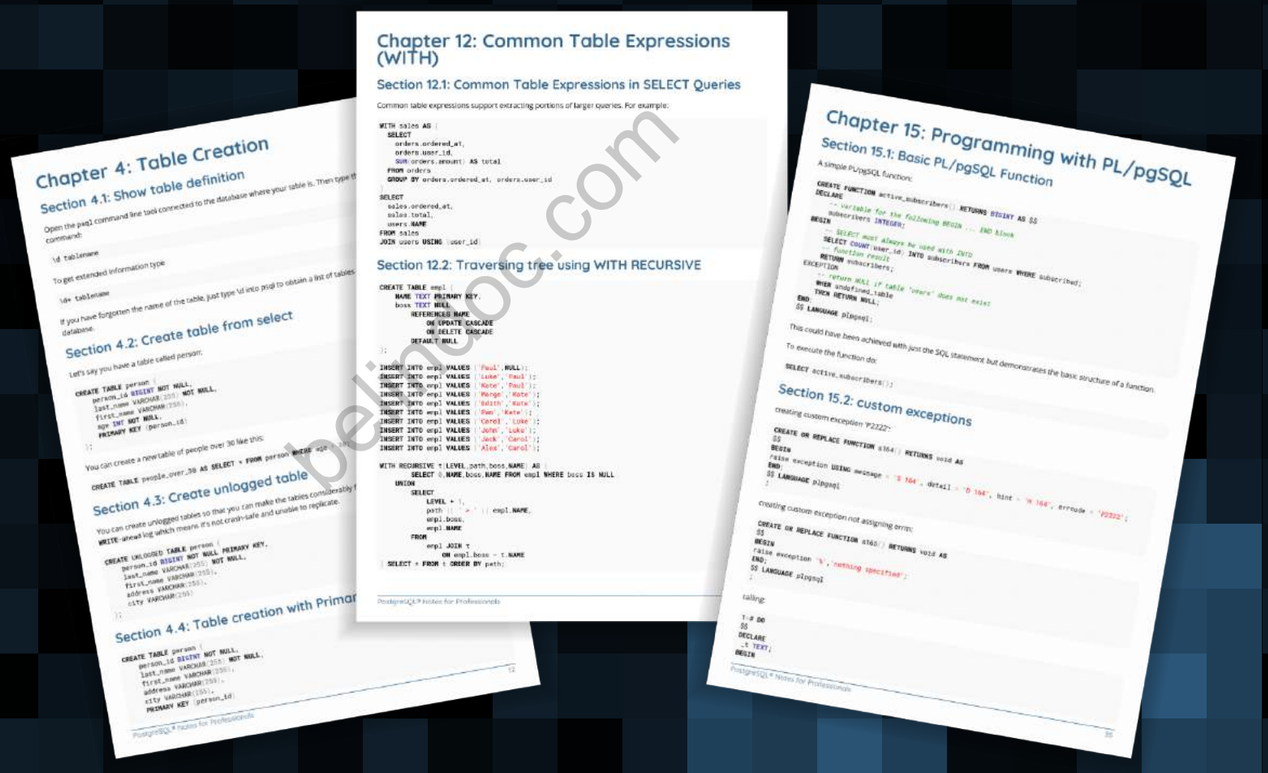
专业人员笔记

PostgreSQL®

专业人员笔记



60+ 页

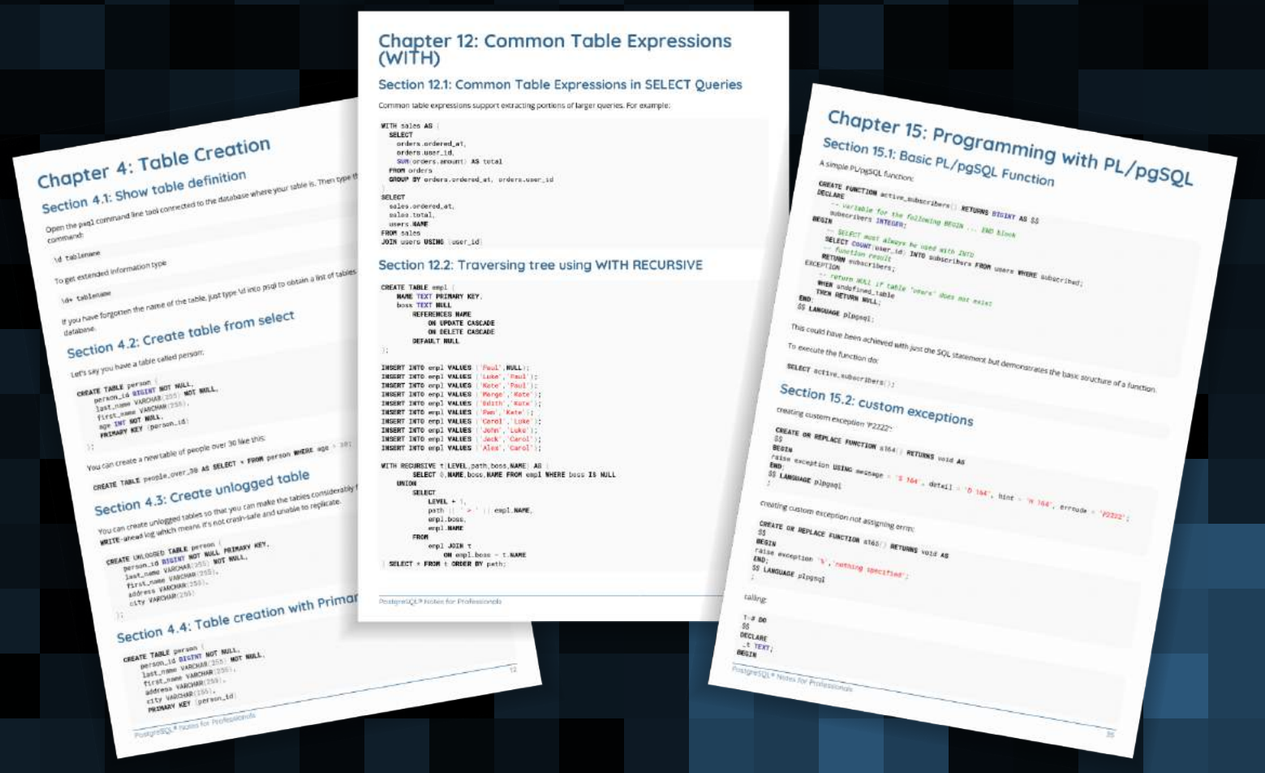
专业提示和技巧

GoalKicker.com

免费编程书籍

PostgreSQL®

Notes for Professionals



60+ pages

of professional hints and tricks

GoalKicker.com

Free Programming Books

目录

Contents

关于

欢迎免费与任何人分享此PDF，
本书的最新版本可从以下网址下载：
https://goalkicker.com/PostgreSQLBook

这是一本非官方的免费书籍，旨在教育用途，与官方PostgreSQL®组织或公司及Stack Overflow无关。
所有商标和注册商标均为其各自公司所有者的财产

本书中提供的信息不保证正确或准确，使用风险自负

请将反馈和更正发送至web@petercv.com

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/PostgreSQLBook

This is an unofficial free book created for educational purposes and is not affiliated with official PostgreSQL® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：PostgreSQL入门

版本	发布日期	终止支持日期
10.0	2017-10-05	2022-10-01
9.6	2016-09-29	2021-09-01
9.5	2016-01-07	2021-01-01
9.4	2014-12-18	2019-12-01
9.3	2013-09-09	2018-09-01
9.2	2012-09-10	2017-09-01
9.1	2011-09-12	2016-09-01
9.0	2010-09-20	2015-09-01
8.4	2009-07-01	2014-07-01

第1.1节：在Windows上安装PostgreSQL

虽然在生产服务器上使用基于Unix的操作系统（例如Linux或BSD）是良好的实践，但你可以轻松地在Windows上安装PostgreSQL（希望仅作为开发服务器）。

从EnterpriseDB下载Windows安装二进制文件：
http://www.enterprisedb.com/products-services-training/pgdownload 这是一家由PostgreSQL项目核心贡献者创办的第三方公司，他们针对Windows优化了二进制文件。

选择最新的稳定版（非Beta版）（撰写时为9.5.3）。你很可能需要Win x86-64包，但如果你运行的是32位Windows版本（在旧电脑上很常见），则选择Win x86-32。

注意：在Beta版和稳定版之间切换将涉及复杂的操作，如导出和恢复。仅在Beta版或稳定版内部升级时，只需重启服务。

你可以通过进入控制面板 -> 系统和安全 -> 系统 -> 系统类型，查看你的Windows版本是32位还是64位，那里会显示“##位操作系统”。这是Windows 7的路径，其他版本的Windows可能略有不同。

在安装程序中选择你想使用的软件包。例如：

- pgAdmin（https://www.pgadmin.org）是一个免费的数据库管理图形界面，我强烈推荐使用。在9.6版本中，它将默认安装。
- PostGIS（http://postgis.net）提供基于GPS坐标、距离等的地理空间分析功能，在GIS开发者中非常流行。
- 语言包提供官方支持的过程语言PL/Python、PL/Perl和PL/Tcl所需的库。
- 其他如pgAgent、pgBouncer和Slony等包适用于更大型的生产服务器，仅在需要时进行检查。

所有这些可选包都可以通过“应用程序堆栈构建器”后续安装。

注意：还有其他非官方支持的语言，如PL/V8、PL/Lua和PL/Java可用。

打开pgAdmin，通过双击服务器名称连接到服务器，例如“PostgreSQL 9.5 (localhost:5432)”。

从这里你可以参考诸如优秀书籍《PostgreSQL: Up and Running, 2nd Edition》（http://shop.oreilly.com/product/0636920032144.do）等指南。

Chapter 1: Getting started with PostgreSQL

Version	Release date	EOL date
10.0	2017-10-05	2022-10-01
9.6	2016-09-29	2021-09-01
9.5	2016-01-07	2021-01-01
9.4	2014-12-18	2019-12-01
9.3	2013-09-09	2018-09-01
9.2	2012-09-10	2017-09-01
9.1	2011-09-12	2016-09-01
9.0	2010-09-20	2015-09-01
8.4	2009-07-01	2014-07-01

Section 1.1: Installing PostgreSQL on Windows

While it's good practice to use a Unix based operating system (ex. Linux or BSD) as a production server you can easily install PostgreSQL on Windows (hopefully only as a development server).

Download the Windows installation binaries from EnterpriseDB:
http://www.enterprisedb.com/products-services-training/pgdownload This is a third-party company started by core contributors to the PostgreSQL project who have optimized the binaries for Windows.

Select the latest stable (non-Beta) version (9.5.3 at the time of writing). You will most likely want the Win x86-64 package, but if you are running a 32 bit version of Windows, which is common on older computers, select Win x86-32 instead.

Note: Switching between Beta and Stable versions will involve complex tasks like dump and restore. Upgrading within beta or stable version only needs a service restart.

You can check if your version of Windows is 32 or 64 bit by going to Control Panel -> System and Security -> System -> System type, which will say "##-bit Operating System". This is the path for Windows 7, it may be slightly different on other versions of Windows.

In the installer select the packages you would like to use. For example:

- pgAdmin (https://www.pgadmin.org) is a free GUI for managing your database and I highly recommend it. In 9.6 this will be installed by default .
- PostGIS (http://postgis.net) provides geospatial analysis features on GPS coordinates, distances etc. very popular among GIS developers.
- The Language Package provides required libraries for officially supported procedural language PL/Python, PL/Perl and PL/Tcl.
- Other packages like pgAgent, pgBouncer and Slony are useful for larger production servers, only checked as needed.

All those optional packages can be later installed through "Application Stack Builder".

Note: There are also other non-officially supported language such as PL/V8, PL/Lua PL/Java available.

Open pgAdmin and connect to your server by double clicking on its name, ex. "PostgreSQL 9.5 (localhost:5432).

From this point you can follow guides such as the excellent book PostgreSQL: Up and Running, 2nd Edition (http://shop.oreilly.com/product/0636920032144.do).

可选：手动服务启动类型

PostgreSQL作为后台服务运行，这与大多数程序略有不同。这在数据库和网络服务器中很常见。其默认启动类型为自动，意味着它会始终运行，无需你的任何操作。

为什么你会想手动控制PostgreSQL服务？如果你有时将电脑用作开发服务器，同时也用来玩电子游戏，例如，PostgreSQL运行时可能会稍微拖慢你的系统速度。

你为什么不想手动控制？如果经常启动和停止服务，会很麻烦。

如果你没有注意到速度上的差异，并且想避免麻烦，那么就将其启动类型保持为自动
并忽略本指南的其余部分。否则……

进入控制面板 -> 系统和安全 -> 管理工具。

从列表中选择“服务”，右键点击其图标，选择发送到 -> 桌面，以便更
方便地访问。

关闭管理工具窗口，然后从你刚创建的桌面图标启动服务。

向下滚动，直到看到名称类似 postgresql-x##-9.# 的服务（例如“postgresql-x64-9.5”）。

右键点击 postgres 服务，选择属性 -> 启动类型 -> 手动 -> 应用 -> 确定。你也可以同样轻松地将其改回自动。

如果列表中看到其他与 PostgreSQL 相关的服务，如“pgbouncer”或“PostgreSQL 调度代理 - pgAgent”，你也可以将它们的启动类型改为手动，因为如果 PostgreSQL 没有运行，它们也没多大用处。虽然这意味着每次启动和停止时会更麻烦，但这取决于你。它们使用的资源没有 PostgreSQL 本身多，可能对系统性能没有明显影响。

如果服务正在运行，其状态会显示为已启动，否则表示未运行。

要启动服务，右键点击并选择启动。会显示加载提示，应该很快自动消失。如果出现错误，请再试一次。如果仍然不行，说明安装时出现了问题，可能是因为你更改了 Windows 中大多数人不会更改的某些设置，因此查找问题
可能需要一些侦查。

要停止Postgres，请右键点击服务并选择停止。

如果在尝试连接数据库时遇到错误，请检查服务以确保其正在运行。

有关EDB PostgreSQL安装的其他非常具体的细节，例如特定PostgreSQL版本官方语言包中的python运行时版本，请始终参考官方EDB安装指南，链接中的版本请更改为您安装程序的主版本号。

第1.2节：在Linux上从源码安装PostgreSQL

依赖项：

- GNU Make版本 > 3.80
- 一个ISO/ANSI C编译器（例如gcc）
- 一个解压工具，如tar或gzip
- zlib-devel

Optional: Manual Service Startup Type

PostgreSQL runs as a service in the background which is slightly different than most programs. This is common for databases and web servers. Its default Startup Type is Automatic which means it will always run without any input from you.

Why would you want to manually control the PostgreSQL service? If you're using your PC as a development server some of the time and but also use it to play video games for example, PostegreSQL could slow down your system a bit while its running.

Why wouldn't you want manual control? Starting and stopping the service can be a hassle if you do it often.

If you don't notice any difference in speed and prefer avoiding the hassle then leave its Startup Type as Automatic and ignore the rest of this guide. Otherwise...

Go to Control Panel -> System and Security -> Administrative Tools.

Select "Services" from the list, right click on its icon, and select Send To -> Desktop to create a desktop icon for more convenient access.

Close the Administrative Tools window then launch Services from the desktop icon you just created.

Scroll down until you see a service with a name like postgresql-x##-9.# (ex. "postgresql-x64-9.5").

Right click on the postgres service, select Properties -> Startup type -> Manual -> Apply -> OK. You can change it back to automatic just as easily.

If you see other PostgreSQL related services in the list such "pgbouncer" or "PostgreSQL Scheduling Agent - pgAgent" you can also change their Startup Type to Manual because they're not much use if PostgreSQL isn't running. Although this will mean more hassle each time you start and stop so it's up to you. They don't use as many resources as PostgreSQL itself and may not have any noticeable impact on your systems performance.

If the service is running its Status will say Started, otherwise it isn't running.

To start it right click and select Start. A loading prompt will be displayed and should disappear on its own soon after. If it gives you an error try a second time. If that doesn't work then there was some problem with the installation, possibly because you changed some setting in Windows most people don't change, so finding the problem might require some sleuthing.

To stop postgres right click on the service and select Stop.

If you ever get an error while attempting to connect to your database check Services to make sure its running.

For other very specific details about the EDB PostgreSQL installation, e.g. the python runtime version in the official language pack of a specific PostgreSQL version, always refer to the official EBD installation guide , change the version in link to your installer's major version.

Section 1.2: Install PostgreSQL from Source on Linux

Dependencies:

- GNU Make Version > 3.80
- an ISO/ ANSI C-Compiler (e.g. gcc)
- an extractor like tar or gzip
- zlib-devel

- readline-devel或libedit-devel

源码：最新源码链接（9.6.3）

现在您可以解压源码文件：

```
tar -xzvf postgresql-9.6.3.tar.gz
```

PostgreSQL的配置选项非常多样：

完整安装步骤的完整链接

可用选项的小列表：

- --prefix=PATH 所有文件的路径
- --exec-prefix=PATH 架构相关文件的路径
- --bindir=PATH 可执行程序的路径
- --sysconfdir=PATH 配置文件的路径
- --with-pgport=NUMBER 指定服务器端口
- --with-perl 添加 Perl 支持
- --with-python 添加 Python 支持
- --with-openssl 添加 OpenSSL 支持
- --with-ldap 添加 LDAP 支持
- *--with-blocksize=BLOCKSIZE* 设置页面大小（KB）
    - BLOCKSIZE 必须是 2 的幂，且介于 1 到 32 之间
- *--with-wal-segsize=SEGSIZE* 设置WAL段大小，单位为MB
    - SEGSIZE 必须是1到64之间的2的幂

进入新创建的文件夹，使用所需选项运行配置脚本：

```
./configure --exec=/usr/local/pgsql
```

运行 make 以创建目标文件

运行 make install 以从构建的文件安装 PostgreSQL

运行 make clean 以清理整理

对于扩展，切换目录 cd contrib，运行 make 和 make install

第1.3节：在GNU+Linux上的安装

在大多数GNU+Linux操作系统上，可以使用操作系统的包管理器轻松安装PostgreSQL。

红帽系列

仓库可在此处找到：https://yum.postgresql.org/repopackages.php

使用以下命令将仓库下载到本地机器

```
yum -y install https://download.postgresql.org/pub/repos/yum/X.X/redhat/rhel-7-x86_64/pgdg-redhatXX-X.X-X.noarch.rpm
```

查看可用的软件包：

- readline-devel oder libedit-devel

Sources: Link to the latest source (9.6.3)

Now you can extract the source files:

```
tar -xzvf postgresql-9.6.3.tar.gz
```

There are a large number of different options for the configuration of PostgreSQL:

Full Link to the full installation procedure

Small list of available options:

- *--prefix=PATH* path for all files
- *--exec-prefix=PATH* path for architectur-dependet file
- *--bindir=PATH* path for executable programs
- *--sysconfdir=PATH* path for configuration files
- *--with-pgport=NUMBER* specify a port for your server
- *--with-perl* add perl support
- *--with-python* add python support
- *--with-openssl* add openssl support
- *--with-ldap* add ldap support
- *--with-blocksize=BLOCKSIZE* set pagesize in KB
    - BLOCKSIZE must a power of 2 and between 1 and 32
- *--with-wal-segsize=SEGSIZE* set size of WAL-Segment size in MB
    - SEGSIZE must be a power of 2 between 1 and 64

Go into the new created folder and run the cofigure script with the desired options:

```
./configure --exec=/usr/local/pgsql
```

Run make to create the objectfiles

Run make install to install PostgreSQL from the built files

Run make clean to tidy up

For the extension switch the directory cd contrib, run make and make install

Section 1.3: Installation on GNU+Linux

On most GNU+Linux operating systems, PostgreSQL can easily be installed using the operating system package manager.

Red Hat family

Respositories can be found here: https://yum.postgresql.org/repopackages.php

Download the repository to local machine with the command

```
yum -y install https://download.postgresql.org/pub/repos/yum/X.X/redhat/rhel-7-x86_64/pgdg-redhatXX-X.X-X.noarch.rpm
```

View available packages:

```
yum list available | grep postgres*
```

所需的软件包有：postgresqlXX postgresqlXX-server postgresqlXX-libs postgresqlXX-contrib

这些可以通过以下命令安装：yum -y install postgresqlXX postgresqlXX-server postgresqlXX-libs postgresqlXX-contrib

安装完成后，您需要以服务所有者身份（默认是postgres）启动数据库服务。此操作通过pg_ctl命令完成。

```
sudo -su postgres
./usr/pgsql-X.X/bin/pg_ctl -D /var/lib/pgsql/X.X/data start
```

要在命令行界面访问数据库，输入 `psql`

Debian 系列

在 Debian 及其衍生操作系统上，输入：

```
sudo apt-get install postgresql
```

这将安装 PostgreSQL 服务器软件包，版本为操作系统软件包仓库提供的默认版本。

如果默认安装的版本不是您想要的，可以使用包管理器搜索可能同时提供的特定版本。

您还可以使用PostgreSQL项目提供的Yum仓库（称为PGDG）来获取不同的版本。这可能允许获取操作系统软件包仓库尚未提供的版本。

第1.4节：如何通过MacPorts在OSX上安装PostgreSQL

为了在OSX上安装PostgreSQL，您需要知道当前支持哪些版本。

使用此命令查看您可用的版本。

```
sudo port list | grep "^postgresql[[:digit:]]\{2\}[[:space:]]"
```

您应该会得到类似以下的列表：

```
postgresql80                @8.0.26         databases/postgresql80
postgresql81                @8.1.23         databases/postgresql81
postgresql82                @8.2.23         databases/postgresql82
postgresql83                @8.3.23         databases/postgresql83
postgresql84                @8.4.22         databases/postgresql84
postgresql90                @9.0.23         databases/postgresql90
postgresql91                @9.1.22         databases/postgresql91
postgresql92                @9.2.17         databases/postgresql92
postgresql93                @9.3.13         databases/postgresql93
postgresql94                @9.4.8          databases/postgresql94
postgresql95                @9.5.3          databases/postgresql95
postgresql96                @9.6beta2        databases/postgresql96
```

在此示例中，支持的最新PostgreSQL版本是9.6，因此我们将安装该版本。


```
yum list available | grep postgres*
```

Neccesary packages are: postgresqlXX postgresqlXX-server postgresqlXX-libs postgresqlXX-contrib

These are installed with the following command: yum -y install postgresqlXX postgresqlXX-server postgresqlXX-libs postgresqlXX-contrib

Once installed you will need to start the database service as the service owner (Default is postgres). This is done with the pg_ctl command.

```
sudo -su postgres
./usr/pgsql-X.X/bin/pg_ctl -D /var/lib/pgsql/X.X/data start
```

To access the DB in CLI enter `psql`

Debian family

On Debian and derived operating systems, type:

```
sudo apt-get install postgresql
```

This will install the PostgreSQL server package, at the default version offered by the operating system's package repositories.

If the version that's installed by default is not the one that you want, you can use the package manager to search for specific versions which may simultaneously be offered.

You can also use the Yum repository provided by the PostgreSQL project (known as PGDG) to get a different version. This may allow versions not yet offered by operating system package repositories.

Section 1.4: How to install PostgreSQL via MacPorts on OSX

In order to install PostgreSQL on OSX, you need to know which versions are currently supported.

Use this command to see what versions you have available.

```
sudo port list | grep "^postgresql[[:digit:]]\{2\}[[:space:]]"
```

You should get a list that looks something like the following:

```
postgresql80                @8.0.26         databases/postgresql80
postgresql81                @8.1.23         databases/postgresql81
postgresql82                @8.2.23         databases/postgresql82
postgresql83                @8.3.23         databases/postgresql83
postgresql84                @8.4.22         databases/postgresql84
postgresql90                @9.0.23         databases/postgresql90
postgresql91                @9.1.22         databases/postgresql91
postgresql92                @9.2.17         databases/postgresql92
postgresql93                @9.3.13         databases/postgresql93
postgresql94                @9.4.8          databases/postgresql94
postgresql95                @9.5.3          databases/postgresql95
postgresql96                @9.6beta2       databases/postgresql96
```

In this example, the most recent version of PostgreSQL that is supported in 9.6, so we will install that.

```
sudo port install postgresql96-server postgresql96
```

您将看到如下的安装日志：

```
--->  计算 postgresql96-server 的依赖关系
--->  需要安装的依赖项：postgresql96
--->  正在获取 postgresql96 的归档文件
--->  正在尝试从
https://packages.macports.org/postgresql96 获取 postgresql96-9.6beta2_0.darwin_15.x86_64.tbz2
--->  正在尝试从
https://packages.macports.org/postgresql96 获取 postgresql96-9.6beta2_0.darwin_15.x86_64.tbz2.rmd160
--->  正在安装 postgresql96 @9.6beta2_0
--->  正在激活 postgresql96 @9.6beta2_0

要使用 postgresql 服务器，请安装 postgresql96-server 端口

--->  正在清理 postgresql96
--->  正在获取 postgresql96-server 的归档文件
--->  正在尝试从
https://packages.macports.org/postgresql96-server 获取 postgresql96-server-9.6beta2_0.darwin_15.x86_64.tbz2
---> 正在尝试从
https://packages.macports.org/postgresql96-server 获取 postgresql96-server-9.6beta2_0.darwin_15.x86_64.tbz2.rmd160
---> 安装 postgresql96-server @9.6beta2_0
---> 激活 postgresql96-server @9.6beta2_0

安装后要创建数据库实例，请执行
sudo mkdir -p /opt/local/var/db/postgresql96/defaultdb
sudo chown postgres:postgres /opt/local/var/db/postgresql96/defaultdb
sudo su postgres -c '/opt/local/lib/postgresql96/bin/initdb -D
/opt/local/var/db/postgresql96/defaultdb'

---> 清理 postgresql96-server
---> 计算 postgresql96 的依赖关系
--->  正在清理 postgresql96
---> 更新二进制文件数据库
---> 扫描二进制文件的链接错误
---> 未发现损坏的文件。
```

日志中提供了安装剩余步骤的说明，接下来我们将执行这些步骤。

```
sudo mkdir -p /opt/local/var/db/postgresql96/defaultdb
sudo chown postgres:postgres /opt/local/var/db/postgresql96/defaultdb
sudo su postgres -c '/opt/local/lib/postgresql96/bin/initdb -D
/opt/local/var/db/postgresql96/defaultdb'
```

现在我们启动服务器：

```
sudo port load -w postgresql96-server
```

验证我们是否能连接到服务器：

```
su postgres -c psql
```

你将看到来自postgres的提示：

```
psql (9.6.1)
输入 "help" 获取帮助。
```



```
sudo port install postgresql96-server postgresql96
```

You will see an installation log like this:

```
--->  Computing dependencies for postgresql96-server
--->  Dependencies to be installed: postgresql96
--->  Fetching archive for postgresql96
--->  Attempting to fetch postgresql96-9.6beta2_0.darwin_15.x86_64.tbz2 from
https://packages.macports.org/postgresql96
--->  Attempting to fetch postgresql96-9.6beta2_0.darwin_15.x86_64.tbz2.rmd160 from
https://packages.macports.org/postgresql96
--->  Installing postgresql96 @9.6beta2_0
--->  Activating postgresql96 @9.6beta2_0

To use the postgresql server, install the postgresql96-server port

--->  Cleaning postgresql96
--->  Fetching archive for postgresql96-server
--->  Attempting to fetch postgresql96-server-9.6beta2_0.darwin_15.x86_64.tbz2 from
https://packages.macports.org/postgresql96-server
--->  Attempting to fetch postgresql96-server-9.6beta2_0.darwin_15.x86_64.tbz2.rmd160 from
https://packages.macports.org/postgresql96-server
--->  Installing postgresql96-server @9.6beta2_0
--->  Activating postgresql96-server @9.6beta2_0

To create a database instance, after install do
sudo mkdir -p /opt/local/var/db/postgresql96/defaultdb
sudo chown postgres:postgres /opt/local/var/db/postgresql96/defaultdb
sudo su postgres -c '/opt/local/lib/postgresql96/bin/initdb -D
/opt/local/var/db/postgresql96/defaultdb'

--->  Cleaning postgresql96-server
--->  Computing dependencies for postgresql96
--->  Cleaning postgresql96
--->  Updating database of binaries
--->  Scanning binaries for linking errors
--->  No broken files found.
```

The log provides instructions on the rest of the steps for installation, so we do that next.

```
sudo mkdir -p /opt/local/var/db/postgresql96/defaultdb
sudo chown postgres:postgres /opt/local/var/db/postgresql96/defaultdb
sudo su postgres -c '/opt/local/lib/postgresql96/bin/initdb -D
/opt/local/var/db/postgresql96/defaultdb'
```

Now we start the server:

```
sudo port load -w postgresql96-server
```

Verify that we can connect to the server:

```
su postgres -c psql
```

You will see a prompt from postgres:

```
psql (9.6.1)
Type "help" for help.
```

```
postgres=#
```

在这里你可以输入查询以确认服务器正在运行。

```
postgres=#SELECT setting FROM pg_settings WHERE NAME='data_directory';
```

并查看响应：

```
设置
----------------------------------------
/opt/local/var/db/postgresql96/defaultdb
(1 行)
postgres=#
```

输入 \q 退出：

```
postgres=#\q
```

然后你将回到你的 shell 提示符。

恭喜！你现在已经在 macOS 上运行了一个 PostgreSQL 实例。

第1.5节：在Mac上使用brew安装postgresql

Homebrew 自称为“macOS 缺失的包管理器”。它可以用来构建和安装应用程序和库。安装完成后，你可以使用brew命令来安装PostgreSQL及其依赖，方法如下：

```
brew UPDATE
brew install postgresql
```

Homebrew 通常安装最新的稳定版本。如果你需要其他版本，可以使用`brew SEARCH postgresql` 来列出可用版本。如果你需要带有特定选项构建的PostgreSQL，可以使用`brew info postgresql` 来列出支持的选项。如果你需要不支持的构建选项，可能需要自己编译，
但仍然可以使用 Homebrew 来安装常见依赖。

启动服务器：

```
brew services START postgresql
```

打开 PostgreSQL 提示符

```
psql
```

如果 psql 报告没有对应的用户数据库，请运行 CREATEDB。

第1.6节：Mac OSX 的 Postgres.app

一个非常简单的在 Mac 上安装 PostgreSQL 的工具是下载 Postgres.app。
您可以更改偏好设置，使 PostgreSQL 在后台运行或仅在应用程序运行时运行。


```
postgres=#
```

Here you can type a query to see that the server is running.

```
postgres=#SELECT setting FROM pg_settings WHERE NAME='data_directory';
```

And see the response:

```
               setting
----------------------------------------
/opt/local/var/db/postgresql96/defaultdb
(1 row)
postgres=#
```

Type \q to quit:

```
postgres=#\q
```

And you will be back at your shell prompt.

Congratulations! You now have a running PostgreSQL instance on OS/X.

Section 1.5: Install postgresql with brew on Mac

Homebrew calls itself '*the missing package manager for macOS*'. It can be used to build and install applications and libraries. Once installed, you can use the `brew` command to install PostgreSQL and it's dependencies as follows:

```
brew UPDATE
brew install postgresql
```

Homebrew generally installs the latest stable version. If you need a different one then `brew SEARCH postgresql` will list the versions available. If you need PostgreSQL built with particular options then `brew info postgresql` will list which options are supported. If you require an unsupported build option, you may have to do the build yourself, but can still use Homebrew to install the common dependencies.

Start the server:

```
brew services START postgresql
```

Open the PostgreSQL prompt

```
psql
```

If psql complains that there's no corresponding database for your user, run **CREATEDB**.

Section 1.6: Postgres.app for Mac OSX

An extremely simple tool for installing PostgreSQL on a Mac is available by downloading Postgres.app.
You can change preferences to have PostgreSQL run in the background or only when the application is running.

第2章：数据类型

PostgreSQL 为用户提供了丰富的原生数据类型。用户可以使用

CREATE TYPE 命令向 PostgreSQL 添加新类型。

https://www.postgresql.org/docs/9.6/static/datatype.html

第2.1节：数值类型

名称	存储大小	描述	范围
小整数	2 字节	小范围整数	-32768 到 +32767
整数	4 字节	整数的典型选择	-2147483648 到 +2147483647
BIGINT	8 字节	大范围整数	-9223372036854775808 到 +9223372036854775807
DECIMAL	可变	用户指定精度，精确到小数点前最多131072位；最多小数点后16383位	
数字	可变	用户指定精度，精确到小数点前最多131072位；最多小数点后16383位	
实数	4 字节	可变精度，非精确	6位小数精度
双精度 8字节		可变精度，非精确	15位小数精度
smallserial	2 字节	1到32767的小型自增整数	
serial	4 字节	自增整数	1到2147483647
BIGSERIAL	8 字节	大自动递增整数，范围从1到9223372036854775807	
int4range		整数范围	
int8range		大整数范围	
numrange		数值范围	

第2.2节：日期/时间类型

名称	存储大小	描述	低价值	高精度	分辨率
时间戳（无时区）	8 字节	日期和时间（无时区）	公元前4713年	公元294276年	1微秒 / 14位数字
时间戳（含时区）	8 字节	日期和时间，含时区	公元前4713年	公元294276年	1微秒 / 14位数字
日期	4 字节	日期（无具体时间）	公元前4713年	公元5874897年	1天
时间（无时区）	8 字节	时间（无日期）	00:00:00	24:00:00	1微秒 / 14位数字
时间（含时区）	12字节	仅限一天中的时间，带有时区	00:00:00+1459	24:00:00-1459	1微秒 / 14位数字
区间	16 字节	时间间隔	-178000000 年	178000000 年	1 微秒 / 14 数字
tsrange		无时区时间戳范围			
tstzrange		带时区的时间戳范围			
日期范围		日期范围			

Chapter 2: Data Types

PostgreSQL has a rich set of native data types available to users. Users can add new types to PostgreSQL using the CREATE TYPE command.

https://www.postgresql.org/docs/9.6/static/datatype.html

Section 2.1: Numeric Types

Name	Storage Size	Description	Range
SMALLINT	2 bytes	small-range integer	-32768 to +32767
INTEGER	4 bytes	ypical choice for integer	-2147483648 to +2147483647
BIGINT	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
DECIMAL	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
NUMERIC	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
REAL	4 bytes	variable-precision, inexact	6 decimal digits precision
DOUBLE PRECISION	8 bytes	variable-precision, inexact	15 decimal digits precision
smallserial	2 bytes	small autoincrementing integer	1 to 32767
serial	4 bytes	autoincrementing integer	1 to 2147483647
BIGSERIAL	8 bytes	large autoincrementing integer	1 to 9223372036854775807
int4range		Range of integer	
int8range		Range of bigint	
numrange		Range of numeric	

Section 2.2: Date/ Time Types

Name	Storage Size	Description	Low Value	High Value	Resolution
TIMESTAMP (without time zone)	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond / 14 digits
TIMESTAMP (with time zone)	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond / 14 digits
DATE	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
TIME (without time zone)	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond / 14 digits
TIME (with time zone)	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459	1 microsecond / 14 digits
INTERVAL	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond / 14 digits
tsrange		range of timestamp without time zone			
tstzrange		range of timestamp with time zone			
daterange		range of date			

第2.3节：几何类型

名称	存储大小	描述	表示
点	16 字节	平面上的点	(x,y)
线	32 字节	无限直线	{A,B,C}
线段	32 字节	有限线段	((x1,y1),(x2,y2))
盒子	32 字节	矩形盒子	((x1,y1),(x2,y2))
路径	16+16n 字节	闭合路径（类似多边形）	((x1,y1),...)
路径	16+16n 字节	开放路径	[(x1,y1),...]
多边形	40+16n 字节	多边形（类似闭合路径）	((x1,y1),...)
圆	24 字节	圆	<(x,y),r>（中心点和半径）

第2.4节：网络地址类型

名称	存储大小	描述
CIDR	7 或 19 字节	IPv4 和 IPv6 网络
INET	7 或 19 字节	IPv4 和 IPv6 主机及网络
macaddr	6 字节	MAC地址

第2.5节：字符类型

名称	描述
CHARACTER varying(n), varchar(n)	可变长度且有限制
character(n), char(n)	定长，空白填充
TEXT	可变无限长度

第2.6节：数组

在PostgreSQL中，您可以创建任何内置类型、用户定义类型或枚举类型的数组。默认情况下，数组没有长度限制，但您可以指定限制。

声明数组

```
SELECT INTEGER[];
SELECT INTEGER[3];
SELECT INTEGER[][];
SELECT INTEGER[3][3];
SELECT INTEGER ARRAY;
SELECT INTEGER ARRAY[3];
```

创建数组

```
SELECT '{0,1,2}';
SELECT '{{0,1},{1,2}}';
SELECT ARRAY[0,1,2];
SELECT ARRAY[ARRAY[0,1],ARRAY[1,2]];
```

访问数组

默认情况下，PostgreSQL 对数组使用从1开始的编号方式，也就是说，包含 n 个元素的数组从 ARRAY[1] 开始，到 ARRAY[n] 结束。

```
--访问特定元素
```


Section 2.3: Geometric Types

Name	Storage Size	Description	Representation
point	16 bytes	Point on a plane	(x,y)
line	32 bytes	Infinite line	{A,B,C}
lseg	32 bytes	Finite line segment	((x1,y1),(x2,y2))
BOX	32 bytes	Rectangular box	((x1,y1),(x2,y2))
path	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
path	16+16n bytes	Open path	[(x1,y1),...]
polygon	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
CIRCLE	24 bytes	Circle	<(x,y),r> (center point and radius)

Section 2.4: Network Adress Types

Name	Storage Size	Description
CIDR	7 or 19 bytes	IPv4 and IPv6 networks
INET	7 or 19 bytes	IPv4 and IPv6 hosts and networks
macaddr	6 bytes	MAC addresses

Section 2.5: Character Types

Name	Description
CHARACTER varying(n), varchar(n)	variable-length with limit
character(n), char(n)	fixed-length, blank padded
TEXT	variable unlimited length

Section 2.6: Arrays

In PostgreSQL you can create Arrays of any built-in, user-defined or enum type. In default there is no limit to an Array, but you *can* specify it.

Declaring an Array

```
SELECT INTEGER[];
SELECT INTEGER[3];
SELECT INTEGER[][];
SELECT INTEGER[3][3];
SELECT INTEGER ARRAY;
SELECT INTEGER ARRAY[3];
```

Creating an Array

```
SELECT '{0,1,2}';
SELECT '{{0,1},{1,2}}';
SELECT ARRAY[0,1,2];
SELECT ARRAY[ARRAY[0,1],ARRAY[1,2]];
```

Accessing an Array

By default PostgreSQL uses a one-based numbering convention for arrays, that is, an array of n elements starts with ARRAY[1] and ends with ARRAY[n].

```
--accesing a spefific element
```

```
WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT int_arr[1] FROM arr;

int_arr
---------
       0
(1 行)

--切片数组
WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT int_arr[1:2] FROM arr;

int_arr
---------
    {0,1}
(1 ROW)
```

获取数组信息

```
--数组维度（文本形式）
WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT ARRAY_DIMS(int_arr) FROM arr;

array_dims
------------
      [1:3]
(1 行)

--数组维度的长度
 WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT ARRAY_LENGTH(int_arr,1) FROM arr;

 array_length
 --------------
            3
 (1 行)

--所有维度的元素总数
 WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT cardinality(int_arr) FROM arr;

基数
 -------------
           3
(1 行)
```

数组函数

> 将被添加


```
WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT int_arr[1] FROM arr;

int_arr
---------
       0
(1 ROW)

--sclicing an array
WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT int_arr[1:2] FROM arr;

int_arr
---------
    {0,1}
(1 ROW)
```

Getting information about an array

```
--array dimensions (as text)
WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT ARRAY_DIMS(int_arr) FROM arr;

array_dims
------------
      [1:3]
(1 ROW)

--length of an array dimension
 WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT ARRAY_LENGTH(int_arr,1) FROM arr;

 array_length
 --------------
            3
 (1 ROW)

--total number of elements across all dimensions
 WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT cardinality(int_arr) FROM arr;

 cardinality
 -------------
           3
(1 ROW)
```

Array functions

> will be added

第3章：日期、时间戳和时间间隔

第3.1节：选择当月最后一天

您可以选择当月的最后一天。

```
SELECT (DATE_TRUNC('MONTH', ('201608'||'01')::DATE) + INTERVAL '1 MONTH - 1 day')::DATE;
```

201608可以用变量替代。

第3.2节：将时间戳或间隔转换为字符串

您可以使用TO_CHAR()函数将TIMESTAMP或INTERVAL值转换为字符串：

```
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP, 'DD Mon YYYY HH:MI:SSPM');
```

该语句将生成字符串 "12 Aug 2016 04:40:32PM"。格式化字符串可以以多种不同方式修改；完整的模板模式列表可以在这里找到。

请注意，您也可以在格式化字符串中插入纯文本，并且可以以任意顺序使用模板模式：

```
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP,
               '"Today is "FMDay", the "DDth" day of the month of "FMMonth" of "YYYY');
```

这将生成字符串 "Today is Saturday, the 12th day of the month of August of 2016"。不过您应当记住，任何模板模式——即使是像 "I"、"D"、"W" 这样单个字母的——都会被转换，除非纯文本被双引号括起来。作为安全措施，您应当像上面那样将所有纯文本放在双引号中。

您可以通过使用 TM（翻译模式）修饰符将字符串本地化为您选择的语言（日和月的名称）。此选项使用运行PostgreSQL的服务器或连接到它的客户端的本地化设置。

```
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP, 'TMDay, DD" de "TMMonth" del año "YYYY');
```

在西班牙语区域设置下，这将生成 "Sábado, 12 de Agosto del año 2016"。

第3.3节：统计每周的记录数

```
SELECT DATE_TRUNC('week', <>) AS "Week" , COUNT(*)
FROM <>
GROUP  BY 1
ORDER BY 1;
```


Chapter 3: Dates, Timestamps, and Intervals

Section 3.1: SELECT the last day of month

You can select the last day of month.

```
SELECT (DATE_TRUNC('MONTH', ('201608'||'01')::DATE) + INTERVAL '1 MONTH - 1 day')::DATE;
```

201608 is replaceable with a variable.

Section 3.2: Cast a timestamp or interval to a string

You can convert a `TIMESTAMP` or `INTERVAL` value to a string with the `TO_CHAR()` function:

```
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP, 'DD Mon YYYY HH:MI:SSPM');
```

This statement will produce the string "12 Aug 2016 04:40:32PM". The formatting string can be modified in many different ways; the full list of template patterns can be found here.

Note that you can also insert plain text into the formatting string and you can use the template patterns in any order:

```
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP,
               '"Today is "FMDay", the "DDth" day of the month of "FMMonth" of "YYYY');
```

This will produce the string "Today is Saturday, the 12th day of the month of August of 2016". You should keep in mind, though, that any template patterns - even the single letter ones like "I", "D", "W" - are converted, unless the plain text is in double quotes. As a safety measure, you should put all plain text in double quotes, as done above.

You can localize the string to your language of choice (day and month names) by using the TM (translation mode) modifier. This option uses the localization setting of the server running PostgreSQL or the client connecting to it.

```
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP, 'TMDay, DD" de "TMMonth" del año "YYYY');
```

With a Spanish locale setting this produces "Sábado, 12 de Agosto del año 2016".

Section 3.3: Count the number of records per week

```
SELECT DATE_TRUNC('week', <>) AS "Week" , COUNT(*)
FROM <>
GROUP BY 1
ORDER BY 1;
```

第4章：表的创建

第4.1节：显示表定义

打开连接到包含你的表的数据库的 `psql` 命令行工具。然后输入以下命令：

```
\d tablename
```

要获取扩展信息，请输入

```
\d+ 表名
```

如果你忘记了表的名称，只需在 psql 中输入 \d 即可获得当前数据库中表和视图的列表。

第4.2节：从选择创建表

假设你有一个名为 person 的表：

```
CREATE TABLE person (
    person_id BIGINT NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
age INT NOT NULL,
    PRIMARY KEY (person_id)
);
```

你可以这样创建一个年龄超过30岁的新表：

```
CREATE TABLE people_over_30 AS SELECT * FROM person WHERE age > 30;
```

第4.3节：创建非日志表

你可以创建非日志表，这样可以使表的操作速度大大加快。非日志表跳过写入WRITE-ahead log，这意味着它不具备崩溃安全性且无法复制。

```
CREATE UNLOGGED TABLE person (
    person_id BIGINT NOT NULL PRIMARY KEY,
    last_name VARCHAR(255) NOT NULL,
first_name VARCHAR(255),
    address VARCHAR(255),
    city VARCHAR(255)
);
```

第4.4节：带主键的表创建

```
CREATE TABLE person (
    person_id BIGINT NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
address VARCHAR(255),
    city VARCHAR(255),
    主键 (person_id)
```


Chapter 4: Table Creation

Section 4.1: Show table definition

Open the `psql` command line tool connected to the database where your table is. Then type the following command:

```
\d tablename
```

To get extended information type

```
\d+ tablename
```

If you have forgotten the name of the table, just type \d into psql to obtain a list of tables and views in the current database.

Section 4.2: Create table from select

Let's say you have a table called person:

```
CREATE TABLE person (
    person_id BIGINT NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    age INT NOT NULL,
    PRIMARY KEY (person_id)
);
```

You can create a new table of people over 30 like this:

```
CREATE TABLE people_over_30 AS SELECT * FROM person WHERE age > 30;
```

Section 4.3: Create unlogged table

You can create unlogged tables so that you can make the tables considerably faster. Unlogged table skips writing **WRITE**-ahead log which means it's not crash-safe and unable to replicate.

```
CREATE UNLOGGED TABLE person (
    person_id BIGINT NOT NULL PRIMARY KEY,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    address VARCHAR(255),
    city VARCHAR(255)
);
```

Section 4.4: Table creation with Primary Key

```
CREATE TABLE person (
    person_id BIGINT NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    address VARCHAR(255),
    city VARCHAR(255),
    PRIMARY KEY (person_id)
```

```
);
```

或者，您可以将主键约束直接放在列定义中：

```
CREATE TABLE person (
    person_id BIGINT NOT NULL PRIMARY KEY,
    last_name VARCHAR(255) NOT NULL,
first_name VARCHAR(255),
    address VARCHAR(255),
    city VARCHAR(255)
);
```

建议您对表名以及所有列名使用小写字母。如果使用大写字母名称，如Person，则每次查询时都必须用双引号（"Person"）将该名称括起来，因为PostgreSQL会强制大小写折叠。

第4.5节：创建引用其他表的表

在此示例中，用户表将包含一个引用机构表的列。

```
创建表 agencies ( -- 首先创建机构表
  id SERIAL 主键,
  名称 文本 非空
)

创建表 users (
id SERIAL 主键,
agency_id 非空 整数 引用 agencies(id) 可延迟约束 初始延迟 -- 这是引用你的机构表。

)
```



```
);
```

Alternatively, you can place the **PRIMARY KEY** constraint directly in the column definition:

```
CREATE TABLE person (
    person_id BIGINT NOT NULL PRIMARY KEY,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    address VARCHAR(255),
    city VARCHAR(255)
);
```

It is recommended that you use lower case names for the table and as well as all the columns. If you use upper case names such as `Person` you would have to wrap that name in double quotes (`"Person"`) in each and every query because PostgreSQL enforces case folding.

Section 4.5: Create a table that references other table

In this example, User Table will have a column that references the Agency table.

```
CREATE TABLE agencies ( -- first create the agency table
  id SERIAL PRIMARY KEY,
  NAME TEXT NOT NULL
)

CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  agency_id NOT NULL INTEGER REFERENCES agencies(id) DEFERRABLE INITIALLY DEFERRED -- this is going
to references your agency table.
)
```

第5章：SELECT

第5.1节：使用WHERE的SELECT

本节我们将基于以下用户表：

```
创建表 sch_test.user_table
(
id serial NOT NULL,
username CHARACTER VARYING,
  pass CHARACTER VARYING,
  first_name CHARACTER varying(30),
  last_name CHARACTER varying(30),
  CONSTRAINT user_table_pkey PRIMARY KEY (id)
)
```

```
+----+------------+-----------+----------+------+
| id | first_name | last_name | username | pass |
+----+------------+-----------+----------+------+
| 1  | hello      | world     | hello    | word |
+----+------------+-----------+----------+------+
| 2  | root       | me        | root     | toor |
+----+------------+-----------+----------+------+
```

语法

选择所有内容：

```
SELECT * FROM schema_name.table_name WHERE <condition>;
```

选择部分字段：

```
SELECT field1, field2 FROM schema_name.table_name WHERE <condition>;
```

示例

```
-- 选择所有内容，条件是 id = 1
SELECT * FROM schema_name.table_name WHERE id = 1;

-- SELECT id where username = ? and pass = ?
SELECT id FROM schema_name.table_name WHERE username = 'root' AND pass = 'toor';

-- 选择 id 不等于 1 的 first_name
SELECT first_name FROM schema_name.table_name WHERE id != 1;
```


Chapter 5: SELECT

Section 5.1: SELECT using WHERE

In this topic we will base on this table of users :

```
CREATE TABLE sch_test.user_table
(
  id serial NOT NULL,
  username CHARACTER VARYING,
  pass CHARACTER VARYING,
  first_name CHARACTER varying(30),
  last_name CHARACTER varying(30),
  CONSTRAINT user_table_pkey PRIMARY KEY (id)
)
```

```
+----+------------+-----------+----------+------+
| id | first_name | last_name | username | pass |
+----+------------+-----------+----------+------+
| 1  | hello      | world     | hello    | word |
+----+------------+-----------+----------+------+
| 2  | root       | me        | root     | toor |
+----+------------+-----------+----------+------+
```

Syntax

Select every thing:

```
SELECT * FROM schema_name.table_name WHERE <condition>;
```

Select some fields :

```
SELECT field1, field2 FROM schema_name.table_name WHERE <condition>;
```

Examples

```
-- SELECT every thing where id = 1
SELECT * FROM schema_name.table_name WHERE id = 1;

-- SELECT id where username = ? and pass = ?
SELECT id FROM schema_name.table_name WHERE username = 'root' AND pass = 'toor';

-- SELECT first_name where id not equal 1
SELECT first_name FROM schema_name.table_name WHERE id != 1;
```

第6章：查找字符串长度 / 字符长度

要获取“character varying”、“text”字段的长度，使用 char_length() 或 character_length()。

第6.1节：获取 character varying 字段长度的示例

示例1，查询：`SELECT CHAR_LENGTH('ABCDE')`

结果：

```
5
```

示例2，查询：`SELECT CHARACTER_LENGTH('ABCDE')`

结果：

```
5
```


Chapter 6: Find String Length / Character Length

To get length of "character varying", "text" fields, Use char_length() or character_length().

Section 6.1: Example to get length of a character varying field

Example 1, Query: `SELECT CHAR_LENGTH('ABCDE')`

Result:

```
5
```

Example 2, Query: `SELECT CHARACTER_LENGTH('ABCDE')`

Result:

```
5
```

第7章：COALESCE 函数

Coalesce 返回一组参数中第一个非空（non null）参数。只返回第一个非空参数，后续参数将被忽略。如果所有参数均为空，则函数返回 null。

第7.1节：单个非空参数

```
PGSQL> SELECT COALESCE(NULL, NULL, 'HELLO WORLD');

COALESCE
--------
'HELLO WORLD'
```

第7.2节：多个非空参数

```
PGSQL> SELECT COALESCE(NULL, NULL, '第一个非空', NULL, NULL, '第二个非空');

coalesce
--------
'第一个非空'
```

第7.3节：全部为空参数

```
PGSQL> SELECT COALESCE(NULL, NULL, NULL);

COALESCE
--------
```


Chapter 7: COALESCE

Coalesce returns the first none null argument from a set of arguments. Only the first non null argument is return, all subsequent arguments are ignored. The function will evaluate to null if all arguments are null.

Section 7.1: Single non null argument

```
PGSQL> SELECT COALESCE(NULL, NULL, 'HELLO WORLD');

COALESCE
--------
'HELLO WORLD'
```

Section 7.2: Multiple non null arguments

```
PGSQL> SELECT COALESCE(NULL, NULL, 'first non null', NULL, NULL, 'second non null');

coalesce
--------
'first non null'
```

Section 7.3: All null arguments

```
PGSQL> SELECT COALESCE(NULL, NULL, NULL);

COALESCE
--------
```

第8章：插入（INSERT）

第8.1节：使用COPY插入数据

COPY 是 PostgreSQL 的批量插入机制。这是一种在文件和表之间传输数据的便捷方式，但当一次添加几千行以上时，它的速度远快于INSERT。

让我们先创建一个示例数据文件。

```
cat > samplet_data.csv

1,Yogesh
2,Raunak
3,Varun
4,Kamal
5,Hari
6,Amit
```

我们需要一个两列的表来导入这些数据。

```
CREATE TABLE copy_test(id INT, NAME varchar(8));
```

现在进行实际的复制操作，这将在表中创建六条记录。

```
COPY copy_test FROM '/path/to/file/sample_data.csv' DELIMITER ',';
```

可以不使用磁盘上的文件，而是从STDIN插入数据

```
COPY copy_test FROM STDIN DELIMITER ',';
输入要复制的数据，后跟换行符。
以反斜杠和句号单独位于一行结尾。
>> 7,Amol
>> 8,Amar
>> \.
时间: 85254.306 毫秒

选择 * 从 copy_test ;

编号 |  名字
----+--------
1 | Yogesh
3 | Varun
5 | Hari
7 | Amol
2 | Raunak
4 | Kamal
6 | Amit
8 | Amar
```

你也可以像下面这样将数据从表复制到文件：

```
COPY copy_test 到 'path/to/file/sample_data.csv'  分隔符 ',';
```

有关COPY的更多详情你可以查看 here

Chapter 8: INSERT

Section 8.1: Insert data using COPY

COPY is PostgreSQL's bulk-insert mechanism. It's a convenient way to transfer data between files and tables, but it's also far faster than **INSERT** when adding more than a few thousand rows at a time.

Let's begin by creating sample data file.

```
cat > samplet_data.csv

1,Yogesh
2,Raunak
3,Varun
4,Kamal
5,Hari
6,Amit
```

And we need a two column table into which this data can be imported into.

```
CREATE TABLE copy_test(id INT, NAME varchar(8));
```

Now the actual copy operation, this will create six records in the table.

```
COPY copy_test FROM '/path/to/file/sample_data.csv' DELIMITER ',';
```

Instead of using a file on disk, can insert data from **STDIN**

```
COPY copy_test FROM STDIN DELIMITER ',';
Enter DATA TO be copied followed BY a newline.
END WITH a backslash AND a period ON a line BY itself.
>> 7,Amol
>> 8,Amar
>> \.
TIME: 85254.306 ms

SELECT * FROM copy_test ;

id |  name
----+--------
1 | Yogesh
3 | Varun
5 | Hari
7 | Amol
2 | Raunak
4 | Kamal
6 | Amit
8 | Amar
```

Also you can copy data from a table to file as below:

```
COPY copy_test TO 'path/to/file/sample_data.csv'  DELIMITER ',';
```

For more details on COPY you can check here

第8.2节：插入多行

你可以同时向数据库插入多行数据：

```
插入到 person (姓名, 年龄) 值
  ('john doe', 25),
  ('简·多', 20);
```

第8.3节：插入数据和返回值

如果你向一个带有自增列的表中插入数据，并且想要获取自增列的值。

假设你有一个名为my_table的表：

```
CREATE TABLE my_table
(
id serial NOT NULL, -- serial数据类型是自动递增的四字节整数
NAME CHARACTER VARYING,
contact_number INTEGER,
CONSTRAINT my_table_pkey PRIMARY KEY (id)
);
```

如果你想向my_table插入数据并获取该行的id：

```
INSERT INTO my_table(NAME, contact_number) VALUES ( 'USER', 8542621) RETURNING id;
```

上述查询将返回新记录插入行的id。

第8.4节：基本插入

假设我们有一个名为 person 的简单表：

```
CREATE TABLE person (
    person_id BIGINT,
    NAME VARCHAR(255).
age INT,
    city VARCHAR(255)
);
```

最基本的插入操作是向表中插入所有值：

```
INSERT INTO person VALUES (1, 'john doe', 25, 'new york');
```

如果只想插入特定的列，则需要明确指出哪些列：

```
INSERT INTO person (NAME, age) VALUES ('john doe', 25);
```

请注意，如果表中存在任何约束，例如 NOT NULL，则无论哪种情况都必须包含这些列。

第8.5节：从 select 插入

你可以将 select 语句的结果插入到表中：

Section 8.2: Inserting multiple rows

You can insert multiple rows in the database at the same time:

```
INSERT INTO person (NAME, age) VALUES
  ('john doe', 25),
  ('jane doe', 20);
```

Section 8.3: INSERT data and RETURING values

If you are inserting data into a table with an auto increment column and if you want to get the value of the auto increment column.

Say you have a table called `my_table`:

```
CREATE TABLE my_table
(
id serial NOT NULL, -- serial data type is auto incrementing four-byte integer
NAME CHARACTER VARYING,
contact_number INTEGER,
CONSTRAINT my_table_pkey PRIMARY KEY (id)
);
```

If you want to insert data into `my_table` and get the id of that row:

```
INSERT INTO my_table(NAME, contact_number) VALUES ( 'USER', 8542621) RETURNING id;
```

Above query will return the id of the row where the new record was inserted.

Section 8.4: Basic INSERT

Let's say we have a simple table called person:

```
CREATE TABLE person (
    person_id BIGINT,
    NAME VARCHAR(255).
    age INT,
    city VARCHAR(255)
);
```

The most basic insert involves inserting all values in the table:

```
INSERT INTO person VALUES (1, 'john doe', 25, 'new york');
```

If you want to insert only specific columns, you need to explicitly indicate which columns:

```
INSERT INTO person (NAME, age) VALUES ('john doe', 25);
```

Note that if any constraints exist on the table , such as NOT NULL, you will be required to include those columns in either case.

Section 8.5: Insert from select

You can insert data in a table as the result of a select statement:

```
INSERT INTO person SELECT * FROM tmp_person WHERE age < 30;
```

请注意，select 的投影必须与 insert 所需的列匹配。在本例中，tmp_person表与person表具有相同的列。

第8.6节：UPSERT - INSERT ... ON CONFLICT DO UPDATE..

自版本9.5起，PostgreSQL 提供了带有INSERT语句的UPSERT功能。

假设你有一个名为 my_table 的表，在之前的几个示例中创建。我们插入一行，返回插入行的主键值：

```
b=# INSERT INTO my_table (name,contact_number) values ('one',333) RETURNING id;
id
----
2
(1 行)

INSERT 0 1
```

现在如果我们尝试插入具有已存在唯一键的行，将会引发异常：

```
b=# INSERT INTO my_table VALUES (2,'one',333);
错误：重复的KEY VALUE违反了UNIQUE CONSTRAINTmy_table_pkey"
详细信息：KEY(id)=(2)已存在。
```

Upsert 功能提供了无论如何插入的能力，解决冲突：

```
b=# INSERT INTO my_table values (2,'one',333) ON CONFLICT (id) DO UPDATE SET name =
my_table.name||' changed to: "two" at '||now() returning *;
id |                        name                                 | contact_number
----+-------------------------------------------------------------------------------------------------------
--------------+----------------
2 | one changed to: "two" at 2016-11-23 08:32:17.105179+00 |             333
(1 行)

INSERT 0 1
```

第8.7节：将SELECT数据导入文件

您可以COPY表并将其粘贴到文件中。

```
postgres=# select * from my_table;
c1 | c2 | c3
----+----+----
1 |  1 |  1
2 |  2 |  2
3 |  3 |  3
4 |  4 |  4
5 |  5 |
(5 行)

postgres=# copy my_table to '/home/postgres/my_table.txt' using delimiters '|' with null as
'null_string' csv header;
COPY 5
```



```
INSERT INTO person SELECT * FROM tmp_person WHERE age < 30;
```

Note that the projection of the select must match the columns required for the insert. In this case, the `tmp_person` table has the same columns as `person`.

Section 8.6: UPSERT - INSERT ... ON CONFLICT DO UPDATE..

since version 9.5 postgres offers UPSERT functionality with **`INSERT`** statement.

Say you have a table called my_table, created in several previous examples. We insert a row, returning PK value of inserted row:

```
b=# INSERT INTO my_table (name,contact_number) values ('one',333) RETURNING id;
id
----
2
(1 row)

INSERT 0 1
```

Now if we try to insert row with existing unique key it will raise an exception:

```
b=# INSERT INTO my_table VALUES (2,'one',333);
ERROR:  duplicate KEY VALUE violates UNIQUE CONSTRAINT "my_table_pkey"
DETAIL:  KEY (id)=(2) already EXISTS.
```

Upsert functionality offers ability to insert it anyway, solving the conflict:

```
b=# INSERT INTO my_table values (2,'one',333) ON CONFLICT (id) DO UPDATE SET name =
my_table.name||' changed to: "two" at '||now() returning *;
id |                        name                                 | contact_number
----+-------------------------------------------------------------------------------------------------------
--------------+----------------
2 | one changed to: "two" at 2016-11-23 08:32:17.105179+00 |             333
(1 row)

INSERT 0 1
```

Section 8.7: SELECT data into file

You can COPY table and paste it into a file.

```
postgres=# select * from my_table;
c1 | c2 | c3
----+----+----
1 |  1 |  1
2 |  2 |  2
3 |  3 |  3
4 |  4 |  4
5 |  5 |
(5 rows)

postgres=# copy my_table to '/home/postgres/my_table.txt' using delimiters '|' with null as
'null_string' csv header;
COPY 5
```

```
postgres=# \! cat my_table.txt
c1|c2|c3
1|1|1
2|2|2
3|3|3
4|4|4
5|5|null_string
```



```
postgres=# \! cat my_table.txt
c1|c2|c3
1|1|1
2|2|2
3|3|3
4|4|4
5|5|null_string
```

第9章：更新

第9.1节：基于连接另一张表更新表

你也可以基于另一张表的数据来更新表中的数据：

```
UPDATE person
SET state_code = cities.state_code
FROM cities
WHERE cities.city = city;
```

这里我们将person表的city列与cities表的city列连接，以获取该城市的州代码。然后用该代码更新person表中的state_code列。

第9.2节：更新表中的所有行

您只需提供一个列名= 值即可更新表中的所有行：

```
UPDATE person SET planet = '地球';
```

第9.3节：更新满足条件的所有行

```
UPDATE person SET state = '纽约州' WHERE city = '纽约市';
```

第9.4节：更新表中的多个列

您可以在同一条语句中更新表中的多个列，使用逗号分隔col=val对：

```
UPDATE person
   SET country = '美国',
       state = '纽约州'
WHERE city = '纽约市';
```


Chapter 9: UPDATE

Section 9.1: Updating a table based on joining another table

You can also update data in a table based on data from another table:

```
UPDATE person
SET state_code = cities.state_code
FROM cities
WHERE cities.city = city;
```

Here we are joining the `person` `city` column to the `cities` `city` column in order to get the city's state code. This is then used to update the `state_code` column in the `person` table.

Section 9.2: Update all rows in a table

You update all rows in table by simply providing a `column_name = VALUE`:

```
UPDATE person SET planet = 'Earth';
```

Section 9.3: Update all rows meeting a condition

```
UPDATE person SET state = 'NY' WHERE city = 'New York';
```

Section 9.4: Updating multiple columns in table

You can update multiple columns in a table in the same statement, separating `col=val` pairs with commas:

```
UPDATE person
   SET country = 'USA',
       state = 'NY'
WHERE city = 'New York';
```

第10章：JSON支持

JSON - Java Script对象表示法，Postgresql自9.2版本起支持JSON数据类型。提供了一些预定义的函数和操作符来访问JSON数据。操作符->返回JSON列的键。操作符->>返回JSON列的值。

第10.1节：使用JSONb操作符

创建数据库和表

```
如果存在则删除数据库 books_db;创建数据
库 books_db 编码='UTF8' 模板 template0;



创建表 books (
  id 序列 主键,
  客户 文本 非空,
  数据 JSONb 非空
);
```

填充数据库

```
插入到 books(客户, 数据) 值 (
    '乔',
    '{ "标题": "悉达多", "作者": { "名": "赫尔曼", "姓": "黑塞" } }'
),(
    '珍妮',
    '{ "标题": "达摩流浪者", "作者": { "名": "杰克", "姓": "凯鲁亚克" } }'
),(
    '珍妮',
    '{ "标题": "百年孤独", "作者": { "名": "加博", "姓": "马尔克斯" }
}'
);
```

查看表 books 中的所有内容：

```
选择 * 从 books;
```

输出：

id integer	client character varying	data jsonb
1	Joe	{"title": "Siddhartha", "author": {"last name": "Hesse", "first name": "Herman"}}
2	Jenny	{"title": "Dharma Bums", "author": {"last name": "Kerouac", "first name": "Jack"}}
3	Jenny	{"title": "100 años de soledad", "author": {"last name": "Marquéz", "first name": "Gabo"}}

- >   运算符返回 JSON 列中的值

选择 1 列：

```
SELECT&amp;nbsp;&amp;nbsp;client,

    DATA&amp;nbsp;&amp;nbsp;-&amp;nbsp;&amp;nbsp;>&amp;nbsp;&amp;nbsp;'title'&amp;nbsp;&amp;nbsp;AS&amp;nbsp;&amp;nbsp;title

    FROM&amp;nbsp;&amp;nbsp;books;
```

输出：

Chapter 10: JSON Support

JSON - Java Script Object Notation , Postgresql support JSON Data type since 9.2 version. There are some predefined function and operators to access the JSON data. The `->` operator returns the key of JSON column. The `->>` operator returns the value of JSON Column.

Section 10.1: Using JSONb operators

Creating a DB and a Table

```
DROP DATABASE IF EXISTS books_db;
CREATE DATABASE books_db WITH ENCODING='UTF8' TEMPLATE template0;

DROP TABLE IF EXISTS books;

CREATE TABLE books (
  id SERIAL PRIMARY KEY,
  client TEXT NOT NULL,
  DATA JSONb NOT NULL
);
```

Populating the DB

```
INSERT INTO books(client, DATA) VALUES (
    'Joe',
    '{ "title": "Siddhartha", "author": { "first_name": "Herman", "last_name": "Hesse" } }'
),(
    'Jenny',
    '{ "title": "Dharma Bums", "author": { "first_name": "Jack", "last_name": "Kerouac" } }'
),(
    'Jenny',
    '{ "title": "100 años de soledad", "author": { "first_name": "Gabo", "last_name": "Marquéz" }
}'
);
```

Lets see everything inside the table books:

```
SELECT * FROM books;
```

Output:

id integer	client character varying	data jsonb
1	Joe	{"title": "Siddhartha", "author": {"last name": "Hesse", "first name": "Herman"}}
2	Jenny	{"title": "Dharma Bums", "author": {"last name": "Kerouac", "first name": "Jack"}}
3	Jenny	{"title": "100 años de soledad", "author": {"last name": "Marquéz", "first name": "Gabo"}}

`->` operator returns values out of JSON columns

Selecting 1 column:

```
SELECT client,
    DATA->'title' AS title
    FROM books;
```

Output:

	client character varying	title jsonb
1	Joe	"Siddhartha"
2	Jenny	"Dharma Bums"
3	Jenny	"100 años de soledad"

选择 2 列：

```
SELECT  client,
    DATA  -  'title' AS title, DATA  -  'author' AS author
    FROM books;
```

输出：

client character varying	title jsonb	author jsonb
Joe	"Siddhartha"	{"last_name": "Hesse", "first_name": "Herman"}
Jenny	"Dharma Bums"	{"last name": "Kerouac", "first name": "Jack"}
Jenny	"100 años de soledad"	{"last name": "Marquéz", "first name": "Gabo"}

- > 运算符返回原始 JSON 类型（可能是对象），而 - >> 返回文本。

返回嵌套对象

你可以使用 -> 来返回嵌套对象，从而链式调用操作符：

```
SELECT  client,
   DATA->'author'->'last_name' AS author
   FROM books;
```

输出：

client character varying	author jsonb
Joe	"Hesse"
Jenny	"Kerouac"
Jenny	"Marquéz"

过滤

根据 JSON 中的值选择行：

```
SELECT
client,
DATA->'title' AS title
FROM books
 WHERE DATA->'title' = '"Dharma Bums"';
```

注意 WHERE 使用了 `->` ，所以我们必须与 JSON `'"Dharma Bums"'`

进行比较，或者我们可以使用 `->>` 并与 `'Dharma Bums'` 比较

输出：

	client character varying	title jsonb
1	Joe	"Siddhartha"
2	Jenny	"Dharma Bums"
3	Jenny	"100 años de soledad"

Selecting 2 columns:

```
SELECT client,
   DATA->'title' AS title, DATA->'author' AS author
   FROM books;
```

Output:

client character varying	title jsonb	author jsonb
Joe	"Siddhartha"	{"last_name": "Hesse", "first_name": "Herman"}
Jenny	"Dharma Bums"	{"last name": "Kerouac", "first name": "Jack"}
Jenny	"100 años de soledad"	{"last name": "Marquéz", "first name": "Gabo"}

`->` vs `->>`

The `->` operator returns the original JSON type (which might be an object), whereas `->>` returns text.

Return NESTED objects

You can use the `->` to return a nested object and thus chain the operators:

```
SELECT client,
   DATA->'author'->'last_name' AS author
   FROM books;
```

Output:

client character varying	author jsonb
Joe	"Hesse"
Jenny	"Kerouac"
Jenny	"Marquéz"

Filtering

Select rows based on a value inside your JSON:

```
SELECT
client,
DATA->'title' AS title
FROM books
 WHERE DATA->'title' = '"Dharma Bums"';
```

Notice WHERE uses `->` so we must compare to JSON `'"Dharma Bums"'`

Or we could use `->>` and compare to `'Dharma Bums'`

Output:

client character varying	title jsonb
Jenny	"Dharma Bums"

嵌套过滤

根据嵌套 JSON 对象的值查找行：

```
SELECT
 client,
 DATA->'title' AS title
 FROM books
  WHERE DATA->'author'->>'last_name' = 'Kerouac';
```

输出：

client character varying	title jsonb
Jenny	"Dharma Bums"

一个真实的示例

```
创建表 events (
   NAME varchar(200),
  visitor_id varchar(200),
   properties json,
browser json
);
```

我们将把事件存储在此表中，比如页面浏览。每个事件都有属性，这些属性可以是任何内容（例如当前页面），并且还发送有关浏览器的信息（如操作系统、屏幕分辨率等）。这两者都是完全自由格式的，且可能随时间变化（随着我们想到要跟踪的额外内容）。

```
插入到 events (NAME, visitor_id, properties, browser) 值
(
  'pageview', '1',
  '{ "page": "/" }',
  '{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }'
),(
  'pageview', '2',
  '{ "page": "/" }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1920, "y": 1200 } }'
),(
  'pageview', '1',
  '{ "page": "/account" }',
  '{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }'
),(
  'purchase', '5',
  '{ "amount": 10 }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1024, "y": 768 } }'
),(
  'purchase', '15',
  '{ "amount": 200 }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }'
),(
  'purchase', '15',
  '{ "amount": 500 }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }'
) ;
```


client character varying	title jsonb
Jenny	"Dharma Bums"

Nested filtering

Find rows based on the value of a nested JSON object:

```
SELECT
 client,
 DATA->'title' AS title
 FROM books
  WHERE DATA->'author'->>'last_name' = 'Kerouac';
```

Output:

client character varying	title jsonb
Jenny	"Dharma Bums"

A real world example

```
CREATE TABLE events (
  NAME varchar(200),
  visitor_id varchar(200),
  properties json,
  browser json
);
```

We're going to store events in this table, like pageviews. Each event has properties, which could be anything (e.g. current page) and also sends information about the browser (like OS, screen resolution, etc). Both of these are completely free form and could change over time (as we think of extra stuff to track).

```
INSERT INTO events (NAME, visitor_id, properties, browser) VALUES
(
  'pageview', '1',
  '{ "page": "/" }',
  '{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }'
),(
  'pageview', '2',
  '{ "page": "/" }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1920, "y": 1200 } }'
),(
  'pageview', '1',
  '{ "page": "/account" }',
  '{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }'
),(
  'purchase', '5',
  '{ "amount": 10 }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1024, "y": 768 } }'
),(
  'purchase', '15',
  '{ "amount": 200 }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }'
),(
  'purchase', '15',
  '{ "amount": 500 }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }'
);
```

现在让我们选择所有内容：

```
SELECT * FROM events;
```

输出：

name character varying(200)	visitor_id character varying(200)	properties json	browser json
pageview	1	{ "page": "/" }	{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }
pageview	2	{ "page": "/" }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1920, "y": 1200 } }
pageview	1	{ "page": "/account" }	{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }
purchase	5	{ "amount": 10 }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1024, "y": 768 } }
purchase	15	{ "amount": 200 }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }
purchase	15	{ "amount": 500 }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }

JSON 操作符 + PostgreSQL 聚合函数

使用 JSON 操作符，结合传统的 PostgreSQL 聚合函数，我们可以提取任何我们想要的内容。你可以充分利用关系数据库管理系统的全部功能。

- 让我们看看浏览器的使用情况：

```
SELECT browser->>'name' AS browser,
  COUNT(browser)
  FROM events
  按浏览器分组>>'name';
```

输出：

browser text	count bigint
Firefox	4
Chrome	2

- 每位访客的总收入：

```
选择 visitor_id, SUM(CAST(properties->>'amount' AS INTEGER)) 作为 total
来自 events
条件 CAST(properties->>'amount' AS INTEGER) > 0
按 visitor_id分组;
```

输出：

visitor_id character varying(200)	total bigint
5	10
15	700

- 平均屏幕分辨率

```
选择 AVG(CAST(browser->'resolution'->>'x' AS INTEGER)) 作为 width,
  AVG(CAST(browser->'resolution'->>'y' AS INTEGER)) 作为 height
来自 events;
```

输出：

Now lets select everything:

```
SELECT * FROM events;
```

Output:

name character varying(200)	visitor_id character varying(200)	properties json	browser json
pageview	1	{ "page": "/" }	{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }
pageview	2	{ "page": "/" }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1920, "y": 1200 } }
pageview	1	{ "page": "/account" }	{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }
purchase	5	{ "amount": 10 }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1024, "y": 768 } }
purchase	15	{ "amount": 200 }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }
purchase	15	{ "amount": 500 }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }

JSON operators + PostgreSQL aggregate functions

Using the JSON operators, combined with traditional PostgreSQL aggregate functions, we can pull out whatever we want. You have the full might of an RDBMS at your disposal.

- Lets see browser usage:

```
SELECT browser->>'name' AS browser,
  COUNT(browser)
  FROM events
  GROUP BY browser->>'name';
```

Output:

browser text	count bigint
Firefox	4
Chrome	2

- Total revenue per visitor:

```
SELECT visitor_id, SUM(CAST(properties->>'amount' AS INTEGER)) AS total
FROM events
WHERE CAST(properties->>'amount' AS INTEGER) > 0
GROUP BY visitor_id;
```

Output:

visitor_id character varying(200)	total bigint
5	10
15	700

- Average screen resolution

```
SELECT AVG(CAST(browser->'resolution'->>'x' AS INTEGER)) AS width,
  AVG(CAST(browser->'resolution'->>'y' AS INTEGER)) AS height
FROM events;
```

Output:

width numeric	height numeric
1397.3333333333333333	894.6666666666666667

更多示例和文档 here 和 here。

第10.2节：查询复杂的JSON文档

在表中使用复杂的JSON文档：

```
创建表 mytable (DATA JSONB 非空);
创建索引 mytable_idx 在 mytable 使用 gin (DATA jsonb_path_ops);
插入到 mytable 值($$
{
    "name": "爱丽丝",
    "emails": [
        "alice1@test.com",
        "alice2@test.com"
    ],
    "events": [
        {
            "type": "生日",
            "date": "1970-01-01"
        },
        {
            "type": "周年纪念",
            "date": "2001-05-05"
        }
    ],
    "locations": {
        "home": {
            "city": "伦敦",
            "country": "英国"
        },
        "work": {
            "city": "爱丁堡",
            "country": "英国"
        }
    }
}
$$);
```

查询顶层元素：

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"name":"Alice"}';
```

查询数组中的简单项：

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"emails":["alice1@test.com"]}';
```

查询数组中的对象：

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"events":[{"type":"anniversary"}]}';
```

嵌套对象的查询：

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"locations":{"home":{"city":"London"}}}';
```


width numeric	height numeric
1397.3333333333333333	894.6666666666666667

More examples and documentation here and here.

Section 10.2: Querying complex JSON documents

Taking a complex JSON document in a table:

```
CREATE TABLE mytable (DATA JSONB NOT NULL);
CREATE INDEX mytable_idx ON mytable USING gin (DATA jsonb_path_ops);
INSERT INTO mytable VALUES($$
{
    "name": "Alice",
    "emails": [
        "alice1@test.com",
        "alice2@test.com"
    ],
    "events": [
        {
            "type": "birthday",
            "date": "1970-01-01"
        },
        {
            "type": "anniversary",
            "date": "2001-05-05"
        }
    ],
    "locations": {
        "home": {
            "city": "London",
            "country": "United Kingdom"
        },
        "work": {
            "city": "Edinburgh",
            "country": "United Kingdom"
        }
    }
}
$$);
```

Query for a top-level element:

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"name":"Alice"}';
```

Query for a simple item in an array:

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"emails":["alice1@test.com"]}';
```

Query for an object in an array:

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"events":[{"type":"anniversary"}]}';
```

Query for a nested object:

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"locations":{"home":{"city":"London"}}}';
```

性能比较 @> 与 -> 和 ->>

理解在查询的 WHERE 部分使用 @>、-> 和 ->> 之间的性能差异非常重要。尽管这两个查询看起来大致等价：

```
SELECT DATA FROM mytable WHERE DATA @> '{"name":"Alice"}';
SELECT DATA FROM mytable WHERE DATA->'name' = '"Alice"';
SELECT DATA FROM mytable WHERE DATA->>'name' = 'Alice';
```

第一个语句将使用上述创建的索引，而后两个则不会，需进行完整的表扫描。

在获取结果数据时仍然允许使用 -> 操作符，因此以下查询也将使用索引：

```
SELECT DATA->'locations'->'work' FROM mytable WHERE DATA @> '{"name":"Alice"}';
SELECT DATA->'locations'->'work'->>'city' FROM mytable WHERE DATA @> '{"name":"Alice"}';
```

第10.3节：创建纯JSON表

要创建纯JSON表，您需要提供一个类型为 JSONB 的单个字段：

```
CREATE TABLE mytable (DATA JSONB NOT NULL);
```

你还应该创建一个基本索引：

```
CREATE INDEX mytable_idx ON mytable USING gin (DATA jsonb_path_ops);
```

此时你可以向表中插入数据并高效查询。

Performance of @> compared to -> and ->>

It is important to understand the performance difference between using @>, -> and ->> in the **WHERE** part of the query. Although these two queries appear to be broadly equivalent:

```
SELECT DATA FROM mytable WHERE DATA @> '{"name":"Alice"}';
SELECT DATA FROM mytable WHERE DATA->'name' = '"Alice"';
SELECT DATA FROM mytable WHERE DATA->>'name' = 'Alice';
```

the first statement will use the index created above whereas the latter two will not, requiring a complete table scan.

It is still allowable to use the -> operator when obtaining resultant data, so the following queries will also use the index:

```
SELECT DATA->'locations'->'work' FROM mytable WHERE DATA @> '{"name":"Alice"}';
SELECT DATA->'locations'->'work'->>'city' FROM mytable WHERE DATA @> '{"name":"Alice"}';
```

Section 10.3: Creating a pure JSON table

To create a pure JSON table you need to provide a single field with the type JSONB:

```
CREATE TABLE mytable (DATA JSONB NOT NULL);
```

You should also create a basic index:

```
CREATE INDEX mytable_idx ON mytable USING gin (DATA jsonb_path_ops);
```

At this point you can insert data in to the table and query it efficiently.

第11章：聚合函数

第11.1节：简单统计：min()、max()、avg()

为了确定表中某列值的一些简单统计信息，你可以使用聚合函数。

如果你的individuals表是：

姓名	年龄
艾莉	17
阿曼达	14
艾拉娜	20

您可以编写此语句以获取最小值、最大值和平均值：

```
SELECT MIN(age), MAX(age), AVG(age)
FROM individuals;
```

结果：

最小	最大	平均
14	20	17

第11.2节：regr_slope(Y, X)：由(X, Y)对确定的最小二乘拟合线的斜率

为了说明如何使用regr_slope(Y,X)，我将其应用于一个现实世界的问题。在Java中，如果不正确清理内存，垃圾可能会卡住并占满内存。你每小时会导出不同类的内存使用统计数据，并将其加载到Postgres数据库中进行分析。

所有内存泄漏候选对象都会随着时间推移消耗更多内存。如果你绘制这个趋势，你会想象一条向上且向左的线：

```
    ^
|
s   |  图例：
i   |  *  - 数据点
z   |  -- - 趋势
e    |
(    |
b    |                *
y    |                    --
t    |                --
e    |            * --    *
s    |          --
)    |       *--      *
|     --    *
|   -- *
--------------------------------------->
时间
```

假设你有一个包含堆转储直方图数据的表（类到它们所占内存的映射）：

Chapter 11: Aggregate Functions

Section 11.1: Simple statistics: min(), max(), avg()

In order to determine some simple statistics of a value in a column of a table, you can use an aggregate function.

If your `individuals` table is:

Name	Age
Allie	17
Amanda	14
Alana	20

You could write this statement to get the minimum, maximum and average value:

```
SELECT MIN(age), MAX(age), AVG(age)
FROM individuals;
```

Result:

min	max	avg
14	20	17

Section 11.2: regr_slope(Y, X) : slope of the least-squares-fit linear equation determined by the (X, Y) pairs

To illustrate how to use regr_slope(Y,X), I applied it to a real world problem. In Java, if you don't clean up memory properly, the garbage can get stuck and fill up the memory. You dump statistics every hour about memory utilization of different classes and load it into a postgres database for analysis.

All memory leak candidates will have a trend of consuming more memory as more time passes. If you plot this trend, you would imagine a line going up and to the left:

```
    ^
|
s   |  Legend:
i   |  *  - data point
z   |  -- - trend
e   |
(   |
b   |                *
y   |                    --
t   |                --
e   |            * --    *
s   |          --
)   |       *--      *
|     --    *
|   -- *
--------------------------------------->
time
```

Suppose you have a table containing heap dump histogram data (a mapping of classes to how much memory they consume):

```
CREATE TABLE heap_histogram (
    -- 直方图采集时间
    histwhen TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    -- 对象类型对应的字节
    -- 例如：java.util.String
    CLASS CHARACTER VARYING NOT NULL,
    -- 上述类所使用的字节大小
    bytes INTEGER NOT NULL
);
```

为了计算每个类别的斜率，我们按类别进行分组。HAVING 子句 > 0 确保我们只获得斜率为正的候选项（即向上且向左倾斜的直线）。我们按斜率降序排序，以便将内存增长速率最大的类别排在最前面。

```
-- epoch 返回秒数
SELECT CLASS, REGR_SLOPE(bytes,EXTRACT(epoch FROM histwhen)) AS slope
    FROM public.heap_histogram
    GROUP BY CLASS
    HAVING REGR_SLOPE(bytes,EXTRACT(epoch FROM histwhen)) > 0
    ORDER BY slope DESC ;
```

输出：

```
class              |        slope
---------------------------+---------------------
java.util.ArrayList        |      71.7993806279174
java.util.HashMap          |       49.0324576155785
java.lang.String           |       31.7770770326123
joe.schmoe.BusinessObject  |       23.2036817108056
java.lang.ThreadLocal      |       20.9013528767851
```

从输出中我们看到，java.util.ArrayList 的内存消耗增长最快，达到每秒 71.799 字节，可能是内存泄漏的一部分。

第11.3节：string_agg(expression, delimiter)

您可以使用 STRING_AGG() 函数将字符串按分隔符连接起来。

如果你的individuals表是：

姓名	年龄	国家
艾莉	15	美国
阿曼达	14	美国
阿拉娜	20	俄罗斯

你可以编写SELECT ... GROUP BY语句来获取每个国家的名称：

```
SELECT STRING_AGG(NAME, ', ') AS NAMES, country
FROM individuals
GROUP BY country;
```

注意你需要使用GROUP BY子句，因为STRING_AGG()是一个聚合函数。

结果：


```
CREATE TABLE heap_histogram (
    -- when the heap histogram was taken
    histwhen TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    -- the object type bytes are referring to
    -- ex: java.util.String
    CLASS CHARACTER VARYING NOT NULL,
    -- the size in bytes used by the above class
    bytes INTEGER NOT NULL
);
```

To compute the slope for each class, we group by over the class. The HAVING clause > 0 ensures that we get only candidates with a positive slop (a line going up and to the left). We sort by the slope descending so that we get the classes with the largest rate of memory increase at the top.

```
-- epoch returns seconds
SELECT CLASS, REGR_SLOPE(bytes,EXTRACT(epoch FROM histwhen)) AS slope
    FROM public.heap_histogram
    GROUP BY CLASS
    HAVING REGR_SLOPE(bytes,EXTRACT(epoch FROM histwhen)) > 0
    ORDER BY slope DESC ;
```

Output:

```
         class              |        slope
----------------------------+---------------------
java.util.ArrayList         |     71.7993806279174
java.util.HashMap           |     49.0324576155785
java.lang.String            |     31.7770770326123
joe.schmoe.BusinessObject   |     23.2036817108056
java.lang.ThreadLocal       |     20.9013528767851
```

From the output we see that java.util.ArrayList's memory consumption is increasing the fastest at 71.799 bytes per second and is potentially part of the memory leak.

Section 11.3: string_agg(expression, delimiter)

You can concatenate strings separated by delimiter using the `STRING_AGG()` function.

If your `individuals` table is:

Name	Age	Country
Allie	15	USA
Amanda	14	USA
Alana	20	Russia

You could write **`SELECT ... GROUP BY`** statement to get names from each country:

```
SELECT STRING_AGG(NAME, ', ') AS NAMES, country
FROM individuals
GROUP BY country;
```

Note that you need to use a **`GROUP BY`** clause because `STRING_AGG()` is an aggregate function.

Result:

names	country
Allie, Amanda	美国
Alana	俄罗斯

这里描述了更多PostgreSQL聚合函数

names	country
Allie, Amanda	USA
Alana	Russia

More PostgreSQL aggregate function described here

第12章：公共表表达式 (WITH)

第12.1节：SELECT查询中的公共表表达式

公共表表达式支持从较大的查询中提取部分内容。例如：

```
WITH sales AS (
  SELECT
orders.ordered_at,
    orders.user_id,
     SUM(orders.amount) AS total
  FROM orders
  GROUP BY orders.ordered_at, orders.user_id
)
SELECT
sales.ordered_at,
  sales.total,
users.NAME
FROM sales
JOIN users USING (user_id)
```

第12.2节：使用WITH RECURSIVE遍历树

```
创建表 empl (
    姓名 文本 主键,
上司 文本 可为空
        引用 姓名
            更新时级联
            删除时级联
        默认值 NULL
);

插入到 empl 值为 ('Paul',NULL);
插入到 empl 值为 ('Luke','Paul');
插入到 empl 值为 ('Kate','Paul');
插入到 empl 值为 ('Marge','Kate');
插入到 empl 值为 ('Edith','Kate');
插入到 empl 值为 ('Pam','Kate');
插入到 empl 值为 ('Carol','Luke');
插入到 empl 值为 ('John','Luke');
插入到 empl 值为 ('Jack','Carol');
插入到 empl 值为 ('Alex','Carol');

使用递归 t(级别,路径,上司,姓名) 作为 (
        选择 0,姓名,上司,姓名 从 empl 哪里 上司 为空
    联合
        SELECT
            级别 + 1,
路径 || ' > ' || 员工.姓名,
          员工.上司,
empl.姓名
        来自
empl 加入 t
                在 empl.boss = t.姓名
) 选择 * 来自 t 按 路径排序;
```


Chapter 12: Common Table Expressions (WITH)

Section 12.1: Common Table Expressions in SELECT Queries

Common table expressions support extracting portions of larger queries. For example:

```
WITH sales AS (
  SELECT
    orders.ordered_at,
    orders.user_id,
    SUM(orders.amount) AS total
  FROM orders
  GROUP BY orders.ordered_at, orders.user_id
)
SELECT
  sales.ordered_at,
  sales.total,
  users.NAME
FROM sales
JOIN users USING (user_id)
```

Section 12.2: Traversing tree using WITH RECURSIVE

```
CREATE TABLE empl (
    NAME TEXT PRIMARY KEY,
    boss TEXT NULL
        REFERENCES NAME
            ON UPDATE CASCADE
            ON DELETE CASCADE
        DEFAULT NULL
);

INSERT INTO empl VALUES ('Paul',NULL);
INSERT INTO empl VALUES ('Luke','Paul');
INSERT INTO empl VALUES ('Kate','Paul');
INSERT INTO empl VALUES ('Marge','Kate');
INSERT INTO empl VALUES ('Edith','Kate');
INSERT INTO empl VALUES ('Pam','Kate');
INSERT INTO empl VALUES ('Carol','Luke');
INSERT INTO empl VALUES ('John','Luke');
INSERT INTO empl VALUES ('Jack','Carol');
INSERT INTO empl VALUES ('Alex','Carol');

WITH RECURSIVE t(LEVEL,path,boss,NAME) AS (
        SELECT 0,NAME,boss,NAME FROM empl WHERE boss IS NULL
    UNION
        SELECT
            LEVEL + 1,
            path || ' > ' || empl.NAME,
            empl.boss,
            empl.NAME
        FROM
            empl JOIN t
                ON empl.boss = t.NAME
) SELECT * FROM t ORDER BY path;
```

第13章：窗口函数

第13.1节：通用示例

准备数据：

```
创建表 wf_example(i 整数, t 文本,ts 时间戳带时区,b 布尔值);
插入到 wf_example 选择 1,'a','1970.01.01',真;
插入到 wf_example 选择 1,'a','1970.01.01',假;
插入到 wf_example 选择 1,'b','1970.01.01',假;
插入到 wf_example 选择 2,'b','1970.01.01',假;
插入到 wf_example 选择 3,'b','1970.01.01',假;
插入到 wf_example 选择 4,'b','1970.02.01',假;
插入到 wf_example 选择 5,'b','1970.03.01',假;
插入到 wf_example 选择 2,'c','1970.03.01',真;
```

运行：

```
选择 *
, DENSE_RANK() OVER (按 i排序) 按i的密集排名
  , LAG(t) OVER () 前一个t
, NTH_VALUE(i, 6) OVER () 第n个值
  , COUNT(TRUE) OVER (PARTITION BY i) 按i分区的计数
  , COUNT(TRUE) OVER () 全窗口计数
, NTILE(3) over() ntile函数
FROM wf_example
;
```

结果：

```
i | t |           ts           | b | dist_by_i | prev_t | nth | num_by_i | num_all | ntile
---+---+------------------------+---+-----------+--------+-----+----------+---------+-------
1 | a | 1970-01-01 00:00:00+01 | f |         1 |        |   3 |        3 |       8 |     1
1 | a | 1970-01-01 00:00:00+01 | t |         1 | a      |   3 |        3 |       8 |     1
1 | b | 1970-01-01 00:00:00+01 | f |         1 | a      |   3 |        3 |       8 |     1
2 | c | 1970-03-01 00:00:00+01 | t |         2 | b      |   3 |        2 |       8 |     2
2 | b | 1970-01-01 00:00:00+01 | f |         2 | c      |   3 |        2 |       8 |     2
3 | b | 1970-01-01 00:00:00+01 | f |         3 | b      |   3 |        1 |       8 |     2
4 | b | 1970-02-01 00:00:00+01 | f |         4 | b      |   3 |        1 |       8 |     3
5 | b | 1970-03-01 00:00:00+01 | f |         5 | b      |   3 |        1 |       8 |     3
(8 rows)
```

说明：

dist_by_i: DENSE_RANK() OVER (ORDER BY i) 类似于每个不同值的行号。可用于计算i的不同值数量（COUNT(DISTINCT i) 不适用）。只需使用最大值。

prev_t: LAG(t) OVER () 是整个窗口中t的前一个值。注意第一行该值为null。

nth: `NTH_VALUE(i, 6) OVER ()` 是整个窗口中第六行的i列的值

num_by_i: `COUNT(TRUE) OVER (PARTITION BY i)` 是每个i值对应的行数

num_all: `COUNT(TRUE) OVER ()` 是整个窗口的行数

ntile: NTILE(3) over() 将整个窗口尽可能均分为3个数量相等的部分

Chapter 13: Window Functions

Section 13.1: generic example

Preparing data:

```
CREATE TABLE wf_example(i INT, t TEXT,ts timestamptz,b BOOLEAN);
INSERT INTO wf_example SELECT 1,'a','1970.01.01',TRUE;
INSERT INTO wf_example SELECT 1,'a','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 1,'b','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 2,'b','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 3,'b','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 4,'b','1970.02.01',FALSE;
INSERT INTO wf_example SELECT 5,'b','1970.03.01',FALSE;
INSERT INTO wf_example SELECT 2,'c','1970.03.01',TRUE;
```

Running:

```
SELECT *
  , DENSE_RANK() OVER (ORDER BY i) dist_by_i
  , LAG(t) OVER () prev_t
  , NTH_VALUE(i, 6) OVER () nth
  , COUNT(TRUE) OVER (PARTITION BY i) num_by_i
  , COUNT(TRUE) OVER () num_all
  , NTILE(3) over() ntile
FROM wf_example
;
```

Result:

```
 i | t |           ts           | b | dist_by_i | prev_t | nth | num_by_i | num_all | ntile
---+---+------------------------+---+-----------+--------+-----+----------+---------+-------
1 | a | 1970-01-01 00:00:00+01 | f |         1 |        |   3 |        3 |       8 |     1
1 | a | 1970-01-01 00:00:00+01 | t |         1 | a      |   3 |        3 |       8 |     1
1 | b | 1970-01-01 00:00:00+01 | f |         1 | a      |   3 |        3 |       8 |     1
2 | c | 1970-03-01 00:00:00+01 | t |         2 | b      |   3 |        2 |       8 |     2
2 | b | 1970-01-01 00:00:00+01 | f |         2 | c      |   3 |        2 |       8 |     2
3 | b | 1970-01-01 00:00:00+01 | f |         3 | b      |   3 |        1 |       8 |     2
4 | b | 1970-02-01 00:00:00+01 | f |         4 | b      |   3 |        1 |       8 |     3
5 | b | 1970-03-01 00:00:00+01 | f |         5 | b      |   3 |        1 |       8 |     3
(8 rows)
```

Explanation:

dist_by_i: `DENSE_RANK() OVER (ORDER BY i)` is like a row_number per distinct values. Can be used for the number of distinct values of i (`COUNT(DISTINCT i)` wold not work). Just use the maximum value.

prev_t: `LAG(t) OVER ()` is a previous value of t over the whole window. mind that it is null for the first row.

nth: `NTH_VALUE(i, 6) OVER ()` is the value of sixth rows column i over the whole window

num_by_i: `COUNT(TRUE) OVER (PARTITION BY i)` is an amount of rows for each value of i

num_all: `COUNT(TRUE) OVER ()` is an amount of rows over a whole window

ntile: `NTILE(3) over()` splits the whole window to 3 (as much as possible) equal in quantity parts

第13.2节：列值 vs dense_rank vs rank vs row_number

这里你可以找到这些功能。

使用前面示例中创建的表 wf_example，运行：

```
SELECT i
, DENSE_RANK() OVER (ORDER BY i)
  , ROW_NUMBER() OVER ()
, RANK() OVER (ORDER BY i)
FROM wf_example
```

结果是：

```
i | dense_rank | row_number | rank
---+------------+------------+------
1 |          1 |          1 |    1
1 |          1 |          2 |    1
1 |          1 |          3 |    1
2 |          2 |          4 |    4
2 |          2 |          5 |    4
3 |          3 |          6 |    6
4 |          4 |          7 |    7
5 |          5 |          8 |    8
```

- dense_rank 按照 i 在窗口中出现的顺序对 VALUES 进行排序。 i=1 出现，所以第一行的 dense_rank 是 1，接下来第二和第三个 i 值不变，因此 dense_rank 仍显示为 1。第四行 i=2，是 i 遇到的第二个不同值，所以 dense_rank 显示为 2，后面一行也是如此。第六行遇到值 i=3，所以显示为 3。其余两个 i 值同理。因此最后的 dense_rank 值是 i 不同值的数量。

- row_number 按照行的列出顺序对 ROWS 进行排序。

- rank 不同于 dense_rank，此函数对 i 值的 ROW NUMBER 进行排序。开始时前三个都是 1，但下一个值是 4，表示 i=2（新值）在第 4 行出现。同理 i=3 在第 6 行出现。

等等。。

Section 13.2: column values vs dense_rank vs rank vs row_number

here you can find the functions.

With the table wf_example created in previous example, run:

```
SELECT i
  , DENSE_RANK() OVER (ORDER BY i)
  , ROW_NUMBER() OVER ()
  , RANK() OVER (ORDER BY i)
FROM wf_example
```

The result is:

```
 i | dense_rank | row_number | rank
---+------------+------------+------
1 |          1 |          1 |    1
1 |          1 |          2 |    1
1 |          1 |          3 |    1
2 |          2 |          4 |    4
2 |          2 |          5 |    4
3 |          3 |          6 |    6
4 |          4 |          7 |    7
5 |          5 |          8 |    8
```

- *dense_rank* orders **VALUES** of **i** by appearance in window. `i=1` appears, so first row has dense_rank, next and third i value does not change, so it is `dense_rank` shows *1* - FIRST value not changed. fourth row `i=2`, it is second value of *i* met, so `dense_rank` shows 2, andso for the next row. Then it meets value `i=3` at 6th row, so it show 3. Same for the rest two values of *i*. So the last value of `dense_rank` is the number of distinct values of *i*.

- *row_number* orders **ROWS** as they are listed.

- *rank* Not to confuse with `dense_rank` this function orders **ROW NUMBER** of **i** values. So it starts same with three ones, but has next value 4, which means `i=2` (new value) was met at row 4. Same `i=3` was met at row 6. Etc..

第14章：递归查询

实际上没有真正的递归查询！

第14.1节：整数求和

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT SUM(n) FROM t;
```

文档链接

Chapter 14: Recursive queries

There are no real recursive querys!

Section 14.1: Sum of Integers

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT SUM(n) FROM t;
```

Link to Documentation

第15章：使用PL/pgSQL编程

第15.1节：基本的PL/pgSQL函数

一个简单的 PL/pgSQL 函数：

```
CREATE FUNCTION active_subscribers() RETURNS BIGINT AS $$
DECLARE
    -- 以下 BEGIN … END 块的变量
subscribers INTEGER;
BEGIN
    -- SELECT 必须总是与 INTO 一起使用
    SELECT COUNT(user_id) INTO subscribers FROM users WHERE subscribed;
    -- 函数结果
    RETURN subscribers;
EXCEPTION
    -- 如果表 "users" 不存在则返回 NULL
    WHEN undefined_table
    THEN RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

这本可以仅通过SQL语句实现，但演示了函数的基本结构。

执行该函数的方法是：

```
SELECT active_subscribers();
```

第15.2节：自定义异常

创建自定义异常 'P2222'：

```
CREATE OR REPLACE FUNCTION s164() RETURNS void AS
$$
BEGIN
raise exception USING message = 'S 164', detail = 'D 164', hint = 'H 164', errcode = 'P2222';
END;
$$ LANGUAGE plpgsql
;
```

创建自定义异常但未分配errm：

```
创建或替换函数 s165() 返回 void 作为
$$
BEGIN
引发异常 '%','未指定';
结束;
$$ LANGUAGE plpgsql
;
```

调用：

```
t=# 执行
$$
声明
 _t 文本;
开始
```


Chapter 15: Programming with PL/pgSQL

Section 15.1: Basic PL/pgSQL Function

A simple PL/pgSQL function:

```
CREATE FUNCTION active_subscribers() RETURNS BIGINT AS $$
DECLARE
    -- variable for the following BEGIN ... END block
    subscribers INTEGER;
BEGIN
    -- SELECT must always be used with INTO
    SELECT COUNT(user_id) INTO subscribers FROM users WHERE subscribed;
    -- function result
    RETURN subscribers;
EXCEPTION
    -- return NULL if table "users" does not exist
    WHEN undefined_table
    THEN RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

This could have been achieved with just the SQL statement but demonstrates the basic structure of a function.

To execute the function do:

```
SELECT active_subscribers();
```

Section 15.2: custom exceptions

creating custom exception 'P2222':

```
CREATE OR REPLACE FUNCTION s164() RETURNS void AS
$$
BEGIN
raise exception USING message = 'S 164', detail = 'D 164', hint = 'H 164', errcode = 'P2222';
END;
$$ LANGUAGE plpgsql
;
```

creating custom exception not assigning errm:

```
CREATE OR REPLACE FUNCTION s165() RETURNS void AS
$$
BEGIN
raise exception '%','nothing specified';
END;
$$ LANGUAGE plpgsql
;
```

calling:

```
t=# DO
$$
DECLARE
 _t TEXT;
BEGIN
```

```
执行 s165();
异常 当 SQLSTATE 'P0001' 时 引发信息 '%','捕获状态 P0001: '||SQLERRM;
  执行 s164();

结束;
$$
;
信息： 捕获状态 P0001: 未指定
错误：  S 164
详情：  D 164
提示：H 164
上下文：SQL 语句 "SELECT s164()"
PL/pgSQL 函数 内联代码块 行 7 在 PERFORM
```

这里自定义的 P0001 已处理，P2222 未处理，正在中止执行。

另外，维护一个异常表非常有意义，比如这里：http://stackoverflow.com/a/2700312/5315974

第15.3节：PL/pgSQL 语法

```
CREATE [OR REPLACE] FUNCTION 函数名 (某参数 '参数类型')
RETURNS '数据类型'
AS $_block_name_$
DECLARE
    --声明某些内容
BEGIN
    --执行某些操作
    --返回某些内容
结束;
$_block_name_$
语言 plpgsql;
```

第15.4节：RETURNS块

PL/pgSQL函数中的返回选项：

- 数据类型所有数据类型列表
- 表(列名 列类型，...)
- SETOF '数据类型' 或 '表列'


```
  perform s165();
  exception WHEN SQLSTATE 'P0001' THEN raise info '%','state P0001 caught: '||SQLERRM;
  perform s164();

END;
$$
;
INFO:  state P0001 caught: NOTHING specified
ERROR:  S 164
DETAIL:  D 164
HINT:  H 164
CONTEXT:  SQL STATEMENT "SELECT s164()"
PL/pgSQL FUNCTION inline_code_block line 7 AT PERFORM
```

here custom P0001 processed, and P2222, not, aborting the execution.

Also it makes huge sense to keep a table of exceptions, like here: http://stackoverflow.com/a/2700312/5315974

Section 15.3: PL/pgSQL Syntax

```
CREATE [OR REPLACE] FUNCTION functionName (someParameter 'parameterType')
RETURNS 'DATATYPE'
AS $_block_name_$
DECLARE
    --declare something
BEGIN
    --do something
    --return something
END;
$_block_name_$
LANGUAGE plpgsql;
```

Section 15.4: RETURNS Block

Options for returning in a PL/pgSQL function:

- `Datatype` List of all datatypes
- `Table(column_name column_type, ...)`
- `SETOF 'Datatype' OR 'table_column'`

第16章：继承

第16.1节：创建子表

```
CREATE TABLE users (username TEXT, email TEXT);
CREATE TABLE simple_users () 继承自 (users);
CREATE TABLE users_with_password (PASSWORD TEXT) 继承自 (users);
```

我们的三个表如下所示：

users

列	类型
username	text
email	text

simple_users

Column	Type
username	text
email	text

users_with_password

Column	Type
username	text
email	text
password	text

Chapter 16: Inheritance

Section 16.1: Creating children tables

```
CREATE TABLE users (username TEXT, email TEXT);
CREATE TABLE simple_users () INHERITS (users);
CREATE TABLE users_with_password (PASSWORD TEXT) INHERITS (users);
```

Our three tables look like this:

users

Column	Type
username	text
email	text

simple_users

Column	Type
username	text
email	text

users_with_password

Column	Type
username	text
email	text
password	text

第17章：导出PostgreSQL数据库表头和数据到CSV文件

Adminer管理工具有导出MySQL数据库为CSV文件的选项，但PostgreSQL数据库没有。这里我将展示导出PostgreSQL数据库CSV的命令。

第17.1节：从查询复制

```
COPY (SELECT oid,relname FROM pg_class LIMIT 5) TO STDOUT;
```

第17.2节：导出PostgreSQL表到带有表头的CSV文件，针对某些列

```
COPY products(is_public, title, discount) TO 'D:\csv_backup\products_db.csv' DELIMITER ',' CSV HEADER;

COPY categories(NAME) TO 'D:\csv_backup\categories_db.csv' DELIMITER ',' CSV HEADER;
```

第17.3节：带表头的完整表备份为csv

```
COPY products TO 'D:\csv_backup\products_db.csv' DELIMITER ',' CSV HEADER;

COPY categories TO 'D:\csv_backup\categories_db.csv' DELIMITER ',' CSV HEADER;
```


Chapter 17: Export PostgreSQL database table header and data to CSV file

From Adminer management tool it's has export to csv file option for mysql database But not available for postgresql database. Here I will show the command to export CSV for postgresql database.

Section 17.1: copy from query

```
COPY (SELECT oid,relname FROM pg_class LIMIT 5) TO STDOUT;
```

Section 17.2: Export PostgreSQL table to csv with header for some column(s)

```
COPY products(is_public, title, discount) TO 'D:\csv_backup\products_db.csv' DELIMITER ',' CSV HEADER;

COPY categories(NAME) TO 'D:\csv_backup\categories_db.csv' DELIMITER ',' CSV HEADER;
```

Section 17.3: Full table backup to csv with header

```
COPY products TO 'D:\csv_backup\products_db.csv' DELIMITER ',' CSV HEADER;

COPY categories TO 'D:\csv_backup\categories_db.csv' DELIMITER ',' CSV HEADER;
```

第18章：触发器和触发函数

该触发器将关联指定的表或视图，并在发生特定事件时执行指定的函数 function_name。

第18.1节：触发器类型

触发器可以指定在以下情况下触发：

- **在对行尝试操作之前 - 插入、更新或删除；**
- **在操作完成之后 - 插入、更新或删除；**
- **对于视图上的插入、更新或删除操作，替代该操作执行。**

标记为的触发器：

- `FOR EACH ROW` 表示对操作修改的每一行调用一次；
- `FOR EACH STATEMENT` 表示对任何给定操作调用一次。

准备执行示例

```
创建表 company (
    id          SERIAL 主键 不为空,
    NAME        文本 不为空,
created_at  时间戳,
    modified_at 时间戳 默认 NOW()
)

创建表 log (
id         SERIAL 主键 不为空,
    table_name  文本 不为空,
table_id    文本 不为空,
    description 文本 不为空,
    created_at  时间戳 默认 NOW()
)
```

单条插入触发器

步骤1：创建你的函数

```
创建或替换函数 add_created_at_function()
  返回触发器 AS $BODY$
BEGIN
  NEW.created_at := NOW();
  返回 NEW;
END $BODY$
LANGUAGE plpgsql;
```

步骤 2：创建触发器

```
CREATE TRIGGER add_created_at_trigger
BEFORE INSERT
ON company
FOR EACH ROW
EXECUTE PROCEDURE add_created_at_function();
```

步骤 3：测试它

```
INSERT INTO company (NAME) VALUES ('我的公司');
SELECT * FROM company;
```

多用途触发器

步骤 1：创建函数

Chapter 18: Triggers and Trigger Functions

The trigger will be associated with the specified table or view and will execute the specified function function_name when certain events occur.

Section 18.1: Type of triggers

Trigger can be specified to fire:

- `BEFORE` the operation is attempted on a row - insert, update or delete;
- `AFTER` the operation has completed - insert, update or delete;
- `INSTEAD OF` the operation in the case of inserts, updates or deletes on a view.

Trigger that is marked:

- `FOR EACH ROW` is called once for every row that the operation modifies;
- `FOR EACH STATEMENT` is called onde for any given operation.

Preparing to execute examples

```
CREATE TABLE company (
    id          SERIAL PRIMARY KEY NOT NULL,
    NAME        TEXT NOT NULL,
    created_at  TIMESTAMP,
    modified_at TIMESTAMP DEFAULT NOW()
)

CREATE TABLE log (
    id          SERIAL PRIMARY KEY NOT NULL,
    table_name  TEXT NOT NULL,
    table_id    TEXT NOT NULL,
    description TEXT NOT NULL,
    created_at  TIMESTAMP DEFAULT NOW()
)
```

Single insert trigger

Step 1: create your function

```
CREATE OR REPLACE FUNCTION add_created_at_function()
  RETURNS TRIGGER AS $BODY$
BEGIN
  NEW.created_at := NOW();
  RETURN NEW;
END $BODY$
LANGUAGE plpgsql;
```

Step 2: create your trigger

```
CREATE TRIGGER add_created_at_trigger
BEFORE INSERT
ON company
FOR EACH ROW
EXECUTE PROCEDURE add_created_at_function();
```

Step 3: test it

```
INSERT INTO company (NAME) VALUES ('My company');
SELECT * FROM company;
```

Trigger for multiple purpose

Step 1: create your function

```
CREATE OR REPLACE FUNCTION add_log_function()
  RETURNS TRIGGER AS$BODY$
DECLARE
vDescription TEXT;
  vId INT;
  vReturn RECORD;
BEGIN
vDescription := TG_TABLE_NAME || ' ';
     IF (TG_OP = 'INSERT') THEN
vId := NEW.id;
vDescription := vDescription || 'added. Id: ' || vId;
        vReturn := NEW;
    ELSIF (TG_OP = 'UPDATE') THEN
vId := NEW.id;
vDescription := vDescription || 'updated. Id: ' || vId;
        vReturn := NEW;
    ELSIF (TG_OP = 'DELETE') THEN
vId := OLD.id;
vDescription := vDescription || '已删除。编号: ' || vId;
       vReturn := OLD;
    END IF;

RAISE NOTICE '触发器在 % 上被调用 - 日志: %', TG_TABLE_NAME, vDescription;

    INSERT INTO log
        (table_name, table_id, description, created_at)
        VALUES
        (TG_TABLE_NAME, vId, vDescription, NOW());

    RETURN vReturn;
END $BODY$
  LANGUAGE plpgsql;
```

步骤 2：创建触发器

```
CREATE TRIGGER add_log_trigger
AFTER INSERT OR UPDATE OR DELETE
ON company
FOR EACH ROW
EXECUTE PROCEDURE add_log_function();
```

步骤 3：测试它

```
INSERT INTO company (NAME) VALUES ('公司 1');
INSERT INTO company (NAME) VALUES ('公司 2');
INSERT INTO company (NAME) VALUES ('公司 3');
UPDATE company SET NAME='公司 新 2' WHERE NAME='公司 2';
DELETE FROM company WHERE NAME='公司 1';
SELECT * FROM log;
```

第18.2节：基本的PL/pgSQL触发器函数

这是一个简单的触发器函数。

```
创建或替换函数 my_simple_trigger_function()
返回触发器 作为
$BODY$

BEGIN
    -- TG_TABLE_NAME ：触发器调用的表名
```



```
CREATE OR REPLACE FUNCTION add_log_function()
  RETURNS TRIGGER AS $BODY$
DECLARE
  vDescription TEXT;
  vId INT;
  vReturn RECORD;
BEGIN
    vDescription := TG_TABLE_NAME || ' ';
     IF (TG_OP = 'INSERT') THEN
        vId := NEW.id;
        vDescription := vDescription || 'added. Id: ' || vId;
        vReturn := NEW;
    ELSIF (TG_OP = 'UPDATE') THEN
        vId := NEW.id;
        vDescription := vDescription || 'updated. Id: ' || vId;
        vReturn := NEW;
    ELSIF (TG_OP = 'DELETE') THEN
        vId := OLD.id;
        vDescription := vDescription || 'deleted. Id: ' || vId;
        vReturn := OLD;
    END IF;

    RAISE NOTICE 'TRIGER called on % - Log: %', TG_TABLE_NAME, vDescription;

    INSERT INTO log
        (table_name, table_id, description, created_at)
        VALUES
        (TG_TABLE_NAME, vId, vDescription, NOW());

    RETURN vReturn;
END $BODY$
  LANGUAGE plpgsql;
```

Step 2: create your trigger

```
CREATE TRIGGER add_log_trigger
AFTER INSERT OR UPDATE OR DELETE
ON company
FOR EACH ROW
EXECUTE PROCEDURE add_log_function();
```

Step 3: test it

```
INSERT INTO company (NAME) VALUES ('Company 1');
INSERT INTO company (NAME) VALUES ('Company 2');
INSERT INTO company (NAME) VALUES ('Company 3');
UPDATE company SET NAME='Company new 2' WHERE NAME='Company 2';
DELETE FROM company WHERE NAME='Company 1';
SELECT * FROM log;
```

Section 18.2: Basic PL/pgSQL Trigger Function

This is a simple trigger function.

```
CREATE OR REPLACE FUNCTION my_simple_trigger_function()
RETURNS TRIGGER AS
$BODY$

BEGIN
    -- TG_TABLE_NAME :name of the table that caused the trigger invocation
```

```
如果 (TG_TABLE_NAME = 'users') 则

    --TG_OP ：触发器触发的操作
  IF (TG_OP = 'INSERT') THEN
    --NEW.id 保存新的数据库行值（这里id是users表中的id列）
    --DELETE操作时NEW将返回null
    插入到 log_table (date_and_time, description) 值为 (NOW(), '新用户已插入。用户ID：
'|| NEW.id);
    RETURN NEW;

  ELSIF (TG_OP = 'DELETE') THEN
    --OLD.id 保存旧的数据库行值（这里的 id 是 users 表中的 id 列）
    --对于 INSERT 操作，OLD 将返回 null
    INSERT INTO log_table (date_and_time, description) VALUES (NOW(), '用户已删除.. 用户 ID: ' ||
OLD.id);
    RETURN OLD;

  END IF;

RETURN NULL;
END IF;

结束;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
```

将此触发器函数添加到 users 表中

```
CREATE TRIGGER my_trigger
AFTER INSERT OR DELETE
ON users
FOR EACH ROW
EXECUTE PROCEDURE my_simple_trigger_function();
```



```
IF (TG_TABLE_NAME = 'users') THEN

    --TG_OP : operation the trigger was fired
  IF (TG_OP = 'INSERT') THEN
    --NEW.id is holding the new database row value (in here id is the id column in users table)
    --NEW will return null for DELETE operations
    INSERT INTO log_table (date_and_time, description) VALUES (NOW(), 'New user inserted. User ID:
'|| NEW.id);
    RETURN NEW;

  ELSIF (TG_OP = 'DELETE') THEN
    --OLD.id is holding the old database row value (in here id is the id column in users table)
    --OLD will return null for INSERT operations
    INSERT INTO log_table (date_and_time, description) VALUES (NOW(), 'User deleted.. User ID: ' ||
OLD.id);
    RETURN OLD;

  END IF;

RETURN NULL;
END IF;

END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
```

Adding this trigger function to the `users` table

```
CREATE TRIGGER my_trigger
AFTER INSERT OR DELETE
ON users
FOR EACH ROW
EXECUTE PROCEDURE my_simple_trigger_function();
```

第19章：事件触发器

事件触发器将在数据库中发生与其关联的事件时被触发。

第19.1节：记录DDL命令开始事件

事件类型-

- DDL_COMMAND_START
- DDL_COMMAND_END
- SQL_DROP

这是创建事件触发器并记录DDL_COMMAND_START事件的示例。

```
创建表 TAB_EVENT_LOGS(
  DATE_TIME 时间戳,
EVENT_NAME 文本,
  REMARKS 文本
);

创建或替换函数 FN_LOG_EVENT()
  返回 EVENT_TRIGGER
  语言 SQL
  作为
  $main$
    INSERT INTO TAB_EVENT_LOGS(DATE_TIME,EVENT_NAME,REMARKS)
     VALUES(NOW(),TG_TAG,'事件记录');
  $main$;

CREATE EVENT TRIGGER TRG_LOG_EVENT ON DDL_COMMAND_START
  EXECUTE PROCEDURE FN_LOG_EVENT();
```


Chapter 19: Event Triggers

Event Triggers will be fired whenever event associated with them occurs in database.

Section 19.1: Logging DDL Command Start Events

Event Type-

- DDL_COMMAND_START
- DDL_COMMAND_END
- SQL_DROP

This is example for creating an Event Trigger and logging DDL_COMMAND_START events.

```
CREATE TABLE TAB_EVENT_LOGS(
  DATE_TIME TIMESTAMP,
  EVENT_NAME TEXT,
  REMARKS TEXT
);

CREATE OR REPLACE FUNCTION FN_LOG_EVENT()
  RETURNS EVENT_TRIGGER
  LANGUAGE SQL
  AS
  $main$
    INSERT INTO TAB_EVENT_LOGS(DATE_TIME,EVENT_NAME,REMARKS)
      VALUES(NOW(),TG_TAG,'Event Logging');
  $main$;

CREATE EVENT TRIGGER TRG_LOG_EVENT ON DDL_COMMAND_START
  EXECUTE PROCEDURE FN_LOG_EVENT();
```

第20章：角色管理

第20.1节：创建带密码的用户

通常你应该避免在应用程序中使用默认的数据库角色（通常是postgres）。你应该改为创建一个权限较低的用户。这里我们创建一个名为 niceusername的用户，并为其设置密码very-strong-PASSWORD

```
CREATE ROLE niceusername WITH PASSWORD 'very-strong-password' LOGIN;
```

问题在于，输入到psql控制台的查询会被保存在用户主目录下的历史文件.psql_history中，也可能被记录到PostgreSQL数据库服务器日志中，从而暴露密码。

为避免此问题，使用\PASSWORD命令设置用户密码。如果执行该命令的用户是超级用户，则不会询问当前密码。（必须是超级用户才能修改超级用户的密码）

```
CREATE ROLE niceusername WITH LOGIN;
\PASSWORD niceusername
```

第20.2节：授予和撤销权限

假设，我们有三个用户：

1. 数据库管理员 > admin
2. 对其数据拥有完全访问权限的应用程序 > read_write
3. 只读访问 > read_only

```
--访问数据库
REVOKE CONNECT ON DATABASE nova FROM PUBLIC;
GRANT  CONNECT ON DATABASE nova  TO USER;
```

通过上述查询，不受信任的用户将无法连接到数据库。

```
--访问模式
REVOKE ALL     ON SCHEMA public FROM PUBLIC;
GRANT  USAGE   ON SCHEMA public  TO USER;
```

下一组查询撤销了未认证用户的所有权限，并为

read_write 用户提供了有限的权限集。

```
--访问表
撤销 PUBLIC 对 public 模式中所有表的所有权限；
授予 read_only 对 public 模式中所有表的 SELECT 权限；
授予 read_write 对 public 模式中所有表的 SELECT、INSERT、UPDATE、DELETE 权限；
授予 ADMIN 对 public 模式中所有表的所有权限；

--访问序列
撤销所有序列在模式public上的权限从PUBLIC;
授予选择权限在模式public上的所有序列给read_only;-- 允许使用CURRVAL
授予更新权限在模式public上的所有序列给read_write;-- 允许使用NEXTVAL和SETVAL
授予使用权限在模式public上的所有序列给read_write;-- 允许使用CURRVAL和NEXTVAL
授予所有权限在模式public上的所有序列给ADMIN;
```


Chapter 20: Role Management

Section 20.1: Create a user with a password

Generally you should avoid using the default database role (often `postgres`) in your application. You should instead create a user with lower levels of privileges. Here we make one called `niceusername` and give it a password `very-strong-PASSWORD`

```
CREATE ROLE niceusername WITH PASSWORD 'very-strong-password' LOGIN;
```

The problem with that is that queries typed into the `psql` console get saved in a history file `.psql_history` in the user's home directory and may as well be logged to the PostgreSQL database server log, thus exposing the password.

To avoid this, use the `\PASSWORD` command to set the user password. If the user issuing the command is a superuser, the current password will not be asked. (Must be superuser to alter passwords of superusers)

```
CREATE ROLE niceusername WITH LOGIN;
\PASSWORD niceusername
```

Section 20.2: Grant and Revoke Privileges

Suppose, that we have three users :

1. The Administrator of the database > admin
2. The application with a full access for her data > read_write
3. The read only access > read_only

```
--ACCESS DB
REVOKE CONNECT ON DATABASE nova FROM PUBLIC;
GRANT  CONNECT ON DATABASE nova  TO USER;
```

With the above queries, untrusted users can no longer connect to the database.

```
--ACCESS SCHEMA
REVOKE ALL     ON SCHEMA public FROM PUBLIC;
GRANT  USAGE   ON SCHEMA public  TO USER;
```

The next set of queries revoke all privileges from unauthenticated users and provide limited set of privileges for the `read_write` user.

```
--ACCESS TABLES
REVOKE ALL ON ALL TABLES IN SCHEMA public FROM PUBLIC ;
GRANT SELECT                         ON ALL TABLES IN SCHEMA public TO read_only ;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO read_write ;
GRANT ALL                            ON ALL TABLES IN SCHEMA public TO ADMIN ;

--ACCESS SEQUENCES
REVOKE ALL   ON ALL SEQUENCES IN SCHEMA public FROM PUBLIC;
GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO read_only; -- allows the use of CURRVAL
GRANT UPDATE ON ALL SEQUENCES IN SCHEMA public TO read_write; -- allows the use of NEXTVAL and SETVAL
GRANT USAGE  ON ALL SEQUENCES IN SCHEMA public TO read_write; -- allows the use of CURRVAL and NEXTVAL
GRANT ALL    ON ALL SEQUENCES IN SCHEMA public TO ADMIN;
```

第20.3节：创建角色和匹配数据库

为了支持某个应用程序，通常会创建一个新的角色和匹配的数据库。

需要运行的shell命令如下：

```
$ CREATEUSER -P blogger
输入新角色的密码： ********
再次输入: ********

$ CREATEDB -O blogger blogger
```

这假设pg_hba.conf已被正确配置，配置内容大致如下：

```
# 类型   数据库          用户              地址                  方法
host     sameuser         ALL               localhost                md5
LOCAL    sameuser          ALL                                       md5
```

第20.4节：更改用户的默认search_path

通过以下命令，可以设置用户的默认search_path。

1. 设置默认schema之前，检查search_path。

```
postgres=# \c postgres user1
您现在已以用户"user1"连接到数据库"postgres"。
postgres=> SHOW search_path;
  search_path
----------------
 "$user",public
(1 ROW)
```

2. 使用ALTER USER命令设置search_path以追加新的schemamy_schema

```
postgres=> \c postgres postgres
您现在已连接到数据库"postgres"以用户"postgres"身份。
postgres=# 修改用户 user1 设置 search_path='my_schema, "$user", public';
修改角色
```

3. 执行后检查结果。

```
postgres=# \c postgres user1
用户 user1的密码：
您现在已以用户"user1"连接到数据库"postgres"。
postgres=> 显示 search_path;
 search_path
-------------
my_schema, "$user", public
(1 行)
```

替代方案：

```
postgres=# 设置角色 user1;
postgres=# 显示 search_path;
 search_path
-------------
my_schema, "$user", public
(1 行)
```


Section 20.3: Create Role and matching database

To support a given application, you often create a new role and database to match.

The shell commands to run would be these:

```
$ CREATEUSER -P blogger
Enter PASSWORD FOR the NEW ROLE: ********
Enter it again: ********

$ CREATEDB -O blogger blogger
```

This assumes that `pg_hba.conf` has been properly configured, which probably looks like this:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    sameuser        ALL             localhost               md5
LOCAL   sameuser        ALL                                     md5
```

Section 20.4: Alter default search_path of user

With the below commands, user's default search_path can be set.

1. Check search path before set default schema.

```
postgres=# \c postgres user1
You are now connected TO DATABASE "postgres" AS USER "user1".
postgres=> SHOW search_path;
  search_path
----------------
 "$user",public
(1 ROW)
```

2. Set `search_path` with **ALTER USER** command to append a new schema `my_schema`

```
postgres=> \c postgres postgres
You are now connected TO DATABASE "postgres" AS USER "postgres".
postgres=# ALTER USER user1 SET search_path='my_schema, "$user", public';
ALTER ROLE
```

3. Check result after execution.

```
postgres=# \c postgres user1
PASSWORD FOR USER user1:
You are now connected TO DATABASE "postgres" AS USER "user1".
postgres=> SHOW search_path;
 search_path
-------------
 my_schema, "$user", public
(1 ROW)
```

Alternative:

```
postgres=# SET ROLE user1;
postgres=# SHOW search_path;
 search_path
-------------
 my_schema, "$user", public
(1 ROW)
```

第20.5节：创建只读用户

```
创建用户 readonly 使用加密密码 'yourpassword';
授予 连接 数据库<database_name> 给 readonly;

授予 USAGE 权限于模式 public 给 readonly;
授予 SELECT 权限于所有 序列 在模式 public 给 readonly;
授予 SELECT 权限于所有 表 在模式 public 给 readonly;
```

第20.6节：授予对未来创建对象的访问权限

假设，我们有三个用户：

1. 数据库管理员 > ADMIN
2. 对其数据具有完全访问权限的应用程序 > read_write
3. 只读访问 > read_only

通过以下查询，您可以设置指定模式中未来创建对象的访问权限。

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema 授予 SELECT 权限                    于 表 给
read_only;
ALTER DEFAULT PRIVILEGES IN SCHEMAmyschema GRANT SELECT,INSERT,DELETE,UPDATE ON TABLES TO
read_write;
   ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT ALL                        ON TABLES TO ADMIN;
```

或者，您可以为指定用户创建的未来对象设置访问权限。

```
ALTER DEFAULT PRIVILEGES FOR ROLE ADMIN GRANT SELECT  ON TABLES TO read_only;
```


Section 20.5: Create Read Only User

```
CREATE USER readonly WITH ENCRYPTED PASSWORD 'yourpassword';
GRANT CONNECT ON DATABASE <database_name> TO readonly;

GRANT USAGE ON SCHEMA public TO readonly;
GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO readonly;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly;
```

Section 20.6: Grant access privileges on objects created in the future

Suppose, that we have `three users` :

1. The Administrator of the database > **`ADMIN`**
2. The application with a full access for her data > `read_write`
3. The read only access > `read_only`

With below queries, you can set access privileges on objects created in the future in specified schema.

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT                    ON TABLES TO
read_only;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT,INSERT,DELETE,UPDATE ON TABLES TO
read_write;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT ALL                       ON TABLES TO ADMIN;
```

Or, you can set access privileges on objects created in the future by specified user.

```
ALTER DEFAULT PRIVILEGES FOR ROLE ADMIN GRANT SELECT  ON TABLES TO read_only;
```

第21章：Postgres加密函数

在Postgres中，可以通过使用pgcrypto模块来启用加密函数。CREATE EXTENSION pgcrypto;

第21.1节：digest

DIGEST()函数生成给定数据的二进制哈希。该函数可用于创建随机哈希。

用法：`digest(DATA TEXT, TYPE TEXT) RETURNS BYTEA`

或者：`digest(DATA BYTEA, TYPE TEXT) RETURNS BYTEA`

示例：

- `SELECT DIGEST('1', 'sha1')`
- **`SELECT DIGEST(CONCAT(CAST(CURRENT_TIMESTAMP AS TEXT), RANDOM()::TEXT), 'sha1')`**

Chapter 21: Postgres cryptographic functions

In Postgres, cryptographic functions can be unlocked by using pgcrypto module. CREATE EXTENSION pgcrypto;

Section 21.1: digest

`DIGEST()` functions generate a binary hash of the given data. This **can** be used to create a random hash.

Usage: `digest(DATA TEXT, TYPE TEXT) RETURNS BYTEA`

Or: `digest(DATA BYTEA, TYPE TEXT) RETURNS BYTEA`

Examples:

- **`SELECT`** `DIGEST('1', 'sha1')`
- **`SELECT`** `DIGEST(CONCAT(CAST(CURRENT_TIMESTAMP AS TEXT), RANDOM()::TEXT), 'sha1')`

第22章：PostgreSQL中的注释

COMMENT 主要目的是定义或更改数据库对象的注释。

任何数据库对象只能有一个注释（字符串）。COMMENT帮助我们了解特定数据库对象的定义目的及其实际用途。

关于COMMENT ON ROLE的规则是，必须是超级用户才能对超级用户角色添加注释，或者拥有CREATEROLE权限才能对非超级用户角色添加注释。当然，超级用户可以对任何对象添加注释。

第22.1节：表的注释

```
COMMENT ON TABLE table_name IS '这是学生详细信息表';
```

第22.2节：删除注释

```
COMMENT ON TABLE student IS NULL;
```

执行上述语句后，注释将被删除。

Chapter 22: Comments in PostgreSQL

COMMMENT main purpose is to define or change a comment on database object.

Only a single comment(string) can be given on any database object. COMMENT will help us to know what for the particular database object has been defined whats its actual purpose is.

The rule for ***COMMENT ON ROLE*** is that you must be superuser to comment on a superuser role, or have the ***CREATEROLE*** privilege to comment on non-superuser roles. Of course, a *superuser can comment on anything*

Section 22.1: COMMENT on Table

```
COMMENT ON TABLE table_name IS 'this is student details table';
```

Section 22.2: Remove Comment

```
COMMENT ON TABLE student IS NULL;
```

Comment will be removed with above statement execution.

第23章：备份与恢复

第23.1节：备份一个数据库

```
pg_dump -Fc -f DATABASE.pgsql DATABASE
```

选项 -Fc 选择"自定义备份格式"，它比原始 SQL 更强大；详情请参见 pg_restore 。如果你想要一个普通的 SQL 文件，可以改用以下命令：

```
pg_dump -f DATABASE.sql DATABASE
```

或者甚至

```
pg_dump DATABASE > DATABASE.sql
```

第23.2节：恢复备份

```
psql < backup.sql
```

一个更安全的替代方法是使用 -1 将恢复操作包裹在一个事务中。选项 -f 指定文件名，而不是使用 shell 重定向。

```
psql -1f backup.sql
```

自定义格式文件必须使用带有 -d 选项指定数据库的 pg_restore 来恢复：

```
pg_restore -d DATABASE DATABASE.pgsql
```

自定义格式也可以转换回SQL：

```
pg_restore backup.pgsql > backup.sql
```

推荐使用自定义格式，因为您可以选择恢复的内容，并且可以选择启用并行处理。

如果您从一个PostgreSQL版本升级到更新版本，可能需要先执行pg_dump，然后再执行pg_restore。

第23.3节：备份整个集群

```
$ pg_dumpall -f backup.sql
```

这在后台通过对服务器的每个数据库建立多个连接并执行
pg_dump 来实现。

有时，您可能想将此设置为cron任务，因此希望在文件名中包含备份的日期：

```
$ postgres-backup-$(DATE +%Y-%m-%d).sql
```

但是，请注意，这可能会每天产生较大的文件。Postgresql 有一个更好的机制用于定期备份——WAL 归档

Chapter 23: Backup and Restore

Section 23.1: Backing up one database

```
pg_dump -Fc -f DATABASE.pgsql DATABASE
```

The `-Fc` selects the "custom backup format" which gives you more power than raw SQL; see `pg_restore` for more details. If you want a vanilla SQL file, you can do this instead:

```
pg_dump -f DATABASE.sql DATABASE
```

or even

```
pg_dump DATABASE > DATABASE.sql
```

Section 23.2: Restoring backups

```
psql < backup.sql
```

A safer alternative uses `-1` to wrap the restore in a transaction. The `-f` specifies the filename rather than using shell redirection.

```
psql -1f backup.sql
```

Custom format files must be restored using `pg_restore` with the `-d` option to specify the database:

```
pg_restore -d DATABASE DATABASE.pgsql
```

The custom format can also be converted back to SQL:

```
pg_restore backup.pgsql > backup.sql
```

Usage of the custom format is recommended because you can choose which things to restore and optionally enable parallel processing.

You may need to do a pg_dump followed by a pg_restore if you upgrade from one postgresql release to a newer one.

Section 23.3: Backing up the whole cluster

```
$ pg_dumpall -f backup.sql
```

This works behind the scenes by making multiple connections to the server once for each database and executing `pg_dump` on it.

Sometimes, you might be tempted to set this up as a cron job, so you want to see the date the backup was taken as part of the filename:

```
$ postgres-backup-$(DATE +%Y-%m-%d).sql
```

However, please note that this could produce large files on a daily basis. Postgresql has a much better mechanism for regular backups - WAL archives

pg_dumpall 的输出足以恢复到配置完全相同的 Postgres 实例，但位于 $PGDATA 目录下的配置文件（pg_hba.conf 和 postgresql.conf）不包含在备份中，因此您需要单独备份它们。

```
postgres=# SELECT pg_start_backup('my-backup');
postgres=# SELECT pg_stop_backup();
```

要进行文件系统备份，必须使用这些函数以确保在准备备份时 Postgres 处于一致状态。

第23.4节：使用 psql 导出数据

数据可以使用 copy 命令导出，或者利用 psql 命令的命令行选项导出。

将表 user 的数据导出为 csv 文件：

```
psql -p \<端口> -U \<用户名> -d \<数据库> -A -F<分隔符> -c\<要执行的 SQL> \> \<带路径的输出文件名>


        psql -p 5432 -U postgres -d test_database -A -F, -c "select * from user" > /home/USER/user_data.CSV
```

这里 -A 和 -F 的组合起到了关键作用。

-F 用于指定分隔符

```
-A 或 --no-align
```

切换到未对齐输出模式。（默认输出模式为对齐模式。）

第23.5节：使用Copy导入

从CSV文件复制数据到表中

```
COPY <表名> FROM '<带路径的文件名>';
```

从位于 /home/USER/ 目录下名为 user_data.CSV 的文件插入数据到表 USER：

```
COPY USER FROM '/home/user/user_data.csv';
```

从管道分隔文件复制数据到表中

```
COPY USER FROM '/home/user/user_data' WITH DELIMITER '|';
```

注意：如果没有选项**WITH DELIMITER**，默认分隔符为逗号，

导入文件时忽略表头行

使用 Header 选项：

```
COPY USER FROM '/home/user/user_data' WITH DELIMITER '|' HEADER;
```

注意：如果数据被引用，默认的数据引用字符是双引号。如果数据使用其他字符引用，使用 QUOTE 选项；但该选项仅在使用 CSV 格式时允许。

The output from pg_dumpall is sufficient to restore to an identically-configured Postgres instance, but the configuration files in $PGDATA (pg_hba.conf and postgresql.conf) are not part of the backup, so you'll have to back them up separately.

```
postgres=# SELECT pg_start_backup('my-backup');
postgres=# SELECT pg_stop_backup();
```

To take a filesystem backup, you must use these functions to help ensure that Postgres is in a consistent state while the backup is prepared.

Section 23.4: Using psql to export data

Data can be exported using copy command or by taking use of command line options of psql command.

To Export csv data from table user to csv file:

```
psql -p \<port> -U \<username> -d \<DATABASE> -A -F<DELIMITER> -c\<sql TO EXECUTE> \> \<output
filename WITH path>

psql -p 5432 -U postgres -d test_database -A -F, -c "select * from user" > /home/USER/user_data.CSV
```

Here combination of -A and -F does the trick.

-F is to specify delimiter

```
-A OR --no-align
```

Switches to unaligned output mode. (The default output mode is otherwise aligned.)

Section 23.5: Using Copy to import

To Copy Data from a CSV file to a table

```
COPY <tablename> FROM '<filename with path>';
```

To insert into table `USER` from a file named `user_data.CSV` placed inside `/home/USER/`:

```
COPY USER FROM '/home/user/user_data.csv';
```

To Copy data from pipe separated file to table

```
COPY USER FROM '/home/user/user_data' WITH DELIMITER '|';
```

Note: In absence of the option **WITH DELIMITER**, the default delimiter is comma ,

To ignore header line while importing file

Use the Header option:

```
COPY USER FROM '/home/user/user_data' WITH DELIMITER '|' HEADER;
```

Note: If data is quoted, by default data quoting characters are double quote. If the data is quoted using any other character use the **QUOTE** option; however, this option is allowed only when using CSV format.

第 23.6 节：使用 Copy 导出

将表复制到标准输出

```
COPY <tablename> TO STDOUT (DELIMITER '|');
```

导出表 user 到标准输出：

```
COPY USER TO STDOUT (DELIMITER '|');
```
将表复制到文件

```
COPY USER FROM '/home/user/user_data' WITH DELIMITER '|';
```
将 SQL 语句的输出复制到文件

```
COPY (sql STATEMENT) TO '<filename with path>'; COPY (SELECT * FROM USER WHERE user_name LIKE 'A%')
TO '/home/user/user_data';
```
复制到压缩文件

```
COPY USER TO PROGRAM 'gzip > /home/user/user_data.gz';
```

这里执行程序 gzip 来压缩 user 表数据。

Section 23.6: Using Copy to export

To Copy table to standard o/p

```
COPY <tablename> TO STDOUT (DELIMITER '|');
```

To export table user to Standard output:

```
COPY USER TO STDOUT (DELIMITER '|');
```
To Copy table to file

```
COPY USER FROM '/home/user/user_data' WITH DELIMITER '|';
```
To Copy the output of SQL statement to file

```
COPY (sql STATEMENT) TO '<filename with path>'; COPY (SELECT * FROM USER WHERE user_name LIKE 'A%')
TO '/home/user/user_data';
```
To Copy into a compressed file

```
COPY USER TO PROGRAM 'gzip > /home/user/user_data.gz';
```

Here program gzip is executed to compress user table data.

第24章：生产数据库的备份脚本

参数	详细信息
保存数据库	主备份目录
dbProd	次备份目录
日期	备份日期（指定格式）
dbprod	要保存的数据库名称
/opt/postgres/9.0/bin/pg_dump	pg_dump二进制文件的路径
-h	指定服务器运行所在机器的主机名，例如：localhost
-p	指定服务器监听连接的TCP端口或本地Unix域套接字文件扩展名，例如5432
-U	连接时使用的用户名。

第24.1节：saveProdDb.sh

通常，我们倾向于使用pgAdmin客户端备份数据库。以下是一个用于保存数据库（在Linux下）为两种格式的sh脚本：

- SQL文件：用于在任何版本的PostgreSQL上可能恢复数据。
- 转储文件：用于高于当前版本的版本。

```
#!/bin/sh
cd /save_db
#rm -R /save_db/*
DATE=$(date +%d-%m-%Y-%Hh%M)
echo -e "备份数据库于 ${DATE}"
mkdir prodDir${DATE}
cd prodDir${DATE}

#转储文件
/opt/postgres/9.0/bin/pg_dump -i -h localhost -p 5432 -U postgres -F c -b -w -v -f
"dbprod${DATE}.backup" dbprod

#SQL 文件
/opt/postgres/9.0/bin/pg_dump -i -h localhost -p 5432 -U postgres --format plain --verbose  -f
"dbprod${DATE}.sql" dbprod
```


Chapter 24: Backup script for a production DB

parameter	details
save_db	The main backup directory
dbProd	The secondary backup directory
DATE	The date of the backup in the specified format
dbprod	The name of the database to be saved
/opt/postgres/9.0/bin/pg_dump	The path to the pg_dump binary
-h	Specifies the host name of the machine on which the server is running, Example : localhost
-p	Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections, Example 5432
-U	User name to connect as.

Section 24.1: saveProdDb.sh

In general, we tend to back up the DB with the pgAdmin client. The following is a sh script used to save the database (under linux) in two formats:

- **SQL file**: for a possible resume of data on any version of PostgreSQL.
- **Dump file**: for a higher version than the current version.

```
#!/bin/sh
cd /save_db
#rm -R /save_db/*
DATE=$(date +%d-%m-%Y-%Hh%M)
echo -e "Sauvegarde de la base du ${DATE}"
mkdir prodDir${DATE}
cd prodDir${DATE}

#dump file
/opt/postgres/9.0/bin/pg_dump -i -h localhost -p 5432 -U postgres -F c -b -w -v -f
"dbprod${DATE}.backup" dbprod

#SQL file
/opt/postgres/9.0/bin/pg_dump -i -h localhost -p 5432 -U postgres --format plain --verbose  -f
"dbprod${DATE}.sql" dbprod
```

第25章：以编程方式访问数据

第25.1节：使用C-API访问PostgreSQL

C-API是访问PostgreSQL最强大的方式，而且使用起来出乎意料地方便。

编译和链接

在编译时，必须将PostgreSQL的包含目录添加到包含路径中，该目录可以通过pg_config`--includedir`命令找到。
必须链接PostgreSQL客户端共享库（UNIX上为libpq.so，Windows上为libpq.dll）。该库位于PostgreSQL的库目录中，可以通过pg_config --libdir命令找到。

注意：出于历史原因，该库名为libpq.so，而不是libpg.so，这常常是初学者容易犯的错误。

假设以下代码示例位于文件coltype.c中，编译和链接命令如下

```
gcc -Wall -I "$(pg_config --includedir)" -L "$(pg_config --libdir)" -o coltype coltype.c -lpq
```

使用GNU C编译器（建议添加`-Wl,-rpath,"$(pg_config --libdir)"`以添加库搜索路径）或
使用

```
cl /MT /W4 /I <include directory> coltype.c <path TO libpq.lib>
```

在Windows上使用Microsoft Visual C编译。

示例程序

```
/* 所有PostgreSQL客户端程序必需，且应放在首位 */
#include <libpq-fe.h>

#include <stdio.h>
#include <string.h>

#ifdef TRACE
#define TRACEFILE "trace.out"
#endif

int main(int argc, char **argv) {
#ifdef TRACE
    FILE *trc;
#endif
PGconn *conn;
PGresult *res;
    int rowcount, colcount, i, j, firstcol;
    /* 参数类型应由PostgreSQL推断 */
    const Oid paramTypes[1] = { 0 };
    /* 参数值 */
    const char * const paramValues[1] = { "pg_database" };

    /*
* 使用空的连接字符串将对所有内容使用默认值。
* 如果设置，将使用环境变量 PGHOST、PGDATABASE、PGPORT 和
     * PGUSER。
     */
conn = PQconnectdb("");
```


Chapter 25: Accessing Data Programmatically

Section 25.1: Accessing PostgreSQL with the C-API

The C-API is the most powerful way to access PostgreSQL and it is surprisingly comfortable.

Compilation and linking

During compilation, you have to add the PostgreSQL include directory, which can be found with `pg_config --includedir`, to the include path.
You must link with the PostgreSQL client shared library (`libpq.so` on UNIX, `libpq.dll` on Windows). This library is in the PostgreSQL library directory, which can be found with `pg_config --libdir`.

Note: For historical reason, the library is called `libpq.so`and *not* `libpg.so`, which is a popular trap for beginners.

Given that the below code sample is in file `coltype.c`, compilation and linking would be done with

```
gcc -Wall -I "$(pg_config --includedir)" -L "$(pg_config --libdir)" -o coltype coltype.c -lpq
```

with the GNU C compiler (consider adding `-Wl,-rpath,"$(pg_config --libdir)"` to add the library search path) or with

```
cl /MT /W4 /I <include directory> coltype.c <path TO libpq.lib>
```

on Windows with Microsoft Visual C.

Sample program

```
/* necessary for all PostgreSQL client programs, should be first */
#include <libpq-fe.h>

#include <stdio.h>
#include <string.h>

#ifdef TRACE
#define TRACEFILE "trace.out"
#endif

int main(int argc, char **argv) {
#ifdef TRACE
    FILE *trc;
#endif
    PGconn *conn;
    PGresult *res;
    int rowcount, colcount, i, j, firstcol;
    /* parameter type should be guessed by PostgreSQL */
    const Oid paramTypes[1] = { 0 };
    /* parameter value */
    const char * const paramValues[1] = { "pg_database" };

    /*
     * Using an empty connectstring will use default values for everything.
     * If set, the environment variables PGHOST, PGDATABASE, PGPORT and
     * PGUSER will be used.
     */
    conn = PQconnectdb("");
```

```
    /*
* 只有在没有足够内存
     * 分配 PGconn 结构时才会发生这种情况。
     */
    if (conn == NULL)
    {
        fprintf(stderr, "连接 PostgreSQL 时内存不足。");return 1;

    }

    /* 检查连接尝试是否成功 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));
        /*
* 即使连接失败，PGconn 结构体也已被
         * 分配，必须释放。
         */
PQfinish(conn);
        return 1;
    }

#ifdef TRACE
    if (NULL == (trc = fopen(TRACEFILE, "w")))
    {
        fprintf(stderr, "打开跟踪文件 \"%s\"失败！", TRACEFILE);          PQfinish(conn);

        return 1;
    }

    /* 用于客户端-服务器通信的跟踪 */
PQtrace(conn, trc);
#endif

    /* 该程序期望数据库返回UTF-8编码的数据 */
PQsetClientEncoding(conn, "UTF8");

    /* 使用参数执行查询 */
res = PQexecParams(
        conn,
         "SELECT column_name, data_type "
             "FROM information_schema.columns "
             "WHERE table_name = $1",
         1,           /* 一个参数 */
         paramTypes,
         paramValues,
         NULL,         /* 字符串参数不需要指定长度 */
         NULL,         /* 所有参数均为文本格式 */
         0           /* 结果应为文本格式 */
    );

    /* 内存不足或服务器通信中断 */
    if (NULL == res)
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));          PQfi
nish(conn);
#ifdef TRACE
        fclose(trc);
#endif
        return 1;
    }
```



```
    /*
     * This can only happen if there is not enough memory
     * to allocate the PGconn structure.
     */
    if (conn == NULL)
    {
        fprintf(stderr, "Out of memory connecting to PostgreSQL.\n");
        return 1;
    }

    /* check if the connection attempt worked */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
        /*
         * Even if the connection failed, the PGconn structure has been
         * allocated and must be freed.
         */
        PQfinish(conn);
        return 1;
    }

#ifdef TRACE
    if (NULL == (trc = fopen(TRACEFILE, "w")))
    {
        fprintf(stderr, "Error opening trace file \"%s\"!\n", TRACEFILE);
        PQfinish(conn);
        return 1;
    }

    /* tracing for client-server communication */
    PQtrace(conn, trc);
#endif

    /* this program expects the database to return data in UTF-8 */
    PQsetClientEncoding(conn, "UTF8");

    /* perform a query with parameters */
    res = PQexecParams(
        conn,
        "SELECT column_name, data_type "
            "FROM information_schema.columns "
            "WHERE table_name = $1",
        1,            /* one parameter */
        paramTypes,
        paramValues,
        NULL,        /* parameter lengths are not required for strings */
        NULL,        /* all parameters are in text format */
        0            /* result shall be in text format */
    );

    /* out of memory or sever communication broken */
    if (NULL == res)
    {
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
        PQfinish(conn);
#ifdef TRACE
        fclose(trc);
#endif
        return 1;
    }
```

```c
    /* SQL语句应返回结果 */
    if (PGRES_TUPLES_OK != PQresultStatus(res))
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));          PQfi
nish(conn);
#ifdef TRACE
        fclose(trc);
#endif
        return 1;
    }

    /* 获取结果行数和列数 */
     rowcount = PQntuples(res);
    colcount = PQnfields(res);

                    /* 打印列标题 */
firstcol = 1;

    printf("表 \"pg_database\"的描述");for (j=0; j<colcount; ++j)


    {
        if (firstcol)
            firstcol = 0;
        else
            printf(": ");

        printf(PQfname(res, j));
    }

    printf("");/* 遍历结

    果行 */for (i=0; i<rowcount; ++i){


        /* 打印所有列数据 */
firstcol = 1;

        for (j=0; j<colcount; ++j)
        {
            if (firstcol)
firstcol = 0;
            else
                printf(": ");

            printf(PQgetvalue(res, i, j));
        }

        printf("");
    }

    /* 这必须在每个语句之后执行以避免内存泄漏 */
PQclear(res);
    /* 关闭数据库连接并释放内存 */
PQfinish(conn);
#ifdef TRACE
    fclose(trc);
#endif
    return 0;
}
```



```c
    /* SQL statement should return results */
    if (PGRES_TUPLES_OK != PQresultStatus(res))
    {
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
        PQfinish(conn);
#ifdef TRACE
        fclose(trc);
#endif
        return 1;
    }

    /* get count of result rows and columns */
    rowcount = PQntuples(res);
    colcount = PQnfields(res);

    /* print column headings */
    firstcol = 1;

    printf("Description of the table \"pg_database\"\n");

    for (j=0; j<colcount; ++j)
    {
        if (firstcol)
            firstcol = 0;
        else
            printf(": ");

        printf(PQfname(res, j));
    }

    printf("\n\n");

    /* loop through rosult rows */
    for (i=0; i<rowcount; ++i)
    {
        /* print all column data */
        firstcol = 1;

        for (j=0; j<colcount; ++j)
        {
            if (firstcol)
                firstcol = 0;
            else
                printf(": ");

            printf(PQgetvalue(res, i, j));
        }

        printf("\n");
    }

    /* this must be done after every statement to avoid memory leaks */
    PQclear(res);
    /* close the database connection and release memory */
    PQfinish(conn);
#ifdef TRACE
    fclose(trc);
#endif
    return 0;
}
```

第25.2节：使用psycopg2从Python访问PostgreSQL

你可以在这里找到该驱动的描述。

快速示例是：

```
import psycopg2

db_host = 'postgres.server.com'
db_port = '5432'
db_un = 'user'
db_pw = 'password'
db_name = 'testdb'

conn = psycopg2.connect("dbname={} host={} user={} password={}".format(
                        db_name,  db_host, db_un, db_pw),
cursor_factory=RealDictCursor)
cur = conn.cursor()
sql = 'select * from testtable where id > %s and id < %s'
args = (1, 4)
cur.execute(sql, args)

print(cur.fetchall())
```

结果将是：

```
[{'id': 2, 'fruit': 'apple'}, {'id': 3, 'fruit': 'orange'}]
```

第25.3节：使用

Npgsql提供程序从.NET访问PostgreSQL

Postgresql中较受欢迎的.NET提供程序之一是Npgsql，它兼容ADO.NET，使用方式几乎与其他.NET数据库提供程序相同。

典型的查询是通过创建命令、绑定参数，然后执行命令来完成的。在C#中：

```
var connString = "Host=myserv;Username=myuser;Password=mypass;Database=mydb";
using (var conn = new NpgsqlConnection(connString))
{
    var querystring = "INSERT INTO data (some_field) VALUES (@content)";

    conn.Open();
    // 使用CommandText和Connection构造函数创建新命令
    using (var cmd = new NpgsqlCommand(querystring, conn))
    {
        // 添加参数并使用NpgsqlDbType枚举设置其类型
        var contentString = "Hello World!";
cmd.Parameters.Add("@content", NpgsqlDbType.Text).Value = contentString;

        // 执行不返回结果的查询
cmd.ExecuteNonQuery();


        /* 可以重用命令对象和打开的连接，而不是创建新的 */


        // 创建一个新的查询并设置其参数
```


Section 25.2: Accessing PostgreSQL from python using psycopg2

You can find description of the driver here.

The quick example is:

```
import psycopg2

db_host = 'postgres.server.com'
db_port = '5432'
db_un = 'user'
db_pw = 'password'
db_name = 'testdb'

conn = psycopg2.connect("dbname={} host={} user={} password={}".format(
                        db_name,  db_host, db_un, db_pw),
                        cursor_factory=RealDictCursor)
cur = conn.cursor()
sql = 'select * from testtable where id > %s and id < %s'
args = (1, 4)
cur.execute(sql, args)

print(cur.fetchall())
```

Will result:

```
[{'id': 2, 'fruit': 'apple'}, {'id': 3, 'fruit': 'orange'}]
```

Section 25.3: Accessing PostgreSQL from .NET using the Npgsql provider

One of the more popular .NET providers for Postgresql is Npgsql, which is ADO.NET compatible and is used nearly identically as other .NET database providers.

A typical query is performed by creating a command, binding parameters, and then executing the command. In C#:

```
var connString = "Host=myserv;Username=myuser;Password=mypass;Database=mydb";
using (var conn = new NpgsqlConnection(connString))
{
    var querystring = "INSERT INTO data (some_field) VALUES (@content)";

    conn.Open();
    // Create a new command with CommandText and Connection constructor
    using (var cmd = new NpgsqlCommand(querystring, conn))
    {
        // Add a parameter and set its type with the NpgsqlDbType enum
        var contentString = "Hello World!";
        cmd.Parameters.Add("@content", NpgsqlDbType.Text).Value = contentString;

        // Execute a query that returns no results
        cmd.ExecuteNonQuery();


        /* It is possible to reuse a command object and open connection instead of creating new ones
*/


        // Create a new query and set its parameters
```

```
        int keyId = 101;
cmd.CommandText = "SELECT primary_key, some_field FROM data WHERE primary_key = @keyId";
        cmd.Parameters.Clear();
cmd.Parameters.Add("@keyId", NpgsqlDbType.Integer).Value = keyId;

        // 执行命令并逐行读取数据
        using (NpgsqlDataReader reader = cmd.ExecuteReader())
        {
            while (reader.Read())   // 对于0行返回false，或读取完结果的最后一行后返回false

            {
                // 读取一个整数值
                int primaryKey = reader.GetInt32(0);
                // 或者
primaryKey = Convert.ToInt32(reader["primary_key"]);

                // 读取文本值
                string someFieldText = reader["some_field"].ToString();
            }
        }
    }
}   // C# 的 'using' 指令会自动调用 conn.Close() 和 conn.Dispose()
```

第25.4节：使用 Pomm2 从 PHP 访问 PostgreSQL

在底层驱动的基础上，有 pomm。它提出了模块化方法、数据转换器、监听/通知支持、数据库检查器等功能。

假设已通过 composer 安装了 Pomm，以下是一个完整示例：

```
<?php
use PommProject\Foundation\Pomm;
$loader = require __DIR__ . '/vendor/autoload.php';
$pomm = new Pomm(['my_db' => ['dsn' => 'pgsql://user:pass@host:5432/db_name']]);

// 表 comment (
// comment_id uuid 主键，created_at 带时区时间戳 非空，
// is_moderated 布尔型 非空 默认 false，
// content 文本 非空 CHECK (content !~ '^\s+$')，author_email 文本 非空)
$sql = <<<SQL
SELECT
  comment_id,
  created_at,
  is_moderated,
  content,
  author_email
FROM comment
INNER JOIN author USING (author_email)
WHERE
age(now(), created_at) < $*::interval
ORDER BY created_at ASC
SQL;

// 参数将按上述查询中的类型转换原样转换
$comments = $pomm['my_db']
    ->getQueryManager()
    ->query($sql, [DateInterval::createFromDateString('1 day')]);

if ($comments->isEmpty()) {
    printf("自昨天以来没有新评论。");
} else {
```



```
        int keyId = 101;
        cmd.CommandText = "SELECT primary_key, some_field FROM data WHERE primary_key = @keyId";
        cmd.Parameters.Clear();
        cmd.Parameters.Add("@keyId", NpgsqlDbType.Integer).Value = keyId;

        // Execute the command and read through the rows one by one
        using (NpgsqlDataReader reader = cmd.ExecuteReader())
        {
            while (reader.Read())   // Returns false for 0 rows, or after reading the last row of
the results
            {
                // read an integer value
                int primaryKey = reader.GetInt32(0);
                // or
                primaryKey = Convert.ToInt32(reader["primary_key"]);

                // read a text value
                string someFieldText = reader["some_field"].ToString();
            }
        }
    }
}   // the C# 'using' directive calls conn.Close() and conn.Dispose() for us
```

Section 25.4: Accessing PostgreSQL from PHP using Pomm2

On the shoulders of the low level drivers, there is pomm. It proposes a modular approach, data converters, listen/notify support, database inspector and much more.

Assuming, Pomm has been installed using composer, here is a complete example:

```
<?php
use PommProject\Foundation\Pomm;
$loader = require __DIR__ . '/vendor/autoload.php';
$pomm = new Pomm(['my_db' => ['dsn' => 'pgsql://user:pass@host:5432/db_name']]);

// TABLE comment (
// comment_id uuid PK, created_at timestamptz NN,
// is_moderated bool NN default false,
// content text NN CHECK (content !~ '^\s+$'), author_email text NN)
$sql = <<<SQL
SELECT
  comment_id,
  created_at,
  is_moderated,
  content,
  author_email
FROM comment
  INNER JOIN author USING (author_email)
WHERE
  age(now(), created_at) < $*::interval
ORDER BY created_at ASC
SQL;

// the argument will be converted as it is cast in the query above
$comments = $pomm['my_db']
    ->getQueryManager()
    ->query($sql, [DateInterval::createFromDateString('1 day')]);

if ($comments->isEmpty()) {
    printf("There are no new comments since yesterday.");
} else {
```

```
    foreach ($comments as $comment) {
        printf(
                "%s 已发布于 %s。 %s",$comment
                ['author_email'],$comment['cre
                ated_at']->format("Y-m-d H:i:s"),$comment['is_moderated'
                ] ? '[已审核]' : '');
    }
}
```

Pomm的查询管理模块会对查询参数进行转义以防止SQL注入。当参数被转换时，
它还会将它们从PHP表示转换为有效的Postgres值。结果是一个迭代器，内部使用游标。
每一行都会被即时转换，布尔值转换为布尔值，时间戳转换为 \DateTime 等。


```
    foreach ($comments as $comment) {
        printf(
                "%s has posted at %s. %s\n",
                $comment['author_email'],
                $comment['created_at']->format("Y-m-d H:i:s"),
                $comment['is_moderated'] ? '[OK]' : '');
    }
}
```

Pomm's query manager module escapes query arguments to prevent SQL injection. When the arguments are cast, it also converts them from a PHP representation to valid Postgres values. The result is an iterator, it uses a cursor internally. Every row is converted on the fly, booleans to booleans, timestamps to \DateTime etc.

第26章：从

Java连接PostgreSQL

从Java使用关系数据库的API是JDBC。

该API由JDBC驱动程序实现。

要使用它，需要将驱动程序的JAR文件放到JAVA类路径中。

本说明展示了如何使用JDBC驱动连接数据库的示例。

第26.1节：使用 java.sql.DriverManager 连接

这是最简单的连接方式。

首先，必须将驱动程序注册到java.sql.DriverManager，以便它知道使用哪个类。
这是通过加载驱动类来完成的，通常使用java.lang.CLASS.forname(;driver class name>)。

```
/**
 * 连接到PostgreSQL数据库。
 * @param url 要连接的JDBC URL；必须以 "jdbc:postgresql:" 开头
 * @param user 连接的用户名
 * @param password 连接的密码
 * @return 已建立连接的连接对象
 * @throws ClassNotFoundException 如果在Java类路径中找不到驱动类
 * @throws java.sql.SQLException 如果连接数据库失败
 */
private static java.sql.Connection connect(String url, String user, String password)
    throws ClassNotFoundException, java.sql.SQLException
{
    /*
* 注册PostgreSQL JDBC驱动。
* 这可能会抛出ClassNotFoundException。
     */
    Class.forName("org.postgresql.Driver");
    /*
* 告诉驱动管理器连接到指定URL的数据库。
* 这可能会抛出一个 SQLException。
     */
    return java.sql.DriverManager.getConnection(url, user, password);
}
```

注意，用户和密码也可以包含在 JDBC URL 中，这种情况下你不必在 getConnection 方法调用中指定它们。

第26.2节：使用 java.sql.DriverManager 和 Properties 连接

你可以将连接参数如用户和密码（完整列表见 here）打包到一个 java.util.Properties 对象中，而不是在 URL 中或作为单独参数指定：

```
/**
 * 连接到PostgreSQL数据库。
 * @param url 要连接的 JDBC URL。必须以 "jdbc:postgresql:" 开头
 * @param user 连接的用户名
 * @param password 连接的密码
```


Chapter 26: Connect to PostgreSQL from Java

The API to use a relational database from Java is JDBC.

This API is implemented by a JDBC driver.

To use it, put the JAR-file with the driver on the JAVA class path.

This documentation shows samples how to use the JDBC driver to connect to a database.

Section 26.1: Connecting with java.sql.DriverManager

This is the simplest way to connect.

First, the driver has to be *registered* with `java.sql.DriverManager` so that it knows which class to use.
This is done by loading the driver class, typically with `java.lang.CLASS.forname(`;driver class name>).

```
/**
 * Connect to a PostgreSQL database.
 * @param url the JDBC URL to connect to; must start with "jdbc:postgresql:"
 * @param user the username for the connection
 * @param password the password for the connection
 * @return a connection object for the established connection
 * @throws ClassNotFoundException if the driver class cannot be found on the Java class path
 * @throws java.sql.SQLException if the connection to the database fails
 */
private static java.sql.Connection connect(String url, String user, String password)
    throws ClassNotFoundException, java.sql.SQLException
{
    /*
     * Register the PostgreSQL JDBC driver.
     * This may throw a ClassNotFoundException.
     */
    Class.forName("org.postgresql.Driver");
    /*
     * Tell the driver manager to connect to the database specified with the URL.
     * This may throw an SQLException.
     */
    return java.sql.DriverManager.getConnection(url, user, password);
}
```

Not that user and password can also be included in the JDBC URL, in which case you don't have to specify them in the `getConnection` method call.

Section 26.2: Connecting with java.sql.DriverManager and Properties

Instead of specifying connection parameters like user and password (see a complete list here) in the URL or a separate parameters, you can pack them into a `java.util.Properties` object:

```
/**
 * Connect to a PostgreSQL database.
 * @param url the JDBC URL to connect to. Must start with "jdbc:postgresql:"
 * @param user the username for the connection
 * @param password the password for the connection
```

```
 * @return 已建立连接的连接对象
 * @throws ClassNotFoundException 如果在 Java 类路径中找不到驱动类
 * @throws java.sql.SQLException 如果连接数据库失败
 */
private static java.sql.Connection connect(String url, String user, String password)
    throws ClassNotFoundException, java.sql.SQLException
{
    /*
* 注册PostgreSQL JDBC驱动。
* 这可能会抛出ClassNotFoundException。
     */
    Class.forName("org.postgresql.Driver");
    java.util.Properties props = new java.util.Properties();
    props.setProperty("user", user);
props.setProperty("password", password);
    /* 不使用服务器预编译语句 */
props.setProperty("prepareThreshold", "0");
    /*
* 告诉驱动管理器连接到指定URL的数据库。
* 这可能会抛出一个 SQLException。
     */
    return java.sql.DriverManager.getConnection(url, props);
}
```

第26.3节：使用连接池通过javax.sql.DataSource连接

在应用服务器容器中，通常会使用javax.sql.DataSource结合JNDI，您可以在某个名称下注册数据源，并在需要连接时查找它。

以下代码演示了数据源的工作原理：

```
/**
* 为PostgreSQL连接创建带连接池的数据源
 * @param url 连接的JDBC URL，必须以"jdbc:postgresql:"开头
 * @param user 连接的用户名
* @param password 连接的密码
 * @return 设置了正确属性的数据源
 */
private static javax.sql.DataSource createDataSource(String url, String user, String password)
{
    /* 使用带连接池的数据源 */
org.postgresql.ds.PGPoolingDataSource ds = new org.postgresql.ds.PGPoolingDataSource();
    ds.setUrl(url);
ds.setUser(user);
    ds.setPassword(password);
    /* 连接池将包含10到20个连接 */
ds.setInitialConnections(10);
    ds.setMaxConnections(20);
    /* 使用SSL连接但不检查服务器证书 */
ds.setSslMode("require");
ds.setSslfactory("org.postgresql.ssl.NonValidatingFactory");

    return ds;
}
```

一旦通过调用此函数创建了数据源，您可以这样使用它：

```
/* 从连接池获取一个连接 */
java.sql.CONNECTION conn = ds.getConnection();
```



```
 * @return a connection object for the established connection
 * @throws ClassNotFoundException if the driver class cannot be found on the Java class path
 * @throws java.sql.SQLException if the connection to the database fails
 */
private static java.sql.Connection connect(String url, String user, String password)
    throws ClassNotFoundException, java.sql.SQLException
{
    /*
     * Register the PostgreSQL JDBC driver.
     * This may throw a ClassNotFoundException.
     */
    Class.forName("org.postgresql.Driver");
    java.util.Properties props = new java.util.Properties();
    props.setProperty("user", user);
    props.setProperty("password", password);
    /* don't use server prepared statements */
    props.setProperty("prepareThreshold", "0");
    /*
     * Tell the driver manager to connect to the database specified with the URL.
     * This may throw an SQLException.
     */
    return java.sql.DriverManager.getConnection(url, props);
}
```

Section 26.3: Connecting with javax.sql.DataSource using a connection pool

It is common to use `javax.sql.DataSource` with JNDI in application server containers, where you register a data source under a name and look it up whenever you need a connection.

This is code that demonstrates how data sources work:

```
/**
 * Create a data source with connection pool for PostgreSQL connections
 * @param url the JDBC URL to connect to. Must start with "jdbc:postgresql:"
 * @param user the username for the connection
 * @param password the password for the connection
 * @return a data source with the correct properties set
 */
private static javax.sql.DataSource createDataSource(String url, String user, String password)
{
    /* use a data source with connection pooling */
    org.postgresql.ds.PGPoolingDataSource ds = new org.postgresql.ds.PGPoolingDataSource();
    ds.setUrl(url);
    ds.setUser(user);
    ds.setPassword(password);
    /* the connection pool will have 10 to 20 connections */
    ds.setInitialConnections(10);
    ds.setMaxConnections(20);
    /* use SSL connections without checking server certificate */
    ds.setSslMode("require");
    ds.setSslfactory("org.postgresql.ssl.NonValidatingFactory");

    return ds;
}
```

Once you have created a data source by calling this function, you would use it like this:

```
/* get a connection from the connection pool */
java.sql.CONNECTION conn = ds.getConnection();
```

```
/* 执行一些操作 */

/* 将连接交还给连接池 - 它不会被关闭 */
conn.close();
```



```
/* do some work */

/* hand the connection back to the pool - it will not be closed */
conn.close();
```

第27章：PostgreSQL高可用性

第27.1节：PostgreSQL中的复制

- **配置主服务器**
  - **要求：**
    - 用于复制活动的复制用户
    - 存储WAL归档的目录
  - **创建复制用户**

```
CREATEUSER -U postgres replication -P -c 5 --replication
```

```
+ 选项 -P 会提示您输入新密码
+ OPTION -c 用于最大连接数。 5个连接对于复制来说足够了
  + -复制将会授予该用户复制权限
```

  - **在数据目录中创建归档目录**

```
mkdir $PGDATA/archive
```

  - **编辑 pg_hba.conf 文件**

这是基于主机的认证文件，包含客户端认证的设置。添加以下条目：

```
#主机类型    数据库名称     用户名       主机名/IP      认证方法
      host          replication       replication      <从机IP>/32     md5
```

  - **编辑 postgresql.conf 文件**

这是 PostgreSQL 的配置文件。

wal_level = hot_standby

该参数决定从服务器的行为。

```
   `hot_standby` 记录了 接受 只读查询 所需的内容 在 从服务器 上。

   `streaming` 记录了 只需在从服务器上应用 WAL 所需的内容。

`archive` 记录了归档所需的内容。
```

archive_mode=ON

该参数允许使用archive_command 参数将 WAL 段发送到归档位置。

```
archive_command = 'test ! -f /path/to/archivedir/%f && cp %p /path/to/archivedir/%f'
```

基本上，上述archive_command 的作用是将 WAL 段复制到归档目录。

wal_senders = 5 这是 WAL 发送进程的最大数量。

现在重启主服务器。

Chapter 27: PostgreSQL High Availability

Section 27.1: Replication in PostgreSQL

- **Configuring the Primary Server**
  - **Requirements:**
    - Replication User for replication activities
    - Directory to store the WAL archives
  - **Create Replication user**

```
CREATEUSER -U postgres replication -P -c 5 --replication
```

```
  + OPTION -P will prompt you FOR NEW PASSWORD
  + OPTION -c IS FOR maximum connections. 5 connections are enough FOR replication
  + -replication will GRANT replication PRIVILEGES TO the USER
```

  - **Create a archive directory in data directory**

```
mkdir $PGDATA/archive
```

  - **Edit the pg_hba.conf file**

This is host base authentication file, contains the setting for client autherntication. Add below entry:

```
     #hosttype    database_name     user_name       hostname/IP      method
      host        replication       replication     <slave-IP>/32    md5
```

  - **Edit the postgresql.conf file**

This is the configuration file of PostgreSQL.

wal_level = hot_standby

This parameter decides the behavior of slave server.

```
   `hot_standby` logs what IS required TO accept READ ONLY queries ON slave SERVER.

   `streaming` logs what IS required TO just apply the WAL's on slave.

   `archive` which logs what is required for archiving.
```

archive_mode=ON

This parameters allows to send WAL segments to archive location using archive_command parameter.

archive_command = 'test ! -f /path/to/archivedir/%f && cp %p /path/to/archivedir/%f'

Basically what above archive_command does is it copies the WAL segments to archive directory.

wal_senders = 5 This is maximum number of WAL sender processes.

Now restart the primary server.

- **将主服务器备份到从服务器**

  在对服务器进行更改之前，请先停止主服务器。

> 重要提示：在完成所有配置和备份步骤之前，不要重新启动服务。您必须使备用服务器处于准备作为备份服务器的状态。这意味着所有配置设置必须到位，且数据库必须已经同步。否则，流复制将无法启动。

- **现在运行 pg_basebackup 工具**

  pg_basebackup工具将主服务器数据目录中的数据复制到从服务器数据目录。

```
$ pg_basebackup -h <主服务器IP> -D /var/lib/postgresql/<版本>/main -U replication -v -P --
xlog-method=stream
```

-D: 这告诉 pg_basebackup 在哪里进行初始备份

-h: 指定 系统 查找 主服务器 的位置

-xlog-method=stream: 这将 强制 pg_basebackup 打开另一个连接并在备份运行时流式传输足够的 xlog。

它 还 确保可以启动新的备份，无需回退 使用 归档。

- **配置备用服务器**

  要配置备用服务器，您需要编辑 postgresql.conf 并创建一个名为 recovery.conf 的新配置文件。

  hot_standby = ON

  这指定了在恢复期间是否允许运行查询

  - **创建 recovery.conf 文件**

    standby_mode = ON

    设置主服务器的连接字符串。将其替换为主服务器的外部 IP 地址。将其替换为名为 replication 的用户密码

    `primary_conninfo = 'host= port=5432 user=replication password='

    （可选）设置触发文件位置：

    trigger_file = '/tmp/postgresql.trigger.5432'

    您指定的 trigger_file 路径是您想要系统切换到备用服务器时添加文件的位置。文件的存在会“触发”故障切换。或者，您也可以使用 pg_ctl promote 命令来触发故障切换。

- **Backing up the primay server to the slave server**

  Before making changes on the server stop the primary server.

> Important: Don't start the service again until all configuration and backup steps are complete. You must bring up the standby server in a state where it is ready to be a backup server. This means that all configuration settings must be in place and the databases must be already synchronized. Otherwise, streaming replication will fail to start`

- **Now run the pg_basebackup utility**

  pg_basebackup utility copies the data from primary server data directory to slave data directory.

```
$ pg_basebackup -h <PRIMARY IP> -D /var/lib/postgresql/<VERSION>/main -U replication -v -P --
xlog-method=stream
```

-D: This IS tells pg_basebackup WHERE TO the initial backup

-h: Specifies the SYSTEM WHERE TO look FOR the PRIMARY SERVER

-xlog-method=stream: This will FORCE the pg_basebackup TO open another CONNECTION AND stream enough xlog WHILE backup IS running.
It ALSO ensures that fresh backup can be started WITHOUT failing back TO USING an archive.

- **Configuring the standby server**

  To configure the standby server, you'll edit postgresql.conf and create a new configuration file named recovery.conf.

  hot_standby = ON

  This specifies whether you are allowed to run queries while recovering

  - **Creating recovery.conf file**

    standby_mode = ON

    Set the connection string to the primary server. Replace with the external IP address of the primary server. Replace with the password for the user named replication

    `primary_conninfo = 'host= port=5432 user=replication password='

    (Optional) Set the trigger file location:

    trigger_file = '/tmp/postgresql.trigger.5432'

    The `trigger_file` path that you specify is the location where you can add a file when you want the system to fail over to the standby server. The presence of the file "triggers" the failover. Alternatively, you can use the pg_ctl promote command to trigger failover.

- **启动备用服务器**

  您现在已经准备就绪，可以启动备用服务器了

归属

本文主要来源于并归属于《如何设置PostgreSQL以实现高可用性和热备份复制》，格式和示例上有少量修改，并删除了一些内容。该来源发布于知识共享公共许可证3.0，现保留于此。

- **Start the standby server**

  You now have everything in place and are ready to bring up the standby server

Attribution

This article is substantially derived from and attributed to How to Set Up PostgreSQL for High Availability and Replication with Hot Standby, with minor changes in formatting and examples and some text deleted. The source was published under the Creative Commons Public License 3.0, which is maintained here.

第28章：扩展 dblink 和 postgres_fdw

第28.1节：扩展 FDW

FDW 是 dblink 的一种实现，更加实用，使用方法如下：

1. 创建扩展：

```
CREATE EXTENSION postgres_fdw;
```

2. 创建服务器：

```
CREATE SERVER name_srv FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'hostname',
dbname 'bd_name', port '5432');
```

3. 为Postgres服务器创建用户映射

```
为 postgres 服务器 name_srv 创建用户映射 选项(用户 'postgres', 密码 'password');
```

4. 创建外部表：

```
创建外部表 table_foreign (id 整数, 代码 变长字符)
服务器 name_srv 选项(模式名 'schema', 表名 'table');
```

5. 像在数据库中一样使用此外部表：

```
选择 * 从 table_foreign;
```

第28.2节：外部数据包装器

要访问服务器数据库的完整模式而非单个表，请按照以下步骤操作：

1. 创建扩展：

```
CREATE EXTENSION postgres_fdw;
```

2. 创建服务器：

```
创建服务器 server_name 外部数据包装器 postgres_fdw 选项 (主机 'host_ip',
数据库名 'db_name', 端口 'port_number');
```

3.创建用户映射：

```
为当前用户创建用户映射
服务器 server_name
选项 (用户 'user_name', 密码 'password');
```

4. 创建新模式以访问服务器数据库的模式：

```
创建模式 schema_name;
```


Chapter 28: EXTENSION dblink and postgres_fdw

Section 28.1: Extention FDW

FDW is an implimentation of dblink it is more helpful, so to use it:

1. Create an extention:

```
CREATE EXTENSION postgres_fdw;
```

2. Create SERVER:

```
CREATE SERVER name_srv FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'hostname',
dbname 'bd_name', port '5432');
```

3. Create user mapping for postgres server

```
CREATE USER MAPPING FOR postgres SERVER name_srv OPTIONS(USER 'postgres', PASSWORD 'password');
```

4. Create foreign table:

```
CREATE FOREIGN TABLE table_foreign (id INTEGER, code CHARACTER VARYING)
SERVER name_srv OPTIONS(schema_name 'schema', table_name 'table');
```

5. use this foreign table like it is in your database:

```
SELECT * FROM table_foreign;
```

Section 28.2: Foreign Data Wrapper

To access complete schema of server db instead of single table. Follow below steps:

1. Create EXTENSION :

```
CREATE EXTENSION postgres_fdw;
```

2. Create SERVER :

```
CREATE SERVER server_name FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'host_ip',
dbname 'db_name', port 'port_number');
```

3. Create USER MAPPING:

```
CREATE USER MAPPING FOR CURRENT_USER
SERVER server_name
OPTIONS (USER 'user_name', PASSWORD 'password');
```

4. Create new schema to access schema of server DB:

```
CREATE SCHEMA schema_name;
```

5. 导入服务器模式：

```
导入 外部模式 schema_name_to_import_from_remote_db
     来自服务器 server_name
     进入 schema_name;
```

6. 访问服务器模式的任意表：

```
选择 * 来自 schema_name.table_name;
```

这可以用来访问远程数据库的多个模式。

第28.3节：dblink扩展

dblink扩展是一种连接另一个数据库并对该数据库进行操作的技术，要实现此操作，您需要：

1-创建dblink扩展：

```
CREATE EXTENSION dblink;
```

2-执行您的操作：

例如，从另一个数据库中的另一张表选择某些属性：

```
SELECT * FROM
dblink ('dbname = bd_distance port = 5432 host = 10.6.6.6 user = username
password = passw@rd', 'SELECT id, code FROM schema.table')
AS newTable(id INTEGER, code CHARACTER VARYING);
```


5. Import server schema:

```
IMPORT FOREIGN SCHEMA schema_name_to_import_from_remote_db
FROM SERVER server_name
INTO schema_name;
```

6. Access any table of server schema:

```
SELECT * FROM schema_name.table_name;
```

This can be used to access multiple schema of remote DB.

Section 28.3: Extention dblink

dblink EXTENSION is a technique to connect another database and make operation of this database so to do that you need:

1-Create a dblink extention:

```
CREATE EXTENSION dblink;
```

2-Make your operation:

For exemple Select some attribute from another table in another database:

```
SELECT * FROM
dblink ('dbname = bd_distance port = 5432 host = 10.6.6.6 user = username
password = passw@rd', 'SELECT id, code FROM schema.table')
AS newTable(id INTEGER, code CHARACTER VARYING);
```

第29章：Postgres技巧与窍门

第29.1节：Postgres中的DATEADD替代方法

- `SELECT CURRENT_DATE + '1 day'::INTERVAL`
- `SELECT '1999-12-11'::TIMESTAMP + '19 days'::INTERVAL`
- `SELECT '1 month'::INTERVAL + '1 month 3 days'::INTERVAL`

第29.2节：列的逗号分隔值

```
SELECT
    STRING_AGG(<TABLE_NAME>.<COLUMN_NAME>, ',')
FROM
    <SCHEMA_NAME>.<TABLE_NAME> T
```

第29.3节：从postgres表中删除重复记录

```
DELETE
    FROM <SCHEMA_NAME>.<Table_NAME>
WHERE
ctid NOT IN
        (
        SELECT
            MAX(ctid)
        FROM
            <SCHEMA_NAME>.<TABLE_NAME>
        GROUP BY
            <SCHEMA_NAME>.<TABLE_NAME>.*
        )
;
```

第29.4节：两个表之间带连接的更新查询PostgreSQL不支持更新查询中的连接时的替代方案

```
    UPDATE <SCHEMA_NAME>.<TABLE_NAME_1> AS A
    SET <COLUMN_1> = TRUE
    FROM <SCHEMA_NAME>.<TABLE_NAME_2> AS B
    WHERE
A.<COLUMN_2> = B.<COLUMN_2> AND
A.<COLUMN_3> = B.<COLUMN_3>
```

第29.5节：两个日期时间戳按月和按年计算的差异

两个日期（时间戳）按月的差异

```
SELECT
    (
        (DATE_PART('year', AgeonDate) - DATE_PART('year', tmpdate)) * 12
        +
        (DATE_PART('月', AgeonDate) - DATE_PART('月', tmpdate))
    )
FROM dbo."Table1"
```


Chapter 29: Postgres Tip and Tricks

Section 29.1: DATEADD alternative in Postgres

- `SELECT CURRENT_DATE + '1 day'::INTERVAL`
- `SELECT '1999-12-11'::TIMESTAMP + '19 days'::INTERVAL`
- `SELECT '1 month'::INTERVAL + '1 month 3 days'::INTERVAL`

Section 29.2: Comma separated values of a column

```
SELECT
    STRING_AGG(<TABLE_NAME>.<COLUMN_NAME>, ',')
FROM
    <SCHEMA_NAME>.<TABLE_NAME> T
```

Section 29.3: Delete duplicate records from postgres table

```
DELETE
    FROM <SCHEMA_NAME>.<Table_NAME>
WHERE
    ctid NOT IN
        (
        SELECT
            MAX(ctid)
        FROM
            <SCHEMA_NAME>.<TABLE_NAME>
        GROUP BY
            <SCHEMA_NAME>.<TABLE_NAME>.*
        )
;
```

Section 29.4: Update query with join between two tables alternative since Postresql does not support join in update query

```
    UPDATE <SCHEMA_NAME>.<TABLE_NAME_1> AS A
    SET <COLUMN_1> = TRUE
    FROM <SCHEMA_NAME>.<TABLE_NAME_2> AS B
    WHERE
        A.<COLUMN_2> = B.<COLUMN_2> AND
        A.<COLUMN_3> = B.<COLUMN_3>
```

Section 29.5: Difference between two date timestamps month wise and year wise

Monthwise difference between two dates(timestamp)

```
SELECT
    (
        (DATE_PART('year', AgeonDate) - DATE_PART('year', tmpdate)) * 12
        +
        (DATE_PART('month', AgeonDate) - DATE_PART('month', tmpdate))
    )
FROM dbo."Table1"
```

两个日期（时间戳）之间按年份的差异

```
SELECT (DATE_PART('年', AgeonDate) - DATE_PART('年', tmpdate)) FROM dbo."Table1"
```

第29.6节：将表数据从一个数据库复制/移动/传输到具有相同模式的另一个数据库表的查询

首先执行

```
CREATE EXTENSION DBLINK;
```

然后

```
INSERT INTO
    <SCHEMA_NAME>.<TABLE_NAME_1>
选择 *
FROM
DBLINK(
    'HOST=<IP-ADDRESS> USER=<USERNAME> PASSWORD=<PASSWORD> DBNAME=<DATABASE>',
    'SELECT * FROM <SCHEMA_NAME>.<TABLE_NAME_2>')
    作为 <TABLE_NAME>
    (
    <COLUMN_1> <DATATYPE_1>,
    <COLUMN_1> <DATATYPE_2>,
    <COLUMN_1> <DATATYPE_3>
    );
```


Yearwise difference between two dates(timestamp)

```
SELECT (DATE_PART('year', AgeonDate) - DATE_PART('year', tmpdate)) FROM dbo."Table1"
```

Section 29.6: Query to Copy/Move/Transafer table data from one database to other database table with same schema

First Execute

```
CREATE EXTENSION DBLINK;
```

Then

```
INSERT INTO
    <SCHEMA_NAME>.<TABLE_NAME_1>
SELECT *
FROM
    DBLINK(
    'HOST=<IP-ADDRESS> USER=<USERNAME> PASSWORD=<PASSWORD> DBNAME=<DATABASE>',
    'SELECT * FROM <SCHEMA_NAME>.<TABLE_NAME_2>')
    AS <TABLE_NAME>
    (
    <COLUMN_1> <DATATYPE_1>,
    <COLUMN_1> <DATATYPE_2>,
    <COLUMN_1> <DATATYPE_3>
    );
```

鸣谢

非常感谢所有来自Stack Overflow Documentation的人员提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

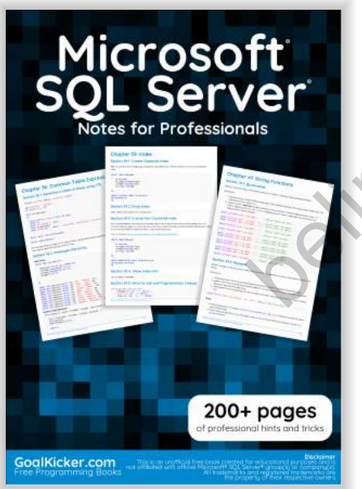
艾莉森·S	第1章和第11章
安德鲁·奇霍茨基	第1章和第15章
ankidaemon	第23章
AstraSerg	第25章
本	第1、20和22章
本·H	第1、2、14、15、20、21、23和29章
bignose	第1章
bilelovitch	第20章和第24章
Blackus	第20章
brichins	第25章
chalitha geekiyanage	第8章和第18章
commonSenseCode	第10章
Dakota Wagner	第1章
丹尼尔·莱昂斯	第20章和第23章
德米尔詹·切莱比	第1章
德米特里·戈尔德林	第1章
e4c5	第1章、第4章、第20章和第23章
evuez	第16章
戈尔曼	第15章
gpdude_	第8章和第27章
格雷格	第20章和第25章
雅库布·费迪查克	第12章
杰森赵	第1章
杰斐逊	第4章
jgm	第10章
约瑟夫	第11章
凯文·西尔维斯特	第12章
金	第3、4、8和20章
基兰·库马尔·马塔姆	第22章
基里尔·索科洛夫	第11章
柯克·罗伊巴尔	第1章
劳伦茨·阿尔贝	第15、25和26章
leeor	第4、8和9章
穆罕默德·纳瓦斯	第6章
莫卡迪利昂	第1和7章
内森尼尔·韦斯布罗特	第8章
努里·塔斯德米尔	第3章
帕特里克	第3章、第11章和第27章
重启	第20章
里娅·班萨尔	第28章
skj123	第21章和第29章
塔金德尔	第19章
汤姆·格肯	第3章
乌德雷·纳蒂	第18章
user_0	第二章
袁村	第8、13、15和17章
wOwhOw	第17章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Alison S	Chapters 1 and 11
AndrewCichocki	Chapters 1 and 15
ankidaemon	Chapter 23
AstraSerg	Chapter 25
Ben	Chapters 1, 20 and 22
Ben H	Chapters 1, 2, 14, 15, 20, 21, 23 and 29
bignose	Chapter 1
bilelovitch	Chapters 20 and 24
Blackus	Chapter 20
brichins	Chapter 25
chalitha geekiyanage	Chapters 8 and 18
commonSenseCode	Chapter 10
Dakota Wagner	Chapter 1
Daniel Lyons	Chapters 20 and 23
Demircan Celebi	Chapter 1
Dmitri Goldring	Chapter 1
e4c5	Chapters 1, 4, 20 and 23
evuez	Chapter 16
Goerman	Chapter 15
gpdude_	Chapters 8 and 27
greg	Chapters 20 and 25
Jakub Fedyczak	Chapter 12
jasonszhao	Chapter 1
Jefferson	Chapter 4
jgm	Chapter 10
joseph	Chapter 11
Kevin Sylvestre	Chapter 12
KIM	Chapters 3, 4, 8 and 20
KIRAN KUMAR MATAM	Chapter 22
Kirill Sokolov	Chapter 11
Kirk Roybal	Chapter 1
Laurenz Albe	Chapters 15, 25 and 26
leeor	Chapters 4, 8 and 9
Mohamed Navas	Chapter 6
Mokadillion	Chapters 1 and 7
Nathaniel Waisbrot	Chapter 8
Nuri Tasdemir	Chapter 3
Patrick	Chapters 3, 11 and 27
Reboot	Chapter 20
Riya Bansal	Chapter 28
skj123	Chapters 21 and 29
Tajinder	Chapter 19
Tom Gerken	Chapter 3
Udlei Nati	Chapter 18
user_0	Chapter 2
Vao Tsun	Chapters 8, 13, 15 and 17
wOwhOw	Chapter 17

你可能也喜欢



You may also like