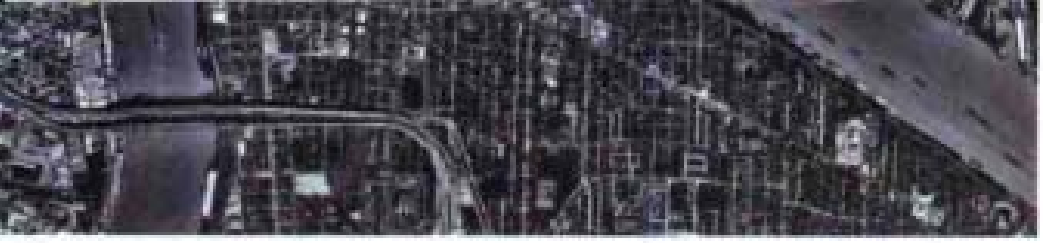
PostgreSQL 必备参考手册

.138SQ

New Riders

〔美〕Barry Stinson 著
云巅工作室 译

人民邮电出版社



美术编辑：胡平利

PostgreSQL 必备参考手册

作为最快、最全面的PostgreSQL向导，本书内容包括：

- 完整的SQL命令参考；
- 创建PostgreSQL函数（包括SQL、过程和C函数）详解；
- PostgreSQL内置操作符、数据类型及函数的完整列表；
- 剖析MVCC（多版本并发系统）及SQL事务操作方式；
- PostgreSQL管理任务（包括创建用户和组、分配权限）；
- PostgreSQL备份和恢复 — 开发自动备份程序以及灾难恢复策略；
- 在PostgreSQL中进行Perl、Python、PHP、C和C++、ODBC及JDBC编程的必备工具。

本书是为PostgreSQL用户及RedHat DB新手量身定做的。对于Web开发员、管理员及程序员也是一本有价值的参考书。本书概念清晰、内容明了，囊括了从基本的SQL命令、常规管理任务到编程应用及使用PostgreSQL API的所有知识。

值得一提的是本书主题鲜明、布局合理、易于查找。每个主题均按任务和字母顺序排列，并提供了功能注释、语法等重要信息及相关实例代码。Barry Stinson撰写此书的目的很明确，这就是让您一旦拥有，别无他求。

本书是第一本完全关于PostgreSQL的技术书籍。其他类似的图书也可能会对PostgreSQL进行深入地探讨，但本书的组织方式使您能够非常方便地查找到所需要的信息。我向所有PostgreSQL的用户郑重推荐此书。

— Korry Douglas，Appx Software研发部主任

ISBN 7-115-10020-9

9 787115 100207 >

ISBN7-115-10020-9/TP·2717
定价：35.00元

人民邮电出版社

TP311.138SQ
S81C

PostgreSQL 必备参考手册

[美]Barry Stinson 著

云巅工作室 译

A0989716

人民邮电出版社

图书在版编目（CIP）数据

Postgre SQL 必备参考手册/（美）斯廷森（Stinson，B.）著；云巅工作室译. —北京：人民邮电出版社，2002.2
ISBN 7-115-10020-9

Ⅰ.P… Ⅱ.①斯…②云… Ⅲ.关系数据库—数据库管理系统，Postgre SQL—技术手册 Ⅳ.TP311.138-62

中国版本图书馆 CIP 数据核字（2002）第 001359 号

版权声明

Barry Stinson: PostgreSQL Essential Reference

Copyright © 2002 by New Riders Publishing.

Authorized translation from the English language edition published by New Riders Publishing.

All rights reserved. For sale in mainland China only.

本书中文简体字版由美国 New Riders 出版公司授权人民邮电出版社出版。未经出版者书面许可，对本书任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

PostgreSQL 必备参考手册

◆ 著　[美] Barry Stinson
译　云巅工作室
责任编辑　陈冀康

◆ 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
读者热线　010-67180876
北京汉魂图文设计有限公司制作
北京鸿伟印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
印张：19.25
字数：452 千字　2002 年 2 月第 1 版
印数：1-4 000 册　2002 年 2 月北京第 1 次印刷

著作权合同登记　图字：01－2001－4083 号

ISBN 7-115-10020-9/TP・2717

定价：35.00 元

本书如有印装质量问题，请与本社联系　电话：(010)67129223

译者的话

在信息量呈指数快速增长的今天，数据库技术也相应得到了蓬勃的发展。现代关系数据库管理系统（RDBMS）更是企业不可缺少的工具。PostgreSQL 是 RDBMS 中的一朵奇葩。PostgreSQL 的魅力主要有两个方面：一是完全开放的源代码适应了 IT 技术发展的新潮流，同时也满足了开发人员进行创造性开发的愿望。也正因为如此，它获得了世界上许多开发团体的技术支持，代码体系日臻完善；二是它具有完整的功能机制。PostgreSQL 不仅包含了现代 RDBMS 所必需的触发器、规则和函数等，还支持多版本并发控制（MVCC）及自定义函数等机制，PostgreSQL 更是提供了广泛的 API 访问解决方案，包括 ODBC、JDBC、C、Perl、PHP 及 Python 等。可以毫不夸张地说，与其他商业化数据库系统相比，尤其针对中小型数据库系统，PostgreSQL 的表现毫不逊色。

本书由富有数据库开发经验的 Barry Stinson 撰写。本书的特色除资料翔实外，还在于编排方式独特，各个主题按列表方式有序排列，便于您的查找。PostgreSQL 支持的所有 SQL 命令也按字母顺序提供了详细的列表。另外，作者几乎对大部分主题均提供了相关实例，让您快速掌握并付之实践。正如作者所说，本书无疑会成为 IT 人士的 PostgreSQL 必备手册。

本书由云巅工作室周良忠博士翻译。译文充分保持了原书作为一本参考手册的编排特色，以期让读者轻松定位知识点。翻译过程中，我们在准确理解的基础上，更力求使行文符合汉语习惯，使读者能顺序学习和掌握所讲授的知识。

由于译者水平有限，错误在所难免，望广大读者不吝指正。云巅工作室联系电子邮件：webmaster@CloudCrown.com。

云巅工作室

2001 年 12 月

前 言

数据库是计算机技术不可缺少的一部分。从某种意义上说，你可以认为计算机最广泛的功能就是作为一个数据库系统。早期的打孔卡片阅读机（它是现代计算机的雏形就被美国政府和大型商业体用来作为搜集、整理及报告数据的工具。这一通常目的也是现代计算机发展的基本目标。

在现代计算机发展的早期阶段，每个程序处理自身的数据存储和读取功能。当然，这给那时的程序员带来了极大的负担，他们不得不编写大量与实用功能无关的额外代码。而且，事实表明，那时的技术很难有效和可靠地保存数据？所以，自然而然诞生了数据库的思想。

早期的数据库系统是一个独特但颇受欢迎的概念。它作为一个独立的应用程序而存在。为了存储数据，程序员再也不需要编写执行低级文件访问功能的代码了。相反，他们可以将其精力投入到程序设计直接所需代码的编写工作中去。应用程序只需简单地通知数据引擎所需保存和读取的数据，数据库系统就可以处理其请求了。20 世纪 60 年代中期，一些公司（如 IBM）售出了若干数据库管理系统，它们满足了以上所提到的许多功能。

早期的数据库系统存在两个问题：一个是兼容性问题，一个是它们只有处理静态数据结构才能更好地工作。因为每个开发商所售系统都是唯一和专有的数据库管理系统，应用程序必须针对不同的系统编写不同的接口。如果所编写的应用程序用于与 IBM 数据库接口，它就不能轻易与其他系统协调工作，反之亦然。而且，早期数据库所处理的数据库为“单调文件”。这意味着如果你想从应用程序中获取另外一套数据，对基本数据结构的修改工作就会十分艰难和耗时。在许多情况下，应用程序的很大部分源代码都必须重写来完成修改工作。

20 世纪 70 年代早期，一个 IBM 研究员 E. F. Codd 所写的一篇论文从根本上改变了实现数据库系统的历史。Codd 建议数据库系统必须根据关系来表达数据集。这就是说，数据库中的表可以通过与不同的索引相互连接生成相应的数据结构，这样的数据结构比以前的单调文件系统更灵活和开放。

IBM 着手开发融入了许多 Codd 的观点的数据库系统，这就是所谓的 R 系统。R 系统在 20 世纪 70 年代中期完成，它还包含了一

个新的功能，这就是众所周知的结构化查询语言（Stnuted very language,SQL）。这一新语言为数据库世界提供了两个非常基本的概念：（1）它是声明性的；（2）美国国家标准协会（ANSI）批准它为数据库标准。

在 SQL 语言出现前，程序员必须从过程上定义数据库所保存数据的访问方式。然而，通过 SQL，程序员只要简单地提供返回数据集中的取舍标准，数据库引擎就会将此请求翻译成所需数据。这就减少了程序员的额外负担，因为他们再也不需要控制实际的数据输入和读取工作了。程序员可将大量精力花在设计查询规划器上，使数据库以最有效的方式执行查询请求。通过这一集体的努力，数据库的有效性和可靠性达到了一个新高度，这是单个开发人员无法完成的。

因为 SQL 被 ANSZ 接纳为数据库的标准语言，所以其他竞争产品可以只有相同的接口。开发人员可以设计兼容性更加良好的应用程序。这意味着当数据库系统改进和要求更改后，应用程序可以比以前更加容易地移植到新数据库系统上。

关系数据结构和 SQL 语言这两个发展点构成了现代关系数据库管理系统（RDBMS）的基石。后来，RDBMS 发展成一个具有自身特色的技术体系。

Ingres 是早期 RDBMS 之一，它包含了当时数据库系统的许多有用功能。20 世纪 90 年代加利福尼亚大学（University of Californai）伯克利（Berkeley）分校启动了一个旨在推进这一思想的项目；此项目冠名为 Postgres，意为 Ingres 之后的系统。

在此 10 年之间，此项目（即后来为人熟知的 Postgres95）更名为 PostgreSQL 并作为开放源代码项目向世界发行。在后续几年间，此项目取得了巨大的发展，并最终发展成一个源代码开放、功能强大、免费获取的数据库系统。这是它与其他高利润商业化 RDBMS 竞争的结果。这是一个值得纪念的成就，而且应该对那些为此项目奉献了宝贵时间和努力的无数开发者致以崇高的敬意。

事实上，PostgreSQL 如今已经吸引了许多商业化的进一步支持和开发。在这些感兴趣的公司中有 Great Bridge 和 Red Hat，两者都对开放源代码和 PostgreSQL 作了一定程度的探索。因为开发团体拥有了商业化的支持，PostgreSQL 注定要成为开放源代码领域里的一颗明星。PostgreSQL 无疑与 Apache 和 Linux 取得了同样的成功。

本书内容

为使本书成为一本名副其实的“参考”手册，我们付出了许多努力。参考手册与传统书籍的写作方法迥然不同。也就是说，参考手册的作者必须时刻牢记所写书的实际使用方式。

参考手册不必顺序阅读；读者可根据需要跳跃式阅读。所以，书籍必须精心布局、精心组织，而且让每个主题的信息量适中。

我尽力按以上原则来撰写本书。本书包含的知识按下面结构组织。

第一部分：SQL 参考

这一部分用单独的一章（即第 1 章“PostgreSQL SQL 参考”）列出了 PostgreSQL 7.1 版本所支持的所有 SQL 命令。每个命令按其字母顺序排列，同时提供了相关用法和实例。

第二部分：PostgreSQL 规范

PostgreSQL 规范包括：

- 第 2 章“PostgreSQL 数据类型”，给出了有效的 PostgreSQL 数据类型及其典型用法的列表。
- 第 3 章“PostgreSQL 操作符”，给出了 PostgreSQL 中操作符列表，同时提供了应用实例。
- 第 4 章“PostgreSQL 函数”，给出了 PostgreSQL 中新包括函数的列表，同时提供了应用实例。
- 第 5 章“其他 PostgreSQL 主题”，内容包括表继承、B-Tree 索引及 OID。本章还讨论了多版本并发控制（MVCC）机制。

第三部分：PostgreSQL 管理

本部分内容旨在帮助数据库管理员（DBA）理解 PostgreSQL 系统的操作方式。详细内容有：

- 第 6 章“用户可执行文件”，介绍用于数据库用户执行的文件。
- 第 7 章“系统可执行文件”，介绍实现系统和服务器功能的文件。
- 第 8 章“系统配置文件和库”，介绍 PostgreSQL 所需的配置文件。
- 第 9 章“数据库和日志文件”，介绍数据库和日志文件本地保存位置。
- 第 10 章“常规管理任务”，简要介绍 DBA 需要执行的常规管理任务。

第四部分：用 PostgreSQL 编程

本部分介绍了开发自定义 PostgreSQL 应用程序可用的选项。包含以下内容：

- 第 11 章“服务器端编程”，介绍了 PL/pgSQL、PL/pgTCL 和 PL/Perl 过程脚本语言。
- 第 12 章“创建自定义函数”，介绍了自定义函数、触发器及规则的使用。
- 第 13 章“客户端编程”，描述了客户端应用程序与后端的接口方式。介绍了 Python、Perl、Libpq、Libpq++、Libpgeasy、Ecpg、ODBC 及 JDBC 接口。
- 第 14 章“高级 PostgreSQL 编程”，进一步介绍了 PostgreSQL 的自定义数据类型、操作符及触发器的创建方法。

第五部分：附录

两个附录提供了有关 PostgreSQL 的进一步信息：

附录 A，“参考资源”，从邮件列表到商业站点，提供技术支持列表。

附录 B，“PostgreSQL 版本信息”，对历年来 PostgreSQL 发行版本的变化进行了回顾。

本书读者对象

本书适用于具备了 SQL 类数据库基础知识但需要获取 PostgreSQL 特定信息的读者。

一般需要读者对 UNIX 类型操作系统有一定的了解。并非严格要求如此，但熟悉 UNIX 类操作系统会使执行安装和管理等任务更容易。

前面提到过，本书更适合于作为技术手册而不宜作为一本教材。而且，根据本书组织方式，跳跃式阅读也许比逐章阅读更有利。无论如何，我真诚地希望本书成为您桌上的一本必备参考书。

约定

本书使用了如下排版约定：

Courier New 等宽字体表示网址、关键字、命令、文件路径及选项。*斜体*表示你必须用自定义值作替换的地方。

在 SQL 语句中，SQL 关键字和函数名用大写。数据库、表和列名用小写。

目　录

第一部分　SQL 参考

第 1 章　PostgreSQL SQL 参考 …… 3

1.1　命令表 …… 3

1.2　命令列表(按字母顺序排列) …… 4

Abort …… 4

ALTER GROUP …… 5

ALTER TABLE …… 6

ALTER USER …… 7

BEGIN …… 8

CLOSE …… 10

CLUSTER …… 11

COMMENT …… 12

COMMIT …… 13

COPY …… 14

CREATE AGGREGATE …… 16

CREATE DATABASE …… 17

CREATE FUNCTION …… 18

CREATE GROUP …… 19

CREATE INDEX …… 20

CREATE LANGUAGE …… 22

CREATE OPERATOR …… 23

CREATE RULE …… 25

CREATE SEQUENCE …… 26

CREATE TABLE …… 28

CREATE TABLE AS …… 33

CREATE TRIGGER …… 34

CREATE TYPE …… 35

CREATE USER …… 37

CREATE VIEW …… 38

DECLARE …… 39

DELETE 40
DROP AGGREGATE 40
DROP DATABASE 41
DROP FUNCTION 42
DROP GROUP 42
DROP INDEX 43
DROP LANGUAGE 43
DROP OPERATOR 44
DROP RULE 45
DROP SEQUENCE 45
DROP TABLE 46
DROP TRIGGER 46
DROP TYPE 47
DROP USER 48
DROP VIEW 48
END 49
EXPLAIN 50
FETCH 51
GRANT 53
INSERT 54
LISTEN 55
LOAD 55
LOCK 56
MOVE 58
NOTIFY 59
REINDEX 60
RESET 60
REVOKE 61
ROLLBACK 62
SELECT 63
SELECT INTO 67
SET 68
SHOW 72
TRUNCATE 72
UNLISTEN 73
UPDATE 73
VACUUM 74

第二部分 PostgreSQL 规范

第 2 章 PostgreSQL 数据类型 …… 79

2.1 数据类型表 …… 79
2.2 几何数据类型 …… 80
BOX …… 80
CIRCLE …… 81
LINE …… 81
LSEG …… 81
PATH …… 82
POINT …… 82
POLYGON …… 83
2.3 逻辑数据类型 …… 83
BOOLEAN …… 83
2.4 网络数据类型 …… 84
CIDR …… 84
INET …… 84
MACADDR …… 85
2.5 数字数据类型 …… 85
BIGINT（或 INT8） …… 86
DECIMAL（或 NUMERIC） …… 86
DOUBLE PRECISION（或 FLOAT8） …… 86
INTEGER（或 INT4） …… 86
REAL（或 FLOAT4） …… 87
SERIAL …… 87
SMALLINT（或 INT2） …… 87
2.6 字符串数据类型 …… 88
CHAR（或 CHARACTER） …… 88
TEXT …… 88
VARCHAR（或 CHARACTER VARYING） …… 88
2.7 时间数据类型 …… 89
DATE …… 89
INTERVAL …… 90
TIME …… 91
TIME WITH TIME ZONE …… 91
TIMESTAMP …… 92
2.8 其他数据类型 …… 92
BIT 和 BIT VARYING …… 92
MONEY …… 92

NAME …… 93
OID …… 93
2.9 更多的数据类型 …… 93

第 3 章 PostgreSQL 操作符 …… 95

3.1 几何类操作符 …… 97
列表 …… 97
注释/示例 …… 97
3.2 逻辑类操作符 …… 98
列表 …… 98
注释/示例 …… 98
3.3 网络类操作符 …… 98
列表 …… 98
注释/示例 …… 98
3.4 数字类操作符 …… 99
列表 …… 99
注释/示例 …… 99
3.5 字符串操作符 …… 100
列表 …… 101
注释/示例 …… 101
3.6 时间操作符 …… 101
列表 …… 101

第 4 章 PostgreSQL 函数 …… 103

4.1 函数表（按类别分组） …… 103
4.2 聚集函数 …… 105
AVG …… 105
COUNT …… 106
MAX …… 106
MIN …… 106
STDDEV …… 106
SUM …… 107
VARIANCE …… 107
4.3 转换函数 …… 107
CAST …… 107
TO_CHAR …… 108
TO_DATE …… 111
TO_NUMBER …… 112
TO_TIMESTAMP …… 112
4.4 几何类函数 …… 112

AREA …… 112
BOX …… 113
CENTER …… 113
CIRCLE …… 113
DIAMETER …… 113
HEIGHT …… 114
ISCLOSED …… 114
ISOPEN …… 114
LENGTH …… 114
LSEG …… 114
NPOINT …… 115
PATH …… 115
PCLOSE …… 115
POINT …… 115
POLYGON …… 116
POPEN …… 116
RADIUS …… 116
WIDTH …… 117
4.5 网络类函数 …… 117
ABBREV …… 117
BROADCAST …… 117
HOST …… 117
MASKLEN …… 118
NETMASK …… 118
NETWORK …… 118
TEXT …… 118
TRUNC …… 118
4.6 数字类函数 …… 119
ABS …… 119
ACOS …… 119
ASIN …… 119
ATAN …… 120
ATAN2 …… 120
CBRT …… 120
CEIL …… 120
COS …… 120
COT …… 120
DEGREES …… 121
EXP …… 121
FLOOR …… 121

LN …… 121
LOG …… 121
PI …… 122
POW 或 POWER …… 122
RADIANS …… 122
RANDOM …… 122
ROUND …… 123
SIN …… 123
SQRT …… 123
TAN …… 123
TRUNC …… 123
4.7 SQL 类函数 …… 124
CASE WHEN …… 124
COALESCE …… 125
NULLIF …… 125
4.8 字符串类函数 …… 125
ASCII …… 125
CHR …… 126
INITCAP …… 126
LENGTH、CHAR_LENGTH 或 CHARACTER_LENGTH …… 126
LOWER …… 127
LPAD …… 127
LTRIM …… 128
OCTET_LENGTH …… 128
POSITION …… 128
STRPOS …… 129
RPAD …… 129
RTRIM …… 129
SUBSTRING …… 130
SUBSTR …… 130
TRANSLATE …… 130
TRIM …… 131
UPPER …… 131
4.9 时间类函数 …… 131
AGE …… 132
CURRENT_DATE …… 132
CURRENT_TIME …… 132
CURRENT_TIMESTAMP …… 132
DATE_PART …… 133
DATE_TRUNC …… 134

EXTRACT …… 134
ISFINITE …… 135
NOW …… 135
TIMEOFDAY …… 135
TIMESTAMP …… 135
4.10 用户类函数 …… 136
CURRENT_USER …… 136
SEEESON_USER …… 136
USER …… 137
4.11 其他类函数 …… 137
ARRAY_DIMS …… 137

第 5 章 其他 PostgreSQL 主题 …… 138

5.1 字段中的数组 …… 138
创建一个数组 …… 138
使用数组字段 …… 138
多维数组 …… 139
扩展数组 …… 139
5.2 继 承 …… 140
5.3 PostgreSQL 索引 …… 141
B-Tree 索引 …… 141
R-Tree 索引 …… 142
散列索引 …… 142
其他索引主题 …… 142
5.4 OID …… 143
5.5 多版本并发控制 …… 144
读已提交（READ COMMITTED）级 …… 145
可串行化（Serializable）级 …… 145

第三部分 PostgreSQL 管理

第 6 章 用户可执行文件 …… 149

文件列表（按字母排序） …… 149
createdb …… 149
createlang …… 150
createuser …… 150
dropdb …… 151
droplang …… 152
dropuser …… 153

ecpg …… 153
pgaccess …… 154
pgadmin …… 155
pg_dump …… 156
pg_dumpall …… 157
pg_restore …… 158
pg_upgrade …… 159
pgtclsh …… 160
pgtksh …… 161
psql …… 161
vacuumdb …… 168

第 7 章　系统可执行文件 …… 170

7.1　文件列表（按字母排序） …… 170
initdb …… 170
initlocation …… 171
ipcclean …… 171
pg_ctl …… 172
pg_passwd …… 173
postgres …… 173
postmaster …… 175

第 8 章　系统配置文件和库 …… 177

8.1　系统配置文件 …… 177
pg_options/postgresql.conf …… 177
/etc/logrotate.d/postgres …… 181
pg_hba.conf …… 181
8.2　库文件 …… 183

第 9 章　数据库和日志文件 …… 185

9.1　PostgreSQL 数据目录 …… 185
系统目录 …… 186
用户定义目录 …… 187
9.2　日志文件 …… 187
自定义日志旋转 …… 187
配置 PostgreSQL 以使用 syslog …… 188

第 10 章　常规管理任务 …… 189

10.1　编译和安装 …… 189

基本于源代码安装 …… 189
基于包的安装 …… 190
10.2 创建用户 …… 191
10.3 授予用户权限 …… 192
10.4 数据库维护 …… 192
10.5 数据库备份/恢复 …… 193
10.6 操作性能优化 …… 194
硬件考虑 …… 194
优化 SQL 代码 …… 194
缓存大小和其他因素 …… 195
通过 EXPLAIN 优化查询 …… 196

第四部分 用 PostgreSQL 编程

第 11 章 服务器端编程 …… 201

11.1 过程语言的优势 …… 201
11.2 安装过程语言 …… 202
SQL 声明 …… 202
使用 createlang 命令 …… 202
11.3 PL/pgSQL …… 203
PL/pgSQL 语言定义 …… 203
11.4 PL/Tcl …… 211
一般 Tcl 语言初步 …… 211
PL/Tcl 语言定义 …… 213
11.5 PL/Perl …… 216
一般 Perl 语言初步 …… 216
PL/Perl 语言定义 …… 220

第 12 章 创建自定义函数 …… 222

12.1 创建自定义函数 …… 222
使用示例 …… 223
12.2 创建自定义触发器 …… 226
12.3 创建自定义规则 …… 228
注释和其他方面 …… 230

第 13 章 客户端编程 …… 232

13.1 ecpg …… 232
声明和定义变量 …… 233
连接到数据库 …… 233

执行查询 …… 233
错误处理 …… 234
13.2 JDBC …… 234
编译驱动 …… 234
安装驱动 …… 235
配置客户端 …… 235
连接 …… 235
执行查询 …… 236
修改记录 …… 236
13.3 libpq …… 236
PQconnectdb …… 236
PQexec …… 238
13.4 libpq++ …… 239
PgConnection …… 239
PgDatabase …… 240
13.5 libpqeasy …… 241
13.6 ODBC …… 242
安装 …… 242
13.7 Perl …… 244
DBI 类（连接） …… 245
DBI 句柄方法（运行查询） …… 245
DBI 语句句柄方法（结果） …… 246
语句句柄属性 …… 246
13.8 Python（PyGreSQL） …… 247
编译 PyGreSQL …… 247
Python 配置 …… 247
PyGreSQL 接口 …… 247
13.9 PHP …… 252

第 14 章 高级 PostgreSQL 编程 …… 256

14.1 扩展函数 …… 257
SQL 函数 …… 257
过程语言函数 …… 258
已编译函数 …… 258
14.2 扩展类型 …… 261
创建数据类型 …… 261
14.3 扩展操作符 …… 263
定义自定义操作符 …… 263
优化注释 …… 264

第五部分　附录

附录 A　参考资源 …… 269

附录 B　PostgreSQL 版本信息 …… 277

第一部分

SQL 参考

第 1 章　PostgreSQL SQL 参考

大部分关系数据库运用结构化查询语言（SQL）来执行特定的数据库操作。在 SQL 出现之前，每种数据库系统均使用自己独有的方法来访问其中的数据。如果开发者试图创建一个兼容性强的前端系统，此系统要能与多种数据库通信，那么，这一现状将会给他们带来许多难题。

解决的方法就是制定一个被所有数据库发行商支持的访问数据库函数的标准方法。此想法提出的次年，第一次诞生了符合该思想的 SQL-86 标准。之后，此标准经过补充其他功能而加以了修订，并被命名为 SQL-89。

1992 年，SQL 标准作了重大的扩展，用以处理更多的数据类型、外部连接、目录规范及其他一些改进。这一 SQL 版本就是 SQL-92（又称 SQL-2），它是许多现代关系数据库管理系统（RDBMS）的基础。

PostgreSQL 支持 SQL-92 标准中规定的大部分功能。表 1-1 列出了 SQL 的命令、语法、选项及如何在 PostgreSQL 中使用 SQL 的例子。虽然 PostgreSQL 支持 SQL-92 规范中的所有主要功能，但也有 PostgreSQL 使用的 SQL 命令与正式 SQL-92 规范不对应的情形。下面按字母顺序排序的命令列表对这些地方作了注释并为用户指出了作用相同的命令。

1.1　命令表

表 1-1 中列出了 PostgreSQL 所支持的 SQL 命令，按所完成的任务类别进行组织。

表 1-1　　命 令 表

	创建类	删除类	使用/修改类
数据库	CREATE DATABASE	DROP DATABASE	COMMENT LOAD VACUUM
表/索引	CREATE TABLE CREATE INDEX CREATE VIEW	DROP TABLE DROP INDEX DROP VIEW TRUNCATE	COMMENT SELECT EXPLAIN ALTER TABLE ALTER INDEX COPY UPDATE INSERT DELETE SELECT INTO VACUUM CLUSTER

续表

	创建类	删除类	使用/修改类
约束	CREATE TRIGGER CREATE CONSTRAINT CREATE RULE CREATE SEQUENCE	DROP TRIGGER DROP CONSTRAINT DROP RULE DROP SEQUENCE	COMMENT
用户	CREATE USER CREATE GROUP	DROP USER DROP GROUP	ALTER USER ALTER GROUP GRANT REVOKE
对话/事务	SET DECLARE BEGIN	CLOSE ABORT COMMIT ROLLBACK END	SHOW RESET FETCH MOVE LISTEN UNLISTEN NOTIFY LOCK UNLOCK
其他	CREATE AGGREGATE CREATE OPERATOR CREATE TYPE CREATE LANGUAGE CREATE FUNCTION	DROP AGGREGATE DROP OPERATOR DROP TYPE DROP LANGUAGE DROP FUNCTION	COMMENT

1.2 命令列表(按字母顺序排列)

下面按字母顺序列出了 PostgreSQL（版本号 7.1）所支持的 SQL 命令。

Abort

语法

```
ABORT [WORK | TRANSACTION]
```

描述

Abort 命令用于终止一个进程中的事务并回滚到表的初始状态。

输入

无。WORK 和 TRANSACTION 为可选项，对结果无影响。

输出

ROLLBACK（如果成功则返回此消息）。

NOTICE: ROLLBACK: no transaction in process（若进程中无事务则返回此消息）。

注意

必须在 BEGIN...COMMIT 命令组合中使用。

SQL-92 兼容性

在 SQL-92 中不使用；可用 ROLLBACK 代替。

示例

下面代码说明了如何使用 ABORT 命令来终止一个进程中的事务并且返回到表的初始状态。首先你看到的是表 mytable 初始状态时的值。接着，通过命令 UPDATE 修改这些值。不过，由于 ABORT 命令在一个 BEGIN…COMMIT 事务中执行，所以我们可以终止(ABORT)当前事务并返回到表的初始状态。

```
SELECT * FROM mytable;
      Name     |   age
      --------------
      Barry    |   29
BEGIN TRANSACTION;
UPDATE mytable SET age=30 WHERE name='Barry';
SELECT * FROM mytable;
      Name     |   age
      --------------
      Barry    |   30
ABORT TRANSACTION;
SELECT * FROM mytable;
      Name     |   age
      --------------
      Barry    |   29
```

ALTER GROUP

用法

```
ALTER GROUP GROUPNAME [ ADD USER|DROP USER ] USERNAME [,…]
```

描述

ALTER GROUP 向一个指定的组中添加或删除用户。

输入

GROUPNAME——欲修改的组名。

USERNAME——欲添加或删除的用户名。

输出

ALTER GROUP（操作成功则返回此消息）。

注意

只有超级用户才能使用这一命令——其他所有用户试图使用此命令均无效。命令被执行前，用户和组必须已经存在。删除一个用户仅仅从组中删除了此用户，并没有从数据库中删除他。

SQL-92 兼容性

SQL-92 中无 ALTER GROUP 命令。SQL-92 中使用的是角色的概念。

示例

下面代码显示如何添加多用户以及从组 admins 中删除用户。

```
ALTER GROUP admins ADD USER frank, mike, bill;
ALTER GROUP admins DROP USER mike;
```

ALTER TABLE

用法

```
ALTER TABLE table [ * ] ADD [COLUMN] column coltype
ALTER TABLE table [ * ] ALTER [COLUMN] column
  {SET DEFAULT value | DROP DEAULT}
ALTER TABLE table [ * ] RENAME [COLUMN] column TO newcolumn
ALTER TABLE table [ * ] RENAME TO newtable
ALTER TABLE table [ * ] ADD table constraint
```

从 PostgreSQL 7.1 开始，ALTER TABLE 命令添加了许多新的功能，如下面所示：

```
ALTER TABLE [only] table [ * ] ADD [COLUMN] column coltype
ALTER TABLE [only] table [ * ]
  ALTER [COLUMN] colum {SET DEFAULT value | DROP DEAULT}
ALTER TABLE table ADD table-constraint-definition
ALTER TABLE table OWNER TO new-owner
```

描述

ALTER TABLE 用于修改一个表或列。它可以让列被修改、重命名或添加到一个现有表中。另外，通过语法 ALTERTABLE…RENAME 可以重命令表本身。如果一个表或列被重命名，其中的数据不会受到影响。

使用 SET DEFAULT 或 DROP DEFAULT 选项，可以设置、修改或删除此列的缺省值。（请参见“注意”部分）

如果在表名称后带有一个星号（*），那么，所有从当前表继承了列属性的表都将被同时修改。（请参见“注意”部分）这一点在 PostgreSQL 7.1 版本中发生了变化，缺省情况下，它级联了继承表格的所有更改。为了限制 PostgreSQL 7.1 及以后版本中指定表的级联变化，可以使用 ONLY 命令。

输入

coltype——添加列的类型。

column——修改列的名称。

constraint——添加到表中的新约束。

newcolumn——列被重命名后的新名称。

new-owner——改变表的所有者到此用户。

newtable——重命名后的新表名称。

table——将要修改的表名称。

value——特定列的缺省设置值。

输出

ALTER（若修改成功则返回此消息）。

ERROR（若列、表或列类型不存在则返回此消息）。

注意

命令 ALTER TABLE 仅由表或当前修改表的类所有者使用。

关键字[COLUMN]是可选项，可以被安全地忽略。

更改一个列的缺省值不会对列中的现有数据产生关联影响。DEFAULTVALVE 只影响刚刚插入的行。要更改所有行的缺省值，必须让 DEFAULT VALUE 子句紧跟一个 UPDATE 命令来将现有行重置为期望值。

如果表是超类，则必须包含星号（*），否则，基于刚修改列的子表的查询就会失败。

只有 FOREIGN KEY 约束可以添加到表中；要添加或删除一个特定的约束，必须创建一个特定的索引。当添加一个 FOREIGN KEY 约束时，外表中必须存在此列名。要在表中添加检查约束，你必须使用 CREATE TABLE 命令重创建和重载此表。

SQL-92 兼容性

ALTER COLUMN 形式与 SQL-92 完全兼容。

ADD COLUMN 形式也是兼容的，只是它并不支持缺省值或约束。

ALTER COLUMN 命令必须用于获取期望结果。

ALTER TABLE 并不支持 SQL-92 中定义的一些功能。

SQL-92 很明确地允许从一个表中删除约束。为了在 PostgreSQL 获得到同样的结果，必须删除索引，或重创建并且重载表。

示例

要在表 authors 中添加一个 VARCHAR[2]类型的列 statecode，你必须使用如下命令：

```
ALTER TABLE authors ADD COLUMN statecode VARCHAR[2];
```

要将列 statecode 重命名为 state，你可以使用如下命令：

```
ALTER TABLE authors RENAME COLUMN statecode TO state;
```

要将列 state 的缺省值改为 TX，可使用如下命令：

```
ALTER TABLE authors ALTER COLUMN old_email SET DEFAULT 'TX';
```

要在表 authors 中添加一个约束 FOREIGN KEY（它将保证字段 state 是一个有效条目，正如外表 us_states 所定义的一样），则可使用如下命令：

```
ALTER TABLE authors ADD CONSTRAINT statechk FOREIGN KEY (state) FREERENCES
us_states (state) MATCH FULL;
```

要将表 authors 改名为 writes，可使用如下命令：

```
ALTER TABLE authors RENAME TO writers;
```

ALTER USER

用法

```
ALTER USER username [WITH PASSWORD password]
      [CREATEDB | NOCREATEDB]
      [CREATEUSER | NOCREATEUSER]
      [VALID UNTIL abstime]
```

描述

ALTER USER 用于修改数据库中现有用户的帐号。

可选子句 CREATEDB 或 NOCREATEDB 决定是否允许用户创建数据库。

可选子句 CREATEUSER 或 NOCREATEUSER 决定是否允许用户创建自身用户。

当密码过期时，可选子句 VALID UNTIL 提供日期和（或）时间。

输入

username——即将修改属性的用户名。

password——此用户的新密码。

abstime——密码过期的日期和（或）时间。

输出

ALTER USER（若此动作成功则返回此消息）。

ERROR: ALTER USER: user 'username' does not exist（若在当前数据库中不存在此用户名，则返回此消息）。

注意

只有数据库的管理人员或超级用户才能修改权限和帐号过期日期。

要从数据库中创建或删除一个用户，可分别使用 CREATE USER 或 DROP USER 命令。

SQL-92 兼容性

SQL-92 并没有定义 USERS 的概念，而是留给每个具体的数据库操作来实现和决定。

示例

要将用户 Charles 的密码改为 querty，可使用如下命令：

```
ALTER USER 'Charles' WITH PASSWORD 'querty';
```

要将用户 Charles 的密码过期日期设为 2005 年 1 月 1 日，可以使用如下命令：

```
ALTER USER 'Charles' VALID UNTIL 'Jan 1 2005';
```

要使用户 Charles 的密码于 2005 年 1 月 1 日 12:35 时过期（时区时间早于 UTC 6 个小时），可使用如下命令：

```
ALTER USER 'Charles' VALID UNTIL 'Jan 1 12:35:00 2005 +6';
```

要授予用户 Charles 创建自身用户但非自身数据库的权限，可使用如下命令：

```
ALTER USER 'Charles' CREATEUSER NOCREATEDB;
```

BEGIN

用法

```
BEGIN [ WORK | TRANSCTION ]
```

描述

缺省情况下，PostgreSQL 中的所有命令都是在隐含事务中执行。BEGIN...COMMIT 子句的显式用法中包含了一系列的 SQL 命令以确保正确执行。如果其中的一个命令失败，它将导致整个事务回滚（ROLLBACK），返回到数据库的原始状态。

PostgreSQL 事务隔离级别通常被设置成 READ COMMITTED，这意味着进程中的事务可能受到其他提交事务的影响。事务执行后，通过执行 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE 命令可改变此行为。它将阻止当前事务受到进程中数据库的变化影响。（请参见“示例”部分。）

输入

无必须输入的参数。WORK 和 TRANSACTION 是无影响的可选项。

输出

BEGIN（一旦事务系列开始，则返回此消息）。

NOTICE: BEGIN: already a transaction in progress（此消息表明当前事务已经开始执行，而且此事务对现有事务无影响）。

注意

参见 ABORT、COMMIT 和 ROLLBACK 获取与事务有关的更多信息。

SQL-92 兼容性

关键字 BEGIN 在 SQL-92 中是隐含的；它是 PostgreSQL 的扩展。通常，在 SQL-92 中，每个事务均带一个隐含的 BEGIN 命令开始，但它需要一个 COMMIT 或 ROLLBACK 命令来实际提交此事务给数据库。

示例

在以下例子中，请着重注意并发操纵数据库的两个用户。这些例子突出显示了事务是如何影响其他用户数据的，尤其是使用命令 READ COMMIT 和 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE 后的情形。（注意：在下列例子中，SELECT 命令也显示列名称以及返回真实数据。为了让代码清单更具可读性，这一输出结果被简化了。）

下例中，显示了用户 user 1 正从事显式事务系列，用户 user 2 正使用隐含事务。此例子显示了每个用户看到的结果。

```
User1                                User2

BEGIN TRANSACTION;
INSERT INTO mytable VALUES ('Pam');
(1) row inserted                     SELECT * FROM mytable;
                                     (0) row found
SELECT * FROM mytable;
(1) row found
COMMIT TRANSACTION;
SELECT * FROM mytable;               SELECT * FROM mytable;
(1) row found                        (1) row found
```

下例显示了两个均使用了 READ COMMIT（缺省情况）的显式事务是如何相互影响的。请特别注意，user 1 执行 COMMIT 命令后，user 2 可以看到其产生的影响。还可将此例与后面的例进行对比，下例使用的是 SERIALIZABLE 命令。

```
User1                                User2
BEGIN TRANSACTION;                   BEGIN TRANSACTION;
SELECT * FROM mytable;               SELECT * FROM mytable;
(0) results found                    (0) results found

INSERT INTO mytable VALUES ('Pam');
(1) results inserted
COMMIT;                              SELECT * FROM mytable;
                                     (1) results inserted
                                     INSERT INTO mytable VALUES('Barry');
                                     (1) results inserted

SELECT * FROM mytable;
(1) results inserted                 COMMIT;

SELECT * FROM mytable;               SELECT * FROM mytable;
```

```
(2) results found                    (2) results found
```

下例中使用了 SERIALIZABLE 命令来显示当事务进行时如何阻止其变化被其他用户看到。实际上，在事务开始前，SERIALIZABLE 命令生成了现有数据库的一个快照，并且将之与其他 COMMIT 命令产生的影响隔绝开。

```
User1                                User2

BEGIN TRANSACTION;                   BEGIN TRANSACTION;
                                     SET TRANSACTION ISOLATION LEVEL
                                     SERIALIZABLE;
SELECT * FROM mytable;               SELECT * FROM mytable;
(0) results found                    (0) results found

INSERT INTO mytable VALUES {'Pam'};
(1) results inserted
COMMIT;                              SELECT * FROM mytable;
                                     (0) results found
                                     INSERT INTO mytable VALUES {'Barry'};
                                     (1) results inserted
SELECT * FROM mytable;
(1) results found
COMMIT;

SELECT * FROM mytable;               SELECT * FROM mytable;
(2) results found                    (2) results found
```

CLOSE

用法

```
CLOSE cursor;
```

描述

此命令将关闭使用 DECLARE 命令打开的游标。关闭一个游标可以释放 PostgreSQL 资源，它只能在此游标不再有用时才可以执行。

输入

cursor——将要关闭的游标名称。

输出

CLOSE（若命令成功执行，则返回此消息）。

NOTICE PerformPortalClose: portal "cursor" not found（若没有发现游标，则返回此消息）。

注意

缺省情况下，如果执行一个 COMMIT 或 ROLLBACK 命令，则关闭一个游标。请参见 DECLARE 了解更多的游标讨论信息。

SQL-92 兼容性

CLOSE 完全兼容于 SQL-92。

示例

下面代码将关闭游标 newchecks：

```
CLOSE newchecks;
```

CLUSTER

用法

```
CLUSTER index ON table;
```

描述

通常，PostgreSQL 从物理上按无序方式保存表中的数据。CLUSTER 将强迫 PostgreSQL 从物理上重新排序表，让数据根据指定索引进行组织。一般而言，当执行 CLUSTER 命令后数据库的操作性能将会得到提高。不过，后续的一些插入并不是按相同方式进行物理组织的。实际上，CLUSTER 命令基于指定的标准创建一个静态索引。如果后续数据被插入或更改，CLUSTER 命令必须重新执行从物理上重新排序此表。

输入

index——操纵集簇所用的索引名称。

table——表名称。

输出

CLUSTER（当此命令成功执行则返回此消息）

ERROR: relation <tablerelation_number> inherits "table"

ERROR: relation "table" does not exist

注意

为了执行数据的重新排序，PostgreSQL 将按索引顺序复制表到一个临时表中并按新顺序重创建和重载此表。在这一转换中将导致许多授予的权限和其他索引丢失。

因为 CLUSTER 命令产生了一个静态顺序，所以大部分用户只在特定情况下才能用到此命令。通过 SELECT 命令中的 ORDER BY 子句可以创建一个动态集簇。（请参见本章后面的 SELECT 命令部分）

完成 CLUSTER 命令要花费几分钟时间。这要取决于表的大小和（或）系统的硬件速度。

SQL-92 兼容性

SQL-92 中没有 CLUSTER 命令。

示例

下面例子显示了一个名称为 authors、拥有一个索引 name 的表。使用 SELECT…ORDER ON name 命令可以达到同样的效果。

```
SELECT * FROM authors;
      name   | Age
      ------------
      Pam    | 25
      Barry  | 29
      Tom    | 32
      Amy    | 43
CLUSTER authors ON name;
```

```
name     | Age
------------
Amy      | 43
Barry    | 29
Pam      | 25
Tom      | 32
```

COMMENT

用法

```
COMMENT ON DATABASE | INDEX | RULE | SEQUENCE | TABLE | TYPE | VIEW | VIEW
    obj_name IS text
```

或

```
COMMENT ON COLUMN table_name.column_name IS tex |
    AGGREGATE agg-name agg_type IS text |
    FUNCTION func_name(arg1,arg2,[,...]) IS text | OPERATOR op_name
    (leftoperand_type, rightoperand_type) IS text |
    TRIGGER trigger_name ON table_name IS text
```

描述

COMMENT 允许用户或管理员为对象添加文本描述。

输入

obj_name——添加注释的对象名称。

table_name——受影响的表名称。

column_name——受影响的特殊列名称。

agg-name——被注释的聚集名称。

agg_type——受影响的聚集类型。

func_name——函数名称。

op_name——操作符名称。

trigger_name——触发器名称。

text——实际添加到对象上的注释文本。

输出

COMMENT（若命令成功执行则返回此消息）

注意

COMMENT 命令对对象无影响；它只是基于文档的目的才被使用。

对象的注释可以通过/dd 命令中的 psql 获取（请参考第 6 章中“psql”、“用户执行文件”一节了解更多信息）。

SQL-92 兼容性

SQL-92 中没有 COMMENT 命令；它是 PostgreSQL 的扩展。

示例

下面将为表 authors 添加注释：

```
COMMENT ON TABLE authors IS 'Listing of our authors';
```

下面再列举其他一些使用 COMMENT 命令的例子：

```
COMMENT ON DATABASE newriders IS 'Database for web·site';
COMMENT ON COLUMN authors.email IS 'Email address for author';
COMMENT ON FUNCTION book_sales(varchar) IS 'Return books sold';
COMMENT ON OPERATOR > (int,int) IS 'Compares two integers';
```

COMMIT

用法

```
COMMIT [ WORK | TRANSACTION]
```

描述

缺省情况下，PostgreSQL 中的所有命令都是在隐含事务中执行的。BEGIN…COMMIT 子句的显式用法包含了一系列 SQL 命令来确保执行正确。如果其中的任一个命令执行失败，它将导致整个事务回滚（ROLLBACK），数据库返回到初始状态。

输入

无。WORK 和 TRANSACTION 是无影响的可选项。

输出

COMMIT（若成功则返回此消息）

NOTICE: COMMIT: no transaction in progress（若无此当前事务则返回此消息）。

注意

请参见 ABORT、BEGIN、COMMIT 和 ROLLBACK 命令获取与事务相关的更多信息。

SQL-92 兼容性

SQL-92 仅定义了 COMMIT 和 COMMIT WORK 形式的命令。不过，它与此命令完全兼容。

示例

此例显示了两个用户并发使用表 mytable 的情形。user 1 执行的 INSERT 命令不能被 user 2 所见，直到执行 COMMIT 命令后才可以看到。（这是在设置了 READ OMMITTED 子句的情况；请参见 BEGIN 命令部分获取更多信息）。

```
User1                                User2
BEGIN TRANSACTION;
INSERT INTO mytable VALUES('Pam');
(1) row inserted                         SELECT * FROM mytable;
                                         (0) row found
SELECT * FROM mytable;
(1) row found
COMMIT TRANSACTION;

SELECT * FROM mytable;                   SELECT * FROM mytable;
(1) row found                            (1) row found
```

COPY

用法

```
COPY [BINARY] table [WITH OIDS] FROM {filename | stdin} [ [USING] DELIMTERS
  delimiter] [WITH NULL AS nullstring]
```

或

```
COPY [BINARY] table [WITH OIDS] TO {filename | stdout} [ [USING] DELIMTERS
  delimiter] [WITH NULL AS nullstring]
```

描述

COPY 命令允许用户从 PostgreSQL 导入或导出表。通过使用 BINARY 关键字，数据以二进制格式来使用，它不能直观阅读。对于 ASCII 格式，通过包含 USING DELIMITERS 关键字来定义分隔符。另外，通过使用 WITH NULL 子句可以定义空字符串。包含 WITH OIDS 子句将导致 PostgreSQL 导出或者期望对象 ID 存在。

（1）文本文件结构

当不带关键字使用 COPY…TO 时，PostgreSQL 将产生一个文本文件，在此文件中，每一行（实例）包含子文件的一个单独行中。如果嵌入在字段中的某个字符与指定的分隔符相匹配，则此嵌入字符前就会冠以一个反斜杠（\）。如果包含 OID，则成为此行的第一项。产生的文本文件格式如下所示：

```
<OID.Row1><delimiter><Field1.Row1><delimiter>…<Fild N.Row1><newline>
<OID.Row2><delimiter><Field1.Row2><delimiter>…<Fild N.Row2><newline>
-
<OID.RowN><delimiter><Field1.RowN><delimiter>…<Fild N.RowN><newline>
(EOF)
```

如果 COPY…TO 被发送到标准输出端（stdout）而不是一个文本文件，那么，文件结尾（EOF）将通过\.<newline>（反斜杠加一个句号再跟一个新行）指定。

使用 COPY…FROM 时，期望文本文件应该具有相同的格式。同样，如果从标准输入端（stdin）开始复制，最后一行应该是\.<newline>（反斜杠加一个句号再跟一个新行）。然而，通过 COPY…FROM 导入文件时，若遇到一个\.<newline>或发生一个<EOF>时进程将终止。

（2）二进制文件结构

如果 COPY…TO 与 BINARY 子句一起使用，PostgreSQL 将按二进制类型产生结果文件。二进制格式文件如下所示：

数据类型	描述
Uint32	文件中元组实例（）的总数量。
Uint32	元组数据的总长度。
Uint32	OID（若被定义了的话）。
Uint32	NULL 属性的数量。
[Uint32,…Uint32]	属性、从 0 开始计数以及元组数据。

Uint32 是一个无符号的 32 位整数。

输入

table——导入或导出表的名称。

filename——导入或导出的文件名。

stdin——指定文件必须来源于标准输入端或管道。

stdout——指定文件必须复制到准输出端或管道。

delimiter——用于分隔字段的单字符分隔符

nullstring——用于标明 NULL 值的字符串。缺省情况下为\N（反斜杠和一个 N）。

输出

COPY（若此命令成功执行则返回此消息）。

ERROR: reason（若命令执行失败，则返回此消息并指出失败原因）。

注意

执行 COPY…TO 或 COPY…FROM 命令的用户必须具有执行 SELECT 或 SELECT INTO 的权限。

缺省情况下，分隔符是 Tab（\t）。如果通过 USING DELINITER 选项指定的分隔符长于一个字符，那么只使用其第一个字符。

当给定一个文件名时，PostgreSQL 会假定它位于当前目录（如$PGDATA）。通常，最好使用文件的完整路径，这样就不会发生混淆。因而，执行 COPY 命令的用户必须具有足够的权限来创建修改或删除指定目录中的文件。当然，这更取决于操作系统（OS）而不是 PostgreSQL 授予用户此类权限。

使用 COPY 命令并不调用表规则或缺省值。不过，触发器仍将继续发挥作用。所以，执行 COPY 命令后，需要进行额外的操作（如替换缺省值）。

一般，使用 BINARY 关键字将导致运行加速。不过这取决于保存在表中的数据。

SQL-92 兼容性

在 SQL-92 中没有具体指明 COPY 命令。它留给每个具体的实现过程来决定如何导入和导出数据。

示例

下例从表 authors 中导出数据到文件/home/sqldata.txt 中，并用逗号作为分隔符：

```
COPY authors TO '/home/sqldata.txt' USING DELIMITERS',';
```

产生的文件包含下面条目：

```
Amy,43
Barry,29
Pam,25
Tom,32
```

如果添加了 OID 子句，则命令和产生的文件如下：

```
COPY authors WITH OIDS TO '/home/sqldata.txt'
   USING DELIMITERS',';
         15001,Amy,43
         15002,Barry,29
         15003,am,25
         15004,Tom,32
```

另一种是添加了关键字 BINARY 的情况，语句将如下所示：

```
COPY BINARY authors TO '/home/sqldata.txt' USING DELIMITERS',';
```

如果你用 unix od c 命令查看数据，则结果如下：

```
004  \0  \0  \0  \f  \0  \0  \0   \0  \0  \0  \0   \a  \0  \0  \0
 A   m   y  \0   +  \0  \0  \0  020  \0  \0  \0   \0  \0  \0  \0
\t  \0  \0  \0   B   a   r   r    y  \0  \0  \0  035  \0  \0  \0
\f  \0  \0  \0  \0  \0  \0  \0   \a  \0  \0  \0    P   a   m  \0
031  \0  \0  \0  \f  \0  \0  \0   \0  \0  \0  \0   \a  \0  \0  \0
120   T   o   m  \0      \0  \0   \0
```

CREATE AGGREGATE

用法

```
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

PostgreSQL 包含大量内置聚集函数，如 sum( )、avg( )、min( )、max( )……通过使用 CREATE AGGREGATE 命令，用户可以扩展 PostgreSQL 到包含用户定义的聚集函数。

一个聚集至少包括一个函数，但最多只能达到 3 个。两个状态转换函数 sfunc1 和 sfunc2 以及一个最终计算函数 ffunc。它们可以按以下方式使用：

```
sfunc1(internal state1,next-data-item)→next-internal-state-1
sfunc2(internal state2)→next-internal-state2
ffunc(internal state1, internal state2) →aggregate-value
```

next-internal-state-1 和 next-internal-state-2 是由 PostgreSQL 创建的临时变量，它们表示计算时聚集的当前内部状态。（这些变量分别为 state1_type 和 state1_type 类型）通过相关函数计算完所有数据后，调用最终函数来计算聚集的输出值。

除此之外，一个聚集函数可以提供一个或两个相关函数的初始值。如果只有一个 sfunc 被使用，此初始值就是可选项。然而，如果指定了 sfunc2，那么必须强制包含 initial_condition2。

输入

name——创建新聚集名称。

input_data_type——聚集运算的数据类型（如 INT、VARCHAR 等等）。

sfunc1——第一个状态转换函数，用于运算所有非 NULL 值（请参见下面的“注意”部分）。

state1_type——第一个状态转换函数的数据类型（即 INT、VARCHAR 等等）。

sfunc2——第二个状态转换函数，用于运算所有非 NULL 值（请参见下面的“注意”部分）。

state2_type——第二个状态转换函数的数据类型（即 INT、VARCHAR 等等）。

ffunc——所有输入完成后计算聚集的最终函数（参见下面的“注意”部分）。

initial_condition1——state_value1 的初始值。

initial_condition2——state_value2 的初始值。

输出

CREATE（若成功执行则返回此消息）。

注意

如果包含两个 sfunc 函数则必须包括 ffunc 函数。如果只使用了一个状态转换函数，则它是可选项。当不包含 ffunc 时，聚集的输出值来自于 sfunc1 计算所得的最后值。

如果两个聚集的运算数据类型不同，那么它们可以有相同的名称。如此一来，PostgreSQL 就会允许使用一个聚集名称，不过，要根据给定的数据类型选择适当的版本。换言之，如果有两个函数，它们的名称相同，且每个所接收的数据类型不同，如 foo（[varchar]）和 foo（[int]），那么，你只需要调用聚集 foo（[our-date-type]），PostgreSQL 就会选择适当的版本来计算输出聚集值。

SQL-92 兼容性

此命令为 PostgreSQL 的扩展；在 SQL-92 中没有与之等价的概念。

示例

下面代码创建了一个名为 complex_sum 的函数，它通过添加复杂的数字支持功能而扩展了标准 sum()函数。

```
CREATE AGGREGATE complex_sum(sfunc=complex_add,
basetype=complex,stype=complex, initcond='(0,0)')
```

然后，运行此代码，可以看到如下输出结果：

```
SELECT complex_sum(salary) FROM authors;

      complex)sum
      ------------
      (34,53,9)
```

CREATE DATABASE

用法

```
CREATE DATABASE name [WITH LOCATION='path']
```

描述

CREATE DATABASE 用于创建新 PostgreSQL 数据库。创建用户必须具有足够的执行此操作的权限。相应地，一旦创建了数据库，创建者就成为此数据库的所有者。

缺省情况下，PostgreSQL 将在标准数据目录（即$PGDATA）中创建数据库。不过，通过包含关键字 WITH LOCATION，PostgreSQL 也可以识别和使用其他路径。

输入

name——将要创建数据库的名称。

path——将要创建数据库文件的路径和（或）文件名。

输出

CREATE DATABASE（若命令成功执行，则返回此消息）。

ERROR: user 'username' not allowed to create/drop database（若用户无创建或删除数据库的权限，则返回此消息）。

ERROR: ceatedb: database 'name' already exist（若此数据库已经存在，则返回此消息）。

ERROR: Single quotes are not allowed in database name（若数据库名称中包含单引号，则返回此消息）。

ERROR: Single quotes are not allowed in database path（若路径名中包含单引号，则返回此消息）。

ERROR: The path 'pathname' is invalid（若路径并不存在，则返回此消息）。

ERROR: createdb: May not be called in transaction block（若试图在一个显式事务中创建数据库，则返回此消息）。

ERROR: Unable to create database directory 'path'

或

ERROR: Could not initialize database directory（通常是由于用户没有指定目录的足够权限而返回此消息）。

注意

如果在路径定义中包含一个斜杠（/），则前导部分应该是一个能被服务器进程识别的环境变量。但如果在 PostgreSQL 编译时，将选项 ALLOW_ABSOLUTE_PATHS 设置成真，则允许使用绝对路径名（例如：/home/barry/pqsql）。缺省情况下，此选项设置成假。

在使用其他路径前，必须准备用 initlocation 命令。欲了解更详细信息，请参见第 7 章“系统可执行文件”的“initlocation”命令部分。

SQL-92 兼容性

数据库等价于 SQL-92 中的目录概念，具体在事务执行时定义。

示例

下面是一个新创建的数据库 sale 的简单例子：

```
CREATE DATABASE sales;
```

下例基于服务器所识别环境变量的基础上，在其他可选路径创建一个数据库：

```
CREATE DATABASE sales WITH LOCATION='PGDATA2/sales';
```

CREATE FUNCTION

用法

```
CREATE FUNCTION name ([ftype [,...]])
RETURN rtype AS definition
LANGUAGE lang_name[WITH (attrib [,...])]
```

或

```
CREATE FUNCTION name ([ftype [,...]])
RETURN rtype AS obj, link_symbol
LANGUAGE 'C' [WITH (attrib [,...])]
```

描述

用户可以通过 CREATE FUNCTION 命令在 PostgreSQL 中创建函数。PostgreSQL 允许操作符重载概念，也就是说，只要每个操作符处理的数据类型不同，几个不同的函数可以使用同一个名称的函数。但必须与 C 中的名称空间区别开来。请参考第 12 章“创建自定义函数”了解更详细信息。

输入

name——创建函数的名称。

ftype——函数参数所需的数据类型。

rtype——函数返回的数据类型。

definition——定义函数的任一个真实代码、一个函数名，或对象文件的路径。

obj——当使用 C 代码时，指定义函数的真实对象文件。

link_symbol——适当时候用于定义对象链接符号。

lang_name——所使用的语言名称。

attrib——用于优化目的的可选信息（参见“注意”部分了解更多信息）。

输出

CREATE（若成功执行则返回此消息）。

注意

创建函数的用户将成为此函数的必然所有者。

使用 DROP FUNCTION 命令来删除 PostgreSQL 中的用户定义函数。

SQL-92 兼容性

CREATE FUNCTION 命令是 PostgreSQL 语言的扩展。

示例

下面代码创建了一个简单的 SQL 函数，函数返回某职员最后一次检查的日期。首先，定义此函数：

```
CREATE FUNCTION last_check(varchar)
RETURN datatime AS
'BEGIN;
SELECT max(check_date) FROM authors WHERE authors.name=$1;
END;
LANGUAGE 'sql';
```

下面通过传递一些数据来进行测试：

```
SELECT last_check('Pam') AS CHECK_DATE;

CHECK_DATE
----------
11/14/2001
```

CREATE GROUP

用法

```
CREATE GROUP name [WITH SYSID gid][USER username[,…]]
```

描述

CREATE GROUP 用于在当前数据库中初始化一个新组。而且，通过指定 USER 关键字可以在新创建的组中添加用户。缺省时，此创建的组赋予下一个组 ID（gid）；但如果指定了 WITH SYSID 子句，用户可以声明所用 gid（如果可用的话）。

输入

name——创建新组的名称。

gid——如果指定的话，将为组分配 ID。

username——如果指定的话，将此用户添加到新创建组中。

输出

CREATE GROUP（若成功执行，则返回此消息）。

注意

如果指定了用户名，则此用户名在使用前必须已经存在。

执行此命令的用户必须具有访问数据库的超级用户权限。

SQL-92 兼容性

在 SQL-92 中无 GROUPS；不过，ROLES 概念与之类似。

示例

创建一个名为 book_authors 的新组：

```
CREATE GROUP book_authors;
```

创建一个新组并指定其用户：

```
CREATE GROUP book_authors WITH USER barry,pam,tom;
```

CREATE INDEX

用法

```
CREATE [UNIQUE]INDEX indexname ON tablename
      [USING idx_method](columnname [oprname][,…])
```

或

```
CREATE [UNIQUE]INDEX indexname ON tablename
      [USING idx_method](funcname [columnname][,…]
      [opr_name][,…])
```

描述

此命令为特定列和指定表创建一个索引。一般，如受影响的列是查询操作的一部分的话，此操作可以提高数据库的执行效率。

除了为指定列创建索引外，PostgreSQL 还允许对函数生成的结果创建索引。它允许为数据创建动态索引，它需要通过标准运算进行重要的转换工作。

缺省时，PostgreSQL 通过 BTREE 方法创建索引。不过，除了使用 USING idx_method 子句外，它还可以指定其他方法。下面的索引方法都是可行的（参见“注意”部分了解详细信息）：

- BTREE　实现 Lehman-Yao 高并发 B-Tree 方法。

■ RTREE　Guttman 的二分算法 R-Tree 方法。

■ HASH　Litwin 的线性散列方法。

除了可以指定索引方法外，PostgreSQL 还允许使用具体指明的操作符类。一般而言，对于字段数据类型，基本的操作符类足以够用；但也有需要进行如此定义的场合。例如，需要基于绝对值和真实值对复杂数字创建索引时，就可考虑在创建索引时定义一个特殊的操作符类来获得一个最有效的索引方法。

输入

UNIQUE——此附加关键字要求指定列中的数据值具有唯一性。如果以后插入的数据不具唯一性，则会产生错误信息。

indexname——创建索引的名称。

tablename——包含此索引的表名。

idx_method——所用的索引方法：BTREE（缺省）、RTREE 或 HASH。

columnname——创建索引的指定列。

funcname——索引相关函数的结果。

oprname——执行索引时所指定的操作符类。

输出

CREATE：（若执行成功则返回此消息）。

ERROR: Cannot create index: 'index_name' already exists.（若索引已经存在则返回此消息）。

注意

BTREE 方法是最常用的（也是缺省情况下的）索引类型。而且 BTREE 是唯一支持多列索引的方法（缺省时，最多支持 16 个）。当使用如下操作符之一进行数据搜索时，BTREE 索引优先使用：

< → 小于
<= → 小于或等于
= → 等于
>= → 大于或等于
> → 大于

判断几何关系时，RTREE 方法最有用。特别是，使用下面操作符时，RTREE 索引方法优先使用：

<< → 对象位于其左侧
&< → 对象与左侧交迭
&> → 对象与右侧交迭
>> → 对象位于其右侧
&& → 对象交迭
@ → 对象包含或位于其上
~= → 相同

HASH 提供了一个非常快速的对比方法，但仅用于使用了下面操作符的场合：

= → 等于

00989716

SQL-92 兼容性

CREATE INDEX 是一个 PostgreSQL 语言扩展命令。SQL-92 并无此命令。

示例

对表 authors 中的列 lastname 创建一个索引：

```
CREATE INDEX name_idx ON authors(lastname);
```

对表 payroll 中的列 check_num 创建一个独特的索引：

```
CREATE INDEX check_num_idx ON payroll(check_num);
```

CREATE LANGUAGE

用法

```
CREATE [TRUSTED] PROCEDURAL LANGUAGE 'lang-name'
HANDLER handler-name
LANCOMPLILER 'comment'
```

描述

为 PostgreSQL 添加新语言是其较高级功能之一。管理员通过 CREATE LANGUAGE 命令可以在 PostgreSQL 中为一个新语言编目，使用它可以创建函数。

须谨慎使用关键字 TRUSTED。包含此关键字表示此特殊语言不提供非特权用户越过访问限制的权限。若不包含关键字 TRUSTED，则表示只有超级用户才能使用此语言来创建新函数。

要了解在 PostgreSQL 注册新语言的更多信息，请参见第四部分“用 PostgreSQL 编程”。

输入

TRUSTED——表示此语言能否被非特权用户信任的关键字。

lang-name——添加到系统的新语言名。新语言名不能凌驾于 PostgreSQL 内置语言之上。

HANDLER *handler-name*——被调用执行新注册语言的现有函数名称。

comment——在这里，注释不起任何实质性作用，只是一个可选项。

输出

CREATE（若命令成功执行，则返回此消息）。

ERROR PL handler func() doesn't exist（若句柄函数没有注册则返回此消息）。

注意

句柄函数不带可选参数并返回一个 opaque 类型、一个占位符或一个非定义数据类型。这就消除了在一个查询中作为标准函数调用句柄函数的可能性。

然而，在目标语言的 PL 函数中实际调用时，还是必须指定可选参数。尤其是下面几个参数必须包含：

- **触发器**当从一个触发管理器调用时，唯一的必需选项就是从程序 pg_proc 获得的对象 ID。
- **函数**当从函数管理器中调用时，必需的参数有：
- 从 pg_proc 获得的对象 ID。

- 传递给 PL 函数的参数数目。
- 以 FmgrValues 结构给出的实际参数。
- 一个表示调用体是否返回 SQL NULL 值的布尔指针。

SQL-92 兼容性

在 SQL-92 中无 CREATE LANGUAGE 语句。它是 PostgreSQL 的一个扩展命令。

示例

下面例子隐含着句柄函数 pl_call_hand 已经存在。首先，你需要将 pl_call_hand 注册为一个函数，然后，才能用于定义一个新语言：

```
CREATE FUNCTION pl_call_hand() RETURNS opaque
AS '/usr/local/pgsql/lib/my_pl_handler.so'
LANGUAGE 'C';

CREATE PROCEDURAL_ LANGUAGE 'my_pl_lang'
HANDLER pl_call_hand
LANCOMPILER 'PL/Sample';
```

CREATE OPERATOR

用法

```
CREATE OPERATOR name(PROCEDURE=function_name
      [,LEFTARG=type1]
      [,RIGHTARG=type2]
      [,COMMUTATOR=comut_op]
      [,NEGATOR=negat_op]
      [,RESTRIC=rest_func]
      [,JOIN=join_func]
      [,HASHES]
      [,SORT1=l_sortop]
      [,SORT2=r_sortop]
```

描述

此命令将从下面可能的符号中命名一个新操作符：

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

对于操作符的命名方式，请注意以下几种不允许情况：

- 双减号（--）或一个/*不能出现在操作符名称的任何地方（这些符号表示一个注释，所以被忽略了）。
- 一个美元符号（$）或一个冒号（:）不能定义为一个单字符名称。不过，允许将它作为一个多字符名称的一部分（如$%）。
- 除非满足一定的条件，一个多字符名称不能以加号（+）或减号（-）结尾。这与 PostgreSQL 解析查询操作符的方式有关。如果操作符以加号或减号结尾，则必须存在下面字符：

```
: $ ~ ! ? ' & | @ # % ^
```

除了命名约定的限制外，还必须定义右边和（或）左边的数据类型。对于一元操作符，

必须定义 LEFTARG 或 RIGHTARG 数据类型。而对于二元操作符两者都必须定义。二元操作符在操作符的每一边都有一个数据类型（即 x操作符 y），而一元操作符只在一边包含数据。

除了进程项目外，CREATE OPERATOR 命令还需要 PROCEDURE。function_name 用于指定一个先前创建的函数，它用来处理传递正确答案所必需的基本工作。

余下的选项（COMMUTATOT、NEGATOR、RESTRICT、JOIN、HASHES、SORT1 和 SORT2）用于帮助查询优化进程。一般不必定义优化协助参数。有时会花费更大的时间开销来完成一个查询。在定义这些选项时必须注意，错误地使用这些优化参数会导致核心倾倒和（或）其他服务器崩溃。

请参阅本书第四部分了解创建操作符的更多信息。

输入

name——创建操作符的名称（参见前面的命名约定）。

function_name——处理操作符应用的函数。

type1——左边参数的数据类型（若存在的话）。

type2——右边参数的数据类型（若存在的话）。

comut_op——左边和右边数据布局交换等价操作符。

megat_op——对当前操作符取否的操作符（如!=对=取否）。

rest_func——当此操作符为 WHERE 子句的一部分，在计算总行数时，此函数用于估计选择性限制（请参见第四部分了解更多知识）。

join_func——如果操作符与字段在一对表中一并使用时，此函数用来估计联合选择性。

HASHES——表示 PostgreSQL 允许基于此操作符对联合使用哈希级别的等同性匹配。

l_sort_op——定义优化融合联结所需的左边排序操作符。

r_sort_op——定义优化融合联结所需的右边排序操作符。

输出

CREATE（若命令执行成功则返回此消息）。

注意

在定义操作符前函数 function_name 必须已经存在。同样，要设置 rest_func 和 join_func 相关选项，此函数必须已经存在。

如果要定义 SORT1 和 SORT2 其中之一的话，必须要同时定义两者。

RESRICT、JOINT 和 HASHES 子句仅能用于返回布尔值的二元操作符中。

SQL-92 兼容性

在 SQL-92 中没有 CREATE OPERATOR 语法，它是 PostgreSQL 的扩展命令。

示例

下面例子显示创建一个用于比较 int4 数据类型的二元操作符=（注意：此操作符已经定义为 PostgreSQL 基本操作符了，此例只是用于演示目的）。

```
CREATE OPERATOR=(PROCEDURE=int4_equalproc,
        LEFTARG=int4,
        RIGHTARG=int4,
        COMMUTATOR==,
```

```
        NEGATOR=!=,
        RESTRICT=int4_restrict_proc,
        JOIN=int4_join_proc,
        HASHES,
        SORT1=<,
        SORT2=<);
```

CREATE RULE

用法

```
CREATE RULE rulename AS ON event
      TO object
      [WHERE condition]
      DO [INSTEAD][action | NOTHING]
```

描述

用户通过 PostgreSQL 来定义指定事件的自动执行行为规则。虽然 RULES 的概念与 TRIGGERS 相近，但两者针对不同的任务存在重要的区别。

RULES 主要用于操作事件级联链来确保特定的 SQL 动作正常执行。执行动作提交前或提交后的数据验证，TRIGGERS 命令更有用。不过，在一定情况下，两者在功能上一致。

能被用于触发规则的事件有 SELECT、UPDATE、INSERT 及 DELETE。以上事件可限用于指定列或整个表。

规则创建中有一个奇怪的方面，这就是 DO INSTEAD 关键字的使用。一般而言，除了最先引发触发器的事件外，在规则定义中指定的动作将被执行。但是，包含关键字 DO INSTEAD 后，PostgreSQL 将直接执行一个其他动作而排除最先触发事件的动作。另外，如果包含 NOTHING 关键字，根本不执行任何动作。

输入

rulename——创建规则的名称。

event——引发此动作的指定事件。必须是 SELECT、UPDATE、INSERT 或 DELETE。

object——捆绑于规则上的列或表。

condition——满足 WHERE 子句的条件。

action——执行期望动作的 SQL 语句。

输出

CREATE（若命令成功执行，则返回此消息）。

注意

当指定规则的条件时，允许使用新（new）或旧（old）临时变量来执行动态查询（参见“示例”部分了解详情）。

当指定级联规则时须注意，你可以通过定义基于循环定义的多规则动作来创建无限循环。在这种情况下，如果 PostgreSQL 认为它将导致无限循环，就会简单地拒绝执行。

对一个表或列定义规则时，你必须具有定义规则的权限。

一个 SQL 规则不能指向一个数组而且不能越过参数。

规则定义中一般不能引用系统属性（如 func(cls)，其中的 cls 是一个类）。然而，

OID 能从规则中访问。

SQL-92 兼容性

CREATE RULE 是 PostgreSQL 的扩展命令，SQL-92 中无此命令。

示例

下例显示了如何应用规则来强制引用完整性。在此例中，如果从表 authors 中删除一个作者，在表 payroll 中就会将此作者的状态标记为'inactive'：

```
CREATE RULE del_author AS
      ON DELETE authors
      DO UPDATE payroll SET status='Inactive'
      WHERE payroll.auth_id=OLD.oid;
```

下一个例子显示了通过如何使用 DO INSTEAD 子句转向用户动作的。在此例中，如果用户不是一个管理者，则无执行动作（注意 current_user 的使用，它是一个包含当前用户的内置环境变量）：

```
CREATE RULE upd_payroll AS
      ON UPDATE payroll
      WHERE current_user<>"Manager"
      DO INSTEAD NOTHING;
```

从下面的例子，你可以看到如何使用规则帮助管理者跟踪能改变整个数据库的导入信息的。此规则将用一个单独的表登记所有高额定购者，管理者就可以通过日常打印和整理来了解概况：

```
CREATE RULE log_highdlr AS
      ON INSERT orders
      WHERE new.invoice_total>1000
      DO
      INSERT INTO rep_table(amt,date,description)
      VALUES
      (new.invoice_total,new.invoice_date,'Big $$
      Invoice');
```

CREATE SEQUENCE

用法

```
CREATE SEQUENCE name
[INCREMENT invalue]
[MINVALUE mnvalue]
[MAXVALUE mxvalue]
[START stvalue]
[CACHE cavalue]
[CYCLE]
```

描述

序列是数字生成器，PostgreSQL 用它来生成一系列应用于整个数据库的连续数字。最通常的情况是，用 CREATE SEQUENCE 命令生成系列特定数字用于插入表中。不过，序列

的使用可以基于许多不同原因，并且与任何表相关函数无关。

创建一个序列后，它将对应于下面函数的调用：

- nextval（sequence）——增加序列并返回下一个数字。
- currval（sequence）——返回序列的当前值（对现有序列无修改的情况）。
- setval（sequence, newvalue）——将当前序列设置一个新值。

输入

name——创建序列的名称。

invalue——用于决定序列方向的值。一个正值（缺省为 1）将导致一个递增序列。一个负值将导致一个递减序列。

mnvalue——序列达到的最小值。递减序列的缺省值是-2147483647，递增序列的缺省值是 1。

mxvalue——序列能达到的最大值。递增序列的缺省值是 2147483647，递减序列的缺省值是-1。

stvalue——序列开始的初始值。

cavalue——表示 PostgreSQL 是否必须预先分配序列数字并保存在内存中以便于较快访问。最小及缺省值是 1。

CYCLE——表示序列是否应该继续越过最大或最小值。如果达到了外部边界（最小值或最大值），序列将重新从反向区域开始（最小值或最大值）。

输出

CREATE（若成功执行则返回此消息）。

ERROR: Relation 'sequence' already exist（若序列已经存在，则返回此消息）。

ERROR: DefineSequence: MINVALUE(start) can't be >=MAXVALUE(max)（若起始值超出了范围则返回此消息）。

ERROR: DefineSequence: START value(start) can't be <MINVALUE(min)（若起始值超出了范围则返回此消息）。

ERROR: DefineSequence: MINVALUE(min) can't be >= MAXVALUE(max)（若最小值和最大值相互冲突则返回此消息）。

注意

序列实际上以一个一行表保存于数据库中。另外一种决定当前值的方法是：

```
SELECT last_value FROM sequence_name;
```

SQL-92 兼容性

在 SQL-92 中无 CREATE SEQUENCE 语句。

示例

下面例子显示了怎样创建一个序列并应用为表中缺省值的：

```
CREATE SEQUENCE chk_num INCREMENT 1 START 1;

CREATE TABLE mytable
      (   check_number int DEFAULT NEXTVAL('chk_num'),
```

```
            description VARCHAR(40),
            amount MONEY
        );
```

CREATE TABLE

用法

```
CREATE[TEMPORARY | TEMP] TABLE tablename
(
     columnname columntype[NULL|NOT NULL][UNIQUE]
       [DEFAULT defvalue][column_constraint|PRIMARY KEY][,...]
    [,PRIMARY KEY(column[,...][,CHECK (condition)]
    [,table_constraint]
)
[INHERITS(inheritable[,...])]
```

描述

CREATE TABLE 是一个用于为当前数据库输入新表类的全功能命令。使用最基本的形式 CREATE TABLE 可以简单列出列名称和数据类型。然而，指定 PRIMARY KEYS、DEFFULTS 和 CONSTRAINTS 后，此命令就会变得更复杂，所以必须作更多的解释。

通过使用 TEMP 或 TEMPORARY 关键字，表示 PostgreSQL 中正被创建的表仅存在于本对话期。一旦当前对话完成，此表将从数据库中自动删除。

CREATE TABLE 命令的语法根据列级或表范围的指示可能被打乱。

（1）列级命令

于列级，你可以指定许多子句，用来约束即将插入字段的可接受数据。使用 NULL 或 NOT 子句来指定在一个列中是否允许空值。

于列级，可使用关键字 UNIQUE 来强制列中的所有值具有唯一性。实际上，PostgreSQL 通过它在期望列上创建了一个唯一的索引。除了关键字 UNIQUE 外，你还可以指定当前列成为一个主键。一个主键意味着其值唯一且非空值，同时还表示其他表可能由于引用完整性原因必须依赖于此列。

通过使用关键字 DEFAULT 来为特定列指定缺省值。包括硬代码缺省值或函数结果。

与使用关键字 NULL、DEFAULT 和 UNIQUE 相比，用 CONSTRAINT 子句可以定义更高级的约束。不过，请注意显示命名约束可能与 CREATE TABLE 命令中的现有关键字交迭。例如，可以使用下面两种方法之一指定列为非空：

```
CREATE TABLE mytable (myfield1 VARCHAR(10) NOT NULL);
```

或

```
CREATE TABLE mytable (myfield VARCHAR(10)
      CONSTRAINT no_nulls NOT NULL);
```

事实上，两种方法都是确保列拒绝空值的有效途径。不过，CONSTRAINT 子句提供了更高级的功能。完整的列 CONSTRAINT 命令语法如下：

```
CONSTRAINT name
(
```

```
        [NULL |NOT NULL]|{UNIQUE|PRIMARY KEY|CHECK constraint|
        REFERENCES reftable(refcolumn)[MATCH mtype][ON DELETE delaction][ON
        UPDATE upaction][[NOT] DEFERRABLE]{INITIALLY chktime]
    }
    [,...]
```

通过使用 CHECK CONSTRAINT 子句，可以包含一个条件表达式来得到一个布尔类型结果。如果返回结果为 TRUE，则 CHECK CONSTRAINT 通过。

下面是更详细的列表，其中包含了有效列级约束子句例子：

■ **非空值（NOT NULL）约束**

列级 NOT NULL 约束的语法如下：

```
CONSTRAINT name NOT NULL
```

■ **唯一性（UNIQUE）约束**

列级 UNIQUE 约束的语法如下：

```
CONSTRAINT name UNIQUE
```

■ **主键（PRIMARY KEY）约束**

列级 PRIMARY KEY 约束的语法如下：

```
CONSTRAINT name PRIMARY KEY
```

■ **检查（CHECK）约束**

CHECK 约束对一个返回布尔值的条件表达式求值。列级语法如下：

```
CONSTRAINT name CHECK(condition[,...])
```

■ **引用（REFERENCES）约束**

REFERENCES 关键字允许外部列可以通过当前列作为引用完整性目的。REFERENCES 列级语法如下：

```
CONSTRAINT name REFERENCES reftable[(refcolumn)]
        [MATCH mtype]
        [ON DELETE delaction]
        [ON UPDATE upaction]
        [[NOT] DEFERRABLE]
        [INITIALLY chktime]
```

表 1-2 列出了 REFERENCES 命令的有效选项。

表 1-2 REFERENCES **命令的有效选项**

选　项	解　释
MATCH mtype	mtype 为下列之一： ■<default type>→可以对多键外来引用部分匹配（也就是说，部分列可能为空值等） ■MATCH FULL→多键外来引用中的所有列必须匹配（也就是说，所有列必须非空值等） ■MATCH PARTIAL→并非当前实现
ON DELETE DELACTION	其中的 delaction 为以下之一： ■NO ACTION→此为缺省选项。如果与外键有悖则产生错误 ■RESTRICT→与 NO ACTION 相同 ■SET DEFAULT→如果引用列删除，则列值被设置成缺省值 ■SET NULL→如果引用列删除，则列值被设置成空值 ■CASCADE→如果引用行删除，则删除当前行

续表

选 项	解 释
ON UPDATE upaction	其中的 upaction 为以下之一： ■NO ACTION→缺省选项。如果与外键有悖则产生错误 ■RESTRICT→与 NO ACTION 相同。 ■SET DEFAULT→如果引用列更新，则列值被设置成缺省值。 ■SET NULL→如果引用列更新，则列值被设置成空值。 ■CASCADE→对当前字段引用列的级联更新。 如果引用列更改则此命令改变当前行。如果引用行更改但引用列无变化，则无变化发生
INITIALLY chktime	其中的 chktime 为以下之一： ■DEFERRED→仅在当前事务末尾检查约束 ■IMMEDIATE→缺省选项。在每一语句后检查约束

（2）表级命令

许多列级命令与表级命令存在直接重叠。大部分情况下，语法相同，只是表级命令仍然要指定命令作用的列。列级命令只局限于当前列。

与列命令中定义的一样，表级命令中同样必须使用 PRIMARY KEY。不过，在这里，语法稍有不同。在表级命令中，使用…PRIMARY KEY(columnname)…格式而不是使用…columnname coltype PRIMARY KEY…格式来指定主键。

而且，CONSTRAINT 子句与表级命令稍有不同。下面列表显示了表级 CONSTRAINT 子句：

```
CONSTRAINT name{PRIMARY KEY|UNIQUE}(columnname[,…])
  [CONSTRAINT name]CHECK(constraint_clause)
  [CONSTRAINT name]FOREIGN KEY(column[,…])
  [REFERENCES reftable(refcolumn[,…])
      [MATCH matchtype]
      [ON DELETE delaction]
      [ON UPDATE upaction]
      [[NOT DEFERRABLE][INITIALLY chktime]]
```

■ **唯一性（UNIQUE）约束**

表级 UNIQUE 约束的语法如下：

```
CONSTRAINT name UNIQUE(column[,…])
```

■ **主键（PRIMARY KEY）约束**

表级 PRIMARY KEY 约束的语法如下：

```
CONSTRAINT name PRIMARY KEY(column[,…])
```

■ **外键（FOREIGN KEY）约束**

表级 FOREIGN KEY 约束的语法如下：

```
CONSTRAINT name FOREIGN KEY(column[,…])
```

■ **引用（REFERENCES）约束**

通过 REFERENCES 关键字可以使外部列对引用完整性目标限制于当前列之上。表级的 REFERENCES 一般语法如下：

```
CONSTRAINT name REFERENCES reftable[(refcolumn])
        [MATCH matype]
        [ON DELETE delaction]
        [ON UPDATE upaction]
        {[NOT DEFERRABLE]
        [INITIALLY chktime]
```

请参考表 1-2 了解表级 REFERENCES 命令有效选项列表。

输入

TEMP——表示新表是否为临时表。

TEMPORARY——表示新表是否为临时表。

tablename——创建表的名称。

columnname——新表中创建的列名称。

columntype——列具有的数据类型。（参见第 2 章“PostgreSQL 数据类型”了解关于数据类型的更多信息。）

NULL——表示列必须允许空值。

NOT NULL——表示列不应允许空值。

UNIQUE——表示所有值必须具有唯一性。

defvalue——供给列缺省值的值或函数。

Column constraint——作用于当前列的约束子句。

PRIMARY KEY——表示一个列的所有值是唯一及非空。

CHECK (condition)——用一个针对列或表的条件表达式来表示是否允许 INSERT 或 UPDATE 命令。

Table constraint——一个作用于当前表的约束子句。

INHERITS (inheritable)——指定当前表将继承所有字段的表。

输出

CREATE（若命令成功执行，则返回此消息）。

ERROR: DEFAULT: type mismatch（若缺省值类型与列数据类型不匹配，则返回此消息）。

注意

可以将一个有效列数据类型指定为数组，但并不强制一致的数组维数。

最新的 PostgreSQL 7.0.X 版本对每行数据有一个编译容量限制，即 8KB。通过改变此选项和重新编译源代码，每行可以达到 32KB 的限制。最新版本的 PostgreSQL 7.1 引入了一个称之为 TOAST（超大属性存储技术）的新功能，它可以提供无极限行大小限制。

虽然可以通过 UNIQUE 和 PRIMARYK KEY 子句交迭列，但最好不要通过此法直接交迭索引。一般存在一个与交迭索引关联的动作请求。

完全正常情况下，通过 MATCH 命令引用的表应该是捆绑了 UNIQUE 或 PRIMARY KEY 的列。但这一点在 PostgreSQL 并非强制性的。

SQL-92 兼容性

由于 CREATE TABLE 命令所带的属性如此之多，因而针对具体情况探讨 SQL-92 的兼

容性更为科学，下面部分就分别加以讨论。

（1）TEMPORARY 子句

在 PostgreSQL 中，临时表只是本地可见。然而，SQL-92 也定义了全局可见临时表的概念。另外，SQL-92 还进一步定义了带 ON COMMIT 子句的全局临时表，事务结束后，它可以用于删除表中行。

（2）UNIQUE 子句

SQL-92 中的 UNIQUE 子句也允许 UNIQUE 子句在表和列级使用这些附加选项：INITIALLY DEFERRED、INITIALLY IMMEDIATE、DEFERRABLE 和 NOT DEFERRABLE。

（3）NOT NULL 子句

在 SQL-92 的规范中，NOT NULL 子句也能使用如下选项：INITIALLY DEFERRED、INITIALLY IMMEDIATE、DEFERRABLE 和 NOT DEFERRABLE。

（4）CONSTRAINT 子句

在执行 CONSTRAINT 子句时，SQL-92 定义了一些在 PostgreSQL 中不存在的附加功能。SQL-92 支持 ASSERTATIONS 和 DOMAINS 的概念。PostgreSQL 并不直接支持这些概念。

（5）CHECK 子句

在 SQL-92 规范中，CHECK 子句也可使用如下选项：INITIALLY DEFERRED、INITIALLY IMMEDIATE、DEFERRABLE 和 NOT DEFERRABLE。

（6）PRIMARY KEY 子句

在 SQL-92 定义中，PRIMARY KEY 子句也可使用如下选项：INITIALLY DEFERRED、INITIALLY IMMEDIATE、DEFERRABLE 和 NOT DEFERRABLE。

示例

下面是一个如何用此命令来创建表 authors 的简单例子。它创建了 4 个字段：1 个主键（限制序列缺省值）、2 个强制性非空字段及 1 个日期字段：

```
CREATE TABLE authors
{
  Author_id INT PRIMARY KEY DEFAULT NEXTTVAL('serial'),
  Author_name VARCHAR(40)NOT NULL,
  Author_SSN VARCHAR(11) NOT NULL,
  Author_DOB DATE
};
```

下面代码创建了一个临时表，临时表有一个可容纳二维数组的字段：

```
CREATE TEMPORARY TABLE mytemp
{
      id       INT NOT NULL,
      matric   INT[][]
};
```

下面简单例子显示了如何使用 CREATE TABLE 命令通过包含 CHECK 约束实现数据的完整性。此例显示了如何使用一个列约束来要求作者必须年长于 18 岁：

```
CREATE TABLE author
(
      id       INT PRIMARY KEY,
      name     VARCHAR(40) NOT NULL,
      age      INT CHECK (author_age>17)
);
```

下面例子与前一例子相似，只是它显示了如何使用一个表约束。注意表约束是如何基于两个返回布尔真值字段条件而工作的：

```
CREATE TABLE author
(
      id       INT PRIMARY KEY,
      name     VARCHAR(40) NOT NULL,
      age      INT
CONSTRAINT chk_it CHECK(AGE>17 AND name<>'')
);
```

最后的例子显示了一些表如何从其他表中继承字段。另外，它示范如何用表级 PRIMARY KEY 子句创建多列主键：

```
CREATE TABLE new-author
{
  new-id INT PRIMARY KEY,
  hew-name VARCHAR(40)NOT NULL,
  CONSTRAINT multikey PRIMARY KEY(new_id,id)
};
 INHERITS(author)
```

CREATE TABLE AS

用法

CREATE TABLE AS *tablename* [(*columnname*[,…]) AS select_criteria

描述

CREATE TABLE AS 命令的功能与 SELECT INTO 命令很相似；它允许查询结果移用于新表上。

如果没有 columnname 子句，则在新表中创建所有的列。

输入

tablename ——创建新表的名称。

Colunname ——选中列的名称。

select_criteria——用于生成表数据的 SELECT 语句。

输出

参见 CREATE TABLE 和 SELECT 命令了解执行此命令的输出信息。

注意

执行此命令的用户将拥有此生成表。同样，用户应该具有创建表和从表中选择数据的权限。

SQL-92 兼容性

此命令是 PostgreSQL 的扩展命令；在 SQL-92 规范中没有定义 CREATE TABLE AS 命令。

示例

下面例子显示了如何从现有表 authors 创建一个名为 tmp_authors 的表，要求作者年长于 40 岁：

```
CREATE TABLE tmp_authors AS
      SELECT * FROM authors WHERE age>40;
```

CREATE TRIGGER

用法

```
CREATE TRIGGER trigname {BEFORE|AFTER}{event [OR…]}
ON table
FOR EACH{ROW | STATEMENT}
EXECUTE PROCEDURE func(args)
```

描述

CREATE TRIGGER 命令用于指定一个限制特定表相关事件的动作。此概念与 RULES 概念许多方面相似，但各自最佳使用对象不同。TRIGGER 最一般的用途是在表事件发生之前或之后维护引用的完整性。RULES 则最可能用于在事件进程中执行级联 SQL 命令。

CREATE TRIGGER 命令指定了何时激发触发器（即 BEFORE 或 AFTER）并指定触发的事件（即 INSERT、UPDATE 或 DELETE）。最后，当这些条件满足时，执行用户定义函数。

若设置为事件前激发触发器，则触发器可以改变（或忽略）插入前的数据。同样，如果设置为事件后激发触发器，则所有更改（包括删除、插入和更改）对于触发器均是可见的。

输入

trigname ——创建触发器的名称。

Table ——捆绑触发器的表的名称。

event——事件，即 INSERT、DELETE 或 UPDATE。

func(args)——当事件条件满足时激发的函数及其参数。

输出

CREATE（若成功执行则返回此消息）。

注意

触发器创建者也必须具有足够的相关权限。当前版本不支持 STATEMENT 触发器。

SQL-92 兼容性

SQL-92 并不包含 CREATE TRIGGER 语句。它是 PostgreSQL 的扩展命令。

示例

此例使用了一个名为 state_check( ) 的函数来检验新插入的州名长度是否大于 3 个字符长。

首先，你需要定义如下函数：

```
CREATE FUNCTION state_check()
```

```
RETURN opaque
      AS 'BEGIN
      IF length(new.statename)<3
      THEN RAISE EXCEPTION 'State names must be greater than 3 characters';
      END IF;
      RETURN new;
      END;'
  LANGUAGE 'plpgsql';
```

然后，你可以按下面方法定义触发器：

```
CREATE TRIGGER statecheck_trigger BEFORE INSERT ON authors EXECUTE PROCEDURE
state_check();
```

现在，你就可以通过插入一些数据来测试你的触发器了：

```
INERT INTO authors (statename) VALUES('Alabama');
(1 row inserted ok)

INSERT INTO authors (statement) VALUES('Al');
ERROR:State names must be greater than 3 characters
```

CREATE TYPE

用法

```
CREATE TYPE typename
(
      INPUT=in_function,
      OUTPUT=out_function,
      INTERNALLENGTH={in_length | VARIABLE}
      [,EXTERNALLENGTH={ext_length|VARIABLE}]
      [,DEFAULT=defaultval]
      [,ELEMENT=element]
      [,DELIMITER=delimiter]
      [,SEND=send_function]
      [,REVEIVE=rec_function]
      [,PASSEDBYVALUE]
)
```

描述

PostgreSQL 包括一些内置数据类型，然而，用户也可以通过 CREATE TYPE 命令来注册自己的数据类型。

在使用 CREATE TYPE 命令定义一个新类型前，两个函数（in_function 和 out_function）必须已经存在。函数 in_function 用于将数据转换成一个内部数据类型，以便于此类型定义中的操作符和函数使用。同样地，out_function 函数将数据还原转换成外部使用类型。

新创建的数据类型长度或固定或可变。定长类型必须在定义新数据类型时明确地指明。PostgreSQL 通过 VARIABLE 关键字假定数据类型是 TEXT 类型且长度可变。

当指定一个新数组数据类型时就要使用 ELEMENT 和 DELIMITER 关键字。ELEMENT 关键字指定元素数据类型为数组，DELIMITER 关键字指定分隔数组元素的分隔符。

当一个外部计算机使用新创建的数据类型时，必须定义 send_function 和 rec_function 函数。以上函数用于将数据转换成一种格式或从一种格式进行转换，此格式为外部系统可用格式。如果没有定义此函数，则假定内部数据类型是整个计算机体系中的可接受格式。

PostgreSQL 通过关键字 PASSEDBYVALUE 来指定使用新数据类型的操作符和函数必须显式传递值，而不是通过引用方式传递。

输入

typename——新创建数据类型的名称。

in_function——将外部数据类型转换成内部数据类型的函数。

out_function——将内部数据类型转换成外部数据类型的函数。

in_length——用于指定内部数据类型内部长度的值或者关键字 VARIABLE。

Defaultcal——当数据不存在时所显示的缺省值。

Element——如果新创建类型是一个数组，它将指定数组中的元素类型。

Delimiter——如果新创建类型是一个数组，它则指定数组元素间的分隔符。缺省分隔符是一个逗号（,）。

send_function——指定将数据转换为便于外部计算机使用形式的函数。

rec_function——指定从数据一种形式转换为本地计算机所需格式、便于外部计算机使用的函数。

PASSEDBYVALUE——如果使用了这一变量，则它表示使用了新数据类型的操作符或函数必须以值而不是引用方式传递参数。

输出

CREATE（若命令成功执行则返回此消息）。

注意

指定的数据类型名称必须具有唯一性，它必须少于 31 个字符长，而且它不能以下划线（_）开头。

in_function 和 out_function 函数必须同时定义接受 1 个或 2 个 opaque 类型参数。

你不能使用 PASSEDBYVALUE 传递内部长度大于 4 个字节的值。

SQL-92 兼容性

SQL-92 并没有定义 CREATE TYPE 命令，不过，在 SQL3 方案中作了定义。

示例

下面例子创建了一个名为 deweydec 的数据类型，它将用于表示 Dewey 小数。此例假定 dewey_in 和 dewey_out 函数早已定义：

```
CREATE TYPE deweydec
        (
            INTERNALLENGTH=16,
            INPUT=dewey_in,
```

```
            OUTPUT=dewey_out
        );
```

CREATE USER

用法

```
CREATE USER username
  [WITH[SYSID uid][PASSWORD password]
  [CREATEDB | NOCREATEDB]
  [CREATEUSER | NOCREATEUSER]
  [IN GROUP groupname [,...]]
  [VALID UNTIL abstime]
```

描述

CREATE USER 命令为当前 PostgreSQL 数据库添加一个新用户。唯一的所需变量就是新用户的名称，此名称必须具有唯一性。缺省情况下，由 PostgreSQL 分配此用户的用户标识号（UID），但也可通过包含 WITH SYSID 子句来指定。

另外，通过指定 IN GROUP 命令可以将一个新用户包含到现有组中。同样，在创建用户的同时还可指定其权限。用户可以通过包含 CREATEUSER 子句授予创建自身用户的权限。同样，可以通过包含 CREATEDB 选项来指定用户创建自身数据库的权限。

PostgreSQL 允许将用户名设为在给定时间内自动失效。通过使用 VALID UNTIL 子句来指定一个有效期绝对时间。

输入

username——创建用户的名称。

Uid——新用户的用户标识号。

Password——新用户密码。

Groupname——新用户所属组。

Abstime——如果存在，则它将指定新用户名失效期的绝对时间。否则，用户名永远有效。

输出

CREATE USER（若成功执行则返回此消息）。

注意

用户名在当前数据库中必须唯一存在。

创建者必须有足够的权限来执行 CREATE USER 命令。而且，创建者将成为通过 CREATE USER 命令创建对象的所有者。

NOCREATEBD 和 NOCREATEUSER 都是缺省项。

SQL-92 兼容性

在 SQL-92 中无 CREATE USER 命令，它是 PostgreSQL 的扩展命令。

示例

在当前数据库中创建一个新用户并指定密码。将用户分配到组 managers：

```
CREATE user bryan WITH PASSWORD '08f30w0'
```

```
        IN GROUP managers;
```

创建一个有密码的新用户。授予此用户创建新用户但不能创建新数据库的权限。并将此用户名失效期设为 2002 年 1 月 30 日：

```
CREATE USER bryan WITH PASSWORD '08f30w0'
      NOCREATEDB CREATEUSER
      VALID UNTIL 'Jan 30 2002';
```

CREATE VIEW

用法

```
CREATE VIEW viewname AS SELECT selectquery
```

描述

视图是一个实现常规查询的有用方法。不需每次需要时使用完整查询，你可以将之定义为一个视图并在每次需要时通过一个非常简单的语法来重新使用它。

输入

viewname——创建视图的名称。

Selectquery——为新创建视图提供列和行定义的 SQL 查询。

输出

CREATE（若成功执行则返回此消息）。

ERROR: Relation 'viewname' already exists（若所定义视图名称已经存在则返回此消息）。

NOTICE: create: attribute name 'column' has an unknown type（若一个显式查询并没有定义静态变量数据类型，则返回此消息。请参见“示例”部分）。

注意

视图有时作为一个实际表被引用，不过，将它看作宏替换更有利于概念上的正确理解。

目前，视图只是可读的。

SQL-92 兼容性

SQL-92 定义的视图是可更改的。而目前 PostgreSQL 版本中的视图是只读的。

示例

为表 books 创建一个视图，其中只是小说书籍应该归还：

```
CREATE VIEW fictionbooks AS
      SELECT * FROM books WHERE genre='fiction';

SELECT * FROM fictionbooks WHERE author_name='Shakespeare,W.";
 author_name     title                    genre       call_number
 ------------------------------------------------------------
 Shakespeare,W.  Complete Works Vol.1     Fiction     842.12 Sha
 Shakespeare,W.  Complete Works Vol.2     Fiction     842.13 Sha
```

创建一个返回静态结果的显示查询：

```
CREATE VIEW errmsg AS SELECT text 'Error: Not Found';
```

DECLARE

用法

```
DECLARE cursorname [BINARY][INSENSITIVE][SCROLL]
      CURSOR FOR selectquery
      [FOR {READ ONLY | UPDATE [OF column[,...]]}]
```

描述

用户可通过 DECLARE 语句创建一个游标来保存和导航查询结果。缺省时，PostgreSQL 返回文本格式数据，不过，若包含 BINARY 关键字也可返回二进制格式数据。

返回的是二进制信息数据，则要求调用体能转换和处理此类数据（标准的 PSQL 前端不能处理二进制数据）。但是，返回纯二进制格式数据也有一定优点，它一般对服务器的工作负荷少、传输数据容量小。

输入

cursorname——创建游标的名称。

INSENSITIVE——SQL-92 的保留关键字。PostgreSQL 忽略此关键字。

SCROLL——SQL-92 的保留关键字。PostgreSQL 忽略此关键字。

selectquery—— 用于定义针对创建游标的行和列的 SQL 查询。

READ ONLY—— 表示游标为只读的关键字。此时 PostgreSQL 仅产生只读游标，PostgreSQL 忽略此关键字。

UPDATE—— 表示游标为可更改型的关键字。PostgreSQL 仅产生只读游标，PostgreSQL 忽略此关键字。

column—— 与 UPDATE 关键字配合使用。而 PostgreSQL 忽略之。

输出

SELECT（若成功执行则返回此消息）。

NOTICE: BlankPortalAdssignName: portal 'cursorname' already exists（若此游标名称已经存在，则返回此消息）。

NOTICE: Named portals may only be used in begin / end transaction block（若在事务块中没有声明游标则返回此消息）。

注意

因为 PostgreSQL 可以返回指定结构的二进制数据，所以，可以按高位元置前或低位元置前顺序排列数据。但返回的所有的文本数据都是中性排列。

SQL-92 兼容性

保留关键字 INSENSITIVE、SCROLL、READ ONLY、UPDATE 及 column 用于保持以后 SQL-92 的兼容性。目前，PostgreSQL 仅能创建只读游标。

SQL-92 只允许游标位于内嵌 SQL 命令或模块中。不过，PostgreSQL 也允许游标存在于交互式方法中。

SQL-92 指定游标通过命令 OPEN 打开。PostgreSQL 假定游标一旦创建则认为已经打开。不过，命令 ecpg（Postgre 内嵌 SQL 预处理器）支持 OPEN 命令按与 SQL-92 规范兼容的方式使用。

关键字 BINARY 是 PostgreSQL 的一个扩展，在 SQL-92 规范中没有此关键字定义。

示例

下例创建了一个用于表 authors 中的游标：

```
DECLARE newauthors CURSOR FOR
      SELECT * FROM authors WHERE status='New';
```

DELETE

用法

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

描述

DELETE 命令用于从表中删除所有的或某些行。使用 WHERE 条件来指定将删除的目标行。

输入

table——包含删除行的表。

Condition——用可选条件 SQL WHERE 来指定删除行。

输出

DELETE count（若目标行数成功删除则返回此消息）。

注意

执行删除行的用户必须具有操作目标表及 WHERE 指定表的权限。

使用不带 WHERE 条件的 DELETE 从会导致表中的所有行都将被删除。虽然 TRUNCATE 命令不是 SQL-92 规范中的一部分，但它能更有效地执行相同功能。

SQL-92 兼容性

DELETE 命令与 SQL-92 兼容。但 SQL-92 还允许 DELETE 作用于游标，而游标在 PostgreSQL 中是只读的。

示例

下面代码从表 authors 中删除所有行：

```
DELETE FROM authors
```

下面代码从表中删除所有作者薪水少于$10,000 的行：

```
DELETE FROM authors WHERE salary<10000;
```

DROP AGGREGATE

用法

```
DROP AGGREGATE aggname type
```

描述

DROP AGGREGATE 命令用于删除根据当前数据库命名聚集的所有引用。

输入

aggname——删除聚集的名称。

Type——聚集的数据类型。

输出

DROP（若命令成功执行则返回此消息）。

NOTICE: RemoveAggregate: aggregate 'agg' for 'type' does not exist（若当前数据库中不存在此聚集则返回此消息）。

注意

只有聚集的所有者或超级用户才能执行此命令。

SQL-92 兼容性

在 SQL-92 规范中没有 CREATE 或 DROP AGGREGATE 命令。它是 PostgreSQL 的扩展命令。

示例

删除聚集 complex_sum:

```
DROP AGGREGATE complex_sum complex;
```

DROP DATABASE

用法

```
DROP DATABASE databasename
```

描述

DROP DATABASE 命令删除数据库及所有相关数据。

输入

databasename——删除数据库的名称。

输出

DROP DATABASE（若成功执行则返回此消息）。

ERROR: user 'username' is not allowed to create/drop database（若用户无足够权限删除数据库则返回此消息）。

ERROR: dropdb: cannot be executed on the template database（若用户试图删除模板数据库则返回此消息）。

ERROR: dropdb: cannot be executed on an open database（若此命令试图作用于当前已打开的数据库，则返回此消息）。

ERROR: dropdb: database 'name' does not exist（若没有发现指定数据库名则返回此消息）。

注意

你不能在当前数据库上执行 DROP DATABASE 命令。一般，此命令作用于与之相连的另一个数据库或在命令行执行 dropdb 命令。

由于需要物理删除文件，所以命令 DROP DATABASE 不能在事务内部执行。通常，一个 DROP 命令只修改系统目录，所以，它们是可以回滚的。因为 ROLLBACK 命令不能恢复所删除的文件系统对象，此命令必须作为原子实体执行，不能嵌入一个显式 BEGIN…commIT 子句中。

用户必须为数据库所有者或拥有超级用户权限才能执行 DROP DATABASE 命令。

SQL-92 兼容性

SQL-92 规范没有定义 DROP DATABASE 这一命令。它是 PostgreSQL 的扩展命令。

示例

此例删除名为 publisher 的数据库：

```
DROP DATABASE publisher
```

DROP FUNCTION

用法

```
DROP FUNCTION funcname ([type [,...]])
```

描述

删除在当前数据库中定义的函数。PostgreSQL 允许函数重载；所以，PostgreSQL 允许通过可选关键字 type 来区分相似名称函数。

输入

funcname——删除函数的名称。

Type——如果使用的话，则它表示函数所需数据类型。

输出

DROP（若成功执行则返回此消息）。

NOTICE: RemoveFunction: Function 'name' ("type") does not exist:（若函数名或数据类型无效则返回此消息）。

注意

用户必须是函数的所有者或具有数据库超级用户权限才能删除它。

SQL-92 兼容性

DROP FUNCTION 是 PostgreSQL 语言的一个扩展命令；SQL-92 规范中没有定义此命令。

示例

此例将从当前数据库中删除名为 last_check 的函数：

```
DROP FUNCTION last_check;
```

DROP GROUP

用法

```
DROP GROUP name
```

描述

从当前数据库中删除指定组。

输入

name——删除组的名称。

输出

DROP GROUP（若成功执行则返回此消息）。

注意

DROP GROUP 命令并没删除数据库中构成组的用户。

SQL-92 兼容性

DROP GROUP 命令是 PostgreSQL 语言的扩展。

示例

下例从当前数据库中删除组 managers：

```
DROP GROUP managers;
```

DROP INDEX

用法

```
DROP INDEX name
```

描述

DROP INDEX 命令用于从当前数据库中删除索引。

输入

name——删除索引的名称。

输出

DROP（若成功执行则返回此消息）。

ERROR: index 'index_name' nonexistent（若此索引名并不存在则返回此消息）。

注意

要执行此命令，用户必须为其所有者或具有超级用户索引的权限。

SQL-92 兼容性

SQL-92 将索引的概念留给具体的数据库来实现。所以，DROP INDEX 命令相对于 SQL-92 来说是一个具体的功能实现命令。

示例

下面例子从当前数据库中删除名为 checknumber 的索引：

```
DROP INDEX checknumber;
```

DROP LANGUAGE

用法

```
DROP LANGUAGE langname
```

描述

DROP LANGUAGE 命令用于从当前数据库中删除用户定义的语言。

输入

langname——删除语言的名称。

输出

DROP（若成功执行则返回此消息）。

ERROR: Language 'name' does not exist （若没有发现指定的此语言名称则返回此消息）。

注意

警告：PostgreSQL 不会检查是否函数依赖于将删除的语言。所以，有可能删除一个系统仍然需要的语言。

要执行 DROP LANGUAGE 命令，用户需要此对象的所有者或具有评定此数据库的超级用户权限。

SQL-92 兼容性

在 SQL-92 中没有 DROP LANGUAGE 命令；它是一个 PostgreSQL 扩展命令。

示例

下例从系统中删除语言 mylang：

```
DROP LANGUAGE mylang;
```

DROP OPERATOR

用法

```
DROP OPERATOR id (type | NONE [,...])
```

描述

此命令用于从当前数据库中删除一个现有操作符。通过使用关键字 type，你可以结合关键字 NONE 来指定左操作符或右操作符。

输入

id——删除操作符的标识。

Type——左或右操作符的数据类型。

输出

DROPF（若成功执行则返回此消息）。

ERROR: RemoveOperator: binary operator 'oper' taking type 'type' and 'type2' does not exist（若指定操作符在当前数据库中并不存在则返回此消息）。

ERROR: RemoveOperator: left unary operator 'oper' taking type 'type' does not exist（若指定的左一元操作符并不存在则返回此消息）。

ERROR: RemoveOperator: right unary operator 'oper' taking type 'type' does not exist（若指定的右一元操作符并不存在则返回此消息）。

注意

DROP OPERATOR 命令并不检查与删除操作符有关的依赖性。所以，用户必须自我确保操作符删除后所有的相关依赖性继续满足要求。

SQL-92 兼容性

DROP OPERATOR 是一个 PostgreSQL 扩展命令。在 SQL-92 中无此命令。

示例

此例对 int4 删除操作符=：

```
DROP OPERATOR =(int4, int4);
```

下面仅删除左一元操作符=：

```
DROP OPERATOR =(none, int4);
```

DROP RULE

用法

```
DROP RULE name
```

描述

DROP RULE 命令删除从当前数据库指定的特殊规则。一旦删除，PostgreSQL 将立即终止所有事件触发器对规则动作的应用。

输入

name——删除规则名称。

输出

DROP（若命令成功执行则返回此消息）。

ERROR: RewriteGetRuleEventRel: rule 'name' not found （若 PostgreSQL 没有发现指定的规则名则返回此消息）。

注意

为了执行此命令，使用者必须是规则的所有者或具有评定当前数据库的超级用户权限。

SQL-92 兼容性

DROP RULE 命令是 PostgreSQL 的扩展；在 SQL-92 中没有定义此命令。

示例

此例从数据库中删除名为 del_author 的规则：

```
DROP RULE del_author;`
```

DROP SEQUENCE

用法

```
DROP SEQUENCE  name [,...]
```

描述

DROP SEQUENCE 命令从当前数据库中删除命名的序列。PostgreSQL 实际上使用一个表来保存序列的当前值，所以 DROP SEQUENCE 命令与一个特殊命令 DROP TABEL 的实际工作原理一样。

输入

name——从当前数据库中删除序列的名称。

输出

DROP（若命令成功执行则返回此消息）。

NOTICE: Relation 'name' does not exist.（若 PostgreSQL 不能找到指定的规则名则返回此消息）。

注意

PostgreSQL 并不对删除序列的相关依赖性作检查。所以，用户的职责应该在执行 DROP SEQUENCE 命令前，应确保系统与删除序列不存在任何依赖性。

使用此命令的用户必须是此序列的所有者或具有数据库的超级用户权限。

SQL-92 兼容性

DROP SEQUENCE 是 PostgreSQL 的一个扩展命令；在 SQL-92 规范中没有等价命令定义。

示例

下面例子从数据库中删除名为 check_numb_seq 的序列：

```
DROP SEQUENCE check_numb_seq;
```

DROP TABLE

用法

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

描述

DROP TABLE 命令从当前数据库中删除被命名的表、相关索引及任何相关视图。

输入

name——删除表的名称。

输出

DROP（若命令成功执行则返回此消息）。

ERROR: Relation 'name' Does not Exist!（若当前数据库中不存在表名则返回此消息）。

注意

PostgreSQL 并不检查或警告由于 DROP TABLE 命令的执行可能给 FOREIGN KEY 带来的关系影响。所以，用户有责任确保其他关系不受到此命令的影响。

由于需要物理删除文件，所以 DROP TABLE 命令不能在事务内部执行。通常 DROP 命令只修改系统目录，所以，它们可以回滚。但由于 ROLLBACK 命令不能删除文件系统对象，所以此命令必须作为原子实体执行而不能嵌入一个显式 BEGIN...COMMIT 子句中。

命令的用户必须为表及相关对象的所有者或具有操作当前数据库的超级用户权限。

SQL-92 兼容性

DROP TABLE 命令与 SQL-92 基本兼容。不过，在 SQL-92 规范在命令中还包含了关键字 RESTRICT 和 CASCADE。这些关键字用于限制或级联删除表与其他引用对象的关系。目前版本的 PostgreSQL 并不支持这些命令。

示例

删除表 authors：

```
DROP TABLE authors;
```

删除表 authors 及 payroll：

```
DROP TABLE authors, payroll;
```

DROP TRIGGER

用法

```
DROP TRIGGER trigname ON tablename
```

描述

DROP TRIGGER 命令将删除当前数据库中指定的触发器。

输入

trigname——从数据库中删除的触发器名称。

Tablename——保留被命名触发器的表名称。

输出

DROP（若命令成功执行则返回此消息）。

ERROR: DropTrgger:there is no Trigger 'name' on relation 'table':（若 PostgreSQL 不能定位指定的触发器名则返回此消息）。

注意

执行此命令的用户必须为此对象的所有者或具有访问当前数据库的超级用户权限。

SQL-92 兼容性

在 SQL-92 规范中没有定义 DROP TRIGGER 命令。它是 PostgreSQL 语言的一个扩展命令。

示例

此例将从表 payroll 中删除触发器 state_checktrigger:

```
DROP TRIGGER state_checktrigger ON payroll;
```

DROP TYPE

用法

```
DROP TYPE name
```

描述

DROP TYPE 命令用于从当前数据库中删除指定类型。

输入

name——删除的数据类型名称。

输出

DROP（若成功执行则返回此消息）。

ERROR: RemoveType: type 'ame' does not exist（若 PostgreSQL 不能定位指定的数据类型名则返回此消息）。

注意

用户必须为对象的所有者或具有访问删除对象类型的超级用户权限。

PostgreSQL 不会对删除的对象类型作任何依赖性检查。所以，用户的职责是必须要确保任何依赖于数据类型的操作符、函数、触发器或其他对象不会因为此数据类型的删除而导致不一致状态出现。

SQL-92 兼容性

SQL-92 没有指定 DROP TYPE 命令；然而，它是 SQL3 规范的一部分。

示例

从数据库中删除数据类型 int4:

```
DROP TYPE int4;
```

警告

此动作可能极为危险：int4 对象是 PostgreSQL 系统的一个重要部分。此例只作演示说明之用——不要执行它！删除 int4 对象将会导致数据库的严重崩溃。

DROP USER

用法

```
DROP USER username
```

描述

DROP USER 命令用于从当前数据库中删除用户。

输入

username—删除的用户名。

输出

DROP USER（若命令成功执行则返回此消息）。

ERROR: DROP USER: user 'name' does not exist（若没有发现指定的用户名则返回此消息）。

DROP USER: user 'name' owns database 'name'（若用户试图删除拥有任何数据库则返回此消息）。

注意

PostgreSQL 不允许删除一个拥有某个数据库的用户。不过，PostgreSQL 并不会对用户拥有对象作依赖性检查。所以，用户的职责是必须要确保当 DROP USER 命令完成后其他数据库对象不会导致不一致状态出现。

SQL-92 兼容性

DROP USER 命令是 PostgreSQL 语言的一个扩展命令。SQL-92 中没有此命令。

示例

从当前数据库中删除用户 bill：

```
DROP USER bill;
```

DROP VIEW

用法

```
DROP VIEW name
```

描述

DROP VIEW 命令将从当前数据库中删除指定的视图。

输入

name—删除的视图名称。

输出

DROP（若命令成功执行则返回此消息）。

ERROR: RewriteGetRuleEventRel: rule '_RETname' not found（若在当

前数据库中不存在此视图名则返回此消息)。

注意

DROP VIEW 命令从当前数据库中删除指定视图。

SQL-92 兼容性

SQL-92 规范为 DROP VIEW 命令定义了一些附加功能：RESTRICT 和 CASCADE。这些关键字决定引用此视图的项目也被同时删除。缺省情况下，PostgreSQL 只删除明确指定的视图。

用户的职责是确保当 DROP VIEW 命令完成后其他数据库对象不会产生不一致状态。

示例

下面命令将从当前数据库中删除视图 fictionbooks：

```
DROP VIEW fictionbooks;
```

END

用法

```
END [WORK | TRANSACTION]
```

描述

关键字 END 用于完成显式 PostgreSQL 事务。

显式 BEGIN...END 子句用于封装一系列 SQL 命令以保证其正确执行。如果系列命令中的任何一个命令执行失败，它就可能导致整个事务回滚、数据库返回到初始状态。

缺省时，PostgreSQL 中的所有命令都将在一个隐含事务中执行。END 关键字等同于关键字 COMMIT。

输入

无。WORK 和 TRANSACTION 是无实际影响的可选项。

输出

COMMIT（若成功执行则返回此消息)。

NOTICE: COMMIT: no transaction in progress（若无当前事务则返回此消息)。

注意

一般情况下，最好使用 PostgreSQL 关键字 COMMIT，从而保持与 SQL-92 的兼容性。

请参见 ABORT、BEGIN 和 ROLLBACK 命令部分了解相关事务更多信息。

SQL-92 兼容性

END 关键字是 SQL-92 的扩展。它等价于 SQL-92 的关键字 COMMIT。

示例

下面例子显示了如何使用 END 关键字来终止一个 PostgreSQL 事务：

```
SELECT * FROM authors;

        name          LastCheck        Status
        ------------------------------
```

```
     Frank        $800.00        Active
     Bill         $500.00        Inactive
BEGIN;
INSERT INTO authors(name, lastcheck, hiredate)
     VALUES('Sam',700.00,'Active');
END;

SELECT * FROM authors;

     name         LastCheck       Status
     ------------------------------
     Frank        $800.00     Active
     Bill         $500.00     Inactive
     Sam          $700.00     Active
```

EXPLAIN

用法

EXPLAIN [VERBOSE]*query*

描述

EXPLAIN 命令用于勾画和跟踪查询的执行过程。它给出了洞察 PostgreSQL 规划器是如何针对提供查询产生一个执行规划的能力。它同时还显示了将使用的索引及应用到的联合算法。

EXPLAIN 命令的输出生成第一个元组返回前的起始时间、对于所有元组的总时间及使用的扫描类型（如序列、索引等等）。

参数 VERBOSE 将导致 EXPLAIN 命令完全倾倒整个内部规划树而不只是概要。此选项的典型应用是性能及高级调试方案。

输入

VERBOSE—可选关键字，它将产生完整的执行计划及所有内部状态。对调试很有用。

query—EXPLAIN 命令所规划的查询。

输出

NOTICE: QUERY PLAH: plan（返回此消息及执行规划）。

EXPLAIN（执行规划成功后返回此消息）。

注意

请参见第 10 章“常规管理任务”及其“操作性能优化”一节了解有关查询优化的更详细信息。

SQL-92 兼容性

SQL-92 中没有 EXPLAIN 命令，它是 PostgreSQL 的扩展命令。

示例

在下例中，假设表 authors 仅有一个数据类型为 int4 的字段及 1000 行数据。另外，此例还假设表 authors 中没有设置索引：

```
EXPLAIN SELECT * FROM authors;
NOTICE: QUERY PLAN:
Seq Scan on authors (cost=0.00..4.68 rows=1000 width=4)
EXPLAIN
```

下面一个例子包含了另一个 WHERE 约束及表 authors 中单字段上的一个索引。注意总开销时间的改进及只有一行返回的事实：

```
EXPLAIN SELECT * FROM authors WHERE i=100;
NOTICE: QUERY PLAN:
Index Scan using fi on authors (cost=0.00..0.38 rows=1 width=4)
EXPLAIN
```

下面最后一个例子在前面例子的基础上还包含了一个 sum()聚集。请注意聚集的起始时间是.38，它也是索引扫描的总时间。当然，这是由于直到数据传递给聚集后，它才能起作用。

```
EXPLAIN SELECT sum(i) FROM authors WHERE i=100;
NOTICE: QUERY PLAN:
Aggregate (cost=0.38..0.38 rows=1 width=4)
Index Scan using fi on authors (cost=0.00..0.38 rows=1 width=4)
EXPLAIN
```

FETCH

用法

```
FETCH [FORWARD | BACKWARD | RELATIVE]
      [number | ALL | NEXT | PRIOR]
      {IN | FROM}cursor
```

描述

FETCH 命令从被定义的游标中取回行。此游标必须已由 DECLARE 语句定义。

取回行的数量可以由一个带正负号的整数定义，也可为 ALL、NEXT 或 PRIOR 之一。

除取回的行数外，同时可指定下一个取回动作的方向。缺省情况下，PostgreSQL 按 FORWARD 方向进行搜索。不过，通过使用一个带符号的整数，搜索方向可以根据指定的关键字来更改。例如，FORWARD -1 与 BACKWARD 1 作用相同。

输入

FORWARD——从当前相关位置向前取回行。

BACKWARD——从当前相关位置向后取回行。

RELATIVE——用于保持与 SQL-92 的兼容，无实际作用。

number——一个表示按指定方向取回行数的带符号整数。

ALL——按指定方向取回剩余的所有行。

NEXT——按指定方向取回下一个单行（例如，等价于使用计数 1）。

PRIOR——按指定方向取回前一个单行（例如，等价于使用计数-1）。

IN 或 FROM——使用关键字之一。

cursor——预定义的游标名称。

输出

若成功执行，则 FETCH 命令将返回要求行。

NOTICE: PerformPortalFetch: portal 'cursor' not found（若指定的游标没有定义则返回此消息）。

NOTICE: FETCH/ABSULUTE not supported, using RELATIVE（因为 PostgreSQL 不支持游标的绝对定位，所以返回此消息）。

ERROR: FETCH/ABSULUTE at current position is not supported（若用户试图执行 FETCH RELATIVE 0 命令，则返回此消息。此命令虽然在 SQL-92 有效，但 PostgreSQL 并不支持）。

注意

通过使用一个带符号的整数及方向语句，搜索方向可以转向。例如，下面命令的功能是相同的：

```
FETCH FORWARD 1 IN mycursor
FETCH FORWARD NEXT IN mycursor
FETCH BACKWARD PRIOR IN mycursor
FETCH BACKWARD -1 IN mycursor
```

目前的 PostgreSQ 版本支持只读游标而不能更改游标。所以，更改必须以显式方向进行且不能发生在游标中。

使用 MOVE 命令来通过游标导航而不必取回行数据。

SQL-92 兼容性

PostgreSQL 允许游标存在于嵌入式应用的外部，这是原来 SQL-92 规范的一个功能扩展。

而且，SQL-92 对 FETCH 命令定义了一些附加功能。通过 ABSOLUTE 命令来绝对定位游标和通过 INTO 命令在变量中保存结果，它们在 SQL-92 作了定义但在 PostgreSQL 中并不存在。

示例

下例显示了从表 authors 中创建的一个游标并使用 FETCH 取回指定行：

```
BEGIN;
DECLARE mycursor CURSOR FOR SELECT * FROM authors;
FETCH FORWARD 3 IN mycursor;

      Name            SSN             HireDate
      --------------------------------------
      Bill            666-66-6666 01/01/1980
      Sam             123-45-6789 05/21/1994
      Amy             999-99-9999 06/05/2001
FETCH BACKWARD 1 IN mycursor;
      Name            SSN             HireDate
      --------------------------------------
      Sam             123-45-6789 05/21/1994
FETCH FORWARD NEXT IN mycursor
      Name            SSN             HireDate
```

```
        ------------------------------------
        Amy             999-99-9999 06/05/2001
CLOSE mycursor;
COMMIT;
```

GRANT

用法

```
GRANT privilege [,...] ON object [,...]
      TO {PUBLIC | GROUP groupname | username}
```

描述

GRANT 命令用于为组、用户或群体分配指定的权限。缺省情况下，对象的创建者相应获得了分配于对象上的所有权限。用户而不是创建者需要授予显式权限或属于一个继承了此权限的组，然后他才能访问此对象。

GRANT 命令允许分配下列权限：

- SELECT——访问表中列的权限。
- INSERT——插入行到表中的权限。
- UPDATE——修改表中数据的权限。
- DELETE——删除表中行的权限。
- RULL——定义表中规则的权限。
- ALL——以上所有权限。

这些权限可以分配于以下对象：

- 表。
- 视图。
- 序列。

输入

privilege——为 SELECT、INSERT、UPDATE、DELETE、RULE 或 All 之一。

object——为 table、view 或 sequence 对象类之一。

PUBLIC——可选关键字，表示权限属于所有用户。

groupname——授予权限的组名称。

username——授予权限的特殊用户名。

输出

CHANGE（若命令成功执行则返回此消息）。

ERROR: ChangeAcl: class 'object' not found:（若指定对象不能定位并分配权限则返回此消息）。

注意

要给一个指定列授予访问权限，必须完成下面过程：

1. 不要授予用户访问表的权限。
2. 根据指定字段创建表的视图。
3. 授予用户访问视图的权限。

欲了解如何删除通过 GRANT 命令分配的权限，请参阅 REMOVE 命令相关信息。

SQL-92 兼容性

SQL-92 为 GRANT 命令定义了一些附加设置。它允许在列级授予权限。除此之外，SQL-92 规范中还包含以下几个方面：

（1）权限

- 引用。
- 用法。

（2）对象

- 字符集。
- 校对。
- 翻译。
- 范围。
- 权限授予选项。

示例

下面例子授予了用户 bill 对于表 authors 的一些权限：

```
GRANT SELECT, UPDATE ON authors TO bill;
```

下面授予组 managers 相对于表 authors 的所有权限：

```
GRANT ALL ON authors TO GROUP manager;
```

INSERT

用法

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

描述

INSERT 命令用于为表添加新行。而且通过使用 SELECT 查询，可以同时添加大量的行。

在插入过程中可以指定特定列，或若不包含的话，PostgreSQL 将试图为此列插入一个缺省值。

如果试图插入一个错误的数据类型到列中，则 PostgreSQL 会努力自动将数据转换成正确的数据类型。

输入

tablename——插入行的表名称。

column——与数据匹配的列的列表。

data——插入表中的真实数据。

query——用于产生插入数据的一个 SQL 查询。

输出

INSERT oid（若插入一行及其对象的 OID 则返回此消息）。

INSERT 0 number（若插入多行则返回此消息；消息中包含插入的行数）。

注意

执行此命令的用户必须具有在此表中执行插入的权限。

SQL-92 兼容性

INSERT 命令与 SQL-92 规范完全兼容。

示例

此例子显示了 INSERT 命令的一个基本应用。在 3 列表 authors 中插入数据：

```
INSERT INTO authors (Name,SSN,LastCheck)
      VALUES('Sam','333-33-3333',450.00);
```

下面例子显示了如何在 INSERT 命令中联合使用 SELECT 命令的。请注意从 SELECT 语句中返回的列是怎样与 INSERT 命令中指定的列匹配的：

```
INSERT INTO authors (Name,SSN,LastCheck)
       INSERT name,SSN, LastCheck from tempTable;
```

LISTEN

用法

```
LISTEN name
```

描述

LISTEN 命令与 NOTIFY 命令联合使用。LISTEN 命令在 PostgreSQL 后端注册一个名称并监听来自 NOTIFY 命令的通知。

多客户可以以同一个 LISTEN 名称进行监听。当监听到一个通知时，所有的客户都将得到通告。

输入

name——通过 PostgreSQL 注册的名称。

输出

LISTEN（若成功执行则返回此消息）。

NOTICE: Async_Listen: We are already listening on 'name'（如果后端已经注册了此 LISTEN 名称则返回此消息）。

注意

如果用双引号括起来，则此 LISTEN 名可以是任何 31 个字符的组合。

SQL-92 兼容性

LISTEN 是 PostgreSQL 的一个扩展命令；在 SQL-92 规范中没有此命令。

示例

此例将通过 LISTEN 命令注册一个名称，然后发送一个通知：

```
LISTEN IAmWaition;
NOTIFY IAmWaiting;
Asynchronous NOTIFY 'IAmWaiting' from backend with pid '2342'
  received.
```

LOAD

用法

```
LOAD filename
```

描述

LOAD 命令用于装载一个对象文件（来自 C 编译文件的一个.o 类型）以供 PostgreSQL 使用。装载此文件后，其中所有包含的函数都将变为可用。

如果 LOAD 命令没有显式给出，PostgreSQL 还可以在函数调用时自动装载所需对象文件。

如果对象文件中的代码改变，将执行 LOAD 命令刷新 PostgreSQL 并使这些变化成为可见。

输入

filename——装载对象文件的路径和文件名。

输出

LOAD 命令（若成功执行则返回此消息）。

ERROR: LOAD: could not open file 'name'（若不能发现指定的文件名则返回此消息）。

注意

PostgreSQL 后端必须具有使用此对象文件的权限；所以，用户在指定文件前需要考虑路径及权限。

在设计对象文件以防止错误发生时必须倍加小心。用户定义的对象文件中的函数不能调用其他用户定义的对象文件。理想情况是，所有的调用函数存在于同一个对象文件内或连接于一个标准 C、math 或 PostgreSQL 库文件上。

SQL-92 兼容性

SQL-92 规范中没有定义 LOAD 命令；它是 PostgreSQL 的一个扩展命令。

示例

装载一个用户定义对象文件以供使用：

```
LOAD '/home/bill/myfile.o'
```

LOCK

用法

```
LOCK [TABLE] tablename
```

或

```
LOCK [TABLE] tablename IN
[ROW | ACCESS]
{SHARE | EXCLUSIVE}MODE
```

或

```
LOCK [TABLE] tablename IN SHARE ROW EXCLUSIVE MODE
```

描述

LOCK TABLE 命令用于控制指定表的同时访问。缺省情况下，PostgreSQL 自动处理许多表锁场合。然而，也存在需要通过 LOCK 来协助工作的情况。

PostgreSQL 提供了以下几种锁类型：

- EXCLUSIVE——在事务执行期间阻止任何类型锁授权于此表。
- SHARE——允许其他用户同时共享此锁，但在事务执行期间阻止互斥型锁的使用。

以上锁型工作在以下授权级别上：

- ACCESS—整个表模式。
- ROWS—仅锁定单独的行。

表 1-3 列出了常用的锁类型、对应的典型用户及其他锁方式产生的冲突。

表 1-3 常用锁类型

锁类型	数据库操作	冲 突
访问共享型	SELECT （所有表查询）	（这是最低限制的锁）
访问互斥型	LOCK TABLE ALTER TABLE DROP TABLE VACUUM	（这是最严格限制的锁）
共享型	CREATE INDEX	ROW EXCLUSIVE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE
共享行互斥型		ROW EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE
互斥型		ROW SHARE ROW EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE
行共享型	SELECT...FOR UPDATE	EXCLUSIVE ACCESS EXCLUSIVE
行互斥型	INSERT UPDATE DELETE	SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE

输入

tablename——执行锁操作的表名称。

SHARE ROW EXCLUSIVE MODE——与 EXCLUSIVE 锁相似，但它允许其他用户使用 SHARE ROW 锁。

输出

LOCK TABLE（若命令成功执行则返回此消息）。

ERROR 'tablename': Table does not exist（若 LOCK 命令不能定位指定表则返回此消息）。

注意

为了防止死锁（当两个事务互相等待对方的完成时发生的暂停），让事务按相同顺序获得对象锁很重要。例如，如果事务更改了第一行，然后再更改第二行，那么一个单独的事务

必须也按此顺序更改第一行及第二行而不能颠倒。而且，如果一个事务中包含多个锁，则必须使用最严格的锁。

PostgreSQL将删除死锁并回滚到至少其中一个等待的事务中去解决它。

大部分锁类型（除ACCESS SHARE/EXFCLUSIVE外）都与Oracle的锁类型兼容。

SQL-92兼容性

在SQL-92的规范中使用SET TRANSACTION子句来定义当前表的访问方式，PostgreSQL也支持这一方式。（请参见SET命令）

LOCK TABLE命令是PostgreSQL的一个扩展。

示例

此例将整个表authosrs锁定来阻止在更改期间任何其他用户对它的访问：

```
BEGIN;
LOCK TABLE authors;
UPDATE authors SET status='active';
COMMIT;
```

MOVE

用法

```
MOVE [direction][count]{IN | FROM}cursorname
```

描述

用户通过MOVE命令可以通过游标导航而不必取回任何数据。它的工作方式与FETCH命令相似，只是它仅定位游标。

输入

direction——指定移动的方向：FORWARD或BACKWARD。

count——一个带正负号的整数或关键字（NEXT或PRIOR）；它们指定从当前位置移动多少行。

IN或FROM——使用其中一个选项；它们的功能相同。

cursorname——移动的游标名称；它必须是已经通过DECLARE语句定义的游标。

输出

MOVE（若命令成功执行则返回此消息）。

注意

通过在方向语句中使用一个带正负号的整数，可以让移动方向反向。例如，下面命令的功能是相同的：

```
MOVE FORWARD 1 IN mycursor
MOVE FORWARD NEXT IN mycursor
MOVE BACKWARD PRIOR IN mycursor
MOVE BACKWARD -1 IN mycursor
```

MOVE的工作方式与FETCH非常相似。请参考FETCH命令了解更多信息。

SQL-92兼容性

SQL-92并没有指定MOVE命令；可以从定义的位置通过FETCH命令取回行。实际上是

隐式移动到指定位置。

示例

下面例子定义了一个游标 mycursor，然后，通过此游标导航，使用 MOVE 命令取回指定的行：

```
BEGIN;
DECLARE mycursor CURSOR FOR SELECT * FROM authors;
MOVE FORWARD 3 IN mycursor;
FETCH NEXT IN mycursor;

     name        SSN             HireDate
     --------------------------------
     Sam S.      123-45-6789     12/01/1998
COMMIT;
```

NOTIFY

用法

```
NOTIFY name
```

描述

NOTIFY 命令与 LISTEN 命令联合使用来给注册了监听名称的客户发送通知消息。这是一个在客户和服务器进程间实现基本通信系统的方式。一个典型的用途是通知客户应用程序指定的表已经修改，提示客户应用程序显示它们的数据。

传递给客户应用程序的信息包含通知名称及后端进程的 PID。

输入

name——先前通过 LISTEN 注册的监听名称，用于接收通知。

输出

NOTIFY（若成功执行命令则返回此消息）。

注意

NOTIFY 事件实际上在一个 PostgreSQL 事务内部执行；所以，它具有一些重要的用途。

首先，直到整个事务提交才会发送通知。当通知为与表有关的规则或触发器一部分时尤为如此。直到整个包含此表的事务完成后才会发送此通知。

其次，如果前端监听在事务进程中收到一个通知，NOTIFY 事件将一直延迟直到事务完成。

让前端应用程序依赖于数个所接收的通知并不是一个好的做法。许多通知有可能在很短时间内连续发送，那么客户可能仅收到一条通知。

SQL-92 兼容性

NOTIFY 是 PostgreSQL 的一个扩展命令；在 SQL-92 规范中没有此命令。

示例

下面例子通过 LISTEN 命令注册了一个监听名称并发送一条通知：

```
LISTEN IAmWaiting;
```

```
NOTIFY IAmWaiting;
Asynchronous NOTIFY 'IAmWaiting' from backend with pid'2342' received.
```

REINDEX

用法

```
REINDEX {TABLE | DATABASE | INDEX} name[FORCE]
```

描述

REINDEX 命令用于恢复崩溃的索引系统。要运行此命令，postmaster 进程必须关闭，而且 PostgreSQL 必须带-o 和-P 选项执行。（它用于阻止 PostgreSQL 从起始处读系统索引。）

输入

TABLE——对指定表重建所有表索引。

SATABASE——对指定数据库重建所有系统索引。

INDEX——重建指定索引。

name——重建指定表名、数据库名或索引名。

FORCE——强制 PostgreSQL 重写当前索引，即使 PostgreSQL 认为此索引仍然有效。

输出

REINDEX（若命令成功执行则返回此消息）。

SQL-92 兼容性

这是 PostgreSQL 语言的一个扩展命令。在 SQL-92 规范中没有定义此命令。

示例

此例强制在数据库 acme 上执行 REINDEX 命令：

```
REINDEX DATABASE acme FORCE;
```

RESET

用法

```
RESET variable
```

描述

RESET 命令将一个运行变量重置回缺省设置。它与 SET variable TO DEFAULT 命令的功能等价。

输入

variable——重置回缺省值的变量名。

输出

SET（若成功执行则返回此消息）。

注意

请参见有关 SET 命令的进一步讨论及运行变量的列表。

SQL-92 兼容性

RESET 是 PostgreSQL 语言的一个扩展命令。在 SQL-92 规范中无 RESET 命令。

示例

此例恢复变量 DateStyle 的缺省设置：

```
RESET DataStyle;
```

REVOKE

用法

```
REVOKE privilege [,...]
      ON OBJECT [,...]
      FROM {PUBLIC | GROUP groupname | username}
```

描述

REVOKE 命令允许对象的所有者（或超级用户）回收授予用户、组或指定对象用户群的权限。

REVOKE 命令允许删除以下几种权限：

- SELECT——访问表中列的权限。
- INSERT——向表中插入行的权限。
- UPDATE——修改表中数据的权限。
- DELETE——删除表中行的权限。
- RULE——定义表中规则的权限。
- ALL——所有以上权限。

以上权限可以从下列对象中回收：

- 表。
- 视图。
- 序列。

输入

privilege——为 SELECT、INSERT、UPDATE、DELETE、RULE 或 ALL 之一。

object——为表、视图或序列对象类之一。

PUBLIC——可选关键字，表示权限属于所有用户。

groupname——删除权限的组名。

username——删除权限的特殊用户名。

输出

CHANGE（若命令成功执行则返回此消息）。

ERROR（若没有发现对象或不能回收指定权限则返回此消息）。

注意

请参见 GRANT 命令了解给用户或组分配权限的更多信息。

SQL-92 兼容性

SQL-92 规范对 REVOKE 命令附加了一些功能。它允许从列级回收权限还允许回收此处没有提及的另外一些权限，它们是：

- 用法。
- 授权选项。

■ 引用。

示例

此例显示了如何删除用户 bill 更改表 authors 中数据的权限：

REVOKE UPDATE，INSERT，DELETE NO authors FROM bill;

删除所有用户浏览或修改表 payroll 的权限：

```
REVOKE ALL ON payroll FROM PUBLIC;
```

ROLLBACK

用法

```
ROLLBACK [WORK | TRANSACTION]
```

描述

ROLLBACK 命令用于终止和倒退当前进程中的事务。当 PostgreSQL 收到一个 ROLLBACK 命令时，对此表所作的任何修改将自动恢复到原始状态。

缺省情况下，PostgreSQL 的所有命令都是在隐含事务中执行。封装了 SQL 命令集的 BEGIN…COMMIT 子句显式用法用于确保执行正确无误。如果命令集中的任何一个命令出错，则启动 ROLLBACK 命令，使数据库恢复到原始状态。

输入

无。WORK 和 TRANSACTION 是无实际功能的可选关键字。

输出

ABORT（若成功执行则返回此消息）。

NOTICE: ROLLBACK: no transaction in progress:（若进程中无当前事务则返回此消息）。

注意

COMMIT 命令用于成功确保事务动作成功完成。

请参见 ABORT、BEGIN 和 COMMIT 命令了解相关事务的更多信息。

SQL-92 兼容性

ROLLBACK 命令与 SQL-92 完全兼容。SQL-92 也将 ROLLBACK WORK 定义为一个有效语句，PostgreSQL 也支持这一语句。

示例

此例显示了如何使用 ROLLBACK 命令终止一个进程中的事务：

```
BEGIN;
SELECT * FROM authors;

      Name        SSN             Status
      ------------------------------
      Greg L. 123-45-6789         Active
      Mike D. 999-99-9999         Active

INSERT INTO authors(Name,SSN,Status)
```

```
        VALUES('Barry S.','555-55-5555','Inactive');

SELECT * FROM authors;

        Name        SSN             Status
        ------------------------------
        Greg L. 123-45-6789         Active
        Mike D. 999-99-9999         Active
        Barry S.555-55-5555         Inactive
```

SELECT

用法

```
SELECT [ALL |DUSTINCT[ON(expression[,…])]expression
  [AS name] [,…]
  [INTO[TEMPORARY|TEMP][TABLE]new_table]
  [FROM[ONLY]fromitem[alias] [,…]]
      [ON JOIN joincondition|USING(joinlist)] [,…]]
  [WHERE wherecondition]
  [GROUP BY column[,…]]
  [HAVING wherecondition[,…]]
  [{UNION [ALL]|INTERSECT|EXCEPT}secselect]
  [ODER BY column [ASC|DESC|USING operator] [,…]
  [FOR UPDATE [OF tablename[,…]]]
  [LIMIT(count | ALL) [{OFFSET |,}start]]
```

描述

SELECT 命令用于从单个或多个表中取回行数据。如果没有给出 WHERE 条件，则返回所有行。

（1）FROM 子句

FROM 子句指定包含于查询中的表。如果 FROM 子句只是一个简单的表名，缺省情况下，它包含根据继承关系所得的行。ONLY 选项将限制结果仅产生于指定的表。

FROM 子句也可指向 SUB-SELECT，而 SUB-SELECT 对执行高级组、聚集和有序函数很有用。

FROM 子句还可以指向 JOIN 语句，它用于联结两个明确的 FROM 地址。下面是 PostgreSQL 支持的 JOIN 类型：

- INNER JOIN | CROSS JOIN。将包含的源行直接联结而不对删除行作资格评价。

下面 3 个项目中讲到的 OUTER JOIN 是 PostgreSQL 7.1 及以上版本中的功能。PostgreSQL 的以前版本并不支持 OUTER JOIN。

- LEFT OUTER JOIN。返回左边全部源行，但只有右边行满足 ON 条件时才被返回。使用 NULLS 关键字来填充缺失的右边行以充分扩展左边行结果。

- RIGHT OUTER JOIN。与LEFT OUTER JOIN相反。返回所有的右边行，只有左边行满足ON条件时才被返回。使用NULLS关键字来填充缺失的左边行以充分扩展右边行结果。
- FULL OUTER JOIN。返回所有左边行数据（使用NULL后将扩展到右边）及所有右边行数据（使用NULL后将扩展到左边）。

（2）DISTINCT子句

DISTINCT子句允许用户指定是否返回复制行。缺省情况下返回所有（ALL），包括复制行。

通过联合使用ORDER BY时指定DISTINCT ON子句，可以基于指定列限制返回的复制行。

（3）WHERE子句

WHERE子句用于限制返回的行。组成一个有效WHERE子句的表达式是一个布尔类型表达式。例如：

```
WHERE expression1 condition expression2
```

再如：

```
WHERE Name='Barry'
```

式中的condition可以是=、<、<=、>、>=、<>、ALL、ANY、IN和LIKE之一。

（4）GROUP BY子句

GROUP BY子句用于将复制行整合成单个实体。所有选择的字段必须包含与整合行相同的行。

当聚集位子字段之中时，聚集函数将对每个组中的所有成员进行计算。

缺省情况下，GROUP BY试图对输入列进行计算。不过，如果使用SELECT BY子句，GROUP BY子句可以对输出列进行计算。另外，GROUP BY子句可以用于序列数。

（5）HAVING子句

HAVING子句过滤GROUP BY命令产生的行中组。

如WHERE对于GROUP BY条件的过滤一样，HAVING子句也是必需的。不过，WHERE子句在GROUP BY命令运行前起作用，而HAVING子句在GROUP BY命令完成后才执行。

（6）ORDER BY子句

ORDER BY子句通过指定列引导PostgreSQL对SELECT命令的输出结果进行排序。如果指定了多个列，则输出结果按指定列从左到右的顺序排列。

通过使用ASC（升序）或DESC（降序）选项来指定排序方向。ASC为缺省选项。

除了指定列名称外，还可使用各个列的顺序数。如果ORDER BY子句所声明的名称模糊不清，它可以假定一个输出列名称。其功能与GROUP BY子句相反。

（7）UNION子句

UNION子句允许输出结果为来自于两个或多个查询的行集合。为此，每个查询必须具有相同的列数及相同的数据类型。

缺省情况下，UNION组合中不包含复制行，但若指定了ALL选项则可以包含之。

（8）INTERSECT 子句

INTERSECT 子句从一组相似查询中搜集输出结果组合。为此，每个查询必须具有相同的列数及相同的数据类型。

INTERSECT 与 UNION 不同，因为只返回查询中的相同行（交集）。

（9）FOR UPDATE 子句

FOR UPDATE 子句对选择的行执行一个互斥型锁操作，帮助数据的修改。

（10）EXCEPT 子句

EXCEPT 子句从一组相似查询中返回一个输出结果组合。为此，每个查询必须具有相同的列数目及相同的数据类型。

EXCEPT 子句与 UNION 子句的不同之处在于返回第一个查询所得的所有行而不只是返回与第二列的非匹配行。

（11）LIMIT 子句

LIMIT 子句用于指定返回的最大行数。如果包含 OFFSET 选项，则在 LIMIT 命令开始起作用前先跳过对应数量行。

当 LIMIT 子句与一个 ORDER BY 命令联合使用时，它通常只返回有意义的结果；否则很难知道返回行的重要性。

输入

expression——表的列名称或一个表达式。

name——对列或表达式指定一个可换名称。通常用于对聚集结果更名（即 SELECT sum(check) AS TotalPayroll）。

TEMPORARY | TEMP——SELECT 执行结果被发送到一个特定的临时表中，一旦对话完成则立即被删除。

new_table——SELECT 执行结果被发送到一个指定名称的新查询中。（请参见 SELECT INTO 命令了解更多信息。）

fromitem——表、子选择或 JOIN 子句选择行名称。（请参见前面的 JOIN 命令讲解）

alias——为前述表定义一个可选名称。用于处理相同表联合时防止混淆。

wherecondition——一旦计算返回一个布尔值的 SQL 语句。输出结果决定查询首先要过滤的行。同时，星号（*）代表 ALL。

column——列名称。

sesselect——在 UNION 中使用的第二个 SELECT 语句。它是标准的 SELECT 语句，但在 SELECT 语句中不能包含 ORDER BY 或 LIMIT 子句，

count——在限制短语中返回的行数。

start——与 OFFSET 命令一起使用，直到返回了指定的起始行数后才开始返回数据。

输出

若成功执行则返回相关数据。

XX ROW（返回表示行数的数据集后发返回此消息）。

注意

执行 SELECT 命令的用户必须具有选择相关表的权限。

SQL-92 兼容性

SELECT 命令是 PostgreSQL 的主要组成部分，除了以下几个方面外，它与 SQL-92 保持兼容：

- LINIT_OFFSET—它是 PostgreSQL 的扩展命令。SQL-92 中没有此命令。
- DISTINCT ON—它是 PostgreSQL 的扩展命令。SQL-92 中没有此命令。
- GROUP BY—在 SQL-92 中，此命令仅针对输入列名称，而 PostgreSQL 两者均能使用。
- ORDER BY—在 SQL-92 中，此命令仅针对输出（结果）列名称，而 PostgreSQL 两者均能使用。
- UNION 子句—在 SQL-92 中，此命令允许包含另一个选项 CORRESPONDING BY。但在 PostgreSQL 中不能使用此选项。

示例

下面例子显示了一个简单的 SELECT 语句，从表 authors 中选择匹配指定名称的行：

```
SELECT * FROM authors WHERE name='Sam';
      Name    SSN              HireDate
      ----------------------------
      Sam     111-11-1111      01-01-1990
      Sam     123-45-6789      04-23-2001
      Sam     999-99-9999      06-22-1971
      Sam     333-33-3333      09-19-1995
```

下面是同一个加入了 ORDER BY 功能的例子：

```
SELECT * FROM authors WHERE name='Sam'ORDER BY HireDate;
      Name    SSN              HireDate
      ----------------------------
      Sam     999-99-9999     06-22-1971
      Sam     111-11-1111     01-01-1990
      Sam     333-33-3333     09-19-1995
      Sam     123-45-6789     04-23-2001
```

下面还是同一个例子。但这一次使用一个 LIMIT 命令来返回两个最新的成员：

```
SELECT * FROM authors WHERE name='Sam'ORDER BY HireDate DESC LIMIT 2;
      Name    SSN              HireDate
      ----------------------------
      Sam     123-45-6789     04-23-2001
      Sam     333-33-3333     09-19-1995
```

下面代码将当前表 authors 和表 payroll 联合得到最后的检查数据：

```
SELECT * FROM authors JOIN payroll ON authors.ssn=payroll.snn;
      Name SSN            HireDate     SSN         LastCheck
      ----------------------------------------------
      Sam  123-45-6789    04-23-2001 123-45-6789  500.00
      Sam  999-99-9999    06-22-1971 999-99-9999  674.00
      Sam  333-33-3333    09-19-1995 333-33-3333  800.00
      Sam  111-11-1111    01-01-1990 111-11-1111  964.15
```

使用聚集函数（如 count()、sum()等等）可以提供简单的归总数据的方法，但这种工作往往是繁琐费力的。在下面例子中，你可以使用 count()函数告诉我们有多少个作者叫 Sam：

```
SELECT count(name) FROM authors WHERE name='Sam';

count
-----
4
```

使用 SUM 函数、GROUP BY 和一个 JOIN 来告诉我们所有叫 Sam 的人支付的总数：

```
SELECT authors.name,sum(payroll.LastCheck) AS Total
FROM authors JOIN payroll ON authors.ssn=payroll.ssn
WHERE authors.name='Sam'
GROUP BY authors.name;
      Name        Total
      -----------------
      Sam         2398.15
```

此例显示了子选择语句是如何工作的。选择 payroll 中最后检查时支付额多于$900 的所有人，然后，他们的名字从与 authors 的联合中显示：

```
SELECT name,ssn FROM authors
      WHERE ssn IN(SELECT ssn FROM payroll WHERE LastCheck>900);
      Name        SSN
      ---------------------
      Sam         111-11-1111
```

SELECT INTO

用法

```
SELECT[ALL|DISTINCT[ON(expression[,…])]expression
      [AS name] [,…]
          [INTO[TEMPORARY|TEMP][TABLE]new_table]
      [FROM[ONLY]fromitem[alias] [,…]]
          [ON JOIN joincondition|USING (joinlist)] [,…]]
      [WHERE wherecondition]
      [GROUP BY column[,…]]
      [HAVING wherecondition[,…]]
      [{UNION [ALL]|INTERSECT|EXCEPT}secselect]
      [ORDER BY column [ASC|DESC|USING operator] [,…]
      [FOR UPDATE [OF tablename[,…]]
      [LIMIT {count|ALL}[{OFFSET|,}start]]
```

描述

SELECT INTO 命令的语法与一个常规 SELECT 命令实质上是一样的；唯一的区别是查

询的输出是一个新表。

输入

请参见 SELECT 命令。

输出

请参见 SELECT 命令讲解中的“输出”部分。

注意

执行此命令的用户将成为新创建表的所有者。

SQL-92 兼容性

请参见 SELECT 命令讲解中的“SQL-92 兼容性”部分。

示例

只根据表 authors 中名为 Sam 的人创建新表：

```
SELECT * FROM authors WHERE name='Sam' INTO TABLE SamTable;
```

SET

用法

```
SET variable{TO|=}{value|'value'|DEFAULT}
```

或

```
SET CONSTRAINT {ALL|list}mode
```

或

```
SET TIME ZONE{'timezone'|LOCAL | DEFAULT}
```

或

```
SET TRANSACTION ISOLATION LEVEL{READ COMMITTED |SERIALIZABLE}
```

描述

实质上，SET 命令在 PostgreSQL 中用于设置一个运行变量。不过，指定的用法变量很大程度上依赖于设置的运行变量。

设置一个变量后，可以使用 SHOW 命令来显示当前的设置，还可使用 RESET 命令来恢复设置至缺省值。

输入

下面介绍有效变量的基本列表及其组合值。

CLIENT_ENCODING | NAMES

参数：*value*。

设置多位元编码，在编译时必须可用。

DATESTYPE

参数：

- ISO——使用 ISO 8601 样式的日期和时间。
- SQL——使用 Oracle/Ingres 样式的日期和时间。
- Postgre——使用标准的 PostgreSQL 样式的日期和时间。
- European——使用“天/月/年”（dd/mm/yyyy）样式日期。

■ NonEuropean——使用“月/天/年”（mm/dd/yyyy）样式日期。

■ German——使用“天.月.年”（dd.mm.yyyy）样式日期。

■ US——使用“天/月/年”（dd/mm/yyyy）样式日期（与 European 相同）。

■ DEFAULT——使用 ISO 样式的日期和时间。

设置日期和时间表示法样式。

SEED

参数：*value*。

根据指定种子设置随机数发生器（在 0 和 1 之间的浮点数）。而且可以通过 setseed 函数来设置此值。

只用 MULTIBYTE 发生作用后此选项才可用。

SERVER_ENCODING

参数：*value*。

设置一个值的多位元编码。

只用 MULTIBYTE 发生作用后此选项才可用。

CONSTRAINT

参数：*constraintlist* 和 *mode*。

控制当前事务中的约束评价级别，其中：

constraintlist——以逗号分隔的约束名称列表。

mode——IMMEDIATE 或 DEFERRED。

TIME ZONE | TIMEZONE

参数：*value*。

设置成依赖于操作系统的时区（对 Linux 类型操作系统，通过 usr/lib/zoneinfo 或 usr/share/zoneinfo 查看有效时区值）。

PG_OPTIONS

PG_OPTIONS 可以带有若干个内部优化参数，它们是：

■ All。

■ deadlock_timeout。

■ executorstats。

■ hostlookup。

■ lock_debug_oidmin。

■ lock_debug_relid。

■ lock_read_priority。

■ locks。

■ malloc。

■ nofsync。

■ notify。

■ palloc。

■ parse。

■ parserstats。

■ plan。

- plannerstats。
- Pretty-Parse。
- pretty_rewritten。
- query。
- rewritten。
- shortlocks。
- showportnumber。
- spinlocks。
- syslog。
- userlocks。
- verbose。

RANDOM_PAGE_COST

参数：*float-value*。

设置优化器对非连续磁盘存储页读取花费时间的估计值。

CPU_TUPLE_COST

参数：*float-value*。

设置优化器对查询中处理每个元组花费时间的估计值。

CPU_INDEX_TUPLE_COST

参数：*float-value*。

设置优化器对查询中处理每个索引元组花费时间的估计值。

CPU_OPERATOR_COST

参数：*float-value*。

设置优化器对查询中处理每个 WHERE 子句中的操作符所花费时间的估计值。

EFFECTIVE_CACHE_SIZE

参数：*float-value*。

设置优化器对磁盘缓冲有效大小的估计值。

ENABLE_SEQSCAN

参数：*ON*（缺省）或 *OFF*。

允许/不允许规划器使用连续扫描类型（注意：实际上此设置不可能完全关闭，但将之设为“不允许”可以在一定程度上起作用）。

ENABLE_INDEXSCAN

参数：*ON*（缺省）或 *OFF*。

允许/不允许规划器使用索引扫描。

ENABLE_TIDSCAN

参数：*ON*（缺省）或 *OFF*。

允许/不允许规划器使用 TID 扫描计划。

ENABLE_SORT

参数：*ON*（缺省）或 *OFF*。

允许/不允许规划器使用显式排序类型。（注意：实际上此设置不可能完全关闭，但将之设为“不允许”可以在一定程度上起作用。）

ENABLE_NESTLOOP

参数：*ON*（缺省）或 *OFF*。

允许/不允许规划器在联合计划中使用嵌套循环（注意：实际上此设置不可能完全关闭，但将之设为“不允许”可以在一定程度上起作用）。

ENABLE_MERGEJOIN

参数：*ON*（缺省）或 *OFF*。

允许/不允许规划器使用融合联合规划。

ENABLE_HASHJOIN

参数：*ON*（缺省）或 *OFF*。

允许/不允许规划器使用哈希联合规划。

GEQO

参数：*ON*（缺省），*ON=value* 或 *OFF*。

设置使用基因优化算法的阈值。

KSQO

参数：*ON*，*OFF*（缺省），或 *DEAULT*（*OFF*）。

设置键集查询优化，它决定是否使用 UNION 查询优化了 OR 和 AND 子句。

（即：WHERE（a=1 和 b=1）或（a>2 和 b>2））。

MAX_EXPR_DEPTH

参数：*integer*。

设置解析器能接受的最大嵌套深度。

警告

将此级别设置得太高可能导致服务器崩溃。

输出

SET VARIABLE（若成功执行则返回此消息）。

NOTICE: Bad value for variable (value)（若声明变量不能使用指定值则返回此消息）

注意

使用 SHOW 命令来显示变量的当前设置值。

SQL-92 兼容性

在 SQL-92 规范中唯一对 SET 命令所作的定义是 SET TRANSACTION ISOLATION LEVEL 和 SET TIME ZONE。除此之外的其他方面应用均属此命令在 PostgreSQL 语言中的扩展。

示例

此例将时区设置成中部时间：

```
SELECT TIME ZONE "CST6CDT";

SELECT CURRENT_TIMESTAMP As RightNow;
```

```
RightNow
----------------------
2001-08-15 09:50:23-06
```

SHOW

用法

```
SHOW variable
```

描述

SHOW 命令用于显示运行变量的当前值。它与 SET 和 RESET 命令联合使用来改变变量设置。

输入

variable——所显示的变量名。

输出

NOTICE: variable is value（若命令成功执行则返回此消息）。

NOTICE: Unrecognized variable value（如果没有发现指定的变量名则返回此消息）。

NOTICE: Time zone is unknown（若没有正确设置 TZ 或 PGTZ 变量则返回此消息）。

注意

请参考 SET 命令来显示有效变量列表。

SQL-92 兼容性

SHOW 是一个 PostgreSQL 扩展命令。SQL-92 规范中没有定义 SHOW 命令。

示例

下面例子显示了当前日期的设置样式：

```
SHOW datastyle;
NOTICE: DataStyle is ISO with US (NonEuropean) conventions
```

TRUNCATE

用法

```
TRUNCATE [TABLE] name
```

描述

TRUNCATE 命令从指定表中快速删除所有行。它与 DELETE 命令的功能一样，但执行速度更快。

输入

name——清空表的名称。

输出

TRUNCATE（若命令成功执行则返回此消息）。

注意

此命令的使用者必须为指定表的所有者或具有删除权限才能执行此命令。

SQL-92 兼容性

它是 PostgreSQL 的扩展命令；SQL-92 中达到此相同效果的方法是使用与之相关的 DELETE 命令。

示例

从表 temptable 中快速删除所有数据：

```
TRUNCATE TABLE temptable;
```

UNLISTEN

用法

```
UNLISTEN {notifyname | *}
```

描述

UNLISTEN 命令用于终止一个前端待执行的 LISTEN 命令。

终止监听的具体名称可以明确指定，或用一个通配符（*）来指定，它将终止所有先前注册名的监听。

输入

notifyname —终止监听的先前注册名称。

*—终止监听所有先前注册名称。

输出

UNLISTEN（若成功执行则返回此消息）。

注意

一旦注销，服务器后面发送的 NOTIFY 命令将被忽略。

SQL-92 兼容性

UNLISTEN 命令是 PostgreSQL 的扩展。在 SQL-92 规范中没有此命令。

示例

此例显示了注册监听名称 mynotify、发送通知及注销此名称的命令代码：

```
LISTEN mynotify;
NOTIFY mynotify;
Asynchronous NOTIFY 'mynotify' from backend with pid '7277' received
UNLISTEN mynotify;
NOTIFY mynotify;
```

UPDATE

用法

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

描述

UPDATE 命令用于更改表中指定行内的数据。如果没有指定 WHERE 条件句，则假定所有行；否则，只更改匹配 WHERE 条件的行。

通过 FROM 关键字，可以让多个表满足 WHERE 条件。

输入

table ——更新表名称。

column——更改数据的指定列。

expression——更改数据的指定值或有效表达式。

fromlist——包含于 WHERE 条件中的可选表列表。

condition——用于约束更改操作的标准 SQL WHERE 条件。（请参见 SELECT 命令获取有关 WHERE 条件的更多信息。）

输出

UPDATE #（若成功执行则返回此消息。输出信息中包含了更改了数据的行数。）

注意

执行 UPDATE 命令的用户必须具有对指定表的写权限，同时还须具有对 WHERE 子句中所需表的选择权限。

SQL-92 兼容性

UPDATE 命令基本与 SQL-92 规范兼容，只要以下几点除外：

FROM fromlist——PostgreSQL 允许多表满足 WHERE 条件。但在 SQL-92 中并不允许。

WHERE CURENT OF cursor——SQL-92 允许基于打开的游标定位更新数据。而这在 PostgreSQL 中是不允许的。

示例

下面例子对表 authors 中所有名为 Bill 的人将列 status 更改为 actiive：

```
UPDATE authors SET status='active' WHERE name='Bill';
```

VACUUM

用法

```
VACUUM[VERBOSE][ANALYZE][table [(column [,…])]]
```

描述：

VACUUM 命令的目的有两个：一个是回收浪费的磁盘空间，二是进行 PostgreSQL 的优化操作。

当 VACUUM 命令运行时，当前数据库中的所有类都被打开，而且回滚事务中的所有旧记录都被清除。另外，系统目录表根据对每一个类的优化统计信息进行更新。而且，如果加上命令 ANALYZE，则列数据的离差信息将被更新以改进查询执行的路径。

输入

VERBOSE——对每个表显示一个详尽的报告。

ANALYZE——对每个表更新列的统计信息。此信息用于查询优化以规划最有效的搜索方式。

table——清理的表名。缺省时为所有表。

column——分析的列名称。缺省时为所有列。

输出

VACUUM（若命令成功执行则返回此消息）。

NOTICE: - Relation 'table' -（指定表的报告头）

NOTICE: Pages XX, Changed XX, Reapped XX, Empty XX, New XX; Tup XXXX: Vac XXXX, Crash XX, Unused XX, MinLen XXX, MaxLen XXX; Re-using: Free/Avail. Space XXXXXXX/XXXXXXX; EndEmpty/Avail. Page X/XX. Elapsed X/X sec（返回分析表信息）

NOTICE: Index 'name': Pages XX; Tuples XXXX: Deleted XXXX. Elapsed X/X sec（返回索引的分析报告）

注意

当前打开的数据库是缺省的清理目标。

VACUUM 是定期运行 cron 任务的理想命令之一。对于在 psql 或其他前端应用程序外部运行此命令的知识，请参考第6章"用户可执行文件"中"VACUUMdb"部分的有关VACUUMdb命令介绍。

对一个数据库进行有效删除或修改后运行 **VACUUM ANALYZE** 命令。

SQL-92 兼容性

在 SQL-92 中没有 VACUUM 语句；它是 PostgreSQL 的一个扩展命令。

示例

下面例子将清理表 authors：

```
VACUUM VERBOSE ANALYZE authors;

NOTICE:--Relation authors--
NOTICE:Pages 10:Changed 6,reaped 10,Empty 0,New 0,Tup 1037:Vac 54,Keep/VTL
0/0 Crash 0,UnUsed 0,MinLen 64,MaxLen64;Re-using:Free/Avail.Space
11108/3608;EndEmpty/Avil Pages 0/9.CPU 0.00s/0.0 |u sec
NOTICE:Index name_idx:Pages 9;Tuples 1037:Deleted 54. CPU 0.00s/0.01u sec
NOTICE:Index ssn_idx:Pages 5; Tuples 1037:Deleted 54. CPU 0.00s/0.00u sec
Notice Rel authors:Pages:10→9;Tuples moved:46.CPU 0.00s/0.0|u sec
NOTICE:Index name_idx:Pages 10;Tuples 1037:Deleted 46. CPU 0.00s/0.01u sec
NOTICE:Index ssn_idx:Pages 5; Tuples 1037:Deleted 46. CPU 0.00s/0.00u sec
VACUUM
```

第二部分

PostgreSQL 规范

第 2 章　PostgreSQL 数据类型

数据类型是所有 RDBMS（关系数据库管理系统）的基石。它们提供了现代数据库高级别功能所需要的整套机制。查询、数据验证、比较操作符、操作函数等等只是数据类型概念的扩展而已。

没有数据类型，要想从一个数据库中准确获取有意义的信息将变得极其困难。数据类型确保了列级中的数据以相一致的格式保存。数据类型使我们按计划方式对处理数据进行比较和操纵变得容易。失去了一致性，处理数据间的比较就如同苹果和橙子间的比较一样（或者字符串与整数相比一样）。

数据类型的另一个好处是它有利于整体操作性能和数据库系统效率的提高。因为数据库知道了保存在指定列中的数据类型，所以它就可以推断如何保存和取回此数据最有效。

2.1　数据类型表

表 2-1 列出了 PostgreSQL 的内置数据类型，根据所表达的数据类型进行排列。你如果知道将要保存的数据类型，但不能确定 PostgreSQL 对此数据类型的规定，此表就会大有用途。

表 2-1　　PostgreSQL 的内置数据类型

数据种类	PostgreSQL 数据类型（别名）	描　述
几何类型	BOX	描述矩形的坐标
	CIRCLE	描述圆的坐标
	LINE	描述一条无限长直线的坐标
	LSEG	描述一条线段的坐标
	PATH	描述一系列点的坐标
	POINT	描述单独一个点的坐标
	POLYGON	描述一个多边形的坐标
逻辑类型	BOOLEAN	真或假布尔值
网络类型	CIDR	IP/网络片段大小
	INET	IP 地址和网络掩码值
	MACADDR	以太网 MAC 地址
数字类型*	GIGINT 或 INT8	整数—1×10^18
	DECIMAL 或 NUMERIC	用户自定义精度和十进制舍入
	DOUBLE PRECISION 或 FLOAT8	浮点值—15 位
	FLOAT	与 FLOAT8 或 DOUBLE PRECISION 相同
	INTEGER 或 INT4	整数—范围为±2,147,483,648
	REAL 或 FLOAT4	浮点数—6 位精度
	SERIAL	整数—0 到+2147483647
	SMALLINT 或 INT2	整数—范围为±32,768

续表

数据种类	PostgreSQL 数据类型（别名）	描　述
字符串类型	CHAR 或 CHARACTER TEXT VARCHAR 或 CHARACTER VARYING	定长字符串，用空白补足 变长字符串 变长字符串，有长度限制
时间类型	DATE INTERVAL TIME TIME WITH TIME ZONE TIMESTAMP	从 4713 BC 到 32767 AD 的某一天 ±178,000,000 年的时间段 一天中从 00:00:00 到 23:59:59 的某一时刻 一天及其时区从 00:00:00+12 到 23:59:59-12 的某一时刻 从 4713 BC 到 1465001 AD 的某一日期/时间
其他类型	BIT BIT VARYING MONEY NAME OID	定长二元数据类型 向右补足 变长（指定了最大值）二元数据类型 固定精度数字，范围±21474836.48（不再使用——用 FLOAT 或 NUMBERIC（x,2）取而代之） 用于系统命名的内部数据类型 PostgreSQL 对象标识符

*一些数字类别的数据类型各仅在 PostgreSQL 7.1 中支持。例如，版本 6.5 并不支持 GEGIN，但支持 INT8。

下面内容将列出 PostgreSQL 的内置数据类型。根据数据类型的定义分开介绍。每种元素均提供了相关描述信息，如保存大小、值范围、兼容性和描述性注释等。

2.2 几何数据类型

PostgreSQL 包括许多表示几何关系的内置数据类型。同时 PostgreSQL 有许多内置函数和操作符用于操作和计算对应的几何数据。

BOX

描述

表示定义矩形的外角坐标。

输入

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

x1——X 轴起点。

y1——Y 轴起点。

x2——X 轴终点。

y2——Y 轴终点。

保存大小

32 字节。

数据举例

((1,1),(50,50))

注意

输入时，数据将被重新排列首先保存右下角坐标，再保存左上角坐标。所以在前一个例子中，保存为(50,50),(1,1)。

CIRCLE

描述

表示一个圆的圆心及半径的坐标。

输入

<(x,y),r>

x——X 轴中心点。

y——Y 轴中心点。

r——半径。

保存大小

24 字节。

数据举例

<(10,10),5>

LINE

描述

表示一条直线的坐标。

输入

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

x1,y1——X 轴、Y 轴起点。

x2,y2——X 轴、Y 轴终点。

保存大小

32 字节。

数据举例

((1,1),(100,100))

注意

从 PostgreSQL 7.1 开始引入直线（LINE）数据类型。

LSEG

描述

表示一个有限线段的坐标。

输入

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

x1,y1——X 轴、Y 轴起点。

x2,y2——X 轴、Y 轴终点。

保存大小

32 字节。

数据举例

((1,1),(100,100))

注意

线段（LSEG）数据类型与直线（LINE）数据相似，不同的是，后者代表一个无限长度的直线而不是一定长度的线段。实际上，通过 LINE 数据类型定义一条直线后，此直线假定可以在同一平面上无限延伸。

PATH

描述

路径代表不同的线段，其中包含了无数点，它创建一个开放或封闭的路径。

输入

((x1,y1),…,(xn,yn))

x1,y1——表示封闭路径的起点。

xn,yn——表示封闭路径的终点。

[(x1,y1),…,(xn,yn)]

x1,y1——表示开放路径的起点。

xn,yn——表示开放路径的终点。

保存大小

4+32n 字节。

数据举例

[(1,1),(3,3),(5,10)]——开放路径。

((1,1),(3,3),(5,10),(1,1))——封闭路径。

注意

封闭路径以一个开放的圆括号开始而开放路径以一个开放的方括号开始。

可用函数 isopen、isclosed、popen 和 pclose 测试和操纵路径。

POINT

描述

表示空间中的一个点的坐标。

输入

(x,y)

x——点的 X 轴。

y——点的 Y 轴。

保存大小

16 字节。

数据举例

(1,5)

POLYGON

描述

表示一个多边形的坐标集。

输入

((x1,y1),…,(xn,yn))

x1,y1——多边形的起点。

xn,yn——多边形的终点（通常与起点重合）。

保存大小

4+32n 字节。

数据举例

((1,1),(5,1),(5,5),(1,5),(1,1))——一个正方多边形。

注意

一个多边形与一个封闭路径很相似。不过，一些函数只作用于多边形（它们是：poly_center、poly_contain、poly_left、poly_right 等）。

2.3 逻辑数据类型

逻辑数据类型用于表示真（true）、假（false）或空（NULL）的概念。通常，此数据类型主要用于指示当前记录状态的标记。值 true 和 false 为自解释性，但值 NULL 通常是“不知道”的同义词。

BOOLEAN

描述

表示一个逻辑真、逻辑假或空值。

输入

True, t, true, y, yes, 1——所有有效真值。

FALSE, f, false, n, no, 0——所有有效假值。

NULL——有效的空值。

保存大小

1 字节。

注意

一般，最后对布尔数据使用 TRUE 和 FALSE 输入形式。这是 SQL 兼容的格式，也较通常，尽管一些 RDBMS 用 1 和 0 代表 TRUE 和 FALSE。一些输入值需要用单引号作为转义符（即：'t'）。不过，与 SQL 兼容的 TRUE 和 FALSE 形式不需要使用引号。

2.4 网络数据类型

在众多 SQL 系统中，PostgreSQL 是唯一的包含针对网络地址内置数据类型的数据库系统。CIDR、INET 和 MACADDR 表示网络地址的各个方面。当使用 PostgreSQL 作为一个 web 应用后端数据库时，这些数据类型就会非常有用。

当 PostgreSQL 中的函数处理网络类数据时，它会优先将网络值保存于网络数据类型中。

CIDR

描述

表示 IP 地址的 Dotted-quad 形式数据和网络掩码中的比特数。此数据类型遵从 CIDR（Classless Internet Domain Routing）约定。

输入

x.x.x.x/y

x.x.x.x——有效 IP 地址。

y——网络掩码中的位。

保存大小

12 字节。

数据举例

192.168.0.1/24

128.1(假定为 128.1.0.0/16)

10(假定为 10.0.0.0/8)

注意

如果忽略网络掩码中的比特值，则认为网络掩码使用了 Dotted-quad 类（例如 255.0.0.0 被认为是 8，255.255.0.0 被认为是 16，255.255.255.0 被认为是 24 等等）。不过，这种假设足以应付所有八位字节地址了。

注意，PostgreSQL 不支持 IPv6。

INET

描述

表示 IP 地址的 Dotted-quad 形式数据和一个可选网络掩码。

输入

x.x.x.x/y

x.x.x.x——有效 IP 地址。

y——如果给出，则为网络掩码；否则认为是主机。

保存大小

12 字节。

数据举例

192.168.0.1（假定为 192.168.0.1/32）

注意

INET 与 CIDR 之间的区别是 INET 可以指一个单主机，而 CIDR 指一个 IP 网络。

MACADDR

描述

表示一个 MAC 地址，它是一个以太网硬件地址。

输入

支持以下几种不同的格式：

xxxxxx:xxxxxx

xxxxxx-xxxxxx

xxxx.xxxx.xxxx

xx-xx-xx-xx-xx-xx

xx:xx:xx:xx:xx:xx（缺省）

保存大小

6 字节。

数据举例

下面两者均表示相同的 MAC 地址：

08-00-2d-01-32-22

08002d:01322

注意

目录$SOURCE/contrib/mac 中包含了根据给定 MAC 地址识别指定网卡制造商的工具。（只有版本 7.1 中才有此目录。）

2.5　数字数据类型

数字数据类型用于保存与数字相关的数据。7.X 系列版本对此作了一些改进。也就是说，PostgreSQL 现在使用了一个更具描述性的数字数据类型命名约定（例如，BIGINT 对应于 INT8）。

近来，在最新的几个版本中，某些数据类型已不推荐使用（如不再推荐使用 MONEY 数据类型，而建议使用 DECIMAL 数据类型）。

BIGINT（或 INT8）

描述

表示一个非常大的整数。

输入

大整数（大约 1×10^18）

保存大小

8 字节。

注意

PostgreSQL7.1 以前版本可能将此数据类型称之为 INT8。

DECIMAL（或 NUMERIC）

描述

表示一个指定了小数位数的用户定义长度数。

输入

(x,y)

x——总长度。

y——小数位。

保存大小

8 字节。

注意

PostgreSQL 7.1 以前版本可能将此数据类型称之为 NUMERIC。

DOUBLE PRECISION（或 FLOAT8）

描述

表示一个大浮点数。

输入

不同精度——15 个小数位。

保存大小

8 字节。

注意

PostgreSQL 7.1 以前版本可能将此数据类型称之为 FLOAT8。

INTEGER（或 INT4）

描述

表示一个整数。

输入

范围从-2,147,483,648 到+2,147,483,648

保存大小

4 字节。

注意

PostgreSQL 7.1 以前版本可能将此数据类型称之为 INT4。

REAL（或 FLOAT4）

描述

表示一个标准浮点数。

输入

最多 6 个小数位的不同精度。

保存大小

4 字节。

注意

PostgreSQL 7.1 以前版本可能将此数据类型称之为 FLOAT4。

SERIAL

描述

表示一个整数，用于自增字段。

输入

从 0 到+2,147,483,647

保存大小

4 字节

注意

SERIAL 数据类型实际上是一个具有一些附加特征的标准整数（INTEGER）类型：SERIAL 数据类型是一个带有在指定列上自动创建序列和索引的整数。当删除一个包含 SERIAL 类型数据的表时，相关的序列也必须要显式删除，这不是自动进行的。

SMALLINT（或 INT2）

描述

表示一个小整数。

输入

从-32,768 到+32,768

保存大小

2 字节。

注意

PostgreSQL 7.1 以前版本可能将此数据类型称之为 INT2。

2.6 字符串数据类型

PostgreSQL 包含 3 个保存字符串相关数据的基本数据类型。与 SQL 标准兼容的类型中，有定长和变长字符串类型。而且，PostgreSQL 还定义了更一般的数据类型，这就是 TEXT，它需要指定上限来限制最大值。不过，这一数据类型仅存在于PostgreSQL定义中，它与SQL-92标准并不兼容。

CHAR（或 CHARACTER）

描述

表示一个指定了最大长度的定长字符串。

输入

CHAR(n)

n——字符串的整数长度值（如忽略则认为是 1）。

保存大小

4+n 字节。

注意

CHAR 是一个与 SQL-92 兼容的数据类型。数据中没有填满指定极限的部分由空白代替。

TEXT

描述

表示一个变长字符串。

保存大小

可变——取决于字符串的长度，但不能小于 4 个字节。

VARCHAR（或 CHARACTER VARYING）

描述

表示一个指定了极限长度的变长字符串。

输入

VARCHAR(n)

n—字符串的整数长度值（如忽略则认为是 1）。

保存大小

可变——取决于字符串的长度，但不能小于 4 个字节。

注意

VARCHAR 和 CHARACTER VARYING 都是与 SQL-92 兼容的数据类型。

2.7 时间数据类型

PostgreSQL 中包含许多用于处理时间和日期相关数据的特别被设计的内置数据类型。一些内置常量对于简化时间项很有用。如下所示：

now——存储时保存一个时间戳的常量。

Today——表示当天午夜的常量。

Tomorrow——表示次日午夜的常量。

Yesterday——表示昨天午夜的常量。

PostgreSQL 在事务开始时对常量求值，这样可以导致不期望的动作。例如，在一个事务中插入一系列的 now 常量就会导致所有行具有相同的时间戳。一个可行的办法是使用 now()函数，此函数在每次调用时求常量值，而不是在事务创建期间求值。

DATE

描述

表示一个描述某特定日子的值。它支持许多不同的输入格式（请参阅下面部分）。

输入

从 4713 BC 到 32767 AD。

可能的输入格式有：

Jun 22,1971—标准日期格式。

June 22,200 BC—指定纪元。

1971-06-22—ISO 格式（yyyy-mm-dd）。

6/22/1971—美国样式。

22/6/1971—欧洲样式（对美国样式无效）。

19710622 或 710622—ISO 格式(yyyymmdd 或 yymmdd)。

1971.174 或 71.174—年及年中某一天。

保存大小

4 字节。

注意

有效的月份格式及其缩写：

January	Jan
February	Feb
March	Mar
April	Apr
May	May
June	Jun
July	Jul
August	Aug

September	Sep 或 Sept
October	Oct
November	Nov
December	Dec

有效的星期日数及其缩写：

Monday	Mon
Tuesday	Tue 或 Tues
Wednesday	Wed 或 Weds
Thursday	Thu，Thur 或 Thurs
Friday	Fri
Saturday	Sat
Sunday	Sun

上面描述的是其输入格式；输出格式由 DATESTYLE 变量指定（请参见 SQL 的 SET 命令）。

INTERVAL

描述

表示一个时间间隔值。

输入

INTERVAL 的输入格式如下：

Qnt unit [Qnt Unit …]Direction

有效的 *Qnt* 值有：

-2147483648 到+2147483648

有效的 *Unit* 值有（复数形式同样有效）：

Second
Minute
Hour
Day
Week
Month
Year
Decade
Century
Millennium

有效的 *Direction* 值有：

Ago—过去时。
[blank]—将来时。

保存大小

12 字节。

数据举例

1 Week Ago

5 Years 3 Months Ago

30 Days

注意

INTERVAL 能精确到 0.000001 秒（1 微秒）。

TIME

描述

表示一个基于时间值的实体。

输入

TIME 的有效范围是从 00:00:00.00 到 23:59:^59. ^99。

TIME 的有效输入格式有：

08:24—ISO 格式

08:24:50—ISO 格式

08:24:50.15—ISO 格式

082450—ISO 格式

08:24 PM—标准

20:24—24 小时格式

z—与 00:00:00 同

zulu—与 00:00:00 同

保存大小

4 字节。

注意

TIME 数据类型是一个与 SQL 兼容的格式。TIME 数据类型可精确到 0.000001 秒（1 微秒）。

TIME WITH TIME ZONE

描述

表示一个基于时间值且包含时区信息的实体。

输入

TIME WITH TIME ZONE 的有效范围是从 00:00:00.00+12 到 23:59: ^59.99 -12。

TIMEWITH TIME 20NE 的有效输入格式有：

08:24-6—ISO 格式

08:24:50-6—ISO 格式

08:24:50.15-6—ISO 格式

082450-6—ISO 格式

保存大小

4 字节。

注意

TIME WITH TIME ZONE 可以接受任何对 TIME 数据类型有效的时间值的输入，但其末尾还要附加除时区之外的信息。

TIME 数据类型是与 SQL 兼容的格式。TIME WITH TIME ZONE 数据类型可精确到 0.000001 秒（1 微秒）。

TIMESTAMP

描述

表示一个代表时间和日期信息的值。

输入

TIMESTAMP 的有效范围是从 4713-01-01 00:00:00:00 BC 到 1465001-12-31 23:59: ^59.99 AD。

TIMESTAMP 的有效输入格式有：

```
Date Time [Era] [Time Zone]
```

例如：

```
2001-11-24 08:23:11—标准时间戳。
2001-11-24 08:23:11 AD -6:00—带纪元和时区的时间戳。
November 11,2001 08:23:11—Prose 样式时间戳。
```

保存大小

8 字节。

注意

因为此数据类型包含时间、日期、纪元和时区信息，所以 TIMESTAMP 成为一个保存临时元素的流行数据类型。

2.8 其他数据类型

下面是一些不常用的不同数据类型。一般，它们仅用于内部系统目标中，但有时你也可能遇到此类数据类型。下面列出这些数据类型供必要时参考。

BIT 和 BIT VARYING

BIT 数据类型保存一系列的 1 和 0 二进制值。BIT 数据类型有一个指定的宽度并用 0 来填充空位，但 BIT VARYING 数据类型允许使用灵活宽度的实体。

MONEY

PostgreSQL 仍然支持货币（MONEY）数据类型，但不再属于常用数据类型。而代而替之用一个设置了小数位的 NUMERIC 或 DECIMAL 数据类型。

NAME

NAME 数据类型保存一个 31 个字符串，但它仅供内部使用。PostgreSQL 使用 NAME 数据类型来保存内部系统目录信息。

OID

OID 数据类型是一个从 0 到 40 亿的整数。PostgreSQL 中创建的每个对象都隐式分配了一个 OID。OID 对于保持数据的完整性很有用，因为每个对象的 OID 在数据库中是唯一的。

缺省时，OID 不可见，但可以通过查询语句明确指定选择和显示 OID。如：

```
SELECT * FROM test;
Name      Age
-----------
Bill      34
Ann       22

SELECT oid, * FROM test;

oid       Name        Age
------------------------
19278     Bill        34
19279     Ann         22
```

通常，OID 用于确保数据的完整性而不是创建显式序列。不过，常由于许多原因并不建议如此。整个数据库中 OID 贯穿所有对象，所以，它们在一个给定的表中并不是连续的。

而且，一旦 OID 序列达到了其上限，它就会从 0 重新开始（或另外的指定最小值）。尽管序列可以完成相同的功能，但具有唯一性的 OID 更强大，因为它们遍布于整个数据库中。

除此之外，当构建一个数据库驱动应用程序时，很多情况下需要知道尚未实际提交使用的下一个序列值。正因为此，查询一个单序列的下一个值比试图猜测 OID 的下一个值更为可靠。

2.9　更多的数据类型

下面列出了 PostgreSQL 中更加少用的数据类型（一些类型仅仅是前面讲过类型的别名）：

abstime——表示绝对系统（UNIX）时间。

aclitemv——表示一个访问控制列表项目。

bpchar——表示一个用空白填充的定长字符串。

bytea——表示一个变长二进制值。

cid——表示一个命令标识类型并在事务中使用。

filename——表示一个文件名并在系统表中使用。

int2vectory——表示一个 16 INT2 类型值的数组。

lztext——表示变长文本并压缩保存。

oidvector——表示一个 16 OID 数组并在系统表中使用。

regproc——表示一个已注册过程。

reltime——表示相对时间（UNIX delta）

timetz——表示一个 ANSI SQL 时间（hh:mm:ss）。

tinterval——表示一个（绝对时间，绝对时间）数组。

varbit——表示一个定长位值。

xid——表示一个事务 ID 序列。

第 3 章　PostgreSQL 操作符

数据类型本身仅对保存数据有用。要进行数据间的比较、排序及选择表数据就需要使用操作符。

大部分操作符对比较标准简单地返回一个隐式布尔值（真或假）。但也有一些操作符，如与数学和字符串类操作符，它们根据给定的元素返回新的结果。

表 3-1 是根据数据类型分组的缺省 PostgreSQL 操作符图表。然后，再更详细地介绍 PostgreSQL 所支持的操作符，包括它们的特定用法、语法和注释等信息。

表 3-1　　操作符图表（按数据类型分组）

数据类型	操作符
几何型	+
	-
	*
	/
	#
	##
	&&
	&<
	&>
	<->
	<<
	<^
	>>
	>^
	?#
	?-
	?-|
	@-@
	?|
	?||
	\@
	@@
	~=
逻辑型	AND
	OR
	NOT

续表

数据类型	操作符
网络型	< <= = >= > <>> < <<= >> >>=
数字型	! !! % * + - / : @ ^ |/ ||/
字符串型	< > = > >= || !!= ~~ !~~ ~ ~* !~ !~*
时间型	#< #<= #<>> #= #> #>= <#> > | -= ><?>

3.1 几何类操作符

PostgreSQL 包含许多有助于比较几何数据类型间关系的操作符。

其中的一些操作符返回隐式布尔值（如<<和>>），而另外一些根据输入元素提供新结果（像数学操作符一样，如+和-）。

列表

+	将几何对象向右平移（如：point '(2,0)' + point '(0,1)'）。
-	将几何对象向左平移（如：point '(2,0)' - point '(0,1)'）。
*	伸缩/旋转（顺时针）。
/	伸缩/旋转（逆时针）。
#	交集。
#	多边形点数。
##	最近点。
&&	重叠。
&<	与左边重叠。
&>	与右边重叠。
<->	相距。
<<	左边于。
<^	下方于。
>>	右边于。
>^	上方于。
? #	相交或重叠。
? -	平行线。
? -|	垂直线。
@-@	周长。
? |	垂直于。
? ||	平行于。
@	包含或位于其上。
@@	取圆心。
~=	相同。

注释/示例

在矩形上添加一个点：

```
box '((0,0),(1,1))' * poinit '(2,0)'
```

选择位于指定矩形左边的所有矩形：

```
SELECT * FROM map WHERE the box<< '((5,5),(4,4))';
```

选择与指定线段平行的所有直线：

```
SELECT * FROM map WHERE the lines ?|| lseg '((1,1),(3,3))';
```

3.2 逻辑类操作符

逻辑类操作符通常用于与表达式一起从列表中得到一个聚集布尔值。

列表

AND　作用同一个逻辑“与”判断连接符。
OR　作用同一个逻辑“或”判断连接符。
NOT　作用同一个“非”符。

注释/示例

```
SELECT * FROM payroll WHERE firstname="Bill" AND lastname="Smith";
SELECT * FROM payroll WHERE firstname="Bill" OR firstname="Sam";
SELECT * FROM payroll WHERE firstname IS NOT NULL;
```

3.3 网络类操作符

在 PostgreSQL 中加入网络类操作符是为了在两个 IP 地址间进行比较。这些操作符同样对 INET 和 CIDR 数据类型起作用。

列表

<　小于。
<=　小于或等于。
=　等于。
>=　大于或等于。
>　大于。
<>　不等于。
<<　包含于。
<<=　包含于或等于。
>>　包含。
>>=　包含或等于。

注释/示例

使用下面命令查找小于指定地址的所有 IP 地址：

```
SELECT * FROM computers WHERE ipaddr<'192.168.0.100'
```

类似地，使用下面命令查找指定子网上的所有 IP 地址：

```
SELECT * FROM computers WHERE ipaddr<<'192.168.0.1/24'
```

3.4 数字类操作符

数字类操作符根据给定元素返回一个新值。

列表

!	阶乘。
!!	阶乘（左操作符）。
%	取模。
*	乘。
+	加。
-	减。
/	除。
:	自然幂。
@	绝对值。
^	求幂。
|/	平方根。
||/	立方根。
&	二元与。
|	二元或。
#	二元异或。
~	二元非。
<<	二元左位移。
>>	二元右位移。

注释/示例

表 3-2 是一些数字操作符的实际运用例子：

表 3-2　　数字操作符实例

操作符	给定值	结果
加	5+5	10
平方根	|/81	9
绝对值	@-8	8
立方根	||/343	7
阶乘	3!	6
左阶乘	!3	6
二元 AND	1&5	1
二元 XOR	1#5	4
二元左移位	1<<4	16

二元操作符同样可作用于 BIT 和 BIT VARYING 数据类型。如表 3-3 中例子。

表 3-3　　二元操作符作用于 BIT 和 BIT VARYING 数据类型

操作符	给定值	结果
二元 AND	B'10001'&B'01101'	B'00001'
二元 NOT	~B'01110'	B'10001'

3.5 字符串操作符

实际上，所有的字符串操作符根据给定的比较式返回隐式布尔值真或假。（例外的情况是在下一个“列表”部分显示的串联操作符。）

进行比较时，要考虑字符在 ANSI 图表中的位置。所以，小写的“a”被认为小于大写的“A”。

字符串操作符使用规则表达式匹配。表达式匹配包括一个内部格式和一个 POSIX 兼容格式。

PostgreSQL 可利用两种独特的模式匹配类型：一个是 ANSI-SQL 方法；一个是 POSIX 的 regex 类型。ANSI-SQL 类型所用关键字是 LIKE 和 NOT LIKE。ANSI-SQL 方法可使用以下几种通配符用于模式匹配（见表 3-4）：

表 3-4　　用于模式匹配的通配符

通配符	意义
%	任何匹配字符
-	任何单个字符

与之相反，POSIX 兼容操作符使用标准 regex 比较式，如表 3-5 下所示：

表 3-5　　POSIX 兼容操作符

POSIX regex 符号	意义
.	单个字符匹配
*	1 个或多个字符的任意字符串
+	序列的重复
?	一个序列的可能重复
[]	包含在方括号中的单个字符列表。匹配任意单个列表中的字符
^[]	包含在方括号中的单个字符列表。拒绝匹配任意单个列表中的字符
...etc...	

对 POSIX 类型规则表达式的全面讨论超出了本书的范围。请参阅 POSIX 类型 regex 的指南了解有关 sed、awk 和 egrep 的更多信息。

大部分 PostgreSQL 版本中包含的 regex 引擎为 POSIX 1003.2 版本的 egrep 类型。许多流行应用程序中均集成了 Henry Spencer 的 regex 库。在源目录$source/backend/regex 中可以找到对应 PostgreSQL 版本中所集成 regex 引擎的更多信息。

列表

<	小于。
<=	小于或等于。
<>	不等于。
=	等于。
>	大于。
>=	大于或等于。
||	串联字符串。
!!=	不相似。
~~	相似。
LIKE	相似。
ILIKE	相似（与大小写无关）。
NOT ILIKE	不相似（与大小写无关）。
NOT LIKE	不相似。
!--	不相似。
~	使用 regex 库进行匹配（大小写敏感）。
~*	使用 regex 库进行匹配（大小写无关）。
!~	使用 regex 库不匹配（大小写敏感）。
!~*	使用 regex 库不匹配（大小写无关）。

注释/示例

从一个表中选择名字为 Bob 的所有记录：

```
SELECT * FROM authors WHERE firstname='Bob';
```

从一个表中选择名字不为 Bob 的所有记录：

```
SELECT * FROM authors WHERE firstname<>'Bob';
```

选择名字以 Bo 开始的所有记录：

```
SELECT * FROM authors WHERE firstname LIKE 'Bo';
```

选择不计大小写且名字以 b 开始的所有记录：

```
SELECT * FROM authors WHERE firstname ILIKE 'b';
```

3.6　时间操作符

时间操作符用于比较时间值且通常返回一个布尔值真或假。

列表

<	时间间隔小于。
<=	时间间隔小于或等于。
<>	时间间隔不等于。

=	时间间隔等于。
>	时间间隔大于。
>=	时间间隔大于或等于。
|	时间间隔起点。

第 4 章　PostgreSQL 函数

PostgreSQL 包含诸多内置函数，它们用于处理指定的数据类型并且返回相应的值。表 4-1 是按种类分组排列的内置函数表。

4.1　函数表（按类别分组）

表 4-1　　函数表（按类别分组）

类　别	函　数
聚集函数	AVG
	COUNT
	MAX
	MIN
	STDDEV
	SUM
	VARIANCE
转换函数	CAST
	TO_CHAR
	TO_DATE
	TO_NUMBER
	TO_TIMESTAMP
几何函数	AREA
	BOX
	CENTER
	CIRCLE
	DIAMETER
	HEIGHT
	ISCLOSED
	ISOPEN
	LENGTH
	LSEG
	NPOINT
	PATH
	PCLOSE
	POINT
	POLYGON
	POPEN
	RADIUS
	WIDTH

续表

类　别	函　　数
网络函数	ABBREV BROADCAST HOST MASKLEN NETMASK NETWORK TEXT TRUNC
数字函数	ABS ACOS ASIN ATAN ATAN2 CBRT CEIL COS COT DEGREES EXP FLOOR LN LOG PI POW 或 POWER RADIANS RANDOM ROUND SIN SQRT TAN TRUNC
SQL 函数	CAST WHEN COALESCE NULLIF
字符串函数	ASCII CHR INITCAP LENGTH, CHAR_LENGTH 或 CHARACTER_LENGTH LOWER LPAD LTRIM OCTET_LENGTH POSITION STRPOS RPAD RTRIM SUBSTRING SUBSTR TRANSLATE TRIM UPPER

续表

类　别	函　　数
时间函数	AGE CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP DATE_PART DATE_TRUNC EXTRACT ISFINITE NOW TIMEOFDAY TIMESTAMP
用户函数	CURRENT_USER SESSION_USER USER
其他函数	ARRAY_DIMS

4.2　聚集函数

PostgreSQL 包含许多聚集函数。一般，聚集函数从一套给定的输入值里计算一个单一返回值。

它与标准函数形成对比的是，一般标准函数对每个给定的输入值返回一个输出值。

AVG

描述

AVG 函数返回输入列或表达式的平均值。

输入

```
AVG(col | expression)
```

示例

从表 payroll 中返回平均工资：

```
SELECT AVG(salary)FROM payroll;
```

下面例子显示了如何在 AVG 函数中使用表达式。其结果是，返回雇员薪水高于$18,000 的工资平均值（请注意，按此标准则不将收入少于$18,000 的雇员计入其中）。

```
SELECT AVG(salary-18000)FROM payroll WHERE salary>18000;
```

注意

AVG 函数支持下列数据类型：smallint、integer、bigint、real、double precision、nemeric 和 interval。

对于任何整数值（即 bigint、integer 等）返回一个 integer 数据类型。对于任何浮点值，则对应返回一个 numeric 数据类型。

对于其他数据类型，如 interval，则返回同类数据类型。

COUNT

描述

COUNT 函数计算行和表达式的数量，返回一个非 NULL 值。

输入

COUNT(*)—计算所有行。

COUNT(*col* | *expression*) —计算指定列或表达式。

示例

```
SELECT COUNT(*)AS Num_Active FROM payroll WHERE status="active ";
```

MAX

描述

MAX 函数从传递给它的一个列或表达式列表中返回其中的最大值。

输入

```
MAX(col | expression)
```

示例

```
SELECT MAX(salary) FROM payroll;
```

MIN

描述

MIN 函数从传递给它的一个列或表达式列表中返回其中的最小值。

输入

```
MIN(col | expression)
```

示例

```
SELECT MIN(salary) FROM payroll;
```

STDDEV

描述

STDDEV 函数返回给定列或表达式列表的标准采样方差（标准差）。

输入

```
STDDEV(col | stddev)
```

示例

```
SELECT STDDEV(price) FROM stocks;
```

注意

STDDEV 函数支持以下数据类型：smallint、integer、bigint、real、double precision 和 nemeric。

SUM

描述

SUM 函数返回所传递的所有列或表达式值的聚集总数。

输入

```
SUM(col | expression)
```

示例

```
SELECT SUM(salary) FROM payroll WHERE checkdate='06-01-2001';
```

注意

SUM 函数支持以下数据类型：smallint、integer、bigint、real、double precision、nemeric 和 interval。

VARIANCE

描述

VARIANCE 函数返回给定列或表达式列表的标准方差的平方值（采样方差）。

输入

```
VARIANCE (col | expression)
```

示例

```
SELECT VARIANCE(price) FROM stocks;
```

4.3 转换函数

PostgreSQL 中包含许多转换函数。它们用于将数据从一种类型转换成另一种类型并且格式化指定的输出样式。

CAST

描述

CAST 函数用于将数据从一种类型转换成另一种类型。一般而言，CAST 是一个相当普通和易用的、适用于大部分数据转换的函数。

输入

```
CAST(value AS newtype)
```

Value——需要转换的值。

Newtype——转换成的新数据类型。

示例

```
CAST('57' as INT) → 57
CAST(57 as CHAR) → '57'
CAST(57 as NUMERIC(4,2)) → 57.00
```

```
CAST('05-23-87' as DATE) → 1987-05-23
```

注意

另一个完成类型转换的方法是将值与所期望的数据类型用双冒号（::）分开。

上面的例子用此法表示如下：

```
'57'::INT →57
57::CHAR →'57'
57::NUMERIC(4,2)→57.00
'05-23-87'::DATE →1987-05-23
```

TO_CHAR

描述

TO_CHAR 函数可以将不同类型的输入数据类型转换成字符串数据类型。除了进行数据转换外，TO_CHAR 函数还可以按期望的格式格式化输出字符串。

输入

不管处理的数据是哪一种类型，TO_CHAR 函数均遵循一个通用的模式用法。所有的 TO_CHAR 函数接受两个参数；第一个是要转换的数据，第二个是构建输出时 PostgreSQL 所使用的格式模板。表 4-2 列举了相关用法。

表 4-2　　TO_CHAR 函数用法

用　法	描　述
TO_CHAR(int,texttemplate)	从整数转换成指定的字符串格式
TO_CHAR(numeric, texttemplate)	从数字转换成指定的字符串格式
TO_CHAR(double precision, texttemplate)	从双精度数转换成指定的字符串格式
TO_CHAR(timestamp, texttemplate)	从时间戳转换成指定的字符串格式

用 TO_CHAR 转换数字（Int、Numerric 或 Double Precision 类型）：

使用下列屏蔽模板来格式化输出，从一个数字数据类型转换成一个字符串类型（见表 4-3）。

（除了下面指定的格式化命令外，TO_CHAR 函数还可以在双引号内接受和显示任意的文本。当需要指定输出数据的标签时，这一功能相当有用。）

表 4-3　　使用屏蔽模板来格式化输出

条　目	描　述
0	前导 0
9	数字占位符
.	小数点
,	千分隔符
G	组分隔符*
D	小数点*

续表

条　目	描　述
S	带减号（-）的负值*
PR	尖括号（<>）内的负值*
L	货币符号*
MI	指定位置的减号（如果 n<0）
PL	指定位置的加号（如果 n>0）
SG	指定位置的加号或减号
RN	输出罗马数字（当 n>1 及 n<3999）
TH	转换成序号
Vn	将值移动 n 位（通过乘以 10^n）*

用 TO_CHAR 转换日期/时间数据类型：

TO_CHAR（和 TO_DATE、TO_TIMESTAMP）函数使用下列日期/时间类屏蔽模板来格式化输出：

表 4-4　　使用日期/时间屏蔽模板来格式化输出

条　目	描　述
SSS	与午夜间隔秒（0-86399）
SS	秒（00-59）
MI	分（00-59）
HH	一天的小时（01-12）**
HH12	一天的小时（01-12）
HH24	一天的小时（00-24）
AM 或 A.M.	正午以前（大写）
PM 或 P.M.	正午以后（大写）
am 或 a.m.	正午以前（小写）
pm 或 p.m.	正午以后（小写）
DAY	星期几的大写（如 MONDAY）
day	星期几的正常大小写（如 Monday）
DY	星期几的大写缩写（如 MON）
dy	星期几的正常大小写缩写（如 Mon）
D	周的某一天（1-7；SUN=1）
DD	月的某一天（01-31）
DDD	年的某一天（001-366）
W	月的某一周（1-5）
WW	年的某一周（1-53；第一周从 1 月 1 日开始）
IW	按 ISO 标准年的某一周（1-53；第一周从 1 月的第 1 个星期四开始）

* 这些条目将使用计算机的本地设置，所以可能结果有所不同。

** 这些条目将使用计算机的本地设置，所以可能结果有所不同。

续表

条　目	描　述
MM	月份（01-12）
MONTH	大写完整月份名（如 JUNE）
Month	正常大小写完整月份名（如 June）
month	小写完整月份名（如 june）
MON	大写月份名缩写（如 JUN）
Mon	正常大小写月份名缩写（如 Jun）
mon	小写月份名缩写（如 jun）
Y	年的最后一位（如 1）
YY	年的最后两位（如 01）
YYY	年的最后三位（如 001）
YYYY	完整年份（4 位或更多）（如 2001）
Y,YYY	完整年份（4 位或更多）（如 2,001）
CC	世纪（如 20）
BC 或 B.C.	纪元标识（大写）
bc 或 b.c.	纪元标识（小写）
AD 或 A.D.	纪元标识（大写）
ad 或 a.d.	纪元标识（小写）
J	Julian 天数（从 01/01/4712 BC 开始）
Q	季度
RM	以罗马数字表示的大写月份（如 I=Jan）
rm	以罗马数字表示的小写月份（如 I=Jan）
TZ	大写时区
tz	小写时区

示例

下面提供转换数字和日期/时间类型数据的例子。

用 TO_CHAR 转换数字例子（见表 4-5）

表 4-5　　数字类型数据转换实例

输　入	输　出
TO_CHAR(123,'999')	123
TO_CHAR(123,'99 9')	12 3
TO_CHAR(123,'0999')	0123
TO_CHAR(123,'999.9')	123.0
TO_CHAR(1234,'9,999')	1,234
TO_CHAR(1234,'9G999')	1,234
TO_CHAR(1234.5,'9999D99')	1234.50
TO_CHAR(123,'999PL')	123+

续表

输　入	输　出
TO_CHAR(123,'PL123')	+123
TO_CHAR(-123,'999MI')	123-
TO_CHAR(-123,'MI123')	-123
TO_CHAR(123,'SG123')	+123
TO_CHAR(123,'SG123')	-123
TO_CHAR(-123,'999PR')	<123>
TO_CHAR(123,'RN')	CXXIII
TO_CHAR(32,'99TH')	32nd
TO_CHAR(123,'9"Hundred and"99')	1 Hundred and 23

用 TO_CHAR 转换日期/时间例子（见表 4-6）。

表 4-6　　日期/时间类型数据转换实例

输　入	输　出
TO_CHAR('November 1 2001','MM"--"DD"--"YY')	11--01-01
TO_CHAR('Jun 22 2001','"year"YYYY "Day"DDD')	Year 2001 Day 174

注意

忽略双引号中的任何项目。所以，要输出保留模板字，只要简单地将它们放在双引号中就行了（即用“YYYY”可输出 YYYY）。

一些特殊字符，如反斜杠（\），可以将它们放在引号中并重复此字符（如“\\”输出后就成了“\”）。

前面的模板可用于许多其他的 TO 样式函数中（如 TO_DATE、TO_NUMBER 等等）。

TO_DATE

描述

TO_DATE 函数将一个文本字符串转换成一个日期格式。TO_DATE 函数带有两个参数：第一个是要转换的字符串，第二个是指定如何显示输出样式的文本模板。

输入

TO_DATE (text,texttemplate)

示例

TO_DATE('01 01 2001','MONTH DD YYYY')→JANUARY 01 2001

注意

文本模板字符串可以指定很多选项。请参考 TO_CHAR 函数了解日期/时间模板可选参数的完整列表。

TO_NUMBER

描述

TO_NUMBER 函数用于将输入的字符串转换成一个数字输出。TO_NUMBER 函数接受两个输入参数：第一个是要转换的文本，第二个是指定如何显示输出格式的文本模板。

输入

```
TO_NUMBER(text,texttemplate)
```

示例

```
TO_CHAR(1234567,'9G999G999')→1,234,567
TO_CHAR(1234.5,'9999D99')→1234.50
```

注意

TO_NUMBER 函数的文本模板可接受很多选项。请参考 TO_CHAR 函数了解所支持选项的完整列表。

TO_TIMESTAMP

描述

TO_TIMESTAMP 函数用于将一个字符串格式转换成一个时间戳数据类型。TO_TIMESTAMP 函数接受两种参数：第一个是将要转换的字符串，第二个是格式化结果输出的日期/时间模板。

输入

```
TO_TIMESTAMP(text ,texttemplate )
```

示例

```
TO_TIMESTAMP('05 December 2001','DD MM YYYY')→12 05 2001
```

注意

日期/时间模板可接受很多格式化输出的选项。请参考 TO_CHAR 函数了解有效的日期/时间格式化选项完整列表。

4.4 几何类函数

在流行 RDBMS 中，PostgreSQL 稍有一些特殊的是，它提供了一套广泛的几何类函数。这些函数及其相关操作符在计算空间类数据组时很有用。

AREA

描述

AREA 函数用于计算给定对象所占的面积。

输入

```
AREA(obj)
```

示例

```
AREA(box '((1,1),(3,3))') →4
```

BOX

描述

BOX 函数有几个版本。大部分版本将其他几何类型转换成 box 数据类型。不过，如果 BOX 函数计算对象为两个重叠的矩形，其结果为两者相交所得矩形。

输入

```
BOX(box,box)——执行相交。
BOX(circle)——将圆转换成矩形。
BOX(point,point)——将点转换成矩形。
BOX(polygon)——将多边形转换成矩形。
```

示例

```
BOX(box '((1,1),(3,3)',box '((2,2),(4,4))')→BOX'(3,3),(2,2)'
BOX(circle'(0,0),2')→BOX'(1.41,1.41),(-1.41,-1.41)'
BOX(point'(0,0)',point '(1,1)')→BOX'(1,1),(0,0)'
BOX(polygon '(0,0),(1,1),(1,0)'→BOX '(1,1),(0,0)'
```

CENTER

描述

CENTER 函数返回计算对象的中心点。

输入

```
CENTER(obj)
```

示例

```
CENTER(box '(0,0),(1,1)')→point '(.5,.5)'
```

CIRCLE

描述

CIRCLE 函数将一个矩形数据类型转换成一个圆。

输入

CIRCLE(box)——将矩形转换成圆。

示例

```
CIRCLE(box'(0,0),(1,1)') → CIRCLE'(.5,.5),.707016…'
```

DIAMETER

描述

DIAMETER 函数返回指定圆的直径。

输入

```
DIAMETER(circle)
```

示例

```
DIAMETER(circle'((0,0),2)') → 4
```

HEIGHT

描述

HEIGHT 函数用于计算指定矩形的垂直高度。

输入

```
HEIGHT(box)
```

示例

```
HEIGHT(box'(0,0),(3,3)') → 3
```

ISCLOSED

描述

ISCLOSED 函数返回一个代表指定路径是否开放或封闭的布尔值。

输入

```
ISCLOSED(path)
```

示例

```
ISCLOSED(path '(0,0),(1,1),(1,0),(0,0)')→t
```

ISOPEN

描述

ISOPEN 函数返回一个代表指定路径是否开放或封闭的布尔值。

输入

```
ISOPEN(path)
```

示例

```
ISOPEN(path '(0,0),(1,1),(1,0),(0,0)')→f
```

LENGTH

描述

LENGTH 函数返回指定 lseg 的长度。

输入

```
LENGTH(lseg)
```

示例

```
LENGTH(lseg '(0,0),(1,1)'→1.41422135623731
```

注意

LENGTH 函数传递参数是一个 BOX 数据类型，它计算的是矩形 lseg 对角的长度。

LSEG

描述

LSEG 函数将一个矩形或一对点转换成一个 lseg 数据类型。

输入

```
LSEG(box)
LSEG(point,point)
```

示例

```
LSEG(box '(0,0),(1,1)') →LSEG '(1,1),(0,0)'
```

NPOINT

描述

NPOINT 函数返回指定路径组合的点数量。

输入

```
NPOINTS(path)
NPOINTS(polygon)
```

示例

```
NPOINTS(path '(0,0),(1,1)') → 2
```

PATH

描述

PATH 函数将一个多边形转换成一个路径。

输入

```
PATH(polygon)
```

示例

```
PATH(polygon '(0,0),(1,1),(1,0)') →PATH '((0,0),(1,1),(1,0))'
```

注意

请注意示例中的闭合代表符号“(”。参见 path 数据类型可以了解开放或封闭路径的更多信息。

PCLOSE

描述

PCLOSE 函数将一个路径转换成封闭类型路径。

输入

```
PCLOSE(path)
```

示例

```
PCLOSE(path '(0,0),(1,1),(1,0)') →PATH '((0,0),(1,1),(1,0))'
```

注意

请参见 path 数据类型了解如何让路径开放或封闭的更多信息。

POINT

描述

POINT 函数根据提供的对象类型可以加载不同的几何转换功能。

输入

POINT(circle)—返回指定圆的圆心。

POINT(lseg,lseg)—返回指定 lseg 的相交点。

POINT(polygon)—返回指定多边形的中心。

示例

```
POINT(circle '((0,0),2)') →POINT '(0,0)'
POINT(polygon'(0,0),(1,1),(1,0)') →POINT '(.66 …),.33 …)'
```

POLYGON

描述

POLYGON 函数将不同的几何类型转换成一个多边形。

输入

POLYGON(box)—将矩形转换成一个 12 个点的多边形。

POLYGON(circle)—将圆转换成一个 12 个点的多边形。

POLYGON(n,circle)—将圆转换成一个 n 个点的多边形。

POLYGON(path)—将路径转换成一个多边形。

示例

```
POLYGON(4,circle '((0,0),4)') →POLYGON '(-4,0),(2.041,4),(4,
4.0827),(-6.12,-4)'
```

POPEN

描述

POPEN 函数将路径转换成开放类型路径。

输入

```
POPEN(path)
```

示例

```
POPEN(path '(0,0),(1,1),(1,0)') →
PATH '[(0,0),(1,1),(1,0)]'
```

注意

请注意返回路径的开放形式。请参见 path 数据类型了解开放或封闭路径形式的更多信息。

RADIUS

描述

RADIUS 函数返回给定圆的半径。

输入

```
RADIUS(circle)
```

示例

```
RADIUS(circle '((0,0),2)') →2
```

WIDTH

描述

WIDTH 函数返回给定矩形的水平大小。

输入

```
WIDTH(box)
```

示例

```
WIDTH(box '(0,0),(2,2)') →2
```

4.5 网络类函数

PostgreSQL 包含许多网络类函数。它们主要用于计算和转换 IP 相关数据。下面部分内容将讨论 PostgreSQL 中包含的网络类函数。

ABBREV

描述

ABBREV 函数返回一个给定 inet 或 cidr 值的缩写文本格式。

输入

```
ABBREV(inet |cidr )
```

示例

```
ABBREV('192.168.0.0/24 ') →"192.168/24 "
```

BROADCAST

描述

BROADCAST 函数返回给定 inet 或 cidr 值的广播地址。

输入

```
BROADCAST(inet|cidr )
```

示例

```
BROADCAST('192.168.0.1/24 ') →'192.168.0.255/24 '
```

HOST

描述

HOST 函数抽取给定 inet 或 cidr 值的主机地址。

输入

```
HOST(inet | cidr )
```

示例

```
HOST('192.168.0.101/24 ') →'192.168.0.101 '
```

MASKLEN

描述

MASKLEN 函数抽取给定 inet 或 cidr 值的网络掩码长度。

输入

```
MASKLEN(inet | cidr )
```

示例

```
MASKLEN('192.168.0.1/24 ') →24
```

NETMASK

描述

NETMASK 函数计算给定 inet 或 cidr 值的网络掩码。

输入

```
NETMASK(inet | cidr )
```

示例

```
NETMASK('192.168.0.1/24 ') →'255.255.255.0 '
```

NETWORK

描述

NETWORK 函数从给定 inet 或 cidr 值中抽取地址的网络部分。

输入

```
NETWORK(inet | cidr )
```

示例

```
NETWORK('192.168.0.155/24 ') →'192.168.1.0/24 '
```

TEXT

描述

TEXT 函数以文本值返回 IP 地址和网络掩码长度。

输入

```
TEXT(inet | cidr )
```

示例

```
TEXT(CIDR '192.168.0.1/24 ') →"192.168.0.1/24 "
```

TRUNC

描述

TRUNC 函数将给定 macaddr 值后 3 个字节置为 0。

输入

```
TRUNC(macaddr)
```

示例

```
TRUNC(macaddr '33:33:33:33:33:aa ') →'33:33:33:00:00:00 '
```

注意

把给定的 MAC 地址与制造商相关联时此函数有用。请参见目录 $SOURCE/contrib/mac（$SOURCE 为 PostgreSQL 源代码所在位置）了解更多信息。

4.6 数字类函数

PostgreSQL 中包含许多有助于数字计算的函数。下面部分内容将讨论 PostgreSQL 中包含的此类数字函数。

ABS

描述

ABS 函数返回给定数字的绝对值。

输入

```
ABS(num)
```

示例

```
ABS(-7) →7
ABS(-7.234) →7.234
```

注意

ABS 函数的返回值与传递参数为同一数据类型。

ACOS

描述

ACOS 函数返回一个反余弦。

输入

```
ACOS(num)
```

ASIN

描述

ASIN 函数返回一个反正弦。

输入

```
ASIN(num)
```

ATAN

描述

ATAN 函数返回一个反正切。

输入

ATAN(*num*)

ATAN2

描述

ATAN2 函数返回一个 y/x 的反正切。

输入

ATAN2(*x*,*y*)

CBRT

描述

CBRT 函数返回给定数的立方根。

输入

CBRT(*num*)

示例

CBRT(27) →3

CEIL

描述

CEIL 函数返回不小于给定值的最小整数。

输入

CEIL(*num*)

示例

CEIL(-22.2) →-22

COS

描述

COS 函数返回一个余弦值。

输入

COS(*num*)

COT

描述

COT 函数返回一个余切值。

输入

```
COT(num)
```

DEGREES

描述

DEGREES 函数将弧度转换为角度。

输入

```
DEGREES(num)
```

示例

```
DEGREES(1) →90
```

EXP

描述

EXP 函数对给定值进行指数计算。

输入

```
EXP(num)
```

示例

```
EXP(0) →1.0
```

FLOOR

描述

FLOOR 函数返回不大于给定值的最大整数。

输入

```
FLOOR(num)
```

示例

```
FLOOR(-22.2) →-23
```

LN

描述

LN 函数对给定数求自然对数。

输入

```
LN(num)
```

示例

```
LN(100) →4.6051701860
```

LOG

描述

LOG 函数对给定值求标准对数（以 10 为底数）。

输入

```
LOG(num)
```

示例

```
LOG(100) →2.0
```

PI

描述

PI 函数返回标准 π（pi）值。

输入

无。

示例

```
PI()→3.1459265358979
```

POW 或 POWER

描述

POW 函数对一个数求指定次幂。

输入

```
POW(num,exp)
```

Num——所求幂的底数。

Exp——所求幂的指数。

示例

```
POW(2,2) →4.0
POW(2,3) →8.0
```

RADIANS

描述

RADIANS 函数将给定角度转换成弧度。

输入

```
RADIANS(num)
```

示例

```
RADIANS(90) →1
```

RANDOM

描述

RANDOM 函数返回一个 0.0 和 1.0 之间的伪随机数。

输入

无。

示例

```
RANDOM→.654387
```

ROUND

描述

ROUND 函数根据指定小数位圆整一个数。

输入

```
ROUND (num,dec)
```

Num——所处理数字。

Dec——表示小数位数的整数。

示例

```
ROUND (1.589,1) →1.6
ROUND (1.589,2) →1.59
```

SIN

描述

SIN 函数对给定值求正弦。

输入

```
SIN(num)
```

SQRT

描述

SQRT 函数返回给定值的平方根。

输入

```
SQRT(num)
```

示例

```
SQRT(9) →3
```

TAN

描述

TAN 函数对给定数求正切。

输入

```
TAN(num)
```

TRUNC

描述

TRUNC 函数按小数位截断指定数（不进行圆整）。

输入

```
TRUNC(num[,dec])
```

num——将要截断的数。

dec——若指定则为要保留的小数位；否则截断所有小数位。

示例

```
TRUNC(1.589999,2) →1.58
TRUNC(1.589999) →1
```

4.7 SQL 类函数

PostgreSQL 包含若干基于当前 SQL 语句中表达式 z 给定的返回值的函数。而且，这些函数并不强制作用于指定数据类型；但它更适合于在一个 SQL 语句中充当控制结构。

CASE WHEN

描述

CASE WHEN 函数是一个简单的条件评估工具。大部分编程语言包含相似的结构。CASE WHEN 与常用的 IF…THEN…ELSE 语句具有相同的功能。

输入

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

示例

下面显示了一个经典的 IF…THEN…ELSE 结构范例，只是其中使用了 CASE WHEN 函数。将职员的年龄与一常量比较，根据对应的年龄可能输出为 minor、adult 或 unknown:

```
SELECT name,age,
CASE WHEN age<18 THEN 'minor '
      WHEN age>=18 THEN 'adult '
      ELSE 'unknown '
END
FROM employees;

name age case
-------------------
Bill 13 minor
Timmy 7 minor
Pam 25 adult
Barry NULL unknown
```

COALESCE

描述

COALESCE 函数接受一个任意的输入参数，并返回第一个非 NULL 值。COALESCE 函数对显示任意数据源的缺省值非常有用。

输入

```
COALESCE(arg1 ,…,argN )
```

示例

返回用户一个缺省信息：

```
SELECT COALESCE(book.title,book.description,'Not Available ');
```

NULLIF

描述

NULLIF 函数接受两个参数。仅当两个参数值相等时，此函数返回一个 NULL 值。否则，它将返回第一个参数值。

输入

```
NULLIF(arg1 ,arg2 )
```

示例

在此例中，返回的是第一个值，因为两个参数值不相等：

```
SELECT NULLIF('hello ','world ');

----------------
'hello '
```

不过，当两参数值相等时返回一个 NULL 值：

```
SELECT NULLIF('hello ',SUBSTR('helloword ',1,5));

NULL
```

注意

NULLIF 函数的作用方式与 COALESCE 函数相反。这些函数对于异常测试有用，方法是用一个变量与一个已知值进行测试。如果变量与已知值相等，则返回 NULL。然而，如果与已知值不匹配，则返回被测试变量。

4.8 字符串类函数

PostgreSQL 包含几个用于修改字符串类型数据的函数。这些函数对于控制输出显示和（或）标准化输入数据特别有用。

ASCII

描述

ASCII 函数返回给定字符的 ASCII 值。

输入

```
ASCII(chr)
```

chr—决定 ASCII 值的字符。

示例

```
ASCII('A ') →65
ASCII('Apple ') →65
```

注意

ASCII 函数处理多字符时，只是对一个字符起作用。

CHR

描述

CHR 函数返回与 ASCII 值对应的字符。

输入

```
CHR(val)
```

val—一个 ASCII 值。

示例

```
CHR(65) →'A '
```

INITCAP

描述

当首字符为大写字母而其余为小写字母时，INITCAP 函数强制返回一个字符串或列。

输入

```
INITCAP(col)
```

或

```
INITCAP(string)
```

示例

```
SELECT INITCAP(name)AS Proper_Name FROM authors;

Proper_Name
--------
Bill
Bob
Sam
```

LENGTH、CHAR_LENGTH 或 CHARACTER_LENGTH

描述

LENGTH（或 CHAR_LENGTH、CHARACTER_LENGTH）函数返回给定列的长度。

输入

```
LENGTH (col)
```

col——一个包含字符串数据类型的列。

示例

```
SELECT name WHERE LENGTH(name)<4 FROM authors;

Name
------
Pam
Sam
Sue
Bob
```

LOWER

描述

LOWER 函数强制返回一个小写字符串或列。

输入

```
LOWER(col)
```

或

```
LOWER(string)
```

示例

```
SELECT LOWER(name)AS Low_Name FROM authors;

Low_Name
--------
bill
bob
sam
```

LPAD

描述

LPAD 函数用指定的字符或空格左填充字符串。

输入

```
LPAD(str,len,fill)
```

Str——左填充字符串。

Len——填充的空格数。

Fill——缺省时为空格；不过，也可定义为任意字符。

示例

```
LPAD('Hello ',3) →' Hello '
LPAD('ello ',3,'H ') →'HHHello '
```

LTRIM

描述

LTRIM 函数从字符串左边删除指定字符。

输入

```
LTRIM(str [,TRIM])
```

Str——修剪的字符串。

TRIM——缺省时为空格；不过，也可定义为其他任意字符。

示例

```
LTRIM(' Hello ') →'Hello '
LTRIM('HHHello ','H ') →'ello '
```

OCTET_LENGTH

描述

OCTET_LENGTH 函数返回列或字符串的长度，包括其中存在的任何多字节数据。

输入

```
OCTET_LENGTH(col)
```

或

```
OCTET_LENGTH(string)
```

示例

```
SELECT OCTET_LENGTH('Hello World');

Octet_length
11
```

注意

OCTET_LENGTH 函数和 LENGTH 通常返回相同值。不过，一个根本的区别是，OCTET_LENGTH 实际上返回一个字符串的字节数。如果保存多字节信息，两者就会显示重要差别了。

POSITION

描述

POSITION 函数返回一个表示指定字符串在已知列（或已知字符串）中位置的整数。

输入

```
POSITION(str IN col)
```

Str——定位的字符串。

Col——预先搜索的列或字符串。

示例

从表 authors 中返回第二个字母是“a”的名字：

```
SELECT name FROM authors WHERE POSITION('a' IN name)=2;
```

```
Name
-----
Pam
Sam
Tammy
Barry
```

STRPOS

描述

STRPOS 函数返回一个表示指定字符串在已知列（或已知字符串）中位置的整数。

输入

```
STRPOS (col,str)
```

Col——预先搜索的列或字符串。

str——定位的字符串。

示例

请参见 POSITION 函数部分示例。

注意

此函数与 POSITION 函数实质功能一样。

RPAD

描述

RPAD 函数用空格或字符右填充指定字符串。

输入

```
RPAD(str,len[,fill])
```

str—右填充的字符串。

len—添加的空格数。

fill—缺省时为空格；不过，也可使用其他任意字符。

示例

```
RPAD('Hello ',3) →'Hello'
RPAD('Hello ',3,'!') → Hello!!!
```

RTRIM

描述

RTRIM 函数从一个字符串的右边删除指定的字符。

输入

```
RTRIM(str [,TRIM])
```

str—右修剪的字符串。

TRIM—缺省时为空格；不过，也可使用其他任意字符。

示例

```
RTRIM('Hello ') →'Hello'
RTRIM('Hello!!!','!') →'Hello'
```

SUBSTRING

描述

SUBSTRING 函数从一个现有字符串中抽取指定部分。

输入

```
SUBSTRING(str FROM POS [FOR len])
```

Str——所处理的字符串。

Pos——起始抽取处。

Len——缺省时抽取剩下字符串部分；不过，也可指定抽取位置。

示例

```
SUBSTRING('Hello ' FROM 2))→'ello '
SUBSTRING('Hello ' FROM 2 FOR 2)) 'el '
```

注意

此函数与 SUBSTR 函数功能相同。

SUBSTR

描述

SUBSTR 函数从一个现有字符串中抽取指定部分。

输入

```
SUBSTR(str POS [,len])
```

Str——所处理的字符串。

Pos——起始抽取处。

Len——缺省时抽取剩下字符串部分；不过，也可指定抽取位置。

示例

```
SUBSTRING('Hello ',2) 'ello '
SUBSTRING('Hello ',2,2) 'el '
```

注意

此函数与 SUBSTRING 函数功能相同。

TRANSLATE

描述

TRANSLATE 函数对一个指定字符串进行查找与替换。对字符串中匹配查找标准部分进行数据替换。请参见下面示例作更详细了解。

输入

```
TRANSLATE(str search,replaceset)
```

Str——查找和修改的基本字符串。

Search——单个或多个字符查找集。

Replaceset——用替换集的成员替换查找集的对应成员。

示例

```
TRANSLATE('HelloW ','W ','!') → 'Hello!'
TRANSLATE('Hello ','Ho ','Jy ') →'Jelly '
```

TRIM

描述

TRIM 函数从左或从右（或从两边）删除给定字符串中的指定字符或空白。

输入

```
TRIM([leading | trailing | both] [trim] FROM str)
```

leading | trailing | both—从哪边删除指定字符。

Trim——缺省时为空格；不过，也可定义为其他任意字符。

Str——要删除的字符。

示例

```
TRIM(both FROM ' Hello ') →Hello '
TRIM(both '!' FROM '!!HELLO!!')  Hello '
```

UPPER

描述

UPPER 函数强制返回大写字符串或列。

输入

```
UPPER(col)
```

或

```
UPPER(string)
```

示例

```
SELECT UPPER(name)AS Upper_Name FROM authors;

Upper_Name
--------
BILL
BOB
SAM
```

4.9 时间类函数

下列函数有助于执行计算基于时间和日期类的数据。对时间类数据组执行计算和转换

时，此函数可以发挥作用。

AGE

描述

AGE 函数返回一个代表当前时间与参数提供时间相关的时间间隔。

输入

AGE(timestamp)——计算给定时间戳和 now()之间的相差数。

AGE(timestamp, timestamp)——计算两个给定时间戳之间的相差数。

示例

```
AGE('03-01-2001 15:56:00 ','11-01-2001 14:22:00 ') →7 mon 30 22:26 ago
```

CURRENT_DATE

描述

CURRENT_DATE 函数返回当前系统日期。

输入

无。

示例

```
SELECT CURRENT_DATE;→2001-06-11
```

注意

请注意此函数尾部没有圆括号“()”。这是为了保持与 SQL 的兼容性。

CURRENT_TIME

描述

CURRENT_TIME 函数返回当前系统时间。

输入

无

示例

```
SELECT CURRENT_TIME;→22:10:31
```

注意

请注意此函数尾部没有圆括号“()”。这是为了保持与 SQL 的兼容性。

CURRENT_TIMESTAMP

描述

CURRENT_TIMESTAMP 函数返回当前系统日期和时间。

输入

无。

示例

```
SELECT CURRENT_TIMESTAMP;→'2001-06-11 22:10:31-06'
```

注意

请注意此函数尾部没有圆括号“()”。这是为了保持与 SQL 的兼容性。此函数与 NOW 函数类似。

DATE_PART

描述

DATE_PART 函数从给定日期/时间参数中抽取指定部分。

输入

```
DATE_PART(formattext, timestamp)
DATE_PART(formattext, interval)
```

Formattext——有效 DATE_PART 格式选项之一；参见下面内容。

timestamp/ interval—给定的时间有关值。

DATE_PART 格式化选项

表 4-7 中关键字为抽取中的有效日期/时间元素。

表 4-7　　DATE_PART 格式化选项中有效关键字

选　项	描　述
Millennium	抽取被 1,000 整除后的年份字段
Century	抽取被 100 整除后的年份字段
Decade	抽取被 10 整除后的年份字段
Year	抽取年份字段
Day	年中的某天（1-366）（仅时间戳）
Quarter	年中的某季度（1-4）（仅时间戳）
Month	年中的某月份（1-12）（仅时间戳）。剩下月份数（仅时间间隔）
Week	年中的某星期（仅时间戳）
Dow	星期中的某天（0-6；0=Sunday）（仅时间戳）
day	月中的某天（1-31）（仅时间戳）
hour	小时字段（0-23）
second	秒字段，包括小数（0-59.99）
milliseconds	乘以 1,000 的秒字段，包括小数
microseconds	乘以 1 百万的秒字段，包括小数
epoch	从 01-01-1970 00:00 开始的秒数（时间戳）。总秒数（时间间隔）。

示例

```
DATE_PART('second ',TIMESTAMP '06-01-2001 12:23:43 ')→43
DATE_PART('hour ',TIMESTAMP '06-01-2001 12:23:43 ') →12
```

注意

当使用 DATE_PART 处理 interval 数据类型时，认识到 DATE_PART 不会进行隐式计算这一点很重要。DATE_PART 只能作为一个抽取工具。例如，如果时间间隔是 1 month

ago，欲从中抽取 days，DATE_PART 将返回 0。

DATE_TRUNC

描述

DATE_TRUNC 函数按指定精度截断给定时间戳。

输入

```
DATE_TRUNC(formattext, timestamp)
```

Formattext——时间戳的截断精度值；参见表 4-8 中选项。

Timestamp——用于截断的给定日期/时间值。

DATE_TRUNC 格式化选项

表 4-8 中列出 DATE_TRUNC 函数可采用的不同精度级别。

表 4-8 **DATE_TRUNC 函数可采用的不同精度级别**

选　项	描　述
Millennium	截断比千年小的时间量
Century	截断比百年小的时间量
Decade	截断比十年小的时间量
Year	截断比年小的时间量
Month	截断比月小的时间量
Day	截断比天小的时间量
Hour	截断比小时小的时间量
Minute	截断比分钟小的时间量
Second	截断比秒小的时间量
Milliseconds	截断比毫秒小的时间量
Microseconds	截断比微秒小的时间量

示例

```
DATE_TRUNC('hour ',TIMESTAMP '2001-11-1 23:11:45 ')
TIMESTAMP '2001-11-1 23:00:00 '
DATE_TRUNC('year ' TIMESTAMP '11-1-2001 23:11:45 ')
TIMESTAMP '2001-01-01 00:00:00 '
```

EXTRACT

描述

EXTRACT 函数从给定时间戳中抽取指定值。

输入

```
EXTRACT(formattext FROM timestamp )
EXTRACT(formattext FROM interval )
```

Formattext——有效日期字段。参见 DATE_PART 部分的有效格式代码列表。

interval/timestamp—给定的时间值。

示例

```
EXTRACT('hour ' FROM TIMESTAMP '2001-11-1 23:33:45 ') →23
```

注意

EXTRACT 函数与 DATE_PART 功能相似。两者语法可互换。

ISFINITE

描述

ISFINITE 函数返回一个表示给定的时间戳或时间间隔是否代表一个有限时间的布尔值。

输入

```
ISFINITE(timestamp )
ISFINITE(interval )
```

示例

```
ISFINITE(TIMESTAMP '2002-05-05 23:13:44 ') t
```

NOW

描述

NOW 函数返回一个表示当前系统时间的时间戳。

输入

无。

示例

```
SELECT now();
'2001-11-1 15:23:54-06 '
```

注意

NOW 函数在概念上与 CURRENT_TIMESTAMP 完全相同。

TIMEOFDAY

描述

TIMEOFDAY 函数返回一个高精度日期和时间值。

输入

无。

示例

```
SELECT TIMEOFDAY(); 'Sun Mar 11 22:23:14.853452 2001 CST '
```

TIMESTAMP

描述

TIMESTAMP 函数将日期或日期-时间数据类型转换成时间戳。

输入

```
TIMESTAMP(date)
TIMESTAMP(date,time)
```

示例

```
TIMESTAMP('06-01-2001 ','23:45:11 ')  '2001-06-01 23:45:11 '
```

4.10 用户类函数

PostgreSQL 中的某些函数用于处理用户和对话问题。下面部分的内容将讨论用户类函数。

CURRENT_USER

描述

CURRENT_USER 返回用于权限检查的用户 ID。

输入

无。

示例

```
SELECT CURRENT_USER;
--------------
webuser
```

注意

目前版本中，CURRENT_USER 和 SESSION_USER 为相同函数，但在将来版本中，为了在 setuid 模式下编程的需要，将会把两者区分开来。

请注意以上函数调用时尾部不带圆括号“（）”，这是为了保持与 SQL 的兼容性。

SEEESON_USER

描述

SEEESON_USER 函数返回当前登陆 PostgreSQL 的用户 ID。

输入

无。

示例

```
SELECT SESSION_USER;
--------------
webuser
```

注意

目前版本中，CURRENT_USER 和 SEEESON_USER 相同，但在将来版本中，为了在 setuid 模式下编程的需要，将会把两者区分开来。

请注意以上函数调用时尾部不带圆括号“（）”，这是为了保持与 SQL 的兼容性。

USER

注意

请参见 CURRENT_USER 函数。

4.11　其他类函数

PostgreSQL 中的其他一些函数不能严格划分于以上类别中。下面部分将介绍此类函数中的一例。

ARRAY_DIMS

描述

ARRAY_DIMS 函数返回保存在数组字段中的元素数目。

输入

```
ARRAY_DIMS(col-)
```

示例

```
SELECT ARRAY_DIMS(testscore) FROM students;
array_dims
-----------
[1:4]
```

第 5 章　其他 PostgreSQL 主题

与所有 RDBMS 一样，PostgreSQL 有自己特定的方式来执行索引和事务控制之类常规任务。另外，PostgreSQL 还定义了一些特有的概念。

本章将介绍 PostgreSQL 执行以下任务的相关知识：

- 字段中的数组。
- 继承。
- 索引。
- 对象标识符（OID）。
- 多版本并发控制（MVCC）。

5.1　字段中的数组

PostgreSQL 支持的优秀特征之一是允许表中的字段容纳数组。这一概念的运用可以使一个字段中保存多个相同数据类型的值。

要在字段中插入数组，创建表时就应该标明此字段可容纳数组。将字段指定为容纳数组后，就可以通过在相应表中指定数组元素来插入、选择或修改数组中的数据。

创建一个数组

例如，让我们创建一个名为 students 的表，表用一个四元素数组字段来保存测试分数：

```
CREATE TABLE students
(name char(20),testscore int [4 ]);
```

注意，在创建表时，通过在数据类型定义后面包含方括号[]来表示此字段支持数组。

使用数组字段

当插入、修改或选择数据时，可以显式选择指定的数组元素。PostgreSQL 中的数组元素以 1 开始。所以，数组 testscore 中的第一个元素被引用为 testscore[1]。

下面让我们在表 students 中插入一些样本数据。注意如何使用大括号{}来选择想要的指定元素：

```
INSERT INTO students (name,testscore)
     VALUES ('Bill ','{90,0,0,0}');
```

```
INSERT INTO students (name,testscore)
     VALUES ('Sam ','{86,0,0,0}');
INSERT INTO students (name,testscore)
     VALUES ('Pam ','{95,0,0,0}');
```

还要注意，通过使用花括号{}来引用数组字段；而且被引用的数组元素是你需要在此处插入数据的数组元素。同样，更改行数据时应采用相同的方法。

事实上，替换对象或者是整个数组或者是指定的元素：

```
UPDATE students SET testscore='{95,86,0,0}' WHERE name=='Pam ';
```

或者，你也可以修改期望的元素：

```
UPDATE students SET testscore [3 ]=98 WHERE name='Pam ';
```

可按同样的方法选择指定元素。例如，要选择在前三场考试中成绩高于 85 分的所有学生，你可以使用如下代码：

```
SELECT *FROM students WHERE testscore [1 ]>85 AND testscore [2]>85 AND
testscore [3]>85;
```

多维数组

PostgreSQL 同时允许创建和使用多维数组。假设你想跟踪每个学生的每半学年考试情况。在此情况下，创建一个多维数组就很容易保存此信息：

```
CREATE TABLE students (name char(20),testscore int [4][4]);
```

你可以像以前一样插入和访问此信息（注意其中双花括号的使用）：

```
INSERT INTO students VALUES ('Bill ',
     '{{75,85,99,68},{88,91,77,87}}');
```

在指定的多维数组中也可选择指定的元素。例如，要查看谁在下半学年的第三场考试中分数高于 90 分：

```
SELECT *FROM students WHERE testscore [2 ][3 ]>90;
```

扩展数组

关于 PostgreSQL 的数组结构，需要提醒（也可以说是 PostgreSQL 的一个优点）的是，数组中的元素大小可以动态扩展。虽然你可以在创建表时显式定义一个最大数组的大小，但使用 UPDATE 命令可以改变此大小。

例如，下面是一个带有数组的表的例子，使用 UPDATE 命令来扩展数组大小：

```
CREATE TABLE students (name char(20),testscore int [3 ]);
INSERT INTO students VALUES ('Bill ','{96,84,98}');
SELECT *FROM students;

Name testscore
-------------------
Bill {96,84,98}
UPDATE students SET testscore='{96,84,98,100}';
SELECT *FROM students;

Name testscore
```

```
-------------------
Bill {96,84,98,100}
```

虽然这可能是一个有用的特征，但如果使用时不小心也可能带来麻烦。它可能导致不同的行所含的数组元素数目不同。

一个处理数组的有用函数是 ARRAY_DIMS 函数。此函数返回数组中的当前元素数量。

请参考第 4 章“PostgreSQL 函数”中的 ARRAY_DIMS 函数讲解。

5.2 继 承

PostgreSQL 允许表从其他表中继承特性和属性。当许多表需要保存非常相似信息时，继承就大有用武之地。在这种情况下，通常可以创建一个父表用于保存通用数据结构，再让其他子表从此父表中继承这些结构。

例如，让我们创建一个名为 employees 的表：

```
CREATE TABLE employees (name char(10),salary numeric(9,2));
```

现在你可以创建一个指定的表 cooks，而且表中的 cooks 刚好需要其他 employee 的所有信息：

```
CREATE TABLE cooks (specialty char(10))INHERITS(employees);

SELECT *FROM cooks;

name salary specialty
---------------------
(0 rows)

INSERT INTO cooks VALUES ('Bill ',877.50,'Steak ');

SELECT *FROM cooks;

name salary specialty
-----------------------------
'Bill ' 877.50 'Steak '
```

继承的真正威力是它能够通过搜索父表来查找保存在所有子表中的信息，而不必在查询中指定子表的名称。

```
SELECT *FROM employees*WHERE name='Bill ';

name salary specialty
-----------------------------
'Bill ' 877.50 'Steak '
```

请注意上一个查询中在表名 employees 后包含一个星号(*)。这是为了告诉 PostgreSQL 将搜索范围扩展到子表中。

从 PostgreSQL 的 7.1 版本开始，缺省情况下，所有的查询都将搜索范围扩展到子表中。

虽然仍支持带星号（*）方式，但在表名后可以不带此额外的星号了。

在版本 7.1 中，为了限制查找仅针对特定表，可使用两个选项。一个选项用于设置环境变量 SQL_Inheritance 为 OFF；另一个选项指定在 SELECT 查询中使用 ONLY 关键字。例如：

```
SELECT *FROM ONLY employees WHERE name='Bill ';

name salary specialty
---------------------
(0 rows)
```

或者，你也可以使用 SET 命令：

```
SET SQL_Inheritance TO OFF;
SELECT *FROM employees WHERE name='Bill ';

name salary specialty
---------------------
(0 rows)
```

虽然表继承是 PostgreSQL 的一个强大功能，但也存在一些限制。这些限制来自于概念上的规划。

尽管并不是 PostgreSQL 本质上的一个限制，然而问题往往会发生，除非仔细规划表继承。譬如，在上一个例子中，就假定每个 cook 都是一个 employee。

当然，也可以规划成一种新的关系，而不是让 cook 归于 employee 类别之中。也许此时使用 volunteer 或 consultant 来描述这种关系更合适。在这一点上，以前的数据库规划存在缺陷，需要重新规划使之更加合理。正如前面提到的，这并不是 PostgreSQL 的一个本质性问题，它仅仅提醒我们在使用继承时需要仔细规划。

5.3　PostgreSQL 索引

实质上，索引的作用是帮助数据库系统更有效地搜索表。所有流行 RDBMS 广泛支持索引的概念，PostgreSQL 也不例外。

缺省情况下，PostgreSQL 可支持三类索引：B-Tree、R-Tree 及散列（hash）。创建索引时，需要指定索引的类型。每一个索引类型最适合于某一类型索引。

使用索引时，按一般经验来决定数据库使用的合适查询。实际上，索引总是存在于频繁使用的查询 WHERE 标准中。

B-Tree 索引

B-tree 索引是一个 Lehman-Yao 高并发 B-tree 的实现。B-Tree 是一个完全动态索引，它并不需要周期性地优化。

这是 PostgreSQL 最常用的缺省索引。事实上，如果调用 CREATE INDEX 命令时没有指

定索引类型，则会产生一个 B-Tree 索引。

使用下列比较操作符的任何时候都会应用 B-Tree 索引：

<,<=,=,=>,>

目前版本中，B-Tree 是唯一支持多列索引的索引。多达 16 个列可聚集到一个 B-Tree 多列索引中（虽然此限制在编译时可改变）。

R-Tree 索引

R-Tree 索引尤其适合于对几何和/或者空间关系比较的快速优化。R-Tree 索引是 Antonin Guttman 的二分算法的实现。R-Tree 索引是一个完全动态索引，它并不需要周期性地优化。在使用下列比较操作符的任何时候，R-Tree 索引将优先使用：

<<,&<,&>,>>,@,~=,&&

散列索引

散列索引是一个实现 Litwin 线性散列（哈希）算法的标准散列索引。散列索引是一个完全动态索引，它并不需要周期性地优化。

散列索引可用于使用了=比较操作符的任何时候。不过，没有实质性证据表明，在 PostgreSQL 中散列索引比 B-Tree 索引快。所以，大部分情况下，人们还是喜欢针对=比较操作符使用 B-Tree 索引。

其他索引主题

还必须谈谈索引的其他一些用途。也就是说，它们还可以用于函数的输出及多列情况下。

1. 函数索引

通常针对函数结果，可以在查询中使用函数索引。例如，如果你需要频繁访问一个表求一个字段的 MAX()值，就可以对函数创建一个单独的包含函数输出的索引：

```
CREATE INDEX max_payroll_idx ON payroll (MAX(salary));
```

当查询调用 WHERE MAX(salary)>n 或其他此类选择标准时，优化速度就会快得多。

函数索引不适用于多列索引。

2. 多列索引

PostgreSQL 中的 B-Tree 索引支持多列索引，且多达 16 个列范围。（这是一个编译时所用选项）大部分情况下，仅当查询中使用了 AND 操作符时才采用多列索引。

例如：

```
CREATE INDEX name_ssn_idx ON payroll (name,ssn);

SELECT *FROM payroll WHERE name='Bill ' AND ssn='555-55-5555 ';
```

上一例中使用了多列 name_ssn_idx。但下例中没有使用：

```
SELECT *FROM payroll WHERE name='Bill ' OR ssn='555-55-5555 ';
```

多列索引一般应尽量少用。大部分情况下，单列索引比多列索引在执行速度和保存空间上更显优势。

不过，当表中包含许多相似信息时，用多列索引来聚集特定行键很有效。它常常用于强

制数据的完整性。例如，假设一个表使用了如下字段：

```
Age Height Name
```

每个单独的字段难以强制加载任何特定约束。毕竟，存在高为 5'10''或年龄为 25 岁的人，但高为 5'10''、25 岁且名叫 Bill Smith 的人肯定很少。这种情况就可使用特定约束的多列索引来强制数据的完整性。

3. 主键与唯一键索引

产生混淆的原因是因为两种键类型的存在，表面上看，它们似乎功能相同。

主键和唯一键均使用索引强制规则要求字段值在表中唯一。不过，两者之间有一些重要的区别和细微的不同之处：

- 主键主要用于在表中将一个字段值与指定行（OID）相关联。这就是在 foreign 表联结中主键能用作关系键的原因。另外，主键不允许输入 NULL 值。
- 唯一键不能关联一个字段值与一个指定行，它们只强制在指定列上执行唯一的子句。虽然它对保持数据的完整性有用，但与主键对 foreign 表的关联性对比，唯一键并非是必需的。而且，唯一键一般允许插入 NULL 值。
- 在一个 UNIQUE NO NULL 和一个主键之间无功能上的区别。关键字 PRIMARY KEY 对用户只是作为一个记忆上的工具，用于提醒用户此索引约束的目的。

举一个两者区别的简单例子：假设在表 employee 中有两个重要的字段。一个字段是 employee_id，它由系统赋值，另外一个是 SSN，用于用户输入数据等。

在此情况下，employee_id 应指定为一个主键，SSN 应该指定为一个唯一键。

5.4　OID

PostgreSQL 使用对象标识（OID）和临时标识（TID）来关联表行与系统表及临时索引项。

在 PostgreSQL 中，插入表中的每一行都有一个唯一的 OID 与之关联。事实上，PostgreSQL 中的每个表包含一个名为 oid 的隐藏列。例如：

```
SELECT *FROM authors;

Name          Title
-----------------------------------------------------
Bill Smith        Cooking for 6 Billion
Sam Jones         Chicken Soup for the Publishers Soul

SELECT oid,*FROM authors;
Oid    Name          Title
-----------------------------------------------------
17887 Bill Smith  Cooking for 6 Billion
18758 Sam Jones   Chicken Soup for the Publishers Soul
```

理解 OID 概念的关键是，要清楚它们在表中并非连续。OID 在整个数据库中用于标识每一行，它们并非固定于一个表。所以，任何一个表不可能含有一个连续的 OID。SERIAL

数据类型或一个自排序的 SEQUENCE 是此类应用的最佳选择。

缺省时，PostgreSQL 中 OID 保留 0 到 16384 的范围仅供系统使用。所以，用户的表行所分配的 OID 总是大于此值。

PostgreSQL 还使用 TID 来构建行数据与索引间的动态联系。此值可变并且只能被内部系统使用。

一个通常的问题是如何创建一个表的准确拷贝，包含原始的 OID。使用 PostgreSQL 提供的 OID 数据类型可达到此目的。例如：

```
CREATE TABLE new_authors
     (orig_oid oid,name char(10),title char(30));
SELECT oid,name,title INTO new_authors FROM authors;
COPY new_authors TO '/tmp/newauth ';
     DELETE FROM new_authors;
COPY new_authors WITH OIDS FROM '/tmp/newauth ';
```

5.5 多版本并发控制

PostgreSQL 使用多版本并发控制（MVCC）来保持数据的一致性。至少从概念上理解 MVCC 是如何工作的对 PostgreSQL 管理员或开发者很有益。

大部分流行 RDBMS 使用表或行锁来保持数据库的一致性。典型情况下，这些锁发生在文件物理级别。这些锁用于阻止两个或多个事例同时写入同一行（或表）。

PostgreSQL 使用了一种更高级的方法来确保数据库的完整性。在 MVCC 中，每个事务可见一个过去最近点存在的数据库版本。防止事务看到真实的数据、而只看到前一版本中的数据很重要。处于并发状态时，这一机制可阻止当前事务处理由其他并发数据库事务产生的数据。实质上，一旦一个事务开始，则此事务被隔离。其中的数据结构也被隔离，不被其他事务操纵。一旦此事务结束，对相应数据库版本所作的更改就会融合到真实数据结构中。

任何 RDBMS 需要处理的并发问题有三类：

- **污染读**（Dirty Read） 一个事务读取了另一个未提交的并发事务写的数据。
- **错误读取**（Phantom read） 一个事务重新执行一个查询，返回一套符合查询条件的行，发现这些行中插入了被其他已提交的事务提交的行。
- **不可重复读**（Nonrepeat read） 一个事务重新读取前面读取过的数据，发现该数据已经被另一个已提交的事务修改过。

PostgreSQL 提供了 READ COMMITTED 和 SERIALIZABLE 保护隔离级别，如表 5-1 所示。

表 5-1 READ COMMITTED 和 SERIALIZABLE 保护隔离级别

级 别	污染读	错误读取	不可重复读
读已提交 (READ COMMITTED)	不可能	可能	可能
可串行化 (Serializable)	不可能	不可能	不可能

读已提交（READ COMMITTED）级

这是 PostgreSQL 的缺省保护隔离方法。READ COMMITTED 级防止查询查看事务启动后的数据变化。不过，此事务将看到以前进程中表所作的数据更改。

理解 READ COMMITTED 隔离的关键一点是清楚另一个事务运行 UPDATE、DELETE 或 SELECT FOR UPDATE 命令后的变化。这种情况下，只有部分隔离起作用。下面步骤描述了可能发生的情况：

1. 事务 A 将等待事务 B 的完成。
2. 如果事务 B 回滚（ROLLBACK），事务 A 则照常进行。
3. 如果事务 B 通过 COMMIT 命令完成，则事务 A 重新执行查询以确保是否条件改变，避免不必要的操作（如此行已被事务 B 删除）。
4. 如果行仍然匹配标准，则修改继续工作。（注意参见此条下一段说明。）
5. 行被重复修改，而且事务 A 中的其他待命语句继续执行。

必须注意的重要一点是第 4 步中所发生的。在这一步中，事务 A 使用了一个新版本的数据库。当事务 A 重新执行查询时，这种情况就会发生。此时，它使用了一个新版本数据库作为其基线。所以，事务 A 中的后续语句将作用于事务 B 更改过的数据上。所以，从这一点上讲，事务隔离仅仅是部分隔离。在与之类似的特定情况下，事务之间是可以互相“渗漏”的。

可串行化（Serializable）级

这一隔离级别与 READ COMMITTED 隔离级的不同之处在于所有事务必须按规定的串行方式发生。可能对其他事务产生修改的事务不能进行。

这将强制执行一个严格事务安排。根据此安排，若一个事务成功完成，则修改另一个事务的读缓冲，然后第二个事务自动执行 ROLLBACK。

这一隔离级别的实际效果是要求数据库系统做到事务失败后能随后重试。在一个负荷较重的系统上，这就意味着由于此严格的规定，很大比例的事务会导致失败。与采用 READ COMMITTED 隔离级别的数据库系统比起来，此负担会使系统运行慢得多。

大部分情况下，READ COMMITTED 隔离级是合适的。不过，也有一些查询要求有严格的方法来保证数据的有效性。

第三部分

PostgreSQL 管理

第 6 章　用户可执行文件

PostgreSQL 包含一些可执行文件来帮助配置和管理数据库系统。虽然本章题为“用户可执行文件”，但不要把此术语与“被任何用户执行”相混淆。通常，这些文件只能被 postgres DBA 帐号用户执行。而且，这些文件可以从一个客户机运行而不需要从保存有后端数据库系统的同一台计算机上运行。

这些文件的大部分操作应用可以通过执行一系列 SQL 命令达到同样效果。将它们做成独立的执行文件是为了帮助 DBA 执行常规系统任务。

下面的讲解同时指出了这些文件的位置。PostgreSQL 最常见的两种安装形式是从源代码安装或从部分 RPM 包安装。合适的安装类型将在下面每个命令的“注释”部分介绍。

文件列表（按字母排序）

createdb

描述

createdb 是 CREATE DATABASE SQL 子句的替代命令行。

用法/选项

```
createdb [options ]name [comment ]
```

参见表6-1。

表 6-1　　createdb 命令选项

选　项	描　述
-e,--echo	将后端信息返回到 stdout
-E,--encoding type	字符编码方式
-h,--host host	服务器所在主机名
-p, -port port	监听服务器的端口或套接字文件
-q,--quiet	禁止从后端返回任何应答
-U,--username user	以此用户名连接
-W,--password	提示输入密码
D,--location path	切换到数据库所在路径
name	创建数据库的名称
comment	对数据库的描述或解释

示例

```
$createdb mydatabase 'My database for holding records'
$createdb -h db.somewebsite.com -p 9333 mydatabase
$createdb -D /usr/local/mydb mydatabase
```

注释/位置

createdb 实际上完成的是在 psql 命令中创建数据库。

因此，系统中必须存在 psql 文件并正确执行以实现 createdb 功能。

文件位置：

RPM——/usr/bin

源文件——/usr/local/pgsql/bin

createlang

描述

createlang 用于在指定的 PostgreSQL 数据库中注册一个新语言。

用法/选项

```
createlang [options ] [language [dbname ]]
```

见表 6-2。

表 6-2　　createlang 命令选项

选　项	描　述
-h,--host host	服务器所在主机名
-p,--port port	监听服务器的端口或套接字文件
-U,--username user	以此用户名连接
-W,--password	提示输入密码
l,--list	列出指定数据库中目前注册的语言
language	在数据库中注册的语言名称
dbname	创建语言的所在数据库名称

示例

```
$createlang pltcl mydatabase
$createlang -h db.someserver.com -p 9999 plsql mydatabase
$createlang -l mydatabase
```

注释/位置

目前版本的 createlang 命令接受 plsql 或 pltcl。

此命令与 CREATE LAGUAGE SQL 命令相当。不过，在添加语言时应优先采用此方法，因为它自动执行一定的系统检查。

文件位置：

RPM——/usr/bin

源文件——/usr/local/pgsql/bin

createuser

描述

createuser 命令为 PostgreSQL 添加一个新用户。

用法/选项

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

见表6-3。

表 6-3　　createuser 命令选项

选　项	描　述
-e,--echo	将后端信息返回到 stdout
-h,--host host	服务器所在主机名
-p,--port port	监听服务器的端口或套接字文件
-q,--quiet	禁止从后端返回任何应答
-d,--createdb	允许用户创建新数据库
-D,--no -createdb	禁止用户创建新数据库
-a,--adduser	允许用户创建另外的用户
-A,--no -adduser	禁止用户创建新的用户
-P,--pwprompt	如果使用认证，则提示输入新用户的密码
-I,--sysid id	允许指定用户的 UID
username	创建唯一用户名

示例

```
$createuser joe
$createuser -h db.someserver.com -p 9999 joe
$createuser -a -d joe
```

注释/位置

createuser 是 psql 命令的集成。所以，系统中必须存在 psql 文件且能被用户执行。

要创建用户，必须设置执行用户的 pg_shadow 标记，以保证成功执行。

文件位置：

RPM——/usr/bin

源文件——/usr/local/pgsql/bin

dropdb

描述

dropdb 是 DROP DATABASE SQL 子句的替代命令行。

用法/选项

```
dropdb [options ]name
```

见表6-4。

表 6-4 **dropdb 命令选项**

选 项	描 述
-e,--echo	将后端信息返回到 stdout
-h,--host host	服务器所在主机名
-p,--port port	监听服务器的端口或套接字文件
-q,--quiet	禁止从后端返回任何应答
-U,--username user	以此用户名连接
-W,--password	提示输入密码
-i,--interactive	删除进程的交互性验证
name	删除数据库的名称

示例

```
$dropdb mydatabase
$dropdb -h db.somewebsite.com -p 9333 mydatabase
```

注释/位置

dropdb 依赖于 psql 命令来实际执行数据库删除动作。所以，psql 必须存在并且能正确执行以实现 dropdb 功能。

文件位置：

RPM——/usr/bin

源文件—/usr/local/pgsql/bin

droplang

描述

droplang 用于从指定的 PostgreSQL 数据库中删除语言。

用法/选项

```
droplang [options ] [language [dbname ]]
```

见表 6-5。

表 6-5 **droplang 命令选项**

选 项	描 述
-h,--host host	服务器所在主机名
-p,--port port	监听服务器的端口或套接字文件
-U,--username user	以此用户名连接
-W,--password	提示输入密码
-l,--list	列出指定数据库中目前注册的语言
Language	将要删除的语言名称
Dbname	删除此数据库中的指定语言

示例

```
$droplang pltcl mydatabase
```

```
$droplang -h db.someserver.com -p 9999 plsql mydatabase
$droplang -l mydatabase
```

注释/位置

此命令为 DROP LANGUAGE SQL 命令的集成，不过，删除语言时优先使用此方法，因为同时能自动进行系统检查。

文件位置：

RPM——/usr/bin

源文件——/usr/local/pgsql/bin

dropuser

描述

dropuser 命令删除 PostgreSQL 的一个用户。

用法/选项

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

见表6-6。

表 6-6 dropuser 命令选项

选　项	描　述
-e,--echo	将后端信息返回到 stdout
-h,--host host	服务器所在主机名
-p,--port port	监听服务器的端口或套接字文件
-q,--quiet	禁止从后端返回任何应答
-i, --interactive	删除前提示确认
Username	删除的唯一用户名

示例

```
$dropuser joe
$dropuser -h db.someserver.com -p 9999 joe
```

注释/位置

dropuser 是 psql 命令的集成。所以，psql 文件必须存在且能被用户执行。要删除用户，必须设置执行用户的 pg_shadow 标识，以保证成功执行。

文件位置：

RPM——/usr/bin

源文件——/usr/local/pgsql/bin

ecpg

描述

ecpg 命令是一个 SQL 预处理器，用于将 SQL 命令嵌入 C 程序内。在 C 程序中使用 SQL 命令实际上分两步过程。第一，SQL 命令文件传递给 ecpg，然后，通过标准 C 编译器来连接和编译它。

用法/选项

```
ecpg [options ]file [,el ]]
```

见表6-7。

表 6-7 **ecpg 命令选项**

选　项	描　述
-v	打印 ecpg 的版本信息
-t	关闭自动事务模式
-I *path*	指定替代包含路径
-d	关闭调试信息
-o *filename*	指定输出文件名；如果忽略，则缺省文件为 file.c
file	处理的文件

示例

```
$ecpg myfile.pgc
```

注释/位置

ecpg 命令的语法讨论超出了此节的范围。要了解有关在 C 程序中嵌入 SQL 的完整讨论，请参阅第 13 章“客户端编程”及其“ecpg”一节。

文件位置：

RPM——/usr/bin

源文件—/usr/local/pgsql/bin

pgaccess

描述

pgaccess 是一个图形化（GUI）交互前端，用于简化许多常规管理任务。

用法/选项

```
pgaccess[dbname]
```

dbname—启动连接到此数据库的 pgaccess。

注释/位置

pgaccess 具有以下功能：

- 打开任何数据库。
- 登陆时指定用户名和密码。
- 指定主机名和（或）连接端口。
- 本地保存优先权。
- 对数据库执行 VACUUM 命令。
- 提供实时性表数据修改。
- 删除表中选择行。
- 为表追加记录。
- 基于给定标准过滤行。
- 指定行的排列顺序。

- 导入/导出表数据。
- 表重命名。
- 删除表。
- 定义和编辑用户定义查询。
- 将查询保存为视图。
- 保存视图布局。
- 使用拖放支持构建查询。
- 用表别名构建查询。
- 在动态查询中提示用户输入参数（如“SELECT * FROM authors WHERE name=[parameter 'Authors Name']”）。
- 定义、检查和删除序列。
- 设计、查看、排序和删除视图。
- 定义、查看和删除函数。
- 定义和生成简单报表。
- 更改字体大小和报表样式。
- 装载和保存报表。
- 设计自定义表单。
- 保存和浏览表单。
- 定义、编辑和执行用户自定义脚本。

pgaccess 依赖于 Tcl/Tk 语言，所以需要安装 Tcl/Tk 来使之正常工作。而且还需安装 PostgreSQL-Tcl 包，或者需要通过--with -tcl 选项来编译源代码。

文件位置：

RPM——/usr/bin

源文件—/usr/bin 或/usr/local/pgaccess

pgadmin

描述

pgadmin 工具是一个用于进行基本 PostgreSQL 管理的 Windows 95/98/NT 工具。（基本 PostgreSQL 包中不包含此工具，它是针对 Windows 用户的第三方工具。）

注释/位置

此工具包括以下功能：

- 运行任意 SQL 命令。
- 创建数据库、表、索引、序列、视图、触发器、函数及语言。
- 授权用户和组权限。
- 数据导入和导出工具。
- 数据库、表、索引、序列、语言及视图的预定义报表。
- 版本跟踪。

标准 PostgreSQL 系统的发行版本中并不包含 pgadmin 工具。请访问 http://www.pgadmin.freeserve.co.uk 获取、安装和使用 pgadmin 工具的更详细信息。

pg_dump

描述

在 PostgreSQL 管理员工具箱中，pg_dump 是一个非常重要的工具。它允许数据库规划和（或）数据倾倒于标准文本。缺省情况下，它将数据写入 stdout，通过 stdout 运用适当的管道符号可以轻易地转向到一个文件。

与 psql 或 pg_restore 结合使用，就是进行数据库备份和恢复的优先方法（请参见下一部分内容“pg_dumpall”）。

PostgreSQL 7.1 版本对 pg_dump 和 pg_dumpall 命令添加了许多重要功能。添加了一个新选项允许按指定格式倾倒。一些新格式类型允许实现某些高级功能，如倾倒（dump）和恢复用户定义对象、选择性恢复等等（请参见“pg_restore”部分了解更为详细信息）。

用法/选项

```
pg_dump [options ]database
```

请参见表6-8。

表 6-8 **pg_dump 命令选项**

选　项	描　述
-h,host	启动服务器所运行的主机
-p,port	指定服务器运行的端口
-u	提示用户/密码认证
-v	指定冗余模式
-a	仅倾倒数据，无规划
-b,--blobs	倾倒数据和 BLOB（7.1 版本功能）
-c	创建前删除规划
-C,--create	包含创建数据库的命令（7.1 版本功能）
-d	以适当的 INSERT 格式倾倒数据
-D	作为 INSERT 及其属性名倾倒数据
-f,--file *name*	将输出信息发送到指定文件（7.1 版本功能）
-Fp	使用一个纯 SQL 文本格式。此为缺省设置（7.1 版本功能）
-Ft	以 tar 格式输出文档（7.1 版本功能）
-Fc	以新自定义的格式输出文档。这是最灵活的选项（7.1 版本功能）
-I	忽略与服务器后端版本的不匹配（pg_dump 只能工作于适当的版本上；仅供实验用）
-n	在 dump 中禁止双引号
-N	在 dump 中包含双引号（缺省）
-o	对所有表倾倒 OID
-O,--no -owner	禁止设置所有权对象来匹配原始数据库（7.1 版本功能）
-R,--no -reconnect	禁止与数据库的连接尝试（7.1 版本功能）
-S,--superuser *name*	当禁止触发器和设置所有权信息时指定使用的超级用户名（DBA）（7.1 版本功能）
-s	仅倾倒规划；无数据
-t *table*	仅倾倒该表信息
-x	禁止倾倒 ACI（授权/撤销）信息
-Z,--compress [0..9]	指定压缩级别（0-9）；目前版本中，仅有自定义格式支持这一功能（7.1 版本功能）

示例

```
$pg_dump authors
$pg_dump -a authors
$pg_dump -t payroll authors
```

注释/位置

pg—dump 不能处理大对象（LO）。

Pg—dump 不能正确抽取所有系统目录元数据。例如，不支持部分索引。

文件位置：

RPM——/usr/bin

源文件—/usr/local/pgsql/bin

pg_dumpall

描述

pg_dumpall 命令与 pg_dump 命令非常相似。不过，pg_dumpall 抽取所有数据库到一个脚本文件。除了在 pg_dump 命令中抽取的标准项目外，pg_dumpall 还包括 pg_shadow 文件内容。

PostgreSQL 7.1 版本对 pg_dump 和 pg_dumpall 命令添加了许多重要功能。添加了一个新选项允许按指定格式倾倒。一些新格式类型允许实现某些高级功能，如倾倒（dump）和恢复用户定义对象、选择性恢复等等（请参见“pg_restore”部分了解更为详细信息）。

用法/选项

pg_dumpall [*options*]

请参见表6-9。

表 6-9　　pg_dumpall 命令选项

选　项	描　述
-h,host	启动服务器所运行的主机
-p,port	指定服务器运行的端口
-u	提示用户/密码认证
-v	指定冗余模式
-a	仅倾倒数据，无规划
-b,--blobs	倾倒数据和 BLOB（7.1 版本功能）
-c	创建前删除规划
C,--create	包含创建数据库的命令（7.1 版本功能）
d	以适当的 INSERT 格式倾倒数据
-D	作为 INSERT 及其属性名倾倒数据
-f,--file name	将输出信息发送到指定文件（7.1 版本功能）
-Fp	使用一个纯 SQL 文本格式。此为缺省设置（7.1 版本功能）
-Ft	以 tar 格式输出文档（7.1 版本功能）
-Fc	以新自定义的格式输出文档，这是最灵活的选项（7.1 版本功能）
-I	忽略与服务器后端版本的不匹配（pg_dump 只能工作于适当的版本上；仅供实验用）
-n	在 dump 中禁止双引号

续表

选　项	描　述
-N	在 dump 中包含双引号（缺省）
-o	对所有表倾倒 OID
-O,--no -owner	禁止设置所有权对象来匹配原始数据库（7.1 版本功能）
-R,--no -reconnect	禁止与数据库的连接尝试（7.1 版本功能）
-S,--superuser *name*	当禁止触发器和设置所有权信息时指定使用的超级用户名（DBA）（7.1 版本功能）
-s	仅倾倒规划；无数据
-x	禁止倾倒 ACI（授权/撤销）信息
-Z,--compress [0..9]	指定压缩级别（0-9）；目前版本中，仅有自定义格式支持这一功能（7.1 版本功能）

示例

```
$pg_dumpall
$pg_dumpall -a
$pg_dumpall -o
```

注释/位置

pg_dumpall 对于系统元数据具有与 pg_dump 相同的限制。请参考“**pg_dump**”了解更多信息。

文件位置：

RPM——/usr/bin

源文件——/usr/local/pgsql/bin

pg_restore

描述

这是 PostgreSQL 7.1 版本中的一个新工具。它用于恢复由 pg_dump 或 pg_dumpall 数据库倾倒生成的数据。

新版本的 pg_dump 命令包含按非文本格式倾倒数据的能力，与传统数据倾倒相比，它具有诸多优点：

- 使用新版本的 pg_dump 格式和 pg_restore 命令可以选择性地恢复。
- 由 pg_dump 生成的新文档格式兼容于多平台。
- 新版本 pg_dump 格式生成的查询允许重新生成所有用户定义类型、函数、表、索引、聚集和操作符。

用法/选项

```
pg_restore [options ]archive-file
```

请参见表6-10。

表 6-10　　pg_restore 命令选项

选　项	描　述
-a,--data -only	仅恢复数据，不恢复规划
-c,--clean	在调用 createdb 前删除规划

续表

选　项	描　　述
-C,--create	包含创建规划的SQL命令
-d,dbname *name*	连接的数据库名称
-f,--file=*filename*	指定保存所生成输出结果的文件
-F t,--format=t	指定文档文件格式为tar
Fc,--format=c	指定文档文件格式pg_dump为自定义格式，这是最灵活的恢复格式
-i, index=name	仅恢复命名索引的信息
-l,--list	仅列出文档内容
-L,--use list file	恢复包含在指定文件中的元素。按其出现顺序恢复，注释行以分号（;）开头
-N,--orig-order	按原始dump顺序恢复条目
-o, -oid order	按原始OID顺序恢复条目
-O, -no owner	禁止恢复所有权信息；对象由当前用户所有
-P,--function name	仅恢复命名的函数
r,-rearrange	按修改OID顺序恢复条目（缺省）
-R,--no-reconnect	禁止pg_restore进行任何数据库连接。当已直接与数据库连接时，此选项有用
-s,-schema-only	仅恢复规划；不恢复数据
-S,--superuser=*name*	当禁止触发器和应用所有权信息时指定超级用户名。缺省情况下，如果当前用户为DBA，pg_restore使用当前用户名
-t *table=name*	仅对指定表恢复规划/数据
-T,--trigger=*name*	仅恢复指定触发器
-v,-verbose	按冗余输出
-x,--no-acl	禁止访问控制列表（ACI）的恢复（即授权/撤销信息）
-h,--host *name*	指定服务器进程运行的主机名
-p,--port *port*	指定所连接端口
-u	强迫进行认证

示例

倾倒一个数据库且使用自定义格式恢复：

```
$pg_dump -Fc newriders $newriders.cust_fmt
$pg_restore -d newriders newriders.cust_fmt
```

仅恢复表payroll表：

```
$pg_restore -d newriders -t payroll newriders.cust_fmt
```

注释/位置

参见pg_dump、pg_dumpall和pg_upgrade部分的讨论。

文件位置：

RPM——/usr/bin

源文件——/usr/local/pgsql/bin

pg_upgrade

描述

pg_upgrade 用于升级数据库系统到一个新版本，而不必重载当前数据库中的所有数据。

目前，PostgreSQL 7.1 及以上版本并不支持此命令。如果你正使用较新版本 PostgreSQL，请参见“pg_restore”部分介绍。

用法/选项

```
pg_upgrade [-f file] old_data_dir
```

-f *file*—指定包含旧数据库规划的文件。

old_data_dir—表示旧数据目录路径。

示例

使用 pg_upgrade 升级数据库的常用方法如下：

（1）备份现有数据（即 pg_dumpall）。

（2）将计划倾倒到一个文件（即 pg_dumpall -s db.out）。

（3）终止当前 postmaster。

（4）对旧数据目录重命名（即 data.old）。

（5）用 make 工具构建新二进制码。

（6）用 make install 安装新二进制码。

（7）在新系统中运行 initdb 来创建一个新数据库结构。

（8）开始 postmaster。

（9）使用$ pg_upgrade -f db.out /usr/local/pgsql/data.old 将旧数据库升级为一个新系统。

（10）复制旧 pg_hba.conf 文件和 pg_options 到新位置（即/usr/local/pgsql/data）。

（11）再次终止和开始 postmaster。

（12）验证连接正常工作。

（13）连接到恢复数据库并仔细检查其内容。

（14）如果数据库无效，从第 1 步中创建的完全倾倒文件中恢复。

（15）如果数据库有效，运行一个 VACUUM 命令来更新查询计划统计（即 cacuumdb -z mydb）。

注释/位置

并不是所有升级都能通过此工具完成。检查新数据库版本注释来查看是否支持 pg_upgrade。

文件位置：

RPM——/usr/bin

源文件—/usr/local/pgsql/bin

pgtclsh

描述

pgtclsh 命令是一个集成程序，它实质上是一个标准装载了 libpgtcl 库的 Tcl shell。

用法/选项

```
pgtclsh [script arg1 [,…]]
```

script—欲处理的可选 Tcl 脚本文件。

arg1—传递到指定脚本文件的可选参数。

示例

```
$pgtclsh myfile.tcl
```

注释/位置

如果不指定脚本文件直接运行 pgtclsh，则它会自动进入 Tcl 交互接口。

文件位置：

RPM——/usr/bin

源文件—/usr/local/pgsql/bin

pgtksh

描述

pgtksh 命令实质上是一个装载了 libpgtcl 库的 Tk(wish) shell。这是 pgaccess 程序的基础。

用法/选项

```
pgtksh [script arg1 [,…]]
```

script—欲处理的可选 Tcl 脚本文件。

arg1—传递到脚本文件的可选参数。

示例

```
$pgtksh myfile.tcl
```

注释/位置

若不指定脚本文件直接运行 pgtclsh，则它会自动进入 Tcl 交互接口。

文件位置：

RPM——/usr/bin

源文件—/usr/local/pgsql/bin

psql

描述

psql 是 PostgreSQL 系统的一个交互式前端工具。psql 的接口功能强大，它具有几乎能控制 PostgreSQL 系统方方面面的无数选项。

一旦启动 psql 并连接到指定数据库上，用户就进入了交互式接口。在此模式下，命令可以 PostgreSQL 后端执行，并且可实时看见其反应。

用法/选项

```
psql [options ] [database [user ]]
```

psql 选项分为两类：一是命令行选项（表 6-11），它在启动 psql 时使用；二是 shell 选项（表 6-12），它可以在 psql shell 内使用。

表 6-11 psql 命令行选项

命令行选项	描 述
-a, --echo -all	将处理行返回到屏幕。当运行一个脚本来监视过程或用于调试时可用此选项
-A, --no -align	将输出切换到一个非对齐布局
-C, --command *query*	指定 psql 执行单个 SQL 命令或单个交互性 shell 命令。此查询必须为纯 SQL 或 psql shell 命令。不允许使用混合类型
-d, --dbname *database*	指定所连接的数据库
-e, --echo -queries	返回发送到后端的所有查询
-E, --echo -hidden	作为 shell 命令（如\dt 等）结果返回所有查询，甚至包括隐藏查询
-f, --file *file*	读取指定文件并执行其中包含的 SQL 查询。完成后终止此过程。
-F, --field -separator *sep*	使用指定字段分隔符
-h, --host *hostname*	连接到指定主机（服务器所运行主机）
-H, --html	以 HTML 格式生成表输出
-l, --list	列出现有的所有可用数据库
-o, --output *file*	用指定文件捕获所有查询输出
-P, -port *port*	使用指定端口连接
-p, --pset *val*	允许设置缺省打印（输出）方式（如 aligned、unaligned、html 或 latex）
q	安静模式
-R, --record -separator sep	在记录中使用指定分隔符
-s, --single -step	在每个查询执行前提示用户。用于调试或控制 SQL 脚本的执行
-S, --single -line	以单行方式运行 PostgreSQL，用回车终止查询（缺省是用分号）
-t, --tuples -only	关闭列名及所有结果的打印。仅打印返回的数据（元组）
-T, --table-attr options	指定在 HTML 表输出中包含的选项
-u	强行提示输入用户名与密码
-U, --username name	以指定用户连接到数据库
-v, --variable --set var=val	给变量赋值。若不设置变量，则在变量后不带等号或值
-V, --version	显示 psql 版本信息
-W, --password	强迫 PostgreSQL 提示用户输入密码
-x, expanded	打开扩展行格式模式
-X, --no-psqlrc	禁止读起始文件-/.psqlrc
-?, --help	弹出帮助屏幕来显示 psql 选项

在 psql shell 中，大部分选项在前面冠有反斜杠（\）。

表 6-12　psql shell 选项

shell 选项	描　　述
\a	在打印表元素时切换字段对齐模式（开或关）
\C *title*	在每个查询结果集起始处设置指定的标题
\c, \connect *db [user]*	关闭当前连接并连接到指定数据库上。可以指定用户进行连接
\copy table [with oids] {from | to} filename | stdin | stdout [using delimiters char] [with null as nullstr]	执行前端版本的 SQL COPY 命令。指定复制方式及是否转向某个文件或是使用 stdin 或 stdout。而且还可以指定 null 字符的分隔符
\copyright	显示 PostgreSQL 版权信息
\d	与\dtvs 相同。显示指定关系信息
\dt	显示当前数据库中表的信息
\dv	显示当前数据库中视图的信息
\ds	显示当前数据库中序列的信息
\di	显示当前数据库中索引的信息
\da *[pattern]*	显示当前数据库中聚集的信息。也可以只显示与指定模式（如 max）匹配的信息
\dd *[obj]*	显示当前数据库中与所有对象关联的注释。也可以只显示与指定对象相关联的注释
\df *[pattern]*	显示当前数据库中函数的信息。也可以只显示与指定模式匹配的函数信息
\dl	列出当前数据库中的所有大对象（与\lo_list 同）
\dp *[pattern]*	显示当前数据库中与对象权限有关的信息。也可以只显示与指定模式匹配的对象信息（同\z）
\dS	显示当前数据库中与系统表有关的信息
\dT *[pattern]*	显示当前数据库中数据类型的信息。也可以仅显示与指定模式匹配的对象的信息
\e, \edit *file*	启动外部编辑器（缺省为 vi）来编辑指定文件
\echo *text*	返回指定文本或执行替代命令
\encoding *type*	设置编码为指定类型，或如果无参数则列出当前编码类型
\f *str*	将指定字符串设置为字段分隔符。缺省为管道符号（|）（参见 psql）
\g *[file | command]*	将查询输出发送到指定的文件或通过指定命令管道输出（与\o 相似）
\h, \help *[command]*	显示所有有效 SQL 命令的列表。也可以显示指定命令的更详细帮助
\i *file*	在指定文件中读取输入并执行
\l, \list	列出所有已知的数据库及其所有者（若包含一个“+”则同时显示注释）
\lo_export *oid file*	根据指定 OID 导出大对象到指定文件名中
\lo_import file *[comment]*	根据指定的文件名导入大对象。也可以提供一个与 LO 关联的描述性注释
\lo_list	列出当前数据库中的所有已知大对象
\lo_unlink oid	根据指定 OID 从当前数据库中删除大对象
\o [file | command]	将未来的所有查询结果发送到指定的文件名或通过给定的命令管道输出
\p	打印当前的查询缓存

续表

shell 选项	描　　述
\pset parameter	允许用户手工设置下列影响当前数据库的若干参数之一： format 按指定格式设置表的输出方式：unaligned、aligned、html 或 latex（即\pset format=latex） border 设置边框宽或类型。在 HTML 中，0 表示无边框，1 表示虚线，2 表示新框架（即\pset border=0） expanded 在常规和扩展格式间切换 null 指定显示 NULL 字段值的方式（即\pset null 'N/A'） fieldsep 指定用作字段分隔符的字符（即\pset fieldsep '#'） recordset 指定用作记录分隔符的字符（即\pset recordsep '%'） tuples_only 去掉查询结果显示中的头和尾信息。仅从查询中返回数据 title 指定用作后面表的标题（即\pset title 'Our Bank Account'） tableattr 指定包含在 HTML 输出中的属性（即\pset tableattr bgcolor="#FFFF00"） pager 切换使用逐页显示。缺省情况下，使用 more 来处理页面显示，但可定义 PAGER 变量为适当的句柄
\q	退出当前 psql shell
\qecho text	在当前\o 输出定向处回显指定文本。用于在重定向输出文件中添加注释
\r	清除（重置）当前查询缓存
\s file	在指定文件名中保存当前 psql 命令历史。（注意：PostgreSQL 7 及以上版本在退出时自动完成）

续表

shell 选项	描　　述
\set var value	设置 psql 环境变量为指定值。(注意：它与 SQL SET 命令不同) 下面是有效环境变量列表： DBNAME 当前连接数据库的名称 ECHO psql 当前所设置的回显方式。all 表示回显所有输出；queries 表示仅回显查询输出 ECHO_HIDDEN 指定是否在 stdout 回显隐藏查询（即如\dt 的隐藏查询） ENCODING 指定命名的编码方式。如果不能使用多字节编码，则此设置为 SQL_ASCII HISTCONTROL 控制进入命令历史缓存中的内容。ignorespace 将忽略以空白开始的命令，ignoredups 将拒绝输入重复项，ignoreboth 则包含以上两种情况 HISTSIZE 保存于历史缓存中的命令数量（缺省数是 500） HOST 当前连接所运行的主机 IGNOREEOF 如果不进行设置，则发送一个 EOF（Ctrl+D）来终止当前 psql 进程否则，若设置成一个数字变量，它将在终止前忽略插入的 EOF（缺省数字是 10） LASTTOID 最后受影响的 OID 值 LO_TRANSACTION 指定执行大对象事件时采取的动作（如：\loexport，\lo_import 等等）。rollback 值将强迫回滚到进程中的任一个事务。commit 将强迫执行 commit 命令，nothing 则指定无任何动作发生。当 LO 事件中封装了显式 BEGIN…COMMIT 时，后一种选项值被保留 ON_ERROR_STOP 指定若非交互性脚本遇到错误则终止进程，缺省时，即便遇到了不正常的 SQL 语句，psql 仍将继续执行语句

续表

shell选项	描　　述
\set var value	PORT 当前对话所连接的端口 PROMPT1 指定正常提示的样式（缺省为%/%R%#） PROMPT2 psql希望得到更多数据时发出提示（缺省为%/%R%#） PROMPT3 当调用一个SQL COPY且接口需要输入元组时发出提示（缺省为>>）。 提示类型： %M—完整主机名 %m—删剪过的主机名 %>—端口号 %n—连接用户名 %/—当前数据库 %~—与%/相似，但如果为缺省数据库则带一个前缀~ %#—如果为DBA则带一个前缀#，否则为>>>

示例

启动psql，执行一个查询并立即退出：

```
$psql -c 'SELECT *FROM authors ' newriders

name      |   age
----------------
Sam       |   25
Bill      |   67
```

要从命令行运行整个脚本文件mydb.sql（针对数据库newriders）：

```
$psql -f mydb.sql newriders
```

用同样的例子，但这次用HTML方式显示（对CGI编程有用）：

```
$psql -H -c 'SELECT *FROM authors ' newriders
<table border=1>
<tr>
<th align=center>name</th>
<th align=center>age</th>
     </tr>
     <tr valign=top>
         <td align=left>Sam</td>
         <td align=left>25</td>
     </tr>
     <tr valign=top>
         <td align=left>Bill</td>
         <td align=left>67</td>
     </tr>
</table>
```

要从psql shell将查询重定向到一个输出文件，而且对每个数据倾倒都要包含描述性标题：

```
psql=>\o mycapture.txt
psql=>\qecho Listing of all authors
psql=>\qecho **********************
psql=>SELECT *FROM authors;
psql=>\qecho And their payroll info
psql=>\qecho **********************
psql=>SELECT *FROM payroll;
```

从psql shell接口列出当前目录中所有以.sql结尾的文件（注意反引号的使用）：

```
psql=>\echo 'ls *.sql '
authors.table.sql
payroll.table.sql
```

注释/位置

psql shell环境还支持变量替换。最基本的方式是用一个值与一个变量名关联，如：

```
psql=>\set myvar name='Sam '
psql=>\echo :myvar
```

```
psql=>name='Sam '
psql=>SELECT *FROM authors WHERE :myvar;
name      |   age
Sam       |   44
```

从中可以看到，通过在名称前加一个冒号（:）来引用变量名：

要在使用\e 命令时改变缺省的编辑器，就必须给 PSQL_EDITOR 变量指定正确的值。

文件位置：

RPM——/usr/bin

源文件—/usr/local/pgsql/bin

vacuumdb

描述

vacuumdb 命令是 VACUUM SQL 语句的等效程序。虽然两者在执行上无真正区别，但当通过一个 cron 任务运行时，通常选择 vacuumdb 命令。

用法/选项

```
vacuumdb [connection-options ] [analyze options ]
```

请参见表6-13。

表 6-13　　vacuumdb 命令选项

连接选项	描　　述
-h, --host host	服务器所在主机名
-P, --port *port*	监听服务器的端口或套接字文件
-U, --username *user*	以指定用户身份连接
-W, --password	强迫提示输入密码
-e, --echo	回显后端信息到 stdout
-q, --quiet	禁止从后端返回任何响应
-d, --dbname *name*	清理数据库的名称
-z, --analyze	为查询优化器计算统计值
-a, --alldb	清理所有数据库
-v, --verbose	冗余输出
-t, --table *table*	仅清除或分析表
-t, --table *table(col)*	仅分析列（必须使用-z）

示例

清除 newriders 数据库（两者等价）：

```
$vacuumdb -d newriders
$
$vacuumdb newriders
```

清除所有数据库然后分析指定表：

```
$vacuumdb -a
$
```

```
$vacuumdb -z -d newriders -t authors
```

注释/位置

文件位置：

RPM——/usr/bin

源文件—/usr/local/pgsql/bin

第 7 章　系统可执行文件

虽然大部分系统可执行文件能通过用户帐号进行执行，但为便于使用，此类文件被单独罗列。与第 6 章“用户可执行文件”中讨论的日常使用命令不同，它们通常用于处理特定的数据库系统事件。一般，这些文件用于服务器控制而不是客户端应用。

下面介绍中还列出了所讨论命令的典型文件位置。根据数据库系统安装方式的不同（根据源代码安装或根据 RPM 包安装），这些文件的位置会不尽相同。

7.1　文件列表（按字母排序）

initdb

描述

initdb 命令用于为新 PostgreSQL 系统准备一个目录位置。initdb 命令通常分几步执行，简单描述如下：

1.以 root 创建一个空数据库目录来装载数据。

2.用 chown 将目录所有权交给 DBA 用户。

3.用 login（或 su）登陆到 DBA 用户帐号。

4.按合适选项执行 initdb 命令。

5. initdb 生成共享目录表。

6. initdb 生成 template1 数据库。（每次创建一个新数据库均从 template1 生成）

用法/选项

```
initdb -D path [options ]
```

请参见表7-1。

表 7-1　　initdb 选项

选　项	描　　述
-D, --pgdata *path*	PostgreSQL 数据库的路径
-i, --sysid=*id*	指定 DBA 的 UID
-W, --pwprompt	强制密码提示
-E, --encoding=*type*	指定所用的编码类型（系统的多字节编码标志应该设置为 true）
-d, --debug	从后端打印调试信息

续表

选　项	描　　述
-n, --noclean	缺省时，如果 initdb 执行失败，则它自动清除所创建的所有文件。此选项可阻止其他清除动作的发生
L, --*path*	指定 initdb 查找输入文件的路径（这是一个特殊情况选项，很少需要使用）

示例

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

注释/位置

initdb 命令不能以 root 身分运行。

文件位置：

RPM——/usr/bin

源文件—/usr/local/pgsql/bin

initlocation

描述

initlocation 用于创建一个初始数据存储区。在某种程度上，此命令与 initdb 命令相似，只是运行 initlocation 命令时不发生许多内部目录操作。另外，此命令可根据需要随意运行，而 initdb 命令一般只在每次安装时运行一次。

用法/选项

```
initlocation path
```

path—新数据库存储的路径。

示例

```
$initlocation /usr/local/pgsql/data2
```

注释/位置

此命令必须以 DBA 帐号而不是 root 身份运行。

文件位置：

RPM——/usr/bin

源代码——/usr/local/pgsql/bin

ipcclean

描述

ipcclean 命令是一个 shell 脚本，用于后端服务器退出后清除孤立信号和共享内存。

用法/选项

```
ipcclean
```

注释/位置

此命令对 ipcs 输出格式的命名约定作了一定的假设。所以，此 shell 脚本不可能对所有操作系统兼容。

警告

当一个数据库服务器正运行时若运行 ipcclean 命令，可能导致数据库系统的常规崩溃。

pg_ctl

描述

pg_ctl 用控制 PostgreSQL postmaster 服务器的许多不同方面特性。

用法/选项

```
pg_ctl [-w ] [-D path ] [-p path ] [-o "options "] start
pg_ctl [-w ] [-D path ] [-m mode ] stop
pg_ctl [-w ] [-D path ] [-m mode ] [-o "options "] restart
pg_ctl [-D path ] status
```

请参见表7-2。

表 7-2 **pg_ctl 选项**

选　项	描　　述
-W	监视 pid 文件的创建或删除（$PGDATA/postmaster.pid）。60 秒后停止
-D *path*	数据库路径
-p *path*	指定 postmaster 文件路径
-m *mode*	指定下面关闭模式之一： s，smart 等待客户退出（缺省） f，fast 发送 SIGTERM 到后端，活动事务立即回滚（ROLLBACK） i，immediate 发送 SIGUSR1 到所有后端；在此模式下，下一个系统启动时需要恢复数据库
-o "options"	发送 postmaster 指定选项。为了确保正确执行，选项一般用引号包含
start	启动 postmaster
stop	终止 postmaster
restart	重启动 postmaster（自动终止/启动）
status	显示 postmaster 状态

示例

```
$pg_ctl start
$pg_ctl -m smart stop
```

注释/位置

文件位置：

RPM—/usr/bin

源代码—/usr/local/pgsql/bin

pg_passwd

描述

在 PostgreSQL 中需要认证时，可用 pg_passwd 命令创建和操纵所需的密码文件。

用法/选项

```
pg_passwd filename
```

filename—创建和操纵的密码文件路径和文件名。

示例

```
$pg_passwd /usr/local/pgsql/data/pg_pword
File "/usr/local/pgsql/data/pg_pword " does not exist,,Create?(y/n):Y
Username:barry
Password:
Re-enter password:
```

注释/位置

密码文件必须位于用于客户认证的 PostgreSQL 数据库路径之中。另外，该认证方法必须在 pg_hha.conf 配置文件中声明。

文件位置：

RPM—/usr/bin

源代码—/usr/local/pgsql/bin

postgres

描述

postgres 文件是 PostgreSQL 中处理查询的实际服务器进程。它常常由多进程 postmaster 调用。(两者都是同一个文件，postmaster 是 postgres 进程的一个符号连接)

通常不直接启动 postgres 服务器，只是要执行时，将许多选项传递给 postgres 进程。

尽管不直接调用它，但可以交互方式来执行 postgres 进程，这样就允许输入查询并执行。不过，如果正在运行 postmaster 进程，则不要如此操作，因为可能导致数据崩溃。

用法/选项

```
postgres [options ]database
```

请参见表7-3。

表 7-3　postgres 选项

选　项	描　述
-A 0|1	是否允许检查声明（编译时只有打开这一选项才可以使用此调试工具，如果编译时打开了此选项，则缺省值是允许）
-B *val*	所使用的 8KB 共享缓存数。缺省值是 64
-c *var=val*	设置不同的运行过程选项。请参见本章后面的“高级选项”列表
-d *level*	设置调试级别。值越高，输出到日志中的项目越多。缺省值是 0；有效范围可达 4
-D *path*	数据所在目录

续表

选 项	描 述
-F	禁止 fsync 系统调用，可提高操作性能，但有数据崩溃的危险。一般只是具有充足的理由时才使用此选项；标准操作中不推荐使用
-e	将数据类型设置为欧洲样式（即 dd-mm-yyyy 样式）
-o *file*	将所有调试信息发送到指定文件
-P	禁止使用系统索引的元组扫描/修改（注意：REINDEX 命令需要使用这一项）
-s	对每个所处理查询发送时间统计数据到 stdout，用于操作性能的优化
-S *val*	指定系统调用临时文件前用于内部排序和散列的 KB 数。内存数表示每个系统排序和（或）散列可用的内存数。当系统处理复杂排序时，将使用多排序/散列，而每个排序/散列将占用此数量的内存。缺省值是 512KB
-E	在 stdout 上回显所有查询
-N	禁止使用新行作为查询分隔符

并非每个人都适合使用

这些高级选项（表 7-4）不推荐用于一般用途。它们只适用于高级调试场合或由 PostgreSQL 开发员使用。而且，在各个版本中包含这些选项可能情况不尽相同。不能保证这些选项在 PostgreSQL 的某一版本中存在。

表 7-4 postgres 高级选项

高级选项	描 述
-fi	禁止索引扫描
-fs	禁止序列扫描
-fn	禁止嵌套循环联结
-fm	禁止融合联结
-fh	禁止散列联结
-i	阻止执行查询但显示规划
-L	禁止使用系统锁
-O	允许修改系统表
-p *database*	表示指定数据库已由 postmaster 启动，影响缓存大小、文件描述符等
-tpa	为系统解析器打印时间信息（不能带-s 选项使用）
-tpl	为系统规划器打印时间信息（不能带-s 选项使用）
-te	为系统运行程序打印时间信息（不能带-s 选项使用）
-v *val*	指定所使用的协议版本
-W *sec*	在启动前睡眠指定秒数，用于开发人员需要在中途启动调试程序的情况

注释/位置

当开始 postgres 进程后，当前 OS 用户名就选择为 PostgreSQL 的用户名。如果当前用户名是一个无效 PostgreSQL 用户，则此进程终止。

postgres 和 postmaster 是同一个文件（实际上，postmaster 是 postgres 执行文件的符号连接）。不过，你不能用一个命令替换另一个命令并期望两者结果相同。Postgres 执行文件注册它所调用的名称，而且如果此调用者为 postmaster，则可以指

定某些选项和假设。

通过配置文件，许多选项能被（或已被）传递给 postgres 进程。（请参见第 8 章“系统配置文件和库”中的“pg_options/postgresql.conf”一节的内容）

文件位置：

RPM——/usr/bin

源代码——/usr/local/pgsql/bin

postmaster

描述

postmaster 是 postgres 应用的多用户实现。大部分情况下，此进程从启动时间开始，并且将日志文件转向到一个合适的文件中。

一个 postmaster 事例要求管理每个数据库簇。通过指定单独的数据位置及连接端口可启动多事例。

用法/选项

请参见表 7-5。

表 7-5　postmaster 选项

选　项	描　述
-A 0|1	是否允许检查声明（编译时只有打开这一选项才可以使用此调试工具，如果编译时打开了此选项，则缺省值是允许）
-B val	所使用的 8KB 共享缓存数，缺省值是 64
-b path	指定后端执行路径（通常 postgres）
-c var=val	设置不同的运行过程选项，参见下面列表： shared_buffers=integer debug_level=integer fsync=BOOLEAN virtual_host=integer tcpip_socket=BOOLEAN unix_socket_directory=integer ssl=BOOLEAN max_connections=integer port=integer enable_indexscan=BOOLEAN enable_hashjoin=BOOLEAN enable_mergejoin=BOOLEAN enable_nestloop=BOOLEAN enable_seqscan=BOOLEAN enable_tidscan=BOOLEAN sort_mem=integer show_query_stats=BOOLEAN show_parser_stats=BOOLEAN show_planner_stats=BOOLEAN show_executor_stats=BOOLEAN （注意：参见第 8 章的配置选项了解以上设置的更多信息）
-d level	设置调试级别，值越高，输出到日志中的项目越多，缺省值是 0；有效范围可达 4

续表

选　项	描　　述
-D path	数据所在目录
-F	禁止 fsync 系统调用。可提高操作性能，但有数据崩溃的危险。一般只是具有充足的理由时才使用此选项；标准操作中不推荐使用
-h host	指定服务器响应查询的主机。缺省监听所有配置接口
-i	允许客户通过 TCP/IP 连接。缺省情况下，在 UNIX 上允许域套接字
-k path	指定 postmaster 监听 UNIX 域套接字所使用的目录（缺省为/tmp）
-l	允许使用 SSL 连接（注意：使用此选项要求编译时打开了 SSL 而且使用了-i 选项）
-N val	指定允许与数据库后端的最多连接。缺省值是 32，但如果系统支持更多进程的话，也可高达 1,024（注意：为使正常，选项-B 必须至少是选项-N 的两倍）
-o options	传递给 postgres 后端的命令行选项。如果选项字符串包含空白，则必须使用引号（注意：参见 postgres 的有效命令行开关）
-P port	开始连接监听的 TCP/IP 端口。缺省端口为 5432 或编译时的设置值（注意：如果设置为非缺省端口，所有客户应用程序都必须指定此端口号来顺利连接）
-S	从当前终端（如 daemon）以单独进程启动 postmaster。不过，所有的错误信息都将转向/dev/null 而不是 stdout（注意：使用此选项调试工作几乎不可能进行。最好作为后台进程启动 postmaster，然后将错误信息重定向到指定文件，请参见下面章节的例子）

示例

以前台进程启动 postmaster：

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

以后台进程启动 postmaster，并指定数据目录及将所有错误信息定向到指定的日志文件中：

```
$postmaster -D /usr/local/pgsql/data >pglog 2>&1 &
```

注释/位置

当启动一个 postmaster 进程时，则将当前 OS 用户名作为 PostgreSQL 用户名。如果当前用户名并非一个有效的 PostgreSQL 用户，则此进程终止。

postgres 和 postmaster 是同一个文件（实际上，postmaster 是 postgres 执行文件的符号连接）。不过，你不能用一个命令替换另一个命令并期望两者结果相同。Postgres 执行文件注册它所调用的名称，而且如果此调用者为 postmaster，则可以指定某些选项和假设。

文件位置：

RPM—/usr/bin

源代码—/usr/local/pgsql/bin

第 8 章　系统配置文件和库

除了在前几章中提到的可执行文件外，PostgreSQL 还包括调节配置设置的文件。而且，根据所需功能，还可以包含许多库。下面列出了 PostgreSQL 中所包含的配置文件及库文件。

8.1　系统配置文件

pg_options/postgresql.conf

描述

配置文件用于定义服务器作为 postmaster 启动时所用的选项。文件名随版本的不同而不同，可以是 pg_options、postmaster.opts 或 postgresql.conf，完全取决于当前使用的版本。配置文件中的准确的语法及可用选项也依版本的不同而不同。

注释/位置

实质上，此文件是一个文本文件，它包含了不同的命令行开头。一个标准的配置文件可能如下所示：

```
-p 5432                          →Use TCP/IP port 5432
-D /usr/local/pgsql/data             →Path to data directory
-B 64                            →Start with 64 8KB buffers
-b /usr/local/pgsql/bin/postgres     →Path to executable
-N 32                            →32 Max connections
```

配置文件的准确语法随 PostgreSQL 运行版本的不同而不同。postgresql.conf 文件（版本 7.1 中使用的文件）所接受的选项有：

CHECKPOINT_SEGMENTS(integer)

自动 WAL（预写式日志）检查点间的最大距离。

CHECKPOINT_TIMEOUT(integer)

自动 WAL 检查点间的最长时间，以秒为单位。

CPU_INDEX_TUPLE_COST(floating point)

索引扫描中，设置处理每个元组的估计开销。

CPU_OPERATOR_COST(floating point)

设置处理 WHERE 子句中每个操作符的估计开销。

CPU_TUPLE_COST(floating point)

设置序列扫描中处理一个元组的估计开销。

DEADLOCK_TIMEOUT(integer)

以微秒为单位设置在检查是否存在死锁条件前等待锁的时间。

DEBUG_ASSERTIONS(boolean)

是否允许不同调试声明检查的布尔值。

DEBUG_LEVEL(integer)

此值决定调试输出的冗余方式。缺省值是 0，它意味着无调试输出。有效值范围可到 4。

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

指定调试信息中打印内容。打印查询、解析树、执行计划或在服务器日志中查询重写输出。

EFFECTIVE_CACHE_SIZE (floating point)

设置假定磁盘缓存大小。根据磁盘页面进行衡量（通常每个 8KB）。

ENABLE_HASHJOIN (boolean)

是否允许散列联结的布尔值。缺省值为允许。

ENABLE_INDEXSCAN (boolean)

是否允许使用索引扫描规划类型的布尔值。缺省值为允许。

ENABLE_MERGEJOIN (boolean)

是否允许使用融合联结规划类型的布尔值。缺省值为允许。

ENABLE_NESTLOOP (boolean)

是否允许使用嵌套联结规划的布尔值。不可能完全禁止嵌套联结，但是这一变量可约束规划器的使用。

ENABLE_SEQSAN (boolean)

是否允许使用序列扫描规划类型的布尔值。不可能完全禁止序列扫描，但是这一变量可约束规划器的使用。

ENABLE_SORT (boolean)

是否允许使用排序步骤的布尔值。不可能完全禁止排序，但是这一变量可约束规划器的使用。

ENABLE_TIDSCAN (boolean)

是否允许使用 TID 扫描类型的布尔值。缺省值为允许。

FSYNC (boolean)

是否允许 PostgreSQL 在几个地方使用 fsync()系统调用来确定更改已写入磁盘并且没有滞留在核心缓存器的布尔值。这将增加操作系统或硬件崩溃后多次不能安装数据库的机会。不过，使用这一选项将降低系统操作性能。缺省值是不允许。

GEQO (boolean)

是否允许基因查询优化的布尔值。缺省值是允许。

GEQO_EFFORT (integer), GEQO_GENERATIONS (integer), GEQO_POOL_SIZE (integer), GEQO_RANDOM_SEED (integer), GEQO_SELECTION_BIAS (floating point)

针对基因查询优化算法的不同调节参数。

GEQO_THRESHOLD (integer)

指定使用 CEQO 优化前的 FROM 项目。缺省值是 11。

HOSTNAME_LOOKUP (boolean)

指定是否转换 IP 地址为主机名的布尔值。缺省情况下，连接日志仅显示 IP 地址。

KRB_SERVER_KEYFILE(string)

指定 Kerberos 服务器核心文件的位置。

KSQO(boolean)

键集查询优化（KSQO）导致查询规划器转换 WHERE 子句中包含许多 OR 和 AND 子句的查询。当与 Microsoft Access 协同工作时，如果可能生成此类查询，则通常会用到 KSQO。缺省值是不允许。

LOG_CONNECTIONS(boolean)

是否允许将每次成功连接写入日志的布尔值。缺省值是不允许。

LOG_PID(boolean)

是否允许在日志前加上每个后端进程的进程 ID 信息的称值。缺省值是不允许。

LOG_TIMESTAMP(boolean)

是否允许每个日志信息包括一个时间戳。缺省值是不允许。

MAX_CONNECTIONS(integer)

决定数据库服务器允许的最多并发连接。缺省值是 32。

MAX_EXPR_DEPTH(integer)

设置解析器能允许的最大表达式嵌套深度。缺省值足以满足任何普通查询，但需要，你也可以加大此值。（然而，如果此值太大，可能发生由于堆栈溢出而导致的后端崩溃。）

PORT(integer)

服务器监听的 TCP 端口。缺省值是 5432。

RANDOM_PAGE_COST(floating point)

设置执行随机、非顺序页面抓取的估计开销。

SHARED_BUFFERS(integer)

设置数据库服务器使用的 8KB 共享内存缓冲数。缺省值是 64。

SHOW_QUERY_STATS(boolean),SHOW_PARSER_STATS(boolean),SHOW_PLANNER_STATS(boolean),SHOW_EXCUTOR_STATS(boolean)

设置向服务器日志中写入每种模块的性能统计选项布尔值。

SHOW_SOURCE_PORT(boolean)

是否显示连接用户的端口号的称值。缺省值是不允许。

SILENT_MODE(boolean)

决定 postmaster 是否安静运行的布尔值。如果设置此选项，则 postmaster 自动在后台运行，并且控制丢弃所有 tty，无信息输出到 stdout 或 stderr（使用

postmaster 的-S 选项能达到同样的效果)。除非使用了一些日志系统（如 syslog），我们并不推荐使用该选项，因为它将屏蔽可能的错误信息。

SORT_MEM(integer)

指定在恢复到临时磁盘文件前，内部排序和散列使用的内存数。此值单位是 KB，缺省值为 512KB。

SQL_INHERITANCE(boolean)

决定缺省情况下是否在查询中包含子表的布尔值。缺省时，7.1 及以上版本允许包含；不过，以前的版本并非如此。如果你需要遵循旧的习惯，你可以将此变量设为关闭。

SSL(boolean)

是否允许 SSL 连接的布尔值。缺省值是不允许。

SYSLOG(integer)

决定 postgres 使用 syslog 编写日志的方式值。如果值为 1，则日志信息同时输出到 syslog 和标准输出设备。如果值为 2，则只输出到 syslog。缺省值是 0，即不使用 syslog。要使用 syslog，必须用…enable-syslog 选项配置 postgres。

SYSLOG_FACILITY(string)

此选项决定 syslog 的“设备”（若 syslog 打开）。可选值为 LOCAL0、LOCAL1、LOCAL2、LOCAL3、LOCAL4、LOCAL5、LOCAL6 及 LOCAL7。缺省值是 LOCAL7。

SYSLOG_IDENT(string)

如果 syslog 打开，则此选项决定用于标识 syslog 日志信息中 PostgreSQL 信息的程序名。缺省值为 Postgres。

TCPIP_SOCKET(boolean)

是否允许 TCP/IP 连接的布尔值。缺省值是不允许。

TRACE_NOTIFY(boolean)

是否允许对 LISTEN 和 NOTIFY 命令调试输出的布尔值。缺省值是不允许。

UNIX_SOCKET_DIRECTORY(string)

指定 postmaster 用于客户应用程序连接监听的 UNIX 域名套接字目录。缺省值一般为/tmp。

UNIX_SOCKET_GROUP(string)

设置 UNIX 域名套接字组所有者。

UNIX_SOCKET_PERMISSIONS(integer)

设置 UNIX 域名套接字访问权限。缺省权限是 0777，即表示任意人均可连接。

VIRTUAL_HOST(string)

指定 postmaster 从客户应用程序连接监听的 TCP/IP 主机名或地址。缺省值是监听所有的配置地址（包括本地机）。

WAL_BUFFERS (integer)

WAL 日志在共享内存中所占的磁盘页面缓冲数。

WAL_DEBUG (integer)

若为非零，则打开 WAL 相关调试标准错误输出。

WAL_FILES (integer)

检查点前预先创建日志文件数。

WAL_SYNC_METHOD (string)

强制 WAL 更新到磁盘上的方法。可能值有：FSYNC、FDATASYNC、OPEN_SYNC 和 OPEN_DATASYNC。以上值并非可用于所有平台上。

以上所有布尔值的可接受值为表 8-1 中几种形式。

表 8-1　可接受的布尔值形式

TRUE 值	FALSE 值
ON	OFF
TRUE	FALSE
YES	NO
1	0

文件位置：

RPM—/var/lib/pgsql/data/

源代码—/usr/local/pgsql/data

/etc/logrotate.d/postgres

描述

负责旋转日志文件。

注释/位置

虽然这它不是 PostgreSQL 发行版本中的官方组成部分，但许多系统都包含一个文件来管理和旋转 PostgreSQL 生成的日志文件。

它们通常为 cron 任务，每天或每周按计划运行。这些 RPM 附加配置文件通常用于运行打开 syslog 的 PostgreSQL 日志。

不作为 syslog 配置安装时，不推荐你旋转日志文件。PostgreSQL 时刻打开着与日志文件的连接，所以，当 postmaster 仍然执行时若旋转日志文件可能会导致不可预见的结果。

如果配置 PostgreSQL 打开 syslog 运行不是一个选项，下一个最好的解决办法就是简单地终止 postmaster 服务，旋转日志文件，然后重新启动数据库系统。

有关 PostgreSQL 日志文件和 syslog 的更多信息，请参阅第 9 章“数据库和日志文件”。

文件位置：

RPM—/etc/logrotate.d/postgres

注意：

正因为 syslog 和 PostgreSQL 的日志产生的混淆，此文件在最近发布的 RPM 包中被删除了。不过，与 syslog 可协同工作的旧版本仍然在指定位置包含了此文件。

pg_hba.conf

描述

pg_hha.conf 文件是一个负责基于主机访问控制的配置文件。实质上，它是一个文本文件，它详细描述了用户允许连接到 PostgreSQL 后端的方式。

此文件对于本地或远程（TCP/IP）用户，允许连接数据库及认证方法划分单独的保存区。

根据是否指定了一个 TCP/IP 或本地 UNIX 连接，PostgreSQL 访问控制文件格式会有所不同。基本格式如下：

TCP/IP：

```
Host DB IP Netmask Auth-Type [Auth-Args]
```

Local：

```
Local db Auth-Type [Auth-Args]
```

其中的选项见表 8-2。

表 8-2 **pg_hha.conf 选项**

选　项	描　述
Host	此选项可以为 host、hostssl 或 local，取决于指定的访问方式是否为标准 TCP/IP 连接、安全 TCP/IP 连接或本地 UNIX 连接
DB	访问控制列表有效的数据库名称。用 all 来指定所有数据库或用 sameuser 指定用户仅可以连接到与用户名同名的数据库上
IP	对于 TCP/IP 连接，此选项指定了可与 PostgreSQL 后端连接的有效客户 IP 地址
Netmask	有效客户机的网络掩码
Auth-Type	在授予访问权限前所使用的认证方法。可以为下列方法之一： trust 无需认证：相信此用户 password 与主机提供密码匹配。缺省情况下检查 pg_shadow，除非在 Auth-Args 部分提供了可选值 crypt 与上一个相同，但密码不以明码文本发送；它在传输前被加密 ident 使用 ident 协议（RFC 1413）。通常存在一个名为 pg_ident.conf 的文件，它将 ident 用户名映射成对应的 PostgreSQL 用户名（本地连接不支持） krb4 使用 Kerberos V4（本地连接不支持） krb5 使用 Kerberos V5（本地连接不支持） reject 拒绝连接尝试
Auth-Args	由认证方法指定的不同选项

注释/位置

一个 pg_hha.conf 文件例子如下：

```
local all          trust
host      web          192.168.0.0 255.255.255.0        trust
host      payroll      192.168.0.0 255.255.255.0        crypt
```

在此例中，允许所有的本地连接。同样，允许 IP 地址范围为 192.168.0.0 到 192.168.0.245

的 web 数据库连接。不过，此地址块中的用户若想连接到数据库 payroll，则必须提供由 crypt 方法授予的认证。

文件位置：

RPM——/usr/local/pgsql/data

源代码——/var/lib/pgsql/data/

8.2 库文件

由 PostgreSQL 安装的库文件受很多因素的影响而不尽相同。编译时的选项、特定的包、版本及辅助程序，所有这些因素都决定了安装库的类型。所以，下面列出了被安装的最常见库及它们的典型位置。“→”符号表示一个符号连接，版本号中的“X”或“Y”表示主和次版本号。可以用适当值代表。

源文件：/usr/local/pgsql/lib

RedHat：/usr/lib

ecpg 库文件：

```
libecpg.a
libecpg.so→libecpg.so.X.Y.Z
libecpg.so.X→libecpg.so.X.Y.Z
libecpg.so.X.Y.Z
```

与 libpq 库简化集成的库文件：

```
libpgeasy.a
libpgeasy.so→libpgeasy.so.X.Y
libpgeasy.so.X→libpgeasy.so.X.Y
libpgeasy.so.X.Y
```

标准 C 程序接口库文件：

```
libpq.a
libpq.so→libpq.so.X.Y
libpq.so.X→libpq.so.X.Y
libpq.so.X.Y
```

C++编程接口库文件：

```
libpq++.a
libpq++.so→libpq++.so.X.Y
libpq++.so.X→libpq++.so.X.Y
libpq++.so.X.Y
```

ODBC 接口库文件：

```
libpsqlodbc.a
libpsqlodbc.so→libpsqlodbc.so.X.Y
libpsqlodbc.so.X→libpsqlodbc.so.X.Y
libpsqlodbc.so.X.Y
```

tcl 接口库文件：

```
libpgtcl.a
libpgtcl.so→libpgtcl.so.X.Y
libpgtcl.so.X→libpgtcl.so.X.Y
libpgtcl.so.X.Y
libpgsql.so
```

Perl 接口库文件：

```
/usr/lib/perl5/site_perl/5.005/<arch>/Pg.so
```

Python 接口库文件：

```
/usr/lib/python1.5/site_packages/_pgmodule.so
```

PHP 接口库文件：

```
/usr/lib/php3/pgsql.so
/usr/lib/php4/pgsql.so
```

第 9 章　数据库和日志文件

根据安装方式的不同，日志文件和数据库文件的位置也不相同。一般，它们作为基本的 PostgreSQL 数据文件位于相同的目录中。

对于基于源文件安装的系统，它通常是/usr/local/pgsql/data。对于基于 RPM 安装的系统，它通常为/var/lib/pgsql/data/。不过，以上位置均不是“官方规定”的位置 inux FHS（文件层定义）规定 ar/log/pgsql 或/var/log/postgres 为数据库正确位置。在下面列表中，$PGBASE 指数据库系统安装路径。）

9.1　PostgreSQL 数据目录

每个 PostgreSQL 安装过程都需要指定数据目录。通常这一工作由 initdb 命令在安装中完成。此目录包含许多文件。表 9-1 列出了缺省 PostgreSQL 数据目录中的典型文件。（注意：这些文件可能随版本的不同而不同，但相差不大。

表 9-1　　PostgreSQL 数据目录中的典型文件

文　件	描　　述
$PGBASE/PG_VERSION	文件包含创建此数据目录的 PostgreSQL 的版本号
$PGBASE/base/	包含用户自定义数据库及缺省 template1 数据库的目录
$PGBASE/base/template1	用于所有其他用户创建数据库的缺省模板
$PGBASE/PG_control	PostgreSQL 内部控制文件，用于保持废止事务的检查点、位置跟踪等等
$PGBASE/pg_database	PostgreSQL 内部控制文件，用于保持所有系统上创建数据库的记录
$PGBASE/pg_gego	用于定义基因查询优化（GEQO）的配置文件
$PGBASE/pg_gego.sample	GEQO 样本文件
$PGBASE/pg_group	保存组信息及用户成员资格的内部控制文件
$PGBASE/pg_group_name_index	pg_group 名称的索引文件
$PGBASE/pg_group_sysid_index	pg_group 系统 ID（UID）的索引文件
$PGBASE/pg_log	保持当前事务状态，或提交或不提交
$PGBASE pg_options	当启动 postmaster 时所使用的配置文件（可能列为 postmaster.opts）
$PGBASE/pg_pwd	包含用户名和密码的纯文本文件
$PGBASE/pg_shadow	包含用户名、密码及相关用户权限的系统目录

续表

文 件	描 述
$PGBASE/pg_variable	用于保存当前变量设置（如下一个 OID 等）的内部控制文件
$PGBASE/pg_xlog	放置由 WAL 生成的日志文件的目录。这些文件通过新型预写式日志（版本 7.1 中的功能）确保了数据库的完整性
$PGBASE/postmaster.opts	启动 postmaster 时所使用的配置文件（可能列为 pg_options）
$PGBASE/postgresql.conf	启动 postmaster 时所使用的配置文件（版本 7.1 中的功能）

你将注意到所有的用户定义数据库都保存在$PGBASE/base 目录中。在 PostgreSQL 中创建的每个数据库保存在$PGBASE/base 目录下自己的子目录中。每个目录中的文件包含两个主要类文件：系统目录和用户创建文件。

注意

从版本 7.1 开始文件的位置开始发生了变化。尤其是，现在版本中有一个文件 template0，它是 template1 文件的只读性拷贝。另外，以上提到的很多文件根据它们的 PID 号进行命名，这一变化是为了帮助预读式日志（WAL）的实现。请参考你所使用系统中附带的最新文档了解更多信息。

系统目录

每次创建一个新数据库时，PostgreSQL 都从 template1 数据库中抽取一套基本系统目录。这些文件用于跟踪表、索引、聚集、操作符、数据类型等等。

一套基本系统目录如表 9-2 所示。（不同的版本包含不同的目录文件，以下只是一个代表性的示例）

表 9-2 基本系统目录

文 件	描 述
$PGBASE/pg_aggregate	包含聚集函数的定义
$PGBASE/pg_attrdef	包含表示使用缺省值条件的列缺省值
$PGBASE/pg_attribute	包含每个表中描述每个列属性（即名称、数据类型等）的一个行
$PGBASE/pg_class	包含所有类（表、索引、视图等等）的信息
$PGBASE/pg_database	与整个簇共享，它包含可用数据库的信息
$PGBASE/pg_description	放置用 COMMENT SQL 命令创建的注释
$PGBASE/pg_group	定义组及其成员
$PGBASE/pg_index	包含所有定义索引的信息
$PGBASE/pg_inherits	包含表继承的信息
$PGBASE/pg_language	为可用 PostgreSQL 语言注册调用接口
$PGBASE/pg_operator	数据库中每个操作符类型的一个行
$PGBASE/pg_proc	包含所有定义函数的信息
$PGBASE/pg_relcheck	保存检查约束的信息
$PGBASE/pg_shadow	保存用户信息、密码及有效用户权限
$PGBASE/pg_type	保存数据库中所有可用数据类型的信息

意识到通过标准 SQL 接口（从 DBA 帐号）可以访问这些对象很重要。例如，从 psql，这些系统目录的名称方式就与一般 SQL 表相似。

警告

虽然查看信息很方便，但小心查看时所作的修改。如果作出的错误的改动，你的数据库也许很快瘫痪。

用户定义目录

此目录还包含了用户自定义表、索引及序列等的名称。例如，查找数据库 newriders 的目录，你可以看到：

$PGBASE/authors	authors 表
$PGBASE/auth_idx	表 authors 的索引文件
$PGBASE/payroll	表 payroll
$PGBASE/payroll_idx	payroll 索引文件
$PGBASE/next_check_seq	计算下一个检查号的自创建序列

9.2　日志文件

缺省情况下，postmaster 进程将日志信息发送到 stdout。不过，通常情况下是将输出重定向到一个指定的日志文件。

```
>postmaster -D /usr/local/pgsql/data >pglog 2>&1 &
```

上面命令将把 postmaster 进程的标准输出重定向到 pglog 文件中，并且同时将 postmaster 的 stderr 设备重定向到 stdout（即自身重定向到指定日志文件）。

这种安排方法虽方便但存在一些长期性的问题：

- 如果日志文件不进行维护可能日益庞大。
- 系统日志文件和数据库日志文件应该分区存放。这可导致调试系统产生问题。
- 难以将日志文件重定向到为此目的设计的外部“日志”服务器上。

根据数据库的大小、使用的频率及网络结构，这可能是一个好的解决方案。但对以上提到的问题，可采用以下两种方法来解决：

- 应用自定义日志旋转解决方案。
- 配置 PostgreSQL 来使用 syslog。

自定义日志旋转

如果闲置帐号允许，就可以自定义日志旋转。因为要旋转日志，必须要终止 postmaster 进程。

此类方案通常要求使用 cron 和 shell 脚本。此过程描述如下：

1. postmaster 于指定时间终止（通常在晚上）。
2. 运行日志旋转脚本。
3. 几分钟后，重新启动 postmaster。

根据硬件系统，通常此过程需要 1 或 2 分钟完成。

如何配置 cron 来执行这些任务超出了本书的范围，但此过程一般比较简单。

处理实际日志旋转的脚本可以作为一个简单的 shell 脚本或在 Perl、Python 等语言中完成。一个典型的旋转计划通常要重命名文件（使用 mv）命令，仅保持少量指定的历史。例如：

```
logfile (current)     → logfile.1
logfile.1             → logfile.2
logfile.2             → logfile.3
logfile.3             → logfile.4
logfile.4             → logfile.5
logfile.5             → /dev/null
```

一般，可每天或每周运行这些事件。但对于使用极多的系统，建议您采用一个更加频繁的执行规划。

配置 PostgreSQL 以使用 syslog

对于大部分 PostgreSQL 服务器安装，前面的办法很不错；但仍然还有一些问题没有指出，也就是：

- 日志文件没有与其他系统日志文件集成（这将使调试更加困难）。
- 将日志文件重定向到外部日志系统仍然相当困难。

解决的办法是使用大部分 UNIX（Linux）系统上均存在的 syslog 设备。

一般，需要 3 个步骤来配置 PostgreSQL 使用 syslog：

1. 使用适当的选项来编译源文件来打开日志（即…enable-syslog）或下载支持此功能的 RPM 包。
2. 在 pg_option（或等价命令）文件中打开 syslog 选项（即 syslog=1）。
3. 编辑文件/etc/syslog.conf 来正确捕获 PostgreSQL 系统日志调用，如：

```
local0.*          /var/log/postgresql
```

进行大型安装或远程数据库系统监测时，建议你使用 syslog。

第 10 章　常规管理任务

PostgreSQL 系统的管理任务可分为以下几种：

- 编译和安装。
- 创建用户。
- 分配用户权限。
- 进行常规数据库维护。
- 进行数据库备份和恢复。
- 系统调节。

完成每项任务都需要对系统的相关领域知识有一定的了解。下面将对这些知识进行详细介绍。

10.1　编译和安装

对编译和安装的详细描述（包括所有选项）超出了本书的范围。不过，这里将介绍两种最流行的安装方式：基于源代码安装和包安装。

基于源代码安装

源代码文件可以从 PostgreSQL 的 FTP 站点（ftp. postgresql.org）或全球其他镜像站点获取。

下载后的文件可能为 *tar* 压缩格式。为了能编译，首先要解压缩。将文件移到一个空目录中（例如：/usr/src/postgres）并执行下面命令：

```
>tar xzf postgresql -7.1.tar.gz
```

源代码文件被释放后，如果磁盘空间紧张，可以删除原始文件 tar.gz，否则，将它移一个安全位置。

下一步，阅读安装注释所在目录中的 INSTALL 文本文件。余下的步骤简单介绍如下：

1. 创建一个用户帐号用作 DBA 帐号（postgres 是一个习惯选择）。完成此任务的命令是：userconf、useradd 或系统提供的用于用户管理的任何工具。

2. 查看系统安装选项。下面列出了部分选项（输入./configure -help 可得到完整列表）：

--prefix=BASEDIR（BASEDIR 是所选择的路径。）

--enable-locale

--enable-multibyte（包含对中文等多字节字符的支持。）

--enable-syslog（打开 syslog 功能。）

--enable-assert（允许评估检查；调试功能。）

--enable-debug（打开调试标志进行编译。）

--with-perl（包含对 Perl 接口的支持。）

--with-tcl（包含对 tcl 接口的支持。）

--with-odbc（包含 ODBC 驱动。）

3. 根据所选选项配置源代码（例如 configure -with -odbc）。

4. 输入 make（或 gmake）来生成二进制代码。

5. 如果 make 命令执行失败，请检查生成的日志文件查清编译工作失败的原因（日志文件通常位于./config.log）。

6. 输入 make install 安装二进制代码到指定位置（缺省位置是/usr/local/pgsql）。

7. 通知计算机库所在位置，方法有两种：一是设置 LD_LIBRARY_PATH 环境变量值为<BASEDIR>/lib 路径，二是通过编辑文件/etc/id.so.conf 来包含此库路径。

8. 在用户或系统的搜索路径（即/etc/profile）里包含路径<BASEDIR>/BIN。

9. 创建存放数据库的目录，将所有权改为 DBA，并且初始化此位置（假设存在用户 postgres）

```
# mkdir /usr/local/pgsql/data
# chown postgres /usr/local/pgsql/data
# su - postgres
> /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

10. 后台启动 postmaster 服务器（以 DBA 帐号）。指定前面创建的数据目录，如：

```
>/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data &
```

11. 以 DBA，使用命令 createuser 创建所需用户。

12. 切换到所创建用户并创建所需数据库（即 createdb）。

错误信息输出

第 11 步产生的错误信息将会输出到执行该命令的终端。为了让此错误信息输出到日志文件，要在命令结尾处添上>>server.log 2>>1。请参阅 INSTALL 说明文件了解详细信息。

基于包的安装

实际上，基于包的安装（如 RPM 或 DEM）是一种自动安装过程。不过，阅读前面一节中讲解的基于源代码的安装方式仍然对你大有裨益，因为你至少可以知道安装包对系统所作的变化。

根据计算机上所安装的包管理系统的不同，所使用的命令也有所不同。下面的讲解假设你安装了基于 RPM 包的管理工具，不过，Debian 包管理系统与之在概念上很相似：

1. 下载所需的 RPM 文件列表（下载地址ftp.postgresql.org/pub/binary）。

示例：

```
postgresql-server-7.0.3-2.i386.rpm→Server programs(req)
postgresql-7.0.3-2.i386.rpm→Clients & Utilities(req)
postgresql-devel-7.0.3-2.i386.rpm→Development Libraries
postgresql-odbc-7.0.3-2.i386.rpm→ODBC Libraries
postgresql-perl-7.0.3-2.i386.rpm→Perl interface
postgresql-python-7.0.3-2.i386.rpm→Python interface
postgresql-tcl-7.0.3-2.i386.rpm→TCL interface
postgresql-tk-7.0.3-2.i386.rpm→Tk interface
postgresql-test-7.0.3-2.i386.rpm→Regression Test Routiones
```

2. 安装文件。
3. 检查/etc/passwd（或等价目录）验证创建的PostgreSQL用户。
4. 切换到DBA用户帐号（通常是postgres）并创建所需用户（如createuser web）。切换到此用户并创建工作数据库（如createdb web site）。

10.2 创建用户

数据库用户是与普通操作系统用户分离的实体。根据应用的不同，可能总共只有一个或二个数据库用户。不过，如果多个用户需要连接到数据库——每个用户都有自己的所属访问权限——这就需要创建单独用户帐号了。

创建用户的最简单办法就是运用行命令createuser。

创建新用户时，需要考虑3个主要方面：

- 他们能创建自己的用户吗？（他们是超级用户吗？）
- 他们能创建自己的数据库吗？
- 是否需要认证？如果需要，认证的类型与密码是什么？

创建用户的实际动作可以通过命令行或在一个交互SQL对话中进行。

从命令行创建用户：

```
>createuser web
>Shall the new user be allowed to create databases(y/n)?N
>Shall the new user be allowed to create users(y/n)?N
```

或者，从SQL对话创建：

```
psql=>CREATE USER web NOCREATEDB NOCREATEUSER;
CREATE
```

从SQL对话创建用户可以使用另外一些从命令行创建不支持的选项。例如，密码、组成员和帐号过期均可通过此方法来加以设置。

除此之外，为了更容易进行权限管理，PostgreSQL允许将用户列入逻辑组中。要创建一个组，在SQL对话中必须输入以下命令：

```
CREATE GROUP webusers;
```

然后就可以从组中添加或删除用户了，如下所示：

```
ALTER GROUP webusers ADD USER bill,mary,amy,jane;
```

```
ALTER GROUP webusers DROP mary;
```

10.3 授予用户权限

在PostgreSQL数据库系统中有4种基本权限：

- 选择（读）。
- 插入（写）。
- 更改/删除（写）。
- 规则（写/执行）。

缺省时，数据库创建者隐式拥有对数据库所有对象的所有权限。对于DBA超级用户账户，这些权限是永恒的。

要分配其他用户数据库权限，可使用GRANT和REVOLE两个SQL命令，如：

```
GRANT SELECT,UPDATE ON authors TO bill;
REVOKE ALL ON payroll FROM joe;
```

PostgreSQL系统还有一个保留关键字PUBLIC可用于系统中的每一个用户（DBA除外）。它使设置全局规则和权限更加容易。

```
GRANT SELECT,UPDATE ON authors TO PUBLIC;
REVOKE UPDATE,DELETE on payroll FROM PUBLIC;
```

对大量用户帐户逐个定义权限是一个繁琐的工作。结合组使用GRANT和REVOKE命令是处理权限管理工作的一个有效方法。

一般情况下，用户应该划分成逻辑组，将相似权限的用户组合在一起。整个组的权限就可以通过GRANT和REVOKE命令加以分配和回收，而不必指定单个用户。

```
GRANT SELECT,UPDATE ON authors TO GROUP staff;
REVOKE UPDATE,DELETE on payroll FROM GROUP staff;
GRANT SELECT, UPDATE,DELETE on payroll TO GROUP managers;
```

10.4 数据库维护

适当的数据库维护可确保系统一贯以最优状态运作，而且可以有效外理出现的问题。数据库维护有3个主要方面：

- 监测日志文件。
- 规划常规VACUUM和VACUUM ANALYZE事件。
- 常规备份。

对日志文件的经常监测可告诫管理员潜在的问题，这些问题可以Pt它们成为大问题前得以更正。一些管理员甚至自己编写一些脚本来分析日志文件并将一些可疑的现象发送到某个邮件地址，以让管理员采取进一步的措施。

cron也是一个有用的数据库维护工具。尤其适合于执行vacuumdb和日志旋转一类常规任务。vacuumdb能作为一个自动cron任务的主要原因是vacuumdb可以作为一个单

独的行命令执行。

10.5 数据库备份/恢复

数据库维护计划的最重要部分是数据库的备份和恢复过程。同样，PostgreSQL 通过提供 pg_dump、pg_dumpall 和 pg_restore 等命令行工具使这一管理工作更加容易。与 vacuumdb 行命令一样，以上命令适合于作为 cron 任务执行。

缺省情况下，pg_dump 和 pg_dumpall 简单地将输出倾倒到 stdout。不过，使用适当的 UNIX 重定向符号可以将之重定向到某个文件。

```
>pg_dump newriders > nr.backup
```

运用此命令将输出重定向到标准 OS 文件后，可用标准备份工具安全保存它。

当评价一个最佳备份方案时，应当考虑以下几个因素：

■ 是整个系统还是仅有一个数据库需要备份？

如果只是一个数据库需要备份，pg_dump 命令足以应付。如果整个数据库簇需要备份，则需要使用 pg_dumpall 命令。

这两个命令功能几乎一样，但 pg_dump 不能对它所倾倒的所有数据库进行认证。在需要认证的后端运行 pg_dumpall 命令，将环境变量 PGPASSWORD 设置成正确的密码，就可以自动继续每个数据库连接了。

■ 你需要选择性地恢复数据库文件吗（也就是指定表等）？

PostgreSQL 的 7.1 版本对 pg_dump、pg_dumpall 和 pg_restore 命令作了一些改进。这些命令允许数据库倾倒以特定格式保存。

这一新格式为恢复数据提供了极大的灵活性。数据库规划、数据函数或指定表都能选择性地恢复。另外，新格式以压缩格式保存数据，所以处理非常大的数据库时就会遇到较少的问题。

例如，以特定格式倾倒数据库 newriders，然后选择性地恢复表 payroll：

```
>pg_dump -Fc newriders > nr.backup.cust_fmt
>pg_restore -d newriders -t payroll nr.backup.cust_fmt
```

■ 倾倒所得的最后文件有多大？

许多操作系统（如某些版本的 Linux）对单个文件的大小有限制（如 2GB）。所以，对于大数据库系统可能会遇到难题。

解决这一难题有许多方法，从升级你的 PostgreSQL 数据库系统到通过特殊工具管道输出。

前面已提及，PostgreSQL 7.1 版本为 pg_dump 和 pg_dumpall 命令引入了一些新功能，包括新自定义倾倒格式的使用。这些附加功能包括一个新命令行选项，它用来表示所期望的压缩比。

例如，用最大的压缩比来倾倒数据库 newriders（以降低速度为代价）：

```
>pg_dump能 -Fc -z9 newriders > nr.backup.cust_fmt
```

同时，另外一个能达到同样效果的方法是通过 gzip 命令管道输出 pg_dump 结果。任何一个版本的 PostgreSQL 和标准 UNIX 系统命令均支持这一方法。

```
>pg_dump newriders | gzip >nr.backup.zip
```

还可通过下面命令来恢复：

```
>gzip -c nr.backup.zip | psql newriders
```

如果经压缩的结果文件仍然太大，另一个选项就是使用 split 命令。下面例子将输出文件分割成多个 1GB 大小的文件：

```
>pg_dump newriders | split -b 1024m -nr.backup
```

而且它可以通过下面命令来恢复：

```
>cat nr.backup.* | psql newriders
```

- 配置文件后，pg_options、pg_hha.conf 和 pg_pwd 等文件是否进行了常规保存？

对于复杂的安装，丢失配置文件后，手工重新创建它们将是十分费时的工作。请确保你已经安全、异地备份了所有 PostgreSQL 配置文件。

10.6 操作性能优化

一般而言，获得数据库系统的最佳操作性能没有百分之百成功的办法。不过，下面一些原则有助于管理员成功实现操作性能调节策略。

硬件考虑

如果你注意到你的数据库系统一直 CPU 高负荷工作或产生过大量硬盘页面，那你的硬件需要升级了。

与数据库操作性能有关的最大硬件问题有：

- **RAM** 不足的 RAM 会导致数据库经常不得不在硬盘上交换内存内容。这一耗时的操作可能会导致操作性能的直线下降。
- **硬盘** 慢速硬盘和控制器可能导致操作性能不佳。升级到新的控制器和（或）驱动会带来系统速度的显著提高。特别是，使用 RAID 磁盘阵列可以为系统性能提高带来相当大的好处。
- **CPU** 不足的 CPU 资源可能会降低系统反应速度，特别是同时处理大量查询时，这一影响更为突出。因为 PostgreSQL 为非多线程工作方式，所以多 CPU 系统并不能给它带来直接的好处。但每个连接只接收自身进程，从这一点讲，它可以从多 CPU 系统受益。
- **网络** 不管系统的硬件多么强劲，如果存在网络问题，操作性能同样会大打折扣。升级网卡、添加局域网开头及增加带宽容量无疑会提升系统操作性能。

优化 SQL 代码

虽然通常将数据库的效率低下归咎于硬件的薄弱，但往往只要对数据库代码施加优化就可提高其操作性能。

优化数据库代码的一般性原则有：

- 对常用查询字段编制索引。

对于那些存在联结或多个 SELECT…WHERE 子句经常访问的字段应该编制索引。不过，应该把握好字段的索引数量与操作性能间的平衡点。索引有利于选择但不利于插入或修改。所以，对每个字段均进行索引并非上策。

■ 使用显式事务。

如插入或修改多个表，将语句封装在 BEGIN…COMMIT 子句内能显著提高操作性能。

■ 使用游标。

使用游标能大幅提高系统操作性能。特别是，使用游标来创建用户选择列表比运行多个单独的查询更有效。

■ 限制触发器和规则的使用。

虽然触发器和规则是数据完整性的重要部分，但滥用它们将会严重影响系统操作性能。

■ 使用显式联结。

从 PostgreSQL 7.1 版本开始，通过使用显式 JOINT 语法可以控制查询规划器的操作方式。例如，下面两个查询结果相同，但第二个明确地指定了查询规划器处理顺序：

```
SELECT * FROM x,y,z WHERE x.name=y.name AND y.age=z.age;
SELECT * FROM x JOIN (y JOIN z ON (y.age=z.age)) ON (x.name=y.name);
```

■ 在前端强制使用逻辑机制。

在数据库应用前端强制使用一些最低限度的标准能整体提高系统操作性能。检查输入字段的有效性和（或）最少信息需求可以避免执行极其费时的查询。例如，强制要求前端输入的姓多于三个字母将可防止后端处理查询时返回所有姓以一个“S”开始的记录，而这可能是一个非常耗时的查询过程，而且对用户无任何真正的价值。

缓存大小和其他因素

PostgreSQL 对缓存大小、并发连接及排序内存提供了缺省值或预设值。通常这些设置对单一数据库能满足要求。不过，这些设置力求在系统空闲时对系统的影响尽量小。

对于大型专业服务器，它们所拥有的数据达几百或几千 MB，这时，就需要调整以上预设值了。

通常假设选项的设置值越高，系统操作性能自动提升越大。一般而言，通过调整这些设置不可能得到超过 20%的性能提升。保留足够的 RAM 给核心任务很重要；处理网络连接、管理虚拟内存、控制进度及系统管理尤其需要有充足的内存数量。没有适当的容许量，系统的操作性能和反应速度就会受到负面影响。

影响数据库操作性能的 3 个至关重要的运行设置项为：共享缓存、排序内存及并发连接数。

1. 共享缓存

共享内存选项（-B）决定所有服务器进程可用的 RAM 数量。最小值应该至少设为所允许并发连接数量的两倍。

共享内存可以在 postgresql.conf 文件中设置或直接在 postmaster 后端运行行命令选项来设置。缺省情况下，许多 PostgreSQL 安装时预设此值为 64。每个缓存消耗 8KB 的系统 RAM。所以，根据缺省设置，512KB 的 RAM 用于共享缓存。

如果设置对象是一个可能处理非常大数据库或多个并发连接的专业数据库，可能需要将之设置为高达系统 RAM 的 15%。例如，在一个拥有 512MB RAM 的计算机上，共享缓存应

设置为 9,000。

理想情况下，缓存空间应足以完全将大部分经常访问表装载于内存中。然而，它应该足够小以避免与内核的（页面）交换。

2. 排序内存

通过 postgre 后端（通常只称作为 postmaster 进程）选项（-S）来决定查询排序可用内存数。此值决定了在处理排序进程或散列类函数时，在重排序到磁盘空间前所消耗的物理 RAM。

此设置值以 KB 为单位，标准缺省值为 512KB。

对于复杂的查询，许多排序和散列可能平行运行，而且在交换到硬盘之前，它们允许使用同样数量的内存。这一点需要强调：如果你盲目将此值设为 4,096，每个复杂查询和排序将允许占用 4M 之多的内存。根据计算机可用资源的多寡，这种情况可能导致内核虚拟子系统溢出。不幸的是，这通常比只允许 PostgreSQL 首先创建临时文件要慢得多。

3. 并发连接

postmaster 选项（-N）用于设置 PostgreSQL 允许的并发连接数量。缺省情况下，此值设为 32。不过，它可以设置为高达 1,024 个连接。（记住，共享缓存需要设置为此值的两倍）还请记住，PostgreSQL 并不支持多线程方式；所以，每个连接都将启动一个新进程。在一些系统上，如 UNIX，它不会造成大的问题。但在 NT 上，它往往可以引发问题的发生。

通过 EXPLAIN 优化查询

EXPLAIN 命令用来评估给定查询的查询规划。它返回以下信息：

- **启动开销** 这是开始输出扫描前时间开销预估值。一般情况下，如果在开始前需要等待另一个查询的完成的话，此值为非 0。子选择和联结就属于这种情况。
- **总开销** 返回所有行所花时间的预估值。计算时不论是否有其他因素（如一个 LIMIT 子句）阻止所有行的返回。
- **输出行** 这是返回行数的预估值。与前面一样，即使像 LIMIT 子句等因素阻止了行的返回，预估仍然有效。
- **预估平均宽度** 它指平均行宽，以字节为单位。

上面所提的时间量与客观时间无关；它们表示完成查询所需的磁盘页面读取时间。

例如：

```
EXPLAIN SELECT * FROM authors;

NOTICE: QUERY PLAN
Seq Scan on authors (cost=0.00..92.10 rows=5510 width=20)
```

上面代码中的 EXPLAIN 语句将列出以下预估计值：

- 使用序列扫描（与索引对应）。
- 启动时间无延迟。
- 提交整个查询开销为 92.10。
- 返回行预估值 5,510。
- 平均行宽 20 字节。

■ 修改查询将产生不同的结果：

```
EXPLAIN SELECT * FROM authors WHERE age<10000;

NOTICE: QUERY PLAN
Seq Scan on authors (cost=0.00..102.50 rows=5510 width=20)
```

在这一个例子中，你可以看到总开销略有提高。有趣的是，尽管表中对 age 有一个索引，但查询规划器仍然使用了顺序扫描。这是因为搜索标准太宽松；使用索引扫描没有任何好处。（显然，age 列中所有值都小于 10,000）

如果你对搜索标准稍加限制，你就可以看到发生的一些变化：

```
EXPLAIN SELECT * FROM authors WHERE age<75;

NOTICE: QUERY PLAN
Seq Scan on authors (cost=0.00..102.50 rows=5332 width=20)
```

下面你再次使用索引扫描，虽然返回的行现在较少。进一步的约束可使结果变化更大：

```
EXPLAIN SELECT * FROM authors WHERE age<30;

NOTICE: QUERY PLAN
Index Scan using age_idx on authors (cost=0.00..32.20 rows=991 width=20)
```

此结果的多个方面很有意思。首先，足够多的约束条件迫使查询规划器使用了 age_idx 索引。其次，总开销和返回的行数大大减少了。

最后，让我们试试下面的例子：

```
EXPLAIN SELECT * FROM authors WHERE age<27;

NOTICE: QUERY PLAN
Index Scan using age_idx on authors (cost=0.00..3.80 rows=71 width=20)
```

你可以看到，使用这样一个限制性的标准后，查询速度获得了极大提高。

在复杂查询中使用 EXPLAIN 命令可以揭示数据库结构中的一些潜在问题。

```
EXPLAIN SELECT * FROM authors, payroll WHERE
      authors.name=payroll.name;

NOTICE: QUERY PLAN

Merge Join (cost=69.83..425.08 rows=85134 width=36)
      ->Index Scan using name_idx in authors
          (cost=0.00..273.80 rows=5510 width=20)
      ->Sort (cost=69.83..69.83 rows=1000 width=16)
          ->Seq scan on payroll
              (cost=0.00..20.00 rows=1000, width=16)
```

此输出说明了一些有关数据库结构的有趣现象。显然，表 authors 有一个对 name 的索引，但 payroll 表好像重新排序以使用顺序扫描以及排序与字段匹配。

经过调查，事实上，payroll 表并没有针对此联结的索引。所以，创建一个索引后，你可以得到如下结果：

```
CREATE INDEX pr_name_idx ON payroll(name);

EXPLAIN SELECT * FROM authors, payroll WHERE
      authors.name=payroll.name;

NOTICE: QUERY PLAN

Merge Join (cost=0.00..350.08 rows=44134 width=36)
      ->Index Scan using name_idx in authors
          (cost=0.00..273.86 rows=5510 width=20)
      ->Index Scan using pr_name_idx in payroll
          (cost=0.00..29.50 rows=500, width=16)
```

通过在 payroll 表中加入索引，查询速度得到了 25%的提升。

在查询中使用 EXPLAIN 是一条发现影响系统操作性能潜在瓶颈的好途径。

事实上，对于非硬件引发问题，EXPLAIN 命令是 DBA 中唯一最好的用来解决操作性能问题的工具。EXPLAIN 命令提供了智能分配系统资源（如共享缓存）、优化查询和索引以提高操作性能的所需信息。

使用 EXPLAIN 命令的最好方式之一是利用基准生成工具。按此方法，当表结构、索引、硬件或操作系统改变时，此工具就会对变化前后作一个有效的比较，显示变化对系统操作性能的影响程度。

第四部分

用 PostgreSQL 编程

第 11 章　服务器端编程

通常，在 PostgreSQL 里有两种编程方法：使用内部声明的一种过程语言（PL）或使用外部安装的应用编程接口（API）。两者间最基本的区别就是过程语言面向服务器端，而 API 面向客户端访问。

服务器编程的代码实际上在 PostgreSQL 后端系统编写、运用和执行。一般情况下，这些代码的意图是扩展基本系统的功能并让其他查询或 SQL 语句能使用这些自定义功能。

客户端编程是为了让 PostgreSQL 后端外部的应用能插入、操纵或读取 PostgreSQL 数据库引擎内的数据。

语言和方法的选择与若干因素密切相关。过程语言编程相对简单所以缩短开发周期。过程语言编程是用于一般性扩展基本 PostgreSQL 系统的首选方法，而且它可以最大限度地重复利用代码。

当要求精确控制和（或）执行效率时，使用外部 API 是合适的。而且，在特定情况下，使用外部 API 也许是实现前端应用与后端系统交互操作的唯一方法。

11.1　过程语言的优势

本章着重讲解 PostgreSQL 中过程语言的使用。不管选择哪种 PL，对于开发者都有以下几个共同优势：

- **扩展性**使用内部 PL 可以让开发者迅速创建自定义函数、触发器及规则来添加基本系统中不存在的功能。而且，一旦完成了功能的扩展，系统中的其他 SQL 语句也能利用此功能。
- **控制结构**缺省情况下，SQL 语言不允许程序员使用其他常用编程语言中包含的丰富控制结构集及条件判断。由于此原因，内部 PL 允许开发者将传统控制结构与 SQL 语言结合。这一点对于创建复杂的计算和触发器特别有用。
- **高效性和兼容性**使用内部 PostgreSQL PL，开发者可以访问所有基本系统中已经存在的内部数据类型、操作符和函数。这可以大大提高开发的效率，因为程序员不需要在自己的代码中重新创建已经由 PostgreSQL 定义的常规元素。另外，还可以确保返回的数据类型和比较结果与 PostgreSQL 后端保持高度兼容。
- **安全性**后端系统信任内部 PostgreSQL PL，而且 PL 仅访问有限的系统函数集。特别是，PL 在系统级别（与 postgres 用户具有相同的权限）进行操作。这是因为 PostgreSQL 认为外来文件系统对象是安全的。

11.2 安装过程语言

按缺省安装方式，PostgreSQL 将自动包含访问 PL/pgSQL 语言代码的功能。通过设置单独的编译变量（即-with -tcl 或 - with -perl）还可以包含 PL/Tcl 和 PL/Perl 语言。

不过，系统安装后，为了能运用指定的 PL，还必须完成几个定义步骤。完成方法有两个：通过 SQL 语句显式声明或使用 createlang 命令。createlang 命令可以自动完成许多手工创建新语言所需的步骤，且这是一种首选方法。两个方法在下面讲解中都会涉及到。

SQL 声明

作为一个句柄的共享库的位置可能随安装方式的不同而不同。基于 RPM 包的安装通常将它放在/usr/lib/pgsql 目录中。源代码安装根据指定的安装脚本路径也有所不同。在这种情况下，UNIX 的 find 命令可以发挥作用。

通过显式 SQL 语句在 PostgreSQL 中添加一个新语言的步骤如下：

1. 通过指向包含对象源代码的文档编译共享对象（如 plpgsql.so）。编译完成后，将此对象复制到适当的库目录中。

2. 使用 CREATE FUNCTION 子句在 PostgreSQL 中声明句柄。（请参阅第 1 章“PostgreSQL SQL 参考”中的“CREATE FUNCTION”一节了解此命令的语法）

如：

```
CREATE FUNCTION plpgsql_call_handler()
      RETURN OPAQUE AS
      '/usr/local/pgsql/lib/plpsql.so' LANGUAGE 'c';
```

3. 使用 CREATE LANGUAGE 子句创建此信任语言（请参阅第 1 章“PostgreSQL SQL 参考”中的“CREATE LANGUAGE”一节了解此命令的语法）。

如：

```
CREATE TRUSTED PROCEDURAL LANGUAGE 'plpsql'
      HANDLER plpsql_call_handler
      LANCOMPILER 'PL/pgSQL';
```

使用 createlang 命令

还有一种定义 PostgreSQL 基本系统中语言（如 PL/Tcl）的方法，就是使用 createlang。这一方法可简化许多步骤。（参阅第 6 章“用户可执行文件”中的“createlang”命令部分了解此命令的选项）

目前，createlang 可用于在后端服务器注册 PL/Tcl 或 PL/Perl 语言。而且，createlang 命令中还可以指定用于声明语言注册数据库对象的选项。如果语言注册于 template1 数据库，则此语言可用于将来创建的所有数据库中。

如：

```
>createlang pltcl tempate1
```

此命令将自动在数据库 template1 及后续所有数据库中注册 PL/Tcl 语言。

11.3 PL/pgSQL

PL/pgSQL 语言是一般用于服务器端编程的缺省语言。它集成了 SQL 的简单和脚本语言的强大。

通过 PL/pgSQL 语言，你可以创建自定义函数、操作符及触发器。一个标准应用就是在数据库内部组合查询。许多 RDBMS 称之为“存储过程”，客户应用通过它可以快速请求指定的数据库服务，而不必花较长时间来启动事务。在客户端和服务器端建立对话通常会明显降低系统速度。

当创建一个 PL/pgSQL 函数时，它在内部编译成 8 位元组代码。每次调用此函数时就会调用生成的与二进制相近的代码。PostgreSQL 执行经 PL/pgSQL 编译过的代码而不必重新解释 SQL 命令。所以，与单独运行相同的 SQL 命令相比，调用此函数可大大提高操作性能。

使用 PL/pgSQL 的另一个好处是兼容性问题。因为 PL/pgSQL 可以在整个 PostgreSQL 系统内执行，这就意味着 PL/pgSQL 代码可以运行于任一个安装了 PostgreSQL 的系统上。

PL/pgSQL 语言定义

PL/pgSQL 代码基本结构如下：

```
<label declaration>
[DECLARE
      …Statements … ]
BEGIN
      …Statements …
END;
```

相互间可以封装任意数量的块，如：

```
<label declaration>
[DECLARE
      …Statements … ]
BEGIN
      [DECLARE
          …Statements … ]
      BEGIN
          …Statements …
      END;
          …Statements …
END;
```

当 PostgreSQL 遇到多组 RECLARE…BEGIN…END 语句时，它将所有变量解释成针对各个组的局部变量。实际上，子组中的变量不能被邻组或父组中的变量所使用。例如，在下面例子中，所有的 myvar 都是各自子组中的局部变量：

```
CREATE FUNCTION myfunc()RETURNS INTEGER AS'
DECLARE
      myvar INTEGER :=1;
```

```
BEGIN
        RAISE NOTICE "My Variable is %",myvar;
    DECLARE
        myvar VARCHAR :="Hello World ";
    BEGIN
        RAISE NOTICE "My Variable is %",myvar;
    END;
END;' LANGUAGE 'plpgsql ';
```

此例中，两个 myvar 变量不仅包含了不同的数据，而且它们还用来装载不同类型的数据。

1. 注释语句

PL/pgSQL 有两种不同类型的注释：一个是内嵌注释（如--），一个是注释块（如/*…*/）。例如：

```
BEGIN
    Some-code                --this is a comment
    <…>
    <…>
    <…>
    Some-more-code
    <…>
    /*And this
    is a comment
    block */
END;
```

2. 变量赋值与声明

在 PL/pgSQL 语句的 DECLARE 块中进行变量的声明。任何有效的 SQL 数据类型都可以赋值给 PL/pgSQL 变量。声明语句语法为：

```
name [ CONSTANT ]type [ NOT NULL ] [ {{DEFAULT | :-}value }
```

name ——定义变量名。

CONSTANT—表示变量为只读的关键字。

Type—SQL 数据类型（如 INTEGER,INTERVAL,VARCHAR 等）。

NOT NULL—缺省情况下，所有的变量都被初始为 NULL，除非赋予了某个值。包含此关键字后，则禁止将变量值设为 NULL，且要求赋予一个缺省值或其他值。

DEFAULT—表示设置为缺省值的关键字。

value—缺省值或显式声明。

若变量类型未知，程序员可以使用%TYPE 和%ROWTYPE 命令，它将在一个数据库表中自动搜集指定变量类型或整个行。

例如，想让变量 myvar 类型自动与表/字段 payroll.salary 一致，可以使用下面代码：

```
CREATE FUNCTION yearlysalary(INTEGER,INTEGER)RETURN INTEGER AS '
    DECLARE
```

```
        myvar payroll.salary%TYPE;
    BEGIN
        RETURN myvar*2;
    END;
    ' LANGUAGE 'plpgsql ';
```

另外，还可以使用%ROWTYPE 语法指定整个数据库行的类型。例如：

```
CREATE FUNCTION yearlysalary(INTEGER,INTEGER)RETURN INTEGER AS '
    DECLARE
        myvar payroll%TYPE;
    BEGIN
        RETURN myvar.salary*2;
    END;
    ' LANGUAGE 'plpgsql ';
```

3. 传递变量给函数

PL/pgSQL 可以容纳多达 16 个传递变量。根据各自的序号指向变量。编号序列从 1 开始；所以$1 表示传递第 1 个变量，$2 表示传递第 2 个变量，依次类推。

无需申请所传递变量的数据类型；PL/pgSQL 将自动根据适当的变量值作为正确的数据类型。

不过，使用关键字 ALIAS，可以让程序员为某一序号指定更多的描述性变量名。例如：

```
CREATE FUNCTION addnumbers(INTEGER,INTEGER)RETURN INTEGER AS '
    DECLARE
        Number_1 ALIAS FOR $1;
        Number_2 ALIAS FOR $2;
    BEGIN
        RETURN Number_1 +Number_2;
    END;
    ' LANGUAGE 'plpgsql ';
```

另外，RENAME 命令可用于重命名当前变量为其他的名称。例如：

```
CREATE FUNCTION addnumbers(INTEGER,INTEGER)RETURN INTEGER AS '
    DECLARE
        Number_1 ALIAS FOR $1;
        Number_2 ALIAS FOR $2;
    BEGIN
        RENAME Number_1 TO Orig_Number
        RETURN Orig_Number +Number_2;
    END;
    ' LANGUAGE 'plpgsql ';
```

4. 控制语句

PL/pgSQL 支持大部分常规控制结构，如 IF…THEN、WHILE 循环及 FOR 语句。这些语句的大部分语法与其他语言一样。下面介绍这些控制语句的基本格式。

（1）IF…THEN…ELSE…ELSE IF

除基本的 IF…THEN 语句外，PL/pgSQL 还允许执行 ELSE 和 ELSE IF 测试。一个 ELSE IF

条件测试字符串与一个 CASE 或 SWITCH 语句相当，这两个语句经常用于其他编程语言中。

```
IF conditional-expression THEN
      execute-statement;
END IF;

IF conditional-expression THEN
      execute-statement;
ELSE
      execute-statement;
END IF;

IF conditional-expression THEN
      execute-statement;
ELSE IF conditional-expression2 THEN
      execute-statement;
END IF;
```

（2）LOOPS

与其他编程语言一样，PL/pgSQL 允许创建当满足一定条件时就执行的循环。循环对于遍历一系列表中行和完成一些操纵特别有用。

使用下面模板可创建一个无限循环：

```
LOOP
      Statements;
END LOOP;
```

如：

```
LOOP
      x:=x+1;
END LOOP;
```

或者，EXIT 转向语句可用在 IF…THEN 语句中创建一个退出点。

```
LOOP
      x:=x+1;
      IF x>10 THEN
          EXIT;
      END IF;
END LOOP;
```

执行以上任务的另一个方法是使用 EXIT WHEN 语句，如：

```
LOOP
      x:=x+1;
      EXIT WHEN x>10;
END LOOP;
```

可使用 WHILE 子句简洁地实现以上功能：

```
WHILE x<10 LOOP
      x:=x+1;
END LOOP;
```

与 WHILE 类型循环比较而言，FOR 循环用于执行一个固定次数的循环。FOR 语句的语法如下：

```
FOR name IN [REVERSE] expression_start..expression_end LOOP
```

例如，下面两个例子分别从 1 计数到 100 及从 100 倒计数到 1：

```
FOR a IN 1..100 LOOP
     RAISE NOTICE '%',a;
END LOOP;

FOR a IN REVERSE 1..100 LOOP
     RAISE NOTICE '%',a;
END LOOP;
```

虽然这些例子与 WHILE 循环功能相似，但 FOR 循环的真正威力在于遍历记录集。例如，下面例子中通过 FOR 遍历 payroll 表并计算指定时期的薪水发放总数。

```
CREATE FUNCTION totalpay(DATETIME)REAL AS '
     DECLARE
             recs RECORD;
             payroll_period ALIAS $1;
             retval REAL :=0;
     BEGIN
             FOR recs IN SELECT *FROM PAYROLL WHERE
                 payperiod=payroll_period LOOP
                 retval:=retval+PAYROLL.SALARY;
             END LOOP;
             RETURN retval
     END;
     ' LANGUAGE 'plpgpsql ';
```

（3）使用 SELECT

PL/pgSQL 中的 SELECT 语句在代码块中的运行方式与标准 **SQL** 中的 SELECT 语句有一些细微差别。SELECT…INTO 命令通常创建一个新表；然而，在一个 **PL/pgSQL** 代码块中，它将选择行分配给一个变量占位符。例如，下面例子声明了变量 *myrecs* 作为一个 RECORD 并用 SELECT 查询结果填充它：

```
CREATE FUNCTION checkemail()RETURNS INTEGER AS'
     DECLARE
         myrecs RECORD;
     BEGIN;
         SELECT INTO myrecs *FROM authors WHERE
             name='Barry ';
         IF myrecs.email IS NULL THEN
             RETURN 0;
         ELSE
             RETURN 1;
         END IF;
     END;
```

```
' LANGUAGE 'plpgsql ';
```

在上面例子中，根据与一个 SQL NULL 值的比较来判断一个 email 的存在。NOT FOUND 子句也能用于 SELECT INTO 查询中。例如：

```
CREATE FUNCTION checkemail()RETURNS INTEGER AS '
    DECLARE
        myrecs RECORD;
    BEGIN;
        SELECT INTO myrecs *FROM authors WHERE
            name='Barry ';
        IF NOT FOUND THEN
            RETURN 0;
        ELSE
            RETURN 1;
        END IF;
  END;
  ' LANGUAGE 'plpgsql ';
```

5. 执行函数内代码

有两个执行当前代码块内代码的基本方法。如果不要求返回值，开发者可以通过命令 PERFORM 来调用代码。

如果要求动态查询，就可使用 EXCUTE 命令。

下面的例子中就显示了 PERFORM 命令的使用方法。首先，定义一个自定义函数 addemp，它接受创建 employee 所需的参数。但如果 employee 已经存在，函数就退出且返回退出代码值 0。不过，创建 employee 后，退出代码值为 1。下面是第一个函数的示例：

```
CREATE FUNCTION addemp(VARCHAR,INTEGER,INTEGER)
RETURNS INTEGER AS '
    DECLARE
        Name ALIAS FOR $1;
        EmpID ALIAS FOR $2;
        Age ALIAS FOR $3;
        EmpRec RECORD;
    BEGIN
        /*Check to see if emp exist */
        SELECT INTO EmpRec *FROM employee WHERE
            Employee.emp_id=EmpID;
        IF NOT FOUND THEN
            /*Doesn 't exist,so add them
            INSERT INTO employee VALUES (Name,EmpID,Age);
            RETURN 1;
        ELSE
            /*Emp already exist,exit status 0 */
            RETURN 0;
        END IF;
```

```
END;
'LANGUAGE 'plpgsql ';
```

创建以上函数后，你就可以从其他函数使用 PERFORM 语句调用此函数。如前所述，PERFORM 语句忽略从调用函数返回的任何值。所以，在上例中，返回值 0 或 1 将被忽略。不过，根据我们使用 addemp 函数的本来目的，这一点无关紧要。

```
<function is created>
      <…其他代码 …>
      …
      /*Traverse List and run against addemp function */
      FOR emp IN SELECT *FROM TempEmps;
      PERFORM addemp(emps.name,emps.emp_id,emps.age);
      END LOOP;
      …
      <…其他代码 …>
<End Function>
```

在上面例子中，从 PERFORM addemp 子句中无任何返回值。在这里，这正是所期望的，因为 addemp 函数只在它适合于添加 employee 时才执行添加动作。

EXCUTE 语句与 PERFORM 命令相比，它不是执行预定义函数，EXCUTE 语句用于处理动态查询。

例如，下面的代码片断简单演示了它的使用方法：

```
CREATE FUNCTION orderemp(VARCHAR)RETURNS OPAQUE AS '
      DECLARE
          SortOrder ALIAS FOR $1;
          QueryStr VARCHAR;
      BEGIN
           /*Determine Sorting Order */
           IF SortOrder :="Age " THEN
                  QueryStr :="age ";
           ELSE IF SortOrder :="ID " THEN
                  QueryStr :="emp_id ";
           ELSE IF SortOrder :="FName " THEN
                  QueryStr :="first_name ";
           ELSE IF SortOrder :="LName " THEN
                  QueryStr :="last_name ";
           ELSE
RAISE NOTICE "Unknown value:"||SortOrder;
RETURN 0;
          END IF;
EXECUTE "SELECT *FROM employee ORDER BY "||
QueryStr;
         RETURN 1;
      END IF;
' LANGUAGE'plpgsql ';
```

上面例子显示的是如何使用 EXCUTE 语句创建一个基本动态查询。不过，还可进行更复杂的实际应用。事实上，完全可以在另一个函数中使用 EXCUTE 语句创建自定义函数。

6. 其他与通知

PL/pgSQL 使用 RAISE 语句向 PostgreSQL 日志系统中插入信息。RAISE 语句的基本格式如下：

```
RAISE level 'format' [,identifier [...]];
```

level——DEBUG、NOTICE 或 EXCEPTION。

format——使用%字符表示 *identifier* 以逗号分隔的列表中的占位符。

identifier——写入日志的信息列表（文本字符串或变量）。

如果关闭调试（编译选项）则 DEBUG 被忽略。NOTICE 将信息写入客户端应用并输入到 PostgreSQL 系统日志文件中。EXCEPTION 将执行 NOTICE 以外的所有动作而且还强制从父事务回滚。

下面是一些简单的例子：

```
RAISE NOTICE "Warning!Salary change attempted by non-manager ";
RAISE NOTICE "User %not found in payroll table ",user_id;
RAISE EXCEPTION "Invalid Entry in Payroll Table..aborting ";
```

不幸的是，PL/pgSQL 没有内置基于 RAISE 事件的检测或错误恢复机制。通过设置指定变量或客户应用中的显式陷阱可以完成以上功能。不过，在大部分情况下——特别是，放弃事务后——自动恢复内容较少；通常在某些时候可以人工干预来完成。

7. 取回系统变量

PL/pgSQL 还允许函数在进程中从 PostgreSQL 后端取回一定的诊断设置值。使用 GET DIAGNOSTICS 来取回 ROW_COUNT 和 RESULT_OID。其语法如下：

```
GET DIAGNOSTICS mycount =ROW_COUNT;
GET DIAGNOSTICS last_id =RESULT_OID;
```

只有在上面代码中立即执行插入动作后的 RESULT_OID 才有意义。

8. 注释

用于定义 PL/pgSQL 代码块的 BEGIN 和 END 语句与 BEGIN...END SQL 事务子句并不相同。SQL BEGIN...END 语句用于定义事务语句的启动和提交。一个 PL/pgSQL 函数自动从属于调用它的 SQL 查询中显式或隐式事务的一部分。因为 PL/pgSQL 并不支持嵌套事务，所以不可能使一个事务成为一个调用函数的一部分。

与标准 SQL 声明一样，在 PL/pgSQL 中，可使用标准符号来定义数组（如 myint INTERGER(5);）。

PL/pgSQL 和 Oracle 过程语言之间的主要区别是 PostgreSQL 可以重载函数，PostgreSQL 中不需要 CURSORS，PostgreSQL 中调用函数允许使用缺省参数，而且 PostgreSQL 使用了转义单引号。（因为函数本身已经存在于引号中，函数中的查询必须使用一系列引号来保持正确的级别）。还有另外一些区别，但大部分区别是针对语法定义的；请参阅有关 Oracle PL/pgSQL 书籍了解详情。

11.4 PL/Tcl

在 PostgreSQL 中创建自定义函数或触发器时，PL/Tcl 语言允许利用信任版本的流行工具命令语言（Tcl）。虽然对 Tcl 语言的完整讲解超出了本书的范围，但我们还是会讲解其中的重要特征并提供一些例子。

普通 Tcl 语言与 PL/Tcl 语言之间的主要区别在于后者以信任方式运行。这就意味着不执行 OS 级别的动作。而且，仅使用有限的 Tcl 命令集。事实上，Tcl 函数不能在 PostgreSQL 中用于创建新数据类型。

如果期望执行 OS 级别操作，可在 PostgreSQL 编程中使用一个更强大的 PL/Tcl 版本，这就是 PL/TclU（Tcl 非信任）。缺省情况下，基本发行版本中并不包含此语言，必须通过命令来添加。但是，读者必须万分小心，因为错误的脚本可能会导致系统发生错误或崩溃。

一般 Tcl 语言初步

PL/Tcl 的许多语法与 Tcl 语言一般都是相同的。下面使用 Tcl 语言的方法概要。

1. 注释语句

与许多脚本语言一样，缺省的注释标识是#号。所有以此符号开头的行都被忽略。例如：

```
CREATE FUNCTION addit (arg1)RETURNS INTEGER AS'
      #Set variable to arg1
      Set myvar $1
      #This is another comment
'LANGUAGE 'pltcl ';
```

2. 变量赋值

Tcl 允许变量赋值。例如，要给一个变量赋值，可以按以下方法：

```
Set myval 10
Set mystr "This is my string "
Set myval_2 myval+100
```

前面两个例子显然易见：变量 myval 设为数字 10，变量 mystr 设为 string。不过，最后一个例子具有欺骗性，初看一下，变量 myval-2 应该等于 110，但实际上它等于字符串 myval+100。要执行变量替换，使用下面语法：

```
Set myval_2 [expr $myval+100 ]
```

PL/Tcl 使用$符号来表示变量的引用。另外，括号（[]）中的内容被看作 Tcl 代码。

3. 控制结构

与所有行进脚本语言一样，Tcl 也拥有决定代码执行路径的流控制机制。例如，标准的 IF 块如下所示：

```
if {conditional-expression}{
      #code block
}
```

Tcl 同时支持 IF…ELSE 控制结构，如：

```
if {conditional-expression}{
      #code block
}
else {
      #something else
}
```

Tcl 还支持标准 WHILE 和 FOR 循环结构，如：

```
while {$x <100}{
      #其他代码
      incr x 1
}
#loop has exited -run more code
```

或者使用 FOR 循环，FOR 循环结构的语法如下：

```
for {initial condition}{test condition}{modification}{
      #其他代码
}
```

例如：

```
for {set x 0}{$x <100}{incr x 1}{
      #其他代码
}
#loop has exited -run more code
```

在上面例子中使用了 Tcl 命令 incr，它根据给定级数增加变量值。(注意：1 为缺省增加级数，这并不需要明确给定。)

Tcl 语言还支持一个更强大的 FOR 类型循环 FOREACH。其基本语法如下：

```
foreach variable(s)list(s){
      #其他代码
}
```

例如：

```
foreach month {Apr May Jun}{
      #Run quarterly report
}
```

另外，使用多变量名和列表可以创建更复杂的 FOREACH 结构。例如：

```
foreach xpoint ypoint {10 200 20 400}{
      #On first run xpoint=10 ,ypoint=200
      #On second run xpoint=20 ,ypoint=400
}
```

或者：

```
foreach xpoint {10,200}ypoint {20 400}{
      #On first run xpoint=10 ,ypoint=200
      #On second run xpoint=20 ,ypoint=400
}
```

Tcl 还支持 SWITCH 控制结构。其基本语法如下：

```
switch option test-expression {
```

```
        test_case1 {code statement }
        test_case2 {code statement }
    default {default code statement }
    }
```

其中的 OPTION 通常指-exact，-glob 或-regexp，它们指定对给定测试条件进行准确匹配、模式匹配或规则表达式匹配。

DEFAULT 关键字用于其他比较测试均失败时才采用的匹配。另外，代码语句中的“-”符号表示下面首行代码语句将主动运行。例如：

```
set myvar "Barry "
switch -glob $myvar {
      arry {puts 1}
      *arry -
      Ba*-
      Bar*{puts 4}
      default {puts "Not Barry ")
}
```

上面例子将返回一个 4。请注意如何使用连续符号“-”连接成一个正确匹配链的。

4. 字符串和列表

Tcl 包含许多与列表和字符串相关的命令（见表 11-1）。

表 11-1　　Tcl 中包含的与列表和字符串相关的命令

命　令	描　　述
list {1 2 3 4}	返回给定元素的列表
concat {1 2} {3 4}	返回元素的连接列表
lappend {1 2} {3 4}	在列表中添加元素
lindex {1 2 3} 2	返回指定的第 N 个元素（从 0 开始）
linsert {1 2 4} 2 {3}	在索引点插入一个项目
join {1 {2 3} 4}	将所有元素联结到一个单独的单调元素中
llength {1 2 3 4}	返回列表的长度（元素）
lreplace {1 2 3} 1 2 a	从索引的 1 到 2 用“a”替换元素
lsearch {a b c} b	返回“b”值搜索后的索引（即 1）
lsort {b z a c}	返回排序列表
split this, is, split ,	根据所给分隔符（在此是逗号）分隔元素

PL/Tcl 语言定义

至此，我们已经讨论了 Tcl 语言的一般特征；在此语言的基础上，PL/ Tcl 语言添加了一些新的功能。

1. 基本结构

下面是 PL/Tcl 语言的基本格式：

```
CREATE FUNCTION function_name (arg1 [,argN ])RETURNS type
      AS '
          #PL/tcl Code
```

```
' LANGUAGE 'pltcl ';
```

它与 PostgreSQL 中的所有 PL 语言用法相似，但对于 PL/pgSQL，必须注意正确使用转义引号中的字符串。

传递给 PL/ Tcl 的参数从$1 开始且按顺序执行，与在 PL/pgSQL 中的工作方式一样。PL/ Tcl 还接受数组形式的参数。传递数组通常要使用它的属性名指向所需元素。例如：

```
CREATE FUNCTION ispaid(payroll_array)RETURNS INTEGER
     AS '
         if {$1(salary)>0}{
             return 1
         }
         if {$1(hourly)>0}{
             return 1
         }
         return 0
     ' LANGUAGE 'pltcl ';
```

2. 全局数据（GD）指南

根据在 PL/ Tcl 内执行查询的本来规律，在 PL/ Tcl 代码块内不同操作间保存全局访问数据很重要。

要做到这一点，PL/ Tcl 使用了一个内部可用数组“GD”。推荐你使用这一变量在过程中发布共享信息。（参见下面使用 GD 变量过程的例子）

3. 从 PL/ Tcl 访问数据

与 PL/pgSQL 不同，你不能简单地在 PL/ Tcl 中嵌入标准 SQL 语句。它提供了一些特殊的内置命令来访问数据库后端。

（1）直接执行查询

spi_exec 命令用于直接向数据库查询引擎提交查询。spi_exec 命令的语法如下：

```
spi_exec -options query {
      loop-statements
}
options                ——下列选项之一：
                           -计数 n 仅从查询中返回 N 行
                           -数组 name 根据给定名称在相关数组中保存结果
query                  ——将要执行的查询字符串
loop-statements   ——对返回的每一行执行对应命令
```

下面是使用 spi_exec 命令的一些例子：

```
spi_exec "SELECT *FROM authors "
spi_exec -count 10 "SELECT *FROM authors ORDER BY name "
spi_exec -array myrecs "SELECT *FROM authors "
```

（2）准备和执行一个查询

前面是通过 spi_exec 命令直接向查询引擎提交查询来执行它。大许多情况下，这一方法较理想。不过，如果你计划多次执行同一基本查询——也许只修改一下查询标准——更有效的办法是先准备（prepare）此查询再执行之。

准备一个查询时，首先提交给查询规划器，然后规划器对提交对象准备和保存一个查询规划。接着就可以使用此规划来执行实际的查询了，如果使用恰当，这一方法可以提高操作性能。

使用 spi_prepare 命令来准备查询，其语法如下：

```
spi_prepare query typelist
query      ——将要执行的 SQL 查询。
typelist   ——如果要从 PL/Tcl 代码中将参数传递给查询，那么必须给
  定它们的数据类型列表。
```

下面是使用 spi_prepare 命令的一个例子。注意使用双反斜杠来正确转义符号$。另外，还注意由$1 PL/ Tcl 变量指定 VARCHAR 数据类型。

```
spi_prepare "SELECT *FROM authors WHERE name=\\$1 " VARCHAR
```

查询准备完毕，就可以执行 spi_execp 命令了。此命令与 spi_exec 命令相似，不同的是，spi_execp 命令用于执行已经准备好的查询。spi_execp 命令的语法如下：

```
spi_execp options queryID value-list {
loop-statements
}
options              ——下列选项之一：
                        .计数 n 仅从查询中返回 N 行。
                        .数组 name 根据给定名称在相关数组中保存结果。
                        .空值 str 使用指定所有字符串值为非空值。
queryID              —— 一从 spi_execp 返回的查询 OID。
value-list           ——若给 spi_execp 提供了 typelist，那么，这些值的列表将会
                        提供给 spi_execp。
loop-statements      —— 对返回的每一行都将执行的一个 PL/Tcl 语句。
```

下面是一个使用 spi_execp 命令的例子，注意其中 GD 全局系统变量的使用。特别是，下面例子只在首次调用时创建查询规划；对于后续调用，只简单执行前面保存过的规划：

```
CREATE FUNCTION count_checks(int4)RETURNS int4 AS '
    #Check to see if plan exists
    if {![ info exists GD((plan)]}{
         set GD(plan)[ spi__prepare "SELECT count(*)AS
         chk_count FROM payroll WHERE emp_id=\\$1 " int 4 ]
    }
    #Plan has been created or already exists
    spi_execp $GD(plan)[ list $$1 ]
    return $chk_count
' LANGUAGE 'pltcl ';
```

（3）构造查询

访问 PostgreSQL 后端时一个有用的语句就是 quote。此命令对于使用变量替换构造查询字符串很有用。下面是一个使用 quote 命令的例子：

```
set myval "Barry "
quote "SELECT *FROM authors WHERE name=$myval "
```

如果发送到查询解析器，上例产生的文本如下：

```
"SELECT *FROM authors WHERE name='Barry '"
```

还要注意容易忽略的一点，这就是变量值已经包含一个单引号或双引号的情况。quote 命令很难应付这种情况，此时，可能从 PostgreSQL 查询解析器产生错误。请看下面例子：

```
set myval "Barrys '"
quote "SELECT *FROM authors WHERE name=$myval "
```

如果发送到查询解析器，上例产生的文本如下：

```
"SELECT *FROM authors WHERE name='Barry 's "
```

为了纠正这一明显的错误，可使用下面语法：

```
set myval "Barrys '"
"SELECT *FROM authors WHERE name='[quote $myval ]'"
```

（4）访问 PostgreSQL 日志系统

与 PL/pgSQL 一样，PL/Tcl 提供了访问 PostgreSQL 日志系统的命令。elog 命令的使用语法如下：

```
elog level message
level     ——NOTICE、ERROR、FATAL、DEBUG 或 NOIND。
message   ——传递给日志的文本信息。
```

（欲了解上面所涉及的 elog 级别的更详细信息，请参考第 13 章“客户端编程”中对 elog C 函数的讨论。）

4. 注释

当安装 PL/Tcl 时——无论在编译时还是在编译后——要想成功安装，必须确保目标系统中已经存在 Tcl 语言及相关库。

11.5 PL/Perl

Perl 是最常用的脚本语言之一。它几乎可以在任务平台上运行而且得到了广泛开发团体的支持。正由于这些原因，在选择 PostgreSQL PL 语言时，PL/Perl 可作为一个较理想的选择。

与 PL/Tcl 一样，PostgreSQL 仅允许 PL/Perl 使用指定的受信任的命令。实质上，Perl 中的任何显式处理文件系统、环境设置或外部模块的命令都被禁止。

不过，PL/Perl 中生成的错误代码仍然可能对系统产生消极影响。大部分问题的产生原因在于 PL/Perl 仍然允许消耗内存及创建无限循环。所以，必须对 PL/Perl 所创建代码严格检查确保代码运行后不会对现有系统造成问题。

一般 Perl 语言初步

PL/Perl 中的许多语法与一般的 Perl 相同。下面概要介绍 Perl 语言的使用。显然，如果你是 Perl 的初学者，请参阅有关书籍或访问适合于初学者的相关站点。

1. 注释语句

与许多脚本语言一样，缺省的注释标识是#号。任何以此符号开始的整行将被忽略。

2. 控制结构

Perl 包含大部分其他语言中具有的常规控制结构。标准 IF 结构如下：

```
if (expression ){
      code-statement
}
```

Perl 同时支持更复杂的 IF 语句，如 IF…ELSE 和 ELSEIF 语句。例如：

```
if (expression ){
      code-statement
}elseif (another-expression ){
      other-code-statement;
}elseif (another-expression ){
      other-code-statement;
}else (final-expression ){
      final-code;
}
```

注意上面代码是如何联合 ELSEIF 和 ELSE 语句来创建一个测试条件链的，如果无测试条件满足要求，则执行最后的缺省语句。

Perl 还支持 WHILE、UNTIL、DO、FOR 和 FOREACH 循环，下面是相关例子：

```
while ($a<10){
    print $a;
    $a++;
}

until ($a>10){
    print $a;
    $a++;
}

do {
    print $a;
    $a++;
}while ($a<10)

for ($a=0;$a<10;$a++){
     print "Printing 10 times …";
}
@lst =("Jan ","Feb ","Mar ");
foreach $a (@lst){
     print $a;
}
```

Perl 还包含跳出控制结构的方法，如 LAST、NEXT 和 REDO 以及运用标签块。例如：

```
while ($a<10){
    print $a;
    if ($a=5){
```

```
            #a is 5,so exit loop
            last;
        }
        $a++;
    }
    print "exited loop ";
```

上面代码将继续循环，直到满足两个条件之一：或$a 大于或等于 10；或$a 等于 5（实际上，在这一例子中，代码永远不会到 10，因为到 5 时就退出了）。

其他语句与之工作相似。NEXT 语句将重复此循环并忽略余下的代码；REDO 语句不验证测试条件重复运行此循环。例如：

```
while ($a<10){
    $a++;
    print $a;
    if ($a=5){
        #a is 5,so loop again
        next;
    }
}
print "exited loop ";
```

除了重复循环外，定义标签可与 NEXT、LAST 和 REDO 语句结合使用来控制程序流程：

```
OUTER:while ($a<10){
    $a++;
    print $a;
    INNER:if ($a=5){
        REALLYINNER:if ($b=1){
        last OUTER;
    }
    }
}
print "exited loop ";
```

3. 联合数组

Perl 语言的强大功能之一就是创建和操纵联合数组。下面的例子创建了一个二维数组并给予赋值：

```
$employee("name ")="Fred ";
$employee("age ")=29;
```

为了获得数组中的键列表，可以使用 keys 函数。例如：

```
@lst =keys(%employee);
#lst now equals ("name ","age ")
```

或者，你希望列出保存在数组中的值，你可以使用 values 函数。例如：

```
@lst =values(%employee);
#lst now equals ("Fred ",29)
```

如果你希望一起返回键及其值，可使用 each 函数。此函数可用于一个循环内，对每一个连续的调用，它将返回下一个键/值对。例如：

```
while (($name,$age)=each(%employee)){
      #其他代码…
}
```

要从一个联合数组中删除一个元素，可以使用 delete 函数。如下列所示：

```
$employee("name ")="Fred ";
$employee("age ")=29;
$employee("shoesize ")=10;
#The employee array is 3 elements wide
delete $employee("shoesize ");
#Now just 2
```

4. 数组访问函数

在 Perl 中，数组序号从 0 开始并对其中的元素顺序编号。可以使用一个以逗号隔开的列表来定义元素列表。负数指从元素列表结尾开始的元素。下面是一个简单的数组元素列表例子：

```
@lst=("one ","two ","three ","four ");
#Set tmp variable to 'one '
$tmp=$lst [0 ];
#Set tmp to 'two ' and 'four '
$tmp=$lst [1,3 ];
#Set tmp to 'four '
$tmp=$lst [-1 ];
```

数组的一个普通用法就是作为一个装载信息的队列。队列一般需要按可预见和指定的方式删除和添加元素。**Perl** 包含几个完成以上功能的函数：pop、push、shift 和 unshift。pop 和 push 函数作用于数组的右边，shift 和 unshift 作用于左边。

例如：

```
@queue=(54,123,65643);
#Return and Remove 65643
$myval=pop(@queue);
#Add 111 to queue
push(@queue,111);
#
#Return and Remove 54
$myval=shift(@queue);
#Add 222 to left side
unshift(@queue,222);
```

要保留或重排序元素列表，可以使用 reserve 或 sort 函数，如下所示：

```
@lst=(10,1,5);
@lst=reverse(@lst);#Now lst =(5,1,10)
@lst=sort(@lst);#Now lst =(1,5,10)
```

5. Perl 和规则表达式

Perl 之所以广泛应用的原因之一就是规则表达式的运用。实际上，规则表达式是给定模板和源文本间匹配模式的一种方法。regex 的完整解释不在本书范围之内，不过，表 11-2

和表 11-3 提供了部分例子。

表 11-2 regex 中模式匹配所用字符

regex 模式	描　　述
/anytext/	anytext 为搜索目标文本
.	代表任何字符
*	0 或更多的先前字符
+	1 或更多的先前字符
?	任何单个字符
[^anytext]	不包含 anytext
^anytext	以文本 anytext 开始

表 11-3 搜索及其结果示例

源文本	regex 模板	注释/匹配
Pos*greSQL is good	/POSTGRESQL/	无匹配。大小写敏感
	/POSTGRESQL/i	匹配。大小不写敏感
	/eS/	eS 匹配（PostgreSQL）
	' /	匹配（发现空格）
	/[efgh]ood/	匹配（good）
	/[e-h]ood/	匹配。与上相同（good）
	/Postgre[^SQL]/	无匹配。非“SQL”起始
	/^P/	匹配。以“P”起始（PostgreSQL）
	/g..d/	匹配（good）
	/g.*d/	匹配（good）
	/goo*/	无匹配。不以“o”结尾
	'i*s/	匹配，0 匹配“i”（is）
	/i+s/	无匹配（is）
	/PostgreSQl?/	匹配（PostgreSQL）
	!/good/	无匹配。非“good”

PL/Perl 语言定义

PL/Perl 语言的基本格式如下：

```
CREATE FUNCTION name (Arg1 [,ArgN ])RETURNS type AS '
     Return $_[0 ]'
LANGUAGE 'plperl ';
```

1. 转义字符

与 PL/pgSQL 和 PL/Tcl 一样，记住 PL/Perl 函数之内的引号字符串需要正确转义很重要。可以使用 Perl 函数 q[]、qq[]和 qw[]创建正确的转义变量替换序列。

2. 变量替换

缺省情况下，变量以“$_”形式传递给相应的 Perl 函数。当没有显式指定变量时，缺省的变量就是 Perl 的名称空间。因而，此名称空间将继承 PL/Perl 变量。如：

```
CREATE FUNCTION getproduct(INTEGER,INTEGER)RETURNS INTEGER AS '
```

```
        $newval=$_[0 ] *$_[1 ];
        return $newval;'
LANGUAGE 'plperl ';
```

而且，整个元组都可以传递给 PL/Perl 函数。在 PL/Perl 代码内，联合数组的键是传递元组的字段名。显然，联合数组的值保存了字段数据。例如：

```
CREATE FUNCTION citystate(employee)RETURNS INTEGER AS '
        $empl =shift;
        return $empl->{"city "}+$empl->{"state "};
LANGUAGE 'plperl ';
```

3. 注释

当安装 PL/Perl 时——不管在编译时还是在编译后——为保证安装成功，需要确保目标系统中已存在 Perl 语言和相关库。而且，共享库版本的 `libperl` (即 `libperl.so`)必须存在，使 PostgreSQL 能顺利访问。

第 12 章　创建自定义函数

对于数据库自身而言，除了装载数据外一无所有。真正使数据库发挥作用的是函数和其他工具。设计一个有效数据库的许多工作就是构建所需要的交易规则模型。开发适当的表结构是在数据库内构建交易模型的方法之一；另一个方法就是通过创建自定义函数、触发器和规则来完成。

作为一个简单地回顾，对函数、触发器和规则比较如下：

- **函数**　用户定义函数（也叫存储过程）是最有用的在后端内部实现常规调用代码的途径。
- **触发器**　运行 SELECT、INSERT 或 UPDATE SQL 命令时，通过触发器可以执行其他动作。与规则不同的是，它们是基于逐行进行调用。
- **规则**　运行 SELECT、INSERT 或 UPDATE 命令时，规则可以自动重写给定查询来实现替换或其他动作。规则与触发器不同的是，当动作影响到其他表时才使用规则。

12.1　创建自定义函数

PostgreSQL 包含许多预定义函数来帮助处理数据（参见第 4 章“PostgreSQL 函数”了解函数完整列表）。其中的一部分函数用于一般目的的转换、格式化或聚集数据。

在许多情况下，系统则需要使用用户自定义函数。当同样的信息需要重复访问时，函数就能大显身手。在这种情况下，就可以创建一个保存于服务器中的用户自定义函数。

一般，当创建一个函数时，首先预编译查询规划并且保存起来以备执行。这对提高系统速度大有好处，因为客户应用仅需要请求执行此函数，而不必提供 SQL 代码以及等待查询的解析、执行并返回值。

使用自创建函数的好处不仅限于速度方面。在许多场合，标准 SQL 语言不能对期望的动作提供足够的控制。例如，如果需要条件分支、循环重复或复杂变量替换，创建自定义函数是目前唯一能胜任的方法。

如第 11 章“服务器端编程”中所讲，PostgreSQL 包含许多过程语言，它们可以用于编写自定义函数。虽然每种语言都有自己的特色，整体权衡之下，PL/pgSQL 恐怕最值得一用。

PL/pgSQL 允许包含标准 SQL 命令，同时允许包含更高级的控制结构，像循环、if-then-else 结构及变量替换。

使用示例

在这一节，你将看到几种对创建自定义函数发挥作用的实例。

1. 代码再利用

前面已经提到，需要使用自定义函数的一种情况就是你希望避免重复性的工作。

在此例中，让我们看看名称为 homestate 的函数，它接受一个 employee id 值并返回此 employee 的家庭状况。

```
CREATE FUNCTION homestate(int)RETURNS text
AS'
      DECLARE
            empid ALIAS FOR $1;
      BEGIN
            SELECT state FROM payroll WHERE
            employee_id=empid;
            IF FOUND THEN
            RETURN state;
            ELSE
            RETURN "N/A ";
            END IF;
      END
'LANGUAGE 'plpgsql ';
```

创建上面的函数时，PostgreSQL 对表 payroll 执行一个查询规划并等待它的执行。

严格设计的数据库应包括许多这样预设的查询，它们总的来说给系统带来了如下好处：

- **速度和效率**　执行过的查询规划已经存在于数据库引擎中并等待其运行。
- **对返回数据的格式控制**　在上一个例子中，当没有发现一个职员的家庭状况时，则自动返回一个“N/A”值。它由自定义函数直接提供，而无需任何客户端的干预步骤。
- **精简**　在上一个例子中，客户应用不直接具备数据库结构的知识。因而基本表结构可能变化很大，但只要函数输入输出接口保持相同方式，客户应用将无需任何修改。
- **间接优点**　仅包含此类预定义查询函数的另一好处就是改进了数据库和应用结构。也就是说，它强迫程序员/DBA 考虑访问的大部分数据类型。所以，它可以导致系统整体上效率的提高。

2. 函数捆绑

通过捆绑函数可以创建一个更复杂但一致的数据库。

在此一例子中，你将创建一个指定的用户接口（UI）功能。当用户向系统中输入信息时，程序员应注意，用户希望能在对话框中输入职员名称或职员 ID。

如果假定所有的职员 ID 严格由数字组成，而职员名称仅由字母组成，则完成此任务相对简单得多。

不必对每个应用实例定义此特征，只要创建一个简单接受输入（ID 或名称）并返回职员 ID 的通用函数即可。

而且，仅将职员 ID 保存在与 payroll 表联结的表中。它有助于数据标准化的实现，此思想是为了保持数据的一致性以及数据库保存数据的非冗余性。

所以，可以编写下面函数来接受任一种格式并返回职员的 ID。（此函数的功能直到最后才可全部显现）

```
CREATE FUNCTION getempid(varchar)RETURNS int
AS '
     DECLARE
         empval ALIAS FOR $1;
     BEGIN
         /*Determine if empval is name
             if so,return the emp_id,
             otherwise,return back the emp_id
         */
         IF empval ~'[a-zA-Z ]' THEN
             SELECT employee_id FROM payroll WHERE
                 last_name=empval;
             RETURN last_name
         ELSE
             RETURN empval;
         END IF;
     END
 'LANGUAGE 'plpgsql ';
```

初看此函数，它似乎并非如此有用。这只是简单地判断传递的变量是否是一个数字或字母，并返回此职员的 ID。而且，好像如果此函数已经传递了职员的 ID，它只是简单地将返回值直接返回。从表面上看，这似乎是一种浪费。然而，当它与其他函数捆绑时，它的真正威力就体现出来了。

例如，将第一个函数 homestate 与最后一个函数捆绑，你可以通过它接受职员的姓或职员的 ID。在这种情况下，你使用最后一个函数用作一个封装器以确保输入值的灵活范围。客户端代码如下：

```
SELECT homestate(getempid('Stinson '));
```

或

```
SELECT homestate(getempid(592915));
```

或，最后

```
SELECT homestate(getempid(strInputValue));
```

通过捆绑此两个函数，就允许输入更灵活范围的数据，而仍然以一致性格式在后端保存数据。而且，如果开发人有一天意识到他们应该允许用户同时输入社会安全号，他们只要修改一个 getempid 函数就行了。

3. 存储过程

实际上，存储过程与函数属于相同的事物。也就是说，它们是通过 CREATE FUNCTION 命令创建的语句代码集。所不同的就是概念上的名词差异。

一般，函数接受一个输入值，对其进行一些查找和处理，然后返回一个输出值。一个经典的函数例子就是 upper 函数。此函数接受一个字符串，将它转换成大写，并返回处理过的字符串。例如：

```
>select upper('abcdefg ');
```

```
>ABCDEFG
```

不过，存储过程不只限于接受一值并返回数据。一般，它们执行一些基本的过程或改变数据库表。例如，请看下面的例子。

在此例中，我们希望使用一种简单的办法在表 employee 中执行调整或插入。用户肯定需要一个简单方法对新来或现有职员进行新工作描述。此函数的功能要求如下：

- 如果职员不存在，则添加此人并分配指定的工作给他或她。
- 如果职员已经存在，则更换其工作。

根据以上规定，一个完成此任务的存储过程例子为：

```
CREATE FUNCTION assignjob(int,varchar)RETURNS int
AS'
      DECLARE
        empid ALIAS FOR $1;
        jobdesc ALIAS FOR $2;
        retval INTEGER:=0;
        emprec RECORD;
      BEGIN
        /*Determine if employee exist in table*/
        SELECT INTO emprec WHERE employee_id=empid;
        IF FOUND THEN
            /*Emp exist,modify his job description*/
            UPDATE employee SET job_description=jondesc
            WHERE employee_id=empid;
            retval :=1;
        ELSE
            /*Emp doesn 't exist,add him*/
            INSERT INTO employee VALUES (empid,jobdesc);
            retval :=1;
        END IF;
        RETURN retval;
      END
' LANGUAGE 'plpgsql ';
```

虽然上一个例子仍被视作一个函数，但实际上，它对数据库表进行了修改而不仅仅是计算返回值。因此，像这种示例被称作为存储过程。

这可能仅仅看作语义上的一种差别。它指出了两者概念上的差异。

存储过程对于完成表的定期自动操纵极其有用。例如，废除工资支票就是一个很好的例子。一般，此过程需要在标准帐号系统设置中修改很多表。虽然可以直接通过在客户应用中编程解决，但有可能最后导致更为复杂的应用。

例如，在当前系统中，需要修改表 payroll、employee、AP 和 GL 来废除打印支票。实际上，直接从客户计算机上编程此过程代码完成此任务毫无问题。但如果将来又有一个新表 JobCost 需要修改，以上修改就可能成为复杂但徒劳的工作。它可能需要在成打或成百个应用中修改代码。

一个较好的办法就是在数据库后端利用存储过程（即函数）来完成废除支票的工作。这

一方法的好处是，客户只需要简单地调用函数 voidcheck 并将剩下的实际工作留给服务器来完成。服务器端，对函数的修改而影响其他表相当小，所以，整个系统更加灵活。

12.2 创建自定义触发器

在存储过程（即函数）和触发器间有一定程度的功能重复。它们都是执行通过 CREATE FUNCTION 语句创建的代码。不过，更常用触发器来自动响应表相关事件，而不是作为被客户应用直接调用的动作。

触发器通过 CREATE FUNCTION 语句将函数捆绑到 DELETE、UPDATE 或 INSERT 表事件中。客户应用不能直接感觉触发器的存在；它只是简单执行请求动作，它将导致服务器引发适当的触发器事件。

触发器用于执行与访问表对应的动作。触发器通常用作一个确保数据或交易规则完整性的机制。例如，请看下面的函数及触发器对：

```
CREATE FUNCTION trig_insert_update_check_emp()RETURNS opaque AS
     'BEGIN
         /*Check employee age,state,and name
         and enforce certain checks */
         IF new.age >20 THEN
                new.adult ='yes ';
         ELSE
                new.adult ='no ';
         END IF;
         IF new.state ~'^[A-Za-z ][A-Za-z ]$' THEN
                new.state =upper(new.state);
         ELSE
                RAISE EXCEPTION 'Alphabetical State Desc Only ';
         END IF;
         IF new.name ~'^[a-zA-Z ]*' THEN
                new.name =initcap(new.name);
         ELSE
                RAISE EXCEPTION 'Alphabetical Name Only ';
         END IF;
     END;
' LANGUAGE 'plpgsql ';

CREATE FUNCTION trig_delete_check_emp()RETURNS opaque AS
     'BEGIN
         /*Make sure a manager isn 't deleted*/
         IF old.manager='yes ' THEN
                RAISE EXCEPTION 'Cannot Delete Managers!'
         END IF;
```

```
        END;
    ' LANGUAGE 'plpgsql ';
```

上例中的两个函数使用了 new 和 old 关键字。当作为一个触发器事件部分调用时，这些关键字指那些刚刚插入或删除的数据。下面，创建一个触发器事件并捆绑于每个函数上。

```
CREATE TRIGGER employee_insert_update
      BEFORE INSERT OR UPDATE
      ON employee
      FOR EACH ROW EXECUTE PROCEDURE trig_insert_update_check_emp();

CREATE TRIGGER employee_update
      BEFORE DELETE
      ON employee
      FOR EACH ROW EXECUTE PROCEDURE trig_delete_check_emp();
```

现在，已经创建了触发器，它们可以测试如下：

```
>INSERT INTO employee (name,age,state,manager)
     VALUES ('sean ',29,'T8 ','yes ');
>ERROR:Alphabetical State Desc Only

>INSERT INTO employee (name,age,state,manager)
     VALUES ('sean ',29,'tx ','yes ');
>INSERT 323003 1

>SELECT *FROM employee WHERE name='Sean ';
name  age     state   manager     adult
-------------------------------------
Sean  29      TX      yes         yes

>DELETE FROM employee WHERE name='Sean ';
>ERROR:Cannot Delete Managers!

>UPDATE employee SET manager='no ' WHERE name=='Sean ';
>UPDATE 1

>DELETE FROM employee WHERE name='Sean ';
>DELETE 1
```

在上面的例子中，请注意触发器行为与列约束典型行为方式的相似性。列约束一般在允许插入（INSERT）或修改（UPDATE）前检查指定字段的有效性。

不过，触发器和列约束在行为上并非相互排斥。如果创建触发器时使用了关键字 BEFORE，则会在字段（或表）约束检查前引发触发器。所以，如果触发器依赖于 OID 或特定的索引，它将不能正常工作。

与之相似，当指定了 AFTER 关键字时，触发器将在指定表动作（INSERT、UPDATE 或 DELETE）完成后引发。而且，关键字 AFTER 的使用将导致直到所有的表或字段约束被处理过才被引发。

12.3 创建自定义规则

规则与触发器在概念上很相似，但也有一些根本性的区别。触发器通常专指被处理的表，而规则作用于外部表。另外，除了动作正在执行外，触发器均可被引发。例如，无论在插入动作完成前或完成后，INSERT 触发器均可引发此事件。

另一方面，创建规则也可使用可选关键字 INSTEAD。这种情况下，规则在指定的动作中执行。

规则的一个典型用途是，当在指定表中发生表相关事件时，在外部表中执行动作。规则的一个简单用处是对重要表的日志跟踪。例如，假设管理者需要查看开销大于$1000 的每周报表。你可以按下面方法实现：

```
CREATE RULE log_payables AS
    ON INSERT TO accounts_payable
    WHERE new.amount >1000 DO
    INSERT INTO audit VALUES(new.vendor_id,new.amount,new.user);
```

你可以按下面方法测试规则：

```
>SELECT *FROM accounts_payable;

vendor amount user
--------------------
0 Results Found

>SELECT *FROM audit;

vendor    amount      user
--------------------
0 Results Found

>INSERT INTO accounts_payable VALUES('Acme,Inc ',2500,'Sean ');
INSERT 231431 1
INSERT 231432 1

SELECT *FROM audit;

vendor        amount          user
------------------------------
Acme,Inc  2500.00         Sean
```

规则还可与函数配合使用创建更复杂的动作。例如，假设在数据库中有两个表 payroll 和 paytotals。payroll 用于保存发放工资支票的单独记录。而表 paytotals 保存每个职员年度工资发放总数。

在此例中，假设系统自动保持表 paytotals 为最新状态是一项重要工作。完成此任务

就可使用规则/函数组合体，如下所示：

```
CREATE FUNCTION getpaytotal(int)RETURN real AS
      'DECLARE
              empid ALIAS FOR $1;
      BEGIN
              SELECT sum(amount)AS paysum FROM payroll
              WHERE employee_id=empid;
              RETURN paysum;
      END;
' LANGUAGE 'plpgsql ';
```

创建此函数后，再添加规则集：

```
CREATE RULE compute_paytotal AS
    ON INSERT TO payroll DO
    UPDATE paytotals SET amount=getpaytotal(new.employee_id)
         WHERE paytotals.employee_id=new.employee_id;
```

最后的规则可测试如下：

```
>SELECT *FROM payroll;

empid     name     amount  checknum
--------------------------------
123       Sean     100.00  5411
123       Sean     100.00  5412

>SELECT *FROM paytotals;

empid     name     amount
--------------------
123       Sean     200.00

>INSERT INTO payroll VALUES (123,'Sean ',200,5413);
INSERT 243411 1
UPDATE 1

>SELECT *FROM paytotals;

empid     name     amount
--------------------
123       Sean     400.00
```

正如前面所提到的，CREATE RULE 命令也允许包含关键字 INSTEAD。当指定此关键字后，将执行另一个动作。对于前面的例子，我们假设管理者决定表 accounts_payable 中的所有超过$1000 的帐务都必须转入另一个表中，直到通过审核为止。例如：

```
CREATE RULE defer_ap AS
    ON INSERT TO accounts_payable
```

```
    WHERE new.amount >1000 DO INSTEAD
    INSERT INTO ap_hold VALUES (new.vendor,new.amount,new.user);
```

这将导致对表 accounts_payable 中的任何插入动作都将转向表 ap_hold，等待通过管理者的审核。

与触发器不一样的是，还可以在 SELECT 语句内定义规则。利用它这一点可以实现一些有趣的应用。例如，请看下面的例子：

```
>SELECT *FROM accounts_payable;
vendor        amount  user
-------------------------
Widgets       500.00  Barry
Acme,Inc  2500.00     Sean

>CREATE TABLE my_ap INHERITS (accounts_payable);

>SELECT *FROM my_ap;

vendor        amount  user
-------------------------
Widgets       500.00  Barry
Acme,Inc  2500.00     Sean

>CREATE RULE my_select AS
    ON SELECT TO my_ap DO INSTEAD
    SELECT *FROM my_ap WHERE amount>1000;

>SELECT *FROM my_ap;

vendor        amount  user
-------------------------
Acme,Inc  2500.00     Sean
```

在上面例子中，我们创建了一个继承所有基本表属性的表。然后，在新表中定义了一个规则，用于重写 SELECT 语句并强迫施加一个匹配标准。其动作看起来就像一个标准的 VIEW 命令一样。这并非偶然，因为 PostgreSQL 经常将规则作为实现 CREATE VIEW 命令的一种方法。

注释和其他方面

显然，在很多场合使用规则集很有利。但当设计规则时必须小心，因为这里存在误用的可能。例如，请看下面两个相关规则：

```
CREATE RULE insert_1 AS
    ON INSERT TO apple DO
    INSERT INTO orange VALUES(new.weight);

CREATE RULE insert_2 AS
```

```
ON INSERT TO orange DO
INSERT INTO apple VALUES(new.weight);
```

此例显示了一个危险的规则使用方法。在这里，如果对任意一个表使用 INSERT 命令，则会启动一个无限循环的级联插入动作。实际上，PostgreSQL 如此智能化，它不会允许此类情况的发生，当执行太多的递归查询时，PostgreSQL 会自动终止此动作。

不过，一般来讲，规则应该仅针对无任何现有关联规则的表。也就是说，一个规则集应该与另一个规则集分别对待。在拥有上百个表的大型数据库中，如果使用了无数个规则，那么管理和预测结果将变得十分复杂。

另外，规则只是访问指定的系统类，即访问 OID 属性。这就意味着规则定义不能直接在任何系统属性上起作用。所以，如果表被看作一个系统的类，函数（如 func(table)）就会失败。

特定规则的代码体可通过查看 pg_rules 目录来访问。

第 13 章　客户端编程

PostgreSQL 提供了许多允许客户应用访问后端数据库的接口。除了 PostgreSQL 提供的 API 外，很多其他语言也提供了针对 PostgreSQL 的接口。

客户端语言的选择取决于许多因素。C 和 C++在精确控制和执行效率上表现优秀，Python 和 Perl 是快速利用原型及获取灵活性的理想选择，PHP 是一个良好的基于 web 的解决方案，而 ODBC 和 JDBC 提供了从 Windows 或 Java 客户端访问的便利。每种语言所提供的接口都将下面章节中介绍。

13.1　ecpg

ecpg 是一套设计用于简单实现在 C 源代码中包含 SQL 命令的应用程序和库。将 SQL 嵌入 C 中，或 ecpg 中，是一个被许多 RDBMS 支持的多平台工具。

嵌入 SQL 所隐含的思想就是让开发者可以直接地在他或她的 C 源代码中输入 SQL 查询。而且 ecpg 前处理器将那些简单的 SQL 语句翻译成更复杂的函数，从而避免了许多需要开发者完成的工作。

ecpg 程序的输出是标准的 C 代码；它可以与 libpq 和 ecpg 库相连接并直接编译成二进制代码。

通过 ecpg 创建一个程序的一般流程如图 13-1 所示。

程序（内嵌 SQL 命令）
|
[ecpg 前处理器]
|
标准 C 代码输出
|
[标准 C 编译器（如 gcc）]
|
执行代码

图 13-1　通过 ecpg 创建程序的流程

有关 ecpg 的行命令选项完整讨论可参考第 6 章“用户可执行文件”中的“ecpg”一节。

嵌入 SQL 使用下面即将介绍的语法来执行标准数据库操作。

声明和定义变量

与 PostgreSQL 间进行数据传递时，下面的代码用于定义 C 程序中所需的变量。

```
EXEC SQL BEGIN DECLARE SECTION;
[…Variable Definitions…]
EXEC SQL END DECLARE SECTION;
```

例如，下面代码用来定义装载假想数据库中 employee_id 和 employee_name 的变量：

```
EXEC SQL BEGIN DECLARE SECTION;
int empl_id;
varchar empl_name[30];
EXEC SQL END DECLARE SECTION;
```

显然，这一部分代码必须在使用 empl_id 和 empl_name 变量之前执行，而且，此类型必须与相应的 PostgreSQL 数据类型匹配。表 13-1 简单列出了 PostgreSQL 和标准 C 数据类型间的匹配情况。

表 13-1　　PostgreSQL 和标准 C 数据类型间的匹配

PostgreSQL	C
SMALLINT	short
INTEGER	int
INT2	short
INT4	int
FLOAT	float
FLOAT4	float
FLOAT8	double
DOUBLE	double
DECIMAL(p,s)	double
CHAR(n)	char x[n+1]
VARCHAR(n)	struct
DATE	char[12]
TIME	char[9]
TIMESTAMP	char[28]

连接到数据库

C 中嵌入 SQL 连接到后端服务器时使用以下语法：

```
EXEC SQL CONNECT TO dbname
```

实际的数据库名称可按下面方法指定：

```
dbname[@server][:port]
```

执行查询

与数据库连接成功后，可以使用以下语法将查询送入后端处理：

```
EXEC SQL query
```

一般，几乎所有的查询动作要求运行一个显式 COMMIT 命令。SELECT 命令除外：它们可以单行运行。下面是一些典型用法：

```
EXEC SQL SELECT * FROM payroll WHERE name='Jason Smith';

EXEC SQL SELECT INTO payroll VALUES
     ('Steven','Berkeley','CA');
EXEC SQL COMMIT;

EXEC SQL UPDATE payroll SET l_name='Wickes'
     WHERE f_name=' Steven';
EXEC SQL COMMIT;

EXEC SQL DECLARE my_cur CURSOR FOR
     SELECT * FROM payroll
     WHERE state='CA';
EXEC SQL FETCH my_cur INTO :name;
[…其他代码…]
EXEC SQL CLOSE my_cur;
EXEC SQL COMMIT;

EXEC SQL SELECT * FROM payroll;
EXEC SQL COMMIT;
```

错误处理

必须通过下面命令来定义 ecpg 的通信区：

```
EXEC SQL INCLUDE sqlca;
```

另外，可以通过下面语法打开错误报表：

```
EXEC SQL WHENEVER sqlerror sqlprint;
```

13.2 JDBC

PostgreSQL 提供了一种类型 4 的 JDBC 驱动程序。类型 4 表示此驱动程序由纯 Java 编写与平台无关。所以，一旦编译完成，此驱动程序就可以应用于任何系统。

编译驱动

要在系统编译时生成驱动程序，需要 configure 命令的--with -java 选项。要不然，如果系统已经安装，你仍然可以输入目录/src/interfaces/jdbc 和运行 make install 命令来编译它。

如果你已经有了一个基于包安装的系统，你也可以找到适当的安装 JDBC 的 RPM 包（例如，postgresql-jdbc-7.1.2-4PGDG.i386.rpm）。一旦安装后，它通常位于

/usr/share/pgsql 之中。

完成后，JDBC 驱动将位于当前目录中，即

postgresql.jar

安装驱动

为了安装驱动，需要在环境变量 CLASSPATH 中包含 jar 文件 postgresql.jar。例如，通过假想 Java 应用程序 foo.jar 装载驱动，可以使用下面命令（假设使用 Bash shell）：

```
$ CLASSPATH=/usr/local/pgsql/lib/postgresql.jar
$ export CLASSPATH
$ java ./foo.jar
```

配置客户端

任何使用 JDBC 的 Java 源程序需要使用下面命令导入 java.sql 包：

```
import java.sql.*;
```

不要导入 postgresql 包

不要导入 postgresql 包。如果这样做，源代码将不能编译。

连接

要进行连接，你需要从 JDBC 获得一个连接事例，可以使用 DriverManager.getConnection()方法：

```
Connection db=DriverManager.getConnection(url,user,pwd);
```

其中的选项可参见表 13-2。

表 13-2　JDBC 连接选项一

选　项	描　述
url	数据库 URL
User	用以连接的用户名
pwd	用户密码

通过 JDBC，数据库以统一资源定位器（URL）方式提交。通过 PostgreSQL，可采用下列形式之一：

```
jdbc:postgresql:database
jdbc:postgresql://host/database
jdbc:postgresql://hostport/database
```

其中的选项可参见表 13-3。

表 13-3　JDBC 连接选项二

选　项	描　述
database	数据库名
host	服务器主机名（缺省为本地主机）
port	服务器端口数（缺省为 5432）

执行查询

要向数据库提交查询，需要一个语句对象。excuteQuery()方法返回一个包含返回值的 ResultSet 事例。例如：

```
Statement myst=db.createStatement();
ResultSet myst=myst.executeQuery("SELECT * FROM payroll");
[…更多代码…]
```

在允许访问实际的 ResultSet 前，必须首先调用一个 rs.next()函数。如果存在多个结果则返回一个真值并准备所处理的元组。

修改记录

要修改指定元素或执行一个不会导致ResultSet的语句，可以使用executeUpdate()方法。例如：

```
Statement myst=db.createStatement();
myst.executeUpdate("UPDATE payroll SET first_name='Stevee'";
```

13.3 libpq

libpq 库是一个 C 语言 API，它提供了访问 PostgreSQL 后端的接口。事实上，大部分客户工具（如 psql）均使用此库作为连接到后端的途径。

libpq 提供的函数几乎能控制客户/服务器的各个方面。虽然对每个函数的深入讨论超出了本章的范围，但在这里还是要讨论两个最流行的函数，它们是：

- PQconnectdb 连接到数据库。
- PQexec 发送一个查询到后端并执行。

PQconnectdb 同时还提供了下列函数：

- PQreset 重置客户/服务器通信。
- PQfinish 关闭数据库连接。

PQconnectdb 和 PQexec 将在下面章节中详细讨论。

PQconnectdb

Pqconnectdb 函数接受表 13-4 中几类选项。在这里，你的用户定义对象名称是 PGconnectID，但也可以为其他形式。

```
PGconnectID *PQconnectdb(const char *conninfo)
```

conninfo 包含表 13-4 中选项之一（选项形式为：选项=值）。

表 13-4　　PQconnectdb 接受的选项

选　项	描　　述
host	连接的主机名（或 UNIX 路径）
hostaddr	对于 TCP/IP 连接为 IP 地址
port	服务器端口
dbname	指定连接的数据库

续表

选　项	描　　述
user	用以连接的用户
password	如需要认证，则为密码
options	跟踪/调试选项
tty	发送调试信息的文件或 tty
requiressl	设为 1 要求 SSL 连接

如果没有定义 conninfo 字符串，可以通过设置下面的环境变量来指定连接选项：

PGDATABASE　设置数据库名。
PGSATESTYLE　设置缺省数据类型。
PGGEQO　设置缺省基因优化器。
PGHOST　设置数据库主机。
PGOPTIONS　设置不同的运行过程选项。
PGPASSWORD　设置用户密码。
PGPORT　设置服务器端口。
PGREALM　设置 Kerberos 域。
PGTTY　设置调试/错误 tty 或文件。
PGTZ　设置缺省时区。
PGUSER　设置用户名。

下面的函数取决于前面返回的连接指针（在这里是 PGconnectID）。

■ char *PQdb(const PGconnectID, *conn)

返回当前连接数据库的名称。

■ char *PQuser(const PGconnectID, *conn)

返回当前连接用户的名称。

■ char *PQpass(const PGconnectID, *conn)

返回当前连接的密码。

■ char *PQhost(const PGconnectID, *conn)

返回后端服务器的主机名。

■ char *PQport(const PGconnectID, *conn)

返回当前连接的服务器端口。

■ char *PQtty(const PGconnectID, *conn)

返回连接的调试文件/tty。

■ char *PQoptions(const PGconnectID, *conn)

返回调试/跟踪连接选项。

■ char *PQerrormessage(const PGconnectID, *conn)

返回连接产生的最后错误信息。

■ int *PQbackendPID(const PGconnectID, *conn)

返回后端服务器进程的 PID。

■ ConnStatusType PQstatus(const PGconnectID, *conn)

返回下面状态条件之一：

```
CONNECTION_STARTED
CONNECTION_MADE
CONNECTION_AWAITING_RESPONSE
CONNECTION_AUTH_OK
CONNECTION_SETENV
CONNECTION_OK
CONNECTION_BAD
```

PQconnectdb 还提供了如下函数：

■ void PQreset(PGconn *conn)

重置与后端的通信端口。

■ void PQfinish(PGconn *conn)

关闭与后端的连接并释放由 PGconn 对象使用的内存。

PQexec

Pqexec 函数将请求的查询发送到一个连接的后端服务器上。当收到反应时，则将之保存在一个指针内。在这里，指针叫 PGresult，不过，也可以与之不同。

```
PGresult *PQexec(PGconn *conn,const char *query)
```

下面函数作用于前面返回的指针（PGresult）：

■ ExecStatusType PQresultStatus(const PQresult *res)

提供关于最后执行查询的信息。返回以下值之一：

```
PGRES_EMPTY_QUERY
PGRES_COMMAND_OK
PGRES_TUPLES_OK
PGRES_COPY_OUT
PGRES_COPY_IN
PGRES_BAD_RESPINSE
PGRES_NONFATAL_ERROR
PGRES_FATAL_ERROR
```

■ char * PQresultMessage(const PQresult *res)

此函数返回与一个特定 PGresult 关联的最后错误信息。它与 PQerrormessage 不同，后者返回与特定连接关联的最后错误信息而不是针对指定结果。

■ int PQntuples(const PQresult *res)

返回查询结果的行数。

■ int PQnfield(const PQresult *res)

返回查询结果中每行中的字段数。

■ char *PQfname(const PQresult *res,int field_index)

返回与给定字段索引关联的字段名称。字段索引从 0 开始。

■ int PQfnumber(const PQresult *res,const char *field_name)

返回与指定字段名称关联的字段索引。如果给定名称不能与任何字段匹配则返回值为一1。

■ Oid PQftype(const PQresult *res, int field_index)

返回一个表示与字段索引关联的字段类型的整数。系统表 pg_type 中包含不同数据类型的名称和属性。内置数据类型 OID 在源代码树的 src/include/catalog/pg_type.h 中定义。

■ int PQfsize(const PQresult *res, int field_index)

返回指定字段索引中字段数据的字节数。

■ char* PQgetvalue(const PQresult *res,int tup_num,int field_num)

返回 PGresult 中一行的一个单独字段值。大部分情况下，PQgetvalue 的返回值是一个表示此值的以 NULL 结尾的 ASCII 字符串。

■ int PQgetlength(const PQresult *res, int tup_num,int field_num)

返回字段的字节长度。

■ int PQgetisnull(const PQresult *res, int tup_num,int field_num)

如果字段为 NULL，则返回 1；否则返回 0。

■ char * PQcmdTuples(const PQresult *res)

返回由 SQL 命令影响的行数。此函数仅估量 INSERT、DELETE 或 UPDATE 命令的影响。

■ Oid PQoidValue(const PQresult *res)

如果 SQL 命令是 INSERT，则返回插入元组的 OID；否则，返回 InvalidOid。

■ void PQclear(PQresult *res)

释放与 PGresult 有关的保存空间。每个查询结果当不再需要时都必须通过 PQclear 释放。使用此命令后 PGresult 不会消失，即使关闭了连接。操作失败将会导致前端应用的内存泄漏。

13.4 libpq++

libpq++库提供了 C++应用程序在 PostgreSQL 的后端接口。这基本上与 libpq 库的使用方式相同，不同的是，它的许多操作是通过类来实现的。

libpq++提供了两个主要类：PgConnection 和 PgDatabase。

PgConnection

这个类提供了连接到 PostgreSQL 数据库所需的函数。

PgConnection::PgConnection(const char *conninfo)

连接信息可以通过连接字符串参数（与前相同）或通过明确设置以下环境变量来指定：

PGDATABASE	设置数据库名。
PGSATESTYLE	设置缺省数据类型。
PGGEQO	设置缺省基因优化器。
PGHOST	设置数据库宿主。
PGOPTIONS	设置不同的运行过程选项。
PGPASSWORD	设置用户密码。
PGPORT	设置服务器端口。
PGREALM	设置 Kerberos 域。

PGTTY　　设置调试/错误 tty 或文件。

PGTZ　　设置缺省时区。

PGUSER　　设置用户名。

如果没有使用环境变量，则连接字符串参数可通过表 13-5 中选项来指定（形式为：选项=值）：

表 13-5　　PgConnection 接受的选项

选　项	描　述
Dbname	指定连接的数据库
Host	连接的主机名（或 UNIX 路径）
Hostaddr	对于 TCP/IP 连接为 IP 地址
Options	跟踪/调试选项
Password	如需要认证，则为密码
Port	服务器端口
Requiressl	设为 1 要求 SSL 连接
Tty	发送调试信息的文件或 tty
User	用以连接的用户

此类提供了若干用于数据库连接的函数：

■ int PgConnection::ConnectionBad()

如果连接成功则返回 TRUE；否则返回 FALSE。

■ int ConnStatusType PgConnection::Status()

返回与后端服务器的连接状态，或 CONNECTION_OK 或 CONNECTION_BAD。

■ ExecStatusType PgConnection::Exec(ocnst char * query)

发送查询到后端服务器并执行。返回查询的结果。状态报告为下面之一：

PGRES_EMPTY_QUERY
PGRES_COMMAND_OK
PGRES_TUPLES_OK
PGRES_COPY_OUT
PGRES_COPY_IN
PGRES_BAD_RESPINSE
PGRES_NONFATAL_ERROR
PGRES_FATAL_ERROR

■ const char * PgConnection::ErrorMessage()

返回最后一个错误信息文本。

PgDatabase

PgDatabase 类提供了访问返回数据集内元素的方法。特别是，此类对返回一个给定查询所影响的行数或字段数信息很有用。下面是类函数：

■ int PgDatabase::Tuples()

返回查询结果中的行数。

■ int PgDatabase::CmdTuples()

返回受 INSERT、UPFATE 或 DELETE 命令影响的行数。如果为其他命令，则返回－1。

■ int PgDatabase::Fields()

返回查询结果中的字段数。

■ const char * PgDatabase::FieldName(int field_num)

返回与给定索引关联的字段名称。字段索引以 0 开始。

■ int PgDatabase::FieldNum(const char * field_name)

返回与指定字段名称关联的字段索引。

■ Oid PgDatabase::fieldType(int field_num)

返回与指定索引关联的字段类型。返回的整数为对应类型的内部代码。

■ short PgDatabase::FieldSize(int field_num)

返回由给定字段占用的字节数。字段索引以 0 开始。

■ const char * PgDatabase::GetValue(int tup_num, int field_num)

从 PGresult 的一行返回一单独字段值。行和字段索引以 0 开始。对于大部分查询，GetValue 的返回值是一个以 NULL 结尾的 ASCII 字符串。

■ int PgDatabase::Getlength(int tup_num, int field_num)

返回字段的字节长度。

■ void PgDatabase::PrintTuples(FILE *out=0, int printAttName=1,int terseOutput=0,int width=0)

打印所有的元组和（或）其属性名。

■ nt PgDatabase::GetLine(char * string, int length)

从套接字直接读入一行。

■ void PgDatabase::PutLine(const char * string)

在连接套接字中直接写入一行。

■ int PgDatabase::EndCopy()

确保客户与服务器同步，以防直接访问方法导致通信异步。

13.5　libpqeasy

实质上，libpqeasy 库只是为 libpq C 库提供了一个较简单的接口。libpqeasy 的典型应用如下：

■ connectdb。连接到数据库。

■ doquery。执行给定的查询。

■ fetch。从后端取回结果。

■ disconnectdb。关闭数据库连接。

libpqeasy 提供的完成上列任务的函数有：

■ PGconn * connectdb(char * options);

■ PGresult * doquery(char * query);

■ int fetch(void * param);

■ int fetchwithnulls(void * param);

- void reset_fetch();
- void disconnectdb();

13.6 ODBC

开放数据库连接（ODBC）是一个 API，它提供了一个前端应用与数据库服务器间的产品无关性接口。ODBC 驱动主要用于连接 Windows 类应用与不同的 RDBMS。不过，ODBC 驱动几乎在所有平台下均可使用，包括 UNIX、Mac 及其他平台。

安装

可以在编译 PostgreSQL（或从包安装）时添加所需的 ODBC 驱动。虽然 PostgreSQL 内置了一些 ODBC 驱动，但其他方案更受广泛支持。其中一个更流行的 ODBC 访问方法是目前的 unixODBC 方案（可参见 www.unixodbc.org）。

安装 ODBC 驱动可分为以下 5 步：

（1）安装和配置一个 ODBC 管理器。

（2）编译指定的 PostgreSQL ODBC 驱动。

（3）在基本目录中添加 ODBC 扩展。

（4）在客户机上安装 PostgreSQL ODBC 驱动。

（5）配置.ini 文件或使用给定的 GUI。

在实际安装所选 ODBC 驱动开始前，必须确保系统中已经存在一个 ODBC 管理器。从 Windows 95 开始的所有 Windows 版本都包含了 ODBC 管理器。对于 UNIX/Linux 客户，有几种选择余地。有 unixODBC 管理器小程序，有免费的 ODBC 客户端 i ODBC。（请从 www.unixodbc.org或www.iodbc.org网站获取更多信息。）

如果你的系统是从源代码安装的，在编译时就应该选择了选项--enable-odbc（请参见第 10 章“常规管理任务”了解更多的编译选项知识）。同样，大部分基于包安装方式也提供了一个包含所需 ODBC 功能的可选包（例如 postgresql-odbc-7.1.2-4PGDG.i386.rpm）。

另外一种情况是，如果系统以前没有带 ODBC 选项进行编译，你仍然可以通过在适当目录（如 src/interface/odbc）中运行命令 make install 来编译它。

为了让 ODBC 完全兼容还要对基本目录作一些修改。文件 odbc.sql 列出了对基本目录结构所需作出的修改。它作为一个脚本执行而且需要人工干预。要自动进行这些修改，以 PostgreSQL DBA 用户身份执行如下命令：

```
>psql -d template1 -f PATH/odbc.sql
```

另外，请确保以-i 选项启动一个 postmaster（或对文件 postgresql.conf 作适当修改），它可以允许从 TCP/IP 连接的访问方式。而且大部分系统要求对文件 pg_hha.conf 进行编辑。（请参见第 10 章了解 PostgreSQL 管理的更详细信息。）否则，客户想要成功连接则需要定位。

对于客户机的安装，最简单的办法就是下载 Windows 可执行文件来自动安装和配置你的 Windows 系统。安装程序可以从下面网站获取（也可从其镜像站点获取）：

ftp://ftp.postgesql.org/pub/odbc/versions/full/

而且 MS 安装程序（MSI）或纯 DLL 版本驱动程序还可从下面地址获得：

ftp://ftp.postgresql.org/pub/odbc/versions/msi/

ftp://ftp.postgresql.org/pub/odbc/versions/dll/

下一步就是配置文件 odbc.ini（或如果愿意，也可使用提供的 GUI 管理工具）。

odbc.ini 有 3 个必需部分：

■ [ODBC Data Source]

数据库名称列表。

此部分必须包含如下内容：

```
Driver=path(如:prefix/lib/libpsqlodbc.so)
Database=DatabaseName
Servername=localhost
Port=5432
```

■ [Data Source Specification]

对每个 ODBC 数据源的配置部分。

■ [ODBC]

定义 InstallDir 关键字。

另外一种指定.ini 文件内所有选项的方法是运用 Windows 驱动程序提供的 GUI 配置工具。

此工具提供的选项有：

■ **禁止使用基因优化器**（Disable Genetic Optimizer）在连接时自动关闭后端基因优化器。

■ **键集查询优化**（Keyset Query Optimization，KSQO）一些应用程序，特别是 MS Jet 数据库引擎，使用了“键集”查询。许多查询如果没有 KSQO 很可能在后端崩溃。

■ **日志通信**（Commlog）日志文件与后端之间的通信。

■ **识别唯一索引**（Recognize Unique Index）这一设置允许 Access 95 和 97 在连接询问用户是什么索引。

■ **只读**（ReadOnly）（**缺省**）新数据源将继承旧数据源的只读属性。

■ **使用申请/回读**（Use Declare/Fetch）如果为真（缺省值），驱动程序自动使用游标申明/回读来处理 SELECT 语句并在缓存中保存 100 行。

■ **解析语句**（Parse Statement）如果允许，驱动程序将解析一个 SQL 查询语句来识别列和表并搜集相关统计信息，如准确性、无用性、别名等等。

■ **未知大小**（Unknown Size）它用于控制对于字符数据类型 SQLDescribeCol 和 SQLColAttributes 返回的精度。它主要针对于以前的 PostgreSQL 6.4 版本。其选项如下：

最大值（Maximum）允许返回数据类型的最大精度。

未知（Don't Know）返回一个“未知”值并由应用程序自己决定。

最大（Longest）返回任一行的列的最长字符串长度。

■ **数据类型选项**（Data Type Option）它将影响数据类型映射的方式。选项如下：

文本类型 LongVarCharPostgres 的 TEXT 类型映射成 LongVarChar；否则就映射成 SQLVarChar。

未知类型 LongVarChar 未知类型（数组等）被映射成 LongVarChar；否则就映射成 SQLVarChar。

布尔类型 Char 布尔返回值映射成 SQL_CHAR；否则，就映射成 SQL_BIT。

- **缓存大小**（Cache Size）使用游标时，它指元组缓存的行大小。如果没有使用游标，它则指在任意指定时间分配多少元组缓存。
- **VarChar 最大精度**（Max VarChar）VarChar 和 BPChar（char[x]）类型的最大精度。
- **LongVarChar 最大精度**（Max LongVarChar）LongVarChar 类型的最大精度。
- **系统表前缀**（SysTable Prefixes）缺省时，名称以 pg_开始者被视作系统表。也允许定义其他的前缀。用分号（；）分开每个前缀。
- **连接设置**（Connect Setting）连接时这些命令被发送到后端。使用分号（；）来分隔各命令。

驱动程序还提供了如下数据源/连接选项：

- **只读**（ReadyOnly）决定数据源是否允许修改。
- **行版本**（Row Versioning）当你试图修改一个行时，此选项允许应用侦查数据是否已被其他用户修改。驱动程序使用 Postgres 的 xmin 系统字段来允许行版本侦查。
- **显示系统表**（Show System Tables）驱动程序把系统表当作普通表看待。
- **OID 选项**（OID Options）：

  显示列（Show Column）显示 OID。

  伪索引（Fake Index）伪造一个 OID 的特定索引。这主要用于旧的 MS 应用访问方式。
- **协议**（Protocol）：

6.2 强制驱动使用 Postgres 6.2 协议，它具有不同的位元排序、协议和其他定义。

6.3 使用 6.3 协议。它与 6.3 及 6.4 后端兼容。

6.4 使用 6.4 协议。它仅与 6.4 兼容。

13.7 Perl

PostgreSQL 中已经包含了能够运行 Perl 脚本的过程语言 PL/Perl。不过，从外部 Perl 脚本访问 PostgreSQL 需要使用 Perl 独立数据库（DBI）模块。Perl DBI 定义了一套函数、变量和 Perl 脚本约定，而不管实际所使用的后端数据库是什么。

Perl DBI 提供了一个脚本兼容性接口，因而其代码兼容性更好、灵活性更强。DBI 只是一个一般用途的接口，不过，要连接到指定的数据库仍然需要数据库驱动。

Perl 系统并不包含名为 pg 的旧版本，非 DBI PostgreSQL 访问模块。不过，这是一个旧版本模块，现在的开发工作开始使用较新的 DBI 兼容模拟了。

Perl DBI 系统的整体结构如图 13-2 所示。

PostgreSQL 驱动名为 DBI::Pg，而且要成功执行必须已经存在且正确安装。此类和驱动集紧随 libpg 库函数之后开发。所以，定义它们在 C 程序中的工作方式与函数接口相似。

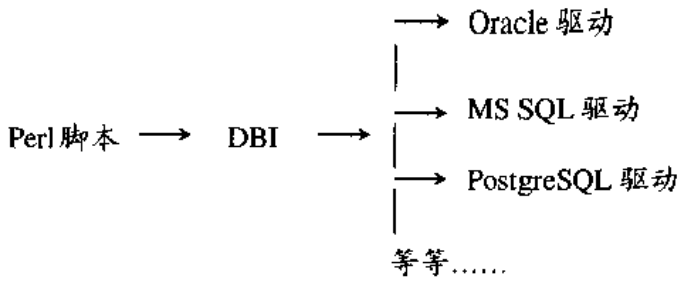


图 13-2　Perl DBI 系统的整体结构图

DBI 类（连接）

DBI 是由接口系统提供的基本类。它提供的方法如下：

■ connect($data_source,$username,$password,\%attr);

选项：

data_source：数据库驱动名（如 dbi:pg）。

username：以此用户身份连接。

password：用户密码。

\%attr：指定此驱动的不同选项。

描述：建立一个数据库连接。如果成功，返回一个有效数据库句柄对象。

■ available_drivers

描述：返回有效数据库驱动列表。

■ data_source($driver)

描述：返回对指定驱动有效的数据库列表。

■ trace($level[,$file])

描述：指定跟踪/调试级别和一个日志文件（如果指定的话）。

DBI 句柄方法（运行查询）

一旦 DBI 类返回一个有效句柄对象，它将提供以下方法：

■ prepare($statement[,\%attr])

描述：此函数连同相关选项一起将查询语句发送到数据库引擎用于准备工作。（此方法并不进行 PostgreSQL 中的准备；这只是简单地将查询放入缓存等待调用。）

■ do($statement[,\%attr][,@bind_values])

描述：准备和执行给定的查询语句。另外，可以指定可选属性及结果的捆绑位置。

■ commit

描述：在数据库后端运行一个 COMMIT 命令。

■ rollback

描述：在数据库后端运行一个 ROLLBACK 命令。

■ disconnect

描述：断开与数据库的连接。

■ ping

描述：决定数据库服务器是否仍然运行。

■ quote($string)

描述：转义任何特殊字符。用于在提交给后端前格式化一个查询字符串。

DBI 语句句柄方法（结果）

返回一个 ResultSet 后，此对象可提供如下方法：

■ execute([@bind_values])

描述：执行先前准备的语句。或者，在执行语句前为每个元素捆绑值。

■ fetchrow_arrayref

描述：读取下一行的数据值；返回一个此数组的引用。

■ fetchrow_array

描述：读取下一行的数据值；返回一个此数组的数组。

■ fetchrow_hashref

描述：读取下一行的数据值；返回一个某个数组的引用。

■ fetchall_arrayref

描述：读取所有行的数据值；返回一个某个数组的引用。

■ finish

描述：指示后端无更多的行要读取；允许服务器回收资源。

■ rows

描述：返回最后一个查询所影响的行数。

■ bind_col$column_number,\$var_to_bind,\%attr);

描述：捆绑指定 PostgreSQL 列和 Perl 变量。

语句句柄属性

返回的语句句柄提供了如下属性：

■ NUM_OF_FILES

描述：返回当前行的字段数。

■ NUM_OF_PARAMS

描述：返回准备语句中的占位符数。

■ NAME

描述：对每列返回一个包含字段名称的数组引用。

■ pg_size

描述：对每列返回一个整数值数组的引用。此整数显示了列的字节大小。不同长度的列由-1 来表示。

■ pg_type

描述：对每列返回一个字符串数组的引用。此字符串显示了数据类型的名称。

■ pg_oid_status

描述：PostgreSQL 指定属性，它返回最后插入（INSERT）命令的 OID。

■ pg_cmd_status

描述：PostgreSQL 指定属性，它返回最后命令的类型。可能类型为：INSERT、DELETE、

UPDATE 及 SELECT。

13.8 Python（PyGreSQL）

PyGreSQL 是一个对 PostgreSQL 数据库的 Python 接口。它由 D'Arcy J.M. Cain 编写并在很大程度上基于 Pascal Andre 的代码而完成。

PyGreSQL 通过 3 个部分来实现：

C 共享模块：_pg.so

两个 Python 封装器：pg.py 和 pgdb.py

编译 PyGreSQL

在包含 pgmodule.c 的目录中运行如下命令：

```
cc -fpic -shared -o _pg.so -I[pyInc] -I[pgInc] -L[pgLib] -lpq pgmodule.c
```

编译选项有：

[pyInc]=Python 包含路径（Python.h）。

[pgInc]=PostgreSQL 包含路径（postgres.h）。

[pgLib]=PostgreSQL 库路径（lipq.so/libpq.a）。

编译时可以指定一些关键字：

-DNO_DEF_VAR 禁止支持缺省变量。

-DNO_DIRECT 禁止直接访问方法。

-DNO_LARGE 禁止大对象支持。

-DNO_PQSOCKET 旧 PostgreSQL 版本。

-DNO_SNPRINT 不能调用 snprintf。

Python 配置

定位 Python 的动态装载包位置（如 usr/lib/ python/lib-dynload）。将结果文件_pg.so 复制到此位置。

复制 pg.py 和 pgdc.py 文件到 Python 的标准库目录（如/usr/local/lib/Python）。

pg.py 文件使用了传统的接口，而 pgdb.py 文件与由 Python DB-SIG 开发的 DB-API 2.0 定义兼容。

下面内容只介绍旧版本的 pgAPI。你可以按下面网址阅读新版本的 DB-SIG API。

www.python.org/topics/database/DatabaseAPI-2.0.html

或参阅以下网址的指南：

www2.linuxjournal.com/lj-issues/issue49/2605.html

PyGreSQL 接口

PyGreSQL 对 PostgreSQL 数据库服务器提供了两个单独的接口。可以通过以下两个封装模块之一来访问：

■ pg。标准 PyGreSQL 接口。

■ pgdb。DB 2.0 API 接口。

虽然新的开发努力进一步定义了 DBI 兼容接口，但目前标准 PyGreSQL 接口更规范。本节集中讲解标准接口，可能也会涉及到 DBI 2.0 接口的部分信息：

www.python.org/topics/database/DatabaseAPI-2.0.html

标准 pg 模块提供了以下属性：

■ connect(dbname,host,port,opt,tty,user,password)

描述：打开一个 PostgreSQL 连接。

参数：

dbname	连接的数据库名。
host	服务器主机名。
port	数据库服务器所用端口。
opt	连接选项。
tty	调试终端。
user	PostgreSQL 用户。
password	用户密码。

例如：

```
>>>import pg
>>>database=pg.connect(dbname="newriders",
    host=127.0.0.1)
```

■ get_defport()

描述：返回当前缺省主机信息。

■ get_defport()

描述：返回当前缺省端口信息。

■ get_defopt()

描述：返回当前缺省连接选项。

■ get_deftty()

描述：返回当前缺省调试终端。

■ set_deftty(tty)

参数：

tty　新调试终端

描述：设置新连接的调试终端值。如果不提供任何参数（NONE），则在将来连接中使用环境变量。

■ get_defbase(0

描述：返回当前数据库名称。

1. 发送查询到数据库对象

一旦与数据库连接后，返回一个 pgobject。该对象嵌入了定义此连接的指定参数。函数调用时的可用参数如下：

■ query(command)

参数：

command　　SQL命令字符串

描述：发送指定 SQL 查询（命令）到数据库。如果查询是一个插入语句，则返回值是新行的 OID。如果是一个查询，则不返回结果，即返回 NONE。对于 SELECT 语句，返回一个 pgqueryobject 对象，通过 getresult 或 dictresult 方法可访问它。

例如：

```
>>>import pg
>>>database=pg.connect("newriders")
>>>result=database.query("SELECT * FROM authors")
```

■ close

描述：关闭数据库连接。当删除连接时，连接自动关闭，此方法允许运行一个显式关闭命令。

■ fileno

描述：返回处理套接字 ID，用于与数据库连接。

■ getnotify

描述：从服务器接收 NOTIFY 信息。如果没有返回任何通知，此方法返回 None。否则，它返回一个元组（relname,pid），其中的 relname 指通知的名称，pid 指触发此通知连接的进程 ID。记住，首先作一个监听查询；否则，getbotify 将总是返回 None。

■ inserttable

描述：允许在表中快速插入大数据库块。此列表列出了定义每个插入行的值的元组/列表。

■ putline

描述：向连接套接字中直接写入一个字符串。

■ getline

描述：此方法直接从服务器套接字中读入一个字符串。

■ endcopy

描述：确保客户与服务器同步，防止直接访问方法导致通信异步。

2. 从数据库连接中访问大对象

通过数据库的 pg 连接访问大对象，可使用下面函数：

■ getlo

描述：通过对象的 OID 得到一个大对象。

■ locreate

描述：在数据库中创建一个大对象。

■ loimport

描述：此方法允许你以非常简单的方式创建大对象。你只需给定包含使用数据的文件名。

■ open

描述：此方法打开一个大对象用于读/写，与 UNIX open()函数方式相同。

■ close

描述：此方法关闭先前打开的大对象，与 UNIX close()函数方式相同。

■ read

描述：此函数允许你从当前位置开始读一个大对象。

■ write

描述：此函数允许你从当前位置开始写一个大对象。

■ tell

描述：此方法用于获取大对象的当前位置。

■ seek

描述：大对象中移动位置游标。

■ unlink

描述：删除一个 PostgreSQL 大对象。

■ size

描述：返回大对象的大小。当前，大对象需要已经打开。

■ export

描述：倾倒大对象内容到运行 Python 程序的主机上，并非服务器主机。

3. 从 pgobject 访问结果

一旦查询运行了数据库查询命令，如果返回了结果，就可以通过以下途径来访问它：

■ getresult

描述：返回包含 pgqueryobject 的值列表。对于它的更详细信息可以通过 listfields、fieldname 或 fieldnum 方法来访问。

■ dictresult

描述：返回包含 pgqueryobject 的值列表，用作键索引的字段名称字典。

■ iistfields

描述：列出先前查询结果的字段名称。字段与结果值排序相同。

■ fieldname(int)

描述：从顺序序列号（整数）中找到一个字段名称。字段与结果值排序相同。

■ fieldnum(str)

描述：从名称（字符串）中返回字段号。

■ ntuples

描述：返回查询中发现的元组数。

■ reset

描述：重置当前数据库。

例如：

```
>>>import pg
>>>database=pg.connect("newriders")
>>>results=database.query("SELECT * FROM payroll")
>>>resultsmtuples2340
>>>mydict=results.dictresult()
```

4. DB 封装器

前面的函数是通过 pg 模块封装的。这一模块还提供了一个特殊的封装器，这就是 DB。

此封装器改进了许多与数据库交互的连接和访问机制。前面的函数同时包含在其中，所以不需导入两个模块。使用这一模块的最好方法是：

```
>>>import pg
>>>db=pg.DB('payroll',localhost')
>>>db.query("INSERT INTO checks VALUES ('Erica',200)")
>>>db.query("SELECT * FROM checks")

Name      Amount
--------------
Erica 200
```

下面列表描述了此类的方法和变量（它们与基本 pg 方法很相似，只有一些细微的差别）：

■ pkey(table)

描述：此方法返回表的主键。注意如果表没有主键则会导致意外。

■ get_databases

描述：虽然只需单独地选择即可完成此任务，但把它添加在这里是为了使用的方便。

■ get_tables

描述：返回当前数据库中可用的表列表。

■ get_attnames

描述：返回属性名称列表。

■ get(table,arg,[keyname[]

参数：

table　　表名称。

arg　　字典或查找值。

keyname　　用作键的字段名（可选）。

描述：得到一个单独行。假设此键指定了一个特定行；如果没有指定 keyname，则使用表的主键。

■ insert(table,a)

参数：

table　　表名称。

a　　值字典。

描述：在指定表中插入值，使用字典中的值。然后，通过被规则、触发器等修改的值来更新字典。

■ update(table,a)

参数：

table　　表名称。

a　　值字典。

描述：修改现有行。修改基于从 get 得到的 OID 来进行。返回一个数组，反映由于触发器、规则、缺省值的修改而导致的变化。

■ clear(table,[a])

参数：

table　　表名称。

a　　值字典。

描述：清除所有字段为 clear 值，由数据类型决定。数字类型设置为 0，日期类型设置为 TODAY，其他设置为 NULL。如果存在参数 a，则它用作数组，并且与属性名匹配的所有实体都将用未变化部分清除。

■ dclete(table,a)

参数：

table　　表名称。

a　　值字典。

描述：基于从 get 中获得的 OID，从表中删除行。

13.9 PHP

PHP 是一个用于构建动态 web 页的脚本语言。它包含许多其他商业软件如 ASP 和 ColdFussion 的高级功能。

它包含若干内置数据库接口，包括用于与 MySQL 和 PostgreSQL 通信的函数。下面是针对 PostgreSQL 的函数列表：

■ pg_close(*connection_id*)

描述：关闭一个 PostgreSQL 连接。如果不是一个有效连接则返回一个 false；否则返回 true。

■ pg_cmdtuples(*result_id*)

描述：返回受 INSERT、UPDATE 或 DELETE 影响的事例数量。如果无元组受影响，则函数返回 0。

■ pg_connect([*host*],[*port*],[*options*],[*tty*],dbname)

描述：打开一个与 PostgreSQL 数据库的连接。连接成功则返回连接索引，若连接失败则返回 false。

例：

```
<?php
$dbconn=pg_Connect("dbname=newriders");
$dbconn2=pg_Connect("host=localhost port=5432 dbname=newriders");
?>
```

■ pg_dbname(*connection_id*)

描述：返回指定连接索引对应的数据库名。否则，如果连接非有效连接索引则返回 false.

■ pg_end_copy([*resourse connection*])

描述：执行复制操作后，使前端应用与后端同步。此命令必须运行；否则，后端可能与前端异步。

■ pg_errormessage(*connection_id*)

描述：从前一个数据库操作中返回一个包含错误信息的字符串；否则，返回 false。

■ pg_exec(*connection_id*, *query*)

描述：执行查询中的SQL命令后返回一个结果索引。否则，返回一个false值。对于一个成功的命令执行，此函数的返回值是一个索引，它用于从其他PostgreSQL函数访问此结果。

■ pg_fetch_array(*result_id,row,[result_type]*)

描述：返回一个与读取行对应的数组；否则，如果无更多行则返回false。

pg_fetch_array()是pg_fetch_row()的扩展版本。除了在结果数组的数字索引中保存数据外，它还使用字段名作为键来保存关联索引中的数据。

例如：

```
<?php
$conn=pg_pconnect("dbname=newriders");

$rst=pg_exec($conn,"SELECT * FROM authors");
if (!$rst){
      echo "An error occurred.\n;
      exit;
}
$rst_array=pg_fetch_array($rst,0);
echo $rst_array[0]. "First Row -First Field\n";

$rst_array=pg_fetch_array($rst,1);
echo $rst_array["author"]. "Second Row -Author Field\n";
?>
```

■ pg_fetch_object(*result_id,row, [result_type]*)

描述：返回与读取行对应的属性；否则，如果无更多行则返回false。

pg_fetch_object()与pg_fetch_array()相似，只有一个区别——返回的是对象而不是数组。所以，你只能通过字段名访问数据，而不是通过他们的序号。

■ pg_fetch_row(*result_id,row*)

描述：作为数组返回指定行。每个结果列保存在一个数组偏移中，偏移从0开始。

■ pg_fieldisnull(*result_id,field*)

描述：如果给定行中的字段非空NULL返回0。如果给定行中的字段为NULL则返回1。字段可以指定为数字或字段名。

■ pg_fieldname(*result_id,field_number*)

描述：返回与指定字段索引号对应的字段名。字段号从0开始。

■ pg_fieldnum(*result_id,field_name*)

描述：返回指定列名的字段号。

■ pg_fieldprtlen(*result_id,row, field_name*)

描述：返回给定行中指定字段的字符数。

■ pg_ fieldsize(*result_id, field_number*)

描述：返回给定字段号所占用保存大小的字节数。字段大小为-1表示一个不同长度的字段。

■ pg_ fieldtype(*result_id, field_number*)

描述：返回一个包含由给定字段号表示的字段数据类型的字符串。

■ pg_freeresult(*result_id*)

描述：调用时，自动释放所有的结果内存。一般，只有你确定内存匮乏时才使用它，因为一旦关闭连接 PHP 会自动释放内存。

■ pg_getlastoid(*result_id*)

描述：返回最后分配给插入元组的 OID。此标识符在 pg_exec()发送的最后命令中使用。

■ pg_host(*connection_id*)

描述：返回连接 PostgreSQL 服务器的主机名。

■ pg_loclose(*file_id*)

描述：关闭一个大对象。file_id 是一个从 pg_loopen()对大对象的文件描述符。

■ pg_locreate (*connection_id*)

描述：创建一个大对象并返回其 ID。

■ pg_loexport(*oid,file_path*[,*int connection_id*])

描述：指定导出的大对象的 ID，而且文件名参数指定文件的路径名。

■ pg_loimport(*file_path*,[*connection_id*])

描述：指定作为一个大对象导入的文件路径名。在 PostgreSQL 中对大对象的所有操作都必须发生在一个事务内。

■ pg_loopen (*connection_id,obj_oid,string mode*)

描述：打开一个大对象并返回文件描述符。文件描述符封装了有关连接的信息。不要在关闭大对象文件描述符之前关闭此连接。obj_oid 指定了一个有效的大对象 OID。方式可为："r"、"w"、或"rw"。

■ pg_loread(*file_id,length*)

描述：从大对象中阅读指定长度并以字符串类型返回值。file_id 指定了一个有效的大对象描述符。

■ pg_loreadall(*field_id*)

描述：读一个大对象并直接传递给浏览器。

■ pg_lounlink (*connection_id,large_obj_id*)

描述：根据 large_obj_id 中指定的 OID 删除一个大对象。

■ pg_lowrite(*file_id,buffer*)

描述：从指定缓存中写入大对象。返回实际写入的字节数，或者当 file_id 从 pg_loopen()中指向大对象文件描述符时，返回 false。

■ pg_numfields(*result_id*)

描述：返回结果中的字段数。result_id 是一个由 pg_exec()返回的有效结果标识符。

■ pg_numrows(*result_id*)

描述：返回结果中的行数。result_id 是一个由 pg_exec()返回的有效结果标识符。

■ pg_options (*connection_id*)

描述：返回在给定连接标识符上有效的指定选项字符串。

■pg_connect([*host*],[*port*],[*tty*],[*options*],*dbname*,[*user*],[*passw*

ord])

描述：打开一个其他 PHP 函数所需的与 PostgreSQL 数据库的稳固连接。

■ pg_port(*connection_id*)

描述：返回 PostgreSQL 服务器的端口号。

■ pg_put_line (*connection_id,data*)

描述：给 PostgreSQL 服务器发送一个以 NULL 结尾的字符串。例如，它可用于通过启动一个 PostgreSQL 复制操作非常高速地在表中插入数据。

例如：

```
<?php
$conn=pg_pconnect ("dbname=foo");
pg_exec($conn,"create table bar (a int4,b char(16), d float8)");
pg_exec($conn,"copy bar from stdin");
pg_put_line($conn,"3\thello world\t4.5\n");
pg_put_line($conn,"4\tgoodbye world\t7.11\n");
pg_put_line($conn,"\\.\n:);
pg_end_copy($conn);
?>
```

■ pg_result(*result_id,row_number,fieldname*)

描述：从 pg_exec 生成的结果标识符中返回值。row_number 和 fieldname 定义了返回的数组元素。不必使用字段名，你可以使用字段索引作为一个参数结束数字。

■ pg_set_client_encoding (*connection_id,encoding*)

描述：设置客户编码类型。编码可以为：SQL_ASCII、EUC_JP、EUC_CN、EUC_KR、EUC_TW、UNICODE、MULE_INTERNAL、LATIN1…LATIN9、KOI8、WIN、ALT、SJIS、BIG5 或 WIN1250。若成功返回 0，若发生错误则返回-1。

■ pg_client_encoding (*connection_id*)

描述：以字符类型返回客户编码。可以是 pg_set_client-encoding 函数的设置值之一。

■ pg_trace(*filename*,[*mode*,[*connection_id*]])

描述：允许跟踪 PostgreSQL 前端/后端间的通信并写入一个调试文件。对于帮助调试通信问题有用。

■ pg_tty(*connection_id*)

描述：返回发送服务器端调试输出的 tty 名称。

■ pg_untrace(*connection_id*)

描述：终止由 pg_trace 启动的跟踪。

第 14 章　高级 PostgreSQL 编程

与许多商业 RDBMS 相比，PostgreSQL 的真正优势之一在于它可以进行常规扩展。例如，PostgreSQL 本身并没有针对 Dewey-decimail 对象的数据类型，但是，如果你正创建一个数据库，此数据库即将作为一个使用了 Dewey Decimal 系统的大数据库后端系统，那么这一数据类型将很有用。

在数据库中插入新功能和对象就是所谓的“扩展”。PostgreSQL 主要通过用户编写新的基于 C 程序的对象和使用结果函数作为指定数据类型、操作符或所需聚集的句柄来实现扩展。

扩展 PostgreSQL 包括以下几个基本步骤：

1. 创建一个基于 C 的共享对象用来执行期望的功能。

2. 在 PostgreSQL 后端通过 CREATE FUNCTION 命令来注册此函数。

3. 连接适当的 SQL 命令（例如：CREATE TYPE、CREATE OPERATOR 等等）和此注册对象。

要理解实现扩展的工作方式，首先要对 PostgreSQL 的目录系统有一个大概了解。系统目录实质上就是特殊的表。不过，它不是用于保存用户数据，这些表用于保存与操作符、函数、数据类型、聚集、规则和触发器的定义方式有关的信息。所以，使用这些已有的机制来修改表，就可以达到扩展 PostgreSQL 系统本身的目的。

这些表中所保存的信息种类之一就是编译过的处理指定数据库函数的共享对象指针。实质上，CREATE FUNCTION、CREATE OPERATOR、CREATE TYPE 和 CREATE AGGREGATE 命令通过修改这些系统目录来包含额外功能定义的。

系统目录的进一步分类见表 14-1。

表 14-1　　系统目录

表	描　　述
pg_aggregate	聚集及聚集函数
pg_am	访问方法
pg_amop	访问方法操作符
pg_amproc	访问方法支持函数
pg_attribute	表列
pg_class	表
pg_database	数据库
pg_index	次索引
pg_opcclass	访问方法操作符类

续表

表	描　　述
pg_operator	操作符
pg_proc	过程（C 及 SQL）
pg_type	类型（基本及复杂的）

14.1　扩展函数

大部分扩展行为需要定义特殊函数。例如，要定义一个新数据类型，必须首先创建一个描述此新数据类型的 C 共享函数。主要有 3 类自定义函数（也可以参阅第 12 章“创建自定义函数”了解创建 SQL 或 PL 函数的更多相关讨论）：

- **SQL 函数**　这些函数包含纯标准 SQL 代码。执行它不需要必须存在外部数据库对象。它们可以自由定义而不管基本系统的配置如何。
- **PL 函数**　这些函数用非自身代码编写（例如用 PL/pgTCL 编写）。为了让它们执行，必须存在一个外部共享对象句柄。在执行前，句柄函数必须首先在数据库后端注册。
- **编译函数**　这些函数是典型的经过 C 语言编译过的函数。一般而言，这些函数定义了一个指定的输入——输出响应。它们可从标准 SQL 代码中调用（如 upper()函数）。但在激活它们前，它们必须首先定义为一个 C 共享对象然后在数据库中注册。

SQL 函数

SQL 语言函数只是简单的指定了名称的预定义查询。不过，他们却支持输入类型以及提供返回值。编写 SQL 函数不需要修改基本系统或特殊功能。例如：

```
CREATE FUNCTION getpay(int4) RETURN float8 AS
'SELECT sum(amount) FROM payroll
     WHERE employee_id=$1;
'LANGUAGE 'sql';
```

标准 SQL 函数还能处理相互间传递的类。例如：

```
CREATE FUNCTION getshortname(payroll) RETURN varchar AS
'SELECT left($1.last_name,4) AS S_NAME;
'LANGUAGE 'sql';

SELECT last_name FROM payroll WHERE emp_id=12345;

last name
--------------
Parody

SELECT getshortname(payroll) WHERE emp_id=12345;

last_name
```

```
--------------
Paro
```

过程语言函数

通过可装载模块来提供过程语言函数。例如，PL/pgSQL 语言就取决于可装载模块 plqsql.so。创建共享对象后，它们就通过 CREATE LANGUAGE 命令定义为句柄。

创建一个有效句柄对象的所需步骤超出了本书的讨论范围，下面只提供一些基本或一般性的步骤：

1. 在 C 中编译共享对象句柄（如 plfoobar.so）。
2. 创建定义此对象的函数。此函数的返回类型必须设置为 OPAQUE。例如：

```
CREATE FUNCTION plfoobar_handler() RETURN OPAQUE AS
'/usr/local/pgsql/lib/plfoobar.so' LANGUAGE 'C';
```

3. 定义一个句柄，将此对象的语言请求发送给先前创建的函数。例如：

```
CREATE TRUSTED PROCEDURAL LANGUAGE 'PLFOOBAR'
  HANDLER plfoobar_handler
  LANCOMPILER 'PL/FooBar';
```

定义一个语言后，就可以通过它创建函数和存储过程了。目前版本中，PostgreSQL 支持 PL/pgSQL、PL/Tcl 及 PL/Perl。请参考第 11 章“服务器端编程”了解创建过程语言函数的更多信息。

已编译函数

已编译函数是通过 CREATE FUNCTION 命令在数据库中已注册的共享对象。创建自定义已编译函数比创建脚本函数更为复杂，但它们对提高执行速度大有好处。

创建成功的 C 函数需要正确交换 PostgreSQL 和 C 数据类型。表 14-2 中列出了 PostgreSQL 数据类型、对应的 C 数据类型及所定义的 C 头文件。

表 14-2 系统目录

PostgreSQL 数据类型	C 数据类型	C 头文件
abstime	AbsoluteTime	utils/nabstime.h
bool	bool	include/c.h
box	(BOX *)	utils/geo.decls.h
bytea	(bytea *)	include/postgres.h
char	char	N/A
cid	CID	include/postgres.h
datetime	(DateTime *)	include/c.h 或 include/postgres.h
float4	(float4 *)	include/c.h 或 include/postgres.h
float8	(float8 *)	include/c.h 或 include/postgres.h
int2	int2 或 int6	include/postgres.h
int2vector	(int2vector *)	include/postgres.h
int4	int4 或 int32	include/postgres.h
lseg	(LSEG *)	include/geo-decls.h
name	(Name)	include/postgres.h

续表

PostgreSQL 数据类型	C 数据类型	C 头文件
oid	oid	include/postgres.h
oidvector	(oidvector *)	include/postgres.h
path	(PATH *)	utils/geo-decls.h
point	(POINT *)	utils/geo-decls.h
regproc	regproc 或 REGPROC	include/postgres.h
reltime	RelativeTime	utils/nabstime.h
text	(text *)	include/postgres.h
tid	ItemPointer	storage/itemptr.h
timespan	(TimeSpan *)	include/c.h 或 include/postgres.h
tinterval	TimeInterval	utils/nabstime.h
xid	(XID *)	include/postgres.h

数据在内部按以下 3 种方法之一传递给已编译函数：

- 按值传递。
- 按引用传递（固定长度）。
- 按引用传递（可变长度）。

一般来讲，按值传递的数据必须为 1、2 或 4 个字节长度（虽然某些结构能同时支持 8 个字节）。固定长度或可变长度调用适用于任意大小的数据类型。

1. 调用基于 C 的函数

与基于 C 的函数的接口有两个单独的约定：

- 版本 0 这一方法是旧方法，它现在逐渐被抛弃。虽然此法简单易用，但使用此方法的函数在各种架构中移植时会遇到兼容性的问题。
- 版本 1 这是较新的接口标准。它克服了版本 0 调用中的许多不足。它依赖于宏封装传递参数来实现，所以，生成的代码兼容性更好。

因为版本 0 调用方法已被淘汰，所以下面演示一些版本 1 函数的简单例子。（对于版本 0 调用的更多信息，请参考网站www.postgresql.org中的《PostgreSQL 程序员向导》部分内容）

版本 1 兼容性函数必须以两个宏开始：PG_FUNCTION_INFO_V1 和 PG_FUNCTION_ARGS。下面是一个按值传递的简单例子：

```
/*    Program Name: add_it.c
      Description:adds two int32 numbers */

#include "postgres.h"
#include "fmgr.h"

PG_DUNCTION_INFO_V1(add_it);

Datum add_it(PG_FUNCTION_ARGS)
{
```

```
  int32 arg1=PG_GETARG_INT32(0);
  int32 arg2=PG_GETARG_INT32(1);
  PG_RETURN_INT32(arg1+arg2);
}
```

以上代码编译成一个共享对象后，它可以定义和运用如下：

```
CREATE FUNCTION add_it(int4) RETURN int4 AS
      '/usr/local/pgsql/lib/add_it.so' LANGUAGE 'C';
>SELECT add_it(4,8) AS Answer;

Answer
------
12
```

除了处理简单的按值传递外，复合对象，像行对象也能通过 C 函数来传递和操纵。例如，下面例子定义了函数 isminor()，它根据职员是否越过 21 岁来返回值 TRUE 或 FALSE：

```
/* Program Name islegal.c
   Description:Determines if an employee is legal age */

#include "postgres.h"
#include "executor/exector.h"
#include "fmgr.h"

PG_FUNCTION_INFO_V1(islegal);

Datum
islegal(PG_FUNCTION_ARGS)
{
      /*Get the current table row, assign to pointer t*/
      TupleTableSlot *t=(TupleTableSlot *) PG_GETARG_POINTER(0);
      /*Declare Variables needed*/int32 emp_age;
      bool isnull;

      /*Get the 'age' attribute from the row, this function defined in
      executor.h*/

      emp_age=DatumGetInt32(GetAttributeByName(t,"age",&isnull));

      /*if not a valid result,return NULL*/
      if(isnull)
      {
          PG_RETURN_NULL();
      }

      /*Return Age Comparison*/
      PG_RETURN_BOOL(age>20);
```

```
}
```

将此函数编译成共享对象后，它可以被定义和用于 PostgreSQL。例如：

```
CREATE FUNCTION islegal(payroll) RETURN bool AS
      '/usr/local/pgsql/lib/islegal.so'
      LAGUAGE 'C';

>SELECT islegal(payroll) FROM payroll WHERE name='Barry';

islegal
-------
t
```

2. 编码技巧

下面的技巧和指南摘自 *PostgreSQL Programmer's Guide*（请从网站www.postgresql.org了解此向导的更多信息）：

- 包含安装在目录/usr/local/pgsql/include 或等价路径中的文件。
- 分配内存时，使用 Postgres 程序 palloc 和 pfree 而不是使用标准 C 函数 malloc 和 free。palloc 保留内存对每个事务自动释放，以此防止内存泄漏。
- 经常使用 memset 或 bzero 将结构归零。即使你初始化了结构中的所有字段，结构中仍可能有几个字节包含垃圾值的对齐性填充（结构空洞）。
- 通常，程序要求至少包含 postgres.h 和 fmgr.h 文件。postgres.h 中声明了内部 Postgres 类型，函数管理器接口（PG_FUNCTION_ARGS 等等）位于 fmgr.h 中。由于兼容性原因，最好在其他系统或用户头文件之前首先包含 fmgr.h。
- 在对象文件内定义的符号名称不能相互冲突或与 PostgreSQL 服务器执行文件中的定义的其他符号冲突。如果由此产生了错误信息，你必须重新命名函数或变量。

14.2　扩展类型

PostgreSQL 拥有极多的内置数据类型（请参见第 2 章“PostgreSQL 数据类型”）。不过，在某些场合，创建自定义函数更为有利。

PostgreSQL 中的所有数据类型都可以被定义为以下两种类型之一：基本类型和复合类型。

基本类型，如 int4，用 C 写成再编译到系统中。不过，自定义数据类型可以编译成共享对象并使用 CREATE TYPE 命令与后端连接。

任何时候创建新表，同时创建了复合类型。起初，你可能非直观性地将表看作一个类型。但是，表仅仅是按指定顺序排列单个数据类型集合。依此而论，表可以看作为简单单个元素数据类型的一种“复合”或复杂的集合。

创建数据类型

要创建一个自定义基本类型，必须定义两类函数：一个是输入（input）函数，一个是

输出（output）函数。

input 函数负责接收一个 NULL 分隔符字符串并放入内存，同时它返回一个内部象征值。

output 函数访问元素的内部象征值并返回原始的 NULL 分隔符字符串。

PostgreSQL 7.1 Programmer's Guide 中有一些创建自定义数据类型的优秀例子。

首先你必须定义复杂数据类型的结构：

```
type struct Complex{
  double  x;
  double      y;
}Complex;
```

下一步，定义 input 和 output 函数：

```
Complex *
Complex_in(char *str)
{
      double x,y;
      Complex *result;
      if(sscanf(str,"(%lf,%lf)",&x,&y)!2){
          elog(ERROR," complex_in:error in parsing %s",str);
          return NULL;
      }
      result=(Complex *)palloc(sizeof(Complex));
      result->x=x;
      result->y=y;
      return (result);
}

char *
complex_out(Complex * complex)
{
      char *result;
      if (complex==NULL)
          result(NULL);
      result=(char*)palloc(60);
      sprintf(result,"(%g,%g)",complex->x,complex->y);
      return(result);
}
```

请注意一定要确保 input 和 output 函数相互呼应。否则，倾倒出的数据（即复制到一个文件中）不能正确回读。

上面的代码编译成静态对象后，必须创建对应的 SQL 函数在数据库中注册它：

```
CREATE FUNCTION complex_in(opaque)
      return complex
      AS 'PGROOT/tutorial/obj/ complex.so'
      LANGUAGE 'C';

CREATE FUNCTION complex_OUT(opaque)
```

```
        return opaque
        AS 'PGROOT/tutorial/obj/ complex.so'
        LANGUAGE 'C';
```

最后，使用 CREATE TYPE 命令来定义刚才创建的自定义基本类型的属性：

```
CREATE TYPE complex(
        internallength=16,
        input= complex_in,
        output= complex_out);
```

14.3　扩展操作符

PostgreSQL 将操作符用作实现数据比较和聚集的方法。PostgreSQL 操作符有 3 类：左一元、右一元和二元。二元操作符也许是最常用的。实质上，当操作符位于两上单独的数据类型之间时，它就是二元的（如 21>20）。二元数据类型的一个典型例子是大于号（>）：它位于两个数据元素中间并根据比较结果返回一个布尔值。更基础的是加号操作符（+），它将两边值相加并返回和值（如 2+3 返回 5）。

一元操作符仅从一边接受数据，所以又分左一元或右一元操作符。右一元操作符的一个例子是阶乘操作符（!）：它位于一个整数的左边并返回阶乘结果（如!4）。

操作符必须针对需要处理的指定数据类型来定义。例如，>操作符根据处理对象是整数还是几何元素而执行不同的动作。因此，必须明确指定自定义操作符将要处理的数据类型。

定义自定义操作符

定义一个操作符前，必须首先创建相关函数。这些函数可以定义为过程函数（如 SQL、PL/pgSQL 等等）或与编译过的 C 对象文件连接。

在下面例子中，创建了一个接受两个整数的函数。它将相加此两整数。如果结果大于 100，则返回一个 TRUE；否则返回 FLASE。为了执行此动作创建了一个简单的 SQL 函数：

```
CREATE FUNCTION addhund(int4,int4) RETURNS boolean AS '
BEGIN
        IF ($2+$1)>100 THEN
            RETURN 't';
        END IF;
        RETURN 'f';
END;
'LANGUAGE 'plpgsql' WITH (iscachable);
```

你可以直接测试此函数：

```
SELECT addhund(99,99) AS answer;

answer
------
t
```

```
SELECT addhund(9,9) AS answer;

answer
------
f
```

下一步，此函数使用 CREATE OPERATOR 命令捆绑于指定的操作符字符：

```
CREATE OPERATOR +++)
      leftarg=int4,
      rightarg=int4,
      procedure=addhund,
      commutator=+++);
```

以上命令将它定义为一个二元操作符，左右边数据为 int4 数据类型。另外，它还定义了此操作符的 COMMUTATION 优化为它自身。

此新操作符可以测试如下：

```
SELECT 11 +++ 90 AS answer;

answer
-------
t

SELECT 9 +++ 90 AS answer;

answer
-------
f
```

优化注释

操作符优化是为了告知数据库不同的操作符之间的关联性如何。在创建操作符时可以使用若干优化设置。

COMMUTATION

上一个例子为+++操作符定义了 COMMUTATION 优化为它自身。一般，COMMUTATION 优化定义仅对二元操作符有意义。它描述了操作符两边的数据转向后的关系变化。例如，请对比以下两个加操作符的关系：

3+8=11

8+3=11

可以看到，加操作符具有自身可交换性。这就意味着操作符两边的数据位置交换后不会对结果有影响。与之相反，减号操作符就有所不同：

3-8=-5

8-3=5

在这里，数据元素的位置变化会对结果产生影响。所以，减号不具有自身可交换性。

NEGATOR

创建操作符时可用的另一个设置是为当前定义取反（如果可以的话）。例如，等号操作符取反就是不等于操作符（如，a=b 取反就成了 a<>b）。

RESTRICT

RESTRICT 优化子句只对返回布尔结果的二元操作符有效（如 a>b）。它为查询优化提供满足一般 WHERE 子句特定选择的线索。标准评估器如表 14-3 所示。

表 14-3　　标准评估器

评估器	描　述	用　途
eqsel	等于选择	=
neqsel	不等于选择	<>
scalarltsel	标量小于选择	<或<=
scalargtsel	标量大于选择	>或>=

JOIN

JOIN 优化一般只对返回布尔结果的二元操作符有效（如 a=b）。JOIN 优化提供对一般 WHERE 子句中两个表相互匹配行数量的评估（如 payroll.empid=employee.empid）。

评估子句中的可定义值如表 14-4 所示。

表 14-4　　评估子句中的可定义值

评估器	描　述	用　途
Eqjoinsel	等于选择	=
Nejoinsel	不等于选择	<>
scalarltjoinsel	标量小于选择	<或<=
scalargtjoinsel	标量大于选择	>或>=
areajoinsel	2D 面积比较	N/A
positionjoinsel	2D 位置比较	N/A
contjoinsel	2D 包含比较	N/A

HASHES

如果存在 HASHES 子句，就表示对操作符允许尝试散列联结。HASHES 优化只对返回布尔结果的二元操作符有效。

一般，当表示两个数据类型绝对相等（如 a=b）时 HASHES 才有意义。若操作符并不提供绝对相等性的比较，则散列联结用处不大。

SORT1 和 SORT2

此子句集用于提示优化器是否允许对操作符左边或右边尝试融合联结。

以上优化选项的使用是非常有限的。实际上，它们通常只对等号（=）操作符有效。而且，这两个引用操作符常为<。

CREATE OPERATOR 命令不会对优化选项的有效性作任何完备性检查。所以，即使成功创建了指定的操作符，但实际应用中仍有可能失败。事实上，当不满足下面条件之一时，使用 SORT1 和 SORT2 优化选项就会导致失败：

- 融合联结相等操作符必须有一个换向器（如果两个数据类型相同则应该为自身）。
- 必须存在<和>操作符，将相同数据类型作为指定的排序操作符。

第五部分

附　录

附录 A　参考资源

大量的资源可供 PostgreSQL 用户参考。大量邮件列表、网站及书籍提供了相关材料，它们或繁或简。

PostgreSQL 与其他 RDBMS

初学者对 RDBMS 的第一个问题是“谁是最好的？”如果对数据库所需功能缺乏完全的了解，此问题几乎无法回答。

比较数据库尤如比较交通工具一样。每种交通工具适应于特定的任务；摩托车与小型卡车相比，在某些场合可能具有优越性，在其他场合就可能更具危险性。同样，在正确选择数据库前，必须明白 RDBMS 的所需功能。

若你非要对“苹果”和“橙子”加以比较，下面我们就给出现在流行 RDBMS 的简短列表，同时列出它们的优势、弱点及典型应用。

PostgreSQL

优势：

- 广泛的支持和开发团体。
- 众多的商业支持（如 Greate Bridge、Red Hat 及其他）。
- 完全开发源代码。
- 无使用授权费用。
- 拥有健全的功能集。
- 支持众多内部过程语言，它们可创建存储过程、触发器、规则和函数。
- 广泛的 API 访问解决方案，包括 ODBC、JDBC、C、Perl、PHP 及 Python。
- 完整事务执行。
- 优秀用户扩展。
- 支持数据库版本并发控制（MVCC）机制。
- 完全 ACID 兼容数据库（原子性、并发、I 和 D）。
- 提供了与 PostgreSQL 接口的许多 web 工具。
- 支持外键。
- 允许在线备份恢复。

缺点：

- 在现行版本（7.1）中无复制。
- 无多线程（在 NT 环境中可能导致问题）。
- 目前只有少数 GUI 管理工具可用。

典型应用：

- 当你审视 PostgreSQL 的功能、开发工具、可行性及操作性能时，你很难找到其缺点。作为一个中型数据库，PostgreSQL 不可能战无不胜。虽然它仍是一个源码开放的项目，但得到了许多商业实体的支持。
- 评价 PostgreSQL 时唯一可能需要注意的是它即将运行的特定环境。虽然可以使用 NT 版本的数据库，但在 UNIX 之类环境下运行更佳。

MySQL

优势：

- 广泛的支持和开发团体。
- 作为 web 后端非常流行。
- 开放源代码，因而可以自定义编译使用。
- 无使用授权费用。
- 执行基本 SELECT 效率高。
- 运行所需空间小。
- 运行于不同的操作系统。
- 具备与许多流行 web 开发工具的接口。

缺点：

- 不能执行复杂的联结和子选择。
- 本身不支持事务。
- 对锁无原子性。
- 不支持存储过程。
- 无触发器。
- 无外键。

典型应用：

- MySQL 是服务于动态 web 页的上佳选择，尤其是不需要事务或复杂查询时。另外，MySQL 易于安装、配置和管理。
- 由于 MySQL 功能针对性如此强，所以它在执行原始 SELECT、INSERT 或 UPDATE 的速度上要胜过其他 RDBMS。然而，因为 MySQL 缺乏真正的事务支持及行级锁，如果同时执行多个 INSERT 或 UPDATE 命令的话，其速度就会下降。
- 一般，MySQL 是小型到中型数据库的理想选择，尤其当无需同时执行多个 INSERT 和 UPDATE 查询时。

Microsoft SQL Server

优势：

- GUI 界面便于安装和管理。
- 培训和支持随处可及。

- 对事务和原子化的良好支持。
- 与其他 Microsoft Office 应用程序无缝集成。
- 支持复制和簇。

缺点：

- 仅能运行于 Microsoft 操作系统上。
- 由开发者自行控制的商业化软件，不能自定义编译使用。
- 安装开销较大。

典型应用：

- 整体上讲，MS-SQL 是中小型数据库方案的有效引擎。若限运行于 Microsoft 操作系统和应用程序上，则选择它较适合。如果需要高度的自定义功能及多样化支持，则选择其他数据库更有效。

Interbase

优势：

- 最新的开发源代码。
- 支持包括事务、触发器、用户自定义函数等许多功能。
- 无使用授权费用。
- 可运行于 Windows、Linux 和 Solaris。
- GUI 界面使 Windows 平台上的安装和配置更加容易。
- 与 Dephi 和 PowerBuilder 接口良好。
- 具备 Borland 及其他商业支持。
- 一些强大的企业级功能。

缺点：

- 新开放的源代码。
- 缺少一些更高级的 SQL 语句（如 CASE、NULLIF 及 COALESCE）。
- 函数只能用 C 函数编写。
- Linux 版本相当新。
- 略显零散的开发团体。

典型应用：

- Interbase 是一个全功能的 RDBMS，它过去是由开发公司控制的商业化软件，现在已是开放的源代码了。虽然这对它的进一步发展和安全性提高大有好处，但也同时存在缺点。开放的源代码仍需要得到此项目团体的继续开发，但开发团体现在有所离析。
- Interbase 接口能与许多流行开发语言配合良好。而且，经过了长期的发展，它几乎拥有了现代 RDBMS 所要求的所有功能。

DB2

优势：

- 扩展性强，包括簇。
- 拥有很先进的复制功能。

- 支持多字节数据库。
- 支持现代 RDBMS 的所有功能。
- 多年的产品完善。
- 支持多平台，包括小型嵌入式环境。
- 可用 7×24 支持选项。
- 随处可及的培训和用户组支持。

缺点：

- 商业化软件。
- 授权使用费庞大。
- 简单任务的配置相当复杂。

典型应用：

- 企业用于大型、复杂数据库，包括需求全功能 RDBMS 场合。
- 正确安装和配置相当复杂。因而，DB2 对于中小型数据库方案无太大意义。不过，作为大型数据库系统的后端，它是无以胜出的。

Oracle

优势：

- 扩展性强，包括簇。
- 支持大型平行结构。
- 支持多字节数据库。
- 支持现代 RDBMS 的所有功能。
- 多年的产品完善。
- 支持多平台，包括小型嵌入式环境。
- 可用 7×24 支持选项。
- 随处可及的培训和用户组支持。

缺点：

- 商业化软件。
- 授权使用费庞大。
- 简单任务的配置相当复杂。

典型应用：

- Oracle 通常被视作旗舰级的商业化 RDBMS。Oracle 公司耗费了巨大的时间和金钱来开发此保证几乎 100%无故障运行的可靠系统。不过，实现这些高级功能耗费了极其巨大的财力。
- 一般而言，正确安装和配置可能相当复杂。因而，Oracle 对于中小型数据库方案无太大意义。不过，作为大型数据库系统的后端，它是无以胜出的。

在线 PostgreSQL 资源

下面提供许多在线资源，帮助您安装、管理和开发 PostgreSQL。

网站

可以在站点www.postgresql.org找到完整的镜像站点列表。在那里，还可以找到许多商业支持站点。

1. 镜像站点

澳大利亚

postgresql.planetmirror.com

加拿大

www.ca. postgresql.org/index.html

德国

postgresql.bnv-bamberg.de

意大利

www. postgresql.uli.it

俄国

postgresql.rinet.ru

美国

postgresql.readysetnet.com

2. 商业支持站点

Great Bridge（www.greatbridge.com/）。PostgreSQL 商业产品、服务及支持。

PostgreSQL 公司（www.pgsql.com/）。对 PostgreSQL、数据库主机的支持及提供宣传材料。

软件研究协会（Software Research Associates）（osb.sra.co.jp/）。对开放源代码软件的支持——从 1999 年 4 月开始提供帮助客户开发源代码基本软件系统的诸多服务。

Cybertec Geschwinde & Schvnig OEG（postgres.cybertec.at/）。位于奥地利的维也纳。在全德语语种区（奥地利、德国和瑞士）提供培训课程、技术支持、咨询、高效终端系统、高可靠性解决方案。

dbExperts（www. dbexperts.com.br）。位于巴西。以葡萄牙语提供 PostgreSQL 的培训课程、开发专业化支持及商业产品

Applinet（www.applinet.nl/）。位于荷兰。提供 PostgreSQL 的咨询服务。

Command Prompt 公司（www.commandprompt.com/）。提供 Linux 管理服务及 PostgreSQL 支持。位于太平洋西北，Command Prompt 公司致力于 Linux 和 PostgreSQL 的技术支持，包括定制 PostgreSQL、C++ 、PHP 和 Perl 程序。

邮件列表

PostgreSQL 用户和开发团体拥有一套活跃的邮件列表。订阅其中列表的过程如下：

（1）致信<groupname>-request@postgresql.org（其中 groupname 为下面组名之一，如 pgsql-hackers-request@postgresql.org）。

（2）在信正文中写上“subscribe”或“unsubscribe”。

（3）或者，也可以使用“set digest”订阅合订邮件而不是订阅多封单个邮件。

（4）也可以在信正文中写上“set nomail”。它将终止发送邮件但保持订阅身份。（对于下面新闻组选项有用。）

下面是目前可用的邮件列表：

■ pgsql-admin@postgresql.org

主题包括 PostgreSQL 管理和相关问题。

■ pgsql-announce@ postgresql.org

对第三方和相关问题的布告组。

■ pgsql-bugs@ postgresql.org

报告和检查已发现病毒的邮件列表。

■ pgsql-cygwin@postgresql.org

使用 cygwin 在 Windows 计算机上运行 PostgreSQL 的讨论组。

■ pgsql-general@postgresql.org

一般讨论区。不包括安装、编译或错误。通常，这是最活跃的列表。

■ pgsql-hackers@postgresql.org

开发者或对 PostgreSQL 代码库感兴趣者的邮件列表。

■ pgsql-interface@postgresql.org

讨论 PostgreSQL 后端外部 API 的列表。（注意：对于 ODBC 和 JDBC 接口有单独的列表）

■ pgsql-jdbc@postgresql.org

用于讨论外部 JDBC Java 接口。

■ pgsql-odbc@postgresql.org

用于讨论外部 ODBC 接口。

■ pgsql-php@postgresql.org

讨论 PostgreSQL 和 PHP 的使用。

■ pgsql-sql@postgresql.org

SQL 语言各方面的讨论区。

新闻组

你可以从 PostgreSQL 新闻服务器（news://news.postgresq.org）上订阅许多新闻组。虽然每人都可以阅读这些新闻组，但你必须先订阅以上邮件列表之一。

comp.database.postgresql.admin

comp.database.postgresql.announce

comp.database.postgresql.bugs

comp.database.postgresql.committers

comp.database.postgresql.docs

comp.database.postgresql.geeral

comp.database.postgresql.hackers

comp.database.postgresql.hackers.rmgr

comp.database.postgresql.hackers.oo
comp.database.postgresql.hackers.smgr
comp.database.postgresql.hackers.wal
comp.database.postgresql.interfaces
comp.database.postgresql.interfaces.jdbc
comp.database.postgresql.interfaces.odbc
comp.database.postgresql.interfaces.php
comp.database.postgresql.mirrors
comp.database.postgresql.novice
comp.database.postgresql.patches
comp.database.postgresql.ports
comp.database.postgresql.ports.cygwin
comp.database.postgresql.questions
comp.database.postgresql.sql

FTP 站点

镜像 FTP 站点是获得 PostgreSQL 相关源代码及二进制包的主要途径。主要 web 站点 www.postgresql.org列出了此地址集。下面是较流行的站点：

- 澳大利亚

ftp.planetmirror.com/pub/ postgresql

- 加拿大

ftp.jack-of-all-trades.net/www. postgresql.org
looking-glass.usask.ca/pub/ postgresql
postgresql.wavefire.com

- 德国

ftp.leo.org/pub/comp/os/unix/database/ postgresql
ftp-stud.fht-esslingen.de/pub/Mirrors/ftp. postgresql.org

- 意大利

ftp. postgresql.uli.it
postgresql.theomnistore.com/mirror/ postgresql
bo.mirror.garr.it/mirror/ postgres

- 日本

ring、asahi-net.or.jp/pub/misc/db/ postgresql

- 俄国

ftp.chg.ru/pub/databases/ postgresql
postgresql.rinet.ru

- 英国

postgresql.rmplc.co.uk/pub/ postgresql

- 美国

postgresql.readysetnet.com/pub/ postgresql

download.sourceforge.net/pub/mirrors/ postgresql
ftp.digex.net /pub/packages/database/ postgresql
ftp.crimelabs.net/pub/ postgresql

参 考 书 籍

Momjian, Bruce、*PostgreSQL: Introduction and concepts*、Reading, MA: Addison Wesley, 2000。

Lockhart, Thomas、*PostgreSQL Programmer's Guide*、New York: iUniverse.com, 2000。

Matthew, Neil 等、*Professienal Linux Programming*、Chicago: Wrox Press, Inc., 2000。

附录 B　PostgreSQL 版本信息

PostgreSQL 在不断发展，而且近几年，PostgreSQL 又增加了许多新的功能。下面简单列出了每个新版本的主要变化/更正错误。有关更完整的列表可查看文件 ChangeLog（通常位于/usr/local/pgsql/ChangeLogs）。

版本 7.1.2（2001 年 5 月发布）

更正 PL/PgSQL SELECT 无行返回时的错误。
更正 psql 反斜杠导致核心倾倒的错误。
更正引用完整性权限。
pg_dump 的清除的错误更正。

版本 7.1.1（2001 年 5 月发布）

允许 pg_dump 在 7.0 版本数据库上操作。
允许 EXTRACT 接受字符串参数。
JOIN 错误更正。
ODBC 错误更正。
Python 错误更正。
函数中的整个元组错误更正。
AIX、MSWIN、VAX 和 N32K 错误更正。

版本 7.1（2001 年 4 月发布）

实现了 WAL（预读式日志）。改进了数据库的一致性并减少了由系统崩溃导致的数据丢失的可能性。

实现了 TOAST（超属性保存技术）。允许在表中保存任意大小的行。删除了以前固定行长度的限制。

实现了外部联结。

更正了实现64位CPU运行中的错误。
优化查询引擎。
查询父表时缺省情况访问其继承表。

版本 7.0.3（2000年11月发布）

大对象错误更正。
SELECT FOR UPDATE 错误更正。
在配置中添加 enable --with -syslog。
允许 PL/PgSQL 接受非 ASCII 标识符。
强迫 VACUUM 经常冲洗缓存。
更正查询中的!=错误。
更正了实现当发生写错误时终止数据库启动的错误。
更正处理 TIME 聚集中的错误。
更正向 CHAR 类型数据中插入多字节长字符串的错误。

版本 7.0.2（2000年6月发布）

更正许多 CLUSTER 失败的错误。
ALTER TABLE RENAME 命令可用于索引。
更正 PL/PgSQL 处理 interval 和 timestamps 对话的错误。
更正在 pgaccess 中创建用户函数的错误。
IRIX 和 QNX 错误更正。
JDBC 结果集错误更正。
更正 UNLISTEN 失败的错误。
更正 RENAME TABLE 失败后不能正确恢复的错误。

版本 7.0（2000年5月发布）

实现外键（FOREIGN KEYS，除 PATIAL MATCH FKS 外）。
对优化器的主要检查。
Psql 客户应用功能增强。
Date-time 数据类型与 SQL-92 兼容。
删除了查询字符串的固定长度限制。
索引中的键最大数由8增加到16。
排序和散列的错误更正，使之能处理大于2GB的数据。

版本 6.5.2（1999 年 9 月发布）

更正子选择和 CASE 中的错误。
更正 WHERE 联结中 CASE 的错误。
修复对大量 UNIQUE 和 PRIMARY KEY 索引的检查。
改进引用完整性；允许检查多列约束。
更正允许 MB 时 Win32 MAKE 的错误。
修正和减少 VACUUM 的内存消耗。
更正 timestamp(date,time)的错误。
更正一元操作符在规则解析器中的错误。
更新 pgaccess 0.98 版本。

版本 6.5.1（1999 年 7 月发布）

修正对 linux_ppc、Irix、alpha 及 OpenBSD 的兼容性问题。
淘汰 QUERY_LIMIT 而使用 SELECT...LIMIT。
更正 EXPLAIN 在继承上的错误。
添加允许 VACUUM 作用于多片断表上的补丁。
更正 R-Tree 优化器选择性的错误。
更正 ACL 文件描述符泄漏的错误。
避免对只读事务进行磁盘写操作。
更正实现如果放弃最后一个事务删除临时表中的错误。
更正实现防止在 plpgsql 中创建太大元组的错误。
允许连接端口号 32KB-64KB。
添加^前导符。
修正时间值上的微秒错误。
新添 linux_m68k 端口。
更正某些情况下 NULL 的排序错误。
更正共享库的独立性错误。
更正子选择中 GROUP BY 的失灵。

版本 6.5（1999 年 6 月发布）

多版本并发控制（MVCC）——删除了旧版本中的表级锁机制。
从 pg_dump 的热备份——允许操作数据库时备份/恢复它。

数字数据类型——添加新的数字数据类型。

临时表——在数据库对话期间保证临时表具有唯一的名称，对话结束时予以删除。

端口——扩展端口列表，包括 WinNT/ix86 和 NetBSD/arm32。

新添 SQL 功能——添加 CASE、INTERSECT 和 EXCEPT 语句。还新添了 LIMIT/OFFSET、SET TRANSACTION ISOLATION LEVEL 和 SELECT...FOR UPDATE 命令，还改进了 LOCK TABLE 命令。

添加了 cacuumdb 命令。

允许 EXPLAIN 所有使用的索引。

新添 pg_dump 表输出格式。

添加字符串 min()/max()函数。

更新 pgaccess 至 0.96（约束）。

改进 substr()函数。

改进多字节处理（Tatsuo）。

新添 SERIALIZED 事务模式。

更正表超过 2GB 时的错误。

新添 SET TRANSCTION ISOLATION LEVEL。

新添 LOCK TABLE IN...MODE。

更新 ODBC 驱动。

新添 SELECT FOR UPDATE。

新添 TCL_ARRAYS（Massimo）选项。

新添 INTERSECT 和 EXCEPT（Stefan）。

新添 READ COMMITTED 隔离级别。

新添 TEMP 表/索引。

允许多规则动作。

新添 int8 和 text/carchar 类型间转换的程序。

允许缺省的右侧查询。

添加新的-O 选项允许变化系统表结构。

支持 char()数组和 varchar()字段。

UNION 支持非目标列表中的列 ORDER BY。

INET 类型在比较时检查网络掩码。

允许对 UNION 执行 VIEW。

版本 6.4.1（1999 年 12 月发布）

添加 pg_dump -N 标志来强迫在标识符周围使用双引号。此为缺省设置。

更正了 WHERE 子句中的 NOT 导致崩溃的错误。

EXPLAIN VERBOSE 核心倾倒错误更正。

更正测试表是否允许大小写混合及表名称中空格的错误。

更改从 SPI-*到 spi_*.的内置函数。

更新 pgaccess 至 0.93。

时区错误更正。

对于缺省值匹配强迫使用隐含类型。

版本 6.4（1998 年 10 月发布）

根据 Jan Wieck 重写规则系统中的大量新代码，视图和规则功能增强。他还在《程序员向导》中撰写了一章的内容。

与原来的 PL/pgTCL 一起，添加了第二个过程语言 PL/PgSQL。

支持多字节字符集。

解析器根据操作符和函数进行参数的自动类型强迫匹配以及目标列与列和表达式间的强迫匹配。它利用了 Postgres 类型扩展性功能的一般机制。“用户向导”里新的一章讲到了此部分内容。

添加了 3 个新的数据类型。两个类型 inet 和 cidr 支持不同形式的 IP 网络、子网和计算机地址。现在在一些平台上可用 8 字节的整数类型了。第 4 个类型 serial 被作用 int4、序列和唯一索引的混合类型。

添加了更多的 SQL-92 兼容性语法特征，包括 INSERT DEFAULT VALUES。

显示 EXPAIN 中使用的索引。

EXPLAIN 调用规则系统并显示计划用于重写查询。

通过 configure 提示许多数据类型和函数中的多字节类型。

新添 configure --with -mb 选项。

新添 inidb --pgencoding 选项。

新添 createdb -E 多字节选项。

Libpq 现在允许客户异步。

允许从客户后端查询中取消。

NOTIFY 发送发送者的 PID，因此你可以判断它是否属于自己的通知。

添加了 varchar 和 bpchar 间转换的程序。

添加了允许在目标列中调节 varchar 和 bpchar 类型大小的程序。

添加了数据读取时的位标志来支持 timezone 小时和分钟。

实现了 SQL-92 定义中的 TIMEZONE_HOUR、TIMEZONE_MINUTE。

检查和适当时忽略外键列约束。

新添 psql 命令"SET CLIENT_ENCODING TO 'encoding'"；对于多字节，参见 /doc/README.mb。

允许在 Win32 下编译 Libpq。

对引用表/列名称的更好支持。

在 pg_dump 中用双引号包括表和列名称。

允许子选择中的 UNION。

新添 HAVING 子句，对子选择和联合全面支持。

支持 SQL-92 语法"SET NAMES"。

支持 LATIN2-5。

允许用 OR 子句进行索引。

EXPLAIN VERBISE 可完美输出规则到 postmaster 日志文件。

允许对函数执行 GROUP BY。

新的重写系统更正了规则和视图的许多错误。

系统索引允许多键。

删除 oidint2、oidint4 和 oidname 类型。

新添 SERIAL 数据类型：自增序列/索引。

新添 UNLISTEN 命令。

在命令行可用 createuser 选项。

添加了针对 64 位整数（int8）的代码并进行了测试。

新添 pg_upgrade 命令。

新添 CREATE TABLE DEFAULT VALUES 语句。

新添 INSERT INTO TABLE DEFAULT VALUES 语句。

新添 DECLARE 和 FETCH 功能。

允许多达 8 个键的索引。

删除不再使用的 ARCHIVE 关键字。

新添 SET QUERY_LIMIT。

版本 6.3（1998 年 3 月发布）

允许带 EXISTS、IN、ALL 和 ANY 关键字的子选择（Vadim、Bruce 和 Thomas）。

添加 SQL-92 “常量”：CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP 和 CURRENT_USER。

修改约束语法使之与 SQL-92 兼容。

使用索引实现 SQL-92 PRIMARY KEY 和 UNIQUE 子句。

识别 SQL-92 的 FOREIGN KEY 语法。

允许使用 NOT NULL UNIQUE 子句（以前只允许单独使用）。

允许使用非常量的 Postgres 样式表达（::）。

添加了对 SQL3 TRUE 和 FALSE 布尔常量的支持。

支持 SQL-92 语法 IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE。

允许更简短的布尔值字符串（如 t、tr、tru）。

允许使用 SQL-92 分隔标识符。

实现了 SQL-92 二进制和十六进制字符串解码（b'10'和 x'1F'）。

支持 SQL-92 语法：强迫字符串表示为某类型（如''DATETIME 'now'）。

添加了 int2、int4 和 OID 类型与文本间的相互转换。

新添 SQL 语句 CREATE PROCEDURAL LANGUAGE。

新添 Postgres 过程语言（PL）后端接口。

对 LIKE 和~、!~操作符使用索引。

对 datetime 和 timespan 添加散列函数。
对后端和前端库添加支持 UNIX 域套接字。
实现 CREATE DATABASE/WITH LOCATION 和 initlocation 应用。
后端环境变量 TZ 使用 SET/SHOW/RESET TIME ZONE。
实现了 SET keyword = DEAULT 和 SET ZONE DEFAULT。
系统表/索引名称的 16 个字符限制增加到 32 个。
重新命名系统索引。
添加 SET DATESTYLE 的'GERMAN'选项。
通过 hh:mm:ss 域定义 ISO 样式的时间范围输出格式。
实现在 date_part()中输入某年的某日。
定义 timespan_finite()和 text_timespan()函数。
允许从系统密码文件中分离的 pg_password 认证数据库。
倾倒 ACLS、GRANT 和 REVOKE 权限。
定义 text、varchar 和 bpchar 字符串长度函数。
更正处理继承和开销计算查询中的错误。
实现 CREATE TABLE/AS SELECT（也可以为 SELECT/INTO）。
允许在约束中使用 NOT、IS NULL 的 IS NOT NULL。
实现 SELECT 的 UNION。
在 INSERT 中添加了 UNION、GROUP 和 DISTINCT。
JDBC 的大补丁。
新添 LOCK 命令并锁定描述死锁的手册页。
新添 psql \da、\dd、\df、\do、\dS 和\dT 命令。
在 psql \d 表中显示 NOT NULL 和 DEFAULT。
在 contrib/ip_and_mac 中新添 IP 和 MAC 地址类型。
新添 Python 接口（PyGreSQL 2.0）。
指定了新前端/后端协议的版本号和网络字节顺序。
pg_hha.conf 的安全性提高并写入文档；进行了许多清除。
在 SQL 预处理器中嵌入了 ecpg。

版本 6.2.1（1997 年 10 月发布）

新添 JDBC 驱动接口。
添加 pg_password 命令。
返回受 INSERT/UPDATE/DELETE 等影响或插入的元组数。
实现由 CREATE TRIGGER（SQL3）创建触发器。
SPI（服务器编程接口）允许在 C 函数内执行查询。
现实 SQL-92 标准中的 NOT NULL。
现实使用排它申明扩展注释（/*...*/）。
添加了//单行注释。

实现表的 DEFAULT 和 CONSTRAINT（SQL-92）。

添加文本串联操作符和函数（SQL-92）。

支持 WITH TIME ZONE 语法（SQL-92）。

支持时间间隔的单位到单位语法（SQL-92）。

定义了类型 DOUBLE PRECISION、INTERVAL、CHARACTER 和 CHARACTER VARYING（SQL-92）。

定义类型 FLOAT(P)和基本的 DECIMAL(p,s)、NUMBER(p,s)（SQL-92）。

定义了 EXTRACT()、POSITION()、SUBSTRING()和 TRIM()（SQL-92）。

定义了 CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP（SQL-92）。

添加了 UNION、HAVING、INNER 和 OUTER JOIN 的语法和警告（SQL-92）。

允许对 timespan/reltime 类型使用 hh:mm:ss 时间项。

对 lseg、path 和 polygon 添加了 center()程序。

对 circle-polygon 和 polygon- polygon 添加了 distance()程序。

添加了 circle-box 转换的程序。

用<->代替<= = =>距离操作符。

用>^代替 above（在上面）操作符!^，用<^代替 below（在下面）!|。

添加首尾文本修剪、子字符串和字符串定位程序。

添加 circle(box)和 poly(circle)间的转换程序。

允许在内存中而不是在文件中保存内部排序。

引用完整性的一般性触发器函数。

实现了 MOVE 功能。

版本 6.2（1997 年 6 月发布）

添加唯一（UNIQUE）索引功能。

添加主机名/用户访问级别控制而不是仅仅通过主机名和用户。

添加<>的同义词!=。

允许 select oid, * from table 的形式。

允许 BY、ORDER BY 通过序号或非别名 table.column 来指定列。

允许从前端执行 COPY。

允许 GROUP BY 中使用列别名。

允许限制创建 C 函数的用户。

改变从 float4 到 float8 的缺省小数常量表示。

当启动 postmaster 时设置欧洲日期格式。

如果没有找到大小写准确的名称则执行小写名称的函数。

本地用户的 identd 认证。

实现 BETWEEN 限定。

实现 IN 限定。

pg_dump 允许倾倒 oid。

pg_dumpall 倾倒所有的数据库和用户表。
防止 postmaster 以 root 身份运行。
Psql 允许在行中使用反斜杠和分号。
本地用户的安全性认证。
Vacuum 命令允许 VERBOSE 选项。

版本 6.1（1997 年 5 月发布）

BTREE UNIQUE 添加到大容量装载代码中。
对 libpg++的大量修改（Leo）。
新添加 GEQO 优化器加速了表的多表优化。
新添对在唯一键中进行非唯一性插入的警告信息。
新添纯文本密码函数。
新添 ANSI timestamp 函数。
新添 ANSI time 和 date 类型。
多列 B-Tree 索引。
新添 SET var TO 赋值命令。
新添对字符类型的本地设置。
新添 SEQUENCE 序列号生成器。
允许使用 GROUP BY 函数。
新添 MONEY 数据类型。
对属性统计和特定列新添 VACUUM 选项。
新添 SET、SHOW 和 RESET 命令。
新添\connect database USER 选项。
新添 destroydb -i 选项。
新添\dt 和\di psql 命令。
SELECT \n 转义成一个新行。

版本 Postgres95 .01（1995 年 5 月发布）

原始版本。