第八章 管理PG安全

在第7章，编写存储过程中，我们学习了存储过程和编写服务器端代码。在介绍了许多重要的主题之后，现在是时候转向PostgreSQL的安全问题了。在这里，我们将学习如何保证服务器的安全，并配置权限以避免安全宣扬。

本章将涉及以下主题。

管理网络安全

挖掘行级安全（RLS）

检查权限

重新分配对象和放弃用户

在本章结束时，我们将能够专业地配置PostgreSQL安全。现在让我们从管理网络安全开始吧。

1.管理网络安全

在继续讨论现实世界的实际例子之前，让我们简单地关注一下我们将要处理的各种安全层。在处理安全问题时，为了有组织地处理与安全有关的问题，牢记这些层次是有意义的。

以下是我的思想模型:

- 绑定地址：postgresql.conf文件中的listen_addresses
- 基于主机的访问控制：pg_hba.conf文件
- 实体级权限：用户、角色、数据库创建、登录和复制
- 数据库级权限：连接、创建模式等等
- 模式级权限：使用模式和在模式内创建对象

- 表级权限：选择、插入、更新，以及更多
- 列级权限：允许或限制对列的访问
- RLS: 限制对行的访问

为了读取一个值，PostgreSQL 必须确保我们在每个级别上都有足够的权限。整个权限链必须正确。多年来，我的小模型一直很好地帮助我在实际应用程序中反复调试与安全相关的问题。它有望帮助您使用更系统的方法，从而提高安全性。

2.了解绑定地址和连接

在配置PostgreSQL服务器时，首先需要做的一件事是定义远程访问。默认情况下，PostgreSQL不接受远程连接。这里重要的是，PostgreSQL甚至不拒绝连接，因为它根本不监听端口。如果我们尝试连接，错误信息实际上将来自操作系统，因为PostgreSQL根本不关心。

假设在192.168.0.123上有一个使用默认配置的数据库服务器，将发生以下情况。

```
iMac:~ hs$ telnet 192.168.0.123 5432
Trying 192.168.0.123...
telnet: connect to address 192.168.0.123: Connection refused
telnet: Unable to connect to remote host
```

Telnet试图在5432端口建立一个连接，但立即被远程拒绝。从外面看，好像PostgreSQL根本就没有运行。

在postgresql.conf文件中可以找到成功的关键。

```
# - Connection Settings -
# listen_addresses = 'localhost'
# what IP address(es) to listen on;
# comma-separated list of addresses;
# defaults to 'localhost'; use '*' for all
# (change requires restart)
```

listen_addresses设置将告诉PostgreSQL要听哪些地址。从技术上讲，这些地址是绑定地址。这实际上是什么意思呢？假设我们的机器上有四个网卡。我们可以监听，比如，其中三个互联网协议（IP）地址。PostgreSQL会考虑到对这三块网卡的请求，而不会去监听第四块网卡。该端口被简单地关闭。

> 我们必须把我们服务器的IP地址放到listen_addresses中，而不是客户的IP。

如果我们在PostgreSQL中加入*，我们将监听分配给你的机器的每个IP。

> 请记住，改变listen_addresses需要重新启动PostgreSQL服务。如果不重启，就不能临时改变它。

然而，还有更多与连接管理有关的设置是需要了解，并且非常重要。它们如下：

```
#port = 5432
 # (change requires restart)
max_connections = 100
 # (change requires restart)
# Note: Increasing max_connections costs ~400 bytes of
# shared memory per
# connection slot, plus lock space
# (see max_locks_per_transaction)
#superuser_reserved_connections = 3
 # (change requires restart)
#unix_socket_directories = '/tmp'
```

```
 # comma-separated list of directories
 # (change requires restart)
#unix_socket_group = ''
 # (change requires restart)
#unix_socket_permissions = 0777
 # begin with 0 to use octal notation
 # (change requires restart)
```

首先，PostgreSQL只监听一个传输控制协议（TCP）端口，其默认值是5432。请记住，PostgreSQL将只监听一个端口。每当有请求进入时，Postmaster将分叉并创建一个新的进程来处理该连接。 默认情况下，最多允许100个正常连接。在此基础上，为超级用户保留3个额外的连接。这意味着，我们可以有97个连接，加上3个超级用户，或者100个超级用户连接。 为了开始工作，我们将首先看一下连接和性能

> 请注意，这些与连接有关的设置也需要重新启动。其原因是，共享内存分配了一个静态的内存量，不能临时改变。

3.检查连接和性能

在做咨询的时候，很多人问我提高连接限制是否会对一般的性能产生影响。答案是不多，因为由于上下文切换，总是有一些开销。至于有多少个连接，这没有什么区别。然而，有区别的是开放快照的数量。开放的快照越多，数据库方面的开销就越大。换句话说，我们可以廉价地增加max_connections。在下一节，我们将学习如何出于安全原因避免使用TCP。

> 如果你对一些真实世界的数据感兴趣，可以考虑看一下 https://www.cybertec-postgresql.com/en/max_connections-performance-impacts/

4.生活在没有 TCP 的世界里

在某些情况下，我们可能不希望使用网络。经常发生的情况是，无论如何，数据库只能与本地应用程序对话。也许我们的PostgreSQL数据库已经和我们的应用程序一起被运送了，或者我们只是不想承担使用网络的风险。在这种情况下，Unix套接字是你需要的。Unix套接字是一种无网络的通信方式。你的应用程序可以通过Unix套接字在本地进行连接，而不向外界暴露任何信息。

然而，我们需要的是一个目录。默认情况下，PostgreSQL 将使用 /tmp 目录。但是，如果每台机器运行着多个数据库服务器，则每个服务器都需要一个单独的数据目录来存放

除了安全之外，不使用网络可能是个好主意的原因还有很多。其中一个原因是性能。使用 Unix 套接字比通过环回设备 (127.0.0.1) 快很多。如果这听起来令人惊讶，请不要担心；它适用于很多人。但是，如果您只运行非常小的查询，则不应低估实际网络连接的开销。

为了说明这到底意味着什么，我包括了一个简单的基准。

我们将创建一个script.sql文件。这是一个简单的脚本，用于创建一个随机数并进行选择。这是最简单的语句。没有什么比获取一个数字更简单的了。

因此，让我们在一台普通的笔记本电脑上运行这个简单的基准测试。为此，我们将写一个叫做script.sql的小东西。它将被下面的基准测试所使用。

```
[hs@linuxpc ~]$ cat /tmp/script.sql
SELECT 1
```

然后，我们可以简单地运行 pgbench 来一遍又一遍地执行 SQL。 -f 选项允许我们将 SQL 的名称传递给脚本。 -c 10 表示我们希望处于活动状态 5 秒 (-T 5)内有 10 个并发连接。基准测试以 postgres 用户身份运行，并且应该使用默认情况下应该存在的 postgres 数据库。请注意，以下示例适用于 Red Hat Enterprise Linux (RHEL) 衍生产品。基于 Debian 的系统将使用不同的路径：

```
[hs@linuxpc ~]$ pgbench -f /tmp/script.sql -c 10 -T 5 -U postgres postgres 2>
/dev/null
transaction type: /tmp/script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
duration: 5 s
number of transactions actually processed: 871407
latency average = 0.057 ms
tps = 174278.158426 (including connections establishing)
tps = 174377.935625 (excluding connections establishing)
```

我们可以看到，没有向pgbench传递主机名，因此该工具在本地连接到Unix套接字，并尽可能快地运行脚本。在这个四核英特尔盒子上，系统能够实现每秒约174,000次交易

如果加上-h localhost会怎样？性能将发生变化，正如你在下面的代码段中看到的那样。

```
[hs@linuxpc ~]$ pgbench -f /tmp/script.sql  -h localhost -c 10 -T 5 -U postgres
postgres 2> /dev/null
transaction type: /tmp/script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
duration: 5 s
number of transactions actually processed: 535251
latency average = 0.093 ms
tps = 107000.872598 (including connections establishing)
tps = 107046.943632 (excluding connections establishing)
```

吞吐量将像石头一样下降到每秒107000个交易。这种差异显然与网络开销有关。

> 通过使用-j选项（分配给pgbench的线程数），我们可以从我们的系统中挤出一些更多的事务。然而，在我们的情况下，这并没有改变基准测试的整体情况。在其他测试中，它确实如此，因为如果你不提供足够的CPU能力，pgbench可能是一个真正的瓶颈

正如我们所看到的，网络不仅可以是一个安全问题，也是一个性能问题。然而，性能并不是唯一重要的方面。 由于我们在这里主要讨论的是安全问题，因此下一个难题是关于 pg_hba.conf 的。

5.管理pg_hba.conf

在配置完绑定地址后，我们可以进入下一个层次。 pg_hba.conf文件将告诉PostgreSQL如何对通过网络来的人进行认证。一般来说，pg_hba.conf文件的条目有如下布局。

```
# local DATABASE USER METHOD [OPTIONS]
# host DATABASE USER ADDRESS METHOD [OPTIONS]
# hostssl DATABASE USER ADDRESS METHOD [OPTIONS]
# hostnossl DATABASE USER ADDRESS METHOD [OPTIONS]
# hostgssenc DATABASE USER ADDRESS METHOD [OPTIONS]
# hostnogssenc DATABASE USER ADDRESS METHOD [OPTIONS]
```

有四种类型的规则可以放入pg_hba.conf文件中:

- local:这可以用来配置本地Unix套接字连接。
- host:这可以用于安全套接字层（SSL）和非SSL连接。

- hostssl:这只对SSL连接有效。要利用这个选项，SSL必须被编译到服务器中，如果我们使用PostgreSQL的预包装版本，就会出现这种情况。除此之外，ssl = on必须在postgresql.conf文件中设置。这个文件在服务器启动时被调用。
- hostnossl:这适用于非SSL连接。
- hostgssenc:这个规则定义了只有在可以进行GSSAPI加密的情况下才会创建一个连接。否则就会失败。
- hostnogssenc:这条规则与hostgssenc完全相反。

可以将规则列表合并到 pg_hba.conf 文件中。这是一个例子：

```
# TYPE DATABASE USER ADDRESS METHOD
# "local" is for Unix domain socket connections only
local all all trust
# IPv4 local connections:
host all all 127.0.0.1/32 trust
# IPv6 local connections:
host all all ::1/128 trust
```

您可以看到三个简单的规则：

- 本地记录说，来自本地Unix套接字的所有数据库的用户都要被信任。信任方法意味着不需要向服务器发送密码，人们可以直接登录
- 另外两条规则说，同样适用于来自127.0.0.1 localhost和::1/128的连接，后者是一个IPv6地址

由于没有密码的连接肯定不是远程访问的最佳选择，PostgreSQL提供了各种认证方法，可以用来灵活地配置pg_hba.conf文件。下面是一个可能的认证方法的列表:

- trust：这允许认证而不提供密码。所需的用户必须在PostgreSQL端可用。
- reject：连接将被拒绝。
- md5和password：连接可以使用密码来创建。md5意味着密码在通过网络发送时被加密了。在密码的情况下，凭证是以明文形式发送的，在现代系统中不应再这样做，md5已不再被认为是安全的。你应该使用scram-sha-256来代替PostgreSQL 10及以后的版本。
- scram-sha-256：这个设置是md5的继承者，使用的哈希值比以前的版本要安全得多。
- gss and sspi：这使用通用安全服务应用程序接口（GSSAPI）或安全支持提供者接口（SSPI）认证。这只可能用于TCP/IP连接。这里的想法是允许单点登录。
- ident：这通过联系客户的身份服务器获得客户的操作系统用户名，并检查它是否与请求的数据库用户名相匹配。
- Peer: 假设我们在Unix上以abc的身份登录。如果peer被启用，我们只能以abc的身份登录PostgreSQL。如果我们试图改变用户名，我们将被拒绝。美中不足的是，abc不需要密码就可以进行验证。这里的想法是，只有数据库管理员可以在Unix系统上登录数据库，而不是其他只是拥有密码的人或同一台机器上的Unix帐户。这只对本地连接起作用，
- pam。这使用了可插拔认证模块（PAM）。如果你想使用PostgreSQL不提供的认证方式，这一点特别重要。要使用PAM，在你的Linux系统上创建一个叫做/etc/pam.d/postgresql的文件，并把你打算使用的PAM模块放到配置文件中。 使用PAM，我们甚至可以针对不太常见的组件进行认证。然而，它也可以用来连接到活动目录等。
- ldap：这个配置允许你使用轻量级目录访问协议（LDAP）进行认证。注意，PostgreSQL只要求LDAP进行认证；如果一个用户只存在于LDAP端，而不存在于PostgreSQL端，你就不能登录。你还应该注意，PostgreSQL必须知道你的LDAP服务器在哪里。 所有这些信息都必须存储在pg_hba.conf文件中，如官方文档https://www.postgresql.org/docs/10/static/auth-methods.html#AUTH-LDAP.
- radius：远程认证拨入用户服务（RADIUS）是一种执行单点登录的方法。同样，参数是使用配置选项传递的。
- cert：这种认证方法使用SSL客户证书来执行认证，因此只有在使用SSL的情况下才有可能。这里的优点是不需要发送密码。证书的CN属性将与请求的数据库用户名进行比较，如果它们匹配，将允

许登录。可以使用一个地图来允许用户的映射

规则可以简单地一个接一个地列出。这里重要的是，顺序确实有区别，如下面的例子所示。

```
host all all 192.168.1.0/24 scram-sha-256
host all all 192.168.1.54/32 reject
```

当PostgreSQL浏览pg_hba.conf文件时，它将使用第一个匹配的规则。因此，如果我们的请求来自192.168.1.54，在我们进入第二条规则之前，第一条规则总是会匹配。这意味着，如果密码和用户都是正确的，192.168.1.54将能够登录；因此，第二条规则是没有意义的。

如果我们想排除IP，我们需要确保这两条规则被调换。在下一节中，我们将看一下SSL，以及你如何能够轻松地使用它

6.处理ssl

PostgreSQL允许在服务器和客户端之间的传输是加密的。加密是非常有益的，特别是在我们进行长距离通信的时候。SSL提供了一个简单而安全的方法，确保没有人能够监听你的通信。

在本节中，我们将学习如何设置 SSL：

- 首先要做的是在服务器启动时，在postgresql.conf文件中将ssl参数设置为on。在下一步，我们可以把SSL证书放到$PGDATA目录中。如果我们不希望证书放在其他目录中，我们需要改变以下参数

```
#ssl_cert_file = 'server.crt' # (change requires restart)
#ssl_key_file = 'server.key' # (change requires restart)
#ssl_ca_file = '' # (change requires restart)
#ssl_crl_file = '' # (change requires restart)
```

- 如果我们想使用自签名的证书，我们需要执行以下步骤

```
openssl req -new -text -out server.req
```

回答OpenSSL所问的问题。确保我们输入本地主机名作为公共名称。我们可以把密码留空。这个调用将生成一个受密码保护的密钥；它不接受长度少于四个字符的密码。

- 要删除口令（如果你想自动启动服务器，你必须这样做），请运行以下代码。

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

- 输入旧的口令来解锁现有的密钥。现在，使用下面的代码把证书变成一个自签名的证书，并把密钥和证书复制到服务器要找的地方

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
```

- 这样做之后，确保这些文件有正确的权限。

```
chmod og-rwx server.key
```

- 一旦在pg_hba.conf文件中放入适当的规则，我们就可以使用SSL连接到你的服务器。为了验证我们是否在使用SSL，可以考虑查看pg_stat_ssl函数。它将告诉我们每一个连接以及它是否使用了SSL，

并将提供一些关于加密的重要信息

```
test=# \d pg_stat_ssl
 View "pg_catalog.pg_stat_ssl"
 Column | Type | Collation | Nullable | Default
---------------+---------+-----------+----------+---------
 pid | integer | | |
 ssl | boolean | | |
 version | text | | |
 cipher | text | | |
 bits | integer | | |
 compression | boolean | | |
 client_dn | text | | |
 client_serial | numeric | | |
 issuer_dn | text | | |
```

- 如果一个进程的ssl字段包含true，PostgreSQL会做我们所期望的事情

```
postgres=# select * from pg_stat_ssl;
-[ RECORD 1 ]
----------------------------
pid | 20075
ssl | t
version | TLSv1.2
cipher | ECDHE-RSA-AES256-GCM-SHA384
bits | 256
compression | f
clientdn |
client_serial |
issuer_dn |
```

一旦你配置好了SSL，就该看看实例级的安全问题了

7.处理实例级安全

到目前为止，我们已经配置了绑定地址，并且告诉PostgreSQL在哪些IP范围内使用哪种认证方式。到现在为止，这些配置纯粹是与网络有关的。

在这一节，我们可以把注意力转移到实例级的权限上。最重要的是要知道PostgreSQL中的用户是否存在于实例级别上。如果我们创建了一个用户，它不仅仅是在一个数据库中可见，它可以被所有的数据库看到。一个用户可能只有访问一个数据库的权限，但用户基本上是在实例级创建的。

对于那些刚接触PostgreSQL的人来说，还有一件事你应该记住：用户和角色是同样的东西。CREATE ROLE和CREATE USER子句有不同的默认值（唯一的区别是，角色默认不获得LOGIN属性），但是，归根结底，用户和角色是一样的。因此，CREATE ROLE和CREATE USER子句支持非常相同的语法。下面的列表包含了CREATE USER的语法概述

```
postgres=# \h CREATE USER
Command: CREATE USER
Description: define a new database role
Syntax:
CREATE USER name [ [ WITH ] option [ ... ] ]
where option can be:
 SUPERUSER | NOSUPERUSER
```

```
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
  | BYPASSRLS | NOBYPASSRLS
  | CONNECTION LIMIT connlimit
  | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
  | VALID UNTIL 'timestamp'
  | IN ROLE role_name [, ...]
  | IN GROUP role_name [, ...]
  | ROLE role_name [, ...]
  | ADMIN role_name [, ...]
  | USER role_name [, ...]
  | SYSID uid
URL: https://www.postgresql.org/docs/13/sql-createuser.html
```

让我们逐一讨论这些语法元素。我们可以看到的第一件事是，一个用户可以是超级用户，也可以是普通用户。如果某人被标记为SUPERUSER，就不再有普通用户必须面对的任何限制。SUPERUSER可以随心所欲地删除对象（例如，数据库）。

下一个重要的事情是，创建一个新的数据库需要实例级别的权限。

> 注意，当有人创建一个数据库时，这个用户将自动成为数据库的所有者。

规则是这样的：创建者总是自动成为一个对象的所有者（除非另有指定，如可以用CREATE DATABASE子句来做）。 这样做的好处是，对象所有者也可以再次放弃一个对象:

> CREATEROLE或NOCREATEROLE子句定义了是否允许某人创建新用户/角色。

下一个重要的事情是INHERIT或NOINHERIT子句。如果设置了INHERIT子句（这是默认值），一个用户可以继承其他用户的权限。使用继承的权限允许我们使用角色，这是一种抽象权限的好方法。例如，我们可以创建bookkeeper这个角色，并使许多其他角色继承bookkeeper的权限。这个想法是，我们只需要告诉PostgreSQL一次，一个bookkeeper角色被允许做什么，即使我们有很多人从事会计工作。

LOGIN或NOLOGIN子句定义了一个角色是否被允许登录到实例。

> 请注意，LOGIN子句并不足以实际连接到数据库。要做到这一点，需要更多的权限a

在这一点上，我们正试图使其进入实例，这是通往实例内所有数据库的大门。让我们回到我们的例子：bookkeeper可能被标记为NOLOGIN，因为我们希望人们用他们的真实姓名登录。你所有的会计人员（比如说，Joe和Jane）可能被标记为LOGIN子句，但可以继承bookkeeper角色的所有权限。像这样的结构可以很容易地确保所有的bookkeeper将拥有相同的权限，同时确保他们的个人活动在各自的身份下被操作和记录

如果我们计划用流式复制来运行PostgreSQL，我们可以用超级用户来做所有的交易日志流。然而，从安全的角度来看，这并不推荐。为了保证我们不必成为超级用户来流式传输xlog，PostgreSQL允许我们给普通用户以复制的权利，然后可以用它来做流式传输。通常的做法是创建一个特殊的用户，专门用于管理流复制。

正如我们在本章后面将看到的，PostgreSQL提供了一个叫做行级安全（RLS）的功能。其原理是，我们可以将行从用户的范围内排除。如果一个用户明确应该绕过RLS，把这个值设置为BYPASSRLS。默认值是NOBYPASSRLS。

有时，限制一个用户允许的连接数是有意义的。CONNECTION LIMIT正是允许我们这样做的。注意，总的来说，连接数不可能超过postgresql.conf文件中定义的数量（max_connections）。然而，我们总是可以把某些用户限制在一个较低的数值上。

默认情况下，PostgreSQL会在系统表中存储加密的密码，这是一个很好的默认行为。然而，假设你正在做一个培训课程，有10个学生参加，每个人都连接到你的盒子上。你可以百分之百的肯定，这些人中有一个会偶尔忘记自己的密码。由于你的设置不是安全关键，你可能会决定以明文方式存储密码，这样你就可以很容易地查找它并把它交给学生。如果你正在测试软件，这个功能可能也会派上用场。

通常情况下，我们已经知道某人会很快离开组织。VALID UNTIL子句允许我们在一个特定用户的账户过期时自动锁定他们

IN ROLE子句列出了一个或多个现有的角色，新的角色将被立即添加为新的成员。这有助于避免额外的手工步骤。IN ROLE的一个替代方法是IN GROUP。

ROLE子句将定义被自动添加为新角色成员的角色。

ADMIN子句与ROLE子句相同，但它增加了WITH ADMIN OPTION。 最后，我们可以使用SYSID子句为用户设置一个特定的ID（这类似于一些Unix管理员在操作系统层面上对用户名的处理）。偶尔也要对一个用户进行修改。下面一节解释了如何做到这一点。

8.创建和修改用户

在这个理论介绍之后，现在是时候实际创建用户，看看事情如何在实际例子中使用。

```
test=# CREATE ROLE bookkeeper NOLOGIN;
CREATE ROLE
test=# CREATE ROLE joe LOGIN;
CREATE ROLE
test=# GRANT bookkeeper TO joe;
GRANT ROLE
```

这里做的第一件事是，创建了一个叫做bookkeeper的角色。

请注意，我们不希望人们以bookkeeper的身份登录，所以该角色被标记为NOLOGIN。

> 你还应该注意，如果你使用CREATE ROLE子句，NOLOGIN是默认值。如果你喜欢使用CREATE USER子句，默认设置是LOGIN。

然后，joe角色被创建并标记为LOGIN。最后，bookkeeper角色被分配给joe角色，这样他们就可以做bookkeeper实际被允许做的所有事情。

一旦用户到位，我们就可以测试我们目前拥有的东西。

```
[hs@zenbook ~]$ psql test -U bookkeeper
psql: FATAL: role "bookkeeper" is not permitted to log in
```

正如预期的那样，bookkeeper角色不被允许登录系统。如果joe角色试图登录，会发生什么？请看下面的代码片断。

```
[hs@zenbook ~]$ psql test -U joe
...
test=>
```

这实际上将按预期工作。然而，请注意，命令提示符已经改变。这只是PostgreSQL向你显示你没有以超级用户身份登录的一种方式。 一旦创建了一个用户，可能有必要对其进行修改。我们可能想改变的一件事是密码。在PostgreSQL中，用户被允许改变他们自己的密码。下面是它的工作原理。

```
test=> ALTER ROLE joe PASSWORD 'abc';
ALTER ROLE
test=> SELECT current_user;
 current_user
--------------
 joe
(1 row)
```

> 请注意，ALTER ROLE会改变角色的属性。如果已经配置了数据定义语言（DDL）日志，PASSWORD实际上会使密码显示在日志文件中。这不是很理想。最好是用一个可视化工具来改变密码。在这种情况下，有一些协议支持，可以确保密码不会以明文形式在网上发送。直接使用ALTER ROLE来改变密码，根本不是一个好主意。

ALTER ROLE子句（或ALTER USER）将允许我们改变在创建用户时可以设置的大部分设置。然而，管理用户的内容甚至更多。在许多情况下，我们想给一个用户分配特殊的参数。ALTER USER子句为我们提供了这样的方法。

```
ALTER ROLE { role_specification | ALL }
 [ IN DATABASE database_name ]
 SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE { role_specification | ALL }
 [ IN DATABASE database_name ]
 SET configuration_parameter FROM CURRENT
ALTER ROLE { role_specification | ALL }
 [ IN DATABASE database_name ] RESET configuration_parameter
ALTER ROLE { role_specification | ALL }
 [ IN DATABASE database_name ] RESET ALL
```

语法相当简单，而且相当直截了当。为了说明为什么这真的很有用，我添加了一个真实世界的例子。让我们假设Joe碰巧住在毛里求斯岛。当他登录时，他希望在自己的时区，即使他的数据库服务器位于欧洲。 让我们在每个用户的基础上设置时区

```
test=> ALTER ROLE joe SET TimeZone = 'UTC-4';
ALTER ROLE
test=> SELECT now();
 now
-------------------------------
2020-10-09 20:36:48.571584+01
(1 row)
test=> \q
[hs@zenbook ~]$ psql test -U joe
...
test=> SELECT now();
 now
-------------------------------
2020-10-09 23:36:53.357845+04
(1 row)
```

ALTER ROLE子句将修改用户。当joe重新连接时，时区将已经为他设置好了。

> 时区不会被立即改变。你应该重新连接，或者使用SET ... TO DEFAULT子句。

这里重要的是，对于一些内存参数，如work_mem，也可以这样做，这在本书前面已经讲过了

9.定义数据库级安全性

在实例层面上配置完用户后，就可以深入挖掘，看看在数据库层面上可以做什么。出现的第一个主要问题是：我们明确允许joe登录数据库实例，但谁或什么允许joe实际连接到其中一个数据库？也许你不希望joe访问你系统中的所有数据库。限制对某些数据库的访问，正是我们在这个层面上可以实现的。

对于数据库，可以使用GRANT子句来设置以下权限。

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
 | ALL [ PRIVILEGES ] }
 ON DATABASE database_name [, ...]
 TO role_specification [, ...] [ WITH GRANT OPTION ]
```

在数据库层面，有两个主要的权限值得密切关注。

- CREATE：这允许某人在数据库中创建一个模式。 注意，CREATE子句不允许创建表；它是关于模式的。在PostgreSQL中，表驻留在模式中，所以你必须先进入模式级别，以便能够创建一个表。
- CONNECT（连接）。这允许某人连接到一个数据库

现在的问题是：没有人明确地给joe角色分配任何CONNECT权限，那么这些权限究竟从何而来？答案是这样的：有一个叫做public的东西，它类似于Unix的世界。如果世界被允许做某事，joe也是，他是public的一部分。

最主要的是，public不是一个角色，它可以被放弃和重新命名。我们可以简单地把它看作是系统中每个人的等同物。

因此，为了确保不是每个人都能在任何时候连接到任何数据库，CONNECT可能必须从一般人那里撤销。为此，我们可以以超级用户的身份连接并解决这个问题。

```
[hs@zenbook ~]$ psql test -U postgres
...
test=# REVOKE ALL ON DATABASE test FROM public;
REVOKE
test=# \q
[hs@zenbook ~]$ psql test -U joe
psql: FATAL: permission denied for database "test"
DETAIL: User does not have CONNECT privilege.
```

我们可以看到，joe角色不再被允许连接。在这一点上，只有超级用户可以访问测试

一般来说，在创建其他数据库之前，撤销postgres数据库的权限是个好主意。这个概念背后的想法是，这些权限不会再出现在所有那些新创建的数据库中。如果有人需要访问某个数据库，就必须明确授予这些权限。这些权利不再是自动存在的。

如果我们想允许joe角色连接到测试数据库，请以超级用户身份尝试以下一行。

```
[hs@zenbook ~]$ psql test -U postgres
...
test=# GRANT CONNECT ON DATABASE test TO bookkeeper;
GRANT
test=# \q
[hs@zenbook ~]$ psql test -U joe
...
test=>
```

这里有两个选择：

- 我们可以直接允许joe角色，这样只有joe角色能够连接。

- 或者，我们也可以授予记账员角色权限。 记住，joe角色将继承bookkeeper角色的所有权限，所以如果我们希望所有会计都能连接到数据库，给bookkeeper角色分配权限似乎是一个有吸引力的想法。

如果我们给bookkeeper角色授予权限是没有风险的，因为这个角色首先不允许登录到实例，所以它纯粹是作为一个权限的来源。

10.调整模式层面的权限

一旦我们完成了数据库层面的配置，看一下模式层面是有意义的。

在实际查看模式之前，让我们运行一个小测试。

```
test=> CREATE DATABASE test;
ERROR: permission denied to create database
test=> CREATE USER xy;
ERROR: permission denied to create role
test=> CREATE SCHEMA sales;
ERROR: permission denied for database test
```

我们可以看到，joe今天过得很糟糕，除了连接到数据库外，其他的都不允许。

然而，有一个小小的例外，它让很多人感到惊讶。

```
test=> CREATE TABLE t_broken (id int);
CREATE TABLE
test=> \d
 List of relations
Schema | Name | Type | Owner
--------+----------+--------+-------
 public | t_broken | table | joe
(1 rows)
```

默认情况下，public被允许与public模式一起工作，而public模式总是在周围。如果我们对确保我们的数据库安全很感兴趣的话，请确保这个问题得到解决。否则，普通用户将有可能用各种表来干扰你的public模式，整个设置可能会受到影响。你还应该记住，如果有人被允许创建一个对象，这个人也是它的所有者。所有权意味着所有的权限都自动提供给创建者，包括销毁该对象。

要把这些权限从public那里拿走，请以超级用户身份运行下面一行。

```
test=# REVOKE ALL ON SCHEMA public FROM public;
REVOKE
```

从现在起，没有人可以在没有正确权限的情况下把东西放到你的public模式中。下一个列表就是证明。

```
[hs@zenbook ~]$ psql test -U joe
...
test=> CREATE TABLE t_data (id int);
ERROR: no schema has been selected to create in
LINE 1: CREATE TABLE t_data (id int);
```

正如我们所看到的，该命令将失败。这里重要的是将显示的错误信息。PostgreSQL不知道应该把这些表放在哪里。默认情况下，它将尝试把表放到以下模式中的一个。

```
test=> SHOW search_path ;
 search_path
-----------------
 "$user", public
(1 row)
```

由于没有叫joe的模式，这不是一个选项，所以PostgreSQL将尝试public模式。由于没有权限，它将抱怨说它不知道把表放在哪里。

如果表格有明确的前缀，情况就会立即改变。

```
test=> CREATE TABLE public.t_data (id int);
ERROR: permission denied for schema public
LINE 1: CREATE TABLE public.t_data (id int);
```

在这种情况下，我们会得到你所期望的错误信息。PostgreSQL拒绝了对public模式的访问。

现在，下一个合乎逻辑的问题是：在模式层面可以设置哪些权限，以便给joe角色更多的权力？让我们来看看。

```
GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
 ON SCHEMA schema_name [, ...]
TO role_specification [, ...] [ WITH GRANT OPTION ]
```

CREATE意味着有人可以将对象放入模式中。USAGE意味着有人被允许进入该模式。注意，进入模式并不意味着模式内的东西可以被实际使用；这些权限还没有被定义。这只是意味着用户可以看到这个模式的系统目录。

为了允许joe角色访问它之前创建的表，以下一行将是必要的（以超级用户身份执行）。

```
test=# GRANT USAGE ON SCHEMA public TO bookkeeper;
GRANT
```

joe 角色现在能够按预期读取其表：

```
[hs@zenbook ~]$ psql test -U joe
test=> SELECT count(*) FROM t_broken;
 count
-------
 0
(1 row)
```

joe角色也能够添加和修改行，因为它正好是表的所有者。然而，尽管它已经可以做很多事情，但joe角色还不是万能的。请看下面的语句

```
test=> ALTER TABLE t_broken RENAME TO t_useful;
ERROR: permission denied for schema public
```

让我们仔细看一下实际的错误信息。我们可以看到，该消息抱怨的是模式上的权限，而不是表本身的权限（记住，joe角色拥有该表）。 要解决这个问题，必须在模式层面而不是表层面上解决。以超级用户身份运行下面一行。

```
test=# GRANT CREATE ON SCHEMA public TO bookkeeper;
GRANT
```

public模式上的CREATE权限被分配给bookkeeper。

joe角色现在可以把它的表的名字改成一个更有用的名字。

```
[hs@zenbook ~]$ psql test -U joe
test=> ALTER TABLE t_broken RENAME TO t_useful;
ALTER TABLE
```

请记住，如果使用DDL，这是必要的。在我作为PostgreSQL支持服务提供者的日常工作中，我已经看到了几个问题，这变成了一个问题。

11.使用表格

在处理了绑定地址、网络认证、用户、数据库和模式之后，我们终于来到了表的级别。下面的代码片断显示了可以为一个表设置哪些权限。

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE
 | REFERENCES | TRIGGER }
 [, ...] | ALL [ PRIVILEGES ] }
 ON { [ TABLE ] table_name [, ...]
 | ALL TABLES IN SCHEMA schema_name [, ...] }
 TO role_specification [, ...] [ WITH GRANT OPTION ]
```

让我来逐一解释这些权限:

- SELECT：这允许你读取一个表
- INSERT：这允许你向表中添加行（这也包括复制等，它不仅仅是关于INSERT子句）。注意，如果你被允许插入，你不会自动被允许读取。为了能够读取你所插入的数据，需要使用SELECT和INSERT子句。
- UPDATE：这是修改一个表的内容。
- DELETE:：这是用来从表中删除记录的。
- TRUNCATE：这允许你使用TRUNCATE子句。注意DELETE和TRUNCATE子句是两个独立的权限，因为TRUNCATE子句会锁定表，而DELETE子句不会这样做（即使没有WHERE条件也不会）。
- REFERENCES：这允许创建外键。在引用列和被引用列上都必须有这个权限，否则，键的创建就不会成功。
- TRIGGER：这允许创建触发器

> GRANT子句的好处是，我们可以同时对一个模式中的所有表设置权限。

这大大简化了调整权限的过程。也可以使用WITH GRANT OPTION子句。这个想法是为了确保正常用户可以将权限传递给其他人，这样做的好处是能够相当显著地减少管理员的工作负担。试想一下，一个为数百个用户提供访问的系统。管理所有这些人的工作可能开始变得非常繁重，因此管理员可以指定一些人自己管理数据的一个子集。

12.处理列级安全

在某些情况下，不是每个人都被允许看到所有的数据。试想一下一家银行。有些人可能会看到一个银行账户的全部信息，而其他人可能只被限制在数据的一个子集。在现实世界中，有人可能不被允许阅读余额栏，而其他人可能看不到人们的贷款利率。

另一个例子是，人们被允许看到其他人的资料，但不允许看到他们的照片或其他一些私人信息。现在的问题是：如何使用列级安全

为了证明这一点，我们将在现有的表中增加一个属于joe角色的列。

```
test=> ALTER TABLE t_useful ADD COLUMN name text;
ALTER TABLE
```

该表现在由两列组成。这个例子的目的是确保用户只能看到其中的一列。

```
test=> \d t_useful
 Table "public.t_useful"
 Column | Type | Modifiers
--------+---------+-----------
 id | integer |
 name | text |
```

作为一个超级用户，让我们创建一个用户，并给他们访问包含我们表的模式的权限。

```
test=# CREATE ROLE paul LOGIN;
CREATE ROLE
test=# GRANT CONNECT ON DATABASE test TO paul;
GRANT
test=# GRANT USAGE ON SCHEMA public TO paul;
GRANT
```

不要忘记给新来的人以CONNECT权限，因为在本章的早些时候，CONNECT被撤销了公共的。因此，明确的授予是绝对必要的，以确保我们可以进入到表。

可以给paul角色提供SELECT的权限。下面是它的工作原理。

```
test=# GRANT SELECT (id) ON t_useful TO paul;
GRANT
```

这已经足够了。已经可以以paul用户的身份连接到数据库并读取该列。

```
[hs@zenbook ~]$ psql test -U paul
...
test=> SELECT id FROM t_useful;
id
----
(0 rows)
```

如果我们使用列级权限，有一件重要的事情要记住。我们应该停止使用SELECT *，因为它将不再起作用。

```
test=> SELECT * FROM t_useful;
ERROR: permission denied for relation t_useful
```

*仍然意味着所有的列，但由于没有办法访问所有的列，事情就会立即出错。

13.配置默认权限

到目前为止，很多东西都已经配置好了。问题是，如果有新的表被添加到系统中会怎样？一个一个地处理这些表，并设置适当的权限，这可能是相当痛苦和冒险的。 如果这些事情能自动发生，那不是很好吗？这正是ALTER DEFAULT PRIVILEGES子句的作用。这个想法是给用户一个选项，使PostgreSQL在一个对象出现时自动设置所需的权限。现在不可能简单地忘记设置这些权限了。

下面的列表显示了语法规范的第一部分。

```
postgres=# \h ALTER DEFAULT PRIVILEGES
Command: ALTER DEFAULT PRIVILEGES
Description: define default access privileges
Syntax:
ALTER DEFAULT PRIVILEGES
 [ FOR { ROLE | USER } target_role [, ...] ]
 [ IN SCHEMA schema_name [, ...] ]
 abbreviated_grant_or_revoke
where abbreviated_grant_or_revoke is one of:
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
 [, ...] | ALL [ PRIVILEGES ] }
 ON TABLES
 TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

该语法的工作方式与GRANT子句相似，因此使用起来很简单、很直观。为了告诉我们它是如何工作的，我编译了一个简单的例子。这个想法是，如果joe角色创建了一个表，paul角色将自动能够使用它。下面的列表显示了如何分配默认权限。

```
test=# ALTER DEFAULT PRIVILEGES FOR ROLE joe  IN SCHEMA public GRANT ALL ON
TABLES TO paul;
ALTER DEFAULT PRIVILEGES
```

在我的例子中，公共模式中的所有表都被授予所有权限。

现在让我们以 joe 角色连接并创建一个表：

```
[hs@zenbook ~]$ psql test -U joe
...
test=> CREATE TABLE t_user (id serial, name text, passwd text);
CREATE TABLE
```

可以看到，现在可以成功创建表了。

以paul角色连接将证明该表已被分配到适当的权限集。

```
[hs@zenbook ~]$ psql test -U paul
...
test=> SELECT * FROM t_user;
 id | name | passwd
----+------+--------
(0 rows)
```

该表也可以很好地阅读。

14.深入研究 RLS

到目前为止，一个表总是作为一个整体来显示。当表包含100万行时，有可能从中检索出100万行。 如果有人有权利阅读一张表，那就是指整个表。在许多情况下，这是不足够的。通常情况下，不允许一个用户看到所有的行是可取的。

考虑下面这个现实世界的例子，一个会计正在为许多人做会计工作。包含税率的表格确实应该对每个人都是可见的，因为每个人都必须支付相同的税率。 然而，当涉及到实际交易时，会计师可能想确保每个人只被允许看到自己的交易。 不应允许A人看到B人的数据。除此之外，允许一个部门的老板看到他们那部分公司的所有数据也是有意义的。

RLS的设计正是为了做到这一点，它使你能够以一种快速和简单的方式建立多租户系统。配置这些权限的方法是制定策略。CREATE POLICY命令为我们提供了一个编写这些规则的方法。让我们来看看CREATE POLICY命令的一个例子。

```
postgres=# \h CREATE POLICY
Command: CREATE POLICY
Description: define a new row level security policy for a table
Syntax:
CREATE POLICY name ON table_name
 [ AS { PERMISSIVE | RESTRICTIVE } ]
 [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
 [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
 [ USING ( using_expression ) ]
 [ WITH CHECK ( check_expression ) ]
URL: https://www.postgresql.org/docs/13/sql-createpolicy.html
```

语法并不难理解。然而，当然要提供一个例子。为了描述如何编写策略，让我们首先以超级用户身份登录，并创建一个包含几个条目的表。

```
test=# CREATE TABLE t_person (gender text, name text);
CREATE TABLE
test=# INSERT INTO t_person
 VALUES ('male', 'joe'),
 ('male', 'paul'),
 ('female', 'sarah'),
 (NULL, 'R2- D2');
INSERT 0 4
```

一个表被创建，数据被成功添加。然后，访问权被授予joe角色，从下面的代码中可以看出。

```
test=# GRANT ALL ON t_person TO joe;
GRANT
```

到目前为止，一切都很正常，由于没有RLS的存在，joe角色实际上可以读取整个表。但是，让我们看看如果为该表启用ROW LEVEL SECURITY会发生什么。

```
test=# ALTER TABLE t_person ENABLE ROW LEVEL SECURITY;
ALTER TABLE
```

有一个拒绝，而且所有的默认策略都到位了，所以joe角色实际上会得到一个空表。

```
test=> SELECT * FROM t_person;
 gender | name
--------+------
(0 rows)
```

默认策略有很大的意义，因为用户被迫明确地设置权限。

现在，该表在RLS的控制下，可以以超级用户的身份编写策略。

```
test=# CREATE POLICY joe_pol_1 ON t_person  FOR SELECT TO joe USING (gender =
'male');
CREATE POLICY
```

以joe角色登录并选择所有的数据将只返回两行。

```
test=> SELECT * FROM t_person;
 gender | name
---------+------
 male | joe
 male | paul
(2 rows)
```

让我们以更详细的方式检查我们刚刚创建的策略。我们可以看到的第一件事是，该策略实际上有一个名字。它还与一个表相连，并允许进行某些操作（在本例中，是SELECT子句）。然后是USING子句。它定义了允许joe角色查看的内容。因此，USING子句是连接到每个查询的一个强制性过滤器，只选择我们的用户应该看到的行。

还有一个重要的附带说明，如果有不止一个策略，PostgreSQL将使用OR条件。简而言之，更多的策略将使你默认看到更多的数据。在PostgreSQL 9.6中，情况一直是这样的。然而，随着PostgreSQL 10.0的引入，用户可以选择条件是通过OR还是AND的方式连接的

```
PERMISSIVE | RESTRICTIVE
```

默认情况下，PostgreSQL是PERMISSIVE的，所以OR连接是在工作。如果我们决定使用RESTRICTIVE，那么这些子句将用AND来连接

现在，假设由于某种原因，已经决定让joe角色也可以看到机器人。要实现我们的目标，有两个选择。第一个选择是简单地使用ALTER POLICY子句来改变现有的策略。

```
postgres=# \h ALTER POLICY
Command: ALTER POLICY
Description: change the definition of a row level security policy
Syntax:
ALTER POLICY name ON table_name RENAME TO new_name
ALTER POLICY name ON table_name
 [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
 [ USING ( using_expression ) ]
 [ WITH CHECK ( check_expression ) ]
URL: https://www.postgresql.org/docs/13/sql-alterpolicy.html
```

这个列表显示了ALTER POLICY的语法。它与CREATE POLICY相当相似。

第二个选择是创建第二个策略，如下面的例子所示。

```
test=# CREATE POLICY joe_pol_2 ON t_person FOR SELECT TO joe USING (gender IS
NULL);
CREATE POLICY
```

创建政策的工作成功了。美中不足的是，除非使用RESTRICTIVE，否则这些策略只是使用前面所说的OR条件连接。因此，PostgreSQL现在将返回三条记录而不是两条。

```
test=> SELECT * FROM t_person;
 gender | name
--------+-------
 male | joe
 male | paul
 | R2-D2
(3 rows)
```

R2-D2的角色现在也包括在结果中，因为它与第二个策略相匹配。

为了描述PostgreSQL是如何运行查询的，我决定包括一个查询的执行计划。

```
test=> explain SELECT * FROM t_person;
 QUERY PLAN
-------------------------------------------------------------
 Seq Scan on t_person (cost=0.00..21.00 rows=9 width=64)
 Filter: ((gender IS NULL) OR (gender = 'male'::text))
(2 rows)
```

正如我们所看到的，USING子句都被作为强制过滤器添加到查询中。你可能已经注意到在语法定义中，有两种类型的子句。

- USING：该子句过滤已经存在的行。这与 SELECT 和 UPDATE 子句等相关。
- CHECK：该子句过滤即将创建的新行，因此它们与INSERT和UPDATE子句有关，以此类推。

如果我们尝试插入一行，会发生以下情况：

```
test=> INSERT INTO t_person VALUES ('male', 'kaarel');
ERROR: new row violates row-level security policy for table "t_person"
```

由于没有INSERT子句的策略，该语句自然会出错。下面是允许插入的策略。

```
test=# CREATE POLICY joe_pol_3 ON t_person FOR INSERT TO joe WITH CHECK (gender
IN ('male', 'female'));
CREATE POLICY
```

joe角色被允许在表中添加男性和女性，这在下面的列表中显示。

```
test=> INSERT INTO t_person VALUES ('female', 'maria');
INSERT 0 1
```

但是，也有一个问题。考虑以下示例：

```
test=> INSERT INTO t_person VALUES ('female', 'maria') RETURNING *;
ERROR: new row violates row-level security policy for table "t_person"
```

请记住，只有一个政策是选择男性。这里的麻烦是，该语句将返回一个女性，这是不允许的，因为joe角色是在只选择男性的策略下。

RETURNING * 子句仅适用于男性：

```
test=> INSERT INTO t_person VALUES ('male', 'max') RETURNING *;
 gender | name
---------+------
 male | max
(1 row)
INSERT 0 1
```

如果我们不想要这种行为，我们必须写一个真正包含适当的USING条款的策略。RLS是每个数据库侧安全系统中的一个重要组成部分。一旦我们定义了我们的RLS策略，退一步讲，看看你如何检查权限是有意义的。

15.检查权限

当所有的权限都被设定后，有时需要知道谁有哪些权限。对于管理员来说，找出谁被允许做什么是至关重要的。不幸的是，这个过程并不那么容易，需要一些知识。通常情况下，我是一个命令行使用的忠实粉丝。 然而，在权限系统的情况下，使用图形化的用户界面来做事情确实是有意义的。

在我告诉你如何读取PostgreSQL的权限之前，让我们把权限分配给joe角色，这样我们就可以在下一步检查它们。

```
test=# GRANT ALL ON t_person TO joe;
GRANT
```

关于权限的信息可以用psql中的z命令来检索。

```
test=# \x
Expanded display is on.
test=# \z t_person
Access privileges
-[ RECORD 1 ]-----+-------------------------------------------------------
----
Schema | public
Name | t_person
Type | table
Access privileges | postgres=arwdDxt/postgres
+
 | joe=arwdDxt/postgres
Column privileges |
Policies | joe_pol_1 (r):
+ | (u): (gender = 'male'::text)
+ | to: joe
+ | joe_pol_2 (r):
+ | (u): (gender IS NULL)
+ | to: joe
+ | joe_pol_3 (a):
+ | (c): (gender = ANY (ARRAY['male'::text, 'female'::text]))
+ | to: joe
```

这将返回所有这些政策，以及关于访问权限的信息。不幸的是，这些快捷方式很难读懂，而且我感觉它们并没有被管理员广泛理解。在这个例子中，joe角色从PostgreSQL中获得了arwdDxt。这些快捷键到底是什么意思？让我们来看看。

- a：这附加于 INSERT 子句
- r：读取 SELECT 子句
- w：这为 UPDATE 子句写入
- d：这为 DELETE 子句删除
- D：这用于TRUNCATE子句（引入时，t已经被采用）
- x：用于引用
- t：这用于触发器

如果你不知道这段代码，还有第二种方法可以使事情变得更容易阅读。考虑一下下面的函数调用

```
test=# SELECT * FROM aclexplode('{joe=arwdDxt/postgres}');
 grantor | grantee | privilege_type | is_grantable
---------+---------+----------------+--------------
 10 | 18481 | INSERT | f
 10 | 18481 | SELECT | f
 10 | 18481 | UPDATE | f
 10 | 18481 | DELETE | f
 10 | 18481 | TRUNCATE | f
 10 | 18481 | REFERENCES | f
 10 | 18481 | TRIGGER | f
(7 rows)
```

正如我们所看到的，权限集以一个简单的表格形式返回，这使工作变得非常简单。对于那些仍然认为在PostgreSQL中检查权限有些麻烦的人，我们 Cybertec 最近实施了一个额外的解决方案 (https://www.cybertec-postgresql.com) :pg_permission。其目的是为你提供简单的视图来更深入地检查安全系统。

要下载pg_permission，请访问我们的GitHub资源库页面：https://github.com/cybertec-postgresql/pg_permissions

你可以很容易地克隆版本库，如以下列表所示。

```
git clone https://github.com/cybertec-postgresql/pg_permission.git
Cloning into 'pg_permission'...
remote: Enumerating objects: 111, done.
remote: Total 111 (delta 0), reused 0 (delta 0), pack-reused 111
Receiving objects: 100% (111/111), 33.08 KiB | 292.00 KiB/s, done.
Resolving deltas: 100% (52/52), done.
```

一旦完成，进入该目录，运行make install即可。这两件事已经足够部署扩展了。

```
test=# CREATE EXTENSION pg_permissions;
CREATE EXTENSION
```

pg_permission将部署少量的视图，它们的结构都是相同的。

```
test=# \d all_permissions
```

```
 View "public.all_permissions"
 Column | Type | Collation | Nullable | Default
 -------------+-----------+-----------+----------+---------
 object_type | obj_type | | |
 role_name | name | | |
 schema_name | name | | |
 object_name | text | C | |
 column_name | name | | |
 permission | perm_type | | |
 granted | boolean | | |
Triggers:
 permissions_trigger INSTEAD OF UPDATE ON all_permissions
 FOR EACH ROW EXECUTE FUNCTION permissions_trigger_func()
```

all_permissions是一个简单的视图，为你提供所有权限的整体视图。你可以简单地按对象类型过滤，等等，来挖掘细节。同样有趣的是，这个视图上有一个触发器，所以如果你想改变权限，你可以简单地在所有这些视图上运行UPDATE，而pg_permissions将为你改变系统的权限。在下一节中，你将学习如何重新分配所有权。

16.重新分配对象和删除用户

在分配权限和限制访问之后，可能会发生用户从系统中被删除的情况。不出所料，这样做的命令是DROP ROLE和DROP USER命令。下面是DROP ROLE的语法。

```
test=# \h DROP ROLE
Command: DROP ROLE
Description: remove a database role
Syntax:
DROP ROLE [ IF EXISTS ] name [, ...]
URL: https://www.postgresql.org/docs/13/sql-droprole.html
```

一旦讨论了DROP ROLE的语法，我们可以试一试。下面的列表显示了它是如何工作的。

```
test=# DROP ROLE joe;
ERROR: role "joe" cannot be dropped because some objects depend on it
DETAIL: target of policy joe_pol_3 on table t_person
target of policy joe_pol_2 on table t_person
target of policy joe_pol_1 on table t_person
privileges for table t_person
owner of table t_user
owner of sequence t_user_id_seq
owner of default privileges on new relations belonging to role joe in schema
public
owner of table t_useful
```

PostgreSQL会发出错误信息，因为只有当一个用户的所有东西都被夺走了，才能将其删除。这是有道理的，原因如下：只是假设有人拥有一个表。PostgreSQL应该如何处理这个表呢？总得有人拥有它。

要把表从一个用户重新分配给下一个用户，可以考虑看一下REASSIGN子句。

```
test=# \h REASSIGN
Command: REASSIGN OWNED
Description: change the ownership of database objects owned by a database role
Syntax:
REASSIGN OWNED BY { old_role | CURRENT_USER | SESSION_USER } [, ...]
 TO { new_role | CURRENT_USER | SESSION_USER }
URL: https://www.postgresql.org/docs/13/sql-reassign-owned.html
```

语法也很简单，有助于简化切换过程。下面是一个例子。

```
test=# REASSIGN OWNED BY joe TO postgres;
REASSIGN OWNED
```

那么，让我们再次尝试删除 joe 角色：

```
test=# DROP ROLE joe;
ERROR: role "joe" cannot be dropped because some objects depend on it
DETAIL: target of policy joe_pol_3 on table t_person target of policy joe_pol_2
on table t_person
target of policy joe_pol_1 on table t_person privileges for table t_person
owner of default privileges on new relations belonging to role joe in schema
public
```

正如我们所看到的，问题的清单已经大大减少。我们现在能做的是一个接一个地解决所有这些问题，然后放弃这个角色。据我所知，没有任何捷径。使之更有效率的唯一方法是确保尽可能少的权限被分配给真实的人。试着把尽可能多的权限抽象成角色，而这些角色又可以被很多人使用。如果个别权限不分配给真实的人，一般来说，事情就会变得容易。

17.总结

数据库安全是一个广泛的领域，一个30页的章节很难涵盖PostgreSQL安全的所有方面。很多东西，比如SELinux和SECURITY DEFINER/INVOKER，都没有被触及。然而，在这一章中，我们学到了作为PostgreSQL开发者和数据库管理员所要面对的最常见的东西。我们还学习了如何避免基本的陷阱，以及如何使我们的系统更加安全。

在第9章，处理备份和恢复，我们将学习PostgreSQL流式复制和增量备份。本章还将介绍故障转移的情况。

18.问题

- 如何配置对PostgreSQL的网络访问?
- 什么是用户，什么是角色?
- 如何改变密码?
- 什么是RLS?

这些问题的答案可以在GitHub仓库中找到（https://github.com/PacktPublishing/Mastering-PostgreSQL-13-Fourth-Edition）。