

自由软件丛书

PostgreSQL

# 对象关系数据库开发

彭晓明 等 编著

人民邮电出版社

## 内 容 提 要

本书全面介绍了自由的对象关系数据库管理系统 PostgreSQL 的安装、配置、管理、使用 and 开发技巧,以及基于 PostgreSQL 的 Web 数据库应用开发技术、ODBC 应用开发技术等。最后还给出了 PostgreSQL 命令、PostgreSQL 应用程序及工具的详细参考资料。

本书内容翔实、可操作性强,是一本 PostgreSQL 使用、管理和开发应用手册,适合软件开发人员和 UNIX/Linux 爱好者阅读。对于大专院校计算机软件专业的学生来说,也是一本很好的数据库技术类参考书。

图数 ( ) 书目P版编 I  
PostgreSQL 对象关系数据库开发/彭晓明编著.—北京：人民邮电出版社，2001.6  
(自由软件丛书)  
ISBN 7-115-09325-3  
I.P... II.彭... III.关系数据库—数据库管理系统，PostgreSQL IV.TP311.138  
中国版本图书馆 CIP 数据核字 (2001) 第 027135 号

自由软件丛书

### PostgreSQL 对象关系数据库开发

---

- ◆ 编 著 彭晓明 等  
责任编辑 张瑞喜
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号  
邮编 100061 电子函件 315@pptph.com.cn  
网址 <http://www.pptph.com.cn>  
北京汉魂图文设计有限公司制作  
北京 厂印刷  
新华书店总店北京发行所经销
- ◆ 开本：787×1092 1/16  
印张：  
字数：676 千字 2001 年 6 月第 1 版  
印数：1 - 0 000 册 2001 年 6 月北京第 1 次印刷

ISBN 7-115-09325- /TP

---

定价：00.00 元

## 丛书前言

自由软件的出现，改变了传统的以公司为主体的封闭式的软件开发模式。自由软件采用了开放和协作的开发模式，无偿提供源代码，允许任何人取得、修改和重新发布自由软件的源代码。这种开发模式激发了世界各地的软件开发人员的积极性和创造热情。大量软件开发人员投入到自由软件的开发中。软件开发人员的集体智慧得到充分发挥，大大减少了不必要的重复劳动，并使自由软件的脆弱点能够及时被发现和克服。任何一家公司都不可能投入如此强大的人力去开发和检验商品化软件。这种开发模式使自由软件具有强大的生命力。

就目前我国计算机软件状况而言，系统软件和大部分应用软件平台基本上被国外软件公司所垄断，民族软件产业的发展面临着极大的困难。自由软件无保留地提供源代码，使我们可以在高起点起步，非常有利于打破垄断；有利于我国软件行业在较短的时间内彻底改变目前被动的局面。自由软件的免费使用和自由传播的特性十分适合于我国目前的经济状况。

为适应这样一种形势，我们组织编写了这套自由软件丛书。丛书以介绍最新自由软件的技术和使用技巧为主。自由软件的缺点是缺乏开发公司的技术支持，文档通常也不齐全，给软件的使用带来了较大的不便，本丛书正好弥补这方面的不足。丛书不仅仅涉及 Linux 操作系统，而且还涉及与之相关的网络服务器、数据库、多媒体等众多自由软件以及自由软件开发技术。力图较为全面地介绍和讲解自由软件的相关技术。

希望这套丛书的出版，能够推进自由软件在我国的发展进程，能够给广大的软件工作者的学习和研究带来一定的帮助。同时热切期待广大读者对丛书提出宝贵意见，也欢迎读者参与丛书的编写，让我们共同努力，为自由软件在我国生根、开花、结果做出贡献。

自由软件丛书 编委会

# 前言

PostgreSQL 是一种自由的对象关系型数据库管理系统，经过十几年的发展，它已成为先进的开放源代码的数据库管理系统，被广泛地应用于信息处理的各个领域。

PostgreSQL 被誉为是数据库新技术的“探路者”，商业数据库管理系统的许多技术都源自这个系统。它提供了多版本并行控制，支持几乎所有 SQL 标准特性（包括子查询、事务、用户定义类型和函数），并且有非常广泛的开发语言支持（包括 C、C++、Java、perl、tcl 及 python 等）。在稳定、高效、安全的核心的支持下，PostgreSQL 能够满足大多数数据库应用系统的需要。随着自由软件的发展和普及，PostgreSQL 将在我国的信息化建设中扮演越来越重要的角色。

本书不仅介绍了 PostgreSQL 的基本原理和使用技巧，而且还讲述了基于 PostgreSQL 的应用系统的开发技术。书中详细介绍了 PostgreSQL 安装和配置的方法和技巧、PostgreSQL 的基本使用技术、扩展 PostgreSQL 功能的各种方法、基于 PostgreSQL 的 Web 数据库应用系统以及 ODBC 应用系统的开发方法，并详细地分析了一个著名的 PostgreSQL Web 管理工具。本书可操作性强，结合本书介绍的操作步骤同步实践，能够很轻松地学习和使用这个系统。本书的配套光盘收录了本书的所有例子，并附有许多有价值的自由软件和参考资料。

本书的主要作者有彭晓明、刘庆华、程新明、秦清、余科见、吴琪等，其中彭晓明编写大部分章节并负责全书统稿，其他作者编写了部分章节。此外，余海涵、王坚也做了许多实质性的工作。

在此向所有关心、支持和帮助过本书编写的领导、朋友和家人表示诚挚的谢意！

由于作者的能力和水平有限，加上时间仓促，本书可能会有不妥和错误之处，如能给予指正将不胜感谢！

作者的 E-mail 地址：pengxm@263.net

作者



# 目 录

第 1 章 绪论.....	1
1.1 PostgreSQL 的历史.....	2
1.1.1 起源.....	2
1.1.2 PostgreSQL 全球开发小组.....	3
1.2 版权.....	4
1.3 PostgreSQL 的特点.....	5
1.4 运行平台.....	6
1.5 相关资源.....	7
第 2 章 系统安装.....	9
2.1 一般安装.....	10
2.2 rpm 安装.....	13
2.2.1 RedHat 6.x 下的安装.....	13
2.2.2 RedHat 7.x 下的安装.....	15
2.3 主要文件.....	17
第 3 章 基本原理.....	19
3.1 系统组成.....	20
3.2 后端服务器.....	21
3.3 查询处理过程.....	22
3.3.1 解析查询请求.....	22
3.3.2 重写查询树.....	22
3.3.3 规划优化查询树.....	23
3.3.4 执行查询.....	24
3.4 规则系统.....	25





3.4.1 查询树.....	25
3.4.2 规则系统和视图.....	26
3.4.3 视图的能力.....	31
3.4.4 规则和权限.....	32
3.4.5 规则与触发器的比较.....	32
 第 4 章 用户与数据库.....	 35
4.1 安全认证.....	36
4.1.1 安全认证.....	36
4.1.2 用户认证.....	37
4.2 PostgreSQL 用户.....	39
4.2.1 创建用户.....	39
4.2.2 用户组.....	41
4.2.3 删除用户.....	42
4.3 数据库.....	43
4.3.1 创建数据库.....	43
4.3.2 删除数据库.....	44
 第 5 章 psql 基本操作.....	 45
5.1 psql.....	46
5.1.1 简介.....	46
5.1.2 语法.....	46
5.1.3 两种执行方式.....	47
5.1.4 专有命令.....	47
5.2 启动数据库会话.....	49
5.2.1 连接数据库.....	49
5.2.2 与数据库交互.....	49
5.3 命令缓冲区.....	50
5.3.1 命令的输入.....	50
5.3.2 显示缓冲区.....	51
5.3.3 编辑缓冲区.....	51
5.3.4 清除缓冲区.....	52
5.4 联机帮助.....	52



5.4.1 专有命令.....	52
5.4.2 SQL 命令.....	52
<b>第 6 章 基本 SQL 命令.....</b>	<b>55</b>
6.1 SQL 语法.....	56
6.1.1 关键字.....	56
6.1.2 注释.....	60
6.1.3 名字.....	61
6.1.4 常量.....	61
6.2 创建表.....	62
6.3 插入数据.....	64
6.4 查询数据.....	65
6.5 控制查询输出格式.....	68
6.6 删除数据.....	74
6.7 修改数据.....	75
6.8 删除表.....	76
<b>第 7 章 数据类型.....</b>	<b>77</b>
7.1 基本数据类型.....	78
7.1.1 字符串类型.....	78
7.1.2 数值类型.....	78
7.1.3 时间类型.....	80
7.1.4 逻辑类型.....	81
7.1.5 几何类型.....	81
7.1.6 网络类型.....	82
7.2 数组类型.....	84
7.3 大对象类型.....	87
7.4 数据类型转换.....	88
7.5 预定义变量.....	89



第 8 章 函数	91
8.1 SQL 函数	92
8.2 数学函数	92
8.3 字符串函数	93
8.4 日期时间函数	93
8.5 格式化函数	94
8.6 几何函数	97
8.7 网络函数	98
第 9 章 运算符和表达式	99
9.1 运算符简介	100
9.1.1 种类	100
9.1.2 优先级	101
9.2 通用运算符	101
9.3 数值运算符	102
9.4 几何运算符	103
9.5 时间间隔运算符	103
9.6 网络运算符	104
9.7 表达式	104
9.7.1 常量表达式	105
9.7.2 字段表达式	105
9.7.3 函数表达式	105
9.7.4 聚集表达式	106
9.7.5 复合表达式	106
9.7.6 目标列表表达式	107
9.7.7 FROM 列表表达式	108



<b>第 10 章 复杂查询</b> .....	109
10.1 更灵活的数据插入.....	110
10.1.1 空值.....	110
10.1.2 默认值.....	113
10.1.3 利用其他表插入数据.....	114
10.2 复杂条件查询.....	115
10.3 规则表达式查询.....	116
10.3.1 规则表达式简介.....	117
10.3.2 在查询中的应用.....	117
10.4 CASE 子句.....	118
10.5 控制查询结果.....	120
10.5.1 删除重复行.....	120
10.5.2 限制行数.....	121
10.5.3 游标.....	122
10.6 聚集查询.....	124
10.6.1 聚集函数.....	124
10.6.2 GROUP BY.....	127
10.6.3 HAVING.....	129
<b>第 11 章 连接查询</b> .....	131
11.1 表标识和字段引用.....	132
11.2 表的连接.....	133
11.3 复杂连接查询.....	138
<b>第 12 章 集合查询</b> .....	143
12.1 集合运算简介.....	144
12.1.1 表集合运算.....	144
12.1.2 字段集合运算.....	145
12.2 表集合查询.....	146



12.2.1 UNION 查询.....	146
12.2.2 INTERSECT 查询.....	147
12.2.3 EXCEPT 查询.....	148
12.3 字段集合查询.....	150
12.3.1 ANY 查询.....	150
12.3.2 ALL 查询.....	150
12.3.3 EXISTS 查询.....	152
 第 13 章 唯一性编号.....	 153
13.1 对象标识编号.....	154
13.1.1 关于对象标识编号.....	154
13.1.2 对象标识编号的用途.....	155
13.1.3 对象标识编号的局限性.....	155
13.2 序列.....	156
13.2.1 关于序列.....	156
13.2.2 序列的创建.....	156
13.2.3 序列在连续唯一性行号中的应用.....	158
13.3 串列类型.....	159
 第 14 章 提高效率.....	 161
14.1 索引.....	162
14.1.1 关于索引.....	162
14.1.2 索引的使用.....	163
14.1.3 唯一性索引.....	165
14.2 集簇.....	166
14.3 表的清理.....	167
14.4 查询分析.....	168
 第 15 章 并发控制.....	 171
15.1 多版本并发控制.....	172



15.1.1 事务简介.....	172
15.1.2 多版本并发控制.....	172
15.1.3 事务隔离.....	172
15.2 锁定.....	174
15.2.1 表级锁.....	174
15.2.2 行级锁.....	175
15.2.3 索引与锁.....	175
15.2.4 数据完整性检查.....	175
15.3 事务的操作.....	176
15.4 事务的可见性.....	178
15.4.1 读提交.....	178
15.4.2 可串行化.....	179
15.5 锁的使用.....	181
 第 16 章 表的维护.....	 183
16.1 修改表结构.....	184
16.2 访问权限.....	186
16.3 继承.....	188
16.4 视图.....	191
16.5 规则.....	192
16.5.1 INSTEAD 规则.....	193
16.5.2 DO 规则.....	193
16.5.3 视图的更新.....	195
16.6 临时表.....	198
16.7 消息交换.....	199
16.8 数据的导入和导出.....	199
 第 17 章 约束.....	 201
17.1 非空值约束.....	202



17.2 唯一性约束.....	203
17.3 主键约束.....	204
17.4 外键约束.....	205
17.4.1 一般使用.....	206
17.4.2 主键的更新.....	207
17.4.3 空值问题.....	209
17.4.4 外键检查时机.....	211
17.5 数据检验.....	212
<b>第 18 章 编程接口.....</b>	<b>215</b>
18.1 编程接口简介.....	216
18.2 LIBPQ 接口.....	217
18.3 LIBPGEASY 接口.....	219
18.4 ECPG 接口.....	220
18.5 LIBPQ++接口.....	222
18.6 编译程序.....	223
18.7 程序变量.....	224
18.8 ODBC 与 JDBC 接口.....	224
18.9 脚本语言接口.....	226
18.9.1 Perl.....	226
18.9.2 TCL/TK(PGTCLSH/PGTKSH) .....	227
18.9.3 Python.....	228
18.9.4 PHP.....	228
18.9.5 脚本语言的安装.....	230
<b>第 19 章 自定义函数.....</b>	<b>231</b>
19.1 简介.....	232
19.2 SQL 函数.....	232



19.2.1 命令.....	232
19.2.2 例子.....	233
19.3 C 语言函数.....	238
19.3.1 编写 C 语言源代码.....	238
19.3.2 编译 C 语言源代码.....	239
19.3.3 注册新函数.....	239
<b>第 20 章 PL/pgSQL 语言.....</b>	<b>241</b>
20.1 语言简介.....	242
20.2 语言元素.....	243
20.2.1 程序结构.....	243
20.2.2 变量的定义.....	244
20.2.3 类型与表达式.....	244
20.2.4 语句.....	246
20.2.5 例外处理.....	248
20.3 PL/pgSQL 函数.....	248
20.4 触发器.....	254
20.4.1 触发器函数.....	254
20.4.2 触发器的创建.....	255
20.4.3 触发器的使用.....	255
<b>第 21 章 Web 数据库应用.....</b>	<b>257</b>
21.1 环境安装与配置.....	258
21.1.1 Apache 服务器.....	258
21.1.2 PHP 脚本语言.....	259
21.1.3 配置.....	260
21.2 数据库访问函数.....	261
21.3 数据库操作.....	269
21.3.1 连接数据库服务器.....	269
21.3.2 查询数据.....	271
21.3.3 插入数据.....	272



第 22 章	phpPgAdmin.....	275
22.1	安装与配置.....	276
22.1.1	简介.....	276
22.1.2	安装.....	276
22.1.3	配置.....	278
22.1.4	用户认证.....	282
22.2	使用方法.....	284
22.2.1	启动.....	284
22.2.2	基本操作.....	284
22.3	关键技术分析.....	286
22.3.1	用户认证.....	286
22.3.2	库和表的列表.....	288
22.3.3	表的维护.....	289
22.3.4	用户的管理.....	292
第 23 章	ODBC 应用.....	293
23.1	ODBC 的获取与安装.....	294
23.2	ODBC 的配置.....	295
23.3	注意事项.....	298
23.4	ODBC 的使用.....	299
第 24 章	命令与工具.....	301
24.1	SQL 命令.....	302
24.2	系统程序和工具.....	381
附录	常见问题解答.....	419
	配套光盘说明.....	434

## 第1章

# 绪论

---

PostgreSQL起源于美国加州大学伯克利分校的Postgres项目。经过十多年的发展，PostgreSQL已成为一个具有面向对象特性、高性能、自由的数据库管理系统。本章介绍了PostgreSQL的有关情况，主要包括：

PostgreSQL 的历史

PostgreSQL 的特点

相关因特网资源



## 1.1 PostgreSQL 的历史

### 1.1.1 起源

PostgreSQL 对象关系型数据库管理系统是由美国加州大学伯克利分校的 Postgres 软件包演变而来的。经过十几年的发展, PostgreSQL 已成为世界上可自由获得的、先进的开放源代码数据库管理系统。它提供了多版本并行控制, 支持几乎所有 SQL 标准特性(包括子查询、事务和用户定义类型和函数), 并且有非常广泛的开发语言支持(包括 C、C++、Java、perl、tcl 及 python 等)。

#### 1. 伯克利的 Postgres 项目

Postgres 项目始于 1986 年, 经历了几次主要的版本更新。第一个演示性的系统发表于 1987 年, 并且在 1988 年的 ACM-SIGMOD 大会上展出。这个版本没有对外发布, 主要由开发小组内部使用。1989 年的发布版本 1 提供给一些外部用户使用, 并陆续收集了一些用户的反馈意见。作为对第一个规则系统批评的响应, 开发者重新设计了规则系统, 并在 1990 年 6 月发布了使用新规则系统的版本 2。版本 3 于 1991 年发表, 这个版本增加了对多存储管理器的支持, 并且改进了查询执行器, 重新编写了重写规则系统。从此以后, 直到 Postgres95 版本发布前, 所有的开发工作几乎都集中在移植性和可靠性的改进上。

Postgres 被用于许多研究或实际的应用之中, 包括财务数据分析系统、喷气引擎性能监控软件包、小行星跟踪数据库、医疗信息数据库和一些地理信息系统等。Postgres 还被许多大学用作数据库教学实例。最后, Illustra Information Technologies 公司(后来并入 Informix 公司)取得了 Postgres 的源代码并使之商业化。在 1992 年末, Postgres 成为 Sequoia 2000 科学计算项目的首要数据管理器。

到了 1993 年, 随着外部用户数量的急剧增加, 源代码的维护工作量也变得越来越大了。为了集中精力进行数据库新理论和新技术的研究, Postgres 开发小组决定停止开发工作, 于是, 该项目在版本 4.2 时正式终止。

#### 2. Postgres95

1994 年, Andrew Yu 和 Jolly Chen 为 Postgres 增加了 SQL 语言解释器, 更名为 Postgres95, 并将 Postgres95 源代码放在因特网上供全世界的用户下载、研究和使用, 使之成为一个开放源代码的数据库管理系统。

Postgres95 所有源代码都完全符合 ANSI C 规范, 比 Postgres 的代码量减少了 25%, 并且进行了许多内部修改以便提高性能和代码的维护性。Postgres95 v1.0.x 在进行 Wisconsin

Benchmark 测试时比 Postgres v4.2 快 30%~50%。除了修正了一些错误，Postgres95 的主要改进包括：

- (1) 原来的查询语言 Postquel 被 SQL 取代（在服务器端实现）。
- (2) 以用户自定义 SQL 函数的方式来实现 SQL 子查询；重新实现了聚集；增加了对“GROUP BY”SQL 特性的支持。
- (3) 在监控程序方面，增加了利用 GNU readline 进行交互查询的监控程序（psql）。
- (4) 增加了新的前端库 libpgtcl，用以支持以 Tcl 为基础的客户端。
- (5) 彻底重写了大对象的接口，保留了将大对象转置（Inversion）作为存储大对象的唯一机制。
- (6) 去掉了实例级（instance-level）的规则系统。

除了上述改进之外，在发布的源代码中，还增加了一个简单常用的 SQL 和 Postgres95 特有 SQL 的教程。

为了进一步向开放源代码组织靠拢，源代码的编译工具改为 GNU make，取代了原来的 BSD make。

### 3. PostgreSQL

到了 1996 年，开发小组认为 Postgres95 这个名字经不起时间的考验，于是他们便选择了一个新名字——PostgreSQL。PostgreSQL 这个名字隐含了该数据库管理系统与伯克利分校的关系以及 SQL 特性，因此受到了人们的认可和普遍欢迎，同时版本号的编排也重新从 6.0 开始。

PostgreSQL 主要的改进有：

- (1) 多版本并行控制机制取代表级锁定机制，这样就允许读操作在写操作活跃时连续地读取一致的数据，并且使数据库在等待执行查询时用 pg\_dump 进行热备份成为可能。
- (2) 实现了许多重要的后端特性，包括子查询、默认值、约束和触发器等。
- (3) 增加了附加的 SQL92 兼容语言特性，包括主键、引用标识、强制的语义字符串类型转换、类型转换以及二进制和十六进制整数的输入。
- (4) 改进了内建的数据类型，包括对新的大范围日期时间类型和几何数据类型的支持。

经过改进后，后端代码的速度提高了大约 20%-40%，而且自 v6.0 起，后端的启动时间缩短了 80%。

#### 1.1.2 PostgreSQL 全球开发小组

早期的 Postgres 开发者的主要工作是整理老的邮件列表、分析和评估其他 Postgres 爱好者提供的补丁程序。这时的系统代码还相当脆弱，并且不易被理解。经过一段时间的整理和开发后，专家们仍然感觉到问题重重，简单的补丁可能导致整个系统的崩溃，并且难以找到系统崩溃的原因；用户反馈的各种问题使专家们绞尽脑汁，百思而不得其解，处理这些问题需要大量的精力和时间。于是他们意识到急需一个专门的组织来进行 Postgres 的进一步开发，这个组织就是后来的“PostgreSQL 全球开发小组”（PGDT，PostgreSQL Global Development Team）。

PGDT 开始以 CVS 方式维护 PostgreSQL 源代码。CVS 是一种分布式软件开发管理工具，允许参与某个软件项目的开发者保持最新的源代码副本而不必重复下载全部源代码。

与此同时，PGDT 也开始以每隔三到五个月的时间发布一个新的 PostgreSQL 发行版。在每一个新版本的发行期限中，大约有两到三个月的时间用于开发；一个月的时间用于 beta 测试，几周的时间用于修正一些严重错误。与文字处理、游戏等软件不同，数据库软件是一种多用户系统，许多用户数据存放在数据库中，软件可靠性是至关重要的。因此，PGDT 在新版本的推出上没有一个固定、严格的时间表，软件的质量和可靠性始终被摆在第一位。

许多软件爱好者对 PostgreSQL 的开发表现出了极大的兴趣，然而数据库的工作原理和 PostgreSQL 的设计细节和技术并不是在短时间内就能够掌握的。为了帮助开发者缩短学习时间，PGDT 推出了一个名为“TODO”的列表文档，以帮助开发者快速掌握 PostgreSQL 的开发技巧。除此之外，PGDT 还编写了开发者 FAQ(Developer's FAQ)文档，对 PostgreSQL 开发者提出的各种问题给出了详细的解答。在这些文档的帮助下，开发者在修正错误和为 PostgreSQL 增加新功能等方面的工作有了更高的效率，同时为 PostgreSQL 开发者队伍的发展和壮大奠定了坚实的基础。

虽然从 Postgres 继承下来的源代码的模块化工作做得非常好，但由于大多数代码是用于新技术的研究和测试的，因此这些代码的风格与后来新增加的代码的风格相差甚远。于是 PGDT 编写了一个工具，对全部源代码的风格进行了整理，使它们一致起来。除此之外，他们还编写了一个脚本程序，该程序能够自动找出所有带有“static”性质的函数，还能找出所有无用的函数。这些工具优化了 PostgreSQL 的源代码风格，提高了源代码的质量。

正是由于 PGDT 卓有成效的努力，PostgreSQL 得到了飞速发展，在 Internet 的 Web 数据库方面得到了广泛的应用。今天，每一个 PostgreSQL 新版本都是对前一个版本的重大改进。PGDT 已经完全掌握了从伯克利 Postgres 继承的代码，并且就 PostgreSQL 的每一个模块而言，至少有一名开发小组成员掌握了其细节。PGDT 成员的规模在不断地扩大，成员的经验水平也在不断地提高，因此，有理由相信 PostgreSQL 前途无量。

## 1.2 版权

PostgreSQL 是一个自由的数据库管理系统，但它的版权与 Linux 所遵循的 GPL 协议有所不同，以下是 PostgreSQL 版权声明的中文翻译，原文请参考 PostgreSQL 发行软件包中的 COPYRIGHT 文件。

PostgreSQL 数据库管理系统（以前称为 Postgres 和 Postgre95）

部分版权（c）1996-2000，PostgreSQL，Inc 部分版权（c）1994-6 加州大学董事

允许以任何目的使用、复制、修改和分发这个软件和它的文档而不收取任何费用，并且无须签署因此而产生的证明，前提是上面的版权声明和本段以及下面两段文字出现在所有的复制软件包中。

在任何情况下，加州大学都不承担因使用本软件及其文档而导致的对任何当事人的直接

的、间接的、特殊的、附加的或者伴随的损坏包括利益损失的责任，即使加州大学已经提示了这些损失的可能性时也是如此。

加州大学明确放弃任何保证，包括但不限于某一特定用途的商业和利益的隐含保证。这里提供的这份软件是基于“AS IS”的，因而加州大学没有责任提供维护、支持、更新、增强或者修改的服务。

## 1.3 PostgreSQL 的特点

虽然PostgreSQL是一种自由的数据库管理系统软件，但其有着自己鲜明的特色。

### 1. 面向对象

PostgreSQL包含一些面向对象的技术特性，主要包括类和继承。例如，用户在创建一个表的时候，不必从头开始，只需从一个已存在的表中继承一些属性，然后再加入一些新的属性即可。

### 2. 数据类型丰富

PostgreSQL不仅支持传统的数字、字符、日期的数据类型，同时还支持一些特殊的数据类型，如点、线、多边形等几何数据类型。除此之外，还允许用户自定义数据类型。

### 3. 全面支持SQL

在所有的自由数据库管理系统中，PostgreSQL对SQL语言的支持是最全面的，不仅全面支持SQL89，而且支持SQL92的大部分特性。

### 4. 与Web的集成

通过PHP3或Perl等脚本语言，PostgreSQL能够与Web服务器（如Apache）紧密集成在一起，为用户提供一种免费的高性能Web解决方案。另外，利用ODBC，PostgreSQL也支持传统的客户/服务器应用。

### 5. 大数据库

当数据库的规模超过2GB的时候，有些数据库产品在微机上的运行性能将急剧下降，而PostgreSQL则能保持良好的性能，甚至当数据库的规模达到100GB时，PostgreSQL仍然能够高效率地运行。

### 6. 2000年兼容

截止目前为止，尚未见有关PostgreSQL存在2000年兼容问题的报道。下面是开发者关于PostgreSQL2000年兼容问题的论述。

PostgreSQL 全球开发小组将提供 Postgres 软件的代码树作为一种公众服务，对其特性和性能不做任何保证和承诺，但是，到目前为止：

(1) 作为一个从 1996 年 11 月开始从事 Postgres 支持的志愿者，并未发现任何 Postgres 的代码与 2000 年 1 月 1 日的时间切换(Y2K)相关。

(2) 本声明的作者并未收到当前或最近版本的 Postgres 任何与 2000 年问题相关的报告，不论是在进行测试还是在其他领域的使用中。考虑到装机量和邮件列表的活跃性，如果问题存在，作者应该可以获得消息。

(3) 据作者所知，Postgres 对两位数年份的一些假设在用户手册中的日期类型章节中能够找到。就两位数年份而言，关键的切换年份是 1970 年，而不是 2000 年。例如，“70-01-01”被看作是 1970 年 1 月 1 日，而“69-01-01”则被看作是 2069 年 1 月 1 日。

当然，任何因操作系统取“当前时间”命令而造成的 2000 问题都可能传染到 Postgres。

PostgreSQL 具有许多超前的技术特性，一些商业化的数据库产品也借鉴了 PostgreSQL 的技术。1999 年，Linux World 对在 Linux 上数据库产品进行了综合测试后，授予 PostgreSQL 该年度的“最佳数据库产品”称号。



图1-1 1999年度Linux World编辑选择奖

在2000年，PostgreSQL又荣获Linux杂志编辑选择“最佳数据库”奖。



图1-2 2000年度Linux Journal编辑选择奖

## 1.4 运行平台

PostgreSQL主要运行在各种UNIX平台（见表1-1）上，并且都有良好的运行记录。

在 PostgreSQL 发行软件包中有一个名为 win31.mak 的文件，它用于制作 Win32 格式的 libpq 库和 psql 程序。

libpq 库、psql 程序以及其他接口和文件可以编译成 Win32 二进制代码。这种情况下，客户端在 MS Windows 上运行，通过 TCP/IP 与一个运行在 UNIX 平台上的服务器进行通信。

数据库服务器现在可以通过使用 Cygnus Unix/NT 移植库在 Windows NT 上运行。

<http://www.freebsd.org/~kevlo/postgres/portNT.html> 和 <http://surya.wipro.com/uwin/ported.html>中讨论了将PostgreSQL移植到Windows平台的方法。

表 1-1

PostgreSQL 运行平台

运行平台	说明
aix	IBM on AIX 3.2.5 或 4.x 版
alpha	DEC Alpha AXP on Digital Unix 2.0、3.2、4.0 版
BSD44_derived	由 4.4-lite BSD 发展而来的操作系统，包括 NetBSD、FreeBSD
bsdi	BSD/OS 2.x、3.x、4.x 版
dgux	DG/UX 5.4R4.11
hpux	HP PA-RISC on HP-UX 9.*、10.*
i386_solaris	i386 Solaris
irix5	SGI MIPS on IRIX 5.3
linux	Intel i86 Alpha SPARC PPC M68k
sco	SCO 3.2v5 Unixware
sparc_solaris	SUN SPARC on Solaris 2.4、2.5、2.5.1 版
sunos4	SUN SPARC on SunOS 4.1.3
svr4	Intel x86 on Intel SVR4 and MIPS
ultrix4	DEC MIPS on Ultrix 4.4

本书主要介绍 Linux 平台上的 PostgreSQL。

## 1.5 相关资源

1 . <http://www.postgresql.org>

这是PostgreSQL的官方站点，提供各种版本的PostgreSQL下载服务，包括源代码、二进制运行代码、相关接口、工具以及文档。除此之外，本站点还为PostgreSQL的开发者提供各种技术服务。

2 . <http://www.pgsql.com>

PostgreSQL Inc公司率先为PostgreSQL提供商业化的技术支持，这是该公司的技术支持站点。该公司主要以CD-ROM的方式为用户提供PostgreSQL软件和各种应用解决方案，为PostgreSQL的开发、应用和普及提供了便利。

3 . <http://www.greatbridge.com>

这是另外一个为PostgreSQL提供技术支持的公司(GreatBridge)站点。该公司为PostgreSQL制作了一个更容易安装和使用的发行版本，并以CD-ROM的方式对外出售。除此之外，为满足数据库用户的不同需要，该公司还开发了一组具有专业水准的技术支持软件包，并编写了更加通俗易懂的操作和管理手册。

4. 其他有用的站点

<http://postgres.cybertec.at>——提供各种培训课程、技术支持和应用解决方案。

<http://www.srainc.com>——提供简单的技术支持。



<http://www.flex.ro/pgaccess>——PgAccess是具有图形界面 (Tcl/Tk) 的PostgreSQL管理工具, 可以在这个站点上查询到该工具的最新动态。

<http://www.troubador.com/~keidav/index.html>——本站点提供MPSQL软件。MPSQL是一个窗口式的PostgreSQL查询工具, 它为使用者提供了一个图形化的SQL控制界面。MPSQL与Oracle的SQL Worksheet或微软SQL Server的查询工具类似, 有一个漂亮的图形化和命令历史记录, 也可以进行剪贴, 除此之外还有其他有助提高效率的功能。

<http://www.mutinybaysoftware.com>——本站点提供PostgreSQL图形化系统管理工具MPMGR。

<http://www.man.ac.uk/~whaley/ag/appgen.html>——本站点提供AppGEN工具。AppGEN是一种高级的第四代程序语言和应用程序生成器(application generator), 可以用来生成基于因特网的程序, 这些程序一般在因特网或公司内联网使用。AppGen程序以符合CGI标准的C脚本文件写成, 大部分Web都可支持它的运行。

<http://www.cis-computer.com/dbengine>——本站点提供一个即插即用的因特网界面工具DBENGINE。

<http://www.insightdist.com/psqlodbc>——提供PostgreSQL ODBC驱动程序的站点。

<http://www.openlinksw.com>——本站点提供另外一种PostgreSQL驱动程序UDBC。UDBC是一种独立于驱动程序管理器 (driver managers)和DLL支持的静态ODBC版本, 用来直接将数据库连接能力嵌入到应用软件中。

<http://www.demon.co.uk/finder/postgres/index.html>——提供PostgreSQL JDBC驱动程序的站点。

<http://www.oswego.edu/Earp>——本站点提供一个PostgreSQL因特网数据库设计/完成工具EARP。

<http://www.druid.net/pygresql>——本站点提供基于Python的PostgreSQL数据库接口PyGreSQL。它嵌入PostgreSQL查询函数库, 使得在Python脚本中使用PostgreSQL变得十分简单。

<http://www.perl.com/CPAN/modules/by-module/DBD/>——这个站点提供Perl数据库接口(DBI)。DBI是Perl语言的一个数据库存取应用程序接口API, 它定义了一组函数、变量和惯例, 以提供一个一致而独立于实际所用的数据库接口。

<http://www.php.net>——本站点提供PostgreSQL的PHP接口函数库。

## 第2章

# 系统安装

---

早期的PostgreSQL的安装工作比较繁琐，但由于RedHat公司的努力，安装工作现在已变得相当简单。本章详细介绍PostgreSQL的安装方法，主要包括：

- 一般安装方法

- rpm 安装方法

- 目录结构



### 2.1 一般安装

PostgreSQL 可以安装在多种 UNIX 平台上，基本方法是下载源代码、配置源代码、编译源代码、安装二进制文件和库。

#### 1. 下载源代码

可以从 <ftp.postgresql.org> 站点下载最新版本的 PostgreSQL 源代码，文件名通常为 `postgresql-x.x.tar.gz`，其中 `x.x` 为版本号。

接下来建立源程序目录 `/usr/local/src/pgsql` 和安装目录 `/usr/local/pgsql`：

```
#mkdir /usr/local/pgsql
#chown postgres:postgres /usr/local/pgsql
#mkdir /usr/local/src
#mkdir /usr/local/src/pgsql
#chown postgres:postgres /usr/local/src/pgsql
```

下载完毕后，可以使用下面的命令将其解包。

```
#gunzip postgresql-7.0.tar.gz
#tar -xf postgresql-7.0.tar
```

#### 2. 编译前的准备

编译 PostgreSQL 需要 3.75 版以上的 GNU make (可用 `gmake -v` 检查版本号)，2.7.2 版以上的 GNU C (用 `gcc -v` 检查版本号) 以及 bison 和 flex 工具 (通常这两种工具都已经安装)。在 Linux 系统中，GNU make 是默认的编译工具。在其他系统中，GNU make 的名字可能有所不同。可以在 <ftp://ftp.gnu.org> 找到各种版本的 GNU make。

PostgreSQL 所支持的平台信息可以在 <http://www.postgresql.org/docs/admin/ports.htm> 中找到。通常来说，大多数 UNIX 兼容平台都可以运行 PostgreSQL。在发行软件包的 doc 子目录中，有一些与平台相关的 FAQ 和 README，如果碰到编译困难，可资参考。

尽管运行 PostgreSQL 的最小内存需求为 8MB，但建议将内存增加到 96MB。大容量的内存可以显著提高运行性能。

接下来要做的工作是检查是否有足够的磁盘空间。PostgreSQL 默认安装位置为 `/usr/local/pgsql/`，系统文件约需 3~10MB 空间。附带的测试程序在运行时需要约 20MB 空间，所以安装时应注意预留足够的空间，建议 `/usr/local/pgsql/` 目录下保证有 50MB 以上空间。另外展开和编译源程序约需 30~60MB 空间。

检查磁盘空间的命令是：

```
df -k
```

### 3. 创建 PostgreSQL 超级用户

应该为 PostgreSQL 设置一个专用用户名，例如 postgres，作为 PostgreSQL 超级用户。

另外 PostgreSQL 使用了 System V 的共享内存机制。FreeBSD 默认状态不支持该机制。如使用中的内核设置文件中无以下项目，则需追加后重新编译内核：

```
options SYSVSHM
options SYSVSEM
options SYSVMSG
```

以 root、bin 或者其他具有特殊权限的用户身份运行 PostgreSQL 存在安全问题，实际上 postmaster 拒绝以 root 身份运行。

### 4. 配置 PostgreSQL 源代码

在这个阶段，可以为编译过程指定实际安装路径，并选择需要安装的内容。进入 src 子目录并输入：

```
./configure
```

后面可以跟着任何合法的 PostgreSQL 编译选项。对于第一次安装，可以不带选项。如果想得到所有合法的 PostgreSQL 编译选项，可以使用下面的命令：

```
./configure --help
```

以下列出的是最常用的编译选项：

--prefix=BASEDIR——为安装 PostgreSQL 指定一个不同于默认路径的基本路径。默认的基本路径为/usr/local/pgsql。

--enable-locale——使用本地化支持。

--enable-multibyte——允许使用多字节字符编码。这个选项主要用于像日语、韩语或中文等多字节编码语言。

--with-perl——编译 Perl 接口和 plperl 扩展语言。需要注意的是 Perl 接口将安装到 Perl 模块的默认位置( 典型的是/usr/lib/perl )，因此只有具有 root 权限的用户才能使用这个选项。

--with-odbc——编译 ODBC 驱动程序包。

--with-tcl——编译 Tcl/Tk 所需的接口库和程序，包括 libpgtcl、pgtclsh 和 pgtksh。

### 5. 编译

源代码的配置工作完成后，可以使用 gmake 命令开始编译 PostgreSQL。如果编译成功，最后一个信息行应该是 “ All of PostgreSQL is successfully made. Ready to install. ”。

### 6. 安装

如果编译成功，就可以开始安装 PostgreSQL 的可执行文件和库文件。

```
gmake install
```

正常完成后，PostgreSQL 的执行文件和库文件等被安装到/usr/local/pgsql 目录下。接下来可以继续安装附带文档：

```
gmake install-man
```

```
cd /usr/local/src/postgresql-vx.x/doc
```

## 2 系统安装

`make install`

至此，PostgreSQL 的编译安装已经完成。

### 7. 设置

编译、安装完毕后，PostgreSQL 并不能立即运行，需要进行一些必要的设置。

#### (1) 设置环境变量：

如果使用的 shell 为 bash，则在 .bashrc 中添加以下命令：

```
PATH="$PATH":/usr/local/pgsql/bin
export POSTGRES_HOME=/usr/local/pgsql
export PGLIB=$POSTGRES_HOME/lib
export PGDATA=$POSTGRES_HOME/data
export MANPATH="$MANPTH":$POSTGRES_HOME/man
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH":$PGLIB
```

然后执行 `source ~/.bashrc`。

如果使用的 shell 为 csh/tcsh，则在 .cshrc 中添加以下命令：

```
setenv PATH="$PATH":/usr/local/pgsql/bin
setenv POSTGRES_HOME=/usr/local/pgsql
setenv PGLIB=$POSTGRES_HOME/lib
setenv PGDATA=$POSTGRES_HOME/data
setenv MANPATH="$MANPTH":$POSTGRES_HOME/man
setenv LD_LIBRARY_PATH="$LD_LIBRARY_PATH":$PGLIB
```

然后执行 `source ~/.cshrc`。

以上环境变量是所有使用数据库的用户都需要设置的。

#### (2) 初始化数据库目录：

`initdb`

可以使用的参数有：

`pgdata=pgsql/db`——指定数据库目录，默认使用环境变量 PGDATA 指定的位置。

`pgencoding=EUC_CN`——指定数据库的字符编码，默认使用 configure 时指定的编码。

需要注意的是：执行 `initdb` 命令的用户将拥有所建数据库目录的管理权。

接下来就可以使用 `postmaster` 命令来启动 PostgreSQL，用 `psql` 测试 PostgreSQL 启动是否成功。

## 2.2 rpm 安装

前一节介绍了PostgreSQL的一般安装过程，这也是PostgreSQL随机文档中介绍的安装过程。从中可以看出，PostgreSQL的安装过程非常复杂，主要原因是各种UNIX之间存在二进制不兼容的问题，因此安装过程只有从源代码的编译开始。

对于Linux用户来说，还有更为简单的安装方法。rpm是RedHat公司开发的Linux软件安装和管理工具，利用它能够显著地简化软件包的发行和安装。RedHat公司同样为PostgreSQL准备了各种版本的rpm安装软件包，用户可以在RedHat公司的站点[www.redhat.com](http://www.redhat.com)中找到PostgreSQL的rpm包。下面分别介绍在RedHat Linux 6.x和RedHat Linux 7.x下利用rpm安装PostgreSQL的方法。

### 2.2.1 RedHat 6.x 下的安装

RedHat Linux 6.x发行版本将PostgreSQL作为安装组件之一，只要在Linux安装阶段的软件配置项目下选择PostgreSQL，Linux安装程序就可以完成大部分的安装工作。如果在安装阶段没有选择PostgreSQL，则可按下列步骤进行安装：

(1) 为了确保安装的成功，必须对Linux的环境进行清理。首先检查系统中是否已经安装PostgreSQL。以超级用户身份执行：

```
rpm -qa|grep ^postgresql
```

如果有类似于下面的信息，则表明系统中已经安装了PostgreSQL：

```
postgresql-6.4.2-3
```

```
postgresql-clients-6.4.2-3
```

```
postgresql-devel-6.4.2-3
```

如果要重新安装PostgreSQL，可以先将已安装的软件删除，使用的命令是：

```
rpm -e postgresql-6.4.2-3
```

```
rpm -e postgresql-clients-6.4.2-3
```

```
rpm -e postgresql-devel-6.4.2-3
```

接下来检查是否存在postgres用户，如果存在，将其删除。postgres是PostgreSQL默认的系统管理员，安装时会重建这个用户。删除用户的命令是：

```
userdel -r postgres
```

最后确定/var/lib/pgsql目录为空。

(2) 将Linux安装盘放入光驱。

(3) 装载光驱，使用的命令是：

## 2 系统安装

```
mount /mnt/cdrom
```

(4) 进入rpm软件包目录，使用的命令是：

```
cd /mnt/cdrom/RedHat/RPMS
```

(5) 执行下面的命令，分别安装PostgreSQL服务器、客户端和开发工具：

```
rpm -ivh postgresql-6.4.2-3.i386.rpm
```

```
rpm -ivh postgresql-clients-6.4.2-3.i386.rpm
```

```
rpm -ivh postgresql-devel-6.4.2-3.i386.rpm
```

(6) 更改postgres用户的口令。PostgreSQL 在安装过程中自动创建一个系统管理员账号——postgres，这个用户具有对数据库进行操作的所有权限。可以使用passwd postgres命令为postgres设置口令。

与HOWTO文档中的介绍不同，PostgreSQL默认的PGLIB目录为/usr/lib/pgsql，PGDATA目录为/var/lib/pgsql。

(7) 初始化。按照文档中的介绍，安装完毕后就可以使用，但事实上并非如此。安装完毕后还需要作一些初始化工作。

以postgres身份登录进行数据库的初始化工作。使用的命令是：

```
initdb --pglib=/usr/lib/pgsql --pgdata=/var/lib/pgsql
```

正常情况下，屏幕上应该出现下面的信息：

```
We are initializing the database system with username postgres (uid=102).
This user will own all the files and must also own the server process.
```

```
Creating Postgres database system directory /var/lib/pgsql/base
```

```
Creating template database in /var/lib/pgsql/base/templatel
```

```
Creating global classes in /var/lib/pgsql/base
```

```
Adding templatel database to pg_database...
```

```
Vacuuming templatel
```

```
Creating public pg_user view
```

```
Creating view pg_rules
```

```
Creating view pg_views
```

```
Creating view pg_tables
```

```
Creating view pg_indexes
```

如果出现上面的信息，表明PostgreSQL初始化成功，否则需要重新初始化甚至重新安装PostgreSQL。

(8) 启动。以root身份登录后，用下面的命令启动PostgreSQL。

```
/etc/rc.d/init.d/postgresql start
```

屏幕上应该出现类似于下面的信息：

```
Starting postgresql service: postmaster [696]
```

postgresql是一个启动PostgreSQL的shell脚本文件，PostgreSQL后端服务器执行程序名是postmaster。下面的命令同样可以启动PostgreSQL。

```
postmaster -S -D=/var/lib/pgsql
```

值得注意的是：第一种启动命令最好以root的身份执行，第二种启动命令则决不能以root身份执行。

如果想在Linux系统启动时自动启动PostgreSQL，使用下面的命令：

```
chkconfig --add postgresql
```

(9) 测试。以postgres身份登录，用下面的命令创建一个测试数据库。

```
createdb testdb
```

然后使用下面的命令进入PostgreSQL客户端程序。

```
psql testdb
```

屏幕上应该出现类似于下面的信息：

```
Welcome to the POSTGRESQL interactive sql monitor:
```

```
Please read the file COPYRIGHT for copyright terms of POSTGRESQL
```

```
type \? for help on slash commands
```

```
type \q to quit
```

```
type \g or terminate with semicolon to execute query
```

```
You are currently connected to the database: testdb
```

```
testdb=>
```

如果出现了上面的信息，表明PostgreSQL的安装和初始化工作已经成功地完成了。这一个界面是PostgreSQL的面向命令行的监控程序。

## 2.2.2 RedHat 7.x 下的安装

RedHat Linux 7.x发行版本同样将PostgreSQL作为安装组件之一，但在文件目录的组织上有所不同。在Linux安装阶段选择SQL Server是安装PostgreSQL最简单的方法。以下是手工安装PostgreSQL的过程。

(1) 检查系统中是否已经安装了PostgreSQL。以超级用户身份执行：

```
rpm -qa|grep ^postgresql
```

如果有类似于下面的信息，则表明系统中已经安装了PostgreSQL：

```
postgresql-devel-7.0.2-17
```

```
postgresql-7.0.2-17
```

```
postgresql-server-7.0.2-17
```

如果要重新安装PostgreSQL，可以先将已安装的软件删除，使用的命令是：

```
rpm -e postgresql-devel-7.0.2-17
```

```
rpm -e postgresql-7.0.2-17
```

```
rpm -e postgresql-server-7.0.2-17
```

接下来检查是否存在postgres用户，如果存在，将其删除。postgres是PostgreSQL默认的



## 2 系统安装

系统管理员，安装时会重建这个用户。删除用户的命令是：

```
userdel -r postgres
```

最后确定/var/lib/pgsql目录为空。

(2) 将Linux安装盘放入光驱。

(3) 装载光驱，使用的命令是：

```
mount /mnt/cdrom
```

(4) 进入rpm软件包目录，使用的命令是：

```
cd /mnt/cdrom/RedHat/RPMS
```

(5) 执行下面的命令，安装PostgreSQL的rpm包：

```
rpm -ivh postgresql-7.0.2-17.i386.rpm
```

```
rpm -ivh postgresql-server-7.0.2-17.i386.rpm
```

```
rpm -ivh postgresql-devel-7.0.2-17.i386.rpm
```

(6) 更改postgres用户的口令。PostgreSQL在安装过程中自动创建一个系统管理员账号——postgres，这个用户具有对数据库进行操作的所有权限，使用passwd postgres命令为postgres设置口令。

与HOWTO文档中的介绍不同，PostgreSQL默认的PGLIB目录为/usr/lib/pgsql，PGDATA目录为/var/lib/pgsql。

(7) 启动PostgreSQL。与RedHat Linux 6.x版不同的是，在RedHat Linux 7.x中，postgresql启动脚本能够自动检查是否已经进行了PostgreSQL的初始化。如果没有初始化，postgresql能够自动进行PostgreSQL的初始化。

以root身份登录后，用下面的命令启动PostgreSQL。

```
/etc/rc.d/init.d/postgresql start
```

屏幕上应该出现类似于下面的信息：

```
Checking postgresql installation: [ OK ]
```

```
Starting postgresql service: [ OK ]
```

(8) 测试。以postgres身份登录，用下面的命令创建一个测试数据库。

```
createdb testdb
```

然后使用下面的命令进入PostgreSQL客户端程序。

```
psql testdb
```

屏幕上应该出现类似于下面的信息：

```
Welcome to the POSTGRESQL interactive sql monitor:
```

```
Please read the file COPYRIGHT for copyright terms of POSTGRESQL
```

```
type \? for help on slash commands
```

```
type \q to quit
```

```
type \g or terminate with semicolon to execute query
```

```
You are currently connected to the database: testdb
```

```
testdb=>
```

如果出现了上面的信息，表明PostgreSQL的安装工作已经成功地完成了。

## 2.3 主要文件

在Linux中, PostgreSQL的目录结构与其文档中的介绍有较大的不同, 为了便于读者使用和分析, 下面给出Linux下PostgreSQL的主要文件。

/etc/rc.d/init.d/postgresql	PostgreSQL启动脚本
/usr/bin/createdb	创建数据库
/usr/bin/createlang	创建语言
/usr/bin/createuser	创建用户
/usr/bin/dropdb	删除数据库
/usr/bin/droplang	删除语言
/usr/bin/dropuser	删除用户
/usr/bin/ecpg	内建的SQL C预处理器
/usr/bin/initdb	初始化PostgreSQL数据库
/usr/bin/initlocation	创建第二个PostgreSQL数据库存储区域
/usr/bin/ipcclean	清除已终止的后端遗留的共享内存和信号灯
/usr/bin/pg_ctl	启动、停止、重启postgre master
/usr/bin/pg_dump	导出一个PostgreSQL数据库
/usr/bin/pg_dumpall	导出所有的PostgreSQL数据库
/usr/bin/pg_encoding	
/usr/bin/pg_id	存放Postmaster进程号的文件
/usr/bin/pg_passwd	管理平面口令文件
/usr/bin/pg_upgrade	升级PostgreSQL数据库
/usr/bin/pg_version	
/usr/bin/postgres	启动一个单用户的PostgreSQL后端
/usr/bin/postgresql-dump	导出并删除一个与老版本不兼容的数据库
/usr/bin/postmaster	启动一个多用户的PostgreSQL后端
/usr/bin/psql	交互式PostgreSQL前端工具
/usr/bin/vacuumdb	整理数据库
/usr/lib/libp*.so.*	兼容库文件
/usr/lib/pgsql/*.*	PostgreSQL主要的配置和系统文件
/usr/include/pgsql/*.*	C语言头文件
/usr/share/man/man1	man帮助页文件
/usr/share/doc/postgresql-7.0.2	PostgreSQL随机文档

## 2 系统安装

---

<code>/usr/share/doc/postgresql-7.0.2/admin</code>	PostgreSQL管理员手册
<code>/usr/share/doc/postgresql-7.0.2/postgres</code>	PostgreSQL参考手册
<code>/usr/share/doc/postgresql-7.0.2/programmer</code>	PostgreSQL程序员手册
<code>/usr/share/doc/postgresql-7.0.2/tutorial</code>	PostgreSQL教程
<code>/usr/share/doc/postgresql-7.0.2/user</code>	PostgreSQL用户手册
<code>/var/lib/pgsql</code>	默认的数据库目录
<code>/var/lib/pgsql/backups</code>	备份目录
<code>/var/lib/pgsql/data</code>	数据目录
<code>/var/lib/pgsql/data/base</code>	用户数据库文件目录
<code>/var/lib/pgsql/data/pg_hba.conf</code>	访问控制文件

## 第3章

# 基本原理

---

与大多数数据库管理系统相比，PostgreSQL在技术上有许多独特的东西。充分地理解这些技术，将有助于更深入地掌握和使用PostgreSQL。本章主要包括：

- 系统结构
- 后端服务器
- 查询处理过程
- 规则系统



## 3.1 系统组成

PostgreSQL 数据库系统环境由 3 个主要部分组成，如图 3-1 所示。同一个计算机上可以运行多个数据库服务器，每一个服务器有一个称为 Postmaster 的主进程。Postmaster 进程的主要任务是管理所有的数据库输入请求，为客户端与一个或多个数据库后端之间建立连接。对于任何的客户端请求，如果 Postmaster 确认该请求具有合法的访问权限，它就为该请求派生一个称为“Postgres Backend”的后端进程。每一个客户端连接请求都有一个唯一的 Postgres 进程，对数据库的任何操作都由客户端发起。

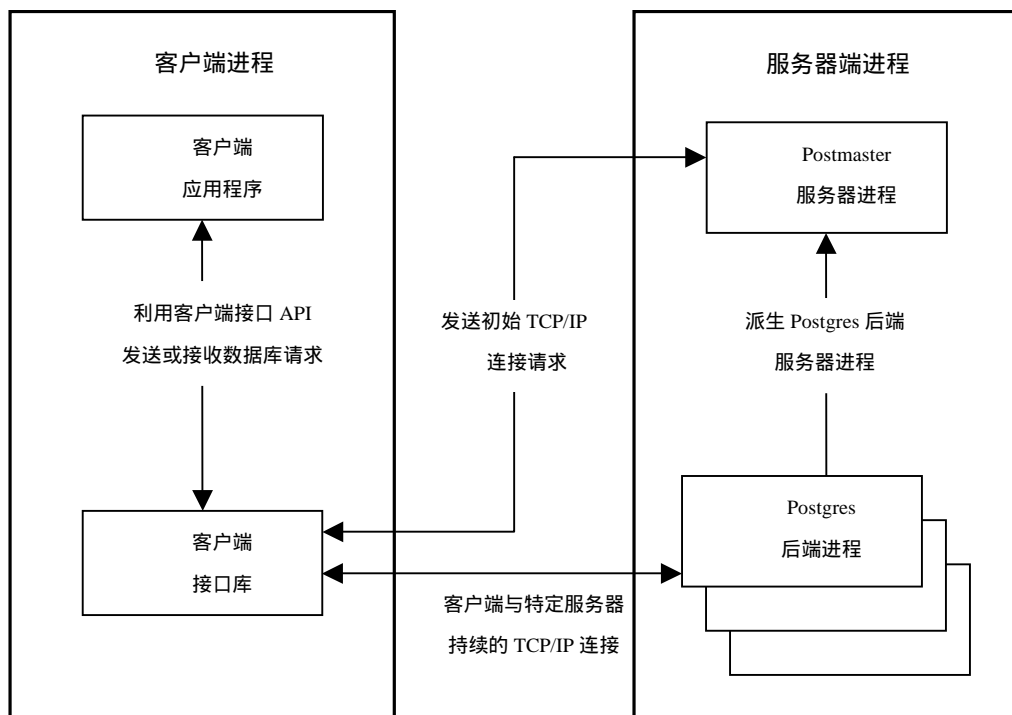


图 3-1 PostgreSQL 数据库环境

绝大多数客户端应用程序都是通过客户端接口库与后端PostgreSQL进行连接的。客户端接口库包含一组标准的应用程序编程接口，这些接口负责与PostgreSQL进行通信。虽然客户端应用程序也可以使用TCP/IP协议直接与PostgreSQL进行交互，但利用客户端接口库能更加便利地管理与服务器的连接。与PostgreSQL基本软件一起发布的客户端接口库有多种，最常用的一种称为libpq；另外一种常用的接口是ODBC，在开放源代码社区可以找到多种免费的ODBC驱动程序。

PostgreSQL的体系结构是相当灵活的，它允许数据库管理员使用不同的技术同时管理运

行在同一台计算机上的多个数据库。另外，每一个PostgreSQL安装实例都提供一组标准的配置文件，这组文件也符合PostgreSQL数据库格式，通常称为template1。用户可以对该数据库进行修改，以适用于特定的服务器和特定的应用。这些配置信息也可以通过createdb程序，复制到新的数据库中。

## 3.2 后端服务器

为客户端提供请求服务的后端服务器进程Postgres，由多个部分组成，Postgres后端服务器的处理过程如图3-2所示。各部分之间通过共享内存以及服务器资源实现彼此之间的通信。每个客户端请求经过一系列的处理步骤，最终实现与PostgreSQL数据库之间的数据交换。

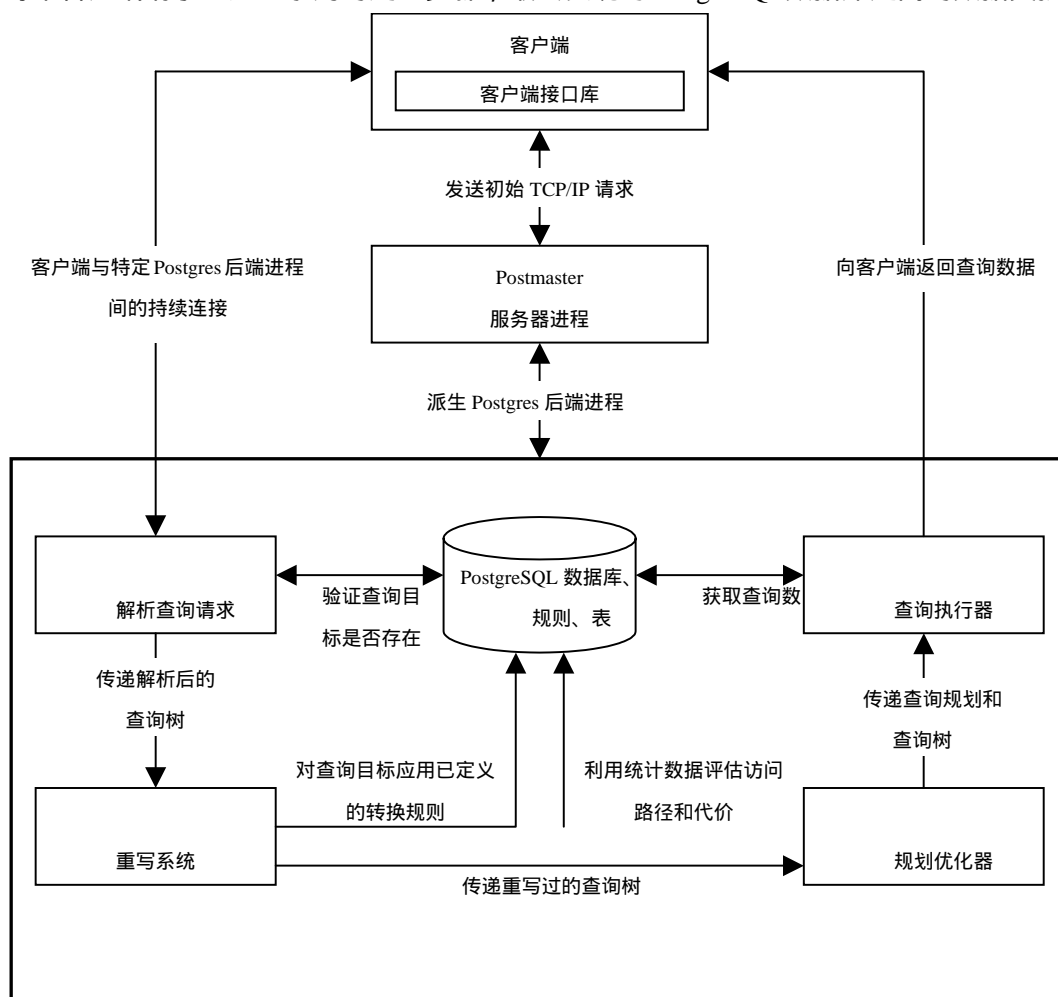


图 3-2 查询请求的处理过程

Postmaster始终在一个指定的端口（默认为5432）监听网络请求，因此客户端应用程序

与PostgreSQL数据库服务器之间的连接是通过Postmaster开始建立的。

利用安全配置文件 `pg_hba.conf`，Postmaster 验证对 Postgres 服务器或特定数据库的访问请求是否合法。如果客户请求是合法的，系统将派生一个特定的 Postgres 后端服务器进程，客户请求将被传递到该进程。客户端与 Postgres 后端进程之间的初始连接只需由 Postmaster 进行一次。一旦建立起来了连接，postgres 后端进程将负责处理其后所有的通信。

### 3.3 查询处理过程

从图3-2可以看出，为了处理客户端的一次查询请求，Postgres后端服务器需要依次完成4个方面的工作。

#### 3.3.1 解析查询请求

为了处理客户的查询请求，后端进程需要调用解析器（Parser）对查询请求进行解析。解析工作包括两个部分：语法解析、转换。

解析器首先检查客户端 SQL 请求的语法是否正确，接下来为查询请求建立一个内部数据结构——查询树。这个查询树将被 Postgres 各后续步骤使用。查询树建立起来之后，解析器将进一步检查查询树所涉及到的数据库对象和属性是否存在于 PostgreSQL 数据库中。如果发现错误或无法解析数据库引用，解析器将向客户端发送一条错误信息，并终止解析过程。

词法分析器 `lexer` 在文件 `scan.l` 中定义，负责识别标识符、SQL 关键字等信息。对于发现的每个关键字或者标识符都会生成一个记号并且传递给解析器。

解析器在文件 `gram.y` 中定义并且包含一套语法规则和触发规则的执行动作。动作代码（实际上是 C 语言代码）用于建立查询树。

文件 `scan.l` 被 `lex` 转换成 C 语言源文件，而 `gram.y` 被 `yacc` 转换成 `gram.c`。在完成这些转换后，通用的 C 语言编译器就能够创建解析器。

#### 3.3.2 重写查询树

如果 Postgres 解析器成功地建立起了查询树，查询请求将转入查询重写系统。在 PostgreSQL 的内部，有一个功能强大的规则系统，它提供处理和转换规则，它的一个主要任务是处理对数据库视图的访问。重写系统使用 PostgreSQL 的规则系统来判断查询树中的任何目标的转换规则是否已经定义。如果已经定义，重写系统将利用已定义的转换规则重写查询树。

重写系统是一个位于解析器和规划优化器之间的一个模块。它处理解析器生成的查询树（该查询树代表用户查询），如果存在一条必须应用的规则，这个模块就将解析树重写成一个变化了的形式。

下面简要分析一下查询重写系统的算法。为了更好地说明问题，这里用视图 test\_view 作为查询的例子。

先给出下面规则：

```
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;
```

当检测到对视图 test\_view 进行 SELECT 查询时，系统就会触发上面给出的规则。这时，查询语句将执行规则里的动作部分，而不是从视图 test\_view 中选择记录。

对于下面的查询：

```
select sname
from test_view where sname <> 'Smith';
```

当用户查询作用于视图 test\_view 时，查询重写系统执行下面的一系列步骤：

- (1) 获取在规则的动作部分给出的查询。
- (2) 调整其目标列表以便与用户查询给出的字段的数目和顺序相匹配。
- (3) 将用户查询的 where 子句中的条件部分，追加到规则动作部分的查询条件上。

经过上面的步骤后，用户查询将被重写为下面的形式：

```
select s.sname
from supplier s, sells se, part p
where s.sno = se.sno and
    p.pno = se.pno and
    s.sname <> 'Smith';
```

从上面的分析可以看出，如果查询树中使用了一个数据库视图，重写系统将对视图的访问转换为对基本数据库表的访问。实际上所有的视图都由 PostgreSQL 规则系统管理。在对查询树应用了所有的数据库规则后，重写结果将传递到规划优化器。

### 3.3.3 规划优化查询树

接下来的工作由规划优化器来进行。规划优化器利用一个久经考验的算法来决定查询的最佳路径和方法，并评估查询代价。根据查询代价，它能够自动选择最有效的查询规划。最后查询规划和查询树将传递给查询执行器，以便得到最终的查询结果。

规划优化器的任务是创建一个优化了的执行规划。它首先合并所有对出现在查询中的关系进行扫描和联合的方法。这样创建的所有路径都将导致相同的结果，而优化器的任务就是计算每个路径的开销并且找出开销最小的路径。

规划优化器以关系索引类型为基础，判断应该生成哪些规划。对一个关系总可以进行一次顺序查找，所以总会创建只使用顺序查找的规划。假设一个关系上定义了一个索引（例如 B-tree 索引），并且一条查询包含约束条件“关系属性 比较符 常量”。如果“关系属性”



正好匹配 B-tree 索引的关键字，并且“比较符”又不是“ $\lt$ ”，系统将会创建另一个使用 B-tree 索引扫描该关系的规划。如果还有其他索引，而且查询中的约束条件与索引的关键字匹配，则还会生成更多的规划。

在找到扫描一个关系所有可能的规划后，接着创建连接 (Join) 各个关系的规划。规划优化器认为只有在每两个关系之间存在连接，对应地在 where 条件中存在一条连接子句。规划优化器为可能的连接关系对生成一个规划。有以下三种可能的连接策略：

(1) 嵌套的反复连接 (nested iteration join)：对在左边关系中找到每条记录都对右边关系进行一次扫描。这个策略容易实现，但是可能会很耗费时间。

(2) 融合排序连接 (merge sort join)：在连接开始之前，每个关系都对连接字段进行排序。然后将两个关系融合到一起。这种联合更有吸引力，因为每个关系都只用扫描一次。

(3) 哈希连接 (hash join)：右边的关系首先对它的连接字段进行哈希运算，然后扫描左边的关系，并将找到的每条记录的合适值作为哈希键字用以定位右边关系的记录。

规划优化器执行的另一个任务是填补运算符 id。PostgreSQL 支持范围广泛的数据类型，甚至可以使用用户定义类型。为了能够维护这些数目巨大的函数和运算符，有必要将它们存放在系统表中。每个函数和运算符有一个唯一的操作符 id。根据使用的字段的类型，必须选用合适的运算符 id。

### 3.3.4 执行查询

查询执行器根据查询规划和查询树，开始从数据库中取出满足要求的数据。当所有的数据被取出后，结果将传回到客户端。

执行器接收由规划优化器生成的规划，开始处理查询树的顶端节点。

在进行任何融合之前，首先要读取两条记录（从每个子规划中读取一条）。执行器递归地调用它自己以处理子规划。新的顶端节点（左边子规划的顶端节点）是一个 SeqScan 节点，同样必须先读取一条记录然后才能处理节点本身。执行器再次递归地调用自身，处理附加在 lefttree 的 SeqScan 节点上的子规划。

现在新的顶端节点是一个 Sort 节点。因此，要对整个关系进行排序。当第一次访问 Sort 节点时，执行器开始从 Sort 节点的子查询中读取记录并将它们在一个临时关系里面（在内存或文件中）排序。

每当处理 Sort 节点需要新的记录时，执行器都会为作为子规划附加上来的 SeqScan 节点递归地调用自己，然后对该关系进行扫描，检索下一条记录。如果记录满足附加在 qpqual 上的条件树，则将其返回；否则读取下一条记录。如果处理到了关系的最后一条记录，则返回一个 NULL 指针。

在 MergeJoin 的 lefttree 返回一条记录后，用同样方法处理右边的子查询树 e。如果两条记录都存在，执行器就处理 MergeJoin 节点。每当有一个子规划需要一条新的记录时，都进行一次执行器的递归调用以获取记录。如果可以创建一条联合的记录，那么就把这条记录返回并且完成一次完整的规划树的处理。

上面描述的步骤对每条记录执行一次，直到对 MergeJoin 节点的处理返回一个 NULL 指针为止。至此，一次客户查询请求全部处理完毕。

客户端与PostgreSQL服务器之间的连接将保持到客户端终止与数据库的连接为止。

## 3.4 规则系统

PostgreSQL 的规则系统在它的查询处理工作中占有非常重要的地位。为了更好地理解 PostgreSQL 工作原理，有必要了解规则系统的一些细节。

规则系统（全称是查询重写规则系统）与存储过程和触发器完全不同，它将查询修改为需要考虑规则的形式，然后将修改过的查询传递给查询优化器执行。这是一个非常有效的工具，还可以用于查询语言过程、视图和版本等数据库对象。

许多其他数据库系统定义动态的数据库规则，通常是存储过程和触发器。在 PostgreSQL 中，这些都是通过函数和触发器来实现的，因此在 PostgreSQL 中是找不到存储过程的。

### 3.4.1 查询树

为了理解规则系统的工作原理，首先要弄清楚规则何时被激发以及它的输入和结果是什么。

规则系统位于查询解析器和优化器之间。它以解析器的输出（查询树）以及 `pg_rewrite` 系统表中存储的重写规则作为输入。重写规则也是一个查询树，只不过增加了一些扩展信息，然后创建零个或者多个查询树作为结果。因此，规则系统的输入和输出总是那些解析器可以生成的元素，换句话说，规则系统所看到的内容都是可以用 SQL 语句表达的。

#### 1. 什么是查询树

查询树是 SQL 语句的一种内部表现形式，组成该语句的每个部分都是分别存储的。当用调试级别（`debuglevel`）4 运行 Postgres 后端并且交互地输入查询命令时，可以看到查询树的各组成部分。在 `pg_rewrite` 系统表中的规则动作也是以查询树的方式存储的，不过不同于调试输出的格式，但内容是完全一样的。

#### 2. 查询树的组成

查询树的成员包括：

- (1) 命令类型。它存储查询的 SQL 语句的动词，包括 `SELECT`、`INSERT`、`UPDATE`、`DELETE` 等。
- (2) 可排列元素。可排列元素是查询中使用的关系列表。在 `SELECT` 语句中，它是在 `FORM` 关键字后面给出的关系。

每个可排列元素表示一个表或一个视图，表明查询中哪个成员调用了它。在查询树中，可排列元素是用索引而不是用名字引用的，所以不用像 SQL 语句那样担心是否有重名问

题。

(3) 结果关系。这是一个可排列元素的索引，用于标识查询结果之间的关系。

SELECT 查询通常没有结果关系。SELECT INTO 等同于 CREATE TABLE、INSERT ... SELECT 序列。

在 INSERT、UPDATE 和 DELETE 查询中，结果关系是查询所影响的表或视图。

(4) 目标列表。目标列表是一组定义查询结果的表达式。在 SELECT 查询中，这些表达式构建查询的最终输出。它们位于 SELECT 和 FROM 关键字之间。

DELETE 查询不需要目标列表，因为它们不产生任何结果。实际上优化器会向空目标列增加一个特殊条目。

在 INSERT 查询中，目标列表描述应该进入结果集的新行。忽略的字段将由优化器自动赋予一个常量 NULL。这些就是 VALUES 子句中的表达式或在 INSERT ... SELECT 语句中的 SELECT 子句。

在 UPDATE 查询中，目标列表描述应该替换旧行的新行。这时，优化器将通过插入从旧行中提取数据到新行的表达式向新行追加丢失的字段，并且它也会像 DELETE 查询那样增加特殊的条目。

目标列表中的每个元素可以是常量，也可以是一个指向某个可排列元素中的关系字段的变量指针，或是一个由函数调用、常量、变量及操作符等构成的表达式。

(5) 查询资格。查询资格是一个表达式，它非常类似于包含在目标列表中的条目。这个表达式的值是一个布尔值，通过此值来判断对最终结果行是否要执行操作（INSERT、UPDATE、DELETE 或 SELECT）。它是 SQL 语句中的 WHERE 子句。

(6) 其他。查询树的其他部分，如 ORDER BY 子句，在这里不作详细讨论。规则系统在附加规则时将在 ORDER BY 子句处替换规则。

### 3.4.2 规则系统和视图

#### 1. 视图的实现

在 PostgreSQL 中，视图是通过规则系统来实现的。实际上，命令：

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

与下面的两条命令：

```
CREATE TABLE myview (same attribute list as for mytab);
```

```
CREATE RULE "_RETmyview" AS ON SELECT TO myview DO INSTEAD  
SELECT * FROM mytab;
```

之间没有绝对区别，因为后两条命令就是 CREATE VIEW 命令在 PostgreSQL 内部实际执行的内容。

#### 2. SELECT 规则

ON SELECT 的规则在最后一步作用于所有查询，即使查询命令是一条 INSERT、UPDATE 或 DELETE，而且与其他规则有不同的语义。因此，这里先介绍 SELECT 规则。

在SELECT规则中，只可能发生一个动作而且必须是一个INSTEAD的SELECT动作。这个限制是为了规则的安全性，以致于普通用户也可以使用，并且它能对真正的视图规则实施ON SELECT规则限制。

这里的例子是两个联合视图，它们进行一些运算并且会涉及到更多视图的使用。这两个视图之一稍后将利用INSERT、UPDATE和DELETE操作附加规则来客户化，这样做的最终结果就会使这个视图表现得像一个具有特殊功能的真正表。

本例子中用到的数据库名是al\_bundy，而且这个例子需要安装过程语言PL/pgSQL，因为它需要一个min()函数，它的功能是返回两个整数值中的较小的整数值。创建这个函数的命令是：

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS
    'BEGIN
        IF $1 < $2 THEN
            RETURN $1;
        END IF;
        RETURN $2;
    END; '
LANGUAGE 'plpgsql';
```

需要用到的表的定义是：

```
CREATE TABLE shoe_data (
    shoename    char(10),      -- primary key
    sh_avail    integer,      -- available # of pairs
    slcolor     char(10),      -- preferred shoelace color
    slminlen    float,        -- minimum 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
);
```

视图的定义为：

```
CREATE VIEW shoe AS
```

### 3 基本原理

```
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;
```

用于shoelace的CREATE VIEW命令将创建一个关系表（shoelace），并且在 pg\_rewrite 表中增加一个记录，告诉系统有一个重写规则应用于所有索引了的关系表（shoelace）的查询。该规则没有规则资格，并且它是INSTEAD型的。需要注意的是，规则资格与查询资格不一样。

规则动作（action）是一个查询树，实际上就是创建视图命令中的SELECT语句。

接下来，输入数据并使用SELECT观察结果。

```
testdb=# INSERT INTO unit VALUES ('cm', 1.0);
testdb=# INSERT INTO unit VALUES ('m', 100.0);
testdb=# INSERT INTO unit VALUES ('inch', 2.54);
testdb=# INSERT INTO shoe_data VALUES
```

```

testdb=# ('sh1', 2, 'black', 70.0, 90.0, 'cm');
testdb=# INSERT INTO shoe_data VALUES
testdb=# ('sh2', 0, 'black', 30.0, 40.0, 'inch');
testdb=# INSERT INTO shoe_data VALUES
testdb=# ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
testdb=# INSERT INTO shoe_data VALUES
testdb=# ('sh4', 3, 'brown', 40.0, 50.0, 'inch');
testdb=# INSERT INTO shoelace_data VALUES
testdb=# ('sl1', 5, 'black', 80.0, 'cm');
testdb=# INSERT INTO shoelace_data VALUES
testdb=# ('sl2', 6, 'black', 100.0, 'cm');
testdb=# INSERT INTO shoelace_data VALUES
testdb=# ('sl3', 0, 'black', 35.0, 'inch');
testdb=# INSERT INTO shoelace_data VALUES
testdb=# ('sl4', 8, 'black', 40.0, 'inch');
testdb=# 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
.....
(8 rows)

```

SELECT \* FROM shoelace查询被解析器解释成下面的分析树。

```

SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace;

```

接下来，解析器将这些信息交给规则系统。规则系统将可排列元素过滤一遍，检查在pg\_rewrite表中有没有适用该关系的规则。当为shoelace处理可排列元素时，它会发现查询树中有规则\_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);

```

此时，解析器已经将查询中的计算表达式和资格转换成了相应的函数。重写的第一步是将两个可排列元素归并在一起，结果生成的查询树是：

```

SELECT shoelace.sl_name, shoelace.sl_avail,

```

## 3 基本原理

```
        shoelace.sl_color, shoelace.sl_len,
        shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace, shoelace *OLD*,
        shoelace *NEW*, shoelace_data s,
        unit u;
```

第二步，将规则动作追加到查询树中，结果是：

```
SELECT shoelace.sl_name, shoelace.sl_avail,
        shoelace.sl_color, shoelace.sl_len,
        shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace, shoelace *OLD*,
        shoelace *NEW*, shoelace_data s,
        unit u

WHERE bpchareq(s.sl_unit, u.un_name);
```

第三步，将分析树中的所有变量，用规则动作中对应的目标列表达式替换，这些变量是引用了可排列元素的变量。这就生成了最后的查询命令：

```
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 shoelace, shoelace *OLD*,
        shoelace *NEW*, shoelace_data s,
        unit u

WHERE bpchareq(s.sl_unit, u.un_name);
```

上面的结果相当于：

```
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;
```

这是应用了第一个规则的结果。在此之后，可排列元素增加了，所以规则系统会继续检查下一个规则（shoelace \*OLD\*）。对于该规则，由于可排列元素没有被查询树中的任何变量引用，所以被忽略。所有剩下的可排列元素要么在pg\_rewrite表里面没有记录，要么没有被引用，因而到达了排列元素结尾，重写过程结束。

上面的结果就是给优化器的最终结果。优化器忽略分析树中多余的没有被变量引用的可排列元素。

值得注意的是，目前的规则系统中没有用于终止视图规则递归的机制。

### 3. 非 SELECT 规则

有两个查询树的细节在上面的介绍中没有涉及到：命令类型和结果关系。

SELECT的查询树与其他命令的查询树只有少数区别。除了命令类型不同和返回结果关

系指向生成结果的可排列元素入口外，其他方面都完全是一样的。例如下面两个语句的查询树几乎是一样的。

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

```
UPDATE t1 SET b = t2.b WHERE t1.a = t2.a;
```

它们的查询树都包含：可排列元素，包含表t1和t2的记录；目标列，包含一个指向字段表t2的可排列元素b；资格表达式，为比较两个表的字段a。

结果是，两个查询树生成相似的执行规划，它们都是两个表的联合。对于UPDATE查询，优化器将t1默认的字段的追加到目标列，因而最终查询树应该是：

```
UPDATE t1 SET a = t1.a, b = t2.b WHERE t1.a = t2.a;
```

对于SELECT查询，最终的查询树是：

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

但是，对于UPDATE而言，执行器不关心它正在处理的联合结果的含义是什么，只是产生一个行的结果集。除此之外，在UPDATE和DELETE语句的目标列表中，需要增加另外一个条目，即当前的记录ID（ctid）。这是一个有着特殊性质的系统字段，它包含行在存储块中的存储块数和位置信息。在已知表的情况下，ctid可以通过简单地查找某一数据块在有百万条记录的表中查找某一特定行。在将ctid加到目标列表中去以后，最终的结果可以定义为：

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

这时，表的行还没有被覆盖，这就是ABORT TRANSACTION的执行速度快的原因。在UPDATE语句中，新的结果行插入到表中（在通过 ctid 查找之后）并且将ctid指向的 cmax 和xmax的行记录头设置为当前命令计数器和当前事务ID。这样旧的行就被隐藏起来，并且在事务提交之后，vacumm cleaner便可以将它们真正地删除掉。

有了这些后，就可以简单的将视图规则应用到任意命令中。

### 3.4.3 视图的能力

在规则系统中实现视图的好处是优化器在一个查询树中拥有所有信息：应该扫描哪个表、表之间的关系、视图的资格限制、初始查询的资格（条件）等。由于优化器必须决定执行查询的最优路径，因此它拥有的信息越多，决策结果就越好，并且规则系统的实现保证这些信息是当前能够获得的所有信息。

在很长一段时间里，PostgreSQL规则系统被认为是有问题的，规则的使用不被推荐且能工作的部分只有视图规则。在6.4版中，规则系统被彻底改写，结果是令人满意的。尽管如此，PostgreSQL的规则系统仍然还有一些无法处理的构造和失效的地方，主要原因是这些东西现在不被PostgreSQL查询优化器所支持。

问题之一是带聚集字段的视图。在资格列表中的聚集表达式必须通过子查询实现。目前还不可能进行两个带集聚字段的视图的联合查询，限制是：每个视图都有一个聚集字段，并且与资格列表中的两个聚集的结果进行比较。但可以将这些聚集表达式放到函数中，通过合适的参数在视图定义中使用它们。

联合视图目前也不被支持。尽管可以很容易地将一个简单的SELECT重写成联合，但是如果该视图是一个正在做更新联合的一部分时就会有问题。其他的限制还有：视图中不能使



用ORDER BY 子句、不能使用DISTINCT关键字。

### 3.4.4 规则和权限

由于 PostgreSQL 规则系统对查询的重写,最终的查询操作可能需要访问没有在原始查询命令中显式指定的表或视图;另外,更新规则可能包括对表的写权限,因此涉及到操作权限问题。

重写规则并不拥有一个独立的所有者,表或视图的所有者自动成为重写规则的默认所有者,规则系统具有改变默认的访问控制系统的特性。因规则而使用的关系在规则重写时要对定义规则的关系所有者的权限进行检查。这意味着一个用户只需要对原始查询中涉及的表或视图拥有所需的权限即可。

例如,某用户有一个电话号码列表,其中一些是私人的,另外的一些是办公室秘书需要的。他可以用下面方法构建查询:

```
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;
```

除了他(还有数据库超级用户)以外,没有人可以访问 phone\_data 表。但因为有了 GRANT,秘书可以利用 phone\_number 视图进行查询。规则系统将 phone\_number 中的查询(SELECT)命令重写为从 phone\_data 中进行的查询命令,并将增加资格(条件),只有私人条件为假的记录才可以取出。

权限检查是按规则逐条进行的,所以此时秘书是唯一的一个可以看到公共电话号码的人。但秘书可以设立另一个视图并且赋予该视图公共权限。这样,任何人都可以通过秘书的视图看到 phone\_number 数据。秘书不能做的事情是创建一个直接访问 phone\_data 的视图。而且用户很快会认识到,秘书开放了他的 phone\_number 视图后,他还可以撤回他的访问权限。这样,所有对秘书视图的访问就失效了。

有些人会认为这种逐条规则的检查是一个安全漏洞,但事实上不是。如果这样做不能奏效,秘书将必须建立一个与 phone\_number 有相同字段的表并且每天复制一次数据。那么,这些复制的数据就是他自己的数据,因而他可以赋予任何人访问的权力。

这个机制同样可以用于更新规则。

### 3.4.5 规则与触发器的比较

许多用触发器完成的任务都可以用 PostgreSQL 规则系统来完成。目前不能用规则来实现的任务是某些约束。

一个用于视图插入的触发器可以做到与规则一样,但对于更新和删除就有所不同。这是因为,视图中没有可供扫描的真实数据,因而触发器将永远不被调用。这时只有规则可用。

对于两者都可用的情况,哪个更好取决于对数据库的使用。触发器一般用于只涉及一行数据的操作,而规则系统则可修改查询树或生成另外一个查询树,因此如果一个语句中

涉及到多行数据，规则通常可能会比触发器要好一些。

例如，下面的表：

```
CREATE TABLE computer (  
    hostname      text      -- indexed  
    manufacturer  text      -- indexed  
);  
  
CREATE TABLE software (  
    software      text,      -- indexed  
    hostname      text      -- indexed  
);
```

这两个表都可能有好几千行数据，并且 hostname 是唯一的，它是完整的计算机域名。规则或触发器应该对主机已删除的记录行的动作进行约束。因为触发器在每个独立的行删除的时候都要调用，可以使用下面语句：

```
DELETE FROM software WHERE hostname = $1;
```

在规则系统中，可以像下面描述的那样定义一个规则：

```
CREATE RULE computer_del AS ON DELETE TO computer  
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

现在来看看这两种不同删除命令的结果。

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

上面的命令对表 computer 使用索引进行扫描，并且对由触发器定义的查询也用索引进行扫描。

对于规则系统，使用下面的命令：

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'  
    AND software.hostname = computer.hostname;
```

因为已经建立了合适的索引，优化器将创建下面的规划：

```
Nestloop  
    -> Index Scan using comp_hostidx on computer  
    -> Index Scan using soft_hostidx on software
```

所以在规则和触发器的实现之间没有太多的速度差别。在下面的例子中，希望删掉所有主机名以“old”开头的计算机。有两个可用的查询，一个是：

```
DELETE FROM computer WHERE hostname >= 'old'  
    AND hostname < 'ole'
```

这样的规则查询的规划将会是：

```
Hash Join  
    -> Seq Scan on software  
    -> Hash  
        -> Index Scan using comp_hostidx on computer
```

另一个可能的查询是：

```
DELETE FROM computer WHERE hostname ~ '^old';
```

### 3 基本原理

它的执行规划是：

Nestloop

```
-> Index Scan using comp_hostidx on computer
-> Index Scan using soft_hostidx on software
```

触发器将在任何要被删除的“old”计算机时被调用一次，结果是对 computer 的一次索引扫描和对 software 的多次索引扫描。但是，规则系统将只对两个索引进行查询。

另外一个查询的例子是：

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

同样，这也会导致从 computer 表中删除多行数据，所以触发器同样会向执行器提交很多查询请求。但是，规则规划又将只作用于两个索引扫描的内部循环，只对 computer 使用另外一个索引：

Nestloop

```
-> Index Scan using comp_manufidx on computer
-> Index Scan using soft_hostidx on software
```

使用规则系统的查询实际上是：

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
AND software.hostname = computer.hostname;
```

综上所述，在任何情况下，利用规则系统的查询相对独立于查询中涉及到的行的数量。

另一情况是更新（UPDATE），这时某字段的更改决定一个规则动作是否被执行。在 PostgreSQL 版本 6.4 中，对规则事件的字段说明被取消了。所以，创建像在 showlace\_log 中定义的规则唯一的办法是用一个规则资格（条件）来实现。结果是引入了一个永远执行的额外查询，即使用户关心的字段因为没有在原始查询目标列表中出现而不能修改，情况也是一样的。当这个特性（字段声明）重新可用后，这将是规则优于触发器的又一个方面。在这种情况下，触发器的定义必然会无法优化，因为触发器的动作只有在声明字段的更新被隐含在触发器的功能里面时才能执行。对触发器的执行只允许到行的级别，所以当涉及到行时，触发器就会按定义而被触发。规则系统将通过目标列表的扫描获知是否动作，并且在字段没有被涉及到时将多余的查询完全去掉。所以不管规则合格与否，只要有事情要做，都将进行扫描。

规则只是在它们的动作生成了复杂的条件联合时才比触发器有较大速度差异，这时优化器将失效。有人将规则比喻为一个榔头。榔头的不慎使用会导致大破坏，但合理使用，它可以钉上任何钉子！

## 第4章

# 用户与数据库

---

本章主要介绍PostgreSQL最基本的对象：用户和数据库，  
内容包括：

- 安全认证

- PostgreSQL 用户

- 数据库



## 4.1 安全认证

PostgreSQL 是一个具有完善安全认证机构的多用户数据库，任何对它的数据库的访问请求都必须经过它的安全认证之后才能进行。本节简要介绍 PostgreSQL 的安全认证机构。

### 4.1.1 安全认证

为了保证数据库和核心文件的安全，PostgreSQL 提供了多种安全认证机制。

(1) 数据库文件保护。PostgreSQL 所有的数据库文件都受到 UNIX 文件和目录保护机制的保护，除了 PostgreSQL 超级用户外，其他用户均不能读取。

在 Linux 中，PostgreSQL 默认的数据库文件存放目录是 `/var/lib/pgsql`，其目录结构请参考第 2 章。默认情况下，PostgreSQL 的超级用户的名字为 `postgres`。只有它才能访问 `/var/lib/pgsql` 目录。PostgreSQL 的所有数据库对象文件都是独立的文件，并且基本上没有采取任何加密措施。只要权限许可，任何人都能读取并分析这些文件。因此，PostgreSQL 的数据库必须妥善保护。

需要注意的是，Linux 的超级用户也能访问并操作 PostgreSQL 的数据库目录。

(2) 网络保护。默认情况下，客户端只能通过一个本地的 UNIX 套接字连接到数据库服务器，而不能通过 TCP/IP 套接字连接。必须带 `-i` 选项启动 `postmaster` 才能允许非本地的客户端进行数据库连接。

在 RedHat 6.x 或兼容的 Linux 中，可以修改 `/etc/rc.d/init.d/postgresql` 脚本文件来允许或禁止网络连接。修改的方法是：以 `root` 身份登录，将目录切换到 `/etc/rc.d/init.d`，用 `vi postgresql`（或其他文本编辑器）编辑 `postgresql` 文件。图 4-1 说明了允许通过 TCP/IP 访问 PostgreSQL 的修改方法。

```
# See how we were called.
case "$1" in
  start)
    echo -n "Starting postgresql service: "
    su -l postgres -c '/usr/bin/postmaster -S -i -D/var/lib/pgsql'
    sleep 1
    pid=`pidof postmaster`
    echo -n "postmaster [$pid]"
    touch /var/lock/subsys/postgresql
    echo $pid > /var/run/postmaster.pid
    echo
    ;;
  stop)
```



图 4-1 修改 PostgreSQL 启动脚本

在 RedHat 7.x 或兼容的版本中,情况有所不同。在新的版本中,postgresql 脚本文件已经有了较大幅度的修改。新版本的 postgresql 通过 pg\_ctl 来启动 postmaster,而不是像老版本那样直接启动 postmaster,并且在一个专门的 postmaster 配置文件中指定 postmaster 的启动选项,这个文件的名称是 postmaster.opts.default,存放在/var/lib/pgsql/data 目录中。默认情况下,该文件只包含一个启动“-i”。这表明,默认情况下 PostgreSQL 允许客户通过 TCP/IP 连接数据库。

必须将所有的 postmaster 启动选项写在 postmaster.opts.default 文件的第一行中。例如,“-i p 6666”。这一行表明,允许客户通过 TCP/IP 连接数据库,使用的 TCP/IP 端口为 6666 (不是默认的 5432)。

为了保证系统的安全,除非必要,否则应该禁止通过 TCP/IP 连接数据库。

(3) 限定通过网络连接的主机。可以通过/var/lib/pgsql/data 中的 pg\_hba.conf 文件来限制可连接的客户端的 IP 地址或主机名、用户名。当然,也可以通过其他外部认证软件包来进行客户连接的认证。

(4) 用户认证。PostgreSQL 为每一个用户都赋予了一个用户名和一个可选的口令。默认情况下,用户对不是由他们创建的数据库没有写权限。

用户可以被赋予组,而且表的访问权限可以以组为基础进行设置。

### 4.1.2 用户认证

用户认证是后端服务器和 postmaster 认定正在请求访问数据的用户的过程。所有合法的 PostgreSQL 用户都登记在 pg\_user 系统视图中(结构见表 4-1 所示),用户认证工作就是依据这个系统表的内容来进行的。

表 4-1 pg\_user 系统表的结构

字段名	类型	意义
username	name	用户名
usesysid	integer	用户的系统 id 号
usecreatedb	boolean	创建数据库的权限
usetrace	boolean	跟踪权限
usesuper	boolean	是否为 PostgreSQL 的超级用户
usecatupd	boolean	
passwd	text	以"*****"表示的用户口令

pg\_user 实际上不是一个真正的表,而是 pg\_shadow 系统表的一个视图,定义它的命令是:

```
SELECT pg_shadow.username, pg_shadow.usesysid, pg_shadow.usecreatedb,  
       pg_shadow.usetrace, pg_shadow.usesuper, pg_shadow.usecatupd,  
       '*****'::text AS passwd, pg_shadow.valuntil FROM pg_shadow;
```

对用户实际身份的验证可以通过以下两种方式进行。

#### 1. shell 用户

对于 shell 用户,后端数据库服务器进行 setuid 调用,将用户标识号转换为用户之前检查用户的有效用户标识号。有效的用户标识号被当作访问控制检查的基础,不再进行其他的认证。

## 2. 网络用户

在网络环境中，任何人都可以访问 postmaster 进程的 TCP 端口。数据库管理员可以配置数据库目录中的 pg\_hba.conf 文件，PostgreSQL 的安全验证机构根据这个文件来验证访问的合法性。在 UNIX 中，以主机为基础的认证不是无懈可击的，有能力入侵者也可能伪装源主机的 IP 地址，但这些安全性问题已超越了 PostgreSQL 的范畴。

每个正在访问数据库的客户端必须被 pg\_hba.conf 中的一条记录涵盖，否则所有从该客户端发来的连接请求都将被拒绝，错误信息为“User authentication failed”(用户认证失败)。

文件 pg\_hba.conf 的常用格式是记录，每行一条记录。空行或者以#开头的行被忽略。一条记录是由若干个空格或 tab 分隔的字段组成。

客户端的连接可以使用 UNIX 套接字或 TCP/IP 建立。用 UNIX 域套接字连接的控制记录格式是：

```
local database authentication method
```

其中：

database——指定需要连接的数据库，all 表示连接所有数据库。

authentication method——指定用户在用 UNIX 域套接字与数据库连接时用于认证的方法。主要的认证方法有：

trust——无条件地允许连接。

reject——无条件地拒绝连接。

crypt——要求客户端提供一个口令。该口令被加密后（使用 crypt 3）发送，然后与 pg\_shadow 系统表中的口令进行比较。如果口令匹配，则允许连接。

password——要求客户端提供一个口令。该口令以明文发送，然后与 pg\_shadow 系统表中的口令进行比较。如果口令匹配，则允许连接。可以在 password 关键字的后面指定一个可选的文件名，用于提供对应的口令而不是使用 pg\_shadow 系统表。

下面的认证方式只能用于 TCP/IP 域套接字：

krb4——利用 Kerberos V4 进行用户认证。

krb5——利用 Kerberos V5 进行用户认证。

ident——客户端的 ident 服务器用于认证该用户（RFC 1413）。可以在 ident 关键字的后面指定一个可选的映射名，这样允许 ident 用户名映射成 PostgreSQL 用户名。映射放在文件 pg\_ident.conf 中。

用 TCP/IP 连接的控制记录是：

```
host database TCP/IP address TCP/IP mask authentication method
```

TCP/IP address 逻辑上与 TCP/IP mask 和正在连接的客户端的 TCP/IP 地址分别相加。如果两个结果相等则该记录可用于该连接。如果一个连接匹配多条记录，那么使用文件中的第一条匹配的记录。TCP/IP address 和 TCP/IP mask 都是用“.”分隔的十进制数。authentication method 为认证方法，可用的方法有：trust、reject、crypt 和 password，前面已经介绍了它们的意义。

下面是 pg\_hba.conf 的一个例子。

```
# 允许所有 UNIX 域套接字类型的数据库连接
```

```
local    trust
# 允许本机用户的所有 TCP/IP 连接
host     all         127.0.0.1         255.255.255.255      trust
# 禁止来自 192.168.0.10 主机的 TCP/IP 连接
host     all         192.168.0.10      255.255.255.0        reject
# 以明文口令的形式验证来自 192.168.0.3 主机的连接
host     all         192.168.0.3       255.255.255.0        password
# 以密文口令的形式验证来自 192.168.0.0 主机的连接
host     all         192.168.0.0       255.255.255.0        crypt
```

## 4.2 PostgreSQL 用户

只有 PostgreSQL 的合法用户才能访问它的数据库对象。PostgreSQL 的用户与 Linux 系统的用户是不同的，PostgreSQL 的用户不一定是 Linux 的用户，Linux 的用户也不一定是 PostgreSQL 的用户。换句话说，PostgreSQL 用户与 Linux 用户是两类概念完全不同的用户。

### 4.2.1 创建用户

#### 1. 创建用户的命令

```
createuser [ options ] [ username ]
```

createuser 是一个 shell 脚本，它通过 PostgreSQL 交互式前端工具 psql，封装了 SQL 命令 CREATE USER。只有具有用户创建权限的 PostgreSQL 才能创建新的用户。

其中，username 是要创建的 PostgreSQL 用户名称。该名称必须在所有 PostgreSQL 用户中唯一。options 是命令的可选项，主要的可选项包括：

- h, --host host——正在运行 postmaster 的主机名。
- p, --port port——postmaster 正在侦听的 TCP/IP 端口号或本地 UNIX 域套接字的文件扩展（描述符）。
- e, --echo——回显 createdb 生成的查询并发送给后台。
- q, --quiet——不显示响应。
- d, --createdb——允许该用户创建数据库。
- D, --no-createdb——禁止该用户创建数据库。
- a, --adduser——允许该用户创建其他用户。
- A, --no-adduser——禁止该用户创建其他用户。
- P, --pwprompt——如果给出此选项，createuser 将显示一个提示符要求输入新用户的口



令。如果不使用口令认证，这一可选项是不必要的。

`-i, --sysid uid`——允许新用户使用非默认用户标识，这个可选项不是必须的。

如果没有在命令行上指定名称或其他必要信息，脚本会提示输入这些信息。

选项 `-h`、`-p` 和 `-e` 将逐字地传递给 `psql`。`psql` 选项 `-U` 和 `-W` 也可以使用，但是这些开关的使用在这个环境中可能有些混乱。

如果用户创建成功，则返回信息“`CREATE USER`”；否则，返回信息“`createuser: creation of user "username" failed`”，表明创建用户失败。

值得注意的是，只有具有创建用户权限的 PostgreSQL 合法用户，才能利用这个命令创建新的用户。非 PostgreSQL 用户，即使是 Linux 的超级用户，也没有创建 PostgreSQL 新用户的能力。

### 2. 例子

PostgreSQL 的安装程序自动建立了一个用户 `postgres`，这个用户是 PostgreSQL 超级用户，同时它也是一个合法的 Linux 用户。一般情况，可以利用这个用户的身份，创建其他合法的 PostgreSQL 用户。

安装程序并没有为 `postgres` 用户指定登录口令，因此还不能以这个用户的身份登录 Linux。必须以 `root` 超级用户身份登录 Linux，为 `postgres` 用户指定登录口令。相关的 Linux 命令是：

```
adduser postgres
```

然后按照屏幕上的提示为 `postgres` 指定口令：

```
Changing password for user postgres
```

```
New UNIX password:
```

```
Retype new UNIX password:
```

```
passwd: all authentication tokens updated successfully
```

现在，可以以 `postgres` 用户的身份登录 Linux 了。下面的例子创建本书所使用的一个 PostgreSQL 用户——`testuser`。

```
//在默认数据库服务器上创建一个用户 testuser
```

```
createuser testuser
```

```
Is the new user allowed to create databases? (y/n) y
```

```
Shall the new user be allowed to create more new users? (y/n) n
```

```
CREATE USER
```

这里所创建的用户是 PostgreSQL 用户，而不是 Linux 的用户。为了便于操作，应该建立一个同名的 Linux 用户，以便能够在 Linux 下以 `testuser` 身份登录，并在 Linux 下以 `testuser` 的身份使用数据库。使用的命令（以超级用户身份）是：

```
adduser testuser
```

注意，上面的 `adduser` 命令是 Linux 的命令，而不是 PostgreSQL 的命令。接下来，应该为 Linux 用户 `testuser` 指定登录口令，使用的命令是：

```
passwd testuser
```

当然，如果不想在 Linux 下使用 PostgreSQL，不必建立这个 Linux 用户。这样，只能

通过其他方式来使用 PostgreSQL 了。

以下是更复杂的创建 PostgreSQL 用户的例子。

//用在主机 eden 上的 postmaster 创建用户 joe, 端口是 5000, 避免提示并且显示执行信息

```
$ createuser -p 5000 -h eden -D -A -e joe
```

```
CREATE USER "joe" NOCREATEDB NOCREATEUSER
```

```
CREATE USER
```

## 4.2.2 用户组

与 Linux 的用户管理机制一样, PostgreSQL 也可以将用户分为若干个用户组。要将一个用户或者一组用户指派到一个用户组中, 先定义组的名称, 然后再将用户分派到该组中。

创建用户组的命令是:

```
CREATE GROUP name
```

```
[ WITH
```

```
[ SYSID gid ]
```

```
[ USER username [, ...] ] ]
```

例如:

```
create group testgroup
```

这条命令只能在 PostgreSQL 的前端工具 psql 中使用, 因此在这里无法进行详细的介绍。用户组的信息存放在系统表 pg\_group 中。

指派用户到一个用户组的命令是:

```
ALTER GROUP name ADD USER username [, ... ]
```

例如:

//将用户testuser添加到testgroup组中

```
testdb=# alter group testgroup
```

```
testdb=# add user testuser
```

```
testdb=# ;
```

```
ALTER GROUP
```

```
testdb=#
```

从一个用户组中删除一个用户的命令是:

```
ALTER GROUP name DROP USER username [, ... ]
```

例如:

//将用户testuser从testgroup组中删除

```
testdb=# alter group testgroup
```

```
testdb=# drop user testuser
```

```
testdb=# ;
```

```
ALTER GROUP
```

```
testdb=#
```

## 4 用户与数据库

也可以使用 SQL 命令直接操作 pg\_group 系统表。pg\_group 主要的字段有：

groname——组名称。这个名称应该由纯数字和字母组成，不能包含下划线和其他符号。

grosysid——组标识号。这是一个 int4 类型的字段，这个字段的值应该在各个组之间唯一。

grolist——属于这个组的 pg\_user id（标识）的列表。

下面的例子直接建立一个名为 testgroup 的组，并将标识号为 5443 和 8261 的用户指派到这个组中。

```
insert into pg_group (groname, grosysid, grolist)  
values ('testgroup', '1234', '{5443, 8261}');
```

删除用户组的命令是：

```
DROP GROUP name
```

一般情况下，用户组在 PostgreSQL 中没有多大的意义，因此不推荐建立用户组。

值得注意的是，进行用户组管理的用户必须是 PostgreSQL 超级用户，否则 PostgreSQL 会给出错误信息并终止对用户组的操作。

例如，testuser 不具有超级用户权限，不能进行用户组的操作。

```
testdb=> ALTER GROUP testgroup  
testdb-> ADD USER user1;  
ERROR: ALTER GROUP: permission denied  
testdb=>
```

### 4.2.3 删除用户

删除用户的命令是：

```
dropuser [ options ] [ username ]
```

dropuser 也是一个 shell 脚本，通过 PostgreSQL 交互前端 psql，封装 SQL 命令 DROP USER。dropuser 删除一个 PostgreSQL 用户和该用户所有的数据库。只有 PostgreSQL 超级用户才能删除 PostgreSQL 用户。

命令的可选项的意义与 createuser 命令相同。

例如：

```
createuser tmpuser  
Shall the new user be allowed to create databases? (y/n) y  
Shall the new user be allowed to create more new users? (y/n) y  
CREATE USER  
dropuser tmpuser  
DROP USER
```

## 4.3 数据库

PostgreSQL 的所有数据库对象，如表、视图、索引等，都存储在特定的数据库中。安装完毕后，PostgreSQL 自动建立一个名为 `template1` 的数据库。用户可以根据需要建立新的数据库。

### 4.3.1 创建数据库

#### 1. 创建数据库的命令

```
createdb [ options ] dbname [ description ]
```

实际上，`createdb` 是一个 shell 脚本，通过 PostgreSQL 交互前端 `psql`，封装了 SQL 命令 `CREATE DATABASE`。执行此命令的用户必须具有创建数据库的权限。一旦创建了一个数据库，创建者就是该数据库的拥有者和管理员。

其中，`options` 是命令的可选项。主要的可选项有：

- h, --host host——正在运行 `postmaster` 的主机名。
- p, --port port——`postmaster` 侦听等待连接的 TCP/IP 端口或本地 UNIX 域套接字描述符。
- U, --username username——进行连接的用户名。
- W, --password——强制口令提示符。
- e, --echo——回显 `createdb` 生成的查询并且将它发送到后台。
- q, --quiet——不显示响应。
- D, --location datadir——数据库安装（节点）的位置。这是数据库的安装系统目录，而不是数据库的位置。
- E, --encoding encoding——用于此数据库的字符编码方式。

命令中的 `dbname` 是要创建的数据库名。数据库名必须以字母开头而且少于 31 个字符，可以通过配置和重新编译 PostgreSQL 改变这个值。该名称应该在本节点的 PostgreSQL 所有数据库中是唯一的。默认的数据库名与当前系统用户同名。

命令中的可选项 `description` 定义一个与新创建的数据库相关的注解。

选项 -h, -p, -U, -W, 和 -e 将被逐字节地传递给 `psql`。

如果数据创建成功，则返回信息 “`CREATE DATABASE`”；否则返回信息 “`createdb: Database creation failed.`”。

## 2. 例子

下面说明创建本书所使用的第一个数据库 testdb 的方法。

首先以 testuser 的身份登录 Linux，然后利用下面的命令创建属于 testuser 的数据库 testdb。

```
createdb testdb
```

```
CREATE DATABASE
```

以下是更为复杂的创建数据库的例子：

//用在主机 eden 上的 postmaster 创建数据库 demo，端口是 5000，使用 LATIN1 编码方式，并且显示执行信息

```
createdb -p 5000 -h eden -E LATIN1 -e demo
```

```
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
```

```
CREATE DATABASE
```

### 4.3.2 删除数据库

删除数据库的命令是：

```
dropdb [ options ] dbname
```

dropdb 是一个 shell 脚本，通过 PostgreSQL 交互终端 psql，封装了 SQL 命令 drop\_database。执行这条命令的用户必须是数据库所有者，或者是数据库的超级用户。

命令中的可选项的意义与 createdb 命令相同。

## 第5章

# psql 基本操作

---

psql是PostgreSQL的一个重要的交互式前端工具命令,类似于ORACLE的SQL \* plus。它工作在服务器环境中,是PostgreSQL所提供的标准数据库访问工具。本章的主要内容包括:

- psql 简介
- 启动数据库会话
- 命令缓冲区
- 联机帮助



## 5.1 psql

psql 是 PostgreSQL 的一个重要的交互式前端工具命令，类似于 ORACLE 的 SQL \* plus。它工作在服务器环境中，是 PostgreSQL 所提供的标准数据库访问工具。

### 5.1.1 简介

psql 提供一种基于命令行的交互式接口，虽然其界面不如其他图形化工具友好，但更直截了当，功能也更完备。它接收用户输入的查询命令，将命令提交给 PostgreSQL 后端执行，然后显示查询结果。输入也可以来自一个文件。另外，它还提供一些专门的命令和许多类似 shell 风格的特性，供用户写脚本和完成某些自动化工作。

在 psql 中，用户可以执行各种 SQL 命令。除此之外，它还提供一系列专有命令，用于获取数据库对象的各种信息，如数据库中包含的表、视图、函数、用户、表的结构、权限等。

psql 支持各种文本编辑器，如 vi、emacs、pico、jed、kedit 等，用户可以利用它们方便地编辑输入的命令行。文本编辑器由环境变量 PSQL\_EDITOR、EDITOR 或 VISUAL 指定。psql 同时还保存着用户输入的命令行的历史记录，因此用户可以非常方便地调出以前输入的命令，极大地简化了用户的输入工作。

### 5.1.2 语法

```
psql [ options ] [ dbname [ user ] ]
```

psql 的语法非常复杂，本书的附录中有对它的详细描述，因此这里不再重复。

最常见和最简单的用法是：

```
psql dbname
```

上面的命令以当前 shell 用户的身份连接由 dbname 指定的数据库。利用 4.2 节创建的用户 testuser 登录 Linux，并使用 4.3 节创建的数据库，执行 psql 的结果是：

```
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=#
```

最后一行“testdb=#”是psql的命令行提示符，其中testdb是当前连接的数据库名称。对于不同的数据库，命令行提示符都不一样。在命令行提示符下，用户可以执行SQL命令和psql的专有命令。

### 5.1.3 两种执行方式

psql通过命令行可选项提供两种不同的执行方式：交互方式和直接方式。

5.1.2小节介绍的例子的执行方式就是交互方式。在交互方式下，psql循环地等待用户输入命令、执行命令，直到用户发出了“\q”才退出。

直接方式是psql的另外一种执行方式。这种方式由psql命令可选项“-c 命令”来启动，psql执行完指定的命令后自动退出。

例如：

```
[testuser@linux pgsql]$ psql testdb -c "SELECT CURRENT_DATE"
      date
-----
2001-01-31
(1 row)
```

```
[testuser@linux pgsql]$
```

这一特性在shell脚本中很有用。实际上，createdb和createuser命令都是用这种方式执行的。有些SQL和psql命令的输出非常多，超过了一个Linux终端屏幕的容量，难以看清。这时也可以借助上述特性，将psql的输出重定向到一个文件中，然后用文本编辑器来阅读。

例如，为了得到psql所有的专有命令，可以使用下面的命令：

```
psql testdb -c "\?">help
```

### 5.1.4 专有命令

除了SQL命令外，psql还提供了许多用于系统管理的命令，即psql专有命令。

在psql中输入的、任何以不带引号的反斜杠（“\”）开头的命令都是psql专有命令，这些命令中的大多数并不会发送到PostgreSQL后台，而是由psql自己处理。这些命令也是令psql可用于管理或书写脚本的原因。专有命令又称为斜杠命令或反斜杠命令。

一个psql专有命令的格式是反斜杠后面紧跟一个命令动词，然后是参数。参数与命令动词以及参数之间以任意个空格字符分隔。

如果需要在参数中使用空格，则必须用单引号将空格包围起来。同样，如果需要在参数中使用单引号，应该在其前面加一个反斜杠。psql会对任何包含在单引号内的信息进行替换，将\n（新行）、\t（tab）、\digits、\0digits和\0xdigits（给出的十进制，八进制或十六进制码的字符）替换掉。

如果一个不带引号的参数以冒号（“:”）开头，该参数会被当作一个变量，并且该变量的值为参数的真正值。



## 5 psql 基本操作

用反单引号（“`”）引起的内容被当作一个命令行传入 shell。该命令的输出（删除了结尾的新行）为参数值。上面描述的转意（字符）序列在反单引号内也有效。

有些命令以一个 SQL 标识的名称（如表名）为参数。这些参数遵循 SQL 语法中关于双引号的规则：不带双引号的标识符强制转换为小写字母。对于所有其他命令，双引号没有特殊含义并且将被当成参数的一部分。

对参数的分析在遇到另一个不带引号的反斜杠时停止。该处会被认为是一个新的专有命令的开始。特殊序列\\（双反斜杠）标识参数的结尾并将继续分析后面的SQL查询。这样SQL和psql命令可以自由地在一行内混合。但是，任何情况下一条专有命令的参数不能超过行尾。

表5-1列出了所有的psql专有命令，有关它们的详细介绍请参考本书第24章。

表 5-1 psql 专有命令

命令	简要说明
\a	转换对齐和非对齐输出模式
\c[onnect] [dbname]- [user]	连接到新的数据库
\C <title>	指定表的标题
\copy ...	执行 SQL COPY 命令
\copyright	显示 PostgreSQL 的版权信息
\d <table>	显示表、视图、索引或序列的定义
\d{t i s v}	列出所有的表、索引、序列或视图
\d{p S l}	列出所有的许可、系统表或大目标
\da	列出所有的集聚函数
\dd [object]	列出表、类型、函数或操作符的注释
\df	列出所有的函数
\do	列出所有的运算符
\dT	列出所有的数据类型
\e [file]	利用外部编辑器编辑当前的查询命令缓冲区
\echo <text>	在标准输出设备上输出 text 文本
\encoding <encoding>	设置客户编码
\f <sep>	改变字段分隔符
\g [file]	将查询送到后端执行，结果存入 file 文件或管道中
\h [cmd]	显示命令的帮助信息。当 cmd=*时，显示所有命令的简单帮助信息
\H	转换 HTML 输出方式
\i <file>	读取并执行 file 文件中的查询命令
\l	列出所有的数据库
\lo_export, \lo_import, \lo_list, \lo_unlink	大目标操作
\o [file]	将所有的查询送到文件 file 或管道中
\p	显示当前查询命令缓冲区的内容
\pset <opt>	设置表的输出格式。<opt> = {format border expanded fieldsep null recordsep tuples_only title tableattr pager}
\q	退出 psql
\qecho <text>	将 text 输出到查询输出流
\r	重置（清除）查询命令缓冲区
\s [file]	显示查询命令历史记录，或将其写入文件 file
\set <var> <value>	设置内部变量的值
\t	仅显示数据行
\T <tags>	HTML 表标记
\unset <var>	清除内部变量
\w <file>	将当前查询命令缓冲区写入文件 file
\x	转换扩展输出标记
\z	列出表的访问许可权限
\! [cmd]	执行 shell 命令

## 5.2 启动数据库会话

PostgreSQL的工作模式为客户/服务器（Client/Server）模式。PostgreSQL服务器等待客户端的连接和查询请求，处理客户请求，并将处理结果返回给客户端。这一过程循环往复，直到服务器被关闭为止。

由于PostgreSQL是作为独立的操作系统进程来运行的，因此客户端应用程序不能直接地与之进行交互，而必须使用特定的PostgreSQL接口。作为标准的客户端交互式管理工具，psql利用标准的接口库（libpq）与PostgreSQL服务器进行通信，其他接口工具将在后续章节介绍。

### 5.2.1 连接数据库

任何一个与 PostgreSQL 服务器的连接必须在一个特定的数据库上进行。也就是说，如果不指定数据库，无法与服务器通信。

对于 psql，有三种连接数据库的方法。第一种方法已经在第 4 章中介绍过，基本形式是：

```
psql dbname
```

在这种连接形式中，数据库为 dbname，连接用户为当前登录的 Linux shell 用户。

第二种形式为：

```
psql
```

由于没有指定数据库和用户名，psql 将使用默认值，即：用户名为当前登录的 Linux 用户名，数据库为与用户名同名的数据库。

启动 psql 后，在其命令提示符下还可以利用它的专有命令“\c [数据库名] [用户名]”，这是连接数据库的第三种形式。例如：

```
pengxm=# \c testdb
```

```
You are now connected to database testdb.
```

```
testdb=#
```

这种形式主要用于在 psql 工作过程中，动态地改变与数据库的连接。

### 5.2.2 与数据库交互

如果在指定的数据库上与PostgreSQL连接成功，用户就可以在psql的命令提示符下输入各种命令，并接收服务器返回的信息。例如，输入“SELECT CURRENT\_TIME;”并按回车键，可以查询到当前的服务器时间。

```
testdb=# SELECT CURRENT_TIME;  
time
```

## 5 psql 基本操作

```
-----
```

```
10:39:03
```

```
(1 row)
```

```
testdb=#
```

psql显示服务器返回查询的结果，并等待用户输入下一条查询命令。又例如：

```
testdb=# SELECT CURRENT_USER;
```

```
getpgusername
```

```
-----
```

```
postgres
```

```
(1 row)
```

```
testdb=#
```

上面的查询结果为当前与PostgreSQL连接的用户的用户名。

## 5.3 命令缓冲区

### 5.3.1 命令的输入

与Linux的bash外壳一样，psql以缓冲区的形式管理用户输入的内容。在psql中，分号“；”或“\g”表明一条命令结束。在用户输入“；”或“\g”并按下回车键后，psql将输入的命令存放在缓冲区历史记录中，并将命令发送到后端服务器执行。

例如：

```
testdb=# SELECT
```

```
testdb-# 25/5
```

```
testdb-# ;
```

```
?column?
```

```
-----
```

```
5
```

```
(1 row)
```

```
testdb=#
```

从上面的例子可以看出，一条命令可以分多行输入，只有在碰到“；”后，psql才认为一条命令的输入结束。psql的专有命令“\g”与“；”有着同样的功能，例如：

```
testdb=# SELECT
```

```
testdb-# SIN(12)
```

```
testdb=# \g
      sin
-----
-0.536572918000435
(1 row)
```

```
testdb=#
```

细心的读者可能已经发现，后续命令行的提示符与第一个命令行的提示符有所不同，其中的“=”变成了“-”。

如果Linux终端支持方向键、`Ctrl`、`Alt`、`Shift`，可以利用`Ctrl`、`Alt`键改变光标的位置以便修正输入错误、利用`Ctrl`、`Alt`键调出命令历史记录。

值得注意的是，上述多行编辑特性只适用于SQL命令的编辑。对于psql的专有命令，必须在一行内输入完毕，并且按回车键立即启动命令的执行。

### 5.3.2 显示缓冲区

可以使用专有命令显示psql命令缓冲区的内容。例如：

```
testdb=# \p
SELECT
SIN(12)

testdb=#
```

在命令的输入过程中，也可以利用“`\p`”随时监视命令缓冲区的内容。例如：

```
testdb=# SELECT
testdb=# 1+2
testdb=# \p
SELECT
1+2
testdb=# \g
?column?
-----
      3
(1 row)

testdb=#
```

### 5.3.3 编辑缓冲区

由于psql是基于命令行的，因此它的命令编辑器为行编辑器，不适合于多行、复杂命令

的编辑。为此，psql提供了一种利用外部文本编辑器编辑命令行的命令“\e”，默认的外部编辑器为vi。在psql命令提示符下，输入“\e”启动外部编辑器。退出外部编辑器后，自动执行编辑过的命令。

### 5.3.4 清除缓冲区

专有命令“\r”用于清除命令缓冲区的内容。例如：

```
testdb=# \r
Query buffer reset (cleared).
testdb=#
```

## 5.4 联机帮助

psql 提供有较为完善的专有命令和SQL命令的联机帮助。

### 5.4.1 专有命令

专有命令“\?”用于列出所有的专有命令信息。例如：

```
testdb=# \?
\a                toggle between unaligned and aligned mode
\c[onnect] [dbname|- [user]]
                  connect to new database (currently 'testdb')
\C <title>        table title
\copy ...         perform SQL COPY with data stream to the client machine
\copyright        show PostgreSQL usage and distribution terms
.....
\z               list table access permissions
\! [cmd]         shell escape or command
```

### 5.4.2 SQL 命令

专有命令“\h”用于列出所有的SQL命令清单。例如：

```
testdb=# \h
Available help:

ABORT              CREATE TRIGGER      FETCH
ALTER GROUP        CREATE TYPE         GRANT
ALTER TABLE       CREATE USER         INSERT
```

ALTER USER	CREATE VIEW	LISTEN
BEGIN	DECLARE	LOAD
CLOSE	DELETE	LOCK
CLUSTER	DROP AGGREGATE	MOVE
COMMENT	DROP DATABASE	NOTIFY
COMMIT	DROP FUNCTION	REINDEX
COPY	DROP GROUP	RESET
CREATE AGGREGATE	DROP INDEX	REVOKE
CREATE CONSTRAINT TRIGGER	DROP LANGUAGE	ROLLBACK
CREATE DATABASE	DROP OPERATOR	SELECT
CREATE FUNCTION	DROP RULE	SELECT INTO
CREATE GROUP	DROP SEQUENCE	SET
CREATE INDEX	DROP TABLE	SHOW
CREATE LANGUAGE	DROP TRIGGER	TRUNCATE
CREATE OPERATOR	DROP TYPE	UNLISTEN
CREATE RULE	DROP USER	UPDATE
CREATE SEQUENCE	DROP VIEW	VACUUM
CREATE TABLE	END	
CREATE TABLE AS	EXPLAIN	

如果想要得到特定SQL命令的帮助信息，可以使用“\h SQL命令动词”。例如：

```
testdb=> \h SELECT
```

```
Command:      SELECT
```

```
Description: Retrieve rows from a table or view.
```

```
Syntax:
```

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    expression [ AS name ] [, ...]
    [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table ]
    [ FROM table [ alias ] [, ...] ]
    [ WHERE condition ]
    [ GROUP BY column [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
    [ ORDER BY column [ ASC | DESC | USING operator ] [, ...] ]
    [ FOR UPDATE [ OF class_name [, ...] ] ]
    LIMIT { count | ALL } [ { OFFSET | , } start ]
```

如果想要得到所有SQL命令的帮助信息，可以使用“\h \*”。例如：

```
testdb=> \h *
```

```
Command:      ABORT
```

```
Description: Aborts the current transaction
```

# 5

## psql 基本操作

---

Syntax:

ABORT [ WORK | TRANSACTION ]

Command: ALTER GROUP

Description: Add users to a group, remove users from a group

Syntax:

ALTER GROUP name ADD USER username [, ... ]

ALTER GROUP name DROP USER username [, ... ]

.....

Command: UPDATE

Description: Replaces values of columns in a table

Syntax:

UPDATE table SET col = expression [, ...]

[ FROM fromlist ]

[ WHERE condition ]

Command: VACUUM

Description: Clean and analyze a Postgres database

Syntax:

VACUUM [ VERBOSE ] [ ANALYZE ] [ table ]

VACUUM [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]

## 第6章

# 基本 SQL 命令

---

PostgreSQL以SQL语言为数据库查询和操纵语言。SQL语言的功能非常强大，使用方法也非常复杂。本章结合psql，介绍最常用的SQL命令的基本使用技巧，主要包括：

- SQL 基本语法

- 创建表

- 插入数据

- 查询数据

- 控制查询结果的输出格式

- 删除数据

- 修改数据

- 删除表





## 6.1 SQL 语法

SQL 是结构化查询语言 (Structured Query Language) 的缩写, 它是最通用的和标准的数据库操纵语言。PostgreSQL 实现了一个扩展了的 SQL92 和 SQL3 语言标准的子集。由于 PostgreSQL 的可扩展性, 一些语言元素在 PostgreSQL 中的实现不如 SQL 标准所描述的那样严格。

### 6.1.1 关键字

PostgreSQL 为 SQL 定义了有明确意义的关键字。有些关键字是保留字, 表明它们只限于出现在某些特定的环境中。其他关键字是非限制关键字, 在某些特定的环境中表明它们有特殊含义, 否则没有什么限制。

#### 1. 保留关键字

在 SQL 标准中, 保留关键字是 SQL 语句中的基本元素, 不能用作标识符或者用于其他任何目的。PostgreSQL 对保留关键字也有类似约束, 具体来说, 这种关键字不允许用来命名字段或者表, 但有时候可以用作字段标签。

下面列出的是 PostgreSQL 特有的保留关键字, 它们既不是 SQL92 的保留关键字, 也不是 SQL3 的保留关键字。这些保留关键字允许用作字段标记, 但不允许用作标识符。

```
ABORT ANALYZE
BINARY
CLUSTER CONSTRAINT COPY
DO
EXPLAIN EXTEND
LISTEN LOAD LOCK
MOVE
NEW NONE NOTIFY
OFFSET
RESET
SETOF SHOW
UNLISTEN UNTIL
VACUUM VERBOSE
```

以下列出的是 PostgreSQL 的保留关键字, 同时也是 SQL92 或 SQL3 的保留关键字。这些保留关键字允许作为字段标记出现, 但不允许用作标识符。

ALL ANY ASC BETWEEN BIT BOTH  
CASE CAST CHAR CHARACTER CHECK COALESCE COLLATE COLUMN  
CONSTRAINT CROSS CURRENT CURRENT\_DATE CURRENT\_TIME  
CURRENT\_TIMESTAMP CURRENT\_USER  
DEC DECIMAL DEFAULT DESC DISTINCT  
ELSE END EXCEPT EXISTS EXTRACT  
FALSE FLOAT FOR FOREIGN FROM FULL  
GLOBAL GROUP  
HAVING  
IN INNER INTERSECT INTO IS  
JOIN  
LEADING LEFT LIKE LOCAL  
NATURAL NCHAR NOT NULL NULLIF NUMERIC  
ON OR ORDER OUTER OVERLAPS  
POSITION PRECISION PRIMARY PUBLIC  
REFERENCES RIGHT  
SELECT SESSION\_USER SOME SUBSTRING  
TABLE THEN TO TRANSACTION TRIM TRUE  
UNION UNIQUE USER  
VARCHAR  
WHEN WHERE

下面是 PostgreSQL 的保留关键字，同时也是 SQL92 或 SQL3 的保留关键字。

ADD ALTER AND AS  
BEGIN BY  
CASCADE CLOSE COMMIT CREATE CURSOR  
DECLARE DEFAULT DELETE DESC DISTINCT DROP  
EXECUTE EXISTS EXTRACT  
FETCH FLOAT FOR FROM FULL  
GRANT  
HAVING  
IN INNER INSERT INTERVAL INTO IS  
JOIN  
LEADING LEFT LIKE LOCAL  
NAMES NATIONAL NATURAL NCHAR NO NOT NULL  
ON OR OUTER  
PARTIAL PRIMARY PRIVILEGES PROCEDURE PUBLIC  
REFERENCES REVOKE RIGHT ROLLBACK  
SELECT SET SUBSTRING  
TO TRAILING TRIM

UNION UNIQUE UPDATE USING  
VALUES VARCHAR VARYING VIEW  
WHERE WITH WORK

下面列出的是 SQL92 的保留字，但不是 PostgreSQL 的保留字。注意，如果将它用作函数名，则会被自动转换为函数 CHAR\_LENGTH。

CHARACTER\_LENGTH

下面列出的是 SQL92 或 SQL3 的保留字，但不是 PostgreSQL 的保留字。但是，如果将它们用作类型名，则会被转换成一个可替换的本机类型。

BOOLEAN DOUBLE FLOAT INT INTEGER INTERVAL REAL SMALLINT

下面列出的不是任何类型的关键字，但如果在一个类型名环境中使用它们，就会被转换成一个 PostgreSQL 本机类型；如果在一个函数名环境中使用它们，则会被转换成一个本机函数。

DATETIME  
TIMESPAN

它们将会被相应地转换成 TIMESTAMP 和 INTERVAL。这个特性用于帮助向 v7.0 转换，下一个完全发布版本可能会删除这些关键字。

下面列出的是 SQL92 或 SQL3 的保留字，但不是 PostgreSQL 的保留字。这些保留字在目前的版本中没有任何使用限制，但是在将来有可能成为保留字，因此建议尽量不要使用它们。

ALLOCATE ARE ASSERTION AT AUTHORIZATION AVG  
BIT\_LENGTH  
CASCADED CATALOG CHAR\_LENGTH CHARACTER\_LENGTH COLLATION  
CONNECT CONNECTION CONTINUE CONVERT CORRESPONDING COUNT  
CURRENT\_SESSION  
DATE DEALLOCATE DEC DESCRIBE DESCRIPTOR  
DIAGNOSTICS DISCONNECT DOMAIN  
ESCAPE EXCEPT EXCEPTION EXEC EXTERNAL  
FIRST FOUND  
GET GO GOTO  
IDENTITY INDICATOR INPUT INTERSECT  
LAST LOWER  
MAX MIN MODULE  
OCTET\_LENGTH OPEN OUTPUT OVERLAPS  
PREPARE PRESERVE  
ROWS  
SCHEMA SECTION SESSION SIZE SOME  
SQL SQLCODE SQLERROR SQLSTATE SUM SYSTEM\_USER  
TEMPORARY TRANSLATE TRANSLATION  
UNKNOWN UPPER USAGE

VALUE

WHenever WRITE

## 2. 非保留关键字

SQL 92 和 SQL3 定义了一些非保留关键字，它们在语言中有受约束的含义，但是允许用作标识符。在 PostgreSQL 中，这些关键字允许用于命名字段或者表。

下面列出的是 PostgreSQL 的非保留关键字，但不是 SQL 92 和 SQL3 的非保留关键字：

ACCESS AFTER AGGREGATE

BACKWARD BEFORE

CACHE COMMENT CREATEDB CREATEUSER CYCLE

DATABASE DELIMITERS

EACH ENCODING EXCLUSIVE

FORCE FORWARD FUNCTION

HANDLER

INCREMENT INDEX INHERITS INSENSITIVE INSTEAD ISNULL

LANCOMPILER LOCATION

MAXVALUE MINVALUE MODE

NOCREATEDB NOCREATEUSER NOTHING NOTIFY NOTNULL

OIDS OPERATOR

PASSWORD PROCEDURAL

RECIPE REINDEX RENAME RETURNS ROW RULE

SEQUENCE SERIAL SHARE START STATEMENT STDIN STDOUT

TEMP TRUSTED

UNLISTEN UNTIL

VALID VERSION

下面列出的是 PostgreSQL 的非保留关键字，但它们是 SQL92 或 SQL3 的保留关键字。

ABSOLUTE ACTION

CONSTRAINTS

DAY DEFERRABLE DEFERRED

HOURL

IMMEDIATE INITIALLY INSENSITIVE ISOLATION

KEY

LANGUAGE LEVEL

MATCH MINUTE MONTH

NEXT

OF ONLY OPTION

PENDANT PRIOR PRIVILEGES

READ RELATIVE RESTRICT

SCROLL SECOND

TIME TIMESTAMP TIMEZONE\_HOUR TIMEZONE\_MINUTE TRIGGER

YEAR

ZONE

下面列出的是 PostgreSQL 的非保留关键字，同时也是 SQL92 或 SQL3 的非保留关键字。

COMMITTED SERIALIZABLE TYPE

下面列出的是 SQL92 或 SQL3 的非保留关键字，但不是任何类型的 PostgreSQL 的保留字。

ADA

C CATALOG\_NAME CHARACTER\_SET\_CATALOG CHARACTER\_SET\_NAME

CHARACTER\_SET\_SCHEMA CLASS\_ORIGIN COBOL COLLATION\_CATALOG

COLLATION\_NAME COLLATION\_SCHEMA COLUMN\_NAME

COMMAND\_FUNCTION CONDITION\_NUMBER

CONNECTION\_NAME CONSTRAINT\_CATALOG CONSTRAINT\_NAME

CONSTRAINT\_SCHEMA CURSOR\_NAME

DATA DATE\_TIME\_INTERVAL\_CODE DATE\_TIME\_INTERVAL\_PRECISION

DYNAMIC\_FUNCTION

FORTRAN

LENGTH

MESSAGE\_LENGTH MESSAGE\_OCTET\_LENGTH MORE MUMPS

NAME NULLABLE NUMBER

PAD PASCAL PLI

REPEATABLE RETURNED\_LENGTH RETURNED\_OCTET\_LENGTH

RETURNED\_SQLSTATE ROW\_COUNT

SCALE SCHEMA\_NAME SERVER\_NAME SPACE SUBCLASS\_ORIGIN

TABLE\_NAME

UNCOMMITTED UNNAMED

### 6.1.2 注释

PostgreSQL 允许在 SQL 文本中使用两种格式的注释。一种是标准的 SQL 格式：注释以两个“-”开始，直到行的结尾。例如：

```
-- 这是一行标准 SQL 注释
```

PostgreSQL 还支持 C 语言风格的块注释，注释块从“/\*”开始，直到第一个“\*/”出现时结束。例如：

```
/* C 语言风格的
   多行
   注释块
*/
```

### 6.1.3 名字

PostgreSQL 的数据库对象都有一个唯一确定的名字, PostgreSQL 对这些名字有一些特殊的规定。

任何对象的名字都必须以字母或者下划线开头, 其后的字符可以是字母、数字 (0~9) 或下划线。名字的最大长度不能超过 NAMEDATALEN-1, 默认情况下, NAMEDATALEN 为 32, 因此名字的最大长度是 31。在编译 PostgreSQL 时, 可以通过修改 `src/include/postgres_ext.h` 中的 NAMEDATALEN 的值来改变默认值。PostgreSQL 允许使用超长的名字, 但超过的部分会被截掉。

PostgreSQL 对名字中字母的大小写不敏感, 实际上, 大写字母总是被转换为小写字母。例如, FOO 和 foo 在 PostgreSQL 中是相同的对象名称。

如果名字中需要用到其他字符, 则必须将这些字符用双引号包围起来。这样, 表或字段的名字中可以包含一些非法名字字符, 如空格、&等。需要注意的是, 对于双引号中的字母, PostgreSQL 是区分大小写的。例如, “foo” 和 “Foo” 是两个不同的名字。

SQL 关键字不能用作名字, 但如果出现在双引号中, 则没有这个限制。例如, IN 是一个关键字, 而 “IN” 则是一个合法的名字。

在最新版本的 PostgreSQL 中, 允许以汉字作为表的字段名。

### 6.1.4 常量

在 PostgreSQL 中, 可以使用三种隐含常量: 字符串、整数和浮点数。也可以显式地为常量指定数据类型, 这样可以获得更精确的表示和更有效的后端处理, 这种常量称为显式常量。本节仅介绍隐含常量。

#### 1. 字符串常量

字符串常量是用单引号包围的任意 ASCII 字符序列, 例如: '这是一个字符串常量'。SQL92 允许通过两个相邻的单引号, 在字符串中嵌入单引号, 例如: '这是一个包含单引号''的字符串'。在 PostgreSQL 中, 单引号可以通过用反斜杠转义的方法来嵌入, 例如: '这是一个包含单引号\'的字符串'。与 C 语言一样, 反斜杠在 PostgreSQL 是字符转义标志, 因此, 如果字符串常量中需要使用反斜杠, 则必须用 “\\” 的形式来转义, 例如: '这是一个包含反斜杠\\的字符串'。反斜杠同样可以用于其他不可打印字符的嵌入, 例如: '\tab', 在字符串常量中嵌入制表符。

#### 2. 整数常量

整数常量是一个没有小数点的数字集合, 合法的数值范围为 -2147483648 到 +2147483647。这个范围会因操作系统和主机硬件的不同而不同。

对于更大的整数, 则必须使用 SQL92 字符串符号或 PostgreSQL 类型符号声明为 int8。例如, int8 '4000000000' (字符串风格的大整数) '4000000000':int8 (通过 PostgreSQL 类型转换的整数)。

## 6 基本 SQL 命令

### 3. 浮点数常量

浮点数常量包含一个整数部分、一个小数点和一个小数部分，或者使用科学记数法。格式为：

```
{dig}.{dig} [e [+ -] {dig}]
```

在 PostgreSQL 中，浮点数常量的类型是 float8。float4 类型的浮点数常量可以用 SQL92 字符串表示，或者利用 PostgreSQL 类型表示法显式地定义。例如，float4 '1.23'（字符串风格）'1.23'::float4（PostgreSQL 类型转换）。

### 4. 用户定义类型的常量

任何类型的常量都可以用下面任何一种方法表示。

```
type 'string'
'string'::type
CAST 'string' AS type
```

### 5. 数组常量

数组常量是任意 PostgreSQL 类型的数组。数组常量的通用格式是：

```
{val1 delim val2 delim}
```

其中，delim 是 pg\_type 系统表中定义的该类型的分隔符。对于内建类型，分隔符是逗号。例如：

```
{{1,2,3},{4,5,6},{7,8,9}}
```

这个常量是一个 3\*3 的二维数组，数组的元素为整数。

## 6.2 创建表

与所有的关系数据库一样，在 PostgreSQL 中，表是保存数据最基本的单位。

### 1. 命令语法

创建表的命令是标准的 SQL 命令 CREATE TABLE。

```
CREATE [ TEMPORARY | TEMP ] TABLE table (
    column type
    [ NULL | NOT NULL ] [ UNIQUE ] [ DEFAULT value ]
    [ column_constraint_clause | PRIMARY KEY ] [ ... ] ]
    [, ... ]
    [, PRIMARY KEY ( column [, ...] ) ]
    [, CHECK ( condition ) ]
    [, table_constraint_clause ]
```

```
) [ INHERITS ( inherited_table [, ...] ) ]
```

CREATE TABLE 向当前数据库中追加一个新的表,新表为执行本命令的用户所有。关于这个命令的详细介绍参见本书的附录,有关字段的合法类型将在后续章节介绍。本命令只是建立表的结构,即表的框架,表的数据则由另外的命令管理。

PostgreSQL 对表的基本限制是:一个表的字段数目不能超过 1600 个,每一个字段的大小不能超过 8192 字节(大对象字段除外),用户表不能与系统表同名。

## 2. 例子

下面的例子创建本书的第一个表 friend。

--创建表 friend

```
testdb=> CREATE TABLE friend (  
testdb(>   name CHAR(8),           --姓名  
testdb(>   city CHAR(10),         --城市  
testdb(>   prov CHAR(6),          --省份  
testdb(>   age INTEGER);          --年龄  
CREATE
```

以上是实时地在psql中输入命令、创建表的方法。也可以利用文本编辑器将SQL命令保存在一个文本文件中,然后利用psql的专有命令“\i 文件名”执行存放在文件中的SQL命令。例如,假定建立表friend的SQL存放在文件friend.sql中,内容是:

```
CREATE TABLE friend (  
name CHAR(8),  
city CHAR(10),  
prov CHAR(6),  
age INTEGER);
```

然后,在psql中执行命令:

```
testdb=> \i friend.sql  
CREATE
```

上述两种方式执行命令的结果是一样的。第一种方式的特点是简单直接,但如果在输入命令的过程中出现输入错误,则不容易更正。后一种方式在实际工作中更常用,原因是可以方便地重复执行同一条命令,也便于更正错误和移植。

在上面的SQL命令中,涉及到PostgreSQL字段的数据类型。与其他数据库系统相比,PostgreSQL的字段数据类型更加丰富,可扩展性更强,有关的详细介绍见第7章。这里用到的CHAR和INTEGER类型是PostgreSQL最基本的数据类型,也是SQL标准所定义的类型。前者表示字符串类型,后者表示整数类型。

## 3. 显示表结构

psql的专有命令“\d 表名称”用于显示表的结构。例如:

```
testdb=> \d friend  
Table "friend"
```



## 6

## 基本 SQL 命令

Attribute	Type	Modifier
name	char(8)	
city	char(10)	
prov	char(6)	
age	integer	

专有命令 “\dt” 用于列出当前数据库中所有的用户表。例如：

```
testdb=> \dt
      List of relations
  Name | Type | Owner
-----+-----+-----
 friend | table | testuser
(1 row)
```

## 6.3 插入数据

为表添加数据的方法有很多种，最典型的方法是利用 INSERT 命令向表中插入数据。

### 1. 命令语法

```
INSERT INTO table [ ( column [, ...] ) ]
    { VALUES ( expression [, ...] ) | SELECT query }
```

目标列表中的字段可以按任何顺序排列。对于目标列表中没有出现的字段，如果为该字段定义了默认值，则插入默认值；否则，插入 NULL。如果向定义为 NOT NULL 的字段插入 NULL 值，PostgreSQL 将拒绝整个插入操作。如果表达式的数据类型不正确，系统将试图进行自动的类型转换。执行本命令的用户必须具有该表的插入权限，同样，也必须具有查询权限，以便处理 WHERE 子句中指定的任何表。

### 2. 例子

```
testdb=> INSERT INTO friend VALUES(
testdb(> '张祥',
testdb(> '武汉',
testdb(> '湖北',
testdb(> 18);
INSERT 19648 1
```

如果数据插入成功，psql 会输出一条类似于 “INSERT 19648 1” 的信息。其中，“INSERT” 表示当前执行的命令是 INSERT 命令；“19648” 是 PostgreSQL 为当前记录分

配的对象标识编号( OID ),这个编号在所有表记录中唯一,因此可以唯一地代表一条记录;  
“1”表示插入记录的数目。

下面的命令插入本书将要用到的一些数据。

```
testdb=> INSERT INTO friend VALUES(  
testdb(> '李志建',  
testdb(> '大冶',  
testdb(> '湖北',  
testdb(> 27);  
INSERT 19649 1  
testdb=> INSERT INTO friend VALUES(  
testdb(> '李江',  
testdb(> '广州',  
testdb(> '广东',  
testdb(> 24);  
INSERT 19650 1
```

## 6.4 查询数据

从表中查询数据最典型的命令是 SELECT。SELECT 命令的功能非常强大,本节仅仅介绍最基本的功能,其他更复杂的用法将在后续章节介绍。

### 1. 命令语法

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]  
      expression [ AS name ] [, ...]  
      [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table ]  
      [ FROM table [ alias ] [, ...] ]  
      [ WHERE condition ]  
      [ GROUP BY column [, ...] ]  
      [ HAVING condition [, ...] ]  
      [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]  
      [ ORDER BY column [ ASC | DESC | USING operator ] [, ...] ]  
      [ FOR UPDATE [ OF class_name [, ...] ] ]  
LIMIT { count | ALL } [ { OFFSET | , } start ]
```

这条命令从一个或多个表、视图中返回满足 WHERE 条件的所有数据行。如果省略了 WHERE 子句,命令返回表中的所有行。

本命令也可用于调用 PostgreSQL 的函数。在这种情况下,命令与任何的表无关,只是

# 6

## 基本 SQL 命令

执行函数并返回结果。例如：

```
testdb=> SELECT sin(1.5);
sin
-----
0.997494986604054
(1 row)
```

### 2. 查询所有数据

```
testdb=> SELECT * FROM friend;
name | city | prov | age
-----+-----+-----+-----
张祥 | 武汉 | 湖北 | 18
李志建 | 大冶 | 湖北 | 27
李江 | 广州 | 广东 | 24
(3 rows)
```

由于没有指定任何条件，因此，上面的命令返回 friend 表中的所有数据。

### 3. 查询满足条件的数据

下面的例子查询所有 age 超过 20 的记录。

```
testdb=> SELECT *
testdb-> FROM friend
testdb-> WHERE age>=20;
name | city | prov | age
-----+-----+-----+-----
李志建 | 大冶 | 湖北 | 27
李江 | 广州 | 广东 | 24
(2 rows)
```

下面的例子查询所有 prov 为“湖北”、age 在 20 以下的记录。这是一个使用复杂条件表达式的 SELECT 查询。

```
testdb=> SELECT *
testdb-> FROM friend
testdb-> WHERE prov='湖北' AND age<20;
name | city | prov | age
-----+-----+-----+-----
张祥 | 武汉 | 湖北 | 18
(1 row)
```

### 4. 查询指定的字段

许多情况下，查询结果并不需要表中的所有字段，而只是需要特定的几个字段。这时，

可以在 SELECT 命令动词的后面 指定查询所需要的字段名。例如 , 下面的查询仅列出 name 和 age 字段。

```
testdb=> SELECT name,age
testdb-> FROM friend;
  name | age
-----+-----
  张祥 |  18
  李志建 |  27
  李江 |  24
(3 rows)
```

### 5. 改变字段标识

默认情况下, 输出结果中的第一行列出了结果集中的所有字段名。有时, 字段名并不能很好地说明字段的意义。PostgreSQL 允许为字段名指定另外的字段标识字符串。例如:

```
testdb=> SELECT name AS 姓名,
testdb-> age AS 年龄
testdb-> FROM friend;
  姓名 | 年龄
-----+-----
  张祥 |  18
  李志建 |  27
  李江 |  24
(3 rows)
```

### 6. 模糊查询

下面的例子查询所有的 name 字段以“李”字开头的记录。

```
testdb=> SELECT *
testdb-> FROM friend
testdb-> WHERE name LIKE '李%';
  name | city | prov | age
-----+-----+-----+-----
  李志建 | 大冶 | 湖北 |  27
  李江 | 广州 | 广东 |  24
(2 rows)
```

## 6.5 控制查询输出格式

正如在前一节所看到的，默认情况下，SELECT 查询的输出格式为对齐的表格格式。第一行为字段名，第二行为分隔符，其后是按对齐格式排列的数据。可以利用 psql 的专有命令改变查询输出格式，这在某些脚本形式的应用中非常有用。这里介绍的输出格式控制命令不仅影响 SELECT 命令，而且影响其他 SQL 命令和专有命令。

### 1. 字段名标识

默认情况下，查询结果的第一行为字段名标识，第二行为分隔行。可以使用专有命令“\t”取消或设置这两行。例如：

```
testdb=> \t
Showing only tuples.
testdb=> SELECT * FROM friend;
张祥      | 武汉      | 湖北      | 18
李志建    | 大冶      | 湖北      | 27
李江      | 广州      | 广东      | 24
```

“\t”是开关形式的命令。默认情况下，使用一次“\t”命令，关闭字段名标识和分隔行；再使用一次“\t”，可打开字段名标识和分隔行。例如：

```
testdb=> \t
Tuples only is off.
testdb=> SELECT * FROM friend;
   name   |   city   |   prov   | age
-----+-----+-----+-----
张祥      | 武汉      | 湖北      | 18
李志建    | 大冶      | 湖北      | 27
李江      | 广州      | 广东      | 24
(3 rows)
```

### 2. 对齐

默认情况下，查询结果按对齐方式排列。可以使用专有命令“\a”取消或设置对齐方式。例如：

```
testdb=> \a
Output format is unaligned.
testdb=> SELECT * FROM friend;
```

```

name|city|prov|age
张祥   |武汉   |湖北   |18
李志建 |大冶   |湖北   |27
李江   |广州   |广东   |24
(3 rows)

```

从结果可以看出，字段名标识与结果数据不再按对齐方式排列。

“\a”也是开关形式的命令。默认情况下，使用一次“\a”命令关闭对齐方式；再使用一次“\a”可打开对齐方式。

### 3. 表标题

默认情况下，输出结果中不含表的标题。可以使用专有命令“\C 表标题”为查询的输出结果指定一个标题。例如：

```

testdb=> \C 这是本书的第一个表 friend
Title is "这是本书的第一个表 friend".
testdb=> SELECT * FROM friend;
      这是本书的第一个表 friend
name  | city  | prov | age
-----+-----+-----+-----
张祥   | 武汉   | 湖北   | 18
李志建 | 大冶   | 湖北   | 27
李江   | 广州   | 广东   | 24
(3 rows)

```

不带任何参数调用“\C”命令，取消表标题的设置。例如：

```

testdb=> \C
Title is unset.
testdb=> SELECT * FROM friend;
name  | city  | prov | age
-----+-----+-----+-----
张祥   | 武汉   | 湖北   | 18
李志建 | 大冶   | 湖北   | 27
李江   | 广州   | 广东   | 24
(3 rows)

```

### 4. HTML 格式

默认情况下，查询结果以标准表格的形式输出。可以使用专有命令“\H”将输出结果转换为 HTML 表格代码。例如：

```

testdb=> \H
Output format is html.
testdb=> SELECT * FROM friend;

```

```

<table border=1>
  <tr>
    <th align=center>name</th>
    <th align=center>city</th>
    <th align=center>prov</th>
    <th align=center>age</th>
  </tr>
  <tr valign=top>
    <td align=left>张祥    </td>
    <td align=left>武汉    </td>
    <td align=left>湖北    </td>
    <td align=right>18</td>
  </tr>
  <tr valign=top>
    <td align=left>李志建  </td>
    <td align=left>大冶    </td>
    <td align=left>湖北    </td>
    <td align=right>27</td>
  </tr>
  <tr valign=top>
    <td align=left>李江    </td>
    <td align=left>广州    </td>
    <td align=left>广东    </td>
    <td align=right>24</td>
  </tr>
</table>
(3 rows)<br>

```

“\H”也是一个开关命令，使用它可以在普通表格和 HTML 表格之间进行切换。

## 5. 扩展格式

默认情况下，查询结果以标准表格的形式输出。可以使用专有命令“\x”将结果格式转换为扩展格式。例如：

```

testdb=> \x
Expanded display is on.
testdb=> SELECT * FROM friend;
-[ RECORD 1 ]----
name | 张祥
city | 武汉
prov | 湖北

```

```

age | 18
-[ RECORD 2 ]----
name | 李志建
city | 大冶
prov | 湖北
age | 27
-[ RECORD 3 ]----
name | 李江
city | 广州
prov | 广东
age | 24

```

重复使用一次 “\x” 命令，结果格式重新变为标准的表格格式。

## 6. 表格形式的控制

对于表格形式的输出结果，可以使用专有命令 “\pset” 来控制其外观格式。

(1) \pset format 格式。控制输出的格式，可用的格式有：unaligned（非对齐）、aligned（对齐）、HTML（HTML 表格）、latex（latex 格式）。最后一种格式为 LaTeX 编辑器特有的文件格式，可更精确地排列文本。例如：

```

testdb=> \pset format latex
Output format is latex.
testdb=> SELECT * FROM friend;
\begin{tabular}{l|l|l|r}
name & city & prov & age \\
\hline
张祥 & 武汉 & 湖北 & 18 \\
李志建 & 大冶 & 湖北 & 27 \\
李江 & 广州 & 广东 & 24 \\
\end{tabular}

```

(3 rows) \\

(2) \pset border 数字。设置表格的边界。其中，如果数字为 0，不显示表格边界；如果数字为 1，显示表格的内分隔线；如果数字为 2，显示所有的表格线。

```

testdb=> \pset border 0
Border style is 0.
testdb=> SELECT * FROM friend;
name      city      prov  age
-----
张祥      武汉      湖北   18
李志建    大冶      湖北   27

```



## 6 基本 SQL 命令

```
李江      广州      广东      24
```

```
(3 rows)
```

```
testdb=> \pset border 1
```

```
Border style is 1.
```

```
testdb=> SELECT * FROM friend;
```

name	city	prov	age
张祥	武汉	湖北	18
李志建	大冶	湖北	27
李江	广州	广东	24

```
(3 rows)
```

```
testdb=> \pset border 2
```

```
Border style is 2.
```

```
testdb=> SELECT * FROM friend;
```

name	city	prov	age
张祥	武汉	湖北	18
李志建	大冶	湖北	27
李江	广州	广东	24

```
(3 rows)
```

(3) \pset expanded。在标准显示格式和扩展显示格式之间进行切换，功能同专有命令“\x”。

(4) \pset null 字符串。对于没有任何值的字段，默认情况下它的值显示为“null”。可以使用这个命令改变默认显示方式。例如：

```
testdb=> INSERT INTO friend
```

```
testdb-> (name,age)
```

```
testdb-> values('阳阳',4);
```

```
INSERT 19660 1
```

```
testdb=> SELECT * FROM friend;
```

name	city	prov	age
张祥	武汉	湖北	18
李志建	大冶	湖北	27
李江	广州	广东	24
阳阳			4

```
(4 rows)
```

```
testdb=> \pset null 空值
```

Null display is "空值".

```
testdb=> SELECT * FROM friend;
```

```

  name |    city    | prov | age
-----+-----+-----+-----
  张祥  |    武汉    | 湖北 |  18
  李志建 |    大冶    | 湖北 |  27
  李江   |    广州    | 广东 |  24
  阳阳   |    空值    | 空值 |   4
(4 rows)
```

(5) \pset fieldsep 字符串。设置非对齐方式下的字段分隔符，默认的分隔符为“|”。例如：

```
testdb=> \a
```

Output format is unaligned.

```
testdb=> \pset fieldsep ','
```

Field separator is ','.

```
testdb=> SELECT * FROM friend;
```

```

name,city,prov,age
张祥  ,武汉      ,湖北  ,18
李志建 ,大冶      ,湖北  ,27
李江   ,广州      ,广东  ,24
阳阳   ,空值,空值,4
(4 rows)
```

分隔字符串中可以包含任意的字符，包括以“\”转义的字符。

(6) \pset recordsep 字符串。设置非对齐方式下的行分隔，默认的行分隔符为“\n”，即换行符号。例如：

```
testdb=> \pset fieldsep '|'
```

Field separator is '|'.

```
testdb=> \pset recordsep '\n=====\n'
```

Record separator is '

====='.

'.

```
testdb=> SELECT * FROM friend;
```

```

name|city|prov|age
=====
张祥  |武汉    |湖北 |18
=====
李志建 |大冶    |湖北 |27
=====
李江   |广州    |广东 |24
```

## 6 基本 SQL 命令

=====

阳阳 | 空值 | 空值 | 4

=====

(4 rows)

(7) \pset tuples\_only。同 “\t” 命令。

(8) \pset title HTML 页面标题。如果结果输出格式为 HTML 表格，可以使用这个命令指定页面标题。

(9) \pset tableattr 表标题。同 “\C” 命令。

(10) \pset pager。分页或不分页显示查询结果。

## 6.6 删除数据

### 1. 命令语法

```
DELETE FROM table [ WHERE condition ]
```

这条命令从指定的表 table 中删除满足 condition (条件) 的数据行。如果 WHERE condition 子句不存在，命令将删除表中所有行，结果是一个有效的空表。执行本命令的用户必须有写权限，同样也必须有读表的权限，这样才能对符合条件的行进行读取操作。

### 2. 例子

为了保证后面的例子有数据可供操作，本节的删除命令将放在一个临时表上进行。下面的命令利用 friend 表的结构和数据，创建一个新的表 tmp。

```
testdb=> SELECT * INTO TABLE tmp FROM friend;
```

```
SELECT
```

这样，tmp 表的结构与 friend 相同，并且含有与 friend 表一样的数据。

```
testdb=> SELECT * FROM tmp;
```

name	city	prov	age
张祥	武汉	湖北	18
李志建	大冶	湖北	27
李江	广州	广东	24
阳阳	空值	空值	4

(4 rows)

下面的例子删除 tmp 表中所有 age 小于 20 的记录。

```
testdb=> DELETE FROM tmp WHERE age<=20;
```

```
DELETE 2
```

```
testdb=> SELECT * FROM tmp;
  name | city | prov | age
-----+-----+-----+-----
李志建 | 大冶 | 湖北 | 27
李江   | 广州 | 广东 | 24
(2 rows)
```

下面的例子删除 tmp 中所有的记录。

```
testdb=> DELETE FROM tmp;
DELETE 2
testdb=> SELECT * FROM tmp;
  name | city | prov | age
-----+-----+-----+-----
(0 rows)
```

## 6.7 修改数据

### 1. 命令语法

```
UPDATE table SET col = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

本命令改变满足条件的所有数据行中指定字段的值。执行此命令的用户必须具有写数据的权限，对 WHERE 条件中涉及的任何表也必须具有读权限。

### 2. 例子

下面的例子将表 friend 中所有记录的 age 字段的值加 1。

```
testdb=> SELECT * FROM friend;
  name | city | prov | age
-----+-----+-----+-----
张祥   | 武汉 | 湖北 | 18
李志建 | 大冶 | 湖北 | 27
李江   | 广州 | 广东 | 24
阳阳   | 空值 | 空值 | 4
(4 rows)

testdb=> UPDATE friend
testdb-> SET age = age + 1;
```

## 6

## 基本 SQL 命令

UPDATE 4

```
testdb=> SELECT * FROM friend;
```

name	city	prov	age
张祥	武汉	湖北	19
李志建	大冶	湖北	28
李江	广州	广东	25
阳阳	空值	空值	5

(4 rows)

下面的例子将 name 为“阳阳”的记录中的 city 改为“武汉”、prov 改为“湖北”。

```
testdb=> UPDATE friend
```

```
testdb-> SET city = '武汉',
```

```
testdb-> prov = '湖北'
```

```
testdb-> WHERE name = '阳阳';
```

UPDATE 1

```
testdb=> SELECT * FROM friend;
```

name	city	prov	age
张祥	武汉	湖北	19
李志建	大冶	湖北	28
李江	广州	广东	25
阳阳	武汉	湖北	5

(4 rows)

## 6.8 删除表

### 1. 命令语法

```
DROP TABLE name [, ...]
```

只有表的所有者才能执行本命令。如果被删除的表有从属索引，它们将首先被删除。

### 2. 例子

下面的例子删除 6.6 节中建立的临时表 tmp。

```
testdb=> DROP TABLE tmp;
```

DROP

## 第7章

# 数据类型

---

在介绍PostgreSQL更深入的使用和开发技术之前，先介绍一些PostgreSQL基本语言元素，包括数据类型、函数、操作符和表达式。本章主要介绍数据类型，内容包括：

- 基本数据类型

- 数组类型

- 大对象类型

- 数据类型转换

- 预定义变量



## 7.1 基本数据类型

PostgreSQL 的数据类型非常丰富,并且用户也可以自行定义任意多的数据类型,因此,可以说 PostgreSQL 支持无限多的数据类型。这些数据类型又称为安装数据类型(Installed Type)或内建数据类型,以便与用户自定义的数据类型区别开来。

### 7.1.1 字符串类型

字符串类型是一种最常用的数据类型。这种类型的变量可以存放任意的字母、数字、标点符号和其他合法的字符。如,人的名字、通信地址、电子邮件地址等字符串类型的数据。

表 7-1 列出了 PostgreSQL 支持的四种不同字符串类型:TEXT、VARCHAR(length)、CHAR 和 CHAR(length)。其中,TEXT 类型不限制字符串的数目,通常用于描述长度变化较大或长度不可预知的数据字段,如雇员的简历、公文正文等。VARCHAR 类型用于描述长度不超过 length 个字符的可变长字符串。CHAR 用于描述单字节的字段。CHAR(length)与 VARCHAR 类似,同样限制字符串的长度,但 CHAR(length)在任何情况下都存放 length 个字符到字符串中,对于长度不足 length 的字符串,用空格补足到 length 个字符。一般情况下,对 CHAR(length)的处理速度要比对 TEXT 和 VARCHAR 的处理速度稍快一些。

表 7-1 字符串类型

类型	存储空间	说明
CHAR	1 字节	单字节,SQL 92 兼容
CHAR(length)	4+length 字节	定长,不足部分以空格补齐,SQL 92 兼容
TEXT	4+x 字节	变长
VARCHAR(length)	4+length 字节	变长,但有长度限制,SQL 92 兼容

PostgreSQL 还有一种定长字符串类型:name。该类型主要用于 PostgreSQL 内部,为数据库对象定义名字,通常不允许一般用户使用。该类型长度为 32 个字符,但可以在编译 PostgreSQL 时,通过改变 NAMEDATALEN 变量的值来重新定义。

### 7.1.2 数值类型

数值类型是另外一种最常用的数据类型。这种类型的变量只能存放由数字、小数点和正负号组成的合法数据,如 127、15.38、-1480 等。PostgreSQL 支持的数值类型见表 7-2。

表7-2

数值类型

类型	存储空间	说明
DECIMAL	可变	用户指定的精度，最多 8000 个数字
FLOAT4	4 字节	可变精度浮点数，6 位有效数字
FLOAT8	8 字节	可变精度浮点数，15 位有效数字
INT2	2 字节	固定精度整数，表示范围为-32768 至+32767
INT4、INTEGER	4 字节	固定精度整数，表示范围为-2147483648 至+2147483647
INT8	8 字节	极大范围的固定精度整数。范围为+、-18 位数字
NUMERIC	可变	用户指定的精度，长度没有限制
SERIAL	4 字节	标识或交叉索引，范围为 0 至+2147483647

INTEGER、INT2 和 INT8 用于描述存放整型数值的变量。在内部存储结构上，INTEGER 类型的变量占用 4 个字节的存储容量，可表示的数据范围为  $\pm 2^{31}$ ；INT2 类型的变量占用 2 个字节，可表示的数据范围为  $\pm 2^{15}$ ；INT8 类型的变量占用 8 个字节，可表示的数据范围为  $\pm 4 \times 10^{18}$ 。可表示的数据范围越大，占用的存储空间越多，处理速度也越慢。数值类型对应有一套完整的数学操作符和函数。

NUMERIC(precision,decimal)用于描述用户自定义精度的实数变量。其中，precision 为有效数据位数的数目，这个数目中包含小数位数、整数位数和小数点位数（位数为 1）。在所有的数值类型中，这种类型的处理速度最慢。

FLOAT（同 FLOAT8）和 FLOAT4 用于描述浮点类型变量，前者的精度为 15 位数据，后者的精度为 6 位数据。与 NUMERIC 类型有所不同的是，FLOAT 和 FLOAT4 的小数点是分开存放的，因此可以表示巨大的数值，如 4.78145e+32。FLOAT 和 FLOAT4 的运算速度通常比 NUMERIC 快，但在计算时可能会产生影响精确度的舍入，这是因为许多十进制数值无法精确地用 FLOAT 或 FLOAT4 来表示。如果要保证足够的精确度，最好使用 NUMBER 类型。

SERIAL 类型是 PostgreSQL 用其他类型构造出来的一种特殊类型。典型的用途是创建表的唯一标识。在当前的版本中，下面的命令：

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

等同于：

```
CREATE SEQUENCE tablename_colname_seq;
```

```
CREATE TABLE tablename
```

```
    (colname INT4 DEFAULT nextval('tablename_colname_seq'));
```

```
CREATE UNIQUE INDEX tablename_colname_key on tablename (colname);
```

在删除一个包含 SERIAL 类型的表时，隐含的支持序列不会被自动删除。因此下面按顺序执行的命令是无效的：

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

```
DROP TABLE tablename;
```

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

除非显式地使用 DROP SEQUENCE 命令，否则序列不会被删除，而是一直存放在数据库中。



### 7.1.3 时间类型

时间类型用于描述存放日期、时间和时间间隔信息的变量，表 7-3 列出了 PostgreSQL 支持的时间类型。

表 7-3 数值类型

类型	存储空间	说明
TIMESTAMP	8 字节	包含日期和时间，精度为毫秒，14 位数字
INTERVAL	12 字节	时间间隔，精度为毫秒
DATE	4 字节	只包含日期数据，精度为 1 天
TIME	4 字节	只包含时间数据，精度为 1 毫秒

DATE 类型用于描述日期变量，包括年、月、日，它的格式由 DATESTYLE 配置项的值来决定。例如：

```
testdb=> SHOW DATESTYLE;
NOTICE: DateStyle is ISO with US (NonEuropean) conventions
SHOW VARIABLE
testdb=> SET DATESTYLE TO 'SQL, EUROPEAN';
SET VARIABLE
testdb=> SHOW DATESTYLE;
NOTICE: DateStyle is SQL with European conventions
SHOW VARIABLE
testdb=> RESET DATESTYLE;
RESET VARIABLE
testdb=> SHOW DATESTYLE;
NOTICE: DateStyle is ISO with US (NonEuropean) conventions
SHOW VARIABLE
```

TIME 类型用于描述时间数据变量，包括时、分、秒，三者之间用“:”分隔，例如：

```
testdb=> SELECT CURRENT_TIME;
?column?
-----
00:20:15
(1 row)
```

TIMESTAMP 类型用于描述时间戳数据变量，包括年、月、日和时、分、秒，例如：

```
testdb=> SELECT CURRENT_TIMESTAMP;
?column?
-----
2000-11-01 00:21:34+08
(1 row)
```

INTERVAL 类型用于描述时间间隔变量，包括年、月、日和时、分、秒等分量。

### 7.1.4 逻辑类型

逻辑类型 BOOLEAN 用于描述逻辑变量。一个逻辑类型的变量只能存放两个值：true 和 false，因此只占用 1 字节的存储空间。在 PostgreSQL 中，t、yes、y 和 1 均可以看作是 true 的同义词；而 f、no、n 和 0 可以看作是 false 的同义词。

### 7.1.5 几何类型

几何类型是 PostgreSQL 特有的数据类型，主要用于描述存放平面二维几何图元参数的变量。表 7-4 列出了 PostgreSQL 支持的几何类型。

表 7-4 几何数据类型

类型	存储空间	说明
POINT	16 字节	平面上的一点，格式为(x,y)
LINE	32 字节	平面上的一条直线，格式为((x1,y1),(x2,y2))
LSEG	32 字节	平面上的一条线段，格式为((x1,y1),(x2,y2))
BOX	32 字节	平面上的一个矩形，格式为((x1,y1),(x2,y2))
PATH	4+32*n 字节	平面上的一条闭合路径，格式为((x1,y1),...)
PATH	4+32*n 字节	平面上的一条开放路径，格式为[(x1,y1),...]
POLYGON	4+32*n 字节	平面上的一个多边形，格式为((x1,y1),...)
CIRCLE	24 字节	平面上的一个圆，格式为<(x,y),r>

PostgreSQL 提供了一系列丰富的函数和操作符，利用它们可以进行各种几何计算，如度量、转换、旋转和相交计算等。

#### 1. POINT（点）

点是最基本的平面图形构造单位。点的描述语法是：

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

其中：x 是用一个浮点数表示的 x 坐标；y 是用一个浮点数表示的 y 坐标。

#### 2. LSEG（线段）

线段用一对点的坐标来描述，格式是：

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

其中参数(x1,y1)和(x2,y2) 是线段的两个端点。Line（直线）与此类型相似。

#### 3. BOX（矩形）

矩形用两个对角点的坐标来描述，格式是：

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

其中，(x1,y1)是矩形左下角坐标；(x2,y2)是矩形右上角坐标。

## 4. PATH (路径)

路径由一系列的点连接而成。路径可能是“开放”的，也就是第一个点和最后一个点没有连接；路径也可能是“封闭”的，这时第一个点和最后一个点相连。PostgreSQL 提供的函数 `popen(p)` 和 `pclose(p)` 用来选择路径的开放和闭合。

PATH 的语法格式是：

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

其中，(x,y)是路径经过的点的坐标。“[”表明所定义的路径为开放路径；“(”表明定义的路径为封闭路径。

## 5. POLYGON (多边形)

多边形实际上是一种闭合路径，只是存储方式不同而且有自己的一套过程和函数。语法格式是：

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

其中参数(x,y)是多边形的顶点。

## 6. CIRCLE (圆)

圆由一个圆心和一个半径来描述，语法格式是：

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

其中，(x,y)为圆心的坐标，r 为圆的半径。

## 7.1.6 网络类型

网络类型主要用于描述网络设备的地址信息，表 7-5 列出了 PostgreSQL 支持的网络类型。

表7-5 网络数据类型

类型	存储空间	说明
CIDR	可变	CIDR 格式的 IP v4 地址
INET	可变	IP v4 格式的 IP 地址或主机名

## 1. CIDR

CIDR 类型定义一个 CIDR 网络，格式为 “x.x.x.x/y”。其中，“x.x.x.x” 为 IP 地址，而 “/y” 为网络掩码中 1 的个数。如果 “/y” 部分没有指明，那么掩码部分用旧的有类网络分类假设进行计算。

## 2. INET

INET 类型用来描述主机名称或地址的字段。

除上面介绍的数据类型外，PostgreSQL 还定义了许多供系统内部使用的数据类型。在 psql 中用 “\dT” 命令可以列出所有的数据类型。例如：

```
testdb=> \dT
```

List of types	
Type	Description
SET	set of tuples
abstime	absolute, limited-range date and time (Unix system time)
aclitem	access control list
bit	fixed-length bit string
bool	boolean, 'true'/'false'
box	geometric box '(lower left,upper right)'
bpchar	char(length), blank-padded string, fixed storage length
bytea	variable-length string, binary values escaped
char	single character
cid	command identifier type, sequence in transaction id
cidr	network IP address/netmask, network address
circle	geometric circle '(center,radius)'
date	ANSI SQL date
filename	filename used in system tables
float4	single-precision floating point number, 4-byte storage
float8	double-precision floating point number, 8-byte storage
inet	IP address/netmask, host address, netmask optional
int2	-32 thousand to 32 thousand, 2-byte storage
int2vector	array of 16 int2 integers, used in system tables
int4	-2 billion to 2 billion integer, 4-byte storage
int8	~18 digit integer, 8-byte storage
interval	@ <number> <units>, time interval
line	geometric line '(pt1,pt2)'
lseg	geometric line segment '(pt1,pt2)'
lztext	variable-length string, stored compressed
macaddr	XX:XX:XX:XX:XX, MAC address

money	\$d,ddd.cc, money
name	31-character type for storing system identifiers
numeric	numeric(precision, decimal), arbitrary precision number
oid	object identifier(oid), maximum 4 billion
oidvector	array of 16 oids, used in system tables
path	geometric path '(pt1,...)'
point	geometric point '(x, y)'
polygon	geometric polygon '(pt1,...)'
regproc	registered procedure
reltime	relative, limited-range time interval (Unix delta time)
smgr	storage manager
text	variable-length string, no limit specified
tid	(Block, offset), physical location of tuple
time	hh:mm:ss, ANSI SQL time
timestamp	date and time
timetz	hh:mm:ss, ANSI SQL time
tinterval	(abstime,abstime), time interval
unknown	
varbit	fixed-length bit string
varchar	varchar(length), non-blank-padded string, variable storage length
xid	transaction id

(47 rows)

## 7.2 数组类型

数组型字段是 PostgreSQL 的一个独创，这种字段可以用于存放多个简单类型的数据。数组字段可以是一维的，也可以是二维的，甚至可以是多维的。

### 1. 定义方法

与一般字段的唯一区别是，在定义数组字段时必须用方括号指定数组的维数。例如：

```
testdb=> CREATE TABLE test_arr_1 (
testdb(>   col integer[5]
testdb(> );
```

CREATE

上面例子定义的字段是固定长度的数组，也可以定义非固定长度的数组。例如：

```
testdb=> CREATE TABLE test_arr_2 (  
testdb(>   col integer[][]  
testdb(> );  
CREATE
```

在这个例子中，字段 col 是二维数组，并且数组两个维的长度都是不固定的。

另外，还可以将数组字段的部分维定义为固定长度，部分维定义为非固定长度。例如：

```
testdb=> CREATE TABLE test_arr_3 (  
testdb(>   col integer[2][2][]  
testdb(> );  
CREATE
```

在上面的例子中，字段 col 的第一维和第二维的长度是固定的，而第三维的长度是可变的。

## 2. 数据的插入

对于一维数组字段，插入数据的方法是：

'{元素 1,元素 2,......,元素 n}'

例如：

```
testdb=> INSERT INTO test_arr_1  
testdb-> VALUES (  
testdb(>   '{1,2,3,4,5}'  
testdb(> )  
testdb-> ;  
INSERT 19739 1  
testdb=> SELECT * FROM test_arr_1;  
      col
```

```
-----  
{1,2,3,4,5}
```

```
(1 row)
```

也可以只为数组字段指定部分元素。下面的例子仅指定数组字段 col 的第一个和第二个字段。

```
testdb=> INSERT INTO test_arr_1  
testdb-> VALUES('{1,2}');  
INSERT 19740 1  
testdb=> SELECT * FROM test_arr_1;  
      col
```

```
-----  
{1,2,3,4,5}
```

```
{1,2}
```

```
(2 rows)
```

对于二维数组字段，插入数据的方法是：

```
'{{元素 11,元素 12,...},{元素 21,元素 22,...},..., {元素 n1,元素 n2,...}}
```

例如：

```
testdb=> INSERT INTO test_arr_2
testdb-> VALUES(
testdb(> ' {{11,12},{21,22},{31,32}} '
testdb(> )
testdb-> ;
INSERT 19741 1
testdb=> SELECT * FROM test_arr_2;
      col
-----
 {{11,12},{21,22},{31,32}}
(1 row) }
```

多维数组字段的数据插入方式类似，每增加一维，数据字符串中的“{、}”增加一层。

### 3. 数组字段的引用

在表达式中，可以整体地引用一个数组字段，例如：

```
testdb=> SELECT * FROM test_arr_1
testdb-> WHERE col='{1,2}';
      col
-----
 {1,2}
(1 row)
```

也可以直接引用数组字段中的一个元素，例如：

```
testdb=> SELECT * FROM test_arr_1
testdb-> WHERE col[3]=3;
      col
-----
 {1,2,3,4,5}
(1 row)
```

### 4. 用途

对于长度不固定、对数据类型有特别要求的数据，数组字段特别有用。例如，为描述每一天的天气情况，在传统的数据库中，可以采用下列两种方式之一创建表。

```
CREATE TABLE tbl (
    dt date,
    weather char(10)
);
```

```
CREATE TABLE tb2 (  
    year integer,  
    mon integer,  
    weather1 char(10),  
    weather2 char(10),  
    .....  
    weather31 char(10),  
);
```

无论采用哪种方式，都会有较大的数据冗余。对于第一种方式，日期中的年份和月份会被重复地存储多次。对于第二种方式，由于并不是每个月都有31天，因此数据冗余度也较大。如果采用PostgreSQL的数组字段，则可以将数据的冗余度减到最低限度。例如：

```
CREATE TABLE tb3 (  
    year integer,  
    mon integer,  
    weather char(10)[]  
);
```

数组类型为描述可变长数据提供了非常有效的手段，如果妥善使用，将能够有效地减少数据冗余，提高代码的可读性。

## 7.3 大对象类型

在PostgreSQL中，单个字段的最大容量为8096个字节；另外，二进制数据也便于利用单引号插入到数据库中。为了解决这些问题，PostgreSQL提供了一种用于存储超大型文本和二进制数据的类型——大对象类型（BLOB，Binary Large Object）。利用大对象类型字段，可以存储任何形式的文件，如图像、文本文件、二进制文件等。

### 1. 定义方法

可以利用保留字 `OID` 定义大对象字段。例如：

```
testdb=> CREATE TABLE person (  
testdb(>   name char(10),  
testdb(>   photo OID);  
CREATE
```

### 2. 大对象的引用

可以利用PostgreSQL的函数 `lo_import()` 将大对象插入到数据库中，也可以利用 `lo_export()` 函数从数据库中取出大对象。例如：



## 7

## 数据类型

```
testdb=> INSERT INTO person
testdb-> VALUES ('钟立', lo_import('/usr/images/zhli.jpg'));
INSERT 19750 1
testdb=> SELECT lo_export(person.photo, '/tmp/outimage.jpg')
testdb-> FROM person
testdb-> WHERE name = '钟立';
lo_export
-----
1
(1 row)
test=> SELECT lo_unlink(person.photo) FROM person;
lo_unlink
-----
1
(1 row)
```

在上面的例子中，函数`lo_import()`将文件`/usr/images/zhli.jpg`插入到数据库中。每一个大对象在PostgreSQL中都有一个唯一的对象标识号OID，`lo_import()`函数实际上是为导入的大对象创建一个OID，并返回这个OID。字段`photo`中存放的是大对象的OID，而不是大对象本身，大对象本身存放在`person.photo`文件中。

函数`lo_export()`根据OID从数据库中取出大对象，并将其存入文件`/tmp/outimage.jpg`。如果大对象被成功地取出，函数返回1。函数`lo_unlink()`用以删除大对象。

由于数据库服务器和客户端工作在不同的目录中，因此所有的文件名必须包含完全路径。

由于大对象使用本地文件系统，因此通过网络连接的用户不能利用`lo_import()`和`lo_export()`函数操纵大对象。而应该使用psql的`lo_import`和`lo_export`命令。

## 7.4 数据类型转换

PostgreSQL 对数据类型的要求不像某些高级语言(如 PASCAL)那样严格，许多情况下允许不同数据类型的变量在同一个表达式中进行运算。尽管如此，在一些特殊情况下，仍然需要进行数据类型转换。

PostgreSQL 提供的通用数据类型转换命令为 `CAST`，基本用法为：

`CAST(<原变量/常量> AS <新数据类型>)`

或

`<原变量/常量> :: <新数据类型>`

例如：

```
testdb=> CREATE TABLE CA
testdb-> XCHAR char(4),
testdb-> XDATE date,
testdb-> XINT integer);
CREATE
testdb=> INSERT INTO CASTTEST VALUES (
testdb-> '12.1',
testdb-> '2000-11-1',
testdb-> 15);
INSERT 22859 1
testdb-> SELECT 'DATE:=' || XDATE FROM CASTTEST;
ERROR:  There is more than one possible operator '||' for types 'unknown'
and 'date'
```

You will have to retype this query using an explicit cast

上面例子的最后一条命令在执行时出现了问题，原因是错误地将字符常量与日期字段 XDATE 执行了字符串拼接操作。为了避免错误，必须进行数据类型转换。例如：

```
testdb=> SELECT 'DATE:=' || CAST(XDATE AS TEXT) FROM CASTTEST;
?column?
-----
DATE:=Wed Nov 01 00:00:00 2000 CST
(1 row)
```

除通用 CAST 命令外，PostgreSQL 为不同的数据类型提供了不同的数据类型转换函数，有关细节请参考第 8 章。

## 7.5 预定义变量

PostgreSQL 定义了一些变量，用于描述服务器的当前时间和用户。

### 1. CURRENT\_DATE

这个变量跟踪数据库服务器的当前日期。例如：

```
testdb=> SELECT CURRENT_DATE;
date
-----
2001-02-03
(1 row)
```

### 2. CURRENT\_TIME

这个变量跟踪数据库服务器的当前时间。例如：

```
testdb=> SELECT CURRENT_TIME;
      time
-----
 13:46:06
(1 row)
```

### 3 . CURRENT\_TIMESTAMP

这个变量跟踪数据库服务器的当前时间戳（包括日期和时间）。例如：

```
testdb=> SELECT CURRENT_TIMESTAMP;
      timestamp
-----
2001-02-03 13:47:33+08
(1 row)
```

### 4 . CURRENT\_USER

这个变量的值是当前连接数据库服务器的 PostgreSQL 用户名。例如：

```
testdb=> SELECT CURRENT_USER;
      getpgusername
-----
      testuser
(1 row)
```

## 第8章

# 函数

---

PostgreSQL提供了大约940个函数，功能涉及数据操作的方方面面。利用它们，可以对表的字段或常量进行各种计算和操作。本章主要以表格的形式列出以下函数：

- SQL 函数
- 数学函数
- 字符串函数
- 日期时间函数
- 格式化函数
- 几何函数
- 网络函数



## 8.1 SQL 函数

SQL 函数由 SQL 92 标准定义，它们的语法与一般函数相同，但实现方法有所不同。  
表 8-1 列出了 PostgreSQL 所支持的 SQL 函数。

表8-1		SQL函数	
函数	返回值	说明	例子
COALESCE(list)	非 NULL	返回列表 list 中第一个非 NULL 值	COALESCE(c1+1, c2 + 5, 0)
IFNULL(input,non-NULL substitute)	非 NULL	如果第一个参数为 NULL, 返回第二个参数, 否则返回第一个参数	IFNULL(c1, 'N/A')
CASE WHEN expr THEN expr [...] ELSE expr END	expr	返回逻辑值为真的子句所对应的表达式	CASE WHEN c1 = 1 THEN 'match' ELSE 'no match' END

## 8.2 数学函数

表8-2		数学函数	
函数	返回值类型	说明	例子
abs(float8)	float8	计算绝对值	abs(-17.4)
degrees(float8)	float8	将弧度转换为角度	degrees(0.5)
exp(float8)	float8	求 e 的指数幂	exp(2.0)
ln(float8)	float8	计算自然对数	ln(2.0)
log(float8)	float8	计算以 10 为底的对数	log(2.0)
pi()	float8	圆周率常量	pi()
pow(float8,float8)	float8	对指定的底数, 求指定指数的幂	pow(2.0, 16.0)
radians(float8)	float8	将角度转换为弧度	radians(45.0)
round(float8)	float8	取整数	round(42.4)
sqrt(float8)	float8	求平方根	sqrt(2.0)
cbirt(float8)	float8	求立方根	cbirt(27.0)
trunc(float8)	float8	向零的方向, 截断浮点数	trunc(42.4)
float(int)	float8	将整数转换为浮点数	float(2)
float4(int)	float4	将整数转换成 float4 型的浮点数	float4(2)
integer(float)	int	将浮点数转换为整数	integer(2.0)
acos(float8)	float8	计算反余弦	acos(10.0)
asin(float8)	float8	计算反正弦	asin(10.0)
atan(float8)	float8	计算反正切	atan(10.0)
atan2(float8,float8)	float8	计算反余切	atan3(10.0,20.0)
cos(float8)	float8	计算余弦	cos(0.4)
cot(float8)	float8	计算余切	cot(20.0)
sin(float8)	float8	计算正弦	cos(0.4)
tan(float8)	float8	计算正切	tan(0.4)

表 8-2 中列出的用于 FLOAT8 的大部分函数同样可以用于 NUMERIC 类型。

## 8.3 字符串函数

表8-3 字符串函数

函数	返回值类型	说明	例子
char_length(string)	int4	求字符串的长度	char_length('jose')
character_length(string)	int4	求字符串的长度，同 char_length 函数	char_length('jose')
lower(string)	string	将字符串中的大写字母转换为小写字母	lower('TOM')
octet_length(string)	int4	求字符串的存储长度	octet_length('jose')
position(string in string)	int4	求子字符串在字符串中的位置	position('o' in 'Tom')
substring(string [from int] [for int])	string	取子字符串	substring('Tom' from 2 for 2)
trim([leading trailing both] [string] from string)	string	删除字符串中首部和（或）尾部指定的字符串	trim(both 'x' from 'xTomx')
upper(text)	text	将字符串中的小写字母转换为大写字母	upper('tom')
char(text)	char	将 text 类型转换为 char 类型	char('text string')
char(varchar)	char	将 varchar 类型转换为 char 类型	char(varchar 'varchar string')
initcap(text)	text	将每个单词的首字母转为大写字母	initcap('thomas')
lpad(text,int,str)	text	用字符串 str 将字符串 text 从左边填充到指定的长度	lpad('hi',4,'??')
ltrim(text,text)	text	将字符串左边指定字符删除	ltrim('xxxxtrim','x')
textpos(text,str)	text	计算 str 在 text 中的位置	position('high','ig')
rpadd(text,int,str)	text	用字符串 str 将字符串 text 从右边填充到指定的长度	rpadd('hi',4,'x')
rtrim(text,text)	text	将字符串右边指定字符删除	rtrim('trimxxxx','x')
substr(text,int[,int])	text	取子字符串	substr('hi there',3,5)
text(char)	text	将 char 类型转换为 text 类型	text('char string')
text(varchar)	text	将 varchar 类型转换为 text 类型	text(varchar 'varchar string')
translate(text,from,to)	text	将字符串 text 中的 from 字符串转换为 to 字符串	translate('12345', '1', 'a')
varchar(char)	varchar	将 char 类型转换为 varchar 类型	varchar('char string')
varchar(text)	varchar	将 text 类型转换为 varchar 类型	varchar('text string')

大部分字符串函数可用于 text、varchar()和 char() 类型。

## 8.4 日期时间函数

表8-4 日期时间函数

函数	返回值类型	说明	例子
abstime(timestamp)	abstime	将时间戳转换为绝对时间	abstime(timestamp 'now')
age(timestamp)	interval	取时间戳中的月和年	age(timestamp '1957-06-13')
age(timestamp,timestamp)	interval	取时间戳中的月和年	age('now', timestamp '1957-06-13')

续表

函数	返回值类型	说明	例子
date_part(text,timestam p)	float8	取时间戳中的日期	date_part('dow',timestam p 'now')
date_part(text,interval)	float8	取时间戳中的时间	date_part('hour',interval '4 hrs 3 mins')
date_trunc(text,timesta mp)	timestamp	截断日期	date_trunc('month',absti me 'now')
interval(reftime)	interval	将实时时间转换为时间间隔	interval(reftime '4 hours')
isfinite(timestamp)	bool	判断指定的时间是否为有效时间	isfinite(timestamp 'now')
isfinite(interval)	bool	判断指定的时间间隔是否为有效时间	isfinite(interval '4 hrs')
reftime(interval)	reftime	将时间间隔类型转换为 reftime 类型	reftime(interval '4 hrs')
timestamp(date)	timestamp	将日期类型转换为时间戳类型	timestamp(date 'today')
timestamp(date,time)	timestamp	将日期和时间合成为时间戳类型	timestamp(timestamp '1998-02-24',time '23:07');
to_char(timestamp,text)	text	将时间戳转换为字符串	to_char(timestamp '1998-02-24','DD');

对于 date\_part 和 date\_trunc 函数而言,参数 text 可以是'year'、'month'、'day'、'hour'、'minute'和'second',还可以是更特殊的单位'decade'、'century'、'millenium'、'millisecond'和'microsecond'。date\_part 允许使用'dow'返回某天在一个星期中的序数 (date of week); 用'week'返回 ISO 所定义的一年中的星期;用'epoch'返回自 1970(对于 timestamp 类型)以来的秒数,或用'epoch'返回总共流逝的秒数(对于 interval 类型)。

## 8.5 格式化函数

PostgreSQL 格式化函数提供了一套有效的工具,用于把各种数据类型(日期/时间、int、float 和 numeric 等)转换为格式化的字符串,以及将格式化的字符串转换为指定类型的数据。

表8-5 格式化函数

函数	返回值类型	说明	例子
to_char(timestamp, text)	text	将时间戳转换为字符串	to_char(timestamp 'now','HH12:MI:SS')
to_char(int, text)	text	将 int4 或 int8 型整数转换为字符串	to_char(125, '999')
to_char(float, text)	text	将 float4 或 float8 类型的浮点数转换为字符串	to_char(125.8, '999D99')
to_char(numeric, text)	text	将 numeric 类型的实数转换为字符串	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')
to_timestamp(text, text)	date	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 类型的实数	to_number('12,454.8-', '99G999D9S')

所有格式化函数的第二个参数都是用于转换的模板,各种转换模板见表 8-6、表 8-7 和表 8-8。

表8-6

日期时间转换模板

模板	说明
HH	一天的小时数 (01-12)
HH12	一天的小时数 (01-12)
HH24	一天的小时数 (00-23)
MI	分 (00-59)
SS	秒 (00-59)
SSSS	午夜后的秒 (0-86399)
AM or A.M. or PM or P.M.	正午标识 (大写)
am or a.m. or pm or p.m.	正午标识 (小写)
Y,YYY	带逗号的年 (4 位和更多位)
YYYY	年 (4 位和更多位)
YYY	年的后三位
YY	年的后两位
Y	年的最后一位
BC or B.C. or AD or A.D.	年标识 (大写)
bc or b.c. or ad or a.d.	年标识 (小写)
MONTH	全长大写月份名 (9 字符)
Month	全长混合大小写月份名 (9 字符)
month	全长小写月份名 (9 字符)
MON	大写月份名缩写 (3 字符)
Mon	混合大小写的月份名缩写 (3 字符)
mon	小写的月份名缩写 (3 字符)
MM	月份 (01-12)
DAY	全长大写日期名 (9 字符)
Day	全长混合大小写日期名 (9 字符)
day	全长小写日期名 (9 字符)
DY	大写的日期名缩写 (3 字符)
Dy	混合的大小写日期名缩写 (3 字符)
dy	小写的日期名缩写 (3 字符)
DDD	一年中的天(001-366)
DD	一个月中的天(01-31)
D	一个星期中的天(1-7, 星期日=1)
W	一个月中的星期数
WW	一年中的星期数
CC	世纪 (2 位)
J	Julian 日期 (自公元前 4712 年 1 月 1 日来的日期)
Q	季度
RM	罗马数字的月份 (I-XII; I=JAN) - 大写
rm	罗马数字的月份 (I-XII; I=JAN) - 小写

表8-7

日期时间转换模板的后缀

后缀	说明	例子
FM	填充模式前缀	FMMonth
TH	大写顺序数后缀	DDTH
th	小写顺序数后缀	DDTH
FX	固定模式全局选项 (见下面)	FX Month DD Day
SP	拼写模式 (还未实现)	DDSP

如果没有使用 FX 选项, `to_timestamp` 和 `to_date` 忽略空格。FX 必须是模板的第一部分。反斜杠必须用 “\ ” 转义的形式来插入, 例如 “\HH\MI\SS”。双引号之间的字符串被忽略并且不被分析。如果要输出双引号, 必须在双引号前面放置一个双反斜杠, 例如 “\\”YYYY Month\\””。`to_char` 支持不带前导双引号的文本, 但是在双引号之间的任何字符串会被迅速



处理并且还保证不会被当作模板关键字来解释，例如"Hello Year: "YYYY"。

表8-8 用于to\_char(numeric)的模板

模板	说明
9	指定数据位数
0	纯小数的小数点前冠以数字 0
. (句点)	小数点
, (逗号)	分组(千)分隔符
PR	尖括号内负值
S	带负号的负值(支持本地化)
L	货币符号(支持本地化)
D	小数点(支持本地化)
G	分组分隔符(支持本地化)
MI	在指定位置插入负号(如果数字 < 0)
PL	在指定位置插入正号(如果数字 > 0)
SG	在指定位置插入正/负号
RN	罗马数字(在 1 和 3999 之间)
TH or th	转换成序数
V	将参数乘以 10 <sup>n</sup> , 其中 n 是 V 后数字的个数
EEEE	科学记数, 目前版本不支持

'SG'、'PL'或'MI'模板总是使正负号出现在结果的第一个字符的位置，而不是紧邻数字。

模板'S'使正负号紧邻数字。例如：

```
testdb=> SELECT to_char(-12,'SG9999');
```

```
to_char
```

```
-----
```

```
- 12
```

```
(1 row)
```

```
testdb=> SELECT to_char(-12,'S9999');
```

```
to_char
```

```
-----
```

```
-12
```

```
(1 row)
```

在模板中，9 对应结果中的一个数字。如果没有可用的数字，那么用一个空格补充。TH 不转换小于零的数值，也不转换小数，它是一个 PostgreSQL 扩展。利用 V 可以方便地将入口数值乘以 10<sup>n</sup>，这里 n 是跟在 V 后面的 9 的个数。例如：

```
testdb=> SELECT to_char(12,'999999999V999');
```

```
to_char
```

```
-----
```

```
12000
```

```
(1 row)
```

to\_char 不支持将 V 与一个小数点捆绑在一起使用，例如，'99.9V99'是不允许的。表 8-9 是 to\_char 函数的各种使用实例。

表8-9

to\_char应用实例

例子	输出结果
to_char(now(), 'Day, HH12:MI:SS')	'Tuesday , 05:39:18'
to_char(now(), 'FMDay, HH12:MI:SS')	'Tuesday, 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-decimal: "999" Post-decimal: " .999')	'Pre-decimal: 485 Post-decimal: .800'
to_char(12, '99V999')	'12000'
to_char(12.4, '99V999')	'12400'
to_char(12.45, '99V9')	'125'

## 8.6 几何函数

表8-10

几何函数

函数	返回值类型	说明	例子
area(object)	float8	求目标的面积	area(box '((0,0),(1,1))')
box(box, box)	box	求两个矩形相交的部分	box(box '((0,0),(1,1))', box '((0.5,0.5),(2,2))')
center(object)	point	求图形对象的中心	center(box '((0,0),(1,2))')
diameter(circle)	float8	返回圆直径	diameter(circle '((0,0),(2,0))')
height(box)	float8	求矩形的垂直高度	height(box '((0,0),(1,1))')
isclosed(path)	bool	判断一个路径是否为封闭路径	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	bool	判断一个路径是否为开放路径	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	float8	求一个对象的长度	length(path '((-1,0),(1,0))')
pclose(path)	path	将路径转换为封闭路径	popen(path '[(0,0),(1,1),(2,0)]')
npoint(path)	int4	计算路径中的点的数目	npointh(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	将路径转换为开放路径	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	float8	返回圆的半径	radius(circle '((0,0),(2,0))')
width(box)	float8	计算矩形的宽度	width(box '((0,0),(1,1))')

表8-11 几何转换函数

函数	返回值类型	说明	例子
box(circle)	box	将圆转换为矩形	box('((0,0),2,0)::circle)
box(point,point)	box	将点转换为矩形	box('(0,0)::point,(1,1)::point)
box(polygon)	box	将多边形转换为矩形	box('((0,0),(1,1),(2,0))::polygon')
circle(box)	circle	将矩形转换为圆	circle('((0,0),(1,1))::box')
circle(point,float8)	circle	将点和一个浮点数转换为圆	circle('(0,0)::point,2,0)
lseg(box)	lseg	将矩形转换为线段	lseg('((-1,0),(1,0))::box')
lseg(point,point)	lseg	将点转换为线段	lseg('(-1,0)::point,(1,0)::point')
path(polygon)	point	将多边形转换为路径	path('((0,0),(1,1),(2,0))::polygon')
point(circle)	point	将圆心转换为点	point('((0,0),2,0)::circle')
point(lseg,lseg)	point	将两个线段的相交处转换为点	point('((-1,0),(1,0))::lseg,'((-2,-2),(2,2))::lseg')
point(polygon)	point	将多边形的中心转换为点	point('((0,0),(1,1),(2,0))::polygon')
polygon(box)	polygon	将矩形转换为多边形	polygon('((0,0),(1,1))::box')
polygon(circle)	polygon	将圆转换为多边形	polygon('(0,0),2,0)::circle')
polygon( <i>pts</i> ,circle)	polygon	将点和圆转换为多边形	polygon(12,'(0,0),2,0)::circle')
polygon(path)	polygon	将路径转换为多边形	polygon('(0,0),(1,1),(2,0))::path')

表8-12 几何升级函数

函数	返回值类型	说明	例子
isoldpath(path)	path	测试是否 v6.1 前的路径	isoldpath('(1,3,0,0,1,1,2,0)::path')
revertpoly(polygon)	polygon	转换 v6.1 前的多边形	revertpoly('((0,0),(1,1),(2,0))::polygon')
upgradeopath(path)	path	升级 v6.1 前的路径	upgradeopath('(1,3,0,0,1,1,2,0)::path')
upgradeopoly(polygon)	polygon	升级 v6.1 前的多边形	upgradeopoly('(0,1,2,0,1,0)::polygon')

## 8.7 网络函数

表8-13 网络函数

函数	返回值类型	说明	例子
broadcast(cidr)	text	从 CIDR 格式的地址中取广播地址	broadcast('192.168.1.5/24')
broadcast(inet)	text	从 INET 格式的地址中取广播地址	broadcast('192.168.1.5/24')
host(inet)	text	从 INET 格式的地址中取主机地址	host('192.168.1.5/24')
masklen(cidr)	int4	从 CIDR 格式的地址中计算网络掩码的长度	masklen('192.168.1.5/24')
masklen(inet)	int4	从 INET 格式的地址中计算网络掩码的长度	masklen('192.168.1.5/24')
netmask(inet)	text	从 INET 格式的地址中取网络掩码	netmask('192.168.1.5/24')

以上列出的仅仅是一些常用的函数，其他函数可在 psql 中用 “\df” 列出。除了这些标准的函数外，PostgreSQL 还允许用户自行定义函数。

## 第9章

# 运算符和表达式

---

种类丰富、功能强大的运算符和表达式为用户提供了操作数据的强有力工具，这是PostgreSQL的一个主要特色。本章主要包括：

各种运算符

各种表达式



## 9.1 运算符简介

### 9.1.1 种类

PostgreSQL 提供了大量的运算符，用于进行各种计算，运算符也可以称为操作符。在目前的版本中，运算符的数量大约有 560 个，在 psql 中利用 “\do” 命令可以列出所有预定义的运算符，这些运算符在系统表 pg\_operator 中定义。在 pg\_operator 中，每一条记录定义一个运算符，包括运算符的实现过程名和输入、输出类型的对象标识号（OID）。

例如，如果想了解运算符 “||”（连接字符串）的所有变种，可以使用下面命令：

```
testdb=> SELECT oprleft,oprright,oprresult,opcode
testdb-> FROM pg_operator
testdb-> WHERE oprname = '||';
oprleft | oprright | oprresult | opcode
```

```
-----+-----+-----+-----
      25 |      25 |      25 | textcat
    1042 |    1042 |    1042 | textcat
    1043 |    1043 |    1043 | textcat
    1560 |    1560 |    1560 | bitcat
    1562 |    1562 |    1562 | varbitcat
```

(5 rows)

其中，oprleft 是左操作数类型的 OID，oprright 是右操作数类型的 OID，oprresult 是运算结果类型的 OID，opcode 是实现运算符功能的过程名或函数名。操作数和计算结果类型的 OID 在系统表 pg\_type 中定义，可以利用类似于下面的命令查询与某个类型 OID 对应的类型名。

```
testdb=> SELECT typename FROM pg_type
testdb-> WHERE OID = 25;
typename
-----
text
(1 row)
testdb=> SELECT typename FROM pg_type
testdb-> WHERE OID = 1042;
typename
```

```
-----  
bpchar  
(1 row)
```

从结果可以看出,OID 为 25 的类型名是“text”,而 OID 为 1042 的类型名是“bpchar”。不难查出,运算符“||”的操作数和结果可以是类型“text”、“bpchar”、“varchar”、“bit”和“varbit”。由于 PostgreSQL 没有提供关于运算符的帮助文档,因此可以利用上述查询方法得出某个运算符的有关细节。

PostgreSQL 的运算符大致可以分为五大类:通用运算符、数学运算符、几何运算符、时间间隔运算符和网络运算符。除此以外,用户还可以自行定义运算符。

### 9.1.2 优先级

与大多数高级语言一样,PostgreSQL 不同的运算符有不同的优先级。在目前的版本中,运算符的优先级“硬编码”于系统的内部,因此无法动态地改变某个运算符的优先级。表 9-1 列出了常用运算符的优先级。

表9-1 运算符优先级

运算符	优先级	说明
UNION	左	联合两个查询结果
::		类型转换
[]	左	数组分隔符
.	左	表名称与字段名称间的分隔符
-	右	一元减操作
:	右	幂
	左	间隔开始
* / %	左	乘、除、取模
+ -	左	加、减
IS		测试 TRUE、FALSE、NULL
ISNULL		测试 NULL
NOTNULL		测试 NOT NULL
其他操作符		系统和用户自定义的运算符
IN		设置成员(set membership)
BETWEEN		判断一个对象是否在一个范围之内
OVERLAPS		判断时间间隔是否重叠
LIKE		字符串模式匹配
<>		判断两个对象是否不相等
=	右	判断两个对象是否相等
NOT	右	逻辑非
AND	左	逻辑与
OR	左	逻辑或

## 9.2 通用运算符

通用运算符可以作用于大多数类型的数据,表 9-2 列出了常用的通用运算符。

表9-2

通用运算符

运算符	说明	例子
<	如果左边的对象小于右边的对象，结果为真；否则，结果为假	1 < 2
<=	如果左边的对象小于或等于右边的对象，结果为真；否则，结果为假	1 <= 2
<>	如果左边的对象不等于右边的对象，结果为真；否则，结果为假	1 <> 2
=	如果左边的对象等于右边的对象，结果为真；否则，结果为假	1 = 1
>	如果左边的对象大于右边的对象，结果为真；否则，结果为假	2 > 1
>=	如果左边的对象大于或等于右边的对象，结果为真；否则，结果为假	2 >= 1
	连接两个字符串	'Postgre'    'SQL'
!=	如果左边的对象不在右边的对象之中，结果为真；否则，结果为假	3 != i
~~	如果左边的对象类似于右边的对象，结果为真；否则，结果为假	'scrappy,marc,hermit' ~~ '%scrappy%'
!~~	如果左边的对象不类似于右边的对象，结果为真；否则，结果为假	'bruce' !~~ '%al%'
~	如果左边的对象匹配于(大小写相关)右边的规则表达式，结果为真；否则，结果为假	'thomas' ~ '.*thomas.*'
~*	如果左边的对象匹配于(大小写无关)右边的规则表达式，结果为真；否则，结果为假	'thomas' ~* '.*Thomas.*'
!~	如果左边的对象不匹配于(大小写相关)右边的规则表达式，结果为真；否则，结果为假	'thomas' !~ '.*Thomas.*'
!~*	如果左边的对象不匹配于(大小写无关)右边的规则表达式，结果为真；否则，结果为假	'thomas' !~* '.*vadim.*'

## 9.3 数值运算符

数值运算符主要作用于各种数值类型的数据，如整数、浮点数、实数等。

表9-3

数值运算符

运算符	说明	例子
!	阶乘	3 !
!!	阶乘 (左操作符)	!! 3
%	取模	5 % 4
%	取整	% 4.5
*	乘	2 * 3
+	加	2 + 3
-	减	2 - 3
/	除	4 / 2
: 或 exp()	自然幂	: 3.0
@	绝对值	@ -5.0
^	求幂	2.0 ^ 3.0
/	平方根	/ 25.0
/	立方根	/ 27.0

操作符 “:” 可能在新的版本中删除，取而代之的是函数 exp()。

## 9.4 几何运算符

几何运算符主要作用于各种几何类型的数据，如点、直线、多边形等。

表9-4 几何运算符

运算符	说明	例子
+	将几何对象向右平移	'((0,0),(1,1))::box + '(2,0,0)::point
-	将几何对象向左平移	'((0,0),(1,1))::box - '(2,0,0)::point
*	将几何对象伸缩或旋转（顺时针）	'((0,0),(1,1))::box * '(2,0,0)::point
/	将几何对象伸缩或旋转（逆时针）	'((0,0),(2,2))::box / '(2,0,0)::point
#	计算两个矩形的交	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	计算多边形顶点数	# '(1,0),(0,1),(-1,0)'
##	最近点	'(0,0)::point ## '(2,0),(0,2)::lseg
&&	判断两个对象是否重叠	'((0,0),(1,1))::box && '((0,0),(2,2))::box
&<	判断第一个对象是否与第二个对象在左边重叠	'((0,0),(1,1))::box &< '((0,0),(2,2))::box
&>	判断第一个对象是否与第二个对象在右边重叠	'((0,0),(3,3))::box &> '((0,0),(2,2))::box
<->	计算两个对象的距离	'((0,0),1)::circle <-> '((5,0),1)::circle
<<	判断第一个对象是否在第二个对象的左边	'((0,0),1)::circle << '((5,0),1)::circle
<^	判断第一个对象是否低于第二个对象	'((0,0),1)::circle <^ '((0,5),1)::circle
>>	判断第一个对象是否在第二个对象的右边	'((5,0),1)::circle >> '((0,0),1)::circle
>^	判断第一个对象是否高于第二个对象	'((0,5),1)::circle >^ '((0,0),1)::circle
?#	判断两个对象是否相交或重叠	'((-1,0),(1,0))::lseg ?# '((-2,-2),(2,2))::box;
?-	判断两个点是否在同一条水平线上	'(1,0)::point ?- '(0,0)::point
?	判断两条线是否垂直	'((0,0),(0,1))::lseg ?  '((0,0),(1,0))::lseg
@-@	计算几何对象的长度或周长	@-@ '((0,0),(1,0))::path
?	判断两个对象是否垂直	'(0,1)::point ?  '(0,0)::point
?	判断两个对象是否平行	'((-1,0),(1,0))::lseg ?   '((-1,2),(1,2))::lseg
@	判断一个对象是否包含在另一个对象中	'(1,1)::point @ '((0,0),2)::circle
@@	取圆心	@@ '((0,0),10)::circle
~=	判断两个对象是否为同一个图形	'((0,0),(1,1))::polygon ~= '((1,1),(0,0))::polygon

## 9.5 时间间隔运算符

时间间隔运算符主要作用于时间间隔（interval）类型的数据。

表9-5 时间间隔运算符

运算符	说明
#<	判断左边的时间间隔是否小于右边的时间间隔
#<=	判断左边的时间间隔是否小于或等于右边的时间间隔
#<>	判断左边的时间间隔是否不等于右边的时间间隔
#=	判断左边的时间间隔是否等于右边的时间间隔
#>	判断左边的时间间隔是否大于右边的时间间隔
#>=	判断左边的时间间隔是否大于或等于右边的时间间隔



续表

运算符	说明
<#>	将对象转换成时间间隔
<<	判断左边的时间间隔是否小于指定的值
	间隔开始
~=	判断两个时间间隔是否相同
<?>	判断一个时间是否在时间间隔之内

## 9.6 网络运算符

网络运算符主要作用于网络类型的数据，表 9-6 列出了适用于 CIDR 类型网络地址的运算符，而表 9-7 列出的是 INET 类型的运算符。

表9-6

IP V4 CIDR运算符

运算符	说明	例子
<	判断左边的地址是否小于右边的地址	'192.168.1.5'::cidr < '192.168.1.6'::cidr
<=	判断左边的地址是否小于或等于右边的地址	'192.168.1.5'::cidr <= '192.168.1.5'::cidr
=	判断左边的地址是否等于右边的地址	'192.168.1.5'::cidr = '192.168.1.5'::cidr
>=	判断左边的地址是否大于或等于右边的地址	'192.168.1.5'::cidr >= '192.168.1.5'::cidr
>	判断左边的地址是否大于右边的地址	'192.168.1.5'::cidr > '192.168.1.4'::cidr
<>	判断左边的地址是否不等于右边的地址	'192.168.1.5'::cidr <> '192.168.1.4'::cidr
<<	判断左边的地址是否包含于右边的地址	'192.168.1.5'::cidr << '192.168.1/24'::cidr
<=<	判断左边的地址是否包含于或等于右边的地址	'192.168.1/24'::cidr <=< '192.168.1/24'::cidr
>>	判断左边的网络是否包含右边的地址	'192.168.1/24'::cidr >> '192.168.1.5'::cidr
>=>	判断左边的网络是否包含或等于右边的地址	'192.168.1/24'::cidr >=> '192.168.1/24'::cidr

表9-7

IP V4 INET运算符

运算符	说明	例子
<	判断左边的地址是否小于右边的地址	'192.168.1.5'::inet < '192.168.1.6'::inet
<=	判断左边的地址是否小于或等于右边的地址	'192.168.1.5'::inet <= '192.168.1.5'::inet
=	判断左边的地址是否等于右边的地址	'192.168.1.5'::inet = '192.168.1.5'::inet
>=	判断左边的地址是否大于或等于右边的地址	'192.168.1.5'::inet >= '192.168.1.5'::inet
>	判断左边的地址是否大于右边的地址	'192.168.1.5'::inet > '192.168.1.4'::inet
<>	判断左边的地址是否不等于右边的地址	'192.168.1.5'::inet <> '192.168.1.4'::inet
<<	判断左边的地址是否包含于右边的地址	'192.168.1.5'::inet << '192.168.1/24'::inet
<=<	判断左边的地址是否包含于或等于右边的地址	'192.168.1/24'::inet <=< '192.168.1/24'::inet
>>	判断左边的地址是否包含右边的地址	'192.168.1/24'::inet >> '192.168.1.5'::inet
>=>	判断左边的地址是否包含或等于右边的地址	'192.168.1/24'::inet >=> '192.168.1/24'::inet

## 9.7 表达式

表达式一般用于计算、限制条件，主要的 SQL 语句中都可以使用表达式。前面的例子或多或少地涉及了各种表达式，相信读者对 PostgreSQL 的表达式已经有了一个感性的认

识。因此，这里只介绍与表达式有关的一般规定。

PostgreSQL 的表达式由常量、字段、变量、参数、运算符和函数组成。

### 9.7.1 常量表达式

常量是最简单的表达式。例如：

```
testdb=# SELECT '这是常量表达式';
      ?column?
-----
  这是常量表达式
(1 row)
```

### 9.7.2 字段表达式

字段表达式主要出现在 SELECT 命令中，用于取出表中字段的值。例如：

```
testdb=# SELECT name FROM friend;
      name
-----
    张祥
    李志建
    李江
    阳阳
(4 rows)
```

### 9.7.3 函数表达式

函数表达式是一个合法 SQL 函数的名称，在其后的括号中指定函数的入口参数列表。  
基本形式是：

```
function_name (a_expr [, a_expr ... ] )
```

其中，a\_expr 也是表达式。

例如：

```
testdb=# SELECT sin(1.5);
      sin
-----
  0.997494986604054
(1 row)
testdb=# SELECT char_length(name)
testdb=# FROM friend;
 char_length
-----
```

```

      8
      8
      8
      8
(4 rows)

```

有关于常用函数的简要介绍可参见本书的第 7 章。

### 9.7.4 聚集表达式

聚集表达式将多个输入缩减为单个输出值，例如，对输入的求和、平均。聚集表达式主要由聚集函数组成，主要的语法格式有：

聚集函数名 (表达式)

聚集函数名 (ALL 表达式)

聚集函数名 (DISTINCT 表达式)

聚集函数名 (\*)

其中，表达式是不包含聚集函数的任意表达式。

第一种形式的聚集表达式对表达式的值进行聚集计算，生成一个非空值；第二种形式与第一种相同，因为 ALL 是默认值。第三种形式对相异的非空值进行聚集计算；最后一种形式进行表数据行数量的统计，不管该行是空还是非空值。由于没有指定输入参数，因此这种形式通常只是对 count() 聚集函数有用。

下面的例子计算 age 字段的总和。

```

testdb=# SELECT sum(age)
testdb=# FROM friend;

sum
-----
  77
(1 row)

```

下面的例子统计表的记录数目。

```

testdb=# SELECT count(*)
testdb=# FROM friend;

count
-----
     4
(1 row)

```

### 9.7.5 复合表达式

可以利用运算符将常量、变量、字段和函数组合为一个复合表达式。例如：

```

testdb=# SELECT 1+2;

?column?

```

```

-----
          3
(1 row)

testdb=# SELECT * FROM friend
testdb=# WHERE name LIKE '李%' AND age>20;
   name |   city   | prov | age
-----+-----+-----+-----
 李志建 |   大冶    | 湖北 |  28
   李江 |   广州    | 广东 |  25
(2 rows)

```

### 9.7.6 目标列表表达式

目标列表表达式用于指定查询的目标，它由一个或多个用逗号分隔的元素组成。每个元素都必须为下面形式：

```
a_expr [ AS result_attname ]
```

其中，`a_expr` 是查询的目标，可以是常量、变量、字段或表达式；`result_attname` 是查询目标的别名。如果不指定 `result_name`，则以 `a_expr` 作为目标的名称。

例如：

```
testdb=# SELECT sin(1.5) AS 正弦值;
      正弦值
-----
0.997494986604054
(1 row)
```

```
testdb=# SELECT sin(1.5);
      sin
-----
0.997494986604054
(1 row)
```

```
testdb=# SELECT name AS 姓名,
testdb=#      age AS 年龄
testdb=# FROM friend;
   姓名 |  年龄
-----+-----
   张祥 |    19
 李志建 |    28

```

```
李江      |    25
阳阳      |     5
(4 rows)
```

### 9.7.7 FROM 列表表达式

FROM 列表表达式指定查询语句的数据来源，通常是表或视图的名字。基本语法格式是：

```
[ class_reference ] instance_variable
    {, [ class_ref ] instance_variable... }
```

其中，class\_reference 的格式是：

```
class_name [ * ]
```

有关的例子已经有了很多，因此在这里不再重复。

## 第10章

# 复杂查询

---

本章主要介绍更为深入的查询技巧，即定制查询功能和结果输出格式，主要包括：

- 更灵活的数据插入
- 复杂条件查询
- 规则表达式查询
- CASE 子句
- 控制查询结果
- 聚集查询



## 10.1 更灵活的数据插入

在 6.3 节中已经介绍过向数据库插入数据的简单方法，在那里的例子中，要求为表的所有字段都提供确定的值。然而在有些情况下，在插入新记录时无法为表的某些字段提供确定的值，或者某些字段具有默认值。这时，需要使用更高级的数据插入技巧。

### 10.1.1 空值

以 friend 表为例，假定在生成一条新的记录时，不知道 city 和 prov 字段的确切值。在这种情况下，可以利用空值(NULL)为 city 和 prov 指定初始值。

#### 1. 空值的插入

向记录中插入空值的方法类似于：

```
testdb=# INSERT INTO friend
testdb=# VALUES('张健',NULL,NULL,18);
INSERT 19777 1
testdb=# SELECT * FROM friend;
   name   |   city   |   prov   | age
-----+-----+-----+-----
张祥      | 武汉     | 湖北     | 19
李志建    | 大冶     | 湖北     | 28
李江      | 广州     | 广东     | 25
阳阳      | 武汉     | 湖北     | 5
张健      |          |          | 18
(5 rows)
```

从查询的结果可以看出，name 为“张健”的记录中，city 和 prov 字段没有任何信息。

更为简单的方法是，只在 INSERT 中给出有确定值的字段。对于 INSERT 没有给出的字段，PostgreSQL 自动为它们赋以空值。例如：

```
testdb=# INSERT INTO friend
testdb=# (name,age)
testdb=# VALUES('李鸿',22);
INSERT 19778 1
testdb=# SELECT * FROM friend;
   name   |   city   |   prov   | age
-----+-----+-----+-----
李鸿      |          |          | 22
```

```

-----+-----+-----+-----
张祥      | 武汉      | 湖北      | 19
李志建    | 大冶      | 湖北      | 28
李江      | 广州      | 广东      | 25
阳阳      | 武汉      | 湖北      | 5
张健      |           |           | 18
李鸿      |           |           | 22
(6 rows)

```

第二种方法的效果与第一种方法相同，但命令形式更为简洁。在空值字段较多时，第二种方法更有效。

## 2. 空值的特性

空值兼容于所有的数据类型，但既不同于空格字符串，也不同于数字0，仅表示“未知值”或“未指定值”。下面的一些例子有助于对空值的理解。

```

testdb=# INSERT INTO friend
testdb=#   (name) VALUES ('王芳');
INSERT 19792 1
testdb=# INSERT INTO friend
testdb=#   (name,age)
testdb=#   VALUES ('怪物',-10);
INSERT 19793 1
testdb=# SELECT * FROM friend
testdb=#   ORDER BY age DESC;
   name   |   city   |   prov   | age
-----+-----+-----+-----
李志建    | 大冶      | 湖北      | 28
李江      | 广州      | 广东      | 25
李鸿      |           |           | 22
张祥      | 武汉      | 湖北      | 19
张健      |           |           | 18
阳阳      | 武汉      | 湖北      | 5
怪物      |           |           | -10
王芳      |           |           |
(8 rows)

```

从查询结果可以看出，空值比任何数据都小。

```

testdb=# SELECT * FROM friend
testdb=#   WHERE age<>99
testdb=#   ORDER BY age;
   name   |   city   |   prov   | age
-----+-----+-----+-----

```



## 10 复杂查询

```
-----+-----+-----+-----
怪物      |      |      | -10
阳阳      | 武汉  | 湖北  | 5
张健      |      |      | 18
张祥      | 武汉  | 湖北  | 19
李鸿      |      |      | 22
李江      | 广州  | 广东  | 25
李志建    | 大冶  | 湖北  | 28
(7 rows)
```

从查询结果可以看出，空值与任何数据都不相等，因此“=”或“<>”运算符都不能应用于空值字段。判断一个字段是否为空值的唯一方法是使用“IS”和“IS NOT”运算符。例如：

```
testdb=# SELECT * FROM friend
testdb=# WHERE age IS NULL;
 name | city | prov | age
-----+-----+-----+-----
王芳  |     |     |
(1 row)

testdb=# SELECT * FROM friend
testdb=# WHERE age IS NOT NULL;
 name | city | prov | age
-----+-----+-----+-----
张祥  | 武汉  | 湖北  | 19
李志建 | 大冶  | 湖北  | 28
李江  | 广州  | 广东  | 25
阳阳  | 武汉  | 湖北  | 5
张健  |      |      | 18
李鸿  |      |      | 22
怪物  |      |      | -10
(7 rows)
```

基于同样的原因，在进行两个字段的比较时，只要其中的一个字段为空值，该记录就不会出现在查询结果中。例如：

```
testdb=# SELECT * FROM friend
testdb=# WHERE city=prov;
 name | city | prov | age
-----+-----+-----+-----
(0 rows)

testdb=# SELECT * FROM friend
testdb=# WHERE city<>prov;
```

name	city	prov	age
张祥	武汉	湖北	19
李志建	大冶	湖北	28
李江	广州	广东	25
阳阳	武汉	湖北	5

(4 rows)

事实上，在name的值为“张健”、“李鸿”和“怪物”的记录中，city和prov字段都为空值。

最后需要说明的是，空值也不等同于不含任何字符的空字符串。例如：

```
testdb=# INSERT INTO friend
testdb=# (name,city)
testdb=# VALUES('非空城市','');
INSERT 19796 1
testdb=# SELECT * FROM friend
testdb=# WHERE city IS NULL;
name | city | prov | age
-----+-----+-----+-----
张健  |      |      | 18
李鸿  |      |      | 22
王芳  |      |      |
怪物  |      |      | -10
(4 rows)
testdb=# SELECT * FROM friend
testdb=# WHERE city='';
name | city | prov | age
-----+-----+-----+-----
非空城市 |      |      |
(1 row)
```

在name为“非空城市”的记录中，虽然city是为空的字符串，但并非为空值。

### 10.1.2 默认值

有些数据字段在大多数情况下的值是固定的。如描述天气情况的字段，在大多数情况下，天气情况是“晴天”。在这种情况下，可以为天气字段指定一个默认值（“晴天”）。为某个字段指定了默认值后，如果在插入记录时没有为该字段显式地指定值，PostgreSQL就自动地为该字段赋予默认值。

在目前的版本中，只有在定义表结构时才能为字段指定默认值。例如：

```
testdb=# CREATE TABLE weather (
```

## 10 复杂查询

```
testdb=# dt date,
testdb=# wt char(10) DEFAULT '晴天');
```

```
CREATE
```

```
testdb=# \d weather
```

```
Table "weather"
```

```
Attribute | Type | Modifier
```

```
-----+-----+-----
```

```
dt      | date |
```

```
wt      | char(10) | default '晴天'
```

这样，weather 表中的 wt 字段的默认值为“晴天”。在插入记录时，如果不为 wt 指定值，wt 的值将是“晴天”。例如：

```
testdb=# INSERT INTO weather
```

```
testdb=# (dt)
```

```
testdb=# VALUES('2001-1-31'::date);
```

```
INSERT 19790 1
```

```
testdb=# INSERT INTO weather
```

```
testdb=# VALUES('2001-2-1'::date,'阴天');
```

```
INSERT 19791 1
```

```
testdb=# SELECT * FROM weather;
```

```
dt      | wt
```

```
-----+-----
```

```
2001-01-31 | 晴天
```

```
2001-02-01 | 阴天
```

```
(2 rows)
```

### 10.1.3 利用其他表插入数据

除了在 INSERT 命令中为字段直接指定值外，还可以利用其他表向一个表插入数据。为了演示这种方法，先建立一个临时表。

```
testdb=# CREATE TABLE tmp (
```

```
testdb=# name char(8),
```

```
testdb=# sex char(2) DEFAULT '男',
```

```
testdb=# age integer);
```

```
CREATE
```

然后利用 INSERT...SELECT 命令从 friend 表中取出数据，并插入到 tmp 表中。例如：

```
testdb=# INSERT INTO tmp
```

```
testdb=# (name,age)
```

```
testdb=# SELECT name,age
```

```
testdb=# FROM friend;
```

```
INSERT 0 9
testdb=# SELECT * FROM tmp;
   name   | sex | age 
-----+-----+-----
 张祥     | 男  | 19  
 李志建   | 男  | 28  
 李江     | 男  | 25  
 阳阳     | 男  | 5   
 张健     | 男  | 18  
 李鸿     | 男  | 22  
 王芳     | 男  |    
 怪物     | 男  | -10  
 非空城市 | 男  |    
(9 rows)
```

## 10.2 复杂条件查询

前面的例子所使用的查询条件表达式大部分是简单的逻辑表达式，可以利用逻辑运算符 AND、OR 或 NOT 将简单表达式组合为复杂的查询表达式，以完成更复杂的查询。

下面的例子查询 city 的值为“武汉”或“广州”的记录。

```
testdb=# SELECT * FROM friend
testdb=# WHERE city='武汉' OR city='广州';
   name   | city   | prov | age 
-----+-----+-----+-----
 张祥     | 武汉   | 湖北 | 19  
 李江     | 广州   | 广东 | 25  
 阳阳     | 武汉   | 湖北 | 5   
(3 rows)
```

下面的例子查询 age 小于 20，并且 prov 的值为“湖北”或“广东”的记录。

```
testdb=# SELECT * FROM friend
testdb=# WHERE age<20 AND (prov='湖北' OR prov='广东');
   name   | city   | prov | age 
-----+-----+-----+-----
 张祥     | 武汉   | 湖北 | 19  
 阳阳     | 武汉   | 湖北 | 5   
(2 rows)
```

## 10 复杂查询

下面的例子查询 age 在 10 至 20 之间的记录。

```
testdb=# SELECT * FROM friend
testdb=# WHERE age BETWEEN 10 AND 20;

 name | city | prov | age 
-----+-----+-----+-----
 张祥  | 武汉 | 湖北 | 19  
 张健  |      |      | 18  
(2 rows)
```

上面的查询条件使用了“age BETWEEN 10 AND 20”形式的表达式，这个表达式等同于“age>=10 AND age<=20”。“BETWEEN...AND”是 SQL 的标准运算符，用于判断一个数值是否在某一个范围之内。

下面的例子列出了 name 的第一个汉字为“张”的记录，这种查询方法俗称为“模糊查询”。

```
testdb=# SELECT * FROM friend
testdb=# WHERE name LIKE '张%';

 name | city | prov | age 
-----+-----+-----+-----
 张祥  | 武汉 | 湖北 | 19  
 张健  |      |      | 18  
(2 rows)
```

“LIKE”是实现模糊查询的关键，“LIKE”之后是模糊查询的模板，模板中的“%”匹配任意的字符串，而“\_”匹配单个的字符。表 10-1 列出了 LIKE 模板的几个例子。

表 10-1 LIKE 模板的例子

例子	说明
col LIKE 'D%'	如果字段 col 的第一个字符为 D，运算结果为真，否则结果为假
col LIKE '_D%'	如果字段 col 的第二个字符为 D，运算结果为真，否则结果为假
col LIKE 'D%e%'	如果字段 col 的第一个字符为 D 且含有字母 e，运算结果为真，否则结果为假
col LIKE 'D%e%f%'	如果字段 col 的第一个字符为 D 且依次含有字母 e 和 f，运算结果为真，否则结果为假
col NOT LIKE '%明'	如果字段 col 的最后一个字符串不为“明”，运算结果为真，否则结果为假
col NOT LIKE '%公司%'	如果字段 col 中不含字符串“公司”，运算结果为真，否则结果为假

这种查询方法常用于对字符型字段的模糊查询。

## 10.3 规则表达式查询

规则表达式是 UNIX 平台上常用的一种字符串比较方法，将其引入查询表达式是 PostgreSQL 的一个创举，为查询表达式中的字符串比较提供了更为强大的工具。

### 10.3.1 规则表达式简介

规则表达式是使用特殊语法进行字符串匹配的字符串，它允许用户利用少数几个特殊字符构造无穷数目的字符串比较表达式。用于匹配规则表达式的运算符有“~”、“~\*”、“!”、“~”和“!~\*”，有关的介绍请参考表 9-2。用于规则表达式的特殊字符见表 10-2。

表10-2 规则表达式特殊字符

特殊字符	意义
^	以指定的字符串开头
\$	以指定的字符串结束
.	匹配任意的单个字符
[ccc]	匹配字符集合[ccc]中的任意字符
[^ccc]	不匹配字符集合[ccc]中的任意字符
[c-c]	匹配范围[c-c]之间的任意字符
[^c-c]	不匹配范围[c-c]之间的任意字符
?	参照前一个字符，匹配零个或一个这样的字符
*	参照前一个字符，匹配零个或多个这样的字符
+	参照前一个字符，匹配一个或多个这样的字符
	逻辑或(OR)运算符

如果要取消上述字符的特殊意义，则必须用“\”来转意。表 10-3 列出了一些规则表达式的例子。

表10-3 规则表达式的例子

规则表达式	意义
col ~ 'D'	如果字段 col 以字符 D 开头，结果为真，否则结果为假
col ~ 'D'	如果字段 col 包含字符 D，结果为真，否则结果为假
col ~ '^D'	如果字段 col 的第二个字符为 D，结果为真，否则结果为假
col ~ 'D.*e'	如果字段 col 的第一个字符为 D，且包含字符 e，结果为真，否则结果为假
col ~ 'D.*e.*f'	如果字段 col 的第一个字符为 D，且依次包含字符 e 和 f，结果为真，否则结果为假
col ~ '[A-D]' 或 col ~ '[ABCD]'	如果字段 col 包含字母 A、B、C、D，结果为真，否则结果为假
col ~ '*a' 或 col ~ '[Aa]'	如果字段 col 包含字母 A 或 a，结果为真，否则结果为假
col !~ 'D'	如果字段 col 不包含 D，结果为真，否则结果为假
col !~ '^D' 或 col ~ '^[^D]'	如果字段 col 不以 D 开头，结果为真，否则结果为假
col ~ '^ ?D'	如果字段 col 以 D 开头或以一个空格加 D 开头，结果为真，否则结果为假。注意：?号之前有一个空格
col ~ '^ *D'	如果字段 col 以 D 开头或以若干个空格加 D 开头，结果为真，否则结果为假。注意：*号之前有一个空格
col ~ '^ +D'	如果字段 col 以至少一个空格加 D 开头，结果为真，否则结果为假。注意：+号之前有一个空格
col ~ 'G *\$'	如果字段 col 以 G 加上若干个空格结尾，结果为真，否则结果为假。注意：*号之前有一个空格

### 10.3.2 在查询中的应用

规则表达式提供了比 LIKE 更为灵活的字符串比较手段。

下面的例子查询 name 字段以“李”开头的记录。

```
testdb=# SELECT * FROM friend
testdb=# WHERE name ~ '^李';
name | city | prov | age
-----+-----+-----+-----
```

## 10 复杂查询

```
李志建 | 大冶 | 湖北 | 28
李江 | 广州 | 广东 | 25
李鸿 | | | 22
```

(3 rows)

下面的例子查询 name 字段中第二个汉字为“志”的记录。

```
testdb=# SELECT * FROM friend
```

```
testdb=# WHERE name ~ '^..志';
```

```
name | city | prov | age
```

```
-----+-----+-----+-----
李志建 | 大冶 | 湖北 | 28
```

(1 row)

下面的例子查询 name 字段最后一个汉字为“健”的记录。

```
testdb=# INSERT INTO friend
```

```
testdb=# VALUES('刘健康',NULL,NULL,20);
```

```
INSERT 19872 1
```

```
testdb=# SELECT * FROM friend
```

```
testdb=# WHERE name ~ '健 *$';
```

```
name | city | prov | age
```

```
-----+-----+-----+-----
张健 | | | 18
```

(1 row)

下面的例子查询 name 字段中含有“健”的记录。

```
testdb=# SELECT * FROM friend
```

```
testdb=# WHERE name ~ '健';
```

```
name | city | prov | age
```

```
-----+-----+-----+-----
张健 | | | 18
```

```
刘健康 | | | 20
```

(2 rows)

## 10.4 CASE 子句

几乎所有的程序设计语言都有条件分支语句，如C语言中的“if ()...else...”。这种语句能够根据某个逻辑表达式的值来决定执行语句的路径。虽然SQL不是一种过程化的程序设计语言，但它仍然允许根据条件来控制查询结果的值。在SELECT语句的WHERE子句中，可以利用CASE来进行条件控制。其基本语法格式是：

```

CASE
    WHEN 条件1 THEN 值1
    WHEN 条件2 THEN 值2
    .....
    WHEN 条件n THEN 值n
    ELSE 值n+1
END

```

如果“条件1”成立，则结果为“值1”；如果“条件2”成立，则结果为“值2”；.....；如果“条件n”成立，则结果为“值n”；如果所有的条件都不成立，则结果为“值n+1”。

例如：

```

testdb=# SELECT name,age,
testdb=# CASE
testdb=# WHEN age>=18 THEN '成年人'
testdb=# ELSE '非成年人'
testdb=# END

```

```

testdb=# FROM friend;

```

name	age	case
张祥	19	成年人
李志建	28	成年人
李江	25	成年人
阳阳	5	非成年人
张健	18	成年人
李鸿	22	成年人
王芳		非成年人
怪物	-10	非成年人
非空城市		非成年人
刘健康	20	成年人

(10 rows)

更为复杂的例子是：

```

testdb=# SELECT name,prov,
testdb=# CASE
testdb=# WHEN prov IS NULL THEN '未知省份'
testdb=# WHEN prov='湖北' THEN '本省'
testdb=# ELSE '外省'
testdb=# END AS 本省否
testdb=# FROM friend;

```

name	prov	本省否
------	------	-----



## 10 复杂查询

张祥	湖北	本省
李志建	湖北	本省
李江	广东	外省
阳阳	湖北	本省
张健		未知省份
李鸿		未知省份
王芳		未知省份
怪物		未知省份
非空城市		未知省份
刘健康		未知省份

(10 rows)

## 10.5 控制查询结果

### 10.5.1 删除重复行

先来看一个例子：

```
testdb=# SELECT city FROM friend;
```

city
------

-----

武汉
大冶
广州
武汉

(10 rows)

在查询结果中，有多行是重复的。许多情况下，需要将重复的行去掉，DISTINCT关键字可以实现这一点。例如：

```
testdb=# SELECT DISTINCT(city)
testdb=# FROM friend;
```

```
city
-----
```

```
广州
武汉
大冶
```

(5 rows)

上面的结果中，第一行为空值，第五行（最后一行）为不包含任何字符的空字符串。这说明，使用了DISTINCT之后，结果的数据行都互不相同。

## 10.5.2 限制行数

默认情况下，SELECT输出所有的结果行。可以利用LIMIT子句，限定结果的输出行数。基本语法格式是：

```
LIMIT n1 OFFSET n2
```

PostgreSQL将查询的结果按行进行编号，第一行的编号为1，第二行的编号为2，其余行的编号依次类推。LIMIT子句的意义是：从结果的第n2+1行开始，输出n1行，其他行均不输出。如果不指定OFFSET，则从结果的第一行开始输出n1行。例如：

--不限定输出行数

```
testdb=# SELECT * FROM friend
```

```
testdb=# ORDER BY age;
```

name	city	prov	age
怪物			-10
阳阳	武汉	湖北	5
张健			18
张祥	武汉	湖北	19
刘健康			20
李鸿			22
李江	广州	广东	25
李志建	大冶	湖北	28
王芳			
非空城市			

(10 rows)

--输出结果中的头3行

```
testdb=# SELECT * FROM friend
```

```
testdb=# ORDER BY age
```

```
testdb=# LIMIT 3;
```

## 10 复杂查询

```
name | city | prov | age
-----+-----+-----+-----
怪物 |      |      | -10
阳阳 | 武汉 | 湖北 | 5
张健 |      |      | 18
(3 rows)
```

--从第六行开始，输出2行

```
testdb=# SELECT * FROM friend
```

```
testdb=# ORDER BY age
```

```
testdb=# LIMIT 2 OFFSET 5;
```

```
name | city | prov | age
-----+-----+-----+-----
李鸿 |      |      | 22
李江 | 广州 | 广东 | 25
(2 rows)
```

值得注意的是，上面的每一个查询都使用了ORDER BY子句。如果没有这个子句，LIMIT可能以随机的顺序从查询中读取数据行，因此结果可能并不是用户所需要的。

由于减少了向客户端传送数据的数量，因此，LIMIT的使用能够改善服务器的性能，并减轻网络的负载。如果表存在索引且匹配于ORDER BY，某些情况下LIMIT可能不需要执行整个查询就能得到结果。

### 10.5.3 游标

通常情况下，查询所产生的所有数据行都会传送到客户端，而不管客户端是否需要所有的结果行。游标则允许客户端“按需”地选取数据行。

#### 1. 游标的定义

```
DECLARE cursorname [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] }
```

本命令创建一个游标，有关的详细介绍请参考附录。

可以利用 FETCH 语句从游标中读出数据，FETCH 的语法是：

```
FETCH [ selector ] [ count ] { IN | FROM } cursor
```

```
FETCH [ RELATIVE ] [ { [ # | ALL | NEXT | PRIOR ] } ] FROM ] cursor
```

一个游标使用完毕后，应该及时关闭，以便释放游标所占用的资源。释放游标的命令是：

```
CLOSE cursor
```

游标只能在事务中使用，可以使用 BEGIN、COMMIT 或 ROLLBACK 定义一个事务。

## 2. 例子

--开始一个事务

```
testdb=# BEGIN WORK;
```

```
BEGIN
```

--定义一个游标

```
testdb=# DECLARE friend_cursor CURSOR FOR
```

```
testdb=# SELECT * FROM friend;
```

```
SELECT
```

--从游标中取出一行数据

```
testdb=# FETCH 1 FROM friend_cursor;
```

name	city	prov	age
张祥	武汉	湖北	19

(1 row)

--从游标中取出另一行数据

```
testdb=# FETCH 1 FROM friend_cursor;
```

name	city	prov	age
李志建	大冶	湖北	28

(1 row)

--从游标中取出两行数据

```
testdb=# FETCH 2 FROM friend_cursor;
```

name	city	prov	age
李江	广州	广东	25
阳阳	武汉	湖北	5

(2 rows)

--取出当前游标的前一行数据

```
testdb=# FETCH -1 FROM friend_cursor;
```

name	city	prov	age
李江	广州	广东	25

(1 row)

## 10 复杂查询

```
--移动游标的位置
testdb=# MOVE 2 FROM friend_cursor;

MOVE

--取当前游标的前一行数据
testdb=# FETCH -1 FROM friend_cursor;
   name  |   city   | prov | age
-----+-----+-----+-----
  阳阳   |   武汉   | 湖北 |   5
(1 row)

--关闭游标
testdb=# CLOSE friend_cursor;

CLOSE

--提交事务
testdb=# COMMIT WORK;

COMMIT
```

## 10.6 聚集查询

许多情况下，用户希望得到表的某些统计信息，如合计值、平均值、最大值、最小值、满足条件的记录数等。这类查询统称为“聚集”查询。

### 10.6.1 聚集函数

#### 1. 聚集函数类别

SQL使用聚集函数来得到聚集查询结果。常用的聚集函数有：

COUNT(\*)——统计结果行数。

SUM(colname)——求指定数字字段的合计。

MAX(colname)——求指定数字字段的最大值。

MIN(colname)——求指定数字字段的最小值。

AVG(colname)——求指定数字字段的平均值。

#### 2. 例子

利用聚集函数与WHERE子句的组合，可以得到各种复杂的统计结果。例如：

```
testdb=# SELECT avg(age)
```

```
testdb=# FROM friend
testdb=# WHERE age>=21;
avg
-----
25
(1 row)
```

上面的查询求出了age在21以上的所有记录中age字段的平均值。其他的例子有：

--查询表中的记录总数

```
testdb=# SELECT count(*)
testdb=# FROM friend;
count
-----
10
(1 row)
```

--查询age字段的总和

```
testdb=# SELECT sum(age)
testdb=# FROM friend;
sum
-----
127
(1 row)
```

--求age字段的最大值

```
testdb=# SELECT max(age)
testdb=# FROM friend;
max
-----
28
(1 row)
```

--求age字段的最小值

```
testdb=# SELECT min(age)
testdb=# FROM friend;
min
-----
-10
(1 row)
```

### 3. 空值与聚集查询

## 10 复杂查询

如果聚集函数所作用的字段包含空值，则应该注意查询结果的变化。例如，

--求age非空的记录个数

```
testdb=# SELECT count(age)
```

```
testdb=# FROM friend;
```

```
count
```

```
-----
```

```
8
```

```
(1 row)
```

在上面的查询中，结果为8，而不是想象中的10。其原因是，有两条记录的age字段为空值，这两条记录不计入计算结果。

为了进一步地说明其他聚集函数对空值的处理方法，下面创建一个新的表，并给出几个典型的聚集查询例子。

```
testdb=# CREATE TABLE tmp (num integer);
```

```
CREATE
```

```
testdb=# INSERT INTO tmp
```

```
testdb=# VALUES(NULL);
```

```
INSERT 19913 1
```

```
testdb=# SELECT * FROM tmp;
```

```
num
```

```
-----
```

```
(1 row)
```

--对空值字段求和，结果仍为空值

```
testdb=# SELECT sum(num)
```

```
testdb=# FROM tmp;
```

```
sum
```

```
-----
```

```
(1 row)
```

--对空值字段求最大值，结果仍为空值

```
testdb=# SELECT max(num)
```

```
testdb=# FROM tmp;
```

```
max
```

```
-----
```

```
(1 row)
```

--空值字段不影响count(\*)函数的结果

```
testdb=# SELECT count(*)
```

```
testdb=# FROM tmp;
```

```
count
```

```
-----
```

```
1
```

```
(1 row)
```

```
testdb=# INSERT INTO tmp
```

```
testdb=# VALUES(5);
```

```
INSERT 19914 1
```

```
testdb=# SELECT * FROM tmp;
```

```
num
```

```
-----
```

```
5
```

```
(2 rows)
```

--空值字段不参与平均值的计算

```
testdb=# SELECT avg(num)
```

```
testdb=# FROM tmp;
```

```
avg
```

```
-----
```

```
5
```

```
(1 row)
```

--空值字段不计入count(字段名)形式的查询结果

```
testdb=# SELECT count(num)
```

```
testdb=# FROM tmp;
```

```
count
```

```
-----
```

```
1
```

```
(1 row)
```

空值字段对聚集查询结果的种种影响，容易被忽略，因此在实际应用中必须时刻注意。

## 10.6.2 GROUP BY

简单的聚集查询总是返回一个结果。然而，许多情况下，需要根据某个表达式的值，分组计算聚集结果，并返回多个结果。这时就需要使用GROUP BY子句，它的基本形式是：



## 10 复杂查询

```
GROUP BY column [, ...]
```

这个子句将所有在字段列表上有相同值的行压缩为一行。对于聚集查询，这些聚集函数将计算每个组的所有行，并且为每个组计算一个独立的值。

例如：

```
testdb=# SELECT DISTINCT(prov)
testdb=# FROM friend;
prov
```

```
-----
```

广东

湖北

```
(3 rows)
```

```
testdb=# SELECT sum(age)
testdb=# FROM friend
testdb=# GROUP BY prov;
```

```
sum
```

```
-----
```

25

52

50

```
(3 rows)
```

上述查询结果的意义是，prov为“广东”的所有记录的age字段的和为25，prov为“湖北”的所有记录的age字段的和为52，prov为空值的所有记录的age的字段和为50。从结果可以看出，聚集查询对表中的数据行按prov字段的值进行了分组，并分别计算出聚集结果。更为复杂的例子是：

```
testdb=# SELECT prov,max(age),min(age),avg(age)
testdb=# FROM friend
testdb=# GROUP BY prov
testdb=# ORDER BY 4 DESC;
```

```
prov | max | min | avg
```

```
-----+-----+-----+-----
```

广东 | 25 | 25 | 25

湖北 | 28 | 5 | 17

| 22 | -10 | 12

```
(3 rows)
```

这个例子同时求出了age的最大值、最小值和平均值，并对结果按平均值进行了排序。请注意ORDER BY子句的用法。由于排序的依据是一个聚集函数，因此无法以通常的方法给出排序字段，而是给出排序对象在SELECT目标列表中的序号。

如果SELECT目标列表中出现了一个聚集表达式，那么其他目标要么是另外的聚集表达

式，要么是GROUP BY子句中的字段，否则会导致语法错误。例如，下面查询的目标列表中出现了name字段，因此PostgreSQL无法执行。

```
testdb=# SELECT name,max(age)
testdb=# FROM friend
testdb=# GROUP BY prov;
ERROR: Attribute friend.name must be GROUPed or used in an aggregate function
```

### 10.6.3 HAVING

如果要在查询中对聚集结果进行条件判断和控制，可以使用HAVING子句。它的语法格式是：

```
HAVING cond_expr
```

其中，cond\_expr 是一个逻辑表达式，与 WHERE 子句中的条件表达式一样，遵循同样的语法规则。HAVING 子句迫使命令从查询结果集合中丢弃不符合 cond\_expr 条件的数据行。HAVING 与 WHERE 不同：WHERE 在应用 GROUP BY 之前过滤出单独的行，而 HAVING 过滤由 GROUP BY 创建的行；HAVING 的 cond\_expr 中可以使用聚集函数，而在 WHERE 的 cond\_expr 中不允许使用聚集函数。除非在聚集查询中，否则在 cond\_expr 中引用的每个字段应该明确地指定为某个组的字段。例如：

--列出所有的聚集查询结果

```
testdb=# SELECT prov,count(*),sum(age)
testdb=# FROM friend
testdb=# GROUP BY prov;
   prov   | count | sum
-----+-----+-----
  广东   |      1 |  25
  湖北   |      3 |  52
         |      6 |  50
(3 rows)
```

--仅列出记录数超过2的分组的聚集查询结果

```
testdb=# SELECT prov,count(*),sum(age)
testdb=# FROM friend
testdb=# GROUP BY prov
testdb=# HAVING count(*)>2;
   prov   | count | sum
-----+-----+-----
  湖北   |      3 |  52
         |      6 |  50
(2 rows)
```

## 10 复杂查询

---

--列出age超过20的分组的聚集查询结果

```
testdb=# SELECT prov,count(*),sum(age)
```

```
testdb=# FROM friend
```

```
testdb=# GROUP BY prov
```

```
testdb=# HAVING avg(age)>20;
```

prov	count	sum
广东	1	25

(1 row)

HAVING通常与GROUP BY一起使用，以控制分组查询结果的输出。

## 第11章

# 连接查询

---

连接是将两个或多个表拼接成一个更大表的操作，生成的表中包含满足连接条件的所有记录，是最基本的查询操作之一。本章通过一些难度较大的例子，深入地介绍连接查询的方法，主要包括：

- 表标识和字段引用

- 表的连接

- 复杂连接查询



## 11.1 表标识和字段引用

在介绍连接查询之前，先介绍SQL语言的一个重要特性——表标识和字段引用。

### 1. 表标识

表标识实际上就是表的别名。为一个表指定表标识的基本形式是：

FROM 表名称 表标识

其中，表标识是一个合法的PostgreSQL标识符。一旦为一个表指定了一个标识，那么在同一条查询语句中，就可以用这个标识来实现对表的引用。例如：

```
testdb=# SELECT * FROM friend frd
testdb=# WHERE name IS NOT NULL;
 name | city | prov | age
-----+-----+-----+-----
 张祥  | 武汉 | 湖北 | 19
 李志建 | 大冶 | 湖北 | 28
 李江  | 广州 | 广东 | 25
 阳阳  | 武汉 | 湖北 | 5
 张健  |      |      | 18
 李鸿  |      |      | 22
 王芳  |      |      |
 XX    |      |      | -10
 NNC   |      |      |
 刘健康 |      |      | 20
(10 rows)
```

### 2. 字段引用

迄今为止，在所有的例子中，基本上只涉及单个表，并且只是通过表和字段的名字来引用数据库对象。在复杂的查询中，为了避免混淆，需要为表字段指定别名，这种别名就是字段引用。字段引用的基本形式是：

表标识.字段名

上述引用表明，由“字段名”指定的字段属于由“表标识”所指定的表。

当一条查询语句涉及到多个表且不同的表含有相同名字的字段时，字段引用就可以派上用场了，用来说明某个字段属于哪一个表。

例如：

```

testdb=# \d person
          Table "person"
Attribute |  Type  | Modifier
-----+-----+-----
name      | char(10) |
photo     | oid      |
testdb=# \d friend
          Table "friend"
Attribute |  Type  | Modifier
-----+-----+-----
name      | char(8)  |
city      | char(10) |
prov      | char(6)  |
age       | integer  |
testdb=# SELECT p.name,f.city,f.age
testdb=# FROM person p,friend f
testdb=# WHERE p.name=f.name;
   name   |  city  | age
-----+-----+-----
  阳阳    |  武汉  |   5
(1 row)

```

上面最后一条查询语句涉及到两个表——person和friend，这两个表的表标识分别为“p”和“f”，且都有一个名为“name”的字段。“p.name”表明它是表person的字段“name”，而“f.name”表明它是表friend的字段“name”。

如果某个字段的名称在所有涉及到的表中唯一，则可以不使用字段引用而直接使用字段名。例如，上面例子中的“f.city”，由于city只出现在表friend中，因此可以直接使用“city”。

## 11.2 表的连接

连接是将两个或多个表的字段拼接成一个更大表的操作，生成的表中包含满足连接条件的所有记录。连接过程是通过连接条件来控制的，连接条件中将出现不同表中的公共字段，这个公共字段在一个表中是关键字，而在另外一个表中是外关键字。表连接操作的基本语法格式是：

```

SELECT 表1.字段1[, ....., 表n.字段n]
FROM 表1[, ....., 表n]
[WHERE 表i.字段x=表j.字段y]

```

## 11 连接查询

为了便于说明，下面给出一个较为复杂的数据库的例子。

假设为某学校设计一个学生成绩管理系统。原始的学生成绩登记表如表11-1所示：

表11-1 学生成绩登记表

学号	姓名	性别	年龄	系别	课程名称	成绩	任课老师
9803101	李小波	男	20	计算机	操作系统	85	李玲
9803102	王前	男	20	计算机	操作系统	79	李玲
9803103	仲林	女	21	计算机	操作系统	83	李玲
9803104	黄启	女	21	计算机	操作系统	88	李玲
9803105	元亮	男	23	计算机	操作系统	66	李玲
9803101	李小波	男	20	计算机	数据库	83	钟福利
9803102	王前	男	20	计算机	数据库	91	钟福利
9803103	仲林	女	21	计算机	数据库	85	钟福利
9803114	刘淇	女	21	计算机	数据库	84	钟福利
9803115	徐营串	男	23	计算机	数据库	77	钟福利
9810101	期荣强	男	19	机械	机械制图	83	黄为
9810102	钟恢复	男	21	机械	机械制图	91	黄为
9810103	王吉林	女	22	机械	机械制图	85	黄为
9810104	苏强	女	21	机械	机械制图	84	黄为
9810105	李源	男	22	机械	机械制图	77	黄为

按照关系数据库设计理论，这样的表存在着各种异常问题，因此需要对其进行分解。分解后得到的表有以下三个。

表11-2 学生基本情况表(s)

学号	姓名	性别	年龄	系别
9803101	李小波	男	20	计算机
9803102	王前	男	20	计算机
9803103	仲林	女	21	计算机
9803104	黄启	女	21	计算机
9803105	元亮	男	23	计算机
9803114	刘淇	女	21	计算机
9803115	徐营串	男	23	计算机
9810101	期荣强	男	19	机械
9810102	钟恢复	男	21	机械
9810103	王吉林	女	22	机械
9810104	苏强	女	21	机械
9810105	李源	男	22	机械

表11-3 课程基本情况表(c)

课程号	课程名称	任课老师
03014	操作系统	李玲
03011	数据库	钟福利
10021	机械制图	黄为

表11-4 成绩表(sc)

学号	课程号	成绩
9803101	03014	85
9803102	03014	79
9803103	03014	83
9803104	03014	88
9803105	03014	66
9803101	03011	83
9803102	03011	91
9803103	03011	85
9803114	03011	84
9803115	03011	77

续表

学号	课程号	成绩
9810101	10021	83
9810102	10021	91
9810103	10021	85
9810104	10021	84
9810105	10021	77

创建上述表及插入数据的命令是：

```
CREATE TABLE s (          --学生基本情况表
    sno char(7),           --学号，关键字
    sname char(8),         --姓名
    sex char(2),           --性别
    age integer,           --年龄
    dept char(10)          --系别
);

CREATE TABLE c (          --课程基本情况表
    cno char(5),           --课程号，关键字
    cname char(20),        --课程名称
    teacher char(8)        --任课老师
);

CREATE TABLE sc (         --成绩表
    sno char(7),           --学号，用于与表s进行连接的外关键字
    cno char(5),           --课程号，用于与表c进行连接的外关键字
    grade integer          --成绩
);

INSERT INTO s VALUES(
    '9803101','李小波','男',20,'计算机');
INSERT INTO s VALUES(
    '9803102','王前','男',20,'计算机');
INSERT INTO s VALUES(
    '9803103','仲林','女',21,'计算机');
INSERT INTO s VALUES(
    '9803104','黄启','女',21,'计算机');
INSERT INTO s VALUES(
    '9803105','元亮','男',23,'计算机');
INSERT INTO s VALUES(
    '9803114','刘淇','女',21,'计算机');
INSERT INTO s VALUES(
    '9803115','徐营串','男',23,'计算机');
INSERT INTO s VALUES(
```



```
'9810101','期荣强','男',19,'机械');
INSERT INTO s VALUES(
    '9810102','钟恢复','男',21,'机械');
INSERT INTO s VALUES(
    '9810103','王吉林','女',22,'机械');
INSERT INTO s VALUES(
    '9810104','苏强','女',21,'机械');
INSERT INTO s VALUES(
    '9810105','李源','男',22,'机械');
INSERT INTO c VALUES(
    '03014','操作系统','李玲');
INSERT INTO c VALUES(
    '03011','数据库','钟福利');
INSERT INTO c VALUES(
    '10021','机械制图','黄为');
INSERT INTO sc VALUES(
    '9803101','03014',85);
INSERT INTO sc VALUES(
    '9803102','03014',79);
INSERT INTO sc VALUES(
    '9803103','03014',83);
INSERT INTO sc VALUES(
    '9803104','03014',88);
INSERT INTO sc VALUES(
    '9803105','03014',66);
INSERT INTO sc VALUES(
    '9803101','03011',83);
INSERT INTO sc VALUES(
    '9803102','03011',91);
INSERT INTO sc VALUES(
    '9803103','03011',85);
INSERT INTO sc VALUES(
    '9803114','03011',84);
INSERT INTO sc VALUES(
    '9803115','03011',77);
INSERT INTO sc VALUES(
    '9810101','10021',83);
INSERT INTO sc VALUES(
    '9810102','10021',91);
```

```
INSERT INTO sc VALUES(
    '9810103','10021',85);
INSERT INTO sc VALUES(
    '9810104','10021',84);
INSERT INTO sc VALUES(
    '9810105','10021',77);
```

利用连接操作，下面的命令可将三个表连接成与表11-1相同的表。

```
SELECT s.sno AS 学号,
       s.sname AS 姓名,
       s.sex AS 性别,
       s.age AS 年龄,
       s.dept AS 系别,
       c.cname AS 课程名称,
       sc.grade AS 成绩,' ',
       c.teacher AS 任课老师
FROM s,c,sc
WHERE s.sno=sc.sno AND sc.cno=c.cno
ORDER BY c.cname;
```

学号	姓名	性别	年龄	系别	课程名称	成绩	任课老师
9803101	李小波	男	20	计算机	操作系统	85	李玲
9803102	王前	男	20	计算机	操作系统	79	李玲
9803103	仲林	女	21	计算机	操作系统	83	李玲
9803104	黄启	女	21	计算机	操作系统	88	李玲
9803105	亓亮	男	23	计算机	操作系统	66	李玲
9803101	李小波	男	20	计算机	数据库	83	钟福利
9803102	王前	男	20	计算机	数据库	91	钟福利
9803103	仲林	女	21	计算机	数据库	85	钟福利
9803114	刘淇	女	21	计算机	数据库	84	钟福利
9803115	徐营串	男	23	计算机	数据库	77	钟福利
9810101	期荣强	男	19	机械	机械制图	83	黄为
9810102	钟恢复	男	21	机械	机械制图	91	黄为
9810103	王吉林	女	22	机械	机械制图	85	黄为
9810104	苏强	女	21	机械	机械制图	84	黄为
9810105	李源	男	22	机械	机械制图	77	黄为

(15 rows)

## 11.3 复杂连接查询

本节通过几个例子，介绍一些更复杂的连接查询方法。

### 1. 查询“李小波”所修的全部课程名称

这一查询涉及到表 *s* 中的姓名和 *c* 表中的课程名。由于 *s* 表与 *c* 表没有直接的对应关系，因此需要通过 *sc* 来间接地进行连接。

```
testdb=# SELECT s.sname AS 姓名,
testdb=#      c.cname AS 课程名称
testdb=# FROM s,sc,c
testdb=#      WHERE s.sno = sc.sno AND sc.cno = c.cno AND
testdb=#      s.sname = '李小波';
 姓名  | 课程名称
-----+-----
李小波 | 数据库
李小波 | 操作系统
(2 rows)
```

在本例的 WHERE 子句中，既有连接表达式，又有结果选择表达式。

### 2. 查询所有成绩在 80 分以上的学生姓名和所在的系

这一查询涉及到表 *s* 中的姓名、系别以及 *sc* 表中的成绩，这两个表通过 *cno* 字段进行连接。

```
testdb=# SELECT DISTINCT s.sname AS 姓名,
testdb=#      s.dept AS 所在的系
testdb=# FROM s,sc
testdb=#      WHERE s.sno = sc.sno AND
testdb=#      (s.sno NOT IN (
testdb=#      SELECT sc.sno FROM
testdb=#      sc WHERE grade<80)
testdb=#      );
 姓名  | 所在的系
-----+-----
黄启   | 计算机
期荣强 | 机械
```

李小波	计算机
刘淇	计算机
苏强	机械
王吉林	机械
钟恢复	机械
仲林	计算机

(8 rows)

请注意分数限制条件的写法。其中 IN 运算符的功能是判断一个数据库对象的值是否在一个集合中。可以这样来理解上述查询条件,先求出只要有一门课程成绩在 80 分以下的学生集合(第一个集合),然后用连接操作生成所有学生的集合(第二个集合),最后从第二个集合中去掉包含于第一个集合的元素,得到最终的结果。如果将成绩限制条件写成“sc.grade>=80”则不能得到正确的结果。“sc.grade>=80”表明,只要有一门成绩在 80 分以上,该学生就在结果之列。

### 3. 查询没有选修“操作系统”课程的学生姓名

这一查询涉及到表 s 中的姓名和 c 表中的课程名。由于 s 表与 c 表没有直接的对应关系,因此需要通过 sc 来间接地进行连接。

```
testdb=# SELECT DISTINCT s.sname AS 姓名
testdb=# FROM s,sc,c
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno AND
testdb=# (s.sno NOT IN (
testdb=# SELECT scl.sno FROM sc scl,c cl
testdb=# WHERE scl.cno = cl.cno and cl.cname='操作系统')
testdb=# );
```

姓名

-----

期荣强

李源

刘淇

徐营串

苏强

王吉林

钟恢复

(7 rows)

与前一个例子一样,可以从集合的角度来理解这个例子的课程限制条件。唯一不同之处在于,这个例子的第一个集合是通过连接操作来实现的。这个例子的查询语句有另外一种写法:

```
SELECT DISTINCT s.sname AS 姓名
FROM s,sc,c
```

## 11 连接查询

```
WHERE s.sno = sc.sno AND sc.cno = c.cno AND
(
  s.sno NOT IN (
    SELECT sc.sno FROM sc
    WHERE sc.cno IN (
      SELECT cno FROM c
      WHERE cname='操作系统' )
    )
  );
```

### 4. 查询“操作系统”成绩比“数据库”成绩好的学生姓名

这一查询涉及到 s 表中的姓名、sc 表中的成绩以及 c 表中的课程名。

```
testdb=# SELECT DISTINCT s.sname AS 姓名
testdb=# FROM s,sc sc1,sc sc2
testdb=# WHERE s.sno = sc1.sno AND s.sno = sc2.sno AND
testdb=# sc1.cno = (
testdb=# SELECT cno
testdb=# FROM c
testdb=# WHERE cname='操作系统' ) AND
testdb=# sc2.cno = (
testdb=# SELECT cno
testdb=# FROM c
testdb=# WHERE cname='数据库' ) AND
testdb=# sc1.grade > sc2.grade;
姓名
-----
李小波
(1 row)
```

各课程的成绩存放在 sc 表的各个记录中,因此本例需要进行同一个表不同记录字段之间的纵向比较。由于 SQL 语言不提供纵向比较操作命令或运算符,因此需要用到一些变通的技巧。基本思路是:通过选择操作生成两个虚拟的表 sc1 和 sc2,前者存放“操作系统”课程成绩,后者存放“数据库”课程的成绩;利用这两个虚拟表,将表内的纵向比较转化成表之间的横向比较,也就是将 sc1 和 sc2 连接起来并以“sc1.grade>sc2.grade”为过滤条件,从而得到满足要求的查询结果。

这个例子具有一定的代表性,所有涉及到纵向比较的查询都可以采用这种方法实现。

### 5. 查询至少选修两门课程的学生姓名和性别

这一查询涉及到 s 表中的姓名、性别以及 sc 表中的课程号。

```
testdb=# SELECT DISTINCT s.sname AS 姓名,s.sex AS 性别
```

```

testdb=# FROM s,sc
testdb=# WHERE s.sno= sc.sno AND s.sno IN (
testdb=# SELECT DISTINCT sno FROM sc
testdb=# GROUP BY sno
testdb=# HAVING count(*)>=2);
 姓名  | 性别
-----+-----
李小波 | 男
王前   | 男
仲林   | 女
(3 rows)

```

这个例子涉及到集聚结果的限制条件。可以这样来理解查询条件：根据 sc 表得到所有选修了课程的学生，然后对结果按学生进行分组，最后将含有两条或两条以上记录的组作为最终的结果。实现这个查询的关键是 GROUP BY 与 HAVING 的组合。

#### 6. 查询没有选修李老师所讲课程的学生姓名、性别和所在的系

```

testdb=# SELECT DISTINCT s.sname AS 姓名,
testdb=# s.sex AS 性别,s.dept AS 系别
testdb=# FROM s,sc
testdb=# WHERE s.sno= sc.sno AND sc.sno NOT IN (
testdb=# SELECT sc.sno
testdb=# FROM sc,c
testdb=# WHERE sc.cno = c.cno AND
testdb=# c.teacher LIKE '李%'
testdb=# );
 姓名  | 性别 | 系别
-----+-----+-----
期荣强 | 男   | 机械
李源   | 男   | 机械
刘淇   | 女   | 计算机
徐营串 | 男   | 计算机
苏强   | 女   | 机械
王吉林 | 女   | 机械
钟恢复 | 男   | 机械
(7 rows)

```

本例的查询思路是：先求出选修了李老师所讲课程的学生集合，然后在总的学生集合中去掉前面的集合，结果就是没有选修李老师所讲课程的学生。

## 7. 查询选修李老师所讲课程的学生数

```
testdb=# SELECT count(*)
testdb=# FROM sc,c
testdb=# WHERE sc.cno = c.cno AND
testdb=# c.teacher LIKE '李%';
count
-----
      5
(1 row)
```

这是一个涉及到聚集函数的查询。

## 8. 查询对于同一门课程，比所有女生成绩都好的男生姓名

```
testdb=# SELECT s1.sname
testdb=# FROM sc sc1,s s1
testdb=# WHERE sc1.sno = s1.sno AND s1.sex='男' AND
testdb=# sc1.grade>(SELECT max(sc2.grade)
testdb=# FROM sc sc2,s s2
testdb=# WHERE sc2.sno=s2.sno AND s2.sex='女' AND
testdb=# sc2.cno = sc1.cno
testdb=# );
sname
-----
王前
钟恢复
(2 rows)
```

这也是一个比较复杂的查询，需要用到多重的表连接操作。为了便于分析，可以将查询要求改为“对于同一门课程，查询比成绩最好的女学生的成绩还要好的男学生”。第一重连接操作找出所有男学生的集合，第二重连接找出某门课程成绩最好的女学生的集合；然后对这两个集合进行比较，找出第一个集合中成绩比第二个集合中同门课程成绩好的学生集合，生成最终的结果。

从理论上讲，只要运用得当，连接操作可以实现大多数复杂的查询。

## 第12章

# 集合查询

---

集合查询是SQL语言的另外一种重要查询手段，本章介绍 PostgreSQL所提供的两种级别的集合查询方法，主要包括：

集合运算简介

表集合查询

字段集合查询





## 12.1 集合运算简介

SQL 提供有两种级别上的集合运算：表集合运算和字段集合运算。

### 12.1.1 表集合运算

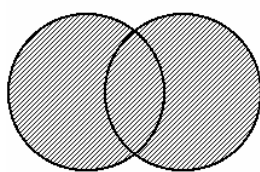
表一级的集合运算利用 UNION (并)、INTERSECT (交) 和 EXCEPT (差) 关键字实现，主要用于将两个或多个查询结果集合合并为一个集合。

#### 1. UNION (并)

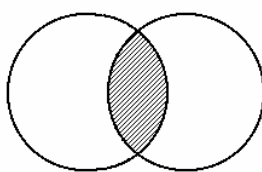
设 A 和 B 是两个具有相同结构的集合，A 与 B 进行 UNION 运算标记为：

$A \cup B$

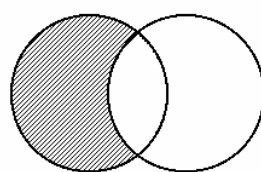
运算的结果是一个集合，结果集合中既包含集合 A 的元素又包含集合 B 的元素，如图 12-1 (a) 中的阴影部分所示。



(a) UNION 运算



(b) INTERSECT 运算



(c) EXCEPT 运算

图 12-1 表的集合运算

#### 2. INTERSECT (交)

设 A 和 B 是两个具有相同结构的集合，A 与 B 进行 INTERSECT 运算标记为：

$A \cap B$

运算的结果是一个集合，结果集合中包含既存在于集合 A 又存在于集合 B 的元素，如图 12-1 (b) 中的阴影部分所示。

#### 3. EXCEPT (差)

设 A 和 B 是两个具有相同结构的集合，A 与 B 进行 EXCEPT 运算标记为：

$A - B$

运算的结果是一个集合，结果集合中包含存在于集合 A 但不存在于集合 B 的元素，如图 12-1 (c) 中的阴影部分所示。

所有的集合运算都可以级联使用，如  $A \cup B \cap C$ 、 $A \cap B - C$  等。

### 12.1.2 字段集合运算

字段集合运算主要用于判断一个字段对象的值是否在一个集合中，运算结果为逻辑值（真或假）。第 11.3 节已经介绍了一些利用 IN（或 NOT IN）运算符进行字段集合运算的例子。IN（或 NOT IN）只是字段集合运算中最简单的一种，其他字段集合运算还有 ANY、ALL 和 EXISTS。

#### 1. ANY

ANY运算的基本语法格式是：

```
col operator ANY(...)
```

其中，col为字段名，operator为比较运算符，（...）为一个集合（它通常由一个“SELECT.....”语句产生）。

以“=”比较符为例，对于“col=ANY（...）”形式的运算，如果col等于集合“（...）”中的任意一个元素，则结果为真。例如，col=ANY(5,7,9)，如果col的值等于5、7、9中的任何一个，运算结果为真，否则结果为假。这种形式的运算等同于IN运算。

#### 2. ALL

ALL运算的基本语法格式是：

```
col operator ALL(...)
```

其中，col为字段名，operator为比较运算符，（...）为一个集合（它通常由一个“SELECT.....”语句产生）。

以“<>”比较符为例，对于“col<>ALL（...）”形式的运算，如果col不等于集合“（...）”中的所有元素，则结果为真。例如，col<>ALL(5,7,9)，如果col的值不等于5、7、9中的任何一个，运算结果为真，否则结果为假。这种形式的运算等价于NOT IN运算。

#### 3. EXISTS

EXISTS运算的基本语法格式是：

```
EXISTS(...)
```

其中（...）为一个集合，它通常由一个“SELECT.....”语句产生。如果集合中存在一个或多个记录，运算结果为真，否则运算结果为假。

## 12.2 表集合查询

### 12.2.1 UNION 查询

在 SQL 语言中,参与 UNION 运算的集合由 SELECT 语句产生,运算的结果就是最终查询的结果。基本形式是:

```
SELECT .....  
UNION [ALL]
```

```
SELECT .....
```

其中,“SELECT.....”可以是任意复杂的查询,“ALL”用于控制同时出现在两个“SELECT.....”结果集中的记录。

下面的例子查询选修了“操作系统”或“数据库”课程的学生。

```
testdb=# SELECT s.sname AS 姓名  
testdb=# FROM s,sc,c  
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno AND  
testdb=# c.cname = '操作系统'  
testdb=# UNION  
testdb=# SELECT s.sname AS 姓名  
testdb=# FROM s,sc,c  
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno AND  
testdb=# c.cname = '数据库';
```

姓名

-----

黄启

李小波

刘淇

徐营串

王前

亢亮

仲林

(7 rows)

可以看出，结果集合中删除了重复记录（“李小波”、“王前”和“仲林”）。可以利用可选项“ALL”来避免这种删除。

```
testdb=# SELECT s.sname AS 姓名
testdb=# FROM s,sc,c
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno AND
testdb=# c.cname = '操作系统'
testdb=# UNION ALL
testdb=# SELECT s.sname AS 姓名
testdb=# FROM s,sc,c
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno AND
testdb=# c.cname = '数据库';
 姓名
-----
李小波
王前
仲林
黄启
元亮
李小波
王前
仲林
刘淇
徐营串
(10 rows)
```

### 12.2.2 INTERSECT 查询

参与 INTERSECT 运算的集合由 SELECT 语句产生，运算的结果就是最终查询的结果。基本形式是：

```
SELECT .....
INTERSECT
SELECT .....
```

其中“SELECT.....”可以是任意复杂的查询。

下面的例子查询同时选修了“操作系统”和“数据库”课程的学生。

```
testdb=# SELECT s.sname AS 姓名
testdb=# FROM s,sc,c
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno AND
testdb=# c.cname = '操作系统'
testdb=# INTERSECT
```

## 12 集合查询

```
testdb=# SELECT s.sname AS 姓名
testdb=# FROM s,sc,c
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno AND
testdb=# c.cname = '数据库';
姓名
-----
李小波
王前
仲林
(3 rows)
```

如果使用非表级集合查询，则上面查询的实现比较复杂。

### 12.2.3 EXCEPT 查询

参与 EXCEPT 运算的集合由 SELECT 语句产生，运算的结果就是最终查询的结果。基本形式是：

```
SELECT .....
EXCEPT
SELECT .....
```

其中“SELECT.....”可以是任意复杂的查询。

对于第 11.3 节中的第 2 个例子，也可以利用 EXCEPT 查询来实现。

```
testdb=# SELECT sname AS 姓名
testdb=# FROM s
testdb=# EXCEPT
testdb=# SELECT s.sname AS 姓名
testdb=# FROM s,sc
testdb=# WHERE s.sno = sc.sno AND
testdb=# sc.grade<80;
姓名
-----
李小波
仲林
黄启
刘淇
期荣强
钟恢复
王吉林
苏强
(8 rows)
```

不难看出，这里的查询语句比 11.3 节中第 2 个例子所用的查询语句更容易理解。同样，对于 11.3 节中的第 3 个例子和第 6 个例子，都可以利用 EXCEPT 来实现。

--查询没有选修“操作系统”课程的学生

```
testdb=# SELECT sname AS 姓名
testdb=# FROM s
testdb=# EXCEPT
testdb=# SELECT s.sname AS 姓名
testdb=# FROM s,sc,c
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno AND
testdb=# c.cname = '操作系统';
姓名
```

-----

刘淇  
徐营串  
期荣强  
钟恢复  
王吉林  
苏强  
李源

(7 rows)

--查询没有选修李老师所任课程的学生姓名、性别和所在的系

```
testdb=# SELECT DISTINCT sname AS 姓名,
testdb=# sex AS 性别,
testdb=# dept AS 系别
testdb=# FROM s
testdb=# EXCEPT
testdb=# SELECT DISTINCT sname AS 姓名,
testdb=# sex AS 性别,
testdb=# dept AS 系别
testdb=# FROM s,sc,c
testdb=# WHERE s.sno= sc.sno AND sc.cno = c.cno AND
testdb=# c.teacher LIKE '李%';
姓名 | 性别 | 系别
```

-----+-----+-----

期荣强	男	机械
李源	男	机械
刘淇	女	计算机
徐营串	男	计算机
苏强	女	机械

## 12 集合查询

```
王吉林  | 女  | 机械
钟恢复  | 男  | 机械
(7 rows)
```

### 12.3 字段集合查询

本节重点介绍利用 ANY、ALL、EXISTS 进行查询的方法。

#### 12.3.1 ANY 查询

对于 11.3 节中的第 3 个例子，利用 ANY 运算实现查询的方法是：

--查询所有没有选修“操作系统”课程的学生姓名

```
testdb=# SELECT sname
testdb=# FROM s
testdb=# WHERE NOT sno = ANY (
testdb=# SELECT DISTINCT sno
testdb=# FROM sc,c
testdb=# WHERE sc.cno = c.cno AND cname='操作系统'
testdb=# );
sname
-----
刘淇
徐营串
期荣强
钟恢复
王吉林
苏强
李源
(7 rows)
```

#### 12.3.2 ALL 查询

对于 11.3 节中的第 2 个例子，利用 ALL 运算实现查询的方法是：

--查询所有成绩都在 80 分以上的学生姓名及所在的系

```
testdb=# SELECT sname,dept
testdb=# FROM s
testdb=# WHERE sno <> ALL (
```

```

testdb=# SELECT DISTINCT sno
testdb=# FROM sc
testdb=# WHERE grade<80
testdb=# );
  sname  | dept
-----+-----
李小波   | 计算机
仲林     | 计算机
黄启     | 计算机
刘淇     | 计算机
期荣强   | 机械
钟恢复   | 机械
王吉林   | 机械
苏强     | 机械
(8 rows)

```

对于 11.3 节中的第 3 个例子，利用 ALL 运算实现查询的方法是：

--查询所有没有选修“操作系统”课程的学生姓名

```

testdb=# SELECT sname
testdb=# FROM s
testdb=# WHERE sno <> ALL (
testdb=# SELECT DISTINCT sno
testdb=# FROM sc,c
testdb=# WHERE sc.cno = c.cno AND cname='操作系统'
testdb=# );
  sname
-----
刘淇
徐营串
期荣强
钟恢复
王吉林
苏强
李源
(7 rows)

```

对于 11.3 节中的第 8 个例子，利用 ALL 运算实现查询的方法是：

--查询对于同一门课程，比所有女生成绩都好的男生姓名

```

testdb=# SELECT sname,dept
testdb=# FROM s s1, sc s1
testdb=# WHERE s1.sno = s1.sno AND s1.sex = '男' AND

```



## 12 集合查询

```
testdb=#      sc1.grade > ALL (
testdb=#      SELECT sc2.grade
testdb=#      FROM s s2, sc sc2
testdb=#      WHERE s2.sno = sc2.sno AND s2.sex = '女' AND
testdb=#      sc1.cno = sc2.cno
testdb=# );
  sname  | dept
-----+-----
  王前   | 计算机
  钟恢复 | 机械
(2 rows)
```

### 12.3.3 EXISTS 查询

对于 11.3 节中的第 3 个例子，利用 EXISTS 运算实现查询的方法是：

--查询所有没有选修“操作系统”课程的学生姓名

```
testdb=# SELECT sname
testdb=# FROM s
testdb=# WHERE NOT EXISTS (
testdb=#     SELECT '-'
testdb=#     FROM sc,c
testdb=#     WHERE sc.cno = c.cno AND
testdb=#           c.cname = '操作系统' AND
testdb=#           s.sno = sc.sno
testdb=# );
  sname
-----
  刘淇
  徐营串
  期荣强
  钟恢复
  王吉林
  苏强
  李源
(7 rows)
```

## 第13章

# 唯一性编号

---

唯一性编号主要用于唯一地标识表中的记录。本章主要介绍PostgreSQL所提供的各种产生唯一性编号的方法，主要内容包括：

- 安装前的准备工作
- 第一安装阶段
- 第二安装阶段
- 安装后配置系统的方法
- rpm 的使用



## 13.1 对象标识编号

### 13.1.1 关于对象标识编号

PostgreSQL表的每一行都有一个唯一的编号，它就是对象标识编号——OID (Object Identification number)。这个编号由PostgreSQL自动产生，并且在通常情况下是不可见的。

当使用initdb命令对PostgreSQL进行初始化时，该命令自动生成一个用于OID的计数器，并将这个计数器的值设置为17000左右的一个整数。小于17000的整数通常由PostgreSQL内部使用，而用户表的OID则大于17000。虽然数据库和表会不断地被创建和删除，但此计数器的值只会不断地增加而不会减少。它的作用范围为所有的数据库，因此产生的OID是唯一的，也就是说，任何两个表的行或同一个表的两个不同行具有不相同的OID。

前面曾提到过，OID在通常情况下是不可见的，其意是指，当使用SELECT命令进行查询时，结果中没有任何与OID有关的内容。但是，实际上OID的值确实是与表的数据存放在一起的，从INSERT命令的执行结果中可以看到这一点。

```
INSERT INTO student VALUES(.....);  
INSERT 20486 1
```

INSERT命令的执行结果包括三个部分，其中“INSERT”是执行的命令名称，“20486”就是OID，“1”是插入的数据行数。每执行一次INSERT命令，都会有类似的结果，这说明任何插入到PostgreSQL数据库中的数据行都有一个唯一的OID。例如：

```
INSERT INTO student VALUES('张三',.....);  
INSERT 20486 1  
  
INSERT INTO student VALUES('李四',.....);  
INSERT 20487 1
```

如果只有一个用户对数据库进行了插入操作，则产生的OID按顺序递增。

通常情况下，OID只在INSERT的返回结果中自动显示出来。在其他命令中，如果要得到OID，必须将OID当作表的一列，写在命令的适当位置上。例如：

```
testdb=# CREATE TABLE oidtest(age INTEGER);  
CREATE  
  
testdb=# INSERT INTO oidtest VALUES(7);  
INSERT 20681 1  
  
testdb=# SELECT oid,age FROM oidtest;  
oid | age  
-----+-----
```

```
20681 | 7
(1 row)
```

虽然在CREATE TABLE命令中没有显式地指定OID列属性，但PostgreSQL自动地为每一个表添加了一个名为OID的列，并且可以只在需要时才访问这个列。例如，下面命令的结果中不会出现OID的值。

```
SELECT * FROM student;
```

以下命令的结果中才会出现OID。

```
SELECT oid,* FROM student;
```

### 13.1.2 对象标识编号的用途

OID 可用作表的主关键字，也可以用作连接外关键字。由于每一行都有一个唯一的OID，因此没有必要增加额外的、用来唯一地标识每一行的字段。

例如，对于消费者（customer）表，许多人为之设计一个字段 customer\_id，存放消费者的编号，以便区分不同的消费者。实际上，这一列是没有必要的，它的任务完全可以由OID来承担。

在这种情况下，customer.oid 唯一性地标识每一个消费者。

OID 类型是 PostgreSQL 特有的数据类型，专门用于描述存放 OID 数据的字段属性。值得注意的是，OID 字段与 OID 类型是两个完全不同的概念。前者是一个由 PostgreSQL 自动产生的、唯一标识一个数据行的列名称，每一个 PostgreSQL 都有一个这样的字段，它的类型由 PostgreSQL 自动指定（为 OID 型）；后者是一个 PostgreSQL 特有的数据类型，任何整数字段的类型都可以指定为 OID，但并不表示它们是唯一性的“对象标识编号”，每一个表可以有零个、一个或多个用户自定义的 OID 类型的字段。

在 PostgreSQL 中，除名字为 OID 的字段外，系统不为具有 OID 类型的字段自动指定任何值。换句话说，除名为 OID 的字段外，其他 OID 类型的字段没有任何的特殊性。

### 13.1.3 对象标识编号的局限性

虽然 OID 能够简化许多数据库设计工作，但也存在一些局限性。

#### 1. 非连续性

用于产生 OID 的计数器是全局性的，系统中的所有表共用这个计数器。这一特性决定了 OID 的非连续性，即同一个表中相邻两行的 OID 不一定是连续的。例如：

```
INSERT INTO oidtest VALUES(1);
INSERT 21518 1
INSERT INTO student VALUES('aaa',...);
INSERT 21519 1
INSERT INTO oidtest VALUES(2);
```

```
INSERT 21520 1
```

.....

在上面的例子中，共插入了三行数据。第一行和第三行数据插入到 `oidtest` 表中，而第二行数据插入到 `student` 表中。这样，`oidtest` 表中相邻两行（1 和 2）的 OID 不连续。不管插入顺序如何，OID 总是按递增的规律增加。

如果仅将 OID 用于系统内部，而不显示在用户的终端上，OID 的这种不连续性也许不是什么问题。反之，OID 的不连续性可能会使用户感到迷惑不解。

### 2. 不可更改性

每一行数据的 OID 是在数据的插入期间由系统自动指定的。只要该行数据没有被删除，它的 OID 会一直存在，并且任何命令（包括 `UPDATE` 在内）都不可能改变 OID 的值。这意味着，用户无法按自己的意愿指定或改变数据行的 OID。

### 3. 不可备份性

在对数据库进行备份时，默认情况下，OID 是不能自动地复制到备份介质中的。如果想将 OID 连同数据一起备份，必须使用特殊的命令可选项。

## 13.2 序列

### 13.2.1 关于序列

由于OID的一些局限性，许多情况下需要借助其他方法标识数据行，序列(Sequences)就是常用的方法之一。与OID一样，序列可以用作数据行的唯一性编号，但比OID有更好的性质。

序列是PostgreSQL的一种特殊计数器，它由用户创建，由系统管理。创建一个序列后，可以将其作为某个字段的默认值。在INSERT命令的执行期间，序列的值会被自动生成，并作为数据行的唯一性编号，存放在该数据行相应的字段中。

一个序列通常只与某个表相关联，因此避免了唯一性编号的不连续性，即同一个表相邻两行的序列编号在数值上是连续的，所以非常适合于用作对用户可见的唯一性数据行编号。同样以customers表为例，即使两个消费者的数据在不同的时间插入到数据表中，它们的序列编号在数值上也是连续的，这是因为没有其他表与之共享序列计数器。

### 13.2.2 序列的创建

与OID不同，序列不能由系统自动创建，而必须用CREATE SEQUENCE命令显式地创建。

命令格式是：

```
CREATE SEQUENCE <序列名>;
```

其中序列名必须是合法的PostgreSQL标识符。序列一旦创建成功后，就可以用于各种需要产生唯一性编号的场合，包括INSERT命令、SELECT命令等。表13-1列出了三个常用的序列操作函数。

表13-1 序列操作函数

函数	功能
nextval(<序列名>)	函数返回下一个可用的序列计数器的值，并自动将序列计数器的值加 1
currval(<序列名>)	函数返回当前可用的序列计数器的值，并且不修改序列计数器的值
setval(<序列名>,新计数器值)	函数将序列计数器的值设置为指定的值

例如：

```
--创建序列seqtest
testdb=# CREATE SEQUENCE seqtest;
CREATE
--取序列的下一个计数值，并将计数器的值加1
testdb=# SELECT nextval('seqtest');
nextval
-----
1
(1 row)
--取序列的下一个计数值，并将计数器的值加1
testdb=# SELECT nextval('seqtest');
nextval
-----
2
(1 row)
--取序列的当前计数值，计数器的值保持不变
testdb=# SELECT currval('seqtest');
currval
-----
2
(1 row)
--将序列的计数器值设置为100
testdb=# SELECT setval('seqtest',100);
setval
-----
100
(1 row)
--取序列的下一个计数值，并将计数器的值加1
testdb=# SELECT nextval('seqtest');
```

```
nextval
-----
      101
(1 row)
```

在上面的例子中，第一条命令创建一个名为“seqtest”的序列，接下来的几条命令通过SELECT调用各种序列函数。对命令的执行结果进行分析不难发现，一个序列创建成功后，其计数器的值为0；每调用一次nextval会导致计数器的值加1；setval函数改变序列计数器的值，而currval函数不会改变序列计数器的值。

## 13.2.3 序列在连续唯一性行号中的应用

利用序列为数据表产生唯一性行号的操作步骤是：

### 1. 创建一个序列

例如：

```
testdb=# CREATE SEQUENCE customer_seq;
CREATE
```

为了保证数据行编号的唯一性和连续性，必须确保一个数据表对应一个序列。如果多个数据表共用同一个序列，将不能保证行号的连续性。

### 2. 创建数据表，并将其中一列的默认值定义为 nextval

例如：

```
testdb=# CREATE TABLE customer(
testdb(#   customer_id INTEGER DEFAULT nextval('customer_seq'),
testdb(#   name CHAR(30)
testdb(# );
CREATE
```

### 3. 插入数据

在插入数据的过程中，要么调用nextval函数指定与序列相关的列值，要么不为该列指定任何值，否则不能保证数据列的值的唯一性和连续性。当没有显式地为与序列相关的列指定数值时，该列的值取默认值，即由nextval函数产生的值。例如：

```
testdb=# INSERT INTO customer VALUES(nextval('customer_seq'),
testdb(#   'name 1');
INSERT 20727 1

testdb=# INSERT INTO customer(name) VALUES('name 2');
INSERT 20728 1

testdb=# INSERT INTO customer(name) VALUES('name 3');
```

```
INSERT 20729 1
testdb=# SELECT * FROM customer;
customer_id |          name
-----+-----
          1 | name 1
          2 | name 3
          3 | name 3
(3 rows)
```

最后需要提醒读者的是，OID与序列有着本质的不同。一方面，OID的值由PostgreSQL自动产生，任何数据表都有一个隐含的OID列；序列的值虽然也是由系统自动产生的，但PostgreSQL并不自动地为数据表产生存放序列值的字段。另一方面，OID的值是不可修改的，因此能保证OID值的唯一性；而序列的值是可修改的，因此如果使用不当，序列值将可能不是唯一性的。

## 13.3 序列类型

除序列外，PostgreSQL 提供了另外一种产生唯一性连续编号的机制，它就是序列类型（SCT，Serial Column Type）。SCT 是 PostgreSQL 的一种特殊数据类型，类似于 PARADOX 数据库中的 Autoincrement 类型。当某字段的类型被定义为序列类型时，PostgreSQL 自动为该字段创建一个相应的序列，并且自动地为该字段定义相应的默认值表达式。例如：

```
testdb=# CREATE TABLE sctest(
testdb(#      customer_id SERIAL,
testdb(#      name CHAR(30)
testdb(# );
NOTICE: CREATE TABLE will create implicit sequence 'sctest_customer_id_seq'
for SERIAL column 'sctest.customer_id'
NOTICE: CREATE TABLE/UNIQUE will create implicit index
'sctest_customer_id_key' for table 'sctest'
CREATE
testdb=# \d sctest
Table "sctest"
Attribute | Type | Modifier
-----+-----+-----
customer_id | integer | not null default extval('sctest_customer_id_seq'::te
xt)
name | char(30) |
```



## 13 唯一性编号

Index: sctest\_customer\_id\_key

```
testdb=# INSERT INTO sctest
testdb=# (name) VALUES ('Sname 1');
INSERT 20761 1
testdb=# INSERT INTO sctest
testdb=# (name) VALUES ('Sname 2');
INSERT 20762 1
testdb=# SELECT * FROM sctest;
customer_id |          name
-----+-----
          1 | Sname 1
          2 | Sname 2
(2 rows)
```

从上面的结果可以看出，序列类型实际上并不是一种PostgreSQL基本数据类型，而是序列的一种简写，序列类型最终被转化为具有序列性质的整数类型。除此之外，PostgreSQL 还为具有此类型的列创建一个相应的序列和一个默认值表达式，并为相应的数据表在该列上创建一个唯一性的索引。序列的名字和索引的名字均由PostgreSQL自动产生，序列名字的格式为：

<表名称>\_<列名称>\_seq

索引名字的格式为：

<表名称>\_<列名称>\_key

在多用户环境特别是网络环境下，由于并发的原因，多个用户可能同时对同一个表进行数据更新操作，因此难以通过人工方式为数据列指定唯一性的编号，也难以数据行指定唯一性连续的行号。PostgreSQL 的对象标识编号、序列以及序列类型等机制，为解决这类问题提供了一些方便、高效的途径。只要使用得当，一定会取得事半功倍的效果。

## 第14章

# 提高效率

---

一个精心组织的数据库与一个随意建立的数据库相比，前者的运行效率可能比后者的效率高出成千上万倍。因此，当数据库的规模较大时，必须认真对待数据库系统的运行效率问题。本章主要包括：

索引

集簇

表的清理

查询效率



## 14.1 索引

### 14.1.1 关于索引

一般情况下，PostgreSQL在访问一个表时，总是从表的第一条记录开始扫描整个表，以便找到满足要求的记录。当表的记录数非常多的时候，扫描过程会耗费大量的时间，并且需要进行大量的磁盘操作。

如果建立了合适的索引，PostgreSQL就会在索引中寻找特定的值。由于避免了对整个表的扫描，查询效率将会有大幅度的提高。因此，当数据库的规模较大时，必须认真对待数据库系统的运行效率问题。

假定有一个表idxtest，它包含字段col (integer)，查询命令为“SELECT \* FROM idxtest WHERE col = 10”。如果没有建立索引，为了找到“col=10”的记录，PostgreSQL需要扫描整个表。如果在col字段上建立了索引，PostgreSQL就会跳过所有其他记录，而直接访问“col=10”的那条记录。对于大的表，扫描工作可能需要几分钟甚至更长时间，但如果利用了合适的索引，查询工作将可能只需要一两秒钟。

在PostgreSQL中，创建索引的命令是：

```
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_name ] ( column [ ops_name ] [, ...] )
```

或：

```
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_name ] ( func_name( column [, ...] ) [ ops_name ] )
```

CREATE INDEX 在指定的表 table 上构建一个名为 index\_name 的索引。在第一种形式的命令中，索引键是以字段名的形式指定的，可以为同一个索引指定多个字段。在第二种形式的命令中，索引是以用户定义函数 func\_name 的结果来指定的，该函数作用于一个表的一个或多个字段。利用这些函数索引可获取更快的数据访问速度，在需要进行基本数据转换时，效果更为明显。

PostgreSQL 为索引提供了 btree、rtree 和 hash 三种访问模式。btree 访问模式是 Lehman-Yao 高并发 btrees 的一种实现。rtree 访问模式用 Guttman 的二分算法实现了标准的 rtrees。hash 访问模式是 Litwin 线性散列的一个实现。单独地列出所用的算法是要表明所有这些访问模式都是完全动态的，并且不必进行周期性的优化。

值得注意的是，当一个索引了的字段涉及到 <、<=、=、>= 或 > 比较操作时，PostgreSQL 的查询优化器将考虑在扫描过程中使用 btree 索引。

当一个索引了的字段涉及到 <<、&<、&>、>>、@、~= 或 && 比较操作时，PostgreSQL

的查询优化器将考虑在扫描过程中使用 `rtree` 索引。

当一个索引了的字段涉及到=比较操作时，PostgreSQL 的查询优化器将考虑在扫描过程中使用 `hash` 索引。

目前，只有 `btree` 访问模式支持多字段索引。默认时最多可以指定 16 个索引键（这个限制可以在编译 PostgreSQL 时修改）。

在内部，PostgreSQL 将数据存放在操作系统文件中，每个表都有一个独立的文件，表中的数据一个记录紧接着一个记录存放。索引文件也是独立的操作系统文件，其中存放的内容有索引键的值、指向表文件记录的指针，以及一些索引组织结构数据。

删除索引的命令是：

```
DROP INDEX index_name
```

有关索引更详细的介绍，请参考本书的附录。

### 14.1.2 索引的使用

PostgreSQL 并不为表自动建立索引，因此，如果用户需要索引，必须自行建立。通常情况下，应该以在查询条件（WHERE 子句）中经常出现的字段为索引键，来建立表的索引。

这里以 `friend` 表为例，介绍索引的一些实际用法。下面的命令在 “`name`” 字段上创建索引 “`friend_name_index`”。

```
testdb=# CREATE INDEX friend_name_idx
```

```
testdb=# ON friend(name);
```

```
CREATE
```

```
testdb=# \d friend
```

```
Table "friend"
```

```
Attribute | Type      | Modifier
```

```
-----+-----+-----
```

```
name      | char(8)   |
```

```
city      | char(10)  |
```

```
prov      | char(6)   |
```

```
age       | integer   |
```

```
Index: friend_name_idx
```

一旦建立了索引，PostgreSQL 就会自动在查询过程中使用它。例如：

```
testdb=# SELECT * FROM friend
```

```
testdb=# WHERE name='王芳';
```

```
name | city | prov | age
```

```
-----+-----+-----+-----
```

```
王芳  |      |      |
```

```
(1 row)
```

在对一个记录不是很多的表进行查询时，索引对效率的影响也许并不明显。由于这个索引的索引键只包含 “`name`” 字段，如果查询条件中不含 “`name`” 字段，那么这个索引对查询

效率也没有任何帮助。另外，如果过多的索引也会给运行效率带来负面影响，这是因为每当进行数据更新时，需要同时更新索引。

PostgreSQL允许建立多字段索引，多字段索引的数据以第一个索引字段的名义存放。当第一个索引字段有多个相同的值时，索引的排序工作依据第二个索引字段进行，其他索引字段依次类推。下面是多字段索引的例子。

```
testdb=# CREATE INDEX friend_m_idx
testdb=# ON friend(age,city);
CREATE
testdb=# \d friend
          Table "friend"
Attribute |   Type   | Modifier
-----+-----+-----
name      | char(8)   |
city      | char(10)  |
prov      | char(6)   |
age       | integer   |
Indices: friend_m_idx,
          friend_name_idx
```

这个索引会在下面的查询例子中得到充分利用。

```
testdb=# SELECT * FROM friend
testdb=# WHERE age = 28 AND city = '大冶';
 name | city | prov | age
-----+-----+-----+-----
李志建 | 大冶 | 湖北 | 28
(1 row)
```

```
testdb=# SELECT * FROM friend
testdb=# WHERE age = 5;
 name | city | prov | age
-----+-----+-----+-----
阳阳  | 武汉 | 湖北 | 5
(1 row)
```

在查询条件仅包含“city”字段时，PostgreSQL不会使用索引“friend\_m\_idx”。这是因为没有指定“age”字段的值。因此，在下面的查询中，这个索引对查询效率没有任何帮助。

```
SELECT * FROM friend
WHERE city = '武汉';
```

索引同样有助于提高表连接操作和ORDER BY子句的效率。

### 14.1.3 唯一性索引

唯一性索引类似于普通的索引,唯一的区别是不允许索引键的值在表中重复。也就是说,如果在某个字段上建立了唯一性索引,那么不允许在两个或两个以上记录中,该字段的值相同。利用“UNIQUE”保留字可以创建唯一性索引。

```
testdb=# CREATE TABLE uitest (
testdb=#   id integer);
CREATE
testdb=# CREATE UNIQUE INDEX uitest_u_id_idx
testdb=#   ON uitest(id);
CREATE
testdb=# \d uitest
          Table "uitest"
  Attribute | Type   | Modifier
-----+-----+-----
   id      | integer |
Index: uitest_u_id_idx
```

创建了这个唯一性索引后,如果为不同记录的id字段指定相同值,PostgreSQL将会给出错误信息。例如:

```
testdb=# INSERT INTO uitest
testdb=#   VALUES(1);
INSERT 20789 1
testdb=# INSERT INTO uitest
testdb=#   VALUES(1);
ERROR:  Cannot insert a duplicate key into unique index uitest_u_id_idx
testdb=# SELECT * FROM uitest;
 id
----
  1
(1 row)
```

上面例子中的第二条插入命令不会起作用,因为表中已经有一条记录的“id”字段的值为1。需要说明的是,唯一性索引对空值不作任何限制,也就是说允许多个索引键的值为空。例如:

```
testdb=# INSERT INTO uitest
testdb=#   VALUES(null);
INSERT 20791 1
testdb=# INSERT INTO uitest
testdb=#   VALUES(null);
INSERT 20792 1
```

## 14 提高效率

```
testdb=# SELECT * FROM uitest;
id
----
1
```

(3 rows)

唯一性索引的作用是双重的，既有助于保证数据的唯一性，也有利于提高查询效率。

## 14.2 集簇

对于某些表，即使在查询条件所涉及的字段上建立了索引，查询效率可能仍然很低，原因是索引键的值存在着大量的重复。

例如，假定在friend表的“age”字段上建立了非唯一性索引，但表中存在成千上万条“age”值相同的记录。这些记录可能散布在磁盘的各个角落，因此需要频繁地移动磁盘的磁头，并且需要大量的缓存空间。可以想象，在这种情况下，查询效率不可能很高，即使建立了索引也无济于事。

集簇是一种基于索引的存储管理技术，它根据索引对表的存储结构进行调整，将具有相同索引键值的记录存放在一起。由于相同索引键值的记录存放在同一个磁盘块或相邻的磁盘块中，因此访问它们不需要频繁地移动磁头，所需的缓存空间也要小得多。显然，查询效率也要高的多。

建立集簇的方法有两种。

第一种是用CLUSTER命令，此命令将原表按索引重新排列。这个动作在操作大表时可能会很慢，因为每一行都必须从磁盘存储页面中按索引顺序取出。如果存储页面没有排序，整个表随机地存放在各个页面中，那么每行都要进行磁盘页面操作。虽然PostgreSQL有一个缓冲，但一个大表的主体是不可能全部都放到缓冲去的。

CLUSTER命令的语法格式是：

```
CLUSTER indexname ON table
```

这个命令依据索引“indexname”为表“table”建立集簇。

例如：

```
testdb=# CREATE INDEX friend_age_idx
testdb=# ON friend(age);
CREATE
testdb=# CLUSTER friend_age_idx
testdb=# ON friend;
CLUSTER
```

```
testdb=# \d friend
          Table "friend"
Attribute |  Type  | Modifier
-----+-----+-----
name      | char(8) |
city      | char(10)|
prov      | char(6) |
age       | integer |
Index: friend_age_idx
```

另一种建立集簇的方法是使用下面的命令：

```
SELECT columnlist INTO TABLE newtable
FROM table ORDER BY columnlist
```

这种用法使用子句ORDER BY来匹配索引，在对排序过的数据进行操作时速度要快得多。然后可以删除旧表，用ALTER TABLE/RENAME将temp改为原来的表名，并且重建所有索引。唯一的问题是OID将不能被保留。这时再建立集簇将要快得多，因为大多数堆栈数据已经排过序而且可以使用已有的索引。

集簇是静态建立的，也就是说，当表的内容更新后，PostgreSQL 不会为改变的内容建立集簇，同样也不会对更新的记录重新建立集簇。如果需要的话，可以通过这个命令重新建立集簇。

值得注意的是，一个建立了集簇的表中的内容实际上按索引顺序复制到了一个临时表中，然后重新改为原来的名字。因此，在建立集簇前所有赋予的权限和其他索引都将丢失。

## 14.3 表的清理

在当前事务中，当更新一个数据行时，PostgreSQL会为该行保留一个更新前的副本，并将这个副本的状态标记为“过时”（expired）。其他事务可能需要使用这个“过时”副本。进行删除操作时，处理过程也基本相同，被删除的行并没有真正地从表中删除掉，而是被标记为“过时”。

PostgreSQL有一个称为“VACUUM”的专门机制来清理这些“过时”的数据。正如它的名字所暗示的那样，它的作用是像“真空吸尘器”一样将“过时”的数据从表中清除出去。

VACUUM命令的语法格式是：

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ table ]
VACUUM [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

例如：

```
--整理表friend
testdb=# VACUUM friend;
```



```
VACUUM
```

```
--整理当前数据库的所有表
```

```
testdb=# VACUUM;
```

```
VACUUM
```

VACUUM命令的另外一个用途是为PostgreSQL的查询优化器搜集统计信息,以便更好地进行查询的优化。使用方法是:

```
testdb=# VACUUM ANALYZE;
```

```
VACUUM
```

应该周期性地使用VACUUM命令对表进行清理,以使数据库始终在高效率的状态下运行。

## 14.4 查询分析

如果想了解PostgreSQL执行查询的过程和效率,可以使用EXPLAIN命令。

```
EXPLAIN [ VERBOSE ] query
```

这条命令显示 PostgreSQL 规划器为查询“query”生成的执行规划,执行规划内容包括查询引用的表是如何被扫描的(是简单的顺序扫描,还是索引扫描等),如果引用了多个表,那么系统将采用了什么样的联合算法从每个表中取出所需要的记录。

显示信息中最关键的部分是预计的查询执行开销,就是规划器对执行该查询所需时间的估计(以磁盘页面存取为单位计量)。这个时间包括两个部分:返回第一条记录前的启动时间,以及返回所有记录的总时间。对于大多数查询而言,用户关心的是总时间。但是,在某些情况下,比如一个 EXISTS 子查询,规划器将选择最小启动时间而不是最小总时间,这是因为执行器在获取一条记录后总要停下来。同样,如果用一条 LIMIT 子句限制返回的记录数,规划器会在最终的开销上做一个合理的折衷以便计算哪个规划开销最小。

VERBOSE 选项强迫命令输出规划树在系统内部的完整内容,而不仅仅是一个概要,并且还将它发送给 postmaster 日志文件。通常这个选项只是在对 PostgreSQL 进行调试时才有用。例如:

```
testdb=# EXPLAIN SELECT s.sname,c.cname
```

```
testdb=# FROM sc,s,c
```

```
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno;
```

```
NOTICE: QUERY PLAN:
```

```
Merge Join (cost=4.19..4.44 rows=4 width=72)
```

```
-> Sort (cost=2.82..2.82 rows=7 width=48)
```

```
-> Merge Join (cost=2.50..2.72 rows=7 width=48)
```

```
-> Sort (cost=1.44..1.44 rows=15 width=24)
```

```

-> Seq Scan on sc (cost=0.00..1.15 rows=15 width=24)
-> Sort (cost=1.05..1.05 rows=3 width=24)
-> Seq Scan on c (cost=0.00..1.03 rows=3 width=24)
-> Sort (cost=1.37..1.37 rows=13 width=24)
-> Seq Scan on s (cost=0.00..1.13 rows=13 width=24)

```

EXPLAIN

EXPLAIN可以用于查询效率的评价。在第11章和第12章中可以看到，对于同一个查询，实现的方法有多种，究竟哪一种方法最好呢？EXPLAIN能够给出量化的查询代价，因此，可以利用这个命令来近似地评价各种查询方法的好坏。

--查询没有选修“操作系统”课程的学生

--第一种查询方法

```

testdb=# EXPLAIN SELECT sname AS 姓名
testdb=# FROM s
testdb=# EXCEPT
testdb=# SELECT s.sname AS 姓名
testdb=# FROM s,sc,c
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno AND
testdb=# c.cname = '操作系统';
NOTICE: QUERY PLAN:

```

```

Seq Scan on s (cost=0.00..71.24 rows=13 width=12)
SubPlan
-> Materialize (cost=5.39..5.39 rows=1 width=60)
-> Nested Loop (cost=0.00..5.39 rows=1 width=60)
-> Nested Loop (cost=0.00..2.38 rows=2 width=36)
-> Seq Scan on c (cost=0.00..1.04 rows=1 width=12)
-> Seq Scan on sc (cost=0.00..1.15 rows=15 width=24)
-> Seq Scan on s (cost=0.00..1.13 rows=13 width=24)

```

EXPLAIN

--第二种查询方法

```

testdb=# EXPLAIN SELECT DISTINCT s.sname AS 姓名
testdb=# FROM s,sc,c
testdb=# WHERE s.sno = sc.sno AND sc.cno = c.cno AND
testdb=# (s.sno NOT IN (
testdb=# SELECT sc1.sno FROM sc sc1,c c1
testdb=# WHERE sc1.cno = c1.cno and c1.cname='操作系统')
testdb=# );

```

## 14 提高效率

NOTICE: QUERY PLAN:

```
Unique (cost=35.38..35.39 rows=0 width=60)
  -> Sort (cost=35.38..35.38 rows=4 width=60)
        -> Merge Join (cost=35.10..35.35 rows=4 width=60)
              -> Sort (cost=2.82..2.82 rows=7 width=36)
                    -> Merge Join (cost=2.50..2.72 rows=7 width=36)
                          -> Sort (cost=1.44..1.44 rows=15 width=24)
                                -> Seq Scan on sc (cost=0.00..1.15 rows=15
width=24)
                                      -> Sort (cost=1.05..1.05 rows=3 width=12)
                                            -> Seq Scan on c (cost=0.00..1.03 rows=3
width=12)
                                                  -> Sort (cost=32.28..32.28 rows=13 width=24)
                                                        -> Seq Scan on s (cost=0.00..32.04 rows=13 width=24)
                                                                SubPlan
                                                                      -> Materialize (cost=2.38..2.38 rows=2 width=36)
                                                                              -> Nested Loop (cost=0.00..2.38 rows=2
width=36)
                                                                                        -> Seq Scan on c c1 (cost=0.00..1.04
rows=1 width=12)
                                                                                              -> Seq Scan on sc sc1 (cost=0.00..1.15
rows=15 width=24)
```

EXPLAIN

第一个查询的总开销为71.24(下层节点的开销包含在上层节点中),第二个查询的总开销为35.39。显然,第二种查询方法的代价比第一种方法的代价要少一半。

## 第15章

# 并发控制

---

在网络和多用户环境中，应用程序并发地对数据库进行各种操作，因此需要进行并发控制，以维护数据库的完整性和一致性。本章主要包括：

- 多版本并发控制和锁定基本概念

- 事务的操作

- 事务的可见性

- 锁的使用



## 15.1 多版本并发控制

### 15.1.1 事务简介

数据一致性是指表示同一事物状态的数据，不管出现在何时、何地都是一致的、正确的和完整的。也就是说，数据一致性应在任何机器上提供给用户一致的、可预见的全貌，没有任何导致数据毁坏或丢失的失败操作。

在网络或多用户环境下，数据库是一种共享资源，可以为多个应用程序所共享。这些程序可串行运行，但在许多情况下，由于应用程序涉及的数据量可能很大，数据库中的输入和输出操作相当频繁。为了有效地利用数据库资源，多个程序可能需要同时存取数据库中的信息，这就是数据库的并发操作。此时，如果不对并发操作进行控制，则可能会存取到不正确的数据，并且影响到数据库数据的一致性。

为了保证并发环境下数据的一致性，大多数数据库系统都使用事务来隔离不同的数据库操作。

一个事务是由一组逻辑上相关的 SQL 语句组成，它被用来完成某一个特定的任务。在一个事务结束时，它对数据库状态所作的更改可以被提交（Commit），所有的更改被永久性地写入数据库，并使其他用户可见；事务也可以被全部回滚，这样事务对数据库的任何更改都会取消。

事务的一个重要特性是它的原子性：一个事务要么被成功地提交，数据库将进入一个新的状态；要么被完全取消，数据库保持原有状态不变。这一特性又称为“all-or-nothing”。

### 15.1.2 多版本并发控制

实现事务原子性的方法有多种，PostgreSQL 是利用多版本并发控制（MVCC，Multi-Version Concurrency Control）技术来实现事务原子性的。

在进行数据库操作时，每个事务在开始时都能得到数据库的一个快照，即数据库当前的版本。有了这个快照，对每个数据库会话进行事务隔离时，就可以避免一个事务因其他并发事务更新相同数据而得到不一致的数据。

MVCC 与锁定技术的主要区别是：在 MVCC 中，检索（读）数据的锁请求与写数据的锁请求不冲突，因此读操作不会阻塞写操作，反过来也一样。

### 15.1.3 事务隔离

在并发环境下，并发事务之间可能存在三种必须避免的现象：读污染（dirty reads）

不可重复的读（non-repeatable reads）以及错误读（phantom read）。

所谓的读污染是指一个事务读取了被另一个事务更改了但尚未提交的数据；不可重复的读是指一个事务重新读取以前读取过的数据，但发现该数据已经被另一个已提交的事务修改过；错误读取是指一个事务重新执行一个查询，返回一套符合查询条件的数据行，但发现这些行中插入了被其他已提交的事务提交的数据行。

### 1. 事务隔离级别

为此，SQL 标准定义了四种事务隔离级别，如表 15-1 所示。

表15-1 事务隔离级别

现象 隔离级别	读污染	不可重复的读	错误读
读未提交 (Read uncommitted)	可能	可能	可能
读已提交 (Read committed)	不可能	可能	可能
可重复读 (Repeatable read)	不可能	不可能	可能
可串行化 (Serializable)	不可能	不可能	不可能

PostgreSQL 提供读已提交和可串行化两种隔离级别。

### 2. 读已提交隔离级别

读已提交是 PostgreSQL 的默认隔离级别。当一个事务运行在这个隔离级别时，一个查询只能得到该查询开始之前的数据，无法得到污染数据（读污染）或者在查询执行时其他并发事务提交的数据。

如果一个执行 UPDATE 语句（或 DELETE、SELECT FOR UPDATE）的查询返回的数据行正在被另一个未提交事务更新，那么第二个试图更新此行的事务将等待另一个事务的提交或者回滚。如果发生了回滚，等待中的事务可以继续修改此行。如果发生了提交（并且此行仍然存在，即未被另一个事务删除），这个查询将对该行再执行一次以检查新版本行是否满足查询条件。如果新版本行满足查询搜索条件，那么该行将被更新（或删除或标记为更新）。

值得注意的是，SELECT 或 INSERT 语句执行的结果（在一个查询中）不受并发事务影响。

### 3. 可串行化隔离级别

可串行化是最高的事务隔离级别。当一个事务处于可串行化级别时，一个查询只能得到该事务开始之前提交的数据，无法得到污染数据或事务执行中其他并发事务提交的修改。所以，这个级别模拟串行事务执行，就好像事务被一个接着一个地串行执行，而不是并发地执行。

如果一个正在执行 UPDATE 语句（或 DELETE、SELECT FOR UPDATE）的查询返回的数据行正在被另一个未提交的事务更新，那么第二个试图更新此行的事务将等待另一个事务的提交或者回滚。如果发生了回滚，等待中的事务可以继续修改此行。如果发生一个

并发事务的提交，一个可串行化的事务将被回滚，并返回下面信息。

ERROR: Can't serialize access due to concurrent update

这是因为一个可串行化的事务在可串行化事务开始之后不能更改被其他事务更改过的数据行。

同样，SELECT 或 INSERT 语句执行的结果（在一个查询中）不会受并发事务的影响。

## 15.2 锁定

PostgreSQL 提供多种用来控制对表数据并发访问的锁定模式。有些锁模式下，在查询命令执行之前，PostgreSQL 自动锁定相关的数据库对象；在另外的模式下，数据库对象的锁定是由发出命令的用户施加的。一个事务所要求的所有锁定模式（除了 AccessShareLock 外）都在整个事务期间保持有效。

除了锁定机制以外，短期的共享、排它锁在共享的缓冲池中用于控制对表页面的读、写访问。这些锁在一条记录被读取或者更新后立即释放。

### 15.2.1 表级锁

表级锁作用于整个表，主要的表级锁包括以下几种。

(1) AccessShareLock（访问共享锁）。这是一个内部锁定模式，进行查询时由 PostgreSQL 自动施加在被查询的表上。语句执行完成后，PostgreSQL 自动释放这些锁。这种锁只与 AccessExclusiveLock 冲突。

(2) RowShareLock（行共享锁）。这种锁是 SELECT FOR UPDATE 和 IN ROW SHARE MODE 的 LOCK TABLE 语句所要求的。它与 ExclusiveLock 和 AccessExclusiveLock 冲突。

(3) RowExclusiveLock（行排它锁）。这种锁是 UPDATE, DELETE, INSERT 和 IN ROW EXCLUSIVE MODE 的 LOCK TABLE 语句所要求的。它与 ShareLock、ShareRowExclusiveLock、ExclusiveLock 以及 AccessExclusiveLock 冲突。

(4) ShareLock（共享锁）。这种锁是 CREATE INDEX 和 IN SHARE MODE 的 LOCK TABLE 语句所要求的。它与 RowExclusiveLock、ShareRowExclusiveLock、ExclusiveLock 以及 AccessExclusiveLock 冲突。

(5) ShareRowExclusiveLock（共享行排它锁）。这种锁是 IN SHARE ROW EXCLUSIVE MODE 的 LOCK TABLE 语句所要求的。它与 RowExclusiveLock、ShareLock、ShareRowExclusiveLock、ExclusiveLock 以及 AccessExclusiveLock 冲突。

(6) ExclusiveLock（排它锁）。这种锁是 IN EXCLUSIVE MODE 的 LOCK TABLE 语句所要求的。它与 RowShareLock、RowExclusiveLock、ShareLock、ShareRowExclusiveLock、ExclusiveLock 以及 AccessExclusiveLock 冲突。

(7) AccessExclusiveLock（访问排它锁）。这种锁是 ALTER TABLE、DROP TABLE、

VACUUM 以及 LOCK TABLE 语句所要求的。它与 RowShareLock、RowExclusiveLock、ShareLock、ShareRowExclusiveLock、ExclusiveLock 以及 AccessExclusiveLock 冲突。

注意，只有 AccessExclusiveLock 才能阻塞不带 FOR UPDATE 子句的 SELECT 语句。

### 15.2.2 行级锁

行级锁作用于某一个数据行，当某行的一个字段被更新（删除或标记为更新）时要求使用这种锁。PostgreSQL 在内存中不记录已更新的行，因而对锁定的行数没有任何限制，也没有锁定级别递增。需要注意的是，SELECT FOR UPDATE 会更改所选定的行以标记它们，因而会导致磁盘写动作。行级别的锁不影响数据查询，它们只是用于阻塞对同一行数据的写操作。

### 15.2.3 索引与锁

尽管 PostgreSQL 提供对表数据访问的非阻塞读、写，但并非所有 PostgreSQL 的索引访问模式都能够进行非阻塞读、写。

对于 GiST 和 R-Tree 索引，共享、排它索引级锁用于读、写访问。这些锁在语句完成后释放。

对于 Hash（散列）索引，共享、排它的页面级锁用于读、写访问。这些锁在页面处理完成后释放。页面级锁比索引级锁提供了更好的并发性，但容易产生死锁。

对于 Btree 索引，短期的共享、排它页面级的锁用于读、写访问。锁在索引记录被插入或读取后立即释放。Btree 索引提供最高级的无死锁条件的并发性。

### 15.2.4 数据完整性检查

由于 PostgreSQL 在进行读操作时不锁定数据，因此不管一个事务处在何种隔离级别，该事务读取的数据都可能被另一个事务覆盖。换句话说，如果一条 SELECT 语句返回了一行，这并不意味着在返回该行时该行还存在；也意味着在当前事务提交或者回滚前，不能保证该行不被并发事务删除或更新。这就要求用户在开发应用程序时，适当地考虑数据完整性检查的问题。

要保证一个数据行实际存在并且不会被其他事务更新，必须使用 SELECT FOR UPDATE 或者合适的 LOCK TABLE 语句。当从其他环境向 PostgreSQL 移植可串行化模式应用程序时，一定要考虑这些问题。由于 6.5 版本以前的 PostgreSQL 使用了读动作锁，因而在进行 PostgreSQL 版本升级时也要考虑这些问题。



## 15.3 事务的操作

前面两节介绍了 PostgreSQL 有关并发控制的一些基本知识,接下来结合一些例子,介绍在 PostgreSQL 中进行并发控制的一些具体方法。

在 PostgreSQL 中,默认情况下,一条 SQL 语句是一个独立的事务。在一条 SQL 语句开始执行时,PostgreSQL 自动为其创建一个事务。如果语句执行成功,PostgreSQL 自动提交该语句的事务,否则回滚事务。

然而,许多情况下,一个完整的任务需要多条 SQL 来完成,也就是说,一个事务由多条 SQL 语句组成。显然,PostgreSQL 默认的事务管理方法不能满足要求。为此,PostgreSQL 提供了专门的事务控制命令 BEGIN 和 END。

BEGIN 命令通知 PostgreSQL 一个新的事务开始,它的语法格式是:

```
BEGIN [ WORK | TRANSACTION ]
```

其中保留字 WORK 和 TRANSACTION 是为了兼容而设置的,因此可以认为是多余的。

END 命令提交并终止当前事务,它的语法格式是:

```
END [ WORK | TRANSACTION ]
```

同样,保留字 WORK 和 TRANSACTION 是为了兼容而设置的,因此也可以认为是多余的。如果当前事务未被提交或回滚,END 隐含地执行一条提交命令。

前面曾经介绍过,一个事务可以被提交,也可以被回滚。提交一个事务的命令是:

```
COMMIT [ WORK | TRANSACTION ]
```

如果命令执行成功,所作的任何更改对其他事务都将是可见的,而且能保证在崩溃发生时数据的一致性和完整性。

回滚一个事务的命令是:

```
ROLLBACK [ WORK | TRANSACTION ]
```

本命令取消当前事务对数据库所作的任何更新。与之功能相同的命令还有 ABORT。

例如:

--开始一个事务

```
testdb=# BEGIN;
```

```
BEGIN
```

--插入数据

```
testdb=# INSERT INTO friend
```

```
testdb=# VALUES('TRAC1','1','1',NULL);
```

```
INSERT 21024 1
```

--插入数据

```
testdb=# INSERT INTO friend
```

```
testdb=# VALUES('TRAC2','2','2',NULL);
INSERT 21025 1
--终止一个事务，隐含地执行一条 COMMIT 命令
testdb=# END;
COMMIT
testdb=# SELECT * FROM friend;
   name   |   city   |   prov   | age
-----+-----+-----+-----
XX        |          |          | -10
阳阳      | 武汉     | 湖北     | 5
张健      |          |          | 18
张祥      | 武汉     | 湖北     | 19
刘健康    |          |          | 20
李鸿      |          |          | 22
李江      | 广州     | 广东     | 25
李志建    | 大冶     | 湖北     | 28
王芳      |          |          |
NNC       |          |          |
TRAC1     | 1        | 1        |
TRAC2     | 2        | 2        |
(12 rows)
```

从最后一条查询语句的执行结果可以看出，事务所作的插入被永久性地写入表中。又例如：

```
--开始一个事务
testdb=# BEGIN;
BEGIN
--插入数据
testdb=# INSERT INTO friend
testdb=# VALUES('TRAN3','3','3',NULL);
INSERT 21028 1
--插入数据
testdb=# INSERT INTO friend
testdb=# VALUES('TRAN4','4','4',NULL);
INSERT 21029 1
--回滚事务
testdb=# ROLLBACK;
ROLLBACK
--终止事务
testdb=# END;
```

## 15 并发控制

NOTICE: COMMIT: no transaction in progress

COMMIT

testdb=# SELECT \* FROM friend;

name	city	prov	age
XX			-10
阳阳	武汉	湖北	5
张健			18
张祥	武汉	湖北	19
刘健康			20
李鸿			22
李江	广州	广东	25
李志建	大冶	湖北	28
王芳			
NNC			
TRAC1	1	1	
TRAC2	2	2	

(12 rows)

从最后一条查询语句的执行结果可以看出，事务所作的插入被取消，表的数据保持不变，这是因为事务被 ROLLBACK 回滚。

## 15.4 事务的可见性

### 15.4.1 读提交

除了原子性外，事务的另外一个重要特点是它的可见性。在PostgreSQL中，默认情况下，当前事务能够看到它所作的任何更新，但只有成功提交的事务所作的更新对其他事务才是可见的。表15-2中的例子演示了这一点。

表15-2 事务的可见性

事务 1	事务 2
--事务 1 开始 testdb=# <u>BEGIN;</u> BEGIN	--事务 2 开始 testdb=# <u>BEGIN;</u> BEGIN

续表

事务 1	事务 2
<pre>--查询表 c testdb=# SELECT * FROM c;  cno        cname        teacher -----+-----+-----  03014   操作系统          李玲  03011   数据库            钟福利  10021   机械制图          黄为 (3 rows)</pre>	<pre>--查询表 c testdb=# SELECT * FROM c;  cno        cname        teacher -----+-----+-----  03014   操作系统          李玲  03011   数据库            钟福利  10021   机械制图          黄为 (3 rows)</pre>
<pre>--向表 c 插入一条记录 testdb=# INSERT INTO c testdb=# VALUES('20001','科技英语','李娟'); INSERT 21088 1</pre>	
<pre>--事务 1 能够看到上面的插入 testdb=# SELECT * FROM c;  cno        cname        teacher -----+-----+-----  20001   科技英语          李娟  03014   操作系统          李玲  03011   数据库            钟福利  10021   机械制图          黄为 (4 rows)</pre>	<pre>--事务 2 不能看到事务 1 的插入 --因为事务 1 尚未提交 testdb=# SELECT * FROM c;  cno        cname        teacher -----+-----+-----  03014   操作系统          李玲  03011   数据库            钟福利  10021   机械制图          黄为 (3 rows)</pre>
<pre>--提交事务 1 testdb=# COMMIT;</pre>	
<pre>--事务 1 能够看到上面的插入 testdb=# SELECT * FROM c;  cno        cname        teacher -----+-----+-----  20001   科技英语          李娟  03014   操作系统          李玲  03011   数据库            钟福利  10021   机械制图          黄为 (4 rows)</pre>	<pre>--事务 2 能够看到事务 1 的插入 --因为事务 1 已经提交 testdb=# SELECT * FROM c;  cno        cname        teacher -----+-----+-----  20001   科技英语          李娟  03014   操作系统          李玲  03011   数据库            钟福利  10021   机械制图          黄为 (4 rows)</pre>
	<pre>--终止事务 2 testdb=# END;</pre>

事务1和事务2可以是同一个用户创建的两个不同事务，也可以是不同用户创建的两个不同事务。从表15-1可以看出，当事务1向表c插入了一条记录但尚未提交时，事务2是不能看到新插入的数据的。

这种特性称为事务的隔离性。默认情况下，PostgreSQL的隔离级别为读提交（READ COMMITTED）。这种隔离级别保证了处于一个事务中的用户能够读取其他事务提交的更新，同时能够保证不会读取到未提交事务的更新，以防止用户得到不完整的数据库视图。

## 15.4.2 可串行化

可串行化是PostgreSQL提供的另外一种隔离级别。对于仅含SELECT命令的事务，可串行化隔离级别为用户提供了稳定、一致的数据库视图。对于含有UPDATE和DELETE的事务，情况要复杂一些。可串行化隔离级别迫使对数据库操作的事务按串行方式执行，即一个事务接着一个事务地执行。

如果两个并发事务试图更新同一个数据行，要保证事务的串行性是不可能的。在这种情况下，PostgreSQL强迫一个事务回滚，而让另外一个事务正常执行。

# 15 并发控制

对于纯SELECT事务，如果在事务期间不想看到其他已提交事务的更新，可以使用可串行化隔离。对于UPDATE和DELETE事务，可串行化隔离可以阻止对同一行数据的并发更新。

由于可串行化不是PostgreSQL的默认隔离级别，因此如果想使用这个隔离级别必须显式地设置。将隔离级别设置为可串行化的命令是：

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

“TRANSACTION ISOLATION LEVEL”实际上是PostgreSQL的环境变量，该变量定义PostgreSQL的隔离级别。SET命令将环境变量设置为指定的值。

表15-3演示了可串行化隔离级别的一些特性。

表15-3 可串行化隔离级别

事务 1	事务 2
--事务 1 开始 testdb=# <u>BEGIN;</u> BEGIN	--事务 2 开始 testdb=# <u>BEGIN;</u> BEGIN
--将事务 1 的隔离级别设置为可串行化 testdb=# <u>SET TRANSACTION ISOLATION LEVEL</u> testdb=# <u>SERIALIZABLE;</u> SET VARIABLE	
--查询表 c testdb=# <u>SELECT * FROM c;</u> <div> <div>cno   cname   teacher</div> <div>-----+-----</div> <div>20001   科技英语   李娟</div> <div>03014   操作系统   李玲</div> <div>03011   数据库   钟福利</div> <div>10021   机械制图   黄为</div> <div>(4 rows)</div> </div>	--查询表 c testdb=# <u>SELECT * FROM c;</u> <div> <div>cno   cname   teacher</div> <div>-----+-----</div> <div>20001   科技英语   李娟</div> <div>03014   操作系统   李玲</div> <div>03011   数据库   钟福利</div> <div>10021   机械制图   黄为</div> <div>(4 rows)</div> </div>
-	--向表 c 插入一条记录 testdb=# <u>INSERT INTO c</u> testdb=# <u>VALUES('30001','物理','龚旦');</u> INSERT 21120 1 --提交事务 2 testdb=# <u>COMMIT;</u> COMMIT
--事务 1 不能够看到事务 2 已提交的插入 testdb=# <u>SELECT * FROM c;</u> <div> <div>cno   cname   teacher</div> <div>-----+-----</div> <div>20001   科技英语   李娟</div> <div>03014   操作系统   李玲</div> <div>03011   数据库   钟福利</div> <div>10021   机械制图   黄为</div> <div>(4 rows)</div> </div>	--事务 2 能看到自己的插入 testdb=# <u>SELECT * FROM c;</u> <div> <div>cno   cname   teacher</div> <div>-----+-----</div> <div>30001   物理   龚旦</div> <div>20001   科技英语   李娟</div> <div>03014   操作系统   李玲</div> <div>03011   数据库   钟福利</div> <div>10021   机械制图   黄为</div> <div>(5 rows)</div> </div>
--终止事务 1 testdb=# <u>END;</u> COMMIT	
--新的事务才能看到 c 表的全貌 testdb=# <u>SELECT * FROM c;</u> <div> <div>cno   cname   teacher</div> <div>-----+-----</div> <div>20001   科技英语   李娟</div> <div>30001   物理   龚旦</div> <div>03014   操作系统   李玲</div> <div>03011   数据库   钟福利</div> <div>10021   机械制图   黄为</div> <div>(5 rows)</div> </div>	

从表中的查询结果可以看出，即使事务2已经提交，事务1仍然不能看到事务2所作的更新。这是因为事务1的隔离级别为可串行化。

一旦设置了新的隔离级别，该级别对本次数据库会话的后续事务都将有效，直到本次数据库会话结束为止。将隔离级别改为默认的读提交的命令是：

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

## 15.5 锁的使用

PostgreSQL提供多种锁，但最常用的是排它锁（也称为写入锁）。排它锁主要用于防止其他事务修改某行或表。

例如，被UPDATE和DELETE修改的行在整个事务期间都被自动地施加排它锁，其他事务如果想修改这些行，必须等待锁定事务提交或回滚后才能进行。表15-4说明了这一点。

表15-4 排它锁的使用

事务 1	事务 2
--事务 1 开始 testdb=# BEGIN;	--事务 2 开始 testdb=# BEGIN;
--修改表 c 中 cno 为'30001'的记录 testdb=# UPDATE c testdb=# SET teacher = '周力' testdb=# WHERE cno = '30001'; UPDATE 1 --事务 1 对表 c 修改成功，同时锁定 cno 为'30001'的记录。	
	--修改表 c 中 cno 为'30001'的记录 testdb=# UPDATE c testdb=# SET cname = '大学物理' testdb=# WHERE cno = '30001'; --系统挂起，等待事务 1 的提交，原因是 cno 为'30001'的记录被事务 1 锁定
--事务 1 提交 testdb=# COMMIT;	UPDATE 1 --系统激活，原因是事务 1 的提交解除了对记录的锁定。
COMMIT	

系统挂起仅仅发生在不同事务修改同一行数据时，如果不同事务修改的是不同的数据，则不会出现系统挂起现象。另外，纯SELECT事务不会引起系统的挂起。

以上是默认隔离级别情况下的例子。如果隔离级别为可串行化，则当事务1被提交时，事务2会被自动回滚。

大多数情况下，PostgreSQL提供的自动锁定能够满足要求。然而，在一些特殊情况，需要由用户自行锁定某些数据库对象。

SELECT.....FOR UPDATE命令锁定满足条件的数据行，这些数据行的锁定在事务结束时自动解除。例如：

```
testdb=# BEGIN;
BEGIN
testdb=# SELECT * FROM s
testdb=# WHERE dept = '计算机'
testdb=# FOR UPDATE;
sno | sname | sex | age | dept
```

```
-----+-----+-----+-----+-----+
9803101 | 李小波 | 男 | 20 | 计算机
9803102 | 王前   | 男 | 20 | 计算机
9803103 | 仲林   | 女 | 21 | 计算机
9803104 | 黄启   | 女 | 21 | 计算机
9803105 | 亢亮   | 男 | 23 | 计算机
9803114 | 刘淇   | 女 | 21 | 计算机
9803115 | 徐营串 | 男 | 23 | 计算机
(7 rows)
```

--其他需要修改上述记录的事务均会被挂起

```
testdb=# UPDATE s SET age = 22
testdb=# WHERE sno = '9803102';
UPDATE 1
```

利用为锁定专门设计的命令LOCK可以实现更复杂的锁定任务，关于LOCK的详细介绍请参考附录。

锁定机制如果使用不当，则有可能导致系统的死锁。表15-5演示了一种典型的挂起情况。

表15-5 死锁的例子

事务 1	事务 2
--事务 1 开始 testdb=# BEGIN; BEGIN	--事务 2 开始 testdb=# BEGIN; BEGIN
--修改表 c 中 cno 为'30001'的记录 testdb=# UPDATE c testdb=# SET teacher = '周力' testdb=# WHERE cno = '30001'; UPDATE 1 --事务 1 对表 c 修改成功，同时锁定 cno 为'30001'的记录。	--修改表 c 中 cno 为'20001'的记录 testdb=# UPDATE c testdb=# SET cname = '高等数学' testdb=# WHERE cno = '20001'; UPDATE 1 --事务 2 对表 c 修改成功，同时锁定 cno 为'20001'的记录。
--修改表 c 中 cno 为'20001'的记录 testdb=# UPDATE c testdb=# SET teacher = '万里' testdb=# WHERE cno = '20001'; --本事务挂起，原因是 cno 为'20001'的记录被事务 2 锁定。	
	--修改表 c 中 cno 为'30001'的记录 testdb=# UPDATE c testdb=# SET cname = '普通物理' testdb=# WHERE cno = '30001'; --本事务挂起，原因是 cno 为'30001'的记录被事务 1 锁定。
事务 1 与事务 2 互相等待，均无法继续执行，进入死锁状态	

一旦出现死锁，则只有强行回滚其中的一个事务，否则死锁状态将永远保持下去。避免死锁最简单的方法是，所有事务都按相同的顺序锁定数据库资源。

## 第16章

# 表的维护

---

对表的结构和数据进行维护，是数据库系统管理员的一项重要工作。本章通过一些例子，介绍维护表结构和数据的主要方法，主要包括：

- 修改表结构

- 访问权限

- 继承

- 视图

- 规则

- 临时表

- 消息交换

- 数据的导入和导出





## 16.1 修改表结构

对于一个已创建的PostgreSQL表，不仅可以存取其中的数据，也可以修改其结构。修改表结构的命令是：

```
ALTER TABLE table [ * ]
    ADD [ COLUMN ] column type
ALTER TABLE table [ * ]
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }
ALTER TABLE table [ * ]
    RENAME [ COLUMN ] column TO newcolumn
ALTER TABLE table
    RENAME TO newtable
```

下面用实际的例子来介绍修改表结构的方法。

### 1. 增加一个字段

```
--创建一个测试用的表atest
testdb=# CREATE TABLE atest (col1 integer);
CREATE
--为表atest增加一个字段
testdb=# ALTER TABLE atest
testdb=# ADD col2 char(10);
ALTER
--显示表的结构
testdb=# \d atest
          Table "atest"
Attribute |  Type  | Modifier
-----+-----+-----
col1      | integer |
col2      | char(10) |
```

### 2. 定义或取消字段默认值

```
--为字段col1定义一个默认值10
testdb=# ALTER TABLE atest
testdb=# ALTER col1 SET DEFAULT 10;
```

```
ALTER
--插入测试数据
testdb=# INSERT INTO atest
testdb=# (col2) VALUES('First row');
INSERT 21163 1
--检查测试结果
testdb=# SELECT * FROM atest;
 col1 |    col2
-----+-----
    10 | First row
(1 row)
--取消默认值
testdb=# ALTER TABLE atest
testdb=# ALTER col1 DROP DEFAULT;
ALTER
--插入测试数据
testdb=# INSERT INTO atest
testdb=# (col2) VALUES('Second row');
INSERT 21164 1
--检查测试结果
testdb=# SELECT * FROM atest;
 col1 |    col2
-----+-----
    10 | First row
       | Second row
(2 rows)
```

### 3. 字段更名

```
--将字段col2更名为chcol
testdb=# ALTER TABLE atest
testdb=# RENAME col2 TO chcol;
ALTER
testdb=# \d atest
      Table "atest"
Attribute |   Type   | Modifier
-----+-----+-----
 col1     | integer  |
 chcol    | char(10) |
```

## 4. 表更名

```
--将表atest的名字更名为testalter
testdb=# ALTER TABLE atest
testdb=# RENAME TO testalter;
ALTER
testdb=# SELECT * FROM testalter;
 coll |  chcol
-----+-----
      10 | First row
      | Second row
(2 rows)
```

## 16.2 访问权限

一个表的创建者即是该表的拥有者，只有拥有者才有权访问该表。表的拥有者可以将表的部分或全部访问权限授予其他用户，也可以收回授出的权限。

### 1. 权限授出

GRANT命令用于将某个数据库对象的某些权限授予指定的用户，其语法格式是：

```
GRANT privilege [, ...] ON object [, ...]
    TO { PUBLIC | GROUP group | username }
```

可以授出的权限包括：SELECT（查询表或视图的权限）、INSERT（插入数据的权限）、UPDATE（更新数据的权限）、DELETE（删除数据的权限）、RULE（在表或视图上定义规则的权限）以及 ALL（所有权限）。

例如：

```
testdb=# CREATE TABLE permtest (col INTEGER);
CREATE
--只有当前用户才能使用表 permtest
--将查询权限授予用户 testuser
testdb=# GRANT SELECT ON permtest TO testuser;
CHANGE
--现在 testuser 用户可以查询 permtest 表
--
--将表 permtest 的所有权限授予所有用户
testdb=# GRANT ALL ON permtest TO PUBLIC;
```

CHANGE

--现在所有的用户都可以使用表permtest

## 2. 权限回收

数据库对象的所有者可以回收授出的权限，命令是：

```
REVOKE privilege [, ...]
```

```
ON object [, ...]
```

```
FROM { PUBLIC | GROUP groupname | username }
```

可以回收的权限包括：SELECT（查询表或视图的权限）、INSERT（插入数据的权限）、UPDATE（更新数据的权限）、DELETE（删除数据的权限）、RULE（在表或视图上定义规则的权限）以及 ALL（所有权限）。

例如：

--收回 testuser 用户对 permtest 的查询权限

```
testdb=# REVOKE SELECT
```

```
testdb=# ON permtest
```

```
testdb=# FROM testuser;
```

CHANGE

--收回所有用户对表 permtest 的所有权限

```
testdb=# REVOKE ALL
```

```
testdb=# ON permtest
```

```
testdb=# FROM PUBLIC;
```

CHANGE

## 3. 权限查询

在 psql 中，可以利用 “\z” 命令查询某个或全部数据库对象的权限情况。例如：

```
testdb=# GRANT SELECT,UPDATE
```

```
testdb=# ON permtest
```

```
testdb=# TO testuser;
```

CHANGE

```
testdb=# \z permtest
```

Access permissions for database "testdb"

Relation	Access permissions
permtest	{ "=", "testuser=rw" }

-----+-----

permtest | { "=", "testuser=rw" }

(1 row)

```
testdb=# GRANT ALL
```

```
testdb=# ON permtest
```

```
testdb=# TO testuser;
```

CHANGE

## 16 表的维护

```
testdb=# \z permtest
Access permissions for database "testdb"
Relation | Access permissions
-----+-----
permtest | {"=arwR","testuser=rw"}
(1 row)
```

### 16.3 继承

继承是面向对象技术的一个主要特性，PostgreSQL部分地支持这种特性。  
在PostgreSQL中，通过继承，可以利用一个已经存在的表创建一个新的表。例如：

--friend是一个已经存在的表

```
testdb=# \d friend
      Table "friend"
  Attribute |   Type   | Modifier
-----+-----+-----
name       | char(8)  |
city       | char(10) |
prov       | char(6)  |
age        | integer  |
Index: friend_age_idx
```

--通过继承，创建一个新的表ifriend

```
testdb=# CREATE TABLE ifriend
testdb=# (addr char(20))
testdb=# INHERITS (friend);
CREATE
testdb=# \d ifriend
      Table "ifriend"
  Attribute |   Type   | Modifier
-----+-----+-----
name       | char(8)  |
city       | char(10) |
prov       | char(6)  |
age        | integer  |
addr       | char(20) |
testdb=# SELECT * FROM ifriend;
```

```
name | city | prov | age | addr
```

```
-----+-----+-----+-----+-----
```

```
(0 rows)
```

虽然表的创建语句只为该表指定了一个字段，但由于使用了继承，因此该表包含有父表的所有字段。

利用继承，可以创建一个由表组成的树。最原始的表为树的根，其他通过继承而创建的表是树的中间节点或树的叶子。在查询语句中，如果在父表名的后面接一个“\*”，则可以得到父表及所有子表的数据。例如：

```
--向表ifriend插入数据
```

```
testdb=# INSERT INTO ifriend
```

```
testdb=# (name) VALUES('子表1');
```

```
INSERT 21217 1
```

```
--查询表ifriend
```

```
testdb=# SELECT * FROM ifriend;
```

```
name | city | prov | age | addr
```

```
-----+-----+-----+-----+-----
```

```
子表1 |      |      |     |
```

```
(1 row)
```

```
--查询friend及其子表ifriend
```

```
testdb=# SELECT * FROM friend*;
```

```
name | city | prov | age
```

```
-----+-----+-----+-----
```

```
XX |      |      | -10
```

```
阳阳 | 武汉 | 湖北 | 5
```

```
张健 |      |      | 18
```

```
张祥 | 武汉 | 湖北 | 19
```

```
刘健康 |      |      | 20
```

```
李鸿 |      |      | 22
```

```
李江 | 广州 | 广东 | 25
```

```
李志建 | 大冶 | 湖北 | 28
```

```
王芳 |      |      |
```

```
NNC |      |      |
```

```
TRAC1 | 1 | 1 |
```

```
TRAC2 | 2 | 2 |
```

```
子表1 |      |      |
```

```
(13 rows)
```

```
--创建表jfriend
```

```
testdb=# CREATE TABLE jfriend
```

```
testdb=# (postcode char(6))
```

## 16 表的维护

```
testdb=# INHERITS(ifriend);
CREATE
--向jfriend插入数据
testdb=# INSERT INTO jfriend
testdb=# (name) VALUES ('子子表1');
INSERT 21235 1
--查询jfriend表
testdb=# SELECT * FROM jfriend;
   name   | city | prov | age | addr | postcode
-----+-----+-----+-----+-----+-----
子子表1   |      |      |     |      |
(1 row)

--查询ifriend表及其子表jfriend
testdb=# SELECT * FROM ifriend*;
   name   | city | prov | age | addr
-----+-----+-----+-----+-----
子表1     |      |      |     |
子子表1   |      |      |     |
(2 rows)

--查询friend表及其子表ifriend和jfriend
testdb=# SELECT * FROM friend*;
   name   | city | prov | age
-----+-----+-----+-----
XX        |      |      | -10
阳阳      | 武汉 | 湖北 | 5
张健      |      |      | 18
张祥      | 武汉 | 湖北 | 19
刘健康    |      |      | 20
李鸿      |      |      | 22
李江      | 广州 | 广东 | 25
李志建    | 大冶 | 湖北 | 28
王芳      |      |      |
NNC       |      |      |
TRAC1     | 1    | 1    |
TRAC2     | 2    | 2    |
子表1     |      |      |
子子表1   |      |      |
(14 rows)
```

除 SELECT 语句外，UPDATE、DELETE 以及 ALTER 都支持“\*”。

在 7.1 或更新版本的 PostgreSQL 中，查询语句自动访问所有子表而不必使用 “\*”。如果要阻止对子表的访问，必须使用 ONLY 保留字。

## 16.4 视图

视图是从一个或多个基本表（或视图）导出的“虚表”。之所以称视图为“虚表”，是因为数据字典中只存储视图的定义，而不存放对应的数据，视图的数据是相关基本表中的数据。

视图通常用于为基本表定义一个子集，即从基本表中选择一定数量的字段和满足某种条件的数据行，组成一个新的数据对象。视图是数据库系统提供给用户以多种角度观察数据库中数据的重要机制，体现了数据库的本质和最重要的特色及功能。视图能够简化用户数据模型，提供数据库的逻辑独立性，实现数据共享和安全保密。

视图在使用之前必须创建。视图一旦创建，就可以像基本表一样被查询，也可以在视图上创建新的视图。创建视图的命令是：

```
CREATE VIEW view AS SELECT query
```

其中“SELECT query”是一个任意复杂的查询语句。

删除视图的命令是：

```
DROP VIEW name
```

例如：

```
--创建男学生视图ms
```

```
testdb=# CREATE VIEW ms
```

```
testdb=# AS SELECT * FROM s WHERE sex = '男';
```

```
CREATE 21261 1
```

```
--查询男学生视图
```

```
testdb=# SELECT * FROM ms;
```

sno	sname	sex	age	dept
9803101	李小波	男	20	计算机
9803102	王前	男	20	计算机
9803105	亓亮	男	23	计算机
9803115	徐营串	男	23	计算机
9810101	期荣强	男	19	机械
9810102	钟恢复	男	21	机械
9810105	李源	男	22	机械
9803106	龚安	男	22	计算机系

(8 rows)



## 16 表的维护

--创建所有课程分数在80分以上学生的视图

```
testdb=# CREATE VIEW gt80 AS
testdb=# SELECT s.sname,s.dept
testdb=# FROM s,sc
testdb=# WHERE s.sno = sc.sno AND
testdb=# (s.sno NOT IN (
testdb=# SELECT DISTINCT sc.sno FROM sc
testdb=# WHERE grade<80)
testdb=# );
CREATE 21291 1
testdb=# SELECT DISTINCT * FROM gt80;
sname | dept
```

```
-----+-----
黄启   | 计算机
期荣强 | 机械
李小波 | 计算机
刘淇   | 计算机
苏强   | 机械
王吉林 | 机械
钟恢复 | 机械
仲林   | 计算机
(8 rows)
```

在目前版本的PostgreSQL中，视图是只读的。也就是说，不允许对视图进行插入、删除或更新。

## 16.5 规则

在PostgreSQL中，规则是一组操作的集合，在对数据对象进行查询、更新、插入或删除时，这组操作由规则系统自动执行。也就是说，可以利用规则来修改SELECT、INSERT、UPDATE和DELETE的行为。

定义规则的命令是：

```
CREATE RULE name AS ON event
TO object [ WHERE condition ]
DO [ INSTEAD ] [ action | NOTHING ]
```

删除规则的命令是：

```
DROP RULE name
```

在一个记录被访问、更新、插入或删除时，系统保存一个旧记录（用于检索、更新和删除）和一个新记录（用于更新和追加）。如果 ON 子句所指定的事件和 WHERE 子句所指定的条件对于旧记录都为真，那么动作（action）部分定义的规则就会被自动执行。但是，旧记录的各字段值和新记录将先用 current.attribute-name 和 new.attribute-name 取代。action 部分的命令和事务标识与激活该规则的用户命令相同。

规则的类型有两种：DO 和 INSTEAD。

### 16.5.1 INSTEAD 规则

INSTEAD 规则定义的动作完全取代用户命令。例如：

```
--创建一个用于测试的表
testdb=# CREATE TABLE ruletest (col INTEGER);
CREATE
--定义一个阻止插入新记录的规则
testdb=# CREATE RULE ruletest_insert AS
testdb=# ON INSERT TO ruletest      --为表 ruletest 定义 INSERT 规则
testdb=# DO INSTEAD                --规则的类型为 INSTEAD
testdb=# NOTHING;                  --规则的动作 NOTHING(不执行任何动作)
CREATE 21328 1
--测试规则是否起作用
testdb=# INSERT INTO ruletest
testdb=# VALUES(1);
--插入命令没有任何输出
testdb=# SELECT * FROM ruletest;
col
-----
(0 rows)
--插入命令对表没有产生任何影响
```

按照规则ruletest\_insert的定义，对表ruletest施加的INSERT操作被重新定义为NOTHING。也就是说，该规则阻止了任何利用INSERT命令向表ruletest插入数据的操作。

### 16.5.2 DO 规则

DO规则定义的动作是作为用户命令的附加部分来执行的，下面的例子比较全面地说明了DO规则的用途。

假定这个例子的应用背景为某公司的客户服务中心，表service\_request记录有当前的客户服务请求，service\_request\_log跟踪service\_request表的变化。例子为表service\_request创建了两个DO类型的规则。每当service\_request被更新时，service\_request\_update规则就向service\_request\_log插入一条记录。特殊的保留字“old”用来引用更新之前的字段值，而保留字“new”则用来引用新的查询数据。第二个规则service\_request\_delete，通过向service\_request

## 16 表的维护

\_log插入特定的数据，来跟踪表service\_request的删除操作。为了区分更新和删除操作，对于字段mod\_type，用“U”表示更新，用“D”表示删除。

```
--创建表service_request
testdb=# CREATE TABLE service_request (
testdb=#   customer_id INTEGER,
testdb=#   description text,
testdb=#   cre_user text DEFAULT CURRENT_USER,
testdb=#   cre_timestamp timestamp DEFAULT CURRENT_TIMESTAMP
testdb=# );
CREATE

--创建表service_request_log
testdb=# CREATE TABLE service_request_log (
testdb=#   customer_id INTEGER,
testdb=#   description text,
testdb=#   mod_type char(1),
testdb=#   mod_user text DEFAULT CURRENT_USER,
testdb=#   mod_timestamp timestamp DEFAULT CURRENT_TIMESTAMP
testdb=# );
CREATE

--创建更新规则
testdb=# CREATE RULE service_request_update AS
testdb=#   ON UPDATE TO service_request
testdb=#   DO
testdb=#     INSERT INTO service_request_log (customer_id,
testdb=#       description, mod_type)
testdb=#     VALUES (old.customer_id, old.description, 'U');
CREATE 21358 1

--创建删除规则
testdb=# CREATE RULE service_request_delete AS
testdb=#   ON DELETE TO service_request
testdb=#   DO
testdb=#     INSERT INTO service_request_log (customer_id,
testdb=#       description, mod_type)
testdb=#     VALUES (old.customer_id, old.description, 'D');
CREATE 21359 1

--测试规则的作用
--登记一个客户 72321 的服务请求
testdb=# INSERT INTO service_request (customer_id, description)
testdb=#   VALUES (72321, '修理打印机');
```

```

INSERT 21360 1
--修改客户 72321 的服务请求
testdb=# UPDATE service_request
testdb=# SET description = '修理激光打印机'
testdb=# WHERE customer_id = 72321;
UPDATE 1
--客户 72321 的服务请求已处理，删除这个请求
testdb=# DELETE FROM service_request
testdb=# WHERE customer_id = 72321;
DELETE 1
--客户服务请求的跟踪日志
testdb=# SELECT *
testdb=# FROM service_request_log
testdb=# WHERE customer_id = 72321;
 customer_id| description | mod_type | mod_user |      mod_timestamp
-----+-----+-----+-----+-----
          72321 | 修理打印机 | U       | postgres | 2001-02-13 22:11:34+08
          72321 | 修理激光打印机 | D       | postgres | 2001-02-13 22:12:25+08
(2 rows)

```

### 16.5.3 视图的更新

前面曾经介绍过，目前版本的 PostgreSQL 不支持视图的更新，更确切地讲，视图会忽略 INSERT、UPDATE 和 DELETE。但是，可以利用规则系统间接地实现视图的更新。

例如：

```

--创建一个用于测试的表
testdb=# CREATE TABLE rvTABLE
testdb=# ( col INTEGER);
CREATE
--为表 rvTABLE 创建一个视图
testdb=# CREATE VIEW rvVIEW
testdb=# AS SELECT * FROM rvTABLE;
CREATE 21381 1
--向基本表插入数据
testdb=# INSERT INTO rvTABLE
testdb=# VALUES(1);
INSERT 21382 1
--向视图插入数据
testdb=# INSERT INTO rvVIEW

```

## 16 表的维护

```
testdb=# VALUES(2);
INSERT 21383 1
testdb=# SELECT * FROM rvVIEW;
 col
-----
  1
(1 row)

testdb=# SELECT * FROM rvTABLE;
 col
-----
  1
(1 row)
--用于视图的 INSERT 语句被忽略
--
--为视图定义插入规则
testdb=# CREATE RULE rv_insert AS
testdb=# ON INSERT TO rvVIEW
testdb=# DO INSTEAD
testdb=# INSERT INTO rvTABLE
testdb=# VALUES (new.col);
CREATE 21384 1
--为视图定义更新规则
testdb=# CREATE RULE rv_update AS
testdb=# ON UPDATE TO rvVIEW
testdb=# DO INSTEAD
testdb=# UPDATE rvTABLE
testdb=# SET col = new.col
testdb=# WHERE col = old.col;
CREATE 21385 1
--为视图定义删除规则
testdb=# CREATE RULE rv_delete AS
testdb=# ON DELETE TO rvVIEW
testdb=# DO INSTEAD
testdb=# DELETE FROM rvTABLE
testdb=# WHERE col = old.col;
CREATE 21386 1
--
--向视图插入数据
```

```
testdb=# INSERT INTO rvVIEW
testdb=# VALUES(3);
INSERT 21387 1
--检查插入结果
testdb=# SELECT * FROM rvVIEW ORBEY BY col;
 col
-----
   1
   3
(2 rows)
--插入成功
--
--修改视图中的数据
testdb=# UPDATE rvVIEW
testdb=# SET col = col + 1
testdb=# WHERE col = 1;
UPDATE 1
--检查修改结果
testdb=# SELECT * FROM rvVIEW ORDER BY col;
 col
-----
   2
   3
(2 rows)
--修改成功
--
--删除视图中的数据
testdb=# DELETE FROM rvVIEW;
DELETE 2
--检查删除结果
testdb=# SELECT * FROM rvVIEW ORDER BY col;
 col
-----
(0 rows)
--删除成功
```

规则赋予PostgreSQL更强大和更灵活的功能，这是PostgreSQL的一个主要特色。

## 16.6 临时表

临时表只在一次数据库会话期间存在，数据库会话结束后，它就会被自动删除，并且临时表只对当前事务可见，主要用于存放事务所产生的一些临时结果。

创建临时表的命令是：

```
CREATE TEMPORARY TABLE.....
```

表16-1说明了临时表的主要特性。

表16-1 临时表的例子

事务 1	事务 2
--事务 1 开始 testdb=# <u>BEGIN;</u> BEGIN	--事务 2 开始 testdb=# <u>BEGIN;</u> BEGIN
--事务 1 创建临时表 tpTEST1 testdb=# <u>CREATE TEMPORARY TABLE tpTEST1</u> testdb=# <u>(col INTEGER);</u> CREATE	
--临时表 tpTEST1 只对当前事务可见 testdb=# <u>INSERT INTO tpTEST1</u> testdb=# <u>VALUES(1);</u> INSERT 21417 1 testdb=# <u>SELECT * FROM tpTEST1;</u> col ----- 1 (1 row)	--事务 1 创建的 tpTEST1 对事务 2 不可见 testdb=# <u>INSERT INTO tpTEST1</u> testdb=# <u>VALUES(2);</u> ERROR: Relation 'tpTEST1' does not exist
	--事务 2 创建临时表 tpTEST2 testdb=# <u>CREATE TEMPORARY TABLE tpTEST2</u> testdb=# <u>(col char(10));</u> CREATE
--事务 2 创建的 tpTEST2 对事务 1 不可见 testdb=# <u>INSERT INTO tpTEST2</u> testdb=# <u>VALUES('2');</u> ERROR: Relation 'tpTEST2' does not exist	--临时表 tpTEST2 只对当前事务可见 testdb=# <u>INSERT INTO tpTEST2</u> testdb=# <u>VALUES('TEST2');</u> INSERT 21417 1 testdb=# <u>SELECT * FROM tpTEST2;</u> col ----- TEST2 (1 row)
--提交事务 1 并关闭当前会话 testdb=# <u>COMMIT;</u> COMMIT testdb=# <u>\q</u> --启动一次新的会话 psql testdb testdb=# <u>SELECT * FROM tpTEST1;</u> ERROR: Relation 'tpTEST1' does not exist --临时表被删除	--提交事务 2 并关闭当前会话 testdb=# <u>COMMIT;</u> COMMIT testdb=# <u>\q</u> --启动一次新的会话 psql testdb testdb=# <u>SELECT * FROM tpTEST2;</u> ERROR: Relation 'tpTEST2' does not exist --临时表被删除

就像高级语言子程序中的局部变量一样，临时表是一次数据库会话的局部数据表。正如局部变量一样，不同数据库会话的临时表名可以相同，但不会发生冲突。退出会话后，临时表就不存在了。

## 16.7 消息交换

PostgreSQL允许不同的数据库会话之间交换消息，这对于多用户环境下用户的协调十分有用。例如，当一个用户完成了某个表的更新后，可以将这一事件通知给其他用户。

实现消息交换的命令是：LISTEN和NOTIFY。

LISTEN name

NOTIFY name

其中，name为消息名称。LISTEN命令强迫当前事务侦听指定的消息，NOTIFY用于发送指定名称的消息。表16-2演示了不同数据库会话间的消息传递。

表16-2 消息交换的例子

会话 1	会话 2
--会话 1 开始侦听消息 sgl testdb=# <u>LISTEN sgl</u> ; LISTEN	
	--会话 2 发送消息 sgl testdb=# <u>NOTIFY sgl</u> ; NOTIFY
--会话 1 接收消息 sgl testdb=# <u>;</u> Asynchronous NOTIFY 'sgl' from backend with pid '937' received. --当前会话中的任何命令都可以激活消息的接收	

## 16.8 数据的导入和导出

PostgreSQL提供了各种数据备份措施，表数据的导出和导入就是其中的一种。导出数据的命令是：

```
COPY [ BINARY ] table [ WITH OIDS ]  
TO { 'filename' | stdout }  
[ [USING] DELIMITERS 'delimiter' ]  
[ WITH NULL AS 'null string' ]
```

本命令将一个表的数据导出到一个文件中。将导出数据命令中的“TO”改为“FROM”即为导入数据的命令。

例如：

--创建用于测试的表

testdb=# CREATE TABLE copytest (



## 16 表的维护

```
testdb=# intcol INTEGER, numcol NUMERIC(16,2),
testdb=# textcol TEXT, boolcol BOOLEAN
testdb=# );
CREATE
--插入测试数据
testdb=# INSERT INTO copytest
testdb=# VALUES (1, 23.99, 'fresh spring water', 't');
INSERT 174656 1
testdb=# INSERT INTO copytest
testdb=# VALUES (2, 55.23, 'bottled soda', 't');
INSERT 174657 1
--导出数据
testdb=# COPY copytest TO '/tmp/copytest.out';
COPY
--删除数据
testdb=# DELETE FROM copytest;
DELETE 2
--导入数据
testdb=# COPY copytest FROM '/tmp/copytest.out';
COPY
```

数据的导入/导出不仅可以以表为单位，也可以以数据库为单位。导出数据库的命令是：

```
pg_dump dbname > dbname.pgdump
```

其中，dbname为需要导出的数据库名，导出的数据（包括其他数据库对象）重定向到操作系统文件dbname.pgdump中。导入数据的命令是：

```
cat dbname.pgdump | psql dbname
```

数据库的导入和导出实际上是一种备份数据库或移动数据的方法。但是，当数据库很大时，上面介绍的方法有一个问题。一些版本的Linux（或其他UNIX）对单个文件的大小有一定的限制，例如单个文件不能超过2G。当数据导出命令产生文件的尺寸超过最大文件尺寸时，导出命令将会失败。解决问题的方法有两种。

第一种方法是使用压缩技术对导出文件进行压缩。

导出：`pg_dump dbname | gzip > filename.dump.gz`

导入：`gunzip -c filename.dump.gz | psql dbname`

由于压缩命令的压缩比率有限，因此第一种方法仍然存在导出文件可能过大的问题。

第二种方法，也是比较理想的方法，是将导出文件分割为较小的文件。下面的导出命令将导出文件分割成若干个大小为20M的文件。

导出：`pg_dump dbname | split -b 20m - filename.dump`

导入：`cat filename.dump.* | pgsqldb dbname`

## 第17章

# 约束

---

为了保证数据的一致性和完整性，必须对表的数据进行必要的约束。本章介绍了对表的数据进行约束的几种方法，主要内容包括：

- 非空值约束
- 唯一性约束
- 主键约束
- 外键约束
- 数据检验



## 17.1 非空值约束

前面的章节曾经多次介绍过字段取空值的情况，空值实际上是PostgreSQL为字段赋予的默认值。然而，在有些表中，某些字段是不允许取空值的。例如，在学生基本情况登记表中，学生的姓名是不允许取空值的，因为任何学生都有一个名字。这种字段称为非空字段，相关的约束称为非空值约束。

在为表创建结构时，使用NOT NULL保留字可以防止用户给某字段赋空值。例如：

--创建一个用于测试的表

```
testdb=# CREATE TABLE student
```

```
testdb=#     sno(sno char(7),                --学号
```

```
testdb=#     sname char(8) NOT NULL);      --学生姓名，非空字段
```

```
CREATE
```

--插入一条合法的记录

```
testdb=# INSERT INTO student
```

```
testdb=#     VALUES('9901001','万荣');
```

```
INSERT 21569 1
```

--试图插入一条非法记录

```
testdb=# INSERT INTO student
```

```
testdb=#     (sno) VALUES('9901002');
```

--因为名字为空，因此PostgreSQL拒绝插入

```
ERROR: ExecAppend: Fail to add null value in not null attribute sname
```

```
testdb=# SELECT * FROM student;
```

```
   sno   | sname
```

```
-----+-----
```

```
9901001 | 万荣
```

```
(1 row)
```

对于已经创建的表，PostgreSQL目前还没有提供将允许空值的字段修改为非空字段的命令。为了实现这一点，只能先将原始表复制到一个临时表中，然后删除原始表、重建原始表，最后利用临时表恢复原始表的数据。例如，在student表中，sno字段也不能为空。修改的方法是：

--创建临时表并复制数据

```
testdb=# CREATE TABLE tmp AS
```

```
testdb=#     SELECT * FROM student;
```

```
SELECT
```

```
--删除原始表
testdb=# DROP TABLE student;
DROP
--重建原始表，将sno说明为非空字段
testdb=# CREATE TABLE student
testdb=# (sno char(7) NOT NULL,
testdb=# (sname char(8) NOT NULL);
CREATE
--恢复原始表的数据
testdb=# INSERT INTO student
testdb=# SELECT * FROM tmp;
INSERT 21591 1
--删除临时表
testdb=# DROP TABLE tmp;
DROP
testdb=# SELECT * FROM student;
sno | sname
-----+-----
9901001 | 万荣
(1 row)
```

## 17.2 唯一性约束

许多表的字段值在表中不能重复。例如，在学生表中，每个学生都有一个唯一性的学号。也就是说，不允许两个不同学生的学号相同。这样的字段称为唯一性字段，相关的约束称为唯一性约束。

在PostgreSQL中，利用UNIQUE保留字可以将字段标识为唯一性字段。例如：

```
testdb=# DROP TABLE student;
DROP
testdb=# CREATE TABLE student
testdb=# (sno char(7) NOT NULL UNIQUE, --学号，非空且唯一
testdb=# (sname char(8) NOT NULL); --学生姓名，非空
NOTICE: CREATE TABLE/UNIQUE will create implicit index 'student_sno_key' for
table 'student'
CREATE
--对于唯一性字段，PostgreSQL自动为其建立一个唯一性的索引
```

## 17 约束

--插入一条记录

```
testdb=# INSERT INTO student
testdb=# VALUES('9901001','万荣');
INSERT 21605 1
```

--插入学号重复的记录

```
testdb=# INSERT INTO student
testdb=# VALUES('9901001','李兵');
ERROR: Cannot insert a duplicate key into unique index student_sno_key
```

--插入不成功，因为学号不允许重复

```
testdb=# SELECT * FROM student;
 sno |  sname
```

-----+-----

```
9901001 | 万荣
```

```
(1 row)
```

如果要限定多个字段的组合不能为空，则必须使用下面形式的命令。

```
testdb=# CREATE TABLE uqTEST
testdb=# (col1 integer,
testdb=# col2 integer,
testdb=# UNIQUE (col1,col2));
NOTICE: CREATE TABLE/UNIQUE will create implicit index 'uqtest_coll_key' for
table 'uqtest'
CREATE
```

## 17.3 主键约束

主键（或称为主关键字）约束相当于非空值约束和唯一性约束的组合，这种约束不仅能保证字段值只能取非空值，而且能保证字段的值在表中唯一。

例如：

```
testdb=# DROP TABLE student;
DROP
testdb=# CREATE TABLE student
testdb=# (sno char(7) PRIMARY KEY,      --学号，主键
testdb=# sname char(8) NOT NULL);      --学生姓名，非空
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'student_pkey'
for
table 'student'
```

```

CREATE
--PostgreSQL同样为主键字段创建一个唯一性的索引
--插入合法数据
testdb=# INSERT INTO student
testdb=# VALUES('9902001','李超');
INSERT 21633 1
--
--插入非法数据
testdb=# INSERT INTO student
testdb=# VALUES('9902001','王强');
ERROR: Cannot insert a duplicate key into unique index student_pkey
--插入不成功，因为学号重复
--
--插入非法数据
testdb=# INSERT INTO student
testdb=# VALUES(NULL,'王强');
ERROR: ExecAppend: Fail to add null value in not null attribute sno
--插入不成功，因为学号为空
testdb=# SELECT * FROM student;
   sno   |  sname
-----+-----
 9902001 | 李超
(1 row)

```

如果主键由多个字段组成，则必须使用下面形式的命令。

```

testdb=# CREATE TABLE pkTEST
testdb=# (col1 integer,
testdb=# col2 integer,
testdb=# PRIMARY KEY (col1,col2));
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'pktest_pkey'
for table 'pktest'
CREATE

```

## 17.4 外键约束

外键约束主要用于保证数据库的参照完整性。

以第11章的学生成绩数据库为例。对于sc表，假定插入了一条记录('2001001','13011',80)。

如果在s表中不存在学号为“2001001”的学生，则sc表中这条记录的意义是不完整的。这就说明表sc的参照不完整。对于sc表的字段cno，也存在类似的情况。

外键约束就是为了避免上述情况的发生。在sc表中，字段sno和cno都称为外键。被sc表参照的s.sno和c.cno分别是各自相关表的主键。利用外键，通过表的连接操作就可以找到sc表中sno和cno字段所代表的学生名和课程名。

### 17.4.1 一般使用

下面的例子简要地说明外键的使用方法。

```
--创建表provname
testdb=# CREATE TABLE provname (
testdb(#   code  char(2) PRIMARY KEY,      --省市代码，主键
testdb(#   name  charR(6)                  --省市名称
testdb(# );
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'provname_pkey'
for table 'provname'
CREATE
--插入数据
testdb=# INSERT INTO provname
testdb=#   VALUES('01','北京市');
INSERT 21677 1
--创建表customer
testdb=# CREATE TABLE customer (
testdb(#   cno  integer,                    --消费者代码
testdb(#   name char(30),                  --消费者名称
testdb(#   tel  char(20),                  --电话
testdb(#   prov char(2) REFERENCES provname); --所在省市代码，参照表provname
NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
--插入数据
testdb=# INSERT INTO customer
testdb=#   VALUES(1,'继达公司','66185082','01');
INSERT 21696 1
--数据插入成功，因为参照是完整的
--插入数据
testdb=# INSERT INTO customer
testdb=#   VALUES(2,'网际快车','63105265','02');
ERROR:  <unnamed> referential integrity violation - key referenced from
customer not found in provname
```

--数据插入不成功，因为provname中不存在编号为“02”的省市

```
testdb=# SELECT * FROM customer;
```

cno	name	tel	prov
-----	------	-----	------

1	继达公司	66185082	01
---	------	----------	----

(1 row)

## 17.4.2 主键的更新

如果被参照表的主键被修改，同样会造成参照的不完整性，因此PostgreSQL在默认情况下禁止此类的操作。例如：

```
testdb=# UPDATE provname
```

```
testdb=# SET code = '10'
```

```
testdb=# WHERE code = '01';
```

ERROR: <unnamed> referential integrity violation - key in provname still referenced from customer

--修改被拒绝，因为code为“01”的记录被表customer用于参照。

--

```
testdb=# DELETE FROM provname
```

```
testdb=# WHERE code = '01';
```

ERROR: <unnamed> referential integrity violation - key in provname still referenced from customer

--基于同样的原因，删除操作被拒绝。

```
testdb=# SELECT * FROM provname;
```

code	name
------	------

01	北京市
----	-----

(1 row)

然而，在实际应用中，不允许修改或删除被参照的主键是不现实的。为此，PostgreSQL允许为外键定义当参照主键被更新时的处理方法。例如：

```
testdb=# DROP TABLE customer;
```

NOTICE: DROP TABLE implicitly drops referential integrity trigger from table "provname"

NOTICE: DROP TABLE implicitly drops referential integrity trigger from table "provname"

DROP

--重新创建表customer

```
testdb=# CREATE TABLE customer (
```

```
testdb(# cno integer,
```



## 17 约束

```
testdb=# name char(30),
testdb=# tel char(20),
testdb=# prov char(2) REFERENCES provname
testdb=# ON UPDATE CASCADE --与主键同步修改
testdb=# ON DELETE SET NULL --当主键被删除时，置为空值
testdb=# );
```

NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)  
CREATE

--插入数据

```
testdb=# INSERT INTO customer
testdb=# VALUES(1001,'昆仑',82899746,'01');
INSERT 21746 1
```

```
testdb=# SELECT * FROM customer;
```

cno	name	tel	prov
1001	昆仑	82899746	01

(1 row)

--修改provname表中被参照的主键

```
testdb=# UPDATE provname
testdb=# SET code = '10'
testdb=# WHERE code = '01';
UPDATE 1
```

--表customer中的外键prov被同步修改

```
testdb=# SELECT * FROM customer;
```

cno	name	tel	prov
1001	昆仑	82899746	10

(1 row)

--删除被参照的主键“10”

```
testdb=# DELETE FROM provname
testdb=# WHERE code = '10';
DELETE 1
```

--表customer中的外键prov被自动置为空值

```
testdb=# SELECT * FROM customer;
```

cno	name	tel	prov
1001	昆仑	82899746	

(1 row)

在上面的例子中，定义外键的同时还定义了当被参照的主键被修改时的动作（ON

DELETE)，以及当被参照的主键被删除时的动作（ON DELETE）。CASCADE表明当主键被修改时，对外键作同样的修改；如果主键被删除，外键所在的行也被自动删除。SET NULL表明当主键被修改或主键所在的行被删除时，将外键设置为空值。除此之外，可用的动作还有以下两个：

NO ACTION——不允许修改或删除被参照的主键。这是默认的动作。

SET DEFAULT——当主键被修改或删除时，外键自动设置为默认值。

### 17.4.3 空值问题

如果主键只包含一个字段，则主键不可能为空值，这是由主键的性质所决定的。然而，有些表的主键由多个字段组成，在这种情况下，组成主键的部分字段可能为空值。必须妥善处理这种情况，否则同样不能保证参照的完整性。例如：

```
--重新创建测试用的表pkTEST
testdb=# DROP TABLE pkTEST;
NOTICE: DROP TABLE implicitly drops referential integrity trigger from table
"fktest"
DROP
testdb=# CREATE TABLE pkTEST(
testdb(# col1 INTEGER,
testdb(# col2 INTEGER,
testdb(# PRIMARY KEY(col1, col2)          --组合主键
testdb(# );
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'pktest_pkey'
for table 'pktest'
CREATE
--创建新的测试表
testdb=# CREATE TABLE fkTEST(
testdb(# col3 INTEGER,
testdb(# col4 INTEGER,
testdb(# FOREIGN KEY (col3, col4) REFERENCES pkTEST --组合外键
testdb(# );
NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
--插入主键
testdb=# INSERT INTO pkTEST
testdb=# VALUES (1,2);
INSERT 21776 1
--插入外键
testdb=# INSERT INTO fkTEST
```

## 17 约束

```
testdb=# VALUES(1,2);
INSERT 21777 1
--将外键中的一个字段设置为空值
testdb=# UPDATE fkTEST
testdb=# SET col4 = NULL
testdb=# WHERE col3 = 1 AND col4 = 2;
UPDATE 1
--更新操作被接收，但此时的参照完整性已经遭到破坏
testdb=# SELECT * FROM fkTEST;
 col3 | col4
-----+-----
    1 |
(1 row)
```

为了避免用户部分地将外键设置为空值，PostgreSQL提供了一个特殊的保留字——**MATCH FULL**。对于多字段组合而成的外键，这个保留字强迫PostgreSQL进行完全匹配操作。当外键中的某个字段为空值时，完全匹配操作将不会成功，这样就能够阻止用户将外键的部分字段设置为空。例如：

```
--重建测试用表fkTEST
testdb=# DROP TABLE fkTEST;
DROP
testdb=# CREATE TABLE fkTEST(
testdb=#   col3 INTEGER,
testdb=#   col4 INTEGER,
testdb=#   FOREIGN KEY (col3, col4) REFERENCES pkTEST
testdb=#   MATCH FULL          --强迫完全匹配外键
testdb=# );
NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
--插入数据
testdb=# INSERT INTO fkTEST
testdb=#   VALUES(1,2);
INSERT 21794 1
--将col4改为空值
testdb=# UPDATE fkTEST
testdb=#   SET col4 = NULL
testdb=#   WHERE col3 = 1 AND col4 = 2;
ERROR:  <unnamed> referential integrity violation - MATCH FULL doesn't allow
mixing of NULL and NON-NULL key values
--拒绝修改，因为外键匹配不完全
```

```
testdb=# SELECT * FROM fkTEST;
 col3 | col4
-----+-----
    1 |    2
(1 row)
```

#### 17.4.4 外键检查时机

默认情况下，外键的参照完整性检查在INSERT、DELETE和UPDATE命令结束时立即进行。对于由多条命令组成的大型事务，这种频繁的完整性检查会严重地影响运行效率。对于复杂的数据更新操作，要始终保持参照的完整性也比较困难。极端的例子是两个表互相有外键参照，按照默认的参照完整性检查规则，无法向这两个表插入记录。

解决问题的方法是使用外键的DEFERRABLE可选项以及SET CONSTRAINTS命令。这样，参照的完整性检查将在一个事务结束时进行，而不是在每个更新命令执行完时进行。例如：

```
--创建测试用表
testdb=# CREATE TABLE pkTEST_df
testdb=# (col integer PRIMARY KEY);
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'pktest_df_pkey'
for table 'pktest_df'
CREATE
testdb=# CREATE TABLE fkTEST_df
testdb=# (col integer REFERENCES pkTEST_df
testdb=# DEFERRABLE
testdb=# );
NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
--插入数据
testdb=# INSERT INTO fkTEST_df
testdb=# VALUES(11);
ERROR: <unnamed> referential integrity violation - key referenced from
fktest_df not found in pktest_df
--因为参照不完整，插入命令被拒绝
testdb=# COMMIT;
COMMIT
--开始一个新的事务
testdb=# BEGIN;
BEGIN
--将参照完整性检查推迟到事务提交时进行
```

## 17 约束

```
testdb=# SET CONSTRAINTS ALL DEFERRED;
SET CONSTRAINTS
testdb=# INSERT INTO fkTEST_df
testdb=# VALUES(11);
INSERT 21838 1
--不进行参照完整性检查
testdb=# INSERT INTO pkTEST_df
testdb=# VALUES(11);
INSERT 21839 1
testdb=# COMMIT;
COMMIT
--通过了完整性检查，事务提交成功
testdb=# SELECT * FROM fkTEST_df;
 col
-----
  11
(1 row)
--开始另外一个新的事务
testdb=# BEGIN;
BEGIN
testdb=# SET CONSTRAINTS ALL DEFERRED;
SET CONSTRAINTS
testdb=# INSERT INTO fkTEST_df
testdb=# VALUES(100);
INSERT 21841 1
testdb=# COMMIT;
ERROR:  <unnamed> referential integrity violation - key referenced from
fktest_df not found in pktest_df
--由于参照不完整，故事务提交失败
```

## 17.5 数据检验

除了前面介绍的约束外，在实际工作中经常需要进行更细致的数据检验。例如，对于人的性别来说，合法的值只有“男”和“女”；对于年龄来说，负数显然不符合常规的逻辑。对于这类数据的检验，必须使用PostgreSQL的CHECK（数据检验）机制。

在定义表的结构时，可以利用CHECK来规定字段的合法值。基本形式是：

CHECK (表达式)

其中表达式为一个与给定字段的值相关的逻辑表达式。只有在使表达式为真时，给字段赋的值才能被接收，否则，系统给出错误信息并终止当前操作。

例如：

--创建测试用表

```
testdb=# CREATE TABLE ckTEST (
testdb(#   name char(8) NOT NULL,           --姓名，非空
testdb(#   age integer CHECK(age>=0 AND age<=150), --年龄，在[0,150]范围内
testdb(#   gender char(2) CHECK(gender IN ('男','女')) --性别，只能是男或女
testdb(# );
```

CREATE

```
testdb=# \d ckTEST
```

Table "cktest"

Attribute	Type	Modifier
-----------	------	----------

-----+-----+-----

name	char(8)	not null
------	---------	----------

age	integer	
-----	---------	--

gender	char(2)	
--------	---------	--

Constraints: ((age >= 0) AND (age <= 150))

((gender = '男'::bpchar) OR (gender = '女'::bpchar))

```
testdb=# INSERT INTO ckTEST
```

```
testdb=# VALUES('朱琳',-1,'男');
```

ERROR: ExecAppend: rejected due to CHECK constraint cktest\_age

--数据插入不成功，因为年龄非法

--

```
testdb=# INSERT INTO ckTEST
```

```
testdb=# VALUES('朱琳',12,'x');
```

ERROR: ExecAppend: rejected due to CHECK constraint cktest\_gender

--数据插入不成功，因为性别非法

--

```
testdb=# INSERT INTO ckTEST
```

```
testdb=# VALUES('朱琳',12,'女');
```

INSERT 21855 1

--数据插入成功

```
testdb=# SELECT * FROM ckTEST;
```

name	age	gender
------	-----	--------

-----+-----+-----

朱琳	12	女
----	----	---

(1 row)

## 17 约束

在PostgreSQL中，不仅可以以字段为单位定义CHECK表达式，也可以以表为单位定义CHECK表达式。例如：

```
--创建测试用表
testdb=# CREATE TABLE ckTEST_tb
testdb=# (col1 integer,
testdb=# col2 char(10),
testdb=# col3 date,
testdb=# CHECK( col1>=10 AND col1<=100 AND          --表的CHECK条件
testdb=# (col2 = 'M' OR col2 = 'F'))
testdb=# );
CREATE
testdb=# INSERT INTO ckTEST_tb
testdb=# VALUES(1,'M',NULL);
ERROR:  ExecAppend: rejected due to CHECK constraint $1
--数据插入不成功，因为1不在col1的合法值范围之内
```

## 第18章

# 编程接口

---

PostgreSQL提供有丰富多彩的应用程序编程接口，利用这些接口可以开发出专业水准的应用程序。本章以一个简单的应用为例，介绍各种接口的基本编程方法，主要包括：

- 编程接口简介

- LIBPQ 接口

- LIBPGEASY 接口

- ECPG 接口

- LIBPQ++接口

- ODBC 与 JDBC 接口

- 脚本语言接口





## 18.1 编程接口简介

在前面所有章节的例子中，与 PostgreSQL 之间的数据库会话都是通过 psql 程序进行的。对于执行 SQL 命令或运行操纵数据库的脚本，psql 能够胜任大多数任务，但不适合于编写应用程序。

为了满足应用程序开发的需要，PostgreSQL 提供了多种编程接口，如表 18-1 所示。这些编程接口允许应用软件向 PostgreSQL 提交查询请求并接收返回的结果。

表18-1 编程接口

接口名称	适用的语言	执行方式	特点
LIBPQ	C	编译	本地接口
LIBPGEASY	C	编译	简化的 C 语言
ECPG	C	编译	ANSI 嵌入 SQL C 语言
LIBPQ++	C++	编译	面向对象的 C 语言
ODBC	ODBC	编译	开放数据库连接接口
JDBC	Java	编译/解释	跨平台
PERL	Perl	解释	文本处理
PGTCLSH	TCL/TK	解释	界面、窗口
PYTHON	Python	解释	面向对象
PHP	HTML	解释	动态 Web 页面

本章将用表 18-1 列出的各种接口实现同一个例子。这个例子相当简单，它首先提示用户输入美国某个州的代码，当用户输入了代码并按回车键后，例子程序输出与代码对应的州名称（后面用“州代码”来简称这个例子）。例如：

```
Enter a state code: AL
```

```
Alabama
```

这个例子需要一个 PostgreSQL 表，下面的命令创建这个表，并向表中插入一些用于测试的数据。

```
testdb=# CREATE TABLE statename (
testdb(#   code CHAR(2) PRIMARY KEY,           --州代码，主键
testdb(#   name CHAR(30)                       --州名称
testdb(# );
CREATE
testdb=# INSERT INTO statename VALUES ('AL', 'Alabama');
INSERT 18934 1
testdb=# INSERT INTO statename VALUES ('AK', 'Alaska');
INSERT 18934 1
```

## 18.2 LIBPQ 接口

LIBPQ 是 PostgreSQL 提供的 C 语言接口，它包含一套允许客户程序向 PostgreSQL 后端服务进程发送查询并且获得查询返回的库函数，同时也是其他几个 PostgreSQL 编程接口的基础，包括 LIBPQ++(C++)、LIBPGTCL(Tcl)、Perl5 和 ECPG 等。psql 就是使用 LIBPQ 实现的。

图18-1说明了LIBPQ在数据库连接中的作用。应用代码与用户终端通信并利用LIBPQ访问数据库，LIBPQ向数据库服务器提交查询请求，接收返回的结果。

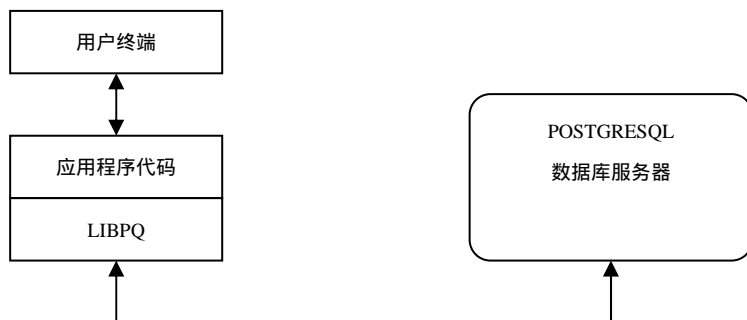


图18-1 LIBPQ数据流程

本节的例子程序用LIBPQ访问PostgreSQL，主要完成以下任务：

- (1) 建立一个数据库连接。
- (2) 提示并读入州代码。
- (3) 构建一个适当的SQL查询命令。
- (4) 向LIBPQ提交该SQL查询命令。
- (5) PostgreSQL执行查询命令。
- (6) 获取从LIBPQ返回的查询结果。
- (7) 将结果显示给用户。
- (8) 终止数据库连接。

所有与数据库的交互操作都是通过LIBPQ函数完成的，本例调用了以下LIBPQ函数：

PQconnectdb()——连接数据库。

PQexec()——向数据库提交查询。

PQntuples()——返回查询结果的行数。

PQgetvalue()——返回查询结果指定的行和列。

PQclear()——释放查询结果所用到的资源。

PQninish()——关闭数据库连接。

## 18 编程接口

这些函数大部分是通用的LIBPQ函数，有关它们的详细介绍请参考PostgreSQL的程序员手册。

```
/*
 * LIBPQ 例子程序
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"          /* LIBPQ 头文件 */
                                /* 所有使用 LIBPQ 库的程序必须包含这个头文件 */

int
main()
{
    char state_code[3];        /* 存放用户输入的州代码 */
    char query_string[256];    /* 存放构造的查询命令 */
    PGconn *conn;              /* 存放数据库连接信息 */
    PGresult *res;             /* 存放查询结果 */
    int i;

    conn = PQconnectdb("dbname=testdb"); /* 连接数据库 */

    if (PQstatus(conn) == CONNECTION_BAD) /* 数据库连接成功吗？ */
    {
        fprintf(stderr, "数据库连接失败！\n");
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit(1);
    }

    printf("Enter a state code: "); /* 提示用户输入州代码 */
    scanf("%2s", state_code);

    sprintf(query_string,          /* 构造查询命令字符串 */
            "SELECT name \
            FROM statename \
            WHERE code = '%s'", state_code);

    res = PQexec(conn, query_string); /* 执行查询命令 */

    if (PQresultStatus(res) != PGRES_TUPLES_OK) /* 查询成功吗？ */
```

```

    {
        fprintf(stderr, "查询失败!\n");
        PQclear(res);
        PQfinish(conn);
        exit(1);
    }

    for (i = 0; i < PQntuples(res); i++) /* 循环处理每一个结果行 */
        printf("%s\n", PQgetvalue(res, i, 0)); /* 显示返回的结果*/

    PQclear(res); /* 释放结果所占的空间*/

    PQfinish(conn); /* 关闭数据库连接*/

    return 0;
}

```

## 18.3 LIBPGEASY 接口

LIBPGEASY是一种简化的C语言接口，由一系列简化了的C语言函数组成，这些C语言函数封装了LIBPQ函数。在使用LIBPGEASY库的程序中，默认情况下不必进行错误检查，因为出错时LIBPGEASY会自动终止程序。利用on\_error\_continue()可以改变这一默认设置。

```

/*
 * LIBPGEASY例子程序
 */
#include <stdio.h>
#include <libpq-fe.h>
#include <libpgeasy.h> /* LIBPGEASY头文件*/
int
main()
{
    char state_code[3]; /* 存放用户输入的州代码*/
    char query_string[256]; /* 存放构造的SQL命令*/
    char state_name[31]; /* 存放返回的州名称*/

    connectdb("dbname=testdb"); /* 连接数据库*/

```

```

printf("Enter a state code: "); /* 提示用户输入州代码 */
scanf("%2s", state_code);

sprintf(query_string,          /* 构造SQL查询命令 */
        "SELECT name \
        FROM statename \
        WHERE code = '%s'", state_code);
doquery(query_string);        /* 执行查询命令*/

while (fetch(state_name) != END_OF_TUPLES) /* 循环处理每一个结果行 */
printf("%s\n", state_name);          /* 显示返回的结果 */

disconnectdb();                  /* 终止数据库连接*/
return 0;
}

```

## 18.4 ECPG 接口

ECPG称为嵌入式的C语言接口。这种接口允许将SQL命令直接嵌入到C语言程序中，因此比函数调用形式的编程接口更容易理解和使用。

在 C 语言程序中嵌入 SQL 命令的基本方式是：

```
EXEC SQL query
```

对于嵌入了 SQL 命令的 C 语言程序，在编译之前，ECPG 预编译器进行预处理。这个预编译器将嵌入的 SQL 命令转换为以变量为参数的函数调用。预处理的结果是标准的 C 语言程序，可以使用标准的 C 语言编译器进行编译。在进行链接时，编译结果程序会与一个包含所用函数的特殊库链接。这些函数从参数中提取信息，用通常的方法（LIBPQ）执行 SQL 查询命令，并且将结果放回到输出参数中。

```

/*
 * ECPG例子程序
 */
#include <stdio.h>

EXEC SQL INCLUDE sqlca;          /* ECPG头文件 */
EXEC SQL WHENEVER SQLERROR sqlprint;

int

```

```

main()
{
    EXEC SQL BEGIN DECLARE SECTION;

    char state_code[3];                /* 存放用户输入的州代码 */
    char *state_name = NULL;           /* 存放查询的返回结果 */
    char query_string[256];            /* 存放构造的SQL查询命令 */
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;        /* 连接数据库 */

    printf("Enter a state code: ");    /* 提示用户输入州代码 */
    scanf("%2s", state_code);

    sprintf(query_string,              /* 构造SQL查询命令字符串 */
        "SELECT name \
        FROM statename \
        WHERE code = '%s'", state_code);

    EXEC SQL PREPARE s_statename FROM :query_string;
    EXEC SQL DECLARE c_statename CURSOR FOR s_statename; /* 定义游标 */

    EXEC SQL OPEN c_statename;         /* 打开游标 */

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)                          /* 循环处理所有的返回数据行 */
    {
        EXEC SQL FETCH IN c_statename INTO :state_name;
        printf("%s\n", state_name);    /* 显示查询结果 */
        state_name = NULL;
    }

    free(state_name);                  /* 释放结果所占的空间 */

    EXEC SQL CLOSE c_statename;        /* 关闭游标 */

    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT;               /* 终止数据库连接 */
}

```

```

    return 0;
}

```

## 18.5 LIBPQ++接口

LIBPQ++是 PostgreSQL 的 C++语言接口，这个接口允许用面向对象的方法访问数据库。LIBPQ++实际上是一个 C++类的集合，这些类允许客户端程序与 PostgreSQL 后端服务器进行连接。连接的形式有两种：数据库类和大对象类。

数据库类用于操作数据库，可以向 PostgreSQL 后端服务器发送任何 SQL 查询命令并得到服务器的返回结果。大对象类用于操作数据库中的大对象。尽管一个大对象实例可以给 PostgreSQL 后端发送正常的查询，但通常只是用于那些不返回任何数据的简单查询。一个大对象应该看作是一个文件流。

```

/*
 * LIBPQ++例子程序
 */
#include <iostream.h>
#include <libpq++.h> // LIBPQ++头文件

int main()
{
    char state_code[3]; // 存放用户输入的州代码
    char query_string[256]; // 存放构造的SQL查询命令
    PgDatabase data("dbname=testdb"); // 连接数据库

    if ( data.ConnectionBad() ) // 数据库连接成功吗？
    {
        cerr << "数据库连接失败!" << endl
        << "错误为: " << data.ErrorMessage() << endl;
        exit(1);
    }

    cout << "Enter a state code: "; // 提示用户输入州代码
    cin.get(state_code, 3, '\n');

    sprintf(query_string, // 构造SQL查询命令

```

```

        "SELECT name \
        FROM statename \
        WHERE code = '%s'", state_code);

    if ( !data.ExecTuplesOk(query_string) )        // 执行查询
    {
        cerr << "SELECT查询失败!" << endl;
        exit(1);
    }

    for (int i=0; i < data.Tuples(); i++)        // 循环处理所有的结果行
        cout << data.GetValue(i,0) << endl;    // 显示返回结果

    return 0;
}

```

## 18.6 编译程序

到目前为止，所有的接口都是基于C和C++语言的，它们都需要某些包含文件和库文件来生成可执行程序。

典型情况下，接口的包含文件安装在/usr/local/pgsql/include目录中。应该使用编译标记-I，确保编译程序能够找到包含文件，例如，-I/usr/local/pgsql/include。

接口库文件安装在/usr/local/pgsql/lib目录中。应该使用编译标记-L，确保编译程序能够找到库文件，例如，-L/usr/local/pgsql/lib。

编译标记-l用来使编译程序连接指定的库文件。如果要连接libpq.a或libpq.so，则需要使用-lpq标记。

假定需要编译的程序名为myapp，各种接口的编译命令如下：

```

LIBPQ cc -I/usr/local/pgsql/include -o myapp myapp.c -L/usr/local/pgsql/lib
-lpq
LIBPGEASY cc -I/usr/local/pgsql/include -o myapp myapp.c
-L/usr/local/pgsql/lib -lpgeasy
ECPG ecpg myapp.pgc cc -I/usr/local/pgsql/include -o myapp myapp.c
-L/usr/local/pgsql/lib -lecpq
LIBPQ++ cc++ -I/usr/local/pgsql/include -o myapp myapp.cpp
-L/usr/local/pgsql/lib -lpq++

```

需要注意的是，各种接口有各自的库文件。ECPG接口要求编译前先执行ecpg预处理程序。



LIBPQ++要用不同的编译器。

## 18.7 程序变量

PostgreSQL是可用于网络环境的数据库，也就是说，数据库服务器和用户的应用可以在不同计算机上运行。因为字符串在所有计算机上的表示方法都相同，它们被用于用户程序与数据库服务器之间的通信。查询以字符串形式提交，结果也以字符串形式返回。即使所涉及的两台计算机大相径庭，用这种方式也能提供可靠的通信。

“州代码”例子程序在长度为30个字符的字段上进行SELECT查询。因为查询结果以字符串形式返回，所以返回值可以直接赋值给程序变量。与此相对，非字符字段，如整数或浮点数，就不能直接赋给整型或浮点型变量，而需要进行转换。

例如，当使用LIBPQ或LIBPQ++在一个整形字段上进行查询时，数据库中返回的不是一个整数而是一个字符串，应用程序必须将其转换为整数。一个整数返回的是'983'而不是983。要把这个值赋给一个整型变量，需要用到C语言的库函数atoi()，如var = atoi(colval)。

有一个例外是二进制指针，它返回某字段值的二进制表示。可以将二进制指针所得的结果直接赋给程序变量。然而，因为它们以二进制形式返回字段值，应用程序和数据库服务器必须运行在相同的计算机上，或至少要在有相同CPU指令的机器上。

LIBPGEASY用fetch()直接将查询结果返回给程序变量。这一函数将结果赋予字符串变量或二进制指针。

ECPG在将结果赋给程序变量之前，会自动将PostgreSQL返回的数据转换成适当的格式。本章后面涉及到的解释语言支持无类型数据，因此不存在这些问题。

## 18.8 ODBC 与 JDBC 接口

ODBC ( Open Database Connectivity , 开放数据库互连 ) 是一套应用编程接口，它提供一个与产品无关的前端应用和后端数据库服务器之间的接口，允许用户编写可以运行于不同厂商的数据库服务器上的应用程序。这个中间件层一般不直接用于编程，而是用于与其他应用程序进行通信。有关ODBC的详细使用方法见第21章。

JDBC 是 Java 1.1 及以后的核心 API，它为 SQL 兼容的数据库提供了一个标准的接口集合。PostgreSQL 提供了类型 4 JDBC 驱动程序，类型 4 表明该驱动程序是用纯 Java 编写的，并且与数据库之间使用数据库自己的网络协议通信。因此，JDBC 与平台无关的。一旦编译成功，该驱动可以用于任意平台。

下面的程序是“州代码”例子程序的Java版。

```
/*
 * Java例子程序
 */
import java.io.*;
import java.sql.*;

public class sample
{
    Connection conn;           // 存放数据库连接信息
    Statement stmt;            // 存放SQL命令
    String state_code;          // 存放用户输入的州代码

    public sample() throws ClassNotFoundException, FileNotFoundException,
    IOException, SQLException
    {
        Class.forName("org.postgresql.Driver");        // 装载数据库接口
        // 连接数据库
        conn = DriverManager.getConnection("jdbc:postgresql:testdb",
"testuser", "");
        stmt = conn.createStatement();

        System.out.print("Enter a state code: ");      // 提示用户输入州代码
        System.out.flush();
        BufferedReader r = new BufferedReader(new InputStreamReader
(System.in));
        state_code = r.readLine();

        ResultSet res = stmt.executeQuery(              // 执行查询
            "SELECT name " +
            "FROM statename " +
            "WHERE code = '" + state_code + "'");
        if (res != null)
            while(res.next())
            {
                String state_name = res.getString(1);
                System.out.println(state_name);
            }
        res.close();
        stmt.close();
    }
}
```

```

        conn.close();
    }
    public static void main(String args[])
    {
        try {
            sample test = new sample();
        } catch (Exception exc)
        {
            System.err.println("Exception caught.\n" + exc);
            exc.printStackTrace();
        }
    }
}

```

## 18.9 脚本语言接口

在此之前讨论的接口都是编译语言接口，编译语言要求将用户程序编译成CPU指令。剩下的接口采用脚本语言。脚本语言比编译语言执行起来慢得多，但却有许多编译语言不具备的优点：不需编译、更有效的命令、自动创建变量、变量可保存任意类型的数据等。

### 18.9.1 Perl

编写脚本和小应用程序时，Perl是一个不错的选择。它在处理文本文件和用CGI(Common Gateway Interface)创建动态Web网页方面功能比较强，因此受到了程序员的普遍欢迎。

下面的程序是“州代码”的Perl语言版本。

```

#!/usr/local/bin/perl
#
# Perl例子程序
#
use Pg;                                # 装入数据库例程

$conn = Pg::connectdb("dbname=testdb"); # 连接数据库
# 数据库连接成功吗？
die $conn->errorMessage unless PGRES_CONNECTION_OK eq $conn->status;

print "Enter a state code: ";          # 提示用户输入州代码

```

```

$state_code = <STDIN>;
chomp $state_code;

$result = $conn->exec(                                # 执行查询
    "SELECT name \
    FROM statename \
    WHERE code = '$state_code'");
# 查询执行成功吗？

die $conn->errorMessage unless PGRES_TUPLES_OK eq $result->resultStatus;

while (@row = $result->fetchrow) {                    # 循环处理所有的结果行rows returned
    print @row, "\n";                                  # 显示查询结果
}

```

### 18.9.2 TCL/TK(PGTCLSH/PGTKSH)

这种接口的特点是可访问其他工具包和应用程序。TK图形接口工具包就是其中一例，TCL利用它可实现图形应用。TK工具包目前比较流行，不少其他脚本语言也用它作为自己的图形接口库。

下面的程序是“州代码”的TCL版。

```

#!/usr/local/pgsql/bin/pgtclsh
#
# pgtclsh例子程序
#
set conn [pg_connect testdb]                ;# 连接数据库

puts -nonewline "Enter a state code: "      ;# 提示用户输入州代码
flush stdout
gets stdin state_code
;# 执行查询
set res [pg_exec $conn \
    "SELECT name \
    FROM statename \
    WHERE code = '$state_code'"]

set ntups [pg_result $res -numTuples]

for {set i 0} {$i < $ntups} {incr i} {      ;# 循环处理所有返回的数据行
    puts stdout [lindex [pg_result $res -getTuple $i] 0] ;# 显示查询结果
}

```

```
}
pg_disconnect $conn                ;# 终止数据库连接
```

## 18.9.3 Python

Python是一种面向对象的脚本语言，被誉为设计良好的语言，代码易于阅读和维护。下面的程序是“州代码”的Python版本。

```
#!/usr/local/bin/python
#
# Python例子程序
#
import sys

from pg import DB                # 装载数据库例程

conn = DB('testdb')            # 连接数据库

sys.stdout.write('Enter a state code: ')    # 提示用户输入州代码
state_code = sys.stdin.readline()
state_code = state_code[:-1]

for name in conn.query(          # 执行查询
    "SELECT name \
    FROM statename \
    WHERE code = '"+state_code+"'").getresult():
sys.stdout.write('%s\n' % name)    # 显示结果
```

## 18.9.4 PHP

PHP是一种服务器端嵌入式脚本语言，在PHP的帮助下，可以利用Web浏览器访问PostgreSQL。

利用PHP实现“州代码”例子需要两个Web页：一个用于录入数据，一个用于显示。用于录入数据的页面代码为：

```
<!--
-- PHP例子程序，数据录入页面
-->

<HTML>

<BODY>

<!-- 提示用户输入州代码 -->

<FORM ACTION="<? echo $SCRIPT_NAME ?>/pg/sample2.phtml?state_code"
```

```

method="POST">

    Client Number:

    <INPUT TYPE="text" name="state_code" value="<? echo $state_code ?>"
maxlength=2 size=2>

    <BR>

    <INPUT TYPE="submit" value="Continue">

</FORM>
</BODY>
</HTML>

用于执行查询和显示结果的页面代码是：
<!--
-- PHP例子程序，输出结果
-->
<HTML>
<BODY>
<?
$database = pg_Connect("", "", "", "", "testdb"); # 连接数据库

if (!$database) # 数据库连接成功吗？
{
    echo "数据库连接失败！ ";
    exit;
}

$result = pg_Exec($database,                # 执行查询
    "SELECT name " .
    "FROM statename " .
    "WHERE code = '$state_code'");

for ($i = 0; $i < pg_NumRows($result); $i++) # 循环处理所有的返回数据行
{
    echo pg_Result($result,$i,0);            # 显示结果
    echo "<BR>";
}
?>
</BODY>
</HTML>

```

### 18.9.5 脚本语言的安装

本章所介绍的接口语言都要求在语言中安装PostgreSQL数据库接口，通过重新编译语言或动态装载接口可以实现接口的安装。

下面列出的是各个接口的一些安装细节：

- (1) Perl通过USE指令将PostgreSQL接口例程装入Perl解释程序。
- (2) TCL/TK 提供三种接口选择：预建的TCL解释程序（pgtclsh）、预建的TCL/TK解释程序（pgtksh）以及可装载的库（libpgtcl）。
- (3) Python通过“from pg import DB”指令，将PostgreSQL接口例程装入Python解释程序。
- (4) PHP的情况比较复杂，既涉及到PHP本身，又涉及到Web服务器。关于PHP的安装请参考第24章。

本章介绍的所有接口都有各自的优点，有的接口简洁明了，有的接口运行效率高，还有一些接口在特定的环境中工作得比较好。利用这些接口，可以开发出专业水准的数据库应用程序。

## 第19章

# 自定义函数

---

为了扩充功能，PostgreSQL为用户提供了自定义函数的机制。本章介绍SQL函数和C语言函数的定义方法，主要包括：

- 自定义函数的一般过程

- SQL 函数

- C 语言函数





## 19.1 简介

前一章着重于用户机器上的客户端程序的编写以及与 PostgreSQL 之间的交互方法。服务器端函数，有时又被称为存储过程（Stored Procedure），是在数据库服务器上运行，而不是在用户的机器上运行。

服务器端函数有一些很好的性质。例如，如果一个函数被多个应用程序调用，那么将其嵌入在数据库服务器中会给函数的调用带来很大的方便。利用这种方法，每个应用程序不必再复制该函数，在需要该函数时只需简单地调用它就行了。与客户端函数不同，服务器端函数可以被 SQL 命令所调用。同样，安装在服务器内部的函数也很容易被修改。当某个函数被修改后，所有客户端应用程序就可以立即使用该函数的新版本。

第 8 章介绍了 PostgreSQL 预定义的服务器端函数，如 upper()、date\_part() 等。本章主要介绍如何创建用户自己的函数，此外还要介绍称为触发器的特殊服务器端函数，当表被修改时，系统会自动调用这些函数。

PostgreSQL 提供三种类型的函数：SQL 函数（用 SQL 命令实现的函数）、过程语言函数（用 PLTCL 或 PLSQL 等语言编写的函数）、编程语言函数（用类似于 C 的编译语言编写的函数）。

每种函数都可以以一个基本类型、一个复合类型或者二者的某种组合作为参数。另外，每种函数都可以返回一个基本类型或一个复合类型值。

本章主要介绍 SQL 函数和 C 语言函数，SQL 和 C 语言是 PostgreSQL 默认支持的语言。

## 19.2 SQL 函数

SQL 函数由多个 SQL 命令组成，函数的结果是最后一个 SQL 命令的执行结果。默认情况下，SQL 函数通常返回一个集合（用 setof 说明）。如果 SQL 函数的返回类型没有被说明为 setof，那么函数将返回最后一条 SQL 命令查询结果中的任意元素。

### 19.2.1 命令

创建 SQL 函数的命令是：

```
CREATE FUNCTION name ( [ ftype [, ...] ] )  
    RETURNS rtype
```

```
AS definition
LANGUAGE 'sql'
[ WITH ( attribute [, ...] ) ]
```

AS 后面的 definition 是 SQL 函数的函数体,它应该用空格字符分隔和用单引号括起来的 SQL 命令列表。如果函数体中的 SQL 命令需要使用单引号,那么必须用两个反斜杠的方法进行转义。

SQL 函数的参数在 SQL 命令中以 \$n 的方式引用:\$1 指第一个参数,\$2 指第二个参数,其余的参数依此类推。如果参数的类型为复合类型,那么可以用小数点表示法表示。例如,\$1.emp 即访问第一个参数中的 emp 字段。

删除 SQL 函数的命令是:

```
DROP FUNCTION name ( [ type [, ...] ] )
```

SQL 函数允许为一组 SQL 命令取一个名字,并存储于数据库中供用户访问。

SQL 函数的创建命令需要如下信息:

- (1) 函数名。函数名在所有数据库中必须是唯一的。
- (2) 函数的参数个数。
- (3) 每个参数的数据类型。
- (4) 函数返回的类型。
- (5) 函数的主体,由一系列 SQL 命令组成。

## 19.2.2 例子

### 1. 温度转换函数

下面例子所定义的函数将华氏温度转变成摄氏温度。其中,函数名为 ftoc;函数有一个参数,类型为 float,表示待转换的华氏温度;函数返回的类型为 float;函数的主体由一条 SQL 命令组成,命令为“SELECT (\$1 - 32.0) \* 5.0 / 9.0”。

```
--创建函数
testdb=# CREATE FUNCTION ftoc(float)                                --函数名
testdb=# RETURNS float                                             --函数的结果类型
testdb=# AS 'SELECT ($1 - 32.0) * 5.0 / 9.0;'                      --函数的主体
testdb=# LANGUAGE 'sql';                                           --函数所用的语言(SQL)
CREATE
--函数的使用
testdb=# SELECT ftoc(100);
          ftoc
-----
37.7777777777778
(1 row)
testdb=# SELECT ftoc(32);
```

## 19 自定义函数

```
ftoc
-----
0
(1 row)
```

ftoc()函数利用SELECT完成计算，这个函数不访问任何表。在函数主体中，SELECT命令的\$1被自动替换为函数的第一个参数。

执行“SELECT ftoc(100)”查询命令时，系统自动调用ftoc()，该函数用100替代\$1，然后进行计算。可以将这个命令理解为在一个SELECT中嵌套另外一个SELECT。外层的SELECT调用ftoc()，ftoc()又用它自己的SELECT完成计算。

函数中的常量是浮点数，可以进行浮点运算，否则执行整数运算。例如，命令“SELECT 1/4”的返回值为0，而“SELECT 1.0/4.0”的返回值为0.25。

### 2. 税款计算函数

下面的例子定义了一个用于计算税款的SQL函数。在函数体中，必须把查询结果的类型指定为NUMERIC(8,2)，因为计算结果要求两位小数。这个函数用紧连着的冒号而不是CAST来进行类型转换。

```
--创建函数
testdb=# CREATE FUNCTION tax(numeric)
testdb=# RETURNS numeric
testdb=# AS 'SELECT ($1 * 0.06::numeric(8,2))::numeric(8,2);'
testdb=# LANGUAGE 'sql';
CREATE
--测试函数
testdb=# SELECT tax(100);
 tax
-----
6.00
(1 row)
```

### 3. 条件计算

下面的例子定义了一种进行更复杂计算的函数。假定这个函数的功能是计算货物的运费，不同的货物重量范围运费费率也不相同，因此需要使用进行条件判断。

```
--定义函数
testdb=# CREATE FUNCTION shipping(numeric)
testdb=# RETURNS numeric
testdb=# AS 'SELECT CASE
testdb'# WHEN $1 < 2 THEN CAST(3.00 AS numeric(8,2))
testdb'# WHEN $1 >= 2 AND $1 < 4 THEN CAST(5.00 AS numeric(8,2))
testdb'# WHEN $1 >= 4 THEN CAST(6.00 AS numeric(8,2))
```

```

testdb=#      END;
testdb=#      LANGUAGE 'sql';
CREATE
--测试函数
testdb=# SELECT shipping(1);
shipping
-----
        3.00
(1 row)

testdb=# SELECT shipping(2);
shipping
-----
        5.00
(1 row)

testdb=# SELECT shipping(5);
shipping
-----
        6.00
(1 row)

```

#### 4. 在表查询中使用函数

服务器端函数主要用于表数据的SQL查询，下面的例子说明了这种用法。先创建一个零件表，接着插入3行数据，最后，利用SELECT命令查询出零件表中的字段，并进一步计算税款和成本加税款之和。

```

--创建测试用表
testdb=# CREATE TABLE part (
testdb(#   part_id   INTEGER,
testdb(#   name      CHAR(30),
testdb(#   cost      NUMERIC(8,2),
testdb(#   weight    FLOAT
testdb(# );
CREATE
--插入测试数据
testdb=# INSERT INTO part VALUES (637, 'cable', 14.29, 5);
INSERT 21918 1
testdb=# INSERT INTO part VALUES (638, 'sticker', 0.84, 1);
INSERT 21919 1

```

## 19 自定义函数

```
testdb=# INSERT INTO part VALUES (639, 'bulb', 3.68, 3);
INSERT 21920 1
testdb=# SELECT part_id,
testdb=#      name,
testdb=#      cost,
testdb=#      tax(cost),                                --调用自定义的tax函数
testdb=#      cost + tax(cost) AS total                 --调用自定义的tax函数
testdb=# FROM part
testdb=# ORDER BY part_id;
 part_id |          name          | cost | tax | total
-----+-----+-----+-----+-----
    637 | cable                  | 14.29 | 0.86 | 15.15
    638 | sticker                |  0.84 | 0.05 |  0.89
    639 | bulb                   |  3.68 | 0.22 |  3.90
(3 rows)
```

### 5. 访问表的 SQL 函数

服务器端SQL函数也可以访问数据库中的表。下面的例子定义一个函数，其功能是统计学生基本情况表中男学生或女学生的人数。

```
--定义函数
testdb=# CREATE FUNCTION sexcount(text)
testdb=# RETURNS INTEGER
testdb=# AS 'SELECT count(*)::integer'                --访问表s
testdb'# FROM s
testdb'# WHERE sex = $1;'
testdb=# LANGUAGE 'sql';
CREATE
--计算男学生的人数，并检验结果
testdb=# SELECT sexcount('男');
sexcount
-----
      8
(1 row)
testdb=# SELECT * FROM s
testdb=# WHERE sex = '男';
 sno |  sname  | sex | age | dept
-----+-----+-----+-----+-----
9803101 | 李小波  | 男  |  20 | 计算机
9803102 | 王前    | 男  |  20 | 计算机
```

```

9803105 | 元亮      | 男 | 23 | 计算机
9803115 | 徐营串    | 男 | 23 | 计算机
9810101 | 期荣强    | 男 | 19 | 机械
9810102 | 钟恢复    | 男 | 21 | 机械
9810105 | 李源      | 男 | 22 | 机械
9803106 | 龚安      | 男 | 22 | 计算机系
(8 rows)

```

--计算女学生的人数，并检验结果

```

testdb=# SELECT sexcount('女');
sexcount
-----
          5
(1 row)

```

```
testdb=# SELECT * FROM s
```

```
testdb=# WHERE sex = '女';
```

```

sno  | sname  | sex | age | dept
-----+-----+-----+-----+-----
9803103 | 仲林    | 女 | 21 | 计算机
9803104 | 黄启    | 女 | 21 | 计算机
9803114 | 刘淇    | 女 | 21 | 计算机
9810103 | 王吉林  | 女 | 22 | 机械
9810104 | 苏强    | 女 | 21 | 机械
(5 rows)

```

## 6. 多 SQL 命令函数体

前面所有例子定义的SQL函数体都只包含一条SQL命令，实际上，PostgreSQL允许在函数体中使用多条SQL命令。SQL函数体中的SQL语句可以是SELECT、INSERT、UPDATE和DELETE，不过，最后的命令必须是一条结果类型与函数返回类型一致的SELECT语句。

例如：

--创建函数

```
testdb=# CREATE FUNCTION clean_table ()
```

```
testdb=# RETURNS integer
```

```
testdb=# AS '
```

```
testdb'# DELETE FROM uitest; --第一条SQL命令
```

```
testdb'# SELECT 1 AS 表uitest的数据已被删除; --第二条SQL命令
```

```
testdb'# '
```

```
testdb=# LANGUAGE 'sql';
```

CREATE

--测试函数

## 19 自定义函数

```
testdb=# SELECT clean_table();
clean_table
-----
1
(1 row)
```

### 19.3 C 语言函数

顾名思义，C 语言函数就是利用 C 语言编写的 PostgreSQL 服务器端函数。用 C 语言编写的函数可以被编译成可动态装载的对象，该对象用于实现用户自定义函数。当用户自定义的函数第一次被后端服务器调用时，动态装载器将函数的目标代码装入内存，然后将这个函数与正在运行的 PostgreSQL 可执行文件链接起来。

由于 C 语言是一种编译型的语言，因此创建 C 语言函数的过程较复杂。基本步骤如下：

- (1) 编写 C 语言函数代码。
- (2) 利用编译器将 C 语言函数代码编译成目标文件。
- (3) 利用 CREATE FUNCTION 命令注册创建的函数。

利用 C 语言还可以创建新的运算符、数据类型以及聚集函数，创建步骤与函数基本相似，唯一不同的是注册命令。注册运算符的命令是 CREATE OPERATOR，注册数据类型的命令是 CREATE TYPE，注册聚集函数的命令是 CREATE AGGREGATE。

本章以一个例子为线索，介绍创建 C 语言函数的过程。这个例子定义一个将摄氏温度转换为华氏温度的函数。

#### 19.3.1 编写 C 语言源代码

```
#include "postgres.h"

double *ctof(double *deg)
{
    double *ret = palloc(sizeof(double));
    *ret = (*deg * 9.0 / 5.0) + 32.0;
    return ret;
}
```

需要注意的是，尽管可以自由地利用标准的 C 语言语句编写 C 语言函数，但同时也应该遵循 PostgreSQL 的一些特殊约定。当申请内存时，最好使用 PostgreSQL 的函数 palloc 和 pfree，尽量避免使用标准的 C 语言库函数 malloc 和 free。这是因为用 palloc 申请的内存事务结束时会自动释放，从而避免了内存泄露。

大多数的 PostgreSQL 内部类型定义在 postgres.h 文件中，所以在定义 C 语言函数的程

序中应该包含这个文件。postgres.h 会自动包含 elog.h 和 pallocc.h 文件。

### 19.3.2 编译 C 语言源代码

编译的目的是将 C 语言源代码文件编译成由 CPU 指令构成的目标文件，并且这个目标文件必须是一个特殊的能动态链接到 PostgreSQL 服务器的目标文件。很多操作系统要求用特殊的编译可选项来创建能够动态链接的目标文件。找到所要求的标记的最好办法是在 pgsql/src/test/regress 目录下，输入 “make clean” 和 “make regress.so”。此命令显示生成可动态链接的目标文件 regress.so 的编译命令。编译标记 *-I* 允许搜索 include 文件，其他标记用于生成可动态链接的目标文件。

### 19.3.3 注册新函数

前面的步骤已经创建了一个可动态链接的目标文件，接下来必须将目标文件定义的新函数注册到 PostgreSQL 中去。

注册 C 语言函数的命令仍然是 “CREATE FUNCTION”，语法格式如下：

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS obj_file , link_symbol
    LANGUAGE 'C'
    [ WITH ( attribute [, ...] ) ]
```

本命令用两种方法之一将 PostgreSQL 函数与 C 语言函数链接起来。如果 PostgreSQL 函数与 C 语言函数同名，使用语句的第一种形式，即 AS 子句中的字符串参数中包含已经编译好的可动态装载对象的完整路径名。如果 C 语言函数的名字与 PostgreSQL 函数的名字不同，则使用第二种形式。在第二种形式中，AS 子句接受两个字符串参数，第一个参数是可动态装载目标文件的完整路径，第二个参数是动态装载器搜索的符号链接。这个符号链接是 C 语言源代码中的函数名称。

在第一次使用之后，一个动态装载的用户函数仍然停留在内存中，因而对该函数的更进一步的调用只是简单的符号表查找。

指定目标文件的参数（AS 子句中的字符串）应该是该函数目标文件的完整路径，并且必须用引号括起来。如果在 AS 子句中使用了符号链接，该符号链接也应该用单引号括起来，并且必须是 C 语言源代码中的函数名称。

假定前一小节创建的可动态链接目标文件为 “/users/pgman/sample/ctof.so”，注册该函数的命令是：

```
testdb=# CREATE FUNCTION ctof(float)
testdb=# RETURNS float
testdb=# AS '/users/pgman/sample/ctof.so'
testdb=# LANGUAGE 'C';
CREATE
```

函数 ctof 有一个浮点型的参数并返回浮点型的结果，ctof 函数所使用 SQL 的浮点类型



## 19 自定义函数

---

与 C 语言的双精度类型是一致的。

一个目标文件可以包含多个函数，但必须分别用“CREATE FUNCTION”命令注册每一个函数。

一旦一个函数被成功地注册，该函数就可以像 PostgreSQL 的内部函数一样被调用。例如：

```
testdb=# SELECT ctof(20);
 ctof
-----
   68
(1 row)
```

## 第20章

# PL/pgSQL 语言

---

PL/pgSQL语言是扩展PostgreSQL功能的最常用语言，不仅可以用于创建普通函数，也可以用来创建触发器函数。本章介绍PL/pgSQL语言的基本编程方法，主要包括：

- 语言简介和安装

- 基本语言元素

- 函数的定义方法

- 触发器



## 20.1 语言简介

PL/pgSQL 语言是 PostgreSQL 数据库系统的一个可装载的过程语言，它是一种真正的编程语言，提供了变量、条件判断以及循环等基本语言元素。它的主要特点包括：

- (1) 可动态装载。
- (2) 可用于创建函数和触发器过程。
- (3) 为 SQL 语言增加了控制结构。
- (4) 可以执行复杂的计算。
- (5) 继承所有用户自定义的类型、函数和运算符。
- (6) 可以定义为受服务器信任的语言。
- (7) 容易使用。

利用 PL/pgSQL 语言定义的函数在第一次被后端服务器调用时，PL/pgSQL 的调用控制器分析函数源代码，并生成二进制形式的指令树。所生成的字节代码在调用控制器中是用函数的对象标识来标记的，这样就保证了：如果使用 DROP/CREATE 命令修改了函数，不必建立一个新的数据库连接就能使修改生效。

除了用于用户自定义类型的输入/输出转换和计算函数以外，任何可以在 C 语言函数中定义的对象都可以在 PL/pgSQL 中使用。因此，可以创建复杂的条件计算函数，并随后将其用于定义运算符。

在 Linux 中，默认情况下，PostgreSQL 并没有安装 PL/pgSQL 语言，因此在使用这种语言之前必须手工安装。

首先以某用户的身份登录 Linux，这个用户应该是 PostgreSQL 的合法用户。然后设置安装语言所需的环境变量 PGLIB。在 Linux 中，如果是利用 RPM 包安装 PostgreSQL 的，则 PGLIB 的值应该是“/usr/lib/pgsql”。假定当前用户所使用的 shell 为 bash，则定义环境变量的命令是：

```
export PGLIB=/usr/lib/pgsql
```

接下来，利用下面的命令安装 PL/pgSQL 语言：

```
createlang plpgsql testdb
```

其中，testdb 是使用 plpgsql 语言的数据库，读者应该根据自己的情况进行修改。

删除语言的命令是：

```
droplang plpgsql testdb
```

上面的命令都必须在 Linux 系统提示符下运行，而不是在 psql 中运行。

为了检验 PL/pgSQL 语言是否安装成功，可以在 psql 中按照下面的方法进行验证。

-- 创建一个 PL/pgSQL 函数

```
testdb=# CREATE FUNCTION add_one (int4)
```

```
testdb=# RETURNS int4 AS '  
testdb'# BEGIN  
testdb'# RETURN $1 + 1;  
testdb'# END;  
testdb'# '  
testdb=# LANGUAGE 'plpgsql';  
CREATE  
--测试函数  
testdb=# SELECT add_one(10);  
add_one  
-----  
11  
(1 row)
```

如果上面命令的执行情况正常，表明 PL/pgSQL 已被成功地安装。

## 20.2 语言元素

### 20.2.1 程序结构

PL/pgSQL 是一种面向块的语言，定义一个块的基本格式是：

```
[<<label>>]  
--定义变量  
[DECLARE  
    declarations]  
BEGIN          --程序开始  
    statements --语句  
END;           --程序结束
```

PL/pgSQL 语言对大小写字母不敏感，所有关键字和标识符都可以任意使用大小写字母。

在 PL/pgSQL 中有两种形式的注释。一种形式的注释是 SQL 风格的注释：以 “--” 开始直到该行的行尾都是注释信息。另一种形式的注释是传统 C 语言风格的注释块：以 “/\*” 开始，直到 “\*/” 结束。

### 20.2.2 变量的定义

DECLARE 语句用于开始一个定义段，在定义段中可以定义语句块中需要使用的变量。一个语句块可以包含多个子语句块，每一个子语句块也可以利用 DECLARE 语句定义属于自己的变量，子语句块也可以屏蔽该语句块外面的变量。在定义段中定义的变量，在每次进入该语句块时都进行初始化，而不是在每次函数调用时初始化一次。

在一个语句块或者它的子语句块中用到的所有变量、行和记录，都必须在一个语句块的定义段中进行定义，唯一的例外是 FOR 循环的整型循环变量。PL/pgSQL 函数的参数用常用的标识 \$n 定义。

定义变量的形式有以下几种：

(1) `name [ CONSTANT ] >type> [ NOT NULL ] [ DEFAULT | := value ];`

这种形式的语句定义了一个基本数据类型变量。如果变量被定义为 CONSTANT，则变量值不能修改；如果使用 NOT NULL 可选项，则不允许给该变量赋空值。由于默认情况下所有变量的默认值都是 NULL，因此定义为 NOT NULL 的变量都必须有非空的默认值。

默认值是在函数调用时计算的。如果为一个 datetime 类型的变量定义一个默认值“new”，那么每次被初始化时该变量的值实际是函数调用时的时间，而不是函数被编译时的时间。

(2) `name class%ROWTYPE;`

这种形式的语句定义了一个带有指定表结构的行，这里的表必须是一个在数据库中存在的表或视图，行的字段是通过点的形式来访问的。函数的参数可以是复合类型（表的完整行）。这时，对应的标识 \$n 是一个行类型（rowtype），但是它必须用下面描述的 ALIAS 命令取个别名。在一个数据行中，只有用户字段可以被访问，OID 或其他系统字段都不可访问。

行类型（rowtype）字段继承 char() 等数据类型的字段尺寸或精度。

(3) `name RECORD;`

记录类型类似于行类型，不同之处在于记录类型没有预定义的结构。它们主要用于 SELECT 和 FOR 循环，目的是为了保存 SELECT 操作中的一个数据行。当没有实际的行存在于其中时，试图访问一条记录或为某个变量赋值都将导致一个运行错误。

(4) `name ALIAS FOR $n;`

为了让代码更具可读性，可以利用这种形式的语句为函数的位置参数定义一个别名。

当将复合类型作为参数传递给一个函数时要求使用别名，SQL 函数中的点表示法（如 \$1.salary）在 PL/pgSQL 中不允许使用。

(5) `RENAME oldname TO newname;`

这种形式的语句主要用于改变变量、记录或者行的名称。该语句当 NEW 或 OLD 在触发器中被另一个名字引用时很有用。

### 20.2.3 类型与表达式

变量的数据类型可以是数据库现有的任意基本类型。在定义段中，type 的格式为：

Postgres-基本类型

```
variable%TYPE
```

```
class.field%TYPE
```

其中，variable 是变量名，class 是表或视图的名称，field 是字段的名称。

在 PL/pgSQL 语言中，所有表达式都由后端服务器的执行器进行处理，所以对于 PL/pgSQL 分析器而言，除了 NULL 关键字以外，它是不可能识别真正常量的值的。所有表达式都是通过 SPI 管理器在内部执行“SELECT expression”命令来计算的，表达式中出现的变量都被参数的实际值所代替。所有在 PL/pgSQL 函数中用到的表达式都只预处理和存储一次。

PostgreSQL 的主分析器所进行的类型检查对常量转换来说有一些副作用。例如，下面的两个函数有一些区别：

```
CREATE FUNCTION logfunc1 (text) RETURNS datetime AS '
    DECLARE
        logtxt ALIAS FOR $1;
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
        RETURN 'now';
    END;
' LANGUAGE 'plpgsql';
```

和

```
CREATE FUNCTION logfunc2 (text) RETURNS datetime AS '
    DECLARE
        logtxt ALIAS FOR $1;
        curtime datetime;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
        RETURN curtime;
    END;
' LANGUAGE 'plpgsql';
```

在 logfunc1() 中，PostgreSQL 的主分析器在准备 INSERT 的规划时认为，字符串“now”应该解释为 datetime，因为 logtable 的目标字段也是这种类型。因此，它将从中生成一个常量，并且在所有后端服务器生存期内，只要调用 logfunc1() 就使用该常量。很显然，这不是程序员希望的。

在 logfunc2() 中，PostgreSQL 的主分析器并不知道“now”应该转换的类型，因此它返回一个包含字符串“now”的 text 数据类型。在给局部变量 curtime 赋值时，PL/pgSQL 解释器通过调用 text\_out() 和 datetime\_in() 将这个字符串转换成 datetime 类型的变量。

PostgreSQL 主分析器的类型检查是在 PL/pgSQL 接近完成时实现的。在版本 6.3 和版本 6.4 之间有所不同并且影响所有使用 SPI 管理器规划特性的函数。使用上面提到的局部

变量的方法是目前能让 PL/pgSQL 对数值进行正确解释的唯一方法。

如果在表达式或语句中用到记录 (record) 字段, 字段的数据类型在同一个表达式的不同调用中不应该改变。

#### 20.2.4 语句

任何 PL/pgSQL 分析器不能理解的内容, 都会被放入查询命令中并发送给数据库引擎执行, 生成的查询应该不返回任何数据。

##### 1. 赋值语句

给一个变量、行或记录赋值的语句格式是:

```
identifier := expression;
```

如果表达式的结果数据类型和变量数据类型不一致, 或者变量具有已知的尺寸或精度, 结果值将隐含地被 PL/pgSQL 字节码解释器用结果类型的输出函数和变量类型的输入函数进行转换。需要注意的是, 这种转换可能导致运行时错误。

下面方法可以将一个完整的查询结果存放在一条记录或者行中:

```
SELECT expressions INTO target FROM ...;
```

其中, target 可以是一条记录、一行变量或一个由逗号分隔的变量列表和记录字段。

如果将一行或者一个变量列表当作目标, 查询的数值必须与目标列的结构完全一样, 否则会导致运行时错误。

一个名为 FOUND 的特殊逻辑型变量可以在 SELECT INTO 之后立刻用于检查赋值是否成功。例如:

```
SELECT * INTO myrec FROM EMP WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

如果查询命令返回多个数据行, 只有第一行被放到目标列表中, 其他所有的行都会被丢弃。

##### 2. 条件判断语句

```
IF expression THEN
    statements1
[ELSE
    statements2]
END IF;
```

其中, 表达式 expression 必须返回一个能够转换为逻辑值的数值。当 expression 为真时, 执行 statements1 语句系列, 否则执行 statements2 语句系列。statements2 也可以省略, 这样当 expression 不为真时, 不执行任何语句。

### 3. 循环语句

PL/pgSQL 提供三种形式的循环语句。

(1) LOOP 循环，基本格式是：

```
[<<label>>]
LOOP
    statements
END LOOP;
```

这是一个无条件循环，它的循环过程必须用一条 EXIT 语句显式地终止。可选的标记 label 可以被 EXIT 用于说明应该结束哪一层循环。

(2) WHILE 循环，基本格式是：

```
[<<label>>]
WHILE expression LOOP
    statements
END LOOP;
```

在 expression 为真时，重复执行 statements 语句序列，直到 expression 为假时终止。

(3) FOR 循环，基本格式是：

```
[<<label>>]
FOR name IN [ REVERSE ] expression .. expression LOOP
    statements
END LOOP;
```

这是一种在某个整数范围内执行的循环，变量 name 由系统自动以整数类型创建，并且只对循环内部的语句可见。两个表达式分别给出循环范围区间，它们只是在进入循环后才被计算出数值，循环步进值总是 1。

下面是另外一种形式的 FOR 循环，主要用于对表数据的处理。

```
[<<label>>]
FOR record | row IN select_clause LOOP
    statements
END LOOP;
```

select 语句的结果赋给记录或行，并且 statements 语句序列对每一条记录或行都运行一次。如果用 EXIT 语句退出循环，最后赋值的行在循环之后仍然可以访问。

所有这些循环语句都可以用 EXIT 语句强行结束，该语句的格式是：

```
EXIT [ label ] [ WHEN expression ];
```

如果没有定义标记 label，最内层的循环将被终止，但 END LOOP 后面的语句仍将继续执行。如果定义了标记 label（该标记必须是当前或者上层嵌套的循环块），那么命名的循环或语句块将被终止并且控制落到循环/语句块对应 END 的后面一条语句。

### 4. 其他语句

(1) PERFORM。所有 PostgreSQL 函数都返回一个值，因此，调用函数的方法是执行一条 SELECT 命令，或为一个变量赋值。但是，有些时候用户只需要函数执行某些动作，



而对函数结果不感兴趣。在这种情况下可以使用下面的语句：

```
PERFORM query
```

这个语句在 SPI 管理器上执行一个“SELECT query”，然后丢弃结果。

(2) RETURN expression。这条语句使函数结束并且将表达式 expression 的值返回给上层执行器。如果控制到达一个函数体的最顶层但没有遇到 RETURN 语句，将产生一个运行时错误。表达式的结果将被自动转换为函数的返回类型。

(3) RAISE。RAISE 语句可以输出一条信息到 PostgreSQL 的 elog 日志中，它的语法格式是：

```
RAISE level 'format' [, identifier [...]];
```

其中，format 描述信息的格式模板；level 为日志级别，可用的级别包括 DEBUG（安静地停止数据库）、NOTICE（向数据库书写日志并向前端客户应用程序发送信息）、EXCEPTION（向数据库书写日志并且退出事务）。

例如：

```
w friends%ROWTYPE;  
for w in SELECT * FROM friend loop  
    raise notice 'friend name: %',w.name;  
end loop;
```

### 20.2.5 例外处理

PostgreSQL 没有一个很好的例外处理模块。当分析器、规划优化器或者执行器认为一个语句不能继续处理时，整个事务都会被终止，并且系统跳回主循环等待客户的下一个查询。

可以用类似于 Windows 的“挂钩”机制来捕捉错误情况的发生，但到目前版本为止，PostgreSQL 仍然没有能力告诉用户什么原因导致了退出，并且此时的数据库后端可能处在一种不连贯的状态，所以退回到上层执行器或执行更多的命令可能摧毁整个数据库。此时事务退出的信息可能已经发送给了客户端应用，所以继续操作没有任何意义。

因此，PL/pgSQL 在函数或触发器操作时遇到退出的唯一一项操作是在 DEBUG 级别运行时输出一些附加的日志信息，报告哪个函数在哪里（行号和语句类型）出了错。

## 20.3 PL/pgSQL 函数

本节通过一些例子来介绍利用 PL/pgSQL 开发用户自定义函数的基本方法。

### 1. 字符串拼接函数

下面的例子定义了一个将两个字符串拼接为一个字符串的函数。

```
--创建函数
testdb=# CREATE FUNCTION concat_text (text, text)
testdb=# RETURNS text AS '
testdb'# BEGIN
testdb'# RETURN $1 || $2;
testdb'# END;
testdb'# LANGUAGE 'plpgsql';
CREATE
--测试函数
testdb=# SELECT concat_text('Hello','world!');
concat_text
-----
Hello,world!
(1 row)
```

## 2. 表查询函数

下面的例子定义了一个表查询函数，该函数以姓名为入口参数，通过查询 friend 表，函数返回指定人所在的省市。

```
--定义函数
testdb=# CREATE FUNCTION getprovname(char(8))
testdb=# RETURNS char(6)
testdb=# AS 'DECLARE ret char(6);
testdb'# BEGIN
testdb'# SELECT CAST(prov AS char(6))
testdb'# INTO ret
testdb'# FROM friend
testdb'# WHERE name = $1;
testdb'# IF NOT FOUND THEN
testdb'# RETURN ''不存在'';
testdb'# ELSE
testdb'# IF ret IS NULL THEN
testdb'# RETURN ''未知'';
testdb'# ELSE
testdb'# RETURN ret;
testdb'# END IF;
testdb'# END IF;
testdb'# END;'
testdb=# LANGUAGE 'plpgsql';
CREATE
```

--测试函数

```
testdb=# select getprovname('李江');
```

```
getprovname
```

```
-----
```

```
广东
```

```
(1 row)
```

```
testdb=# SELECT getprovname('怪物');
```

```
getprovname
```

```
-----
```

```
未知
```

```
(1 row)
```

```
testdb=# SELECT getprovname('外星人');
```

```
getprovname
```

```
-----
```

```
不存在
```

```
(1 row)
```

在上面的函数中，用到了“SELECT.....INTO.....”命令，它的功能是将查询的结果存放在指定的内存变量中。函数还使用了条件判断语句，如果 friend 表中不存在指定的人，函数返回“不存在”；如果指定的人存在但省市字段为空，函数返回“未知”。

### 3. 复杂的字符串操作函数

下面的例子定义了一个更为复杂的PL/pgSQL函数。这个函数以一个字符串为入口参数，函数将字符串中的小写字母改为大写字母，并在各字符之间插入一个空格。这是一个典型的使用了WHILE循环的函数。

--创建函数

```
testdb=# CREATE FUNCTION spread(text)
```

```
testdb=# RETURNS text
```

```
testdb=# AS 'DECLARE
```

```
testdb'# str text;
```

```
testdb'# ret text;
```

```
testdb'# i integer;
```

```
testdb'# len integer;
```

```
testdb'# BEGIN
```

```
testdb'# str := upper($1);
```

```
testdb'# ret := ''';
```

```
testdb'# i := 1;
```

```
testdb'# len := length(str);
```

```
testdb'# WHILE i <= len LOOP
```

```
testdb'# ret := ret || substr(str, i, 1) || ' ';
```

```

testdb'#          i := i + 1;
testdb'#          END LOOP;
testdb'#          RETURN ret;
testdb'#        END; '
testdb-#  LANGUAGE 'plpgsql';
CREATE
--测试函数
testdb=# SELECT spread('Hello,world!');
          spread
-----
 H E L L O , W O R L D !
(1 row)

```

#### 4. 数据更新函数

用户自定义函数不仅可以进行各种计算,也可以进行数据库操作,如数据的插入、更新、删除、校验等。下面的例子定义了一个数据更新函数。

```

--创建函数
testdb=# CREATE FUNCTION change_prov(char(8), char(6))
testdb-# RETURNS text
testdb-# AS '
testdb-# DECLARE
testdb-#     f_name ALIAS FOR $1;
testdb-#     f_prov ALIAS FOR $2;
testdb-#     f_rec RECORD;
testdb-#
testdb-# BEGIN
testdb-#     IF length(f_name) = 0 THEN
testdb-#         RETURN '未指定姓名';
testdb-#     ELSE
testdb-#         IF length(f_prov) != 0 THEN
testdb-#             SELECT *
testdb-#             INTO f_rec
testdb-#             FROM friend
testdb-#             WHERE name = f_name;
testdb-#             IF NOT FOUND THEN
testdb-#                 INSERT INTO friend
testdb-#                 (name,prov)
testdb-#                 VALUES (f_name, f_prov);
testdb-#             RETURN '数据插入成功';

```

```

testdb'# _____ ELSE
testdb'# _____ UPDATE friend
testdb'# _____ SET prov = f_prov
testdb'# _____ WHERE name = f_name;
testdb'# _____ RETURN ''省市名更新成功'';
testdb'# _____ END IF;
testdb'# _____ ELSE
testdb'# _____ SELECT *
testdb'# _____ INTO f_rec
testdb'# _____ FROM friend
testdb'# _____ WHERE name = f_name;
testdb'# _____ IF FOUND THEN
testdb'# _____ DELETE FROM friend
testdb'# _____ WHERE name = f_name;
testdb'# _____ RETURN ''数据删除成功'';
testdb'# _____ ELSE
testdb'# _____ RETURN ''没有找到指定的人'';
testdb'# _____ END IF;
testdb'# _____ END IF;
testdb'# _____ END IF;
testdb'# _____ END; '
testdb-# _____ LANGUAGE 'plpgsql';
CREATE

```

上面命令定义了PL/pgSQL函数change\_prov，这个函数为维护friend表的数据提供了一种服务器端接口。调用函数时需要提供姓名和省市名称。如果表中不存在指定姓名的人，再插入一条记录；如果存在指定姓名的人，则更新与之对应的省市名称；如果调用函数时第二个参数为空字符串，则删除指定姓名的人的记录。无论是何种操作，函数都会返回该操作是否成功的信息。

以下是测试这个函数功能的例子。

```

--
--测试函数的数据更新功能
--
testdb=# SELECT * FROM friend
testdb-# WHERE name = '王芳';
   name  | city | prov | age
-----+-----+-----+-----
 王芳    |      |      |
(1 row)
testdb=# SELECT change_prov('王芳','北京');

```

```

change_prov
-----

省市名更新成功
(1 row)
testdb=# SELECT * FROM friend
testdb=# WHERE name = '王芳';
   name  | city | prov | age
-----+-----+-----+-----
 王芳    |      | 北京 |
(1 row)
--
--测试函数的数据插入功能
--
testdb=# SELECT * FROM friend
testdb=# WHERE name = '王平';
   name  | city | prov | age
-----+-----+-----+-----
(0 rows)
testdb=# SELECT change_prov('王平','上海');
change_prov
-----

数据插入成功
(1 row)
testdb=# SELECT * FROM friend
testdb=# WHERE name = '王平';
   name  | city | prov | age
-----+-----+-----+-----
 王平    |      | 上海 |
(1 row)
--
--测试函数的删除功能
--
testdb=# SELECT * FROM friend
testdb=# WHERE name = 'TRAC1';
   name  | city | prov | age
-----+-----+-----+-----
 TRAC1   | 1    | 1    |
(1 row)
testdb=# SELECT change_prov('TRAC1','');

```

```
change_prov
-----

数据删除成功
(1 row)

testdb=# SELECT * FROM friend
testdb=# WHERE name = 'TRAC1';
 name | city | prov | age 
-----+-----+-----+-----
(0 rows)
```

## 20.4 触发器

PL/pgSQL语言不仅可以用来定义普通函数，也可以用来定义触发器。触发器提供一种实现插入、更新或删除的特殊方法，适用于在写入数据之前对数据进行检查和修改。利用触发器，每次修改一个数据行时可以调用一些特殊的服务器端函数，PL/pgSQL就是编写这种函数的常用语言。

### 20.4.1 触发器函数

在创建触发器之前，必须先创建触发器所使用的函数，也就是触发器被触发时所进行的动作。

触发器函数的特点是没有任何入口参数，函数返回值的类型为 OPAQUE。此外，PostgreSQL 为触发器函数定义了一些特殊的变量，主要的变量有：

- (1) NEW，数据类型为 RECORD。这个变量保存行级触发器在进行 INSERT、UPDATE 操作时的新数据行。
- (2) OLD，数据类型为 RECORD。这个变量保存行级触发器在进行 INSERT、UPDATE 操作时的旧数据行。
- (3) TG\_NAME，数据类型为 name。这个变量包含实际触发的触发器名称。
- (4) TG\_WHEN，数据类型为 text。这个变量存储一个由触发器定义所决定的字符串，这个字符串或者是“BEFORE”，或者是“AFTER”。
- (5) TG\_LEVEL，数据类型为 text。这个变量也存储一个由触发器定义所决定的字符串，“ROW”或“STATEMENT”。
- (6) TG\_OP，数据类型为 text。这个变量存储一个说明触发器实际进行操作的字符串，可以是“INSERT”、“UPDATE”或“DELETE”。
- (7) TG\_RELID，数据类型为 oid。这个变量存储导致触发器触发的对象标识号（OID）。
- (8) TG\_RELNAME，数据类型为 name。这个变量存储激活触发器的表名称。

(9) TG\_NARGS, 数据类型为 integer。表明 CREATE TRIGGER 命令赋予触发器过程的参数个数。

(10) TG\_ARGV[], 数据类型为 text 数组, CREATE TRIGGER 命令中的参数。下标从 0 开始, 并且可以由一个表达式来表示。非法下标 (< 0 或 >= tg\_nargs) 将导致一个 NULL 值的返回。

另外, 触发器函数的返回值只有两种可能, 一种是 NULL, 另一种是一个与导致触发器触发的表记录结构完全一样的数据。AFTER 类型的触发器总是返回一个没有意义的 NULL 值。对于 BEFORE 类的触发器, 如果返回一个 NULL, 系统将发送一个信号给触发器管理器, 忽略对数据行的实际操作; 否则, 返回的记录将代替插入或更新操作中的原始数据行。

## 20.4.2 触发器的创建

创建触发器的命令是:

```
CREATE TRIGGER trigger [ BEFORE | AFTER ] [ INSERT | DELETE | UPDATE [ OR ... ] ]
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE func ( arguments )
```

其中, trigger 为触发器的名字。BEFORE (AFTER) 指明触发器函数在事件发生之前 (之后) 调用。INSERT、DELETE、UPDATE 规定在什么事件上触发该函数, 多个事件可以用 OR 分隔。ROW (STATEMENT) 规定触发器为每个受影响的行触发 (在整个语句完成之前或之后触发)。

## 20.4.3 触发器的使用

下面的例子为 friend 表创建一个触发器, 主要功能是检验数据的合法性:

- (1) 姓名不能为空, 并且长度不能小于 4。
- (2) 年龄不能为空, 且必须在 0 至 80 范围之内。
- (3) 省市名称的长度不能小于 4。

--创建触发器函数

```
testdb=# CREATE FUNCTION trigger_friend()
testdb=# RETURNS opaque
testdb=# AS 'BEGIN
testdb'# IF new.name IS NULL THEN
testdb'# RAISE EXCEPTION '姓名不能为空';
testdb'# END IF;
testdb'# IF length(trim(new.name))<4 THEN
testdb'# RAISE EXCEPTION '姓名至少有两个汉字';
testdb'# END IF;
testdb'# IF new.age IS NULL THEN
```



```

testdb=# RAISE EXCEPTION '年龄不能为空';
testdb=# END IF;
testdb=# IF new.age<0 OR new.age>80 THEN
testdb=# RAISE EXCEPTION '年龄超出范围(0~90)';
testdb=# END IF;
testdb=# IF new.age<0 OR new.age>80 THEN
testdb=# RAISE EXCEPTION '年龄超出范围(0~90)';
testdb=# END IF;
testdb=# IF new.prov IS NOT NULL AND length(trim(new.prov))<4 THEN
testdb=# RAISE EXCEPTION '省市名称至少有两个汉字';
testdb=# END IF;
testdb=# RETURN new;
testdb=# END;
testdb=# LANGUAGE 'plpgsql';
CREATE
--创建触发器
testdb=# CREATE TRIGGER trigger_friend_ID
testdb=# BEFORE INSERT OR UPDATE
testdb=# ON friend
testdb=# FOR EACH ROW
testdb=# EXECUTE PROCEDURE trigger_friend();
CREATE
--测试触发器的作用
testdb=# INSERT INTO friend
testdb=# VALUES('李','西安','山西',10);
ERROR:  姓名至少有两个汉字
testdb=# INSERT INTO friend
testdb=# VALUES(NULL,'西安','山西',10);
ERROR:  姓名不能为空
testdb=# INSERT INTO friend
testdb=# VALUES('李刚','西安','山西',-10);
ERROR:  年龄超出范围(0~90)

```

触发器为保证数据的一致性和完整性提供了强有力的手段。

## 第21章

# Web 数据库应用

---

PostgreSQL被誉为是“天然”的Web数据库，能够与Apache以及PHP很好地协调工作，因此在因特网上有着非常广泛的应用。本章介绍将PostgreSQL应用于Web数据库服务器的基本方法，主要包括：

- 环境的安装与配置

- 数据库访问函数

- 数据库操作



## 21.1 环境安装与配置

与 Web 服务器以及其他工具相结合, PostgreSQL 可用于建立 Web 数据库应用系统。在 Linux 平台上, Apache+PHP+PostgreSQL 被誉为 Web 数据库应用的经典组合。本节主要介绍 Apache 和 PHP 的安装与配置方法。

### 21.1.1 Apache 服务器

Web 数据库的应用环境首先需要一个 Web 服务器。在众多的支持 HTTP 协议的 Web 服务器中, Apache 是一种使用非常广泛的 Web 服务器, 据统计, Internet 上超过半数的 WWW 主机使用 Apache。Apache 服务器的主要特点是稳定性高、速度快、功能强、可扩展性好、开放源代码, 可以免费获得和使用。Apache 可广泛运行于多种操作系统, 包括 Linux、FreeBSD、Windows、OS/2 以及其他各种版本的 UNIX。

绝大多数 Linux 发行版都包含 Apache 服务器软件, 并将 Apache 作为网络组件之一, 因此只要在安装 Linux 时选择 Apache 选项就可以完成 Apache 的安装。

如果在 Linux 安装时没有选择 Apache, 也可以在 Linux 安装完毕后, 手工安装 Apache。在不同的 Linux 版本中, 安装 Apache 的方法也有所不同。这里介绍在 RedHat Linux 6.x 版以及 RedHat Linux 7.x 版下安装 Apache 的基本方法。

#### 1. RedHat Linux 6.x

RedHat Linux 6.x(或与之相当的其他 Linux 发行版, 如 Turbo Linux 4.x、Tom Linux 1.0 等)中, Apache 软件包位于光盘的/RedHat/RPMS 目录中, 文件名为 apache-1.3.6-4.i386.rpm 和 apache-devel-1.3.6-4.i386.rpm, 前者为 Apache 服务器的二进制执行代码, 后者为开发工具。还有一个名为 apache\_php3-1.3.6-3.i386.rpm 的文件, 这是带有 php3(一种服务器端包含语言)模块的 Apache 服务器。php3 功能强大, 是实现动态 Web 页面和 Web 数据库的理想工具, 因此大多数用户选择这种版本的 Apache。下面是手工安装 Apache+php3 的命令:

```
rpm -ivh apache-devel-1.3.6-4.i386.rpm
rpm -ivh apache_php3-1.3.6-3.i386.rpm
```

在 Turbo Linux 中, 也可以利用 turbopkg 实用程序来安装 Apache。

一般情况下, Linux 发行版本所带的 Apache 版本不是最新版本, 最新版本可以在 [www.apache.org](http://www.apache.org) 站点上下载。遗憾的是, 该站点提供的文件不是 Linux 专用的 RPM 包格式, 而是 UNIX 常见的压缩文件格式, 所以需要自行解压缩、编译和安装。

对于 RedHat Linux 6.x 的用户, 建议升级到 7.x 或更新版本, 至少应该升级 Apache 的版本。升级版的安装方法可以参照 RedHat Linux 7.x 下安装 Apache 的方法。

## 2. RedHat Linux 7.x

在 RedHat Linux 7.x 中, Apache 软件包也存放在/RedHat/RPMS 目录中, 但文件有所不同, 主要的有 apache-1.3.12-25.i386.rpm、apache-devel-1.3.12-25.i386.rpm 和 apache-manual-1.3.12-25.i386.rpm。

对于一般的应用, 只需安装 apache-1.3.12-25.i386.rpm, 使用的命令是:

```
rpm -ivh apache-1.3.12-25.i386.rpm
```

在 RedHat Linux 7.0 中, Apache 的版本比较新, 稳定性也比较好。特别需要说明的是, 新版的 Apache 对可装载的 Apache 模组的支持更加完善, 各种扩充 Apache 功能的代码都可以以模组的形式由 Apache 在运行需要时动态装载。PHP 也可以作为模组被 Apache 装载和使用, 因此在新的 Linux 版本中, 不再有 apache\_php 之类的 RPM 软件包了。

### 21.1.2 PHP 脚本语言

在 Web 数据库应用中, 除了 Web 服务器和数据库服务器外, 还必须有一种“粘合剂”将二者连接起来, PHP 就是这样的一种“粘合剂”。

PHP 是近年来发展起来的一种服务器端嵌入式脚本语言, 也是一种自由软件, 最新版本为 4.0.3。PHP 的功能非常强大, 许多商业网站都使用 PHP 作为服务器端的嵌入脚本语言。

将 PHP 代码嵌入 HTML 页面代码之中, 是制作动态 Web 页面的典型方法之一。基本嵌入方法类似于下面的形式:

```
<HTML>
<HEAD>
<TITLE>php3</TITLE>
</HEAD>
<BODY BGCOLOR=#FFFFFF TEXT=#000000>
<H1><?
    echo "Hello,world!"
?></H1>
</BODY>
</HTML>
```

对各种数据库提供全面的支持是 PHP 的主要特色之一, 并且, PostgreSQL 是 PHP 默认支持的数据库之一。

对于 RedHat Linux 6.x, 前面介绍的 Apache 安装方法已经安装了 PHP。

在 RedHat Linux 7.x 中, PHP 以模组的形式由 Apache 动态装载。PHP 模组文件存放在发行版的第二张光盘的/RedHat/RPMS 目录中, 文件名为 mod\_php-4.0.1pl2-9.i386.rpm。用于安装的命令是:

```
rpm -ivh mod_php-4.0.1pl2-9.i386.rpm
```

安装完毕后, Apache 模组目录/etc/httpd/modules 中就会有一个可动态装载的模组文件 libphp4.so。

### 21.1.3 配置

Apache 和 PHP 的安装成功并不表明二者就能很好的协调工作，还需要进行一些配置。

首先，必须让 Apache 能够自动装载 PHP 模组（如果安装的是 apache\_php3，则不必进行这一配置工作）。下面以 RedHat 7.x 为例，介绍配置装载 PHP 模组的方法。

- (1) 以 root 身份登录 Linux。
- (2) 进入/etc/httpd/conf 目录。这个目录是 Apache 配置文件目录。
- (3) 利用文本编辑器（如 vi），按照下面的提示编辑 Apache 的主配置文件 httpd.conf。

```
.....
#LoadModule put_module          modules/mod_put.so
<IfDefine HAVE_PERL>
LoadModule perl_module          modules/libperl.so
</IfDefine>
#<IfDefine HAVE_PHP>                #注释掉这一行
LoadModule php4_module          modules/libphp4.so    #PHP 的模组文件
#</IfDefine>                        #注释掉这一行
<IfDefine HAVE_PHP3>
.....
```

第二个需要进行的配置工作是使 Apache 能够通过文件扩展名识别出哪些页面文件中包含 PHP 脚本，这样 Apache 才能利用 PHP 模组解释 PHP 脚本。配置方法也是修改 httpd.conf 文件，接着上面的操作，在 httpd.conf 文件中找到下面的配置命令行，并按下面的提示进行修改：

```
.....
# The following is for PHP4 (conflicts with PHP/FI, below):
#<IfModule mod_php4.c>            #注释这一行或删除这一行
    AddType application/x-httpd-php .php4 .php3 .phtml .php
    AddType application/x-httpd-php-source .phps
#</IfModule>                      #注释这一行或删除这一行
```

最后，保存所进行的修改，退出文本编辑器，并重新启动 Apache。接下来就可以在 HTML 页面中嵌入 PHP 脚本，也可以利用 PHP 访问 PostgreSQL 数据库了。

可以用下面的方法测试 Apache 和 PHP 的安装与配置是否成功。

在/var/www/html 目录下，建立一个名为 testphp.php 文件，文件的内容为：

```
<HTML>
<HEAD>
<TITLE>php3</TITLE>
</HEAD>
<BODY BGCOLOR=#FFFFFF TEXT=#000000>
<H1><?
    echo "今天的日期是".date( "l dS of F Y h:i:s A" );"
```

```

phpinfo();

?></H1>

</BODY>

</HTML>

```

然后，在网络中的任一机器上，利用浏览器访问“http://<主机>/testphp.ph”，其中，“<主机>”是 Apache 服务器所在机器的主机名或 IP 地址。图 21-1 是在 Windows 98 下用 IE 浏览器所看到的效果。

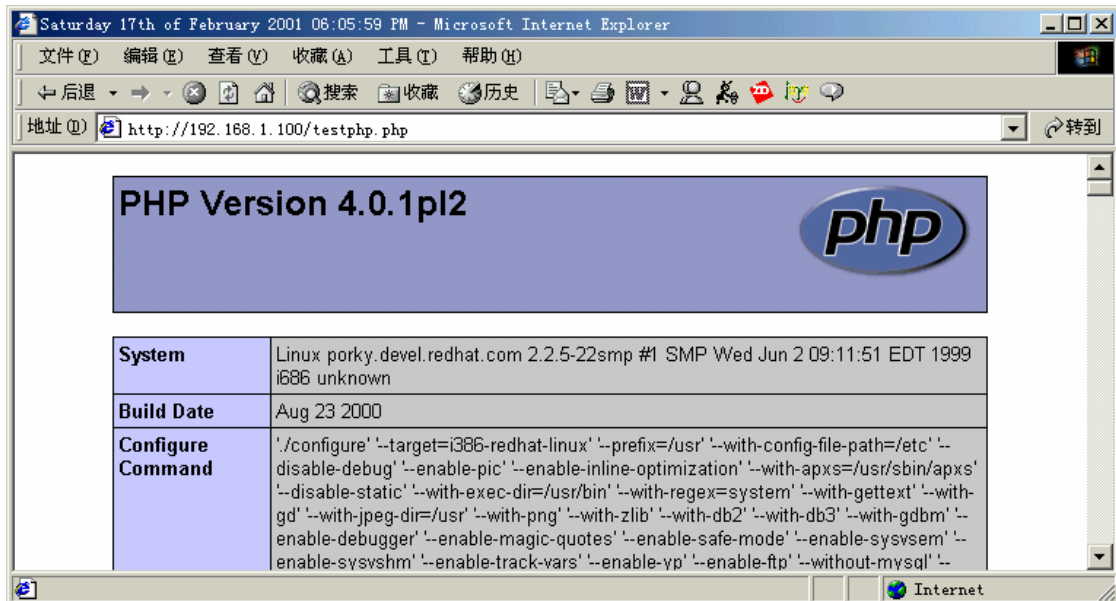


图 21-1 PHP 浏览效果

## 21.2 数据库访问函数

PHP 提供了一组用于访问 PostgreSQL 的函数(以 pg 为函数名头两个字母的函数),利用这些函数可以连接 PostgreSQL 数据库、查询数据、更新数据。

### 1. pg\_close (关闭数据库连接)

语法: boolean pg\_close(int connection)

返回: 逻辑值。

说明 本函数关闭与 PostgreSQL 服务器的连接。参数 connection 为连接代码。成功返回 true, 失败返回 false。

### 2. pg\_cmdtuples (获取 SQL 命令所影响的行数)

语法：int pg\_cmdtuples(int result\_id)

返回：整数。

说明：本函数用来获取PostgreSQL的SQL命令所影响的数据行数，这些命令包括INSERT、UPDATE与DELETE等。函数的返回值为行数，如果没有数据，则返回0。

例子：

```
<?
    $result=pg_exec($conn,"INSERT INTO verlag VALUES('Autor')");
    $cmdtuples=pg_cmdtuples($result);
    echo $cmdtuples."行数据<p>\n";
?>
```

### 3. pg\_connect（连接数据库）

语法：int pg\_connect(string [host], string [port], string [options], string [tty], string database)

返回：整数。

说明：本函数打开与PostgreSQL服务器的连接。参数host为服务器名称、port 为连接端口号、options为可选项、tty为终端名称、database为数据库名。如果连接成功，函数返回连接代码，否则返回false。

例子：

```
<?
    pg_connect("dbname=testdb user=wilson password=haha port=5433");
?>
```

### 4. pg\_dbname（获取当前数据库名称）

语法：string pg\_dbname(int connection)

返回：字符串。

说明：本函数用来获取PostgreSQL当前正在使用的数据库名称。参数connection为连接代码。

### 5. pg\_errormessage（返回错误信息）

语法：string pg\_errormessage(int connection)

返回：字符串。

说明：本函数用于获取PostgreSQL服务器的错误信息。参数connection为连接代码。

### 6. pg\_exec（执行查询命令）

语法：int pg\_exec(int connection, string query)

返回：整数。

说明：本函数用来执行由query指定的查询命令。参数connection为连接代码，参数query为要执行的命令。

## 7. pg\_fetch\_array (返回数组数据)

语法：array pg\_fetch\_array(int result, int row)

返回：数组。

说明：本函数用来将查询结果拆分到数组变量中。如果没有数据，则返回false。参数result为使用pg\_Exec()所返回的代码值，参数row 为行数。

例子：

```
<?
$conn=pg_pconnect("", "", "", "", "publisher");
if(!$conn){
    echo "错误，无法连接\n";
    exit;
}
$result=pg_Exec($conn, "SELECT*FROM authors");
if(!$result){
    echo "错误，无法查询\n";
    exit;
}
$arr=pg_fetch_array($result, 0);
echo $arr[0]. " < array\n";
$arr=pg_fetch_array($result, 1);
echo $arr["author"]. " < array\n";
?>
```

## 8. pg\_fetch\_object (返回对象数据)

语法：object pg\_fetch\_object(int result, int row)

返回：类。

说明：本函数用来将查询结果拆分到对象变量中。如果result没有数据，则返回false。参数result为pg\_Exec()返回的代码值，参数row为行数。

例子：

```
<?
$database="verlag";
$db_conn=pg_connect("localhost", "5432", "", "", $database);
if(!$db_conn):?>
    <b>无法连接 <? echo $database ?> 数据库</b><?
    exit;
endif;
$qu=pg_exec($db_conn, "SELECT*FROM verlagorderBY autor");
$row=0;
while($data=pg_fetch_object($qu, $row)):
    <? echo $data->author . " < array\n";
    $row++;
endwhile;
?>
```



```

        echo $data>autor . "(" ;
        echo $data>jahr . "): " ;
        echo $data>titel . "<BR>" ;

        $row++ ;
    endwhile ; ?>
<pre><?
$fields[]=Array("autor" , "Author") ;
$fields[]=Array("jahr" , " Year") ;
$fields[]=Array("titel" , " Title") ;
$row=0 ;
while($data=pg_fetch_object($qu , $row)) :
    echo "<hr><p>\n" ;
    reset($fields) ;
    while(list( , $item)=each($fields)) :
        echo $item[1] . " : " . $data>$item[0] . "\n" ;
    endwhile ;
    $row++ ;
endwhile ;
echo "<p><hr>" ; ?>
</pre><?
pg_freeResult($qu) ;
pg_close($db_conn) ;
?>

```

## 9. pg\_fetch\_row (返回数据行的各字段)

语法：array pg\_fetch\_row(int result , int row)

返回：数组。

说明：本函数用以将查询结果中某行数据拆分到数组变量中。数组的索引是数字索引，第一个索引值是0。如果result没有数据，则返回false。参数row为行数。

例子：

```

<?
$conn=pg_pconnect("", "", "", "", "publisher");
if(!$conn){
    echo "连接失败\n";
    exit;
}
$result=pg_Exec($conn, "SELECT*FROM authors");
if(!$result){
    echo "查询失败\n";
}

```

```
        exit ;
    }
    $row=pg_fetch_row($result,0);
    echo $row[0]. " < row\n";
    $row=pg_fetch_row($result,1);
    echo $row[0]. " < row\n";
    $row=pg_fetch_row($result,2);
    echo $row[1]. " < row\n";
?>
```

#### 10. pg\_fieldisnull (检查字段是否为空)

语法：int pg\_fieldisnull(int result,int row,mixed field)

返回：整数。

说明：本函数可检查指定的字段是否有数据。参数result为查询结果代码；参数row为指定行号，第一行的行号为0；参数field为指定的字段。返回0表示有数据，返回1表示无数据(Null)。

#### 11. pg\_fieldname (返回指定字段的名称)

语法：string pg\_fieldname(int result,int field)

返回：字符串。

说明：本函数用来获取指定字段的名称。参数result为查询结果代码。参数field为指定的字段编号，第一个字段的编号为0。

#### 12. pg\_fieldnum (获取指定字段的行数)

语法：int pg\_fieldnum(int result,mixed field)

返回：整数。

说明：本函数用来获取指定字段的行数。参数result为查询结果代码；参数field为指定的字段的编号，第一个字段的编号为0。如果有错误则返回1。

#### 13. pg\_fieldprtlen (计算字段显示长度)

语法：int pg\_fieldprtlen(int result,int row,mixed field)

返回：整数。

说明：本函数用来获取指定字段的显示长度。参数result为查询结果代码。参数row为指定的行。参数field为指定字段的编号，第一个字段的编号为0。如果有错误则返回1。

#### 14. pg\_fieldsize (计算指定字段的长度)

语法：int pg\_fieldsize(int result,mixed field)

返回：整数。

说明：本函数用来获取指定字段的长度。参数result为查询结果代码；参数field为指定字

段的编号，第一个字段的编号为0。如果有错误则返回1。

### 15. pg\_fieldtype (获得当前字段的类型)

语法：`string pg_fieldtype(int result, mixed field)`

返回：字符串。

说明：本函数可以得到当前字段的类型格式。返回的字符串为字段的类型，包括int、real、string等等。参数result为查询结果代码；参数field为指定字段的编号，第一个字段的编号为0。如果有错误则返回1。

### 16. pg\_freeresult (释放内存)

语法：`boolean pg_freeresult(int result)`

返回：逻辑值。

说明：本函数可以用来释放当前PostgreSQL数据库查询结果所占用的内存。一般只在内存不足的情况下才会需要使用本函数。PHP程序会在结束时自动释放内存。参数result为查询结果代码。

### 17. pg\_getlastoid (获取最后的对象标识号)

语法：`int pg_getlastoid(int result)`

返回：整数。

说明：本函数获取最后的对象标识号，对象标识号是由pg\_Exec()所执行的INSERT命令产生的。参数result为查询结果代码。如果执行的命令不是INSERT，则会产生错误并返回1。

### 18. pg\_host (获取连接机器名称)

语法：`string pg_host(int connection)`

返回：字符串。

说明：本函数用来获取当前连接的PostgreSQL服务器名称。参数connection为连接代码。

### 19. pg\_loclose (关闭大对象)

语法：`void pg_loclose(int fd)`

返回：无。

说明：本函数用来关闭大对象(Inversion Large Object)。参数fd是由函数pg\_loopen()打开的文件代码。

### 20. pg\_locreate (建立大对象)

语法：`int pg_locreate(int connection)`

返回：整数。

说明：本函数用来建立大对象。参数connection为连接代码。PostgreSQL数据库的存取模式只有INV\_READ和INV\_WRITE 两种。函数的返回值为对象代码。

### 21. pg\_loopen (打开大对象)

语法：`int pg_loopen(int connection, int objoid, string mode)`

返回：整数。

说明：本函数用来打开大对象。参数`connection`为连接代码。参数`objoid`为大对象代码；参数`mode`的值有只读(r)、只写(w)及可读写(rw)等三种。函数的返回值为文件代码。

## 22. pg\_loread (读取大对象)

语法：`string pg_loread(int fd, int len)`

返回：字符串。

说明：本函数用来读取大对象。参数`fd`为类的文件代码。参数`len`为要读取的大对象的最大长度。

## 23. pg\_loreadall (读取大对象并输出)

语法：`void pg_loreadall(int fd)`

返回：无。

说明：本函数用来读取大对象，并将结果输出到标准输出设备中。参数`fd`为类的文件代码。

## 24. pg\_lounlink (删除大对象)

语法：`void pg_lounlink(int connection, int lobjid)`

返回：无。

说明：本函数用来删除大对象。参数`connection`为连接代码；参数`objoid`为对象代码。

## 25. pg\_lowrite (写大对象)

语法：`int pg_lowrite(int fd, string buf)`

返回：整数。

说明：本函数用来写入大对象。参数`fd`为类的文件代码；参数`buf`为要写入大对象的字符串。如果函数执行有错误则返回`false`，否则返回写入字符串的字节数。

## 26. pg\_numfields (获取结果的字段数)

语法：`int pg_numfields(int result)`

返回：整数。

说明：本函数可以得到查询结果中的字段数目。参数`result`为PostgreSQL查询的返回代码。如果有错误则返回1。

## 27. pg\_numrows (获取结果行数)

语法：`int pg_numrows(int result)`

返回：整数。

说明：本函数可以得到查询结果中的数据行的数目。参数`result`为PostgreSQL查询的返回代码。如果有错误则返回1。

### 28. pg\_options (获取服务器可选项)

语法：`string pg_options(int connection)`

返回：字符串。

说明：本函数用来获取当前连接中PostgreSQL服务器的可选项。参数connection为连接代码。

### 29. pg\_pconnect (建立持续的数据库连接)

语法：`int pg_pconnect(string [host], string [port], string [options], string [tty], string database)`

返回：整数。

说明：本函数打开与PostgreSQL服务器的持续连接。参数host为服务器名称；参数port 为连接端口号；参数options为可选项；参数tty为终端名称；参数database为数据库名。如果连接成功，函数返回连接代号；否则返回false。本函数与pg\_Connect()不同之处在于，使用本函数打开数据库时，程序会先寻找是否曾经执行过本函数，如果执行过则返回老的连接代码。此外，程序结束后不会关闭这种连接，以便其他程序使用。

### 30. pg\_port (获取连接端口)

语法：`int pg_port(int connection)`

返回：整数。

说明：本函数用来获取当前连接中PostgreSQL服务器的端口号。参数connection为连接代码。

### 31. pg\_result (获取查询结果)

语法：`mixed pg_result(int result, int row, mixed field)`

返回：混合类型数据。

说明：本函数获取查询结果中的一行数据。参数field可以是字段名称、字段序号或者fieldName.TableName格式的字段。

### 32. pg\_tty (获取连接终端名)

语法：`string pg_tty(int connection)`

返回：字符串。

说明：本函数用来获取当前与 PostgreSQL 服务器连接的终端名称，供调试使用。参数connection 为连接代码。

## 21.3 数据库操作

本节通过一些例子，介绍在 Web 页面中利用 PHP 操作 PostgreSQL 数据库的方法。这些例子的重点是数据库的操作，因此页面的布局和外观可能比较粗糙。

### 21.3.1 连接数据库服务器

在对操作数据库进行操作之前，必须先建立与数据库服务器的连接。在PHP中，有两个用于建立数据库连接的函数：`pg_connect`和`pg_pconnect`。前者建立的连接在PHP程序结束时会自动关闭；后者建立的连接称为持续连接，在PHP程序结束时不会自动关闭。在建立这种连接时，PHP总是先寻找是否曾经执行过本函数，如果执行过则返回老的连接代码。显然，后者能够在一定程度上提高系统运行效率，因为它能够显著地减少向服务器请求连接数据库的次数，但消耗的服务器内存资源较多。

连接函数要求提供一些连接参数，主要的有：

(1) 主机名称。如果利用TCP/IP建立与数据库服务器的连接，则必须指定PostgreSQL服务器所在的主机名称。除此之外，还必须对PostgreSQL进行配置，使之允许建立TCP/IP连接，本书的4.1节介绍了有关的一些知识。

另外，还可以通过修改PostgreSQL的主机访问控制文件`pg_hba.conf`（在`/var/lib/pgsql/data`目录中），来更精确地进行安全控制。

例如，为了允许IP地址为192.168.1.10的机器能够通过TCP/IP连接PostgreSQL数据库服务器，可以将下面的语句加入到`pg_hba.conf`中。

```
host    all            192.168.1.10          255.255.255.0          trust
```

有关 `pg_hba.conf` 文件的详细配置方法，请参考该文件中的说明信息。

如果 Apache 服务器与 PostgreSQL 服务器共用一台机器，那么就没有必要通过 TCP/IP 建立连接，可以直接使用本地认证，不必提供主机名称。

(2) TCP/IP 端口。默认情况下，PostgreSQL 在 5432 号端口监听 TCP/IP 请求，用户也可以为 PostgreSQL 指定其他端口。如果通过 TCP/IP 连接数据库服务器，并且数据库服务器使用了非默认的端口，则在连接数据库时必须指定连接端口。

(3) 用户名和口令。如果通过 TCP/IP 建立与数据库服务器的连接，必须提供用户名和口令。这里的用户是 PostgreSQL 的合法用户，口令也必须是 PostgreSQL 的用户口令。

在本章以前的所有章节中，都是利用 PostgreSQL 的 shell 用户认证机制。在这种机制下，对于 shell 用户，后端数据库服务器进行 `setuid` 调用，在将用户标识号转换为用户之前，检查用户的有效用户标识号。有效的用户标识号被当作访问控制检查的基础，不再进行其他的认证。因此，在利用 `psql` 连接 PostgreSQL 时，不必指定口令。

通过网络连接 PostgreSQL 的用户必须提供合法的用户名和口令。PostgreSQL 合法的用户名和口令信息存放在系统表 `pg_shadow` 中, PostgreSQL 的超级用户(典型的是 `postgres`)可以修改这个表,如更改口令、权限等。利用 `createuser` 创建的 PostgreSQL 用户的口令为空,这不利于系统的安全性。

(4) 数据库。由于任何与 PostgreSQL 的连接必须在某个数据库上进行,因此这个参数是必须指定的。

`pg_connect` 函数指定了入口参数的顺序,可以按照这个顺序填写连接参数。PHP 允许使用命名变量的形式填写函数的入口参数,这种参数填写方法更直观,也不容易发生错误。下面的例子都使用这种方法。

第一个连接的例子使用 PostgreSQL 的本地安全认证机制进行用户认证。

```
<?
$conn=pg_connect("           //本地连接的例子
    dbname=testdb           //连接的数据库名
    user=postgres");        //连接的用户名
if(!$conn) {
    echo "错误,无法连接";
    exit;
} else {
    echo "连接成功";
}
?>
```

如果 `pg_connect` 函数成功地建立了与 PostgreSQL 的连接,那么函数将返回一个非 0 值,这个值称为连接代码,其他操作数据库的函数都要使用这个连接代码。如果连接不成功, `pg_connect` 函数返回 0。

```
<?
$conn=pg_connect("           //通过网络连接的例子
    host=192.168.1.10        //连接的数据库服务器主机名
    dbname=testdb           //连接的数据库名
    user=postgres           //连接的用户名
    password=970617         //口令
if(!$conn) {
    echo "错误,无法连接";
    exit;
} else {
    echo "连接成功";
}
?>
```

### 21.3.2 查询数据

下面的代码以表格的形式列出了 friend 表中的所有数据。

```
<?
# 连接数据库
$pgconn = pg_pconnect("dbname=testdb user=postgres")
    or die("无法连接数据库");

# 执行查询 SELECT * FROM friend
$result = pg_exec($pgconn,"SELECT * FROM friend;")
    or die("无法查询数据");

$title = 'PostgreSQL Web 数据库查询';
# 获取查询结果的行数。
$count = pg_numrows($result);

?>
<html><head><title>
<?
    echo $title
?>
</title>
</head>
<body>
<FONT FACE="Verdana, Arial, Helvetica">
<h1 align=center>friend 表的数据</h1>
<table align=center border="2">
    <tr>
        <td>姓名</td>
        <td>年龄</td>
        <td>所在的省份</td>
        <td>所在的城市</td>
    </tr>
<?
# 显示查询结果。
for ($i=1;$i<=$count;$i++) {
    $arr = pg_fetch_row($result,$i-1);
    echo "<tr>";
    echo "    <td>".$arr[0]."</td>";
    echo "    <td>".$arr[3]."</td>";
    echo "    <td>".$arr[2]."</td>";
    echo "    <td>".$arr[1]."</td>";
```



```

        echo "</tr>";
    }
    ?>
</table>
</body>
</html>

```

执行结果如图 21-2 所示。



图 21-2 数据库查询结果

在上面的代码中，数据库服务器连接成功后，连接代码存放在变量\$pgconn 中以供后面的函数使用。pg\_exec 函数在由\$pgconn 所指定的连接（实际上是一次数据库会话）上执行指定的查询。如果执行成功，该函数返回标识该查询结果的代码。pg\_numrows 函数计算查询结果中的数据行数。在此后的循环中，pg\_fetch\_row 函数从查询结果中取出指定的一行数据，并存入数组变量\$arr 中；而 echo 语句将\$arr 中的数据组织成 HTML 表格的形式输出。

### 21.3.3 插入数据

与利用 Web 页面查询数据库中的数据相比，插入数据的页面代码要复杂一些。为了插入数据，至少需要两个页面，一个页面提供给用户输入数据，另外一个页面将数据插入到数据库中。

下面的代码是提供给用户输入数据的页面（文件名为 testins.php）。

```

<html>
<head>
<title>更新数据</title>
</head>

```

```

<body>
<form method="POST" action="do_insert.php">

<p>姓名:<br>
<input type="text" name="f_name" size=30>
</p>

<p>所在的省份:<br>
<input type="text" name="f_prov" size=30>
</p>

<p>所在的城市:<br>
<input type="text" name="f_city" size=30>
</p>

<p>年龄:<br>
<input type="text" name="f_age" size=30>
</p>

<input type="submit" value="提交数据">

</form>

</body>
</html>

```

下面的代码用于插入数据（文件名为 do\_insert.php）

```

<? php_track_vars ?>
<?
# 连接数据库
$pgconn = pg_pconnect("dbname=testdb user=postgres")
        or die("无法连接数据库");

# 插入数据
$cmd = "INSERT INTO friend VALUES('$f_name', '$f_city', '$f_prov',
$f_age);" ;

$result = pg_exec($pgconn,$cmd)
        or die("数据插入不成功");

echo "<HTML><HEAD><TITLE>数据插入</TITLE></HEAD><BODY>";
echo "<H1 align=center>数据插入成功!</H1>";
echo "</BODY></HTML>";

```

?>

在第一个页面 (testins.php) 中, 用户输入了姓名、省份、城市和年龄。当用户按下了“提交数据”按钮后, 这些信息分别存放在 \$f\_name、\$prov、\$city 和 \$age 变量中。在第二个页面中, “php\_track\_vars” 语句使 PHP 跟踪第一个页面传递过来的变量值, 也就是 \$f\_name、\$prov、\$city 和 \$age 变量的值。接下来的代码, 将这些变量的值作为新记录的数据插入到表 friend 中。

执行结果如图 21-3 和图 21-4 所示。

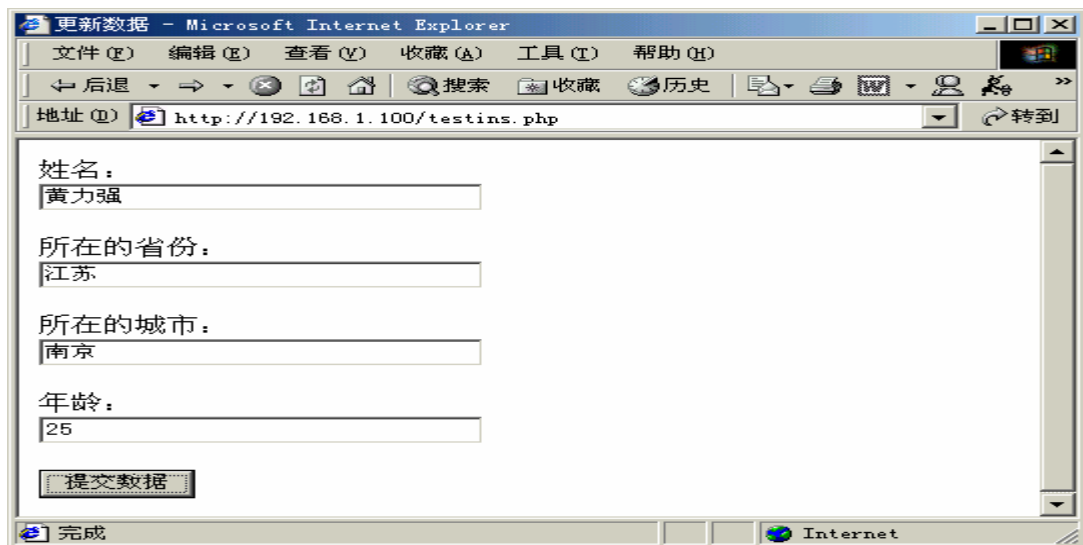


图 21-3 用户输入数据页面

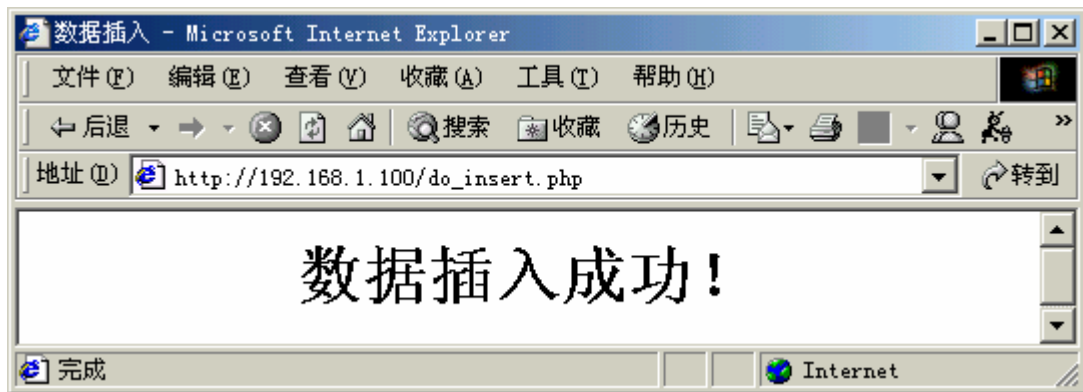


图 21-4 数据插入页面

上面的代码没有对可能出现的错误进行任何处理。

除了查询和插入数据外, PHP 还提供了更新数据、删除数据、管理表的结构、修改表的结构以及系统维护方面的功能。实际上, 在 PHP 中, 可以执行任何的 SQL 命令, 因此可以完成几乎所有的 PostgreSQL 数据库操作。

## 第22章

# phpPgAdmin

---

phpPgAdmin不仅是一个功能较强的数据库管理工具，也是一个典型的利用 Web 操纵 PostgreSQL 的例子。本章介绍 phpPgAdmin 的安装和基本使用方法，并对它的实现技术进行了初步的分析，主要包括：

- 安装与配置

- 使用方法

- 关键技术分析



## 22.1 安装与配置

### 22.1.1 简介

phpPgAdmin 是一个 PostgreSQL 数据库 Web 管理工具，借助这个工具，用户可以在 Web 浏览器中对 PostgreSQL 数据库进行管理和操作。它是一个自由软件，可以免费使用和自由传播。

phpPgAdmin 是从另外一个数据库管理工具（phpMyAdmin，MySQL 数据库管理工具）移植过来的，是用 HTML 和 PHP 语言编写的，可以在 Apache Web 服务器的支持下运行。它的主要功能包括：

- (1) 创建和删除 PostgreSQL 数据库。
- (2) 创建、复制、删除和修改表、视图、序列和函数。
- (3) 利用 PostgreSQL 的扩展特性，编辑字段属性或增加字段。
- (4) 执行任何的 SQL 命令，包括批量查询。
- (5) 管理关键字。
- (6) 将文本文件插入到数据表中。
- (7) 表数据的导入和导出管理。
- (8) 维护数据库。
- (9) 管理用户。

phpPgAdmin 目前的最新版本为 2.1，可以在网站 <http://www.greatbridge.org> 上下载 phpPgAdmin 代码（文件名为 phppgadmin\_xxx.tar.gz 或 phppgadmin\_xxx.zip，其中 xxx 为版本号）或查询它的最新信息。

phpPgAdmin 含有与 Web 数据库应用开发有关的各种代码，通过对它的分析，可以学习到许多 web 数据库开发技术。另外，phpPgAdmin 是一个功能相当完善的 PostgreSQL 管理工具，正好弥补了 PostgreSQL 缺乏易于操作的管理工具的缺陷。掌握 phpPgAdmin 的使用，有助于更方便、高效地管理 PostgreSQL。本章主要介绍在 Linux 中安装和使用 phpPgAdmin 的方法，并对它的实现细节进行了初步的分析。

### 22.1.2 安装

phpPgAdmin 的安装非常简单，在 Apache 的文档目录（典型的路径是 /var/www/html）下展开 phpPgAdmin 代码文件即可。主要的文件及其功能如表 22-1 所示。

表22-1 phpPgAdmin文件

函数	说明
index.php	phpPgAdmin 的主页面
config.inc.php	phpPgAdmin 的配置文件
left.php	phpPgAdmin 页面的左部框架，主要显示数据库的名称
main.php	创建数据库的页面
footer.inc.php	HTML 页面的尾部代码
header.inc.php	HTML 页面的首部代码
lib.inc.php	主要的数据库管理函数
catala.inc.php	( 西班牙 ) 加泰罗尼亚语言版本的屏幕提示信息
chinese_big5.inc.php	繁体中文 ( big5 码 ) 版本的屏幕提示信息
chinese_gb.inc.php	简体中文 ( gb 2312 码 ) 版本的屏幕提示信息
danish.inc.php	丹麦语版本的屏幕提示信息
english.inc.php	英语版本的屏幕提示信息
french.inc.php	法语版本的屏幕提示信息
german.inc.php	德语版本的屏幕提示信息
italian.inc.php	意大利语版本的屏幕提示信息
norwegian.inc.php	挪威语版本的屏幕提示信息
russian_koi8.inc.php	俄语 ( koi8 ) 版本的屏幕提示信息
russian_win1251.inc.php	俄语 ( win1251 ) 版本的屏幕提示信息
spanish.inc.php	西班牙语版本的屏幕提示信息
db_create.php	创建数据库
db_details.php	查询数据库详细情况
db_dump.php	导出数据库
db_readdump.php	导入数据库
func_create.php	创建函数
func_properties.php	查询函数细节
grp_admin.php	管理用户组
ldi_check.php	为装入文本文件生成 SQL 语句
ldi_table.php	向一个数据表插入一个文本格式的文件
seq_create.php	创建序列
tbl_addfield.php	为表增加一个字段
tbl_alter.php	修改表结构
tbl_create.php	创建表
tbl_dump.php	导出表
tbl_properties.php	查询表的详细定义
tbl_qbe.php	QBE 查询
tbl_rename.php	重新命名数据表
tbl_select.php	查询表数据
user_admin.php	管理用户
view_create.php	创建视图
ChangeLog	phpPgAdmin 软件的更新历史记录
DEVELOPERS	phpPgAdmin 开发者文档
Documentation.html	phpPgAdmin 配置说明，以及常见问题解答
INSTALL	安装说明文件
README	phpPgAdmin 说明文件
extchg.sh	更改 phpPgAdmin 文件扩展名的 shell 脚本

由于数据库管理工作可能涉及到对PostgreSQL的关键性操作，因此应该对phpPgAdmin的各个文件进行严格的保护。有关的保护方法请参考Apache Web服务器的有关文档。

较早版本（版本3或更低）PHP的默认文件扩展名为.php3，而phpPgAdmin中所有嵌入了PHP代码的文件扩展名均为.php，因此，在老版本系统上运行phpPgAdmin会有一些问题。可以利用phpPgAdmin自带的shell脚本extchg.sh将扩展名改为.php3。

## 22.1.3 配置

安装完毕后，需要对phpPgAdmin进行一些必要的配置，phpPgAdmin的配置文件的为config.inc.php。下面列出的是这个配置文件的内容，有关的配置方法也在文件中给出。

```
<?php
/* $Id: config.inc.php,v 1.7 2000/07/14 05:17:52 dan Exp $ */

// Set the name and version
$configProgName    = "phpPgAdmin"; //phpPgAdmin程序名
$configVersion     = "2.1";        //phpPgAdmin的版本号

//The default database is used to connect to the database to check the adv_auth
//This can actually be any database you currently have on your system. It just
//needs _a_ database to connect and check the system tables.
$configDefaultDB   = "template1"; //默认连接的数据库，这是PostgreSQL自动建立的库
//这个数据库主要用于用户身份认证

// You should change the superuser if different from postgres
// This is just used to filter out the system functions when listing
$configSuperUser   = "postgres";  //PostgreSQL超级用户名字

//Set to true if you want to authenticate against the passwd as well as the
username
// In order to use adv_auth, you must update the passwords in the user admin
section.
// It is suggested that you leave this as false until you are able to get
in and update the passwords.
$configUsePass     = true;        //是否使用用户名+口令进行用户认证

// Set to true if you want to use the valuntil field in the pg_user table
to verify users.
// WARNING: you must set the expiration field on the account with which
you want to login or else you will not get in.
// It is suggested that you leave this as false until you are able
to get in and update the expire fields.
$configUseExpire   = false;      //用户是否有使用期限的限制

// If you want to be able to view the contents and structure of the System
```

```

Catalog/Tables set this to true.

// If you are new to Postgres or are not familiar with the system tables,
it is suggested you leave this as false

$cfgSysTables    = false;           //是否需要浏览系统表

// If you want the web interface to administer postgres user, set this as
true.

$cfgUserAdmin    = true;           //是否需要进行PostgreSQL的用户管理

// Set this to true during development of phpPgAdmin
$cfgDebug        = false;         //是否需要phpPgAdmin输出调试信息

// If you do want to quote all relations, set this to false
$cfgQuotes       = true;          //是否列出数据库所有的关系（表、视图）

// If you do not want to display the tables of a database in the left frame,
add it to this array
// This feature is very useful when you have a database that has many tables
and slows down the load of the left frame.
// $cfgNoTables[] = "big_table";

// The $cfgServers array starts with $cfgServers[1]. Do not use $cfgServers[0].
// You can disable a server config entry by setting host to ''.
//phpPgAdmin可以定义三组PostgreSQL连接参数，无论这些连接是否在同一台机器上
//第一组连接参数
$cfgServers[1]['local']    = true;           //是否允许使用第一组参数
$cfgServers[1]['host']     = 'localhost';    //主机名
$cfgServers[1]['port']     = '5432';         //连接的TCP/IP端口号
$cfgServers[1]['adv_auth'] = true;           //是否对连接用户进行认证
$cfgServers[1]['stduser']  = '';             //标准用户名
$cfgServers[1]['stdpass']  = '';             //标准用户口令
$cfgServers[1]['user']     = '';             //连接的用户名
$cfgServers[1]['password'] = '';             //连接的用户口令
$cfgServers[1]['only_db']  = '';             //连接的数据库
//如果指定了连接数据库，则只能访问这个数据库，否则可以访问所有的数据库

//第二组连接参数，参数的意义同第一组
$cfgServers[2]['local']    = false;
$cfgServers[2]['host']     = '';

```



```

$cfgServers[2]['port']      = '5432';
$cfgServers[2]['adv_auth']  = true;
$cfgServers[2]['stduser']   = '';
$cfgServers[2]['stdpass']   = '';
$cfgServers[2]['user']      = '';
$cfgServers[2]['password']  = '';
$cfgServers[2]['only_db']   = '';

//第三组连接参数，参数的意义同第一组
$cfgServers[3]['local']     = false;
$cfgServers[3]['host']      = '';
$cfgServers[3]['port']      = '5432';
$cfgServers[3]['adv_auth']  = true;
$cfgServers[3]['stduser']   = '';
$cfgServers[3]['stdpass']   = '';
$cfgServers[3]['user']      = '';
$cfgServers[3]['password']  = '';
$cfgServers[3]['only_db']   = '';

// If you have more than one server configured, you can set $cfgServerDefault
// to any one of them to autoconnect to that server when phpPGAdmin is started,
// or set it to 0 to be given a list of servers without logging in
// If you have only one server configured, $cfgServerDefault *MUST* be
// set to that server.

$cfgServerDefault           = 1;    //默认的连接参数
$cfgServer                  = '';    //the selected server is copied here for
easier access

unset($cfgServers[0]); // Since 0 = no server, $cfgServers[0] must not be
used

//访问PostgreSQL文档的URL
$cfgManualBase              = "http://www.postgresql.org/docs/user";
// $cfgManualBase           = "file:/usr/local/pgsql/doc/user";

$cfgConfirm                 = true; //对可能造成数据丢失的操作，是否提示用户确认
$cfgPersistentConnections   = false; //是否与PostgreSQL建立持续连接

//重启PostgreSQL脚本的路径，目前尚未实现重启功能
// Path to Postgres programs

```

```
// Control Script that can be passed a restart.
// You may have to change the script to use a restart comment.
// If you do not have one on your system, you may use the one included in
this distribution (postgres.sh).

/***** Still not implemented *****/
$cfgPgAdmin      = "/usr/local/bin/pgadmin";
/*****

// -- I made this a symlink to my /etc/rc.d/init.d/postgres script

// If you have a table with several thousands of records, you will want to
set this false.
$cfgCountRecs          = true; //是否返回表的记录数

// If you want the fields alphabetized when inserting/editing
$cfgDoOrder             = false; //在插入或更新时, 是否按字母排序字段

//数据显示表格的外观参数
$cfgBorder              = "0";
$cfgThBgcolor           = "#D3DCE3";
$cfgBgcolorOne          = "#CCCCCC";
$cfgBgcolorTwo          = "#DDDDDD";
$cfgMaxRows             = 30;
$cfgMaxInputsize        = "300px";
$cfgOrder               = "ASC";

$cfgShowBlob            = false; //是否显示大对象数据
$cfgShowSQL             = true; //是否显示SQL命令

$cfgMember              = "#CCCCFF";
$cfgNonMember           = "#CCCC99";
$cfgMaxTries            = 10;

// Set your language preferences here.
//设置屏幕提示信息的语言版本, 对于简体中文用户, 可以使用chinese_gb.inc.php
//这个简体中文版屏幕提示信息有些地方不符合大陆的语言习惯, 需要作一些修改
include("chinese_gb.inc.php");

// Not even close to a complete list
// 设置用于对字段进行处理的函数, 这里列出的函数并不完整, 用户可以根据需要添加
```

```
$cfgFunctions = array(
    "CURVAL",
    "NEXTVAL",
    "TEXTCAT",
    "TEXTLEN"
);

?>
```

### 22.1.4 用户认证

需要特别注意的配置工作是用户认证方法。为了建立与PostgreSQL数据库服务器的连接，必须有合法的PostgreSQL用户名和口令，以便进行用户认证。phpPgAdmin提供两种形式的用户认证：基本认证和扩展认证。

#### 1. 基本认证

在基本认证形式中，利用浏览器访问phpPgAdmin的用户不必提供用户名和口令，连接PostgreSQL所需的用户名和口令由配置文件config.inc.php中定义的变量\$cfgServers[i]['user']和\$cfgServers[i]['password']指定，其中i为phpPgAdmin连接参数的组号（i=1、2或3）。

以第一组连接参数为例，为了使用这种认证方式，需要将配置文件中的\$cfgServers[1]['adv\_auth']设置为false，并将\$cfgServers[1]['user']和\$cfgServers[1]['password']设置为PostgreSQL合法的用户名和口令。如果Linux与PostgreSQL在同一台机器上，可以使用本地连接，这种情况下不需要提供口令。

这种认证方式的优点是用户不必知道用户名和口令，使用起来比较方便。但缺点也是十分明显的。首先，由于用户不需要提供用户名和口令，因此任何用户都可以进行PostgreSQL的管理。很明显，这样会存在严重的安全问题。当然，安全问题可以借助Apache的安全认证机制部分地解决。

其次，如果配置文件中的用户不是PostgreSQL的超级用户，那么利用phpPgAdmin只能进行该用户权限允许范围内的管理和操作，不能充分发挥phpPgAdmin的管理能力。如果配置文件中的用户为PostgreSQL超级用户，则会给不怀好意的用户以可乘之机。

为了得到数据库的核心信息，需要PostgreSQL超级用户身份，\$cfgSuperUser变量定义了一个合法的超级用户。

#### 2. 扩展认证

如果要得到更好的安全性和操作的简便性，最好使用phpPgAdmin的扩展认证方式对用户进行身份认证。

仍然以第一组连接参数为例，为了让phpPgAdmin使用扩展认证方式，需要将配置文件中的\$cfgServers[1]['adv\_auth']设置为true，并将\$cfgServers[1]['stduser']和\$cfgServers[1]['stdpass']设置为PostgreSQL合法用户名和口令。在扩展认证方式中，phpPgAdmin不使用变量\$cfgSer-

vers[1]['user']和\$cfgServers[1]['password']，可以将它们设置为空。

当用户试图访问phpPgAdmin的主页面时，浏览器首先会弹出一个窗口（如图22-1所示）询问用户名和口令。phpPgAdmin页面接收到用户名和口令后，会以变量的形式传递给PHP代码。PHP代码首先以 \$cfgServers[1]['stduser']和\$cfgServers[1]['stdpass'] 所标明的身份与PostgreSQL服务器建立一个连接，然后访问系统表pg\_shadow，验证用户名和口令是否合法。如果合法，用户就可以以它所提供的用户的身份操作数据库；否则phpPgAdmin会要求用户重新输入用户名和口令，或给出错误信息。

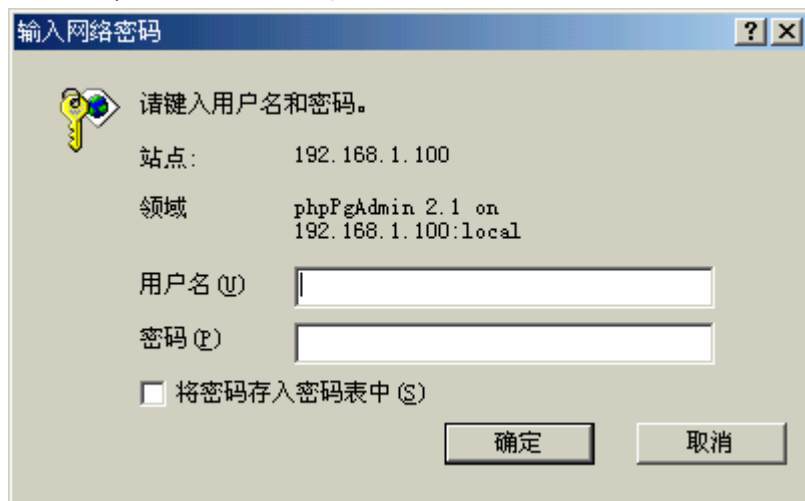


图22-1 浏览器用户登录窗口

从身份验证过程可以看出，首先，\$cfgServers[1]['stduser']和\$cfgServers[1]['stdpass']变量所标明的用户必须是PostgreSQL的合法用户，并且这个用户必须具有访问系统表pg\_shadow的权限。通常只有PostgreSQL的超级用户才具有这样的权限，对于普通用户，必须为其授予访问pg\_shadow的权限。

其次，浏览器用户所提供的用户名和口令必须与pg\_shadow中登记的某个用户匹配。不仅用户名要合法，而且口令也必须合法。特别值得注意的是，口令不能为空，否则同样不能通过身份认证。

当身份认证成功后，phpPgAdmin以用户提供的用户名和口令重新建立与PostgreSQL数据库的连接，其后所有的操作均以这个用户的身份进行。

这种认证方式不仅要求用户提供合法的身份认证信息，而且用户只能进行自己权限范围之内的操作，因此具有较高的安全性。存在的主要问题是，由于用户名和口令以明文的形式在网络上传输，因此存在被非法截取的可能性。解决这个问题需要更高的技巧，可以参考Apache Web服务器的有关资料。

## 22.2 使用方法

### 22.2.1 启动

phpPgAdmin的主页面文件为index.php，在任何HTTP兼容的浏览器中都可以启动phpPgAdmin，使用的URL为http://主机名/phpPgAdmin目录/index.php。启动并输入了合法的用户名和口令后，进入phpPgAdmin的主窗口，如图22-2所示。

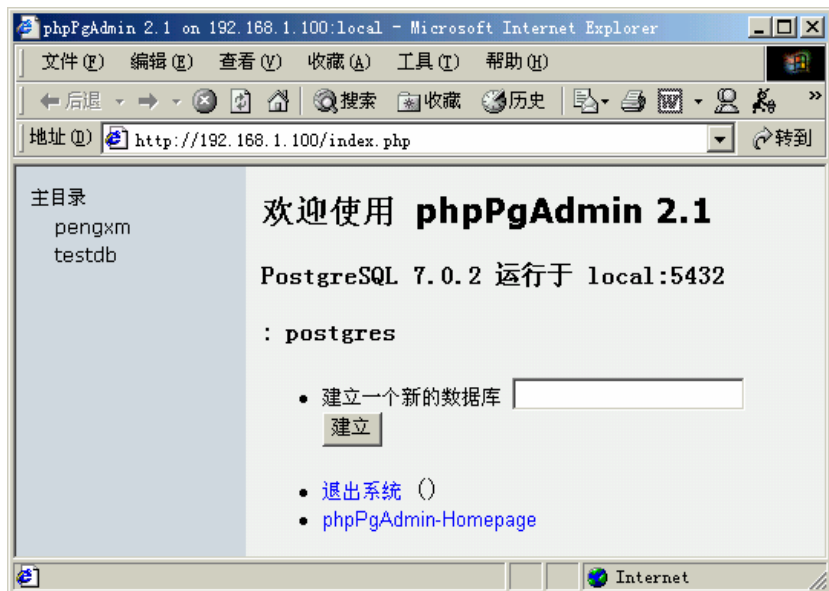


图22-2 phpPgAdmin主页面

### 22.2.2 基本操作

在phpPgAdmin中，所有的数据库和表的操作都通过HTML链接或按钮触发，具体的操作画面在相关的页面中。

主页面分为左右两个部分，左边显示的是主目录和所有的用户数据库，这些都是HTML链接，点击其中的“主目录”，进入phpPgAdmin的欢迎页面，也就是图22-2右边的画面。右边的页面是操作窗口，在其中可以创建一个数据库、退出系统或访问phpPgAdmin的主页（要求有Internet连接）。

主页面左边“主目录”之下的条目是当前连接的PostgreSQL服务器中的所有用户数据库，点击其中的任何一个进入该数据库的维护页面（如图22-3所示）。在维护页面中，可以浏览已有的表，插入数据，删除数据，修改表结构，创建函数、序列、视图，执行任何的SQL命令，导出表的数据等。

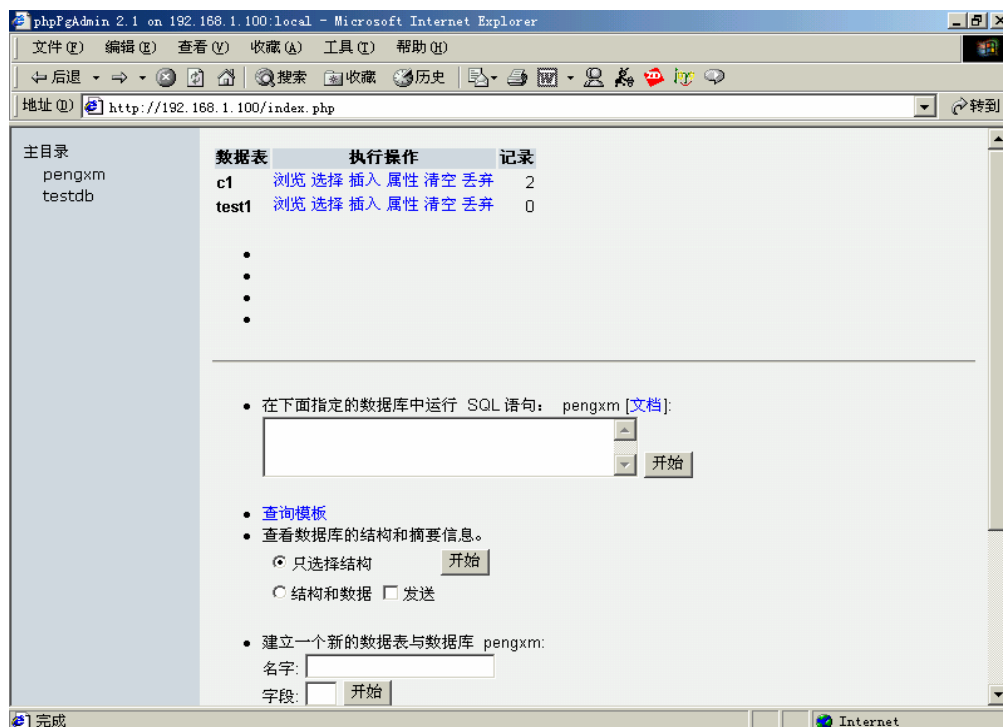


图22-3 数据库维护页面

窗口中，横线之上是当前数据库中的用户表和相关的操作链接。利用这些链接可以浏览表的数据（浏览），进行表的查询（选择），插入数据（插入），显示表的结构（属性），清空表的数据（清空），删除表（丢弃）。

横线之下的编辑窗口用于输入任何的SQL查询命令。SQL命令输入完毕后，按对应的“开始”键执行。

在接下来的“查询模板”所链接的页面中，可以以QBE的方式定制特殊的查询。

利用“查看数据库的结构和摘要信息”所对应的“开始”按钮，可以查询数据库的各种信息。

利用“建立一个新的数据表……”所对应的“开始”按钮，可以创建一个新的表。

利用HTML链接或按钮，可以创建序列、删除数据库、创建函数。

具体的数据库操作方法请按页面中的提示进行，这里就不再作进一步的介绍了。

## 22.3 关键技术分析

phpPgAdmin的功能并不复杂，但其实现技术不仅涉及到PostgreSQL，而且还涉及到PHP和Apache，因此有一些编程技巧值得分析和学习。本节将对主要的技术细节进行一些初步的介绍。

### 22.3.1 用户认证

如果phpPgAdmin使用扩展身份认证方式进行用户认证，第一次使用phpPgAdmin时，浏览器会弹出一个要求输入用户名和口令的窗口（见图22-1）。实际上，第一次访问任何phpPgAdmin页面时浏览器都将会弹出这个窗口。这是如何实现的呢？

这是PHP脚本利用Header函数向客户端浏览器发送一个“Authentication Required”消息的结果。当用户输入用户名和口令后，包含PHP脚本的URL将会被再次调用，这时变量\$PHP\_AUTH\_USER、\$PHP\_AUTH\_PW以及\$PHP\_AUTH\_TYPE被自动设置为用户输入的用户名、口令以及认证方式。PHP代码就可以根据这些变量进行身份认证。

下面的代码简要地说明了上述过程。

```
<?
if (!isset($PHP_AUTH_USER)) {
    Header("WWW-Authenticate: Basic realm=\"认证测试\"");
    Header("HTTP/1.0 401 Unauthorized");
    echo "您没有提供用户名或口令";
    exit;
} else {
    echo "您好, $PHP_AUTH_USER 。<p>";
    echo "您的密码是$PHP_AUTH_PW 。<p>";
    //.....身份验证代码
}
?>
```

当这个页面第一被访问时，变量\$PHP\_AUTH\_USER没有被设置和赋值，因此第一个Header函数会被执行。这个函数使客户端浏览器弹出类似于图22-1所示的窗口。如果在该窗口中按了“取消”按钮，\$PHP\_AUTH\_USER变量仍然没有被设置，页面将显示“您没有提供用户名或口令”的信息。如果输入了用户名和口令，并且按下了“确定”按钮，\$PHP\_AUTH\_USER和\$PHP\_AUTH\_PW变量就会被设置，根据这两个变量的值就可以对用户的身份进行确认。

在phpPgAdmin中，进行身份认证的代码在lib.inc.php文件中，相关的代码如下：

```

if(empty($PHP_AUTH_USER) || empty($PHP_AUTH_PW)) {
    header("status: 401 Unauthorized");
    header("HTTP/1.0 401 Unauthorized");
    //要求用户输入用户名和口令
    header("WWW-authenticate: basic realm=\"${realm}\"");
    //用户按下了“取消”按钮，显示错误信息
    echo "<HTML><HEAD><TITLE>" . $strAccessDenied . "</TITLE></HEAD>\n";
    echo "<BODY BGCOLOR=#FFFFFF><BR><BR><CENTER><H1>" . $strWrongUser .
    "</H1>\n";
    echo "</CENTER></BODY></HTML>";
    exit;
} else {
    //用户输入了用户名和口令，并按下了“确定”按钮
    //以下的代码根据$PHP_AUTH_USER和$PHP_AUTH_PW变量的值进行身份认证
    //首先连接数据库
    if ($dbh = pg_connect($conn_str)) {
        //数据库连接成功，从pg_shadow表中查找指定的用户
        //首先生成查询命令字符串
        $sql_get_auth = "SELECT username, passwd
                        FROM pg_shadow
                        WHERE username = '$PHP_AUTH_USER' ";
        if ($cfgUsePass) {
            //如果$cfgUsePass为真，验证用户的口令
            $sql_get_auth .= "AND passwd = '$PHP_AUTH_PW' ";
        }
        if ($cfgUseExpire) {
            //如果$cfgUseExpire为真，验证用户的有效期限
            $sql_get_auth .= "AND (valuntil > 'now' OR valuntil = NULL)";
        }
        //执行查询
        $rs = pg_exec($dbh, $sql_get_auth) or pg_die($dbh, $sql_get_auth);

        if (!pg_numrows($rs)) {
            //没有找到指定的用户，或口令错，或超过有效期
            $PHP_AUTH_USER = "";
            //重新要求提供用户名和口令
            Header("status: 401 Unauthorized");
            Header("HTTP/1.0 401 Unauthorized");
            header("WWW-authenticate: basic realm=\"${realm}\"");

```



```

        echo "<HTML><HEAD><TITLE>" . $strAccessDenied . "</TITLE></HEAD>\n";
        echo "<BODY BGCOLOR=#FFFFFF><BR><BR><CENTER><H1>" . $strWrongUser .
"</H1>\n";

        echo "</CENTER></BODY></HTML>";
        exit;
    }
    //用户身份认证成功，代码继续执行
} else {
    //无法与数据库进行连接，给出错误信息，停止继续使用
    pg_die("Unable to connect to server");
}
}

```

从上面的分析可以看出，在得到用户名和口令后，phpPgAdmin在PostgreSQL的系统表pg\_shadow中查找该用户，并进行口令和有效期的验证。这就要求必须有一个连接PostgreSQL的合法用户，并且这个用户还必须具有访问pg\_shadow表的权限。这个用户就是22.1.4小节中所介绍的扩展认证所需的用户。

### 22.3.2 库和表的列表

#### 1. 数据库列表

phpPgAdmin主页面的左边列出了所有的数据库名称，这些数据库名称信息是如何得到的呢？

在PostgreSQL中，所有的数据库名称都存放在系统表pg\_database中。在left.php文件中，有下面这样的一条命令，它从pg\_database中查询出用户数据库（除系统自己产生的template1外）。

```

SELECT datname
FROM pg_database
WHERE datname != 'template1' ORDER BY datname

```

#### 2. 数据表列表

当用户点击主页面左边列出的某个数据库时，主页面右边就会列出该数据库所拥有的所有数据表，这个数据表列表也是通过访问系统表而得到的。在left.php文件中有这样的一条命令：

```

SELECT tablename
FROM pg_tables
WHERE tablename !~* 'pg_*'
ORDER BY tablename;

```

这条命令的功能是查询当前连接的数据库所有的用户数据表（表名不以pg开头），其中

pg\_tables是一个视图，其基本表是pg\_class。也可以直接从pg\_class中得到所有的数据表。例如，在left.php中有这样的命令：

```
SELECT relname
FROM pg_class
WHERE not relname ~ 'pg_.*' and relkind = 'r'
ORDER BY relname;
```

除了表的列表外，通过同样的方法还可以得到视图列表、索引列表、序列列表和函数列表。下面的命令取自于db\_details.php文件。

--获取视图列表

```
SELECT viewname
FROM pg_views
WHERE viewname !~ 'pg_.*'
ORDER BY viewname;
```

--获取索引列表

```
SELECT relname
FROM pg_class
WHERE not relname ~ 'pg_.*' and relkind = 'i'
ORDER BY relname;
```

--获取序列列表

```
SELECT relname
FROM pg_class
WHERE not relname ~ 'pg_.*' and relkind = 'S'
ORDER BY relname;
```

--获取函数列表

```
SELECT
    proname,
    pt.typname as result,
    oidvectortypes(pc.proargtypes) as arguments
FROM
    pg_proc pc, pg_user pu, pg_type pt
WHERE
    pu.username != '$cfgSuperUser'
    AND pc.proowner = pu.usesysid
    AND pc.prorettype = pt.oid
```

### 22.3.3 表的维护

下面命令的功能是查询一个表的结构，其中\$stable为表的名字。这条命令取自于tbl\_details.php文件。

```

SELECT
    a.attnum,
    a.attname AS field,
    t.typname AS type,
    a.attlen AS length,
    a.atttypmod AS lengthvar,
    a.attnotnull AS notnull,
    a.atthasdef as hasdefault
FROM
    pg_class c,
    pg_attribute a,
    pg_type t
WHERE
    c.relname = '$table'
    and a.attnum > 0
    and a.attrelid = c.oid
    and a.atttypid = t.oid

```

tbl\_properties.php 列出某个表的所有字段，并允许用户改变字段的名字和默认值（tbl\_alter.php）、删除某个字段（tbl\_alter\_drop.php）、在某个字段上建立索引（sql.php）、指定某个字段为主键（sql.php）、添加新的字段（tbl\_addfield.php）。除此之外，还允许用户更改表的名字（tbl\_rename.php）和复制表的结构和数据（tbl\_copy.php）。

更为有趣的是，phpPgAdmin 允许用户将表的结构（或数据）导出到一个文件中。操作方法如图22-4所示。

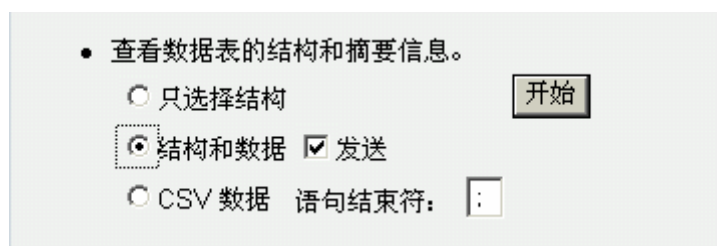


图22-4 导出表的结构和数据

为了将结构（和数据）导出到一个文件中，必须选中“发送”复选框。按下“开始”键后，浏览器弹出一个类似于下载文件的窗口，要求用户指定导出文件的文件名。当用户指定了文件名后，phpPgAdmin 会按要求导出结构（和数据）并存放在指定的文件中。

实现这一功能的代码在tbl\_dump.php文件中，核心的代码如下：

```

.....
//仿真以HTTP方式下载文件的HTTP头部信息
header("Content-disposition: filename=$table.$ext");
header("Content-type: application/octetstream");

```

```

header("Pragma: no-cache");
header("Expires: 0");

// doing some DOS-CRLF magic...
$client = getenv("HTTP_USER_AGENT");
if (ereg('[^]*\((.*)\[^\)]*', $client, $regs)) {
    $os = $regs[1];

    // this looks better under WinX
    if (ereg("Win", $os)) $crlf="\r\n";
}
//接下来是输出表结构和数据的代码
//注释信息
print "$crlf/* -----
$crlf";

print " $cfgProgName $cfgVersion DB Dump$crlf";
print " http://www.phpwizard.net/phpPgAdmin/$crlf";
print " $strHost: " . $cfgServer['host'];

if (!empty($cfgServer['port'])) {
    print ":" . $cfgServer['port'];
}
print "$crlf $strDatabase: $db$crlf";
print " $strTableStructure $cfgQuotes$table$cfgQuotes $crlf";
print "----- */ $crlf
$crlf";

//输出表的结构
print get_table_def($link, $table, $crlf)."$crlf";

if ($what == "data") {
    //输出表的数据
    print "$crlf/* -----
$crlf";

    print " $strDumpingData $cfgQuotes$table$cfgQuotes $crlf";
    print "----- */
$crlf";

    //输出表注释信息
    get_table_content($link, $table, "my_handler");
}

```

```
} else { // $what != "csv"
    get_table_csv($link, $table, $separator, "my_csvhandler");
}
.....
```

### 22.3.4 用户的管理

phpPgAdmin虽然提供了用户管理功能,但并没有将其纳入主页面之中。如果要使用这一功能,必须直接访问user\_admin.php文件。

PostgreSQL的用户信息存放在系统表pg\_shadow中,视图pg\_user提供了更为安全的信息访问途径。获得所有普通用户的命令是:

```
SELECT * FROM pg_user
    WHERE username NOT IN ('root', '$cfgSuperUser')
    ORDER BY usesysid;
```

user\_admin.php文件仅提供删除用户、更改用户属性的功能。

## 第23章

# ODBC 应用

---

借助ODBC，可以利用PostgreSQL建立传统的客户/服务器应用。本章介绍如何在Windows 95/98/NT下，利用Delphi访问和处理Linux服务器上的PostgreSQL数据库中的数据，主要内容包括：

- ODBC 的获取与安装

- ODBC 的配置

- 注意事项

- ODBC 的使用



## 23.1 ODBC 的获取与安装

用于 PostgreSQL 的 ODBC 驱动程序可在 <http://www.insightdist.com/psqlodbc> 站点上下载，当前版本为 6.50.000，文件名为 postdrv.exe，长度为 1154KB。postdrv.exe 是一个自展开安装文件，双击该文件图标后即可进行软件的安装。

在安装过程中会出现如图23-1所示的画面：

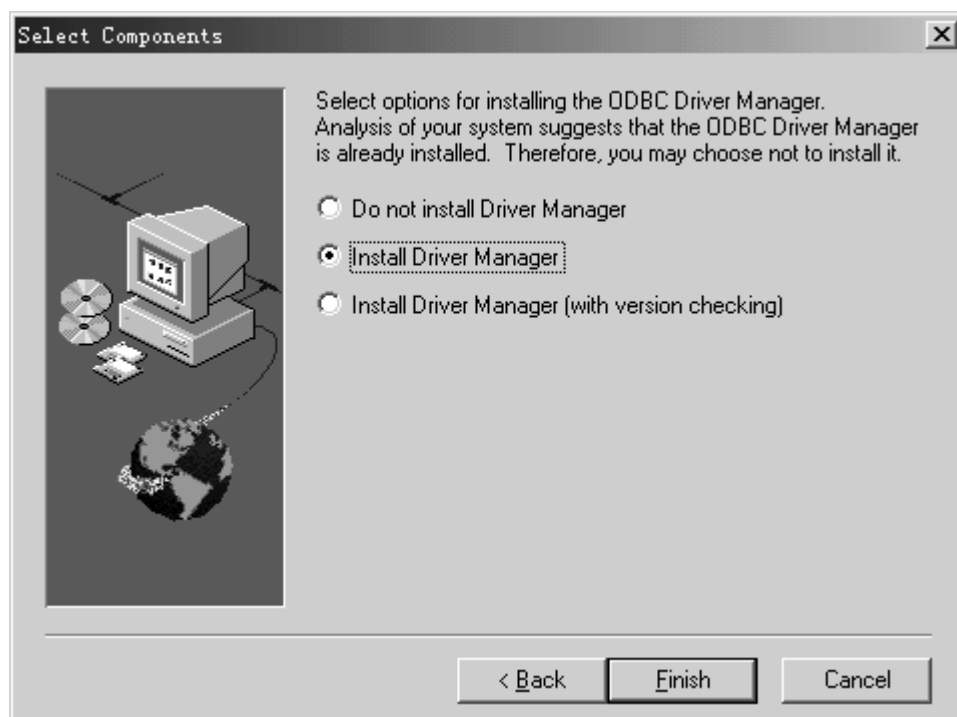


图 23-1 PostgreSQL ODBC 的安装

在这个窗口中，请选择第二项 (Install Driver Manager) 或第三项 (Install Driver Manager (with version checking))。安装的其他项目按照屏幕上的提示进行操作即可。安装完毕后，需要重新启动 Windows。重新启动后，在控制面板的“32 位 ODBC”中就可以看到 PostgreSQL 的 ODBC 驱动项目。

这个 ODBC 驱动程序提供对所有 PostgreSQL 标准数据类型的支持，包括 bool、int2、int4、float4、float8、date、time、abstime、datetime、timestamp、char、varchar 和 text。对其他数据类型提供部分支持，包括 point、circle、box 和 arrays，对它们的支持是通过字符串类型实现的。换句话说，对于这些非标准的数据类型，驱动程序总是以 SQL\_VARCHAR 类型返回数据，但可以像任何标准数据类型那样显示和更新。最终的表现结果可能因每个

应用和数据类型的不同而不同。

## 23.2 ODBC 的配置

安装完毕后，需要对 PostgreSQL 的 ODBC 驱动程序进行一些必要的配置才能使用。

进入控制面板，双击“32 位 ODBC”图标，首先配置“System DSN”，配置结果如图 23-2 所示。



图 23-2 System DSN 的配置

也可以在“User DSN”中配置 PostgreSQL 的 ODBC 驱动程序。“System DSN”与“User DSN”的主要区别在于允许访问的用户不同。

点击“Configure...”继续配置“System DSN”的属性，配置画面如图 23-3 所示。

其中“Database”必须是已存在的数据库；“Server”必须是可访问的主机名称；“User Name”和“Password”也必须是合法的 PostgreSQL 用户名和口令；“Port”的值取默认的“5432”；“Description”可以是任意的说明性字符串。

接下来配置 ODBC 的驱动程序特性，在图 23-3 所示的窗口中点击“Driver”按钮进入图 23-4 所示的画面。



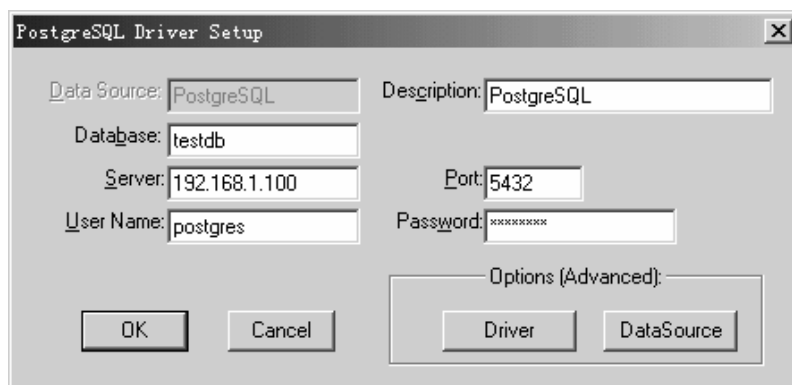


图 23-3 PostgreSQL ODBC 的属性配置

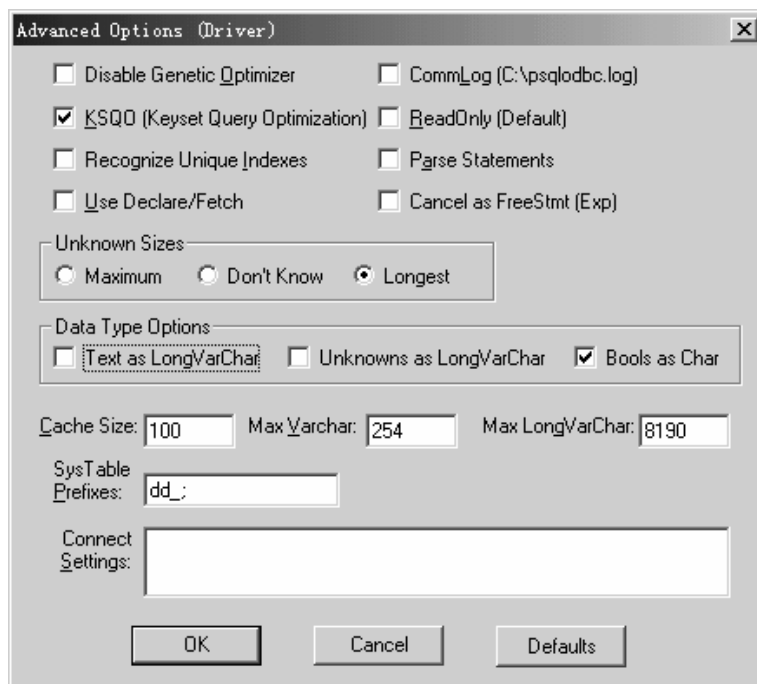


图 23-4 ODBC Driver 的配置

对于 Borland 公司的产品（如 Delphi 和 C++ Builder），应该禁选“Data Type Options”中的“Text as LongVarChar”，禁选“Unknowns as LongVarChar”。

需要注意的是，在 6.4 版本之前的 PostgreSQL 中，后端服务器在查询结果中不返回 varchar 或 char 数据类型的大小，而 Borland 公司的产品在进行简单查询和数据字典输入时非常依赖这个数据。因此，驱动程序设置了一些驱动器选项来弥补这个缺陷：

“Parse Statements”（分析语句选项）。驱动器分析 SQL 语句并且检索用于字段的字符，如 precision、nullability、aliases 等。如果使用的是 6.4 以前版本的 PostgreSQL，应该

选中这个选项。

“Unknown Sizes”(未知尺寸选项)。“longest”将返回基于结果集中最长数据行的数据精度。

在目前的版本中,如果选中了“Parse Statements”,在分析器不能处理某个字段时,它将退回到执行前的语句。因此,把“Unknown Sizes”设置为“longest”是比较安全的作法。

按“OK”键返回图23-3所示的窗口。

最后对数据源进行配置。在图23-3所示的窗口中按“DataSource”按钮,并按图23-5配置ODBC数据源。



图 23-5 ODBC 数据源的配置

如果选中了“ReadOnly”复选框,则只能通过 ODBC 驱动程序查询 PostgreSQL 数据库中的数据,不能插入、更新或删除数据。如果选中了“Show System Tables”复选框,则可以通过 ODBC 操作 PostgreSQL 的系统表(以“pg\_”开头的表)。如果选中了“Row Versioning”复选框,则能够使用 PostgreSQL 的多版本并发控制机制。

PostgreSQL 在不同的版本中使用了不同的通信协议(Protocol)。其中,6.2 及其以下版本使用一种协议,6.3 版使用另外一种协议,而 6.4 或更高版本所使用的协议也有所不同。必须根据所使用的 PostgreSQL 版本的不同选择不同的协议。例如,对于 7.x 版本的 PostgreSQL,可以使用“6.5/6.4”协议。在将来的版本中,PostgreSQL 的协议还可能被修改,请注意及时地在 PostgreSQL 的官方网站上了解协议的变化情况。

如果选中了“OID Option”中的“Show Column”复选框,那么 PostgreSQL 为每个表自动设置的 OID(对象标识号)将会以一个字段的形式返回在查询结果中。

至此,PostgreSQL ODBC 驱动程序的配置工作全部完成。

## 23.3 注意事项

在利用 ODBC 开发基于 PostgreSQL 的客户/服务器应用程序时，有以下事项需要特别注意。

### 1. 数据源

在指定 PostgreSQL ODBC 数据源时，需要提供以下参数：

- (1) 有效的和可解析的主机名，或 IP 地址。
- (2) 与 PostgreSQL 数据库服务器连接的有效 TCP/IP 端口号，默认的端口号为 5432。
- (3) 有效的并且存在的数据库名。
- (4) 有效的用户名。
- (5) 有效的口令。如果在 `pg_hba.conf` 将用户认证设置为口令认证，则必须提供有效的口令。

### 2. PostgreSQL 服务器端的配置

- (1) Postmaster 必须处于运行状态。
- (2) Postmaster 必须带有 `-i` 选项运行，这样 PostgreSQL 才允许远程连接。
- (3) `$PGPATH/data` 目录中的 `pg_hba.conf` 文件必须配置为允许特定的远程连接 PostgreSQL 数据库服务器。

### 3. 主键的访问

在某些情况下，在浏览带有主键的表的时候，PostgreSQL 后端服务器可能会返回一个内存耗尽的错误信息。

导致这个问题的主要原因是过多地使用了主键。在下面的例子中，一条查询语句使用了许多含主键 (`v1`、`v2`、`v3`) 的表达式：

```
select ... from foo where
  (v1 = "?" AND v2 = "?" AND v3 = "?") OR      -- line 1
  (v1 = "?" AND v2 = "?" AND v3 = "?") OR      -- line 2
  .....
  (v1 = "?" AND v2 = "?" AND v3 = "?") OR      -- line 9
  (v1 = "?" AND v2 = "?" AND v3 = "?");        -- line 10
```

其中，“？”是某些具体的数值。

在 6.4 版本以前的 PostgreSQL 中，这条查询语句存在执行问题。不过，在 6.4 版本中，有个补丁可用来避免问题的产生。这个补丁称为 KSQO (Keyset Query Optimization，键集

查询优化，见图 23-4)。对于 6.4 版本，可以打开这个驱动程序可选项。

#### 4. 大对象的使用

大对象在驱动程序中被映射为“LONG VARBINARY”，允许存储类似于 Microsoft Access 中的 OLE 对象，发送和接收这些对象进行多次 SQLPutData 和 SQLGetData 调用。驱动程序创建一个新的大对象后，简单地将它的标识符插入到相应的数据表中。不过，由于 PostgreSQL 使用对象标识号( OID)来标识大对象，因此有必要创建一个新的 PostgreSQL 类型来区分普通 OID 和大对象 OID。在这个新类型成为 PostgreSQL 官方版本的一部分之前，必须将它加到相应的数据库中，并且每次连接都要将它取出来。在驱动中所用的类型称为“lo”，下面是创建这个类型的命令：

```
create type lo (  
    internallength=4, externallength=10,  
    input=int4in, output=int4out,  
    default='', passedbyvalue  
);
```

该类型的使用方法类似于：

```
create table employee (  
    id integer,  
    name varchar(30),  
    picture lo  
);
```

一旦创建了专用于大对象的 OID 类型，只需要使用“lo”类型定义大对象字段即可。当驱动程序遇到一个“lo”类型时，它将把该类型当作 SQL\_LONGVARBINARY 来处理。

但是，这个新类型缺少一些必要的功能。例如，这个类型在完成更新或删除操作后不进行表的清理工作，因此会留下一些“垃圾”数据，占用一些磁盘空间。目前版本的 PostgreSQL 不支持对大对象的清理。

为了解决这个问题，用户可以自行编写一个在服务器间歇期运行的清理程序。基本实现原理是：查找 pg\_attribute 表中的 lo 数据类型，然后通过查询所有包含 lo 的表建立一个 lo 字段列表。最后将这个列表与 pg\_class 中的 xinv.\*进行对比，没有指针的 xinv.\*就是应该删除掉的“垃圾”数据。

也许在将来的版本中，大对象类型会成为 PostgreSQL 的基本类型。

## 23.4 ODBC 的使用

本节通过一个例子，介绍在 Delphi 中通过 ODBC 访问 PostgreSQL 表 friend 的方法。操作步骤是：

- (1) 进入 Delphi，创建一个新的应用程序。
- (2) 在主表单 Form1 中加入 TTable 类型的组件 Table1。将 Table1 的属性 DatabaseName 的值设置为 PostgreSQL，这是 PostgreSQL ODBC 驱动程序的名字。将属性 TableName 的值设置为 friend。在 Form1 中双击 Table1 图标，加入所有的字段。将 Table1 的 Active 属性设置为 true。
- (3) 在 Form1 中加入 TDataSource 类型的组件 DataSource1，并将它的属性 DataSet 设置为 Table1。
- (4) 在 Form1 中加入 TDBGrid 类型的组件 DBGrid1，并将它的 DataSource 属性设置为 DataSource1。双击这个组件的图标，编辑这个组件的栏目。
- (5) 保存应用程序。
- (6) 编译并运行应用程序，结果如图 23-6 所示。



图 23-6 ODBC 的使用

## 第24章

# 命令与工具

---

本章分别按照字母顺序给出SQL命令、PostgreSQL系统程序和工具命令的详细说明。

SQL 命令

系统程序和工具



## 24.1 SQL 命令

### 1. ABORT（回滚当前事务）

语法: `ABORT [ WORK | TRANSACTION ]`

描述: 本命令回滚当前事务并且废弃当前事务中所作的所有更新。此命令是 PostgreSQL 基于历史原因所作的扩展, ROLLBACK 是本命令在 SQL92 中的等价命令。为了兼容于其他关系数据库系统, 建议使用 ROLLBACK。

用 COMMIT 命令提交并结束一个事务。

输入: 无。

输出: 如果回滚成功, 命令返回信息 “ROLLBACK”; 否则返回信息 “NOTICE: ROLLBACK: no transaction in progress”, 表明当前进程中没有任何事务存在。

用法:

--回滚当前事务, 即取消自上次事务提交以来对数据库所作的任何更新。

```
ABORT WORK;
```

```
ROLLBACK
```

兼容性: 兼容于 SQL92 中的 ROLLBACK 命令。

### 2. ALTER GROUP（更改组属性）

语法:

```
ALTER GROUP name ADD USER username [, ... ]
```

```
ALTER GROUP name DROP USER username [, ... ]
```

描述: 本命令用于向组中增加用户或者从组中删除用户。只有数据库超级用户才能使用这条命令。向组中增加用户并不创建新用户, 同样从组中删除用户也不删除用户本身。

使用 CREATE GROUP 命令创建新组, 使用 DROP GROUP 命令删除一个组。

输入:

name——要更改的组名称。

username——向组中增加或从组中删除的用户名, 该用户名必须已经存在。

输出:

如果更改成功, 则返回信息 “ALTER GROUP”。

用法:

--向组中增加用户:

```
ALTER GROUP staff ADD USER karl, john
```

--从组中删除用户：

```
ALTER GROUP workers DROP USER beth
```

兼容性：SQL92 里没有 ALTER GROUP 命令，在 SQL92 中，角色（roles）的概念类似于 PostgreSQL 中的用户。

### 3. ALTER TABLE（更改表属性）

语法：

```
ALTER TABLE table [ * ]
```

```
    ADD [ COLUMN ] column type
```

```
ALTER TABLE table [ * ]
```

```
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }
```

```
ALTER TABLE table [ * ]
```

```
    RENAME [ COLUMN ] column TO newcolumn
```

```
ALTER TABLE table
```

```
    RENAME TO newtable
```

```
ALTER TABLE table
```

```
    ADD table constraint definition
```

描述：本命令更改一个表的定义。其中，ADD COLUMN 用于向表增加一个新的字段；ALTER COLUMN 用于修改或删除字段的默认值，新的默认值只对新插入的数据行有效；RENAME 用于重新命名字段，这个操作不影响相关表中的任何数据，表或字段在此命令执行后仍将保持相同的尺寸和类型；ADD table constraint definition 子句使用与 CREATE TABLE 相同的语法，向表增加一个新的约束。

只有表的所有者，才能使用本命令。任何用户都不能更改系统表结构的任何部分。

如果在表的名称之后加上一个“\*”号，该命令将作用于该表以及所有继承级别低于该表的表。默认时，该命令不影响任何子表或修改任何子表的相关名称。

在目前的版本中，新字段的默认值和约束子句会被忽略。可以用 ALTER TABLE 命令中的 SET DEFAULT 子句设置默认值，并且还必须用 UPDATE 将已存在的行更新为默认值。

在目前的版本中，只有外键(FOREIGN KEY)约束才可以增加到表中。要创建或者删除一个唯一性约束，可以先创建一个唯一性索引（参阅 CREATE INDEX）。要想增加检查（check）约束，需要重建和重载该表。

输入：

table——要修改的表名称。

column——要修改的字段名称。

type——新的字段类型。

newcolumn——现有字段的新名称。

newtable——现有表的新名称。

table constraint definition——表的新约束定义。

输出：

如果修改成功，返回信息“ALTER”；否则返回信息“ERROR”。



用法:

--向表中增加一个 VARCHAR 列:

```
ALTER TABLE distributors ADD COLUMN address VARCHAR(30);
```

--对现有列改名:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

--对现有表改名:

```
ALTER TABLE distributors RENAME TO suppliers;
```

--向表中增加一个外键约束:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address)
REFERENCES addresses(address) MATCH FULL
```

兼容性: 除了上面提到的默认值和约束外, ADD COLUMN 子句与 SQL92 中相应的命令兼容; 而 ALTER COLUMN 子句则完全兼容于 SQL92。

SQL92 对 ALTER TABLE 命令规定了一些附加的、Postgres 目前还不直接支持的功能, 例如:

ALTER TABLE table DROP CONSTRAINT constraint { RESTRICT | CASCADE }。增加或删除表的约束(如检查约束、唯一性约束或外键约束)。要创建或删除一个唯一性约束, 必须相应地创建或删除一个唯一性索引; 要修改其他类型的约束, 需要重建和重载该表, 并使用与 CREATE TABLE 命令相同的其他参数。

例如, 删除表 distributors 的任何约束:

```
CREATE TABLE temp AS SELECT * FROM distributors;
```

```
DROP TABLE distributors;
```

```
CREATE TABLE distributors AS SELECT * FROM temp;
```

```
DROP TABLE temp;
```

```
ALTER TABLE table DROP [ COLUMN ] column { RESTRICT | CASCADE }
```

要删除一个现有的字段, 表必须重新创建和重新装载:

```
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;
```

重命名字段的名称和表的名称是 PostgreSQL 扩展, SQL92 不提供这些功能。

## 4. ALTER USER (更改用户属性)

语法: ALTER USER username

[ WITH PASSWORD 'password' ]

[ CREATEDB | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]

[ VALID UNTIL 'abstime' ]

描述：本命令用于更改 PostgreSQL 用户的属性。只有数据库超级用户才能使用这个命令更改用户的权限和口令有效期。普通用户只能更改他们自己的口令。

使用 CREATE USER 创建新用户；使用 DROP USER 删除用户。

输入：

username——用户名。

password——新口令。

CREATEDB, NOCREATEDB——这个子句定义该用户创建数据库的能力。如果指定了 CREATEDB，该用户可以创建自己的数据库。NOCREATEDB 将剥夺一个用户创建数据库的权力。

CREATEUSER, NOCREATEUSER——这个子句决定一个用户能否创建新用户。这个选项同样会使该用户成为超级用户，可以超越所有访问限制。

abstime——该用户帐号口令的有效日期（和可选的时间）。

输出：如果更改成功，则返回信息“ALTER USER”；否则，返回信息“ERROR: ALTER USER: user "username" does not exist”，表明指定的用户不存在。

用法：

--更改用户口令

```
ALTER USER davide WITH PASSWORD 'hu8jmn3';
```

--更改用户有效期

```
ALTER USER manuel VALID UNTIL 'Jan 31 2030';
```

--更改用户有效期，指定其权限应该在用比 UTC 早一小时的时区记时的 1998 年 5 月 4 日正午失效

```
ALTER USER chris VALID UNTIL 'May 4 12:00:00 1998 +1';
```

--赋予一用户创建新用户和新数据库的权限。

```
ALTER USER miriam CREATEUSER CREATEDB;
```

兼容性：SQL92 中没有 ALTER USER 命令，该标准将用户定义部分留给具体数据库实现处理。

## 5. BEGIN（开始一个新事务）

语法：BEGIN [ WORK | TRANSACTION ]

描述：默认时，PostgreSQL 以非链接模式（unchained mode）执行一个事务，在其他数据库系统中，这种模式被称之为“自动提交”（autocommit）。也就是说，每个用户命令都是在其自身的事务中运行，在命令结束时隐含地调用一个提交（commit）命令，如果命令执行成功则提交，否则调用一个回滚命令。BEGIN 以链接模式（chained mode）初始化一个用户事务，也就是说所有 BEGIN 命令之后的用户命令都将在一个事务中执行，直到显式地调用 COMMIT、ROLLBACK 命令或退出。很明显，在链接模式下命令的执行速度要快得多，因为事务开始和提交（start、commit）需要占用大量的 CPU 时间和进行大量的磁盘操作。在一个事务内部执行多条命令时，可能修改若干个相关的表，因此同样需要保持一致性。

在 PostgreSQL 中，默认的事务隔离级别是 READ COMMITTED，这时在事务内部所

进行的查询，只能看到查询提交之前的数据。所以，如果需要更严格的事务隔离，必须在 BEGIN 后立即使用 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE。在 SERIALIZABLE 模式下，查询将只能看到整个事务开始之前的修改（实际上是在一个可串行化事务内部第一个数据库管理命令执行前的数据）。

如果提交了事务，PostgreSQL 将保证要么实现所有更新，要么所有更新都不实现，即保证事务的 ACID（原子性、一致性、隔离性、持续性，atomic、consistent、isolatable、durable）属性。

请参考 LOCK 命令获取关于在事务内部锁定一个表的详细信息。

使用 COMMIT 或 ROLLBACK 结束一个事务。

输入：

WORK, TRANSACTION——是可选的关键字，它们没有什么作用。

输出：如果能够正确地启动一个新的事务，命令返回信息“BEGIN”；否则，返回信息“NOTICE: BEGIN: already a transaction in progress”，表明该事务已经运行，当前事务没有被影响。

用法：

--开始一个用户事务：

```
BEGIN WORK;
```

兼容性：BEGIN 是 PostgreSQL 所作的扩展。在 SQL92 中没有显式的 BEGIN 命令，事务初始化总是隐含地进行，而且命令 COMMIT 或 ROLLBACK 导致事务的终止。

注意：许多关系型数据库都提供一个自动提交（autocommit）特性。

BEGIN 关键字在嵌入 SQL 中有多种用途，因此在移植数据库应用时，应该仔细检查事务的语意。SQL92 还要求事务的默认隔离级别为 SERIALIZABLE。

## 6. CLOSE（关闭游标）

语法：CLOSE cursor

描述：本命令释放与一个游标关联的资源。一个游标关闭后，不允许对其再进行任何操作。一个不再使用的游标应该及时关闭掉。

如果用 COMMIT 或 ROLLBACK 终止了一个事务，则每个处于打开状态的游标将自动关闭。

值得注意的是，PostgreSQL 没有显式的 OPEN（打开游标）命令，一个游标在定义时自动打开。使用 DECLARE 命令定义一个游标。

输入：

cursor——待关闭的游标名字。

输出：如果游标关闭成功，则返回信息“CLOSE”；否则，返回信息“NOTICE PerformPortalClose: portal "cursor" not found”，表明该游标没有定义或已经关闭。

用法：

--关闭游标 liahona：

```
CLOSE liahona;
```

兼容性：与 SQL92 完全兼容。

## 7. CLUSTER (建立集簇)

语法: CLUSTER indexname ON table

描述: 本命令指示 PostgreSQL 近似地基于索引 indexname 的度量对表 table 建立存储集簇。在此命令执行前, 必须保证该表已经建立了索引。

为一个表建立了集簇后, 该表的物理存储将基于索引信息进行。集是静态建立的, 也就是说, 当表的内容被更新后, 改变的内容不会建立集簇。同样不会对更新的记录重新建立集簇, 如果需要, 可以通过手工执行本命令的方法重新建立集簇。

值得注意的是, 一个建立了集簇的表的内容实际上按索引顺序复制到了一个临时表中, 然后重新改成原名。因此, 在建立集簇时所有赋予的权限和其他索引都将丢失。

如果只是随机地访问表中的行, 那么表数据的实际存储顺序是无关紧要的。但是, 如果对某些数据的访问多于其他数据, 而且有一个索引将这些数据分组, 那么就会从集簇中获益 (主要是指访问速度)。

另一个发挥集簇作用的例子是当用索引从一个表中取出几个记录时。如果在一个表中计算一定索引范围的值, 或者是一个索引过的值对应多行时, 集簇也会有助于应用。因为如果索引标识出第一匹配行所在的堆存储页, 所有其他行也可能已经在同一组存储页中, 这样便节省了磁盘访问的时间, 加速了查询。

有两种建立集簇的方法。第一种是用 CLUSTER 命令, 此命令将原表按索引重新排列。这个动作在操作大表时可能会很慢, 因为每一行都必须从存储页中按索引顺序取出。如果存储页表没有排序, 整个表随机地存放在各个页面, 每行都要进行依次磁盘页面操作。虽然 PostgreSQL 有一个缓冲, 但一个大表的主体是不可能全部都放到缓冲中去的。

另一个对数据建立集簇的方法是使用下面的命令:

```
SELECT columnlist INTO TABLE newtable
FROM table ORDER BY columnlist
```

这种用法使用 PostgreSQL 的排序子句 ORDER BY 来匹配索引, 在对未排序的数据进行操作时速度要快得多。然后删除旧表, 用 ALTER TABLE/RENAME 将 temp 改成旧表名, 并且重建所有索引。唯一的问题是 OID 将不能被保留。这时再建立集簇要快得多, 因为大多数堆栈数据已经排过序而且使用了现有的索引。

输入:

indexname——索引的名称。

table——表的名称。

输出: 如果建簇成功, 则返回信息 “CLUSTER”; 否则, 返回信息 “ERROR: relation <tablrelation\_number> inherits "table"”, 或 “ERROR: Relation table does not exist!”, 表明该表 tablrelation\_number 是从表 table 继承来的, 或表 table 根本不存在。

用法:

--以雇员的薪水属性对雇员关系建簇

```
CLUSTER emp_ind ON emp;
```

兼容性: SQL92 规范中没有 CLUSTER 命令。

## 8. COMMENT（定义对象注释）

语法：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'

描述：本命令为一个对象增加注释，这种注释可以很容易地用 psql 的 \dd 命令检索出来。要删除一个注释，使用 NULL 作为注释正文。当删除对象时，注释自动被删除。

输入：

object\_name、table\_name、column\_name、agg\_name、func\_name、op、trigger\_name —— 注释的对象名称。

text——注释正文。

输出：COMMENT，表明成功地为对象增加了注释。

用法：

--给表 mytable 增加注释

```
COMMENT ON mytable IS 'This is my table.';
```

--其他一些例子

```
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 float8 IS 'Computes sample variance';
```

```
COMMENT ON FUNCTION my_function (datetime) 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 规范中没有 COMMENT 命令。

## 9. COMMIT（提交当前事务）

语法：COMMIT [ WORK | TRANSACTION ]

描述：本命令用于提交当前事务。成功提交的事务所作的任何更改对其他事务而言都是可见的，而且能保证崩溃发生时的可持续性。

关键字 WORK 和 TRANSACTION 都可以忽略。

使用 ROLLBACK 命令回滚一个事务。

输入：

WORK, TRANSACTION——可选关键字。

输出：如果事务提交成功，则返回信息“ COMMIT ”；否则，返回信息“ NOTICE: COMMIT: no transaction in progress ”，表明当前过程中没有事务可提交。

用法：

--让所有变更永久化

COMMIT WORK;

兼容性：SQL92 只定义了两种形式的提交命令：COMMIT 和 COMMIT WORK。

## 10. COPY（导出/导入表数据）

语法：

```
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' ]
```

描述：本命令用于实现 PostgreSQL 表和标准 UNIX 文件之间交换数据。COPY 指示 PostgreSQL 后端直接从文件中读写数据。该文件必须为后端可见，而且文件名必须从后端的角度指定。如果指定的文件是 stdin 或 stdout，数据通过客户前端流到后端。

BINARY 关键字将强制使用二进制对象而不是文本存储或读取所有数据。这样做在一定程度上比传统的复制命令快，但移植性不是很好，而且生成的文件也较大。

默认情况下，复制命令使用 tab ( "\t" ) 字符作为分隔符。可以使用关键字 USING DELIMITERS 将分隔符改成任何其他的字符。如果数据中存在与分隔符相同的字符，则自动用引号将其区别开来。

对任何要复制出来的数据，命令的使用者必须有 SELECT 权限；对任何要复制到表中的数据，命令的使用者必须有 INSERT 和 UPDATE 权限。使用 COPY 命令时，后端同样需要对文件操作的合法的 UNIX 权限。

关键字 USING DELIMITERS 定义一个用于分隔字段的分隔字符。如果在分隔符字符串中定义了多个字符，则只使用第一个字符。

不要将 COPY 和 psql 的命令 \copy 混淆。

COPY 不会激活规则，也不会处理字段默认值，不过它会激活触发器。

COPY 会在第一个错误处停下来。这些在 COPY FROM 中不应该导致问题，但在 COPY

TO 时目标表会被部分地改变。因此,应该在一次失败的复制后用 VACUUM 命令进行一些清除工作。

因为 PostgreSQL 的后端工作目录通常和用户的工作目录不一样,本地用户使用一个文件 "foo" (没有附加的路径信息) 可能会产生不可预见的结果。这时,foo 将生成成为 \$PGDATA/foo。通常,指定文件时要加上相对后端服务器的完全路径。

不管是在本地硬盘还是在网络文件系统上,作为 COPY 参数定义的文件名必须是数据库服务器可访问的地方。

如果使用了一个从一台机器到另一台机器的 TCP/IP 连接,而且定义了目标文件,那么目标文件将会写到后端运行的机器上,而不是用户的机器上。

文件格式: 当不带 BINARY 选项使用 COPY TO 命令时,生成的文件每条记录占一行,每个字段用分隔符分开。内嵌的分隔符字符由一个反斜杠 ("\") 开头。字段值本身是利用与该字段类型相关的输出函数生成的字符串。某一类型的输出函数本身不应该生成反斜杠,这个任务由 COPY 本身完成。

每条记录的实际格式是:

```
<attr1><separator><attr2><separator>.....<separator><attrn><newline>
```

如果使用了 WITH OIDST 可选项,OID 将被放在每行的开头。

如果 COPY 将它的输出送到标准输出而不是一个文件,在复制结束时,它会在一个新行上输出一个反斜杠 ("\") 和一个句点 ("."),最后以一个换行符作为结束符。类似地,如果 COPY 从标准输入读入数据,它将行首由一个反斜杠 ("\") 一个句号 (".") 以及一个换行符组成的三个连续字符作为文件结束符。不过,如果在这三个字符组合之前碰到一个真正的 EOF (文件结束符),COPY 命令也会结束。

数据中的斜杠有其他含义。NULL 属性输出为"\N"。一个反斜杠字符输出成两个连续的反斜杠 ("\\")。一个 tab 字符用一个反斜杠后面跟一个 tab 代表。一个新行字符用一个反斜杠和一个新行代表。当装载不是由 PostgreSQL 生成的文件时,需要将反斜杠字符 ("\") 转换成双反斜杠 ("\\") 以保证正确装载。

二进制格式: 当使用 COPY BINARY 时,文件的头四个字节是文件中的记录数目。如果数值是零,COPY BINARY 命令将一直读到文件尾;否则,它将在达到这个数目时停止读取,文件中剩余的数据将被忽略。

文件中每一个实例的格式如表 24-1 所示,无符号的四字节整数数量在表中记为 uint32。

表24-1 二进制文间格式

文件开始	
uint32	记录个数
每条记录	
uint32	记录数据总长度
uint32	oid (如果存在)
uint32	null 字段的个数
[uint32,.....,uint32]	字段个数 (attribute numbers of attributes), 从 0 开始
-	<字段数据>

二进制数据的对齐规则是：在 Sun-3s 机器上，2 字节字段以 2 字节为界对齐，而所有整数字段以 4 - 字节为界对齐。字符字段以 1 字节为界对齐。在大部分机器上，所有大于 1 字节的整数是按照 4 - 字节为边界对齐的。注意，变长字段的字段长度在前，数组只是简单的数组元素类型的连续数据流。

输入：

BINARY——改变字段格式属性，强制所有数据都使用二进制格式存储和读取。

table——表的名称。

WITH OIDS——复制每行的内部唯一对象标识号 (OID)。

filename——输入或输出的 UNIX 文件的绝对路径 (文件) 名。

stdin——指定输入是来自管道还是终端。

stdout——指定输出是进入管道还是终端。

delimiter——用于分隔输入或输出的字段的分隔符。

null print——一个代表 NULL 值的字串。因历史原因，默认是“\N”(反斜杠-N)。值得注意的是，对于入复制 (copy in)，任何匹配这个字串的字串将被存储为 NULL 值，所以应该确保所用的字符串和复制出 (copy out) 相同。

输出：如果复制成功，则返回信息“COPY”；否则，返回信息“ERROR: reason.....”，指明复制失败的原因。

用法：

--下面的例子将一个表复制到标准输出，使用竖直线 (|) 作为域分隔符：

```
COPY country TO stdout USING DELIMITERS '|' ;
```

--从一个 UNIX 文件中复制数据到表 "country"：

```
COPY country FROM '/usr1/proj/bray/sql/country_data' ;
```

--下面是一些可以从标准输入 stdin 输入数据的例子 (在最后有结束符)：

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
...
ZM      ZAMBIA
ZW      ZIMBABWE
\.
```

--同样的数据，输出到一个 Linux/i586 机器的二进制文件中。数据是用 UNIX 应用 od -c 显示的。表里有三个域：第一个是 char(2)，第二个是 text，所有记录在第三字段有空 (null) 值。注意 char(2) 字段是如何用空 (null) 补齐成四个字节的以及 text 字段是如何在前面补长度的：

```
355  \0 \0 \0 027 \0 \0 \0 001 \0 \0 \0 002 \0 \0 \0
006  \0 \0 \0  A  F  \0 \0 017 \0 \0 \0  A  F  G  H
    A  N  I  S  T  A  N 023 \0 \0 \0 001 \0 \0 \0 002
    \0 \0 \0 006 \0 \0 \0  A  L  \0 \0 \v \0 \0 \0  A
    L  B  A  N  I  A 023 \0 \0 \0 001 \0 \0 \0 002 \0
```



```

\0 \0 006 \0 \0 \0 D Z \0 \0 \v \0 \0 \0 A L
G E R I A
...\n \0 \0 \0 Z A M B I A 024 \0
\0 \0 001 \0 \0 \0 002 \0 \0 \0 006 \0 \0 \0 Z W
\0 \0 \f \0 \0 \0 Z I M B A B W E

```

兼容性：SQL92 中没有 COPY 命令。

## 11. CREATE AGGREGATE (定义聚集函数)

语法：CREATE AGGREGATE name ( BASETYPE = input\_data\_type  
 [ , SFUNC1 = sfunc1, STYPE1 = state1\_type ]  
 [ , SFUNC2 = sfunc2, STYPE2 = state2\_type ]  
 [ , FINALFUNC = ffunc ]  
 [ , INITCOND1 = initial\_condition1 ]  
 [ , INITCOND2 = initial\_condition2 ] )

描述：本命令允许用户或程序员通过定义新的聚集函数来扩展 PostgreSQL 的功能。一些用于基本类型的聚集函数如 min(int4) 和 avg(float8) 等已经包含在 PostgreSQL 中。如果需要定义一个新类型或需要一个 PostgreSQL 未提供的聚集函数，这时可用 CREATE AGGREGATE 来创建所需要的聚集函数。

一个聚集函数是用它的名字和输入数据类型来标识的。如果两个聚集的输入数据不同，它们可以有相同的名字。为了避免冲突，不要编写一个与聚集同名而且输入函数也相同的普通函数。

一个聚集函数是由一到三个普通函数组成的：两个状态转换函数 sfunc1、sfunc2 以及一个最终计算函数 ffunc。它们是这样使用的：

```

sfunc1(internal-state1, next-data-item) ---> next-internal-state1
sfunc2(internal-state2)                ---> next-internal-state2
ffunc(internal-state1, internal-state2) ---> aggregate-value

```

PostgreSQL 创建一个或两个临时变量（数据类型是 stype1 和 stype2），用于保存转换函数参数的中间结果。对于每个输入数据条目，都调用状态转换函数计算内部状态值的新数值。在处理完所有数据后，调用一次最终处理函数以计算聚集函数的输出值。

如果定义了两个转换函数，那么还必须定义 ffunc。如果只定义一个转换函数，那么 ffunc 是可选的。在没有提供 ffunc 时的默认动作是返回所使用的内部状态值的最后值，这时，聚集函数的输出类型与状态值的类型相同。

一个聚集函数还可能提供一到两个初始条件，也就是所用的内部状态值的初始值。这些值是以类型为 text 的数据域存储在数据库中的，不过它们必须是状态值数据类型的有效外部表现形式的常量。如果定义了 sfunc1 而没有定义 initcond1 的值，那么系统不会对第一个输入项目调用 sfunc1；而是用第一个输入项目初始化内部状态值 1，从第二个输入项目的开始调用 sfunc1。这像 MIN 和 MAX 之类的函数是很有用的。请注意，一个使用这种特性的聚集函数在没有输入值时将返回 NULL。对状态值 2 没有类似的处理，如果定义了 sfunc2，那么必须有一个 initcond2。

使用 DROP AGGREGATE 删除聚集函数。

CREATE AGGREGATE 的参数可以以任何顺序指定，而不必遵循上面规定的顺序。

聚集函数可能由各种各样不同的状态和最终处理函数组成。例如，count 聚集函数需要 Ssfunc2（一个递增函数）但不需要 sfunc1 或 ffunc；sum 聚集函数需要 sfunc1（一个累加函数）但不需要 sfunc2 或 ffunc，而 avg 聚集函数需要上面所有状态函数和一个 ffunc（一个除法函数）来计算结果。在任何情况下，至少要定义一个状态函数，而且任何 sfunc2 都有一个对应的 initcond2。

输入：

name——要创建的聚集函数名。

input\_data\_type——本聚集函数要处理的基本数据类型。

sfunc1——用于处理源数据列中的每一个非空（non-NULL）数据的状态转换函数。它必须是一个带两个参数的函数，第一个参数的类型是 state1\_type，第二个参数的类型是 input\_data\_type。此函数必须返回一个类型为 state1\_type 的值。这个函数接受当前状态值 1 和当前输入数据条目，返回下个状态值 1。

state1\_type——聚集函数的第一个状态值的数据类型。

sfunc2——用于处理源数据列里的每一个非空（non-NULL）数据的状态转换函数。它是一个单个参数的函数，参数的类型是 state2\_type，返回相同的类型。这个函数接受当前状态值 2 和当前输入数据条目，而返回下个状态值 2。

state2\_type——聚集函数的第二个状态值的类型。

ffunc——在转换完所有输入字段后调用的最终处理函数。如果两个状态值都使用了，最终处理函数必须以两个类型分别为 state1\_type 和 state2\_type 的数作为参数。如果只使用了一个状态值，最终处理函数必须以一个该状态值类型的数作为参数。聚集函数的输出数据类型被定义为此函数的返回类型。

initial\_condition1——状态值 1 的初始值。

initial\_condition2——状态值 2 的初始值。

输出：如果命令执行成功，则返回信息“CREATE”。

用法：

请参考 PostgreSQL 程序员手册中聚集函数章节的聚集函数部分以获取完整的例子。

兼容性：CREATE AGGREGATE 是 PostgreSQL 的扩展，在 SQL92 中没有 CREATE AGGREGATE 命令。

## 12. CREATE CONSTRAINT TRIGGER（创建约束触发器）

语法：CREATE CONSTRAINT TRIGGER name

AFTER events ON

relation constraint attributes

FOR EACH ROW EXECUTE PROCEDURE func '(' args ')'

描述：本命令被 CREATE/ALTER TABLE 内部使用以及被 pg\_dump 用于创建特殊的用于参考完整性的触发器，一般用户不使用本命令。

输入:

name——约束触发器的名称。

events——触发该触发器的事件范围。

relation——被触发关系的表名称。

constraint——实际的约束。

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: 成功创建后的返回信息为 “ CREATE CONSTRAINT ”。

## 13. CREATE DATABASE (创建新数据库)

语法: CREATE DATABASE name [ WITH LOCATION = 'dbpath' ]

描述: 本命令用于创建一个新的 PostgreSQL 数据库, 创建者即为新数据库的管理员。

可以为数据库选择一个存放位置, 例如, 在另一个硬盘上存放数据库。其中的路径必须是事先用 initlocation 准备好的路径。如果路径包含斜杠, 那么斜杠前面的部分被解释成一个环境变量, 该变量必须为服务进程所知。这样, 数据库管理员可以对能够在那里创建的数据库进行控制。如果服务器被编译成带有 ALLOW\_ABSOLUTE\_DBPATHS (默认时没有) 选项, 以斜杠开头为标识的绝对路径 (如 /usr/local/pgsql/data) 同样也允许。

在用绝对路径指定的可选数据库位置时, 有一些安全和数据完整性的问题, 而且默认时只有被后端识别的环境变量才可以指定为可选的路径。

CREATE DATABASE 是 PostgreSQL 语言的扩展。

使用 drop\_database 删除一个数据库。

程序 createdb 是这个命令的 shell 脚本的封装, 使用起来更方便。

输入:

name——要创建的数据库名。

dbpath——在文件系统中存储新数据库的可选位置。

输出:

如果数据库创建成功, 命令返回信息 “ CREATE DATABASE ”, 否则返回下列错误信息之一:

“ ERROR: user 'username' is not allowed to create/drop databases ”, 当前用户没有创建用户的权限。

“ ERROR: createdb: database "name" already exists ”, 待创建的数据库已经存在。

“ ERROR: Single quotes are not allowed in database names. ”, “ ERROR: Single quotes are not allowed in database paths. ”, 数据库名 name 和 dbpath 不能包含单引号。这样要求是为了创建数据库目录的 shell 命令能够正确执行。

“ ERROR: The path 'xxx' is invalid. ”, 对指定的 dbpath 扩展失败。这时应该检查输入的路径, 或者检查引用的环境变量是否存在。

“ ERROR: createdb: May not be called in a transaction block. ”, 如果有一个显式的事务块正在处理, 不能调用 CREATE DATABASE, 必须先结束事务。

“ ERROR: Unable to create database directory 'path'. ”, “ ERROR: Could not initialize

database directory.”，导致这类错误的最典型原因是对数据目录进行操作的权限不够，磁盘已满或其他文件系统问题。

用法：

--创建一个新数据库

```
create database lusiadas;
```

--在另一个地方创建新数据库

```
mkdir private_db
```

```
initlocation ~/private_db
```

```
Creating PostgreSQL database system directory /home/olly/private_db/base
```

```
psql olly
```

```
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
```

```
CREATE DATABASE elsewhere WITH LOCATION = '/home/olly/private_db';
```

```
CREATE DATABASE
```

兼容性：SQL92 中没有 CREATE DATABASE 命令。

#### 14. CREATE FUNCTION（定义函数）

语法：

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
```

```
RETURNS rtype
```

```
AS definition
```

```
LANGUAGE 'langname'
```

```
[ WITH ( attribute [, ...] ) ]
```

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
```

```
RETURNS rtype
```

```
AS obj_file , link_symbol
```

```
LANGUAGE 'C'
```

```
[ WITH ( attribute [, ...] ) ]
```

描述：本命令允许一个 PostgreSQL 用户在一个数据库中定义一个新函数，这个用户将被看作是这个函数的所有者。

PostgreSQL 允许函数“重载”，也就是说，同一个函数名可以用于几个不同的函数，只要它们的参数彼此不同即可。不过，这个功能在用于 INTERNAL 和 C 语言的函数时要小心。

输入参数和返回值可以使用所有 SQL92 的数据类型。不过,有些类型指定的细节(例如,numeric 类型的精度域)是由下面的函数负责的,并且不能被 CREATE FUNCTION 识别。

两个内部函数拥有相同 C 语言名称时链接时肯定会发生错误。要解决这个问题,必须赋予它们不同的 C 语言名称(例如,用参数类型作为 C 语言名称的一部分),然后在 CREATE FUNCTION 的 AS 子句中指定这些名字。如果 AS 子句为空,那么 CREATE FUNCTION 假设函数的 C 语言名称与 SQL 名称一样。

如果用 C 语言函数重载 SQL 函数,必须为每个 C 语言函数的实例定义一个独立的名称,并且使用 CREATE FUNCTION 中的 AS 子句的不同形式,来确保重载的 SQL 函数名称能正确地解释为相应的动态链接对象。

输入:

name——要创建的函数名。

ftype——函数参数的数据类型。输入参数的类型可以是基本类型、组合类型或者 opaque。opaque 表明该函数可以接受一个非法的、类似于 char \* 之类的类型。

rtype——返回数据类型。输入参数的类型可以是基本类型、组合类型、setof type 或者 opaque。setof 表示该函数将返回一组目标,而不是单条目标。

attribute——一个关于函数的可选信息,主要用于优化。目前唯一支持的属性是 iscachable。iscachable 表示此函数在输入相同时总是返回相同的值,也就是说,它不进行数据库查找或者使用没有直接在它的参数列表出现的信息。优化器使用 iscachable 来确认对该函数的调用进行预先计算是否安全。

definition——一个定义函数的字符串,其含义取决于所使用的语言。可以是一个内部函数名、一个指向目标文件的路径、一个 SQL 查询或者一种过程语言文本。

obj\_file, link\_symbol——这种形式的 AS 子句用于动态链接的 C 语言函数,这时该函数在 C 源代码中的名称和 SQL 函数的名称不同。字符串 obj\_file 是含有可动态装载对象的文件名,而 link\_symbol 是对象的链接符号,这个符号与 C 源代码中的函数名相同。

langname——可以是 'C'、'sql'、'internal' 或 'plname',这里的'plname'是所创建过程的语言名,参考 CREATE LANGUAGE 获取详细信息。

输出: 如果命令成功地执行,则返回信息 "CREATE"。

用法:

--创建一个简单的 SQL 函数

```
CREATE FUNCTION one() RETURNS int4
AS 'SELECT 1 AS RESULT'
LANGUAGE 'sql';
SELECT one() AS answer;
```

answer

-----

1

--下面的例子通过调用一个用户创建的共享库路径创建一个 C 函数。该路径计算一个检测位,如果

函数参数里的检测位正确就返回一个 TRUE。这些是通过使用一个 CHECK 约束实现的。

```
CREATE FUNCTION ean_checkdigit(bpchar, bpchar) RETURNS bool
AS '/usr1/proj/bray/sql/funcs.so' 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))
);
```

--下面的例子创建一个在用户定义类型 complex 和内部类型 point 之间进行类型转换的函数。该函数是用一个从 C 源代码编译的动态装载的对象来实现的。对于 PostgreSQL 而言，要自动寻找类型转换函数，sql 函数必须和返回类型同名，而且不能被重载。该函数名通过使用 SQL 定义里 AS 子句的第二种类型来重载。

```
CREATE FUNCTION point(complex) RETURNS point
AS '/home/bernie/pgsql/lib/complex.so', 'complex_to_point'
LANGUAGE 'c';
```

--该函数的 C 定义是：

```
Point * complex_to_point (Complex *z)
{
    Point *p;

    p = (Point *) malloc(sizeof(Point));
    p->x = z->x;
    p->y = z->y;

    return p;
}
```

兼容性：CREATE FUNCTION 是 PostgreSQL 的扩展。

## 15. CREATE GROUP (创建用户组)

语法：CREATE GROUP name

```
[ WITH
  [ SYSID gid ]
  [ USER username [, ...] ] ]
```

描述：本命令用于在数据库节点上创建一个新的用户组。只有数据库超级用户才能使用这条命令。

使用 ALTER GROUP 修改组成员，使用 DROP GROUP 删除一个组。

输入:

name——组名。

gid——SYSID 子句可以用于选择 PostgreSQL 新组的组标识 (group id) 号。不过, 这样做不是必须的。如果没有指定 gid, 系统将使用从 1 开始自动为 gid 编号, 以已分配的最高组标识号加 1 作为默认值。

username——组的用户列表。用户必须已经存在。

输出: 组成功创建后, 返回信息 “CREATE GROUP”。

用法:

--创建一个空组:

```
CREATE GROUP staff
```

--创建一个有成员的组:

```
CREATE GROUP marketing WITH USER jonathan, david
```

兼容性: SQL92 中没有 CREATE GROUP 命令。Roles 在概念上与组类似。

## 16. CREATE INDEX (创建索引)

语法:

```
CREATE [ UNIQUE ] INDEX index_name ON table
```

```
[ USING acc_name ] ( column [ ops_name ] [, ...] )
```

```
CREATE [ UNIQUE ] INDEX index_name ON table
```

```
[ USING acc_name ] ( func_name( column [, ...] ) [ ops_name ] )
```

描述: 本命令在指定的表 table 上创建一个名为 index\_name 的索引。索引主要用来提高数据库性能, 但是如果使用不当将导致性能下降。

在上面的第一种语法中, 索引的关键字以字段名指定。如果索引的访问模式支持多字段索引, 也可以指定多个字段。

在上面的第二种语法中, 索引是以用户定义的函数 func\_name 的结果定义的, 该函数作用于一个表的一个或多个字段。利用这些函数索引可获取更快的数据访问速度, 这些数据的操作是基于一些需要做一定转换才能用于基本数据的运算符时, 效果更为明显。

PostgreSQL 为索引提供了 btree、rtree 和 hash 三种访问模式。btree 访问模式是一个 Lehman-Yao 高并发 btrees 的实现。rtree 访问模式用 Guttman 的二分算法实现了标准的 rtrees。hash 访问模式是 Litwin 的线性散列的一个实现。单独地列出所用的算法是要表明所有这些访问模式都是完全动态的, 并且不必进行周期性的优化。

使用 DROP INDEX 删除一个索引。

值得注意的是, 当一个索引了的字段涉及到 <、<=、=、>= 或 > 比较操作时, PostgreSQL 的查询优化器将考虑在扫描过程中使用 btree 索引。

当一个索引了的字段涉及到 <<、&<、&>、>>、@、~= 或 && 比较操作时, PostgreSQL 的查询优化器将考虑在扫描过程中使用 rtree 索引。

当一个索引了的字段涉及到 = 比较操作时, PostgreSQL 的查询优化器将考虑在扫描过程中使用散列 (hash) 索引。

目前, 只有 btree 访问模式支持多字段索引。默认时最多可以指定 16 个索引键 (这个

限制可以在编译 PostgreSQL 时修改)。

可以为索引的每个字段指定一个运算符表。运算符表规定将要被该索引用于该字段的运算符。例如，一个 4 字节整数的 btree 索引将使用 int4\_ops 表，这个运算符表包括 4 字节整数的比较函数。实际上，该域的数据类型的默认运算符表一般就足够了。某些数据类型有运算符表的原因是，它们可能有多于一个的有意义的顺序。例如，对复数类型排序时，有可能以绝对值或者以实数部分来进行。可以通过为该数据类型定义两个运算符表，然后在建立索引的时候选择合适的表来实现。有些运算符表有特殊的用途：

运算符表 box\_ops 和 bigbox\_ops 都支持对 box 数据类型的 rtree 索引。两者的区别是 bigbox\_ops 将矩形的坐标按比例缩小，以避免在对非常大的浮点数坐标做乘法、加法和减法时出现浮点错误。如果矩形所在范围的大小是 20,000 个单位的平面或更大，应该用 bigbox\_ops。

int24\_ops 运算符表在为 int2 类型的数据构建索引并且与查询资格条件里的 int4 数据进行比较时很有用。类似的，int42\_ops 支持对要和查询里的 int2 数据进行比较的 int4 数据进行索引。

下面的查询显示所有已定义的运算符表：

```
SELECT am.amname AS acc_name,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
FROM pg_am am, pg_amop amop,
     pg_opclass opc, pg_operator opr
WHERE amop.amopid = am.oid AND
      amop.amopclaid = opc.oid AND
      amop.amopopr = opr.oid
ORDER BY acc_name, ops_name, ops_comp
```

输入：

UNIQUE——要求系统检测当索引创建时（如果数据已经存在）和每次添加数据时表中是否有重复值。如果插入或更新的值会导致重复的记录，系统将提示一个错误信息。

index\_name——要创建的索引名。

table——要索引的表名。

acc\_name——用于索引的访问模式，默认访问模式是 BTREE。PostgreSQL 提供三种用于从索引的访问模式：

BTREE——一个 Lehman-Yao 高度并发 btrees 的实现。

RTREE——Guttman 二分法标准 rtrees 的实现。

HASH——一个 Litwin 线性哈希（散列）法的实现。

column——表的字段名。

ops\_name——一个关联的运算符表。

func\_name——用户定义的函数，其返回值可被索引。

输出：

如果索引创建成功，返回信息“CREATE”；否则返回信息“ERROR: Cannot create index:



'index\_name' already exists. "。

用法：

--在表 films 的 title 字段创建一个 btree 索引

```
CREATE UNIQUE INDEX title_idx
    ON films (title);
```

兼容性：CREATE INDEX 是 PostgreSQL 的扩展，SQL92 中没有 CREATE INDEX 命令。

## 17. CREATE LANGUAGE（创建语言）

语法：CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'langname'  
 HANDLER call\_handler  
 LANCOMPILER 'comment'

描述：使用本命令，PostgreSQL 用户可以在 PostgreSQL 中注册一种新的语言。函数和触发器过程可以使用这种新定义的语言。注册新语言的用户必须具有 PostgreSQL 超级用户权限。

过程语言的调用句柄必须用一种编译语言（如 C 语言），并且在 PostgreSQL 中注册为一个无参数输入、返回值是 opaque 类型（一个用于未指定或未定义类型的容器）的函数，这样就能避免句柄直接被查询当作函数调用。

但是，如果想在句柄提供的语言里进行 PL 函数或触发器过程的实际调用，必须提供参数。

当从触发器管理器调用过程时，唯一的参数是该过程的 pg\_proc 对象 ID。所有从触发器管理器返回的其他信息都可以通过全局量 CurrentTriggerData 指针访问。

当从函数管理器调用过程时，参数是过程 pg\_proc 入口的对象标识（object ID）、传递给 PL 函数的参数个数、在 FmgrValues 结构中的参数和一个指向逻辑变量的指针，函数通过这个指针告诉调用者返回值是否为 SQL NULL。

取得 pg\_proc 表和分析被调用过程参数及返回值类型的任务属于调用句柄。过程中 CREATE FUNCTION 的 AS 子句可以在 pg\_proc 表的 prosrc 字段里找到。这里有可能是过程语言的源文件本身（像 PL/Tcl），也可能是一个文件路径或是其他任何一个告诉调用句柄如何处理细节的东西。

可以从 PostgreSQL 的系统表 pg\_language 中查询语言的有关情况。

由于所有过程语言的调用句柄都必须在 PostgreSQL 中用 C 语言注册，因此它继承了所有 C 函数的功能和限制，同时，过程语言的定义一旦建立便不能更改。

输入：

TRUSTED——TRUSTED 说明对该语言的调用句柄是安全的，也就是说，它不会提供使非特权用户绕过访问限制的功能。如果不使用这个关键字，则只有 PostgreSQL 超级用户才可以使用这种语言创建新的函数。

langname——新的过程化语言的名称，语言名对大小写是不敏感的。一个过程化语言不能超越 PostgreSQL 内建的语言。

HANDLER call\_handler——call\_handler 是预先注册过的函数名，它将被用于执行 PL

过程。

comment——LANCOMPILER 参数是将被插入 pg\_language 表 LANCOMPILER 字段的字符串。在目前版本中，PostgreSQL 不使用这个字段。

输出：如果语言创建成功，命令返回信息“CREATE”；否则，返回信息“ERROR: PL handler function funcname() doesn't exist”，表明函数 funcname()不存在。

用法：

--下面是一个用 C 语言写的 PL 句柄的模板

```
#include "executor/spi.h"
#include "commands/trigger.h"
#include "utils/elog.h"
#include "fmgr.h"          /* for FmgrValues struct */
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"
```

Datum

```
plsample_call_handler(
    Oid      prooid,
    int      pronargs,
    FmgrValues *proargs,
    bool      *isNull)
{
    Datum      retval;
    TriggerData *trigdata;

    if (CurrentTriggerData == NULL) {
        /*
         * Called as a function
         */

        retval = ...
    } else {
        /*
         * Called as a trigger procedure
         */
        trigdata = CurrentTriggerData;
        CurrentTriggerData = NULL;
    }
}
```

```

        retval = ...
    }

    *isNull = false;
    return retval;
}

--只需要在程序中的“.....”处添加几行代码就可以完成 PL 调用句柄。
--下面的命令用于注册例子过程语言
CREATE FUNCTION plsample_call_handler () RETURNS opaque
    AS '/usr/local/pgsql/lib/plsample.so'
    LANGUAGE 'C';
CREATE PROCEDURAL LANGUAGE 'plsample'
    HANDLER plsample_call_handler
    LANCOMPILER 'PL/Sample';

```

兼容性：CREATE LANGUAGE 是 PostgreSQL 的扩展，SQL92 中没有 CREATE LANGUAGE 命令。

## 18. CREATE OPERATOR（创建运算符）

语法：CREATE OPERATOR name ( PROCEDURE = func\_name  
 [, LEFTARG = type1 ] [, RIGHTARG = type2 ]  
 [, COMMUTATOR = com\_op ] [, NEGATOR = neg\_op ]  
 [, RESTRICT = res\_proc ] [, JOIN = join\_proc ]  
 [, HASHES ] [, SORT1 = left\_sort\_op ] [, SORT2 = right\_sort\_op ] )

描述：本命令用于创建一个新的运算符，创建该运算符的用户为该运算符的所有者。

运算符 name 是一个最多 NAMEDATALEN-1 长的（默认为 31 个）由下列字符组成的字串：

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

在选择运算符的名字时，注意以下的限制：

"\$"和":"不能定义为单字符运算符，它们可以是一个多字符运算符名称的一部分。

"--"和"/"不能在运算符名字的任何地方出现，因为它们会被认为是一个注释的开始。

一个多字符的运算符名字不能以"+"或"-"结尾，除非该名字还包含至少下面字符之一：

~ ! @ # % ^ & | ` ? \$ :

例如，@-是一个允许的运算符名，但\*-不是合法的运算符名称。

值得注意的是，当使用非 SQL 标准运算符名时，通常需要用空格将连接的运算符分离开以避免含混。例如，如果定义了一个左目运算符，名为"@”，不能写 X\*@Y；必须写成 X\*  
 @ Y，以保证 PostgreSQL 能将它识别为两个运算符而不是一个。

运算符"!="在输入时映射成"<>"，因为这两个运算符总是等价的。

至少需要定义一个 LEFTARG 或 RIGHTARG。对于双目运算符来说，两者都需要定义。对右目运算符来说，只需要定义 LEFTARG；而对于左目运算符来说，只需要定义

RIGHTARG。

同样，func\_name 过程必须已经用 CREATE FUNCTION 定义过，而且必须被定义为能接受正确数量的参数（一个或是两个）。

换向运算符用于使 PostgreSQL 转换运算符的方向。例如，面积小于运算符（<<<）就有一个换向运算符，即面积大于运算符（>>>）。因此，查询优化器可以自由的将下面查询从：

```
box '((0,0),(1,1))' >>> MYBOXES.description
```

转换到：

```
MYBOXES.description <<< box '((0,0),(1,1))'
```

这就允许执行代码总是使用后面的形式，在某种程度上简化了查询优化器。

类似地，如果存在负号运算符则应该显式地指定。假设一个运算符（面积相等）=== 存在，同样有一个面积不等运算符（!==）。负号运算符允许查询优化器，将：

```
NOT MYBOXES.description === box '((0,0),(1,1))'
```

简化成：

```
MYBOXES.description != box '((0,0),(1,1))'
```

如果提供了一个转换器运算符名称，PostgreSQL 将在目录中查找它。如果能找到，而且其本身没有一个转换器，那么转换器表将被更新，以当前最新运算符作为它的转换器。这一规则同样适用于负号运算符，也就是说允许定义两个互为转换器或负号符的运算符。第一个运算符应该定义为没有转换器或负号符（as appropriate）。当定义第二个运算符时，将第一个符号作为转换器或负号符。第一个运算符将因上述的副作用而被更新（同样获得转换器或负号符）。

HASHES、SORT1 和 SORT2 选项将为查询优化器在进行联合查询时提供支持。PostgreSQL 能够总是用语义替换来计算一个联合。该子句有两个记录变量，这两个变量被一个运算符分开，最后这个子句返回一个逻辑量。另外，PostgreSQL 准备实现一个哈希 - 联合算法（hash-join algorithm）。目前的哈希 - 联合算法只对相等测试运算符有效；而且，数据类型的相等必须意味着类型的表现是按位相等的。例如，一个包含未用位的数据类型，这些位对相等测试没有影响，但却不能用于哈希联合。HASHES 标记告诉优化器，对这个运算符可以安全地使用哈希联合。

类似的，二目排序运算符告诉查询优化器一个融合 - 排序（merge-sort）是否是一个可用的联合策略，并且告诉优化器使用哪个运算符来对这两个操作数表排序。排序运算符应该只提供给相等运算符，并且它们应该对应用于对应的左边和右边数据类型的小于运算符。

如果发现其他联合策略可用，PostgreSQL 将更改优化器和运行系统以利用这些策略，并且在定义一个运算符时将需要更多的指定。

RESTRICT 和 JOIN 选项帮助优化器计算结果的尺寸大小。如果命令 MYBOXES.description <<< box '((0,0),(1,1))' 在判断条件中出现，那么 PostgreSQL 将不得不估计 MYBOXES 中满足该子句的记录数量的范围大小。函数 res\_proc 必须是一个注册过的函数（也就是说它已经用 CREATE FUNCTION 定义过了），它接受一个正确数据的数据类型作为参数，返回一个浮点数。查询优化器只是简单地调用这个函数，将参数 ((0,0),(1,1)) 传入并且将结果乘以关系（表）尺寸以获得所需要的记录数值。

类似地，当运算符的操作数都包含记录变量时，优化器必须计算联合结果的尺寸。函数 `join_proc` 将返回另一个浮点数，这个数就是将两个表相关的记录相乘，计算出预期结果的尺寸。

函数 `my_procedure_1` (`MYBOXES.description`, `box '((0,0),(1,1))'`) 和运算符 `MYBOXES.description === box '((0,0),(1,1))'` 之间的区别是：PostgreSQL 试图优化运算符，并且决定使用索引来缩小相关运算符的搜索区间。但是，对函数将不会有任何优化的动作，而且是强制执行的。最后，函数可有任意个参数，而运算符仅限于一个或两个。

输入：

`name`——定义的运算符名。

`func_name`——用于实现该运算符的函数。

`type1`——如果存在的话，为运算符左边的参数类型。如果是右目运算符，这个参数可以省略。

`type2`——如果存在的话，为运算符右边的参数类型；如果是左目运算符，这个参数可以省略。

`com_op`——对应的换向（commutative）运算符。

`neg_op`——对应的负号运算符。

`res_proc`——此运算符约束选择性评估函数。

`join_proc`——此运算符的联合选择性评估函数。

`HASHES`——表明此运算符支持哈希（散列）联合算法。

`left_sort_op`——如果此运算符支持融合联合（join），为运算符的左边数据的排序运算符。

`right_sort_op`——如果此运算符支持融合联合（join），为运算符的右边数据的排序运算符。

输出：如果运算符创建成功，命令返回信息“CREATE”。

用法：

--下面命令定义一个新运算符，面积相等，用于 `BOX` 数据类型。

```
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：CREATE OPERATOR 是 PostgreSQL 的扩展，SQL92 中没有 CREATE

OPERATOR 命令。

### 19. CREATE RULE (创建规则)

语法: CREATE RULE name AS ON event  
 TO object [ WHERE condition ]  
 DO [ INSTEAD ] [ action | NOTHING ]

描述: 在从数据库或表中更新、插入或删除数据时, PostgreSQL 规则系统允许用户定义一个可选的动作来执行。目前, 规则主要用于实现表视图。

在一个单独的记录正被访问、更新、插入或删除时, 系统同时存在一个旧记录(用于检索、更新和删除)和一个新记录(用于更新和追加)。如果在 ON 子句中所指定的事件(event)和在 WHERE 命令里面所指定的条件(condition)对于旧记录都为真, 那么动作(action部分)的规则就会被自动执行。但是, 旧记录的各字段值和新记录将先用 current.attribute-name 和 new.attribute-name 取代。

规则 action 部分执行时的命令和事务标识与激活该规则的用户命令相同。

一个关于 SQL 规则的注意事项是顺序。如果相同的表名或记录变量出现在规则的 event、condition 和 action 部分。它们将会被认为是不同的记录变量。更准确地说, 只有 new 和 current 在这些子句中共享记录变量。例如, 下面两条规则有相同的语意:

```
ON UPDATE TO emp.salary WHERE emp.name = "Joe"
DO
    UPDATE emp SET ... WHERE ...
ON UPDATE TO emp-1.salary WHERE emp-2.name = "Joe"
DO
    UPDATE emp-3 SET ... WHERE ...
```

每条规则都可以有可选的标记 INSTEAD。如果没有这个标记, action 将在规则的条件(condition)部分的事件(event)发生时作为用户命令的附加部分执行。否则, 动作(action)部分将取代用户命令执行。对于后者, action 可以是关键字 NOTHING。

特别需要指出的是: 重写(rewrite)规则系统既不检测也不执行循环规则。例如, 尽管下面两条规则都能被 PostgreSQL 所接收, 检索命令将导致 PostgreSQL 报错, 因为该查询循环太多次。

--循环重写(rewrite)规则样例。

```
CREATE RULE bad_rule_combination_1 AS
    ON SELECT TO emp
    DO INSTEAD
        SELECT TO toyemp;
CREATE RULE bad_rule_combination_2 AS
    ON SELECT TO toyemp
    DO INSTEAD
        SELECT TO emp;
```

下面的命令试图从 EMP 中检索, 这将导致 PostgreSQL 产生一个错误, 因为查询循环

太多。

```
SELECT * FROM emp;
```

用户必须具有对某个表进行规则定义的权限，这样才能在其上面定义规则。可以使用 GRANT 和 REVOKE 更改权限。

一条 SQL 规则中的对象不能是一个数组引用，也不能有参数。

除了 OID 字段外，一个规则中任何地方都不能引用系统表属性。这意味着在规则的 anywhere 都不能调用记录函数。

规则系统将规则文本和查询规划按文本 (text) 属性合并存储。这意味着当创建的规则加上各种其内部表达式超过一次存储页面请求的值 (8KB) 时，规则创建可能失败。

输入：

name——创建的规则名。

event——事件是 select、update、delete 或 insert 之一。

object——对象是 table 或 table.column。

condition——任何 SQL WHERE 命令。new 或 current 可以取代记录变量出现在任何 SQL 允许记录变量的地方。

action——任何 SQL 命令。new 或 current 可以取代记录变量出现在任何 SQL 允许记录变量的地方。

输出：如果规则创建成功，则返回信息 “CREATE”。

用法：

--令 Sam 获得与 Joe 一样的薪水调整

```
CREATE RULE example_1 AS
    ON UPDATE emp.salary WHERE old.name = "Joe"
DO
    UPDATE emp
    SET salary = new.salary
    WHERE emp.name = "Sam";
```

--当 Joe 获得薪水调整后，事件将为真，Joe 的当前记录和提供的新记录可被执行过程获得。因此，他的新薪水将带入动作部分，随后动作部分被执行。这样 Sam 的薪水就会和 Joe 的一样了。

--下面的规则实现：当 Bill 访问数据库（薪水）时，令 Bill 获得 Joe 薪水的信息。

```
CREATE RULE example_2 AS
    ON SELECT TO EMP.salary
    WHERE old.name = "Bill"
DO INSTEAD
    SELECT emp.salary
    FROM emp
    WHERE emp.name = "Joe";
```

--下面的规则实现：拒绝 Joe 访问雇员的薪水，当他在鞋部时（current\_user 返回当前用户的名称）。

```
CREATE RULE example_3 AS
```

```

ON
SELECT TO emp.salary
WHERE old.dept = "shoe" AND current_user = "Joe"
DO INSTEAD NOTHING;
--下面的规则创建一个玩具部工作的雇员视图。
CREATE toyemp(name = char16, salary = int4);

```

```

CREATE RULE example_4 AS
ON SELECT TO toyemp
DO INSTEAD
SELECT emp.name, emp.salary
FROM emp
WHERE emp.dept = "toy";

```

--下面规则限制新雇员的薪水只能少于或等于 5000。

```

CREATE RULE example_5 AS
ON INERT TO emp WHERE new.salary > 5000
DO
UPDATE NEWSET SET salary = 5000;

```

兼容性: CREATE RULE 命令是 PostgreSQL 的扩展,SQL92 中没有 CREATE RULE 命令。

## 20. CREATE SEQUENCE (创建序列)

语法: CREATE SEQUENCE seqname [ INCREMENT increment ]  
[ MINVALUE minvalue ] [ MAXVALUE maxvalue ]  
[ START start ] [ CACHE cache ] [ CYCLE ]

描述: 本命令用于向当前数据库添加一个新的序列生成器,包括创建和初始化一个新的名为 seqname 的单行表。新的序列生成器将为使用此命令的用户所有。

在序列创建后,可以使用函数 nextval(seqname)从序列中获得新的数字;函数 currval('seqname')用于获取序列的当前值;函数 setval('seqname', newvalue)用于为序列设置当前值。

使用像 SELECT \* FROM sequence\_name 这样的查询可以获得序列的参数。除了获取最初的参数外,可以用 SELECT last\_value FROM sequence\_name 命令获得后端分配的最后值。

如果用于序列对象的缓存大于 1,而且该对象被多个后端同时使用,就有可能产生不可预料的结果。每个后端在访问过序列对象并递增序列对象的最后值后,将分配跟在序列值后面的缓存数。这样,该后端在后面的缓存数-1 次返回序列值时,将使用预分配好的数值,而不对共享对象作任何更新。所以,已经分配但在当前会话中没有使用的数字将会丢失。而且,尽管多个后端保证分配的是独立的序列值,当考虑所有的后端时该数值却有可能是乱序的。例如,设置缓存数为 10,后端 A 可能保留数值 1..10 并且返回 nextval=1,而后端 B 可能保留数值 11..20 并在后端 A 生成 nextval=2 之前返回 nextval=11。因此,



将缓存数设为 1，可以安全地假设 nextval 的数值是顺序生成的；当缓存数设置大于 1，只能假设 nextval 值都是独立的，而不能假设它们都是按顺序生成的。同样，last\_value 将反映由任何后端保留的最后数值，不管它是不是 nextval 曾返回过的值。

每个后端使用其自身的缓存来存储分配的数字。已分配但当前会话没有使用的数字将丢失，这会导致序列中出现“空洞”。

输入：

seqname——将要创建的序列号名。

increment——INCREMENT increment 子句是可选的。一个正数将生成一个递增的序列，一个负数将生成一个递减的序列。默认的增量值为 1。

minvalue——可选的子句 MINVALUE minvalue 决定一个序列可生成的最小值。默认值分别是：递增序列为 1，递减为-2147483647。

maxvalue——使用可选子句 MAXVALUE maxvalue 决定序列的最大值。默认值分别是：递增序列为 2147483647，递减序列为-1。

start——可选的 START，start 子句使序列可以从任意位置开始。默认初始值是：递增序列为 minvalue，递减序列为 maxvalue。

cache——CACHE cache 选项预分配序列号，并且为了快速访问将这些预分配的序列号存储在内存中。最小值（也是默认值）是 1，一次只能生成一个值，也就是说没有缓存。

CYCLE——可选的 CYCLE 关键字用于使序列到达最大值（maxvalue）或最小值（minvalue）时复位并继续下去。如果达到极限，生成的下一个数据将分别是最小值（minvalue）或最大值（maxvalue）。

输出：如果序列生成器创建成功，则返回信息“CREATE”；否则，返回下列错误信息之一：

“ERROR: Relation 'seqname' already exists”，序列已经存在。

“ERROR: DefineSequence: MINVALUE (start) can't be >= MAXVALUE (max)”，初始值超出范围（最大值）。

“ERROR: DefineSequence: START value (start) can't be < MINVALUE (min)”，初始值超出范围（最小值）。

“ERROR: DefineSequence: MINVALUE (min) can't be >= MAXVALUE (max)”，最小值和最大值不连贯。

用法：

--创建一个名为 serial 的递增序列，从 101 开始

```
CREATE SEQUENCE serial START 101;
```

--从此序列中选出下一个数字

```
SELECT NEXTVAL ('serial');
```

```
nextval
```

```
-----
```

```
114
```

--在一个 INSERT 中使用此序列

```
INSERT INTO distributors VALUES (NEXTVAL('serial'),'nothing');
```

--在一个 COPY FROM 后设置序列

```
CREATE FUNCTION distributors_id_max() RETURNS INT4
```

```
AS 'SELECT max(id) FROM distributors'
```

```
LANGUAGE 'sql';
```

```
BEGIN;
```

```
COPY distributors FROM 'input_file';
```

```
SELECT setval('serial', distributors_id_max());
```

```
END;
```

兼容性：CREATE SEQUENCE 是 PostgreSQL 的扩展，SQL92 中没有 CREATE SEQUENCE 命令。

## 21. CREATE TABLE (创建表)

语法：CREATE [ TEMPORARY | TEMP ] TABLE table (  
     column type  
     [ NULL | NOT NULL ] [ UNIQUE ] [ DEFAULT value ]  
     [column\_constraint\_clause | PRIMARY KEY ] [ ... ]  
     [, ... ]  
     [, PRIMARY KEY ( column [, ... ] ) ]  
     [, CHECK ( condition ) ]  
     [, table\_constraint\_clause ]  
     ) [ INHERITS ( inherited\_table [, ... ] ) ]

描述：本命令用于向当前数据库中追加一个新的表，新表归执行本命令的用户所有。

每个 type 可以是简单类型、复合（集）类型或者数组类型。每个字段都可以指定为非空并且每个字段都可以有一个默认值。

可选的 INHERITS 子句指定一个表集合，所创建的表自动从这个表集合中的表继承所有字段。如果任何继承的字段出现次数多于一次，PostgreSQL 报告一个错误。PostgreSQL 允许所创建的表继承那些在继承分级中级别比它高的表的函数。函数的继承是根据公共 Lisp 对象系统（CLOS，Common Lisp Object System）的习惯进行的。

每个新表或表 table 自动被创建为一个类型，因此表中的一条或更多实例也就是一个类型，因而可以用于 ALTER TABLE 或其他 CREATE TABLE 命令。

新表将作为一个没有初始值的堆创建。一个表可以有不超过 1600（这是因为受字段大小必须小于 8192 字节的限制）个字段，但是，这个限制在一些节点上可以通过适当的配置来降低。一个用户表不能与系统表同名。

输入：

TEMPORARY——此表只是为这次会话创建，并且在会话结束后自动删除。当临时表存在时，同名的永久表是不可见的。

table——将要创建的新表的名称。

column——字段名。

type——字段类型。这里可以包括数组的类型。

DEFAULT value——字段的默认值。

column\_constraint\_clause——可选的字段约束子句，指定一系列完整性约束和测试，当对表进行更新或插入操作时，必须满足这些约束条件才能成功。每个约束必须生成一个逻辑式。尽管 SQL92 要求 column\_constraint\_clause 只用于指定某一行，但 PostgreSQL 允许在一个字段的约束中引用多个字段。

table\_constraint\_clause——可选的表约束（CONSTRAINT）子句，指定一系列字段的约束，当对表进行更新或插入时必须满足这些约束。每个约束必须生成一个逻辑表达式，可以对多个字段使用同一个约束。一个表只能指定一个 PRIMARY KEY 子句，PRIMARY KEY column（表约束）和 PRIMARY KEY（列/字段约束）是互斥的。

INHERITS inherited\_table——可选的 INHERITS（继承）子句指定一系列表名，这个表将自动从这些表继承所有字段。如果任何继承字段出现的次数超过一次，PostgreSQL 将报告一个错误。Postgres 允许所创建的表继承所有其父表的函数。

输出：如果表创建成功，则返回信息“CREATE”，否则，返回错误信息。

以下是本命令各子句的说明。

(1) DEFAULT value 子句。DEFAULT 子句为表的字段指定默认值，默认值的类型必须与字段定义的数据类型相同。默认值可以是一个字符串、一个用户函数或一个 niladic 函数。一个插入（INSERT）操作如果包括了一个没有默认值字段，这时如果没有显式地提供该字段的数据值，系统将用一个 NULL 代替该字段的值。默认的字符串意味着默认值是指定的常量，默认的 niladic 函数或用户函数意味着默认值是在插入（INSERT）时指定的函数值。

有两类 niladic 函数：一类是 niladic USER，其中的 USER 可以是 CURRENT\_USER/USER、SESSION\_USER、SYSTEM\_USER（目前未予以实现）。另一类是 niladic datetime，其中 datetime 可以是 CURRENT\_DATE、CURRENT\_TIME、CURRENT\_TIMESTAMP。

例如：

--给字段 did 和 number 赋予一个常量值作为默认值

```
CREATE TABLE video_sales (
    did      VARCHAR(40) DEFAULT 'luso films',
    number   INTEGER DEFAULT 0,
    total    CASH DEFAULT '$0.0'
);
```

--将一个现有的序列作为 did 的默认值

```
CREATE TABLE distributors (
    did      DECIMAL(3) DEFAULT NEXTVAL('serial'),
    name     VARCHAR(40) DEFAULT 'luso films'
);
```

(2) 字段 CONSTRAINT 子句。语法格式为：

[ CONSTRAINT name ] { [ NULL | NOT NULL ] | UNIQUE | PRIMARY KEY | CHECK constraint | REFERENCES

```

reftable
(refcolumn)
[ MATCH matchtype ]
[ ON DELETE action ]
[ ON UPDATE action ]
[ [ NOT ] DEFERRABLE ]
[ INITIALLY checktime ] }
[, ...]

```

其中：

name——约束的名称。如果没有指定 name，系统将根据表和字段的名称生成一个唯一的名称用于 name。

NULL——字段允许包含 NULL 值，这是默认值。

NOT NULL——字段不允许包含 NULL 值。

UNIQUE——字段的值必须唯一。在 PostgreSQL 中，这是通过在表上隐含地创建一个唯一索引来实现的。

PRIMARY KEY——表明本字段是一个主键，暗示着其唯一性是由系统强制提供的，而且其他表可以依赖此字段作为行标识。

constraint——约束的定义。

可选的约束子句可以用于定义某种约束或者测试，当进行插入或者更新操作时，新的或者更新的记录必须满足这个约束或测试，否则操作无法成功地进行。每个约束必须是一个逻辑表达式。多个字段可以在一个约束中引用。作为表约束的 PRIMARY KEY 的使用是与作为字段约束的 PRIMARY KEY 互相冲突和不可兼容的。

约束是一个命名的规则：它是一个 SQL 对象，通过对 INSERT、UPDATE 或 DELETE 等基本表的操作结果进行限制，进而可以获得有效的结果集。

有两种方法定义整合约束：表约束和字段约束。

一个字段约束是作为字段定义的一部分的一个整合约束，而且逻辑上一旦创建就会成为表约束。可用的字段约束有：

PRIMARY KEY——[ CONSTRAINT name ] PRIMARY KEY。其中，CONSTRAINT name 为约束的名称。PRIMARY KEY 字段约束表明表中的一个字段只能包含唯一的（不重复）非空的数值。在该字段的 PRIMARY KEY 约束定义中不需要显式地包含 NOT NULL 约束。一个表只能指定一个 PRIMARY KEY。

值得注意的是，PostgreSQL 自动创建一个唯一索引以保证数据的完整性。在同一个表中，PRIMARY KEY 约束定义的字段应该和其他定义了 UNIQUE 约束的字段不同名（即不能为同一个字段），否则会导致等价索引的重复和增加不必要的处理，虽然，PostgreSQL 并没有明文禁止这样做。

```

REFERENCES——[ CONSTRAINT name ] REFERENCES reftable [ ( refcolumn ) ]
[ MATCH matchtype ]
[ ON DELETE action ]
[ ON UPDATE action ]

```

[ [ NOT ] DEFERRABLE ]

[ INITIALLY checktime ]

REFERENCES ( 参考 ) 字段约束指明一个表的某字段只能包含匹配一个被参考表的某参考字段的数据值。向这个字段追加的数值使用给出的匹配类型与被参考表的参考字段进行匹配。另外, 当被参考列的数据被修改, 动作是对该列的当前数据进行比较。

REFERENCES 约束定义这样一个规则: 一个字段的数值要与另外一个字段的数值进行对比检查。REFERENCES 还可以作为一个 FOREIGN KEY 表约束的一部分。其中:

CONSTRAINT name 为约束名称。

reftable 包含对比检查数据的表名称。

refcolumn 为对比检查的字段名称, 如果没有指定, 则使用表 PRIMARY KEY。

MATCH matchtype 为匹配类型, 有三种匹配类型: MATCH FULL、MATCH PARTIAL 和一种默认的匹配类型 ( 如果什么都没有指定的话 ); 除非所有的外键字段都是 NULL, 否则 MATCH FULL 将不允许一个多字段外键的某个列为 NULL。默认的 MATCH 类型允许某些外键字段为 NULL, 而其他外键字段不能为 NULL。MATCH PARTIAL 目前不支持。

ON DELETE action 定义一个当参考表中的参考行被删除时要执行的动作。可用的动作有: NO ACTION ( 如果违反外键则产生错误, 这是默认值 )、RESTRICT ( 与 NO ACTION 相同 )、CASCADE ( 删除任何引用参考行的行 )、SET NULL ( 将参考字段的值设置为 NULL )、SET DEFAULT ( 将参考字段的值设置为默认值 )。

ON UPDATE action 定义一个当参考表中的参考字段被更新时要执行的动作, 如果行被更新而参考字段没有改变, 不发生任何动作。可用的动作有: NO ACTION ( 如果违反外键则产生错误, 这是默认值 )、RESTRICT ( 与 NO ACTION 相同 )、CASCADE ( 将参考字段的值更新为被参考字段的新值 )、SET NULL ( 将参考字段的值设置为 NULL )、SET DEFAULT ( 将参考字段的值设置为默认值 )。

[ NOT ] DEFERRABLE。这个选项控制该约束是否可以推迟到事务的结尾。如果为 DEFERRABLE, SET CONSTRAINTS ALL DEFERRED 将导致只是在事务的结束时才检查外键。默认为 NOT DEFERRABLE。

INITIALLY checktime 定义约束检查时间。checktime 有两个可能的值用以指定检查约束的默认时间。一个值是 DEFERRED, 只在事务结束时检查约束; 另一个值是 IMMEDIATE, 在每条命令执行完毕时检查约束, 这是默认值。

UNIQUE——[ CONSTRAINT name ] UNIQUE。其中, CONSTRAINT name 为赋予一个约束的任意标记。UNIQUE 约束定义这样的规则: 表中一组由一个或多个独立字段组成的集合中只能包含一个唯一的数值。一个字段定义可以包含 UNIQUE 约束, 但不一定要包含 NOT NULL 约束。对于一个没有 NOT NULL 约束的字段, 如果有多个空值并不违反 UNIQUE 约束。在这一点上, PostgreSQL 与 SQL92 的定义不一致, 但却更有意义。

每个 UNIQUE 字段约束必须赋予一个该表中没有被其他 UNIQUE 或 PRIMARY KEY 约束定义过的字段上。

值得注意的是, PostgreSQL 自动为每个 UNIQUE 约束创建一个唯一索引, 以保证数据的完整性。

例如:

--为表 distributors 定义一个 UNIQUE 字段约束

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40) UNIQUE
);
```

--与下面的表约束相同

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

**CHECK**——[ CONSTRAINT name ] CHECK ( condition [, ...] )。其中，name 为赋予约束的任意名称，condition 为一个能产生一个逻辑值的条件表达式。CHECK 约束定义一个字段合法数据的限制条件。CHECK 约束也可以作为表约束。SQL92 CHECK 字段约束只能对表中的一个字段进行定义或使用，而 PostgreSQL 没有这个限制。

**NOT NULL**——[ CONSTRAINT name ] NOT NULL。NOT NULL 约束表明一个字段只能包含非空 (non-null) 数值。NOT NULL 约束只是一个字段约束，不允许作为一个表约束。例如：

--在表 distributors 上定义两个非空的字段约束，其中一个是命名约束

```
CREATE TABLE distributors (
    did      DECIMAL(3) CONSTRAINT no_null NOT NULL,
    name     VARCHAR(40) NOT NULL
);
```

(3) 表约束 CONSTRAINT 子句。语法格式为：

```
[ CONSTRAINT name ] { PRIMARY KEY | UNIQUE } ( column [, ...] )
[ CONSTRAINT name ] CHECK ( constraint )
[ CONSTRAINT name ] FOREIGN KEY ( column [, ...] )
    REFERENCES reftable
    (refcolumn [, ...] )
    [ MATCH matchtype ]
    [ ON DELETE action ]
    [ ON UPDATE action ]
    [ [ NOT ] DEFERRABLE ]
    [ INITIALLY checktime ]
```

其中，CONSTRAINT name 为完整性约束的名称；column [, ...] 为定义唯一索引、PRIMARY KEY 或 NOT NULL 约束的字段名；CHECK ( constraint ) 为约束估值的逻辑表达式。表约束用于对一个基本表的一个或多个字段定义的完整性进行约束。“表约束”的四个元素为 UNIQUE、CHECK、PRIMARY KEY 和 FOREIGN KEY。

**UNIQUE 约束**——[ CONSTRAINT name ] UNIQUE ( column [, ...] )。其中

CONSTRAINT name 为约束的名称；column 为表的字段名。UNIQUE 约束定义表中由一个或多个独立字段组成的集合只能包含唯一的数。表的 UNIQUE 约束和对应的字段约束的特性是一样的，区别在于表约束可以跨越多个字段。

例子：

--给一个表 distributors 定义一个 UNIQUE 表约束

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

PRIMARY KEY 约束——[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )。其中，CONSTRAINT name 为约束的名称；column [, ...] 为表的一个或者多个字段。PRIMARY KEY 约束定义表的由一个或多个字段组成的集合只能包含唯一的、不重复的非空数值。对字段定义的 PRIMARY KEY 约束不需要包含 NOT NULL 约束。

PRIMARY KEY 表约束与字段约束相似，区别在于前者具有控制多字段的能力。

REFERENCES 约束——[ CONSTRAINT name ] FOREIGN KEY ( column [, ...] )

```
REFERENCES reftable [ ( refcolumn [, ...] ) ]
[ MATCH matchtype ]
[ ON DELETE action ]
[ ON UPDATE action ]
[ [ NOT ] DEFERRABLE ]
[ INITIALLY checktime ]
```

REFERENCES ( 参考 ) 约束定义这样一个规则：一个字段的数值与另外一个字段的数值进行对比检查。REFERENCES 还可以作为一个 FOREIGN KEY 表约束的一部分。FOREIGN KEY 约束定义另外一个规则：某表的一个或多个独立字段的组与被参考表中的一组独立字段相关，FOREIGN KEY 表约束与相当的字段约束相似，只是多了控制多字段的能力。其中：

CONSTRAINT name 为约束的名称。

column [, ...] 为表的一个或者多个字段的名称。

reftable 为对比检查数据表的名称。

referenced column [, ...] 为对比检查的一个或多个字段的名称，如果没有指定，使用表的 PRIMARY KEY。

MATCH matchtype 为匹配类型。有三种匹配类型：MATCH FULL、MATCH PARTIAL 和一种默认的匹配类型。除非所有的外键字段都是 NULL，否则 MATCH FULL 不允许一个多字段外键的某个字段为 NULL。默认的 MATCH 类型是允许某些外键字段为 NULL，而其他部分的外键字段不是 NULL。MATCH PARTIAL 在目前的 PostgreSQL 版本中不支持。

ON DELETE action 定义当一个被参考表中的被参考行被删除时，要进行的动作。可供使用的动作包括：NO ACTION ( 如果违反外键则产生错误，这是默认值 )、RESTRICT ( 与 NO ACTION 相同 )、CASCADE ( 删除任何引用参考行的行 )、SET NULL ( 将参考字段的

值设置为 NULL ) SET DEFAULT ( 将参考字段的值设置为默认值 )

ON UPDATE action 定义当一个被参考表中的被参考字段更新时,要进行的动作。如果行被更新而参考字段没有改变,不发生任何动作。可供使用的动作包括:NO ACTION ( 如果违反外键则产生错误,这是默认值)、RESTRICT ( 不允许被参考的字段更新)、CASCADE(将参考字段的值更新为被参考字段的新值)、SET NULL ( 将参考字段的值设置为 NULL ) SET DEFAULT ( 将参考字段的值设置为默认值 )

[ NOT ] DEFERRABLE。这个选项控制该约束是否可以推迟到事务的结尾。如果设置为 DEFERRABLE ,SET CONSTRAINTS ALL DEFERRED 将控制只是在事务的结束时才检查外键。默认值为 NOT DEFERRABLE。

INITIALLY checktime 定义约束检查时间。checktime 有两个可能的值用以指定检查约束的默认时间。一个是 IMMEDIATE, 在每条命令执行结束时检查约束,这是默认值; DEFERRED, 只在事务结尾检查约束。

例子:

--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(03) PRIMARY KEY DEFAULT NEXTVAL('serial'),
    name      VARCHAR(40) NOT NULL CHECK (name <> '')
);
```

--创建一个有 2 维数组的表

```
CREATE TABLE array (
    vector INT[][]
);
```

--给 films 表定义一个 UNIQUE 表约束, UNIQUE 可以定义在表的一个或多个字段上

```
CREATE TABLE films (
    code      CHAR(5),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT production UNIQUE(date_prod)
```



```
);
--定义一个 CHECK 字段约束
CREATE TABLE distributors (
    did      DECIMAL(3) CHECK (did > 100),
    name     VARCHAR(40)
);
--定义一个 CHECK 表约束
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40)
    CONSTRAINT con1 CHECK (did > 100 AND name > '')
);
--为表 films 定义一个 PRIMARY KEY 表约束, PRIMARY KEY 表约束可以对一个或多个字段进行
```

定义

```
CREATE TABLE films (
    code     CHAR(05),
    title    VARCHAR(40),
    did      DECIMAL(03),
    date_prod DATE,
    kind     CHAR(10),
    len      INTERVAL HOUR TO MINUTE,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

--为表 distributors 定义一个 PRIMARY KEY 字段约束, PRIMARY KEY 字段约束只能对表中的一个字段定义

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     CHAR VARYING(40),
    PRIMARY KEY(did)
);
CREATE TABLE distributors (
    did      DECIMAL(03) PRIMARY KEY,
    name     VARCHAR(40)
);
```

兼容性: PostgreSQL 的 CREATE TABL 命令与 SQL92 的 CREATE TABLE 命令有多处不兼容的地方,主要表现在:

(1) 除了本地可见的临时表外,SQL92 还定义了一条 CREATE GLOBAL TEMPORARY TABLE 命令和一个可选的 ON COMMIT 子句,语法格式为:

```
CREATE GLOBAL TEMPORARY TABLE table ( column type [
```

```

    DEFAULT value ] [ CONSTRAINT column_constraint ] [, ...] )
    [ CONSTRAINT table_constraint ] [ ON COMMIT { DELETE | PRESERVE }
ROWS ]

```

对于临时表，CREATE TEMPORARY TABLE 命令定义一个其他客户端可见的新表和定义表的字段和约束。

CREATE TEMPORARY TABLE 可选的 ON COMMIT 子句用于指定，当提交事务时是否需要将临时表的行清空。如果省略 ON COMMIT 子句（默认值），则在提交时清空临时表中的数据。

例如：

```

CREATE TEMPORARY TABLE actors (
    id          DECIMAL(03),
    name        VARCHAR(40),
    CONSTRAINT actor_id CHECK (id < 150)
) ON COMMIT DELETE ROWS;

```

(2) SQL92 对 UNIQUE 定义了一些附加的功能。

表约束定义：

```

[ CONSTRAINT name ] UNIQUE ( column [, ...] )
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]

```

字段约束定义：

```

[ CONSTRAINT name ] UNIQUE
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]

```

(3) NULL 约束是一个 PostgreSQL 对 SQL92 的扩展，将它包含进来是为了和 NOT NULL 子句对称。

(4) SQL92 对 NOT NULL 定义了一些附加的功能：

```

[ CONSTRAINT name ] NOT NULL
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]

```

(5) SQL92 对约束(CONSTRAIN) 定义了一些附加的功能，并且还定义了断言和域(domain) 约束。PostgreSQL 目前还不支持域(domain) 和断言。一个断言是一种特殊类型的完整性约束，并且和其他约束共享相同的名字空间(namespace)。不过，一个断言不像约束那样必须依赖于某个基本表，所以 SQL-92 提供了一个 CREATE ASSERTION 命令作为定义约束的一个可选手段：

```

CREATE ASSERTION name CHECK ( condition )

```

域约束是用 CREATE DOMAIN 或 ALTER DOMAIN 命令定义的：

```

[ CONSTRAINT name ] CHECK constraint
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]

```

表约束定义：

```
[ CONSTRAINT name ] { PRIMARY KEY ( column, ... ) | FOREIGN KEY constraint |
UNIQUE constraint | CHECK constraint }
[ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
[ [ NOT ] DEFERRABLE ]
```

字段约束定义：

```
[ CONSTRAINT name ] { NOT NULL | PRIMARY KEY | FOREIGN KEY constraint |
UNIQUE | CHECK constraint }
[ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
[ [ NOT ] DEFERRABLE ]
```

一个 CONSTRAINT (约束) 定义可以包含一个任意顺序的延迟属性的子句和/或一个初始化约束式子句。

NOT DEFERRABLE 意味着每条命令执行后都必须校验是否违反约束规则。SET CONSTRAINTS ALL DEFERRED 对这类约束没有影响。

DEFERRABLE 控制约束是否可以被推迟到事务的结束。如果使用了 SET CONSTRAINTS ALL DEFERRED 或者约束被设置成为 INITIALLY DEFERRED, 外键只在事务结束的时候检查。

SET CONSTRAINT, 只为当前事务改变外键约束模式。

INITIALLY IMMEDIATE, 只在事务结束的时候检查约束, 这是默认值。

INITIALLY DEFERRED, 在每条命令后检查约束。

(6) SQL92 对 CHECK 定义了一些附加功能。

表约束定义：

```
[ CONSTRAINT name ] CHECK ( VALUE condition )
[ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
[ [ NOT ] DEFERRABLE ]
```

字段约束定义：

```
[ CONSTRAINT name ] CHECK ( VALUE condition )
[ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
[ [ NOT ] DEFERRABLE ]
```

(7) SQL92 为 PRIMARY KEY 定义了一些附加的功能。

表约束定义:

```
[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
[ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
[ [ NOT ] DEFERRABLE ]
```

字段约束定义：

```
[ CONSTRAINT name ] PRIMARY KEY
[ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
[ [ NOT ] DEFERRABLE ]
```

## 22. CREATE TABLE AS (利用表创建表)

语法: CREATE TABLE table [ (column [, ...] ) ]  
AS select\_clause

描述: 本命令允许用户利用现有表创建一个新表。它具有与 SELECT INTO 命令相同的功能, 但 SELECT INTO 有更多、更直接的语法。

输入:

table——要创建的表名。

column——字段的名称, 多字段的名称可以用逗号分隔。

select\_clause——一个有效的查询命令。请参考 SELECT 获取允许的语法信息。

输出: 请参考 CREATE TABLE 和 SELECT。

## 23. CREATE TRIGGER (创建触发器)

语法: CREATE TRIGGER name { BEFORE | AFTER } { event [OR ...] }  
ON table FOR EACH { ROW | STATEMENT }  
EXECUTE PROCEDURE func ( arguments )

描述: 本命令用于向现有的数据库增加一个新的触发器, 新触发器与表 table 相连并且执行定义的函数 funcname。

触发器可以定义为在对记录进行操作之前 (在检查约束之前和 INSERT、UPDATE 或 DELETE 执行前) 或之后 (在检查约束之后和完成了 INSERT、UPDATE 或 DELETE 操作) 触发。如果触发器在事件之前触发, 触发器可能略过当前记录的操作或改变被插入的 (当前) 记录 (只对 INSERT 和 UPDATE 操作有效)。如果触发器在事件之后, 所有更改, 包括最后的插入、更新或删除对触发器都是“可见”的。

CREATE TRIGGER 是一个 PostgreSQL 的扩展。

只有表的所有者才能为该表创建触发器。

在当前的版本 (v7.0) 中, STATEMENT 触发器还没有实现。

输入:

name——触发器名称。

table——表名称。

event——触发事件, 可取的值为 INSERT、DELETE 或 UPDATE。

funcname——一个用户提供的函数。

输出: 如果触发器成功创建, 返回信息“CREATE”。

用法:

--在插入或更新表 films 之前检查分销商代码是否存在于 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');
```

--在删除或更新一个分销商的内容之前，将所有记录移到表 films 中

```
CREATE TRIGGER if_film_exists
    BEFORE DELETE OR UPDATE ON distributors FOR EACH ROW
    EXECUTE PROCEDURE check_foreign_key (1, 'CASCADE', 'did', 'films',
'did');
```

--上面第二个例子可以使用一个 FOREIGN KEY 约束实现

```
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 中没有 CREATE TRIGGER 命令。

## 24. CREATE TYPE（创建数据类型）

语法：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 ] )

描述：本命令允许用户在 PostgreSQL 的当前数据库中创建一个新的用户数据类型。定义该类型的用户成为其所有者。typename 是新类型的名称，而且必须在所定义的数据库中唯一。

CREATE TYPE 需要在定义类型之前先注册两个函数（用创建函数命令）。新的基本类型的形式由 input\_function 决定，它将该类型的外部形式转换成可以被对该类型操作的运算符和函数识别的形式。output\_function 的用途相反。输入和输出函数都必须定义为接收一个或两个类型为“opaque”的参数的函数。

新的基本数据类型可定义为定长，这时 internallength 是一个正整数；也可以是变长的，这时 PostgreSQL 假定新类型的格式和 PostgreSQL 所支持的“text”类型一样。如果要定义一个变长类型，应该将 internallength 设成 VARIABLE。类似的，外部形式也要用 externallength 关键字指定。

要指明一个类型是数组以及指明一个类型有数组元素，应该使用 element 关键字。例如，定义一个 4 字节整数（“int4”），方法如下：

```
ELEMENT = int4
```

要指定用于这种类型数组的分隔符，可设置 delimiter 分隔符。默认的分隔符是逗号（“,”）。

一个可选的默认值可用于令用户指定某种位模式来代表“数据不存在”，使用的关键字为 DEFAULT。

可选的函数 `send_function` 和 `receive_function` 用于请求 PostgreSQL 的应用程序和 PostgreSQL 数据库不在同一台机器的场合。在这种情况下，运行 PostgreSQL 的机器所用的数据类型格式可能和远端机器的不一样。这时，系统将服务器到客户端的数据转换成一个标准格式，当服务器收到从客户端传来的数据时再转换成适用于机器的格式。如果没有这样的函数指定，系统就假设内部数据格式可以被任何相关的硬件体系所接受。例如，如果在 Sun-4 和一台 DECstation 之间传递数据，单字节数据就不必转换，但许多其他类型需要转换。

可选项 `PASSEDBYVALUE` 表明使用该数据类型的运算符或函数应该传递一个参数的值而不是引用（形参）。值得注意的是，对内部格式超过 4 个字节的类型，任何时候都不能传递参数值。

对于新的基本类型，用户可以通过这里描述的一些功能定义运算符、函数和聚集。

两个通用内建函数，`array_in` 和 `array_out`，用于快速建立变长数组类型。这些函数可对任何现有的 PostgreSQL 数据类型进行操作。

一个“正常”PostgreSQL 类型最多只能有 8192 字节长。如果需要更大的类型，必须创建大对象类型（Large Object type）。所有大对象类型的长度总是变长（VARIABLE）的。

输入：

`typename`——要创建的类型名。

`internallength`——一个字符串，说明新类型的内部长度。

`externallength`——一个字符串，说明新类型的外部长度。

`input_function`——一个函数的名称，由 CREATE FUNCTION 创建，将数据从外部类型转换成内部类型。

`output_function`——一个函数的名称，由 CREATE FUNCTION 创建，将数据从内部格式转换成适合于显示的形式。

`element`——被创建的类型是数组，它定义数组元素的类型。

`delimiter`——数组的分隔字符。

`default`——用于显示的表示“数据不存在”的默认字符串。

`send_function`——用 CREATE FUNCTION 创建的函数名，该函数将该类型的数据转换成一个适合传输到其他机器的形式。

`receive_function`——用 CREATE FUNCTION 创建的函数名，该函数将该类型从适合于传输给其他机器的形式转换为内部形式。

输出：如果创建类型成功，返回信息“CREATE”。

例子：

--这个命令创建长方形数据类型，并且将这种类型用于一个表的定义

```
CREATE TYPE box (INTERNALLENGTH = 8,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2);
CREATE TABLE myboxes (id INT4, description box);
```

--这条命令创建一个变长数组类型，其数组元素的类型是整数

```
CREATE TYPE int4array (INPUT = array_in, OUTPUT = array_out,
    INTERNALLENGTH = VARIABLE, ELEMENT = int4);
CREATE TABLE myarrays (id int4, numbers int4array);
--这条命令创建一个大对象类型并用其创建了一个表
CREATE TYPE bigobj (INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE);
CREATE TABLE big_objs (id int4, obj bigobj);
```

--类型名不能以下划线 ( "\_" ) 开头而且最多只能有 31 个字符长。这是因为 PostgreSQL 自动地为每种基本类型创建了一个数组类型，而且该数组类型的名字是在基本类型名前面加一个下划线。

兼容性：CREATE TYPE 是 SQL3 命令。

## 25. CREATE USER (创建数据库用户)

语法：CREATE USER username

```
[ WITH
  [ SYSID uid ]
  [ PASSWORD 'password' ] ]
[ CREATEDB | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
[ IN GROUP      groupname [, ...] ]
[ VALID UNTIL   'abstime' ]
```

描述：本命令用于向一个 PostgreSQL 节点增加一个新用户，只有数据库超级用户才能执行这条命令。

使用 ALTER USER 修改用户的口令和权限；使用 DROP USER 删除一个用户；使用 ALTER GROUP 从组中增加或删除用户。PostgreSQL 中有一个脚本 createuser，该脚本与本命令的功能相同，实际上，它是调用这条命令来完成用户创建的，可以在命令行上运行。

输入：

username——用户名。

uid——SYSID 子句可以用于选择待创建用户的 PostgreSQL 用户标识号。这个用户标识号不需要与 UNIX 系统的用户标识号匹配，但是也可以相同。如果没有指定 uid，系统自动分配用户标号。

password——设置用户的口令。如果不准备使用口令认证，那么可以省略这个选项；否则该用户将不能连接到一个基于口令认证的服务器上。

CREATEDB、NOCREATEDB——这个子句定义用户的创建数据库权限。如果使用了 CREATEDB，被定义的用户将允许创建其自己的数据库。而使用 NOCREATEDB 将取消该用户的创建数据库的能力。如果忽略本子句，默认是 NOCREATEDB。

CREATEUSER、NOCREATEUSER——该子句决定一个用户是否能创建一个新的用户。这个选项同样将次用户变成数据库超级用户，可以跨越所有访问限制。省略这个参数时，用户的这个属性为 NOCREATEUSER。

groupname——新用户所在的组的名称。

abstime——VALID UNTIL (有效期) 子句设置一个绝对时间，过了该时间后用户的

PostgreSQL 登录将不再有效。如果省略这个子句，登录将总是有效的。

输出：如果命令成功完成，返回信息“CREATE USER”。

用法：

--创建一个没有口令的用户

```
CREATE USER jonathan
```

--创建一个有口令的用户

```
CREATE USER davide WITH PASSWORD 'jw8s0F4'
```

--创建一个有口令的用户，其帐号在 2001 年底失效，即当 2002 年走过一秒后，该帐号将不再有效

```
CREATE USER miriam WITH PASSWORD 'jw8s0F4' VALID UNTIL 'Jan 1 2002'
```

--创建一个拥有创建数据库权限的用户

```
CREATE USER manuel WITH PASSWORD 'jw8s0F4' CREATEDB
```

兼容性：SQL92 中没有 CREATE USER 命令。

## 26. CREATE VIEW（创建视图）

语法：CREATE VIEW view AS SELECT query

描述：本命令用于定义一个视图。这个视图不是物理上实际存在于磁盘，系统自动生成一个改写索引规则的查询以支持在视图上的检索。在目前版本的 PostgreSQL 中，视图是只读的。

输入：

view——将要创建的视图名称。

query——一个将为视图提供行和列的 SQL 查询。

输出：如果视图创建成功，返回信息“CREATE”；否则，返回下列错误信息之一：

“ERROR: Relation 'view' already exists”，数据库中已经存在所指定的视图。

“NOTICE create: attribute named "column" has an unknown type”，要创建的视图中有一个未知类型的字段。例如，下面命令返回一个警告：

```
CREATE VIEW vista AS SELECT 'Hello World'
```

而下面命令将不出现警告：

```
CREATE VIEW vista AS SELECT text 'Hello World'
```

用法：

--创建一个由所有 Comedy（喜剧）电影组成的视图

```
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 为 CREATE VIEW 定义了一些附加的功能:

```
CREATE VIEW view [ column [, ...] ]
    AS SELECT expression [ AS colname ] [, ...]
    FROM table [ WHERE condition ]
    [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

完整的 SQL92 命令可选的子句包括:

- (1) CHECK OPTION。这个选项用于可更新视图。所有对视图的 INSERT 和 UPDATE 都要经过视图定义条件的校验。如果没有通过校验,更新将被拒绝。
- (2) LOCAL。对视图进行完整性检查。
- (3) CASCADE。对此视图和任何相关视图进行完整性检查。在既没有指定 CASCADE,也没有指定 LOCAL 情况下,假设为 CASCADE。

## 27. DECLARE (定义游标)

```
语法: DECLARE cursorname [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
      CURSOR FOR query
      [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] }
```

描述: 本命令允许用户创建游标,该游标用于在一个大的查询中检索少数几行数据。使用 FETCH,游标既可以返回文本,也可以返回二进制格式。

通常,游标返回数据的格式为文本格式、ASCII 格式,或者是某种由 PostgreSQL 特定的后端决定的编码格式。因为数据在系统内部是用二进制格式存储的,所以系统必须对数据进行一定的转换后才能生成文本格式。另外,文本格式一般都比对应的二进制格式占用的存储空间大。一旦格式转换回文本,客户应用需要将文本转换为二进制格式来操作。BINARY 选项要求游标以内部二进制的形式返回数据。

例如,如果查询从一个整数字段返回一个 1,在通常的游标中,用户将获得一个字符串“1”。如果是一个二进制查询,用户将得到一个 4 字节的等于 ctrl-A (‘^A’) 的数值。

应该小心使用 BINARY 游标。一些应用程序如 psql 是不识别二进制游标的,而且期望返回的数据是文本格式。

字符串表示方式是与硬件体系无关的,但二进制格式会因硬件体系不同而不同,而且 PostgreSQL 对二进制游标不进行字节解析或者其他格式转换。因此,如果客户机和服务器使用不同的格式,用户可能不会希望数据以二进制格式返回。所以二进制游标将比文本略微快一点,因为二进制方式在服务器和客户端之间进行传输时需要的转换较少。

游标只能在事务中使用。使用 BEGIN、COMMIT 和 ROLLBACK 定义一个事务块。

在 SQL92 中,游标只能在嵌入 SQL (ESQL) 应用中使用。PostgreSQL 没有一个显式的 OPEN cursor 命令,一个游标被认为在定义时就已经打开了。不过,ecpg (PostgreSQL 嵌入的 SQL 预编译器) 支持 SQL92 的习惯。

输入:

cursorname——FETCH 操作中使用的游标名。

BINARY——令游标以二进制而不是文本格式返回数据。

INSENSITIVE——SQL92 关键字,表明从游标检索出来的数据不应该受其他进程或游标的更新动作影响。因为在 PostgreSQL 中,游标的操作总是发生在事务里,所以总是符合上面描述。这个关键字实际上没有什么意义。

SCROLL——SQL92 关键字,表明每个 FETCH 操作可以检索出多行数据。在 PostgreSQL 中,任何情况下都允许这样做,所以这个关键字也没有意义。

query——一个 SQL 查询,它提供由游标控制的数据行。

READ ONLY——SQL92 关键字,表明游标工作于只读模式。因为这是 PostgreSQL 唯一的游标访问模式,所以该关键字没有意义。

UPDATE——SQL92 关键字,表明游标将被用于更新。因为游标更新目前还不被 PostgreSQL 支持,所以这个关键字将产生一个错误信息。

column——将被更新的列。因为游标更新目前不被 PostgreSQL 支持,所以 UPDATE 子句将产生一个错误信息。

输出: 如果 SELECT 成功运行,返回信息“SELECT”;否则,返回下列错误信息之一:

“NOTICE BlankPortalAssignName: portal "cursorname" already exists”,如果 cursorname 已经定义,返回此信息。

“ERROR: Named portals may only be used in begin/end transaction blocks”,如果一个游标没有在事务块内部定义,返回此信息。

用法:

--定义一个游标

```
DECLARE liahona CURSOR
FOR SELECT * FROM films;
```

兼容性: SQL92 只允许在嵌入的 SQL 中和模块中使用游标。PostgreSQL 允许交互地使用游标。SQL92 允许嵌入或模块的游标更新数据库信息。所有 PostgreSQL 的游标都是只读的。BINARY 关键字是 PostgreSQL 的扩展。

## 28. DELETE (删除数据行)

语法: DELETE FROM table [ WHERE condition ]

描述: 本命令用于从指定的表中删除满足条件 condition 的行。

如果 WHERE 子句不存在,命令将删除表中所有行,结果是一个有效的空表。

如果要对表进行修改,用户必须具有写权限,同样也必须具有读权限,这样才能对符合条件的行进行读取操作。

输入:

table——表的名称。

condition——这是一个 SQL 选择查询,它返回被选取的行。

输出: 如果行被成功地删除,返回信息“DELETE count”,其中 count 为被删除的行数。如果 count 为 0,没有行被删除。

用法:

--删除所有电影 (films) 但不删除音乐 (musicals)

## 24 命令与工具

```
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 允许使用游标定位的 DELETE 命令:

DELETE FROM table WHERE

CURRENT OF cursor

其中 cursor 为一个已经打开的游标。

### 29. DROP AGGREGATE (删除聚集函数)

语法: DROP AGGREGATE name type

描述: 本命令用于删除聚集函数。执行这条命令的用户必须是该聚集函数的所有者。

输入:

name——聚集函数名。

type——聚集函数类型。

输出: 如果命令执行成功, 返回信息“ DROP ”; 否则, 返回信息“ WARN RemoveAggregate: aggregate 'agg' for 'type' does not exist ”, 表明函数在数据库中不存在。

用法:

--将类型 int4 的聚集函数 myavg 删除

```
DROP AGGREGATE myavg int4;
```

兼容性: SQL92 中没有 DROP AGGREGATE 命令, DROP AGGREGATE 命令是 PostgreSQL 的扩展。

### 30. DROP DATABASE (删除数据库)

语法: DROP DATABASE name

描述: 本命令用于删除一个数据库, 同时删除该数据库的目录。只有数据库所有者才

能执行这条命令。

值得注意的是，这条命令在和目标数据库连接时不能执行。通常更好的做法是用 `dropdb` 脚本代替，该脚本是这个命令的一个封装。

输入：

name——被删除数据库的名称。

输出：如果命令成功执行，返回信息“`DROP DATABASE`”；否则，返回下列错误信息之一：

“`ERROR: user 'username' is not allowed to create/drop databases`”，删除数据库的用户必须有特殊的 `CREATEDB` 权限。

“`ERROR: dropdb: cannot be executed on the template database`”，`template` 数据库不能被删除。

“`ERROR: dropdb: cannot be executed on an open database`”，无法与删除的数据库建立连接。

“`ERROR: dropdb: database 'name' does not exist`”，如果指定的数据库不存在，返回此信息。

“`ERROR: dropdb: database 'name' is not owned by you`”，执行本命令的用户必须是数据库所有者。数据库所有者通常也就是数据库的创建者。

“`ERROR: dropdb: May not be called in a transaction block.`”，执行这条命令之前必须先结束正在处理的事务。

“`NOTICE: The database directory 'xxx' could not be removed.`”，数据库已经被删除了，但是存储数据的目录无法删除，用户必须手工删除。

兼容性：`DROP DATABASE` 是一个 PostgreSQL 的扩展。在 SQL92 中没有 `DROP DATABASE`。

### 31. DROP FUNCTION（删除函数）

语法：`DROP FUNCTION name ([ type [, ...] ])`

描述：本命令用于删除一个函数。要执行这条命令，用户必须是函数的所有者。必须指定函数的输入参数类型，因为只有给定函数名和参数类型，函数才能被正确地删除。

本命令对依赖于该函数的类型、运算符和访问方式是否已被事先删除不进行任何校验。

输入：

name——函数名称。

type——函数参数的类型。

输出：如果命令成功执行，返回信息“`DROP`”；否则，返回错误“`WARN RemoveFunction: Function "name" ("types") does not exist`”，表明当前数据库中不存在指定的函数。

用法：

--这条命令删除平方根函数

```
DROP FUNCTION sqrt(int4);
```

兼容性：`DROP FUNCTION` 是 PostgreSQL 的扩展。

### 32. DROP GROUP（删除用户组）

语法：DROP GROUP name

描述：本命令用于从数据库中删除指定的组。组中的用户不被删除。

使用 CREATE GROUP 增加新组，使用 ALTER GROUP 修改组的成员。

输入：

name——组名称。

输出：如果组被成功地删除，返回信息“DROP GROUP”。

用法：

--删除一个名为 staff 的组

```
DROP GROUP staff;
```

兼容性：SQL92 中没有 DROP GROUP。

### 33. DROP INDEX（删除索引）

语法：DROP INDEX index\_name

描述：本命令用于从数据库中删除一个索引。执行这条命令的用户必须是索引的所有者。

DROP INDEX 是 PostgreSQL 的扩展。

请参考 CREATE INDEX 命令获取如何创建索引的信息。

输入：

index\_name——要删除的索引名。

输出：如果索引删除成功，返回信息“DROP”；否则，返回信息“ERROR: index “index\_name” nonexistent”，表明 index\_name 不是数据库的索引。

用法：

--删除 title\_idx 索引

```
DROP INDEX title_idx;
```

兼容性：索引是一个与具体实现相关的特性，因此在 SQL92 语言中没有给出具体的定义。

### 34. DROP LANGUAGE（删除语言）

语法：DROP PROCEDURAL LANGUAGE 'name'

描述：本命令用于删除注册过的过程语言 name。

DROP PROCEDURAL LANGUAGE 命令是 PostgreSQL 的扩展。

请参考 CREATE LANGUAGE 获取如何创建过程语言的信息。

本命令不校验用这种语言注册的函数或触发器是否仍然存在。要想重新使用这些东西而不用删除和重新创建所有这些函数，函数 pg\_proc 的 prolang 字段必须调整为 PL，并重新创建新对象标识号（OID）。

输入:

name——语言的名称。

输出: 如果语言删除成功, 返回信息“ DROP ”; 否则, 返回错误信息“ ERROR: Language "name" doesn't exist ”, 表明没有找到指定的语言。

用法:

--下面命令删除 PL/Sample 语言

```
DROP PROCEDURAL LANGUAGE 'plsample';
```

兼容性: SQL92 中没有 DROP PROCEDURAL LANGUAGE 。

### 35. DROP OPERATOR (删除运算符)

语法: DROP OPERATOR id ( type | NONE [...])

描述: 本命令用于从数据库中删除一个运算符。执行这条命令的用户必须是运算符所有者。

左目运算符的右类型或右目运算符的左类型可以指定为 NONE。

DROP OPERATOR 命令是 PostgreSQL 的扩展。

请参考 CREATE OPERATOR 获取如何创建运算符的信息。

删除任何依赖于被删除的运算符的访问模式和运算符表是用户的责任。

输入:

id——运算符的标识号。

type——函数参数的类型。

输出: 如果命令执行成功, 返回信息“ DROP ”函数; 否则, 返回下列错误信息之一:

“ ERROR: RemoveOperator: binary operator 'oper' taking 'type' and 'type2' does not exist ”, 指定的二目运算符不存在。

“ ERROR: RemoveOperator: left unary operator 'oper' taking 'type' does not exist ”, 指定的左目运算符不存在。

“ ERROR: RemoveOperator: right unary operator 'oper' taking 'type' does not exist ”, 指定的右目运算符不存在。

用法:

--将用于 int4 的幂运算符 a^n 删除:

```
DROP OPERATOR ^ (int4, int4);
```

--删除用于逻辑变量的左目负号运算符 (b !):

```
DROP OPERATOR ! (none, bool);
```

--删除用于 int4 的右目介乘运算符 (! i):

```
DROP OPERATOR ! (int4, none);
```

兼容性: SQL92 中没有 DROP OPERATOR 命令。

### 36. DROP RULE (删除规则)

语法: DROP RULE name

描述：本命令用于从指定的 PostgreSQL 规则系统中删除一个规则。PostgreSQL 将立即停止使用这个规则，并且将它从系统表中清除出去。

DROP RULE 命令是 PostgreSQL 的扩展。

请参考 CREATE RULE 获取如何创建规则的信息。

一旦一个规则被删除了，该规则所写的历史记录信息将有可能被同时删除。

输入：

name——要删除的规则名称。

输出：如果规则删除成功，返回信息“ DROP ”；否则，返回信息“ ERROR: Rewrite GetRule EventRel: rule "name" not found ”，表明指定的规则不存在。

用法：

--删除重写规则 (rewrite rule) newrule

DROP RULE newrule;

兼容性：SQL92 中没有 DROP RULE 命令。

### 37. DROP SEQUENCE（删除序列）

语法：DROP SEQUENCE name [, ...]

描述：本命令用于从数据库中删除指定的序列号生成器。因为目前的序列是作为一个特殊的表来实现的，所以此命令的功能如同 DROP TABLE。

DROP SEQUENCE 命令是 PostgreSQL 的扩展。

请参考 CREATE SEQUENCE 命令获取如何创建一个序列的信息。

输入：

name——序列名称。

输出：如果序列删除成功，返回信息“ DROP ”；否则，返回信息“ WARN: Relation "name" does not exist. ”，表明指定的序列不存在。

用法：

--从数据库中删除序列 serial

DROP SEQUENCE serial;

兼容性：SQL92 中没有 DROP SEQUENCE。

### 38. DROP TABLE（删除表）

语法：DROP TABLE name [, ...]

描述：本命令用于从数据库中删除表。只有表或视图的所有者才能执行本命令。

如果被删除的表有索引，它们将首先被删除。索引的删除对所属表的内容没有任何影响。

请参考 CREATE TABLE 和 ALTER TABLE 获取如何创建或更改表的信息。

输入：

name——要删除的表或视图的名称。

输出：如果命令执行成功，返回信息“ DROP ”；否则，返回错误信息“ ERROR Relation

"name" Does Not Exist! ", 表明指定的表或视图在数据库中不存在。

用法:

--删除 films 和 distributors 表

```
DROP TABLE films, distributors;
```

兼容性: SQL92 为 DROP TABLE 定义了一些附加的功能:

DROP TABLE table { RESTRICT | CASCADE }

RESTRICT:

确保只有不存在相关视图或完整性约束的表才可以被删除。

CASCADE:

任何引用的视图或完整性约束都将被删除。

### 39. DROP TRIGGER (删除触发器)

语法: DROP TRIGGER name ON table

描述: 本命令用于删除一个触发器。执行这条命令的用户必须是触发器的所有者。

DROP TRIGGER 是 PostgreSQL 的扩展。

请参考 CREATE TRIGGER 获取如何创建触发器的信息。

输入:

name——触发器的名称。

table——表的名称。

输出: 如果触发器删除成功, 返回信息 " DROP "; 否则, 返回错误信息 " ERROR: DropTrigger: there is no trigger name on relation "table" ", 表明指定的触发器不存在。

用法:

--删除表 films 的 f\_dist\_exists 触发器

```
DROP TRIGGER if_dist_exists ON films;
```

兼容性: SQL92 中没有 DROP TRIGGER 命令。

### 40. DROP TYPE (删除数据类型)

语法: DROP TYPE typename

描述: 本命令用于从系统表中删除一个用户自定义类型。

只有类型所有者可以删除类型。

DROP TYPE 命令是 PostgreSQL 的扩展。

请参考 CREATE TYPE 获取如何创建类型的信息。

用户有责任删除任何使用了被删除类型的运算符、函数、聚集、访问模式、子类型和表。

如果删除了一个内建的类型, 后端的表现将不可预测。

输入:

typename——类型的名称。

输出: 如果命令执行成功, 则返回信息 " DROP "; 否则, 返回错误信息 " ERROR:



RemoveType: type 'typename' does not exist ", 表明没有找到指定的类型。

用法:

--删除 box 类型

DROP TYPE box;

兼容性: DROP TYPE 是 SQL3 命令。

## 41. DROP USER (删除用户)

语法: DROP USER name

描述: 本命令用于从数据库中删除指定的用户, 但不删除数据库中该用户所拥有的任何表、视图或其他数据库对象。如果该用户拥有任何数据库, 系统会报告一个错误信息。

使用 CREATE USER 增加新用户, 使用 ALTER USER 修改用户属性。PostgreSQL 还有一个脚本 dropuser, 这个脚本和这条命令功能相同 (实际上, 脚本就是调用这个命令), 但是可以在操作系统命令行上运行。

输入:

name——待删除的用户名称。

输出: 如果用户被成功删除, 返回信息 " DROP USER "; 否则, 返回下列错误信息之一:

" ERROR: DROP USER: user "name" does not exist ", 表明指定的用户名没有找到。

" DROP USER: user "name" owns database "name", cannot be removed ", 必须先删除数据库或者改变其所有者。

用法:

--删除一个用户帐户

DROP USER jonathan;

兼容性: SQL92 中没有 DROP USER 。

## 42. DROP VIEW (删除视图)

语法: DROP VIEW name

描述: 本命令用于从数据库中删除一个视图。执行这条命令的用户必须是视图的所有者。

PostgreSQL 的 DROP TABLE 命令同样也可用于删除视图。

请参考 CREATE VIEW 获取关于如何创建视图的信息。

输入:

name——视图的名称。

输出: 如果命令执行成功, 返回信息 " DROP "; 否则, 返回错误信息 " ERROR: RewriteGetRuleEventRel: rule "\_REtname" not found ", 表明在数据库中没有找到指定的视图。

用法:

--下面命令删除视图 kinds

DROP VIEW kinds;

兼容性: SQL92 为 DROP VIEW 定义了一些附加的功能:

DROP VIEW view { RESTRICT | CASCADE }

RESTRICT:

确保只有不存在关联视图或完整性约束的视图可以被删除。

CASCADE:

任何引用的视图和完整性约束都将被删除。

#### 43. END (提交并终止当前事务)

语法: END [ WORK | TRANSACTION ]

描述: END 是 PostgreSQL 的扩展, 在 SQL92 中与之兼容的同义命令是 COMMIT。

关键字 WORK 和 TRANSACTION 是多余的, 可以省略。

使用 ROLLBACK 回滚一个事务。

输入:

WORK, TRANSACTION——可选关键字, 没有作用。

输出: 如果事务提交成功, 返回信息 “ COMMIT ”; 否则, 返回错误信息 “ NOTICE: COMMIT: no transaction in progress ”, 表明没有正在处理的事务。

用法:

--有改变生效

END WORK;

兼容性: END 是 PostgreSQL 的扩展, 提供与 COMMIT 相同的功能。

#### 44. EXPLAIN (显示命令执行细节)

语法: EXPLAIN [ VERBOSE ] query

描述: 这条命令显示 PostgreSQL 规划器为查询生成的执行规划。执行规划显示查询引用的表是如何被扫描的(是简单的顺序扫描, 还是索引扫描等), 并且, 如果引用了多个表, 那么系统将采用什么样的联合算法从每个表中取出所需要的记录呢?

显示信息中最关键的部分是预计的查询执行开销, 这就是规划器对运行该查询所需时间的估计(以磁盘页面存取为单位计量)。实际上显示了两个成分: 返回第一条记录前的启动时间, 以及返回所有记录的总时间。对于大多数查询而言, 用户关心的是总时间。但是, 在某些环境下, 比如一个 EXISTS 子查询, 规划器将选择最小启动时间而不是最小总时间, 这是因为执行器在获取一条记录后总是要停下来。同样, 如果用一条 LIMIT 子句限制返回的记录数, 规划器会在最终的开销上做一个合理的折衷以计算哪个规划开销最省。

VERBOSE 选项强迫命令输出规划树在系统内部的完整内容, 而不仅仅是一个概要, 并且还将它发送给 postmaster 日志文件。通常这个选项只是在对 PostgreSQL 进行调试时才有用。

在 PostgreSQL 中, 只有很少的关于使用优化器开销的文档。通常关于查询优化开销的估算可以在数据库的手册中找到。请参考程序员手册中关于索引和查询优化器的章节以便

获取更多信息。

输入：

VERBOSE——显示详细查询规划的标志。

query——查询命令。

输出：

NOTICE: QUERY PLAN: plan——PostgreSQL 后端显式的查询规划。

EXPLAIN——查询规划显示后发送的标志。

用法：

--显示一个对只有一个 int4 字段和 128 行数据的表的简单查询规划

```
EXPLAIN SELECT * FROM foo;
```

```
NOTICE: QUERY PLAN:
```

```
Seq Scan on foo (cost=0.00..2.28 rows=128 width=4)
```

```
EXPLAIN
```

--对同一个拥有支持查询 equijoin 条件的索引表，EXPLAIN 显示一个不同的规划

```
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，同一个拥有支持查询 equijoin 条件的索引表，EXPLAIN 对使用一个聚集函数的查询将显示下面内容

```
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，这里显示的数字，甚至还有选择的查询策略都有可能各个 PostgreSQL 版本之间不同，这是因为规划器在不断改进。

兼容性：SQL92 中没有 EXPLAIN 命令。

## 45. FETCH（利用游标获取数据行）

语法：FETCH [ selector ] [ count ] { IN | FROM } cursor

FETCH [ RELATIVE ] [ { [ # | ALL | NEXT | PRIOR ] } ] FROM ] cursor

描述：本命令允许用户使用游标检索表的数据行，要检索的行数用#定义。如果游标中剩下的行数小于#，那么只有那些可用的行才会被读取出来。如果用关键字 ALL 代替数字，则会读取出游标中所有剩余的数据行。可以向前（FORWARD）读取记录，也可以向后

(BACKWARD) 读取记录。默认的方向是向前。

可以用负数作为行数，符号等同于颠倒读取数据的方向关键字。例如，FORWARD -1 等同于 BACKWARD 1。

FORWARD 和 BACKWARD 关键字是 PostgreSQL 的扩展，SQL92 语法也支持，在此命令的第二种形式中指定。

在游标中更新数据目前还不被 PostgreSQL 支持，因为将游标更新影射回基本表是不太可能的，这一点对 VIEW 更新也一样。因此，用户必须显式地使用 UPDATE 命令来更新数据。

游标只能用于事务内部，因为它们存储的数据跨越了多个用户的查询。

使用 MOVE 命令改变游标位置；使用 DECLARE 命令定义一个游标；使用 BEGIN、COMMIT 和 ROLLBACK 命令定义或终止事务。

输入：

selector——数据读取方向。它可以是下述之一：FORWARD (读取后面的行，这是默认值)、BACKWARD (读取前面行)、RELATIVE (为 SQL92 兼容设置的多余关键字)。

count——读取的数据行数。可以是下列之一：

#：表明读取数据行数的整数。注意负整数等同于改变 FORWARD 和 BACKWARD 属性。

ALL：检索所有剩余的行。

NEXT：等同于将 count 设置为 1。

PRIOR——等同于将 count 设置为 -1。

cursor——一个打开的游标名称。

输出：FETCH 返回由指定游标定义的查询结果。如果查询失败，将返回下面的信息：

“NOTICE: PerformPortalFetch: portal "cursor" not found”，表明游标没有定义，游标必须在一个事务块中定义。

“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 允许使用下面命令在“当前位置”不停地检索游标：

```
FETCH RELATIVE 0 FROM cursor
```

PostgreSQL 目前不支持这种用法。实际上，零被保留用于检索所有行，相当于 ALL 关键字。如果使用 RELATIVE 关键字，PostgreSQL 假设用户试图使用 SQL92 的特性，因而返回此错误。

用法：

-- 下面的例子用一个游标跨过一个表

---- 建立一个游标

```
BEGIN WORK;
```

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

--读取头 5 行数据

```
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

--读取前面的数据行

```
FETCH BACKWARD 1 IN liahona;
```

code	title	did	date_prod	kind	len
P_301	Vertigo	103	1958-11-14	Action	02:08

--关闭游标并提交事务

```
CLOSE liahona;
```

```
COMMIT WORK;
```

兼容性：非嵌入式游标的使用是 PostgreSQL 的扩展。游标的语法、用途和定义与 SQL92 中定义的嵌入式用法相似。

SQL92 允许游标在 FETCH 中的绝对定位，并且允许将结果放在变量中。

```
FETCH ABSOLUTE #
```

```
FROM cursor
```

```
INTO :variable [, ...]
```

```
ABSOLUTE
```

其中，:variable 为目标宿主变量。上面命令将游标位置放置在指定的绝对行数上。在 PostgreSQL 中所有的行数都是相对数量，所以不支持这一功能。

#### 46. GRANT（授予访问权限）

语法：GRANT privilege [, ...] ON object [, ...]

```
TO { PUBLIC | GROUP group | username }
```

描述：本命令允许对象的创建者给某用户、某组或所有用户（PUBLIC）赋予某些特定的权限。一个对象创建后，除非创建者赋予权限，否则其他人没有访问该对象的权限。

一旦用户拥有某对象的权限，他就可以使用该特权。不需要给创建者赋予对象的权限，创建者自动拥有对象的所有权限，包括删除它的权限。

目前,要想在 PostgreSQL 中只赋予某用户几行数据的权限,必须创建一个包含那几列的视图,然后将权限赋予该视图。

使用 `psql \z` 命令可获取关于对象权限的更多信息:

```

Database      = lusitania
+-----+-----+
| Relation    | Grant/Revoke Permissions |
+-----+-----+
| mytable     | {"=rw","miriam=arwR","group todos=rw"} |
+-----+-----+

```

Legend:

```

      unname=arwR -- privileges granted to a user
group gname=arwR -- privileges granted to a GROUP
      =arwR -- privileges granted to PUBLIC

```

```

      r -- SELECT
      w -- UPDATE/DELETE
      a -- INSERT
      R -- RULE
      arwR -- ALL

```

REVOKE 命令回收访问权限。

输入:

privilege——权限。可用的权限包括:

```

SELECT——查询表或视图所有字段数据的权限。
INSERT——向指定表中插入所有字段数据的权限。
UPDATE——更新指定表中所有字段数据的权限。
DELETE——从指定表中删除数据的权限。
RULE——在表或视图上定义规则的权限。
ALL——所有权限。

```

object——赋予权限的对象名。可用的对象包括: table、view、sequence 等。

PUBLIC——所有用户的简写。

GROUP group——将要赋予权限的组名称。

username——将要赋予权限的用户名。PUBLIC 是代表所有用户的简写。

输出: 如果命令执行成功, 返回信息 “CHANGE”; 否则, 返回错误信息 “ERROR: ChangeAcl: class "object" not found”, 表明指定的对象不可用, 或不可能对指定的组或用户赋予权限。

用法:

```
--给所有用户赋予向表 films 插入记录的权限
```

```
GRANT INSERT ON films TO PUBLIC;
```

/ 赋予用户 manuel 操作视图 kinds 的所有权限

```
GRANT ALL ON kinds TO manuel;
```

兼容性: SQL92 GRANT 语法允许对表中的某单独字段设置权限, 并且具有向其他用户赋予权限的权限:

```
GRANT privilege [, ...]
```

```
ON object [ ( column [, ...] ) ] [, ...]
```

```
TO { PUBLIC | username [, ...] } [ WITH GRANT OPTION ]
```

这些字段与 PostgreSQL 实现是兼容的, 下面一些情况例外:

privilege——SQL92 允许指定附加的权限, 包括 SELECT、REFERENCES ( 允许在一个表的完整性约束中使用某些或全部字段 )、USAGE ( 允许使用一个域、字符集、集合或事务 )。如果指定的对象不是表或视图, privilege 只能指定为 USAGE。

object——SQL92 允许指定附加的目标, 包括 [ TABLE ] table ( 一个附加的非功能性关键字 TABLE )、CHARACTER SET ( 允许使用指定的字符集 )、COLLATION ( 允许使用指定的集合序列 )、TRANSLATION ( 允许使用指定的字符集转换 )、DOMAIN ( 允许使用指定的域 )、WITH GRANT OPTION ( 允许向别人赋予同样权限 )。

## 47. INSERT (插入数据行)

语法: INSERT INTO table [ ( column [, ...] ) ]

```
{ VALUES ( expression [, ...] ) | SELECT query }
```

描述: 本命令允许用户向表中插入新的数据行。一次可以插入一行或多行作为查询结果。目标列表中的字段可以按任何顺序排列。

对于目标列表中没有出现的字段, 如果为该字段定义了默认值, 则插入默认值; 否则, 插入 NULL。如果向定义为 NOT NULL 的字段插入 NULL 值, PostgreSQL 将拒绝整个插入操作。

如果每行的表达式不是正确的数据类型, 系统将试图进行自动的类型转换。

如果要向表中插入数据, 用户必须具有该表的插入权限, 同样也必须具有选择权限, 以便处理 WHERE 子句中指定的任何表。

输入:

table——表的名称。

column——表中的字段名。

expression——赋予字段的一个有效表达式或值。

query——一个有效的查询命令。

输出: 如果只插入了一行数据, 返回信息 “INSERT oid 1”, 其中 oid 为被插入行的数字标识。如果插入的数据超过一行, 返回信息 “INSERT 0 #”, 其中 # 是插入的行数。

用法:

```
--向表 films 插入一行数据
```

```
INSERT INTO films VALUES
```

```
( 'UA502', 'Bananas', 105, '1971-07-13', 'Comedy', INTERVAL '82 minute' );
```

--在下面的例子中省略了字段 `date_prod`，因此该字段只存储默认的 `NULL` 值

```
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, DATE '1961-06-16', 'Drama');
```

--向表 `distributors` 插入一行数据，注意，只指定了字段 `name`，而没有指定的字段 `did` 将被赋予它的默认值

```
INSERT INTO distributors (name) VALUES ('British Lion');
```

--从表 `tmp` 中插入几行数据到表 `films` 中

```
INSERT INTO films SELECT * FROM tmp;
```

--插入数组

```
INSERT INTO tictactoe (game, board[1:3][1:3])
```

```
VALUES (1, '{{"", "", ""}, {"", ""}, {"", ""}}');
```

```
INSERT INTO tictactoe (game, board[3][3])
```

```
VALUES (2, '{}');
```

```
INSERT INTO tictactoe (game, board)
```

```
VALUES (3, '{,}, {,}, {,}, {,}');
```

兼容性：INSERT 命令与 SQL92 完全兼容。可能遇到的关于 query 子句特性限制在 SELECT 命令中有详细的介绍。

#### 48. LISTEN（监听信息）

语法：LISTEN name

描述：本命令用于将当前 PostgreSQL 后端注册为通知条件 `name` 的监听器。

当执行 NOTIFY name 命令后，不管是此后端进程还是其他连接到同一数据库的进程，所有正在监听此通知条件的后端进程都将收到该通知，并且接下来每个后端将通知与其相连的前端应用。

使用 UNLISTEN 命令，可以将在一个后端中注册的通知条件删除。同样，后端进程退出时自动删除该后端正在监听的已注册通知条件。

前端应用检测通知事件的方法取决于 PostgreSQL 应用使用的编程接口。如果使用基本的 libpq 库，前端应用将 LISTEN 当作普通 SQL 命令使用，而且必须周期地调用 PQnotifies 过程来检测是否有通知到达。其他像 libpqctl 接口提供了更高级的控制通知事件的方法。实际上，使用 libpqctl 接口时，应用程序员不应该直接使用 LISTEN 或 UNLISTEN 命令。

NOTIFY 的 man 手册页中包含有更广泛和更深入的关于 LISTEN 和 NOTIFY 使用方法的讨论。

name 是任何可以作为名称的有效字符串，它不需要与任何实际表相对应。如果 name 被双引号包围，它甚至可以不是一个有效的命令串，而是任何小于 31 字符的字符串。

在一些老的 PostgreSQL 版本中，如果 name 不与任何现有的表名对应就必须用双引号括起来。现在不再有这个限制了。

输入：

name——通知条件的名称。



输出: 如果注册成功, 返回信息“LISTEN”; 否则返回错误信息“NOTICE Async\_Listen: We are already listening on name”, 表明后端已经注册了该通知条件。

用法:

--在 psql 中配制和执行一个监听、通知序列

LISTEN virtual;

NOTIFY virtual;

Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.

兼容性: SQL92 中没有 LISTEN。

## 49. LOAD (动态装载对象文件)

语法: LOAD 'filename'

描述: 本命令用于装载一个对象 (或 ".o") 文件到 PostgreSQL 后端的地址空间。一旦一个文件被装载, 该文件内所有函数都可以被访问。这个功能用于支持用户自定义的类型和函数。

如果一个文件没有使用 LOAD 装载, PostgreSQL 将在函数第一次调用时自动装载该文件。LOAD 还可用于一个重新编辑和编译后的目标文件的重新装载。目前只支持用 C 语言创建的对象文件。

被装载对象文件内的函数不应该调用其他通过 LOAD 命令装载的对象文件内部的函数。例如, 所有文件 A 内的函数可以互相调用, 可以调用标准库、数学库中的函数或 PostgreSQL 自身内部的函数, 但它们不能调用定义在另一个装载文件 B 内的函数。这是因为如果 B 被重装载, PostgreSQL 装载器将不能够重新定位从 A 中函数调入 B 中函数的地址空间。但是如果 B 没有重装载, 就不会有问题。

对象文件必须编译成位置无关代码。例如, 在 DECstations 机器上, 必须使用带-G 0 选项的/bin/cc 编译将要装载的对象文件。

输入:

filename——用于动态装载的对象文件。

输出: 如果命令执行成功, 返回信息“LOAD”; 否则, 返回错误信息“ERROR: LOAD: could not open file 'filename'”, 表明指定的文件没有找到。文件必须可被 Postgres 后端读写, 要避免此信息, 应该指定合适的全路径。

用法:

--装载文件/usr/PostgreSQL/demo/circle.o:

LOAD '/usr/PostgreSQL/demo/circle.o'

兼容性: SQL92 中没有 LOAD。

## 50. LOCK (锁定表)

语法: LOCK [ TABLE ] name

LOCK [ TABLE ] name IN [ ROW | ACCESS ] { SHARE | EXCLUSIVE } MODE

**LOCK [ TABLE ] name IN SHARE ROW EXCLUSIVE MODE**

描述: PostgreSQL 在可能的情况下尽可能使用最小约束的锁模式。LOCK TABLE 在需要时提供更有约束力的锁。

RDBMS 锁定使用下面术语:

EXCLUSIVE——排它锁, 防止其他(事务)锁的产生。

SHARE——允许其他(事务)共享锁。

ACCESS——锁定表结构。

ROW——锁定独立的数据行。

如果没有指定 EXCLUSIVE 或 SHARE, 系统则假设锁的模式为 EXCLUSIVE。锁存在于事务周期内。

例如, 一个应用在 READ COMMITTED 隔离级别上运行事务, 并且它需要保证表中的数据在事务的运行过程中都存在。要实现这一点, 可以在查询之前对表使用 SHARE 锁模式进行锁定。这样, 系统将保护数据不被并行修改并且为任何更进一步的对表的读操作提供实际状态的数据。因为 SHARE 锁模式与任何写操作需要的 ROW EXCLUSIVE 模式冲突, 并且 LOCK TABLE name IN SHARE MODE 命令将等到所有并行的写操作提交或回滚后才执行。

当在 SERIALIZABLE 隔离级别运行事务, 而且需要读取真实状态的数据时, 必须在执行任何 DML 命令(这时事务定义什么样的并行修改对它自己是可见的)之前, 运行一个 LOCK TABLE 命令。

除了上面的要求外, 如果一个事务准备修改一个表中的数据, 那么应该使用 SHARE ROW EXCLUSIVE 锁模式以避免死锁情况。当两个并发的事务试图以 SHARE 锁模式锁住表。更改表中的数据时, 两个事务都隐含需要 ROW EXCLUSIVE 锁模式, 而此模式与并发的 SHARE 锁冲突。

应该遵循两个通用的规则以避免死锁条件:

(1) 事务应该以相同的顺序对相同的对象请求锁定。例如, 如果一个应用更新行 R1, 然后更新行 R2(在同一的事务里), 那么, 第二个应用如果稍后要更新行 R1 时不应该更新行 R2(在同一事务里)。相反, 它应该与第一个应用以相同的顺序更新行 R1 和 R2。

(2) 事务请求两个互相冲突的锁模式的前提是: 其中一个锁模式是自冲突的(也就是说, 一次只能被一个事务持有)。如果涉及多种锁模式, 那么事务应该总是最先请求最严格的锁模式。

PostgreSQL 不检测死锁, 并将回滚至少一个等待的事务以解决死锁。

LOCK 是 PostgreSQL 的扩展。

除了 ACCESS SHARE/EXCLUSIVE 锁模式外, 所有其他 PostgreSQL 锁模式和 LOCK TABLE 命令都与 Oracle 兼容。

LOCK 只在事务内部使用。

输入:

name——要锁定的表名称。

ACCESS SHARE MODE——访问共享模式。这是最小限制的锁模式, 只与 ACCESS EXCLUSIVE 模式冲突。它用于保护被查询的表免于被并发的 ALTER TABLE、DROP

TABLE 和 VACUUM 对同一表操作的命令修改。这个锁模式对被查询的表自动生效。

ROW SHARE MODE——行共享模式。与 EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。任何 SELECT FOR UPDATE 命令执行时自动使用本模式。它是一个共享锁，以后可能更新为 ROW EXCLUSIVE 锁。

ROW EXCLUSIVE MODE——行互斥模式。与 SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。任何 UPDATE、DELETE、INSERT 命令执行时自动使用本模式。

SHARE MODE——共享模式。任何 CREATE INDEX 命令执行时自动使用本模式，它与 ROW EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。这个模式防止一个表被并行更新。

SHARE ROW EXCLUSIVE MODE——这个模式类似 EXCLUSIVE MODE，但是允许其他事务的 SHARE ROW 锁。它与 ROW EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。

EXCLUSIVE MODE——这个模式同样比 SHARE ROW EXCLUSIVE 更有约束力，它阻塞所有并行的 SELECT FOR UPDATE 查询，并与 ROW SHARE、ROW EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 模式冲突。

ACCESS EXCLUSIVE MODE——访问互斥模式。命令 ALTER TABLE、DROP TABLE 和 VACUUM 执行时自动使用本模式。这是最严格的约束锁，它与所有其他的锁模式冲突并且保护一个被锁定的表不被任何其他并行的操作更改。

输出：如果锁定成功，返回信息“LOCK TABLE”；否则，返回错误信息“ERROR name: Table does not exist.”，表明指定的表不存在。

用法：

-- 一个在向一个外键表上插入时有主键的表上使用的 SHARE 锁

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
WHERE name = 'Star Wars: Episode I - The Phantom Menace';
```

-- 如果记录没有返回则回滚

```
INSERT INTO films_user_comments VALUES
(_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

-- 在执行删除操作时对一个有主键的表进行 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 中没有 LOCK TABLE，可以使用 SET TRANSACTION 来指定当前事

务的级别。

## 51. MOVE（移动游标位置）

语法：MOVE [ selector ] [ count ]  
           { IN | FROM } cursor

描述：本命令允许用户对游标进行行的移动。MOVE 的工作类似于 FETCH 命令，但只是定位游标而不返回数据行。

请参考 FETCH 命令以便获取语法和参数的详细信息。

MOVE 是 PostgreSQL 的扩展。

请参考 FETCH 获取有效参数的描述。使用 DECLARE 定义游标。请参考 BEGIN、COMMIT 和 ROLLBACK 命令获取关于事务的详细信息。

用法：

--设置和使用一个游标

```
BEGIN WORK;
```

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

--忽略开头 5 行数据

```
MOVE FORWARD 5 IN liahona;
```

```
MOVE
```

--读取游标 liahona 中的第六行数据

```
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 中没有 MOVE 命令。但是，SQL92 允许从一个绝对游标位置读取数据行，该操作隐含地将游标移动到正确位置。

## 52. NOTIFY（发送监听消息）

语法：NOTIFY name

描述：本命令用于向当前数据库中所有执行过 LISTEN name、正在监听特定通知条件的前端应用发送一个通知事件。

传递给前端的通知事件包括通知条件名和发出通知的后端进程的 PID。数据库设计者有责任定义用于某个数据库的条件名和每个通知条件的含义。

通常，通知条件名与数据库中表的名字相同，通知时间实际上意味着“我修改了此数据库，请看一看有什么新东西”。但 NOTIFY 和 LISTEN 命令并不强制这种联系。例如，数据库设计者可以使用几个不同的条件名来标识一个表的几种不同改变。

NOTIFY 为访问 PostgreSQL 数据库的一组进程提供了一种简单的信号形式或 IPC（进程间通信）机制。更高级的机制可以通过使用数据库中的表从通知者传递数据到被通知者。

当 NOTIFY 用于通知某一特定表修改的动作发生，一个实用的编程技巧是将 NOTIFY 放在一个由表更新触发的规则里。用这种方法，通知将在表更新的时候自动触发，而且应用程序员不会忘记处理它。

NOTIFY 和 SQL 事务用某种重要的方法进行交换。首先，如果 NOTIFY 在事务内部执行，通知事件直到事务提交才会送出。这样做的原因是，如果事务退出了，用户希望在其中的所有命令都丧失功能，包括 NOTIFY，但有时用户希望通知事件能够及时发送。其次，当一个正在监听的后端在一次事务内收到一个通知信号，直到本次事务完成（提交或回滚）之前，该通知事件将不被送到与之相连的前端。同样，如果一个通知在事务内部发送出去了，而该事务稍后又退出了，用户就希望通知可以在某种程度上被撤消，但通知一旦发送出去，后端便不能从前端“收回”通知。所以，通知时间只是在事务之间传递。这一点就要求使用 NOTIFY 作为实时信号的使用时，应该确保一个事务尽可能短。

NOTIFY 在某方面的表现类似于 UNIX 的信号。如果同一条件名在短时间内发出了多条信号，接收者几次执行 NOTIFY 可能只回收到一条通知信息。所以依赖于收到的通知条数的方法是很不可靠的。因而，使用 NOTIFY 唤醒需要关注某事的应用，同时还要使用数据库对象（如序列号）来跟踪事件发生了几次。

前端经常会自己发送与正在监听的通知名一样的 NOTIFY。这时，它（前端）也和其他正在监听的前端一样收到一个通知事件。这样可能导致一些无用的工作（与应用逻辑有关）。例如，对前端刚写过的表又进行一次读操作以发现是否有更新。在 Postgres 6.4 或更新的版本中，可以通过检查后端进程的 PID（在通知事件中提供）是否与自己后端的 PID 一致（从 libpq 中取得）。当它们一样时，说明这是其自身回弹的信息，可以忽略。这是一个安全技巧，PostgreSQL 能保持将自身的通知和其他的通知区分开来，所以即使屏蔽了自己的通知也不会丢失外部的通知。

name 可以是作为名称的任何字串，它不需要与任何实际的表的名称对应。如果用双引号将 name 括起，它甚至可以不是语法上有效的名称，可以是任何小于 31 字符长的字串。

在以前的 PostgreSQL 版本，name 如果不与任何现有表的名字对应，就必须用双引号引起来。现在不需要这样做了。

在早于 6.4 版本的 PostgreSQL 中，送出通知信息的后端进程的 PID，总是前端自己的后端 PID。所以在早期版本里，不可能将自身的通知信息与别的客户端的通知信息区分开。

输入：

name——生成信号（通知）的通知条件。

输出：

NOTIFY——确认通知命令已经执行了。

Notify events——事件发送给正在监听的前端，前端是否响应或怎样响应取决于它自身的程序。

用法:

--在 psql 中配置和执行一个监听、通知对

=> LISTEN virtual;

=> NOTIFY virtual;

Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.

兼容性: SQL92 中没有 NOTIFY 命令。

### 53. REINDEX (重建索引)

语法: REINDEX { TABLE | DATABASE | INDEX } name [ FORCE ]

描述: 本命令用于恢复损坏了的系统索引。为了运行 REINDEX 命令, 必须关闭 Postmaster, PostgreSQL 必须带-O 和-P (一个忽略系统索引的选项) 选项运行。不能依赖系统索引来恢复系统索引。

输入:

TABLE——重新建立指定表的所有索引。

DATABASE——恢复指定数据库的所有系统索引。

INDEX——重新建立指定的索引。

name——要重建的表、数据库、索引的名称。

FORCE——强制性重新建立索引。如果没有这个关键字, 除非目标索引非法, 否则 REINDEX 不做任何事情。

输出: 如果索引重建成功, 返回信息 “REINDEX”。

用法:

--重建表 mytable 的索引

REINDEX TABLE mytable;

--更多例子

REINDEX DATABASE my\_database FORCE;

REINDEX INDEX my\_index;

兼容性: SQL92 中没有 REINDEX。

### 54. RESET (重置运行参数)

语法: RESET variable

描述: 本命令用于将变量恢复为默认值。请参考 SET 命令以便获取允许的变量值和默认值的详细信息。RESET 是 SET variable = DEFAULT 命令的一个变种。

使用 SET 和 SHOW 命令设置、显示值。

输入:

variable——变量的名称。请参考 SET 命令以便获取有关可用的参数的详细说明。

输出: 如果 variable 成功地设置为默认值, 返回信息 “RESET VARIABLE”。

用法:

--将 DateStyle 重新设置为默认值

```
RESET DateStyle;
```

--将 GEQO 重新设为默认值

```
RESET GEQO;
```

兼容性：SQL92 中没有 RESET。

## 55. REVOKE（回收访问权限）

语法：REVOKE privilege [, ...]

ON object [, ...]

FROM { PUBLIC | GROUP groupname | username }

描述：本命令允许对象的创建者回收以前赋予某用户、组或所有用户的权限。

使用 `psql \z` 命令获取现有对象权限的更多信息：

```
Database = lusitania
```

```
+-----+-----+
| Relation | Grant/Revoke Permissions |
+-----+-----+
| mytable  | {"=rw","miriam=arwR","group todos=rw"} |
+-----+-----+
```

`uname=arwR` 为赋予一个用户的权限，`group gname=arwR` 为赋予一个组（GROUP）的权限，`=arwR` 为赋予所有用户的权限。其中 `r` 表示 SELECT 权限，`w` 表示 UPDATE/DELETE 权限，`a` 表示 INSERT 权限，`R` 表示 RULE 权限，`arwR` 表示 ALL 权限（即所有权限）。

在早期的版本中，如果要创建一个组（GROUP），只能手工地向表 `pg_group` 插入数据：

```
INSERT INTO pg_group VALUES ('todos');
```

```
CREATE USER miriam IN GROUP todos;
```

目前的版本中已经有了 CREATE GROUP 命令，利用这个命令可以创建组。

输入：

privilege——权限。可用的权限包括：

SELECT：对表、视图的所有字段的访问权限。

INSERT：对表的所有字段插入数据的权限。

UPDATE：对表所有字段的更新权限。

DELETE：对表所有字段的删除权限。

RULE：在表、视图上定义规则的权限。

ALL：回收所有权限。

object——要回收权限的对象名。可用的对象包括：table、view、sequence。

group——被回收权限的组名称。

username——被回收权限的用户名称。使用 PUBLIC 关键字废除所有用户。

PUBLIC——回收所有用户的权限。

输出：如果命令执行成功，则返回信息“CHANGE”；否则，返回错误信息“ERROR”，表明指定的对象不可用或不可能回收组或用户的权限。

用法:

--回收所有用户对表 `films` 的插入权限

```
REVOKE INSERT ON films FROM PUBLIC;
```

--回收用户 `manuel` 对视图 `kinds` 的所有权限

```
REVOKE ALL ON kinds FROM manuel;
```

兼容性: SQL92 的 `REVOKE` 语法有一些附加的功能, 包括一些用于回收表中某些字段的权限:

```
REVOKE { SELECT | DELETE | USAGE | ALL PRIVILEGES } [, ...]
```

```
ON object
```

```
FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

```
REVOKE { INSERT | UPDATE | REFERENCES } [, ...] [ ( column [, ...] ) ]
```

```
ON object
```

```
FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

请参考 `GRANT` 命令获取独立字段的细节。

```
REVOKE GRANT OPTION FOR privilege [, ...]
```

```
ON object
```

```
FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

废除一个用户给其他用户赋予权限的权限。

可用的对象包括:

[ `TABLE` ]表、视图

`CHARACTER SET` 字符集

`COLLATION` 字符集

`TRANSLATION` 字符集

`DOMAIN` 域

如果用户 1 赋予用户 2 的权限带有 `WITH GRANT OPTION` 可选项, 然后用户 2 赋予用户 3 同样的权限, 则用户 1 可以使用 `CASCADE` 关键字废除用户 2 和 3 的权限。

如果用户 1 赋予用户 2 的权限带有 `WITH GRANT OPTION` 可选项, 然后用户 2 又将该权限赋予了用户 3, 则如果用户 1 试图使用 `RESTRICT` 关键字废除这个权限就会失败。

## 56. ROLLBACK (回滚当前事务)

语法: `ROLLBACK [ WORK | TRANSACTION ]`

描述: 本命令用于回滚当前事务并取消当前事务的所有更新。

使用 `COMMIT` 命令将一次事务成功停止, `ABORT` 是 `ROLLBACK` 的同义词。

输入: 无。

输出: 如果命令执行成功, 返回信息 “`ABORT`”; 否则, 返回错误信息 “`NOTICE: ROLLBACK: no transaction in progress`”, 表明当前进程没有任何事务。

用法:

--取消所有更改

```
ROLLBACK WORK;
```



兼容性：SQL92 只定义了两种形式的回滚命令：ROLLBACK 和 ROLLBACK WORK。

## 57. SELECT（查询数据）

语法：SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]  
 expression [ AS name ] [, ...]  
 [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new\_table ]  
 [ FROM table [ alias ] [, ...] ]  
 [ WHERE condition ]  
 [ GROUP BY column [, ...] ]  
 [ HAVING condition [, ...] ]  
 [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]  
 [ ORDER BY column [ ASC | DESC | USING operator ] [, ...] ]  
 [ FOR UPDATE [ OF class\_name [, ...] ] ]  
 LIMIT { count | ALL } [ { OFFSET | , } start ]

描述：本命令用于从一个或多个表、视图中返回数据行，结果是满足 WHERE 条件的所有数据行。如果省略了 WHERE 子句，命令返回表中的所有数据行。

DISTINCT 从查询的结果集中删除所有重复的行。ALL（默认）将返回所有结果行，包括重复的行。

DISTINCT ON 删除匹配所有表达式的行，只保留每个重复集合的第一行。注意，这里每个重复集的“第一行”是不可预料的，除非用 ORDER BY 来保证希望的行最先出现。例如，

```
SELECT DISTINCT ON (location) location, time, report
FROM weatherReports
ORDER BY location, time DESC;
```

上面的命令检索出每个地区最近的天气预报。但是，如果没有使用 ORDER BY 来强制每个地区按时间值降序排列，得到的将是每个地区不可预料的时间的报告。

HAVING 允许只选择那些满足指定条件的行组。

ORDER BY 导致返回的行按照指定的顺序排列。如果没有给出 ORDER BY，命令的输出将以系统认为开销最小的顺序产生。

UNION 运算符允许结果集是那些涉及到的查询所返回的行的集合。

INTERSECT 给出两个查询公共的行。

EXCEPT 给出存在于第一个查询而不存在于第二个查询的行。

FOR UPDATE 子句允许 SELECT 命令对选出的行执行排他锁定。

LIMIT 子句允许给用户返回查询结果的一个子集。

执行本命令的用户必须具有 SELECT 权限。

以下是命令中各子句的详细描述。

(1) WHERE 子句。WHERE 条件常见的形式为：

WHERE boolean\_expr

boolean\_expr 可以包含任意个逻辑表达式。通常表达式的形式为 expr cond\_op expr 或

log\_op expr。这里的 cond\_op 可以是关系比较符和条件运算符，包括=、<、<=、>、>=、<>以及 ALL、ANY、IN、LIKE 等，也可以是用户定义的运算符。log\_op 可以为 AND、OR、NOT。SELECT 将忽略所有 WHERE 条件不为 TRUE 的行。

(2) GROUP BY 子句。GROUP BY 产生一个分组的表，该表源于下面的子句：

```
GROUP BY column [, ...]
```

GROUP BY 将所有在组合的列上有同样值的行压缩成一行。如果存在聚集函数，这些聚集函数将计算每个组的所有行，并且为每个组计算一个独立的值；而如果没有 GROUP BY，聚集函数对选出的所有行计算出一个数值。使用 GROUP BY 时，除非在聚集函数中，否则 SELECT 输出表达式对任何非组合列的引用都是非法的，因为对一个非组合列可能会有多于一个的返回值。

一个在 GROUP BY 中的条目还可以是输出列的名称或者序号 (SELECT 表达式)，或者是一个以输入列的数值形成的任意表达式。当存在语义模糊时，GROUP BY 名称将被解释成为一个输入字段名称而不是一个输出字段名称。

(3) HAVING 子句。可选的 HAVING 条件有如下形式：

```
HAVING cond_expr
```

这里的 cond\_expr 与 WHERE 子句中的 cond\_expr 相同。

HAVING 子句迫使命令从查询结果集合中丢弃那些不符合 cond\_expr 的数据行。HAVING 与 WHERE 不同之处在于：WHERE 在应用 GROUP BY 之前过滤出单独的行，而 HAVING 过滤由 GROUP BY 创建的行。

除非在一个聚集函数里，否则在 cond\_expr 中引用的每个字段，都应该明确地指定为一个组的字段。

(4) ORDER BY 子句。这个子句的格式为：

```
ORDER BY column [ ASC | DESC ] [, ...]
```

其中 column 既可以是一个字段名，也可以是一个序数。序数指的是字段的顺序编号。这个特性可以使得对没有一个合适名称的字段的排序成为可能。但是，这一点可能永远也没有用，因为总是可以通过 AS 子句给一个要计算的字段赋予一个名称，例如：

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

column 还可以是任意表达式，包括没有出现在 SELECT 结果列表中的域，这是 PostgreSQL 对 SQL92 的扩展。因此下面的命令是合法的：

```
SELECT name FROM distributors ORDER BY code
```

请注意，如果一个 ORDER BY 条目是一个与结果匹配的字段和输入字段的简单名称，ORDER BY 就将它解释成结果字段的名称。这与 GROUP BY 在同样情况下的选择正相反。这样的不一致是由 SQL92 标准强制的。

可以给 ORDER BY 子句中的每个字段加入一个关键字 DESC (降序) 或 ASC (升序)，默认值为 ASC，也可以定义一个排序运算符来实现排序。ASC 等同于使用 '<'，而 DESC 等同于使用 '>'。

(5) UNION 子句。这个子句的格式为：

```
table_query UNION [ ALL ] table_query  
[ ORDER BY column [ ASC | DESC ] [, ...] ]
```

这里的 `table_query` 为任何没有 `ORDER BY` 或 `LIMIT` 子句的选择表达式。

`UNION` 运算符允许结果集是那些涉及到的查询所返回的结果集合。两个作为 `UNION` 直接操作数的 `SELECT` 必须生成相同数目的字段,并且对应的字段必须有兼容的数据类型。

默认情况下,`UNION` 的结果不包含任何重复的行(除非使用了 `ALL` 子句)。

同一 `SELECT` 命令中的多个 `UNION` 运算符是从左向右计算的。注意,`ALL` 关键字不一定是全局的,可能只是应用在当前表的结果上。

(6) `INTERSECT` 子句。本子句的语法格式为:

```
table_query INTERSECT table_query  
[ ORDER BY column [ ASC | DESC ] [, ...] ]
```

这里的 `table_query` 为任何没有 `ORDER BY` 或者 `LIMIT` 子句的选择表达式。

`INTERSECT` 给出两个查询公共的行。两个作为 `INTERSECT` 直接操作数的 `SELECT` 的结果必须有相同数目的字段,并且对应的字段必须有兼容的数据类型。

除非用圆括号指明顺序,否则同一 `SELECT` 命令中的多个 `INTERSECT` 运算符是从左向右计算的。

(7) `EXCEPT` 子句。本子句的语法格式为:

```
table_query EXCEPT table_query  
[ ORDER BY column [ ASC | DESC ] [, ...] ]
```

这里的 `table_query` 为任何没有 `ORDER BY` 或 `LIMIT` 子句的选择表达式。

`EXCEPT` 给出存在于第一个查询而不存在于第二个查询的行。两个作为 `EXCEPT` 直接操作数的 `SELECT` 的结果必须有相同数目的字段,并且对应的字段必须有兼容的数据类型。

除非用圆括号指明顺序,否则同一 `SELECT` 命令中的多个 `EXCEPT` 运算符是从左向右计算的。

(8) `LIMIT` 子句。本子句的语法格式为:

```
LIMIT { count | ALL } [ { OFFSET | , } start ]  
OFFSET start
```

这里的 `count` 指定返回的最大行数,`start` 指定开始返回数据行之前忽略的行数。

`LIMIT` 允许有限制地从查询结果中取出一部分数据行。如果给出了限制数值,那么返回的行数不会超过该限制。如果给出了一个偏移量,那么在开始返回行之前会忽略该数值指定的数据行。

使用 `LIMIT` 时,一个好习惯是使用 `ORDER BY` 子句将结果行限制成唯一的顺序,否则可能会得到无法预料的查询子集。

在 PostgreSQL 7.0 中,查询优化器在生成查询规划时将 `LIMIT` 考虑进去了,所以很有可能因为给出的 `LIMIT` 和 `OFFSET` 值不同而得到不同的规划和不同的行序。因此,除非利用 `ORDER BY` 强制生成一个可以预料顺序的结果,否则不同的 `LIMIT/OFFSET` 值选择不同的查询结果子集将不会产生一致的结果。这实际上是 SQL 语言与生俱来的特点:除非使用 `ORDER BY` 约束顺序,否则 SQL 不保证查询结果的顺序。

输入:

`expression`——表的字段名或一个表达式。

`name`——使用 `AS` 子句为一个字段或一个表达式指定另一个名称。这个名称主要用于

字段的显示。它可以在 ORDER BY 和 GROUP BY 子句里代表字段的值。但是, name 不能用于 WHERE 或 HAVING 子句。

TEMPORARY、TEMP——如果指定了 TEMPORARY 或 TEMP, 则该表只在当前会话中唯一, 并且将在会话结束后自动删除。

new\_table——如果指定了 INTO TABLE 子句, 查询的结果将存储在指定名称的另一个表(即目标表)中。目标表(new\_table)将被自动创建并且在此命令执行之前不应该存在。

table——表的名称。

alias——正在处理的表 table 的别名, 用于缩写或消除一个表内部连接时的混乱。

condition——一个结果为真或假的逻辑表达式。

column——字段的名称。

select——除了 ORDER BY 和 LIMIT 子句外的所有特性的选择命令。

输出:

Rows——查询返回的所有结果集的行。

count——查询返回的行的记数。

用法:

--将表 films 和表 distributors 连接在一起

```
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

## 24 命令与工具

(17 rows)

--统计用 kind 分组的所有电影和组的字段长度和

```
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)

--统计所有电影 ( films ) 组的字段长度和, 用 kind 分组并且显示小于 5 小时的组总和

```
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)

--下面两个例子是根据第二列 ( name ) 的内容对单独的结果排序的常规方法

```
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)

```

--下面的例子演示如何获得表 distributors 和 actors 的连接，只将每个表中以字母 W 开头的取出来。因为只取了不相关的行，所以关键字 ALL 被省略了

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:

(1) PostgreSQL 允许在一个查询中省略 FROM 子句。这个特性是从最初的 PostQuel 查询语言保留下来的:

```

SELECT distributors.* WHERE name = 'Westwood';
 did | name
-----+-----
 108 | Westward

```

(2) 在 SQL92 规范中，关键字“AS”是多余的，可以忽略掉而不会对命令产生任何影响。PostgreSQL 分析器在重命名字段时需要这个关键字，因为类型扩展的特性会导致上下

文语意不清。

(3) DISTINCT ON 语法不是 SQL92 的标准，LIMIT 和 OFFSET 也不是。

(4) 在 SQL92 中，一个 ORDER BY 子句只可以作用于结果字段名称或者序号上，而 GROUP BY 子句只能用于输入字段上。PostgreSQL 将这两个子句都扩展为允许另一种选择，但是，如果出现冲突则使用标准解释。PostgreSQL 还允许两个子句使用任意的表达式。值得注意的是，在表达式中出现的名称将总是被当做输入字段的名称，而不是结果字段名称。

(5) SQL92 的 UNION 语法允许一个附加的 CORRESPONDING BY 子句：

```
table_query UNION [ALL]
    [CORRESPONDING [BY (column [,...])]]
table_query
CORRESPONDING BY 目前还不被 PostgreSQL 支持。
```

## 58. SELECT INTO（利用表创建表）

语法：SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]  
 expression [ AS name ] [, ...]  
 [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new\_table ]  
 [ FROM table [ alias ] [, ...] ]  
 [ WHERE condition ]  
 [ GROUP BY column [, ...] ]  
 [ HAVING condition [, ...] ]  
 [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]  
 [ ORDER BY column [ ASC | DESC | USING operator ] [, ...] ]  
 [ FOR UPDATE [ OF class\_name [, ...] ] ]  
 LIMIT { count | ALL } [ { OFFSET | , } start ]

描述：本命令利用一条查询命令创建一个新表。比较典型情况的是，这条查询命令从一个现有的表中提取数据。

CREATE TABLE AS 在功能上与 SELECT INTO 命令相同。

输入：

所有输入的参数在 SELECT 中都有详细描述。

输出：

所有输出结果在 SELECT 中都有详细描述。

## 59. SET（设置运行参数）

语法：SET variable { TO | = } { value | 'value' | DEFAULT }  
 SET CONSTRAINTS { ALL | constraintlist } mode  
 SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }  
 SET TRANSACTION ISOLATION LEVEL { READ COMMITTED |

SERIALIZABLE }

描述：本命令用于在一个会话过程中为变量修改配置参数。

当前值可以使用 SHOW 获得，而且可以使用 RESET 命令将各变量的值设置为默认值。参数和值都是大小写无关的。注意参数值总是用字符串来表达，所以必须用单引号括起来。

SET TIME ZONE 改变会话的默认时区偏移量。一个 SQL 会话总是以默认的时区偏移作为初始值。SET TIME ZONE 命令用于改变当前的 SQL 会话的默认时区偏移量。

SET variable 命令是 PostgreSQL 的扩展。

SHOW 和 RESET 命令显示或重置当前值。

输入：

variable——全局参数变量的名称。

value——参数的新值。DEFAULT 可以用于将参数恢复为默认值。允许使用字符串数组，但是更复杂的结构需要用单引号或者双引号引起来。

可用的变量和允许的值包括：

CLIENT\_ENCODING | NAMES：多字节客户端编码，其中 value 为客户端多字节编码值，此编码必须是后端所支持的编码。这一特性只有在 PostgreSQL 的配置中定义了 MULTIBYTE 后才有效。

DATESTYLE：日期/时间格式。可用的格式包括：ISO（ISO 8601 格式）、SQL（Oracle/Ingres 格式）、PostgreSQL（传统的 PostgreSQL 格式）、European（dd/mm/yyyy 格式）、NonEuropean（mm/dd/yyyy 格式）、German（dd.mm.yyyy 格式）、US（同 NonEuropean 格式）、DEFAULT（默认格式，同 ISO）。

日期格式可以用下面方法初始化：

设置 PGDATESTYLE 环境变量。如果一个基于 libpq 的客户端的环境变量中设置了 PGDATESTYLE，libpq 将在连接启动时自动将 DATESTYLE 设置为 PGDATESTYLE 的值。

用 -o -e 参数运行 postmaster，可以将日期设置成 European 格式。需要注意的是，这只是对一些日期风格的组合有影响，ISO 等类型的风格不受这个参数影响。

改变 src/backend/utils/init/globals.c 中的变量。修改如下：

```
bool EuroDates = false | true
```

```
int DateStyle = USE_ISO_DATES | USE_POSTGRESQL_DATES | USE_SQL_DATES |
USE_GERMAN_DATES
```

SEED：设置随机数生成器内部种子，其中 value 为种子的值，它由随机范围函数使用。典型的种子值是介于 0 和 1 之间的浮点数，这个数将被乘以 RAND\_MAX 再使用。如果使用了超出范围的数值，生成的积将会溢出。

种子还可以通过调用 setseed SQL 函数设置：

```
SELECT setseed(value);
```

这一特性只有在 PostgreSQL 的配置阶段定义了 MULTIBYTE 后才有效。

SERVER\_ENCODING：设置多字节服务器端编码。这一特性只有在 PostgreSQL 的配置阶段定义了 MULTIBYTE 后才有效。

CONSTRAINTS：定义当前事务约束计算的表现形式。在 SQL3 规范中，表现形式有：constraintlist（以逗号分隔的、可推迟的约束名列表）、mode 约束模式（允许值包括



DEFERRED 和 IMMEDIATE)。在 IMMEDIATE 模式下,外键约束在每个查询结束时检查。在 DEFERRED 模式下,标记为 DEFERRABLE 的外键约束只是在事务提交时或约束模式被显式地设为 IMMEDIATE 时才检查。

TIME\_ZONE, TIMEZONE: 设置时区。可用的时区值与操作系统有关。在 Linux 中, /usr/lib/zoneinfo 中包含时区的所有值。一般情况下, 可用的时区值包括: PST8PDT (California、加州)、Portugal (葡萄牙)、Europe/Rome (意大利的罗马)、DEFAULT (本地时区, 即 TZ 环境变量的值)。

如果指定了一个无效时区, 时区就会变成 GMT。

如果一个基于 libpq 客户端的环境中设置了 PGTZ, libpq 将在连接启动时自动将时区设置为 PGTZ 的值。

TRANSACTION ISOLATION LEVEL: 设置当前事务的隔离级别, 可用的隔离级别包括: READ COMMITTED (当前事务查询只读取在查询之前提交的数据, 此级别为默认值), SERIALIZABLE (当前事务查询只能读取在该事务中第一个 DML 命令执行前的数据)。

除了上面介绍的几种 SET 参数外, SET 命令还可以设置以下参数, 这些参数主要用于内部和优化:

PG\_OPTIONS: 设置各种后端参数。

RANDOM\_PAGE\_COST: 设置优化器对非连续磁盘页面读取的开销预计值。

CPU\_TUPLE\_COST: 设置优化器处理查询中每条记录的开销预计值。

CPU\_INDEX\_TUPLE\_COST: 设置优化器进行索引扫描时, 处理每条索引记录的开销预计值。

CPU\_OPERATOR\_COST: 设置优化器处理 WHERE 子句中每个运算符的开销预计值。

EFFECTIVE\_CACHE\_SIZE: 设置优化器对磁盘缓冲有效尺寸的假设。

ENABLE\_SEQSCAN: 打开或者关闭规划器对顺序扫描规划类型的使用 (ON/OFF)。

ENABLE\_INDEXSCAN: 打开或者关闭规划器对索引扫描规划类型的使用 (ON/OFF)。

ENABLE\_TIDSCAN: 打开或者关闭规划器对 TID 扫描规划类型的使用 (ON/OFF)。

ENABLE\_SORT: 打开或者关闭规划器对显式排序步骤的使用 (ON/OFF)。

ENABLE\_NESTLOOP: 打开或者关闭规划器对嵌套循环联合规划的使用 (ON/OFF)。

ENABLE\_MERGEJOIN: 打开或者关闭规划器对融合 (merge join) 联合规划的使用 (ON/OFF)。

ENABLE\_HASHJOIN: 打开或者关闭规划器对哈希 (散列) 规划的使用 (ON/OFF)。

GEQO: 为使用基因优化器算法设置阈值 (ON/OFF)。

KSQO: Key Set Query Optimizer (键集查询优化) 可使查询规划器将 WHERE 子句带有许多 OR 的 AND 子句的查询 (如 WHERE (a=1 AND b=2) OR (a=2 AND b=3) ...), 转换为一个 UNION 查询。这个方法可以比默认的实现更快, 但是它不一定得出相同的结果, 因为 UNION 隐含增加了一条 SELECT DISTINCT 子句以消除相同的行。KSQO 常与 Microsoft Access 等软件并用。可用的值为 ON (打开优化) 和 OFF (关闭优化)。

MAX\_EXPR\_DEPTH: 设置分析器可接受的最大表达式嵌套深度。默认值对任何常见查询都足够高了, 但是可以根据需要提高它。不过太深的嵌套深度, 可能会使堆栈溢出, 从而导致后端崩溃。

用法:

--设置时区风格为 ISO (参数不需要引号)

```
SET DATESTYLE TO ISO;
```

--对 4 个或更多表的查询,使 GEQO 生效

```
SET GEQO = 'ON=4';
```

--设置 GEQO 为默认值

```
SET GEQO = DEFAULT;
```

--将时区设置为 Berkeley, California

```
SET TIME ZONE "PST8PDT";
```

```
SELECT CURRENT_TIMESTAMP AS today;
```

```
today
```

```
-----
```

```
1998-03-31 07:41:21-08
```

--设置时区为 Italy

```
SET TIME ZONE 'Europe/Rome';
```

```
SELECT CURRENT_TIMESTAMP AS today;
```

```
today
```

```
-----
```

```
1998-03-31 17:41:31+02
```

兼容性: 在 SQL92 中没有 SET variable 命令,除了 SET TRANSACTION ISOLATION LEVEL 外。SQL92 中,用于 SET TIME ZONE 的语法有些不同,只允许为时区指定一个整数:

```
SET TIME ZONE { interval_value_expression | LOCAL }
```

## 60. SHOW (显示运行参数)

语法: SHOW keyword

描述: 本命令用于在会话中显示当前变量的配置参数。

会话可以通过 SET 命令来配置,而且使用 RESET 命令可以将数值恢复成默认值。参数和数值都是大小写无关的。

输入: keyword

请参考 SET 获取关于可用变量的信息。

输出: 如果找到了指定的变量,返回信息“NOTICE: variable is value”;否则,返回信息“NOTICE: Unrecognized variable value”,表明指定的变量不存在;或返回信息“NOTICE: variable is unknown”,表明没有设置指定的变量。

用法:

--显示当前 DateStyle 的设置

```
SHOW DateStyle;
```

```
NOTICE: DateStyle is ISO with US (NonEuropean) conventions
```

--显示当前基因优化器 (geqo) 设置

`SHOW GEQO;`

NOTICE: GEQO is ON beginning with 11 relations

兼容性: SQL92 中没有定义 SHOW。

## 61. TRUNCATE (清空表)

语法: `TRUNCATE [ TABLE ] name`

描述: 本命令用于快速地从表中删除所有的数据行, 它与无条件的 DELETE 命令有同样的效果。由于这条命令不进行表数据的扫描, 因此在执行速度上要比 DELETE 命令快得多, 尤其对数据量大的表更是如此。

输入:

name——要清空的表。

输出: 如果成功地清除了表的数据, 输出信息 “TRUNCATE”。

用法:

--清空表 bigtable

`TRUNCATE TABLE bigtable;`

兼容性: SQL92 中没有 TRUNCATE 命令。

## 62. UNLISTEN (停止信息监听)

语法: `UNLISTEN { notifyname | * }`

描述: 本命令用于删除一个已注册的消息监听, 它取消当前 PostgreSQL 会话中所有对通知条件 notifyname 的监听。特殊的条件通配符 “\*” 取消对当前会话的所有通知条件的监听。

classname 不必是一个有效的表名, 而可以是任何少于 32 个字符的有效字符串名。

如果对一个没有监听的事件执行 UNLISTEN, 后端不会报错。每个后端在退出时都会自动执行 UNLISTEN \*。

在一些早期的 PostgreSQL 版本中, 一个不代表具体表的 classname (表名) 必须使用双引号, 目前版本已经取消了这个限制。

输入:

notifyname——注册过的通知条件名称。

\*——清除所有后端监听。

输出: 如果命令执行成功, 则返回信息 “UNLISTEN”。

用法:

--提交一个现有的注册

`LISTEN virtual;`

LISTEN

`NOTIFY virtual;`

NOTIFY

```
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received
```

--一旦执行了 UNLISTEN, 以后的 NOTIFY 命令将被忽略

```
UNLISTEN virtual;
```

```
UNLISTEN
```

```
NOTIFY virtual;
```

```
NOTIFY
```

```
-- notice no NOTIFY event is received
```

兼容性: SQL92 中没有 UNLISTEN 命令。

### 63. UPDATE (更新数据)

语法: UPDATE table SET col = expression [, ...]

[ FROM fromlist ]

[ WHERE condition ]

描述: 本命令用于改变满足条件的所有数据行中的指定字段的值。只有要更改的字段才需要在命令中出现。

数组的引用语法与 SELECT 相同。单个数组元素、数组元素的某个范围或整个数组都可以用一个查询命令更新。

执行此命令的用户必须有写表数据的权限, 对 WHERE 条件中涉及的任何表也必须有读权限。

输入:

table——表名称。

column——字段名称。

expression——赋予字段的有效值或表达式。

fromlist——这是 PostgreSQL 的一个非标准的扩展, 允许其他表中的字段出现在 WHERE 条件中。

condition——请参考 SELECT 命令以便获得 WHERE 子句的进一步描述。

输出: 如果命令执行成功, 返回信息 “UPDATE #”, 其中, # 为更新的行数; 如果 # 等于 0, 则表明没有数据行被更新。

用法:

--将字段 kind 中的词 “Drama” 用 “Dramatic” 代替

```
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

兼容性: SQL92 对于 UPDATE 命令定义了一些不同的语法:

```
UPDATE table SET column = expression [, ...]
```

```
WHERE CURRENT OF cursor
```

其中, cursor 为一个已经打开的游标。

#### 64. VACUUM (清理和分析数据库)

语法:

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ table ]
```

```
VACUUM [ VERBOSE ] ANALYZE [ table [ (column [, ...]) ] ]
```

描述: 本命令在 PostgreSQL 中有两个用途, 一个是重新整理存储区, 另一个是为优化器收集信息。

VACUUM 打开数据库中每个表, 清除回滚事务记录, 并且更新系统目录中的统计信息。数据库系统维护的统计信息包括记录条数和所有表使用的存储页面数。

VACUUM ANALYZE 收集代表每一行数据的开销统计信息。当存在多个可能的执行路径时, 这些信息是非常珍贵的。

定期地运行 VACUUM 可以提高数据库处理用户查询的速度。

打开的数据库是 VACUUM 的目标。

建议在每天夜间清理数据库, 以保证统计数据足够新。不过, VACUUM 查询可以在任何时候进行, 尤其是在向 PostgreSQL 复制了一个大表或删除了大量记录后, 执行一个 VACUUM ANALYZE 查询是一个很好的习惯。这样做将更新系统目录为最近的状态, 并且允许 PostgreSQL 查询优化器在规划用户查询时有更好的选择。

输入:

VERBOSE——为每个表显示一份详细的清理工作报告。

ANALYZE——更新用于优化器的字段统计信息, 以决定执行查询的最有效方法。这些统计信息反映了每个字段的数据开销。当查询中有多条执行路径的可能时, 这些信息将是非常珍贵的。

table——要清理的表名称。默认情况下为所有表。

column——要分析的字段名称。默认情况下为所有字段。

输出: 如果命令被接受并且数据库被清理了, 系统输出下面的信息:

```
VACUUM
```

```
NOTICE: --Relation table--
```

```
//表 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.
```

```
//表 table 自身的分析。
```

```
NOTICE: Index index: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/0 sec.
```

```
//目标表的索引的分析。
```

用法:

--下面是一个在 regression 数据库中的某个表上执行 VACUUM 的一个例子:

```
vacuum verbose analyze onek;
NOTICE:  --Relation onek--
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.

NOTICE:  Index onek_stringul: Pages 28; Tuples 1000: Deleted 3000. Elapsed
0/0 sec.

NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 3000. Elapsed
0/0 sec.

NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 3000. Elapsed
0/0 sec.

NOTICE:  Index onek_uniquel: Pages 17; Tuples 1000: Deleted 3000. Elapsed
0/0 sec.

NOTICE:  Rel onek: Pages: 98 --> 25; Tuple(s) moved: 1000. Elapsed 0/1 sec.
NOTICE:  Index onek_stringul: Pages 28; Tuples 1000: Deleted 1000. Elapsed
0/0 sec.

NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 1000. Elapsed
0/0 sec.

NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 1000. Elapsed
0/0 sec.

NOTICE:  Index onek_uniquel: Pages 17; Tuples 1000: Deleted 1000. Elapsed
0/0 sec.

VACUUM
兼容性: SQL92 中没有 VACUUM 命令。
```

## 24.2 系统程序和工具

### 1. createdb (创建数据库)

语法: createdb [ options ] dbname [ description ]

描述: 本命令用于创建一个新的 PostgreSQL 数据库。执行此命令的用户为该数据库的管理员。

## 24 命令与工具

实际上, createdb 是一个 shell 脚本, 通过 PostgreSQL 前端工具 psql, 封装了 SQL 命令 CREATE DATABASE。这意味着, psql 必须能被脚本找到并且有一个数据库服务器在目标主机上运行。同样, 任何 psql 和 libpq 前端库可获得的默认设置和环境变量都将生效。

输入:

-h, --host host——正在运行 postmaster 的主机名。

-p, --port port——postmaster 侦听等待连接的 TCP/IP 端口或本地 UNIX 域套接字文件扩展 (描述符)。

-U, --username username——进行连接的用户名。

-W, --password——强制口令提示符。

-e, --echo——回显 createdb 生成的查询并且将它发送到后端。

-q, --quiet——不显示响应。

-D, --location datadir——数据库安装 (节点) 的位置。这是数据库的安装系统目录, 而不是指定的数据库位置。

-E, --encoding encoding——用于此数据库的字符编码方式。

dbname——要创建的数据库名。该名称应该在本节点的 PostgreSQL 所有数据库中是唯一的。默认的数据库名与当前系统用户同名。

description——这个选项定义一个与新创建的数据库相关的注解。

选项 -h, -p, -U, -W, 和 -e 将直接传递给 psql。

输出: 如果数据创建成功, 则返回信息“CREATE DATABASE”, 否则返回信息“createdb: Database creation failed.”。

用法:

--用默认数据库服务器创建一个数据库 demo

```
$ createdb demo
```

```
CREATE DATABASE
```

--响应信息与运行 CREATE DATABASESQL 命令时一样。

--用在主机 eden 上的 postmaster 创建数据库 demo, 端口是 5000, 使用 LATIN1 编码方式, 并且显示执行的查询

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
```

```
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
```

```
CREATE DATABASE
```

### 2. createlang (创建语言)

语法:

```
createlang [ connection options ] [ langname [ dbname ] ]
```

```
createlang [ connection options ] --list|-l
```

描述: 本命令用于创建一种新的编程语言。

输入:

langname——即将被定义的后端编程语言的名称。如果没有在命令行上指定, createlang 提示输入一个 langname。

`[-d, --dbname] dbname`——增加该语言的数据名称。

`-l, --list`——显示一个在目标数据库（必须指定）中已经安装的语言列表。

`createlang` 接受下列命令行参数作为连接参数：

`-h, --host host`——正在运行 `postmaster` 的主机名。

`-p, --port port`——`postmaster` 侦听的 TCP/IP 端口或本地 UNIX 域套接字文件扩展（描述符）。

`-U, --username username`——进行连接的用户名。

`-W, --password`——强制口令提示符。

输出：大多数错误信息都是可以在字面上理解其意义的。如果没有信息输出，可以带 `--echo` 参数运行 `createlang`，然后在相应的 SQL 命令下面检查细节；还可以参考 `psql` 获取更多的信息。

注意：

`createlang` 是一个用于向 PostgreSQL 数据库增加新的编程语言的工具。

`createlang` 目前接受两种语言，`plsql` 和 `pltcl`。

尽管可以用 SQL 命令直接增加后端编程语言，但还是推荐使用 `createlang`，因为它进行了一些检查而且更容易使用。参阅 `CREATE LANGUAGE` 获取更多信息。

使用 `droplang` 删除一种语言。

用法：

`--安装 pltcl`

```
$ createlang pltcl
```

### 3. `createuser`（创建用户）

语法：`createuser [ options ] [ username ]`

描述：本命令用于创建一个新的 PostgreSQL 用户。只有在 `pg_shadow` 表中拥有 `usesuper` 权限的用户才可以创建新的 PostgreSQL 用户。

`createuser` 是一个 shell 脚本，通过 PostgreSQL 前端工具 `psql`，封装了 SQL 命令 `CREATE USER`。这意味着 `psql` 必须能被脚本找到并且有一个数据库服务器在目标主机上运行。同样，任何 `psql` 和 `libpq` 前端库可获得的默认设置和环境变量都将生效。

输入：

`-h, --host host`——正在运行 `postmaster` 的主机名。

`-p, --port port`——`postmaster` 正在侦听的 TCP/IP 端口号或本地 UNIX 域套接字的文件扩展（描述符）。

`-e, --echo`——回显 `createdb` 生成的查询并发送给后端。

`-q, --quiet`——不显示响应。

`-d, --createdb`——允许该用户创建数据库。

`-D, --no-createdb`——禁止该用户创建数据库。

`-a, --adduser`——允许该用户创建其他用户。

`-A, --no-adduser`——禁止该用户创建其他用户。

`-P, --pwprompt`——给出此开关，`createuser` 将显示一个提示符要求输入新用户的口令。



如果不使用口令认证，这一可选项是不必要的。

-i, --sysid uid——允许给新用户使用非默认用户标识。这个也不是必须的，但是有些人习惯这样做。

username——要创建的 PostgreSQL 用户名称。该名称必须在所有 PostgreSQL 用户中唯一。

如果没有在命令行上指定名称或其他必要信息，脚本会提示输入这些信息。

选项-h、-p 和-e 将直接传递给 psql。psql 选项-U 和-W 也可以使用，但是这些开关的使用在这个环境中可能有些混乱。

输出：如果创建用户成功，则返回信息“CREATE USER”；否则，返回信息“createuser: creation of user "username" failed”，表明创建用户失败。

如果出现错误，命令将会显示后端错误信息。参阅 CREATE USER 和 psql 获取可能信息描述。

用法：

--在默认数据库服务器上创建一个用户 joe

```
$ createuser joe
```

```
Is the new user allowed to create databases? (y/n) n
```

```
Shall the new user be allowed to create more new users? (y/n) n
```

```
CREATE USER
```

--用在主机 eden 上的 postmaster 创建用户 joe，端口是 5000，避免提示并且显示执行的信息

```
$ createuser -p 5000 -h eden -D -A -e joe
```

```
CREATE USER "joe" NOCREATEDB NOCREATEUSER
```

```
CREATE USER
```

## 4. dropdb（删除数据库）

语法：dropdb [ options ] dbname

描述：本命令用于删除一个 PostgreSQL 数据库。执行这条命令的用户必须是数据库所有者，或者是数据库的超级用户。

dropdb 是一个 shell 脚本，通过 PostgreSQL 前端工具 psql，封装了 SQL 命令 drop\_database。这意味着 psql 必须能被脚本找到并且有一个数据库服务器在目标主机上运行。同样，任何 psql 和 libpq 前端库可获得的默认设置和环境变量都将生效。

输入：

-h, --host host——正在运行 postmaster 的主机名。

-p, --port port——postmaster 正在侦听的 TCP/IP 端口号或本地的 UNIX 域套接字文件句柄。

-U, --username username——进行连接的用户名。

-W, --password——强制口令提示符。

-e, --echo——回显 dropdb 生成的信息并且将它们发送到后端。

-q, --quiet——不显示响应。

-i, --interactive——在进行任何破坏性动作前给出提示。

dbname——要删除的数据库名。该数据库必须是该 PostgreSQL 节点中已经存在的数据库之一。

选项-h、-p、-U、-W 和-e 将直接传递给 psql。

输出：如果数据库删除成功，返回信息“DROP DATABASE”；否则，返回信息“dropdb: Database removal failed.”，表明无法删除指定的数据库。

如果出现错误，命令将会显示后端错误信息。参阅 drop\_database 和 psql 获取可能信息描述。

用法：

--删除默认数据库服务器上的数据库 demo

```
$ dropdb demo
```

```
DROP DATABASE
```

--用主机 eden 上的 postmaster 删除数据库 demo，端口是 5000，需要确认和显示执行信息

```
$ dropdb -p 5000 -h eden -i -e demo
```

```
Database "demo" will be permanently deleted.
```

```
Are you sure? (y/n) y
```

```
DROP DATABASE "demo"
```

```
DROP DATABASE
```

## 5. droplang（删除语言）

语法：

```
droplang [ connection options ] [ langname [ dbname ] ]
```

```
droplang [ connection options ] --list|-l
```

描述：本命令是一个从 PostgreSQL 数据库中删除一种现有编程语言的工具。

目前 droplang 接受两种语言：plsql 和 pltcl。

尽管可以用 SQL 命令直接删除后端编程语言，但还是推荐使用 droplang，因为它可进行一些检查而且更容易使用。参阅 DROP LANGUAGE 获取更多信息。

使用 createlang 增加一种语言。

输入：

langname——将要被删除的后端编程语言的名称。如果没有在命令行上指定，createlang 提示输入一个 langname。

[-d, --dbname] dbname——删除该语言的数据库名称。

-l, --list——显示在目标数据库中已经安装的语言列表。

droplang 接受下列命令行参数作为连接参数：

-h, --host host——正在运行 postmaster 的主机名。

-p, --port port——postmaster 正在侦听的 TCP/IP 端口或本地 UNIX 域套接字文件扩展（描述符）。

-U, --username username——进行连接的用户名。

-W, --password——强制口令提示符。

输出：大多数错误信息都是可以从字面上理解其意义的。如果没有明确的错误信息，

可带`--echo` 参数运行 `droplang`，然后在相应的 SQL 命令下检查细节。还可以参考 `psql` 获取更多的信息。

用法：

```
--删除 pltcl
$ droplang pltcl
```

## 6. dropuser (删除用户)

语法: `dropuser [ options ] [ username ]`

描述: 本命令用于删除一个现有 PostgreSQL 用户和该用户所有的数据库。只有在 `pg_shadow` 表中拥有 `usesuper` 集的用户才可以删除 PostgreSQL 用户。

`dropuser` 是一个 shell 脚本，通过 PostgreSQL 前端工具 `psql`，封装了 SQL 命令 `DROP USER`。这意味着 `psql` 必须能被脚本找到并且有一个数据库服务器在目标主机上运行。同样，任何 `psql` 和 `libpq` 前端库可获得的默认设置和环境变量都将生效。

输入：

-h, --host host——正在运行 postmaster 的主机名。

-p, --port port——postmaster 正在侦听的 TCP/IP 端口号或本地 UNIX 域套接字的文件扩展 (描述符)。

-e, --echo——回显 dropuser 生成的信息并发送给后端。

-q, --quiet——不显示响应。

-i, --interactive——在真正删除用户前给出提示。

username——要删除的 PostgreSQL 用户名，该名字必须存在于 PostgreSQL 节点。如果没有在命令行上指定该名称，`dropuser` 将提示用户输入。

选项-h、-p 和-e 将直接传递给 `psql`。`psql` 选项-U 和-W 也可以使用，但是这些开关的使用在这个环境中可能有些混乱。

输出: 如果用户成功地被删除，返回信息“`DROP USER`”；否则，返回信息“`dropuser: deletion of user "username" failed`”。

用法：

```
--删除默认数据库服务器上的用户 joe
```

```
$ dropuser joe
```

```
DROP USER
```

```
--用主机 eden 上的 postmaster 删除用户 joe，端口是 5000，避免提示并且显示执行信息
```

```
$ dropuser -p 5000 -h eden -i -e joe
```

```
User "joe" and any owned databases will be permanently deleted.
```

```
Are you sure? (y/n) y
```

```
DROP USER "joe"
```

```
DROP USER
```

7. `ecpg` (嵌入式 SQL C 语言预处理器)

语法: `ecpg [-v] [-t] [-I include-path] [-o outfile] file1 [file2] [...]`

描述: `ecpg` 是一个嵌入的用于 PostgreSQL 和 C 语言的 SQL 预编译器。它使得用嵌入的 SQL 代码书写 C 程序成为可能。

Linus Tolke 是 `ecpg` 最初的作者 (直到版本 0.2)。Michael Meskes 是目前 `ecpg` 的维护人员和作者。Thomas Good 是 `ecpg` 手册最新版本的作者。

输入:

`-v`——显示版本信息。

`-t`——关闭自动事务模式。

`-I path`——附加的包含路径。默认是“.”、`/usr/local/include`, PostgreSQL 包含路径是在编译时定义的。

`-o`——指定 `ecpg` 输出文件。如果没有给出这个选项, 输出信息将写入 `name.c`, 假设的输入文件是 `name.pgc`。

`file`——要处理的文件。

输出: `ecpg` 将创建一个文件或者写到标准输出( `stdout` )。命令成功执行后, `ecpg` 给 shell 返回一个 0, 如果出错则返回 -1。

用法:

--在编译一个嵌入了 SQL 命令的源文件前, 必须进行预处理

```
ecpg [-d] [-o file] file.pgc
```

--编译和链接

--假设 PostgreSQL 二进制文件目录为 `/usr/local/pgsql`, 需要像下面那样编译和链接预处理过的源文件

```
gcc -g -I /usr/local/pgsql/include [-o file] file.c -L /usr/local/pgsql/lib
-lecpg -lpq
```

--预处理器将包含两个目录到源文件里

```
#include <ecpgtype.h>
```

```
#include <ecpglib.h>
```

--变量定义

--在 `ecpg` 源程序里的变量定义必须前导

```
EXEC SQL BEGIN DECLARE SECTION;
```

--类似的, 变量定义段必须以下面命令结束

```
EXEC SQL END DECLARE SECTION;
```

--注意: 在 2.1.0 版以前, 每个变量都必须在一个独立的行中。2.1.0 版以后, 可以在一行中定义多个变量

```
char foo(16), bar(16);
```

--错误控制

--SQL 交互部分定义如下

```
EXEC SQL INCLUDE sqlca;
```

--注意:sqlca是小写。尽管可以使用SQL传统格式,也就是说,利用大写字母来区分嵌入的SQL和C命令,而sqlca(它包含sqlca.h头文件)必须小写。这是因为EXEC SQL前缀表明这个INCLUDE将被ecpg分析。ecpg区分大小写地检查头文件(SQLCA.h将不会被找到。)考虑大小写后,EXEC SQL INCLUDE还可以包含其他头文件。

--sqlprint 命令和EXEC SQL WHENEVER 命令一起使用以打开整个程序的错误控制

EXEC SQL WHENEVER sqlerror sqlprint;

--和

EXEC SQL WHENEVER not found sqlprint;

--注意:这些并不是EXEC SQL WHENEVER 命令全部用法的例子。其用法的更多的例子可以在SQL手册里找到。

--与数据库服务器连接

--一种连接使用命令

EXEC SQL CONNECT dbname;

--这里的数据库名称没有用引号引起。在2.1.0以前,数据库的名称要求被单引号引起。

--在连接命令中指定服务器名和端口名也同样可以。语法是:

--dbname[@server][:port]

--或者

--<tcp|UNIX>:postgresql:--server[:port][/dbname][?options]

--查询

--通常,其他应用,如psql,可以接收的SQL查询都可以嵌入C代码。下面是几个如何编程的例子。

--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;

--单字段选择:

EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';

--使用游标选择:

EXEC SQL DECLARE foo\_bar CURSOR FOR

SELECT number, ascii FROM foo

ORDER BY ascii;

EXEC SQL 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：这里没有 EXEC SQL PREPARE 命令。
--所有结构定义必须列在定义节里面。
```

#### 8. pgaccess (图形化交互客户端)

语法: pgaccess [ dbname ]

描述: pgaccess 为 PostgreSQL 提供了一个图形化的界面, 可以用它管理表, 编辑表, 定义查询、序列号和函数。

通过 tcl 访问 PostgreSQL 的另一个方法是使用 pgtklsh 或 pgtksh。

pgaccess 能够完成的管理工作包括:

- (1) 在指定主机、端口、用户名和口令的节点上打开任何数据库。
- (2) 执行 VACUUM。
- (3) 在 ~/.pgaccessrc 文件中保存客户化配置。

对于表, pgaccess 能够完成的工作包括:

- (1) 打开多个表进行查询, 最多可显示 n 条记录 (可配置)。
- (2) 通过拖动格栅竖直线改变列的大小。
- (3) 在单元中包含文本。
- (4) 编辑时动态调节行的高度。
- (5) 为每个表保存布局。
- (6) 输入/输出外部文件 (SDF、CSV)。
- (7) 使用过滤器功能, 如 price>3.14 之类的过滤器。
- (8) 指定排序顺序, 手工输入排序字段。
- (9) 现场编辑。
- (10) 删除记录。
- (11) 增加新记录。
- (12) 利用助手创建表。
- (13) 重命名和删除表。
- (14) 检索表的信息, 包括所有者、字段信息、索引等。

对于查询, pgaccess 能够完成的工作包括:

- (1) 定义、编辑和存储用户定制的查询。
- (2) 保存视图布局。
- (3) 将查询作为视图存储。
- (4) 带着可选的用户输入参数执行, 例如:

`select * from invoices where year=[parameter "Year of selection"]`

- (5) 查看任何选择查询的结果。
- (6) 进行动作查询（插入、更新、删除）。
- (7) 使用一个支持拖放的可视化的查询生成器构建查询、表别名。

对于序列，pgaccess 能够完成的工作包括：

- (1) 定义新实例。
- (2) 检查现有实例。
- (3) 删除。

对于视图，pgaccess 能够完成的工作包括：

- (1) 通过将查询存为视图来定义视图。
- (2) 带着过滤器和排序功能查看视图。
- (3) 设计新视图。
- (4) 删除现有视图。

对于函数，pgaccess 能够完成的工作包括：

- (1) 定义。
- (2) 查看。
- (3) 删除。

对于报表，pgaccess 能够完成的工作包括：

- (1) 从一个表中生成简单的报表。
- (2) 改变字段和标签的字体、大小和风格。
- (3) 从数据库中装载和保存报表。
- (4) 预览表，样例的 postscript 打印。

对于表单（forms），pgaccess 能够完成的工作包括：

- (1) 打开用户定义表单。
- (2) 使用一个表单设计模块。
- (3) 使用查询控件访问记录集。

对于脚本，pgaccess 能够完成的工作包括：

- (1) 定义。
- (2) 修改。
- (3) 调用用户定义脚本。

输入：

dbname——要访问的数据库名称。

输出：无

## 9. pgadmin（Windows 平台上的数据库管理和设计工具）

语法：pgadmin [ datasourcename [ username [ password ] ] ]

描述：pgadmin 是一个设计、维护和管理 PostgreSQL 数据库的通用工具。它运行在 Windows 95/98 和 NT 上。它的特性包括：

- (1) 执行任意的 SQL 命令。
- (2) 可用于数据库、表、索引、序列、视图、触发器、函数和语言的浏览器和构造器。
- (3) 用户、组和权限配置对话框。
- (4) 带有升级脚本生成功能的版本跟踪。
- (5) Microsoft MSysConf 表的配置。
- (6) 数据输入和输出向导。
- (7) 数据库迁移向导。
- (8) 对数据库、表、索引、序列、语言和视图预定义的报表。

pgadmin 是与 PostgreSQL 分开发布的, 可以从 <http://www.pgadmin.freemove.co.uk> 下载。

输入:

datasourcename——PostgreSQL ODBC 系统或者用户数据源 ( User Data Source ) 的名称。

username——datasourcename 上的有效用户名。

password——datasourcename 上的有效口令。

输出: 无

#### 10. pg\_ctl (启动、停止或重启 postmaster)

语法:

```
pg_ctl [-w] [-D datadir] [-p path] [-o "options"] start
```

```
pg_ctl [-w] [-D datadir] [-m [s|f|i]] stop
```

```
pg_ctl [-w] [-D datadir] [-m [s|f|i]]
```

```
[-o "options"] restart
```

```
pg_ctl [-D datadir] status
```

描述: pg\_ctl 是一个用于启动、停止和重启 postmaster 的 shell 脚本。

输入:

-w——通过监视 pid 文件 ( PGDATA/postmaster.pid ) 的创建, 等待数据库服务器的启动, 超时限值为 60 秒。

-D datadir——数据库的安装位置。

-p path——postmaster 映像的位置。

-o "options"——直接传递给 postmaster 的选项, 通常这些参数用单或双引号引起, 以保证它们以一组的方式传递。

-m mode——关闭模式。可选的模式包括: smart ( s, 智能模式, 等待所有客户端退出。这是默认模式 )、fast ( f, 快速模式, 向后端发送 SIGTERM 信号, 这意味着活跃的事务都将被回滚 )、immediate ( i, 立即模式, 向后端发送 SIGUSR1 信号, 并且让它们退出。在这种模式, 下一次启动必须进行数据库恢复工作 )。

start——启动 postmaster。

stop——关闭 postmaster。

restart——重新启动 postmaster, 相当于执行一次 stop 和 start。

status——显示 postmaster 的当前状态。



## 24 命令与工具

输出：如果启动成功，则返回信息“pg\_ctl: postmaster is state (pid: #)”，其中#为 postmaster 进程的 ID 号。如果出现错误，命令将显示出后端给出的错误信息。

用法：

--启动 postmaster

`pg_ctl start`

--如果使用了-w，pg\_ctl 将通过观察 pid 文件（PGDATA/postmaster.pid）的创建，等待数据库服务器的启动，最长等待时间为 60 秒。

--一个启动 postmaster，并等到 postmaster 启动后才退出的例子是：

`pg_ctl -w start`

--对一个使用端口 5433，并且运行在无磁盘同步模式的 postmaster，使用：

`pg_ctl -o "-o -F -p 5433" start`

--停止 postmaster

`pg_ctl stop`

--重启 postmaster

--这个功能几乎等同于停止 postmaster 然后再次启动它，但是，停止以前的参数还要使用。这是通过在\$PGDATA/postmaster.opts 文件里存储这些参数实现的。-w ,-D ,-m ,-fast ,-immediate 也可以在重启模式使用，并且和上面有一样的含义。

--用最简单的方法重启 postmaster：

`pg_ctl restart`

--重启 postmaster，等它停止并重新启动：

`pg_ctl -w restart`

--在 5433 端口重启并且重启后关闭 fsync：

`pg_ctl -o "-o -F -p 5433" restart`

--从 postmaster 获取状态信息：

`pg_ctl status`

--下面是 pg\_ctl 输出的一些信息

pg\_ctl: postmaster is running (pid: 13718)

options are:

/usr/local/src/pgsql/current/bin/postmaster

-p 5433

-D /usr/local/src/pgsql/current/data

-B 64

-b /usr/local/src/pgsql/current/bin/PostgreSQL

-N 32

-o '-F'

### 11. pg\_dump（导出数据库）

语法：

`pg_dump [ dbname ]`

```
pg_dump [ -h host ] [ -p port ]
        [ -t table ]
        [ -a ] [ -c ] [ -d ] [ -D ] [ -i ] [ -n ] [ -N ]
        [ -o ] [ -s ] [ -u ] [ -v ] [ -x ]
        [ dbname ]
```

描述: `pg_dump` 是一个将 PostgreSQL 数据库输出到一个包含查询命令脚本文件的工具。脚本文件为文本文件格式并且可以用于重建数据库,甚至可以在其他机器或其他硬件体系的机器上重建数据库。`pg_dump` 将输出用于重建用户定义的类型、函数、表、索引聚集和运算符所必须的查询命令。另外,所有数据是用文本格式复制出来的,因此可以很容易地复制回去,也很容易用工具进行编辑。

`pg_dump` 在从 PostgreSQL 节点向另一个节点转移数据时很有用。在运行 `pg_dump` 后,应该检查输出脚本中的任何警告。

`pg_dump` 有一些限制。限制主要源于从系统表中抽取某些专有信息的困难性。

`pg_dump` 不能理解部分索引。原因与上面所述相同,部分索引谓词被作为规划存储。

`pg_dump` 不能处理大对象。大对象将被忽略,因而必须进行手工操作。

当只进行数据输出时,`pg_dump` 使用查询在插入数据前关闭用户表上的触发器并在重新插入数据后恢复。如果重载在途中停止,系统表可能停留在错误的状态。

输入:

`dbname`——将要导出的数据库名。`dbname` 默认为 `USER` 环境变量的值。

`-a`——只输出数据,不输出表结构。

`-c`——创建前删除表定义。

`-d`——将数据输出为合适的插入字串。

`-D`——将数据作为带字段名的插入命令输出。

`-i`——忽略在 `pg_dump` 和数据库服务器之间的版本差别。因为 `pg_dump` 知道许多关于系统表的信息,任何给定版本的 `pg_dump` 只能和对应的数据库服务器版本一并使用。只有在需要跨越版本检查时,才使用这个选项。

`-n`——除非特殊要求,否则禁止标识范围的双引号。如果有保留字用于标识符,这么做有可能在装载输出的数据时导致麻烦。这是 6.4 版本以前的 `pg_dump` 默认特性。

`-N`——以双引号为标识范围。这是默认值。

`-o`——为每个表导出对象标识 (OID)。

`-s`——只导出表结构,不输出数据。

`-t table`——只导出表的数据。

`-u`——使用口令认证。提示输入用户名和口令。

`-v`——指定冗余模式。

`-x`——避免输出 ACL (赋予/撤消 命令) 和表的所有者关系信息。

`pg_dump` 同样接受下面的命令行参数作为连接参数:

`-h host`——正在运行 `postmaster` 的主机名。默认是使用本地 UNIX 主控套接字,而不是一个 IP 连接。

`-p port`——`postmaster` 正在侦听的 TCP/IP 端口或本地 UNIX 域套接字文件句柄。默认

的端口号是 5432，或者环境变量 PGPORT 的值（如果存在的话）。

-u——使用口令认证，提示输入用户名和口令。

输出：如果导出成功，pg\_dump 将创建一个文件或输出到标准输出（stdout）；否则，返回下面的信息之一：

“Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port 'port'? ”。pg\_dump 无法与指定主机和端口上的 postmaster 进程连接。如果看到这条信息，应该进一步确认 postmaster 是否正在给定的主机上的某个端口上运行。如果节点使用了认证系统，确认是否已经获取了认证所需的信息。

“Connection to database 'dbname' failed. FATAL 1: SetUserId: user 'username' is not in 'pg\_shadow' ”。执行命令的用户在 pg\_shadow 中没有有效的记录，因而不允许访问 PostgreSQL。

“dumpSequence(table): SELECT failed ”。执行命令的用户没有读数据库的权限。

用法：

--导出与用户名同名的数据库

`pg_dump > db.out`

--重载该数据库：

`psql -e database < db.out`

## 12. pg\_dumpall（导出所有数据库）

语法：

`pg_dumpall`

`pg_dumpall [-h host] [-p port] [-a] [-d] [-D] [-O] [-s] [-u] [-v] [-x]`

描述：pg\_dumpall 是一个用于将所有 PostgreSQL 数据库导出到一个文件中去的应用程序。它同时还输出全局表 pg\_shadow。pg\_dumpall 表包含一些命令可以在装载数据之前自动创建每一个输出的数据库。

pg\_dumpall 拥有所有的 pg\_dump 选项，但是 -f、-t 和 dbname 应该被忽略掉。

请参考 pg\_dump 获取这些功能的更多信息。

输入：

-a——只输出数据，不输出表结构。

-d——将数据输出为合适的插入字符串。

-D——将数据输出为带字段名的插入（命令）。

-n——除必需的以外，禁用标识符范围的双引号。如果标识符有保留字，这样做可能会在装载输出的数据时引起麻烦。

-o——为每个表输出对象标识（OID）。

-s——只输出表结构，不输出数据。

-u——使用口令认证，提示输入用户名和口令。

-v——指定冗余模式

-x——避免输出 ACL（赋予/撤消命令）和表的所有者关系信息。

pg\_dumpall 还接收下面的命令行参数用于连接参数：

-h host——运行 postmaster 的主机名。默认的使用一个本地 UNIX 主控套接字而不是一个 IP 连接。

-p port——postmaster 正在侦听的 TCP/IP 端口号或本地的 UNIX 主控套接字文件句柄。默认的端口号是 5432，或者是环境变量 PGPORT 的值（如果存在的话）。

-u——使用口令认证，提示输入用户名和口令。

输出：如果导出成功，pg\_dumpall 将创建一个文件或输出到标准输出（stdout）；否则，返回下面的信息之一：

“Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port 'port'? ”。pg\_dumpall 不能与指定的主机和端口号上的 postmaster 进程连接。如果看到这条信息，应该确认 postmaster 是否正在正确的主机上指定的端口运行。如果节点使用一个认证系统，应该确认是否已经获得了通过认证所需的信息。

“Connection to database 'dbname' failed. FATAL 1: SetUserId: user 'username' is not in 'pg\_shadow' ”。执行本命令的用户在 pg\_shadow 中没有有效的记录，因此不允许访问 PostgreSQL。

“dumpSequence(table): SELECT failed ”。执行本命令的用户没有读数据库的权限。

用法：

--导出所有数据库

```
pg_dumpall > db.out
```

--重新装载这个数据库

```
psql -e template1 < db.out
```

### 13. psql（基于命令行的交互式前端工具）

语法：psql [ options ] [ dbname [ user ] ]

描述：psql 是一个基于终端的 PostgreSQL 前端工具。它接收用户输入的查询命令，并将命令提交给 PostgreSQL 执行，然后显示查询结果。执行的命令也可以来自一个文件。另外，它还提供一些专门的命令（meta-commands）和许多类似 shell 风格的特性，供用户写脚本和完成某些自动化工作。

(1) 与一个数据库连接：

psql 是一个普通的 PostgreSQL 客户端应用程序。为了与数据库连接，用户需要指定目标数据库、服务器主机名和端口号，以及以哪个用户的身份进行连接等信息。可以通过命令行参数告诉 psql 这些信息，命令可选项分别是 -d、-h、-p 和 -U。如果某个参数不属于任何选项开关，那么它会被解释成为数据库名。不是所有选项都是必须的，默认也可以。如果省略主机名，psql 将通过域套接字与本地主机的服务器相连。默认的端口号是编译时确定的，因为所有 PostgreSQL 数据库服务器使用同样的默认值，所以在大多数情况下，不需要指定端口号。默认的用户名是当前用户的 UNIX 用户名，与数据库同名。需要注意的是，不能用任意用户名与任何数据库相连。

如果因为任何原因而无法与数据库相连，如权限不够、Postmaster 没有运行等，psql

将返回一个错误并退出。

(2) 输入查询：

通常情况下，psql 提供一个带有 psql 正在与之连接的数据库名的、后缀 "=>" 的提示符。例如：

```
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=>
```

用户可以在这个提示符下输入 SQL 命令。通常，输入的命令将在命令终止分号“；”出现时送到后端，回车键并不启动命令的执行。如果命令被正确地发送出去而且没有错误，查询结果会显示在屏幕上。例如：

当查询正在进行时，psql 同样监视由 LISTEN 和 NOTIFY 生成的异步通知消息。

(3) psql 专有命令：

在 psql 中输入的、任何以不带引号的反斜杠（“\”）开头的命令都是 psql 专有命令，这些命令并不会发送到 PostgreSQL 后端，而是由 psql 自己处理。这些命令也是令 psql 可用于管理或书写脚本的原因。通常，通俗地称专有命令为斜杠命令或反斜杠命令。

一个 psql 命令的格式是反斜杠后面紧跟一个命令动词，然后是参数。参数与命令动词以及其他参数之间以任意个空格字符分隔。

如果需要在参数中使用空格，则必须用单引号将空格包围起来。同样，如果需要在参数中使用单引号，应该在其前面加一个反斜杠。psql 会对任何包含在单引号内的信息进行替换，将\n（新行）、\t（tab）、\digits、\odigits 和\oxdigits（给出的十进制，八进制，或十六进制码的字符）替换掉。

如果一个不带引号的参数以冒号（“：”）开头，该参数会被当作一个变量，并且该变量的值为参数的真正值。

用“反单引号”（“`”）引起的内容被当作一个命令行传入 shell。该命令的输出（删除了结尾的新行）为参数值。上面描述的转意（字符）序列在反单引号内也有效。

有些命令以一个 SQL 标识的名称（如表名）为参数。这些参数遵循 SQL 语法中关于双引号的规则：不带双引号的标识强制成小写字母。对于所有其他命令，双引号没有特殊含义并且将被当成参数的一部分。

对参数的分析在遇到另一个不带引号的反斜杠时为停止。该处会被认为是一个新的专有命令的开始。特殊序列\\（双反斜杠）标识参数的结尾并将继续分析后面的 SQL 查询。这样，SQL 和 psql 命令可以自由地在一行内混合。但是，任何情况下一条专有命令的参数不能超过行尾。

下面是目前版本 `psql` 的专有命令。

`\a`——如果当前表输出格式为不对齐格式，本命令将格式切换为对齐格式。如果当前格式为对齐格式，则切换为不对齐格式。这条命令是为了向后兼容而设计的。参阅 `\pset` 获取一个通用的解决方法。

`\C [ title ]`——将正在输出的表标题设置为一个查询结果或者取消这样的设置。这条命令等同于 `\pset title title`。

`\connect (or \c) [ dbname [ username ] ]`——与一个新的数据库建立一个连接，在此之前的连接将关闭。如果 `dbname` 为 `-`，则建立与当前数据库的连接。如果省略 `username`，则假定连接用户名为当前用户名。

作为一条特殊规则，不带任何参数运行 `\connect` 时，系统将以默认用户身份与默认数据库连接，其效果与不带任何参数运行 `psql` 的效果一样。

如果连接失败，典型的原因如用户名错、访问拒绝等，那么先前的连接将被保留，但这仅仅是在 `psql` 处于交互模式下如此。如果运行的是非交互脚本，命令会立即停止，并返回一个错误信息。这样设计的目的，一方面是为了便于用户使用，另一方面为了保证安全。

`\copy table [ with oids ] { from | to } filename | stdin | stdout [ with delimiters 'characters' ] [ with null as 'string' ]`——执行前端（客户端）复制。这是一个运行 SQL COPY 命令的操作，不同的是 SQL COPY 是后端读写指定的文件，对应地需要访问后端和特殊的用户权限，以及受到后端对文件系统的访问权限的限制；而在本命令中，`psql` 读写文件并通过一个本地的文件系统路由从后端中取出或写入数据。

这条命令的语法类似于 SQL COPY 命令，请参考该命令的描述以获取有关细节。需要注意的是，有一些特殊的分析规则应用于 `\copy` 命令，尤其是变量替换规则和反斜杠替换规则不起作用。

本命令不像 SQL COPY 命令那样高效，因为所有数据必须通过客户/服务器或套接字连接。对于大数据量的操作，最好不要使用本命令。

在前端和后端复制时，对 `stdin` 和 `stdout` 的解释是有区别的：在前端复制时，二者分别是指 `psql` 的输入和输出流；在后端复制时，`stdin` 是指来自 COPY 本身的标准输入（如一个带有 `-f` 选项的脚本），而 `stdout` 则指查询输出流（参阅下面的 `\o` 专有命令）。

`\copyright`——显示 PostgreSQL 的版权和版本信息。

`\d relation`——显示关系（可以是表、视图、索引或序列）的所有列。如果存在的话，它们的类型和存在的特殊属性（如 NOT NULL、默认值等）也会显示出来。如果该关系为一个表，任何已定义的索引也会显示出来。如果该关系是一个视图，视图的定义也会显示出来。

`\da [ pattern ]`——列出所有可用聚集函数以及它们操作的数据类型。如果指定了 `pattern`（一个规则表达式），那么只显示与规则表达式匹配的聚集函数。

`\dd [ object ]`——显示对 `object`（可以是一个规则表达式）的描述，如果没有给出参数，则显示所有对象。这里所谓的对象包括聚集、函数、运算符、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)

可以用 COMMENT ON SQL 命令生成对对象的描述。

PostgreSQL 在 pg\_description 系统表中存储对象的描述。

`\df [ pattern ]`——列出所有可用函数，以及它们的参数和返回的数据类型。如果指定了 pattern（一个规则表达式），那么只显示与规则表达式匹配的函数。如果使用了 `\df+` 的形式，每个函数的附加信息，包括语言和描述也会显示出来。

`\distvS [ pattern ]`——这不是一个真正的命令名称，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）。可以以任意顺序指定任意或者所有这些字母获得这些对象的一个列表，以及它们的所有者。

如果指定了 pattern，那么命令的输出将只限制与规则表达式匹配的条目。如果在命令名称后面加一个“+”，那么，每个对象的相关描述也会显示出来。

`\dl`——这是 `\lo_list` 的别名，功能是显示一个大对象的列表。

`\do [ name ]`——列出所有可用运算符，以及它们的操作数和返回的数据类型。如果指定了 pattern，那么只显示与之匹配的运算符。

`\dp [ pattern ]`——这是 `\z` 的别名。

`\dT [ pattern ]`——列出所有的（或匹配 pattern 的）数据类型。这条命令的 `\dT+` 形式可显示更多信息。

`\edit (or \e) [ filename ]`——如果指定了 filename，则编辑此文件并且在编辑器退出后将其内容复制回查询缓冲区。如果没有给出参数，则将当前查询缓冲区内容复制到一个临时文件然后以相同方式编辑。

编辑完毕后，根据一般的 psql 规则重新分析查询缓冲区，这时将整个缓冲区当作一个单行。这还意味着如果该查询以分号结尾（或者包含分号），它就会马上被执行。否则它只是在查询缓冲区里等待。

psql 按顺序搜索环境变量 PSQL\_EDITOR、EDITOR 和 VISUAL，查找应该使用的编辑器。如果上面的变量都没有设置，则以 `/bin/vi` 为编辑器。

`\echo text [ ... ]`——向标准输出打印参数，用一个空格分隔并且最后跟着一个新行。这个特性在显示脚本的输出时有用。例如：

```
\echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

如果第一个参数是一个无引号的 -n，那么不会输出结尾的新行。

如果使用 `\o` 命令重定向查询的输出，可能会用 `\qecho` 取代这条命令。

`\encoding [ encoding ]`——如果使用多字节编码，则可以使用本命令设置客户端编码方式。不带参数时，这条命令显示当前的编码方式。

`\f [ string ]`——为不对齐的查询输出方式设置域分隔符。默认时是“|”（一个管道符号）。参阅 `\pset` 获取设置输出选项的通用方法。

`\g [ { filename | command } ]`——将当前查询输入缓冲区的内容发送到后端，并将结果

输出到可选的 filename 或者将输出定向到一个独立的 UNIX shell 命令。单独一个 \g 实际上等同于一个分号。一个带有参数的 \g 是“一次性”的 \o 命令代用品。

\help (or \h) [ command ]——给出指定 SQL 命令的语法帮助。如果没有给出 command，那么 psql 将列出可获得语法帮助的所有命令。如果 command 是一个星号（“\*”），则显示所有 SQL 命令的语法帮助。

为简化输入，包含多个单字的命令不需要用括号引起。例如：\help alter table。

\H——打开 HTML 查询输出格式。如果 HTML 格式已经打开，则切换回默认的对齐文本格式。这个命令是为了兼容和方便，参阅 \pset 获取设置其他输出选项的内容。

\i filename——从文件 filename 中读取信息并将其内容当作查询命令。

如果想在屏幕上看到读入的行，必须对所有行设置变量 ECHO。

\l (or \list)——列出服务器上所有数据库和它们的所有者。在命令名称后面加一个“+”，还可以看到对数据库的任何描述。如果 Postgres 节点是带多字节支持编译的，还可以看到每个数据库的编码方式。

\lo\_export loid filename——从数据库读取 OID 为 loid 的大对象并将它写到 filename 文件中。这个功能与服务器函数 lo\_export 有些微小的区别，lo\_export 运行时带着运行数据库服务器的用户权限，而且是在服务器的文件系统上。

使用 \lo\_list 可以查看大对象的 OID。

\lo\_import filename [ comment ]——将文件存储为一个 PostgreSQL “大对象”。可以带着一个该对象的注解选项。例如：

```
\lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

执行结果表明，此大对象得到一个对象标识 152801，如果还想访问该对象，就应该记住这个对象标识。因此，建议给每个对象关联一个可读的注解。这样就可以用 \lo\_list 命令看到这些注解。

这条命令与服务器端的 lo\_import 有一些区别，区别在于该命令是以本地用户在本地文件系统上操作，而不是以服务器用户在服务器文件系统上操作。

\lo\_list——显示目前存储在该数据库中的所有 PostgreSQL “大对象”和它们的所有者的列表。

\lo\_unlink loid——从数据库中删除 OID 为 loid 的大对象。

\o [ {filename | command} ]——将其后查询命令的查询结果保存在文件 filename 中或者将后面的查询结果定向到一个独立的 UNIX shell 命令。如果没有指定参数，查询输出重置为 stdout。

“查询结果”包括所有表、命令响应和从数据库服务器来的提示，同时还有各种各样查询数据库的反斜杠命令的输出（如 \d），但是没有错误信息。

如果要分散查询结果之间的输出，用 \qecho。

\p——将当前查询缓冲区的内容送到标准输出。

\pset parameter [ value ]——设置影响查询结果表输出的选项。parameter 描述要设置的选项是哪一个。value 的语意也取决于它。

可用的打印选项有：



`format`：设置输出格式，可以为 Unaligned、Aligned、HTML 或 LaTeX。允许使用唯一性的缩写。

Unaligned（不对齐）格式将一条记录的所有字段都输出到一行，用当前有效的分隔符分隔。这主要用于生成由其他程序读取的输出。

Aligned（对齐）格式是标准的、可读的、格式化的文本输出，也是默认的输出格式。

HTML 和 LaTeX 格式将表输出为可用于文档的对应标记语言。它们还不是完整的文档。可能对于 HTML 变化还不是太大，但是在 LaTeX 中，必须有一个完整的文档包装器。

`border`：第二个参数必须是一个数字。通常，数字越大，表的边界越宽，这个参数取决于实际的格式。在 HTML 格式中，这个参数会直接翻译成 `border=...` 属性；在其他的模式中，只有值 0（无边界）、1（内部分隔线）和 2（表框架）有意义。

`expanded` (or `x`)：在正常和扩展格式之间切换。当打开扩展格式时，所有的输出都是两列，域名称在左，数据在右。这个模式在数据无法放进通常“水平”的屏幕时很有用。

所有四种输出模式都支持扩展模式。

`null`：定义 null 的输出方式。第二个参数是一个字符串，用以代表域的值为 null 时的打印方式。默认情况下，null 域不显示任何信息，这样很容易与空字符串混淆。因此，可能选择 `\pset null "(null)"`。

`fieldsep`：定义非对齐格式的域分隔符。这样就可以创建其他程序希望的分隔输出。要设置 `tab` 为域分隔符，可以使用 `\pset fieldsep "\t"`。默认域分隔符是“|”（一个管道符号）。

`recordsep`：定义非对齐格式的记录分隔符。默认是新行字符。

`tuples_only` (or `t`)：在完全显示和只显示记录之间切换。完全显示将显示列头、标题和各种脚注信息；只显示记录模式则只显示实际的表数据。

`title [ text ]`：为任何随后打印的表设置标题。如果不带参数，重置标题。

在以前的版本中，这个参数只影响 HTML 模式，现在可以在任何输出模式里设置标题。

`tableattr` (or `T`) [ `text` ]：允许指定放在 HTML table 标记中的任何属性，如 `cellpadding` 或者 `bgcolor`。

`pager`：激活分页器进行表输出。如果设置了环境变量 `PAGER`，输出被定向到指定程序，否则使用 `more`。

有很多用于 `\pset` 的快速命令。参阅 `\a`、`\C`、`\H`、`\t`、`\T` 和 `\x`。

`\q`——退出 `psql` 程序。

`\qecho text [ ... ]`——这条命令等同于 `\echo`，区别在于本命令的所有输出写入由 `\o` 设置的输出通道。

`\r`——重置（清空）查询缓冲区。

`\s [ filename ]`——将命令行历史记录显示出来或是存放在由 `filename` 指定的文件中。如果省略 `filename`，历史记录将输出到标准输出。这个选项只有在将 `psql` 配置成使用 GNU 历史库的情况才生效。

对于 `psql v7.0`，这个 GNU 历史库不再是必须的了。实际上，程序在结束时自动保存命令行历史。每次 `psql` 启动都会装载命令行历史。

`\set [ name [ value [ ... ] ] ]`——设置内部变量 `name` 为 `value`。如果给出的值多于一个，

则将变量的值设置为所有这些值的连接结果。如果没有给出第二个参数，只设变量不设值。要重置一个变量，使用 `\unset` 命令。

有效的变量名可以包含字符、数字和下划线。

尽管可以设置任何变量为任意值，`psql` 仍对一些变量特殊对待。

`\t`——切换输出的字段名信息头和行记数标记。这条命令等同于 `\pset tuples_only`，提供这条命令的目的主要是为了方便使用。

`\T table_options`——允许在使用 HTML 输出格式时，定义放在 `table` 标记中的选项。这条命令等同于 `\pset tableattr table_options`。

`\w {filename | command}`——将当前查询缓冲区的内容输出到文件 `filename` 中，或者重定向到 UNIX 命令 `command` 中。

`\x`——切换扩展行格式。等同于 `\pset expanded`。

`\z [ pattern ]`——生成一个带有访问权限列表的数据库中所有表的列表。如果给出任何参数，则被当成一个规则表达式，输出列表将受规则表达式的限制。

```
test=> \z
Access permissions for database "test"
Relation |          Access permissions
-----+-----
my_table | {"=r","joe=arwR", "group staff=ar"}
(1 row )
```

其中权限缩写的意思如下：

`=r`：PUBLIC 拥有对此表的读（SELECT）权限。

`joe=arwR`：用户 `joe` 拥有读、写（UPDATE，DELETE）、追加（INSERT）权限，以及在表上创建规则的权限。

`group staff=ar`：组 `staff` 拥有 SELECT 和 INSERT 权限。

命令 GRANT 和 REVOKE 用于设置访问权限。

`!\ [ command ]`——返回到一个独立的 UNIX shell 或者执行 UNIX 命令 `command`。参数不会被进一步解释，shell 将看到全部参数。

`\?`——获得关于反斜杠（“\”）命令的帮助信息。

(4) 命令行选项：

如果配置正确，`psql` 能够接收和理解标准的 UNIX 短选项和 GNU 风格的长选项。

`-a, --echo-all`——在读取行时，在屏幕上显示所有内容。这个选项在脚本处理时比交互模式时更有用。这个选项等同于设置变量 `ECHO` 为 `all`。

`-A, --no-align`——切换为非对齐输出模式，默认输出模式是对齐的。

`-c, --command query`——执行一条查询命令 `query`，然后退出。这一选项在 shell 脚本中很有用。

`query` 必须是一条完全可以被后端分析的查询字符串，也就是说，它不能包含 `psql` 特有的特性或者反斜杠命令，不能混合使用 SQL 和 `psql` 专有命令。如果要混合使用这两种类型的命令，可以将字符串定向到 `psql`，例如 `echo "\x \select * from foo;" | psql`。

`-d, --dbname dbname`——指定连接的数据库名称。等同于在命令行上将 `dbname` 指定为

第一个无选项参数。

-e, --echo-queries——显示查询结果，等同于将变量 ECHO 设置为 queries。

-E, --echo-hidden——显示由 \d 和其他反斜杠命令生成的实际查询。如果希望在程序中包含类似的功能，可以使用这个选项。这个选项等同于在 psql 中设置变量 ECHO\_HIDDEN。

-f, --file filename——以 filename 所指定文件作为查询命令的输入源。在处理完文件中的命令后，psql 自动结束。这个选项在很多方面等同于内部命令 \i。

使用这个选项与 psql < filename 有微小的区别。通常，两者都会按照预期形式运行，但是使用 -f 会有一些优点，如显示带行号的错误信息。而且，使用这个选项还会有减小启动负荷的机会。另一方面，如果将所有内容手工输入，使用 shell 输入定向的方式（理论上）能保证生成和已经得到的完全一样的输出。

-F, --field-separator separator——使用 separator 作为域分隔符。等同于 \pset fieldsep 或 \f。

-h, --host hostname——指定正在运行 postmaster 的主机名。如果没有此选项，将使用本地的 UNIX 域套接字进行通信。

-H, --html——打开 HTML 格式输出。等同于 \pset format html 或 \H 命令。

-l, --list——列出所有可用的数据库后退出，其他非连接选项将被忽略。类似于内部命令 \list。

-o, --output filename——将所有查询输出定向到文件 filename。这个选项等同于命令 \o。

-p, --port port——指定被 postmaster 用于侦听的 TCP/IP 端口或使用的默认本地 UNIX 主控套接字文件句柄。默认值是环境变量 PGPORT 的值，如果没有设置的话则为编译时指定的端口，通常是 5432。

-P, --pset assignment——允许在命令行上以 \pset 的风格设置打印选项。需要注意的是，在这里用等号分隔名称和值，而不能使用空格。因此要将输出格式设置为 LaTeX，应该输入 -P format=latex。

-q——使 psql “安静地”执行任务。默认情况下，psql 将显示欢迎信息和许多其他输出信息。如果使用了此选项，这些都不出现。这在与 -c 选项一起使用时很有效。在 psql 中，可以通过设置 QUIET 变量来实现同样效果。

-R, --record-separator separator——使用 separator 做为记录分隔符。等同于 \pset recordsep 命令。

-s, --single-step——进入单步模式运行。意味着每个查询在发往后端之前都要提示用户，用这个选项也可以取消执行。此选项主要用于调试脚本。

-S, --single-line——进入单行运行模式，这时每个查询都将由换行符结束。

-t, --tuples-only——关闭显示列名称和结果行计数标注等信息。完全等同于 \t。

-T, --table-attr table\_options——允许指定放在 HTML table 标记中的选项。参阅 \pset 获取细节。

-u——让 psql 在与数据库连接之前提示输入用户的用户名和口令。

不赞成使用这个选项，因为它在概念上有漏洞。提示输入非默认用户名和提示输入后端要求的口令完全是两回事。建议使用 -U 和 -W 选项取代。

-U, --username username——以用户 username 代替默认用户与数据库连接。

-v, --variable, --set assignment——进行一次变量分配，类似于内部命令 \set。注意，如果有变量名和值的话，必须在命令行上用等号分隔它们。要重置一个变量，去掉等号。这个分配是在启动的很早阶段进行的，所以为内部使用保留的变量可能被再次覆盖。

-V, --version——显示 psql 版本。

-W, --password——要求 psql 在与一个数据库连接前提示用户输入口令。这个选项将在整个会话过程中有效，即使用专有命令 \connect 改变了所连接的数据库。

对于版本 7.0，如果后端要求口令认证，psql 自动提出一个口令提示符。因为目前这个特性是以一个“hack”为基础，自动识别有可能神秘地失效，因此用这个选项强制一个提示符。如果没有指定口令提示符而后端要求口令认证，那么连接企图将失败。

-x, --expanded——打开扩展行格式模式。等同于 \x。

-X, --no-psqlrc——不读取启动文件 ~/.psqlrc。

?, --help——显示关于 psql 命令行参数的帮助。

(5) 高级特性：

变量。

psql 提供类似 UNIX 命令 shell 那样的变量替换特性。这个特性是新引入的，目前还不是很复杂，但是今后将扩展它。变量只是简单的名称、值的对，这里的值可以是任何长度的任何值。要设置一个变量，使用 psql 专有命令 \set：

```
testdb=> \set foo bar
```

将变量“foo”设置为值“bar”。要检索变量的内容，在变量名前面插入冒号然后将它用在任意斜杠命令中：

```
testdb=> \echo :foo
```

```
bar
```

注意：\set 的参数服从和其他命令一样的替换规则。因此可以构造有趣的引用，如 \set :foo 'something'，可以获得像 Perl 或 PHP 那样的“软链接”(“soft links”)或“变量变量”(“variable variables”)。不幸的是，用这些构造不能做任何有用的事情。另一方面，\set bar :foo 还是一个非常有效的复制变量的方法。

如果不带第二个参数调用 \set，那么只是设置这个变量而没有值。要重置（或删除）一个变量，使用命令 \unset。

psql 的内部变量可以包括任意顺序的、任意数量的字母、数字和下划线。有一些常用变量被 psql 另眼相待。它们是一些选项设置，这些选项在运行时可以通过改变变量的值或者改变一些应用的表现状态而改变。尽管可以将这些变量用于其他用途，但是不鼓励这么做，因为程序的特性可能会很快变得非常奇怪。通常，所有特殊对待的变量都是由大写字母组成（可能还有数字和下划线）。为了保证和未来版本最大限度的兼容性，应该避免使用这样的变量。下面是所有特殊对待的变量列表。

DBNAME——正在连接的数据库名称。每次与一个数据库连接都会设置这个值（包括程序启动），但是可以删除。

ECHO——如果设置为“all”，输入或者来自一个脚本的所有行在分析或执行前都写到标准输出。要在程序启动时指定这一特性，使用-a 开关。如果设置为“queries”，psql 只是

在查询发送给后端之前显示信息。实现这个功能的命令行选项是 `-e`。

**ECHO\_HIDDEN**——当设置了这个变量并且用一个反斜杠命令查询数据库时，首先显示查询。这样可以借鉴 PostgreSQL 内部的东西，并且在程序中提供类似功能。如果设置该变量的值为 “noexec”，查询只是显示出来但是实际上不发送到后端和执行。

**ENCODING**——当前的客户端多字节编码方式。如果没有设置使用多字节字符，这个变量将总是 “SQL\_ASCII”。

**HISTCONTROL**——如果这个变量设置为 `ignoreSpace`，以空格开始的行将不会进入历史列表。如果设置为变量 `ignoredups`，与以前历史记录匹配的行也不会进入历史记录。变量值 `ignoreboth` 是前面两种情况的结合。如果删除此变量或者其值为任何与上面不同的信息，所有交互模式读入的行都会保存到历史列表。

**HISTSIZE**——存放在命令历史记录缓冲区的命令数目。默认值是 500。

**HOST**——当前正连接的数据库服务器主机。这是在每次与数据库连接时（包括程序启动）设置的，但是可以删除。

**IGNOREEOF**——如果删除此变量，向一个交互的 `psql` 会话发送一个 EOF 字符（通常是 Control-D）将终止应用程序。如果设置为一个数字值，那么在应用程序终止前该数值的 EOF 字符将被忽略。如果设置了此变量但是没有数字值，默认是 10。

**LASTOID**——最后影响的 oid 值，即为从一条 `INSERT` 或 `lo_insert` 命令返回的值。此变量只保证在下一条 SQL 命令的结果显示之前有效。

**LO\_TRANSACTION**——如果使用 PostgreSQL 大对象接口存储无法放进一条记录的特殊数据，所有操作必须包含在一个事务块中（参阅关于大对象接口的文档获取更多信息）。因为 `psql` 在调用它的内部命令 `\lo_export`、`\lo_import`、`\lo_unlink` 之一时，无法跟踪是否有一个正在处理的事务，所以它必须采取一些任意的动作。这个动作可能是回滚任何正在处理的事务，或者提交这样的事务，或者什么也不做。对于后面一种情况，必须提交 `BEGIN TRANSACTION/COMMIT` 块，否则结果将是不可预料的。

要选择希望做的事，可以将此变量设置为 “rollback”、“commit”、“nothing” 之一，默认值是回滚事务。如果只是希望装载一个或者少数几个对象，这个方法很好。但是如果希望传输许多大对象，建议给所有命令提供一个显式的事务块。

**ON\_ERROR\_STOP**——默认时，如果非交互的脚本碰到一个错误，如一条错误的 SQL 查询或者内部专有命令，处理会继续进行。这是 `psql` 的传统特性，但是有时候不太希望这样。如果设置了这个变量，脚本处理将马上停止。如果该脚本是由另外一个脚本调用的，那个脚本也会按同样的方式停止。如果最外层的脚本不是从一次交互的 `psql` 会话中调用的而是用 `-f` 选项调用的，`psql` 将返回错误代码 3，表示这个情况与致命错误条件的区别（错误代码 1）。

**PORT**——当前正在连接的数据库服务器的端口。这是在每次与数据库连接时（包括程序启动）设置的，但是可以删除。

**PROMPT1, PROMPT2, PROMPT3**——`psql` 提示符。

**QUIET**——这个变量等同于命令行选项 `-q`。可能在交互模式下没有什么用。

**SINGLELINE**——这个变量等同于设置命令行选项 `-S`。可以在运行时删除或设置它。

**SINGLESTEP**——这个变量等同于命令行选项 `-s`。

USER——当前正用于连接的数据库用户。这是在每次与数据库连接时（包括程序启动）设置的，但是可以删除/重置。

SQL Interpolation。

一个附加 psql 变量的有用特性是可以将它们改写成正规的 SQL 命令。这样做的语法格式同样是在变量名前面加一个冒号（:）。

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

上面的命令将执行对表 my\_table 的查询。变量的值是逐字复制的，所以它甚至可以包含不对称的引号或反斜杠命令。必须保证输入的东西是有意义的。变量替换将不会在由引号引起来的 SQL 命令内生效。

利用这个功能的一个有趣的应用是通过使用一个随后的命令中最后插入的 OID 建立一个外键。另一个可能用到这个机制的地方是将一个文件的内容复制到一个域。首先将文件装载到一个变量，然后再按上面那样进行处理。

```
testdb=> \set content '\'' `cat my_file.txt` '\''
testdb=> INSERT INTO my_table VALUES (:content);
```

这样处理的一个问题是 my\_file.txt 可能包含单引号。这些需要被转义以免在处理时导致语法错误。可以使用程序 sed 来进行这种处理：

```
testdb=> \set content `sed -e "s/'/\\\\\\\\\\\\\\\\'/g" < my_file.txt`
```

观察正确数量的反斜杠（6 个）！可以这样解释它：在 psql 分析完这行后，它将 sed -e "s/'/\\\\\\\\\\\\\\\\'/g" < my\_file.txt 传递给 shell。shell 将对双引号中的信息进行处理，然后用参数 -e 和 s/'/\\\\\\\\\\\\\\\\'/g 执行 sed。当 sed 进行分析时，它将双反斜杠替换为单个反斜杠，然后进行其他替换。虽然很多时候认为所有 UNIX 命令使用同一个转义字符是件好事，但具有讽刺意味的是，可能不得不转义所有反斜杠，因为 SQL 文本常量同样也惨遭这种解释。这种情况下最好在外部准备文件。

因为冒号也可以合法地出现在查询中，所以有下面规则的应用：如果没有设置变量，字符序列“冒号+名称”不会被改变。在任何情况下都可以用反斜杠转义冒号以保护它免于被解释。变量的冒号语法是 SQL 用于嵌入查询语言的标准，如 ecpg。用于数组片段和类型转换的冒号语法是 PostgreSQL 的扩展，因此有冲突。

提示符。

psql 使用的提示符可以根据用户的喜好来个性化。三个变量 PROMPT1、PROMPT2 和 PROMPT3 包含描述提示符外观的字串和特殊转义序列。PROMPT1 是 psql 请求一个新查询时使用的正常提示符；PROMPT2 是在一个查询输入期待更多输入时（因为查询没有用一个分号结束或者引号没有关闭）显示的提示符；PROMPT3 是在运行一个 SQL COPY 命令和等待在终端上键入记录时使用的提示符。

除非碰到一个百分号（“%”），否则相应提示符变量的值是按原样显示输出的。这时，某些其他的文本被替换，替换为何物取决于下一个字符。已定义的替换包括：

- %M——数据库服务器的主机名。
- %m——数据库服务器的主机名删去第一个点后面的部分剩下的东西，即短域名。
- %>——数据库服务器正在侦听的端口号。

%n——连接使用的用户名。

%/——当前数据库名称。

%~——类似 %/，但如果数据库是默认数据库，则输出“~”。

%#——如果当前用户是数据库超级用户，使用“#”，否则使用“>”。

%R——对于 PROMPT1 通常是“=”，但是如果是单行模式则是“^”，而如果会话与数据库断开（如果 \connect 失败可能发生）则是“!”。对于 PROMPT2 该序列被“-”、“\*”、一个单引号或者一个双引号代替，这取决于 psql 是否等待更多的输入（因为查询没有终止，或者正在一个/\* ... \*/注释中，或者因为在引号里面）。对于 PROMPT3，该序列不作任何解释。

%digits——如果 digits 以 0x 开头，那么其余字符将被解释成一个十六进制数字并且替换为对应的字符。如果第一个数字是 0，该字符将被解释成一个八进制数字并且替换为对应的字符。否则认为是一个十进制数字。

%:name:——psql 变量 name 的值。

%`command`——command 的输出，类似于通常的反单引号（“back-tick”）替换。

如果想在提示符中插入百分号，可以使用%%。默认提示符等同于%/R%#，用于 PROMPT1 和 PROMPT2，>>用于 PROMPT3。

杂项。

psql 正常结束时向 shell 返回 0；发生自身致命错误（如用光内存、文件没有找到等）时返回 1；与后端连接出错和会话不是交互的时候返回 2；如果在一个脚本里发生错误或者变量 ON\_ERROR\_STOP 被设置了返回 3。

在启动之前，psql 试图读取并执行文件\$HOME/.psqlrc 里的命令。这个特性可以用来个性化设置客户端或者服务器（用 \set 和 SET 命令）。

GNU readline。

psql 为了编辑和检索命令行的方便，支持 readline 和历史库。命令历史存放在根目录中的一个名为.psql\_history 的文件中，并且在 psql 启动时会自动装载进来。Tab 补齐同样也被支持，尽管该补齐逻辑并不是一个 SQL 分析器必备的。如果可能，psql 会自动使用这些特性。如果因某些原因不喜欢 tab 补齐，可以将下面几行放在根目录中的一个名为 .inputrc 的文件中，以便关闭这个特性：

```
$if psql
set disable-completion on
$endif
```

如果安装了行读（readline）库，而 psql 没有使用它，必须确保 PostgreSQL 的顶级 configure 脚本能够找到该库。configure 需要能够在合适的目录里找到库 libreadline.a（或者一个等同的共享库）和头文件 readline.h、history.h（或 readline/readline.h 和 readline/history.h）。

GNU 行读库可以从 GNU 计划的 FTP 服务器 ftp://ftp.gnu.org 得到。

(6) 例子：

--第一个例子演示了如何将一个查询分成多个行进行输入。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 |
```

```
--这里决定将提示符变成更有趣的东西：
```

```
testdb=> \set PROMPT1 '%n@m %~%R%# '
```

```
peter@localhost testdb=>
```

```
--假设用数据填充了表：
```

```
peter@localhost testdb=> SELECT * FROM my_table;
```

```
first | second
```

```
-----+-----
```

```
1 | one
```

```
2 | two
```

```
3 | three
```

```
4 | four
```

```
(4 rows)
```

--注意 int4 列是怎样右对齐的以及 text 列是如何左对齐的。可以用 \pset 命令让这个查询看起来不一样。

```
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
```



## 24 命令与工具

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
```

--还可以用短(缩写)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
```

#### (7) 其他说明。

一些 psql 的早期版本允许第一个参数(单字母)直接跟在命令后面。出于兼容性原因,这个特性现在仍然在某种程度上被支持,但不鼓励这样使用。不过,如果收到莫名其妙的信息,应该想到可能是由上述原因导致的。例如:

```
testdb=> \foo
Field separator is "oo".
```

上述命令的结果可能不是所期望的东西。

psql 只能与同版本的服务器平稳地工作。但这并不意味着其他组合会完全失败,可能会有微小的或者较大的问题。

在一个“写复制”(数据发送给服务器)过程中按下 Control-C 并不显示出最典型的特征。如果收到一条类似于下面这样的信息“PQexec: you gotta get out of a COPY state yourself”,只需要输入 \c -- 重置连接即可。

### 14. pgtclsh (TCL shell 客户端)

语法: pgtclsh [ dbname ]

描述: pgtclsh 为 PostgreSQL 提供一个 TCL shell 接口。

另一个通过 tcl 访问 PostgreSQL 的方法是使用 pgtksh 或 pgaccess。

输入:

dbname——要访问的数据库名称。

输出: 无

### 15. pgtksh (图形化 TCL/TK shell)

语法: pgtksh [ dbname ]

描述: pgtksh 为 PostgreSQL 提供一个图形化 TCL/TK shell 接口。

另一个通过 tcl 访问 PostgreSQL 的方法是使用 pgtclsh 或 pgaccess。

输入:

dbname——要访问的现有数据库名称。

输出: 无

### 16. vacuumdb (清理和分析数据库)

语法: vacuumdb [ options ] [ --analyze | -z ]

[ --alldb | -a ] [ --verbose | -v ]

[ --table 'table [ ( column [,...] ) ]' ] [ [-d] dbname ]

描述: vacuumdb 是一个用于整理 PostgreSQL 数据库的工具。vacuumdb 还会生成用于 PostgreSQL 查询优化器的内部统计数据。

`vacuumdb` 是一个 shell 脚本，通过 PostgreSQL 前端工具 `psql` 封装了 SQL 命令 `VACUUM`。这意味着 `psql` 必须能被脚本找到并且有一个数据库服务器在目标主机上运行。同样，任何 `psql` 和 `libpq` 前端库可获得的默认设置和环境变量都将生效。

输入：

`vacuumdb` 接受下列命令行参数：

- d dbname, --dbname dbname——被清理或分析的数据库名称。
- z, --analyze——计算用于优化器的该数据库的统计值。
- a, --alldb——清理所有数据库。
- v, --verbose——在处理过程中显示详细信息。
- t table [ (column [...]) ], --table table [ (column [...]) ]——只是清理或分析 table。列名称只是在与 `--analyze` 选项联合使用时才需要指定。

`vacuumdb` 还接受下面的命令行参数用于连接参数：

- h host, --host host——运行 postmaster 的主机名。
- p port, --port port——postmaster 正在侦听连接的 TCP/I 端口号或一个本地的 UNIX 主控套接字文件句柄。
- U username, --username username——进行连接的用户名。
- W, --password——强制口令提示符。
- e, --echo——回显 `vacuumdb` 生成的查询并且将它发送到后端。
- q, --quiet——不显示响应。

输出：如果命令执行正常，返回信息“VACUUM”；否则，返回错误信息“`vacuumdb: Vacuum failed.`”。

用法：

```
--整理数据库 test :
$ vacuumdb test
--为优化器分析一个名为 bigdb 的数据库 :
$ vacuumdb --analyze bigdb
--为优化器分析数据库 xyzzy 的表 foo 中的字段 bar :
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

## 17. initdb（创建数据库节点）

语法：initdb [ --pgdata|-D dbdir ]

[ --sysid|-i sysid ]

[ --pwprompt|-W ]

[ --encoding|-E encoding ]

[ --pglib|-L libdir ]

[ --noclean | -n ] [ --debug | -d ] [ --template | -t ]

描述：本命令用于创建一个新的 PostgreSQL 数据库系统。一个数据库系统是由一个 UNIX 用户配置的并由单个 postmaster 管理的数据库集合。

创建数据库系统包括创建数据库数据的主目录，生成共享的系统表（不属于任何特定数据库的表）和创建 `template1` 数据库。当创建一个数据库时，`template1` 数据库中所有内容都会复制过来，并包括内建类型那样的系统表。

不能以 `root` 身份运行 `initdb`。这是因为不能以 `root` 身份运行数据库服务器，但是服务器必须能够访问 `initdb` 创建的文件。在初始化阶段，由于尚未安装用户和访问控制，PostgreSQL 用当前的 UNIX 用户名进行连接，所以必须以拥有服务器进程的帐号登录。

尽管 `initdb` 会尝试创建相应的数据目录，但还是有可能发生它没有权限做这些事情的情况。因此在运行 `initdb` 前，先创建数据目录并且将所有权限赋予数据库的超级用户，这是一个很好的习惯。

输入：

`--pgdata=dbdir, -D dbdir, PGDATA`——这个选项指定数据库在文件系统中的存放位置。这是 `initdb` 必需的唯一信息，也可以通过设置 `PGDATA` 环境变量来避免在命令行上输入这一信息，这样做可能方便一些，因为稍后数据库服务器（`postmaster`）可以通过同一个变量找到数据库目录。

`--sysid=sysid, -i sysid`——选择数据库超级用户的系统标识（`system id`）。默认值是运行 `initdb` 的有效用户标识（`user id`）。超级用户的系统标识是什么并不重要，可以选择从 0 或 1 这样的数字开始。

`--pwprompt, -W`——让 `initdb` 提示输入数据库超级用户的口令。如果不准备使用口令认证，则不必使用这个选择项；否则将不能完成口令认证。

`--encoding=encoding, -E encoding`——选择模板数据库的多字节编码方式。这将是以后创建的数据库的默认编码方式，除非在创建数据库时覆盖了它。要使用多字节编码特性，必须在编译时指定，还要为这个选项选择默认值。

其他不常用的参数还有：

`--pglib=libdir, -l libdir`——`initdb` 需要几个输入文件初始化数据库。这个选项告诉它到哪里能找到这些文件。通常不必担心这些，因为 `initdb` 知道最常见的安装布局而且能够自己找到这些文件。如果需要显式地指定这些文件的路径，系统会给出详细的提示。

`--template, -t`——在不改变其他任何内容的前提下，替换数据库系统中的 `template1` 数据库。当需要使用 `initdb` 从一个新的 PostgreSQL 版本中升级 `template1` 数据库时，或者 `template1` 数据库因为一些系统问题崩溃了时，这个选择项非常有用。通常，`template1` 的内容在整个数据库系统的生存周期中必须保持一致。带 `--template` 参数运行 `initdb` 不能破坏任何数据。

`--noclean, -n`——默认时，如果 `initdb` 发现一些错误妨碍它完成创建数据库系统的工作，它将在检测到不能结束工作之前将其创建的所有文件删除。这个选项禁止任何清理动作，因而对调试很有用。

`--debug, -d`——打印初始后端上的调试输出和一些其他普通用户不太感兴趣的信息。初始后端是 `initdb` 用于创建目录表的进程。这个选项生成一大堆输出。

输出：

`initdb` 将在指定的数据区内创建完成安装所需的系统表文件和框架文件。

## 18. initlocation（创建数据库存储区）

语法: `initlocation directory`

描述: 本命令用于创建一个新的 PostgreSQL 从属数据库存储区。参阅 `CREATE DATABASE` 中关于如何管理和使用从属存储区的讨论。如果参数不包含一个斜杠而且也不是一个有效的路径, 它会被认为是对一个环境变量的引用。

为使用这条命令, 必须以数据库超级用户的身份登录。

输入:

`directory`——可选数据库的存放位置。

输出:

`initlocation` 将在指定位置创建目录。

用法:

--用环境变量在一个可选位置创建一个数据库:

```
$ export PGDATA2=/opt/PostgreSQL/data
```

```
$ initlocation PGDATA2
```

```
$ createdb 'testdb' -D 'PGDATA2'
```

--或者, 如果允许使用绝对路径, 可以这样:

```
$ initlocation /opt/PostgreSQL/data
```

```
$ createdb testdb -D '/opt/PostgreSQL/data/testdb'
```

## 19. ipcclean（清理遗留的共享内存和信号灯）

语法: `ipcclean`

描述: 本命令通过删除所有属于 PostgreSQL 的 IPC 实例, 清空一个退出后端的共享内存和信号灯空间。只有数据库管理员才能运行这条命令, 因为如果在多用户下执行, 会导致异常的现象。如果在启动 `postmaster` 或者后端服务器时出现 “`semget: No space left on device`” 等信息时, 也应该执行这条命令。

如果在 `postmaster` 运行时执行这条命令, `postmaster` 分配的共享内存和信号灯将被删除。这通常会导致该 `postmaster` 后端服务器错误。

这个脚本对 `ipcs` 输出格式的假设在不同平台之间并不总是正确的, 因此它可能在某些操作系统上无法工作。

输入: 无。

输出: 无。

## 20. pg\_passwd（管理口令文件）

语法: `pg_passwd filename`

描述: `pg_passwd` 是一个操作 PostgreSQL 口令文件的工具。这样的口令认证方式并非每一个安装所必须的, 但它是 PostgreSQL 支持的几种安全机制之一。

在\$PGDATA/pg\_hba.conf 中有与 Ident 认证相同风格的口令文件：

```
host unv 133.65.96.250 255.255.255.255 password passwd
```

上面这一行允许 IP 地址为 133.65.96.250 的用户使用\$PGDATA/passwd 中列出的口令来访问数据库系统。该口令文件的格式遵循/etc/passwd 和/etc/shadow 格式。第一个域是用户名，第二个域是加密的口令，其他域完全没有用。因此，下面三行定义了同样的用户、口令对：

```
pg_guest:/nB7.w5Auq.BY:10031::::::
```

```
pg_guest:/nB7.w5Auq.BY:93001:930::/home/guest:/bin/tcsh
```

```
pg_guest:/nB7.w5Auq.BY:93001
```

在将工作目录切换到 PGDATA 后，执行下面的命令给用户 pg\_guest 指定新口令：

```
% pg_passwd passwd
```

```
Username: pg_guest
```

```
Password:
```

```
Re-enter password:
```

这里的 Password:和 Re-enter password:提示需要相同的口令输入，这些口令不会在终端显示。初始的口令文件被改为 passwd.bk。

psql 使用-u 选项激活这个风格的认证。

用法：

--下面行显示这个选项的用法示例：

```
% psql -h hyalos -u unv
```

```
Username: pg_guest
```

```
Password:
```

```
Welcome to the PostgreSQL interactive sql monitor:
```

```
Please read the file COPYRIGHT for copyright terms of PostgreSQL
```

```
type \? for help on slash commands
```

```
type \q to quit
```

```
type \g or terminate with semicolon to execute query
```

```
You are currently connected to the database: unv
```

```
unv=>
```

--Perl5 认证使用新的 Pg.pm 风格，如：

```
$conn = Pg::connectdb("host=hyalos dbname=unv
                      user=pg_guest password=xxxxxxx");
```

--更多细节，请参考 src/interfaces/perl5/Pg.pm。

--Pg{tcl,tk}sh 认证使用 pg\_connect 带 -conninfo 选项的命令，如：

```
% set conn [pg_connect -conninfo \\\
                      "host=hyalos dbname=unv \\\
```

```
user=pg_guest password=xxxxxxx " ]
```

--可以通过执行下面命令列出该选项的所有关键字：

```
% puts [ pg_conndefaults]
```

## 21. pg\_upgrade (升级版本)

语法: `pg_upgrade [ -f filename ] old_data_dir`

描述: `pg_upgrade` 是一个从老版本的 PostgreSQL 升级而不必重载数据的工具。不是所有 PostgreSQL 版本的迁移都可以用这种方法实现。请检查版本信息获取细节。用 `pg_upgrade` 升级 PostgreSQL 的基本操作步骤是：

(1) 备份现有的数据目录，最好用 `pg_dumpall` 做一次完整输出。

(2) 不带任何数据输出老数据库的表定义。

```
% pg_dumpall -s >db.out
```

(3) 停止老的 `postmaster` 和所有后端进程。

(4) 将老的 `pgsql data/` 目录改成（用 `mv` 命令）`data.old/`。

(5) 运行 `make install` 安装新的二进制文件。

(6) 运行 `initdb` 为新版本创建新的包含系统表的 `template1` 数据库。

(7) 运行新的 `postmaster`。在数据库升级完成前不能有用户与数据库连接。可能需要不带 `-i` 参数启动 `postmaster` 和暂时修改 `pg_hba.conf`。

(8) 将工作目录切换到 `pgsql` 主目录，然后输入：

```
% pg_upgrade -f db.out data.old
```

该程序会进行一些检查以确保所有内容都被正确配置好了，然后运行 `db.out` 脚本创建原来数据库和表，但这时还没有装入数据。最后它将那些不含系统表和索引的文件物理地从 `data.old` 移到合适的 `data/` 子目录，代替在运行 `db.out` 脚本时创建的空文件。

在必要时需要恢复老的 `pg_hba.conf`，以允许用户登录。

(9) 停止并重新启动 `postmaster`。

(10) 仔细检查升级的表内容。如果发现问题，需要利用 `pg_dump` 创建的备份来恢复数据库的表。如果升级成功，可以删除 `data.old/` 目录。

升级完的数据库处于未清理的状态。在开始工作前可能需要运行 `VACUUM ANALYZE`。

## 22. PostgreSQL (启动单用户后端)

语法: `PostgreSQL [ dbname ]`

```
PostgreSQL [ -B nBuffers ] [ -C ] [ -D DataDir ] [ -E ] [ -F ]
```

```
[ -O ] [ -Q ] [ -S SortSize ] [ -d [ DebugLevel ] ] [ -e ]
```

```
[ -o ] [ OutputFile ] [ -s ] [ -v protocol ] [ dbname ]
```

描述: `PostgreSQL` 可以直接从用户 `shell` 执行，但只有在数据库管理员对系统进行调

试时才可以这样做。但是,在数据库的其他 PostgreSQL 后端正在被一个 postmaster 管理时,决不能使用这个命令。

输入:

dbname——可选的参数 dbname 为要访问的数据库名。dbname 的默认值是环境变量 USER 的值。

-B nBuffers——如果后端通过 postmaster 运行,nBuffers 是 postmaster 为它启动的后端进程分配和管理的共享内存缓冲区数量。如果后端是独立运行的,这个参数为要分配的缓冲区数目。默认值为 64 个缓冲区,每个缓冲区为 8k 字节(或者是在 config.h 中 BLCKSZ 指定的大小)。

-C——不显示服务器版本号。

-D DataDir——数据库目录树的根目录。如果没有给出-D,默认目录名是环境变量 PGDATA 的值。如果没有设置 PGDATA,那么使用的目录是\$POSTGRESHOME/data。如果两个环境变量和命令行选项都没有给出,默认目录为编译时设置的值。

-E——回显所有查询。

-F——在每次事务执行完毕后关闭对 fsync()的自动调用。这个选项是为了提高性能,但是,如果一个事务正在处理过程中操作系统突然崩溃,这可能导致最近输入的数据的丢失。没有 fsync()调用,数据将被操作系统缓冲,然后在某个时候写入磁盘。

-O——超越限制,因而可以修改系统表。这些表一般是那些表名称带有 "pg\_" 前缀的表。

-Q——指定执行模式为 quiet (安静) 模式。

-S SortSize——指定内部排序和散列在使用临时磁盘文件之前可以使用的内存数量。该值是以 KB (千字节) 为单位的,默认值为 512 KB。对于复杂查询,可能有好几个并行的排序和/或散列,而在它们将数据放到临时文件前,每个任务都会允许使用最多 SortSize KB 的内存。

-d [ DebugLevel ]——可选的参数 DebugLevel 决定后端服务器所产生的调试输出信息的数量。如果 DebugLevel 为 1,系统将跟踪所有连接动作,而忽略其他动作;对于级别 2 或更高级别,后端进程的调试被打开并且 postmaster 将显示更多信息,包括后端环境和处理动作。如果没有给后端服务器指定输出文件,则输出会出现在父 postmaster 的控制台上。

-e——这个选项控制输入到数据库和从数据库输出的日期解释方式。如果使用了-e 开关,那么日期将假设为“European”格式(DD-MM-YYYY),否则日期将假设为“American”格式(MM-DD-YYYY)。后端可以接受的日期格式是多种多样的,对于输入日期而言,这个选项主要影响易混淆的解释。

-o OutputFile——将所有调试和错误信息输出到 OutputFile。如果后端是由 postmaster 运行的,错误信息仍然发送给前端,也发送给 OutputFile,但是调试信息将只发送给该 postmaster 的控制台。

-s——在每条查询结束时显示时间信息和其他统计信息。这个开关对测试性能和调节缓冲区数量有好处。

-v protocol——指定前、后端协议的版本。

除上述选项之外,还有几个用于调试的选项。这些选项一般只给 PostgreSQL 系统开发



人员使用，一般用户不要使用这些选项。另外，这些选项中的任何一项都可能在任何时间消失。

-A n|r|b|Q|fIn|fP|X|fIn|fP——这个选项产生数量巨大的、用于调试的信息。

-L——关闭锁定系统。

-N——关闭将新行作为查询分隔符。

-f [ s | i | m | n | h ]——禁止某种扫描和联合方法的使用，s 和 i 分别关闭顺序和索引扫描，而 n、m、和 h 分别关闭嵌套循环、融合和散列联合。

顺序扫描和嵌套循环都不可能完全被关闭。-fs 和 -fn 选项仅仅是在存在其他方法时阻碍优化器使用这些方法而已。

-i——避免查询的执行，而只是显示规划树。

-p dbname——告诉后端服务器，它是由一个 postmaster 启动的，并对缓冲区管理和文件描述符做出不同的假设。

-t pa[rser] | pl[anner] | e[xecutor]——显示与每个主要系统模块相关的查询时间统计信息，不能与 -s 选项一起使用。

输出：直接运行后端服务器，可能会出现许多错误信息，最有可能出现的信息是“semget: No space left on device”。如果看到这条信息，应该运行 ipcclean 命令。然后，试着重新启动 postmaster。如果还有问题，可能需要按照安装指导里描述的那样配置内核共享内存和信号灯。如果内核共享内存和/或信号灯被限制得很小，可能要重新配置内核以增加共享内存和信号灯参数。

## 23. postmaster（启动多用户后端）

语法：postmaster [ -B nBuffers ] [ -D DataDir ] [ -N maxBackends ] [ -S ]  
[ -d DebugLevel ] [ -i ] [ -l ]  
[ -o BackendOptions ] [ -p port ] [ -n | -s ]

描述：postmaster 管理前端和后端进程之间的通信，负责分配共享缓冲池和 SysV 信号灯（在没有测试和设置指令的机器上）。postmaster 本身并不与用户交互并且应该作为一个后端进程启动。

每个 PostgreSQL 安装每次应该只启动一个 postmaster。一个安装的意思是指一个数据库目录和 postmaster 端口号。可以在一台机器上运行多个 postmaster，前提是每个 postmaster 有不同的目录和端口号。

不要使用 SIGKILL 终止 postmaster 进程，而应该使用 SIGHUP、SIGINT 或 SIGTERM。用 kill -KILL 或者其可选形式 kill -9 会阻止 postmaster 在退出前释放它占有的系统资源（如共享内存和信号灯）。

输入：

-B nBuffers——postmaster 为它启动的后端进程分配和管理的共享内存缓冲区数量。默认为 64 个缓冲区，每个缓冲区为 8k 字节（或为 config.h 中 BLCKSZ 定义的大小）。

-D DataDir——数据库目录树的根目录。如果没有给出 -D，默认目录名是环境变量 PGDATA 的值。如果没有设置 PGDATA，那么使用的目录为 \$POSTGRESHOME/data。如果两个环境变量和命令行选项都没有给出，默认目录为编译时设置的目录。

-N maxBackends——允许启动的后端服务器最大数目。默认配置时，该值为 32，如果系统能支持更多进程，该值最大可以设置为 1024。默认值和最大值都可以在编译 PostgreSQL 时修改（参阅 `src/include/config.h`）。

-S——以安静模式启动 postmaster 进程。也就是说，它将与用户的控制台脱离并且启动其自身的进程组。这个选项不应和调试选项结合使用，因为向标准输出设备和标准错误设备输出的任何信息都被丢弃。

请注意，使用这个开关会令除错工作变得非常困难，因为通常由这个 postmaster 和它生成的后代产生的跟踪和日志信息都将被丢弃。

-d DebugLevel——后端服务器的调试级别，它决定后端服务器信息输出的数量。如果 DebugLevel 为 1（级别 1），系统将跟踪所有连接动作，而忽略其他所有动作。对于级别 2 或更高，后端进程的调试被打开并且 postmaster 将显示更多信息，包括后端环境和处理动作。如果没有给后端服务器指定信息输出文件，则输出会出现在父 postmaster 的控制台上。

-i——这个选项打开 TCP/IP 或者域套接字通信。没有这个选项 则只能进行本地 UNIX 域套接字通信。

-l——这个选项打开 SSL 套接字通信。同样还需要 -i 选项。如果要使用这个选项，编译时必须打开了 SSL 选项。

-o BackendOptions——在 BackendOptions 中指定的 PostgreSQL 选项都传递给所有由这个 postmaster 启动的后端服务进程。如果选项字符串包含任何空白，整个字符串必须引起来。

-p port——指定 postmaster 侦听并等待连接的互联网 TCP/IP 端口或本地 UNIX 域套接字文件扩展（描述符）。默认的端口号是环境变量 PGPORT 的值。如果没有设置 PGPORT，默认是 PostgreSQL 编译时建立的值（通常是 5432）。如果指定了一个非默认端口，那么所有前端应用（包括 psql）都必须用命令行选项或者 PGPORT 指定连接端口。

还有几个命令行选项用于调试。这些选项用于控制 postmaster。这些特殊选项包括：

-n——postmaster 将不会重新初始化共享数据结构。一个有经验的系统程序员这时就可以使用 shmemdoc 程序检查共享内存和信号灯状态。

-s——postmaster 将通过发送信号 SIGSTOP 停止所有其他后端进程，但不会导致它们退出。这样就允许系统程序员手工地从所有后端进程收集转储的核心（core dumps）。

输出：下面给出的信息是 postmaster 启动时的出错信息。

(1) semget: No space left on device。如果看到这条信息，应该运行 ipcclean 命令，然后试着重新启动 postmaster。如果还有问题，可能需要按照安装指导描述的那样配置内核共享内存和信号灯。如果在一台主机上运行多个 postmaster 实例，或者内核共享内存和/或信号灯被限制得很小，可能要重新配置内核以增加共享内存和信号灯参数。

也可以通过降低 -B 的参数以减少 PostgreSQL 对共享内存的消耗，或者降低 -N 参数以减少 PostgreSQL 对信号灯的消耗。

(2) StreamServerPort: cannot bind to port。如果看到这样的信息，应该确保没有其他 postmaster 进程正在运行。判断这个情况的最简单的办法是使用命令 `ps -ax | grep postmaster` 或 `ps -e | grep postmast`。

如果确信没有其他 postmaster 进程在运行但是还是收到这个错误信息，试着用 -p 选项指定一个不同的端口。如果终止 postmaster 后又马上用同一个端口运行它，也有可能得到

这个错误信息。这时，必须多等几秒钟，等操作系统关闭了该端口后再试。如果使用了一个操作系统保留的端口，也可能导致这个错误信息。例如，许多 UNIX 版本认为低于 1024 的端口号是可信任的，因而只有 UNIX 超级用户可以使用它们。

(3) IpcMemoryAttach: shmat() failed: Permission denied。一个可能的解释是另外一个用户企图在同一端口上运行一个 postmaster 进程，该进程请求共享资源然后退出。因为 PostgreSQL 的共享内存键是以 postmaster 端口号为基础的，因此在同一台主机上有两个以上的 postmaster 运行的话，很有可能发生冲突。如果当前没有其他 postmaster 进程正在运行，运行 ipcclean 然后再尝试一下。如果其他 postmaster 镜像正在运行，就不得不与那些进程的所有者协调端口号的分配，或删除未用的共享内存段。

用法：

--用默认值启动 postmaster

```
% nohup postmaster >logfile 2>&1 &
```

--这条命令将在默认端口（5432）启动 postmaster。这是最简单和最常用的启动 postmaster 的方法。

--在指定的端口启动 postmaster：

```
% nohup postmaster -p 1234 &
```

--这条命令将启动通过端口 1234 通信的 postmaster。为了用 psql 与这个 postmaster 连接，需要这样运行（psql）：

```
% psql -p 1234
```

--或者设置环境变量 PGPORT：

```
% setenv PGPORT 1234
```

```
% psql
```

附录

# 常见问题解答

---

本附录根据PostgreSQL随机文档整理，主要包括：

- 一般问题
- 客户端问题
- 管理员问题
- 操作问题
- 扩展问题





这一部分内容根据“*Frequently Asked Questions for PostgreSQL*”整理，原文的最后修改时间为“Wed Jul 26 13:31:44 EDT 2000”，当前的维护者为“Bruce Momjian” (pgman@candle.pha.pa.us)，文件的最新版浏览 <http://www.PostgreSQL.org/docs/faq-english.html>。

### 1. 一般问题

#### (1) 什么是 PostgreSQL?

PostgreSQL 是 Postgres 系统（新一代 DBMS 研究原型）的增强型系统。PostgreSQL 保留了 Postgres 强有力的数据模型和丰富的数据类型，它用扩展的 SQL 子集取代了 postquel 查询语言。PostgreSQL 是免费软件，所有的源代码都是公开的。

PostgreSQL 的开发由一组 Internet 开发者完成，他们的名字和电子邮件地址都列在 PostgreSQL 的开发者邮件列表中。目前的协调者是 Marc G. Fournier (scrappy@PostgreSQL.org)。该小组目前对 PostgreSQL 的全部开发负责。

PostgreSQL 1.01 版的开发者是 Andrew Yu 和 Jolly Chen。原始的 Postgres 代码，也就是 PostgreSQL 的起源，是加利福尼亚大学伯克利分校许多研究生、在读本科生和在 Michael Stonebraker 教授指导下工作的其他编程人员共同努力的结果。

在伯克利分校，这个软件最初的名字为 Postgres。1995 年加入了 SQL 功能后，它的名字变为 Postgres95。1996 年底，名字改为 PostgreSQL。它发音为 Post-Gres-Q-L。

#### (2) PostgreSQL 的版权。

PostgreSQL 版权见本书的第一章。

#### (3) PostgreSQL 可运行于哪些 UNIX 平台?

作者在下述平台上编译并测试过 PostgreSQL（其中一些编译需要 gcc）：

- aix - IBM on AIX 3.2.5 或 4.x
- alpha - DEC Alpha AXP on Digital UNIX 2.0, 3.2, 4.0
- BSD44\_derived - OSs derived from 4.4-lite BSD (NetBSD, FreeBSD)
- bsdi - BSD/OS 2.x, 3.x, 4.x
- dgux - DG/UX 5.4R4.11
- hpux - HP PA-RISC on HP-UX 9.\*, 10.\*
- i386\_solaris - i386
- Irix5 - SGI MIPS
- MIPS on IRIX 5.3
- linux - Intel i86 Alpha SPARC PPC M68k
- sco - SCO 3.2v5
- UNIXware
- sparc\_solaris - SUN SPARC on Solaris 2.4, 2.5, 2.5.1
- sunos4 - SUN SPARC on SunOS 4.1.3
- svr4 - Intel x86 on Intel SVR4 and MIPS

- ultrix4 - DEC MIPS on Ultrix 4.4

(4) 哪些非 UNIX 平台运行 PostgreSQL?

可以在 MS Windows 平台上编译运行 libpq C library, psql 和其他接口和二进制程序。在这种情况下,客户端软件在 MS Windows 平台上运行,通过 TCP/IP 协议连接到 UNIX 服务器。分发的产品中包含 win31.mak 文件,利用它可以生成 Win32 libpq library 和 psql。

如果想在 Windows NT 上运行数据库服务器,需要使用 Cygnus UNIX/NT 仿真库,详情见 pgsql/doc/FAQ\_NT。

(5) 哪里可以得到 PostgreSQL?

PostgreSQL 最早的匿名 FTP 站点是 <ftp://ftp.PostgreSQL.org/pub>, 在 <http://www.postgresql.org> 站点上能够找到 PostgreSQL 的镜像站点。

(6) 从哪里可以得到 PostgreSQL 的技术支持?

加利福尼亚大学伯克利分校不提供 PostgreSQL 的支持,它的支持由志愿者来维护。主要邮件地址为 [pgsql-general@PostgreSQL.org](mailto:pgsql-general@PostgreSQL.org), 含有各种关于 PostgreSQL 的讨论。如果想订阅,可在邮件正文中(不能在主题中)加入以下行:

```
subscribe
end
```

除此之外,还有分类摘要可供利用,邮件地址为 [pgsql-general-digest-request@PostgreSQL.org](mailto:pgsql-general-digest-request@PostgreSQL.org)。如果想订阅,可在邮件正文中加入:

```
subscribe
end
```

主邮件列表在收到约 30K 信息后,就把摘要发到每一个列表成员。

Bug 邮件列表的地址是 [pgsql-bugs-request@PostgreSQL.org](mailto:pgsql-bugs-request@PostgreSQL.org)。如果想订阅,可在邮件正文中加入:

```
subscribe
end
```

开发者论坛的邮件列表地址是 [pgsql-hackers-request@PostgreSQL.org](mailto:pgsql-hackers-request@PostgreSQL.org)。如果想订阅,可在邮件正文中加入:

```
subscribe
end
```

其他邮件列表和 PostgreSQL 相关信息可通过 PostgreSQL 的主页查找。

EFNet 的 IRC 信道为: “#PostgreSQL”。用 UNIX 命令 `irc -c '#PostgreSQL' "$USER"` `irc.phoenix.net`。PostgreSQL 的商业支持请浏览: <http://www.pgsql.com/>。

(7) 最新版是什么?

最新版是 version 7.0.2。PostgreSQL 计划每 4 个月做一次大的调整。



(8) 提供哪些文件?

文件包括手册、manual pages 和一些小试验程序。见/doc 目录。还可以浏览在线手册 (<http://www.PostgreSQL.org/docs/postgres>)。

PostgreSQL 书籍信息在 <http://www.PostgreSQL.org/docs/awbook.html>。

psql 有一些很好的带反斜线的命令, 给出了关于类型、操作、函数、集合等的信息。

(9) 如何得知已发现的 bugs 和缺少的特性?

PostgreSQL 支持 SQL-92 的扩展子集。TODO 列表中包含已发现的 bugs、缺少的特性和未来的计划。

(10) 如何学习 SQL?

关于 PostgreSQL 的书籍请浏览: <http://www.PostgreSQL.org/docs/awbook.html>、<http://w3.one.net/~jhoffman/sqltut.htm> 和 [http://ourworld.compuserve.com/homepages/graeme\\_birchall/HTM\\_COOK.HTM](http://ourworld.compuserve.com/homepages/graeme_birchall/HTM_COOK.HTM)。

另外还有《21 天自学 SQL》第二版 (<http://members.tripod.com/er4ebus/sql/index.htm>)。

很多用户喜欢《SQL 实践手册》, 作者为 Bowman, Judith S.、et al., Addison Wesley。也有些人喜欢《SQL 完全参考》, 作者为 Groff et al., McGraw-Hill。

(11) PostgreSQL 有千年虫问题吗?

没有。用户可以自如的控制日期超过公元 2000 以及公元前 2000。

(12) 如何加入开发组?

首先, 从 <http://www.postgresql.org> 站点下载或从分发介质中获得最新的源程序, 阅读 PostgreSQL 开发者文档。然后, 订阅 pgsql-hackers 和 pgsql-patches 邮件列表。第三步, 提交高质量的补丁给 pgsql-patches。大约有 12 人曾经提交特权给 PostgreSQL CVS 存档。他们每个人都提交了大量高质量的、目前不可能维持的补丁。

(13) 如何提交 bug 报告?

填写 bug-template 文件并把它发送到 [pgsql-bugs@PostgreSQL.org](mailto:pgsql-bugs@PostgreSQL.org)。还可以查看 ftp 站点 (<ftp://ftp.PostgreSQL.org/pub>), 看看是否有更新的版本和补丁。

(14) PostgreSQL 与其他 DBMS 相比如何?

评价软件有很多种途径: 特征、性能、可靠性、支持、价格。

特征:

PostgreSQL 具有大型 DBMS 的绝大部分特征, 如事务处理、子查询、触发器、视图、外键参照完整性和复杂锁定。PostgreSQL 还有别的系统不具备的特征, 如用户自定义类型、继承、规则和多版本并发控制以减少互锁冲突。PostgreSQL 目前没有外部连接, 不过正在准备在下一个版本中增加这个特性。

性能:

PostgreSQL 有两种运行模式。常规 fsync 模式将每个完成的事务写到磁盘上，保证在操作系统崩溃或几秒钟后系统就将掉电的情况下，所有的数据都安全地存放在磁盘上。在这种模式下，PostgreSQL 比大多数商用数据库的速度慢，部分原因是极少有人在默认模式下这么保守地不停往磁盘上存数据。在 no-fsync 模式下，PostgreSQL 通常比商用数据库快，可是在这种模式下，操作系统的崩溃会导致数据出错。PostgreSQL 的开发者正致力于提供一种中间模式，性能比 fsync 模式稍弱，而且将保证操作系统崩溃时 30 秒钟内的数据的完整性。

与 MySQL 或更轻便的数据库系统相比，PostgreSQL 的插入和更新速度较慢，因为 PostgreSQL 有额外的事务处理。当然，MySQL 不具备前述的任何一种特征。PostgreSQL 注重灵活性和专门特征，虽然不断通过优化和分析源代码来改进性能。一个有趣的将 PostgreSQL 与 MySQL 进行比较的 Web 页面为 <http://openacs.org/why-not-mysql.html>。

PostgreSQL 创建一个 UNIX 进程来处理每个用户连接。后台进程分享数据缓冲区并信息互锁。多 CPU 情况下，多个后台进程可以运行于不同的 CPU 上。

可靠性：

PostgreSQL 的作者们认识到一个 DBMS 必须可靠，否则就毫无价值。他们尽量发布经过反复测试的、含最少 bug 的稳定代码。每个版本经过至少一个月的 β 测试，而 PostgreSQL 的发行历史显示作者们能提供稳定、可靠的版本。

支持：

PostgreSQL 的邮件列表提供一个巨大的开发者组 and 用户组，帮助解决遇到的任何问题。这些组织不能保证修补，商用 DBMS 也不是总能提供修补。直接访问开发者、用户群体、手册和源代码常常使 PostgreSQL 的支持优于其他 DBMS。还有一些商业性的预包装和支持，需要的人可以使用它们。

价格：

PostgreSQL 对所有用户免费，无论是商用还是非商用。用户可以无限制地在自己的产品中加入 PostgreSQL 的代码，除了前面声明的 BSD 类型许可中所提到的那些。

## 2. 客户问题

### (1) PostgreSQL 有 ODBC 驱动吗？

有两种 ODBC 驱动可用：PsqlODBC 和 OpenLink ODBC。

PsqlODBC 包含在发行版本中，更多信息见 <ftp://ftp.PostgreSQL.org/pub/odbc/>。

OpenLink ODBC 见 <http://www.openlinksw.com>。它带有标准的 ODBC 客户软件，因此用户可以在他们支持的所有客户平台（Win, Mac, UNIX, VMS）上使用 PostgreSQL ODBC。他们可能向需要商用质量支持的用户出售该产品，不过免费软件随处可见。详情请通过 [postgres95@openlink.co.uk](mailto:postgres95@openlink.co.uk) 查询。

还可查看程序员指南中的 ODBC 章节。

### (2) 用何种工具可以将 PostgreSQL 用于 Web 页面？

介绍支持数据库的 Web 页面见 <http://www.webtools.com> 以及 <http://www.phone.net/home/mwm/hotlist/>。





对集成的 web, PHP 是一种优秀的界面, 详情情浏览 <http://www.php.net>。

对复杂的情况, 一般使用 Perl 和 CGI。

基于 WDBusing Perl 的 WWW 网关可在 <http://www.eol.ists.ca/dunlop/wdb-p95> 站点下载。

(3) PostgreSQL 有图形用户接口、报表生成器吗? 有内嵌的查询语言接口吗?

PostgreSQL 有一个很好的称为 pgaccess 的图形用户接口, 它是每个发行版本的一部分, 它还有一个报表生成器。详情见 Web 页面 <http://www.flex.ro/pgaccess>。

PostgreSQL 还包含 ecpg, 它是嵌入式的 C 语言 SQL 查询接口。

(4) 哪些语言可以和 PostgreSQL 通信?

可以与 PostgreSQL 通信的语言有:

- C (libpq)
- C++ (libpq++)
- Embedded C (ecpg)
- Java (jdbc)
- Perl (perl5)
- ODBC (odbc)
- Python (PyGreSQL)
- TCL (libpgtcl)
- C Easy API (libpgeasy)
- Embedded HTML (PHP from <http://www.php.net>。)

### 3. 管理员问题

(1) initdb 为何失败?

试一试:

- 检查 PostgreSQL 目录下是否有以前版本的二进制文件。
- 检查是否有正确的路径设置。
- 检查 postgres 用户拥有正确的文件。

(2) 如何在/usr/local/pgsql 之外的其他地方安装 PostgreSQL?

最简单的方法是在运行 configure 时指定-prefix 选项。如果忘了这么做, 可以编辑 Makefile.global 来修改 PostgreSQL, 或创建 Makefile.custom, 在那里定义 PostgreSQL。

(3) 启动 postmaster 时, 出现 Bad System Call 或 core dumped 信息, 为什么?

问题可能是多种多样的, 不过首先要检查的是系统内核是否安装的 System V 扩展版。PostgreSQL 要求内核支持内存共享和信号量。

(4) 启动 `postmaster` 时，出现 `IpcMemoryCreate` 错误，为什么？

系统内核还是没有配置好共享内存或需要增大内核可用的共享内存。共享内存的确切数目与体系结构、为 `postmaster` 配置的缓冲区大小以及后台进程的数目有关。对大多数系统，默认的缓冲区及后台进程数，至少需要 1MB。

(5) 启动 `postmaster` 时，出现 `IpcSemaphoreCreate` 错误，为什么？

如果错误信息是 `IpcSemaphoreCreate: semget failed (No space left on device)`，那么系统内核没有配置足够的信号量。`Postgres` 每个潜在的后台进程需要一个信号量。一个临时的解决方案是启动 `postmaster` 时控制后台进程的数目，用 `-N` 参数，就可以使数目小于默认的 32 个。更持久的解决方法是：扩大内核的 `SEMMNS` 和 `SEMMNI` 参数。

如果是其他错误，系统内核可能根本没有配置信号量。

(6) 怎样防止其他主机访问本地 PostgreSQL 数据库？

默认情况下，`PostgreSQL` 只允许从使用 `UNIX` 套接字的本地机器连接。其他机器不能连接，除非在 `postmaster` 加上 `-i` 标志，并且适当地修改文件 `$PGDATA/pg_hba.conf` 允许主机识别。这样将允许 `TCP/IP` 连接。

(7) 为什么用户不能从别的机器连接数据库？

默认情况下，`PostgreSQL` 只允许从使用 `UNIX` 套接字的本地机器连接。为了允许 `TCP/IP` 连接，必须保证 `postmaster` 已加上 `-i` 标志，在文件 `pgsql/data/pg_hba.conf` 中加入正确的主机项目。

(8) 为什么用户不能以根用户身份访问数据库？

不应该创建用户标识为“0”（根）的用户。他们不能访问数据库。这是一种安全防范，因为这样的用户有能力将对象模块动态的连接数据库引擎。

(9) 并发访问表使所有服务器崩溃，为什么？

这个问题可能由于内核没有配制对信号量的支持。

(10) 怎样调整数据库引擎获得更好性能？

当然，索引可以加速查询。`EXPLAIN` 命令可以让用户看到 `PostgreSQL` 是怎样解析查询命令的，以及使用了哪些索引。

如果用户进行大量的 `INSERT` 操作，考虑利用 `COPY` 命令在一个批处理中来完成这种工作，这比单独的 `INSERT` 快得多。再有，不在 `BEGIN WORK/COMMIT` 事务处理块中声明的语句被认为是独立的事务，因此应该考虑在单独的事务处理块中进行声明。这样能够有效地减少事务的开销。在进行了大量的数据更新后，还要考虑删除和重建索引。

还有一些调整选项。用户可以在启动 `postmaster` 时通过禁用“`-o -F`”选择项来屏蔽 `fsync()`。这可以避免 `fsync()` 每次事务完成后都写磁盘。

用户可以用 `postmaster -B` 选项来增加后台进程所使用的共享内存数。如果将这个参数



设置得过大, `postmaster` 可能不能启动, 因为已超出内核对共享内存的限制。每个缓冲区 8k, 默认 64 个缓冲区。

用户还可以用后台的 `-S` 选项来增大后台进程临时排序所用的内存数。`-S` 值按 Kb 计算, 默认值是 512(即 512k)。

用户还可以用 `CLUSTER` 命令把表中的数据编组以适应一个索引。详见 `CLUSTER` 手册。

### (11) 有哪些调试方法?

PostgreSQL 有几种特性可以报告状态信息, 这些信息可用于调试。

首先, 运行 `configure` 时带 `-enable-cassert` 选项, 很多 `assert()` 监控后台进程, 在程序发生意外情况时终止程序。

`Postmaster` 和 `postgres` 都有几个调试选项。第一, 启动 `Postmaster` 时, 确保将标准输出和错误信息送到日志文件, 例如:

```
cd /usr/local/pgsql
./bin/postmaster > server.log 2>&1 &
```

这会在 PostgreSQL 的顶级目录中放置一些日志文件。这些文件中包含关于服务器遇到的问题和错误的有用的信息。`postmaster` 的 `-d` 选项可以提供更详细的信息报告。`-d` 选项用一个数字来控制调试级别。注意, 高级别的调试会产生很大的日志文件。

如果 `postmaster` 没有运行, 用户可以从命令行在后台运行 `postgres`, 并直接输入 SQL。这仅仅只能作为调试手段。注意换行就终止查询, 而不是用分号。如果在编译时使用了调试符号, 可以用调试器观察正在发生的事件。因为后台不是从 `postmaster` 开始的, 它运行于不同的环境, 互锁及后台冲突等问题可能无法模拟出来。

如果 `postmaster` 正在运行, 在一个窗口中启动 `psql`, 然后找到 `psql` 所用的 `postgres` 的 PID。用调试器找到 `postgres` 的 PID。可以设置断点并从 `psql` 发出询问。如果要调试 `postgres` 的启动, 设置 `PGOPTIONS="-Wn"`, 然后启动 `psql`。这可以使启动延迟 `n` 秒, 用户就可以用调试器跟踪启动过程。`postgres` 有 `-s`, `-A` 和 `-t` 选项, 它们对调试和性能评价很有用。

用户还可以带 `profiling` 编译 PostgreSQL, 查看哪些功能占用了执行时间。后台的 `profile` 文件放在 `pgsql/data/base/dbname` 目录中。客户的 `profile` 文件放在客户当前的目录下。

### (12) 试图连接时报 `ieSorry, too many clientslg` 错, 为什么?

用户需要增大 `postmaster` 设定的当前可启动的后台进程数。

PostgreSQL 6.5 及以上版本, 默认的后台进程数目为 32。用户可以重新启动 PostgreSQL, 带一个适当的 `-N` 值。在默认配置下, `-N` 值最大可达 1024。如果用户还需要更大的进程数目, 增大 `include/config.h` 中的 `MAXBACKENDS` 并重新编译。如果用户喜欢, 可以用 `config` 的 `with-maxbackends` 开关, 在配置时设定默认的 `-N` 值。

注意, 如果系统的 `-N` 值大于 32, 用户还必须把 `-B` 值设置成大于默认的 64, `-B` 值至少是 `-N` 值的两倍, 而且最好超过两倍才有最好的性能。后台进程数大时, 很可能发现还要增大 UNIX 内核的各种配置参数。要检查的包括: 共享内存块的最大数目 `SHMMAX`、最大信号量个数 `SEMMNS` 和 `SEMMNI`、最大进程数 `NPROC`、每个用户最大进程数

MAXUPRC、最大打开文件数 NFILE and NINODE。PostgreSQL 限制后台进程数是为了系统不至于资源不够。

PostgreSQL 6.5 以前的版本中，最大后台进程数是 64，要改动它需要修改文件 include/storage/sinvaladt.h 中的 MaxBackendId，然后重新编译。

#### (13) 数据库目录下的 pg\_sorttempNNN.NN 文件是什么？

他们是执行查询时产生的临时文件。例如，如果要做依次排序，要求的空间大于后台 -S 参数允许的，就产生一个临时文件存放超出部分的数据。临时文件会被自动删除，但排序时后台出错则可能没被删除。如果系统没有运行后台程序，删除 pg\_tempNNN.NN 是安全的。

### 4. 操作的问题

#### (1) 为什么系统混淆逗号、小数点和数据格式？

检查本地配置。PostgreSQL 用本地用户的设置运行 postmaster 程序。有 postgres 和 psql 设置命令控制数据格式，用户应该按系统操作环境设置它们。

#### (2) 二进制游标和普通游标的区别到底是什么？

见 DECLARE 手册。

#### (3) 怎样仅选择一个查询的前几行？

见 FETCH 手册。或用 SELECT……LIMIT……。

即使只需要前几行，也可能必须完成整个查询。考虑一个带有 ORDER BY 的查询，如果有一个与 ORDER BY 匹配的索引，PostgreSQL 可能值计算需要的前几行，否则可能要执行全部查询来找到所需的几行。

#### (4) 怎样将 psql 中可见的表或其他内容列表显示？

可以阅读文件 pgsqllib/src/bin/psql/psql.c 中的 psql 源代码。其中包括生成 psql 反斜线命令输出的 SQL 命令。用户还可以在启动 psql 时带 -E 选项，它将会把执行命令所用到的查询打印出来。

#### (5) 怎样从一个表中删除一列？

PostgreSQL 目前不支持 ALTER TABLE DROP COLUMN，不过可以这样做：

```
SELECT .....          --选择除用户想删除的行之外的所有行
INTO TABLE new_table
FROM old_table;
DROP TABLE old_table;
ALTER TABLE new_table RENAME TO old_table;
```

#### (6) 行、表、数据库的最大容量分别是多少？



这些限制是：

- 数据库最大容量：无限制（存在 60GB 的数据库）
- 表的最大容量：在所有操作系统中都无限制
- 一行的最大容量：8k 可配置到 32k
- 表的最大行数：无限制
- 表的最大列数：无限制
- 表的最多索引数：无限制

当然，并不是真的无限制，而要受可用磁盘空间的限制。要改变最大行容量，编辑 `include/con_g.h` 文件，改变 `BLCKSZ` 的值。要用大于 8k 的属性，用户还可以使用大对象界面。7.1 版中行长度的限制将被取消。

(7) 从一个典型的文本文件中录入数据需要多大数据库磁盘空间？

一个 PostgreSQL 数据库可能需要 6.5 倍的磁盘空间来存放一个平面文件中的数据。若一个文件有 3000 行，每行有两个整数，平面文件的大小是 2.4MB。包含这些数据的 PostgreSQL 数据库文件估计有 14MB。

36 bytes: each row header (approximate)

+ 8 bytes: two int fields @ 4 bytes each

+ 4 bytes: pointer on page to tuple

-----

48 bytes per row

The data page size in PostgreSQL is 8192 bytes (8 KB), so:

8192 bytes per page

----- = 171 rows per database page (rounded up)

48 bytes per row

300000 data rows

----- = 1755 database pages

171 rows per page

1755 database pages \* 8192 bytes per page = 14,376,960 bytes (14MB)

索引的容量不需要上述那么多，不过要包含索引数据，容量还是很大的。

(8) 怎样找出数据库中定义的索引和操作？

`psql` 有各种反斜线命令可以显示这些信息。用 `\?` 可看见它们。

还可以试试文件 `pgsql/src/tutorial/syscat.source`。它说明了很多从数据库系统表中获取信息的 `SELECTs`。

(9) 查询很慢或不能用索引，为什么？

PostgreSQL 不能自动维护统计表。必须运行 `VACUUM` 来更新统计表。统计表更新后，优化器知道表中有多少行，才能决定是否该用索引。注意，优化器在表很小时不用索引，因为顺序扫描更快。

对指定列的优化统计表,使用 `VACUUM ANALYZE`。`VACUUM ANALYZE` 对复杂的多关联查询很重要,优化器可以估计每个表返回的行数,选择合适的连接顺序。后台不会自动掌握列统计表,所以必须运行 `VACUUM ANALYZE` 周期性地收集这些信息。

索引通常不用于 `ORDER BY` 操作,直接排序进行顺序扫描比用索引扫描一个大的表要快,因为它访问磁盘较少。

使用通配符如 `LIKE` 或 `~`,只有搜索从一个字符串的头开始时才能用索引。所以,要用索引,`LIKE` 搜索不能以 “%” 开头,而 `~` 搜索应该以 “^” 开头。

(10) 怎样才能看到查询优化器如何评价我的查询?

见 `EXPLAIN` 手册。

(11) 什么是 R-tree 索引?

`R-tree` 索引用于对空间数据进行索引。`Hash` 索引不能控制平行搜索。`B-tree` 索引只能控制一维的平行搜索。`R-tree` 索引可控制多维数据。例如:对一个分数类型的属性建立 `R-tree` 索引,系统可以更有效地回答诸如选择一个矩形约束框中的所有点这样的查询。

描述 `R-tree` 索引设计的规范见:Guttman, A.《`R-Trees`: 用于空间搜索的动态索引结构》*Proc of the 1984 ACM SIGMOD Int'l Conf on Mgmt of Data*, 45-57

用户还可以在 Stonebraker 的《数据系统读物》中找到它们。

建立 `R-tree` 索引可以控制多边形和方形。理论上,`R-tree` 索引还可以扩展到控制更高的维数。实际上,扩展 `R-tree` 索引需要做不少工作,PostgreSQL 目前还没有任何这方面的文档。

(12) 什么是 Genetic Query Optimization?

`GEQO` 模式可以在很多表按遗传算法 (`GA`) 关联时加速查询优化。它允许通过非详尽搜索控制大量关联查询。

(13) 怎样进行规则表达式搜索和大小写无关的规则表达式搜索?

“`~`”运算符处理规则表达式匹配,“`~*`”运算符处理大小写无关规则表达式匹配。`LIKE` 运算符没有大小写无关的用法,不过用户可以通过下面的用法获得大小写无关的 `LIKE`:

```
WHERE lower(textfield) LIKE lower(pattern)
```

(14) 查询中如何发现字段为空 `NULL`?

可以用 `IS NULL` 和 `IS NOT NULL`。

(15) 各种字符类型间的差别是什么?

`CHAR`——`char`, 1 个字符。

`CHAR(#)`——`bpchar`, 用空格填补为指定长度。

`VARCHAR(#)`——`varchar`, 指定最大长度, 不填补。

`TEXT`——`text`, 长度只受行最大长度限制。



BYTEA——bytea, 可变长度字节数组。

用户将在检查系统目录时和一些错误信息中看到内部名称。

后 4 种类型是 VARLENA 类型, 即前 4 个字节是长度, 其后是数据。CHAR(#)不论文件中存放了多少数据都分配最大字节数。TEXT、VARCHAR(#)和 BYTEA 在磁盘上有可变长度, 所以它们对性能有一些影响。准确地说, 除第一种字符类型外, 其他字符类型都对性能有所影响。

(16) 怎样创建一个连续/自动增量的字段?

PostgreSQL 支持 SERIAL 数据类型。它自动对行创建序列与索引。例如:

```
CREATE TABLE person (id SERIAL, name TEXT)
```

被 PostgreSQL 自动翻译为:

```
CREATE SEQUENCE person_id_seq;
```

```
CREATE TABLE person (id INT4 NOT NULL DEFAULT nextval('person_id_seq'),  
name TEXT);
```

```
CREATE UNIQUE INDEX person_id_key ON person (id);
```

关于序列的详细信息见 CREATE\_SEQUENCE 手册。还可以用每行的 OID 域作为特征值。然而, 如果需要卸掉并重装载数据库, 用户必须用 pg\_dump 的 -o 选项或 COPY WITH OIDS 选项来保护 OIDs。

(17) 怎样从 SERIAL 插入中获得值?

一种方法是插入前用函数 nextval() 从序列对象获得下一个 SERIAL 值, 然后把它明确地插入到表中。例如:

```
$newSerialID = nextval('person_id_seq');
```

```
INSERT INTO person (id, name) VALUES ($newSerialID, 'Blaise Pascal');
```

用户还可以把新的值存在 \$newSerialID 中共其他查询用 (例如, 作为 person 表的 foreign key)。注意, 自动创建的 SEQUENCE 对象将被命名为 <table>\_<serialcolumn>\_seq, 其中 table 和 serialcolumn 分别是表名和 SERIAL 列名。

另一种方法是插入后用函数 currval() 获得刚刚插入的 SERIAL 值, 例如:

```
INSERT INTO person (name) VALUES ('Blaise Pascal');
```

```
$newID = currval('person_id_seq');
```

最后, 用户可以用从 INSERT 声明中返回的 OID 查看默认值, 这可能是最简便的方法。在 Perl 中, 用 BDI 的 Edmund Mergl's DBD::Pg 模式, 通过 \$sth->{pg\_oid\_status} after \$sth->execute() 使 OID 值可见。

(18) currval() 和 nextval() 会导致并发问题吗?

不会。这是由后台控制的。

(19) 什么是 OID? 什么是 TID?

OID 是 PostgreSQL 唯一的行标识。每个在 PostgreSQL 中生成的行都有一个唯一的 OID。

所有在 initdb 期间生成的 OID 小于 16384（来自 backend/access/transam.h）。所有用户创建的 OID 大于等于 16384。默认情况，这些 OID 不仅在一个表，一个数据库甚至在整个 PostgreSQL 安装范围内是唯一的。

PostgreSQL 在它的内部系统表中利用用户 OID 跨表连接各行。这些 OID 可用于标识指定的用户行和用于关联。建议用户用 column 类型 OID 存放 OID 变量。用户可以对 OID 字段创建索引来加速访问。

所有数据库都用到的中枢区域给每个表的行分配 OID。如果用户想把这个 OID 改成别的值，或者想复制一个带有原 OID 的表，可以使用下面的命令：

```
CREATE TABLE new_table(old_oid oid, mycol int);
SELECT old_oid, mycol INTO new FROM old;
COPY new TO '/tmp/pgtable';
DELETE FROM new;
COPY new WITH OIDS FROM '/tmp/pgtable';
```

TID 用于标识指定的带有块和偏移量的物理行。行被修改或重装后 TID 改变。它们被索引用于指向物理行。

#### (20) PostgreSQL 中一些术语的含义。

一些源代码和老文档用到的术语用得很普遍。有：

- 表，关系，类——table, relation, class
- 行，记录，tuple——row, record, tuple
- 列，域，属性——column, field, attribute
- 查找，选择——retrieve, select
- 替换，更新——replace, update
- 添加，插入——append, insert
- OID，serial 值——OID, serial value
- 入口，指针——portal, cursor
- 范围变量，表名，表别名——range variable, table name, table alias

#### (21) 为什么出现错误信息“FATAL: palloc failure: memory exhausted?”

可能是系统虚拟内存溢出，或内核对某种资源的限制太紧。启动 postmaster 前试试：

```
ulimit -d 65536
```

```
limit datasize 64m
```

这与 shell 相关，只有其中之一可能成功。不过，这些命令可以将进程数据段的限制放宽，因此可能完成查询。如果用户的问题与 SQL 客户端有关，后台返回太多数据，启动客户端前试一试。

#### (22) 怎样获知正在运行的 PostgreSQL 版本？

在 psql 中键入 select version();





(23) large-object 操作出现 invalid large obj descriptor, 为什么?

用户需要把 BEGIN WORK 和 COMMIT 放在大对象句柄前后, 即, 在 lo\_open 和... lo\_clos 前后。目前, PostgreSQL 用在事务提交时关闭大对象句柄来强迫执行这一规则。所以在对句柄做任何操作前要有 invalid large obj descriptor。所以如果用户未用事务处理, 工作代码就会出现错误信息。

如果客户端接口为 ODBC, 用户可能需要将 auto-commit 设置为 off。

(24) 怎样创建一个列作为当前默认列?

用 now():

```
CREATE TABLE test (x int, modtime timestamp DEFAULT now() );
```

(25) 用到 IN 的查询为什么这么慢?

当前, PostgreSQL 把子查询关联到外部查询的方法是对外部查询的每一行顺序浏览子查询的结果。工作区把 IN 用 EXISTS 替代:

```
SELECT * FROM tab WHERE col1 IN (SELECT col2 FROM TAB2)
```

替代为:

```
SELECT * FROM tab WHERE EXISTS (SELECT col2 FROM TAB2 WHERE col1 = col2)
```

希望在以后的版本中修补这一点。

(26) 怎样进行外连接?

PostgreSQL 目前的版本不支持外部关联。可以用 UNION 和 NOT IN 来模拟。例如, 关联 tab1 和 tab2, 下面的查询实现两个表的外部关联:

```
SELECT tab1.col1, tab2.col2
```

```
FROM tab1, tab2
```

```
WHERE tab1.col1 = tab2.col1
```

```
UNION ALL
```

```
SELECT tab1.col1, NULL
```

```
FROM tab1
```

```
WHERE tab1.col1 NOT IN (SELECT tab2.col1 FROM tab2)
```

```
ORDER BY tab1.col1
```

## 5. 扩展 PostgreSQL

(1) 编写了一个用户函数, 为什么在 psql 中运行时出错?

问题可能与很多事情有关。首先在一个独立的测试程序中测试用户自定义函数。

(2) “NOTICE: PortalHeapMemoryFree: 0x402251d0 not in alloc set!” 是什么意思?

用户正用 pfree 释放一些不是 palloc 分配的东西。不要混用 malloc/free 和 palloc/pfree。

(3) 怎样把新类型和函数贡献给 PostgreSQL?

把用户的扩充发送到: `pgsql-hackers` 邮件列表。他们最后会进入 `contrib/`子目录。

(4) 怎样编写 C 函数返回一个元组?

这虽然在理论上可以做到, 但是实现起来非常困难。

(5) 改变了一个源文件, 为什么重编译未发现所作改变?

`Makefiles` 没有适当的考虑 `include` 文件没有适当的。必须进行一次 `make clean`, 然后再进行一次 `make`。

# 配套光盘说明

## 1. 光盘内容

配套光盘包含以下主要文件和目录：

Index.htm

光盘导航主页面

INTRO

本书简介目录

PFY.PDF

扉页与内容摘要

PQY.PDF

前言

PML.PDF

目录

PCH24.PDF

命令与工具参考

ACRD4CHS.EXE

Acrobat Reader 4.0 中文版安装程序

DOCUMENT

文档目录

DOC\_GB

PostgreSQL 文档中文版目录

DOC\_ENG

PostgreSQL 文档英文版目录

MISC

其他文档目录

PostgreSQL FAQ\_GB.html

PostgreSQL 常见问题解答中文版

PostgreSQL FAQ.htm

PostgreSQL 常见问题解答英文版

PSQLODBC FAQ\_GB.html

ODBC 常见问题解答中文版

PSQLODBC FAQ.html

ODBC 常见问题解答英文版

PsqlODBC Configuration.htm

ODBC 配置问题解答

PostgreSQL Developers FAQ.htm

开发者常见问题解答

SOFT

软件目录

PostgreSQL\*

PostgreSQL 源代码

POSTDRV.EXE

ODBC 驱动程序 (Windows)

PHP\*

PHP (Windows)

phpPgAdmin\_2\_1.zip

利用 PHP 编写的 PostgreSQL 管理程序 (Web 界面)

## 2. 导航程序

配套光盘配有简单的导航程序，入口文件为光盘根目录下的 Index.htm，请用 IE 和 Netscape 浏览器浏览，屏幕分辨率最好为 800×600。