。

有关预备语句的创建和用法的更多信息，请参阅PREPARE。

参数

plan_name

要执行的预备语句的名字。

parameter

预备语句某个参数的实际值。它必须是一个表达式，其产生的值的类型与创建该预备语句的PREPARE命令中为此参数位置指定的数据类型兼容。

兼容性

SQL 标准包含了EXECUTE语句，但它只用于嵌入式 SQL。这里的EXECUTE语句版本还使用了略有不同的语法。

EXPLAIN

EXPLAIN — 显示一条语句的执行计划

大纲

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

描述

这条命令显示PostgreSQL规划器为给定语句生成的执行计划。
执行计划显示语句引用的表将被如何扫描——用普通的顺序扫描、
索引扫描等——以及当引用多个表时，将使用什么连接算法把每个输入表中所需的行汇集到一起。

显示中最关键的部分是估计的语句执行代价，
即规划器对运行该语句所需时间的猜测（以磁盘页获取次数为单位度量）。
实际上会显示两个数字：返回第一行之前的启动时间，以及返回所有行的总时间。
对大多数查询来说，总时间才是重要的，但在诸如 EXISTS中的子查询这样的上下文中，规划器会选择最小的启动时间而不是最小的总时间（因为执行器在取到一行后无论如何都会停止）。
另外，如果你用LIMIT子句限制返回的行数，规划器会在端点代价之间做适当的插值，
来估计哪个计划实际上最便宜。

ANALYZE选项使语句被真正执行，而不仅仅是规划。每个计划节点内部花费的总耗时（毫秒）
以及它实际返回的总行数会被添加到显示中。这对于查看规划器的估计是否接近现实很有用。

> **重要**
>
> 请记住，使用ANALYZE时语句会被真正执行。虽然EXPLAIN会丢弃SELECT 本会返回的任何输出，但语句的其他副作用会照常发生。如果你想在INSERT、UPDATE、DELETE 或EXECUTE语句上使用 EXPLAIN ANALYZE而又不让命令影响你的数据，可以使用这样的方法：
>
> ```
> BEGIN;
> EXPLAIN ANALYZE ...;
> ROLLBACK;
> ```

参数

ANALYZE

执行命令并显示实际的运行时间。

VERBOSE

显示计划树的完整内部表示，而不仅仅是一个摘要。通常这个选项只对调试PostgreSQL有用。VERBOSE输出是否美化打印取决于 explain_pretty_print配置参数的设置。

statement

任何你希望看到其执行计划的SELECT、INSERT、UPDATE、DELETE、EXECUTE 或DECLARE语句。

注解

关于优化器在PostgreSQL中对代价信息的使用，目前只有很少的文档。更多信息请参阅第 13.1 节。

为了让PostgreSQL查询规划器在优化查询时能做出充分知情的决定，应当运行ANALYZE 语句来记录表中数据分布的统计信息。如果你没有这样做（或者自上次运行ANALYZE 以来表中数据的统计分布发生了显著变化），估计的代价不太可能符合查询的真实特性，因此可能选择一个较差的查询计划。

在PostgreSQL 7.3 之前，计划以 NOTICE消息的形式发出。现在它作为查询结果出现（格式化为一个只有单个文本列的表）。

示例

要显示对一个只有单个integer列和 10000 行的表的简单查询的计划：

```
EXPLAIN SELECT * FROM foo;

                       QUERY PLAN
-----------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

如果有一个索引，并且我们的查询带有可索引的WHERE 条件，EXPLAIN可能会显示一个不同的计划：

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                         QUERY PLAN
--------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

下面是一个使用聚合函数的查询的查询计划示例：

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                            QUERY PLAN
-------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

下面是使用EXPLAIN EXECUTE显示一个预备查询的执行计划的例子：

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);
```

```
                                                        QUERY PLAN

-----------------------------------------------------------------------
--------------------------------------------------
 HashAggregate  (cost=39.53..39.53 rows=1 width=8) (actual time=0.661.
.0.672 rows=7 loops=1)
   ->  Index Scan using test_pkey on test  (cost=0.00..32.97 rows=1311
 width=8) (actual time=0.050..0.395 rows=99 loops=1)
         Index Cond: ((id > $1) AND (id < $2))
 Total runtime: 0.851 ms
(4 rows)
```

当然，这里显示的具体数字取决于所涉及表的实际内容。还要注意，由于规划器的改进，这些数字、甚至被选中的查询策略都可能随 PostgreSQL版本而变化。此外，`ANALYZE`命令使用随机抽样来估计数据统计；因此，即使表中数据的实际分布没有变化，在重新运行一次 `ANALYZE` 之后，代价估计也有可能改变。

兼容性

SQL 标准中没有定义`EXPLAIN`语句。

FETCH

FETCH — 使用游标从查询中检索行

大纲

```
FETCH [ direction { FROM | IN } ] cursorname
```

其中 *direction* 可以为空，或为下列之一：

```
NEXT
PRIOR
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
BACKWARD
BACKWARD count
BACKWARD ALL
```

描述

FETCH使用先前创建的游标检索行。

游标带有一个关联位置，FETCH会使用该位置。游标位置可以位于查询结果的第一行之前、结果中的任意一行上，或者位于结果的最后一行之后。游标创建时位于第一行之前。提取若干行后，游标会定位在最近提取的那一行上。如果FETCH 越过了可用行的末尾，那么游标会留在最后一行之后；如果是向后提取，则会留在第一行之前。FETCH ALL或者 FETCH BACKWARD ALL总是会让游标位于最后一行之后或第一行之前。

NEXT、PRIOR、FIRST、LAST、ABSOLUTE、RELATIVE 这些形式会在适当地移动游标后提取一行。如果不存在这样的行，则返回空结果，并根据情况将游标定位在第一行之前或最后一行之后。

使用FORWARD和BACKWARD的形式会在向前或向后的方向上提取指定数量的行，并将游标定位在最后返回的那一行上（如果*count*超过可用行数，则定位在所有行之后或之前）。

RELATIVE 0、FORWARD 0以及 BACKWARD 0都会请求提取当前行而不移动游标，也就是重新提取最近一次提取的行。除非游标位于第一行之前或最后一行之后，否则该操作都会成功；在这两种情况下，不会返回任何行。

参数

direction

direction 定义抓取方向和要抓取的行数。它可以是以下值之一：

NEXT

提取下一行。如果省略*direction*，这将是默认值。

PRIOR

提取前一行。

FIRST

提取查询的第一行（与ABSOLUTE 1相同）。

LAST

提取查询的最后一行（与ABSOLUTE -1相同）。

ABSOLUTE *count*

提取查询的第*count*行；如果*count*为负，则提取从末尾算起的第abs(*count*)行。如果 *count*超出范围，则定位在第一行之前或最后一行之后。特别地，ABSOLUTE 0会定位在第一行之前。

RELATIVE *count*

抓取后面的第*count*行，或前面的第abs(*count*)行（当*count*为负时）。RELATIVE 0会重新抓取当前行（如果存在）。

count

提取接下来的*count*行（与FORWARD *count*相同）。

ALL

提取所有剩余行（与FORWARD ALL相同）。

FORWARD

提取下一行（与NEXT相同）。

FORWARD *count*

提取接下来的*count*行。FORWARD 0会重新提取当前行。

FORWARD ALL

提取所有剩余行。

BACKWARD

提取前一行（与PRIOR相同）。

BACKWARD *count*

提取前面的*count*行（反向扫描）。BACKWARD 0会重新提取当前行。

BACKWARD ALL

提取之前的所有行（反向扫描）。

count

*count*是一个可以带符号的整数常量，决定抓取位置或抓取行数。对于FORWARD和BACKWARD，指定负的*count*等同于交换FORWARD和BACKWARD的方向。

cursorname

一个已打开游标的名称。

输出

成功完成时，FETCH命令会返回如下形式的命令标签：

FETCH *count*

其中*count*是提取到的行数（可能为零）。注意，在psql中，命令标签实际上不会显示，因为psql会改为显示提取到的行。

注解

如果打算使用除FETCH NEXT或带正数计数的 FETCH FORWARD之外的任何FETCH 变体，则应当在声明游标时使用SCROLL选项。对于简单查询，PostgreSQL允许从未使用SCROLL 声明的游标向后提取，但最好不要依赖这种行为。如果游标声明为 NO SCROLL，则不允许向后提取。

ABSOLUTE提取并不比通过相对移动到达目标行更快：无论如何，
底层实现都必须遍历所有中间行。
负值的绝对提取甚至更糟：必须先把查询读到末尾以找到最后一行，然后再从那里反向遍历。
不过，回卷到查询起始处（如FETCH ABSOLUTE 0）是很快的。

目前PostgreSQL不支持通过游标更新数据。

DECLARE用于定义游标。使用MOVE可以在不检索数据的情况下改变游标位置。

示例

下面的示例展示了如何使用游标遍历一个表：

```
BEGIN WORK;

-- 建立一个游标:
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- 从游标 liahona 中提取前 5 行:
FETCH FORWARD 5 FROM liahona;

 code  |          title          | did | date_prod  |   kind   |  len
-------+-------------------------+-----+------------+----------+-----
--
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28
```

```
-- 提取前一行:
FETCH PRIOR FROM liahona;

 code  |  title  | did | date_prod  |  kind  |  len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- 关闭游标并结束事务:
CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL 标准只为嵌入式 SQL 定义了FETCH。这里描述的FETCH变体会像 SELECT结果那样返回数据，而不是把数据放入主变量中。除此之外，FETCH与 SQL 标准完全向上兼容。

涉及FORWARD和BACKWARD的 FETCH形式，以及形式FETCH *count*和FETCH ALL（其中隐含了FORWARD）都是 PostgreSQL扩展。

SQL 标准只允许在游标名之前使用FROM；可以使用 IN这一选项是扩展。

GRANT

GRANT — 定义访问权限

大纲

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES |
 TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tablename [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]

GRANT { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE dbname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION funcname ([type, ...]) [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE langname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]

GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schemaname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]
```

描述

GRANT命令把对象（表、视图、序列、数据库、函数、过程语言或模式）
上的特定权限授予一个或多个用户或用户组。这些权限会叠加在已有权限（如果有）之上。

关键字PUBLIC表示权限将授予所有用户，包括以后可能创建的用户。可以把PUBLIC
看作一个隐式定义的、始终包含所有用户的组。任何特定用户所拥有的权限，
是他直接被授予的权限、他被授予的当前所属各组的权限，以及授予PUBLIC的权限之和。

如果指定了WITH GRANT OPTION，权限的接受者随后又可以把它授予其他人。默认情况下不允许这样做。授予选项只能授予单个用户，不能授予组或PUBLIC。

无需把权限授予对象的拥有者（通常是创建它的用户），因为拥有者默认拥有所有权限。
（不过，拥有者也可以出于安全考虑选择撤销他自己的某些权限。）
删除对象或以任何方式更改其定义的权利不属于可授予的权限；它是拥有者固有的，
不能被授予或撤销。拥有者的授予选项也不可能被撤销。

取决于对象的类型，初始的默认权限可能包括把某些权限授予PUBLIC。
默认情况下：表和模式没有公共访问权限；数据库有创建TEMP（临时）表的权限；

函数有EXECUTE权限；语言有USAGE权限。对象的拥有者当然可以撤销这些权限。
（为了最大程度的安全，应在创建对象的同一事务中发出REVOKE；
这样就不存在其他用户可以使用该对象的窗口期。）

可能的权限有：

SELECT

允许对指定表、视图或序列的任意列进行SELECT。还允许使用COPY TO。
在UPDATE或DELETE中引用现有列的值也需要此权限。对于序列，此权限还允许使用
currval函数。

INSERT

允许向指定表中INSERT新行。还允许COPY FROM。

UPDATE

允许对指定表的任意列进行UPDATE。（实践中，任何非平凡的UPDATE命令还需要SELECT
权限，因为它必须引用表的列来确定要更新哪些行，以及/或者为列计算新值。）除SELECT
权限之外，SELECT ... FOR UPDATE还需要此权限。对于序列，此权限允许使用nextval
和setval函数。

DELETE

允许从指定表中DELETE行。（实践中，任何非平凡的DELETE命令还需要SELECT权限，
因为它必须引用表的列来确定要删除哪些行。）

RULE

允许在表/视图上创建规则。（见CREATE RULE语句。）

REFERENCES

要创建外键约束，必须在引用表和被引用表上都拥有此权限。

TRIGGER

允许在指定表上创建触发器。（见CREATE TRIGGER语句。）

CREATE

对于数据库，允许在其中创建新模式。

对于模式，允许在其中创建新对象。要重命名一个现有对象，你必须拥有该对象
并且对包含它的模式拥有此权限。

TEMPORARY
TEMP

允许在使用指定数据库时创建临时表。

EXECUTE

允许使用指定的函数以及使用基于该函数实现的任何操作符。
这是唯一适用于函数的权限类型。（此语法也适用于聚集函数。）

USAGE

对于过程语言，允许使用指定的语言来创建以该语言编写的函数。
这是唯一适用于过程语言的权限类型。

对于模式，允许访问包含在指定模式中的对象（前提是也满足对象自身的权限要求）。
本质上，这允许被授权者在该模式中“查找”对象。

ALL PRIVILEGES

一次性授予适用于该对象的所有权限。在PostgreSQL中`PRIVILEGES`关键字是可选的，
但严格的 SQL 要求它。

其他命令所需的权限列在各自命令的参考页上。

注意

REVOKE命令用于撤销访问权限。

应当注意，数据库超级用户可以访问所有对象，而不受对象权限设置的限制。这与 Unix 系统中`root`的权利类似。与`root`一样，除非绝对必要，以超级用户身份操作是不明智的。

如果超级用户选择发出`GRANT`或`REVOKE`命令，
该命令的执行效果就如同它是受影响对象的拥有者发出的一样。特别地，
通过这种命令授予的权限将表现为由对象拥有者所授予。

目前，在PostgreSQL中要只对少数几个列授予权限，必须创建一个包含所需列的视图，
然后把权限授予该视图。

使用psql的`\z`命令获取现有权限的信息，例如：

```
=> \z mytable

        Access privileges for database "lusitania"
 Schema |  Table  |           Access privileges
--------+---------+-------------------------------------
 public | mytable | {=r/postgres,miriam=arwdRxt/postgres,"group todos=
arw/postgres"}
(1 row)
```

`\z`显示的各项的含义如下：

```
              =xxxx -- privileges granted to PUBLIC
         uname=xxxx -- privileges granted to a user
   group gname=xxxx -- privileges granted to a group

                  r -- SELECT ("read")
                  w -- UPDATE ("write")
                  a -- INSERT ("append")
                  d -- DELETE
                  R -- RULE
                  x -- REFERENCES
                  t -- TRIGGER
                  X -- EXECUTE
                  U -- USAGE
```

```
                C -- CREATE
                T -- TEMPORARY
          arwdRxt -- ALL PRIVILEGES (for tables)
                * -- grant option for preceding privilege

            /yyyy -- user who granted this privilege
```

上面的示例输出是用户miriam在创建表mytable并执行

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO GROUP todos;
```

之后将会看到的。

如果某个对象的“访问权限”列为空，表示该对象拥有默认权限（即其权限列为 null）。默认权限总是包含拥有者的所有权限，并且根据对象类型，可能像上面解释的那样包含授予PUBLIC的某些权限。对对象的第一次GRANT或REVOKE会把默认权限实例化（例如产生{=,miriam=arwdRxt}），然后按指定的请求修改它们。

示例

把表films上的插入权限授予所有用户：

```
GRANT INSERT ON films TO PUBLIC;
```

把视图kinds上的所有权限授予用户manuel：

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

兼容性

按照 SQL 标准，ALL PRIVILEGES中的PRIVILEGES关键字是必需的。SQL 标准不支持在一条命令中为多个对象设置权限。

SQL 标准允许为表中的单个列设置权限：

```
GRANT privileges
    ON table [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] } [ WITH GRANT OPTION ]
```

SQL 标准为其他类型的对象提供了USAGE权限：字符集、排序规则、转换和域。

RULE权限，以及在数据库、模式、语言和序列上的权限是PostgreSQL扩展。

另见

REVOKE

INSERT

INSERT — 在一个表中创建新行

大纲

```
INSERT INTO table [ ( column [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) |
 query }
```

描述

INSERT允许我们向一个表中插入新行。可以一次插入一行，也可以插入一个查询结果的多个行。

目标列表中的列可以按任意顺序列出。目标列表中没有出现的每个列将使用默认值插入，即其声明的默认值或空值。

如果每个列的表达式数据类型不正确，将尝试自动类型转换。

要向一个表插入数据，你必须拥有该表上的INSERT权限。如果使用 *query*子句插入来自查询的行，你还需要拥有查询中使用的任何表上的SELECT权限。

参数

table

一个已存在表的名称（可以有模式限定）。

column

*table*中一个列的名称。

DEFAULT VALUES

所有列都将用它们的默认值填充。

expression

要赋予*column*的一个表达式或值。

DEFAULT

该列将用它的默认值填充。

query

提供要插入行的查询（SELECT语句）。其语法的描述请参考 SELECT语句。

输出

成功完成时，INSERT命令返回如下形式的命令标签

INSERT *oid count*

其中*count*是插入的行数。如果 *count*恰好为一，并且目标表有 OID，那么*oid*就是分配给被插入行的 OID。否则*oid*为零。

示例

向表 `films` 插入一行：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

在第二个例子中，省略了最后一列 `len`，因此它将使用默认值空值：

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

第三个例子对日期列使用 `DEFAULT` 子句而不是指定一个值：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

这个例子从表 `tmp` 向表 `films` 插入若干行：

```
INSERT INTO films SELECT * FROM tmp;
```

这个例子向数组列插入数据：

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
-- (all of these commands create the same board)
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1,'{{"","",""},{},{"",""}}');
INSERT INTO tictactoe (game, board[3][3])
    VALUES (2,'{}');
INSERT INTO tictactoe (game, board)
    VALUES (3,'{{,,},{,,},{,,}}');
```

兼容性

`INSERT`完全符合 SQL 标准。*query*子句可能存在的限制记录在SELECT中。

LISTEN

LISTEN — 监听一个通知

大纲

```
LISTEN name
```

描述

LISTEN把当前会话注册为通知条件*name*的监听者。
如果当前会话已经被注册为该通知条件的监听者，则不做任何事。

每当命令NOTIFY *name*被调用时——无论是被本会话还是被连接到同一数据库的其他会话调用——所有当前正在监听该通知条件的会话都会被通知，
而每个会话又将进而通知它所连接的客户端应用。更多信息请参阅 NOTIFY的讨论。

可以用UNLISTEN命令取消一个会话对给定通知条件的注册。
会话的监听注册在该会话结束时被自动清除。

客户端应用检测通知事件所必须使用的方法取决于它使用的 PostgreSQL应用编程接口。使用libpq库时，应用像发出普通 SQL 命令一样发出 LISTEN，然后必须周期性地调用函数 PQnotifies来查明是否收到了任何通知事件。其他接口（例如libpgtcl）提供了处理通知事件的更高层方法；实际上，使用libpgtcl时，应用程序员甚至不必直接发出LISTEN或 UNLISTEN。更多细节请参阅你所使用的接口的文档。

NOTIFY中包含关于LISTEN和 NOTIFY用法的更广泛的讨论。

参数

name

通知条件的名称（任何标识符）。

示例

在psql中配置并执行一个 listen/notify 序列：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with
 PID 8448.
```

兼容性

SQL 标准中没有LISTEN语句。

另见

NOTIFY, UNLISTEN

LOAD

LOAD — 装载或重新装载一个共享库文件

大纲

```
LOAD 'filename'
```

描述

这条命令把一个共享库文件装载到PostgreSQL 服务器的地址空间中。如果该文件先前已被装载，会先卸载它。
这条命令主要用于卸载并重新装载一个自服务器首次装载它以来已被更改的共享库文件。
要使用该共享库，需要使用CREATE FUNCTION命令声明其中的函数。

文件名的指定方式与CREATE FUNCTION中共享库名的指定方式相同；特别地，可以依赖搜索路径以及系统标准共享库文件扩展名的自动添加。关于这个主题的更多信息，见第 33.3 节。

兼容性

`LOAD`是PostgreSQL 的一个扩展。

另见

CREATE FUNCTION

LOCK

LOCK — 锁定一个表

大纲

```
LOCK [ TABLE ] name [, ...] [ IN lockmode MODE ]

where lockmode is one of:

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

描述

`LOCK TABLE` 获取一个表级锁，必要时会等待任何冲突的锁被释放。一旦取得，该锁会保持到当前事务结束。（没有 `UNLOCK TABLE` 命令；锁总是在事务结束时释放。）

在为引用表的命令自动获取锁时，PostgreSQL 总是使用限制性最低的锁模式。`LOCK TABLE` 用于你可能需要更严格锁定的场合。例如，假设一个应用以读已提交隔离级别运行事务，并且需要确保表中的数据在事务期间保持稳定。为此，你可以在查询之前对该表取得 `SHARE` 锁模式。这将阻止并发数据更改，并确保随后对该表的读取看到已提交数据的稳定视图，因为 `SHARE` 锁模式与写者取得的 `ROW EXCLUSIVE` 锁冲突，而你的 `LOCK TABLE name IN SHARE MODE` 语句会等待任何并发持有 `ROW EXCLUSIVE` 模式锁的事务提交或回滚。因此，一旦你取得了锁，就不存在未提交的写；而且在释放锁之前也不会有新的写开始。

要在以可串行化隔离级别运行事务时达到类似效果，你必须在执行任何数据修改语句之前执行 `LOCK TABLE` 语句。可串行化事务的数据视图会在其第一条数据修改语句开始时冻结。之后再执行 `LOCK TABLE` 仍能阻止并发写——但它无法确保事务读取到的内容对应于最新提交的值。

如果这类事务要更改表中的数据，就应使用 `SHARE ROW EXCLUSIVE` 锁模式而不是 `SHARE` 模式。这确保同一时刻只有一个这类事务在运行。否则可能出现死锁：两个事务可能都取得了 `SHARE` 模式，然后又都无法再取得 `ROW EXCLUSIVE` 模式来实际执行更新。（注意，事务自身的锁永不冲突，因此持有 `SHARE` 模式的事务可以再取得 `ROW EXCLUSIVE` 模式——但如果别人持有 `SHARE` 模式则不行。）为避免死锁，应确保所有事务以相同的顺序对相同的对象获取锁，而且如果单个对象涉及多种锁模式，事务应总是先获取限制性最高的模式。

关于锁模式和锁策略的更多信息，请参见第 12.3 节。

参数

name

要锁定的现有表的名称（可选模式限定）。

命令 `LOCK a, b;` 等价于 `LOCK a; LOCK b;`。各表按 `LOCK` 命令中指定的顺序逐一锁定。

lockmode

锁模式指定该锁会与哪些锁冲突。锁模式见第 12.3 节。

如果未指定锁模式，则使用限制最严格的`ACCESS EXCLUSIVE`模式。

注意

LOCK ... IN ACCESS SHARE MODE 要求在目标表上具有 SELECT 权限。LOCK 的所有其他形式都要求 UPDATE 和/或 DELETE 权限。

LOCK 只在事务块（BEGIN/COMMIT 对）内部有用，因为锁在事务一结束就被丢弃。出现在任何事务块之外的 LOCK 命令构成一个自包含的事务，因此锁一取得就会被丢弃。

LOCK TABLE 只处理表级锁，因此涉及 ROW 的模式名都是名不副实的。这些模式名一般应理解为表明用户意图在被锁表内获取行级锁。此外，ROW EXCLUSIVE 模式是一种可共享的表锁。请记住，就 LOCK TABLE 而言，所有锁模式的语义都相同，只在哪些模式与哪些模式冲突的规则上有差别。

示例

在准备向外键表执行插入时，在主键表上获取一个SHARE锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- 如果未返回记录则执行 ROLLBACK
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

在准备执行删除操作时，在主键表上获取一个SHARE ROW EXCLUSIVE锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

兼容性

SQL 标准中没有LOCK TABLE，而是使用SET TRANSACTION 来指定事务的并发级别。PostgreSQL也支持这一点；详见SET TRANSACTION。

除ACCESS SHARE、ACCESS EXCLUSIVE和 SHARE UPDATE EXCLUSIVE锁模式外，PostgreSQL的锁模式和LOCK TABLE语法与Oracle中的对应语法兼容。

MOVE

MOVE — 定位游标

大纲

```
MOVE [ direction { FROM | IN } ] cursorname
```

描述

MOVE在不检索任何数据的情况下重新定位游标。MOVE与FETCH 命令的工作方式完全相同，只是它仅定位游标而不返回行。

有关语法和用法的细节，请参阅FETCH。

输出

成功完成时，MOVE命令会返回如下形式的命令标签：

```
MOVE count
```

其中*count*是被越过的行数（可能为零）。

示例

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- 跳过前 5 行:
MOVE FORWARD 5 IN liahona;
MOVE 5

-- 从游标 liahona 中提取第 6 行:
FETCH 1 FROM liahona;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- 关闭游标 liahona 并结束事务:
CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL 标准中没有MOVE语句。

NOTIFY

NOTIFY — 生成一个通知

大纲

NOTIFY *name*

描述

NOTIFY 命令向当前数据库中曾为指定通知名执行过 LISTEN *name* 的每个客户端应用发送一个通知事件。

传递给客户端的通知事件信息包括：通知名以及发出通知的会话对应的服务器进程PID。在某个数据库中使用哪些通知名以及各自含义，由数据库设计者自行决定。

常见的做法是让通知名与数据库中的某个表同名，而通知事件基本上意味着："我改动了这张表，去看看有什么新变化"。不过，NOTIFY和LISTEN命令并不会强制这种关联。例如，数据库设计者可以使用多个不同的通道名，来标识同一张表上的不同类型变更；

NOTIFY为访问同一个PostgreSQL数据库的一组进程提供了一种简单的进程间通信机制。如果需要传递结构化数据，也可以借助数据库中的表，把附加数据（而不只是通知名）从通知发送者传递给一个或多个监听者，从而构建更高层的机制。

当NOTIFY用于表明某张特定表发生变化时，一种很有用的编程技巧是把NOTIFY放进由表更新触发的规则中。这样一来，每当表发生变化时就会自动发出通知，应用程序员也不容易忘记这么做。

NOTIFY 在若干重要方面与 SQL 事务交互。首先，如果在事务内部执行了 NOTIFY，通知事件直到且仅当事务提交时才会被送达。这是恰当的，因为如果事务中止，其中的所有命令都不会产生效果，包括 NOTIFY。但如果期望通知事件立即送达，这可能会让人不安。其次，如果一个正在监听的会话在处于事务中时收到一个通知信号，该通知事件要到事务刚完成（提交或中止）之后才会送达其连接的客户端。同样，其理由是：如果一个通知在某个后来被中止的事务内被送达，人们会希望以某种方式撤销该通知——但服务器一旦把通知发给了客户端，就无法"收回"它。因此通知事件只在事务之间送达。由此得出的结论是：使用 NOTIFY 进行实时信号传递的应用应尽量保持事务简短。

NOTIFY在一个重要方面的行为类似 Unix 信号：如果在短时间内多次发出同一通知名的信号，接收者可能对多次执行的NOTIFY只收到一个通知事件。因此，依赖收到通知的数量并不是一个好主意。正确的做法是用NOTIFY唤醒需要关注某事的应用，而用一个数据库对象（例如序列）来跟踪发生了什么以及发生了多少次。

执行NOTIFY的客户端自己同时也在监听同一通知条件，这是很常见的情形。在这种情况下，它会像其他监听会话一样收到一个通知事件。根据应用逻辑，这可能导致无用功，例如再次读取自己刚刚写入更新的数据库表。要避免这种额外工作，可以检查通知事件消息中提供的发出通知的服务器进程PID，是否与当前会话自身的PID（可通过libpq获得）相同。如果二者相同，就说明这是当前会话自己发出的通知回送给自己，可以直接忽略。（尽管前一段有那样的说法，但这是一种安全的技术：PostgreSQL会把自身通知与来自其他会话的通知分开保存，因此忽略自己的通知并不会让你错过来自外部的通知。）

参数

name

要发信号的通知名称（任意标识符）。

示例

在psql中配置并执行一组 listen/notify 操作：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with
 PID 8448.
```

兼容性

SQL 标准中没有`NOTIFY`语句。

另请参阅

LISTEN, UNLISTEN

PREPARE

PREPARE — 为执行准备一个语句

大纲

```
PREPARE plan_name [ (datatype [, ...] ) ] AS statement
```

描述

PREPARE 创建一个预备语句。预备语句是一种可用于优化性能的服务器端对象。执行 PREPARE 语句时，指定的语句会被解析、重写和规划。随后对预备语句发出 EXECUTE 命令时，只需执行该预备语句即可。这样，解析、重写和规划阶段只执行一次，而不是每次执行语句时都进行。

预备语句可以带参数，也就是在执行时会代入语句中的值。要在预备语句中使用参数，需要在PREPARE语句中提供一个数据类型列表，并在被预备的语句本身中，使用$1、$2 等按位置引用这些参数。执行该语句时，需要在 EXECUTE语句中为这些参数指定实际值。有关详情，参见EXECUTE。

预备语句只在当前数据库会话期间存储并存在。当会话结束时，预备语句就被遗忘，
因此再次使用之前必须重新创建它。
这也意味着单条预备语句不能被多个同时的数据库客户端使用；不过，
每个客户端都可以创建自己的预备语句来使用。

当单个会话要执行大量相似语句时，预备语句能带来最大的性能优势。
如果语句在规划或重写时比较复杂，例如查询涉及许多表的连接，或者需要应用多条规则，
那么这种性能差异会特别明显。如果语句相对容易规划和重写，但执行本身代价较高，
则预备语句带来的性能优势就不那么明显。

参数

plan_name

为这个特定预备语句指定的任意名称。它在单个会话内必须唯一，
随后用于执行或释放先前已准备的语句。

datatype

预备语句中某个参数的数据类型。要在预备语句本身中引用参数，可使用 $1、$2 等。

statement

任何SELECT、INSERT、UPDATE 或DELETE语句。

注解

在某些情况下，为预备语句生成的查询计划可能劣于正常提交并执行该语句时生成的计划。
这是因为在对语句做规划且规划器试图确定最优查询计划时，
语句中指定的任何参数的实际值都不可用。PostgreSQL 收集表中数据分布的统计信息，并可以利用语句中的常量值来猜测执行该语句的可能结果。
由于带参数的预备语句在规划时这些数据不可用，所选的计划可能是次优的。要查看
PostgreSQL 为预备语句选择的查询计划，可以使用 EXPLAIN EXECUTE。

有关查询规划以及 PostgreSQL 为此收集的统计信息的更多内容，请参见ANALYZE文档。

兼容性

SQL 标准包含`PREPARE`语句，但它只用于嵌入式 SQL。这里的`PREPARE`语句在语法上也略有不同。

REINDEX

REINDEX — 重建索引

大纲

```
REINDEX { DATABASE | TABLE | INDEX } name [ FORCE ]
```

描述

REINDEX根据表中存储的数据重建一个索引，替换索引的旧副本。使用REINDEX的主要原因有两个：

- 一个索引已经损坏，不再包含有效数据。尽管理论上这不应该发生，但在实践中，索引可能因软件缺陷或硬件故障而损坏。REINDEX提供了一种恢复方法。
- 该索引包含大量未被回收的死索引页。在 PostgreSQL中，B-tree 索引在某些访问模式下可能出现这种情况。REINDEX提供了一种通过写入不含死页的新版本索引来减少索引空间占用的方法。更多信息见第 21.2 节。

参数

DATABASE

重建指定数据库的所有系统索引。用户表上的索引不被处理。此外，共享系统目录上的索引也会被跳过（独立模式除外，见下文）。

TABLE

重建指定表的所有索引。如果该表有次级 “TOAST” 表，该表也会被重建索引。

INDEX

重建一个指定的索引。

name

要重建索引的特定数据库、表或索引的名称。表名和索引名可以是模式限定的。

FORCE

这是一个已废弃的选项；如果指定它将被忽略。

注解

如果你怀疑用户表上的某个索引损坏了，可以简单地用 REINDEX INDEX或REINDEX TABLE重建该索引或该表的所有索引。处理损坏的用户表索引的另一种方法就是删除并重新创建它。如果你希望在此期间维持表面上正常的表操作，这实际上可能更可取。REINDEX获取表的排他锁，而CREATE INDEX只锁定表的写而不锁定读。

如果你需要从系统表索引的损坏中恢复，事情就更困难了。在这种情况下，重要的是系统自身没有使用过任何可疑的索引。（事实上，

在这种场景下你可能会发现服务器进程在启动时立即崩溃，因为依赖了损坏的索引。）
要安全地恢复，服务器必须以 -P选项启动，该选项阻止它在系统目录查找中使用索引。

做到这一点的一种方法是关闭 postmaster，然后启动一个独立的 PostgreSQL服务器，并在其命令行中包含-P选项。然后可以根据你想重建多少，发出REINDEX DATABASE、REINDEX TABLE或REINDEX INDEX。如果不确定，使用REINDEX DATABASE来选择重建该数据库的所有系统索引。然后退出独立服务器会话并重启常规服务器。关于如何与独立服务器界面交互的更多信息，见postgres参考页。

或者，可以在常规服务器会话的命令行选项中包含 -P来启动它。具体的做法因客户端而异，但在所有基于libpq的客户端中，都可以在启动客户端之前把PGOPTIONS环境变量设置为-P。注意，虽然这种方法不需要锁定其他客户端，但在修复完成之前，
阻止其他用户连接到受损的数据库可能仍是明智的。

如果怀疑任何共享系统目录（pg_database、pg_group或 pg_shadow）的索引损坏，则必须使用独立服务器来修复。REINDEX在多用户模式下不处理共享目录。

对于除共享系统目录之外的所有索引，REINDEX是崩溃安全且事务安全的。REINDEX
对共享索引不是崩溃安全的，这就是正常操作期间禁止这种情况的原因。
如果在独立模式下重建这些目录之一时发生故障，在问题得到纠正之前将无法重启常规服务器。
（部分重建的共享索引的典型症状是“index is not a btree”错误。）

在PostgreSQL 7.4 之前，REINDEX TABLE不会自动处理 TOAST 表，因此必须用单独的命令重建这些表的索引。现在仍然可以这样做，但已是多余的。

示例

重建表my_table上的索引：

```
REINDEX TABLE my_table;
```

重建单个索引：

```
REINDEX INDEX my_index;
```

重建特定数据库中的所有系统索引，不信任它们已经是有效的：

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

兼容性

SQL 标准中没有REINDEX命令。

RESET

RESET — 将运行时参数的值恢复为默认值

大纲

```
RESET name
RESET ALL
```

描述

RESET将运行时参数恢复为其默认值。RESET是

```
SET parameter TO DEFAULT
```

的另一种拼写。细节请参阅SET。

默认值指的是：在当前会话中，如果从未对该参数执行过SET，它本应具有的值。
该值的实际来源可能是编译时内置的默认值、配置文件、命令行选项，
或者针对特定数据库或特定用户的默认设置。细节请参阅第 16.4 节。

有关RESET的事务行为，请参阅SET参考页。

参数

name

运行时参数的名称。列表请参阅SET。

ALL

把所有可设置的运行时参数重置为默认值。

示例

把DATESTYLE设为默认值：

```
RESET datestyle;
```

把GEQO设为默认值：

```
RESET geqo;
```

兼容性

RESET是PostgreSQL扩展。

REVOKE

REVOKE — 移除访问权限

大纲

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES |
 TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tablename [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE dbname [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION funcname ([type, ...]) [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE langname [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schemaname [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

描述

`REVOKE` 命令从一个或多个用户或用户组回收先前授予的权限。关键字 `PUBLIC` 指隐式定义的组，即所有用户。

权限类型的含义见GRANT命令的描述。

注意，任何特定用户所拥有的权限是下列权限的总和：直接授予他的权限、
授予他当前所属的任何组的权限，以及授予 `PUBLIC` 的权限。因此，举例来说，从 `PUBLIC` 回收 `SELECT` 权限并不一定意味着所有用户都失去了对该对象的 `SELECT` 权限：直接拥有它或通过组拥有它的那些用户仍将拥有它。

如果指定了 `GRANT OPTION FOR`，只回收权限的转授选项，而不回收权限本身。

如果某个用户持有一个带转授选项的权限，并且已把它授予其他用户，
那么那些其他用户持有的权限称为依赖权限。
如果第一个用户持有的权限或转授选项正在被回收且存在依赖权限，那么在指定了 `CASCADE` 时那些依赖权限也会被回收，否则回收动作将失败。这种递归回收只影响通过一条可追溯到这条 `REVOKE` 命令主体的用户的用户链授予的权限。因此，如果受影响的用户还通过其他用户获得了该权限，他们实际上仍可能保有它。

注意

使用psql的 `\z` 命令显示现有对象上被授予的权限。权限格式的信息另见GRANT。

用户只能回收由该用户直接授予的权限。举例来说，如果用户 A 把一个带转授选项的权限授予了用户 B，而用户 B 又把它授予了用户 C，那么用户 A 无法直接从 C 回收该权限。取而代之的是，用户 A 可以从用户 B 回收转授选项并使用 `CASCADE` 选项，使该权限自动地从用户 C 回收。

如果超级用户选择发出 `GRANT` 或 `REVOKE` 命令，该命令的执行就像它是受影响对象的拥有者发出的一样。由于所有权限最终都来自对象拥有者（可能间接地通过转授选项链），
超级用户可以回收所有权限，但这可能需要像上面说的那样使用 `CASCADE`。

示例

回收 public 在表 `films` 上的插入权限：

```
REVOKE INSERT ON films FROM PUBLIC;
```

回收用户 `manuel` 在视图 `kinds` 上的所有权限：

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

兼容性

GRANT命令的兼容性说明同样适用于 `REVOKE`。语法摘要是：

```
REVOKE [ GRANT OPTION FOR ] privileges
    ON object [ ( column [, ...] ) ]
    FROM { PUBLIC | username [, ...] }
    { RESTRICT | CASCADE }
```

按照标准，`RESTRICT` 或 `CASCADE` 二者必须指定其一，但 PostgreSQL 默认采用 `RESTRICT`。

另见

GRANT

ROLLBACK

ROLLBACK — 中止当前事务

大纲

```
ROLLBACK [ WORK | TRANSACTION ]
```

描述

`ROLLBACK` 回滚当前事务并使该事务所做的所有更新被丢弃。

参数

`WORK`
`TRANSACTION`

可选的关键字。它们没有效果。

注意

使用COMMIT成功地终止一个事务。

在不处于事务中时发出 `ROLLBACK` 没有害处，但它会引发一条警告消息。

示例

要中止所有更改：

```
ROLLBACK;
```

兼容性

SQL 标准只指定了 `ROLLBACK` 和 `ROLLBACK WORK` 两种形式。除此之外，这个命令完全符合标准。

另见

BEGIN, COMMIT

SELECT

SELECT — 从表或视图中检索行

大纲

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
    [ FOR UPDATE [ OF table_name [, ...] ] ]

where from_item can be one of:

    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...]
 ) ] ]
    ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_
alias [, ...] | column_definition [, ...] ) ]
    function_name ( [ argument [, ...] ] ) AS ( column_definition [, .
..] )
    from_item [ NATURAL ] join_type from_item [ ON join_condition |
 USING ( join_column [, ...] ) ]
```

描述

SELECT 从一个或多个表中检索行。SELECT 的一般处理过程如下：

1. 所有FROM列表中的元素都会被计算。（FROM列表中的每个元素都是一个真实或虚拟表。）如果在FROM列表中指定了多个元素，则它们会被交叉连接在一起。（参见下面的FROM 子句。）

2. 如果指定了WHERE子句，则不满足条件的所有行将从输出中删除。（请参见下面的WHERE Clause。）

3. 如果指定了GROUP BY子句，或者存在聚合函数调用，输出将被组合成在一个或多个值上匹配的行组，并计算聚合函数的结果。如果存在HAVING子句，它将消除不满足给定条件的组。（参见GROUP BY Clause和HAVING Clause。）

4. 使用操作符UNION、INTERSECT和EXCEPT，可以把多条SELECT语句的输出合并成一个结果集。UNION操作符返回在一个或两个结果集中的所有行。INTERSECT操作符返回严格同时在两个结果集中的所有行。EXCEPT操作符返回在第一个结果集中但不在第二个结果集中的行。在这三种情况下，除非指定了ALL，否则重复行都会被消除。（见下面的UNION Clause、INTERSECT Clause和EXCEPT Clause。）

5. 实际输出行是使用每个选定行或行组的SELECT输出表达式计算的。（参见下面的SELECT 列表。）

6. 如果指定了ORDER BY子句，则返回的行按指定顺序排序。如果没有给出ORDER BY，则按系统认为最快的顺序返回行。（参见下面的ORDER BY 子句。）

7. SELECT DISTINCT消除结果中的重复行。SELECT DISTINCT ON会消除在所有指定表达式上匹配的行，只保留每组中的第一行。SELECT ALL（默认）将返回所有候选行，包括重复行。（参见下面的DISTINCT 子句。）

8. 如果指定了LIMIT或OFFSET子句，SELECT语句只返回结果行的子集。（参见下面的LIMIT 子句。）

9. FOR UPDATE子句使SELECT语句把选定的行锁定，防止并发更新。（见下面的FOR UPDATE 子句。）

你必须在一个表上拥有SELECT权限才能读取它的值。使用FOR UPDATE还要求UPDATE权限。

参数

FROM 子句

FROM 子句为 SELECT 指定一个或更多源表。如果指定了多个源表，结果将是所有源表的笛卡尔积（交叉连接）。但通常会增加限定条件，把返回的行限制为该笛卡尔积的一个小子集。

FROM 子句的元素可以是：

table_name

一个现有表或视图的名称（可以选择带模式限定）。如果指定了 ONLY，则只扫描该表。如果未指定 ONLY，则扫描该表及其所有后代表（如果有）。* 可以附加到表名后面以指示要扫描后代表，但在当前版本中这是默认行为。（在 7.1 之前的版本中，ONLY 是默认行为。）默认行为可以通过更改 sql_inheritance 配置选项来修改。

alias

包含该别名的FROM项的替代名称。别名可用于简写，
或者消除自连接（同一张表被扫描多次）中的歧义。提供别名后，
它会完全隐藏表或函数的实际名称；例如给定FROM foo AS f，SELECT的其余部分必须把这个FROM 项写成f而不是foo。如果写了别名，还可以写列别名列表，为该表的一个或多个列提供替代名称。

select

子SELECT可以出现在FROM子句中，它的作用就像在这个SELECT
命令的执行期间创建了一个临时表。注意，子SELECT必须用圆括号括起来，并且
必须为其提供别名。

function_name

函数调用可以出现在FROM子句中。（这对于返回结果集的函数尤其有用，
但任何函数都可以使用。）其效果就像在这条SELECT命令执行期间，
将函数的输出创建成了一张临时表。也可以使用别名。如果写了别名，
还可以写一个列别名列表，为函数复合返回类型中的一个或多个属性提供替代名称。

如果函数被定义为返回record数据类型，则必须给出别名或关键字AS，后面跟一个如下形式的列定义列表：(*column_name data_type* [, ...])。列定义列表必须与函数实际返回的列数和类型相匹配。

join_type

以下之一：

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

对于INNER和OUTER连接类型，必须指定一个连接条件，即NATURAL、ON *join_condition*或USING (*join_column* [, ...])三者中恰好之一。其含义见下文。对于CROSS JOIN，这些子句都不能出现。

JOIN 子句组合两个 FROM 项。如有必要可用圆括号确定嵌套的顺序。在没有圆括号时，JOIN 从左到右嵌套。在任何情况下，JOIN 的结合都比分隔 FROM 项的逗号更紧。

CROSS JOIN和INNER JOIN产生简单的笛卡尔积，与在FROM顶层列出这两个表得到的结果相同，但会受到连接条件（如果有）的限制。CROSS JOIN等价于INNER JOIN ON (TRUE)，也就是说，没有行会被条件过滤掉。这些连接类型只是提供了一种方便的记法，因为它们所做的一切都可以用普通的FROM和WHERE完成。

LEFT OUTER JOIN返回已限定笛卡尔积中的所有行（即，通过其连接条件的所有组合行），外加左侧表中每一行的一个副本；对于这些行，不存在通过连接条件的右侧行。这样的左侧行会通过在右侧列中插入空值而扩展到连接表的完整宽度。注意，在决定哪些行有匹配项时，只考虑JOIN子句自身的条件；外层条件是在之后应用的。

相反，RIGHT OUTER JOIN返回所有连接后的行，再加上每个未匹配右侧行对应的一行（左侧用空值扩展）。这只是一种记法上的便利，因为你可以通过交换左右表将其改写成LEFT OUTER JOIN。

FULL OUTER JOIN返回所有连接后的行，再加上每个未匹配的左侧行（右侧用空值扩展），以及每个未匹配的右侧行（左侧用空值扩展）。

ON *join_condition*

join_condition 是一个表达式，其结果类型为 boolean（类似于 WHERE 子句），用于指定连接中哪些行被视为匹配。

USING (*join_column* [, ...])

一个形如USING (a, b, ...)的子句是ON left_table.a = right_table.a AND left_table.b = right_table.b ...的简写。此外，USING意味着只有每对等价列中的一个会包含在连接输出中，而不是两者都包含。

NATURAL

NATURAL 是一个简写，表示一个列出两个表中所有同名列的 USING 列表。

WHERE Clause

可选的WHERE子句的形式

WHERE *condition*

其中*condition* 是任一计算得到boolean类型结果的表达式。任何不满足这个条件的行都会从输出中被消除。如果用一行的实际值替换其中的变量引用后，该表达式返回真，则该行符合条件。

GROUP BY Clause

可选的 GROUP BY 子句的一般形式为

GROUP BY *expression* [, ...]

GROUP BY会把所有在分组表达式上具有相同值的已选中行压缩成单独一行。*expression*可以是输入列名、输出列（SELECT列表项）的名称或序号或者由输入列值构成的任意表达式。在出现歧义时，GROUP BY名称将被解释为输入列名而不是输出列名。

聚合函数（如果使用了的话）会在组成每个组的所有行上进行计算，为每个组产生一个单独的值（而没有GROUP BY时，聚合会产生一个在所有被选中的行上计算出的单一值）。当存在GROUP BY时，SELECT列表表达式引用未分组的列是无效的，除非是在聚合函数内部，因为对一个未分组的列可能会有多个可能的返回值。

HAVING Clause

可选的HAVING子句的形式

HAVING *condition*

其中*condition*与 WHERE子句中指定的条件相同。

HAVING消除不满足该条件的分组行。HAVING与WHERE不同：WHERE会在应用GROUP BY之前过滤个体行，而HAVING过滤由 GROUP BY创建的分组行。*condition*中引用的每一列都必须无歧义地引用某个分组列，除非该引用出现在聚合函数中，或者该非分组列函数依赖于分组列。

UNION Clause

UNION 子句的一般形式是：

select_statement UNION [ALL] *select_statement*

select_statement 是不带 ORDER BY、LIMIT 或 FOR UPDATE 子句的任何 SELECT 语句。（如果用圆括号括起，ORDER BY 和 LIMIT 可以附加到子表达式上。不带圆括号时，这些子句将被认为应用于 UNION 的结果，而不是它的右侧输入表达式。）

UNION 操作符计算所涉及的 SELECT 语句返回的行的集合并集。如果一行出现在至少一个结果集中，它就在两个结果集的集合并集中。表示 UNION 的直接操作数的两个 SELECT 语句必须产生相同数量的列，并且对应的列必须是兼容的数据类型。

UNION 的结果不包含任何重复行，除非指定了 ALL 选项。ALL 阻止消除重复行。

同一 SELECT 语句中的多个 UNION 操作符从左到右求值，除非圆括号另有指示。

当前，FOR UPDATE不能为UNION的结果或UNION的任何输入指定。

INTERSECT Clause

INTERSECT 子句的一般形式是：

select_statement INTERSECT [ALL] *select_statement*

select_statement 是不带 ORDER BY、LIMIT 或 FOR UPDATE 子句的任何 SELECT 语句。

INTERSECT操作符计算相关 SELECT语句返回的行的交集。如果一行同时出现在两个结果集中，它就在交集中。

INTERSECT 的结果不包含任何重复行，除非指定了 ALL 选项。带 ALL 时，在左表中有 m 个重复且在右表中有 n 个重复的行将在结果集中出现 min(m,n) 次。

同一 SELECT 语句中的多个 INTERSECT 操作符从左到右求值，除非圆括号另有指示。INTERSECT 比 UNION 绑定得更紧。也就是说，A UNION B INTERSECT C 将被读作 A UNION (B INTERSECT C)。

EXCEPT Clause

EXCEPT 子句的一般形式是：

select_statement EXCEPT [ALL] *select_statement*

select_statement 是不带 ORDER BY、LIMIT 或 FOR UPDATE 子句的任何 SELECT 语句。

EXCEPT操作符计算位于左侧 SELECT语句的结果中但不在右侧语句结果中的行集合。

EXCEPT 的结果不包含任何重复行，除非指定了 ALL 选项。带 ALL 时，在左表中有 m 个重复且在右表中有 n 个重复的行将在结果集中出现 max(m-n,0) 次。

同一 SELECT 语句中的多个 EXCEPT 操作符从左到右求值，除非圆括号另有指示。EXCEPT 与 UNION 绑定在同一级别。

SELECT 列表

SELECT列表（位于关键词 SELECT和FROM之间）指定构成 SELECT语句输出行的表达式。这些表达式可以（并且通常确实会）引用FROM子句中计算得到的列。使用AS *output_name*子句可以为输出列指定另一个名称。这个名称主要用于给要显示的列做标记。它也可以用来在ORDER BY和GROUP BY子句中引用该列的值，但不能用在WHERE或HAVING子句中；在那里必须写出表达式。

可以在输出列表中写*来取代表达式，它是被选中行的所有列的一种简写方式。还可以写 *table_name*.*，它是只来自那个表的所有列的简写形式。

ORDER BY 子句

可选的 ORDER BY 子句的一般形式为：

ORDER BY *expression* [ASC | DESC | USING *operator*] [, ...]

expression 可以是输出列（SELECT 列表项）的名称或序号，也可以是由输入列值构成的任意表达式。

ORDER BY 子句使结果行按指定的表达式排序。如果两行按最左边的表达式相等，则按下一个表达式比较，依此类推。如果它们按所有指定的表达式都相等，则以依赖于实现顺序的顺序返回。

序号指的是输出列的顺序（从左至右）位置。
这种特性可以为不具有唯一名称的列定义一个顺序。这不是绝对必要的，因为总是可以使用 AS子句为输出列赋予一个名称。

也可以在ORDER BY子句中使用任意表达式，包括没有出现在SELECT输出列表中的列。因此，下面的语句是合法的：

```
SELECT name FROM distributors ORDER BY code;
```

这种特性的一个限制是一个应用在UNION、INTERSECT或EXCEPT子句结果上的 ORDER BY只能指定输出列名称或序号，但不能指定表达式。

如果一个ORDER BY表达式是一个既匹配输出列名称又匹配输入列名称的简单名称，ORDER BY将把它解读成输出列名称。这与在同样情况下GROUP BY会做出的选择相反。这种不一致是为了与SQL 标准兼容。

可以在ORDER BY子句中任一表达式之后附加关键字 ASC（升序）或DESC（降序）。如果没有指定，ASC被假定为默认值。或者，可以在USING 子句中指定一个特定的排序操作符名称。一个排序操作符必须是某个 B-树操作符族的小于或者大于成员。ASC通常等价于 USING <而DESC通常等价于 USING >（但是一种用户定义数据类型的创建者可以准确地定义默认排序顺序是什么，并且它可能会对应于其他名称的操作符）。

空值的排序高于任何其他值。换句话说，在升序排序时，空值排在最后；在降序排序时，
空值排在最前。

字符串数据按照数据库集簇初始化时建立的区域相关排序顺序来排序。

LIMIT 子句

该 LIMIT 子句由两个独立的子句组成：

```
LIMIT { count | ALL }
OFFSET start
```

count 指定最多返回多少行，而 *start* 指定开始返回行之前要跳过的行数。如果两者都指定，则先跳过 *start* 行，然后开始计数并返回 *count* 行。

使用 LIMIT 时，最好使用把结果行约束为唯一顺序的 ORDER BY 子句。否则你会得到查询行的一个不可预测的子集——你可能要的是第十到第二十行，
但是按什么顺序的第十到第二十行？除非指定 ORDER BY，否则你不知道顺序。

查询规划器在生成一个查询计划时会考虑LIMIT，因此根据你使用的LIMIT和OFFSET，
你很可能得到不同的计划（得到不同的行序）。所以，使用不同的 LIMIT/OFFSET值来选择一个查询结果的不同子集将会给出不一致的结果，除非你用ORDER BY强制一种可预测的结果顺序。这不是一个缺陷，它是 SQL 不承诺以任何特定顺序（除非使用 ORDER BY来约束顺序）给出一个查询结果这一事实造成的必然后果。

DISTINCT 子句

如果指定了SELECT DISTINCT，所有重复的行会被从结果集中移除（为每一组重复的行保留一行）。SELECT ALL则指定相反的行为：所有行都会被保留，这也是默认情况。

DISTINCT ON (*expression* [, ...]) 只保留给定表达式求值相等的每组行中的第一行。DISTINCT ON 表达式使用与 ORDER BY 相同的规则解释（见上文）。注意，除非用 ORDER BY 保证想要的行出现在最前，否则每个集合的“第一行”是不可预测的。例如，

```
SELECT DISTINCT ON (location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC;
```

检索每个位置的最新天气报告。但如果我们没有用 ORDER BY 强制每个位置的时间值降序，我们就会得到每个位置一个不可预测时间的报告。

DISTINCT ON表达式必须匹配最左边的 ORDER BY表达式。ORDER BY子句通常将包含额外的表达式，这些额外的表达式用于决定在每一个 DISTINCT ON分组内行的优先级。

FOR UPDATE 子句

FOR UPDATE 子句的形式如下：

```
FOR UPDATE [ OF table_name [, ...] ]
```

FOR UPDATE使SELECT语句检索到的行像被更新一样被锁定。这防止其他事务在当前事务结束之前修改或删除它们。也就是说，其他事务对这些行尝试UPDATE、DELETE或SELECT FOR UPDATE时将被阻塞，直到当前事务结束。此外，如果来自另一个事务的UPDATE、DELETE或SELECT FOR UPDATE已经锁定了被选中的一行或多行，SELECT FOR UPDATE将等待那个事务完成，然后锁定并返回更新后的行（如果该行已被删除，则不返回行）。
进一步的讨论见第 12 章。

如果在FOR UPDATE中命名了特定的表，那么只有来自那些表的行会被锁定；SELECT中用到的任何其他表都只是被照常读取。

在返回的行无法清楚地与各个表行对应起来的上下文中，不能使用FOR UPDATE；例如它不能与聚合一起使用。

FOR UPDATE 为了与 7.3 之前的 PostgreSQL 版本兼容，可以出现在 LIMIT 之前。但它实际上在 LIMIT 之后执行，所以推荐把它写在那里。

示例

连接表 films 和表 distributors：

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did

       title       | did |     name     | date_prod  |   kind
-------------------+-----+--------------+------------+----------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantic
 ...
```

要对所有电影的len列求和并且用 kind对结果分组：

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;

   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
```

```
 Musical  | 06:42
 Romantic | 04:38
```

对所有影片的 len 列求和，按 kind 分组，并显示少于 5 小时的组总计：

```
SELECT kind, sum(len) AS total
    FROM films
    GROUP BY kind
    HAVING sum(len) < interval '5 hours';

   kind   | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
```

下面两个示例都是根据第二列（`name`）的内容来排序结果：

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;

 did |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

接下来的示例展示了如何得到表`distributors`和 `actors`的并集，把结果限制为那些在每个表中以字母 W 开始的行。这里只需要不重复的行，因此省略了关键词 `ALL`。

```
distributors:               actors:
 did |     name              id |     name
-----+--------------        ----+----------------
 108 | Westward                1 | Woody Allen
 111 | Walt Disney             2 | Warren Beatty
 112 | Warner Bros.            3 | Walter Matthau
 ...                          ...

SELECT distributors.name
    FROM distributors
    WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM actors
    WHERE actors.name LIKE 'W%';
```

```
      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

这个例子展示如何在 FROM 子句中使用函数，包括带和不带列定义列表两种情况：

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS '
    SELECT * FROM distributors WHERE did = $1;
' LANGUAGE SQL;

SELECT * FROM distributors(111);
 did |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS '
    SELECT * FROM distributors WHERE did = $1;
' LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

兼容性

当然，`SELECT`语句与 SQL 标准兼容。但它也有一些扩展和缺失的特性。

省略的 FROM 子句

PostgreSQL 允许省略 `FROM` 子句。它可以直接用于计算简单表达式的结果：

```
SELECT 2+2;

 ?column?
----------
        4
```

其他一些 SQL 数据库必须引入一个只有一行的虚拟表，才能从该表执行 `SELECT`。

一个不太明显的用法是缩写对表的普通 SELECT：

```
SELECT distributors.* WHERE distributors.name = 'Westward';

 did |   name
-----+----------
 108 | Westward
```

这之所以可行，是因为对在 SELECT 语句的其他部分中被引用但在 FROM 中未提及的每个表，都会添加一个隐式的 FROM 项。

虽然这是一个方便的缩写，但很容易误用。例如，命令

```
SELECT distributors.* FROM distributors d;
```

很可能是一个错误；用户很可能想要的是

```
SELECT d.* FROM distributors d;
```

而不是他将实际得到的无约束连接

```
SELECT distributors.* FROM distributors d, distributors distributors;
```

。为了帮助检测这类错误，如果在也包含显式 FROM 子句的 SELECT 语句中使用了隐式 FROM 特性，PostgreSQL 会发出警告。此外，可以通过把 ADD_MISSING_FROM 参数设置为 false 来禁用隐式 FROM 特性。

AS 关键字

在 SQL 标准中，可选关键词AS只是一个噪音词，可以省略它而不影响含义。PostgreSQL解析器在重命名输出列时要求这个关键词，因为类型可扩展性特性使得没有它就会出现解析歧义。
不过，AS在FROM项中是可选的。

GROUP BY 和 ORDER BY 可用的命名空间

在 SQL92 标准中，ORDER BY 子句只能使用结果列名或编号，而 GROUP BY 子句只能使用基于输入列名的表达式。PostgreSQL 扩展了这两个子句，允许另一种选择（但在有歧义时使用标准的解释）。PostgreSQL 还允许两个子句指定任意表达式。注意，表达式中出现的名称总是被当作输入列名，而不是结果列名。

SQL99 使用一个略有不同的定义，它与 SQL92 不完全向上兼容。但在大多数情况下，PostgreSQL 对 ORDER BY 或 GROUP BY 表达式的解释与 SQL99 相同。

非标准子句

DISTINCT ON、LIMIT和OFFSET子句未在 SQL 标准中定义。

SELECT INTO

SELECT INTO — 从一个查询的结果创建一个新表

大纲

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]
```

描述

`SELECT INTO`创建一个新表，并用查询计算得到的数据填充该表。与普通的`SELECT`不同，这些数据不会返回给客户端。新表的列具有与`SELECT`输出列对应的名称和数据类型。

参数

`TEMPORARY` or `TEMP`

如果指定，该表将创建为临时表。详见CREATE TABLE。

`new_table`

要创建的表的名称（可选地带模式限定）。

所有其他参数都在SELECT中有详细说明。

注解

CREATE TABLE AS在功能上等价于 `SELECT INTO`。推荐使用 `CREATE TABLE AS` 语法，因为这种形式的 `SELECT INTO` 在 ECPG 或 PL/pgSQL 中不可用，因为它们对 `INTO` 子句的解释不同。

兼容性

SQL 标准用 `SELECT ... INTO` 表示把值选择到宿主程序的标量变量中，而不是创建新表。这确实是 ECPG（见第 30 章）和 PL/pgSQL（见第 37 章）中的用法。PostgreSQL 用 `SELECT INTO` 表示创建表是历史遗留。在新代码中最好用 `CREATE TABLE AS` 来实现这一目的。（`CREATE TABLE AS` 也不是标准的，但它不太容易造成混淆。）

SET

SET — 更改一个运行时参数

大纲

```
SET [ SESSION | LOCAL ] name { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT }
```

描述

SET命令用于更改运行时配置参数。许多在第 16.4 节中列出的运行时参数都可以通过 SET即时更改。（有些参数需要超级用户权限才能更改，
还有一些参数在服务器启动或会话开始之后便无法再更改。）SET只影响当前会话所使用的值。

如果在一个随后被中止的事务中发出SET或SET SESSION，那么在事务回滚时，SET命令的效果就会消失。（这一行为与PostgreSQL 7.3 之前的版本不同，在那些版本中，SET的效果不会在后续出错后回滚。）一旦该事务提交，这些效果就会持续到会话结束，除非被另一个SET覆盖。

SET LOCAL的效果只持续到当前事务结束，无论事务是否提交。一种特殊情况是：在同一个事务中先执行 SET，再执行SET LOCAL。这样SET LOCAL设置的值会一直生效到事务结束，但在此之后（如果该事务被提交），SET设置的值将生效。

参数

SESSION

指定该命令对当前会话生效。（如果既没有出现SESSION，也没有出现LOCAL，
这就是默认情况。）

LOCAL

指定该命令只对当前事务生效。在COMMIT或 ROLLBACK之后，会话级设置会再次生效。注意，SET LOCAL如果在BEGIN块之外执行，看起来不会有任何效果，因为事务会立即结束。

name

可设置的运行时参数名称。可用参数见第 16.4 节以及下文。

value

参数的新值。值可以指定为字符串常量、标识符、数字，或由这些构成的逗号分隔列表。DEFAULT可用于表示把该参数重置为其默认值。

除了第 16.4 节中记录的配置参数之外，还有少数几个只能用 SET 命令调整，或者具有特殊语法：

NAMES

SET NAMES *value*是 SET client_encoding TO *value*的别名。

SEED

设置随机数生成器（即函数 random）的内部种子。允许的值是 0 到 1 之间的浮点数，这些值随后会乘以 2^{31}-1。

种子也可以通过调用函数setseed来设置：

```
SELECT setseed(value);
```

TIME ZONE

SET TIME ZONE *value* 是 SET timezone TO *value* 的别名。SET TIME ZONE 语法允许为时区指定使用特殊语法。下面是一些有效值的例子（但注意其中一些只在某些平台上被接受）：

'PST8PDT'

加利福尼亚州伯克利的时区。

'Portugal'

葡萄牙的时区。

'Europe/Rome'

意大利的时区。

-7

UTC 以西 7 小时的时区（相当于 PDT）。正值表示 UTC 以东。

INTERVAL '-08:00' HOUR TO MINUTE

UTC 以西 8 小时的时区（相当于 PST）。

LOCAL
DEFAULT

把时区设为你的本地时区（服务器操作系统默认的时区）。

有关时区的更多信息见第 8.5 节。

注解

函数set_config提供了等效功能；见第 9.13 节。

示例

设置模式搜索路径：

```
SET search_path TO my_schema, public;
```

将日期风格设置为传统POSTGRES风格，并采用“日在月之前”的输入惯例：

```
SET datestyle TO postgres, dmy;
```

把时区设为加利福尼亚州伯克利的时区，使用引号来保留时区名的大写拼写：

```
SET TIME ZONE 'PST8PDT';
SELECT current_timestamp AS today;
```

```
             today
-------------------------------
 2003-04-29 15:02:01.218622-07
```

兼容性

`SET TIME ZONE`扩展了 SQL 标准定义的语法。标准只允许使用数值形式的时区偏移量，而 PostgreSQL 允许更灵活的时区指定方式。其他所有`SET`特性都是 PostgreSQL扩展。

另见

RESET, SHOW

SET CONSTRAINTS

SET CONSTRAINTS — 设置当前事务的约束模式

大纲

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

描述

`SET CONSTRAINTS`设置当前事务中约束检查的行为。在`IMMEDIATE`模式下，约束在每条语句结束时检查。在`DEFERRED`模式下，约束直到事务提交时才检查。

当你把一个约束的模式改为 `IMMEDIATE`时，新的约束模式会追溯生效：任何原本会在事务结束时（使用 `DEFERRED`时）检查的未完成数据修改，都会改在`SET CONSTRAINTS`命令执行期间检查。

约束在创建时总是被赋予下列三种特性之一：`INITIALLY DEFERRED`、`INITIALLY IMMEDIATE DEFERRABLE`或 `INITIALLY IMMEDIATE NOT DEFERRABLE`。第三类不受`SET CONSTRAINTS`命令的影响。

目前，只有外键约束受此设置影响。检查和唯一约束实际上总是初始立即且不可推迟的。

注解

这条命令只更改当前事务内约束的行为。因此，如果你在事务块（`BEGIN`/`COMMIT`对）之外执行这条命令，它将看不出有任何效果。
如果你希望更改一个约束的行为而又不必在每个事务中都发出一条 `SET CONSTRAINTS`命令，可以在创建约束时指定 `INITIALLY DEFERRED`或`INITIALLY IMMEDIATE`。

兼容性

这条命令符合 SQL 标准定义的行为，但有一个限制：在 PostgreSQL中它只适用于外键约束。

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION — 设置当前会话的会话用户标识符和当前用户标识符

大纲

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION username
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

描述

这条命令把当前 SQL 会话的会话用户标识符和当前用户标识符设置为 *username*。用户名既可以写成标识符，也可以写成字符串字面量。使用这条命令，例如可以暂时变为一个非特权用户，之后再切换回超级用户。

会话用户标识符最初被设置为客户端提供的（可能经过认证的）用户名。当前用户标识符通常等于会话用户标识符，但在 `SECURITY DEFINER`函数及类似机制的上下文中可能发生临时改变。当前用户标识符与权限检查相关。

只有当初始会话用户（已认证用户）拥有超级用户权限时，才能更改会话用户标识符。否则，只有当命令指定的是已认证用户名时它才会被接受。

`SESSION`和`LOCAL`修饰符的作用与常规的SET命令中相同。

`DEFAULT`和`RESET`形式把会话用户标识符和当前用户标识符重置为最初认证的用户名。这些形式可以由任何用户执行。

注意

`SET SESSION AUTHORIZATION`不能在 `SECURITY DEFINER`函数中使用。

示例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

兼容性

SQL 标准允许在字面量*username*的位置出现一些其他表达式，但它们在实践中并不重要。PostgreSQL 允许标识符语法（`"username"`），而 SQL 不允许。SQL 不允许在事务中执行这条

命令；PostgreSQL 没有做这个限制，因为没有理由这样做。执行这条命令所需的权限由标准留给实现定义。

SET TRANSACTION

SET TRANSACTION — 设置当前事务的特性

大纲

```
SET TRANSACTION
    [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ] [ READ WRITE
 | READ ONLY ]
SET SESSION CHARACTERISTICS AS TRANSACTION
    [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ] [ READ WRITE
 | READ ONLY ]
```

描述

`SET TRANSACTION` 命令设置当前事务的事务特性。它对任何后续事务都没有影响。`SET SESSION CHARACTERISTICS` 设置一个会话中每个事务的默认事务特性。对于单个事务，`SET TRANSACTION` 可以覆盖它。

可用的事务特性是事务隔离级别和事务访问模式（读/写或只读）。

事务的隔离级别决定该事务在其他事务并发运行时能看到哪些数据。

`READ COMMITTED`

一条语句只能看到在它开始之前已提交的行。这是默认值。

`SERIALIZABLE`

当前事务只能看到在本事务中第一条查询或数据修改语句执行之前已提交的行。

> **提示**
>
> 直观地说，可串行化意味着两个并发事务给数据库留下的状态，
> 与这两个事务严格地一个接一个按任一顺序执行后的状态相同。

The transaction isolation level cannot be set after the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, `COPY`) of a transaction has been executed. See 第 12 章 for more information about transaction isolation and concurrency control.

事务访问模式决定事务是读/写还是只读。默认为读/写。当事务为只读时，下列 SQL 命令被禁止：`INSERT`、`UPDATE`、`DELETE` 和 `COPY FROM`（如果它们要写入的表不是临时表）；所有 `CREATE`、`ALTER` 和 `DROP` 命令；`COMMENT`、`GRANT`、`REVOKE`、`TRUNCATE`；以及 `EXPLAIN ANALYZE` 和 `EXECUTE`（如果它们要执行的命令在上述之列）。这是高层的只读概念，并不阻止写磁盘。

注解

会话默认的事务隔离级别也可以用命令

```
SET default_transaction_isolation = 'value'
```

以及在配置文件中设置。更多信息请参考第 16.4 节。

兼容性

这两条命令都在 SQL 标准中定义。标准中默认的事务隔离级别是 SERIALIZABLE；在 PostgreSQL 中默认通常是 READ COMMITTED，但你可以按上文所述更改它。PostgreSQL 不提供隔离级别 READ UNCOMMITTED 和 REPEATABLE READ。由于多版本并发控制，SERIALIZABLE 级别并非真正的可串行化。细节见第 12 章。

在 SQL 标准中，还有一项可以用这些命令设置的事务特性：诊断区域的大小。这个概念只在嵌入式 SQL 中使用。

SHOW

SHOW — 显示一个运行时参数的值

大纲

```
SHOW name
SHOW ALL
```

描述

SHOW将显示运行时参数的当前设置。这些变量可以使用SET语句来设置，也可以通过编辑postgresql.conf配置文件、使用 PGOPTIONS环境变量（在使用 libpq或基于libpq的应用时），或者在启动postmaster时通过命令行标志来设置。详见第 16.4 节。

参数

name

运行时参数的名称。可用参数的说明见第 16.4 节以及SET参考页。此外，还有几个可以显示但不能设置的参数：

SERVER_VERSION

显示服务器的版本号。

SERVER_ENCODING

显示服务器端字符集编码。目前，该参数可以显示但不能设置，因为编码在数据库创建时就已确定。

LC_COLLATE

显示数据库用于排序规则（文本排序）的区域设置。目前，该参数可以显示但不能设置，因为该设置在initdb时就已确定。

LC_CTYPE

显示数据库用于字符分类的区域设置。目前，该参数可以显示但不能设置，因为该设置在initdb时就已确定。

IS_SUPERUSER

如果当前会话授权标识符具有超级用户权限，则为真。

ALL

显示所有配置参数的值。

注解

函数current_setting也会产生等效的输出，见第 9.13 节。

示例

显示参数DateStyle的当前设置：

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

显示参数geqo的当前设置：

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

显示所有设置：

```
SHOW ALL;
              name                |                 setting
----------------------------------+-------------------------------------
--
 australian_timezones             | off
 authentication_timeout           | 60
 checkpoint_segments              | 3
    .
    .
    .
 wal_debug                        | 0
 wal_sync_method                  | fdatasync
(94 rows)
```

兼容性

SHOW命令是一种 PostgreSQL扩展。

另见

SET

START TRANSACTION

START TRANSACTION — 开始一个事务块

大纲

```
START TRANSACTION [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
 ] [ READ WRITE | READ ONLY ]
```

描述

这条命令开始一个新事务。如果指定了隔离级别或读/写模式，新事务就具有这些特性，
就像执行了SET TRANSACTION一样。在所有其他方面，这条命令的行为与BEGIN命令相同。

参数

参数的含义请参阅SET TRANSACTION。

兼容性

这条命令符合 SQL 标准；但另见SET TRANSACTION的兼容性一节。

另见

BEGIN, COMMIT, ROLLBACK, SET TRANSACTION

TRUNCATE

TRUNCATE — 清空一个表

大纲

```
TRUNCATE [ TABLE ] name
```

描述

TRUNCATE快速移除一个表中的所有行。它的效果与不带条件的DELETE相同，但由于它并不实际扫描表，因此速度更快。这对大表最有用。

参数

name

要截断的表的名字（可以用模式限定）。

注意

只有表的拥有者才能TRUNCATE它。

如果有来自其他表的外键引用指向该表，则不能使用TRUNCATE。这种情况下检查有效性需要扫描表，而这条命令的要点恰恰是不扫描。

TRUNCATE不会运行该表上可能存在的任何用户定义的 ON DELETE 触发器。

示例

截断表bigtable：

```
TRUNCATE TABLE bigtable;
```

兼容性

SQL 标准中没有TRUNCATE命令。

UNLISTEN

UNLISTEN — 停止监听某个通知

大纲

```
UNLISTEN { name | * }
```

描述

`UNLISTEN`用于移除一个现有的 `NOTIFY`事件注册。`UNLISTEN`取消当前 PostgreSQL会话作为通知*name*监听者的任何现有注册。特殊的通配符`*`取消当前会话的所有监听者注册。

NOTIFY中包含关于`LISTEN`和 `NOTIFY`用法的更广泛的讨论。

参数

name

通知的名称（任何标识符）。

*

清除该会话当前的所有监听注册。

注解

你可以取消监听一个你并没有在监听的东西；不会出现任何警告或错误。

每个会话结束时，`UNLISTEN *`会被自动执行。

示例

建立一个注册：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with
 PID 8448.
```

一旦执行了`UNLISTEN`，后续的`NOTIFY` 命令将被忽略：

```
UNLISTEN virtual;
NOTIFY virtual;
-- no NOTIFY event is received
```

兼容性

SQL 标准中没有`UNLISTEN`命令。

另见

LISTEN, NOTIFY

UPDATE

UPDATE — 更新表中的行

大纲

```
UPDATE [ ONLY ] table SET column = { expression | DEFAULT } [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

描述

`UPDATE` 更改满足条件的所有行中指定列的值。语句中只需要提及要修改的列；没有显式 `SET` 的列保留它们先前的值。

默认情况下，`UPDATE` 将更新指定表及其所有子表中的行。如果只想更新提到的这个特定表，必须使用 `ONLY` 子句。

要更新表，你必须拥有该表上的 `UPDATE` 权限，以及对在 *expression* 或 *condition* 中读取其值的任何表的 `SELECT` 权限。

参数

table

要更新的表的名称（可带模式限定）。

column

table 中一个列的名称。

expression

赋给该列的表达式。该表达式可以使用该表以及其他列的旧值。

`DEFAULT`

把该列设置为它的默认值（如果没有给它分配特定的默认表达式，则为 NULL）。

fromlist

一个表表达式的列表，允许来自其他表的列出现在 `WHERE` 条件和更新表达式中。

condition

一个返回 `boolean` 类型值的表达式。只有该表达式返回 `true` 的行才会被更新。

输出

成功完成时，`UPDATE` 命令返回一个形如

`UPDATE` *count*

的命令标签。*count* 是被更新的行数。如果 *count* 为 0，说明没有行匹配 *condition*（这不被认为是错误）。

示例

把表 `films` 的列 `kind` 中的单词 `Drama` 改为 `Dramatic`：

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

在表 `weather` 的一行中调整温度项并把降水重置为其默认值：

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp =
 DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

兼容性

这个命令符合 SQL 标准。`FROM` 子句是一种 PostgreSQL 扩展。

VACUUM

VACUUM — 垃圾收集并且可选地分析一个数据库

大纲

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (column [, ..
.] ) ] ]
```

描述

VACUUM回收由已删除元组占用的存储空间。在正常的 PostgreSQL操作中，被删除或被更新作废的元组不会从它们的表中被物理移除；它们会一直存在，直到执行一次VACUUM为止。
因此有必要周期性地执行VACUUM，特别是在频繁更新的表上。

不带参数时，VACUUM处理当前数据库中的每个表。带参数时，VACUUM只处理那个表。

VACUUM ANALYZE对每个选中的表先执行一次VACUUM，然后再执行一次ANALYZE。这是例行维护脚本的一种便利组合形式。其处理的更多细节见ANALYZE。

普通VACUUM（不带FULL）只是回收空间并使其可被重新使用。
这种形式的命令可以与对表的正常读写并行运行，因为它不获取排他锁。VACUUM FULL做更广泛的处理，包括跨块移动元组以尽量把表压缩到最少的磁盘块数。这种形式要慢得多，
并且在处理每个表时需要该表上的排他锁。

FREEZE是一个特殊用途的选项，它使元组尽可能早地被标记为“冻结”，
而不是等到它们相当老时。如果在没有同一数据库中其他打开事务的情况下执行它，
就可以保证数据库中的所有元组都是“冻结”的，无论数据库多长时间不清理，都不会受到事务 ID 回卷问题的影响。FREEZE不建议例行使用。它唯一预期的用途是在准备用户定义模板数据库、或者其他完全只读且不会接受例行VACUUM维护操作的数据库时使用。细节见第 21 章。

参数

FULL

选择“full”清理，它可以回收更多空间，但耗时长得得多，并且对表加排他锁。

FREEZE

选择对元组进行激进的“冻结”。

VERBOSE

为每个表打印一份详细的清理活动报告。

ANALYZE

更新规划器用来确定执行查询最有效方式所使用的统计信息。

table

要清理的指定表的名称（可以有模式限定）。默认为当前数据库中的所有表。

column

要分析的指定列的名称。默认为所有列。

输出

当指定VERBOSE时，VACUUM会发出进度消息来指示当前正在处理哪个表。
同时还会打印关于表的各种统计信息。

注意

我们建议对活跃的生产数据库频繁地执行清理（至少每晚一次），以移除过期的行。
在增加或删除大量行之后，对受影响的表发出一条VACUUM ANALYZE命令可能是个好主意。这会用所有最近变更的结果更新系统目录，使 PostgreSQL查询规划器在规划查询时做出更好的选择。

FULL选项不建议例行使用，但在特殊情况下可能有用。例如，
当你删除了一个表中的大部分行并希望该表物理收缩以占用更少的磁盘空间时。VACUUM FULL通常比普通VACUUM能把表收缩得更多。

示例

下面是对回归数据库中的一个表运行VACUUM的例子：

```
regression=# VACUUM VERBOSE ANALYZE onek;
INFO:  vacuuming "public.onek"
INFO:  index "onek_unique1" now contains 1000 tuples in 14 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.18 sec.
INFO:  index "onek_unique2" now contains 1000 tuples in 16 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.07u sec elapsed 0.23 sec.
INFO:  index "onek_hundred" now contains 1000 tuples in 13 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.17 sec.
INFO:  index "onek_stringu1" now contains 1000 tuples in 48 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.09u sec elapsed 0.59 sec.
INFO:  "onek": removed 3000 tuples in 108 pages
DETAIL:  CPU 0.01s/0.06u sec elapsed 0.07 sec.
INFO:  "onek": found 3000 removable, 1000 nonremovable tuples in 143
 pages
DETAIL:  0 dead tuples cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.07s/0.39u sec elapsed 1.56 sec.
INFO:  analyzing "public.onek"
INFO:  "onek": 36 pages, 1000 rows sampled, 1000 estimated total rows
VACUUM
```

兼容性

SQL 标准中没有`VACUUM`语句。

另见

vacuumdb

PostgreSQL 客户端应用

这一部分包含PostgreSQL客户端应用和实用工具的参考信息。并非所有这些命令都是通用的，有些可能需要特殊权限。这些应用的共同特点是它们可以在任何主机上运行，与数据库服务器所在的位置无关。

目录

clusterdb 783
createdb 785
createlang 788
createuser 790
dropdb 793
droplang 795
dropuser 797
ecpg 799
pg_config 801
pg_dump 803
pg_dumpall 809
pg_restore 812
pgtclsh 818
pgtksh 819
psql 820
vacuumdb 842

clusterdb

clusterdb — 对 PostgreSQL 数据库进行聚簇

大纲

`clusterdb` [*connection-option*...] [--table | -t *table*] [*dbname*]
`clusterdb` [*connection-option*...] [[--all] | [-a]]

描述

clusterdb是一个用于重新聚簇PostgreSQL 数据库中各表的工具。它会找出此前已经聚簇过的表，并按照上次使用的同一索引再次对其进行聚簇。从未聚簇过的表不会受到影响。

clusterdb是 SQL 命令CLUSTER的一个包装器。使用该工具聚簇数据库，与使用其他访问服务器的方法聚簇数据库，并没有实质区别。

选项

clusterdb接受以下命令行参数：

`-a`
`--all`

聚簇所有数据库。

`[-d]` *dbname*
`[--dbname=]`*dbname*

指定要聚簇的数据库名。如果未指定并且没有使用 `-a`（或`--all`），则从环境变量 `PGDATABASE` 读取数据库名。如果该环境变量也未设置，则使用为该连接指定的用户名作为数据库名。

`-e`
`--echo`

回显clusterdb生成并发送到服务器的命令。

`-q`
`--quiet`

不显示响应。

`-t` *table*
`--table` *table*

只对 *table* 执行聚簇。

clusterdb还接受以下用于连接参数的命令行参数：

`-h` *host*
`--host` *host*

指定运行服务器的机器的主机名。如果该值以斜杠开头，则它将被用作 Unix 域套接字的目录。

`-p port`
`--port port`

指定服务器监听连接的 TCP 端口，或本地 Unix 域套接字文件扩展名。

`-U username`
`--username username`

用于连接的用户名。

`-W`
`--password`

强制提示输入密码。

环境

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

诊断

如果遇到问题，请参见CLUSTER和psql中关于潜在问题和错误消息的讨论。
数据库服务器必须在目标主机上运行。此外，
libpq前端库所使用的任何默认连接设置和环境变量都会生效。

示例

要聚簇数据库`test`：

```
$ clusterdb test
```

要聚簇名为`xyzzy`的数据库中的单个表`foo`：

```
$ clusterdb --table foo xyzzy
```

参见

CLUSTER

createdb

createdb — 创建一个新的 PostgreSQL 数据库

大纲

`createdb` [*option*...] [*dbname*] [*description*]

描述

createdb创建一个新的PostgreSQL数据库。

通常，执行此命令的数据库用户会成为新数据库的所有者。不过，
如果执行用户具有相应的权限，也可以通过`-O`选项指定不同的所有者。

createdb是SQL命令CREATE DATABASE的包装器。通过此工具创建数据库与通过其他访问服务器的方法创建数据库，在效果上没有区别。

选项

createdb 接受下列命令行参数：

dbname

指定要创建的数据库名称。该名称必须在此数据库集簇中的所有PostgreSQL数据库中唯一。默认情况下，会创建一个与当前系统用户同名的数据库。

description

可选地指定与新创建的数据库关联的注释。

`-D` *location*
`--location` *location*

指定数据库的备选位置。另见initlocation。

`-e`
`--echo`

回显createdb生成并发送到服务器的命令。

`-E` *encoding*
`--encoding` *encoding*

指定该数据库使用的字符编码方案。

`-O` *owner*
`--owner` *owner*

指定将拥有新数据库的数据库用户。

`-q`
`--quiet`

不显示响应。

`-T` *`template`*
`--template` *`template`*

指定用于构建该数据库的模板数据库。

`-D`、`-E`、`-O`和`-T`选项对应底层 SQL 命令CREATE DATABASE的选项；有关这些选项的更多信息请参见该命令。

createdb还接受以下用于连接参数的命令行参数：

`-h` *`host`*
`--host` *`host`*

指定服务器所在机器的主机名。如果该值以斜杠开头，则它将被用作 Unix 域套接字的目录。

`-p` *`port`*
`--port` *`port`*

指定服务器监听连接所用的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U` *`username`*
`--username` *`username`*

以该用户名进行连接。

`-W`
`--password`

强制提示输入密码。

环境

`PGDATABASE`

如果设置了该变量，其值就是要创建的数据库名称，除非被命令行覆盖。

`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数。如果命令行和`PGDATABASE`都没有指定要创建的数据库名称，则`PGUSER`也会决定该名称。

诊断

遇到困难时，请参阅CREATE DATABASE和psql中对潜在问题和错误消息的讨论。数据库服务器必须在目标主机上运行。此外，libpq 前端库使用的任何默认连接设置和环境变量都将适用。

示例

要使用默认数据库服务器创建`demo`数据库：

```
$ createdb demoCREATE DATABASE
```

其响应与你直接运行`CREATE DATABASE` SQL命令所得到的相同。

要使用主机eden上、端口为 5000 的服务器，并以 LATIN1编码方案创建demo数据库，可以看看下面的底层命令：

```
$ createdb -p 5000 -h eden -E LATIN1 -e demoCREATE DATABASE "demo"
 WITH ENCODING = 'LATIN1'CREATE DATABASE
```

参见

dropdb, CREATE DATABASE

createlang

createlang — 定义一种新的 PostgreSQL 过程语言

大纲

```
createlang [connection-option...] langname [dbname]
createlang [connection-option...] [--list] | [-l] dbname
```

描述

createlang是一个向PostgreSQL数据库添加新编程语言的工具。
createlang能处理默认PostgreSQL发行版中提供的所有语言，但不能处理其他方提供的语言。

虽然可以直接使用几条SQL命令来添加后端编程语言，但建议使用 createlang，因为它会执行多项检查，而且用起来容易得多。更多信息见CREATE LANGUAGE。

选项

createlang 接受下列命令行参数：

langname

指定要安装的过程语言的名称。

[-d] *dbname*
[--dbname=]*dbname*

指定要向哪个数据库添加该语言。默认使用与当前系统用户同名的数据库。

-e
--echo

在执行 SQL 命令时把它们显示出来。

-l
--list

显示目标数据库中已安装语言的列表。

-L *directory*

指定语言解释器所在的目录。该目录通常会被自动找到；此选项主要用于调试目的。

createlang 还接受下列用于连接参数的命令行参数：

-h *host*
--host *host*

指定服务器所在主机的主机名。如果该值以斜杠开头，则将其用作 Unix 域套接字所在目录。

-p *port*
--port *port*

指定服务器监听连接所用的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U` *`username`*
`--username` *`username`*

要用来连接的用户名。

`-W`
`--password`

强制提示输入密码。

环境

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

诊断

大多数错误消息都是不言自明的。如果不是，请带 `--echo` 选项运行 createlang，并参见相应的 SQL 命令的详细说明。

注解

使用droplang可移除一个语言。

示例

要将语言 `pltcl` 安装到数据库 `template1` 中：

```
$ createlang pltcl template1
```

参见

droplang, CREATE LANGUAGE

createuser

createuser — 定义一个新的 PostgreSQL 用户账户

大纲

`createuser` [*option*...] [*username*]

描述

createuser创建一个新的 PostgreSQL用户。只有超级用户（在 `pg_shadow`表中`usesuper`被设置的用户）才能创建新的 PostgreSQL用户，因此 createuser必须由能以 PostgreSQL超级用户身份连接的人来调用。

成为超级用户还意味着能够绕过数据库内的访问权限检查，所以不应轻易授予超级用户身份。

createuser是 SQL命令CREATE USER的一个包装。通过这个实用程序创建用户与通过其他访问服务器的方法创建用户没有实质区别。

选项

createuser接受下列命令行参数：

username

指定要创建的 PostgreSQL 用户的名称。这个名称必须在所有 PostgreSQL用户之间唯一。

`-a`
`--adduser`

允许新用户创建其他用户。（注意：实际上，这会使新用户成为超级用户。
这个选项的命名并不恰当。）

`-A`
`--no-adduser`

不允许新用户创建其他用户（即新用户是普通用户，不是超级用户）。这是默认值。

`-d`
`--createdb`

允许新用户创建数据库。

`-D`
`--no-createdb`

不允许新用户创建数据库。这是默认值。

`-e`
`--echo`

显示 createuser 生成并发送给服务器的命令。

`-E`
`--encrypted`

加密存储在数据库中的用户口令。如果未指定，则使用默认的口令行为。

`-i` *`number`*
`--sysid` *`number`*

允许你为新用户选择一个非默认的用户 ID。这并非必需，但有些人喜欢这样。

`-N`
`--unencrypted`

不加密存储在数据库中的用户口令。如果未指定，则使用默认的口令行为。

`-P`
`--pwprompt`

如果给出这个选项，createuser将提示输入新用户的口令。如果你不打算使用口令认证，则不需要它。

`-q`
`--quiet`

不显示响应。

如果命令行上没有指定名称和其他缺少的信息，系统会提示你输入。

createuser还接受下列用于连接参数的命令行参数：

`-h` *`host`*
`--host` *`host`*

指定服务器正在运行的主机名。如果值以斜杠开头，它将被用作 Unix 域套接字的目录。

`-p` *`port`*
`--port` *`port`*

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U` *`username`*
`--username` *`username`*

要连接的用户名（不是要创建的用户名）。

`-W`
`--password`

强制口令提示（用于连接服务器，而不是新用户的口令）。

环境

`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

诊断

如果遇到困难，请参见CREATE USER和psql中关于潜在问题和错误消息的讨论。数据库服务器必须在目标主机上运行。此外，libpq前端库使用的任何默认连接设置和环境变量都将适用。

示例

在默认数据库服务器上创建用户 `joe`：

```
$ createuser joeIs the new user allowed to create databases? (y/
n) nShall the new user be allowed to create more new users? (y/n)
 nCREATE USER
```

使用主机 `eden`上端口 5000 的服务器创建同一个用户 `joe`，避开提示并查看底层命令：

```
$ createuser -p 5000 -h eden -D -A -e joeCREATE USER "joe" NOCREATEDB
 NOCREATEUSERCREATE USER
```

另见

dropuser, CREATE USER

dropdb

dropdb — 移除一个 PostgreSQL 数据库

大纲

`dropdb` [*option*...] *dbname*

描述

dropdb 销毁一个现有的 PostgreSQL 数据库。执行这个命令的用户必须是数据库超级用户或该数据库的拥有者。

dropdb 是 SQL 命令DROP DATABASE的包装。通过这个工具删除数据库和通过其他访问服务器的方法删除数据库之间没有实质差别。

选项

dropdb 接受下列命令行参数：

dbname

指定要移除的数据库的名称。

`-e`
`--echo`

回显 dropdb 生成并发送给服务器的命令。

`-i`
`--interactive`

在做任何破坏性操作之前发出一个确认提示。

`-q`
`--quiet`

不显示响应。

dropdb 还接受下列用于连接参数的命令行参数：

`-h` *host*
`--host` *host*

指定服务器正在运行的机器的主机名。如果值以斜杠开头，它被用作 Unix 域套接字的目录。

`-p` *port*
`--port` *port*

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

`-U` *username*
`--username` *username*

要连接的用户名。

`-W`
`--password`

强制口令提示。

环境

`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数。

诊断

出现问题时，请参阅DROP DATABASE和psql中对潜在问题和错误消息的讨论。数据库服务器必须在目标主机上运行。此外，libpq 前端库使用的任何默认连接设置和环境变量都会适用。

示例

在默认数据库服务器上销毁数据库 `demo`：

```
$ dropdb demoDROP DATABASE
```

使用主机 `eden` 上端口 5000 的服务器销毁数据库 `demo`，带确认并看一眼底层命令：

```
$ dropdb -p 5000 -h eden -i -e demoDatabase "demo" will be permanently
 deleted.
Are you sure? (y/n) yDROP DATABASE "demo"
DROP DATABASE
```

另见

createdb, DROP DATABASE

droplang

droplang — 移除一种 PostgreSQL 过程语言

大纲

```
droplang [connection-option...] langname [dbname]
droplang [connection-option...] [--list] | [-l] dbname
```

描述

droplang是一个从PostgreSQL数据库移除现有编程语言的工具。
droplang可以删除任何过程语言，甚至是那些不由PostgreSQL发行版提供的过程语言。

虽然后端编程语言可以直接用几条SQL命令移除，但建议使用droplang，
因为它会执行一些检查，而且用起来容易得多。更多信息见DROP LANGUAGE。

选项

droplang 接受下列命令行参数：

langname

指定要移除的过程语言的名称。

[-d] *dbname*
[--dbname] *dbname*

指定要从哪个数据库中移除该语言。默认使用与当前系统用户同名的数据库。

-e
--echo

在执行 SQL 命令时把它们显示出来。

-l
--list

显示目标数据库中已安装语言的列表。

droplang 还接受下列用于连接参数的命令行参数：

-h *host*
--host *host*

指定服务器所在主机的主机名。如果该值以斜杠开头，则将其用作 Unix 域套接字所在目录。

-p *port*
--port *port*

指定服务器监听连接所用的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U` *`username`*
`--username` *`username`*

要用来连接的用户名。

`-W`
`--password`

强制提示输入密码。

环境

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

诊断

大多数错误消息都是不言自明的。如果不是，请带 `--echo` 选项运行 droplang，并参见相应的 SQL 命令的详细说明。

注解

使用createlang可添加一个语言。

示例

要移除语言 `pltcl`：

```
$ droplang pltcl dbname
```

参见

createlang, DROP LANGUAGE

dropuser

dropuser — 移除一个 PostgreSQL 用户账户

大纲

`dropuser` [*option*...] [*username*]

描述

dropuser移除现有的PostgreSQL用户以及该用户所拥有的数据库。只有超级用户（即`pg_shadow`表中`usesuper`被置位的用户）才能删除 PostgreSQL用户。

dropuser是SQL命令DROP USER的包装器。通过此工具删除用户与通过其他访问服务器的方法删除用户，在效果上没有区别。

选项

dropuser 接受下列命令行参数：

username

指定要移除的 PostgreSQL 用户的名称。如果命令行上没有指定名称，将会提示你输入。

`-e`
`--echo`

回显dropuser生成并发送给服务器的命令。

`-i`
`--interactive`

在实际移除用户之前提示确认。

`-q`
`--quiet`

不显示响应。

dropuser也接受下列用于连接参数的命令行参数：

`-h` *host*
`--host` *host*

指定服务器运行所在主机的主机名。如果该值以斜杠开头，则它将被用作 Unix 域套接字所在目录。

`-p` *port*
`--port` *port*

指定服务器监听连接所用的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U` *`username`*
`--username` *`username`*

用于连接的用户名（不是要删除的用户名）。

`-W`
`--password`

强制提示输入密码（用于连接服务器，而不是要被删除的用户的密码）。

环境

`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

诊断

遇到困难时，请参阅DROP USER和psql中对潜在问题和错误消息的讨论。数据库服务器必须在目标主机上运行。此外，libpq 前端库使用的任何默认连接设置和环境变量都将适用。

示例

要从默认数据库服务器上移除用户`joe`：

```
$ dropuser joeDROP USER
```

要使用主机`eden`、端口 5000 上的服务器移除用户`joe`，并带确认提示以及查看底层命令：

```
$ dropuser -p 5000 -h eden -i -e joeUser "joe" and any owned databases
 will be permanently deleted.
Are you sure? (y/n) yDROP USER "joe"
DROP USER
```

另见

createuser, DROP USER

ecpg

ecpg — 嵌入式 SQL C 预处理器

大纲

ecpg [*option*...] *file*...

描述

ecpg是供 C 程序使用的嵌入式 SQL 预处理器。它会把嵌入了 SQL 语句的 C 程序转换成普通 C 代码，方法是将 SQL 调用替换为特殊的函数调用。生成的输出文件随后可以由任何 C 编译器工具链处理。

ecpg 会把命令行上给出的每个输入文件转换为对应的 C 输出文件。输入文件最好带有扩展名 .pgc，这种情况下会用 .c 替换该扩展名来确定输出文件名。如果输入文件的扩展名不是 .pgc，则通过在整个文件名后追加 .c 来计算输出文件名。也可以使用 -o 选项覆盖输出文件名。

本参考页不描述嵌入式 SQL 语言。有关该主题的更多信息，见第 30 章。

选项

ecpg 接受下列命令行参数：

-c

自动从 SQL 代码生成某些 C 代码。目前，这适用于EXEC SQL TYPE。

-C *mode*

设置兼容模式。*mode*可以是INFORMIX或INFORMIX_SE。

-D *symbol*

定义一个 C 预处理器符号。

-i

也解析系统 include 文件。

-I *directory*

指定一个附加的 include 路径，用于查找通过EXEC SQL INCLUDE包含的文件。默认路径依次为 .（当前目录）、/usr/local/include、在编译时定义的PostgreSQL include 目录（默认：/usr/local/pgsql/include），以及 /usr/include。

-o *filename*

指定ecpg应将全部输出写入给定的*filename*。

-r *option*

选择一种运行时行为。当前*option* 只能是no_indicator。

`-t`

打开事务自动提交。在此模式下，除非 SQL 命令位于显式事务块中，否则每条 SQL 命令都会自动提交。在默认模式下，只有发出`EXEC SQL COMMIT`时才会提交命令。

`-v`

打印附加信息，包括版本和 include 路径。

`--help`

显示命令用法的简要摘要，然后退出。

`--version`

输出版本信息，然后退出。

注解

在编译预处理后的 C 代码文件时，编译器必须能够在PostgreSQL的 include 目录中找到ECPG头文件。因此，调用编译器时你可能需要使用`-I`选项（例如`-I/usr/local/pgsql/include`）。

使用嵌入式 SQL 的 C 程序必须与`libecpg`库链接，例如可使用链接器选项`-L/usr/local/pgsql/lib -lecpg`。

适合该安装的这两个目录的值可以通过pg_config查出。

示例

如果你有一个名为`prog1.pgc`的嵌入式 SQL C 源文件，就可以使用下列命令序列创建一个可执行程序：

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_config

pg_config — 检索关于已安装版本的PostgreSQL的信息

大纲

`pg_config` {[--bindir] | [--includedir] | [--includedir-server] | [--libdir] | [--pkglibdir] | [--configure] | [--version]...}

描述

pg_config实用程序打印当前安装版本的 PostgreSQL的配置参数。例如，它面向想要与PostgreSQL对接的软件包使用，以便于找到所需的头文件和库。

选项

要使用pg_config，需要提供下列一个或多个选项：

`--bindir`

打印用户可执行文件的位置。例如，可以用它来找到`psql` 程序。这里通常也是`pg_config`程序所在的位置。

`--includedir`

打印客户端接口的 C 头文件的位置。

`--includedir-server`

打印用于服务器编程的 C 头文件的位置。

`--libdir`

打印目标代码库的位置。

`--pkglibdir`

打印动态可装载模块的位置，或服务器搜索它们的位置。
（其他依赖体系结构的数据文件也可能安装在这个目录中。）

`--configure`

打印在配置`configure`脚本以构建 PostgreSQL时传给它的选项。这可以用来重现完全相同的配置，或者查明一个二进制软件包是用什么选项构建的。（不过注意，
二进制软件包常常包含发行商特有的自定义补丁。）

`--version`

打印PostgreSQL的版本并退出。

如果给出多个选项（`--version`除外），信息将按该顺序打印，每项一行。

注解

选项`--includedir-server`是 PostgreSQL 7.2 新增的。在之前的版本中，服务器头文件与客户端头文件安装在同一位置，可以通过选项 `--includedir`查询。要让你的软件包同时处理两种情况，可以先尝试较新的选项，然后测试退出状态以查看它是否成功。

在PostgreSQL 7.1 之前的版本中，`pg_config`出现之前，不存在查找等价配置信息的方法。

历史

`pg_config`实用程序最早出现于 PostgreSQL 7.1。

pg_dump

pg_dump — 把一个 PostgreSQL 数据库抽取到一个脚本文件或其他归档文件中

大纲

`pg_dump` [*`option`*...] [*`dbname`*]

描述

pg_dump 是一个用于备份 PostgreSQL 数据库的工具。即使在数据库被并发使用的情况下，它也能做出一致的备份。pg_dump 不会阻塞其他用户访问数据库（读或写）。

转储可以输出为脚本或归档文件格式。脚本文件是纯文本格式，
包含把数据库重建为保存时状态所需的 SQL 命令。要恢复这些脚本，可以使用psql。稍加修改后，它们甚至可以用于在其他机器和其他架构上重建数据库，甚至用于其他 SQL 数据库产品。

另一种归档文件格式用于配合pg_restore重建数据库，它们还允许 pg_restore 有选择地恢复部分内容，甚至在恢复之前对条目重新排序。归档文件的设计也使其可以跨架构移植。

与某种归档文件格式结合并配合 pg_restore 使用时，pg_dump 提供了一种灵活的归档和传输机制。可以用 pg_dump 备份整个数据库，然后用 pg_restore 检查归档和/或选择要恢复数据库的哪些部分。最灵活的输出文件格式是 “custom” 格式（`-Fc`）。
它允许选择和重新排序所有归档条目，并且默认进行压缩。tar 格式（`-Ft`）不压缩，而且在装载时无法重新排序数据，但除此之外相当灵活；此外，它可以用其他工具（如 `tar`）来处理。

在运行 pg_dump 时，应当检查输出中是否有任何警告（打印在标准错误上），尤其是考虑到下面列出的限制。

选项

下面的命令行选项用于控制输出格式。

`dbname`

指定要转储的数据库名称。如果未指定，则使用环境变量 `PGDATABASE`。如果未设置该变量，则使用为连接指定的用户名。

`-a`
`--data-only`

只转储数据，不转储模式（数据定义）。

此选项只对纯文本格式有意义。对于其他格式，可以在调用 `pg_restore` 时指定该选项。

`-b`
`--blobs`

在转储中包含大对象。

`-c`
`--clean`

输出在创建数据库对象（的命令）之前先清除（删除）它们的命令。

此选项只对纯文本格式有意义。对于其他格式，可以在调用 `pg_restore` 时指定该选项。

`-C`
`--create`

让输出以创建数据库本身并重新连接到新建数据库的命令开始。（使用这种形式的脚本时，在运行脚本之前连接到哪个数据库无关紧要。）

此选项只对纯文本格式有意义。对于其他格式，可以在调用 `pg_restore` 时指定该选项。

`-d`
`--inserts`

将数据转储为 `INSERT` 命令（而不是 `COPY`）。这会使恢复变得非常缓慢；它主要用于生成可装载到非 PostgreSQL 数据库的转储。注意，如果重新排列了列顺序，恢复可能会完全失败。`-D` 选项更安全，不过也更慢。

`-D`
`--column-inserts`
`--attribute-inserts`

将数据转储为带有显式列名的 `INSERT` 命令（`INSERT INTO` *`table`* `(`*`column`*`, ...) VALUES ...`）。这会使恢复变得非常缓慢；它主要用于生成可装载到非 PostgreSQL 数据库的转储。

`-f` *`file`*
`--file=`*`file`*

把输出发送到指定的文件。如果省略，则使用标准输出。

`-F` *`format`*
`--format=`*`format`*

选择输出的格式。*`format`*可以是下列之一：

`p`

输出纯文本 SQL 脚本文件（默认）

`t`

输出一个适合输入到 pg_restore 的 `tar` 归档。使用这种归档格式，允许在恢复数据库时重新排序和/或排除模式元素，还可以限制恢复时重新装载哪些数据。

`c`

输出一个适合输入到 pg_restore 的自定义归档。这是最灵活的格式，因为它既允许重新排序模式元素，也允许重新排序数据装载。这种格式默认还会压缩。

`-i`
`--ignore-version`

忽略 pg_dump 与数据库服务器之间的版本不匹配。

pg_dump 可以处理以前版本的 PostgreSQL 的数据库，但非常旧的版本不再受支持（目前是 7.0 之前）。如果你需要忽略版本检查，可以使用此选项（如果 pg_dump 因此而失败，可别怪没警告过你）。

`-n` *`namespace`*
`--schema=`*`schema`*

只转储 *`schema`* 的内容。如果未指定此选项，将转储目标数据库中的所有非系统模式。

> **注意**
>
> 在此模式下，pg_dump 不会尝试转储所选模式中的对象可能依赖的任何其他数据库对象。因此，
> 不能保证单模式转储的结果能够单独成功地恢复到一个干净的数据库中。

`-o`
`--oids`

为每个表转储对象标识符（OID）。如果你的应用程序以某种方式引用 OID 列（例如在外键约束中），请使用此选项。否则不应使用此选项。

`-O`
`--no-owner`

不输出用于把对象所有权设置成与原始数据库一致的命令。默认情况下，pg_dump 会发出 `SET SESSION AUTHORIZATION` 语句来设置所创建模式元素的所有权。除非脚本由超级用户（或拥有脚本中所有对象的同一用户）启动，否则这些语句会在运行脚本时报错。
若要创建一个可由任何用户恢复、但让该用户拥有所有对象的脚本，可以指定 `-O`。

此选项只对纯文本格式有意义。对于其他格式，可以在调用 `pg_restore` 时指定该选项。

`-R`
`--no-reconnect`

这个选项已经过时，但仍然被接受以保持向后兼容性。

`-s`
`--schema-only`

只转储模式（数据定义），不转储数据。

`-S` *`username`*
`--superuser=`*`username`*

指定在禁用触发器时要使用的超级用户名。这只在使用 `--disable-triggers` 时相关。（通常最好省略它，而是以超级用户身份运行生成的脚本。）

`-t` *`table`*
`--table=`*`table`*

只转储 *`table`* 的数据。在不同模式中可能存在多个同名的表；这种情况下，所有匹配的表都会被转储。要只选中一个表，需要同时指定 `--schema` 和 `--table`。

> **注意**
>
> 在此模式下，pg_dump 不会尝试转储所选表可能依赖的任何其他数据库对象。因此，不能保证单表转储的结果能够单独成功地恢复到一个干净的数据库中。

`-v`
`--verbose`

指定详细模式。这会使 pg_dump 把进度消息打印到标准错误。

`-x`
`--no-privileges`
`--no-acl`

不转储访问权限（grant/revoke 命令）。

`-X use-set-session-authorization`
`--use-set-session-authorization`

这个选项已经过时，但仍然被接受以保持向后兼容性。pg_dump 现在总是以此选项以前所选的方式工作。

`-X disable-triggers`
`--disable-triggers`

此选项只在创建只包含数据的转储时才相关。它指示 pg_dump 在输出中包含一些命令，以便在重新装载数据时临时禁用目标表上的触发器。
如果这些表上存在不希望在数据重新装载期间触发的引用完整性检查或其他触发器，
请使用此选项。

目前，为 `--disable-triggers` 输出的这些命令必须由超级用户执行。因此，还应通过 `-S` 指定一个超级用户名，或者更好的做法是谨慎地以超级用户身份运行生成的脚本。

此选项只对纯文本格式有意义。对于其他格式，可以在调用 `pg_restore` 时指定该选项。

`-Z` *`0..9`*
`--compress=`*`0..9`*

指定所用的压缩级别。零表示不压缩。对于自定义归档格式，
该选项指定对各个表数据段的压缩，默认以适中的级别压缩。对于纯文本输出，
设置非零压缩级别会压缩整个输出文件，就像将其传给 gzip 处理一样；但默认不压缩。tar 归档格式目前完全不支持压缩。

下面的命令行选项控制数据库连接参数。

`-h` *`host`*
`--host=`*`host`*

指定运行服务器的机器的主机名。如果该值以斜杠开头，则将其用作 Unix 域套接字的目录。默认值取自 `PGHOST` 环境变量（如果已设置）；否则会尝试使用 Unix 域套接字连接。

`-p` *`port`*
`--port=`*`port`*

指定服务器正在监听连接的 TCP 端口，或本地 Unix 域套接字文件扩展名。默认值取自 `PGPORT` 环境变量（如果已设置），否则使用编译时默认值。

`-U` *`username`*

以给定用户身份连接。

`-W`

强制提示输入密码。如果服务器要求密码认证，这一步本应自动发生。

环境

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数。

诊断

pg_dump 内部执行 `SELECT` 语句。如果在运行 pg_dump 时遇到问题，请确保你能够使用例如 psql从数据库中选择信息。

注意

如果你的数据库集簇对 `template1` 数据库有任何本地新增内容，请务必把 pg_dump 的输出恢复到一个真正空的数据库中；否则可能会因为新增对象的重复定义而出错。要创建一个没有任何本地新增内容的空数据库，应从 `template0` 而不是 `template1` 复制，例如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

pg_dump 有一些限制：

- 在转储单个表或以纯文本转储时，pg_dump 不处理大对象。必须使用某种非文本归档格式对整个数据库进行转储来转储大对象。
- 当选择只转储数据并使用 `--disable-triggers` 选项时，pg_dump 会在插入数据之前发出禁用用户表上触发器的命令，并在数据插入完成后发出重新启用它们的命令。
  如果恢复在中途中止，系统目录可能会处于错误状态。

tar 归档的成员大小限制为小于 8 GB。（这是 tar 文件格式的固有限制。）因此，如果某个表的文本表示超过该大小，就不能使用这种格式。tar 归档和其他任何输出格式的总大小不受限制（操作系统可能除外）。

恢复之后，明智的做法是对每个已恢复的表运行 `ANALYZE`，这样优化器才有可用的统计信息。

例子

要转储一个数据库：

```
$pg_dump mydb > db.out
```

要重新装载这个数据库：

```
$psql -d database -f db.out
```

要把一个包含大对象的名为 `mydb` 的数据库转储到一个 `tar` 文件：

```
$pg_dump -Ft -b mydb > db.tar
```

要把这个数据库（连同大对象）重新装载到一个已存在的名为 `newdb` 的数据库：

```
$pg_restore -d newdb db.tar
```

历史

pg_dump 工具最早出现在 Postgres95 发行版 0.02 中。非纯文本输出格式是在 PostgreSQL 发行版 7.1 中引入的。

另见

pg_dumpall, pg_restore, psql

pg_dumpall

pg_dumpall — 将一个PostgreSQL数据库集簇转储为脚本文件

大纲

`pg_dumpall` [*option*...]

描述

pg_dumpall是一个实用程序，用于写出（“转储”）
一个集簇中的所有PostgreSQL数据库到一个脚本文件中。该脚本文件包含SQL命令，可作为
psql的输入来恢复这些数据库。它通过对集簇中的每个数据库调用pg_dump来实现这一点。pg_
dumpall还会转储所有数据库共有的全局对象。（pg_dump不会保存这些对象。）
目前包括数据库用户和组，以及适用于整个数据库的访问权限的信息。

因此，pg_dumpall是一个用于备份数据库的一体化解决方案。
但要注意一个局限：它无法转储“大对象”，因为 pg_dump无法把这类对象转储到文本文件中。
如果你的数据库包含大对象，应当使用pg_dump的某种非文本输出模式来转储它们。

由于pg_dumpall会读取所有数据库中的表，因此通常必须以数据库超级用户身份连接，
才能生成完整的转储。另外，要执行保存的脚本，也需要具备超级用户权限，
这样才能添加用户和组并创建数据库。

SQL 脚本将写入标准输出。应使用 shell 操作符将其重定向到文件。

pg_dumpall 需要连接 PostgreSQL 服务器多次，每次都可能询问口令。这种情况下有一个 `$HOME/.pgpass` 文件会很方便。

选项

下列命令行选项用来控制输出的内容和格式。

`-a`
`--data-only`

只转储数据，不转储模式（数据定义）。

`-c`
`--clean`

包含用于在重新创建数据库之前清理（删除）数据库的 SQL 命令。

`-d`
`--inserts`

把数据转储为 `INSERT` 命令（而不是 `COPY`）。这会使恢复非常慢，但它让输出更容易移植到其他 SQL 数据库管理系统。

`-D`
`--column-inserts`
`--attribute-inserts`

把数据转储为带显式列名的 `INSERT` 命令（`INSERT INTO` *`table`* `(`*`column`*`, ...) VALUES ...`）。这会使恢复非常慢，但如果想重新排列列的顺序，这是必要的。

`-g`
`--globals-only`

只转储全局对象（用户和组），不转储数据库。

`-i`
`--ignore-version`

忽略pg_dumpall与数据库服务器之间的版本不匹配。

pg_dumpall可以处理PostgreSQL以前版本的数据库，但非常老的版本不再受支持（目前是7.0 之前）。如果需要跳过版本检查，可使用此选项（如果pg_dumpall随后失败，别怪我们没有提醒你）。

`-o`
`--oids`

转储每个表的对象标识符（OID）。如果你的应用以某种方式引用 OID 列（例如在外键约束中），请使用这个选项。否则不应使用这个选项。

`-s`
`--schema-only`

只转储模式（数据定义），不转储数据。

`-v`
`--verbose`

指定详细模式。这会使 pg_dumpall 向标准错误打印进度消息。

`-x`
`--no-privileges`
`--no-acl`

不转储访问权限（GRANT/REVOKE 命令）。

下列命令行选项控制数据库连接参数。

-h *host*

指定运行数据库服务器的机器的主机名。如果该值以斜杠开头，则将其用作 Unix 域套接字的目录。默认值取自 `PGHOST` 环境变量（如果已设置）；否则会尝试使用 Unix 域套接字连接。

-p *port*

指定服务器监听连接的 TCP 端口，或本地 Unix 域套接字文件扩展名。默认值取自`PGPORT`环境变量（如果已设置），否则使用编译时默认值。

-U *username*

以给定用户身份连接。

-W

强制提示输入密码。如果服务器要求密码认证，这应当会自动发生。

环境

`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

注解

由于pg_dumpall在内部调用pg_dump，因此某些诊断消息会提到pg_dump。

恢复完成后，最好对每个数据库运行`ANALYZE`，以便优化器获得有用的统计信息。也可以运行`vacuumdb -a -z`来分析所有数据库。

示例

要转储所有数据库：

```
$pg_dumpall > db.out
```

要从此文件重新装载数据库，可以使用：

```
$psql -f db.out template1
```

（这里连接哪个数据库并不重要，因为 pg_dumpall 创建的脚本文件会包含适当的命令，用于创建并连接到已保存的数据库。）

另见

有关可能出现的错误情况，请参见pg_dump。

pg_restore

pg_restore — 从由 pg_dump 创建的归档文件中恢复 PostgreSQL 数据库

大纲

`pg_restore` [*option*...] [*filename*]

描述

pg_restore 是一个用于从pg_dump以非纯文本格式创建的归档中恢复 PostgreSQL 数据库的工具。它会发出必要的命令，将数据库重建为保存时的状态。归档文件还允许 pg_restore 有选择地恢复其中的内容，甚至在恢复前重新排列各项的顺序。归档文件被设计为可跨体系结构移植。

pg_restore 可以以两种模式运行：如果指定了数据库名，归档会被直接恢复到该数据库中。（大对象只能通过这种直接数据库连接来恢复。）否则，会创建一个包含重建数据库所需 SQL 命令的脚本（并写入文件或标准输出），这与 pg_dump 纯文本格式创建的脚本类似。因此，一些控制脚本输出的选项与 pg_dump 的选项类似。

显然，pg_restore 无法恢复归档文件中不存在的信息。例如，如果归档是使用“将数据转储为 `INSERT` 命令”选项生成的，pg_restore 就不能使用 `COPY` 语句装载数据。

选项

pg_restore 接受下列命令行参数。

filename

指定要恢复的归档文件的位置。如果没有指定，则使用标准输入。

`-a`
`--data-only`

只恢复数据，不恢复模式（数据定义）。

`-c`
`--clean`

在重新创建数据库对象之前清理（删除）它们。

`-C`
`--create`

在恢复之前先创建数据库。（使用此选项时，由 `-d` 指定的数据库仅用于发出初始的 `CREATE DATABASE` 命令。所有数据都会恢复到归档中出现的数据库名中。）

`-d` *dbname*
`--dbname=`*dbname*

连接到数据库 *dbname*，并直接恢复到该数据库中。

`-f` *filename*
`--file=`*filename*

指定生成脚本的输出文件，或者在与 `-l` 一起使用时指定列表输出文件。默认为标准输出。

`-F format`
`--format=format`

指定归档的格式。没有必要指定格式，因为pg_restore会自动确定格式。如果指定，它可以是下列之一：

`t`

归档是一个 `tar` 归档。

`c`

归档是 pg_dump 的自定义格式。

`-i`
`--ignore-version`

忽略数据库版本检查。

`-I index`
`--index=index`

仅恢复指定名称的索引的定义。

`-l`
`--list`

列出归档的内容。此操作的输出可用作 `-L` 选项的输入。注意，如果把 `-n` 或 `-t` 这样的过滤开关与 `-l` 一起使用，它们会限制被列出的项。

`-L list-file`
`--use-list=list-file`

仅恢复在 *list-file* 中列出的归档元素，并按它们在文件中出现的顺序进行恢复。注意，如果把 `-n` 或 `-t` 这样的过滤开关与 `-L` 一起使用，它们还会进一步限制被恢复的项目。

list-file 通常是通过编辑先前 `-l` 操作的输出来创建的。可以移动或删除行，也可以通过在行首放置分号（`;`）将其注释掉。示例见下文。

`-N`
`--orig-order`

按 pg_dump 中最初生成各项的顺序恢复各项。这个选项没有已知的实际用途，因为 pg_dump 按对它方便的顺序生成各项，而这不太可能是恢复它们的安全顺序。（这不是各项最终在归档目录中列出的顺序。）另见 `-r`。

`-o`
`--oid-order`

按 OID 顺序恢复各项。这个选项的用处有限，因为 OID 只是原始创建顺序的一个近似指示。如果同时指定了 `-N`，本选项会覆盖它。另见 `-r`。

`-O`
`--no-owner`

不输出用于设置对象所有权以匹配原始数据库的命令。默认情况下，pg_restore 会发出 `SET SESSION AUTHORIZATION` 语句，为所创建的模式元素设置所有权。除非对数据库的初始连

接由超级用户发起（或者由脚本中所有对象的同一所有者发起），否则这些语句会失败。使用 -O 时，初始连接可以使用任意用户名，并且该用户将拥有所有创建的对象。

-P *function-name(argtype [, ...])*
--function=*function-name(argtype [, ...])*

仅恢复指定函数。请务必按转储文件目录中显示的形式准确拼写函数名和参数。

-r
--rearrange

按对象类型重排各项（如果给出了 -N 或 -o，则重排发生在其排序之后）。重排的目的是获得尽可能好的恢复性能。

当 -N、-o 和 -r 都未出现时，pg_restore 按各项在转储目录中出现的顺序恢复它们；如果给出了 -L，则按各项在 *list-file* 中出现的顺序。-o 加 -r 的组合复现了 pg_dump 在创建转储目录之前所做的排序，因此通常没有必要指定它。

-R
--no-reconnect

此选项已过时，但为了向后兼容仍然被接受。

-s
--schema-only

只恢复模式（数据定义），不恢复数据。序列值将被重置。

-S *username*
--superuser=*username*

指定在禁用触发器时要使用的超级用户用户名。仅当使用 --disable-triggers 时才相关。

-t *table*
--table=*table*

只恢复指定表的定义和/或数据。

-T *trigger*
--trigger=*trigger*

仅恢复指定名称的触发器。

-v
--verbose

指定详细模式。

-x
--no-privileges
--no-acl

阻止恢复访问权限（grant/revoke 命令）。

`-X use-set-session-authorization`
`--use-set-session-authorization`

这个选项已废弃，但为了向后兼容仍然接受。pg_restore 现在总是按以前由此选项选择的方式行事。

`-X disable-triggers`
`--disable-triggers`

这个选项只在执行仅数据恢复时相关。它指示 pg_restore 执行命令，在重新装入数据期间临时禁用目标表上的触发器。
如果你在表上有不希望在重新装入数据期间调用的参照完整性检查或其他触发器，
请使用此选项。

目前，为 `--disable-triggers` 发出的命令必须以超级用户身份执行。因此，你还应当用 `-S` 指定一个超级用户名，或者更好的是直接以 PostgreSQL 超级用户身份运行 pg_restore。

pg_restore还接受以下用于连接参数的命令行参数：

`-h` *`host`*
`--host=`*`host`*

指定服务器运行所在机器的主机名。如果该值以斜杠开头，则它被用作 Unix 域套接字目录。默认值取自 `PGHOST` 环境变量（如果已设置），否则会尝试使用 Unix 域套接字连接。

`-p` *`port`*
`--port=`*`port`*

指定服务器监听连接所使用的 TCP 端口或本地 Unix 域套接字文件扩展名。默认值取自 `PGPORT` 环境变量（如果已设置），否则使用编译时的默认值。

`-U` *`username`*

以给定用户身份连接。

`-W`

强制提示口令。如果服务器要求口令认证，这应当自动发生。

环境

`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

诊断

当使用 `-d` 选项指定直接数据库连接时，pg_restore 会在内部执行 SQL 语句。如果运行 pg_restore 时遇到问题，请确保你能够使用例如 psql 从该数据库中查询信息。

注解

如果你的安装在 `template1` 数据库中有任何本地添加内容，请务必将 pg_restore 的输出装载到一个真正空的数据库中；否则很可能会因为这些附加对象的重复定义而报错。

要创建一个不带任何本地添加内容的空数据库，应从 `template0` 而不是 `template1` 复制，例如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

pg_restore 的局限性详述如下。

- 当把数据恢复到一个预先存在的表中，并且使用了 `--disable-triggers` 选项时，pg_restore 会在插入数据前发出命令，禁用用户表上的触发器，然后在数据插入后再发出命令重新启用它们。如果恢复在中途停止，系统目录可能会处于错误状态。
- pg_restore 不会为单个表恢复大对象。如果归档中包含大对象，那么所有大对象都将被恢复。

关于 pg_dump 的局限性的细节也可参见pg_dump文档。

恢复完成后，最好对每个恢复的表运行 `ANALYZE`，以便优化器获得有用的统计信息。

示例

要把一个名为 `mydb` 的包含大对象的数据库转储到一个 `tar` 文件中：

```
$pg_dump -Ft -b mydb > db.tar
```

要把这个数据库（连同大对象）重新装载到一个已存在的名为 `newdb` 的数据库中：

```
$pg_restore -d newdb db.tar
```

要重新排列数据库项的顺序，首先需要转储归档的目录：

```
$pg_restore -l archive.file > archive.list
```

列表文件由一个头部和每个项各占一行的内容组成，例如：

```
;
; Archive created at Fri Jul 28 22:28:36 2000
;     dbname: birds
;     TOC Entries: 74
;     Compression: 0
;     Dump Version: 1.4-0
;     Format: CUSTOM
;
;
; Selected TOC Entries:
;
2; 145344 TABLE species postgres
3; 145344 ACL species
4; 145359 TABLE nt_header postgres
5; 145359 ACL nt_header
6; 145402 TABLE species_records postgres
7; 145402 ACL species_records
8; 145416 TABLE ss_old postgres
9; 145416 ACL ss_old
10; 145433 TABLE map_resolutions postgres
11; 145433 ACL map_resolutions
12; 145443 TABLE hs_old postgres
13; 145443 ACL hs_old
```

分号表示一条注释的开始，而行首的数字表示分配给每个项的内部归档 ID。

文件中的行可以被注释掉、删除并重新排序。例如：

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

可以将这样的文件作为 pg_restore 的输入，这样它就只会按该顺序恢复项 10 和 6：

```
$pg_restore -L archive.list archive.file
```

历史

pg_restore 工具最早出现在 PostgreSQL 7.1 中。

参见

pg_dump, pg_dumpall, psql

pgtclsh

pgtclsh — PostgreSQL Tcl shell 客户端

大纲

`pgtclsh` [*`filename`* [*`argument`*...]]

描述

`pgtclsh` 是一个用 PostgreSQL 数据库访问函数扩展的 Tcl shell 接口。（本质上，它就是装载了 `libpgtcl` 的 `tclsh`。）与常规的 Tcl shell 一样，第一个命令行参数是一个脚本文件，其余的参数会被传递给该脚本。如果没有指定脚本文件，则 shell 是交互式的。

一个带 Tk 和 PostgreSQL 函数的 Tcl shell 以pgtksh提供。

另见

pgtksh, 第 29 章（`libpgtcl` 的说明）, tclsh

pgtksh

pgtksh — PostgreSQL Tcl/Tk shell 客户端

大纲

`pgtksh` [*`filename`* [*`argument`*...]]

描述

`pgtksh` 是一个 Tcl/Tk shell 接口，它以 PostgreSQL数据库访问函数进行了扩展。（本质上，它就是加载了 `libpgtcl` 的 `wish`。）与 `wish`（常规的 Tcl/Tk shell）一样，第一个命令行参数是一个脚本文件，其余的任何参数都会被传递给该脚本。特殊选项可以改由 X Window System 库处理。如果没有指定脚本文件，shell 将以交互方式运行。

一个普通的 Tcl shell，带有 PostgreSQL 函数，可通过pgtclsh使用。

参见

pgtclsh, 第 29 章（`libpgtcl` 的说明）, tclsh , wish

psql

psql — PostgreSQL 的交互式终端

大纲

`psql` [*option*...] [*dbname* [*username*]]

描述

psql是PostgreSQL的一个基于终端的前端。它使你能够交互式地输入查询，将其发送给PostgreSQL，并查看查询结果。也可以从文件提供输入。此外，
它还提供了若干元命令和多种类似 shell 的特性，以便于编写脚本和自动化执行各种任务。

选项

`-a`
`--echo-all`

在读入时把所有行打印到屏幕。这对脚本处理比交互模式更有用。这等效于把变量 `ECHO` 设置为 `all`。

`-A`
`--no-align`

切换到非对齐输出模式（默认输出模式是对齐的）。

`-c` *command*
`--command` *command*

指定 psql 执行一个命令字符串 command，然后退出。这在 shell 脚本中很有用。

command 必须是一个服务器完全可解析的命令字符串（即不包含 psql 专有的特性），或者是单个反斜线命令。因此不能用这个选项混合 SQL 和 psql 元命令。要那样做，可以把字符串用管道输送到 psql 中，像这样：`echo "\x \\ select * from foo;" | psql`.

如果命令字符串包含多条 SQL 命令，它们会在单个事务中处理，除非字符串中包含显式的 BEGIN/COMMIT 命令将其分成多个事务。这与把同一字符串送入psql标准输入时的行为不同。

`-d` *dbname*
`--dbname` *dbname*

指定要连接的数据库的名称。这等效于在命令行上把 dbname 指定为第一个非选项参数。

`-e`
`--echo-queries`

显示发送到服务器的所有命令。这等效于把变量 `ECHO` 设置为 `queries`。

`-E`
`--echo-hidden`

回显 `\d` 及其他反斜线命令生成的实际查询。如果你想在自己的程序中包含类似的功能，可以使用它。这等效于在 psql 内部设置变量 `ECHO_HIDDEN`。

`-f` *`filename`*
`--file` *`filename`*

使用文件*`filename`*作为命令来源，而不是交互式地读取命令。文件处理完毕后，psql终止。这在很多方面等价于元命令`\i`。

如果*`filename`*是`-`（连字符），则会读取标准输入。

使用这个选项与写成`psql < filename`有细微差别。通常两种形式都会得到你期望的结果，但使用`-f`可以启用一些有用的特性，例如带行号的错误消息。
使用这个选项也还有一点机会降低启动开销。另一方面，使用 shell 输入重定向的形式在理论上能保证得到与你手工逐行输入时完全相同的输出。

`-F` *`separator`*
`--field-separator` *`separator`*

使用*`separator`*作为字段分隔符。这等效于`\pset fieldsep`或者`\f`。

`-h` *`hostname`*
`--host` *`hostname`*

指定运行服务器的机器的主机名。如果该值以斜线开头，则它会被用作 Unix 域套接字所在的目录。

`-H`
`--html`

切换到HTML表格输出模式。这等效于`\pset format html`或者`\H`命令。

`-l`
`--list`

列出所有可用的数据库，然后退出。其他非连接选项会被忽略。这类似于内部命令 `\list`。

`-n`
`--no-readline`

不使用 readline 进行行编辑，也不使用历史记录。这在剪切粘贴时关闭制表符展开会很有用。

`-o` *`filename`*
`--output` *`filename`*

把所有查询输出放到文件*`filename`*中。这等效于命令`\o`。

`-p` *`port`*
`--port` *`port`*

指定服务器用于监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是 PGPORT 环境变量的值，如果没有设置，则默认为编译时指定的端口，通常是 5432。

`-P` *`assignment`*
`--pset` *`assignment`*

以 `\pset` 的形式指定打印选项。注意，这里必须用一个等号而不是空格来分隔名称和值。例如，要把输出格式设置为 LaTeX，可以写 `-P format=latex`。

`-q`
`--quiet`

指定 psql 安静地工作。默认情况下，它会打印欢迎消息和各种提示信息。如果使用了这个选项，以上那些就都不会输出。这与 -c 选项配合很有用。在 psql 内部你也可以设置 QUIET 变量来达到同样的效果。

`-R` *separator*
`--record-separator` *separator*

把*separator*用作记录分隔符。这等效于`\pset recordsep`命令。

`-s`
`--single-step`

运行在单步模式中。这意味着在每个命令被发送给服务器之前都会提示用户，并允许取消执行。使用这个选项可以调试脚本。

`-S`
`--single-line`

运行在单行模式中，其中换行符会终止一个 SQL 命令，就像分号的作用一样。

> **注意**
>
> 这种模式是为坚持使用它的用户提供的，但并不一定值得推荐。特别是，如果在一行中混合了SQL和元命令，对于没有经验的用户来说，它们的执行顺序未必总是清楚的。

`-t`
`--tuples-only`

关闭列名和结果行计数页脚等的打印。它完全等效于 \t 元命令。

`-T` *table_options*
`--table-attr` *table_options*

指定要放在HTML `table`标签内的选项。详见`\pset`。

`-u`

让 psql 在连接数据库之前提示输入用户名和密码。

这个选项已被弃用，因为它在概念上有缺陷。（提示输入非默认用户名，与因为服务器要求而提示输入密码，实际上是两件不同的事情。）建议你改用`-U`和`-W`选项。

`-U` *username*
`--username` *username*

作为用户*username*而不是默认用户连接到数据库（当然，你必须具有这样做的权限）。

`-v` *assignment*
`--set` *assignment*
`--variable` *assignment*

执行一次变量赋值，和`\set`元命令相似。
注意你必须在命令行上用等号分隔名字和值（如果有）。要取消变量的设置，

去掉等号就行。要把一个变量设为空字符串，使用等号但是去掉值。
这些赋值在启动的非常早期阶段完成，因此为内部目的保留的变量可能会在稍后被覆盖。

`-V`
`--version`

显示 psql 的版本。

`-W`
`--password`

要求 psql 在连接数据库之前提示输入密码。即使你随后用元命令 \connect 改变了数据库连接，这一点在整个会话期间也保持有效。

在当前版本中，只要服务器请求密码认证，psql 就会自动发出密码提示。由于这目前是取巧实现的，自动识别可能莫名失效，因此提供此选项来强制提示。
如果没有发出密码提示而服务器又要求密码认证，连接尝试将失败。

`-x`
`--expanded`

打开扩展表格式模式。这等效于`\x`。

`-X`,
`--no-psqlrc`

不读取启动文件 `~/.psqlrc`。

`-?`
`--help`

显示有关 psql 命令行参数的帮助。

退出状态

如果psql正常结束，它会向 shell 返回 0；如果它自身发生致命错误（例如内存耗尽、找不到文件），则返回 1；如果到服务器的连接发生故障且该会话不是交互式的，则返回 2；如果脚本中发生错误且变量`ON_ERROR_STOP`已设置，则返回 3。

用法

连接到数据库

psql 是一个常规的 PostgreSQL 客户端应用。要连接到数据库，你需要知道目标数据库的名称、服务器的主机名和端口号，以及你想以哪个用户名连接。可以通过命令行选项把这些参数告诉 psql，分别对应 `-d`、`-h`、`-p` 和 `-U`。如果发现一个不属于任何选项的参数，它会被解释为数据库名（如果数据库名也已给出，则解释为用户名）。这些选项并非都是必需的，可以使用默认值。
如果省略主机名，psql 将通过 Unix 域套接字连接到本地主机上的服务器。默认端口号在编译时确定。由于数据库服务器使用相同的默认值，因此在大多数情况下不必指定端口。
默认用户名是你的 Unix 用户名，默认数据库名也是如此。请注意，你不能随意以任意用户名连接到任意数据库。数据库管理员应当已经告知你拥有的访问权限。为了少打一些字，
你还可以把环境变量 PGDATABASE、PGHOST、PGPORT 和 PGUSER 设置为适当的值。

如果由于任何原因（例如权限不足、服务器没有在目标主机上运行等）导致连接无法建立，
psql将返回一个错误并且终止。

输入 SQL 命令

在正常操作时，psql 提供一个提示符，它是 psql 当前连接到的数据库名称后跟字符串 =>。例如：

```
$ psql testdb
Welcome to psql 7.4.30, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=>
```

在提示符下，用户可以输入 SQL 命令。通常，当遇到表示命令结束的分号时，输入行被发送到服务器。行尾并不会终止一条命令。因此为了清晰，命令可以分布在多行上。
如果命令被发送且没有错误，命令的结果会显示在屏幕上。

每当执行命令时，psql也会轮询由LISTEN和NOTIFY生成的异步通知事件。

元命令

你输入到 psql 中的任何以未加引号的反斜线开始的内容都是一个由 psql 自身处理的 psql 元命令。这些命令正是让 psql 在管理和编写脚本方面很有用的原因。元命令更常被称为斜线或反斜线命令。

psql命令的格式是用反斜线后面直接跟上一个命令动词，然后是一些参数。
参数与命令动词和其他参数之间用任意多个空白字符分隔开。

要在参数中包含空白，可以用单引号将它括起来。要在这样的参数中包含一个单引号，
可以在它前面放一个反斜线。单引号中的内容还会接受类似 C 语言的替换：`\n`（换行）、`\t`（制表符）、`\digits`、`\0digits`和 `\0xdigits`（分别表示以十进制、八进制或十六进制给出的码点对应的字符）。

如果未加引号的参数以冒号（:）开头，它会被当作一个 psql 变量，该变量的值将被用作参数。

用反引号（`）包围的参数会被当作传给 shell 的命令行。该命令的输出（去掉末尾的换行符）将被用作参数值。上述转义序列在反引号中同样适用。

有些命令把SQL标识符（例如表名）作为参数。
这些参数遵循SQL的语法规则：未加引号的字母会被强制转换为小写，而双引号（"）
可以保护字母不发生大小写转换，并允许在标识符中包含空白。在双引号内，
成对的双引号会在结果名称中折叠成一个双引号。例如，`FOO"BAR"BAZ`会被解释为`foo BARbaz`，而`"A weird"" name"`会变成`A weird" name`。

参数解析会在行尾或遇到另一个未加引号的反斜线时停止。
未加引号的反斜线会被视为新元命令的开始。特殊序列`\`（两个反斜线）表示参数结束，
并继续解析SQL命令（如果还有）。通过这种方式，
SQL命令和psql命令可以自由地混合在同一行中。但无论如何，
元命令的参数都不能延续到下一行。

定义了下列元命令：

`\a`

如果当前表格输出格式是非对齐，则切换为对齐；否则切换为非对齐。
保留此命令是为了向后兼容。更通用的解决方案参见 \pset。

`\cd` [*directory*]

将当前工作目录更改为*directory*。如果没有参数，则切换到当前用户的主目录。

> **提示**
>
> 要打印当前工作目录，请使用`\! pwd`。

`\C` [*title*]

设置作为查询结果打印的表的标题，或取消此类标题。该命令等价于`\pset title` *title*。（此命令的名称源自“caption”，因为它过去只用于设置HTML表的标题。）

`\connect` (or `\c`) [*dbname* [*username*]]

建立到一个新数据库和/或以另一个用户名的连接。先前的连接会被关闭。如果*dbname*为`-`，则假定使用当前的数据库名。

如果省略*username*，则假定使用当前的用户名。

作为一条特殊规则，不带任何参数的`\connect`
将以默认用户连接到默认数据库（就像你启动psql时不带任何参数一样）。

如果连接尝试失败（用户名错误、访问被拒绝等），当且仅当psql处于交互模式时，
才会保留先前的连接。在执行非交互式脚本时，处理会立即停止并报错。
选择这种区别对待，一方面是为了让用户方便地应对输入错误，
另一方面是作为一种安全机制，防止脚本意外地作用于错误的数据库。

`\copy` *table* [(*column_list*)] { from | to } *filename* | stdin | stdout [with] [oids] [delimiter [as] '*character*'] [null [as] '*string*']

执行前端（客户端）复制。此操作执行的是一条 SQL COPY命令，但不是由服务器读取或写入指定的文件，而是由 psql 读取或写入该文件，并在服务器和本地文件系统之间传送数据。这意味着文件的可访问性和权限是本地用户的，而不是服务器的，并且不需要 SQL 超级用户权限。

该命令的语法类似于 SQL COPY 命令。（细节见其描述。）注意，因此 \copy 命令适用特殊的解析规则。特别是，变量替换规则和反斜线转义不适用。

> **提示**
>
> 此操作不如 SQL COPY 命令高效，因为所有数据都必须通过客户端/服务器连接。对于大量数据，另一种技术可能更可取。

> **注意**
>
> 注意客户端副本与服务器副本对 `stdin` 和 `stdout` 的解释不同：在客户端副本中它们总是指 psql 的输入和输出流。在服务器副本中，`stdin` 来自 `COPY` 本身来自的地方

（例如用 `-f` 选项运行的一个脚本），而 `stdout` 指查询输出流（见下面的 `\o` 元命令）。

`\copyright`

显示 PostgreSQL 的版权和分发条款。

`\d[pattern]`

对每个匹配 pattern 的关系（表、视图、索引或序列），显示所有列、它们的类型以及任何特殊属性（如 NOT NULL 或默认值，如果有）。相关的索引、约束、规则和触发器也会显示，如果该关系是视图则还显示视图定义。(“Matching the pattern” is defined below.)

命令形式 \d+ 相同，但还会显示与表列关联的任何注释。

> **注意**
>
> 如果 \d 不带 pattern 参数使用，它等价于 \dtvs，后者会显示所有表、视图和序列的列表。这纯粹是为了方便。

`\da[pattern]`

列出所有可用的聚合函数及它们所操作的数据类型。如果指定了*pattern*，则只显示名称匹配该模式的聚合函数。

`\dc[pattern]`

列出所有可用的字符集编码之间的转换。如果指定了*pattern*，则只列出名称匹配该模式的转换。

`\dC`

列出所有可用的类型转换。

`\dd[pattern]`

显示匹配 pattern 的对象的描述，如果没有给出参数则显示所有可见对象的描述。但无论哪种情况，都只列出有描述的对象。(“Object” covers aggregates, functions, operators, types, relations (tables, views, indexes, sequences, large objects), rules, and triggers.) For example:

```
=> \dd version
                      Object descriptions
   Schema   |  Name   |  Object  |        Description
------------+---------+----------+---------------------------
 pg_catalog | version | function | PostgreSQL version string
(1 row)
```

对象的描述可以用 `COMMENT` SQL 命令创建。

`\dD[pattern]`

列出所有可用的域。如果指定了*pattern*，则只显示匹配该模式的域。

`\df [ pattern ]`

列出可用的函数及其参数和返回类型。If *pattern* is specified, only functions whose names match the pattern are shown. If the form `\df+` is used, additional information about each function, including language and description, is shown.

> **注意**
>
> 为了减少杂乱，`\df` 不显示数据类型 I/O 函数。这是通过忽略接受或返回 `cstring` 类型的函数实现的。

`\distvS [ pattern ]`

这不是实际的命令名：字母 i、s、t、v、S 分别代表索引、序列、表、视图和系统表。你可以按任意顺序指定其中任意个或全部字母，以获得所有匹配对象的列表。字母 S 把列表限制为系统对象；不带 S 时只显示非系统对象。如果向命令名追加 +，每个对象还会列出其关联的描述（如果有）。

如果指定了 pattern，则只列出名称匹配该模式的对象。

`\dl`

这是`\lo_list`的别名，用于显示大对象列表。

`\dn [ pattern ]`

列出所有可用的模式（命名空间）。如果指定了 *pattern*（正则表达式），则只列出名称匹配该模式的模式。

`\do [ pattern ]`

列出可用的操作符及其操作数和返回类型。如果指定了*pattern*，
则只列出名称匹配该模式的操作符。

`\dp [ pattern ]`

产生所有可用表及其相关访问权限的列表。If *pattern* is specified, only tables whose names match the pattern are listed.

访问权限用GRANT和REVOKE命令设置。更多信息见GRANT。

`\dT [ pattern ]`

列出所有数据类型，或者只列出匹配*pattern*的类型。命令形式`\dT+`显示额外的信息。

`\du [ pattern ]`

列出所有数据库用户，或者只列出匹配*pattern*的用户。

`\edit` (or `\e`) [*filename*]

如果指定了 *filename*，就编辑该文件；编辑器退出后，其内容会被复制回查询缓冲区。如果没有给出参数，则把当前查询缓冲区复制到一个临时文件，然后以同样的方式编辑。

新的查询缓冲区随后按照 psql 的正常规则重新解析，即把整个缓冲区当作单行对待。(Thus you cannot make scripts this way. Use `\i` for that.) This means also that if the query

ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer.

> **提示**
>
> psql 依次搜索环境变量 PSQL_EDITOR、EDITOR 和 VISUAL 来确定要使用的编辑器。如果它们都未设置，则运行 /bin/vi。

`\echo` *`text`* [...]

把参数打印到标准输出，参数之间以一个空格分隔，末尾跟一个换行符。这在脚本的输出中穿插信息时很有用。例如：

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

如果第一个参数是未加引号的 -n，则不写末尾的换行符。

> **提示**
>
> 如果你使用 \o 命令重定向了查询输出，可能希望用 \qecho 代替此命令。

`\encoding` [*`encoding`*]

设置客户端字符集编码。没有参数时，此命令显示当前编码。

`\f` [*`string`*]

设置非对齐查询输出的字段分隔符。默认值是竖线（`|`）。另请参见 `\pset`，那里介绍了设置输出选项的通用方法。

`\g` [{ *`filename`* | `|` *`command`* }]

把当前查询输入缓冲区发送到服务器，并可选地把输出保存到 filename 中，或把输出通过管道送到一个单独的 Unix shell 去执行 command。A bare `\g` is virtually equivalent to a semicolon. A `\g` with argument is a “one-shot” alternative to the `\o` command.

`\help` (or `\h`) [*`command`*]

给出指定SQL命令的语法帮助。如果未指定*`command`*，则psql将列出所有可用语法帮助的命令。如果*`command`*是星号（`*`），则显示所有SQL命令的语法帮助。

> **注意**
>
> 为了简化输入，由多个单词组成的命令不需要加引号。因此，可以直接输入**`\help alter table`**。

`\H`

打开HTML查询输出格式。如果HTML格式已经打开，则切换回默认的对齐文本格式。此命令是为兼容性和便利性而保留的；设置其他输出选项的方法见`\pset`。

`\i`*`filename`*

从文件*`filename`*中读取输入，并像在键盘上输入一样执行它。

> **注意**
>
> 如果想在屏幕上看到被读入的各行，请将变量`ECHO`设置为`all`。

`\l` (or `\list`)

列出服务器中所有数据库的名称、所有者和字符集编码。向命令名追加 `+` 还可以看到数据库的任何描述。

`\lo_export`*`loid filename`*

从数据库中读取具有OID*`loid`*的大对象，并将其写入*`filename`*。请注意，这与服务器函数`lo_export`略有不同，后者使用运行数据库服务器的用户的权限，
并在服务器的文件系统上操作。

> **提示**
>
> 使用`\lo_list`命令来查找大对象的OID。

`\lo_import`*`filename`* [*`comment`*]

把该文件存储为一个 PostgreSQL 大对象。可以选择把给定的注释与该对象关联。例如：

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

该响应表明这个大对象得到的对象 ID 是 152801，如果以后想再访问该对象，就应当记住这个 ID。因此，建议始终为每个对象关联一段人类可读的注释。之后可以用 `\lo_list` 命令查看这些注释。

注意这条命令与服务器端的`lo_import`有微妙的差别，因为它以本地用户在本地文件系统上操作，而不是服务器的用户和文件系统。

`\lo_list`

列出当前存储在数据库中的所有PostgreSQL大对象，以及为它们提供的注释。

`\lo_unlink`*`loid`*

从数据库中删除OID 为 *`loid`*的大对象。

> **提示**
>
> 使用`\lo_list`命令来查找大对象的OID。

`\o` [{*`filename`* | `|`*`command`*}]

把以后的查询结果保存到文件*`filename`*中，或通过管道把以后的结果送给单独的 Unix shell 去执行*`command`*。如果未指定参数，查询输出将重置为标准输出。

“查询结果”包括从数据库服务器获得的所有表、命令响应和提示，
以及各种查询数据库的反斜线命令（例如`\d`）的输出，但不包括错误消息。

> **提示**
>
> 要在查询结果之间穿插文本输出，可使用 `\qecho`。

`\p`

将当前查询缓冲区打印到标准输出。

`\pset`*`parameter`* [*`value`*]

此命令设置影响查询结果表输出的选项。parameter 描述要设置哪个选项。The semantics of *`value`* depend thereon.

可调整的打印选项有：

`format`

把输出格式设置为 `unaligned`、`aligned`、`html` 或 `latex` 之一。允许唯一的缩写。（也就是说一个字母就足够了。）

“非对齐”把一行的所有列写在一行上，以当前有效的字段分隔符分隔。
这是为了创建可能要被其他程序读入的输出（制表符分隔、逗号分隔）。“对齐”
模式是默认的标准、人类可读、排版良好的文本输出。“HTML”和“LaTeX”
模式输出的表格用于包含在使用相应标记语言的文档中。它们不是完整的文档！（在
HTML 中这或许不那么严重，但在 LaTeX 中你必须有完整的文档包装。）

`border`

第二个参数必须是一个数字。一般来说，数字越大表格的边框和线越多，
但这取决于具体的格式。In HTML mode, this will translate directly into the `border=...` attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.

`expanded` (or `x`)

在常规格式和扩展格式之间切换。启用扩展格式后，所有输出都有两列，列名在左、
数据在右。如果数据在正常的“水平”模式下放不到屏幕上，这个模式很有用。

所有四种输出格式都支持扩展模式。

`null`

第二个参数是一个每当列为空值时应打印的字符串。默认是完全不打印任何内容，
这很容易被误认为例如空字符串。因此，可以选择写 \pset null '(null)'。

`fieldsep`

指定在非对齐输出模式中使用的字段分隔符。
这样就可以创建例如其他程序可能更喜欢的制表符或逗号分隔的输出。
要把制表符设置为字段分隔符，输入 \pset fieldsep '\t'。默认的字段分隔符是 '|'（竖线）。

`footer`

切换默认页脚（x 行）的显示。

`recordsep`

指定在非对齐输出模式中使用的记录（行）分隔符。默认是换行符。

`tuples_only` (or `t`)

在只显示元组和完整显示之间切换。完整显示可能显示额外信息，如列标题、标题和各种页脚。In tuples only mode, only actual table data is shown.

`title` [*text*]

为随后打印的任何表设置表标题。这可用于为输出提供描述性标签。如果不带参数，则取消标题。

`tableattr` (or `T`) [*text*]

允许你指定要放在 HTML table 标签内的任何属性。例如可以是 cellpadding 或 bgcolor。注意这里可能不应该指定 border，因为那已由 \pset border 处理。

`pager`

控制查询和 psql 帮助输出对分页器的使用。如果设置了环境变量 PAGER，输出将通过管道送到指定的程序。否则使用依赖于平台的默认值（如 more）。

当 pager 为 off 时，不使用分页器。当 pager 为 on 时，只在合适的情况下使用分页器，即输出到终端且放不到屏幕上时。（psql 对何时使用分页器的估计并不完美。）\pset pager 打开和关闭分页器。pager 也可以设置为 always，这会使分页器总是被使用。

这些不同格式的外观的示例见示例一节。

提示

\pset 有多种快捷命令。参见 \a、\C、\H、\t、\T 和 \x。

注意

不带参数调用 \pset 是一个错误。将来这个调用可能会显示所有打印选项的当前状态。

`\q`

退出psql程序。

`\qecho` *text* [...]

此命令与 \echo 相同，只是所有输出将写入由 \o 设置的查询输出通道。

`\r`

重置（清空）查询缓冲区。

`\s` [*filename*]

打印命令行历史或把它保存到 filename。如果省略 filename，则历史写到标准输出。只有 psql 被配置为使用 GNU history 库时此选项才可用。

> **注意**
>
> 在当前版本中，不再需要保存命令历史，因为程序终止时会自动完成。每次 psql 启动时也会自动加载历史。

`\set` [*name* [*value* [...]]]

把内部变量 name 设置为 value，如果给出多个值则设置为它们的串接。如果没有给出第二个参数，该变量只是被设置为没有值。要取消设置一个变量，使用 \unset 命令。

有效的变量名可以包含字符、数字和下划线。详见变量一节。

尽管你可以随意把任何变量设成任何值，psql 还是把若干变量视为特殊的。它们在变量一节中记录。

> **注意**
>
> 这条命令与 SQL 命令SET完全无关。

`\t`

切换输出中的列名标题和行数页脚的显示状态。这个命令等价于`\pset tuples_only`，提供它是为了使用方便。

`\T`*table_options*

允许你指定要放在 HTML 表输出模式的 table 标签中的属性。此命令等价于 \pset tableattr table_options。

`\timing`

切换以毫秒为单位显示每条 SQL 语句执行耗时的开关状态。

`\w` {*filename* | | *command*}

把当前查询缓冲区输出到文件*filename*，或通过管道送给 Unix 命令*command*。

`\x`

切换扩展表格式模式。它等价于`\pset expanded`。

`\z` [*pattern*]

产生所有可用表及其相关访问权限的列表。如果指定了 *pattern*，则只列出名称匹配该模式的表。

访问权限用GRANT和REVOKE命令设置。更多信息见GRANT。

这是 `\dp`（“显示权限”）的别名。

\![*command*]

进入一个单独的 shell，或执行 shell 命令 *command*。参数不会被进一步解释；shell 会原样看到它们。

\?

显示有关反斜线命令的帮助信息。

很多\d命令都可以用一个*pattern*参数来指定要被显示的对象名称。*表示“任意字符序列”，?表示“任意单个字符”（这种记号方法与 Unix shell 的文件名模式相当）。高级用户还可以使用字符类等正则表达式记法，例如用[0-9]匹配“任意数字”。
要让这些模式匹配字符中的任何一个按字面意思解释，可以用双引号把它括起来。

包含（未加引号的）点号的模式会被解释为模式名称的匹配模式，后接对象名称的匹配模式。
例如，\dt foo*.bar*显示模式名以foo开头且表名以bar开始的所有表。如果没有点号，则该匹配模式只匹配当前模式搜索路径中可见的对象。

每当完全省略*pattern*参数时，\d命令会显示当前模式搜索路径中可见的所有对象。
要查看数据库中的所有对象，可使用模式*.*。

高级特性

变量

psql 提供类似于常见 Unix 命令 shell 的变量替换特性。变量只是名称/值对，其中值可以是任意长度的任意字符串。要设置变量，使用 psql 元命令 \set：

```
testdb=> \set foo bar
```

把变量 foo 设为值 bar。要取得变量的内容，在名称前加上冒号，并把它用作任何斜线命令的参数：

```
testdb=> \echo :foo
bar
```

> **注意**
>
> \set 的参数遵循与其他命令相同的替换规则。因此你可以构造有趣的引用，例如 \set :foo 'something' 并分别得到 Perl 或 PHP 中著名的“软链接”或“可变变量”。不幸的是（或者幸运的是？），这些构造没有任何实际用处。另一方面，\set bar :foo 是一种完全合法的复制变量的方法。

如果调用 \set 时没有第二个参数，该变量会被设置，其值为空字符串。要取消设置（即删除）一个变量，使用命令 \unset。

psql 的内部变量名可以由字母、数字和下划线按任意顺序、任意数量组成。其中一些变量会被 psql 特殊对待。它们表示特定的选项设置（运行时可以通过更改该变量的值来改变），或者表示应用的某种状态。尽管你可以把这些变量用于其他目的，但不建议这样做，
因为程序的行为可能会很快变得非常奇怪。按照惯例，
所有被特殊对待的变量都由全大写字母（以及可能的数字和下划线）组成。
为了确保将来最大的兼容性，请避免把这类变量名用于自己的目的。
下面是所有被特殊对待的变量的列表。

`AUTOCOMMIT`

当为 on（默认）时，每条 SQL 命令在成功完成后自动提交。要在此模式下推迟提交，必须输入 BEGIN 或 START TRANSACTION SQL 命令。当为 off 或未设置时，SQL 命令在你显式发出 COMMIT 或 END 之前不会提交。自动提交关闭模式的工作方式是：在任何尚不在事务块中且本身不是 BEGIN 或其他事务控制命令的命令之前，为你隐式发出一个 BEGIN。

> **注意**
>
> 在自动提交关闭模式中，必须通过`ABORT`或者`ROLLBACK`
> 显式地放弃任何失败的事务。还要记住，如果退出会话时没有提交，
> 则所有的工作都会丢失。

> **注意**
>
> 自动提交开启模式是 PostgreSQL 的传统行为，而自动提交关闭更接近 SQL 规范。如果你更喜欢自动提交关闭，可能希望在你的 .psqlrc 文件中设置它。

`DBNAME`

当前已连接的数据库名称。每次连接到一个数据库时都会设置该变量（包括程序启动时），但可以被取消设置。

`ECHO`

如果设置为 all，所有输入的或来自脚本的行在解析或执行之前都被写到标准输出。要在程序启动时选择此行为，使用开关 -a。If set to `queries`, psql merely prints all queries as they are sent to the server. The switch for this is `-e`.

`ECHO_HIDDEN`

设置了此变量后，当反斜线命令查询数据库时会先显示该查询。这样你就可以研究 PostgreSQL 的内部实现，并在自己的程序中提供类似的功能。（要在程序启动时选择此行为，使用开关 -E。）如果你把该变量设置为 noexec，查询只被显示而不会真正发送到服务器执行。

`ENCODING`

当前的客户端字符集编码。

`HISTCONTROL`

如果这个变量被设置为`ignorespace`，则以一个空格开始的行不会被放入到历史列表中。
如果被设置为值`ignoredups`，则与上一条历史记录相同的行不会被放入。值`ignoreboth`组合了上述两种值。如果未设置，或被设置为上述值以外的其他值，
所有在交互模式中被读入的行都会保存在历史列表中。

> **注意**
>
> 这个特性是可耻地从Bash抄袭过来的。

`HISTSIZE`

存储在命令历史中的命令数量。默认值是 500。

> **注意**
>
> 这个特性是可耻地从Bash抄袭过来的。

`HOST`

当前连接到的数据库服务器主机。
每次连接到数据库时都会设置该变量（包括程序启动时），但可以被取消设置。

`IGNOREEOF`

如果未设置，向 psql 的交互式会话发送一个 EOF 字符（通常是 **Control+D**）将终止应用程序。如果设为数值，则在应用程序终止之前会忽略那么多个 EOF 字符。如果变量已设置但没有数值，默认为 10。

> **注意**
>
> 此特性是从 Bash 厚颜无耻地抄袭来的。

`LASTOID`

最后被影响的 OID 的值，这可能会由`INSERT`或者`\lo_import`命令返回。这个变量只保证在下一个SQL命令的结果被显示完之前有效。

`ON_ERROR_STOP`

默认情况下，如果非交互式脚本遇到错误（如格式错误的 SQL 命令或内部元命令），处理会继续。这是 psql 的传统行为，但有时并不理想。如果设置了此变量，脚本处理将立即终止。如果该脚本是从另一个脚本调用的，后者也会以同样的方式终止。
如果最外层的脚本不是从交互式 psql 会话调用而是使用 -f 选项调用的，psql 将返回错误码 3，以区别于致命错误情况（错误码 1）。

`PORT`

当前连接到的数据库服务器端口。
每次连接到数据库时都会设置该变量（包括程序启动时），但可以被取消设置。

`PROMPT1`
`PROMPT2`
`PROMPT3`

这些变量指定psql发出的提示符的模样。见下文的提示符。

`QUIET`

此变量等价于命令行选项 `-q`。在交互模式下它可能不太有用。

`SINGLELINE`

此变量等价于命令行选项 `-S`。

SINGLESTEP

此变量等价于命令行选项 -s。

USER

当前连接的数据库用户。每次连接到一个数据库时都会设置该变量（包括程序启动时），但可以被取消设置。

VERBOSITY

这个变量可以被设置为值default、verbose或者terse来控制错误报告的详细程度。

SQL 插值

psql 变量的另一个有用特性是，你可以把它们替换（“插值”）到常规的 SQL 语句中。其语法同样是把变量名前加上冒号（:）。

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

将查询表 my_table。变量的值按字面复制，因此它甚至可以包含不平衡的引号或反斜线命令。你必须确保它放在哪里都有意义。带引号的 SQL 实体中不会执行变量插值。

此功能的一个流行应用是在后续语句中引用最后插入的 OID 来构建外键场景。此机制的另一个可能用法是把文件的内容复制到表列中。先把文件装载到一个变量中，然后按上面的做法进行。

```
testdb=> \set content '\'' `cat my_file.txt` '\''
testdb=> INSERT INTO my_table VALUES (:content);
```

此方法的一个可能问题是 my_file.txt 可能包含单引号。需要对这些引号转义，以免在处理第二行时导致语法错误。可以用 sed 程序做到这一点：

```
testdb=> \set content '\'' `sed -e "s/'/\\\\\\'/g" < my_file.txt` '\''
```

注意反斜杠的正确数量（6 个）！它的工作方式是这样的：psql 解析完这一行后，它把 `sed -e "s/'/\\\'/g" < my_file.txt` 传给 shell。shell 会在双引号内做自己的处理，然后带着参数 -e 和 `s/'/\\'/g` 执行 sed。sed 解析它时会把两个反斜杠替换为一个，然后执行替换。也许在某个时刻你曾觉得所有 Unix 命令使用相同的转义字符是很棒的事。而这还没有考虑你可能还必须转义所有反斜杠这一事实，因为 SQL 文本常量同样要经过某些解释。在这种情况下，你或许更适合在外部预先准备好文件。

由于冒号可以合法地出现在 SQL 命令中，因此适用以下规则：字符序列 :name 不会被更改，除非 name 是当前已设置的变量的名称。在任何情况下，你都可以用反斜线转义冒号来保护它不被替换。（变量的冒号语法是嵌入式查询语言（如 ECPG）的标准 SQL 语法。数组切片和类型转换的冒号语法是 PostgreSQL 的扩展，因此存在冲突。）

提示符

psql 发出的提示符可以按你的喜好进行定制。PROMPT1、PROMPT2 和 PROMPT3 这三个变量包含描述提示符外观的字符串和特殊转义序列。提示符 1 是 psql 请求新命令时发出的常规提示符。提示符 2 会在录入命令期间还需要更多输入时发出，例如命令尚未以分号结束，或者引号尚未闭合时。在执行 SQL COPY FROM STDIN 命令并需要在终端中输入一行值时，会发出提示符 3。

选定的提示符变量的值按字面打印，除非遇到百分号（%）。根据下一个字符的不同，会替换为某些其他文本。已定义的替换有：

`%M`

数据库服务器的完整主机名（含域名）；如果通过 Unix 域套接字连接，则为`[local]`；如果 Unix 域套接字不在编译时指定的默认位置，则为`[local:/dir/name]`。

`%m`

数据库服务器的主机名，在第一个点号处截断；如果通过 Unix 域套接字连接，则为`[local]`。

`%>`

数据库服务器监听的端口号。

`%n`

数据库会话用户名。（在数据库会话期间，`SET SESSION AUTHORIZATION`命令可能改变该值的扩展结果。）

`%/`

当前数据库的名称。

`%~`

类似 `%/`，但如果该数据库是你的默认数据库，则输出 `~`（波浪号）。

`%#`

如果会话用户是数据库超级用户，则为`#`，否则为`>`。（在数据库会话期间，`SET SESSION AUTHORIZATION`命令可能改变该值的扩展结果。）

`%R`

在提示符 1 中通常是 `=`，但在单行模式中是 `^`，如果会话与数据库断开连接（\connect 失败时可能发生）则是 `!`。在提示符 2 中该序列被替换为 `-`、`*`、单引号或双引号，取决于 psql expects more input because the command wasn't terminated yet, because you are inside a `/* ... */` comment, or because you are inside a quote. In prompt 3 the sequence doesn't produce anything.

`%x`

事务状态：如果当前不在事务块中，则为空字符串；如果处于事务块中，则为 `*`；如果处于失败的事务块中，则为 `!`；如果事务状态不确定（例如因为当前没有连接），则为 `?`。

`%digits`

替换为指定数字代码对应的字符。如果*digits*以`0x`开头，
其余的字符会被解释为十六进制；否则，如果第一位数字是`0`，这些数字会被解释为八进制；否则，这些数字会被读作十进制数。

`%:name:`

psql 变量 *name* 的值。详见变量。

`%`` `command` ``

command 的输出，类似普通的“反引号”替换。

To insert a percent sign into your prompt, write `%%`. The default prompts are `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

> **注意**
>
> 这个特性是可耻地从tcsh抄袭过来的。

命令行编辑

psql 支持 Readline 库以方便地进行行编辑和历史检索。命令历史保存在你的主目录下名为 `.psql_history` 的文件中，并在 psql 启动时重新装载。也支持 Tab 补全，不过补全逻辑并不自称是一个 SQL 解析器。如果由于某种原因你不喜欢 Tab 补全，可以把它关闭，方法是把以下内容放进你主目录下名为 `.inputrc` 的文件中：

```
$if psql
set disable-completion on
$endif
```

（这不是 psql 的特性而是 Readline 的。详细内容请阅读其文档。）

环境

`HOME`

初始化文件（`.psqlrc`）和命令历史文件（`.psql_history`）所在的目录。

`PAGER`

如果查询结果无法在屏幕上完整显示，则会通过管道传递给这个命令。典型值为 `more` 或 `less`。默认值取决于平台。可以使用 `\pset` 命令禁用分页器的使用。

`PGDATABASE`

要连接的默认数据库

`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数。

`PSQL_EDITOR`
`EDITOR`
`VISUAL`

`\e`命令使用的编辑器。这些变量按列出的顺序检查；使用第一个被设置的变量。

`SHELL`

被`\!`命令执行的命令。

`TMPDIR`

存储临时文件的目录。默认是`/tmp`。

文件

- 启动之前，psql 会尝试读取并执行文件 $HOME/.psqlrc 中的命令。它可用于按喜好设置客户端或服务器（使用 \set 和 SET 命令）。
- 命令行历史存储在文件 `$HOME/.psql_history` 中。

注解

- 在早期，psql 允许单字母反斜线命令的第一个参数直接跟在命令后面，中间没有空白。为了兼容性这一点在一定程度上仍被支持，但我们不打算在这里解释细节，因为不鼓励这种用法。如果你收到奇怪的消息，请记住这一点。For example

  ```
  testdb=> \foo
  Field separator is "oo".
  ```

  which is perhaps not what one would expect.
- psql 只能与相同版本的服务器顺利配合工作。这并不意味着其他组合会完全失败，但可能出现或微妙或不那么微妙的问题。如果服务器的版本不同，反斜线命令特别容易失败。

示例

第一个例子显示如何把一条命令分布在多行输入上。注意提示符的变化：

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text
testdb-> );
CREATE TABLE
```

现在再看看表定义：

```
testdb=> \d my_table
             Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 first     | integer | not null default 0
 second    | text    |
```

现在把提示符改成更有意思的样子：

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

假设你已经在表中填入了一些数据，想看一看它们：

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

用 \pset 命令可以让这个表显示成不同的样子:

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
```

Alternatively, use the short commands:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
```

```
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

vacuumdb

vacuumdb — 对一个 PostgreSQL 数据库进行垃圾收集和分析

大纲

`vacuumdb` [*`connection-option`*...] [[--full] | [-f]] [[--verbose] | [-v]] [[--analyze] | [-z]] [--table | -t *`table`* [(*`column`* [,...])]] [*`dbname`*]
`vacuumdb` [*`connection-options`*...] [[--all] | [-a]] [[--full] | [-f]] [[--verbose] | [-v]] [[--analyze] | [-z]]

描述

vacuumdb 是用于清理 PostgreSQL 数据库的工具。vacuumdb 还会生成供 PostgreSQL 查询优化器使用的内部统计信息。

vacuumdb是 SQL 命令VACUUM的包装器。通过该工具或通过访问服务器的其他方式来清理和分析数据库，在效果上没有区别。

选项

vacuumdb接受下列命令行参数：

`-a`
`--all`

清理所有数据库。

`[-d]` *`dbname`*
`[--dbname]` *`dbname`*

指定要清理或分析的数据库名称。如果未指定并且没有使用 `-a`（或`--all`），则从环境变量 `PGDATABASE` 中读取数据库名。如果该变量未设置，则使用连接时指定的用户名。

`-e`
`--echo`

回显 vacuumdb 生成并发送给服务器的命令。

`-f`
`--full`

执行“完全”清理。

`-q`
`--quiet`

不显示响应。

`-t` *`table`* `[ (`*`column`* `[,...]) ]`
`--table` *`table`* `[ (`*`column`* `[,...]) ]`

只清理或分析 *`table`*。列名只能与 `--analyze` 选项一起指定。

> **提示**
>
> 如果指定了列，你很可能需要在 shell 中转义括号。（见下面的示例。）

`-v`
`--verbose`

在处理过程中输出详细信息。

`-z`
`--analyze`

同时计算供优化器使用的统计信息。

vacuumdb还接受下列用于连接参数的命令行参数：

`-h` *`host`*
`--host` *`host`*

指定服务器所在机器的主机名。如果该值以斜杠开头，则它会被用作 Unix 域套接字目录。

`-p` *`port`*
`--port` *`port`*

指定服务器监听连接所用的 TCP 端口，或本地 Unix 域套接字文件扩展名。

`-U` *`username`*
`--username` *`username`*

连接时使用的用户名。

`-W`
`--password`

强制口令提示。

环境

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

默认连接参数

诊断

如遇困难，请参见VACUUM和psql中关于潜在问题和错误消息的讨论。
数据库服务器必须在目标主机上运行。此外，libpq 前端库所使用的任何默认连接设置和环境变量也都会生效。

注意

vacuumdb 可能需要多次连接 PostgreSQL 服务器，每次都要询问口令。这种情况下准备一个 `$HOME/.pgpass` 文件会很方便。更多信息见第 27.11 节。

示例

清理数据库 `test`：

```
$ vacuumdb test
```

清理并为优化器分析数据库 `bigdb`：

```
$ vacuumdb --analyze bigdb
```

清理单个表 `foo`，它位于数据库 `xyzzy` 中，并为优化器分析该表的单列 `bar`：

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

另见

VACUUM

PostgreSQL 服务器应用

这一部分包含PostgreSQL服务器应用和辅助实用工具的参考信息。
这些命令只有在数据库服务器所在的主机上运行才有意义。其他实用程序列在PostgreSQL 客户端应用中。

目录

initdb 846
initlocation 848
ipcclean 849
pg_controldata 850
pg_ctl 851
pg_resetxlog 855
postgres 856
postmaster 860

initdb

initdb — 创建一个新的 PostgreSQL 数据库集簇

大纲

initdb [*option*...] [--pgdata] | [-D]*directory*

描述

initdb创建一个新的 PostgreSQL数据库集簇。数据库集簇是由单个服务器实例管理的一组数据库的集合。

创建数据库集簇包括：创建存放数据库数据的目录，
生成共享目录表（属于整个集簇而不是任何特定数据库的表），以及创建 template1 数据库。当你以后创建新数据库时，template1 数据库中的所有内容都会被复制。其中包含为内建类型等内容填好的目录表。

initdb初始化数据库集簇的默认区域设置和字符集编码。
某些区域类别在集簇的整个生命期内是固定的，因此在运行initdb时做出正确选择很重要。
其他区域类别可以在以后启动服务器时更改。initdb会把这些区域设置写入 postgresql.conf 配置文件作为默认值，但可以通过编辑该文件来更改。要设置 initdb使用的区域，见--locale选项的描述。字符集编码可以在创建每个数据库时单独设置。initdb决定template1数据库的编码，它将作为所有其他数据库的默认值。要更改默认编码，使用--encoding选项。

initdb必须以将拥有服务器进程的用户身份运行，因为服务器需要访问initdb
创建的文件和目录。由于服务器不能以 root 身份运行，你也不能以 root 身份运行initdb。（事实上它会拒绝这样做。）

虽然initdb会尝试创建指定的数据目录，但它往往没有权限这样做，
因为所需数据目录的父目录常常是 root 所有的目录。要设置这样的安排，可以先以 root 身份创建一个空的数据目录，然后用chown把该目录的所有权移交给数据库用户账户，再su
成为数据库用户，最后以数据库用户身份运行initdb。

选项

-D *directory*
--pgdata=*directory*

这个选项指定数据库集簇的存放目录。这是 initdb所需的唯一信息，但你可以通过设置PGDATA环境变量来免于写出它，这样很方便，因为数据库服务器（postmaster）之后可以通过同一个变量找到数据库目录。

-E *encoding*
--encoding=*encoding*

选择模板数据库的编码。这也会是你以后创建的任何数据库的默认编码，
除非在那里另行覆盖。默认值是 SQL_ASCII。

--locale=*locale*

设置数据库集簇的默认区域。如果未指定这个选项，则从initdb运行的环境中继承区域。

`--lc-collate=locale`
`--lc-ctype=locale`
`--lc-messages=locale`
`--lc-monetary=locale`
`--lc-numeric=locale`
`--lc-time=locale`

类似`--locale`，但只设置指定类别中的区域。

`-U username`
`--username=username`

选择数据库超级用户的用户名。默认为运行`initdb`的有效用户的名称。超级用户叫什么其实并不重要，但即使操作系统用户的名称不同，人们也可能选择保留惯用名 postgres。

`-W`
`--pwprompt`

让`initdb`提示输入要赋予数据库超级用户的口令。如果你不打算使用口令认证，这无关紧要。否则，在设置好口令之前，你将无法使用口令认证。

还有其他一些较少使用的参数：

`-d`
`--debug`

打印引导后端的调试输出以及一些公众较少感兴趣的其他消息。引导后端是`initdb`用来创建目录表的程序。这个选项会产生大量极其无聊的输出。

`-L directory`

指定`initdb`到哪里寻找它初始化数据库集簇所需的输入文件。这通常是不必要的。如果需要显式指定它们的位置，系统会告诉你。

`-n`
`--noclean`

默认情况下，当`initdb`判定某个错误使它无法完全创建数据库集簇时，它会删除在发现无法完成工作之前可能已创建的所有文件。这个选项抑制清理，因此对调试有用。

环境

`PGDATA`

指定数据库集簇的存放目录；可以用`-D`选项覆盖。

另见

postgres, postmaster

initlocation

initlocation — 创建一个辅助的 PostgreSQL 数据库存储区

大纲

`initlocation` *`directory`*

描述

initlocation 创建一个新的 PostgreSQL 辅助数据库存储区。关于如何管理和使用辅助存储区，见CREATE DATABASE下的讨论。如果参数不包含斜杠并且不是有效的路径，就假定它是一个环境变量，并引用它。见末尾的例子。

要使用这个命令，你必须（例如用 `su`）以数据库超级用户身份登录。

示例

要在一个备选位置创建数据库，使用环境变量：

```
$ export PGDATA2=/opt/postgres/data
```

停止并启动 `postmaster` 使它看到 `PGDATA2` 环境变量。系统的配置必须使 `postmaster` 每次启动时都能看到 `PGDATA2`。最后：

```
$initlocation PGDATA2$createdb -D PGDATA2 testdb
```

或者，如果你允许绝对路径，可以写：

```
$initlocation /opt/postgres/data$createdb -D /opt/postgres/data/testdb
 testdb
```

ipcclean

ipcclean — 从异常中止的 PostgreSQL 服务器中移除共享内存和信号量

大纲

```
ipcclean
```

描述

`ipcclean` 移除当前用户拥有的所有共享内存段和信号量集。它打算用于在 PostgreSQL 服务器（postmaster）崩溃后进行清理。注意，立即重启服务器同样会清理共享内存和信号量，因此这个命令实际上没有多大用处。

只有数据库管理员才应执行这个程序，
因为在多用户执行期间运行它可能导致古怪的行为（即崩溃）。
如果在服务器运行时执行这个命令，该服务器分配的共享内存和信号量将被删除，
这对该服务器将产生相当严重的后果。

注意

这个脚本是一个应急之作（hack），但自它写成以来的许多年里，
没有人提出过同样有效且可移植的解决方案。由于 `postmaster` 现在能够自行清理，`ipcclean` 将来不太可能再被改进。

该脚本对 `ipcs` 实用程序的输出格式做了假设，这在不同的操作系统上可能不成立。因此，它可能无法在你的特定操作系统上工作。

pg_controldata

pg_controldata — 显示 PostgreSQL 数据库集簇的控制信息

大纲

`pg_controldata` [*datadir*]

描述

`pg_controldata` 会打印在 `initdb` 期间初始化的信息，例如系统目录版本和服务器区域设置。它还会显示有关预写式日志和检查点处理的信息。这些信息是整个集簇范围内的，不针对任何单个数据库。

该工具只能由初始化该集簇的用户运行，因为它需要对数据目录具有读取权限。你可以在命令行上指定数据目录，也可以使用环境变量 `PGDATA`。

环境

`PGDATA`

默认数据目录位置

pg_ctl

pg_ctl — 启动、停止或重启一个 PostgreSQL 服务器

大纲

`pg_ctl` start [-w] [-s] [-D *datadir*] [-l *filename*] [-o *options*] [-p *path*]
`pg_ctl` stop [-W] [-s] [-D *datadir*] [-m [s[mart]] | [f[ast]] | [i[mmediate]]]
`pg_ctl` restart [-w] [-s] [-D *datadir*] [-m [s[mart]] | [f[ast]] | [i[mmediate]]] [-o *options*]
`pg_ctl` reload [-s] [-D *datadir*]
`pg_ctl` status [-D *datadir*]

描述

pg_ctl 是一个用于启动、停止或重启 PostgreSQL 后端服务器（postmaster），或者显示一个运行中服务器的状态的工具。虽然服务器可以手动启动，但 pg_ctl 封装了诸如重定向日志输出以及恰当地从终端和进程组脱离这样的任务。它还为受控关闭提供了方便的选项。

在 `start` 模式中，会启动一个新的服务器。服务器在后台启动，标准输入被连接到 `/dev/null`。标准输出和标准错误要么被追加到一个日志文件（如果使用了 `-l` 选项），要么被重定向到 pg_ctl 的标准输出（不是标准错误）。如果没有选择日志文件，pg_ctl 的标准输出应当被重定向到一个文件或者通过管道传给另一个进程（例如一个日志轮转程序），否则 `postmaster` 会把它的输出写到控制终端（从后台）并且不会脱离 shell 的进程组。

在 `stop` 模式中，运行在指定数据目录中的服务器会被关闭。用 `-m` 选项可以选择三种不同的关闭方法：“智能”（Smart）模式等待所有客户端断开连接。这是默认方式。“快速”（Fast）模式不等待客户端断开连接。所有活动事务都会被回滚并且客户端被强制断开连接，然后服务器被关闭。“立即”（Immediate）模式将在不做干净关闭的情况下中止所有服务器进程。这会导致重启时运行一次恢复。

`restart` 模式实际上是先执行一次停止再执行一次启动。这允许更改 `postmaster` 的命令行选项。

`reload` 模式只是向 `postmaster` 进程发送一个 SIGHUP 信号，使它重新读取它的配置文件（`postgresql.conf`、`pg_hba.conf` 等）。这允许更改那些不需要完全重启就能生效的配置文件选项。

`status` 模式检查在指定数据目录中是否有一个服务器正在运行。如果有，就显示它的 PID 以及调用它时所用的命令行选项。

选项

`-D` *`datadir`*

指定数据库文件的文件系统位置。如果省略它，就使用环境变量 `PGDATA`。

`-l` *`filename`*

把服务器日志输出追加到 *`filename`*。如果该文件不存在，则创建它。umask 被设置为 077，因此默认情况下禁止其他用户访问该日志文件。

`-m` *`mode`*

指定关闭模式。*`mode`* 可以是 `smart`、`fast` 或 `immediate`，或者是这三者之一的首字母。

`-o` *options*

指定被直接传递给 `postmaster` 命令的选项。

这些选项通常用单引号或双引号包围，以确保它们作为一组被传递。

`-p` *path*

指定 `postmaster` 可执行文件的位置。默认情况下，`postmaster` 可执行文件取自与 `pg_ctl` 相同的目录，若失败则取写死的安装目录。没有必要使用这个选项，除非你在做某种不寻常的事情并且得到了找不到 `postmaster` 可执行文件的错误。

`-s`

只打印错误，不打印信息性消息。

`-w`

等待启动或关闭完成。超时时间为 60 秒。这是关闭的默认方式。成功的关闭以 PID 文件的消失为标志。对于启动，一次成功的 `psql -l` 表示成功。`pg_ctl` 会尝试为 psql 使用正确的端口。如果环境变量 `PGPORT` 存在，就使用它。否则，它会看 `postgresql.conf` 文件中是否设置了端口。如果两者都没有用上，它将使用 PostgreSQL 编译时的默认端口（默认为 5432）。

`-W`

不等待启动或关闭完成。这是启动和重启的默认方式。

环境

`PGDATA`

默认数据目录位置。

`PGPORT`

psql的默认端口（由 -w 选项使用）。

其他环境变量见postmaster。

文件

`postmaster.pid`

该文件在数据目录中的存在被用来帮助 pg_ctl 判断服务器当前是否在运行。

`postmaster.opts.default`

如果该文件存在于数据目录中，pg_ctl（在 `start` 模式中）会把该文件的内容作为选项传递给 `postmaster` 命令，除非被 `-o` 选项覆盖。

`postmaster.opts`

如果该文件存在于数据目录中，pg_ctl（在 `restart` 模式中）会把该文件的内容作为选项传递给 postmaster，除非被 `-o` 选项覆盖。该文件的内容也会在 `status` 模式中显示。

postgresql.conf

当在 start 模式中给出 -w 时，会解析位于数据目录中的该文件，以找到与 psql 一起使用的正确端口。

注意

等待完全启动不是一个定义良好的操作，
如果访问控制的设置使得本地客户端不手动交互就无法连接（例如口令认证），它可能会失败。

示例

启动服务器

启动一个服务器：

```
$pg_ctl start
```

一个启动服务器并阻塞到服务器起来的例子是：

```
$pg_ctl -w start
```

对于一个使用端口 5433 并且运行时不带 fsync 的服务器，使用：

```
$pg_ctl -o "-F -p 5433" start
```

停止服务器

```
$pg_ctl stop
```

停止服务器。使用 -m 开关可以控制后端如何关闭。

重启服务器

重启服务器几乎等价于停止服务器然后再启动它，区别在于 pg_ctl 会保存并重用传递给先前运行实例的命令行选项。要以最简单的形式重启服务器，使用：

```
$pg_ctl restart
```

重启服务器并等待它关闭再起来：

```
$pg_ctl -w restart
```

重启时使用端口 5433 并在重启后禁用 fsync：

```
$pg_ctl -o "-F -p 5433" restart
```

显示服务器状态

下面是一段来自 pg_ctl 的状态输出示例：

```
$pg_ctl statuspg_ctl: postmaster is running (pid: 13718)
Command line was:
/usr/local/pgsql/bin/postmaster '-D' '/usr/local/pgsql/data' '-p'
 '5433' '-B' '128'
```

这是在 restart 模式中将会被调用的命令行。

另见

postmaster

pg_resetxlog

pg_resetxlog — 重置一个 PostgreSQL 数据库集簇的预写式日志和其他控制信息

大纲

pg_resetxlog [-f] [-n] [-o *oid*] [-x *xid*] [-l *fileid*,*seg*] *datadir*

描述

pg_resetxlog 清除预写日志（WAL），并且可以选择性地重置一些其他的控制信息（存储在 pg_control 文件中）。如果这些文件已经被损坏，有时就需要这个功能。它应当只作为最后的手段使用，即当服务器因为这种损坏而无法启动时。

运行这个命令之后，应当就能启动服务器了，但请记住，由于部分提交的事务，
数据库可能包含不一致的数据。你应当立即转储你的数据，运行 initdb，然后重新加载。重新加载之后，检查不一致之处并根据需要进行修复。

这个工具只能由安装服务器的用户运行，因为它需要对数据目录的读/写访问权限。
出于安全原因，你必须在命令行上指定数据目录。pg_resetxlog 不使用环境变量 PGDATA。

如果 pg_resetxlog 抱怨它无法为 pg_control 确定有效的数据，你可以通过指定 -f（强制）开关强制它继续执行。在这种情况下，将用合理的值替代缺失的数据。大多数字段可以期望是匹配的，但下一个 OID、下一个事务 ID、WAL 起始地址和数据库区域字段可能需要人工辅助。其中前三项可以使用下面讨论的开关来设置。pg_resetxlog 自己的环境是它对区域字段猜测的来源；请注意让 LANG 等等与运行 initdb 时的环境相匹配。如果你无法为所有这些字段确定正确的值，仍然可以使用 -f，但恢复出来的数据库必须比平时更加谨慎地对待：必须立即转储并重新加载。在转储之前不要在数据库中执行任何修改数据的操作；
因为任何这样的动作都很可能使损坏更加严重。

-o、-x 和 -l 开关允许手动设置下一个 OID、下一个事务 ID 和 WAL 起始地址的值。只有在 pg_resetxlog 无法通过读取 pg_control 确定合适的值时才需要它们。下一个事务 ID 的安全值可以这样确定：找出数据目录下的 pg_clog 目录中数值最大的文件名，加一，然后乘以 1048576。注意文件名是十六进制的。通常用十六进制指定开关值也最容易。例如，如果 0011 是 pg_clog 中最大的条目，那么 -x 0x1200000 就可以工作（五个尾随零提供了正确的乘数）。WAL 起始地址应当大于数据目录下 pg_xlog 目录中当前存在的任何文件编号。地址也是十六进制的，并且有两部分。例如，如果 000000FF0000003A 是 pg_xlog 中最大的条目，那么 -l 0xFF,0x3B 就可以工作。没有类似的简单方法来确定一个超过数据库中最大 OID 的下一个 OID，但幸运的是，下一个 OID 设置得是否正确并不关键。

-n（无操作）开关指示 pg_resetxlog 打印从 pg_control 重构出来的值，然后不修改任何东西就退出。这主要是一个调试工具，但在允许 pg_resetxlog 真正执行之前作为一次健全性检查可能很有用。

注解

当服务器正在运行时，不得使用这个命令。如果 pg_resetxlog 在数据目录中发现一个服务器锁文件，它将拒绝启动。如果服务器崩溃了，那么可能残留了一个锁文件；
这种情况下你可以删除该锁文件以允许 pg_resetxlog 运行。但在这样做之前，务必再三确认那里没有 postmaster 或任何后端服务器进程仍然存活。

postgres

postgres — 以单用户模式运行一个PostgreSQL服务器

大纲

postgres [-A [0] | [1]] [-B *nbuffers*] [-c *name*=*value*] [-d *debug-level*] [--describe-config] [-D *datadir*] [-e] [-E] [-f [s] | [i] | [t] | [n] | [m] | [h]] [-F] [-N] [-o *filename*] [-O] [-P] [[-s] | [-t [pa] | [pl] | [ex]]] [-S *sort-mem*] [-W *seconds*] [--*name*=*value*] *database*
postgres [-A [0] | [1]] [-B *nbuffers*] [-c *name*=*value*] [-d *debug-level*] [-D *datadir*] [-e] [-f [s] | [i] | [t] | [n] | [m] | [h]] [-F] [-o *filename*] [-O] [-p *database*] [-P] [[-s] | [-t [pa] | [pl] | [ex]]] [-S *sort-mem*] [-v *protocol*] [-W *seconds*] [--*name*=*value*]

描述

postgres可执行文件是真正处理查询的 PostgreSQL服务器进程。通常不会直接调用它，而是启动一个postmaster多用户服务器。

上面的第二种形式是 postgres被postmaster调用的形式（这只是概念上的，因为postmaster和 postgres实际上是同一个程序）；不应直接以这种方式调用它。第一种形式以交互式单用户模式直接调用服务器。这种模式的主要用途是在initdb执行引导期间使用。
有时也会用它来调试或进行灾难恢复。

从 shell 以交互模式调用时，用户可以输入查询，结果会打印到屏幕上，但其形式对开发者更有用，而非面向终端用户。但请注意，运行单用户服务器其实并不适合调试服务器，
因为不会发生真实的进程间通信和锁定。

运行独立服务器时，会话用户会被设置为 ID 为 1 的用户。该用户实际上不必存在，因此独立服务器可用于手工恢复系统目录遭受某些意外损坏的情况。在独立模式下，会向 ID 为 1 的用户隐式授予超级用户权限。

选项

当postgres由postmaster启动时，它继承后者设置的所有选项。此外，可以通过 postmaster的 -o开关传递postgres特有的选项。

通过设置配置文件，你可以避免键入这些选项。详见第 16.4 节。某些（安全的）选项还可以由连接的客户端以应用相关的方式设置。例如，如果设置了环境变量PGOPTIONS，那么基于libpq的客户端就会将该字符串传给服务器，服务器会将其解释为 postgres命令行选项。

通用选项

选项-A、-B、-c、-d、-D、-F和--*name*的含义与postmaster中的相同，只是 -d 0会阻止postmaster的服务器日志级别传播到postgres。

-e

将默认日期风格设置为“European”，即输入日期字段采用 DMY顺序。这也会在某些日期输出格式中使日期显示为日在月前。更多信息见第 8.5 节。

-o *filename*

将所有服务器日志输出发送到 *filename*。如果postgres运行在 postmaster之下，该选项会被忽略，并使用从 postmaster继承的stderr。

`-P`

读取系统表时忽略系统索引，但在修改表时仍然更新这些索引。
这在从损坏的系统索引中恢复时很有用。

`-s`

在每条命令结束时打印时间信息和其他统计信息。
这对基准测试或者调整缓冲区数量很有用。

`-S` *sort-mem*

指定内部排序和哈希在求助于临时磁盘文件之前可使用的内存量。
取值以千字节为单位指定，默认为 1024。注意，对于一个复杂查询，可能有多个排序和/或哈希并行运行，每一个在被允许开始把数据放入临时文件之前，都可以使用多达 *sort-mem* 千字节。

独立模式的选项

database

指定要访问的数据库名称。如果省略，则默认为用户名。

`-E`

回显所有命令。

`-N`

禁用换行符作为语句分隔符。

半内部选项

还有一些其他选项，主要供调试之用。这里列出这些选项，仅供PostgreSQL 系统开发者使用。强烈不建议使用这些选项中的任何一个。此外，
这些选项中的任何一个都可能在未来的版本中不经通知而消失或改变。

`-f { s | i | m | n | h }`

禁止使用特定的扫描和连接方法：`s` 和 `i` 分别禁用顺序扫描和索引扫描，而 `n`、`m` 和 `h` 分别禁用嵌套循环、归并和哈希连接。

> **注意**
>
> 顺序扫描和嵌套循环连接都无法被完全禁用；`-fs`和 `-fn`选项只是在优化器有其他选择时，尽量不使用这些计划类型。

`-O`

允许修改系统表的结构。该选项由 `initdb`使用。

`-p` *database*

指示该进程是由 `postmaster`启动的，并指定要使用的数据库等。

-t pa[rser] | pl[anner] | e[xecutor]

打印与各个主要系统模块相关的每个查询的时间统计信息。该选项不能与-s选项一起使用。

-v *protocol*

指定该特定会话要使用的前端/后端协议版本号。

-W *seconds*

一旦遇到该选项，进程就会休眠指定的秒数。
这为开发者把调试器附着到服务器进程提供了时间。

--describe-config

该选项会以制表符分隔的COPY格式导出服务器内部配置变量、说明以及默认值。
它主要是为管理工具设计的。

环境

PGDATA

默认数据目录位置

对于在单用户模式下影响不大的其他环境变量，见postmaster。

注意

要取消一个正在运行的查询，可向运行该命令的postgres进程发送 SIGINT 信号。

要告诉postgres重新加载配置文件，可发送SIGHUP信号。通常最好向 postmaster发送SIGHUP；postmaster会继而向它的每个子进程发送SIGHUP。但在某些情况下，
可能只需要一个 postgres进程重新加载配置文件。

postmaster 用 SIGTERM 通知 postgres 进程正常退出，用 SIGQUIT 终止而不做正常清理。用户不应使用这些信号。向 postgres 进程发送 SIGKILL 也是不明智的——postmaster 会把这解释为 postgres 发生了崩溃，并会按照其标准的崩溃恢复过程，强制所有兄弟 postgres 进程退出。

用法

用类似下面的命令启动一个独立服务器：

postgres -D /usr/local/pgsql/data *other-options* my_database

使用-D提供正确的数据库目录路径，或者确保已经设置了环境变量PGDATA。
还要指定你想要操作的那个数据库的名称。

通常，独立服务器将换行视为命令输入终止符；它不像psql那样会对分号进行智能处理。
要让一个命令跨越多行，必须在除最后一个换行之外的每个换行前输入反斜线。

但是，如果你使用-N命令行开关，换行并不会终止命令输入。在这种情况下，
服务器会读取标准输入直到文件末尾（EOF）标记，然后把输入作为单个命令字符串处理。
在这种情况下，反斜线-换行不会被特殊处理。

要退出会话，输入EOF（通常是**Control+D**）。如果你使用了`-N`，则需要连续输入两次EOF才能退出。

注意，独立服务器不提供复杂的行编辑功能（例如没有命令历史）。

另请参阅

initdb, ipcclean, postmaster

postmaster

postmaster — PostgreSQL多用户数据库服务器

大纲

postmaster [-A [0] | [1]] [-B *nbuffers*] [-c *name*=*value*] [-d *debug-level*] [-D *datadir*] [-F] [-h *hostname*] [-i] [-k *directory*] [-l] [-N *max-connections*] [-o *extra-options*] [-p *port*] [-S] [--*name*=*value*] [[-n] | [-s]]

描述

postmaster是PostgreSQL多用户数据库服务器。客户端应用要访问数据库时，会（通过网络或本地）连接到一个正在运行的 postmaster。该 postmaster随后会启动一个独立的服务器进程（“postgres”）来处理该连接。postmaster还负责管理各服务器进程之间的通信。

默认情况下，postmaster在前台启动，并将日志消息打印到标准错误流。在实际应用中，postmaster 应作为后台进程启动，例如在系统启动时启动。

一个 postmaster 总是恰好管理一个数据库集群的数据。数据库集群是存储在公共文件系统位置上的一组数据库的集合。postmaster 启动时需要知道数据库集群文件（“数据区”）的位置。这通过 -D 调用选项或 PGDATA 环境变量完成；没有默认值。一个系统上可以同时运行多个 postmaster 进程，只要它们使用不同的数据区和不同的通信端口（见下文）。数据区用initdb创建。

选项

postmaster 接受下列命令行参数。关于这些选项的详细讨论请查阅第 16.4 节。通过设置配置文件，大多数这样的选项还可以省去重复输入。

-A 0|1

启用运行时断言检查，这是一种检测编程错误的调试辅助手段。只有在编译时启用了它才可用。如果是这样，默认为开启。

-B *nbuffers*

设置服务器进程使用的共享缓冲区数量。该值默认为 64 个缓冲区，每个缓冲区为 8 kB。

-c *name=value*

设置一个命名的运行时参数。列表和描述请查阅第 16.4 节。大多数其他命令行选项实际上是这种参数赋值的简短形式。-c 可以出现多次以设置多个参数。

-d *debug-level*

设置调试级别。该值设得越高，写入服务器日志的调试输出就越多。取值范围是 1 到 5。

-D *datadir*

指定数据目录的文件系统位置。见上面的讨论。

`-F`

为了改进性能而禁用 `fsync` 调用，代价是系统崩溃时可能发生数据损坏。这个选项对应于在 `postgresql.conf` 中设置 `fsync=false`。使用前请阅读详细的文档！

`--fsync=true` 的效果与此选项相反。

`-h` *`hostname`*

指定 `postmaster` 用来监听客户端应用连接的 IP 主机名或地址。默认监听所有已配置的地址（包括 localhost）。

`-i`

允许客户端通过 TCP/IP（Internet 域）连接。不使用这个选项时，只接受本地 Unix 域套接字连接。这个选项对应于在 `postgresql.conf` 中设置 `tcpip_socket=true`。

`--tcpip-socket=false` 的效果与此选项相反。

`-k` *`directory`*

指定 `postmaster` 监听来自客户端应用连接的 Unix 域套接字的目录。默认通常是 `/tmp`，但可以在构建时更改。

`-l`

启用使用 SSL 的安全连接。还需要 `-i` 选项。你必须以启用 SSL 的方式编译才能使用这个选项。

`-N` *`max-connections`*

设置这个 `postmaster` 可接受的最大客户端连接数。默认值为 32，但可以设置到你的系统所支持的程度。（注意 `-B` 至少要为 `-N` 的两倍。关于大量客户端连接的系统资源需求的讨论见第 16.5 节。）

`-o` *`extra-options`*

在*`extra-options`*中指定的命令行形式选项，会传给由这个 `postmaster`启动的所有服务器进程。可能的选项见postgres。如果选项字符串中包含空格，整个字符串必须加引号。

`-p` *`port`*

指定`postmaster` 用来侦听客户端应用连接的 TCP/IP 端口或本地 Unix 域套接字文件扩展名。默认值取自`PGPORT`环境变量；如果`PGPORT`未设置，
则默认值就是在编译期间确定的值（通常为 5432）。如果你指定了非默认端口，那么所有客户端应用都必须通过命令行选项或`PGPORT`指定相同的端口。

`-S`

指定`postmaster` 进程应以静默模式启动。也就是说，它将脱离用户的（控制）终端，启动自己的进程组，并把标准输出和标准错误重定向到`/dev/null`。

使用该开关会丢弃所有日志输出，这很可能不是你想要的，
因为它会使问题排查变得非常困难。在后台启动 `postmaster`的更好方法见下文。

`--silent-mode=false`与该选项的效果相反。

--name=value

设置一个命名的运行时参数；这是-c的较短形式。

还有两个附加的命令行选项可用于调试导致服务器进程异常终止的问题。在这种情况下，通常的策略是通知所有其他服务器进程必须终止，然后重新初始化共享内存和信号量。这是因为出错的服务器进程可能在终止前破坏了某些共享状态。这些选项用来选择 postmaster 在这种情况下的一些替代行为。这两个选项都不适用于普通操作。

这些特殊情况选项是：

-n

postmaster 将不重新初始化共享数据结构。经验丰富的系统程序员随后可以使用调试器检查共享内存和信号量的状态。

-s

postmaster 将通过发送SIGSTOP信号暂停所有其他服务器进程，但不使它们终止。这允许系统程序员手工收集所有服务器进程的核心转储。

环境

PGCLIENTENCODING

客户端使用的默认字符编码。（客户端可以单独覆盖它。）该值也可以在配置文件中设置。

PGDATA

默认的数据目录位置

PGDATESTYLE

DATESTYLE 运行时参数的默认值。（不推荐使用这个环境变量。）

PGPORT

默认端口（最好在配置文件中设置）

TZ

服务器时区。

others

其他环境变量可用来指定备选的数据存储位置。更多信息见第 18.5 节。

诊断

提到semget或 shmget的失败消息，很可能表明你需要配置内核，以提供足够的共享内存和信号量。更多讨论见第 16.5 节。

提示

通过减小 shared_buffers 来降低 PostgreSQL 的共享内存消耗，和/或减小 max_connections 来降低信号量消耗，你可能可以推迟重新配置内核。

表明另一个 postmaster 已经在运行的失败消息应仔细检查，例如可根据你的系统使用下面的命令：

```
$ps ax | grep postmaster
```

或

```
$ps -ef | grep postmaster
```

如果你确信没有冲突的 postmaster 在运行，可以删除消息中提到的锁文件，然后重试。

指示无法绑定到某个端口的失败消息可能表示该端口已被某个非 PostgreSQL 进程占用。如果你终止了 postmaster 然后立即用同一端口重启，也可能得到这个错误；这种情况下，只需等待几秒直到操作系统关闭该端口再重试。最后，如果你指定的端口号被操作系统视为保留端口，也可能得到这个错误。例如，许多版本的 Unix 认为 1024 以下的端口号是“受信任的”，只允许 Unix 超级用户访问它们。

注解

只要有可能，就不要使用 SIGKILL杀死 postmaster。这样会阻止 postmaster在终止前释放其持有的系统资源（例如共享内存和信号量）。这可能会导致新的postmaster实例在启动时出现问题。

要正常终止postmaster，可以使用 SIGTERM、SIGINT或 SIGQUIT信号。第一个会在退出前等待所有客户端终止，第二个会强制断开所有客户端，而第三个会在不进行正确关闭的情况下立即退出，从而在重启期间导致一次恢复运行。SIGHUP 信号会重新加载服务器配置文件。

实用命令pg_ctl可用于安全、便捷地启动和关闭postmaster。

--选项在FreeBSD或OpenBSD上不起作用。请改用-c。这是受影响操作系统中的一个缺陷；如果这个问题未被修复，未来版本的PostgreSQL 将提供变通方案。

示例

要使用默认值在后台启动postmaster，输入：

```
$nohup postmaster >logfile 2>&1 </dev/null &
```

要用特定端口启动 postmaster：

```
$postmaster -p 1234
```

这条命令将启动 postmaster 并通过端口 1234 通信。为了用 psql 连接这个 postmaster，你需要将它作为

```
$psql -p 1234
```

来运行，或者设置环境变量 PGPORT：

```
$export PGPORT=1234$psql
```

命名的运行时参数可以用下面两种风格中的任一种设置：

```
$postmaster -c sort_mem=1234$postmaster --sort-mem=1234
```

这两种形式都会覆盖 `postgresql.conf` 中可能存在的 `SORT_MEM` 设置。注意，命令行上参数名中的下划线既可以写成下划线也可以写成短横线。

> **提示**
>
> 除了短期实验之外，与其依赖命令行开关来设置参数，通常更好的做法是直接编辑 `postgresql.conf`中的设置。

另见

initdb、pg_ctl

部分 VII. 内部

本部分包含一些可能对PostgreSQL开发人员有用的各类信息。

目录

42. PostgreSQL 内部概述 868
42.1. 查询的路径 868
42.2. 连接是如何建立的 868
42.3. 解析器阶段 869
42.3.1. 解析器 869
42.3.2. 转换过程 869
42.4. PostgreSQL 规则系统 870
42.5. 规划器/优化器 870
42.5.1. 生成可能的计划 870
42.6. 执行器 871
43. 系统目录 872
43.1. 概述 872
43.2. pg_aggregate 873
43.3. pg_am 873
43.4. pg_amop 874
43.5. pg_amproc 874
43.6. pg_attrdef 875
43.7. pg_attribute 875
43.8. pg_cast 877
43.9. pg_class 877
43.10. pg_constraint 879
43.11. pg_conversion 880
43.12. pg_database 880
43.13. pg_depend 881
43.14. pg_description 882
43.15. pg_group 883
43.16. pg_index 883
43.17. pg_inherits 884
43.18. pg_language 884
43.19. pg_largeobject 885
43.20. pg_listener 885
43.21. pg_namespace 886
43.22. pg_opclass 886
43.23. pg_operator 887
43.24. pg_proc 888
43.25. pg_rewrite 889
43.26. pg_shadow 889
43.27. pg_statistic 890
43.28. pg_trigger 891
43.29. pg_type 892
43.30. 系统视图 895
43.31. pg_indexes 896
43.32. pg_locks 896
43.33. pg_rules 897
43.34. pg_settings 897
43.35. pg_stats 898
43.36. pg_tables 899
43.37. pg_user 900
43.38. pg_views 900
44. 前端/后端协议 901

44.1. 概述 901
44.1.1. 消息传送概述 901
44.1.2. 扩展查询概述 902
44.1.3. 格式和格式代码 902
44.2. 消息流 902
44.2.1. 启动 902
44.2.2. 简单查询 904
44.2.3. 扩展查询 905
44.2.4. 函数调用 907
44.2.5. COPY 操作 908
44.2.6. 异步操作 909
44.2.7. 取消进行中的请求 909
44.2.8. 终止 910
44.2.9. SSL 会话加密 910
44.3. 消息数据类型 911
44.4. 消息格式 911
44.5. 错误和通知消息字段 925
44.6. 自协议 2.0 以来的更改汇总 926
45. PostgreSQL 编码约定 928
45.1. 格式 928
45.2. 在服务器内部报告错误 928
45.3. 错误消息风格指南 930
46. 本地语言支持 935
46.1. 给翻译者 935
46.1.1. 要求 935
46.1.2. 概念 935
46.1.3. 创建和维护消息目录 936
46.1.4. 编辑 PO 文件 937
46.2. 给程序员 937
46.2.1. 实现机制 937
46.2.2. 消息编写指南 938
47. 编写过程语言调用处理器 940
48. 遗传查询优化器 942
48.1. 将查询处理看成是一个复杂的优化问题 942
48.2. 遗传算法 942
48.3. PostgreSQL 中的遗传查询优化（GEQO） 943
48.3.1. PostgreSQL GEQO 的未来实现任务 944
48.4. 进一步阅读 944
49. 索引代价估计函数 945
50. GiST 索引 948
50.1. 介绍 948
50.2. 可扩展性 948
50.3. 实现 948
50.4. 限制 949
50.5. 示例 949
51. 页文件 951
52. BKI 后端接口 954
52.1. BKI 文件格式 954
52.2. BKI 命令 954
52.3. 示例 955

第 42 章 PostgreSQL 内部概述

作者

本章最初源自[sim98] Stefan Simkovics 在维也纳技术大学完成的硕士论文，该论文由 O. Univ.Prof.Dr. Georg Gottlob 和 Univ.Ass. Mag. Katrin Seyr 指导。

本章概述 PostgreSQL 后端的内部结构。阅读以下各节之后，你应该能够大致了解一个查询是如何被处理的。本章并不试图详细描述 PostgreSQL 的内部运作，因为那样的文档会非常庞大。相反，本章旨在帮助读者理解，从后端收到一个查询开始，到将结果返回给客户端为止，内部通常有哪些操作步骤。

42.1. 查询的路径

这里将简要概述一个查询为得到结果必须经过的各个阶段。

1. 必须先建立从应用程序到 PostgreSQL 服务器的连接。应用程序将查询发送给服务器，并等待接收服务器返回的结果。

2. 解析器阶段检查应用程序传来的查询是否具有正确的语法，并创建一棵查询树。

3. 重写系统接收由解析器阶段创建的查询树，并查找任何可应用于该查询树的规则（存储在系统目录中）。它执行这些规则体中给出的转换。
   重写系统的一个应用是实现视图。

   每当对一个视图（即虚拟表）发出查询时，重写系统都会将用户的查询重写成一个改为访问视图定义中给出的基表的查询。

4. 规划器/优化器接收（重写后的）查询树，并创建一个查询计划，将作为执行器的输入。

   它首先生成所有能得到同一结果的可能路径。例如，如果待扫描的某个关系上有一个索引，那么该扫描就有两条路径：一种是简单的顺序扫描，另一种是使用该索引。
   接着会估算执行每个计划的代价，选出代价最低的计划并交回。

5. 执行器会递归地遍历计划树，并以计划所表示的方式提取行。执行器在扫描关系时会使用存储系统，执行排序和连接，计算限定条件，最后返回得到的行。

在后续各节中，我们将更详细地介绍上述各项内容，以便更好地理解 PostgreSQL 的内部控制和数据结构。

42.2. 连接是如何建立的

PostgreSQL 使用一种简单的“每用户一个进程”客户端/服务器模型。在这种模型中，一个客户端进程恰好连接到一个服务器进程。由于事先不知道会建立多少个连接，因此必须使用一个主进程，在每次收到连接请求时派生一个新的服务器进程。这个主进程称为 `postmaster`，在指定的 TCP/IP 端口上监听传入连接。每当检测到连接请求时，`postmaster` 进程就会派生一个新的称为 `postgres` 的服务器进程。各个服务器任务（`postgres` 进程）之间使用信号量和共享内存通信，以确保并发数据访问期间的数据完整性。

客户端进程可以是任何理解 PostgreSQL 协议（见第 44 章）的程序。很多客户端基于 C 语言库 libpq，但该协议也有若干独立实现，例如 Java 的 JDBC 驱动。

一旦连接建立起来，客户端进程就可以向后端（服务器）发送查询。查询以纯文本方式传输，也就是说，前端（客户端）不进行解析。服务器解析查询，创建一个执行计划，执行该计划，并通过已建立的连接把检索到的行返回给客户端。

42.3. 解析器阶段

解析器阶段由两部分组成：

- 解析器定义在 `gram.y` 和 `scan.l` 中，它使用 Unix 工具 yacc 和 lex 构建。
- 转换过程，它会对解析器返回的数据结构做修改和补充。

42.3.1. 解析器

解析器必须检查查询串（以普通 ASCII 文本形式到达）的语法是否有效。如果语法正确，就构建一棵解析树并返回；否则返回错误。解析器和词法分析器是用著名的 Unix 工具yacc和lex实现的。

词法分析器定义在文件 `scan.l` 中，负责识别标识符、SQL 关键字等。每找到一个关键字或标识符，就会生成一个词元并交给解析器。

解析器定义在文件 `gram.y` 中，由一组语法规则和动作组成，每当某条规则被触发时，对应动作就会执行。动作中的代码（实际上是 C 代码）用于构造语法解析树。

文件 `scan.l` 会被转换成 C 源文件 `scan.c`，所用程序是 lex；而 `gram.y` 则会被转换成 `gram.c`，所用程序是 yacc。这些转换完成之后，就可以使用普通的 C 编译器来构建解析器。绝不要修改这些生成出来的 C 文件，因为下一次调用 lex 或 yacc 时，它们都会被覆盖。

> **注意**
>
> 上述转换和编译通常都是借助 makefiles 自动完成的，这些文件随 PostgreSQL 源代码发行版一同提供。

对 yacc 的详细描述，或者对 `gram.y` 中给出的语法规则的说明，都超出了本文的范围。有许多书籍和文档专门讨论 lex 和 yacc。在开始研究之前，你应该先熟悉 yacc，然后再去看 `gram.y` 中给出的语法，否则你将无法理解其中发生了什么。

42.3.2. 转换过程

解析器阶段仅依据 SQL 语法结构的固定规则创建一棵语法解析树。它不会在系统目录中做任何查找，因此不可能理解所请求操作的详细语义。解析器完成之后，
转换过程会接收解析器返回的树，并进行必要的语义解释，以理解查询引用了哪些表、函数和操作符。为表示这些信息而构建的数据结构称为查询树。

之所以将原始解析与语义分析分开，是因为系统目录查找只能在事务内部进行，
而我们不希望在刚收到查询字符串时就立刻启动事务。
原始解析阶段已经足以识别事务控制命令（`BEGIN`、`ROLLBACK` 等），因此这些命令可以在无需进一步分析的情况下被正确执行。一旦我们知道当前处理的是实际查询（例如 `SELECT` 或 `UPDATE`），如果尚未处于事务中，就可以启动事务。只有在那之后才能调用转换过程。

转换过程创建的查询树在大多数地方的结构都与原始语法解析树相似，但在细节上有许多不同。例如，语法解析树中的一个 `FuncCall` 节点表示某个在语法上看起来像函数调用的东西。它可能

被转换成 `FuncExpr` 节点，也可能被转换成 `Aggref` 节点，这取决于被引用的名称最终被判定为普通函数还是聚合函数。此外，
关于列和表达式结果的实际数据类型的信息也会被加入到查询树中。

42.4. PostgreSQL 规则系统

PostgreSQL 提供了一个强大的规则系统，用于定义视图以及处理有歧义的视图更新。最初，PostgreSQL 的规则系统由两种实现组成：

- 第一种实现使用行级处理，并且深度实现于执行器内部。每当访问到单独一行时，
  规则系统就会被调用。这种实现在 1995 年被移除，当时 Berkeley Postgres 项目的最后一个官方发行版被转换成了 Postgres95。
- 规则系统的第二种实现是一种称为查询重写的技术。
  重写系统是位于解析器阶段与规划器/优化器之间的一个模块。这种技术至今仍在使用。

关于查询重写器，第 34 章中已有相当详细的讨论，因此这里没有必要再展开。我们只指出一点：重写器的输入和输出都是查询树，也就是说，树的表示形式以及语义细节层次都不会发生变化。重写可以被看作某种形式的宏展开。

42.5. 规划器/优化器

规划器/优化器的任务是创建一个最优的执行计划。给定的一个 SQL 查询（因此也就是一棵查询树）实际上可以用多种不同方式执行，而这些方式都会产生同样的结果集。如果在计算上可行，查询优化器将考察这些可能执行计划中的每一种，并最终选择预计运行最快的那个执行计划。

> **注意**
>
> 在某些情况下，考察查询的每一种可能执行方式会耗费过多的时间和内存空间，
> 尤其是在执行涉及大量连接操作的查询时更是如此。
> 为了在合理的时间内确定一个合理的（并非最优的）查询计划，PostgreSQL 使用了第 48 章。

在确定出代价最低的路径之后，会构建一棵计划树传递给执行器。
这棵树以足够细致的方式表示了期望的执行计划，使执行器能够运行它。

42.5.1. 生成可能的计划

规划器/优化器根据查询中各关系上所定义索引的类型来决定应该生成哪些计划。
对一个关系总是可以执行顺序扫描，因此总会创建一个只使用顺序扫描的计划。
假设某个关系上定义了一个索引（例如一个 B-树索引），并且查询中包含限制条件 `relation.attribute OPR constant`。如果 `relation.attribute` 恰好匹配该 B-树索引的键，而且 `OPR` 是该索引操作符类中列出的操作符之一，就会再创建一个使用该 B-树索引扫描该关系的计划。如果还存在其他索引，并且查询中的限制条件恰好匹配某个索引的键，则还会考虑更多计划。

在找出扫描单个关系的所有可行计划之后，就会创建连接各关系的计划。规划器/优化器会优先考虑在 `WHERE` 限定条件中存在相应连接子句的任意两个关系之间的连接（即存在类似 `where rel1.attr1=rel2.attr2` 这样的限制）。没有连接子句的连接对只有在别无选择时才会被考虑，也就是说，某个关系与其他任何关系之间都没有可用的连接子句。对于规划器/优化器所考虑的每一对连接，都会生成所有可能的计划。三种可用的连接策略是：

- 嵌套循环连接：左关系中找到的每一行，都会使右关系被扫描一次。这种策略实现起来很容易，但可能非常耗时。（不过，如果右关系可以通过索引扫描，那么这也可能是一种不错的策略。可以把左关系当前行中的值作为右关系索引扫描的键。）

- 归并连接：在连接开始之前，每个关系都会先按照连接属性排序。然后，考虑到两个关系都已经按连接属性排好序，把它们归并到一起。这种连接方式更有吸引力，因为每个关系只需扫描一次。

- 哈希连接：首先扫描右关系，并使用其连接属性作为哈希键将其装入一个哈希表。接着扫描左关系，并将找到的每一行中的相应值作为哈希键，用来在该哈希表中定位匹配的行。

最终完成的计划树由基表的顺序扫描或索引扫描，再加上所需的嵌套循环、归并或哈希连接节点，以及任何需要的辅助步骤（例如排序节点或聚合函数计算节点）组成。这些计划节点类型中的大多数还具有执行选择（丢弃不满足指定布尔条件的行）和投影（根据给定列值计算派生列集，也就是在需要时计算标量表达式）的额外能力。规划器的职责之一，就是把来自 `WHERE` 子句的选择条件以及所需输出表达式的计算附加到计划树中最合适的节点上。

42.6. 执行器

执行器接收由规划器/优化器创建的计划，并递归地处理它，以提取所需的行集合。这本质上是一种按需拉取的流水线机制。每当某个计划节点被调用时，它必须再交付一行，或者报告自己已经完成了行的交付。

为了给出一个具体示例，假设顶层节点是一个 `MergeJoin` 节点。在执行归并之前，必须先取到两行（每个子计划各一行）。因此执行器会递归调用自身去处理这些子计划（从挂接在 `lefttree` 上的子计划开始）。新的顶层节点，也就是左子计划的顶层节点，假设是一个 `Sort` 节点，那么又需要通过递归来取得一个输入行。`Sort` 的子节点可能是一个 `SeqScan` 节点，它表示真正去读取一个表。执行该节点会使执行器从表中取出一行，并把它返回给调用者节点。`Sort` 节点会反复调用它的子节点，以取得所有待排序的行。当输入耗尽时（子节点返回的是 NULL 而不是一行），`Sort` 代码就执行排序，并最终能够返回它的第一条输出行，也就是排序后的第一行。它会把其余行保存起来，以便在后续请求中按排序后的顺序交付这些行。

`MergeJoin` 节点同样会向它的右子计划请求第一行。然后它比较这两行，看它们是否可以连接；如果可以，就向调用者返回一条连接结果行。在下一次调用时，或者在当前这一对输入无法连接时立即，它会前进到某一个表的下一行或另一个表的下一行（取决于比较结果），并再次检查是否匹配。最终，其中一个子计划会耗尽，`MergeJoin` 节点就返回 NULL，以表明无法再形成更多连接结果行。

复杂查询可能涉及许多层计划节点，但总体方法是相同的：每个节点在每次被调用时，都会计算并返回它的下一条输出行。每个节点还负责应用规划器分配给它的任何选择或投影表达式。

执行器机制用于求值全部四种基本 SQL 查询类型：`SELECT`、`INSERT`、`UPDATE` 和 `DELETE`。对于 `SELECT`，顶层执行器代码只需要把查询计划树返回的每一行发送给客户端。对于 `INSERT`，每个返回的行会被插入到为该 `INSERT` 指定的目标表中。（一个简单的 `INSERT ... VALUES` 命令会创建一棵极其简单的计划树，它只包含一个仅计算单条结果行的 `Result` 节点。但 `INSERT ... SELECT` 可能需要执行器机制的全部能力。）对于 `UPDATE`，规划器会安排每个计算得到的行包含所有被更新列的值，再加上原始目标行的 TID（tuple ID，即元组 ID 或行 ID）；执行器顶层利用这些信息创建一个新的更新后行，并将旧行标记为已删除。对于 `DELETE`，计划实际返回的唯一列就是 TID，执行器顶层只需利用该 TID 访问每个目标行并将它们标记为已删除。

第 43 章 系统目录

系统目录是关系型数据库管理系统存放模式元数据的地方，
例如表和列的信息以及内部记账信息。PostgreSQL 的系统目录是普通表。你可以删除并重建这些表、增加列、插入和更新数值，并借此把系统严重搞坏。通常，不应手工修改系统目录，
总是有 SQL 命令可用于完成这些操作。（例如，`CREATE DATABASE` 会向 `pg_database` 目录中插入一行 – 并且实际上会在磁盘上创建该数据库。）对于某些特别深奥的操作存在一些例外，
例如增加索引访问方法。

43.1. 概述

表 43.1列出了系统目录。每个目录更详细的文档见后文。

大多数系统目录在创建数据库时从模板数据库复制而来，此后便是数据库特定的。
少数目录在一个集群中的所有数据库之间物理共享；这些目录在各个目录的说明中有标注。

表 43.1. 系统目录

目录名	用途
pg_aggregate	聚合函数
pg_am	索引访问方法
pg_amop	访问方法操作符
pg_amproc	访问方法支持过程
pg_attrdef	列默认值
pg_attribute	表列（“属性”）
pg_cast	转换（数据类型转换）
pg_class	表、索引、序列（“关系”）
pg_constraint	检查约束、唯一约束、主键约束、外键约束
pg_conversion	编码转换信息
pg_database	此数据库集群中的数据库
pg_depend	数据库对象之间的依赖
pg_description	数据库对象的描述或注释
pg_group	数据库用户组
pg_index	附加的索引信息
pg_inherits	表继承层次
pg_language	编写函数的语言
pg_largeobject	大对象
pg_listener	异步通知支持
pg_namespace	模式
pg_opclass	索引访问方法操作符类
pg_operator	操作符
pg_proc	函数和过程
pg_rewrite	查询重写规则
pg_shadow	数据库用户
pg_statistic	规划器统计信息

目录名	用途
pg_trigger	触发器
pg_type	数据类型

43.2. pg_aggregate

目录pg_aggregate存储关于聚合函数的信息。
聚合函数是对一组值（典型的是每个匹配查询条件的行中的同一个列的值）进行操作的函数，它返回从这些值中计算出的单个值。典型的聚合函数是 sum、count和max。pg_aggregate里的每个项都是一个pg_proc项的扩展。pg_proc项记载该聚合的名字、输入和输出数据类型，以及其他一些和普通函数类似的信息。

表 43.2. pg_aggregate 列

名称	类型	引用	描述
aggfnoid	regproc	pg_proc.oid	聚合函数的pg_proc OID
aggtransfn	regproc	pg_proc.oid	转移函数
aggfinalfn	regproc	pg_proc.oid	最终函数（如果没有就为零）
aggtranstype	oid	pg_type.oid	聚合函数内部转换（状态）数据的类型
agginitval	text		转换状态的初始值。这是一个文本字段，以外部字符串表示的形式包含初始值。如果该值为空，转换状态值从空值开始。

新的聚合函数用 CREATE AGGREGATE 命令注册。关于如何编写聚合函数以及转换函数等的含义，请参见第 33.9 节。

43.3. pg_am

目录 pg_am 存储关于索引访问方法的信息。系统支持的每种索引访问方法各有一行。

表 43.3. pg_am 列

名称	类型	引用	描述
amname	name		访问方法的名字
amowner	int4	pg_shadow.usesysid	所有者的用户 ID（目前未使用）
amstrategies	int2		该访问方法的操作符策略数
amsupport	int2		此访问方法的支持例程数
amorderstrategy	int2		如果索引不提供排序顺序则为零，否则为描述该排序顺序的策略操作符的策略编号
amcanunique	bool		访问方法是否支持唯一索引？

名称	类型	引用	描述
amcanmulticol	bool		访问方法是否支持多列索引?
amindexnulls	bool		访问方法是否支持空值索引项?
amconcurrent	bool		该访问方法是否支持并发更新?
amgettuple	regproc	pg_proc.oid	“取下一个有效元组”函数
aminsert	regproc	pg_proc.oid	“插入这个元组”函数
ambeginscan	regproc	pg_proc.oid	“开始新的扫描”函数
amrescan	regproc	pg_proc.oid	“重启此扫描”函数
amendscan	regproc	pg_proc.oid	“结束此扫描”函数
ammarkpos	regproc	pg_proc.oid	“标记当前扫描位置”函数
amrestrpos	regproc	pg_proc.oid	“恢复被标记的扫描位置”函数
ambuild	regproc	pg_proc.oid	“构建新索引”函数
ambulkdelete	regproc	pg_proc.oid	批量删除函数
amvacuumcleanup	regproc	pg_proc.oid	VACUUM之后的清理函数
amcostestimate	regproc	pg_proc.oid	估计索引扫描开销的函数

支持多列的索引访问方法（amcanmulticol 为真）必须支持在第一列之后的列上索引空值，因为规划器会假定该索引可用于只涉及第一列（前几列）的查询。例如，考虑一个建立在 (a,b) 上的索引和一个带 WHERE a = 4 的查询。系统会假定该索引可用于扫描满足 a = 4 的行；如果访问方法在 b 为空时要保留带有空 b 的项，这就错了。（这种约束之所以存在，是因为在使用索引之前不会先检查 WHERE 子句中引用了哪些列；索引扫描应当对所有可应用的情况都正确，而不是只对查询中给出的一些列。）不过，省略第一个被索引列为空的行是可以的（GiST 目前就是这样做的）。amindexnulls 只有在索引访问方法对所有行（包括空值的任意组合）都建立索引时才应设为真。

43.4. pg_amop

目录 pg_amop 存储与索引访问方法操作符类相关联的操作符的信息。操作符类中的每个操作符各有一行。

表 43.4. pg_amop 列

名称	类型	引用	描述
amopclaid	oid	pg_opclass.oid	此项所属的索引操作符类
amopstrategy	int2		操作符策略号
amopreqcheck	bool		必须重新检查索引命中
amopopr	oid	pg_operator.oid	操作符的OID

43.5. pg_amproc

目录 pg_amproc 存储与索引访问方法操作符类相关联的支持过程的信息。操作符类中的每个支持过程各有一行。

表 43.5. pg_amproc 列

名称	类型	引用	描述
amopclaid	oid	pg_opclass.oid	此项所属的索引操作符类
amprocnum	int2		支持函数编号
amproc	regproc	pg_proc.oid	函数的OID

43.6. pg_attrdef

目录pg_attrdef存储列默认值。列的主要信息存储在pg_attribute中（见下文）。只有在创建表或添加列时显式指定默认值的列，才会在这里有一项。

表 43.6. pg_attrdef 列

名称	类型	引用	描述
adrelid	oid	pg_class.oid	该列所属的表
adnum	int2	pg_attribute.attnum	列的编号
adbin	text		列默认值的内部表示
adsrc	text		适合人阅读的默认值表示

43.7. pg_attribute

目录 pg_attribute 存储关于表列的信息。数据库中每个表的每一列都恰有一行 pg_attribute 记录。（索引和其他对象也会有属性项。参见 pg_class。）

术语属性等同于列，这里使用它只是出于历史原因。

表 43.7. pg_attribute 列

名称	类型	引用	描述
attrelid	oid	pg_class.oid	此列所属的表
attname	name		列名称
atttypid	oid	pg_type.oid	此列的数据类型
attstattarget	int4		attstattarget 控制 ANALYZE 为该列积累的统计信息的详细程度。值为零表示不收集统计信息。负值表示使用系统默认的统计目标。正值的确切含义依赖于数据类型。对于标量数据类型，attstattarget 既是待收集的“最常见值”的数量目标，也是待创建的直方图桶数目标。
attlen	int2		该列数据类型的pg_type.typlen副本
attnum	int2		列的编号。一般列从1开始向上编号。系统列，如

名称	类型	引用	描述
			oid，则拥有（任意）负值编号。
attndims	int4		如果该列是数组类型，则为维数；否则为 0。（目前并不会强制检查数组维数，因此任何非零值实际上都只意味着“这是一个数组”。）
attcacheoff	int4		存储时总是 -1，但装入内存中的行描述符后，它可能被更新为该属性在行中偏移量的缓存。
atttypmod	int4		atttypmod记录了在表创建时提供的类型相关数据（例如一个varchar列的最大长度）。它会被传递给类型相关的输入函数和长度强制转换函数。对于那些不需要atttypmod的类型，这个值通常为 -1。
attbyval	bool		该列类型的pg_type.typbyval的一个拷贝
attstorage	char		通常是该列类型的pg_type.typstorage的一个拷贝。对于可TOAST的数据类型，这可以在列创建后被修改以控制存储策略。
attisset	bool		如果为真，该属性是一个集合（set）。此时，该属性中真正存储的是 pg_proc 目录中某一行的 OID。pg_proc 行包含定义该集合的查询串，即为得到该集合而要运行的查询。因此 atttypid（见上文）指的是该查询返回的类型，而该属性的实际长度是 oid 的长度（大小）。——至少理论上是如此。如今这些很可能已经相当残破了。
attalign	char		该列类型的 pg_type.typalign 的一个拷贝
attnotnull	bool		这代表一个非空约束
atthasdef	bool		该列有一个默认值，在此情况下pg_attrdef目录中会有一个对应项来真正定义该值。
attisdropped	bool		该列被删除且不再有效。一个删除的列仍然物理存在于表中，但是会被解析

名称	类型	引用	描述
			器忽略并因此无法通过SQL访问。
attislocal	bool		该列在此关系中是本地定义的。注意，一个列可能既是本地定义的又是继承来的。
attinhcount	int4		该列直接祖先的数量。祖先数量非零的列不能被删除，也不能被重命名。

43.8. pg_cast

目录 pg_cast 存储数据类型转换路径，包括内置路径和用 CREATE CAST 定义的路径。

表 43.8. pg_cast 列

名称	类型	引用	描述
castsource	oid	pg_type.oid	源数据类型的OID
casttarget	oid	pg_type.oid	目标数据类型的OID
castfunc	oid	pg_proc.oid	用于执行该转换的函数的OID。如果两种数据类型二进制兼容（即执行转换不需要运行时操作），则存储零。
castcontext	char		指示可以在哪些语境中调用该转换。e 表示只能作为显式转换（使用 CAST、:: 或函数调用语法）。a 表示在向目标列赋值时隐式使用，以及显式使用。i 表示在表达式中隐式使用，以及上述其他情况。

43.9. pg_class

目录 pg_class 登记表以及几乎所有其他具有列或类似于表的东西。这包括索引（但另见 pg_index）、序列、视图和一些特殊关系；参见 relkind。下文中，当我们指所有这些类型的对象时，就统称“关系”。并非所有列对所有关系类型都有意义。

表 43.9. pg_class 列

名称	类型	引用	描述
relname	name		表、索引、视图等的名字
relnamespace	oid	pg_namespace.oid	包含该关系的名字空间的OID
reltype	oid	pg_type.oid	与该表对应的数据类型的OID（如果有的话；索引为零，因为索引没有 pg_type 项）
relowner	int4	pg_shadow.usesysid	关系的拥有者

名称	类型	引用	描述
relam	oid	pg_am.oid	如果这是索引，则为使用的访问方法（B-树、hash 等）
relfilenode	oid		该关系在磁盘上的文件名；如果没有则为 0
relpages	int4		该表在磁盘上的表示的大小，以页（大小为 BLCKSZ）计。这只是规划器使用的一个估计值。它由 VACUUM、ANALYZE 和 CREATE INDEX 更新。
reltuples	float4		表中的行数。这只是规划器使用的一个估计值。它由 VACUUM、ANALYZE 和 CREATE INDEX 更新。
reltoastrelid	oid	pg_class.oid	与该表相关联的TOAST表的OID，如果没有则为零。TOAST表将大属性“线外”存储在一个二级表中。
reltoastidxid	oid	pg_class.oid	对于 TOAST 表，为其索引的 OID。如果不是 TOAST 表则为 0。
relhasindex	bool		如果这是一个表并且它有（或最近有过）任何索引，则为真。它由 CREATE INDEX 设置，但 DROP INDEX 不会立即清除。VACUUM 在发现表没有索引时会清除 relhasindex。
relisshared	bool		如果该表在集簇中的所有数据库间共享则为真。只有某些系统目录（如pg_database）是共享的。
relkind	char		r = 普通表，i = 索引，S = 序列，v = 视图，c = 复合类型，s = 特殊，t = TOAST 表
relnatts	int2		关系中用户列的数量（不计系统列）。在pg_attribute中必须有这么多对应项。另请参阅pg_attribute.attnum。
relchecks	int2		表上的检查约束数；参见 pg_constraint 目录
reltriggers	int2		表上的触发器数；参见 pg_trigger 目录

名称	类型	引用	描述
relukeys	int2		未使用（不是唯一键的数目）
relfkeys	int2		未使用（不是表上外键的数目）
relrefs	int2		未使用
relhasoids	bool		如果为该关系的每一行生成一个 OID，则为真。
relhaspkey	bool		如果该表有（或曾经有过）主键，则为真。
relhasrules	bool		表有规则；参见 pg_rewrite 目录
relhassubclass	bool		至少有一个表继承自该表
relacl	aclitem[]		访问权限；详见 GRANT 和 REVOKE 的说明。

43.10. pg_constraint

目录 pg_constraint 存储表上的检查、主键、唯一和外键约束。（列约束不作特殊处理。每个列约束都等价于某个表约束。）非空约束记录在 pg_attribute 目录中。

域上的检查约束也存储在这里。

表 43.10. pg_constraint 列

名称	类型	引用	描述
conname	name		约束名字（不需要唯一！）
connamespace	oid	pg_namespace.oid	包含此约束的名字空间的 OID
contype	char		c = 检查约束，f = 外键约束，p = 主键约束，u = 唯一约束
condeferrable	bool		该约束是否能被延迟？
condeferred	bool		该约束是否默认被延迟？
conrelid	oid	pg_class.oid	该约束所在的表，如果不是表约束则为零
contypid	oid	pg_type.oid	该约束所在的域，如果不是域约束则为0
confrelid	oid	pg_class.oid	如果此约束是一个外键约束，此列为被引用的表，否则为零
confupdtype	char		外键更新动作代码
confdeltype	char		外键删除动作代码
confmatchtype	char		外键匹配类型
conkey	int2[]	pg_attribute.attnum	如果是表约束，为该约束所约束的列的列表
confkey	int2[]	pg_attribute.attnum	如果是一个外键，此列是被引用列的列表

名称	类型	引用	描述
conbin	text		如果是一个检查约束，此列是表达式的一个内部表示
consrc	text		如果是检查约束，则为适合人阅读的表达式表示

注意

被引用对象发生变化时，consrc不会更新；例如，它不会跟踪列重命名。最好使用pg_get_constraintdef()提取检查约束的定义，而不是依赖这个字段。

注意

pg_class.relchecks 必须与该表中为给定关系找到的检查约束项的数目一致。

43.11. pg_conversion

目录 pg_conversion 存储编码转换信息。详见 CREATE CONVERSION。

表 43.11. pg_conversion 列

名称	类型	引用	描述
conname	name		转换的名字（在一个名字空间内唯一）
connamespace	oid	pg_namespace.oid	包含此转换的名字空间的OID
conowner	int4	pg_shadow.usesysid	转换的拥有者
conforencoding	int4		源编码ID
contoencoding	int4		目标编码ID
conproc	regproc	pg_proc.oid	转换函数
condefault	bool		如果这是默认转换则为真

43.12. pg_database

目录 pg_database 存储可用数据库的信息。数据库用 CREATE DATABASE 命令创建。有关某些参数含义的细节，请参见第 18 章。

和大部分系统目录不同，pg_database
是在集簇的所有数据库之间共享的：在一个集簇中只有一份pg_database拷贝，
而不是每个数据库一份。

表 43.12. pg_database 列

名称	类型	引用	描述
datname	name		数据库名称

名称	类型	引用	描述
datdba	int4	pg_shadow.usesysid	数据库的拥有者，通常是创建它的用户
encoding	int4		该数据库的字符编码
datistemplate	bool		如果为真，则该数据库可以在 CREATE DATABASE 的 TEMPLATE 子句中使用，以创建一个作为该数据库克隆的新数据库。
datallowconn	bool		如果为假则没有人能连接到这个数据库。这可以用来保护template0数据库不被修改。
datlastsysoid	oid		数据库中最后一个系统OID；对 pg_dump 尤其有用
datvacuumxid	xid		在此之前的事务 ID 插入或删除的所有行，在该数据库中都已被标记为已知已提交或已知已中止。这用于确定何时可以回收提交日志空间。
datfrozenxid	xid		在此之前的事务 ID 插入的所有行，在该数据库中都已被重新标记为一个永久的（“冻结”的）事务 ID。这有助于检查是否必须尽快对数据库进行 VACUUM，以避免事务 ID 回卷问题。
datpath	text		如果该数据库存储在一个替代位置，则此字段记录该位置。它既可能是环境变量名，也可能是绝对路径，取决于当时的输入方式。
datconfig	text[]		运行时配置变量的会话默认值
datacl	aclitem[]		访问权限

43.13. pg_depend

目录pg_depend记录数据库对象之间的依赖关系。这些信息允许DROP命令查找必须被DROP CASCADE删除的其他对象，或者在DROP RESTRICT情况下阻止删除。

表 43.13. pg_depend 列

名称	类型	引用	描述
classid	oid	pg_class.oid	依赖对象所在系统目录的OID
objid	oid	任意 OID 列	特定依赖对象的OID

名称	类型	引用	描述
objsubid	int4		对于一个表列，这里是列号（objid和classid指表本身）。对于所有其他对象类型，此列为0。
refclassid	oid	pg_class.oid	被引用对象所在的系统目录的OID
refobjid	oid	任意 OID 列	指定被引用对象的OID
refobjsubid	int4		对于一个表列，这里是列号（refobjid和refclassid指表本身）。对于所有其他对象类型，此列为0。
deptype	char		定义该依赖关系具体语义的代码；见正文。

在所有情况下，一个 pg_depend 项都表明：被引用对象不能在不删除依赖对象的情况下被删除。不过，由 deptype 标识的依赖有几种子类型：

DEPENDENCY_NORMAL (n)

分开创建的对象之间的正常关系。可以不影响被引用对象而删除依赖对象。被引用对象只能通过指定 CASCADE 删除依赖对象来删除。例如，表列对其数据类型有一个 normal 依赖。

DEPENDENCY_AUTO (a)

依赖对象可以独立于被引用对象删除，并且在被引用对象被删除时应自动删除（无论 RESTRICT 还是 CASCADE 模式）。例如，表上的一个命名约束对表是 autodependent 的，这样表被删除时它也会消失。

DEPENDENCY_INTERNAL (i)

依赖对象是作为创建被引用对象的一部分而创建的，实际上只是其内部实现的一部分。对依赖对象的 DROP 将被直接禁止（我们会让用户改为对被引用对象发出 DROP）。无论是否指定 CASCADE，对被引用对象的 DROP 都会传播下去删除依赖对象。例如，被视为包含 pg_class 行的 TOAST 表对主表有 internal 依赖。

DEPENDENCY_PIN (p)

没有依赖对象；这种类型的项是一个信号，表明系统自身依赖于被引用对象，因此该对象永远不能被删除。这种类型的项只能由 initdb 创建。依赖对象各列的值为零。

未来的版本中可能还需要其他种类的依赖。

43.14. pg_description

目录 pg_description 可以为每个数据库对象存储可选的描述或注释。描述可以用 COMMENT 命令操作，并用 psql 的 \d 命令查看。许多内置系统对象的描述在 pg_description 的初始内容中提供。

表 43.14. pg_description 列

名称	类型	引用	描述
objoid	oid	任意 OID 列	描述所属对象的OID

名称	类型	引用	描述
classoid	oid	pg_class.oid	该对象所在的系统目录的OID
objsubid	int4		对于一个表列上的一个注释，这里是列号（objoid和classoid指表本身）。对所有其他对象类型，此列为0。
description	text		作为该对象描述的任意文本。

43.15. pg_group

目录 pg_group 定义组，并存储哪些用户属于哪些组。组使用 CREATE GROUP 命令创建。有关用户权限管理的信息，请参阅第 17 章。

由于用户和组的标识是集群范围的，pg_group 在一个集群的所有数据库之间共享：每个集群只有一份 pg_group 的副本，而不是每个数据库一份。

表 43.15. pg_group 列

名称	类型	引用	描述
groname	name		组的名称
grosysid	int4		标识该组的一个任意编号
grolist	int4[]	pg_shadow.usesysid	包含该组中各用户 ID 的数组

43.16. pg_index

目录pg_index包含关于索引的部分信息。其他信息大部分在pg_class中。

表 43.16. pg_index 列

名称	类型	引用	描述
indexrelid	oid	pg_class.oid	此索引的pg_class项的OID
indrelid	oid	pg_class.oid	此索引所对应的基表的pg_class项的OID
indkey	int2vector	pg_attribute.attnum	这是一个由 indnatts（至多 INDEX_MAX_KEYS）个值组成的数组，指明该索引索引了哪些表列。例如，值 1 3 表示第一和第三个表列构成索引键。数组中的零表示相应的索引属性是对表列的表达式，而不是简单的列引用。
indclass	oidvector	pg_opclass.oid	对于索引键中的每一列，这里包含要使用的操作符类的 OID。详见 pg_opclass。

名称	类型	引用	描述
indnatts	int2		索引中的总列数（与pg_class.relnatts重复）
indisunique	bool		如果为真，这是一个唯一索引。
indisprimary	bool		如果为真，该索引表示表的主键。（此为真时indisunique 应总是为真。）
indisclustered	bool		如果为真，表上一次CLUSTER 是基于该索引进行的。
indexprs	text		非简单列引用索引属性的表达式树（以nodeToString()形式）。对于indkey中每一个为0的项，这个列表中都有一个元素。如果所有的索引属性都是简单引用，此列为空。
indpred	text		部分索引谓词的表达式树（以nodeToString()形式）。如果不是部分索引，此列为空。

43.17. pg_inherits

目录 pg_inherits 记录关于表继承层次结构的信息。

表 43.17. pg_inherits 列

名称	类型	引用	描述
inhrelid	oid	pg_class.oid	子表的 OID。
inhparent	oid	pg_class.oid	父表的 OID。
inhseqno	int4		如果一个子表有多个父表（多重继承），此数字指出继承列的排列顺序。计数从 1 开始。

43.18. pg_language

目录 pg_language 登记可以用来编写函数或存储过程的调用接口或语言。有关语言处理器的更多信息，请参见 CREATE LANGUAGE 和第 36 章。

表 43.18. pg_language 列

名称	类型	引用	描述
lanname	name		语言的名称（创建函数时要指定）

名称	类型	引用	描述
lanispl	bool		对于内部语言（如 SQL）为假，对于用户定义的语言为真。目前 pg_dump 仍用它来确定需要转储哪些语言，但将来可能被别的机制取代。
lanpltrusted	bool		这是一个可信语言。其含义参见 CREATE LANGUAGE。如果这是内部语言（lanispl 为假），此列无意义。
lanplcallfoid	oid	pg_proc.oid	对于非内部语言，它引用语言处理器，即一个负责执行用该语言编写的所有函数的特殊函数。
lanvalidator	oid	pg_proc.oid	它引用一个语言验证器函数，负责在创建新函数时检查其语法和有效性。有关验证器的更多信息，参见 CREATE LANGUAGE。
lanacl	aclitem[]		访问权限

43.19. pg_largeobject

目录pg_largeobject保存构成“大对象”的数据。一个大对象在被创建时会被分配一个OID。每个大对象被分解成足够小的段或“页”，以便方便地作为行存储在pg_largeobject中。每页中的数据量被定义为LOBLKSIZE（目前是BLCKSZ/4，通常是 2 kB）。

表 43.19. pg_largeobject 列

名称	类型	引用	描述
loid	oid		包含此页的大对象的标识符
pageno	int4		此页在它所属大对象中的页号（从0开始计）
data	bytea		大对象中实际存储的数据。绝不会超过 LOBLKSIZE 字节，也可能少于该值。

pg_largeobject 的每一行保存大对象一页的数据，从对象内的字节偏移（pageno * LOBLKSIZE）开始。该实现允许稀疏存储：某些页可能缺失，即使不是对象的最后一页，也可能短于 LOBLKSIZE 字节。大对象中缺失的区域读取为零。

43.20. pg_listener

目录 pg_listener 支持 LISTEN 和 NOTIFY 命令。监听者会为它正在监听的每个通知名称在 pg_listener 中创建一个项。通知者扫描 pg_listener 并更新每个匹配的项，以表明已发生一个通知。通知者还会（使用表中记录的 PID）发送一个信号，把监听者从睡眠中唤醒。

表 43.20. pg_listener 列

名称	类型	引用	描述
relname	name		通知条件名称。（该名称不必与数据库中的任何实际关系匹配；relname 这个名字是历史遗留的。）
listenerpid	int4		创建此项的服务器进程的 PID。
notification	int4		如果该监听者没有待处理的事件，则为零。如果有事件待处理，则为发送通知的服务器进程的 PID。

43.21. pg_namespace

目录pg_namespace存储名字空间。
名字空间是SQL模式之下的结构：每个名字空间拥有一个独立的表、类型等的集合，
且其中没有名字冲突。

表 43.21. pg_namespace 列

名称	类型	引用	描述
nspname	name		名字空间的名字
nspowner	int4	pg_shadow.usesysid	名字空间的拥有者
nspacl	aclitem[]		访问权限

43.22. pg_opclass

目录 pg_opclass 定义索引访问方法操作符类。每个操作符类为特定数据类型和特定索引访问方法的索引列定义语义。注意，一个给定的数据类型/访问方法组合可以有多个操作符类，
从而支持不同的行为或语义。

操作符类详见第 33.13 节。

表 43.22. pg_opclass 列

名称	类型	引用	描述
opcamid	oid	pg_am.oid	索引访问方法操作符类所属
opcname	name		操作符类的名称
opcnamespace	oid	pg_namespace.oid	操作符类所属的名字空间
opcowner	int4	pg_shadow.usesysid	操作符类所有者
opcintype	oid	pg_type.oid	操作符类的输入数据类型
opcdefault	bool		如果此操作符类为opcintype的默认值则为真

名称	类型	引用	描述
opckeytype	oid	pg_type.oid	索引数据的类型，如果与 opcintype 相同则为零

定义操作符类的信息大部分实际上不在它的 pg_opclass 行中，而在 pg_amop 和 pg_amproc 中相关联的行里。这些行被视为操作符类定义的一部分--这与一个关系由单个 pg_class 行加上多个 pg_attribute 行定义的方式不无相似之处。

43.23. pg_operator

目录 pg_operator 存储关于操作符的信息。这些操作符参数的细节请参见 CREATE OPERATOR 和第 33.11 节。

表 43.23. pg_operator 列

名称	类型	引用	描述
oprname	name		操作符的名称
oprnamespace	oid	pg_namespace.oid	操作符所属的名字空间的 OID
oprowner	int4	pg_shadow.usesysid	操作符的拥有者
oprkind	char		b = 中缀（“双目”），l = 前缀（“左目”），r = 后缀（“右目”）
oprcanhash	bool		该操作符支持哈希连接
oprleft	oid	pg_type.oid	左操作数类型
oprright	oid	pg_type.oid	右操作数类型
oprresult	oid	pg_type.oid	结果类型
oprcom	oid	pg_operator.oid	该操作符的交换子（如有）
oprnegate	oid	pg_operator.oid	该操作符的否定（如有）
oprlsortop	oid	pg_operator.oid	如果该操作符支持归并连接，则为对左操作数类型排序的操作符（L<L）
oprrsortop	oid	pg_operator.oid	如果该操作符支持归并连接，则为对右操作数类型排序的操作符（R<R）
oprltcmpop	oid	pg_operator.oid	如果该操作符支持归并连接，则为比较左右操作数类型的小于操作符（L<R）
oprgtcmpop	oid	pg_operator.oid	如果该操作符支持归并连接，则为比较左右操作数类型的大于操作符（L>R）
oprcode	regproc	pg_proc.oid	实现该操作符的函数
oprrest	regproc	pg_proc.oid	该操作符的限制选择率估算函数
oprjoin	regproc	pg_proc.oid	该操作符的连接选择率估算函数

未使用的列包含零，例如前缀操作符的 oprleft 为零。

43.24. pg_proc

目录 pg_proc 存储关于函数（或过程）的信息。CREATE FUNCTION 的描述和第 33.3 节包含某些列含义的更多信息。

该表同时包含聚合函数和普通函数的数据。如果proisagg为真，在pg_aggregate中应该有一个相匹配的行。

表 43.24. pg_proc 列

名称	类型	引用	描述
proname	name		函数的名字
pronamespace	oid	pg_namespace.oid	函数所属的名字空间的 OID
proowner	int4	pg_shadow.usesysid	函数的拥有者
prolang	oid	pg_langauge.oid	实现语言或该函数的调用接口
proisagg	bool		该函数是聚合函数
prosecdef	bool		函数是一个安全性定义者（即，一个“setuid”函数）
proisstrict	bool		当任一调用参数为 NULL 时，函数是否返回空值。在这种情况下，函数实际上根本不会被调用。非“strict”函数必须准备好处理空值输入。
proretset	bool		函数是否返回一个集合（即，指定数据类型的多个值）
provolatile	char		provolatile 说明函数的结果是只依赖于输入参数，还是受外部因素影响。i 表示“不可变”（immutable）函数，对相同输入总是给出相同结果。s 表示“稳定”（stable）函数，其结果（对固定输入）在一次扫描内不会变化。v 表示“易变”（volatile）函数，其结果任何时候都可能变化。（有副作用的函数也用 v，这样对它们的调用就不会被优化掉。）
pronargs	int2		参数个数
prorettype	oid	pg_type.oid	返回值的数据类型
proargtypes	oidvector	pg_type.oid	一个由函数参数的数据类型组成的数组
prosrc	text		这个字段告诉函数处理器如何调用该函数。它可能

名称	类型	引用	描述
			是针对解释型语言的真实源码、一个链接符号、一个文件名，或任何其他取决于实现语言/调用约定的内容。
probin	bytea		关于如何调用函数的附加信息。其解释是与语言相关的。
proacl	aclitem[]		访问权限

对于编译型函数（包括内置的和动态装载的），prosrc 包含函数的 C 语言名（链接符号）。对于所有其他语言类型，prosrc 包含函数的源文本。probin 只用于动态装载的 C 函数，此时它给出共享库的名称。

43.25. pg_rewrite

目录pg_rewrite存储对于表和视图的重写规则。

表 43.25. pg_rewrite 列

名称	类型	引用	描述
rulename	name		规则名称
ev_class	oid	pg_class.oid	使用该规则的表
ev_attr	int2		该规则针对的列（目前总是零，表示整个表）
ev_type	char		使用该规则的事件类型：1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE
is_instead	bool		为真表示是一个INSTEAD规则
ev_qual	text		规则条件的表达式树（以nodeToString() 表示）
ev_action	text		规则动作的查询树（以nodeToString() 表示）

注意

如果一个表在这个目录中有任何规则，pg_class.relhasrules必须为真。

43.26. pg_shadow

目录 pg_shadow 包含数据库用户的信息。这个名字源于这样一个事实：由于该表包含口令，它不应对公众可读。pg_user 是 pg_shadow 上一个对公众可读的视图，它把口令字段置为空白。

第 17 章包含有关用户和权限管理的详细信息。

由于用户标识是集群范围的，pg_shadow 在一个集群的所有数据库之间共享：每个集群只有一份 pg_shadow 的副本，而不是每个数据库一份。

表 43.26. pg_shadow 列

名称	类型	引用	描述
usename	name		用户名
usesysid	int4		用户 ID（用于引用该用户的任意编号）
usecreatedb	bool		用户可以创建数据库
usesuper	bool		用户是超级用户
usecatupd	bool		用户可以更新系统目录。（即使是超级用户，也只有当此列为真时才能这样做。）
passwd	text		口令
valuntil	abstime		账号过期时间（仅用于口令认证）
useconfig	text[]		运行时配置变量的会话默认值

43.27. pg_statistic

目录 pg_statistic 存储关于数据库内容的统计数据。项由 ANALYZE 创建，随后被查询规划器使用。每个已分析的表列各有一项。注意，即使假设统计信息是完全最新的，所有统计数据本质上也是近似的。

由于不同种类的数据可能适合不同种类的统计信息，pg_statistic 的设计不对它存储的统计种类做太多假设。只有极其通用的统计（如空值率）在 pg_statistic 中有专门的列。其余所有内容都存储在“槽位”中，槽位是相关联的列组，其内容由槽位各列之一中的代码编号标识。更多信息见 src/include/catalog/pg_statistic.h。

pg_statistic 不应对公众可读，因为即使是关于表内容的统计信息也可能被视为敏感。（例如：工资列的最小值和最大值可能相当引人关注。）pg_stats 是 pg_statistic 上一个对公众可读的视图，它只公开每个人都能看到的信息。

表 43.27. pg_statistic 列

名称	类型	引用	描述
starelid	oid	pg_class.oid	所描述列所属的表
staattnum	int2	pg_attribute.attnum	被描述列的编号
stanullfrac	float4		列中空值所占的比例
stawidth	int4		非空项的平均存储宽度，以字节计
stadistinct	float4		列中不同非空数据值的数目。大于零的值是不同值的实际数目。小于零的值是表中行数的一个分数的相反数（例如，值平均出现约两次的列可以用

名称	类型	引用	描述
			stadistinct = -0.5 表示）。零表示不同值的数目未知。
stakind*N*	int2		一个代码，它表示存储在该pg_statistic行中第*N*个“槽位”的统计类型。
staop*N*	oid	pg_operator.oid	一个用于生成这些存储在第*N*个“槽位”的统计信息的操作符。例如，直方图槽位会使用<操作符，该操作符定义了这些数据的排序顺序。
stanumbers*N*	float4[]		第 *N* 个“槽位”的相应种类的数值统计；如果该槽位种类不涉及数值则为空。
stavalues*N*	anyarray		第*N*个“槽位”的类型的列值，如果该槽位类型不存储任何数据值则为NULL。每个数组的元素值实际上都是指定列的数据类型或者是一个相关类型（如数组元素类型），因此，无法把这些列的类型定义得比anyarray更具体。

43.28. pg_trigger

目录 pg_trigger 存储表上的触发器。更多信息参见 CREATE TRIGGER。

表 43.28. pg_trigger 列

名称	类型	引用	描述
tgrelid	oid	pg_class.oid	触发器所在的表
tgname	name		触发器名（在同一个表的触发器中必须唯一）
tgfoid	oid	pg_proc.oid	要被触发器调用的函数
tgtype	int2		标识触发器触发条件的位掩码
tgenabled	bool		如果触发器启用则为真（目前并非在所有应检查的地方都做了检查，因此通过把它设为假来禁用触发器并不可靠）
tgisconstraint	bool		如果触发器实现一个引用完整性约束则为真
tgconstrname	name		引用完整性约束名
tgconstrrelid	oid	pg_class.oid	引用完整性约束所引用的表
tgdeferrable	bool		如果可推迟则为真

名称	类型	引用	描述
tginitdeferred	bool		如果初始为推迟则为真
tgnargs	int2		传递给触发器函数的参数字符串个数
tgattr	int2vector		目前未使用
tgargs	bytea		传给触发器的参数字符串，每个以空字符结尾

注意

pg_class.reltriggers 必须与该表中的项数一致。

43.29. pg_type

目录 pg_type 存储关于数据类型的信息。基本类型（标量类型）用 CREATE TYPE 创建。数据库中的每个表都会自动创建一个复合类型，以表示该表的行结构。也可以用 CREATE TYPE AS 显式创建复合类型。（还可以用 CREATE DOMAIN 在现有类型之上创建域。）

表 43.29. pg_type 列

名称	类型	引用	描述
typname	name		数据类型的名字
typnamespace	oid	pg_namespace.oid	包含此类型的名字空间的OID
typowner	int4	pg_shadow.usesysid	类型的拥有者
typlen	int2		对于一个固定尺寸的类型，typlen是该类型内部表示的字节数。 对于一个变长类型，typlen为负值。-1表示一个 “varlena” 类型（具有长度字），-2表示一个以空值结尾的C字符串。
typbyval	bool		typbyval 决定内部例程按值还是按引用传递该类型的值。如果 typlen 不是 1、2 或 4（或在 Datum 为 8 字节的机器上不是 8），typbyval 最好为假。变长类型总是按引用传递。注意，即使长度允许按值传递，typbyval 也可以为假；例如 float4 类型目前就是这样。
typtype	char		typtype 为 b 表示基本类型，c 表示复合类型（即表的行类型），d 表示域，p 表示伪类型。另见 typrelid 和 typbasetype。

名称	类型	引用	描述
typisdefined	bool		如果此类型已被定义则为真，如果此类型只是一个表示还未定义类型的占位符则为假。当typisdefined为假，除了类型名字、名字空间和OID之外什么都不能被依赖。
typdelim	char		在分析数组输入时，分隔两个此类型值的字符。注意该分隔符是与数组元素数据类型相关联的，而不是和数组的数据类型关联。
typrelid	oid	pg_class.oid	如果这是复合类型（见typtype），此列指向定义相应表的 pg_class 项。（对于独立的复合类型，该 pg_class 项并不真正代表一个表，但类型的 pg_attribute 项需要链接到它。）基本类型为零。
typelem	oid	pg_type.oid	如果 typelem 不为 0，它就标识 pg_type 中的另一行。当前类型于是可以像数组那样带下标，产生 typelem 类型的值。“真正的”数组类型是变长的（typlen = -1），但某些定长（typlen > 0）类型也有非零的 typelem，例如 name 和 oidvector。如果定长类型有 typelem，其内部表示必须是若干个 typelem 数据类型的值，不含其他数据。变长数组类型有一个由数组子例程定义的头部。
typinput	regproc	pg_proc.oid	输入转换函数（文本格式）
typoutput	regproc	pg_proc.oid	输出转换函数（文本格式）
typreceive	regproc	pg_proc.oid	输入转换函数（二进制格式），如果没有则为零
typsend	regproc	pg_proc.oid	输出转换函数（二进制格式），如果没有则为零
typalign	char		typalign表示存储此类型值时所要求的对齐方式。它适用于磁盘存储，以及该值在 PostgreSQL 内部的大多数表示形式。当多个值连续存放时，例如在磁盘上完整行的表示

名称	类型	引用	描述
			中，这种类型的数据前会插入填充，使其从指定边界开始。对齐参考点是该序列中第一个数据的起始位置。 可能的值为： • c = char对齐，即不需要对齐。 • s = short对齐（在大部分机器上为2字节）。 • i = int对齐（在大部分机器上为4字节）。 • d = double对齐（在很多机器上为8字节，但绝不是全部）。 **注意** 对于系统表中使用的类型，关键的是在pg_type定义的大小和对齐与编译器在表示表行的结构体中布局列的方式一致。
typstorage	char		typstorage用于指示 varlena 类型（即typlen = -1）是否支持TOAST，以及这种类型的属性应采用什么默认策略。可能的值为： • p：值必须始终以普通方式存储。 • e：值可以存储在一个“二级”关系中（如果关系有这样的关系，参见pg_class.reltoastrelid）。 • m：值可以被压缩并内联存储。 • x：值可以被压缩并内联存储，或存储在“二级”存储中。 注意，m列也可以被移动到二级存储，但只能作为

名称	类型	引用	描述
			最后手段（e和x列会先被移动）。
typnotnull	bool		typnotnull表示类型上的一个非空约束。只用于域。
typbasetype	oid	pg_type.oid	如果这是域（见 typtype），typbasetype 标识该域所基于的类型。不是域则为零。
typtypmod	int4		域使用typtypmod来记录被应用于它们基础类型的typmod（如果基础类型不使用typmod，则为-1）。如果此类型不是一个域则为-1。
typndims	int4		typndims 是作为数组的域的数组维数（即 typbasetype 是数组类型；该域的 typelem 将与基本类型的 typelem 一致）。非数组域之外的其他类型为零。
typdefaultbin	text		如果typdefaultbin非空，那么它就是该类型默认表达式的nodeToString()表示形式。此列只用于域。
typdefault	text		typdefault 为空表示该类型没有关联的默认值。如果 typdefaultbin 非空，typdefault 必须包含 typdefaultbin 所表示的默认表达式的人类可读版本。如果 typdefaultbin 为空而 typdefault 非空，则 typdefault 是该类型默认值的外部表示，可以交给该类型的输入转换器以产生一个常量。

43.30. 系统视图

除系统目录之外，PostgreSQL 还提供许多内置视图。系统视图为一些常用的系统目录查询提供了方便的访问方式。其中一些视图还提供对服务器内部状态的访问。

表 43.30列出了这里描述的系统视图。每个视图的更详细文档见下文。还有一些额外的视图提供了对统计信息收集器结果的访问；它们在表 23.1中描述。

信息模式（第 32 章）提供了一组与系统视图功能重叠的替代视图。由于信息模式是SQL标准，而这里描述的视图是PostgreSQL特有的，所以如果信息模式提供了所需的全部信息，通常最好使用信息模式。

除非另有说明，否则这里描述的所有视图都是只读的。

表 43.30. 系统视图

视图名	用途
pg_indexes	索引
pg_locks	当前持有的锁
pg_rules	规则
pg_settings	参数设置
pg_stats	规划器统计信息
pg_tables	表
pg_user	数据库用户
pg_views	视图

43.31. pg_indexes

视图pg_indexes提供了有关数据库中每个索引的有用信息。

表 43.31. pg_indexes 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表和索引的模式名
tablename	name	pg_class.relname	索引所属表的名称
indexname	name	pg_class.relname	索引名
indexdef	text		索引定义（重构出的创建命令）

43.32. pg_locks

视图pg_locks提供了对数据库服务器中打开的事务持有的锁的信息的访问。
有关锁定的更多讨论，请参见第 12 章。

pg_locks 为每个活跃的可锁定对象、所请求的锁模式和相关的事务各包含一行。因此，如果多个事务持有或等待同一可锁定对象上的锁，该对象可能出现多次。不过，
当前没有锁的对象根本不会出现。一个可锁定对象要么是一个关系，要么是一个事务 ID。

注意该视图只包括表级锁，不包括行级锁。如果一个事务在等待行级锁，
它在视图中表现为在等待持有该行锁的事务的事务 ID。

表 43.32. pg_locks 列

名称	类型	引用	描述
relation	oid	pg_class.oid	被锁定关系的 OID，如果可锁定对象是事务 ID 则为 NULL
database	oid	pg_database.oid	被锁定关系所在的数据库的 OID；如果被锁定关系

名称	类型	引用	描述
			是全局共享表则为零；如果可锁定对象是事务 ID 则为 NULL
transaction	xid		事务的 ID，如果可锁定对象是关系则为 NULL
pid	integer		持有或等待该锁的服务器进程的进程 ID
mode	text		该进程持有或想要的锁模式名（见第 12.3.1 节）
granted	boolean		如果锁已持有则为真，如果在等待锁则为假

在表示某会话持有的锁的行中，granted 为真。为假表示该会话当前正在等待获取该锁，这蕴含着某个其他会话在同一可锁定对象上持有一个冲突的锁模式。等待的会话将休眠，直到其他会话释放该锁（或检测到死锁条件）。一个会话在任意时刻最多只能等待获取一个锁。

每个事务在其整个持续时间内都持有其事务 ID 上的一个排他锁。如果一个事务发现必须专门等待另一个事务，它会通过对另一个事务 ID 尝试获取共享锁来实现。这只有在另一个事务结束并释放其事务 ID 锁之后才会成功。

访问 pg_locks 视图时，内部锁管理器的数据结构会被短暂加锁，并制作一份副本供该视图显示。这确保了视图产生一致的结果集，同时不会比必要的更长时间地阻塞正常的锁管理器操作。尽管如此，还是存在一些与性能相关的注意事项：对该视图的频繁访问可能会对数据库性能产生一些影响。

pg_locks提供了集簇中所有锁的全局视图，不仅包括与当前数据库相关的锁。虽然它的relation列可以与pg_class.oid 连接来识别被锁定的关系，但这仅对当前数据库中的关系有效（即 database 列为当前数据库的 OID 或零的那些关系）。

如果启用了统计收集器，pid 列可以连接到 pg_stat_activity 视图的 procpid 列，以获得关于持有或等待持有该锁的会话的更多信息。

43.33. pg_rules

视图pg_rules提供了有关查询重写规则的有用信息。

表 43.33. pg_rules 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式名
tablename	name	pg_class.relname	规则所属表的名称
rulename	name	pg_rewrite.rulename	规则名
definition	text		规则定义（重构出的创建命令）

pg_rules视图排除了视图的 ON SELECT 规则；那些规则可以在pg_views中看到。

43.34. pg_settings

视图 pg_settings 提供对服务器运行时参数的访问。它实质上是 SHOW 和 SET 命令的一种替代接口。它还提供对每个参数一些无法直接从 SHOW 得到的事实的访问，例如最小值和最大值。

表 43.34. pg_settings 列

名称	类型	引用	描述
name	text		运行时配置参数名
setting	text		参数的当前值
context	text		设置该参数的值所需的语境
vartype	text		参数类型（bool、integer、real 或 string）
source	text		当前参数值的来源
min_val	text		参数允许的最小值（非数值类型为 NULL）
max_val	text		参数允许的最大值（非数值类型为 NULL）

pg_settings视图不能插入或删除，但可以更新。对 pg_settings 的某一行执行 UPDATE，等同于对该命名参数执行SET命令。更改只影响当前会话所用的值。
如果在之后被中止的事务中发出了 UPDATE，那么事务回滚时 UPDATE 命令的效果也会消失。一旦外围事务提交，这些效果会持续到会话结束，除非被另一个 UPDATE 或 SET 覆盖。

43.35. pg_stats

视图pg_stats提供对存储在pg_statistic目录中信息的访问。
此视图仅允许访问用户具有读取权限的表对应的pg_statistic行，
因此可以安全地允许对此视图进行公共读取访问。

pg_stats 的设计还在于以比底层目录更易读的形式呈现信息——代价是每当为 pg_statistic 定义新的槽位种类时，它的模式都必须扩展。

表 43.35. pg_stats 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式名
tablename	name	pg_class.relname	表名
attname	name	pg_attribute.attname	该行所描述列的名称
null_frac	real		列项中为空的比例
avg_width	integer		列项的平均宽度（字节）
n_distinct	real		如果大于零，表示列中可区分值的估计个数。如果小于零，是可区分值个数除以行数的负值（当ANALYZE认为可区分值的数量会随着表增长而增加时采用负值的形式，而如果认为列具有固定数量

名称	类型	引用	描述
			的可选值时采用正值的形式）。例如，-1表示一个唯一列，即其中可区分值的个数等于行数。
most_common_vals	anyarray		列中最常见值的列表。（如果没有值看起来比其他值更常见，则为NULL。）
most_common_freqs	real[]		最常见值出现频率的列表，即每个值的出现次数除以总行数。（most_common_vals 为 NULL 时也为 NULL。）
histogram_bounds	anyarray		把列的值划分成人口大致相等的组的一些值。most_common_vals 中的值（如果存在）会从该直方图计算中剔除。（如果列数据类型没有 < 操作符，或者 most_common_vals 列表占了全部人口，此列为NULL。）
correlation	real		物理行序与列值逻辑序之间的统计相关度。取值范围从 -1 到 +1。当值接近 -1 或 +1 时，对该列的索引扫描会被估计为比接近零时更便宜，因为对磁盘的随机访问减少了。（如果列数据类型没有 < 操作符，此列为 NULL。）

most_common_vals 和 histogram_bounds 数组中的最大项数可以逐列用 ALTER TABLE SET STATISTICS 命令设置，或者全局地通过设置运行时参数 default_statistics_target 来设置。

43.36. pg_tables

视图pg_tables提供了对数据库中每个表的有用信息的访问。

表 43.36. pg_tables 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式名
tablename	name	pg_class.relname	表名
tableowner	name	pg_shadow.usename	表所有者的名称
hasindexes	boolean	pg_class.relhasindex	如果表有（或最近有过）任何索引则为真
hasrules	boolean	pg_class.relhasrules	如果表有规则则为真

名称	类型	引用	描述
hastriggers	boolean	pg_class.reltriggers	如果表有触发器则为真

43.37. pg_user

视图pg_user提供数据库用户的信息。它本质上是 pg_shadow 的公开可读视图，并隐藏密码字段。

表 43.37. pg_user 列

名称	类型	引用	描述
usename	name		用户名
usesysid	int4		用户 ID（用于引用该用户的任意编号）
usecreatedb	bool		用户可以创建数据库
usesuper	bool		用户是超级用户
usecatupd	bool		用户可以更新系统目录。（即使是超级用户，也只有当此列为真时才能这样做。）
passwd	text		不是口令（总是读作 ********）
valuntil	abstime		账号过期时间（仅用于口令认证）
useconfig	text[]		运行时配置变量的会话默认值

43.38. pg_views

视图pg_views提供了对数据库中每个视图的有用信息的访问。

表 43.38. pg_views 列

名称	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含视图的模式名
viewname	name	pg_class.relname	视图名
viewowner	name	pg_shadow.usename	视图所有者的名称
definition	text		视图定义（重构出的 SELECT 查询）

第 44 章 前端/后端协议

PostgreSQL使用一种基于消息的协议在前端和后端（客户端与服务器）之间进行通信。
该协议既支持TCP/IP，也支持 Unix 域套接字。端口号 5432 已在 IANA 注册为支持该协议的服务器的惯用 TCP 端口号，但实际上任何非特权端口号都可以使用。

本文档描述协议 3.0 版，它在 PostgreSQL 7.4 及以后版本中实现。早期协议版本的描述参见 PostgreSQL 文档的以前发行版。单个服务器可以支持多个协议版本。最初的启动请求消息告诉服务器客户端试图使用哪个协议版本，然后服务器在有能力的情况下遵循该协议。

建立在该协议之上的更高层特性（例如 libpq 如何在建立连接时传递某些环境变量）将在别处介绍。

为了高效地为多个客户端提供服务，服务器会为每个客户端启动一个新的“后端”进程。
在当前实现中，一旦检测到传入连接，就会立刻创建新的子进程。不过，
这一点对协议而言是透明的。就协议而言，术语“后端”和“服务器”可以互换；同样，“前端”和“客户端”也可以互换。

44.1. 概述

协议分为启动和正常操作两个阶段。在启动阶段，前端打开到服务器的连接，
并完成服务器所要求的认证。（这可能只涉及一条消息，
也可能因所用认证方法不同而需要多条消息。）如果一切顺利，
服务器随后会向前端发送状态信息，并最终进入正常操作。除最初的启动请求消息外，
协议的这一部分由服务器驱动。

在正常操作中，前端向后端发送查询及其他命令，后端则返回查询结果和其他响应。
少数情况下（例如 `NOTIFY`），后端会发送未请求的消息，但会话中的绝大多数交互仍由前端请求驱动。

会话通常由前端选择终止，但在某些情况下也可能由后端强制终止。无论哪种情况，
后端关闭连接时，都会在退出前回滚所有打开的（未完成的）事务。

在正常操作中，SQL 命令可以通过两种子协议之一执行。在“简单查询”协议中，前端只需发送文本形式的查询字符串，后端会立即对其进行解析并执行。在“扩展查询”协议中，
查询处理被拆分为多个步骤：解析、参数值绑定以及执行。这带来了更高的灵活性和性能收益，
但代价是额外的复杂性。

正常操作还包含用于`COPY`等特殊操作的额外子协议。

44.1.1. 消息传送概述

所有通信都通过消息流进行。消息的第一个字节标识消息类型，
接下来的四个字节给出消息其余部分的长度（该长度计数包含自身，但不包括消息类型字节）。
消息剩余内容由消息类型决定。由于历史原因，客户端发送的第一条消息（启动消息）
没有开头的消息类型字节。

为了避免与消息流失去同步，服务器和客户端通常都会先根据字节计数把整条消息读入缓冲区，
然后再处理其内容。这样一来，如果在处理内容时检测到错误，就比较容易恢复。
在极端情况下（例如没有足够内存缓冲整条消息），
接收方也可以利用字节计数判断在恢复读取消息之前需要跳过多少输入。

反过来，服务器和客户端也必须注意绝不能发送不完整的消息。通常的做法是在开始发送之前，先在缓冲区中整理好整条消息。如果在发送或接收消息的途中发生通信故障，唯一合理的做法就是放弃连接，因为几乎不可能重新恢复消息边界的同步。

44.1.2. 扩展查询概述

在扩展查询协议中，SQL 命令的执行被划分为多个步骤。各步骤之间保留的状态由两类对象表示：预备语句和 portal。预备语句表示对文本查询字符串完成解析和语义分析（可选地还有规划）后的结果。预备语句本身还不一定能直接执行，因为它可能缺少特定参数的值。portal 表示一条已经可以执行、或已经部分执行过的语句，其中所有缺失的参数值都已补齐。（对于 `SELECT` 语句，portal 等价于一个打开的游标；但由于游标不能处理非 `SELECT` 语句，这里采用了不同的措辞。）

整个执行周期包括一个解析步骤，它从文本查询字符串创建预备语句；一个绑定步骤，它根据预备语句和所需参数值创建 portal；以及一个执行步骤，用于执行 portal 中的查询。对于返回行的查询（`SELECT`、`SHOW`等），可以要求执行步骤只取回有限数量的行，因此可能需要多次执行步骤才能完成整个操作。

后端可以跟踪多个预备语句和 portal（但请注意，它们只存在于单个会话内，绝不会在会话之间共享）。已有的预备语句和 portal 都通过创建时赋予的名称来引用。此外，还存在一个“未命名”的预备语句和 portal。虽然它们的行为与有名对象大体相同，但对未命名对象的操作是为“只执行一次然后丢弃”的场景优化的，而对有名对象的操作则是基于会被多次使用的预期进行优化的。

44.1.3. 格式和格式代码

某一特定数据类型的数据可以使用多种不同的格式之一进行传输。自 PostgreSQL 7.4 起，当前只支持“文本”和“二进制”两种格式，但协议为未来扩展留出了空间。任意值所需的格式由格式代码指定。客户端可以为每个传输的参数值以及查询结果的每一列指定格式代码。文本格式的代码为零，二进制格式的代码为一，其他格式代码则保留供将来定义。

值的文本表示是相应数据类型的输入/输出转换函数生成和接受的字符串。在传输形式中，值的末尾没有空字符；前端若要将收到的值作为 C 字符串处理，必须自行添加一个。（文本格式也不允许内嵌空字符。）

整数的二进制表示采用网络字节序（最高有效字节在前）。至于其他数据类型，请查阅文档或源代码了解其二进制表示形式。要注意，复杂数据类型的二进制表示可能会在不同服务器版本之间发生变化；文本格式通常是可移植性更好的选择。

44.2. 消息流

本节描述消息流和每种消息类型的语义。（每种消息的精确表示的细节见第 44.4 节.) 有几种不同的子协议，取决于连接的状态：启动、查询、函数调用、COPY 和终止。还有针对异步操作（包括通知响应和命令取消）的特殊规定，它们可以在启动阶段之后的任何时间发生。

44.2.1. 启动

要开始一个会话，前端会打开到服务器的连接并发送一条启动消息。该消息包含用户名以及用户希望连接的数据库名；它还指明要使用的协议版本。（启动消息也可以选择性地包含运行时参数的附加设置。）随后服务器会结合这些信息以及配置文件（例如 `pg_hba.conf`）的内容，初步判断是否接受该连接，以及需要何种额外认证（如果需要）。

接着服务器会发送适当的认证请求消息，
前端必须以适当的认证响应消息进行回应（例如密码）。原则上，认证请求/
响应循环可能需要多次迭代，但目前没有任何认证方法使用一次以上的请求和响应。
在某些方法中，前端根本不需要发送响应，因此也不会出现认证请求。

认证周期要么以服务器拒绝连接（ErrorResponse）结束，要么以 AuthenticationOk 结束。

该阶段服务器可能发送的消息有：

ErrorResponse

连接尝试被拒绝。随后服务器立即关闭连接。

AuthenticationOk

认证交换成功完成。

AuthenticationKerberosV4

前端现在必须参与与服务器之间的 Kerberos V4 认证对话（此处不再描述，它属于 Kerberos 规范的一部分）。如果对话成功，服务器会响应 AuthenticationOk；否则响应 ErrorResponse。

AuthenticationKerberosV5

前端现在必须参与与服务器之间的 Kerberos V5 认证对话（此处不再描述，它属于 Kerberos 规范的一部分）。如果对话成功，服务器会响应 AuthenticationOk；否则响应 ErrorResponse。

AuthenticationCleartextPassword

前端现在必须发送一个以明文形式包含密码的 PasswordMessage。如果密码正确，服务器响应 AuthenticationOk；否则响应 ErrorResponse。

AuthenticationCryptPassword

前端现在必须发送一个 PasswordMessage，其中包含用 crypt(3) 加密的密码，加密时使用 AuthenticationCryptPassword 消息中指定的 2 字符盐。如果密码正确，服务器响应 AuthenticationOk；否则响应 ErrorResponse。

AuthenticationMD5Password

前端现在必须发送一个 PasswordMessage，其中包含使用 AuthenticationMD5Password 消息中指定的 4 字符盐经过 MD5 加密的密码。如果密码正确，服务器响应 AuthenticationOk；否则响应 ErrorResponse。

AuthenticationSCMCredential

此响应仅适用于支持 SCM 凭据消息的平台上的本地 Unix 域连接。前端必须发出一条 SCM 凭据消息，然后发送一个单字节数据。（该数据字节的内容无关紧要；它仅用于确保服务器等待足够长的时间来接收凭据消息。）如果凭据可接受，服务器响应 AuthenticationOk；否则响应 ErrorResponse。

如果前端不支持服务器要求的认证方式，那么它应该马上关闭连接。

在收到 AuthenticationOk 消息之后，前端必须继续等待来自服务器的后续消息。在这个阶段，
一个后端进程正在启动，而前端只是旁观者。启动尝试仍可能失败（ErrorResponse），
但通常情况下，后端会发送一些 ParameterStatus 消息、BackendKeyData，以及最后的 ReadyForQuery。

在这个阶段，后端会尝试应用启动消息中给出的任何额外运行时参数设置。如果成功，这些值就会成为会话的默认值。发生错误则会导致 ErrorResponse 并退出。

后端在这个阶段可能发送的消息有：

BackendKeyData

该消息提供密钥数据。如果前端希望稍后发送取消请求，就必须保存这些数据。前端不应响应该消息，而应继续等待 ReadyForQuery 消息。

ParameterStatus

此消息告知前端后端参数的当前（初始）设置，例如 client_encoding 或 DateStyle。前端可以忽略此消息，或记录这些设置供以后使用；更多细节见第 44.2.6 节。前端不应响应此消息，而应继续等待 ReadyForQuery 消息。

ReadyForQuery

启动成功，前端现在可以发出命令。

ErrorResponse

启动失败，在发送完这个消息之后连接被关闭。

NoticeResponse

已发出一条警告消息。前端应显示该消息，但仍应继续等待 ReadyForQuery 或 Error Response。

ReadyForQuery 消息与后端在每个命令周期结束后发出的消息是同一个。前端可根据自身编码需要，将 ReadyForQuery 视为一个命令周期的开始，或视为启动阶段以及后续每个命令周期的结束。

44.2.2. 简单查询

一个简单查询周期由前端向后端发送一条 Query 消息来启动。该消息包含一条或多条以文本字符串表示的 SQL 命令。后端随后会根据查询命令串的内容向前端发送一条或多条响应消息，最后再发送一条 ReadyForQuery 响应消息。ReadyForQuery 通知前端它现在可以安全地发送新的命令。（实际上，前端并不一定非要等到收到 ReadyForQuery 才发送下一条命令，但这样一来，前端就必须自己处理“先前命令失败而后续已发出的命令成功”这种情况。）

后端可能返回的响应消息有：

CommandComplete

一条 SQL 命令正常完成。

CopyInResponse

后端已准备好把数据从前端复制到表中；参见第 44.2.5 节。

CopyOutResponse

后端已准备好把数据从表中复制到前端；参见第 44.2.5 节。

RowDescription

表示即将返回行作为对`SELECT`、`FETCH`等查询的响应。此消息的内容描述了行的列布局。该消息之后，每个返回给前端的行都对应一条 DataRow 消息。

DataRow

由SELECT、FETCH等查询返回的一组行中的一个。

EmptyQueryResponse

识别出了一条空查询字符串。

ErrorResponse

发生了一个错误。

ReadyForQuery

查询字符串的处理已完成。发送一个单独的消息来指示这一点，
因为查询字符串可能包含多个SQL命令。（Command
Complete标记了一个SQL命令的处理结束，而不是整个字符串的结束。）
无论处理是成功还是出现错误，都将始终发送ReadyForQuery。

NoticeResponse

与查询相关的警告消息已发出。通知是其他响应的补充，即后端将继续处理命令。

SELECT 查询（或其他返回行集的查询，如 EXPLAIN 或 SHOW）的响应通常包含 Row Description、零条或多条 DataRow 消息，以及最后的 CommandComplete。在前端与服务器之间执行 COPY 输入或输出时，会使用第 44.2.5 节所述的特殊协议。所有其他类型的查询通常只产生一条 CommandComplete 消息。

由于查询字符串可能包含若干条查询（以分号分隔），
因此在后端完成整个查询字符串的处理之前，可能会出现多个这样的响应序列。
只有在整个字符串处理完毕且后端已准备好接受新的查询字符串时，才会发出 ReadyForQuery 消息。

如果收到的是一条完全为空的查询字符串（除了空白字符之外没有任何内容），响应就是一条 EmptyQueryResponse，后面跟着 ReadyForQuery。

一旦发生错误，就会发出一条 ErrorResponse 消息，后面跟着 ReadyForQuery。Error Response 会中止该查询字符串中后续所有处理（即使其中还包含其他查询）。请注意，这种情况可能发生在处理单条查询所生成的消息序列中途。

在简单查询模式中，检索到的值的格式总是文本，除非给定的命令是对用 BINARY 选项声明的游标的 FETCH。在这种情况下，检索到的值是二进制格式。RowDescription 消息中给出的格式代码指明使用的是哪种格式。

当前端正在等待其他类型的消息时，也必须准备好接收 ErrorResponse 和 NoticeResponse 消息。参见第 44.2.6 节，了解后端因外部事件而可能生成的消息。

建议以状态机的方式编写前端，使其能够在任何合理的时机接收相应类型的消息，
而不把消息确切顺序的假设写死在代码中。

44.2.3. 扩展查询

扩展查询协议把上文描述的简单查询协议拆分成多个步骤。准备步骤的结果可以重复使用，
从而提高效率。此外，它还提供了额外特性，例如可以把数据值作为独立参数提供，
而不必直接插入查询字符串中。

在扩展协议中，前端首先发送一条 Parse 消息，其中包含文本查询字符串、可选的参数占位符数据类型信息，以及目标预备语句对象的名称（空字符串表示未命名预备语句）。响应要么是 ParseComplete，要么是 ErrorResponse。参数数据类型可以用 OID 指定；如果未给出，解析器会像处理无类型字面字符串常量那样尝试推断其数据类型。

> **注意**
>
> Parse 消息中的查询字符串不能包含多于一条 SQL 语句；否则会报告语法错误。这个限制在简单查询协议中并不存在，但在扩展协议中必须如此，因为若允许预备语句或 portal 包含多条命令，会使协议变得过于复杂。

If successfully created, a named prepared-statement object lasts till 当前会话结束，除非被显式销毁。未命名预备语句只会持续到下一条把未命名语句作为目标的 Parse 语句发出为止。（注意，简单 Query 消息也会销毁未命名语句。）有名预备语句在被另一条 Parse 消息重新定义之前必须显式关闭，但未命名语句则无此要求。有名预备语句也可以在 SQL 命令层使用 `PREPARE` 和 `EXECUTE`。

一旦预备语句存在，就可以用 Bind 消息把它准备为可执行状态。Bind 消息给出源预备语句的名称（空字符串表示未命名预备语句）、目标 portal 的名称（空字符串表示未命名 portal），以及预备语句中所有参数占位符应使用的值。所提供的参数集必须与预备语句所需参数相匹配。Bind 还会指定查询返回数据所使用的格式；格式既可以统一指定，也可以按列指定。响应要么是 BindComplete，要么是 ErrorResponse。

> **注意**
>
> 输出采用文本还是二进制格式，由 Bind 中给出的格式代码决定，而与涉及的 SQL 命令无关。在使用扩展查询协议时，游标声明中的 `BINARY` 属性并不起作用。

如果成功创建，命名 portal 对象会一直持续到当前事务结束，除非被显式销毁。未命名 portal 会在事务结束时销毁，或者在下一条把未命名 portal 作为目标的 Bind 语句发出时立即销毁。（注意，简单 Query 消息也会销毁未命名 portal。）有名 portal 在被另一条 Bind 消息重新定义之前必须显式关闭，而未命名 portal 则无此要求。有名 portal 也可以在 SQL 命令层使用 `DECLARE CURSOR` 和 `FETCH`。

一旦门户存在，就可以用 Execute 消息执行它。Execute 消息指定门户名（空字符串表示未命名门户）和最大结果行数（零表示 “fetch all rows”).）。结果行数只对包含返回行集的命令的门户有意义；在其他情况下命令总是执行到完成，行数被忽略。Execute 可能的响应与上面针对通过简单查询协议发出的查询所描述的相同，只是 Execute 不会导致发出 ReadyForQuery。

如果 Execute 在 portal 执行完成之前终止（因为达到了非零的结果行计数），它会发送一条 PortalSuspended 消息；该消息表明前端应当针对同一个 portal 再发出一条 Execute 消息，以完成此次操作。在 portal 执行完成之前，不会发送表示源 SQL 命令结束的 Command Complete 消息。因此，一个 Execute 阶段总会以下列消息中的恰好一条结束：Command Complete、EmptyQueryResponse（如果 portal 是从空查询字符串创建的）、ErrorResponse 或 PortalSuspended。

在每个扩展查询消息序列结束时，前端应发出一条 Sync 消息。这个无参数消息使后端在不在 BEGIN/COMMIT 事务块内时关闭当前事务（“close” 意思是若无错误则提交，有错误则回滚）。然后发出 ReadyForQuery 响应。Sync 的目的是为错误恢复提供一个重新同步点。当在处理任何扩展查询消息时检测到错误，后端发出 ErrorResponse，然后读取并丢弃消息直到到达 Sync，然

后发出 ReadyForQuery 并恢复正常消息处理。（但注意，如果在处理 Sync 期间检测到错误则不会发生跳过——这确保每个 Sync 发送且只发送一个 ReadyForQuery。）

> **注意**
>
> Sync 不会关闭由`BEGIN`打开的事务块。之所以能够检测出这种情况，是因为 ReadyForQuery 消息中包含事务状态信息。

除了这些基本的、必需的操作之外，在扩展查询协议里还有几种可选的操作可以使用。

Describe 消息（portal 变体）指定一个现有 portal 的名称（或者用空字符串表示未命名 portal）。响应要么是一条 RowDescription 消息，用于描述执行该 portal 时将返回的行；要么是一条 NoData 消息，如果该 portal 不包含会返回行的查询；要么是 ErrorResponse，如果不存在这样的 portal。

Describe 消息（语句变体）指定一个现有预备语句的名称（或者用空字符串表示未命名预备语句）。响应是一条 ParameterDescription 消息，用于描述该语句所需的参数，随后是一条 RowDescription 消息，用于描述该语句最终执行时将返回的行（如果该语句不返回行，则为 NoData 消息）。如果不存在这样的预备语句，则返回 ErrorResponse。请注意，由于还没有发出 Bind，后端尚不知道返回列将使用什么格式；因此在这种情况下，RowDescription 消息中的格式代码字段将为零。

> **提示**
>
> 在大多数场景下，前端都应在发出 Execute 之前先发送某一种 Describe 变体，以确保它知道应如何解释接收到的结果。

Close 消息会关闭一个现有的预备语句或 portal，并释放相关资源。针对不存在的语句或 portal 发出 Close 并不算错误。响应通常是 CloseComplete，但如果在释放资源时遇到困难，也可能返回 ErrorResponse。请注意，关闭预备语句会隐式关闭所有基于该语句构造出的打开 portal。

Flush 消息本身不会产生任何特定输出，但会强制后端发送其输出缓冲区中所有尚待发送的数据。如果前端希望在发出更多命令之前检查某条扩展查询命令的结果，那么除了 Sync 之外，任何扩展查询命令之后都必须发送 Flush。如果没有 Flush，后端返回的消息会尽量合并成最少的数据包，以降低网络开销。

> **注意**
>
> 简单 Query 消息大致等价于一串 Parse、Bind、portal Describe、Execute、Close、Sync 操作，它们使用未命名预备语句和未命名 portal 对象，并且不带参数。不同之处在于，简单 Query 会接受包含多条 SQL 语句的查询字符串，并依次自动为每条语句执行绑定、描述和执行序列；另一个区别是它不会返回 ParseComplete、BindComplete、CloseComplete 或 NoData 消息。

44.2.4. 函数调用

函数调用子协议允许客户端请求直接调用数据库`pg_proc`系统目录中的任意函数。
客户端必须具有该函数的执行权限。

> **注意**
>
> 函数调用子协议是一个遗留的特性，在新代码里可能最好避免用它。
> 类似的结果可以通过设置一个执行`SELECT function($1, ...)`的预备语句得到。这样函数调用周期就可以用 Bind/Execute 代替。

函数调用周期由前端向后端发送一条 FunctionCall 消息来启动。后端随后根据函数调用的结果发送一条或多条响应消息，最后发送一条 ReadyForQuery 响应消息。ReadyForQuery 告知前端，可以安全地发送新的查询或函数调用。

来自后端的可能的响应消息是：

ErrorResponse

发生了一个错误。

FunctionCallResponse

函数调用完成并且在消息中返回一个结果（请注意函数调用协议只能处理单个标量结果，不能处理行类型或者结果集合）。

ReadyForQuery

函数调用处理完成。ReadyForQuery将总是被发送，不管是成功完成处理还是发生一个错误。

NoticeResponse

发出了一条有关该函数调用的警告信息。通知是附加在其他响应上的，也就是说，后端将继续处理该命令。

44.2.5. COPY 操作

`COPY`命令允许在服务器和客户端之间进行高速大批量数据传输。
拷贝入和拷贝出操作每个都把连接切换到一个独立的子协议中，并且持续到操作结束。

拷贝入模式（向服务器传输数据）在后端执行`COPY FROM STDIN` SQL 语句时启动。后端会向前端发送一条 CopyInResponse 消息。随后前端应发送零条或多条 CopyData 消息，构成一条输入数据流。（消息边界与行边界之间没有任何对应关系要求，尽管让它们对齐通常是合理的选择。）前端可以通过发送 CopyDone 消息来结束拷贝入模式（允许成功结束），也可以发送 CopyFail 消息（这会使`COPY`语句以错误失败）。然后后端会恢复到`COPY`开始之前的命令处理模式，也就是简单查询协议或扩展查询协议。接下来它会发送 CommandComplete（成功时）或 ErrorResponse（失败时）。

如果在拷贝入模式期间后端检测到错误（包括收到 CopyFail 消息），后端会发出一条 ErrorResponse 消息。如果`COPY`命令是通过扩展查询消息发出的，那么后端会从此开始丢弃前端消息，直到收到一条 Sync 消息，然后发出 ReadyForQuery 并恢复正常处理。如果`COPY`命令是在简单 Query 消息中发出的，那么该消息的剩余部分会被丢弃，并发送 ReadyForQuery。无论哪种情况，前端随后发出的任何 CopyData、CopyDone 或 CopyFail 消息都会被直接丢弃。

后端会忽略在拷贝入模式期间收到的 Flush 和 Sync 消息。收到任何其他非拷贝类型的消息都会构成错误，并按上述方式中止拷贝入状态。（Flush 和 Sync 的例外是为了方便那些总是在 Execute 消息之后发送 Flush 或 Sync、而不检查待执行命令是否为`COPY FROM STDIN`的客户端库。）

拷贝出模式（数据从服务器发出）是在后端执行一个`COPY TO STDOUT`语句时启动的。后端发出一个CopyOutResponse消息给前端，后面跟着零或者多个CopyData消息（总是每行一个），然后跟着CopyDone。然后后端回退到它在`COPY`开始之前的命令处理模式，然后发送Command Complete。前端不能中止传输（除非是关闭连接或者发出一个Cancel请求），但是它可以抛弃不需要的CopyData和CopyDone消息。

在拷贝出模式中，如果后端检测到错误，那么它将发出一个Error Response消息并且回到正常的处理。前端应该把收到Error Response当作终止拷贝出模式的标志。

NoticeResponse 和 ParameterStatus 消息可能穿插在 CopyData 消息之间；前端必须处理这些情况，并应准备好处理其他异步消息类型（参见第 44.2.6 节）。除此之外，可以将任何除 Copy Data 或 CopyDone 以外的消息类型视为拷贝出模式的终止标志。

CopyInResponse 和 CopyOutResponse 消息包含一些字段，告知前端每行的列数以及每列所使用的格式代码。（就当前实现而言，给定 `COPY` 操作中的所有列都使用相同的格式，但消息的设计并不假定这一点。）

44.2.6. 异步操作

在若干情况下，后端会发送并非由前端命令流直接触发的消息。前端必须随时准备处理这些消息，即使当前并未处于查询过程中。至少，在开始读取查询响应之前应检查这些情况。

NoticeResponse 消息可能因外部活动而产生；例如，如果数据库管理员发起一次“快速”数据库关闭，后端会在关闭连接之前发送一条 NoticeResponse 说明这一事实。因此，前端应始终准备好接收并显示 NoticeResponse 消息，即使连接表面上处于空闲状态。

每当后端认为前端应当知道的任何参数的活动值发生变化时，就会生成 ParameterStatus 消息。最常见的情况是响应前端执行的 SET SQL 命令，这种情况实际上是同步的——但也可能因为管理员更改了配置文件然后向 postmaster 发送 SIGHUP 信号而发生参数状态变化。此外，如果 SET 命令被回滚，也会生成适当的 ParameterStatus 消息来报告当前的有效值。

目前有一组硬编码的参数会为其生成 ParameterStatus：它们是 server_version（一个启动后不能更改的伪参数）、client_encoding、is_superuser、session_authorization 和 DateStyle。这组参数将来可能改变，甚至可能变成可配置的。因此，前端对它不理解或不关心的参数只需忽略 ParameterStatus 即可。

如果前端发出`LISTEN`命令，那么每当针对同一通知名执行`NOTIFY`命令时，后端都会发送一条 NotificationResponse 消息（不要与 NoticeResponse 混淆）。

> **注意**
>
> 目前，NotificationResponse只能在一个事务外面发送，因此它将不会在一个命令响应序列中间出现，但是它可能正好在ReadyForQuery之前出现。不过，在前端逻辑中做上述假设是不明智的。好的做法是在协议的任何点上都可以接受NotificationResponse。

44.2.7. 取消进行中的请求

在一条查询正在处理的时候，前端可以请求取消该查询。这种取消请求不是直接通过打开的连接发送给后端的，这么做是因为实现的效率：我们不希望后端在处理查询的过程中不停地检查前端来的输入。

取消请求应该相对而言比较少见，所以我们把取消做得稍微笨拙一些，
以便不影响正常状况的性能。

要发出取消请求，前端会新建到服务器的连接，并发送 CancelRequest 消息，而不是新连接通常发送的 StartupMessage 消息。服务器处理该请求后便会关闭连接。出于安全原因，服务器不会直接回复取消请求消息。

除非CancelRequest消息包含在连接启动过程中传递给前端的相同的密钥数据（PID 和密钥），
否则它将被忽略。如果该请求匹配当前运行着的后端的PID和密钥，
则中止当前查询的处理（目前的实现里采用的方法是向正在处理该查询的后端进程发送一个特殊的信号）。

取消信号可能有效也可能无效——例如，如果它到达时后端已经处理完查询，
那么它就不会有效果。如果取消有效，则会导致当前命令提前终止并附带一条错误消息。

这么做是对安全性和效率通盘考虑的结果，前端没有直接的方法获知一个取消请求是否成功。
它必须继续等待后端对查询的响应。发出一个取消仅仅是增加了当前查询快些结束的可能性，
同时也增加了当前查询会伴随着一条错误消息失败而不是成功执行的可能性。

由于取消请求是通过一条新的连接发送给服务器，而不是通过常规的前端/后端通信链路发送，
因此发出取消请求的可以是任意进程，而不一定非要是要取消查询的那个前端。
这为构建多进程应用提供了额外的灵活性，同时也带来了安全风险，
因为未授权用户可能会尝试取消查询。通过要求在取消请求中提供动态生成的密钥，
可以缓解这一安全风险。

44.2.8. 终止

通常优雅的终止过程是前端发送一条Terminate消息并且立刻关闭连接。一旦收到消息，
后端马上关闭连接并且终止。

在少数情况下（比如一个管理员命令数据库关闭），
后端可能在没有任何前端请求的情况下断开连接。在这种情况下，
后端将在它断开连接之前尝试发送一个错误或者通知消息给出断开的原因。

其他终止场景来自各种故障，例如任一端发生 core dump、通信链路中断、消息边界同步丢失等。如果前端或后端看到连接意外关闭，就应清理并终止。若前端不想自行终止，
也可以重新联系服务器以启动一个新的后端。如果收到无法识别的消息类型，
同样建议关闭连接，因为这通常意味着消息边界同步已经丢失。

无论是正常还是异常终止，任何打开的事务都会被回滚而不是提交。但应注意，
如果前端在一个非 SELECT 查询正在处理时断开连接，后端很可能在注意到断开之前完成该查询。如果该查询在任何事务块（BEGIN ... COMMIT 序列）之外，那么其结果可能在断开被识别之前提交。

44.2.9. SSL 会话加密

如果 PostgreSQL 构建时带有 SSL 支持，前端/后端通信就可以用 SSL 加密。这在攻击者可能捕获会话流量的环境中提供了通信安全性。

要发起 SSL 加密连接，前端最初发送 SSLRequest 消息而不是 StartupMessage。然后服务器以一个包含 S 或 N 的单字节响应，分别表示它愿意或不愿意执行 SSL。如果前端对响应不满意，可以在此点关闭连接。要在 S 之后继续，与服务器执行 SSL 启动握手（此处不描述，属于 SSL 规范的一部分）。如果成功，继续发送通常的 StartupMessage。在这种情况下 StartupMessage 和所有后续数据都将被 SSL 加密。要在 N 之后继续，发送通常的 StartupMessage 并不加密地继续。

前端还应准备好处理服务器对 SSLRequest 的 ErrorMessage 响应。这只会在服务器早于向 PostgreSQL 添加 SSL 支持的版本时发生。在这种情况下必须关闭连接，但前端可以选择打开一个新连接并且不请求 SSL 地继续。

如果建立连接是为了发送 CancelRequest 消息，也可以先发送 SSLRequest。

虽然协议本身没有提供让服务器强制使用 SSL 加密的方法，但管理员可以配置服务器，使其在认证检查中拒绝未加密的会话。

44.3. 消息数据类型

本节描述消息中使用的基础数据类型。

Int*n*(*i*)

以网络字节序（最高有效字节在前）表示的 *n* 位整数。如果指定了 *i*，则表示其精确值；否则表示该值可变。例如：Int16、Int32(42)。

Int*n*[*k*]

由 *k* 个 *n* 位整数组成的数组，每个整数都按网络字节序排列。数组长度 *k* 总是由消息中更早的某个字段确定。例如：Int16[M]。

String(*s*)

一个以空字符结尾的字符串（C 风格字符串）。字符串没有特定的长度限制。如果指定了 *s*，则表示其精确值；否则表示该值可变。例如：String、String("user")。

> **注意**
>
> 后端返回的字符串长度没有预定义的上限。前端的好做法是使用可扩展的缓冲区，这样凡是能装进内存的内容都能接受。如果做不到这一点，就读取整个字符串，然后丢弃固定大小缓冲区装不下的尾部字符。

Byte*n*(*c*)

精确 *n* 字节。如果字段宽度 *n* 不是常数，它总是可以由消息中更早的字段确定。如果指定了 *c*，则表示其精确值。例如：Byte2、Byte1('\n')。

44.4. 消息格式

本节描述每条消息的详细格式。每条消息都标记了可由前端（F）、后端（B）或双方（F&B）发送。请注意，虽然每条消息开头都带有字节计数，
但大多数消息格式都定义为无需参考该计数也能确定消息边界。
这一设计最初是出于历史原因（早期已废弃的协议 v2 没有显式长度字段），同时也有助于有效性校验。

AuthenticationOk (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32(0)

表示认证成功。

AuthenticationKerberosV4 (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32(1)

表示需要 Kerberos V4 认证。

AuthenticationKerberosV5 (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32(2)

表示需要 Kerberos V5 认证。

AuthenticationCleartextPassword (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32(3)

表示需要明文密码。

AuthenticationCryptPassword (B)

Byte1('R')

将该消息标识为认证请求。

Int32(10)

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32(4)

表示需要经过 crypt() 加密的密码。

Byte2

加密密码所用的盐。

AuthenticationMD5Password (B)

Byte1('R')

将该消息标识为认证请求。

Int32(12)

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32(5)

表示需要经过 MD5 加密的密码。

Byte4

加密密码时使用的盐。

AuthenticationSCMCredential (B)

Byte1('R')

将该消息标识为认证请求。

Int32(8)

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32(6)

表示需要 SCM 凭证消息。

BackendKeyData (B)

Byte1('K')

将此消息标识为取消请求密钥数据。如果前端希望以后能够发送 CancelRequest 消息，就必须保存这些值。

Int32(12)

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32

此后端的进程 ID。

Int32

此后端的密钥。

Bind (F)

Byte1('B')

将该消息标识为 Bind 命令。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

String

目标 portal 的名称（空字符串选择未命名的 portal）。

String

源预备语句的名称（空字符串选择未命名的预备语句）。

Int16

后续参数格式码的数量（下文以 *C* 表示）。可以为零，表示没有参数，或者所有参数都使用默认格式（文本）；也可以为一，此时指定的格式码应用于所有参数；
还可以等于实际参数数量。

Int16[*C*]

参数格式码。目前每个格式码必须为零（文本）或一（二进制）。

Int16

后续参数值的数量（可以为零）。必须与查询所需的参数数量一致。

接下来，每个参数都有以下一对字段：

Int32

参数值的长度，以字节为单位（不包括此长度字段本身）。可以为零。特殊值 -1 表示 NULL 参数值。为 NULL 时，后面不再有值的字节。

Byte*n*

参数值，格式由对应的格式码指明。*n* 为上述长度。

最后一个参数之后是以下字段：

Int16

后续结果列格式码的数量（下文以 *R* 表示）。可以为零，表示没有结果列，或者所有结果列都应使用默认格式（文本）；也可以为一，
此时指定的格式码应用于所有结果列（如果有）；还可以等于查询实际的结果列数量。

Int16[*R*]

结果列格式码。目前每个格式码必须为零（文本）或一（二进制）。

BindComplete (B)

Byte1('2')

将该消息标识为 Bind 完成指示。

Int32(4)

消息内容的长度，以字节为单位，包括此长度字段本身。

CancelRequest (F)

Int32(16)

消息内容以字节计的长度，包括自身。

Int32(80877102)

取消请求代码。此值的最高 16 位为 `1234`，最低 16 位为 `5678`。（为避免混淆，此代码不能与任何协议版本号相同。）

Int32

目标后端的进程 ID。

Int32

目标后端的密钥。

Close (F)

Byte1('C')

将该消息标识为 Close 命令。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

Byte1

“`S`”表示关闭预备语句；“`P`”表示关闭 portal。

String

要关闭的预备语句或 portal 的名称（空字符串选择未命名的预备语句或 portal）。

CloseComplete (B)

Byte1('3')

将该消息标识为 Close 完成指示。

Int32(4)

消息内容的长度，以字节为单位，包括此长度字段本身。

CommandComplete (B)

Byte1('C')

将该消息标识为命令完成响应。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

String

命令标签。通常是一个单词，用来标识已完成的 SQL 命令。

对于 `INSERT` 命令，标签是 `INSERT` *`oid rows`*，其中 *`rows`* 是插入的行数。如果 *`rows`* 为 1 且目标表具有 OID，则 *`oid`* 是插入行的对象 ID；否则 *`oid`* 为 0。

对于`DELETE`命令，标签是`DELETE` *`rows`*，其中*`rows`*表示删除的行数。

对于`UPDATE`命令，标签是`UPDATE` *`rows`*，其中*`rows`*是更新的行数。

对于`MOVE`命令，标签是`MOVE` *`rows`*，其中*`rows`*表示游标位置改变的行数。

对于`FETCH`命令，标签是`FETCH` *`rows`*，其中*`rows`*是从游标中检索出的行数。

CopyData (F & B)

Byte1('d')

标识此消息为 COPY 数据。

Int32

消息内容的长度（字节），包括自身。

Byte*n*

构成 COPY 数据流一部分的数据。从后端发送的消息总是对应单个数据行，但前端发送的消息可以任意划分数据流。

CopyDone (F & B)

Byte1('c')

标识此消息为 COPY 完成指示符。

Int32(4)

消息内容的长度（字节），包括自身。

CopyFail (F)

Byte1('f')

标识此消息为 COPY 失败指示符。

Int32

消息内容的长度（字节），包括自身。

String

作为失败原因报告的错误消息。

CopyInResponse (B)

Byte1('G')

标识此消息为 Start Copy In 响应。前端现在必须发送 copy-in 数据（如果没有准备好这样做，发送 CopyFail 消息）。

Int32

消息内容的长度（字节），包括自身。

Int8

0 表示整体复制格式为文本（行以换行符分隔、列以分隔字符分隔等）。1 表示整体复制格式为二进制（类似于 DataRow 格式）。更多信息见COPY。

Int16

要复制的数据中的列数（下面记作 N）。

Int16[*N*]

用于每列的格式代码。目前每个必须为零（文本）或一（二进制）。
如果整体复制格式为文本，则全部必须为零。

CopyOutResponse (B)

Byte1('H')

标识此消息为 Start Copy Out 响应。此消息后面将跟随 copy-out 数据。

Int32

消息内容的长度（字节），包括自身。

Int8

0 表示整体复制格式为文本（行以换行符分隔、列以分隔字符分隔等）。1 表示整体复制格式为二进制（类似于 DataRow 格式）。更多信息见COPY。

Int16

要复制的数据中的列数（下面记作 N）。

Int16[*N*]

用于每列的格式代码。目前每个必须为零（文本）或一（二进制）。
如果整体复制格式为文本，则全部必须为零。

DataRow (B)

Byte1('D')

标识消息为数据行。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

Int16

后面跟着的列值的数量（可能为零）。

接下来，每列都有以下两个字段：

Int32

列值的长度，以字节为单位（不包括本长度字段自身）。可以为零。特殊值 -1 表示列值为 NULL，此时后面没有值字节。

Byte*n*

列的值，格式由相关的格式代码指示。*n*是上述长度。

Describe (F)

Byte1('D')

标识消息为描述命令。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

Byte1

'S' 表示描述一个预备语句；或者 'P' 表示描述一个 portal。

String

要描述的预备语句或 portal 的名称（空字符串选择未命名的预备语句或 portal）。

EmptyQueryResponse (B)

Byte1('I')

标识消息为对空查询字符串的响应。（此消息替代 CommandComplete。）

Int32(4)

消息内容的长度，以字节为单位，包括此长度字段本身。

ErrorResponse (B)

Byte1('E')

将消息标识为错误。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

消息体由一个或多个带标识的字段组成，最后以一个零字节终止。
字段可以按任意顺序出现。每个字段包含以下内容：

Byte1

一个用于标识字段类型的代码；如果为零，则这是消息终止符，后面没有字符串。
目前定义的字段类型列在第 44.5 节中。由于将来可能会添加更多的字段类型，前端应该静默地忽略未识别类型的字段。

String

字段值。

Execute (F)

Byte1('E')

标识消息为一个执行命令。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

String

要执行的 portal 的名称（空字符串选择未命名的 portal）。

Int32

如果 portal 包含返回行的查询，则这是最多返回的行数（否则忽略此值）。零表示“无限制”。

Flush (F)

Byte1('H')

将消息标识为Flush命令。

Int32(4)

消息内容的长度，以字节为单位，包括此长度字段本身。

FunctionCall (F)

Byte1('F')

标识消息为函数调用。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32

指定要调用的函数的对象 ID。

Int16

后续参数格式代码的数量（以下用 C 表示）。可以为零，表示没有参数，或所有参数都采用默认格式（文本）；也可以为一，表示将指定的格式代码用于所有参数；
还可以等于实际参数数量。

Int16[C]

参数格式代码。每个目前必须是零（文本）或一（二进制）。

Int16

指定传递给函数的参数数量。

接下来，每个参数都有以下两个字段：

Int32

参数值的长度，以字节为单位（不包括本长度字段自身）。可以为零。特殊值 -1 表示参数值为 NULL，此时后面没有值字节。

Byte*n*

参数的值，以相关格式代码指示的格式表示。*n*是上述长度。

最后一个参数之后还有以下字段：

Int16

函数结果的格式代码。目前必须为零（文本）或一（二进制）。

FunctionCallResponse (B)

Byte1('V')

标识消息为函数调用结果。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32

函数结果值的长度，以字节为单位（不包括本长度字段自身）。可以为零。特殊值 -1 表示函数结果为 NULL，此时后面没有值字节。

Byte*n*

函数结果的值，格式由相关的格式代码指示。*n*是上述长度。

NoData (B)

Byte1('n')

将消息标识为无数据指示器。

Int32(4)

消息内容的长度，以字节为单位，包括此长度字段本身。

NoticeResponse (B)

Byte1('N')

将消息标识为通知。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

消息体由一个或多个带标识的字段组成，最后以一个零字节终止。
字段可以按任意顺序出现。每个字段包含以下内容：

Byte1

一个用于标识字段类型的代码；如果为零，则这是消息终止符，后面没有字符串。目前定义的字段类型列在第 44.5 节中。由于将来可能会添加更多的字段类型，前端应该静默地忽略未识别类型的字段。

String

字段值。

NotificationResponse (B)

Byte1('A')

标识消息为通知响应。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

Int32

发出通知的后端进程的进程 ID。

String

发出该通知的通知名。

String

通知进程传来的附加信息。

ParameterDescription (B)

Byte1('t')

标识消息为参数描述。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

Int16

语句使用的参数数量（可以为零）。

接下来，每个参数都有以下内容：

Int32

指定参数数据类型的对象 ID。

ParameterStatus (B)

Byte1('S')

标识消息为运行时参数状态报告。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

String

所报告的运行时参数的名称。

String

参数的当前值。

Parse (F)

Byte1('P')

将消息标识为解析命令。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

String

目标预备语句的名称（空字符串选择未命名的预备语句）。

String

要解析的查询字符串。

Int16

指定的参数数据类型的数量（可以为零）。请注意，
这不是查询字符串中可能出现的参数数量的指示，
而是前端希望为其预先指定类型的参数数量。

接下来，每个参数都有以下内容：

Int32

指定参数数据类型的对象 ID。此处填零等同于不指定类型。

ParseComplete (B)

Byte1('1')

将消息标识为解析完成指示器。

Int32(4)

消息内容的长度，以字节为单位，包括此长度字段本身。

PasswordMessage (F)

Byte1('p')

标识消息为密码响应。请注意，这也用于GSSAPI、SSPI和SASL响应消息。
可以从上下文中推断出确切的消息类型。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

String

密码（如果需要，已加密）。

PortalSuspended (B)

Byte1('s')

标识消息为 portal 挂起指示器。请注意，仅当执行消息的行数限制达到时才会出现此消息。

Int32(4)

消息内容的长度，以字节为单位，包括此长度字段本身。

Query (F)

Byte1('Q')

标识消息为简单查询。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

String

查询字符串本身。

ReadyForQuery (B)

Byte1('Z')

标识消息类型。ReadyForQuery在后端准备好进行新的查询周期时发送。

Int32(5)

消息内容的长度，以字节为单位，包括此长度字段本身。

Byte1

当前后端事务状态指示器。可能的值为'I'，如果空闲（不在事务块中）；'T'，如果在事务块中；或'E'，如果在失败的事务块中（查询将被拒绝，直到块结束）。

RowDescription (B)

Byte1('T')

标识消息为行描述。

Int32

消息内容的长度，以字节为单位，包括此长度字段本身。

Int16

指定一行中的字段数量（可以为零）。

接下来，每个字段都有以下内容：

String

字段名称。

Int32

如果能够确定该字段是某个特定表的列，则为该表的对象 ID；否则为零。

Int16

如果能够确定该字段是某个特定表的列，则为该列的属性编号；否则为零。

Int32

字段数据类型的对象 ID。

Int16

数据类型大小（参见`pg_type.typlen`）。注意，负值表示可变宽度类型。

Int32

类型修饰符（参见`pg_attribute.atttypmod`）。修饰符的含义是特定于类型的。

Int16

字段所使用的格式代码。目前为零（文本）或一（二进制）。对于 Describe 的语句变体所返回的 RowDescription，格式代码尚未确定，始终为零。

SSLRequest (F)

Int32(8)

消息内容的长度（字节），包括自身。

Int32(80877103)

SSL 请求码。该值经过选择，使其高 16 位包含 1234，低 16 位包含 5679。（为避免混淆，此代码不得与任何协议版本号相同。）

StartupMessage (F)

Int32

消息内容以字节计的长度，包括自身。

Int32(196608)

协议版本号。最高 16 位是协议主版本号（本文档描述的协议为 3）。最低 16 位是协议次版本号（本文档描述的协议为 0）。

协议版本号后面跟着一对或多对参数名和参数值字符串。
最后一对名称/值对之后必须有一个零字节作为终止符。参数可以按任意顺序出现。`user` 是必需的，其他参数可选。每个参数指定为：

String

参数名。目前可识别的名称有：

`user`

要连接的数据库用户名。必需；没有默认值。

`database`

要连接的数据库。默认为用户名。

`options`

后端的命令行参数。（已弃用，建议设置单独的运行时参数。）

除上述参数外，还可以列出任何可在后端启动时设置的运行时参数。
这些设置将在后端启动期间应用（如果有的话，在解析命令行选项之后）。
这些值将作为会话默认值。

String

参数的值。

Sync (F)

Byte1('S')

将消息标识为同步命令。

Int32(4)

消息内容的长度，以字节为单位，包括此长度字段本身。

Terminate (F)

Byte1('X')

标识消息为终止。

Int32(4)

消息内容的长度，以字节为单位，包括此长度字段本身。

44.5. 错误和通知消息字段

本节描述了ErrorResponse和NoticeResponse消息中可能出现的字段。
每个字段类型都有一个单字节的标识符。请注意，
任何给定的字段类型在消息中最多只能出现一次。

`S`

严重性：字段内容为 `ERROR`、`FATAL` 或 `PANIC`（在错误消息中），或 `WARNING`、`NOTICE`、`DEBUG`、`INFO` 或 `LOG`（在通知消息中），或这些内容的本地化翻译。始终存在。

`C`

代码：错误的 SQLSTATE 代码（见附录 A）。不可本地化。始终存在。

`M`

消息：主要的可读错误消息。应准确而简洁（通常为一行）。始终存在。

`D`

详细信息：可选的次要错误消息，提供有关该问题的更多细节。可能占多行。

`H`

提示：可选的建议，说明如何处理该问题。它与详细信息的区别在于，
它提供建议（可能并不合适），而非确凿事实。可能占多行。

`P`

位置：字段值是一个十进制ASCII整数，表示错误光标位置，作为原始查询字符串的索引。
第一个字符的索引为1，位置以字符而非字节计量。

`W`

Where：错误发生的上下文的指示。目前这包括活动 PL 函数的调用栈回溯。追踪每行一个条目，最近的在前。

`F`

文件：报告错误的源代码位置所在的文件名。

`L`

行号：报告错误的源代码位置所在的行号。

`R`

例程：报告错误的源代码例程的名称。

客户端负责按自身需要格式化显示的信息；尤其应根据需要将长行断开。
错误消息字段中的换行符应视为段落分隔符，而非行分隔符。

44.6. 自协议 2.0 以来的更改汇总

本节提供一份简要的变更清单，供准备将现有客户端库更新到协议 3.0 的开发者参考。

初始启动包采用灵活的字符串列表格式，取代了固定格式。注意，
运行时参数的会话默认值现在可以直接在启动包中指定。（实际上，以前也能通过 `options` 字段实现，但由于 `options` 的宽度有限，且无法用引号保护值中的空白字符，这种方法并不稳妥。）

现在，所有消息都在消息类型字节之后紧跟一个长度计数（启动包除外，它没有类型字节）。
另请注意，PasswordMessage 现在也有类型字节。

ErrorResponse 和 NoticeResponse（'`E`' 和 '`N`'）消息现在包含多个字段，客户端代码可以利用这些字段组合出所需详细程度的错误消息。注意，各字段通常不会以换行符结束，
而旧协议发送的单个字符串总是以换行符结束。

ReadyForQuery（'`Z`'）消息包含一个事务状态指示器。

BinaryRow 与 DataRow 消息类型不再有区别；单一的 DataRow 消息类型用于返回所有格式的数据。注意，DataRow 的布局已经改变，使其更容易解析。另外，二进制值的表示方式也已改变，不再直接取决于服务器内部的表示方式。

新增了“扩展查询”子协议，其中增加了前端消息类型 Parse、Bind、Execute、Describe、Close、Flush 和 Sync，以及后端消息类型 ParseComplete、BindComplete、Portal

Suspended、ParameterDescription、NoData 和 CloseComplete。现有客户端不必关注这个子协议，但使用它可能有助于改善性能或功能。

COPY 数据现在被封装到 CopyData 和 CopyDone 消息中。有明确的方式从 COPY 期间的错误中恢复。不再需要特殊的“`\.`”最后一行，COPY OUT 期间也不发送它。（COPY IN 期间它仍被识别为终止符，但其使用已废弃并最终将被移除。）支持二进制 COPY。CopyInResponse 和 CopyOutResponse 消息包含指示列数和每列格式的字段。

FunctionCall 和 FunctionCallResponse 消息的布局已经改变。FunctionCall 现在支持向函数传递 NULL 参数，也可以使用文本或二进制格式传递参数和取得结果。由于它不再提供对服务器内部数据表示的直接访问，因此也不再有理由将 FunctionCall 视为潜在的安全漏洞。

在连接启动期间，后端会为其认为客户端库关注的所有参数发送 ParameterStatus（`'S'`）消息。之后，只要这些参数中任何一个的当前值发生变化，就会发送一条 ParameterStatus 消息。

RowDescription（`'T'`）消息针对所描述行中的每一列，增加了表 OID 和列编号字段，并显示各列的格式代码。

后端不再生成 CursorResponse（`'P'`）消息。

NotificationResponse（A）消息有一个额外的字符串字段，它目前为空，
但将来某天可能携带从 NOTIFY 事件发送者传递来的附加数据。

EmptyQueryResponse（`'I'`）消息以前包含一个空字符串参数，现在已将其移除。

第 45 章 PostgreSQL 编码约定

45.1. 格式

源代码格式使用 4 列制表宽度，并保留制表符（即，不会把制表符展开为空格）。每个逻辑缩进层级都对应再增加一个制表位。布局规则（花括号位置等）遵循 BSD 约定。

尽管提交的补丁并不绝对必须遵循这些格式规则，但这样做是个好主意。你的代码会经过 pgindent 处理，因此按另一套格式约定把它写得再漂亮也没有意义。

对于 Emacs，把下列内容（或类似内容）加入你的 `~/.emacs` 初始化文件：

```
;; check for files with a path containing "postgres" or "pgsql"
(setq auto-mode-alist
  (cons '("\\(postgres\\|pgsql\\).*\\.[ch]\\'" . pgsql-c-mode)
        auto-mode-alist))
(setq auto-mode-alist
  (cons '("\\(postgres\\|pgsql\\).*\\.cc\\'" . pgsql-c-mode)
        auto-mode-alist))

(defun pgsql-c-mode ()
  ;; sets up formatting for PostgreSQL C code
  (interactive)
  (c-mode)
  (setq-default tab-width 4)
  (c-set-style "bsd")             ; set c-basic-offset to 4, plus
 other stuff
  (c-set-offset 'case-label '+)   ; tweak case indent to match PG
 custom
  (setq indent-tabs-mode t))      ; make sure we keep tabs when
 indenting
```

对于 vi，你的 `~/.vimrc` 或等价文件应包含：

```
set tabstop=4
```

或在 vi 中等价地尝试

```
:set ts=4
```

文本浏览工具 more 和 less 可以这样调用：

```
more -x4
less -x4
```

以便正确显示制表符。

45.2. 在服务器内部报告错误

服务器代码内生成的错误、警告和日志消息应使用 `ereport`，或其更老的近亲 `elog` 来创建。这一函数的用法相当复杂，因此需要一些解释。

每条消息都必须包含两个元素：严重性级别（范围从 `DEBUG` 到 `PANIC`）以及主消息文本。此外还可以有可选元素，其中最常见的是遵循 SQL 规范 SQLSTATE 约定的错误标识符代码。`ereport`

本身只是一个包装函数，主要为了语法上的便利，使消息生成在 C 源代码中看起来像一次函数调用。ereport 唯一直接接受的参数是严重性级别。主消息文本以及任何可选消息元素都是通过在 ereport 调用中调用辅助函数（例如 errmsg）来生成的。

对ereport的一次典型调用可能如下：

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

这指定了错误严重性级别ERROR（一种普通错误）。errcode调用使用定义在src/include/utils/errcodes.h中的一个宏指定 SQLSTATE 错误代码。errmsg调用提供主消息文本。注意辅助函数调用外面多了一层圆括号 – 它们虽然烦人，但在语法上是必需的。

这里有一个更复杂的示例：

```
ereport(ERROR,
        (errcode(ERRCODE_AMBIGUOUS_FUNCTION),
         errmsg("function %s is not unique",
                func_signature_string(funcname, nargs,
                                      actual_arg_types)),
         errhint("Unable to choose a best candidate function. "
                 "You may need to add explicit typecasts.")));
```

这展示了如何用格式代码把运行时值嵌入消息文本中。此外还提供了一条可选的“提示”消息。

适用于 ereport 的辅助例程有：

- errcode(sqlerrcode) 为该条件指定 SQLSTATE 错误标识符代码。如果不调用这个例程，则默认错误标识符在错误严重性级别为 ERROR 或更高时为 ERRCODE_INTERNAL_ERROR，在错误级别为 WARNING 时为 ERRCODE_WARNING，否则（对于 NOTICE 及以下）为 ERRCODE_SUCCESSFUL_COMPLETION。虽然这些默认值常常很方便，但在省略 errcode() 调用之前，始终要先想想它们是否合适。

- errmsg(const char *msg, ...) 指定主错误消息文本，以及可能要插入其中的运行时值。插入项通过 sprintf 风格的格式代码指定。除了 sprintf 接受的标准格式代码外，还可以使用格式代码 %m 插入 strerror 为 errno 当前值返回的错误消息。[1] %m 不需要在 errmsg 的参数列表中有任何对应项。注意，在处理格式代码之前，消息字符串会先经过 gettext 以便可能进行本地化。

- errmsg_internal(const char *msg, ...) 与 errmsg 相同，只是消息字符串不会被翻译，也不会被收入国际化消息字典。这应当用于那些“不可能发生”、大概不值得为之投入翻译精力的情况。

- errdetail(const char *msg, ...) 提供一条可选的“详情”消息；当有额外信息但似乎不适合放在主消息中时，可使用它。消息字符串的处理方式与 errmsg 完全相同。

- errhint(const char *msg, ...) 提供一条可选的“提示”消息；当要就如何修复问题给出建议（而非关于出了什么问题的事实性详情）时，可使用它。消息字符串的处理方式与 errmsg 完全相同。

- errcontext(const char *msg, ...) 通常不会直接在 ereport 消息处调用；它用于 error_context_stack 回调函数中，提供错误发生时的上下文信息，例如 PL 函数中的当前

[1] 也就是说，是到达 ereport 调用点时的那个值；辅助报告例程内部对 errno 的更改不会影响它。如果显式写出 strerror(errno) 作为 errmsg 的参数列表内容，就不是这样了；因此不要这么做。

位置。消息字符串的处理方式与 `errmsg` 完全相同。与其他辅助函数不同，它在每次 `ereport` 调用中可以被调用多次；这样提供的 successive 字符串会用换行符分隔拼接起来。

- `errposition(int cursorpos)` 指定错误在查询字符串中的文本位置。目前，它只对在查询处理的词法和语法分析阶段检测到的错误有用。
- `errcode_for_file_access()` 是一个便捷函数，用于为与文件访问相关的系统调用失败选择合适的 SQLSTATE 错误标识符。它使用保存下来的 `errno` 来确定要生成哪种错误代码。通常应将它与主错误消息文本中的 `%m` 结合使用。
- `errcode_for_socket_access()` 是一个便捷函数，用于为与套接字相关的系统调用失败选择合适的 SQLSTATE 错误标识符。

还有一个较旧的函数`elog`，至今仍被大量使用。一个`elog`调用：

```
elog(level, "format string", ...);
```

完全等价于：

```
ereport(level, (errmsg_internal("format string", ...)));
```

注意，SQLSTATE 错误代码总会取默认值，而且消息字符串不会被翻译。因此，`elog`只应用于内部错误和低层调试日志。凡是普通用户可能感兴趣的消息，都应通过`ereport`。尽管如此，系统中仍有足够多的内部“不可能发生”错误检查，因此`elog`依然被广泛使用；对这类消息来说，由于记法更简洁，它更受青睐。

关于如何编写良好的错误消息，可参见第 45.3 节。

45.3. 错误消息风格指南

提供这份风格指南，是希望在 PostgreSQL 生成的所有消息中保持一致、对用户友好的风格。

什么内容放在哪里

主消息应简短、陈述事实，并避免提及具体函数名等实现细节。“简短”的意思是“在正常情况下应能放在一行里”。如果为了保持主消息简短，或者你觉得有必要提及特定失败的系统调用之类的实现细节，可使用详情消息。主消息和详情消息都应当陈述事实。提示消息则用于给出如何修复问题的建议，尤其是在该建议未必总是适用时。

例如，不要写：

```
IpcMemoryCreate: shmget(key=%d, size=%u, 0%o) failed: %m
(plus a long addendum that is basically a hint)
```

而要写：

```
Primary:    could not create shared memory segment: %m
Detail:     Failed syscall was shmget(key=%d, size=%u, 0%o).
Hint:       the addendum
```

原理：保持主消息简短，有助于使其切中要点，也便于客户端假定一行足以容纳错误消息，从而安排屏幕空间。详情消息和提示消息可以归入详细模式，或者放到弹出的错误详情窗口中。

此外，为节省空间，详情消息和提示消息通常会从服务器日志中省略。最好避免提及实现细节，因为用户反正也不知道这些细节。

格式

不要在消息文本中写入任何特定的格式假设。应预期客户端和服务器日志会按照各自需要折行。对于较长的消息，可以使用换行字符（\n）来表示建议的分段。不要让消息以换行结束。不要使用制表符或其他格式控制字符。（在错误上下文显示中，会自动添加换行以分隔函数调用等上下文层次。）

原理：消息不一定显示在终端类设备上。在 GUI 显示或浏览器中，这些格式指令充其量也只是被忽略。

引号

英文文本在适合引用时应使用双引号。其他语言的文本应一致地使用一种引号，这种引号应符合该语言的出版习惯以及其他程序的计算机输出习惯。

原理：选择双引号而不是单引号多少有些武断，但通常更符合首选用法。有人建议按照 SQL 约定，根据对象类型来选择引号的种类（即字符串用单引号，标识符用双引号）。但这是语言内部的技术问题，很多用户甚至并不熟悉；它也无法扩展到其他类型的被引术语，更无法翻译到其他语言中，而且本身也没什么意义。

引号的使用

对文件名、用户提供的标识符以及其他可能包含单词的变量，始终使用引号界定。对于不会包含单词的变量（例如操作符名），则不要加引号。

后端中有些函数会在需要时自行给其输出加上双引号（例如 `format_type_be()`）。不要再对这类函数的输出额外加引号。

原理：对象的名称在嵌入消息时可能造成歧义。对于插入的名称从哪里开始、到哪里结束，要保持一致。但也不要用不必要或重复的引号把消息弄得杂乱无章。

语法和标点

主错误消息与详情/提示消息的规则不同：

主错误消息：首字母不要大写。消息末尾不要加句号。更不要想着在消息末尾加感叹号。

详情消息和提示消息：使用完整句子，并且每条都用句号结束。句子的第一个词要首字母大写。

原理：避免使用标点，能让客户端应用更容易把消息嵌入各种语法上下文中。主消息往往本来也不是语法完整的句子。（而如果它们长到不止一句，就应当拆分成主消息和详情消息。）不过，详情消息和提示消息更长，也可能需要包含多个句子。为了保持一致，即便只有一个句子，它们也应遵循完整句子的风格。

大写与小写

消息措辞应使用小写，包括主错误消息的首字母。如果消息中出现 SQL 命令或关键字，则使用大写形式。

原理：这样更容易让所有消息看起来一致，因为有些消息是完整句子，而有些则不是。

避免被动语态

使用主动语态。有施动者时，用完整句子（“A could not do B”）。如果施动者就是程序本身，则使用无主语的电报式写法，但不要把程序写成“I”。

原理：程序不是人。不要假装它是。

现在时与过去时

如果一次尝试做某事失败了，但下次仍可能成功（也许在修复某个问题之后），就使用过去时。如果失败显然是永久性的，就使用现在时。

下面两种句式在语义上有明显差别：

```
could not open file "%s": %m
```

和：

```
cannot open file "%s"
```

第一种表示尝试打开文件失败了。消息应给出原因，例如“磁盘已满”或“文件不存在”。过去时更合适，因为下次磁盘可能就不满了，或者所请求的文件可能已经存在。

第二种形式表示，程序中根本不存在打开该命名文件的功能，或者从概念上就不可能。现在时更合适，因为这种情况会无限期持续下去。

原理：诚然，普通用户未必能仅凭消息的时态得出什么重要结论，但既然语言提供了语法，我们就应当正确使用它。

对象的类型

在引用对象名称时，要说明它是什么类型的对象。

原理：否则没人会知道“foo.bar.baz”指的是什么。

括号

方括号只应用于：（1）命令概要中表示可选参数，或（2）表示数组下标。

原理：其他任何用法都不符合广为人知的习惯用法，而且会让人困惑。

错误消息的组装

当消息中包含其他地方生成的文本时，应按以下方式嵌入：

```
could not open file %s: %m
```

原理：要把这种文本拼进一个流畅的单句中，同时照顾所有可能的错误代码会很困难，因此需要某种标点。也有人建议把嵌入的文本放在圆括号里，但如果嵌入的文本很可能是消息中最重要的部分（而这往往确实如此），那样就显得不自然了。

错误的原因

消息总应说明错误发生的原因。例如：

```
BAD:    could not open file %s
BETTER: could not open file %s (I/O failure)
```

如果不知道原因，最好去修复代码。

函数名

不要在错误文本中包含报告该错误的例程名。需要时，我们有其他机制可以找出它，而且对大多数用户来说，这项信息没有帮助。如果去掉函数名后错误文本就不够清楚，请重新措辞。

```
BAD:    pg_atoi: error in "z": can't parse "z"
BETTER: invalid input syntax for integer: "z"
```

也要避免提及被调用函数的名称；应改为说明代码想要做什么：

```
BAD:    open() failed: %m
BETTER: could not open file %s: %m
```

如果确实有必要，可在详情消息中提及系统调用。（在某些情况下，在详情消息中提供传给该系统调用的实际值也许是合适的信息。）

原理：用户并不知道那些函数都做了什么。

应避免的棘手词语

Unable. “Unable” 几乎就是被动语态。更好的做法是视情况使用 “cannot” 或 “could not” 。

Bad. 像 “bad result” 这样的错误消息，实在很难被合理理解。最好写出结果为什么是 “bad” ，例如 “invalid format” 。

Illegal. “Illegal” 指的是违法，其余情况应使用 “invalid” 。更好的是，直接说明为什么无效。

Unknown. 尽量避免 “unknown” 。考虑 “error: unknown response” 。如果你都不知道响应是什么，又如何知道它有错？ “Unrecognized” 往往是更好的选择。此外，一定要包含被抱怨的值。

```
BAD:    unknown node type
BETTER: unrecognized node type: 42
```

Find vs. Exists. 如果程序为了定位某个资源使用了非平凡算法（例如路径搜索），而该算法失败了，那么说程序没能 “find” 该资源是公平的。另一方面，如果资源的预期位置已知，但程序无法在那里访问它，那么就应说该资源并不 “exist” 。在这种情况下使用 “find” 显得含糊，也会混淆问题。

正确拼写

单词应完整拼写。例如，避免使用：

- spec
- stats
- parens

- auth
- xact

原理：这会提升一致性。

本地化

请记住，错误消息文本需要被翻译成其他语言。请遵循第 46.2.2 节中的指导，避免给翻译者制造麻烦。

第 46 章 本地语言支持

Peter Eisentraut

46.1. 给翻译者

PostgreSQL 程序（服务器和客户端）可以用你喜欢的语言发出消息——前提是消息已被翻译。创建和维护翻译的消息集需要语言说得好并想为 PostgreSQL 做贡献的人的帮助。做这件事你完全不必是程序员。本节说明如何提供帮助。

46.1.1. 要求

我们不会评判你的语言能力——本节讲的是软件工具。理论上，你只需要一个文本编辑器。
但这只在你不想试用翻译的消息这种不太可能的情况下成立。配置源代码树时，请务必使用 `--enable-nls` 选项。这还会检查 libintl 库和 `msgfmt` 程序，所有最终用户反正都需要它们。要试用你的工作成果，请按安装说明中适用的部分操作。

如果你想开始一项新的翻译工作，或者想执行一次消息目录合并（后文会描述），则分别需要 GNU 兼容实现的 `xgettext` 和 `msgmerge` 程序。今后我们会尽量做到：如果你使用打包的源码发布版，就不需要 `xgettext` 了。（如果从 CVS 工作，你仍然需要它。）目前推荐使用 GNU Gettext 0.10.36 或更高版本。

你本地的 gettext 实现应该自带文档。其中有些内容可能与下文重复，但要了解更多细节，你应当去查阅那些文档。

46.1.2. 概念

原始（英语）消息及其（可能存在的）译文对应项，以成对形式保存在消息目录中；
每个程序以及每种目标语言各有一份消息目录（尽管相关程序可以共享同一个消息目录）。
消息目录有两种文件格式：第一种是“PO”文件（Portable Object，可移植对象），它是带有特殊语法的纯文本文件，由翻译者编辑。第二种是“MO”文件（Machine Object，机器对象），它是根据相应的 PO 文件生成的二进制文件，在国际化后的程序运行时使用。翻译者不需要处理 MO 文件；事实上几乎没有人需要。

消息目录文件的扩展名，毫不意外地是 `.po` 或 `.mo`。基本名则视情况而定，要么是其所对应的程序名，要么是该文件所对应的语言。这多少有些令人困惑。例子包括 `psql.po`（psql 的 PO 文件）或 `fr.mo`（法语的 MO 文件）。

PO 文件的格式示例如下：

```
# comment

msgid "original string"
msgstr "translated string"

msgid "more original"
msgstr "another translated"
"string can be broken up like this"

...
```

msgid 行是从程序源代码中提取出来的。（并非一定如此，但这是最常见的做法。）msgstr 行起初为空，由翻译者填入有意义的字符串。字符串可以包含 C 风格的转义字符，也可以像上例那样跨多行延续。（下一行必须从行首开始。）

字符引入注释。如果 # 字符后面紧跟空白，那么这就是由翻译者维护的注释。也可能存在自动注释，即 # 后紧跟非空白字符的注释。这些注释由各种处理 PO 文件的工具维护，目的是帮助翻译者。

```
#. automatic comment
#: filename.c:1023
#, flags, flags
```

#. 风格的注释是从使用该消息的源文件中提取出来的。程序员可能已经为翻译者插入了信息，例如期望的对齐方式。#: 注释指出该消息在源代码中使用的精确位置。翻译者不必查看程序源代码，但如果对正确译法有疑问，也可以去看。#, 注释包含以某种方式描述该消息的标志。目前有两种标志：如果某条消息可能因程序源代码的更改而过期，则会设置 `fuzzy`。翻译者随后可以核实这一点，并在可能时移除 fuzzy 标志。注意，fuzzy 消息不会提供给终端用户。另一个标志是 `c-format`，它表示该消息是一个 `printf` 风格的格式模板。这意味着译文也应该是一个格式字符串，并具有相同数量和类型的占位符。有一些工具会据此验证格式字符串，它们就是根据 c-format 标志来检查的。

46.1.3. 创建和维护消息目录

好，那么该如何创建一个“空白”消息目录呢？首先，进入包含你想翻译其消息的程序的目录。如果那里有一个 `nls.mk` 文件，就说明这个程序已经为翻译做好准备了。

如果已经有一些 `.po` 文件，那么就说明已经有人做过部分翻译工作。这些文件命名为 *language*`.po`，其中 *language* 是 ISO 639-1 两字母语言代码（小写）[1]，例如法语对应 `fr.po`。如果某种语言确实需要进行多套翻译工作，这些文件也可以命名为 *language_region*`.po`，其中 *region* 是 ISO 3166-1 两字母国家代码（大写）[2]，例如巴西葡萄牙语对应 `pt_BR.po`。如果你找到了自己想要的语言，就可以直接开始编辑那个文件。

如果需要启动一项新的翻译工作，请先运行命令：

```
gmake init-po
```

这会创建文件*progname*`.pot`。（用`.pot`来与“正在使用中”的 PO 文件区分，`T`代表“模板”。）把这个文件复制为*language*`.po`并编辑它。为了让别人知道新语言已可用，还要编辑文件`nls.mk`，把语言（或语言和国家）代码加到形如下面的行中：

```
AVAIL_LANGUAGES := de fr
```

（当然也可以是其他语言。）

随着底层程序或库的变化，程序员可能会更改或添加消息。这种情况下不必从头开始，只需运行命令：

```
gmake update-po
```

该命令会创建一个新的空白消息目录文件（即你最初使用的 pot 文件），并把它与现有的 PO 文件合并。如果合并算法对某个消息没有把握，就会如上所述把它标记为“模糊”。万一哪里真的出了问题，旧的 PO 文件会以`.po.old`扩展名保存。

[1] http://lcweb.loc.gov/standards/iso639-2/englangn.html
[2] http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/en_listp1.html

46.1.4. 编辑 PO 文件

PO 文件可以用普通文本编辑器编辑。翻译者只应修改 msgstr 指令后引号之间的内容、添加注释，以及调整 fuzzy 标志。Emacs 里（不出意外地）有一个 PO 模式，相当有用。

PO 文件不必完全填满。如果没有可用译文（或译文为空），软件会自动回退到原始字符串。把不完整的翻译提交到源码树中也没有问题；这样别人就有机会接手你的工作。不过，完成一次合并之后，建议优先清除 fuzzy 条目。记住，fuzzy 条目不会被安装；它们只用来作为可能正确译文的参考。

编辑译文时要记住以下几点：

- 务必确保如果原文以换行结束，译文也同样以换行结束。制表符等也是如此。
- 如果原文是一个 `printf` 格式字符串，译文也必须如此。译文还需要按相同顺序保留相同的格式说明符。有时语言自身的规则会让这一点变得不可能，或者至少很别扭。在这种情况下，你可以像下面这样修改格式说明符：

  ```
  msgstr "Die Datei %2$s hat %1$u Zeichen."
  ```

  这样一来，第一个占位符实际上会使用参数列表中的第二个参数。*digits*`$` 必须紧跟在 `%` 后面，位于其他任何格式修饰符之前。（这个特性确实存在于 `printf` 函数族中。你以前可能没听说过它，因为在消息国际化之外，它几乎没有用武之地。）
- 如果原始字符串包含语言错误，就报告它（或者你自己在程序源代码中修正它），然后照常翻译。等程序源代码更新之后，修正过的字符串就可以被合并进来。如果原始字符串包含事实性错误，就报告它（或者你自己修正它），并且不要翻译它。相反，你可以在 PO 文件里为该字符串加上一条注释。
- 保持原始字符串的风格和语气。特别是，那些不是完整句子的消息（`cannot open file %s`）通常不应以大写字母开头（如果你的语言区分大小写），也不应以句号结尾（如果你的语言使用标点符号）。阅读第 45.3 节可能会有帮助。
- 如果你不知道某条消息是什么意思，或者它有歧义，就到开发者邮件列表里提问。很可能说英语的终端用户也同样看不懂它，或者会觉得它有歧义，因此最好直接改进那条消息。

46.2. 给程序员

46.2.1. 实现机制

本节描述如何在 PostgreSQL 发行版中的程序或库里实现本地语言支持。目前它只适用于 C 程序。

为程序添加 NLS 支持

1. 把下面的代码插入到该程序的启动序列中：

   ```
   #ifdef ENABLE_NLS
   #include <locale.h>
   #endif

   ...
   ```

```
#ifdef ENABLE_NLS
setlocale(LC_ALL, "");
bindtextdomain("progname", LOCALEDIR);
textdomain("progname");
#endif
```

（其中 *progname* 实际上可以自由选择。）

2. 凡是遇到可能需要翻译的消息，都需要插入对 `gettext()` 的调用。例如：

   ```
   fprintf(stderr, "panic level %d\n", lvl);
   ```

   将改成：

   ```
   fprintf(stderr, gettext("panic level %d\n"), lvl);
   ```

   （如果没有配置 NLS 支持，`gettext` 会被定义成一个空操作。）

   这样往往会增加很多杂乱内容。一个常用的简写是：

   ```
   #define _(x) gettext(x)
   ```

   如果该程序的大部分通信都是通过一个或少数几个函数完成的，例如后端中的 `ereport()`，另一种可行的解决办法是让这个函数在内部对所有输入字符串调用 `gettext`。

3. 在程序源代码所在的目录中添加一个文件 `nls.mk`。这个文件将被当作一个 makefile 读取。这里需要进行下列变量赋值：

   `CATALOG_NAME`

   程序名，即 `textdomain()` 调用中提供的名称。

   `AVAIL_LANGUAGES`

   已有翻译的列表——开始时为空。

   `GETTEXT_FILES`

   包含可翻译字符串的文件列表，也就是那些使用 `gettext` 或其他替代方案标记过的文件。最终，这会包含该程序几乎所有的源文件。如果这个列表太长，可以让第一个“文件”是一个 `+`，第二个词则是一个文件名，该文件每行包含一个文件名。

   `GETTEXT_TRIGGERS`

   为翻译者生成消息目录的工具需要知道哪些函数调用包含可翻译字符串。默认只认识 `gettext()` 调用。如果你使用了 `_` 或其他标识符，就需要在这里列出它们。如果可翻译字符串不是第一个参数，条目就需要写成 `func:2` 这样的形式（表示第二个参数）。

构建系统会自动处理消息目录的构建和安装。

46.2.2. 消息编写指南

下面列出一些便于翻译的消息编写指南。

- 不要像下面这样在运行时拼接句子：

```
printf("Files were %s.\n", flag ? "copied" : "removed");
```

句子中的词序在其他语言里可能完全不同。另外，即使你记得对每个片段都调用 gettext()，这些片段分开来看也可能难以妥善翻译。最好稍微重复一点代码，
让每条待翻译消息都成为一个语义完整的整体。只有数字、
文件名之类的运行时变量才应该在运行时插入消息文本中。

- 出于类似的原因，下面这样也行不通：

```
printf("copied %d file%s", n, n!=1 ? "s" : "");
```

因为它假定了复数形式的构造方式。如果你以为可以这样解决：

```
if (n==1)
    printf("copied 1 file");
else
    printf("copied %d files", n):
```

那就会失望了。有些语言的复数形式不止两种，而且规则相当特殊。
通常最好从设计上彻底避开这个问题，例如这样：

```
printf("number of copied files: %d", n);
```

- 如果你想向翻译者传达某些信息，例如一条消息应如何与其他输出对齐，
就在该字符串出现之前放置一个以 `translator` 开头的注释，例如：

```
/* translator: This message is not what it seems to be. */
```

这些注释会被复制到消息目录文件中，这样翻译者就能看到它们。

第 47 章 编写过程语言调用处理器

凡是不是使用当前针对编译型语言的“版本 1”接口编写的函数，在被调用时都会经过该语言专用的调用处理器函数。这包括用户定义过程语言中的函数、用 SQL 编写的函数，以及使用版本 0 编译语言接口的函数。调用处理器负责以恰当的方式执行该函数，例如解释所提供的源文本。本章概述如何编写新的过程语言调用处理器。

过程语言的调用处理器是一个“普通”函数，必须使用诸如 C 这样的编译型语言、按照版本 1 接口编写，并在 PostgreSQL 中注册为不接受参数且返回 `language_handler` 类型。这个特殊伪类型会将该函数标识为调用处理器，并阻止它在 SQL 命令中被直接调用。

调用处理器的调用方式与任何其他函数相同：它接收一个指向 `FunctionCallInfoData struct` 的指针，其中包含参数值和被调用函数的信息，并且它应返回一个 `Datum` 结果（如果希望返回 SQL 空值结果，还可以设置 `FunctionCallInfoData` 结构的 `isnull` 字段）。调用处理器与普通被调用函数的区别在于：`FunctionCallInfoData` 结构的 `flinfo->fn_oid` 字段包含的是要被调用的实际函数的 OID，而不是调用处理器本身的 OID。调用处理器必须用这个字段来确定要执行哪个函数。此外，传递的参数列表是按照目标函数的声明设置的，而不是按照调用处理器设置的。

从系统表 `pg_proc` 中取出函数的条目并分析被调用函数的参数和返回类型是调用处理器的职责。函数的 `CREATE FUNCTION` 中的 `AS` 子句可以在 `pg_proc` 行的 `prosrc` 列中找到。它可能是过程语言自身的源文本（比如 PL/Tcl）、文件的路径名，或者任何告诉调用处理器具体做什么的其他东西。

同一个函数在执行一条 SQL 语句期间往往会被调用很多次。调用处理器可以利用 `flinfo->fn_extra` 字段，避免重复查找被调用函数的信息。该字段起初为 `NULL`，但调用处理器可以把它设置为指向与被调用函数有关的信息。在后续调用中，如果 `flinfo->fn_extra` 已经不是 `NULL`，就可以直接使用它并跳过信息查找步骤。调用处理器必须确保 `flinfo->fn_extra` 指向的内存至少能存活到当前查询结束，因为 `FmgrInfo` 数据结构可能会保留这么久。一种做法是在 `flinfo->fn_mcxt` 指定的内存上下文中分配这些额外数据；这类数据通常会与 `FmgrInfo` 本身具有相同的生命周期。不过，处理器也可以选择使用生命周期更长的内存上下文，以便跨查询缓存函数定义信息。

过程语言函数作为触发器调用时，不会按通常方式传入参数，但 `FunctionCallInfoData` 的 `context` 字段会指向一个 `TriggerData` 结构体，而不是像普通函数调用那样为 `NULL`。语言调用处理器应提供让过程语言函数获取触发器信息的机制。

下面是一个用 C 编写的过程语言处理器的模板：

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
```

```
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * Called as a trigger procedure
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        retval = ...
    }
    else
    {
        /*
         * Called as a function
         */

        retval = ...
    }

    return retval;
}
```

只需用几千行代码代替省略号补充进去，就完成了这个调用处理器。

将处理器函数编译为可加载模块后（参见第 33.7.6 节），可用以下命令注册示例过程语言：

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'filename'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

第 48 章 遗传查询优化器

Martin Utesch, University of Mining and Technology

作者

由 Martin Utesch <utesch@aut.tu-freiberg.de> 为德国弗赖贝格矿业和技术大学自动控制研究所编写。

48.1. 将查询处理看成是一个复杂的优化问题

在所有关系操作符中，最难处理和优化的是连接。随着查询中连接数目的增加，可能的查询计划数量会呈指数增长。为了处理单个连接而支持多种连接方法（例如 PostgreSQL 中的嵌套循环、哈希连接和归并连接）来处理单个连接，以及作为关系访问路径的多种索引（例如 PostgreSQL 中的 R-树、B-树和哈希），也进一步增加了优化工作量。

常规的PostgreSQL查询优化器会在可选策略空间中执行近似穷举搜索。该算法最早出现在“System R”数据库中，能够产生近乎最优的连接顺序；但当查询中的连接数变得很大时，它可能耗费极大的时间和内存空间。这使得普通的PostgreSQL查询优化器不适合诸如人工智能这类需要大量查询的数据库应用领域。

德国弗赖贝格矿业和技术大学自动控制研究所在尝试将 PostgreSQL 用作一个用于电网维护的基于知识的决策支持系统后端时遇到了一些问题。该 DBMS 需要为该基于知识系统的推理机处理大型连接查询。

在探索可能查询计划空间方面的性能困难，催生了开发一种新优化技术的需求。

下文将介绍一种遗传算法的实现，它以适合涉及大量连接的查询的高效方式来解决连接顺序问题。

48.2. 遗传算法

遗传算法（GA）是一种通过确定的、随机化的搜索进行工作的启发式优化方法。优化问题的可能解集合被视为由若干个体组成的种群。个体对其环境的适应程度由其适应度表示。

一个个体在搜索空间中的坐标由染色体表示，本质上是一组字符串。基因是染色体的一个片段，它编码某个待优化参数的值。基因的典型编码可以是二进制或整数。

通过模拟重组、变异和选择这些进化操作，可以找到平均适应度高于前代的新一代搜索点。

根据comp.ai.genetic FAQ 中的说法，再怎么强调也不过分：GA并不是为了求解问题而进行的纯粹随机搜索。GA会使用随机过程，但其结果显然并非随机的（优于随机）。

图 48.1. 遗传算法的结构图

P(t)	时刻 t 的祖先代
P''(t)	时刻 t 的后代代

```
+=========================================+
|>>>>>>>>>>>  Algorithm GA  <<<<<<<<<<<<<<|
+=========================================+
| INITIALIZE t := 0                       |
+=========================================+
| INITIALIZE P(t)                         |
+=========================================+
| evaluate FITNESS of P(t)                |
+=========================================+
| while not STOPPING CRITERION do         |
|   +-------------------------------------+
|   | P'(t)  := RECOMBINATION{P(t)}       |
|   +-------------------------------------+
|   | P''(t) := MUTATION{P'(t)}           |
|   +-------------------------------------+
|   | P(t+1) := SELECTION{P''(t) + P(t)}  |
|   +-------------------------------------+
|   | evaluate FITNESS of P''(t)          |
|   +-------------------------------------+
|   | t := t + 1                          |
+===+=====================================+
```

48.3. PostgreSQL 中的遗传查询优化（GEQO）

GEQO模块旨在以类似于旅行商问题（TSP）的方式解决查询优化问题。
可能的查询计划被编码为整数串。每个串表示查询中从一个关系到下一个关系的连接顺序。
例如，查询树

```
   /\
  /\ 2
 /\ 3
4  1
```

会被编码为整数串 '4-1-3-2'，这意味着先连接关系 '4' 和 '1'，再连接 '3'，最后连接 '2'；其中 1、2、3、4 是PostgreSQL优化器内部的关系 ID。

GEQO模块的部分内容改编自 D. Whitley 的 Genitor 算法。

PostgreSQL 中 GEQO 实现的具体特征是：

- 采用稳态 GA（替换种群中适应度最低的个体，而不是整代替换），能够快速收敛到更优的查询计划。这对于在合理时间内处理查询至关重要；
- 采用边重组交叉，它特别适合在利用 GA 求解 TSP 时将边损失保持在较低水平；
- 不使用变异作为遗传操作符，因此无须借助修复机制来生成合法的 TSP 回路。

GEQO模块使PostgreSQL查询优化器能够通过非穷举搜索有效支持大型连接查询。

48.3.1. PostgreSQL GEQO 的未来实现任务

为了改进遗传算法的参数设置，仍有一些工作要做。在文件 `backend/optimizer/geqo/geqo_params.c` 中的例程`gimme_pool_size`和`gimme_number_generations`里，我们必须为参数设置找到一种折中，以满足两个相互竞争的需求：

- 查询计划的最优性
- 计算时间

48.4. 进一步阅读

下列资源包含关于遗传算法的更多信息：

- 《Hitch-Hiker 演化计算指南》[1]（comp.ai.genetic[2] 的 FAQ）
- 《演化计算及其在艺术和设计中的应用》[3]，Craig Reynolds 著
- [elma99]
- [fong]

[1] http://surf.de.uu.net/encore/www/
[2] news://comp.ai.genetic
[3] http://www.red3d.com/cwr/evolve.html

第 49 章 索引代价估计函数

作者

由 Tom Lane（<tgl@sss.pgh.pa.us>）于 2000-01-24 撰写

注意

这些内容最终应成为关于编写新索引访问方法的更大章节的一部分。

每种索引访问方法都必须提供一个供规划器/优化器使用的代价估计函数。该函数的过程 OID 记录在访问方法的 pg_am 项的 amcostestimate 字段中。

注意

在 PostgreSQL 7.0 之前，注册索引特定的代价估计函数使用的是另一种方案。

amcostestimate 函数会得到一个已判定可用于该索引的 WHERE 子句列表。它必须返回访问索引的代价估计以及 WHERE 子句的选择率估计（即在索引扫描期间将被检索的主表行所占的比例）。对于简单情况，代价估计器几乎全部工作都可以通过调用优化器中的标准例程完成；设置 amcostestimate 函数的意义在于让索引访问方法提供索引类型特定的知识，以便在可能改进标准估计时加以利用。

每个 amcostestimate 函数必须具有如下签名：

```
void
amcostestimate (Query *root,
                RelOptInfo *rel,
                IndexOptInfo *index,
                List *indexQuals,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

前四个参数是输入：

root

正在处理的查询。

rel

索引所在的关系。

index

索引本身。

indexQuals

索引限定条件子句的列表（隐含 AND 连接）；NIL 列表表示没有可用的限定条件。

后四个参数是按引用传递的输出：

*indexStartupCost

设置为索引启动处理的代价

*indexTotalCost

设置为索引处理的总代价

*indexSelectivity

设置为索引选择率

*indexCorrelation

设置为索引扫描顺序与底层表顺序之间的相关系数

注意，代价估计函数必须用 C 编写，而不能用 SQL 或任何可用的过程语言，因为它们必须访问规划器/优化器的内部数据结构。

索引访问代价应按 `src/backend/optimizer/path/costsize.c` 所用的单位计算：顺序磁盘块读取的代价为 1.0，非顺序读取的代价为 random_page_cost，而处理一个索引行的代价通常取为 cpu_index_tuple_cost（这是一个用户可调的优化器参数）。此外，对于索引处理期间调用的任何比较操作符（尤其是 indexQuals 本身的求值），应收取适当的 cpu_operator_cost 倍数。

访问代价应包括扫描索引本身相关的所有磁盘和 CPU 代价，但不包括检索或处理由该索引标识的主表行的代价。

“启动代价”是总扫描代价中在可以开始取第一行之前必须付出的部分。
对大多数索引来说可以取零，但启动代价很高的索引类型可能希望把它设为非零。

indexSelectivity 应设置为索引扫描期间将检索的主表行的估计比例。对于有损索引，这通常会高于实际满足给定限定条件的行所占的比例。

indexCorrelation 应设置为索引顺序与表顺序之间的相关性（取值范围在 -1.0 到 1.0 之间）。它用于调整从主表取行的代价估计。

代价估计

典型的代价估计器按如下步骤进行：

1. 基于给定的限定条件，估计并返回将被访问的主表行所占的比例。
   在没有索引类型特定知识的情况下，使用标准优化器函数 `clauselist_selectivity()`：

   ```
   *indexSelectivity = clauselist_selectivity(root, indexQuals,
                                              rel->relid, JOIN_INNER)
   ;
   ```

2. 估计扫描期间将访问的索引行数。对许多索引类型来说，它等于 indexSelectivity 乘以索引中的行数，但也可能更多。（注意，索引的页数和行数可以从 IndexOptInfo 结构中获得。）

3. 估计扫描期间将检索的索引页数。它可以就是 indexSelectivity 乘以索引的页数。
4. 计算索引访问代价。一个通用估计器可以这样做：

```
    /*
     * Our generic assumption is that the index pages will be read
     * sequentially, so they have cost 1.0 each, not random_page_
cost.
     * Also, we charge for evaluation of the indexquals at each
 index row.
     * All the costs are assumed to be paid incrementally during
 the scan.
     */
    cost_qual_eval(&index_qual_cost, indexQuals);
    *indexStartupCost = index_qual_cost.startup;
    *indexTotalCost = numIndexPages +
        (cpu_index_tuple_cost + index_qual_cost.per_tuple) * num
IndexTuples;
```

5. 估计索引相关性。对于单字段上的简单有序索引，可以从 pg_statistic 中取得相关性。如果相关性未知，保守的估计是零（无相关）。

代价估计器函数的例子可以在 `src/backend/utils/adt/selfuncs.c`中找到。

按照惯例，`amcostestimate`函数的 `pg_proc` 项应显示八个全声明为 `internal` 的参数（因为它们都没有 SQL 已知的类型），而返回类型为 `void`。

第 50 章 GiST 索引

50.1. 介绍

GiST 是广义搜索树（Generalized Search Tree）的缩写。它是一种平衡的树结构访问方法，充当实现任意索引方案的基础模板。B+ 树、R 树和许多其他索引方案都可以在 GiST 中实现。

GiST的一个优点是，它使数据类型所属领域的专家而不是数据库专家，
能够开发带有适当访问方法的自定义数据类型。

这里的一些内容来自加州大学伯克利分校的 GiST 索引项目网站[1]以及 Marcel Kornacker 的论文 Access Methods for Next-Generation Database Systems[2]。PostgreSQL中的GiST 实现主要由 Teodor Sigaev 和 Oleg Bartunov 维护，他们的网站http://www.sai.msu.su/~megera/postgres/gist/上还有更多信息。

50.2. 可扩展性

传统上，实现一种新的索引访问方法意味着大量艰难的工作。必须理解数据库的内部机制，
例如锁管理器和预写式日志。GiST接口具有很高的抽象层次，
只要求访问方法实现者实现被访问数据类型的语义。GiST层本身会处理并发、
日志记录以及树结构的搜索。

这种可扩展性不应与其他标准搜索树在可处理数据方面的可扩展性相混淆。例如，
PostgreSQL支持可扩展的 B+-树和 R 树。这意味着你可以用PostgreSQL在任意数据类型上构建 B+-树或 R 树。但 B+-树只支持范围谓词（<、=、>），而 R 树只支持 n 维范围查询（包含、被包含、相等）。

因此，如果你用PostgreSQL的 B+-树为一个图像集合建立索引，你只能发出诸如“imagex 是否等于 imagey”、“imagex 是否小于 imagey”以及“imagex 是否大于 imagey”之类的查询。取决于你如何在这种上下文中定义“等于”、“小于”和“大于”，这可能仍然有用。不过，
使用基于GiST的索引，你就可以构造出能够提出特定领域问题的查询方式，
例如“找出所有马的图片”或者“找出所有曝光过度的图片”。

要让一个 GiST 访问方法运行起来，只需实现七个用户定义的方法，这些方法定义了树中键的行为。当然，要支持复杂查询，这些方法本身也必须足够巧妙；但对于所有标准查询（B+-树、R 树等），它们都相对直接。简而言之，GiST 把可扩展性与通用性、代码复用以及清晰的接口结合了起来。

50.3. 实现

一个GiST索引操作符类必须提供七个方法：

consistent

给定树页面上的一个谓词p和一个用户查询q，如果可以确定对于给定的数据项p和q
不可能同时为真，该方法将返回假。

union

此方法合并树中的信息。给定一组项，该函数生成一个对所有这些项都为真的新谓词。

[1] http://gist.cs.berkeley.edu/
[2] http://citeseer.nj.nec.com/448594.html

compress

将一个数据项转换成适合在索引页中物理存储的格式。

decompress

`compress`方法的逆操作。将数据项的索引表示转换成数据库能够操作的格式。

penalty

返回一个值，指示把新项插入树的特定分支的“代价”。项将被沿着树中`penalty`最小的路径插入。

picksplit

当必须进行页面分裂时，该函数决定页面上哪些项留在旧页面，哪些项移动到新页面。

same

如果两个项完全相同则返回真，否则返回假。

50.4. 限制

目前PostgreSQL中的GiST实现还有一些重大限制：GiST访问不支持并发；GiST接口不允许开发某些数据类型，例如数字树（digital trees，见 Aoki 等人的论文）；并且尚不支持对GiST索引更新的预写式日志。

并发问题的解决方案出现在 Marcel Kornacker 的论文中，但这些思想尚未在PostgreSQL实现中付诸实践。

缺少预写式日志只是一个小的编程问题，但由于尚未完成，一次崩溃可能使GiST索引变得不一致，从而不得不 REINDEX。

50.5. 示例

要查看使用GiST实现的索引方法的示例，请考察下列 contrib 模块：

btree_gist

B-Tree

cube

多维立方体的索引

intarray

一维 int4 值数组的 RD 树

ltree

树状结构的索引

rtree_gist

R-Tree

seg

“float 范围” 的存储和索引访问

tsearch and tsearch2

全文索引

第 51 章 页文件

对数据库文件页面格式的说明。

本节概述 PostgreSQL 表和索引内部所使用的页面格式。（索引访问方法不要求必须使用这种页面格式。现有的所有索引方法都使用这种基本格式，
但索引元页中保存的数据通常并不完全遵循项布局规则。）TOAST 表和序列的格式与普通表相同。

在下面的说明中，假定一个字节包含 8 个位。另外，术语项指的是存储在页面上的单个数据值。在表中，项是一行；在索引中，项是一条索引条目。

表 51.1展示了页面的基本布局。每个页面包含五个部分。

表 51.1. 页面布局示例

项	描述
PageHeaderData	长度为 20 字节。包含页面的一般信息，包括空闲空间指针。
ItemIdData	指向实际项的（偏移量、长度）对数组。
空闲空间	尚未分配的空间。所有新行都从这里分配，通常从末尾开始。
项	项本身。
特殊空间	索引访问方法特有的数据。不同方法存储不同数据。普通表中为空。

每个页面的前 20 个字节由页头（PageHeaderData）组成。它的格式详见表 51.2。前两个字段处理与 WAL 有关的内容。接下来是三个 2 字节整数字段（`pd_lower`，`pd_upper`，和 `pd_special`）。它们分别表示到未分配空间起始位置、到未分配空间结束位置以及到特殊空间起始位置的字节偏移量。

表 51.2. PageHeaderData 布局

字段	类型	长度	描述
pd_lsn	XLogRecPtr	8 字节	LSN：xlog 最后一个字节之后的下一个字节
pd_sui	StartUpID	4 字节	最近更改的 SUI（目前只有堆 AM 使用它）
pd_lower	LocationIndex	2 字节	到空闲空间起始位置的偏移量。
pd_upper	LocationIndex	2 字节	到空闲空间结束位置的偏移量。
pd_special	LocationIndex	2 字节	到特殊空间起始位置的偏移量。
pd_pagesize_version	uint16	2 字节	页面大小和布局版本号信息。

所有细节都可以在 src/include/storage/bufpage.h 中找到。

特殊空间是页面末尾的一个区域，它在页面初始化时分配，并包含某种访问方法特有的信息。
页头的最后 2 个字节 `pd_pagesize_version` 同时存储页面大小和一个版本指示器。从

PostgreSQL 7.3 起版本号为 1；更早的版本使用版本号 0。（基本页面布局和页头格式并未改变，但堆行头部的布局发生过变化。）页面大小基本上只是用于交叉检查；
一个安装中并不支持同时存在多种页面大小。

页头之后是项标识符（ItemIdData），每个需要四个字节。
一个项标识符包含项起始位置的字节偏移量、其字节长度，以及若干影响解释方式的属性位。
新的项标识符会根据需要从未分配空间的起始处分配。当前已有多少个项标识符，可以通过查看 pd_lower 得知；分配新标识符时它会增加。因为一个项标识符在被释放之前永远不会移动，所以即使项本身为了压缩空闲空间而在页面内移动，其索引仍然可以被长期用来引用该项。
事实上，每个指向项的指针（ItemPointer，也称 CTID）在 PostgreSQL 中都由页号和项标识符索引构成。

项本身存储在从未分配空间末尾开始、向后分配的区域中。
其确切结构取决于表要包含什么内容。表和序列都使用一种名为 HeapTupleHeaderData 的结构体，如下所述。

最后一部分是“特殊部分”，其中可以存放访问方法希望保存的任何内容。
普通表完全不使用特殊部分（通过将 pd_special 设为等于页面大小来表示）。

所有表行的结构都相同。它们都有一个固定大小的头部（在大多数机器上占 23 字节），后面跟着可选的空值位图、可选的对象 ID 字段以及用户数据。头部的详细格式见表 51.3。实际用户数据（行的各列）从 t_hoff 指示的偏移位置开始，它必须始终是该平台 MAXALIGN 对齐单位的整数倍。只有当 HEAP_HASNULL 位在 t_infomask 中被置位时，空值位图才存在。若存在，它紧随固定头部之后，并占用足够多的字节，以便为每个数据列提供一位（也就是说，总共为 t_natts 位）。在这个位图中，1 表示非空，0 表示空值。当位图不存在时，假定所有列都非空。只有当 HEAP_HASOID 位在 t_infomask 中被置位时，对象 ID 才存在。若存在，它位于 t_hoff 边界之前。为了让 t_hoff 成为 MAXALIGN 的整数倍所需的任何填充，都会出现在空值位图与对象 ID 之间。（这又反过来保证了对象 ID 的对齐是合适的。）

表 51.3. HeapTupleHeaderData 布局

字段	类型	长度	描述
t_xmin	TransactionId	4 字节	插入 XID 戳
t_cmin	CommandId	4 字节	插入 CID 戳（与 t_xmax 共用存储）
t_xmax	TransactionId	4 字节	删除 XID 戳
t_cmax	CommandId	4 字节	删除 CID 戳（与 t_xvac 共用存储）
t_xvac	TransactionId	4 字节	VACUUM 操作移动某个行版本时的 XID
t_ctid	ItemPointerData	6 字节	本行版本或更新行版本的当前 TID
t_natts	int16	2 字节	属性数量
t_infomask	uint16	2 字节	各种标志位
t_hoff	uint8	1 字节	到用户数据的偏移量

所有细节都可以在 src/include/access/htup.h 中找到。

要解释实际数据，必须借助从其他表中获得的信息，其中大部分来自 pg_attribute。所需的特定字段是 attlen 和 attalign。除非所有字段都是定宽且没有空值，否则没有办法直接取得某个特定属性。所有这些技巧都封装在 heap_getattr、fastgetattr 和 heap_getsysattr 这些函数中。

读取数据时，需要依次检查每个属性。首先根据空值位图判断该字段是否为 NULL。如果是，就继续下一个。然后确认对齐是否正确。如果字段是定宽字段，那么它的所有字节都是直接摆放的；如果它是变长字段（attlen == -1），情况就会稍复杂一些，需要使用变长结构 `varattrib`。根据标志位的不同，数据可能是线内的、经过压缩的，或者位于另一个表中（TOAST）。

第 52 章 BKI 后端接口

后端接口（BKI）文件是用一种特殊语言编写的脚本，它们是运行在特殊“引导”模式下的 PostgreSQL 后端的输入，这种模式使后端能够在数据库系统尚不存在的情况下执行数据库功能。因此 BKI 文件可以用于最初创建数据库系统。（它们大概对其他任何事情都没有用处。）

initdb 在创建新的数据库集簇时，使用一个 BKI 文件来完成它的一部分工作。initdb 使用的输入文件是构建和安装 PostgreSQL 的一部分，由一个名为 `genbki.sh` 的程序从源码树中一些采用特殊格式的 C 头文件创建。创建的 BKI 文件名为 `postgres.bki`，通常安装在安装树的 `share` 子目录中。

相关信息可以在 initdb 的文档中找到。

52.1. BKI 文件格式

本节描述 PostgreSQL 后端如何解释 BKI 文件。如果手边有一份 `postgres.bki` 文件作为示例，这段说明会更容易理解。你还应当研究 initdb 的源代码，以了解后端是如何被调用的。

BKI 输入由一系列命令组成。根据命令语法的不同，每条命令由若干词元组成。词元通常由空白分隔，但如果不会产生歧义，也可以不分隔。没有专门的命令分隔符；语法上不可能属于前一条命令的下一个词元，就会开始一条新命令。（通常为了清晰，你会把新命令写在新的一行。）词元可以是某些关键字、特殊字符（圆括号、逗号等）、数字，或者双引号字符串。所有内容都区分大小写。

以 `#` 开头的行会被忽略。

52.2. BKI 命令

open *tablename*

打开名为 *tablename* 的表以便进一步操作。

close [*tablename*]

关闭名为 *tablename* 的已打开表。如果 *tablename* 尚未打开，则这是一个错误。如果没有给出 *tablename*，则关闭当前打开的表。

create *tablename* (*name1* = *type1* [, *name2* = *type2*, ...])

创建一个名为 *tablename* 的表，其列在圆括号中给出。

这里的 *type* 不一定就是该列在 SQL 环境中所具有的数据类型；那是由 `pg_attribute` 系统目录决定的。这里的类型本质上只用于分配存储。允许下列类型：`bool`、`bytea`、`char`（1 字节）、`name`、`int2`、`int2vector`、`int4`、`regproc`、`regclass`、`regtype`、`text`、`oid`、`tid`、`xid`、`cid`、`oidvector`、`smgr`、`_int4`（数组）、`_aclitem`（数组）。数组类型也可以通过在元素类型名之后写 `[]` 来表示。

> **注意**
>
> 该表只会创建在磁盘上，它不会自动注册到系统目录中，因此除非在 `pg_class`、`pg_attribute` 等中插入适当的行，否则它是不可访问的。

insert [OID = *oid_value*] (*value1 value2* ...)

用 *value1*、*value2* 等作为列值，用 *oid_value* 作为其 OID，向已打开的表中插入一个新行。如果 *oid_value* 为零（0）或省略该子句，则使用下一个可用的 OID。

可以使用特殊关键字 `_null_` 来指定 NULL 值。包含空格的值必须加双引号。

declare [unique] index *indexname* on *tablename* using *amname* (*opclass1 name1* [, ...])

用 *amname* 访问方法在名为 *tablename* 的表上创建一个名为 *indexname* 的索引。要索引的字段称为 *name1*、*name2* 等，而要使用的操作符类分别为 *opclass1*、*opclass2* 等。

build indices

构建先前声明的那些索引。

52.3. 示例

下面的命令序列将创建一个名为 `test_table` 的表，它有两列 `cola` 和 `colb`，类型分别为 `int4` 和 `text`，并向表中插入两行。

```
create test_table (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

部分 VIII. 附录

目录

A. PostgreSQL 错误代码 963
B. 日期/时间支持 969
B.1. 日期/时间输入解释 969
B.2. 日期/时间关键字 970
B.3. 历法的历史 974
C. SQL 关键字 976
D. SQL 符合性 991
D.1. 支持的特性 991
D.2. 不支持的特性 1001
E. 发布说明 1006
E.1. 版本 7.4.30 1006
E.1.1. 迁移到版本 7.4.30 1006
E.1.2. 变更 1006
E.2. 版本 7.4.29 1007
E.2.1. 迁移到版本 7.4.29 1007
E.2.2. 变更 1007
E.3. 版本 7.4.28 1008
E.3.1. 迁移到版本 7.4.28 1008
E.3.2. 变更 1008
E.4. 版本 7.4.27 1009
E.4.1. 迁移到版本 7.4.27 1009
E.4.2. 变更 1009
E.5. 版本 7.4.26 1010
E.5.1. 迁移到版本 7.4.26 1010
E.5.2. 变更 1010
E.6. 版本 7.4.25 1011
E.6.1. 迁移到版本 7.4.25 1011
E.6.2. 变更 1011
E.7. 版本 7.4.24 1011
E.7.1. 迁移到版本 7.4.24 1011
E.7.2. 变更 1012
E.8. 版本 7.4.23 1012
E.8.1. 迁移到版本 7.4.23 1012
E.8.2. 变更 1012
E.9. 版本 7.4.22 1013
E.9.1. 迁移到版本 7.4.22 1013
E.9.2. 变更 1013
E.10. 版本 7.4.21 1013
E.10.1. 迁移到版本 7.4.21 1013
E.10.2. 变更 1013
E.11. 版本 7.4.20 1014
E.11.1. 迁移到版本 7.4.20 1014
E.11.2. 变更 1014
E.12. 版本 7.4.19 1015
E.12.1. 迁移到版本 7.4.19 1015
E.12.2. 变更 1015
E.13. 版本 7.4.18 1016
E.13.1. 迁移到版本 7.4.18 1016
E.13.2. 变更 1016
E.14. 版本 7.4.17 1016
E.14.1. 迁移到版本 7.4.17 1017

E.14.2. 变更 1017
E.15. 版本 7.4.16 1017
E.15.1. 迁移到版本 7.4.16 1017
E.15.2. 变更 1017
E.16. 版本 7.4.15 1018
E.16.1. 迁移到版本 7.4.15 1018
E.16.2. 变更 1018
E.17. 版本 7.4.14 1018
E.17.1. 迁移到版本 7.4.14 1018
E.17.2. 变更 1018
E.18. 版本 7.4.13 1019
E.18.1. 迁移到版本 7.4.13 1019
E.18.2. 变更 1019
E.19. 版本 7.4.12 1020
E.19.1. 迁移到版本 7.4.12 1020
E.19.2. 变更 1020
E.20. 版本 7.4.11 1021
E.20.1. 迁移到版本 7.4.11 1021
E.20.2. 变更 1021
E.21. 版本 7.4.10 1021
E.21.1. 迁移到版本 7.4.10 1021
E.21.2. 变更 1022
E.22. 版本 7.4.9 1022
E.22.1. 迁移到版本 7.4.9 1022
E.22.2. 变更 1022
E.23. 版本 7.4.8 1023
E.23.1. 迁移到版本 7.4.8 1023
E.23.2. 变更 1024
E.24. 版本 7.4.7 1025
E.24.1. 迁移到版本 7.4.7 1025
E.24.2. 变更 1025
E.25. 版本 7.4.6 1026
E.25.1. 迁移到版本 7.4.6 1026
E.25.2. 变更 1026
E.26. 版本 7.4.5 1027
E.26.1. 迁移到版本 7.4.5 1027
E.26.2. 变更 1027
E.27. 版本 7.4.4 1027
E.27.1. 迁移到版本 7.4.4 1028
E.27.2. 变更 1028
E.28. 版本 7.4.3 1028
E.28.1. 迁移到版本 7.4.3 1028
E.28.2. 变更 1028
E.29. 版本 7.4.2 1029
E.29.1. 迁移到版本 7.4.2 1029
E.29.2. 变更 1030
E.30. 版本 7.4.1 1031
E.30.1. 迁移到版本 7.4.1 1031
E.30.2. 变更 1032
E.31. 版本 7.4 1033
E.31.1. 概述 1033
E.31.2. 迁移到版本 7.4 1034
E.31.3. 变更 1035
E.32. 版本 7.3.21 1048

E.32.1. 迁移到版本 7.3.21 1048
E.32.2. 变更 1048
E.33. 版本 7.3.20 1049
E.33.1. 迁移到版本 7.3.20 1049
E.33.2. 变更 1049
E.34. 版本 7.3.19 1049
E.34.1. 迁移到版本 7.3.19 1049
E.34.2. 变更 1050
E.35. 版本 7.3.18 1050
E.35.1. 迁移到版本 7.3.18 1050
E.35.2. 变更 1050
E.36. 版本 7.3.17 1050
E.36.1. 迁移到版本 7.3.17 1050
E.36.2. 变更 1051
E.37. 版本 7.3.16 1051
E.37.1. 迁移到版本 7.3.16 1051
E.37.2. 变更 1051
E.38. 版本 7.3.15 1051
E.38.1. 迁移到版本 7.3.15 1051
E.38.2. 变更 1052
E.39. 版本 7.3.14 1052
E.39.1. 迁移到版本 7.3.14 1052
E.39.2. 变更 1053
E.40. 版本 7.3.13 1053
E.40.1. 迁移到版本 7.3.13 1053
E.40.2. 变更 1053
E.41. 版本 7.3.12 1054
E.41.1. 迁移到版本 7.3.12 1054
E.41.2. 变更 1054
E.42. 版本 7.3.11 1054
E.42.1. 迁移到版本 7.3.11 1054
E.42.2. 变更 1054
E.43. 版本 7.3.10 1055
E.43.1. 迁移到版本 7.3.10 1055
E.43.2. 变更 1056
E.44. 版本 7.3.9 1056
E.44.1. 迁移到版本 7.3.9 1057
E.44.2. 变更 1057
E.45. 版本 7.3.8 1057
E.45.1. 迁移到版本 7.3.8 1057
E.45.2. 变更 1057
E.46. 版本 7.3.7 1058
E.46.1. 迁移到版本 7.3.7 1058
E.46.2. 变更 1058
E.47. 版本 7.3.6 1058
E.47.1. 迁移到版本 7.3.6 1058
E.47.2. 变更 1058
E.48. 版本 7.3.5 1059
E.48.1. 迁移到版本 7.3.5 1059
E.48.2. 变更 1059
E.49. 版本 7.3.4 1060
E.49.1. 迁移到版本 7.3.4 1060
E.49.2. 变更 1060
E.50. 版本 7.3.3 1061

E.50.1. 迁移到版本 7.3.3 1061
E.50.2. 变更 1061
E.51. 版本 7.3.2 1063
E.51.1. 迁移到版本 7.3.2 1064
E.51.2. 变更 1064
E.52. 版本 7.3.1 1065
E.52.1. 迁移到版本 7.3.1 1065
E.52.2. 变更 1065
E.53. 版本 7.3 1066
E.53.1. 概述 1066
E.53.2. 迁移到版本 7.3 1067
E.53.3. 变更 1067
E.54. 版本 7.2.8 1078
E.54.1. 迁移到版本 7.2.8 1078
E.54.2. 变更 1078
E.55. 版本 7.2.7 1078
E.55.1. 迁移到版本 7.2.7 1079
E.55.2. 变更 1079
E.56. 版本 7.2.6 1079
E.56.1. 迁移到版本 7.2.6 1079
E.56.2. 变更 1079
E.57. 版本 7.2.5 1080
E.57.1. 迁移到版本 7.2.5 1080
E.57.2. 变更 1080
E.58. 版本 7.2.4 1080
E.58.1. 迁移到版本 7.2.4 1080
E.58.2. 变更 1080
E.59. 版本 7.2.3 1081
E.59.1. 迁移到版本 7.2.3 1081
E.59.2. 变更 1081
E.60. 版本 7.2.2 1081
E.60.1. 迁移到版本 7.2.2 1081
E.60.2. 变更 1082
E.61. 版本 7.2.1 1082
E.61.1. 迁移到版本 7.2.1 1082
E.61.2. 变更 1082
E.62. 版本 7.2 1083
E.62.1. 概述 1083
E.62.2. 迁移到版本 7.2 1084
E.62.3. 变更 1084
E.63. 版本 7.1.3 1093
E.63.1. 迁移到版本 7.1.3 1093
E.63.2. 变更 1093
E.64. 版本 7.1.2 1094
E.64.1. 迁移到版本 7.1.2 1094
E.64.2. 变更 1094
E.65. 版本 7.1.1 1094
E.65.1. 迁移到版本 7.1.1 1094
E.65.2. 变更 1094
E.66. 版本 7.1 1095
E.66.1. 迁移到版本 7.1 1095
E.66.2. 变更 1095
E.67. 版本 7.0.3 1099
E.67.1. 迁移到版本 7.0.3 1099

E.67.2. 变更 1099
E.68. 版本 7.0.2 1100
E.68.1. 迁移到版本 7.0.2 1100
E.68.2. 变更 1100
E.69. 版本 7.0.1 1100
E.69.1. 迁移到版本 7.0.1 1101
E.69.2. 变更 1101
E.70. 版本 7.0 1101
E.70.1. 迁移到版本 7.0 1102
E.70.2. 变更 1102
E.71. 版本 6.5.3 1108
E.71.1. 迁移到版本 6.5.3 1108
E.71.2. 变更 1108
E.72. 版本 6.5.2 1109
E.72.1. 迁移到版本 6.5.2 1109
E.72.2. 变更 1109
E.73. 版本 6.5.1 1109
E.73.1. 迁移到版本 6.5.1 1110
E.73.2. 变更 1110
E.74. 版本 6.5 1110
E.74.1. 迁移到版本 6.5 1111
E.74.2. 变更 1112
E.75. 版本 6.4.2 1115
E.75.1. 迁移到版本 6.4.2 1115
E.75.2. 变更 1115
E.76. 版本 6.4.1 1115
E.76.1. 迁移到版本 6.4.1 1115
E.76.2. 变更 1115
E.77. 版本 6.4 1116
E.77.1. 迁移到版本 6.4 1117
E.77.2. 变更 1117
E.78. 版本 6.3.2 1120
E.78.1. 变更 1121
E.79. 版本 6.3.1 1121
E.79.1. 变更 1121
E.80. 版本 6.3 1122
E.80.1. 迁移到版本 6.3 1123
E.80.2. 变更 1123
E.81. 版本 6.2.1 1126
E.81.1. 从版本 6.2 迁移到版本 6.2.1 1127
E.81.2. 变更 1127
E.82. 版本 6.2 1127
E.82.1. 从版本 6.1 迁移到版本 6.2 1127
E.82.2. 从版本 1.x 迁移到版本 6.2 1128
E.82.3. 变更 1128
E.83. 版本 6.1.1 1130
E.83.1. 从版本 6.1 迁移到版本 6.1.1 1130
E.83.2. 变更 1130
E.84. 版本 6.1 1130
E.84.1. 迁移到版本 6.1 1131
E.84.2. 变更 1131
E.85. 版本 6.0 1133
E.85.1. 从版本 1.09 迁移到版本 6.0 1133

E.85.2. 从 1.09 之前版本迁移到版本 6.0 1133
E.85.3. 变更 1133
E.86. 版本 1.09 1135
E.87. 版本 1.02 1135
E.87.1. 从版本 1.02 迁移到版本 1.02.1 1135
E.87.2. 转储/重载程序 1136
E.87.3. 变更 1136
E.88. 版本 1.01 1137
E.88.1. 从版本 1.0 迁移到版本 1.01 1137
E.88.2. 变更 1139
E.89. 版本 1.0 1139
E.89.1. 变更 1139
E.90. Postgres95 Release 0.03 1140
E.90.1. 变更 1140
E.91. Postgres95 Release 0.02 1142
E.91.1. 变更 1142
E.92. Postgres95 Release 0.01 1142
F. 源代码仓库 1143
F.1. 通过 Git 获取源代码 1143
G. 文档 1144
G.1. DocBook 1144
G.2. 工具集 1144
G.2.1. Linux RPM 安装 1145
G.2.2. FreeBSD 安装 1145
G.2.3. Debian 软件包 1146
G.2.4. 从源码手动安装 1146
G.2.5. 被 `configure` 检测 1148
G.3. 构建文档 1148
G.3.1. HTML 1148
G.3.2. 手册页 1148
G.3.3. 通过 JadeTex 生成打印输出 1149
G.3.4. 通过 RTF 生成打印输出 1149
G.3.5. 纯文本文件 1150
G.3.6. 语法检查 1150
G.4. 文档编写 1151
G.4.1. Emacs/PSGML 1151
G.4.2. 其他 Emacs 模式 1152
G.5. 风格指南 1152
G.5.1. 参考页 1152

附录 A. PostgreSQL 错误代码

PostgreSQL 服务器发出的所有消息都被分配了五字符错误代码，遵循 SQL 标准对“SQLSTATE”代码的约定。需要知道发生了哪一种错误条件的应用，通常应测试错误代码，而不是查看错误消息的文本内容。这些错误代码在不同 PostgreSQL 版本之间更不容易发生变化，也不会因错误消息的本地化而改变。

根据标准，错误代码的前两个字符表示错误的类别，后三个字符表示该类别中的特定条件。因此，即使应用无法识别某个具体错误代码，仍可能从错误类别推断出应采取的操作。

表 A.1列出了 PostgreSQL 7.4.30 中定义的所有错误代码。（其中有些当前实际上并未使用，不过它们由 SQL 标准定义。）错误类别也一并显示。对于每个错误类别，都有一个“标准”错误代码，其最后三个字符为`000`。该代码只用于那些属于该类别但没有分配更具体代码的错误情况。

表 A.1. PostgreSQL Error Codes

Error Code	Meaning
Class 00	成功完成
`00000`	SUCCESSFUL COMPLETION
Class 01	警告
`01000`	WARNING
`0100C`	WARNING DYNAMIC RESULT SETS RETURNED
`01008`	WARNING IMPLICIT ZERO BIT PADDING
`01003`	WARNING NULL VALUE ELIMINATED IN SET FUNCTION
`01004`	WARNING STRING DATA RIGHT TRUNCATION
Class 02	无数据（按 SQL99 这也是一个警告类别）
`02000`	NO DATA
`02001`	NO ADDITIONAL DYNAMIC RESULT SETS RETURN ED
Class 03	SQL 语句尚未完成
`03000`	SQL STATEMENT NOT YET COMPLETE
Class 08	连接异常
`08000`	CONNECTION EXCEPTION
`08003`	CONNECTION DOES NOT EXIST
`08006`	CONNECTION FAILURE
`08001`	SQLCLIENT UNABLE TO ESTABLISH SQLCON NECTION
`08004`	SQLSERVER REJECTED ESTABLISHMENT OF SQLCONNECTION
`08007`	TRANSACTION RESOLUTION UNKNOWN
`08P01`	PROTOCOL VIOLATION
Class 09	触发动作异常
`09000`	TRIGGERED ACTION EXCEPTION
Class 0A	不支持的特性
`0A000`	FEATURE NOT SUPPORTED

Error Code	Meaning
Class 0B	无效的事务启动
0B000	INVALID TRANSACTION INITIATION
Class 0F	定位器异常
0F000	LOCATOR EXCEPTION
0F001	INVALID SPECIFICATION
Class 0L	无效的授权者
0L000	INVALID GRANTOR
0LP01	INVALID GRANT OPERATION
Class 0P	无效的角色规范
0P000	INVALID ROLE SPECIFICATION
Class 21	基数违反
21000	CARDINALITY VIOLATION
Class 22	数据异常
22000	DATA EXCEPTION
2202E	ARRAY ELEMENT ERROR
22021	CHARACTER NOT IN REPERTOIRE
22008	DATETIME FIELD OVERFLOW
22012	DIVISION BY ZERO
22005	ERROR IN ASSIGNMENT
2200B	ESCAPE CHARACTER CONFLICT
22022	INDICATOR OVERFLOW
22015	INTERVAL FIELD OVERFLOW
22018	INVALID CHARACTER VALUE FOR CAST
22007	INVALID DATETIME FORMAT
22019	INVALID ESCAPE CHARACTER
2200D	INVALID ESCAPE OCTET
22025	INVALID ESCAPE SEQUENCE
22010	INVALID INDICATOR PARAMETER VALUE
22020	INVALID LIMIT VALUE
22023	INVALID PARAMETER VALUE
2201B	INVALID REGULAR EXPRESSION
22009	INVALID TIME ZONE DISPLACEMENT VALUE
2200C	INVALID USE OF ESCAPE CHARACTER
2200G	MOST SPECIFIC TYPE MISMATCH
22004	NULL VALUE NOT ALLOWED
22002	NULL VALUE NO INDICATOR PARAMETER
22003	NUMERIC VALUE OUT OF RANGE
22026	STRING DATA LENGTH MISMATCH
22001	STRING DATA RIGHT TRUNCATION
22011	SUBSTRING ERROR

Error Code	Meaning
22027	TRIM ERROR
22024	UNTERMINATED C STRING
2200F	ZERO LENGTH CHARACTER STRING
22P01	FLOATING POINT EXCEPTION
22P02	INVALID TEXT REPRESENTATION
22P03	INVALID BINARY REPRESENTATION
22P04	BAD COPY FILE FORMAT
22P05	UNTRANSLATABLE CHARACTER
Class 23	完整性约束违反
23000	INTEGRITY CONSTRAINT VIOLATION
23001	RESTRICT VIOLATION
23502	NOT NULL VIOLATION
23503	FOREIGN KEY VIOLATION
23505	UNIQUE VIOLATION
23514	CHECK VIOLATION
Class 24	无效的游标状态
24000	INVALID CURSOR STATE
Class 25	无效的事务状态
25000	INVALID TRANSACTION STATE
25001	ACTIVE SQL TRANSACTION
25002	BRANCH TRANSACTION ALREADY ACTIVE
25008	HELD CURSOR REQUIRES SAME ISOLATION LEVEL
25003	INAPPROPRIATE ACCESS MODE FOR BRANCH TRANSACTION
25004	INAPPROPRIATE ISOLATION LEVEL FOR BRANCH TRANSACTION
25005	NO ACTIVE SQL TRANSACTION FOR BRANCH TRANSACTION
25006	READ ONLY SQL TRANSACTION
25007	SCHEMA AND DATA STATEMENT MIXING NOT SUPPORTED
25P01	NO ACTIVE SQL TRANSACTION
25P02	IN FAILED SQL TRANSACTION
Class 26	无效的 SQL 语句名称
26000	INVALID SQL STATEMENT NAME
Class 27	触发的数据更改违反
27000	TRIGGERED DATA CHANGE VIOLATION
Class 28	无效的授权规范
28000	INVALID AUTHORIZATION SPECIFICATION
Class 2B	仍存在依赖的权限描述符
2B000	DEPENDENT PRIVILEGE DESCRIPTORS STILL EXIST

Error Code	Meaning
2BP01	DEPENDENT OBJECTS STILL EXIST
Class 2D	无效的事务终止
2D000	INVALID TRANSACTION TERMINATION
Class 2F	SQL 例程异常
2F000	SQL ROUTINE EXCEPTION
2F005	FUNCTION EXECUTED NO RETURN STATEMENT
2F002	MODIFYING SQL DATA NOT PERMITTED
2F003	PROHIBITED SQL STATEMENT ATTEMPTED
2F004	READING SQL DATA NOT PERMITTED
Class 34	无效的游标名称
34000	INVALID CURSOR NAME
Class 38	外部例程异常
38000	EXTERNAL ROUTINE EXCEPTION
38001	CONTAINING SQL NOT PERMITTED
38002	MODIFYING SQL DATA NOT PERMITTED
38003	PROHIBITED SQL STATEMENT ATTEMPTED
38004	READING SQL DATA NOT PERMITTED
Class 39	外部例程调用异常
39000	EXTERNAL ROUTINE INVOCATION EXCEPTION
39001	INVALID SQLSTATE RETURNED
39004	NULL VALUE NOT ALLOWED
39P01	TRIGGER PROTOCOL VIOLATED
39P02	SRF PROTOCOL VIOLATED
Class 3D	无效的目录名称
3D000	INVALID CATALOG NAME
Class 3F	无效的模式名称
3F000	INVALID SCHEMA NAME
Class 40	事务回卷
40000	TRANSACTION ROLLBACK
40002	INTEGRITY CONSTRAINT VIOLATION
40001	SERIALIZATION FAILURE
40003	STATEMENT COMPLETION UNKNOWN
40P01	DEADLOCK DETECTED
Class 42	语法错误或访问规则违反
42000	SYNTAX ERROR OR ACCESS RULE VIOLATION
42601	SYNTAX ERROR
42501	INSUFFICIENT PRIVILEGE
42846	CANNOT COERCE
42803	GROUPING ERROR
42830	INVALID FOREIGN KEY

Error Code	Meaning
42602	INVALID NAME
42622	NAME TOO LONG
42939	RESERVED NAME
42804	DATATYPE MISMATCH
42P18	INDETERMINATE DATATYPE
42809	WRONG OBJECT TYPE
42703	UNDEFINED COLUMN
42883	UNDEFINED FUNCTION
42P01	UNDEFINED TABLE
42P02	UNDEFINED PARAMETER
42704	UNDEFINED OBJECT
42701	DUPLICATE COLUMN
42P03	DUPLICATE CURSOR
42P04	DUPLICATE DATABASE
42723	DUPLICATE FUNCTION
42P05	DUPLICATE PSTATEMENT
42P06	DUPLICATE SCHEMA
42P07	DUPLICATE TABLE
42712	DUPLICATE ALIAS
42710	DUPLICATE OBJECT
42702	AMBIGUOUS COLUMN
42725	AMBIGUOUS FUNCTION
42P08	AMBIGUOUS PARAMETER
42P09	AMBIGUOUS ALIAS
42P10	INVALID COLUMN REFERENCE
42611	INVALID COLUMN DEFINITION
42P11	INVALID CURSOR DEFINITION
42P12	INVALID DATABASE DEFINITION
42P13	INVALID FUNCTION DEFINITION
42P14	INVALID PSTATEMENT DEFINITION
42P15	INVALID SCHEMA DEFINITION
42P16	INVALID TABLE DEFINITION
42P17	INVALID OBJECT DEFINITION
Class 44	违反 WITH CHECK OPTION
44000	WITH CHECK OPTION VIOLATION
Class 53	资源不足
53000	INSUFFICIENT RESOURCES
53100	DISK FULL
53200	OUT OF MEMORY
53300	TOO MANY CONNECTIONS

Error Code	Meaning
Class 54	超出程序限制
54000	PROGRAM LIMIT EXCEEDED
54001	STATEMENT TOO COMPLEX
54011	TOO MANY COLUMNS
54023	TOO MANY ARGUMENTS
Class 55	对象未处于先决状态
55000	OBJECT NOT IN PREREQUISITE STATE
55006	OBJECT IN USE
55P02	CANT CHANGE RUNTIME PARAM
Class 57	操作员干预
57000	OPERATOR INTERVENTION
57014	QUERY CANCELED
57P01	ADMIN SHUTDOWN
57P02	CRASH SHUTDOWN
57P03	CANNOT CONNECT NOW
Class 58	系统错误（PostgreSQL 本身之外的错误）
58030	IO ERROR
58P01	UNDEFINED FILE
58P02	DUPLICATE FILE
Class F0	配置文件错误
F0000	CONFIG FILE ERROR
F0001	LOCK FILE EXISTS
Class XX	内部错误
XX000	INTERNAL ERROR
XX001	DATA CORRUPTED
XX002	INDEX CORRUPTED

附录 B. 日期/时间支持

PostgreSQL 对所有日期/时间输入都使用一个内部的启发式解析器。日期和时间以字符串形式输入，并被拆分为不同的字段，同时初步判断每个字段中可能包含哪类信息。
每个字段都会被解释，并被赋予数值、忽略或拒绝。
解析器内部为所有文本字段都维护了查找表，包括月份、星期几以及时区。

本附录包含关于这些查找表内容的信息，并描述了解析器解码日期和时间时所使用的步骤。

B.1. 日期/时间输入解释

日期/时间输入字符串按照下面的过程进行解码。

1. 将输入字符串拆分为词元，并把每个词元归类为字符串、时间、时区或数字。

   a. 如果数字词元包含冒号（`:`），则它是一个时间字符串。
      把其后的所有数字和冒号都包含进来。

   b. 如果数字词元包含连字符（`-`）、斜杠（`/`）或两个以上的点（`.`），
      则它是一个日期字符串，并且可能带有文本形式的月份。

   c. 如果词元仅由数字组成，那么它要么是单个字段，要么是 ISO 8601 的拼接式日期（例如，`19990113` 表示 1999 年 1 月 13 日）或时间（例如，`141516` 表示 14:15:16）。

   d. 如果词元以加号（`+`）或减号（`-`）开头，那么它要么是数字时区，要么是特殊字段。

2. 如果词元是一个纯字母字符串，则将其与可能的字符串匹配：

   a. 对该词元做二分查找表匹配，看它是否为特殊字符串（例如 `today`）、星期名（例如 `Thursday`）、月份名（例如 `January`）或噪声词（例如 `at`、`on`）。

      设置字段值和字段位掩码。例如，为 `today` 设置年、月、日，并为 `now` 额外设置时、分、秒。

   b. 如果没有找到，则做一次类似的二分查找表匹配，尝试将该词元与时区匹配。

   c. 如果仍然没有找到，则抛出错误。

3. 当词元是一个数字或数字字段时：

   a. 如果有八位或六位数字，并且之前还没有读取到其他日期字段，
      则将其解释为“拼接式日期”（例如 `19990118` 或 `990118`）。其解释方式分别为 `YYYYMMDD` 或 `YYMMDD`。

   b. 如果词元是三位数字，并且已经读取到了年份，则将其解释为一年中的第几天。

   c. 如果是四位或六位数字，并且已经读取到了年份，则将其解释为时间（`HHMM` 或 `HHMMSS`）。

   d. 如果有三位或更多位数字，并且尚未找到任何日期字段，
      则将其解释为年份（这会强制其余日期字段按 yy-mm-dd 的顺序解释）。

   e. 否则，假定日期字段顺序遵循 `DateStyle` 设置：mm-dd-yy、dd-mm-yy 或 yy-mm-dd。如果发现月份字段或日字段超出范围，则抛出错误。

4. 如果指定了 BC，则将年份取反并加一后再用于内部存储。（格里高利历中没有 0 年，因此从数值上说，公元前 1 年会变成年 0。）
5. 如果没有指定 BC，并且年份字段的长度为两位，则将年份调整为四位。如果该字段小于 70，则加上 2000，否则加上 1900。

> **提示**
>
> 格里高利历公元 1-99 年可以使用带前导零的四位数字输入（例如，`0099` 表示公元 99 年）。以前的 PostgreSQL 版本还接受三位数字和单个数字的年份，但从版本 7.0 开始，这些规则被收紧，以减少出现歧义的可能性。

B.2. 日期/时间关键字

表 B.1展示了哪些词元会被识别为月份名称。

表 B.1. 月份缩写

月份	缩写
January	Jan
February	Feb
March	Mar
April	Apr
June	Jun
July	Jul
August	Aug
September	Sep, Sept
October	Oct
November	Nov
December	Dec

> **注意**
>
> 出于显而易见的原因，五月（May）没有缩写。

表 B.2展示了哪些词元会被识别为星期名称。

表 B.2. 星期缩写

星期	缩写
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri

星期	缩写
Saturday	Sat

表 B.3展示了用于各种修饰用途的词元。

表 B.3. 日期/时间字段修饰符

标识符	描述
ABSTIME	忽略的关键字
AM	时间在 12:00 之前
AT	忽略的关键字
JULIAN, JD, J	下一个字段是儒略日
ON	忽略的关键字
PM	时间为 12:00 或之后
T	下一个字段是时间

关键字 ABSTIME 因历史原因而被忽略：在 PostgreSQL 非常早期的版本中，abstime 类型的无效值会被输出为 Invalid Abstime。不过现在情况已经不同，该关键字很可能在未来的版本中被移除。

表 B.4展示了 PostgreSQL 在日期/时间输入值中识别的时区缩写。PostgreSQL 使用内部表进行时区输入解码，因为没有标准的操作系统接口可用于访问通用的跨时区信息。不过，底层操作系统确实被用来为输出提供时区信息。

本表按与 UTC 的时区偏移组织，而不是按字母顺序。这样做的目的是便于在本地用法与已识别缩写可能不一致的情况下，将两者匹配起来。

表 B.4. 时区缩写

时区	与 UTC 的偏移	描述
NZDT	+13:00	新西兰夏令时间
IDLE	+12:00	国际日期变更线以东
NZST	+12:00	新西兰标准时间
NZT	+12:00	新西兰时间
AESST	+11:00	澳大利亚东部夏季标准时间
ACSST	+10:30	中澳夏季标准时间
CADT	+10:30	中澳夏令时间
SADT	+10:30	南澳夏令时间
AEST	+10:00	澳大利亚东部标准时间
EAST	+10:00	澳东标准时间
GST	+10:00	关岛标准时间，俄罗斯 9 区
LIGT	+10:00	澳大利亚墨尔本
SAST	+09:30	南澳标准时间
CAST	+09:30	中澳标准时间
AWSST	+09:00	澳大利亚西部夏季标准时间
JST	+09:00	日本标准时间，俄罗斯 8 区
KST	+09:00	韩国标准时间

时区	与 UTC 的偏移	描述
MHT	+09:00	夸贾林时间
WDT	+09:00	西澳夏令时间
MT	+08:30	摩鹿加时间
AWST	+08:00	澳大利亚西部标准时间
CCT	+08:00	中国沿海时间
WADT	+08:00	西澳夏令时间
WST	+08:00	西澳标准时间
JT	+07:30	爪哇时间
ALMST	+07:00	阿拉木图夏令时间
WAST	+07:00	西澳标准时间
CXT	+07:00	圣诞岛时间
MMT	+06:30	缅甸时间
ALMT	+06:00	阿拉木图时间
MAWT	+06:00	莫森（南极洲）时间
IOT	+05:00	印度查戈斯时间
MVT	+05:00	马尔代夫群岛时间
TFT	+05:00	凯尔盖朗时间
AFT	+04:30	阿富汗时间
EAST	+04:00	安塔那那利佛夏令时间
MUT	+04:00	毛里求斯岛时间
RET	+04:00	留尼汪岛时间
SCT	+04:00	马埃岛时间
IRT, IT	+03:30	伊朗时间
EAT	+03:00	安塔那那利佛、科摩罗时间
BT	+03:00	巴格达时间
EETDST	+03:00	东欧夏令时间
HMT	+03:00	希腊地中海时间（?）
BDST	+02:00	英国双标准时间
CEST	+02:00	中欧夏令时间
CETDST	+02:00	中欧夏令时间
EET	+02:00	东欧时间，俄罗斯 1 区
FWT	+02:00	法国冬令时间
IST	+02:00	以色列标准时间
MEST	+02:00	中欧夏令时间
METDST	+02:00	中欧夏令时间
SST	+02:00	瑞典夏令时间
BST	+01:00	英国夏令时间
CET	+01:00	中欧时间
DNT	+01:00	Dansk Normal Tid
FST	+01:00	法国夏令时间
MET	+01:00	中欧时间

时区	与 UTC 的偏移	描述
MEWT	+01:00	中欧冬令时间
MEZ	+01:00	Mitteleuropäische Zeit
NOR	+01:00	挪威标准时间
SET	+01:00	塞舌尔时间
SWT	+01:00	瑞典冬令时间
WETDST	+01:00	西欧夏令时间
GMT	00:00	格林尼治标准时间
UT	00:00	世界时
UTC	00:00	协调世界时
Z	00:00	与 UTC 相同
ZULU	00:00	与 UTC 相同
WET	00:00	西欧时间
WAT	-01:00	西非时间
FNST	-01:00	费尔南多-迪诺罗尼亚夏令时间
FNT	-02:00	费尔南多-迪诺罗尼亚时间
BRST	-02:00	巴西利亚夏令时间
NDT	-02:30	纽芬兰夏令时间
ADT	-03:00	大西洋夏令时间
AWT	-03:00	（未知）
BRT	-03:00	巴西利亚时间
NFT	-03:30	纽芬兰标准时间
NST	-03:30	纽芬兰标准时间
AST	-04:00	大西洋标准时间（加拿大）
ACST	-04:00	Atlantic/Porto Acre 夏令时间
EDT	-04:00	美国东部夏令时间
ACT	-05:00	Atlantic/Porto Acre 标准时间
CDT	-05:00	美国中部夏令时间
EST	-05:00	美国东部标准时间
CST	-06:00	美国中部标准时间
MDT	-06:00	美国山地夏令时间
MST	-07:00	美国山地标准时间
PDT	-07:00	美国太平洋夏令时间
AKDT	-08:00	阿拉斯加夏令时间
PST	-08:00	美国太平洋标准时间
YDT	-08:00	育空夏令时间
AKST	-09:00	阿拉斯加标准时间
HDT	-09:00	夏威夷/阿拉斯加夏令时间
YST	-09:00	育空标准时间
MART	-09:30	马克萨斯群岛时间
AHST	-10:00	阿拉斯加/夏威夷标准时间
HST	-10:00	夏威夷标准时间

时区	与 UTC 的偏移	描述
CAT	-10:00	阿拉斯加中部时间
NT	-11:00	诺姆时间
IDLW	-12:00	国际日期变更线以西

还要记住，`SET TIMEZONE` 可识别的时区名依赖于操作系统，并且可能与表 B.4关系不大。例如，某些系统在 `SET TIMEZONE` 中识别 `'Europe/Rome'` 这样的值。

澳大利亚时区. 澳大利亚时区名与北美及南美常用时区名之间存在三处命名冲突：`ACST`、`CST` 和 `EST`。如果运行时选项 `australian_timezones` 被设置为 true，那么 `ACST`、`CST`、`EST` 和 `SAT` 将被解释为澳大利亚时区名，如表 B.5所示。如果它为 false（这是默认值），那么 `ACST`、`CST` 和 `EST` 会被当作美国时区名，而 `SAT` 会被解释为一个表示星期六的噪声词。

表 B.5. 澳大利亚时区缩写

时区	与 UTC 的偏移	描述
ACST	+09:30	中澳标准时间
CST	+10:30	澳大利亚中部标准时间
EST	+10:00	澳大利亚东部标准时间
SAT	+09:30	南澳标准时间

B.3. 历法的历史

儒略日由法国学者 Joseph Justus Scaliger（1540-1609）发明，其名称大概取自 Scaliger 的父亲、意大利学者 Julius Caesar Scaliger（1484-1558）。天文学家用儒略纪为自公元前 4713 年 1 月 1 日以来的每一天分配一个唯一的编号。这就是所谓的儒略日（JD）。JD 0 表示从 UTC 公元前 4713 年 1 月 1 日正午到 UTC 公元前 4713 年 1 月 2 日正午的 24 小时。

“儒略日”与“儒略历”并不相同。儒略历由 Julius Caesar 于公元前 45 年引入。在 1582 年各国开始改用格里高利历之前，它在西方世界一直被广泛使用。在儒略历中，回归年近似为 365 1/4 天 = 365.25 天。这会在大约 128 年里累计 1 天的误差。

不断累积的历法误差促使教皇格里高利十三世按照特伦托会议的指示改革历法。
在格里高利历中，回归年近似为 365 + 97 / 400 天 = 365.2425 天。因此，大约要经过 3300 年，回归年与格里高利历之间才会累积出 1 天的偏差。

365+97/400 这一近似值是通过每 400 年设置 97 个闰年来实现的，具体规则如下：

每个能被 4 整除的年份都是闰年。
不过，每个能被 100 整除的年份都不是闰年。
但是，每个能被 400 整除的年份仍然是闰年。

因此，1700、1800、1900、2100 和 2200 都不是闰年，但 1600、2000 和 2400 是闰年。相比之下，较早的儒略历规定所有能被 4 整除的年份都是闰年。

1582 年 2 月发布的教皇诏书规定，应从 1582 年 10 月删去 10 天，使 10 月 15 日紧接在 10 月 4 日之后。这一改革在意大利、波兰、葡萄牙和西班牙得到执行。其他天主教国家不久后也跟进，但新教国家不愿改变，而希腊正教国家直到 20 世纪初才改用格里高利历。大不列颠及其属地（包括今天的美国）在 1752 年实施了这项改革。因此，1752 年 9 月 2 日之后紧接着就是 1752 年 9 月 14 日。这就是为什么提供 `cal` 程序的 Unix 系统会产生如下输出：

```
$ cal 9 1752
```

```
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

> **注意**
>
> SQL 标准指出，“在‘日期时间字面量’的定义中，‘日期时间值’受格里高利历中日期和时间自然规则的约束”。1752-09-03 到 1752-09-13 之间的日期，虽然在某些国家因教皇敕令而被删除，但它们符合“自然规则”，因此是有效日期。

世界上不同地区发展出了不同的历法，其中许多都早于格里高利体系。例如，
中国历法的起源可以追溯到公元前 14 世纪。传说黄帝在公元前 2637 年发明了这种历法。中华人民共和国在民用方面使用格里高利历。中国历法则用于确定节日。

附录 C. SQL 关键字

表 C.1列出了在 SQL 标准以及 PostgreSQL 7.4.30 中被视为关键字的所有词元。背景信息可参见第 4.1.1 节。

SQL 区分保留关键字和非保留关键字。根据标准，保留关键字才是真正的关键字，它们绝不允许作为标识符使用。非保留关键字只在特定上下文中具有特殊含义，
在其他上下文中则可以作为标识符使用。大多数非保留关键字实际上是 SQL 规定的内置表和内置函数名称。非保留关键字这一概念，
本质上只是为了声明某个词在某些上下文中附带了预定义含义。

在PostgreSQL解析器中，情况要更复杂一些。这里的词元分属若干不同类别，
从绝不能用作标识符的词元，到在解析器中完全没有特殊地位、
只是被当作普通标识符处理的词元都有。（后一类通常就是 SQL 规定的函数名。）即使是保留关键字，在PostgreSQL中也不是绝对保留的，它们仍然可以用作列标签（例如`SELECT 55 AS CHECK`，尽管`CHECK`是一个保留关键字）。

在表 C.1中PostgreSQL 一列里，我们把解析器明确识别、但在大多数或所有期望出现标识符的上下文中都允许使用的那些关键字归类为“非保留”。
某些原本属于非保留的关键字不能用作函数名或数据类型名，因此会相应地加以标记。
（这类词大多表示具有特殊语法的内置函数或数据类型。函数或类型本身仍然可用，
只是不能由用户重新定义。）标记为“保留”的是那些只允许用作“AS”
列标签名（也许还有极少数其他上下文）的词元。有些保留关键字还允许用作函数名；
这一点也会在表中给出。

一般来说，如果你在某条命令中把这里列出的关键字当作标识符使用，
并因此遇到意外的解析器错误，应尝试为该标识符加上引号，看看问题是否消失。

在研读表 C.1之前，有一点非常重要：某个关键字在PostgreSQL中不被保留，并不意味着与该词相关的特性尚未实现；反过来，某个关键字的存在也不意味着相应特性一定存在。

表 C.1. SQL 关键字

关键字	PostgreSQL	SQL 99	SQL 92
ABORT	非保留		
ABS		非保留	
ABSOLUTE	非保留	保留	保留
ACCESS	非保留		
ACTION	非保留	保留	保留
ADA		非保留	非保留
ADD	非保留	保留	保留
ADMIN		保留	
AFTER	非保留	保留	
AGGREGATE	非保留	保留	
ALIAS		保留	
ALL	保留	保留	保留
ALLOCATE		保留	保留
ALTER	非保留	保留	保留
ANALYSE	保留		

关键字	PostgreSQL	SQL 99	SQL 92
ANALYZE	保留		
AND	保留	保留	保留
ANY	保留	保留	保留
ARE		保留	保留
ARRAY	保留	保留	
AS	保留	保留	保留
ASC	保留	保留	保留
ASENSITIVE		非保留	
ASSERTION	非保留	保留	保留
ASSIGNMENT	非保留	非保留	
ASYMMETRIC		非保留	
AT	非保留	保留	保留
ATOMIC		非保留	
AUTHORIZATION	保留（可用作函数）	保留	保留
AVG		非保留	保留
BACKWARD	非保留		
BEFORE	非保留	保留	
BEGIN	非保留	保留	保留
BETWEEN	保留（可用作函数）	非保留	保留
BIGINT	非保留（不可用作函数或类型名）		
BINARY	保留（可用作函数）	保留	
BIT	非保留（不可用作函数或类型名）	保留	保留
BITVAR		非保留	
BIT_LENGTH		非保留	保留
BLOB		保留	
BOOLEAN	非保留（不可用作函数或类型名）	保留	
BOTH	保留	保留	保留
BREADTH		保留	
BY	非保留	保留	保留
C		非保留	非保留
CACHE	非保留		
CALL		保留	
CALLED	非保留	非保留	
CARDINALITY		非保留	
CASCADE	非保留	保留	保留
CASCADED		保留	保留
CASE	保留	保留	保留
CAST	保留	保留	保留

关键字	PostgreSQL	SQL 99	SQL 92
CATALOG		保留	保留
CATALOG_NAME		非保留	非保留
CHAIN	非保留	非保留	
CHAR	非保留（不可用作函数或类型名）	保留	保留
CHARACTER	非保留（不可用作函数或类型名）	保留	保留
CHARACTERISTICS	非保留		
CHARACTER_LENGTH		非保留	保留
CHARACTER_SET_CATALOG		非保留	非保留
CHARACTER_SET_NAME		非保留	非保留
CHARACTER_SET_SCHEMA		非保留	非保留
CHAR_LENGTH		非保留	保留
CHECK	保留	保留	保留
CHECKED		非保留	
CHECKPOINT	非保留		
CLASS	非保留	保留	
CLASS_ORIGIN		非保留	非保留
CLOB		保留	
CLOSE	非保留	保留	保留
CLUSTER	非保留		
COALESCE	非保留（不可用作函数或类型名）	非保留	保留
COBOL		非保留	非保留
COLLATE	保留	保留	保留
COLLATION		保留	保留
COLLATION_CATALOG		非保留	非保留
COLLATION_NAME		非保留	非保留
COLLATION_SCHEMA		非保留	非保留
COLUMN	保留	保留	保留
COLUMN_NAME		非保留	非保留
COMMAND_FUNCTION		非保留	非保留
COMMAND_FUNCTION_CODE		非保留	
COMMENT	非保留		
COMMIT	非保留	保留	保留
COMMITTED	非保留	非保留	非保留
COMPLETION		保留	
CONDITION_NUMBER		非保留	非保留
CONNECT		保留	保留

关键字	PostgreSQL	SQL 99	SQL 92
CONNECTION		保留	保留
CONNECTION_NAME		非保留	非保留
CONSTRAINT	保留	保留	保留
CONSTRAINTS	非保留	保留	保留
CONSTRAINT_CATALOG		非保留	非保留
CONSTRAINT_NAME		非保留	非保留
CONSTRAINT_SCHEMA		非保留	非保留
CONSTRUCTOR		保留	
CONTAINS		非保留	
CONTINUE		保留	保留
CONVERSION	非保留		
CONVERT	非保留（不可用作函数或类型名）	非保留	保留
COPY	非保留		
CORRESPONDING		保留	保留
COUNT		非保留	保留
CREATE	保留	保留	保留
CREATEDB	非保留		
CREATEUSER	非保留		
CROSS	保留（可用作函数）	保留	保留
CUBE		保留	
CURRENT		保留	保留
CURRENT_DATE	保留	保留	保留
CURRENT_PATH		保留	
CURRENT_ROLE		保留	
CURRENT_TIME	保留	保留	保留
CURRENT_TIMESTAMP	保留	保留	保留
CURRENT_USER	保留	保留	保留
CURSOR	非保留	保留	保留
CURSOR_NAME		非保留	非保留
CYCLE	非保留	保留	
DATA		保留	非保留
DATABASE	非保留		
DATE		保留	保留
DATETIME_INTERVAL_CODE		非保留	非保留
DATETIME_INTERVAL_PRECISION		非保留	非保留
DAY	非保留	保留	保留
DEALLOCATE	非保留	保留	保留

关键字	PostgreSQL	SQL 99	SQL 92
DEC	非保留（不可用作函数或类型名）	保留	保留
DECIMAL	非保留（不可用作函数或类型名）	保留	保留
DECLARE	非保留	保留	保留
DEFAULT	保留	保留	保留
DEFAULTS	非保留		
DEFERRABLE	保留	保留	保留
DEFERRED	非保留	保留	保留
DEFINED		非保留	
DEFINER	非保留	非保留	
DELETE	非保留	保留	保留
DELIMITER	非保留		
DELIMITERS	非保留		
DEPTH		保留	
DEREF		保留	
DESC	保留	保留	保留
DESCRIBE		保留	保留
DESCRIPTOR		保留	保留
DESTROY		保留	
DESTRUCTOR		保留	
DETERMINISTIC		保留	
DIAGNOSTICS		保留	保留
DICTIONARY		保留	
DISCONNECT		保留	保留
DISPATCH		非保留	
DISTINCT	保留	保留	保留
DO	保留		
DOMAIN	非保留	保留	保留
DOUBLE	非保留	保留	保留
DROP	非保留	保留	保留
DYNAMIC		保留	
DYNAMIC_FUNCTION		非保留	非保留
DYNAMIC_FUNCTION_CODE		非保留	
EACH	非保留	保留	
ELSE	保留	保留	保留
ENCODING	非保留		
ENCRYPTED	非保留		
END	保留	保留	保留
END-EXEC		保留	保留

关键字	PostgreSQL	SQL 99	SQL 92
EQUALS		保留	
ESCAPE	非保留	保留	保留
EVERY		保留	
EXCEPT	保留	保留	保留
EXCEPTION		保留	保留
EXCLUDING	非保留		
EXCLUSIVE	非保留		
EXEC		保留	保留
EXECUTE	非保留	保留	保留
EXISTING		非保留	
EXISTS	非保留（不可用作函数或类型名）	非保留	保留
EXPLAIN	非保留		
EXTERNAL	非保留	保留	保留
EXTRACT	非保留（不可用作函数或类型名）	非保留	保留
FALSE	保留	保留	保留
FETCH	非保留	保留	保留
FINAL		非保留	
FIRST	非保留	保留	保留
FLOAT	非保留（不可用作函数或类型名）	保留	保留
FOR	保留	保留	保留
FORCE	非保留		
FOREIGN	保留	保留	保留
FORTRAN		非保留	非保留
FORWARD	非保留		
FOUND		保留	保留
FREE		保留	
FREEZE	保留（可用作函数）		
FROM	保留	保留	保留
FULL	保留（可用作函数）	保留	保留
FUNCTION	非保留	保留	
G		非保留	
GENERAL		保留	
GENERATED		非保留	
GET		保留	保留
GLOBAL	非保留	保留	保留
GO		保留	保留
GOTO		保留	保留
GRANT	保留	保留	保留

关键字	PostgreSQL	SQL 99	SQL 92
GRANTED		非保留	
GROUP	保留	保留	保留
GROUPING		保留	
HANDLER	非保留		
HAVING	保留	保留	保留
HIERARCHY		非保留	
HOLD	非保留	非保留	
HOST		保留	
HOUR	非保留	保留	保留
IDENTITY		保留	保留
IGNORE		保留	
ILIKE	保留（可用作函数）		
IMMEDIATE	非保留	保留	保留
IMMUTABLE	非保留		
IMPLEMENTATION		非保留	
IMPLICIT	非保留		
IN	保留（可用作函数）	保留	保留
INCLUDING	非保留		
INCREMENT	非保留		
INDEX	非保留		
INDICATOR		保留	保留
INFIX		非保留	
INHERITS	非保留		
INITIALIZE		保留	
INITIALLY	保留	保留	保留
INNER	保留（可用作函数）	保留	保留
INOUT	非保留	保留	
INPUT	非保留	保留	保留
INSENSITIVE	非保留	非保留	保留
INSERT	非保留	保留	保留
INSTANCE		非保留	
INSTANTIABLE		非保留	
INSTEAD	非保留		
INT	非保留（不可用作函数或类型名）	保留	保留
INTEGER	非保留（不可用作函数或类型名）	保留	保留
INTERSECT	保留	保留	保留
INTERVAL	非保留（不可用作函数或类型名）	保留	保留
INTO	保留	保留	保留

关键字	PostgreSQL	SQL 99	SQL 92
INVOKER	非保留	非保留	
IS	保留（可用作函数）	保留	保留
ISNULL	保留（可用作函数）		
ISOLATION	非保留	保留	保留
ITERATE		保留	
JOIN	保留（可用作函数）	保留	保留
K		非保留	
KEY	非保留	保留	保留
KEY_MEMBER		非保留	
KEY_TYPE		非保留	
LANCOMPILER	非保留		
LANGUAGE	非保留	保留	保留
LARGE		保留	
LAST	非保留	保留	保留
LATERAL		保留	
LEADING	保留	保留	保留
LEFT	保留（可用作函数）	保留	保留
LENGTH		非保留	非保留
LESS		保留	
LEVEL	非保留	保留	保留
LIKE	保留（可用作函数）	保留	保留
LIMIT	保留	保留	
LISTEN	非保留		
LOAD	非保留		
LOCAL	非保留	保留	保留
LOCALTIME	保留	保留	
LOCALTIMESTAMP	保留	保留	
LOCATION	非保留		
LOCATOR		保留	
LOCK	非保留		
LOWER		非保留	保留
M		非保留	
MAP		保留	
MATCH	非保留	保留	保留
MAX		非保留	保留
MAXVALUE	非保留		
MESSAGE_LENGTH		非保留	非保留
MESSAGE_OCTET_LENGTH		非保留	非保留
MESSAGE_TEXT		非保留	非保留

关键字	PostgreSQL	SQL 99	SQL 92
METHOD		非保留	
MIN		非保留	保留
MINUTE	非保留	保留	保留
MINVALUE	非保留		
MOD		非保留	
MODE	非保留		
MODIFIES		保留	
MODIFY		保留	
MODULE		保留	保留
MONTH	非保留	保留	保留
MORE		非保留	非保留
MOVE	非保留		
MUMPS		非保留	非保留
NAME		非保留	非保留
NAMES	非保留	保留	保留
NATIONAL	非保留	保留	保留
NATURAL	保留（可用作函数）	保留	保留
NCHAR	非保留（不可用作函数或类型名）	保留	保留
NCLOB		保留	
NEW	保留	保留	
NEXT	非保留	保留	保留
NO	非保留	保留	保留
NOCREATEDB	非保留		
NOCREATEUSER	非保留		
NONE	非保留（不可用作函数或类型名）	保留	
NOT	保留	保留	保留
NOTHING	非保留		
NOTIFY	非保留		
NOTNULL	保留（可用作函数）		
NULL	保留	保留	保留
NULLABLE		非保留	非保留
NULLIF	非保留（不可用作函数或类型名）	非保留	保留
NUMBER		非保留	非保留
NUMERIC	非保留（不可用作函数或类型名）	保留	保留
OBJECT		保留	
OCTET_LENGTH		非保留	保留
OF	非保留	保留	保留

关键字	PostgreSQL	SQL 99	SQL 92
OFF	保留	保留	
OFFSET	保留		
OIDS	非保留		
OLD	保留	保留	
ON	保留	保留	保留
ONLY	保留	保留	保留
OPEN		保留	保留
OPERATION		保留	
OPERATOR	非保留		
OPTION	非保留	保留	保留
OPTIONS		非保留	
OR	保留	保留	保留
ORDER	保留	保留	保留
ORDINALITY		保留	
OUT	非保留	保留	
OUTER	保留（可用作函数）	保留	保留
OUTPUT		保留	保留
OVERLAPS	保留（可用作函数）	非保留	保留
OVERLAY	非保留（不可用作函数或类型名）	非保留	
OVERRIDING		非保留	
OWNER	非保留		
PAD		保留	保留
PARAMETER		保留	
PARAMETERS		保留	
PARAMETER_MODE		非保留	
PARAMETER_NAME		非保留	
PARAMETER_ORDINAL_POSITION		非保留	
PARAMETER_SPECIFIC_CATALOG		非保留	
PARAMETER_SPECIFIC_NAME		非保留	
PARAMETER_SPECIFIC_SCHEMA		非保留	
PARTIAL	非保留	保留	保留
PASCAL		非保留	非保留
PASSWORD	非保留		
PATH	非保留	保留	
PENDANT	非保留		
PLACING	保留		

关键字	PostgreSQL	SQL 99	SQL 92
PLI		非保留	非保留
POSITION	非保留（不可用作函数或类型名）	非保留	保留
POSTFIX		保留	
PRECISION	非保留	保留	保留
PREFIX		保留	
PREORDER		保留	
PREPARE	非保留	保留	保留
PRESERVE	非保留	保留	保留
PRIMARY	保留	保留	保留
PRIOR	非保留	保留	保留
PRIVILEGES	非保留	保留	保留
PROCEDURAL	非保留		
PROCEDURE	非保留	保留	保留
PUBLIC		保留	保留
READ	非保留	保留	保留
READS		保留	
REAL	非保留（不可用作函数或类型名）	保留	保留
RECHECK	非保留		
RECURSIVE		保留	
REF		保留	
REFERENCES	保留	保留	保留
REFERENCING		保留	
REINDEX	非保留		
RELATIVE	非保留	保留	保留
RENAME	非保留		
REPEATABLE		非保留	非保留
REPLACE	非保留		
RESET	非保留		
RESTART	非保留		
RESTRICT	非保留	保留	保留
RESULT		保留	
RETURN		保留	
RETURNED_LENGTH		非保留	非保留
RETURNED_OCTET_LENGTH		非保留	非保留
RETURNED_SQLSTATE		非保留	非保留
RETURNS	非保留	保留	
REVOKE	非保留	保留	保留
RIGHT	保留（可用作函数）	保留	保留

关键字	PostgreSQL	SQL 99	SQL 92
ROLE		保留	
ROLLBACK	非保留	保留	保留
ROLLUP		保留	
ROUTINE		保留	
ROUTINE_CATALOG		非保留	
ROUTINE_NAME		非保留	
ROUTINE_SCHEMA		非保留	
ROW	非保留（不可用作函数或类型名）	保留	
ROWS	非保留	保留	保留
ROW_COUNT		非保留	非保留
RULE	非保留		
SAVEPOINT		保留	
SCALE		非保留	非保留
SCHEMA	非保留	保留	保留
SCHEMA_NAME		非保留	非保留
SCOPE		保留	
SCROLL	非保留	保留	保留
SEARCH		保留	
SECOND	非保留	保留	保留
SECTION		保留	保留
SECURITY	非保留	非保留	
SELECT	保留	保留	保留
SELF		非保留	
SENSITIVE		非保留	
SEQUENCE	非保留	保留	
SERIALIZABLE	非保留	非保留	非保留
SERVER_NAME		非保留	非保留
SESSION	非保留	保留	保留
SESSION_USER	保留	保留	保留
SET	非保留	保留	保留
SETOF	非保留（不可用作函数或类型名）		
SETS		保留	
SHARE	非保留		
SHOW	非保留		
SIMILAR	保留（可用作函数）	非保留	
SIMPLE	非保留	非保留	
SIZE		保留	保留
SMALLINT	非保留（不可用作函数或类型名）	保留	保留

关键字	PostgreSQL	SQL 99	SQL 92
SOME	保留	保留	保留
SOURCE		非保留	
SPACE		保留	保留
SPECIFIC		保留	
SPECIFICTYPE		保留	
SPECIFIC_NAME		非保留	
SQL		保留	保留
SQLCODE			保留
SQLERROR			保留
SQLEXCEPTION		保留	
SQLSTATE		保留	保留
SQLWARNING		保留	
STABLE	非保留		
START	非保留	保留	
STATE		保留	
STATEMENT	非保留	保留	
STATIC		保留	
STATISTICS	非保留		
STDIN	非保留		
STDOUT	非保留		
STORAGE	非保留		
STRICT	非保留		
STRUCTURE		保留	
STYLE		非保留	
SUBCLASS_ORIGIN		非保留	非保留
SUBLIST		非保留	
SUBSTRING	非保留（不可用作函数或类型名）	非保留	保留
SUM		非保留	保留
SYMMETRIC		非保留	
SYSID	非保留		
SYSTEM		非保留	
SYSTEM_USER		保留	保留
TABLE	保留	保留	保留
TABLE_NAME		非保留	非保留
TEMP	非保留		
TEMPLATE	非保留		
TEMPORARY	非保留	保留	保留
TERMINATE		保留	
THAN		保留	

关键字	PostgreSQL	SQL 99	SQL 92
THEN	保留	保留	保留
TIME	非保留（不可用作函数或类型名）	保留	保留
TIMESTAMP	非保留（不可用作函数或类型名）	保留	保留
TIMEZONE_HOUR		保留	保留
TIMEZONE_MINUTE		保留	保留
TO	保留	保留	保留
TOAST	非保留		
TRAILING	保留	保留	保留
TRANSACTION	非保留	保留	保留
TRANSACTIONS_COMMITTED		非保留	
TRANSACTIONS_ROLLED_BACK		非保留	
TRANSACTION_ACTIVE		非保留	
TRANSFORM		非保留	
TRANSFORMS		非保留	
TRANSLATE		非保留	保留
TRANSLATION		保留	保留
TREAT	非保留（不可用作函数或类型名）	保留	
TRIGGER	非保留	保留	
TRIGGER_CATALOG		非保留	
TRIGGER_NAME		非保留	
TRIGGER_SCHEMA		非保留	
TRIM	非保留（不可用作函数或类型名）	非保留	保留
TRUE	保留	保留	保留
TRUNCATE	非保留		
TRUSTED	非保留		
TYPE	非保留	非保留	非保留
UNCOMMITTED		非保留	非保留
UNDER		保留	
UNENCRYPTED	非保留		
UNION	保留	保留	保留
UNIQUE	保留	保留	保留
UNKNOWN	非保留	保留	保留
UNLISTEN	非保留		
UNNAMED		非保留	非保留
UNNEST		保留	
UNTIL	非保留		

关键字	PostgreSQL	SQL 99	SQL 92
UPDATE	非保留	保留	保留
UPPER		非保留	保留
USAGE	非保留	保留	保留
USER	保留	保留	保留
USER_DEFINED_TYPE_CATALOG		非保留	
USER_DEFINED_TYPE_NAME		非保留	
USER_DEFINED_TYPE_SCHEMA		非保留	
USING	保留	保留	保留
VACUUM	非保留		
VALID	非保留		
VALIDATOR	非保留		
VALUE		保留	保留
VALUES	非保留	保留	保留
VARCHAR	非保留（不可用作函数或类型名）	保留	保留
VARIABLE		保留	
VARYING	非保留	保留	保留
VERBOSE	保留（可用作函数）		
VERSION	非保留		
VIEW	非保留	保留	保留
VOLATILE	非保留		
WHEN	保留	保留	保留
WHENEVER		保留	保留
WHERE	保留	保留	保留
WITH	非保留	保留	保留
WITHOUT	非保留	保留	
WORK	非保留	保留	保留
WRITE	非保留	保留	保留
YEAR	非保留	保留	保留
ZONE	非保留	保留	保留

附录 D. SQL 符合性

本节尝试概述 PostgreSQL 在多大程度上符合 SQL 标准。完全符合该标准或给出一份完整的符合性声明既复杂又没有特别的用处，因此本节只能给出一个概览。

SQL 标准的正式名称是 ISO/IEC 9075“数据库语言 SQL”。该标准不时发布修订版本；最近一次出现在 1999 年。那个版本称为 ISO/IEC 9075:1999，或非正式地称为 SQL99。在此之前的版本是 SQL92。PostgreSQL 的开发倾向于在不与传统特性或常识相矛盾的前提下，以符合该标准的最新官方版本为目标。在撰写本文时，该标准的一次新修订正在投票表决中，如果获得批准，它最终将成为未来 PostgreSQL 开发的符合性目标。

SQL92 为符合性定义了三个特性集：入门（Entry）、中级（Intermediate）和完整（Full）。多数宣称符合 SQL 标准的数据库管理系统，实际上只符合入门级别，因为中级和完整级别中的全部特性要么过于庞杂，要么与既有行为相冲突。

SQL99 定义了大量独立特性，而不是 SQL92 中那种效果不佳且过于宽泛的三级分类。其中很大一个子集构成了“核心”特性，也就是每个符合标准的 SQL 实现都必须提供的特性。其余的特性则完全是可选的。某些可选特性被组合在一起，形成“包”，SQL 实现可以声明符合这些包，从而表明符合特定的特性组。

SQL99 标准也分为 5 个部分：框架（Framework）、基础（Foundation）、调用级接口（Call Level Interface）、持久存储模块（Persistent Stored Modules）和宿主语言绑定（Host Language Bindings）。PostgreSQL 只涵盖第 1、2、5 部分。第 3 部分类似于 ODBC 接口，第 4 部分类似于 PL/pgSQL 编程语言，但这两种情况下都没有特别意图实现精确的符合。

在下面两节中，我们先列出 PostgreSQL 支持的特性，然后列出 SQL99 中定义但 PostgreSQL 尚不支持的特性。这两个列表都是近似的：被列为受支持的特性可能存在个别不符合标准的细节，而不受支持的特性也可能实际上已实现了很大一部分。
关于什么可用什么不可用的最准确信息，始终以文档正文为准。

> **注意**
>
> 特性代码中包含连字符的是子特性。因此，如果某个特定的子特性不受支持，那么即使其他一些子特性受支持，该主特性也会被列为不受支持。

D.1. 支持的特性

标识符	包	描述	注释
B012	Core	Embedded C	
B021		Direct SQL	
E011	Core	Numeric data types	
E011-01	Core	INTEGER and SMALLINT data types	
E011-02	Core	REAL, DOUBLE PRECIS ION, and FLOAT data types	
E011-03	Core	DECIMAL and NUMERIC data types	
E011-04	Core	Arithmetic operators	
E011-05	Core	Numeric comparison	

标识符	包	描述	注释
E011-06	Core	Implicit casting among the numeric data types	
E021	Core	Character data types	
E021-01	Core	CHARACTER data type	
E021-02	Core	CHARACTER VARYING data type	
E021-03	Core	Character literals	
E021-04	Core	CHARACTER_LENGTH function	
E021-05	Core	OCTET_LENGTH functi on	
E021-06	Core	SUBSTRING function	
E021-07	Core	Character concatenatio n	
E021-08	Core	UPPER and LOWER functions	
E021-09	Core	TRIM function	
E021-10	Core	Implicit casting among the character data types	
E021-11	Core	POSITION function	
E021-12	Core	Character comparison	
E031	Core	Identifiers	
E031-01	Core	Delimited identifiers	
E031-02	Core	Lower case identifiers	
E031-03	Core	Trailing underscore	
E051	Core	Basic query specificatio n	
E051-01	Core	SELECT DISTINCT	
E051-02	Core	GROUP BY clause	
E051-04	Core	GROUP BY can contain columns not in <select list>	
E051-05	Core	Select list items can be renamed	AS is required
E051-06	Core	HAVING clause	
E051-07	Core	Qualified * in select list	
E051-08	Core	Correlation names in the FROM clause	
E051-09	Core	Rename columns in the FROM clause	
E061	Core	Basic predicates and search conditions	
E061-01	Core	Comparison predicate	
E061-02	Core	BETWEEN predicate	

标识符	包	描述	注释
E061-03	Core	IN predicate with list of values	
E061-04	Core	LIKE predicate	
E061-05	Core	LIKE predicate ESCAPE clause	
E061-06	Core	NULL predicate	
E061-07	Core	Quantified comparison predicate	
E061-08	Core	EXISTS predicate	
E061-09	Core	Subqueries in compar ison predicate	
E061-11	Core	Subqueries in IN predic ate	
E061-12	Core	Subqueries in quanti fied comparison predic ate	
E061-13	Core	Correlated subqueries	
E061-14	Core	Search condition	
E071	Core	Basic query expres sions	
E071-01	Core	UNION DISTINCT table operator	
E071-02	Core	UNION ALL table operator	
E071-03	Core	EXCEPT DISTINCT table operator	
E071-05	Core	Columns combined via table operators need not have exactly the same data type	
E071-06	Core	Table operators in subqueries	
E081-01	Core	SELECT privilege	
E081-02	Core	DELETE privilege	
E081-03	Core	INSERT privilege at the table level	
E081-04	Core	UPDATE privilege at the table level	
E081-06	Core	REFERENCES privilege at the table level	
E081-08	Core	WITH GRANT OPTION	
E091	Core	Set functions	
E091-01	Core	AVG	
E091-02	Core	COUNT	
E091-03	Core	MAX	
E091-04	Core	MIN	

标识符	包	描述	注释
E091-05	Core	SUM	
E091-06	Core	ALL quantifier	
E091-07	Core	DISTINCT quantifier	
E101	Core	Basic data manipu lation	
E101-01	Core	INSERT statement	
E101-03	Core	Searched UPDATE statement	
E101-04	Core	Searched DELETE statement	
E111	Core	Single row SELECT statement	
E121-01	Core	DECLARE CURSOR	
E121-02	Core	ORDER BY columns need not be in select list	
E121-03	Core	Value expressions in ORDER BY clause	
E121-04	Core	OPEN statement	
E121-08	Core	CLOSE statement	
E121-10	Core	FETCH statement implic it NEXT	
E121-17	Core	WITH HOLD cursors	
E131	Core	Null value support (nulls in lieu of values)	
E141	Core	Basic integrity constr aints	
E141-01	Core	NOT NULL constraints	
E141-02	Core	UNIQUE constraints of NOT NULL columns	
E141-03	Core	PRIMARY KEY constr aints	
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	
E141-06	Core	CHECK constraints	
E141-07	Core	Column defaults	
E141-08	Core	NOT NULL inferred on PRIMARY KEY	
E141-10	Core	Names in a foreign key can be specified in any order	
E151	Core	Transaction support	
E151-01	Core	COMMIT statement	

标识符	包	描述	注释
E151-02	Core	ROLLBACK statement	
E152	Core	Basic SET TRANSACTION statement	
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	
E161	Core	SQL comments using leading double minus	
E171	Core	SQLSTATE support	
F021	Core	Basic information schema	
F021-01	Core	COLUMNS view	
F021-02	Core	TABLES view	
F021-03	Core	VIEWS view	
F021-04	Core	TABLE_CONSTRAINTS view	
F021-05	Core	REFERENTIAL_CONSTRAINTS view	
F021-06	Core	CHECK_CONSTRAINTS view	
F031	Core	Basic schema manipulation	
F031-01	Core	CREATE TABLE statement to create persistent base tables	
F031-02	Core	CREATE VIEW statement	
F031-03	Core	GRANT statement	
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause	
F031-13	Core	DROP TABLE statement: RESTRICT clause	
F031-16	Core	DROP VIEW statement: RESTRICT clause	
F031-19	Core	REVOKE statement: RESTRICT clause	
F032		CASCADE drop behavior	
F033		ALTER TABLE statement: DROP COLUMN clause	
F034		Extended REVOKE statement	

标识符	包	描述	注释
F034-01		REVOKE statement performed by other than the owner of a schema object	
F034-02		REVOKE statement: GRANT OPTION FOR clause	
F034-03		REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	
F041	Core	Basic joined table	
F041-01	Core	Inner join (but not necessarily the INNER keyword)	
F041-02	Core	INNER keyword	
F041-03	Core	LEFT OUTER JOIN	
F041-04	Core	RIGHT OUTER JOIN	
F041-05	Core	Outer joins can be nested	
F041-07	Core	The inner table in a left or right outer join can also be used in an inner join	
F041-08	Core	All comparison operat ors are supported (rather than just =)	
F051	Core	Basic date and time	
F051-01	Core	DATE data type (includ ing support of DATE literal)	
F051-02	Core	TIME data type (includ ing support of TIME literal) with fractional seconds precision of at least 0	
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional second s precision of at least 0 and 6	
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types	
F051-05	Core	Explicit CAST betwee n datetime types and character types	
F051-06	Core	CURRENT_DATE	
F051-07	Core	LOCALTIME	

标识符	包	描述	注释
F051-08	Core	LOCALTIMESTAMP	
F052	Enhanced datetime facilities	Intervals and datetime arithmetic	
F081	Core	UNION and EXCEPT in views	
F111-02		READ COMMITTED isolation level	
F131	Core	Grouped operations	
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	
F131-02	Core	Multiple tables suppor ted in queries with grouped views	
F131-03	Core	Set functions suppor ted in queries with grouped views	
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views	
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views	
F171		Multiple schemas per user	
F191	Enhanced integrity management	Referential delete actions	
F201	Core	CAST function	
F221	Core	Explicit defaults	
F222		INSERT statement: DEFAULT VALUES clause	
F231		Privilege Tables	
F231-01		TABLE_PRIVILEGES view	
F231-02		COLUMN_PRIVILEGES view	
F231-03		USAGE_PRIVILEGES view	
F251		Domain support	
F261	Core	CASE expression	
F261-01	Core	Simple CASE	
F261-02	Core	Searched CASE	
F261-03	Core	NULLIF	
F261-04	Core	COALESCE	

标识符	包	描述	注释
F271		Compound character literals	
F281		LIKE enhancements	
F302	OLAP facilities	INTERSECT table operat or	
F302-01	OLAP facilities	INTERSECT DISTINCT table operator	
F302-02	OLAP facilities	INTERSECT ALL table operator	
F304	OLAP facilities	EXCEPT ALL table operator	
F311-01	Core	CREATE SCHEMA	
F311-02	Core	CREATE TABLE for persis tent base tables	
F311-03	Core	CREATE VIEW	
F311-05	Core	GRANT statement	
F321		User authorization	
F361		Subprogram support	
F381		Extended schema manipulation	
F381-01		ALTER TABLE statem ent: ALTER COLUMN clause	
F381-02		ALTER TABLE statem ent: ADD CONSTRAINT clause	
F381-03		ALTER TABLE statem ent: DROP CONSTRAINT clause	
F391		Long identifiers	
F401	OLAP facilities	Extended joined table	
F401-01	OLAP facilities	NATURAL JOIN	
F401-02	OLAP facilities	FULL OUTER JOIN	
F401-03	OLAP facilities	UNION JOIN	
F401-04	OLAP facilities	CROSS JOIN	
F411	Enhanced datetime facilities	Time zone specification	
F421		National character	
F431		Read-only scrollable cursors	
F431-01		FETCH with explicit NEXT	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	

标识符	包	描述	注释
F431-06		FETCH RELATIVE	
F441		Extended set function support	
F471	Core	Scalar subquery values	
F481	Core	Expanded NULL predic ate	
F491	Enhanced integrity management	Constraint manage ment	
F501	Core	Features and confor mance views	
F501-01	Core	SQL_FEATURES view	
F501-02	Core	SQL_SIZING view	
F501-03	Core	SQL_LANGUAGES view	
F502		Enhanced docume ntation tables	
F502-01		SQL_SIZING_PROFILES view	
F502-02		SQL_IMPLEMENTATION_ INFO view	
F502-03		SQL_PACKAGES view	
F511		BIT data type	
F531		Temporary tables	
F555	Enhanced datetime facilities	Enhanced seconds precision	
F561		Full value expressions	
F571		Truth value tests	
F591	OLAP facilities	Derived tables	
F611		Indicator data types	
F651		Catalog name qualif iers	
F701	Enhanced integrity management	Referential update actions	
F711		ALTER domain	
F761		Session management	
F771		Connection manage ment	
F781		Self-referencing operat ions	
F791		Insensitive cursors	
F801		Full set function	
S071	Enhanced object support	SQL paths in function and type name resolu tion	
S111	Enhanced object support	ONLY in query expres sions	

标识符	包	描述	注释
S211	Enhanced object support, SQL/MM support	User-defined cast functions	
T031		BOOLEAN data type	
T141		SIMILAR predicate	
T151		DISTINCT predicate	
T171		LIKE clause in table definition	
T191	Enhanced integrity management	Referential action RESTRICT	
T201	Enhanced integrity management	Comparable data types for referential constr aints	
T211-01	Enhanced integrity management, Active database	Triggers activated on UPDATE, INSERT, or DELETE of one base table	
T211-02	Enhanced integrity management, Active database	BEFORE triggers	
T211-03	Enhanced integrity management, Active database	AFTER triggers	
T211-04	Enhanced integrity management, Active database	FOR EACH ROW triggers	
T211-07	Enhanced integrity management, Active database	TRIGGER privilege	
T212	Enhanced integrity management	Enhanced trigger capability	
T231		SENSITIVE cursors	
T241		START TRANSACTION statement	
T312		OVERLAY function	
T321-01	Core	User-defined functions with no overloading	
T321-03	Core	Function invocation	
T321-06	Core	ROUTINES view	
T321-07	Core	PARAMETERS view	
T322	PSM, SQL/MM support	Overloading of SQL-invoked functions and procedures	
T323		Explicit security for external routines	
T351		Bracketed SQL comments (/*...*/ comments)	

标识符	包	描述	注释
T441		ABS and MOD functions	
T501		Enhanced EXISTS predicate	
T551		Optional key words for default syntax	
T581		Regular expression substring function	
T591		UNIQUE constraints of possibly null columns	

D.2. 不支持的特性

下面这些 SQL99 中定义的特性在这个版本的 PostgreSQL 中尚未实现。在少数情况下，有等价的功能可用。

标识符	包	描述	注释
B011	Core	Embedded Ada	
B013	Core	Embedded COBOL	
B014	Core	Embedded Fortran	
B015	Core	Embedded MUMPS	
B016	Core	Embedded Pascal	
B017	Core	Embedded PL/I	
B031		Basic dynamic SQL	
B032		Extended dynamic SQL	
B032-01		<describe input> statement	
B041		Extensions to embedd ed SQL exception declarations	
B051		Enhanced execution rights	
E081	Core	Basic Privileges	
E081-05	Core	UPDATE privilege at the column level	
E081-07	Core	REFERENCES privilege at the column level	
E121	Core	Basic cursor support	
E121-06	Core	Positioned UPDATE statement	
E121-07	Core	Positioned DELETE statement	
E153	Core	Updatable queries with subqueries	
E182	Core	Module language	
F111		Isolation levels other than SERIALIZABLE	

标识符	包	描述	注释
F111-01		READ UNCOMMITTED isolation level	
F111-03		REPEATABLE READ isolation level	
F121		Basic diagnostics management	
F121-01		GET DIAGNOSTICS statement	
F121-02		SET TRANSACTION statement: DIAGNO STICS SIZE clause	
F181		Multiple module support	
F291		UNIQUE predicate	
F301		CORRESPONDING in query expressions	
F311	Core	Schema definition statement	
F311-04	Core	CREATE VIEW: WITH CHECK OPTION	
F341		Usage tables	
F451		Character set definition	
F461		Named character sets	
F521	Enhanced integrity management	Assertions	
F641	OLAP facilities	Row and table constr uctors	
F661		Simple tables	
F671	Enhanced integrity management	Subqueries in CHECK	intentionally omitted
F691		Collation and transl ation	
F721		Deferrable constraints	foreign keys only
F731		INSERT column privil eges	
F741		Referential MATCH types	no partial match yet
F751		View CHECK enhanc ements	
F811		Extended flagging	
F812	Core	Basic flagging	
F813		Extended flagging for "Core SQL Flagging" and "Catalog Lookup" only	
F821		Local table references	
F831		Full cursor update	

标识符	包	描述	注释
F831-01		Updatable scrollable cursors	
F831-02		Updatable ordered cursors	
S011	Core	Distinct data types	
S011-01	Core	USER_DEFINED_TYPES view	
S023	Basic object support, SQL/MM support	Basic structured types	
S024	Enhanced object support, SQL/MM support	Enhanced structured types	
S041	Basic object support	Basic reference types	
S043	Enhanced object support	Enhanced reference types	
S051	Basic object support	Create table of type	
S081	Enhanced object support	Subtables	
S091	SQL/MM support	Basic array support	
S091-01	SQL/MM support	Arrays of built-in data types	
S091-02	SQL/MM support	Arrays of distinct types	
S091-03	SQL/MM support	Array expressions	
S092	SQL/MM support	Arrays of user-defined types	
S094		Arrays of reference types	
S151	Basic object support	Type predicate	
S161	Enhanced object support	Subtype treatment	
S201		SQL routines on arrays	
S201-01		Array parameters	
S201-02		Array as result type of functions	
S231	Enhanced object support	Structured type locato rs	
S232		Array locators	
S241	Enhanced object support	Transform functions	
S251		User-defined orderings	
S261		Specific type method	
T011		Timestamp in Inform ation Schema	
T041	Basic object support	Basic LOB data type support	
T041-01	Basic object support	BLOB data type	

标识符	包	描述	注释
T041-02	Basic object support	CLOB data type	
T041-03	Basic object support	POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types	
T041-04	Basic object support	Concatenation of LOB data types	
T041-05	Basic object support	LOB locator: non-holdable	
T042		Extended LOB data type support	
T051		Row types	
T111		Updatable joins, unions, and columns	
T121		WITH (excluding RECURSIVE) in query expression	
T131		Recursive query	
T211	Enhanced integrity management, Active database	Basic trigger capability	
T211-05	Enhanced integrity management, Active database	Ability to specify a search condition that must be true before the trigger is invoked	
T211-06	Enhanced integrity management, Active database	Support for run-time rules for the interaction of triggers and constraints	
T211-08	Enhanced integrity management, Active database	Multiple triggers for the same event are executed in the order in which they were created	intentionally omitted
T251		SET TRANSACTION statement: LOCAL option	
T261		Chained transactions	
T271		Savepoints	
T281		SELECT privilege with column granularity	
T301		Functional Dependencies	
T321	Core	Basic SQL-invoked routines	
T321-02	Core	User-defined stored procedures with no overloading	
T321-04	Core	CALL statement	

标识符	包	描述	注释
T321-05	Core	RETURN statement	
T331		Basic roles	
T332		Extended roles	
T401		INSERT into a cursor	
T411		UPDATE statement: SET ROW option	
T431	OLAP facilities	CUBE and ROLLUP operations	
T461		Symmetric BETWEEN predicate	
T471		Result sets return value	
T491		LATERAL derived table	
T511		Transaction counts	
T541		Updatable table refere nces	
T561		Holdable locators	
T571		Array-returning extern al SQL-invoked functi ons	
T601		Local cursor references	

附录 E. 发布说明

发布说明包含每个PostgreSQL发布中的重要变更，并将主要特性和迁移问题列在最前面。
发布说明不包含只影响少数用户的变更，也不包含仅限内部实现、因而对用户不可见的变更。
例如，优化器几乎在每次发布中都会得到改进，
但用户通常只会把这些改进感知为查询速度更快。

每个版本的一份完整变更列表可以通过查看该版本的 Git 日志获得。`pgsql-committers` 邮件列表[1]也记录了所有源代码变更。另有一个Web 界面[2]可显示特定文件的变更。

每个条目旁边标出的姓名表示该条目的主要开发者。当然，
所有变更都经历了社区讨论和补丁审查，因此每个条目实际上都是社区共同努力的成果。

E.1. 版本 7.4.30

发行日期
2010-10-04

本次发布包含对 7.4.29 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

这预计是 PostgreSQL 7.4.X 系列的最后一个发行版。建议用户尽快更新到较新的发行分支。

E.1.1. 迁移到版本 7.4.30

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.26 的版本升级，请参阅 7.4.26 的发行说明。

E.1.2. 变更

- 在 PL/Perl 和 PL/Tcl 中为每个调用方 SQL 用户 ID 使用单独的解释器（Tom Lane）

  此变更防止了可通过破坏稍后将在同一会话中以另一 SQL 用户身份执行的 Perl 或 Tcl 代码（例如在 `SECURITY DEFINER` 函数内）而造成的安全问题。大多数脚本语言都提供许多实现这一点的手段，例如重新定义目标函数调用的标准函数或操作符。如果没有此变更，任何拥有 Perl 或 Tcl 语言使用权的 SQL 用户基本上都能以目标函数所有者的 SQL 权限为所欲为。

  此变更的代价是 Perl 和 Tcl 函数之间的有意通信变得更困难。为提供出路，PL/PerlU 和 PL/TclU 函数继续每会话只使用一个解释器。这不被视为安全问题，因为所有这类函数本来就以数据库超级用户的信任级别执行。

  声称提供可信执行的第三方过程语言很可能存在类似的安全问题。
  我们建议就安全关键用途联系你所依赖的任何 PL 的作者。

  感谢 Tim Bunce 指出此问题（CVE-2010-3433）。

- 禁止以并非它预期配合使用的系统目录列的参数调用 `pg_get_expr()`，以防止可能的崩溃（Heikki Linnakangas、Tom Lane）

[1] http://archives.postgresql.org/pgsql-committers/
[2] http://git.postgresql.org/gitweb?p=postgresql.git;a=summary

- 修复“无法处理未规划的子查询”错误（Tom Lane）

  当子查询包含一个连接别名引用，而该引用展开成包含另一个子查询的表达式时，会出现此错误。

- 写入锁文件（`postmaster.pid` 和套接字锁文件两者）内容时注意对其执行 fsync（Tom Lane）

  如果机器在 postmaster 启动后不久崩溃，这一疏漏可能导致锁文件内容损坏。这又可能使后续启动 postmaster 的尝试失败，直到手工删除锁文件。

- 改进 `contrib/dblink` 对包含已删除列的表的处理（Tom Lane）

- 修复 `contrib/dblink` 中“重复连接名”错误后的连接泄漏（Itagaki Takahiro）

- 更新构建基础设施和文档，反映源代码仓库从 CVS 迁移到 Git 的变化（Magnus Hagander 等）

E.2. 版本 7.4.29

发行日期
2010-05-17

本次发布包含对 7.4.28 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

PostgreSQL 社区将于 2010 年 7 月停止对 7.4.X 版本系列的更新。建议用户尽快更新到较新的版本分支。

E.2.1. 迁移到版本 7.4.29

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.26 的版本升级，请参阅 7.4.26 的发行说明。

E.2.2. 变更

- 用应用于整个解释器的操作码掩码来强制 `plperl` 中的限制，而不是使用 `Safe.pm`（Tim Bunce、Andrew Dunstan）

  近期的事态发展使我们确信 `Safe.pm` 太不安全，不能依靠它使 `plperl` 可信。此变更完全移除了对 `Safe.pm` 的使用，改为使用始终应用操作码掩码的独立解释器。此变更的良好副作用包括：现在可以在 `plperl` 中以自然的方式使用 Perl 的 `strict` 编译指令，Perl 的 `$a` 和 `$b` 变量在排序例程中按预期工作，以及函数编译显著加快。（CVE-2010-1169）

- 防止 PL/Tcl 执行来自 `pltcl_modules` 的不可信代码（Tom）

  PL/Tcl 从数据库表自动装载 Tcl 代码的特性可能被利用于特洛伊木马攻击，因为对谁能创建该表或向其中插入没有限制。此变更禁用该特性，除非 `pltcl_modules` 由超级用户拥有。（但是，不检查该表上的权限，
  因此确实需要较不安全的模块表的安装仍可以向受信任的非超级用户授予适当权限。）此外，除非确实要执行 `pltclu` 函数，否则防止把代码装载到无限制的“普通”Tcl 解释器中。（CVE-2010-1170）

- 不允许非特权用户重置仅超级用户可用的参数设置（Alvaro）

  此前，如果非特权用户为自己运行 `ALTER USER ... RESET ALL`，或为他拥有的数据库运行 `ALTER DATABASE ... RESET ALL`，这将移除该用户或数据库的所有特殊参数设置，包括本应只可由超级用户更改的那些。现在，`ALTER` 只移除用户有权更改的参数。

- 避免后端关闭发生在需要向日志条目添加 `CONTEXT` 内容时导致可能的崩溃（Tom）

  某些情况下上下文打印函数会失败，因为到打印日志消息时当前事务已被回滚。

- 为现代 Perl 版本更新 pl/perl 的 `ppport.h`（Andrew）

- 修复 pl/python 中的若干内存泄漏（Andreas Freund、Tom）

- 确保 `contrib/pgstattuple` 函数及时响应取消中断（Tatsuhito Kasahara）

- 让服务器启动正确处理 `shmget()` 对已存在的共享内存段返回 `EINVAL` 的情形（Tom）

  此行为曾在包括 OS X 在内的 BSD 衍生内核上观察到。它导致完全误导性的启动失败，抱怨共享内存请求大小过大。

E.3. 版本 7.4.28

发行日期
2010-03-15

本次发布包含对 7.4.27 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

PostgreSQL 社区将于 2010 年 7 月停止对 7.4.X 版本系列的更新。建议用户尽快更新到较新的版本分支。

E.3.1. 迁移到版本 7.4.28

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.26 的版本升级，请参阅 7.4.26 的发行说明。

E.3.2. 变更

- 新增配置参数 `ssl_renegotiation_limit`，控制对 SSL 连接进行会话密钥重协商的频率（Magnus）

  可以把它设为零以完全禁用重协商，这在使用有缺陷的 SSL 库时可能是必需的。特别是，某些厂商发布的 CVE-2009-3555 临时补丁会导致重协商尝试失败。

- 让 `bit` 类型的 `substring()` 把任何负长度都视为“字符串的其余全部”（Tom）

  以前的编码只对 `-1` 这样处理，对其他负值会产生无效的结果值，可能导致崩溃（CVE-2010-0442）。

- 修复若干病态缓慢的正则表达式匹配情形（Tom）

- 读取 `pg_hba.conf` 及相关文件时，如果 `@` 出现在引号内，不要把 `@something` 当作文件包含请求；另外，永远不要把单独的 `@` 当作文件包含请求（Tom）

  这防止了角色名或数据库名以 `@` 开头时的古怪行为。如果需要包含路径名包含空格的文件，仍然可以做到，但必须写 `@"/path to/file"` 而不是把引号放在整个构造之外。

- 防止在某些平台上把目录命名为 `pg_hba.conf` 及相关文件中的包含目标时出现无限循环（Tom）

- 确保 PL/Tcl 完全初始化 Tcl 解释器（Tom）

  此疏漏唯一已知的症状是使用 Tcl 8.5 或更高版本时 Tcl 的 `clock` 命令行为异常。

- 防止在向 `dblink_build_sql_*` 函数指定过多键列时 `contrib/dblink` 崩溃（Rushabh Lathia、Joe Conway）

E.4. 版本 7.4.27

发行日期
2009-12-14

本次发布包含对 7.4.26 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.4.1. 迁移到版本 7.4.27

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.26 的版本升级，请参阅 7.4.26 的发行说明。

E.4.2. 变更

- 防范索引函数更改会话局部状态导致的间接安全威胁（Gurjeet Singh、Tom）

  此变更防止据称不可变的索引函数可能破坏超级用户的会话（CVE-2009-4136）。

- 拒绝在公共名（CN）字段中包含嵌入空字节的 SSL 证书（Magnus）

  这防止在 SSL 验证期间证书与服务器或客户端名称的非预期匹配（CVE-2009-4034）。

- 修复后端启动时缓存初始化期间可能的崩溃（Tom）

- 防止信号在不安全的时刻中断 `VACUUM`（Alvaro）

  此修复防止了 `VACUUM FULL` 在已提交其元组移动后被取消时的 PANIC，以及普通 `VACUUM` 在截断表之后被中断时的瞬态错误。

- 修复哈希表大小计算中因整数溢出而可能的崩溃（Tom）

  这可能发生在规划器对哈希连接结果大小的估计极大时。

- 修复 `inet/cidr` 比较中非常罕见的崩溃（Chris Mikkelson）

- 修复 PAM 口令处理使其更健壮（Tom）

已知以前的代码在 Linux `pam_krb5` PAM 模块与 Microsoft Active Directory 域控制器组合下失败。它在其他地方可能也有问题，因为它对 PAM 栈会传递给它什么参数做了无根据的假设。

- 让 postmaster 忽略连接请求包中的任何 `application_name` 参数，以改进与未来 libpq 版本的兼容性（Tom）

E.5. 版本 7.4.26

发行日期
2009-09-09

本次发布包含对 7.4.25 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.5.1. 迁移到版本 7.4.26

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你在 `interval` 列上有任何哈希索引，必须在更新到 7.4.26 后对它们执行 `REINDEX`。另外，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.5.2. 变更

- 不允许在安全定义者函数内使用 `RESET ROLE` 和 `RESET SESSION AUTHORIZATION`（Tom、Heikki）

  这覆盖了先前禁止在安全定义者函数内使用 `SET ROLE` 和 `SET SESSION AUTHORIZATION` 的补丁中遗漏的一种情形。（参见 CVE-2007-6600）

- 修复出现在外层聚合函数参数中的子 SELECT 的处理（Tom）

- 修复 `interval` 数据类型的哈希计算（Tom）

  这纠正了 interval 值上哈希连接的错误结果。它还更改了 interval 列上哈希索引的内容。如果你有任何这样的索引，必须在更新后对它们执行 `REINDEX`。

- 修复 *x* 超过 200 万且使用整数日期时间时 `INTERVAL 'x ms'` 的溢出（Alex Hunsaker）

- 修复点与线段之间距离的计算（Tom）

  这导致多个几何操作符的结果不正确。

- 修复 `money` 数据类型，使其在货币金额没有小数位的区域设置（例如日本）中工作（Itagaki Takahiro）

- 正确地对像 `00:12:57.999999999999999999999999999` 这样的日期时间输入四舍五入（Tom）

- 修复 GiST R-tree 操作符类中页分裂点选择不佳的问题（Teodor）

- 修复 plperl 初始化中的可移植性问题（Andrew Dunstan）

- 改进 libpq 从 `COPY FROM STDIN` 期间错误中恢复的代码的健壮性（Tom）
- 避免在两个库都安装时包含相互冲突的 readline 和 editline 头文件（Zdenek Kotala）

E.6. 版本 7.4.25

发行日期
2009-03-16

本次发布包含对 7.4.24 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.6.1. 迁移到版本 7.4.25

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.6.2. 变更

- 防止编码转换失败时的错误递归崩溃（Tom）

  此变更扩展了上两个小版本中针对相关失败情形所做的修复。
  以前的修复是针对最初的问题报告量身定制的，但我们现在认识到，编码转换函数抛出的
  任何错误都可能在试图报告该错误时导致无限递归。因此，
  解决方案是：如果发现已处于递归错误报告情形，就禁用翻译和编码转换并报告错误消息的纯 ASCII 形式。（CVE-2009-0922）

- 不允许 `CREATE CONVERSION` 为指定的转换函数使用错误的编码（Heikki）

  这防止了编码转换失败的一种可能情形。前一个变更是兜底，防范同一领域其他类型的失败。

- 修复 `to_char()` 被给定与数据参数类型不适当的格式代码时的核心转储（Tom）
- 把 `MUST`（毛里求斯岛夏令时）加入已知时区缩写的默认列表（Xavier Bugaud）

E.7. 版本 7.4.24

发行日期
2009-02-02

本次发布包含对 7.4.23 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.7.1. 迁移到版本 7.4.24

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.7.2. 变更

- 改进 `headline()` 函数中对 URL 的处理（Teodor）
- 改进 `headline()` 函数对超长标题的处理（Teodor）
- 防止在为指定的编码对创建了错误转换函数的编码转换时可能的断言失败或错误转换（Tom、Heikki）
- 避免 `VACUUM` 中对小表的不必要锁定（Heikki）
- 修复 `contrib/tsearch2` 的 `get_covers()` 函数中的未初始化变量（Teodor）
- 修复 `to_char()` 处理 `TH` 格式代码中的缺陷（Andreas Scherbaum）
- 让所有文档在适当之处引用 `pgsql-bugs` 和/或 `pgsql-hackers`，而不是现已停用的 `pgsql-ports` 和 `pgsql-patches` 邮件列表（Tom）

E.8. 版本 7.4.23

发行日期
2008-11-03

本次发布包含对 7.4.22 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.8.1. 迁移到版本 7.4.23

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.8.2. 变更

- 修复客户端编码无法表示本地化错误消息时的后端崩溃（Tom）

  我们此前处理过类似问题，但如果“字符没有等价物”消息本身无法转换，仍会失败。
  修复方法是在检测到这种情形时禁用本地化并发送纯 ASCII 错误消息。

- 修复单个查询项匹配文本首词时 tsearch2 标题生成不正确的问题（Sushant Sinha）
- 修复 `--enable-integer-datetimes` 构建中使用非 ISO datestyle 时 interval 值小数秒显示不当的问题（Ron Mayer）
- 确保 `SPI_getvalue` 和 `SPI_getbinval` 在传入的元组和元组描述符列数不同时行为正确（Tom）

  当表添加或删除过列时这种情况是正常的，但这两个函数没有正确处理它。
  唯一可能的后果是错误的错误指示。

- 修复 ecpg 对 `CREATE USER` 的解析（Michael）

E.9. 版本 7.4.22

发行日期
2008-09-22

本次发布包含对 7.4.21 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.9.1. 迁移到版本 7.4.22

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.9.2. 变更

- 修复日期时间输入函数在 64 位平台上正确检测整数溢出（Tom）
- 改进向 syslog 写很长日志消息的性能（Tom）
- 修复 `SELECT DISTINCT ON` 查询上游标反向扫描的缺陷（Tom）
- 修复规划器，使其估计产生布尔结果的 `GROUP BY` 表达式总是产生两个组，而不考虑表达式的内容（Tom）

  对于某些布尔测试（如 *col* `IS NULL`），这比常规的 `GROUP BY` 估计准确得多。
- 改进 pg_dump 和 pg_restore 在发送 SQL 命令失败后的错误报告（Tom）

E.10. 版本 7.4.21

发行日期
2008-06-12

此发行版在 7.4.20 的基础上包含一个严重的缺陷修复。有关 7.4 主版本新特性的信息，请参阅第 E.31 节。

E.10.1. 迁移到版本 7.4.21

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.10.2. 变更

- 让 `pg_get_ruledef()` 给负常数加括号（Tom）

  在此修复之前，视图或规则中的负常数可能被转储成如 `-42::integer`，这有微妙的错误：由于操作符优先级规则，它应该是 `(-42)::integer`。通常这影响不大，但它可能与另一个最

近的补丁相互作用，导致 PostgreSQL 拒绝原本有效的 `SELECT DISTINCT` 视图查询。由于这可能导致 pg_dump 输出无法重载，它被当作高优先级修复。转储输出确实不正确的已发布版本只有 8.3.1 和 8.2.7。

E.11. 版本 7.4.20

发行日期

从未发布

本次发布包含对 7.4.19 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.11.1. 迁移到版本 7.4.20

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.11.2. 变更

- 修复 ISO-8859-5 与其他编码之间的转换对西里尔“Yo”字符（带两个点的 `e` 和 `E`）的处理（Sergey Burladyan）
- 修复几个数据类型输入函数，
  它们允许结果中未使用的字节包含未初始化的不可预测值（Tom）

  这可能导致两个看起来相同的字面值不被视为相等的失败，导致解析器抱怨 `ORDER BY` 和 `DISTINCT` 表达式不匹配。
- 修复正则表达式子串匹配（`substring(`*`string`*` from `*`pattern`*`)`）中的一个极端情形（Tom）

  问题发生在模式整体有匹配但用户指定了带括号的子表达式而该子表达式没有匹配时。例子是 `substring('foo' from 'foo(bar)?')`。由于 `(bar)` 未匹配，它应返回 NULL，但它错误地返回了整个模式的匹配（即 `foo`）。
- 修复 ecpg 的 `PGTYPEStimestamp_sub()` 函数结果不正确的问题（Michael）
- 修复 `DatumGetBool` 宏，使其在 gcc 4.3 下不失败（Tom）

  此问题影响返回布尔的“旧式”（V0）C 函数。修复已在 8.3 中，但当时没有意识到需要向后移植。
- 修复长期存在的 `LISTEN/NOTIFY` 竞态条件（Tom）

  极少数情况下，刚执行过 `LISTEN` 的会话可能收不到通知，即使按理应当收到，因为观察到执行 `NOTIFY` 的并发事务稍后提交了。

  此修复的一个副作用是：执行了尚未提交的 `LISTEN` 命令的事务如果去查看，将看不到该 `LISTEN` 在 `pg_listener` 中的行；以前能看到。这一行为从未被明确记录，但有些应用程序可能依赖旧行为。
- 修复 `ORDER BY` 和 `GROUP BY` 中常量表达式的显示（Tom）

显式转换的常量会被错误显示。这例如可能导致视图定义在转储和重载期间损坏。

- 修复 libpq，使其在 COPY OUT 期间正确处理 NOTICE 消息（Tom）

  只观察到此失败在用户定义数据类型的输出例程发出 NOTICE 时发生，但不能保证它不会因其他原因发生。

E.12. 版本 7.4.19

发行日期
2008-01-07

此发行版包含 7.4.18 之后的多种修复，包括对重要安全问题的修复。有关 7.4 主版本新特性的信息，请参阅第 E.31 节。

E.12.1. 迁移到版本 7.4.19

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.12.2. 变更

- 防止索引中的函数以运行 `VACUUM`、`ANALYZE` 等的用户的权限执行（Tom）

  每当产生新的表条目时，都会求值索引表达式和部分索引谓词中使用的函数。人们早就明白，如果修改不可信用户拥有的表，这会带来特洛伊木马代码执行的风险。（注意触发器、默认值、检查约束等也构成同类风险。）但索引中的函数带来额外危险，因为它们会由例程维护操作（如 `VACUUM FULL`）执行，而这些操作通常在超级用户账户下自动执行。例如，恶意用户可以通过设置特洛伊木马索引定义并等待下一次例程清理来以超级用户权限执行代码。修复安排标准维护操作（包括 `VACUUM`、`ANALYZE`、`REINDEX` 和 `CLUSTER`）以表所有者而非调用用户的身份执行，使用已用于 `SECURITY DEFINER` 函数的同一权限切换机制。为防止绕过此安全措施，现在禁止在 `SECURITY DEFINER` 上下文中执行 `SET SESSION AUTHORIZATION` 和 `SET ROLE`。（CVE-2007-6600）

- 修复正则表达式包中的若干缺陷（Tom、Will Drewry）

  精心构造的正则表达式模式可能导致崩溃、无限或接近无限的循环和/或大量内存消耗，这些都对接受来自不可信来源的正则搜索模式的应用程序构成拒绝服务风险。（CVE-2007-4769、CVE-2007-4772、CVE-2007-6067）

- 作为安全措施，要求使用 `/contrib/dblink` 的非超级用户只使用口令认证（Joe）

  7.4.18 中出现的修复并不完整，它只堵住了部分 `dblink` 函数的漏洞。（CVE-2007-6601、CVE-2007-3278）

- 修复某些 `WHERE false AND var IN (SELECT ...)` 情形下的规划器失败（Tom）

- 修复使用多字节数据库编码时 `translate()` 可能的崩溃（Tom）

- 修复 PL/Python 在长异常消息上不崩溃（Alvaro）
- ecpg 解析器修复（Michael）
- 让 `contrib/tablefunc` 的 `crosstab()` 把 NULL rowid 本身作为一个类别处理，而不是崩溃（Joe）
- 修复 `tsvector` 和 `tsquery` 输出例程以正确转义反斜杠（Teodor、Bruce）
- 修复 `to_tsvector()` 在超长输入字符串上的崩溃（Teodor）
- 要求在重新生成 `configure` 脚本时使用特定版本的 Autoconf（Peter）

  这只影响开发者和打包者。此变更是为了防止意外使用未经测试的 Autoconf 与 PostgreSQL 版本组合。如果确实想使用不同的 Autoconf 版本，可以移除版本检查，但结果是否可用由你自己负责。

E.13. 版本 7.4.18

发行日期
2007-09-17

此发行版包含 7.4.17 之后的修复。有关 7.4 主版本新特性的信息，请参阅第 E.31 节。

E.13.1. 迁移到版本 7.4.18

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.13.2. 变更

- 防止事务插入行后在同一表上并发 `VACUUM` 接近结束时中止导致的索引损坏（Tom）
- 让 `CREATE DOMAIN ... DEFAULT NULL` 正确工作（Tom）
- 修复 SSL 错误消息的过度记录（Tom）
- 修复 `log_min_error_statement` 日志记录内存耗尽时的崩溃（Tom）
- 防止 `CLUSTER` 因试图处理其他会话的临时表而失败（Alvaro）
- 作为安全措施，要求使用 `/contrib/dblink` 的非超级用户只使用口令认证（Joe）

E.14. 版本 7.4.17

发行日期
2007-04-23

此发行版包含 7.4.16 之后的修复，包括一个安全修复。有关 7.4 主版本新特性的信息，请参阅第 E.31 节。

E.14.1. 迁移到版本 7.4.17

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.14.2. 变更

- 支持在 `search_path` 中显式放置临时表模式，并禁用为函数和操作符搜索它（Tom）

  这是允许安全定义者函数设置真正安全的 `search_path` 值所必需的。没有它，非特权 SQL 用户可以利用临时对象以安全定义者函数的权限执行代码（CVE-2007-2138）。更多信息请参阅 `CREATE FUNCTION`。

- `/contrib/tsearch2` 崩溃修复（Teodor）

- 修复 `VACUUM FULL` 处理 `UPDATE` 链方式中的潜在数据损坏缺陷（Tom、Pavan Deolasee）

- 修复哈希索引扩容期间的 PANIC（7.4.15 中引入的缺陷）（Tom）

E.15. 版本 7.4.16

发行日期
2007-02-05

此发行版包含 7.4.15 之后的多种修复，包括一个安全修复。有关 7.4 主版本新特性的信息，请参阅第 E.31 节。

E.15.1. 迁移到版本 7.4.16

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.15.2. 变更

- 移除允许已连接用户读取后端内存的安全漏洞（Tom）

  该漏洞涉及抑制 SQL 函数返回其声明数据类型的常规检查，或更改 SQL 函数中使用的表列的数据类型（CVE-2007-0555）。此错误很容易被利用来导致后端崩溃，
  而且原则上可能被用来读取用户不应能访问的数据库内容。

- 修复罕见的缺陷：btree 索引页分裂可能因选择不可行的分裂点而失败（Heikki Linnakangas）

- 修复由 `UNION` 触发的罕见 Assert() 崩溃（Tom）

- 收紧对超过三字节的 UTF8 序列的多字节字符处理的安全性（Tom）

E.16. 版本 7.4.15

发行日期
2007-01-08

本次发布包含对 7.4.14 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.16.1. 迁移到版本 7.4.15

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.16.2. 变更

- 改进 AIX 上 `getaddrinfo()` 的处理（Tom）

  这修复了启动统计收集器等问题。

- 修复 `VACUUM` 中的“未能重新找到父键”错误（Tom）
- 修复影响多 GB 哈希索引的缺陷（Tom）
- 修复构造由多个空元素组成的 `ARRAY[]` 时的错误（Tom）
- 对于新的 initdb 安装，`to_number()` 和 `to_char(numeric)` 现在是 `STABLE` 而不是 `IMMUTABLE`（Tom）

  这是因为 `lc_numeric` 可能改变这些函数的输出。

- 改进使用括号的正则表达式的索引利用（Tom）

  这同时也改进了 psql 的 `\d` 性能。

E.17. 版本 7.4.14

发行日期
2006-10-16

本次发布包含对 7.4.13 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.17.1. 迁移到版本 7.4.14

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.17.2. 变更

- 修复把无类型字面值当作 ANYARRAY 时的核心转储

- 修复 `string_to_array()`，处理分隔符字符串的重叠匹配

  例如 `string_to_array('123xx456xxx789', 'xx')`。

- 修复 psql 的 `\d` 命令模式匹配的极端情形

- 修复 /contrib/ltree 中损坏索引的缺陷（Teodor）

- 修复 /contrib/dbmirror 中的反斜杠转义

- 针对美国 DST 法规的最近变更调整回归测试

E.18. 版本 7.4.13

发行日期
2006-05-23

此发行版包含 7.4.12 之后的多种修复，包括针对极其严重安全问题的补丁。有关 7.4 主版本新特性的信息，请参阅第 E.31 节。

E.18.1. 迁移到版本 7.4.13

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

要完全防范 CVE-2006-2313 和 CVE-2006-2314 所述的 SQL 注入攻击，可能需要更改应用程序代码。如果你有把不可信字符串嵌入 SQL 命令的应用程序，应尽快检查它们，确保使用了推荐的转义技术。在大多数情况下，应用程序应使用库或驱动提供的子例程（如 libpq 的 `PQescapeStringConn()`）来执行字符串转义，而不是依靠临时代码来做。

E.18.2. 变更

- 更改服务器，在所有情形下拒绝无效编码的多字节字符（Tatsuo、Tom）

  虽然 PostgreSQL 向这个方向发展已有一段时间，但现在检查统一应用于所有编码和所有文本输入，而且总是错误而不只是警告。此变更可防范 CVE-2006-2313 所述类型的 SQL 注入攻击。

- 拒绝字符串字面量中对 `\'` 的不安全使用

  作为针对 CVE-2006-2314 所述类型 SQL 注入攻击的服务器端防御，服务器现在只接受 `''` 而不接受 `\'` 作为 SQL 字符串字面量中 ASCII 单引号的表示。默认情况下，只有当 `client_encoding` 设为仅客户端编码（SJIS、BIG5、GBK、GB18030 或 UHC）时才拒绝 `\'`，这正是可能发生 SQL 注入的场景。需要时可用新的配置参数 `backslash_quote` 调整此行为。注意，要完全防范 CVE-2006-2314 可能需要客户端更改；`backslash_quote` 的目的之一是让不安全的客户端明显地不安全。

- 修改 libpq 的字符串转义例程，使其感知编码问题和 `standard_conforming_strings`

  这为使用 libpq 的应用程序修复了 CVE-2006-2313 和 CVE-2006-2314 所述的安全问题，并使它们为将来转向 SQL 标准字符串字面量语法做好准备。并发使用多个 PostgreSQL 连接的应用

程序应迁移到 `PQescapeStringConn()` 和 `PQescapeByteaConn()`，以确保为每个数据库连接中使用的设置正确转义。以“手工”方式做字符串转义的应用程序应改为依靠库例程。

- 修复一些不正确的编码转换函数

  `win1251_to_iso`、`alt_to_iso`、`euc_tw_to_big5`、`euc_tw_to_mic`、`mic_to_euc_tw` 都在不同程度上损坏。

- 清理字符串中偶然残留的 `\'` 使用（Bruce、Jan）
- 修复有时导致 OR 索引扫描漏掉本应返回的行的缺陷
- 修复 btree 索引被截断情形的 WAL 重放
- 修复涉及 `|` 的模式的 `SIMILAR TO`（Tom）
- 修复服务器正确使用自定义 DH SSL 参数（Michael Fuhr）
- 修复 Intel Mac 上的 Bonjour（Ashley Clark）
- 修复各种小的内存泄漏

E.19. 版本 7.4.12

发行日期
2006-02-14

本次发布包含对 7.4.11 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.19.1. 迁移到版本 7.4.12

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.11 的版本升级，请参阅 7.4.11 的发行说明。

E.19.2. 变更

- 修复 `SET SESSION AUTHORIZATION` 中可能的崩溃（CVE-2006-0553）

  如果服务器编译时启用了断言（这不是默认），非特权用户可以使服务器进程崩溃，导致对其他用户的瞬时拒绝服务。感谢 Akio Ishida 报告此问题。

- 修复自插入行的行可见性逻辑缺陷（Tom）

  极少数情况下，当前命令插入的行可能被看作已经有效，而不应该如此。修复 7.4.9 和 7.3.11 版本中引入的缺陷。

- 修复 pg_clog 文件创建期间可能导致“文件已存在”错误的竞态条件（Tom）
- 为预备语句中的 `UNKNOWN` 参数正确检查 `DOMAIN` 约束（Neil）
- 修复以允许恢复有对自定义操作符跨模式引用的转储（Tom）
- 修复 configure 期间测试 `finite` 和 `isinf` 存在性的可移植性问题（Tom）

E.20. 版本 7.4.11

发行日期
2006-01-09

本次发布包含对 7.4.10 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.20.1. 迁移到版本 7.4.11

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.8 的版本升级，请参阅 7.4.8 的发行说明。另外，如果你受到下述区域设置或 plperl 问题的影响，可能需要在更新后对文本列上的索引执行 `REINDEX`。

E.20.2. 变更

- 修复在事务之外或失败事务中发出协议级 Describe 消息的问题（Tom）
- 修复把不同字符组合视为相等的区域设置（如匈牙利语）下的字符串比较（Tom）

  这可能需要 `REINDEX` 来修复文本列上的现有索引。
- 在 postmaster 启动期间设置区域设置环境变量，以确保 plperl 之后不会更改区域设置

  这修复了 postmaster 启动时环境变量指定的区域设置与 initdb 所被告知的不同时发生的问题。在这些条件下，任何 plperl 的使用都可能导致索引损坏。如果你遇到过这种情况，可能需要 `REINDEX` 来修复文本列上的现有索引。
- 修复某些很少使用的亚洲多字节字符集中 strpos() 和正则表达式处理的长期缺陷（Tatsuo）
- 修复 `/contrib/pgcrypto` gen_salt 中的缺陷，该缺陷使 MD5 和 XDES 算法不能使用全部可用盐空间（Marko Kreen、Solar Designer）

  Blowfish 和标准 DES 的盐不受影响。
- 修复 `/contrib/dblink`，当指定的列数与查询实际返回的列数不同时抛出错误而不是崩溃（Joe）

E.21. 版本 7.4.10

发行日期
2005-12-12

本次发布包含对 7.4.9 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.21.1. 迁移到版本 7.4.10

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.8 的版本升级，请参阅 7.4.8 的发行说明。

E.21.2. 变更

- 修复事务日志管理中的竞态条件

  存在一个窄窗口，其中可能为错误的页发起 I/O 操作，导致断言失败或数据损坏。

- 防止客户端在当前事务已中止时发送 Bind 协议消息导致的失败
- `/contrib/ltree` 修复（Teodor）
- AIX 和 HPUX 编译修复（Tom）
- 修复外连接的长期规划错误

  此缺陷有时导致虚假错误“RIGHT JOIN 只支持可归并连接的连接条件”。

- 防止表被删除后 pg_autovacuum 的核心转储

E.22. 版本 7.4.9

发行日期
2005-10-04

本次发布包含对 7.4.8 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.22.1. 迁移到版本 7.4.9

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.4.8 的版本升级，请参阅 7.4.8 的发行说明。

E.22.2. 变更

- 修复允许 `VACUUM` 过早移除 `ctid` 链的错误，并在跟随 `ctid` 链接的代码中添加更多检查

  这修复了一个在极罕见情况下可能导致崩溃的长期问题。

- 修复 `CHAR()` 在使用多字节字符集时正确填充空格到指定长度（Yoshiyuki Asaba）

  在以前的版本中，`CHAR()` 的填充是不正确的，因为它只填充到指定的字节数而不考虑存储了多少字符。

- 修正 `COPY` 中只读事务测试的方向

  以前的代码禁止 `COPY TO`，而它应禁止的是 `COPY FROM`。

- 修复只引用内侧关系的外连接 ON 子句的规划问题
- 对 `x FULL JOIN y ON true` 极端情形的进一步修复
- 让 `array_in` 和 `array_recv` 更严格地验证其 OID 参数
- 修复列 `a` 上有 GiST 索引时 `UPDATE a=... WHERE a...` 之类查询丢行的问题

- 改进日期时间解析的健壮性
- 改进对部分写入 WAL 页的检查
- 改进启用 SSL 时信号处理的健壮性
- postmaster 启动期间不要试图打开超过 `max_files_per_process` 个文件
- 各种内存泄漏修复
- 各种可移植性改进
- 修复 PL/pgSQL 在变量为按引用传递类型时正确处理 `var := var`
- 更新 `contrib/tsearch2` 以使用当前的 Snowball 代码

E.23. 版本 7.4.8

发行日期
2005-05-09

此发行版包含 7.4.7 之后的多种修复，包括几个与安全相关的问题。有关 7.4 主版本新特性的信息，请参阅第 E.31 节。

E.23.1. 迁移到版本 7.4.8

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，它是处理在 7.4.X 系统目录初始内容中发现的两个重要安全问题的可能方法之一。用 7.4.8 的 initdb 做转储/initdb/重载序列会自动纠正这些问题。

较大的安全问题是内建字符集编码转换函数可以由非特权用户从 SQL 命令调用，但这些函数并非为此设计，对恶意选择的参数不安全。修复包括更改这些函数的声明参数列表，使其不能再从 SQL 命令调用。（这不影响编码转换机制对它们的正常使用。）

较小的问题是 `contrib/tsearch2` 模块创建了几个被错误声明为返回 `internal` 的函数，而它们并不接受 `internal` 参数。这破坏了所有使用 `internal` 参数的函数的类型安全。

强烈建议所有安装通过 initdb 或按照下述手工修复程序修复这些错误。这些错误至少允许非特权数据库用户使其服务器进程崩溃，而且可能允许非特权用户获得数据库超级用户的权限。

如果你不想做 initdb，请执行以下程序。作为数据库超级用户，执行：

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
     AND proargtypes[2] = 'cstring'::regtype;
-- The command should report having updated 90 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

然后，如果你安装了 `contrib/tsearch2`，执行：

```
BEGIN;
UPDATE pg_proc SET proargtypes[0] = 'internal'::regtype
WHERE oid IN (
   'dex_init(text)'::regprocedure,
   'snb_en_init(text)'::regprocedure,
   'snb_ru_init(text)'::regprocedure,
   'spell_init(text)'::regprocedure,
   'syn_init(text)'::regprocedure
);
-- The command should report having updated 5 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

如果此命令失败并显示类似 “function "dex_init(text)" does not exist” 的消息，则要么该数据库中未安装 `tsearch2`，要么你已经做过此更新。

上述程序必须在安装的每个数据库中执行，包括 `template1`，理想情况下也包括 `template0`。如果不修复模板数据库，之后创建的任何数据库都会包含相同的错误。`template1` 可以像任何其他数据库一样修复，但修复 `template0` 需要额外的步骤。首先，从任意数据库发出：

```
UPDATE pg_database SET datallowconn = true WHERE datname =
 'template0';
```

接下来连接到 `template0` 并执行上述修复程序。最后执行：

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname =
 'template0';
```

E.23.2. 变更

- 更改编码函数签名以防止误用
- 更改 `contrib/tsearch2` 以避免不安全地使用 `INTERNAL` 函数结果
- 修复古老的竞态条件，它允许事务在某些用途（如 SELECT FOR UPDATE）上比其他用途稍早被视为已提交

  这是一个极其严重的缺陷，因为它可能导致应用程序短暂看到表面的数据不一致。
- 修复关系扩展与 VACUUM 之间的竞态条件

  这在理论上可能导致丢失一页刚插入的数据，尽管该情形概率似乎很低。
  没有已知案例造成超过断言失败的后果。
- 修复 `TIME WITH TIME ZONE` 值的比较

  在使用了 `--enable-integer-datetimes` 配置开关的情形下比较代码是错误的。注意：如果你在 `TIME WITH TIME ZONE` 列上有索引，安装此更新后需要对它执行 `REINDEX`，因为该修复纠正了列值的排序顺序。
- 修复 `TIME WITH TIME ZONE` 值的 `EXTRACT(EPOCH)`
- 修复 `INTERVAL` 值负小数秒的错误显示

此错误只在使用了 `--enable-integer-datetimes` 配置开关时发生。

- 确保后端关闭期间完成的操作被统计收集器计数

  这有望解决关于 pg_autovacuum 不够频繁地清理系统目录的报告——它没有被通知后端退出期间临时表删除导致的目录删除。

- plpgsql 中的额外缓冲区溢出检查（Neil）
- 修复 pg_dump，正确转储包含 `%` 的触发器名（Neil）
- 修复 `contrib/pgcrypto` 在较新 OpenSSL 构建上的问题（Marko Kreen）
- `contrib/intagg` 的更多 64 位修复
- 防止对返回 `RECORD` 的函数的错误优化
- 防止 `to_char(interval)` 在月份相关格式上转储核心
- 防止 `COALESCE(NULL,NULL)` 的崩溃
- 修复 `array_map`，使其正确调用 PL 函数
- 修复 `ALTER DATABASE RENAME` 中的权限检查
- 修复 `ALTER LANGUAGE RENAME`
- 让 `RemoveFromWaitQueue` 清理自身

  这修复了一个锁管理错误，它只在事务被踢出锁等待（通常由查询取消）且锁持有者随后在极窄的窗口内释放锁时可见。

- 修复 `INSERT ... SELECT` 中出现无类型参数的问题
- 修复 `ALTER TABLE SET WITHOUT OIDS` 之后 `CLUSTER` 的失败

E.24. 版本 7.4.7

发行日期
2005-01-31

此发行版包含 7.4.6 之后的多种修复，包括几个与安全相关的问题。有关 7.4 主版本新特性的信息，请参阅第 E.31 节。

E.24.1. 迁移到版本 7.4.7

对于运行 7.4.X 的用户，不需要进行转储/恢复。

E.24.2. 变更

- 不允许非超级用户使用 `LOAD`

在会自动执行共享库初始化函数的平台（至少包括 Windows 和基于 ELF 的 Unix）上，`LOAD` 可被用来让服务器执行任意代码。感谢 NGS Software 报告此问题。

- 检查聚合函数的创建者是否有权执行指定的转换函数

  这一疏漏使得绕过函数 EXECUTE 权限的拒绝成为可能。

- 修复 contrib/intagg 的安全和 64 位问题
- 为一些 contrib 函数添加所需的 STRICT 标记（Kris Jurka）
- 避免 plpgsql 游标声明参数过多时的缓冲区溢出（Neil）
- 修复 FULL 和 RIGHT 外连接的规划错误

  连接的结果被错误地认为与左输入的排序相同。这不仅会向用户交付排序错误的输出，而且在嵌套归并连接的情形下可能给出完全错误的答案。

- 修复 plperl 处理元组字段中引号的问题
- 修复 SQL 和 GERMAN datestyle 下负 interval 的显示
- 让 age(timestamptz) 用本地时区而不是 GMT 计算

E.25. 版本 7.4.6

发行日期
2004-10-22

本次发布包含对 7.4.5 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.25.1. 迁移到版本 7.4.6

对于运行 7.4.X 的用户，不需要进行转储/恢复。

E.25.2. 变更

- 修复磁盘上提示位可能未更新的问题

  极少数情况下，这一疏漏可能导致“无法访问事务状态”失败，这使它成为一个潜在的数据丢失缺陷。

- 确保哈希外连接不漏掉元组

  在恰当的数据分布下，使用哈希连接计划的超大型左连接可能无法输出未匹配的左侧行。

- 不允许以 root 运行 pg_ctl

  这是为了防范任何可能的安全问题。

- 避免 `make_oidjoins_check` 在 `/tmp` 中使用临时文件

这被报告为安全问题，不过几乎不值得担心，因为非开发者本来就没有理由使用此脚本。

- 防止强制后端关闭重新发出先前的命令结果

  极少数情况下，客户端可能认为其最后一条命令成功了，
  而它实际上已被强制数据库关闭中止。

- 修复 `pg_stat_get_backend_idset` 中的缺陷

  这可能导致某些系统统计视图行为异常。

- 修复 postmaster 中的小内存泄漏

- 修复“预期两个交换的表都有 TOAST 表”缺陷

  这可能发生在如 ALTER TABLE DROP COLUMN 之后再 CLUSTER 的情形。

- 防止 `pg_ctl restart` 多次添加 `-D`

- 修复 GiST 索引中 NULL 值的问题

- `::` 不再被解释为 ECPG prepare 语句中的变量

E.26. 版本 7.4.5

发行日期
2004-08-18

此发行版在 7.4.4 的基础上包含一个严重的缺陷修复。有关 7.4 主版本新特性的信息，请参阅第 E.31 节。

E.26.1. 迁移到版本 7.4.5

对于运行 7.4.X 的用户，不需要进行转储/恢复。

E.26.2. 变更

- 修复并发 B-tree 索引插入期间可能的崩溃

  此补丁修复了并发插入 B-tree 索引可能导致服务器 panic 的罕见情形。不会造成永久损坏，但仍值得重新发布。该缺陷在 7.4 之前的版本中不存在。

E.27. 版本 7.4.4

发行日期
2004-08-16

本次发布包含对 7.4.3 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.27.1. 迁移到版本 7.4.4

对于运行 7.4.X 的用户，不需要进行转储/恢复。

E.27.2. 变更

- 防止崩溃期间已提交事务的可能丢失

  由于事务提交与检查点之间的互锁不足，
  最近一次检查点之前刚提交的事务可能在数据库崩溃和重启后全部或部分丢失。这是自 PostgreSQL 7.1 以来就存在的严重缺陷。

- 在求值聚合计划的结果列表之前检查 HAVING 限制
- 避免会话的当前用户 ID 被删除时的崩溃
- 修复零行情形的哈希 crosstab（Joe）
- 在外键中重命名列后强制更新缓存
- 正确美化 UNION 查询的打印
- 让 psql 在 COPY IN 中正确处理 `\r\n` 换行
- pg_dump 对带 grant 选项的 ACL 处理不正确
- 修复 OS X 和 Solaris 的线程支持
- 更新了 JDBC 驱动（build 215），包含各种修复
- ECPG 修复
- 翻译更新（多位贡献者）

E.28. 版本 7.4.3

发行日期
2004-06-14

本次发布包含对 7.4.2 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.28.1. 迁移到版本 7.4.3

对于运行 7.4.X 的用户，不需要进行转储/恢复。

E.28.2. 变更

- 修复使用非哈希聚合时的临时内存泄漏（Tom）

- ECPG 修复，包括一些针对 Informix 兼容性的（Michael）
- 修复线程安全编译的问题，特别是 Solaris（Bruce）
- 修复使用旧网络协议时 COPY IN 结束时的错误（ljb）
- pg_autovacuum 的几个重要修复，包括大表、无符号 oid、稳定性、临时表和调试模式的修复（Matthew T. O'Connor）
- 修复在 NetBSD 和 BSD/OS 上读取 tar 格式转储的问题（Bruce）
- 若干 JDBC 修复
- 修复 last_value 等于重启值时的 ALTER SEQUENCE RESTART（Tom）
- 修复嵌套子查询未重新计算的问题（Tom）
- 修复 LIMIT/OFFSET 中非常量表达式的问题
- 支持无连接子句的 FULL JOIN，如 X FULL JOIN Y ON TRUE（Tom）
- 修复另一个零列表缺陷（Tom）
- 改进子查询中 GROUP BY 子句里非限定标识符的处理（Tom）

  子查询内的选择列表别名现在优先于来自外层查询级别的名称。
- 反编译规则时不生成“NATURAL CROSS JOIN”（Tom）
- 为二进制 COPY 中的无效字段长度添加检查（Tom）

  这修复了一个难以利用的安全漏洞。
- 避免 `ANALYZE` 与 `LISTEN/NOTIFY` 之间的锁定冲突
- 大量翻译更新（多位贡献者）

E.29. 版本 7.4.2

发行日期
2004-03-08

本次发布包含对 7.4.1 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.29.1. 迁移到版本 7.4.2

对于运行 7.4.X 的用户，不需要进行转储/恢复。不过，作为合并 7.4.X 系统目录初始内容中发现的两个错误的修复的最简单方法，它可能是可取的。用 7.4.2 的 initdb 做转储/initdb/重载序列会自动纠正这些问题。

两个错误中较严重的是数据类型 `anyarray` 的对齐标签错误；这是一个问题，因为 `pg_statistic` 系统目录使用 `anyarray` 列。当规划涉及双对齐列（如 `float8` 和 `timestamp`）

上 WHERE 子句的查询时，错误标记可能导致规划器误估甚至崩溃。强烈建议所有安装通过 initdb 或按照下述手工修复程序修复此错误。

较小的错误是系统视图 pg_settings 应被标记为具有公共更新访问权，以允许用 UPDATE pg_settings 替代 SET。这也可以通过 initdb 或手工修复，但除非你想用 UPDATE pg_settings，否则不必修复。

如果你不想做 initdb，以下程序可以修复 pg_statistic。作为数据库超级用户，执行：

```
-- clear out old data in pg_statistic:
DELETE FROM pg_statistic;
VACUUM pg_statistic;
-- this should update 1 row:
UPDATE pg_type SET typalign = 'd' WHERE oid = 2277;
-- this should update 6 rows:
UPDATE pg_attribute SET attalign = 'd' WHERE atttypid = 2277;
--
-- At this point you MUST start a fresh backend to avoid a crash!
--
-- repopulate pg_statistic:
ANALYZE;
```

这可以在运行中的数据库上完成，但要注意在重新填充 pg_statistic 之前，被更改数据库中运行的所有后端都必须重启才是安全的。

要修复 pg_settings 错误，只需执行：

```
GRANT SELECT, UPDATE ON pg_settings TO PUBLIC;
```

上述程序必须在安装的每个数据库中执行，包括 template1，理想情况下也包括 template0。如果不修复模板数据库，之后创建的任何数据库都会包含相同的错误。template1 可以像任何其他数据库一样修复，但修复 template0 需要额外的步骤。首先，从任意数据库发出：

```
UPDATE pg_database SET datallowconn = true WHERE datname =
 'template0';
```

接下来连接到 template0 并执行上述修复程序。最后执行：

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname =
 'template0';
```

E.29.2. 变更

7.4.2 版本包含 7.3.6 版本中的所有修复，外加以下修复：

- 修复可能导致优化器崩溃的 pg_statistics 对齐缺陷

  有关此问题的详情见上文。

- 允许非超级用户更新 pg_settings

- 修复几个优化器缺陷，其中大多数导致“在子计划目标列表中未找到变量”错误

- 避免大型多索引扫描启动期间的内存不足失败
- 修复可能导致 `COPY IN` 期间“内存不足”错误的多字节问题
- 修复从无 OID 的表 `SELECT INTO` / `CREATE TABLE AS` 的问题
- 修复并行测试期间 `alter_table` 回归测试的问题
- 修复达到打开文件限制的问题，特别是在 OS X 上（Tom）
- 对土耳其语区域设置问题的部分修复

  initdb 现在能在土耳其语区域设置下成功，但仍有一些与 `i/I` 问题相关的不便。
- 让 pg_dump 在恢复时设置客户端编码
- 其他次要的 pg_dump 修复
- 允许 ecpg 再次使用 C 关键字作为列名（Michael）
- 为 ecpg 的 `SELECT/INSERT/UPDATE/DELETE` 添加 `WHENEVER NOT_FOUND`（Michael）
- 修复调用返回集合函数的查询导致的 ecpg 崩溃（Michael）
- 各种其他 ecpg 修复（Michael）
- Borland 编译器的修复
- 线程构建改进（Bruce）
- 各种其他构建修复
- 各种 JDBC 修复

E.30. 版本 7.4.1

发行日期
2003-12-22

本次发布包含对 7.4 的多方面修复。有关 7.4 大版本中新特性的信息，请参见第 E.31 节。

E.30.1. 迁移到版本 7.4.1

对于运行 7.4 的用户，不需要进行转储/恢复。

如果你想在信息模式中安装这些修复，需要把它重新装载到数据库中。这可以通过运行 `initdb` 初始化新集群来完成，或者在安装新版本后，以超级用户身份在 psql 中对每个数据库（理想情况下包括 `template1`）运行以下 SQL 命令序列：

```
DROP SCHEMA information_schema CASCADE;
\i /usr/local/pgsql/share/information_schema.sql
```

在第二条命令中替换你的安装路径。

E.30.2. 变更

- 修复 ECPG 中 `CREATE SCHEMA` 解析的缺陷（Michael）
- 修复 `--enable-thread-safety` 与 `--with-perl` 一起使用时的编译错误（Peter）
- 修复使用哈希连接的子查询（Tom）

  某些使用哈希连接的子查询会因不当共享的结构而崩溃。
- 修复空闲空间映射压缩缺陷（Tom）

  这修复了空闲空间映射的压缩可能导致数据库服务器关闭的缺陷。
- 修复 libpq 的 Borland 编译器构建（Bruce）
- 修复 `netmask()` 和 `hostmask()`，使其返回最大长度的掩码长度（Tom）

  修复这些函数使其返回与 7.4 之前版本一致的值。
- 若干 `contrib/pg_autovacuum` 修复

  修复包括不当的变量初始化、`TRUNCATE` 后缺失的清理以及长清理的时长计算溢出。
- 允许 `contrib/cube` 在 Cygwin 下编译（Jason Tishler）
- 修复 Solaris 在未定义口令时使用口令文件的问题（Tom）

  修复 Solaris 上因在未定义口令时使用任何类型的口令认证导致的崩溃。
- JDBC 线程问题修复及其他修复
- 修复 `bytea` 索引查找（Joe）
- 修复信息模式对位数据类型的支持（Peter）
- 强制在从 WAL 恢复期间开启 zero_damaged_pages
- 防止“变量不在子计划目标列表中”的一些晦涩情形
- 使 `PQescapeBytea` 与 `byteaout` 相互一致（Joe）
- 对大于 0x7e 的字节转义 `bytea` 输出（Joe）

  如果 `bytea` 输出和输入使用不同的客户端编码，`bytea` 值可能被不同的编码破坏。此修复转义了所有可能受影响的字节。
- 为 dblink 的 `get_tuple_of_interest()` 添加缺失的 `SPI_finish()` 调用（Joe）
- 新的捷克语 FAQ
- 修复信息模式视图 `constraint_column_usage` 对外键的支持（Peter）
- ECPG 修复（Michael）
- 修复多个 `IN` 子查询及子查询内连接的缺陷（Tom）
- 允许 `COUNT('x')` 工作（Tom）

- 把 Informix 兼容的 ECPG 头文件安装到单独的目录（Peter）

  一些 Informix 兼容的 ECPG 头文件名与操作系统头文件冲突。通过把它们安装在自己的目录中，减少了名称冲突。

- 修复 SSL 内存泄漏（Neil）

  此发行版修复了 7.4 中 SSL 未释放其分配的全部内存的缺陷。

- 防止 `pg_service.conf` 把服务名用作默认数据库名（Bruce）
- 修复 FreeBSD 上的本地 ident 认证（Tom）

E.31. 版本 7.4

发行日期
2003-11-17

E.31.1. 概述

此发行版的主要变更：

`IN` / `NOT IN` 子查询现在效率高得多

在以前的版本中，`IN`/`NOT IN` 子查询通过顺序扫描子查询寻找匹配来与上层查询连接。7.4 的代码使用与普通连接相同的复杂技术，因此快得多。`IN` 现在通常与等价的 `EXISTS` 子查询一样快或更快；这颠倒了适用于以前版本的传统观念。

通过使用哈希桶改进了 `GROUP BY` 处理

在以前的版本中，要分组的行必须先排序。7.4 的代码可以不做排序就完成 `GROUP BY`，方法是把结果累积到每组一个条目的哈希表中。但如果估计哈希表太大放不进 `sort_mem`，它仍会使用排序技术。

新的多键哈希连接能力

在以前的版本中，哈希连接只能在单键上发生。此版本允许多列哈希连接。

使用显式 `JOIN` 语法的查询现在得到更好的优化

以前的版本只按语法隐含的顺序求值使用显式 `JOIN` 语法的查询。7.4 允许对这些查询做完全优化，即优化器考虑所有可能的连接顺序并选择最高效的。
但外连接仍必须遵循声明的顺序。

更快、更强大的正则表达式代码

整个正则表达式模块已被 Henry Spencer 的新版本（最初为 Tcl 编写）取代。该代码大幅改进了性能并支持多种风格的正则表达式。

对简单 SQL 函数的函数内联

简单 SQL 函数现在可以通过把其 SQL 包含在主查询中来内联。这通过消除每次调用的开销改进了性能。这意味着简单 SQL 函数现在的行为像宏。

全面支持 IPv6 连接和 IPv6 地址数据类型

以前的版本只允许 IPv4 连接，IP 数据类型也只支持 IPv4 地址。此版本在这两方面都增加了完整的 IPv6 支持。

SSL 性能和可靠性的重大改进

几位非常熟悉 SSL API 的人彻底改造了我们的 SSL 代码，以改进 SSL 密钥协商和错误恢复。

让空闲空间映射高效地重用空索引页面，以及其他空闲空间管理改进

在以前的版本中，因行被删除而变空的 B-tree 索引页只能被索引值与该页上最初索引的行相似的行重用。在 7.4 中，`VACUUM` 记录空索引页并允许它们被未来的任何索引行重用。

SQL 标准的信息模式

信息模式提供了访问数据库中定义的模式对象信息的标准化且稳定的方式。

游标更贴近 SQL 标准

`FETCH` 和 `MOVE` 命令已被彻底改造，以更贴近 SQL 标准。

游标可以存在于事务之外

这些游标也称为可保持游标。

新的客户端-服务器协议

新协议增加了错误码、更多状态信息、更快的启动、对二进制数据传输的更好支持、与 SQL 命令分离的参数值、协议级的预备语句以及从 `COPY` 失败更干净的恢复。服务器和客户端仍支持较旧的协议。

libpq 和 ECPG 应用现在完全线程安全

虽然以前的 libpq 版本已支持线程，但此版本通过修复数据库连接启动期间使用的一些非线程安全代码改进了线程安全。必须使用 `configure` 选项 `--enable-thread-safety` 才能启用此特性。

新版本的全文索引

`contrib/tsearch2` 中提供了新的全文索引套件。

新的自动清理（autovacuum）工具

`contrib/autovacuum` 中的新 autovacuum 工具监控数据库统计表中的 `INSERT`/`UPDATE`/`DELETE` 活动，并在需要时自动清理表。

数组处理得到了改进并被移入服务器核心

数组的许多限制已被移除，数组的行为更像完全受支持的数据类型。

E.31.2. 迁移到版本 7.4

希望从任何以前的版本迁移数据的用户，需要使用 pg_dump 进行转储/恢复。

请注意以下不兼容之处：

- 服务器端的自动提交设置被移除，并在客户端应用程序和语言中重新实现。
  服务器端自动提交给想要控制自己自动提交行为的语言和应用程序带来了太多问题，
  因此自动提交从服务器中移除并酌情添加到各个客户端 API。
- 此版本中错误消息的措辞发生了实质变化。投入了大量努力使消息更一致、更面向用户。
  如果你的应用程序试图通过解析错误消息来检测不同的错误条件，
  强烈建议改用新的错误码机制。
- 使用显式 `JOIN` 语法的内连接可能行为不同，因为它们现在得到了更好的优化。
- 为清晰起见，许多服务器配置参数已被重命名，主要是与日志相关的那些。
- `FETCH 0` 或 `MOVE 0` 现在不做任何事情。在以前的版本中，`FETCH 0` 会获取所有剩余行，`MOVE 0` 会移到游标末尾。
- `FETCH` 和 `MOVE` 现在返回实际获取/移动的行数，如果位于游标开头/末尾则返回零。以前的版本会返回传给命令的行数，而不是实际获取或移动的行数。
- `COPY` 现在能处理使用回车或回车/换行行尾序列的文件。数据值中不再接受字面回车和换行；请改用 `\r` 和 `\n`。
- 从 `char(n)` 转换为 `varchar(n)` 或 `text` 时现在会修剪尾随空格。这本来就是大多数人一直期望的。
- 数据类型 `float(p)` 现在用二进制位而不是十进制位度量 *p*。新行为遵循 SQL 标准。
- 有歧义的日期值现在必须匹配 `datestyle` 设置指定的顺序。在以前的版本中，即使 `datestyle` 指定日在前，`10/20/03` 的日期规格也会被解释为十月的日期。7.4 将在日期规格对当前 `datestyle` 设置无效时抛出错误。
- 函数 `oidrand`、`oidsrand` 和 `userfntest` 已被移除。这些函数被认定不再有用。
- 指定随时间变化的日期/时间值的字符串字面量（如 `'now'` 或 `'today'`）在列默认表达式中不再按预期工作；它们现在使表创建时刻成为默认值，而不是插入时刻。应改用 `now()`、`current_timestamp` 或 `current_date` 等函数。

  在以前的版本中，有特殊代码使 `'now'` 之类的字符串在 `INSERT` 时而不是表创建时被解释，但这一变通没有覆盖所有情形。7.4 版本现在要求用 `now()` 或 `current_timestamp()` 等函数正确定义默认值。这些在所有情形下都能工作。
- 操作符名中不再允许美元符号（`$`）。它可以用作标识符中的非首字符。
  这是为了改进与其他数据库系统的兼容性，并避免参数占位符（`$n`）
  紧邻操作符书写时的语法问题。

E.31.3. 变更

下面你将找到 7.4 版本与上一个主版本之间变更的详细说明。

E.31.3.1. 服务器操作变更

- 允许 IPv6 服务器连接（Nigel Kukard、Johan Jordaan、Bruce、Tom、Kurt Roeckx、Andrew Dunstan）
- 修复 SSL 以干净地处理错误（Nathan Mueller）

  在以前的版本中，某些 SSL API 错误报告未被正确处理。此版本修复了这些问题。

- SSL 协议安全和性能改进（Sean Chittenden）

  SSL 密钥重协商过于频繁，导致 SSL 性能不佳。此外，初始密钥处理也得到了改进。

- 检测到死锁时打印锁信息（Tom）

  这使调试死锁情形更容易。

- 定期更新 `/tmp` 套接字的修改时间以避免它们被删除（Tom）

  这应有助于防止清理 `/tmp` 目录的管理脚本删除服务器套接字文件。

- 为 Mac OS X 启用 PAM（Aaron Hillegass）

- 使 B-tree 索引完全 WAL 安全（Tom）

  在以前的版本中，某些罕见情形下服务器崩溃可能导致 B-tree 索引损坏。此版本消除了最后几个罕见情形。

- 允许 B-tree 索引压缩和空页重用（Tom）

- 修复首个根页分裂期间索引查找不一致的问题（Tom）

  在以前的版本中，当单页索引分裂为两页时，
  有一小段时间另一个数据库会话可能看不到某个索引条目。此版本修复了这一罕见失败情形。

- 改进空闲空间映射的分配逻辑（Tom）

- 在服务器重启之间保留空闲空间信息（Tom）

  在以前的版本中，postmaster 停止时空闲空间映射不被保存，因此新启动的服务器没有空闲空间信息。此版本保存空闲空间映射，并在服务器重启时重新装载它。

- 为 `pg_stat_activity` 添加开始时间（Neil）

- 检测损坏磁盘页的新代码；用 `zero_damaged_pages` 清除（Tom）

- 新的客户端/服务器协议：更快、无用户名长度限制、允许从 `COPY` 干净退出（Tom）

- 为客户端/服务器协议添加事务状态、表 ID、列 ID（Tom）

- 为客户端/服务器协议添加二进制 I/O（Tom）

- 移除服务器端自动提交设置；移至客户端应用程序（Tom）

- 新的错误消息措辞、错误码和三级错误细节（Tom、Joe、Peter）

E.31.3.2. 性能改进

- 为 `GROUP BY` 聚合添加哈希（Tom）

- 使嵌套循环连接更聪明地使用多列索引（Tom）

- 允许多键哈希连接（Tom）

- 改进常量折叠（Tom）

- 新增内联简单 SQL 函数的能力（Tom）

- 降低使用复杂函数的查询的内存使用（Tom）

  在以前的版本中，返回已分配内存的函数直到查询完成才释放它。
  此版本允许在函数调用完成时释放函数分配的内存，减少了函数使用的总内存。

- 改进 GEQO 优化器性能（Tom）

  此版本修复了 GEQO 优化器管理潜在查询路径方式中的几个低效之处。

- 允许 `IN`/`NOT IN` 通过哈希表处理（Tom）
- 改进 `NOT IN (subquery)` 的性能（Tom）
- 允许大多数 `IN` 子查询作为连接处理（Tom）
- 模式匹配操作可以使用索引而不受区域设置影响（Peter）

  非 ASCII 区域设置无法使用标准索引做 `LIKE` 比较。此版本增加了一种为 `LIKE` 创建特殊索引的方法。

- 允许 postmaster 用 `preload_libraries` 预装载库（Joe）

  对于需要很长时间装载的共享库，此选项使库可以在 postmaster 中预装载并被所有数据库会话继承。

- 改进优化器代价计算，特别是对子查询（Tom）
- 当子查询 `ORDER BY` 与上层查询匹配时避免排序（Tom）
- 从 `WHERE a.x = b.y AND b.y = 42` 推出 `a.x = 42`（Tom）
- 允许复杂连接上的哈希/归并连接（Tom）
- 允许更多数据类型的哈希连接（Tom）
- 允许显式内连接的连接优化，可用 `join_collapse_limit` 禁用（Tom）
- 新增参数 `from_collapse_limit`，控制子查询到连接的转换（Tom）
- 使用来自 Tcl 的更快更强大的正则表达式代码（Henry Spencer、Tom）
- 在优化器中使用位图关系集（Tom）
- 改进连接启动时间（Tom）

  新的客户端/服务器协议启动数据库会话需要更少的网络包。

- 改进触发器/约束的性能（Stephan）
- 改进 `col IN (const, const, const, ...)` 的速度（Tom）
- 修复罕见情形下损坏的哈希索引（Tom）
- 改进哈希索引的并发和速度（Tom）

  以前的版本哈希索引性能不佳，特别是在高并发情形下。此版本修复了这一点，
  开发组有兴趣了解比较 B-tree 与哈希索引性能的报告。

- 把共享缓冲区对齐到 32 字节边界以改进复制速度（Manfred Spraul）

某些 CPU 在地址按 32 字节对齐时执行更快的数据复制。

- 为更好的性能重新实现了 `numeric` 数据类型（Tom）

  `numeric` 以前以 100 为基数存储。新代码使用 10000 为基数，性能显著提升。

E.31.3.3. 服务器配置变更

- 把服务器参数 `server_min_messages` 改名为 `log_min_messages`（Bruce）

  这是为了使大多数控制服务器日志的参数以 `log_` 开头。

- 把 `show_*_stats` 改名为 `log_*_stats`（Bruce）
- 把 `show_source_port` 改名为 `log_source_port`（Bruce）
- 把 `hostname_lookup` 改名为 `log_hostname`（Bruce）
- 新增 `checkpoint_warning`，对过度检查点发出警告（Bruce）

  在以前的版本中，很难确定检查点是否发生得过于频繁。
  此特性在过度检查点发生时向服务器日志添加警告。

- 新的本地化只读服务器参数（Tom）
- 把调试服务器日志消息改为以 `DEBUG` 而不是 `LOG` 输出（Bruce）
- 防止服务器日志变量被非超级用户关闭（Bruce）

  这是一个安全特性，使非超级用户不能禁用管理员启用的日志。

- `log_min_messages`/`client_min_messages` 现在控制 `debug_*` 输出（Bruce）

  这集中了客户端调试信息，使所有调试输出都可以发送到客户端或服务器日志。

- 新增 Mac OS X Rendezvous 服务器支持（Chris Campbell）

  这允许 Mac OS X 主机向网络查询可用的 PostgreSQL 服务器。

- 新增用 `log_min_duration_statement` 只打印慢语句的能力（Christopher）

  这是常被要求的调试特性，允许管理员在服务器日志中只看到慢查询。

- 允许 `pg_hba.conf` 接受 CIDR 格式的网络掩码（Andrew Dunstan）

  这允许管理员把主机 IP 地址和网络掩码字段合并为 `pg_hba.conf` 中的单个 CIDR 字段。

- 新的只读参数 `is_superuser`（Tom）
- 新参数 `log_error_verbosity`，控制错误细节（Tom）

  它与新的错误报告特性一起工作，提供提示、文件名和行号等额外错误信息。

- `postgres --describe-config` 现在转储服务器配置变量（Aizaz Ahmed、Peter）

  此选项对需要知道配置变量名及其最小值、最大值、默认值和描述的管理工具很有用。

- 为 pg_settings 新增列：context、type、source、min_val、max_val（Joe）
- 如果可能，把默认 shared_buffers 设为 1000、max_connections 设为 100（Tom）

  以前的版本默认 64 个共享缓冲区，以便 PostgreSQL 甚至能在很老的系统上启动。此版本测试平台允许的共享内存量并在可能时选择更合理的默认值。当然，
  仍鼓励用户评估其资源负载并相应地设定 shared_buffers 的大小。
- 新的 pg_hba.conf 记录类型 hostnossl，阻止 SSL 连接（Jon Jensen）

  在以前的版本中，如果客户端和服务器都支持 SSL，就没有办法阻止 SSL 连接。此选项提供了该能力。
- 移除参数 geqo_random_seed（Tom）
- 新增服务器参数 regex_flavor，控制正则表达式处理（Tom）
- 让 pg_ctl 更好地处理非标准端口（Greg）

E.31.3.4. 查询变更

- 新的 SQL 标准信息模式（Peter）
- 新增只读事务（Peter）
- 在外键违反消息中打印键名和值（Dmitry Tkach）
- 允许用户在 pg_stat_activity 中看到自己的查询（Kevin Brown）

  在以前的版本中，只有超级用户能用 pg_stat_activity 看到查询字符串。现在普通用户可以看到自己的查询字符串。
- 修复子查询中的聚合以符合 SQL 标准（Tom）

  SQL 标准规定，如果嵌套子查询中出现的聚合函数的参数只包含外层查询变量，则它属于外层查询。以前的 PostgreSQL 版本没有正确处理这个细节。
- 新增阻止自动添加查询中引用的表的选项（Nigel J. Andrews）

  默认情况下，查询中提到的表如果不在 FROM 子句中会被自动添加到其中。这与历史 POSTGRES 行为兼容但违背 SQL 标准。此选项允许选择标准兼容的行为。
- 允许 UPDATE ... SET col = DEFAULT（Rod）

  这允许 UPDATE 把列设为其声明的默认值。
- 允许在 LIMIT/OFFSET 中使用表达式（Tom）

  在以前的版本中，LIMIT/OFFSET 只能使用常数，不能用表达式。
- 实现 CREATE TABLE AS EXECUTE（Neil、Peter）

E.31.3.5. 对象操作变更

- 使 CREATE SEQUENCE 语法更符合 SQL:2003（Neil）
- 新增语句级触发器（Neil）

虽然这允许触发器在语句结束时触发，但不允许触发器访问该语句修改的所有行。此能力计划在未来版本中提供。

- 为域新增检查约束（Rod）

  这通过允许域使用检查约束大幅提高了域的用处。

- 新增 `ALTER DOMAIN`（Rod）

  这允许操纵现有域。

- 修复几个零列表缺陷（Tom）

  PostgreSQL 支持零列表。这修复了使用此类表时发生的各种缺陷。

- 让 `ALTER TABLE ... ADD PRIMARY KEY` 添加非空约束（Rod）

  在以前的版本中，`ALTER TABLE ... ADD PRIMARY` 会添加唯一索引但不添加非空约束。此版本修复了这一点。

- 新增 `ALTER TABLE ... WITHOUT OIDS`（Rod）

  这允许控制新行和更新的行是否有 OID 列。这对节省存储空间最有用。

- 新增 `ALTER SEQUENCE`，修改最小值、最大值、增量、缓存、循环值（Rod）

- 新增 `ALTER TABLE ... CLUSTER ON`（Alvaro Herrera）

  此命令被 `pg_dump` 用来记录每个先前聚簇过的表的聚簇列。该信息被数据库范围的 cluster 用来聚簇所有先前聚簇过的表。

- 改进域的自动类型转换（Rod、Tom）

- 允许标识符中的美元符号，但不能作为首字符（Tom）

- 不允许操作符名中的美元符号，使 `x=$1` 可行（Tom）

- 允许用 `LIKE` *`subtable`* 复制表模式，还有 SQL:2003 特性 `INCLUDING DEFAULTS`（Rod）

- 为 `GRANT` 新增 `WITH GRANT OPTION` 子句（Peter）

  这使 `GRANT` 能授予其他用户在对象上授予权限的能力。

E.31.3.6. 工具命令变更

- 为 `CREATE TABLE` 的临时表新增 `ON COMMIT` 子句（Gavin）

  这增加了在事务提交时删除表或删除所有行的能力。

- 允许用 `WITH HOLD` 在事务之外使用游标（Neil）

  在以前的版本中，游标在创建它的事务结束时被删除。现在可以用 `WITH HOLD` 选项创建游标，这允许在创建事务提交后继续访问它们。

- `FETCH 0` 和 `MOVE 0` 现在不做任何事情（Bruce）

  在以前的版本中，`FETCH 0` 获取所有剩余行，`MOVE 0` 移到游标末尾。

- 按照 SQL 标准，让 `FETCH` 和 `MOVE` 返回获取/移动的行数，如果位于游标开头/末尾则返回零（Bruce）

  在以前的版本中，`FETCH` 和 `MOVE` 返回的行数没有准确反映处理的行数。

- 正确处理游标的 `SCROLL`，否则报告错误（Neil）

  对某些类型的查询允许随机访问（前向和后向滚动）需要一些额外的工作才能做到。
  如果创建游标时指定了 `SCROLL`，将执行这些额外工作。此外，如果游标以 `NO SCROLL` 创建，则不允许随机访问。

- 为 `FETCH` 和 `MOVE` 实现 SQL 兼容的选项 `FIRST`、`LAST`、`ABSOLUTE` *n*、`RELATIVE` *n*（Tom）

- 允许对 `DECLARE CURSOR` 使用 `EXPLAIN`（Tom）

- 允许 `CLUSTER` 默认使用标记为预聚簇的索引（Alvaro Herrera）

- 允许 `CLUSTER` 聚簇所有表（Alvaro Herrera）

  这允许用单条命令重新聚簇数据库中所有先前聚簇过的表。

- 禁止对部分索引使用 `CLUSTER`（Tom）

- 允许 `COPY` 文件中的 DOS 和 Mac 行尾（Bruce）

- 不允许字面回车作为数据值，反斜杠-回车和 `\r` 仍被允许（Bruce）

- `COPY` 变更（二进制、`\.`）（Tom）

- 从 `COPY` 失败中干净恢复（Tom）

- 防止 `COPY` 中可能的内存泄漏（Tom）

- 使 `TRUNCATE` 事务安全（Rod）

  `TRUNCATE` 现在可以在事务内使用。如果事务中止，`TRUNCATE` 所做的更改会自动回滚。

- 允许预备/绑定 `FETCH` 和 `EXPLAIN` 等工具命令（Tom）

- 新增 `EXPLAIN EXECUTE`（Neil）

- 通过减少 WAL 流量改进索引上的 `VACUUM` 性能（Tom）

- 函数索引已被推广为表达式上的索引（Tom）

  在以前的版本中，函数索引只支持应用于一个或多个列名的简单函数。
  此版本允许任何类型的标量表达式。

- 让 `SHOW TRANSACTION ISOLATION` 与 `SET TRANSACTION ISOLATION` 的输入匹配（Tom）

- 让对非本地数据库的 `COMMENT ON DATABASE` 生成警告而不是错误（Rod）

  数据库注释存储在数据库本地表中，因此对数据库的注释必须存储在每个数据库中。

- 改进 `LISTEN`/`NOTIFY` 的可靠性（Tom）

- 允许 REINDEX 可靠地重新索引非共享系统目录索引（Tom）

  这允许重新索引系统表而不需要独立会话，这在以前的版本中是必需的。
  现在唯一需要独立会话来重新索引的是全局系统表 pg_database、pg_shadow 和 pg_group。

E.31.3.7. 数据类型和函数变更

- 新服务器参数 extra_float_digits，控制浮点数的精度显示（Pedro Ferreira、Tom）

  这控制导致回归测试问题的输出精度。

- 允许 +1300 作为数字时区说明符，用于 FJST（Tom）

- 移除很少使用的函数 oidrand、oidsrand 和 userfntest（Neil）

- 把 md5() 函数加入主服务器，此前已在 contrib/pgcrypto 中（Joe）

  MD5 函数是常被要求的。需要更复杂的加密能力时，请使用 contrib/pgcrypto。

- 扩大 timestamp 的日期范围（John Cochran）

- 更改 EXTRACT(EPOCH FROM timestamp)，使 timestamp without time zone 被假定处于本地时间而不是 GMT（Tom）

- 以防操作系统不防止除以零，捕获它（Tom）

- 把 numeric 数据类型内部改为以 10000 为基数（Tom）

- 新函数 hostmask()（Greg Wickham）

- to_char() 和 to_timestamp() 的修复（Karel）

- 允许接受任意参数数据类型并返回任意数据类型的函数，使用 anyelement 和 anyarray（Joe）

  这允许创建能用于任何数据类型的函数。

- 数组现在可以指定为 ARRAY[1,2,3]、ARRAY[['a','b'],['c','d']] 或 ARRAY[ARRAY[ARRAY[2]]]（Joe）

- 允许数组的正确比较，包括 ORDER BY 和 DISTINCT 支持（Joe）

- 允数组列上的索引（Joe）

- 允许用 || 连接数组（Joe）

- 允许 WHERE 限定 *exprop* ANY/SOME/ALL (*array_expr*)（Joe）

  这允许数组表现得像值列表，用于诸如 SELECT * FROM tab WHERE col IN (array_val) 之类的用途。

- 新数组函数 array_append、array_cat、array_lower、array_prepend、array_to_string、array_upper、string_to_array（Joe）

- 允许用户定义的聚合使用多态函数（Joe）

- 允许给空数组赋值（Joe）
- 允许 `time`、`timestamp` 和 `interval` 输入值的秒字段为 60（Tom）

  闰秒需要六十秒的值。
- 允许 `cidr` 数据类型转换为 `text`（Tom）
- 不允许 SET TIMEZONE 中的无效时区名
- 当 `char` 转换为 `varchar` 或 `text` 时修剪尾随空格（Tom）
- 让 `float(p)` 用二进制位而不是十进制位度量精度 *p*（Tom）
- 为 `inet` 和 `cidr` 数据类型添加 IPv6 支持（Michael Graff）
- 新增 `family()` 函数，报告地址是 IPv4 还是 IPv6（Michael Graff）
- 让 `SHOW datestyle` 生成与 `SET datestyle` 所用类似的输出（Tom）
- 让 `EXTRACT(TIMEZONE)` 和 `SET/SHOW TIME ZONE` 遵循 SQL 对时区偏移符号的约定，即正值为 UTC 以东（Tom）
- 修复 `date_trunc('quarter', ...)`（Böjthe Zoltán）

  以前的版本对此函数调用返回不正确的值。
- 使 `initcap()` 更兼容 Oracle（Mike Nolan）

  `initcap()` 现在把任何非字母数字字符之后出现的字母大写，而不只是空白之后。
- 对非 ISO-8601 格式的日期值只允许 `datestyle` 字段顺序（Greg）
- 新增 `datestyle` 值 `MDY`、`DMY` 和 `YMD` 来设置输入字段顺序；为向后兼容保留 `US` 和 `European`（Tom）
- 像 `'now'` 或 `'today'` 这样的字符串字面量将不再可用作列默认值。请改用 `now()`、`current_timestamp` 等函数。（预备语句需要此变更）（Tom）
- 在 `min()`/`max()` 中把 NaN 当作大于任何其他值（Tom）

  在大多数用途中 NaN 已排在普通数值之后，但 `min()` 和 `max()` 没有正确处理。
- 防止 interval 抑制 `:00` 秒的显示
- 新函数 `pg_get_triggerdef(prettyprint)` 和 `pg_conversion_is_visible()`（Christopher）
- 允许时间指定为 `040506` 或 `0405`（Tom）
- 输入日期顺序现在必须是 `YYYY-MM-DD`（4 位年份）或匹配 `datestyle`
- 让 `pg_get_constraintdef` 支持唯一、主键和检查约束（Christopher）

E.31.3.8. 服务器端语言变更

- 防止对零行记录变量使用 `RETURN NEXT` 时 PL/pgSQL 崩溃（Tom）

- 让 PL/Python 的 `spi_execute` 接口正确处理空值（Andrew Bosma）
- 允许 PL/pgSQL 不用 `%ROWTYPE` 声明组合类型的变量（Tom）
- 修复 PL/Python 的 `_quote()` 函数以处理大整数
- 把 PL/Python 改为不可信语言，现名 `plpythonu`（Kevin Jacobs、Tom）

  Python 语言不再支持受限执行环境，因此删除了可信版 PL/Python。如果这种情况改变，将重新添加可供非超级用户使用的 PL/Python 版本。
- 允许多态 PL/pgSQL 函数（Joe、Tom）
- 允许多态 SQL 函数（Joe）
- 改进了 PL/pgSQL 的编译函数缓存机制，完全支持多态（Joe）
- 在 PL/pgSQL 中新增表示函数实际返回类型的新参数 `$0`（Joe）
- 允许 PL/Tcl 和 PL/Python 在多个表上使用同一触发器（Tom）
- 修复 PL/Tcl 的 `spi_prepare`，使其接受参数类型列表中完全限定的类型名（Jan）

E.31.3.9. psql 变更

- 新增 `\pset pager always`，总是使用分页器（Greg）

  这强制使用分页器，即使行数少于屏幕高度。这对跨多个屏幕行的行很有价值。
- 改进 Tab 补全（Rod、Ross Reedstrom、Ian Barwick）
- 把 `\?` 帮助重排为分组（Harald Armin Massa、Bruce）
- 新增列出模式、转换和编码转换的反斜杠命令（Christopher）
- `\encoding` 现在根据服务器参数 `client_encoding` 变化（Tom）

  在以前的版本中，`\encoding` 不知道用 `SET client_encoding` 所做的编码更改。
- 把编辑器缓冲区保存到 readline 历史（Ross）

  当用 `\e` 编辑查询时，结果被保存到 readline 历史中，可用上箭头检索。
- 改进 `\d` 显示（Christopher）
- 增强 HTML 模式使其更符合标准（Greg）
- 新的 `\set AUTOCOMMIT off` 能力（Tom）

  这取代了被移除的服务器参数 `autocommit`。
- 新的 `\set VERBOSITY`，控制错误细节（Tom）

  这控制新的错误报告细节。
- 新的提示符转义序列 `%x`，显示事务状态（Tom）
- psql 的长选项现在在所有平台上可用

E.31.3.10. pg_dump 变更

- 多个 pg_dump 修复，包括 tar 格式和大对象
- 允许 pg_dump 转储特定的模式（Neil）
- 让 pg_dump 保留列存储特性（Christopher）

  这保留 `ALTER TABLE ... SET STORAGE` 信息。
- 让 pg_dump 保留 `CLUSTER` 特性（Christopher）
- 让 pg_dumpall 用 `GRANT/REVOKE` 转储数据库级权限（Tom）
- 允许 pg_dumpall 支持 pg_dump 的选项 `-a`、`-s`、`-x`（Tom）
- 防止 pg_dump 把命令行上指定的标识符转为小写（Tom）
- pg_dump 选项 `--use-set-session-authorization` 和 `--no-reconnect` 现在无作用，所有转储都使用 `SET SESSION AUTHORIZATION`

  pg_dump 不再重新连接来切换用户，而是始终使用 `SET SESSION AUTHORIZATION`。这将减少恢复期间的口令提示。
- pg_dump 的长选项现在在所有平台上可用

  PostgreSQL 现在包含自己的长选项处理例程。

E.31.3.11. libpq 变更

- 新增用于在 Windows 上释放内存的函数 `PQfreemem`，建议用于 `NOTIFY`（Bruce）

  Windows 要求库中分配的内存由同一库中的函数释放，因此 `free()` 不能释放 libpq 分配的内存。`PQfreemem` 是释放 libpq 内存的正确方式，特别是在 Windows 上，也建议在其他平台上使用。
- 记录服务能力并添加示例文件（Bruce）

  这允许客户端在客户端机器上的中心文件中查找连接信息。
- 让 `PQsetdbLogin` 与 `PQconnectdb` 有相同的默认值（Tom）
- 允许 libpq 在结果集过大时干净地失败（Tom）
- 改进函数 `PQunescapeBytea` 的性能（Ben Lamb）
- 用 `configure` 选项 `--enable-thread-safety` 允许线程安全的 libpq（Lee Kindness、Philip Yarra）
- 允许函数 `pqInternalNotice` 接受格式串和参数而不只是预先格式化的消息（Tom、Sean Chittenden）
- 用 `sslmode` 值 `disable`、`allow`、`prefer` 和 `require` 控制 SSL 协商（Jon Jensen）
- 允许新的错误码和文本级别（Tom）
- 允许访问查询结果的基础表和列（Tom）

这对想知道与特定结果集关联的基础表和列名的查询构建器应用程序很有帮助。

- 允许访问当前事务状态（Tom）
- 新增把二进制数据直接传递给服务器的能力（Tom）
- 新增函数 `PQexecPrepared` 和 `PQsendQueryPrepared`，执行先前预备语句的绑定/执行（Tom）

E.31.3.12. JDBC 变更

- 允许可更新结果集上的 `setNull`
- 允许预备语句上的 `executeBatch`（Barry）
- 支持 SSL 连接（Barry）
- 处理结果集中的模式名（Paul Sorenson）
- 新增 refcursor 支持（Nic Ferrier）

E.31.3.13. 其他接口变更

- 防止 libpgtcl 关闭期间可能的内存泄漏或核心转储（Tom）
- 为 ECPG 新增 Informix 兼容性（Michael）

  这允许 ECPG 处理使用某些 Informix 扩展编写的嵌入式 C 程序。
- 为 ECPG 新增定长的 `decimal` 类型，用于 Informix（Michael）
- 用 `configure` 选项 `--enable-thread-safety` 允许线程安全的嵌入式 SQL 程序（Lee Kindness、Bruce）

  这允许多个线程同时访问数据库。
- 把 Python 客户端 PyGreSQL 移到http://www.pygresql.org（Marc）

E.31.3.14. 源代码变更

- 消除对单独的平台几何回归结果文件的需要（Tom）
- 改进的 PPC 锁定原语（Reinhard Max）
- 新函数 `palloc0`，分配并清零内存（Bruce）
- 修复 s390x CPU（64 位）的锁定代码（Tom）
- 允许 OpenBSD 使用本地 ident 凭据（William Ahern）
- 使查询计划树对执行器只读（Tom）
- 新增 Darwin 启动脚本（David Wheeler）
- 允许 libpq 用 Borland C++ 编译器编译（Lester Godwin、Karl Waclawek）

- 需要时使用我们自己的 `getopt_long()` 版本（Peter）
- 把管理脚本转换为 C（Peter）
- 如果从 CVS 构建，现在需要 Bison >= 1.85 来构建 PostgreSQL 语法
- 把文档合并为一本书（Peter）
- 新增 Windows 兼容函数（Bruce）
- 允许客户端接口在 MinGW 下编译（Bruce）
- 新的用于错误报告的 `ereport()` 函数（Tom）
- 支持 Linux 上的 Intel 编译器（Peter）
- 改进 Linux 启动脚本（Slawomir Sudnik、Darko Prenosil）
- 新增对 AMD Opteron 和 Itanium 的支持（Jeffrey W. Baker、Bruce）
- 从 `configure` 移除 `--enable-recode` 选项

  既然有了 `CREATE CONVERSION`，这不再需要。
- 如果未找到自旋锁代码则产生编译错误（Bruce）

  没有自旋锁代码的平台现在将编译失败，而不是静默使用信号量。此失败可以用新的 `configure` 选项禁用。

E.31.3.15. 附加模块变更

- 把 dbmirror 许可证改为 BSD
- 改进 earthdistance（Bruno Wolff III）
- pgcrypto 的可移植性改进（Marko Kreen）
- 防止 xml 中的崩溃（John Gray、Michael Richards）
- 更新 oracle
- 更新 mysql
- 更新 cube（Bruno Wolff III）
- 更新 earthdistance 以使用 cube（Bruno Wolff III）
- 更新 btree_gist（Oleg）
- 新的 tsearch2 全文搜索模块（Oleg、Teodor）
- 为 tablefuncs 新增基于哈希的 crosstab 函数（Joe）
- 为 tablefuncs 新增用来对 `connectby()` 兄弟节点排序的 serial 列（Nabil Sayegh、Joe）
- 为 dblink 新增命名持久连接（Shridhar Daithanka）

- 新的 pg_autovacuum 允许自动 `VACUUM`（Matthew T. O'Connor）
- 让 pgbench 遵循环境变量 `PGHOST`、`PGPORT`、`PGUSER`（Tatsuo）
- 改进 intarray（Teodor Sigaev）
- 改进 pgstattuple（Rod）
- 修复 fuzzystrmatch 中 `metaphone()` 的缺陷
- 改进 adddepend（Rod）
- 更新 spi/timetravel（Böjthe Zoltán）
- 修复 dbase 的 `-s` 选项并改进非 ASCII 处理（Thomas Behr、Márcio Smiderle）
- 移除 array 模块，因为其功能现已默认包含（Joe）

E.32. 版本 7.3.21

发行日期
2008-01-07

此发行版包含 7.3.20 之后的多种修复，包括对重要安全问题的修复。

这预计是 PostgreSQL 7.3.X 系列的最后一个发行版。建议用户尽快更新到较新的发行分支。

E.32.1. 迁移到版本 7.3.21

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.13 的版本升级，请参阅 7.3.13 的发行说明。

E.32.2. 变更

- 防止索引中的函数以运行 `VACUUM`、`ANALYZE` 等的用户的权限执行（Tom）

  每当产生新的表条目时，都会求值索引表达式和部分索引谓词中使用的函数。人们早就明白，如果修改不可信用户拥有的表，这会带来特洛伊木马代码执行的风险。（注意触发器、默认值、检查约束等也构成同类风险。）但索引中的函数带来额外危险，因为它们会由例程维护操作（如 `VACUUM FULL`）执行，而这些操作通常在超级用户账户下自动执行。例如，恶意用户可以通过设置特洛伊木马索引定义并等待下一次例程清理来以超级用户权限执行代码。修复安排标准维护操作（包括 `VACUUM`、`ANALYZE`、`REINDEX` 和 `CLUSTER`）以表所有者而非调用用户的身份执行，使用已用于 `SECURITY DEFINER` 函数的同一权限切换机制。为防止绕过此安全措施，现在禁止在 `SECURITY DEFINER` 上下文中执行 `SET SESSION AUTHORIZATION` 和 `SET ROLE`。（CVE-2007-6600）

- 作为安全措施，要求使用 `/contrib/dblink` 的非超级用户只使用口令认证（Joe）

  7.3.20 中出现的修复并不完整，它只堵住了部分 `dblink` 函数的漏洞。（CVE-2007-6601、CVE-2007-3278）

- 修复使用多字节数据库编码时 `translate()` 可能的崩溃（Tom）
- 让 `contrib/tablefunc` 的 `crosstab()` 把 NULL rowid 本身作为一个类别处理，而不是崩溃（Joe）
- 要求在重新生成 `configure` 脚本时使用特定版本的 Autoconf（Peter）

  这只影响开发者和打包者。此变更是为了防止意外使用未经测试的 Autoconf 与 PostgreSQL 版本组合。如果确实想使用不同的 Autoconf 版本，可以移除版本检查，但结果是否可用由你自己负责。

E.33. 版本 7.3.20

发行日期
2007-09-17

此发行版包含 7.3.19 之后的修复。

E.33.1. 迁移到版本 7.3.20

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.13 的版本升级，请参阅 7.3.13 的发行说明。

E.33.2. 变更

- 防止事务插入行后在同一表上并发 `VACUUM` 接近结束时中止导致的索引损坏（Tom）
- 让 `CREATE DOMAIN ... DEFAULT NULL` 正确工作（Tom）
- 修复 `log_min_error_statement` 日志记录内存耗尽时的崩溃（Tom）
- 作为安全措施，要求使用 `/contrib/dblink` 的非超级用户只使用口令认证（Joe）

E.34. 版本 7.3.19

发行日期
2007-04-23

此发行版包含 7.3.18 之后的修复，包括一个安全修复。

E.34.1. 迁移到版本 7.3.19

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.13 的版本升级，请参阅 7.3.13 的发行说明。

E.34.2. 变更

- 支持在 `search_path` 中显式放置临时表模式，并禁用为函数和操作符搜索它（Tom）

  这是允许安全定义者函数设置真正安全的 `search_path` 值所必需的。没有它，非特权 SQL 用户可以利用临时对象以安全定义者函数的权限执行代码（CVE-2007-2138）。更多信息请参阅 `CREATE FUNCTION`。

- 修复 `VACUUM FULL` 处理 `UPDATE` 链方式中的潜在数据损坏缺陷（Tom、Pavan Deolasee）

E.35. 版本 7.3.18

发行日期
2007-02-05

此发行版包含 7.3.17 之后的多种修复，包括一个安全修复。

E.35.1. 迁移到版本 7.3.18

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.13 的版本升级，请参阅 7.3.13 的发行说明。

E.35.2. 变更

- 移除允许已连接用户读取后端内存的安全漏洞（Tom）

  该漏洞涉及更改 SQL 函数中使用的表列的数据类型（CVE-2007-0555）。此错误很容易被利用来导致后端崩溃，而且原则上可能被用来读取用户不应能访问的数据库内容。

- 修复罕见的缺陷：btree 索引页分裂可能因选择不可行的分裂点而失败（Heikki Linnakangas）

- 收紧对超过三字节的 UTF8 序列的多字节字符处理的安全性（Tom）

E.36. 版本 7.3.17

发行日期
2007-01-08

此发行版包含 7.3.16 之后的多种修复。

E.36.1. 迁移到版本 7.3.17

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.13 的版本升级，请参阅 7.3.13 的发行说明。

E.36.2. 变更

- 对于新的 initdb 安装，`to_number()` 和 `to_char(numeric)` 现在是 `STABLE` 而不是 `IMMUTABLE`（Tom）

  这是因为 `lc_numeric` 可能改变这些函数的输出。

- 改进使用括号的正则表达式的索引利用（Tom）

  这同时也改进了 psql 的 `\d` 性能。

E.37. 版本 7.3.16

发行日期
2006-10-16

此发行版包含 7.3.15 之后的多种修复。

E.37.1. 迁移到版本 7.3.16

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.13 的版本升级，请参阅 7.3.13 的发行说明。

E.37.2. 变更

- 修复 psql 的 `\d` 命令模式匹配的极端情形
- 修复 /contrib/ltree 中损坏索引的缺陷（Teodor）
- 向后移植 7.4 自旋锁代码，以改进性能并更好地支持 64 位架构
- 修复 libpq 中与 SSL 相关的内存泄漏
- 修复 /contrib/dbmirror 中的反斜杠转义
- 针对美国 DST 法规的最近变更调整回归测试

E.38. 版本 7.3.15

发行日期
2006-05-23

此发行版包含 7.3.14 之后的多种修复，包括针对极其严重安全问题的补丁。

E.38.1. 迁移到版本 7.3.15

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.13 的版本升级，请参阅 7.3.13 的发行说明。

要完全防范 CVE-2006-2313 和 CVE-2006-2314 所述的 SQL 注入攻击，可能需要更改应用程序代码。如果你有把不可信字符串嵌入 SQL 命令的应用程序，应尽快检查它们，确保使用了推荐的转义技术。在大多数情况下，应用程序应使用库或驱动提供的子例程（如 libpq 的 `PQescapeStringConn()`）来执行字符串转义，而不是依靠临时代码来做。

E.38.2. 变更

- 更改服务器，在所有情形下拒绝无效编码的多字节字符（Tatsuo、Tom）

  虽然 PostgreSQL 向这个方向发展已有一段时间，但现在检查统一应用于所有编码和所有文本输入，而且总是错误而不只是警告。此变更可防范 CVE-2006-2313 所述类型的 SQL 注入攻击。

- 拒绝字符串字面量中对 `\'` 的不安全使用

  作为针对 CVE-2006-2314 所述类型 SQL 注入攻击的服务器端防御，服务器现在只接受 `''` 而不接受 `\'` 作为 SQL 字符串字面量中 ASCII 单引号的表示。默认情况下，只有当 `client_encoding` 设为仅客户端编码（SJIS、BIG5、GBK、GB18030 或 UHC）时才拒绝 `\'`，这正是可能发生 SQL 注入的场景。需要时可用新的配置参数 `backslash_quote` 调整此行为。注意，要完全防范 CVE-2006-2314 可能需要客户端更改；`backslash_quote` 的目的之一是让不安全的客户端明显地不安全。

- 修改 libpq 的字符串转义例程，使其感知编码问题

  这为使用 libpq 的应用程序修复了 CVE-2006-2313 和 CVE-2006-2314 所述的安全问题。并发使用多个 PostgreSQL 连接的应用程序应迁移到 `PQescapeStringConn()` 和 `PQescapeByteaConn()`，以确保为每个数据库连接中使用的设置正确转义。以“手工”方式做字符串转义的应用程序应改为依靠库例程。

- 修复一些不正确的编码转换函数

  `win1251_to_iso`、`alt_to_iso`、`euc_tw_to_big5`、`euc_tw_to_mic`、`mic_to_euc_tw` 都在不同程度上损坏。

- 清理字符串中偶然残留的 `\'` 使用（Bruce、Jan）

- 修复服务器正确使用自定义 DH SSL 参数（Michael Fuhr）

- 修复各种小的内存泄漏

E.39. 版本 7.3.14

发行日期
2006-02-14

此发行版包含 7.3.13 之后的多种修复。

E.39.1. 迁移到版本 7.3.14

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.13 的版本升级，请参阅 7.3.13 的发行说明。

E.39.2. 变更

- 修复 `SET SESSION AUTHORIZATION` 中可能的崩溃（CVE-2006-0553）

  如果服务器编译时启用了断言（这不是默认），非特权用户可以使服务器进程崩溃，导致对其他用户的瞬时拒绝服务。感谢 Akio Ishida 报告此问题。

- 修复自插入行的行可见性逻辑缺陷（Tom）

  极少数情况下，当前命令插入的行可能被看作已经有效，而不应该如此。修复 7.3.11 版本中引入的缺陷。

- 修复 pg_clog 文件创建期间可能导致“文件已存在”错误的竞态条件（Tom）
- 修复以允许恢复有对自定义操作符跨模式引用的转储（Tom）
- 修复 configure 期间测试 `finite` 和 `isinf` 存在性的可移植性问题（Tom）

E.40. 版本 7.3.13

发行日期
2006-01-09

此发行版包含 7.3.12 之后的多种修复。

E.40.1. 迁移到版本 7.3.13

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.10 的版本升级，请参阅 7.3.10 的发行说明。另外，如果你受到下述区域设置或 plperl 问题的影响，可能需要在更新后对文本列上的索引执行 `REINDEX`。

E.40.2. 变更

- 修复把不同字符组合视为相等的区域设置（如匈牙利语）下的字符串比较（Tom）

  这可能需要 `REINDEX` 来修复文本列上的现有索引。

- 在 postmaster 启动期间设置区域设置环境变量，以确保 plperl 之后不会更改区域设置

  这修复了 postmaster 启动时环境变量指定的区域设置与 initdb 所被告知的不同时发生的问题。在这些条件下，任何 plperl 的使用都可能导致索引损坏。如果你遇到过这种情况，可能需要 `REINDEX` 来修复文本列上的现有索引。

- 修复某些很少使用的亚洲多字节字符集中 strpos() 和正则表达式处理的长期缺陷（Tatsuo）
- 修复 `/contrib/pgcrypto` gen_salt 中的缺陷，该缺陷使 MD5 和 XDES 算法不能使用全部可用盐空间（Marko Kreen、Solar Designer）

  Blowfish 和标准 DES 的盐不受影响。

- 修复 `/contrib/dblink`，当指定的列数与查询实际返回的列数不同时抛出错误而不是崩溃（Joe）

E.41. 版本 7.3.12

发行日期
2005-12-12

此发行版包含 7.3.11 之后的多种修复。

E.41.1. 迁移到版本 7.3.12

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.10 的版本升级，请参阅 7.3.10 的发行说明。

E.41.2. 变更

- 修复事务日志管理中的竞态条件

  存在一个窄窗口，其中可能为错误的页发起 I/O 操作，导致断言失败或数据损坏。

- `/contrib/ltree` 修复（Teodor）

- 修复外连接的长期规划错误

  此缺陷有时导致虚假错误“RIGHT JOIN 只支持可归并连接的连接条件”。

- 防止表被删除后 pg_autovacuum 的核心转储

E.42. 版本 7.3.11

发行日期
2005-10-04

此发行版包含 7.3.10 之后的多种修复。

E.42.1. 迁移到版本 7.3.11

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，如果你是从早于 7.3.10 的版本升级，请参阅 7.3.10 的发行说明。

E.42.2. 变更

- 修复允许 `VACUUM` 过早移除 `ctid` 链的错误，并在跟随 `ctid` 链接的代码中添加更多检查

  这修复了一个在极罕见情况下可能导致崩溃的长期问题。

- 修复 `CHAR()` 在使用多字节字符集时正确填充空格到指定长度（Yoshiyuki Asaba）

在以前的版本中，`CHAR()` 的填充是不正确的，因为它只填充到指定的字节数而不考虑存储了多少字符。

- 修复列 `a` 上有 GiST 索引时 `UPDATE a=... WHERE a...` 之类查询丢行的问题
- 改进对部分写入 WAL 页的检查
- 改进启用 SSL 时信号处理的健壮性
- 各种内存泄漏修复
- 各种可移植性改进
- 修复 PL/pgSQL 在变量为按引用传递类型时正确处理 `var := var`

E.43. 版本 7.3.10

发行日期
2005-05-09

此发行版包含 7.3.9 之后的多种修复，包括几个与安全相关的问题。

E.43.1. 迁移到版本 7.3.10

对于运行 7.3.X 的用户，不需要进行转储/恢复。不过，它是处理在 7.3.X 系统目录初始内容中发现的严重安全问题的可能方法之一。用 7.3.10 的 initdb 做转储/initdb/重载序列会自动纠正此问题。

安全问题是内建字符集编码转换函数可以由非特权用户从 SQL 命令调用，但这些函数并非为此设计，对恶意选择的参数不安全。修复包括更改这些函数的声明参数列表，使其不能再从 SQL 命令调用。（这不影响编码转换机制对它们的正常使用。）强烈建议所有安装通过 initdb 或按照下述手工修复程序修复此错误。该错误至少允许非特权数据库用户使其服务器进程崩溃，而且可能允许非特权用户获得数据库超级用户的权限。

如果你不想做 initdb，请执行以下程序。作为数据库超级用户，执行：

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
     AND proargtypes[2] = 'cstring'::regtype;
-- The command should report having updated 90 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

上述程序必须在安装的每个数据库中执行，包括 `template1`，理想情况下也包括 `template0`。如果不修复模板数据库，之后创建的任何数据库都会包含相同的错误。`template1` 可以像任何其他数据库一样修复，但修复 `template0` 需要额外的步骤。首先，从任意数据库发出：

```
UPDATE pg_database SET datallowconn = true WHERE datname =
 'template0';
```

接下来连接到 `template0` 并执行上述修复程序。最后执行：

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname =
 'template0';
```

E.43.2. 变更

- 更改编码函数签名以防止误用
- 修复古老的竞态条件，它允许事务在某些用途（如 SELECT FOR UPDATE）上比其他用途稍早被视为已提交

  这是一个极其严重的缺陷，因为它可能导致应用程序短暂看到表面的数据不一致。
- 修复关系扩展与 VACUUM 之间的竞态条件

  这在理论上可能导致丢失一页刚插入的数据，尽管该情形概率似乎很低。
  没有已知案例造成超过断言失败的后果。
- 修复 `TIME WITH TIME ZONE` 值的比较

  在使用了 `--enable-integer-datetimes` 配置开关的情形下比较代码是错误的。注意：如果你在 `TIME WITH TIME ZONE` 列上有索引，安装此更新后需要对它执行 `REINDEX`，因为该修复纠正了列值的排序顺序。
- 修复 `TIME WITH TIME ZONE` 值的 `EXTRACT(EPOCH)`
- 修复 `INTERVAL` 值负小数秒的错误显示

  此错误只在使用了 `--enable-integer-datetimes` 配置开关时发生。
- plpgsql 中的额外缓冲区溢出检查（Neil）
- 修复 pg_dump，正确转储包含 `%` 的触发器名（Neil）
- 防止 `to_char(interval)` 在月份相关格式上转储核心
- 修复 `contrib/pgcrypto` 在较新 OpenSSL 构建上的问题（Marko Kreen）
- `contrib/intagg` 的更多 64 位修复
- 防止对返回 `RECORD` 的函数的错误优化

E.44. 版本 7.3.9

发行日期
2005-01-31

此发行版包含 7.3.8 之后的多种修复，包括几个与安全相关的问题。

E.44.1. 迁移到版本 7.3.9

对于运行 7.3.X 的用户，不需要进行转储/恢复。

E.44.2. 变更

- 不允许非超级用户使用 `LOAD`

  在会自动执行共享库初始化函数的平台（至少包括 Windows 和基于 ELF 的 Unix）上，`LOAD` 可被用来让服务器执行任意代码。感谢 NGS Software 报告此问题。

- 检查聚合函数的创建者是否有权执行指定的转换函数

  这一疏漏使得绕过函数 EXECUTE 权限的拒绝成为可能。

- 修复 contrib/intagg 的安全和 64 位问题
- 为一些 contrib 函数添加所需的 STRICT 标记（Kris Jurka）
- 避免 plpgsql 游标声明参数过多时的缓冲区溢出（Neil）
- 修复 FULL 和 RIGHT 外连接的规划错误

  连接的结果被错误地认为与左输入的排序相同。这不仅会向用户交付排序错误的输出，而且在嵌套归并连接的情形下可能给出完全错误的答案。

- 修复 plperl 处理元组字段中引号的问题
- 修复 SQL 和 GERMAN datestyle 下负 interval 的显示

E.45. 版本 7.3.8

发行日期
2004-10-22

此发行版包含 7.3.7 之后的多种修复。

E.45.1. 迁移到版本 7.3.8

对于运行 7.3.X 的用户，不需要进行转储/恢复。

E.45.2. 变更

- 修复磁盘上提示位可能未更新的问题

  极少数情况下，这一疏漏可能导致“无法访问事务状态”失败，这使它成为一个潜在的数据丢失缺陷。

- 确保哈希外连接不漏掉元组

  在恰当的数据分布下，使用哈希连接计划的超大型左连接可能无法输出未匹配的左侧行。

- 不允许以 root 运行 pg_ctl

  这是为了防范任何可能的安全问题。

- 避免 make_oidjoins_check 在 /tmp 中使用临时文件

  这被报告为安全问题，不过几乎不值得担心，因为非开发者本来就没有理由使用此脚本。

E.46. 版本 7.3.7

发行日期
2004-08-16

此发行版在 7.3.6 的基础上包含一个关键修复和一些次要项。

E.46.1. 迁移到版本 7.3.7

对于运行 7.3.X 的用户，不需要进行转储/恢复。

E.46.2. 变更

- 防止崩溃期间已提交事务的可能丢失

  由于事务提交与检查点之间的互锁不足，
  最近一次检查点之前刚提交的事务可能在数据库崩溃和重启后全部或部分丢失。这是自 PostgreSQL 7.1 以来就存在的严重缺陷。

- 移除 tsearch 中不对称的词处理（Teodor）

- 在 pg_dump 转储 CAST 时正确地对函数名加模式限定

E.47. 版本 7.3.6

发行日期
2004-03-02

此发行版包含 7.3.5 之后的多种修复。

E.47.1. 迁移到版本 7.3.6

对于运行 7.3.* 的用户，不需要进行转储/恢复。

E.47.2. 变更

- 回退规则权限检查中的错误变更

7.3.3 中为修复规则权限检查极端情形而应用的补丁结果在许多并非极端的情形中禁用了与规则相关的权限检查。这例如会允许用户向他们本无权插入的视图插入。因此我们回退了 7.3.3 的补丁。原始缺陷将在 8.0 中修复。

- 修复 GetNewTransactionId() 中错误的操作顺序

  此缺陷可能导致磁盘空间耗尽条件下失败，包括释放磁盘空间后也无法重启。

- 确保即使提供了外部 CFLAGS 值，configure 也选择 -fno-strict-aliasing

  在某些平台上，用 -fstrict-aliasing 构建会导致缺陷。

- 让 pg_restore 正确处理 64 位 off_t

  此缺陷妨碍了从超过 4 GB 的归档文件正确恢复。

- 让 contrib/dblink 不假定本地和远程类型 OID 匹配（Joe）
- 正确引用 connectby() 的 start_with 参数（Joe）
- 当 plpgsql 函数的行类型参数为 NULL 时不崩溃
- 避免在规划 LIKE 操作时的极端情形生成无效字符编码序列
- 确保 text_position() 在多字节情形不能扫描越过源字符串末尾（Korea PostgreSQL Users' Group）
- 修复 bytea 列上 LIKE 操作的索引优化和选择率估计（Joe）

E.48. 版本 7.3.5

发行日期
2003-12-03

这包含 7.3.4 之后的多种修复。

E.48.1. 迁移到版本 7.3.5

对于运行 7.3.* 的用户，不需要进行转储/恢复。

E.48.2. 变更

- 强制在从 WAL 恢复期间开启 zero_damaged_pages
- 防止“变量不在子计划目标列表中”的一些晦涩情形
- 强制统计进程从共享内存脱离，确保更干净的关闭
- 使 PQescapeBytea 与 byteaout 相互一致（Joe）
- 为 dblink 的 get_tuple_of_interest() 添加缺失的 SPI_finish() 调用（Joe）

- 修复规则重写 INSERT 时可能的外键违反（Jan）
- 支持 PL/Tcl 的 spi_prepare 命令中限定类型名（Jan）
- 让 pg_dump 处理位于 pg_catalog 中的过程语言处理器
- 让 pg_dump 处理自定义操作符类在另一个模式中的情形
- 让 pg_dump 正确转储二进制兼容的转换（Jan）
- 修复把包含子查询的表达式插入规则体的问题
- 修复 clusterdb 脚本中不正确的参数处理（Anand Ranganathan）
- 修复 plpython 触发器中已删除列的问题
- 修复 to_char() 读取越过输入字符串末尾的问题（Karel）
- 修复 GB18030 映射错误（Tatsuo）
- 修复 SSL 错误处理和异步 SSL I/O 的几个问题
- 移除 JDBC 中把值列表绑定到单个参数的能力（防止可能的 SQL 注入攻击）
- 修复 HAVE_INT64_TIMESTAMP 代码路径中的一些错误
- 修复与首个根页分裂并行的 btree 搜索的极端情形

E.49. 版本 7.3.4

发行日期
2003-07-24

这包含 7.3.3 之后的多种修复。

E.49.1. 迁移到版本 7.3.4

对于运行 7.3.* 的用户，不需要进行转储/恢复。

E.49.2. 变更

- 修复 2000 年之前日期的时间戳到日期转换的损坏
- 防止服务器启动失败的罕见可能（Tom）
- 修复 interval 到 time 转换中的缺陷（Tom）
- 在 pg_dump 的几处添加约束名（Rod）
- 改进多参数函数的性能（Tom）
- 修复 to_ascii() 缓冲区溢出（Tom）

- 防止恢复数据库注释时抛出错误（Tom）
- 绕过某些 Solaris 版本中存在缺陷的 strxfrm()（Tom）
- 正确转义 jdbc setObject() 字符串以改进安全（Barry）

E.50. 版本 7.3.3

发行日期
2003-05-22

此发行版包含 7.3.2 版本的多种修复。

E.50.1. 迁移到版本 7.3.3

对于运行 7.3.* 版本的用户，不需要进行转储/恢复。

E.50.2. 变更

- 修复崩溃后 StartUpID 有时不正确的计算
- 避免一个事务中大量延迟触发器的缓慢（Stephan）
- 当 `UPDATE` 未更改外键值时不锁定被引用行（Jan）
- 在 Sparc 上使用 `-fPIC` 而不是 `-fpic`（Tom Callaway）
- 修复 contrib/reindexdb 缺乏模式感知的问题
- 修复 contrib/intarray 对零元素结果数组的错误（Teodor）
- 确保 createuser 脚本在 control-C 时退出（Oliver）
- 修复已删除列的类型本身也被删除时的错误
- `CHECKPOINT` 在非关键步骤失败时不再导致数据库 panic
- 接受 timestamp、time、interval 输入值秒字段中的 60
- 当 `TIMESTAMP`、`TIME` 或 `INTERVAL` 精度过大时发出通知而不是错误
- 修复 `abstime-to-time` 转换函数（除非 initdb，否则不应用修复）
- 修复 `timestampt_izone` 的 pg_proc 条目（除非 initdb，否则不应用修复）
- 让 `EXTRACT(EPOCH FROM timestamp without time zone)` 把输入视为本地时间
- 如果事务早些时候更改了时区，`'now'::timestamptz` 给出错误答案
- 带时区 time 的 `HAVE_INT64_TIMESTAMP` 代码覆写了其输入
- 接受 `GLOBAL TEMP/TEMPORARY` 作为 `TEMPORARY` 的同义词

- 避免外键触发器中不当的模式权限检查失败
- 修复 `SET DEFAULT` 动作的外键触发器中的缺陷
- 修复 `UPDATE` 和 `DELETE` 触发器行获取中不正确的 time-qual 检查
- `ALTER TABLE ADD COLUMN` 中外键子句被解析但被忽略
- 修复处理函数已存在时 createlang 脚本的损坏
- 修复 pg_dump、COPY、ANALYZE 等处对零列表的行为异常
- 修复 `func_error()` 对包含 '%' 的类型名的行为异常
- 修复 `replace()` 对包含 '%' 的字符串的行为异常
- 包含某些多字节字符的正则表达式模式失败
- 在连接大小估计中在更多情形正确考虑 `NULL`
- 避免与 `isblank()` 函数或宏的系统定义冲突
- 修复 EUC_TW 转换中未能转换大码点值的问题（Tatsuo）
- 修复 `SSL_read`/`SSL_write` 调用的错误恢复
- 不提前做类型强制表达式的常量折叠
- 读入任何页后立即验证页头字段
- 修复未命名连接中对未分组变量的不正确检查
- 修复 `to_ascii` 中的缓冲区溢出（Guido Notari）
- contrib/ltree 修复（Teodor）
- 修复 char 为无符号的机器上死锁检测的核心转储
- 避免多路索引扫描耗尽缓冲区（7.3 中引入的缺陷）
- 修复规划器的选择率估计函数以正确处理域
- 修复 dbmirror 内存分配缺陷（Steven Singer）
- 防止 `ln(numeric)` 因舍入误差无限循环
- 如果存在多个相等的 GROUP BY 项，`GROUP BY` 会混乱
- 修复继承的 `UPDATE`/`DELETE` 引用另一个继承表时的坏计划
- 防止在不完整（部分或不存储 NULL）的索引上聚簇
- 如果关闭请求在启动期间到达，在正确时刻处理它
- 修复临时索引中的左链接（可能使反向扫描漏掉条目）
- 修复 postgresql.conf 中 client_encoding 设置的不正确处理（Tatsuo）

- 修复 Async_NotifyHandler 运行后不响应 `pg_ctl stop -m fast` 的问题
- 修复规则包含多个同类型语句情形的 SPI
- 修复规则查询中检查错误类型访问权限的问题
- 修复 `CREATE RULE` 中 `EXCEPT` 的问题
- 防止删除带 serial 列的临时表的问题
- 修复复杂视图中 replace_vars_with_subplan_refs 的失败
- 修复单字节编码中正则表达式的缓慢（Tatsuo）
- 允许 `CREATE CAST` 和 `DROP CAST` 中的限定类型名
- 接受 `SETOF type[]`，以前必须写成 `SETOF _type`
- 修复某些带过程语言情形下 pg_dump 的核心转储
- 为可移植性在 pg_dump 输出中强制 ISO datestyle（Oliver）
- pg_dump 未能处理 `lo_read` 的错误返回（Oleg Drokin）
- pg_dumpall 对没有成员的组失败（Nick Eskelinen）
- pg_dumpall 未能识别 --globals-only 开关
- 如果指定了 -X disable-triggers，pg_restore 未能恢复 blob
- 修复 plpgsql 中的函数内内存泄漏
- 如果给定错误参数，pltcl 的 `elog` 命令转储核心（Ian Harding）
- plpython 使用了错误的 `atttypmod` 值（Brad McLean）
- 修复 Python 接口中布尔值的不当引用（D'Arcy）
- 为 JDBC 的 PGConnection 接口添加 `addDataType()` 方法
- 修复 JDBC 可更新 ResultSet 的各种问题（Shawn Green）
- 修复 JDBC DatabaseMetaData 的各种问题（Kris Jurka、Peter Royal）
- 修复 JDBC 中解析表 ACL 的问题
- JDBC 中字符集转换问题的更好错误消息

E.51. 版本 7.3.2

发行日期
2003-02-04

此发行版包含 7.3.1 版本的多种修复。

E.51.1. 迁移到版本 7.3.2

对于运行 7.3.* 版本的用户，不需要进行转储/恢复。

E.51.2. 变更

- 恢复 CREATE TABLE AS / SELECT INTO 中 OID 列的创建
- 修复转储带注释视图时 pg_dump 的核心转储
- 正确转储 DEFERRABLE/INITIALLY DEFERRED 约束
- 修复子表列编号与父表不同时的 UPDATE
- 提高 max_fsm_relations 的默认值
- 修复单行查询游标反向获取时的问题
- 让 SELECT DISTINCT 查询游标的反向获取正确工作
- 修复装载包含 contrib/lo 用法的 pg_dump 文件的问题
- 修复全数字用户名的问题
- 修复 libpgtcl 断开连接期间可能的内存泄漏和核心转储
- 让 plpython 的 spi_execute 命令正确处理空值（Andrew Bosma）
- 调整 plpython 错误报告使其回归测试再次通过
- 与 bison 1.875 一起工作
- 正确处理 plpgsql %type 中的大小写混合名（Neil）
- 修复执行被规则重写的查询时 pltcl 的核心转储
- 修复数组下标越界（据 Yichen Xie 报告）
- 把浮点情形的 MAX_TIME_PRECISION 从 13 减到 10
- 在逐数据库和逐用户设置中正确折叠变量名的大小写
- 修复 SELECT INTO 记录不返回行时 plpgsql RETURN NEXT 的核心转储
- 修复 python 客户端接口中对 pg_type.typprtlen 的过时使用
- 正确处理 JDBC 驱动中时间戳的小数秒
- 改进 JDBC 中 getImportedKeys() 的性能
- 使共享库符号链接在 HPUX 上按标准工作（Giles）
- 修复 timestamp、time、interval 不一致的舍入行为
- SSL 协商修复（Nathan Mueller）
- 使 libpq 的 ~/.pgpass 特性在用 PQconnectDB 连接时工作

- 更新 my2pg、ora2pg
- 翻译更新
- 在 contrib/lo 中添加 lo 与 oid 类型之间的转换
- fastpath 代码现在检查调用函数的权限

E.52. 版本 7.3.1

发行日期
2002-12-18

此发行版包含 7.3 版本的多种修复。

E.52.1. 迁移到版本 7.3.1

对于运行 7.3 版本的用户，不需要进行转储/恢复。但应注意，主 PostgreSQL 接口库 libpq 在此版本有了新的主版本号，某些情况下可能需要重新编译客户端代码。

E.52.2. 变更

- 修复客户端/服务器编码不匹配时 COPY TO 的核心转储（Tom）
- 允许 pg_dump 与 7.2 之前的服务器一起工作（Philip）
- contrib/adddepend 修复（Tom）
- 修复删除逐用户/逐数据库配置设置的问题（Tom）
- contrib/vacuumlo 修复（Tom）
- 即使 pg_shadow 包含 MD5 口令也允许 'password' 加密（Bruce）
- contrib/dbmirror 修复（Steven Singer）
- 优化器修复（Tom）
- contrib/tsearch 修复（Teodor Sigaev、Magnus）
- 允许区域设置名为大小写混合（Nicolai Tufar）
- 递增 libpq 库的主版本号（Bruce）
- pg_hba.conf 错误报告修复（Bruce、Neil）
- 新增 SCO Openserver 5.0.4 为受支持平台（Bruce）
- 防止 EXPLAIN 使服务器崩溃（Tom）
- SSL 修复（Nathan Mueller）
- 防止通过 ALTER TABLE 创建组合列（Tom）

E.53. 版本 7.3

发行日期
2002-11-27

E.53.1. 概述

此发行版的主要变更：

Schemas

模式允许用户在独立的命名空间中创建对象，因此两个人或应用程序可以拥有同名的表。还有用于共享表的 public 模式。可以通过移除 public 模式上的权限来限制表/索引的创建。

Drop Column

PostgreSQL 现在支持 `ALTER TABLE ... DROP COLUMN` 功能。

Table Functions

返回多行和/或多列的函数现在比以前好用得多。可以在 `SELECT` 的 `FROM` 子句中调用这样的“表函数”，把其输出当作表。此外，PL/pgSQL 函数现在可以返回集合。

Prepared Queries

PostgreSQL 现在支持预备查询，以改进性能。

Dependency Tracking

PostgreSQL 现在记录对象依赖，这在许多方面带来了改进。`DROP` 语句现在接受 `CASCADE` 或 `RESTRICT` 来控制是否也删除依赖对象。

Privileges

函数和过程语言现在有权限，函数可以定义为以其创建者的权限运行。

Internationalization

多字节和区域设置支持现在总是启用。

Logging

多种日志选项得到了增强。

Interfaces

大量接口已被移到 http://gborg.postgresql.org，在那里可以独立开发和发布。

Functions/Identifiers

默认情况下，函数现在最多可接受 32 个参数，标识符最长可达 63 字节。此外，`OPAQUE` 现已废弃：有特定的“伪数据类型”来表示函数参数和结果类型中 `OPAQUE` 以前的每种含义。

E.53.2. 迁移到版本 7.3

希望从任何以前的版本迁移数据的用户，需要使用 pg_dump 进行转储/恢复。如果你的应用程序检查系统目录，由于 7.3 中引入了模式，还需要额外的更改；更多信息参见http://developer.postgresql.org/~momjian/upgrade_tips_7.3。

请注意以下不兼容之处：

- 不再支持 6.3 之前的客户端。
- `pg_hba.conf` 现在有用户名列和额外特性。现有文件需要调整。
- 几个 `postgresql.conf` 日志参数已被重命名。
- `LIMIT #,#` 已被禁用；请使用 `LIMIT # OFFSET #`。
- 带列列表的 `INSERT` 语句必须为每个指定的列提供值。例如 `INSERT INTO tab (col1, col2) VALUES ('val1')` 现在无效。如果 `INSERT` 没有列列表，仍允许提供比预期少的列。
- `serial` 列不再自动 `UNIQUE`；因此不会自动创建索引。
- 已中止事务内的 `SET` 命令现在会被回滚。
- `COPY` 不再把缺失的尾随列视为空。需要指定所有列。（不过，可以通过在 `COPY` 命令中指定列列表达到类似效果。）
- 数据类型 `timestamp` 现在等价于 `timestamp without time zone`，而不是 `timestamp with time zone`。
- 装载到 7.3 的 7.3 之前数据库不会有 `serial` 列、唯一约束和外键的新对象依赖。参见 `contrib/adddepend/` 目录中的详细说明和添加此类依赖的脚本。
- 不再允许空字符串（`''`）作为整数字段的输入。以前它被静默解释为 0。

E.53.3. 变更

E.53.3.1. 服务器操作

- 新增显示锁的 pg_locks 视图（Neil）
- 口令协商内存分配的安全修复（Neil）
- 移除对版本 0 FE/BE 协议的支持（PostgreSQL 6.2 及更早）（Tom）
- 为超级用户保留最后几个后端槽，新增参数 superuser_reserved_connections 来控制（Nigel J. Andrews）

E.53.3.2. 性能

- 通过只调用一次 localtime() 改进启动（Tom）
- 把系统目录信息缓存在平面文件中以加快启动（Tom）
- 改进索引信息的缓存（Tom）

- 优化器改进（Tom、Fernando Nasser）
- 目录缓存现在存储失败的查找（Tom）
- 哈希函数改进（Neil）
- 改进查询词法化和网络处理的性能（Peter）
- 大对象恢复的速度改进（Mario Weilguni）
- 首次查找时标记过期索引条目，节省后续堆获取（Tom）
- 避免过多的 NULL 位图填充（Manfred Koizar）
- 为 Solaris 添加 BSD 许可的 qsort()，以提升性能（Bruce）
- 把每行开销减少四字节（Manfred Koizar）
- 修复 GEQO 优化器缺陷（Neil Conway）
- 使 WITHOUT OID 实际每行节省四字节（Manfred Koizar）
- 新增 default_statistics_target 变量，指定 ANALYZE 桶数（Neil）
- 临时表使用本地缓冲区缓存，因此没有 WAL 开销（Tom）
- 改进大表上空闲空间映射的性能（Stephen Marshall、Tom）
- 改进的 WAL 写并发（Tom）

E.53.3.3. 权限

- 新增函数和过程语言上的权限（Peter）
- 为 CREATE DATABASE 新增 OWNER，使超级用户能代表非特权用户创建数据库（Gavin Sherry、Tom）
- 新增对象权限位 EXECUTE 和 USAGE（Tom）
- 新增 SET SESSION AUTHORIZATION DEFAULT 和 RESET SESSION AUTHORIZATION（Tom）
- 允许函数以其所有者的权限执行（Peter）

E.53.3.4. 服务器配置

- 服务器日志消息现在标记为 LOG 而不是 DEBUG（Bruce）
- 为 pg_hba.conf 添加用户列（Bruce）
- 让 log_connections 在日志文件中输出两行（Tom）
- 从 postgresql.conf 移除 debug_level，现为 server_min_messages（Bruce）
- 新的 ALTER DATABASE/USER ... SET 命令，用于逐用户/数据库初始化（Peter）
- 新参数 server_min_messages 和 client_min_messages，控制哪些消息发送到服务器日志或客户端应用程序（Bruce）

- 允许 pg_hba.conf 指定以逗号分隔的用户/数据库列表、以 + 为前缀的组名和以 @ 为前缀的文件名（Bruce）
- 移除辅助口令文件能力和 pg_password 工具（Bruce）
- 新增变量 db_user_namespace，用于数据库本地用户名（Bruce）
- SSL 改进（Bear Giles）
- 使存储口令的加密成为默认（Bruce）
- 允许通过调用 pg_stat_reset() 重置 pg_statistics（Christopher）
- 新增 log_duration 参数（Bruce）
- 把 debug_print_query 改名为 log_statement（Bruce）
- 把 show_query_stats 改名为 show_statement_stats（Bruce）
- 新增参数 log_min_error_statement，在出错时把命令打印到日志（Gavin）

E.53.3.5. 查询

- 使游标不敏感，即其内容不变（Tom）
- 禁用 LIMIT #,# 语法；现在只支持 LIMIT # OFFSET #（Bruce）
- 把标识符长度增加到 63（Neil、Bruce）
- UNION 修复：合并不同长度的 3 列以上（Tom）
- 为 INSERT 新增 DEFAULT 关键字，例如 INSERT ... (..., DEFAULT, ...)（Rod）
- 允许视图用 ALTER COLUMN ... SET DEFAULT 拥有默认值（Neil）
- 对列列表未提供所有列值的 INSERT 失败，例如 INSERT INTO tab (col1, col2) VALUES ('val1')；（Rod）
- 连接别名的修复（Tom）
- FULL OUTER JOIN 的修复（Tom）
- 改进无效标识符和位置的报告（Tom、Gavin）
- 修复 OPEN cursor(args)（Tom）
- 允许在视图中使用 'ctid' 以及 currtid(viewname)（Hiroshi）
- 带 UNION 的 CREATE TABLE AS 的修复（Tom）
- SQL99 语法改进（Thomas）
- 新增 statement_timeout 变量来取消查询（Bruce）
- 允许用 PREPARE/EXECUTE 预备查询（Neil）
- 允许 FOR UPDATE 出现在 LIMIT/OFFSET 之后（Bruce）
- 新增变量 autocommit（Tom、David Van Wie）

E.53.3.6. 对象操作

- 使 CREATE DATABASE 中的等号可选（Gavin Sherry）
- 让 ALTER TABLE OWNER 也更改索引所有权（Neil）
- 新 ALTER TABLE tabname ALTER COLUMN colname SET STORAGE 控制 TOAST 存储和压缩（John Gray）
- 新增模式支持，CREATE/DROP SCHEMA（Tom）
- 为临时表创建模式（Tom）
- 新增用于模式搜索的变量 search_path（Tom）
- 新增 ALTER TABLE SET/DROP NOT NULL（Christopher）
- 新 CREATE FUNCTION 易变性级别（Tom）
- 使规则名只在每个表内唯一（Tom）
- 为 DROP RULE 和 COMMENT ON RULE 新增 'ON tablename' 子句（Tom）
- 新增 ALTER TRIGGER RENAME（Joe）
- 新 current_schema() 和 current_schemas() 查询函数（Tom）
- 允许函数返回多行（表函数）（Joe）
- 为一致性使 CREATE DATABASE 中的 WITH 可选（Bruce）
- 新增对象依赖跟踪（Rod、Tom）
- 为 DROP 命令新增 RESTRICT/CASCADE（Rod）
- 为非 CHECK 约束新增 ALTER TABLE DROP（Rod）
- 删除带 SERIAL 的表时自动销毁序列（Rod）
- 如果列被外键使用则阻止删除列（Rod）
- 对象被删除时自动删除约束/函数（Rod）
- 新增 CREATE/DROP OPERATOR CLASS（Bill Studenmund、Tom）
- 新增 ALTER TABLE DROP COLUMN（Christopher、Tom、Hiroshi）
- 防止继承列被删除或重命名（Alvaro Herrera）
- 修复外键约束在中间数据库状态不报错（Stephan）
- 把列或表重命名传播到外键约束
- 新增 CREATE OR REPLACE VIEW（Gavin、Neil、Tom）
- 新增 CREATE OR REPLACE RULE（Gavin、Neil、Tom）
- 让规则按字母顺序执行，返回更可预测的值（Tom）
- 触发器现在按字母顺序触发（Tom）

- 新增 /contrib/adddepend 来处理 7.3 之前的对象依赖（Rod）
- 插入/更新值时允许更好的转换（Tom）

E.53.3.7. 工具命令

- 让 COPY TO 把嵌入的回车和换行输出为 \r 和 \n（Tom）
- 允许 COPY FROM 中的 DELIMITER 为 8 位干净（Tatsuo）
- 让 pg_dump 使用 ALTER TABLE ADD PRIMARY KEY 以提升性能（Neil）
- 禁用多语句规则中的方括号（Bruce）
- 禁止在函数内调用 VACUUM（Bruce）
- 允许 dropdb 和其他脚本使用带空格的标识符（Bruce）
- 把数据库注释更改限制在当前数据库
- 允许对操作符添加注释，独立于底层函数（Rod）
- 回退已中止事务中的 SET 命令（Tom）
- EXPLAIN 现在作为查询输出（Tom）
- 在 EXPLAIN 中显示条件表达式和排序键（Tom）
- 新增 'SET LOCAL var = value'，为单个事务设置配置变量（Tom）
- 允许 ANALYZE 在事务中运行（Bruce）
- 用新 WITH 子句改进 COPY 语法，保持向后兼容（Bruce）
- 修复 pg_dump 在非 ASCII 转储中一致输出标签（Bruce）
- 使转储文件中的外键约束更清晰（Rod）
- 新增 COMMENT ON CONSTRAINT（Rod）
- 允许 COPY TO/FROM 指定列名（Brent Verner）
- 把 UNIQUE 和 PRIMARY KEY 约束转储为 ALTER TABLE（Rod）
- 让 SHOW 输出查询结果（Joe）
- 对短 COPY 行产生失败而不是填充 NULL（Neil）
- 修复 CLUSTER 以保留所有表属性（Alvaro Herrera）
- 新 pg_settings 表，用于查看/修改 GUC 设置（Joe）
- 为 pg_dump 输出添加智能引用和可移植性改进（Peter）
- 把 serial 列转储为 SERIAL（Tom）
- 为 pg_dump 启用大文件支持（>2G）（Peter、Philip Warner、Bruce）
- 不允许对涉及参照约束的表使用 TRUNCATE（Rod）

- 让 TRUNCATE 也自动截断关系的 toast 表（Tom）
- 新增 clusterdb 工具，基于先前的 CLUSTER 操作自动聚簇整个数据库（Alvaro Herrera）
- 彻底改造 pg_dumpall（Peter）
- 允许对 TOAST 表执行 REINDEX（Tom）
- 按 SQL99 实现 START TRANSACTION（Neil）
- 修复页分裂影响批量删除时的罕见索引损坏（Tom）
- 修复继承的 ALTER TABLE ... ADD COLUMN（Alvaro Herrera）

E.53.3.8. 数据类型和函数

- 修复 factorial(0) 使其返回 1（Bruce）
- 日期/时间/时区改进（Thomas）
- 数组切片提取的修复（Tom）
- 修复 extract/date_part 为 timestamp 报告正确的微秒（Tatsuo）
- 允许 text_substr() 和 bytea_substr() 更高效地读取 TOAST 值（John Gray）
- 新增域支持（Rod）
- 使 WITHOUT TIME ZONE 成为 TIMESTAMP 和 TIME 数据类型的默认（Thomas）
- 允许用 configure 的 --enable-integer-datetimes 为日期/时间类型使用 64 位整数的替代存储方案（Thomas）
- 让 timezone(timestamptz) 返回 timestamp 而不是字符串（Thomas）
- 允许 1BC 之前日期的日期/时间类型带小数秒（Thomas）
- 把 timestamp 数据类型的精度限制为 6 位小数（Thomas）
- 把时区转换函数从 timetz() 改为 timezone()（Thomas）
- 新增配置变量 datestyle 和 timezone（Tom）
- 新增 OVERLAY()，允许替换字符串中的子串（Thomas）
- 新增 SIMILAR TO（Thomas、Tom）
- 新增正则表达式 SUBSTRING(string FROM pat FOR escape)（Thomas）
- 新增 LOCALTIME 和 LOCALTIMESTAMP 函数（Thomas）
- 用 CREATE TYPE typename AS (column) 新增命名组合类型（Joe）
- 允许在表别名子句中定义组合类型（Joe）
- 新增 API 以简化 C 语言表函数的创建（Joe）
- 从与标准不匹配的 SQL99 函数调用中移除 ODBC 兼容的空括号（Thomas）
- 允许 macaddr 数据类型接受无分隔符的 12 个十六进制数字（Mike Wyer）

- 新增 CREATE/DROP CAST（Peter）
- 新增 IS DISTINCT FROM 操作符（Thomas）
- 新增 SQL99 TREAT() 函数，CAST() 的同义词（Thomas）
- 新增 pg_backend_pid()，输出后端 pid（Bruce）
- 新增 IS OF / IS NOT OF 类型谓词（Thomas）
- 允许不完全指定长度的位串常量（Thomas）
- 允许 8 字节整数与位串之间的转换（Thomas）
- 实现十六进制字面量到位串字面量的转换（Thomas）
- 允许表函数出现在 FROM 子句中（Joe）
- 把函数参数的最大数量增加到 32（Bruce）
- 不再自动为 SERIAL 列创建索引（Tom）
- 新增 current_database()（Rod）
- 修复 cash_words() 使其不溢出缓冲区（Tom）
- 新增函数 replace()、split_part()、to_hex()（Joe）
- 修复 bytea 作为右侧参数的 LIKE（Joe）
- 防止 SELECT cash_out(2) 导致的崩溃（Tom）
- 修复 to_char(1,'FM999.99') 使其返回句点（Karel）
- 修复返回 OPAQUE 的触发器/类型/语言函数，使其返回正确的类型（Tom）

E.53.3.9. 国际化

- 新增编码：韩语（JOHAB）、泰语（WIN874）、越南语（TCVN）、阿拉伯语（WIN1256）、简体中文（GBK）、韩语（UHC）（Eiji Tokuya）
- 默认启用区域设置支持（Peter）
- 新增区域设置变量（Peter）
- 在 PQescapeBytea/PQunescapeBytea 中为多字节转义 >= 0x7f 的字节（Tatsuo）
- 为正则表达式字符类添加区域感知
- 默认启用多字节支持（Tatsuo）
- 新增 GB18030 多字节支持（Bill Huang）
- 新增 CREATE/DROP CONVERSION，允许可装载的编码（Tatsuo、Kaori）
- 新增 pg_conversion 表（Tatsuo）
- 新增 SQL99 CONVERT() 函数（Tatsuo）
- pg_dumpall、pg_controldata 和 pg_resetxlog 现在支持国家语言（Peter）

- 新增和更新的翻译

E.53.3.10. 服务器端语言

- 允许递归 SQL 函数（Peter）
- 更改 PL/Tcl 构建以使用配置的编译器和 Makefile.shlib（Peter）
- 彻底改造 PL/pgSQL 的 FOUND 变量以更兼容 Oracle（Neil、Tom）
- 允许 PL/pgSQL 处理带引号的标识符（Tom）
- 允许返回集合的 PL/pgSQL 函数（Neil）
- 使 PL/pgSQL 感知模式（Joe）
- 移除一些内存泄漏（Nigel J. Andrews、Tom）

E.53.3.11. psql

- 为兼容 7.2.0，psql \connect 不再把数据库名转为小写（Tom）
- 新增 psql \timing 来为用户查询计时（Greg Sabino Mullane）
- 让 psql \d 显示索引信息（Greg Sabino Mullane）
- 新 psql \dD 显示域（Jonathan Eisler）
- 允许 psql 显示视图上的规则（Paul ?）
- psql 变量替换的修复（Tom）
- 允许 psql \d 显示临时表结构（Tom）
- 允许 psql \d 显示外键（Rod）
- 修复 \? 以遵循 \pset pager（Bruce）
- 让 psql 在启动时报告其版本号（Tom）
- 允许 \copy 指定列名（Tom）

E.53.3.12. libpq

- 新增 ~/.pgpass 来存储主机/用户口令组合（Alvaro Herrera）
- 为 libpq 新增 PQunescapeBytea() 函数（Patrick Welche）
- 修复通过非阻塞连接发送大查询的问题（Bernhard Herzog）
- 修复 libpq 在 Win9X 上使用定时器的问题（David Ford）
- 允许 libpq notify 处理标识符长度不同的服务器（Tom）
- 把 libpq 的 PQescapeString() 和 PQescapeBytea() 添加到 Windows（Bruce）
- 修复非阻塞连接的 SSL 问题（Jack Bates）
- 新增 libpq 连接超时参数（Denis A Ustimenko）

E.53.3.13. JDBC

- 允许 JDBC 用 JDK 1.4 编译（Dave）
- 新增 JDBC 3 支持（Barry）
- 允许 JDBC 通过在连接 URL 中添加 ?loglevel=X 来设置 loglevel（Barry）
- 新增打印版本号的 Driver.info() 消息（Barry）
- 新增可更新结果集（Raghu Nidagal、Dave）
- 新增对可调用语句的支持（Paul Bethe）
- 新增查询取消能力
- 新增 refresh row（Dave）
- 修复多字节服务器的 MD5 加密处理（Jun Kawai）
- 新增对预备语句的支持（Barry）

E.53.3.14. 其他接口

- 修复 ECPG 关于单引号中八进制数的缺陷（Michael）
- 把 src/interfaces/libpgeasy 移到 http://gborg.postgresql.org（Marc、Bruce）
- 改进 Python 接口（Elliot Lee、Andrew Johnson、Greg Copeland）
- 新增 libpgtcl 连接关闭事件（Gerhard Hintermayer）
- 把 src/interfaces/libpq++ 移到 http://gborg.postgresql.org（Marc、Bruce）
- 把 src/interfaces/odbc 移到 http://gborg.postgresql.org（Marc）
- 把 src/interfaces/libpgeasy 移到 http://gborg.postgresql.org（Marc、Bruce）
- 把 src/interfaces/perl5 移到 http://gborg.postgresql.org（Marc、Bruce）
- 从主树移除 src/bin/pgaccess，现位于 http://www.pgaccess.org（Bruce）
- 为 libpgtcl 新增 pg_on_connection_loss 命令（Gerhard Hintermayer、Tom）

E.53.3.15. 源代码

- 并行 make 的修复（Peter）
- 链接 Tcl 的 AIX 修复（Andreas Zeugswetter）
- 允许 PL/Perl 在 Cygwin 下构建（Jason Tishler）
- 改进 MIPS 编译（Peter、Oliver Elphick）
- 要求 Autoconf 2.53 版本（Peter）
- configure 默认要求 readline 和 zlib（Peter）
- 允许 Solaris 使用紧密共享内存（ISM）以提升性能（Scott Brunza、P.J. Josh Rovero）

- 编译时总是启用 syslog，移除 --enable-syslog 选项（Tatsuo）
- 编译时总是启用多字节，移除 --enable-multibyte 选项（Tatsuo）
- 编译时总是启用区域设置，移除 --enable-locale 选项（Peter）
- Win9x DLL 创建的修复（Magnus Naeslund）
- 修复 WAL 代码在 Windows、BeOS 上使用 link() 的问题（Jason Tishler）
- 把 sys/types.h 添加到 c.h，从主文件移除（Peter、Bruce）
- 修复 SMP 机器上的 AIX 挂起（Tomoyuki Niijima）
- AIX SMP 挂起修复（Tomoyuki Niijima）
- 修复较新 glibc 库上 1970 年之前的日期处理（Tom）
- 修复 PowerPC SMP 锁定（Tom）
- 防止使用 gcc -ffast-math（Peter、Tom）
- 开发者构建现在要求 Bison >= 1.50
- Kerberos 5 支持现在能用 Heimdal 构建（Peter）
- 在用户指南中新增列出 SQL 特性的附录（Thomas）
- 改进可装载模块链接以使用 RTLD_NOW（Tom）
- 新错误级别 WARNING、INFO、LOG、DEBUG[1-5]（Bruce）
- 新 src/port 目录存放替代的 libc 函数（Peter、Bruce）
- 用于模式的新 pg_namespace 系统目录（Tom）
- 为模式新增 pg_class.relnamespace（Tom）
- 为模式新增 pg_type.typnamespace（Tom）
- 为模式新增 pg_proc.pronamespace（Tom）
- 重构聚合使其有 pg_proc 条目（Tom）
- 系统关系现在有自己的命名空间，不需要 pg_* 测试（Fernando Nasser）
- 把 TOAST 索引名改为 *_index 而不是 *_idx（Neil）
- 为操作符、操作符类新增命名空间（Tom）
- 为服务器控制文件新增额外检查（Thomas）
- 新波兰语 FAQ（Marcin Mazurek）
- 新增 Posix 信号量支持（Tom）
- 记录重新索引的需要（Bruce）
- 重命名一些内部标识符以简化 Windows 编译（Jan、Katherine Ward）

- 新增关于计算磁盘空间的文档（Bruce）
- 从 GUC 移除 KSQO（Bruce）
- 修复 rtree 中的内存泄漏（Kenneth Been）
- 为一致性修改一些错误消息（Bruce）
- 移除未使用的系统表列（Peter）
- 在适当处使系统列为 NOT NULL（Tom）
- 清理 sprintf 的使用，改用 snprintf()（Neil、Jukka Holappa）
- 移除 OPAQUE 并创建特定子类型（Tom）
- 数组内部处理的清理（Joe、Tom）
- 不允许 pg_atoi('')（Bruce）
- 移除参数 wal_files，因为 WAL 文件现在被回收（Bruce）
- 为堆页新增版本号（Tom）

E.53.3.16. 附加模块

- 允许 /contrib/array 中的 inet 数组（Neil）
- GiST 修复（Teodor Sigaev、Neil）
- 升级 /contrib/mysql
- 新增 /contrib/dbsize，无需清理即可显示表大小（Peter）
- 新增 /contrib/intagg，整数聚合例程（mlw）
- 改进 /contrib/oid2name（Neil、Bruce）
- 改进 /contrib/tsearch（Oleg、Teodor Sigaev）
- /contrib/rserver 的清理（Alexey V. Borzov）
- 更新 /contrib/oracle 转换工具（Gilles Darold）
- 更新 /contrib/dblink（Joe）
- 改进 /contrib/vacuumlo 支持的选项（Mario Weilguni）
- /contrib/intarray 的改进（Oleg、Teodor Sigaev、Andrey Oktyabrski）
- 新增 /contrib/reindexdb 工具（Shaun Thomas）
- 为 /contrib/isbn_issn 新增索引（Dan Weston）
- 新增 /contrib/dbmirror（Steven Singer）
- 改进 /contrib/pgbench（Neil）
- 新增 /contrib/tablefunc 表函数示例（Joe）

- 新增用于树结构的 /contrib/ltree 数据类型（Teodor Sigaev、Oleg Bartunov）
- 把 /contrib/pg_controldata、pg_resetxlog 移入主树（Bruce）
- /contrib/cube 的修复（Bruno Wolff）
- 改进 /contrib/fulltextindex（Christopher）

E.54. 版本 7.2.8

发行日期
2005-05-09

此发行版包含 7.2.7 之后的多种修复，包括一个与安全相关的问题。

E.54.1. 迁移到版本 7.2.8

对于运行 7.2.X 的用户，不需要进行转储/恢复。

E.54.2. 变更

- 修复古老的竞态条件，它允许事务在某些用途（如 SELECT FOR UPDATE）上比其他用途稍早被视为已提交

  这是一个极其严重的缺陷，因为它可能导致应用程序短暂看到表面的数据不一致。

- 修复关系扩展与 VACUUM 之间的竞态条件

  这在理论上可能导致丢失一页刚插入的数据，尽管该情形概率似乎很低。
  没有已知案例造成超过断言失败的后果。

- 修复 `TIME WITH TIME ZONE` 值的 `EXTRACT(EPOCH)`
- plpgsql 中的额外缓冲区溢出检查（Neil）
- 修复 pg_dump 正确转储包含 % 的索引名和触发器名（Neil）
- 防止 `to_char(interval)` 在月份相关格式上转储核心
- 修复 `contrib/pgcrypto` 在较新 OpenSSL 构建上的问题（Marko Kreen）

E.55. 版本 7.2.7

发行日期
2005-01-31

此发行版包含 7.2.6 之后的多种修复，包括几个与安全相关的问题。

E.55.1. 迁移到版本 7.2.7

对于运行 7.2.X 的用户，不需要进行转储/恢复。

E.55.2. 变更

- 不允许非超级用户使用 `LOAD`

  在会自动执行共享库初始化函数的平台（至少包括 Windows 和基于 ELF 的 Unix）上，`LOAD` 可被用来让服务器执行任意代码。感谢 NGS Software 报告此问题。

- 为一些 contrib 函数添加所需的 STRICT 标记（Kris Jurka）

- 避免 plpgsql 游标声明参数过多时的缓冲区溢出（Neil）

- 修复 FULL 和 RIGHT 外连接的规划错误

  连接的结果被错误地认为与左输入的排序相同。这不仅会向用户交付排序错误的输出，而且在嵌套归并连接的情形下可能给出完全错误的答案。

- 修复 SQL 和 GERMAN datestyle 下负 interval 的显示

E.56. 版本 7.2.6

发行日期
2004-10-22

此发行版包含 7.2.5 之后的多种修复。

E.56.1. 迁移到版本 7.2.6

对于运行 7.2.X 的用户，不需要进行转储/恢复。

E.56.2. 变更

- 修复磁盘上提示位可能未更新的问题

  极少数情况下，这一疏漏可能导致“无法访问事务状态”失败，这使它成为一个潜在的数据丢失缺陷。

- 确保哈希外连接不漏掉元组

  在恰当的数据分布下，使用哈希连接计划的超大型左连接可能无法输出未匹配的左侧行。

- 不允许以 root 运行 pg_ctl

  这是为了防范任何可能的安全问题。

- 避免 make_oidjoins_check 在 /tmp 中使用临时文件

  这被报告为安全问题，不过几乎不值得担心，因为非开发者本来就没有理由使用此脚本。

- 更新到更新版本的 Bison

E.57. 版本 7.2.5

发行日期
2004-08-16

此发行版包含 7.2.4 之后的多种修复。

E.57.1. 迁移到版本 7.2.5

对于运行 7.2.X 的用户，不需要进行转储/恢复。

E.57.2. 变更

- 防止崩溃期间已提交事务的可能丢失

  由于事务提交与检查点之间的互锁不足，
  最近一次检查点之前刚提交的事务可能在数据库崩溃和重启后全部或部分丢失。这是自 PostgreSQL 7.1 以来就存在的严重缺陷。

- 修复与首个根页分裂并行的 btree 搜索的极端情形
- 修复 `to_ascii` 中的缓冲区溢出（Guido Notari）
- 修复 char 为无符号的机器上死锁检测的核心转储
- 修复 Async_NotifyHandler 运行后不响应 `pg_ctl stop -m fast` 的问题
- 修复 pg_dump 中的内存泄漏
- 避免与 `isblank()` 函数或宏的系统定义冲突

E.58. 版本 7.2.4

发行日期
2003-01-30

此发行版包含 7.2.3 版本的多种修复，包括防止可能数据丢失的修复。

E.58.1. 迁移到版本 7.2.4

对于运行 7.2.* 版本的用户，不需要进行转储/恢复。

E.58.2. 变更

- 修复 VACUUM "No one parent tuple was found" 错误的一些额外情形

- 禁止在函数内调用 VACUUM（Bruce）
- 确保在标记检查点完成之前 pg_clog 更新已同步到磁盘
- 避免大型哈希连接期间的整数溢出
- 使 GROUP 命令在 pg_group.grolist 大到需要 TOAST 时也能工作
- 修复日期时间表中的错误；一些时区名未被识别
- 修复 circle_poly()、path_encode()、path_add() 中的整数溢出（Neil）
- 修复 lseg_eq()、lseg_ne()、lseg_center() 中长期存在的逻辑错误

E.59. 版本 7.2.3

发行日期
2002-10-01

此发行版包含 7.2.2 版本的多种修复，包括防止可能数据丢失的修复。

E.59.1. 迁移到版本 7.2.3

对于运行 7.2.* 版本的用户，不需要进行转储/恢复。

E.59.2. 变更

- 防止压缩事务日志的可能丢失（Tom）
- 防止非超级用户推进最近的清理信息（Tom）
- 处理较新版本 glibc 中 1970 年之前的日期值（Tom）
- 修复服务器关闭期间可能的挂起
- 防止 SMP PPC 机器上的自旋锁挂起（Tomoyuki Niijima）
- 修复 pg_dump 正确转储 FULL JOIN USING（Tom）

E.60. 版本 7.2.2

发行日期
2002-08-23

此发行版包含 7.2.1 版本的多种修复。

E.60.1. 迁移到版本 7.2.2

对于运行 7.2.* 版本的用户，不需要进行转储/恢复。

E.60.2. 变更

- 允许在 PL/pgSQL 中 EXECUTE "CREATE TABLE AS ... SELECT"（Tom）
- 修复压缩事务日志 id 回卷的问题（Tom）
- 修复 PQescapeBytea/PQunescapeBytea 使其处理 > 0x7f 的字节（Tatsuo）
- 修复 psql 和 pg_dump 以不存在的长选项调用时的崩溃（Tatsuo）
- 修复调用几何操作符时的崩溃（Tom）
- 允许 OPEN cursor(args)（Tom）
- rtree_gist 索引构建的修复（Teodor）
- 转储用户定义聚合的修复（Tom）
- contrib/intarray 修复（Oleg）
- 修复使用括号的复杂 UNION/EXCEPT/INTERSECT 查询（Tom）
- pg_convert 的修复（Tatsuo）
- 长 DATA 字符串崩溃的修复（Thomas、Neil）
- repeat()、lpad()、rpad() 与长字符串的修复（Neil）

E.61. 版本 7.2.1

发行日期
2002-03-21

此发行版包含 7.2 版本的多种修复。

E.61.1. 迁移到版本 7.2.1

对于运行 7.2 版本的用户，不需要进行转储/恢复。

E.61.2. 变更

- 确保序列计数器在崩溃后不会倒退（Tom）
- 修复 pgaccess 汉字转换键绑定（Tatsuo）
- 优化器改进（Tom）
- Cash I/O 改进（Tom）
- 新俄语 FAQ
- 缺失 AuthBlockSig 的编译修复（Heiko）
- 附加时区和时区修复（Thomas）

- 允许 psql \connect 处理大小写混合的数据库和用户名（Tom）
- 即使有 ON INSERT 规则在命令完成时也返回正确的 OID（Tom）
- 允许 COPY FROM 使用 8 位 DELIMITERS（Tatsuo）
- 修复 extract/date_part 对毫秒/微秒的缺陷（Tatsuo）
- 改进多个不同长度 UNION 的处理（Tom）
- contrib/btree_gist 改进（Teodor Sigaev）
- contrib/tsearch 词典改进，附加安装步骤见 README.tsearch（Thomas T. Thai、Teodor Sigaev）
- 数组下标处理的修复（Tom）
- 允许在 PL/pgSQL 中 EXECUTE "CREATE TABLE AS ... SELECT"（Tom）

E.62. 版本 7.2

发行日期
2002-02-04

E.62.1. 概述

此发行版改进了 PostgreSQL 在高流量应用程序中的使用。

此发行版的主要变更：

VACUUM

清理不再锁定表，因此允许在清理期间正常的用户访问。新 `VACUUM FULL` 命令通过锁定表并收缩表的磁盘副本执行旧式清理。

Transactions

超过四十亿事务的安装不再有问题。

OIDs

OID 现在是可选的。用户现在可以在 OID 使用过多的情形下创建不带 OID 的表。

Optimizer

系统现在在 `ANALYZE` 期间计算直方图列统计，使优化器的选择好得多。

Security

新的 MD5 加密选项允许更安全地存储和传输口令。Linux 和 BSD 系统上有新的 Unix 域套接字认证选项。

Statistics

管理员可以使用新的表访问统计模块获取表和索引使用的细粒度信息。

Internationalization

程序和库消息现在可以用多种语言显示。

E.62.2. 迁移到版本 7.2

希望从任何以前的版本迁移数据的用户，需要使用 `pg_dump` 进行转储/恢复。

请注意以下不兼容之处：

- `VACUUM` 命令的语义在此版本中已更改。你可能需要相应地更新维护程序。
- 在此版本中，使用 `= NULL` 的比较总是返回假（更准确地说是 NULL）。以前的版本自动把此语法转换为 `IS NULL`。旧行为可以用 `postgresql.conf` 参数重新启用。
- `pg_hba.conf` 和 `pg_ident.conf` 配置现在只在收到 SIGHUP 信号后重新装载，而不是每次连接时。
- 函数 `octet_length()` 现在返回未压缩的数据长度。
- 日期/时间值 `'current'` 不再可用。你需要重写你的应用程序。
- `timestamp()`、`time()` 和 `interval()` 函数不再可用。不要用 `timestamp()`，请改用 `timestamp 'string'` 或 `CAST`。

`SELECT ... LIMIT #,#` 语法将在下一个版本中移除。你应把查询改为使用独立的 LIMIT 和 OFFSET 子句，例如 `LIMIT 10 OFFSET 20`。

E.62.3. 变更

E.62.3.1. 服务器操作

- 在单独的目录中创建临时文件（Bruce）
- postmaster 启动时删除孤立的临时文件（Bruce）
- 为一些系统表添加唯一索引（Tom）
- 系统表操作符重组（Oleg Bartunov、Teodor Sigaev、Tom）
- 把 pg_log 改名为 pg_clog（Tom）
- 启用 SIGTERM、SIGQUIT 来终止后端（Jan）
- 移除后端数量的编译期限制（Tom）
- 信号量资源失败时更好的清理（Tatsuo、Tom）
- 允许安全的事务 ID 回卷（Tom）
- 从一些系统表移除 OID（Tom）
- 移除"触发数据更改违反"错误检查（Tom）
- 预备/保存计划的 SPI 门户创建（Jan）
- 允许 SPI 列函数用于系统列（Tom）

- 长值压缩改进（Tom）
- 表、索引访问的统计收集器（Jan）
- 把超长序列名截断为合理的值（Tom）
- 以毫秒测量事务时间（Thomas）
- 修复 TID 顺序扫描（Hiroshi）
- 超级用户 ID 现在固定为 1（Peter E）
- pg_ctl 新 "reload" 选项（Tom）

E.62.3.2. 性能

- 优化器改进（Tom）
- 优化器的新直方图列统计（Tom）
- 重用预写日志文件而不是丢弃它们（Tom）
- 缓存改进（Tom）
- IS NULL、IS NOT NULL 优化器改进（Tom）
- 改进锁管理器以减少锁争用（Tom）
- 为索引访问支持函数保留 relcache 条目（Tom）
- 使 NUMERIC 中的 NaN 和无穷大有更好的选择率（Tom）
- R-tree 性能改进（Kenneth Been）
- B-tree 分裂更高效（Tom）

E.62.3.3. 权限

- 把 UPDATE、DELETE 权限改为彼此独立（Peter E）
- 新 REFERENCES、TRIGGER 权限（Peter E）
- 允许一次对多个用户 GRANT/REVOKE（Peter E）
- 新 has_table_privilege() 函数（Joe Conway）
- 允许非超级用户清理数据库（Tom）
- 新 SET SESSION AUTHORIZATION 命令（Peter E）
- 修复新建表上权限修改的缺陷（Tom）
- 禁止非超级用户访问 pg_statistic，新增用户可访问的视图（Tom）

E.62.3.4. 客户端认证

- 在认证前 fork postmaster 以防止挂起（Peter E）

- 在 Linux、*BSD 上通过 Unix 域套接字新增 ident 认证（Helge Bahmann、Oliver Elphick、Teodor Sigaev、Bruce）
- 新增使用 MD5 加密的口令认证方法（Bruce）
- 允许用 MD5 加密存储的口令（Bruce）
- PAM 认证（Dominic J. Eidson）
- 只在启动和 SIGHUP 时装载 pg_hba.conf 和 pg_ident.conf（Bruce）

E.62.3.5. 服务器配置

- 一些时区缩写解释为澳大利亚而不是北美，现在可在运行时设置（Bruce）
- 设置默认事务隔离级别的新参数（Peter E）
- 启用把 "expr = NULL" 转换为 "expr IS NULL" 的新参数，默认关闭（Peter E）
- 控制 VACUUM 内存使用的新参数（Tom）
- 设置客户端认证超时的新参数（Tom）
- 设置最大打开文件数的新参数（Tom）

E.62.3.6. 查询

- INSERT 规则添加的语句现在在 INSERT 之后执行（Jan）
- 阻止目标列表中不加修饰的关系名（Bruce）
- ORDER BY 中 NULL 现在排在所有正常值之后（Tom）
- 新 IS UNKNOWN、IS NOT UNKNOWN 布尔测试（Tom）
- 新 SHARE UPDATE EXCLUSIVE 锁模式（Tom）
- 新 EXPLAIN ANALYZE 命令，显示运行时间和行数（Martijn van Oosterhout）
- 修复 LIMIT 与子查询的问题（Tom）
- 修复推入子查询的 LIMIT、DISTINCT ON（Tom）
- 修复嵌套的 EXCEPT/INTERSECT（Tom）

E.62.3.7. 模式操作

- 修复临时表中的 SERIAL（Bruce）
- 允许临时序列（Bruce）
- 序列现在内部使用 int8（Tom）
- 新 SERIAL8 创建带序列的 int8 列，默认仍为 SERIAL4（Tom）
- 用 WITHOUT OIDS 使 OID 可选（Tom）
- 为 CREATE TYPE 新增 %TYPE 语法（Ian Lance Taylor）

- 为 CHECK 约束新增 ALTER TABLE / DROP CONSTRAINT（Christopher Kings-Lynne）
- 新 CREATE OR REPLACE FUNCTION 更改现有函数（保留函数 OID）（Gavin Sherry）
- 新增 ALTER TABLE / ADD [UNIQUE | PRIMARY]（Christopher Kings-Lynne）
- 允许视图中重命名列
- 让 ALTER TABLE / RENAME COLUMN 更新索引的列名（Brent Verner）
- 修复继承表的 ALTER TABLE / ADD CONSTRAINT ... CHECK（Stephan Szabo）
- ALTER TABLE RENAME 正确更新外键触发器参数（Brent Verner）
- DROP AGGREGATE 和 COMMENT ON AGGREGATE 现在接受 aggtype（Tom）
- 为 SQL 函数新增自动返回类型数据转换（Tom）
- 允许 GiST 索引处理 NULL 和多键索引（Oleg Bartunov、Teodor Sigaev、Tom）
- 启用部分索引（Martijn van Oosterhout）

E.62.3.8. 工具命令

- 新增 RESET ALL、SHOW ALL（Marko Kreen）
- CREATE/ALTER USER/GROUP 现在允许选项任意顺序（Vince）
- 新增 LOCK A, B, C 功能（Neil Padgett）
- CREATE/ALTER USER 的新 ENCRYPTED/UNENCRYPTED 选项（Bruce）
- 新的轻量 VACUUM 不锁定表；旧语义以 VACUUM FULL 提供（Tom）
- 禁用视图上的 COPY TO/FROM（Bruce）
- COPY DELIMITERS 字符串必须恰好一个字符（Tom）
- 关于索引元组少于堆的 VACUUM 警告现在只在适当时出现（Martijn van Oosterhout）
- 修复 CREATE INDEX 的权限检查（Tom）
- 禁止不当使用 CREATE/DROP INDEX/TRIGGER/VIEW（Tom）

E.62.3.9. 数据类型和函数

- SUM()、AVG()、COUNT() 现在内部使用 int8 以提升速度（Tom）
- 新增 convert()、convert2()（Tatsuo）
- 新函数 bit_length()（Peter E）
- 使 CHAR(n)/VARCHAR(n) 中的 "n" 表示字符数而不是字节数（Tatsuo）
- CHAR()、VARCHAR() 现在拒绝过长的字符串（Peter E）
- BIT VARYING 现在拒绝过长的位串（Peter E）
- BIT 现在拒绝与声明大小不匹配的位串（Peter E）

- INET、CIDR 文本转换函数（Alex Pilosov）
- INET、CIDR 操作符 << 和 <<= 可索引（Alex Pilosov）
- Bytea \### 现在要求有效的三位八进制数
- Bytea 比较改进，现在支持 =、<>、>、>=、< 和 <=
- Bytea 现在支持 B-tree 索引
- Bytea 现在支持 LIKE、LIKE...ESCAPE、NOT LIKE、NOT LIKE...ESCAPE
- Bytea 现在支持连接
- 新 bytea 函数：position、substring、trim、btrim 和 length
- 新 encode() 函数模式 "escaped"，在最小转义的 bytea 与 text 之间转换
- 新增 pg_database_encoding_max_length()（Tatsuo）
- 新增 pg_client_encoding() 函数（Tatsuo）
- now() 返回毫秒精度的时间（Thomas）
- 新 TIMESTAMP WITHOUT TIMEZONE 数据类型（Thomas）
- 新增带 "T" 的 ISO 日期/时间规格，yyyy-mm-ddThh:mm:ss（Thomas）
- 新 xid/int 比较函数（Hiroshi）
- 为 TIME、TIMESTAMP 和 INTERVAL 数据类型新增精度（Thomas）
- 修改类型强制逻辑，先尝试二进制兼容函数（Tom）
- 新 encode() 函数默认安装（Marko Kreen）
- 改进的 to_*() 转换函数（Karel Zak）
- 优化单字节编码下的 LIKE/ILIKE（Tatsuo）
- contrib/pgcrypto 中的新函数：crypt()、hmac()、encrypt()、gen_salt()（Marko Kreen）
- 修正 translate() 函数的描述（Bruce）
- 为 SET TIME ZONE 新增 INTERVAL 参数（Thomas）
- 新增 INTERVAL YEAR TO MONTH（等）语法（Thomas）
- 优化单字节编码下的长度函数（Tatsuo）
- 修复 path_inter、path_distance、path_length、dist_ppath 以处理闭合路径（Curtis Barrett、Tom）
- octet_length(text) 现在返回未压缩的长度（Tatsuo、Bruce）
- 处理日期/时间字面量中的 "July" 全名（Greg Sabino Mullane）
- 一些 datatype() 函数调用现在的求值方式不同
- 新增对儒略和 ISO 时间规格的支持（Thomas）

E.62.3.10. 国际化

- psql、pg_dump、libpq 和服务器的国家语言支持（Peter E）
- 中文（简体、繁体）、捷克语、法语、德语、匈牙利语、俄语、瑞典语的消息翻译（Peter E、Serguei A. Mokhov、Karel Zak、Weiping He、Zhenbang Wei、Kovacs Zoltan）
- 使 trim、ltrim、rtrim、btrim、lpad、rpad、translate 感知多字节（Tatsuo）
- 新增 LATIN5,6,7,8,9,10 支持（Tatsuo）
- 新增 ISO 8859-5,6,7,8 支持（Tatsuo）
- 更正 LATIN5 为 ISO-8859-9 而不是 ISO-8859-5（Tatsuo）
- 使 mic2ascii() 不感知非 ASCII（Tatsuo）
- 拒绝无效的多字节字符序列（Tatsuo）

E.62.3.11. PL/pgSQL

- 现在 SELECT 循环使用门户，允许超大结果集（Jan）
- CURSOR 和 REFCURSOR 支持（Jan）
- 现在可以返回打开的游标（Jan）
- 新增 ELSEIF（Klaus Reger）
- 改进 PL/pgSQL 错误报告，包括错误位置（Tom）
- 为兼容允许游标声明中的 IS 或 FOR 关键字（Bruce）
- SELECT ... FOR UPDATE 的修复（Tom）
- 返回多行的 PERFORM 的修复（Tom）
- 让 PL/pgSQL 使用服务器的类型强制代码（Tom）
- 内存泄漏修复（Jan、Tom）
- 使尾部分号可选（Tom）

E.62.3.12. PL/Perl

- 新不可信 PL/Perl（Alex Pilosov）
- 即使 libperl 不是共享的，PL/Perl 现在也能在一些平台上构建（Peter E）

E.62.3.13. PL/Tcl

- 现在报告 errorInfo（Vsevolod Lobko）
- 新增 spi_lastoid 函数（bob@redivi.com）

E.62.3.14. PL/Python

- ……是新的（Andrew Bosma）

E.62.3.15. psql

- \d 按唯一、主键分组显示索引（Christopher Kings-Lynne）
- 允许反斜杠命令中的尾部分号（Greg Sabino Mullane）
- 如果可能从 /dev/tty 读取口令
- 更改用户和数据库时强制新的口令提示（Tatsuo、Tom）
- 为 Unicode 格式化正确的列数（Patrice）

E.62.3.16. libpq

- 新函数 PQescapeString()，转义命令字符串中的引号（Florian Weimer）
- 新函数 PQescapeBytea()，转义二进制字符串以用作 SQL 字符串字面量

E.62.3.17. JDBC

- 返回 INSERT 的 OID（Ken K）
- 处理更多数据类型（Ken K）
- 处理字符串中的单引号和换行（Ken K）
- 处理 NULL 变量（Ken K）
- 时区处理的修复（Barry Lind）
- 改进的 Druid 支持
- 允许非多字节服务器使用 8 位字符（Barry Lind）
- 支持 BIT、BINARY 类型（Ned Wolpert）
- 降低内存使用（Michael Stephens、Dave Cramer）
- 更新 DatabaseMetaData（Peter E）
- 新增 DatabaseMetaData.getCatalogs()（Peter E）
- 编码修复（Anders Bengtsson）
- Get/setCatalog 方法（Jason Davies）
- DatabaseMetaData.getColumns() 现在返回列默认值（Jason Davies）
- DatabaseMetaData.getColumns() 性能改进（Jeroen van Vianen）
- 一些 JDBC1 和 JDBC2 的合并（Anders Bengtsson）
- 事务性能改进（Barry Lind）
- 数组修复（Greg Zoller）
- 序列化添加
- 批处理修复（Rene Pijlman）

- ExecSQL 方法重组（Anders Bengtsson）
- GetColumn() 修复（Jeroen van Vianen）
- 修复 isWriteable() 函数（Rene Pijlman）
- 更好地通过 JDBC2 一致性测试（Rene Pijlman）
- 新增 bytea 类型能力（Barry Lind）
- 新增 isNullable()（Rene Pijlman）
- JDBC 日期/时间测试套件修复（Liam Stewart）
- SELECT 'id' AS xxx FROM table 的修复（Dave Cramer）
- 修复 DatabaseMetaData 以正确显示精度（Mark Lillywhite）
- 新 getImported/getExported 键（Jason Davies）
- MD5 口令加密支持（Jeremy Wohl）
- 修复为真正使用类型缓存（Ned Wolpert）

E.62.3.18. ODBC

- 移除查询大小限制（Hiroshi）
- 移除文本字段大小限制（Hiroshi）
- 多字节模式下 SQLPrimaryKeys 的修复（Hiroshi）
- 允许 ODBC 过程调用（Hiroshi）
- 改进布尔处理（Aidan Mountford）
- 大多数配置选项现在可通过 DSN 设置（Hiroshi）
- 多字节、性能修复（Hiroshi）
- 允许驱动与 iODBC 或 unixODBC 一起使用（Peter E）
- MD5 口令加密支持（Bruce）
- 为 odbc.sql 添加更多兼容函数（Peter E）

E.62.3.19. ECPG

- 实现 EXECUTE ... INTO（Christof Petig）
- 多行描述符支持（如 CARDINALITY）（Christof Petig）
- GRANT 参数的修复（Lee Kindness）
- 修复 INITIALLY DEFERRED 缺陷
- 各种缺陷修复（Michael、Christof Petig）
- 指示符变量数组的自动分配（int *ind_p=NULL）

- 字符串数组的自动分配（char **foo_pp=NULL）
- 修复 ECPGfree_auto_mem
- 所有有外部链接的函数名现在都以 ECPG 为前缀
- 结构数组的修复（Michael）

E.62.3.20. 其他接口

- Python 修复 fetchone()（Gerhard Haring）
- 在适当处在 Tcl 中使用 UTF、Unicode（Vsevolod Lobko、Reinhard Max）
- 新增 Tcl COPY TO/FROM（ljb）
- 防止 pg_dump 输出默认索引操作符类（Tom）
- 修复 libpgeasy 内存泄漏（Bruce）

E.62.3.21. 构建和安装

- Configure、动态装载器和共享库修复（Peter E）
- QNX 4 移植的修复（Bernd Tegge）
- Cygwin 和 Windows 移植的修复（Jason Tishler、Gerhard Haring、Dmitry Yurtaev、Darko Prenosil、Mikhail Terekhov）
- Windows 套接字通信失败的修复（Magnus、Mikhail Terekhov）
- Hurd 编译修复（Oliver Elphick）
- BeOS 修复（Cyril Velter）
- 移除 configure --enable-unicode-conversion，现由多字节启用（Tatsuo）
- AIX 修复（Tatsuo、Andreas）
- 修复并行 make（Peter E）
- 把 SQL 语言手册页安装到操作系统特定的目录（Peter E）
- 把 config.h 改名为 pg_config.h（Peter E）
- 重新组织头文件的安装布局（Peter E）

E.62.3.22. 源代码

- 移除 SEP_CHAR（Bruce）
- 新 GUC 钩子（Tom）
- 合并 GUC 和命令行处理（Marko Kreen）
- 移除 EXTEND INDEX（Martijn van Oosterhout、Tom）
- 新的缩进 java 代码的 pgjindent 工具（Bruce）

- 在 C++ 下编译时移除 true/false 的定义（Leandro Fanzone、Tom）
- pgindent 修复（Bruce、Tom）
- 在适当处用 strcmp() 取代 strcasecmp()（Peter E）
- Dynahash 可移植性改进（Tom）
- 在自旋锁结构中添加 'volatile' 用法
- 改进信号处理逻辑（Tom）

E.62.3.23. 附加模块

- 新 contrib/rtree_gist（Oleg Bartunov、Teodor Sigaev）
- 新 contrib/tsearch 全文索引（Oleg、Teodor Sigaev）
- 新增用于远程数据库访问的 contrib/dblink（Joe Conway）
- contrib/ora2pg Oracle 转换工具（Gilles Darold）
- contrib/xml XML 转换工具（John Gray）
- contrib/fulltextindex 修复（Christopher Kings-Lynne）
- 新 contrib/fuzzystrmatch，含 levenshtein 和 metaphone，合并了 soundex（Joe Conway）
- 为 contrib/intarray 新增布尔查询、二分搜索和修复（Oleg Bartunov）
- 新 pg_upgrade 工具（Bruce）
- 新增 pg_resetxlog 选项（Bruce、Tom）

E.63. 版本 7.1.3

发行日期
2001-08-15

E.63.1. 迁移到版本 7.1.3

对于运行 7.1.X 的用户，不需要进行转储/恢复。

E.63.2. 变更

移除大事务未使用的 WAL 段（Tom）
多动作规则修复（Tom）
PL/pgSQL 内存分配修复（Jan）
VACUUM 缓冲区修复（Tom）
回归测试修复（Tom）
pg_dump 对视图上 GRANT/REVOKE/注释及用户定义类型的修复（Tom）

修复带 DISTINCT ON 或 LIMIT 的子查询 (Tom)
BeOS 修复
禁用视图上的 COPY TO/FROM (Tom)
Cygwin 构建 (Jason Tishler)

E.64. 版本 7.1.2

发行日期
2001-05-11

这包含 7.1.1 之后的一个修复。

E.64.1. 迁移到版本 7.1.2

对于运行 7.1.X 的用户，不需要进行转储/恢复。

E.64.2. 变更

修复返回零行时的 PL/pgSQL SELECT
修复 psql 反斜杠核心转储
参照完整性权限修复
优化器修复
pg_dump 清理

E.65. 版本 7.1.1

发行日期
2001-05-05

这包含 7.1 之后的多种修复。

E.65.1. 迁移到版本 7.1.1

对于运行 7.1 的用户，不需要进行转储/恢复。

E.65.2. 变更

修复 numeric MODULO 操作符 (Tom)
pg_dump 修复 (Philip)
pg_dump 可转储 7.0 数据库 (Philip)
readline 4.2 修复 (Peter E)
JOIN 修复 (Tom)
AIX、MSWIN、VAX、N32K 修复 (Tom)
多字节修复 (Tom)
Unicode 修复 (Tatsuo)

优化器改进（Tom）
修复函数中的整行（Tom）
修复 pg_ctl 及带空格的选项字符串（Peter E）
ODBC 修复（Hiroshi）
EXTRACT 现在可接受字符串参数（Thomas）
Python 修复（Darcy）

E.66. 版本 7.1

发行日期
2001-04-13

此发行版专注于移除 PostgreSQL 代码中存在多年的限制。

此发行版的主要变更：

Write-ahead Log (WAL)

为了在操作系统崩溃时保持数据库一致性，以前的 PostgreSQL 版本强制在每次事务提交前把所有数据修改写到磁盘。有了 WAL，只需把一个日志文件刷到磁盘，大幅改进了性能。如果你在以前的版本中一直用 -F 禁用磁盘刷写，可能要考虑停止使用它。

TOAST

TOAST——以前的版本有编译期行长度限制，通常为 8k - 32k。这一限制使长文本字段的存储很困难。有了 TOAST，任意长度的长行都能以良好的性能存储。

Outer Joins

我们现在支持外连接。不再需要用 UNION/NOT IN 变通实现外连接。我们使用 SQL92 外连接语法。

Function Manager

以前的 C 函数管理器不能正确处理空值，也不支持 64 位 CPU（Alpha）。新的函数管理器可以。你可以继续使用旧的自定义函数，
但将来可能想重写它们以使用新的函数管理器调用接口。

Complex Queries

以前版本不支持的大量复杂查询现在可以工作。视图、聚合、UNION、LIMIT、游标、
子查询和继承表的许多组合现在都能正常工作。继承表现在默认被访问。FROM 中的子查询现在受支持。

E.66.1. 迁移到版本 7.1

希望从任何以前的版本迁移数据的用户，需要使用 pg_dump 进行转储/恢复。

E.66.2. 变更

缺陷修复

大量多字节/Unicode/区域设置修复 (Tatsuo 等)
更可靠的 ALTER TABLE RENAME (Tom)
Kerberos V 修复 (David Wragg)
修复目标列表带子查询的 INSERT INTO...SELECT (Tom)
在标准错误上提示用户名/口令 (Bruce)
大对象 inv_read/inv_write 修复 (Tom)
to_char()、to_date()、to_ascii() 和 to_timestamp() 的修复 (Karel、Daniel Baldoni)
防止查询表达式泄漏内存 (Tom)
允许更新数组元素 (Tom)
取消时唤醒锁等待者 (Hiroshi)
修复使用哈希连接时游标的罕见崩溃 (Tom)
修复回滚事务中的 DROP TABLE/INDEX (Hiroshi)
修复启用 MULTIBYTE 时 \l+ 导致 psql 崩溃的问题 (Peter E)
修复 CREATE VIEW 期间规则名截断的问题 (Ross Reedstrom)
修复 PL/perl (Alex Kapranoff)
禁止视图上的 LOCK (Mark Hollomon)
禁止视图上的 INSERT/UPDATE/DELETE (Mark Hollomon)
禁止视图上的 DROP RULE、CREATE INDEX、TRUNCATE (Mark Hollomon)
允许 PL/pgSQL 接受非 ASCII 标识符 (Tatsuo)
允许视图正确处理 GROUP BY、聚合、DISTINCT (Tom)
修复 TRUNCATE 命令的罕见失败 (Tom)
允许 UNION/INTERSECT/EXCEPT 与 ALL、子查询、视图、DISTINCT、ORDER BY、SELECT...INTO 一起使用 (Tom)
修复中止事务期间的解析器失败 (Tom)
允许临时关系正确清理索引 (Bruce)
修复 VACUUM 在同页移动行的问题 (Tom)
修改 pg_dump 以更好地处理 template1 中的用户定义项 (Philip)
允许 VIEW 中使用 LIMIT (Tom)
要求游标 FETCH 遵循 LIMIT (Tom)
允许继承列上的 PRIMARY/FOREIGN KEY 定义 (Stephan)
允许子查询中的 ORDER BY、LIMIT (Tom)
允许 CREATE RULE 中的 UNION (Tom)
使 ALTER/DROP TABLE 可回滚 (Vadim、Tom)
把 initdb 的排序规则存储在 pg_control 中使排序规则不可更改 (Tom)
修复带规则的 INSERT...SELECT (Tom)
修复视图和子查询内的 FOR UPDATE (Tom)
修复 OVERLAPS 操作符在 NULL 方面符合 SQL92 规范 (Tom)
修复 lpad() 和 rpad() 以处理长度小于输入字符串的情形 (Tom)
修复某些规则中 NOTIFY 的使用 (Tom)
彻底改造 btree 代码 (Tom)
修复 PL/pgSQL 变量中 NOT NULL 的使用 (Tom)
彻底改造 GIST 代码 (Oleg)
修复 CLUSTER 以保留约束和列默认值 (Tom)
改进的死锁检测处理 (Tom)
允许表中有多个 SERIAL 列 (Tom)
防止偶发索引损坏 (Vadim)

增强

添加 OUTER JOIN (Tom)
函数管理器彻底改造 (Tom)
允许对索引 ALTER TABLE RENAME (Tom)

改进 CLUSTER (Tom)
改进更多平台的 ps 状态显示 (Peter E、Marc)
改进 CREATE FUNCTION 失败消息 (Ross)
JDBC 改进 (Peter、Travis Bauer、Christopher Cain、William Webber、Gunnar)
大一统配置方案/GUC。现在许多选项可以在 data/postgresql.conf、postmaster/postgres 标志或 SET 命令中设置 (Peter E)
改进文件描述符缓存的处理 (Tom)
关于自动创建表别名条目的新警告代码 (Bruce)
彻底改造 initdb 流程 (Tom、Peter E)
继承表的彻底改造；继承表现在默认被访问；新 ONLY 关键字可阻止 (Chris Bitmead、Tom)
ODBC 清理/改进 (Nick Gorham、Stephan Szabo、Zoltan Kovacs、Michael Fork)
允许重命名临时表 (Tom)
彻底改造内存管理器上下文 (Tom)
pg_dumpall 使用 CREATE USER 或 CREATE GROUP 而不是 COPY (Peter E)
彻底改造 pg_dump (Philip Warner)
允许 pg_hba.conf 辅助口令文件只指定用户名 (Peter E)
创建临时表时允许 TEMPORARY 或 TEMP 关键字 (Bruce)
新内存泄漏检查器 (Karel)
新 SET SESSION CHARACTERISTICS (Thomas)
允许嵌套块注释 (Thomas)
添加 WITHOUT TIME ZONE 类型限定符 (Thomas)
新 ALTER TABLE ADD CONSTRAINT (Stephan)
INTEGER 聚合使用 NUMERIC 累加器 (Tom)
彻底改造聚合代码 (Tom)
新 VARIANCE 和 STDDEV() 聚合
改进 pg_dump 的依赖排序 (Philip)
新 pg_restore 命令 (Philip)
新 pg_dump tar 输出选项 (Philip)
新 pg_dump 大对象支持 (Philip)
LIKE 的新 ESCAPE 选项 (Thomas)
新不区分大小写的 LIKE——ILIKE (Thomas)
允许函数索引使用二进制兼容类型 (Tom)
允许 SQL 函数用于更多上下文 (Tom)
新 pg_config 工具 (Peter E)
新 PL/pgSQL EXECUTE 命令，允许动态 SQL 和工具语句 (Jan)
新 PL/pgSQL GET DIAGNOSTICS 语句用于 SPI 值访问 (Jan)
新 quote_identifiers() 和 quote_literal() 函数 (Jan)
新 ALTER TABLE table OWNER TO user 命令 (Mark Hollomon)
允许 FROM 中的子查询，即 FROM (SELECT ...) [AS] alias (Tom)
把 PyGreSQL 更新到 3.1 版 (D'Arcy)
以 OID 命名的文件存储表 (Vadim)
新 SQL 函数 setval(seq,val,bool) 供 pg_dump 使用 (Philip)
要求用 DROP VIEW 删除视图，不再用 DROP TABLE (Mark)
允许 DROP VIEW view1, view2 (Mark)
允许 DROP INDEX、DROP RULE 和 DROP TYPE 中的多个对象 (Tom)
允许与 Unicode 的自动转换 (Tatsuo、Eiji)
新 /contrib/pgcrypto 哈希函数 (Marko Kreen)
新 pg_dumpall --globals-only 选项 (Peter E)
WAL 的新 CHECKPOINT 命令，创建新 WAL 日志文件 (Vadim)

新 AT TIME ZONE 语法 (Thomas)
允许配置 Unix 域套接字的位置 (David J. MacKenzie)
允许 postmaster 监听特定 IP 地址 (David J. MacKenzie)
允许通过前导斜杠在主机名中指定套接字路径名
    (David J. MacKenzie)
允许 CREATE DATABASE 指定模板数据库 (Tom)
新工具把 MySQL 模式转储转换为 SQL92 和 PostgreSQL (Thomas)
新 /contrib/rserv 复制工具包 (Vadim)
COPY BINARY 的新文件格式 (Tom)
新 /contrib/oid2name 把数字文件映射到表名 (B Palmer)
新 "idle in transaction" ps 状态消息 (Marc)
更新到 pgaccess 0.98.7 (Constantin Teodorescu)
pg_ctl 关闭时现在默认 -w (等待), 新 -l (日志) 选项
向 pg_dump 添加初步的依赖检查 (Philip)

类型

修复 INET/CIDR 类型排序并添加新函数 (Tom)
使 OID 表现为无符号类型 (Tom)
允许 BIGINT 作为 INT8 的同义词 (Peter E)
新 int2 和 int8 比较操作符 (Tom)
新 BIT 和 BIT VARYING 类型 (Adriaan Joubert、Tom、Peter E)
因 TOAST, CHAR() 不再比 VARCHAR() 快 (Tom)
新 GIST seg/cube 示例 (Gene Selkov)
改进的 round(numeric) 处理 (Tom)
修复 CIDR 输出格式化 (Tom)
新 CIDR abbrev() 函数 (Tom)

性能

预写日志 (WAL) 以更少性能开销提供崩溃恢复
    (Vadim)
VACUUM 的 ANALYZE 阶段不再独占锁定表 (Bruce)
减少文件寻道 (Denis Perchine)
改进重复键的 BTREE 代码 (Tom)
把所有大对象存储在单个表中 (Denis Perchine、Tom)
改进内存分配性能 (Karel、Tom)

源代码

新函数管理器调用约定 (Tom)
SGI 可移植性修复 (David Kaelbling)
新 configure --enable-syslog 选项 (Peter E)
新 BSDI README (Bruce)
configure 脚本移到顶级而不是 /src (Peter E)
Makefile/配置/编译彻底改造 (Peter E)
新 configure --with-python 选项 (Peter E)
Solaris 清理 (Peter E)
彻底改造 /contrib Makefile (Karel)
新 OpenSSL 配置选项 (Magnus、Peter E)
AIX 修复 (Andreas)
QNX 修复 (Maurizio)
新 heap_open()、heap_openr() API (Tom)
移除冒号和分号操作符 (Thomas)

视图的新 pg_class.relkind 值 (Mark Hollomon)
把 ichar() 改名为 chr() (Karel)
btrim()、ascii()、chr()、repeat() 的新文档 (Karel)
NT/Cygwin 修复 (Pete Forman)
AIX 移植修复 (Andreas)
新 BeOS 移植 (David Reid、Cyril Velter)
把校对者的更改添加到文档 (Addison-Wesley、Bruce)
新 Alpha 自旋锁代码 (Adriaan Joubert、Compaq)
UnixWare 移植彻底改造 (Peter E)
新 Darwin/Mac OS X 移植 (Peter Bierman、Bruce Hartzler)
新 FreeBSD Alpha 移植 (Alfred)
彻底改造共享内存段 (Tom)
添加 IBM S/390 支持 (Neale Ferguson)
把 macmanuf 移到 /contrib (Larry Rosenman)
Syslog 改进 (Larry Rosenman)
不含用户添加内容的新 template0 数据库 (Tom)
新 /contrib/cube 和 /contrib/seg GIST 示例代码 (Gene Selkov)
允许用 NetBSD 的 libedit 代替 readline (Peter)
改进的汇编语言源代码格式 (Bruce)
新 contrib/pg_logger
createdb 的新 --template 选项
新 contrib/pg_control 工具 (Oliver)
新 FreeBSD 工具 ipc_check、start-scripts/freebsd

E.67. 版本 7.0.3

发行日期
2000-11-11

这包含 7.0.2 之后的多种修复。

E.67.1. 迁移到版本 7.0.3

对于运行 7.0.* 的用户，不需要进行转储/恢复。

E.67.2. 变更

Jdbc 修复 (Peter)
大对象修复 (Tom)
修复 COPY WITH OIDS 泄漏的倾向 (Tom)
修复反向索引扫描 (Tom)
修复 SELECT ... FOR UPDATE 使其检查重复键 (Hiroshi)
为 configure 添加 --enable-syslog (Marc)
修复罕见情形下后端退出时中止事务的问题 (Tom)
启用多字节时 psql \l+ 的修复 (Tatsuo)
允许 PL/pgSQL 接受非 ASCII 标识符 (Tatsuo)
使 vacuum 总是刷新缓冲区 (Tom)
修复以允许在等待锁时取消 (Hiroshi)
用户认证代码中内存分配问题的修复 (Tom)

移除对 int4out() 的虚假使用 (Tom)
COALESCE 或 BETWEEN 中多个子查询的修复 (Tom)
某些情形下堆打开时触发器失败的修复 (Jeroen van Vianen)
修复不相等的选择率错误 (Tom)
修复 strcmp() 的错误使用 (Tom)
修复存储管理器访问文件末尾之外条目的缺陷 (Tom)
修复以在所有 smgr elog 消息中包含内核 errno 消息 (Tom)
构建时 '.' 不在 PATH 中的修复 (SL Baur)
文件描述符耗尽错误的修复 (Tom)
修复以使 pg_dump 转储函数的 'iscachable' 标志 (Tom)
Append 节点目标列表中子查询的修复 (Tom)
归并连接计划的修复 (Tom)
修复带索引关系上的 TRUNCATE 失败 (Tom)
避免写错误时数据库范围重启 (Hiroshi)
修复 nodeMaterial 通过重算输出以遵循 chgParam (Tom)
修复当行版本的源和目的地位于同一页时 VACUUM 移动更新行版本链的问题 (Tom)
修复 user.c CommandCounterIncrement (Tom)
to_char() 中 AM/PM 边界问题的修复 (Karel Zak)
修复 TIME 聚合处理 (Tom)
修复 to_char() 以避免对 NULL 输入核心转储 (Tom)
缓冲区修复 (Tom)
修复向 char() 数据类型插入/复制较长多字节字符串的问题 (Tatsuo)
修复中止时后端崩溃的问题 (Tom)

E.68. 版本 7.0.2

发行日期
2000-06-05

这是 7.0.1 的重新打包，附加了文档。

E.68.1. 迁移到版本 7.0.2

对于运行 7.* 的用户，不需要进行转储/恢复。

E.68.2. 变更

已把文档添加到压缩包。

E.69. 版本 7.0.1

发行日期
2000-06-01

这是 7.0 的清理版本。

E.69.1. 迁移到版本 7.0.1

对于运行 7.0 的用户，不需要进行转储/恢复。

E.69.2. 变更

修复许多 CLUSTER 失败 (Tom)
允许 ALTER TABLE RENAME 用于索引 (Tom)
修复 plpgsql 处理 datetime->timestamp 和 timespan->interval (Bruce)
新的 configure --with-setproctitle 开关以使用 setproctitle() (Marc、Bruce)
修复 6.5.3 起 ResultSet 的差一错误等
jdbc ResultSet 修复 (Joseph Shraibman)
优化器调优 (Tom)
为 pgaccess 修复 create user
修复 UNLISTEN 失败
IRIX 修复 (David Kaelbling)
QNX 修复 (Andreas Kardos)
降低 COPY IN 锁级别 (Tom)
把 libpqeasy 改为使用 PQconnectdb() 风格参数 (Bruce)
修复 pg_dump 处理 OID 索引 (Tom)
修复小内存泄漏 (Tom)
Solaris 上 createdb/dropdb 的修复 (Tatsuo)
非阻塞连接修复 (Alfred Perlstein)
修复 RENAME TABLE 失败后的不当恢复 (Tom)
把 pg_ident.conf.sample 复制到安装的 /lib 目录 (Bruce)
新增 SJIS UDC (NEC selection IBM 汉字) 支持 (Eiji Tokuya)
修复过长的 syslog 消息 (Tatsuo)
修复带引号索引名过长的问题 (Tom)
JDBC ResultSet.getTimestamp() 修复 (Gregory Krasnow & Floyd Marinescu)
ecpg 变更 (Michael)

E.70. 版本 7.0

发行日期
2000-05-08

此发行版包含许多领域的改进，展示了 PostgreSQL 的持续成长。7.0 的改进和修复比以往任何版本都多。开发者确信这是迄今最好的版本；我们尽力只发布扎实的版本，这个版本也不例外。

此发行版的主要变更：

Foreign Keys

外键现在已实现，PARTIAL MATCH 外键除外。许多用户一直要求此特性，我们很高兴提供它。

Optimizer Overhaul

延续一年前开始的工作，优化器得到改进，允许更好的查询计划选择，性能更快且内存使用更少。

Updated psql

我们的交互式终端监视器 psql 已更新，新增多种特性。详情见 psql 手册页。

Join Syntax

SQL92 连接语法现在受支持，但此版本只支持 `INNER JOIN`。`JOIN`、`NATURAL JOIN`、`JOIN/USING` 和 `JOIN/ON` 都可用，列相关名也可用。

E.70.1. 迁移到版本 7.0

希望从任何以前的 PostgreSQL 版本迁移数据的用户，需要使用 pg_dump 进行转储/恢复。对于从 6.5.* 升级的用户，可以改用 pg_upgrade 升级到此版本；但完整的转储/重装安装始终是最可靠的升级方法。

新版本需要考虑的接口和兼容性问题包括：

- 日期/时间类型 `datetime` 和 `timespan` 已被 SQL92 定义的 `timestamp` 和 `interval` 取代。虽然我们做了一些努力来简化过渡，允许 PostgreSQL 识别废弃的类型名并转换为新类型名，但这一机制对你的现有应用程序不可能是完全透明的。
- 优化器在查询代价估计方面得到显著改进。在某些情况下，
  由于优化器为首选计划做出更好的选择，查询时间会缩短。
  但在少数情形下（通常涉及病态的数据分布），查询时间可能上升。如果你处理大量数据，
  可能想检查查询以验证性能。
- JDBC 和 ODBC 接口已升级和扩展。
- 字符串函数 `CHAR_LENGTH` 现在是原生函数。以前的版本把它转换为对 `LENGTH` 的调用，这可能与实现 `LENGTH` 的其他类型（如几何类型）产生歧义。

E.70.2. 变更

```
缺陷修复
---------
防止函数调用超过最大参数数量 (Tom)
改进 CASE 构造 (Tom)
修复 SELECT coalesce(f1,0) FROM int4_tbl GROUP BY f1 (Tom)
修复 SELECT sentence.words[0] FROM sentence GROUP BY sentence.words[0]
 (Tom)
修复 GROUP BY 扫描缺陷 (Tom)
SQL 语法处理的改进 (Tom)
修复涉及 INSERT ... SELECT ... 的视图的问题 (Tom)
修复 SELECT a/2, a/2 FROM test_missing_target GROUP BY a/2 (Tom)
修复 INSERT ... SELECT 中的子查询 (Tom)
防止 INSERT ... SELECT ... ORDER BY (Tom)
大于 2GB 关系的修复，包括 vacuum
改进系统表变更向其他后端的传播 (Tom)
改进用户表变更向其他后端的传播 (Tom)
修复复杂情形下临时表的处理 (Bruce、Tom)
允许在表打开时锁定表，改进并发可靠性 (Tom)
在 pg_dump 中正确引用序列名 (Ross J. Reedstrom)
防止他人访问时 DROP DATABASE
若无行被处理则防止 GROUP BY 返回任何行 (Tom)
```

修复无行匹配 WHERE 时 SELECT COUNT(1) FROM table WHERE ... (Tom)
修复 pg_upgrade 使其适用于 MVCC (Tom)
修复 SELECT ... WHERE x IN (SELECT ... HAVING SUM(x) > 1) (Tom)
修复 "f1 datetime DEFAULT 'now'" (Tom)
修复 DEFAULT 中使用 CURRENT_DATE 的问题 (Tom)
允许纯注释行，也允许 ;;; 行。 (Tom)
改进磁盘写入失败、磁盘满后的恢复 (Hiroshi)
修复 FROM 中提到表但未连接的情形 (Tom)
允许无聚合函数的 HAVING 子句 (Tom)
修复 "--" 注释且无尾随换行的问题，如 perl 接口中所见
改进 pg_dump 失败错误报告 (Bruce)
允许排序和哈希超过 2GB 文件大小 (Tom)
修复 pg_dump 转储继承规则的问题 (Tom)
修复 NULL 处理比较的问题 (Tom)
修复失败 CREATE/DROP 命令导致的不一致状态 (Hiroshi)
修复带连字符的数据库名的问题
防止 DROP INDEX 干扰其他后端 (Tom)
修复 verify_password() 中的文件描述符泄漏
修复 "Unable to identify an operator =$" 问题
修复 ODBC 在启用 CommLog 和 Debug 时不再段错误 (Dirk Niggemann)
修复递归 exit 调用 (Massimo)
修复超长时区的问题 (Jeroen van Vianen)
使 pg_dump 保留主键信息 (Peter E)
防止带单引号的数据库名 (Peter E)
防止事务内 DROP DATABASE (Peter E)
ecpg 内存泄漏修复 (Stephen Birch)
修复 SELECT null::text、SELECT int4fac(null) 和 SELECT 2 + (null) (Tom)
Y2K 时间戳修复 (Massimo)
修复 VACUUM 'HEAP_MOVED_IN was not expected' 错误 (Tom)
修复含空格表/列名的视图的问题 (Tom)
防止索引上的权限 (Peter E)
修复生成错误时自旋锁卡住的问题 (Hiroshi)
修复 Linux 上的 ipcclean
修复 NULL 约束条件的处理 (Tom)
修复 odbc 驱动中的内存泄漏 (Nick Gorham)
修复 UNION 表上的权限检查 (Tom)
修复以允许 SELECT 'a' LIKE 'a' (Tom)
修复 SELECT 1 + NULL (Tom)
CHAR 的修复
修复 numeric 类型上的 log() (Tom)
废弃 ':' 和 ';' 操作符
允许清理临时表
不允许与新列同名的继承列
磁盘空间耗尽时恢复或强制失败 (Hiroshi)
修复 AS 列与结果列匹配的 INSERT INTO ... SELECT
修复 INSERT ... SELECT ... GROUP BY 按目标列而非源列分组的问题 (Tom)
修复带 INSERT 的 CREATE TABLE test (a char(5) DEFAULT text '', b int4) (Tom)
修复带 LIMIT 的 UNION
修复 CREATE TABLE x AS SELECT 1 UNION SELECT 2
修复 CREATE TABLE test(col char(2) DEFAULT user)
修复 CREATE TABLE ... DEFAULT 中的类型不匹配
修复 SELECT * FROM pg_class where oid in (0,-1)
修复 SELECT COUNT('asdf') FROM pg_class WHERE oid=12

防止能创建数据库的用户修改 pg_database 表 (Peter E)
修复 btree 在键 > 1/2 (页 - 开销) 时给出有用的 elog (Tom)
修复向 DECIMAL(4,4) 字段插入 0.0 的问题 (Tom)

增强

基于 SQL3/SQL98 的新 CLI 接口头文件 sqlcli.h
移除查询长度的所有限制，行长度限制仍存在 (Tom)
把 jdbc 协议更新到 2.0 (Jens Glaser <jens@jens.de>)
添加 TRUNCATE 命令以快速截断关系 (Mike Mascari)
修复以给超级用户和 createdb 用户适当的更新目录权限 (Peter E)
允许 ecpg bool 变量具有 NULL 值 (Christof)
对无 NULL 指示符变量的 NULL 值发出 ecpg 错误 (Christof)
允许 ^C 取消 COPY 命令 (Massimo)
添加 SET FSYNC 和 SHOW PG_OPTIONS 命令 (Massimo)
动态装载 C 函数的函数名重载 (Frankpitt)
向 libpq++ 添加 CmdTuples() (Vince)
新 CREATE CONSTRAINT TRIGGER 和 SET CONSTRAINTS 命令 (Jan)
允许 CREATE FUNCTION/WITH 子句用于所有语言类型
configure --enable-debug 添加 -g (Peter E)
configure --disable-debug 移除 -g (Peter E)
允许更复杂的默认表达式 (Tom)
第一个真正可用的 FOREIGN KEY 约束触发器功能 (Jan)
添加 FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)
添加 FOREIGN KEY ... MATCH <unspecified> 参照动作 (Don Baccus)
允许对 ctid (物理堆位置) 的 WHERE 限制 (Hiroshi)
把 pginterface 从 contrib 移到 interface 目录并改名为 pgeasy (Bruce)
更改 pgeasy connectdb() 参数顺序 (Bruce)
要求 SELECT DISTINCT 目标列表包含所有 ORDER BY 列 (Tom)
添加 Oracle 的 COMMENT ON 命令 (Mike Mascari <mascarim@yahoo.com>)
libpq 的 PQsetNoticeProcessor 函数现在返回先前的钩子 (Peter E)
防止 PQsetNoticeProcessor 被设为 NULL (Peter E)
使 COPY 中的 USING 可选 (Bruce)
允许目标列表中的子查询 (Tom)
允许比较操作符左侧的子查询 (Tom)
新并行回归测试 (Jan)
更改后端侧 COPY 使写文件的权限为 644 而不是 666 (Tom)
强制 PGDATA 目录权限安全，即使它已存在 (Tom)
添加 psql LASTOID 变量以返回最后插入的 oid (Peter E)
允许并发 vacuum 并移除 pg_vlock vacuum 锁文件 (Tom)
为 vacuum 添加权限检查 (Peter E)
新的允许异步连接的 libpq 函数：PQconnectStart()、
  PQconnectPoll()、PQresetStart()、PQresetPoll()、PQsetenvStart()、
  PQsetenvPoll()、PQsetenvAbort (Ewan Mellor)
新 libpq PQsetenv() 函数 (Ewan Mellor)
create/alter user 扩展 (Peter E)
$PGDATA 下新的 postmaster.pid 和 postmaster.opts (Tatsuo)
新 create/drop user/db 脚本 (Peter E)
psql 重大改造 (Peter E)
向 libpq 接口添加 const (Peter E)
新 libpq 函数 PQoidValue (Peter E)
显示导致 GROUP BY 问题的特定非聚合 (Tom)
使 pg_shadow 的更改重建 pg_pwd 文件 (Peter E)
添加 aggregate(DISTINCT ...) (Tom)

允许用标志控制 NULL 的 COPY 输入/输出 (Peter E)
使 postgres 用户默认有口令 (Peter E)
添加 CREATE/ALTER/DROP GROUP (Peter E)
所有管理脚本现在支持 --long 选项 (Peter E、Karel)
Vacuumdb 脚本现在支持 --all 选项 (Peter E)
ecpg 新的可移植 FETCH 语法
添加 ecpg 的 EXEC SQL IFDEF、EXEC SQL IFNDEF、EXEC SQL ELSE、EXEC SQL ELIF
和 EXEC SQL ENDIF 指令
添加 pg_ctl 脚本以控制后端启动 (Tatsuo)
添加 postmaster.opts.default 文件以存储启动标志 (Tatsuo)
允许 --with-mb=SQL_ASCII
把索引键的最大数量增加到 16 (Bruce)
把函数参数的最大数量增加到 16 (Bruce)
允许配置索引键和参数的最大数量 (Bruce)
允许非特权用户更改自己的口令 (Peter E)
启用口令认证；新用户必须使用 (Peter E)
不允许删除拥有数据库的用户 (Peter E)
把 initdb 选项 --with-mb 改为 --enable-multibyte
添加 initdb 提示超级用户口令的选项 (Peter E)
允许复杂类型转换如 col::numeric(9,2) 和 col::int2::float8 (Tom)
更新 initdb、initlocation、pg_dump、ipcclean 的用户界面 (Peter E)
新 pg_char_to_encoding() 和 pg_encoding_to_char() 函数 (Tatsuo)
libpq 非阻塞模式 (Alfred Perlstein)
改进未指定长度转换中类型的转换
新 plperl 内部编程语言 (Mark Hollomon)
允许 COPY IN 读取不以换行结尾的文件 (Tom)
指示长标识符何时被截断 (Tom)
允许聚合使用类型等价 (Peter E)
添加 Oracle 的 to_char()、to_date()、to_datetime()、to_timestamp()、to_number()
转换函数 (Karel Zak <zakkr@zf.jcu.cz>)
添加 SELECT DISTINCT ON (expr [, expr ...]) targetlist ... (Tom)
检查以确保 ORDER BY 与 DISTINCT 操作兼容 (Tom)
向 ODBC 添加 NUMERIC 和 int8 类型
改进 Append、Group、Agg、Unique 的 EXPLAIN 结果 (Tom)
添加 ALTER TABLE ... ADD FOREIGN KEY (Stephan Szabo)
允许 PL/pgSQL 中的 SELECT .. FOR UPDATE (Hiroshi)
即使到达 EOF 也启用反向顺序扫描 (Hiroshi)
添加布尔值及 >= 和 <= 的 btree 索引 (Don Baccus)
COPY FROM 失败时打印当前行号 (Massimo)
识别 POSIX 时区，如 "PST+8" 和 "GMT-8" (Thomas)
添加 DEC 作为 DECIMAL 的同义词 (Thomas)
添加 SESSION_USER 作为 SQL92 关键字，同 CURRENT_USER (Thomas)
实现 SQL92 列别名（也称相关名） (Thomas)
实现 SQL92 连接语法 (Thomas)
使 INTERVAL 保留字可用作列标识符 (Thomas)
实现 REINDEX 命令 (Hiroshi)
接受聚合函数 SUM(ALL col) 中的 ALL (Tom)
防止 GROUP BY 使用列别名 (Tom)
新 psql \encoding 选项 (Tatsuo)
允许 PQrequestCancel() 在等待锁状态下终止 (Hiroshi)
允许在所有情形下对负数取负
添加 ecpg 描述符 (Christof、Michael)

允许 CREATE VIEW v AS SELECT f1::char(8) FROM tbl
允许带长度的转换，如 foo::char(8)
新 libpq 函数 PQsetClientEncoding()、PQclientEncoding() (Tatsuo)
添加对 SJIS 用户定义字符的支持 (Tatsuo)
支持更大的视图/规则
使 libpq 的 PQconndefaults() 线程安全 (Tom)
禁用 // 作为注释以符合 ANSI，应使用 -- (Tom)
允许视图上的列别名 CREATE VIEW name (collist)
修复带子查询的视图的问题 (Tom)
允许 UPDATE table SET fld = (SELECT ...) (Tom)
SET 命令选项不再需要引号
把 pgaccess 更新到 0.98.6
新 SET SEED 命令
新 pg_options.sample 文件
新 SET FSYNC 命令 (Massimo)
允许创建表时 pg_descriptions
允许创建类型、列和函数时 pg_descriptions
允许 psql \copy 使用分隔符 (Peter E)
允许 psql 把 null 打印为与 "" 不同的 [null] (Peter E)

类型

大量数组修复 (Tom)
允许裸列名作为数组下标 (Tom)
改进 int 和 float 常量的类型转换 (Tom)
清理 int8 输入、范围检查和类型转换 (Tom)
修复 SELECT timespan('21:11:26'::time) (Tom)
netmask('x.x.x.x/0') 为 255.255.255.255 而不是 0.0.0.0 (Oleg Sharoiko)
在 NUMERIC 上添加 btree 索引 (Jan)
含 NUL 字符大对象的 Perl 修复 (Douglas Thomson)
大对象的 ODBC 修复 (free)
修复 cidr 数据类型的索引
修复以太网 MAC 地址 (macaddr 类型) 比较的问题
修复计算中发生溢出时日期/时间类型的问题 (Tom)
允许 int8 上的数组 (Peter E)
修复 NUMERIC 类型 (如 NUMERIC(4,4)) 的舍入/溢出问题 (Tom)
允许 NUMERIC 数组
修复 NUMERIC ceil() 和 floor() 函数中的缺陷 (Tom)
使 char_length()/octet_length 包含尾随空白 (Tom)
使 abstime/reltime 使用 int4 而不是 time_t (Peter E)
用于压缩文本字段的新 lztext 数据类型
修改处理 int 和 float 常量强制的代码 (Tom)
开始编写实现 BIT 和 BIT VARYING 类型的新代码 (Adriaan Joubert)
NUMERIC 现在接受科学记数法 (Tom)
NUMERIC 到 int4 舍入 (Tom)
正确地把 float4/8 转换为 NUMERIC (Tom)
允许与 NUMERIC 的类型转换 (Thomas)
使 ISO 日期风格 (2000-02-16 09:33) 成为默认 (Thomas)
添加 NATIONAL CHAR [VARYING] (Thomas)
允许 NUMERIC round 和 trunc 接受负标度 (Tom)
新 TIME WITH TIME ZONE 类型 (Thomas)
在 time 类型上添加 MAX()/MIN() (Thomas)
为 int8 添加 abs()、mod()、fac() (Thomas)
把函数改名为 float8 的 round()、sqrt()、cbrt()、pow() (Thomas)

为 float8 添加超越数学函数（如 sin()、acos()）（Thomas）
为 NUMERIC 类型添加 exp() 和 ln()
把 NUMERIC power() 改名为 pow()（Thomas）
改进的 TRANSLATE() 函数（Edwin Ramirez、Tom）
允许 X=-Y 操作符（Tom）
允许 SELECT float8(COUNT(*))/(SELECT COUNT(*) FROM t) FROM t GROUP BY f1;（Tom）
允许 LOCALE 在正则表达式搜索中使用索引（Tom）
允许创建函数索引时使用默认类型

性能

防止大量 AND 和 OR 的指数级空间消耗（Tom）
收集系统列的属性选择率值（Tom）
降低聚合的内存使用（Tom）
修复 LIKE 优化以在多字节编码下使用索引（Tom）
修复 r-tree 索引优化器选择率（Thomas）
改进优化器选择率计算和函数（Tom）
优化存在许多相等键情形的 btree 搜索（Tom）
只在有索引时启用快速 LIKE 索引处理（Tom）
重用带重复项索引页上的空闲空间（Tom）
改进哈希连接处理（Tom）
结果已排序时防止降序排序（Hiroshi）
允许索引扫描查询限定的交换（Tom）
在需要 ORDER BY/GROUP BY 的情形优先索引扫描（Tom）
为性能把大内存请求按固定大小块分配（Tom）
通过减少内存分配请求修复 vacuum 的性能（Tom）
实现常量表达式化简（Bernard Frankpitt、Tom）
用辅助列确定索引扫描的起点（Hiroshi）
防止内部排序时四倍的磁盘空间使用（Tom）
通过调用更少函数加快排序（Tom）
创建与所有系统缓存匹配的系统索引（Bruce、Hiroshi）
使系统缓存使用系统索引（Bruce）
使所有系统索引唯一（Bruce）
改进 pg_statistics 管理以提升 VACUUM 速度（Tom）
降低后端缓存刷新频率（Tom、Hiroshi）
COPY 现在重用先前的内存分配，改进性能（Tom）
改进优化代价估计（Tom）
改进范围查询 x > lowbound AND x < highbound 的优化器估计（Tom）
在适当处使用 DNF 而不是 CNF（Tom、Taral）
进一步清理 OR-of-AND WHERE 子句（Tom）
使用 OR 子句中的索引 (x = 1 AND y = 2) OR (x = 2 AND y = 4)（Tom）
随机索引页访问的更智能优化器计算（Tom）
控制优化器代价的新 SET 变量（Tom）
基于 LIMIT、OFFSET 和 EXISTS 限定优化查询（Tom）
减少优化器连接路径的内部管理以提速（Tom）
子查询大幅提速（Tom）
未禁用 fsync 时减少 fsync 写（Tom）
改进的 LIKE 优化器估计（Tom）
防止 SELECT-only 查询中的 fsync（Vadim）
使索引创建使用 psort 代码，因为它现在更快（Tom）
允许创建 > 1GB 的排序临时表

源代码树变更

```
-------------------
修复 linux PPC 编译
新通用表达式树遍历子例程 (Tom)
把 form() 改为 varargform() 以防止可移植性问题
改进 Alpha 上大整数的范围检查
清理 /include 目录中的 #include (Bruce)
添加检查 include 的脚本 (Bruce)
从 *.c 文件移除不需要的 #include (Bruce)
更改 #include 以酌情使用 <> 和 "" (Bruce)
启用 libpq 的 Windows 编译
来自 Uncle George <gatgul@voicenet.com> 的 Alpha 自旋锁修复
优化器数据结构的彻底改造 (Tom)
修复 cygipc 库 (Yutaka Tanida)
允许 pgsql 在更新的 Cygwin 快照上工作 (Dan)
新目录版本号 (Tom)
添加 Linux ARM
把 heap_replace 改名为 heap_update
为 QNX 更新 (Dr. Andreas Kardos)
新平台特定的回归处理 (Tom)
把 oid8 改名为 oidvector、int28 改名为 int2vector (Bruce)
把所有 yacc 和 lex 文件包含进发行版 (Peter E.)
移除 lextest, 不再需要 (Peter E)
修复 Windows 上的 libpq 和 psql (Magnus)
内部把 datetime 和 timespan 改为 timestamp 和 interval (Thomas)
修复 BSD/OS 上的 plpgsql
向回归测试添加 SQL_ASCII 测试用例 (Tatsuo)
configure --with-mb 现已废弃 (Tatsuo)
NT 修复
NetBSD 修复 (Johnny C. Lam <lamj@stat.cmu.edu>)
Alpha 编译修复
新多字节编码
```

E.71. 版本 6.5.3

发行日期
1999-10-13

这基本上是 6.5.2 的清理版本。我们添加了 6.5.2 中缺失的新 PgAccess，并安装了 NT 特定的修复。

E.71.1. 迁移到版本 6.5.3

对于运行 6.5.* 的用户，不需要进行转储/恢复。

E.71.2. 变更

```
更新了 pgaccess 0.98 版本
NT 特定补丁
修复继承表上规则的转储
```

E.72. 版本 6.5.2

发行日期
1999-09-15

这基本上是 6.5.1 的清理版本。我们修复了 6.5.1 用户报告的多种问题。

E.72.1. 迁移到版本 6.5.2

对于运行 6.5.* 的用户，不需要进行转储/恢复。

E.72.2. 变更

子查询+CASE 修复 (Tom)
为 solaris_i386 和 solaris_sparc 移植添加 SHLIB_LINK 设置 (Daren Sefcik)
WHERE 连接子句中 CASE 的修复 (Tom)
修复 BTScan 中止 (Tom)
修复对冗余 UNIQUE 和 PRIMARY KEY 索引的检查 (Thomas)
改进为可检查多列约束 (Thomas)
修复启用 MB 时 Windows 的制作问题 (Hiroki Kataoka)
允许 BSD yacc 和 bison 编译 pl 代码 (Bruce)
修复 SET NAMES 的功能
int8 修复 (Thomas)
修复 vacuum 的内存消耗 (Hiroshi、Tatsuo)
降低 vacuum 的总内存消耗 (Tom)
修复 timestamp(datetime)
规则反编译缺陷修复 (Tom)
修复 mkMakefile.tcldefs.sh.in 和 mkMakefile.tkdefs.sh.in 中的引号问题 (Tom)
这是为重用 vacuum 释放的索引页空间 (Vadim)
为 pg_dump 记录 -x (Bruce)
修复规则反编译器中的一元操作符 (Tom)
在 mdtruncate() 期间注释掉多余段的 FileUnlink (Tom)
来自 Yu Cao >yucao@falcon.kla-tencor.com< 的 IRIX 链接修复
修复 LIKE 中的逻辑错误：到达模式末尾而文本未结束时不应返回 LIKE_ABORT (Tom)
修复事务中止期间堆内存分配的不当清理 (Tom)
更新 pgaccess 0.98 版本

E.73. 版本 6.5.1

发行日期
1999-07-15

这基本上是 6.5 的清理版本。我们修复了 6.5 用户报告的多种问题。

E.73.1. 迁移到版本 6.5.1

对于运行 6.5 的用户，不需要进行转储/恢复。

E.73.2. 变更

添加 NT README 文件
linux_ppc、IRIX、linux_alpha、OpenBSD、alpha 的可移植性修复
移除 QUERY_LIMIT，使用 SELECT...LIMIT
修复继承上的 EXPLAIN (Tom)
补丁允许对多段表进行清理 (Hiroshi)
R-Tree 优化器选择率修复 (Tom)
ACL 文件描述符泄漏修复 (Atsushi Ogawa)
新表达式子树代码 (Tom)
只读事务避免磁盘写 (Vadim)
修复最后事务中止时临时表的移除 (Bruce)
防止创建过大的行 (Bruce)
plpgsql 修复
允许 32k - 64k 的端口号 (Bruce)
添加 ^ 优先级 (Bruce)
把名为 pg_temp 的排序文件改名为 pg_sorttemp (Bruce)
修复时间值中的微秒 (Tom)
教程源码清理
新 linux_m68k 移植
修复某些情形下 NULL 的排序 (Tom)
修复共享库依赖 (Tom)
修复影响子查询中 GROUP BY 的故障 (Tom)
修复一些编译器警告 (Tomoaki Nishiyama)
新增 Win1250 (捷克语) 支持 (Pavel Behal)

E.74. 版本 6.5

发行日期
1999-06-09

此版本标志着开发团队对从 Berkeley 继承的源代码的掌握迈出了一大步。你会看到我们现在能轻松添加重大特性，这要归功于我们全球开发团队不断扩大的规模和日益丰富的经验。

以下是较显著变更的简要总结：

多版本并发控制（MVCC）

这移除了我们旧的表级锁定，代之以优于大多数商业数据库系统的锁定系统。
在传统系统中，每个被修改的行都被锁定直到提交，阻止其他用户读取。MVCC 利用 PostgreSQL 天生的多版本特性，允许读者在写入者活动期间继续读取一致的数据。写入者继续使用紧凑的 pg_log 事务系统。这一切都不需要像传统数据库系统那样为每行分配锁。因此，基本上我们不再受简单表级锁定的限制；我们有了比行级锁定更好的东西。

来自 pg_dump 的热备份

pg_dump 利用新的 MVCC 特性，在数据库保持在线并可用于查询的同时给出一致的数据库转储/备份。

Numeric data type

我们现在有了真正的数字数据类型，精度由用户指定。

Temporary tables

临时表保证在数据库会话内名称唯一，并在会话退出时销毁。

新 SQL 特性

我们现在支持 CASE、INTERSECT 和 EXCEPT 语句。新增了 LIMIT/OFFSET、SET TRANSACTION ISOLATION LEVEL、SELECT ... FOR UPDATE 以及改进的 LOCK TABLE 命令。

Speedups

得益于团队内的多种才能，我们继续加快 PostgreSQL 的速度。我们加速了内存分配、优化、表连接和行传输例程。

Ports

我们继续扩大移植列表，这次包括 Windows NT/ix86 和 NetBSD/arm32。

Interfaces

大多数接口有了新版本，现有功能得到改进。

Documentation

文档中到处都有新的和更新的材料。SGI 和 AIX 平台有了新的 FAQ。教程有 Stefan Simkovics 编写的 SQL 入门信息。用户指南有涵盖 postmaster 和更多工具程序的参考页，新附录包含日期/时间行为的细节。管理员指南有 Tom Lane 编写的故障排除新章节。程序员指南有 Stefan 编写的查询处理描述，以及通过匿名 CVS 和 CVSup 获取 PostgreSQL 源代码树的细节。

E.74.1. 迁移到版本 6.5

希望从任何以前的 PostgreSQL 版本迁移数据的用户，需要使用 pg_dump 进行转储/恢复。pg_upgrade 不能用于升级到此版本，因为表的磁盘结构相对于以前的版本已更改。

新的多版本并发控制（MVCC）特性在多用户环境中可能给出略为不同的行为。
请阅读并理解以下章节，以确保你的现有应用程序能给出你需要的行为。

E.74.1.1. 多版本并发控制

由于 6.5 中的读者不锁定数据（无论事务隔离级别如何），一个事务读取的数据可能被另一个事务覆写。换句话说，如果 `SELECT` 返回了一行，并不意味着该行在被返回时（即语句或事务开始之后的某个时刻）确实存在，也不意味着该行在当前事务提交或回滚之前受到保护，
不被并发事务删除或更新。

为确保某行确实存在并保护它不被并发更新，必须使用 `SELECT FOR UPDATE` 或适当的 `LOCK TABLE` 语句。从以前的 PostgreSQL 版本和其他环境移植应用程序时应考虑到这一点。

如果你在使用 `contrib/refint.*` 触发器做参照完整性，请记住上述内容。现在需要额外的技术。一种方法是：如果事务要更新/删除主键，使用 `LOCK parent_table IN SHARE ROW EXCLUSIVE MODE` 命令；如果事务要更新/插入外键，使用 `LOCK parent_table IN SHARE MODE` 命令。

> **注意**
>
> 注意，如果你以 SERIALIZABLE 模式运行事务，则必须在执行事务中的任何 DML 语句（SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO）之前执行上述 LOCK 命令。

当实现读取脏（未提交）数据的能力（无论隔离级别）和真正的参照完整性后，
这些不便将消失。

E.74.2. 变更

```
缺陷修复
---------
修复 text<->float8 和 text<->float4 转换函数 (Thomas)
修复创建带大小写混合约束表的问题 (Billy)
更改 exp()/pow() 行为以在下溢/上溢时报错 (Jan)
修复 pg_dump -z 中的缺陷
内存越界清理 (Tatsuo)
修复 lo_import 崩溃 (Tatsuo)
调整数据类型名的处理以抑制双引号 (Thomas)
用类型强制匹配列和 DEFAULT (Thomas)
修复死锁，使其在一秒睡眠后只检查一次 (Bruce)
聚合和 PL/pgsql 的修复 (Hiroshi)
修复子查询崩溃 (Vadim)
修复 libpq 函数 PQfnumber 和大小写不敏感名称的问题 (Bahman Rafatjoo)
修复大对象中间写入、无额外块和内存消耗的问题 (Tatsuo)
修复 pg_dump -d 或 -D 以及 INSERT 中特殊字符的引号问题
修复 dynahash 的严重问题 (Tom)
修复 INET/CIDR 可移植性问题
修复 ALTER TABLE ADD COLUMN 中选择率错误的问题 (Bruce)
修复执行器使不同列类型的归并连接可用 (Tom)
修复 Alpha OR 选择率缺陷
修复 OR 索引选择率问题 (Bruce)
修复 \d 为 char()/varchar() 显示正确长度的问题 (Ryan)
修复教程代码 (Clark)
改进 destroyuser 检查 (Oliver)
修复 Kerberos 的问题 (Rodney McDuff)
修复存在脏缓冲区时删除数据库的问题 (Bruce)
修复使 sequence nextval() 可区分大小写的问题 (Bruce)
修复 !!= 操作符
销毁数据库文件前丢弃缓冲区 (Bruce)
修复执行器两次求值函数的情形 (Tatsuo)
允许 sequence nextval 动作区分大小写 (Bruce)
修复优化器对负数索引不工作的问题 (Bruce)
修复执行器中 fjIsNull 的内存泄漏
修复聚合内存泄漏 (Erik Riedel)
允许含连字符的用户名授予权限
清理 inet 类型中的 NULL
清理系统表缺陷 (Tom)
修复 PAGER 和 \? 命令的问题 (Masaaki Sakaida)
把默认多段文件大小限制降为 1GB (Peter)
修复 CREATE OPERATOR 的转储 (Tom)
修复游标反向扫描 (Hiroshi Inoue)
```

修复使用 \i 时的 COPY FROM STDIN (Tom)
修复子查询在表达式内比较的问题 (Jan)
修复返回行时的错误报告处理 (Tom)
修复数组类型引用的问题 (Tom、Jan)
防止 UPDATE SET oid (Jan)
修复 pg_dump 使 -t 选项可处理大小写敏感的表名
特殊情形 GROUP BY 的修复 (Tom、Jan)
修复失败查询的内存泄漏 (Tom)
DEFAULT 现在支持大小写混合的标识符 (Tom)
修复表和索引 DROP/RENAME 的多段使用 (Ole Gjerde)
禁用 pg_dump 同时使用 -o 和 -d 选项 (Bruce)
允许 pg_dump 正确转储组权限 (Bruce)
修复 INSERT INTO table SELECT * FROM table2 中的 GROUP BY (Jan)
修复视图中的计算 (Jan)
修复数组索引上的聚合 (Tom)
修复 DEFAULT 处理值中需要过多引号的单引号的问题
修复非超级用户导入/导出大对象的安全问题 (Tom)
创建表的事务回滚现在正确清理 (Tom)
修复以允许长表名和列名生成正确的序列名 (Tom)

增强

添加 "vacuumdb" 工具
通过更好的内存分配加速 libpq (Tom)
EXPLAIN 显示所有使用的索引 (Tom)
实现 CASE、COALESCE、NULLIF 表达式 (Thomas)
新 pg_dump 表输出格式 (Constantin)
添加字符串 min()/max() 函数 (Thomas)
把新类型强制技术扩展到聚合 (Thomas)
新 moddatetime contrib (Terry)
更新到 pgaccess 0.96 (Constantin)
为单字节 "char" 类型添加例程 (Thomas)
改进的 substr() 函数 (Thomas)
改进的多字节处理 (Tatsuo)
多版本并发控制/MVCC (Vadim)
新 Serialized 模式 (Vadim)
修复超过 2GB 表的问题 (Peter)
新 SET TRANSACTION ISOLATION LEVEL (Vadim)
新 LOCK TABLE IN ... MODE (Vadim)
更新 ODBC 驱动 (Byron)
新 NUMERIC 数据类型 (Jan)
新 SELECT FOR UPDATE (Vadim)
处理输入值的 "NaN" 和 "Infinity" (Jan)
改进的日期/年份处理 (Thomas)
改进的后端连接处理 (Magnus)
日志文件的新选项 ELOG_TIMESTAMPS 和 USE_SYSLOG (Massimo)
新 TCL_ARRAYS 选项 (Massimo)
新 INTERSECT 和 EXCEPT (Stefan)
用于主键跟踪的新 pg_index.indisprimary (D'Arcy)
允许在创建前删除表的新 pg_dump 选项 (Brook)
行输出例程加速 (Tom)
新 READ COMMITTED 隔离级别 (Vadim)
新 TEMP 表/索引 (Bruce)
结果已排序时防止排序 (Jan)

新内存分配优化 (Jan)
允许 psql 执行 \p\g (Bruce)
允许多个规则动作 (Jan)
添加 LIMIT/OFFSET 功能 (Jan)
改进连接大量表时的优化器 (Bruce)
来自 S. Simkovics 硕士论文的新 SQL 介绍 (Stefan、Thomas)
来自 S. Simkovics 硕士论文的后端处理新介绍 (Stefan)
改进的 int8 支持 (Ryan Bradetich、Thomas、Tom)
int8 与 text/varchar 类型之间转换的新例程 (Thomas)
新浓密计划，其中连接元表 (Bruce)
默认启用右手查询 (Bruce)
允许在 configure 时设置可靠的最大后端数
(--with-maxbackends 和 postmaster 开关 (-N backends)) (Tom)
因优化器提速，GEQO 默认现为 10 表 (Tom)
为 MS-SQL 可移植性允许 NULL=Var (Michael、Bruce)
修改 contrib check_primary_key() 使其可为 "automatic" 或
"dependent" (Anand)
允许 psql 对视图的 \d 显示查询 (Ryan)
LIKE 提速 (Bruce)
Ecpg 修复/特性，参见 src/interfaces/ecpg/ChangeLog 文件 (Michael)
JDBC 修复/特性，参见 src/interfaces/jdbc/CHANGELOG (Peter)
使 % 操作符具有与 / 相同的优先级 (Bruce)
添加新 postgres -O 选项以允许系统表结构更改 (Bruce)
更新 contrib/pginterface/findoidjoins 脚本 (Tom)
带索引已删行 vacuum 的大幅提速 (Vadim)
允许非 SQL 函数根据参数运行不同版本 (Tom)
添加 -E 选项显示 \dt 等发送的实际查询 (Masaaki Sakaida)
在 psql 启动横幅中添加版本号 (Masaaki Sakaida)
新 contrib/vacuumlo 移除未被引用的大对象 (Peter)
新表大小初始化使未清理的表性能更好 (Tom)
改进连接被拒绝时的错误消息 (Tom)
支持 char() 和 varchar() 字段数组 (Massimo)
哈希代码的彻底改造以提高可靠性和性能 (Tom)
更新到 PyGreSQL 2.4 (D'Arcy)
更改调试选项使 -d4 和 -d5 产生不同的节点显示 (Jan)
新 pg_options: pretty_plan、pretty_parse、pretty_rewritten (Jan)
更好的系统表访问优化统计 (Tom)
更好地处理非默认块大小 (Massimo)
改进 GEQO 优化器内存消耗 (Tom)
UNION 现在支持对不在目标列表中的列 ORDER BY (Jan)
libpq++ 的重大改进 (Vince Vielhaber)
pg_dump 现在默认使用 -z (ACL) (Bruce)
后端缓存和内存提速 (Tom)
让 pg_dump 在一个快照事务中完成所有工作 (Vadim)
修复大对象内存泄漏和 pg_dump 问题 (Tom)
INET 类型现在比较时遵循网络掩码
使 VACUUM ANALYZE 只使用读锁 (Vadim)
允许 UNION 上的视图 (Jan)
pg_dump 现在能在活动数据库上生成一致快照 (Vadim)

源代码树变更

改进移植匹配 (Tom)
SunOS 可移植性修复

添加 Windows NT 后端移植并启用动态装载 (Magnus 和 Daniel Horak)
新移植到运行 Linux 的 Cobalt Qube (Mips) (Tatsuo)
移植到 NetBSD/m68k (Mr. Mutsuki Nakajima)
移植到 NetBSD/sun3 (Mr. Mutsuki Nakajima)
移植到 NetBSD/macppc (Toshimi Aoki)
修复 tcl/tk 配置 (Vince)
移除规则查询的 CURRENT 关键字 (Jan)
NT 动态装载现在可用 (Daniel Horak)
添加 ARM32 支持 (Andrew McMurry)
更好地支持 HP-UX 11 和 UnixWare
改进文件处理使其更统一，防止文件描述符泄漏 (Tom)
plpgsql 的新安装命令 (Jan)

E.75. 版本 6.4.2

发行日期
1998-12-20

6.4.1 版本打包不当。此版本还包含一个额外的缺陷修复。

E.75.1. 迁移到版本 6.4.2

对于运行 6.4.* 的用户，不需要进行转储/恢复。

E.75.2. 变更

修复某些平台上日期时间常量的问题 (Thomas)

E.76. 版本 6.4.1

发行日期
1998-12-18

这基本上是 6.4 的清理版本。我们修复了 6.4 用户报告的多种问题。

E.76.1. 迁移到版本 6.4.1

对于运行 6.4 的用户，不需要进行转储/恢复。

E.76.2. 变更

新增 pg_dump -N 标志以强制标识符周围加双引号。这是默认 (Thomas)
修复 where 子句中 NOT 导致崩溃的问题 (Bruce)
EXPLAIN VERBOSE 核心转储修复 (Vadim)
修复 Linux 上的共享库问题
修复表存在性测试以允许表名中的大小写混合和空白 (Thomas)

修复几个 pg_dump 缺陷
Configure 更好地匹配 template/.similar 条目 (Tom)
把内建函数名从 SPI_* 改为 spi_*
OR WHERE 子句修复 (Vadim)
大小写混合表名的修复 (Billy)
contrib/linux/postgres.init.csh/sh 修复 (Thomas)
libpq 内存越界修复
SunOS 修复 (Tom)
更改 exp() 行为在下溢时报错 (Thomas)
pg_dump 对内存泄漏、继承约束、布局变更的修复
把 pgaccess 更新到 0.93
64 位平台的原型修复
多字节修复 (Tatsuo)
新 ecpg 手册页
内存越界修复 (Tatsuo)
lo_import() 崩溃修复 (Bruce)
更好地搜索安装程序 (Tom)
时区修复 (Tom)
HP-UX 修复 (Tom)
用隐式类型强制匹配 DEFAULT 值 (Thomas)
添加帮助处理单字节（内部）字符类型的例程 (Thomas)
Windows 下 libpq 编译修复 (Magnus)
升级到 PyGreSQL 2.2 (D'Arcy)

E.77. 版本 6.4

发行日期

1998-10-30

此版本有许多新特性和改进。感谢我们的开发者和维护者，
自上一个版本以来系统的几乎每个方面都得到了关注。以下是简要而不完整的总结：

- 得益于 Jan Wieck 在重写规则系统中的大量新代码，视图和规则现在可以工作了。他还为程序员指南写了相关章节。
- Jan 还贡献了第二个过程语言 PL/pgSQL，与他在上个版本贡献的最初的 PL/pgTCL 过程语言相伴。
- 我们有 Tatsuo Ishii 提供的可选多字节字符集支持，以补充现有的区域设置支持。
- 客户端/服务器通信得到了清理，感谢 Tom Lane，对异步消息和中断的支持更好了。
- 解析器现在执行自动类型强制，以把参数匹配到可用的操作符和函数，
  并把列和表达式与目标列匹配。这使用了支持 PostgreSQL 类型可扩展特性的通用机制。用户指南有涵盖此主题的新章节。
- 新增了三种数据类型。其中两种，inet 和 cidr，支持各种形式的 IP 网络、子网和机器寻址。某些平台上现在有 8 字节整数类型可用。详情见用户指南中的数据类型章节。第四种类型 serial 现在被解析器支持，作为 int4 类型、序列和唯一索引的混合体。
- 新增了更多 SQL92 兼容的语法特性，包括 INSERT DEFAULT VALUES
- 自动配置和安装系统得到了一些关注，应比以往在更多平台上更健壮。

E.77.1. 迁移到版本 6.4

希望从任何以前的 PostgreSQL 版本迁移数据的用户，需要使用 pg_dump 或 pg_dumpall 进行转储/恢复。

E.77.2. 变更

```
缺陷修复
---------
修复 PQsetdb/PQfinish 中的小内存泄漏 (Bryan)
移除 char2-16 数据类型，使用 char/varchar (Darren)
Pqfn 现在处理 NOTICE 消息 (Anders)
降低多后端时自旋锁的忙等开销 (dg)
卡死自旋锁检测 (dg)
修复 "ISO 风格" timespan 的解码和编码 (Thomas)
修复事务回滚后删除表的问题 (Vadim)
更改错误消息并移除无效的更新消息 (Vadim)
修复 COPY 数组检查的问题
修复 SELECT 1 UNION SELECT NULL
修复大对象调用中的缓冲区泄漏 (Pascal)
把 owner 从 oid 类型改为 int4 (Bruce)
修复 Oracle 兼容函数 btrim()、ltrim() 和 rtrim() 中的缺陷
修复共享失效缓存溢出 (Massimo)
防止失败的 COPY 中的文件描述符泄漏 (Bruce)
修复 libpgtcl 的 pg_select 中的内存泄漏 (Constantin)
修复超过 8 字符的用户名/口令问题 (Tom)
修复后端处理异步 NOTIFY 的问题 (Tom)
修复许多错误的系统表条目 (Tom)

增强
------------
升级 ecpg 和 ecpglib，参见 src/interfaces/ecpc/ChangeLog (Michael)
在 EXPLAIN 中显示使用的索引 (Zeugswetter)
EXPLAIN 调用规则系统并显示被重写查询的计划 (Jan)
通过 configure 使许多数据类型和函数感知多字节 (Tatsuo)
新 configure --with-mb 选项 (Tatsuo)
新 initdb --pgencoding 选项 (Tatsuo)
新 createdb -E 多字节选项 (Tatsuo)
Select version(); 现在返回 PostgreSQL 版本 (Jeroen)
libpq 现在允许异步客户端 (Tom)
允许从客户端取消后端查询 (Tom)
psql 现在可用 Control-C 取消查询 (Tom)
libpq 用户不必发送哑查询即可获取 NOTIFY 消息 (Tom)
NOTIFY 现在发送发送者的 PID，可以判断是否是自己的 (Tom)
PGresult 结构现在包含相关的错误消息（如果有） (Tom)
定义 date_part() 的 "tz_hour" 和 "tz_minute" 参数 (Thomas)
添加 varchar 和 bpchar 之间转换的例程 (Thomas)
添加允许把 varchar 和 bpchar 调整到目标列大小的例程 (Thomas)
添加位标志以支持数据检索中的时区小时和分钟 (Thomas)
允许有效浮点数的更多变体（如 ".1"、"1e6"） (Thomas)
修复带前导空格的一元负号解析 (Thomas)
按 SQL92 规范实现 TIMEZONE_HOUR、TIMEZONE_MINUTE (Thomas)
检查并正确忽略 FOREIGN KEY 列约束 (Thomas)
```

按 SQL92 规范定义 USER 为 CURRENT_USER 的同义词 (Thomas)
启用 HAVING 子句但其他地方尚无修复。
使 "char" 类型成为 "char(1)" 的同义词 (实际实现为 bpchar) (Thomas)
为 DEFAULT 子句处理保存指定的字符串类型 (Thomas)
强制涉及不同数据类型的操作 (Thomas)
允许对不同类型列的某些索引使用 (Thomas)
添加自动类型转换能力 (Thomas)
清理大对象，使文件在打开时被截断 (Peter)
Readline 清理 (Tom)
允许 psql \f \ 用空格作为分隔符 (Bruce)
把 pg_attribute.atttypmod 传递给前端以获取列字段长度 (Tom、Bruce)
/contrib 中的 Msql 兼容库 (Aldrin)
移除 ORDER/GROUP BY 子句标识符必须
包含在目标列表中的要求 (David)
转换列以匹配 UNION 子句中的列 (Thomas)
移除 fork()/exec() 只做 fork() (Bruce)
Jdbc 清理 (Peter)
在 ps 命令行上显示后端状态 (只在某些平台上有效) (Bruce)
Pg_hba.conf 的 database 字段现在有 sameuser 选项
使 lo_unlink 接受 oid 参数而不是 int4
为无法处理我们宏的编译器新增 DISABLE_COMPLEX_MACRO (Bruce)
Libpgtcl 现在把 NOTIFY 作为 Tcl 事件处理，不必发送哑查询 (Tom)
libpgtcl 清理 (Tom)
向 libpgtcl 的 pg_result 命令添加 -error 选项 (Tom)
新区域设置补丁，参见 docs/README/locale (Oleg)
修复 pg_dump 使 CONSTRAINT 和 CHECK 语法正确 (ccb)
新 contrib/lo 代码用于移除孤立大对象 (Peter)
多字节特性的新 psql 命令 "SET CLIENT_ENCODING TO 'encoding'",
参见 /doc/README.mb (Tatsuo)
contrib/noupdate 代码用于撤销列上的更新权限
libpq 现在可以在 Windows 上编译 (Magnus)
在 libpq 中添加 PQsetdbLogin()
新 8 字节整数类型，由 configure 检查操作系统支持 (Thomas)
更好地支持带引号的表/列名 (Thomas)
在 pg_dump 中用双引号包围表和列名 (Thomas)
PQreset() 现在可用于口令 (Tom)
处理 GROUP BY 目标列表列号越界的情形 (David)
允许子查询中的 UNION
向 \d? 命令添加屏幕自动调整大小 (Bruce)
用 UNION 在一个查询中显示所有 \d? 结果 (Bruce)
添加 \d? 字段搜索特性 (Bruce)
Pg_dump 发出更少的 \connect 请求 (Tom)
使 pg_dump -z 标志更好地工作并在手册页中记录 (Tom)
添加对子查询和 union 完全支持的 HAVING 子句 (Stephan)
contrib/fulltextindex 中的全文索引例程 (Maarten)
事务 id 现在存储在共享内存中 (Vadim)
发出 COPY 命令时的新 PGCLIENTENCODING (Tatsuo)
支持 SQL92 语法 "SET NAMES" (Tatsuo)
支持 LATIN2-5 (Tatsuo)
添加 UNICODE 回归测试用例 (Tatsuo)
锁管理器清理，LLL 的新锁定模式 (Vadim)
允许 OR 子句使用索引 (Bruce)
允许 "SELECT NULL ORDER BY 1;"
Explain VERBOSE 打印计划，现在还把计划美化打印到

postmaster 日志文件 (Bruce)
向 \d 命令添加索引显示 (Bruce)
允许对函数 GROUP BY (David)
大对象的新 pg_class.relkind (Bruce)
把 libpq NOTICE 消息发送到其他位置的新方法 (Tom)
psql 的新 \w 写入命令 (Bruce)
新 /contrib/findoidjoins 扫描 oid 列以发现连接关系 (Bruce)
检查含常量的限制子句的有效索引时
允许考虑二进制兼容的索引 (Thomas)
/contrib/isbn_issn 中的新 ISBN/ISSN 代码
允许 NOT LIKE、IN、NOT IN、BETWEEN 和 NOT BETWEEN 约束 (Thomas)
新重写系统修复了规则和视图的许多问题 (Jan)
    * 关系上的规则可用
    * insert/update/delete 上的事件限定可用
    * 新的 OLD 变量引用 CURRENT，CURRENT 未来将被移除
    * 更新规则可在规则限定/动作中引用 NEW 和 OLD
    * 视图上的 insert/update/delete 规则可用
    * 现在支持多个规则动作，用括号包围
    * 普通用户可以在有 RULE 权限的表上创建视图/规则
    * 规则和视图继承创建者的权限
    * 没有列级别的规则
    * 没有 UPDATE NEW/OLD 规则
    * 新 pg_tables、pg_indexes、pg_rules 和 pg_views 系统视图
    * SELECT 规则只有单个动作
    * 彻底的重写改造，也许在 6.5
    * 处理子查询
    * 处理视图上的聚合
    * 处理 insert into select from view 可用
系统索引现在是多键的 (Bruce)
移除 Oidint2、oidint4 和 oidname 类型 (Bruce)
更多系统表查找使用系统缓存 (Bruce)
backend/pl 中的新后端编程语言 PL/pgSQL (Jan)
新 SERIAL 数据类型，自动创建序列/索引 (Thomas)
无需重新编译即可启用断言检查 (Massimo)
用户锁增强 (Massimo)
设置序列值的新 setval() 命令 (Massimo)
启动时若无 postmaster 运行则自动移除 unix 套接字文件 (Massimo)
条件跟踪包 (Massimo)
新 UNLISTEN 命令 (Massimo)
psql 和 libpq 现在可用 win32.mak 在 Windows 下编译 (Magnus)
Lo_read 不再存储尾随 NULL (Bruce)
标识符现在内部截断为 31 字符 (Bruce)
Createuser 选项现在可在命令行上使用
添加 64 位整数支持代码，configure 已测试，int8 类型 (Thomas)
防止失败 COPY 的文件描述符泄漏 (Bruce)
新 pg_upgrade 命令 (Bruce)
更新的 /contrib 目录 (Massimo)
新 CREATE TABLE DEFAULT VALUES 语句可用 (Thomas)
新 INSERT INTO TABLE DEFAULT VALUES 语句可用 (Thomas)
新 DECLARE 和 FETCH 特性 (Thomas)
libpq 的内部结构现在不导出 (Tom)
允许多达 8 键的索引 (Bruce)
移除 ARCHIVE 关键字，它不再使用 (Thomas)
pg_dump -n 标志以抑制标识符周围的引号

禁用视图的系统列 (Jan)
用于网络地址的新 INET 和 CIDR 类型 (TomH、Paul)
psql 输出中不再有双引号
pg_dump 现在转储视图 (Terry)
新 SET QUERY_LIMIT (Tatsuo、Jan)

源代码树变更

/contrib 清理 (Jun)
内联一些每行都调用的小函数 (Bruce)
Alpha/linux 修复
HP-UX 清理 (Tom)
多字节回归测试 (Soonmyung.)
从 configure 移除 --disabled 选项
定义 PGDOC 默认使用 POSTGRESDIR
使回归可选
移除 pgindent 的额外花括号代码 (Bruce)
添加 bsdi 共享库支持 (Bruce)
新 --without-CXX 支持 configure 选项 (Brook)
新 FAQ_CVS
更新 tools/backend 中的后端流程图 (Bruce)
把 atttypmod 从 int16 改为 int32 (Bruce、Tom)
为没有 Getrusage() 的平台提供修复 (Tom)
把 PQconnectdb、PGUSER、PGPASSWORD 添加到 libpq 手册页
NS32K 平台修复 (Phil Nelson、John Buller)
SCO 7/UnixWare 2.x 修复 (Billy 等)
Sparc/Solaris 2.5 修复 (Ryan)
Pgbuiltin.3 已过时，移到 doc 文件 (Thomas)
更多的文档 (Thomas)
Nextstep 支持 (Jacek)
Aix 支持 (David)
pginterface 手册页 (Bruce)
所有共享库都有版本号
把所有操作系统特定的共享库定义合并到一个文件
更智能的 TCL/TK 配置检查 (Billy)
更智能的 perl 配置 (Brook)
未找到 install 脚本时 configure 使用随附的 install-sh (Tom)
用于共享库配置的新 Makefile.shlib (Tom)

E.78. 版本 6.3.2

发行日期
1998-04-07

这是 6.3.x 的缺陷修复版本。有关 6.3 版本系列新特性更完整的摘要，请参阅 6.3 版本的发行说明。

摘要：

- 修复某些平台（包括 Linux）的自动配置支持，该损坏是 6.3.1 中无意引入的。

- 正确处理 BETWEEN 和 LIKE 子句左侧的函数调用。

对于运行 6.3 或 6.3.1 的用户，不需要进行转储/恢复。只需做一次 `make distclean`、`make` 和 `make install`。最后一步应在 postmaster 未运行时执行。你应重新链接任何使用 PostgreSQL 库的自定义应用程序。

对于从 6.3 之前安装的升级，请参阅 6.3 版本的安装和迁移说明。

E.78.1. 变更

对 `tcl/tk` 的配置检测改进 (`Brook Milligan`、`Alvin`)
手册页改进 (`Bruce`)
`BETWEEN` 和 `LIKE` 修复 (`Thomas`)
修复 `pg_dump` 所用 `psql \connect` 的问题 (`Oliver Elphick`)
新 `odbc` 驱动
`pgaccess 0.86` 版
移除 `qsort`，现使用 `libc` 版本并清理 (`Jeroen`)
修复检测到的缓冲区越界 (`Maurice Gittens`)
修复 `libpgtcl` 中的缓冲区越界 (`Randy Kunkee`)
修复带 `DISTINCT` 或 `ORDER BY` 的 `UNION` (`Bruce`)
`gettimeofday configure` 检查 (`Doug Winterburn`)
修复 `"indexes not used"` 缺陷 (`Vadim`)
文档补充 (`Thomas`)
修复后端内存泄漏 (`Bruce`)
`libreadline` 清理 (`Erwan MAS`)
移除 `DISTDIR` (`Bruce`)
`Makefile` 依赖清理 (`Jeroen van Vianen`)
`ASSERT` 修复 (`Bruce`)

E.79. 版本 6.3.1

发行日期
1998-03-23

摘要：

- 对多字节字符集的额外支持。
- 修复混合端序客户端和服务器的字节顺序。
- 对允许的 SQL 语法做小的更新。
- 改进安装时配置的自动检测。

对于运行 6.3 的用户，不需要进行转储/恢复。只需做一次 `make distclean`、`make` 和 `make install`。最后一步应在 postmaster 未运行时执行。你应重新链接任何使用 PostgreSQL 库的自定义应用程序。

对于从 6.3 之前安装的升级，请参阅 6.3 版本的安装和迁移说明。

E.79.1. 变更

`ecpg` 清理/修复，现为 `1.1` 版 (`Michael Meskes`)

pg_user 清理 (Bruce)
pg_dump 和 tclsh 的大对象修复 (alvin)
修复多个相邻下划线的 LIKE
修复内建函数的重定义 (Thomas)
ultrix4 清理
升级到 pg_access 0.83
更新 CLUSTER 手册页
多字节字符集支持，参见 doc/README.mb (Tatsuo)
configure --with-pgport 修复
pg_ident 修复
后端通信的大端序修复 (Kataoka)
SUBSTR() 和 substring() 修复 (Jan)
若干 jdbc 修复 (Peter)
libpgtcl 改进，参见 libptcl/README (Randy Kunkee)
修复 "Datasize = 0" 错误 (Vadim)
防止 \do 折行 (Bruce)
移除重复的俄语字符集条目
Sunos4 清理
允许 LOCK 和 SELECT INTO 中可选的 TABLE 关键字 (Thomas)
允许负整数的 CREATE SEQUENCE 选项 (Thomas)
把 "PASSWORD" 添加为允许的列标识符 (Thomas)
为 UNION 目标字段添加检查 (Bruce)
修复 Alpha 移植 (Dwayne Bailey)
修复含引号文本数组的问题 (Doug Gibson)
Solaris 编译修复 (Albert Chin-A-Young)
更好地识别 tcl 和 tk 库及头文件 (Bruce)

E.80. 版本 6.3

发行日期

1998-03-01

此版本有许多新特性和改进。以下是简要而不完整的总结：

- 许多新的 SQL 特性，包括完整的 SQL92 子查询能力（只缺目标列表子查询，其余一切都有了）。
- 支持用客户端环境变量指定时区和日期样式。
- 客户端/服务器连接的套接字接口。现在这是默认，所以你可能需要用 -i 标志启动 postmaster。
- 更好的口令授权机制。默认表权限已更改。
- 旧式时间旅行已被移除。性能得到改进。

注意

Bruce Momjian 写了以下说明来介绍新版本。

我想提一些 6.3 的普遍问题。这里只是一句话说不完的大项。仍需审阅详细变更列表。

首先，我们现在有了子查询。既然有了它们，我想指出：没有子查询的 SQL 是一种非常受限的语言。子查询是一项重大特性，你应该检查自己的代码，
看哪些地方子查询能为你的查询提供更好的解决方案。
我想你会发现子查询的用途比你可能想象的更多。Vadim 用子查询（而且是功能完整的子查询）把我们带上了 SQL 的大地图。子查询唯一不能做的事是在目标列表中使用它们。

其次，6.3 默认使用 Unix 域套接字而不是 TCP/IP。要启用来自其他机器的连接，必须使用新的 postmaster -i 选项，当然还要编辑 `pg_hba.conf`。此外，由于这个原因，`pg_hba.conf` 的格式已经更改。

第三，`char()` 字段现在允许比 `varchar()` 或 `text` 更快的访问。具体来说，`text` 和 `varchar()` 在访问该类型第一列之后的任何列时有性能损失。`char()` 过去也有这种访问损失，但现在没有了。这可能提示你重新设计一些表，特别是如果你有定义为 `varchar()` 或 `text` 的短字符列。这项和其他变更使 6.3 比以前的版本更快。

我们现在可以独立于任何 Unix 文件定义口令。有新的 SQL USER 命令。更多信息见管理员指南。还有一个新表 pg_shadow，用于存储用户信息和用户口令，默认只有 postgres 超级用户可以 SELECT。pg_user 现在是 pg_shadow 的视图，可被 PUBLIC SELECT。你的应用程序应继续使用 pg_user 而无需更改。

用户创建的表现在默认不再对 PUBLIC 授予 SELECT 权限。这样做是因为 ANSI 标准要求这样做。如果需要，你可以通过 GRANT 语句更改这些权限。

我们还有真正的死锁检测代码。不再有六十秒超时。而且新的锁定代码更好地实现了 FIFO，因此在重度使用时资源饥饿应更少。

以前的版本中有很多关于文档不足的抱怨。Thomas 在许多新文档上投入了大量精力。看看 doc/目录。

出于性能原因，时间旅行已被移除，但可以用触发器实现（参见 `pgsql/contrib/spi/README`）。请查看新的 `\d` 命令以了解类型、操作符等。此外，视图现在有自己的权限，不再基于底层表，因此必须单独设置视图的权限。查看 `/pgsql/interfaces` 了解与 PostgreSQL 交互的一些新方式。

这是第一个真正需要为现有用户做解释的版本。在许多方面，这是必要的，
因为新版本移除了许多限制，人们过去使用的变通方法不再需要。

E.80.1. 迁移到版本 6.3

希望从任何以前的 PostgreSQL 版本迁移数据的用户，需要使用 pg_dump 或 pg_dumpall 进行转储/恢复。

E.80.2. 变更

```
缺陷修复
---------
修复被 MOVE 实现破坏的二进制游标 (Vadim)
修复 tcl 库崩溃 (Jan)
修复数组处理的问题，来自 Gerhard Hintermayer
修复 acl 错误并移除重复的 pqtrace (Bruce)
修复 psql \e 对空文件的问题 (Bruce)
修复 varchar() 字段上的 textcat (Bruce)
```

修复 DBT Sendproc (Zeugswetter Andres)
修复 vacuum analyze 语法问题 (Bruce)
修复国际标识符的问题 (Tatsuo)
修复继承表上的聚合 (Bruce)
修复 substr() 的越界数据问题
修复 select 1=1 or 2=2、select 1=1 and 2=2 和 select sum(2+2) (Bruce)
修复无 tty 输出以显示状态结果。-q 选项仍会关闭它 (Bruce)
修复 count(*)、带视图和多表的聚合以及 sum(3) (Bruce)
修复 cluster (Bruce)
修复 PQtrace 多次启动/停止的问题 (Bruce)
修复各种锁定问题，如较新的锁等待者先于较旧者
获得锁、写者等待锁时读者不共享锁、等待的写者
没有优先于等待的读者 (Bruce)
修复从外部文件执行查询时 psql 的崩溃 (James)
修复多列 order by 且第一列含 NULL 值的问题 (Jeroen)
为 float8 和 int4 使用正确的哈希表支持函数 (Thomas)
重新启用 CREATE OPERATOR 语句中的 JOIN= 选项 (Thomas)
更改布尔操作符的优先级以符合预期行为 (Thomas)
对过大整数生成 elog(ERROR) (Bruce)
允许约束子句中的多参数函数 (Thomas)
检查布尔输入字面量 'true'、'false'、'yes'、'no'、'1'、'0'，
无法识别时抛出 elog(ERROR) (Thomas)
重要的大对象修复
修复 GROUP BY 显示重复项的问题 (Vadim)
修复 MergeJoin 中的索引扫描 (Vadim)

增强

带 EXISTS、IN、ALL、ANY 关键字的子查询 (Vadim、Bruce、Thomas)
新用户手册 (Thomas 等)
通过内联一些频繁调用的函数提速
真正的死锁检测，不再超时 (Bruce)
添加 SQL92 "常量" CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP、
CURRENT_USER (Thomas)
修改约束语法以符合 SQL92 (Thomas)
用索引实现 SQL92 PRIMARY KEY 和 UNIQUE 子句 (Thomas)
识别 FOREIGN KEY 的 SQL92 语法。抛出 elog 通知 (Thomas)
允许 NOT NULL UNIQUE 约束子句（以前各自单独允许） (Thomas)
允许非常量的 PostgreSQL 风格转换 ("::") (Thomas)
添加对 SQL3 TRUE 和 FALSE 布尔常量的支持 (Thomas)
支持 IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE 的 SQL92 语法 (Thomas)
允许更短的布尔字面量字符串（如 "t"、"tr"、"tru") (Thomas)
允许 SQL92 分隔标识符 (Thomas)
实现 SQL92 二进制和十六进制字符串解码 (b'10' 和 x'1F') (Thomas)
支持字面量字符串类型强制的 SQL92 语法
(如 "DATETIME 'now'") (Thomas)
添加 int2、int4 和 OID 类型与 text 之间的转换 (Thomas)
构建索引时使用共享锁 (Vadim)
事务块内为用户查询分配的内存在该查询完成后释放，
此功能在 <= 6.2.1 中被关闭 (Vadim)
新 SQL 语句 CREATE PROCEDURAL LANGUAGE (Jan)
新 PostgreSQL 过程语言 (PL) 后端接口 (Jan)
把 pg_dump -H 选项改名为 -h (Bruce)
添加 Java 对口令和欧洲日期的支持 (Peter)

为 LIKE 和 ~、!~ 操作使用索引 (Bruce)
添加 datetime 和 timespan 的哈希函数 (Thomas)
移除时间旅行 (Vadim、Bruce)
为 \d 和 \z 添加分页并修复 \i (Bruce)
向后端和前端库添加 Unix 域套接字支持 (Goran)
实现 CREATE DATABASE/WITH LOCATION 和 initlocation 工具 (Thomas)
允许更多 SQL92 和/或 PostgreSQL 保留字作为列标识符 (Thomas)
增强对 SQL92 SET TIME ZONE... 的支持 (Thomas)
SET/SHOW/RESET TIME ZONE 使用 TZ 后端环境变量 (Thomas)
实现 SET keyword = DEFAULT 和 SET TIME ZONE DEFAULT (Thomas)
启用使用 TZ 环境变量的 SET TIME ZONE (Thomas)
把 PGDATESTYLE 环境变量添加到前端和后端初始化 (Thomas)
添加 PGTZ、PGCOSTHEAP、PGCOSTINDEX、PGRPLANS、PGGEQO
前端库初始化环境变量 (Thomas)
回归测试时区用 "setenv PGTZ PST8PDT" 自动设置 (Thomas)
添加 pg_description 表以保存表、列、操作符、类型和
聚合的信息 (Bruce)
把系统表/索引名 16 字符的限制增加到 32 字符 (Bruce)
重命名系统索引 (Bruce)
向 SET DATESTYLE 添加 'GERMAN' 选项 (Thomas)
定义带 "hh:mm:ss" 字段的 "ISO 风格" timespan 输出格式 (Thomas)
允许 delta time 的小数值 (如 '2.5 days') (Thomas)
更仔细地验证 delta time 的数字输入 (Thomas)
实现把年积日作为 date_part() 的可能输入 (Thomas)
定义 timespan_finite() 和 text_timespan() 函数 (Thomas)
移除归档内容 (Bruce)
允许有独立于系统口令文件的 pg_password 认证数据库 (Todd)
转储 ACL、GRANT、REVOKE 权限 (Matt)
定义 text、varchar 和 bpchar 字符串长度函数 (Thomas)
修复继承的查询处理和代价计算 (Bruce)
实现 CREATE TABLE/AS SELECT (SELECT/INTO 的替代) (Thomas)
允许约束中的 NOT、IS NULL、IS NOT NULL (Thomas)
实现 SELECT 的 UNION (Bruce)
向 INSERT 添加 UNION、GROUP、DISTINCT (Bruce)
varchar() 在磁盘上只存储必要的字节 (Bruce)
修复 BLOB 的问题 (Peter)
JDBC 超大补丁……变更列表见 README_6.3 (Peter)
从 PQconnectdb() 移除未使用的 "option"
新 LOCK 命令和描述死锁的锁手册页 (Bruce)
添加新 psql \da、\dd、\df、\do、\dS 和 \dT 命令 (Bruce)
增强 psql \z 以显示序列 (Bruce)
在 psql \d 表中显示 NOT NULL 和 DEFAULT (Bruce)
新 psql .psqlrc 文件启动 (Andrew)
修改 contrib/linux 中的示例启动脚本以显示 syslog (Thomas)
contrib/ip_and_mac 中 IP 和 MAC 地址的新类型 (TomH)
contrib/unixdate 中与日期/时间类型的 Unix 系统时间转换 (Thomas)
contrib 内容的更新 (Massimo)
向 DBD::Pg 添加 Unix 套接字支持 (Goran)
新 python 接口 (PyGreSQL 2.0) (D'Arcy)
新前端/后端协议有版本号和网络字节序 (Phil)
pg_hba.conf 的安全特性得到增强和记录，大量清理 (Phil)
CHAR() 现在比 VARCHAR() 或 TEXT 访问更快
ecpg 嵌入式 SQL 预处理器
降低系统列开销 (Vadmin)

移除 pg_time 表 (Vadim)
添加 pg_type 属性以标识需要长度的类型 (bpchar、varchar)
COPY 命令失败时添加对问题行的报告
允许视图权限与底层表分开设置。
为安全起见，酌情对视图使用 GRANT/REVOKE (Jan)
表现在没有默认的 GRANT SELECT TO PUBLIC。你必须
显式授予此类权限。
清理教程示例 (Darren)

源代码树变更

添加新的 html 开发工具和 /tools/backend 中的流程图
修复 SCO 编译
Stratus 计算机移植 (Robert Gillies)
为 BSD44_derived & i386_solaris 添加 shlib 支持
使 configure 更加自动化 (Brook)
添加检查回归测试结果的脚本
把解析器函数拆分为更小的文件并分组 (Bruce)
把 heap_create 改名为 heap_create_and_catalog，把 heap_creatr
改名为 heap_create() (Bruce)
Sparc/Linux 的锁定补丁 (TomS)
移除 PORTNAME 并重组移植特定内容 (Marc)
添加优化器 README 文件 (Bruce)
移除优化器中的一些递归并清理其中一些代码 (Bruce)
修复 NetBSD 锁定 (Henry)
修复 libptcl 的制作 (Tatsuo)
AIX 补丁 (Darren)
把 IS TRUE、IS FALSE、... 更改为使用 "=" 的表达式而不是
对 istrue() 或 isfalse() 的函数调用以允许优化 (Thomas)
各种 NetBSD/Sparc 相关修复 (TomH)
Alpha linux 锁定 (Travis、Ryan)
把 elog(WARN) 改为 elog(ERROR) (Bruce)
FreeBSD 的 FAQ (Marc)
把 PostODBC 源代码树纳入我们的标准发行版 (Marc)
HP/UX 10 与 9 的小补丁 (Stan)
新 pg_attribute.atttypmod 用于 varchar 长度等类型特定信息 (Bruce)
UnixWare 补丁 (Billy)
自旋锁 asm 的新 i386 'lock' (Billy)
移除对多路复用后端的支持
开始 OpenBSD 移植
开始 AUX 移植
开始 Cygnus 移植
向回归套件添加字符串函数 (Thomas)
扩展一些以前截断为 16 字符的函数名 (Thomas)
移除不需要的 malloc() 调用并用 palloc() 替换 (Bruce)

E.81. 版本 6.2.1

发行日期
1997-10-17

6.2.1 是 6.2 之上的缺陷修复和易用性版本。

摘要：

- 按照 SQL92，允许字符串跨行。
- 包含在表更新时插入用户名的示例触发器函数。

这是 6.2 之上的次要缺陷修复版本。从 6.2 之前的系统升级需要完整的转储/重载。请参阅 6.2 版本的发行说明了解操作说明。

E.81.1. 从版本 6.2 迁移到版本 6.2.1

这是次要缺陷修复版本。从 6.2 版本不需要转储/重载，但从 6.2 之前的任何版本需要。

从 6.2 版本升级时，如果你选择转储/重载，你会发现 avg(money) 现在计算正确。所有其他缺陷修复在更新可执行文件后生效。

避免转储/重载的另一种方法是从 `psql` 使用以下 SQL 命令更新现有的系统表：

```
update pg_aggregate set aggfinalfn = 'cash_div_flt8'
 where aggname = 'avg' and aggbasetype = 790;
```

这需要对每个现有数据库执行，包括 template1。

E.81.2. 变更

允许 `TIME` 和 `TYPE` 作为列名 (`Thomas`)
允许更大范围的 `true/false` 布尔值 (`Thomas`)
支持 `"now"` 和 `"current"` 的输出 (`Thomas`)
正确处理 `INSERT NULL` 与 `DEFAULT` (`Vadim`)
修复缓冲区管理器中关系引用计数的问题 (`Vadim`)
像 `ANSI` 一样允许字符串跨行 (`Thomas`)
修复带 `ORDER BY` 的反向游标 (`Vadim`)
修复 `avg(cash)` 计算 (`Thomas`)
修复 `ORDER/GROUP BY` 中重复指定列的问题 (`Vadim`)
记录返回受影响行数的新 `libpq` 函数 `PQcmdTuples` (`Bruce`)
用于在 `INSERT/UPDATE` 时插入用户名的触发器函数 (`Brook Milligan`)

E.82. 版本 6.2

发行日期
1997-10-02

希望从以前的 PostgreSQL 版本迁移数据的用户，需要进行转储/恢复。

E.82.1. 从版本 6.1 迁移到版本 6.2

此迁移需要完整转储 6.1 数据库并在 6.2 中恢复。

注意，应使用 6.2 的 `pg_dump` 和 `pg_dumpall` 工具来转储 6.1 数据库。如果使用 6.2 之前版本的这些工具，把 template0 复制到 template1 数据库的步骤会失败。

E.82.2. 从版本 1.*x* 迁移到版本 6.2

从更早的 1.* 版本迁移的用户应先升级到 1.09，因为 COPY 输出格式自 1.02 版本起得到了改进。

E.82.3. 变更

```
缺陷修复
---------
修复 pg_dump 对继承、序列、归档表的问题 (Bruce)
修复 Solaris 上因移位、无符号和坏原型
        导致的溢出编译错误 (Diab Jerius)
修复几何线段算术中的缺陷（错误的交点计算） (Thomas)
在端点处检查几何相交以避免舍入难看 (Thomas)
捕获无效的删除尝试 (Vadim)
把时间函数名改得更一致 (Michael Reifenberg)
检查除零 (Michael Reifenberg)
修复使命令更改/插入的行对命令本身可见的很久以来的缺陷
        （因此曾出现已更新行的多次更新等） (Vadim)
修复 SELECT null, 'fail' FROM pg_am (Patrick)
现在允许 SELECT NULL as EMPTY_FIELD (Patrick)
从 contrib/pginterface 移除不需要的信号代码
修复 OR (where x != 1 or x isnull 不返回 x 为 NULL 的行) (Vadim)
修复 time_cmp 函数 (Vadim)
修复 WHERE 子句中第一个参数非属性的
       函数的处理 (Vadim)
修复条目顺序与目标列表顺序不同时的
       GROUP BY (Vadim)
修复 pg_dump 对无 sfunc1 聚合的处理 (Vadim)

增强
------------
默认遗传优化器 GEQO 参数现在是 8 (Bruce)
允许在目标列表中使用函数内带聚合的参数 (Vadim)
添加 JDBC 驱动作为接口 (Adrian & Peter)
pg_password 工具
返回 INSERT/UPDATE/DELETE 等插入/影响的行数 (Vadim)
用 CREATE TRIGGER 实现触发器 (SQL3) (Vadim)
SPI (服务器编程接口) 允许在 C 函数内执行查询 (Vadim)
实现 NOT NULL (SQL92) (Robson Paniago de Miranda)
包含用于字符串处理、外连接和并集的保留字 (Thomas)
用独占状态实现扩展注释 ("/* ... */") (Thomas)
添加 "//" 单行注释 (Bruce)
移除操作符名中字符的一些限制 (Thomas)
实现表的 DEFAULT 和 CONSTRAINT (SQL92) (Vadim & Thomas)
添加文本连接操作符和函数 (SQL92) (Thomas)
支持 WITH TIME ZONE 语法 (SQL92) (Thomas)
支持 INTERVAL unit TO unit 语法 (SQL92) (Thomas)
定义类型 DOUBLE PRECISION、INTERVAL、CHARACTER
       和 CHARACTER VARYING (SQL92) (Thomas)
定义类型 FLOAT(p) 和基础的 DECIMAL(p,s)、NUMERIC(p,s) (SQL92) (Thomas)
定义 EXTRACT()、POSITION()、SUBSTRING() 和 TRIM() (SQL92) (Thomas)
定义 CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP (SQL92) (Thomas)
为 UNION、HAVING、INNER 和 OUTER JOIN 添加语法和警告 (SQL92) (Thomas)
```

添加更多保留字，主要为符合 SQL92 (Thomas)
允许 timespan/reltime 类型的 hh:mm:ss 时间输入 (Thomas)
为 lseg、path、polygon 添加 center() 例程 (Thomas)
为 circle-polygon、polygon-polygon 添加 distance() 例程 (Thomas)
用轴交叉算法显式检查多边形内包含的点和多边形 (Thomas)
添加 circle-box 转换例程 (Thomas)
合并不同几何数据类型的冲突操作符 (Thomas)
把距离操作符 "<===>" 替换为 "<->" (Thomas)
把"上方"操作符 "!^" 替换为 ">^", "下方"操作符 "!|" 替换为 "<^" (Thomas)
添加两端文本修剪、子串和字符串位置例程 (Thomas)
添加转换例程 circle(box) 和 poly(circle) (Thomas)
允许内部排序存储在内存中而不是文件中 (Bruce & Vadim)
允许内部相同类型上的函数和操作符成功 (Bruce)
通过性能分析加快后端启动 (Bruce)
内联频繁调用的函数以提升性能 (Bruce)
减少 open() 调用 (Bruce)
psql: 为 \h 和 \? 添加 PAGER, \C 修复
修复无 tty 时 psql 分页器的问题 (Bruce)
新 entab 工具 (Bruce)
用于参照完整性的一般触发器函数 (Vadim)
用于时间旅行的一般触发器函数 (Vadim)
用于 AUTOINCREMENT/IDENTITY 特性的一般触发器函数 (Vadim)
MOVE 实现 (Vadim)

源代码树变更

HP-UX 10 补丁 (Vladimir Turin)
添加 SCO 支持 (Daniel Harris)
MkLinux 补丁 (Tatsuo Ishii)
把几何 box 术语从 "length" 改为 "width" (Thomas)
废弃几何代码中的临时未存储斜率字段 (Thomas)
从 INSTALL 移除重启说明 (Bruce)
先在 /usr/ucb 中查找 install (Bruce)
修复 c++ 复制示例代码 (Thomas)
在 psql 手册页添加 -o (Bruce)
防止 relname 未分配字符串长度被复制进数据库 (Bruce)
清理 NAMEDATALEN 的使用 (Bruce)
修复输出中超过 15 字符的 pg_proc 名 (Bruce)
添加 strNcpy() 函数 (Bruce)
移除一些不必要的 (void) 转换 (Bruce)
新的 interfaces 目录 (Marc)
用对 fd.c 函数的调用替换 fopen() 调用 (Bruce)
尽可能把函数设为 static (Bruce)
把未使用的函数包在 #ifdef NOT_USED 中 (Bruce)
移除时间戳支持中对 difftime() 的调用以修复 SunOS (Bruce & Thomas)
针对 Digital Unix 的变更
pg_dumpall 的可移植性修复 (Bruce)
把 pg_attribute.attnvals 改名为 attdispersion (Bruce)
"intro/unix" 手册页现为 "pgintro" (Bruce)
"built-in" 手册页现为 "pgbuiltin" (Bruce)
"drop" 手册页现为 "drop_table" (Bruce)
添加 "create_trigger"、"drop_trigger" 手册页 (Thomas)
添加约束回归测试 (Vadim & Thomas)
添加注释语法回归测试 (Thomas)

添加 PGINDENT 和支持程序 (Bruce)
对所有 *.c 和 *.h 文件运行 PGINDENT 的大规模提交 (Bruce)
文件移到 /src/tools 目录 (Bruce)
SPI 和触发器编程指南 (Vadim & D'Arcy)

E.83. 版本 6.1.1

发行日期
1997-07-22

E.83.1. 从版本 6.1 迁移到版本 6.1.1

这是次要缺陷修复版本。从 6.1 版本不需要转储/重载，但从 6.1 之前的任何版本需要。更多细节请参阅 6.1 的发行说明。

E.83.2. 变更

修复带选项的 SET (Thomas)
允许 pg_dump/pg_dumpall 保留所有表/对象的所有权 (Bruce)
新 psql \connect 选项允许只更改用户名而不更改数据库
修复 initdb --debug 选项 (Yoshihiko Ichikawa))
lextest 清理 (Bruce)
哈希修复 (Vadim)
修复日期/时间的月份边界算术 (Thomas)
修复某些移植的时区夏令时处理 (Thomas、Bruce、Tatsuo)
timestamp 彻底改造为使用标准函数 (Thomas)
日期/时间例程的其他代码清理 (Thomas)
psql 的 \d 现在不区分大小写 (Bruce)
psql 的反斜杠命令现在可以带尾部分号 (Bruce)
修复使用 \g 时 psql 的内存泄漏 (Bruce)
与服务器通信的字节序处理重大修复 (Thomas、Tatsuo)
Solaris 汇编器和头文件的修复 (Yoshihiko Ichikawa)
允许用户名中使用下划线 (Bruce)
pg_dumpall 现在返回正确状态，可移植性修复 (Bruce)

E.84. 版本 6.1

发行日期
1997-06-08

回归测试已经为 PostgreSQL 6.1 版本做了适配和大量修改。

新增了三种数据类型（datetime、timespan 和 circle），已加入到 PostgreSQL 本地类型集中。点、框、路径和多边形在各数据类型间的输出格式已统一。misc.out 中的多边形输出只相对原始回归输出做了抽查。

PostgreSQL 6.1 引入了使用遗传算法的新替代优化器。当查询包含多个限定条件或多个表时（优化器可选择求值顺序），这些算法会给查询结果的顺序引入随机行为。
已有几个回归测试被修改为显式排序结果，因此对优化器的选择不敏感。
少数回归测试针对本质上无序的数据类型（如点和时间间隔），涉及这些类型的测试显式地用 `set geqo to 'off'` 和 `reset geqo` 括起来。

数组说明符（原子值周围的花括号）
的解释似乎在最初的回归测试生成之后的某个时候发生了变化。当前的 `./expected/*.out` 文件反映了这一新解释，它可能并不正确！

float8 回归测试至少在某些平台上失败。这是由于 `pow()` 和 `exp()` 实现的差异以及上溢/下溢条件所用的信号机制不同。

random 测试中的“random”结果会使“random”测试被标记为“failed”，因为回归测试是用简单的 diff 评估的。不过，“random”在我的测试机（Linux/gcc/i686）上似乎不会产生随机结果。

E.84.1. 迁移到版本 6.1

此迁移需要完整转储 6.0 数据库并在 6.1 中恢复。

从更早的 1.* 版本迁移的用户应先升级到 1.09，因为 COPY 输出格式自 1.02 版本起得到了改进。

E.84.2. 变更

```
缺陷修复
---------
库例程中的包长度检查
锁管理器优先级补丁
检查 float8 的下溢/上溢 (Bruce)
多表连接修复 (Vadim)
SIGPIPE 崩溃修复 (Darren)
大对象修复 (Sven)
允许 btree 索引处理 NULL (Vadim)
时区修复 (D'Arcy)
无行时 select SUM(x) 可返回 NULL (Thomas)
内部优化器、执行器缺陷修复 (Vadim)
修复 < 或 <= 内层循环无行的问题 (Vadim)
防止连接索引子句重新交换 (Vadim)
修复多表的连接子句 (Vadim)
修复数组的哈希和哈希连接 (Vadim)
修复 abstime 类型的 btree (Vadim)
大对象修复 (Raymond)
修复哈希索引中的缓冲区泄漏 (Vadim)
修复 rtree 以用于内层扫描 (Vadim)
修复 gist 以用于内层扫描并清理 (Vadim、Andrea)
避免不必要的本地缓冲区分配 (Vadim、Massimo)
修复事务中止时本地缓冲区泄漏 (Vadim)
修复文件管理器内存泄漏并清理 (Vadim、Massimo)
修复存储管理器内存泄漏 (Vadim)
修复 btree 重复项处理 (Vadim)
修复 vacuum 导致已删行复活的问题 (Vadim)
修复 SELECT varchar()/char() INTO TABLE 生成零长度字段的问题 (Bruce)
用 Purify 修复许多 psql、pg_dump 和 libpq 内存泄漏 (Igor)
```

增强

属性优化统计 (Bruce)
快得多的新 btree 批量装载代码 (Paul)
批量装载代码加入 BTREE UNIQUE (Vadim)
新锁调试代码 (Massimo)
libpg++ 的大规模变更 (Leo)
新 GEQO 优化器加速多表优化 (Martin)
对唯一键非唯一插入的新 WARN 消息 (Marc)
update x=-3 (无空格) 现在有效 (Bruce)
移除标识符的大小写敏感处理 (Bruce、Thomas、Dan)
调试后端现在美化打印树 (Darren)
新 Oracle 字符函数 (Edmund)
新明文口令函数 (Dan)
"no such class or insufficient privilege" 拆分为不同的消息 (Dan)
新 ANSI timestamp 函数 (Dan)
新 ANSI Time 和 Date 类型 (Thomas)
在后端中移动大数据块 (Martin)
多列 btree 索引 (Vadim)
新 SET var TO value 命令 (Martin)
读取时更新事务状态 (Dan)
字符类型的新区域设置 (Oleg)
新 SEQUENCE 序列号生成器 (Vadim)
现在可以使用 GROUP BY 函数 (Vadim)
重新组织回归测试 (Thomas、Marc)
新优化器操作权重 (Vadim)
新 psql \z 授予/许可选项 (Marc)
新 MONEY 数据类型 (D'Arcy、Thomas)
tcp 套接字通信速度改进 (Vadim)
VACUUM 的属性统计及特定列新选项 (Vadim)
许多几何类型改进 (Thomas、Keith)
附加回归测试 (Thomas)
新 datestyle 变量 (Thomas、Vadim、Martin)
更多用于排序的比较操作符 (Thomas)
新转换函数 (Thomas)
新更紧凑的 btree 格式 (Vadim)
允许 pg_dumpall 保留数据库所有权 (Bruce)
新 SET GEQO=# 和 R_PLANS 变量 (Vadim)
旧 (!GEQO) 优化器可使用右侧计划 (Vadim)
SQL 解析器类型检查改进 (Bruce)
新 SET、SHOW、RESET 命令 (Thomas、Vadim)
新 \connect database USER 选项
新 destroydb -i 选项 (Igor)
新 \dt 和 \di psql 命令 (Darren)
SELECT "\n" 现在转义换行 (A. Duursma)
从旧格式转换的新几何转换函数 (Thomas)

源代码树变更

新配置脚本 (Marc)
添加 readline 配置选项 (Marc)
移除操作系统特定的配置选项 (Marc)
新的操作系统特定模板文件 (Marc)

不再需要编辑 Makefile.global (Marc)
重新安排头文件 (Marc)
nextstep 补丁 (Gregor Hoffleit)
移除 Windows 特定代码 (Bruce)
移除 postmaster -e 选项，现在只有 postgres -e 选项 (Bruce)
合并前后端中重复的库代码 (Martin)
现在可与 eBones、国际 Kerberos 一起工作 (Jun)
更多共享库支持
c++ 头文件清理 (Bruce)
对有缺陷的 flex 发出警告 (Bruce)
DG/UX、Ultrix、IRIX、AIX 可移植性修复

E.85. 版本 6.0

发行日期
1997-01-29

希望从以前的 PostgreSQL 版本迁移数据的用户，需要进行转储/恢复。

E.85.1. 从版本 1.09 迁移到版本 6.0

此迁移需要完整转储 1.09 数据库并在 6.0 中恢复。

E.85.2. 从 1.09 之前版本迁移到版本 6.0

从更早的 1.* 版本迁移的用户应先升级到 1.09，因为 COPY 输出格式自 1.02 版本起得到了改进。

E.85.3. 变更

缺陷修复

ALTER TABLE 缺陷—运行中的 postgres 进程需要重新读取表定义
允许对一个表或整个数据库运行 vacuum (Bruce)
数组修复
修复数组内存写的越界 (Kurt)
修复难以捉摸的 btree 范围/非范围缺陷 (Dan)
修复某些类型（如 time 和 date）上的哈希索引
修复 pg_log 大小爆炸的问题
修复 lo_export() 的权限 (Bruce)
修复未初始化的内存读取 (Kurt)
修复 ALTER TABLE ... char(3) 缺陷 (Bruce)
修复了几个小的内存泄漏
修复 EXPLAIN 的选项处理并更改了 full_path 选项名
修复组 acl 权限的输出
内存泄漏（用 Purify 等工具查找并消灭）(Kurt)
规则系统的小改进
NOTIFY 修复
新的运行检查断言
彻底改造 parser/analyze 代码以正确报告错误并提高速度

Pg_dump -d 现在正确处理 NULL (Bruce)
防止 SELECT NULL 使服务器崩溃 (Bruce)
INSERT ... SELECT 列不匹配时正确报告错误
插入列名不正确时正确报告错误
psql \g filename 现在可用 (Bruce)
修复 psql 一行多语句多输出的问题
移除重复的系统 OID
SELECT * INTO TABLE . GROUP/ORDER BY 在表已存在时给出 unlink 错误 (Bruce)
几项对使后端崩溃查询的修复
插入字符串中起始引号的错误 (Bruce)
提交空查询现在返回空状态，而不只是 " " 查询 (Bruce)

增强

添加 EXPLAIN 手册页 (Bruce)
添加 UNIQUE 索引能力 (Dan)
添加主机名/用户级访问控制，而不只是主机名和用户
为 <> 添加 != 同义词 (Bruce)
允许 "select oid,* from table"
允许 BY、ORDER BY 按编号或按非别名的 table.column 指定列 (Bruce)
允许从前端 COPY (Bryan)
允许 GROUP BY 使用别名列名 (Bruce)
允许真正的压缩，而不只是同一页上的重用 (Vadim)
允许安装配置选项自动添加所有本地用户 (Bryan)
允许 libpq 区分文本值 '' 和 null (Bruce)
允许有 createdb 权限的非 postgres 用户 destroydb
允许限制谁可以创建 C 函数 (Bryan)
允许限制谁可以做后端 COPY (Bryan)
可以收缩表、pg_time 和 pg_log (Vadim & Erich)
更改调试级别 2 为只打印查询，更改调试标题布局 (Bruce)
把十进制常量的默认表示从 float4 改为 float8 (Bruce)
postmaster 启动时现在设置欧洲日期格式
找不到精确大小写时执行小写函数名
聚合/GROUP 处理的修复，允许 'select sum(func(x),sum(x+y) from z'
Gist 现在包含在发行版中 (Marc)
本地用户的 Ident 认证 (Bryan)
实现 BETWEEN 限定符 (Bruce)
实现 IN 限定符 (Bruce)
libpq 有 PQgetisnull() (Bruce)
libpq++ 改进
initdb 的新选项 (Bryan)
Pg_dump 允许转储 OID (Bruce)
Pg_dump 为提高速度在表装载后创建索引 (Bruce)
Pg_dumpall 转储所有数据库和用户表
Pginterface 对 NULL 值的补充 (Bruce)
防止以 root 运行 postmaster
psql \h 和 \? 现在可读 (Bruce)
psql 允许行内任意位置的反斜杠和分号 (Bruce)
psql 更改查询中或引号中行的命令提示符 (Bruce)
psql char(3) 现在在 \d 输出中显示为 (bp)char (Bruce)
psql 返回码现在更准确 (Bryan?)
psql 更新的帮助语法 (Bruce)
重新检查并修复 vacuum (Vadim)
减小回归 diff 的大小，移除时区名差异 (Bruce)

移除编译期参数以支持二进制发行 (Bryan)
反转 HBA 掩码的含义 (Bryan)
本地用户的安全认证 (Bryan)
加快 vacuum (Vadim)
Vacuum 现在有 VERBOSE 选项 (Bruce)

源代码树变更

所有函数现在都有与调用比较的原型
允许从 Makefile.global 轻松禁用断言 (Bruce)
把代码中使用的 oid 常量改为 #define 名称
解耦 sparc 和 solaris 定义 (Kurt)
Gcc -Wall 干净编译，警告只来自无法修复的构造
主要的头文件重组/精简 (Marc)
编译失败时 make 现在停止 (Bryan)
Makefile 重构 (Bryan、Marc)
把 bsdi_2_1 合并到 bsdi (Bruce)
移除 Monitor 程序
从 Postgres95 改名为 PostgreSQL
新 config.h 文件 (Marc、Bryan)
PG_VERSION 现在设为 6.0 并被 postmaster 使用
可移植性补充，包括 Ultrix、DG/UX、AIX 和 Solaris
减少 #define 的数量，集中化 #define
移除系统表中的重复 OID (Dan)
移除重复的系统目录信息或报告不匹配 (Dan)
移除了许多操作系统特定的 #define
重构目标文件的生成/位置 (Bryan、Marc)
重构移植特定文件的位置 (Bryan、Marc)
纠正未使用/未初始化的变量

E.86. 版本 1.09

发行日期
1996-11-04

抱歉，我们没有记录 1.02 到 1.09 的变更。6.0 中列出的一些变更实际上包含在 1.02.1 到 1.09 的版本中。

E.87. 版本 1.02

发行日期
1996-08-01

E.87.1. 从版本 1.02 迁移到版本 1.02.1

这是 1.02.1 的新迁移文件。它包括 'copy' 变更和一个把旧 ASCII 文件转换为新格式的脚本。

> **注意**
>
> 以下说明是为了帮助希望把数据库从 Postgres95 1.01 和 1.02 迁移到 Postgres95 1.02.1 的用户。
>
> 如果你全新开始使用 Postgres95 1.02.1 而不需要迁移旧数据库，则无需再往下读。

为了把较旧的 Postgres95 1.01 或 1.02 版本数据库升级到 1.02.1 版本，需要以下步骤：

1. 启动新的 1.02.1 postmaster

2. 把 1.02.1 的新内建函数和操作符添加到 1.01 或 1.02 数据库。方法是用新的 1.02.1 服务器运行你自己的 1.01 或 1.02 数据库，并应用文件末尾随附的查询。这可以通过 `psql` 轻松完成。如果你的 1.01 或 1.02 数据库命名为 `testdb`，并且你已从本文件末尾剪下命令并保存到 `addfunc.sql`：

   ```
   % psql testdb -f addfunc.sql
   ```

   升级 1.02 数据库的用户在执行文件中的最后两条语句时会得到一个警告，因为它们已存在于 1.02 中。这不必担心。

E.87.2. 转储/重载程序

如果你试图重新装载用以前版本生成的 pg_dump 或文本模式 `copy tablename to stdout` 输出，需要先对 ASCII 文件运行随附的 `sed` 脚本再装入数据库。旧格式用 '.' 作为数据结束，而 '\.' 现在是数据结束标记。此外，空字符串现在装填为 '' 而不是 NULL。完整细节参见 copy 手册页。

```
sed 's/^\.$/\\./g' <in_file >out_file
```

如果你在装载较旧的二进制备份或非 stdout 副本，则没有数据结束字符，因此无需转换。

```
-- following lines added by agc to reflect the case-insensitive
-- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure =
 texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure =
 texticregexne);
```

E.87.3. 变更

```
源代码维护和开发
  * 全球志愿者团队
  * 源代码树现在在 ftp.ki.net 的 CVS 中

增强
  * psql（及底层 libpq 库）现在有更多输出格式化选项，
    包括 HTML
```

```
 * pg_dump 现在输出模式和/或数据，并有许多提升完整性的修复。
 * 管理 shell 脚本中用 psql 代替 monitor。
   monitor 将在下个版本中废弃。
 * 日期/时间函数增强
 * NULL 插入/更新/比较修复/增强
 * TCL/TK 库和 shell 修复为同时支持 tck7.4/tk4.0 和 tcl7.5/tk4.1

缺陷修复（几乎多到无法列举）
 * 索引
 * 存储管理
 * 解引用前检查 NULL 指针
 * Makefile 修复

新移植
 * 新增 SolarisX86 移植
 * 新增 BSD/OS 2.1 移植
 * 新增 DG/UX 移植
```

E.88. 版本 1.01

发行日期
1996-02-23

E.88.1. 从版本 1.0 迁移到版本 1.01

以下说明是为了帮助希望把数据库从 Postgres95 1.0 迁移到 Postgres95 1.01 的用户。

如果你全新开始使用 Postgres95 1.01 而不需要迁移旧数据库，则无需再往下读。

为了在用 Postgres95 1.0 版本创建的数据库上运行 Postgres95 1.01 版本，需要以下步骤：

1. 把 `src/Makefile.global` 中 `NAMEDATALEN` 的定义改为 16，把 `OIDNAMELEN` 的定义改为 20。

2. 决定是否要使用基于主机的认证。

   a. 如果要用，你必须在顶级数据目录（通常是 `$PGDATA` 的值）中创建名为 `pg_hba` 的文件。`src/libpq/pg_hba` 给出了示例语法。

   b. 如果不想使用基于主机的认证，可以注释掉这一行：

      ```
      HBA = 1
      ```

      （位于 `src/Makefile.global`）

      注意，基于主机的认证默认开启，如果你不执行上述 A 或 B 步骤，开箱即用的 1.01 将不允许你连接到 1.0 数据库。

3. 编译并安装 1.01，但不要执行 `initdb` 步骤。

4. 在做任何其他事情之前，终止你的 1.0 postmaster，并备份现有的 `$PGDATA` 目录。

5. 把 PGDATA 环境变量指向你的 1.0 数据库，但把路径设置为使用 1.01 二进制文件。
6. 把文件 $PGDATA/PG_VERSION 从 5.0 改为 5.1
7. 启动新的 1.01 postmaster
8. 把 1.01 的新内建函数和操作符添加到 1.0 数据库。方法是用新的 1.01 服务器运行你自己的 1.0 数据库，并应用随附保存在 1.0_to_1.01.sql 文件中的查询。这可以通过 psql 轻松完成。如果你的 1.0 数据库名为 testdb：

   ```
   % psql testdb -f 1.0_to_1.01.sql
   ```

   然后执行以下命令（从此处剪切粘贴）：

   ```
   -- add builtin functions that are new to 1.01

   create function int4eqoid (int4, oid) returns bool as 'foo'
   language 'internal';
   create function oideqint4 (oid, int4) returns bool as 'foo'
   language 'internal';
   create function char2icregexeq (char2, text) returns bool as 'foo'
   language 'internal';
   create function char2icregexne (char2, text) returns bool as 'foo'
   language 'internal';
   create function char4icregexeq (char4, text) returns bool as 'foo'
   language 'internal';
   create function char4icregexne (char4, text) returns bool as 'foo'
   language 'internal';
   create function char8icregexeq (char8, text) returns bool as 'foo'
   language 'internal';
   create function char8icregexne (char8, text) returns bool as 'foo'
   language 'internal';
   create function char16icregexeq (char16, text) returns bool as
    'foo'
   language 'internal';
   create function char16icregexne (char16, text) returns bool as
    'foo'
   language 'internal';
   create function texticregexeq (text, text) returns bool as 'foo'
   language 'internal';
   create function texticregexne (text, text) returns bool as 'foo'
   language 'internal';

   -- add builtin functions that are new to 1.01

   create operator = (leftarg = int4, rightarg = oid, procedure =
    int4eqoid);
   create operator = (leftarg = oid, rightarg = int4, procedure =
    oideqint4);
   create operator ~* (leftarg = char2, rightarg = text, procedure =
    char2icregexeq);
   create operator !~* (leftarg = char2, rightarg = text, procedure =
    char2icregexne);
   create operator ~* (leftarg = char4, rightarg = text, procedure =
    char4icregexeq);
   ```

```
create operator !~* (leftarg = char4, rightarg = text, procedure =
 char4icregexne);
create operator ~* (leftarg = char8, rightarg = text, procedure =
 char8icregexeq);
create operator !~* (leftarg = char8, rightarg = text, procedure =
 char8icregexne);
create operator ~* (leftarg = char16, rightarg = text, procedure =
 char16icregexeq);
create operator !~* (leftarg = char16, rightarg = text, procedure
 = char16icregexne);
create operator ~* (leftarg = text, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = text, rightarg = text, procedure =
 texticregexne);
```

E.88.2. 变更

不兼容变更:
* 只要用户按照 MIGRATION_from_1.0_to_1.01 文件中概述的步骤操作，1.01 与 1.0 数据库向后兼容。
  如果未采取这些步骤，1.01 与 1.0 数据库不兼容。

增强:
* 为 libpq 添加 PQdisplayTuples() 并更改 monitor 和 psql 使用它
* 新增 NeXT 移植（需要 SysVIPC 实现）
* 新增 CAST .. AS 语法
* 为 libpq 添加 PQtrace()/PQuntrace()
* 新增为表字段设置默认值的能力
* 新增了出色的 QNX 移植
* 新增了许多新的内置函数
* 在 tclsh 中支持字符串数组

E.89. 版本 1.0

发行日期
1995-09-05

E.89.1. 变更

版权变更:
* Postgres 1.0 的版权已放宽为可自由修改并可用于任何目的。
  请阅读 COPYRIGHT 文件。感谢 Michael Stonebraker 教授使之成为可能。

不兼容变更:
* 日期格式必须为 MM-DD-YYYY（或使用欧式风格时的 DD-MM-YYYY）。这遵循 SQL-92 规范。
* "double precision"（浮点数）而不是 "float8" 字面量
  （float8 现在自动提升为 double precision）
* 直接调用内部日期/时间例程（例如 abstimein）的输入现在产生错误

（内部例程继续存在，但从 SQL 解释器调用它们现在给出错误）
* 在 "create" 语句中，现在允许（且仅允许）为列加上标签

新特性：
* 为 SELECT 结果提供终端监控器 psql（附带 readline 支持）
* 为列增加唯一的行标识符，使用新的类型名 "oid"
* 服务器同时支持 SQL 和 POSTQUEL 语法
* 无重复字段名限制；可以有任意数量的同名列
* DELETE 语句可在 RSELECT 语句中使用子句
* 若启用了访问控制，则所有用户需要认证访问

E.90. Postgres95 Release 0.03

发行日期
1995-07-21

E.90.1. 变更

不兼容变更：
* BETA-0.3 与以前版本创建的数据库不兼容
（由于系统目录和索引结构的变更）。
* 双引号（"）作为字符串字面量的引号字符已废弃；
需要把它们转换为单引号（'）。
* 聚合名（如 int4sum）已按照 SQL 标准改名（如 sum）。
* CHANGE ACL 语法被 GRANT/REVOKE 语法取代。
* 浮点字面量（如 3.14）现在为 float4 类型（以前版本中是 float8）；
如果你依赖它是 float8 类型，可能需要做类型转换。如果忽略类型转换
而把浮点字面量赋给 float8 类型的字段，可能会存储不正确的值！
* LIBPQ 已彻底改造，使前端应用程序
可以连接多个后端
* pg_user 中的 usesysid 字段从 int2 改为 int4，
以允许更大范围的 Unix 用户 id。
* netbsd/freebsd/bsd 操作系统移植已合并为
单一的 BSD44_derived 移植。（感谢 Alistair Crooks）

SQL 标准符合性（以下详述使 postgres95
更符合 SQL-92 标准的变更）：
* 以下 SQL 类型现在是内建的：smallint、int(eger)、float、real、
char(N)、varchar(N)、date 和 time。

以下是对现有 postgres 类型的别名：
```
smallint -> int2
integer, int -> int4
float, real  -> float4
```
char(N) 和 varchar(N) 实现为截断的文本类型。此外，
char(N) 做空白填充。
* 单引号（'）用于引用字符串字面量；''（除
\' 外）也支持作为在字符串中插入单引号的手段
* 使用 SQL 标准聚合名（MAX、MIN、AVG、SUM、COUNT）
（此外，聚合现在可以重载，即你可以定义自己的 MAX 聚合来接受用户定义的类型。）

```
 * 移除 CHANGE ACL。添加 GRANT/REVOKE 语法。
   - 可以使用 "GROUP" 关键字向组授予权限。
       例如:
               GRANT SELECT ON foobar TO GROUP my_group;
       也支持关键字 'PUBLIC' 表示所有用户。

       一次只能向一个用户或组授予或撤销权限。

       不支持 "WITH GRANT OPTION"。只有类所有者可以更改
       访问控制
   - 默认访问控制是给用户只读访问。
     必须显式向用户授予插入/更新权限。要更改这一点，请修改
               src/backend/utils/acl.h
     中定义 ACL_WORLD_DEFAULT 的那一行

缺陷修复:
 * 修复了空表聚合不运行的缺陷。现在，
   对空表运行的聚合将返回聚合的初始条件。因此，空表的 COUNT 现在正确返回 0。
   空表的 MAX/MIN 将返回值为 NULL 的一行。
 * 允许在 monitor 中使用 \;
 * LISTEN/NOTIFY 异步通知机制现在可用
 * 规则动作体中的 NOTIFY 现在可用
 * 哈希索引可用，访问方法总体上性能应更好。
   大 btree 索引的创建应快得多。(感谢 Paul
   Aoki)

其他变更和增强:
 * 新增 EXPLAIN 语句用于解释查询执行计划
   (例如 "EXPLAIN SELECT * FROM EMP" 打印该查询的执行计划)。
 * WARN 和 NOTICE 消息不再带时间戳。要打开错误消息的时间戳，请取消注释
   src/backend/utils/elog.h 中的这一行:
       /* define ELOG_TIMESTAMPS */
 * 发生访问控制违规时，将给出消息
       "Either no such class or insufficient privilege"。
   这与找不到类时返回的消息相同。这可以打消非特权用户
   猜测特权类是否存在的念头。
 * 做了一些用户不可见的额外系统目录变更。

libpgtcl 变更:
 * 向 "pg_result" tcl 命令添加了 -oid 选项。
   pg_result -oid 返回最后插入行的 oid。如果
   最后一条命令不是 INSERT，则 pg_result -oid 返回 ""。
 * 大对象接口以 pg_lo* tcl 命令提供:
   pg_lo_open、pg_lo_close、pg_lo_creat 等。

可移植性增强和新移植:
 * flex/lex 的问题已澄清。现在，你应该可以在任何平台上使用
   flex 代替 lex。我们不再根据你使用的平台假设
   你使用什么词法分析器。
 * 现在支持 Linux-ELF 移植。已测试各种配置：已知以下配置可用:
       kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
   全部为 ELF 格式，

新工具:
```

```
 * ipcclean 添加到发行版中
   通常不需要运行 ipcclean，但如果你的后端崩溃
   并留下悬空的共享内存段，ipcclean 会
   帮你清理。

新文档：
 * 用户手册已修订并添加了 libpq 文档。
```

E.91. Postgres95 Release 0.02

发行日期
1995-05-25

E.91.1. 变更

```
不兼容变更：
* 创建数据库的 SQL 语句是 'CREATE DATABASE' 而不是 'CREATEDB'。
  类似地，删除数据库是 'DROP DATABASE' 而不是 'DESTROYDB'。
  但可执行文件名 'createdb' 和 'destroydb' 保持不变。

新工具：
* pgperl - Postgres95 的 Perl (4.036) 接口
* pg_dump - 转储 postgres 数据库的工具
* 启动 Postgres95 服务器的 initdb 工具

高级特性：
* 打开数据库的异步 Postgres 信号量 I/O
* 支持基于主机的访问控制文件 postmaster/pg_hba.conf
```

E.92. Postgres95 Release 0.01

发行日期
1995-05-01

初始版本。

附录 F. 源代码仓库

PostgreSQL 源代码使用 Git 版本控制系统存储和管理。主仓库的一个公共镜像可用；它在主仓库的任何变更之后一分钟内更新。

我们的 wiki，http://wiki.postgresql.org/wiki/Working_with_Git，上有一些关于使用 Git 工作的讨论。

注意，从源代码仓库构建 PostgreSQL 需要相当新的版本的 bison、flex 和 Perl。从发行版 tarball 构建时不需要这些工具，因为它们用来构建的文件已包含在 tarball 中。其他工具要求与第 14 章中所示的相同。

F.1. 通过 Git 获取源代码

使用 Git 你将在本地机器上复制整个代码仓库，因此你可以离线访问所有历史和分支。这是开发或测试补丁的最快、最灵活的方式。

Git

1. 你需要一份已安装的 Git，可以从http://git-scm.com获得。许多系统默认已经安装了较新版本的 Git，或者可以在它们的软件包分发系统中获得。

2. 要开始使用 Git 仓库，克隆官方镜像：

   ```
   git clone git://git.postgresql.org/git/postgresql.git
   ```

   这会把完整的仓库复制到你的本地机器，所以可能需要一段时间才能完成，
   尤其是在网络连接较慢时。文件将被放在当前目录的一个新子目录 `postgresql` 中。

   如果例如防火墙阻挡了对 Git 协议的访问，也可以通过 HTTP 协议访问 Git 镜像。只需把 URL 前缀改为 `http`，如下：

   ```
   git clone http://git.postgresql.org/git/postgresql.git
   ```

   HTTP 协议不如 Git 协议高效，所以使用起来会更慢。

3. 每当你想获取系统中的最新更新时，`cd` 进入仓库，运行：

   ```
   git fetch
   ```

Git 能做的事情远不止获取源代码。要获取更多信息，请查阅 Git 的 man 手册页，或访问http://git-scm.com上的网站。

附录 G. 文档

PostgreSQL有四种主要的文档格式：

- 纯文本，用于安装前信息
- HTML，用于在线浏览和参考
- PDF 或 PostScript，用于打印
- 手册页，用于快速参考。

另外，若干纯文本`README`文件可以在 PostgreSQL源代码树各处找到，用于说明各种实现方面的问题。

HTML文档和手册页属于标准发行包的一部分，并会默认安装。PDF 和 PostScript 格式的文档可单独下载。

G.1. DocBook

文档源码使用DocBook编写，这是一种表面上类似于HTML的标记语言。
这两种语言都是标准通用标记语言（SGML）的应用，后者本质上是一种描述其他语言的语言。
下文会同时使用 DocBook 和 SGML 这两个术语，但从技术上讲，它们不能互换。

DocBook允许作者指定技术文档的结构和内容，而无需操心展示细节。
文档样式定义这些内容如何被渲染成若干最终形式之一。DocBook 由 OASIS[1] 组维护。官方 DocBook 站点[2]提供了很好的入门和参考文档，以及一本可供在线阅读的完整 O'Reilly 图书。FreeBSD Documentation Project[3]也使用 DocBook，并提供了一些很有价值的信息，其中包括若干值得参考的风格指南。

G.2. 工具集

以下工具用于处理文档。其中一些可能是可选的，具体见说明。

DocBook DTD[4]

这是 DocBook 自身的定义。我们当前使用 3.1 版；不能使用更高或更低的版本。注意还有 DocBook 的 XML 版本 —— 不要使用那个。

ISO 8879 character entities[5]

DocBook 需要这些实体，但由于它们由 ISO 维护，因此单独分发。

OpenJade[6]

这是处理 SGML 的基础软件包。它包含一个 SGML 解析器、一个 DSSSL 处理器（即使用 DSSSL 样式表将 SGML 转换为其他格式的程序），以及若干相关工具。Jade 现在由 Open Jade 小组维护，而不再由 James Clark 维护。

[1] http://www.oasis-open.org
[2] http://www.oasis-open.org/docbook
[3] http://www.freebsd.org/docproj/docproj.html
[4] http://www.oasis-open.org/docbook/sgml/
[5] http://www.oasis-open.org/cover/ISOEnts.zip
[6] http://openjade.sourceforge.net

DocBook DSSSL Stylesheets[7]

它们包含将 DocBook 源文件转换成其他格式（例如 HTML）的处理指令。

DocBook2X tools[8]

这个可选软件包用于创建手册页。它自身还有许多先决软件包。请查看该网站。

JadeTeX[9]

如果你愿意，还可以安装 JadeTeX，用 TeX 作为 Jade 的格式化后端。JadeTeX 可以创建 PostScript 或 PDF 文件（后者带书签）。

不过，JadeTeX 的输出质量不如 RTF 后端。问题特别多的地方是表格，以及各种垂直和水平间距的瑕疵。另外，也没有机会对结果进行手工润色。

我们已经记录了多种安装处理文档所需工具的方法，下面将加以介绍。
这些工具也可能还有其他打包发行形式。请将软件包状态报告到文档邮件列表，
我们会把这些信息补充到这里。

G.2.1. Linux RPM 安装

大多数发行版厂商都在其发行版中提供了用于 DocBook 处理的完整 RPM 集合。安装时查找“SGML”选项，或者查找下列软件包：`sgml-common`、`docbook`、`stylesheets`、`openjade`（或 `jade`）。可能还需要 `sgml-tools`。如果你的发行版厂商没有提供这些软件包，那么应该可以使用其他足够兼容的厂商提供的软件包。

G.2.2. FreeBSD 安装

FreeBSD Documentation Project 本身就大量使用 DocBook，因此 FreeBSD 提供了完整的文档工具“ports”也就不足为奇了。要在 FreeBSD 上构建文档，需要安装以下 ports。

- `textproc/sp`
- `textproc/openjade`
- `textproc/docbook-310`
- `textproc/iso8879`
- `textproc/dsssl-docbook-modular`

`/usr/ports/print` 中的一些内容（`tex`、`jadetex`）可能也值得安装。

有可能 ports 没有更新 `/usr/local/share/sgml/catalog` 中的主 catalog 文件。请确保其中包含下面这一行：

```
CATALOG "/usr/local/share/sgml/docbook/3.1/catalog"
```

如果不想编辑该文件，也可以把环境变量 `SGML_CATALOG_FILES` 设置为以冒号分隔的 catalog 文件列表（例如上面那些）。

[7] http://docbook.sourceforge.net/projects/dsssl/index.html
[8] http://docbook2x.sourceforge.net
[9] http://jadetex.sourceforge.net

更多关于 FreeBSD 文档工具的信息，请参阅FreeBSD Documentation Project 的说明[10]。

G.2.3. Debian 软件包

Debian GNU/Linux 也提供了完整的文档工具软件包集。只需使用以下命令即可安装：

```
apt-get install jade
apt-get install docbook
apt-get install docbook-stylesheets
```

G.2.4. 从源码手动安装

手动安装 DocBook 工具的过程有些复杂，因此如果有预构建的软件包，就使用它们。这里只描述一种标准配置，使用合理且标准的安装路径，不使用任何“花哨”功能。有关细节，应查阅各软件包的文档，并阅读SGML入门资料。

G.2.4.1. 安装 OpenJade

1. OpenJade 的安装采用 GNU 风格的`./configure; make; make install`构建过程。详细信息可在 OpenJade 源码发行包中找到。简要来说：

   ```
   ./configure --enable-default-catalog=/usr/local/share/sgml/catalog
   make
   make install
   ```

   务必记住“默认目录文件”的存放位置，后面会用到它。也可以不指定，但这样以后每次使用jade时，都必须设置环境变量`SGML_CATALOG_FILES`，使其指向该文件。（如果 OpenJade 已经安装，而你希望在本地安装工具链的其余部分，也可以采用这种方法。）

2. 此外，还应该把`dsssl`目录中的文件 `dsssl.dtd`、`fot.dtd`、`style-sheet.dtd`和`catalog` 安装到某个位置，例如 `/usr/local/share/sgml/dsssl`。最简单的做法可能是复制整个目录：

   ```
   cp -R dsssl /usr/local/share/sgml
   ```

3. 最后，创建文件 `/usr/local/share/sgml/catalog`，并向其中加入这一行：

   ```
   CATALOG "dsssl/catalog"
   ```

   （这是对安装到步骤 2中的文件的相对路径引用。如果你选择了不同的安装布局，务必作相应调整。）

G.2.4.2. 安装 DocBook DTD 工具包

1. 获取 DocBook V3.1[11] 发行包。

2. 创建目录`/usr/local/share/sgml/docbook31`并切换到该目录。
   （具体位置无关紧要，但这个位置在本文所采用的布局中是合理的。）

   ```
   $ mkdir /usr/local/share/sgml/docbook31$ cd /usr/local/share/sgml/docbook31
   ```

3. 解压归档包。

[10]http://www.freebsd.org/doc/en_US.ISO8859-1/books/fdp-primer/tools.html
[11]http://www.oasis-open.org/docbook/sgml/3.1/docbk31.zip

```
$ unzip -a ...../docbk31.zip
```

（归档包会把它包含的文件解压到当前目录。）

4. 编辑文件/usr/local/share/sgml/catalog（或者安装时告诉 jade 的其他位置），并加入类似下面的一行：

```
CATALOG "docbook31/docbook.cat"
```

5. 可以选择编辑文件 docbook.cat，注释掉或删除包含 DTDDECL 的那一行。如果不这样做，你会从 jade 收到警告，但没有进一步的危害。

6. 下载 ISO 8879 字符实体[12]归档包，将其解压，然后把文件放入存放 DocBook 文件的同一目录。

```
$ cd /usr/local/share/sgml/docbook31$ unzip ...../ISOEnts.zip
```

7. 在包含 DocBook 和 ISO 文件的目录中运行以下命令：

```
perl -pi -e 's/iso-(.*).gml/ISO\1/g' docbook.cat
```

（这会修正 DocBook 目录文件中使用的名称与 ISO 字符实体文件的实际名称之间的混淆。）

G.2.4.3. 安装 DocBook DSSSL 样式表

要安装样式表，先解压发行包，再把它移到合适的位置，例如 /usr/local/share/sgml。（归档会自动创建一个子目录。）

```
$gunzip docbook-dsssl-1.xx.tar.gz$tar -C /usr/local/share/sgml -xf
 docbook-dsssl-1.xx.tar
```

/usr/local/share/sgml/catalog 中通常的目录项也可以加上：

```
CATALOG "docbook-dsssl--1.xx/catalog
```

由于样式表变化相当频繁，而且有时尝试其他版本是有益的，PostgreSQL不使用这个目录项。关于如何改为选择样式表，参见第 G.2.5 节。

G.2.4.4. 安装 JadeTeX

要安装和使用JadeTeX，你需要可用的 TeX和LaTeX2e安装，包括受支持的tools和 graphics宏包、Babel、AMS 字体和 AMS-LaTeX、PSNFSS扩展及“35 种字体”配套包、用于生成 PostScript的dvips程序，以及宏包fancyhdr、hyperref、minitoc、url和ot2enc。所有这些都可以在邻近的CTAN[13]站点上找到。TeX基础系统的安装远远超出了本介绍的范围。
任何能运行TeX的系统都应该有可用的二进制软件包。

在将JadeTeX用于 PostgreSQL文档源之前，你需要增大 TeX内部数据结构的尺寸。相关细节可以在 JadeTeX的安装说明中找到。

完成之后，就可以安装JadeTeX：

```
$gunzip jadetex-xxx.tar.gz$tar xf jadetex-xxx.tar$cd jadetex$make
 install$mktexlsr
```

[12]http://www.oasis-open.org/cover/ISOEnts.zip
[13]http://www.ctan.org

最后两条需要以root身份执行。

G.2.5. 被 configure 检测

在构建文档之前，需要运行configure脚本，就像构建PostgreSQL程序本身时一样。
检查运行即将结束时的输出，它应该类似于：

```
checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V3.1... yes
checking for DocBook stylesheets... /usr/lib/sgml/stylesheets/nwalsh-
modular
checking for sgmlspl... sgmlspl
```

如果onsgmls和nsgmls都没有找到，你就看不到余下的 4 行。nsgmls是 Jade 软件包的一部分。如果没有找到“DocBook V3.1”，则说明 DocBook DTD 工具包没有安装到 Jade 能找到的位置，或者目录文件没有正确设置。请参阅上面的安装提示。DocBook 样式表会在若干个较为标准的位置中查找，但如果你把它们放在其他地方，则应该设置环境变量DOCBOOKSTYLE指向该位置，然后重新运行configure。

G.3. 构建文档

一切设置妥当后，切换到doc/src/sgml目录，
并运行后续各小节中介绍的某个命令来构建文档。（记得使用 GNU make。）

G.3.1. HTML

要构建文档的 HTML 版本：

```
doc/src/sgml$ gmake html
```

这也是默认目标。

构建 HTML 文档时，该过程还会为索引项生成链接信息。因此，如果你希望文档最后带有一个概念索引，就需要先构建一次 HTML 文档，然后再按你喜欢的任意格式重新构建一次文档。

为了便于在最终发行包中处理，构成 HTML 文档的各文件存储在一个 tar 归档中，并在安装时解包。要创建 HTML 文档包，使用命令

```
cd doc/src
gmake postgres.tar.gz
```

在发行包中，这些归档位于 doc 目录中，并会随 gmake install 默认安装。

G.3.2. 手册页

我们使用 docbook2man 工具将 DocBook refentry 页面转换为适合手册页的 *roff 输出。手册页也以 tar 归档的形式分发，与 HTML 版本类似。要创建手册页包，使用命令

```
cd doc/src
gmake man.tar.gz
```

这会在 doc/src 目录中生成一个 tar 文件。

要生成高质量的手册页，可能需要使用转换工具的修改版本，或者做一些手工后处理。所有手册页在分发之前都应人工检查。

G.3.3. 通过 JadeTex 生成打印输出

如果你想使用 JadeTex 生成文档的可打印版本，可以使用下列命令之一：

- 要制作 DVI 版本：

  ```
  doc/src/sgml$ gmake postgres.dvi
  ```

- 要从 DVI 生成 Postscript：

  ```
  doc/src/sgml$ gmake postgres.ps
  ```

- 要制作 PDF：

  ```
  doc/src/sgml$ gmake postgres.pdf
  ```

  （当然，你也可以用 Postscript 生成 PDF 版本，但如果直接生成 PDF，它将带有超链接和其他增强特性。）

G.3.4. 通过 RTF 生成打印输出

你也可以把PostgreSQL文档转换为 RTF并用办公套件做一些小的格式修正，从而生成可打印版本。视具体办公套件的能力而定，随后可以把文档转换为 PostScript 或 PDF。下面的过程以 Applixware 为例说明。

> **注意**
>
> 当前版本的PostgreSQL文档似乎会触发 OpenJade 的某个错误，或者超出其大小限制。如果RTF版本的构建过程长时间挂起且输出文件大小仍为 0，你可能是遇到了这个问题。（不过请记住，正常构建也需要 5 到 10 分钟，所以不要太早中止。）

Applixware RTF 清理

OpenJade没有为正文文本指定默认样式。过去，这个未确诊的问题导致目录生成过程极其漫长。不过，在 Applixware方面的大力帮助下，症状已得到诊断，并且有了可用的变通方法。

1. 输入下面的命令生成 RTF 版本：

   ```
   doc/src/sgml$ gmake postgres.rtf
   ```

2. 修复 RTF 文件，使其正确指定所有样式，特别是默认样式。如果文档包含 `refentry`小节，还必须替换把前一段落与当前段落绑定的格式提示，改为把当前段落与后一段落绑定。`doc/src/sgml` 中提供了一个实用程序`fixrtf`来完成这些修复：

   ```
   doc/src/sgml$ ./fixrtf --refentry postgres.rtf
   ```

   该脚本会添加`{\s0 Normal;}`作为文档的第 0 号样式。按照Applixware的说法，RTF 标准不允许添加隐式的第 0 号样式，不过 Microsoft Word 恰好能处理这种情况。对于修复`refentry`小节，脚本会把`\keepn` 标记替换为`\keep`。

3. 在Applixware Words中打开一个新文档，然后导入RTF文件。
4. 使用Applixware生成新目录（ToC）。
   a. 从第一行第一个字符开始到最后一行最后一个字符为止，选中现有的 ToC 行。
   b. 使用Tools → Book Building → Create Table of Contents构建新的 ToC。选择让 ToC 包含前三级标题。这会用Applixware原生的 ToC 替换从 RTF 导入的现有行。
   c. 使用Format → Style 调整 ToC 格式，依次选择三种 ToC 样式，并调整`First` 和`Left`的缩进。使用以下数值：

样式	首行缩进（英寸）	左缩进（英寸）
TOC-Heading 1	0.4	0.4
TOC-Heading 2	0.8	0.8
TOC-Heading 3	1.2	1.2

5. 在整个文档中完成以下工作：
   - 调整分页。
   - 调整表格列宽。
6. 用正确的值替换 ToC 中 Examples 和 Figures 部分右对齐的页码。这只需要几分钟。
7. (可选的) 如果索引节为空，从文档中删除它。
8. 重新生成并调整目录。
   a. 选中 ToC 域。
   b. 选择Tools → Book Building → Create Table of Contents。
   c. 通过选择Tools → Field Editing → Unprotect 解除 ToC 的保护。
   d. 删除 ToC 中的第一行，那是 ToC 自身的条目。
9. 将文档保存为Applixware Words原生格式，以便日后更容易地进行最后时刻的编辑。
10. 把文档 “打印” 到 PostScript 格式的文件。

G.3.5. 纯文本文件

若干文件以纯文本形式分发，供安装过程中阅读。`INSTALL` 文件对应于第 14 章，并有一些次要的改动以适应文本介质。如果因为某个原因需要重新生成该文件，切换到目录`doc/src/sgml` 并输入 **`gmake INSTALL`**。这会创建一个 `INSTALL.html` 文件，可以用 Netscape Navigator把它保存为文本，并放到现有文件的位置上。Netscape似乎为 HTML 到文本的转换提供了最好的质量（优于 lynx和 w3m）。

文件`HISTORY`可以类似地用命令 **`gmake HISTORY`**创建。对于文件 `src/test/regress/README`，命令是 **`gmake regress_README`**。

G.3.6. 语法检查

构建文档可能非常耗时。但有一种方法可以只检查文档文件的语法是否正确，这只需要几秒钟：

```
doc/src/sgml$ gmake check
```

G.4. 文档编写

SGML和DocBook的开源编写工具并不多。
最常用的工具组合是Emacs/XEmacs编辑器加上适当的编辑模式。在某些系统上，
典型的完整安装会包含这些工具。

G.4.1. Emacs/PSGML

PSGML是编辑SGML文档最常用、也最强大的模式。正确配置后，可以使用
Emacs插入标签并检查标记一致性。它也可以用于HTML。下载、安装说明和详细文档请参阅
PSGML 网站[14]。

使用PSGML时有一点需要注意：它的作者假定主SGML DTD目录是`/usr/local/lib/sgml`。如果像本章示例一样使用`/usr/local/share/sgml`，就必须作相应调整，可以设置`SGML_CATALOG_FILES`环境变量，也可以定制PSGML的安装配置（其手册介绍了具体方法）。

将以下内容放入 `~/.emacs` 环境文件中（将路径名调整为适合你系统的值）：

```
; ********** for SGML mode (psgml)

(setq sgml-omittag t)
(setq sgml-shorttag t)
(setq sgml-minimize-attributes nil)
(setq sgml-always-quote-attributes t)
(setq sgml-indent-step 1)
(setq sgml-indent-data t)
(setq sgml-parent-document nil)
(setq sgml-default-dtd-file "./reference.ced")
(setq sgml-exposed-tags nil)
(setq sgml-catalog-files '("/usr/local/share/sgml/catalog"))
(setq sgml-ecat-files nil)

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

并在同一文件中为 SGML 加入一项，添加到现有的变量定义中。该变量是 `auto-mode-alist`：

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
   ))
```

目前，每个 SGML 源文件的末尾都有下面这一块：

```
<!-- Keep this comment at the end of the file
Local variables:
mode: sgml
sgml-omittag:t
sgml-shorttag:t
sgml-minimize-attributes:nil
sgml-always-quote-attributes:t
```

[14] http://www.lysator.liu.se/projects/about_psgml.html

```
sgml-indent-step:1
sgml-indent-data:t
sgml-parent-document:nil
sgml-default-dtd-file:"./reference.ced"
sgml-exposed-tags:nil
sgml-local-catalogs:("/usr/lib/sgml/catalog")
sgml-local-ecat-files:nil
End:
-->
```

即使你不设置 `~/.emacs` 文件，这也会设置好一批编辑模式参数，但有点不幸的是，如果你按照上面的安装说明操作，catalog 路径将与你的位置不匹配。因此你可能需要关闭局部变量：

```
(setq inhibit-local-variables t)
```

PostgreSQL 发布中包含一个已解析的 DTD 定义文件 `reference.ced`。使用 PSGML 时，你可能会发现，处理这些分别保存书籍各部分的文件，有一种方便的方式：在编辑时插入适当的 `DOCTYPE` 声明。例如，当前这个源码文件是一章附录，因此可以将它指定为 DocBook 文档的 “appendix” 实例，把第一行写成这样：

```
<!DOCTYPE appendix PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
```

这样，所有读取 SGML 的工具都能正确处理该文档，并且可以用 `nsgmls -s docguide.sgml` 验证文档。（但在构建整套文档之前，需要移除这一行。）

G.4.2. 其他 Emacs 模式

GNU Emacs 自带另一种 SGML 模式，功能不如 PSGML 强大，但更容易理解，也更轻量。它还提供语法高亮（font lock），这可能很有帮助。

Norm Walsh 提供了一种专门用于 DocBook 的主模式[15]，也带有 font-lock 和若干减少输入量的功能。

G.5. 风格指南

G.5.1. 参考页

参考页应遵循标准布局。这能让用户更快找到所需信息，
也能促使作者记录命令的所有相关方面。不仅希望PostgreSQL参考页彼此保持一致，
也希望它们与操作系统及其他软件包提供的参考页保持一致。因此制定了下列准则。
它们在很大程度上与多个操作系统所建立的类似准则保持一致。

描述可执行命令的参考页应按以下顺序包含这些小节。不适用的小节可以省略。
只有在特殊情况下才应增加顶层小节；这些信息通常应放在 “用法” 小节中。

名称

本节自动生成。其中包含命令名以及一句简短的功能摘要。

概要

本节包含命令的语法图。概要通常不应列出每一个命令行选项；这些内容会在下文说明。
相反，它应列出命令行的主要组成部分，例如输入文件和输出文件的位置。

[15] http://nwalsh.com/emacs/docbookide/index.html

描述

用若干段解释该命令的作用。

选项

用列表描述每个命令行选项。如果选项很多，可以使用小节。

退出状态

如果程序以 0 表示成功、以非零表示失败，那么你不需要记录这一点。如果不同的非零退出代码有各自的含义，就在这里列出它们。

用法

描述程序的任何子语言或运行时接口。如果程序不是交互式的，通常可以省略本节。否则，本节就是用于描述运行时特性的总括性小节。适当时可以使用小节。

环境

列出程序可能使用的所有环境变量。尽量完整；即便像 `SHELL` 这样看似微不足道的变量，也可能让用户感兴趣。

文件

列出程序可能会隐式访问的任何文件。也就是说，
不要列出在命令行上指定的输入和输出文件，而应列出配置文件等内容。

诊断

解释程序可能产生的任何不寻常输出。不要去列出所有可能的错误消息。这项工作量很大，而在实践中用处不大。不过，例如如果错误消息有一种用户可以解析的标准格式，
那么这里就是解释它的地方。

注解

放不进其他地方的任何内容，尤其包括错误、实现缺陷、安全性考虑和兼容性问题。

示例

示例

历史

如果该程序的历史上有一些重要里程碑，可以在这里列出。通常本节可以省略。

另见

交叉引用按以下顺序列出：其他 PostgreSQL 命令参考页、PostgreSQL SQL 命令参考页、对 PostgreSQL 手册的引用、其他参考页（例如操作系统或其他软件包）、其他文档。同一组中的条目按字母顺序排列。

描述 SQL 命令的参考页应包含下列小节：名称、概要、描述、参数、输出、注解、示例、兼容性、历史、另见。参数小节类似于选项小节，但在决定列出命令的哪些子句时有更大的自由度。只有当命令返回的内容不是默认的命令完成标签时，才需要输出小节。
兼容性小节应解释该命令在多大程度上符合 SQL 标准，或者它与哪种其他数据库系统兼容。SQL 命令的另见小节应先列出 SQL 命令，再列出对程序的交叉引用。

参考书目

这里列出了一些关于 SQL 和 PostgreSQL 的参考文献与读物。

来自最初的 POSTGRES 开发团队的一些白皮书和技术报告，可在加州大学伯克利分校计算机科学系网站[1]上获取。

SQL 参考书

SQL 特性的参考文本。

[bowman93] The Practical SQL Handbook. Bowman et al, 1996. Using Structured Query Language. Third Edition. Judith Bowman、Sandra Emerson和Marcy Darnovsky. ISBN 0-201-44787-8. 1996. Addison-Wesley. 版权 © 1996 Addison-Wesley Longman, Inc..

[date97] A Guide to the SQL Standard. Date and Darwen, 1997. A user's guide to the standard database language SQL. Fourth Edition. C. J. Date和Hugh Darwen. ISBN 0-201-96426-0. 1997. Addison-Wesley. 版权 © 1997 Addison-Wesley Longman, Inc..

[date94] An Introduction to Database Systems. Date, 1994. Sixth Edition. C. J. Date. Volume 1. 1994. Addison-Wesley. 版权 © 1994 Addison-Wesley Longman, Inc..

[elma99] Fundamentals of Database Systems. 3rd Edition. Ramez Elmasri和Shamkant Navathe. ISBN 0-805-31755-4. August 1999. Addison-Wesley.

[melt93] Understanding the New SQL. Melton and Simon, 1993. A complete guide. Jim Melton和Alan R. Simon. ISBN 1-55860-245-3. 1993. Morgan Kaufmann. 版权 © 1993 Morgan Kaufmann Publishers, Inc..

[ull88] Principles of Database and Knowledge. Base Systems. Ullman, 1988. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.

PostgreSQL 专属文档

本节收录相关文档。

[sim98] Enhancement of the ANSI SQL Implementation of PostgreSQL. Simkovics, 1998. Stefan Simkovics.

讨论 SQL 的历史和语法，并描述向 PostgreSQL 中加入 `INTERSECT` 和 `EXCEPT` 构造的过程。在维也纳技术大学 O. Univ. Prof. Dr. Georg Gottlob 和 Univ. Ass. Mag. Katrin Seyr 的支持下作为硕士论文完成。

November 29, 1998. Department of Information Systems, Vienna University of Technology. Vienna, Austria.

[yu95] The Postgres95. User Manual. Yu and Chen, 1995. A. Yu和J. Chen. . Sept. 5, 1995. University of California. Berkeley, California.

[fong] The design and implementation of the POSTGRES query optimizer[2]. Zelaine Fong. University of California, Berkeley, Computer Science Department.

[1] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/
[2] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/UCB-MS-zfong.pdf

会议论文和期刊文章

本节收录文章和简报。

[olson93] Partial indexing in POSTGRES: research project. Olson, 1993. Nels Olson. 1993. UCB Engin T7.49.1993 O676. University of California. Berkeley, California.

[ong90] "A Unified Framework for Version Modeling Using Production Rules in a Database System". Ong and Goh, 1990. L. Ong和J. Goh. ERL Technical Memorandum M90/33. April, 1990. University of California. Berkely, California.

[rowe87] "The POSTGRES data model[3]". Rowe and Stonebraker, 1987. L. Rowe和M. Stonebraker. VLDB Conference, Sept. 1987.

[seshadri95] "Generalized Partial Indexes[4]". Seshardri, 1995. P. Seshadri和A. Swami. Eleventh International Conference on Data Engineering, 6-10 March 1995. 1995. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, California. 420-7.

[ston86] "The design of POSTGRES[5]". Stonebraker and Rowe, 1986. M. Stonebraker和L. Rowe. ACM-SIGMOD Conference on Management of Data, May 1986.

[ston87a] "The design of the POSTGRES. rules system". Stonebraker, Hanson, Hong, 1987. M. Stonebraker、E. Hanson和C. H. Hong. IEEE Conference on Data Engineering, Feb. 1987.

[ston87b] "The design of the POSTGRES storage system[6]". Stonebraker, 1987. M. Stonebraker. VLDB Conference, Sept. 1987.

[ston89] "A commentary on the POSTGRES rules system[7]". Stonebraker et al, 1989. M. Stonebraker、M. Hearst和S. Potamianos. SIGMOD Record 18(3). Sept. 1989.

[ston89b] "The case for partial indexes[8]". Stonebraker, M, 1989b. M. Stonebraker. SIGMOD Record 18(4). 4-11. Dec. 1989.

[ston90a] "The implementation of POSTGRES[9]". Stonebraker, Rowe, Hirohama, 1990. M. Stonebraker、L. A. Rowe和M. Hirohama. Transactions on Knowledge and Data Engineering 2(1). IEEE. March 1990.

[ston90b] "On Rules, Procedures, Caching and Views in Database Systems[10]". Stonebraker et al, ACM, 1990. M. Stonebraker、A. Jhingran、J. Goh和S. Potamianos. ACM-SIGMOD Conference on Management of Data, June 1990.

[3] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-13.pdf
[4] http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z
[5] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M85-95.pdf
[6] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-06.pdf
[7] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-82.pdf
[8] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf
[9] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-34.pdf
[10] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-36.pdf

索引

符号

$，30
$libdir，453
*，66
.pgpass，318
事务隔离级别，176，
信号量，230
共享内存，230
共享库，461
关闭，235
别名
  查询中的表名，11
动态加载，
区域设置，211
历史
  PostgreSQL 的，xxiii
口令
  超级用户的，211
只读事务，
圆括号，30
大对象
  在 JDBC 中，386
  在 pgtcl 中，338
大小写敏感性
  SQL 命令，24
字符串
  串接，104，113
  反斜杠引号，
字符集，
客户端认证
  期间的超时，
密码，239
引号
  与标识符，25
引导
  期间启动服务器，211
操作符类，171，482
数据区域 (见数据库集簇)
数据库
  创建权限，239
数据库集簇，210
日期样式，
时区，
  澳大利亚，
有效数字，
服务器日志，221
标识符
  长度，24
模式
  搜索路径，
正则表达式，
死锁
  期间的超时，
浮点
  显示，
端口，
类路径，380
索引
  基于表达式，170
  唯一，170
  检查使用情况，174
  部分，172
索引扫描，
线程
  与 JDBC，409
继承，
规则
  与触发器比较，507
触发器
  与规则比较，507
超时
  客户端认证，
  死锁，
超级用户，238
连接，10
  外连接，11
  自连接，11
连接池
  在 JDBC 中，409
通知处理器，316
通知接收器，316
遗传查询优化，
配置
  服务器的，213
重载
  操作符，477
顺序扫描，
预备查询
  在 PL/pgSQL 中，519

A

ABORT，604
aggregate function，12
  built-in，153
  invocation，31
  user-defined，472
alias
  in the FROM clause，61
  in the select list，66
ALL，155
all，157
ALTER AGGREGATE，605
ALTER CONVERSION，606

ALTER DATABASE，607
ALTER DOMAIN，609
ALTER FUNCTION，611
ALTER GROUP，612
ALTER LANGUAGE，613
ALTER OPERATOR CLASS，614
ALTER SCHEMA，615
ALTER SEQUENCE，616
ALTER TABLE，618
ALTER TRIGGER，623
ALTER USER，239，624
ANALYZE，263，627
AND (operator)，98
any，96，157
ANY，155
anyarray，96
anyelement，96
array，88
  constant，89
  constructor，33
  of user-defined type，476
ARRAY
  determination of result type，166
auto-increment (见 serial)
autocommit，189
average，12，154

B

B-tree (见 index)
backslash_quote configuration parameter，
backup，266
base type，444
BEGIN，629
between，99
BETWEEN，99
bigint，26，71
bigserial，73
binary data，76
  functions，113
binary string
  length，115
bison，197
bit string
  constant，26
  data type，88
BLOB (见 large object)
Boolean
  data type，83
  operators (见 operators, logical)
box (data type)，85
BSD/OS
  IPC 配置，
  shared library，
bytea，76
  在 JDBC 中，386
  in libpq，306

C

C，289，363
canceling
  SQL command，309
CASCADE
  with DROP，53
  foreign key action，43
CASE，145
  determination of result type，166
char，74
character，74
character set，257
character string
  constant，25
  data types，74
  length，105，107
character varying，74
check constraint，39
checkpoint，279
CHECKPOINT，630
cid，95
cidr，86
circle，86
CLASSPATH，380
client authentication，246
CLOSE，631
cluster
  of databases (见 database cluster)
CLUSTER，632
clusterdb，783
cmax，38
cmin，37
COALESCE，147
column，7，36
  adding，46
  removing，47
  renaming，48
  system column，37
column reference，30
col_description，147
comment
  about database objects，147
  in SQL，28
COMMENT，634
COMMIT，636
comparison
  of rows，157
  operators，98

compiling
  libpq applications，319
composite type，444
concurrency，176
conditional expression，145
configuration
  of the server
    functions，147
configure，198
conjunction，98
constant，25
constraint，38
  adding，47
  check，39
  foreign key，42
  name，39
  NOT NULL，40
  primary key，41
  removing，47
  unique，41
COPY，8，637
  with libpq，312
count，12
CREATE AGGREGATE，643
CREATE CAST，645
CREATE CONSTRAINT，648
CREATE CONVERSION，649
CREATE DATABASE，241，651
CREATE DOMAIN，653
CREATE FUNCTION，655
CREATE GROUP，659
CREATE INDEX，660
CREATE LANGUAGE，662
CREATE OPERATOR，664
CREATE OPERATOR CLASS，667
CREATE RULE，670
CREATE SCHEMA，672
CREATE SEQUENCE，674
CREATE TABLE，7，677
CREATE TABLE AS，685
CREATE TRIGGER，686
CREATE TYPE，688
CREATE USER，238，693
CREATE VIEW，695
createdb，3，242，785
createlang，788
createuser，238，790
cross join，
crypt，251
  thread safety，319
cstring，96
ctid，38
CTID，495
currval，144
cursor
  in PL/pgSQL，536

D

data type，70
  base，444
  category，161
  composite，444
  constant，26
  conversion，160
  internal organisation，453
  numeric，71
  type cast，32
  user-defined，474
database，241
  creating，3
database activity
  monitoring，271
database cluster，7
DataSource，409
date，77，79
  constants，81
  current，138
  output format，81
    (参见 formatting)
DBI，559
deadlock，180
DEALLOCATE，697
decimal (见 numeric)
DECLARE，698
default value，38
  changing，47
DELETE，13，55，701
deleting，55
Digital UNIX (见 Tru64 UNIX)
dirty read，176
disjunction，98
disk drive，280
disk space，262
disk usage，276
DISTINCT，9，67
double precision，73
DROP AGGREGATE，703
DROP CAST，704
DROP CONVERSION，705
DROP DATABASE，244，706
DROP DOMAIN，707
DROP FUNCTION，708
DROP GROUP，709
DROP INDEX，710
DROP LANGUAGE，711
DROP OPERATOR，712
DROP OPERATOR CLASS，714

DROP RULE，715
DROP SCHEMA，716
DROP SEQUENCE，717
DROP TABLE，8，718
DROP TRIGGER，719
DROP TYPE，720
DROP USER，238，721
DROP VIEW，722
dropdb，244，793
droplang，795
dropuser，238，797
duplicate，9
duplicates，67
dynamic loading，452
dynamic_library_path，，453

E

ECPG，363
ecpg，799
elog，928
  in PL/Perl，
  in PL/Python，563
  in PL/Tcl，
embedded SQL
  in C，363
END，723
environment variable，317
ereport，928
error codes
  list of，963
error message，296
escaping strings，305
EXCEPT，67
EXECUTE，724
EXISTS，155
EXPLAIN，183，725
expression
  order of evaluation，34
  syntax，29
extending SQL，444

F

false，83
FAQ，xxiv
fast path，310
FETCH，728
field selection，31
flex，197
float4 (见 real)
float8 (见 double precision)
floating point，73
foreign key，15，42
formatting，125
FreeBSD
  IPC 配置，
  shared library，
  启动脚本，211
FROM
  缺失，
fsync，，278
function，98
  in the FROM clause，62
  internal，452
  invocation，31
  polymorphic，445
  type resolution in an invocation，164
  user-defined，445
    in C，446
    in SQL，446

G

GEQO (见遗传查询优化)
get_bit，114
get_byte，114
global data
  in PL/Python，562
  in PL/Tcl，552
GRANT，239，732
group，239
GROUP BY，12，64
grouping，64

H

hash (见 index)
has_database_privilege，147
has_function_privilege，147
has_language_privilege，147
has_schema_privilege，147
has_table_privilege，147
HAVING，12，65
hierarchical database，7
host name，289
HP-UX
  IPC 配置，
  shared library，

I

ident，252
identifier
  syntax of，24
IN，155
in，157
index，168
  B-tree，168
  for user-defined data type，481
  hash，169

locks，181
multicolumn，169
R-tree，169
inet (data type)，86
information schema，414
inheritance，17
initdb，210，846
initlocation，244，848
input function，474
of a data type，474
INSERT，8，54，736
inserting，54
installation，195
on Windows，195，209
instr，544
int2 (见 smallint)
int4 (见 integer)
int8 (见 bigint)
integer，26，71
internal，96
INTERSECT，67
interval，77，80
ipcclean，849
IRIX
shared library，
IS NULL，

J

Java，380
JDBC，380
JNDI，412
join，58
controlling the order，188
cross，
left，59
natural，58
outer，
right，59

K

Kerberos，252
key word
list of，976
syntax of，24

L

label (见 alias)
language_handler，96
large object，330
backup，268
ldconfig，204
left join，59
length
of a binary string (见 binary strings, length)
of a character string (见 character strings, length)
libperl，195
libpgtcl，338
libpq，289
libpq-fe.h，289，294
libpq-int.h，294，320
libpython，196
LIKE，116
LIMIT，69
line segment，85
Linux
IPC 配置，
shared library，
启动脚本，212
LISTEN，738
LOAD，739
locale，255
lock，178
monitoring，275
LOCK，178，740
loop
in PL/pgSQL，533
lo_close，332
lo_creat，330
lo_export，331，332
lo_import，331，332
lo_lseek，332
lo_open，331
lo_read，331
lo_tell，332
lo_unlink，332
lo_write，331
lseg，85

M

MAC address (见 macaddr)
macaddr (data type)，87
MacOS X
IPC 配置，
shared library，
maintenance，262
make，195
MANPATH，204
max，12
MD5，251
memory context
in SPI，584
min，12
monitoring
database activity，271
MOVE，742

MVCC，176

N

name
  qualified，49
  syntax of，24
  unqualified，50
natural join，58
negation，98
NetBSD
  IPC 配置，
  shared library，
  启动脚本，212
network
  data types，86
nextval，144
nonblocking connection，291，307
nonrepeatable read，176
NOT (operator)，98
NOT IN，155
not in，157
not-null constraint，40
notice processing
  in libpq，316
NOTIFY，743
  in libpq，311
  在 pgtcl 中，350
null value
  with check constraints，40
  comparing，99
  default value，38
  in DISTINCT，67
  in libpq，
  in PL/Perl，558
  in PL/Python，562
  with unique constraints，41
nullif，147
number
  constant，26
numeric，26
numeric (data type)，72

O

object identifier
  data type，95
object-oriented database，7
obj_description，147
OFFSET，69
OID
  in libpq，305
  列，37
oid，95
ONLY，58
opaque，96
OpenBSD
  IPC 配置，
  shared library，
  启动脚本，212
OpenSSL，
  (参见 SSL)
operator，98
  invocation，31
  logical，98
  precedence，28
  syntax，27
  type resolution in an invocation，161
  user-defined，477
OR (operator)，98
Oracle
  porting from PL/SQL to PL/pgSQL，541
ORDER BY，9，68
  与区域，256
ordering operator，486
outer join，
output function，474
  of a data type，474
overlay，105
overloading
  functions，472
owner，239

P

palloc，460
PAM，201，254
parameter
  syntax，30
password
  authentication，251
password file，318
PATH，204
path (data type)，85
pattern matching，115
Perl，558
permission (见 privilege)
pfree，460
PGCLIENTENCODING，318
PGconn，289
PGCONNECT_TIMEOUT，318
PGDATA，210
PGDATABASE，317
PGDATESTYLE，318
PGGEQO，318
PGHOST，317
PGHOSTADDR，317
PGOPTIONS，318
PGPASSWORD，318

PGPORT，317
PGREALM，318
PGREQUIRESSL，318
PGresult，298
PGSERVICE，318
PGSSLMODE，318
pgtcl，338
pgtclsh，818
pgtksh，819
PGTZ，318
PGUSER，317
pg_aggregate，873
pg_am，873
pg_amop，874
pg_amproc，874
pg_attrdef，875
pg_attribute，875
pg_cast，877
pg_class，877
pg_config，801
  with libpq，319
  用于用户定义的 C 函数，460
pg_conndefaults，342
pg_connect，340
pg_constraint，879
pg_controldata，850
pg_conversion，880
pg_conversion_is_visible，147
pg_ctl，211，851
pg_database，243，880
pg_depend，881
pg_description，882
pg_disconnect，341
pg_dump，803
pg_dumpall，809
  use during upgrade，197
pg_exec，343
pg_execute，348
pg_function_is_visible，147
pg_get_constraintdef，147
pg_get_expr，147
pg_get_indexdef，147
pg_get_ruledef，147
pg_get_triggerdef，147
pg_get_userbyid，147
pg_get_viewdef，147
pg_group，883
pg_hba.conf，246
pg_ident.conf，253
pg_index，883
pg_indexes，896
pg_inherits，884
pg_language，884
pg_largeobject，885
pg_listen，350
pg_listener，885
pg_locks，896
pg_lo_close，354
pg_lo_creat，352
pg_lo_export，361
pg_lo_import，360
pg_lo_lseek，357
pg_lo_open，353
pg_lo_read，355
pg_lo_tell，358
pg_lo_unlink，359
pg_lo_write，356
pg_namespace，886
pg_on_connection_loss，351
pg_opclass，886
pg_opclass_is_visible，147
pg_operator，887
pg_operator_is_visible，147
pg_proc，888
pg_restore，812
pg_result，344
pg_rewrite，889
pg_rules，897
pg_select，346
pg_settings，897
pg_shadow，889
pg_statistic，187，890
pg_stats，187，898
pg_tables，899
pg_table_is_visible，147
pg_trigger，891
pg_type，892
pg_type_is_visible，147
pg_user，900
pg_views，900
phantom read，176
PIC，461，461
PID
  determining PID of server process
    in libpq，296
PL/Perl，558
PL/PerlU，560
PL/pgSQL，519
PL/Python，562
PL/SQL (Oracle)
  porting to PL/pgSQL，541
PL/Tcl，551
point，84
polygon，85
polymorphic function，445
polymorphic type，445
port，290
POSTGRES，xxiii

postgres 用户，210
Postgres95，xxiii
postgresql.conf，213
postgres（程序），856
postmaster，3，211，860
PQbackendPID，296
PQbinaryTuples，303
  with COPY，312
PQclear，301
PQcmdStatus，304
PQcmdTuples，304
PQconndefaults，293
PQconnectdb，289
PQconnectPoll，291
PQconnectStart，291
PQconsumeInput，308
PQdb，294
PQendcopy，315
PQerrorMessage，296
PQescapeBytea，306
PQescapeByteaConn，306
PQescapeString，305
PQescapeStringConn，305
PQexec，297
PQexecParams，297
PQexecPrepared，298
PQfformat，302
  with COPY，312
PQfinish，293
PQflush，310
PQfmod，302
PQfn，310
PQfname，301
PQfnumber，301
PQfreemem，307
PQfsize，303
PQftable，302
PQftablecol，302
PQftype，302
PQgetCopyData，313
PQgetisnull，303
PQgetlength，303
PQgetline，314
PQgetlineAsync，314
PQgetResult，308
PQgetssl，296
PQgetvalue，303
PQhost，295
PQisBusy，309
PQisnonblocking，310
PQmakeEmptyPGresult，301
PQnfields，301
  用于 COPY，312
PQnotifies，311
PQntuples，301
PQoidStatus，305
PQoidValue，305
PQoptions，295
PQparameterStatus，296
PQpass，294
PQport，295
PQprint，304
PQprotocolVersion，296
PQputCopyData，313
PQputCopyEnd，313
PQputline，315
PQputnbytes，315
PQrequestCancel，309
PQreset，293
PQresetPoll，294
PQresetStart，294
PQresStatus，299
PQresultErrorField，299
PQresultErrorMessage，299
PQresultStatus，298
PQsendQuery，307
PQsendQueryParams，308
PQsendQueryPrepared，308
PQsetdb，291
PQsetdbLogin，291
PQsetErrorVerbosity，316
PQsetnonblocking，309
PQsetNoticeProcessor，316
PQsetNoticeReceiver，316
PQsocket，296
PQstatus，295
PQtrace，316
PQtransactionStatus，295
PQtty，295
PQunescapeBytea，307
PQuntrace，316
PQuser，294
preload_libraries，
PREPARE，745
PreparedStatement，382
preparing a query
  in PL/Python，563
  in PL/Tcl，553
primary key，41
privilege，48，239
  querying，149
  with rules，505
  for schemas，51
  with views，505
procedural language，517
  handler for，940
ps
  to monitor activity，271

psql，5，820
Python，562

Q

qualified name，49
query，9，57
query plan，183
query tree，488
quotation marks
  escaping，25
quote_ident，108
  use in PL/pgSQL，529
quote_literal，108
  use in PL/pgSQL，529

R

R-tree (见 index)
range table，
readline，195
real，73
record，96
rectangle，85
referential integrity，15，42
regclass，95
regoper，95
regoperator，95
regproc，95
regprocedure，95
regression test，202
regression tests，281
regtype，95
regular expression，116，117
  (参见 pattern matching)
reindex，265
REINDEX，747
relation，7
relational database，7
RESET，749
RESTRICT
  with DROP，53
  foreign key action，43
ResultSet，382
REVOKE，239，750
right join，59
ROLLBACK，752
row，7，36
rule，488
  and views，489
  for DELETE，496
  for INSERT，496
  for SELECT，490
  for UPDATE，496

S

savepoint，278
scalar (见 expression)
schema，49，241
  creating，49
  current，50，147
  public，50
  removing，50
SCO OpenServer
  IPC 配置，
search path，50
  current，147
search_path，50，
SELECT，9，57，753
  select list，66
SELECT INTO，763
  in PL/pgSQL，527
sequence，144
  and serial type，73
serial，73
serial4，73
serial8，73
server log
  log file maintenance，265
SET，147，764
SET CONSTRAINTS，767
set difference，67
set intersection，67
set operation，67
SET SESSION AUTHORIZATION，768
SET TRANSACTION，770
set union，67
SETOF，446，446
  (参见 function)
setval，144
set_bit，114
set_byte，114
shared library，203
SHMMAX，231
SHOW，147，772
SIGHUP，214，249，253
SIGINT，235
SIGQUIT，235
SIGTERM，235
SIMILAR TO，116
sliced bread (见 TOAST)
smallint，71
Solaris
  IPC 配置，
  shared library，
  启动脚本，212
SOME，155
some，157

sorting，68
SPI，565
SPI_connect，566
SPI_copytuple，588
SPI_copytupledesc，589
SPI_copytupleintoslot，590
SPI_cursor_close，576
SPI_cursor_fetch，574
SPI_cursor_find，573
SPI_cursor_move，575
SPI_cursor_open，572
SPI_exec，568
SPI_execp，571
SPI_finish，567
SPI_fname，578
SPI_fnumber，579
SPI_freeplan，594
SPI_freetuple，592
SPI_freetuptable，593
SPI_getbinval，581
SPI_getrelname，584
SPI_gettype，582
SPI_gettypeid，583
SPI_getvalue，580
spi_lastoid，
SPI_modifytuple，591
SPI_palloc，585
SPI_pfree，587
SPI_prepare，570
SPI_repalloc，586
SPI_saveplan，577
ssh，236
SSL，，236
  in libpq，296
  with libpq，290
ssl_renegotiation_limit configuration parameter，
standard deviation，154
START TRANSACTION，774
Statement，382
statistics，272
  of the planner，186，263
string (见 character string)
subquery，12，33，62，155
subscript，30
substring，106，106，106，114，116
sum，12
superuser，5
syntax
  SQL，24
syslog，221
system catalog
  schema，52

T

table，7，36
  creating，36
  modifying，46
  removing，37
  renaming，48
table expression，57
table function，62
tableoid，37
target list，
Tcl，338，551
TCP/IP，211，214
template0，242
template1，241，242
test，281
text，74
threads
  with libpq，318
tid，95
time，77，79
  constants，81
  current，138
  output format，81
    (参见 formatting)
time span，77
time with time zone，77，79
time without time zone，77，79
time zone，82
  abbreviations，971
  conversion，137
timestamp，77，80
timestamp with time zone，77，80
timestamp without time zone，77，80
TOAST，330
  and user-defined types，477
token，24
to_char，125
transaction，16
transaction ID
  wraparound，263
transaction isolation，176
transaction isolation level
  read committed，177
  serializable，177
transaction log (见 WAL)
trigger，96，510
  arguments for trigger functions，510
  in C，511
  in PL/Perl，561
  in PL/pgSQL，539
  in PL/Python，563
  in PL/Tcl，555
Tru64 UNIX

shared library，
true，83
TRUNCATE，775
trusted
PL/Perl，560
type (见 data type)
polymorphic，445
type cast，26，32

U

UNION，67
determination of result type，166
unique constraint，41
Unix domain socket，289
UnixWare
IPC 配置，
shared library，
UNLISTEN，776
unqualified name，50
UPDATE，13，55，777
updating，55
upgrading，197，269
user，238
current，147

V

vacuum，262
VACUUM，779
vacuumdb，842
value expression，29
varchar，74
variance，154
version，5，147
compatibility，269
view，15
implementation through rules，489
updating，500
void，96

W

WAL，278
WHERE，63

X

xid，95
xmax，37
xmin，37

Y

yacc，197