PostgreSQL 7.2.8 手册

PostgreSQL 全球开发组
翻译：冯若航[1]，Pigsty 项目组[2]

法律声明

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

目录

PostgreSQL 7.2.8 教程 x
欢迎 xiii
1. 从头开始 1
1.1. 安装 1
1.2. 架构基础 1
1.3. 创建数据库 1
1.4. 访问数据库 3
2. SQL语言 5
2.1. 简介 5
2.2. 概念 5
2.3. 创建新表 5
2.4. 向表中填充行 6
2.5. 查询表 7
2.6. 表之间的连接 8
2.7. 聚合函数 10
2.8. 更新 11
2.9. 删除 11
3. 高级特性 13
3.1. 简介 13
3.2. 视图 13
3.3. 外键 13
3.4. 事务 14
3.5. 继承 15
3.6. 小结 16
PostgreSQL 7.2.8 用户指南 xvii
前言 xxv
1. 什么是PostgreSQL? xxv
2. PostgreSQL简史 xxv
3. 文档资源 xxvii
4. 术语和记法 xxviii
5. 缺陷报告指南 xxviii
6. Y2K 声明 xxxi
1. SQL 语法 1
1.1. 词法结构 1
1.2. 列 5
1.3. 值表达式 6
1.4. 词法优先级 9
2. 查询 11
2.1. 概述 11
2.2. 表表达式 11
2.3. 选择列表 16
2.4. 组合查询 18
2.5. 行排序 18
2.6. LIMIT 和 OFFSET 19
3. 数据类型 20
3.1. 数字类型 21
3.2. 货币类型 24
3.3. 字符类型 24
3.4. 二进制串 26
3.5. 日期/时间类型 27
3.6. 布尔类型 33

3.7. 几何类型 34
3.8. 网络地址数据类型 37
3.9. 位串类型 38
4. 函数和操作符 40
4.1. 逻辑操作符 40
4.2. 比较操作符 40
4.3. 数学函数和操作符 42
4.4. 字符串函数和操作符 46
4.5. 二进制串函数和操作符 50
4.6. 模式匹配 52
4.7. 数据类型格式化函数 55
4.8. 日期/时间函数和操作符 59
4.9. 几何函数和操作符 67
4.10. 网络地址类型函数 70
4.11. 序列操作函数 72
4.12. 条件表达式 73
4.13. 杂项函数 75
4.14. 聚合函数 76
4.15. 子查询表达式 77
5. 类型转换 82
5.1. 简介 82
5.2. 概述 82
5.3. 操作符 83
5.4. 函数 86
5.5. 查询目标 88
5.6. UNION 和 CASE 结构 89
6. 数组 91
7. 索引 95
7.1. 简介 95
7.2. 索引类型 95
7.3. 多列索引 96
7.4. 唯一索引 97
7.5. 函数索引 97
7.6. 操作符类 98
7.7. 键 98
7.8. 部分索引 99
7.9. 检查索引使用情况 101
8. 继承 103
9. 多版本并发控制 106
9.1. 介绍 106
9.2. 事务隔离 106
9.3. 读已提交隔离级别 106
9.4. 可串行化隔离级别 107
9.5. 应用级别的数据一致性检查 107
9.6. 锁与表 108
9.7. 锁与索引 109
10. 管理数据库 111
10.1. 数据库创建 111
10.2. 访问数据库 111
10.3. 删除数据库 112
11. 性能提示 113
11.1. 使用EXPLAIN 113
11.2. 规划器使用的统计信息 116

11.3. 用显式 JOIN 控制规划器 118
11.4. 填充一个数据库 119
A. 日期/时间支持 121
A.1. 日期/时间关键字 121
A.2. 时区 122
A.3. 历法的历史 126
B. SQL 关键字 128
参考书目 143
PostgreSQL 7.2.8 管理员指南 cxlv
1. 安装说明 1
1.1. 简短版本 1
1.2. 需求 1
1.3. 获取源代码 2
1.4. 如果你是在升级 2
1.5. 安装过程 3
1.6. 安装后设置 9
1.7. 受支持的平台 10
2. 在 Windows 上安装 15
3. 服务器运行时环境 16
3.1. PostgreSQL 用户账户 16
3.2. 创建数据库集簇 16
3.3. 启动数据库服务器 17
3.4. 运行时配置 19
3.5. 管理内核资源 29
3.6. 关闭服务器 33
3.7. 使用 SSL 的安全 TCP/IP 连接 34
3.8. 使用 SSH 隧道的安全 TCP/IP 连接 35
4. 客户端认证 36
4.1. pg_hba.conf 文件 36
4.2. 认证方法 40
4.3. 认证问题 43
5. 本地化 44
5.1. 区域支持 44
5.2. 多字节支持 46
5.3. 单字节字符集重编码 52
6. 管理数据库 54
6.1. 创建数据库 54
6.2. 删除数据库 56
7. 数据库用户和权限 57
7.1. 数据库用户 57
7.2. 组 57
7.3. 权限 58
7.4. 函数和触发器 58
8. 日常数据库维护任务 59
8.1. 一般性讨论 59
8.2. 例行清理 59
8.3. 日志文件维护 62
9. 备份与恢复 63
9.1. SQL 转储 63
9.2. 文件系统级别的备份 65
9.3. 版本间的迁移 66
10. 监控数据库活动 67
10.1. 标准 Unix 工具 67

10.2. 统计收集器 67
11. 预写式日志（WAL） 71
11.1. 总体描述 71
11.2. 实现 72
11.3. WAL 配置 72
12. 磁盘存储 74
13. 数据库故障 75
13.1. 磁盘已满 75
13.2. 磁盘故障 75
14. 回归测试 76
14.1. 简介 76
14.2. 运行测试 76
14.3. 测试结果评估 77
14.4. 平台相关的比较文件 79
A. 发布说明 80
A.1. 版本 7.2.8 80
A.2. 版本 7.2.7 80
A.3. 版本 7.2.6 81
A.4. 版本 7.2.5 82
A.5. 版本 7.2.4 82
A.6. 版本 7.2.3 83
A.7. 版本 7.2.2 83
A.8. 版本 7.2.1 84
A.9. 版本 7.2 85
A.10. 版本 7.1.3 96
A.11. 版本 7.1.2 96
A.12. 版本 7.1.1 97
A.13. 版本 7.1 97
A.14. 版本 7.0.3 101
A.15. 版本 7.0.2 102
A.16. 版本 7.0.1 103
A.17. 版本 7.0 104
A.18. 版本 6.5.3 111
A.19. 版本 6.5.2 111
A.20. 版本 6.5.1 112
A.21. 版本 6.5 113
A.22. 版本 6.4.2 117
A.23. 版本 6.4.1 118
A.24. 版本 6.4 119
A.25. 版本 6.3.2 123
A.26. 版本 6.3.1 124
A.27. 版本 6.3 125
A.28. 版本 6.2.1 129
A.29. 版本 6.2 130
A.30. 版本 6.1.1 132
A.31. 版本 6.1 133
A.32. 版本 6.0 136
A.33. 版本 1.09 138
A.34. 版本 1.02 138
A.35. 版本 1.01 140
A.36. 版本 1.0 142
A.37. Postgres95 版本 0.03 143
A.38. Postgres95 版本 0.02 145
A.39. Postgres95 版本 0.01 146

PostgreSQL 7.2.8 程序员指南 cxlvii
I. 客户端接口 1
1. libpq - C 库 5
2. 大对象 33
3. libpq++ - C++ 绑定库 41
4. pgtcl - Tcl 绑定库 48
5. libpgeasy - 简化 C 库 69
6. ecpg - C 中的嵌入式 SQL 70
7. ODBC 接口 79
8. JDBC 接口 86
9. PyGreSQL - Python 接口 115
II. 服务器编程 168
10. 体系结构 171
11. 扩展 SQL：概览 174
12. 扩展 SQL：函数 177
13. 扩展 SQL：类型 197
14. 扩展 SQL：操作符 199
15. 扩展 SQL：聚合 204
16. 规则系统 206
17. 索引扩展接口 228
18. 索引代价估计函数 234
19. GiST 索引 237
20. 触发器 239
21. 服务器编程接口 246
III. 过程语言 283
22. 过程语言 286
23. PL/pgSQL - SQL 过程语言 288
24. PL/Tcl - Tcl 过程语言 320
25. PL/Perl - Perl 过程语言 326
26. PL/Python - Python 过程语言 330
PostgreSQL 7.2.8 参考手册 cccxxxiii
前言 cccxxxvii
I. SQL 命令 338
ABORT 340
ALTER GROUP 341
ALTER TABLE 342
ALTER USER 346
ANALYZE 348
BEGIN 350
CHECKPOINT 352
CLOSE 353
CLUSTER 354
COMMENT 356
COMMIT 358
COPY 359
CREATE AGGREGATE 365
CREATE CONSTRAINT TRIGGER 367
CREATE DATABASE 368
CREATE FUNCTION 371
CREATE GROUP 375
CREATE INDEX 377
CREATE LANGUAGE 380
CREATE OPERATOR 382
CREATE RULE 386

CREATE SEQUENCE 389
CREATE TABLE 392
CREATE TABLE AS 400
CREATE TRIGGER 402
CREATE TYPE 404
CREATE USER 408
CREATE VIEW 410
DECLARE 412
DELETE 415
DROP AGGREGATE 417
DROP DATABASE 418
DROP FUNCTION 419
DROP GROUP 421
DROP INDEX 422
DROP LANGUAGE 423
DROP OPERATOR 424
DROP RULE 426
DROP SEQUENCE 427
DROP TABLE 428
DROP TRIGGER 430
DROP TYPE 431
DROP USER 433
DROP VIEW 434
END 436
EXPLAIN 437
FETCH 440
GRANT 443
INSERT 446
LISTEN 448
LOAD 450
LOCK 451
MOVE 456
NOTIFY 457
REINDEX 459
RESET 461
REVOKE 462
ROLLBACK 464
SELECT 465
SELECT INTO 475
SET 477
SET CONSTRAINTS 481
SET SESSION AUTHORIZATION 482
SET TRANSACTION 483
SHOW 485
TRUNCATE 487
UNLISTEN 488
UPDATE 490
VACUUM 492
II. PostgreSQL 客户端应用 495
createdb 496
createlang 498
createuser 500
dropdb 502
droplang 504

dropuser 506
ecpg 508
pgaccess 512
pg_config 514
pg_dump 516
pg_dumpall 521
pg_restore 523
psql 529
pgtclsh 550
pgtksh 551
vacuumdb 552
III. PostgreSQL 服务器应用 555
initdb 556
initlocation 558
ipcclean 559
pg_ctl 560
pg_passwd 563
postgres 564
postmaster 567
PostgreSQL 7.2.8 开发人员指南 ..... dlxxi
1. PostgreSQL 源代码 1
1.1. 格式 1
2. PostgreSQL 内部概述 2
2.1. 查询的路径 2
2.2. 连接是如何建立的 2
2.3. 解析器阶段 3
2.4. PostgreSQL 规则系统 4
2.5. 规划器/优化器 6
2.6. 执行器 7
3. 系统目录 8
3.1. 概述 8
3.2. pg_aggregate 9
3.3. pg_attrdef 9
3.4. pg_attribute 9
3.5. pg_class 11
3.6. pg_database 12
3.7. pg_description 13
3.8. pg_group 14
3.9. pg_index 14
3.10. pg_inherits 15
3.11. pg_language 15
3.12. pg_largeobject 16
3.13. pg_listener 16
3.14. pg_operator 17
3.15. pg_proc 17
3.16. pg_relcheck 18
3.17. pg_rewrite 19
3.18. pg_shadow 19
3.19. pg_statistic 20
3.20. pg_trigger 21
3.21. pg_type 22
4. 前端/后端协议 25
4.1. Overview 25
4.2. Protocol 25

4.3. 消息数据类型 30
4.4. 消息格式 31
5. gcc 默认优化 38
6. BKI 后端接口 39
6.1. BKI 文件格式 39
6.2. BKI 命令 39
6.3. 示例 40
7. 页文件 41
8. 遗传查询优化 42
8.1. 将查询处理看成是一个复杂的优化问题 42
8.2. 遗传算法 42
8.3. PostgreSQL 中的遗传查询优化（GEQO） 43
8.4. 进一步阅读 44
9. 本地语言支持 45
9.1. 给翻译者 45
9.2. 给程序员 47
A. CVS 代码库 50
A.1. 通过匿名 CVS 获取源码 50
A.2. CVS 树的组织 51
A.3. 通过 CVSup 获取源码 53
B. 文档 58
B.1. DocBook 58
B.2. Toolsets 58
B.3. 构建文档 62
B.4. 文档编写 65
索引 68

PostgreSQL 7.2.8 教程

PostgreSQL 全球开发组

PostgreSQL 7.2.8 教程

PostgreSQL 全球开发组

法律声明

目录

欢迎 xiii
1. 从头开始 1
    1.1. 安装 1
    1.2. 架构基础 1
    1.3. 创建数据库 1
    1.4. 访问数据库 3
2. SQL语言 5
    2.1. 简介 5
    2.2. 概念 5
    2.3. 创建新表 5
    2.4. 向表中填充行 6
    2.5. 查询表 7
    2.6. 表之间的连接 8
    2.7. 聚合函数 10
    2.8. 更新 11
    2.9. 删除 11
3. 高级特性 13
    3.1. 简介 13
    3.2. 视图 13
    3.3. 外键 13
    3.4. 事务 14
    3.5. 继承 15
    3.6. 小结 16

欢迎

欢迎使用PostgreSQL和 PostgreSQL 教程。下面几章旨在为对 PostgreSQL、关系数据库概念或 SQL 语言中任一方面的新手提供简单的入门介绍。我们只假定你对如何使用计算机有一些一般的了解。不需要特别的 Unix 或编程经验。

在学完本教程后，你可能想继续阅读PostgreSQL 7.2.8 用户指南以获得对 SQL 语言更正式的了解，或阅读PostgreSQL 7.2.8 程序员指南以了解为PostgreSQL开发应用的信息。

希望你使用PostgreSQL的经历愉快。

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，请参阅管理员指南中的安装说明，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：

```
psql: could not connect to server: Connection refused
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
createdb: database creation failed
```

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：

```
ERROR:  CREATE DATABASE: permission denied
createdb: database creation failed
```

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 31 个字符。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。

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 ApplixWare（通过 ODBC）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 PostgreSQL 程序员指南。

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, 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 7.2devel on i586-pc-linux-gnu, compiled by GCC 2.96
(1 row)
```

```
mydb=> SELECT current_date;
    date
------------
 2001-08-31
(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的完整功能见参考手册。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 执行简单的操作。本教程只想给你一个入门，绝不是 SQL 的完整教程。关于 SQL92 已有许多著作，包括[melt93]和[date97]。你应当知道，某些 PostgreSQL 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 命令从指定文件中读入命令。-s 选项让你进入单步模式，它会在把每个查询发送给服务器之前暂停。本节使用的命令在文件 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）。

表被组织成数据库，而由单个 PostgreSQL 服务器实例管理的一组数据库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');
```

注意所有数据类型都使用相当明显的输入格式。不是简单数值的常量通常必须用单引号（'）包围，就像例子中那样。date 列实际上接受相当灵活的输入，但在本教程中我们将坚持使用这里所示的明确格式。

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';
```

其中源文件的文件名必须是后端服务器机器（而不是客户端）能访问到的，
因为文件是由后端服务器直接读取的。关于 COPY 命令的更多信息见参考手册。

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)
```

你可以在目标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，你可以要求选择的结果按排序顺序返回，或者去掉重复的行。（为了确保不让下面的内容困扰你：DISTINCT 和 ORDER BY 可以分开使用。）

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

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

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)
```

这个查询叫做左外连接，因为位于连接运算符左侧的表的每一行至少会在输出中出现一次，
而右侧的表只有那些与左表某行匹配的行才会被输出。当输出的左表行在右表中没有匹配时，
右表的列会以空（NULL）值代替。

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)
```

如果我们想知道这个读数出现在哪个（哪些）城市，我们可能会试试

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

但这行不通，因为聚合 max 不能用在 WHERE 子句中。（这个限制的存在是因为 WHERE 子句决定哪些行进入聚合阶段，所以它必须在聚合函数计算之前求值。）不过，正如常见的情况，可以把查询改写成另一种形式来达到想要的结果，这里用的是子选择：

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

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

这样是可以的，因为子选择是一个独立的计算，它独立于外层选择所做的事情计算自己的聚合。

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%'
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

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

你可以用 UPDATE 命令更新现有的行。假设你发现从 11 月 28 日起所有温度读数都偏了 2 度，你可以这样更新数据：

```
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)
```

对如下形式的查询应当保持警惕

```
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:  <unnamed> referential integrity violation - key referenced
 from weather not found in 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，
更多信息请参见参考手册。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。
这些特性在用户指南和程序员指南中有更详细的讨论。

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

PostgreSQL 7.2.8 用户指南

PostgreSQL 全球开发组

PostgreSQL 7.2.8 用户指南

PostgreSQL 全球开发组

法律声明

目录

前言 xxv
1. 什么是PostgreSQL? xxv
2. PostgreSQL简史 xxv
2.1. 伯克利 POSTGRES 项目 xxv
2.2. Postgres95 xxvi
2.3. PostgreSQL xxvi
3. 文档资源 xxvii
4. 术语和记法 xxviii
5. 缺陷报告指南 xxviii
5.1. 识别缺陷 xxix
5.2. 报告哪些内容 xxix
5.3. 到哪里报告缺陷 xxxi
6. Y2K 声明 xxxi
1. SQL 语法 1
1.1. 词法结构 1
1.1.1. 标识符和关键词 1
1.1.2. 常量 2
1.1.3. 操作符 4
1.1.4. 特殊字符 4
1.1.5. 注释 5
1.2. 列 5
1.3. 值表达式 6
1.3.1. 列引用 7
1.3.2. 位置参数 7
1.3.3. 操作符调用 7
1.3.4. 函数调用 7
1.3.5. 聚合表达式 8
1.3.6. 类型转换 8
1.3.7. 标量子查询 9
1.4. 词法优先级 9
2. 查询 11
2.1. 概述 11
2.2. 表表达式 11
2.2.1. FROM 子句 11
2.2.2. WHERE 子句 14
2.2.3. GROUP BY 和 HAVING 子句 15
2.3. 选择列表 16
2.3.1. 列标签 17
2.3.2. DISTINCT 17
2.4. 组合查询 18
2.5. 行排序 18
2.6. LIMIT 和 OFFSET 19
3. 数据类型 20
3.1. 数字类型 21
3.1.1. 整数类型 22
3.1.2. 任意精度数字 22
3.1.3. 浮点类型 23
3.1.4. serial 类型 23
3.2. 货币类型 24
3.3. 字符类型 24
3.4. 二进制串 26

3.5. 日期/时间类型 27
3.5.1. 日期/时间输入 28
3.5.2. 日期/时间输出 31
3.5.3. 时区 32
3.5.4. 内部实现 33
3.6. 布尔类型 33
3.7. 几何类型 34
3.7.1. 点 34
3.7.2. 线段 35
3.7.3. 方框 35
3.7.4. 路径 35
3.7.5. 多边形 36
3.7.6. 圆 36
3.8. 网络地址数据类型 37
3.8.1. inet 37
3.8.2. cidr 37
3.8.3. inet vs cidr 38
3.8.4. macaddr 38
3.9. 位串类型 38
4. 函数和操作符 40
4.1. 逻辑操作符 40
4.2. 比较操作符 40
4.3. 数学函数和操作符 42
4.4. 字符串函数和操作符 46
4.5. 二进制串函数和操作符 50
4.6. 模式匹配 52
4.6.1. 使用 LIKE 进行模式匹配 52
4.6.2. POSIX正则表达式 53
4.7. 数据类型格式化函数 55
4.8. 日期/时间函数和操作符 59
4.8.1. EXTRACT, date_part 62
4.8.2. date_trunc 65
4.8.3. 当前日期/时间 66
4.9. 几何函数和操作符 67
4.10. 网络地址类型函数 70
4.11. 序列操作函数 72
4.12. 条件表达式 73
4.13. 杂项函数 75
4.14. 聚合函数 76
4.15. 子查询表达式 77
5. 类型转换 82
5.1. 简介 82
5.2. 概述 82
5.3. 操作符 83
5.4. 函数 86
5.5. 查询目标 88
5.6. UNION 和 CASE 结构 89
6. 数组 91
7. 索引 95
7.1. 简介 95
7.2. 索引类型 95
7.3. 多列索引 96
7.4. 唯一索引 97

7.5. 函数索引 97
7.6. 操作符类 98
7.7. 键 98
7.8. 部分索引 99
7.9. 检查索引使用情况 101
8. 继承 103
9. 多版本并发控制 106
9.1. 介绍 106
9.2. 事务隔离 106
9.3. 读已提交隔离级别 106
9.4. 可串行化隔离级别 107
9.5. 应用级别的数据一致性检查 107
9.6. 锁与表 108
9.6.1. 表级锁 108
9.6.2. 行级锁 109
9.7. 锁与索引 109
10. 管理数据库 111
10.1. 数据库创建 111
10.2. 访问数据库 111
10.3. 删除数据库 112
11. 性能提示 113
11.1. 使用EXPLAIN 113
11.2. 规划器使用的统计信息 116
11.3. 用显式 JOIN 控制规划器 118
11.4. 填充一个数据库 119
11.4.1. 禁用自动提交 119
11.4.2. 使用 COPY FROM 119
11.4.3. 移除索引 120
11.4.4. 之后的 ANALYZE 120
A. 日期/时间支持 121
A.1. 日期/时间关键字 121
A.2. 时区 122
A.2.1. 澳大利亚时区 124
A.2.2. 日期/时间输入解释 125
A.3. 历法的历史 126
B. SQL 关键字 128
参考书目 143

表格清单

1.1. 操作符优先级（递减） 9
3.1. 数据类型 20
3.2. 数字类型 21
3.3. 货币类型 24
3.4. 字符类型 24
3.5. 特殊字符类型 25
3.6. 二进制串类型 26
3.7. SQL 字面量转义字节 26
3.8. SQL 输出转义字节 26
3.9. SQL99 二进制串与 PostgreSQL `BYTEA` 类型的对比 27
3.10. 日期/时间类型 27
3.11. 日期输入 28
3.12. 时间输入 29
3.13. 带时区的时间输入 29
3.14. 时区输入 30
3.15. 特殊日期/时间常量 31
3.16. 日期/时间输出风格 31
3.17. 日期顺序惯例 32
3.18. 几何类型 34
3.19. 网络地址数据类型 37
3.20. `cidr` 类型输入示例 37
4.1. 比较操作符 40
4.2. 数学操作符 42
4.3. 位串二进制操作符 42
4.4. 数学函数 43
4.5. 三角函数 46
4.6. SQL字符串函数和操作符 46
4.7. 其他字符串函数 48
4.8. SQL二进制串函数和操作符 50
4.9. 其他二进制串函数 51
4.10. 正则表达式匹配操作符 53
4.11. 格式化函数 55
4.12. 用于日期/时间转换的模板模式 56
4.13. 用于日期/时间转换的模板模式修饰符 57
4.14. 用于数值转换的模板模式 58
4.15. `to_char`示例 58
4.16. 日期/时间操作符 60
4.17. 日期/时间函数 60
4.18. 几何操作符 67
4.19. 几何函数 68
4.20. 几何类型转换函数 69
4.21. `cidr` 和 `inet` 操作符 70
4.22. `cidr` 和 `inet` 函数 70
4.23. `macaddr` 函数 71
4.24. 序列函数 72
4.25. 会话信息函数 75
4.26. 系统信息函数 75
4.27. 访问权限查询函数 75
4.28. 目录信息函数 76
4.29. 注释信息函数 76
4.30. 聚合函数 76

9.1. 隔离级别 106
11.1. pg_stats 的列 117
A.1. 月份缩写 121
A.2. 星期缩写 121
A.3. 字段修饰符 121
A.4. 时区 122
A.5. 澳大利亚时区 125
B.1. SQL 关键字 128

范例清单

3.1. 使用字符类型 25
3.2. 使用boolean类型 33
3.3. 使用位串类型 38
5.1. 指数操作符类型解析 84
5.2. 字符串连接操作符类型解析 85
5.3. 绝对值与阶乘操作符的类型解析 85
5.4. 阶乘函数参数类型解析 87
5.5. substr 函数类型解析 87
5.6. character 的存储类型转换 88
5.7. 联合中未充分指定的类型 89
5.8. 简单联合中的类型转换 89
5.9. 转置联合中的类型转换 89
7.1. 建立一个部分索引以排除常见值 100
7.2. 建立一个部分索引以排除不感兴趣的值 100
7.3. 建立一个部分唯一索引 101

前言

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项目由 Michael Stonebraker 教授领导，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 公司的赞助。

PostgreSQL是这一原始伯克利代码的开源后代。它提供 SQL92/SQL99 语言支持和其他现代特性。

POSTGRES开创的许多对象-关系概念如今才在一些商业数据库中变得可用。
传统的关系数据库管理系统（RDBMS）支持的数据模型由一组命名的关系组成，
关系中包含特定类型的属性。在当前的商业系统中，可用的类型包括浮点数、整数、字符串、货币和日期。人们普遍认识到，这个模型对于未来的数据处理应用是不够的。
关系模型之所以能够成功地取代以前的模型，部分原因在于它的“斯巴达式的简洁”。然而，
这种简洁性使某些应用的实现变得非常困难。
PostgreSQL通过以下面这种方式纳入下列附加概念，提供了可观的额外能力，
使用户可以轻松地扩展系统：

- 继承
- 数据类型
- 函数

其他特性提供了额外的能力和灵活性：

- 约束
- 触发器
- 规则
- 事务完整性

这些特性使PostgreSQL属于被称为对象-关系的数据库类别。注意，这不同于那些被称为
面向对象的数据库，后者通常不太适合支持传统的关系数据库语言。因此，尽管 PostgreSQL有一些面向对象的特性，它仍然坚定地站在关系数据库的世界中。事实上，
一些商业数据库最近纳入了PostgreSQL开创的特性。

2. PostgreSQL简史

如今名为 PostgreSQL（并曾短暂称为 Postgres95）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 已成为当今任何地方都能获得的最先进的开源数据库，它提供多版本并发控制，支持几乎所有 SQL 结构（包括子查询、事务以及用户定义的类型和函数），并有广泛的语言绑定可用（包括 C、C++、Java、Perl、Tcl 和 Python）。

2.1. 伯克利 POSTGRES 项目

POSTGRES DBMS 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]。

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

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。`libpq` 接口仍然可用于 C 程序。
- 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。）
- 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。

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

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。

PostgreSQL 中的主要增强包括：

- 表级锁已被多版本并发控制取代，它允许读取者在写入者活动期间继续读取一致的数据，并使得在数据库保持可查询的同时，可以用 pg_dump 进行热备份。
- 实现了重要的后端特性，包括子查询、默认值、约束和触发器。
- 增加了额外的符合 SQL92 的语言特性，包括主键、带引号的标识符、字面串类型强制转换、类型转换以及二进制和十六进制整数输入。
- 内置类型得到了改进，包括新的大范围日期/时间类型和额外的几何类型支持。
- 后端代码的整体速度提升了大约 20-40%，而且自 6.0 版发布以来，后端启动时间缩短了 80%。

3. 文档资源

本手册集被组织为若干部分：

教程

面向新用户的非正式介绍

用户指南

记录 SQL 查询语言环境，包括数据类型和函数。

程序员指南

面向应用程序员的高级信息。主题包括类型和函数可扩展性、库接口以及应用设计问题。

管理员指南

安装和服务器管理信息

参考手册

SQL 命令语法以及客户端和服务器程序的参考页

开发者指南

面向 PostgreSQL 开发者的信息。此书面向为 PostgreSQL 项目做贡献的人；应用开发的信息见程序员指南。

除本手册集之外，还有其他资源可以帮助你安装和使用 PostgreSQL：

man 手册页

参考手册的页面采用传统的 Unix man 格式。

FAQs

常见问题（FAQ）列表既记录一般性问题，也记录一些平台特定的问题。

READMEs

某些附带的软件包提供了 README 文件。

网站

PostgreSQL 网站[5]提供最新版本的详细信息、即将推出的特性，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。

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 网站的 User's Lounge[6] 栏目。

你自己！

PostgreSQL 是一项开源工作。因此，它的持续支持依赖于用户社区。当你开始使用 PostgreSQL 时，你会依赖他人的帮助——无论是通过文档还是通过邮件列表。请考虑回馈你的知识。如果你学到了文档中没有的东西，把它写出来并贡献回去。
如果你给代码添加了特性，也把它们贡献出去。

即使经验不多的人也能为文档提供更正和小改动，而这是一个很好的起点。<pgsql-docs@postgresql.org> 邮件列表就是开始的地方。

4. 术语和记法

术语“PostgreSQL”和“Postgres”将互换使用，指的是随本文档一同发布的软件。

一般来说，管理员是负责安装和运行服务器的人。用户则是任何正在使用或想要使用 PostgreSQL 系统任一部分的人。这些术语不应解释得过于狭隘；本文档集对系统管理规程没有固定的假定。

我们使用 /usr/local/pgsql/ 作为安装的根目录，/usr/local/pgsql/data 作为存放数据库文件的目录。这些目录在你的站点上可能不同，细节可以在管理员指南中推导得到。

在命令概要中，方括号（[和]）表示可选的短语或关键字。花括号（{ 和 }）中含有竖线（|）时表示你必须选择其中一个替代项。

示例将展示从不同账户和程序执行的命令。从 Unix shell 执行的命令前可能带有美元符（“$”）。从特定用户账户（如 root 或 postgres）执行的命令会特别标出并解释。SQL 命令前可能带有“=>”提示符，也可能没有前导提示符，视上下文而定。

> **注意**
>
> 在整个文档集中，标记命令的记法并不完全一致。如发现问题请报告给文档邮件列表 <pgsql-docs@postgresql.org>。

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。

[5] http://www.postgresql.org
[6] http://www.postgresql.org/users-lounge/

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**
>
> 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`postmaster --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。你还可以查看源代码目录中的 `README` 文件，或者查看你的发行文件或软
件包的名称。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 CVS 快照，也请说明，并给出它的日期和时间。

  如果你的版本早于 7.2.8，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。

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“postmaster”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。

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-ports@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。

6. Y2K 声明

> **作者**
>
> 由 Thomas Lockhart（<lockhart@fourpalms.org>）写于 1998-10-22。更新于 2000-03-31。

PostgreSQL 全球开发组将 PostgreSQL软件代码树作为公共服务提供，不对其行为或性能作任何保证，也不承担任何责任。不过，在撰写本文时：

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

- 本声明的作者自 1996 年 11 月起就是 PostgreSQL 支持团队的一名志愿者，他不知道PostgreSQL代码库中存在任何与 2000 年 1 月 1 日前后的时间过渡（Y2K）相关的问题。
- 本声明的作者没有听说在回归测试中或在 PostgreSQL近期或当前版本的其他实际使用中发现过任何 Y2K 问题的报告。考虑到已安装的用户基础以及用户在支持邮件列表上的积极参与，如果存在问题，我们本应有所耳闻。
- 就作者所知，PostgreSQL 对用两位数字年份指定的日期所做的假定记录在当前用户指南关于数据类型的章节中。对于两位年份，关键的过渡年份是 1970 而不是 2000；例如`70-01-01`被解释为 1970-01-01，而`69-01-01`被解释为 2069-01-01。
- 底层操作系统中与获取“当前时间”相关的任何 Y2K 问题都可能传播为 PostgreSQL中表面的Y2K 问题。

关于 Y2K 问题的进一步讨论，特别是与开源、免费软件相关的讨论，请参阅 The GNU Project[8] 和 The Perl Institute[9]。

[8] http://www.gnu.org/software/year2000.html
[9] http://language.perl.com/news/y2k.html

第 1 章 SQL 语法

本章描述 SQL 的语法。

1.1. 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参考手册。

1.1.1. 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 B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、数字（`0-9`）或下划线，不过 SQL 标准不会定义包含数字或以下划线开头、结尾的关键词。

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 32，因此标识符的最大长度是 31（但在编译系统时，可以修改 `src/include/postgres_ext.h` 中的 `NAMEDATALEN`）。

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。
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。[1]

1.1.2. 常量

在 PostgreSQL 中有四种隐式类型常量：字符串、位串、整数和浮点数。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。

1.1.2.1. 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 ASCII 代码的字符。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，PostgreSQL 在这一点上与 SQL9x 一致。

1.1.2.2. 位串常量

[1] 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。

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。位串常量可以以与常规字符串常量相同的方式跨行继续。

1.1.2.3. 整数常量

SQL 中的整数常量是不带小数点也不带指数的十进制数字（0 到 9）序列。合法值的范围取决于所用的整数数据类型，但普通的 integer 类型接受从 -2147483648 到 +2147483647 的值。（可选的正号或负号实际上是一个单独的一元操作符，而不是整数常量的一部分。）

1.1.2.4. 浮点常量

浮点常量接受下列一般形式：

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

其中 *digits* 是一个或多个十进制数字。小数点前后必须至少有一位数字。如果存在指数定界符（e），其后必须至少跟随一位数字。这样，浮点常量通过小数点或指数子句（或两者）的存在与整数常量区分开来。常量中不能嵌入空格或其他字符。

这些是合法浮点常量的一些示例：

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

浮点常量的类型是 DOUBLE PRECISION。REAL 可以用 SQL 字符串记法或 PostgreSQL 类型记法显式指定：

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

1.1.2.5. 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 1.3.6 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 1.3.6 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。

1.1.2.6. 数组常量

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。（对于所有内建类型，这都是逗号字符“,”。）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。一个数组常量的示例是

```
'{{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。

单个数组元素可以放在双引号（"）之间，以避免与空白相关的歧义问题。若不加引号，
数组值解析器将跳过前导空白。

（数组常量实际上只是前一节讨论的一般类型常量的一种特例。
该常量最初被当作字符串处理并被传给数组的输入转换例程。可能需要显式的类型说明。）

1.1.3. 操作符

一个操作符是由下列列表中的字符组成的一个序列，长度最多为 NAMEDATALEN-1 个字符（默认是 31）：

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

不过，操作符名称也有一些限制：

- $（美元符号）不能作为单字符操作符，但它可以作为多字符操作符名称的一部分。
- -- 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。

1.1.4. 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中的位置参数。在其他上下文中，
  美元符号可以是操作符名称的一部分。

- 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 6 章。
- 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 6 章）。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（.）被用在浮点常量中，并且被用来分割表和列名。

1.1.5. 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 SQL92 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 SQL99 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。

1.2. 列

一个列既可以是给定表的一个用户定义列，也可以是下列系统定义列之一：

`oid`

一行的对象标识符（对象 ID）。这是 PostgreSQL 自动为所有表行添加的一个序列号（除非表创建时使用了 WITHOUT OIDS，这种情况下该列不存在）。

`tableoid`

包含该行的表的 OID。对于从继承层次中选择的查询来说，这个属性特别有用，因为没有它就很难分辨一行来自哪个具体的表。`tableoid` 可以与 `pg_class` 的 `oid` 列连接来获得表名。

`xmin`

插入该元组的事务的标识（事务 ID）。（注意：元组是一行的单个状态；对一行的每次更新都会为同一逻辑行创建一个新元组。）

`cmin`

插入事务内部的命令标识符（从零开始）。

`xmax`

删除事务的标识（事务 ID），对于未删除的元组为零。在一个可见元组中该字段也可能非零：这通常表示删除事务尚未提交，或者一次尝试的删除已被回滚。

`cmax`

删除事务内部的命令标识符，或为零。

`ctid`

该元组在其表中的元组 ID。这是一对（块号，块内元组索引），标识该元组的物理位置。注意，虽然 `ctid` 可以用来非常快速地定位元组，但一行的 `ctid` 在每次被更新或被 `VACUUM FULL` 移动时都会改变。因此 `ctid` 作为长期行标识符是没有用的。应该使用 OID，或者更好的是用户定义的序列号来标识逻辑行。

OID 是 32 位的量，从一个集群范围的计数器分配。在一个大的或长期运行的数据库中，该计数器有可能回绕。因此，除非你采取措施确保 OID 唯一，否则假设 OID 唯一是不好的做法。使用 OID 进行行标识时的推荐做法是，为每个将使用 OID 的表在 OID 列上创建一个唯一约束。绝不要假设 OID 跨表唯一；如果需要一个数据库范围的标识符，请使用 `tableoid` 与行 OID 的组合。（PostgreSQL 的未来版本很可能为每个表使用单独的 OID 计数器，因此 `tableoid` 必须被包括进来才能得到全局唯一的标识符。）

事务标识符是 32 位的量。在一个长期运行的数据库中，事务 ID 有可能回绕。在有适当维护过程的情况下这并不是致命问题；详见管理员指南。然而，依赖事务 ID 在长期（超过十亿个事务）内的唯一性是不明智的。

命令标识符也是 32 位的量。这带来一个单个事务内 2^{32}（40 亿）条 SQL 命令的硬限制。实践中这个限制不成问题——注意该限制针对的是 SQL 查询的数目，而不是处理的元组数目。

1.3. 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；见第 1.1.2 节。
- 一个列引用
- 在一个函数声明体中的一个位置参数引用
- 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
- (*expression*)

圆括号被用来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 4 章中的合适位置解释。一个示例是IS NULL子句。

我们已经在第 1.1.2 节中讨论过常量。下面的小节会讨论剩下的选项。

1.3.1. 列引用

一个列可以以下面的形式被引用：

correlation.*columnname* `['*subscript*`]'

correlation 是一个表的名字、一个由 FROM 子句定义的表别名，或者是关键词 NEW 或 OLD 之一。（NEW 和 OLD 只能出现在规则的动作部分，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。如果 *column* 是一个数组类型，那么可选的 *subscript* 选择该数组中的一个或多个元素。如果没有提供下标，那么选中整个数组。（关于数组的更多信息见第 6 章。）

1.3.2. 位置参数

位置参数引用用来表示 SQL 函数中的一个参数。这典型地用在 SQL 函数定义语句中。参数的形式是：

$*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。

1.3.3. 操作符调用

对于一次操作符调用，有三种可能的语法：

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

其中 *operator* 词元遵循第 1.1.3 节的语法规则，或者是关键词 AND，OR 和 NOT 之一。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 4 章描述了内置操作符。

1.3.4. 函数调用

函数调用的语法是：函数名（须遵循第 1.1.1 节中标识符的语法规则）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 4 章中。其他函数可以由用户增加。

1.3.5. 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，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 4.14 节中描述。其他聚合函数可以由用户增加。

1.3.6. 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 SQL92，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 1.1.2.5 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），
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。

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，由于解析器冲突，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。

1.3.7. 标量子查询

标量子查询是一个放在圆括号中的 SELECT，它恰好返回一行一列。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。另见第 4.15 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;
```

1.4. 词法优先级

操作符的优先级和结合性是固定写在解析器中的。大多数操作符具有相同的优先级，
并且是左结合的。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 1.1. 操作符优先级（递减）

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

第 2 章 查询

2.1. 概述

查询是从数据库检索数据的过程或用于检索数据的命令。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。最简单的一种查询具有形式

```
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，客户端库则会提供检索单个行和列的函数。）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，或者在检索之前先对这些列进行计算；参见第 2.3 节。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）。

FROM table1是一种特别简单的表表达式。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();
```

2.2. 表表达式

表表达式指定一个表。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变换的流水线，这些变换作用于由 FROM 子句派生出的表。所有这些变换所产生的派生表提供输入行，用于按选择列表中的列值表达式计算输出行。

2.2.1. 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，也可以是子查询、表连接等推导表，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），形成派生表，随后该派生表可以被 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。

2.2.1.1. 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。支持 INNER、OUTER 和 CROSS JOIN。

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。交叉连接等效于 `INNER JOIN ON TRUE`。

> **提示**
>
> `FROM T1 CROSS JOIN T2`等效于 `FROM T1, T2`。

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`对所有连接都是可选的。`INNER`是默认值；`LEFT`、`RIGHT`和`FULL`表示 OUTER JOIN。

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 TRUE，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 JOIN 有以下几种类型：

INNER JOIN

对于 T1 的每一行 R1，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 INNER JOIN。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 NULL 值补齐。因此，连接表里为来自 T1 的每一行都无条件地至少包含一行。

RIGHT OUTER JOIN

首先，执行一次 INNER JOIN。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 NULL 值补齐。这是左连接的反面：结果表里为来自 T2 的每一行都无条件地包含一行。

FULL OUTER JOIN

首先，执行一次 INNER JOIN。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。可以在 JOIN 子句周围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。

2.2.1.2. 子查询

指定派生表的子查询必须用圆括号括起来，并且必须用 AS 子句命名。（参见第 2.2.1.3 节。）

```
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。

2.2.1.3. 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。

```
FROM table_reference AS alias
```

这里，*alias*可以是任意常规标识符。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;
```

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。

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 ...
```

```
FROM table_reference alias
```

这种形式与前面处理的那种等价；AS关键字只是可选的语法噪音。

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

在这种形式中，除了像上面描述的那样重命名表之外，
还会为表的列赋予临时名字供外围查询使用。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。

2.2.1.4. 例子

```
FROM T1 INNER JOIN T2 USING (C)
FROM T1 LEFT OUTER JOIN T2 USING (C)
FROM (T1 RIGHT OUTER JOIN T2 ON (T1.C1=T2.C1)) AS DT1
FROM (T1 FULL OUTER JOIN T2 USING (C)) AS DT1 (DT1C1, DT1C2)

FROM T1 NATURAL INNER JOIN T2
FROM T1 NATURAL LEFT OUTER JOIN T2
FROM T1 NATURAL RIGHT OUTER JOIN T2
FROM T1 NATURAL FULL OUTER JOIN T2

FROM (SELECT * FROM T1) DT1 CROSS JOIN T2, T3
FROM (SELECT * FROM T1) DT1, T2, T3
```

上面是一些连接表和复杂派生表的例子。注意 AS 子句如何重命名或命名派生表，以及其后可选的逗号分隔列名列表如何重命名列。最后两个 FROM 子句从 T1、T2 和 T3 产生相同的派生表。在把子查询命名为 DT1 时省略了 AS 关键字。关键字 OUTER 和 INNER 也是可以省略的噪音。

2.2.2. 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 1.3 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，派生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 NULL）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 语法移植到其他产品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 A ON/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。

```
FROM FDT WHERE
    C1 > 5

FROM FDT WHERE
    C1 IN (1, 2, 3)
FROM FDT WHERE
    C1 IN (SELECT C1 FROM T2)
FROM FDT WHERE
    C1 IN (SELECT C3 FROM T2 WHERE C2 = FDT.C1 + 10)

FROM FDT WHERE
    C1 BETWEEN (SELECT C3 FROM T2 WHERE C2 = FDT.C1 + 10) AND 100

FROM FDT WHERE
    EXISTS (SELECT C1 FROM T2 WHERE C2 > FDT.C1)
```

在上面的例子中，`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。
限定列名即使在不必要的情况下也能增加清晰度。
这说明了外层查询的列命名作用域如何扩展到其内部查询。

2.2.3. 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（这与 ORDER BY 子句不同）。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，或者获得应用于各组的聚合。

一旦一个表被分组，没有参与分组的列就不能被引用，除非是在聚合表达式中，
因为那些列中的特定值是有歧义的——它应该来自组中的哪一行？参与分组的列可以在选择列表的列表达式中引用，因为它们在每个组都有已知的常量值。
对未分组列的聚合函数提供的是跨一个组的各行（而不是整张表）的值。例如，
在按产品代码分组的表上，`sum(sales)`给出每种产品的总销售额，
而不是所有产品的总销售额。在未分组列上计算的聚合代表的是组，
而未分组列的单个值则不代表组。

例子：

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

在这个示例里，列`pid`、`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。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 去根据查询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，用于从分组表中删除某些组。PostgreSQL允许 HAVING 子句在没有 GROUP BY 的情况下使用，这时它的作用就像另一个 WHERE 子句，但这样使用 HAVING 的意义并不明确。一个好的经验法则是 HAVING 条件应该引用聚合函数的结果。不涉及聚合的限制条件用 WHERE 子句表达更高效。其语法是

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_
expression
```

一个更实际的例子：

```
SELECT pid    AS "Products",
       p.name AS "Over 5000",
       (sum(s.units) * (p.price - p.cost)) AS "Past Month Profit"
  FROM products p LEFT JOIN sales s USING ( pid )
  WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
  GROUP BY pid, 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，而 HAVING 子句则把输出限制为总销售收入超过 5000 的组。

2.3. 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。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 1.3 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 2.2.1.3 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.b, tbl1.c FROM ...
```

（另请参阅第 2.2.2 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检索到的行计算一次，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。

2.3.1. 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 ...
```

如果没有通过 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 2.2.1.3 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。

2.3.2. 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。NULL 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，因此输出中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 中的子选择，可以避免使用这种结构，但它往往是最方便的替代方案。

2.4. 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
```

它实际上说的是

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

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，除非指定了 ALL。

INTERSECT返回那些同时存在于*query1*和*query2*的结果中的行，除非指定了 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。同样，除非指定了 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 5.6 节中描述的那样。

2.5. 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 2.3.1 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 DBMS。

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。

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说明所决定的顺序下相等的行继续排序。

2.6. LIMIT 和 OFFSET

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

LIMIT 允许你仅检索查询其余部分所生成行的一部分。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。LIMIT ALL 的效果和省略 LIMIT 子句一样。

OFFSET 说明在开始向客户端返回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，那么最好用一个 ORDER BY 子句把结果行约束成一个唯一的顺序。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 3 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 3.1列出了标准发行版中包含的所有通用数据类型。“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 3.1. 数据类型

类型名	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		二维平面上的矩形框
bytea		二进制数据
character(*n*)	char(*n*)	定长字符串
character varying(*n*)	varchar(*n*)	变长字符串
cidr		IP 网络地址
circle		二维平面上的圆
date		日历日期（年、月、日）
double precision	float8	双精度浮点数
inet		IP 主机地址
integer	int, int4	有符号4字节整数
interval(*p*)		通用时间段
line		二维平面上的无限直线
lseg		二维平面上的线段
macaddr		MAC 地址
money		美式货币
numeric [(*p*, *s*)]	decimal [(*p*, *s*)]	精度可选择的精确数值
oid		对象标识符
path		二维平面上的开放和闭合几何路径
point		二维平面上的几何点
polygon		二维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

类型名	别名	描述
time [(*p*)] with time zone	timetz	一天中的时间，包括时区
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

SQL 指定了下列类型（或其拼写）：bit、bit varying、boolean、char、character、character varying、varchar、date、double precision、integer、interval、numeric、decimal、real、smallint、time、timestamp（带或不带时区）。

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）。
大多数对应于基本类型（如整数和浮点数）的输入和输出函数会做一些错误检查。
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下溢或上溢。

3.1. 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 3.2. 数字类型

类型名	存储尺寸	描述	范围
smallint	2字节	定点数	-32768 到 +32767
integer	4字节	定点数的常用选择	-2147483648 到 +2147483647
bigint	8字节	极大范围定点数	-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	自动递增的整数	1到 9223372036854775807

数字类型常量的语法在第 1.1.2 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 4 章。下面的几节详细描述这些类型。

3.1.1. 整数类型

smallint、integer 和 bigint 类型存储整数，即不带小数部分的数，取值范围各异。试图存储超出允许范围的值将导致错误。

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，它们也被其他各种 RDBMS 产品共用。

> **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。

3.1.2. 任意精度数字

numeric 类型可以存储大小和精度几乎不受限制的数字，同时能精确存储所有数字并精确执行全部计算。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。

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

3.1.3. 浮点类型

real 和 double precision 数据类型是不精确的可变精度数值类型。实践中，在底层处理器、操作系统和编译器支持的范围内，这些类型通常实现为 IEEE 754 二进制浮点（分别为单精度和双精度）。

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。

通常，real 类型的取值范围至少是 -1E+37 到 +1E+37，精度至少为 6 位十进制数字。double precision 类型通常的取值范围约为 -1E+308 到 +1E+308，精度至少 15 位。太大或太小的值会导致错误。输入数字精度过高时可能发生四舍五入。与零太接近而无法与零区分的数字会导致下溢错误。

3.1.4. serial 类型

serial 数据类型并不是真正的类型，而只是为在表中设置唯一标识符列提供的一种记法上的便利。在当前的实现中，指定

```
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') UNIQUE
 NOT NULL
);
```

这样，我们创建了一个整数列，并安排其默认值由一个序列生成器指派。同时应用了 UNIQUE 和 NOT NULL 约束，以确保显式插入的值也永远不会重复。

类型名 serial 和 serial4 等效：两者都创建 integer 列。类型名 bigserial 和 serial8 的作用方式相同，只是它们创建 bigint 列。如果你预计在表的整个生命周期中会使用超过 2^{31} 个标识符，就应使用 bigserial。

支撑 serial 类型的隐式序列不会在包含 serial 类型的表被删除时自动删除。因此，按顺序执行下面的命令很可能会失败：

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

该序列会一直留在数据库中，直到用 DROP SEQUENCE 显式删除。（这一烦扰在将来的某个版本中可能会被改变。）

3.2. 货币类型

> **已弃用**
>
> money 类型已被弃用。请改用 numeric 或 decimal 类型，并结合 to_char 函数。在未来的版本中，money 类型可能会变成 numeric 类型之上一个感知区域设置的层。

money 类型以固定小数点表示存储美式货币。如果 PostgreSQL 编译时包含了区域设置支持，则 money 类型使用随区域而定的输出格式。

输入接受多种格式，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'。输出采用后一种形式。

表 3.3. 货币类型

类型名	存储尺寸	描述	范围
money	4字节	定点数	-21474836.48 到 +21474836.47

3.3. 字符类型

表 3.4. 字符类型

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

SQL 定义了两种主要的字符类型：character(*n*) 和 character varying(*n*)，其中 *n* 是一个正整数。这两种类型最多都能存储长度为 *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将导致错误，除非超出的字符全都是空格——此时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。）如果要存储的串比声明的长度短，character 类型的值会以空格填充；character varying 类型的值则直接存储较短的串。

> **注意**
>
> 在 PostgreSQL 7.2 之前，过长的字符串会被静默截断而不引发错误。

记法 `char(n)` 和 `varchar(n)` 分别是 `character(n)` 和 `character varying(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 还支持更通用的 `text` 类型，它可以存储任意长度的字符串。与 `character varying` 不同，`text` 不要求为字符串大小显式声明上限。尽管 `text` 类型不在 SQL 标准中，许多其他 RDBMS 软件包也有它。

这些类型的数据的存储需求是 4 字节加实际字符串，对于 `character` 再加填充。长字符串会被系统自动压缩，因此磁盘上的物理需求可能更少。在任何情况下，可存储的最长字符串约为 1 GB。（数据类型声明中允许的 *n* 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 1.1.2.1 节；关于可用的运算符和函数，参见第 4 章。

例 3.1. 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)
SELECT b, char_length(b) FROM test2;
   b   | char_length
-------+-------------
 ok    |           2
 good  |           5
```

1 函数`char_length`在第 4.4 节中讨论。

PostgreSQL 中还有另外两种定长字符类型。`name` 类型仅用于存储内部系统目录的名称，并非供普通用户使用。其长度目前定义为 32 字节（31 个可用字符加终止符），但应使用宏 `NAMEDATALEN` 引用。该长度在编译时设定（因此可为特殊用途调整）；默认的最大长度在未来的版本中可能改变。类型 `"char"`（注意引号）与 `char(1)` 的不同之处在于它只使用一个字节的存储。它在系统目录内部被用作一种简易的枚举类型。

表 3.5. 特殊字符类型

类型名	存储尺寸	描述
"char"	1字节	单字符内部类型

类型名	存储尺寸	描述
name	32字节	31 字符内部类型

3.4. 二进制串

bytea 数据类型允许存储二进制串。

表 3.6. 二进制串类型

类型名	存储尺寸	描述
bytea	4字节外加实际的二进制串	变长（无特定限制）的二进制串

二进制串是不关联任何字符集或排序规则的八位组序列。Bytea 明确允许存储值为零的字节和其他“不可打印”字节。

某些值的字节必须转义（但所有字节值都可以转义）在 SQL 语句中用作字符串字面量的一部分时。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。某些字节值有替代的转义序列，如表 3.7所示。

表 3.7. SQL 字面量转义字节

十进制字节值	描述	输入转义表示	示例	打印结果
0	零字节	'\\000'	select '\\000'::bytea;	\000
39	单引号	'\\'' or '\\047'	select '\''::bytea;	'
92	反斜线	'\\\\' or '\\134'	select '\\\\'::bytea;	\\

注意，上面每个示例的结果都恰好是一个字节长，
尽管零字节和反斜杠的输出表示多于一个字符。Bytea 的输出字节也会被转义。一般而言，每个“不可打印”字节的十进制值都会被转换为等值的三位八进制数，并在前面加一个反斜杠。
大多数“可打印”字节以其在客户端字符集中的标准表示出现。十进制值为 92（反斜杠）的字节有一个特殊的替代输出表示。细节见表 3.8。

表 3.8. SQL 输出转义字节

十进制字节值	描述	输出转义表示	示例	打印结果
92	反斜线	\\	select '\\134'::bytea;	\\
0 to 31 and 127 to 255	“不可打印”字节	\### (octal value)	select '\\001'::bytea;	\001
32 to 126	“可打印”字节	ASCII 表示	select '\\176'::bytea;	~

SQL 字符串字面量（输入串）必须以两个反斜杠为前导，因为它们在 PostgreSQL 后端中必须经过两个解析器。第一个反斜杠会被字符串字面量解析器解释为转义字符并因此被消耗，
留下其后的字节。剩下的反斜杠被 bytea 输入函数识别为三位八进制值的前缀。例如，以 '\\001' 形式传给后端的字符串字面量，经过字符串字面量解析器之后变成 '\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。

由于类似的原因，反斜杠必须输入为 '\\\\'（或 '\\134'）。第一和第三个反斜杠会被字符串字面量解析器解释为转义字符并因此被消耗，留给 bytea 输入函数的字符串中有两个反斜杠，而

该函数把它们解释为表示单个反斜杠。例如，以 '\\\\' 形式传给后端的字符串字面量，经过字符串字面量解析器之后变成 '\\'。然后 '\\' 被送到 bytea 输入函数，在那里它被转换为十进制值为 92 的单个字节。

单引号稍有不同：它必须输入为 '\''（或 '\\134'），而不是 '\\''。这是因为，虽然字面量解析器把单引号当作特殊字符处理并会消耗那个反斜杠，但 bytea 输入函数不把单引号识别为特殊字节。因此，以 '\'' 形式传给后端的字符串字面量，经过字符串字面量解析器之后变成 '''。然后 ''' 被送到 bytea 输入函数，在那里它保持其十进制值为 39 的单个字节不变。

根据你所用的 PostgreSQL 前端不同，在对 bytea 字符串进行转义和反转义方面可能还有额外的工作要做。例如，如果你的接口会自动翻译换行符和回车符，你可能还必须对它们转义。
如果你所用语言的解析器也把反斜杠当作转义字符，你可能不得不把反斜杠再翻倍。

Bytea 提供了 SQL99 第 4.3 节二进制串类型的大部分功能。SQL99 二进制串与 PostgreSQL bytea 的对比见表 3.9。

表 3.9. SQL99 二进制串与 PostgreSQL BYTEA 类型的对比

SQL99	BYTEA
数据类型名为 BINARY LARGE OBJECT 或 BLOB	数据类型名为 BYTEA
不关联任何字符集或排序规则的八位组序列。	相同
由包含数据类型名和最大字节长度的二进制数据类型描述符描述	由包含数据类型名但不指定最大长度的二进制数据类型描述符描述
所有二进制串都可按照比较谓词的规则相互比较。	相同
二进制串的值只能比较相等。	二进制串的值可以比较相等、大于、大于等于、小于、小于等于
对二进制串进行操作并返回二进制串的运算符包括连接、子串、覆盖和修剪	对二进制串进行操作并返回二进制串的运算符包括连接、子串和修剪。修剪的 leading 和 trailing 参数尚未实现。
涉及二进制串的其他运算符包括长度、位置和 like 谓词	相同
二进制串字面量由偶数个十六进制数字组成，用单引号括起，前带“X”，例如 X'1a43fe'	二进制串字面量由按表 3.7所示规则转义的字节组成

3.5. 日期/时间类型

PostgreSQL 支持全套 SQL 日期和时间类型。

表 3.10. 日期/时间类型

类型	描述	存储尺寸	最早	最晚	精度
timestamp [(*p*)] without time zone	日期和时间	8字节	4713 BC	AD 1465001	1微秒 / 14位
timestamp [(*p*)] [with time zone]	日期和时间	8字节	4713 BC	AD 1465001	1微秒 / 14位
interval [(*p*)]	用于时间间隔	12字节	-178000000年	178000000年	1微秒
date	仅日期	4字节	4713 BC	32767 AD	1日

类型	描述	存储尺寸	最早	最晚	精度
time [(*p*)] [without time zone]	仅一天中的时间	8字节	00:00:00.00	23:59:59.99	1微秒
time [(*p*)] with time zone	仅一天中的时间	12字节	00:00:00.00+12	23:59:59.99-12	1微秒

time、timestamp 和 interval 接受可选的精度值 *p*，它指定秒字段保留的小数位数。默认情况下精度没有显式上限。精度的有效上限由底层的双精度浮点数决定（interval 以秒计，timestamp 以相对 2000-01-01 的秒数计）。*p* 的有用范围对 timestamp 是 0 到约 6，对 interval 则可能更大。系统将接受 0 到 13 的 *p*。

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。

为了确保从早于 7.0 的 PostgreSQL 版本升级的路径，我们识别 datetime（等价于 timestamp）和 timespan（等价于 interval）。这些类型现在被限制为只具有到 timestamp 和 interval 的隐式转换，对它们的支持将在 PostgreSQL 的下一个版本（可能名为 7.3）中移除。

abstime 和 reltime 类型是内部使用的低精度类型。不鼓励在新应用中使用这些类型，并建议在合适时把旧的迁移过去。这些内部类型中的任何一种或全部都可能在未来的版本中消失。

3.5.1. 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 PostgreSQL 格式等。对于某些格式，日期输入中月和日的顺序可能有歧义，因此支持指定所期望的这些字段的顺序。命令 SET DateStyle TO 'US' 或 SET DateStyle TO 'Non European' 指定“月在前”的变体，命令 SET DateStyle TO 'European' 则设置“日在前”的变体。ISO 风格是默认值，但这一默认可以在编译时或运行时改变。

PostgreSQL 在处理日期/时间方面比 SQL 标准要求的更灵活。关于日期/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），参见附录 A。

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 1.1.2.5 节。SQL9x 要求使用下列语法

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

其中可选精度说明中的 *p* 是一个对应于秒字段小数位数的整数。可以为 time、timestamp 和 interval 类型指定精度。

3.5.1.1. date

下面是 date 类型的一些可能输入。

表 3.11. 日期输入

示例	描述
January 8, 1999	无歧义

示例	描述
1999-01-08	ISO-8601 格式，推荐
1/8/1999	美式；在欧洲模式下读作 8 月 1 日
8/1/1999	欧式；在美式模式下读作 8 月 1 日
1/18/1999	美式；在任何模式下都读作 1 月 18 日
19990108	ISO-8601 年、月、日
990108	ISO-8601 年、月、日
1999.008	年和一年中的日子
99008	年和一年中的日子
J2451187	儒略日
January 8, 99 BC	公元前的 99 年

3.5.1.2. time [(*p*)] [without time zone]

按 SQL99，此类型可以写作 `time` 或 `time without time zone`。可选精度 *p* 应在 0 到 13 之间，默认取输入时间字面量的精度。

下面是有效的 `time` 输入。

表 3.12. 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
allballs	和 00:00:00 一样

3.5.1.3. time [(*precision*)] with time zone

此类型由 SQL92 定义，但该定义表现出一些使其用途令人怀疑的特性。在大多数情况下，`date`、`time`、`timestamp without time zone` 和 `timestamp with time zone` 的组合应能提供任何应用所需的完整日期/时间功能。

可选精度 *p* 应在 0 到 13 之间，默认取输入时间字面量的精度。

`time with time zone` 接受所有对 `time` 类型合法的输入，再加上一个合法的时区，如下：

表 3.13. 带时区的时间输入

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

更多时区的示例参见表 3.14。

3.5.1.4. timestamp [(*precision*)] without time zone

`timestamp [ (p) ] without time zone` 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。（见下文。）因此

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

是一个符合 ISO 的有效 `timestamp without time zone` 值。此外，广泛使用的格式

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

也被支持。

可选精度 *p* 应在 0 到 13 之间，默认取输入 `timestamp` 字面量的精度。

对于 `timestamp without time zone`，输入中指定的任何显式时区都会被静默吞掉。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。

3.5.1.5. timestamp [(*precision*)] with time zone

`timestamp` 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。（见下文。）因此

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

是一个符合 ISO 的有效 `timestamp` 值。此外，广泛使用的格式

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

也被支持。

可选精度 *p* 应在 0 到 13 之间，默认取输入 `timestamp` 字面量的精度。

表 3.14. 时区输入

时区	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

3.5.1.6. interval [(*precision*)]

`interval`值可以用下列语法书写：

```
  Quantity Unit [Quantity Unit...] [Direction]
@ Quantity Unit [Quantity Unit...] [Direction]
```

其中：`Quantity`（数量）是一个数字（可带符号）；`Unit`（单位）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`（方向）可以是 `ago`（之前）或为空。艾特符号（`@`）是可有可无的噪声。不同单位的量会按适当的符号隐式累加。

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。

可选精度 *p* 应在 0 到 13 之间，默认取输入字面量的精度。

3.5.1.7. 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输入：CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP。后两个函数接受可选的精度说明。

PostgreSQL 还为方便使用支持几种特殊常量。

表 3.15. 特殊日期/时间常量

常量	描述
epoch	1970-01-01 00:00:00+00（Unix系统时间0）
infinity	晚于其他有效时间
-infinity	早于其他有效时间
invalid	非法输入
now	当前事务时间
today	今天午夜
tomorrow	明天午夜
yesterday	昨天午夜
zulu, allballs, z	00:00:00.00 GMT

'now' 在值首次被解释时求值。

> **注意**
>
> 从 PostgreSQL 7.2 版开始，'current' 不再被支持作为日期/时间常量。以前，'current' 被存储为一个特殊值，只有在用于表达式或类型转换时才求值为 'now'。

3.5.2. 日期/时间输出

输出格式可以用 SET DateStyle 设为四种风格之一：ISO 8601、SQL（Ingres）、传统 PostgreSQL 和德式。默认是 ISO 格式。

表 3.16. 日期/时间输出风格

样式说明	描述	示例
'ISO'	ISO-8601 标准	1997-12-17 07:37:16-08
'SQL'	传统样式	12/17/1997 07:37:16.00 PST
'PostgreSQL'	原始样式	Wed Dec 17 07:37:16 1997 PST
'German'	地区样式	17.12.1997 07:37:16.00 PST

date 和 time 风格的输出当然只是上述示例中相应的日期或时间部分。

SQL 风格有欧式和非欧式（美式）两种变体，它们决定月跟在日后还是日跟在月后。（关于此设置如何影响输入值的解释，另见第 3.5.1 节。）

表 3.17. 日期顺序惯例

样式说明	描述	示例
European	*day*/*month*/*year*	17/12/1997 15:37:16.00 MET
US	*month*/*day*/*year*	12/17/1997 07:37:16.00 PST

`interval` 的输出与输入格式相似，区别在于像 `week` 或 `century` 这样的单位会被转换为年和天。ISO 模式下输出形如

```
[ Quantity Units [ ... ] ] [ Days ] Hours:Minutes [ ago ]
```

有几种方法可以影响日期/时间类型的外观：

- postmaster 启动时后端直接使用的 `PGDATESTYLE` 环境变量。
- 会话启动时前端 libpq 使用的 `PGDATESTYLE` 环境变量。
- `SET DATESTYLE` SQL 命令。

3.5.3. 时区

PostgreSQL 力求在典型用法上与 SQL92 定义兼容。但 SQL92 标准对日期和时间类型及能力有一种奇怪的组合。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 GMT/UTC 的固定整数偏移。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 SQL92 的 `time with time zone` 类型（尽管为了旧应用以及与其他 RDBMS 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。此外，时区支持源自底层操作系统的时区能力，因此可以处理夏令时和其他预期行为。

对于 1902 到 2038 年之间的日期，PostgreSQL 从底层操作系统获得时区支持（接近 Unix 风格系统的典型日期界限）。在此范围之外，所有日期都假定以协调世界时（UTC）指定和使用。

所有日期和时间在内部都以 UTC（传统上称为格林尼治标准时间 GMT）存储。时间在发送给客户端前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。

有几种方法可以影响时区行为：

- postmaster 启动时后端直接使用 `TZ` 环境变量作为默认时区。
- 如果在客户端设置了 `PGTZ` 环境变量，libpq 会在连接时用它向后端发送 `SET TIME ZONE` 命令。
- SQL 命令 `SET TIME ZONE` 设置会话的时区。
- SQL92 的限定词

  timestamp `AT TIME ZONE '`*zone*`'`

  其中 *zone* 可以指定为文本时区（例如 `'PST'`）或区间（例如 `INTERVAL '-08:00'`）。

> **注意**
>
> 如果指定了无效的时区，时区会变成 GMT（至少在大多数系统上如此）。

> **注意**
>
> 如果设置了运行时选项 AUSTRALIAN_TIMEZONES，那么 CST 和 EST 指的是澳大利亚时区而不是美国时区。

3.5.4. 内部实现

PostgreSQL 在所有日期/时间计算中使用儒略日。它们有一个好的特性：在假定一年长度为 365.2425 天的前提下，能正确预测/计算从公元前 4713 年到遥远未来的任何日期。

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。

3.6. 布尔类型

PostgreSQL 提供 SQL99 类型 boolean。boolean 只能处于两种状态之一：“真”或“假”。第三种状态“未知”由 SQL 空值状态表示。

“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 3.2. 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 3.2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 4.12 节。

boolean使用 1 字节存储。

3.7. 几何类型

几何类型表示二维的空间物体。最基础的类型是点，它是所有其他类型的基础。

表 3.18. 几何类型

几何类型	存储尺寸	表示	描述
point	16字节	(x,y)	空间中的点
line	32字节	((x1,y1),(x2,y2))	无限直线
lseg	32字节	((x1,y1),(x2,y2))	有限线段
box	32字节	((x1,y1),(x2,y2))	矩形框
path	4+32n 字节	((x1,y1),...)	封闭路径（类似于多边形）
path	4+32n 字节	[(x1,y1),...]	开放路径
polygon	4+32n 字节	((x1,y1),...)	多边形（类似于封闭路径）
circle	24字节	<(x,y),r>	圆（圆心和半径）

有一套丰富的函数和运算符可用于执行各种几何操作，如缩放、平移、旋转以及求交。

3.7.1. 点

点是几何类型的基础二维构件。

point 用下列语法指定：

```
( x , y )
  x , y
```

其中的参数是

x

x 轴坐标，以浮点数表示

y

y 轴坐标，以浮点数表示

3.7.2. 线段

线段（lseg）由点对表示。

lseg 用下列语法指定：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中的参数是

(*x1*,*y1*)
(*x2*,*y2*)

线段的端点

3.7.3. 方框

方框由作为其两个对角的点对表示。

box 用下列语法指定：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

其中的参数是

(*x1*,*y1*)
(*x2*,*y2*)

方框的对角

方框使用第一种语法输出。输入时角点会被重新排序，先存右上角，再存左下角。
也可以输入方框的其他角点，但左下角和右上角由输入确定并存储。

3.7.4. 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) 可用于在查询中测试这两种类型。

path 用下列语法指定：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
```

```
( x1 , y1   , ... ,   xn , yn )
  x1 , y1   , ... ,   xn , yn
```

其中的参数是

(*x*,*y*)

组成路径的各线段的端点。前导方括号（"["）表示开放路径，前导圆括号（"("）表示封闭路径。

路径使用第一种语法输出。

3.7.5. 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` 用下列语法指定：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

其中的参数是

(*x*,*y*)

组成多边形边界的各线段的端点

多边形的输出使用第一种语法。

3.7.6. 圆

圆由一个圆心和一个半径表示。

`circle` 用下列语法指定：

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

其中的参数是

(*x*,*y*)

圆的圆心

r

圆的半径

圆的输出用第一种语法。

3.8. 网络地址数据类型

PostgreSQL 提供存储 IP 和 MAC 地址的数据类型。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 3.19. 网络地址数据类型

名字	存储尺寸	描述	范围
cidr	12字节	IP 网络	有效的 IPv4 网络
inet	12字节	IP 主机和网络	有效的 IPv4 主机或网络
macaddr	6字节	MAC地址	惯用格式

尚不支持 IPv6。

3.8.1. inet

inet 类型在单个字段中保存一个 IP 主机地址以及可选的其所在子网的标识。子网标识由地址网络部分的位数（即“网络掩码”）表示。如果网络掩码是 32，则该值不表示子网而只表示单个主机。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 *x.x.x.x/y*，其中 *x.x.x.x* 是一个 IP 地址，*y* 是网络掩码的位数。如果省略了 */y* 部分，则网络掩码取 32，该值只表示单个主机。显示时，如果网络掩码为 32，则会省略 */y* 部分。

3.8.2. cidr

cidr 类型保存一个 IP 网络规范。输入和输出格式遵循无类别域间路由（Classless Internet Domain Routing）惯例。指定无类别网络的格式是 *x.x.x.x/y*，其中 *x.x.x.x* 是网络，*y* 是网络掩码的位数。如果省略 *y*，则按较老的有类别编号系统的假定来计算，但至少会大到包含输入中写出的所有八位组。

下面是一些示例：

表 3.20. cidr 类型输入示例

CIDR 输入	CIDR 显示	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

3.8.3. 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。

3.8.4. 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。

3.9. 位串类型

位串是由 1 和 0 组成的串。它们可用于存储或可视化位掩码。SQL 有两种位类型：BIT(*x*) 和 BIT VARYING(*x*)；其中 *x* 是一个正整数。

BIT 类型的数据必须与长度 *x* 完全一致；试图存储更短或更长的位串都是错误的。BIT VARYING 长度可变，最大为声明的最大长度 *x*；更长的串将被拒绝。不带长度的 BIT 等价于 BIT(1)，而不带长度说明的 BIT VARYING 则表示长度不受限。

> **注意**
>
> 在 PostgreSQL 7.2 之前，BIT 类型的数据会在右侧补零。为符合 SQL 标准，这一点已被改变。要实现右侧补零的位串，需要把连接运算符与 substring 函数组合使用。

关于位串常量的语法，参见第 1.1.2.2 节。可用的还有位逻辑运算符和字符串操作函数；见第 4 章。

例 3.3. 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 does not match type bit(3)
SELECT SUBSTRING(b FROM 1 FOR 2) FROM test;
```

第 4 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。
用户也可以按程序员指南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 实现中也存在，而且在很多情况下这些功能在各种产品之间是兼容且一致的。

4.1. 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 NULL 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

4.2. 比较操作符

表 4.1. 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 NULL，可以使用结构

```
expression IS NULL
expression IS NOT NULL
```

或者使用等价但较不标准的结构

```
expression ISNULL
expression NOTNULL
```

请不要写 *expression* = NULL，因为 NULL 不“等于” NULL。（NULL 表示一个未知的值，而两个未知的值是否相等是未知的。）

一些应用可能（错误地）要求 *expression* = NULL 在 *expression* 求值为 NULL 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”。

4.3. 数学函数和操作符

许多 PostgreSQL 类型都提供数学操作符。对于所有可能的组合没有通用数学约定的类型（例如日期/时间类型），我们在后续小节中描述其实际行为。

表 4.2. 数学操作符

名称	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
&	二进制与	91 & 15	11
|	二进制或	32 | 3	35
#	二进制异或	17 # 5	20
~	二进制非	~1	-2
<<	二进制左移	1 << 4	16
>>	二进制右移	8 >> 2	2

位串类型 BIT 和 BIT VARYING 也可以使用“二进制”操作符。

表 4.3. 位串二进制操作符

例子	结果
B'10001' & B'01101'	00001
B'10001' | B'01101'	11101
B'10001' # B'01101'	11110
~ B'10001'	01110
B'10001' << 3	01000
B'10001' >> 2	00100

&、| 和 # 的位串参数必须等长。进行移位时，字符串的原始长度保持不变，如上所示。

表 4.4. 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(numeric)`
返回类型	`numeric`
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)`
返回类型	`dp`
描述	指数
示例	`exp(1.0)`
结果	2.71828182845905
函数	`floor(numeric)`
返回类型	`numeric`
描述	不大于参数的最大整数
示例	`floor(-42.8)`
结果	-43
函数	`ln(dp)`
返回类型	`dp`
描述	自然对数
示例	`ln(2.0)`

结果	0.693147180559945
函数	`log(dp)`
返回类型	`dp`
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(`*e* `dp, `*n* `dp)`
返回类型	`dp`
描述	把一个数提升到指数 *e*
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 之间的值
示例	`random()`

结果	
函数	round(dp)
返回类型	dp
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* integer)
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	sign(numeric)
返回类型	numeric
描述	参数的符号（-1、0、+1）
示例	sign(-8.4)
结果	-1
函数	sqrt(dp)
返回类型	dp
描述	平方根
示例	sqrt(2.0)
结果	1.4142135623731
函数	trunc(dp)
返回类型	dp
描述	向零截断
示例	trunc(42.8)
结果	42
函数	trunc(numeric, *s* integer)
返回类型	numeric
描述	截断到 *s* 位小数
示例	trunc(42.4382, 2)
结果	42.43

在上面的表中，dp 表示 double precision。函数 exp、ln、log、pow、round（1 参数）、sqrt 和 trunc（1 参数）也适用于 numeric 类型以代替 double precision。除非另有说明，返回 numeric 结果的函数接受 numeric 输入参数。这些函数中的许多是在宿主系统的 C 库之上实现的；因此边界情况下的精度和行为可能因宿主系统而异。

表 4.5. 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*)	*y*/*x* 的反正切
cos(*x*)	余弦
cot(*x*)	余切
sin(*x*)	正弦
tan(*x*)	正切

所有三角函数的参数和返回值的类型都是 double precision。

4.4. 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 所有类型的值。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 4.6。这些函数也使用常规的函数调用语法实现（见表 4.7）。

表 4.6. 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	'Postgre' || 'SQL'
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	lower(*string*)
返回类型	text
描述	把字符串转换为小写。
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	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	trim([leading | trailing | both] [*characters*] from *string*)
返回类型	text
描述	移除字符串开始端/结束端/两端（默认为两端）仅含指定字符（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	把字符串转换为大写。
示例	upper('tom')
结果	TOM

下面还有一些其他的字符串处理函数。其中一些在内部用于实现上面列出的 SQL 标准字符串函数。

表 4.7. 其他字符串函数

函数	ascii(text)
返回类型	integer
描述	返回参数第一个字符的 ASCII 码。
示例	ascii('x')
结果	120
函数	btrim(*string* text, *trim* text)
返回类型	text
描述	移除*string*开头和结尾处仅由*trim*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	返回给定 ASCII 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	用 *dest_encoding* 转换 *string*。原来的编码由 *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 表示的text_in_unicode
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	字符串的长度
示例	length('jose')
结果	4
函数	lpad(*string* text, *length* integer [, *fill* text])
返回类型	text

描述	在 *string* 的前面添加字符 *fill*（默认为空格）把它填充到 *length* 长度。如果 *string* 已经超过 *length*，则把它截断（在右边）。
示例	`lpad('hi', 5, 'xy')`
结果	`xyxhi`
函数	`ltrim(`*string* `text, `*trim* `text)`
返回类型	`text`
描述	移除字符串开头处仅由*trim*中的字符组成的最长字符串。
示例	`ltrim('zzzytrim', 'xyz')`
结果	`trim`
函数	`pg_client_encoding()`
返回类型	`name`
描述	返回当前客户端编码的名称。
示例	`pg_client_encoding()`
结果	`SQL_ASCII`
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	`rpad(`*string* `text, `*length* `integer [, `*fill* `text])`
返回类型	`text`
描述	在 *string* 的后面添加字符 *fill*（默认为空格）把它填充到 *length* 长度。如果 *string* 已经超过 *length*，则把它截断。
示例	`rpad('hi', 5, 'xy')`
结果	`hixyx`
函数	`rtrim(`*string* text, *trim* text)
返回类型	`text`
描述	移除字符串结尾处仅由*trim*中的字符组成的最长字符串。
示例	`rtrim('trimxxxx', 'x')`
结果	`trim`
函数	`strpos(`*string*, *substring*`)`
返回类型	`text`
描述	定位指定子串（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	提取指定的子串（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	把文本从多字节编码转换为 ASCII。
示例	to_ascii('Karel')
结果	Karel
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
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	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

to_ascii 函数只支持从 LATIN1、LATIN2、WIN1250（CP1250）转换。

4.5. 二进制串函数和操作符

本节描述用于检查和操作二进制串值的函数和操作符。这里的字符串包括 BYTEA 类型的值。

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 4.8。其中一些函数也使用常规的函数调用语法实现（见表 4.9）。

表 4.8. 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	`'\\\\Postgre'::bytea || '\\047SQL\\000'::bytea`
结果	`\\Postgre'SQL\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]` *`characters`* `from` *`string`*`)`
返回类型	`bytea`
描述	移除字符串开始端/结束端/两端仅含指定字符的最长字符串。
示例	`trim('\\000'::bytea from '\\000Tom\\000'::bytea)`
结果	`Tom`

下面还有一些其他的二进制串处理函数。其中一些在内部用于实现上面列出的 SQL 标准字符串函数。

表 4.9. 其他二进制串函数

函数	`btrim(`*`string`* `bytea, `*`trim`* `bytea)`
返回类型	`bytea`
描述	移除*`string`*开头和结尾处仅由*`trim`*中的字符组成的最长字符串。
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	`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`
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`

4.6. 模式匹配

PostgreSQL 提供了两种独立的模式匹配方法：SQL 的 `LIKE` 操作符和 POSIX 风格的正则表达式。

> **提示**
>
> 如果你的模式匹配需求超出了这些，或者想进行由模式驱动的替换或翻译，可以考虑用 Perl 或 Tcl 编写用户定义的函数。

4.6.1. 使用 LIKE 进行模式匹配

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，
必须在查询中写两个反斜线。这样，要写一个实际匹配字面量反斜线的模式，
就意味着在查询中写四个反斜线。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，
因此你仍需要写两个。）

也可以通过写 ESCAPE '' 选择不使用转义字符。这样就无法关闭模式中下划线和百分号的特殊含义了。

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。

4.6.2. POSIX正则表达式

表 4.10. 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 函数更强大的模式匹配手段。许多 Unix 工具（如 egrep、sed 或 awk）使用的模式匹配语言与此处描述的类似。

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。

正则表达式（RE）按 POSIX 1003.2 的定义有两种形式：现代 RE（大致相当于 egrep 的那些；1003.2 称之为“扩展” RE）和过时的 RE（大致相当于 ed 的那些；1003.2 的“基本” RE）。PostgreSQL 实现的是现代形式。

一个（现代）RE 是一个或多个非空分支，由 | 分隔。它能匹配其中任何一个分支所能匹配的内容。

一个分支是一个或多个片段的串接。它匹配第一个片段的一个匹配后跟第二个片段的一个匹配，依此类推。

一个片段是一个原子，后面可能跟着一个 *、+、? 或界限。原子后跟 * 匹配该原子的 0 个或多个匹配组成的序列。原子后跟 + 匹配该原子的 1 个或多个匹配组成的序列。原子后跟 ? 匹配该原子的 0 个或 1 个匹配组成的序列。

一个界限是 { 后跟一个无符号十进制整数，后面可能跟 ,，再后面可能跟另一个无符号十进制整数，最后总是跟 }。这些整数必须在 0 到 RE_DUP_MAX（255）之间（含），如果有两个整数，第一个不能超过第二个。原子后跟含一个整数 *i* 而没有逗号的界限匹配恰好 *i* 个该原子的匹配

组成的序列。原子后跟含一个整数 *i* 和一个逗号的界限匹配 *i* 个或更多个该原子的匹配组成的序列。原子后跟含两个整数 *i* 和 *j* 的界限匹配 *i* 到 *j*（含）个该原子的匹配组成的序列。

> **注意**
>
> 重复操作符（?、*、+ 或界限）不能跟在另一个重复操作符后面。重复操作符不能作为表达式或子表达式的开头，也不能跟在 ^ 或 | 后面。

一个原子是一个由 () 括起来的正则表达式（匹配该正则表达式的一个匹配）、一个空的 ()（匹配空串）、一个方括号表达式（见下文）、.（匹配任意单个字符）、^（匹配输入字符串开头的空串）、$（匹配输入字符串结尾的空串）、一个 \ 后跟 ^.[$()|*+?{\ 中的一个字符（匹配作为普通字符的该字符）、一个 \ 后跟其他任何字符（匹配作为普通字符的该字符，如同 \ 不存在一样），或者一个没有其他含义的单个字符（匹配该字符）。一个 { 后跟非数字字符时是普通字符，不是界限的开始。以 \ 结束一个 RE 是非法的。

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，必须在查询中写两个反斜线。

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。

要在列表中包含字面字符]，可以让它作为第一个字符（跟在可能的 ^ 之后）。要包含字面字符 -，可以让它作为第一个或最后一个字符，或者一个范围的第二个端点。要把字面 - 用作范围的第一个端点，可以把它放在 [. 和 .] 中使它成为一个排序元素（见下文）。除了这些和某些使用 [的组合（见下一段）之外，所有其他特殊字符，包括 \，在方括号表达式内都失去它们的特殊含义。

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。

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：方括号表达式 [[:<:]] 和 [[:>:]] 分别匹配一个单词开头和结尾的空串。单词定义为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。这是一个与 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。

在一个 RE 可以匹配给定字符串的多个子串的情况下，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，它匹配最长的那个。在整个匹配尽可能长的约束之下，子表

达式也匹配最长的可能子串，在 RE 中靠前的子表达式比靠后的优先。注意，高层子表达式因而比其低层的组成子表达式优先。

匹配长度是以字符衡量的，而不是以排序元素。空串被认为比完全不匹配长。例如，`bb*` 匹配 `abbbc` 的中间三个字符，`(wee|week)(knights|nights)` 匹配 `weeknights` 的全部十个字符，当 `(.*).*` 与 `abc` 匹配时，带圆括号的子表达式匹配全部三个字符，而当 `(a*)*` 与 `bc` 匹配时，整个 RE 和带圆括号的子表达式都匹配空串。

如果指定不区分大小写的匹配，其效果非常类似于字母表中的所有大小写差别都消失了。
当存在多种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，
它实际上被转换为包含两种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，它的所有大小写对应字符都被加入该方括号表达式，因此（例如）`[x]` 变成 `[xX]`，`[^x]` 变成 `[^xX]`。

RE 的长度没有特定的限制，除了内存有限之外。内存使用量与 RE 大小大致成线性关系，并且很大程度上对 RE 的复杂度不敏感，有界重复除外。有界重复通过宏展开实现，如果计数很大或有界重复嵌套，它在时间和空间上的代价都很高。像 `((((a{1,100}){1,100}){1,100}){1,100}){1,100}` 这样的 RE 会（最终）耗尽几乎任何现有机器的交换空间。[1]

4.7. 数据类型格式化函数

作者

由 Karel Zak（<zakkr@zf.jcu.cz>）于 2000-01-24 编写

PostgreSQL 格式化函数提供了一套强大的工具，用于把各种数据类型（日期/时间、整数、浮点、numeric）转换为格式化的字符串以及把格式化的字符串转换回原始的数据类型。
这些函数都使用一个公共的调用约定：第一个参数是要格式化的值，
第二个参数是定义输出或输入格式的模板。

表 4.11. 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(timestamp 'now','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`	把 int4/int8 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, '999D9')`
`to_char(numeric, text)`	`text`	将 numeric 转换为字符串	`to_char(numeric '-125.8', '999D99S')`
`to_date(text, text)`	`date`	将字符串转换为日期	`to_date('05 Dec 2000', 'DD Mon YYYY')`

[1] 注意这是 1994 年写的。数字可能已经变了，但问题依然存在。

函数	返回类型	描述	示例
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')

在输出模板字符串中，有一些特定的模式会被识别，并被替换为来自待格式化值的、格式恰当的数据。任何不是模板模式的文本都简单地按字面复制。类似地，在输入模板字符串中，模板模式标识要查看的输入数据字符串的部分以及要在其中找到的值。

表 4.12. 用于日期/时间转换的模板模式

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	分钟（00-59）
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	大写的月份全称（空格补齐到 9 字符）
Month	首字母大写的月名全称（空格填充到 9 个字符）
month	小写的月份全称（空格补齐到 9 字符）
MON	简写的大写形式的月名（3 个字符）
Mon	简写的首字母大写月名（3 个字符）
mon	简写的小写形式的月名（3 个字符）
MM	月份编号（01-12）
DAY	大写的星期全称（空格补齐到 9 字符）
Day	首字母大写的星期名全称（空格填充到 9 个字符）
day	小写的星期全称（空格补齐到 9 字符）
DY	大写的星期简称（3 个字符）
Dy	简写的首字母大写星期名（3 个字符）
dy	小写的星期简称（3 个字符）

模式	描述
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；SUN=1）
W	月内的周序号（1-5），第一周从该月的第一天开始
WW	一年中的周序号（1-53），第一周从该年的第一天开始
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

某些修饰符可以应用于任何模板模式以改变其行为。例如，“FMMonth”是带“FM”前缀的“Month”模式。

表 4.13. 用于日期/时间转换的模板模式修饰符

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	添加大写序数后缀	DDTH
th 后缀	添加小写序数后缀	DDth
FX 前缀	固定格式全局选项（见下文）	FX Month DD Day
SP 后缀	拼写模式（尚未实现）	DDSP

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') 是对的，但 to_timestamp('2000 JUN','FXYYYY MON') 返回一个错误，因为 to_timestamp 只要求一个空格。
- 如果想在字符串常量中使用反斜杠（“\”），必须输入双反斜杠（“\\”）；例如 '\\HH\\MI\\SS'。对 PostgreSQL 中的任何字符串常量都是如此。
- 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\\"'。
- 如果使用多于 4 位数字的年份，从字符串到 timestamp 或 date 的 YYYY 转换会受限制：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；更好的做法是

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')。

- 在从字符串到时间戳的转换中，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。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 4.14. 用于数值转换的模板模式

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	带负号的负值（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	指定位置上的负号（若数字 < 0）
PL	指定位置上的正号（若数字 > 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	转换为序数
V	移动 *n* 位数字（参阅注解）
EEEE	科学记数法（未实现）

用法说明：

- 使用 SG、PL 或 MI 格式化的符号不是数字中的锚点；例如 to_char(-12,'S9999') 产生 '-12'，但 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 指定一个位数与 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，也不转换十进制数。
- 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 4.15. to_char示例

输入	输出
to_char(now(),'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'

输入	输出
to_char(now(),'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,'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,'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'
to_char(12,'99V999')	' 12000'
to_char(12.4,'99V999')	' 12400'
to_char(12.45,'99V9')	' 125'

4.8. 日期/时间函数和操作符

表 4.17列出了可用于日期/时间值处理的函数。表 4.16图解了基本算术操作符（+、* 等）的行为。格式化函数请参阅第 4.7 节。你应当熟悉日期/时间数据类型的背景信息（见第 3.5 节）。

下面描述的日期/时间操作符对涉及时区的类型和不涉及时区的类型的行为类似。

表 4.16. 日期/时间操作符

名称	例子	结果
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00'
+	time '01:00' + interval '3 hours'	time '04:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00'
-	time '05:00' - interval '2 hours'	time '03:00'
-	interval '2 hours' - time '05:00'	time '03:00:00'
*	interval '1 hour' * int '3'	interval '03:00'
/	interval '1 hour' / int '3'	interval '00:20'

日期/时间函数汇总如下，更多细节在后续小节中给出。

表 4.17. 日期/时间函数

名称	age(timestamp)
返回类型	interval
描述	从今天减去
例子	age(timestamp '1957-06-13')
结果	43 years 8 mons 3 days
名称	age(timestamp, timestamp)
返回类型	interval
描述	参数相减
例子	age('2001-04-10', timestamp '1957-06-13')
结果	43 years 9 mons 27 days
名称	current_date
返回类型	date
描述	今天的日期；见下文
例子	
结果	
名称	current_time
返回类型	time
描述	一天中的时间；见下文
例子	

结果	
名称	current_timestamp
返回类型	timestamp
描述	日期和时间；见下文
例子	
结果	
名称	date_part(text,timestamp)
返回类型	double precision
描述	获取子字段（等价于 extract）；另见下文
例子	date_part('hour', timestamp '2001-02-16 20:38:40')
结果	20
名称	date_part(text,interval)
返回类型	double precision
描述	获取子字段（等价于 extract）；另见下文
例子	date_part('month', interval '2 years 3 months')
结果	3
名称	date_trunc(text,timestamp)
返回类型	timestamp
描述	截断到指定精度；另见下文
例子	date_trunc('hour', timestamp '2001-02-16 20:38:40')
结果	2001-02-16 20:00:00+00
名称	extract(*field* from timestamp)
返回类型	double precision
描述	获取子字段；另见下文
例子	extract(hour from timestamp '2001-02-16 20:38:40')
结果	20
名称	extract(*field* from interval)
返回类型	double precision
描述	获取子字段；另见下文
例子	extract(month from interval '2 years 3 months')
结果	3
名称	isfinite(timestamp)
返回类型	boolean
描述	测试时间戳是否有限（既非无效也非无穷）
例子	isfinite(timestamp '2001-02-16 21:28:30')

结果	true
名称	isfinite(interval)
返回类型	boolean
描述	测试间隔是否有限
例子	isfinite(interval '4 hours')
结果	true
名称	now()
返回类型	timestamp
描述	当前日期和时间（等价于 current_timestamp）；见下文
例子	
结果	
名称	timeofday()
返回类型	text
描述	当前日期和时间；见下文
例子	timeofday()
结果	Wed Feb 21 17:01:13.000126 2001 EST
名称	timestamp(date)
返回类型	timestamp
描述	date 转换为 timestamp
例子	timestamp(date '2000-12-25')
结果	2000-12-25 00:00:00
名称	timestamp(date, time)
返回类型	timestamp
描述	date 和 time 转换为 timestamp
例子	timestamp(date '1998-02-24',time '23:07')
结果	1998-02-24 23:07:00

4.8.1. 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。下面是有效的取值：

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（结果可能为负。）；对于 interval 值，是该间隔中的总秒数

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：982352320

```
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[2]）

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
```
结果：40

```
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
```

[2]如果操作系统实现了闰秒则为 60

结果：28.5

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

从 timestamp 值计算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 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。

```
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 4.7 节。

date_part 函数仿照传统 Ingres 中与 SQL 函数 extract 等价的函数建模：

```
date_part('field', source)
```

注意这里的 *field* 值需要是字符串。date_part 的有效字段取值与 extract 的相同。

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4
```

4.8.2. 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 是一个 timestamp 类型的值表达式（date 和 time 类型的值会被自动转换）。*field* 选择把时间戳值截断到哪种精度。返回值的类型是 timestamp，其中所有低于所选字段的字段都被置为零（日和月则置为一）。

field 的有效值有：

microseconds
milliseconds
second
minute

hour
day
month
year
decade
century
millennium

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
```
结果：2001-02-16 20:00:00+00

```
SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
```
结果：2001-01-01 00:00:00+00

4.8.3. 当前日期/时间

可以使用下面的函数获得当前日期和/或时间：

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME ( precision )
CURRENT_TIMESTAMP ( precision )
```

CURRENT_TIME 和 CURRENT_TIMESTAMP 可以选择性地接受一个精度参数，它使结果被舍入到那么多位小数。如果没有精度参数，结果将给出完整的可用精度。

> **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。

> **注意**
>
> SQL99 标准要求这些函数在不带精度参数时不写任何圆括号。从 PostgreSQL 7.2 开始，可以写一个空的圆括号对，但这已被废弃，在未来的版本中可能被移除。

```
SELECT CURRENT_TIME;
14:39:53.662522-05

SELECT CURRENT_DATE;
2001-12-23

SELECT CURRENT_TIMESTAMP;
2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
2001-12-23 14:39:53.66-05
```

now()是PostgreSQL中与CURRENT_TIMESTAMP等价的传统函数。

还有 timeofday()，由于历史原因它返回一个文本字符串而不是 timestamp 值：

```
SELECT timeofday();
 Sat Feb 17 19:07:32.000126 2001 EST
```

十分重要的是要认识到 CURRENT_TIMESTAMP 及相关函数返回的都是当前事务开始时的时间；在事务运行期间它们的值不增长。但 timeofday() 返回实际的当前时间。

所有日期/时间数据类型还接受特殊的字面值 now 来指定当前日期和时间。因此，下面三个都返回相同的结果：

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';
```

> **注意**
>
> 在创建表时指定 DEFAULT 值不要使用第三种形式。系统会在解析常量时立即把 now 转换为 timestamp，因此在需要默认值时，使用的将是建表时刻的时间！前两种形式在默认值被使用之前不会被求值，因为它们是函数调用。
> 因此它们会给出在插入行时取默认值的期望行为。

4.9. 几何函数和操作符

几何类型 point、box、lseg、line、path、polygon 和 circle 有一大套原生支持的函数和操作符。

表 4.18. 几何操作符

操作符	描述	用法
+	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)'
#	交集	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	多边形中的点数	# '((1,0),(0,1),(-1,0))'
##	最近邻近点	point '(0,0)' ## lseg '((2,0),(0,2))'
&&	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))'
<->	两者之间的距离	circle '((0,0),1)' <-> circle '((5,0),1)'

操作符	描述	用法
<<	是否位于左侧?	`circle '((0,0),1)' << circle '((5,0),1)'`
<^	是否位于下方?	`circle '((0,0),1)' <^ circle '((0,5),1)'`
>>	是否位于右侧?	`circle '((5,0),1)' >> circle '((0,0),1)'`
>^	是否位于上方?	`circle '((0,5),1)' >^ circle '((0,0),1)'`
?#	相交或重叠	`lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'`
?-	是否水平?	`point '(1,0)' ?- point '(0,0)'`
?-|	是否互相垂直?	`lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'`
@-@	长度或周长	`@-@ path '((0,0),(1,0))'`
?|	是否竖直?	`point '(0,1)' ?| point '(0,0)'`
?||	是否平行?	`lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'`
@	包含于或位于其上	`point '(1,1)' @ circle '((0,0),2)'`
@@	中心	`@@ circle '((0,0),10)'`
~=	是否相同?	`polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'`

表 4.19. 几何函数

函数	返回类型	描述	示例
`area`(object)	`double precision`	对象的面积	`area(box '((0,0),(1,1))')`
`box`(box, box)	`box`	交集矩形框	`box(box '((0,0),(1,1))',box '((0.5,0.5),(2,2))')`
`center`(object)	`point`	对象的中心	`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`	对象的长度	`length(path '((-1,0),(1,0))')`

函数	返回类型	描述	示例
pclose(path)	path	将路径转换为闭合路径	popen(path '[(0,0),(1,1),(2,0)]')
npoint(path)	integer	点数	npoints(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	把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	水平尺寸	width(box '((0,0),(1,1))')

表 4.20. 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	转换为圆	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	把点转换为圆	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	中心	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	中心	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	12 点多边形	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	12 点多边形	polygon(circle '((0,0),2.0)')
polygon(*npts*, circle)	polygon	*npts* 点多边形	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))')

4.10. 网络地址类型函数

表 4.21. cidr 和 inet 操作符

操作符	描述	用法
<	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'

inet 的所有操作符也都可以应用于 cidr 值。操作符 <<、<<=、>> 和 >>= 测试子网包含：它们只考虑两个地址的网络部分，忽略任何主机部分，并判断一个网络部分是否与另一个相同或是它的子网。

表 4.22. 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	`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	以文本形式提取 IP 地址和掩码长度
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	以文本形式提取缩写显示
示例	`abbrev(cidr '10.1.0.0/16')`
结果	`10.1/16`

inet 的所有函数也都可以应用于 cidr 值。`host()`、`text()` 和 `abbrev()` 函数主要用于提供替代的显示格式。

表 4.23. 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`

函数 `trunc(macaddr)` 返回一个把最后 3 个字节置为 0 的 MAC 地址。这可以用于把剩余的前缀与一个制造商关联起来。源代码发行版中的 `contrib/mac` 目录包含一些用于创建和维护这种关联表的工具。

`macaddr`类型还支持用于字典序排序的标准关系操作符（`>`、`<=`等）。

4.11. 序列操作函数

表 4.24. 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

本节描述 PostgreSQL 中用于操作序列对象的函数。序列对象（也称为序列生成器或简称序列）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唯一标识符。本节中描述的序列函数都对一个 `regclass` 参数操作，它是序列的 OID。

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
```

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`

返回当前服务器进程中此序列最近一次由 `nextval` 获取的值。（如果本进程中从未对这个序列调用过 `nextval`，则会报告一个错误。）注意，由于返回的是进程局部的值，即使其他服务器进程同时在执行 `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);     同上
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。

如果序列对象是用默认参数创建的，对它的 nextval() 调用将返回从 1 开始的连续值。其他行为可以通过在 CREATE SEQUENCE 命令中使用特殊参数来获得；更多信息见它的命令参考页。

4.12. 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。

CASE

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

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 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 *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 NULL。

一个例子：

```
=>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能强制转换为一个单一的输出类型。更多细节参见第 5.6 节。

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

这个“简单”的 CASE 表达式是上面一般形式的一个特化变体。*expression* 被计算并与 WHEN 子句中的所有 *value* 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 NULL）。

上面的例子可以用简单的 CASE 语法写成：

```
=>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
```

COALESCE

```
COALESCE(value[, ...])
```

COALESCE 函数返回它的第一个非 NULL 参数。这在为显示检索数据时用默认值替换 NULL 值常常很有用，例如：

```
SELECT COALESCE(description, short_description, '(none)') ...
```

NULLIF

```
NULLIF(value1, value2)
```

当且仅当 *value1* 和 *value2* 相等时，NULLIF 函数返回 NULL。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)') ...
```

提示

COALESCE 和 NULLIF 只是 CASE 表达式的缩写。它们实际上在处理的非常早期阶段就被转换为 CASE 表达式，后续处理认为自己在处理 CASE。因此，不正确地使用 COALESCE 或 NULLIF 可能会得到引用 CASE 的错误消息。

4.13. 杂项函数

表 4.25. 会话信息函数

名称	返回类型	描述
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

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。当前它总是等于会话用户，但将来可能会有“setuid”
函数和其他允许当前用户临时更改的设施。用 Unix 的说法，会话用户是“真实用户”，当前用户是“有效用户”。

注意这些函数在 SQL 中具有特殊的语法地位：调用它们时不能带尾部的圆括号。

Deprecated

函数 getpgusername() 是 current_user 的已废弃等价形式。

表 4.26. 系统信息函数

名称	返回类型	描述
version	text	PostgreSQL 版本信息

version() 返回一个描述 PostgreSQL 服务器版本的字符串。

表 4.27. 访问权限查询函数

名称	返回类型	描述
has_table_privilege(*user*, *table*, *access*)	boolean	用户是否有访问表的权限
has_table_privilege(*table*, *access*)	boolean	当前用户是否有访问表的权限

has_table_privilege 判断一个用户能否以某种特定方式访问一个表。用户可以按名称或按 ID（pg_user.usesysid）指定，如果省略该参数则假定为 current_user。表可以按名称或按 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区分。）所需的访问类型由一个文本字符串指定，它必须求值为 SELECT、INSERT、UPDATE、DELETE、RULE、REFERENCES 或 TRIGGER 之一。（但字符串的大小写无关紧要。）

表 4.28. 目录信息函数

名称	返回类型	描述
pg_get_viewdef(*viewname*)	text	获取视图的 CREATE VIEW 命令
pg_get_ruledef(*rulename*)	text	获取规则的 CREATE RULE 命令
pg_get_indexdef(*indexOID*)	text	获取索引的 CREATE INDEX 命令
pg_get_userbyid(*userid*)	name	根据给定的 ID 获取用户名

这些函数从系统目录中提取信息。pg_get_viewdef()、pg_get_ruledef() 和 pg_get_indexdef() 分别重构造视图、规则或索引的创建命令。（注意，这是反编译的重构造，不是该命令的逐字文本。）pg_get_userbyid() 根据一个 usesysid 值提取用户名。

表 4.29. 注释信息函数

名称	返回类型	描述
obj_description(*objectOID*, *tablename*)	text	获取数据库对象的注释
obj_description(*objectOID*)	text	获取数据库对象的注释（已废弃）
col_description(*tableOID*, *columnnumber*)	text	获取表列的注释

这些函数提取先前用 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 NULL。

obj_description() 的双参数形式返回由其 OID 和包含它的系统目录的名称指定的数据库对象的注释。例如，obj_description(123456,'pg_class') 会检索 OID 为 123456 的表的注释。obj_description() 的单参数形式只需要对象 OID。它现在已被废弃，因为无法保证 OID 在不同的系统目录中是唯一的；因此可能返回错误的注释。

col_description() 返回表列的注释，由其表的 OID 和列号指定。obj_description() 不能用于表列，因为列没有自己的 OID。

4.14. 聚合函数

Author

由 Isaac Wilcox <isaac@azartmedia.com> 于 2000-06-16 编写

聚合函数从一组输入值中计算出一个单一的结果值。聚合函数的特殊语法考虑在第 1.3.5 节中解释。更多介绍信息请参阅 PostgreSQL 教程。

表 4.30. 聚合函数

函数	描述	说明
AVG(*expression*)	所有输入值的平均值（算术平均）	可以对下列数据类型求平均值：smallint、integer、

函数	描述	说明
		bigint、real、double precision、numeric、interval。对于任何整数类型输入，结果的类型是 numeric；对于浮点输入是 double precision；否则与输入数据类型相同。
count(*)	输入值的数目	返回值的类型是 bigint。
count(*expression*)	统计*expression*的值不为 NULL 的输入值的个数。	返回值的类型是 bigint。
max(*expression*)	*expression* 在所有输入值中的最大值	可用于所有数字、字符串和日期/时间类型。结果的类型与输入表达式相同。
min(*expression*)	*expression* 在所有输入值中的最小值	可用于所有数字、字符串和日期/时间类型。结果的类型与输入表达式相同。
stddev(*expression*)	输入值的样本标准差	可以对下列数据类型求标准差：smallint、integer、bigint、real、double precision、numeric。对于浮点输入，结果的类型是 double precision；否则是 numeric。
sum(*expression*)	*expression* 在所有输入值上的总和	可以对下列数据类型求和：smallint、integer、bigint、real、double precision、numeric、interval。对于 smallint 或 integer 输入，结果的类型是 bigint；对于 bigint 输入是 numeric；对于浮点输入是 double precision；否则与输入数据类型相同。
variance(*expression*)	输入值的样本方差	方差是标准差的平方。支持的数据类型和结果类型与标准差的相同。

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 NULL。特别地，没有行时 SUM 返回 NULL，而不是人们可能预期的零。必要时可以使用 COALESCE 函数把 NULL 替换为零。

4.15. 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。

EXISTS

```
EXISTS ( subquery )
```

EXISTS 的参数是一个任意的 SELECT 语句，即子查询。对子查询求值以确定它是否返回任何行。如果它返回至少一行，EXISTS 的结果是 TRUE；如果子查询不返回行，EXISTS 的结果是 FALSE。

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。

子查询通常只会执行到足以确定是否至少返回一行，而不会完全执行到结束。
编写有副作用的子查询（如调用序列函数）是不明智的；副作用是否发生可能难以预测。

由于结果只取决于是否返回任何行，而不取决于这些行的内容，
子查询的输出列表通常无关紧要。一个常见的编码惯例是把所有 EXISTS 测试写成 `EXISTS( SELECT 1 WHERE ...)` 的形式。但这条规则也有例外，比如使用 `INTERSECT` 的子查询。

这个简单的例子类似于在 col2 上的内连接，但即使有多个匹配的 tab2 行，它对每个 tab1 行也最多产生一个输出行：

```
SELECT col1 FROM tab1
    WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

IN (scalar form)

expression IN (*value*[, ...])

这种形式的 `IN` 的右边是一个由圆括号括起的标量表达式列表。如果左边表达式的结果等于任何一个右边表达式，结果为 TRUE。这是一种简写记法：

```
expression = value1
OR
expression = value2
OR
...
```

注意，如果左边表达式产生 NULL，或者没有相等的右边值且至少一个右边行产生 NULL，`IN` 结构的结果将是 NULL 而不是 FALSE。这符合 SQL 对 NULL 布尔组合的正常规则。

> **注意**
>
> 这种形式的 `IN` 并不是真正的子查询表达式，但把它与子查询 `IN` 记录在同一地方似乎最合适。

IN (subquery form)

expression IN (*subquery*)

这种形式的 `IN` 的右边是一个带圆括号的子查询，它必须恰好返回一列。左边的表达式被求值并与子查询结果的每一行比较。如果找到任何相等的子查询行，`IN` 的结果为 TRUE。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 FALSE。

注意，如果左边表达式产生 NULL，或者没有相等的右边值且至少一个右边行产生 NULL，`IN` 结构的结果将是 NULL 而不是 FALSE。这符合 SQL 对 NULL 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*[, ...]) 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 TRUE。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 FALSE。

与通常一样，表达式或子查询行中的 NULL 按照 SQL 布尔表达式的正常规则组合。如果两个行的所有对应成员都非空且相等，则认为它们相等；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）。如果所有行比较的结果要么不相等要么为 NULL，且至少有一个为 NULL，那么 IN 的结果为 NULL。

NOT IN (scalar form)

expression NOT IN (*value*[, ...])

这种形式的 NOT IN 的右边是一个由圆括号括起的标量表达式列表。如果左边表达式的结果与所有右边表达式都不相等，结果为 TRUE。这是一种简写记法：

```
expression <> value1
AND
expression <> value2
AND
...
```

注意，如果左边表达式产生 NULL，或者没有相等的右边值且至少一个右边行产生 NULL，NOT IN 结构的结果将是 NULL 而不是人们可能天真期望的 TRUE。这符合 SQL 对 NULL 布尔组合的正常规则。

> **提示**
>
> x NOT IN y 在所有情况下都等价于 NOT (x IN y)。但是，使用 NOT IN 时 NULL 比使用 IN 时更容易让新手犯错。如果可能，最好用肯定的形式表达你的条件。

NOT IN (subquery form)

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，它必须恰好返回一列。左边的表达式被求值并与子查询结果的每一行比较。
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 TRUE。如果找到任何相等的行，结果为 FALSE。

注意，如果左边表达式产生 NULL，或者没有相等的右边值且至少一个右边行产生 NULL，NOT IN 结构的结果将是 NULL 而不是 TRUE。这符合 SQL 对 NULL 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*[, ...]) 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 TRUE。如果找到任何相等的行，结果为 FALSE。

与通常一样，表达式或子查询行中的 NULL 按照 SQL 布尔表达式的正常规则组合。如果两个行的所有对应成员都非空且相等，则认为它们相等；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）。如果所有行比较的结果要么不相等要么为 NULL，且至少有一个为 NULL，那么 NOT IN 的结果为 NULL。

ANY

expressionoperator ANY (*subquery*)
expression operator SOME (*subquery*)

这种形式的 ANY 的右边是一个带圆括号的子查询，它必须恰好返回一列。左边的表达式使用给定的 *operator* 与子查询结果的每一行比较，该操作符必须产生布尔结果。如果得到任何真结果，ANY 的结果为 TRUE。如果没有找到真结果（包括子查询不返回行的特殊情况），结果为 FALSE。

SOME是ANY的同义词。IN等价于= ANY。

注意，如果没有成功且至少一个右边行对操作符的结果产生 NULL，ANY 结构的结果将是 NULL 而不是 FALSE。这符合 SQL 对 NULL 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 查询中只允许使用 = 和 <> 操作符。相应地，如果找到相等或不等的行，ANY 的结果为 TRUE。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 FALSE。

与通常一样，表达式或子查询行中的 NULL 按照 SQL 布尔表达式的正常规则组合。如果两个行的所有对应成员都非空且相等，则认为它们相等；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）。如果至少有一个行比较结果为 NULL，那么 ANY 的结果不可能是 FALSE；它将是 TRUE 或 NULL。

ALL

expressionoperator ALL (*subquery*)

这种形式的 ALL 的右边是一个带圆括号的子查询，它必须恰好返回一列。左边的表达式使用给定的 *operator* 与子查询结果的每一行比较，该操作符必须产生布尔结果。如果所有行都产生真（包括子查询不返回行的特殊情况），ALL 的结果为 TRUE。如果找到任何假结果，结果为 FALSE。

NOT IN等价于<> ALL。

注意，如果没有失败但至少一个右边行对操作符的结果产生 NULL，ALL 结构的结果将是 NULL 而不是 TRUE。这符合 SQL 对 NULL 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 TRUE。如果发现任何一行不相等或相等，结果为 FALSE。

与通常一样，表达式或子查询行中的 NULL 按照 SQL 布尔表达式的正常规则组合。如果两个行的所有对应成员都非空且相等，则认为它们相等；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）。如果至少有一个行比较结果为 NULL，那么 ALL 的结果不可能是 TRUE；它将是 FALSE 或 NULL。

Row-wise comparison

```
(expression, expression[, ...]) operator (subquery)
(expression, expression[, ...]) operator (expression, expression[, ...])
```

左边是标量表达式的列表。右边既可以是相同长度的标量表达式列表，
也可以是一个带圆括号的子查询，该子查询必须恰好返回与左边表达式数量相同的列。此外，
子查询不能返回多于一行。（如果它返回零行，结果视为 NULL。）左边被求值并与子查询结果的唯一一行或右边的表达式列表逐行比较。相应地，如果这两行相等或不相等，结果为 TRUE。

与通常一样，表达式或子查询行中的 NULL 按照 SQL 布尔表达式的正常规则组合。如果两个行的所有对应成员都非空且相等，则认为它们相等；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 5 章 类型转换

5.1. 简介

SQL 查询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强制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 3 章和第 4 章中的相关章节。

程序员指南中有关于隐式类型转换和强制转换所用确切算法的更多细节。

5.2. 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 RDBMS 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。大多数扩展类型首先被词法化为字符串。SQL 语言定义允许用字符串指定类型名，这一机制可以在 PostgreSQL 中用来让解析器走上正确的路径。例如，查询

```
tgl=> SELECT text 'Origin' AS "Label", point '(0,0)' AS "Value";
 Label  | Value
--------+-------
 Origin | (0,0)
(1 row)
```

有两个字面常量，类型分别为 `text` 和 `point`。如果没有为字符串字面量指定类型，则最初会赋予占位类型 unknown（未知），并在后面阶段按如下所述解析。

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。函数调用有一个或多个参数，对任何具体的查询，这些参数都必须与系统目录中可用的函数匹配。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。

查询目标

SQL 的 `INSERT` 和 `UPDATE` 语句把表达式的结果放到表中。查询中的表达式必须与目标列的类型匹配，必要时还要转换成目标列的类型。

UNION 和 CASE 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 结构的结果表达式必须强制转换成公共类型，使 CASE 表达式整体有已知的输出类型。

许多一般类型转换规则使用了建立在 PostgreSQL 函数和操作符系统表之上的简单约定。转换规则中还包含一些启发式规则，以便更好地支持 SQL 标准固有类型（如 smallint、integer 和 real）的约定。

PostgreSQL 解析器使用的约定是：所有类型转换函数都接受一个源类型的参数，并且以目标类型的名称命名。满足这些标准的任何函数都被认为是有效的转换函数，
解析器可以把它当作转换函数使用。
这个简单的假设使解析器无需硬编码就能探索类型转换的可能性，
让扩展的用户定义类型透明地使用这些相同的特性。

解析器中还提供了一个额外的启发式规则，以便对 SQL 标准类型的正确行为做出更好的猜测。定义了几种基本的类型类别：boolean、numeric、string、bitstring、datetime、timespan、geometric、network 以及用户自定义。除用户自定义外，每个类别都有一个在有歧义时会被优先选择的首选类型。在用户自定义类别中，每个类型都是自己的首选类型。
有歧义的表达式（有多个候选解析解的表达式）在有多个可能的内置类型时通常可以解决，
但在用户定义类型有多个选择时则会报错。

所有类型转换规则的设计都遵循若干原则：

- 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）。
- 用户定义的类型之间没有关联。目前，PostgreSQL 没有关于类型之间关系的信息，只有内置类型的硬编码启发式规则以及基于目录中可用函数的隐式关系。
- 如果查询不需要隐式类型转换，解析器或执行器就不应有额外的开销。也就是说，
  如果查询构造良好且类型已经匹配，查询就应当继续进行，不必在解析器上花费额外时间，
  也不必向查询中引入不必要的隐式转换函数。

  此外，如果某个查询通常需要为函数做隐式转换，
  而之后用户用正确的参数类型定义了一个显式函数，解析器就应使用这个新函数，
  不再用旧函数做隐式转换。

5.3. 操作符

操作符调用的操作数类型按下面的过程解析。注意这个过程间接受所涉操作符优先级的影响。
更多信息见第 1.4 节。

操作数类型解析

1. 在 pg_operator 系统目录中检查精确匹配。

   - (可选的) 如果二元操作符的一个参数是 unknown 类型，那么在这一步检查中假定它与另一个参数的类型相同。其他涉及 unknown 的情况在这一步永远找不到匹配。

2. 寻找最佳匹配。

   a. 列出所有同名且输入类型匹配或可以强制转换成匹配的操作符。（为此目的，假定 unknown 字面量可以转换成任何类型。）如果只有一个，就使用它；否则继续下一步。

b. 遍历所有候选，保留输入类型精确匹配最多的。如果没有任何精确匹配，
则保留全部候选。如果只剩一个候选，就使用它；否则继续下一步。

c. 遍历所有候选，保留输入类型精确匹配或二进制兼容匹配最多的。
如果没有任何精确或二进制兼容匹配，则保留全部候选。如果只剩一个候选，
就使用它；否则继续下一步。

d. 遍历所有候选，保留在需要类型强制转换的最多个位置上接受首选类型的。
如果没有任何候选接受首选类型，则保留全部候选。如果只剩一个候选，就使用它；
否则继续下一步。

e. 如果有输入参数是“unknown”，检查其余候选在这些参数位置上接受的类型类别。
在每个位置上，如果有候选接受 "string" 类别就选择该类别（这种偏向字符串的做法是
恰当的，因为未知类型的字面量看起来确实像字符串）。否则，
如果所有剩余候选接受相同的类型类别，就选择该类别；否则失败，
因为没有更多线索就无法推断出正确的选择。
还要注意是否有候选接受所选类别中的首选数据类型。
现在丢弃不接受所选类型类别的操作符候选；此外，
如果某个候选在给定的参数位置上接受首选类型，
就丢弃在该参数上接受非首选类型的候选。

f. 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。

Examples

例 5.1. 指数操作符类型解析

目录中只定义了一个取幂运算符，它接受 `double precision` 类型的参数。扫描器给这个查询表达式的两个参数都赋予了初始类型 `integer`：

```
tgl=> SELECT 2 ^ 3 AS "Exp";
 Exp
-----
   8
(1 row)
```

于是解析器对两个操作数都做类型转换，查询等价于

```
tgl=> SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision)
 AS "Exp";
 Exp
-----
   8
(1 row)
```

或

```
tgl=> SELECT 2.0 ^ 3.0 AS "Exp";
 Exp
-----
   8
(1 row)
```

> **注意**
>
> 最后这种形式的开销最小，因为没有调用任何函数来做隐式类型转换。
> 这对小查询不是问题，但可能影响涉及大表的查询的性能。

例 5.2. 字符串连接操作符类型解析

类字符串的语法既用于处理字符串类型，也用于处理复杂的扩展类型。
未指定类型的字符串会与可能的操作符候选匹配。

一个参数未指定类型的例子：

```
tgl=> 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。

未指定类型上的串接：

```
tgl=> SELECT 'abc' || 'def' AS "Unspecified";
 Unspecified
-------------
 abcdef
(1 row)
```

这种情况下没有关于使用哪个类型的初始提示，因为查询中没有指定类型。
于是解析器查找所有候选操作符，发现有候选既接受 string 类别也接受 bit-string 类别的输入。
由于 string 类别可用时会被优先选择，就选择了该类别，然后把字符串的“首选类型” `text` 用作
解析未知字面量的具体类型。

例 5.3. 绝对值与阶乘操作符的类型解析

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：

```
tgl=> select @ text '-4.5' as "abs";
 abs
-----
 4.5
(1 row)
```

这里系统在应用选中的运算符之前做了一次隐式的 text 到 float8 转换。我们可以验证用的是
float8 而不是别的类型：

```
tgl=> select @ text '-4.5e500' as "abs";
ERROR:  Input '-4.5e500' is out of range for float8
```

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：

```
tgl=> select text '44' ! as "factorial";
ERROR:  Unable to identify a postfix operator '!' for type 'text'
        You may need to add parentheses or an explicit cast
```

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：

```
tgl=> select cast(text '44' as int8) ! as "factorial";
      factorial
----------------------
 2659396885588443136
(1 row)
```

5.4. 函数

函数调用的参数类型按下列步骤解析。

函数参数类型解析

1. 在 pg_proc 系统目录中检查精确匹配。（涉及 unknown 的情况在这一步永远找不到匹配。）

2. 如果目录中没有精确匹配，看看这个函数调用是否是一个平凡的类型强制转换请求。
   当函数调用只有一个参数且函数名与某个数据类型的（内部）名相同时就是这种情况。
   此外，函数参数必须是未知类型的字面量或与该数据类型二进制兼容的类型。
   满足这些条件时，函数参数被强制转换为该数据类型，不会有任何显式的函数调用。

3. 寻找最佳匹配。

   a. 列出所有同名、参数个数相同且输入类型匹配或可以强制转换成匹配的函数。
      （为此目的，假定 unknown 字面量可以转换成任何类型。）如果只有一个，就使用它；
      否则继续下一步。

   b. 遍历所有候选，保留输入类型精确匹配最多的。如果没有任何精确匹配，
      则保留全部候选。如果只剩一个候选，就使用它；否则继续下一步。

   c. 遍历所有候选，保留输入类型精确匹配或二进制兼容匹配最多的。
      如果没有任何精确或二进制兼容匹配，则保留全部候选。如果只剩一个候选，
      就使用它；否则继续下一步。

   d. 遍历所有候选，保留在需要类型强制转换的最多个位置上接受首选类型的。
      如果没有任何候选接受首选类型，则保留全部候选。如果只剩一个候选，就使用它；
      否则继续下一步。

   e. 如果有输入参数是 unknown，检查其余候选在这些参数位置上接受的类型类别。在每个
      位置上，如果有候选接受 string 类别就选择该类别（这种偏向字符串的做法是恰当
      的，因为未知类型的字面量看起来确实像字符串）。否则，
      如果所有剩余候选接受相同的类型类别，就选择该类别；否则失败，
      因为没有更多线索就无法推断出正确的选择。
      还要注意是否有候选接受所选类别中的首选数据类型。
      现在丢弃不接受所选类型类别的候选；此外，
      如果某个候选在给定的参数位置上接受首选类型，
      就丢弃在该参数上接受非首选类型的候选。

   f. 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。

Examples

例 5.4. 阶乘函数参数类型解析

pg_proc 目录中只定义了一个 int4fac 函数。因此下面的查询自动把 int2 参数转换为 int4：

```
tgl=> SELECT int4fac(int2 '4');
 int4fac
---------
      24
(1 row)
```

并且实际上被解析器变换为

```
tgl=> SELECT int4fac(int4(int2 '4'));
 int4fac
---------
      24
(1 row)
```

例 5.5. substr 函数类型解析

pg_proc 中声明了两个 substr 函数。但只有一个接受两个参数，类型是 text 和 int4。

如果用未指定类型的字符串常量调用，该类型会直接与唯一的候选函数类型匹配：

```
tgl=> SELECT substr('1234', 3);
 substr
--------
     34
(1 row)
```

如果该字符串被声明为 varchar 类型（例如它来自一张表时就是这种情况），则解析器会试图把它强制转换成 text：

```
tgl=> SELECT substr(varchar '1234', 3);
 substr
--------
     34
(1 row)
```

它被解析器变换成

```
tgl=> SELECT substr(text(varchar '1234'), 3);
 substr
--------
     34
(1 row)
```

> **注意**
>
> 实际上，解析器知道 text 和 varchar 是二进制兼容的，即一个可以传给接受另一个的函数而不做任何物理转换。因此，这种情况下实际上不会插入显式的类型转换调用。

而且，如果用 int4 调用该函数，解析器会试图把它转换成 text：

```
tgl=> SELECT substr(1234, 3);
 substr
--------
     34
(1 row)
```

它实际执行为

```
tgl=> SELECT substr(text(1234), 3);
 substr
--------
     34
(1 row)
```

这之所以成功，是因为系统目录中有一个转换函数 text(int4)。

5.5. 查询目标

要插入表的值按下列步骤强制转换为目标列的数据类型。

查询目标类型解析

1. 检查是否与目标类型精确匹配。
2. 否则，尝试把表达式强制转换成目标类型。如果两个类型已知二进制兼容，或者存在转换函数，转换就会成功。如果表达式是未知类型的字面量，就把字面字符串的内容送给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，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 5.6. character 的存储类型转换

对于声明为 character(20) 的目标列，下面的查询确保目标的大小被正确调整：

```
tgl=> CREATE TABLE vv (v character(20));
CREATE
tgl=> INSERT INTO vv SELECT 'abc' || 'def';
INSERT 392905 1
tgl=> SELECT v, length(v) FROM vv;
          v           | length
----------------------+--------
 abcdef               |     20
(1 row)
```

这里实际发生的是：两个未知字面量默认被解析为 text，使 || 操作符被解析为 text 连接。然后操作符的 text 结果被强制转换为 bpchar（“blank-padded char”，字符数据类型的内部名）以匹配目标列类型。（由于解析器知道 text 和 bpchar 二进制兼容，这个转换是隐式的，不会插入任何真实的函数调用。）最后，在系统目录中找到尺寸调整函数 bpchar(bpchar, integer)，应用到操作符的结果和存储的列长度上。这个类型特定的函数执行所需的长度检查并添加填充空格。

5.6. UNION 和 CASE 结构

UNION 结构必须把可能不相似的类型匹配起来构成单一结果集。解析算法对联合查询的每个输出列分别应用。INTERSECT 和 EXCEPT 结构用与 UNION 相同的方式解析不相似的类型。CASE 结构也使用同一算法来匹配其组成表达式并选择结果数据类型。

UNION 和 CASE 类型解析

1. 如果所有输入都是 unknown 类型，就解析为 text 类型（string 类别的首选类型）。否则，在选择类型时忽略 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. 如果一个或多个非 unknown 输入是该类别中的首选类型，就解析为该类型。
4. 否则，解析为第一个非 unknown 输入的类型。
5. 把所有输入强制转换为所选类型。

Examples

例 5.7. 联合中未充分指定的类型

```
tgl=> SELECT text 'a' AS "Text" UNION SELECT 'b';
 Text
------
 a
 b
(2 rows)
```

这里，未知类型的字面量 'b' 会被解析成 text 类型。

例 5.8. 简单联合中的类型转换

```
tgl=> SELECT 1.2 AS "Double" UNION SELECT 1;
 Double
--------
      1
    1.2
(2 rows)
```

字面量 1.2 是 double precision 类型，即数值类别中的首选类型，所以使用该类型。

例 5.9. 转置联合中的类型转换

这里 union 的输出类型被强制匹配 union 中第一个子句的类型：

```
tgl=> SELECT 1 AS "All integers"
tgl-> UNION SELECT CAST('2.2' AS REAL);
 All integers
--------------
            1
            2
```

```
(2 rows)
```

由于 REAL 不是首选类型，解析器找不到选它而不选 INTEGER（也就是 1 的类型）的理由，于是退回到使用第一个备选的规则。
这个例子说明首选类型机制并没有编码我们希望的那样多的信息。PostgreSQL　的未来版本可能支持更一般的类型偏好概念。

第 6 章 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。为了说明它们的用途，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。
上面的查询将创建一个名为 sal_emp的表，其中的列包括一个 text字符串（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）。

现在我们做一些INSERT。注意，要写一个数组值，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。（更多细节见下文。）

```
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"}}');
```

现在，我们可以在sal_emp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 subscript* : *upper subscript*来表示。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。

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';
```

也可以在单个元素上更新：

```
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，可以扩大数组。例如，如果一个数组值当前有 4 个元素，那么向`array[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`array[-2:7]`赋值来创建一个下标值从 -2 到 7 的数组。

`CREATE TABLE`的语法允许定义定长数组：

```
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。

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。

要在数组中搜索一个值，必须检查数组的每个值。这可以手工完成（如果你知道数组的大小）：

```
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，而且在数组大小未知时也帮不上忙。虽然它不属于PostgreSQL 的主发行版，但有一个可用的扩展为遍历数组值定义了新的函数和操作符。使用它，上面的查询可以写成：

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1:4] *= 10000;
```

要搜索整个数组（而不只是指定的列），可以使用：

```
SELECT * FROM sal_emp WHERE pay_by_quarter *= 10000;
```

此外，你还可以找出数组中所有值都等于 10 000 的行：

```
SELECT * FROM sal_emp WHERE pay_by_quarter **= 10000;
```

要安装这个可选模块，请查看PostgreSQL 源码发行版中的contrib/array目录。

> **提示**
>
> 数组不是集合；按上一段描述的方式使用数组往往是数据库设计不当的标志。
> 数组字段一般应该拆分到一个独立的表中。表显然很容易搜索。

> **注意**
>
> 当前数组实现的一个限制是：数组的单个元素不能是 SQL NULL。整个数组可以设置为NULL，但不能让数组的一部分元素为 NULL 而另一部分非 NULL。修复这个问题已列入待办事项。

数组输入和输出语法。. 数组值的外部表示由若干项构成，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（datatypes）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。

引用数组元素。. 如上所示，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（datatypes），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（datatypes），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。）

提示

请记住，你在 SQL 查询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，查询中可能需要多达八个反斜线。）

第 7 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。

7.1. 简介

需要使用索引的经典例子是：假设有一张类似这样的表：

```
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;
```

通常情况下，系统必须逐行扫描整个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：当系统认为使用索引比顺序表扫描更高效时，
就会在查询中使用它。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，另请阅读第 11 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。

7.2. 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。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：<, <=, =, >=, >

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 4.9 节。）

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**
>
> 由于 hash 索引的用途有限，通常应该优先选择 B-树索引而不是 hash 索引。我们没有充分的证据表明，即使在 = 比较上，hash 索引也确实比 B-树更快。此外，hash 索引需要更粗粒度的锁；参见第 9.7 节。

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）。

7.3. 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 16 个列。（在构建 PostgreSQL 时可以改变这个限制；参见文件 pg_config.h。）

查询优化器可以把多列索引用于涉及索引中最先的 *n* 个连续列（当与合适的操作符一起使用时）的查询，最大可以达到索引定义中指定的总列数。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。）

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。
超过三个列的索引几乎肯定是不合适的。

7.4. 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。NULL 值不被视为相等。

当为表声明唯一约束或主键时，PostgreSQL会在构成主键或唯一约束的列（必要时是多列索引）
上自动创建唯一索引，以强制执行该约束。稍后任何时候都可以再向表中添加唯一索引，
用来增加一个唯一约束。

> **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。

7.5. 函数索引

函数索引定义在一个函数应用于单个表的一列或多列所得的结果上。
函数索引可用于基于函数调用的结果快速访问数据。

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';
```

如果在`lower(column)`操作的结果上定义了索引，这个查询就可以使用该索引：

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

索引定义中的函数可以接受多个参数，但这些参数必须是表的列，而不能是常量。
函数索引始终是单列的（即函数的结果），即使函数使用了多个输入字段；
不存在包含函数调用的多列索引。

> **提示**
>
> 上一段中提到的限制很容易绕过：定义一个自定义函数用在索引定义中，
> 在内部计算任何想要的结果即可。

7.6. 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，建立在四字节整数上的 B-树索引会使用 int4_ops类；该操作符类包含针对四字节整数的比较函数。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排序方式。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。此外还有一些特殊用途的操作符类：

- 操作符类box_ops和bigbox_ops都支持box数据类型上的 R-树索引。它们的区别在于，bigbox_ops会将框的坐标按比例缩小，以避免在对非常大的浮点坐标做乘法、
  加法和减法时出现浮点异常。如果你的矩形所在的范围大约在 20000 个单位见方或更大，就应该使用bigbox_ops。

下面的查询会显示所有已定义的操作符类：

```
SELECT am.amname AS acc_method,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    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 acc_method, ops_name, ops_comp
```

7.7. 键

作者

由 Herouth Maoz（<herouth@oumail.openu.ac.il>）撰写。它最初于 1998-03-02 发表在用户邮件列表上，作为对问题 “What is the difference between PRIMARY KEY and UNIQUE constraints?” 的回答。

Subject: Re: [QUESTIONS] PRIMARY KEY | UNIQUE

下面两者有什么区别：

PRIMARY KEY(fields,...) 和
UNIQUE (fields,...)

- 这是一个别名吗？
- 如果 PRIMARY KEY 已经是唯一的，那为什么
还会有另一种名为 UNIQUE 的键？

主键是用来标识特定行的字段。例如，用社会保障号码标识一个人。

一个单纯的 UNIQUE 字段组合与标识行无关，它只是一种完整性约束。例如，我有若干链接集合。每个集合由一个唯一的编号标识，这个编号就是主键。这个键用于关系。

不过，我的应用还要求每个集合有一个唯一的名称。
为什么？为了让想要修改集合的人能够识别它。如果你有两个名为 “Life Science” 的集合，就很难知道究竟标记为 24433 的那个才是你需要的，而标记为 29882 的那个不是。

因此，用户通过名称来选择集合。我们因而在数据库中确保名称是唯一的。但是，
数据库中没有其他表通过集合名称与 collections 表关联，那样做会非常低效。

而且，集合名称尽管唯一，实际上并不定义这个集合！例如，如果有人决定把集合的名称从 “Life Science” 改为 “Biology”，它仍然是同一个集合，只是名称不同而已。只要名称是唯一的，
这就没有问题。

所以：

- 主键：
  - 用于标识行并与其他数据相关联。
  - 不可能（或很难）更新。
  - 不应允许 NULL。

- 唯一字段：
  - 用作访问行的另一种途径。
  - 可以更新，只要保持唯一。
  - 可以接受 NULL。

至于为什么标准 SQL 语法中没有显式定义非唯一键？你必须理解，索引是依赖于实现的。SQL 不定义实现，只定义数据库中数据之间的关系。PostgreSQL 确实允许非唯一索引，但用于强制 SQL 键的索引总是唯一的。

因此，你可以按列的任意组合查询一个表，即使你在这些列上并没有索引。索引只是每种 RDBMS 提供给你的实现辅助手段，目的是让常用的查询执行得更高效。有些 RDBMS 还可能提供额外的手段，例如把键保存在主存中。它们会有一个特殊命令，例如

```
CREATE MEMSTORE ON table COLUMNS cols
```

（这不是一个真实存在的命令，只是一个例子。）

事实上，当你创建主键或唯一的字段组合时，SQL 规范中没有任何地方说会创建索引，也没有说按键检索数据会比顺序扫描更高效！

所以，如果你想用一个不唯一的字段组合作为辅助键，
其实完全不必指定任何东西——直接按那个组合开始检索就行了！不过，如果你想让检索高效，
就得求助于你的 RDBMS 供应商提供的手段——无论是索引、我虚构的 `MEMSTORE` 命令，还是一个智能的 RDBMS：它会根据你曾基于特定键组合发送过许多查询这一事实，在你不知情的情况下创建索引……（它从经验中学习）。

7.8. 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 7.1展示了这种思路的一种可能应用。

例 7.1. 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 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 7.2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 7.2. 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 7.2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谓词。（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的 WHERE 条件完全匹配，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 7.3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 7.3. 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 bool,
                    ...);
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject,
 target)
    WHERE success;
```

当成功的试验很少而不成功的试验很多时，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]。

7.9. 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。检查索引使用情况可以使用 EXPLAIN 命令；它在这方面的应用见第 11.1 节。

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（在管理员指南中有描述）。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。计划节点的代价可以通过运行时参数来调整（在管理员指南中有描述）。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（参见 ALTER TABLE 参考页）。

  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 8 章 继承

我们来创建两个表。capitals 表包含各州首府，它们同时也是城市。自然地，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 用于变长 ASCII 字符串的原生类型。属性 population 的类型是 `float`，这是 PostgreSQL 用于双精度浮点数的原生类型。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，下面的查询找出所有位于海拔 500 英尺以上的城市（包括州首府）的名称：

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

它返回：

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

另一方面，下面的查询找出所有不是州首府且位于海拔 500 英尺或以上的城市：

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

+----------+----------+
|name      | altitude |
```

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

这里 cities 前面的“ONLY”表示查询只应在 cities 上执行，而不包括继承层次中位于 cities 之下的表。我们已经讨论过的许多命令—— SELECT、UPDATE 和 DELETE——都支持这种“ONLY”记法。

有时你可能想知道某个特定元组来自哪个表。每个表中有一个名为 TABLEOID 的系统列，它可以告诉你来源表：

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

它返回：

```
+---------+----------+----------+
|tableoid |name      | altitude |
+---------+----------+----------+
|37292    |Las Vegas | 2174     |
+---------+----------+----------+
|37280    |Mariposa  | 1953     |
+---------+----------+----------+
|37280    |Madison   | 845      |
+---------+----------+----------+
```

如果与 pg_class 做连接，就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 |
+---------+----------+----------+
|capitals |Las Vegas | 2174     |
+---------+----------+----------+
|cities   |Mariposa  | 1953     |
+---------+----------+----------+
|cities   |Madison   | 845      |
+---------+----------+----------+
```

> **已废弃**
>
> 在以前的 PostgreSQL 版本中，默认不访问子表。这被发现容易出错，也违反 SQL99。在旧语法下，要获得子表，需在表名后追加 *。例如

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

仍然可以通过追加 * 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同时扫描其子表，而以前的默认行为则不是这样。要获得旧的默认行为，可把配置选项 SQL_Inheritance 设为 off，例如

```
SET SQL_Inheritance TO OFF;
```

或者在你的 postgresql.conf 文件中加一行。

继承特性的一个局限是，索引（包括唯一约束）和外键约束只作用于单个表，
而不作用于其继承子表。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，但不包含首府名。这一缺陷可能会在未来的某个版本中修复。

第 9 章 多版本并发控制

多版本并发控制（MVCC）是一种在多用户环境下提升数据库性能的高级技术。Vadim Mikheev（<vadim@krs.ru>）为 PostgreSQL 提供了实现。

9.1. 介绍

与大多数使用锁进行并发控制的其他数据库系统不同，PostgreSQL 通过使用多版本模型来维护数据一致性。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。

多版本模型与锁模型的主要区别在于：在 MVCC 中，为查询（读取）数据而获取的锁不会与为写入数据而获取的锁冲突，因此读取永远不会阻塞写入，写入也永远不会阻塞读取。

9.2. 事务隔离

ANSI/ISO 的 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：

脏读

一个事务读取了由并发未提交事务写入的数据。

不可重复读

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）。

幻读

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 9.1。

表 9.1. 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。

9.3. 读已提交隔离级别

读已提交是 PostgreSQL 中的默认隔离级别。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。（不过，SELECT 能看到同一事务中先前执行的更

新效果，即使这些更新尚未提交。）注意，如果其他事务在第一个 SELECT 执行期间提交了更改，即使两个连续的 SELECT 命令位于同一个事务中，它们也可能看到不同的数据。

如果查询在执行 UPDATE 语句（或 DELETE 或 SELECT FOR UPDATE）时找到的目标行，已经被一个并发未提交事务更新，那么试图更新该行的第二个事务会等待另一个事务提交或回滚。若对方回滚，等待的事务可以继续更改该行。若对方提交（且该行仍然存在，即没有被另一个事务删除），查询会针对该行重新执行，以检查新行版本是否仍然满足查询的搜索条件。如果新行版本满足查询搜索条件，该行将被更新（或删除或标记为更新）。注意，更新的起点将是新行版本；而且，更新之后这个被更新过两次的行对当前事务中后续的 SELECT 可见。因此，当前事务能够看到另一个事务对这一特定行的效果。

读已提交级别所提供的部分事务隔离对许多应用来说已经足够，而且这一级别快速又易于使用。不过，对于执行复杂查询和更新的应用，可能需要保证比读已提交级别更严格一致的数据库视图。

9.4. 可串行化隔离级别

可串行化提供最高的事务隔离。该级别模拟串行的事务执行，就好像事务是一个接一个串行执行的，而不是并发执行的。不过，使用这一级别的应用必须准备好在出现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，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 能看到同一事务中先前执行的更新效果，即使这些更新尚未提交。）这与读已提交的不同之处在于，SELECT 看到的是事务开始时刻的快照，而不是事务内当前查询开始时刻的快照。

如果查询在执行 UPDATE 语句（或 DELETE 或 SELECT FOR UPDATE）时找到的目标行，已经被一个并发未提交事务更新，那么试图更新该行的第二个事务会等待另一个事务提交或回滚。若对方回滚，等待的事务可以继续更改该行。若并发事务提交，可串行化事务将被回滚并收到以下消息：

```
ERROR:  Can't serialize access due to concurrent update
```

因为可串行化事务不能修改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，只有更新型事务才可能需要重试——只读事务永远不会发生串行化冲突。

可串行化事务级别严格保证每个事务都看到完全一致的数据库视图。不过，当并发更新使得无法维持串行执行的假象时，应用必须准备好重试事务，而重做复杂事务的代价可能很高。因此，只有当更新查询包含的逻辑复杂到在读已提交级别下可能给出错误结果时，才推荐使用这一级别。

9.5. 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（即当前事务开始之后的某个时刻）仍然存在；该行可能已经被一个在本事务开始之后提交的已提交事务修改或删除。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；
但如果客户端能通过数据库之外的渠道通信，就可能引发严重的混乱。

要确保某一行当前仍然存在，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 保护整张表。）从其他环境向 PostgreSQL 移植应用时，应当考虑这一点。

> **注意**
>
> 在 6.5 版之前，PostgreSQL 使用的是读锁，因此从较早的 PostgreSQL 版本升级到 6.5（或更高）时，上述考虑同样适用。

9.6. 锁与表

PostgreSQL 提供多种锁模式来控制对表中数据的并发访问。其中一些锁模式由 PostgreSQL 在语句执行之前自动获取，另一些则供应用使用。事务中获取的所有锁模式在该事务期间一直保持。

9.6.1. 表级锁

AccessShareLock

在被查询的表上自动获取的一种读锁模式。

只与 AccessExclusiveLock 冲突。

RowShareLock

由 SELECT FOR UPDATE 以及 IN ROW SHARE MODE 的 LOCK TABLE 语句获取。

与 ExclusiveLock 和 AccessExclusiveLock 模式冲突。

RowExclusiveLock

由 UPDATE、DELETE、INSERT 以及 IN ROW EXCLUSIVE MODE 的 LOCK TABLE 语句获取。

与 ShareLock、ShareRowExclusiveLock、ExclusiveLock 和 AccessExclusiveLock 模式冲突。

ShareUpdateExclusiveLock

由（不带 FULL 的）VACUUM 以及 IN SHARE UPDATE EXCLUSIVE MODE 的 LOCK TABLE 语句获取。

与 ShareUpdateExclusiveLock、ShareLock、ShareRowExclusiveLock、ExclusiveLock 和 AccessExclusiveLock 模式冲突。

ShareLock

由 CREATE INDEX 以及 IN SHARE MODE 的 LOCK TABLE 语句获取。

与 RowExclusiveLock、ShareUpdateExclusiveLock、ShareRowExclusiveLock、ExclusiveLock 和 AccessExclusiveLock 模式冲突。

ShareRowExclusiveLock

由 `IN SHARE ROW EXCLUSIVE MODE` 的 `LOCK TABLE` 语句获取。

与 RowExclusiveLock、ShareUpdateExclusiveLock、ShareLock、ShareRowExclusiveLock、ExclusiveLock 和 AccessExclusiveLock 模式冲突。

ExclusiveLock

由 `IN EXCLUSIVE MODE` 的 `LOCK TABLE` 语句获取。

与 RowShareLock、RowExclusiveLock、ShareUpdateExclusiveLock、ShareLock、ShareRowExclusiveLock、ExclusiveLock 和 AccessExclusiveLock 模式冲突。

AccessExclusiveLock

由 `ALTER TABLE`、`DROP TABLE`、`VACUUM FULL` 和 `LOCK TABLE` 语句获取。

与所有模式冲突（AccessShareLock、RowShareLock、RowExclusiveLock、ShareUpdateExclusiveLock、ShareLock、ShareRowExclusiveLock、ExclusiveLock 和 AccessExclusiveLock）。

注意

只有 AccessExclusiveLock 会阻塞（不带 `FOR UPDATE` 的）`SELECT` 语句。

9.6.2. 行级锁

行级锁在行被更新（或删除或标记为更新）时获取。行级锁不影响数据查询。
它们只阻塞同一行的写者。

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。

除了表锁和行锁之外，还使用短期的共享/排他锁来控制对共享缓冲池中表页的读写访问。
这些锁在取到或更新一个元组后立即释放。应用开发者通常不需要关心页级锁，
但为完整性起见我们提一下。

9.7. 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：

GiST 和 R-树索引

读/写访问使用共享/排他的索引级锁。锁在语句完成之后释放。

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，但可能发生死锁。

B-树索引

读/写访问使用短期的共享/排他页级锁。锁在每个索引元组被取到或插入之后立即释放。

B-树索引在不产生死锁条件的前提下提供最高的并发性。

总之，对于并发应用，推荐使用 B-树索引类型。

第 10 章 管理数据库

虽然站点管理员负责 PostgreSQL 安装的整体管理，但安装中的某些数据库可以由另一个人管理，这个人被指定为数据库管理员。这种职责分配发生在创建数据库时。
一个用户可以被授予创建数据库和/或创建新用户的显式权限。
被同时授予这两种权限的用户可以在 PostgreSQL 内执行大多数管理任务，但默认不会拥有与站点管理员相同的操作系统权限。

管理员指南更详细地讨论这些主题。

10.1. 数据库创建

数据库由从 PostgreSQL 内部发出的 `CREATE DATABASE` 命令创建。createdb 是一个 shell 脚本，用来在 Unix 命令行提供同样的功能。

无论哪种方法，要成功执行都必须有 PostgreSQL 后端在运行，而且发出命令的用户必须是 PostgreSQL 超级用户，或者已被超级用户授予了数据库创建权限。

要从命令行创建一个名为 `mydb` 的新数据库，输入

```
% createdb mydb
```

要在 psql 内做同样的事，输入

```
=> CREATE DATABASE mydb;
```

如果你没有创建数据库所需的权限，你会看到下列消息：

```
ERROR:  CREATE DATABASE: Permission denied.
```

你会自动成为你刚创建的数据库的数据库管理员。数据库名的第一个字符必须是字母，
并且长度限制为 31 个字符。PostgreSQL 允许你在给定站点创建任意数量的数据库。

管理员指南更详细地讨论数据库创建，包括 `CREATE DATABASE` 命令的高级选项。

10.2. 访问数据库

一旦构造好数据库，就可以通过以下方式访问它：

- 运行 PostgreSQL 的交互式终端程序 psql，它允许你交互式地输入、编辑和执行 SQL 命令。
- 使用现有的图形化前端工具，例如 PgAccess 或 ApplixWare（通过 ODBC）来创建和操作数据库。这些可能性不在本教程讨论范围内。
- 编写自定义应用程序，使用若干可用语言绑定中的一种。这些可能性在PostgreSQL 程序员指南中有进一步讨论。

你大概想启动 psql，来试试本手册中的例子。可以通过输入以下命令为 mydb 数据库激活它：

```
% psql mydb
```

你会看到下面的欢迎消息：

```
Welcome to psql, 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=>
```

这个提示表明 `psql` 正在监听你，你可以把 SQL 查询输入到由终端监视程序维护的工作区中。psql 程序本身响应以反斜杠字符 `\` 开头的特殊命令。例如，你可以获得各种 PostgreSQL SQL 命令的语法帮助，方法是输入：

```
mydb=> \h
```

一旦在工作区中输入完查询，你可以通过输入下列内容把工作区的内容传递给 PostgreSQL 服务器：

```
mydb=> \g
```

这告诉服务器处理该查询。如果你用分号终止查询，`\g` 就不是必需的。psql 会自动处理以分号终止的查询。要从文件（例如 `myFile`）中读取查询而不是交互式地输入，输入：

```
mydb=> \i myFile
```

要退出 psql 并返回 Unix，输入

```
mydb=> \q
```

然后 psql 将退出并把你返回到命令 shell。（关于更多转义码，在 `psql` 提示符下输入 `\?`。）空白（即空格、制表符和换行符）可以在 SQL 查询中自由使用。单行注释用 `--` 表示。连字符之后直到行尾的所有内容都被忽略。多行注释以及行内注释用 `/* ... */` 表示。

10.3. 删除数据库

如果你是数据库 mydb 的拥有者，你可以使用 SQL 命令

```
=> DROP DATABASE mydb;
```

或 Unix shell 脚本

```
% dropdb mydb
```

销毁它。这个动作会物理地删除与该数据库关联的所有 Unix 文件，而且无法撤销，因此只应在深思熟虑之后才进行。

第 11 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。

11.1. 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，例如在 SORT 节点中完成排序所需的时间）。
- 估计总代价（如果所有元组都被检索；不过它们也可能不被检索——例如带 LIMIT 的查询会在付出全部代价之前停止，举例来说）。
- 该计划节点输出的估计行数（同样，不考虑任何 LIMIT）。
- 该计划节点输出元组的估计平均宽度（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，可以看管理员指南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，
它没有考虑将结果元组传输给前端所消耗的时间——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元组集。）

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 条件子句的估计选择率。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 7.2 开发版源码）：

```
regression=# EXPLAIN SELECT * FROM tenk1;
NOTICE:  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，添加一个限定子句：

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..358.00 rows=1007 width=148)
```

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 个元组，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。

进一步收紧限定条件：

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50;
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..181.09 rows=49
 width=148)
```

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 个元组，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。

再向限定条件添加另一个条件：

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50 AND
regression-# stringu1 = 'xxx';
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..181.22 rows=1
 width=148)
```

新增的子句stringu1 = 'xxx'降低了估计输出行数，但没有降低代价，因为仍然必须访问同一组元组。

使用之前讨论的字段，尝试连接两个表：

```
regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1
 < 50
regression-# AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..330.41 rows=49 width=296)
  ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..181.09 rows=49 width=148)
  ->  Index Scan using tenk2_unique2 on tenk2 t2
              (cost=0.00..3.01 rows=1 width=148)
```

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unique1 < 50这个 WHERE 子句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，当前外层扫描元组的 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，再加上每个外层元组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，循环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关系的 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 11.3 节。）

```
regression=# set enable_nestloop = off;
SET VARIABLE
regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1
 < 50
regression-# AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Hash Join  (cost=181.22..564.83 rows=49 width=296)
  ->  Seq Scan on tenk2 t2
              (cost=0.00..333.00 rows=10000 width=148)
  ->  Hash  (cost=181.09..181.09 rows=49 width=148)
        ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..181.09 rows=49 width=148)
```

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，对每一个tenk2元组在哈希表中探测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之前不会有任何元组输出。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 NOT 按 10000 乘以 181.09 来计费；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（runtime），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：

```
regression=# EXPLAIN ANALYZE
regression-# SELECT * FROM tenk1 t1, tenk2 t2
regression-# WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..330.41 rows=49 width=296) (actual time=1.31..
28.90 rows=50 loops=1)
  ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..181.09 rows=49 width=148) (actual time=0.
69..8.84 rows=50 loops=1)
  ->  Index Scan using tenk2_unique2 on tenk2 t2
              (cost=0.00..3.01 rows=1 width=148) (actual time=0.28..
0.31 rows=1 loops=50)
Total runtime: 30.67 msec
```

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个元组执行一次。
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，以及处理结果元组所花费的时间。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 查询，总运行时间可能大得多，因为它包括了处理输出元组所花费的时间。在这些查询中，
顶层计划节点的时间本质上就是计算新元组和/或定位旧元组所花费的时间，
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。

11.2. 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pg_class的reltuples和relpages列中。可以使用类似下面的查询来查看：

```
regression=# select relname, relkind, reltuples, relpages from pg_
class
regression-# 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 条件中各个子句的行所占比例。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：

```
regression=# 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)
regression=#
```

从PostgreSQL 7.2 开始，pg_stats中存在下列列：

表 11.1. pg_stats 的列

名字	类型	描述
tablename	name	包含该列的表的名字
attname	name	此行描述的列
null_frac	real	列中为 NULL 的项所占比例
avg_width	integer	列的项的平均宽度（字节）
n_distinct	real	如果大于零，是列中不同值的估计数量。如果小于零，是不同值数量除以行数的负值。（当 ANALYZE 认为不同值的数量可能随表的增长而增加时，使用取负形式；当列似乎有固定数量的可能值时，使用正形式。）例如，-1 表示一个唯一列，其不同值的数量与行数相同。
most_common_vals	text[]	列中最常见值的列表。（如果看起来没有任何值比其他值更常见，则省略。）
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。

名字	类型	描述
histogram_bounds	text[]	把列的值划分成数量大致相等的组的值列表。most_common_vals（如果存在）会被排除在直方图计算之外。（如果列的数据类型没有<操作符，或者most_common_vals列表占了整个总体，则省略。）
correlation	real	物理行顺序与列值逻辑顺序之间的统计相关性。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，因为减少了磁盘的随机访问。（如果列的数据类型没有<操作符，则省略。）

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。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。

11.3. 用显式 JOIN 控制规划器

从PostgreSQL 7.1 开始，就可以用显式 JOIN 语法在一定程度上控制查询规划器。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 子句 a.id = b.id 将 A 连接到 B，然后再利用另一个 WHERE 子句把 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，
管理员指南中有描述。）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。

在PostgreSQL 7.1 中，规划器把所有显式 JOIN 语法都当作带有连接顺序约束来处理，即使对内连接来说这种约束在逻辑上并不必要。因此，尽管下面这些查询给出的结果相同：

```
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);
```

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。

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 列表中使用 JOIN 操作符。例如，

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

这会强制规划器先将 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。

如果在一个复杂查询中混合使用外连接和内连接，
你可能不希望约束规划器在外连接内部对内连接的良好顺序的搜索。在 JOIN 语法中无法直接做到这一点，但可以通过使用子查询绕过这一语法限制。例如，

```
SELECT * FROM d LEFT JOIN
        (SELECT * FROM a, b, c WHERE ...) AS ss
        ON (...);
```

这里，连接 D 必须是查询计划的最后一步，但规划器可以自由地考虑 A,B,C 的各种连接顺序。

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。

11.4. 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。

11.4.1. 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就必须为每条记录的加入执行大量工作。

11.4.2. 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。

11.4.3. 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DROP INDEX、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。

11.4.4. 之后的 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。

附录 A. 日期/时间支持

PostgreSQL 对所有日期/时间支持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。

A.1. 日期/时间关键字

表 A.1. 月份缩写

月份	缩写
April	Apr
August	Aug
December	Dec
February	Feb
January	Jan
July	Jul
June	Jun
March	Mar
November	Nov
October	Oct
September	Sep, Sept

> **注意**
>
> 出于显而易见的原因，月份 `May` 没有缩写。

表 A.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
Saturday	Sat

表 A.3. PostgreSQL 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

标识符	描述
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，无效的 `ABSTIME` 字段会被输出为 “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。

A.2. 时区

PostgreSQL 内部包含用于时区解码的表格式信息，因为没有标准的 *nix 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。

下表列出了 PostgreSQL 识别的时区，它按与 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 A.4. PostgreSQL 识别的时区

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，苏联 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，苏联 8 区
KST	+09:00	韩国标准时间
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

时区	与 UTC 的偏移	描述
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
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
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，苏联 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
MEWT	+01:00	中欧冬令时间
MEZ	+01:00	中欧时区
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

时区	与 UTC 的偏移	描述
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
WAT	-01:00	西非时间
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）
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
ACT	-05:00	Atlantic/Porto Acre 标准时间
EDT	-04:00	美国东部夏令时间
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
AHST	-10:00	阿拉斯加-夏威夷标准时间
HST	-10:00	夏威夷标准时间
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

A.2.1. 澳大利亚时区

澳大利亚时区及其命名变体占了 PostgreSQL 时区查找表中整整四分之一的条目。它们与美国常用的时区存在两处命名冲突：`CST` 和 `EST`。

如果设置了运行时选项 `AUSTRALIAN_TIMEZONES`，那么 `CST`、`EST` 和 `SAT` 将被解释为澳大利亚时区名。不设置该选项时，`CST` 和 `EST` 会被当作美国时区名，而 `SAT` 会被解释为一个表示 `Saturday` 的噪声词。

表 A.5. PostgreSQL 澳大利亚时区

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

A.2.2. 日期/时间输入解释

日期/时间类型都使用一组公共的例程进行解码。

日期/时间输入解释

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。
    其后的所有数字和冒号都包括在内。

    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. 如果词元是纯数字，那么它要么是一个单个字段，要么是一个 ISO-8601 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，那么它要么是一个时区，
    要么是一个特殊字段。

2. 如果词元是一个文本字符串，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. 如果没有找到，则抛出错误。

3. 该词元是一个数字或数字字段。

    a. 如果多于 4 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`）。8 位和 6 位数字被解释为年、月、日，而 7 位和 5 位数字则分别被解释为年、年积日。

    b. 如果词元是三位数字并且已经解码了年份，则解释为年积日。

    c. 如果是四位或六位数字并且已经读取了年份，则解释为时间。

    d. 如果是四位或更多位数字，则解释为年份。

    e. 如果处于欧洲日期模式，并且还没有读取日字段，并且该值小于或等于 31，则解释为日。

f. 如果还没有读取月份字段，并且该值小于或等于 12，则解释为月。

g. 如果还没有读取日字段，并且该值小于或等于 31，则解释为日。

h. 如果是两位数字或四位及更多位数字，则解释为年份。

i. 否则，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，`1BC` 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 4 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 4 位数字输入（例如，0099 表示公元 99 年）。以前的 PostgreSQL 版本接受三位数字和单个数字的年份，但从 7.0 版开始，规则已被收紧以减少歧义的可能性。

A.3. 历法的历史

> **注意**
>
> 由 José Soares（`<jose@sferacarta.com>`）贡献

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”（Julian Day）与“儒略积日”（Julian Date）并不相同。儒略历由 Julius Caesar 于公元前 45 年引入。直到 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，回归年相对于格里高利历大约每 3300 年偏移一天。

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：

能被 4 整除的年份是闰年。
但是，能被 100 整除的年份不是闰年。
但是，能被 400 整除的年份仍是闰年。

因此，1700、1800、1900、2100 和 2200 年不是闰年，而 1600、2000 和 2400 年是闰年。相比之下，在较老的儒略历中只有能被 4 整除的年份是闰年。

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。意大利、波兰、葡萄牙和西班牙执行了这一规定。其他天主教国家随后不久也照办，

但新教国家不愿改变，而希腊正教国家直到本世纪初才改历。
大不列颠及其领地（包括现在的美国）在 1752 年执行了这一改革。因此 1752 年 9 月 2 日之后紧跟着的是 1752 年 9 月 14 日。这就是 Unix 系统的 cal 会输出下面内容的原因：

```
% 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**
>
> SQL92 指出，“在‘日期时间字面量’的定义中，‘日期时间值’受格里高利历下日期和时间的自然规则约束”。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 B. SQL 关键字

表 B.1列出了在 SQL 标准以及 PostgreSQL 7.2.8 中被视为关键字的所有词元。背景信息可参见第 1.1.1 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 B.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 B.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 B.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	
AS	保留	保留	保留
ASC	保留	保留	保留
ASENSITIVE		非保留	
ASSERTION		保留	保留
ASSIGNMENT		非保留	
ASYMMETRIC		非保留	
AT	非保留	保留	保留
ATOMIC		非保留	
AUTHORIZATION	非保留	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
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		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	
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
CATALOG		保留	保留
CATALOG_NAME		非保留	非保留

关键字	PostgreSQL	SQL 99	SQL 92
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		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
CONNECTION		保留	保留
CONNECTION_NAME		非保留	非保留

关键字	PostgreSQL	SQL 99	SQL 92
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
CONVERT		非保留	保留
COPY	非保留		
CORRESPONDING		保留	保留
COUNT		非保留	保留
CREATE	非保留	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		保留	保留
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

关键字	PostgreSQL	SQL 99	SQL 92
DEFAULT	保留	保留	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	
DELETE	非保留	保留	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		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
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
EXCLUSIVE	非保留		

关键字	PostgreSQL	SQL 99	SQL 92
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		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		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	非保留	保留	保留
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	

关键字	PostgreSQL	SQL 99	SQL 92
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
IMPLEMENTATION		非保留	
IN	保留（可用作函数）	保留	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		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		保留	保留
INTEGER		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
INVOKER		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		

关键字	PostgreSQL	SQL 99	SQL 92
LANGUAGE	非保留	保留	保留
LARGE		保留	
LAST		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	
LOCALTIMESTAMP		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
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

关键字	PostgreSQL	SQL 99	SQL 92
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
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	

关键字	PostgreSQL	SQL 99	SQL 92
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	
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		
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		保留	保留
PRESERVE		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		

关键字	PostgreSQL	SQL 99	SQL 92
PROCEDURE	非保留	保留	保留
PUBLIC	保留（可用作函数）	保留	保留
READ	非保留	保留	保留
READS		保留	
REAL		保留	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		
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
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	保留	
ROWS		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

关键字	PostgreSQL	SQL 99	SQL 92
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	
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		非保留	
SIMPLE		非保留	
SIZE		保留	保留
SMALLINT		保留	保留
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	
START	非保留	保留	
STATE		保留	
STATEMENT	非保留	保留	
STATIC		保留	

关键字	PostgreSQL	SQL 99	SQL 92
STATISTICS	非保留		
STDIN	非保留		
STDOUT	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	
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		保留	

关键字	PostgreSQL	SQL 99	SQL 92
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		
UPDATE	非保留	保留	保留
UPPER		非保留	保留
USAGE		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		
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		

关键字	PostgreSQL	SQL 99	SQL 92
VIEW	非保留	保留	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		保留	保留
YEAR	非保留	保留	保留
ZONE	非保留	保留	保留

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

PostgreSQL 7.2.8 管理员指南

The PostgreSQL Global Development Group

PostgreSQL 7.2.8 管理员指南

The PostgreSQL Global Development Group

法律声明

目录

1. 安装说明 1
    1.1. 简短版本 1
    1.2. 需求 1
    1.3. 获取源代码 2
    1.4. 如果你是在升级 2
    1.5. 安装过程 3
    1.6. 安装后设置 9
        1.6.1. 共享库 9
        1.6.2. 环境变量 9
    1.7. 受支持的平台 10
2. 在 Windows 上安装 15
3. 服务器运行时环境 16
    3.1. PostgreSQL 用户账户 16
    3.2. 创建数据库集簇 16
    3.3. 启动数据库服务器 17
        3.3.1. 服务器启动失败 18
        3.3.2. 客户端连接问题 19
    3.4. 运行时配置 19
        3.4.1. 规划器和优化器调整 20
        3.4.2. 日志和调试 22
        3.4.3. 常规操作 24
        3.4.4. WAL 27
        3.4.5. 短选项 28
    3.5. 管理内核资源 29
        3.5.1. 共享内存和信号量 29
        3.5.2. 资源限制 33
    3.6. 关闭服务器 33
    3.7. 使用 SSL 的安全 TCP/IP 连接 34
    3.8. 使用 SSH 隧道的安全 TCP/IP 连接 35
4. 客户端认证 36
    4.1. `pg_hba.conf` 文件 36
    4.2. 认证方法 40
        4.2.1. trust 认证 40
        4.2.2. 密码认证 40
        4.2.3. Kerberos 认证 41
        4.2.4. 基于 ident 的认证 42
    4.3. 认证问题 43
5. 本地化 44
    5.1. 区域支持 44
        5.1.1. 概述 44
        5.1.2. 优点 45
        5.1.3. 问题 45
    5.2. 多字节支持 46
        5.2.1. 启用多字节支持 46
        5.2.2. 设置编码 47
        5.2.3. 服务器和客户端之间的自动编码翻译 48
        5.2.4. 关于 Unicode 50
        5.2.5. 如果无法完成转换会怎样? 50
        5.2.6. 参考资料 50
        5.2.7. 历史 50
        5.2.8. Windows/ODBC 上的 WIN1250 52

5.3. 单字节字符集重编码 52
6. 管理数据库 54
6.1. 创建数据库 54
6.1.1. 模板数据库 54
6.1.2. 备选位置 55
6.2. 删除数据库 56
7. 数据库用户和权限 57
7.1. 数据库用户 57
7.1.1. 用户属性 57
7.2. 组 57
7.3. 权限 58
7.4. 函数和触发器 58
8. 日常数据库维护任务 59
8.1. 一般性讨论 59
8.2. 例行清理 59
8.2.1. 回收磁盘空间 59
8.2.2. 更新规划器统计信息 60
8.2.3. 防止事务 ID 回卷故障 60
8.3. 日志文件维护 62
9. 备份与恢复 63
9.1. SQL 转储 63
9.1.1. 恢复转储 63
9.1.2. 使用pg_dumpall 64
9.1.3. 大型数据库 64
9.1.4. 注意事项 65
9.2. 文件系统级别的备份 65
9.3. 版本间的迁移 66
10. 监控数据库活动 67
10.1. 标准 Unix 工具 67
10.2. 统计收集器 67
10.2.1. 统计信息收集配置 68
10.2.2. 查看收集到的统计信息 68
11. 预写式日志（WAL） 71
11.1. 总体描述 71
11.1.1. WAL 的直接收益 71
11.1.2. 未来的收益 71
11.2. 实现 72
11.2.1. 用 WAL 进行数据库恢复 72
11.3. WAL 配置 72
12. 磁盘存储 74
13. 数据库故障 75
13.1. 磁盘已满 75
13.2. 磁盘故障 75
14. 回归测试 76
14.1. 简介 76
14.2. 运行测试 76
14.3. 测试结果评估 77
14.3.1. 错误消息差异 77
14.3.2. Locale 差异 77
14.3.3. 日期和时间差异 78
14.3.4. 浮点差异 78
14.3.5. 多边形差异 78
14.3.6. 行顺序差异 78

14.3.7. “random” 测试 79
14.4. 平台相关的比较文件 79
A. 发布说明 80
A.1. 版本 7.2.8 80
A.1.1. 迁移到版本 7.2.8 80
A.1.2. 变更 80
A.2. 版本 7.2.7 80
A.2.1. 迁移到版本 7.2.7 80
A.2.2. 变更 81
A.3. 版本 7.2.6 81
A.3.1. 迁移到版本 7.2.6 81
A.3.2. 变更 81
A.4. 版本 7.2.5 82
A.4.1. 迁移到版本 7.2.5 82
A.4.2. 变更 82
A.5. 版本 7.2.4 82
A.5.1. 迁移到版本 7.2.4 82
A.5.2. 变更 82
A.6. 版本 7.2.3 83
A.6.1. 迁移到版本 7.2.3 83
A.6.2. 变更 83
A.7. 版本 7.2.2 83
A.7.1. 迁移到版本 7.2.2 83
A.7.2. 变更 84
A.8. 版本 7.2.1 84
A.8.1. 迁移到版本 7.2.1 84
A.8.2. 变更 84
A.9. 版本 7.2 85
A.9.1. 概述 85
A.9.2. 迁移到版本 7.2 86
A.9.3. 变更 86
A.10. 版本 7.1.3 96
A.10.1. 迁移到版本 7.1.3 96
A.10.2. 变更 96
A.11. 版本 7.1.2 96
A.11.1. 迁移到版本 7.1.2 96
A.11.2. 变更 96
A.12. 版本 7.1.1 97
A.12.1. 迁移到版本 7.1.1 97
A.12.2. 变更 97
A.13. 版本 7.1 97
A.13.1. 迁移到版本 7.1 98
A.13.2. 变更 98
A.14. 版本 7.0.3 101
A.14.1. 迁移到版本 7.0.3 102
A.14.2. 变更 102
A.15. 版本 7.0.2 102
A.15.1. 迁移到版本 7.0.2 103
A.15.2. 变更 103
A.16. 版本 7.0.1 103
A.16.1. 迁移到版本 7.0.1 103
A.16.2. 变更 103
A.17. 版本 7.0 104
A.17.1. 迁移到版本 7.0 104

A.17.2. 变更 105
A.18. 版本 6.5.3 111
A.18.1. 迁移到版本 6.5.3 111
A.18.2. 变更 111
A.19. 版本 6.5.2 111
A.19.1. 迁移到版本 6.5.2 111
A.19.2. 变更 111
A.20. 版本 6.5.1 112
A.20.1. 迁移到版本 6.5.1 112
A.20.2. 变更 112
A.21. 版本 6.5 113
A.21.1. 迁移到版本 6.5 114
A.21.2. 变更 114
A.22. 版本 6.4.2 117
A.22.1. 迁移到版本 6.4.2 118
A.22.2. 变更 118
A.23. 版本 6.4.1 118
A.23.1. 迁移到版本 6.4.1 118
A.23.2. 变更 118
A.24. 版本 6.4 119
A.24.1. 迁移到版本 6.4 119
A.24.2. 变更 119
A.25. 版本 6.3.2 123
A.25.1. 变更 123
A.26. 版本 6.3.1 124
A.26.1. 变更 124
A.27. 版本 6.3 125
A.27.1. 迁移到版本 6.3 126
A.27.2. 变更 126
A.28. 版本 6.2.1 129
A.28.1. 从版本 6.2 迁移到版本 6.2.1 129
A.28.2. 变更 130
A.29. 版本 6.2 130
A.29.1. 从版本 6.1 迁移到版本 6.2 130
A.29.2. 从版本 1.x 迁移到版本 6.2 130
A.29.3. 变更 130
A.30. 版本 6.1.1 132
A.30.1. 从版本 6.1 迁移到版本 6.1.1 133
A.30.2. 变更 133
A.31. 版本 6.1 133
A.31.1. 迁移到版本 6.1 134
A.31.2. 变更 134
A.32. 版本 6.0 136
A.32.1. 从版本 1.09 迁移到版本 6.0 136
A.32.2. 从 1.09 之前版本迁移到版本 6.0 136
A.32.3. 变更 136
A.33. 版本 1.09 138
A.34. 版本 1.02 138
A.34.1. 从版本 1.02 迁移到版本 1.02.1 138
A.34.2. 转储/重载程序 139
A.34.3. 变更 139
A.35. 版本 1.01 140
A.35.1. 从版本 1.0 迁移到版本 1.01 140

A.35.2. 变更 142
A.36. 版本 1.0 142
A.36.1. 变更 142
A.37. Postgres95 版本 0.03 143
A.37.1. 变更 143
A.38. Postgres95 版本 0.02 145
A.38.1. 变更 145
A.39. Postgres95 版本 0.01 146

表格清单

3.1. 短选项对照 28
3.2. System V IPC参数 29
5.1. 编码 46
5.2. 通信编码 48
10.1. 标准统计视图 68
10.2. 统计访问函数 69

范例清单

4.1. 一个 pg_hba.conf 文件示例 38
4.2. 一个 pg_ident.conf 文件示例 42

第 1 章 安装说明

1.1. 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。

1.2. 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 1.7 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 需要满足以下先决条件：

- 必须使用 GNU make；其他 make 程序不能用。GNU make 常以 `gmake` 的名称安装；本文档将始终用该名称指代它。（在某些系统上 GNU make 是名为 `make` 的默认工具。）要测试 GNU make，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 来解包发行版。如果你正在阅读本文档，你很可能已经跨过了这道坎。

- GNU Readline 库（用于舒适的行编辑和命令历史检索）如果被找到就会自动使用。你可能希望在继续之前安装它，但它不是必需的。（在 NetBSD 上，`libedit` 库与 readline 兼容，找不到 `libreadline` 时会使用它。）

- 从头构建需要 GNU Flex 和 Bison，但从发行的源码包构建时不需要它们，因为发行包中包含了预生成的输出文件。只有从 CVS 树构建或修改了实际的扫描器和解析器定义文件时才需要这些程序。如果需要它们，务必使用 Flex 2.5.4 或更高版本以及 Bison 1.28 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。

- 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。

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 30 MB，安装目录大约需要 10 MB。空的数据库集簇约占 20 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，临时还需要额外的 20 MB。用 `df` 命令检查磁盘空间。

1.3. 获取源代码

PostgreSQL 7.2.8 的源码可以通过匿名 FTP 从ftp://ftp.postgresql.org/pub/postgresql-7.2.8.tar.gz获取。可能的话请使用镜像站点。取得文件之后，解包：

```
gunzip postgresql-7.2.8.tar.gztar xf postgresql-7.2.8.tar
```

这会创建目录 `postgresql-7.2.8`（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。

1.4. 如果你是在升级

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“7.2.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. 要转储你的数据库安装，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 9.1.4 节。

   务必使用你当前运行版本中的 `pg_dumpall` 命令。7.2.8 的 `pg_dumpall` 不能用于较旧的数据库。

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`
   ```

   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 7.2.8 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。

你也可以把新版本与旧版本并行安装以减少停机时间。这些主题在第 9.3 节中有详细讨论，无论如何都建议你阅读。

1.5. 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

   此脚本会运行一系列测试来猜测各种系统相关变量的值，检测你的操作系统的一些特性，最后在构建树中创建若干文件记录其发现。

   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**
>
> 我们已设法使 PostgreSQL 可以安装到共享的安装位置（例如 /usr/local/include）而不干扰系统其余部分的名字空间。首先，
> 除非完全展开的目录名已经包含字符串“postgres”或“pgsql”，否则字符串“/postgresql”会自动追加到 datadir、sysconfdir 和 docdir。例如，如果你选择 /usr/local 作为前缀，文档将安装在 /usr/local/doc/postgresql；但如果前缀是 /opt/postgres，则会安装在 /opt/postgres/doc。其次，7.2 版重新组织了 C 和 C++ 头文件的安装布局。客户端接口的公共头文件安装到 includedir 且名字空间是干净的。内部头文件和服务器头文件安装到 includedir 下的私有目录。如何获取各接口的头文件，参见程序员指南。最后，如果合适，还会在 libdir 下为动态可装载模块创建一个私有子目录。

--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-locale`

启用区域设置支持。区域设置支持会带来性能损失，但如果你不在英语环境中，你很可能需要它。

`--enable-recode`

启用单字节字符集 recode 支持。关于此特性，参见第 5.3 节。

`--enable-multibyte`

允许使用多字节字符编码（包括 Unicode）和字符集编码转换。细节见第 5.2 节。

注意，某些接口（如 Tcl 或 Java）期望所有字符串都是 Unicode，因此需要此选项才能正确支持这些接口。

`--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，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下载附加软件包：http://www.postgresql.org/~petere/gettext.html。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。

`--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-CXX`

构建 C++ 接口库。

`--with-perl`

构建 Perl 接口模块。Perl 接口将安装在 Perl 模块的惯常位置（通常在 `/usr/lib/perl` 之下），因此你必须有 root 权限才能执行安装步骤（见步骤 4）。使用此选项需要已安装 Perl 5。

`--with-python`

构建 Python 接口模块。你需要 root 权限才能把 Python 模块安装到其默认位置（`/usr/lib/python`*`x.y`*）。要使用此选项，你必须已安装 Python，而且你的系统需要支持共享库。如果你想构建一个新的完整解释器二进制，就得手动做了。

`--with-tcl`

构建需要 Tcl/Tk 的组件，即 libpgtcl、pgtclsh、pgtksh、PgAccess 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 和 PgAccess）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。

`--enable-odbc`

构建 ODBC 驱动。默认情况下该驱动独立于驱动管理器。要与你的系统上已安装的驱动管理器更好地配合，请在此选项之外再使用下面选项之一。更多信息见程序员指南。

`--with-iodbc`

构建供 iODBC 使用的 ODBC 驱动。

`--with-unixodbc`

构建供 unixODBC 使用的 ODBC 驱动。

`--with-odbcinst=`*`DIRECTORY`*

指定 ODBC 驱动查找其 `odbcinst.ini` 配置文件的目录。默认为 `/usr/local/pgsql/etc` 或你通过 `--sysconfdir` 指定的值。应当让驱动与驱动管理器读取同一个文件。

如果使用了 `--with-iodbc` 或 `--with-unixodbc` 选项，此选项将被忽略，因为那时驱动管理器会处理配置文件的位置。

`--with-java`

构建 JDBC 驱动及相关的 Java 软件包。此选项要求已安装 Ant（当然还需要 JDK）。更多信息参见程序员指南中的 JDBC 驱动文档。

`--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`

构建时加入对 PAM（可插拔认证模块）的支持。

`--enable-syslog`

使 PostgreSQL 服务器能够使用 syslog 日志设施。（使用此选项并不意味着你必须用 syslog 记录日志，甚至也不是默认就会这样做；它只是使运行时打开该选项成为可能。）

`--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。

如果你希望使用与 `configure` 所选不同的 C 或 C++ 编译器，可以分别将环境变量 `CC` 或 `CXX` 设置为你选择的程序。类似地，你可以用 `CFLAGS` 和 `CXXFLAGS` 变量覆盖默认的编译器标志。例如：

```
env CC=/opt/bin/gcc CFLAGS='-O2 -pipe' ./configure
```

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
   ```

   （这不能以 root 身份运行；请以非特权用户身份执行。）由于错误消息措辞或浮点结果的差异，可能有些测试会失败。有关解释测试结果的详细信息，见第 14 章。以后任何时候都可以用同一命令重复此测试。

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 1.4 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。

   如果你构建了 Perl 或 Python 接口，而在执行上述命令时你不是 root 用户，那么安装的这一部分很可能失败了。这种情况下你应当成为 root 用户然后执行

   ```
   gmake -C src/interfaces/perl5 installgmake -C src/interfaces/
   python install
   ```

   如果你没有超级用户权限，
   那就只能自己想办法了：你仍然可以拿走所需的文件并把它们放到 Perl 或 Python 能找到的其他目录中，但具体怎么做就留作练习了。

   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
   ```

   要撤销安装，使用命令 `gmake uninstall`。但是，这不会删除任何已创建的目录。

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。

如果你完成了一次构建之后才发现 configure 选项有误，或者你更改了 configure 所探测的任何内容（例如安装了 GNU Readline），那么最好在重新配置和重新构建之前先执行 `gmake distclean`。否则，你对配置选项的更改可能不会传播到所有需要它的地方。

1.6. 安装后设置

1.6.1. 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就设置好环境变量 `LD_RUN_PATH`。

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。

1.6.2. 环境变量

如果你安装到了 `/usr/local/pgsql` 或其他默认不会搜索程序的位置，就需要把 `/usr/local/pgsql/bin`（或你在步骤 1中为 `--bindir` 设置的任何值）加入你的 `PATH`。为此，请

把下面的内容加入你的 shell 启动文件，例如 `~/.bash_profile`（如果你想让每个用户都受影响，则用 `/etc/profile`）：

```
PATH=/usr/local/pgsql/bin:$PATH
```

如果你使用的是 `csh` 或 `tcsh`，则使用此命令：

```
set path = ( /usr/local/pgsql/bin $path )
```

要让你的系统能找到 man 文档，你需要在某个 shell 启动文件中加入类似下面的一行：

```
MANPATH=/usr/local/pgsql/man:$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。

1.7. 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> 或 <pgsql-ports@postgresql.org>，而不是写给这里列出的人。

OS	处理器	版本	报告者	备注
AIX	RS6000	7.2	2001-12-19, Andreas Zeugswetter (<ZeugswetterA@sparda t.at>), Tatsuo Ishii (<t-ishii@ sra.co.jp>)	参见 doc/FAQ_AIX
BeOS	x86	7.2	2001-11-29, Cyril Velter (<cyril. velter@libert ysurf.fr>)	5.0.4
BSD/OS	x86	7.2	2001-11-27, Bruce Momjian (<pgman@candle. pha.pa.us>)	4.2
FreeBSD	Alpha	7.2	2001-12-18, Chris Kings-Lynne (<chriskl@ familyhealth. com.au>)	
FreeBSD	x86	7.2	2001-11-14, Chris Kings-Lynne (<chriskl@	

OS	处理器	版本	报告者	备注
			familyhealth.com.au>)	
HP-UX	PA-RISC	7.2	2001-11-29, Joseph Conway (<Joseph.Conway@home.com>), Tom Lane (<tgl@sss.pgh.pa.us>)	11.00 和 10.20；另请参阅 doc/FAQ_HPUX
IRIX	MIPS	7.2	2001-11-28, Luis Amigo (<lamigo@atc.unican.es>)	6.5.13, MIPSPro 7.30
Linux	Alpha	7.2	2001-11-16, Tom Lane (<tgl@sss.pgh.pa.us>)	2.2.18；在 SourceForge 测试
Linux	armv4l	7.2	2001-12-10, Mark Knox (<segfault@hardline.org>)	2.2.x
Linux	MIPS	7.2	2001-11-15, Hisao Shibuya (<shibuya@alpha.or.jp>)	2.0.x；Cobalt Qube2
Linux	PlayStation 2	7.2	2001-12-12, Permaine Cheung <pcheung@redhat.com>)	#undef HAS_TEST_AND_SET、slock_t
Linux	PPC74xx	7.2	2001-11-16, Tom Lane (<tgl@sss.pgh.pa.us>)	2.2.18；Apple G3
Linux	S/390	7.2	2001-12-12, Permaine Cheung <pcheung@redhat.com>)	
Linux	Sparc	7.2	2001-11-28, Doug McNaught (<doug@wirebo ard.com>)	2.2.19
Linux	x86	7.2	2001-11-15, Thomas Lockhart (<lockhart@fourpalms.org>)	2.0.x、2.2.x、2.4.x
MacOS X	PPC	7.2	2001-11-28, Gavin Sherry (<swm@linuxworld.com.au>)	10.1.x
NetBSD	Alpha	7.2	2001-11-20, Thomas Thai (<tom@minnesota.com>)	1.5W
NetBSD	arm32	7.1	2001-03-21, Patrick Welche (	1.5E

OS	处理器	版本	报告者	备注
			<prlw1@cam.ac.uk>)	
NetBSD	m68k	7.0	2000-04-10, Henry B. Hotz (<hotz@jpl.nasa.gov>)	Mac 8xx
NetBSD	PPC	7.2	2001-11-28, Bill Studenmund (<wrstuden@netbsd.org>)	1.5
NetBSD	Sparc	7.2	2001-12-03, Matthew Green (<mrg@eterna.com.au>)	32 位和 64 位构建
NetBSD	VAX	7.1	2001-03-30, Tom I. Helbekkmo (<tih@kpnQwest.no>)	1.5
NetBSD	x86	7.2	2001-11-28, Bill Studenmund (<wrstuden@netbsd.org>)	1.5
OpenBSD	Sparc	7.2	2001-11-27, Brandon Palmer (<bpalmer@crimelabs.net>)	3.0
OpenBSD	x86	7.2	2001-11-26, Brandon Palmer (<bpalmer@crimelabs.net>)	3.0
Open UNIX	x86	7.2	2001-11-28, OU-8 Larry Rosenman (<ler@lerctr.org>), UW-7 Olivier Prenant (<ohp@pyrenet.fr>)	另请参阅 doc/FAQ_SCO
QNX 4 RTOS	x86	7.2	2001-12-10, Bernd Tegge (<tegge@repas-aeg.de>)	4.25；另请参阅 doc/FAQ_QNX4
Solaris	Sparc	7.2	2001-11-12, Andrew Sullivan (<andrew@libertyrms.com>)	2.6-8；另请参阅 doc/FAQ_Solaris
Solaris	x86	7.2	2001-11-28, Martin Renters (<martin@datafax.com>)	2.8；另请参阅 doc/FAQ_Solaris
SunOS 4	Sparc	7.2	2001-12-04, Tatsuo Ishii (<t-	

OS	处理器	版本	报告者	备注
			ishii@sra.co.jp>)	
Tru64 UNIX	Alpha	7.2	2001-11-26, Alessio Bragadini (<alessio@albourne.com>), Bernd Tegge (<tegge@repas-aeg.de>)	5.0；4.0g 用 cc 和 gcc
Windows	x86	7.2	2001-12-13, Dave Page (<dpage@vale-housing.co.uk>), Jason Tishler (<jason@tishler.net>)	带 Cygwin；见 doc/FAQ_MSWIN
Windows	x86	7.2	2001-12-10, Dave Page (<dpage@vale-housing.co.uk>)	本机方式只有客户端；见第 2 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 7.2 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	报告者	备注
DG/UX 5.4R4.11	m88k	6.3	1998-03-01, Brian E Gallew (<geek+@cmu.edu>)	近期没有报告
MkLinux DR1	PPC750	7.0	2001-04-03, Tatsuo Ishii (<t-ishii@sra.co.jp>)	7.1 需要操作系统更新?
NeXTSTEP	x86	6.x	1998-03-01, David Wetzel (<dave@turbocat.de>)	疑似代码老化
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 5	x86	6.5	1999-05-25, Andrew Merrill (<andrew@compclass.com>)	7.2.8 应该可以工作，但没有相关报告；另请参阅 doc/FAQ_SCO
System V R4	m88k	6.2.1	1998-03-01, Doug Winterburn (<dlw@seavme.xroads.com>)	需要新的 TAS 自旋锁代码
System V R4	MIPS	6.4	1998-10-28, Frank Ridderbusch (<ridderbusch.pad@sni.de>)	近期没有报告

OS	处理器	版本	报告者	备注
Ultrix	MIPS	7.1	2001-03-26	未检测到 TAS 自旋锁代码
Ultrix	VAX	6.x	1998-03-01	

第 2 章 在 Windows 上安装

在 Windows 上构建、安装和使用 PostgreSQL 客户端库的说明

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 1 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。

要使用这些库，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。）

第 3 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。

3.1. PostgreSQL 用户账户

与任何连接到外部世界的服务器守护进程一样，建议在单独的用户账户下运行 PostgreSQL。该用户账户应只拥有服务器所管理的数据本身，不应与其他守护进程共用。（因此，使用“nobody”用户是个坏主意。）不建议把可执行文件安装为由该用户账户所有，因为那样会有用户自定义函数行为失常或任何其他利用手段破坏可执行程序的风险。

要向系统添加用户账户，请查找useradd或 adduser命令。用户名 postgres 很常用，但并非必须。

3.2. 创建数据库集簇

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 标准则说目录集簇。）数据库集簇是一组可通过单个运行中的数据库服务器实例访问的数据库。初始化之后，数据库集簇会包含一个名为 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。

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，并将其所有权转移给 PostgreSQL 用户账户。过程可能如下：

```
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属于一个已经初始化过的安装，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，必须妥善保护它免遭未经授权的访问。因此，initdb 会撤销除 PostgreSQL 用户账户以外所有人的访问权限。

不过，尽管目录内容是安全的，pg_hba.conf 默认的 trust 认证方法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，把 trust 认证改为 md5 或 password。（其他可能更方便的方法包括使用 ident 认证或文件系统权限来限制连接。更多信息见第 4 章。）

运行 initdb 时你可能遇到的一个意外是类似这样的通知：

```
NOTICE:  Initializing database with en_US collation order.
        This locale setting will prevent use of index optimization for
        LIKE and regexp searches.  If you are concerned about speed of
        such queries, you may wish to set LC_COLLATE to "C" and
        re-initdb.  For more information see the Administrator's
 Guide.
```

此通知是为了警告你，当前选择的区域设置会导致索引按某种顺序排序，使它们无法用于 LIKE 和正则表达式搜索。如果你需要这类搜索的良好性能，应当把当前区域设置改为 C 并重新运行 initdb。在大多数系统上，设置当前区域设置的方法是更改环境变量 LC_ALL 或 LANG 的值。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，现在就作出正确选择很重要。

3.3. 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。数据库服务器称为 postmaster。postmaster 必须知道应到哪里查找它要处理的数据。这通过 -D 选项指定。因此，启动服务器最简单的方法例如：

```
$ postmaster -D /usr/local/pgsql/data
```

这会让服务器在前台运行。同样，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；如果两者都不可用，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 &
```

按这里建议的方式把服务器的stdout和 stderr输出保存在某个地方，是极好的做法。这对审计和诊断问题都有帮助。（关于日志文件处理的更深入讨论，见第 8.3 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 3.4 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），还必须指定 -i 选项。

这些 shell 语法很快就会让人觉得繁琐。因此提供了封装部分任务的 shell 脚本包装器 pg_ctl。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 选项含义与直接调用 postmaster 时相同。pg_ctl 还实现了一个对称的“stop”操作。

通常，你会希望在计算机启动时就启动数据库服务器。这不是必需的；PostgreSQL 服务器可以在非特权账户下无需 root 干预地成功运行。

不同系统在引导时启动守护进程的惯例各不相同，因此建议你熟悉它们。许多系统有 /etc/rc.local 或 /etc/rc.d/rc.local 文件，把这样的命令放在那里几乎肯定没有坏处。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，你大概总是想把命令写成 su -c '...' postgres 这样的形式，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。（务必替换为你所选的正确安装目录和用户名。）

- 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，把启动和关闭集成进运行级别系统。

- 在NetBSD上，可以根据喜好以 FreeBSD或Linux 的启动脚本为示例，并把文件放在 `/usr/local/etc/rc.d/postgresql`。
- 在Solaris上，创建一个名为 `/etc/init.d/postgresql`的文件，包含以下单行内容：

```
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 保存在数据目录中的 `postmaster.pid`文件里。这用作防止多个 postmaster 在同一个数据目录上运行的互锁，也可以用来关闭 postmaster。

3.3.1. 服务器启动失败

postmaster 启动失败有几种常见原因。请检查 postmaster 的日志文件，或者手工启动它（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。
有些可能的错误消息是相当不言自明的，下面列出一些不是的。

```
FATAL: StreamServerPort: bind() failed: Address already in use
        Is another postmaster already running on that port?
```

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第二个 postmaster。不过，如果内核错误消息不是 `Address already in use`或该措辞的某种变体，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
FATAL: StreamServerPort: bind() failed: Permission denied
        Is another postmaster already running on that port?
```

像：

```
IpcMemoryCreate: shmget(key=5440001, size=83918612, 01600) failed:
 Invalid argument
```

```
FATAL 1:  ShmemCreate: cannot create region
```

这样的消息可能表示你的内核对共享内存区域大小的限制小于 PostgreSQL 尝试创建的缓冲区区域（本例中为 83918612 字节）。也可能表示你的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 postmaster（-B 开关）。不过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 postmaster，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：

```
IpcSemaphoreCreate: semget(key=5440026, num=16, 01600) failed: No
 space left on device
```

这样的错误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，你也许可以通过以减少的后端进程数（-N 开关）启动 postmaster 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，很可能是内核中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 3.5.1 节。

3.3.2. 客户端连接问题

客户端一侧可能出现的错误条件几乎无穷无尽且依赖于具体应用，
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使用 -i 选项，使 postmaster 能接受 TCP/IP 连接。

另一种情况是，尝试通过 Unix 套接字与本地 postmaster 通信时，会得到以下消息：

```
psql: could not connect to server: Connection refused
        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 postmaster，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 postmaster 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 4.3 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。

3.4. 运行时配置

有大量配置参数以各种方式影响数据库系统的行为。这里我们描述如何设置它们，
随后的小节将逐一详细讨论。

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、字符串，如下所述。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。

设置这些选项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
```

如你所见，选项每行一个。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。

每当 postmaster 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已在运行的后端进程，使现有会话也获得新的默认值。另外，你也可以只把信号直接发送给单个后端进程。

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
```

其效果与前面的例子相同。命令行选项覆盖 postgresql.conf 中任何与之冲突的设置。

有时，只给某个特定的后端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
```

（这适用于任何客户端应用程序，不只是 psql。）注意，对那些在服务器启动后必然固定的选项（例如端口号），这种方式不起作用。

最后，有些选项可以在单个 SQL 会话中用 SET 命令更改，例如：

```
=> SET ENABLE_SEQSCAN TO OFF;
```

语法的细节参见 SQL 命令语言参考。

3.4.1. 规划器和优化器调整

CPU_INDEX_TUPLE_COST (floating point)

设置查询优化器对索引扫描期间处理每个索引元组的代价的估计。
它以顺序获取一个页面的代价的一个比例来计量。

CPU_OPERATOR_COST (floating point)

设置优化器对处理 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。

CPU_TUPLE_COST (floating point)

设置查询优化器对查询期间处理每个元组的代价的估计。
它以顺序获取一个页面的代价的一个比例来计量。

EFFECTIVE_CACHE_SIZE (floating point)

设置优化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 8 kB。

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。
这主要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。
这主要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。
这主要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。
这主要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。
这主要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。
这主要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。这主要用于调试查询规划器。

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_EFFORT (integer)
GEQO_GENERATIONS (integer)
GEQO_POOL_SIZE (integer)
GEQO_RANDOM_SEED (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。随机种子（random seed）可以设置，以从算法获得可复现的结果。如果设置为 -1，则算法的行为是非确定性的。

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 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。此参数还控制优化器将子查询的 FROM 子句合并到上层查询的积极程度。

KSQO (boolean)

键集查询优化器（KSQO）使查询规划器把 WHERE 子句包含许多 OR 连接的 AND 子句的查询（例如 WHERE (a=1 AND b=2) OR (a=2 AND b=3) ...）转换为联合查询。这种方法可以比默认实现更快，但它不一定会给出完全相同的结果，因为 UNION 隐式添加了一个 SELECT DISTINCT 子句来消除相同的输出行。使用 Microsoft Access 之类的产品时通常会用到 KSQO，它们倾向于生成这种形式的查询。

KSQO 算法过去对包含许多 OR 连接的 AND 子句的查询绝对不可或缺，但在 PostgreSQL 7.0 及更高版本中，标准规划器已经能相当好地处理这些查询。因此默认是关闭的。

RANDOM_PAGE_COST (floating point)

设置查询优化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。

> **注意**
>
> 遗憾的是，没有定义良好的方法来确定刚才描述的这组“COST”变量的理想值。
> 鼓励你进行实验并分享你的发现。

3.4.2. 日志和调试

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（见 configure 选项 --enable-cassert）。注意，如果 PostgreSQL 是以这种方式构建的，DEBUG_ASSERTIONS 默认是打开的。

DEBUG_LEVEL (integer)

此值设置得越高，服务器运行期间在服务器日志中生成的各种“调试”输出就越多。
此选项默认为 0，表示没有调试输出。目前大约最高到 4 的值是有意义的。

DEBUG_PRINT_QUERY (boolean)
DEBUG_PRINT_PARSE (boolean)
DEBUG_PRINT_REWRITTEN (boolean)
DEBUG_PRINT_PLAN (boolean)
DEBUG_PRETTY_PRINT (boolean)

这些标志启用向服务器日志发送各种调试输出。对每个执行的查询，打印查询文本、所得的语法解析树、查询重写器输出或执行计划之一。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。把 DEBUG_LEVEL 设置为大于零会隐式打开其中一些标志。

HOSTNAME_LOOKUP (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_CONNECTIONS (boolean)

在服务器日志中打印一行，告知每个成功的连接。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_PID (boolean)

在服务器日志的每条消息前加上后端进程的进程 ID。这有助于分清哪些消息属于哪个连接。
默认是关闭的。

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。

SHOW_QUERY_STATS (boolean)
SHOW_PARSER_STATS (boolean)
SHOW_PLANNER_STATS (boolean)
SHOW_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。

SHOW_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。

STATS_COMMAND_STRING (boolean)
STATS_BLOCK_LEVEL (boolean)
STATS_ROW_LEVEL (boolean)

这些标志决定后端向统计收集器进程发送什么信息：当前命令、
块级活动统计或行级活动统计。全部默认关闭。启用统计收集会让每个查询花费少量时间，
但对调试和性能调优非常宝贵。

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。

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。

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。

要使用 syslog，PostgreSQL 的构建必须用 --enable-syslog 选项配置。

SYSLOG_FACILITY (string)

此选项决定启用 syslog 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。

TRACE_NOTIFY (boolean)

为 LISTEN 和 NOTIFY 命令生成大量调试输出。

3.4.3. 常规操作

AUSTRALIAN_TIMEZONES (bool)

如果设置为真，CST、EST 和 SAT 会被解释为澳大利亚时区，而不是北美中部/东部时区和星期六。默认是假。

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，服务器会毫不客气地断开连接。
这防止了挂起的客户端无限期占用连接。此选项只能在服务器启动时或在 postgresql.conf 文件中设置。

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，
所以我们不想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，然后才开始追问它究竟还能否被解锁。
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。此选项只能在服务器启动时设置。

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隔离级别被设置为什么。默认是读已提交。

更多信息参见 PostgreSQL 用户指南以及命令 SET TRANSACTION。

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。）示例值：

```
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，
但注意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。

FSYNC (boolean)

如果此选项打开，PostgreSQL 后端会在多处使用 fsync() 系统调用，确保更新被物理写入磁盘，而不是滞留在内核缓冲区缓存中。
这会大大增加数据库安装在操作系统或硬件崩溃之后仍然可用的机会。
（数据库服务器自身的崩溃不影响此考虑。）

不过，此操作会减慢 PostgreSQL，因为在所有那些位置它都必须阻塞并等待操作系统刷新缓冲区。不使用 fsync 时，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，这可以带来可观的性能提升。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。

此选项是 PostgreSQL 用户和开发者社区中一场永恒争论的主题。有些始终关闭它，有些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），
还有些只是为保险起见而保持打开。因为它是安全的一侧，打开也是默认值。
如果你信任你的操作系统、硬件和电力公司（更好的是你有 UPS），可能需要禁用 fsync。

应当指出，做 fsync 的性能损失在 PostgreSQL 7.1 版中已比以前的版本显著降低。如果你过去因性能问题禁用了 fsync，可以重新考虑你的选择。

此选项只能在服务器启动时或在 postgresql.conf 文件中设置。

KRB_SERVER_KEYFILE (string)

设置 Kerberos 服务器密钥文件的位置。详情见第 4.2.3 节。

MAX_CONNECTIONS (integer)

决定数据库服务器允许的并发连接数。默认值是 32（除非在构建服务器时被更改）。这个参数只能在服务器启动时设置。

MAX_EXPR_DEPTH (integer)

设置解析器可接受的最大表达式嵌套深度。默认值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后端崩溃的风险。）

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。代码实际使用的限制是此设置与 sysconf(_SC_OPEN_MAX) 结果中的较小者。因此，在 sysconf 返回合理限制的系统上，你无需担心此设置。但在某些平台上（特别是大多数 BSD 系统），sysconf 返回的值远大于当大量进程都试图打开那么多文件时系统真正能够支持的值。如果你发现出现“Too many open files”失败，可尝试调低此设置。此选项只能在服务器启动时或在 postgresql.conf 配置文件中设置；如果在配置文件中更改，只影响随后启动的服务器子进程。

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）的最大数量。默认值是 100。这个选项只能在服务器启动时设置。

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。默认值是 10000。这个选项只能在服务器启动时设置。

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。

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，此标志决定口令是否被加密。默认是关闭的（不加密口令），但这一选择在未来的版本中可能改变。

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。

SHARED_BUFFERS (integer)

设置数据库服务器将使用的共享内存缓冲区数量。默认值是 64。每个缓冲区通常为 8192 字节。这个选项只能在服务器启动时设置。

SILENT_MODE (bool)

让 postmaster 静默运行。如果设置了此选项，postmaster 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 之类的日志系统，否则不建议使用此选项，因为它会使你无法看到错误消息。

SORT_MEM (integer)

指定内部排序和哈希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 512 千字节。注意，对于复杂查询，可能同时运行多个排序和/或哈希，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。而且别忘了，
每个运行中的后端可能正在做一个或多个排序。因此所需的总内存可能是 SORT_MEM 值的许多倍。

SQL_INHERITANCE (bool)

此参数控制继承语义，特别是各种命令在考虑范围中是否默认包含子表。7.1 之前的版本不是这样。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，参见 SQL 语言参考和用户指南。

SSL (boolean)

启用SSL连接。使用前请阅读第 3.7 节。默认是关闭的。

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。

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 NULL 值时返回真，否则返回假。*expr* = NULL 的正确行为是总是返回 NULL（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 NULL，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 NULL，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 NULL 的表达式的语义感到困惑，所以我们不默认打开此选项。

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。

相关信息参见用户指南。

UNIX_SOCKET_DIRECTORY (string)

指定 postmaster 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 postmaster 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。合理的其他取值可以是 0770（仅用户和组，另见 UNIX_SOCKET_GROUP）和 0700（仅用户）。（注意，对 Unix 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 4 章中描述的机制。

这个选项只能在服务器启动时设置。

VACUUM_MEM (integer)

指定 VACUUM 用于跟踪待回收元组所用的最大内存量。值以千字节为单位指定，默认为 8192 千字节。对包含大量已删除元组的大表进行清理时，更大的设置可能提升清理速度。

VIRTUAL_HOST (string)

指定 postmaster 用于监听来自客户端应用连接的 TCP/IP 主机名或地址。默认监听所有已配置的地址（包括 localhost）。

3.4.4. WAL

关于 WAL 调优的细节另见第 11.3 节。

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）。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。这个选项只能在服务器启动时或在 postgresql.conf 文件中设置。

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，
该延迟就只是白白浪费时间。因此，只有当一个后端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。

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。

WAL_BUFFERS (integer)

共享内存中用于 WAL 日志的磁盘页缓冲区数量。这个选项只能在服务器启动时设置。

WAL_DEBUG (integer)

如果非零，在标准错误上打开与 WAL 相关的调试输出。

WAL_FILES (integer)

检查点时预先创建的日志文件数量。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）或 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。

3.4.5. 短选项

为方便起见，许多参数还有单字母的选项开关。它们在下表中描述。

表 3.1. 短选项对照

短选项	等价形式	备注
-B *x*	shared_buffers = *x*	
-d *x*	debug_level = *x*	
-F	fsync = off	
-h *x*	virtual_host = *x*	

短选项	等价形式	备注
-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	enable_indexscan=off ,enable_hashjoin=off, enable_mergejoin=off ,enable_nestloop=off ,enable_seqscan=off, enable_tidscan=off	*
-S *x*	sort_mem = *x*	*
-s	show_query_stats = on	*
-tpa, -tpl, -te	show_parser_stats=on, show_planner_stats=on, show_executor_stats=on	*

由于历史原因，标有“*”的选项必须通过 -o postmaster 选项传给单个后端进程，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。

3.5. 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。

3.5.1. 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，postmaster 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 3.3.1 节）。相关内核参数在不同系统上的命名基本一致，表 3.2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 3.2. 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）	250kB + 8.2 kB * shared_buffers + 14.2 kB * max_connections，或无穷大

名称	描述	合理的值
SHMMIN	共享内存段的最小尺寸（字节）	1
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	>= ceil(max_connections / 16)
SEMMNS	系统范围内的最大信号量数目	ceil(max_connections / 16) * 17 + 其他应用的空间
SEMMSL	每个集合中信号量的最大数目	>= 17
SEMMAP	信号量映射中的项数	见文本
SEMVMX	信号量的最大值	>= 255（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，即一个共享内存段所能拥有的最大尺寸（以字节计）。如果 shmget 返回类似 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共享缓冲区数量和允许的连接数量（-B/-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来说最多大约在 256 kB 上下（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连接设置再加上每 16 个允许连接一个的额外信号量（见表 3.2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，可能最终还需要增大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，其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）。因此，
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 postmaster 支持的共享缓冲区数量，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 4.1 或更高版本的用户，只需做上述更改、重新编译内核并重启。对于运行更早版本的用户，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。

Semaphores. 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。

在内核配置文件中设置你想要的值，例如：

```
options "SEMMNI=40"
options "SEMMNS=240"
options "SEMUME=40"
options "SEMMNU=120"
```

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。）

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*/shmparam.h和/usr/src/linux/include/linux/sem.h。

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（以字节计），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，为共享内存段设置的默认最大尺寸对 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://www.sunworld.com/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（以字节计）。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。

3.5.2. 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。尤其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控制登录时各种资源限制所设置的值。细节参见 login.conf。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。）

内核通常也对某些资源有依赖实现的系统范围限制。

- 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。

3.6. 关闭服务器

根据你的需要，在工作结束时关闭数据库服务器有几种方式。
其区别在于你向服务器进程发送什么信号。

SIGTERM

收到 SIGTERM 后，postmaster 不再接受新连接，但让现有的后端正常结束其工作。只有在所有后端都因客户端请求而终止之后，它才会关闭。这是智能关闭。

SIGINT

postmaster 不再接受新连接，并向所有现有的后端发送 SIGTERM，使它们中止当前事务并迅速退出。然后它等待后端退出，最终关闭数据库。这是快速关闭。

SIGQUIT

这是立即关闭，它会使 postmaster 向所有后端发送 SIGQUIT 并立即退出（不做对数据库系统的妥善关闭）。后端收到 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 postmaster。这会阻止 postmaster 释放共享内存和信号量，之后你可能不得不手工完成。

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 postmaster 提供了更方便的接口。

3.7. 使用 SSL 的安全 TCP/IP 连接

PostgreSQL 原生支持通过 SSL 的连接来加密客户端/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，并且构建时启用了相应支持（见第 1 章）。

编译了 SSL 支持后，PostgreSQL 服务器可以用 `-l`（ell）参数启动以启用 SSL 连接。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。这两个文件应分别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/IP 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。关于如何在服务器端强制某些连接使用 SSL，见第 4 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 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 cert.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。该脚本会生成一个受 passphrase 保护的密钥；它不接受少于四个字符的 pass phrase。要去除 passphrase（如果想让服务器自动启动就必须去除），运行命令

```
openssl rsa -in privkey.pem -out cert.pem
```

输入旧 passphrase 以解锁现有密钥。然后执行

```
openssl req -x509 -in cert.req -text -key cert.pem -out cert.cert
cp cert.pem $PGDATA/server.key
cp cert.cert $PGDATA/server.crt
```

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。

3.8. 使用 SSH 隧道的安全 TCP/IP 连接

致谢

此创意取自 Gene Selkov, Jr.（<selkovjr@mcs.anl.gov>）于 1999-09-08 回答 Eric Marsden 提问的一封电子邮件。

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，这应能提供足够安全的网络连接。

首先确保 ssh 服务器在 PostgreSQL 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

还有其他一些产品可以用与刚才描述的概念类似的步骤提供安全隧道。

第 4 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 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——有关的更多信息见第 7 章。因此，显然必须限制给定客户端能以哪些数据库用户名连接。

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）以请求的用户名进行连接。

PostgreSQL 提供了多种不同的客户端认证方法。可以使用的方法可以根据（客户端）主机和数据库来选择；某些认证方法还允许你按用户名加以限制。

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用户并没有本地账号，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。

4.1. 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。）当数据区域由 `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。空行和以井号（“#”）开头的行会被忽略。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。记录不能跨行续写。

每条记录指定一种连接类型、一个客户端 IP 地址范围（如果该连接类型需要）、一个或多个数据库名，以及对匹配这些参数的连接要使用的认证方法。第一条与某次连接尝试的类型、
客户端地址和请求数据库名匹配的记录被用来执行认证步骤。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，就不会再考虑后面的记录。
如果没有任何记录匹配，访问将被拒绝。

记录可以采用以下三种格式之一

```
local   database authentication-method [ authentication-option ]
host    database IP-address IP-mask authentication-method [
 authentication-option ]
hostssl database IP-address IP-mask authentication-method [
 authentication-option ]
```

各字段的含义如下：

`local`

该记录针对经由 Unix domain 套接字的连接尝试。

`host`

该记录针对经由 TCP/IP 网络的连接尝试。注意，除非服务器启动时带了 `-i` 开关或设置了等效的配置参数，否则 TCP/IP 连接被完全禁用。

`hostssl`

该记录针对在 TCP/IP 上使用 SSL 的连接尝试。要使用此选项，服务器编译时必须带有 SSL 支持启用。此外，服务器启动时必须用 `-l` 选项或等效的配置设置启用 SSL。（注意：`host` 记录既匹配 SSL 连接尝试也匹配非 SSL 连接尝试，但 `hostssl` 记录只匹配 SSL 连接。）

`database`

指定此记录适用的数据库。值 `all` 表示适用于所有数据库，而值 `sameuser` 标识与连接用户同名的那个数据库。否则，这就是某个具体 PostgreSQL 数据库的名称。

`IP address`
`IP mask`

这两个字段根据客户端机器的 IP 地址指定一条 `host` 或 `hostssl` 记录适用于哪些客户端机器。（当然，IP 地址可以伪造，但这一考虑超出了 PostgreSQL 的范围。）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。

`authentication method`

指定用户在这条认证记录的控制下连接时必须用来认证自身的方法。可选方法在此概述，细节见第 4.2 节。

`trust`

无条件地允许连接。此方法允许任何对客户端主机有登录访问权限的用户以任何 PostgreSQL 用户身份连接。

`reject`

无条件地拒绝连接。这主要用于从一组主机中“过滤掉”某些主机。

`password`

要求客户端提供一个密码，该密码须与为该用户设置的数据库密码匹配。

可以在 `password` 关键字之后指定一个可选的文件名。该文件应包含可以使用此记录连接的用户列表，还可以可选地包含他们的备用密码。

密码以明文形式在线路上发送。为了更好的保护，请使用 `md5` 或 `crypt` 方法。

`md5`

类似于 `password` 方法，但密码使用一个简单的质询-应答协议加密后在线路上发送。这可以防范偶然的线路嗅探。现在这是基于密码的认证的推荐选择。

`md5` 关键字后面可以跟一个文件名。该文件包含可以使用此记录连接的用户列表。

`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`。`crypt` 方法与在 `pg_shadow` 中加密密码不兼容，而且在客户端和服务器机器的 crypt() 库例程实现不同时可能失败。

`krb4`

使用 Kerberos V4 认证用户。这只适用于 TCP/IP 连接。

`krb5`

使用 Kerberos V5 认证用户。这只适用于 TCP/IP 连接。

`ident`

用户在登录操作系统时确定的身份被 PostgreSQL 用来确定该用户是否被允许以请求的数据库用户身份连接。对 TCP/IP 连接，用户的身份通过联系客户端主机上的 ident 服务器来确定。（注意，这只有在远程 ident 服务器可靠时才可靠；对管理员不可信的远程主机绝不应使用 ident 认证。）在支持对 Unix domain 套接字的 `SO_PEERCRED` 请求的操作系统上，本地连接也可以进行 ident 认证；此时会向系统询问连接用户的身份。

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。

跟在 `ident` 关键字后面的 *`authentication option`* 指定一个ident 映射的名称，该映射指定哪些操作系统用户等同于哪些数据库用户。细节见下文。

`pam`

这种认证类型的工作方式与 password 类似，主要区别是它使用 PAM（可插拔认证模块）作为认证机制。跟在 `pam` 关键字后面的 *`authentication option`* 指定要传递给 PAM 的服务名。默认服务名为 `postgresql`。关于 PAM 的更多信息，请阅读 Linux-PAM 页面[1]和/或 Solaris PAM 页面[2]。

`authentication option`

此字段的解释因认证方法而异，如上所述。

由于对每次连接尝试都是顺序检查 `pg_hba.conf` 记录，记录的顺序非常重要。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 连接使用 `trust` 认证，但对远程 TCP 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。

`pg_hba.conf` 文件在启动时以及 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 4.1。各种认证方法的细节见下文。

例 4.1. 一个 `pg_hba.conf` 文件示例

```
# TYPE       DATABASE    IP_ADDRESS    MASK               AUTHTYPE
 MAP

# Allow any user on the local system to connect to any
# database under any username, but only via an IP connection:
```

[1] http://www.kernel.org/pub/linux/libs/pam/
[2] http://www.sun.com/software/solaris/pam/

```
host         all         127.0.0.1     255.255.255.255    trust

# The same, over Unix-socket connections:

local        all                                          trust

# Allow any user from any host with IP address 192.168.93.x to
# connect to database "template1" as the same username that ident on
 that
# host identifies him as (typically his Unix username):

host         template1   192.168.93.0  255.255.255.0      ident
 sameuser

# Allow a user from host 192.168.12.10 to connect to database
 "template1"
# if the user's password in pg_shadow is correctly supplied:

host         template1   192.168.12.10 255.255.255.255    md5

# In the absence of preceding "host" lines, these two lines will
 reject
# all connection attempts from 192.168.54.1 (since that entry will be
# matched first), but allow Kerberos V5-validated 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:

host         all        192.168.54.1   255.255.255.255    reject
host         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":

host         all        192.168.0.0    255.255.0.0        ident
 omicron

# If these are the only two lines for local connections, they will
 allow
# local users to connect only to their own databases (database named
 the
# same as the user name), except for administrators who may connect to
# all databases.  The file $PGDATA/admins lists the user names who are
# permitted to connect to all databases.  Passwords are required in
 all
```

```
# cases.  (If you prefer to use ident authorization, an ident map can
# serve a parallel purpose to the password list file used here.)

local        sameuser                                        md5
local        all                                             md5  admins
```

4.2. 认证方法

下面详细描述这些认证方法。

4.2.1. trust 认证

当trust认证被指定时，PostgreSQL假设任何能连接到 postmaster 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。只有当在系统层对进入 postmaster 端口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 postmaster 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，可设置 postgresql.conf 中的参数 unix_socket_permissions（可能还有 unix_socket_group），如第 3.4.3 节中所述。或者，也可以设置 unix_socket_directory 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 连接不会被它限制。因此，如果你想利用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 的行所允许连接到 postmaster 的每台机器上的每个用户时，trust 认证才适合用于 TCP 连接。对来自 localhost（127.0.0.1）以外的任何 TCP 连接使用 trust，通常都不合理。

4.2.2. 密码认证

基于密码的认证方法包括 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；如果必须支持过时的客户端，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。通常，每个数据库用户的密码存储在 pg_shadow 系统目录表中。密码可以用查询语言命令 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 NULL，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，
可把用户集合放进一个单独的文件（每行一个用户名），该文件与 pg_hba.conf 位于同一目录中，并在 pg_hba.conf 中 password、md5 或 crypt 关键字之后分别提及该文件的（基）名。如果不使用此特性，那么数据库系统已知的任何用户都可以连接到任何数据库（当然，
只要他提供正确的密码）。

这些文件还可用于对某个特定数据库或一组数据库应用一组不同的密码。这种情况下，
文件的格式类似于标准的 Unix 口令文件 /etc/passwd，即

username:*password*

密码后面任何额外的冒号分隔字段都被忽略。密码应使用系统的 `crypt()` 函数加密。随 PostgreSQL 一起安装的工具程序 pg_passwd 可用来管理这些密码文件。

次要密码文件中可以混合带密码和不带密码的行。不带密码的行表示使用 `pg_shadow` 中由 `CREATE USER` 和 `ALTER USER` 管理的主密码。带密码的行将导致使用该密码。密码条目为“+”也表示使用 pg_shadow 密码。

使用 `md5` 或 `crypt` 方法时不能使用备用密码。文件仍会被照常读取，但密码字段将被简单地忽略，并总是使用 `pg_shadow` 密码。

注意，像这样使用备用密码意味着不能再使用 `ALTER USER` 来更改自己的密码。它看起来会成功，但你更改的密码并不是系统最终使用的密码。

4.2.3. 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[3] 或 MIT Project Athena[4] 可以作为入门的起点。存在若干 Kerberos 发行版来源。

要使用 Kerberos，必须在构建时启用对它的支持。Kerberos 4 和 5 都受支持（分别为 `./configure --with-krb4` 或 `./configure --with-krb5`），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域名。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）（见第 3.1 节）。密钥文件的位置由 `krb_server_keyfile` run time 配置参数指定。（另见第 3.4 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 mod_auth_krb 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。

[3] http://www.nrl.navy.mil/CCS/people/kenh/kerberos-faq.html
[4] ftp://athena-dist.mit.edu

4.2.4. 基于 ident 的认证

“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

在支持对 Unix domain 套接字的 `SO_PEERCRED` 请求的系统上，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。

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 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 4.1中的 `pg_hba.conf` 文件配合使用的 `pg_ident.conf` 文件见例 4.2。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 4.2. 一个 `pg_ident.conf` 文件示例

```
#MAP            IDENT-NAME    POSTGRESQL-NAME

omicron         bryanh        bryanh
omicron         ann           ann
# bob has username robert on these machines
omicron         robert        bob
# bryanh can also connect as guest1
```

```
omicron         bryanh       guest1
```

4.3. 认证问题

真正的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。

```
No pg_hba.conf entry for host 123.123.123.123, user joeblow, 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` 配置文件中找到授权条目。

```
Password authentication failed for user 'joeblow'
```

这样的消息表示你已经联系到了服务器，而且服务器也愿意与你交谈，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。

```
FATAL 1:  user "joeblow" does not exist
```

指定的用户名不存在。

```
FATAL 1:  Database "testdb" does not exist in the system catalog.
```

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，但这不一定是对的。

注意，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，
请检查日志。

第 5 章 本地化

从管理员的角度描述可用的本地化特性。

PostgreSQL通过三种方式支持本地化：

- 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。
- 使用在PostgreSQL服务器中定义的显式多字节字符集，
  来支持所需字符数超出单字节容量的语言，并提供客户端和服务器之间的字符集重编码。
  支持的字符集数量在编译服务器时固定，而字符串比较等内部操作需要把每个字符扩展为一个 32 位字。
- 单字节字符重编码为使用多种但仍为单字节字符集的用户提供了更轻量的解决方案。

5.1. 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 和类POSIX区域设置机制。更多信息请参考系统文档。

5.1.1. 概述

区域支持默认并不内置于PostgreSQL；要启用它，需要给`configure`脚本提供`--enable-locale`选项：

```
$ ./configure --enable-locale
```

区域支持只影响服务器；所有客户端都与带或不带区域支持的服务器兼容。

若要让消息可以翻译为用户偏好的语言，必须使用 `--enable-nls`选项。该选项独立于其他区域支持。

关于使用哪些特定文化规则的信息由标准环境变量决定。
如果你已经从其他程序得到了本地化行为，那么你大概已经设置好了这些变量。
设置本地化信息最简单的方式是`LANG`变量，例如：

```
export LANG=sv_SE
```

这会把区域设置为瑞典（`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，例如使用美国排序规则但使用西班牙语消息。为此，
存在一组环境变量，它们为特定类别覆盖`LANG`的默认值：

`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

此外，所有这些特定变量以及LANG变量都可以被LC_ALL环境变量覆盖。

> **注意**
>
> 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 gettext手册页。

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，
或者干脆取消设置所有与区域相关的变量。

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设置好这些变量。
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。

LC_COLLATE和LC_CTYPE变量影响索引的排序顺序。因此，对任何特定的数据库集簇，
这些值都必须保持固定，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；只有其他LC_类别可以在服务器启动时从环境设置。简而言之，一个数据库集簇中只能使用一种排序顺序，
而且它是在 initdb时选定的。

5.1.2. 优点

区域支持尤其会影响下列特性：

- ORDER BY查询中的排序顺序。
- to_char函数族
- 用于模式匹配的LIKE和~操作符

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。
特别要注意，选择非 C 区域会禁用LIKE和~操作符的索引优化，这可能对使用这些操作符的搜索速度产生巨大影响。

5.1.3. 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正确配置。要检查某个给定区域是否已安装并可用，
例如可以使用Perl。Perl 也支持区域设置，如果区域损坏，perl -v会给出类似下面这样的抱怨：

```
$export LC_CTYPE='not_exist'$perl -vperl: warning: Setting locale
 failed.
perl: warning: Please check that your locale settings:
LC_ALL = (unset),
LC_CTYPE = "not_exist",
LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

请检查你的区域文件是否位于正确的位置。可能的位置包括：/usr/lib/locale（Linux、Solaris）、/usr/share/locale（Linux）、/usr/lib/nls/loc（DUX 4.0）。如果不确定，请查阅系统的 locale 手册页。

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设置是在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）由启动 postmaster 的环境决定，通过简单地重启 postmaster 即可更改。你可以使用contrib/pg_controldata实用程序检查数据库的LC_COLLATE和LC_CTYPE设置。

src/test/locale目录包含 PostgreSQL区域设置支持的一个测试套件。

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。如果你创建这样的应用，就需要制定应对这种情况的方案。嵌入式 SQL 接口（ecpg）也受此问题影响。目前建议与ecpg应用交互的服务器配置为以英语发送消息。

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，请参考开发者指南，或向开发者邮件列表写信。

5.2. 多字节支持

作者

Tatsuo Ishii（<ishii@postgresql.org>），最后更新于 2000-03-22。更多信息请查看Tatsuo 的网站[1]。

多字节（MB）支持旨在让 PostgreSQL能够处理多字节字符集，例如EUC（扩展 Unix 编码）、Unicode 和 Mule 内部编码。启用MB后，你可以在正则表达式（regexp）、LIKE 以及其他一些函数中使用多字节字符集。默认编码系统是在使用initdb初始化 PostgreSQL安装时选定的。注意，这可以在你使用createdb创建数据库或使用 SQL 命令 CREATE DATABASE时被覆盖。因此你可以拥有多个数据库，并让每个数据库使用不同的编码系统。

5.2.1. 启用多字节支持

在运行 configure 时带上多字节选项：

```
./configure --enable-multibyte[=encoding_system]
```

其中*encoding_system*可以是下表中的值之一：

表 5.1. 字符集编码

编码	描述
SQL_ASCII	ASCII
EUC_JP	日语 EUC
EUC_CN	中文 EUC
EUC_KR	韩语 EUC

[1] http://www.sra.co.jp/people/t-ishii/PostgreSQL/

编码	描述
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	ECMA-113 拉丁/西里尔
ISO-8859-6	ECMA-114 拉丁/阿拉伯
ISO-8859-7	ECMA-118 拉丁/希腊
ISO-8859-8	ECMA-121 拉丁/希伯来
KOI8	KOI8-R(U)
WIN	Windows CP1251
ALT	Windows CP866

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），就应该非常小心这个变化。

重要

并非所有 API 都支持上面列出的编码。例如，PostgreSQL的 JDBC 驱动不支持 MULE_INTERNAL、LATIN6、LATIN8和LATIN10。

这里有一个把PostgreSQL默认配置为使用日语编码的例子：

```
$ ./configure --enable-multibyte=EUC_JP
```

如果省略了编码系统（./configure --enable-multibyte），则假定使用SQL_ASCII。

5.2.2. 设置编码

initdb为PostgreSQL 安装定义默认编码。例如：

```
$ initdb -E EUC_JP
```

这会把默认编码设为EUC_JP（日语的扩展 Unix 编码）。注意，如果你喜欢较长的选项形式，也可以用--encoding代替-E。如果既没有给出-E也没有给出 --encoding，就使用在 configure 时指定的编码。

你可以创建使用不同编码的数据库：

```
$ createdb -E EUC_KR korean
```

这会创建一个名为korean的数据库，使用 EUC_KR编码。另一种实现方式是使用 SQL 命令：

```
CREATE DATABASE korean WITH ENCODING = 'EUC_KR';
```

数据库的编码表示为pg_database系统目录中的一个编码列。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)
```

5.2.3. 服务器和客户端之间的自动编码翻译

对于某些编码，PostgreSQL支持在服务器和客户端之间进行自动编码翻译。可用的组合列在表 5.2中。

表 5.2. 客户端/服务器字符集编码

服务器编码	可用的客户端编码
SQL_ASCII	SQL_ASCII, UNICODE, MULE_INTERNAL
EUC_JP	EUC_JP, SJIS, UNICODE, MULE_INTERNAL
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 MULE_INTERNAL
LATIN6	LATIN6, UNICODE MULE_INTERNAL
LATIN7	LATIN7, UNICODE MULE_INTERNAL
LATIN8	LATIN8, UNICODE MULE_INTERNAL
LATIN9	LATIN9, UNICODE MULE_INTERNAL

服务器编码	可用的客户端编码
LATIN10	LATIN10, UNICODE MULE_INTERNAL
ISO_8859_5	ISO_8859_5, UNICODE
ISO_8859_6	ISO_8859_6, UNICODE
ISO_8859_7	ISO_8859_7, UNICODE
ISO_8859_8	ISO_8859_8, UNICODE
ISO_8859_9	ISO_8859_9, WIN, ALT, KOI8R, UNICODE, MULE_INTERNAL
UNICODE	EUC_JP, SJIS, EUC_KR, EUC_CN, EUC_TW, BIG5, LATIN1 to LATIN10, ISO_8859_5, ISO_8859_6, ISO_8859_7, ISO_8859_8, WIN, ALT, KOI8
MULE_INTERNAL	EUC_JP, SJIS, EUC_KR, EUC_CN, EUC_TW, BIG5, LATIN1 to LATIN5, WIN, ALT, WIN1250
KOI8	ISO_8859_9, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
WIN	ISO_8859_9, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
ALT	ISO_8859_9, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL

若要启用自动编码转换，必须告诉PostgreSQL　你希望客户端使用哪种编码。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得到：

  ```
  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 'encoding';
  ```

  也可以为此使用 SQL92 语法SET NAMES：

  ```
  SET NAMES 'encoding';
  ```

  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。）

5.2.4. 关于 Unicode

从PostgreSQL 7.1 开始支持 Unicode 与其他编码之间的自动编码翻译。在 7.1 中它默认不启用。要启用这个特性，运行 configure 时带上--enable-unicode-conversion选项。注意这还要求同时带--enable-multibyte选项。

对于 7.2，--enable-unicode-conversion不再必要。只要指定了--enable-multibyte，Unicode 转换功能就会自动启用。

5.2.5. 如果无法完成转换会怎样?

假设你为服务器选择了EUC_JP，为客户端选择了LATIN1，那么某些日语字符无法翻译为LATIN1。在这种情况下，无法用LATIN1字符集表示的字母会被变换为：

```
(HEXA DECIMAL)
```

5.2.6. 参考资料

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。

5.2.7. 历史

```
Dec 7, 2000
 * 实现了 Unicode 与其他编码之间的自动编码转换
 * 以上变更将出现在 7.1 中

May 20, 2000
 * SJIS UDC (NEC 选定 IBM 汉字) 支持，由 Eiji Tokuya 贡献
 * 以上变更将出现在 7.0.1 中

Mar 22, 2000
 * 新增 libpq 函数 PQsetClientEncoding、PQclientEncoding
 * ./configure --with-mb=EUC_JP
```

```
   已废弃。请改用 ./configure --enable-multibyte=EUC_JP
    * 新增 SQL_ASCII 回归测试用例
 * 新增 SJIS 用户自定义字符（UDC）支持
 * 以上全部将出现在 7.0 中

July 11, 1999
 * 新增对 WIN1250（Windows 捷克语）作为客户端编码的支持（由 Pavel Behal 贡献）
 * 修复一些编译器警告（由 Tomoaki Nishiyama 贡献）

Mar 23, 1999
 * 新增对 KOI8(KOI8-R)、WIN(CP1251)、ALT(CP866) 的支持（感谢 Oleg
 Broytmann 测试）
 * 修复 MB 与 locale 相关的问题

Jan 26, 1999
 * 新增 Big5 作为前端编码的支持（要使用 Big5，需用 EUC_TW 创建数据库）
 * 新增 EUC_TW 的回归测试用例（由 Jonah Kuo <jonahkuo@mail.ttn.com.tw>）

Dec 15, 1998
 * 修复了与 SQL_ASCII 支持相关的一些错误

Nov 5, 1998
 * 6.4 发布。此版本中，pg_database 增加了表示数据库编码的 "encoding" 列

Jul 22, 1998
 * 在 initdb/createdb 时确定编码，而非编译时
 * 在发出 COPY 命令时支持 PGCLIENTENCODING
 * 支持 SQL92 语法 "SET NAMES"
 * 支持 LATIN2-5
 * 新增 UNICODE 回归测试用例
 * 新的 MB 测试套件
 * 清理源代码文件

Jun 5, 1998
 * 新增后端与前端之间的编码转换支持
 * 新增 SET CLIENT_ENCODING 等新命令
 * 新增 LATIN1 字符集支持
 * 增强 8 位洁净性

April 21, 1998 一些增强/修复
 * character_length()、position()、substring() 现在能识别多字节字符
 * 新增 octet_length()
 * 为 configure 新增 --with-mb 选项
 * 新的 EUC_KR 回归测试（由 Soonmyung Hong 贡献）
 * 为 EUC_JP 回归测试新增一些测试用例
 * 修复 System V 情况下 regress/regress.sh 中的问题
 * 修复 toupper()、tolower() 以处理 8 位字符

Mar 25, 1998 MB PL2 被并入 PostgreSQL 6.3.1

Mar 10, 1998 PL2 发布
 * 为 EUC_JP、EUC_CN 和 MULE_INTERNAL 新增回归测试
 * 新增一份英文文档（本文件）
 * 修复与 8 位单字节字符有关的问题
```

Mar 1, 1998 PL1 发布

5.2.8. Windows/ODBC 上的 WIN1250

Windows 客户端平台上的 WIN1250 字符集可以配合启用了区域支持的 PostgreSQL使用。

应记住以下几点：

- 成功与否取决于正确的系统区域设置。这已经在Red Hat 6.0和Slackware 3.6上用 cs_CZ.iso8859-2区域测试过。
- 决不要尝试把服务器的数据库编码设为 WIN1250。始终改用 LATIN2，因为 Unix 中没有 WIN1250 区域。
- WIN1250 编码只对 Windows ODBC 客户端可用。字符会被动态转码，以便正确显示和存回。

Windows/ODBC 上的 WIN1250

1. 在启用区域支持、服务器端编码设为LATIN2的情况下编译 PostgreSQL。
2. 设置你的安装。不要忘记在环境中创建区域变量。例如（这可能不适用于你的环境）：

   ```
   LC_ALL=cs_CZ.ISO8859-2
   ```

3. 必须带着区域设置启动服务器！
4. 用捷克语试一试，并在查询上进行排序。
5. 在你的 Windows 机器上为PostgreSQL安装 ODBC 驱动。
6. 正确设置你的数据源。在 ODBC 配置对话框的Connect Settings字段中加入这一行：

   ```
   SET CLIENT_ENCODING = 'WIN1250';
   ```

7. 现在再试一次，但这次是在带 ODBC 的 Windows 中。

5.3. 单字节字符集重编码

你可以通过configure的--enable-recode选项启用这个特性。
这个选项以前被描述为“西里尔重编码支持”，那并没有体现它的全部能力。它可以用于
任何单字节字符集的重编码。

这种方法使用位于数据库目录（PGDATA）中的 charset.conf文件。它是一个典型的配置文本文件，其中空格和换行分隔项与记录，# 表示注释。这里识别三个关键字，语法如下：

```
BaseCharset      server_charset
RecodeTable      from_charset to_charset file_name
HostCharset      host_spec    host_charset
```

BaseCharset定义数据库服务器的编码。所有字符集名只在charset.conf内部用于映射，
因此你可以随意使用便于键入的名称。

RecodeTable记录指定服务器和客户端之间的转换表。文件名相对于PGDATA目录。
表文件的格式非常简单。没有关键字，字符由单行上的一对十进制或十六进制（0x 前缀）值表示：

char_valuetranslated_char_value

HostCharset记录按 IP 地址定义客户端字符集。你可以使用单个 IP 地址、从给定地址开始的 IP 掩码范围或 IP 区间（例如 127.0.0.1、192.168.1.100/24、192.168.1.20-192.168.1.40）。

charset.conf文件总是被处理到末尾，因此你可以方便地为前面的规则指定例外。在src/data/ 目录中你可以找到charset.conf的一个示例和一些重编码表。

由于这个方案基于客户端的 IP 地址和字符集映射，显然也有一些限制。你不能在同一台主机上同时使用不同的编码。当你的客户端主机启动到多个操作系统时也不方便。不过，
当这些限制不构成障碍且你不需要多字节字符时，它是一个简单而有效的方案。

第 6 章 管理数据库

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，属于整个安装，可以从安装中的每个数据库访问。）
连接到数据库服务器的应用在它的连接请求中指定它想要连接的数据库的名称。
每个连接不能访问多于一个数据库。
（但一个应用可以打开的到同一数据库或其他数据库的连接数量不受限制。）

> **注意**
>
> SQL 把数据库称为“目录”，但在实践中没有区别。

要创建或删除数据库，PostgreSQL postmaster 必须已经启动并正在运行（见第 3.3 节）。

6.1. 创建数据库

数据库使用查询语言命令 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 7.1.1 节。

引导（Bootstrapping）：. 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 3.2 节）。这个数据库按惯例叫做 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。更多细节见下文。

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。它在内部使用 psql 程序。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，这未必是你想要的。

6.1.1. 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 plpgsql 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。

可以创建额外的模板数据库，实际上也可以通过将安装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后端进程（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：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元组都已被“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对它进行例行的 VACUUM 维护。更多信息见第 8.2.3 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，也是多种脚本（如 createdb）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。

6.1.2. 备选位置

可以在安装的默认位置之外的地方创建数据库。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。（见第 3.6 节和第 3.3 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 3.1 节）。然后在命令行输入

```
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
> ```

6.2. 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，后者将是删除某个给定安装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 7 章 数据库用户和权限

管理数据库用户及其权限在概念上类似于管理 Unix 操作系统的用户，但细节并不相同。

7.1. 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。

为了方便，提供了 shell 脚本 `createuser` 和 `dropuser` 作为这些 SQL 命令的包装。

为了引导数据库系统，新初始化的系统总是包含一个预定义的用户。这个用户的固定 id 为 1，默认情况下（除非在运行 `initdb` 时更改）它与初始化该区域的操作系统用户同名（并且大概也被用作运行服务器的用户）。习惯上这个用户被命名为 postgres。要创建更多用户，你必须先以这个初始用户身份连接。

特定数据库连接所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` 程序用 `-U` 命令行选项指明要连接的用户。给定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 4 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。）

7.1.1. 用户属性

数据库用户可以拥有若干定义其权限并与客户端认证系统交互的属性。

superuser

数据库超级用户绕过所有权限检查。而且，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`。

数据库创建

必须显式给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`。

password

只有当客户端认证使用口令认证时，口令才有意义。数据库口令与操作系统口令是分开的。
创建用户时用 `CREATE USER name PASSWORD 'string'` 指定口令。

A user's attributes can be modified after creation with `ALTER USER`. See the reference pages for `CREATE USER` and `ALTER USER` for details.

7.2. 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, ...
```

7.3. 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。
目前没有更改数据库对象所有者的完善接口。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（读）、INSERT（追加）、UPDATE（写）、DELETE、RULE、REFERENCES（外键）和 TRIGGER。（更详细的信息见 GRANT 手册页。）修改或销毁对象的权利始终只属于拥有者。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，写访问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。

7.4. 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（例如写入 SQL 字段的关系）和触发器。对系统目录 pg_class、pg_shadow 和 pg_group 做审计跟踪和告警也是可行的。

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 8 章 日常数据库维护任务

8.1. 一般性讨论

为了保持PostgreSQL安装平稳运行，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 9 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 8.2 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 8.3 节中讨论。

与其他一些数据库产品相比，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。

8.2. 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的频率和范围，因各安装的需求而异。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，但不包括对表结构的更改）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。

8.2.1. 回收磁盘空间

在PostgreSQL的正常操作中，对一行执行UPDATE或DELETE
并不会立即删除旧的元组（该行的旧版本）。为了获得多版本并发控制的好处（见用户指南），
这种做法是必要的：在其他事务仍可能看到某个元组时，不能删除它。但最终，
一个过期或已删除的元组不再被任何事务关注。它占用的空间必须被回收以供新元组重用，
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元组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 比不频繁的 VACUUM FULL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。（如果安装中有多个数据库，
别忘了对每个数据库都进行清理；vacuumdb脚本可能会有帮助。）
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元组的情况下，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。

8.2.2. 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。

8.2.3. 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 比当前事务 XID 新的元组处于“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）的安装将遭遇事务 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。
新方法允许一个安装无限期保持运行，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，一旦一个元组以某个普通 XID 创建，无论我们谈论的是哪个普通 XID，该元组在接下来的 20 亿个事务中都会显得处于“过去”。如果该元组在超过 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，必须在旧的元组达到 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”，因此这样的元组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 亿个事务的元组。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，大多数元组可能终生都不会被“冻结”）。按照这个策略，任何表上两次VACUUM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、未被重新指定的元组现在可能已超过 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字段。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;
NOTICE:  Some databases have not been vacuumed in 1613770184
 transactions.
        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元组都会被冻结。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元组都被冻结。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。

8.3. 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，先确保PostgreSQL构建时使用了 --enable-syslog配置选项，然后在postgresql.conf中把 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。

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脚本更有用。如果你用 pg_ctl启动 postmaster，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，但网上有很多可用的；例如，Apache 发行版中就包含一个。

第 9 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

9.1. 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 Unix 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 4 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。

9.1.1. 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。

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。

9.1.2. 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，同时确保用户和组等全局数据的状态得到保留。pg_dumpall的调用序列很简单：

```
pg_dumpall > outfile
```

得到的转储可以按上文所述用psql恢复。但这种情况下你肯定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。

9.1.3. 大型数据库

> **致谢**
>
> 本文最初由 Hannu Krosing（<hannu@trust.ee>）写于 1999-06-19

由于PostgreSQL允许表大于系统上的最大文件尺寸，把该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 *nix 工具来绕过这个潜在问题。

使用压缩转储。. 使用你喜欢的压缩程序，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。. 它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。

9.1.4. 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。

9.2. 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 3. 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 3. 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，因为那将使数据库集群中的所有其他表都变得不可用。

另外注意，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。）

9.3. 版本间的迁移

作为一般规则，PostgreSQL的各版本之间内部数据存储格式可能发生变化。
这不适用于不同的“补丁级别”；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 1 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 4 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 /usr/src/postgresql-7.2.8
gmake install
initdb -D /usr/local/pgsql/data
postmaster -D /usr/local/pgsql/data
psql < backup
```

有关启动和停止服务器的方式及其他细节，见第 3 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 10 章 监控数据库活动

数据库管理员常常想知道“系统现在正在做什么？”本章讨论如何弄清这个问题。

有多种工具可用于监控数据库活动和分析性能。本章的大部分内容都在描述 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。用户指南讨论了 `EXPLAIN` 以及理解单个查询行为的其他方法。

10.1. 标准 Unix 工具

在大多数平台上，PostgreSQL 会修改 `ps` 报告的命令标题，这样单个服务器进程就可以很容易地识别。示例显示如下

```
$ ps auxww | grep ^postgres
postgres   960  0.0  1.1  6104 1480 pts/1    SN   13:17   0:00
 postmaster -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。它显示的命令参数与启动它时给出的相同。接下来的两个进程实现统计收集器，下一节将详细描述它。
（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。）
其余的每个进程都是处理一个客户端连接的服务器进程。
每个这样的进程都把它的命令行显示设置为如下形式

```
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，
但活动指示符会变化。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，如果服务器当前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，我们可以推断进程 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。你必须使用 `/usr/ucb/ps` 而不是 `/bin/ps`。还必须使用两个 `w` 标志，而不只是一个。此外，你最初调用 postmaster 时的 `ps` 状态显示必须比每个后端提供的更短。如果这三点有一点没做到，每个后端的 `ps` 输出就都是最初的 postmaster 命令行。

10.2. 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查询。

10.2.1. 统计信息收集配置

由于统计收集会给查询执行增加一些开销，系统可以配置为收集或不收集信息。这由通常在 postgresql.conf 中设置的配置变量控制（设置配置变量的详情见第 3.4 节）。

要让统计收集器能够启动，必须把变量 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。

变量 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。它们分别决定一个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。通常这些变量在 postgresql.conf 中设置，从而作用于所有服务器进程，但可以用 SET 命令在单个服务器进程中开关它们。（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 更改这些变量。）

> **重要**
>
> 由于变量 STATS_COMMAND_STRING、STATS_BLOCK_LEVEL 和 STATS_ROW_LEVEL 默认为 false，默认配置下实际上不收集任何统计信息！必须先打开其中一个或多个，统计显示函数才会给出有用的结果。

10.2.2. 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。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（默认 500 毫秒）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 10.1. 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 PID、数据库、用户和当前查询。当前查询列只有超级用户可用；对其他用户读取为 NULL。（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，显示活动后端数、该数据库中已提交和已回滚的事务总数、读取的磁盘块总数，以及缓

视图名	描述
	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：顺序扫描和索引扫描的总次数、每种扫描返回的元组总数，以及元组插入、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：使用该索引的索引扫描总次数、读取的索引元组数，以及成功取出的堆元组数。（当存在指向过期堆元组的索引条目时，后者可能更少。）
pg_stat_sys_indexes	与 pg_stat_all_indexes 相同，但只显示系统表上的索引。
pg_stat_user_indexes	与 pg_stat_all_indexes 相同，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：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：该索引中的磁盘块读取数和缓冲区命中数。
pg_statio_sys_indexes	与 pg_statio_all_indexes 相同，但只显示系统表上的索引。
pg_statio_user_indexes	与 pg_statio_all_indexes 相同，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：该序列中的磁盘块读取数和缓冲区命中数。
pg_statio_sys_sequences	与 pg_statio_all_sequences 相同，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	与 pg_statio_all_sequences 相同，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_ 视图主要用来判断缓冲缓存的有效性。当实际磁盘读取数远小于缓冲区命中数时，缓存正在不调用内核的情况下满足大多数读请求。

通过编写使用与这些标准视图相同的底层统计访问函数的查询，
可以建立查看统计信息的其他方式。每个数据库的访问函数接受一个数据库 OID 来标识要报告的数据库。每表和每索引的函数接受一个表或索引 OID（注意，用这些函数只能看到当前数据库中的表和索引）。每个后端的访问函数接受一个后端 ID 号，其范围从一到当前活动后端的数量。

表 10.2. 统计访问函数

函数	返回类型	描述
pg_stat_get_db_numbackends(oid)	integer	数据库中的活动后端数

函数	返回类型	描述
pg_stat_get_db_xact_commit(oid)	bigint	数据库中已提交的事务数
pg_stat_get_db_xact_rollback(oid)	bigint	数据库中已回滚的事务数
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	数据库的在缓存中找到的磁盘块请求数
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	参数为表时顺序扫描读取的元组数，或参数为索引时读取的索引元组数
pg_stat_get_tuples_fetched(oid)	bigint	参数为表时顺序扫描取出的有效（未过期）表元组数，或参数为索引时用该索引的索引扫描取出的元组数
pg_stat_get_tuples_inserted(oid)	bigint	插入表的元组数
pg_stat_get_tuples_updated(oid)	bigint	表中更新的元组数
pg_stat_get_tuples_deleted(oid)	bigint	从表中删除的元组数
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	表或索引的在缓存中找到的磁盘块请求数
pg_stat_get_backend_idset()	set of integer	当前活动后端 ID 的集合（从 1 到 N，N 为活动后端数）。用法示例见下文。
pg_stat_get_backend_pid(integer)	integer	后端进程的 PID
pg_stat_get_backend_dbid(integer)	oid	后端进程的数据库 ID
pg_stat_get_backend_userid(integer)	oid	后端进程的用户 ID
pg_stat_get_backend_activity(integer)	text	后端进程的当前查询（调用者不是超级用户时为 NULL）

注意：blocks_fetched 减去 blocks_hit 得到对该表、索引或数据库发出的内核 read() 调用数；但由于内核级缓冲，实际物理读取数通常更低。

函数 pg_stat_get_backend_idset 提供一种为每个活动后端生成一行的便捷方法。例如，要显示所有后端的 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;
```

第 11 章 预写式日志（WAL）

Author

Vadim Mikheev 和 Oliver Elphick

11.1. 总体描述

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。遵循这一过程时，我们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）。

11.1.1. WAL 的直接收益

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. 索引元组指向不存在的表行
2. 索引元组在分裂操作中丢失
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。

11.1.2. 未来的收益

在 WAL 的首个版本中，UNDO 操作尚未实现，因为时间不够。这意味着由中止的事务所做的更改仍会占用磁盘空间，而且我们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。

11.2. 实现

从 7.1 版开始 WAL 自动启用。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 11.3 节）。

WAL 日志存储在目录 *$PGDATA*/pg_xlog 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，由后端处理；它们由轻量级锁保护。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。

如果日志与主数据库文件位于不同的磁盘上会更有利。这可以通过把目录 pg_xlog 移到另一个位置（当然要在 postmaster 关闭时）并从 *$PGDATA* 中的原位置创建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 日志文件的磁盘不做这样的虚假报告。

11.2.1. 用 WAL 进行数据库恢复

做完检查点并刷写日志之后，检查点的位置保存在 pg_control 文件中。因此，进行恢复时，后端先读取 pg_control 和检查点记录；然后从检查点记录所指示的日志位置向前扫描执行 REDO 操作。因为检查点后第一次修改页时数据页的完整内容都保存在日志中，检查点之后更改过的所有页面都将被恢复到一致的状态。

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。

11.3. WAL 配置

有若干影响数据库性能的与 WAL 相关的参数。本节说明它们的用法。设置配置参数的详情请查阅第 3.4 节。

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（例如元组插入）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 postgresql.conf 的 WAL_BUFFERS 参数来增加 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，undo 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。）

检查点生成器还能为将来的使用创建几个日志段，以避免 LogInsert 或 LogFlush 花时间创建它们。（如果发生那种情况，整个数据库系统都会被创建操作拖延，所以最好由不处于任何人关键路径上的检查点生成器来创建这些文件。）默认情况下，只有当前段用了超过 75% 时才创建新的 16MB 段文件。如果系统在检查点之间产生超过 4MB 的日志输出，这就不够了。可以通过修改配置参数 WAL_FILES 让服务器在检查点时预创建最多 64 个日志段。

postmaster 每隔一段时间派生一个专门的后端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。

16MB 段文件的数量总是至少 WAL_FILES + 1 个，通常不会超过 WAL_FILES + MAX(WAL_FILES, CHECKPOINT_SEGMENTS) + 1 个。这可以用来估算 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（改名为将来的下一个顺序段）。如果由于日志输出率的短期峰值，段文件超过了 WAL_FILES + MAX(WAL_FILES, 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。（如果这种情况经常发生，应当增大 WAL_FILES 来避免。删除以后还得重新创建的日志段既昂贵又无意义。）

COMMIT_DELAY 参数定义后端在用 LogInsert 把提交记录写入日志之后、执行 LogFlush 之前休眠多少微秒。这个延迟允许其他后端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，或者当前未处于活跃事务中的其他后端少于 COMMIT_SIBLINGS 个，就不会休眠；这避免了在不太可能有其他后端很快提交时休眠。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 调用都被记录到标准错误。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。

第 12 章 磁盘存储

本节尚待编写。FAQ 中有一些相关信息。有人愿意吗？- thomas 1998-01-11

第 13 章 数据库故障

必须假定数据库故障（或其可能性）正在潜伏，随时准备在未来某个时刻发动袭击。
一个谨慎的数据库管理员会为各种可能故障的必然性做好规划，并在故障发生
之前就准备好适当的计划和规程。

在发生硬件或软件故障时，数据库恢复是必要的。故障有多种类别；
其中一些只需要对数据库做相对较小的调整，
而另一些则可能依赖于事先准备好的数据库转储和其他恢复数据集。需要强调的是，
如果你的数据很重要和/或难以重新生成，那么你应当已经考虑并为各种故障场景做好了准备。

13.1. 磁盘已满

数据盘被填满可能导致数据库索引随后损坏，但不会损坏基本的数据表。如果 WAL 文件与数据在同一个磁盘上（默认配置就是这样），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，数据库将拒绝启动。
你必须释放磁盘上的额外空间（或者把 WAL 区域移到另一个磁盘；参见第 11.3 节），然后重启postmaster来从此状态中恢复。

13.2. 磁盘故障

任何与活动数据库相关的磁盘（或 RAID 子系统之类的逻辑存储设备）发生故障，都要求从事先准备好的数据库转储中恢复数据库。这个转储必须使用pg_dumpall 准备，并且在数据库安装被转储之后发生的数据库更新将会丢失。

第 14 章 回归测试

14.1. 简介

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。这个测试套件最初由 Jolly Chen 和 Andrew Yu 开发，并经 Marc Fournier 和 Thomas Lockhart 大幅修订和重新打包。从 PostgreSQL 6.1 起回归测试对每个正式发行版都是最新的。

14.2. 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，例如平台相关的“expected”文件和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 77 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 14.3 节。

> **注意**
>
> 由于这种测试方法运行的是一个临时服务器，当你以 root 用户操作时它将无法工作（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。

> **提示**
>
> 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后端、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，可以编辑文件 src/test/regress/parallel_schedule 把较大的并发测试集拆分成更多可管理的组。

提示

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，你也可以像上面建议的那样修改并行测试调度。

要在安装之后运行测试（见第 1 章），初始化一个数据区域并启动服务器，如第 3 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。

14.3. 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。）

14.3.1. 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。

14.3.2. Locale 差异

这些测试期望在普通的“C” locale 中运行。当你针对临时安装运行测试时，这应当不会造成任何问题，因为回归测试驱动程序会确保以 C locale 启动服务器。但是，如果你针对一个使用非 C locale 设置的已安装服务器运行测试，可能会看到由字符串排序规则、数字和货币值格式等方面的差异引起的问题。

在某些 locale 中，产生的差异很小，很容易通过检查确认。但是，在改变了数字值格式规则的 locale 中（通常是通过交换逗号和小数点的用法），某些数据值的输入将会失败，从而在测试后面本应使用这些缺失数据值的地方引起大量差异。

14.3.3. 日期和时间差异

如果你在一个夏令时切换日或其前一天或后一天运行 `timestamp` 测试，其中少数查询会失败。这些查询假定昨天午夜、
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。

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，你必须为 `PST8PDT` 时区提供系统库支持，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 GMT 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。

14.3.4. 浮点差异

某些测试涉及从表列计算 64 位（`double precision`）数。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。

14.3.5. 多边形差异

若干测试涉及对加利福尼亚州奥克兰/伯克利街道地图地理数据的操作。
地图数据被表示为多边形，其顶点用一对 `double precision`数（十进制纬度和经度）表示。
首先创建一些表并装入地理数据，然后创建一些使用多边形相交操作符（`##`）
连接两个表的视图，再对视图做一次 select。

在比较不同平台的结果时，差异出现在小数点右侧第 2 或第 3 位。出现这些问题的 SQL 语句是：

```
SELECT * from street;
SELECT * from iexit;
```

14.3.6. 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。

14.3.7. “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。）

14.4. 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 `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 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 HPPA 平台上消除虚假的“失败”消息，`resultmap` 中包含了

```
horology/hppa=horology-no-DST-before-1970
```

它会在任何 `config.guess` 输出以“hppa”开头的机器上触发。`resultmap` 中的其他行为其他合适的平台选择变体比较文件。

附录 A. 发布说明

A.1. 版本 7.2.8

发行日期
2005-05-09

本次发布包含对 7.2.7 的多方面修复，其中包括一项与安全相关的问题。

A.1.1. 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。

A.1.2. 变更

- 修复一个古老的竞态条件：它允许一个事务在某些用途上（例如 SELECT FOR UPDATE）比在其他用途上稍早被看作已提交

  这是一个极其严重的缺陷，因为它可能导致应用程序短暂地看到表面上的数据不一致。

- 修复关系扩展与 VACUUM 之间的竞态条件

  理论上这可能造成丢失一整页新插入的数据，尽管该场景发生的概率似乎极低。
  尚无已知案例表明它造成过比断言失败更严重的后果。

- 修复 `TIME WITH TIME ZONE` 值的 `EXTRACT(EPOCH)`

- plpgsql 中更多的缓冲区溢出检查（Neil）

- 修复 pg_dump，使其能正确转储包含 `%` 的索引名和触发器名（Neil）

- 防止 `to_char(interval)` 在与月份相关的格式下转储核心

- 修复 `contrib/pgcrypto` 以适配较新的 OpenSSL 构建（Marko Kreen）

A.2. 版本 7.2.7

发行日期
2005-01-31

本次发布包含对 7.2.6 的多方面修复，其中包括若干与安全相关的问题。

A.2.1. 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。

A.2.2. 变更

- 禁止非超级用户执行 `LOAD`

  在会自动执行共享库初始化函数的平台上（至少包括 Windows 和基于 ELF 的 Unix 系统），`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 函数添加缺失的 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

  连接结果被错误地认为与左侧输入的排序相同。这不仅可能向用户交付排序错误的输出，而且在嵌套归并连接的情况下可能给出完全错误的结果。

- 修复 SQL 和 GERMAN 日期样式中负时间间隔的显示

A.3. 版本 7.2.6

发行日期
2004-10-22

本次发布包含对 7.2.5 的多方面修复。

A.3.1. 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。

A.3.2. 变更

- 修复磁盘上提示位可能未更新的问题

  在罕见情况下，这一疏漏可能导致“无法访问事务状态”失败，因此可归为潜在数据丢失缺陷。

- 确保哈希外连接不遗漏元组

  在使用哈希连接计划的超大型左连接中，若数据分布恰好合适，可能无法输出左侧未匹配的行。

- 禁止以 root 身份运行 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这个脚本。

- 更新到较新版本的 Bison

A.4. 版本 7.2.5

发行日期
2004-08-16

本次发布包含对 7.2.4 的多方面修复。

A.4.1. 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。

A.4.2. 变更

- 防止崩溃期间可能丢失已提交事务

  由于事务提交与检查点操作之间的互锁不足，在数据库崩溃并重启之后，
  恰好在最近一次检查点之前提交的事务可能整体或部分丢失。这是自 PostgreSQL 7.1 以来就存在的一个严重缺陷。

- 修复与第一次根页分割并发的 B-树搜索的极端情况
- 修复 `to_ascii` 中的缓冲区溢出（Guido Notari）
- 修复 char 为无符号类型的机器上死锁检测的核心转储
- 修复 Async_NotifyHandler 运行后对 `pg_ctl stop -m fast` 无响应的问题
- 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

A.5. 版本 7.2.4

发行日期
2003-01-30

本次发布包含对 7.2.3 的多方面修复，其中包括防止可能的数据丢失的修复。

A.5.1. 迁移到版本 7.2.4

对于运行 7.2.* 的用户，不需要进行转储/恢复。

A.5.2. 变更

- 修复 VACUUM“No one parent tuple was found”错误的更多情况
- 禁止在函数内调用 VACUUM（Bruce）

- 确保 pg_clog 更新在标记检查点完成之前同步到磁盘
- 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 时仍能工作
- 修复 datetime 表中的错误；某些时区名未被识别
- 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

A.6. 版本 7.2.3

发行日期
2002-10-01

本次发布包含对 7.2.2 的多方面修复，其中包括防止可能的数据丢失的修复。

A.6.1. 迁移到版本 7.2.3

对于运行 7.2.* 的用户，不需要进行转储/恢复。

A.6.2. 变更

- 防止可能的事务日志压缩数据丢失（Tom）
- 防止非超级用户增大最近的 vacuum 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 以正确转储 FULL JOIN USING（Tom）

A.7. 版本 7.2.2

发行日期
2002-08-23

本次发布包含对 7.2.1 的多方面修复。

A.7.1. 迁移到版本 7.2.2

对于运行 7.2.* 的用户，不需要进行转储/恢复。

A.7.2. 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 使其能处理 > 0x7f 的字节（Tatsuo）
- 修复 psql 和 pg_dump 在以不存在的长选项调用时崩溃的问题（Tatsuo）
- 修复调用几何操作符时的崩溃（Tom）
- 允许 OPEN cursor(args)（Tom）
- 修复 rtree_gist 索引构建（Teodor）
- 修复用户定义聚合的转储（Tom）
- contrib/intarray 修复（Oleg）
- 修复使用圆括号的复杂 UNION/EXCEPT/INTERSECT 查询（Tom）
- 修复 pg_convert（Tatsuo）
- 修复长 DATA 字符串导致的崩溃（Thomas、Neil）
- 修复 repeat()、lpad()、rpad() 与长字符串的问题（Neil）

A.8. 版本 7.2.1

发行日期
2002-03-21

本次发布包含对 7.2 的多方面修复。

A.8.1. 迁移到版本 7.2.1

对于运行 7.2 的用户，不需要进行转储/恢复。

A.8.2. 变更

- 确保序列计数器在崩溃之后不会倒退（Tom）
- 修复 pgaccess 的汉字转换按键绑定（Tatsuo）
- 优化器改进（Tom）
- Cash 输入/输出改进（Tom）
- 新的俄语 FAQ
- 修复缺失 AuthBlockSig 的编译问题（Heiko）

- 更多时区和时区修复（Thomas）
- 允许 psql \connect 处理大小写混合的数据库名和用户名（Tom）
- 即使有 ON INSERT 规则也在命令完成时返回正确的 OID（Tom）
- 允许 COPY FROM 使用 8 位 DELIMITERS（Tatsuo）
- 修复 extract/date_part 中毫秒/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，额外的安装步骤参见 README.tsearch（Thomas T. Thai、Teodor Sigaev）
- 修复数组下标处理（Tom）
- 允许在 PL/pgSQL 中 EXECUTE "CREATE TABLE AS ... SELECT"（Tom）

A.9. 版本 7.2

发行日期
2002-02-04

A.9.1. 概述

此发行版改进了 PostgreSQL 在高负载应用中的使用。

此发行版的主要变更：

VACUUM

清理（vacuum）不再锁定表，因此在清理期间允许正常的用户访问。新的 `VACUUM FULL` 命令执行旧式清理：锁定表并收缩表的磁盘副本。

事务

超过四十亿事务的安装不再有问题。

OID

OID 现在是可选的。对于 OID 使用过度的情况，用户现在可以创建不带 OID 的表。

优化器

系统现在会在 `ANALYZE` 期间计算直方图列统计信息，使优化器能够做出更好的选择。

安全性

新的 MD5 加密选项允许更安全地存储和传输口令。Linux 和 BSD 系统上还提供了新的 Unix 域套接字认证选项。

统计

管理员可以使用新的表访问统计模块获取有关表和索引使用情况的细粒度信息。

国际化

程序和库消息现在可以用多种语言显示。

A.9.2. 迁移到版本 7.2

希望从任何以前的发行版迁移数据的用户，需要使用 `pg_dump` 进行转储/恢复。

注意以下不兼容之处：

- 此发行版中 `VACUUM` 命令的语义发生了变化。你可能需要相应地更新你的维护流程。
- 在此发行版中，使用 `= NULL` 的比较将总是返回 false（更准确地说，是 NULL）。以前的发行版会自动将此语法转换为 `IS NULL`。可以通过 `postgresql.conf` 参数重新启用旧行为。
- `pg_hba.conf` 和 `pg_ident.conf` 配置现在只在收到 SIGHUP 信号之后重新加载，而不是在每次连接时加载。
- 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。你需要改写你的应用程序。
- 函数 `timestamp()`、`time()` 和 `interval()` 不再可用。请用 `timestamp 'string'` 或 `CAST` 代替 `timestamp()`。

`SELECT ... LIMIT #,#` 语法将在下一个发行版中移除。你应当把查询改为使用独立的 LIMIT 和 OFFSET 子句，例如 `LIMIT 10 OFFSET 20`。

A.9.3. 变更

A.9.3.1. 服务器操作

- 在独立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）
- 将 pg_log 更名为 pg_clog（Tom）
- 启用 SIGTERM、SIGQUIT 来杀死后端（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）
- 从一些系统表中移除 OID（Tom）

- 移除 "triggered data change violation" 错误检查（Tom）
- 预备/保存计划的 SPI 门廊创建（Jan）
- 允许 SPI 列函数对系统列工作（Tom）
- 长值压缩改进（Tom）
- 用于表、索引访问的统计收集器（Jan）
- 将超长序列名截断为合理值（Tom）
- 以毫秒为单位测量事务时间（Thomas）
- 修复 TID 顺序扫描（Hiroshi）
- 超级用户 ID 现在固定为 1（Peter E）
- 新的 pg_ctl "reload" 选项（Tom）

A.9.3.2. 性能

- 优化器改进（Tom）
- 供优化器使用的新的直方图列统计信息（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竞争（Tom）
- 为索引访问支持函数保留 relcache 项（Tom）
- 改进 NUMERIC 中 NaN 和无穷大的选择率（Tom）
- R-tree 性能改进（Kenneth Been）
- B-树分割更高效（Tom）

A.9.3.3. 权限

- 将 UPDATE、DELETE 权限改为彼此独立（Peter E）
- 新的 REFERENCES、TRIGGER 权限（Peter E）
- 允许一次对多个用户执行 GRANT/REVOKE（Peter E）
- 新的 has_table_privilege() 函数（Joe Conway）
- 允许非超级用户清理数据库（Tom）
- 新的 SET SESSION AUTHORIZATION 命令（Peter E）
- 修复新创建表上权限修改的缺陷（Tom）

- 禁止非超级用户访问 pg_statistic，添加用户可访问的视图（Tom）

A.9.3.4. 客户端认证

- 在认证之前 fork postmaster 以防止挂起（Peter E）
- 在 Linux、*BSD 上添加基于 Unix 域套接字的 ident 认证（Helge Bahmann、Oliver Elphick、Teodor Sigaev、Bruce）
- 添加使用 MD5 加密的口令认证方法（Bruce）
- 允许使用 MD5 加密存储的口令（Bruce）
- PAM 认证（Dominic J. Eidson）
- 只在启动和 SIGHUP 时加载 pg_hba.conf 和 pg_ident.conf（Bruce）

A.9.3.5. 服务器配置

- 某些时区缩写按澳大利亚而不是北美解释，现在可以在运行时设置（Bruce）
- 设置默认事务隔离级别的新参数（Peter E）
- 将 "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）

A.9.3.6. 查询

- INSERT 规则添加的语句现在在 INSERT 之后执行（Jan）
- 禁止目标列表中出现未修饰的关系名（Bruce）
- ORDER BY 中 NULL 现在排在所有正常值之后（Tom）
- 新的 IS UNKNOWN、IS NOT UNKNOWN 布尔测试（Tom）
- 新的 SHARE UPDATE EXCLUSIVE 锁模式（Tom）
- 新的 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 LIMIT、DISTINCT ON 被下推到子查询的问题（Tom）
- 修复嵌套 EXCEPT/INTERSECT（Tom）

A.9.3.7. 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）
- 新的 SERIAL8 创建带序列的 int8 列，默认仍是 SERIAL4（Tom）
- 使用 WITHOUT OIDS 使 OID 变为可选（Tom）
- 为 CREATE TYPE 添加 %TYPE 语法（Ian Lance Taylor）
- 为 CHECK 约束添加 ALTER TABLE / DROP CONSTRAINT（Christopher Kings-Lynne）
- 新的 CREATE OR REPLACE FUNCTION 用于更改现有函数（保留函数 OID）（Gavin Sherry）
- 添加 ALTER TABLE / ADD [UNIQUE | PRIMARY]（Christopher Kings-Lynne）
- 允许重命名视图中的列
- 使 ALTER TABLE / RENAME COLUMN 更新索引的列名（Brent Verner）
- 修复继承表上 ALTER TABLE / ADD CONSTRAINT ... CHECK 的问题（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 函数添加自动的返回类型数据转换（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）

A.9.3.8. 实用命令

- 添加 RESET ALL、SHOW ALL（Marko Kreen）
- CREATE/ALTER USER/GROUP 现在允许任意顺序的选项（Vince）
- 添加 LOCK A, B, C 功能（Neil Padgett）
- 为 CREATE/ALTER USER 添加新的 ENCRYPTED/UNENCRYPTED 选项（Bruce）
- 新的轻量级 VACUUM 不锁定表；旧语义以 VACUUM FULL 提供（Tom）
- 禁止对视图执行 COPY TO/FROM（Bruce）
- COPY DELIMITERS 字符串必须恰好为一个字符（Tom）
- 索引元组少于堆元组的 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）
- 禁止对 CREATE/DROP INDEX/TRIGGER/VIEW 的不当使用（Tom）

A.9.3.9. 数据类型和函数

- SUM()、AVG()、COUNT() 现在内部使用 int8 以提升速度（Tom）
- 添加 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 现在拒绝与声明长度不匹配的位串（Peter E）
- INET、CIDR 文本转换函数（Alex Pilosov）
- INET、CIDR 的操作符 << 和 <<= 现在可索引（Alex Pilosov）
- Bytea 的 \### 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-树索引
- Bytea 现在支持 LIKE、LIKE...ESCAPE、NOT LIKE、NOT LIKE...ESCAPE
- Bytea 现在支持连接
- 新的 bytea 函数：position、substring、trim、btrim 和 length
- encode() 函数的新模式 "escaped"，在最少转义的 bytea 与 text 之间转换
- 添加 pg_database_encoding_max_length()（Tatsuo）
- 添加 pg_client_encoding() 函数（Tatsuo）
- now() 返回毫秒精度的时间（Thomas）
- 新的 TIMESTAMP WITHOUT TIMEZONE 数据类型（Thomas）
- 添加带 "T" 的 ISO 日期/时间规范，yyyy-mm-ddThh:mm:ss（Thomas）
- 新的 xid/int 比较函数（Hiroshi）
- 为 TIME、TIMESTAMP 和 INTERVAL 数据类型添加精度（Thomas）
- 修改类型强制转换逻辑以优先尝试二进制兼容函数（Tom）
- 新的 encode() 函数默认安装（Marko Kreen）
- 改进的 to_*() 转换函数（Karel Zak）
- 使用单字节编码时优化 LIKE/ILIKE（Tatsuo）
- contrib/pgcrypto 中的新函数：crypt()、hmac()、encrypt()、gen_salt()（Marko Kreen）
- 更正 translate() 函数的描述（Bruce）
- 为 SET TIME ZONE 添加 INTERVAL 参数（Thomas）
- 添加 INTERVAL YEAR TO MONTH（等）语法（Thomas）

- 使用单字节编码时优化 length 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) 现在返回未压缩长度（Tatsuo、Bruce）
- 处理日期/时间字面量中 "July" 这样的全名（Greg Sabino Mullane）
- 一些 datatype() 函数调用的求值方式发生了变化
- 添加对儒略和 ISO 时间规范的支持（Thomas）

A.9.3.10. 国际化

- psql、pg_dump、libpq 和服务器的国家语言支持（Peter E）
- 简体中文、繁体中文、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）
- 添加 LATIN5、6、7、8、9、10 支持（Tatsuo）
- 添加 ISO 8859-5、6、7、8 支持（Tatsuo）
- 更正 LATIN5 使其表示 ISO-8859-9 而不是 ISO-8859-5（Tatsuo）
- 使 mic2ascii() 感知非 ASCII（Tatsuo）
- 拒绝无效的多字节字符序列（Tatsuo）

A.9.3.11. PL/pgSQL

- 现在 SELECT 循环使用门廊，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）
- 添加 ELSEIF（Klaus Reger）
- 改进 PL/pgSQL 错误报告，包括错误位置（Tom）
- 为兼容性，允许在游标声明中使用 IS 或 FOR 关键字（Bruce）
- 修复 SELECT ... FOR UPDATE（Tom）
- 修复 PERFORM 返回多行的问题（Tom）
- 使 PL/pgSQL 使用服务器的类型强制转换代码（Tom）
- 内存泄漏修复（Jan、Tom）
- 使末尾分号变为可选（Tom）

A.9.3.12. PL/Perl

- 新的非信任 PL/Perl（Alex Pilosov）
- 即使 libperl 不是共享库，PL/Perl 现在也能在某些平台上构建（Peter E）

A.9.3.13. PL/Tcl

- 现在报告 errorInfo（Vsevolod Lobko）
- 添加 spi_lastoid 函数（bob@redivi.com）

A.9.3.14. PL/Python

- ……是新增的（Andrew Bosma）

A.9.3.15. psql

- \d 按唯一、主键分组显示索引（Christopher Kings-Lynne）
- 允许反斜杠命令带末尾分号（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）

A.9.3.16. 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

A.9.3.17. 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）
- 修复时区处理（Barry Lind）
- 改进的 Druid 支持
- 允许非多字节服务器使用八位字符（Barry Lind）
- 支持 BIT、BINARY 类型（Ned Wolpert）
- 减少内存使用（Michael Stephens、Dave Cramer）
- 更新 DatabaseMetaData（Peter E）

- 添加 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）
- JDBC1 和 JDBC2 的部分合并（Anders Bengtsson）
- 事务性能改进（Barry Lind）
- 数组修复（Greg Zoller）
- 序列化添加
- 修复批处理（Rene Pijlman）
- ExecSQL 方法重组（Anders Bengtsson）
- GetColumn() 修复（Jeroen van Vianen）
- 修复 isWriteable() 函数（Rene Pijlman）
- 改进 JDBC2 一致性测试的通过率（Rene Pijlman）
- 添加 bytea 类型能力（Barry Lind）
- 添加 isNullable()（Rene Pijlman）
- JDBC 日期/时间测试套件修复（Liam Stewart）
- 修复 SELECT 'id' AS xxx FROM table（Dave Cramer）
- 修复 DatabaseMetaData 以正确显示精度（Mark Lillywhite）
- 新的 getImported/getExported 键（Jason Davies）
- MD5 口令加密支持（Jeremy Wohl）
- 修复以真正使用类型缓存（Ned Wolpert）

A.9.3.18. ODBC

- 移除查询大小限制（Hiroshi）
- 移除文本字段大小限制（Hiroshi）
- 修复多字节模式下的 SQLPrimaryKeys（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）

- 多字节和性能修复（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）

A.9.3.19. ECPG

- 实现 EXECUTE ... INTO（Christof Petig）
- 多行描述符支持（例如 CARDINALITY）（Christof Petig）
- 修复 GRANT 参数（Lee Kindness）
- 修复 INITIALLY DEFERRED 缺陷
- 若干缺陷修复（Michael、Christof Petig）
- 指示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
- 所有具有外部链接的函数名现在都以 ECPG 为前缀
- 结构体数组的修复（Michael）

A.9.3.20. 其他接口

- 修复 Python fetchone()（Gerhard Haring）
- 在适当处在 Tcl 中使用 UTF、Unicode（Vsevolod Lobko、Reinhard Max）
- 添加 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）

A.9.3.21. 构建和安装

- configure、动态加载器和共享库修复（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）
- 修复 Windows 套接字通信失败（Magnus、Mikhail Terekhov）
- Hurd 编译修复（Oliver Elphick）
- BeOS 修复（Cyril Velter）

- 移除 configure --enable-unicode-conversion，现在由多字节启用（Tatsuo）
- AIX 修复（Tatsuo、Andreas）
- 修复并行 make（Peter E）
- 将 SQL 语言手册页安装到操作系统特定的目录（Peter E）
- 将 config.h 重命名为 pg_config.h（Peter E）
- 重组头文件的安装布局（Peter E）

A.9.3.22. 源代码

- 移除 SEP_CHAR（Bruce）
- 新的 GUC 钩子（Tom）
- 合并 GUC 与命令行处理（Marko Kreen）
- 移除 EXTEND INDEX（Martijn van Oosterhout、Tom）
- 缩进 Java 代码的新 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() 替换 strcasecmp()（Peter E）
- Dynahash 可移植性改进（Tom）
- 在自旋锁结构中添加 'volatile' 的使用
- 改进信号处理逻辑（Tom）

A.9.3.23. Contrib

- 新的 contrib/rtree_gist（Oleg Bartunov、Teodor Sigaev）
- 新的 contrib/tsearch 全文索引（Oleg、Teodor Sigaev）
- 添加 contrib/dblink 用于远程数据库访问（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）
- 新的 contrib/fuzzystrmatch，合并了 levenshtein、metaphone 和 soundex（Joe Conway）
- 为 contrib/intarray 添加布尔查询、二分查找和修复（Oleg Bartunov）
- 新的 pg_upgrade 工具（Bruce）

- 为 pg_resetxlog 添加新选项（Bruce、Tom）

A.10. 版本 7.1.3

发行日期
2001-08-15

A.10.1. 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。

A.10.2. 变更

移除大事务的无用 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 修复视图、用户定义类型上的 GRANT/REVOKE/注释（Tom）
修复带 DISTINCT ON 或 LIMIT 的子选择（Tom）
BeOS 修复
禁止对视图执行 COPY TO/FROM（Tom）
Cygwin 构建（Jason Tishler）

A.11. 版本 7.1.2

发行日期
2001-05-11

本次发布包含对 7.1.1 的一项修复。

A.11.1. 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。

A.11.2. 变更

修复 PL/pgSQL SELECT 未返回行时的问题
修复 psql 反斜杠命令核心转储
引用完整性权限修复
优化器修复
pg_dump 清理

A.12. 版本 7.1.1

发行日期
2001-05-05

本次发布包含对 7.1 的多方面修复。

A.12.1. 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。

A.12.2. 变更

修复 numeric 的 MODULO 操作符 (Tom)
pg_dump 修复 (Philip)
pg_dump 可以转储 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 处理带空格选项字符串的问题 (Peter E)
ODBC 修复 (Hiroshi)
EXTRACT 现在可以接受字符串参数 (Thomas)
Python 修复 (Darcy)

A.13. 版本 7.1

发行日期
2001-04-13

此发行版的重点是移除 PostgreSQL 代码中存在多年的限制。

此发行版的主要变更：

预写日志（WAL）

为了在操作系统崩溃时保持数据库一致性，以前的 PostgreSQL 发行版强制在每次事务提交之前将所有数据修改写入磁盘。有了 WAL，只需将一个日志文件刷写到磁盘，从而大大改进了性能。如果你在以前的发行版中使用 -F 来禁用磁盘刷写，现在可以考虑停止使用它。

TOAST

TOAST —— 以前的发行版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。

外连接

我们现在支持外连接。用于模拟外连接的 UNION/NOT IN 变通方法不再需要。我们使用 SQL92 外连接语法。

函数管理器

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，
但将来可能需要重写它们以使用新的函数管理器调用接口。

复杂查询

大量以前发行版不支持的复杂查询现在可以工作了。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。默认情况下现在会访问继承表。FROM 中的子查询现在受支持。

A.13.1. 迁移到版本 7.1

希望从任何以前的发行版迁移数据的用户，需要使用 pg_dump 进行转储/恢复。

A.13.2. 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罕见的游标崩溃 (Tom)
修复已回滚事务中的 DROP TABLE/INDEX (Hiroshi)
修复启用 MULTIBYTE 时 psql \l+ 的崩溃 (Peter E)
修复 CREATE VIEW 期间规则名被截断的问题 (Ross Reedstrom)
修复 PL/perl (Alex Kapranoff)
禁止对视图执行 LOCK (Mark Hollomon)
禁止对视图执行 INSERT/UPDATE/DELETE (Mark Hollomon)
禁止对视图执行 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)
修复已中止事务期间解析器的失败 (Tom)
允许临时关系正确清理索引 (Bruce)
修复 VACUUM 在同一页内移动行的问题 (Tom)
修改 pg_dump 以更好地处理 template1 中的用户自定义项 (Philip)

允许在视图中使用 LIMIT (Tom)
要求游标 FETCH 遵守 LIMIT (Tom)
允许在继承列上定义 PRIMARY/FOREIGN KEY (Stephan)
允许子查询中的 ORDER BY、LIMIT (Tom)
允许在 CREATE RULE 中使用 UNION (Tom)
使 ALTER/DROP TABLE 可回滚 (Vadim、Tom)
将 initdb 的排序规则存入 pg_control，使排序规则无法被更改 (Tom)
修复带规则的 INSERT...SELECT (Tom)
修复视图和子选择内部的 FOR UPDATE (Tom)
修复 OVERLAPS 操作符使其在 NULL 处理上符合 SQL92 规范 (Tom)
修复 lpad() 和 rpad() 以处理长度小于输入字符串的情况 (Tom)
修复某些规则中 NOTIFY 的使用 (Tom)
全面翻修 btree 代码 (Tom)
修复 PL/pgSQL 变量中 NOT NULL 的使用 (Tom)
全面翻修 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)
允许表中有多列 SERIAL (Tom)
防止偶发的索引损坏 (Vadim)

增强

添加外连接 (Tom)
函数管理器全面翻修 (Tom)
允许对索引执行 ALTER TABLE RENAME (Tom)
改进 CLUSTER (Tom)
改进更多平台上的 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的新的警告代码 (Bruce)
全面翻修 initdb 流程 (Tom、Peter E)
继承表的全面翻修；默认情况下现在会访问继承表；
  新的 ONLY 关键字可阻止这一点 (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)
全面翻修内存管理器上下文 (Tom)
pg_dumpall 使用 CREATE USER 或 CREATE GROUP 而不是 COPY (Peter E)
全面翻修 pg_dump (Philip Warner)
允许 pg_hba.conf 辅助口令文件只指定用户名 (Peter E)
创建临时表时允许使用 TEMPORARY 或 TEMP 关键字 (Bruce)
新的内存泄漏检查器 (Karel)
新的 SET SESSION CHARACTERISTICS (Thomas)
允许嵌套块注释 (Thomas)
添加 WITHOUT TIME ZONE 类型限定符 (Thomas)
新的 ALTER TABLE ADD CONSTRAINT (Stephan)
整数聚合使用 NUMERIC 累加器 (Tom)
全面翻修聚合代码 (Tom)
新的 VARIANCE 和 STDDEV() 聚合
改进 pg_dump 的依赖排序 (Philip)

新的 pg_restore 命令 (Philip)
新的 pg_dump tar 输出选项 (Philip)
新的大对象 pg_dump (Philip)
LIKE 的新 ESCAPE 选项 (Thomas)
新的不区分大小写的 LIKE — ILIKE (Thomas)
允许函数索引使用二进制兼容类型 (Tom)
允许 SQL 函数在更多上下文中使用 (Tom)
新的 pg_config 工具 (Peter E)
新的 PL/pgSQL EXECUTE 命令，允许动态 SQL 和工具语句 (Jan)
用于 SPI 值访问的新的 PL/pgSQL GET DIAGNOSTICS 语句 (Jan)
新的 quote_identifiers() 和 quote_literal() 函数 (Jan)
新的 ALTER TABLE table OWNER TO user 命令 (Mark Hollomon)
允许 FROM 中的子选择，即 FROM (SELECT ...) [AS] alias (Tom)
更新 PyGreSQL 到 3.1 版 (D'Arcy)
表存储为以 OID 命名的文件 (Vadim)
新的 SQL 函数 setval(seq,val,bool) 供 pg_dump 使用 (Philip)
要求用 DROP VIEW 删除视图，不能用 DROP TABLE (Mark)
允许 DROP VIEW view1, view2 (Mark)
允许 DROP INDEX、DROP RULE 和 DROP TYPE 中有多个对象 (Tom)
允许与 Unicode 的自动双向转换 (Tatsuo、Eiji)
新的 /contrib/pgcrypto 哈希函数 (Marko Kreen)
新的 pg_dumpall --globals-only 选项 (Peter E)
用于 WAL 的新的 CHECKPOINT 命令，它创建新的 WAL 日志文件 (Vadim)
新的 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)
将 MySQL 模式转储转换为 SQL92 和 PostgreSQL 的新工具 (Thomas)
新的 /contrib/rserv 复制工具包 (Vadim)
COPY BINARY 的新文件格式 (Tom)
新的 /contrib/oid2name，将数字文件映射为表名 (B Palmer)
新的 "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），新的 -l（日志）选项
为 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)
新的 int2 和 int8 比较操作符 (Tom)
新的 BIT 和 BIT VARYING 类型 (Adriaan Joubert、Tom、Peter E)
由于 TOAST，CHAR() 不再比 VARCHAR() 快 (Tom)
新的 GIST seg/cube 示例 (Gene Selkov)
改进的 round(numeric) 处理 (Tom)
修复 CIDR 输出格式 (Tom)
新的 CIDR abbrev() 函数 (Tom)

性能

预写日志 (WAL) 以更低的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)
将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

新的函数管理器调用约定 (Tom)
SGI 可移植性修复 (David Kaelbling)
新的 configure --enable-syslog 选项 (Peter E)
新的 BSDI README (Bruce)
configure 脚本移到顶层，不在 /src 中 (Peter E)
Makefile/配置/编译全面翻修 (Peter E)
新的 configure --with-python 选项 (Peter E)
Solaris 清理 (Peter E)
全面翻修 /contrib Makefile (Karel)
新的 OpenSSL 配置选项 (Magnus、Peter E)
AIX 修复 (Andreas)
QNX 修复 (Maurizio)
新的 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)
将 ichar() 更名为 chr() (Karel)
btrim()、ascii()、chr()、repeat() 的新文档 (Karel)
NT/Cygwin 修复 (Pete Forman)
AIX 移植修复 (Andreas)
新的 BeOS 移植 (David Reid、Cyril Velter)
将校对者的修改添加到文档 (Addison-Wesley、Bruce)
新的 Alpha 自旋锁代码 (Adriaan Joubert、Compaq)
UnixWare 移植全面翻修 (Peter E)
新的 Darwin/MacOS X 移植 (Peter Bierman、Bruce Hartzler)
新的 FreeBSD Alpha 移植 (Alfred)
全面翻修共享内存段 (Tom)
添加 IBM S/390 支持 (Neale Ferguson)
将 macmanuf 移到 /contrib (Larry Rosenman)
syslog 改进 (Larry Rosenman)
新的不含用户添加内容的 template0 数据库 (Tom)
新的 /contrib/cube 和 /contrib/seg GIST 示例代码 (Gene Selkov)
允许使用 NetBSD 的 libedit 代替 readline (Peter)
改进的汇编语言源代码格式 (Bruce)
新的 contrib/pg_logger
createdb 的新 --template 选项
新的 contrib/pg_control 工具 (Oliver)
新的 FreeBSD 工具 ipc_check、start-scripts/freebsd

A.14. 版本 7.0.3

发行日期
2000-11-11

本次发布包含对 7.0.2 的多方面修复。

A.14.1. 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。

A.14.2. 变更

Jdbc 修复 (Peter)
大对象修复 (Tom)
修复 COPY WITH OIDS 的内存泄漏 (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)
修复罕见情况下后端退出时中止事务的问题 (Tom)
修复启用多字节时的 psql \l+ (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)
修复用户认证代码中的内存分配问题 (Tom)
移除对 int4out() 的错误使用 (Tom)
修复 COALESCE 或 BETWEEN 中多个子查询的问题 (Tom)
修复某些情况下堆打开时触发器的失败 (Jeroen van Vianen)
修复不等号选择率的错误 (Tom)
修复 strcmp() 的错误使用 (Tom)
修复存储管理器访问文件末尾之后条目的缺陷 (Tom)
修复以在所有 smgr elog 消息中包含内核 errno 消息 (Tom)
修复构建时 '.' 不在 PATH 中的问题 (SL Baur)
修复文件描述符耗尽错误 (Tom)
修复以使 pg_dump 转储函数的 'iscachable' 标志 (Tom)
修复 Append 节点目标列表中的子选择 (Tom)
修复归并连接计划 (Tom)
修复 TRUNCATE 在带索引关系上的失败 (Tom)
避免写错误时整个数据库重启 (Hiroshi)
修复 nodeMaterial 以通过重算输出来遵守 chgParam (Tom)
修复行版本的源和目标位于同一页时 VACUUM 移动更新行版本链的问题 (Tom)
修复 user.c 的 CommandCounterIncrement (Tom)
修复 to_char() 中 AM/PM 边界的问题 (Karel Zak)
修复 TIME 聚合处理 (Tom)
修复 to_char() 以避免在 NULL 输入时核心转储 (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)
修复中止时后端的崩溃 (Tom)

A.15. 版本 7.0.2

发行日期
2000-06-05

这是对 7.0.1 的重新打包，并添加了文档。

A.15.1. 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。

A.15.2. 变更

向 tarball 添加了文档。

A.16. 版本 7.0.1

发行日期
2000-06-01

这是针对 7.0 的清理发行版。

A.16.1. 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。

A.16.2. 变更

修复许多 CLUSTER 失败 (Tom)
允许 ALTER TABLE RENAME 用于索引 (Tom)
修复 plpgsql 以处理 datetime->timestamp 和 timespan->interval (Bruce)
新的 configure --with-setproctitle 开关以使用 setproctitle() (Marc、Bruce)
修复 6.5.3 以来 ResultSet 的差一错误，以及更多修复。
jdbc ResultSet 修复 (Joseph Shraibman)
优化器调整 (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 的锁级别 (Tom)
更改 libpqeasy 以使用 PQconnectdb() 风格的参数 (Bruce)
修复 pg_dump 以处理 OID 索引 (Tom)
修复小的内存泄漏 (Tom)
Solaris 上 createdb/dropdb 的修复 (Tatsuo)
修复非阻塞连接 (Alfred Perlstein)
修复 RENAME TABLE 失败后的不当恢复 (Tom)
安装时将 pg_ident.conf.sample 复制到 /lib 目录 (Bruce)
添加 SJIS UDC (NEC selection IBM 汉字) 支持 (Eiji Tokuya)
修复过长的 syslog 消息 (Tatsuo)
修复带引号的过长索引的问题 (Tom)

JDBC ResultSet.getTimestamp() 修复 (Gregory Krasnow & Floyd Marinescu)
ecpg 变更 (Michael)

A.17. 版本 7.0

发行日期
2000-05-08

此发行版包含许多方面的改进，展示了 PostgreSQL 的持续成长。7.0 中的改进和修复比以往任何发行版都多。开发者们相信这是迄今最好的发行版；我们尽力只发布稳固的发行版，
这一版也不例外。

此发行版的主要变更：

外键

外键现已实现，PARTIAL MATCH 外键除外。许多用户一直在请求这一特性，我们很高兴能提供它。

优化器全面翻修

延续一年前开始的工作，优化器得到了改进，能够更好地选择查询计划，
性能更快且内存使用更少。

更新的 psql

我们的交互式终端监视器 psql 已经更新，增加了多种新特性。详情参见 psql 手册页。

连接语法

现在支持 SQL92 连接语法，不过此发行版中仅支持 INNER JOIN。JOIN、NATURAL JOIN、JOIN/USING 和 JOIN/ON 都可用，列相关名也可用。

A.17.1. 迁移到版本 7.0

希望从任何以前的 PostgreSQL 发行版迁移数据的用户，需要使用 pg_dump 进行转储/ 恢复。对于从 6.5.* 升级的用户，也可以改用 pg_upgrade 升级到此发行版；不过，完整的转储/重载安装始终是升级最稳健的方法。

新发行版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 定义的类型 timestamp 和 interval 取代。尽管已经做了一些工作来简化过渡 —— 允许 PostgreSQL 识别已废弃的类型名并将它们转换为新类型名 —— 但这一机制对你现有的应用程序可能无法完全透明。
- 优化器在查询代价估算方面得到了实质性改进。在某些情况下，
由于优化器为首选计划做出了更好的选择，查询时间会缩短。但是，
在少数情况下（通常涉及病态的数据分布），你的查询时间可能上升。如果你处理大量数据，
可能需要检查你的查询以验证性能。

- JDBC 和 ODBC 接口已升级并扩展。
- 字符串函数 CHAR_LENGTH 现在是原生函数。以前的版本将它转换为对 LENGTH 的调用，这可能与其他实现 LENGTH 的类型（例如几何类型）产生歧义。

A.17.2. 变更

缺陷修复

防止函数调用超出参数数量上限 (Tom)
改进 CASE 结构 (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 中的子选择 (Tom)
禁止 INSERT ... SELECT ... ORDER BY (Tom)
修复大于 2GB 的关系（包括 vacuum）的问题
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)
修复复杂情况下临时表的处理 (Bruce、Tom)
允许在表打开时锁定表，改进并发可靠性 (Tom)
在 pg_dump 中正确为序列名加引号 (Ross J. Reedstrom)
防止他人正在访问时 DROP DATABASE
防止 GROUP BY 未处理任何行时返回行 (Tom)
修复 'SELECT COUNT(1) FROM table WHERE ...'（无匹配 WHERE 的行时） (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，以及 ;;; 行。 (Tom)
改进磁盘写入失败、磁盘满之后的恢复 (Hiroshi)
修复表在 FROM 中提到但未连接的情况 (Tom)
允许不带聚合函数的 HAVING 子句 (Tom)
修复 "--" 注释且无末尾换行的问题，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)
修复失败的 CREATE/DROP 命令导致的不一致状态 (Hiroshi)
修复带连字符的数据库名
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，使得启用 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)
禁止在事务内 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 的 'HEAP_MOVED_IN was not expected' 错误 (Tom)
修复带空格表/列的视图的问题 (Tom)
禁止对索引授权 (Peter E)
修复出错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 的若干修复
修复 numeric 类型的 log() (Tom)
废弃 ':' 和 ';' 操作符
允许清理临时表
禁止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 INSERT INTO ... SELECT 中 AS 列与结果列匹配的问题
修复 INSERT ... SELECT ... GROUP BY 按目标列而不是源列分组的问题 (Tom)
修复 CREATE TABLE test (a char(5) DEFAULT text '', b int4) 与
INSERT (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 中类型不匹配的问题
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, 当键 > 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

新的 CLI 接口头文件 sqlcli.h, 基于 SQL3/SQL98
移除查询长度的所有限制, 行长度限制仍存在 (Tom)
将 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)
修复以赋予超级用户和 createdb 用户正确的目录更新权限 (Peter E)
允许 ecpg 布尔变量取 NULL 值 (Christof)
当无 NULL 指示符的变量取 NULL 值时发出 ecpg 错误 (Christof)
允许用 ^C 取消 COPY 命令 (Massimo)
添加 SET FSYNC 和 SHOW PG_OPTIONS 命令 (Massimo)
动态加载 C 函数的函数名重载 (Frankpitt)
向 libpq++ 添加 CmdTuples() (Vince)
新的 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)
第一个真正可用的外键约束触发器功能 (Jan)
添加 FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)
添加 FOREIGN KEY ... MATCH <unspecified> 引用动作 (Don Baccus)
允许对 ctid (物理堆位置) 使用 WHERE 限制 (Hiroshi)

将 pginterface 从 contrib 移到 interface 目录并更名为 pgeasy (Bruce)
更改 pgeasy connectdb() 的参数顺序 (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 函数现在返回之前的钩子 (Peter E)
防止 PQsetNoticeProcessor 被设置为 NULL (Peter E)
使 COPY 中的 USING 变为可选 (Bruce)
允许目标列表中的子选择 (Tom)
允许比较操作符左侧的子选择 (Tom)
新的并行回归测试 (Jan)
将后端侧 COPY 写文件的权限从 666 改为 644 (Tom)
即使 PGDATA 目录已存在也强制其权限安全 (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)
libpq 新函数支持异步连接: PQconnectStart()、
  PQconnectPoll()、PQresetStart()、PQresetPoll()、PQsetenvStart()、
  PQsetenvPoll()、PQsetenvAbort (Ewan Mellor)
新的 libpq PQsetenv() 函数 (Ewan Mellor)
create/alter user 扩展 (Peter E)
$PGDATA 下新的 postmaster.pid 和 postmaster.opts (Tatsuo)
用于 create/drop user/db 的新脚本 (Peter E)
psql 全面翻修 (Peter E)
向 libpq 接口添加 const (Peter E)
新的 libpq 函数 PQoidValue (Peter E)
显示导致 GROUP BY 问题的特定非聚合 (Tom)
使对 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
将索引键最大数量增加到 16 (Bruce)
将函数参数最大数量增加到 16 (Bruce)
允许配置索引键和参数的最大数量 (Bruce)
允许无特权用户更改自己的口令 (Peter E)
启用口令认证; 新用户必需 (Peter E)
禁止删除拥有数据库的用户 (Peter E)
将 initdb 选项 --with-mb 改为 --enable-multibyte
为 initdb 添加提示输入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)
新的 pg_char_to_encoding() 和 pg_encoding_to_char() 函数 (Tatsuo)
libpq 非阻塞模式 (Alfred Perlstein)
改进未指定长度的类型转换中的类型转换
新的 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)
允许在 PL/pgSQL 中使用 SELECT .. FOR UPDATE (Hiroshi)
即使到达 EOF 也启用反向顺序扫描 (Hiroshi)
添加布尔值的 btree 索引, >= 和 <= (Don Baccus)
COPY FROM 失败时打印当前行号 (Massimo)
识别 POSIX 时区, 例如 "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 (又称相关名) (Thomas)
实现 SQL92 连接语法 (Thomas)
使保留字 INTERVAL 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)
新的 psql \encoding 选项 (Tatsuo)
允许 PQrequestCancel() 在等待锁状态下终止 (Hiroshi)
允许在所有情况下对负数取负
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)
新的 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
将 pgaccess 更新到 0.98.6
新的 SET SEED 命令
新的 pg_options.sample 文件
新的 SET FSYNC 命令 (Massimo)
允许创建表时使用 pg_descriptions
允许创建类型、列和函数时使用 pg_descriptions
允许 psql \copy 使用分隔符 (Peter E)
允许 psql 将空值打印为与 "" 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使用 (Tom)
改进 int 和 float 常量的类型转换 (Tom)
int8 输入、范围检查和类型转换的清理 (Tom)

修复 SELECT timespan('21:11:26'::time) (Tom)
netmask('x.x.x.x/0') 现在是 255.255.255.255 而不是 0.0.0.0 (Oleg
Sharoiko)
在 NUMERIC 上添加 btree 索引 (Jan)
Perl 修复包含 NUL 字符的大对象 (Douglas Thomson)
ODBC 大对象修复 (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 数组 (Peter E)
修复 NUMERIC 类型的舍入/溢出，如 NUMERIC(4,4) (Tom)
允许 NUMERIC 数组
修复 NUMERIC ceil() 和 floor() 函数中的缺陷 (Tom)
使 char_length()/octet_length 包含尾部空格 (Tom)
使 abstime/reltime 使用 int4 而不是 time_t (Peter E)
新的 lztext 数据类型用于压缩文本字段
修订处理 int 和 float 常量转换的代码 (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 进行舍入 (Tom)
正确地将 float4/8 转换为 NUMERIC (Tom)
允许与 NUMERIC 的类型转换 (Thomas)
使 ISO 日期样式 (2000-02-16 09:33) 成为默认 (Thomas)
添加 NATIONAL CHAR [VARYING] (Thomas)
允许 NUMERIC 的 round 和 trunc 接受负标度 (Tom)
新的 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)
将 float8 的函数更名为 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()
将 NUMERIC 的 power() 更名为 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 导致的指数级空间消耗 (Tom)
为系统列收集属性选择率值 (Tom)
减少聚合的内存使用 (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时的 btree 搜索 (Tom)
仅在存在索引时启用快速 LIKE 索引处理 (Tom)
重用索引页上重复项的空闲空间 (Tom)
改进哈希连接处理 (Tom)
防止结果已排序时降序排序 (Hiroshi)
允许索引扫描查询条件的交换 (Tom)
在需要 ORDER BY/GROUP BY 的情况下优先使用索引扫描 (Tom)

为提升性能，以固定大小的块分配大内存请求（Tom）
通过减少内存分配请求改进 vacuum 的性能（Tom）
实现常量表达式化简（Bernard Frankpitt、Tom）
使用次要列确定索引扫描的起点（Hiroshi）
防止内部排序时四倍占用磁盘空间（Tom）
通过调用更少的函数加快排序（Tom）
创建系统索引以匹配所有系统缓存（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）
改进优化代价估算（Tom）
改进优化器对范围查询 x > lowbound AND x < highbound 的估计（Tom）
在适当处使用 DNF 而不是 CNF（Tom、Taral）
对 OR-of-AND WHERE 子句的进一步清理（Tom）
在 OR 子句中使用索引 (x = 1 AND y = 2) OR (x = 2 AND y = 4)（Tom）
对随机索引页访问的更智能优化器计算（Tom）
新的 SET 变量以控制优化器代价（Tom）
基于 LIMIT、OFFSET 和 EXISTS 条件优化查询（Tom）
减少优化器对连接路径的内部管理以加速（Tom）
子查询的重大加速（Tom）
fsync 未禁用时更少的 fsync 写（Tom）
改进的 LIKE 优化器估计（Tom）
防止仅 SELECT 查询中的 fsync（Vadim）
使索引创建使用 psort 代码，因为它现在更快（Tom）
允许创建大于 1 Gig 的排序临时表

源代码树变更

修复 linux PPC 编译
新的通用表达式树遍历子例程（Tom）
将 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 文件中移除不需要的 #include（Bruce）
酌情将 #include 改为使用 <> 和 ""（Bruce）
启用 libpq 的 Windows 编译
来自 Uncle George 的 Alpha 自旋锁修复 <gatgul@voicenet.com>
优化器数据结构的全面翻修（Tom）
cygipc 库的修复（Yutaka Tanida）
允许 pgsql 在较新的 Cygwin 快照上工作（Dan）
新的目录版本号（Tom）
添加 Linux ARM
将 heap_replace 更名为 heap_update
为 QNX 更新（Dr. Andreas Kardos）
新的平台特定回归处理（Tom）
将 oid8 更名为 oidvector，int28 更名为 int2vector（Bruce）
将所有 yacc 和 lex 文件纳入发行版（Peter E.）
移除不再需要的 lextest（Peter E）
修复 Windows 上的 libpq 和 psql（Magnus）
内部将 datetime 和 timespan 更改为 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 编译的修复
新的多字节编码

A.18. 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 中缺失的新版 PgAccess，并安装了一个 NT 特定的修复。

A.18.1. 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。

A.18.2. 变更

更新版本的 pgaccess 0.98
NT 特定补丁
修复继承表上规则的转储

A.19. 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。

A.19.1. 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。

A.19.2. 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 的构建问题 (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)
这是为了重用 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 的 IRIX 链接修复 >yucao@falcon.kla-tencor.com<
修复 LIKE 中的逻辑错误：不应返回 LIKE_ABORT
(在到达模式末尾而文本未结束时) (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

A.20. 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。

A.20.1. 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。

A.20.2. 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）

A.21. 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 数据类型

我们现在有了真正的数字数据类型，精度由用户指定。

临时表

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。

提速

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。

移植

我们继续扩大移植列表，这次包括 Windows NT/ix86 和 NetBSD/arm32。

接口

大多数接口有了新版本，现有功能得到改进。

文档

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。

A.21.1. 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。

A.21.1.1. 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。

A.21.2. 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)
允许为包含连字符的用户名授权
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)
把新类型强制转换技术扩展到聚合 (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)

带索引已删行清理（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）
哈希代码的全面翻修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加载（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 动态加载现在可用（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）

A.22. 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。

A.22.1. 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。

A.22.2. 变更

修复某些平台上日期时间常量的问题（Thomas）

A.23. 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。

A.23.1. 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。

A.23.2. 变更

为 pg_dump 添加 -N 标志以强制在标识符周围加上双引号。这是
默认行为（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，以允许表名中包含
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）
修复大小写混合表名的多处问题（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）
修复 pg_dump 的内存泄漏、继承约束、布局更改问题
把 pgaccess 更新到 0.93
修复 64 位平台上的原型
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）
改进对 install 程序的查找（Tom）
时区修复（Tom）
HP-UX 修复（Tom）
用隐式类型强制转换匹配 DEFAULT 值（Thomas）
添加用于辅助单字节（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)

A.24. 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转换，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。

A.24.1. 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。

A.24.2. 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)

A.25. 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，该问题是 6.3.1 版本无意中引入的。
- 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。

A.25.1. 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)

A.26. 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。

A.26.1. 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
修复重定义内建函数的问题 (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
修复后端通信的大端问题 (Kataoka)
修复 SUBSTR() 和 substring() (Jan)
若干 jdbc 修复 (Peter)
libpgtcl 改进，见 libptcl/README (Randy Kunkee)
修复 "Datasize = 0" 错误 (Vadim)
防止 \do 折行 (Bruce)
移除重复的俄文字符集条目

Sunos4 清理
允许 LOCK 和 SELECT INTO 中使用可选的 TABLE 关键字 (Thomas)
CREATE SEQUENCE 选项允许负整数 (Thomas)
添加 "PASSWORD" 作为允许的列标识符 (Thomas)
添加对 UNION 目标字段的检查 (Bruce)
修复 Alpha 移植 (Dwayne Bailey)
修复包含引号的 text 数组 (Doug Gibson)
Solaris 编译修复 (Albert Chin-A-Young)
更好地识别 tcl 与 tk 库和头文件 (Bruce)

A.27. 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。
- 更好的口令认证机制。默认表权限已经更改。
- 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。这里只是无法用一句话描述的大项。仍需查阅详细变更列表。

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，你应当检查自己的代码，
看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，
子查询的用途比你想象的更多。Vadim 用子查询让我们登上了 SQL 的大舞台，而且是功能完备的子查询。子查询唯一不能做的，是用在目标列表中。

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 标准的要求。当然，你可以在表创建之后随意 GRANT 你想要的任何权限。系统表仍然可被 PUBLIC SELECT。

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，因此在重度使用期间资源饥饿应该会更少。

以前的版本中有很多关于文档不足的抱怨。Thomas 在此版本的许多新手册上投入了大量精力。请查看 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，人们以前使用的变通方法不再需要。

A.27.1. 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。

A.27.2. 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 notty 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)
修复多种锁定问题，例如较新的锁等待者先于较旧的等待者获得锁、
        有写者等待锁时读锁持有者不共享锁、以及等待的写者没有
        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)
允许在约束子句中使用多参数函数 (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)
大型对象的重要修复
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，不再依赖超时 (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)
允许对非常量使用 PostgreSQL 风格的类型转换 ("::") (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)
支持字面字符串类型强制转换的 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)
事务块内的用户查询完成后释放为其分配的内存,
　　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 表以存放表、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 Query 对继承的处理以及代价计算 (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)
新的前端/后端协议带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
降低系统列的开销 (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 命令失败时报告出错的行
允许对视图设置的权限独立于底层表。
        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

将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)
移除优化器中的部分递归并清理那里的代码 (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)
将 PostODBC 源码树纳入标准发行版 (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
在回归测试套件中添加字符串函数 (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)

A.28. 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。

A.28.1. 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在 psql 中使用以下 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。

A.28.2. 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)

A.29. 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。

A.29.1. 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。

A.29.2. 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。

A.29.3. 变更

缺陷修复

修复 pg_dump 对继承、序列、归档表的问题 (Bruce)
修复因移位、无符号和 Solaris 的错误原型
        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)
检查几何对象在端点处的相交以避免舍入问题 (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)
修复一个非常老的缺陷：命令本身可以看见
由该命令更改/插入的行（因此会导致对已更新
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 中移除不需要的信号代码
修复 OR (where x != 1 or x isnull 不返回 x NULL 的行) (Vadim)
修复 time_cmp 函数 (Vadim)
修复 WHERE 子句中对首参数为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带聚合的函数参数 (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)
纳入用于字符串处理、外连接和联合的保留字 (Thomas)
用独占状态实现扩展注释 ("/* ... */") (Thomas)
添加 "//" 单行注释 (Bruce)
移除对操作符名称中字符的一些限制 (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)
将距离操作符 "<===>" 替换为 "<->" (Thomas)
把"上方"操作符 "!^" 替换为 ">^"，"下方"操作符 "!|" 替换为 "<^" (Thomas)
添加两端修剪文本、子串和字符串位置的例程 (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)
允许内部相同的类型上的函数和操作符成功执行 (Bruce)

在性能分析后加速后端启动 (Bruce)
为性能内联频繁调用的函数 (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)
用于引用完整性的通用触发器函数 (Vadim)
用于时间旅行的通用触发器函数 (Vadim)
用于 AUTOINCREMENT/IDENTITY 特性的通用触发器函数 (Vadim)
MOVE 实现 (Vadim)

源码树变更

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)
废弃几何代码中临时不存储的斜率字段 (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)

A.30. 版本 6.1.1

发行日期
1997-07-22

A.30.1. 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。

A.30.2. 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)

A.31. 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备选优化器。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 “随机” 结果本应导致 “random” 测试 “失败”，因为回归测试是用简单的 diff 评估的。不过，在我的测试机（Linux/gcc/i686）上，“random” 似乎并不会产生随机结果。

A.31.1. 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。

A.31.2. 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 WIN32 专用代码 (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

A.32. 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。

A.32.1. 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。

A.32.2. 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。

A.32.3. 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

A.33. 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 版本中。

A.34. 版本 1.02

发行日期
1996-08-01

A.34.1. 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. 将 1.02.1 新增的内建函数和操作符添加到 1.01 或 1.02 数据库中。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，并应用本文件末尾附带的查询。这可以通过 `psql` 轻松完成。如果你的 1.01 或 1.02 数据库名为 `testdb`，并且你已把命令从本文件末尾剪出并保存到 `addfunc.sql` 中：

```
% psql testdb -f addfunc.sql
```

从 1.02 升级的用户在执行文件中的最后两条语句时会得到警告，因为它们已经存在于 1.02 中。这无需担心。

A.34.2. 转储/重载程序

如果你试图重新装载由旧版本生成的 pg_dump 或文本模式 `copy tablename to stdout` 输出，需要先对 ASCII 文件运行附带的 `sed` 脚本，再将其装入数据库。旧格式用 '.' 作为数据结束标记，而现在的数据结束标记是 '\.'。此外，空字符串现在装载为 '' 而非 NULL。完整细节见 copy 手册页。

```
sed 's/^\.$/\\./g' <in_file >out_file
```

如果你装载的是较旧的二进制 copy 或非 stdout copy，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);
```

A.34.3. 变更

源代码维护与开发
 * 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` 现在可以输出模式和/或数据，并有许多修复
   以提高完整性。
 * 管理 `shell` 脚本中用 `psql` 代替 `monitor`。
   `monitor` 将在下一个发行版中废弃。
 * 日期/时间函数得到增强
 * `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（多到几乎无法列举）
 * 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

A.35. 版本 1.01

发行日期
1996-02-23

A.35.1. 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，OIDNAMELEN 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 src/Makefile.global 中的这一行：

      ```
      HBA = 1
      ```

      注意，基于主机的认证默认开启，如果你不执行上面的 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. 将 1.01 新增的内建函数和操作符添加到 1.0 数据库中。方法是用新的 1.01 服务器运行你自己的 1.0 数据库，并应用随附保存在文件 1.0_to_1.01.sql 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（从此处剪切并粘贴）：

   ```
   -- 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);
```

A.35.2. 变更

不兼容之处：
 * 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
 * 添加了 ASC 和 DESC 关键字
 * 为 CREATE FUNCTION 添加 'internal' 作为可能的语言
   内部函数是已静态链接
   到 postgres 后端中的 C 函数。
 * 为系统标识符（表名、
   属性名等）添加了新类型 "name"。它取代了旧的 char16 类型。
   name 的长度由 src/Makefile.global 中的 NAMEDATALEN #define 设置
 * 一本描述查询语言的易读参考手册。
 * 添加了基于主机的访问控制。配置文件（$PGDATA/pg_hba）
   用于保存配置数据。如果不想要基于主机的访问控制，
   请注释掉 src/Makefile.global 中的 HBA=1。
 * 更改正则表达式处理为统一使用 Henry Spencer 的正则代码，
   与平台无关。正则代码包含在发行版中
 * 添加了用于不区分大小写正则表达式的函数和操作符。
   操作符为 ~* 和 !~*。
 * pg_dump 使用 COPY 而非 SELECT 循环以获得更好性能

缺陷修复：
 * 修复了一个优化器缺陷，当在 WHERE 子句的比较中使用
   函数调用时会导致核心转储
 * 将所有 getuid 的使用改为 geteuid，以使用有效 uid
 * psql 使用 -c 时出错现在返回非零状态
 * 应用了公开补丁 1-14

A.36. 版本 1.0

发行日期
1995-09-05

A.36.1. 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。

不兼容之处：
* 日期格式必须为 MM-DD-YYYY（如果使用
欧洲风格，则为 DD-MM-YYYY）。这遵循 SQL-92 规范。

* "delimiters" 现在是一个关键字

增强：
* 添加了 SQL LIKE 语法
* copy 命令现在接受可选的 USING DELIMITER 说明。
分隔符可以是任何单字符字符串。
* 添加了 IRIX 5.3 移植。
感谢 Paul Walmsley 等人。
* 更新 pg_dump 以配合新 libpq 工作
* psql 中添加了 \d
感谢 Keith Parks
* 对于使用 POSIX 正则的体系结构，正则性能
通过缓存预编译模式得到了改进。
感谢 Alistair Crooks
* 新版本的 libpq++
感谢 William Wanders

缺陷修复：
* 可以在 createuser 脚本中指定任意用户 id
* psql 中连接其他数据库的 \c 现在可以工作了。
* 修复了 float4inc() 的错误 pg_proc 条目
* 设置了 usecreatedb 字段的用户现在可以创建数据库，
而无需是 usesuper
* 当条目不再有任何权限时移除访问控制条目
* 修复了不可移植的 datetimes 实现
* 在 src/backend/Makefile 中添加 kerberos 标志
* libpq 现在支持 kerberos
* 用户手册中的排印错误已更正。
* 多索引的 btree 从未工作过，现在尝试使用时会
明确告知它们不可用

A.37. Postgres95 版本 0.03

发行日期
1995-07-21

A.37.1. 变更

不兼容变更：
 * BETA-0.3 与用以前版本创建的数据库不兼容
   （由于系统目录更改和索引结构更改）。
 * 双引号（"）作为字符串字面量的引号字符已被废弃；
   你需要将它们转换为单引号（'）。
 * 聚合的名称（如 int4sum）已按照
   SQL 标准重命名（如 sum）。
 * CHANGE ACL 语法被 GRANT/REVOKE 语法取代。
 * 浮点字面量（如 3.14）现在是 float4 类型（而不是以前版本的
   float8）；如果你依赖它
   是 float8 类型，可能需要进行类型转换。如果忽略类型转换而把

浮点字面量赋给 float8 类型的字段，可能会得到错误的
存储值!
* LIBPQ 经过彻底翻新，前端应用程序
可以连接多个后端
* pg_user 中的 usesysid 字段已从 int2 改为 int4，以
允许更宽范围的 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 标准的更改）：
* 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 text 类型。此外，
char(N) 会做空白填充。
* 用单引号（'）引用字符串字面量；''（除
\' 外）也可用于在字符串中插入单引号
* 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
接受用户自定义类型的 MAX 聚合。）
* 移除 CHANGE ACL。添加 GRANT/REVOKE 语法。
- 可以使用 "GROUP" 关键字把权限授予组。
例如：

```
GRANT SELECT ON foobar TO GROUP my_group;
```

关键字 'PUBLIC' 也受支持，表示所有用户。

每次只能对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权限。
你必须显式授予用户插入/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：
* 空表上的聚合不被运行的缺陷已修复。现在，
在空表上运行的聚合将返回聚合的初始条件。因此，空表的 COUNT 现在会正确返回 0。
空表的 MAX/MIN 将返回值为 NULL 的元组。
* 允许在 monitor 中使用 \;
* LISTEN/NOTIFY 异步通知机制现在可以工作
* 规则动作体中的 NOTIFY 现在可以工作
* 哈希索引可以工作了，访问方法总体上性能应该更好。
创建大型 btree 索引应该快得多。（感谢 Paul
Aoki）

其他更改与增强：

 * 添加了用于解释查询执行计划的 EXPLAIN 语句（例如 "EXPLAIN SELECT * FROM
 EMP" 会打印出该查询的执行计划）。
 * 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 */
 * 访问控制被违反时，会给出消息
        "Either no such class or insufficient privilege"。
   这与找不到类时返回的消息相同。这可以阻止无权限的用户
   猜测受权限保护的类的存在。
 * 还做了一些用户不可见的
   系统目录更改。

libpgtcl 更改：
 * "pg_result" tcl 命令添加了 -oid 选项。
   pg_result -oid 返回最后插入元组的 oid。如果最后的命令不是 INSERT，则 pg_
result -oid 返回 ""。
 * 大对象接口以 pg_lo* tcl 命令的形式提供：
   pg_lo_open, pg_lo_close, pg_lo_creat, etc.

可移植性增强与新移植：
 * flex/lex 的问题已经解决。现在，在任何平台上都应该可以用 flex 代替 lex。
   我们不再根据你所用的平台假设你使用的词法分析器。
 * 现在支持 Linux-ELF 移植。已测试多种配置：已知以下配置可以工作：
        kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
   所有内容均为 ELF 格式，

新实用程序：
 * ipcclean 已加入发行版
   通常不需要运行 ipcclean，但如果你的后端崩溃
   并留下共享内存段，ipcclean 会为你清理它们。

新文档：
 * 用户手册已经修订并添加了 libpq 文档。

A.38. Postgres95 版本 0.02

发行日期
1995-05-25

A.38.1. 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 - 把 postgres 数据库转储为包含查询命令的

脚本文件的工具。脚本文件为 ASCII 格式，可用于重建数据库，甚至在其他机器和其他体系结构上。（也适合把 Postgres 4.2 数据库转换为 Postgres95 数据库。）

以下移植已并入 postgres95-beta-0.02：
 * Alistair Crooks 的 NetBSD 移植
 * Mike Tung 的 AIX 移植
 * Jon Forrest 的 Windows NT 移植（内容更多但尚未完成）
 * Brian Gallew 的 Linux ELF 移植

postgres95-beta-0.02 中修复了以下缺陷：
 * COPY OUT 中换行未转义以及首属性为 '.' 时 COPY OUT 的问题
 * createuser 中无法键入回车以使用默认用户 id
 * 大表上的 SELECT DISTINCT 崩溃
 * Linux 安装问题
 * monitor 不允许使用 'localhost' 作为 PGHOST
 * psql 执行 \c 或 \l 时核心转储
 * src/bin/pgtclsh/Makefile 中缺少 "pgtclsh" 目标
 * libpgtcl 有硬编码的默认端口号
 * SELECT DISTINCT INTO TABLE 挂起
 * CREATE TYPE 不接受 'variable' 作为 internallength
 * 在 SELECT 中使用多个聚合时结果错误

A.39. Postgres95 版本 0.01

发行日期
1995-05-01

初始发行版。

PostgreSQL 7.2.8 程序员指南

PostgreSQL 全球开发组

PostgreSQL 7.2.8 程序员指南

PostgreSQL 全球开发组

法律声明

目录

I. 客户端接口 1
1. libpq - C 库 5
1.1. 简介 5
1.2. 数据库连接函数 5
1.3. 命令执行函数 11
1.4. 异步查询处理 15
1.5. 快速路径接口 18
1.6. 异步通知 18
1.7. 与 COPY 命令相关的函数 19
1.8. libpq 跟踪函数 21
1.9. libpq 控制函数 21
1.10. 环境变量 21
1.11. 线程行为 22
1.12. 构建 Libpq 程序 22
1.13. 示例程序 24
2. 大对象 33
2.1. 简介 33
2.2. 实现特性 33
2.3. 接口 33
2.4. 服务器端内建函数 35
2.5. 从 Libpq 访问大对象 35
3. libpq++ - C++ 绑定库 41
3.1. 简介 41
3.2. 控制与初始化 41
3.3. libpq++ 类 42
3.4. 数据库连接函数 42
3.5. 查询执行函数 43
3.6. 异步通知 46
3.7. 与 COPY 命令相关的函数 46
4. pgtcl - Tcl 绑定库 48
4.1. 简介 48
4.2. 把 pgtcl 装载进你的应用程序 49
4.3. pgtcl 命令参考信息 49
5. libpgeasy - 简化 C 库 69
6. ecpg - C 中的嵌入式 SQL 70
6.1. Why Embedded SQL? 70
6.2. 概念 70
6.3. How To Use ecpg 70
6.4. Limitations 73
6.5. 从其他 RDBMS 软件包移植 74
6.6. 给开发者 74
7. ODBC 接口 79
7.1. 简介 79
7.2. 安装 79
7.3. 配置文件 80
7.4. Windows应用 81
7.5. ApplixWare 82
8. JDBC 接口 86
8.1. 设置 JDBC 驱动 86
8.2. 使用驱动 87
8.3. 发出查询并处理结果 89

8.4. 执行更新 90
8.5. 创建和修改数据库对象 90
8.6. 存储二进制数据 91
8.7. PostgreSQL 对 JDBC API 的扩展 93
8.8. 在多线程（multi-threaded）或 servlet 环境中使用驱动 114
8.9. 延伸阅读 114
9. PyGreSQL - Python 接口 115
9.1. pg 模块 115
9.2. pg 模块函数 116
9.3. 连接对象：pgobject 128
9.4. 数据库包装类：DB 141
9.5. 查询结果对象：pgqueryobject 151
9.6. 大对象：pglarge 157
9.7. DB-API 接口 167
II. 服务器编程 168
10. 体系结构 171
10.1. PostgreSQL 体系结构概念 171
11. 扩展 SQL：概览 174
11.1. 可扩展性如何运作 174
11.2. PostgreSQL 类型系统 174
11.3. 关于 PostgreSQL 系统目录 174
12. 扩展 SQL：函数 177
12.1. 简介 177
12.2. 查询语言（SQL）函数 177
12.3. 过程语言函数 181
12.4. 内部函数 181
12.5. C 语言函数 181
12.6. 函数重载 193
12.7. 过程语言处理器 194
13. 扩展 SQL：类型 197
14. 扩展 SQL：操作符 199
14.1. 简介 199
14.2. 示例 199
14.3. 操作符优化信息 199
15. 扩展 SQL：聚合 204
16. 规则系统 206
16.1. 简介 206
16.2. 什么是查询树? 206
16.3. 视图和规则系统 208
16.4. INSERT、UPDATE 和 DELETE 上的规则 215
16.5. 规则和权限 225
16.6. 规则与触发器 225
17. 索引扩展接口 228
17.1. 简介 228
17.2. 访问方法 228
17.3. 访问方法策略 229
17.4. 访问方法支持例程 229
17.5. 操作符类 230
17.6. 创建操作符和支持例程 230
18. 索引代价估计函数 234
19. GiST 索引 237
20. 触发器 239
20.1. 触发器创建 239

20.2. 与触发器管理器的交互 240
20.3. 数据更改的可见性 242
20.4. 示例 242
21. 服务器编程接口 246
21.1. 接口函数 246
21.2. 接口支持函数 259
21.3. 内存管理 267
21.4. 数据更改的可见性 279
21.5. 示例 279
III. 过程语言 283
22. 过程语言 286
22.1. 简介 286
22.2. 安装过程语言 286
23. PL/pgSQL - SQL 过程语言 288
23.1. 概述 288
23.2. PL/pgSQL的结构 290
23.3. 声明 291
23.4. 表达式 293
23.5. 基本语句 294
23.6. 控制结构 298
23.7. 游标 302
23.8. 错误和消息 304
23.9. 触发器过程 305
23.10. 示例 307
23.11. 从 Oracle PL/SQL 移植 308
24. PL/Tcl - Tcl 过程语言 320
24.1. 概述 320
24.2. 描述 320
25. PL/Perl - Perl 过程语言 326
25.1. 概述 326
25.2. 构建和安装 PL/Perl 326
25.3. 描述 327
26. PL/Python - Python 过程语言 330
26.1. 简介 330
26.2. 安装 330
26.3. 使用 PL/Python 330

插图清单

10.1. 如何建立连接 172
11.1. PostgreSQL 的主要系统目录 175

表格清单

4.1. pgtcl 命令 48
11.1. 目录 174
12.1. 等价 C 类型 183
17.1. 索引访问方法模式 228
17.2. B-树 229
23.1. 单引号转义表 309

范例清单

1.1. libpq示例程序 1 24
1.2. libpq示例程序 2 26
1.3. libpq 示例程序 3 28
2.1. 用 Libpq 操作大对象的示例程序 36
4.1. pgtcl 示例程序 48
8.1. 在 JDCB 中处理一个简单查询 89
8.2. 简单删除示例 90
8.3. 删除表示例 90
8.4. 二进制数据示例 91
22.1. PL/pgSQL 的手工安装 287
23.1. 一个PL/pgSQL触发器过程示例 306
23.2. 一个让整数加一的简单PL/pgSQL函数 307
23.3. 一个连接文本的简单PL/pgSQL函数 307
23.4. 一个操作复合类型的PL/pgSQL函数 307
23.5. 一个简单的函数 309
23.6. 一个创建另一个函数的函数 310
23.7. 一个带大量字符串操作和 OUT 参数的过程 311

部分 I. 客户端接口

手册的这一部分描述各种语言的客户端编程接口和支持库。

目录

1. libpq - C 库 5
    1.1. 简介 5
    1.2. 数据库连接函数 5
    1.3. 命令执行函数 11
        1.3.1. 主要例程 11
        1.3.2. 用于在 SQL 查询中嵌入字符串的转义 12
        1.3.3. 用于在 SQL 查询中嵌入二进制串的转义 12
        1.3.4. 检索 SELECT 结果信息 13
        1.3.5. 检索 SELECT 结果值 14
        1.3.6. 检索非 SELECT 结果信息 15
    1.4. 异步查询处理 15
    1.5. 快速路径接口 18
    1.6. 异步通知 18
    1.7. 与 COPY 命令相关的函数 19
    1.8. libpq 跟踪函数 21
    1.9. libpq 控制函数 21
    1.10. 环境变量 21
    1.11. 线程行为 22
    1.12. 构建 Libpq 程序 22
    1.13. 示例程序 24
2. 大对象 33
    2.1. 简介 33
    2.2. 实现特性 33
    2.3. 接口 33
        2.3.1. 创建一个大对象 34
        2.3.2. 导入一个大对象 34
        2.3.3. 导出一个大对象 34
        2.3.4. 打开一个现有的大对象 34
        2.3.5. 向大对象写入数据 34
        2.3.6. 从大对象读取数据 34
        2.3.7. 在大对象上定位 35
        2.3.8. 关闭一个大对象描述符 35
        2.3.9. 移除一个大对象 35
    2.4. 服务器端内建函数 35
    2.5. 从 Libpq 访问大对象 35
3. libpq++ - C++ 绑定库 41
    3.1. 简介 41
    3.2. 控制与初始化 41
        3.2.1. 环境变量 41
    3.3. libpq++ 类 42
        3.3.1. 连接类：`PgConnection` 42
        3.3.2. 数据库类：`PgDatabase` 42
    3.4. 数据库连接函数 42
    3.5. 查询执行函数 43
        3.5.1. 主例程 43
        3.5.2. 获取 SELECT 结果信息 43
        3.5.3. 获取 SELECT 结果值 44
        3.5.4. 获取非 SELECT 结果信息 45
    3.6. 异步通知 46
    3.7. 与 COPY 命令相关的函数 46
4. pgtcl - Tcl 绑定库 48

4.1. 简介 48
4.2. 把 pgtcl 装载进你的应用程序 49
4.3. pgtcl 命令参考信息 49
5. libpgeasy - 简化 C 库 69
6. ecpg - C 中的嵌入式 SQL 70
6.1. Why Embedded SQL? 70
6.2. 概念 70
6.3. How To Use ecpg 70
6.3.1. Preprocessor 70
6.3.2. Library 70
6.3.3. 错误处理 71
6.4. Limitations 73
6.5. 从其他 RDBMS 软件包移植 74
6.6. 给开发者 74
6.6.1. The Preprocessor 74
6.6.2. 一个完整的例子 77
6.6.3. 库 77
7. ODBC 接口 79
7.1. 简介 79
7.2. 安装 79
7.3. 配置文件 80
7.4. Windows应用 81
7.4.1. 编写应用 81
7.5. ApplixWare 82
7.5.1. 配置 82
7.5.2. 常见问题 83
7.5.3. 调试ApplixWare ODBC 连接 83
7.5.4. 运行ApplixWare演示 84
7.5.5. 有用的宏 85
8. JDBC 接口 86
8.1. 设置 JDBC 驱动 86
8.1.1. 获取驱动 86
8.1.2. 设置类路径 87
8.1.3. 为 JDBC 准备数据库 87
8.2. 使用驱动 87
8.2.1. 导入 JDBC 87
8.2.2. 载入驱动 88
8.2.3. 连接到数据库 88
8.2.4. 关闭连接 89
8.3. 发出查询并处理结果 89
8.3.1. 使用 `Statement` 或 `PreparedStatement` 接口 89
8.3.2. 使用 `ResultSet` 接口 90
8.4. 执行更新 90
8.5. 创建和修改数据库对象 90
8.6. 存储二进制数据 91
8.7. PostgreSQL 对 JDBC API 的扩展 93
8.7.1. 访问扩展 94
8.7.2. 几何数据类型 98
8.7.3. 大对象 111
8.8. 在多线程（multi-threaded）或 servlet 环境中使用驱动 114
8.9. 延伸阅读 114
9. PyGreSQL - Python 接口 115
9.1. `pg` 模块 115

9.1.1. 常量 115
9.2. pg 模块函数 116
9.3. 连接对象：pgobject 128
9.4. 数据库包装类：DB 141
9.5. 查询结果对象：pgqueryobject 151
9.6. 大对象：pglarge 157
9.7. DB-API 接口 167

第 1 章 libpq - C 库

1.1. 简介

libpq 是 PostgreSQL 的 C 应用程序员接口。libpq 是一组库例程，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。

本节末尾包含三个简短程序，展示如何编写使用 libpq 的程序。下列目录中还有几个完整的 libpq 应用程序示例：

```
src/test/examples
src/bin/psql
```

使用 libpq 的前端程序必须包含头文件 libpq-fe.h，并且必须与 libpq 库链接。

1.2. 数据库连接函数

下面的例程用于建立与 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 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)
  ```

  此例程使用从字符串 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，该例程的参数集可以在不改变函数签名的情况下扩展，因此应用程序编程首选使用此例程或其非阻塞版本 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 \' 或 \\。）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 IP 地址。其格式应为 BSD 函数 inet_aton 等使用的标准点分数字形式。如果指定了非空长度的字符串，则使用 TCP/IP 通信。

  使用 hostaddr 代替主机名可以让应用避免主机名查找，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。如果指定了主机名而没有指定 hostaddr，则强制进行主机名查找。如果指定了 hostaddr 而没有指定主机名，

hostaddr 的值给出远程地址；若使用了 Kerberos，这会引起反向名称查询。如果主机名和 hostaddr 都被指定，hostaddr 的值给出远程地址；主机名的值被忽略，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。

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。

user

要以哪个用户名连接。

password

如果服务器要求密码认证则使用的密码。

options

要发送给服务器的跟踪/调试选项。

tty

用于后端可选调试输出的文件或 tty。

requiressl

设为 1 时要求与后端的 SSL 连接。如果服务器不支持 SSL，Libpq 将拒绝连接。设为 0（默认值）时与服务器协商。

如果任何参数未指定，则检查相应的环境变量（见第 1.10 节）。如果环境变量也未设置，则使用硬编码的默认值。返回值是指向一个抽象 struct 的指针，该结构表示到后端的连接。

- 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，参数个数固定但功能相同。

- 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。它主要是为了与旧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)
  ```

  这两个例程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。

  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，自行确保套接字处于适当的状态，如下所述。

  首先，调用 conn=PQconnectStart("*connection_info_string*")。如果 conn 为 NULL，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。循环如下：默认将连接视为“非活跃”。如果 PQconnectPoll 上次返回了 PGRES_POLLING_ACTIVE，则改将其视为“活跃”。如果 PQconnectPoll(conn) 上次返回 PGRES_POLLING_READING，则在 PQsocket(conn) 上为读取执行 select()。如果上次返回 PGRES_POLLING_WRITING，则在 PQsocket(conn) 上为写入执行 select()。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。如果 select() 表明套接字已就绪，将其视为“活跃”。一旦确定此连接处于“活跃”状态，就再次调用 PQconnectPoll(conn)。如果这次调用返回 PGRES_POLLING_FAILED，连接过程已失败。如果返回 PGRES_POLLING_OK，连接已成功建立。

  注意，用 select() 确保套接字就绪只是一个（常见的）例子；如果还有其他可用手段，例如 poll() 调用，当然可以改用它。

  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。如上所述，这两种状态都应能从 PQconnectPoll 的返回值中同样检测到。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。这些状态可能包括：

  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；等待连接启动继续进行。

CONNECTION_SETENV

正在协商环境（连接启动的一部分）。

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。应用程序可以这样处理：

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Connecting...";
        break;

    case CONNECTION_MADE:
        feedback = "Connected to server...";
        break;
.
.
.
    default:
        feedback = "Connecting...";
}
```

注意，如果 PQconnectStart 返回非 NULL 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 PQconnectStart 或 PQconnectPoll 的调用失败，也必须这样做。

当前，如果 libpq 编译时定义了 USE_SSL，PQconnectPoll 会阻塞。此限制将来可能被移除。

这些函数会把套接字置于非阻塞状态，就像调用过 PQsetnonblocking 一样。

- PQconndefaults 返回默认的连接选项。

```
PQconninfoOption *PQconndefaults(void)

struct PQconninfoOption
{
    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 */
}
```

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 struct 数组，该数组以一个 keyword 指针为 NULL 的条目结尾。注意，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。

在 PostgreSQL 7.0 之前的版本中，PQconndefaults 返回指向静态数组的指针，而不是动态分配的数组。那样不是线程安全的，因此行为已被改变。

- PQfinish 关闭与后端的连接，同时释放 PGconn 对象使用的内存。

  ```
  void PQfinish(PGconn *conn)
  ```

  注意，即使与后端的连接尝试失败（由 PQstatus 指示），应用也应调用 PQfinish 释放 PGconn 对象使用的内存。调用过 PQfinish 之后，不应再使用该 PGconn 指针。

- PQreset 重置与后端的通信端口。

  ```
  void PQreset(PGconn *conn)
  ```

  此函数将关闭与后端的连接，
  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。

- PQresetStartPQresetPoll 以非阻塞方式重置与后端的通信端口。

  ```
  int PQresetStart(PGconn *conn);

  PostgresPollingStatusType PQresetPoll(PGconn *conn);
  ```

  这些函数将关闭与后端的连接，
  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。

  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。

libpq 应用程序员应注意维护 PGconn 的抽象。请使用下面的访问函数获取 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 开始，libpq-fe.h 中甚至不再提供 struct PGconn 的定义。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。）

- PQdb 返回连接的数据库名。

  ```
  char *PQdb(const PGconn *conn)
  ```

  PQdb 及其后面几个函数返回在建立连接时确定的值。这些值在 PGconn 对象的整个生命周期内保持不变。

- 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。

  ```
  char *PQtty(const PGconn *conn)
  ```

- PQoptions 返回连接中使用的后端选项。

  ```
  char *PQoptions(const PGconn *conn)
  ```

- 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 或 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。

- 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 函数在失败时都会设置 PQerrorMessage。注意，按照 libpq 的惯例，非空的 PQerrorMessage 会包含一个末尾换行符。

- PQbackendPID 返回处理此连接的后端服务器的进程 ID。

  ```
  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 NULL。

  ```
  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 SSL 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。

1.3. 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。

1.3.1. 主要例程

- PQexec 向服务器提交一个命令并等待结果。

  ```
  PGresult *PQexec(PGconn *conn,
                   const char *query);
  ```

  返回一个 PGresult 指针，也可能返回 NULL 指针。除内存耗尽或无法将命令发送到后端等严重错误外，一般会返回非 NULL 指针。如果返回了 NULL，应将其视同 PGRES_FATAL_ERROR 结果。使用 PQerrorMessage 获取该错误的更多信息。

PGresult 结构封装了后端返回的结果。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。（从 PostgreSQL 6.4 开始，libpq-fe.h 中甚至不再提供 struct PGresult 的定义。如果你有直接访问 PGresult 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。）

- 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 —— 发送给后端的字符串为空。
  - PGRES_COMMAND_OK —— 不返回数据的命令成功完成
  - PGRES_TUPLES_OK —— 查询成功执行
  - 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
  - PGRES_FATAL_ERROR

  如果结果状态为 PGRES_TUPLES_OK，就可以使用下面描述的例程检索查询返回的行。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 响应往往暴露客户端软件中的 bug。

- 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 返回与查询相关联的错误消息；如果没有错误则返回空字符串。

```
char *PQresultErrorMessage(const PGresult *res);
```

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。

- PQclear 释放与 PGresult 关联的存储。每个查询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；发出新查询时它不会消失，即使关闭连接也不会。要清除它，必须调用 PQclear。不这样做将导致前端应用内存泄漏。

- 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 对象的内部例程。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 NULL 且 status 指示一个错误，连接的当前 errorMessage 会被复制到 PGresult. 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。

1.3.2. 用于在 SQL 查询中嵌入字符串的转义

PQescapeString 对字符串进行转义，使其可用于 SQL 查询。

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

如果想把从不可信来源（例如随机用户输入）收到的字符串包含进来，
出于安全原因你不能把它们直接放进 SQL 查询。相反，你必须对那些会被 SQL 解析器解释的特殊字符加引号处理。

PQescapeString 执行这种操作。*from* 指向待转义字符串的首字符，*length* 参数给出该字符串的字符数（既不需要也不计入末尾零字节）。*to* 应指向一个缓冲区，其容量至少为 *length* 值的两倍加一个字符，否则行为未定义。调用 PQescapeString 会把 *from* 字符串的转义版本写入 *to* 缓冲区，替换特殊字符使它们不会造成任何危害，并添加一个末尾零字节。包围 PostgreSQL 字符串字面量所必需的单引号不是结果字符串的一部分。

PQescapeString 返回写到 *to* 的字符数，不包括末尾零字节。当 *to* 与 *from* 字符串重叠时行为未定义。

1.3.3. 用于在 SQL 查询中嵌入二进制串的转义

PQescapeBytea 对二进制串（bytea 类型）进行转义，使其可用于 SQL 查询。

```
unsigned char *PQescapeBytea(unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

在 SQL 语句中作为 bytea 字符串字面量的一部分使用时，某些 ASCII 字符必须转义（但所有字符都可以转义）。一般而言，转义一个字符时，把它转换为等于其十进制 ASCII 值的八进制三位数，并在前面加两个反斜杠。单引号（'）和反斜杠（\）字符有特殊的替代转义序列。更多信息见用户指南。PQescapeBytea 执行此操作，只对最少必需的字符进行转义。

from 参数指向待转义字符串的首字符，*from_length* 参数反映该二进制串的字符数（既不需要也不计入末尾零字节）。*to_length* 参数应指向一个适合存放转义后字符串长度的缓冲区。结果字符串长度不含结果的末尾零字节。

PQescapeBytea 把 *from* 参数二进制字符串的转义版本返回到调用者提供的缓冲区。返回字符串中的所有特殊字符都已替换，使 PostgreSQL 字符串字面量解析器和 bytea 输入函数能正确处理它们。还会添加一个末尾零字节。包围 PostgreSQL 字符串字面量所必需的单引号不是结果字符串的一部分。

1.3.4. 检索 SELECT 结果信息

- PQntuples 返回查询结果中的元组（行）数。

  ```
  int PQntuples(const PGresult *res);
  ```

- PQnfields 返回查询结果中每一行的字段（列）数。

  ```
  int PQnfields(const PGresult *res);
  ```

- PQfname 返回与给定字段下标相关联的字段（列）名。字段下标从 0 开始。

  ```
  char *PQfname(const PGresult *res,
                int field_index);
  ```

- PQfnumber 返回与给定字段名相关联的字段（列）下标。

  ```
  int PQfnumber(const PGresult *res,
                const char *field_name);
  ```

  如果给定的名称不匹配任何字段，返回 -1。

- PQftype 返回与给定字段下标相关联的字段类型。返回的整数是该类型的内部编码。字段下标从 0 开始。

  ```
  Oid PQftype(const PGresult *res,
              int field_index);
  ```

  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 返回与给定字段下标相关联的字段的类型专属修饰数据。字段下标从 0 开始。

  ```
  int PQfmod(const PGresult *res,
             int field_index);
  ```

- PQfsize 返回与给定字段下标相关联的字段的大小，以字节计。字段下标从 0 开始。

  ```
  int PQfsize(const PGresult *res,
              int field_index);
  ```

PQfsize 返回数据库元组中为该字段分配的空间，换句话说，即服务器对该数据类型的二进制表示的大小。如果字段是变长的则返回 -1。

- PQbinaryTuples 如果 PGresult 包含二进制元组数据则返回 1，包含 ASCII 数据则返回 0。

```
int PQbinaryTuples(const PGresult *res);
```

目前，只有从二进制游标提取数据的查询才能返回二进制元组数据。

1.3.5. 检索 SELECT 结果值

- PQgetvalue 返回 PGresult 中某个元组（行）的单个字段（列）值。元组和字段下标都从 0 开始。

```
char* PQgetvalue(const PGresult *res,
                 int tup_num,
                 int field_num);
```

对大多数查询而言，PQgetvalue 返回的是属性值的以空字符结尾的字符串表示。但如果 PQbinaryTuples() 为 1，PQgetvalue 返回的就是该类型以后端服务器内部格式表示的二进制表示（若字段为变长则不含大小字）。此时由程序员负责把数据转换成正确的 C 类型。PQgetvalue 返回的指针指向属于 PGresult 结构的存储空间。不应修改它；如果需要在 PGresult 结构的生命周期结束后继续使用该值，就必须显式地将它复制到其他存储空间。

- PQgetisnull 测试一个字段是否为 NULL 条目。元组和字段下标从 0 开始。

```
int PQgetisnull(const PGresult *res,
                int tup_num,
                int field_num);
```

如果字段包含 NULL，此函数返回 1；包含非 NULL 值则返回 0。（注意，对于 NULL 字段，PQgetvalue 返回的是空字符串而不是空指针。）

- PQgetlength 返回字段（属性）值的长度，以字节计。元组和字段下标从 0 开始。

```
int PQgetlength(const PGresult *res,
                int tup_num,
                int field_num);
```

这是该特定数据值的实际数据长度，即 PQgetvalue 所指对象的大小。注意，对于以字符表示的值，此大小与 PQfsize 报告的二进制大小几乎没有关系。

- PQprint 打印出所有的元组以及（可选的）属性名到指定的输出流。

```
void PQprint(FILE* fout,      /* output stream */
             const PGresult *res,
             const PQprintOpt *po);

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;   /* insert to HTML table ... */
    char    *caption;    /* HTML caption */
    char    **fieldName; /* null terminated array of replacement
 field names */
} PQprintOpt;
```

此函数以前被 psql 用于打印查询结果，但现在已不再如此，此函数也不再被积极维护。

1.3.6. 检索非 SELECT 结果信息

- PQcmdStatus 返回生成该 PGresult 的 SQL 命令的命令状态字符串。

  ```
  char * PQcmdStatus(const PGresult *res);
  ```

- PQcmdTuples 返回受 SQL 命令影响的行数。

  ```
  char * PQcmdTuples(const PGresult *res);
  ```

  如果生成该 PGresult 的 SQL 命令是 INSERT、UPDATE 或 DELETE，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，返回所插入行的对象 ID。否则返回 InvalidOid。

  ```
  Oid PQoidValue(const PGresult *res);
  ```

  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。

- PQoidStatus 如果 SQL 命令是一个 INSERT，返回一个含有所插入行对象 ID 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，由 PQoidValue 取代，且不是线程安全的。

1.4. 异步查询处理

PQexec 函数足以在简单的同步应用中提交命令。但它有几个主要缺陷：

- 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。
- 由于控制权埋在 PQexec 内部，前端很难在需要时决定尝试取消正在进行的命令。（这可以在信号处理器中完成，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。

使用此功能的旧程序以及 PQputline 和 PQputnbytes 都可能在等待向后端发送数据时阻塞。为了解决这一问题，加入了 PQsetnonblocking 函数。

旧应用可以不使用 PQsetnonblocking 而保持旧的、可能阻塞的行为。新程序则可以使用 PQsetnonblocking 实现到后端的完全非阻塞连接。

- PQsetnonblocking 设置连接的非阻塞状态。

  ```
  int PQsetnonblocking(PGconn *conn, int arg)
  ```

  arg 为 1 时把连接置为非阻塞状态，*arg* 为 0 时置为阻塞状态。成功返回 0，出错返回 -1。

  在非阻塞状态下，对 PQputline、PQputnbytes、PQsendQuery 和 PQendcopy 的调用不会阻塞；如果需要再次调用，它们会返回错误。

  当数据库连接被设置为非阻塞模式后又调用了 PQexec 时，它会临时把连接置为阻塞状态，直到该 PQexec 完成。

  预计不久的将来，libpq 的更多部分将变得对 PQsetnonblocking 功能安全。

- PQisnonblocking 返回数据库连接的阻塞状态。

  ```
  int PQisnonblocking(const PGconn *conn)
  ```

  连接设置为非阻塞模式时返回 1，阻塞时返回 0。

- 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 *query);
  ```

  成功调用 PQsendQuery 后，调用一次或多次 PQgetResult 获取结果。在 PQgetResult 返回 NULL（表示命令已完成）之前，不得（在同一连接上）再次调用 PQsendQuery。

- PQgetResult 等待先前一次 PQsendQuery 的下一个结果并将其返回。查询完成且不再有结果时返回 NULL。

  ```
  PGresult *PQgetResult(PGconn *conn);
  ```

  必须反复调用 PQgetResult 直到它返回 NULL，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 NULL。）对 PQgetResult 返回的每个非 NULL 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，只有当查询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条 SQL 命令，这些命令的结果可以分别获取。（顺便说一句，这也支持一种简单的重叠处理：前端可以处理某条查询的结果，同时后端继续处理同一命令字符串中后面的查询。）但调用 PQgetResult 仍会使前端阻塞，直到后端完成下一条 SQL 命令。恰当使用另外三个函数可以避免这一点：

- PQconsumeInput 如果后端有可用的输入，消费它。

  ```
  int PQconsumeInput(PGconn *conn);
  ```

  PQconsumeInput 正常返回 1 表示“没有错误”；发生问题时返回 0（此时 PQerrorMessage 会被设置）。注意，返回结果并不说明是否实际读取了输入数据。调用 PQconsumeInput 后，应用可以检查 PQisBusy 和/或 PQnotifies，看它们的状态是否发生了变化。

即使应用尚未准备好处理结果或通知，也可以调用 PQconsumeInput。该例程会读取可用数据并保存到缓冲区中，从而使 select() 的可读就绪指示消失。因此应用可以用 PQconsumeInput 立即清除 select() 的就绪条件，随后再从容地检查结果。

- PQisBusy 如果查询仍在忙碌则返回 1，也就是说 PQgetResult 会阻塞等待输入。返回 0 表示可以放心调用 PQgetResult 而不会阻塞。

  ```
  int PQisBusy(PGconn *conn);
  ```

  PQisBusy 本身不会尝试从后端读取数据；因此必须先调用 PQconsumeInput，否则忙碌状态永远不会结束。

- PQflush 尝试刷新排队发往后端的任何数据，成功（或发送队列为空）返回 0，因某种原因失败返回 EOF。

  ```
  int PQflush(PGconn *conn);
  ```

  在非阻塞连接上，调用 select() 判断响应是否到达之前需要先调用 PQflush。返回 0 可确保没有已排队但尚未真正发送到后端的数据。只有使用了 PQsetnonblocking 的应用才需要这样做。

- PQsocket 获取后端连接套接字的文件描述符号。有效的描述符将 >= 0；结果为 -1 表示当前没有打开的后端连接。

  ```
  int PQsocket(const PGconn *conn);
  ```

  应当用 PQsocket 获取后端套接字描述符，为执行 select() 做准备。这使使用阻塞连接的应用可以同时等待后端响应或其他条件。如果 select() 的结果表明可以从后端套接字读取数据，就应调用 PQconsumeInput 读取数据；随后即可用 PQisBusy、PQgetResult 和/或 PQnotifies 处理响应。

  非阻塞连接（使用了 PQsetnonblocking 的连接）在 PQflush 返回 0、表明没有缓冲数据等待发往后端之前，不应使用 select()。

使用这些函数的典型前端会有一个主循环，用 select 等待它必须响应的所有条件。其中一个条件是后端有可读的输入；用 select 的话说，就是由 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。然后可以调用 PQisBusy，若 PQisBusy 返回假（0），接着调用 PQgetResult。还可以调用 PQnotifies 检测 NOTIFY 消息（见第 1.6 节）。

使用 PQsendQuery/PQgetResult 的前端还可以尝试取消一个仍随后端处理中的命令。

- PQrequestCancel 请求 PostgreSQL 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（例如因为后端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 发出的查询。注意，如果连接当前未打开或后端当前没有在处理命令，PQrequestCancel 将不起作用。

1.5. 快速路径接口

PostgreSQL 提供一个快速路径接口来向后端发送函数调用。这是通向系统内部的一个天窗，可能成为潜在的安全漏洞。大多数用户不需要此特性。

- PQfn 通过快速路径接口请求执行一个后端函数。

```
PGresult* PQfn(PGconn* conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);
```

fnid 参数是要执行的函数的对象标识符。*result_buf* 是存放返回值的缓冲区。调用者必须已分配足够的空间来存储返回值（这里不做检查！）。实际结果长度将返回到 *result_len* 所指向的整数中。如果预期结果是 4 字节整数，将 *result_is_int* 设为 1；否则设为 0。（将 *result_is_int* 设为 1 会告诉 libpq 在必要时做字节序交换，使值以适合客户端机器的 int 值送达。当 *result_is_int* 为 0 时，后端发送的字节串原样返回。）*args* 和 *nargs* 指定要传给函数的参数。

```
typedef struct {
    int len;
    int isint;
    union {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

PQfn 总是返回有效的 PGresult*。使用结果前应检查其状态。不再需要时，调用者负责用 PQclear 释放 PGresult。

1.6. 异步通知

PostgreSQL 通过 LISTEN 和 NOTIFY 命令支持异步通知。后端用 LISTEN 命令注册它对某个特定通知条件的兴趣（并可以用 UNLISTEN 命令停止监听）。当任何后端执行带该条件名的 NOTIFY 命令时，所有监听该条件的后端都会被异步通知。通知者不会向监听者传递任何附加信息。因此，通常需要通信的任何实际数据都通过一个数据库关系传递。
条件名通常与相关的关系同名，但并不要求必须有相关的关系。

libpq 应用把 LISTEN 和 UNLISTEN 命令作为普通 SQL 命令提交。随后，可以通过调用 PQnotifies 检测 NOTIFY 消息的到达。

- PQnotifies 从后端发来的尚未处理的通知消息列表中返回下一条通知。没有待处理的通知时返回 NULL。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[NAMEDATALEN];        /* name of relation
                                       * containing data */
    int  be_pid;                      /* process id of backend */
} PGnotify;
```

处理完 PQnotifies 返回的 PGnotify 对象后，务必用 free() 释放它，以避免内存泄漏。

> **注意**
>
> 在 PostgreSQL 6.4 及之后的版本中，be_pid 是发出通知的后端的 PID，而在较早的版本中它总是你自己的后端的 PID。

第二个示例程序演示了异步通知的使用。

PQnotifies() 并不真正读取后端数据；它只是返回先前被其他 libpq 函数吸收的消息。在 libpq 的早期版本中，确保及时收到 NOTIFY 消息的唯一方法是不断地提交查询（哪怕是空查询），然后在每次 PQexec() 之后检查 PQnotifies()。这种方法虽然仍然有效，但由于浪费处理能力已被弃用。

在没有有用查询可提交时检查 NOTIFY 消息的更好办法是调用 PQconsumeInput()，然后检查 PQnotifies()。可以用 select() 等待后端数据到达，这样除非有事可做就不会消耗 CPU。（获取用于 select() 的文件描述符号的方法见 PQsocket()。）注意，无论你用 PQsendQuery/PQgetResult 提交查询还是直接用 PQexec，这样做都没有问题。但应记住在每次 PQgetResult 或 PQexec 之后检查 PQnotifies()，看查询处理期间是否有通知到来。

1.7. 与 COPY 命令相关的函数

PostgreSQL 中的 COPY 命令可以选择从 libpq 所用的网络连接读取或向其写入。因此，需要能直接访问此网络连接的函数，应用才能利用这一能力。

只有在从 PQexec 或 PQgetResult 得到 PGRES_COPY_OUT 或 PGRES_COPY_IN 结果对象之后，才应执行这些函数。

- PQgetline 把一行由后端服务器传来的以换行符结尾的字符读入大小为 length 的缓冲区字符串。

  ```
  int PQgetline(PGconn *conn,
                char *string,
                int length)
  ```

  与 fgets 一样，此例程最多把 length-1 个字符复制到 string 中；但与 gets 相似，它会把结尾的换行符转换为零字节。PQgetline 在输入结束时返回 EOF，整行已读完返回 0，缓冲区已满但结尾换行符尚未读到返回 1。

  注意，应用必须检查新的一行是否由两个字符 \. 组成，这表示后端服务器已发送完 copy 命令的结果。如果应用可能收到超过 length-1 个字符的行，就需要小心确保正确识别 \. 行（例如，不要把长数据行的结尾误当作终结行）。src/bin/psql/copy.c 中的代码包含正确处理 copy 协议的例程示例。

- PQgetlineAsync 把一行由后端服务器传来的以换行符结尾的字符读入缓冲区而不阻塞。

  ```
  int PQgetlineAsync(PGconn *conn,
  ```

```
                      char *buffer,
                      int bufsize)
```

此例程类似于 PQgetline，但可供必须异步（即不阻塞地）读取 COPY 数据的应用使用。发出 COPY 命令并得到 PGRES_COPY_OUT 响应后，应用应交替调用 PQconsumeInput 和 PQgetlineAsync，直到检测到数据结束信号。与 PQgetline 不同，此例程自己负责检测数据结束。每次调用时，如果 libpq 的输入缓冲区中已有完整的一行以换行符结尾的数据行，或者到来的数据行太长放不进调用者提供的缓冲区，PQgetlineAsync 就会返回数据。否则，在该行剩余部分到达之前不返回数据。

如果已识别到 copy 数据结束标记，此例程返回 -1；没有可用数据返回 0；返回正数时该数给出所返回数据的字节数。返回 -1 时，调用者接下来必须调用 PQendcopy，然后回到正常处理。返回的数据不会跨过换行符。可能时一次返回整行；
但如果调用者提供的缓冲区太小装不下后端发来的一行，就会返回部分数据行。
这可以通过检查最后返回的字节是否为 \n 来检测。返回的字符串不以空字符结尾。（如果想加上结尾空字符，务必让传入的 *bufsize* 比实际可用空间小一。）

- PQputline 向后台服务器发送一个以空字符结尾的字符串。成功返回 0，无法发送字符串返回 EOF。

  ```
  int PQputline(PGconn *conn,
                const char *string);
  ```

  注意，应用必须在最后一行显式发送两个字符 \.，以告知后端它已发送完数据。

- PQputnbytes 向后台服务器发送一个不以空字符结尾的字符串。成功返回 0，无法发送字符串返回 EOF。

  ```
  int PQputnbytes(PGconn *conn,
                  const char *buffer,
                  int nbytes);
  ```

  它与 PQputline 完全一样，只是由于要发送的字节数是直接指定的，数据缓冲区不需要以空字符结尾。

- PQendcopy 与后端同步。此函数等待后端完成 copy。它应当在用 PQputline 向后端发送完最后一个字符串之后，或用 PGgetline 从后端收到最后一个字符串之后发出。必须发出此函数，否则后端可能与前端“失去同步”。从此函数返回后，后端就准备好接收下一条 SQL 命令了。成功完成时返回值为 0，否则非零。

  ```
  int PQendcopy(PGconn *conn);
  ```

  例如：

  ```
  PQexec(conn, "CREATE TABLE foo (a int4, b char(16), d double
   precision)");
  PQexec(conn, "COPY foo FROM STDIN");
  PQputline(conn, "3\thello world\t4.5\n");
  PQputline(conn,"4\tgoodbye world\t7.11\n");
  ...
  PQputline(conn,"\\.\n");
  PQendcopy(conn);
  ```

使用 PQgetResult 时，应用对 PGRES_COPY_OUT 结果的响应应是反复执行 PQgetline，看到终结行后再执行 PQendcopy。然后应回到 PQgetResult 循环，直到 PQgetResult 返回 NULL。类似地，PGRES_COPY_IN 结果由一系列 PQputline 调用加 PQendcopy 处理，然后回

到 PQgetResult 循环。这样的安排可确保嵌入在一系列 SQL 命令中的 copy in 或 copy out 命令被正确执行。

较老的应用可能通过 PQexec 提交 copy in 或 copy out，并认为 PQendcopy 之后事务就完成了。只有当 copy in/out 是命令字符串中唯一的 SQL 命令时，这样做才是正确的。

1.8. libpq 跟踪函数

- PQtrace 把前端/后端通信的跟踪打开到调试文件流。

```
void PQtrace(PGconn *conn
             FILE *debug_port)
```

- PQuntrace 关闭由 PQtrace 启动的跟踪。

```
void PQuntrace(PGconn *conn)
```

1.9. libpq 控制函数

- PQsetNoticeProcessor 控制 libpq 产生的通知和警告消息的报告方式。

```
typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

默认情况下，libpq 把来自后端的通知消息以及它自己产生的少量错误消息打印到 stderr。可以通过提供一个对这些消息另做处理的回调函数来改变这一行为。
回调函数会收到错误消息的文本（包括末尾换行符），外加一个与传给 PQsetNoticeProcessor 的指针相同的 void 指针。（需要时可以用该指针访问应用特定的状态。）默认的通知处理器只是

```
static void
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

要使用特定的通知处理器，只需在创建新的 PGconn 对象之后立即调用 PQsetNoticeProcessor。

返回值是指向前一个通知处理器的指针。如果传入 NULL 回调函数指针，则不采取任何动作，但仍返回当前指针。

设置通知处理器之后，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，PGconn 当前的通知处理器指针会被复制到 PGresult 中，供 PQgetvalue 等例程在需要时使用。

1.10. 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 或 PQsetdbLogin 将使用这些值。它们有助于避免把数据库名硬编码进简单的应用程序。

- PGHOST 设置默认的服务器名。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）。
- PGPORT 设置与 PostgreSQL 后端通信所用的默认 TCP 端口号或 Unix 域套接字文件扩展名。
- PGDATABASE 设置默认的 PostgreSQL 数据库名。
- PGUSER 设置连接数据库和认证所用的用户名。
- PGPASSWORD 设置当后端要求密码认证时使用的密码。不建议这样做，因为在某些平台上，其他人可以用带特殊选项的 ps 命令读到该密码。
- 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，PostgreSQL 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。仅当后端选择了 Kerberos 认证时才会使用此环境变量。
- PGOPTIONS 设置 PostgreSQL 后端的附加运行时选项。
- PGTTY 设置显示来自后端服务器的调试消息的文件或 tty。

下面的环境变量可用于为每个 PostgreSQL 会话指定用户级的默认行为：

- PGDATESTYLE 设置日期/时间表示的默认风格。
- PGTZ 设置默认时区。
- PGCLIENTENCODING 设置默认的客户端编码（如果配置 PostgreSQL 时选择了多字节支持）。

下面的环境变量可用于为每个 PostgreSQL 会话指定默认的内部行为：

- PGGEQO 设置遗传优化器的默认模式。

关于这些环境变量的正确取值，请参阅 SET SQL 命令。

1.11. 线程行为

从 PostgreSQL 7.0 起，libpq 是线程安全的，前提是没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查询。（如果需要运行并发查询，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 加密方法的 Libpq 客户端依赖操作系统的 crypt() 函数，而它常常不是线程安全的。最好使用 MD5 加密，它在所有平台上都是线程安全的。

1.12. 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* 选项让它指向 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` 或者没有指定正确的路径。

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。

1.13. 示例程序

例 1.1. libpq示例程序 1

```
/*
 * testlibpq.c
 *
 * Test the C version of libpq, the PostgreSQL frontend
 * library.
 */
#include <stdio.h>
#include <libpq-fe.h>

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    /* FILE *debug; */

    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */
    dbName = "template1";

    /* make a connection to the database */
```

```
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", db
Name);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* debug = fopen("/tmp/trace.out","w"); */
    /* PQtrace(conn, debug);  */

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
 avoid
     * memory leaks
     */
    PQclear(res);

    /*
     * fetch rows from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor CURSOR FOR SELECT * FROM pg_
database");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);
    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples
 properly\n");
        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 cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    /* fclose(debug); */
    return 0;

}
```

例 1.2. 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;
 *
 * Or, if you want to get fancy, try this:
 * Populate a database with the following:
 *
 *   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
 *
 *   INSERT INTO TBL1 values (10);
 *
```

```
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */
    dbName = getenv("USER");    /* change this to the name of your
 test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", db
Name);
```

```
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "LISTEN TBL2");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
 avoid
     * memory leaks
     */
    PQclear(res);

    while (1)
    {

        /*
         * wait a little bit between checks; waiting with select()
         * would be more efficient.
         */
        sleep(1);
        /* collect any asynchronous backend messages */
        PQconsumeInput(conn);
        /* check for asynchronous notify messages */
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                 "ASYNC NOTIFY of '%s' from backend pid '%d' received
\n",
                    notify->relname, notify->be_pid);
            free(notify);
        }
    }

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

例 1.3. libpq 示例程序 3

```
/*
 * testlibpq3.c Test the C version of Libpq, the PostgreSQL frontend
 * library. tests the binary cursor interface
 *
 *
 *
```

```
 * populate a database by doing the following:
 *
 * CREATE TABLE test1 (i int4, d real, p polygon);
 *
 * INSERT INTO test1 values (1, 3.567, polygon '(3.0, 4.0, 1.0, 2.0)')
;
 *
 * INSERT INTO test1 values (2, 89.05, polygon '(4.0, 3.0, 2.0, 1.0)')
;
 *
 * the expected output is:
 *
 * tuple 0: got i = (4 bytes) 1, d = (4 bytes) 3.567000, p = (4
 * bytes) 2 points   boundbox = (hi=3.000000/4.000000, lo =
 * 1.000000,2.000000) tuple 1: got i = (4 bytes) 2, d = (4 bytes)
 * 89.050003, p = (4 bytes) 2 points   boundbox =
 * (hi=4.000000/3.000000, lo = 2.000000,1.000000)
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "utils/geo_decls.h"    /* for the POLYGON type */

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;
    int         i_fnum,
                d_fnum,
                p_fnum;
    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
```

```
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */

    dbName = getenv("USER");    /* change this to the name of your
 test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", db
Name);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
 avoid
     * memory leaks
     */
    PQclear(res);

    /*
     * fetch rows from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor BINARY CURSOR FOR SELECT *
 FROM test1");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
```

```
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples
properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    i_fnum = PQfnumber(res, "i");
    d_fnum = PQfnumber(res, "d");
    p_fnum = PQfnumber(res, "p");

    for (i = 0; i < 3; i++)
    {
        printf("type[%d] = %d, size[%d] = %d\n",
               i, PQftype(res, i),
               i, PQfsize(res, i));
    }
    for (i = 0; i < PQntuples(res); i++)
    {
        int        *ival;
        float      *dval;
        int         plen;
        POLYGON    *pval;

        /* we hard-wire this to the 3 fields we know about */
        ival = (int *) PQgetvalue(res, i, i_fnum);
        dval = (float *) PQgetvalue(res, i, d_fnum);
        plen = PQgetlength(res, i, p_fnum);

        /*
         * plen doesn't include the length field so need to
         * increment by VARHDSZ
         */
        pval = (POLYGON *) malloc(plen + VARHDRSZ);
        pval->size = plen;
        memmove((char *) &pval->npts, PQgetvalue(res, i, p_fnum),
plen);
        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d,\n",
               PQgetlength(res, i, i_fnum), *ival);
        printf(" d = (%d bytes) %f,\n",
               PQgetlength(res, i, d_fnum), *dval);
        printf(" p = (%d bytes) %d points \tboundbox = (hi=%f/%f, lo =
%f,%f)\n",
               PQgetlength(res, i, d_fnum),
               pval->npts,
               pval->boundbox.xh,
               pval->boundbox.yh,
               pval->boundbox.xl,
               pval->boundbox.yl);
```

```
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

第 2 章 大对象

2.1. 简介

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已被声明为大对象的用户数据。

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对数据的随机访问，也就是读写大值中一小块一小块内容的能力。未来计划为 TOAST 也配备这种功能。

本节描述 PostgreSQL 大对象数据的实现以及编程和查询语言接口。本节的例子使用 libpq C 库，但 PostgreSQL 原生的大多数编程接口都支持等价的功能。其他接口可能在内部使用大对象接口来提供对大值的通用支持。这里不对此进行描述。

2.2. 实现特性

大对象的实现把大对象拆分成“块”并把块存储在数据库的元组中。一个 B-tree 索引保证在做随机访问读写时能快速搜索到正确的块号。

2.3. 接口

下面描述 PostgreSQL 提供的访问大对象的设施，既包括后端中作为用户自定义函数一部分的用法，也包括前端中作为使用该接口的应用一部分的用法。对于熟悉 POSTGRES 4.2 的用户来说，PostgreSQL 有一套新函数，提供了更连贯的接口。

> **注意**
>
> 所有对大对象的操作必须在一个 SQL 事务中进行。这一要求从 PostgreSQL 6.5 起被严格执行，尽管在之前的版本中它一直是隐含的要求，忽略它会导致错误行为。

PostgreSQL 的大对象接口模仿 Unix 文件系统接口设计，有与 `open(2)`、`read(2)`、`write(2)`、`lseek(2)` 等对应的函数。用户函数调用这些例程来只从大对象中检索感兴趣的数据。例如，假设存在一个存储面孔照片的名为 `mugshot` 的大对象类型，那么可以在 `mugshot` 数据上声明一个名为 `beard` 的函数。`beard` 可以查看照片的下三分之一部分，并确定那里出现的胡须颜色（如果有的话）。整个大对象的值不需要被 `beard` 函数缓冲，甚至不需要被检查。大对象可以从动态装载的 C 函数或链接了该库的数据库客户端程序访问。PostgreSQL 提供了一组支持打开、读取、写入、关闭和在大对象上定位的例程。

2.3.1. 创建一个大对象

例程

```
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。下面的命令创建一个大对象：

```
inv_oid = lo_creat(INV_READ|INV_WRITE);
```

2.3.2. 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。

2.3.3. 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。

2.3.4. 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。

2.3.5. 向大对象写入数据

例程

```
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。

2.3.6. 从大对象读取数据

例程

```
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。

2.3.7. 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)
```

此例程把 *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、SEEK_CUR 和 SEEK_END。

2.3.8. 关闭一个大对象描述符

可以通过调用下面的函数来关闭一个大对象：

```
int lo_close(PGconn *conn, int fd)
```

其中 *fd* 是由 lo_open 返回的大对象描述符。成功时 lo_close 返回零。发生错误时，返回值为负。

2.3.9. 移除一个大对象

要从数据库中移除一个大对象，调用

```
Oid lo_unlink(PGconn *conn, Oid lobjId)
```

lobjId 参数指定要删除的大对象的 OID。

2.4. 服务器端内建函数

有两个内置的已注册函数 lo_import 和 lo_export，它们便于在 SQL 查询中使用。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';
```

2.5. 从 Libpq 访问大对象

例 2.1是一个示例程序，展示了 libpq 中大对象接口的用法。程序的一部分被注释掉了，但为了读者受益仍保留在源码中。该程序可以在源码发行包的 src/test/examples/testlo.c 中找到。使用 libpq 中大对象接口的前端应用应当包含头文件 libpq/libpq-fs.h 并链接 libpq 库。

例 2.1. 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 3 章 libpq++ - C++ 绑定库

3.1. 简介

libpq++ 是 PostgreSQL 的 C++ API。libpq++ 是一组允许客户端程序连接到 PostgreSQL 后端服务器的类。这些连接有两种形式：数据库类和大对象类。

数据库类用于操作数据库。你可以向 PostgreSQL 后端服务器发送各种 SQL 查询和命令，并获取服务器的响应。

大对象类用于操作数据库中的大对象。虽然大对象实例可以向 PostgreSQL 后端服务器发送普通查询，但它只适用于不返回任何数据的简单查询。应当把大对象看作一个文件流。
将来它应当表现得非常像 C++ 文件流 `cin`、`cout` 和 `cerr`。

本章基于 libpq C 库的文档（见第 1 章）。源代码发行版的 `src/interfaces/libpq++/examples` 中有若干 libpq++ 应用的示例。

3.2. 控制与初始化

3.2.1. 环境变量

下列环境变量可用于为环境设置默认值，避免把数据库名硬编码到应用程序中：

> **注意**
>
> 完整的可用连接选项列表见第 1.10 节。

下列环境变量可用于选择默认的连接参数值，如果调用代码没有直接指定值，`PQconnectdb` 或 `PQsetdbLogin` 将使用它们。这些变量有助于避免在简单的应用程序中硬编码数据库名。

> **注意**
>
> `libpq++` 只使用环境变量或 `libpq` 的 `PQconnectdb` *`conninfo`* 风格字符串。

- `PGHOST` 设置默认的服务器名。如果它以斜杠开头，则指定 Unix 域通信而不是 TCP/IP 通信；该值是套接字文件所在目录的名称（默认为 `/tmp`）。
- `PGPORT` 设置与 PostgreSQL 后端通信的默认 TCP 端口号或 Unix 域套接字文件扩展名。
- `PGDATABASE` 设置默认的 PostgreSQL 数据库名。
- `PGUSER` 设置用于连接数据库和进行认证的用户名。
- `PGPASSWORD` 设置在后端要求密码认证时使用的密码。不推荐这种方式，因为在某些平台上，其他人可以用带特殊选项的 `ps` 命令读取该密码。
- `PGREALM` 设置与 PostgreSQL 一起使用的 Kerberos 域（如果它与本地域不同）。如果设置了 `PGREALM`，PostgreSQL 应用将尝试向该域的服务器认证，并使用单独的票据文件以避免与本地票据文件冲突。只有在后端选择了 Kerberos 认证时才会使用此环境变量。
- `PGOPTIONS` 为 PostgreSQL 后端设置额外的运行时选项。

- PGTTY 设置显示来自后端服务器的调试消息的文件或 tty。

下列环境变量可用于为每个 PostgreSQL 会话指定用户级的默认行为：

- PGDATESTYLE 设置日期/时间表示的默认风格。
- PGTZ 设置默认时区。

下列环境变量可用于为每个 PostgreSQL 会话指定默认的内部行为：

- PGGEQO 设置遗传优化器的默认模式。

关于这些环境变量的正确取值，请参阅 SQL 命令 SET。

3.3. libpq++ 类

3.3.1. 连接类：PgConnection

连接类实际建立与数据库的连接，并被所有访问类继承。

3.3.2. 数据库类：PgDatabase

数据库类提供了带有到后端服务器连接的 C++ 对象。要创建这样的对象，首先需要可供后端访问的合适环境。下列构造函数负责从 C++ 程序建立到后端服务器的连接。

3.4. 数据库连接函数

- PgConnection 建立到后端数据库服务器的新连接。

  ```
  PgConnection::PgConnection(const char *conninfo)
  ```

  conninfo 字符串与底层 libpq 的 PQconnectdb 函数所用的相同。

  虽然通常是从某个访问类中调用，但也可以通过创建 PgConnection 对象来建立到后端服务器的连接。

- ConnectionBad 返回到后端服务器的连接是成功还是失败。

  ```
  bool PgConnection::ConnectionBad() const
  ```

  如果连接失败则返回真。

- Status 返回到后端服务器的连接的状态。

  ```
  ConnStatusType PgConnection::Status()
  ```

  根据连接的状态返回 CONNECTION_OK 或 CONNECTION_BAD。

- PgDatabase 建立到后端数据库服务器的新连接。

  ```
  PgDatabase(const char *conninfo)
  ```

  在创建了 PgDatabase 之后，应当先确认到数据库的连接已经成功，然后再向该对象发送查询。这很容易做到：用 Status 或 ConnectionBad 方法获取 PgDatabase 对象的当前状态。

- DBName 返回当前数据库的名字。

  ```
  const char *PgConnection::DBName()
  ```

- Notifies 从后端收到的未处理通知消息列表中返回下一条通知。

  ```
  PGnotify* PgConnection::Notifies()
  ```

  细节见 libpq 中的 PQnotifies。

3.5. 查询执行函数

3.5.1. 主例程

- Exec 向后端服务器发送一条命令。通常更可取的是使用下面两个函数之一。

  ```
  ExecStatusType PgConnection::Exec(const char* query)
  ```

  返回命令的结果状态。可能得到下列状态值：

  ```
  PGRES_EMPTY_QUERY
  PGRES_COMMAND_OK（命令不是查询时）
  PGRES_TUPLES_OK（查询成功返回元组时）
  PGRES_COPY_OUT
  PGRES_COPY_IN
  PGRES_BAD_RESPONSE（收到意外响应时）
  PGRES_NONFATAL_ERROR
  PGRES_FATAL_ERROR
  ```

- ExecCommandOk 向后端服务器发送一条非查询命令（不返回行的命令）。

  ```
  int PgConnection::ExecCommandOk(const char *query)
  ```

  命令成功时返回真（1）。

- ExecTuplesOk 向后端服务器发送一条查询命令（返回行的命令）。

  ```
  int PgConnection::ExecTuplesOk(const char *query)
  ```

  查询成功时返回真（1）。

- ErrorMessage 返回最后一条错误消息的文本。

  ```
  const char *PgConnection::ErrorMessage()
  ```

3.5.2. 获取 SELECT 结果信息

- Tuples 返回查询结果中元组（行）的数目。

  ```
  int PgDatabase::Tuples() const
  ```

- Fields 返回查询结果中每个元组的字段（列）数目。

  ```
  int PgDatabase::Fields()
  ```

- FieldName 返回与给定字段索引相关联的字段（列）名。字段索引从 0 开始。

  ```
  const char *PgDatabase::FieldName(int field_num) const
  ```

- FieldNum 返回与给定字段名相关联的字段（列）索引。

  ```
  int PgDatabase::FieldNum(const char* field_name) const
  ```

  如果给定的名称不匹配任何字段，则返回 -1。

- FieldType 返回与给定字段索引相关联的字段类型。返回的整数是该类型的内部编码。字段索引从 0 开始。

  ```
  Oid PgDatabase::FieldType(int field_num) const
  ```

- FieldType 返回与给定字段名相关联的字段类型。返回的整数是该类型的内部编码。字段索引从 0 开始。

  ```
  Oid PgDatabase::FieldType(const char* field_name) const
  ```

- FieldSize 返回与给定字段索引相关联的字段的字节大小。字段索引从 0 开始。

  ```
  int PgDatabase::FieldSize(int field_num) const
  ```

  返回数据库元组中为该字段分配的空间（按字段编号给出）。换句话说，
  就是服务器上该数据类型二进制表示的大小。如果字段是变长的，则返回 -1。

- FieldSize 返回与给定字段索引相关联的字段的字节大小。字段索引从 0 开始。

  ```
  int PgDatabase::FieldSize(const char *field_name) const
  ```

  返回数据库元组中为该字段分配的空间（按字段名给出）。换句话说，
  就是服务器上该数据类型二进制表示的大小。如果字段是变长的，则返回 -1。

3.5.3. 获取 SELECT 结果值

- GetValue 返回 PGresult 中某个元组的单个字段（列）值。元组和字段索引从 0 开始。

  ```
  const char *PgDatabase::GetValue(int tup_num, int field_num) const
  ```

  对大多数查询来说，GetValue 返回的是属性值的以空字符结尾的字符串表示。但如果 BinaryTuples 为真，GetValue 返回的是该类型以后端服务器内部格式表示的二进制表示（如果字段是变长的，则不包括长度字）。这时由程序员负责把数据造型和转换为正确的 C 类型。GetValue 返回的指针指向 PGresult 结构一部分的存储。不应修改它，如果要在 PGresult 结构自身的生存期之后使用该值，必须把它显式复制到其他存储中。Binary Tuples 尚未实现。

- GetValue 返回 PGresult 中某个元组的单个字段（列）值。元组和字段索引从 0 开始。

  ```
  const char *PgDatabase::GetValue(int tup_num, const char *field_
  name) const
  ```

  对大多数查询来说，GetValue 返回的是属性值的以空字符结尾的字符串表示。但如果 BinaryTuples 为真，GetValue 返回的是该类型以后端服务器内部格式表示的二进制表示（如果字段是变长的，则不包括长度字）。这时由程序员负责把数据造型和转换为正确的 C 类型。GetValue 返回的指针指向 PGresult 结构一部分的存储。不应修改它，如果要

在 PGresult 结构自身的生存期之后使用该值，必须把它显式复制到其他存储中。Binary Tuples 尚未实现。

- GetLength 返回字段（列）的字节长度。元组和字段索引从 0 开始。

  ```
  int PgDatabase::GetLength(int tup_num, int field_num) const
  ```

  这是该数据值的实际数据长度，即 GetValue 所指向对象的大小。注意，对于以字符表示的值，这个大小与 PQfsize 报告的二进制大小关系不大。

- GetLength 返回字段（列）的字节长度。元组和字段索引从 0 开始。

  ```
  int PgDatabase::GetLength(int tup_num, const char* field_name) const
  ```

  这是该数据值的实际数据长度，即 GetValue 所指向对象的大小。注意，对于以字符表示的值，这个大小与 PQfsize 报告的二进制大小关系不大。

- GetIsNull 返回某个字段是否为空值。

  ```
  bool GetIsNull(int tup_num, int field_num) const
  ```

  注意，对于空值字段，GetValue 返回的是空字符串而不是 NULL 指针。

- GetIsNull 返回某个字段是否为空值。

  ```
  bool GetIsNull(int tup_num, const char *field_name) const
  ```

  注意，对于空值字段，GetValue 返回的是空字符串而不是 NULL 指针。

- DisplayTuples 把所有元组以及可选的属性名打印到指定的输出流。

  ```
  void PgDatabase::DisplayTuples(FILE *out = 0, bool fillAlign = true,

  const char* fieldSep = "|", bool printHeader = true, bool quiet =
   false) const
  ```

  该函数已过时。

- PrintTuples 把所有元组以及可选的属性名打印到指定的输出流。

  ```
  void PgDatabase::PrintTuples(FILE *out = 0, bool printAttName =
   true,
  bool terseOutput = false, bool fillAlign = false) const
  ```

  该函数已过时。

3.5.4. 获取非 SELECT 结果信息

- CmdTuples 返回 INSERT、UPDATE 或 DELETE 之后受影响的行数。如果是其他命令，则返回 -1。

  ```
  int PgDatabase::CmdTuples() const
  ```

- OidStatus

  ```
  const char *PgDatabase::OidStatus() const
  ```

3.6. 异步通知

PostgreSQL 通过 LISTEN 和 NOTIFY 命令支持异步通知。后端用 LISTEN 命令注册它对某个通知条件的关注。当另一个后端执行同名的 NOTIFY 时，所有正在监听该条件的后端都会被异步通知。通知者不会向监听者传递任何额外的信息。因此，
通常任何需要通信的实际数据都是通过一个关系传递的。

每当一个已连接的后端收到异步通知时，libpq++ 应用都会得到通知。但是，从后端到前端的通信并不是异步的。libpq++ 应用必须轮询后端，查看是否有待处理的通知信息。在执行一条命令之后，前端可以调用 PgDatabase::Notifies 查看后端当前是否有可用的通知数据。PgDatabase::Notifies 从后端的未处理通知列表中返回通知。如果没有来自后端的待处理通知，该函数返回 NULL。PgDatabase::Notifies 的行为像弹出一个栈。一旦某条通知被 PgDatabase::Notifies 返回，它就被视为已处理，并将从通知列表中移除。

- PgDatabase::Notifies 从服务器取回待处理的通知。

  ```
  PGnotify* PgDatabase::Notifies()
  ```

第二个示例程序演示了异步通知的用法。

3.7. 与 COPY 命令相关的函数

PostgreSQL 中的 COPY 命令有一些选项，可以读取或写入 libpq++ 所使用的网络连接。因此，需要一些函数来直接访问这个网络连接，使应用可以充分利用这一能力。

- PgDatabase::GetLine 把后端服务器传输的一行以换行符结尾的字符读入大小为 *length* 的缓冲区 *string*。

  ```
  int PgDatabase::GetLine(char* string, int length)
  ```

  与 Unix 系统例程 fgets() 一样，此例程最多复制*length*-1 个字符到 *string*。但它又像 gets()，会把结尾的换行符转换为一个零字节。

  PgDatabase::GetLine 在文件末尾返回 EOF，整行已读取时返回 0，而缓冲区已满但尚未读到终止换行符时返回 1。

  注意，应用程序必须检查新行是否由一个反斜杠后跟一个句点（\.）组成，
  这表示后端服务器已经完成了 COPY 结果的发送。因此，如果应用预期会收到长度超过 *length*-1 个字符的行，务必非常仔细地检查 PgDatabase::GetLine 的返回值。

- PgDatabase::PutLine 向后端服务器发送一个以空字符结尾的 *string*。

  ```
  void PgDatabase::PutLine(char* string)
  ```

  应用程序必须显式发送字符 \.，向后端表明它已完成数据的发送。

- PgDatabase::EndCopy 与后端同步。

  ```
  int PgDatabase::EndCopy()
  ```

  该函数等待后端完成对 COPY 的处理。无论是用 PgDatabase::PutLine 向后端发送完最后一个字符串，还是用 PgDatabase::GetLine 从后端接收完最后一个字符串之后，都应当调用它。必须调用它，否则后端可能与前端“失去同步”。该函数返回后，
  后端就准备好接收下一条命令了。

成功完成时返回值为 0，否则为非零。

例如：

```
PgDatabase data;
data.Exec("CREATE TABLE foo (a int4, b char(16), d double precision)")
;
data.Exec("COPY foo FROM STDIN");
data.PutLine("3\tHello World\t4.5\n");
data.PutLine("4\tGoodbye World\t7.11\n");
...
data.PutLine("\\.\n");
data.EndCopy();
```

第 4 章 pgtcl - Tcl 绑定库

4.1. 简介

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。

本包最初由 Jolly Chen 编写。

表 4.1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 4.1. pgtcl 命令

命令	描述
pg_connect	打开一个到后端服务器的连接
pg_disconnect	关闭一个连接
pg_conndefaults	获取连接选项及其默认值
pg_exec	向后端发送一个查询
pg_result	操作查询的结果
pg_select	循环遍历 SELECT 语句的结果
pg_listen	为 NOTIFY 消息建立回调
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	读取一个大对象
pg_lo_write	写入一个大对象
pg_lo_lseek	在大对象中定位到某个位置
pg_lo_tell	返回大对象的当前寻位位置
pg_lo_unlink	删除一个大对象
pg_lo_import	把 Unix 文件导入为大对象
pg_lo_export	把大对象导出到 Unix 文件

pg_lo_* 例程是 PostgreSQL 大对象特性的接口。这些函数的设计模仿标准 Unix 文件系统接口中的类似文件系统函数。pg_lo_* 例程应当在 BEGIN/COMMIT 事务块内使用，因为 pg_lo_open 返回的文件描述符只在当前事务内有效。pg_lo_import 和 pg_lo_export 必须在 BEGIN/COMMIT 事务块内使用。

例 4.1展示了一个使用这些例程的小例子。

例 4.1. 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
}
```

4.2. 把 pgtcl 装载进你的应用程序

在使用 pgtcl 命令之前，必须把 `libpgtcl` 装载进你的 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`。

除非系统的动态加载器知道到哪里寻找 `libpgtcl` 共享库文件，否则 `load` 命令会失败。你可能需要使用 `ldconfig`，或设置环境变量 `LD_LIBRARY_PATH`，或在你的平台上使用某种等价设施才能使其工作。更多信息参见 PostgreSQL 安装说明。

`libpgtcl` 又依赖于 `libpq`，因此动态加载器还必须能找到 `libpq` 共享库。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目录中，但在某些配置下可能成为绊脚石。

如果你的应用程序使用自定义的可执行文件，可以选择把 `libpgtcl` 静态绑定进可执行文件，从而避免 `load` 命令和动态链接的潜在问题。示例参见 pgtclsh 的源代码。

4.3. pgtcl 命令参考信息

pg_connect

pg_connect — 打开一个到后端服务器的连接

大纲

```
pg_connect -conninfo connectOptions
pg_connect dbName [-host hostName]
  [-port portNumber] [-tty pqtty]
  [-options optionalBackendArgs]
```

输入（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 的 PQconnectdb() 手册页中找到。

输入（旧风格）

dbName

指定一个有效的数据库名。

[-host *hostName*]

指定 *dbName* 所在后端服务器的主机名。

[-port *portNumber*]

指定 *dbName* 所在后端服务器的 IP 端口号。

[-tty *pqtty*]

指定用于后端可选调试输出的文件或 tty。

[-options *optionalBackendArgs*]

指定 *dbName* 所在后端服务器的选项。

输出

dbHandle

成功时返回一个数据库连接的句柄。句柄以 pgsql 前缀开头。

描述

pg_connect 打开一个到 PostgreSQL 后端的连接。

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 语句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。
关于较新语法中可用选项的信息参见 pg_conndefaults。

用法

pg_disconnect

pg_disconnect — 关闭一个到后端服务器的连接

大纲

```
pg_disconnect dbHandle
```

输入

dbHandle

指定一个有效的数据库句柄。

输出

None

描述

`pg_disconnect` 关闭一个到 PostgreSQL 后端的连接。

pg_conndefaults

pg_conndefaults — 获取关于默认连接参数的信息

大纲

```
pg_conndefaults
```

输入

无。

输出

option list

结果是一个列表，描述可能的连接选项及其当前默认值。
列表中的每个条目是如下格式的子列表：

```
{optname label dispchar dispsize value}
```

其中 *optname* 可用作 `pg_connect -conninfo` 中的一个选项。

描述

`pg_conndefaults` 返回关于 `pg_connect -conninfo` 中可用的连接选项及每个选项当前默认值的信息。

用法

```
pg_conndefaults
```

pg_exec

pg_exec — 向服务器发送一个命令字符串

大纲

```
pg_exec dbHandle queryString
```

输入

dbHandle

指定一个有效的数据库句柄。

queryString

指定一个有效的 SQL 查询。

输出

resultHandle

如果 pgtcl 未能获得后端响应，将返回一个 Tcl 错误。否则，会创建一个查询结果对象并返回其句柄。可以把这个句柄传给 `pg_result` 以获取查询的结果。

描述

`pg_exec` 向 PostgreSQL 后端提交一个查询并返回结果。查询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 Tcl 错误并不证明查询成功！后端返回的错误消息会作为带有失败状态的查询结果被处理，而不是让 pg_exec 产生 Tcl 错误。

pg_result

pg_result — 获取关于查询结果的信息

大纲

```
pg_result resultHandle resultOption
```

输入

resultHandle

一个查询结果的句柄。

resultOption

指定若干可能选项之一。

选项

`-status`

结果的状态。

`-error`

错误消息（如果状态表示出错）；否则为空字符串。

`-conn`

产生该结果的连接。

`-oid`

如果命令是 INSERT，为所插入元组的 OID；否则为空字符串。

`-numTuples`

查询返回的元组数。

`-numAttrs`

每个元组中的属性数。

`-assign arrayName`

把结果赋给一个数组，使用形如 `(tupno,attributeName)` 的下标。

`-assignbyidx arrayName ?appendstr?`

把结果赋给一个数组，用第一个属性的值和其余属性的名作为键。如果给出了 *appendstr*，则把它追加到每个键之后。简而言之，每个元组中除第一个字段外的所有字段都存入数组，使用形如 `(firstFieldValue,fieldNameAppendStr)` 的下标。

`-getTuple tupleNumber`

以列表形式返回所指元组的各字段。元组编号从零开始。

`-tupleArray tupleNumber arrayName`

把该元组的各字段存入数组 *`arrayName`*，以字段名为索引。元组编号从零开始。

`-attributes`

返回元组各属性名的列表。

`-lAttributes`

返回子列表的列表，每个元组属性对应一个 `{name ftype fsize}`。

`-clear`

清除该查询结果对象。

输出

结果取决于所选的选项，如上所述。

描述

`pg_result` 返回关于由先前的 `pg_exec` 创建的查询结果的信息。

查询结果可以保留任意长的时间，但用完之后，务必通过执行 `pg_result -clear` 将其释放。否则会造成内存泄漏，Pgtcl 最终会抱怨你创建了太多查询结果对象。

pg_select

pg_select — 循环遍历 SELECT 语句的结果

大纲

pg_select *dbHandle queryString arrayVar queryProcedure*

输入

dbHandle

指定一个有效的数据库句柄。

queryString

指定一个有效的 SQL select 查询。

arrayVar

用于存放返回元组的数组变量。

queryProcedure

对找到的每个元组运行的过程。

输出

无。

描述

pg_select 向 PostgreSQL 后端提交一个 SELECT 查询，并对结果中的每个元组执行一段给定的代码。*queryString* 必须是一条 SELECT 语句。其他语句都会返回错误。*arrayVar* 变量是循环中使用的数组名。对每个元组，*arrayVar* 都会以字段名作为数组下标填入该元组的字段值。然后执行 *queryProcedure*。

除字段值外，数组中还会写入以下特殊条目：

.headers

SELECT 返回的列名列表。

.numcols

SELECT 返回的列数。

.tupno

当前元组编号，从零开始，循环体每迭代一次递增一次。

用法

如果表 table 有字段 control 和 name（或许还有其他字段），下面这样写即可：

```
 pg_select $pgconn "SELECT * FROM table" array {
```

```
 puts [format "%5d %s" $array(control) $array(name)]
}
```

pg_listen

pg_listen — 设置或更改异步 NOTIFY 消息的回调

大纲

```
pg_listen dbHandle notifyName callbackCommand
```

输入

dbHandle

指定一个有效的数据库句柄。

notifyName

指定要开始或停止监听的 notify 条件名。

callbackCommand

如果提供且非空，给出匹配的通知到达时要执行的命令字符串。

输出

None

描述

`pg_listen` 创建、更改或取消对来自 PostgreSQL 后端的异步 NOTIFY 消息的监听请求。带 *callbackCommand* 参数时，会建立请求，或替换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。

`pg_listen` 请求建立之后，每当带有给定名称的 NOTIFY 消息从后端到达时，就执行指定的命令字符串。任何 PostgreSQL 客户端应用程序发出引用该名称的 NOTIFY 命令时都会发生这种情况。（注意，该名称可以是数据库中已存在关系的名称，但不必一定如此。）命令字符串在 Tcl 空闲循环中执行。这是用 Tk 编写的应用程序的正常空闲状态。在非 Tk 的 Tcl shell 中，可以执行 `update` 或 `vwait` 来进入空闲循环。

使用 `pg_listen` 时，不应直接调用 SQL 语句 `LISTEN` 或 `UNLISTEN`。Pgtcl 会替你发出这些语句。但如果想自己发送 NOTIFY 消息，可以用 `pg_exec` 调用 SQL 的 NOTIFY 语句。

pg_lo_creat

pg_lo_creat — 创建一个大对象

大纲

```
pg_lo_creat conn mode
```

输入

conn

指定一个有效的数据库连接。

mode

指定大对象的访问模式

输出

objOid

所创建大对象的 oid。

描述

`pg_lo_creat` 创建一个 Inversion 大对象。

用法

mode 可以是 `INV_READ` 和 `INV_WRITE` 的任意按位或组合。“或”操作符是 `|`。

```
[pg_lo_creat $conn "INV_READ|INV_WRITE"]
```

pg_lo_open

pg_lo_open — 打开一个大对象

大纲

```
pg_lo_open conn objOid mode
```

输入

conn

指定一个有效的数据库连接。

objOid

指定一个有效的大对象 oid。

mode

指定大对象的访问模式

输出

fd

供后续 pg_lo* 例程使用的文件描述符。

描述

`pg_lo_open` 打开一个 Inversion 大对象。

用法

模式可以是 `r`、`w` 或 `rw`。

pg_lo_close

pg_lo_close — 关闭一个大对象

大纲

```
pg_lo_close conn fd
```

输入

conn

指定一个有效的数据库连接。

fd

供后续 pg_lo* 例程使用的文件描述符。

输出

无

描述

`pg_lo_close` 关闭一个 Inversion 大对象。

用法

pg_lo_read

pg_lo_read — 读取一个大对象

大纲

```
pg_lo_read conn fd bufVar len
```

输入

conn

指定一个有效的数据库连接。

fd

pg_lo_open 返回的大对象文件描述符。

bufVar

指定一个有效的缓冲区变量，用于容纳大对象片段。

len

指定大对象片段的最大允许大小。

输出

无

描述

`pg_lo_read` 从大对象中读取至多 *len* 字节到名为 *bufVar* 的变量中。

用法

bufVar 必须是有效的变量名。

pg_lo_write

pg_lo_write — 写入一个大对象

大纲

```
pg_lo_write conn fd buf len
```

输入

conn

指定一个有效的数据库连接。

fd

pg_lo_open 返回的大对象文件描述符。

buf

指定要写入大对象的有效字符串变量。

len

指定要写入字符串的最大大小。

输出

无

描述

`pg_lo_write` 从变量 *buf* 向大对象写入至多 *len* 字节。

用法

buf 必须是要写入的实际字符串，而不是变量名。

pg_lo_lseek

pg_lo_lseek — 在大对象中定位到某个位置

大纲

```
pg_lo_lseek conn fd offset whence
```

输入

conn

指定一个有效的数据库连接。

fd

pg_lo_open 返回的大对象文件描述符。

offset

指定以字节计的从零开始的偏移量。

whence

whence 可以是 SEEK_CUR、SEEK_END 或 SEEK_SET

输出

无

描述

pg_lo_lseek 定位到距大对象开头 *offset* 字节处。

用法

whence 可以是 SEEK_CUR、SEEK_END 或 SEEK_SET。

pg_lo_tell

pg_lo_tell — 返回大对象的当前寻位位置

大纲

```
pg_lo_tell conn fd
```

输入

conn

指定一个有效的数据库连接。

fd

pg_lo_open 返回的大对象文件描述符。

输出

offset

以字节计的从零开始的偏移量，适合作为 `pg_lo_lseek` 的输入。

描述

`pg_lo_tell` 返回当前距大对象开头的 *offset*（以字节计）。

用法

pg_lo_unlink

pg_lo_unlink — 删除一个大对象

大纲

```
pg_lo_unlink conn lobjId
```

输入

conn

指定一个有效的数据库连接。

lobjId

大对象的标识符。

输出

无

描述

`pg_lo_unlink` 删除指定的大对象。

用法

pg_lo_import

pg_lo_import — 从文件导入一个大对象

大纲

```
pg_lo_import conn filename
```

输入

conn

指定一个有效的数据库连接。

filename

Unix 文件名。

输出

无

描述

`pg_lo_import` 读取指定的文件并把内容放入一个大对象。

用法

`pg_lo_import` 必须在 BEGIN/END 事务块内调用。

pg_lo_export

pg_lo_export — 把大对象导出到文件

大纲

```
pg_lo_export conn lobjId filename
```

输入

conn

指定一个有效的数据库连接。

lobjId

大对象标识符。

filename

Unix 文件名。

输出

无

描述

`pg_lo_export` 把指定的大对象写入一个 Unix 文件。

用法

`pg_lo_export` 必须在 BEGIN/END 事务块内调用。

第 5 章 libpgeasy - 简化 C 库

作者

由 Bruce Momjian 编写（<pgman@candle.pha.pa.us>），最后更新于 2000-03-30

pgeasy 让你能够干净地接口到 libpq 库，更像是 4GL 的 SQL 接口。关于 libpq 的更多信息请参考第 1 章。

它由一组封装了 libpq 功能的简化 C 函数组成。这些函数是：

- `PGresult   *doquery(char *query);`
- `PGconn     *connectdb(char *options);`
- `void        disconnectdb();`
- `int         fetch(void *param,...);`
- `int         fetchwithnulls(void *param,...);`
- `void        reset_fetch();`
- `void        on_error_continue();`
- `void        on_error_stop();`
- `PGresult   *get_result();`
- `void        set_result(PGresult *newres);`
- `void        unset_result(PGresult *oldres);`

许多函数会返回一个结构或值，因此在需要时你可以对结果做更多的工作。

基本上你用 `connectdb` 连接到数据库，用 `doquery` 发出查询，用 `fetch` 取回结果，最后用 `disconnectdb` 结束。

对于 `SELECT` 查询，`fetch` 允许你把指针作为参数传递，返回时这些变量将被你打开的二进制游标中的数据填充。如果你运行 pgeasy 客户端的系统与数据库服务器的体系结构不同，这些二进制游标就无法使用。如果你传递一个 NULL 指针参数，该列会被跳过。`fetchwithnulls` 允许你取回字段的 NULL 状态，方法是在每个结果指针之后传递一个 `int*`，字段为空时它返回真或假。你总是可以在 `doquery` 返回的 `PGresult` 指针上使用 libpq 函数。`reset_fetch` 把取回操作重置回开头。

`get_result`、`set_result` 和 `unset_result` 允许你同时处理多个结果集。

源代码目录中有若干示例程序。

第 6 章 ecpg - C 中的嵌入式 SQL

Linus Tolke
Michael Meskes

本文介绍 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@debian.org>）编写。该软件包随 PostgreSQL 发行版一起安装，并带有类似的许可证。

6.1. Why Embedded SQL?

与处理 SQL 查询的其他方法相比，嵌入式 SQL 有其优势。它负责了在 C 或 C++ 程序的变量之间传递信息的繁琐工作。许多 RDBMS 软件包都支持这种嵌入式语言。

有一个 ANSI 标准描述了这种嵌入式语言应当如何工作。ecpg 的设计尽可能符合这个标准。可以把为其他 RDBMS 编写的嵌入式 SQL 程序移植到 PostgreSQL。

6.2. 概念

你用带有特殊 SQL 构造的 C/C++ 编写程序。声明要在 SQL 语句中使用的变量时，需要把它们放在特殊的 `declare` 区中。SQL 查询要使用特殊的语法。

编译之前，先用嵌入式 SQL C 预处理器处理文件，它把你使用的 SQL 语句转换成以变量为参数的函数调用。查询的输入变量和结果输出变量都会被传递。

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 查询，并把结果放到指定用于输出的参数中。

6.3. How To Use ecpg

本节介绍如何使用 ecpg。

6.3.1. Preprocessor

预处理器叫做 ecpg。安装后它位于 PostgreSQL 的 `bin/` 目录中。

6.3.2. Library

ecpg 库叫做 `libecpg.a` 或 `libecpg.so`。此外，该库用 `libpq` 库与 PostgreSQL 服务器通信。你必须用 *`-lecpg -lpq`* 链接你的程序。

这个库还有一些“隐藏”但可能证明有用的方法。

- `ECPGdebug(int` *`on`*`, FILE *`*`stream`*`)` 在第一个参数非零时打开调试日志。Debug logging is done on *`stream`*. Most SQL statement log their arguments and results.

  最重要的函数 `ECPGdo` 会同时用展开后的字符串（即插入了所有输入变量的字符串）和 PostgreSQL 服务器的结果记录所有 SQL 语句。在查找 SQL 语句中的错误时这非常有用。

- `ECPGstatus()` 这个方法在已连接到数据库时返回 TRUE，否则返回 FALSE。

6.3.3. 错误处理

为了检测来自 PostgreSQL 服务器的错误，包含像下面这样的一行：

```
exec sql include sqlca;
```

这会定义一个结构以及一个名为 *sqlca* 的变量，如下：

```
struct sqlca
{
 char sqlcaid[8];
 long sqlabc;
 long sqlcode;
 struct
 {
  int sqlerrml;
  char sqlerrmc[70];
 } sqlerrm;
 char sqlerrp[8];
 long sqlerrd[6];
 /* 0: empty                                        */
 /* 1: OID of processed tuple if applicable         */
 /* 2: number of rows processed in an INSERT, UPDATE */
 /*    or DELETE statement                          */
 /* 3: empty                                        */
 /* 4: empty                                        */
 /* 5: empty                                        */
 char sqlwarn[8];
 /* 0: set to 'W' if at least one other is 'W'      */
 /* 1: if 'W' at least one character string         */
 /*    value was truncated when it was              */
 /*    stored into a host variable.                 */
 /* 2: empty                                        */
 /* 3: empty                                        */
 /* 4: empty                                        */
 /* 5: empty                                        */
 /* 6: empty                                        */
 /* 7: empty                                        */
 char sqlext[8];
} sqlca;
```

如果上一条 SQL 语句没有发生错误，*sqlca.sqlcode* 将为 0（ECPG_NO_ERROR）。如果 *sqlca.sqlcode* 小于零，这是严重的错误，例如数据库定义与查询不匹配。如果大于零，则是普通错误，例如表中没有所请求的行。

sqlca.sqlerrm.sqlerrmc 将包含一个描述错误的字符串。该字符串以源文件中的行号结尾。

可能发生的错误有：

```
-12, Out of memory in line %d.
```

通常不应出现。这表示你的虚拟内存已耗尽。

`-200 (ECPG_UNSUPPORTED): Unsupported type %s on line %d.`

通常不应出现。这表示预处理器生成了库不认识的内容。
也许你运行的预处理器和库的版本不兼容。

`-201 (ECPG_TOO_MANY_ARGUMENTS): Too many arguments line %d.`

这表示 PostgreSQL 返回的参数比我们拥有的匹配变量多。也许你在 `INTO :var1,:var2` 列表中漏掉了几个宿主变量。

`-202 (ECPG_TOO_FEW_ARGUMENTS): Too few arguments line %d.`

这表示 PostgreSQL 返回的参数比我们拥有的宿主变量少。也许你在 `INTO :var1,:var2` 列表中放了太多宿主变量。

`-203 (ECPG_TOO_MANY_MATCHES): Too many matches line %d.`

这表示查询返回了多行，但指定的变量不是数组。`SELECT` 命令不唯一。

`-204 (ECPG_INT_FORMAT): Not correctly formatted int type: %s line %d.`

这表示宿主变量的类型是 `int`，而 PostgreSQL 数据库中的字段是另一种类型，包含不能被解释为 `int` 的值。库用 `strtol()` 做这个转换。

`-205 (ECPG_UINT_FORMAT): Not correctly formatted unsigned type: %s line %d.`

这表示宿主变量的类型是 `unsigned int`，而 PostgreSQL 数据库中的字段是另一种类型，包含不能被解释为 `unsigned int` 的值。库用 `strtoul()` 做这个转换。

`-206 (ECPG_FLOAT_FORMAT): Not correctly formatted floating-point type: %s line %d.`

这表示宿主变量的类型是 `float`，而 PostgreSQL 数据库中的字段是另一种类型，包含不能被解释为 `float` 的值。库用 `strtod()` 做这个转换。

`-207 (ECPG_CONVERT_BOOL): Unable to convert %s to bool on line %d.`

这表示宿主变量的类型是 `bool`，而 PostgreSQL 数据库中的字段既不是 `'t'` 也不是 `'f'`。

`-208 (ECPG_EMPTY): Empty query line %d.`

PostgreSQL 返回了 `PGRES_EMPTY_QUERY`，很可能因为查询确实是空的。

`-209 (ECPG_MISSING_INDICATOR): NULL value without indicator in line %d.`

PostgreSQL 返回 `ECPG_MISSING_INDICATOR`，因为返回了 NULL 而没有提供 NULL 指示符变量。

`-210 (ECPG_NO_ARRAY): Variable is not an array in line %d.`

PostgreSQL 返回 `ECPG_NO_ARRAY`，因为在需要数组的地方使用了普通变量。

`-211 (ECPG_DATA_NOT_ARRAY): Data read from backend is not an array in line %d.`

PostgreSQL 返回 `ECPG_DATA_NOT_ARRAY`，因为数据库在需要数组值的地方返回了普通变量。

`-220 (ECPG_NO_CONN): No such connection %s in line %d.`

程序试图访问一个不存在的连接。

`-221 (ECPG_NOT_CONN): Not connected in line %d.`

程序试图访问一个确实存在但未打开的连接。

`-230 (ECPG_INVALID_STMT): Invalid statement name %s in line %d.`

你试图使用的语句还没有被准备。

`-240 (ECPG_UNKNOWN_DESCRIPTOR): Descriptor %s not found in line %d.`

找不到指定的描述符。你试图使用的语句还没有被准备。

`-241 (ECPG_INVALID_DESCRIPTOR_INDEX): Descriptor index out of range in line %d.`

指定的描述符索引超出范围。

`-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM): Descriptor %s not found in line %d.`

找不到指定的描述符。你试图使用的语句还没有被准备。

`-243 (ECPG_VAR_NOT_NUMERIC): Variable is not a numeric type in line %d.`

数据库返回了数值而变量不是数值型。

`-244 (ECPG_VAR_NOT_CHAR): Variable is not a character type in line %d.`

数据库返回了非数值而变量是数值型。

`-400 (ECPG_PGSQL): Postgres error: %s line %d.`

某种 PostgreSQL 错误。消息中包含来自 PostgreSQL 后端的错误消息。

`-401 (ECPG_TRANS): Error in transaction processing line %d.`

PostgreSQL 表示我们无法开始、提交或回滚事务。

`-402 (ECPG_CONNECT): Could not connect to database %s in line %d.`

连接数据库失败。

`100 (ECPG_NOT_FOUND): Data not found line %d.`

这是一个“正常”错误，告诉你所查询的内容找不到，或者你已经位于游标的末尾。

6.4. Limitations

永远不会被包含的内容以及为什么做不到：

Oracle 的单任务

AIX 3 上的 Oracle 7.0 版使用共享内存中由 OS 支持的锁，允许应用设计者以“单任务”的方式链接应用。不是每个应用进程启动一个客户端进程，
而是数据库部分和应用部分运行在同一个进程中。在 Oracle 的后续版本中这已不再受支持。

这需要彻底重新设计 PostgreSQL 的访问模型，而性能收益不值得付出这种努力。

6.5. 从其他 RDBMS 软件包移植

ecpg 的设计遵循 SQL 标准。从标准的 RDBMS 移植应该不成问题。遗憾的是并不存在所谓标准的 RDBMS。因此 ecpg 会尽量理解语法扩展，只要它们不与标准冲突。

下面列出所有已知的不兼容之处。如果你发现未列出的问题，请通知开发者。不过要注意，
我们只列出从其他 RDBMS 的预编译器到 ecpg 的不兼容，而不列出这些 RDBMS 不支持的 ecpg 特性。

FETCH 的语法

FETCH 的标准语法是：

FETCH [direction] [amount] IN|FROM *cursor*.

然而 Oracle 不使用关键字 IN 或 FROM。这个特性无法添加，因为会造成解析冲突。

6.6. 给开发者

本节解释 ecpg 的内部工作原理。它包含有助于用户理解如何使用 ecpg 的宝贵信息。

6.6.1. The Preprocessor

`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。

当它遇到 `EXEC SQL` 语句时，会介入并改变它。`EXEC SQL` 语句可以是下列之一：

声明节

`Declare` 节以：

```
exec sql begin declare section;
```

开始，以：

```
exec sql end declare section;
```

结束。此节内只允许变量声明。
此节内声明的每个变量都按名称索引连同其对应类型存储在一个变量列表中。

特别是结构或联合的定义也必须列在 `declare` 区内。否则 ecpg 无法处理这些类型，因为它不知道定义。

声明也会被回显到文件中，使其成为普通的 C 变量。

特殊类型 `VARCHAR` 和 `VARCHAR2` 会为每个变量转换成一个命名的结构。像这样的声明：

```
VARCHAR var[180];
```

会被转换为：

```
struct varchar_var { int len; char arr[180]; } var;
```

包含语句

include 语句的形式是：

```
exec sql include filename;
```

注意这不同于：

```
#include <filename.h>
```

Instead the file specified is parsed by ecpg so the contents of the file are included in the resulting C code. This way you are able to specify EXEC SQL commands in an include file.

连接语句

connect 语句的形式是：

```
exec sql connect to connection target;
```

它创建到指定数据库的一个连接。

connection target 可以用下列方式指定：

- dbname[@server][:port][as *connection name*][user *user name*]
- tcp:postgresql://server[:port][/dbname][as *connection name*][user *user name*]
- unix:postgresql://server[:port][/dbname][as *connection name*][user *user name*]
- *character variable*[as *connection name*][user *user name*]
- *character string*[as *connection name*][*user*]
- default
- user

还有几种指定用户名的方式：

- *userid*
- *userid*/*password*
- *userid* identified by *password*
- *userid* using *password*

最后，*userid* 和 *password* 可以是常量文本、字符变量或字符串。

断开语句

disconnect 语句的形式是：

```
exec sql disconnect [connection target];
```

它关闭到指定数据库的连接。

connection target 可以用下列方式指定：

- *connection name*
- default
- current
- all

打开游标语句

open cursor 语句的形式是：

```
exec sql open cursor;
```

并且不被复制到输出中。相反，会使用游标的 DECLARE 命令，因为它也会打开游标。

提交语句

commit 语句的形式是：

```
exec sql commit;
```

回滚语句

rollback 语句的形式是：

```
exec sql rollback;
```

其他语句

其他 SQL 语句以 exec sql 开始、以 ; 结束的方式使用。中间的一切都被当作 SQL 语句并解析变量替换。

当符号以冒号（:）开头时发生变量替换。此时会在先前于 declare 节中声明的变量里查找具有该名字的变量。根据该变量用于输入还是输出，
指向该变量的指针会被写到输出中以允许函数访问。

对于作为 SQL 查询一部分的每个变量，函数会得到其他参数：

- The type as a special symbol.
- 指向值的指针，或指向指针的指针。
- 如果变量是 char 或 varchar，则为变量的大小。
- 数组中的元素个数（用于数组抓取）。
- 到数组中下一个元素的偏移（用于数组抓取）。
- 作为特殊符号的指示符变量的类型。

- 指向指示符变量值的指针，或指向指示符变量指针的指针。
- 0.
- 指示符数组中的元素个数（用于数组抓取）。
- 到指示符数组中下一个元素的偏移（用于数组抓取）。

6.6.2. 一个完整的例子

下面是一个完整的例子，描述预处理器对文件 `foo.pgc` 的输出：

```
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;
```

is translated into:

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int result;
/* exec sql end declare section */
...
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"
```

(The indentation in this manual is added for readability and not something the preprocessor does.)

6.6.3. 库

库中最重要的函数是 `ECPGdo`。它接受数量可变的参数。但愿没有计算机会限制 `varargs()` 函数可接受的变量数。这很容易累积到 50 个左右的参数。

参数有：

A line number

这是原语句的行号；只用于错误消息。

A string

这是要发出的 SQL 查询。它被输入变量修改，即那些编译时未知但要输入查询的变量。在变量应出现的位置，字符串中包含 ?。

输入变量

如关于预处理器的一节所述，每个输入变量得到十个参数。

ECPGt_EOIT

一个枚举，表示不再有输入变量。

输出变量

如关于预处理器的一节所述，每个输入变量得到十个参数。These variables are filled by the function.

ECPGt_EORT

一个枚举，表示不再有变量。

在默认模式下，只有在发出 `exec sql commit` 时查询才被提交。Ecpg 还通过 `-t` 命令行选项或 `exec sql set autocommit to on` 语句支持事务的自动提交。在 `autocommit` 模式下，除非处于显式的事务块中，每条查询都会自动提交。这个模式可以用 `exec sql set autocommit to off` 显式关闭。

第 7 章 ODBC 接口

Tim Goeke
Thomas Lockhart

7.1. 简介

> **注意**
>
> 背景信息最初由 Tim Goeke（<tgoeke@xpressway.com>）提供

ODBC（开放数据库互连）是一个抽象的 API，它让你能够编写可与各种 RDBMS服务器互操作的应用。ODBC在前端应用和数据库服务器之间提供了一个与产品无关的接口，
让用户或开发者能够编写在不同厂商的服务器之间可移植的应用。

ODBC API在后端对应一个 ODBC兼容的数据源。它可以是任何东西，从一个文本文件到 Oracle或PostgreSQL RDBMS。

后端访问来自ODBC驱动，或者说来自允许访问数据的厂商特定驱动。
psqlODBC就是这样一种驱动，它包含在PostgreSQL 发行版中；此外还有其他可用的驱动，例如OpenLink的ODBC驱动。

一旦你编写了一个ODBC应用，只要数据库模式相同，无论后端数据库来自哪个厂商，你
都应该能连接到任何后端数据库。

例如，你可以有一台MS SQL Server服务器和一台数据完全相同的PostgreSQL服务器。使用ODBC，你的 Windows 应用发出的调用将完全相同，后端数据源（在 Windows 应用看来）看起来也一样。

7.2. 安装

要使用ODBC驱动，在ODBC 驱动要使用的系统上必须存在一个驱动管理器。据我们所知，类Unix 操作系统上有两种免费的 ODBC驱动管理器：iODBC[1] 和 unixODBC[2]。安装这些驱动管理器的说明可以在各自的发行版中找到。通过 ODBC提供数据库访问的软件应当提供它自己的驱动管理器（很可能就是这两种之一）。话虽如此，你能为你的平台找到的任何驱动管理器都应该支持PostgreSQL 的ODBC驱动，或者任何其他ODBC 驱动。

> **注意**
>
> unixODBC发行版自带了一个 PostgreSQL的ODBC 驱动，它与PostgreSQL发行版中包含的驱动类似。想用哪一个由你决定。我们计划在未来更好地协调两个驱动的开发。

要安装ODBC，你只需在构建整个 PostgreSQL发行版时向 `configure`脚本提供 `--enable-odbc`选项。这样该库就会随其余程序一起被自动构建和安装。

[1] http://www.iodbc.org
[2] http://www.unixodbc.org

如果你忘了这个选项或者想稍后构建 ODBC 驱动，可以进入目录src/interfaces/odbc并在那里执行 make和make install。

也可以把驱动构建成专门为iODBC或 unixODBC调优的形式。这尤其意味着驱动将使用驱动管理器的例程来处理配置文件，这可能是令人满意的，因为它在你的系统上创建了一个更一致的ODBC环境。如果你想这样做，就向configure提供 --with-iodbc或--with-unixodbc 选项（但不要同时提供）。

如果你构建的是“独立”驱动（不绑定 iODBC或unixODBC），那么你可以指定驱动应当到哪里查找配置文件 odbcinst.ini。默认情况下它会是 /usr/local/pgsql/etc/目录或等价目录，具体取决于你向configure提供了哪些 --prefix和/或--sysconfdir选项。要选择PostgreSQL安装布局之外的特定位置，可以使用--with-odbcinst选项。
为了获得最好的效果，应当设法让驱动和驱动管理器读取同一个配置文件。

此外，你应当安装 ODBC 目录扩展。它会提供一批 ODBC 标准规定、而PostgreSQL默认不提供的函数。文件 /usr/local/pgsql/share/odbc.sql（在默认安装布局中）包含适当的定义，你可以这样安装：

```
psql -d template1 -f LOCATION/odbc.sql
```

把template1指定为目标数据库可以保证之后所有新建的数据库都拥有这些相同的定义。
如果出于某种原因你想再删除这些函数，用psql执行 odbc-drop.sql文件即可。

7.3. 配置文件

~/.odbc.ini包含用户为 psqlODBC驱动指定的访问信息。该文件使用Windows注册表文件典型的约定。

.odbc.ini文件有三个必需的节。第一个是 [ODBC Data Sources]，它是你想访问的每个数据库的任意名称和描述的列表。第二个必需的节是数据源说明，
每个数据库都会有一个这样的节。每个节必须用 [ODBC Data Sources]中给出的名称标记，并且必须包含下列条目：

```
Driver = prefix/lib/libpsqlodbc.so
Database = DatabaseName
Servername = localhost
Port = 5432
```

> **提示**
>
> 记住PostgreSQL数据库名通常是单个词，不带任何形式的路径名。PostgreSQL 服务器管理对数据库的实际访问，你只需要从客户端指定名字。

还可以插入其他条目来控制显示的格式。第三个必需的节是 [ODBC]，它必须包含InstallDir 关键字，还可以包含其他选项。

下面是一个.odbc.ini文件的例子，展示三个数据库的访问信息：

```
[ODBC Data Sources]
DataEntry = Read/Write Database
QueryOnly = Read-only Database
Test = Debugging Database
Default = Postgres Stripped

[DataEntry]
```

```
ReadOnly = 0
Servername = localhost
Database = Sales

[QueryOnly]
ReadOnly = 1
Servername = localhost
Database = Sales

[Test]
Debug = 1
CommLog = 1
ReadOnly = 0
Servername = localhost
Username = tgl
Password = "no$way"
Port = 5432
Database = test

[Default]
Servername = localhost
Database = tgl
Driver = /opt/postgres/current/lib/libpsqlodbc.so

[ODBC]
InstallDir = /opt/applix/axdata/axshlib
```

7.4. Windows应用

在现实世界中，驱动之间的差异以及ODBC支持水平的不同削弱了ODBC的潜力：

- Access、Delphi 和 Visual Basic 都直接支持ODBC。
- 在 C++（例如 Visual C++）下，你可以使用 C++ 的 ODBC API。
- 在 Visual C++ 中，你可以使用`CRecordSet`类，它把ODBC API集封装在一个 MFC 4.2 类中。如果你在 Windows NT 下进行 Windows C++ 开发，这是最容易的路线。

7.4.1. 编写应用

“如果我为PostgreSQL编写一个应用，我能否通过ODBC调用访问 PostgreSQL服务器来编写它，还是说只有当 MS SQL Server 或 Access 之类的其他数据库程序需要访问数据时才这么做？”

ODBC API正是该走的路。关于Visual C++编程，你可以在 Microsoft 的网站或你的Visual C++ 文档中找到更多信息。

Visual Basic 和其他RAD工具有 `Recordset`对象，它们直接使用 ODBC访问数据。使用数据感知控件，你可以（非常迅速地）快速链接到 ODBC后端数据库。

玩玩MS Access会帮你弄清楚这一切。试着使用 File → Get External Data。

> **提示**
>
> 你必须先设置一个DSN。

7.5. ApplixWare

ApplixWare有一个至少在部分平台上受到支持的ODBC数据库接口。在 Linux 下，已经演示过 ApplixWare 4.4.2 通过 PostgreSQL发行版中包含的 psqlODBC驱动与 PostgreSQL 7.0 配合工作。

7.5.1. 配置

ApplixWare必须被正确配置，才能访问 PostgreSQL的ODBC 软件驱动。

启用ApplixWare数据库访问

这些说明针对的是Linux上的 ApplixWare 4.4.2 版。更详细的信息请参阅Linux Sys Admin在线图书。

1. 你必须修改`axnet.cnf`，让 `elfodbc`能找到`libodbc.so`（ODBC驱动管理器）共享库。这个库随 ApplixWare发行版一起提供，但 `axnet.cnf`需要被修改为指向正确的位置。

   以 root 身份编辑文件 *`applixroot`*`/applix/axdata/axnet.cnf`。

   a. 在`axnet.cnf`的底部，找到以

      ```
      #libFor elfodbc /ax/...
      ```

   b. 把这一行改为

      ```
      libFor elfodbc applixroot/applix/axdata/axshlib/lib
      ```

      这会告诉`elfodbc`到这个目录中查找 ODBC支持库。通常 Applix安装在`/opt`，因此完整路径是 `/opt/applix/axdata/axshlib/lib`，但如果你把 Applix安装在了其他地方，请相应地修改路径。

2. 按照第 7.3 节中的说明创建 `.odbc.ini`。你可能还想加上标志

   ```
   TextAsLongVarchar=0
   ```

   加到`.odbc.ini`的数据库特定部分，这样文本字段就不会显示为`**BLOB**`。

测试ApplixWare ODBC 连接

1. 启动Applix Data
2. 选择你感兴趣的PostgreSQL数据库。

   a. 选择Query → Choose Server。

   b. 选择ODBC，然后点击 Browse。你在`.odbc.ini` 中配置的数据库应该会被显示出来。确保Host: 字段为空（如果不为空，`axnet`会试图联系另一台机器上的`axnet`来查找数据库）。

   c. 在Browse弹出的框中选择数据库，然后点击OK。

   d. 在登录标识对话框中输入用户名和口令，然后点击OK。

   你应该会在数据窗口左下角看到 Starting elfodbc server。如果你得到一个错误对话框，请参见下面的调试一节。

3. “Ready” 消息会出现在数据窗口左下角。这表示你现在可以输入查询了。

4. 从Query → Choose tables中选择一张表，然后选择 Query → Query 来访问数据库。表中前 50 行左右应该会出现。

7.5.2. 常见问题

在试图通过Applix Data建立 ODBC连接时可能出现下列消息：

```
Cannot launch gateway on server
```

elfodbc找不到libodbc.so。检查你的axnet.cnf。

```
Error from ODBC Gateway: IM003::[iODBC][Driver Manager]Specified driver
could not be loaded
```

libodbc.so找不到 .odbc.ini中列出的驱动。核对设置。

```
Server: Broken Pipe
```

驱动进程由于某种其他问题已经终止。你可能没有最新版本的 PostgreSQL ODBC包。

```
setuid to 256: failed to launch gateway
```

ApplixWare 4.4.1 的九月版（第一个在 Linux 下正式支持ODBC的版本）在用户名长度超过八（8）个字符时会出问题。问题描述由 Steve Campbell（<scampbell@lear.com>）提供。

作者

由 Steve Campbell（<scampbell@lear.com>）供稿，1998-10-20

axnet程序的安全系统看起来有点可疑。axnet代表用户做事，
在一个真正的多用户系统上它确实应该以 root 权限运行（这样它才能在每个用户的目录中读写）。不过我不太愿意推荐这样做，因为我们完全不知道这会打开什么安全漏洞。

7.5.3. 调试ApplixWare ODBC 连接

调试连接问题的一个好工具是 Unix 系统实用程序 strace。

用strace调试

1. 启动ApplixWare。

2. 对axnet进程启动 strace。例如，如果

   $**ps -aucx | grep ax**

   显示

   ```
   cary   10432  0.0  2.6  1740   392  ?  S  Oct  9  0:00 axnet
   cary   27883  0.9 31.0 12692  4596  ?  S   10:24  0:04 axmain
   ```

   然后运行

```
$strace -f -s 1024 -p 10432
```

3. 检查strace输出。

> **来自 Cary 的提示**
>
> ApplixWare的许多错误消息会发往 stderr，但我不确定stderr 被送到了哪里，所以strace是查明这一点的手段。

例如，在得到Cannot launch gateway on server 之后，我对axnet运行了strace，得到

```
[pid 27947] open("/usr/lib/libodbc.so", O_RDONLY) = -1 ENOENT (No such
 file or directory)
[pid 27947] open("/lib/libodbc.so", O_RDONLY) = -1 ENOENT (No such
 file or directory)
[pid 27947] write(2, "/usr2/applix/axdata/elfodbc: can't load library
 'libodbc.so'\n", 61) = -1 EIO (I/O error)
```

所以发生的事情是applix elfodbc在搜索 libodbc.so但找不到它。这就是为什么需要修改axnet.cnf。

7.5.4. 运行ApplixWare演示

要完成ApplixWare Data Tutorial，你需要创建教程中引用的示例表。用于创建这些表的 ELF 宏试图在许多数据库列上使用 NULL 条件，而PostgreSQL 目前不允许这样做。

要绕开这个问题，你可以这样做：

修改ApplixWare演示

1. 把/opt/applix/axdata/eng/Demos/sqldemo.am 复制到一个本地目录。
2. 编辑这份sqldemo.am的本地副本：
   a. 搜索null_clause = "NULL"。
   b. 把它改为null_clause = ""。
3. 启动Applix Macro Editor。
4. 从Macro Editor中打开sqldemo.am文件。
5. 选择File → Compile and Save。
6. 退出Macro Editor。
7. 启动Applix Data。
8. 选择* → Run Macro。
9. 输入值sqldemo，然后点击OK。

   你应该能在数据窗口的状态行（左下角）看到进度。

10. 现在你应该可以访问演示表了。

7.5.5. 有用的宏

你可以把数据库登录和口令信息加到标准的 Applix启动宏文件中。下面是一个 `~/axhome/macros/login.am`文件的例子：

```
macro login
set_set_system_var@("sql_username@","tgl")
set_system_var@("sql_passwd@","no$way")
endmacro
```

> **小心**
>
> 对于任何包含用户名和口令信息的文件，你都应当小心设置文件保护。

第 8 章 JDBC 接口

作者

最初由 JDBC 驱动最初的作者 Peter T. Mount（<peter@retep.org.uk>）编写。

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。这里使用的是其中的基本示例。

8.1. 设置 JDBC 驱动

8.1.1. 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。不过只有在你修改源代码时才应该需要这么做。

从 PostgreSQL 7.1 版开始，JDBC 驱动使用 Ant 构建，这是一种专门用于构建基于 Java 的包的工具。在继续之前，你应该从Ant 网站[2]下载 Ant 并安装。预编译的 Ant 发行版通常会读取当前用户主目录中的 `.antrc` 文件进行配置。例如，要使用与默认值不同的 JDK，可以这样：

```
JAVA_HOME=/usr/local/sun-jdk1.3
JAVACMD=$JAVA_HOME/bin/java
```

要构建驱动，请在你的 `configure` 命令行中加上 `--with-java` 选项，例如：

```
$./configure --prefix=xxx --with-java ...
```

这样，当你执行 `make/gmake` 和 `make/gmake install` 命令时，驱动会与 PostgreSQL 包的其余部分一起构建和安装。如果你只想构建驱动而不构建 PostgreSQL 的其余部分，可以进入目录 `src/interfaces/jdbc` 并在那里执行相应的 `make/gmake` 命令。关于配置和构建过程的更多信息，请参阅 PostgreSQL 安装说明。

从源码构建驱动时，创建的 jar 文件将名为 `postgresql.jar`。构建会在 `src/interfaces/jdbc/jars` 目录中生成该文件。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 JDK 构建，会得到一个支持 jdbc1 规范的版本；如果用 Java2 JDK（即 JDK1.2 或 JDK1.3）构建，会得到一个支持 jdbc2 规范的版本。

[1] http://jdbc.postgresql.org
[2] http://jakarta.apache.org/ant/index.html

注意

不要试图通过直接调用 `javac` 来构建驱动，因为驱动出于性能原因使用了一些动态加载技术，而 `javac` 无法应对。也不要试图直接运行 `ant`，因为有些配置信息是通过 makefile 传递的。在不提供这些参数的情况下直接运行 `ant` 会得到一个有问题的驱动。

8.1.2. 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 `jdbc7.2-1.1.jar` 或 `jdbc7.2-1.2.jar`，分别对应 jdbc1 和 jdbc2 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。默认情况下，该 jar 归档安装在 `/usr/local/pgsql/share/java` 目录中。如果你运行 `configure` 时使用了 `--prefix` 选项，或者你使用的二进制发行版把它放在了别的位置，那么它可能在不同的目录中。

例如，我有一个应用程序使用 JDBC 驱动访问一个包含天体数据的大型数据库。该应用程序和 JDBC 驱动都安装在 `/usr/local/lib` 目录中，而 Java JDK 安装在 `/usr/local/jdk1.3.1`。要运行该应用程序，我可以使用：

```
export CLASSPATH=/usr/local/lib/finder.jar[1]:/usr/local/pgsql/share/
java/postgresql.jar:.
java Finder
```

1 `finder.jar` 包含 Finder 应用程序。

在应用程序内载入驱动的内容在第 8.2 节中介绍。

8.1.3. 为 JDBC 准备数据库

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。详情请参阅管理员指南。JDBC 驱动支持 trust、ident、password、md5 和 crypt 认证方法。

8.2. 使用驱动

8.2.1. 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.*;
```

重要

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。

8.2.2. 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。

8.2.3. 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`*

其中：

`host`

服务器的主机名。默认为 `localhost`。

`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);
```

8.2.4. 关闭连接

要关闭数据库连接，只需对 Connection 调用 close() 方法：

```
db.close();
```

8.3. 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。例 8.1演示了这个过程。

例 8.1. 在 JDCB 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();
```

8.3.1. 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 8.8 节，其中涵盖了一些要点。
- 用完 `Statement` 或 `PreparedStatement` 之后，你应该关闭它。

8.3.2. 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。
- `ResultSet` 目前是只读的。你无法通过 `ResultSet` 更新数据。如果你想更新数据，就需要用老办法，即发出一条 SQL 更新语句。这符合 JDBC 规范 —— 该规范并不要求驱动提供此功能。

8.4. 执行更新

要更改数据（执行插入、更新或删除），你使用 `executeUpdate()` 方法。`executeUpdate()` 类似于发出 select 所用的 `executeQuery()`，但它不返回 `ResultSet`，而是返回被该插入、更新或删除语句影响的记录数。

例 8.2. 简单删除示例

这个例子将发出一个简单的删除，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();
```

8.5. 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。`execute` 类似于发出 select 所用的 `executeQuery()`，但它不返回结果。

例 8.3. 删除表示例

这个例子将删除一个表。

```
Statement st = db.createStatement();
```

```
ResultSet rs = st.executeQuery("DROP TABLE mytable");
st.close();
```

8.6. 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 PostgreSQL 的二进制数据类型 `bytea` 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 1Gig 的二进制数据，但处理这样大的值需要大量的内存（RAM）。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 中，这些方法操作与大对象相关联的 `OID` 数据类型。通过把 `Connection` 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` API，或者使用 `getBLOB()` 和 `setBLOB()` 方法。

> **重要**
>
> 对于 PostgreSQL，你必须在一个 SQL 事务内访问大对象。你可以通过以 `false` 作为输入参数调用 `setAutoCommit()` 方法来打开一个事务。

> **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 8.4. 二进制数据示例

例如，假设你有一个表存储一幅图像的文件名，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 stream 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
conn.setAutoCommit(false);

// Get the Large Object Manager to perform operations with
LargeObjectManager lobj = ((org.postgresql.Connection)conn).getLarge
ObjectAPI();

//create a new large object
int oid = lobj.create(LargeObjectManager.READ | LargeObjectManager.
WRITE);

//open the large object for write
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
conn.setAutoCommit(false);

// Get the Large Object Manager to perform operations with
LargeObjectManager lobj = ((org.postgresql.Connection)conn).getLarge
ObjectAPI();

PreparedStatement ps = con.prepareStatement("SELECT imgOID FROM images
LO 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();
```

8.7. PostgreSQL 对 JDBC API 的扩展

PostgreSQL　是一个可扩展的数据库系统。你可以向后端添加自己的函数，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。

8.7.1. 访问扩展

要访问其中一些扩展，你需要使用 `org.postgresql.Connection` 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.Connection)db).getFastpathAPI();
```

8.7.1.1. `org.postgresql.Connection` 类

```
public class PGConnection
```

这些是用于访问 PostgreSQL 扩展的额外方法。`java.sql.Connection` 中定义的方法不在此列出。

8.7.1.1.1. 方法

- `public Fastpath getFastpathAPI() throws SQLException`

  这将返回当前连接的 Fastpath 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.Connection)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 后端上函数的 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.Connection)myconn).getLarge
  ObjectAPI();
  ```

  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.Connection)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。

8.7.1.2. 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后端上执行一个函数。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.Connection)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.Connection，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() 方法来执行一个后端函数。

参见：. FastpathFastpathArg、LargeObject

8.7.1.2.1. 方法

- ```
  public Object fastpath(int fnid,
                         boolean resulttype,
                         FastpathArg args[]) throws SQLException
  ```

  向 PostgreSQL 后端发送一次函数调用。

  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 fastpath 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 后端发送一次函数调用。

> **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 在后端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 fastpath 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。

8.7.1.3. org.postgresql.fastpath.FastpathArg 类

```
public class FastpathArg extends Object
```

```
java.lang.Object
   |
   +----org.postgresql.fastpath.FastpathArg
```

每次 fastpath 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

8.7.1.3.1. 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。

8.7.2. 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。后端目前尚未实现 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

8.7.3. 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。

8.7.3.1. 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

8.7.3.1.1. 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

8.7.3.1.2. 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

8.7.3.2. `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.Connection 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.Connection)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。

8.7.3.2.1. 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。

8.7.3.2.2. 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”。

8.8. 在多线程（multi-threaded）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，这对数据库引擎来说会是一件糟糕的事情。

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在使用数据库。

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。如果它是一条常规的 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 `Fastpath` 调用（例如从一个 `LargeObject` 读取一个块），那么这就是发送和取回该块的时间。

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。对 servlet 来说，连接上可能承受很重的负载。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，而这可能不是你想要的效果。

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`。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。一旦该线程用完了连接，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。

因此，对 servlet 来说，是用单个连接还是连接池由你决定。连接池的好处是线程不会受到单个网络连接造成的瓶颈的影响。缺点是它增加了服务器的负载，因为每个 `Connection` 都会创建一个后端进程。这取决于你和你的应用程序的需求。

8.9. 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），以及 JDBC 规范。两者都可以从 http://java.sun.com/products/jdbc/index.html 获得。

http://jdbc.postgresql.org 包含本文档未收录的最新信息，并且还提供预编译的驱动。

第 9 章 PyGreSQL - Python 接口

作者

由 D'Arcy J.M. Cain（<darcy@druid.net>）编写。大量基于 Pascal Andre <andre@chimay.via.ecp.fr> 编写的代码。

9.1. pg 模块

你可以选择使用 pg 模块提供的旧而成熟的接口，也可以使用符合 Python DB-SIG 开发的 DB-API 2.0[1] 规范的较新 pgdb 接口。

这里我们只描述较旧的 pg API。只要 PyGreSQL 还没有收录 DB-API 的描述，你就应该在 http://www.python.org/topics/database/DatabaseAPI-2.0.html 阅读该 API 的说明。

可以在 http://www2.linuxjournal.com/lj-issues/issue49/2605.html 找到一篇 DB-API 的教程式介绍

pg 模块定义了三个对象：

- pgobject，它处理连接以及对数据库的所有请求，
- pglargeobject，它处理对 PostgreSQL 大对象的所有访问，以及
- pgqueryobject，它处理查询结果。

如果你想看到使用其中一些函数的简单示例，请访问 http://www.druid.net/rides，在那里我在页面底部放了一个指向该页面实际 Python 代码的链接。

9.1.1. 常量

pg 模块字典中定义了一些常量。它们旨在用作方法调用的参数。关于它们的更多信息，请参阅 libpq 的描述（第 1 章）。这些常量是：

```
INV_READ
INV_WRITE
```

大对象的访问模式，由 (pgobject.)locreate 和 (pglarge.)open 使用。

```
SEEK_SET
SEEK_CUR
SEEK_END
```

位置标志，由 (pglarge.)seek 使用。

```
version
__version__
```

给出当前版本的常量

[1] http://www.python.org/topics/database/DatabaseAPI-2.0.html

9.2. pg 模块函数

`pg` 模块只定义了少量方法，用于连接数据库以及定义覆盖 PostgreSQL 所用环境变量的“默认变量”。

这些“默认变量”的设计意图是让你能够处理常规连接参数，而无需在程序中写大量代码。
你可以提示用户输入一个值，把它放入默认变量，然后就不用再管它，而不必修改你的环境。
可以在 Python `Setup` 文件中设置 -DNO_DEF_VAR 选项来禁用默认变量支持。与此相关的方法以标记 [DV] 标明。

在模块初始化时，所有变量都被设为 `None`，表示应使用标准环境变量。

connect

connect — 打开到数据库服务器的连接

大纲

```
connect([dbname], [host], [port], [opt], [tty], [user], [passwd])
```

参数

dbname

所连接数据库的名称（字符串/None）。

host

服务器主机名（字符串/None）。

port

数据库服务器使用的端口（整数/-1）。

opt

服务器选项（字符串/None）。

tty

用于后端可选调试输出的文件或 tty（字符串/None）。

user

PostgreSQL 用户（字符串/None）。

passwd

用户口令（字符串/None）。

返回类型

pgobject

成功时，返回一个处理数据库连接的对象。

异常

TypeError

参数类型错误，或参数过多。

SyntaxError

重复的参数定义。

pg.error

在定义 `pg` 连接期间发生了某个错误。

（加上与对象分配有关的所有异常）

描述

此方法打开到给定 PostgreSQL 服务器上指定数据库的连接。这里可以像 Python 教程中描述的那样使用关键字。关键字的名称就是语法行中给出的参数名。关于参数的精确描述，请参阅 PostgreSQL 用户手册。

示例

```
import pg

con1 = pg.connect('testdb', 'myhost', 5432, None, None, 'bob', None)
con2 = pg.connect(dbname='testdb', host='localhost', user='bob')
```

get_defhost

get_defhost — 获取默认主机名 [DV]

大纲

```
get_defhost()
```

参数

none

返回类型

string or None

默认主机说明

异常

SyntaxError

参数过多。

描述

`get_defhost()` 返回当前的默认主机说明，若应使用环境变量则返回 None。环境变量将不会被查找。

set_defhost

set_defhost — 设置默认主机名 [DV]

大纲

```
set_defhost(host)
```

参数

host

新的默认主机（字符串/None）。

返回类型

string or None

之前的默认主机说明。

异常

TypeError

参数类型错误，或参数过多。

描述

`set_defhost()` 为新连接设置默认主机值。如果以参数形式给出 None，则后续连接将使用环境变量。它返回之前的默认主机设置。

get_defport

get_defport — 获取默认端口 [DV]

大纲

```
get_defport()
```

参数

none

返回类型

integer or None

默认端口说明

异常

SyntaxError

参数过多。

描述

`get_defport()` 返回当前的默认端口说明，若应使用环境变量则返回 None。环境变量将不会被查找。

set_defport

set_defport — 设置默认端口 [DV]

大纲

```
set_defport(port)
```

参数

port

新的默认端口（整数/-1）。

返回类型

integer or None

之前的默认端口说明。

异常

TypeError

参数类型错误，或参数过多。

描述

`set_defport()` 为新连接设置默认端口值。如果以参数形式给出 -1，则后续连接将使用环境变量。它返回之前的默认端口设置。

get_defopt

get_defopt — 获取默认选项说明 [DV]

大纲

```
get_defopt()
```

参数

none

返回类型

string or None

默认选项说明

异常

SyntaxError

参数过多。

描述

`get_defopt()` 返回当前的默认连接选项说明，若应使用环境变量则返回 None。环境变量将不会被查找。

set_defopt

set_defopt — 设置默认选项说明 [DV]

大纲

```
set_defopt(options)
```

参数

options

新的默认连接选项（字符串/None）。

返回类型

string or None

之前的默认选项说明。

异常

TypeError

参数类型错误，或参数过多。

描述

`set_defopt()` 为新连接设置默认连接选项值。如果以参数形式给出 None，则后续连接将使用环境变量。它返回之前的默认连接选项设置。

get_deftty

get_deftty — 获取默认连接调试终端说明 [DV]

大纲

```
get_deftty()
```

参数

none

返回类型

string or None

默认调试终端说明

异常

SyntaxError

参数过多。

描述

`get_deftty()` 返回当前的默认调试终端说明，若应使用环境变量则返回 None。环境变量将不会被查找。

set_deftty

set_deftty — 设置默认连接调试终端说明 [DV]

大纲

```
set_deftty(terminal)
```

参数

terminal

新的默认调试终端（字符串/None）。

返回类型

string or None

之前的默认调试终端说明。

异常

TypeError

参数类型错误，或参数过多。

描述

`set_deftty()` 为新连接设置默认终端值。如果以参数形式给出 None，则后续连接将使用环境变量。它返回之前的默认终端设置。

get_defbase

get_defbase — 获取默认数据库名说明 [DV]

大纲

```
get_defbase()
```

参数

none

返回类型

string or None

默认调试数据库名说明

异常

SyntaxError

参数过多。

描述

`get_defbase()` 返回当前的默认数据库名说明，若应使用环境变量则返回 None。环境变量将不会被查找。

set_defbase

set_defbase — 设置默认数据库名说明 [DV]

大纲

```
set_defbase(database)
```

参数

database

新的默认数据库名（字符串/None）。

返回类型

string or None

之前的默认数据库名说明。

异常

TypeError

参数类型错误，或参数过多。

描述

`set_defbase()` 为新连接设置默认数据库名。如果以参数形式给出 None，则后续连接将使用环境变量。它返回之前的默认数据库名设置。

9.3. 连接对象：pgobject

此对象处理到 PostgreSQL 数据库的连接。它内嵌并隐藏了定义此连接的所有参数，只在函数调用中留下真正有意义的参数。

一些方法允许直接访问连接套接字。它们以标记 [DA] 标明。除非你真的知道自己在做什么，否则不要使用它们。如果你更愿意禁用它们，请在 Python `Setup` 文件中设置 `-DNO_DIRECT` 选项。

另一些方法允许访问大对象。如果想在模块中禁止这些访问，请在 Python Setup 文件中设置 `-DNO_LARGE` 选项。它们以标记 [LO] 标明。

每个 `pgobject` 定义了一组只读属性，描述连接及其状态。这些属性是：

host

服务器主机（字符串）

port

服务器的端口（整数）

db

选定的数据库（字符串）

options

连接选项（字符串）

tty

连接调试终端（字符串）

user

数据库系统上的用户名（字符串）

status

连接的状态（整数：1 - 正常，0 - 故障）

error

来自服务器的最近一条警告/错误消息（字符串）

query

query — 执行一条 SQL 命令

大纲

`query(`*`command`*`)`

参数

`command`

SQL 命令（字符串）。

返回类型

pgqueryobject or None

结果值。

异常

TypeError

参数类型错误，或参数过多。

ValueError

空的 SQL 查询。

pg.error

查询处理期间出错，或无效的连接。

描述

`query()` 方法向数据库发送一条 SQL 查询。如果查询是插入语句，返回值是新插入行的 OID。如果是无结果的其他查询（即不是某种 `SELECT` 语句），则返回 None。否则，它返回一个 `pgqueryobject`，可以通过 `getresult()` 或 `dictresult()` 方法访问，也可以直接打印。

reset

reset — 重置连接

大纲

```
reset()
```

参数

none

返回类型

none

异常

TypeError

（任意）参数过多。

描述

`reset()` 方法重置当前数据库。

close

close — 关闭数据库连接

大纲

```
close()
```

参数

none

返回类型

none

异常

TypeError

（任意）参数过多。

描述

`close()` 方法关闭数据库连接。连接被删除时无论如何都会被关闭，但此方法允许在需要时显式地关闭它。它存在主要是为了让 DB-SIG API 包装能够实现一个 close 函数。

fileno

fileno — 返回用于连接数据库的套接字

大纲

```
fileno()
```

参数

none

返回类型

socket id

用于连接数据库的底层套接字 id。

异常

TypeError

（任意）参数过多。

描述

`fileno()` 方法返回用于连接数据库的底层套接字 id。这在 `select` 调用等场合很有用。

getnotify

getnotify — 获取来自服务器的最近一次通知

大纲

```
getnotify()
```

参数

none

返回类型

tuple, None

来自服务器的最近一次通知

异常

TypeError

（任意）参数过多。

pg.error

无效的连接。

描述

`getnotify()` 方法尝试获取来自服务器的通知（来自 `SQL` 语句 `NOTIFY`）。如果服务器没有返回通知，该方法返回 None。否则，它返回一个元组（对）`(relname, pid)`，其中 `relname` 是通知的名称，`pid` 是触发该通知的连接的进程 id。记住先做一次 listen 查询，否则 getnotify 将始终返回 None。

inserttable

inserttable — 把一个列表插入表中

大纲

```
inserttable(table, values)
```

参数

table

表名（字符串）。

values

要插入的行值列表（列表）。

返回类型

none

异常

TypeError

参数类型错误，或（任意）参数过多。

pg.error

无效的连接。

描述

`inserttable()` 方法允许把大块数据快速插入表中：它把整个值列表插入给定的表。该列表是一个元组/列表的列表，每个元组/列表定义一行要插入的值。行的值可以包含字符串、整数、long 或 double（real）值。务必小心：此方法不会按照表定义对字段做类型检查；它只查看自己是否知道如何处理这些类型。

putline

putline — 向服务器套接字写一行 [DA]

大纲

```
putline(line)
```

参数

line

要写的行（字符串）。

返回类型

none

异常

TypeError

参数类型错误，或（任意）参数过多。

pg.error

无效的连接。

描述

`putline()` 方法允许直接向服务器套接字写一个字符串。

getline

getline — 从服务器套接字读一行 [DA]

大纲

```
getline()
```

参数

none

返回类型

string

读到的行。

异常

TypeError

参数类型错误，或（任意）参数过多。

pg.error

无效的连接。

描述

`getline()` 方法允许直接从服务器套接字读一个字符串。

endcopy

endcopy — 同步客户端和服务器 [DA]

大纲

```
endcopy()
```

参数

none

返回类型

none

异常

TypeError

参数类型错误，或（任意）参数过多。

pg.error

无效的连接。

描述

使用直接访问方法可能使客户端和服务器失去同步。
此方法确保命令结束时客户端和服务器是同步的。

locreate

locreate — 在数据库中创建一个大对象 [LO]

大纲

```
locreate(mode)
```

参数

mode

大对象创建模式。

返回类型

pglarge

处理 PostgreSQL 大对象的对象。

异常

TypeError

参数类型错误，或参数过多。

pg.error

无效的连接，或创建错误。

描述

`locreate()` 方法在数据库中创建一个大对象。模式可以通过把 pg 模块中定义的常量（`INV_READ and INV_WRITE`）按位 OR 来定义。

getlo

getlo — 根据给定的 OID 构建一个大对象 [LO]

大纲

`getlo(*oid*)`

参数

oid

现有大对象的 `oid`（整数）。

返回类型

pglarge

处理 PostgreSQL 大对象的对象。

异常

TypeError

参数类型错误，或参数过多。

pg.error

无效的连接。

描述

`getlo()` 方法允许通过 `pglarge` 接口复用先前创建的大对象，前提是你知道它的 `oid`。

loimport

loimport — 把文件导入为一个 PostgreSQL 大对象 [LO]

大纲

```
loimport(filename)
```

参数

filename

要导入的文件名（字符串）。

返回类型

pglarge

处理 PostgreSQL 大对象的对象。

异常

TypeError

参数类型错误，或参数过多。

pg.error

无效的连接，或文件导入期间出错。

描述

`loimport()` 方法允许以非常简单的方式创建大对象。你只需给出包含要导入数据的文件名，它就会在数据库中创建该对象。

9.4. 数据库包装类：DB

`pg` 模块包含一个名为 `DB` 的类。所有 `pgobject` 方法也都包含在这个类中。下面描述若干额外的 `DB` 类方法。使用本模块的推荐方式如下（见下文初始化方法的描述）：

```
import pg

db = pg.DB(...)

for r in db.query(
    "SELECT foo,bar
       FROM foo_bar_table
      WHERE foo !~ bar"
    ).dictresult():

    print '%(foo)s %(bar)s' % r
```

下面描述该类的方法和变量。

DB 类使用与 pg.connect 方法相同的参数初始化。它还初始化几个内部变量。语句 db = DB() 会像 pg.connect() 那样，以用户名打开本地数据库。

pkey

pkey — 返回一个表的主键

大纲

```
pkey(table)
```

参数

table

表名。

返回类型

string

作为表主键的字段名。

描述

`pkey()` 方法返回一个表的主键。注意，如果表没有主键，这会抛出异常。

get_databases

get_databases — 获取系统中数据库的列表

大纲

```
get_databases()
```

参数

none

返回类型

list

系统中数据库的列表。

描述

虽然你可以用一个简单的 select 做到这一点，但为了方便起见还是把它加在了这里

get_tables

get_tables — 获取所连接数据库中表的列表

大纲

```
get_tables()
```

参数

none

返回类型

list

所连接数据库中表的列表。

描述

虽然你可以用一个简单的 select 做到这一点，但为了方便起见还是把它加在了这里

get_attnames

get_attnames — 返回一个表的属性名

大纲

```
get_attnames(table)
```

参数

table

表名。

返回类型

dictionary

字典的键是属性名，值是属性的类型名。

描述

给定一个表的名称，挖出它的属性名和类型集合。

get

get — 从数据库表中获取一个元组

大纲

```
get(table, arg, [keyname])
```

参数

table

表名。

arg

一个字典，或者要查找的值。

[*keyname*]

用作键的字段名（可选）。

返回类型

dictionary

一个把属性名映射到行值的字典。

描述

此方法是获取单个行的基本机制。它假定键指定了唯一的一行。如果没有给出 keyname，则使用该表的主键。如果 arg 是一个字典，就从它取键的值，并且它会被修改为包含新的值，必要的地方替换现有的值。oid 也会被放入字典中，但为了让调用者能够处理多个表，属性名经过了改写以使其唯一。它由字符串 `oid_` 后跟表名组成。

insert

insert — 把一个元组插入数据库表

大纲

```
insert(table, a)
```

参数

table

表名。

a

一个值的字典。

返回类型

integer

新插入行的 OID。

描述

此方法把值插入指定的表，用字典中的值填充。然后带着数据库中的值把字典重新载入。这会使字典被规则、触发器等修改过的值更新。

update

update — 更新数据库表

大纲

```
update(table, a)
```

参数

table

表名。

a

一个值的字典。

返回类型

integer

新更新行的 OID。

描述

类似于 insert，但更新现有的行。更新基于 get 处理后的 OID 值。返回的数组就是被修改并发回数据库的那个。

clear

clear — 清空数据库表

大纲

```
clear(table, [a])
```

参数

table

表名。

[*a*]

一个值的字典。

返回类型

dictionary

一个带空行的字典。

描述

此方法把所有属性清除为由类型决定的值。数值类型被设为 0，日期被设为 `'today'`，其他一切都设为空字符串。如果给出 array 参数，就把它用作数组，与属性名匹配的所有条目被清除，其余内容保持不变。

delete

delete — 从表中删除一行

大纲

```
delete(table, [a])
```

参数

table

表名。

[*a*]

一个值的字典。

返回类型

none

描述

此方法从表中删除行。它基于上文所述处理后的 OID 进行删除。

9.5. 查询结果对象：pgqueryobject

getresult

getresult — 获取查询返回的值

大纲

```
getresult()
```

参数

none

返回类型

list

元组的列表。

异常

SyntaxError

参数过多。

pg.error

无效的前一个结果。

描述

`getresult()` 方法返回查询返回的值的列表。关于这个结果的更多信息可以用 `listfields`、`fieldname` 和 `fieldnum` 方法访问。

dictresult

dictresult — 以字典列表的形式获取查询返回的值

大纲

```
dictresult()
```

参数

none

返回类型

list

字典的列表。

异常

SyntaxError

参数过多。

pg.error

无效的前一个结果。

描述

`dictresult()` 方法以字典列表的形式返回查询返回的值，每个元组都是一个以字段名作为字典索引的字典。

listfields

listfields — 列出查询结果的字段名

大纲

```
listfields()
```

参数

none

返回类型

list

字段名

异常

SyntaxError

参数过多。

pg.error

无效的查询结果，或无效的连接。

描述

`listfields()` 方法返回为查询结果定义的字段名列表。这些字段与结果值的顺序相同。

fieldname

fieldname — 按编号获取字段名

大纲

```
fieldname(i)
```

参数

i

字段编号（整数）。

返回类型

string

字段名。

异常

TypeError

参数类型错误，或参数过多。

ValueError

无效的字段编号。

pg.error

无效的查询结果，或无效的连接。

描述

`fieldname()` 方法允许根据序号查找字段名。它在显示结果时可能很有用。这些字段与结果值的顺序相同。

fieldnum

fieldnum — 按名称获取字段编号

大纲

```
fieldnum(name)
```

参数

name

字段名（字符串）。

返回类型

integer

字段编号（整数）。

异常

TypeError

参数类型错误，或参数过多。

ValueError

未知的字段名。

pg.error

无效的查询结果，或无效的连接。

描述

`fieldnum()` 方法根据名称返回字段编号。它可以用来构建一个把结果列表字符串转换为正确类型的函数，使用一张硬编码的表定义。返回的编号是该字段在结果值列表中的序号。

ntuples

ntuples — 返回查询对象中的元组数

大纲

```
ntuples()
```

参数

none

返回类型

integer

查询对象中的元组数。

异常

SyntaxError

参数过多。

描述

`ntuples()` 方法返回查询中找到的元组数。

9.6. 大对象：pglarge

此对象处理与 PostgreSQL 大对象有关的所有请求。它内嵌并隐藏了所有“重复”变量（对象 OID 和连接），方式与 `pgobject` 所做的完全相同，因此只在函数调用中保留有意义的参数。它保留一个对创建它时所用的 `pgobject` 的引用，通过其参数发送请求。除了取消引用之外，对 `pgobject` 的任何修改都会影响 `pglarge` 对象。对初始 `pgobject` 取消引用不是问题，因为在大对象取消对其引用之前，Python 不会释放它。所有函数在调用出错时都返回一条通用的错误消息，无论实际错误是什么。对象的 `error` 属性允许获取确切的错误消息。

`pglarge` 对象定义了一组只读属性，允许获取关于它的一些信息。这些属性是：

oid

与该对象关联的 oid

pgcnx

用于创建该对象的 `pgobject`

error

连接的最近一条警告/错误消息

> **小心**
>
> 在多线程环境中，`error` 可能被另一个使用同一 `pgobject` 的线程修改。请记住，这些对象是共享的，而不是复制的；在这种情况下如果要检查错误消息，你应当提供某种锁。

OID 属性非常有用，因为它允许你之后重用该 OID，用一个 `pgobject getlo()` 方法调用创建 `pglarge` 对象。

关于 PostgreSQL 大对象接口的更多信息，另见第 2 章。

open

open — 打开大对象

大纲

`open(`*`mode`*`)`

参数

`mode`

打开模式定义（整数）。

返回类型

none

异常

TypeError

参数类型错误，或参数过多。

IOError

对象已打开，或打开错误。

pg.error

无效的连接。

描述

`open()` 方法打开一个大对象用于读/写，方式与 UNIX `open()` 函数相同。模式值可以通过把 pg 模块中定义的常量（`INV_READ`, `INV_WRITE`）按位 OR 来获得。

close

close — 关闭大对象

大纲

```
close()
```

参数

none

返回类型

none

异常

SyntaxError

参数过多。

IOError

对象未打开，或关闭错误。

pg.error

无效的连接。

描述

`close()` 方法关闭先前打开的大对象，方式与 UNIX `close()` 函数相同。

read

read — 从大对象读取

大纲

`read(*size*)`

参数

size

要读取的缓冲区的最大大小（整数）。

返回类型

string

读到的缓冲区。

异常

TypeError

参数类型错误，或参数过多。

IOError

对象未打开，或读取错误。

pg.error

无效的连接或无效的对象。

描述

`read()` 方法允许从大对象的当前位置开始读取数据。

write

write — 写入大对象

大纲

```
write(string)
```

参数

string

要写入的缓冲区（字符串）。

返回类型

none

异常

TypeError

参数类型错误，或参数过多。

IOError

对象未打开，或写入错误。

pg.error

无效的连接或无效的对象。

描述

`write()` 方法允许向大对象的当前位置开始写入数据。

seek

seek — 改变大对象中的当前位置

大纲

```
seek(offset, whence)
```

参数

offset

位置偏移（整数）。

whence

位置参数（整数）。

返回类型

integer

对象中新的当前位置。

异常

TypeError

参数类型错误，或参数过多。

IOError

对象未打开，或定位错误。

pg.error

无效的连接或无效的对象。

描述

`seek()` 方法允许移动大对象中的游标位置。whence 参数可以通过把 `pg` 模块中定义的常量（`SEEK_SET`, `SEEK_CUR`, `SEEK_END`）按位 OR 来获得。

tell

tell — 返回大对象中的当前位置

大纲

```
tell()
```

参数

none

返回类型

integer

对象中的当前位置。

异常

SyntaxError

参数过多。

IOError

对象未打开，或定位错误。

pg.error

无效的连接或无效的对象。

描述

`tell()` 方法允许获取大对象中的当前位置。

unlink

unlink — 删除大对象

大纲

```
unlink()
```

参数

none

返回类型

none

异常

SyntaxError

参数过多。

IOError

对象未关闭，或删除错误。

pg.error

无效的连接或无效的对象。

描述

`unlink()` 方法删除（unlink）大对象。

size

size — 返回大对象的大小

大纲

```
size()
```

参数

none

返回类型

integer

大对象的大小。

异常

SyntaxError

参数过多。

IOError

对象未打开，或定位/查询位置错误。

pg.error

无效的连接或无效的对象。

描述

`size()` 方法允许获取大对象的大小。之所以实现它，是因为这个函数对带 WWW 接口的数据库非常有用。目前，必须先打开大对象。

export

export — 把大对象保存到文件

大纲

```
export(filename)
```

参数

filename

要创建的文件。

返回类型

none

异常

TypeError

参数类型错误，或参数过多。

IOError

对象未关闭，或导出错误。

pg.error

无效的连接或无效的对象。

描述

`export()` 方法允许以非常简单的方式转储大对象的内容。导出的文件在程序所在的主机上创建，而不是在服务器主机上创建。

9.7. DB-API 接口

参阅 http://www.python.org/topics/database/DatabaseAPI-2.0.html 中对 DB-API 2.0 的描述。

部分 II. 服务器编程

手册的这一第二部分解释　PostgreSQL　的可扩展性方法，并描述用户如何通过添加用户定义的类型、操作符、聚集以及查询语言和编程语言函数来扩展 PostgreSQL。在讨论 PostgreSQL 规则系统之后，我们讨论触发器和 SPI 接口。

目录

10. 体系结构 171
    10.1. PostgreSQL 体系结构概念 171
11. 扩展 SQL：概览 174
    11.1. 可扩展性如何运作 174
    11.2. PostgreSQL 类型系统 174
    11.3. 关于 PostgreSQL 系统目录 174
12. 扩展 SQL：函数 177
    12.1. 简介 177
    12.2. 查询语言（SQL）函数 177
        12.2.1. 示例 177
        12.2.2. 基本类型上的 SQL 函数 178
        12.2.3. 复合类型上的 SQL 函数 179
        12.2.4. 返回集合的 SQL 函数 180
    12.3. 过程语言函数 181
    12.4. 内部函数 181
    12.5. C 语言函数 181
        12.5.1. 动态装载 182
        12.5.2. C 语言函数中的基础类型 183
        12.5.3. C 语言函数的版本 0 调用约定 185
        12.5.4. C 语言函数的版本 1 调用约定 187
        12.5.5. C 语言函数中的复合类型 189
        12.5.6. 编写代码 190
        12.5.7. 编译和链接动态装载的函数 191
    12.6. 函数重载 193
    12.7. 过程语言处理器 194
13. 扩展 SQL：类型 197
14. 扩展 SQL：操作符 199
    14.1. 简介 199
    14.2. 示例 199
    14.3. 操作符优化信息 199
        14.3.1. COMMUTATOR 200
        14.3.2. NEGATOR 200
        14.3.3. RESTRICT 201
        14.3.4. JOIN 201
        14.3.5. HASHES 202
        14.3.6. SORT1 and SORT2 202
15. 扩展 SQL：聚合 204
16. 规则系统 206
    16.1. 简介 206
    16.2. 什么是查询树？ 206
        16.2.1. 查询树的组成部分 206
    16.3. 视图和规则系统 208
        16.3.1. PostgreSQL 中视图的实现 208
        16.3.2. SELECT 规则如何工作 208
        16.3.3. 非 SELECT 语句中的视图规则 213
        16.3.4. PostgreSQL 中视图的能力 214
        16.3.5. 那么更新视图会怎么样？ 214
    16.4. INSERT、UPDATE 和 DELETE 上的规则 215
        16.4.1. 与视图规则的区别 215
        16.4.2. 这些规则如何工作 215
        16.4.3. 与视图的协作 219

16.5. 规则和权限 225
16.6. 规则与触发器 225
17. 索引扩展接口 228
17.1. 简介 228
17.2. 访问方法 228
17.3. 访问方法策略 229
17.4. 访问方法支持例程 229
17.5. 操作符类 230
17.6. 创建操作符和支持例程 230
18. 索引代价估计函数 234
19. GiST 索引 237
20. 触发器 239
20.1. 触发器创建 239
20.2. 与触发器管理器的交互 240
20.3. 数据更改的可见性 242
20.4. 示例 242
21. 服务器编程接口 246
21.1. 接口函数 246
21.2. 接口支持函数 259
21.3. 内存管理 267
21.4. 数据更改的可见性 279
21.5. 示例 279

第 10 章 体系结构

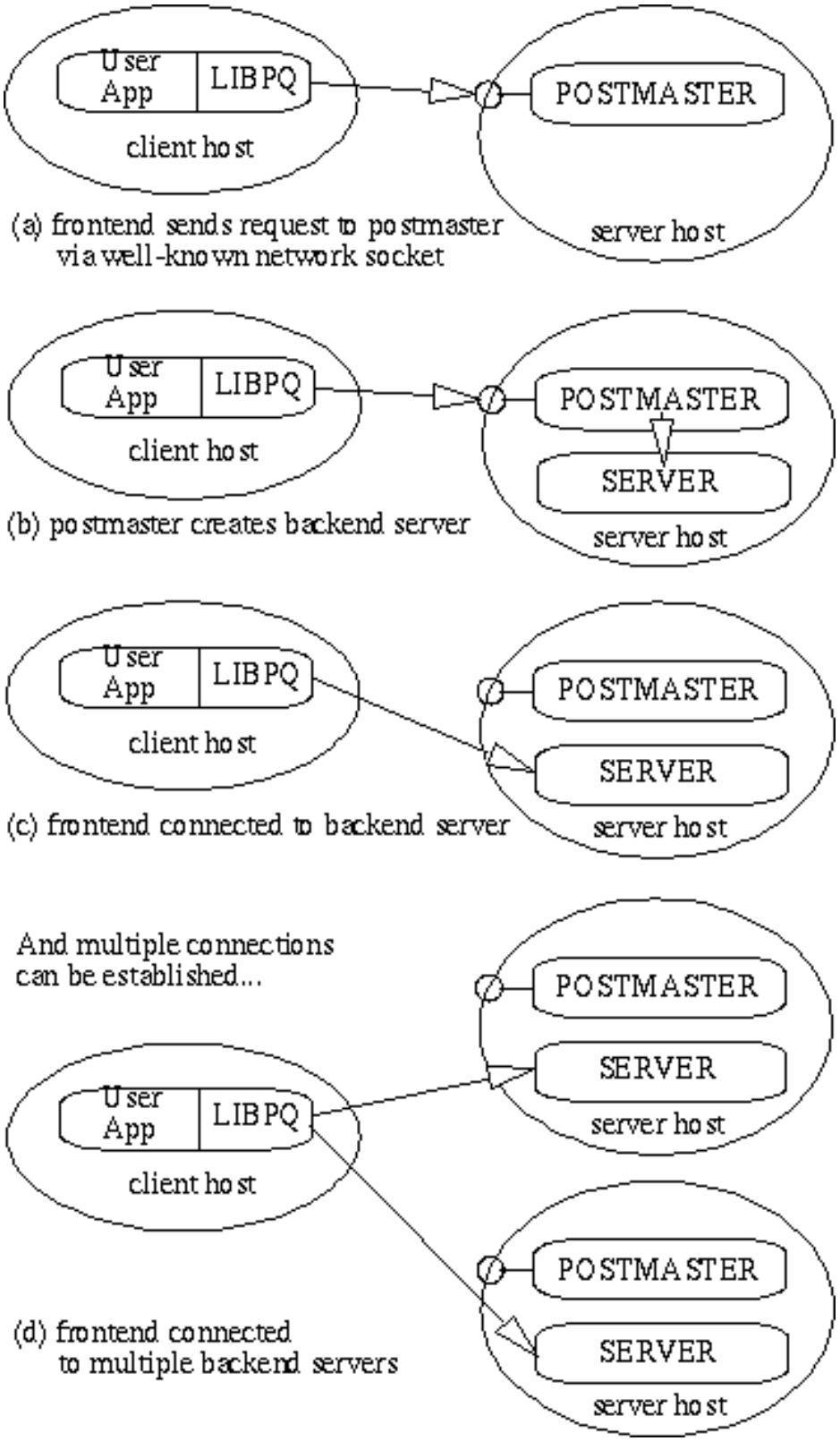
10.1. PostgreSQL 体系结构概念

在开始之前，你应当理解 PostgreSQL 系统的基本体系结构。理解 PostgreSQL 各部分的交互方式会让下一章更容易理解。用数据库术语来说，PostgreSQL 使用简单的"每用户一进程"客户端/服务器模型。一个 PostgreSQL 会话由下列协作的 Unix 进程（程序）组成：

- 一个管理守护进程（postmaster），
- 用户的前端应用（例如 psql 程序），以及
- 一个或多个后端数据库服务器（postgres 进程本身）。

单个 postmaster 管理单个主机上给定的数据库集合。这样的数据库集合称为（数据库）集群。希望访问集群中某个数据库的前端应用调用链接进该应用的接口库（例如 libpq）。该库通过网络把用户请求发送给 postmaster（图 10.1(a)），后者再启动一个新的后端服务器进程（图 10.1(b)）

图 10.1. 如何建立连接


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
SERVER
(b) postmaster creates backend server
(c) frontend connected to backend server
And multiple connections can be established...
(d) frontend connected to multiple backend servers

并把前端进程连接到新服务器（图 10.1(c)）。从那时起，前端进程和后端服务器之间的通信不再需要 postmaster 干预。因此，postmaster 总是在运行并等待连接请求，而前端和后端进程则来来去去。`libpq` 库允许单个前端对后端进程建立多个连接。但是，每个后端进程都是单线程进程，一次只能执行一个查询；因此任何一条前端到后端连接上的通信都是单线程的。

这一体系结构的一个含义是 postmaster 和后端总是运行在同一台机器（数据库服务器）上，而前端应用可以运行在任何地方。你应当牢记这一点，
因为在客户端机器上可以访问的文件在数据库服务器机器上可能无法访问（或者只能用不同的路径名访问）。

你还应当知道 postmaster 和 postgres 服务器以 PostgreSQL “超级用户”的用户 ID 运行。注意 PostgreSQL 超级用户不必是任何特定的用户（例如名为 `postgres` 的用户），虽然许多系统就是这样安装的。而且，PostgreSQL 超级用户绝对不应该是 Unix 超级用户 `root`！就周围的 Unix 系统而言，PostgreSQL 超级用户是一个普通的、无特权的用户是最安全的。无论如何，与数据库相关的所有文件都应属于这个 Postgres 超级用户。

第 11 章 扩展 SQL：概览

在后面的各节中，我们将讨论如何通过添加下列内容来扩展 PostgreSQL 的 SQL 查询语言：

- 函数
- 数据类型
- 操作符
- 聚集

11.1. 可扩展性如何运作

PostgreSQL 之所以可扩展，是因为它的操作是目录驱动的。如果你熟悉标准的关系系统，就知道它们把关于数据库、表、列等的信息存储在通常所谓的系统目录中。
（有些系统称之为数据字典。）这些目录在用户看来与其他表一样，但 DBMS 把它的内部簿记存储在其中。PostgreSQL 与标准关系系统之间的一个关键区别是，PostgreSQL 在其目录中存储的信息多得多——不仅有关于表和列的信息，还有关于其类型、函数、访问方法等的信息。
用户可以修改这些表，而由于 PostgreSQL 把它的内部操作建立在这些表之上，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更改 DBMS 内部的硬编码过程或装载由 DBMS 厂商专门编写的模块来扩展。

PostgreSQL 与大多数其他数据管理器的另一个不同之处在于，服务器可以通过动态装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新类型或函数的目标代码文件（例如共享库），PostgreSQL 会按需装载它。用 SQL 编写的代码加入服务器更是轻而易举。这种“即时”修改操作的能力使 PostgreSQL 特别适合新应用和存储结构的快速原型开发。

11.2. PostgreSQL 类型系统

PostgreSQL 的类型系统可以从几个方面细分。类型分为基本类型和复合类型。基本类型是像 `int4` 这样用 C 之类的语言实现的类型。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方法操作这类类型，并且只在用户描述它们的程度上理解这类类型的行为。复合类型在用户创建表时创建。

PostgreSQL 只以一种方式存储这些类型（在存储表的所有行的文件内），但用户可以从查询语言“窥视”这些类型的属性，并通过（例如）在属性上定义索引来优化它们的检索。PostgreSQL 的基本类型进一步分为内置类型和用户定义类型。内置类型（如 `int4`）是编译进系统的那些。用户定义类型是用户以后面要描述的方式创建的类型。

11.3. 关于 PostgreSQL 系统目录

介绍了基本的可扩展性概念之后，我们现在可以看看目录实际上是如何组织的。
现在可以跳过本节，但后面的某些章节如果没有这里给出的信息将无法理解，
所以请给这一页做个标记以便以后参考。所有系统目录的名称都以 `pg_` 开头。下表包含可能对最终用户有用的信息。（还有很多其他系统目录，但很少需要直接查询它们。）

表 11.1. PostgreSQL 系统目录

目录名	描述
pg_database	数据库
pg_class	表
pg_attribute	表列

目录名	描述
pg_index	索引
pg_proc	过程/函数
pg_type	数据类型（基本和复合）
pg_operator	操作符
pg_aggregate	聚集函数
pg_am	访问方法
pg_amop	访问方法操作符
pg_amproc	访问方法支持函数
pg_opclass	访问方法操作符类

图 11.1. PostgreSQL 的主要系统目录

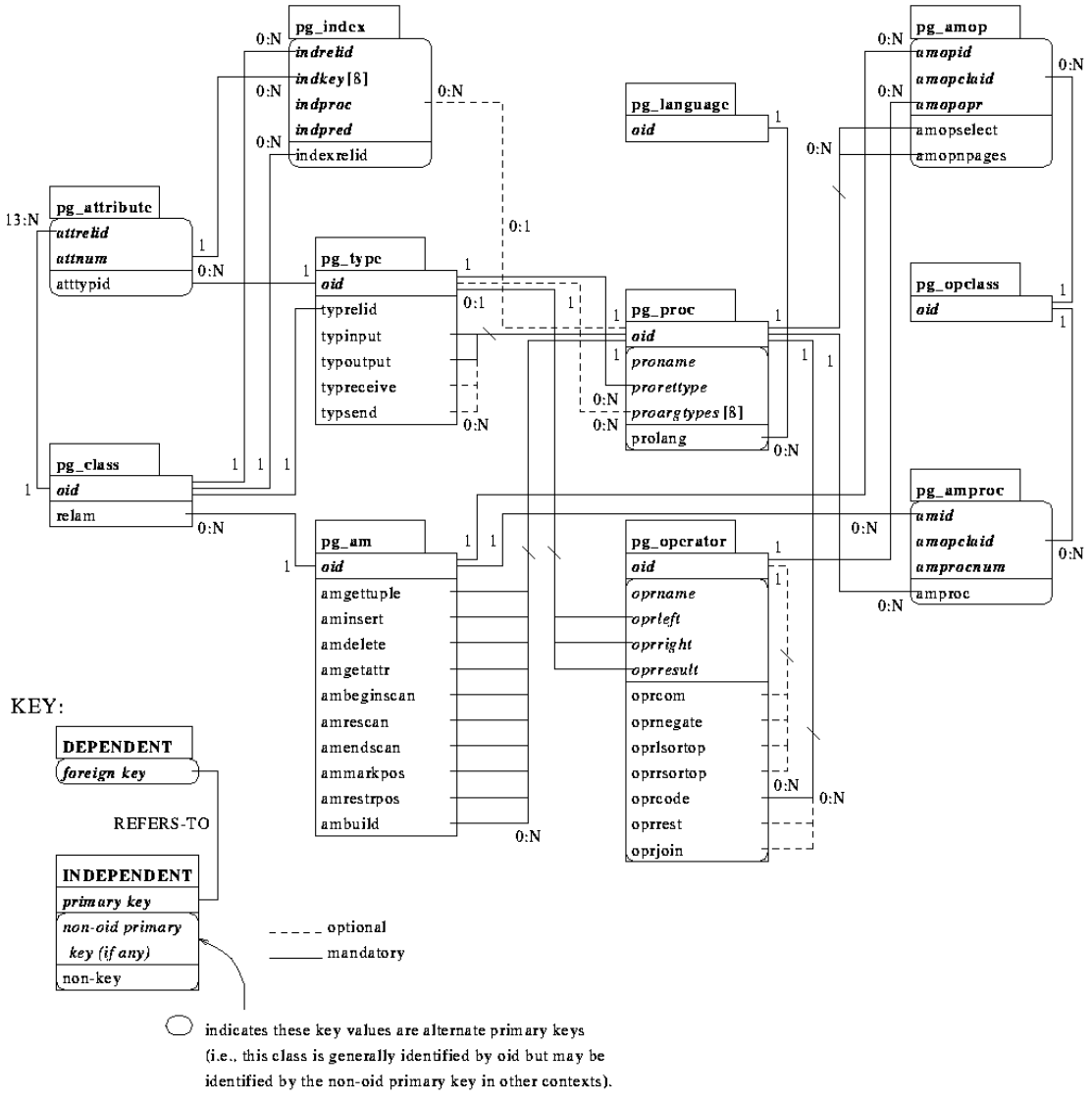


Figure 3. The major POSTGRES system catalogs.

The Developer's Guide gives a more detailed explanation of these catalogs and their columns. However, 图 11.1 shows the major entities and their relationships in the system catalogs. (Columns that do not refer to other entities are not shown unless they are part of a primary key.) This diagram is more or less incomprehensible until you actually start looking at the contents of the catalogs and see how they relate to each other. For now, the main things to take away from this diagram are as follows:

- 在后面的几节中，我们将给出系统目录上的各种连接查询，
  它们显示我们扩展系统所需的信息。
  看这张图应该会让其中一些连接查询（它们通常是三路或四路连接）更容易理解，
  因为你将能看到查询中使用的列在其他表中构成外键。
- 许多不同的特性（表、列、函数、类型、访问方法等）在这个模式中紧密集成。一条简单的 create 命令就可能修改其中许多目录。
- 类型和过程是这个模式的核心。

  注意

  我们或多或少可以互换地使用过程和函数这两个词。

  几乎每个目录都包含对这两个表中之一或两者的行的引用。例如，PostgreSQL　经常使用类型签名（例如函数和操作符的签名）来标识其他目录的唯一行。
- 有许多列和关系的含义是显而易见的，但也有许多（特别是那些与访问方法有关的）
  并非如此。`pg_am`、`pg_amop`、`pg_amproc`、`pg_operator` 和 `pg_opclass` 之间的关系特别难以理解，我们将在讨论了基本扩展之后（在第 17 章中）深入描述它们。

第 12 章 扩展 SQL：函数

12.1. 简介

事实证明，定义新类型的一部分工作就是定义描述其行为的函数。因此，
虽然可以不定义新类型就定义新函数，反过来却不成立。所以我们在描述如何向 PostgreSQL 添加新类型之前，先描述如何添加新函数。

PostgreSQL提供四种函数：

- 查询语言函数（用 SQL 编写的函数）
- 过程语言函数（例如用 PL/Tcl 或 PL/pgSQL 编写的函数）
- 内部函数
- C 语言函数

每种函数都可以接受基本类型、复合类型或它们的某种组合作为参数。此外，
每种函数都可以返回基本类型或复合类型。定义 SQL 函数最容易，所以我们从它开始。本节的示例也可以在教程目录的 `funcs.sql` 和 `funcs.c` 中找到。

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。

12.2. 查询语言（SQL）函数

SQL 函数执行一个任意的 SQL 语句列表，返回列表中最后一个查询的结果，该查询必须是一个 `SELECT`。在简单（非集合）的情况下，将返回最后一个查询结果的第一行。（请记住，除非使用 `ORDER BY`，多行结果的“第一行”并没有良好定义。）如果最后一个查询碰巧不返回任何行，则返回 NULL。

或者，也可以把 SQL 函数声明为返回集合，方法是把函数的返回类型指定为 `SETOF` *`sometype`*。此时将返回最后一个查询结果的所有行。更多细节见下文。

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使用语法 `$`*`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.emp`）访问该参数的属性。

12.2.1. 示例

为说明一个简单的 SQL 函数，考虑下面这个可以用来给银行账户扣款的例子：

```
CREATE FUNCTION tp1 (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 tp1(17, 100.0);
```

实践中人们可能希望函数返回比常量“1”更有用的结果，所以更可能的定义是

```
CREATE FUNCTION tp1 (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。这些命令既可以是数据修改（即 INSERT、UPDATE 和 DELETE），也可以是 SELECT 查询。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。

```
CREATE FUNCTION clean_EMP () RETURNS integer AS '
    DELETE FROM EMP
        WHERE EMP.salary <= 0;
    SELECT 1 AS ignore_this;
' LANGUAGE SQL;

SELECT clean_EMP();

 x
---
 1
```

12.2.2. 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
```

注意我们在函数体内为函数结果定义了一个列别名（名为 RESULT），但这个列别名在函数之外不可见。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
```

12.2.3. 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` 计算你的薪资翻倍后的值：

```
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。

也可以构建返回复合类型的函数。（不过，如下文所见，
对这种函数的使用存在一些不幸的限制。）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或表达式。
定义该函数时有两点重要注意事项：

- 查询中的目标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。
- 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
```

在 PostgreSQL 的当前版本中，对返回复合类型的函数的使用存在一些不便的限制。简而言之，调用一个返回行的函数时，我们无法检索整个行。我们必须要么从行中投影出单个属性，
要么把整行传递给另一个函数。（试图显示整个行值只会得到一个无意义的数字。）例如，

```
SELECT name(new_emp());

 name
------
 None
```

这个示例使用了投影属性的函数记法。简单的解释是：`attribute(table)` 和 `table.attribute` 这两种记法通常可以互换使用：

```
--
-- 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
```

一般而言，我们必须使用函数语法投影函数返回值的属性，
原因就是解析器还不理解与函数调用组合使用点记法做投影。

```
SELECT new_emp().name AS nobody;
ERROR:  parser: parse error at or near "."
```

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)
```

12.2.4. 返回集合的 SQL 函数

如前所述，SQL 函数可以声明为返回 `SETOF` *`sometype`*。此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作为结果集的一个元素返回。

返回集合的函数只能在 `SELECT` 查询的目标列表中调用。对 `SELECT` 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。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` 对这些输入返回空集，因此不会生成输出行。

12.3. 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。其他语言可以由用户定义。更多信息参阅第 22 章。开发新过程语言的基础见第 12.7 节。

12.4. 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，服务器后端中出现的所有内部函数都在数据库集簇初始化（`initdb`）期间声明，
但用户可以用 `CREATE FUNCTION` 为内部函数创建额外的别名。内部函数在 `CREATE FUNCTION` 中用语言名 `internal` 声明。例如，为 `sqrt` 函数创建一个别名：

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE INTERNAL
    WITH (isStrict);
```

（大多数内部函数都期望被声明为“严格”的。）

> **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。

12.5. 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写一个 PG_FUNCTION_INFO_V1() 宏调用来表明，如下所示。没有这样的宏则表示是旧风格（“版本 0”）函数。无论哪种情况，CREATE FUNCTION 中指定的语言名都是 C。旧风格函数由于可移植性问题和功能欠缺现已弃用，但出于兼容性原因仍受支持。

12.5.1. 动态装载

在一个后端会话中，某个可装载对象文件里的用户自定义函数第一次被调用时，
动态装载器会把该对象文件装载到内存中，以便能够调用该函数。因此，用户自定义 C 函数的 CREATE FUNCTION 必须为函数指定两项信息：可装载对象文件的名称，以及该对象文件内要调用的特定函数的 C 名称（链接符号）。如果没有显式指定 C 名称，则假定它与 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。

> **注意**
>
> 运行 PostgreSQL 服务器的用户 ID 必须能够遍历到你要装载的文件的路径。把文件或更高层目录变得对“postgres”用户不可读和/或不可执行是一个常见错误。

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。对象文件必须在被 CREATE FUNCTION 命令引用之前编译好。更多信息见第 12.5.7 节。

> **注意**
>
> 动态装载的对象文件在第一次使用后会保留在内存中。
> 同一会话中以后对该文件中函数的调用只会产生符号表查找这一很小的开销。
> 如果你需要强制重新装载一个对象文件（例如重新编译之后），可以使用 LOAD 命令或开始一个新会话。

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。

> **注意**
>
> 在 PostgreSQL 7.2 版之前，CREATE FUNCTION 中只能指定对象文件的精确绝对路径。这种做法现已弃用，因为它使函数定义产生不必要的不可移植性。
> 最好只指定不带路径和扩展名的共享库名，让搜索机制去提供那些信息。

12.5.2. C 语言函数中的基础类型

表 12.1给出了将要装载到 PostgreSQL 中的 C 函数的参数所需的 C 类型。“定义于”列给出为获得类型定义而需要包含的头文件。（实际定义可能在所列文件包含的另一个文件中。
建议用户坚持使用已定义的接口。）注意，你应该始终包含 postgres.h 放在任何源文件的最前面，因为它声明了许多你反正需要的东西。

表 12.1. 内置 PostgreSQL 类型的等价 C 类型

SQL 类型	C 类型	定义于
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

SQL 类型	C 类型	定义于
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

在内部，PostgreSQL 把基本类型视为一个“内存块”。你针对某个类型定义的用户自定义函数转而定义了 PostgreSQL 能对它进行操作的方式。也就是说，PostgreSQL 只负责在磁盘上存储和检索数据，而使用你的用户自定义函数来输入、处理和输出数据。
基本类型可以有以下三种内部格式之一：

- 传值，定长
- 传引用，定长
- 传引用，变长

传值类型的长度只能是 1、2 或 4 字节（如果你的机器上 `sizeof(Datum)` 为 8，也可以是 8 字节）。你应当小心地定义类型，使其在所有体系结构上具有相同的大小（以字节计）。例如，`long` 类型是危险的，因为它在某些机器上是 4 字节而在另一些机器上是 8 字节；而 `int` 类型在大多数 Unix 机器上是 4 字节。Unix 上 `int4` 类型的合理实现可以是：

```
/* 4-byte integer, passed by value */
typedef int int4;
```

PostgreSQL 会自动把事情安排好，使整数类型真正具有其宣称的大小。

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-byte structure, passed by reference */
typedef struct
{
    double  x, y;
} Point;
```

在 PostgreSQL 函数中传入和传出这类类型时只能使用指向它们的指针。要返回这种类型的值，用 `palloc()` 分配适量内存，填充所分配的内存，并返回指向它的指针。（或者，你也可以通过返回同类型输入值的指针来返回该输入值。但绝不要修改传引用输入值的内容。）

最后，所有变长类型也必须以传引用方式传递。所有变长类型都必须以一个恰好 4 字节的长度字段开始，而该类型中要存储的所有数据都位于紧随该长度字段之后的内存中。
长度字段是结构的总长度（即它包含长度字段本身的大小）。我们可以这样定义 text 类型：

```
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 中无法声明变大小的结构，我们依赖这样一个知识：C 编译器不会对数组下标做范围检查。我们只分配所需数量的空间，然

后就像数组声明了正确长度一样去访问它。（如果你对这个技巧不熟悉，
可能需要先花些时间读一本 C 编程入门教科书，再深入研究 PostgreSQL 服务器编程。）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 text 结构中存储 40 字节，可以使用如下代码片段：

```
#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。

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。

12.5.3. C 语言函数的版本 0 调用约定

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。

下面是一些示例：

```
#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(int4) RETURNS int4
     AS 'PGROOT/tutorial/funcs' LANGUAGE C
     WITH (isStrict);

-- note overloading of SQL function name add_one()
CREATE FUNCTION add_one(float8) RETURNS float8
     AS 'PGROOT/tutorial/funcs',
        'add_one_float8'
     LANGUAGE C WITH (isStrict);

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'PGROOT/tutorial/funcs' LANGUAGE C
```

```
     WITH (isStrict);

CREATE FUNCTION copytext(text) RETURNS text
     AS 'PGROOT/tutorial/funcs' LANGUAGE C
     WITH (isStrict);

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'PGROOT/tutorial/funcs' LANGUAGE C
     WITH (isStrict);
```

这里 *PGROOT* 表示 PostgreSQL 源码树的全路径。（更好的风格是在把 *PGROOT*/tutorial 加入搜索路径之后，在 AS 子句中只写 'funcs'。无论哪种情况，我们都可以省略共享库的系统特定扩展名，通常是 .so 或 .sl。）

注意我们把这些函数指定为“严格”的，意思是如果任何输入值为 NULL，系统就应自动假定结果为 NULL。这样做就避免了在函数代码中检查 NULL 输入。不这样做的话，我们就得显式检查 NULL，例如为每个传引用参数检查空指针。（对传值参数，我们甚至没有办法检查！）

虽然这个调用约定使用简单，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 的数据类型存在问题。此外，也没有返回 NULL 结果的简单方法，除了把函数声明为严格之外也无法以任何方式处理 NULL 参数。接下来介绍的版本 1 约定克服了这些缺点。

12.5.4. C 语言函数的版本 1 调用约定

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（按惯例写在函数本身之前）。对 internal 语言的函数不需要此宏调用，因为 PostgreSQL 目前假定所有内部函数都是版本 1。但对动态装载的函数它是必需的。

在版本 1 函数中，每个实际参数用与该参数数据类型对应的 PG_GETARG_*xxx*() 宏取得，结果用与返回类型对应的 PG_RETURN_*xxx*() 宏返回。

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);
}
```

```
/* By 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。

乍一看，版本 1 编码约定可能显得像无谓的故弄玄虚。但它们确实提供了不少改进，因为宏可以隐藏不必要的细节。例如，在编写 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 宏隐藏了处理获取“经 TOAST 处理”（压缩或行外存储）值的需要。上面所示的旧风格 copytext 和 concat_text 函数在存在经 TOAST 处理的值时实际上是错误的，因为它们没有对其输入调用 pg_detoast_datum()。（旧风格动态装载函数的处理器目前会处理这一细节，但效率低于版本 1 函数所能达到的水平。）

版本 1 函数的一大改进是对 NULL 输入和结果的更好处理。宏 PG_ARGISNULL(*n*) 允许函数测试每个输入是否为 NULL（当然，只有在未声明为“严格”的函数中才有必要这样做）。与 PG_GETARG_*xxx*() 宏一样，输入参数从零开始计数。注意，在确认参数不是 NULL 之前应避免执行 PG_GETARG_*xxx*()。要返回 NULL 结果，执行 PG_RETURN_NULL()；它在严格和非严格函数中都可用。

版本 1 函数调用约定使得返回“集合”结果、实现触发器函数和过程语言调用处理器成为可能。版本 1 代码也比版本 0 更具可移植性，因为它不违反 ANSI C 对函数调用协议的限制。更多细节见源码发行包中的 src/backend/utils/fmgr/README。

12.5.5. C 语言函数中的复合类型

复合类型没有像 C 结构那样的固定布局。复合类型的实例可能包含空字段。此外，属于某个继承层次的复合类型可能具有与同一继承层次其他成员不同的字段。因此，PostgreSQL 提供了一个从 C 访问复合类型字段的过程式接口。当 PostgreSQL 处理一组行时，每一行都会作为一个 TUPLE 类型的不透明结构传入你的函数。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';
```

在上面的查询中，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;
```

```
    int32 salary;

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
    if (isnull)
        return (false);
    return salary > limit;
}

/* In version-1 coding, the above would look like this: */

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，返回当前行中的属性。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。

下面的命令让 PostgreSQL 知道 `c_overpaid` 函数：

```
CREATE FUNCTION c_overpaid(emp, int4)
RETURNS bool
AS 'PGROOT/tutorial/funcs'
LANGUAGE C;
```

虽然有办法在 C 函数内构造新行或修改现有行，但这些方法过于复杂，本手册不予讨论。示例请查阅后端源代码。

12.5.6. 编写代码

现在我们转向编写编程语言函数这一更困难的任务。
请注意：本手册的这一部分不会使你成为程序员。在尝试为 PostgreSQL 编写 C 函数之前，你必须对 C（包括指针和 malloc 内存管理器的使用）有良好的理解。虽然也许可以把用 C 以外语言编写的函数装载到 PostgreSQL，但这通常很困难（即使在可能的时候），因为其他语言（如 FORTRAN 和 Pascal）往往不遵循与 C 相同的调用约定。也就是说，其他语言在函数之间传递参数和返回值的方式不同。因此，我们将假定你的编程语言函数是用 C 编写的。

构建 C 函数的基本规则如下：

- 使用 `pg_config --includedir-server` 找出 PostgreSQL 服务器头文件安装在你的系统（或你的用户将要运行的系统）上的位置。此选项是 PostgreSQL 7.2 新增的。对 PostgreSQL 7.1，应使用选项 `--includedir`。（如果遇到未知选项，`pg_config` 会以非零状态退出。）对 7.1 之前的版本你只能猜测，但由于那是在当前调用约定引入之前，你不太可能想支持那些版本。
- 分配内存时，使用 PostgreSQL 的例程 `palloc` 和 `pfree`，而不是相应的 C 库例程 `malloc` 和 `free`。`palloc` 分配的内存会在每个事务结束时自动释放，从而防止内存泄漏。
- 总是使用 `memset` 或 `bzero` 把结构体的字节清零。有几个例程（如哈希访问方法、哈希连接和排序算法）会计算结构体中原始位的函数。即使你初始化了结构体的所有字段，结构体中仍可能存在若干包含垃圾值的对齐填充字节（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`。
- 对象文件中定义的符号名不得相互冲突，也不得与 PostgreSQL 服务器可执行文件中定义的符号冲突。如果收到此类错误消息，你必须重命名你的函数或变量。
- 编译和链接你的目标代码使其能被动态装载到 PostgreSQL 总是需要特殊的标志。关于如何针对你的特定操作系统进行操作，参见第 12.5.7 节的详细说明。

12.5.7. 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
```

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**
>
> 如果你想把扩展模块打包以便广泛分发，应该考虑使用 GNU Libtool[1] 来构建共享库。它把平台差异封装到一个通用而强大的接口之后。严肃的打包还需要考虑库版本管理、符号解析方法以及其他问题。

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 12.5.1 节。

12.6. 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。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 ...
```

[1] http://www.gnu.org/software/libtool/

```
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

对诸如 `test(1, 1.5)` 这样的简单输入，哪个函数会被调用并不直接明了。当前实现的解析规则在用户指南中描述，但设计一个微妙地依赖此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。

在 PostgreSQL 7.0 之前，这种替代语法不存在。有一个绕过该问题的技巧：先定义一组名称不同的 C 函数，然后定义一组同名的 SQL 函数包装器，它们接受适当的参数类型并调用相匹配的 C 函数。

12.7. 过程语言处理器

对以编译语言当前“版本 1”接口以外的语言编写的函数的所有调用（包括用户自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），都要经过该特定语言的调用处理器函数。调用处理器的职责是以有意义的方式执行函数，例如解释所提供的源文本。
本节描述如何编写语言调用处理器。这并不是一项常见任务——事实上，在 PostgreSQL 的历史上它只被完成过寥寥几次——但这一主题自然属于本章，而且这些材料也许能让人对 PostgreSQL 系统的可扩展本质有所认识。

过程语言的调用处理器是一个“普通”函数，它必须用一种编译语言（如 C）编写，并向 PostgreSQL 注册为不带参数并返回 `opaque` 类型（未指定或未定义类型的占位符）。这防止调用处理器在查询中作为函数被直接调用。（不过，当要执行该处理器所提供语言中的函数时，在实际调用处理器时可以提供参数。）

> **注意**
>
> 在 PostgreSQL 7.1 及之后的版本中，调用处理器必须遵循“版本 1”函数管理器接口，而不是旧风格接口。

调用处理器的调用方式与其他函数相同：它收到一个指向 `FunctionCallInfoData` 结构（包含参数值和所调用函数的信息）的指针，并应返回一个 `Datum` 结果（如果它想返回 SQL NULL 结果，还可以设置 `FunctionCallInfoData` 结构的 `isnull` 字段）。调用处理器与普通被调函数的区别在于，`FunctionCallInfoData` 结构的 `flinfo->fn_oid` 字段将包含要调用的实际函数的 OID，而不是调用处理器自身的 OID。调用处理器必须使用此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（而不是调用处理器）的声明建立的。

获取 `pg_proc` 条目并分析被调用过程的参数和返回类型是调用处理器的职责。过程 `CREATE FUNCTION` 的 AS 子句可在 `pg_proc` 表条目的 `prosrc` 属性中找到。它可能是过程语言本身的源文本（如 PL/Tcl），也可能是指向文件的路径名，或者是任何能详细告诉调用处理器该做什么的东西。

同一函数在一个 SQL 语句中往往被调用多次。调用处理器可以利用 flinfo->fn_extra 字段避免对被调用函数信息的重复查找。该字段最初为 NULL，但调用处理器可以把它设置为指向有关 PL 函数的信息。在后续调用中，如果 flinfo->fn_extra 已非 NULL，就可以直接使用它并跳过信息查找步骤。调用处理器必须小心让 flinfo->fn_extra 指向至少能存活到当前查询结束的内存，因为 FmgrInfo 数据结构可能被保存那么久。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；这样的数据通常与 FmgrInfo 本身具有相同的生存期。但处理器也可以选择使用生存期更长的上下文，以便跨查询缓存函数定义信息。

当 PL 函数作为触发器被调用时，不传递显式参数，但 FunctionCallInfoData 的 context 字段指向一个 TriggerData 结点而不是像普通函数调用那样为 NULL。语言处理器应提供让 PL 函数获取触发器信息的机制。

这是一个用 C 编写的 PL 处理器的模板：

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "utils/elog.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;
}
```

只需再加上几千行代码代替这些省略号即可完成调用处理器。关于如何把它编译成可装载模块，参见第 12.5 节。

下面的命令随后注册这个示例过程语言：

```
CREATE FUNCTION plsample_call_handler () RETURNS opaque
    AS '/usr/local/pgsql/lib/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

第 13 章 扩展 SQL：类型

如前所述，PostgreSQL中有两类类型：基本类型（用编程语言定义）和组合类型。
本章描述如何定义新的基本类型。

本节中的示例可以在 tutorial 目录下的 `complex.sql`和`complex.c`中找到。组合类型的示例在`funcs.sql`中。

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输入，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，并返回一个以空字符结尾的字符串。

假设我们想定义一个表示复数的 complex 类型。很自然，我们会选择用下面的C结构在内存中表示复数：

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

并选择形如`(x,y)`的字符串作为外部字符串表示。

这些函数通常不难编写，输出函数尤其如此。不过，有几点需要记住：

- 在定义你的外部（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：

  ```
  Complex *
  complex_in(char *str)
  {
      double x, y;
      Complex *result;
      if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2) {
          elog(ERROR, "complex_in: error in parsing %s", str);
          return NULL;
      }
      result = (Complex *)palloc(sizeof(Complex));
      result->x = x;
      result->y = y;
      return (result);
  }
  ```

  输出函数可以简单地写成：

  ```
  char *
  complex_out(Complex *complex)
  {
      char *result;
      if (complex == NULL)
          return(NULL);
      result = (char *) palloc(60);
  ```

```
    sprintf(result, "(%g,%g)", complex->x, complex->y);
    return(result);
}
```

- 你应当尽量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（比如读到另一台计算机上某人的数据库中），
  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。

要定义complex类型，我们需要在创建类型之前先创建 complex_in和complex_out 这两个用户定义的函数：

```
CREATE FUNCTION complex_in(opaque)
    RETURNS complex
    AS 'PGROOT/tutorial/complex'
    LANGUAGE C;

CREATE FUNCTION complex_out(opaque)
    RETURNS opaque
    AS 'PGROOT/tutorial/complex'
    LANGUAGE C;
```

最后，我们可以声明该数据类型：

```
CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out
);
```

如前面所讨论的，PostgreSQL完全支持基本类型的数组。此外，PostgreSQL也支持用户定义类型的数组。当你定义一个类型时，PostgreSQL会自动提供对该类型数组的支持。由于历史原因，数组类型的名称与用户定义类型相同，只是在前面加上下划线字符_。

组合类型不需要在其上定义任何函数，因为系统已经知道它们内部是什么样子。

如果你的数据类型（datatype）的值（内部形式）大小可能超过几百字节，
就应当小心地把它们标记为支持 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）。然后，所有接受该类型值的函数都必须小心地对提供的值调用 pg_detoast_datum()——如果你的函数不是 strict 的，则要在检查该值不为 NULL 之后进行。最后，在给出 CREATE TYPE命令时选择合适的存储选项。

第 14 章 扩展 SQL：操作符

14.1. 简介

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，同一个操作符名可用于操作数数量和类型不同的多个操作符。
如果出现歧义情况而系统无法确定该用哪个操作符，它会返回一个错误。
你可能需要对左操作数和/或右操作数进行类型转换，帮助系统理解你想用哪个操作符。

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。本章的大部分篇幅将用于解释这些额外信息。

14.2. 示例

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 13 章）。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 'PGROOT/tutorial/complex'
    LANGUAGE C;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

现在我们可以这样做：

```
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文。

14.3. 操作符优化信息

作者

由 Tom Lane 撰写。

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后端崩溃、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 7.2.8 版本理解的所有子句。

14.3.1. 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。

被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。

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。（注意：这一过程在 6.5 之前的 PostgreSQL 版本中不能可靠工作，但现在它是推荐的做法。）

14.3.2. 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。

操作符的求反器必须与该操作符本身具有相同的左和/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。

提供求反器对查询优化器很有帮助，因为它允许把形如 NOT (x = y) 的表达式化简为 x <> y。这种情况比你想象的更常见，因为其他整理过程可能会插入 NOT 操作。

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。

14.3.3. 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 src/backend/utils/adt/selfuncs.c 中函数 convert_to_scalar() 所理解的范围。（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。

src/backend/utils/adt/geo_selfuncs.c 中还有为几何操作符设计的额外选择率函数：areasel、positionsel 和 contsel。在写作本文时它们只是存根，但你可能还是想用它们（甚至更好的是，改进它们）。

14.3.4. 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，是猜测一对表中有多少比例的行会满足如下形式的 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

14.3.5. HASHES

如果给出 HASHES 子句，它告诉系统可以对此操作符上的连接使用哈希连接方法。HASHES 只对返回 boolean 的二元操作符有意义，而且实践中该操作符最好是某种数据类型上的相等操作。

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。

事实上，仅逻辑上的相等也不够；该操作符最好表示纯粹的按位相等，
因为哈希函数是按值的内存表示计算的，而不管这些位是什么含义。例如，
时间间隔的相等就不是按位相等；时间间隔相等操作符认为两个时间间隔只要时长相同就相等，
无论其端点是否一致。这意味着在时间间隔字段上使用 = 的连接，如果实现为哈希连接，得到的结果会与其他实现方式不同，因为本应匹配的很大一部分值对会被哈希到不同的值，
永远不会被哈希连接比较。而如果优化器选择使用另一种连接，
相等操作符判定相等的所有值对都会被找到。我们不想要这种不一致，
因此我们没有把时间间隔的相等操作标记为可哈希。

还有一些与机器相关的情形可能让哈希连接出错。例如，
如果你的数据类型是一个可能含有无关填充位的结构，把它的相等操作符标记为 HASHES 是不安全的。（除非也许你把其他操作符写成保证未用的位始终为零。）
另一个例子是浮点数据类型用于哈希连接是不安全的。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），但按定义它们比较相等。因此，
如果把浮点数据类型上的相等操作符标记为 HASHES，负零和正零可能不会被哈希连接配对，而任何其他连接过程都会把它们配上。

归根结底：你可能只应对那些（可以）用 memcmp() 实现的相等操作符使用 HASHES。

14.3.6. SORT1 and SORT2

如果给出 SORT 子句，它们告诉系统可以对此操作符上的连接使用归并连接方法。指定其一就必须同时指定另一个。当前操作符必须是某对数据类型上的相等操作，而 SORT1 和 SORT2 子句分别命名左边和右边数据类型的排序操作符（“<” 操作符）。

归并连接基于这样的思想：把左右两边的表排好序，然后并行扫描它们。因此，
两个数据类型都必须能够全序化，而且连接操作符必须只在处于排序次序中“相同位置”
的值对上才可能成功。实践中这意味着连接操作符的行为必须像相等一样。
但与哈希连接不同——哈希连接要求左右数据类型最好相同（或至少按位等价）——只要两个不同的数据类型在逻辑上兼容，就可以对它们做归并连接。例如，int2 对 int4 的相等操作符就是可归并连接的。我们只需要能把两个数据类型排成逻辑兼容次序的排序操作符。

指定归并排序操作符时，当前操作符和两个被引用的操作符都必须返回 boolean；SORT1 操作符的两个输入数据类型都必须等于当前操作符的左操作数类型，SORT2 操作符的两个输入数据类型都必须等于当前操作符的右操作数类型。（与 COMMUTATOR 和 NEGATOR 一样，这意味着只凭操作符名就足以指定操作符，而且如果你碰巧先定义了相等操作符、后定义其他操作符，系统也能够创建占位操作符条目。）

实践中你只应对 = 操作符写 SORT 子句，而且两个被引用的操作符应当总是命名为 <。试图对名称不是这些的操作符使用归并连接将导致不可救药的混乱，原因我们稍后就会看到。

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，归并连接在运行时可能失败：

- 可归并连接的相等操作符必须有交换子（两个数据类型相同时就是它自身，不同时则是一个相关的相等操作符）。
- 必须存在与可归并连接操作符本身具有相同左右操作数数据类型的 < 和 > 排序操作符。这些操作符必须命名为 < 和 >；这件事你没有选择的余地，因为没有显式指定它们的条款。注意，如果左右数据类型不同，这两个操作符都不是任何一个 SORT 操作符。但它们最好与 SORT 操作符以兼容的方式对数据值排序，否则归并连接将无法工作。

第 15 章 扩展 SQL：聚合

PostgreSQL 中的聚合函数被表示为状态值和状态转换函数。也就是说，聚合可以按照每当处理一个输入项就被修改的状态来定义。要定义一个新的聚合函数，
需要为状态值选择一种数据类型、为状态选择一个初始值，以及一个状态转换函数。
状态转换函数只是一个普通函数，也可以在聚合上下文之外使用。还可以指定一个最终函数，
以防聚合的期望输出与需要保存在运行状态值中的数据不同。

因此，除了聚合的用户所能看到的输入和结果数据类型之外，还有一个内部的状态值数据类型，
它可能与输入和结果类型都不同。

如果我们定义一个不使用最终函数的聚合，得到的就是一个对每行列值计算运行函数的聚合。
`Sum` 就是这类聚合的一个例子。`Sum` 从零开始，总是把当前行的值加到它的运行总和上。例如，
如果我们想让 `sum` 聚合作用于复数的数据类型，我们只需要该数据类型的加法函数。聚合定义是：

```
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)
```

（实践中我们只会把该聚合命名为 `sum`，让 PostgreSQL 自行判断对 `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” 转换函数的另一项默认行为是：每当遇到 NULL 输入值时，先前的状态值保持不变。这样，NULL 值就被忽略了。如果你需要 NULL 输入有其他行为，只需把转换函数定义为非 strict，并让它测试 NULL 输入并做任何需要的处理。

`Avg`（平均值）是聚合的一个更复杂的例子。
它需要两个运行状态：输入的总和以及输入的数量。最终结果通过这两个量相除得到。
平均值通常用二元数组作为转换状态值来实现。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}'
);
```

更多细节参见参考手册中对 `CREATE AGGREGATE` 命令的描述。

第 16 章 规则系统

作者

由 Jan Wieck 编写。7.1 的更新由 Tom Lane 完成。

16.1. 简介

产生式规则系统在概念上很简单，但在实际使用时会涉及许多微妙之处。其中一些要点，以及PostgreSQL规则系统的理论基础，可以在[ston90b]中找到。

有些其他数据库系统定义了主动型数据库规则。它们通常是存储过程和触发器，在PostgreSQL中以函数和触发器的形式实现。

查询重写规则系统（下文简称规则系统）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。这一规则系统的能力在[ong90]以及[ston90b]中有讨论。

16.2. 什么是查询树?

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。

规则系统位于查询解析器和规划器之间。它接收解析器的输出，即一棵查询树，以及来自 `pg_rewrite` 目录的重写规则；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。如果以调试级别 4 启动 PostgreSQL 后端，并在交互式后端界面中输入查询，就可以看到这些查询树。`pg_rewrite` 系统目录中的规则动作同样以查询树的形式存储。它们的格式与调试输出不同，但包含的却是完全相同的信息。

阅读查询树需要一些经验，我刚开始在规则系统上工作时也经历了一段艰难的时光。我记得我曾站在咖啡机前，看到目标列表里有一个杯子，范围表里有水和咖啡粉，而所有的按钮出现在一个条件表达式中。由于查询树的SQL表示已足以理解规则系统，本文不会讲解如何阅读它们。学习阅读查询树也许会有帮助，而且在后续的描述中需要用到这些命名约定。

16.2.1. 查询树的组成部分

在阅读本文中查询树的 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，说明了是哪一种命令（SELECT、INSERT、UPDATE、DELETE）产生了解析树。

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 FROM 关键字后面给出的关系。

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。
本文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、INSERT ... SELECT 序列几乎相同，这里不再单独讨论。

在 INSERT、UPDATE 和 DELETE 查询中，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。（* 只是某个关系全部属性名的缩写。解析器会把它展开为各个独立属性，因此规则系统永远不会看到它。）

DELETE 查询不需要目标列表，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。

在 INSERT 查询中，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。结果关系中缺失的列将由规划器填入常量空值表达式。

在 UPDATE 查询中，目标列表描述要替换旧行的新行。在规则系统中，它只包含查询中 SET attribute = 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补上缺失列。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属性的变量、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）。它就是 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。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的项，但这与规则系统的基本原理关系不大。

16.3. 视图和规则系统

16.3.1. PostgreSQL 中视图的实现

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 attribute list as for mytab);
CREATE RULE "_RETmyview" 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，对查询解析器而言，表和视图完全没有区别。它们是同一种东西——关系。这是目前最重要的一点。

16.3.2. 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。它们与其他规则在语义上不同，因为它们是就地修改解析树，而不是创建新的解析树。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。

本文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。

用于操作这些示例的数据库名为 al_bundy。你很快就会明白为什么取这个名字。它还需要安装过程语言 PL/pgSQL，因为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 '
    BEGIN
        IF $1 < $2 THEN
            RETURN $1;
        END IF;
        RETURN $2;
    END;
' LANGUAGE plpgsql;
```

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   char(10),      -- primary key
    sh_avail   integer,       -- available # of pairs
    slcolor    char(10),      -- preferred shoelace color
    slminlen   float,         -- miminum shoelace length
    slmaxlen   float,         -- maximum shoelace length
    slunit     char(8)        -- length unit
);

CREATE TABLE shoelace_data (
    sl_name    char(10),      -- primary key
    sl_avail   integer,       -- available # of pairs
    sl_color   char(10),      -- shoelace color
    sl_len     float,         -- shoelace length
    sl_unit    char(8)        -- length unit
);

CREATE TABLE unit (
    un_name    char(8),       -- the primary key
    un_fact    float          -- factor to transform to cm
);
```

我想我们大多数人都穿鞋，都能看出这确实是很有用的数据。当然，
世上有一些不需要鞋带的鞋，但这并不能让 Al 的日子好过一点，所以我们忽略这一点。

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（这是我们这里最简单的一个例子）的 CREATE VIEW 命令会创建一个关系 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。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 *CURRENT*），与 SELECT 规则无关。

现在我们填充 unit、shoe_data 和 shoelace_data，然后 Al 在他的一生中第一次输入 SELECT：

```
al_bundy=> INSERT INTO unit VALUES ('cm', 1.0);
al_bundy=> INSERT INTO unit VALUES ('m', 100.0);
al_bundy=> INSERT INTO unit VALUES ('inch', 2.54);
al_bundy=>
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh1', 2, 'black', 70.0, 90.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh2', 0, 'black', 30.0, 40.0, 'inch');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh4', 3, 'brown', 40.0, 50.0, 'inch');
al_bundy=>
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl1', 5, 'black', 80.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl2', 6, 'black', 100.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl3', 0, 'black', 35.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl4', 8, 'black', 40.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl5', 4, 'brown', 1.0 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl6', 0, 'brown', 0.9 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl7', 7, 'brown', 60 , 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl8', 1, 'brown', 40 , 'inch');
al_bundy=>
al_bundy=> 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 Al 在这些视图上能执行的最简单的 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，会生成如下解析树：

```
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 pg_rewrite 中是否有关系对应的规则。在处理 shoelace 的范围表项时（到目前为止只有这一个），它会找到带有如下解析树的规则 _RETshoelace：

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       float8mul(s.sl_len, u.un_fact) AS sl_len_cm
  FROM shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE bpchareq(s.sl_unit, u.un_name);
```

注意，解析器把计算和限定条件都改写成了对相应函数的调用。
但这实际上并没有改变任何东西。

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解析树，
然后用这个范围表条目替换原先引用视图的条目。所得的重写后解析树几乎等同于 Al 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。

现在我们让 Al 面对这样一个问题：布鲁斯兄弟（Blues Brothers）来到他的店里，想买一些新鞋；而且正如布鲁斯兄弟的本色那样，他们要穿一模一样的鞋。他们还想立刻就穿上，
所以还需要鞋带。

Al 需要知道，商店里目前哪些鞋子有颜色和尺码都匹配的鞋带，并且完全匹配的总双数大于等于二。我们教会他怎么做，于是他询问他的数据库：

```
al_bundy=> 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)
```

Al 是一位鞋类行家，所以他知道只有 sh1 型号的鞋才合适（鞋带 sl7 是棕色的，而需要棕色鞋带的鞋是布鲁斯兄弟绝不会穿的鞋）。

这一次解析器的输出是如下解析树：

```
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 int4ge(shoe_ready.total_avail, 2);
```

首先应用的是用于 shoe_ready 视图的规则，它得到如下解析树：

```
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 int4ge(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 int4ge(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 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（使后端不断膨胀，
> 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。

16.3.3. 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解析树的两个细节：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 的解析树与其他任何命令的解析树之间只有少数差别。显然，它们的命令类型不同，而且这一次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，都具有属性 a 和 b，那么下面两条语句的解析树：

```
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。

- 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 的范围表项中的属性 b。
- 条件表达式会比较两个范围中的属性 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，这两个解析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（通过查看解析树）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）。这是一个系统属性，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。目前，表行不会被覆盖，这也是 ABORT TRANSACTION 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 所指向的行，其元组头中的 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。

16.3.4. 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解析树。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解析树（unit 以不同名称使用了两次）。

16.3.4.1. 好处

用规则系统实现视图的好处在于：规划器能够在单棵解析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。现在，规划器必须决定执行查询的最佳路径。它掌握的信息越多，这个决定就能做得越好。PostgreSQL实现的规则系统能够保证，到此为止，关于该查询的全部可用信息都已经集中在这里。

16.3.5. 那么更新视图会怎么样？

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非 SELECT 查询的行为。这是下一节的主题。

16.4. INSERT、UPDATE 和 DELETE 上的规则

16.4.1. 与视图规则的区别

定义在 ON INSERT、UPDATE 和 DELETE 上的规则与前一节描述的视图规则完全不同。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。
- INSTEAD 关键字是可选的。
- 伪关系 NEW 和 OLD 可以派上用场。
- 它们可以有规则条件。

第二，它们不是就地修改解析树，而是创建零棵或多棵新解析树，并且可能丢弃原始查询树。

16.4.2. 这些规则如何工作

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 ON INSERT、UPDATE 或 DELETE 上的规则。

当解析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解析树列表，初始时该列表为空。
动作可以有零个（NOTHING　关键字）、一个或多个。为简化说明，我们来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 NEW 和/或 OLD 伪关系，它们基本上就是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解析树。

- 没有条件，也不是 INSTEAD：
  - 规则动作的解析树，其中附加了原始解析树的条件。
- 没有条件，但带有 INSTEAD：
  - 规则动作的解析树，其中附加了原始解析树的条件。
- 给出了条件，不是 INSTEAD：
  - 规则动作的解析树，其中附加了规则条件和原始解析树的条件。
- 给出了条件，带有 INSTEAD：
  - 规则动作的解析树，其中附加了规则条件和原始解析树的条件。

- 原始解析树，其中附加了规则条件取反后的条件。

最后，如果规则不是 INSTEAD，就把未经更改的原始解析树加入列表。由于只有带条件的 INSTEAD 规则已经加入了原始解析树，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解析树。

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。

由规则动作生成的解析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解析树。因此，规则动作中的解析树必须具有不同的命令类型，
或者具有不同的结果关系。否则，这种递归过程就会陷入循环。目前内置的递归上限是 10 次迭代。如果经过 10 次迭代之后仍有更新规则要应用，规则系统就会认为出现了多个规则定义之间的循环，并报告一个错误。

`pg_rewrite` 系统目录中动作里的解析树只是模板。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 NULL（对于 INSERT）。任何对 OLD 的引用，都会被替换为对作为结果关系的那个范围表项的引用。

在我们完成应用更新规则之后，再对生成的解析树应用视图规则。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。

16.4.2.1. 第一个规则：逐步分析

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    char(10),      -- 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 Al 执行了：

```
al_bundy=> UPDATE shoelace_data SET sl_avail = 6

al_bundy->     WHERE sl_name = 'sl7';
```

然后我们查看日志表：

```
al_bundy=> 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。
解析器创建了解析树（这一次突出显示的是原始解析树中的部分，因为对更新规则而言，操作的基础是规则动作）：

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE bpchareq(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：

```
int4ne(NEW.sl_avail, OLD.sl_avail)
```

以及一个动作：

```
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 子句只是为了表明解析树中存在用于 *NEW* 和 *OLD* 的范围表项。之所以需要这些项，是为了让 INSERT 命令的 querytree 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解析树：修改后的规则动作，以及原始解析树。第一步中，原始查询的范围表会并入规则动作的解析树，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 int4ne(*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 步把原始解析树的条件加进去，把结果集进一步限制为只有那些原始解析树会触及的行：

```
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 int4ne(*NEW*.sl_avail, *OLD*.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

第 4 步把 NEW 引用替换为原始解析树中的目标列表项，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 int4ne(6, *OLD*.sl_avail)
   AND bpchareq(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 int4ne(6, shoelace_data.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

至此就完成了。由于规则不是 INSTEAD，我们还要输出原始解析树。简而言之，规则系统输出的是一个包含两棵解析树的列表，它们等同于以下语句：

```
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，而这正是该规则所定义的行为。
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，原始解析树不包含 sl_avail 的目标列表项，因此 NEW.sl_avail 会被 shoelace_data.sl_avail 替换，于是规则产生的额外查询是：

```
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。因此，如果 Al 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。这一次，原始解析树的条件不同，因此规则会产生额外的解析树：

```
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';
```

这棵解析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，为什么原始解析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。

16.4.3. 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尝试 INSERT、UPDATE 和 DELETE，一种简单的办法是让那些解析树直接被丢弃。我们创建如下规则：

```
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 Al 尝试对视图关系 `shoe` 执行这些操作中的任何一种，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，生成的解析树列表将为空；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。

> **注意**
>
> 这种方式可能会让前端应用程序感到困惑，因为数据库上绝对什么都没有发生，因而后端不会为该查询返回任何东西，甚至连 `PGRES_EMPTY_QUERY` 也不会在 libpq 中出现。在 psql 中，什么都不会发生。这一点将来可能会改变。

另一种更完善的做法，是创建一些规则，把解析树重写成在真实表上执行正确操作的解析树。要在 `shoelace` 视图上做到这一点，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;
```

现在有一批鞋带到货进入 Al 的商店，随附一份很长的部件清单。Al 不太擅长计算，因此我们不想让他手工更新 shoelace 视图。于是我们建立两个小表：一个让他可以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    char(10),
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     char(10),
    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 Al 可以坐下来随便干点什么，直到

```
al_bundy=> SELECT * FROM shoelace_arrive;
arr_name  |arr_quant
----------+---------
sl3       |       10
sl6       |       20
sl8       |       20
(3 rows)
```

与部件清单上的内容完全一致为止。我们快速查看一下当前数据，

```
al_bundy=> 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)
```

把到货的鞋带入库：

```
al_bundy=> INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

然后检查结果：

```
al_bundy=> 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)

al_bundy=> 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)
```

从一条 INSERT ... SELECT 到这些结果，中间要经历相当长的一段过程。对它的描述将是本文中的最后一个（但不是最后一个例子 :-)）。首先是解析器的输出：

```
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 = int4pl(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 bpchareq(shoelace.sl_name, showlace_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，随后应用的规则 `shoelace_upd` 产生：

```
UPDATE shoelace_data SET
       sl_name = shoelace.sl_name,
       sl_avail = int4pl(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 showlace_data
 WHERE bpchareq(shoelace.sl_name, showlace_arrive.arr_name)
   AND bpchareq(shoelace_data.sl_name, shoelace.sl_name);
```

这同样是一条 INSTEAD 规则，因此前一个解析树会被丢弃。注意，这个查询仍然使用视图 shoelace。但规则系统在这个循环中尚未完成，因此它会继续在其上应用规则 _RETshoelace，于是得到：

```
UPDATE shoelace_data SET
       sl_name = s.sl_name,
       sl_avail = int4pl(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 showlace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE bpchareq(s.sl_name, showlace_arrive.arr_name)
   AND bpchareq(shoelace_data.sl_name, s.sl_name);
```

又有一条更新规则被应用，于是车轮继续转动，我们进入了第 3 轮重写。这一次应用的是规则 log_shoelace，这产生了额外的解析树：

```
INSERT INTO shoelace_log SELECT
       s.sl_name,
       int4pl(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 showlace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u,
       shoelace_data *OLD*, shoelace_data *NEW*
       shoelace_log shoelace_log
 WHERE bpchareq(s.sl_name,  showlace_arrive.arr_name)
   AND bpchareq(shoelace_data.sl_name, s.sl_name);
   AND int4ne(int4pl(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，返回生成的解析树。于是我们最终得到两棵解析树，它们等同于以下 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。观察这两个查询就会发现，`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
```

后者完全会在日志关系中产生相同的项。因此，规则系统导致了对 `shoelace_data` 关系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。

最后来演示一下 PostgreSQL 规则系统及其威力。有一位漂亮的金发女郎卖鞋带。而 Al 永远不会意识到的是，她不仅漂亮，还很聪明——甚至有点太聪明了。于是，Al 不时会订购一些完全卖不出去的鞋带。这一次他订购了 1000 双品红色（magenta）鞋带，而且由于另一种鞋带暂时缺货、而他已承诺购买一些，他又为粉色（pink）的鞋带做好了数据库准备：

```
al_bundy=> INSERT INTO shoelace VALUES
```

```
al_bundy->     ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
al_bundy=> INSERT INTO shoelace VALUES
al_bundy->     ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

由于这种事经常发生，我们必须不时地找出那些绝对不适合任何鞋子的鞋带项。
我们可以每次都用一条复杂的语句来完成，也可以为此建立一个视图。这个视图是：

```
CREATE VIEW shoelace_obsolete AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

它的输出是：

```
al_bundy=> SELECT * FROM shoelace_obsolete;
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
```

那 1000 双品红色鞋带，我们得先向 Al 讨了债才能把它们扔掉，不过那是另一个问题了。粉色的条目我们要删掉。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_candelete AS
    SELECT * FROM shoelace_obsolete WHERE sl_avail = 0;
```

然后这样执行：

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_candelete
             WHERE sl_name = shoelace.sl_name);
```

Voilà：

```
al_bundy=> 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解析树，从真正的表中删除所请求的数据。

我想在现实世界中，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。

实情是：. 在写这一章的过程中，我又发现了一个 bug。不过修完那个 bug 之后，这一切居然能工作，倒是让我有点惊讶。

16.5. 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 bool);
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，对 phone_data 的读访问会按照该用户的权限进行检查，于是查询被判定为允许。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中止事务）。而且，一旦用户发现秘书开放了其 phone_number 视图，用户就可以 REVOKE 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。

有人可能会认为这种逐条规则的检查是一个安全漏洞，但实际上并不是。如果它不这样工作，
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，他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，然后使用 REVOKE 了。

这种机制对更新规则同样有效。在上一节的示例中，Al 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 al。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，Al 也能看到日志记录。但他不能伪造记录，也不能操纵或删除已有记录。

> **警告**
>
> GRANT ALL 目前会包含 RULE 权限。这意味着被授权的用户可以先删除规则、做出更改、再把规则装回去。我认为这一点应该尽快改变。

16.6. 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。目前不能用规则实现的内容是一些种类的约束。

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，把查询重写成 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，目前就必须使用触发器。

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，而规则会修改解析树或生成额外的解析树。因此，
如果一条语句会影响很多行，那么发出一条额外查询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。

例如：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` 上的索引是唯一索引。`hostname` 列包含计算机的完整域名。规则或触发器应实现这样一个约束：从 software 中删除引用了被删除主机的行。由于从 computer 中删除的每一行都会调用触发器，它可以在一个准备并保存好的计划中使用语句：

```
DELETE FROM software WHERE hostname = $1;
```

并通过参数传入 `hostname`。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），由触发器发出的查询也会是一次索引扫描（同样很快）。规则产生的额外查询则是：

```
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。有两种查询可以做到这件事。其中一种是：

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

规则查询的计划将是：

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

另一种可能的查询是：

```
DELETE FROM computer WHERE hostname ~ '^old';
```

其执行计划为：

```
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 上 hostname 的限制同样也可以用于 software 上的索引扫描，而在该查询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。规则实现则用两条走索引的查询完成这件事。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。即使所有用于查找的索引块很快都会进入缓存，通过 SPI 管理器执行 2000 条查询仍然要耗费一些时间。

我们要看的最后一个查询是：

```
DELETE FROM computer WHERE manufacurer = 'bim';
```

同样，这也可能导致从 computer 中删除很多行。因此触发器又会向执行器触发很多条查询。但规则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
```

它来自规则的查询：

```
DELETE FROM software WHERE computer.manufacurer = 'bim'
                       AND software.hostname = computer.hostname;
```

在上述任何一种情况下，规则系统产生的额外查询都或多或少与该查询影响的行数无关。

另一种情况是 UPDATE 中是否执行动作取决于某个属性是否发生变化。在 PostgreSQL 6.4 版中，规则事件的属性说明被禁用了（它最迟会在 6.5 恢复，也许更早——敬请期待）。因此，目前创建像 shoelace_log 例子中那样规则的唯一方式是使用规则条件。这会导致一条额外的查询总是被执行，即使所关注的属性根本不可能变化，因为它没有出现在初始查询的目标列表中。
等这一功能重新启用后，它将成为规则相对触发器的又一个优势。在这种情况下，
触发器的优化按定义必然失败，
因为"其动作只在某个特定属性被更新时执行"这一事实隐藏在它的功能之中。
触发器的定义只允许在行级指定，因此每当某一行被触及，触发器就必须被调用来做决定。
而规则系统会通过查看目标列表知道这一点，并在属性未被涉及时完全抑制那条额外的查询。
所以，无论带不带条件，规则只会在可能有事可做时才执行其扫描。

只有当规则动作导致了规模很大且条件很差的连接（规划器无能为力的情形）时，
规则才会明显慢于触发器。规则是一把大锤。不加小心地使用大锤可能造成巨大的破坏。
但用对了手法，它们能把任何钉子一击入帽。

第 17 章 索引扩展接口

17.1. 简介

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，
我们还不能在新类型或其操作符上定义一个二级索引（例如 B-树、R-树或哈希访问方法）。

回头看图 11.1。右半部分显示了我们必须修改的目录，以便告诉 PostgreSQL 如何在索引中使用用户定义的类型和/或用户定义的操作符（即 pg_am, pg_amop, pg_amproc, pg_operator 和 pg_opclass）。遗憾的是，没有一条简单的命令可以完成这件事。我们将通过一个贯穿始终的示例来演示如何修改这些目录：为 B-tree 访问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。

17.2. 访问方法

pg_am 表为每个索引访问方法保存一行。堆访问方法的支持内置于 PostgreSQL 中，但所有其他访问方法都在 pg_am 中描述。其模式如表 17.1所示。

表 17.1. 索引访问方法模式

列	描述
amname	访问方法的名称
amowner	属主的用户 ID（当前未使用）
amstrategies	该访问方法的策略数目（见下文）
amsupport	该访问方法的支持例程数目（见下文）
amorderstrategy	如果索引不提供排序顺序则为零，否则是描述该排序顺序的策略操作符的策略号
amcanunique	该访问方法是否支持唯一索引？
amcanmulticol	该访问方法是否支持多列索引？
amindexnulls	该访问方法是否支持 NULL 索引项？
amconcurrent	该访问方法是否支持并发更新？
amgettuple	
aminsert	
...	访问方法接口例程的过程标识符。例如，打开、关闭访问方法以及从中获取行的 regproc ID 就出现在这里。

pg_am 中该行的对象 ID 被用作许多其他表中的外键。你不需要向这个表添加新行；你所关心的只是你想扩展的访问方法的对象 ID：

```
SELECT oid FROM pg_am WHERE amname = 'btree';

 oid
-----
 403
(1 row)
```

我们稍后会在 WHERE 子句中使用该查询。

17.3. 访问方法策略

amstrategies 列的存在是为了标准化跨数据类型的比较。例如，B-树对键施加了严格的从小到大的顺序。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。事实上，某些访问方法根本不施加任何排序。例如，R-树表达一种矩形包含关系，而哈希数据结构只表达基于哈希函数值的按位相似性。PostgreSQL 需要某种一致的方式，来接受查询中的条件、查看操作符，然后判断是否存在可用的索引。这蕴含着 PostgreSQL 需要知道，例如，<= 和 > 操作符划分一个 B-树。PostgreSQL 使用策略来表达操作符与它们可用于扫描索引的方式之间的这些关系。

定义一组新的策略超出了本讨论的范围，但我们将解释 B-树策略如何工作，因为你要添加一个新的 B-树操作符类就需要了解它。在 pg_am 表中，amstrategies 列设置该访问方法所定义的策略数目。对 B-树来说，这个数是 5。这些策略的含义如表 17.2所示。

表 17.2. B-树策略

操作	索引号
小于	1
小于等于	2
等于	3
大于等于	4
大于	5

这个思路是：你需要把与这些策略对应的操作符添加到 pg_amop 关系中（见下文）。访问方法代码可以使用这些策略号（不管数据类型是什么）来弄清如何划分 B-树、计算选择性等等。现在还不用担心添加操作符的细节；只需理解：对 int2、int4、oid 以及 B-树可以操作的所有其他数据类型，都必须存在一组这样的操作符。

17.4. 访问方法支持例程

有时，仅靠策略信息不足以让系统弄清如何使用索引。
某些访问方法还需要额外的支持例程才能工作。例如，B-树访问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-
树访问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 查询条件中使用的操作符；它们是访问方法内部使用的管理例程。

为了在所有 PostgreSQL 访问方法之间一致地管理多样的支持例程，pg_am 包含一个名为 amsupport 的列。该列记录一个访问方法所使用的支持例程数目。对 B-树来说，这个数是一：接受两个键并根据第一个键小于、等于还是大于第二个键而返回 -1、0 或 +1 的例程。（严格地说，该例程可以返回一个负数（< 0）、零或一个非零正数（> 0）。）

pg_am 中的 amstrategies 项只是所论访问方法定义的策略数目。小于、小于等于等操作符并不出现在 pg_am 中。类似地，amsupport 只是该访问方法所需的支持例程数目。实际的例程列在别处。

顺便说一下，amorderstrategy 列表明访问方法是否支持有序扫描。零表示不支持；如果支持，amorderstrategy 就是对应排序操作符的策略例程编号。例如，B-树的 amorderstrategy = 1，即它的“小于”策略号。

17.5. 操作符类

下一个我们关心的表是 pg_opclass。这个表为给定索引访问方法所支持的每个操作符类定义操作符类名和输入数据类型。同一个类名可以用于多个不同的访问方法（例如，B-树和哈希访问方法都有名为 oid_ops 的操作符类），但每个访问方法必须在 pg_opclass 中有单独的一行。pg_opclass 行的 OID 被用作其他表中的外键，以把具体的操作符和支持例程与该操作符类关联起来。

你需要把带有你的操作符类名（例如 complex_abs_ops）的一行添加到 pg_opclass：

```
INSERT INTO pg_opclass (opcamid, opcname, opcintype, opcdefault,
 opckeytype)
    VALUES (
         (SELECT oid FROM pg_am WHERE amname = 'btree'),
         'complex_abs_ops',
         (SELECT oid FROM pg_type WHERE typname = 'complex'),
         true,
         0);

SELECT oid, *
    FROM pg_opclass
    WHERE opcname = 'complex_abs_ops';

  oid   | opcamid |     opcname     | opcintype | opcdefault |
 opckeytype
--------+---------+-----------------+-----------+------------+-------
-----
 277975 |     403 | complex_abs_ops |    277946 | t          |
   0
(1 row)
```

注意，你的 pg_opclass 行的 OID 会不同！不过不必担心。稍后我们会像这里获取类型的 OID 一样从系统中取得这个数。

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，只需把 opcdefault 设为 false 即可。opckeytype 在此不作描述；对 B-树操作符类它应当总是为零。

17.6. 创建操作符和支持例程

现在我们有了一个访问方法和一个操作符类。我们还需要一组操作符。
定义操作符的过程已在第 14 章中讨论过。对于 B-树上的 complex_abs_ops 操作符类，我们需要的操作符是：

- 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）

假设实现这些函数的代码存储在文件 *PGROOT*/src/tutorial/complex.c 中，并且我们已把它编译成 *PGROOT*/src/tutorial/complex.so。C 代码的一部分如下所示：

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

        bool
        complex_abs_eq(Complex *a, Complex *b)
        {
            double amag = Mag(a), bmag = Mag(b);
            return (amag==bmag);
        }
```

（注意，在余下的示例中我们只展示相等操作符。其他四个操作符非常类似。细节请参阅 complex.c 或 complex.source。）

我们像下面这样让 PostgreSQL 知道该函数：

```
CREATE FUNCTION complex_abs_eq(complex, complex) RETURNS boolean
    AS 'PGROOT/src/tutorial/complex'
    LANGUAGE C;
```

这里有几件重要的事情：

- 首先，注意这里正在为 complex 定义小于、小于等于、等于、大于等于和大于操作符。我们只能有一个名为（例如）= 并且两个操作数都取 complex 类型的操作符。在本例中我们没有其他用于 complex 的 = 操作符，但如果我们在构造一种实用的数据类型，我们可能希望 = 是复数的普通相等操作。在那种情况下，我们就需要为 complex_abs_eq 使用其他某个操作符名。

- 其次，尽管 PostgreSQL 能处理名称相同但输入数据类型不同的操作符，C 却只能处理给定名称的一个全局例程。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。

- 第三，我们本可以把该函数的 PostgreSQL 名称取为 abs_eq，并依靠 PostgreSQL 通过输入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。

- 最后，注意这些操作符函数返回布尔值。实际上，
  所有被定义为索引访问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。（另一方面，支持函数返回的是特定访问方法所期望的类型--在本例中就是一个有符号整数。）

文件中的最后一个例程是我们在讨论 pg_am 表的 amsupport 列时提到的“支持例程”。我们稍后会用到它。现在先忽略它。

现在我们准备好定义操作符了：

```
CREATE OPERATOR = (
     leftarg = complex, rightarg = complex,
     procedure = complex_abs_eq,
     restrict = eqsel, join = eqjoinsel
         );
```

这里的重要内容是过程名（即上面定义的 C 函数）以及限制和连接选择性函数。你应当直接使用示例中所用的选择性函数（见 complex.source）。注意，小于、等于和大于情形有不同的此类函数。必须提供这些函数，否则优化器将无法有效地使用该索引。

下一步是把这些操作符的项添加到 pg_amop 关系中。为此，我们需要刚才定义的这些操作符的 OID。我们将查找所有取两个 complex 类型操作数的操作符的名称，并挑出我们的那些：

```
SELECT o.oid AS opoid, o.oprname
    INTO TEMP TABLE complex_ops_tmp
    FROM pg_operator o, pg_type t
    WHERE o.oprleft = t.oid and o.oprright = t.oid
      and t.typname = 'complex';

 opoid  | oprname
--------+---------
 277963 | +
 277970 | <
 277971 | <=
 277972 | =
 277973 | >=
 277974 | >
(6 rows)
```

（同样，你的某些 OID 号几乎肯定会有所不同。）我们感兴趣的操作符是 OID 从 277970 到 277974 的那些。你得到的值很可能不同，你应当用它们替换下文中的值。我们将用一条 select 语句来完成这件事。

现在我们可以为新操作符类向 pg_amop 中插入项了。这些项必须把我们所需的每个操作符与正确的 B-树策略号关联起来。插入小于操作符的命令如下：

```
INSERT INTO pg_amop (amopclaid, amopstrategy, amopreqcheck, amopopr)
    SELECT opcl.oid, 1, false, c.opoid
        FROM pg_opclass opcl, complex_ops_tmp c
        WHERE
            opcamid = (SELECT oid FROM pg_am WHERE amname = 'btree')
 AND
            opcname = 'complex_abs_ops' AND
            c.oprname = '<';
```

然后对其他操作符照此办理，替换上文第二行中的 1 和最后一行中的 <。注意顺序：“小于”是 1，“小于等于”是 2，“等于”是 3，“大于等于”是 4，“大于”是 5。

字段 amopreqcheck 在此不作讨论；对 B-树操作符它应当总是 false。

最后一步是注册此前在讨论 pg_am 时描述过的“支持例程”。该支持例程的 OID 存储在 pg_amproc 表中，以操作符类 OID 和支持例程编号为键。

首先，我们需要在 PostgreSQL 中注册该函数（回想一下，我们把实现该例程的 C 代码放在了实现操作符例程的那个文件的底部）：

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'PGROOT/src/tutorial/complex'
    LANGUAGE C;

SELECT oid, proname FROM pg_proc
    WHERE proname = 'complex_abs_cmp';

  oid   |     proname
--------+-----------------
 277997 | complex_abs_cmp
(1 row)
```

（同样，你的 OID 号很可能会有所不同。）

我们可以像下面这样添加新行：

```
INSERT INTO pg_amproc (amopclaid, amprocnum, amproc)
    SELECT opcl.oid, 1, p.oid
        FROM pg_opclass opcl, pg_proc p
        WHERE
            opcamid = (SELECT oid FROM pg_am WHERE amname = 'btree')
 AND
            opcname = 'complex_abs_ops' AND
            p.proname = 'complex_abs_cmp';
```

这样就完成了！（呼。）现在应该可以在 `complex` 列上创建并使用 B-树索引了。

第 18 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元组所占的比例）。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，而处理一个索引元组的代价通常取为 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元组的代价。

“启动代价”是总扫描代价中在可以开始取第一个元组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元组的估计比例。对于有损索引，这通常会高于实际满足给定限定条件的元组所占的比例。

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元组的代价估计。

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元组所占的比例。
   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,
                                              lfirsti(rel->relids));
   ```

2. 估计扫描期间将访问的索引元组数。对许多索引类型来说，它等于 indexSelectivity 乘以索引中的元组数，但也可能更多。（注意，索引的页数和元组数可以从 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 tuple.
     * All the costs are assumed to be paid incrementally during
 the scan.
     */
    *indexStartupCost = 0;
    *indexTotalCost = numIndexPages +
        (cpu_index_tuple_cost + cost_qual_eval(indexQuals)) * 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：

```
prorettype = 0
pronargs = 8
proargtypes = 0 0 0 0 0 0 0 0
```

由于所有参数都没有 pg_type 中已知的类型，我们对全部参数使用零（"opaque"）。

第 19 章 GiST 索引

Gene Selkov

关于 GIST 的信息在 http://GiST.CS.Berkeley.EDU:8000/gist/，关于不同索引和排序方案的更多内容在 http://s2k-ftp.CS.Berkeley.EDU:8000/personal/jmh/。另外还有一些有意思的阅读材料在 http://epoch.cs.berkeley.edu:8000/ 和 http://www.sai.msu.su/~megera/postgres/gist/。

> **作者**
>
> 这段摘录来自 Eugene Selkov, Jr.（<selkovjr@mcs.anl.gov>）发送的一封电子邮件，其中包含关于 GiST 的有用信息。希望我们将来能学到更多并更新这些信息。- thomas 1998-03-01

好吧，我不能说完全理解了其中的原理，但我（几乎）成功地把 GiST 示例移植到了 linux。GiST 访问方法已经在 postgres 树中（`src/backend/access/gist`）。

伯克利的示例[1]附带这些方法的概述，并演示了二维矩形、多边形、
整数区间和文本的空间索引机制（另见伯克利的 GiST[2]）。在矩形示例中，使用 GiST 索引本应看到性能提升；它对我来说确实有效，但我没有规模足够大的矩形集合来验证这一点。
其他示例也都有效，只有多边形除外：执行

```
test=> CREATE INDEX pix ON polytmp
test-> USING GIST (p:box gist_poly_ops) WITH (ISLOSSY);
ERROR:  cannot open pix

(PostgreSQL 6.3              Sun Feb  1 14:57:30 EST 1998)
```

我没搞懂这条错误消息的含义；它似乎是我们更应该去问开发者的问题（另见下面的注 4）。我这里想建议的是，你们当中的 linux 用户（linux==gcc?）去取上面提到的原始源码并应用我的补丁（见附件），然后告诉我们你们的感受。我觉得很酷，但周围有这么多能人，
我不想把它扣着不放。

关于这些源码的几点说明：

1. 我没能用上原始的（HP-UX）Makefile，于是把古老的 postgres95 教程里的 Makefile 改了改来干活。我试着让它通用一些，但我写 Makefile 的水平很差——只是做了些照猫画虎的活儿。抱歉，不过我想它现在比原来的 makefile 稍微可移植一点了。

2. 我直接在 pgsql/src 之下构建了示例源码（只是把 tar 文件解压在那里）。前面提到的 Makefile 假定它位于 pgsql/src 下一级（在我们的例子中即 pgsql/src/pggist）。

3. 我对 *.c 文件的改动都只是关于 #include、函数原型和类型转换。除此之外，我只是扔掉了一堆未用的变量，并加了几个括号来取悦 gcc。希望我没有搞砸太多 :)

4. polyproc.sql 中有一条注释：

```
-- -- there's a memory leak in rtree poly_ops!!
-- -- CREATE INDEX pix2 ON polytmp USING RTREE (p poly_ops);
```

[1] ftp://s2k-ftp.cs.berkeley.edu/pub/gist/pggist/pggist.tgz
[2] http://gist.cs.berkeley.edu:8000/gist/

收到！！我以为它可能与几个版本之前的 PostgreSQL 有关，于是试着执行了该查询。我的系统疯了，大约十分钟后我不得不干掉了 postmaster。

我会继续研究一阵子 GiST，但我也希望看到更多 R 树用法的示例。

第 20 章 触发器

PostgreSQL 有多种服务器端函数接口。服务器端函数可以用 SQL、PLPGSQL、TCL 或 C 编写。除 SQL 外，触发器函数可以用这些语言中的任何一种编写。注意，当前版本不支持语句（STATEMENT）级触发器事件。目前可以把 BEFORE 或 AFTER 指定在一个元组的 INSERT、DELETE 或 UPDATE 上作为触发器事件。

20.1. 触发器创建

一旦触发器事件发生，触发器管理器（由执行器调用）就会建立一个 TriggerData 信息结构（下文描述），并调用触发器函数来处理该事件。

必须先定义触发器函数，然后才能创建触发器，它声明为不接受参数并返回 opaque 的函数。如果函数用 C 编写，它必须使用“版本 1”函数管理器接口。

创建触发器的语法如下：

```
CREATE TRIGGER trigger [ BEFORE | AFTER ] [ INSERT | DELETE | UPDATE [
 OR ... ] ]
    ON relation FOR EACH [ ROW | STATEMENT ]
    EXECUTE PROCEDURE procedure
     (args);
```

其中各个参数为：

trigger

触发器的名称在你将来需要删除该触发器时使用。它被用作 DROP TRIGGER 命令的一个参数。

BEFORE
AFTER

决定函数是在事件之前还是之后被调用。

INSERT
DELETE
UPDATE

命令的下一个元素决定哪些事件会触发该函数。可以用 OR 分隔指定多个事件。

relation

关系名指出该事件作用于哪个表。

ROW
STATEMENT

FOR EACH 子句决定触发器是针对每个受影响的行触发，还是在整条语句完成之前（或之后）触发。目前只支持 ROW 的情形。

procedure

过程名是要被调用的函数。

args

通过 TriggerData 结构传给函数的参数。向函数传递参数的目的，是为了让需求相似的不同触发器能够调用同一个函数。

另外，*procedure* 也可以用于触发不同的关系（这类函数被称为通用触发器函数）。

作为同时使用上述两个特性的例子，可以有一个通用函数，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。这样就可以在 INSERT 事件上编写触发器，例如自动跟踪某个事务表中记录的创建。如果用在 UPDATE 事件上，它还可以作为一个“最近更新”函数使用。

触发器函数向调用它的执行器返回 HeapTuple。对于在 INSERT、DELETE 或 UPDATE 操作之后触发的触发器，该返回值会被忽略，但它允许 BEFORE 触发器：

- 返回 NULL，以跳过对当前元组的操作（这样该元组就不会被插入/更新/删除）。
- 返回一个指向另一个元组的指针（仅 INSERT 和 UPDATE），该元组将被插入（如果是 UPDATE，则作为被更新元组的新版本），以取代原来的元组。

注意，CREATE TRIGGER 处理器不执行任何初始化。这一点将来会改变。另外，如果为同一关系上的同一事件定义了多个触发器，触发器的触发顺序是不可预测的。这一点将来也可能会改变。

如果触发器函数执行 SQL 查询（使用 SPI），那么这些查询可能会再次触发触发器。这就是所谓的级联触发器。对级联层数没有明确的限制。

如果一个触发器由 INSERT 触发，而又向同一个关系插入了一个新元组，那么该触发器会再次被触发。目前，对于这类情形没有提供任何同步（等）机制，但这一点将来可能改变。目前，回归测试中的函数 funny_dup17() 使用了一些技巧来停止对自身的递归（级联）……

20.2. 与触发器管理器的交互

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函数语言，那么这些细节会替你处理。

> **注意**
>
> 这里描述的接口适用于 PostgreSQL 7.1 及之后的版本。更早的版本通过一个全局变量 CurrentTriggerData 传递 TriggerData 指针。

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 TRUE，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。该函数不得修改 TriggerData 结构或它指向的任何数据。

struct TriggerData 定义在 src/include/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

如果这是一个触发器事件，则总是 `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)

如果触发器在之前（BEFORE）触发则返回 TRUE。

TRIGGER_FIRED_AFTER(tg_event)

如果触发器在之后（AFTER）触发则返回 TRUE。

TRIGGER_FIRED_FOR_ROW(event)

如果触发器针对行（ROW）级事件触发则返回 TRUE。

TRIGGER_FIRED_FOR_STATEMENT(event)

如果触发器针对语句（STATEMENT）级事件触发则返回 TRUE。

TRIGGER_FIRED_BY_INSERT(event)

如果触发器由 INSERT 引发则返回 TRUE。

TRIGGER_FIRED_BY_DELETE(event)

如果触发器由 DELETE 引发则返回 TRUE。

TRIGGER_FIRED_BY_UPDATE(event)

如果触发器由 UPDATE 引发则返回 TRUE。

tg_relation

一个指针，指向描述被触发关系的结构体。关于这个结构体的细节请参看 src/include/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。它不是 char*，而是 NameData。如果你需要名称的一个副本，可以使用 SPI_getrelname(tg_relation) 来获得 char*）。

tg_trigtuple

一个指针，指向引发该触发器的元组。这就是正在被插入（INSERT）、删除（DELETE）或更新（UPDATE）的元组。如果是 INSERT/DELETE，那么当你不想（INSERT 的情形下）用另一个元组替换该元组或跳过该操作时，就应当把它返回给执行器。

tg_newtuple

一个指针，指向元组的新版本（如果 UPDATE）；如果是 INSERT 或 DELETE 则为 NULL。当事件是 UPDATE 而你不想用另一个元组替换该元组或跳过该操作时，就应当把它返回给执行器。

tg_trigger

一个指针，指向 Trigger 结构，它定义在 src/include/utils/rel.h 中：

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    bool        tgenabled;
    bool        tgisconstraint;
    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。

20.3. 数据更改的可见性

PostgreSQL 的数据更改可见性规则：在一次查询执行期间，该查询自身（通过 SQL 函数、SPI 函数、触发器）所做的数据更改对查询扫描是不可见的。例如，在查询

```
INSERT INTO a SELECT * FROM a;
```

中，被插入的元组对 SELECT 扫描不可见。其效果是在数据库表内部复制该表自身（当然要受唯一索引规则的约束），而不会递归。

但请记住 SPI 文档中关于可见性的这段说明：

> 查询 Q 所做的更改，对所有在查询 Q 之后启动的查询都可见，无论这些查询是在 Q 内部（即 Q 执行期间）启动，还是在 Q 完成之后启动。

这对触发器同样成立，因此，虽然正在被插入的元组（tg_trigtuple）对 BEFORE 触发器中的查询不可见，但这个元组（刚刚插入的）对 AFTER 触发器中的查询可见，也对在此之后触发的 BEFORE/AFTER 触发器中的查询可见！

20.4. 示例

更多更复杂的示例见 `src/test/regress/regress.c` 和 `contrib/spi`。

下面是一个非常简单的触发器用法示例。函数 trigf 报告被触发关系 ttest 中的元组数，并且当查询试图向 x 插入 NULL 时跳过该操作（也就是说，它起到一个 NOT NULL 约束的作用，但不会中止事务）。

```
#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 trigdata is pointing at what I expect */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not fired 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 NULLs ? */
    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 tuples in relation */
    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 as of PG 7.2, so be careful to convert */
```

```
    i = (int) DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                          SPI_tuptable->tupdesc,
                                          1,
                                          &isnull));

    elog (NOTICE, "trigf (fired %s): there are %d tuples in ttest",
 when, i);

    SPI_finish();

    if (checknull)
    {
        (void) SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

现在，编译并创建触发器函数：

```
CREATE FUNCTION trigf () RETURNS OPAQUE AS
'...path_to_so' LANGUAGE 'C';

CREATE TABLE ttest (x int4);

vac=> CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON
 ttest
FOR EACH ROW EXECUTE PROCEDURE trigf();
CREATE
vac=> CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
FOR EACH ROW EXECUTE PROCEDURE trigf();
CREATE
vac=> INSERT INTO ttest VALUES (NULL);
NOTICE:trigf (fired before): there are 0 tuples in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

vac=> SELECT * FROM ttest;
 x
-
(0 rows)

vac=> INSERT INTO ttest VALUES (1);
NOTICE:trigf (fired before): there are 0 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
                                       ^^^^^^^^
                              remember what we said about visibility.
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
-
```

```
 1
(1 row)

vac=> INSERT INTO ttest SELECT x * 2 FROM ttest;
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
                                       ^^^^^^^^
                               remember what we said about visibility.
INSERT 167794 1
vac=> SELECT * FROM ttest;
 x
-
 1
 2
(2 rows)

vac=> UPDATE ttest SET x = null WHERE x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
UPDATE 0
vac=> UPDATE ttest SET x = 4 WHERE x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
-
 1
 4
(2 rows)

vac=> DELETE FROM ttest;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 0 tuples in ttest
                                       ^^^^^^^^
                               remember what we said about visibility.
DELETE 2
vac=> SELECT * FROM ttest;
 x
-
(0 rows)
```

第 21 章 服务器编程接口

Vadim Mikheev

服务器编程接口（SPI）使用户能够在用户定义的 C 函数中运行 SQL 查询。

> **注意**
>
> 可用的过程语言（PL）提供了另一种构建可执行查询的函数的方法。

实际上，SPI 只是一组简化对解析器、规划器、优化器和执行器访问的本地接口函数。SPI 还做一部分内存管理工作。

为避免误解，我们用函数指 SPI 接口函数，而用过程指使用 SPI 的用户定义 C 函数。

使用 SPI 的过程由执行器调用。这些 SPI 调用又会递归地调用执行器来运行查询。当执行器被递归调用时，它自己也可能调用过程，而这些过程又可能发起 SPI 调用。

注意，如果在执行来自过程的查询期间事务被中止，控制将不会返回到你的过程。相反，
所有工作都会被回滚，服务器会等待客户端的下一条命令。
这一点在未来的版本中可能会被改变。

一个相关的限制是无法执行 BEGIN、END 和 ABORT（事务控制语句）。这一点将来也会被改变。

成功时，SPI 函数返回非负结果（或者通过整型返回值返回，或者如下文所述存放在全局变量 SPI_result 中）。出错时，将返回负值或 NULL。

21.1. 接口函数

SPI_connect

SPI_connect — 将你的过程连接到 SPI 管理器。

大纲

```
int SPI_connect(void)
```

输入

无

输出

int

返回状态

→ SPI_OK_CONNECT

已连接时

→ SPI_ERROR_CONNECT

未连接时

描述

SPI_connect 打开从过程调用到 SPI 管理器的连接。如果需要执行查询，就必须调用此函数。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 → SPI_ERROR_CONNECT 错误。注意，当某个直接调用了 SPI_connect 的过程又直接调用另一个自身调用 SPI_connect 的过程时，就可能发生这种情况。虽然当 SPI 查询调用另一个使用 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。

用法

算法

SPI_connect 执行以下操作：初始化用于查询执行和内存管理的 SPI 内部结构。

SPI_finish

SPI_finish — 将你的过程与 SPI 管理器断开连接。

大纲

```
SPI_finish(void)
```

输入

无

输出

int

→ SPI_OK_FINISH 已正确断开连接时
→ SPI_ERROR_UNCONNECTED 从未连接的过程中调用时

描述

SPI_finish 关闭到 SPI 管理器的现有连接。在过程的本次调用期间完成所需的 SPI 操作之后，必须调用此函数。

如果在当前没有有效连接的情况下调用 SPI_finish，可能会得到错误返回 → SPI_ERROR_UNCONNECTED。这并不是什么根本性的问题；它只表示 SPI 管理器没有做任何事情。

用法

已连接的过程必须以调用 SPI_finish 作为最后一步，否则可能得到不可预测的结果！不过，如果事务是通过 elog(ERROR) 中止的，则不必担心这一点。在这种情况下，SPI 会自行清理。

算法

SPI_finish 执行以下操作：将你的过程与 SPI 管理器断开连接，并释放该过程自 SPI_connect 以来通过 palloc 所做的全部内存分配。这些分配不能再使用了！参见内存管理一节。

SPI_exec

SPI_exec — 创建一个执行计划（解析器+规划器+优化器）并执行查询。

大纲

```
SPI_exec(query, tcount)
```

输入

char *query

包含查询计划的字符串

int tcount

要返回的最大元组数

输出

int

→ SPI_ERROR_UNCONNECTED 从未连接的过程中调用时
→ SPI_ERROR_ARGUMENT query 为 NULL 或 tcount < 0 时。
→ SPI_ERROR_UNCONNECTED 过程未连接时。
→ SPI_ERROR_COPY 执行 COPY TO/FROM stdin 时。
→ SPI_ERROR_CURSOR 执行 DECLARE/CLOSE CURSOR、FETCH 时。
→ SPI_ERROR_TRANSACTION 执行 BEGIN/ABORT/END 时。
→ SPI_ERROR_OPUNKNOWN 查询类型未知时（这不应发生）。

如果查询执行成功，则会返回下列（非负）值之一：

→ SPI_OK_UTILITY 执行了某个工具命令（例如 CREATE TABLE ...）时
→ SPI_OK_SELECT 执行了 SELECT（但不是 SELECT ... INTO!）时
→ SPI_OK_SELINTO 执行了 SELECT ... INTO 时
→ SPI_OK_INSERT 执行了 INSERT（或 INSERT ... SELECT）时
→ SPI_OK_DELETE 执行了 DELETE 时
→ SPI_OK_UPDATE 执行了 UPDATE 时

描述

SPI_exec 创建一个执行计划（解析器+规划器+优化器），并针对 tcount 个元组执行该查询。

用法

此函数只应从已连接的过程中调用。如果 tcount 为零，则会对查询扫描返回的所有元组执行该查询。使用 tcount > 0 可以限制该查询将要执行的元组数（很像一个 LIMIT 子句）。例如，

```
SPI_exec ("INSERT INTO tab SELECT * FROM tab", 5);
```

最多只允许向表中插入 5 个元组。如果查询执行成功，则会返回一个非负值。

> **注意**
>
> 你可以在一个字符串中传递多条查询，查询字符串也可能被 RULE 改写。SPI_exec 返回最后执行的那条查询的结果。

（最后一条）查询实际执行所处理的元组数，会通过全局变量 SPI_processed 返回（返回值不是 → SPI_OK_UTILITY 时）。如果返回了 → SPI_OK_SELECT 且 SPI_processed > 0，则可以使用全局指针 SPITupleTable *SPI_tuptable 访问结果元组。

SPI_exec 可能返回下列（负）值之一：

→ SPI_ERROR_ARGUMENT query 为 NULL 或 *tcount* < 0 时。
→ SPI_ERROR_UNCONNECTED 过程未连接时。
→ SPI_ERROR_COPY 执行 COPY TO/FROM stdin 时。
→ SPI_ERROR_CURSOR 执行 DECLARE/CLOSE CURSOR、FETCH 时。
→ SPI_ERROR_TRANSACTION 执行 BEGIN/ABORT/END 时。
→ SPI_ERROR_OPUNKNOWN 查询类型未知时（这不应发生）。

结构

如果返回了 → SPI_OK_SELECT 且 SPI_processed > 0，则可以使用全局指针 SPITupleTable *SPI_tuptable 访问选出的元组。

结构 SPITupleTable 定义在 spi.h 中：

```
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;        /* 元组描述符 */
    HeapTuple  *vals;           /* 元组 */
} SPITupleTable;
```

vals 是一个指向元组的指针数组（有效项数由 SPI_processed 给出）。TupleDesc tupdesc 是一个元组描述符，可以传给处理元组的 SPI 函数。tuptabcxt、alloced 和 free 是内部字段，不供 SPI 调用者使用。

> **注意**
>
> 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某次 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_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。

SPI_prepare

SPI_prepare — 为查询准备一个计划，但暂不执行

大纲

```
SPI_prepare(query, nargs, argtypes)
```

输入

query

查询字符串

nargs

输入参数的数量（$1 ... $nargs —— 与 SQL 函数中相同）

argtypes

指向输入参数类型的类型 OID 数组的指针

输出

void *

指向执行计划（解析器+规划器+优化器）的指针

描述

`SPI_prepare` 创建并返回一个执行计划（解析器+规划器+优化器），但不执行该查询。只应从已连接的过程中调用。

用法

当同一条或相似的查询需要反复执行时，只做一次查询规划可能是有利的。`SPI_prepare` 把查询字符串转换成一个执行计划，之后可以反复传给 `SPI_execp`。

预备查询可以通过在普通查询原本应写常量的位置写入参数（$1、$2 等）而得到泛化。这些参数的值会在调用 `SPI_execp` 时指定。这样，预备查询就能适用于比不带参数时更广泛的场景。

> **注意**
>
> 不过，这也有一个不利之处：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简单查询下更差。

如果查询使用了参数，那么在调用 `SPI_prepare` 时必须指定它们的数量和数据类型。

`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 `SPI_saveplan` 把计划保存得更久。

成功时会返回一个非空指针。否则，将得到 NULL 计划。无论成功还是出错，SPI_result 都会被设成与 SPI_exec 返回值类似的值；但如果 query 为 NULL，或 nargs < 0，或 nargs > 0 且 argtypes 为 NULL，则会将其设为 → `SPI_ERROR_ARGUMENT`。

SPI_execp

SPI_execp — 执行来自 SPI_prepare 的计划

大纲

```
SPI_execp(plan,
values,
nulls,
tcount)
```

输入

void *plan

执行计划

Datum *values

实际参数值

char *nulls

描述哪些参数为 NULL 的数组

n 表示 NULL（忽略 values[] 中的对应项）
空格表示非 NULL（values[] 中的对应项有效）

int tcount

该计划将要执行的元组数

输出

int

返回与 SPI_exec 相同的值，以及

→ SPI_ERROR_ARGUMENT*plan* 为 NULL 或 *tcount* < 0 时
→ SPI_ERROR_PARAM*values* 为 NULL，而 *plan* 在准备时带有参数时。

SPI_tuptable

成功时，其初始化方式与 SPI_exec 中相同

SPI_processed

成功时，其初始化方式与 SPI_exec 中相同

描述

SPI_execp 执行由 SPI_prepare 准备的计划。*tcount* 的含义与 SPI_exec 中相同。

用法

如果 *nulls* 为 NULL，则 SPI_execp 假定所有参数（如果有的话）都非 NULL。

注意

如果预备计划所引用的对象（关系、函数等）
在会话期间被删除（由你的后端或另一个进程删除），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

```
SPI_cursor_open(name,
plan,
values,
nulls)
```

输入

char *_name_

portal 的名称，或为 NULL 以让系统选择名称

void *_plan_

执行计划

Datum *_values_

实际参数值

char *_nulls_

描述哪些参数为 NULL 的数组

`n` 表示 NULL（忽略 values[] 中的对应项）
空格表示非 NULL（values[] 中的对应项有效）

输出

Portal

指向包含该游标的 Portal 的指针，出错时为 NULL

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。

用法

如果 _nulls_ 为 NULL，则 `SPI_cursor_open` 假定所有参数（如果有的话）都非 NULL。

SPI_cursor_find

SPI_cursor_find — 按名称查找现有的游标（Portal）

大纲

```
SPI_cursor_find(name)
```

输入

char *name

portal 的名称

输出

Portal

指向具有给定名称的 Portal 的指针，找不到时为 NULL

描述

`SPI_cursor_find` 按名称查找一个已存在的 Portal。这主要用于解析由其他函数以文本形式返回的游标名称。

SPI_cursor_fetch

SPI_cursor_fetch — 从一个游标中取出一些行

大纲

```
SPI_cursor_fetch(portal,
forward,
count)
```

输入

Portal *portal*

包含游标的 portal

bool *forward*

为 true 时向前取，为 false 时向后取

int *count*

要取出的最大行数

输出

SPI_tuptable

成功时，其初始化方式与 `SPI_exec` 中相同

SPI_processed

成功时，其初始化方式与 `SPI_exec` 中相同

描述

`SPI_cursor_fetch` 从一个游标中取出一些（更多）行。它等价于 SQL 命令 `FETCH`。

SPI_cursor_move

SPI_cursor_move — 移动游标

大纲

```
SPI_cursor_move(portal,
forward,
count)
```

输入

Portal *portal*

包含游标的 portal

bool *forward*

为 true 时向前移动，为 false 时向后移动

int *count*

要移动的最大行数

输出

无

描述

`SPI_cursor_move` 在游标中跳过一些行。它等价于 SQL 命令 `MOVE`。

SPI_cursor_close

SPI_cursor_close — 关闭一个游标

大纲

```
SPI_cursor_close(portal)
```

输入

Portal *portal*

包含游标的 portal

输出

无

描述

`SPI_cursor_close` 关闭一个先前创建的游标，并释放其 Portal 存储。

用法

所有打开的游标都会在事务结束时被隐式关闭。只有当希望更早释放资源时，才需要调用 `SPI_cursor_close`。

SPI_saveplan

SPI_saveplan — 保存传入的计划

大纲

```
SPI_saveplan(plan)
```

输入

void **query*

传入的计划

输出

void *

执行计划的位置。不成功时为 NULL。

SPI_result

→ `SPI_ERROR_ARGUMENT` plan 为 NULL 时
→ `SPI_ERROR_UNCONNECTED` 过程未连接时

描述

`SPI_saveplan` 把由 `SPI_prepare` 准备的计划存储在安全内存中，使其不会被 `SPI_finish` 或事务管理器释放。

在当前版本的 PostgreSQL 中，还无法把预备计划存储在系统目录中并在执行时从那里取出。这一点将在未来的版本中实现。作为替代，可以在当前会话中该过程的后续调用里复用预备计划。使用 `SPI_execp` 执行这个保存的计划。

用法

`SPI_saveplan` 把传入的计划（由 `SPI_prepare` 准备）保存在受保护的内存中，使其不会被 `SPI_finish` 和事务管理器释放，并返回指向该已保存计划的指针。可以把返回的指针保存在局部变量中。无论是准备计划时，还是在 SPI_execp 中使用已准备好的计划时（见下文），都请始终检查该指针是否为 NULL。

> **注意**
>
> 如果预备计划所引用的对象（关系、函数等）
> 在会话期间被删除（由你的后端或另一个进程删除），那么对该计划执行 `SPI_execp` 的结果将是不可预知的。

21.2. 接口支持函数

这里介绍的函数为从 `SPI_exec` 及其他 SPI 接口函数返回的元组集中提取信息提供了便捷接口。

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。

SPI_fnumber

SPI_fnumber — 查找指定属性名称的属性编号

大纲

```
SPI_fnumber(tupdesc, fname)
```

输入

TupleDesc *tupdesc*

输入元组描述

char * *fname*

字段名

输出

int

属性编号

属性的有效基于 1 的索引编号
→ SPI_ERROR_NOATTRIBUTE 找不到指定名称的属性时

描述

SPI_fnumber 返回 fname 中指定名称的属性的属性编号。

用法

属性编号从 1 开始。

如果给定的 fname 指的是一个系统属性（例如 oid），则会返回相应的负属性编号。调用者应当注意通过与 → SPI_ERROR_NOATTRIBUTE 的精确相等来判断是否出错；除非希望拒绝系统属性，否则用结果 <= 0 来判断是不正确的。

SPI_fname

SPI_fname — 查找指定属性编号的属性名称

大纲

```
SPI_fname(tupdesc, fnumber)
```

输入

TupleDesc *tupdesc*

输入元组描述

int *fnumber*

属性编号

输出

char *

属性名称

fnumber 超出范围时为 NULL
出错时 SPI_result 被设置为 → `SPI_ERROR_NOATTRIBUTE`

描述

`SPI_fname` 返回指定属性的属性名称。

用法

属性编号从 1 开始。

算法

返回一个新分配的属性名称副本。（用完之后可用 pfree() 释放该副本。）

SPI_getvalue

SPI_getvalue — 返回指定属性的字符串值

大纲

```
SPI_getvalue(tuple, tupdesc, fnumber)
```

输入

HeapTuple *tuple*

要检查的输入元组

TupleDesc *tupdesc*

输入元组描述

int *fnumber*

属性编号

输出

char *

属性值，以下情况返回 NULL：

属性为 NULL
fnumber 超出范围（SPI_result 被设置为 → SPI_ERROR_NOATTRIBUTE）
没有可用的输出函数（SPI_result 被设置为 → SPI_ERROR_NOOUTFUNC）

描述

SPI_getvalue 返回指定属性值的外部（字符串）表示形式。

用法

属性编号从 1 开始。

算法

结果以 palloc 分配的字符串返回。（用完之后可用 pfree() 释放该字符串。）

SPI_getbinval

SPI_getbinval — 返回指定属性的二进制值

大纲

```
SPI_getbinval(tuple, tupdesc, fnumber, isnull)
```

输入

HeapTuple *tuple*

要检查的输入元组

TupleDesc *tupdesc*

输入元组描述

int *fnumber*

属性编号

输出

Datum

属性的二元值

bool * *isnull*

属性值是否为空的标志

SPI_result

→ SPI_ERROR_NOATTRIBUTE

描述

SPI_getbinval 以内部形式（作为一个 Datum）返回指定属性的值。

用法

属性编号从 1 开始。

算法

不为该 datum 分配新的空间。对于传引用的数据类型，该 Datum 将是指向给定元组内部的指针。

SPI_gettype

SPI_gettype — 返回指定属性的类型名称

大纲

```
SPI_gettype(tupdesc, fnumber)
```

输入

TupleDesc *tupdesc*

输入元组描述

int *fnumber*

属性编号

输出

char *

指定属性编号的类型名称

SPI_result

→ SPI_ERROR_NOATTRIBUTE

描述

SPI_gettype 返回指定属性类型名称的一个副本，出错时返回 NULL。

用法

属性编号从 1 开始。

算法

返回一个新分配的类型名称副本。（用完之后可用 pfree() 释放该副本。）

SPI_gettypeid

SPI_gettypeid — 返回指定属性的类型 OID

大纲

```
SPI_gettypeid(tupdesc, fnumber)
```

输入

TupleDesc *tupdesc*

输入元组描述

int *fnumber*

属性编号

输出

OID

指定属性编号的类型 OID

SPI_result

→ SPI_ERROR_NOATTRIBUTE

描述

`SPI_gettypeid` 返回指定属性的类型 OID。

用法

属性编号从 1 开始。

SPI_getrelname

SPI_getrelname — 返回指定关系的名称

大纲

```
SPI_getrelname(rel)
```

输入

Relation *rel*

输入关系

输出

char *

指定关系的名称

描述

`SPI_getrelname` 返回指定关系的名称。

算法

返回一个新分配的关系名称副本。（用完之后可用 pfree() 释放该副本。）

21.3. 内存管理

PostgreSQL 在内存上下文中分配内存，这为管理那些在许多不同位置创建、且需要存活时间各不相同的内存分配提供了便捷方法。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` 创建一个新的内存上下文并将其设为当前上下文。`SPI_finish` 恢复先前的当前内存上下文，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` 来分配该返回对象，至少在连接到 SPI 的期间不能这么做。如果这样做，该对象会在 `SPI_finish` 期间被释放，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` 从“上层执行器”内存中分配空间 —— 也就是调用 `SPI_connect` 时的当前内存上下文，这正是过程返回值最合适的上下文。

如果在未连接到 SPI 时调用，`SPI_palloc` 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，因此该过程通过 `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` 之后，当前上下文是由 `SPI_connect` 创建的过程私有上下文。通过 `palloc/repalloc` 或 SPI 辅助函数所做的所有分配（`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，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后端函数（palloc 等）相同。

SPI_copytuple

SPI_copytuple — 在上层执行器上下文中创建元组的副本

大纲

```
SPI_copytuple(tuple)
```

输入

HeapTuple *tuple*

要复制的输入元组

输出

HeapTuple

复制得到的元组

→ 非 NULL*tuple* 不为 NULL 且复制成功时
→ NULL 仅当 *tuple* 为 NULL 时

描述

SPI_copytuple 在上层执行器上下文中创建元组的一个副本。

用法

TBD

SPI_copytupledesc

SPI_copytupledesc — 在上层执行器上下文中创建元组描述符的副本

大纲

```
SPI_copytupledesc(tupdesc)
```

输入

TupleDesc *tupdesc*

要复制的输入元组描述符

输出

TupleDesc

复制得到的元组描述符

→ 非 NULL*tupdesc* 不为 NULL 且复制成功时
→ NULL 仅当 *tupdesc* 为 NULL 时

描述

SPI_copytupledesc 在上层执行器上下文中创建 tupdesc 的一个副本。

用法

TBD

SPI_copytupleintoslot

SPI_copytupleintoslot — 在上层执行器上下文中创建元组及其描述符的副本

大纲

```
SPI_copytupleintoslot(tuple, tupdesc)
```

输入

HeapTuple *tuple*

要复制的输入元组

TupleDesc *tupdesc*

要复制的输入元组描述符

输出

TupleTableSlot *

包含复制得到的元组及其描述符的元组槽

→ 非 NULL*tuple* 和 *tupdesc* 都不为 NULL 且复制成功时
→ NULL 仅当 *tuple* 或 *tupdesc* 为 NULL 时

描述

SPI_copytupleintoslot 在上层执行器上下文中创建元组的一个副本，并以填充好的 TupleTableSlot 的形式返回它。

用法

TBD

SPI_modifytuple

SPI_modifytuple — 通过替换给定元组中的选定字段来创建一个元组

大纲

```
SPI_modifytuple(rel, tuple, nattrs, attnum, Values, Nulls)
```

输入

Relation *rel*

仅用作元组的元组描述符来源。（传关系而不是元组描述符是一种缺陷设计。）

HeapTuple *tuple*

要修改的输入元组

int *nattrs*

attnum 数组中属性编号的数量

int * *attnum*

要被修改的属性的编号组成的数组

Datum * *Values*

指定属性的新值

char * *Nulls*

哪些新值为 NULL（如果有的话）

输出

HeapTuple

带修改内容的新元组

→ 非 NULL*tuple* 不为 NULL 且修改成功时
→ NULL 仅当 *tuple* 为 NULL 时

SPI_result

→ SPI_ERROR_ARGUMENT rel 为 NULL，或 tuple 为 NULL，或 natts <= 0，或 attnum 为 NULL，或 Values 为 NULL 时。
→ SPI_ERROR_NOATTRIBUTE attnum 中存在无效的属性编号时（attnum <= 0，或大于元组中的属性数量）

描述

SPI_modifytuple 为选定的属性替换新值，并在其他位置复制原元组的属性，从而创建一个新元组。输入元组不会被修改。

用法

成功时，返回指向新元组的指针。新元组在上层执行器上下文中分配。

SPI_palloc

SPI_palloc — 在上层执行器上下文中分配内存

大纲

```
SPI_palloc(size)
```

输入

Size *size*

要分配的存储空间的字节大小

输出

void *

指定大小的新存储空间

描述

`SPI_palloc` 在上层执行器上下文中分配内存。

用法

TBD

SPI_repalloc

SPI_repalloc — 在上层执行器上下文中重新分配内存

大纲

```
SPI_repalloc(pointer, size)
```

输入

void * *pointer*

指向现有存储的指针

Size *size*

要分配的存储空间的字节大小

输出

void *

指定大小的新存储空间，内容从现有区域复制而来

描述

`SPI_repalloc` 在上层执行器上下文中重新分配内存。

用法

此函数与普通的 `repalloc` 已无区别。保留它只是为了向后兼容现有代码。

SPI_pfree

SPI_pfree — 在上层执行器上下文中释放内存

大纲

```
SPI_pfree(pointer)
```

输入

void * *pointer*

指向现有存储的指针

输出

无

描述

`SPI_pfree` 在上层执行器上下文中释放内存。

用法

此函数与普通的 `pfree` 已无区别。保留它只是为了向后兼容现有代码。

SPI_freetuple

SPI_freetuple — 释放先前在上层执行器上下文中分配的元组

大纲

```
SPI_freetuple(pointer)
```

输入

HeapTuple *pointer*

指向已分配元组的指针

输出

无

描述

`SPI_freetuple` 释放先前在上层执行器上下文中分配的一个元组。

用法

此函数与普通的 `heap_freetuple` 已无区别。保留它只是为了向后兼容现有代码。

SPI_freetuptable

SPI_freetuptable — 释放由 `SPI_exec` 或类似函数创建的元组集

大纲

```
SPI_freetuptable(tuptable)
```

输入

SPITupleTable * *tuptable*

指向元组表的指针

输出

无

描述

`SPI_freetuptable` 释放由先前的 SPI 查询函数（例如 `SPI_exec`）创建的元组集。

用法

如果 SPI 过程需要执行多条查询，并且不希望先前查询的结果一直保留到过程结束，此函数就很有用。注意，任何未释放的元组集最终都会在 `SPI_finish` 时被释放。

SPI_freeplan

SPI_freeplan — 释放一个先前保存的计划

大纲

```
SPI_freeplan(plan)
```

输入

void **plan*

传入的计划

输出

int

→ `SPI_ERROR_ARGUMENT` plan 为 NULL 时

描述

`SPI_freeplan` 释放先前由 `SPI_prepare` 返回或由 `SPI_saveplan` 保存的一个查询计划。

21.4. 数据更改的可见性

PostgreSQL 的数据更改可见性规则：在一次查询执行期间，该查询自身（通过 SQL 函数、SPI 函数、触发器）所做的数据更改对查询扫描是不可见的。例如，在查询

```
INSERT INTO a SELECT * FROM a
```

中，被插入的元组对 SELECT 的扫描不可见。其效果是在数据库表内部复制该表自身（当然要受唯一索引规则的约束），而不会递归。

查询 Q 所做的更改，对所有在查询 Q 之后启动的查询都可见，无论这些查询是在 Q 内部（即 Q 执行期间）启动，还是在 Q 完成之后启动。

21.5. 示例

这个 SPI 用法的示例演示了可见性规则。在 src/test/regress/regress.c 和 contrib/spi 中还有更复杂的示例。

这是一个非常简单的 SPI 用法示例。过程 execq 的第一个参数接受一条 SQL 查询，第二个参数接受 tcount，它使用 SPI_exec 执行该查询，并返回该查询所处理的元组数：

```
#include "executor/spi.h"   /* this is what you need to work with SPI
 */

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
```

```
    char *query;
    int ret;
    int proc;

    /* Convert given TEXT object to a C string */
    query = DatumGetCString(DirectFunctionCall1(textout,
                                                PointerGetDatum(sql))
);

    SPI_connect();

    ret = SPI_exec(query, cnt);

    proc = SPI_processed;
    /*
     * If this is SELECT and some tuple(s) fetched -
     * returns tuples to the caller via elog (NOTICE).
     */
    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++)
                sprintf(buf + strlen (buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog (NOTICE, "EXECQ: %s", buf);
        }
    }

    SPI_finish();

    pfree(query);

    return (proc);
}
```

现在，编译并创建该函数：

```
CREATE FUNCTION execq (text, integer) RETURNS integer
    AS '...path_to_so'
    LANGUAGE C;

vac=> SELECT execq('CREATE TABLE a (x INTEGER)', 0);
execq
-----
    0
```

```
(1 row)

vac=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)',0));
INSERT 167631 1
vac=> SELECT execq('SELECT * FROM a',0);
NOTICE:EXECQ:  0 <<< 由 execq 插入

NOTICE:EXECQ:  1 <<< 由 execq 返回并被外层 INSERT 插入的值

execq
-----
    2
(1 row)

vac=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a',1);
execq
-----
    1
(1 row)

vac=> SELECT execq('SELECT * FROM a', 10);
NOTICE:EXECQ:  0

NOTICE:EXECQ:  1

NOTICE:EXECQ:  2 <<< 0 + 2, 只插入了一个元组 —— 正如所指定的那样

execq
-----
    3            <<< 10 只是最大值，3 才是实际的元组数
(1 row)

vac=> DELETE FROM a;
DELETE 3
vac=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 167712 1
vac=> SELECT * FROM a;
x
-
1                <<< a 中没有元组 (0) + 1
(1 row)

vac=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
NOTICE:EXECQ:  0
INSERT 167713 1
vac=> SELECT * FROM a;
x
-
1
2                <<< a 中原先只有一个元组 + 1
(2 rows)

--    这演示了数据更改可见性规则：
```

```
vac=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  2
INSERT 0 2
vac=> SELECT * FROM a;
x
-
1
2
2                <<< 2 个元组 * 1（第一个元组中的 x）
6                <<< 3 个元组（刚插入的 2 + 1）* 2（第二个元组中的 x）
(4 rows)             ^^^^^^^^
                     execq() 在不同调用中可见的元组
```

部分 III. 过程语言

这一部分记录 PostgreSQL 发行版本中可用的过程语言以及有关过程语言的一般问题。

目录

22. 过程语言 286
22.1. 简介 286
22.2. 安装过程语言 286
23. PL/pgSQL - SQL 过程语言 288
23.1. 概述 288
23.1.1. 使用 PL/pgSQL 的优势 289
23.1.2. 用PL/pgSQL开发 289
23.2. PL/pgSQL的结构 290
23.2.1. 词法细节 290
23.3. 声明 291
23.3.1. 函数参数的别名 291
23.3.2. 行类型 292
23.3.3. 记录 292
23.3.4. 属性 292
23.3.5. RENAME 293
23.4. 表达式 293
23.5. 基本语句 294
23.5.1. 赋值 294
23.5.2. SELECT INTO 295
23.5.3. 执行没有结果的表达式或查询 296
23.5.4. 执行动态查询 296
23.5.5. 获取结果状态 298
23.6. 控制结构 298
23.6.1. 从函数返回 298
23.6.2. 条件语句 298
23.6.3. 简单循环 300
23.6.4. 在查询结果上循环 301
23.7. 游标 302
23.7.1. 声明游标变量 303
23.7.2. 打开游标 303
23.7.3. 使用游标 304
23.8. 错误和消息 304
23.8.1. 异常 305
23.9. 触发器过程 305
23.10. 示例 307
23.11. 从 Oracle PL/SQL 移植 308
23.11.1. 主要差异 308
23.11.2. 移植函数 309
23.11.3. 过程 313
23.11.4. 包 315
23.11.5. 其他需要注意的事项 316
23.11.6. 附录 316
24. PL/Tcl - Tcl 过程语言 320
24.1. 概述 320
24.2. 描述 320
24.2.1. PL/Tcl 函数和参数 320
24.2.2. PL/Tcl 中的数据值 321
24.2.3. PL/Tcl 中的全局数据 321
24.2.4. 从 PL/Tcl 访问数据库 322
24.2.5. PL/Tcl 中的触发器过程 324
24.2.6. 模块与 `unknown` 命令 325

24.2.7. Tcl 过程名称 325
25. PL/Perl - Perl 过程语言 326
25.1. 概述 326
25.2. 构建和安装 PL/Perl 326
25.3. 描述 327
25.3.1. PL/Perl 函数和参数 327
25.3.2. PL/Perl 中的数据值 328
25.3.3. 从 PL/Perl 访问数据库 328
25.3.4. 缺失的特性 328
26. PL/Python - Python 过程语言 330
26.1. 简介 330
26.2. 安装 330
26.3. 使用 PL/Python 330

第 22 章 过程语言

22.1. 简介

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编程语言函数，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 12.7 节中描述。标准 PostgreSQL 发行版提供了几种过程语言，可用作示例。

22.2. 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 template1 数据库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 shell 脚本 `createlang` 来代替手工完成各个细节。例如，要把 PL/pgSQL 安装到 template1 数据库，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。

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 12.5.7 节。

2. 必须用如下命令声明该调用处理器：

   ```
   CREATE FUNCTION handler_function_name ()
       RETURNS OPAQUE AS
       'path-to-shared-object' LANGUAGE C;
   ```

   特殊返回类型`OPAQUE`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` 指明是否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、PL/Perl 和 PL/

Python 已知是受信任的；而语言 PL/TclU 和 PL/PerlU 旨在提供无限制的功能，因此不应被标记为受信任的。

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/Python。`createlang` 脚本会自动完成上文所述的步骤 2和步骤 3。

例 22.1. 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 OPAQUE 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。

第 23 章 PL/pgSQL - SQL 过程语言

PL/pgSQL是PostgreSQL 数据库系统的一个可装载过程语言。

这个软件包最初由 Jan Wieck 编写。本文档的一部分由 Roberto Mello（<rmello@fslc.usu.edu>）撰写。

23.1. 概述

PL/pgSQL的设计目标是创建一种可装载的过程语言，它

- 可以用来创建函数和触发器过程，
- 向SQL语言中增加控制结构，
- 能执行复杂的计算，
- 继承所有用户定义的类型、函数和操作符，
- 可以被定义为服务器所信任，
- 易于使用。

PL/pgSQL调用处理器在函数第一次被调用时（在任一后端进程内）
解析函数的源文本并生成一棵内部二进制指令树。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 查询并不会立即被翻译。

当函数中的每个表达式和SQL查询第一次被使用时，
PL/pgSQL解释器会创建一个预备好的执行计划（使用SPI管理器的SPI_prepare 和SPI_saveplan函数）。之后再遇到该表达式或查询时会重用这个预备好的计划。因此，
一个包含大量可能需要执行计划的条件语句的函数，
只会准备和保存那些在数据库连接生命期内真正被用到的计划。这可以节省大量的解析活动。
一个缺点是某个特定表达式或查询中的错误可能直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查询生成了查询计划，
它就会在数据库连接的整个生命期内重用该计划。这通常对性能有利，
但如果你动态地更改数据库模式，它也可能引起一些问题。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，它会为 PERFORM 语句生成的查询计划引用 my_function()的 OID。
之后如果你删除并重建my_function()，那么 populate()将再也无法找到 my_function()。此时你必须重建 populate()，或者至少启动一个新的数据库会话让它重新编译。

由于PL/pgSQL以这种方式保存执行计划，直接出现在
PL/pgSQL函数中的查询在每次执行时必须引用相同的表和字段；也就是说，
你不能在查询中把参数用作表名或字段名。要绕开这一限制，你可以使用PL/pgSQL 的 EXECUTE 语句构造动态查询——代价是每次执行都要构造一个新的查询计划。

除了用户定义类型的输入/输出转换和计算函数之外，凡是能用 C 语言函数定义的东西也都可以用PL/pgSQL完成。你可以创建复杂条件计算函数，之后用它们来定义操作符，
或在函数索引中使用它们。

23.1.1. 使用 PL/pgSQL 的优势

- 更好的性能（参见第 23.1.1.1 节）
- SQL 支持（参见第 23.1.1.2 节）
- 可移植性（参见第 23.1.1.3 节）

23.1.1.1. 更好的性能

SQL是PostgreSQL（以及大多数其他关系数据库）用作查询语言的语言。它可移植且易于学习。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必须把每个查询发送到数据库服务器，等待它处理，接收结果，
做一些计算，然后再把其他查询发送给服务器。所有这些都会带来进程间通信开销，
而且如果你的客户端与数据库服务器不在同一台机器上，还会带来网络开销。

使用PL/pgSQL，你可以把一块计算和一系列查询封装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。这可以带来相当可观的性能提升。

23.1.1.2. SQL 支持

PL/pgSQL把过程语言的能力加到了 SQL的灵活和易用之上。使用 PL/pgSQL，你可以使用 SQL 的所有数据类型、列、操作符和函数。

23.1.1.3. 可移植性

由于PL/pgSQL函数运行在PostgreSQL 内部，这些函数可以在PostgreSQL运行的任何平台上运行。因此你可以重用代码并降低开发成本。

23.1.2. 用PL/pgSQL开发

用PL/pgSQL开发相当直接，尤其是如果你曾经用其他数据库过程语言（例如 Oracle 的PL/SQL）开发过。用PL/pgSQL开发的两个好办法是：

- 使用文本编辑器并用`psql`重新装载文件
- 使用PostgreSQL的 GUI 工具：PgAccess

用PL/pgSQL开发的一个好办法就是用你喜欢的文本编辑器创建函数，然后在另一个控制台用`psql`（PostgreSQL 的交互监视器）装载这些函数。如果你采用这种方式，一个好主意是用`CREATE OR REPLACE FUNCTION`编写函数。这样你就可以重新装载文件来更新函数定义。例如：

```
CREATE OR REPLACE FUNCTION testfunc(INTEGER) RETURNS INTEGER AS '
    ....
end;
' LANGUAGE 'plpgsql';
```

在`psql`运行期间，你可以用下面的命令装载或重新装载这样一个函数定义文件：

```
    \i filename.sql
```

然后立即发出 SQL 命令来测试该函数。

用PL/pgSQL开发的另一个好办法是使用 PostgreSQL的 GUI 工具PgAccess。它会替你做一些好事，比如转义单引号，并且让重建和调试函数变得容易。

23.2. 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。块的定义是：

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END;
```

块的语句部分中的任何语句都可以是一个子块。子块可以用于逻辑分组，
或者把变量局部化到一小组语句中。

块前面的声明部分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 sub-block
   --
   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';
```

重要的一点是不要把PL/pgSQL中用于给语句分组的 BEGIN/END 与用于事务控制的数据库命令混淆。PL/pgSQL的 BEGIN/END 只用于分组；它们不开始也不结束事务。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。

23.2.1. 词法细节

块中的每条语句和每个声明都以分号结尾。

所有关键字和标识符都可以用大小写混合的形式书写。标识符会被隐式地转换为小写，
除非用双引号括起来。

PL/pgSQL中有两种注释。双横线-- 开始一个延伸到行尾的注释。/*开始一个块注释，它延伸到下一个*/出现的位置。块注释不能嵌套，但双横线注释可以被包进块注释中，
而双横线也可以隐藏块注释的定界符/*和*/。

23.3. 声明

块中使用的所有变量、行和记录都必须在块的声明部分中声明。
（唯一的例外是遍历一段整数取值范围的 FOR 循环的循环变量，它会被自动声明为整数变量。）

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;
```

变量声明的一般语法是：

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];
```

DEFAULT 子句（如果给出）指定进入块时赋给变量的初始值。如果没有给出 DEFAULT 子句，那么变量会被初始化为 SQL的 NULL 值。

CONSTANT 选项防止变量被赋值，从而让它的值在块的整个期间保持常量。如果指定了 NOT NULL，赋予 NULL 值会导致一个运行时错误。所有声明为 NOT NULL 的变量都必须指定一个非 NULL 的默认值。

默认值在每次进入块时都会被求值。因此举例来说，把 'now'赋给一个timestamp类型的变量会让该变量拥有当前这次函数调用的时间，而不是函数被预编译时的时间。

例子：

```
quantity INTEGER DEFAULT 32;
url varchar := ''http://mysite.com'';
user_id CONSTANT INTEGER := 10;
```

23.3.1. 函数参数的别名

```
name ALIAS FOR $n;
```

传递给函数的参数用标识符$1、$2等命名。为了增强可读性，可以选择为 $n参数名声明别名。之后既可以引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';
```

23.3.2. 行类型

nametablename%ROWTYPE;

复合类型的变量被称为行变量（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，只要该查询的列集合与变量声明的类型相匹配。
行值的各个字段使用通常的点号记法访问，例如rowvar.field。

目前，行变量只能用%ROWTYPE记法声明；虽然人们可能期望裸表名也能用作类型声明，但在 PL/pgSQL函数中它不会被接受。

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，可以从中选取字段，例如 $1.user_id。

行类型变量只能访问表行的用户定义属性，而不能访问 OID 或其他系统属性（因为该行可能来自一个视图）。行类型的字段继承表中该字段的尺寸或精度（对于char(n)这样的数据类型）。

23.3.3. 记录

name RECORD;

记录变量与行类型变量类似，但它们没有预定义的结构。它们会采用在 SELECT 或 FOR 命令期间赋给它们的行的实际行结构。记录变量的子结构在每次被赋值时都可能改变。
这带来的一个后果是：在记录变量第一次被赋值之前，它没有子结构，
任何访问其中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。因此举例来说，你不能声明一个返回 RECORD 的函数。

23.3.4. 属性

使用%TYPE和%ROWTYPE属性，你可以声明与另一个数据库项（例如一个表字段）
具有相同数据类型或结构的变量。

variable%TYPE

%TYPE提供变量或数据库列的数据类型。你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你不需要知道你引用的结构的数据类型，而且最重要的是，
如果被引用项的数据类型将来发生改变（例如你把 user_id 的表定义从 INTEGER 改为 REAL），你可能不需要修改你的函数定义。

table%ROWTYPE

%ROWTYPE提供与指定表的整行相对应的复合数据类型。*table* 必须是数据库中已存在的表或视图的名字。

```
DECLARE
    users_rec users%ROWTYPE;
    user_id users.user_id%TYPE;
BEGIN
    user_id := users_rec.user_id;
    ...
```

```
CREATE FUNCTION does_view_exist(INTEGER) RETURNS bool AS '
   DECLARE
        key ALIAS FOR $1;
        table_data cs_materialized_views%ROWTYPE;
   BEGIN
        SELECT INTO table_data * FROM cs_materialized_views
               WHERE sort_key=key;

        IF NOT FOUND THEN
           RETURN false;
        END IF;
        RETURN true;
   END;
' LANGUAGE 'plpgsql';
```

23.3.5. RENAME

RENAME *oldname* TO *newname*;

使用 RENAME 声明可以更改变量、记录或行的名字。这主要在需要在触发器过程内部用另一个名字引用 NEW 或 OLD 时有用。另见 ALIAS。

例子:

```
RENAME id TO user_id;
RENAME this_var TO that_var;
```

> **注意**
>
> 截至 PostgreSQL 7.2，RENAME 似乎已损坏。修复它的优先级较低，因为 ALIAS 覆盖了 RENAME 的大部分实际用途。

23.4. 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。表面上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 解析器不可能识别除 NULL 关键字之外的真正常量值。所有表达式都通过SPI管理器在内部执行查询

SELECT *expression*

来求值。在表达式中，PL/pgSQL变量标识符的出现会被参数替代，
而变量的实际值则通过参数数组传递给执行器。这样 SELECT 的查询计划只需准备一次，之后就可以重用于后续的求值。

PostgreSQL主解析器所做的求值对常量值的解释有一些副作用。详细说来，
下面这两个函数的行为是有差别的：

```
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';
```

对于`logfunc1()`，PostgreSQL 主解析器在为 INSERT 准备计划时就知道字符串`'now'`应当被解释为`timestamp`，因为`logtable`的目标字段就是该类型。于是它会在此刻把它变成一个常量，
而这个常量值随后在后端的整个生命期内`logfunc1()`的所有调用中都被使用。不用说，
这并不是程序员想要的结果。

对于`logfunc2()`，PostgreSQL 主解析器不知道`'now'`应当变成什么类型，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`赋值时，PL/pgSQL解释器通过调用 `text_out()`和`timestamp_in()` 函数把这个字符串转换成`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。

记录变量的可变本性在此方面带来一个问题。当记录变量的字段被用在表达式或语句中时，
这些字段的数据类型在同一表达式的各次调用之间不得改变，
因为表达式将使用第一次到达该表达式时出现的数据类型来制定计划。
在编写为多张表处理事件的触发器过程时要牢记这一点。（必要时可以用 EXECUTE 绕开这个问题。）

23.5. 基本语句

在本节和接下来的几节中，我们描述PL/pgSQL 显式理解的所有语句类型。任何没有被识别为这些语句类型之一的东西都被假定为一个 SQL 查询，并被发送到主数据库引擎执行（在替换语句中使用的任何PL/pgSQL变量之后）。因此，例如 SQL 的`INSERT`、`UPDATE` 和`DELETE`命令可以被认为是PL/pgSQL 的语句，但它们没有在这里专门列出。

23.5.1. 赋值

把一个值赋给变量或行/记录字段写作：

identifier := *expression*;

如上所述，这类语句中的表达式通过发送给主数据库引擎的 SQL SELECT命令求值。表达式必须产生一个单一值。

如果表达式的结果数据类型与变量的数据类型不匹配，或者变量具有特定的尺寸/精度（如char(20)），结果值将被 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，这可能导致输入函数产生运行时错误。

例子:

```
user_id := 20;
tax := subtotal * 0.06;
```

23.5.2. SELECT INTO

产生多列（但只有一行）结果的 SELECT 命令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 expressions FROM ...;
```

其中*target*可以是一个记录变量、一个行变量，或者一个由简单变量和记录/
行字段组成的逗号分隔列表。注意这与PostgreSQL对 SELECT INTO 的通常解释 ——INTO 的目标是一张新创建的表——完全不同。（如果你想在 PL/pgSQL函数内部从 SELECT 结果创建表，请使用CREATE TABLE ... AS SELECT语法。）

如果用一行或一个变量列表作为目标，所选中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成查询结果列的行类型。

除 INTO 子句外，该 SELECT 语句与普通的 SQL SELECT 查询相同，可以使用 SELECT 的全部能力。

如果 SELECT 查询返回零行，NULL 会被赋给目标。如果 SELECT 查询返回多行，第一行会被赋给目标而其余的行被丢弃。（注意，除非你使用了 ORDER BY，否则“第一行”并没有良好的定义。）

目前，INTO 子句几乎可以出现在 SELECT 查询的任何位置，但建议像上面展示的那样把它紧放在 SELECT 关键字之后。PL/pgSQL的未来版本可能不会这么宽容地对待 INTO 子句的位置。

有一个名为 FOUND 的boolean类型特殊变量，可以在 SELECT INTO 之后立即使用它来检查赋值是否成功（也就是 SELECT 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;
```

另一种办法是，你可以用 IS NULL（或 ISNULL）条件来测试 RECORD/ROW 结果是否为 NULL。注意没有办法判断是否还有更多的行被丢弃。

```
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;
```

23.5.3. 执行没有结果的表达式或查询

有时你会想求值一个表达式或查询但丢弃其结果（典型情况是正在调用一个有有用的副作用但没有有用的结果值的函数）。在 PL/pgSQL中要做到这一点，使用 PERFORM 语句：

```
PERFORM query;
```

它执行一个SELECT *query*并丢弃结果。PL/pgSQL变量照常被替换进查询中。

> **注意**
>
> 人们可能以为不带 INTO 子句的 SELECT 就能实现这个结果，但目前唯一被接受的做法是 PERFORM。

一个例子：

```
PERFORM create_mv(''cs_session_page_requests_mv'',''
     SELECT   session_id, page_id, count(*) AS n_hits,
              sum(dwell_time) AS dwell_time, count(dwell_time) AS
 dwell_count
     FROM     cs_fact_table
     GROUP BY session_id, page_id '');
```

23.5.4. 执行动态查询

你经常会想在PL/pgSQL函数内部生成动态查询，
也就是每次执行时会涉及不同的表或不同数据类型的查询。
PL/pgSQL通常为查询缓存计划的做法在这类场景下不起作用。为了处理这类问题，提供了 EXECUTE 语句：

```
EXECUTE query-string;
```

其中*query-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 *query*。这个字符串被原样送入 SQL 引擎。

特别注意，查询字符串上的PL/pgSQL变量不会被替换。
变量的值必须在构造查询字符串时就被插入其中。

在使用动态查询时，你必须面对PL/pgSQL中单引号的转义问题。请参阅第 23.11 节中的表格，那里有能为你节省一些力气的详细解释。

与PL/pgSQL中的所有其他查询不同，由 EXECUTE 语句运行的*query*不会在服务器的生命期内只准备和保存一次。相反，*query*在语句每次运行时都会被重新准备。*query-string* 可以在过程内动态创建，以便对可变的表和字段执行操作。

SELECT 查询的结果会被 EXECUTE 丢弃，而且 EXECUTE 中目前不支持 SELECT INTO。因此，从动态创建的 SELECT 中提取结果的唯一办法是使用稍后描述的 FOR-IN-EXECUTE 形式。

一个例子：

```
EXECUTE ''UPDATE tbl SET ''
        || quote_ident(fieldname)
        || '' = ''
        || quote_literal(newvalue)
        || '' WHERE ...'';
```

这个例子展示了quote_ident(TEXT)　和quote_literal(TEXT)函数的用法。包含字段和表标识符的变量应当传给 quote_ident()函数。包含动态查询字符串的字面量成分的变量应当传给quote_literal()。这两个函数都会采取适当的步骤，返回被单引号或双引号括起、且内嵌的特殊字符被正确转义的输入文本。

下面是一个大得多的动态查询和 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'''';'';

    -- This works 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';
```

23.5.5. 获取结果状态

```
GET DIAGNOSTICS variable = item [ , ... ] ;
```

这条命令允许检索系统状态指示器。每个 *item*都是一个关键字，标识要赋给指定变量的状态值（该变量应当具有接收它的正确数据类型）。当前可用的状态项有：ROW_COUNT，发送到SQL 引擎的最后一条SQL查询所处理的行数；以及 RESULT_OID，最近的SQL查询插入的最后一行的Oid。注意RESULT_OID只在 INSERT 查询之后有用。

23.6. 控制结构

控制结构可能是PL/pgSQL中最有用（也最重要）的部分。借助PL/pgSQL的控制结构，
你可以以非常灵活而强大的方式操纵PostgreSQL数据。

23.6.1. 从函数返回

```
RETURN expression;
```

函数终止，*expression*的值会被返回给上层执行器。
表达式的结果会被自动转换成函数的返回类型，转换方式与赋值中描述的相同。

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碰到 RETURN 语句，就会发生运行时错误。

23.6.2. 条件语句

IF语句让你可以基于某些条件执行命令。PL/pgSQL有四种 IF 形式：IF-THEN、IF-THEN-ELSE、IF-THEN-ELSE IF 和 IF-THEN-ELSIF-THEN-ELSE。

23.6.2.1. IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

IF-THEN 语句是 IF 的最简单形式。如果条件为真，THEN 和 END IF 之间的语句会被执行。否则，它们会被跳过。

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

23.6.2.2. IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
```

```
END IF;
```

IF-THEN-ELSE 语句在 IF-THEN 的基础上增加了一种能力：你可以指定一组在条件求值为 FALSE 时执行的备选语句。

```
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;
```

23.6.2.3. IF-THEN-ELSE IF

IF 语句可以嵌套，如下面的例子所示：

```
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，你实际上是把一个 IF 语句嵌套在外层 IF 语句的 ELSE 部分中。因此每个嵌套的 IF 都需要一个 END IF 语句，外层的 IF-ELSE 也需要一个。这是可行的，但当要检查的备选分支很多时就会变得乏味。

23.6.2.4. 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 提供了一种在一条语句中检查多个备选分支的更方便的方法。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;
```

最后的 ELSE 部分是可选的。

23.6.3. 简单循环

使用 LOOP、EXIT、WHILE 和 FOR 语句，你可以让你的 PL/pgSQL函数重复执行一系列命令。

23.6.3.1. LOOP

```
[<<label>>]
LOOP
    statements
END LOOP;
```

LOOP 定义一个无条件循环，它会无限重复，直到被 EXIT 或 RETURN 语句终止。可选的标签可以被嵌套循环中的 EXIT 语句用来指定应该终止哪一层嵌套。

23.6.3.2. EXIT

```
EXIT [ label ] [ WHEN expression ];
```

如果没有给出*label*，最内层的循环会被终止，接下来执行 END LOOP 之后的语句。如果给出了 *label*，它必须是当前循环或嵌套循环/块的某个外层的标签。此时被命名的循环或块被终止，控制转移到该循环/块对应的 END 之后的语句。

如果出现了 WHEN，只有当指定条件为真时循环才会退出，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;
END LOOP;

BEGIN
    -- some computations
    IF stocks > 100000 THEN
```

```
        EXIT;  -- illegal. Can't use EXIT outside of a LOOP
    END IF;
END;
```

23.6.3.3. 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执行一系列语句。条件在每次进入循环体之前被检查。

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;
```

23.6.3.4. FOR（整数 for 循环）

```
[<<label>>]
FOR name IN [ REVERSE ] expression .. expression LOOP
    statements
END LOOP;
```

这种形式的 FOR 创建一个遍历一段整数值范围的循环。变量 *name*被自动定义为 integer 类型并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被求值一次。迭代步长通常是 1，但指定 REVERSE 时是 -1。

整数 FOR 循环的一些例子：

```
FOR i IN 1..10 LOOP
  -- some expressions here

    RAISE NOTICE ''i is %'',i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- some expressions here
END LOOP;
```

23.6.4. 在查询结果上循环

使用另一种类型的 FOR 循环，你可以遍历一个查询的结果并相应地操纵这些数据。语法是：

```
[<<label>>]
FOR record | row IN select_query LOOP
    statements
```

```
END LOOP;
```

记录或行变量会被依次赋以 SELECT 查询产生的所有行，并且循环体对每一行都会执行。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，最后赋值的行在循环之后仍然可以访问。

FOR-IN-EXECUTE 语句是在记录上迭代的另一种方式：

```
[<<label>>]
FOR record | row IN EXECUTE text_expression LOOP
    statements
END LOOP;
```

它与前一种形式类似，区别在于源 SELECT 语句被指定为一个字符串表达式，该表达式在每次进入 FOR 循环时被求值并重新制定计划。这样程序员就可以像使用普通 EXECUTE 语句那样，在预计划查询的速度和动态查询的灵活性之间做出选择。

> **注意**
>
> PL/pgSQL解析器目前通过检查 FOR 后面提到的目标变量是否已被声明为记录/行变量来区分两种 FOR 循环（整数型或返回记录型）。如果不是，它就被认为是整数 FOR 循环。当真正的问题比如说只是 FOR 变量名拼写错误时，这会导致相当不直观的错误消息。

23.7. 游标

与一次执行整个查询不同，可以先建立一个封装该查询的游标，
然后每次读取查询结果的若干行。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 循环会在内部自动使用游标来避免内存问题。）一个更有趣的用法是函数可以返回一个指向它所建立的游标的引用，
让调用方去读取这些行。这提供了从函数返回行集的一种办法。

23.7.1. 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，游标变量总是特殊数据类型refcursor。
创建游标变量的一种办法就是把它声明为refcursor类型的变量。
另一种办法是使用游标声明语法，其一般形式是：

```
name CURSOR [ ( arguments ) ] FOR select_query ;
```

（为了与 Oracle 兼容，FOR可以替换为 IS。）*arguments*（如果有）是一个由*name datatype*对组成的逗号分隔列表，这些名字在给定查询中将被参数值替换。替换这些名字的实际值将在稍后打开游标时指定。

一些例子：

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * from tenk1;
    curs3 CURSOR (key int) IS SELECT * from tenk1 where unique1 = key;
```

这三个变量的数据类型都是refcursor，但第一个可以与任何查询一起使用，
第二个已经有一个完整指定的查询绑定在它上面，而最后一个绑定了一个参数化查询。
（打开游标时key会被一个整数参数值替换。）变量curs1被称为未绑定的，
因为它没有绑定到任何特定的查询。

23.7.2. 打开游标

在游标可以用来检索行之前，它必须被打开。（这等价于 SQL 命令DECLARE CURSOR。）PL/pgSQL有四种形式的 OPEN 语句，其中两种用于未绑定的游标变量，另外两种用于绑定的游标变量。

23.7.2.1. OPEN FOR SELECT

```
OPEN unbound-cursor FOR SELECT ...;
```

游标变量被打开并被赋予要执行的指定查询。该游标不能已经处于打开状态，
而且它必须已被声明为未绑定游标（也就是一个简单的refcursor变量）。SELECT 查询的处理方式与PL/pgSQL中的其他 SELECT 相同：会替换 PL/pgSQL变量名，并且查询计划会被缓存以便可能的重用。

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

23.7.2.2. OPEN FOR EXECUTE

```
OPEN unbound-cursor FOR EXECUTE query-string;
```

游标变量被打开并被赋予要执行的指定查询。该游标不能已经处于打开状态，
而且它必须已被声明为未绑定游标（也就是一个简单的refcursor变量）。查询以与 EXECUTE 命令相同的方式指定为字符串表达式。与通常一样，这让查询可以在每次运行之间变化，
提供了灵活性。

```
OPEN curs1 FOR EXECUTE ''SELECT * FROM '' || quote_ident($1);
```

23.7.2.3. 打开绑定的游标

```
OPEN bound-cursor [ ( argument_values ) ];
```

这种形式的 OPEN 用于打开其查询在声明时就已绑定到它上面的游标变量。该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，才必须出现一个实际参数值表达式的列表。这些值会被替换进查询中。绑定游标的查询计划总被认为是可缓存的 ——这种情况下没有 EXECUTE 的等价物。

```
OPEN curs2;
OPEN curs3(42);
```

23.7.3. 使用游标

游标一旦被打开，就可以用这里描述的语句来操纵它。

这些操纵不必发生在最初打开游标的同一个函数中。你可以从函数返回一个refcursor值，让调用方操作该游标。（在内部，refcursor值只是包含该游标的活动查询的 Portal 的字符串名字。这个名字可以被传递、赋给其他 refcursor变量等等，而不会干扰 Portal。）

所有 Portal 都会在事务结束时被隐式关闭。因此 refcursor值只在事务结束之前引用打开的游标才有用。

23.7.3.1. FETCH

```
FETCH cursor INTO target;
```

FETCH 从游标中检索下一行放到目标中，目标可以是行变量、记录变量或由简单变量组成的逗号分隔列表，与 SELECT INTO 相同。与 SELECT INTO 一样，可以检查特殊变量 FOUND 来看是否取得了行。

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo,bar,baz;
```

23.7.3.2. CLOSE

```
CLOSE cursor;
```

CLOSE 关闭一个打开的游标底层的 Portal。这可以用来在事务结束之前更早地释放资源，或者腾出游标变量以便再次打开它。

```
CLOSE curs1;
```

23.8. 错误和消息

使用 RAISE 语句来报告消息和引发错误。

```
RAISE level 'format' [, variable [...]];
```

可用的级别有 DEBUG（把消息写进 postmaster 日志）、NOTICE（把消息写进 postmaster 日志并转发给客户端应用）和 EXCEPTION（引发一个错误，中止事务）。

在格式字符串中，%会被下一个可选参数的外部表示替换。要发出字面的%，写成 %%。注意可选参数目前必须是简单变量而不是表达式，而且格式必须是一个简单的字符串字面量。

例子：

```
RAISE NOTICE ''Calling cs_create_job(%)'',v_job_id;
```

在这个例子中，v_job_id 的值会替换字符串中的 %。

```
RAISE EXCEPTION ''Inexistent ID --> %'',user_id;
```

这会以给定的错误消息中止事务。

23.8.1. 异常

PostgreSQL没有一个非常聪明的异常处理模型。每当解析器、规划器/优化器或执行器判定一条语句无法再继续处理时，整个事务都会被中止，系统跳回主循环去从客户端应用获取下一条查询。

可以钩进错误机制来注意到这种情况的发生。但目前无法判断究竟是什么导致了中止（输入/输出转换错误、浮点错误、解析错误）。而且此时数据库后端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，当PL/pgSQL在函数或触发器过程执行期间遇到中止时，它目前唯一会做的事情就是写一些额外的 NOTICE 级别日志消息，指出这件事发生在哪个函数中以及在哪里（行号和语句类型）。该错误总是会让函数停止执行。

23.9. 触发器过程

PL/pgSQL可以用来定义触发器过程。触发器过程用CREATE FUNCTION命令创建，是一个没有参数、返回类型为OPAQUE的函数。注意即使函数期望接收CREATE TRIGGER中指定的参数，它也必须被声明为没有参数——触发器参数通过TG_ARGV传递，如下所述。

当一个PL/pgSQL函数作为触发器被调用时，顶层块中会自动创建几个特殊变量。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。

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。

TG_NAME

数据类型name；包含实际被触发的触发器名字的变量。

TG_WHEN

数据类型text；依据触发器定义而是 BEFORE或AFTER 之一的字符串。

TG_LEVEL

数据类型text；依据触发器定义而是 ROW或STATEMENT 之一的字符串。

TG_OP

数据类型text；INSERT、UPDATE 或DELETE之一的字符串，指出触发器是为哪个操作而触发。

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字。

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[]

text类型的数组；来自CREATE TRIGGER 语句的参数。索引从 0 开始计数并且可以用表达式给出。无效的索引（< 0 或 >= tg_nargs）得到 NULL 值。

触发器函数必须返回 NULL，或者返回一个与触发该触发器的表的结构完全相同的记录/行值。BEFORE 触发器可以返回 NULL 来通知触发器管理器跳过这一行操作的其余部分（即不再触发后续触发器，而且这一行的 INSERT/UPDATE/DELETE 不会发生）。如果返回的是非 NULL 值，操作就会以那个行值继续进行。注意返回一个与 NEW 的原始值不同的行值会改变将被插入或更新的行。可以直接在 NEW 中替换单个值然后返回它，也可以构造一个完整的新记录/行来返回。

AFTER 触发器的返回值会被忽略；它总是返回 NULL 值也无妨。但 AFTER 触发器仍然可以通过引发错误来中止操作。

例 23.1. 一个PL/pgSQL触发器过程示例

这个示例触发器保证每当表中有行被插入或更新时，当前的用户名和时间都会被盖章到该行中。而且它还保证给出了员工的名字并且薪水是一个正值。

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS '
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname ISNULL THEN
            RAISE EXCEPTION ''empname cannot be NULL value'';
        END IF;
        IF NEW.salary ISNULL THEN
            RAISE EXCEPTION ''% cannot have NULL salary'', NEW.
empname;
        END IF;

        -- Who works for us when she must pay for?
```

```
        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();
```

23.10. 示例

这里只有几个函数用来展示编写PL/pgSQL函数有多么容易。想要更复杂的例子，程序员可以去看 PL/pgSQL的回归测试。

在PL/pgSQL中编写函数的一个令人头疼的细节是单引号的处理。CREATE FUNCTION 中函数的源文本必须是一个字面量字符串。字面量字符串内部的单引号必须加倍或者用反斜杠引用。我们仍在寻找优雅的替代方案。在此期间，应当像下面的例子那样把单引号加倍。PostgreSQL未来版本中对这一问题的任何解决方案都将是向前兼容的。

有关在不同情况下如何转义单引号的详细解释和例子，请参阅第 23.11.1.1 节。

例 23.2. 一个让整数加一的简单PL/pgSQL函数

下面两个PL/pgSQL函数与 C 语言函数讨论中的对应函数完全相同。这个函数接收一个integer 并让它加一，返回加一后的值。

```
CREATE FUNCTION add_one (integer) RETURNS INTEGER AS '
    BEGIN
        RETURN $1 + 1;
    END;
' LANGUAGE 'plpgsql';
```

例 23.3. 一个连接文本的简单PL/pgSQL函数

这个函数接收两个text参数并返回它们连接后的结果。

```
CREATE FUNCTION concat_text (TEXT, TEXT) RETURNS TEXT AS '
    BEGIN
        RETURN $1 || $2;
    END;
' LANGUAGE 'plpgsql';
```

例 23.4. 一个操作复合类型的PL/pgSQL函数

在这个例子中，我们把EMP（一张表）和一个 integer作为函数的参数，函数返回一个 boolean。如果EMP表的 salary字段为NULL，我们返回f。否则我们拿该字段与传给函数的integer比较，并返回比较的boolean 结果（t 或 f）。这是 C 函数中那个例子在 PL/pgSQL中的等价物。

```
CREATE FUNCTION c_overpaid (EMP, INTEGER) RETURNS BOOLEAN AS '
    DECLARE
        emprec ALIAS FOR $1;
        sallim ALIAS FOR $2;
    BEGIN
        IF emprec.salary ISNULL THEN
            RETURN ''f'';
        END IF;
        RETURN emprec.salary > sallim;
    END;
' LANGUAGE 'plpgsql';
```

23.11. 从 Oracle PL/SQL 移植

Roberto Mello <rmello@fslc.usu.edu>

作者

Roberto Mello（<rmello@fslc.usu.edu>）

本节解释 Oracle 的 PL/SQL 与PostgreSQL 的PL/pgSQL语言之间的差异，希望能帮助开发者把应用从 Oracle 移植到PostgreSQL。这里的多数代码来自ArsDigita[1] 的Clickstream 模块[2]，那是我在 2000 年夏到 OpenForce Inc.[3]实习时移植到PostgreSQL的。

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）。PL/SQL 比它在 PostgreSQL中的对应物拥有多得多的特性，但 PL/pgSQL提供了大量的功能，而且它在不断改进。

23.11.1. 主要差异

从 Oracle 移植到PostgreSQL时应当记住的一些事情：

- PostgreSQL没有默认参数。
- 在PostgreSQL中可以重载函数。这常被用来变通解决缺少默认参数的问题。
- 赋值、循环和条件语句是相似的。
- PostgreSQL中不需要游标，只要把查询放进 FOR 语句即可（见下面的例子）
- 在PostgreSQL中你需要转义单引号。参见第 23.11.1.1 节。

23.11.1.1. 转义单引号

在PostgreSQL中，你需要转义函数定义内部的单引号。这有时会导致相当有趣的代码，
尤其是当你在创建一个会生成其他函数的函数时，就像例 23.6中那样。在转义大量单引号时要记住的一点是：除开头/结尾的引号外，其余引号都会成对出现。

表 23.1给出了全部诀窍。（你会爱上这张小表的。）

[1] http://www.arsdigita.com
[2] http://www.arsdigita.com/asj/clickstream
[3] http://www.openforce.net

表 23.1. 单引号转义表

引号数	用途	示例	结果
1	开始/结束函数体	`CREATE FUNCTION foo() RETURNS INTEGER AS '...' LANGUAGE 'plpgsql';`	原样
2	在赋值、SELECT 中定界字符串等	`a_output := ''Blah''; SELECT * FROM users WHERE f_name=''foobar'';`	`SELECT * FROM users WHERE f_name='foobar';`
4	当你需要在结果字符串中出现两个单引号而又不终止该字符串时。	`a_output := a_output || '' AND name LIKE ''''foobar'''' AND ...''`	`AND name LIKE 'foobar' AND ...`
6	当你想在结果字符串中出现两个单引号并且终止该字符串时。	`a_output := a_output || '' AND name LIKE ''''foobar''''''`	`AND name LIKE 'foobar'`
10	当你想在结果字符串中出现两个单引号（这需要 8 个引号）并且终止该字符串（再需要 2 个）时。只有在你用一个函数生成其他函数时才可能需要这样做（就像例 23.6中那样）。	`a_output := a_output || '' if v_'' || referrer_keys.kind || '' like '''''''''' || referrer_keys.key_string || '''''''''' then return ''''''  || referrer_keys.referrer_type || ''''''; end if;'';`	`if v_<...> like ''<...>'' then return ''<...>''; end if;`

23.11.2. 移植函数

例 23.5. 一个简单的函数

下面是一个 Oracle 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声明别名。
- Oracle 的函数可以有IN、OUT 和INOUT参数。例如INOUT 意味着该参数接收一个值并返回另一个值。PostgreSQL只有 “IN” 参数，而且函数只能返回单个值。
- 函数原型（而非函数体）中的RETURN关键字在PostgreSQL中变成 RETURNS。
- 在PostgreSQL中，创建函数使用单引号作为定界符，
  因此你必须转义函数内部的单引号（这有时会相当令人厌烦；参见第 23.11.1.1 节）。
- /show errors命令在 PostgreSQL中不存在。

那么让我们看看这个函数移植到PostgreSQL 之后是什么样子：

```
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 23.6. 一个创建另一个函数的函数

下面的过程从一条SELECT语句抓取行，并为了效率把结果构建成带IF语句的一个大函数。
请特别注意游标、FOR循环的差异，以及在 PostgreSQL中转义单引号的必要性。

```
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中最终的样子：

```
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'''';'';

    -- This works 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 23.7. 一个带大量字符串操作和 OUT 参数的过程

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）。它是一个过程，因为在PL/pgSQL 中函数只能返回一个值（参见第 23.11.3 节）。在 PostgreSQL中，绕开这个问题的一种办法是把该过程拆成三个不同的函数：一个返回主机，一个返回路径，另一个返回查询。

```
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, '//'); -- PostgreSQL doesn't have an instr
 function

    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;
```

下面是这个过程可以怎样为PostgreSQL 改写：

```
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';
```

> **注意**
>
> PostgreSQL没有instr函数，因此你可以用其他函数的组合来变通。我厌倦了这样做，于是创建了自己的instr函数，其行为与 Oracle 的完全一致（这让生活更轻松）。代码见第 23.11.6 节。

23.11.3. 过程

Oracle 的过程给开发者多了一点灵活性，因为不需要显式返回任何东西，但也可以通过INOUT或OUT参数返回。

一个例子:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;1
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;2

    SELECT count(*) INTO a_running_job_count
    FROM cs_jobs
    WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock3
        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 exists4
    END;
    COMMIT;
END;
/
show errors
```

这样的过程可以很容易地转换成返回INTEGER的 PostgreSQL函数。这个特定的过程很有意思，因为它能教给我们一些东西：

1. PostgreSQL没有pragma 语句。
2. 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。
3. 你也不能在PL/pgSQL过程中使用事务。整个函数（以及从中调用的其他函数）在一个事务中执行，如果出了问题PostgreSQL会回滚结果。因此只允许一个BEGIN语句。
4. exception when 必须被替换成一个IF语句。

那么让我们看看把这个过程移植到PL/pgSQL 的一种做法：

```
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;
    -- PRAGMA AUTONOMOUS_TRANSACTION;
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
        -- COMMIT; -- free lock
        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,
 sysdate());
        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）。

23.11.4. 包

> **注意**
>
> 我自己没有怎么用过包，所以如果这里有错误，请告诉我。

包是 Oracle 提供的一种把 PL/SQL 语句和函数封装到一个实体中的手段，就像 Java 的类一样，你在其中定义方法和对象。你可以用“.”（点号）访问这些对象/方法。下面是 ACS 4（ArsDigita Community System[4]）中一个 Oracle 包的例子：

```
CREATE OR REPLACE PACKAGE BODY acs
AS
  FUNCTION add_user (
    user_id     IN users.user_id%TYPE DEFAULT NULL,
    object_type     IN acs_objects.object_type%TYPE DEFAULT 'user',
    creation_date   IN acs_objects.creation_date%TYPE DEFAULT sysdate,
    creation_user   IN acs_objects.creation_user%TYPE DEFAULT NULL,
    creation_ip     IN acs_objects.creation_ip%TYPE DEFAULT NULL,
  ...
  ) RETURN users.user_id%TYPE
  IS
    v_user_id       users.user_id%TYPE;
    v_rel_id        membership_rels.rel_id%TYPE;
  BEGIN
    v_user_id := acs_user.new (user_id, object_type, creation_date,
                 creation_user, creation_ip, email, ...
    RETURN v_user_id;
  END;
END acs;
/
show errors
```

我们把 Oracle 包的各个对象以标准命名约定创建为函数，从而把它移植到PostgreSQL。我们还必须注意其他一些细节，比如PostgreSQL函数缺少默认参数。上面的包会变成类似这样：

```
CREATE FUNCTION acs__add_user(INTEGER,INTEGER,VARCHAR,DATETIME,
INTEGER,INTEGER,...)
RETURNS INTEGER AS '
DECLARE
    user_id ALIAS FOR $1;
    object_type ALIAS FOR $2;
    creation_date ALIAS FOR $3;
    creation_user ALIAS FOR $4;
    creation_ip ALIAS FOR $5;
    ...
    v_user_id users.user_id%TYPE;
```

[4] http://www.arsdigita.com/doc/

```
    v_rel_id membership_rels.rel_id%TYPE;
BEGIN
    v_user_id := acs_user__new(user_id,object_type,creation_date,
creation_user,creation_ip, ...);
    ...

    RETURN v_user_id;
END;
' LANGUAGE 'plpgsql';
```

23.11.5. 其他需要注意的事项

23.11.5.1. EXECUTE

PostgreSQL版本的EXECUTE 工作得很好，但你必须记住按照第 23.5.4 节中的说明使用quote_literal(TEXT)和 quote_string(TEXT)。像 EXECUTE ''SELECT * from $1'';这类构造如果不使用这些函数就无法工作。

23.11.5.2. 优化PL/pgSQL函数

PostgreSQL提供了两个函数创建修饰符来优化执行：iscachable（给定相同参数时函数总是返回相同结果）和isstrict（任一参数为 NULL 时函数返回NULL）。细节请查阅CREATE FUNCTION参考。

要使用这些优化属性，你必须在CREATE FUNCTION语句中使用WITH修饰符。类似这样：

```
CREATE FUNCTION foo(...) RETURNS INTEGER AS '
...
' LANGUAGE 'plpgsql'
WITH (isstrict, iscachable);
```

23.11.6. 附录

23.11.6.1. 我的instr函数的代码

```
--
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1,string2,[n],[m]) where [] denotes optional
 params.
--
-- 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).
--
-- by Roberto Mello (rmello@fslc.usu.edu)
-- modified by Robert Gaszewski (graszew@poland.com)
-- Licensed under the GPL v2 or later.
--

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,INTEGER) 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';

--
-- Written by Robert Gaszewski (graszew@poland.com)
-- Licensed under the GPL v2 or later.
--
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 24 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。

本包最初由 Jan Wieck 编写。

24.1. 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后端的内部，也无法像 C 函数那样在 PostgreSQL 用户 ID 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 过程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` 脚本，例如 `createlang pltcl` *`dbname`* 或 `createlang pltclu` *`dbname`*。

24.2. 描述

24.2.1. 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，只是语言应指定为 `'pltclu'`。

函数体就是一段 Tcl 脚本。调用函数时，参数值以变量 `$1` ... `$n` 传入 Tcl 脚本。结果由 Tcl 代码按通常方式用 `return` 语句返回。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' WITH (isStrict);
```

注意子句 `WITH (isStrict)`，它让我们不必考虑 NULL 输入：如果传入的是 NULL，函数根本不会被调用，而是会自动返回 NULL 结果。

在非严格函数中，如果某个参数的实际值为 NULL，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 tcl_max 在一个参数为 null、另一个非 null 时返回非 null 的参数，而不是返回 NULL：

```
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 数组传递给过程。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 NULL，它就不会出现在数组中！下面是一个在 PL/Tcl 中定义 overpaid_2 函数（见旧版 PostgreSQL 文档）的例子：

```
CREATE FUNCTION overpaid_2 (EMP) RETURNS bool AS '
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
' LANGUAGE 'pltcl';
```

目前尚不支持返回复合类型的结果值。

24.2.2. PL/Tcl 中的数据值

提供给 PL/Tcl 函数脚本的参数值，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。

24.2.3. PL/Tcl 中的全局数据

有时需要在两次过程调用之间保留某些全局状态数据，或者在不同过程之间共享。
这很容易做到，因为同一后端中执行的所有 PL/Tcl 过程共享同一个安全的 Tcl 解释器。因此，任何全局 Tcl 变量都可以被所有 PL/Tcl 过程调用访问，并在 SQL 客户端连接的整个期间持续存在。（注意 PL/TclU 函数同样共享全局数据，但它们位于另一个不同的 Tcl 解释器中，无法与 PL/Tcl 函数通信。）

为帮助防止 PL/Tcl 过程无意间彼此干扰，每个过程都可通过upvar命令访问一个全局数组。该变量的全局名称是过程的内部名称，局部名称是GD。建议将 GD 用于保存过程的私有状态数据。只有当你明确希望某些值在多个过程之间共享时，才应使用常规的 Tcl 全局变量。

下面spi_execp的示例展示了GD的用法。

24.2.4. 从 PL/Tcl 访问数据库

在 PL/Tcl 过程体中可以使用下列命令来访问数据库：

`spi_exec?-count *n*? ?-array *name*? *query* ?*loop-body*?`

执行以字符串形式给出的 SQL 查询。查询出错时会引发错误。否则，该命令的返回值是查询处理的行数（选出、插入、更新或删除的行），如果查询是工具语句则返回零。此外，如果查询是 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` 此查询最多处理多少行，其效果类似于将查询设为游标后执行 `FETCH n`。

如果查询是 SELECT 语句，则该语句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中，SELECT 列名用作数组索引。

如果查询是 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 SELECT 结果中的每一行执行一次（注意：如果给定的查询不是 SELECT，则忽略 *loop-body*）。在每次迭代之前，当前行各字段的值会被存入 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 DEBUG 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 SELECT 结果的某个字段为 NULL，则其目标变量会被“unset”，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后端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 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，其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

返回由最后一条被 spi_exec 或被 spi_execp 执行的查询所插入行的 OID（当该查询是单行 INSERT 时；否则得到零）。

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 查询中的字符串加引号。例如，设想类似下面这样的查询字符串：

```
"SELECT '$val' AS ret"
```

其中 Tcl 变量 val 的实际内容是 doesn't。这会得到最终查询字符串：

```
SELECT 'doesn't' AS ret
```

这会在 spi_exec 或 spi_prepare 期间导致解析错误。提交的查询应当包含：

```
SELECT 'doesn''t' AS ret
```

它在 PL/Tcl 中可以构成为：

```
"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 查询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、NOTICE、ERROR 和 FATAL。DEBUG 和 NOTICE 只是像 elog 后端 C 函数那样把给定的消息发到 postmaster 日志中（NOTICE

还会发送给客户端）。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后端关闭。（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）。

24.2.5. PL/Tcl 中的触发器过程

触发器过程可以用 PL/Tcl 编写。按照 PostgreSQL 的惯例，要作为触发器调用的过程必须声明为一个无参数、返回类型为 opaque 的函数。

来自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

表字段名构成的 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元素名时，返回的元素编号会从 1 开始表示第一列，这与 PostgreSQL 中字段的惯常编号方式一致。

$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

一个包含新表行值的关联数组，用于 INSERT/UPDATE 动作；对于 DELETE 则为空。该数组以字段名为索引。值为 NULL 的字段不会出现在数组中！

$OLD

一个包含旧表行值的关联数组，用于 UPDATE/DELETE 动作；对于 INSERT 则为空。该数组以字段名为索引。值为 NULL 的字段不会出现在数组中！

$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/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 OPAQUE 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。

24.2.6. 模块与 unknown 命令

PL/Tcl 支持在使用时自动装载（auto-loading）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，并在创建解释器之后立即装载到 Tcl 解释器中。

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 “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。

24.2.7. Tcl 过程名称

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 中该过程所在行的对象 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 25 章 PL/Perl - Perl 过程语言

PL/Perl 是一种可装载的过程语言，它使 Perl[1] 编程语言可以用来编写 PostgreSQL 函数。

25.1. 概述

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后端的内部，也无法在 PostgreSQL 用户 ID 的权限下获得 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。

有时需要编写不受限制的 Perl 函数--例如，可能需要一个发送邮件的 Perl 函数。为了处理这些情况，PL/Perl 还可以作为一种“非受信任”语言安装（通常命名为 plperlu）。这种情况下完整的 Perl 语言都可用。PL/PerlU 函数的编写者必须注意函数不能被用来做任何不希望的事，因为它将能做以数据库管理员身份登录的用户能做的任何事。
注意数据库系统只允许数据库超级用户在非受信任语言中创建函数。

25.2. 构建和安装 PL/Perl

如果向 configure 脚本提供了 --with-perl 选项，PostgreSQL 构建过程就会尝试构建 PL/Perl 共享库并将其安装到 PostgreSQL 库目录中。

在大多数平台上，由于 PL/Perl 是一个共享库，libperl 库也必须是共享库。在撰写本文时，预构建的 Perl 包几乎都不是这样。如果你遇到这个困难，构建期间会出现像这样的消息来指出这个事实：

```
*** 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.
```

如果看到这条消息，你就必须手工重新构建并安装 Perl 才能构建 PL/Perl。在 Perl 的配置过程中，请要求生成共享库。

重新安装 Perl 之后，切换到 PostgreSQL 源代码树中的 src/pl/plperl 目录并执行命令

```
gmake clean
gmake all
gmake install
```

以完成 PL/Perl 共享库的构建和安装。

要在特定数据库中安装 PL/Perl 和/或 PL/PerlU，请使用 createlang 脚本，例如 createlang plperl *dbname* 或 createlang plperlu *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。

[1] http://www.perl.com

25.3. 描述

25.3.1. 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;
```

PL/PerlU 与此相同，只是语言应指定为 `plperlu`。

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;
```

如果把一个 NULL 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 NULL 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 `WITH (isStrict)`，让 PostgreSQL 做更合理的事情：如果传入 NULL，函数根本不会被调用，而是自动返回 NULL 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 NULL：

```
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 $a; }
    if ($a > $b) { return $a; }
    return $b;
' LANGUAGE plperl;
```

如上所示，要让 PL/Perl 函数返回 NULL，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 的扩展引号函数（`q[]`、`qq[]`、`qw[]`）来避免写单引号。

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。

注意，如果同一个函数是由超级用户用语言 `plperlu` 创建的，执行就会成功。

25.3.2. PL/Perl 中的数据值

提供给 PL/Perl 函数脚本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。

25.3.3. 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`、`NOTICE` 和 `ERROR`。`DEBUG` 和 `NOTICE` 只是把给定的消息发送到 postmaster 日志（对 `NOTICE` 而言还会发送给客户端）。`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。

25.3.4. 缺失的特性

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。

[2] http://www.cpan.org/modules/by-module/DBD/APILOS/
[3] http://www.cpan.org/SITES.html

PL/Perl 目前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。

第 26 章 PL/Python - Python 过程语言

26.1. 简介

PL/Python 过程语言允许用 Python[1] 语言编写 PostgreSQL 函数。

当前版本的 PL/Python 只作为受信任的语言工作；对文件系统和其他本地资源的访问被禁用。特别地，PL/Python 使用 Python 的受限执行环境，并进一步限制它以防止使用文件 `open` 调用，而且只允许导入一个特定列表中的模块。目前，该列表包括：array、bisect、binascii、calendar、cmath、codecs、errno、marshal、math、md5、mpz、operator、pcre、pickle、random、re、regex、sre、sha、string、StringIO、struct、time、whrandom 和 zlib。

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。

26.2. 安装

要构建 PL/Python，需要在运行 `configure` 时指定 `--with-python` 选项。如果构建和安装之后你有一个名为 `plpython.so` 的文件（扩展名可能不同），那么一切顺利。否则你应当看到过一条像这样飞过的通知：

```
*** 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方法来做这件事。我们能提供给你的最接近的东西是 Python FAQ 3.30[2] 中的信息。在某些操作系统上你其实不必构建共享库，但那样你就得让 PostgreSQL 构建系统相信这一点。细节请查阅 `src/pl/plpython` 目录中的 `Makefile`。

26.3. 使用 PL/Python

`plpython_function.sql` 中有示例函数。你编写的 Python 代码会被转换成一个函数。例如，

```
CREATE FUNCTION myfunc(text) RETURNS text AS
'return args[0]'
LANGUAGE 'plpython';
```

会被转换成

```
def __plpython_procedure_myfunc_23456():
 return args[0]
```

其中 23456 是该函数的 Oid。

[1] http://www.python.org
[2] http://www.python.org/doc/FAQ.html#3.30

如果你不提供返回值，Python 返回默认的 None，这可能符合也可能不符合你的需要。语言模块把 Python 的 None 转换成 SQL NULL。

PostgreSQL 的函数变量可以在全局 args 列表中得到。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 变量，args[1] 包含 integer 变量。

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，对一个后端中的所有 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受限执行对象，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。

当一个函数被用于触发器时，字典 TD 包含与事务相关的值。触发器元组在 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 或 UNKNOWN）之一。TD["level"] 包含 ROW、STATEMENT 或 UNKNOWN 之一。TD["name"] 包含触发器的名字，TD["relid"] 包含触发器发生所在表的关系 id。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。

如果触发器的 “when” 是 BEFORE，你可以从 Python 函数返回 None 或 "OK" 来表示该元组未被修改，"SKIP" 中止该事件，或者 "MODIFIED" 表示你修改了该元组。

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.error("msg")、plpy.fatal("msg")、plpy.debug("msg") 和 plpy.notice("msg")。它们大体上等价于调用 elog(*LEVEL*, "msg")，其中 *LEVEL* 是 DEBUG、ERROR、FATAL 或 NOTICE。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 解释器中长跳转出来可能不太好。raise plpy.ERROR("msg") 和 raise plpy.FATAL("msg") 等价于调用 plpy.error 或 plpy.fatal。

此外，plpy 模块提供两个分别名为 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字段名访问。它有这些额外的方法：nrows() 返回查询返回的行数，status 是 SPI_exec 的返回变量。结果对象可以被修改。

```
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_field，可以这样访问它：

```
foo = rv[i]["my_field"]
```

第二个函数 plpy.prepare 在查询中有绑定变量时用一个查询字符串和一个参数类型列表调用。

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name =
 $1", [ "text" ])
```

text 是你将作为 $1 传递的变量的类型。准备好之后你用函数 plpy.execute 来运行它。

```
rv = plpy.execute(plan, [ "name" ], 5)
```

调用 plpy.execute 时 limit 参数是可选的。

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 21 章）了解这意味着什么。要点是如果你这样做

```
plan = plpy.prepare("SOME QUERY")
plan = plpy.prepare("SOME OTHER QUERY")
```

你就在泄漏内存，因为我不知道有什么办法释放一个已保存的计划。
而使用未保存计划的替代方案（对我来说）更加痛苦。

PostgreSQL 7.2.8 参考手册

PostgreSQL 全球开发组

PostgreSQL 7.2.8 参考手册

PostgreSQL 全球开发组

法律声明

目录

前言 cccxxxvii
I. SQL 命令 338
ABORT 340
ALTER GROUP 341
ALTER TABLE 342
ALTER USER 346
ANALYZE 348
BEGIN 350
CHECKPOINT 352
CLOSE 353
CLUSTER 354
COMMENT 356
COMMIT 358
COPY 359
CREATE AGGREGATE 365
CREATE CONSTRAINT TRIGGER 367
CREATE DATABASE 368
CREATE FUNCTION 371
CREATE GROUP 375
CREATE INDEX 377
CREATE LANGUAGE 380
CREATE OPERATOR 382
CREATE RULE 386
CREATE SEQUENCE 389
CREATE TABLE 392
CREATE TABLE AS 400
CREATE TRIGGER 402
CREATE TYPE 404
CREATE USER 408
CREATE VIEW 410
DECLARE 412
DELETE 415
DROP AGGREGATE 417
DROP DATABASE 418
DROP FUNCTION 419
DROP GROUP 421
DROP INDEX 422
DROP LANGUAGE 423
DROP OPERATOR 424
DROP RULE 426
DROP SEQUENCE 427
DROP TABLE 428
DROP TRIGGER 430
DROP TYPE 431
DROP USER 433
DROP VIEW 434
END 436
EXPLAIN 437
FETCH 440
GRANT 443
INSERT 446

LISTEN 448
LOAD 450
LOCK 451
MOVE 456
NOTIFY 457
REINDEX 459
RESET 461
REVOKE 462
ROLLBACK 464
SELECT 465
SELECT INTO 475
SET 477
SET CONSTRAINTS 481
SET SESSION AUTHORIZATION 482
SET TRANSACTION 483
SHOW 485
TRUNCATE 487
UNLISTEN 488
UPDATE 490
VACUUM 492
II. PostgreSQL 客户端应用 495
createdb 496
createlang 498
createuser 500
dropdb 502
droplang 504
dropuser 506
ecpg 508
pgaccess 512
pg_config 514
pg_dump 516
pg_dumpall 521
pg_restore 523
psql 529
pgtclsh 550
pgtksh 551
vacuumdb 552
III. PostgreSQL 服务器应用 555
initdb 556
initlocation 558
ipcclean 559
pg_ctl 560
pg_passwd 563
postgres 564
postmaster 567

前言

本参考手册中的条目旨在以合理的篇幅提供关于相应主题的权威、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PostgreSQL 文档集的其他部分找到。请参阅每个参考页上列出的交叉引用。

参考手册条目也可以以传统的“man”页的形式获得。

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 340
ALTER GROUP 341
ALTER TABLE 342
ALTER USER 346
ANALYZE 348
BEGIN 350
CHECKPOINT 352
CLOSE 353
CLUSTER 354
COMMENT 356
COMMIT 358
COPY 359
CREATE AGGREGATE 365
CREATE CONSTRAINT TRIGGER 367
CREATE DATABASE 368
CREATE FUNCTION 371
CREATE GROUP 375
CREATE INDEX 377
CREATE LANGUAGE 380
CREATE OPERATOR 382
CREATE RULE 386
CREATE SEQUENCE 389
CREATE TABLE 392
CREATE TABLE AS 400
CREATE TRIGGER 402
CREATE TYPE 404
CREATE USER 408
CREATE VIEW 410
DECLARE 412
DELETE 415
DROP AGGREGATE 417
DROP DATABASE 418
DROP FUNCTION 419
DROP GROUP 421
DROP INDEX 422
DROP LANGUAGE 423
DROP OPERATOR 424
DROP RULE 426
DROP SEQUENCE 427
DROP TABLE 428
DROP TRIGGER 430
DROP TYPE 431
DROP USER 433
DROP VIEW 434

END 436
EXPLAIN 437
FETCH 440
GRANT 443
INSERT 446
LISTEN 448
LOAD 450
LOCK 451
MOVE 456
NOTIFY 457
REINDEX 459
RESET 461
REVOKE 462
ROLLBACK 464
SELECT 465
SELECT INTO 475
SET 477
SET CONSTRAINTS 481
SET SESSION AUTHORIZATION 482
SET TRANSACTION 483
SHOW 485
TRUNCATE 487
UNLISTEN 488
UPDATE 490
VACUUM 492

ABORT

ABORT — 中止当前事务

大纲

```
ABORT [ WORK | TRANSACTION ]
```

输入

无。

输出

`ROLLBACK`

成功时返回的消息。

`NOTICE: ROLLBACK: no transaction in progress`

当前没有正在进行的事务时给出。

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SQL92命令`ROLLBACK` 完全相同，其存在仅出于历史原因。

注解

使用`COMMIT`来成功终止一个事务。

用法

中止所有更改：

```
ABORT WORK;
```

兼容性

SQL92

这条命令是一个因历史原因而保留的PostgreSQL 扩展。`ROLLBACK`是等价的SQL92 命令。

ALTER GROUP

ALTER GROUP — 向组中添加用户或从组中移除用户

大纲

```
ALTER GROUP name ADD USER username [, ... ]
ALTER GROUP name DROP USER username [, ... ]
```

输入

name

要修改的组的名称。

username

要添加到组中或从组中移除的用户。这些用户名必须已经存在。

输出

`ALTER GROUP`

更改成功时返回的消息。

描述

`ALTER GROUP`用于向组中添加用户或从组中移除用户。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。

使用 CREATE GROUP 来创建一个新组，使用 DROP GROUP 来移除一个组。

用法

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

SQL92

`ALTER GROUP` 在 SQL92 中没有语句。角色的概念与之相似。

ALTER TABLE

ALTER TABLE — 更改一个表的定义

大纲

```
ALTER TABLE [ ONLY ] table [ * ]
    ADD [ COLUMN ] column type [ column_constraint [ ... ] ]
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column SET STATISTICS integer
ALTER TABLE [ ONLY ] table [ * ]
    RENAME [ COLUMN ] column TO newcolumn
ALTER TABLE table
    RENAME TO new_table
ALTER TABLE table
    ADD table_constraint_definition
ALTER TABLE [ ONLY ] table
 DROP CONSTRAINT constraint { RESTRICT | CASCADE }
ALTER TABLE table
 OWNER TO new_owner
```

输入

table

要更改的现有表的名称。

column

新列或现有列的名称。

type

新列的类型。

newcolumn

现有列的新名称。

new_table

表的新名称。

table_constraint_definition

表的新表约束

new_owner

表的新所有者的用户名。

输出

ALTER

列或表重命名后返回的消息。

ERROR

表或列不可用时返回的消息。

描述

ALTER TABLE 更改一个现有表的定义。ADD COLUMN 形式使用与CREATE TABLE相同的语法向表添加一个新列。ALTER COLUMN SET/DROP DEFAULT 形式允许你为列设置或移除默认值。注意，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。ALTER COLUMN SET STATISTICS 形式允许你为后续 ANALYZE 操作设置统计信息收集目标。RENAME 子句更改表、列、索引或序列的名称，而不更改任何数据。命令执行后，数据仍保持相同的类型和大小。ADD *table_constraint_definition* 子句使用与CREATE TABLE相同的语法向表增加一个新约束。DROP CONSTRAINT *constraint* 子句删除表（及其子表）上所有与 *constraint* 匹配的约束。OWNER 子句把表的所有者更改为用户 *new user*。

要更改表的模式，你必须拥有该表。

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。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。）

在 DROP CONSTRAINT 中，RESTRICT 关键字是必需的，但尚未检查依赖关系。CASCADE 选项不受支持。目前 DROP CONSTRAINT 只删除 CHECK 约束。要移除 PRIMARY 或 UNIQUE 约束，请用 DROP INDEX 命令删除相关的索引。要移除 FOREIGN KEY 约束，你需要用CREATE TABLE命令的其他参数重建并重新装载该表。

例如，要删除表 distributors 上的所有约束：

```
CREATE TABLE temp AS SELECT * FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors AS SELECT * FROM temp;
DROP TABLE temp;
```

要更改表，你必须拥有它。不允许更改系统目录模式的任何部分。PostgreSQL 用户指南中还有关于继承的更多信息。

有关有效参数的进一步说明，请参见CREATE TABLE。

用法

要添加类型为varchar的列到表中：

```
ALTER TABLE distributors ADD COLUMN address VARCHAR(30);
```

要重命名一个现有列：

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

要重命名一个现有的表：

```
ALTER TABLE distributors RENAME TO suppliers;
```

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 RESTRICT;
```

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

SQL92

`ADD COLUMN` 形式符合标准，但不支持默认值和 NOT NULL 约束，如上所述。`ALTER COLUMN` 形式完全符合标准。

SQL92 为 `ALTER TABLE` 语句规定了一些 PostgreSQL 尚不直接支持的附加能力：

```
ALTER TABLE table DROP [ COLUMN ] column { RESTRICT |
 CASCADE }
```

从表中移除一个列。目前，要移除一个现有列，必须重建并重新装载该表：

```
CREATE TABLE temp AS SELECT did, city FROM distributors;
```

```
DROP TABLE distributors;
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT 1,
    name     VARCHAR(40) NOT NULL
);
INSERT INTO distributors SELECT * FROM temp;
DROP TABLE temp;
```

重命名表、列、索引和序列的子句是 PostgreSQL 对 SQL92 的扩展。

ALTER USER

ALTER USER — 更改一个数据库用户账户

大纲

```
ALTER USER username [ [ WITH ] option [ ... ] ]

where option can be:

    [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
         | CREATEDB | NOCREATEDB
         | CREATEUSER | NOCREATEUSER
         | VALID UNTIL 'abstime'
```

输入

username

要更改其详细信息的用户的名称。

password

此帐号要使用的新密码。

ENCRYPTED
UNENCRYPTED

这些关键字控制口令是否以加密形式存储在 pg_shadow 中。（关于这一选择的更多信息见 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，被定义的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）。

输出

ALTER USER

更改成功时返回的消息。

```
ERROR: ALTER USER: user "username" does not exist
```

如果数据库不认识指定的用户，则返回此错误消息。

描述

`ALTER USER` 用于更改用户 PostgreSQL 账户的属性。命令中未提及的属性保持其原有设置。

只有数据库超级用户才能用此命令更改权限和口令过期。普通用户只能更改自己的口令。

`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 CREATE USER 创建新用户，使用 DROP USER 删除用户。

用法

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 1998 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 1998 +1';
```

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

SQL92

SQL92 中没有 `ALTER USER` 语句。该标准把用户的定义留给实现。

ANALYZE

ANALYZE — 收集关于数据库的统计信息

大纲

```
ANALYZE [ VERBOSE ] [ table [ (column [, ...] ) ] ]
```

输入

VERBOSE

启用进度消息的显示。

table

要分析的一个指定表的名称。默认为所有表。

column

要分析的一个指定列的名称。默认为所有列。

输出

→ ANALYZE

命令已完成。

描述

ANALYZE收集PostgreSQL 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，这种情况下只更新这些列的统计信息。

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。

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。

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用户指南。

分析的深度可以通过用 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所花费的时间和 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

SQL92

ANALYZE 在 SQL92 中没有语句。

BEGIN

BEGIN — 开始一个事务块

大纲

```
BEGIN [ WORK | TRANSACTION ]
```

输入

WORK
TRANSACTION

可选的关键字。它们没有效果。

输出

```
BEGIN
```

这表示一个新的事务已经启动。

```
NOTICE: BEGIN: already a transaction in progress
```

这表示已经有一个事务在进行中。当前事务不受影响。

描述

默认情况下，PostgreSQL 以 unchained mode（非链模式，在其他数据库系统中也称为 “autocommit”（自动提交））执行事务。换句话说，
每条用户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）。`BEGIN` 以链模式开启一个用户事务，也就是说，`BEGIN` 命令之后的所有用户语句都将在单个事务中执行，直到显式的 COMMIT、ROLLBACK 或执行中止。链模式中的语句执行得快得多，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对于在更改多个相关表时确保一致性也是必需的。

PostgreSQL 中的默认事务隔离级别是 READ COMMITTED，其中事务内的查询只看到在查询执行之前提交的更改。因此，如果你需要更严格的事务隔离，就必须使用 `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`，紧跟在 `BEGIN` 之后。在 SERIALIZABLE 模式中，查询只会看到在整个事务开始之前（实际上，是在一个可串行化事务中第一条 DML 语句执行之前）提交的更改。

如果事务被提交，PostgreSQL 将确保要么所有更新都完成，要么一个都不完成。事务具有标准的 ACID（原子性、一致性、隔离性、持久性）属性。

注意

关于在事务中锁定表的进一步信息，请参阅 LOCK。

使用 COMMIT 或 ROLLBACK 终止一个事务。

用法

开始一个用户事务：

```
BEGIN WORK;
```

兼容性

SQL92

`BEGIN` 是一种 PostgreSQL 语言扩展。没有显式的 `BEGIN` 命令出现在 SQL92 中；事务的开启总是隐式的，它以 `COMMIT` 或 `ROLLBACK` 语句终止。

> **注意**
>
> 许多关系数据库系统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。

SQL92 还要求 SERIALIZABLE 是默认的事务隔离级别。

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 PostgreSQL 管理员指南。

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。

另见

PostgreSQL 管理员指南

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 cursor
```

输入

cursor

要关闭的打开游标的名称。

输出

`CLOSE`

游标成功关闭时返回的消息。

`NOTICE PerformPortalClose: portal "`*cursor*`" not found`

如果 *cursor* 没有被声明或者已经被关闭，就会给出这条警告。

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。

当事务被 `COMMIT` 或 `ROLLBACK` 终止时，每个打开的游标都会被隐式关闭。

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。

用法

关闭游标`liahona`:

```
CLOSE liahona;
```

兼容性

SQL92

`CLOSE` 与 SQL92 完全兼容。

CLUSTER

CLUSTER — 根据一个索引对某个表进行聚簇

大纲

```
CLUSTER indexname ON tablename
```

输入

indexname

一个索引的名称。

table

一个表的名称。

输出

CLUSTER

聚簇已成功完成。

ERROR: relation <*tablerelation_number*> inherits "*table*"

ERROR: Relation *table* does not exist!

描述

CLUSTER指示 PostgreSQL近似地基于 *indexname*所指定的索引，对 *table*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。聚簇是静态的。换句话说，随着表被更新，这些更改不会被聚簇。系统不会尝试让新实例或更新过的元组保持聚簇。如果需要，可以通过再次发出该命令来手动重新聚簇。

注意

表实际上被按索引顺序复制到一个临时表中，然后改回原来的名称。因此，执行聚簇时会丢失所有 GRANT 权限和其他索引。

如果你在表中随机访问单个行，数据在堆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。

CLUSTER有帮助的另一个场合是使用索引从一个表中取出多行时。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。

有两种聚簇数据的方法。第一种是使用 CLUSTER命令，它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。

另一种聚簇数据的方法是使用

```
SELECT columnlist INTO TABLE newtable
     FROM table ORDER BY columnlist
```

它在 ORDER BY 子句中使用 PostgreSQL的排序代码来匹配索引，对于无序数据它要快得多。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。唯一的问题是 OID 不会被保留。此后，CLUSTER应该会很快，因为堆数据大部分已经有序，并且使用的是现有的索引。

用法

基于其 salary 属性聚簇 employees 关系：

```
CLUSTER emp_ind ON emp;
```

兼容性

SQL92

SQL92 中没有CLUSTER语句。

COMMENT

COMMENT — 定义或更改对象的注释

大纲

```
COMMENT ON
[
  [ DATABASE | INDEX | RULE | SEQUENCE | TABLE | TYPE | VIEW ] object_
name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type) |
  FUNCTION func_name (arg1, arg2, ...) |
  OPERATOR op (leftoperand_type rightoperand_type) |
  TRIGGER trigger_name ON table_name
] IS 'text'
```

输入

object_name, *table_name*, *column_name*, *agg_name*, *func_name*, *op*, *trigger_name*

要加注释的对象的名称。

text

要添加的注释。

输出

COMMENT

表成功加上注释时返回的消息。

描述

COMMENT 存储关于某个数据库对象的注释。注释可以用 psql 的 \dd、\d+ 或 \l+ 命令轻松检索。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。

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。

用法

给表 mytable 加注释：

```
COMMENT ON mytable IS 'This is my table.';
```

更多示例：

```
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee id';
COMMENT ON RULE my_rule IS 'Logs UPDATES of employee records';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON TABLE my_table IS 'Employee Information';
COMMENT ON TYPE my_type IS 'Complex Number support';
COMMENT ON VIEW my_view IS 'View of departmental costs';
COMMENT ON COLUMN my_table.my_field IS 'Employee ID number';
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes
 sample variance';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman
 Numeral';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two
 text';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for R.I.';
```

兼容性

SQL92

SQL92 中没有 COMMENT。

COMMIT

COMMIT — 提交当前事务

大纲

```
COMMIT [ WORK | TRANSACTION ]
```

输入

WORK
TRANSACTION

可选的关键字。它们没有任何作用。

输出

```
COMMIT
```

事务成功提交时返回的消息。

```
NOTICE: COMMIT: no transaction in progress
```

如果当前没有正在进行的事务。

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。

注意

关键字 WORK 和 TRANSACTION 只是噪音，可以省略。

使用 ROLLBACK 中止一个事务。

用法

要使所有更改永久生效：

```
COMMIT WORK;
```

兼容性

SQL92

SQL92 只规定了 `COMMIT` 和 `COMMIT WORK` 两种形式。其他方面完全兼容。

COPY

COPY — 在文件和表之间复制数据

大纲

```
COPY [ BINARY ] table [ WITH OIDS ]
    FROM { 'filename' | stdin }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
COPY [ BINARY ] table [ WITH OIDS ]
    TO { 'filename' | stdout }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
```

输入

BINARY

更改字段格式化的行为，强制所有数据以二进制而不是文本格式存储或读取。DELIMITERS 和 WITH NULL 选项与二进制格式无关。

table

一个现有表的名称。

WITH OIDS

指定为每一行复制内部对象 id（OID）。

filename

输入或输出文件的绝对 Unix 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。

delimiter

分隔文件每行（一行数据）中各字段的字符。

null string

表示 NULL 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 in 中，任何匹配此字符串的数据项都会被存储为 NULL 值，所以应确保使用与 copy out 相同的字符串。

输出

`COPY`

复制成功完成。

`ERROR:` *reason*

复制因错误消息中所述的原因而失败。

描述

`COPY` 在 PostgreSQL 表和标准文件系统文件之间移动数据。`COPY TO` 把一个表的全部内容复制到文件，而 `COPY FROM` 把数据从文件复制到表（把数据追加到表中已有内容之后）。

`COPY` 带文件名时指示 PostgreSQL 后端直接读取或写入文件。该文件必须可由后端访问，且名称必须从后端的角度指定。当指定 `stdin` 或 `stdout` 时，数据通过客户端前端流经后端。

> **提示**
>
> 不要把 `COPY` 与 psql 指令 `\copy` 混淆。`\copy` 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后端。

注解

`COPY`只能用于普通表，不能用于视图。

BINARY 关键字强制所有数据以二进制而不是文本格式存储/读取。它比普通的复制命令略快，但二进制复制文件不能跨机器体系结构移植。

默认情况下，文本复制使用制表符（"\t"）作为字段之间的分隔符。可以用关键字短语 USING DELIMITERS 把字段分隔符改为任何其他单个字符。数据字段中恰好匹配分隔符的字符将被反斜线引用。

你必须对其值被 `COPY` 读取的任何表拥有 select 访问权限，并对正被 `COPY` 插入值的表拥有 insert 或 update 访问权限。对于被 `COPY` 读取或写入的任何文件，后端还需要适当的 Unix 权限。

`COPY TO` 既不调用规则也不作用于列默认值。它确实会调用触发器和检查约束。

`COPY` 在第一个错误处停止操作。这对 `COPY FROM` 不会导致问题，但在 `COPY TO` 中目标关系已经接收了较早的行。这些行不可见也不可访问，但它们仍占用磁盘空间。如果失败发生在大型复制操作的后期，这可能造成相当大的磁盘空间浪费。你可能希望调用 `VACUUM` 来回收浪费的空间。

`COPY` 命令中命名的文件由后端而不是客户端应用直接读取或写入。因此，它们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。带文件名的 `COPY` 只允许数据库超级用户使用，因为它允许读取或写入后端有权访问的任何文件。

> **提示**
>
> psql 指令 `\copy` 以客户端的权限读写客户端机器上的文件，所以它不限于超级用户。

建议 COPY 中使用的文件名总是指定为绝对路径。在 COPY TO 的情况下这是由后端强制的，但对 COPY FROM 你确实可以选择从相对路径指定的文件读取。该路径将相对于后端的工作目录（$PGDATA 之下的某处）而不是客户端的工作目录解释。

文件格式

文本格式

当 COPY 不带 BINARY 选项使用时，读取或写入的文件是一个文本文件，每个表行占一行。一行中的各列（属性）由分隔字符分隔。
属性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。
指定的空值字符串用于代替为 NULL 的属性。

如果指定了 WITH OIDS，OID 将作为第一列读取或写入，位于用户数据列之前。（如果对没有 OID 的表指定 WITH OIDS 则会报错。）

数据结束可以用只包含反斜线加句点（\.）的单行表示。从 Unix 文件读取时不需要数据结束标记，因为文件结尾已经足够；但在向客户端应用复制数据或从其复制数据时必须提供结束标记。

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，以下字符如果作为属性值的一部分出现，前面必须有反斜线：反斜线自身、
换行符和当前的分隔字符。

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。

绝不要在数据字符 N 或句点（.）前面放反斜线。这样的组合会分别被误认为默认的空值字符串或数据结束标记。上表中未提及的任何其他反斜线引用的字符将被当作它自身。

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（PostgreSQL 7.2 及更早版本）可以不加任何特殊引用地表示数据回车符，用反斜线加换行符表示数据换行符。但在未来的版本中这些表示默认将不被接受。

注意每一行的结尾由 Unix 风格的换行符（"\n"）标记。目前，如果给 COPY FROM 一个包含 DOS 或 Mac 风格换行符的文件，它不会按期望的方式工作。这预计在未来的版本中改变。

二进制格式

COPY BINARY 使用的文件格式在 PostgreSQL v7.1 中改变了。新格式由文件头、零个或多个元组和文件尾组成。

文件头

文件头由 24 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

12 字节序列 `PGBCOPY\n\377\r\n\0` --- 注意，空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。换行翻译过滤器、丢弃的空字节、丢弃的高位或奇偶校验位的改变都会改变这个签名。）

整数布局字段

按源字节序的 int32 常量 0x01020304。如果在此处检测到错误的字节序，读取者原则上可以对后续字段进行字节翻转。

标志域

表示文件格式重要方面的 int32 位掩码。位从 0（LSB）到 31（MSB）编号--注意此字段按源端序存储，后续所有整数字段也是如此。位 16-31 保留用于表示关键的文件格式问题；如果读取者在此范围内发现意外置位的位，应当中止。位 0-15 保留用于发出向后兼容格式问题的信号；读取者应简单地忽略此范围内任何意外置位的位。目前只定义了一个标志位，其余必须为零：

Bit 16

为 1 表示转储中包含 OID；为 0 表示不包含

头部扩展区长度

文件头其余部分的 int32 字节长度，不包括自身。在初始版本中它为零，第一个元组紧随其后。未来的格式更改可能允许文件头中存在附加数据。
读取者应悄悄跳过它不知道如何处理的任何文件头扩展数据。

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

每个元组以一个 int16 的元组字段数开始。（目前表中所有元组都有相同的计数，但将来不一定总是如此。）然后对元组中的每个字段重复：一个 int16 的 typlen 字，后面可能跟随字段数据。typlen 字段这样解释：

Zero

字段为 NULL。没有后续数据。

> 0

字段是定长数据类型。typlen 字后面恰好跟随 N 字节数据。

-1

字段是 varlena 数据类型。接下来的四个字节是 varlena 头，它包含包括自身在内的总值长度。

< -1

保留供将来使用。

对于非 NULL 字段，读取者可以检查 typlen 与目标列的预期 typlen 匹配。这提供了一种简单但非常有用的检查，确保数据符合预期。

字段之间没有对齐填充或任何其他额外数据。
还请注意该格式不区分数据类型是按引用传递还是按值传递。
这两项规定都是有意为之：它们可能有助于提高文件的可移植性（当然字节序和浮点格式问题仍然可能阻止你跨机器移动二进制文件）。

如果转储中包含 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 typlen−−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 NULL。

文件尾

文件尾由一个包含 -1 的 int16 字组成。这很容易与元组的字段计数字区分。

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。

用法

下面的例子把一个表复制到标准输出，使用竖线（|）作为字段分隔符：

```
COPY country TO stdout USING DELIMITERS '|';
```

把数据从一个 Unix 文件复制到表 country：

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

下面是一段适合从 `stdin` 复制到表中的数据示例（所以最后一行有终止序列）：

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
\.
```

注意每行上的空白实际上是一个制表符（TAB）。

下面是相同的数据在一台 Linux/i586 机器上以二进制格式输出的样子。数据经过 Unix 工具 `od -c` 过滤后显示。该表有三个字段；第一个是 `char(2)`，第二个是 `text`，第三个是 `integer`。所有行的第三个字段都是空值。

```
0000000   P   G   B   C   O   P   Y  \n 377  \r  \n  \0 004 003 002
 001
0000020  \0  \0  \0  \0  \0  \0  \0  \0 003  \0 377 377 006  \0  \0
 \0
```

```
0000040   A   F 377 377 017  \0  \0  \0   A   F   G   H   A   N   I
 S
0000060   T   A   N  \0  \0 003  \0 377 377 006  \0  \0  \0   A   L
 377
0000100 377  \v  \0  \0  \0   A   L   B   A   N   I   A  \0  \0 003
 \0
0000120 377 377 006  \0  \0  \0   D   Z 377 377  \v  \0  \0  \0   A
 L
0000140   G   E   R   I   A  \0  \0 003  \0 377 377 006  \0  \0  \0
 Z
0000160   M 377 377  \n  \0  \0  \0   Z   A   M   B   I   A  \0  \0
 003
0000200  \0 377 377 006  \0  \0  \0   Z   W 377 377  \f  \0  \0  \0
 Z
0000220   I   M   B   A   B   W   E  \0  \0 377 377
```

兼容性

SQL92

SQL92 中没有 `COPY` 语句。

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_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ] )
```

输入

name

要创建的聚合函数的名称。

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，可以将它指定为 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_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_type*的参数。
无论哪种情况，该函数都必须返回*state_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_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_type*的参数。
聚合的输出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，输出类型为*state_type*。

initial_condition

状态值的初始设置。这必须是*state_type*数据类型所接受形式的字面常量。如果未指定，
状态值将从 NULL 开始。

输出

CREATE

命令成功完成时返回的消息。

描述

CREATE AGGREGATE 允许用户或程序员通过定义新的聚合函数来扩展 PostgreSQL 的功能。基本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。

聚合函数由其名称和输入数据类型标识。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。为避免混淆，
不要创建与某个聚合同名且具有相同输入数据类型的普通函数。

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输出值。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类型的字段的形式指定和存储，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 NULL。

如果状态转移函数被声明为“strict”的，那么它不能以 NULL 输入被调用。使用这种转移函数时，聚合的执行行为如下。NULL 输入值会被忽略（不调用函数并保留先前的状态值）。如果初始状态值为 NULL，那么第一个非 NULL 输入值会取代状态值，而转移函数从第二个非 NULL 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_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 NULL 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 NULL 输入和 NULL 转移值。这使聚合作者能完全控制聚合对 NULL 的处理。

如果最终函数被声明为“strict”，那么当结束状态值为 NULL 时不会调用它；而是自动输出 NULL 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 NULL。例如，avg的最终函数在发现输入元组数为零时会返回 NULL。

注意

使用DROP AGGREGATE 删除聚合函数。

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。

用法

完整的使用示例请参阅PostgreSQL 程序员指南中关于聚合函数的章节。

兼容性

SQL92

CREATE AGGREGATE 是 PostgreSQL 的语言扩展。SQL92 中没有CREATE AGGREGATE。

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
    relation constraint attributes
    FOR EACH ROW EXECUTE PROCEDURE func '(' args ')'
```

输入

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。

relation

触发关系的表名。

constraint

实际的约束说明。

attributes

约束属性。

func(*args*)

作为触发器处理的一部分调用的函数。

输出

`CREATE CONSTRAINT`

成功时返回的消息。

描述

`CREATE CONSTRAINT TRIGGER` 被 `CREATE/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。

CREATE DATABASE

CREATE DATABASE — 创建一个新数据库

大纲

```
CREATE DATABASE name
    [ WITH [ LOCATION = 'dbpath' ]
           [ TEMPLATE = template ]
           [ ENCODING = encoding ] ]
```

输入

name

要创建的数据库的名称。

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多字节编码方法。指定一个字符串字面量名称（例如 'SQL_ASCII'）、一个整数编码号，或者 DEFAULT 表示使用默认编码。

输出

CREATE DATABASE

命令成功完成时返回的消息。

ERROR: user '*username*' is not allowed to create/drop databases

你必须拥有特殊的 CREATEDB 权限才能创建数据库。见 CREATE USER。

ERROR: createdb: database "*name*" already exists

如果已经存在一个具有所指定 *name* 的数据库，就会出现这个错误。

ERROR: database path may not contain single quotes

数据库位置 *dbpath* 不能包含单引号。这是为了确保创建数据库目录的 shell 命令能够安全执行而要求的。

ERROR: CREATE DATABASE: may not be called in a transaction block

如果有一个显式的事务块正在进行中，你就不能调用 CREATE DATABASE。必须先完成该事务。

```
ERROR: Unable to create database directory 'path'.
ERROR: Could not initialize database directory.
```

这些错误很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。
运行数据库服务器的用户必须能访问该位置。

描述

`CREATE DATABASE` 创建一个新的 PostgreSQL 数据库。创建者成为新数据库的拥有者。

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。

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（如果你的服务器编译时带了多字节编码支持）。
未指定时，它默认为所选模板数据库使用的编码。

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` 是一种 PostgreSQL 语言扩展。

使用 DROP DATABASE 删除一个数据库。

程序createdb是围绕这个命令的 shell 脚本包装，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和数据完整性问题，默认情况下，
只有后端已知的环境变量才能被指定为备选位置。更多信息见《管理员指南》。

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。更多信息见管理员指南。

用法

创建一个新的数据库：

```
olly=>create database lusiadas;
```

要在备选区域 `~/private_db` 中创建一个新数据库：

```
$mkdir private_db$initlocation ~/private_dbThe location will be
 initialized with username "olly".
This user will own all the files and must also own the server process.
Creating directory /home/olly/private_db
```

```
Creating directory /home/olly/private_db/base

initlocation is complete.
    $psql ollyWelcome to psql, 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

olly=>CREATE DATABASE elsewhere WITH LOCATION = '/home/olly/private_
db';CREATE DATABASE
```

兼容性

SQL92

SQL92 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 FUNCTION

CREATE FUNCTION — 定义一个新函数

大纲

```
CREATE [ OR REPLACE ] FUNCTION name ( [ argtype [, ...] ] )
    RETURNS rettype
    AS 'definition'
    LANGUAGE langname
    [ WITH ( attribute [, ...] ) ]
CREATE [ OR REPLACE ] FUNCTION name ( [ argtype [, ...] ] )
    RETURNS rettype
    AS 'obj_file', 'link_symbol'
    LANGUAGE langname
    [ 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。

参数

name

要创建的函数的名称。该名称不必唯一，因为函数可以重载，但同名函数必须有不同的参数类型。

argtype

函数参数的数据类型（如果有的话）。输入类型可以是基本类型或复合类型、opaque，或者与现有列的类型相同。Opaque 表示函数接受非 SQL 类型的参数，例如 char *。用*tablename*.*columnname*%TYPE的形式来指示一个列的类型；有时使用这种写法可以帮助让函数独立于表定义的变化。

rettype

返回数据类型。输出类型可以指定为基本类型、复合类型、setof 类型、opaque，或者与现有列的类型相同。setof 修饰符表示该函数将返回一组结果项，而不是单个项。声明返回类型为 opaque的函数不返回值。这类函数不能被直接调用；触发器函数利用了这个特性。

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 查询，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，动态链接的 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*是对象的链接符号，也就是 C 语言源代码中函数的名称。

langname

可以是SQL、C、internal，或者*plname*，其中*plname*是一种已创建过程语言的名称。详见CREATE LANGUAGE。为了向后兼容，该名称可以用单引号包围。

attribute

关于函数的一条可选信息，用于优化。详见下文。

创建该函数的用户将成为该函数的拥有者。

下列属性可以出现在 WITH 子句中：

iscachable

Iscachable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）。优化器利用iscachable来判断预计算该函数的一次调用是否安全。

isstrict

isstrict表示只要任一参数为 NULL，函数就总是返回 NULL。如果指定了这个属性，当存在 NULL 参数时函数不会被执行；而是自动假定结果为 NULL。当未指定isstrict时，函数将针对 NULL 输入被调用。此时如有必要，由函数作者负责检查 NULL 并做出适当响应。

注意

关于编写外部函数的更多信息，请参考 PostgreSQL 程序员指南中关于通过函数扩展 PostgreSQL 主题的章节。

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。但对 internal 和 C 语言函数，必须谨慎使用这一设施。

两个internal 函数如果 C 名相同，会在链接时引发错误。要解决这个问题，可以给它们起不同的 C 名（例如，把参数类型用作 C 名的一部分），然后在CREATE FUNCTION的 AS 子句中指定这些名字。如果 AS 子句留空，则CREATE FUNCTION 假定函数的 C 名与 SQL 名相同。

类似地，当用多个 C 语言函数重载 SQL 函数名时，给函数的每个 C 语言实例起一个不同的名字，然后在 CREATE FUNCTION语法中使用AS子句的另一种形式，为每个重载的 SQL 函数选择合适的 C 语言实现。

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。

要更新现有函数的定义，可以使用CREATE OR REPLACE FUNCTION。注意，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。

示例

创建一个简单的 SQL 函数：

```
CREATE FUNCTION one() RETURNS integer
    AS 'SELECT 1 AS RESULT;'
    LANGUAGE SQL;

SELECT one() AS answer;
 answer
--------
      1
```

下一个示例通过调用一个用户创建的名为`funcs.so`的共享库（扩展名可能因平台而异）中的例程来创建一个 C 函数。共享库文件会在服务器的动态库搜索路径中寻找。这个特定的例程计算一个校验位，如果函数参数中的校验位正确就返回 TRUE。它用于 CHECK 约束中。

```
CREATE FUNCTION ean_checkdigit(char, char) RETURNS boolean
    AS 'funcs' LANGUAGE C;

CREATE TABLE product (
    id        char(8) PRIMARY KEY,
    eanprefix char(8) CHECK (eanprefix ~ '[0-9]{2}-[0-9]{5}')
                      REFERENCES brandname(ean_prefix),
    eancode   char(6) CHECK (eancode ~ '[0-9]{6}'),
    CONSTRAINT ean    CHECK (ean_checkdigit(eanprefix, eancode))
);
```

这个示例创建一个在用户定义类型 complex 与内部类型 point 之间做类型转换的函数。该函数由一个从 C 源码编译出的动态载入对象实现（我们展示指定共享对象文件确切路径名这种现已废弃的替代方式）。为了让PostgreSQL自动找到类型转换函数，SQL 函数必须与返回类型同名，因此重载不可避免。在 SQL 定义中使用`AS`子句的第二种形式来重载函数名：

```
CREATE FUNCTION point(complex) RETURNS point
    AS '/home/bernie/pgsql/lib/complex.so', 'complex_to_point'
    LANGUAGE C;
```

该函数的 C 声明可以是：

```
Point * complex_to_point (Complex *z)
{
 Point *p;

 p = (Point *) palloc(sizeof(Point));
 p->x = z->x;
 p->y = z->y;

 return p;
}
```

兼容性

`CREATE FUNCTION`命令在 SQL99 中定义。PostgreSQL的版本与之类似但不兼容。属性不可移植，各种可用语言也一样。

另见

DROP FUNCTION , LOAD, PostgreSQL 程序员指南

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 [, ...]
```

输入

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。

输出

CREATE GROUP

命令成功完成时返回的消息。

描述

CREATE GROUP 将在数据库安装中创建一个新组。关于使用组进行认证的信息请参考管理员指南。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。

用法

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

SQL92

SQL92 中没有 `CREATE GROUP` 语句。角色（roles）在概念上与组类似。

CREATE INDEX

CREATE INDEX — 定义一个新索引

大纲

```
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_method ] ( column [ ops_name ] [, ...] )
    [ WHERE predicate ]
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_method ] ( func_name( column [, ... ]) [ ops_name ] )
    [ WHERE predicate ]
```

输入

UNIQUE

使系统在创建索引时（如果数据已存在）以及每次添加数据时检查表中的重复值。
试图插入或更新会导致重复项的数据将产生一个错误。

index_name

要创建的索引名称。

table

要建立索引的表名。

acc_method

用于索引的访问方法的名称。默认访问方法是 `BTREE`。PostgreSQL 为索引提供了四种访问方法：

BTREE

Lehman-Yao 高并发 B-树的一个实现。

RTREE

使用 Guttman 的二次分裂算法实现标准 R-tree。

HASH

Litwin 线性散列的一个实现。

GIST

广义索引搜索树。

column

表中一个列的名称。

ops_name

一个关联的操作符类。详见下文。

func_name

一个返回可索引值的函数。

predicate

定义部分索引的约束表达式。

输出

CREATE

索引成功创建时返回的消息。

ERROR: Cannot create index: 'index_name' already exists.

无法创建索引时出现此错误。

描述

CREATE INDEX 在指定的 *table* 上构建一个索引 *index_name*。

> **提示**
>
> 索引主要用于提升数据库性能。但使用不当会导致性能下降。

在上面所示的第一种语法中，索引的键字段以列名指定。如果索引访问方法支持多列索引，
则可以指定多个字段。

在上面所示的第二种语法中，索引定义在把用户指定的函数 *func_name* 应用于单个表的一个或多个列所得到的结果上。这些函数索引可用于基于操作符的快速数据访问，
而这些操作符通常需要对基础数据做某种变换才能应用。

PostgreSQL 为索引提供了 B-tree、R-tree、hash 和 GiST 访问方法。B-tree 访问方法是 Lehman-Yao 高并发 B-树的一个实现。R-tree 访问方法使用 Guttman 的二次分裂算法实现标准 R-tree。hash 访问方法是 Litwin 线性散列的一个实现。我们提到这些算法只是为了说明所有这些访问方法都是完全动态的，不需要周期性地进行优化（例如静态 hash 访问方法就需要）。

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有价值。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 SELECT 和聚合表达式。

所有在索引定义中使用的函数和操作符都必须是可缓存的，也就是说，
它们的结果只能依赖其输入参数，
而绝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。要在索引中使用用户定义的函数，
记得在创建该函数时将其标记为可缓存的。

使用 DROP INDEX 删除索引。

注解

每当一个被索引的属性出现在使用下列操作符之一的比较中时，PostgreSQL 的查询优化器就会考虑使用 B-tree 索引：<, <=, =, >=, >

每当一个被索引的属性出现在使用下列操作符之一的比较中时，PostgreSQL 的查询优化器就会考虑使用 R-tree 索引：<<, &<, &>, >>, @, ~=, &&

每当一个被索引的属性出现在使用 = 操作符的比较中时，PostgreSQL 的查询优化器就会考虑使用 hash 索引。

目前，只有 B-tree 和 gist 访问方法支持多列索引。默认最多可指定 16 个键（构建 PostgreSQL 时可以更改此限制）。目前只有 B-tree 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-tree 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，字段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种有意义的排序方式。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，然后在创建索引时选择合适的类。还有一些具有特殊用途的操作符类：

- 操作符类 `box_ops` 和 `bigbox_ops` 都支持在 `box` 数据类型上建立 R-tree 索引。它们之间的区别是 `bigbox_ops` 会将 box 坐标缩小，以避免对非常大的浮点坐标做乘法、加法和减法时出现浮点异常。（注意：这在以前是成立的，但目前这两个操作符类都使用浮点，
  实际上是等同的。）

下面的查询列出所有已定义的操作符类：

```
SELECT am.amname AS acc_method,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    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 acc_method, ops_name, ops_comp;
```

用法

要在表 `films` 的字段 `title` 上创建一个 B-tree 索引：

```
CREATE UNIQUE INDEX title_idx
    ON films (title);
```

兼容性

SQL92

CREATE INDEX 是 PostgreSQL 的语言扩展。

SQL92 中没有 `CREATE INDEX` 命令。

CREATE LANGUAGE

CREATE LANGUAGE — 定义一种新的过程语言

大纲

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE langname
    HANDLER call_handler
```

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Programmer's Guide。

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该关键字，
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。

langname

新过程语言的名称。语言名不区分大小写。过程语言不能覆盖 PostgreSQL的内建语言之一。

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 opaque类型的函数注册到 PostgreSQL中，这个类型是为未指定或未定义类型准备的占位类型。

诊断

CREATE

如果语言成功创建，就返回此消息。

ERROR: PL handler function *funcname*() doesn't exist

如果找不到函数 *funcname*()，就返回此错误。

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脚本，它还会安装正确的调用处理器。（createlang会在内部调用CREATE LANGUAGE。）

使用CREATE FUNCTION命令创建新函数。

使用DROP LANGUAGE，或者更好的是droplang脚本，来删除过程语言。

系统目录pg_language记录当前已安装过程语言的信息。

```
        Table "pg_language"
   Attribute   |  Type   | Modifier
---------------+---------+----------
 lanname       | name    |
 lanispl       | boolean |
 lanpltrusted  | boolean |
 lanplcallfoid | oid     |
 lancompiler   | text    |

   lanname   | lanispl | lanpltrusted | lanplcallfoid | lancompiler
-------------+---------+--------------+---------------+-------------
 internal    | f       | f            |             0 | n/a
 C           | f       | f            |             0 | /bin/cc
 sql         | f       | f            |             0 | postgres
```

目前，过程语言的定义一旦创建就不能更改。

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 opaque
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

兼容性

CREATE LANGUAGE是 PostgreSQL扩展。

历史

CREATE LANGUAGE命令最早出现在 PostgreSQL 6.3 中。

另见

createlang, CREATE FUNCTION, droplang, DROP LANGUAGE, PostgreSQL 程序员指南

CREATE OPERATOR

CREATE OPERATOR — 定义一个新的操作符

大纲

```
CREATE OPERATOR name ( PROCEDURE = func_name
     [, LEFTARG = lefttype
     ] [, RIGHTARG = righttype ]
     [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
     [, RESTRICT = res_proc ] [, JOIN = join_proc ]
     [, HASHES ] [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op ]
 )
```

输入

name

要定义的操作符。允许的字符见下文。

func_name

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。

left_sort_op

如果此操作符可以支持归并连接，则为对该操作符左侧数据类型排序的操作符。

right_sort_op

如果此操作符可以支持归并连接，则为对该操作符右侧数据类型排序的操作符。

输出

CREATE

操作符创建成功时返回的消息。

描述

CREATE OPERATOR 定义一个新的操作符 *name*。定义操作符的用户将成为其所有者。

操作符的 *name* 是一个最多由 NAMEDATALEN-1（默认 31）个字符组成的序列，这些字符取自下面的列表：

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

对名称的选择有若干限制：

- $ 不能定义为单字符操作符，尽管它可以是多字符操作符名称的一部分。
- --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 *- 不是。这一限制使 PostgreSQL 能够解析符合 SQL 的查询而无需在记号之间加空格。

> **注意**
>
> 使用非 SQL 标准的操作符名称时，你通常需要用空格分隔相邻的操作符以避免歧义。例如，如果你定义了一个名为 @ 的左一元操作符，你就不能写 X*@Y；你必须写 X* @Y，以确保 PostgreSQL 把它读成两个操作符名而不是一个。

输入!=时会被映射为<>，因此这两个名称始终等效。

LEFTARG 和 RIGHTARG 至少必须定义一个。对于二元操作符，两者都应定义。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。

过程 *func_name* 必须此前已用 CREATE FUNCTION 定义，并且必须定义为接受正确数量的参数（一个或两个），类型为所指明的类型。

如果存在交换子操作符，就应当把它指出来，这样 PostgreSQL 就可以在需要时反转操作数的顺序。例如，面积小于操作符 <<< 大概会有一个交换子操作符--面积大于 >>>。于是，查询优化器就可以自由地把：

```
box '((0,0), (1,1))'  >>> MYBOXES.description
```

转换为

```
MYBOXES.description <<< box '((0,0), (1,1))'
```

这使执行代码总能使用后一种表示，并在一定程度上简化了查询优化器。

类似地，如果存在求反器操作符，也应当把它指出来。假设存在一个操作符--面积等于 ===，同时也存在面积不等于 !==。求反器链接使查询优化器可以把

```
NOT MYBOXES.description === box '((0,0), (1,1))'
```

化简为

```
MYBOXES.description !== box '((0,0), (1,1))'
```

如果给出了交换子操作符的名称，PostgreSQL 会在目录中搜索它。如果找到了，而它自己还没有交换子，那么该交换子的条目就会被更新，把新创建的操作符作为其交换子。
这对求反器同样适用。
这样做的目的是允许定义两个互为交换子或互为求反器的操作符：第一个操作符应当不带交换子或求反器（视情况而定）先定义；定义第二个操作符时，把第一个指名为交换子或求反器，
第一个就会作为副作用被更新。（从 PostgreSQL 6.5 起，让两个操作符互相引用也可以。）

HASHES、SORT1 和 SORT2 选项的存在是为了支持查询优化器执行连接。PostgreSQL 总是能通过迭代替换 [WONG76] 求一个连接的值（即处理一个由返回 boolean 的操作符分隔两个元组变量的子句）。此外，PostgreSQL 还可以使用类似 [SHAP86] 的哈希连接算法；但它必须知道这一策略是否适用。当前的哈希连接算法只对表示相等测试的操作符是正确的；而且，数据类型的相等必须意味着类型表示的按位相等。（例如，
一个包含对相等测试无关紧要的未用位的数据类型就不能做哈希连接。）HASHES 标志向查询优化器指明，此操作符可以安全地用于哈希连接。

类似地，两个排序操作符向查询优化器指明归并排序是否是可用的连接策略，
以及应该用哪些操作符来排序两个操作数类。只应为相等操作符提供排序操作符，
而且它们应分别指向左右两侧数据类型的小于操作符。

如果将来发现其他连接策略切实可行，PostgreSQL 会修改优化器和运行时系统来使用它们，并在定义操作符时要求额外的说明。幸运的是，研究界并不经常发明新的连接策略，
而用户自定义连接策略带来的额外通用性被认为不值得为之增加复杂度。

RESTRICT 和 JOIN 选项帮助查询优化器估算结果大小。如果限定条件中出现了形如：

```
MYBOXES.description <<< box '((0,0), (1,1))'
```

的子句，那么 PostgreSQL 可能必须估算 MYBOXES 中满足该子句的实例比例。函数 *res_proc* 必须是一个已注册的函数（即已用 CREATE FUNCTION 定义），它接受正确数据类型的参数并返回一个浮点数。查询优化器只需调用该函数，传入参数 ((0,0), (1,1))，再把结果乘以关系的大小，就得到预期的实例数。

类似地，当操作符的两个操作数都含有实例变量时，查询优化器必须估算所得连接的大小。函数 join_proc 会返回另一个浮点数，把它乘以所涉及两个表的基数，就得到预期的结果大小。

函数

```
my_procedure_1 (MYBOXES.description, box '((0,0), (1,1))')
```

与操作符

```
MYBOXES.description === box '((0,0), (1,1))'
```

的区别在于：PostgreSQL　会尝试优化操作符，并可在涉及操作符时决定使用索引来限制搜索空间。但系统不会尝试优化函数，函数是靠蛮力执行的。此外，函数可以有任意数量的参数，而操作符只能有一个或两个。

注意

更多信息参见 PostgreSQL 用户指南中关于操作符的章节。参见 `DROP OPERATOR` 以从数据库中删除用户定义的操作符。

用法

下面的命令为 BOX 数据类型定义一个新的操作符——面积相等：

```
CREATE OPERATOR === (
   LEFTARG = box,
   RIGHTARG = box,
   PROCEDURE = area_equal_procedure,
   COMMUTATOR = ===,
   NEGATOR = !==,
   RESTRICT = area_restriction_procedure,
   JOIN = area_join_procedure,
   HASHES,
   SORT1 = <<<,
   SORT2 = <<<
);
```

兼容性

SQL92

`CREATE OPERATOR` 是 PostgreSQL 的扩展。在 SQL92 中没有 `CREATE OPERATOR` 语句。

CREATE RULE

CREATE RULE — 定义一条新的重写规则

大纲

```
CREATE RULE name AS ON event
    TO object [ WHERE condition ]
    DO [ INSTEAD ] action

where action can be:

NOTHING
|
query
|
( query ; query ... )
|
[ query ; query ... ]
```

输入

name

要创建的规则的名称。

event

事件是 SELECT、UPDATE、DELETE 或 INSERT 之一。

object

Object 是 *table* 或 *table*.*column*。（目前，实际上只实现了 *table* 形式。）

condition

任意的 SQL 布尔条件表达式。条件表达式不能引用 new 和 old 之外的任何表。

query

组成 *action* 的一个或多个查询可以是任何 SQL 的 SELECT、INSERT、UPDATE、DELETE 或 NOTIFY 语句。

在 *condition* 和 *action* 中，可以使用特殊表名 new 和 old 来引用被引用表（即 *object*）中的值。new 在 ON INSERT 和 ON UPDATE 规则中有效，用于引用正在被插入或更新的新行。old 在 ON UPDATE 和 ON DELETE 规则中有效，用于引用正在被更新或删除的现有行。

输出

CREATE

规则成功创建时返回的消息。

描述

PostgreSQL 的规则系统允许定义在对数据库表进行插入、更新或删除时所执行的替代动作。规则也用于实现表视图。

一条规则的语义是：当单个实例（行）被访问、插入、更新或删除时，
会有一个旧实例（对于选择、更新和删除）和一个新实例（对于插入和更新）。
给定事件类型和给定目标对象（表）的所有规则会被检查，顺序不确定。如果 WHERE 子句中指定的 *condition*（如果有）为真，该规则的 *action* 部分就会被执行。如果指定了 INSTEAD，*action* 会替代原始查询执行；否则，对于 ON INSERT，它在原始查询之后执行，而对于 ON UPDATE 或 ON DELETE，它在原始查询之前执行。在 *condition* 和 *action* 两部分中，旧实例和/或新实例中字段的值会替换到 `old.`*attribute-name* 和 `new.`*attribute-name* 上。

规则的 *action* 部分可以由一个或多个查询组成。要编写多个查询，需用圆括号或方括号包围它们。这些查询会按指定的顺序执行（而对于一个对象上多条规则的执行顺序则没有任何保证）。*action* 也可以是 NOTHING，表示没有任何动作。因此，一条 DO INSTEAD NOTHING 规则会阻止原始查询的执行（当其条件为真时）；而一条 DO NOTHING 规则没有任何用处。

规则的 *action* 部分与触发激活它的用户命令使用相同的命令标识符和事务标识符执行。

规则与视图

目前，ON SELECT 规则必须是无条件的 INSTEAD 规则，并且其动作必须由单个 SELECT 查询组成。因此，一条 ON SELECT 规则实际上会把对象表变为视图，其可见内容是该规则的 SELECT 查询返回的行，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。

CREATE VIEW 会创建一个虚拟表（没有底层存储）并为其关联一条 ON SELECT 规则。系统不会允许对该视图的更新，因为它知道那里没有真实的表。你可以通过定义 ON INSERT、ON UPDATE 和 ON DELETE 规则（或其中足以满足你需求的任意子集），将视图上的更新动作替换为对其他表的适当更新，从而营造出可更新视图的效果。

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，那么系统仍会拒绝执行该更新动作的尝试，因为它认为在某些情况下最终可能还是会尝试在该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；只需加上一条无条件的 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为非 INSTEAD；在它们触发的情况下，它们会附加在默认的 INSTEAD NOTHING 动作之上。

注解

要在一个表上定义规则，你必须拥有该表的规则定义权限。使用 `GRANT` 和 `REVOKE` 来更改权限。

避免循环规则非常重要。例如，尽管下面两条规则定义中的每一条都被 PostgreSQL 接受，但 select 命令会导致 PostgreSQL 报告错误，因为查询循环了太多次：

```
CREATE RULE "_RETemp" AS
    ON SELECT TO emp
    DO INSTEAD
 SELECT * FROM toyemp;

CREATE RULE "_RETtoyemp" AS
    ON SELECT TO toyemp
```

```
    DO INSTEAD
 SELECT * FROM emp;
```

这个从 EMP 中选择的尝试会导致 PostgreSQL 报错，因为这些查询循环了太多次：

```
SELECT * FROM emp;
```

当前，如果一个规则包含一个 NOTIFY 查询，该 NOTIFY 会被无条件执行 —— 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，在

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

中，在 UPDATE 期间会发送一个 NOTIFY 事件，无论是否存在 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

SQL92

`CREATE RULE` 语句是 PostgreSQL 的语言扩展。SQL92 中没有 `CREATE RULE` 语句。

CREATE SEQUENCE

CREATE SEQUENCE — 定义一个新的序列发生器

大纲

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE seqname [ INCREMENT increment ]
    [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
    [ START start ] [ CACHE cache ] [ CYCLE ]
```

输入

TEMPORARY or TEMP

如果指定了这个选项，序列对象只为本次会话创建，并在会话退出时自动删除。
在临时序列存在期间，（本会话中）同名的现有永久序列不可见。

seqname

要创建的序列的名称。

increment

INCREMENT *increment* 子句是可选的。正值将生成递增序列，负值生成递减序列。默认值为一（1）。

minvalue

可选子句 MINVALUE *minvalue* 决定序列可以生成的最小值。递增和递减序列的默认值分别是 1 和 -2^63-1。

maxvalue

可选子句 MAXVALUE *maxvalue* 决定序列的最大值。递增和递减序列的默认值分别是 2^63-1 和 -1。

start

可选的 START *start* 子句让序列可以从任何地方开始。默认起始值：递增序列为 *minvalue*，递减序列为 *maxvalue*。

cache

CACHE *cache* 选项使序列号可以预分配并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），这也是默认值。

CYCLE

可选的 CYCLE 关键字可以让序列在递增或递减序列分别达到 *maxvalue* 或 *minvalue* 时回绕。如果到达了限制，生成的下一个数将分别是 *minvalue* 或 *maxvalue*。不带 CYCLE 时，到达限制之后 nextval 调用将返回错误。

输出

`CREATE`

命令成功时返回的消息。

`ERROR: Relation 'seqname' already exists`

指定的序列已存在。

`ERROR: DefineSequence: MINVALUE (start) can't be >= MAXVALUE (max)`

指定的起始值超出范围。

`ERROR: DefineSequence: START value (start) can't be < MINVALUE (min)`

指定的起始值超出范围。

`ERROR: DefineSequence: MINVALUE (min) can't be >= MAXVALUE (max)`

最小值和最大值不一致。

描述

`CREATE SEQUENCE` 将向当前数据库中输入一个新的序列号发生器。这包括创建并初始化一个名为 *seqname* 的新的单行表。该发生器将为发出命令的用户所有。

序列创建之后，用函数 `nextval`、`currval` 和 `setval` 操作该序列。这些函数在用户指南中有文档。

虽然你不能直接更新序列，但你可以使用像这样的查询

`SELECT * FROM seqname;`

to examine the parameters and current state of a sequence. In particular, the `last_value` field of the sequence shows the last value allocated by any backend process. (Of course, this value may be obsolete by the time it's printed, if other processes are actively doing `nextval` calls.)

小心

如果对一个将被多个后端并发使用的序列对象使用大于一的 *cache* 设置，可能得到意外的结果。每个后端会在一次访问序列对象期间分配并缓存若干连续的序列值，并相应地增大序列对象的 `last_value`。之后，该后端内接下来 *cache*-1 次 `nextval` 使用只是返回预分配的值，而不再接触共享对象。因此，会话中已分配但未使用的任何数字都会在该会话结束时丢失。此外，虽然可以保证多个后端分配到互不相同的序列值，但综合考虑所有后端时，这些值可能不是按顺序生成的。（例如，*cache* 设为 10 时，后端 A 可能保留值 1..10 并返回 `nextval`=1，然后后端 B 可能保留值 11..20 并在后端 A 生成 nextval=2 之前返回 `nextval`=11。）因此，*cache* 设为一时，可以安全地假定 `nextval` 值是按顺序生成的；*cache* 设为大于一时，只应假定 `nextval` 的值互不相同，而不是纯粹按顺序生成的。另外，`last_value` 反映的是任何后端保留的最新值，无论它是否已被 `nextval` 返回。还有一点要考虑：在这种序列上执行的 `setval` 在其他后端用完它们缓存的预分配值之前不会被它们注意到。

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）。

当 *cache* 大于一时，每个后端用自己的缓存存储预分配的数字。当前会话中已缓存但未使用的数字将会丢失，从而在序列中留下“空洞”。

用法

创建一个名为 serial 的递增序列，从 101 开始：

```
CREATE SEQUENCE serial START 101;
```

从该序列中选择下一个数：

```
SELECT nextval('serial');

nextval
-------
    114
```

在 INSERT 中使用该序列：

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

在 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

SQL92

CREATE SEQUENCE 是一种 PostgreSQL 语言扩展。SQL92 中没有 CREATE SEQUENCE 语句。

CREATE TABLE

CREATE TABLE — 定义一个新表

大纲

```
CREATE [ [ LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name (
    { column_name data_type [ DEFAULT default_expr ] [ column_
constraint [, ... ] ]
    | table_constraint }  [, ... ]
)
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY |
  CHECK (expression) |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL ]
    [ 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 ] [ 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。

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元组类型（结构类型）。因此，表名不能与任何已有的数据类型同名。

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。）表不能与系统目录表同名。

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）。
约束是一条命名的规则：一个通过对在表上执行的插入、更新或删除操作的结果加以限制，来帮助定义有效值集合的 SQL 对象。

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

`[LOCAL] TEMPORARY` or `[LOCAL] 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。
在临时表存在期间，同名的现有持久表对当前会话不可见。
在临时表上创建的任何索引也都会自动成为临时索引。

`LOCAL` 一词是可选的。但参见兼容性。

`table_name`

要创建的表名。

`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。有关数据类型和数组的更多信息，请参阅
用户指南。

`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（不允许子选择，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。

`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` or `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。

`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 NULL 值。这等价于列约束 `CHECK (`*column* `NOT NULL)`。

`NULL`

该列允许包含 NULL 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` 约束指定一条规则：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` 子句指定新行或更新后的行要使插入或更新操作成功必须满足的完整性约束（测试）。每个约束都必须是一个产生布尔结果的表达式。
出现在列定义中的条件只应引用该列的值，而作为表约束出现的条件可以引用多个列。

当前，`CHECK` 表达式不能包含子选择，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)

`REFERENCES` 列约束指定新表的一个或多个列组成的列组只能包含与被引用表 *reftable* 的被引用列 *refcolumn* 的值相匹配的值。如果省略 *refcolumn*，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`，以及未指定时的默认匹配类型。

MATCH FULL 不允许多列外键中的某一列为 NULL，除非所有外键列都为 NULL。默认匹配类型允许某些外键列为 NULL 而外键的其他部分不为 NULL。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。

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 REFERENCES 列上添加索引，使与该 REFERENCES 列关联的 NO ACTION 和 CASCADE 操作能够更高效地执行。

DEFERRABLE or 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 or 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。

诊断

CREATE

表创建成功时返回的消息。

ERROR

表创建失败时返回的消息。通常还伴随一些描述性文字，例如：ERROR: Relation '*table*' already exists，如果指定的表在数据库中已存在，就会在运行时出现该错误。

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）。
- SQL92 标准规定 `CHECK` 列约束只能引用它所作用的列；只有 `CHECK` 表约束才能引用多个列。PostgreSQL 不强制这一限制；它对列检查约束和表检查约束一视同仁。
- 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        CHARACTER(5) CONSTRAINT firstkey PRIMARY KEY,
    title       CHARACTER VARYING(40) NOT NULL,
    did         DECIMAL(3) NOT NULL,
    date_prod   DATE,
    kind        CHAR(10),
    len         INTERVAL HOUR TO MINUTE
);

CREATE TABLE distributors (
     did    DECIMAL(3) 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         DECIMAL(3),
    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     DECIMAL(3) CHECK (did > 100),
    name    VARCHAR(40)
);
```

定义一个表检查约束：

```
CREATE TABLE distributors (
    did     DECIMAL(3),
    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         DECIMAL(3),
    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     DECIMAL(3),
    name    VARCHAR(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     DECIMAL(3) 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     DECIMAL(3) 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     DECIMAL(3),
    name    VARCHAR(40) UNIQUE
);
```

同样的唯一约束用表约束指定：

```
CREATE TABLE distributors (
    did     DECIMAL(3),
    name    VARCHAR(40),
    UNIQUE(name)
);
```

兼容性

CREATE TABLE 遵循 SQL92 Intermediate 以及 SQL99 的一个子集，例外情况见下文以及上文的描述。

临时表

除本地临时表外，SQL92 还定义了 CREATE GLOBAL TEMPORARY TABLE 语句。全局临时表对其他会话也是可见的。

对于临时表，有一个可选的 ON COMMIT 子句：

```
CREATE { GLOBAL | LOCAL } TEMPORARY TABLE table ( ... ) [ ON COMMIT {
 DELETE | PRESERVE } ROWS ]
```

ON COMMIT 子句指定每当执行 COMMIT 时是否清空临时表中的行。如果省略 ON COMMIT 子句，SQL92 规定默认为 ON COMMIT DELETE ROWS。但 PostgreSQL 的行为总是类似于 ON COMMIT PRESERVE ROWS。

NULL“约束”

NULL“约束”（实际上是一个非约束）是 PostgreSQL 对 SQL92 的扩展，加入它是为了与某些其他 RDBMS 兼容（并与 NOT NULL 约束对称）。由于它是任何列的默认情况，它的出现只是噪音。

断言

断言是一种特殊类型的完整性约束，与其他约束共享同一名字空间。不过，
断言不一定像约束那样依赖于某个特定的表，因此 SQL92 提供了 CREATE ASSERTION 语句作为定义约束的另一种方法：

```
CREATE ASSERTION name CHECK ( condition )
```

PostgreSQL 目前没有实现断言。

继承

通过 INHERITS 子句的多重继承是 PostgreSQL 的语言扩展。SQL99（但不是 SQL92）用不同的语法和语义定义了单一继承。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。

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 [ [ 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底层的`SELECT`语句。

参数

`[LOCAL] TEMPORARY` or `[LOCAL] TEMP`

如果指定，该表将创建为临时表。临时表在会话结束时自动删除。当临时表存在时，
同名的现有持久表对当前会话不可见。在临时表上创建的任何索引也自动是临时的。

`LOCAL`一词是可选的。

`table_name`

要创建的新表的名称。该表必须不已存在。不过，
可以创建一个与现有永久表同名的临时表。

`column_name`

新表中一列的名称。可以使用逗号分隔的列名列表指定多个列名。如果没有提供列名，
则从查询的输出列名中取得。

`query`

一条查询语句（即一条`SELECT` 命令）。允许语法的描述参阅SELECT。

诊断

可能的输出消息的摘要请参阅CREATE TABLE和SELECT。

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。SQL92 或 SQL99 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。

历史

CREATE TABLE AS命令自 PostgreSQL 6.3 起可用。

另见

CREATE TABLE，CREATE VIEW , 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 func ( arguments )
```

Inputs

name

给予新触发器的名称。

table

现有表的名称。

event

INSERT、DELETE 或 UPDATE 之一。

func

一个用户提供的函数。

Outputs

CREATE

触发器创建成功时返回此消息。

描述

CREATE TRIGGER 将把一个新触发器登记到当前数据库中。该触发器将与关系 *table* 关联，并执行指定的函数 *func*。

可以指定触发器在操作试图作用于元组之前（BEFORE，即约束检查和 INSERT、UPDATE 或 DELETE 尝试之前）引发，或在操作尝试之后（AFTER，例如约束检查完毕且 INSERT、UPDATE 或 DELETE 已完成之后）引发。如果触发器在事件之前引发，它可以跳过对当前元组的操作，或者修改被插入的元组（仅适用于 INSERT 和 UPDATE 操作）。如果触发器在事件之后引发，那么所有更改（包括最后一次插入、更新或删除）对触发器都是“可见”的。

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。

更多信息参见 PostgreSQL 程序员指南中关于 SPI 和触发器的章节。

Notes

要在表上创建触发器，用户必须拥有该表的TRIGGER权限。

截至当前版本，STATEMENT 触发器尚未实现。

关于如何移除触发器，参见 DROP TRIGGER 命令。

示例

在向表 films 追加或更新行之前，检查指定的发行商代码是否存在于 distributors 表中：

```
CREATE TRIGGER if_dist_exists
    BEFORE INSERT OR UPDATE ON films FOR EACH ROW
    EXECUTE PROCEDURE check_primary_key ('did', 'distributors', 'did')
;
```

在取消一个发行商或更新其代码之前，移除对表 films 的所有引用：

```
CREATE TRIGGER if_film_exists
    BEFORE DELETE OR UPDATE ON distributors FOR EACH ROW
    EXECUTE PROCEDURE check_foreign_key (1, 'CASCADE', 'did', 'films',
 'did');
```

第二个例子也可以使用外键约束来完成，如下所示：

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    CONSTRAINT if_film_exists
    FOREIGN KEY(did) REFERENCES films
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

兼容性

SQL92

SQL92 中没有 CREATE TRIGGER 语句。

SQL99

PostgreSQL 中的 CREATE TRIGGER 语句实现了 SQL99 标准的一个子集。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 只有行级触发器，没有语句级触发器。
- PostgreSQL 只允许被触发动作执行一个存储过程。SQL99 允许执行若干其他 SQL 命令（例如 CREATE TABLE）作为被触发动作。通过创建一个执行这些命令的存储过程，这一限制不难绕过。

另请参阅

CREATE FUNCTION，DROP TRIGGER , PostgreSQL 程序员指南

CREATE TYPE

CREATE TYPE — 定义一种新的数据类型

大纲

```
CREATE TYPE typename ( INPUT = input_function, OUTPUT = output_
function
      , INTERNALLENGTH = { internallength | VARIABLE }
    [ , EXTERNALLENGTH = { externallength | VARIABLE } ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ] [ , DELIMITER = delimiter ]
    [ , SEND = send_function ] [ , RECEIVE = receive_function ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
)
```

输入

typename

要创建的类型的名称。

internallength

一个字面值，指定新类型的内部长度。

externallength

一个字面值，指定新类型的外部（显示）长度。

input_function

一个由 CREATE FUNCTION 创建的函数的名称，它把数据从外部形式转换为该类型的内部形式。

output_function

一个由 CREATE FUNCTION 创建的函数的名称，它把数据从内部形式转换为适合显示的形式。

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。

default

该数据类型的默认值。通常省略此项，使默认值为 NULL。

send_function

一个由 CREATE FUNCTION 创建的函数的名称，它把该类型的数据转换为适合传输到另一台机器的形式。

receive_function

一个由 CREATE FUNCTION 创建的函数的名称，它把该类型的数据从适合从另一台机器传输的形式转换为内部形式。

alignment

该数据类型的存储对齐要求。如果指定，必须是 char、int2、int4 或 double；默认为 int4。

storage

该数据类型的存储技术。如果指定，必须是 plain、external、extended 或 main；默认为 plain。

输出

CREATE

类型创建成功时返回的消息。

描述

CREATE TYPE 允许用户向 PostgreSQL 注册一种新的用户数据类型，供当前数据库使用。定义类型的用户将成为其所有者。*typename* 是新类型的名称，在该数据库已定义的类型中必须唯一。

CREATE TYPE 要求在定义类型之前（用 CREATE FUNCTION）先注册两个函数。新基本类型的表示由 *input_function* 决定，它把该类型的外部表示转换为为该类型定义的操作符和函数可用的内部表示。自然，*output_function* 执行反向的转换。输入函数可以声明为接受一个 opaque 类型的参数，或者接受 opaque、OID、int4 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）。）输出函数可以声明为接受一个 opaque 类型的参数，或者接受 opaque、OID 两个类型的参数。（第一个参数实际上就是该数据类型本身，但由于输出函数必须先声明，把它声明为接受 opaque 类型更容易。第二个参数同样是数组类型时的数组元素类型。）

新的基本数据类型可以是定长的，这时 *internallength* 是一个正整数；也可以是变长的，通过把 *internallength* 设为 VARIABLE 来表示。（在内部，这通过把 typlen 设为 -1 来表示。）所有变长类型的内部表示都必须以一个给出该类型该值总长度的整数开头。

外部表示的长度用 *externallength* 关键字以类似的方式指定。（这个值目前未被使用，通常省略，让它默认为 VARIABLE。）

要指明一个类型是数组，可以用 ELEMENT 关键字指定数组元素的类型。例如，要定义一个 4 字节整数（"int4"）的数组，指定

```
ELEMENT = int4
```

关于数组类型的更多细节见下文。

要指明该类型数组的外部表示中值与值之间使用的分隔符，可以把 *delimiter* 设为某个特定的字符。默认分隔符是逗号（','）。注意，分隔符与数组元素类型相关联，
而不是与数组类型本身。

如果用户希望该数据类型的列默认为 NULL 以外的值，可以指定一个默认值。用 DEFAULT 关键字指定默认值。（这样的默认值可以被附加在特定列上的显式 DEFAULT 子句覆盖。）

可选参数 *send_function* 和 *receive_function* 目前未被使用，通常省略（让它们分别默认为 *output_function* 和 *input_function*）。这两个函数将来可能会恢复用于指定与机器无关的二进制表示。

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。注意，内部表示比 Datum 类型宽度（多数机器上 4 字节，少数机器上 8 字节）更长的类型不能按值传递。

alignment 关键字指定该数据类型所需的存储对齐。允许的取值分别相当于按 1、2、4 或 8 字节边界对齐。注意，变长类型的对齐至少必须是 4，因为它们必然以一个 int4 作为第一个成分。

storage 关键字允许为变长数据类型选择存储策略（定长类型只允许 plain）。plain 对该数据类型禁用 TOAST：它总是内联存储且不压缩。extended 提供完整的 TOAST 能力：系统会先尝试压缩较长的数据值，如果仍然太长，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。（采用这种存储方法的数据项在实在没有其他办法让行放得下时仍可能被移出主表，但相比 extended 和 external 的数据项，它们会被优先保留在主表中。）

数组类型

每当创建用户定义的数据类型时，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，并使用内置的输入和输出函数 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 N 个相同元素组成的数组，并且除了为整个类型计划提供的操作之外，你还希望允许通过下标直接访问这 N 个元素。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浮点数。注意，这一机制只对内部形式恰好是 N 个相同字段序列的定长类型有效。可下标访问的变长类型必须使用 array_in和array_out所使用的广义内部表示。由于历史原因（也就是说，这显然是错的，但要改已经太迟了），定长数组类型的下标从零开始，而不像变长数组那样从一开始。

注意

用户定义的类型名不能以下划线字符（“_”）开头，且长度只能是 30 个字符（或者一般来说是 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。

示例

这个示例创建 box 数据类型，然后在表定义中使用该类型：

```
CREATE TYPE box (INTERNALLENGTH = 16,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2);
CREATE TABLE myboxes (id INT4, description box);
```

如果 box 的内部结构是四个 float4 组成的数组，我们也可以改成这样写：

```
CREATE TYPE box (INTERNALLENGTH = 16,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2,
```

```
    ELEMENT = float4);
```

这样就允许通过下标访问 box 值的各分量浮点数。除此之外该类型的行为与之前完全相同。

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 int4, obj bigobj);
```

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 , PostgreSQL 程序员指南

CREATE USER

CREATE USER — 定义一个新的数据库用户账号

大纲

```
CREATE USER username [ [ WITH ] option [ ... ] ]

where option can be:

   SYSID uid
        | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
        | CREATEDB | NOCREATEDB
        | CREATEUSER | NOCREATEUSER
        | IN GROUP groupname [, ...]
        | VALID UNTIL 'abstime'
```

输入

username

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。这些 ID 完全不必与 UNIX 用户 ID 相匹配，但有些人选择让这些数字保持一致。

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，
但该用户将无法连接到做了口令认证的服务器。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 pg_shadow 中是否以加密形式存储。（如果两者都未指定，默认行为由 PASSWORD_ENCRYPTION 服务器参数决定。）如果给出的字符串已经是 MD5 加密格式，那么它会被原样存储，而不管指定的是 ENCRYPTED 还是 UNENCRYPTED。这允许在转储/恢复期间重新装载加密的口令。

关于如何设置认证机制的细节，见管理员指南中关于客户端认证的一章。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。如果这个子句被省略，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。

输出

`CREATE USER`

命令成功完成时返回的消息。

描述

`CREATE USER` 将向一个 PostgreSQL 实例添加一个新用户。关于管理用户和认证的信息，请参阅管理员指南。你必须是数据库超级用户才能使用这个命令。

使用 ALTER USER 更改用户的口令和权限，使用 DROP USER 删除一个用户。使用 ALTER GROUP 把用户添加到其他组中或从其他组中移除该用户。PostgreSQL 附带一个脚本createuser，它具有与这个命令相同的功能（事实上，它调用这个命令），但可以从命令 shell 运行。

用法

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，其账号有效到 2001 年底。注意 2002 年到来一秒之后，该账号不再有效：

```
CREATE USER miriam WITH PASSWORD 'jw8s0F4' VALID UNTIL 'Jan 1 2002';
```

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

SQL92

SQL92 中没有 `CREATE USER` 语句。

CREATE VIEW

CREATE VIEW — 定义一个新视图

大纲

```
CREATE VIEW view [ ( column name list ) ] AS SELECT query
```

输入

view

要创建的视图的名称。

column name list

一个可选的名称列表，用作视图的列。如果给出，这些名称会覆盖本应从 SQL 查询推导出的列名。

query

一条 SQL 查询，它将提供视图的列和行。

关于有效参数的更多信息，参见 SELECT 语句。

输出

CREATE

视图创建成功时返回的消息。

ERROR: Relation '*view*' already exists

如果指定的视图已存在于数据库中，就会出现这个错误。

NOTICE: Attribute '*column*' has an unknown type

如果你不指定类型，创建出的视图就会带有一个未知类型的列。例如，
下面的命令会产生一条警告：

```
CREATE VIEW vista AS SELECT 'Hello World'
```

而这条命令则不会：

```
CREATE VIEW vista AS SELECT text 'Hello World'
```

描述

CREATE VIEW 将定义一个表的一个视图。该视图不会被物理物化；相反，会自动生成一条查询重写检索规则来支持对视图的检索操作。

注意

目前视图是只读的：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改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。

用法

创建一个由所有喜剧（Comedy）影片组成的视图：

```
CREATE VIEW kinds AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';

SELECT * FROM kinds;

 code  |           title            | did | date_prod  |  kind  | len
-------+----------------------------+-----+------------+--------+-----
--
 UA502 | Bananas                    | 105 | 1971-07-13 | Comedy | 01:22
 C_701 | There's a Girl in my Soup | 107 | 1970-06-11 | Comedy | 01:36
(2 rows)
```

兼容性

SQL92

SQL92 为 CREATE VIEW 语句规定了一些额外的能力：

```
CREATE VIEW view [ column [, ...] ]
    AS SELECT expression [ AS colname ] [, ...]
    FROM table [ WHERE condition ]
    [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

完整 SQL92 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 都会被检查，以确保数据满足定义视图的条件。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。

DECLARE

DECLARE — 定义一个游标

大纲

```
DECLARE cursorname [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] ]
```

输入

cursorname

用于后续 FETCH 操作的游标的名称。

BINARY

使游标以二进制而不是文本格式获取数据。

INSENSITIVE

SQL92 关键字，表示从游标检索的数据应不受其他进程或游标的更新的影响。由于 PostgreSQL 中游标操作发生在事务内，情况总是如此。这个关键字没有任何效果。

SCROLL

SQL92 关键字，表示每次 FETCH 操作可以检索多行。由于 PostgreSQL 总是允许这样做，这个关键字没有任何效果。

query

一个提供将由该游标管理的行的 SQL 查询。关于有效参数的更多信息请参阅 SELECT 语句。

READ ONLY

SQL92 关键字，表示游标将以只读模式使用。由于这是 PostgreSQL 中唯一可用的游标访问模式，这个关键字没有任何效果。

UPDATE

SQL92 关键字，表示游标将用于更新表。由于 PostgreSQL 目前不支持游标更新，这个关键字会引发一条信息性错误消息。

column

要更新的列。由于 PostgreSQL 目前不支持游标更新，UPDATE 子句会引发一条信息性错误消息。

输出

SELECT

SELECT 成功运行时返回的消息。

```
NOTICE: Closing pre-existing portal "cursorname"
```

如果在当前事务块中已经声明过同名游标，就会报告这条消息。之前的定义会被丢弃。

```
ERROR: DECLARE CURSOR may only be used in begin/end transaction blocks
```

如果游标不是在事务块内声明的，就会发生这个错误。

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（ASCII 或其他编码方案，取决于 PostgreSQL 后端的构建方式）返回数据。由于数据本来就是以二进制格式存储的，系统必须做一次转换来产生文本格式。此外，文本格式通常比对应的二进制格式更大。信息以文本形式返回后，客户端应用可能需要把它转换为二进制格式才能处理。BINARY 游标以本机二进制表示形式返回数据。

例如，如果一个查询从整数列返回值一，用默认游标会得到字符串 1，而用二进制游标会得到一个等于 control-A（^A）的 4 字节值。

使用 BINARY 游标应当小心。诸如 psql 这样的用户应用不识别二进制游标，并期望数据以文本格式返回。

字符串表示与体系结构无关，而二进制表示在不同的机器体系结构之间可能不同。PostgreSQL 不为二进制游标解决字节序或表示问题。因此，如果你的客户端机器和服务器机器使用不同的表示（例如 “big-endian” 对 “little-endian” ），你大概不想让数据以二进制格式返回。不过，二进制游标可能稍微高效一些，因为服务器到客户端数据传输中的转换开销更小。

> **提示**
>
> 如果你想以 ASCII 显示数据，以 ASCII 取回它会省去客户端的一些工作。

注解

游标只在事务中可用。使用 BEGIN、COMMIT 和 ROLLBACK 来定义事务块。

在 SQL92 中，游标只在嵌入式 SQL（ESQL）应用中可用。PostgreSQL 后端没有实现显式的 OPEN cursor 语句；游标在声明时即视为打开。不过，ecpg（PostgreSQL 的嵌入式 SQL 预处理器）支持 SQL92 的游标约定，包括涉及 DECLARE 和 OPEN 语句的那些。

用法

声明一个游标：

```
DECLARE liahona CURSOR
    FOR SELECT * FROM films;
```

兼容性

SQL92

SQL92 只允许在嵌入式 SQL 和模块中使用游标。PostgreSQL 允许交互式地使用游标。SQL92 允许嵌入式或模块游标更新数据库信息。所有 PostgreSQL 游标都是只读的。BINARY 关键字是一种 PostgreSQL 扩展。

DELETE

DELETE — 删除一个表中的行

大纲

```
DELETE FROM [ ONLY ] table [ WHERE condition ]
```

输入

table

一个已存在表的名称。

condition

这是一个 SQL 选择查询，它返回要删除的行。

关于 WHERE 子句的进一步描述请参考 SELECT 语句。

输出

`DELETE` *count*

成功删除了项时返回的消息。*count* 是删除的行数。

如果 *count* 为 0，表示没有行被删除。

描述

`DELETE`从指定表中删除满足 WHERE 子句的行。

如果省略condition（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元组。如果只想更新提到的特定表，应该使用 ONLY 子句。

要修改一个表，你必须拥有对它的写权限，同时对*condition*中读取其值的任何表拥有读权限。

用法

删除除音乐剧外的所有电影：

```
DELETE FROM films WHERE kind <> 'Musical';
SELECT * FROM films;
```

```
 code  |           title           | did | date_prod  |  kind   | len
-------+---------------------------+-----+------------+---------+----
---
 UA501 | West Side Story           | 105 | 1961-01-03 | Musical | 02:
32
 TC901 | The King and I            | 109 | 1956-08-11 | Musical | 02:
13
 WD101 | Bed Knobs and Broomsticks | 111 |            | Musical | 01:
57
(3 rows)
```

清空表 `films`：

```
DELETE FROM films;
SELECT * FROM films;

 code | title | did | date_prod | kind | len
------+-------+-----+-----------+------+-----
(0 rows)
```

兼容性

SQL92

SQL92允许定位式 DELETE 语句：

```
DELETE FROM table WHERE
    CURRENT OF cursor
```

其中 *cursor* 标识一个已打开的游标。PostgreSQL中的交互式游标是只读的。

DROP AGGREGATE

DROP AGGREGATE — 删除一个聚集函数

大纲

```
DROP AGGREGATE name ( type )
```

输入

name

一个已存在聚集函数的名称。

type

一个已存在聚集函数的输入数据类型，如果函数接受任意输入类型则为 `*`。（关于数据类型的更多信息请参考 PostgreSQL User's Guide。）

输出

`DROP`

命令成功时返回的消息。

`ERROR: RemoveAggregate: aggregate '`*name*`' for type `*type*` does not exist`

如果指定的聚集函数在数据库中不存在，就会出现这条消息。

描述

`DROP AGGREGATE`将删除一个已存在的聚集定义。要执行这条命令，当前用户必须是该聚集函数的拥有者。

注意

使用 CREATE AGGREGATE 创建聚集函数。

用法

删除类型 `int4` 的 `myavg` 聚集函数：

```
DROP AGGREGATE myavg(int4);
```

兼容性

SQL92

SQL92中没有 `DROP AGGREGATE`语句；这条语句是 PostgreSQL 语言扩展。

DROP DATABASE

DROP DATABASE — 删除一个数据库

大纲

```
DROP DATABASE name
```

输入

name

要删除的现有数据库的名称。

输出

`DROP DATABASE`

命令成功时返回此消息。

`DROP DATABASE: cannot be executed on the currently open database`

你不能连接在你将要删除的数据库上。请改为连接到`template1`或任何其他数据库，然后再次运行这条命令。

`DROP DATABASE: may not be called in a transaction block`

你必须先完成进行中的事务，然后才能调用这条命令。

描述

`DROP DATABASE`删除现有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（通常是创建它的用户）执行。

`DROP DATABASE`无法撤销。请小心使用！

注解

当连接到目标数据库时无法执行这条命令。因此，改用 shell 脚本dropdb可能更方便，它是这条命令的一个包装。

关于如何创建数据库的信息，请参阅 CREATE DATABASE。

兼容性

SQL92

`DROP DATABASE`语句是 PostgreSQL的语言扩展；SQL92中没有这样的命令。

DROP FUNCTION

DROP FUNCTION — 移除一个用户定义的函数

大纲

```
DROP FUNCTION name ( [ type [, ...] ] )
```

Inputs

name

现有函数的名称。

type

该函数参数的类型。

Outputs

`DROP`

命令成功完成时返回的消息。

`NOTICE RemoveFunction: Function "`*name*`" ("`*types*`") does not exist`

如果指定的函数在当前数据库中不存在，则给出此消息。

描述

DROP FUNCTION 将删除现有函数的定义。要执行此命令，用户必须是该函数的所有者。必须指定函数的输入参数类型，因为可能存在多个同名但参数列表不同的函数。

注意

关于创建函数的信息，参见CREATE FUNCTION。

系统不会检查依赖该函数的类型、操作符、访问方法或触发器是否已被先行删除。

示例

此命令删除平方根函数：

```
DROP FUNCTION sqrt(integer);
```

兼容性

SQL99 定义了 `DROP FUNCTION` 语句。它的一种语法形式是：

```
DROP FUNCTION name (arg, ...) { RESTRICT | CASCADE }
```

其中 CASCADE 指定删除所有依赖该函数的对象，而 RESTRICT 在存在依赖对象时拒绝删除该函数。

另请参阅

CREATE FUNCTION

DROP GROUP

DROP GROUP — 移除一个用户组

大纲

```
DROP GROUP name
```

输入

name

现有组的名称。

输出

`DROP GROUP`

组成功删除时返回的消息。

描述

`DROP GROUP` 从数据库中移除指定的组。组中的用户不会被删除。

使用 CREATE GROUP 添加新组，使用 ALTER GROUP 更改组的成员。

用法

删除一个组：

```
DROP GROUP staff;
```

兼容性

SQL92

SQL92 中没有 `DROP GROUP`。

DROP INDEX

DROP INDEX — 移除一个索引

大纲

```
DROP INDEX index_name [, ...]
```

输入

index_name

要移除的索引的名称。

输出

`DROP`

命令成功完成时返回的消息。

`ERROR: index "index_name" does not exist`

如果 *index_name* 不是数据库中的一个索引，就会出现这条消息。

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。

注意

`DROP INDEX` 是一种 PostgreSQL 语言扩展。

关于如何创建索引的信息，请参阅 CREATE INDEX。

用法

这条命令将移除 `title_idx` 索引：

```
DROP INDEX title_idx;
```

兼容性

SQL92

SQL92 定义了访问通用关系数据库的命令。索引是一种实现相关的特性，因此 SQL92 语言中没有索引特定的命令或定义。

DROP LANGUAGE

DROP LANGUAGE — 移除一个用户定义的过程语言

大纲

```
DROP [ PROCEDURAL ] LANGUAGE name
```

输入

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。

输出

`DROP`

语言成功删除时返回此消息。

`ERROR: Language "name" doesn't exist`

如果数据库中没有名为 *name* 的语言，就会出现这条消息。

描述

`DROP PROCEDURAL LANGUAGE` 将删除此前注册的名为 *name* 的过程语言的定义。

注意

`DROP PROCEDURAL LANGUAGE` 语句是一种 PostgreSQL 语言扩展。

关于如何创建过程语言的信息，参见CREATE LANGUAGE。

系统不会检查是否有使用该语言注册的函数或触发器过程仍然存在。
为了不删除并重建全部函数就能重新启用它们，必须把这些函数在 pg_proc 中的 prolang 属性调整为为该 PL 重建的 pg_language 条目的新对象 ID。

用法

这条命令删除 PL/Sample 语言：

```
DROP LANGUAGE plsample;
```

兼容性

SQL92

SQL92 中没有 `DROP PROCEDURAL LANGUAGE`。

DROP OPERATOR

DROP OPERATOR — 移除一个用户定义的操作符

大纲

```
DROP OPERATOR id ( lefttype | NONE , righttype | NONE )
```

输入

id

现有操作符的标识符。

lefttype

操作符左参数的类型；如果操作符没有左参数，写 NONE。

righttype

操作符右参数的类型；如果操作符没有右参数，写 NONE。

输出

DROP

命令成功时返回的消息。

ERROR: RemoveOperator: binary operator '*oper*' taking '*lefttype*' and '*righttype*' does not exist

如果指定的二元操作符不存在，就会出现这条消息。

ERROR: RemoveOperator: left unary operator '*oper*' taking '*lefttype*' does not exist

如果指定的左一元操作符不存在，就会出现这条消息。

ERROR: RemoveOperator: right unary operator '*oper*' taking '*righttype*' does not exist

如果指定的右一元操作符不存在，就会出现这条消息。

描述

DROP OPERATOR 从数据库中删除一个现有的操作符。要执行此命令，你必须是该操作符的所有者。

左一元或右一元操作符的左类型或右类型必须相应地指定为 NONE。

注意

DROP OPERATOR 语句是一种 PostgreSQL 语言扩展。

关于如何创建操作符的信息，参见 CREATE OPERATOR。

删除任何依赖被删操作符的访问方法和操作符类是用户的责任。

用法

移除 `int4` 的幂操作符 `a^n`：

```
DROP OPERATOR ^ (int4, int4);
```

移除 `boolean` 的左一元取反操作符（`! b`）：

```
DROP OPERATOR ! (none, bool);
```

移除 `int4` 的右一元阶乘操作符（`i !`）：

```
DROP OPERATOR ! (int4, none);
```

兼容性

SQL92

SQL92 中没有 `DROP OPERATOR`。

DROP RULE

DROP RULE — 删除一个重写规则

大纲

```
DROP RULE name [, ...]
```

输入

name

要删除的已存在规则的名称。

输出

`DROP`

成功时返回的消息。

`ERROR: Rule or view "name" not found`

如果指定的规则不存在，就会出现这条消息。

描述

`DROP RULE`从指定的 PostgreSQL 规则系统中删除一个规则。PostgreSQL 会立即停止强制执行该规则，并从系统目录中清除其定义。

注意

`DROP RULE`语句是一种 PostgreSQL 语言扩展。

关于如何创建规则的信息，请参考 `CREATE RULE`。

规则一旦被删除，对该规则所写入的历史信息的访问可能会随之消失。

用法

删除重写规则 `newrule`：

```
DROP RULE newrule;
```

兼容性

SQL92

SQL92 中没有 `DROP RULE`。

DROP SEQUENCE

DROP SEQUENCE — 移除一个序列

大纲

```
DROP SEQUENCE name [, ...]
```

输入

name

序列的名称。

输出

`DROP`

序列被成功删除时返回的消息。

`ERROR: sequence "name" does not exist`

指定的序列不存在时出现这条消息。

描述

`DROP SEQUENCE` 从数据库中移除序列号生成器。在当前把序列实现为特殊表的方式下，它的工作方式与 `DROP TABLE` 语句一样。

注意

`DROP SEQUENCE` 语句是一种 PostgreSQL 语言扩展。

关于如何创建序列的信息，请参阅 `CREATE SEQUENCE` 语句。

用法

要从数据库中移除序列 `serial`：

```
DROP SEQUENCE serial;
```

兼容性

SQL92

SQL92 中没有 `DROP SEQUENCE`。

DROP TABLE

DROP TABLE — 删除一个表

大纲

```
DROP TABLE name [, ...]
```

输入

name

要删除的已存在表的名称。

输出

`DROP`

命令成功完成时返回的消息。

`ERROR: table "name" does not exist`

如果指定的表在数据库中不存在。

描述

`DROP TABLE`从数据库中删除表。只有表的拥有者才能删除它。一个表可以通过使用 `DELETE`清空行，而不被删除。

如果被删除的表上有二级索引，会先删除它们。只删除一个二级索引不会影响底层表的内容。

注意

关于如何创建或修改表的信息，请参考 `CREATE TABLE`和 `ALTER TABLE`。

用法

删除两个表 `films` 和 `distributors`：

```
DROP TABLE films, distributors;
```

兼容性

SQL92

SQL92 为 DROP TABLE 规定了一些额外的能力：

```
DROP TABLE table { RESTRICT | CASCADE }
```

RESTRICT

保证只有没有任何依赖视图或完整性约束的表才能被删除。

CASCADE

所有引用它的视图或完整性约束也将被删除。

> **提示**
>
> 目前，要删除一个被引用的视图，必须显式删除它。

DROP TRIGGER

DROP TRIGGER — 移除一个触发器

大纲

```
DROP TRIGGER name ON table
```

输入

name

一个现有触发器的名称。

table

一个表的名称。

输出

`DROP`

触发器被成功删除时返回的消息。

`ERROR: DropTrigger: there is no trigger name on relation "table"`

指定的触发器不存在时出现这条消息。

描述

`DROP TRIGGER` 将移除对一个现有的触发器定义的所有引用。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。

示例

销毁表 `films` 上的 `if_dist_exists` 触发器：

```
DROP TRIGGER if_dist_exists ON films;
```

兼容性

SQL92

SQL92 中没有 `DROP TRIGGER` 语句。

SQL99

PostgreSQL 中的 `DROP TRIGGER` 语句与 SQL99 不兼容。在 SQL99 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 — 移除一个用户定义的数据类型

大纲

```
DROP TYPE typename [, ...]
```

输入

typename

现有类型的名称。

输出

`DROP`

命令成功时返回的消息。

`ERROR: RemoveType: type 'typename' does not exist`

如果找不到指定的类型，就会出现这条消息。

描述

`DROP TYPE` 将从系统目录中删除一个用户类型。

只有类型的所有者才能删除它。

注意

- 删除任何使用被删类型的操作符、函数、聚合、访问方法、子类型和表是用户的责任。不过，相关的数组数据类型（由 `CREATE TYPE` 自动创建）将被自动删除。
- 如果删除了内置类型，服务器的行为将不可预料。

示例

要删除 `box` 类型：

```
DROP TYPE box;
```

兼容性

SQL99 中也有 `DROP TYPE` 语句。与大多数其他 “drop” 命令一样，SQL99 中的 `DROP TYPE` 要求一个 “drop behavior” 子句，用于选择是删除所有依赖对象还是在存在依赖对象时拒绝删除：

```
DROP TYPE name { CASCADE | RESTRICT }
```

PostgreSQL 目前完全忽略依赖关系。

注意，PostgreSQL 的 `CREATE TYPE` 命令和数据类型扩展机制与 SQL99 不同。

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
```

输入

name

一个已存在用户的名称。

输出

`DROP USER`

成功删除用户时返回的消息。

`ERROR: DROP USER: user "name" does not exist`

用户名未找到时出现此消息。

`DROP USER: user "name" owns database "name", cannot be removed`

你必须先删除该数据库或更改其所有权。

描述

`DROP USER` 从数据库中删除指定的用户。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，则会得到一个错误。

使用 CREATE USER 来添加新用户，使用 ALTER USER 来更改用户的属性。PostgreSQL 附带一个脚本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。

用法

删除一个用户账户：

```
DROP USER jonathan;
```

兼容性

SQL92

SQL92 中没有 `DROP USER`。

DROP VIEW

DROP VIEW — 删除一个视图

大纲

```
DROP VIEW name [, ...]
```

输入

name

现有视图的名称。

输出

`DROP`

命令成功时返回的消息。

`ERROR: view name does not exist`

如果指定的视图在数据库中不存在，就会出现此消息。

描述

`DROP VIEW`从数据库中删除一个现有视图。要执行这条命令，你必须是该视图的所有者。

注解

关于如何创建视图的信息，请参阅 `CREATE VIEW`命令。

用法

这条命令将删除名为`kinds`的视图：

```
DROP VIEW kinds;
```

兼容性

SQL92

SQL92为`DROP VIEW`规定了一些额外的能力：

```
DROP VIEW view { RESTRICT | CASCADE }
```

输入

RESTRICT

确保只有没有依赖视图或完整性约束的视图才能被销毁。

CASCADE

所有引用它的视图和完整性约束也将被删除。

注解

目前，要从PostgreSQL数据库中删除一个被引用的视图，你必须显式地删除它。

END

END — 提交当前事务

大纲

```
END [ WORK | TRANSACTION ]
```

输入

WORK
TRANSACTION

可选的关键字。它们没有效果。

输出

`COMMIT`

事务被成功提交时返回的消息。

```
NOTICE: COMMIT: no transaction in progress
```

如果没有正在进行的事务。

描述

`END` 是一种 PostgreSQL 扩展，是 SQL92 兼容的 COMMIT 的同义词。

注意

关键字 WORK 和 TRANSACTION 是噪声词，可以省略。

使用 ROLLBACK 中止一个事务。

用法

要使所有更改永久化：

```
END WORK;
```

兼容性

SQL92

`END` 是一种提供与 COMMIT 等价功能的 PostgreSQL 扩展。

EXPLAIN

EXPLAIN — 显示一条语句的执行计划

大纲

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] query
```

输入

ANALYZE

执行查询并显示实际运行时间（runtimes）的标志。

VERBOSE

显示详细查询计划的标志。

query

任何*query*。

输出

```
NOTICE: QUERY PLAN: plan
```

来自PostgreSQL后端的显式查询计划。

```
EXPLAIN
```

查询计划显示完毕后发出的标志。

描述

这条命令显示PostgreSQL规划器为给定查询生成的执行计划。
执行计划显示查询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元组汇集到一起。

显示中最关键的部分是估计的查询执行代价，
即规划器对运行该查询所需时间的猜测（以磁盘页获取次数为单位度量）。
实际上会显示两个数字：返回第一个元组之前的启动时间，以及返回所有元组的总时间。
对大多数查询来说，总时间才是重要的，但在诸如 EXISTS 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个元组后无论如何都会停止）。
另外，如果你用 LIMIT 子句限制返回的元组数，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 选项使查询被真正执行，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元组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。

VERBOSE 选项输出计划树的完整内部表示，而不仅仅是一个摘要（并且也会发送到 postmaster 日志文件）。通常这个选项只对调试PostgreSQL有用。

小心

请记住，使用 ANALYZE 时查询会被真正执行。虽然EXPLAIN会丢弃 SELECT 本会返回的任何输出，查询的其他副作用会照常发生。如果你想在 INSERT、UPDATE 或 DELETE 查询上使用 EXPLAIN ANALYZE而又不让查询影响你的数据，可以使用这样的方法：

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

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。更多信息请参阅用户指南和程序员指南。

用法

要显示对一个只有单个int4列和 128 行的表的简单查询的查询计划：

```
EXPLAIN SELECT * FROM foo;
    NOTICE:  QUERY PLAN:

Seq Scan on foo  (cost=0.00..2.28 rows=128 width=4)

EXPLAIN
```

对于同一个表，如果有一个支持查询上等值连接条件的索引，EXPLAIN将显示一个不同的计划：

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)

EXPLAIN
```

最后，对于同一个表，如果有支持查询上等值连接条件的索引，EXPLAIN对使用聚合函数的查询将显示如下内容：

```
EXPLAIN SELECT sum(i) FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Aggregate  (cost=0.42..0.42 rows=1 width=4)
  ->  Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)
```

注意，这里显示的具体数字、甚至被选中的查询策略都可能随 PostgreSQL版本因规划器的改进而变化。

兼容性

SQL92

SQL92 中没有定义EXPLAIN语句。

FETCH

FETCH — 使用游标从表中检索行

大纲

```
FETCH [ direction ] [ count ] { IN | FROM } cursor
FETCH [ FORWARD | BACKWARD | RELATIVE ] [ # | ALL | NEXT | PRIOR ] {
 IN | FROM } cursor
```

输入

direction

selector 定义提取方向。它可以是下列之一：

FORWARD

提取下一个（或下一些）行。如果省略 *selector*，这是默认值。

BACKWARD

提取前一个（或前一些）行。

RELATIVE

为兼容 SQL92 而存在的噪声词。

count

count 决定提取多少行。它可以是下列之一：

#

一个有符号整数，指定要提取多少行。注意，负整数等价于反转 FORWARD 和 BACKWARD 的方向。

ALL

检索所有剩余的行。

NEXT

等价于指定数量 `1`。

PRIOR

等价于指定数量 `-1`。

cursor

一个已打开游标的名称。

输出

`FETCH`返回由指定游标定义的查询的结果。如果查询失败，将返回下列消息：

```
NOTICE: PerformPortalFetch: portal "cursor" not found
```

如果*cursor*此前未被声明。游标必须在事务块内声明。

```
NOTICE: FETCH/ABSOLUTE not supported, using RELATIVE
```

PostgreSQL不支持游标的绝对定位。

```
ERROR: FETCH/RELATIVE at current position is not supported
```

SQL92允许使用语法

```
FETCH RELATIVE 0 FROM cursor.
```

反复检索游标“当前位置”上的行。

PostgreSQL目前不支持这一概念；实际上，值零被保留用来表示要检索所有行，等价于指定 ALL 关键字。如果使用了 RELATIVE 关键字，PostgreSQL 假定用户想要的是 SQL92行为，并返回这条错误消息。

描述

FETCH允许用户使用游标检索行。检索的行数由 # 指定。如果游标中剩余的行数少于 #，则只提取那些可用的行。用关键字 ALL 代替数字将检索游标中所有剩余的行。可以沿 FORWARD 和 BACKWARD 两个方向提取行。默认方向是 FORWARD。

> **提示**
>
> 行数允许指定为负数。负数等价于反转 FORWARD 和 BACKWARD 关键字的方向。例如，FORWARD -1与BACKWARD 1相同。

注意

注意，FORWARD 和 BACKWARD 关键字是 PostgreSQL扩展。也支持 SQL92语法，即命令的第二种形式。关于兼容性问题的细节见下文。

PostgreSQL不支持通过游标更新数据，因为把游标更新映射回基表通常不可行，VIEW 更新也是同样的情况。因此，用户必须发出显式的 UPDATE 命令来替换数据。

游标只能在事务内部使用，因为它们保存的数据跨越用户的多次查询。

使用 MOVE 改变游标位置。DECLARE 定义游标。关于事务的更多信息，请参阅 BEGIN、COMMIT 和 ROLLBACK。

用法

下面的例子使用游标遍历一个表。

```
-- Set up and use a cursor:

BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

```
-- Fetch first 5 rows in the cursor liahona:
FETCH FORWARD 5 IN 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

-- Fetch previous row:
FETCH BACKWARD 1 IN liahona;

 code  | title   | did | date_prod | kind   | len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- close the cursor and commit work:

CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL92

> **注意**
>
> 游标的非嵌入式使用是PostgreSQL 扩展。这里把游标的语法和用法与SQL92 定义的嵌入式游标形式相比较。

SQL92允许 FETCH 绝对定位游标，并允许把结果放入显式变量：

```
FETCH ABSOLUTE #
    FROM cursor
    INTO :variable [, ...]
```

ABSOLUTE

游标应被定位到指定的绝对行号。PostgreSQL 中的所有行号都是相对编号，因此不支持这一能力。

:*variable*

目标主变量。

GRANT

GRANT — 定义访问权限

大纲

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES |
 TRIGGER } [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] objectname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]
```

描述

GRANT命令把对象（表、视图、序列）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。注意，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。

除对象的创建者之外的用户对对象没有任何访问权限，除非创建者授予权限。
无需把权限授予对象的创建者，因为创建者自动持有所有权限。（不过，
创建者也可以出于安全考虑选择撤销他自己的某些权限。注意，
授予和撤销权限的能力是创建者固有的，不会丢失。删除对象的权利同样是创建者固有的，
不能被授予或撤销。）

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 FROM。

INSERT

允许向指定表中INSERT新行。还允许COPY TO。

UPDATE

允许对指定表的任意列进行UPDATE。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、currval和setval。

DELETE

允许从指定表中DELETE行。

RULE

允许在表/视图上创建规则。（见CREATE RULE语句。）

REFERENCES

要创建一个带外键约束的表，必须在包含被引用键的表上拥有此权限。

TRIGGER

允许在指定表上创建触发器。（见CREATE TRIGGER语句。）

ALL PRIVILEGES

一次性授予上述所有权限。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

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。

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命令获取现有对象上权限的信息：

```
        Database    = lusitania
+------------------+---------------------------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------------+---------------------------------------------+
| mytable          | {"=rw","miriam=arwdRxt","group todos=rw"}   |
+------------------+---------------------------------------------+
Legend:
      uname=arwR -- privileges granted to a user
group gname=arwR -- privileges granted to a group
           =arwR -- privileges granted to PUBLIC

               r -- SELECT ("read")
               w -- UPDATE ("write")
               a -- INSERT ("append")
               d -- DELETE
               R -- RULE
               x -- REFERENCES
               t -- TRIGGER
         arwdRxt -- ALL PRIVILEGES
```

REVOKE命令用于撤销访问权限。

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

SQL92

ALL PRIVILEGES中的PRIVILEGES关键字是必需的。SQL不支持在一条命令中为多个表设置权限。

SQL92的 GRANT 语法允许为表中的单个列设置权限，也允许设置把相同权限授予他人的权限：

```
GRANT privilege [, ...]
    ON object [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] } [ WITH GRANT OPTION ]
```

SQL允许为其他类型的对象授予 USAGE 权限：CHARACTER SET、COLLATION、TRANSLATION、DOMAIN。

TRIGGER 权限是在 SQL99 中引入的。RULE 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 [, ...] ) | SELECT query }
```

输入

table

一个已存在表的名称。

column

*table*中一个列的名称。

DEFAULT VALUES

所有列都将用 NULL 或创建表时用 DEFAULT 子句指定的值填充。

expression

要赋予*column*的一个有效表达式或值。

query

一个有效的查询。有效参数的进一步描述请参考 SELECT 语句。

输出

`INSERT oid 1`

只插入了一行时返回的消息。→ *oid* 是被插入行的数字 OID。

`INSERT 0 #`

插入了多行时返回的消息。→ *#* 是插入的行数。

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，即声明的 DEFAULT 值或 NULL。如果向声明为 NOT NULL 的列插入 NULL，PostgreSQL将拒绝这个新列。

如果每个列的表达式数据类型不正确，将尝试自动类型强制转换。

要向一个表追加数据，你必须拥有该表上的 insert 权限，并且对 WHERE 子句中指定的任何表拥有 select 权限。

用法

向表 `films` 插入一行：

```
INSERT INTO films VALUES
    ('UA502','Bananas',105,'1971-07-13','Comedy',INTERVAL '82 minute')
;
```

在第二个例子中，省略了最后一列 `len`，因此它将使用默认值 NULL：

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DATE '1961-06-16', 'Drama');
```

向表 distributors 插入一行；注意只指定了列 `name`，因此被省略的列 `did` 将被赋予其默认值：

```
INSERT INTO distributors (name) VALUES ('British Lion');
```

从表 `tmp` 向表 films 插入若干行：

```
INSERT INTO films SELECT * FROM tmp;
```

插入数组（关于数组的更多信息请参考PostgreSQL User's Guide）：

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
-- (all of these queries create the same board attribute)
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

SQL92

`INSERT`与SQL92完全兼容。*`query`*子句特性方面可能存在的限制记录在SELECT中。

LISTEN

LISTEN — 监听一个通知

大纲

```
LISTEN name
```

输入

name

通知条件的名称。

输出

LISTEN

注册成功完成时返回的消息。

NOTICE Async_Listen: We are already listening on *name*

如果该后端已经注册了该通知条件。

描述

LISTEN把当前的 PostgreSQL后端注册为通知条件 *name*的监听者。

每当命令 NOTIFY *name* 被调用时--无论是被本后端还是被连接到同一数据库的其他后端调用--所有当前正在监听该通知条件的后端都会被通知，
而每个后端又将进而通知它所连接的前端应用。更多信息请参阅 NOTIFY的讨论。

可以用UNLISTEN命令取消一个后端对给定通知条件的注册。此外，
后端的监听注册在后端进程退出时会被自动清除。

前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。使用基本的 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例程 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库的文档。

NOTIFY 中包含关于LISTEN和 NOTIFY用法的更广泛的讨论。

注解

name 可以是任何可作为名称使用的有效字符串；它不需要对应任何实际表的名称。如果 *notifyname* 被双引号括起来，它甚至不必是语法上有效的名称，而可以是长度不超过 31 个字符的任何字符串。

在PostgreSQL的一些早期版本中，当*name* 不对应任何现有表名时，即使它在语法上是有效的名称，也必须用双引号把它括起来。现在已不再需要这样。

用法

在psql中配置并执行一个 listen/notify 序列：

```
LISTEN virtual;
NOTIFY virtual;

Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.
```

兼容性

SQL92

`LISTEN` 在 SQL92 中没有。

LOAD

LOAD — 装载或重新装载一个共享库文件

大纲

```
LOAD 'filename'
```

描述

把一个共享库文件装载到 PostgreSQL 后端的地址空间中。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后端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。更多细节见程序员指南。

兼容性

LOAD是PostgreSQL 的一个扩展。

另见

CREATE FUNCTION, PostgreSQL 程序员指南

LOCK

LOCK — 显式锁定一个表

大纲

```
LOCK [ TABLE ] name [, ...]
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
```

Inputs

name

要锁定的现有表的名称。

ACCESS SHARE MODE

> **注意**
>
> 这种锁模式会在被查询的表上自动获取。

这是限制性最低的锁模式。它只与 ACCESS EXCLUSIVE 模式冲突。它用于保护表不被并发的 `ALTER TABLE`、`DROP TABLE` 和 `VACUUM FULL` 命令修改。

ROW SHARE MODE

> **注意**
>
> 由 `SELECT ... FOR UPDATE` 自动获取。

与 EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。

ROW EXCLUSIVE MODE

> **注意**
>
> 由 `UPDATE`、`DELETE` 和 `INSERT` 语句自动获取。

与 SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。

SHARE UPDATE EXCLUSIVE MODE

> **注意**
>
> 由（不带 FULL 的）VACUUM 自动获取。

与 SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。此模式保护表不被并发的模式更改和 VACUUM 影响。

SHARE MODE

> **注意**
>
> 由 CREATE INDEX 自动获取。对整张表加共享锁。

与 ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。此模式保护表不被并发数据更新影响。

SHARE ROW EXCLUSIVE MODE

> **注意**
>
> 这与 EXCLUSIVE MODE 类似，但允许别人持有 ROW SHARE 锁。

与 ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。

EXCLUSIVE MODE

> **注意**
>
> 此模式比 SHARE ROW EXCLUSIVE 限制性更强。它阻塞所有并发的 ROW SHARE/SELECT...FOR UPDATE 查询。

与 ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。此模式只允许并发的 ACCESS SHARE，也就是说，只有对该表的读取才能与持有此锁模式的事务并行进行。

ACCESS EXCLUSIVE MODE

> **注意**
>
> 由 ALTER TABLE、DROP TABLE、VACUUM FULL 语句自动获取。这是限制性最高的锁模式，保护被锁表免受任何并发操作。

> **注意**
>
> 不带限定的 LOCK TABLE（即没有显式锁模式选项的命令）也会获取这种锁模式。

与所有锁模式冲突。

Outputs

LOCK TABLE

锁获取成功。

ERROR *name*: Table does not exist.

如果 *name* 不存在则返回此消息。

描述

LOCK TABLE 在事务期间控制对一个表的并发访问。PostgreSQL 在可能时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。

RDBMS 锁定使用下列术语：

EXCLUSIVE

排他锁阻止授予其他同类型的锁。（注意：ROW EXCLUSIVE 模式没有完全遵循这一命名约定，因为它在表级别是共享的；只对正在被更新的特定行是排他的。）

SHARE

共享锁允许别人也持有同类型的锁，但阻止授予相应的 EXCLUSIVE 锁。

ACCESS

锁定表模式。

ROW

锁定单个行。

例如，假设一个应用以 READ COMMITTED 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存在。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操作看到处于实际当前状态的数据，因为 SHARE 锁模式与写者取得的任何 ROW EXCLUSIVE 锁冲突，而你的 LOCK TABLE *name* IN SHARE MODE 语句会等待任何并发写操作提交或回滚。因此，一旦你取得了锁，就不存在未提交的写。

> **注意**
>
> 要在以 SERIALIZABLE 隔离级别运行事务时读取处于实际当前状态的数据，你必须在执行任何 DML 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 DML 语句开始时冻结。

除上述要求外，如果事务要更改表中的数据，就应取得 SHARE ROW EXCLUSIVE 锁模式，以防止两个并发事务都试图以 SHARE 模式锁定该表、然后再试图更改表中数据时出现死锁--两者（隐式）取得的 ROW EXCLUSIVE 锁模式都与并发的 SHARE 锁冲突。

继续讨论上面提出的死锁（两个事务互相等待）
问题：你应当遵循两条通用规则来防止死锁条件：

- 事务必须以相同的顺序对相同的对象获取锁。

  例如，如果一个应用先更新行 R1 再更新行 R2（在同一个事务中），那么第二个应用如果稍后要更新行 R1（在单个事务中），就不应先更新行 R2。相反，它应按照与第一个应用相同的顺序更新行 R1 和 R2。

- 只有两个冲突锁模式之一是自冲突的（即同一时刻只能被一个事务持有）时，
  事务才应同时获取这两个锁模式。如果涉及多种锁模式，
  事务应总是先获取限制性最高的模式。

  前面在讨论使用 SHARE ROW EXCLUSIVE 模式而非 SHARE 模式时已给出过这条规则的一个例子。

> **注意**
>
> PostgreSQL 确实会检测死锁，并会回滚至少一个等待中的事务来解决死锁。

锁定多个表时，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。

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`）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。

用法

演示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

SQL92

SQL92 中没有 `LOCK TABLE`，而是使用 `SET TRANSACTION` 来指定事务的并发级别。我们也支持这一点；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(TM) 中的对应语法兼容。

MOVE

MOVE — 把游标定位到表中的某一行

大纲

```
MOVE [ direction ] [ count ]
    { IN | FROM } cursor
```

描述

MOVE允许用户把游标位置移动指定的行数。MOVE的工作方式类似FETCH命令，但它只定位游标而不返回行。

有关语法和用法的细节，请参阅 FETCH。

注意

MOVE是一个PostgreSQL 语言扩展。

有效参数的描述请参阅 FETCH。定义游标请参阅 DECLARE。有关事务的更多信息，请参阅 BEGIN、COMMIT 和 ROLLBACK。

用法

建立并使用一个游标：

```
BEGIN WORK;
DECLARE liahona CURSOR  FOR SELECT * FROM films;
-- 跳过前 5 行:
MOVE FORWARD 5 IN liahona;
MOVE
-- 从游标 liahona 中提取第 6 行:
FETCH 1 IN liahona;
FETCH

 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 并提交事务:
CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL92

SQL92中没有MOVE语句。作为替代，SQL92允许从游标的绝对位置 FETCH行，从而隐式地把游标移动到正确的位置。

NOTIFY

NOTIFY — 生成一个通知

大纲

```
NOTIFY name
```

Inputs

notifyname

要发信号的通知条件。

Outputs

NOTIFY

确认 notify 命令已执行。

Notify events

事件被送达正在监听的前端；每个前端应用是否响应以及如何响应取决于它的程序设计。

描述

NOTIFY 命令向当前数据库中每个此前已为指定通知条件执行过 LISTEN *notifyname* 的前端应用发送一个通知事件。

传递给前端的通知事件信息包括通知条件名和发出通知的后端进程的 PID。由数据库设计者来定义给定数据库中将要使用的条件名以及每个条件名的含义。

通常，通知条件名与数据库中某张表的名称相同，通知事件本质上意味着“我改动了这张表，看看它有什么新内容”。但 NOTIFY 和 LISTEN 命令并不强制这种关联。例如，数据库设计者可以用几个不同的条件名来表示对单个表不同种类的更改。

NOTIFY 为访问同一 PostgreSQL 数据库的一组进程提供了一种简单的信号或 IPC（进程间通信）机制。通过使用数据库中的表，可以构建从通知者向监听者传递附加数据（而不仅仅是条件名）的更高层机制。

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，我们希望其中的所有命令都不产生效果，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后端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前端。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——但后端一旦把通知发给了前端，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。

发送 `NOTIFY` 的前端自己同时也在监听同一通知名，这是很常见的情形。在这种情况下，它会像其他所有监听前端一样收到一个通知事件。根据应用逻辑，
这可能导致无用功--例如再次读取数据库表，找到的却正是该前端自己刚刚写入的更新。在 PostgreSQL 6.4 及之后的版本中，可以通过检查发出通知的后端进程的 PID（在通知事件消息中提供）是否与自己的后端的 PID（可从 libpq 获得）相同来避免这种额外工作。二者相同时，该通知事件就是自己发的工作回弹回来，可以忽略。（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后端的通知分开保存，
因此忽略自己的通知并不会让你错过来自外部的通知。）

注意

name 可以是任何作为名称有效的字符串；它不必对应任何实际表的名称。如果 *name* 用双引号括起，它甚至不必是语法上有效的名称，而可以是长度最多 31 个字符的任意字符串。

在 PostgreSQL 的一些早期版本中，当 *name* 不对应任何现有表名时，即使语法上是有效的名称，也必须用双引号括起。这一要求已不再存在。

在 6.4 之前的 PostgreSQL 版本中，通知消息中传递的后端 PID 总是前端自己的后端的 PID。因此在那些早期版本中，无法把自己的通知与其他客户的通知区分开。

用法

在psql中配置并执行一组 listen/notify 操作：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.
```

兼容性

SQL92

SQL92 中没有 `NOTIFY` 语句。

REINDEX

REINDEX — 重建损坏的索引

大纲

```
REINDEX { TABLE | DATABASE | INDEX } name [ FORCE ]
```

输入

TABLE

重建指定表的所有索引。

DATABASE

重建指定数据库的所有系统索引。（不包括用户表上的索引。）

INDEX

重建一个指定的索引。

name

要重建索引的特定表/数据库/索引的名称。

FORCE

强制重建系统索引。不带此关键字时，REINDEX将跳过未被标记为无效的系统索引。FORCE对于REINDEX INDEX或者重建用户索引的情况没有作用。

输出

REINDEX

表成功重建索引后返回的消息。

描述

REINDEX用于重建损坏的索引。尽管理论上这永远不应该成为必要，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。

如果你怀疑用户表上的某个索引已损坏，可以简单地使用 REINDEX INDEX或REINDEX TABLE 重建该索引或该表的所有索引。

> **注意**
>
> 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，
重要的是执行恢复的后端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后端在启动时立即崩溃，因为它们依赖了损坏的索引。）
要安全地恢复，必须关闭 postmaster，改为启动一个独立的PostgreSQL后端，并在它的命令行中给出选项 -O 和 -P（这两个选项分别允许修改系统表和阻止使用系统索引）。然后根据你想重建多少，发出 REINDEX INDEX、REINDEX TABLE或 REINDEX DATABASE。如果不确定，使用 REINDEX DATABASE FORCE强制重建该数据库中的所有系统索引。然后退出独立后端并重启postmaster。

由于这可能是大多数人使用独立后端的唯一场合，这里给出一些用法提示也许有帮助：

- 用类似下面的命令启动后端：

  postgres -D $PGDATA -O -P my_database

  用-D提供数据库区域的正确路径，或者确保已设置环境变量PGDATA。
  同时指定你想要在其中工作的特定数据库的名称。

- 你可以发出任何 SQL 命令，而不仅是REINDEX。
- 要注意，独立后端把换行符当作命令输入的终止符；它不像 psql那样对分号有智能处理。要把一个命令延续到多行，必须在除最后一个之外的每个换行符之前输入反斜线。此外，
  你也不会有任何 readline 处理的便利（例如没有命令历史）。
- 要退出后端，输入 EOF（通常是 control-D）。

更多信息见postgres参考页。

用法

重建表mytable上的索引：

```
 REINDEX TABLE mytable;
```

重建单个索引：

```
REINDEX INDEX my_index;
```

重建所有系统索引（这只在独立后端中可行）：

```
REINDEX DATABASE my_database FORCE;
```

兼容性

SQL92

REINDEX 在 SQL92 中没有。

RESET

RESET — 将运行时参数的值恢复为默认值

大纲

```
RESET variable

RESET ALL
```

输入

variable

运行时参数的名称。列表请参阅SET。

ALL

把所有运行时参数重置为默认值。

描述

RESET将运行时参数恢复为其默认值。细节请参阅SET。RESET是下面语句的替代形式

```
SET variable TO DEFAULT
```

诊断

请参阅SET命令一节。

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 { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES |
 TRIGGER } [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] object [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
```

描述

`REVOKE` 允许一个对象的创建者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。

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。

权限类型的含义见GRANT命令的描述。

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。

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

SQL92

GRANT命令的兼容性说明同样适用于 `REVOKE`。语法摘要是：

```
REVOKE [ GRANT OPTION FOR ] { SELECT | INSERT | UPDATE | DELETE |
 REFERENCES }
    ON object [ ( column [, ...] ) ]
    FROM { PUBLIC | username [, ...] }
    { RESTRICT | CASCADE }
```

如果 user1 把一个权限带 GRANT OPTION 地授予 user2，而 user2 又把它授予 user3，那么 user1 可以使用 CASCADE 关键字级联回收该权限。如果 user1 把一个权限带 GRANT OPTION

地授予 user2，而 user2 又把它授予 user3，那么当 user1 试图回收该权限时，如果指定了 RESTRICT 关键字，回收将失败。

另见

GRANT

ROLLBACK

ROLLBACK — 中止当前事务

大纲

```
ROLLBACK [ WORK | TRANSACTION ]
```

输入

无。

输出

`ABORT`

成功时返回的消息。

`NOTICE: ROLLBACK: no transaction in progress`

如果当前没有正在进行的事务。

描述

`ROLLBACK` 回滚当前事务并使该事务所做的所有更新被丢弃。

注意

使用 COMMIT 成功地终止一个事务。ABORT 是 `ROLLBACK` 的一个同义词。

用法

要中止所有更改：

```
ROLLBACK WORK;
```

兼容性

SQL92

SQL92 只指定了 `ROLLBACK` 和 `ROLLBACK WORK` 两种形式。除此之外完全兼容。

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 ] [, ...] ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]

where from_item can be:

[ ONLY ] table_name [ * ]
    [ [ AS ] alias [ ( column_alias_list ) ] ]
|
( select )
    [ AS ] alias [ ( column_alias_list ) ]
|
from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column_list ) ]
```

输入

expression

表的列名或一个表达式。

output_name

用 AS 子句为输出列指定另一个名称。此名称主要用于为列提供一个便于显示的标签。它也可以用于在 ORDER BY 和 GROUP BY 子句中引用该列的值。但 *output_name* 不能用于 WHERE 或 HAVING 子句；应改写该表达式。

from_item

一个表引用、子 SELECT 或 JOIN 子句。细节见下文。

condition

一个结果为真或假的布尔表达式。见下面的 WHERE 和 HAVING 子句描述。

select

一个除 ORDER BY、FOR UPDATE 和 LIMIT 子句外具有所有特性的 select 语句（当 select 用圆括号括起时，甚至这些也可以使用）。

FROM 项可以包含：

table_name

一个现有表或视图的名称。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。）

alias

前面 *table_name* 的替代名称。别名用于简洁或消除自连接（同一表被扫描多次）的歧义。如果写了别名，还可以写一个列别名列表来为该表的一列或多列提供替代名称。

select

子 SELECT 可以出现在 FROM 子句中。它的行为就像其输出在此单条 SELECT 命令期间被创建为临时表一样。注意子 SELECT 必须用圆括号括起，并且必须为它提供别名。

join_type

下列之一：`[ INNER ] JOIN`, `LEFT [ OUTER ] JOIN`, `RIGHT [ OUTER ] JOIN`, `FULL [ OUTER ] JOIN`, or `CROSS JOIN`. 对于 INNER 和 OUTER 连接类型，必须恰好出现 NATURAL、ON *join_condition* 或 USING (*join_column_list*) 之一。对于 CROSS JOIN，这些项都不能出现。

join_condition

一个限定条件。它类似于 WHERE 条件，只是它只应用于在此 JOIN 子句中连接的两个 from_item。

join_column_list

USING 列列表 (a, b, ...) 是 ON 条件 left_table.a = right_table.a AND left_table.b = right_table.b ... 的简写。

输出

Rows

由查询说明产生的完整行集合。

→ *count*

查询返回的行数。

描述

`SELECT` 将从一个或多个表返回行。选择的候选是满足 WHERE 条件的行；如果省略 WHERE，则所有行都是候选。(See WHERE 子句.)

实际上，返回的行并不直接是 FROM/WHERE/GROUP BY/HAVING 子句产生的行；输出行是通过对每个选中行计算 SELECT 输出表达式而形成的。`*` 可以作为所有选中行的列的简写在输出列表中写。还可以写 *table_name*`.*` 作为只来自该表的列的简写。

`DISTINCT` 将从结果中消除重复行。`ALL`（默认）将返回所有候选行，包括重复行。

DISTINCT ON 消除在所有指定表达式上匹配的行，只保留每组重复行中的第一行。DISTINCT ON 表达式使用与 ORDER BY 项相同的规则解释；见下文。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 weatherReports
ORDER BY location, time DESC;
```

它取回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，就会得到每个位置年龄不可预测的报告。

GROUP BY 子句允许用户把一个表划分为在一个或多个值上匹配的行组。(See GROUP BY 子句.)

HAVING 子句允许只选择满足指定条件的那些行组。(See HAVING 子句.)

ORDER BY 子句使返回的行按指定的顺序排序。如果没有给出 ORDER BY，行按系统认为产生成本最低的任何顺序返回。(See ORDER BY 子句.)

SELECT 查询可以用 UNION、INTERSECT 和 EXCEPT 操作符组合。必要时使用圆括号来确定这些操作符的顺序。

UNION 操作符计算所涉及查询返回的行的集合。除非指定 ALL，否则消除重复行。(See UNION 子句.)

INTERSECT 操作符计算两个查询共同的行。除非指定 ALL，否则消除重复行。(See INTERSECT 子句.)

EXCEPT 操作符计算第一个查询返回的行，但不返回第二个查询的行。除非指定 ALL，否则消除重复行。(See EXCEPT 子句.)

FOR UPDATE 子句允许 SELECT 语句对选中的行执行排他锁定。

LIMIT 子句允许把查询产生的行的一个子集返回给用户。(See LIMIT 子句.)

你必须对一个表拥有 SELECT 权限才能读取它的值（见 GRANT/REVOKE 语句）。

FROM 子句

FROM 子句为 SELECT 指定一个或多个源表。如果指定多个源，结果在概念上是所有源中所有行的笛卡尔积——但通常会添加限定条件把返回的行限制为笛卡尔积的一个小子集。

当 FROM 项是一个简单表名时，它隐式地包括该表的子表（继承子女）的行。ONLY 将抑制该表的子表的行。在 PostgreSQL 7.1 之前这是默认结果，添加子表是通过在表名后面附加 * 完成的。这一旧行为可以通过命令 SET SQL_Inheritance TO OFF; 获得。

FROM 项也可以是用圆括号括起的子 SELECT（注意子 SELECT 必须有别名子句！）。这是一个极其方便的特性，因为它是在单个查询中获得多级分组、聚合或排序的唯一方式。

最后，FROM 项可以是 JOIN 子句，它组合两个更简单的 FROM 项。（必要时使用圆括号来确定嵌套顺序。）

CROSS JOIN 或 INNER JOIN 是简单的笛卡尔积，与在 FROM 顶层列出两个项所得到的相同。CROSS JOIN 等价于 INNER JOIN ON (TRUE)，即没有行被限定移除。这些连接类型只是记法上的便利，因为它们没有做任何用普通 FROM 和 WHERE 做不到的事。

LEFT OUTER JOIN 返回限定笛卡尔积中的所有行（即通过其 ON 条件的所有组合行），再加上左表中每一个没有通过 ON 条件的右行的行的一个副本。这个左表行通过为右表列插入 NULL 扩展到被连接表的全宽度。注意，在决定哪些行有匹配时只考虑 JOIN 自己的 ON 或 USING 条件。外层的 ON 或 WHERE 条件在此之后应用。

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）。

对于除 CROSS JOIN 外的所有 JOIN 类型，必须恰好写下列之一：ON *join_condition*, USING (*join_column_list*), or NATURAL. ON 是最一般的情况：可以写涉及要连接的两个表的任何限定表达式。USING 列列表 (a, b, ...) 是 ON 条件 left_table.a = right_table.a AND left_table.b = right_table.b ... 的简写。另外，USING 隐含着每对等价列中只有一列会包含在 JOIN 输出中，而不是两列都包含。NATURAL 是一个提及两表中所有同名列的 USING 列表的简写。

WHERE 子句

可选的 WHERE 条件具有如下一般形式：

```
WHERE boolean_expr
```

boolean_expr 可以由任何求值为布尔值的表达式组成。在许多情况下，此表达式会是：

exprcond_opexpr

或

log_opexpr

其中 *cond_op* 可以是 =、<、<=、>、>= 或 <> 之一，或像 ALL、ANY、IN、LIKE 这样的条件操作符，或本地定义的操作符，而 *log_op* 可以是 AND、OR、NOT 之一。SELECT 将忽略所有 WHERE 条件不返回 TRUE 的行。

GROUP BY 子句

GROUP BY 指定通过应用此子句导出的分组表：

```
GROUP BY expression [, ...]
```

GROUP BY 把在分组列上共享相同值的所有选中行压缩为单行。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 输出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。

GROUP BY 项可以是输入列名、输出列（SELECT 表达式）的名称或序号，也可以是由输入列值构成的任意表达式。在有歧义时，GROUP BY 名称将被解释为输入列名而不是输出列名。

HAVING 子句

可选的 HAVING 条件具有如下一般形式：

```
HAVING boolean_expr
```

其中 *boolean_expr* 与为 WHERE 子句指定的相同。

HAVING 指定通过消除不满足 *boolean_expr* 的组行而导出的分组表。HAVING 与 WHERE 不同：WHERE 在应用 GROUP BY 之前过滤单独的行，而 HAVING 过滤 GROUP BY 创建的组行。

boolean_expr 中引用的每一列都应无歧义地引用一个分组列，除非该引用出现在聚合函数内。

ORDER BY 子句

```
ORDER BY expression [ ASC | DESC | USING operator ] [, ...]
```

ORDER BY 项可以是输出列（SELECT 表达式）的名称或序号，也可以是由输入列值构成的任意表达式。在有歧义时，ORDER BY 名称将被解释为输出列名。

序号指结果列的（从左到右的）位置。这一特性使得可以基于一个没有恰当名称的列定义排序。但这从来不是绝对必要的，因为总是可以用 AS 子句为结果列指定一个名称，例如：

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

也可以按任意表达式 ORDER BY（对 SQL92 的扩展），包括不出现在 SELECT 结果列表中的字段。因此下面的语句是合法的：

```
SELECT name FROM distributors ORDER BY code;
```

这一特性的一个限制是，应用于 UNION、INTERSECT 或 EXCEPT 查询结果的 ORDER BY 子句只能指定输出列名或编号，不能指定表达式。

注意，如果 ORDER BY 项是一个既匹配结果列名又匹配输入列名的简单名称，ORDER BY 将把它解释为结果列名。这与 GROUP BY 在相同情况下所作的选择相反。这种不一致是 SQL92 标准规定的。

还可以在 ORDER BY 子句中的每个列名后面添加关键字 DESC（降序）或 ASC（升序）。如果未指定，默认为 ASC。或者，可以指定一个特定的排序操作符名称。ASC 等价于 USING <，DESC 等价于 USING >。

空值在域中排序高于任何其他值。换句话说，按升序排序时空值排在最后，
按降序排序时空值排在最前。

UNION 子句

```
table_query UNION [ ALL ] table_query
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

其中 *table_query* 指定不带 ORDER BY、FOR UPDATE 或 LIMIT 子句的任何 select 表达式。（如果子表达式用圆括号括起，ORDER BY 和 LIMIT 可以附加到它上面。不带圆括号时，这些子句将被认为应用于 UNION 的结果，而不是其右手输入表达式。）

UNION 操作符计算所涉及查询返回的行的集合（集合并集）。表示 UNION 的直接操作数的两个 SELECT 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。

目前，不能对 UNION 的结果或 UNION 的输入指定 FOR UPDATE。

INTERSECT 子句

```
table_query INTERSECT [ ALL ] table_query
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

其中 *table_query* 指定不带 ORDER BY、FOR UPDATE 或 LIMIT 子句的任何 select 表达式。

INTERSECT 类似于 UNION，只是它只产生在两个查询输出中都出现的行，而不是在任一输出中出现的行。

INTERSECT 的结果不包含任何重复行，除非指定了 ALL 选项。带 ALL 时，在 L 中有 m 个重复且在 R 中有 n 个重复的行将出现 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，除非圆括号另有指定，A UNION B INTERSECT C 将被读作 A UNION (B INTERSECT C)。

EXCEPT 子句

```
table_query EXCEPT [ ALL ] table_query
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

其中 *table_query* 指定不带 ORDER BY、FOR UPDATE 或 LIMIT 子句的任何 select 表达式。

EXCEPT 类似于 UNION，只是它只产生在左查询输出中出现而在右查询输出中不出现的行。

EXCEPT 的结果不包含任何重复行，除非指定了 ALL 选项。带 ALL 时，在 L 中有 m 个重复且在 R 中有 n 个重复的行将出现 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。

LIMIT 子句

```
LIMIT { count | ALL }
OFFSET start
```

其中 count 指定返回的最大行数，start 指定开始返回行之前要跳过的行数。

LIMIT 允许只取回由查询其余部分产生的行的一部分。如果给出了限制数，则返回不超过那么多的行。如果给出了偏移，则在开始返回行之前跳过那么多的行。

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。

从 PostgreSQL 7.0 起，查询优化器在生成查询计划时会把 LIMIT 考虑在内，所以很可能根据 LIMIT 和 OFFSET 的不同取值得到不同的计划（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会得到不一致的结果，除非用 ORDER BY 强制可预测的结果顺序。这不是缺陷；这是 SQL 不承诺以任何特定顺序交付查询结果（除非用 ORDER BY 约束顺序）这一事实的固有后果。

用法

把表 films 与表 distributors 连接：

```
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
 Une Femme est une Femme   | 102 | Jean Luc Godard  | 1961-03-12 |
 Romantic
 Vertigo                   | 103 | Paramount        | 1958-11-14 |
 Action
 Becket                    | 103 | Paramount        | 1964-02-03 |
 Drama
 48 Hrs                    | 103 | Paramount        | 1982-10-22 |
 Action
 War and Peace             | 104 | Mosfilm          | 1967-02-12 |
 Drama
 West Side Story           | 105 | United Artists   | 1961-01-03 |
 Musical
 Bananas                   | 105 | United Artists   | 1971-07-13 |
 Comedy
 Yojimbo                   | 106 | Toho             | 1961-06-16 |
 Drama
 There's a Girl in my Soup | 107 | Columbia         | 1970-06-11 |
 Comedy
 Taxi Driver               | 107 | Columbia         | 1975-05-15 |
 Action
 Absence of Malice         | 107 | Columbia         | 1981-11-15 |
 Action
 Storia di una donna       | 108 | Westward         | 1970-08-15 |
 Romantic
 The King and I            | 109 | 20th Century Fox | 1956-08-11 |
 Musical
 Das Boot                  | 110 | Bavaria Atelier  | 1981-11-11 |
 Drama
```

```
 Bed Knobs and Broomsticks | 111 | Walt Disney      |            |
 Musical
(17 rows)
```

对所有影片的 len 列求和并按 kind 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
(5 rows)
```

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 hour';

 kind     | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
(2 rows)
```

下面两个例子是按照第二列（name）的内容对各个结果排序的相同方式：

```
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
(13 rows)
```

这个例子展示如何获得表 distributors 和 actors 的并集，把结果限制为每个表中以字母 W 开头的行。只需要不重复的行，所以省略了 ALL 关键字：

```
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
```

兼容性

扩展

PostgreSQL 允许在查询中省略 FROM 子句。此特性是从最初的 PostQuel 查询语言保留下来的。它有一个直接的用途：计算简单常量表达式的结果：

```
SELECT 2+2;
```

```
 ?column?
----------
        4
```

一些其他 DBMS 做不到这一点，除非引入一个单行的哑表来从中做 SELECT。一个不太明显的用法是缩写对一个或多个表的普通 SELECT：

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

```
 did | name
-----+----------
 108 | Westward
```

这之所以可行，是因为对查询中引用但在 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，查询

```
SELECT distributors.* FROM distributors d;
```

很可能是笔误；用户最可能想要的是

```
SELECT d.* FROM distributors d;
```

而不是他实际会得到的无约束连接

```
SELECT distributors.* FROM distributors d, distributors distributors;
```

为了帮助检测这类错误，PostgreSQL 7.1 及以后版本在查询同时使用了隐式 FROM 特性和显式 FROM 子句时将发出警告。

SQL92

SELECT 子句

在 SQL92 标准中，可选的关键字 AS 只是无意义的噪音，可以省略而不影响含义。PostgreSQL 解析器在重命名输出列时要求这个关键字，因为类型可扩展特性在此上下文中会导致解析歧义。不过 AS 在 FROM 项中是可选的。

DISTINCT ON 短语不是 SQL92 的一部分。LIMIT 和 OFFSET 也不是。

在 SQL92 中，ORDER BY 子句只能使用结果列名或编号，而 GROUP BY 子句只能使用输入列名。PostgreSQL 对这两个子句都做了扩展，允许也使用另一种选择（但在有歧义时采用标准的解释）。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将总是被当作输入列名，而不是结果列名。

UNION/INTERSECT/EXCEPT 子句

SQL92 的 UNION/INTERSECT/EXCEPT 语法允许一个附加的 CORRESPONDING BY 选项：

```
table_query UNION [ALL]
    [CORRESPONDING [BY (column [,...])]]
    table_query
```

PostgreSQL 不支持 CORRESPONDING BY 子句。

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 ] [, ...] ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]
    [ LIMIT [ start , ] { count | ALL } ]
    [ OFFSET start ]

where from_item can be:

[ ONLY ] table_name [ * ]
    [ [ AS ] alias [ ( column_alias_list ) ] ]
|
( select )
    [ AS ] alias [ ( column_alias_list ) ]
|
from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column_list ) ]
```

输入

TEMPORARY
TEMP

如果指定了 TEMPORARY 或 TEMP，输出表只在本次会话中创建，并在会话退出时自动删除。在临时表存在期间，（本会话中）同名的现有永久表不可见。
在临时表上创建的任何索引也自动是临时的。

new_table

要创建的新表的名称。该表必须尚不存在。但可以创建与现有永久表同名的临时表。

所有其他输入在SELECT中有详细描述。

输出

可能的输出消息的摘要请参阅CREATE TABLE和SELECT。

描述

SELECT INTO 创建一个新表，并用查询计算出的数据填充它。与普通的 SELECT 不同，数据不返回给客户端。新表的列具有与 SELECT 的输出列相关联的名称和数据类型。

> **注意**
>
> CREATE TABLE AS在功能上等价于 `SELECT INTO`。推荐使用 `CREATE TABLE AS` 语法，因为 `SELECT INTO` 不是标准的。事实上，这种形式的 `SELECT INTO` 在 PL/pgSQL 或 ecpg 中不可用，因为它们对 INTO 子句的解释不同。

兼容性

SQL92 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 PL/pgSQL 和 ecpg 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 variable { TO | = } { value | 'value' | DEFAULT }
SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }
```

输入

variable

一个可设置的运行时参数。

value

参数的新值。DEFAULT 可用于指定把参数重置为默认值。允许使用字符串列表，但更复杂的构造可能需要加单引号或双引号。

描述

SET 命令更改运行时配置参数。下列参数可以更改：

CLIENT_ENCODING
NAMES

设置多字节客户端编码。指定的编码必须为后端所支持。

这个选项只有在 PostgreSQL 以多字节支持构建时才可用。

DATESTYLE

选择日期/时间的表示风格。
这里进行两个独立的设置：默认的日期/时间输出和对有歧义输入的解释。

以下是日期/时间的输出风格：

ISO

使用 ISO 8601 风格的日期和时间（YYYY-MM-DD HH:MM:SS）。这是默认值。

SQL

使用 Oracle/Ingres 风格的日期和时间。注意这种风格与 SQL（它强制使用 ISO 8601 风格）毫无关系，这个选项的命名是一个历史意外。

PostgreSQL

使用传统的 PostgreSQL 格式。

German

数值日期表示使用 dd.mm.yyyy。

下面两个选项既决定 “SQL” 和 “PostgreSQL” 输出格式的子风格，
也决定有歧义日期输入的首选解释。

European

数值日期表示使用 dd/mm/yyyy。

NonEuropean
US

数值日期表示使用 mm/dd/yyyy。

SET DATESTYLE 的值可以取自第一个列表（输出风格），或第二个列表（子风格），或从两个列表中各取一个并用逗号分隔。

日期格式的初始化可以通过以下方式完成：

设置 PGDATESTYLE 环境变量。如果在基于 libpq 的客户端前端环境中设置了 PGDATESTYLE，libpq 会在连接启动时自动把 DATESTYLE 设为 PGDATESTYLE 的值。
使用选项 -o -e 运行 postmaster，把日期设为 European 约定。

DateStyle 选项其实只是为了移植应用而设的。要按你的选择格式化日期/时间值，请使用 to_char 函数族。

SEED

设置随机数生成器的内部种子。

value

random 函数使用的种子的值。允许的值是 0 到 1 之间的浮点数，之后会乘以 2^{31}-1。如果使用范围之外的数，该乘积会静默溢出。

也可以通过调用 setseed SQL 函数来设置种子：

```
SELECT setseed(value);
```

SERVER_ENCODING

设置多字节服务器编码。

这个选项只有在 PostgreSQL 以多字节支持构建时才可用。

TIME ZONE
TIME ZONE

设置会话的默认时区。参数可以是 SQL 时间间隔常量、整数或双精度常量，或者一个表示宿主操作系统所支持时区的字符串。

时区的可用值取决于你的操作系统。例如在 Linux 上，/usr/share/zoneinfo 包含时区数据库。

下面是一些有效的时区值：

'PST8PDT'

把时区设为加利福尼亚州。

'Portugal'

把时区设为葡萄牙。

'Europe/Rome'

把时区设为意大利。

7

把时区设为 GMT 以西 7 小时偏移（相当于 PDT）。

INTERVAL '08:00' HOUR TO MINUTE

把时区设为 GMT 以西 8 小时偏移（相当于 PST）。

LOCAL
DEFAULT

把时区设为你的本地时区（你的操作系统默认的时区）。

如果指定了无效的时区，时区会变成 GMT（至少在大多数系统上如此）。

如果在基于 libpq 的客户端前端环境中设置了 `PGTZ` 环境变量，libpq 会在连接启动时自动把 TIMEZONE 设为 `PGTZ` 的值。

其他运行时参数的完整列表见管理员指南。

使用SHOW显示参数的当前设置。

诊断

`SET VARIABLE`

成功时返回的消息。

`ERROR: not a valid option name: name`

你试图设置的参数不存在。

`ERROR: permission denied`

你必须是超级用户才能访问某些设置。

`ERROR: name can only be set at start-up`

有些参数在服务器启动后就是固定的。

示例

把日期风格设为带欧洲约定的传统 PostgreSQL 风格：

```
SET DATESTYLE TO PostgreSQL,European;
```

把时区设为加利福尼亚州伯克利，
使用双引号来保留时区说明符的大写属性（注意这里显示的日期/时间格式是 ISO）：

```
SET TIME ZONE "PST8PDT";
SELECT CURRENT_TIMESTAMP AS today;

         today
------------------------
 1998-03-31 07:41:21-08
```

把时区设为意大利（注意需要单引号或双引号来处理特殊字符）：

```
SET TIME ZONE 'Europe/Rome';
SELECT CURRENT_TIMESTAMP AS today;

         today
------------------------
 1998-03-31 17:41:31+02
```

兼容性

SQL92

上面所示的第二种语法（`SET TIME ZONE`）试图模仿 SQL92。但 SQL 只允许数值形式的时区偏移。所有其他参数设置以及上面所示的第一种语法都是 PostgreSQL 扩展。

SET CONSTRAINTS

SET CONSTRAINTS — 设置当前事务的约束模式

大纲

```
SET CONSTRAINTS { ALL | constraint [, ...] } { 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。

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。

兼容性

SQL92, SQL99

`SET CONSTRAINT` 定义于 SQL92 和 SQL99。

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 AUTHORIZATION 'username'
```

描述

这条命令把当前 SQL 会话上下文的会话用户标识符和当前用户标识符设置为*username*。

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“setuid”
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，才允许执行这条命令。
这一权限在连接期间一直保持；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。

示例

```
SELECT SESSION_USER, CURRENT_USER;
 current_user | session_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;
 current_user | session_user
--------------+--------------
 paul         | paul
```

兼容性

SQL99

SQL99　　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 }
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
    { READ COMMITTED | SERIALIZABLE }
```

描述

这条命令设置事务的隔离级别。`SET TRANSACTION` 命令设置当前 SQL 事务的特性。它对任何后续事务都没有影响。这条命令不能在事务的第一条查询或数据修改语句（`SELECT`、`INSERT`、`DELETE`、`UPDATE`、`FETCH`、`COPY`）已经执行之后使用。`SET SESSION CHARACTERISTICS` 设置一个会话中每个事务的默认事务隔离级别。对于单个事务，`SET TRANSACTION` 可以覆盖它。

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。

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。更多信息请参考管理员指南。

兼容性

SQL92, SQL99

SERIALIZABLE 是 SQL 中的默认级别。PostgreSQL 不提供隔离级别 `READ UNCOMMITTED` 和 `REPEATABLE READ`。由于多版本并发控制，可串行化级别并非真正的可串行化。细节见用户指南。

在 SQL 中还有另外两项可以用这些命令设置的事务特性：事务是否只读以及诊断区域的大小。这两个概念在 PostgreSQL 中都不受支持。

SHOW

SHOW — 显示一个运行时参数的值

大纲

```
SHOW name

SHOW ALL
```

输入

name

一个运行时参数的名称。列表见SET。

ALL

显示所有当前会话参数。

描述

`SHOW` 将显示一个运行时参数的当前设置。这些变量可以用 `SET` 语句设置，或者在服务器启动时确定。

诊断

`ERROR: not a valid option name: name`

如果 *variable* 不表示一个已存在的参数，则返回此消息。

`ERROR: permission denied`

你必须是超级用户才能查看某些设置。

`NOTICE: Time zone is unknown`

如果没有设置 `TZ` 或 `PGTZ` 环境变量。

示例

显示当前的 `DateStyle` 设置：

```
SHOW DateStyle;
NOTICE:  DateStyle is ISO with US (NonEuropean) conventions
```

显示当前的遗传优化器（`geqo`）设置：

```
SHOW GEQO;
NOTICE:  geqo is on
```

兼容性

SHOW命令是一种 PostgreSQL扩展。

TRUNCATE

TRUNCATE — 清空一个表

大纲

```
TRUNCATE [ TABLE ] name
```

输入

name

要截断的表的名字。

输出

`TRUNCATE`

如果表被成功截断，返回此消息。

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。

`TRUNCATE`不能在一个事务块（`BEGIN`/`COMMIT`对）中执行，因为它没有办法回滚。

用法

截断表`bigtable`：

```
TRUNCATE TABLE bigtable;
```

兼容性

SQL92

SQL92中没有`TRUNCATE`。

UNLISTEN

UNLISTEN — 停止监听某个通知

大纲

```
UNLISTEN { notifyname | * }
```

输入

notifyname

先前注册的通知条件的名称。

*

清除该后端当前的所有监听注册。

输出

→ UNLISTEN

语句已执行的确认。

描述

UNLISTEN 用于移除一个现有的NOTIFY注册。UNLISTEN 取消当前 PostgreSQL会话作为通知条件 *notifyname*监听者的任何现有注册。特殊的条件通配符*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。

注解

notifyname 不需要是有效的类名，而可以是任何可作为名称使用的、长度不超过 32 个字符的字符串。

后端不会因为你 UNLISTEN 一个你并没有在监听的东西而抱怨。每个后端退出时都会自动执行UNLISTEN *。

用法

订阅一个现有的注册：

```
LISTEN virtual;
LISTEN
NOTIFY virtual;
NOTIFY
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received
```

一旦执行了 UNLISTEN，后续的 NOTIFY 命令会被忽略：

```
UNLISTEN virtual;
```

```
UNLISTEN
NOTIFY virtual;
NOTIFY
-- notice no NOTIFY event is received
```

兼容性

SQL92

`UNLISTEN` 在 SQL92 中没有。

UPDATE

UPDATE — 更新表中的行

大纲

```
UPDATE [ ONLY ] table SET col = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

输入

table

一个现有表的名称。

column

table 中一个列的名称。

expression

赋给列的一个有效表达式或值。

fromlist

一个 PostgreSQL 非标准扩展，允许来自其他表的列出现在 WHERE 条件中。

condition

WHERE 子句的进一步描述请参阅 SELECT 语句。

输出

`UPDATE #`

成功时返回的消息。*#* 表示被更新的行数。如果 *#* 为 0，则没有行被更新。

描述

`UPDATE` 更改所有满足条件的行中指定列的值。只有要修改的列才需要作为列出现在语句中。

数组引用使用与SELECT中相同的语法。也就是说，可以用单次查询替换单个数组元素、数组元素的一个范围或整个数组。

要修改表，你必须拥有对它的写权限，以及对 WHERE 条件中提到其值的任何表的读权限。

默认情况下，UPDATE 将更新指定的表及其所有子表中的元组。如果只想更新提到的这个特定表，应当使用 ONLY 子句。

用法

把列 `kind` 上的单词 `Drama` 改为 `Dramatic`：

```
UPDATE films
SET kind = 'Dramatic'
WHERE kind = 'Drama';
SELECT *
FROM films
WHERE kind = 'Dramatic' OR kind = 'Drama';

 code  |     title     | did | date_prod  |   kind   | len
-------+---------------+-----+------------+----------+-------
 BL101 | The Third Man | 101 | 1949-12-23 | Dramatic | 01:44
 P_302 | Becket        | 103 | 1964-02-03 | Dramatic | 02:28
 M_401 | War and Peace | 104 | 1967-02-12 | Dramatic | 05:57
 T_601 | Yojimbo       | 106 | 1961-06-16 | Dramatic | 01:50
 DA101 | Das Boot      | 110 | 1981-11-11 | Dramatic | 02:29
```

兼容性

SQL92

SQL92 为定位式 UPDATE 语句定义了一种不同的语法：

```
UPDATE table SET column = expression [, ...]
    WHERE CURRENT OF cursor
```

其中 *cursor* 标识一个打开的游标。

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 [, ..
.] ) ] ]
```

输入

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

更新优化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。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

→ VACUUM

命令已完成。

NOTICE: --Relation *table*--

*table*的报告头。

NOTICE: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188; Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail. Pages 0/74. Elapsed 0/0 sec.

对*table*本身的分析。

NOTICE: Index *index*: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/0 sec.

对目标表上某个索引的分析。

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 DELETEd 或被 UPDATE 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。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。细节见管理员指南。

注意

我们建议对活跃的生产数据库频繁地执行VACUUM（至少每晚一次），以移除过期的行。
在增加或删除大量记录之后，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查询优化器在规划用户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。

用法

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;
NOTICE:  --Relation onek--
NOTICE:  Index onek_unique1: Pages 14; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.11u sec elapsed 0.12 sec.
NOTICE:  Index onek_unique2: Pages 16; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.10u sec elapsed 0.10 sec.
NOTICE:  Index onek_hundred: Pages 13; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.10u sec elapsed 0.10 sec.
NOTICE:  Index onek_stringu1: Pages 31; Tuples 1000: Deleted 3000.
        CPU 0.01s/0.09u sec elapsed 0.10 sec.
NOTICE:  Removed 3000 tuples in 70 pages.
        CPU 0.02s/0.04u sec elapsed 0.07 sec.
NOTICE:  Pages 94: Changed 0, Empty 0; Tup 1000: Vac 3000, Keep 0, Un
Used 0.
        Total CPU 0.05s/0.45u sec elapsed 0.59 sec.
NOTICE:  Analyzing onek
```

VACUUM

兼容性

SQL92

SQL92中没有VACUUM语句。

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

createdb 496
createlang 498
createuser 500
dropdb 502
droplang 504
dropuser 506
ecpg 508
pgaccess 512
pg_config 514
pg_dump 516
pg_dumpall 521
pg_restore 523
psql 529
pgtclsh 550
pgtksh 551
vacuumdb 552

createdb

createdb — 创建一个新的 PostgreSQL 数据库

大纲

`createdb` [*options*...] [*dbname*] [*description*]

输入

-h, --host *host*

指定服务器所在机器的主机名。如果 host 以斜线开头，它被用作 Unix 域套接字的目录。

-p, --port *port*

指定服务器正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。

-U, --username *username*

以该用户名进行连接。

-W, --password

强制提示输入密码。

-e, --echo

回显 createdb 生成并发送给服务器的查询。

-q, --quiet

不显示响应。

-D, --location *datadir*

指定数据库的备选位置。另见initlocation。

-T, --template *template*

指定用于构建该数据库的模板数据库。

-E, --encoding *encoding*

指定该数据库使用的字符编码方案。

dbname

指定要创建的数据库的名称。该名称在本安装中的所有 PostgreSQL 数据库中必须唯一。默认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。

The options `-h`, `-p`, `-U`, `-W`, and `-e` are passed on literally to psql. The options `-D`, `-T`, and `-E` are converted into options for the underlying SQL command CREATE DATABASE ; see there for more information about them.

输出

`CREATE DATABASE`

数据库创建成功。

`createdb: Database creation failed.`

（一目了然。）

`createdb: Comment creation failed. (Database was created.)`

无法创建数据库的注释/描述。数据库本身已经创建。可以以后用 SQL 命令 `COMMENT ON DATABASE` 创建注释。

If there is an error condition, the backend error message will be displayed. See CREATE DATABASE and psql for possibilities.

描述

createdb 创建一个新的 PostgreSQL 数据库。执行这个命令的用户成为该数据库的所有者。

createdb 是围绕 SQL 命令 CREATE DATABASE 的一个 shell 脚本包装，它通过 PostgreSQL 交互式终端psql执行。因此，用这种方法或其他方法创建数据库并没有什么特别之处。这意味着脚本必须能找到 psql 程序，而且目标端口上必须有数据库服务器在运行。此外，psql 和 libpq 前端库可用的任何默认设置和环境变量都将适用。

用法

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。

用主机 `eden` 上的服务器、端口 5000 创建数据库 `demo`，使用 `LATIN1` 编码方案并看一看底层查询：

```
$ createdb -p 5000 -h eden -E LATIN1 -e demoCREATE DATABASE "demo"
 WITH ENCODING = 'LATIN1'CREATE DATABASE
```

createlang

createlang — 定义一种新的 PostgreSQL 过程语言

大纲

```
createlang [connection-options...] langname [dbname]
createlang [connection-options...] [--list] | [-l] dbname
```

输入

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*

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

显示执行的 SQL 命令。

-l, --list

显示目标数据库（必须指定）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*

指定服务器所在机器的主机名。如果 host 以斜线开头，它被用作 Unix 域套接字的目录。

-p, --port *port*

指定服务器正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。

-U, --username *username*

要连接使用的用户名

-W, --password

强制提示输入密码。

输出

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。更多可能的情况也请检查psql。

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。

虽然可以直接用若干 SQL 命令添加后端编程语言，但建议使用 createlang，因为它会执行若干检查而且使用起来容易得多。更多信息见CREATE LANGUAGE。

注解

使用droplang可移除一个语言。

createlang 是一个多次调用 psql 的 shell 脚本。如果你的安排使得连接需要口令提示，你将被提示输入口令多次。

用法

把 `pltcl` 安装到数据库 `template1`：

```
$ createlang pltcl template1
```

createuser

createuser — 定义一个新的 PostgreSQL 用户账号

大纲

`createuser` [*options*...] [*username*]

输入

-h, --host *host*

指定服务器正在运行的主机名。如果 host 以斜杠开头，它将被用作 Unix 域套接字的目录。

-p, --port *port*

指定服务器正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。

-e, --echo

显示 createuser 生成并发送给服务器的查询。

-q, --quiet

不显示响应。

-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。

-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）。

-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。

-i, --sysid *uid*

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。

-E, --encrypted

加密存储在数据库中的用户口令。如果未指定，则使用默认值。

-N, --unencrypted

不加密存储在数据库中的用户口令。如果未指定，则使用默认值。

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。

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。

选项 -h、-p和 -e会按原样传递给psql。psql的选项 -U和 -W也可用，但在这种场合使用它们可能造成困惑。

输出

CREATE USER

一切正常。

createuser: creation of user "*username*" failed

出了些问题。用户没有被创建。

如果出现错误情况，将显示后端错误消息。参见 CREATE USER 和psql了解各种可能。

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 PostgreSQL超级用户来调用。

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是一个 shell 脚本，它通过 PostgreSQL交互式终端psql包装了 SQL命令 CREATE USER。因此，通过这个或其他方法创建用户没有什么特别之处。这意味着脚本必须能找到 psql，并且目标主机上必须有一个数据库服务器在运行。此外，psql和 libpq前端库使用的任何默认设置和环境变量都将适用。

用法

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查询：

```
$ createuser -p 5000 -h eden -D -A -e joeCREATE USER "joe" NOCREATEDB
 NOCREATEUSERCREATE USER
```

dropdb

dropdb — 移除一个 PostgreSQL 数据库

大纲

`dropdb` [*options*...] *dbname*

输入

-h, --host *host*

指定服务器正在运行的机器的主机名。如果 host 以斜杠开头，它被用作 Unix 域套接字的目录。

-p, --port *port*

指定服务器正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展。

-U, --username *username*

要连接的用户名。

-W, --password

强制口令提示。

-e, --echo

回显 dropdb 生成并发送给服务器的查询。

-q, --quiet

不显示响应。

-i, --interactive

在做任何破坏性操作之前发出一个确认提示。

dbname

指定要移除的数据库的名称。该数据库必须是这个安装中现有的 PostgreSQL 数据库之一。

选项 `-h`、`-p`、`-U`、`-W` 和 `-e` 会被原样传递给psql。

输出

`DROP DATABASE`

数据库已被成功移除。

`dropdb: Database removal failed.`

某些地方出了问题。

如果出现错误情况，将显示后端错误消息。各种可能性请参阅 DROP DATABASE 和psql。

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 的一个 shell 脚本包装，它通过 PostgreSQL 交互式终端psql进行。因此，通过这个或其他方法删除数据库没有什么特别的。这意味着脚本必须能找到 psql，并且目标主机上有一个数据库服务器在运行。此外，psql 和 libpq 前端库可用的任何默认设置和环境变量都适用。

用法

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查询：

```
$ dropdb -p 5000 -h eden -i -e demoDatabase "demo" will be permanently
 deleted.
Are you sure? (y/n) yDROP DATABASE "demo"
DROP DATABASE
```

droplang

droplang — 移除一种 PostgreSQL 过程语言

大纲

`droplang` [*connection-options*...] *langname* [*dbname*]
`droplang` [*connection-options*...] [--list] | [-l] *dbname*

输入

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*

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

显示执行的 SQL 命令。

-l, --list

显示目标数据库（必须指定）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*

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*

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*

要用来连接的用户名。

-W, --password

强制提示输入密码。

输出

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。更多可能的情况也请检查psql。

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。

注解

使用createlang可添加一个语言。

用法

要移除 `pltcl`：

```
$ droplang pltcl dbname
```

dropuser

dropuser — 移除一个 PostgreSQL 用户账户

大纲

`dropuser` [*options*...] [*username*]

输入

-h, --host *host*

指定服务器所在机器的主机名。如果 host 以斜线开头，它被用作 Unix 域套接字的目录。

-p, --port *port*

指定服务器正在监听连接的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。

-e, --echo

回显 createdb 生成并发送给服务器的查询。

-q, --quiet

不显示响应。

-i, --interactive

在实际移除用户之前提示确认。

username

指定要删除的 PostgreSQL 用户的名称。该名称必须存在于 PostgreSQL 安装中。如果命令行上没有指定名称，会提示你输入一个。

选项 `-h`、`-p` 和 `-e` 被原样传递给psql。psql 的选项 `-U` 和 `-W` 也可用，但在此上下文中可能造成混淆。

输出

`DROP USER`

一切正常。

`dropuser: deletion of user "`*username*`" failed`

出了问题。用户没有被删除。

If there is an error condition, the backend error message will be displayed. See DROP USER and psql for possibilities.

描述

dropuser 移除一个现有的 PostgreSQL 用户以及该用户所拥有的数据库。只有 `pg_shadow` 表中 `usesuper` 被设置的用户才能删除 PostgreSQL 用户。

dropuser 是围绕 SQL 命令 DROP USER 的一个 shell 脚本包装，它通过 PostgreSQL 交互式终端psql执行。因此，用这种方法或其他方法删除用户并没有什么特别之处。这意味着脚本必须能找到 psql，而且目标主机上有数据库服务器在运行。此外，psql 和 libpq 前端库可用的任何默认设置和环境变量都适用。

用法

要从默认数据库服务器上移除用户joe：

```
$ dropuser joeDROP USER
```

用主机 eden 上的 postmaster、端口 5000 删除用户 joe，带验证并看一看底层查询：

```
$ 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
```

ecpg

ecpg — 嵌入式 SQL C 预处理器

大纲

`ecpg` [-v] [-t] [-I *include-path*] [-o *outfile*] *file*...

输入

ecpg接受下列命令行参数：

-v

打印版本信息。

-t

打开事务自动提交。在此模式下，除非查询位于显式事务块中，
否则每条查询都会自动提交。在默认模式下，只有发出 `exec sql commit`时才会提交查询。

-I *include-path*

指定一个额外的 include 路径。默认值是 .（当前目录）、`/usr/local/include`、编译时定义的 PostgreSQL include 路径（默认：`/usr/local/pgsql/include`）以及 `/usr/include`。

-o *outfile*

指定 ecpg 应将全部输出写入 *outfile*。如果没有给出这个选项，输出将写入 *name*`.c`（假定输入文件名为 *name*`.pgc`）。如果输入文件确实没有预期的 `.pgc`后缀，那么输出文件名将在输入文件名后追加上 `.pgc`。

file

要处理的文件。

输出

ecpg会创建一个文件或写入 `stdout`。

Return value

ecpg成功完成时向 shell 返回 0，出错时返回非零值。

描述

ecpg 是 C 语言和 PostgreSQL的嵌入式 SQL 预处理器。它支持开发带有嵌入式 SQL 代码的 C 程序。

Linus Tolke（`<linus@epact.se>`）是 ecpg的原作者（直到 0.2 版）。Michael Meskes（`<meskes@debian.org>`）是 ecpg目前的作者和维护者。Thomas Good（`<tomg@q8.nrnet.org>`）是本文件所依据的 ecpg man 页最后一版的作者。

用法

为编译做预处理

嵌入式 SQL 源文件在编译之前必须先做预处理：

```
ecpg [ -d ] [ -o file ] file.pgc
```

其中可选的 `-d` 标志打开调试。`.pgc` 扩展名是表示 ecpg 源的一种随意约定。

你可能想把预处理器的输出重定向到一个日志文件。

编译和链接

假定 PostgreSQL 的可执行文件位于 `/usr/local/pgsql`，你需要这样编译和链接预处理后的源文件：

```
gcc -g -I /usr/local/pgsql/include [ -o file ] file.c -L /usr/local/
pgsql/lib -lecpg -lpq
```

语法

库

预处理器会在源文件开头加上两条指令：

```
#include <ecpgtype.h>
#include <ecpglib.h>
```

变量声明

在 ecpg 源代码中声明的变量必须以以下内容开头：

```
EXEC SQL BEGIN DECLARE SECTION;
```

类似地，变量声明节必须以以下内容结束：

```
EXEC SQL END DECLARE SECTION;
```

> **注意**
>
> 在 2.1.0 版之前，每个变量必须单独一行声明。从 2.1.0 版起，多个变量可以在一行中声明：
>
> ```
> char  foo[16], bar[16];
> ```

错误处理

SQL 通信区用以下方式定义：

```
EXEC SQL INCLUDE sqlca;
```

> **注意**
>
> sqlca是小写的。虽然可以遵循 SQL 惯例（即用大写来区分嵌入式 SQL 与 C 语句），但 sqlca（它包含 sqlca.h头文件）必须是小写的。这是因为 EXEC SQL 前缀表明这个包含将由 ecpg解析。ecpg 区分大小写（SQLCA.h将找不到）。只要注意大小写，EXEC SQL INCLUDE 也可用于包含其他头文件。

sqlprint命令与 EXEC SQL WHENEVER语句一起使用，在整个程序中打开错误处理：

```
EXEC SQL WHENEVER sqlerror sqlprint;
```

和

```
EXEC SQL WHENEVER not found sqlprint;
```

> **注意**
>
> 这不是EXEC SQL WHENEVER语句用法的一个详尽示例。更多用法示例可以在 SQL 手册中找到（例如 Groff 和 Weinberg 的The LAN TIMES Guide to SQL）。

连接到数据库服务器

使用下面的语句连接数据库：

```
EXEC SQL CONNECT TO dbname;
```

其中数据库名不带引号。在 2.1.0 版之前，数据库名必须放在单引号中。

也可以在 connect 语句中指定服务器名和端口名。语法是：

```
dbname[@server][:port]
```

或者

```
<tcp|unix>:postgresql://server[:port][/dbname][?options]
```

查询

一般来说，其他应用（如 psql）可接受的 SQL 查询都可以嵌入到你的 C 代码中。下面是一些示例。

创建表：

```
EXEC SQL CREATE TABLE foo (number int4, ascii char(16));
EXEC SQL CREATE UNIQUE index num1 on foo(number);
EXEC SQL COMMIT;
```

插入：

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

删除：

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

单行 SELECT：

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

使用游标的 SELECT：

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
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;
```

注意

完整的结构定义必须列在 declare 节内。

更多尚未实现的功能请看源码中的 `TODO` 文件。

pgaccess

pgaccess — 一个图形化的 PostgreSQL 客户端应用程序

大纲

`pgaccess` [*dbname*]

选项

dbname

要访问的一个现有数据库的名称。

描述

PgAccess 为 PostgreSQL 提供一个图形界面，你可以在其中管理你的表、编辑它们、定义查询、序列和函数。

PgAccess 可以：

- 在指定主机的指定端口上打开任何数据库，使用指定的用户名和口令。
- 执行 VACUUM。
- 把首选项保存在 `~/.pgaccessrc` 文件中。

对于表，PgAccess 可以：

- 打开多个表进行查看，显示的行数可配置。
- 通过拖动垂直网格线调整列宽。
- 在单元格中折行显示文本。
- 编辑时动态调整行高。
- 为每个表保存表布局。
- 导入/导出到外部文件（SDF、CSV）。
- 使用过滤功能；输入像 `price > 3.14` 这样的过滤器。
- 指定排序顺序；手动输入排序字段。
- 就地编辑；双击你想要更改的文本。
- 删除记录；指向该记录，按 **Delete** 键。
- 添加新记录；用右键点击保存新行。
- 用向导创建表。
- 重命名和删除（drop）表。
- 检索表的信息，包括拥有者、字段信息、索引。

对于查询，PgAccess 可以：

- 定义、编辑和存储用户定义的查询。
- 保存视图布局。
- 把查询存储为视图。
- 带可选的用户输入参数执行，例如，

```
select * from invoices where year=[parameter "Year of selection"]
```

- 查看任何 select 查询的结果。
- 运行动作查询（insert、update、delete）。
- 使用带拖放支持和表别名的可视化查询构建器构造查询。

对于序列，PgAccess 可以：

- 定义新实例。
- 检查现有实例。
- 删除。

对于视图，PgAccess 可以：

- 通过把查询保存为视图来定义它们。
- 查看它们，带过滤和排序功能。
- 设计新视图。
- 删除（drop）现有视图。

对于函数，PgAccess 可以：

- 定义。
- 检查。
- 删除。

对于报表，PgAccess 可以：

- 从表生成简单报表（beta 阶段）。
- 更改字段和标签的字体、大小和样式。
- 从数据库装载和保存报表。
- 预览表、Postscript 打印样例。

对于表单，PgAccess 可以：

- 打开用户定义的表单。
- 使用表单设计模块。
- 使用查询部件访问记录集。

对于脚本，PgAccess 可以：

- 定义。
- 修改。
- 调用用户定义的脚本。

注意

PgAccess 用 Tcl/Tk 编写。你的 PostgreSQL 安装需要带 Tcl 支持构建，PgAccess 才可用。

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 和 C++ 头文件的位置。

`--includedir-server`

打印用于服务器编程的 C 和 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 `--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。

另见

PostgreSQL 程序员指南

pg_dump

pg_dump — extract a PostgreSQL database into a script file or other archive file

大纲

pg_dump [[-a] | [-s]] [-b] [-c] [-C] [[-d] | [-D]] [-f *file*] [-F *format*] [-i] [[-n] | [-N]] [-o] [-O] [-R] [-S] [-t *table*] [-v] [-x] [-X *keyword*] [-Z *0...9*] [-h *host*] [-p *port*] [-U *username*] [-W] *dbname*

描述

pg_dump 是一个把 PostgreSQL 数据库保存为脚本或归档文件的工具。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。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 RDBMS 产品。此外，还有另外几种归档文件格式，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。

pg_dump 会保存重新生成所有用户定义类型、函数、表、索引、聚合和操作符所需的信息。此外，所有数据都以文本格式复制出来，因此既可以方便地重新复制进去，也可以导入编辑工具。

pg_dump 可用于转储数据库的内容，以便从一个 PostgreSQL 安装迁移到另一个安装。

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。

即使数据库正在被并发使用，pg_dump 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）。

选项

pg_dump 接受下列命令行参数。（长选项形式只在某些平台上可用。）

dbname

指定要转储的数据库名称。

-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，但它使归档更容易移植到其他 RDBMS 软件包。

-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，但如果你想要重新排列列顺序，它是必需的。

-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。由于 pg_dump 对系统目录了解甚多，任何给定版本的 pg_dump 都只打算与相应版本的数据库服务器一起使用。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
--no-quotes

除非绝对必要，抑制标识符周围的双引号。如果标识符中使用了保留字，
这可能导致装载这份转储数据时出问题。这是 pg_dump 6.4 之前版本的默认行为。

-N
--quotes

在标识符周围包含双引号。这是默认行为。

-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。通常，pg_dump 会发出（psql 特有的）\connect 语句来设置模式元素的所有权。另见 -R 和 -X use-set-session-authorization 下的说明。注意 -O 并不能阻止所有到数据库的重新连接，只能阻止那些专门用于调整所有权的重连。

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

禁止 pg_dump 输出在恢复期间需要重新连接数据库的脚本。通常的恢复脚本往往要以不同的用户身份重新连接若干次，以设置对象的原始所有权。这个选项是个相当生硬的工具，
因为它会使 pg_dump 丢失这些所有权信息，除非你使用 -X use-set-session-authorization 选项。

恢复期间不希望重新连接的一个可能原因是：访问数据库需要人工交互（例如密码）。

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。

-s
--schema-only

只转储模式（数据定义），不转储数据。

-S *username*
--superuser=*username*

pg_dump 创建的脚本或归档在某些情况下需要超级用户访问权限，例如禁用触发器或设置模式元素的所有权时。此选项指定这些情况下要使用的用户名。

-t *table*
--table=*table*

只转储 *table* 的数据。

-v
--verbose

指定详细模式。

-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

通常，如果 pg_dump 生成的（纯文本模式）脚本必须更改当前数据库用户（例如为了设置正确的对象所有权），它会使用psql的 \connect 命令。这条命令实际上会打开一个新的连接，这可能需要人工交互（例如密码）。如果使用 -X use-set-session-authorization 选项，pg_dump 将改为输出SET SESSION AUTHORIZATION命令。这效果相同，但它要求从生成的脚本恢复数据库的用户是数据库超级用户。此选项实际上会覆盖 -R 选项。

由于SET SESSION AUTHORIZATION是标准的 SQL 命令，而 \connect 只在psql中有效，此选项还能提高输出脚本理论上的可移植性。

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*

指定在支持压缩的归档格式中所用的压缩级别（目前只有自定义归档格式支持压缩）。

pg_dump 还接受下列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 host 以斜杠开头，则将其用作 Unix 域套接字的目录。

-p *port*
--port=*port*

指定服务器正在监听连接的 Internet TCP/IP 端口，或本地 Unix 域套接字文件扩展名。端口号默认为 5432，或 PGPORT 环境变量的值（如果已设置）。

-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。

诊断

```
                Connection to database 'template1' failed.
connectDBStart() -- connect() failed: No such file or directory
        Is the postmaster running locally
        and accepting connections on Unix socket '/tmp/.s.PGSQL.5432'?
```

pg_dump 无法连接到指定主机和端口上的 postmaster 进程。如果你看到这条消息，请确保 postmaster 运行在正确的主机上，并且你指定了正确的端口。

> **注意**
>
> 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

如果你的安装对 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。必须使用某种二进制归档格式对大对象进行完整转储。
- 在进行只转储数据的转储时，pg_dump 会在插入数据之前发出禁用用户表上触发器的查询，并在数据插入完成后发出重新启用它们的查询。如果恢复在中途中止，
  系统目录可能会处于错误状态。

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, PostgreSQL Administrator's Guide

pg_dumpall

pg_dumpall — 把所有 PostgreSQL 数据库抽取到一个脚本文件

大纲

`pg_dumpall` [[-c] | [--clean]] [[-g] | [--globals-only]] [-h *host*] [-p *port*] [-U *username*] [-W]

描述

pg_dumpall 是一个把集群中所有 PostgreSQL 数据库写出（“转储”）到一个脚本文件的实用工具。该脚本文件包含可以用作psql的输入来恢复数据库的 SQL 命令。它通过为集群中的每个数据库调用 pg_dump 来完成这一工作。pg_dumpall 还转储所有数据库共有的全局对象。（pg_dump 不保存这些对象。）这目前包括数据库用户和组的信息。

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。

选项

pg_dumpall 接受下列命令行参数：

-c, --clean

包含在重新创建数据库对象之前清理（删除）它们的 SQL 命令。（这个选项相当无用，因为输出脚本会自行创建数据库；创建出来总是空的。）

-g, --globals-only

只转储全局对象（用户和组），不转储数据库。

-h *host*

指定数据库服务器所在机器的主机名。如果 host 以斜线开头，它被用作 Unix 域套接字的目录。默认值取自 `PGHOST` 环境变量（如果设置了的话），否则尝试 Unix 域套接字连接。

-p *port*

服务器正在监听的端口号。默认为 `PGPORT` 环境变量（如果设置了的话）或编译时的默认值。

-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。

其他任何命令行参数都被传递给底层的 pg_dump 调用。这对于控制输出格式的某些方面很有用，但一些选项如 `-f`、`-t` 和 *`dbname`* 应当避免使用。

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

pg_dump、psql。可能的错误条件的详情请查阅它们。

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 [-a] [-c] [-C] [-d *dbname*] [-f *output-file*] [-F *format*] [-i *index*] [-l] [-L *contents-file*] [[-N] | [-o] | [-r]] [-O] [-P *function-name*] [-R] [-s] [-S] [-t *table*] [-T *trigger*] [-v] [-x] [-X *keyword*] [-h *host*] [-p *port*] [-U *username*] [-W] [*archive-file*]

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。它会发出重新生成所有用户定义类型、函数、表、索引、
聚集和操作符以及表中数据所需的命令。

归档文件包含供 pg_restore 重建数据库的信息，同时还允许 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。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。（长选项形式只在某些平台上可用。）

archive-name

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（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。（使用此开关时，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。大对象只能通过直接数据库连接来恢复。

-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。使用这种归档格式允许在恢复数据库时重排和/或排除模式元素，也可以限制恢复时重新装载哪些数据。

c

归档是 `pg_dump` 的自定义格式。这是最灵活的格式，因为它既允许重排数据装载也允许重排模式元素。这种格式默认还是压缩的。

-i *index*
--index=*index*

仅恢复指定名称 *index* 的定义。

-l
--list

列出归档的内容。此命令的输出可与 `-L` 选项一起使用，以限制并重排被恢复的项。

-L *list-file*
--use-list=*list-file*

仅恢复 *list-file* 中列出的元素，并按它们在文件中出现的顺序进行恢复。可以移动行，也可以通过在行首放置 ; 将其注释掉。

-N
--orig-order

按原始转储顺序恢复各项。默认情况下，`pg_dump` 会按对 `pg_dump` 方便的顺序转储各项，然后按修改过的 OID 顺序保存归档。此选项覆盖 OID 排序。

-o
--oid-order

按 OID 顺序恢复各项。默认情况下，`pg_dump` 会按对 `pg_dump` 方便的顺序转储各项，然后按修改过的 OID 顺序保存归档。此选项强制严格的 OID 排序。

-O
--no-owner

阻止任何恢复原始对象所有权的尝试。对象将属于用来连接数据库的用户名。

-P *function-name*
--function=*function-name*

指定要恢复的过程或函数。

-r
--rearrange

按修改过的 OID 顺序恢复各项。默认情况下，pg_dump 会按对 pg_dump 方便的顺序转储各项，然后按修改过的 OID 顺序保存归档。大多数对象会按 OID 顺序恢复，但有些东西（如规则和索引）会在过程结束时恢复，而不管它们的 OID 如何。此选项是默认值。

-R
--no-reconnect

在恢复归档时，pg_restore 通常不得不以不同的用户名多次重新连接数据库，以设置所创建对象的正确所有权。如果这不可取（例如，
因为每次重新连接都需要手工交互（口令）），此选项可阻止 pg_restore 发出任何重新连接请求。（在纯文本模式下、未连接数据库时的连接请求，是通过发出psql的 \connect 命令完成的。）不过，此选项是一件相当生硬的工具，因为它会使 pg_restore 丢失所有对象所有权信息，除非你使用 -X use-set-session-authorization 选项。

-s
--schema-only

恢复模式（定义），不恢复数据。序列值会被重置。

-S *username*
--superuser=*username*

指定在禁用触发器和/或设置模式元素所有权时要使用的超级用户用户名。默认情况下，pg_restore 会在当前用户是超级用户时使用当前用户名。

-t *table*
--table=*table*

只恢复 *table* 的模式/数据。

-T *trigger*
--trigger=*trigger*

仅恢复 *trigger* 的定义。

-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

通常，如果恢复归档需要更改当前数据库用户（例如设置正确的对象所有权），
就必须打开一个新的数据库连接，这可能需要手工交互（例如口令）。如果你使用 -X use-set-session-authorization 选项，pg_restore 将改用SET SESSION AUTHORIZATION命令。这具有相同的效果，但它要求恢复归档的用户是数据库超级用户。
此选项实际上覆盖了 -R 选项。

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 host 以斜杠开头，则它被用作 Unix 域套接字目录。

-p *port*
--port=*port*

指定服务器监听连接所使用的 Internet TCP/IP 端口或本地 Unix 域套接字文件扩展名。端口号默认为 5432，或 PGPORT 环境变量的值（如果已设置）。

-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。

诊断

```
               Connection to database 'template1' failed.
connectDBStart() -- connect() failed: No such file or directory
        Is the postmaster running locally
        and accepting connections on Unix socket '/tmp/.s.PGSQL.5432'?
```

pg_restore 无法连接到指定主机和端口上的 postmaster 进程。如果你看到这条消息，请确保服务器运行在正确的主机上，并且你指定了正确的端口。如果你的站点使用认证系统，请确保你已获得所需的认证凭据。

> **注意**
>
> 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时，pg_restore 会在插入数据前发出查询禁用用户表上的触发器，然后在数据插入后再发出查询重新启用它们。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 的局限性的细节请参见 pg_dump 文档。

示例

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
```

分号是注释定界符，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, PostgreSQL 管理员指南

psql

psql — PostgreSQL 的交互式终端

大纲

```
psql [ options ] [ dbname [ user ] ]
```

概要

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。

描述

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。psql 可以通过命令行选项（即分别为 -d、-h、-p 和 -U）把这些参数告知它。如果遇到一个不属于任何选项的参数，它将被解释为数据库名（如果数据库名已经给出，则解释为用户名）。并非所有这些选项都是必需的，
会应用默认值。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（如权限不足、postmaster 未在服务器上运行等）无法建立连接，psql 将返回错误并终止。

输入查询

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 =>. For example,

```
$ psql testdb
Welcome to psql, 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 查询。通常，当遇到表示查询结束的分号时，输入行被发送到后端。行尾并不会终止一条查询！因此为了清晰，查询可以分布在多行上。
如果查询被发送且没有错误，查询的结果会显示在屏幕上。

每当执行一条查询时，psql 还会轮询由 LISTEN and NOTIFY .

psql 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，必须用单引号把它括起来。要在这类参数中包含单引号，
可以在它前面加一个反斜线。单引号中的内容还会进一步做类 C 的替换，即 \n（换行）、\t（制表符）、*digits*, \0*digits*, and \0x*digits* (the character with the given decimal, octal, or hexadecimal code).

如果未加引号的参数以冒号（:）开头，它会被当作一个变量，该变量的值将被用作参数。

用反引号（`）括起来的参数被当作传递给 shell 的命令行。命令的输出（去掉末尾的换行符）作为参数值。上述转义序列在反引号中也适用。

有些命令接受一个 SQL 标识符（如表名）作为参数。这些参数遵循 SQL 关于双引号的语法规则：不带双引号的标识符被强制转换为小写。对于所有其他命令，双引号没有特殊含义，
会成为参数的一部分。

当遇到另一个未加引号的反斜线时，参数解析停止。这被当作新元命令的开始。特殊序列 \\（两个反斜线）标志参数的结束并继续解析 SQL 查询（如果有）。这样 SQL 和 psql 命令就可以在一行中自由混用。但无论如何，元命令的参数不能延续到行尾之后。

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；否则设置为非对齐。
保留这条命令是为了向后兼容。通用的解决办法见 \pset。

\cd [*directory*]

把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任何表的标题，或取消任何这样的标题。此命令等价于 \pset title *title*. (The name of this command derives from “caption”, as it was previously only used to set the caption in an HTML table.)

\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* [with oids] { from | to } *filename* | stdin | stdout [using delimiters '*characters*'] [with 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，但不是由后端读取或写入指定的文件（后者因此需要后端访问权限和特殊的用户权限，
并且受限于后端可访问的文件系统），而是由 psql 读取或写入该文件，并在后端和本地文件系统之间传送数据。

该命令的语法类似于 SQL COPY 命令（细节见其描述）。注意，因此对 \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

这个操作不如 SQL 的 COPY 命令高效，因为所有数据都必须通过客户端/服务器的 IP 或套接字连接。对于大量数据，另一种技术可能更可取。

注意

注意前端副本与后端副本对 stdin 和 stdout 的解释不同：在前端副本中它们总是指 psql 的输入和输出流。在后端副本中，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*relation*

Shows all columns of *relation* (which could be a table, view, index, or sequence), 它们的类型以及任何特殊属性（如 NOT NULL 或默认值，如果有）。如果该关系实际上是表，任何已定义的索引、主键、唯一约束和检查约束也会列出。如果该关系是视图，则还显示视图定义。

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 不带任何参数调用，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其操作的数据类型。如果指定了 *pattern*（正则表达式），则只显示匹配的聚合。

\dd [*object*]

显示对象（可以是一个正则表达式）的描述，如果没有给出参数则显示所有对象的描述。（对象涵盖聚合、函数、操作符、类型、关系（表、视图、索引、序列、大对象）、规则和触发器。）例如：

```
=> \dd version
                Object descriptions
  Name   |   What   |         Description
---------+----------+-----------------------------
 version | function | PostgreSQL version string
(1 row)
```

对象的描述可以用 COMMENT ON SQL 命令生成。

> **注意**
>
> PostgreSQL 把对象描述存储在 pg_description 系统表中。

\df [*pattern*]

列出可用的函数及其参数和返回类型。If *pattern* (a regular expression) is specified, only matching functions are shown. If the form \df+ is used, additional information about each function, including language and description, is shown.

\distvS [*pattern*]

这不是实际的命令名：字母 i、s、t、v、S 分别代表索引（index）、序列（sequence）、表（table）、视图（view）和系统表（system table）。可以按任意顺序指定其中任意个或全部，以获得它们的列表以及各自的属主。

如果指定了 *pattern*，它是一个正则表达式，把列表限制为名字匹配的对象。如果在命令名后面加上“+”，每个对象都会连同其相关联的描述一起列出（如果有的话）。

\dl

这是\lo_list的别名，用于显示大对象列表。

\do [*name*]

列出可用的操作符及其操作数和返回类型。如果指定了 *name*，则只显示具有该名称的操作符。

\dp [*pattern*]

这是 \z 的别名，因它更具助记性而保留（display permissions，显示权限）。

\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 ]`

列出所有已配置的用户或只列出匹配 pattern 的用户。

`\edit` (or `\e`) [*filename*]

如果指定了 filename，则编辑该文件；编辑器退出后，其内容被复制回查询缓冲区。If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

然后新的查询缓冲区会按 psql 的正常规则重新解析，整个缓冲区被当作单独一行。（因此你不能用这种方式制作脚本。请用 `\i`。）这也意味着如果查询以分号结尾（或更确切地说包含分号），它会被立即执行。在其他情况下它只会留在查询缓冲区中等待。

> **提示**
>
> psql 依次搜索环境变量 `PSQL_EDITOR`、`EDITOR` 和 `VISUAL`（按此顺序）来寻找要用的编辑器。如果它们全部未设置，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并后跟一个换行符。
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
```

If the first argument is an unquoted `-n` the the trailing newline is not written.

> **提示**
>
> 如果你用 `\o` 命令重定向查询输出，你可能希望用 `\qecho` 而不是这条命令。

`\encoding` [*encoding*]

如果使用多字节编码，则设置客户端编码。不带参数时，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。默认是竖线（|）。设置输出选项的通用方法另见 \pset。

`\g` [{ *filename* | | *command* }]

把当前查询输入缓冲区发送到后端，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，psql 将列出所有可获得语法帮助的命令。如果 command 是星号（*），则显示所有 SQL 命令的语法帮助。

> **注意**
>
> 为简化输入，由多个词组成的命令不必加引号。因此键入 **\help alter table** 就可以。

\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 读取输入并执行，就像从键盘输入一样。

> **注意**
>
> 如果想在读入时在屏幕上看到这些行，必须把变量 ECHO 设置为 all。

\l (or \list)

列出服务器中的所有数据库及其所有者。向命令名追加 + 还可以看到数据库的任何描述。If your PostgreSQL installation was compiled with multibyte encoding support, the encoding scheme of each database is shown as well.

\lo_export*loidfilename*

从数据库读取 OID 为 loid 的大对象并把它写入 filename。注意这条命令与服务器端函数 lo_export 有微妙的差别，后者以数据库服务器运行用户的权限在服务器的文件系统上操作。

> **提示**
>
> 用 \lo_list 查出大对象的 OID。

> **注意**
>
> 关于所有大对象操作的重要信息，见 LO_TRANSACTION 变量的描述。

\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。Example:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

The response indicates that the large object received object id 152801 which one ought to remember if one wants to access the object ever again. For that reason it

is recommended to always associate a human-readable comment with every object. Those can then be seen with the `\lo_list` command.

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。

> **注意**
>
> 关于所有大对象操作的重要信息，见 `LO_TRANSACTION` 变量的描述。

`\lo_list`

显示当前存储在数据库中的所有 PostgreSQL 大对象的列表以及为它们提供的任何注释。

`\lo_unlink`*`loid`*

从数据库删除 OID 为 loid 的大对象。

> **提示**
>
> 用 `\lo_list` 查出大对象的 OID。

> **注意**
>
> 关于所有大对象操作的重要信息，见 `LO_TRANSACTION` 变量的描述。

`\o` [{*`filename`* | `|` *`command`*}]

把以后的查询结果保存到文件 filename 中，或把以后的结果通过管道送到一个单独的 Unix shell 去执行 command。If no arguments are specified, the query output will be reset to `stdout`.

“查询结果”包括从数据库服务器获得的所有表、命令响应和通知，
以及查询数据库的各反斜杠命令（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，请用 `\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”把一个元组的所有字段写在一行上，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。启用扩展格式时，所有输出有两列，字段名在左、数据在右。如果数据在正常的水平模式下放不到屏幕上，此模式很有用。

所有四种输出模式都支持扩展模式。

`null`

第二个参数是一个每当字段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制表符或逗号分隔的输出，其他程序可能更喜欢这样。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 `'|'`（竖线符号）。

`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的信息，如列标题、标题和各种脚注。在只显示元组模式下，只显示实际的表数据。

`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。

> **注意**
>
> 这以前只影响 HTML 模式。现在你可以在任何输出格式中设置标题。

`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`

切换分页器的使用以输出表格。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 `more`。

无论如何，psql 只在看起来合适的时候才使用分页器。这尤其意味着输出是到终端，而且表格通常无法在屏幕上完整显示。由于打印例程的模块化特性，
并不总是能预测实际会打印的行数。因此 psql 在何时使用分页器上可能显得不太挑剔。

Illustrations on how these different formats look can be seen in the 示例 section.

> **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。This option is only available if psql is configured to use the GNU history library.

> **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保存。每次 psql 启动时历史也会自动加载。

\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。详见有关 psql 变量的小节。

尽管欢迎你把任何变量设置为任何值，psql 把若干变量视为特殊的。它们记录在有关变量的小节中。

> **注意**
>
> 此命令与 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 标签中的选项。This command is equivalent to \pset tableattr *table_options*.

\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

切换扩展行格式模式。因此它等价于 \pset expanded。

\z [*pattern*]

产生数据库中所有表及其访问权限的列表。如果给出参数，它被当作一个正则表达式，只列出匹配它的表。

```
test=> \z
Access permissions for database "test"
 Relation |           Access permissions
----------+-------------------------------------
 my_table | {"=r","joe=arwR", "group staff=ar"}
(1 row )
```

这样阅读：

- "=r"：PUBLIC 拥有对该表的读（SELECT）权限。
- "joe=arwR"：用户 joe 拥有读、写（UPDATE、DELETE）、“追加”（INSERT）权限，以及在该表上创建规则的权限。
- "group staff=ar"：组 staff 拥有 SELECT 和 INSERT 权限。

GRANT和REVOKE命令用于设置访问权限。

\! [*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。

\?

获取有关反斜线（\）命令的帮助信息。

命令行选项

如果做了相应配置，psql 既理解标准的 Unix 短选项，也理解 GNU 风格的长选项。后者并非在所有系统上都可用。

-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。This is equivalent to setting the variable ECHO to 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 *query*

指定 psql 执行一个查询字符串 query，然后退出。这在 shell 脚本中很有用。

query 必须是后端完全可解析的查询字符串（即不含 psql 特有的功能），或者是单条反斜杠命令。因此你不能混合 SQL 和 psql 元命令。要做到这一点，可以把字符串管道输入 psql，像这样：echo "\x \\ select * from foo;" | psql。

-d, --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

显示发送到后端的所有查询。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。这等价于在 psql 中设置变量 ECHO_HIDDEN。

-f, --file *filename*

用文件 *filename* 作为查询的来源，而不是交互式地读取查询。文件处理完后，psql 终止。这在很多方面等价于内部命令 \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, --field-separator *separator*

使用*separator*作为字段分隔符。这等效于`\pset fieldsep`或者`\f`。

-h, --host *hostname*

指定 postmaster 所在机器的主机名。如果 host 以斜线开头，它被用作 Unix 域套接字所在的目录。

-H, --html

打开 HTML 表格输出。这等效于 \pset format html 或 \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`。

-o, --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*

指定 postmaster 用于监听连接的 TCP/IP 端口，或者在省略时指定本地 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, --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

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。Within psql you can also set the `QUIET` variable to achieve the same effect.

-R, --record-separator *separator*

把*separator*用作记录分隔符。这等效于`\pset recordsep`命令。

-s, --single-step

以单步模式运行。即每条查询发送到后端之前都会提示用户，并可选择取消执行。
可用它来调试脚本。

-S, --single-line

以单行模式运行，此时换行符像分号一样终止一条查询。

> **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-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。

这个选项已废弃，因为它在概念上有缺陷。
（为非默认用户名而提示与因为后端要求而为口令提示其实是两件事。）建议改看 `-U` 和 `-W` 选项。

-U, --username *username*

以用户 username 而不是默认用户连接到数据库。（当然，你必须有权限这样做。）

-v, --variable, --set *assignment*

执行变量赋值，类似于 \set 内部命令。注意在命令行上必须用等号分隔名称和值。要取消变量的设置，去掉等号即可。要把一个变量设置为没有值，使用等号但去掉值。
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`.

在当前版本中，只要后端要求口令认证，psql 就会自动发出口令提示。由于这目前基于一个技巧，自动识别可能神秘地失败，因此提供这个选项来强制提示。
如果没有发出口令提示而后端又要求口令认证，连接尝试将失败。

-x, --expanded

打开扩展行格式模式。这等效于命令 `\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。

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。此特性还比较新且不太成熟，但将来有扩展它的计划。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” 设置为值 “bar” 。要获取变量的内容，
在名称前加冒号并用作任意斜线命令的参数：

```
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'`，分别得到 Perl 著名的 “软链接” 或 PHP 著名的 “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。但另一方面，`\set bar :foo` 是一种完全合法的复制变量的方法。

如果调用 \set 时不带第二个参数，该变量只是被设置但没有值。要取消设置（或删除）一个变量，使用命令 \unset。

psql 的内部变量名可以由字母、数字和下划线以任意顺序和任意数量组成。A number of regular variables are treated specially by psql. They indicate certain option settings that can be changed at runtime by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid such variables. A list of all specially treated variables follows.

`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。To specify this on program start-up, use the switch `-a`. If set to “`queries`”, psql merely prints all queries as they are sent to the backend. The option 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。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后端执行。

`ENCODING`

当前的客户端多字节编码。如果你没有设置为使用多字节字符，此变量将始终包含 SQL_ASCII。

`HISTCONTROL`

如果此变量设置为 `ignorespace`，以空格开头的行不会进入历史列表。如果设置为值 `ignoredups`，与上一条历史行相同的行不会进入。`ignoreboth` 值组合这两个选项。如果未设置，或设置为上述值以外的其他值，交互模式下读入的所有行都会保存到历史列表中。

> **注意**
>
> 此特性是从 bash 厚颜无耻地抄袭来的。

HISTSIZE

在命令历史中存储的命令数。默认值为 500。

> **注意**
>
> 这个功能无耻地剽窃自 bash。

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 EOF 字符（通常是 Control-D）将终止应用程序。如果设置为数值，则在应用程序终止之前忽略那么多个 EOF 字符。如果该变量已设置但没有数值，默认为 10。

> **注意**
>
> 这个功能无耻地剽窃自 bash。

LASTOID

最后受影响的 oid 的值，由 INSERT 或 lo_insert 命令返回。此变量只在下一条 SQL 命令的结果显示之前保证有效。

LO_TRANSACTION

如果你使用 PostgreSQL 大对象接口来专门存储放不进单个元组的数据，所有操作都必须包含在一个事务块中。（更多信息见大对象接口的文档。）由于 psql 无法在你调用其内部命令（\lo_export、\lo_import、\lo_unlink）
之一时判断你是否已有一个正在进行的事务，它必须采取某种任意操作。
该操作可以是回滚可能已在进行的任何事务，或提交任何这样的事务，或什么都不做。
在最后一种情况下，你必须提供自己的 BEGIN TRANSACTION/COMMIT 块，否则结果将是不可预测的（通常导致期望的操作无论如何都不会执行）。

要选择你想做的操作，把此变量设置为 “rollback” 、 “commit” 或 “nothing” 之一。
默认是回滚事务。如果只想加载一个或少数几个对象，这样即可。然而，
如果打算传输许多大对象，建议在所有命令周围提供一个显式的事务块。

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 查询或内部元命令），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

它们指定 psql 发出的提示符应有的样子。见 “提示符” 下文。

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

此变量由命令行选项 -S 设置。你可以在运行时取消设置或重置它。

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。

SQL Interpolation

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 SQL 语句中。The syntax for this is again to prepend the variable name with a colon (:).

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would then query the table my_table. The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

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字段中。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);
```

One possible problem with this approach is that my_file.txt might contain single quotes. These need to be escaped so that they don't cause a syntax error when the third line is processed. This could be done with the program sed:

```
testdb=> \set content '\'' `sed -e "s/'/\\\\\\'/g" < my_file.txt` '\''
```

Observe the correct number of backslashes (6)! You can resolve it this way: After psql has parsed this line, it passes sed -e "s/'/\\\'/g" < my_file.txt to the shell. The shell will do its own thing inside the double quotes and execute sed with the arguments -e and

`s/'/\\'/g`. When sed parses this it will replace the two backslashes with a single one and then do the substitution. Perhaps at one point you thought it was great that all Unix commands use the same escape character. And this is ignoring the fact that you might have to escape all backslashes as well because SQL text constants are also subject to certain interpretations. In that case you might be better off preparing the file externally.

由于冒号可以合法地出现在查询中，适用以下规则：如果该变量未设置，字符序列“冒号+名称”不会被更改。在任何情况下，你都可以用反斜线转义冒号来保护它不被解释。
（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 扩展，因此存在冲突。）

提示符

psql 发出的提示符可以按你的偏好定制。三个变量 `PROMPT1`、`PROMPT2` 和 `PROMPT3` 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 请求新查询时发出的常规提示符。提示符 2 在查询输入期间期待更多输入时发出，因为查询未以分号结束或引号未闭合。提示符 3 在你运行 SQL COPY 命令并需要在终端上输入元组时发出。

相应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`

The full hostname (with domain name) of the database server, or `[local]` if the connection is over a Unix domain socket, or `[local:`*`/dir/name`*`]`, if the Unix domain socket is not at the compiled in default location.

`%m`

数据库服务器的主机名（在第一个点后截断）；如果连接通过 Unix 域套接字则为 [local]。

`%>`

数据库服务器正在监听的端口号。

`%n`

你连接时所用的用户名（不是本地系统用户名）。

`%/`

当前数据库的名称。

`%~`

类似于 %/，但如果该数据库是你的默认数据库则输出 ~（波浪号）。

`%#`

如果当前用户是数据库超级用户则为 #，否则为 >。

`%R`

在提示符 1 中通常是“=”，单行模式下是“^”，会话与数据库断开时（例如 `\connect` 失败时）是“!”。在提示符 2 中该序列被替换为“-”、“*”、单引号或双引号，具体取决于 psql 期待更多输入是因为查询尚未结束、你在 `/* ... */` 注释内，还是你在带引号的字符串内而期待更多输入。在提示符 3 中该序列不解析为任何内容。

`%digits`

如果 *digits* 以 `0x` 开头，其余字符按十六进制数解释，并替换为相应代码的字符。如果第一位数字是 0，这些字符按八进制数解释，并替换为相应的字符。否则按十进制数假设。

`%:name:`

psql 变量 *name* 的值。详见“变量”一节。

`%`command`

command 的输出，类似于普通的反引号替换。

To insert a percent sign into your prompt, write `%%`. The default prompts are equivalent to `'%/%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。

Miscellaneous

psql 正常结束时向 shell 返回 0；发生自身的致命错误（内存不足、找不到文件）时返回 1；与后端的连接变坏且会话不是交互式时返回 2；脚本中发生错误且设置了变量 ON_ERROR_STOP 时返回 3。

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）。

GNU readline

psql 支持 readline 和 history 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。可用时，psql 会自动构建为使用这些特性。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 而是 readline 的特性。详细内容请阅读其文档。）

如果你安装了 readline 库但 psql 似乎没有使用它，必须确保 PostgreSQL 的顶级 configure 脚本能找到它。`configure` 需要同时找到库 `libreadline.a`（或等价的共享库）以及头文件 `readline.h` 和 `history.h`（或 `readline/readline.h` 和 `readline/history.h`）。如果你把库和头文件安装在不常见的地方，必须把位置告诉 `configure`，例如：

```
$ ./configure --with-includes=/opt/gnu/include --with-libs=/opt/gnu/
lib  ...
```

Then you have to recompile psql (not necessarily the entire code tree).

GNU readline 库可以从 GNU 项目的 FTP 服务器 ftp://ftp.gnu.org 获取。

示例

> **注意**
>
> This section only shows a few examples specific to psql. 如果你想学习 SQL 或熟悉 PostgreSQL，可能希望阅读发行版中包含的教程。

第一个例子显示如何把一条查询分布在多行输入上。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
```

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    |
```

这时你决定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
```

或者，使用简短命令：

```
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
```

附录

缺陷与问题

- 在早期的 psql 中，允许第一个参数直接紧跟在（单字母）命令之后。为了兼容性这一点在一定程度上仍被支持，但我不打算在这里解释细节，因为不鼓励这种用法。
  但如果你收到奇怪的消息，请记住这一点。例如

```
testdb=> \foo
Field separator is "oo",
```

这可能不是人们期望的结果。

- psql 只与相同版本的服务器顺畅配合。这并不意味着其他组合会立即失败，但可能出现或明显或微妙的问题。
- Pressing Control-C during a “copy in” (data sent to the server) doesn't show the most ideal of behaviors. 如果你收到类似 “COPY 状态必须先终止” 的消息，只需输入 \c - - 重置连接。

pgtclsh

pgtclsh — PostgreSQL Tcl shell 客户端

大纲

`pgtclsh` [*`filename`* [*`arguments`*...]]

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 , PostgreSQL 程序员指南（`libpgtcl` 的说明）, tclsh

pgtksh

pgtksh — PostgreSQL Tcl/Tk shell 客户端

大纲

`pgtksh` [*`filename`* [*`arguments`*...]]

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 , PostgreSQL 程序员指南（`libpgtcl` 的说明）, tclsh , wish

vacuumdb

vacuumdb — 对一个 PostgreSQL 数据库进行垃圾收集和分析

大纲

`vacuumdb` [*connection-options*...] [[-d] *dbname*] [[--full] | [-f]] [[--verbose] | [-v]] [[--analyze] | [-z]] [--table '*table* [(*column* [,...])]']
`vacuumdb` [*connection-options*...] [[--all] | [-a]] [[--full] | [-f]] [[--verbose] | [-v]] [[--analyze] | [-z]]

Inputs

vacuumdb 接受下列命令行参数：

-d *dbname*
--dbname *dbname*

指定要清理或分析的数据库名。

-a
--all

清理所有数据库。

-f
--full

执行“完全”清理。

-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。

-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**
>
> 如果你指定了要清理（vacuum）的列，可能需要向 shell 转义括号。

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 host 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 Internet TCP/IP 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。

-e
--echo

回显 vacuumdb 生成并发送给服务器的命令。

-q
--quiet

不显示响应。

Outputs

`VACUUM`

一切正常。

`vacuumdb: Vacuum failed.`

出了问题。vacuumdb 只是一个包装脚本。关于错误消息和潜在问题的详细讨论，参见 VACUUM 和psql。

Description

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 是一个外壳脚本包装，它通过 PostgreSQL 的交互式终端psql调用后端命令 VACUUM。通过这种方式清理数据库与其他方法没有实质差别。脚本必须能找到 psql，而且目标主机上必须有数据库服务器在运行。此外，适用于 psql 和 libpq 前端库的任何默认设置和环境变量也同样有效。

Usage

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
```

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 556
initlocation 558
ipcclean 559
pg_ctl 560
pg_passwd 563
postgres 564
postmaster 567

initdb

initdb — 创建一个新的 PostgreSQL 数据库集簇

大纲

initdb [--pgdata] | [-D]*directory* [[--username] | [-U]*username*] [[--pwprompt] | [-W]] [[--encoding] | [-E]*encoding*] [-L *directory*] [[--noclean] | [-n]] [[--debug] | [-d]]

描述

initdb创建一个新的 PostgreSQL数据库集簇（或数据库系统）。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。

创建数据库系统包括：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

--pgdata=*directory*
-D *directory*

这个选项指定数据库系统的存放目录。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。

--username=*username*
-U *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”。

--pwprompt
-W

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。

--encoding=*encoding*
-E *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。要使用编码特性，必须在构建时启用它，
同时你还要选择这个选项的默认值。

还有其他一些较少使用的参数：

-L *directory*

指定initdb到哪里寻找它初始化数据库系统所需的输入文件。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。

--noclean
-n

默认情况下，当 initdb判定某个错误使它无法完全创建数据库系统时，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。

--debug
-d

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。

环境

PGDATA

指定数据库系统的存放目录；可以用-D选项覆盖。

另见

postgres, postmaster, PostgreSQL Administrator's Guide

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。

用法

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）。如果在一个 postmaster 运行时执行这个命令，该 postmaster 分配的共享内存和信号量将被删除。这将导致该 postmaster 启动的后端服务器全面失效。

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_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postmaster，即 PostgreSQL 后端服务器，或者显示一个运行中 postmaster 的状态的工具。虽然 postmaster 可以手动启动，但 pg_ctl 封装了诸如重定向日志输出、恰当地从终端和进程组脱离这样的任务，并且它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 postmaster。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 postmaster 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，然后数据库被关闭。“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 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` 模式检查是否有一个 postmaster 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 的选项。

这些参数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 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 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。

-W

不等待启动或关闭完成。这是启动和重启的默认方式。

文件

如果文件 `postmaster.opts.default` 存在于数据目录中，该文件的内容将被作为选项传递给 postmaster，除非被 `-o` 选项覆盖。

示例

启动 postmaster

启动一个 postmaster：

```
$pg_ctl start
```

一个启动 postmaster 并阻塞到 postmaster 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 postmaster，使用：

```
$pg_ctl -o "-F -p 5433" start
```

停止 postmaster

```
$pg_ctl stop
```

停止 postmaster。使用 `-m` 开关可以控制后端如何关闭。

重启 postmaster

这几乎等价于停止 postmaster 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。以最简单的形式重启 postmaster：

```
$pg_ctl restart
```

重启 postmaster 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 postmaster 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。

缺陷

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，它可能会失败。应当避免等待。

另见

postmaster、PostgreSQL 管理员指南

pg_passwd

pg_passwd — 更改一个辅助PostgreSQL口令文件

大纲

```
pg_passwd filename
```

描述

pg_passwd是一个操纵平面文本口令文件的工具。
这些文件可以控制PostgreSQL服务器的客户端认证。
有关建立这种认证机制的更多信息可以在管理员指南中找到。

文本口令文件的格式是每行一个条目；每个条目的各个字段用冒号分隔。第一个字段是用户名，第二个字段是加密后的口令。
其他字段被忽略（以便让使用类似格式的应用之间共享口令文件）。pg_
passwd让用户能够向这样的文件中交互式地添加条目、更改已有条目的口令并加密这些口令。

把口令文件的名字作为pg_passwd命令的参数提供。要被 PostgreSQL 使用，该文件必须位于服务器的数据目录中，并且该文件的基本名必须在pg_hba.conf 访问控制文件中指定。

```
$pg_passwd /usr/local/pgsql/data/passwordsFile "/usr/local/pgsql/data/
passwords" does not exist.  Create? (y/n):yUsername:guestPassword:Re-
enter password:
```

其中Password:和Re-enter password:提示要求输入相同的口令，且输入不会显示在终端上。注意，由于标准 crypt(3) 库例程的限制，口令只有前八个字符有效。

原口令文件会被改名为passwords.bk。

要使用这个口令文件，在pg_hba.conf中放入如下一行:

```
host  mydb     133.65.96.250   255.255.255.255 password passwords
```

这一行允许主机 133.65.96.250 使用passwords文件中列出的口令访问数据库 mydb（并且只允许该文件中列出的用户访问）。

> **注意**
>
> 口令文件中存在口令字段为空的条目也很有用。（这与空口令不同。）
> 这样的条目让你能够限制可以访问系统的用户。这些条目无法由 pg_passwd管理，但你可以手工编辑口令文件。

另见

PostgreSQL 管理员指南

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*] [-D *datadir*] [-e] [-E] [-f [s] | [i] | [t] | [n] | [m] | [h]] [-F] [-i] [-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] [-i] [-o *filename*] [-O] [-p *database*] [-P] [[-s] | [-t [pa] | [pl] | [ex]]] [-S *sort-mem*] [-v *protocol-version*] [-W *seconds*] [--*name*=*value*]

描述

postgres 可执行文件就是真正处理查询的 PostgreSQL 服务器进程。它通常不被直接调用；而是启动一个postmaster多用户服务器。

上面的第二种形式是postmaster调用 postgres 的方式（只是概念上如此，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。第一种形式直接以交互式单用户模式启动服务器。这种模式的主要用途是在initdb引导期间。有时它也用于调试或灾难恢复。

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，但其形式对开发者比对最终用户更有用。但要注意，运行单用户后端并不真正适合调试服务器，
因为不会发生真实的进程间通信和锁定。

当运行独立后端时，会话用户将被设为 id 为 1 的用户。这个用户并不需要真的存在，因此独立后端可用于从某些意外损坏系统目录的情况中手动恢复。在独立模式下，id 为 1 的用户被隐式授予超级用户权限。

选项

当 postgres 由postmaster启动时，它会继承后者设置的所有选项。此外，postmaster 可以通过 -o 开关向它传递 postgres 特有的选项。

通过建立配置文件，你可以免于输入这些选项。详情见管理员指南。某些（安全的）
选项也可以由连接中的客户端以依赖于应用的方式设置。例如，如果设置了环境变量 PGOPTIONS，基于 libpq 的客户端会把该字符串传给服务器，服务器会把它解释为 postgres 的命令行选项。

通用选项

选项 -A、-B、-c、-d、-D、-F 和 --name 的含义与postmaster中相同。

-e

把默认日期风格设为“欧洲式”，即用“日在月前”（而不是月在日前）
规则解释有歧义的日期输入，并且在某些日期输出格式中日在月之前打印。更多信息见 PostgreSQL 用户指南。

-o *filename*

把所有调试和错误输出发送到 *filename*。如果后端运行在 postmaster 之下，此选项会被忽略，转而使用从 postmaster 继承的 stderr。

-P

在扫描/更新系统元组时忽略系统索引。对系统表/索引执行 REINDEX 命令要求使用此选项。

-s

在每个查询结束时打印时间信息和其他统计信息。这对基准测试或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 512 千字节。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

回显所有查询。

-N

禁用换行符作为查询分隔符。

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。

-i

阻止查询执行，但显示计划树。

-O

允许修改系统表的结构。这由 initdb 使用。

-p *database*

表明此服务器是由 postmaster 启动的，并对缓冲池管理、文件描述符等做出不同的假定。

-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*

一遇到此选项，进程就会休眠指定的秒数。
这给开发者留出把调试器附加到后端进程的时间。

用法

用类似下面的命令启动一个独立后端：

postgres -D $PGDATA *other-options* my_database

用 -D 提供数据库区域的正确路径，或者确保已设置环境变量 PGDATA。还要指定你想使用的具体数据库的名称。

通常，独立后端把换行符当作命令输入的终结符；它不像 psql 那样对分号有智能处理。要让一个命令跨多行继续，你必须在除最后一行之外的每个换行符之前输入反斜杠。

但如果使用 -N 命令行开关，换行符就不会终结命令输入。后端会读取标准输入直到文件结束（EOF）标记，然后把输入当作单个查询字符串处理。这种情况下反斜杠-
换行不会受到特殊对待。

要退出会话，输入 EOF（通常是 **Control+D**）。如果使用了 -N，则需要连续输入两次 EOF 才能退出。

注意，独立后端不提供复杂的行编辑特性（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 的多用户数据库服务器。客户端应用要访问数据库，需要（通过网络或本地）连接到一个运行中的 postmaster。postmaster 随后启动一个单独的服务器进程（“postgres”）来处理该连接。postmaster 还管理服务器进程之间的通信。

默认情况下 postmaster 在前台启动并把日志消息打印到标准输出。在实际应用中，postmaster 应作为后台进程启动，也许在系统引导时启动。

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管理员指南。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。列表和描述请查阅管理员指南。
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。该值设置得越高，写入服务器日志的调试输出就越多。默认为 0，表示不做调试。最高到 4 的值有用；更高的数字不会产生额外的输出。

-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。使用前请阅读详细的文档！

-h *hostname*

指定 postmaster 用来监听客户端应用连接的 TCP/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。

-k *directory*

指定 postmaster 用来监听客户端应用连接的 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管理员指南。）

-o *extra-options*

指定的命令行风格的选项（*extra-options*）会传递给该 postmaster 启动的所有后端服务器进程。可能的选项见postgres。如果选项字符串包含空格，整个字符串必须加引号。

-p *port*

指定 postmaster 用来监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。默认为 `PGPORT` 环境变量的值，如果未设置 `PGPORT`，则默认为编译时确定的值（通常为 5432）。如果你指定的不是默认端口，所有客户端应用都必须用命令行选项或 `PGPORT` 指定同一端口。

-S

指定 postmaster 进程应以静默模式启动。也就是说，它将与用户的（控制）终端脱离、建立自己的进程组，并把标准输出和标准错误重定向到 `/dev/null`。

使用这个开关会丢弃所有日志输出，这很可能不是你想要的，
因为它会使问题的排查变得非常困难。在后台启动 postmaster 的更好方法见下文。

--*name*=*value*

设置一个命名的运行时参数；这是`-c`的较短形式。

还有两个额外的命令行选项可用于调试导致后端异常死亡的问题。这些选项控制 postmaster 在这种情况下的行为，两个选项都不用于日常操作。

这种情况下的常规策略是通知所有其他后端必须终止，然后重新初始化共享内存和信号量。
这是因为出错的后端在终止之前可能已经破坏了某些共享状态。

这些特殊情况的选项是：

-n

postmaster 不会重新初始化共享数据结构。有经验的系统程序员随后可以用调试器检查共享内存和信号量状态。

-s

postmaster 会通过发送信号 SIGSTOP 停止所有其他后端进程，但不会使它们终止。这让系统程序员可以手工收集所有后端进程的 core 转储。

输出

semget: No space left on device

如果你看到这条消息，应当运行 ipcclean 命令。之后再次尝试启动 postmaster。如果仍然不行，你可能需要按安装说明中的描述为共享内存和信号量配置你的内核。
如果你在单个主机上运行多个 postmaster 实例，或者内核的共享内存和/或信号量限制特别小，就可能需要重新配置内核以增大其共享内存或信号量参数。

> **提示**
>
> 通过减小 -B 来降低 PostgreSQL 的共享内存消耗，和/或减小 -N 来降低信号量消耗，你可能可以推迟重新配置内核。

StreamServerPort: cannot bind to port

如果你看到这条消息，应当确保没有其他 postmaster 进程已在同一端口号上运行。判断这一点的最简单方法是使用命令

```
$ps ax | grep postmaster
```

或

```
$ps -e | grep postmaster
```

（取决于你的系统）。

如果你确信没有其他 postmaster 进程在运行却仍得到这个错误，请尝试用 -p 选项指定一个不同的端口。如果你终止 postmaster 后立即用同一端口重启它，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即可。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 把 1024 以下的端口号视为受信任的，只允许 Unix 超级用户访问它们。

注解

如果有可能，请不要用 SIGKILL 杀死 postmaster。这会阻止 postmaster 在终止前释放它持有的系统资源（例如共享内存和信号量）。

要正常终止 postmaster，可以使用信号 SIGTERM、SIGINT 或 SIGQUIT。第一种会在退出前等待所有客户端终止，第二种会强制断开所有客户端的连接，
第三种会立即退出而不做正常的关闭，导致重启期间进行一次恢复。

实用工具命令pg_ctl可以用来安全而方便地启动和关闭 postmaster。

-- 选项在 FreeBSD 或 OpenBSD 上不起作用。请改用 -c。这是受影响的操作系统中的一个 bug；如果不修复，PostgreSQL 的未来版本将提供一种变通方法。

用法

要在后台使用默认值启动 postmaster，键入：

```
$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 中可能为 sort_mem 存在的设置。注意，在命令行上，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。

PostgreSQL 7.2.8 开发人员指南

PostgreSQL 全球开发组

PostgreSQL 7.2.8 开发人员指南

PostgreSQL 全球开发组

摘要

本文档包含对 PostgreSQL 开发人员可能有用的各种信息。

法律声明

目录

1. PostgreSQL 源代码 1
    1.1. 格式 1
2. PostgreSQL 内部概述 2
    2.1. 查询的路径 2
    2.2. 连接是如何建立的 2
    2.3. 解析器阶段 3
        2.3.1. 解析器 3
        2.3.2. 转换过程 4
    2.4. PostgreSQL 规则系统 4
        2.4.1. 重写系统 5
    2.5. 规划器/优化器 6
        2.5.1. 生成可能的计划 6
        2.5.2. 计划的数据结构 6
    2.6. 执行器 7
3. 系统目录 8
    3.1. 概述 8
    3.2. pg_aggregate 9
    3.3. pg_attrdef 9
    3.4. pg_attribute 9
    3.5. pg_class 11
    3.6. pg_database 12
    3.7. pg_description 13
    3.8. pg_group 14
    3.9. pg_index 14
    3.10. pg_inherits 15
    3.11. pg_language 15
    3.12. pg_largeobject 16
    3.13. pg_listener 16
    3.14. pg_operator 17
    3.15. pg_proc 17
    3.16. pg_relcheck 18
    3.17. pg_rewrite 19
    3.18. pg_shadow 19
    3.19. pg_statistic 20
    3.20. pg_trigger 21
    3.21. pg_type 22
4. 前端/后端协议 25
    4.1. Overview 25
    4.2. Protocol 25
        4.2.1. 启动 25
        4.2.2. Query 27
        4.2.3. 函数调用 28
        4.2.4. 通知响应 29
        4.2.5. 取消进行中的请求 29
        4.2.6. Termination 29
        4.2.7. SSL 会话加密 30
    4.3. 消息数据类型 30
    4.4. 消息格式 31
5. gcc 默认优化 38
6. BKI 后端接口 39
    6.1. BKI 文件格式 39

6.2. BKI 命令 39
6.3. 示例 40
7. 页文件 41
8. 遗传查询优化 42
8.1. 将查询处理看成是一个复杂的优化问题 42
8.2. 遗传算法 42
8.3. PostgreSQL 中的遗传查询优化（GEQO） 43
8.3.1. PostgreSQL GEQO 的未来实现任务 44
8.4. 进一步阅读 44
9. 本地语言支持 45
9.1. 给翻译者 45
9.1.1. 要求 45
9.1.2. 概念 45
9.1.3. 创建和维护消息目录 46
9.1.4. 编辑 PO 文件 46
9.2. 给程序员 47
A. CVS 代码库 50
A.1. 通过匿名 CVS 获取源码 50
A.2. CVS 树的组织 51
A.3. 通过 CVSup 获取源码 53
A.3.1. 准备 CVSup 客户端系统 53
A.3.2. 运行 CVSup 客户端 53
A.3.3. 安装 CVSup 55
A.3.4. 从源码安装 56
B. 文档 58
B.1. DocBook 58
B.2. Toolsets 58
B.2.1. Linux RPM 安装 59
B.2.2. FreeBSD 安装 59
B.2.3. Debian 软件包 60
B.2.4. 从源码手动安装 60
B.3. 构建文档 62
B.3.1. HTML 63
B.3.2. 手册页 63
B.3.3. 硬拷贝生成 63
B.3.4. 纯文本文件 65
B.4. 文档编写 65
B.4.1. Emacs/PSGML 65
B.4.2. 其他 Emacs 模式 67

插图清单

8.1. 遗传算法的结构图 43

表格清单

3.1. 系统目录 8
3.2. pg_aggregate 列 9
3.3. pg_attrdef 列 9
3.4. pg_attribute 列 10
3.5. pg_class 列 11
3.6. pg_database 列 13
3.7. pg_description 列 14
3.8. pg_group 列 14
3.9. pg_index 列 14
3.10. pg_inherits 列 15
3.11. pg_language 列 15
3.12. pg_largeobject 列 16
3.13. pg_listener 列 16
3.14. pg_operator 列 17
3.15. pg_proc 列 17
3.16. pg_relcheck 的列 18
3.17. pg_rewrite 列 19
3.18. pg_shadow 列 20
3.19. pg_statistic 列 20
3.20. pg_trigger 列 21
3.21. pg_type 列 22
7.1. Page Layout 41
B.1. 目录的缩进格式 64

范例清单

2.1. 一个简单的 Select 3

第 1 章 PostgreSQL 源代码

1.1. 格式

源代码格式使用 4 列制表宽度，目前保留制表符（即，不把制表符展开为空格）。

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
```

第 2 章 PostgreSQL 内部概述

作者

本章最初是[sim98]（Stefan Simkovics 在维也纳技术大学完成的硕士论文，由 O.Univ. Prof.Dr. Georg Gottlob 和 Univ.Ass. Mag. Katrin Seyr 指导）的一部分。

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。不要期望在这里看到详细的描述（我认为，一份涵盖 PostgreSQL 中使用的所有数据结构和函数的描述将超过 1000 页！）。本章旨在帮助读者理解后端从收到一个查询到发送结果为止，其内部总体上的控制流和数据流。

2.1. 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，并接收服务器返回的结果。

2. 解析器阶段检查应用程序（客户端）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 querytree 的规则（存储在系统目录中），然后执行这些规则体中给出的转换。
   重写系统的一个应用在于视图的实现。

   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）querytree，并创建一个将作为执行器输入的 queryplan。

   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. 执行器递归地遍历计划树，并以计划所表示的方式提取元组。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，最后返回得到的元组。

在后续各节中，我们将更详细地介绍上述列出的每一项内容，以便更好地理解 PostgreSQL 的内部控制和数据结构。

2.2. 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/服务器模型实现。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。由于我们per se（就其本身而言）并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。
图 \ref{connection} 展示了主进程 `postmaster`、服务器进程 `postgres` 和一个客户端应用程序之间的交互。

客户端进程既可以是 psql 前端（用于交互式 SQL 查询），也可以是任何使用 `libpg` 库实现的用户应用程序。注意，使用 ecpg（PostgreSQL 面向 C 的嵌入式 SQL 预处理器）实现的应用程序同样使用这个库。

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元组返回给客户端。

2.3. 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。

2.3.1. 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。实现时使用了著名的 Unix 工具 lex 和 yacc。

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。

为了更好地理解 PostgreSQL 在处理查询时所使用的数据结构，我们用一个例子来说明这些数据结构在每个阶段所发生的变化。这个例子包含下面这个简单的查询，
它将被用于后续各节的各种描述和图示中。该查询假设 The Supplier Database（供应商数据库）中给出的各个表已经定义好。

例 2.1. 一个简单的 Select

```
select s.sname, se.pno
    from supplier s, sells se
    where s.sno > 2 and s.sno = se.sno;
```

图 \ref{parsetree} 展示了 `gram.y` 中的语法规则和动作为例 2.1中给出的查询所构建的解析树（其中不含 where 子句的操作符树，后者见图 \ref{where_clause}，因为一张图放不下两个数据结构）。

树的顶层节点是一个 `SelectStmt` 节点。对于 SQL 查询 from 子句中出现的每一个表项，都会创建一个 `RangeVar` 节点，其中保存着别名的名称，以及一个指向 `RelExpr` 节点的指针，后者保存着关系的名称。所有 `RangeVar` 节点都被收集到一个列表中，挂接在 `SelectStmt` 节点的 `fromClause` 字段上。

对于 SQL 查询选择列表中出现的每一个表项，都会创建一个 `ResTarget` 节点，其中保存着一个指向 `Attr` 节点的指针。`Attr` 节点保存着该表项的关系名，以及一个指向 `Value` 节点的指针，后者保存着属性的名称。所有 `ResTarget` 节点都被收集到一个列表中，连接到 `SelectStmt` 节点的 `targetList` 字段上。

图 \ref{where_clause} 展示了为例 2.1中给出的 SQL 查询的 where 子句构建的操作符树，它挂接在 `SelectStmt` 节点的 `qual` 字段上。操作符树的顶层节点是一个表示 `AND` 运算的 `A_Expr` 节点。该节点有两个分别称为 `lexpr` 和 `rexpr` 的后继，分别指向两棵子树。挂接在 `lexpr` 上的子树表示限定条件 `s.sno > 2`，挂接在 `rexpr` 上的子树表示 `s.sno = se.sno`。对于每个属性，都会创建一个 `Attr` 节点，其中保存着关系的名称以及一个指向 `Value` 节点的指针，后者保存着属性的名称。对于查询中出现的常量项，则创建一个保存该值的 `Const` 节点。

2.3.2. 转换过程

转换过程以解析器返回的树作为输入，并递归地遍历它。如果找到一个 `SelectStmt` 节点，就把它转换为一个 `Query` 节点，后者将成为新数据结构的最顶层节点。图 \ref{transformed} 展示了转换后的数据结构（转换后的 where 子句部分见图 \ref{transformed_where}，因为一张图放不下所有部分）。

接下来检查 FROM 子句中的关系名是否为系统所知。对于每一个出现在系统目录中的关系名，都会创建一个 RTE 节点，其中包含关系名、别名和关系 id。从这时起，就用关系 id 来引用查询中给出的各个关系。所有 RTE 节点都被收集到范围表项列表中，该列表连接到 `Query` 节点的 `rtable` 字段上。如果在查询中检测到一个系统未知的关系名，将返回错误并且查询处理将被中止。

再接下来检查所用的属性名是否包含在查询给出的各个关系之中。对于找到的每一个属性，都会创建一个 TLE 节点，其中保存着一个指向 `Resdom` 节点（保存着列的名称）的指针和一个指向 `VAR` 节点的指针。`VAR` 节点中有两个重要的编号：`varno` 字段给出包含当前属性的关系在上述范围表项列表中的位置；`varattno` 字段给出该属性在关系内部的位置。如果找不到某个属性的名称，将返回错误并且查询处理将被中止。

2.4. 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：

- 第一种实现使用元组级处理，并且深度实现于执行器内部。每当访问到单独一个元组时，规则系统就会被调用。这种实现在 1995 年被移除，当时 PostgreSQL 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和规划器/优化器之间的一个模块。这种技术至今仍在使用。

有关 PostgreSQL 系统中规则的语法和创建的信息，请参阅 The PostgreSQL User's Guide（PostgreSQL 用户指南）。

2.4.1. 重写系统

查询重写系统是位于解析器阶段和规划器/优化器之间的一个模块。
它处理解析器阶段返回的树（表示一个用户查询），如果存在必须应用于该查询的规则，
就把这棵树重写为另一种形式。

2.4.1.1. 实现视图的技术

下面我们将概述查询重写系统的算法。为了便于说明，我们以如何使用规则实现视图作为例子。

给定下面这条规则：

```
create rule view_rule
as on select
to test_view
do instead
   select s.sname, p.pname
   from supplier s, sells se, part p
   where s.sno = se.sno and
         p.pno = se.pno;
```

每当检测到对关系 `test_view` 的 select 时，上面给出的规则就会被触发。此时执行的将不是从 `test_view` 中选择元组，而是规则动作部分中给出的 select 语句。

再给定下面这个针对 `test_view` 的用户查询：

```
select sname
from test_view
where sname <> 'Smith';
```

下面列出每当出现针对 `test_view` 的用户查询时，查询重写系统所执行的各个步骤。（下面的列表只是对算法的非正式描述，仅用于基本理解。详细描述请参阅[ston89]。）

`test_view` 重写

1. 取规则动作部分中给出的查询。

2. 调整 targetlist，使其符合用户查询中给出的属性的数量和顺序。

3. 把用户查询 where 子句中给出的限定条件，加到规则动作部分所给查询的限定条件上。

依照上面给出的规则定义，用户查询将被重写成下面的形式（注意，
重写是在解析器阶段返回的用户查询内部表示上完成的，
但派生出的新数据结构将表示下面这个查询）：

```
select s.sname
from supplier s, sells se, part p
where s.sno = se.sno and
      p.pno = se.pno and
      s.sname <> 'Smith';
```

2.5. 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。
它首先把查询中出现的各个关系所有可能的扫描和连接方式组合起来。
所创建的所有路径都会得到同一结果，而优化器的任务就是估算执行每条路径的代价，
并找出哪一条代价最低。

2.5.1. 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` 不是 '<>'，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/优化器只考虑在 where 限定条件中存在相应连接子句（即存在类似 `where rel1.attr1=rel2.attr2` 这样的限制）的每两个关系之间的连接。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：

- 嵌套迭代连接：左关系中找到的每一个元组，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。
- 归并排序连接：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：首先在右关系的连接属性上建立哈希。接着扫描左关系，
  并将找到的每个元组中的相应值作为哈希键，用来在右关系中定位元组。

2.5.2. 计划的数据结构

这里我们对计划中出现的节点做一点说明。图 \ref{plan} 展示了为例 \ref{simple_select} 中的查询生成的计划。

计划的顶层节点是一个 `MergeJoin` 节点，它有两个后继，一个挂接在 `lefttree` 字段上，另一个挂接在 `righttree` 字段上。每个子节点代表连接中的一个关系。如前所述，归并排序连接要求每个关系都已排序，因此我们在每个子计划中都会看到一个 `Sort` 节点。查询中给出的附加限定条件（`s.sno > 2`）被尽可能下推，挂接到相应子计划的叶子 `SeqScan` 节点的 `qpqual` 字段上。

`MergeJoin` 节点 `mergeclauses` 字段上挂接的列表包含有关连接属性的信息。`mergeclauses` 列表（以及 `targetlist`）中出现的 `VAR` 节点，其 `varno` 字段的值 `65000` 和 `65001` 表示：应当考虑的不是当前节点的元组，而是下一个“更深”节点（即各子计划的顶层节点）的元组。

注意，图 \ref{plan} 中出现的每一个 `Sort` 和 `SeqScan` 节点都有一个 `targetlist`，但由于空间不足，图中只画出了 `MergeJoin` 节点的那一个。

规划器/优化器的另一项任务是在 `Expr` 和 `Oper` 节点中固定操作符 id。如前所述，PostgreSQL 支持多种不同的数据类型，甚至可以使用用户自定义的类型。为了能够维护数量庞大的函数和操作符，必须把它们存储在一张系统表中。
每个函数和操作符都会得到一个唯一的操作符 id。根据限定条件等之中所使用属性的类型，必须使用相应的操作符 id。

2.6. 执行器

执行器接收由规划器/优化器创建的计划，并开始处理其顶层节点。对于我们的例子（例 \ref{simple_select} 中给出的查询）来说，顶层节点是一个 MergeJoin 节点。

在执行归并之前，必须先取到两个元组（每个子计划各一个）。
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，是一个 SeqScan 节点，而在处理该节点本身之前又必须先取得一个元组。于是执行器再次递归调用自身，去处理 SeqScan 节点的 lefttree 上挂接的子计划。

现在新的顶层节点是一个 Sort 节点。由于排序必须在整个关系上进行，当 Sort 节点第一次被访问时，执行器开始从 Sort 节点的子计划取元组，并把它们排序后放入一个临时关系（内存中或文件中）。（之后再检查 Sort 节点时，总是只从排好序的临时关系中返回一个元组。）

每当 Sort 节点的处理需要一个新元组时，执行器就会被递归调用，去处理作为子计划挂接的 SeqScan 节点。在关系（内部通过 scanrelid 字段给出的值引用）中扫描下一个元组。如果该元组满足挂接在 qpqual 上的树所给出的限定条件，就把它交回；否则继续取下一个元组，直到限定条件被满足。如果关系中的最后一个元组已被处理完毕，就返回一个 NULL 指针。

在 MergeJoin 的 lefttree 交回一个元组之后，righttree 会以同样的方式被处理。如果两个元组都已就绪，执行器就处理 MergeJoin 节点。每当需要来自某个子计划的一个新元组时，都会递归调用执行器来取得它。如果能够构造出一个连接后的元组，就把它交回，
至此计划树的一次完整处理就结束了。

上述步骤会对每个元组执行一次，直到对 MergeJoin 节点的处理返回一个 NULL 指针，表明处理已经完成。

第 3 章 系统目录

3.1. 概述

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。

表 3.1. 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”、“字段”）
pg_class	表、索引、序列（“关系”）
pg_database	此数据库集群中的数据库
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
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_relcheck	检查约束
pg_rewrite	查询重写器规则
pg_shadow	数据库用户
pg_statistic	优化器统计信息
pg_trigger	触发器
pg_type	数据类型

大多数目录的更详细文档见下文。与索引访问方法相关的目录在程序员指南中说明。

3.2. pg_aggregate

pg_aggregate 存储关于聚合函数的信息。聚合函数是对一个值集合（通常是查询条件匹配的每一行中的一列）进行操作、并从这些值计算出单个结果的函数。典型的聚合函数有 sum、count 和 max。

表 3.2. pg_aggregate 列

名称	类型	引用	描述
aggname	name		聚合函数的名称
aggowner	int4	pg_shadow.usesysid	聚合函数的所有者（创建者）
aggtransfn	regproc (function)	pg_proc.oid	转移函数
aggfinalfn	regproc (function)	pg_proc.oid	最终函数
aggbasetype	oid	pg_type.oid	该聚合函数的输入数据类型
aggtranstype	oid	pg_type.oid	聚合函数内部转换（状态）数据的类型
aggfinaltype	oid	pg_type.oid	结果的类型
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。如果该字段为 NULL，转换状态值从 NULL 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，请参见程序员指南。

聚合函数通过名称和参数类型来标识。因此 aggname 和 aggbasetype 构成复合主键。

3.3. pg_attrdef

该目录存储列的默认值。关于列的主要信息存储在 pg_attribute（见下文）中。只有显式指定了默认值的列（在建表时或添加列时）才会在这里有一个项。

表 3.3. 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

3.4. pg_attribute

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 3.4. 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
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。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		该列类型的 pg_type.typlen 的一个副本。
attnum	int2		列的编号。一般列从1开始向上编号。系统列，如 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，但装入内存中的元组描述符后，它可能被更新为该属性在元组中偏移量的缓存。
atttypmod	int4		atttypmod 记录建表时提供的类型特定数据（例如 varchar 列的最大长度）。它作为第三个参数被传递给类型特定的输入和输出函数。对于不需要此类数据的类型，该值一般为 -1。
attbyval	bool		该列类型的 pg_type.typbyval 的一个拷贝
attstorage	char		该列类型的 pg_type.typstorage 的一个副本
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 目录中某一元组的 OID。pg_proc 元组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

名称	类型	引用	描述
			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。
attalign	char		该列类型的 pg_type.typalign 的一个拷贝
attnotnull	bool		表示一个 NOT NULL 约束。可以改变此字段来启用或禁用该约束。
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。

3.5. pg_class

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。下文中，当我们指所有这些类型的对象时，就统称“关系”。并非所有字段对所有关系类型都有意义。

表 3.5. pg_class 列

名称	类型	引用	描述
relname	name		表、索引、视图等的名字
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
relam	oid	pg_am.oid	如果这是索引，则为所用的访问方法（btree、hash 等）
relfilenode	oid		该关系在磁盘上的文件名
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		表中的元组数。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	与该表关联的 TOAST 表的 OID，如果没有则为 0。TOAST 表把大属性“线外”存储在一个二级表中。

名称	类型	引用	描述
reltoastidxid	oid	pg_class.oid	该约束所在的表，如果不是表约束则为零
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，'s' = 特殊，'t' = 二级 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_relcheck 目录
reltriggers	int2		表上的触发器数；参见 pg_trigger 目录
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。

3.6. pg_database

pg_database 目录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，请参见管理员指南。

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 3.6. pg_database 列

名称	类型	引用	描述
datname	name		数据库名称
datdba	int4	pg_shadow.usesysid	数据库的所有者，最初是创建它的人
encoding	int4		该数据库的字符/多字节编码
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 插入或删除的所有元组，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 插入的所有元组，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。

3.7. pg_description

pg_description 表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。客户端应用程序可以通过与这个表连接来查看描述。许多内置系统对象的描述会由 psql 的 \d 命令显示出来。

表 3.7. pg_description 列

名称	类型	引用	描述
objoid	oid	any oid attribute	此描述所属对象的 oid
classoid	oid	pg_class.oid	该对象出现在其中的系统目录的 oid
objsubid	int4		对于表属性的注释，这是该属性的列号（objoid 和 classoid 指表本身）。对于所有其他对象类型，此字段目前为零。
description	text		作为该对象描述的任意文本。

3.8. pg_group

该目录定义组，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，请参见管理员指南。

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 3.8. 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

3.9. pg_index

pg_index 包含索引的一部分信息。其余大部分在 pg_class 中。

表 3.9. pg_index 列

名称	类型	引用	描述
indexrelid	oid	pg_class.oid	该索引的 pg_class 项的 oid
indrelid	oid	pg_class.oid	该索引所属表的 pg_class 项的 oid
indproc	regproc	pg_proc.oid	如果这是函数索引，则为已注册的过程
indkey	int2vector	pg_attribute.attnum	这是一个由至多 INDEX_MAX_KEYS 个值组成的向量（数组），指明该索引涉及哪些表列。例如，值 1 3 表示第一和第三列构成索引键。对于函数索引，这些列是函数的输入参数。

名称	类型	引用	描述
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”的引用。详见 pg_opclass。
indisclustered	bool		未使用
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。）
indreference	oid		未使用
indpred	text		部分索引谓词的表达式树（以 nodeToString 表示的形式）

3.10. pg_inherits

该目录记录关于表继承层次结构的信息。

表 3.10. pg_inherits 列

名称	类型	引用	描述
inhrelid	oid	pg_class.oid	这是对子表的引用，即它记录了所标识的表继承自另外某个表这一事实。
inhparent	oid	pg_class.oid	这是对父表的引用，inhrelid 所引用的表继承自该父表。
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。

3.11. pg_language

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 和程序员指南。

表 3.11. 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）
lanispl	bool		对于内部语言（如 SQL）为假，对于动态装载的语言处理器模块为真。它实质上意味着：如果为真，该语言可以被删除。

名称	类型	引用	描述
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），此字段无意义。
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。
lancompiler	text		目前未使用

3.12. pg_largeobject

pg_largeobject 保存构成“大对象”的数据。大对象在创建时被分配一个 OID 来标识。每个大对象都被拆分成足够小、便于作为行存储在 pg_largeobject 中的段或“页”。每页的数据量由 LOBLKSIZE 定义（目前是 BLCKSZ/4，通常是 2 KB）。

表 3.12. 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。

3.13. pg_listener

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 3.13. pg_listener 列

名称	类型	引用	描述
relname	name		通知条件名。（该名称不必与数据库中的任何实际关系匹配；“relname”一词是历史遗留的。）
listenerpid	int4		创建此项的后端进程的 PID。
notification	int4		如果该监听者没有待处理的事件，则为零。如果有

名称	类型	引用	描述
			事件待处理，则为发送通知的后端的 PID。

3.14. pg_operator

这些操作符参数的细节请参见 CREATE OPERATOR 和程序员指南。

表 3.14. pg_operator 列

名称	类型	引用	描述
oprname	name		操作符的名称
oprowner	int4	pg_shadow.usesysid	操作符的所有者（创建者）
oprprec	int2		未使用
oprkind	char		'b' = 中缀（“两侧”），'l' = 前缀（“左”），'r' = 后缀（“右”）
oprisleft	bool		未使用
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
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
oprcode	regproc		实现该操作符的函数
oprrest	regproc		该操作符的限制选择率估算函数
oprjoin	regproc		该操作符的连接选择率估算函数

3.15. pg_proc

该目录存储关于函数（或过程）的信息。CREATE FUNCTION 的描述和程序员指南包含某些字段含义的更多信息。

表 3.15. pg_proc 列

名称	类型	引用	描述
proname	name		函数的名字
proowner	int4	pg_shadow.usesysid	函数的所有者（创建者）

名称	类型	引用	描述
prolang	oid	pg_language.oid	实现语言或该函数的调用接口
proisinh	bool		未使用
proistrusted	bool		不具功能
proiscachable	bool		函数对相同的输入值返回相同的结果
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。
pronargs	int2		参数个数
proretset	bool		函数返回一个集合（即指定数据类型的多个值）
prorettype	oid	pg_type.oid	返回值的数据类型（如果函数不返回值则为 0）
proargtypes	oidvector	pg_type.oid	一个由函数参数的数据类型组成的向量
probyte_pct	int4		死代码
properbyte_cpu	int4		死代码
propercall_cpu	int4		死代码
prooutin_ratio	int4		死代码
prosrc	text		这个字段告诉函数处理器如何调用该函数。它可能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。

目前，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。

目前，probin 只用于动态装载的 C 函数，此时它给出包含该函数的共享库文件名。

3.16. pg_relcheck

该系统目录存储表上的 CHECK 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。）更多信息参见 CREATE TABLE。

表 3.16. pg_relcheck 的列

名称	类型	引用	描述
rcrelid	oid	pg_class.oid	该检查约束所在的表
rcname	name		约束名

名称	类型	引用	描述
rcbin	text		约束表达式的内部表示
rcsrc	text		约束表达式的人类可读表示

注意

pg_class.relchecks 必须与该表中的项数一致。

3.17. pg_rewrite

该系统目录存储表和视图的重写规则。

表 3.17. pg_rewrite 列

名称	类型	引用	描述
rulename	name		规则名称
ev_type	char		规则针对的事件类型：'1' = SELECT，'2' = UPDATE，'3' = INSERT，'4' = DELETE
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）
is_instead	bool		为真表示是一个 INSTEAD 规则
ev_qual	text		规则限定条件的表达式树（以 nodeToString 表示的形式）
ev_action	text		规则动作的查询树（以 nodeToString 表示的形式）

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。

3.18. pg_shadow

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。

管理员指南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 3.18. 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
usetrace	bool		未使用
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，也只有当此属性为真时才能这样做。）
passwd	text		口令
valuntil	abstime		账号过期时间（仅用于口令认证）

3.19. pg_statistic

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，即使统计信息是最新的，
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 NULL 率）在 pg_statistic 中有专门的字段。其余所有内容都存储在“槽位”中，槽位是相关联的字段组，
其内容由槽位各字段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，它只公开当前用户可读的那些表的相关信息。pg_stats 的设计还在于以比底层 pg_statistic 表更易读的形式呈现信息。

表 3.19. 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		该列各项中为 NULL 的比例
stawidth	int4		非 NULL 项的平均存储宽度（字节）
stadistinct	float4		列中不同非 NULL 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 stadistinct = -0.5 表示）。零表示不同值的数目未知。

名称	类型	引用	描述
stakindN	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。
staopN	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。
stanumbersN	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 NULL。
stavaluesN	text[]		第 N 个“槽位”的相应种类的列数据值；如果该槽位种类不存储任何数据值则为 NULL。为做到与数据类型无关，所有列数据值都被转换为外部文本形式并作为 TEXT 存储项存储。

3.20. pg_trigger

该系统目录存储表上的触发器。更多信息参见 CREATE TRIGGER。

表 3.20. 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（不必唯一）
tgfoid	oid	pg_proc.oid	要被触发器调用的函数
tgtype	int2		标识触发条件的位掩码
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		如果触发器是一个 RI 约束则为真
tgconstrname	name		RI 约束名
tgconstrrelid	oid	pg_class.oid	RI 约束所引用的表
tgdeferrable	bool		如果可推迟则为真
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。

3.21. pg_type

该目录存储关于数据类型的信息。标量类型（“基本类型”）用 CREATE TYPE 创建。数据库中的每个表也会创建一个复杂类型，以表示该表的行结构。

表 3.21. pg_type 列

名称	类型	引用	描述
typname	name		数据类型的名字
typowner	int4	pg_shadow.usesysid	类型的所有者（创建者）
typlen	int2		该类型存储表示的长度，变长则为 -1
typprtlen	int2		未使用
typbyval	bool		typbyval 决定内部例程按值还是按引用传递该类型的值。只有等价于 char、short 和 int 的项可以按值传递，因此如果类型长度不是 1、2 或 4 字节，PostgreSQL 就没有按值传递的选项，因此 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 表示复杂类型（即表的行类型）。如果 typtype 为 c，则 typrelid 是该类型在 pg_class 中项的 OID。
typisdefined	bool		如果类型已定义则为真，如果这是尚未定义类型的占位项则为假。当 typisdefined 为假时，除了类型名和 OID 之外不能依赖任何信息。
typdelim	char		解析数组输入时分隔该类型的两个值的字符。注意该分隔符与数组元素数据类型相关联，而不是数组数据类型。
typrelid	oid	pg_class.oid	如果这是复杂类型（见 typtype），此字段指向定义相应表的 pg_class

名称	类型	引用	描述
			项。理论上表可以用作复合数据类型，但这并不是完全可用的。
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 N 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		输入函数
typoutput	regproc		输出函数
typreceive	regproc		未使用
typsend	regproc		未使用
typalign	char		typalign 是存储该类型的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。当多个值连续存储时（例如磁盘上一个完整行的表示），会在该类型的 datum 之前插入填充，使其从指定的边界开始。对齐参照点是序列中第一个 datum 的起始处。 可能的取值有： • 'c' = CHAR 对齐，即不需要对齐。 • 's' = SHORT 对齐（在大多数机器上为 2 字节）。 • 'i' = INT 对齐（在大多数机器上为 4 字节）。 • 'd' = DOUBLE 对齐（在很多机器上为 8 字节，但绝不是所有机器）。

名称	类型	引用	描述
			注意 对于系统表中使用的类型，在 `pg_type` 中定义的大小和对齐与编译器在表示表行的结构体中布置字段的方式一致，是至关重要的。
typstorage	`char`		`typstorage` 告诉我们对于变长类型（即 `typlen` = -1 的类型），该类型是否为 TOAST 做了准备，以及该类型属性的默认策略应当是什么。可能的取值为 • 'p'：值必须总是以原样（plain）存储。 • 'e'：值可以存储在一个“二级”关系中（如果关系有的话，见 `pg_class.reltoastrelid`）。 • 'm'：值可以内联压缩存储。 • 'x'：值可以内联压缩存储，或移出到“二级”存储中。 注意 'm' 列也可以移出到二级存储，但只作为最后手段（'e' 和 'x' 列会先被移动）。
typdefault	`text`		没有默认值的类型，`typdefault` 为 NULL。如果非 NULL，它包含该类型默认值的外部字符串表示。

第 4 章 前端/后端协议

注意

由 Phil Thompson（phil@river-bank.demon.co.uk）编写。协议 2.0 的更新由 Tom Lane（tgl@sss.pgh.pa.us）完成。

PostgreSQL 使用基于消息的前端与后端之间的通信协议。该协议在 TCP/IP 以及 Unix 域套接字之上实现。PostgreSQL 6.3 在协议中引入了版本号。这样做的目的是仍然允许来自早期版本前端的连接，但本文档不涵盖那些早期版本使用的协议。

本文档描述该协议的 2.0 版，它在 PostgreSQL 6.4 及更高版本中实现。

建立在此协议之上的更高层特性（例如 libpq 在连接建立后如何传递某些环境变量）在别处介绍。

4.1. Overview

前端打开一个到服务器的连接并发送一个启动包。其中包括用户要连接的用户名和数据库名。
然后服务器使用这些信息以及 pg_hba.conf 文件中的信息来确定它要求前端进一步发送什么认证信息（如果有的话），并相应地响应前端。

然后前端发送任何所需的认证信息。服务器验证这些信息后，响应前端已通过认证，
并发送一条消息指示启动成功（正常情况）或失败（例如数据库名无效）。

为了高效地为多个客户端服务，服务器为每个客户端启动一个新的后端进程。
但这对协议是透明的。在当前的实现中，检测到传入连接后会立即创建一个新的子进程。

当前端希望断开连接时，它发送一个适当的包并关闭连接，而不等待后端的响应。

包以数据流的形式发送。第一个字节决定包的其余部分应期待什么。
例外是作为启动和认证交换一部分发送的包，它们由包长度后跟包本身组成。
这一差异是历史原因造成的。

4.2. Protocol

本节描述消息流。有四种不同的流，取决于连接的状态：启动、查询、函数调用和终止。
还有针对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。

4.2.1. 启动

最初，前端发送一个 StartupPacket。服务器使用这些信息以及 `pg_hba.conf` 文件的内容来确定 what authentication method the frontend must use. 然后服务器以下列消息之一响应：

ErrorResponse

然后服务器立即关闭连接。

AuthenticationOk

认证交换完成。

AuthenticationKerberosV4

然后前端必须与服务器进行 Kerberos V4 认证对话（此处不描述，属于 Kerberos 规范的一部分）。如果成功，服务器以 AuthenticationOk 响应，否则以 ErrorResponse 响应。

AuthenticationKerberosV5

然后前端必须与服务器进行 Kerberos V5 认证对话（此处不描述，属于 Kerberos 规范的一部分）。如果成功，服务器以 AuthenticationOk 响应，否则以 ErrorResponse 响应。

AuthenticationCleartextPassword

然后前端必须发送一个包含明文形式密码的 PasswordPacket。如果密码正确，服务器以 AuthenticationOk 响应，否则以 ErrorResponse 响应。

AuthenticationCryptPassword

然后前端必须发送一个 PasswordPacket，其中包含用 crypt(3) 加密的密码，使用 AuthenticationCryptPassword 包中指定的 2 字符盐。如果密码正确，服务器以 AuthenticationOk 响应，否则以 ErrorResponse 响应。

AuthenticationMD5Password

然后前端必须发送一个 PasswordPacket，其中包含用 MD5 加密的密码，使用 AuthenticationMD5Password 包中指定的 4 字符盐。如果密码正确，服务器以 AuthenticationOk 响应，否则以 ErrorResponse 响应。

AuthenticationSCMCredential

此方法只可能在支持 SCM 凭据消息的平台上用于本地 Unix 域连接。前端必须发出一条 SCM 凭据消息，然后发送单个数据字节。（该数据字节的内容无关紧要；它只是用来确保服务器等待足够长的时间以接收凭据消息。）如果凭据可接受，服务器以 AuthenticationOk 响应，否则以 ErrorResponse 响应。

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。

收到 AuthenticationOk 之后，前端应当等待服务器的进一步消息。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。

ReadyForQuery

启动完成。前端现在可以发出查询或函数调用消息。

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 ErrorResponse。

ReadyForQuery 消息与后端在每个查询周期之后将发出的消息相同。取决于前端的编码需要，既可以把 ReadyForQuery 视为一个查询周期的开始（然后 BackendKeyData 表示启动阶段的成功结束），也可以把 ReadyForQuery 视为启动阶段和每个后续查询周期的结束。

4.2.2. Query

查询周期由前端向 backend 发送 Query 消息发起。然后后端根据查询命令串的内容发送一个或多个响应消息，最后发送 ReadyForQuery 响应消息。ReadyForQuery informs the frontend that it may safely send a new query or function call.

后端可能的响应消息有：

CompletedResponse

一条 SQL 命令正常完成。

CopyInResponse

后端准备好把数据从前端复制到表。前端然后应发送 CopyDataRows 消息。后端随后将以带 COPY 标签的 CompletedResponse 消息响应。

CopyOutResponse

后端准备好把数据从表复制到前端。然后它发送 CopyDataRows 消息，再发送带 COPY 标签的 CompletedResponse 消息。

CursorResponse

对 SELECT、FETCH、INSERT、UPDATE 或 DELETE 查询的响应的开始。在 FETCH 的情况下，消息中包含所取游标的名称。否则该消息总是提到“空白”游标。

RowDescription

指示将要对 SELECT 或 FETCH 查询返回行。消息内容描述行的布局。之后对于返回给前端的每一行，都会跟随一条 AsciiRow 或 BinaryRow 消息（取决于是否指定了二进制游标）。

EmptyQueryResponse

识别到一个空查询串。

ErrorResponse

发生了一个错误。

ReadyForQuery

查询串的处理完成。发送单独的消息来指示这一点，因为查询串可能包含多条 SQL 命令。(CompletedResponse marks the end of processing one SQL command, not the whole string.) ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

发出了与查询有关的警告消息。通知是其他响应之外的附加内容，
即后端将继续处理该命令。

对 SELECT 或 FETCH 查询的响应通常由 CursorResponse、RowDescription、零个或多个 AsciiRow 或 BinaryRow 消息以及最后的 CompletedResponse 组成。INSERT、UPDATE 和 DELETE

查询产生 CursorResponse 后跟 CompletedResponse。向前端或从前端 COPY 调用如上所述的特殊协议。所有其他查询类型通常只产生 CompletedResponse 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。

如果收到完全为空（除空白外无内容）的查询串，响应为 EmptyQueryResponse 后跟 ReadyForQuery。(The need to specially distinguish this case is historical.)

一旦发生错误，就会发出一条 ErrorResponse 消息，后面跟着 ReadyForQuery。ErrorResponse 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。

每当期待任何其他类型的消息时，前端必须准备好接受 ErrorResponse 和 NoticeResponse 消息。

实际上，即使前端不期待任何类型的消息（即后端名义上空闲）时，NoticeResponse 也可能到达。（特别是，后端可能被其父进程命令终止。在这种情况下它会在关闭连接之前发送一条 NoticeResponse。）建议前端在发出任何新命令之前检查此类异步通知。

此外，如果前端发出任何 LISTEN 命令，那么它必须准备好在任何时间接受 NotificationResponse 消息；见下文。

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。

4.2.3. 函数调用

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。

后端可能的响应消息有：

ErrorResponse

发生了一个错误。

FunctionResultResponse

函数调用已执行并返回了结果。

FunctionVoidResponse

函数调用已执行但没有返回结果。

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。

每当期待任何其他类型的消息时，前端必须准备好接受 ErrorResponse 和 NoticeResponse 消息。Also, if it issues any `LISTEN` commands then it must be prepared to accept NotificationResponse messages at any time; see below.

4.2.4. 通知响应

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）。

通知响应在协议的任何位置（启动之后）都是允许的，除非在另一个后端消息之内。Thus, the frontend must be prepared to recognize a NotificationResponse message whenever it is expecting any message. Indeed, it should be able to handle NotificationResponse messages even when it is not engaged in a query.

NotificationResponse

A `NOTIFY` command has been executed for a name for which a previous `LISTEN` command was executed. Notifications may be sent at any time.

可能值得指出，listen 和 notify 命令中使用的名称不必与 SQL 数据库中关系（表）的名称有任何关系。通知名称只是任意选择的条件名称。

4.2.5. 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，前端打开一个到服务器的新连接并发送 CancelRequest 消息，而不是通常通过新连接发送的 StartupPacket 消息。服务器将处理此请求然后关闭连接。出于安全原因，不对取消请求消息作直接回复。

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。

4.2.6. Termination

正常的优雅终止过程是前端发送 Terminate 消息并立即关闭连接。收到该消息后，后端立即关闭连接并终止。

不优雅的终止可能由于任一端的软件故障（即核心转储）而发生。
如果前端或后端看到连接意外关闭，应当清理并终止。如果前端不想终止自身，
可以选择重新联系服务器来启动一个新的后端。

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。

4.2.7. SSL 会话加密

近期的 PostgreSQL 版本允许用 SSL 加密前端/后端通信。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 StartupPacket。然后服务器以一个包含 Y 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 Y 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 StartupPacket。在这种情况下 StartupPacket 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 StartupPacket 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。

4.3. 消息数据类型

本节描述消息中使用的基本数据类型。

Int*n*(*i*)

一个按网络字节序的 n 位整数。如果指定了 i，它就是字面值。Eg. Int16, Int32(42).

LimString*n*(*s*)

恰好 n 字节的字符数组，解释为以空字符结尾的字符串。如果空间不足则省略零字节。如果指定了 s，它就是字面值。例如 LimString32、LimString64("user")。

String(*s*)

常规的以空字符结尾的 C 字符串，没有长度限制。如果指定了 s，它就是字面值。Eg. String, String("user").

> **注意**
>
> 后端可返回的字符串的长度没有预定义的限制。
> 前端的一个好编码策略是使用可扩展的缓冲区，
> 这样任何能装进内存的东西都可以接受。如果做不到这一点，
> 就读取完整的字符串并丢弃固定大小缓冲区装不下的尾部字符。

Byte*n*(*c*)

恰好 n 字节。如果指定了 c，它就是字面值。Eg. Byte, Byte1('\n').

4.4. 消息格式

本节描述每种消息的详细格式。每种消息可以由前端（F）、后端（B）或两者（F & B）发送。

AsciiRow (B)

Byte1('D')

标识此消息为 ASCII 数据行。（先前的 RowDescription 消息定义了行中字段的数量及其数据类型。）

Byte*n*

一个位图，行中每个字段一位。第 1 个字段对应第 1 个字节的第 7 位（MSB），第 2 个字段对应第 1 个字节的第 6 位，第 8 个字段对应第 1 个字节的第 0 位（LSB），第 9 个字段对应第 2 个字节的第 7 位，依此类推。如果对应字段的值不是 NULL，则每个位被置位。如果字段数不是 8 的倍数，位图中最后一个字节的剩余部分就被浪费了。

然后，对于每个具有非 NULL 值的字段，有以下内容：

Int32

指定字段的值的尺寸，包括此尺寸本身。

Byte*n*

以 ASCII 字符指定字段本身的值。n 是上面的尺寸减 4。字段数据中没有尾随的零字节；如果前端需要，必须自己添加一个。

AuthenticationOk (B)

Byte1('R')

标识此消息为认证请求。

Int32(0)

指定认证成功。

AuthenticationKerberosV4 (B)

Byte1('R')

标识此消息为认证请求。

Int32(1)

指定需要 Kerberos V4 认证。

AuthenticationKerberosV5 (B)

Byte1('R')

标识此消息为认证请求。

Int32(2)

指定需要 Kerberos V5 认证。

AuthenticationCleartextPassword (B)

Byte1('R')

标识此消息为认证请求。

Int32(3)

指定需要明文密码。

AuthenticationCryptPassword (B)

Byte1('R')

标识此消息为认证请求。

Int32(4)

指定需要 crypt() 加密的密码。

Byte2

加密密码时使用的盐。

AuthenticationMD5Password (B)

Byte1('R')

标识此消息为认证请求。

Int32(5)

指定需要 MD5 加密的密码。

Byte4

加密密码时使用的盐。

AuthenticationSCMCredential (B)

Byte1('R')

标识此消息为认证请求。

Int32(6)

指定需要 SCM 凭据消息。

BackendKeyData (B)

Byte1('K')

标识此消息为取消密钥数据。如果前端希望以后能够发出 CancelRequest 消息，就必须保存这些值。

Int32

此后端的进程 ID。

Int32

此后端的密钥。

BinaryRow (B)

Byte1('B')

标识此消息为二进制数据行。（先前的 RowDescription 消息定义了行中字段的数量及其数据类型。）

Byte*n*

一个位图，行中每个字段一位。第 1 个字段对应第 1 个字节的第 7 位（MSB），第 2 个字段对应第 1 个字节的第 6 位，第 8 个字段对应第 1 个字节的第 0 位（LSB），第 9 个字段对应第 2 个字节的第 7 位，依此类推。如果对应字段的值不是 NULL，则每个位被置位。如果字段数不是 8 的倍数，位图中最后一个字节的剩余部分就被浪费了。

然后，对于每个具有非 NULL 值的字段，有以下内容：

Int32

Specifies the size of the value of the field, excluding this size.

Byte*n*

以二进制格式指定字段本身的值。*n* 为上述大小。

CancelRequest (F)

Int32(16)

包的尺寸（字节）。

Int32(80877102)

取消请求码。该值经过选择，使其高 16 位包含 1234，低 16 位包含 5678。（为避免混淆，此代码不得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。

CompletedResponse (B)

Byte1('C')

标识此消息为完成响应。

String

命令标签。这通常是标识完成了哪条 SQL 命令的单个词。

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 则 oid 是所插入行的对象 ID，否则 oid 为 0。

对于 DELETE 命令，标签是 DELETE rows，其中 rows 是删除的行数。

对于 UPDATE 命令，标签是 UPDATE rows，其中 rows 是更新的行数。

CopyDataRows (B & F)

这是一个行流，其中每行以 Byte1('\n') 结束。然后跟随序列 Byte1('\\')、Byte1('.')、Byte1('\n')。

CopyInResponse (B)

Byte1('G')

标识此消息为 Start Copy In 响应。前端现在必须发送 CopyDataRows 消息。

CopyOutResponse (B)

Byte1('H')

标识此消息为 Start Copy Out 响应。此消息后面将跟随 CopyDataRows 消息。

CursorResponse (B)

Byte1('P')

标识此消息为游标响应。

String

游标的名称。如果游标是隐式的，则为空。

EmptyQueryResponse (B)

Byte1('I')

标识此消息为对空查询串的响应。

String("")

未使用。

ErrorResponse (B)

Byte1('E')

标识此消息为错误。

String

错误消息本身。

FunctionCall (F)

Byte1('F')

标识此消息为函数调用。

String("")

未使用。

Int32

指定要调用的函数的对象 ID。

Int32

指定提供给函数的参数数量。

然后，对每个参数，有下列内容：

Int32

Specifies the size of the value of the argument, excluding this size.

Byte*n*

以二进制格式指定字段本身的值。*n* 为上述大小。

FunctionResultResponse (B)

Byte1('V')

标识此消息为函数调用结果。

Byte1('G')

指定返回了非空结果。

Int32

指定结果的值的尺寸，不包括此尺寸本身。

Byte*n*

以二进制格式指定结果本身的值。n 是上面的尺寸。

Byte1('0')

未使用。（严格说来，FunctionResultResponse 和 FunctionVoidResponse 是同一个东西，只是消息的某些部分是可选的。）

FunctionVoidResponse (B)

Byte1('V')

标识此消息为函数调用结果。

Byte1('0')

指定返回了空结果。

NoticeResponse (B)

Byte1('N')

标识此消息为通知。

String

通知消息本身。

NotificationResponse (B)

Byte1('A')

标识此消息为通知响应。

Int32

发出通知的后端进程的进程 ID。

String

引发此通知的条件名称。

PasswordPacket (F)

Int32

包的尺寸（字节）。

String

口令（如果要求则已加密）。

Query (F)

Byte1('Q')

标识此消息为查询。

String

查询串本身。

ReadyForQuery (B)

Byte1('Z')

标识消息类型。每当后端准备好进入新的查询周期时发送 ReadyForQuery。

RowDescription (B)

Byte1('T')

标识此消息为行描述。

Int16

指定一行中的字段数（可以为零）。

Then, for each field, there is the following:

String

指定字段名。

Int32

指定字段类型的对象 ID。

Int16

指定类型尺寸。

Int32

指定类型修饰符。

SSLRequest (F)

Int32(8)

包的尺寸（字节）。

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。）

StartupPacket (F)

Int32(296)

包的尺寸（字节）。

Int32

协议版本号。高 16 位是主版本号。低 16 位是次版本号。

LimString64

数据库名，如果为空则默认为用户名。

LimString32

用户名。

LimString64

要由服务器传递给后端子进程的任何附加命令行参数。

LimString64

未使用。

LimString64

后端应用于调试消息的可选 tty。（目前此字段不受支持并被忽略。）

Terminate (F)

Byte1('X')

标识此消息为终止。

第 5 章 gcc 默认优化

Brian Gallew

> **注意**
>
> 由 Brian Gallew（<geek+@cmu.edu>）贡献

把 gcc 配置成默认使用某些标志只是编辑 /usr/local/lib/gcc-lib/*platform*/*version*/specs 文件的一件简单事。这个文件的格式相当简单。文件分成若干节，每节三行。第一行是 "**section_name*:"（例如 "*asm:"）。第二行是标志列表，第三行为空。

最容易的修改是把想要的默认标志追加到相应节的列表中。作为示例，假设我在一台 '486 上运行 linux，gcc 2.7.2 安装在默认位置。在文件 /usr/local/lib/gcc-lib/i486-linux/2.7.2/specs 的第 13 行处我找到下面这一节：

```
- ----------SECTION----------
*cc1:


- ----------SECTION----------
```

如你所见，没有任何默认标志。如果我希望 C 代码的编译总是使用 "-m486 -fomit-frame-pointer"，我会把它改成这样：

```
- ----------SECTION----------
*cc1:
- -m486 -fomit-frame-pointer

- ----------SECTION----------
```

如果我还想为另一台闲置的较旧 linux 机器生成 386 代码，就必须把它改成这样：

```
- ----------SECTION----------
*cc1:
%{!m386:-m486} -fomit-frame-pointer

- ----------SECTION----------
```

这样就会总是省略帧指针，并且在命令行上未指定 -m386 时生成 486 优化的代码。

实际上你可以用 specs 文件做相当多的定制。但始终要记住，这些更改是全局的，会影响系统的所有用户。

第 6 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。

6.1. 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。

6.2. 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`、`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。

6.3. 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
```

第 7 章 页文件

对数据库文件默认页格式的描述。

本节概述 PostgreSQL 表使用的页格式。用户定义的访问方法不必使用这种页格式。

在下面的说明中，假定一个字节（byte）包含 8 位。此外，术语项（item）指存储在 PostgreSQL 表中的数据。

表 7.1展示了普通 PostgreSQL 表和 PostgreSQL 索引（例如 B-tree 索引）中的页面是如何组织的。

表 7.1. 页面布局示例

项	描述
itemPointerData	
filler	
itemData...	
未分配空间	
ItemContinuationData	
特殊空间	
“ItemData 2”	
“ItemData 1”	
ItemIdData	
PageHeaderData	

每个页面的前 8 个字节由页头（PageHeaderData）组成。在页头中，前三个 2 字节整数字段（lower、upper 和 special）分别表示到未分配空间开始处、到未分配空间结束处以及到特殊空间开始处的字节偏移。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 opaque 编码页面大小和页面内部碎片的信息。每个页面都存储页面大小，
因为缓冲池中的帧在一个表内可以逐帧细分为大小相等的页面。
内部碎片信息用来帮助确定何时应进行页面重组。

页头之后是项标识符（ItemIdData）。新的项标识符从未分配空间的前四个字节分配。
由于项标识符在被释放之前从不移动，其索引可用来指示一个项在页面上的位置。事实上，
PostgreSQL 创建的每个指向项的指针（ItemPointer）都由一个帧号和一个项标识符的索引组成。项标识符包含到项开始处的字节偏移、以字节为单位的长度以及一组影响其解释的属性位。

项本身存储在从未分配空间末尾向后分配的空间中。通常不对项进行解释。
但当项太长而无法放在单个页面上或希望对项进行分片时，项会被切分，
每一片按下述方式作为不同的项处理。第一片到倒数第二片放在项延续结构（
ItemContinuationData）中。该结构包含指向下一片的 itemPointerData 和这一片本身。最后一片按正常方式处理。

第 8 章 遗传查询优化

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。

8.1. 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。

当前的PostgreSQL查询优化器实现会在可选策略空间中执行近似穷举搜索。
这种查询优化技术不足以支持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。

下文我们提出把遗传算法的实现作为数据库查询优化问题的一种可选方案。

8.2. 遗传算法

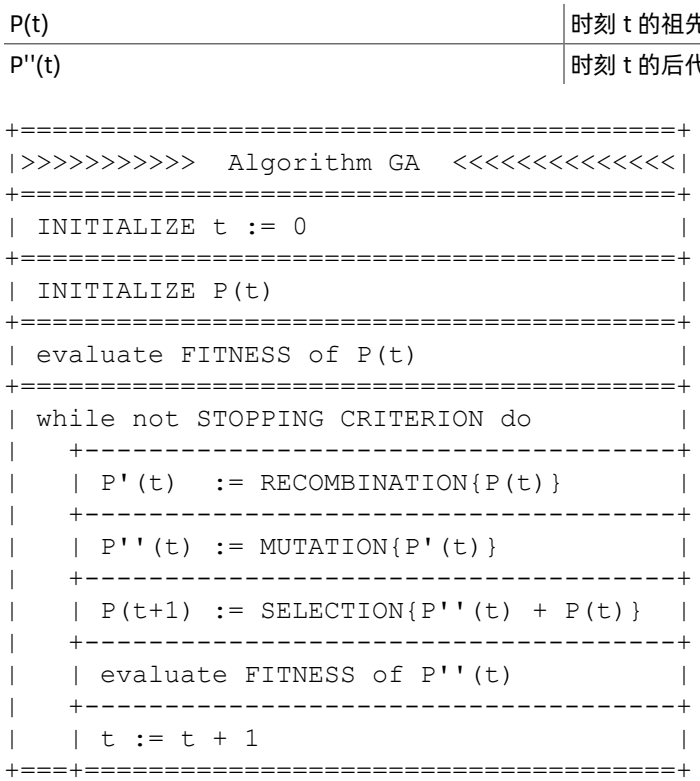
遗传算法（GA）是一种通过确定的、随机化的搜索进行工作的启发式优化方法。
优化问题的可能解集合被视为由若干个体组成的种群。
个体对其环境的适应程度由其适应度表示。

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 8.1. 遗传算法的结构图



8.3. 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。

8.3.1. 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

8.4. 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 9 章 本地语言支持

Peter Eisentraut

9.1. 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。

9.1.1. 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。

如果想开始一项新的翻译工作或想做消息目录合并（后文描述），你分别需要 GNU 兼容实现中的程序 `xgettext` 和 `msgmerge`。以后我们会尽力安排成使用打包的源码发行版时不需要 `xgettext`。（从 CVS 获取则仍需要它。）目前推荐 GNU gettext 0.10.36 或更新版本。

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。

9.1.2. 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。

9.1.3. 创建和维护消息目录

好，那么如何创建一个“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。
如果存在文件 `nls.mk`，说明该程序已经为翻译做好了准备。

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。

如果你需要启动一项新的翻译工作，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 代表什么？我也不知道。）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。

9.1.4. 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。

[1] http://lcweb.loc.gov/standards/iso639-2/englangn.html
[2] http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/en_listp1.html

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。

编辑译文时需要记住下面几点：

- 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 printf 格式串，译文也需要是。译文还需要以相同的顺序包含相同的格式说明符。
  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。
  这种情况下可以使用这种格式：

  ```
  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，并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。
- 保持原字符串的风格和语气。具体来说，不是完整句子的消息（`cannot open file %s`）也许不应以大写字母开头（如果你的语言区分大小写），
  也不应以句点结尾（如果你的语言使用标点符号）。
- 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。

9.2. 给程序员

本节描述如何在属于 PostgreSQL 发行版的程序或库中支持本地语言支持。目前它只适用于 C 程序。

为程序添加 NLS 支持

1. 把这段代码插入程序的启动序列：

   ```
   #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。）

   这往往会增加很多杂乱代码。一个常见的捷径是

   ```
   #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 `elog()`，另一种可行的解决办法是让这个函数在内部对所有输入值调用 `gettext`。

3. 在包含程序源码的目录中添加一个文件 `nls.mk`。这个文件将作为 makefile 读取。这里需要进行下列变量赋值：

   CATALOG_NAME

   程序名，如 `textdomain()` 调用中所提供的那样。

   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的列表，即那些用 `gettext` 或替代方案标记的文件。最终，这将包括该程序几乎所有源文件。如果这个列表太长，可以让第一个“文件”是 `+`，第二个词是一个每行包含一个文件名的文件。

   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。
   默认情况下只有 `gettext()` 调用是已知的。如果你用了 `_` 或其他标识符，需要在此列出它们。如果可翻译字符串不是第一个参数，条目的形式应为 `func:2`（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。

为了便于翻译消息，这里有一些指导原则：

- 不要因为偷懒而像下面这样在运行时拼接句子：

  ```
  printf("Files where %s.\n", flag ? "copied" : "removed");
  ```

  句子中的词序在其他语言里可能不同。

- 出于类似的原因，这样也不行：

  ```
  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。
将来我们可能会为此提供一个解决方案，但目前最好从设计上彻底避开这个问题。
可以这样写：

```
printf("number of copied files: %d", n);
```

- 如果你想向译者传达一些信息，比如消息打算如何与其他输出对齐，
  可以在字符串出现的位置前面加一条以 `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。

附录 A. CVS 代码库

Marc Fournier
Tom Lane
Thomas Lockhart

PostgreSQL 源代码使用 CVS 代码管理系统存储和管理。

至少有两种方法——匿名 CVS 和 CVSup——可以把 CVS 代码树从 PostgreSQL 服务器拉取到你的本地机器上。

A.1. 通过匿名 CVS 获取源码

如果你想定期跟上当前的源码，可以从我们的 CVS 服务器获取它们，然后不时地用 CVS 检索更新。

匿名 CVS

1. 你需要一份本地安装的 CVS（并发版本控制系统），可以从 http://www.cyclic.com/ 或任何 GNU 软件归档站点获取。我们目前推荐 1.10 版（写作时最新的版本）。许多系统默认就装有较新版本的 cvs。

2. 首次登录到 CVS 服务器：

   ```
   $ cvs -d :pserver:anoncvs@anoncvs.postgresql.org:/projects/cvsroot
    login
   ```

   系统会提示你输入口令；直接按 ENTER 即可。这只需要做一次，因为口令会保存在你主目录的 .cvspass 中。

3. 获取 PostgreSQL 源码：

   ```
   cvs -z3 -d :pserver:anoncvs@anoncvs.postgresql.org:/projects/
   cvsroot co -P pgsql
   ```

   这会把 PostgreSQL 源码安装到你当前所在目录的 pgsql 子目录中。

   > **注意**
   >
   > 如果你的因特网链路很快，可能不需要 -z3（它让 CVS 对传输的数据使用 gzip 压缩）。但在调制解调器速度的链路上，它能带来非常大的收益。

   这个初始检出比直接下载一个 tar.gz 文件稍慢一些；如果你用的是 28.8K 调制解调器，预计要花 40 分钟左右。CVS 的优势要等到你以后想更新文件集时才会显现出来。

4. 每当你想更新到最新的 CVS 源码时，cd 进入 pgsql 子目录，然后执行

   ```
   $ cvs -z3 update -d -P
   ```

这只会获取自你上次更新以来的变更。通常只要几分钟就能完成更新，
即使是在调制解调器速度的链路上。

5. 你可以在主目录中创建一个包含以下内容的 `.cvsrc` 文件来省些输入：

```
cvs -z3
update -d -P
```

这会为所有 cvs 命令提供 `-z3` 选项，并为 cvs update 提供 `-d` 和 `-P` 选项。这样你只需输入

```
$ cvs update
```

即可更新你的文件。

> **小心**
>
> 某些较旧版本的 CVS 有一个 bug，会导致所有检出的文件在你的目录中以全局可写的方式存放。如果发现这种情况，可以执行类似这样的命令
>
> ```
> $ chmod -R go-w pgsql
> ```
>
> 来正确设置权限。这个 bug 在 CVS 1.9.28 版中已经修复。

CVS 还能做许多别的事情，例如获取 PostgreSQL 源码的早期版本而不是最新开发版本。更多信息请查阅 CVS 附带的手册，或参见 http://www.cyclic.com/ 上的在线文档。

A.2. CVS 树的组织

> **作者**
>
> 由 Marc G. Fournier（<scrappy@hub.org>）写于 1998-11-05

命令 `cvs checkout` 有一个标志 `-r`，让你检出模块的某个特定版本。有了这个标志，例如在将来任何时候检索组成模块 `tc' 的 6_4 版本的源码都很容易：

```
$ cvs checkout -r REL6_4 tc
```

比如说，如果有人声称那个版本里有一个 bug，而你在当前工作副本中找不到它，这就很有用。

> **提示**
>
> 你也可以用 `-D` 选项检出模块在任一给定日期时的状态。

当你用同一个标记标记多个文件时，
可以把这个标记想成“一条画在文件名与版本号矩阵中穿过的曲线”。假设我们有 5 个文件，版本如下：

```
file1   file2   file3   file4   file5

1.1     1.1     1.1     1.1  /--1.1*      <-*-  TAG
1.2*-   1.2     1.2    -1.2*-
1.3  \- 1.3*-   1.3   / 1.3
1.4          \  1.4  /  1.4
              \-1.5*-   1.5
                1.6
```

那么标记 TAG 将引用 file1-1.2、file2-1.3 等等。

> **注意**
>
> 创建发行分支时，除了在命令中加上 -b 选项之外，其余都是一回事。

于是，为了创建 6.4 发行版，我做了以下操作：

```
$ cd pgsql
$ cvs tag -b REL6_4
```

这会为 RELEASE 树创建标记和分支。

对于拥有 CVS 写权限的人来说，为不同版本创建目录很简单。首先创建 RELEASE 和 CURRENT 两个子目录，免得把两者弄混。然后执行：

```
cd RELEASE
cvs checkout -P -r REL6_4 pgsql
cd ../CURRENT
cvs checkout -P pgsql
```

这会得到两棵目录树：RELEASE/pgsql 和 CURRENT/pgsql。从那时起，CVS 会跟踪哪棵目录树对应仓库的哪个分支，并允许对两棵树独立地进行更新。

如果你只在 CURRENT 源码树上工作，那么一切照旧，就像我们开始标记发行分支之前那样做即可。

在某个分支上完成初始检出之后

```
$ cvs checkout -r REL6_4
```

你在该目录结构中做的任何操作都局限于该分支。如果你对该目录结构打了一个补丁，
然后在其内部执行

```
cvs commit
```

那么补丁会被应用到该分支上，而且只应用到该分支。

A.3. 通过 CVSup 获取源码

检索 PostgreSQL 源码树的另一种匿名 CVS 替代方案是 CVSup。CVSup 由 John Polstra（<jdp@polstra.com>）开发，用于为 FreeBSD 项目[1]分发 CVS 仓库和其他文件树。

使用 CVSup 的一个主要优点是它能可靠地把整个 CVS 仓库复制到你的本地系统上，使 `log` 和 `diff` 等 cvs 操作可以快速本地进行。其他优点包括与 PostgreSQL 服务器的快速同步——这得益于一种高效的流式传输协议，它只发送上次更新以来的变更。

A.3.1. 准备 CVSup 客户端系统

CVSup 工作需要两个目录区域：一个本地 CVS 仓库（如果你获取的是快照而非仓库，那就只是一个目录区域；见下文），以及一个本地 CVSup 簿记区域。它们可以共存于同一棵目录树中。

决定你想把本地 CVS 仓库放在哪里。最近我们在其中一台系统上把仓库建在了 `/home/cvs/`，但以前曾把它放在 `/opt/postgres/cvs/` 的 PostgreSQL 开发树下。如果你打算把仓库放在 `/home/cvs/`，那么把

```
setenv CVSROOT /home/cvs
```

放进你的 `.cshrc` 文件，或者根据你使用的 shell 在 `.bashrc` 或 `.profile` 文件中放入类似的一行。

cvs 仓库区域必须先初始化。设置好 `CVSROOT` 之后，一条命令即可完成：

```
$ cvs init
```

之后列出 `CVSROOT` 目录时，你至少应该看到一个名为 `CVSROOT` 的目录：

```
$ ls $CVSROOT
CVSROOT/
```

A.3.2. 运行 CVSup 客户端

确认 cvsup 在你的路径中；在大多数系统上可以输入下面的命令来检查：

```
which cvsup
```

然后只需像下面这样运行 cvsup：

```
$ cvsup -L 2 postgres.cvsup
```

其中 `-L 2` 会启用一些状态消息，让你能监视更新的进度，而 *`postgres.cvsup`* 是你为 CVSup 配置文件指定的路径和名称。

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

下面是一个针对特定安装修改过的 CVSup 配置文件，它维护一个完整的本地 CVS 仓库：

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
# Modified by lockhart@fourpalms.org 1997-08-28
# - Point to my local snapshot source tree
# - Pull the full CVS repository, not just the latest snapshot
#
# Defaults that apply to all the collections
*default host=cvsup.postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
# enable the following line to get the latest snapshot
#*default tag=.
# enable the following line to get whatever was specified above or by
 default
# at the date specified below
#*default date=97.08.29.00.00.00

# base directory where CVSup will store its 'bookmarks' file(s)
# will create subdirectory sup/
#*default base=/opt/postgres # /usr/local/pgsql
*default base=/home/cvs

# prefix directory where CVSup will store the actual distribution(s)
*default prefix=/home/cvs

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

下面是从 PostgreSQL ftp 站点[2]建议使用的 CVSup 配置文件，它只获取当前快照：

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
#
# Defaults that apply to all the collections
*default host=cvsup.postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
*default tag=.

# base directory where CVSup will store its 'bookmarks' file(s)
*default base=/usr/local/pgsql
```

[2] ftp://ftp.postgresql.org/pub/CVSup/README.cvsup

```
# prefix directory where CVSup will store the actual distribution(s)
*default prefix=/usr/local/pgsql

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

A.3.3. 安装 CVSup

CVSup 有源码、预构建二进制以及 Linux RPM 等形式。用二进制比从源码构建容易得多，主要因为构建需要那个能力很强但体积庞大的 Modula-3 编译器。

从二进制安装 CVSup

如果 PostgreSQL ftp 站点[3]上发布了你所用平台的二进制，或者你运行的是 FreeBSD（CVSup 有对应的 port），就可以使用预构建的二进制。

> **注意**
>
> CVSup 最初是作为分发 FreeBSD 源码树的工具开发的。它以 “port” 的形式提供；对于运行 FreeBSD 的人来说，如果这些信息不足以说明如何获取和安装它，请在此贡献一份操作说明。

在写作本文时，可用的二进制包括 Alpha/Tru64、ix86/xBSD、HPPA/HP-UX 10.20、MIPS/IRIX、ix86/linux-libc5、ix86/linux-glibc、Sparc/Solaris 和 Sparc/SunOS。

1. 取回适合你平台的 cvsup 二进制 tar 文件（作为客户端不需要 cvsupd）。

   a. (可选的) 如果你运行的是 FreeBSD，安装 CVSup 的 port。

   b. (可选的) 如果是其他平台，到 PostgreSQL ftp 站点[4]查找并下载相应的二进制。

2. 检查 tar 文件以确认其内容和目录结构（如果有的话）。至少对 linux 的 tar 文件来说，里面只包含静态二进制和手册页，没有任何目录包装。

   a. 如果二进制位于 tar 文件的顶层，那么直接把 tar 文件解包到你的目标目录即可：

      ```
      $ cd /usr/local/bin
      $ tar zxvf /usr/local/src/cvsup-16.0-linux-i386.tar.gz
      $ mv cvsup.1 ../doc/man/man1/
      ```

   b. 如果 tar 文件里有目录结构，那么把 tar 文件解包到 /usr/local/src 中，再把二进制移动到上面所说的合适位置。

[3] ftp://ftp.postgresql.org/pub
[4] ftp://ftp.postgresql.org/pub

3. 确保新的二进制在你的路径中。

```
$ rehash
$ which cvsup
$ set path=(path to cvsup $path)
$ which cvsup
/usr/local/bin/cvsup
```

A.3.4. 从源码安装

从源码安装 CVSup 并不完全轻松，主要因为大多数系统需要先安装 Modula-3 编译器。这个编译器有 Linux RPM、FreeBSD 软件包或源码等形式。

> **注意**
>
> 从纯净源码安装 Modula-3 大约需要 200MB 磁盘空间，删除源码后缩小到大约 50MB。

Linux 安装

1. 安装 Modula-3。

   a. 从 Polytechnique Montréal[5] 获取 Modula-3 发行包，他们正在积极维护最初由 DEC 系统研究中心[6]开发的代码基础。PM3 的 RPM 发行包压缩后约 30MB。在写作本文时，1.1.10-1 版可以在 RH-5.2 上干净地安装，而 1.1.11-1 版显然是为另一个发行版（RH-6.0？）构建的，无法在 RH-5.2 上运行。

   > **提示**
   >
   > 这个 rpm 打包包含很多个 RPM 文件，所以你多半想把它们放到一个单独的目录中。

   b. 安装 Modula-3 的 rpm：

   ```
   # rpm -Uvh pm3*.rpm
   ```

2. 解包 cvsup 发行包：

```
# cd /usr/local/src
# tar zxf cvsup-16.0.tar.gz
```

3. 构建 cvsup 发行包，禁用 GUI 界面特性以避免依赖 X11 库：

```
# make M3FLAGS="-DNOGUI"
```

如果你想构建一个静态二进制以便移到可能没装 Modula-3 的系统上，试试：

[5] http://m3.polymtl.ca/m3
[6] http://www.research.digital.com/SRC/modula-3/html/home.html

```
# make M3FLAGS="-DNOGUI -DSTATIC"
```

4. 安装构建好的二进制：

```
# make M3FLAGS="-DNOGUI -DSTATIC" install
```

附录 B. 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
- Postscript，用于打印
- 手册页，用于快速参考。

另外，若干纯文本 README 类型的文件可以在 PostgreSQL源代码树各处找到，用于说明各种实现方面的问题。

文档被组织为若干“书”：

- 教程：面向新用户的入门介绍
- 用户指南：介绍 SQL 实现的文档
- 参考手册：程序和 SQL 命令的参考页
- 管理员指南：安装和服务器维护
- 程序员指南：编写客户端应用程序和服务器扩展
- 开发者指南：PostgreSQL 本身开发者的各种信息

所有书都以 HTML 和 Postscript 形式提供。参考手册包含的参考条目也以手册页形式提供。

HTML文档和手册页属于标准发行包的一部分，并会默认安装。Postscript 格式的文档可单独下载。

B.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。

B.2. Toolsets

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。

[1] http://www.oasis-open.org
[2] http://www.oasis-open.org/docbook
[3] http://www.freebsd.org/docproj/docproj.html

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。

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 docs 邮件列表，我们会把这些信息补充到这里。

B.2.1. Linux RPM 安装

许多发行版厂商在其发行版中提供了用于 DocBook 处理的完整 RPM 集合，它们通常基于 Red Hat Software 的 docbook-tools[10] 项目。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。

B.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。

[4] http://www.oasis-open.org/docbook/sgml/
[5] http://www.oasis-open.org/cover/ISOEnts.zip
[6] http://openjade.sourceforge.net
[7] http://docbook.sourceforge.net/projects/dsssl/index.html
[8] http://docbook2x.sourceforge.net
[9] http://jadetex.sourceforge.net
[10] http://sources.redhat.com/docbook-tools/

- 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）。

更多关于 FreeBSD 文档工具的信息，请参阅FreeBSD Documentation Project 的说明[11]。

B.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
```

B.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。

B.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：

[11] http://www.freebsd.org/doc/en_US.ISO8859-1/books/fdp-primer/tools.html

```
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。）

B.2.4.2. 安装 DocBook DTD 工具包

1. 获取 DocBook V3.1[12] 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。

   ```
   $ 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[13]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。）

B.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：

[12] http://www.oasis-open.org/docbook/sgml/3.1/docbk31.zip
[13] http://www.oasis-open.org/cover/ISOEnts.zip

```
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 B.3 节。

B.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[14]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
```

最后两条需要以root身份执行。

B.3. 构建文档

在构建文档之前，你需要像构建程序本身一样运行 `configure` 脚本。检查运行即将结束时的输出，它应该类似于这样：

```
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`。

一切设置妥当后，切换到`doc/src/sgml`目录，并运行下列命令之一：（记得使用 GNU make。）

- 要构建管理员指南的 HTML 版本：

  ```
  doc/src/sgml$ gmake admin.html
  ```

- 同一书的 RTF 版本：

  ```
  doc/src/sgml$ gmake admin.rtf
  ```

- 通过 JadeTeX 获得 DVI 版本：

[14] http://www.ctan.org

```
doc/src/sgml$ gmake admin.dvi
```

- 以及从 DVI 生成 Postscript：

```
doc/src/sgml$ gmake admin.ps
```

> **注意**
>
> 官方的 Postscript 格式文档是用另一种方法生成的。参见下面的第 B.3.3 节。

其他书可以用类似的命令构建，只需把 `admin` 换成 `developer`、`programmer`、`tutorial` 或 `user` 之一。使用 `postgres` 会构建全部 5 本书的集成版本，由于浏览器界面让你可以通过点击轻松地在全部文档之间跳转，这种方式很实用。

B.3.1. HTML

在`doc/src/sgml`中构建 HTML 文档时，某些生成的文件在书与书之间可能（或几乎肯定）会重名。因此在常规发行包中，文件并不放在那个目录里。相反，每本书的文件存储在一个 tar 归档中，并在安装时解包。要创建一组 HTML 文档包，使用命令

```
cd doc/src
gmake postgres.tar.gz
gmake tutorial.tar.gz
gmake user.tar.gz
gmake admin.tar.gz
gmake programmer.tar.gz
gmake install
```

在发行包中，这些归档位于 `doc` 目录中，并会随 `gmake install` 默认安装。

B.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
```

这会在 `doc/src` 目录中生成一个 tar 文件。

man 构建会产生大量令人困惑的输出，而且要产生高质量的结果需要特别小心。这方面仍有改进的余地。

B.3.3. 硬拷贝生成

硬拷贝 Postscript 文档的生成方法是：先把 SGML 源码转换为 RTF，然后导入 ApplixWare。经过少量清理（见下一节）后，把输出 “打印” 到一个 postscript 文件。

在生成 Postscript 硬拷贝时需要处理若干方面的问题，包括 RTF 修复、目录（ToC）生成和分页调整。

Applixware RTF 清理

硬拷贝过程不可或缺的组成部分 jade 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（ToC）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. 输入以下命令生成 RTF 输入（例如）：

   ```
   % cd doc/src/sgml
   % make tutorial.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，还必须替换把前面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：

   ```
   % cd doc/src/sgml
   % fixrtf tutorial.rtf
   ```

   或者

   ```
   % cd doc/src/sgml
   % fixrtf --refentry reference.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 M$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Applix 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.BookBuilding.CreateToC`构建新的 ToC。选择前三级标题。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：

   表 B.1. 目录的缩进格式

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。
   - 把插图插入文档。使用 ApplixWare 工具栏上的居中边距按钮把每幅图在页面上居中。

> **注意**
>
> 并非所有文档都有插图。你可以在 SGML 源文件中 grep 字符串 `graphic`，来找出文档中可能带有插图的部分。有少数插图在文档的不同部分重复出现。

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（每份文档）。
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 Applix 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。
11. 使用 gzip 压缩 Postscript 文件。把压缩后的文件放入 `doc` 目录。

B.3.4. 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` 文件对应于管理员指南中的对应章节，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`**。

B.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。

B.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[15]。

[15] http://www.lysator.liu.se/projects/about_psgml.html

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
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。）

B.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[16]，也带有 font-lock 和若干减少输入量的功能。

[16] http://nwalsh.com/emacs/docbookide/index.html

索引

符号

$libdir，182
.odbc.ini，80
事务隔离级别，
信号量，29
共享内存，29
别名
  查询中的表名，9
动态加载，
大小写敏感性
  SQL 命令，1
引号
  与标识符，2
数据区域 (见数据库集簇)
数据库集簇，16
死锁
  超时，
澳大利亚时区，
端口，
索引
  唯一，97
  基于函数，97
  部分，99
索引扫描，
继承，
超时
  死锁，
  认证，
连接，8
  外连接，9
  自连接，9
遗传查询优化，
配置
  服务器，19
顺序扫描，

A

aggregate，10
aggregate functions，8
  extending，204
alias (见 label)
all，77
and
  operator，40
any，77
ApplixWare，82
arrays，91，198
  constants，4
auto-increment (见 serial)
average，10
  function，76

B

B-tree (见 index)
backup，63
between，41
bigint，21
bigserial，23
binary strings
  concatenation，51
  length，51
bison，1
bit strings
  constants，2
  data type，38
BLOB (见 large object)
Boolean
  data type，33
  operators (见 operators, logical)
box (data type)，35
BSD/OS， ，

C

C++，41
case，73
character strings
  concatenation，46
  constants，2
  data types，24
  length，46，48
cidr，37
circle，36
client authentication，36
cluster，5
column，5
columns
  system columns，5
col_description，75
comments
  in SQL，5
comparison
  operators，40
concurrency，106
conditionals，73
configure，3，326
constants，2
COPY，6
  with libpq，19
count，10
CREATE TABLE，5
createdb，1
currval，72

D

data types，20
  constants，3
  extending，197
  numeric，21
  type casts，8
database，54
  creating，1
date
  constants，31
  current，66
  data type，28
  output format，31
    (参见 Formatting)
decimal (见 numeric)
DELETE，11
Digital UNIX (见 Tru64 UNIX)
dirty reads，106
disk space，59
DISTINCT，7
distinct，17
double precision，21
DROP TABLE，6
duplicate，7
dynamic_library_path， ，182

E

elog， ，
embedded SQL
  in C，70
environment variables，21
error message，10
escaping binary strings，12
escaping strings，12
except，18
exists，77

F

false，33
FETCH
  embedded SQL，
flex，1
float4 (见 real)
float8 (见 double precision)
floating point，21
  constants，3
foreign key，13
formatting，55
FreeBSD，18， ，
fsync，
function，177
  internal，181
  SQL，177
functions，40

G

GEQO (见遗传查询优化)
group，15
GROUP BY，10

H

hash (见 index)
has_table_privilege，75
HAVING，10
hierarchical database，5
HP-UX， ，

I

ident，42
identifiers，1
in，77
index，95
  B-tree，95
  hash，96
  multicolumn，96
  R-tree，96
inet (data type)，37
inheritance，15
initlocation，56
input function，197
INSERT，6
installation，1
  on Windows，1，15
int2 (见 smallint)
int4 (见 integer)
int8 (见 bigint)
integer，21
intersection，18
interval，30
iODBC，79
IRIX，
IS NULL，
isolation levels，106
  read committed，106
  read serializable，107

J

joins，12
  cross，
  left，
  natural，12
  outer，

K

Kerberos，41
key words

  list of，128
  syntax，1

L

label
  column，17
  table，13
large object，33
LC_COLLATE，17
ldconfig，9
length
  binary strings (见 binary strings, length)
  character strings (见 character strings, length)
libperl，326
libpgtcl，48
libpq，5
libpq++，41
libpq-fe.h，9
libpq-int.h，9，23
like，52
limit，19
line，35
Linux，18，　，
locale，44
locking，108
log files，62

M

MAC address (见 macaddr)
macaddr (data type)，38
make，1
MANPATH，10
  (参见 man pages)
max，10
MD5，40
min，10
multibyte，46

N

NetBSD，18，　，
network
  addresses，37
nextval，72
non-repeatable reads，106
nonblocking connection，7，15
not
  operator，40
not in，77
notice processor，21
NOTIFY，18
notify，58
nullif，74
numeric (data type)，21

O

object-oriented database，5
obj_description，75
ODBC，79
odbc.sql，80
offset
  with query results，19
OID，5
OpenBSD，18，　，
OpenSSL，
  (参见 SSL)
operators，40
  logical，40
  precedence，9
  syntax，4
or
  operator，40
Oracle，74，308
ORDER BY，7，45
output function，197
overloading，193

P

password，40
PATH，9
path (data type)，35
Perl，326
PGDATA，16
PGDATABASE，22
PGHOST，22
PGPASSWORD，22
PGPORT，22
pgtcl
  closing，61
  connecting，50，51，52，53，54，56
  creating，59
  delete，66
  export，68
  import，67
  notify，58
  opening，60
  positioning，64，65
  reading，62
  writing，63
PGUSER，22
pg_config，23，191
pg_conndefaults，52
pg_connect，50，51，53，54，56
pg_ctl，17
pg_dumpall，2
pg_get_indexdef，75
pg_get_ruledef，75
pg_get_userbyid，75

pg_get_viewdef，75
pg_hba.conf，36
pg_ident.conf，42
pg_lo_close，61
pg_lo_creat，59
pg_lo_export，68
pg_lo_import，67
pg_lo_lseek，64
pg_lo_open，60
pg_lo_read，62
pg_lo_tell，65
pg_lo_unlink，66
pg_lo_write，63
phantom reads，106
PIC，191
PL/Perl，326
PL/pgSQL，288
PL/Python，330
PL/SQL，308
PL/Tcl，320
point，34
polygon，36
postgres 用户，16
postmaster，1，17
ps
  to monitor activity，67
psql，3
Python，330

Q

query，7
quotation marks
  escaping，2

R

R-tree (见 index)
range table，
readline，1
real，21
referential integrity，13
regression test，8
regular expressions，53
  (参见 pattern matching)
relation，5
relational database，5
row，5
rules，206
  and views，208

S

SCO OpenServer，
SELECT，7
select
  select list，16
sequences，72
  and serial type，23
serial，23
serial4，23
serial8，23
SETOF，177
  (参见 function)
setval，72
shared libraries，9
SHMMAX，30
SIGHUP，20，38，42
sliced bread (见 TOAST)
smallint，21
Solaris，18，　，
some，77
sorting
  query results，18
SPI
  修改元组，272
  分配空间，274，275，276，277，278，279
  复制元组，269，271
  复制元组描述符，270
  执行，249
  断开连接，248
  游标，254，255，256，257，258
  解码元组，261，262，263，264，265，266，267
  连接，247，251，252，259
SPI_connect，247
SPI_copytuple，269
SPI_copytupledesc，270
SPI_copytupleintoslot，271
SPI_cursor_close，258
SPI_cursor_fetch，256
SPI_cursor_find，255
SPI_cursor_move，257
SPI_cursor_open，254
SPI_exec，249
SPI_execp，252
SPI_finish，248
SPI_fname，262
SPI_fnumber，261
SPI_freeplan，279
SPI_freetuple，277
SPI_freetuptable，278
SPI_getbinval，264
SPI_getrelname，267
SPI_gettype，265
SPI_gettypeid，266
SPI_getvalue，263
spi_lastoid，
SPI_modifytuple，272
SPI_palloc，274

SPI_pfree，276
SPI_prepare，251
SPI_repalloc，275
SPI_saveplan，259
ssh，35
SSL， ，34，10
standard deviation，77
statistics，67
strings (见 character strings)
subqueries，13，77
subquery，10
substring，47，51
sum，10
superuser，3
syntax
  SQL，1

T

table，5
Tcl，48，320
TCP/IP，17
text (见 character strings)
threads
  with libpq，22
time
  constants，31
  current，66
  data type，29，29
  output format，31
    (参见 Formatting)
time with time zone
  data type，29
time without time zone
  time，29
time zones，32，122
timestamp
  data type，30
timestamp without time zone
  data type，30
TOAST，33
  and user-defined types，198
transaction ID
  wraparound，60
transactions，14
triggers
  in PL/Tcl，324
Tru64 UNIX，
true，33
types (见 data types)

U

Unicode，50
union，18
unixODBC，79
UnixWare， ，
UPDATE，11
upgrading，2，66
user
  current，75

V

vacuum，59
variance，77
version，3，75
view，13
views
  updating，214

W

where，14

Y

yacc，1