



# PostgreSQL 开发指南

## 从入门到精通

董旭阳 著



2023-02-12

# 前言

本书主要面向 PostgreSQL 初级 DBA 和开发人员，内容包括：PostgreSQL 介绍和安装，用户和角色、数据库与模式的管理、表空间与表的维护、数据库的备份与恢复、SQL 查询、常用函数、DML 语句，以及一些高级功能，例如通用表表达式、窗口函数、数据库事务、索引优化、视图、存储过程/函数、触发器以及 Java、Python、PHP 应用程序编程接口等。

## 第 I 部分 基本概念

第 1 章 PostgreSQL 简介和安装。简单介绍 PostgreSQL 的功能特性和相关信息，以及 PostgreSQL 在不同操作系统上的安装方法。

第 2 章 角色与用户。PostgreSQL 使用角色的概念来管理数据库的访问权限。角色可以被看成是一个数据库用户或者是一个组。GRAN 和 REVOKE 语句分别用于对象的授权和撤销权限。

第 3 章 数据库与模式。一个 PostgreSQL 实例管理一个数据库集群，一个集群可以包含一个或多个数据库（Database），一个数据库包含一个或多个模式（Schema），模式就像是一个命名空间，其中包含了许多数据库对象，例如表、索引等。

第 4 章 管理数据表。介绍如何管理数据库中的表，包括创建表、修改表以及删除表等操作。

第 5 章 管理表空间。在 PostgreSQL 中，表空间（Tablespace）表示数据文件的存放目录。这些数据文件代表了数据库的对象，例如表或索引。

第 6 章 备份与恢复。服务器系统错误、硬件故障或者人为失误都可能导致数据的丢失或损坏。因此，备份和恢复对于数据库的高可用性至关重要。数据库管理员应该根据业务的需求制定合适的备份策略，并提前演练各种故障情况下的恢复过程，做到有备无患。

## 第 II 部分 SQL 语言

第 7 章 简单查询。介绍如何使用 SELECT 和 FROM 查询表中的数据，以及使用 DISTINCT 去除查询结果中的重复值。

第 8 章 查询条件。使用 WHERE 子句返回满足条件的数据。

第 9 章 排序显示。介绍如何使用 ORDER BY 进行查询结果的排序显示，包括单列排序、多列排序，升序和降序排序、空值排序等。

第 10 章 限定结果数量。对于常见的 Top-N 查询和分页功能，PostgreSQL 提供了标准的 FETCH 和 OFFSET 子句，同时还支持其他数据库中的 LIMIT 语法形式。

第 11 章 分组汇总。使用 GROUP BY 子句对数据进行分组，并且应用聚合函数针对每个组进行汇总分析。HAVING 子句可以对分组后的结果进行过滤。如果需要高级分组功能，PostgreSQL 提供了 GROUPING SETS、CUBE 以及 ROLLUP 选项。

第 12 章 多表连接。连接查询（JOIN）基于两个表中的关联字段将数据行拼接到一起，可以同时返回两个表中的相关数据。本篇介绍 PostgreSQL 支持的各种连接查询，包括内连接、左/右外连接、全外连接、交叉连接、自然连接以及自连接。

第 13 章 CASE 条件表达式。CASE 表达式为 SQL 语句增加类似于 IF-THEN-ELSE 的逻辑处理功能，根据不同的条件返回不同的结果。PostgreSQL 支持两种形式的条件表达式以及处理空值的 NULLIF 函数和 COALESCE 函数。

第 14 章 常用函数。介绍 PostgreSQL 常用的数学函数、字符函数、日期时间函数以及类型转换函数。

第 15 章 子查询。子查询（Subquery）是指嵌套在其他 SELECT、INSERT、UPDATE 以及 DELETE 语句中的查询。本篇介绍 PostgreSQL 中的子查询、关联子查询、横向子查询、IN、ALL、ANY 以及 EXISTS 操作符。

第 16 章 集合运算。介绍 PostgreSQL 中的集合操作符，UNION 用于将两个查询结果合并成一个结果集，返回出现在第一个查询或者出现在第二个查询中的数据；INTERSECT 用于返回两个查询结果中的共同部分，即同时出现在第一个查询结果和第二个查询结果中的数据；EXCEPT 用于返回出现在第一个查询结果中，但不在第二个查询结果中的数据。

第 17 章 通用表表达式。通用表表达式（Common Table Expression）是一个临时的查询结果或者临时表，可以在其他 SELECT、INSERT、UPDATE 以及 DELETE 语句中使用。使用 CTE 可以提高复杂查询的可读性，递归 CTE 可以遍历各种层次数据。CTE 和 DML 语句一起使用可以在一个语句中执行多个表的操作。

第 18 章 窗口函数。PostgreSQL 窗口函数（分析函数）基于和当前数据行相关的一组数据计算出一个结果。窗口函数使用 OVER 子句进行定义，包括 PARTITION BY、ORDER BY 以及 frame\_clause 三个选项。常见的窗口函数可以分为以下几类：聚合窗口函数、排名窗口函数以及取值窗口函数。

第 19 章 DML 语句。介绍如何在 PostgreSQL 中对表的数据进行修改操作，包括插入数据的 INSERT 语句、更新数据的 UPDATE 语句、删除数据的 DELETE 语句，以及合并数据的 INSERT ON CONFLICT 语句。

## 第 III 部分 高级功能

第 20 章 事务与并发控制。介绍 PostgreSQL 中的数据库事务概念和 ACID 属性，并发事务可能带来的问题以及 4 种隔离级别，演示了如何使用事务控制语句（TCL）对事务进行处理，包括 BEGIN、COMMIT、ROLLBACK 以及 SAVEPOINT 语句。

第 21 章 索引与优化。本篇介绍 PostgreSQL 中的索引概念，包括 B-树索引、哈希索引等类型，唯一索引、多列索引、函数索引、部分索引以及覆盖索引等方式。如何利用索引优化数据库的查询性能，以及创建索引、查看索引、维护索引等操作。

第 22 章 视图。介绍 PostgreSQL 中视图（View）的概念和作用，如何创建、修改、删除视图，以及可更新视图的使用与控制。

第 23 章 存储过程。介绍如何使用 PL/pgSQL 创建存储过程和函数，包括代码的块结构、变量声明与赋值、条件语句和循环控制结构、游标的使用、错误处理以及事务控制等。

第 24 章 触发器。介绍 PostgreSQL 触发器（trigger）的概念和作用，数据变更触发器（DML 触发器）和事件触发器（DDL 触发器）的创建、修改以及删除操作。

## 第 IV 部分 编程接口

第 25 章 PHP 访问 PostgreSQL。利用 PHP 数据对象（PDO）接口连接和操作 PostgreSQL 数据库，包括创建和删除表、执行数据的增删改查操作、事务的管理以及调用存储过程和函数。

第 26 章 Python 访问 PostgreSQL。利用 Python 驱动程序接口 psycopg 连接和操作 PostgreSQL 数据库，包括创建和删除表、执行数据的增删改查操作、事务的管理以及调用存储过程和函数。

第 27 章 Java 访问 PostgreSQL。在 Java 程序中利用 JDBC 接口连接和操作 PostgreSQL 数据库，包括创建和删除表、执行数据的增删改查操作、事务的管理以及调用存储过程。

## 附录

SQL 完整性约束。SQL 标准中的 6 种完整性约束，以及主流数据库中的实现。

# 第 1 章 PostgreSQL 简介与安装

## 1.1 PostgreSQL 简介

[PostgreSQL](#) 是世界上最先进的开源对象-关系型数据库管理系统(ORDBMS), 简称 Postgres。它最初基于加利福尼亚大学伯克利分校开发的 POSTGRES 项目, 至今已有三十多年的历史。

首先说开源, PostgreSQL 是一个免费并且开源的软件。它的源代码基于 PostgreSQL 许可发行, 这是一个类似于 BSD 或者 MIT 的自由开放源码许可协议。用户可以基于任何目的使用、修改和发布 PostgreSQL, 甚至直接包装一下拿出去卖钱都没问题, 唯一的要求就是保留它的版权声明。这一点相对于 MySQL 社区版的 GPL 协议友好许多。

再说它的先进, PostgreSQL 使用 C 语言进行开发, 最初是为了 UNIX 类平台而设计。不过, PostgreSQL 现在可以支持各种主流的平台, 例如 Linux、BSD、AIX、HP-UX、Mac OS X、Solaris 以及 Windows 等。

PostgreSQL 遵循事务的 ACID 原则, 高度兼容 SQL 标准。2022 年 10 月发布的 PostgreSQL 15 至少符合 SQL: 2016 核心一致性 179 项强制功能中的 170 项。目前还没有任何关系型数据库产品完全符合该标准。

以下是 PostgreSQL 所支持的主要功能和特性, 随着新版本的发布, 将会增加更多的功能:

- **数据类型**
  - 基本类型: 整型、数值、字符串、布尔值
  - 结构化类型: 日期/时间、数组、范围/多值范围、UUID
  - 文档类型: JSON/JSONB、XML、键值存储 (Hstore)
  - 几何类型: 点、线、圆、多边形
  - 定制化类型: 复合类型、自定义类型
- **数据完整性**
  - UNIQUE、NOT NULL
  - 主键
  - 外键
  - 排除约束
  - 显式锁、建议锁
- **并发、性能**
  - 索引: B-tree、复合索引、函数索引、部分索引
  - 高级索引: GiST、SP-Gist、KNN Gist、GIN、BRIN、覆盖索引、布隆过滤器索引
  - 先级的查询计划器/优化器、Index-Only 扫描、多列统计
  - 事务、嵌套事务 (通过保存点实现)
  - 多版本并发控制 (MVCC)
  - 并行查询以及 B-tree 索引的并行创建
  - 表分区
  - SQL 标准中定义的 4 种事务隔离级别, 包括序列化 (Serializable) 事务级别

- 表达式的即时（JIT）编译
- **可靠性、灾难恢复**
  - 预写式日志（WAL）
  - 复制：异步复制、同步复制、逻辑复制
  - 基于时间点的恢复（PITR）、活动备份
  - 表空间
- **安全**
  - 身份认证：GSSAPI、SSPI、LDAP、SCRAM-SHA-256、SSL 证书等
  - 强大的访问控制系统
  - 列级与行级安全性
  - 多因素认证（MFA）
- **可扩展性**
  - 存储函数和存储过程
  - 过程语言：PL/PgSQL、PL/Perl、PL/Python、PL/Tcl、PL/Java、PL/V8（JavaScript）、PL/R、PL/Lua、PL/Rust 等
  - SQL/JSON 路径表达式
  - 外部数据包装器（FDW）：通过标准 SQL 接口连接到其他数据库或数据流
  - 可定制化的表存储接口（存储引擎）
  - 大量提供额外功能的扩展：包括 PostGIS
- **国际化、全文检索**
  - 多种方式支持国际字符集，例如通过 ICU 排序规则
  - 不区分大小写和重音的排序规则
  - 全文检索

对于初学者，可以随着学习的深入慢慢了解这些功能的强大之处。如果想要了解每个 PostgreSQL 版本支持的新特性，可以查看官方的[特性矩阵](#)。该页面详细列出了不同版本支持的各种功能，方便进行比较，同时还可以通过链接查看每个功能的具体介绍。

PostgreSQL 另一个强大之处在于它的高度可扩展性。PostgreSQL 用户可以定义自己的数据类型、索引类型、过程语言等。市场上存在大量基于 PostgreSQL 的数据库产品，例如 Greenplum、EnterpriseDB、TimescaleDB、Citus 等。

如果需要支持，PostgreSQL 拥有一个非常活跃的社区，包括中文社区，通过社区通常总是可以解决你的各种问题。另外，许多公司可以提供商业支持。

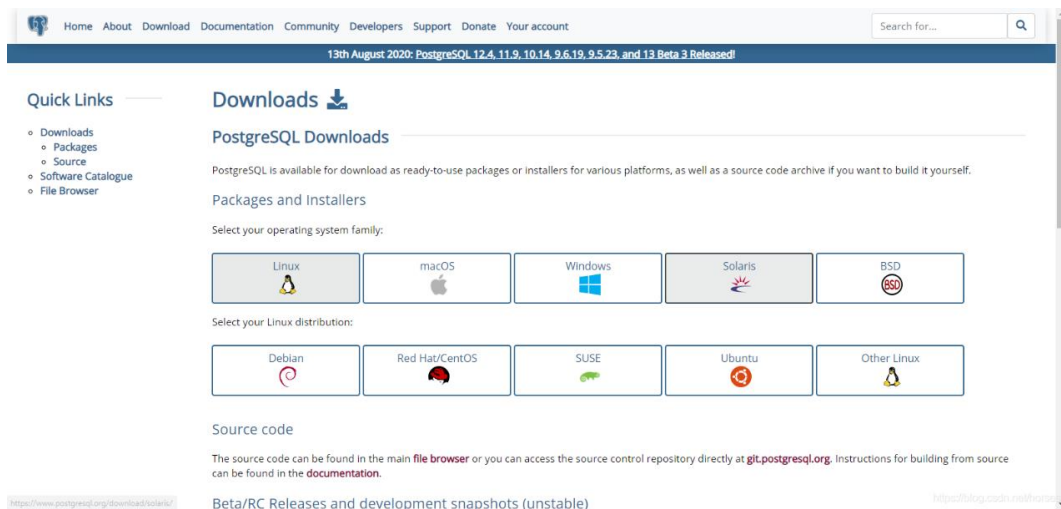
无论是可管理的数据量还是支持的用户并发数，PostgreSQL 都具有高度的可扩展性。在生产环境中已经存在能够管理 TB 级别数据量的 PostgreSQL 集群，以及能够管理 PB 级别数据量的专用系统。

## 1.2 安装 PostgreSQL

PostgreSQL 支持各种平台，包括 Linux、Windows、FreeBSD、OpenBSD、NetBSD、macOS、AIX、HP/UX 以及 Solaris。不同的平台支持不同的安装方法，通常可以分为以下几类：

- **二进制安装包**，通过官方[下载页面](#)，选择相应的操作系统版本进行下载安装。某些操作

系统（Windows、macOS）提供了图形化的安装工具。

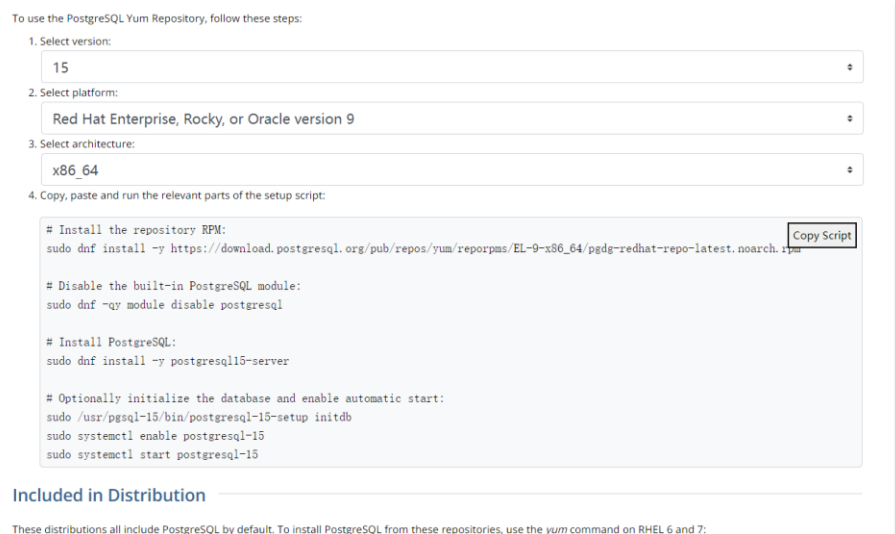


- **源码编译安装**，适合高级用户，通过 [ftp 文件浏览器](#) 或者 [git.postgresql.org](#) 下载源文件代码，参考 [官方文档](#) 执行源码安装。
- **第三方发行版**，同样通过官方 [下载页面](#)，可以找到许多第三方提供的打包软件，例如 LAPP 组合。

本文主要介绍如何通过 Yum 源安装 PostgreSQL 软件，这种方式适用于所有的 Red Hat Linux 家族以及衍生版：

- Red Hat Enterprise Linux
- CentOS
- Fedora
- Rocky Linux
- Oracle Linux

接下来我们基于 Rocky 9 X86\_64 安装 PostgreSQL 15。第一步就是下载并安装 Yum 源，在下载页面点击“[Linux -> Red Hat/Rocky/CentOS](#)”，然后在页面选择相应的 PostgreSQL 版本和操作系统。



按照第 4 步提供的命令安装 Yum 源（使用 root 用户执行操作）：

```
sudo dnf install -y
https://download.postgresql.org/pub/repos/yum/reporepms/EL-9-x86_64/pgdg-redh
at-repo-latest.noarch.rpm
```

安装之前先确认是否已经存在 PostgreSQL：

```
rpm -qa | grep postgres
```

如果存在，使用以下命令删除之前的安装包：

```
rpm -e postgresqlXX
```

然后禁用系统自带的安装源：

```
sudo dnf -qy module disable postgresql
```

安装 PostgreSQL 服务器：

```
sudo dnf install -y postgresql15-server
```

至此，完成了软件的安装。接下来是初始化一个数据库，并且启动 PostgreSQL 服务和开机后的自启动：

```
sudo /usr/pgsql-15/bin/postgresql-15-setup initdb
sudo systemctl enable postgresql-15
sudo systemctl start postgresql-15
```

执行完成后，可以通过操作系统的 ps 命令查看 PostgreSQL 后台进程：

```
[tony@localhost ~]$ ps -ef | grep postgres | grep -v 'grep'
postgres          10624          1    0  22:04  ?                00:00:00
/usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/15/data/
postgres  10672    10624  0  22:04  ?                00:00:00 postgres: logger
postgres  10702    10624  0  22:04  ?                00:00:00 postgres: checkpointer
postgres  10703    10624  0  22:04  ?                00:00:00 postgres: background writer
postgres  10707    10624  0  22:04  ?                00:00:00 postgres: walwriter
postgres  10708    10624  0  22:04  ?                00:00:00 postgres: autovacuum
launcher
postgres  10709    10624  0  22:04  ?                00:00:00 postgres: logical
replication launcher
```

通过以上输出，可以看出 PostgreSQL 安装在“/usr/pgsql-15”目录中，初始化数据库的数据目录为“/var/lib/pgsql/15/data”。同时，操作系统创建了一个新的用户“postgres”。

注意：PostgreSQL 15 不再需要统计收集器（stats collector）进程，而是使用动态共享内存来收集统计信息。

除此之外，还可以安装一些第三方的扩展包和管理工具：

```
sudo dnf install -y postgresql15-contrib
sudo dnf install -y pgadmin4
```

最后，使用命令行工具 psql 测试数据库的连接：

```
[tony@localhost ~]# su - postgres
[postgres@localhost ~]$ psql
psql (15.0)
输入 "help" 来获取帮助信息。
```



```
postgres=# select version();
              version
-----
PostgreSQL 15.0 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 11.2.1 20220127 (Red Hat 11.2.1-9), 64-bit
(1 行记录)
```

默认情况下，PostgreSQL 只接收本机的连接请求。如果需要通过远程客户端进行连接，可以执行以下两个步骤（使用 postgres 用户操作）：

1. 修改 postgresql.conf 文件中的监听地址，该文件位于数据目录(/var/lib/pgsql/15/data/)中。找到以下内容：

```
#listen_addresses = 'localhost' # what IP address(es) to listen on;

listen_addresses = '192.68.56.103'
```

2. 修改 pg\_hba.conf 文件中的客户端认证配置，该文件位于数据目录(/var/lib/pgsql/15/data/)中。增加以下内容，允许所有客户端 IP 访问：

```
host      all             all             0.0.0.0/0      md5

systemctl restart postgresql-15
```

对于 Windows 和 macOS 用户，推荐使用图形化工具进行安装。可以通过官方[下载页面](#)找到相应操作系统的链接。也可以直接使用[EnterpriseDB](#) 图形化安装工具进行安装。

PostgreSQL Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
15.0	<a href="#">postgresql.org</a>	<a href="#">postgresql.org</a>			Not supported
14.5	<a href="#">postgresql.org</a>	<a href="#">postgresql.org</a>			Not supported
13.8	<a href="#">postgresql.org</a>	<a href="#">postgresql.org</a>			Not supported
12.12	<a href="#">postgresql.org</a>	<a href="#">postgresql.org</a>			Not supported
11.17	<a href="#">postgresql.org</a>	<a href="#">postgresql.org</a>			Not supported
10.22					

使用图形化工具进行安装的过程比较简单，安装提示输入信息并点击下一步即可。

## 第 2 章 用户和角色

上一章我们安装了 PostgreSQL 数据库系统管理软件并且初始化了一个数据库集群(Database Cluster)。本章将会介绍 PostgreSQL 中的角色 (role) 和用户 (user) 的概念, 以及如何为数据库对象进行授权操作。

PostgreSQL 通过角色的概念来控制数据库的访问权限。角色又包含了两种概念, 具有登录权限的角色称为用户, 包含其他成员 (也是角色) 的角色称为组 (group)。因此, 一个角色可以是一个用户, 也可以是一个组, 或者两者都是。

角色可以拥有数据库对象 (例如表和函数), 并且可以将这些对象上的权限授予其他角色, 从而控制对象的访问。此外, 一个组中的成员可以拥有该组所拥有的权限。

### 2.1 创建角色

在 PostgreSQL 中, 使用 CREATE ROLE 语句创建角色:

```
CREATE ROLE name;
```

其中, *name* 指定了要创建的角色名称。

如果想要显示当前数据库集群中已有的角色, 可以查询系统目录 pg\_roles:

```
postgres=# SELECT rolname FROM pg_roles;
           rolname
-----
 postgres
 pg_database_owner
 pg_read_all_data
 pg_write_all_data
 pg_monitor
 pg_read_all_settings
 pg_read_all_stats
 pg_stat_scan_tables
 pg_read_server_files
 pg_write_server_files
 pg_execute_server_program
 pg_signal_backend
 pg_checkpoint
(13 行记录)
```

或者使用 psql 中的 \du 命令:

```
postgres=# \du
           List of roles
           |
Role name | Attributes |
Member of |
-----+-----
 postgres | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

其中的 `postgres` 是系统初始化数据库时创建的默认角色，它是一个超级用户。前一个命令中多出的角色都是系统提供的默认角色，用于提供针对一些特定的常用特权和信息的访问权限。每个默认角色包含的具体权限可以参考[官方文档](#)。

## 2.2 角色属性

角色可以拥有属性，属性确定了角色拥有的特权，并且在登录时与客户端认证系统进行交互。常见的角色属性包括：

- **登录特权**，只有具有 `LOGIN` 属性的角色才能连接数据库。具有 `LOGIN` 角色的用户可以被看作一个“数据库用户”。使用以下语句创建具有登录特权的角色：

```
CREATE ROLE name LOGIN;  
CREATE USER name;
```

`CREATE USER` 与 `CREATE ROLE` 都可以用于创建角色，只不过 `CREATE USER` 默认包含了 `LOGIN` 选项，而 `CREATE ROLE` 没有。

- **超级用户**，数据的超级用户可以避开所有的权限检查，只验证登录权限。因此，这是一个很危险的特权，使用时需要特别小心；最好在日常的操作中避免使用超级用户。使用以下命令创建一个新的超级用户：

```
CREATE ROLE name SUPERUSER;
```

只有超级用户才能创建其他的超级用户。

- **创建数据库**，只有明确授权的角色才能够创建数据库（超级用户除外，因为他们可以避开权限检查）。使用以下语句创建一个具有数据库创建特权的角色：

```
CREATE ROLE name CREATEDB;
```

- **创建角色**，只有明确授权的角色才能够创建其他角色（超级用户除外，因为他们可以避开权限检查）。使用以下命令创建一个具有角色创建特权的角色：

```
CREATE ROLE name CREATEROLE;
```

具有 `CREATEROLE` 特权的角色还可以修改或删除其他角色，以及为这些角色授予或者撤销成员角色。但是，针对超级用户的创建、修改、删除，以及它的成员变更，需要超级用户特权；`CREATEROLE` 特权无法针对超级用户执行这些操作。

- **启动复制**，只有明确授权的角色才能够启动流复制（超级用户除外，因为他们可以避开权限检查）。用于流复制的角色还需要拥有 `LOGIN` 特权。使用以下语句创建可以用于流复制的角色：

```
CREATE ROLE name REPLICATION LOGIN;
```

- **密码**，只有当用户连接数据库使用的客户端认证方法要求提供密码时，密码属性才有意义。`password` 和 `md5` 认证方法需要使用密码。数据库的密码与操作系统的密码相互独立。使用以下语句在创建角色时指定密码：

```
CREATE ROLE name PASSWORD 'string';
```

我们在创建角色时，可以根据需要指定某些属性。例如，以下命令创建一个具有登录特权的角色 `tony`，并且为它指定了密码以及密码过期时间：

```
CREATE ROLE tony WITH LOGIN PASSWORD 'Pass2022' VALID UNTIL '2025-01-01';  
-- CREATE USER tony WITH PASSWORD 'Pass2022' VALID UNTIL '2025-01-01';
```

使用该用户连接到 postgres 数据库:

```
[root@localhost ~]# psql -h 192.168.56.101 -p 5432 -U tony postgres
用户 tony 的口令:
psql (15.0)
输入 "help" 来获取帮助信息.

postgres=> \c
您现在已经连接到数据库 "postgres", 用户 "tony".
```

psql 命令行工具支持许多选项, -h 表示数据库服务器的地址, -p 表示服务的监听端口, -U 表示登录使用的用户名, 最后的 postgres 代表要连接的数据库。详细的命令行参数可以使用 psql --help 查看或者参考[官方文档](#)。

以下命令创建一个管理角色 admin, 它具有创建数据库和创建角色的特权:

```
postgres=# CREATE ROLE admin CREATEDB CREATEROLE;
CREATE ROLE
postgres=# \du

                                List of roles
   Role name |                               Attributes                               | Member of
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
admin       | Create role, Create DB, Cannot login                                | {}
postgres    | Superuser, Create role, Create DB, Replication, Bypass RLS         | {}
tony        | Password valid until 2025-01-01 00:00:00+08                        | {}
```

在实践中, 最好创建一个拥有 CREATEDB 和 CREATEROLE 特权, 但不具有超级用户特权的管理角色, 然后使用该角色执行日常的数据库和角色的管理。这种方式可以避免过度使用超级用户可能带来的风险。

一个角色被创建之后, 可以通过 ALTER ROLE 语句修改它的属性。例如, 以下命令可以撤销角色 admin 创建角色的特权:

```
postgres=# ALTER ROLE admin NOCREATEROLE;
ALTER ROLE
postgres=# \du

                                List of roles
   Role name |                               Attributes                               | Member of
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
admin       | Create DB, Cannot login                                              | {}
postgres    | Superuser, Create role, Create DB, Replication, Bypass RLS         | {}
tony        | Password valid until 2025-01-01 00:00:00+08                        | {}
```

## 2.3 对象授权

当我们使用新创建的用户 (tony) 连接数据库 (hrdb) 之后, 执行以下查询语句:

```
hrdb=> SELECT * FROM employees;
ERROR: permission denied for table employees
```

以上语句执行错误是因为用户没有相应对象上的访问权限。

PostgreSQL 使 GRANT 语句进行数据库对象的授权操作。以表为例，基本的授权语法如下：

```
GRANT privilege_list | ALL
    ON [ TABLE ] table_name
    TO role_name;
```

其中，`privilege_list` 权限列表可以是 `SELECT`、`INSERT`、`UPDATE`、`DELETE`、`TRUNCATE` 等，`ALL` 表示表上的所有权限。

例如，使用 `postgres` 用户连接 `hrdb` 数据库后执行以下语句：

```
hrdb=# GRANT SELECT, INSERT, UPDATE, DELETE
hrdb-#   ON employees, departments, jobs
hrdb-#   TO tony;
GRANT
```

该语句将 `employees`、`departments` 和 `jobs` 表上的增删改查权限授予了 `tony` 用户。此时 `tony` 用户就可以访问这些表中的数据：

```
hrdb=> SELECT first_name, last_name FROM employees;
first_name | last_name
-----+-----
Steven      | King
Neena       | Kochhar
Lex         | De Haan
...
```

对表进行授权的 `GRANT` 语句还支持一些其他选项：

```
GRANT privilege_list | ALL
    ON ALL TABLES IN SCHEMA schema_name
    TO role_name;
```

`ALL TABLES IN SCHEMA` 表示某个模式中的所有表，可以方便批量授权操作。例如：

```
hrdb=# GRANT SELECT
hrdb-#   ON ALL TABLES IN SCHEMA public
hrdb-#   TO tony;
GRANT
```

该语句将 `public` 模式中所有表的查询权限授予 `tony` 用户。

我们也可以在 `GRANT` 语句的最后指定一个 `WITH GRANT OPTION`，意味着被授权的角色可以将该权限授权其他角色。例如：

```
hrdb=# GRANT SELECT, INSERT, UPDATE, DELETE
hrdb-#   ON employees, departments, jobs
hrdb-#   TO tony WITH GRANT OPTION;
```

此时，`tony` 用户不但拥有这些表上的访问权限，还可以将这些权限授予其他角色。

除了授权表的访问权限之外，`GRANT` 语句还支持字段、视图、序列、数据库、函数、过程、模式等对象的授权操作。授权操作的语句基本都类似，具体可以参考[官方文档](#)。

## 2.4 撤销授权

与授权操作相反的就是撤销权限，PostgreSQL 使用 **REVOKE** 语句撤销数据库对象上的权限。同样以表为例，基本的撤销授权语句如下：

```
REVOKE privilege_list | ALL
    ON TABLE table_name
    FROM role_name;
```

其中的参数和 **GRANT** 语句一致。例如：

```
hrdb=# REVOKE SELECT, INSERT, UPDATE, DELETE
hrdb=#     ON employees, departments, jobs
hrdb=#     FROM tony;
REVOKE
```

该语句撤销了用户 **tony** 访问 **employees**、**departments** 以及 **jobs** 表的权限。

**REVOKE** 语句也支持对某个模式中的所有对象进行操作：

```
REVOKE privilege_list | ALL
    ON ALL TABLES IN SCHEMA schema_name
    FROM role_name;
```

例如以下语句撤销了用户 **tony** 在 **public** 模式中所有表上的查询权限：

```
hrdb=# REVOKE SELECT
hrdb=#     ON ALL TABLES IN SCHEMA public
hrdb=#     FROM tony;
REVOKE
```

与 **GRANT** 语句对应，**REVOKE** 语句还支持字段、视图、序列、数据库、函数、过程、模式等对象的撤销授权操作。撤销授权的语句基本都类似，具体可以参考[官方文档](#)。

## 2.5 角色成员

在现实的环境中，管理员通常需要管理大量的用户和对象权限。为了便于权限管理，减少复杂度，可以将用户进行分组，然后以组为单位进行权限的授予和撤销操作。

为此，PostgreSQL 引入了组（**group**）角色的概念。具体来说，就是创建一个代表组的角色，然后将该组的成员资格授予其他用户，让其成为该组的成员。

首先，使用以下创建一个组角色：

```
CREATE ROLE group_name;
```

按照习惯，组角色通常不具有 **LOGIN** 特权，也就是不能作为一个用户登录。

例如，我们可以先创建一个组 **managers**：

```
CREATE ROLE managers;
```

然后，使用与对象授权操作相同的 **GRANT** 和 **REVOKE** 语句为组添加和删除成员：

```
GRANT group_name TO user_role, ... ;
REVOKE group_name FROM user_role, ... ;
```

我们将用户 **tony** 添加为组 **managers** 的成员：

```
postgres=# GRANT managers TO tony;
GRANT ROLE
postgres=# \du

                                List of roles
Role name |                               Attributes                               | Member of
-----+-----+-----+-----+-----+-----+
admin     | Create DB, Cannot login                                             | {}
managers  | Cannot login                                                         | {}
postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS         | {}
tony      | Password valid until 2025-01-01 00:00:00+08                         | {managers}
{managers}
```

最后一行输出显示了成员角色（**tony**）所属的组（**managers**）。

也可以将一个组添加为其他组的成员，因为组角色和非组角色并没有什么本质区别。

```
postgres=# GRANT admin TO managers;
GRANT ROLE
postgres=# \du

                                List of roles
Role name |                               Attributes                               | Member of
-----+-----+-----+-----+-----+-----+
admin     | Create DB, Cannot login                                             | {}
managers  | Cannot login                                                         | {admin}
postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS         | {}
tony      | Password valid until 2025-01-01 00:00:00+08                         | {managers}
```

另外，PostgreSQL 不允许设置循环的成员关系，也就是两个角色互相为对方的成员。

```
postgres=# GRANT managers TO admin;
ERROR:  role "managers" is a member of role "admin"
```

最后，不能将特殊角色 **PUBLIC** 设置为任何组的成员。

组角色中的成员可以通过以下方式使用该组拥有的特权：

- 首先，组中的成员可以通过 **SET ROLE** 命令将自己的角色临时性“变成”该组角色。此时，当前数据库会话拥有该组角色的权限，而不是登录用户的权限；并且会话创建的任何数据库对象归组角色所有，而不是登录用户所有。
- 其次，对于具有 **INHERIT** 属性的角色，将会自动继承它所属的组的全部特权，包括这些组通过继承获得的特权。

考虑以下示例：

```
CREATE ROLE user1 LOGIN INHERIT;
CREATE ROLE net_admins NOINHERIT;
CREATE ROLE sys_admins NOINHERIT;
GRANT net_admins TO user1;
GRANT sys_admins TO net_admins;
```

使用角色 **user1** 登录之后，数据库会话将会拥有 **user1** 自身的特权和 **net\_admins** 所有的特权，

因为 user1“继承”了 net\_admins 的特权。但是，会话还不具有 sys\_admins 所有的特权，因为即使 user1 间接地成为了 sys\_admins 的成员，通过 net\_admins 获得的成员资格具有 NOINHERIT 属性，也就不会自动继承权限。

如果执行了以下语句：

```
SET ROLE net_admins;
```

会话将会拥有 net\_admins 所有的特权，但是不会拥有 user1 自身的特权，也不会继承 sys\_admins 所有的特权。

如果执行了以下语句：

```
SET ROLE sys_admins;
```

会话将会拥有 sys\_admins 所有的特权，但是不会拥有 user1 或者 net\_admins 所有的特权。

如果想要恢复初始状态的会话特权，可以执行以下任意语句：

```
SET ROLE user1;
SET ROLE NONE;
RESET ROLE;
```

在 SQL 标准中，用户和角色之间存在明确的差异，用户不会自动继承特权，而角色会继承特权。PostgreSQL 可以实现这种行为，只需要为角色设置 INHERIT 属性，而为用户设置 NOINHERIT 属性。但是，为了兼容 8.1 之前的版本实现，PostgreSQL 默认为所有的角色都设置了 INHERIT 属性，这样用户总是会继承它所在组的权限。

只有数据库对象上的普通权限可以被继承，角色的 LOGIN、SUPERUSER、CREATEDB 以及 CREATEROLE 属性可以被认为是一些特殊的权限，不会被继承。如果想要使用这些权限，必须使用 SET ROLE 命令设置为具有这些属性的角色。

基于上面的示例，我们可以为 net\_admins 角色指定 CREATEDB 和 CREATEROLE 属性。

```
ALTER ROLE net_admins CREATEDB, CREATEROLE;
```

然后再使用 user1 连接数据库，会话不会自动具有这些特权，而是需要执行以下命令：

```
SET ROLE net_admins;
```

## 2.6 删除角色

删除角色的语句如下：

```
DROP ROLE name;
```

如果删除的是组角色，该组中的成员关系会自动从组中删除，但是这些成员角色自身不会受到任何影响。

以下示例删除了角色 admin：

```
postgres=# drop role admin;
DROP ROLE
```



由于角色可以拥有数据库中的对象，也可以拥有访问其他对象的权限，删除角色通常不仅仅只是一个简单的 **DROP ROLE** 语句。在删除角色之前，需要删除它所拥有的对象，或者将这些对象重新赋予其他的角色；同时还需要撤销授予该角色的权限。详细信息可以参考[官方文档](#)。

## 第3章 数据库与模式

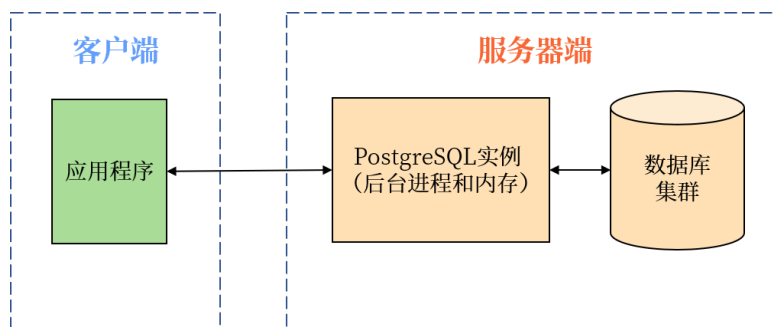
本章我们学习一下如何操作数据库和数据库中的模式。

### 3.1 基本概念

我们先来了解一些基本的概念。

**数据库管理系统(DBMS)**是用于管理数据库的软件系统。常见的关系型 DBMS 有 PostgreSQL、MySQL、Oracle、Microsoft SQL Server、SQLite 等。常见的 NoSQL 数据库有 Redis、MongoDB、Cassandra、Neo4j 等。PostgreSQL 荣获了数据库排名网站 [DB-Engines](#) 2017、2018 以及 2020 年度数据库管理系统称号。

PostgreSQL **数据库系统**由实例（Instance）和物理数据库集群（Database Cluster）组成。通常所说的数据库管理系统也就是指数据库系统。



**实例（Instance）**由 PostgreSQL 后台进程和相关的内存组成。启动服务器进程时创建一个实例，关闭服务器进程时实例随之关闭。启动 PostgreSQL 服务器进程之后，可以通过操作系统的 `ps` 命令查询相关的后台进程：

```
[root@localhost ~]# ps -ef | grep postgres | grep -v 'grep'
postgres          923          1    0  15:39   ?                00:00:00
/usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/15/data/
postgres          948        923    0  15:39   ?                00:00:00 postgres: logger
postgres          949        923    0  15:39   ?                00:00:00 postgres: checkpointer
postgres          950        923    0  15:39   ?                00:00:00 postgres: background writer
postgres          952        923    0  15:39   ?                00:00:00 postgres: walwriter
postgres          953        923    0  15:39   ?                00:00:00 postgres: autovacuum
launcher
postgres          954        923    0  15:39   ?                00:00:00 postgres: logical
replication launcher
```

**数据库集群**，每个 PostgreSQL 实例管理的都是一个数据库集群，它可以包含多个数据库。需要注意，这里的集群不是多台服务器组成的集群。



也可以使用 SQL 语句查看已有的数据库：

```
postgres=# SELECT datname FROM pg_database;
 datname
-----
 postgres
 template1
 template0
(3 行记录)
```

系统默认为我们创建了 3 个数据，其中 `template0` 和 `template1` 是模板数据库，创建新的数据库时默认基于 `template1` 进行复制；`postgres` 数据库是为 `postgres` 用户创建的默认数据库。

使用 SQL 语 `CREATE DATABASE` 创建一个数据库：

```
CREATE DATABASE name;
```

其中的 `name` 指定了数据库的名称。例如，以下语句创建了一个名为 `testdb` 的数据库：

```
postgres=# CREATE DATABASE testdb;
CREATE DATABASE
postgres=# \l

                          List of databases
   Name   | Owner   | Encoding | Collate  |  Ctype  | Access
-----+-----+-----+-----+-----+-----
 postgres | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres
+
          |          |          |          |          |
 postgres=CtC/postgres
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres
+
          |          |          |          |          |
 postgres=CtC/postgres
 testdb    | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
(4 rows)
```

创建数据库时还可以指定许多选项，例如字符集编码、拥有者、默认表空间、最大连接数等等。具体参考官方文档中完整的 [CREATE DATABASE](#) 语句。

### 3.3 修改数据库

创建之后，可以通过 `ALTER DATABASE` 语句修改数据库的属性和配置：

```
ALTER DATABASE name action;
```

其中，`action` 指定了要执行的修改操作，例如修改数据库的名称、所有者、默认表空间、数据库会话变量的默认值等等。

以下语句修改 `testdb` 的名称：

```
postgres=# ALTER DATABASE testdb RENAME TO newdb;
```

```
ALTER DATABASE
postgres=# \l
```

List of databases						
Name	Owner	Encoding	Collate	Ctype	Access privileges	
newdb	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		
template0	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres	
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres	
template1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres	

```
(4 rows)
```

然后将数据库 **newdb** 的拥有者修改为 **tony**:

```
postgres=# ALTER DATABASE newdb OWNER TO tony;
ALTER DATABASE
postgres=# \l
```

List of databases						
Name	Owner	Encoding	Collate	Ctype	Access privileges	
newdb	tony	UTF8	en_US.UTF-8	en_US.UTF-8		
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		
template0	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres	
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres	
template1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres	

```
(4 rows)
```

除了修改常见的属性之外，**ALTER DATABASE** 语句还可以用于修改运行时配置变量的会话默认值。

当用户连接数据库时，PostgreSQL 使用配置文件 **postgresql.conf** 或者启动命令 **postgres** 中设置的变量值作为默认值。使用 **ALTER DATABASE** 语句可以设置指定数据库的这些配置：

```
ALTER DATABASE name SET configuration_parameter { TO | = } { value | DEFAULT }
```

对于随后连接的会话，PostgreSQL 将会使用该命令设置的值覆盖 **postgresql.conf** 文件或者命令行参数的值。注意，只有超级用户或者数据库的拥有者才能修改数据库的默认会话变量。

例如，以下语句将会默认禁用数据库 **newdb** 中的索引扫描：

```
ALTER DATABASE newdb SET enable_indexscan TO off;
```

使用以下命令可以还原默认配置：

```
ALTER DATABASE newdb RESET enable_indexscan;
```

详细的修改选项可以参考官方文档中的 [ALTER DATABASE](#) 语句。

## 3.4 删除数据库

如果不需要，我们可以使用 **DROP DATABASE** 语句删除一个数据库：

```
DROP DATABASE [ IF EXISTS] name;
```

如果使用了 **IF EXISTS**，删除一个不存在的数据库时不会产生错误信息。

删除数据库会同时删除该数据库中所有的对象，以及文件系统中的数据目录。只有数据库的拥有者才能够删除数据库。另外，如果数据库上存在用户连接，无法执行删除操作，可以连接到其他数据库执行删除命令。

以下语句可以用于删除 **newdb** 数据库：

```
postgres=# DROP DATABASE newdb;
DROP DATABASE
postgres=# \l

                                List of databases
   Name   | Owner   | Encoding | Collate  |  Ctype  | Access
privileges
-----+-----+-----+-----+-----+-----
postgres | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres
+
          |          |          |          |          |
postgres=CtC/postgres
template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres
+
          |          |          |          |          |
postgres=CtC/postgres
(3 rows)
```

**DROP DATABASE** 命令的删除操作无法恢复，使用时千万小心！

## 3.5 管理模式

创建了数据库之后，还需要创建模式（Schema）才能够存储数据库对象。通常在创建一个新的数据库时，默认会创建一个模式 **public**。

首先，我们创建一个新的数据库 **testdb**：

```
postgres=# CREATE DATABASE testdb;
CREATE DATABASE
postgres=# \c testdb;
You are now connected to database "testdb" AS user "postgres".
```

```
testdb=# \dn
List of schemas
Name | Owner
-----+-----
public | pg_database_owner
(1 row)
```

其中，\c 用于连接到一个数据库；\dn 用于查看当前数据库中的模式。也可以使用 SQL 语句查询模式：

```
testdb=# SELECT nspname FROM pg_namespace;
 nspname
-----
pg_toast
pg_temp_1
pg_toast_temp_1
pg_catalog
public
information_schema
(6 rows)
```

查询结果还显示了系统提供的其他模式。

与数据库的管理类似，PostgreSQL 也提供了管理模式的语句：

- **CREATE SCHEMA**，创建一个新的模式。
- **ALTER SCHEMA**，修改模式的属性。
- **DROP SCHEMA**，删除一个模式。

来看一个简单的例子，首先在 testdb 中创建一个新的模式：

```
testdb=# CREATE SCHEMA hr;
CREATE SCHEMA
testdb=# \dn
List of schemas
Name | Owner
-----+-----
hr | postgres
public | pg_database_owner
(2 rows)
```

创建模式时还可以指定它的拥有者：

```
testdb=# CREATE SCHEMA app AUTHORIZATION tony;
CREATE SCHEMA
testdb=# \dn
List of schemas
Name | Owner
-----+-----
app | tony
hr | postgres
public | pg_database_owner
(3 rows)
```

以 pg\_开头的名称是系统保留的模式名称，用户无法创建这样的模式。

创建了模式之后，我们就可以在模式中创建各种数据库对象，例如表、数据类型、函数以及

运算符等等。这些内容在后面的篇章中再进行介绍。

如果需要修改已有模式的属性，可以使用 **ALTER SCHEMA** 语句：

```
ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

以上语句分别用于修改模式的名称和拥有者。以下语句将模式 **hr** 的拥有者改为 **tony**：

```
testdb=# ALTER SCHEMA hr OWNER TO tony;
ALTER SCHEMA
testdb=# \dn
      List of schemas
      Name | Owner
      -----+-----
      app  | tony
      hr   | tony
      public | pg_database_owner
(3 rows)
```

如果模式中没有任何对象，使用以下语句即可删除该模式：

```
DROP SCHEMA name;
```

以下示例将会删除模式 **hr**：

```
testdb=# DROP SCHEMA hr;
DROP SCHEMA
testdb=# \dn
      List of schemas
      Name | Owner
      -----+-----
      app  | tony
      public | pg_database_owner
(2 rows)
```

如果模式中还存在其他对象，以上语句无法执行；需要先删除该模式中所有的对象，或者使用以下语句级联删除这些对象：

```
DROP SCHEMA name CASCADE;
```

级联删除可能会删除一些我们意料之外的对象，使用时需要小心。

数据库中的大多数对象都位于某个模式之中，这样设计的好处在于：

- 允许多个用户使用同一个数据库而不会互相干扰，他们可以使用不同的模式来维护自己的数据。
- 将数据库对象进行逻辑上的分组，便于管理。
- 第三方应用可以使用单独的模式，不会与系统中的其他对象产生命名冲突。

在我们常用的数据库对象中，最主要的就是数据表（**table**）。



## 第 4 章 管理数据表

我们已经了解到，表包含在模式中，模式包含在数据库中。接下来介绍如何管理数据库中的表，包括创建表、修改表以及删除表等操作。

### 4.1 创建表

在 PostgreSQL 中，使用 CREATE TABLE 语句创建一个新表：

```
CREATE TABLE table_name
(
    column_name data_type column_constraint,
    column_name data_type,
    ...,
    table_constraint
);
```

该语句包含以下内容：

- 首先，table\_name 指定了新表的名称。
- 括号内是字段的定义，column\_name 是字段的名称，data\_type 是它的类型，column\_constraint 是可选的字段约束；多个字段使用逗号进行分隔。
- 最后，table\_constraint 是可选的表级约束。

来看一个具体的例子：

```
CREATE TABLE departments
( department_id    INTEGER NOT NULL PRIMARY KEY
, department_name  CHARACTER VARYING(30) NOT NULL
) ;
```

以上语句创建了一个新的部门表(departments)。它包含两个字段，部门编号(department\_id)是一个整数类型(INTEGER)，字段的值不可以为空(NOT NULL)，同时它还是这个表的主键(PRIMARY KEY)；部门名称(department\_name)是一个可变长度的字符串，也不允许为空。

PostgreSQL 提供了丰富的内置数据类型，同时还允许用户自定义数据类型。最常见的基本数据类型包括：

- **字符类型**，包括定长字符串 CHAR(n)，变长字符串 VARCHAR(n)，以及支持更大长度的字符串 TEXT。
- **数字类型**，包括整数类型 SMALLINT、INTEGER、BIGINT，精确数字 NUMERIC (p, s)，浮点数 REAL、DOUBLE PRECISION。
- **时间类型**，包括日期 DATE、时间 TIME、时间戳 TIMESTAMP。

更多详细的数据类型介绍可以查看[官方文档](#)。

PostgreSQL 支持 SQL 标准中的所有字段约束和表约束。其中，字段约束包括：

- **NOT NULL**，非空约束，该字段的值不能为空(NULL)。
- **UNIQUE**，唯一约束，该字段每一行的值不能重复。不过，PostgreSQL 允许该字段存在

多个 NULL 值，并且将它们看作不同的值。需要注意的是 SQL 标准只允许 UNIQUE 字段中存在一个 NULL 值。

- **PRIMARY KEY**，主键约束，包含了 NOT NULL 约束和 UNIQUE 约束。如果主键只包含一个字段，可以通过列级约束进行定义（参考上面的示例）；但是如果主键包含多个字段（复合主键）或者需要为主键指定一个自定义的名称，需要使用表级约束进行定义（参见下文示例）。
- **REFERENCES**，外键约束，字段中的值必需已经在另一个表中存在。外键用于定义两个表之间的参照完整性（referential integrity），例如，员工的部门编号字段必须是一个已经存在的部门。
- **CHECK**，检查约束，插入或更新数据时检查数据是否满足某个条件。例如，产品的价格必需大于零。
- **DEFAULT**，默认值，插入数据时，如果没有为这种列指定值，系统将会使用默认值代替。

表级约束和字段约束类似，只不过它是基于整个表定义的约束，还能够为约束指定自定义的名称。PostgreSQL 支持的表级约束包括：

- **UNIQUE(column1,...)**，唯一约束，括号中的字段值或字段值的组合必须唯一。
- **PRIMARY KEY(column1,...)**，主键约束，定义主键或者复合主键。
- **REFERENCES**，定义外键约束。
- **CHECK**，定义检查约束。

以下示例创建了员工表（employees）：

```
CREATE TABLE employees
(
    employee_id    INTEGER NOT NULL
    , first_name    CHARACTER VARYING(20)
    , last_name     CHARACTER VARYING(25) NOT NULL
    , email         CHARACTER VARYING(25) NOT NULL
    , phone_number  CHARACTER VARYING(20)
    , hire_date     DATE NOT NULL
    , salary        NUMERIC(8,2)
    , commission_pct NUMERIC(2,2)
    , manager_id    INTEGER
    , department_id INTEGER
    , CONSTRAINT    emp_emp_id_pk
                    PRIMARY KEY (employee_id)
    , CONSTRAINT    emp_salary_min
                    CHECK (salary > 0)
    , CONSTRAINT    emp_email_uk
                    UNIQUE (email)
    , CONSTRAINT    emp_dept_fk
                    FOREIGN KEY (department_id)
                    REFERENCES departments(department_id)
    , CONSTRAINT    emp_manager_fk
                    FOREIGN KEY (manager_id)
                    REFERENCES employees(employee_id)
) ;
```

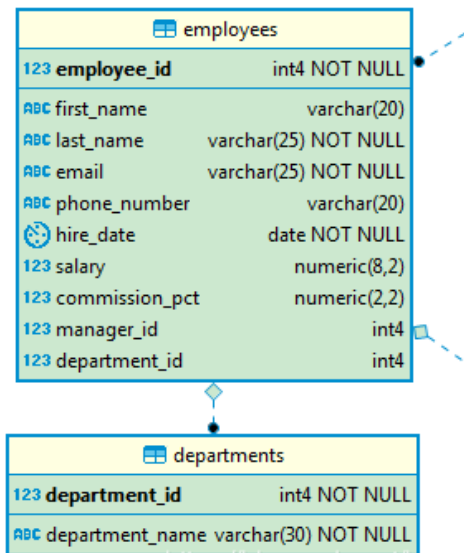
员工表包含以下字段和约束：

- **employee\_id**，员工编号，整数类型，主键（通过表级约束为主键指定了名称

emp\_emp\_id\_pk) ;

- first\_name, 名字, 字符串;
- last\_name, 姓氏, 字符串, 不能为空;
- email, 电子邮箱, 字符串, 不能为空, 必须唯一 (emp\_email\_uk) ;
- phone\_number, 电话号码, 字符串;
- hire\_date, 雇佣日期, 日期类型, 不能为空;
- salary, 薪水, 数字类型, 必须大于零 (emp\_salary\_min) ;
- commission\_pct, 佣金百分比, 数字类型;
- manager\_id, 经理编号, 外键 (通过外键 emp\_manager\_fk 引用员工表的员工编号) ;
- department\_id, 部门编号, 外键 (通过外键 emp\_dept\_fk 引用部门表 departments 的编号 department\_id)

下图是这两个表的实体关系图:



除了自己定义表的结构之外, PostgreSQL 还提供了另一个创建表的方法, 就是通过一个查询的结果创建新表:

```
CREATE TABLE table_name
AS query;
```

或者:

```
SELECT ...
INTO new_table
FROM ...;
```

例如, 我们可以基于 **employees** 复制出两个新的表:

```
CREATE TABLE emp1
AS
SELECT *
FROM employees;

SELECT *
```

```
INTO emp2
FROM employees;
```

这种方法除了复制表结构之外，还可以复制数据。具体说明请参考官方文档中的 [CREATE TABLE AS](#) 语句和 [SELECT INTO](#) 语句。

#### 4.1.1 模式搜索路径

在 PostgreSQL 中，表属于某个模式（schema）。当我们创建表时，更完整的语法应该是：

```
CREATE TABLE schema_name.table_name
...
```

访问表的时候也是一样。但是我们在前面创建示例表的时候，并没有加上模式名称的限制。这里涉及到一个模式的搜索路径概念。

我们先看一下当前的搜索路径：

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

搜索路径是一个逗号分隔的模式名称。当我们使用表的时候，PostgreSQL 会依次在这些模式中进行查找，返回第一个匹配的表名；当我们创建一个新表时，如果没有指定模式名称，PostgreSQL 会在第一个模式中进行创建。

第一个模式默认为当前用户名，如果不存在该模式，使用后面的公共模式（public）。

```
testdb=> SELECT user;
      user
-----
    tony
(1 row)

testdb=> \dn
      List of schemas
      Name | Owner
-----+-----
    app   | tony
    public | pg_database_owner
(2 rows)
```

当前用户名为 tony，但是不存在名为 tony 的模式，因此我们创建的表会位 public 模式中。

```
testdb=> \d
      List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | departments    | table | tony
 public | emp1            | table | tony
 public | emp2            | table | tony
 public | employees       | table | tony
(4 rows)
```

我们可以通过 SET 命令修改默认的搜索路径：

```
SET search_path TO app,public;
```

此时，如果我们再创建新表而不指定模式名称时，默认会在模式 `app` 中创建。

除了表之外，其他的模式对象，例如索引、函数、类型等等，也遵循相同的原则。

如果想要了解更多关于模式的信息，可以参考[官方文档](#)。

## 4.2 修改表

当我们创建好一个表之后，可能会由于业务变更或者其他原因需要修改它的结构。PostgreSQL 使 `ALTER TABLE` 语句修改表的定义：

```
ALTER TABLE name action;
```

其中的 `action` 表示要执行的操作。常见的修改操作包括：

- 添加字段
- 删除字段
- 添加约束
- 删除约束
- 修改字段默认值
- 修改字段数据类型
- 重命名字段
- 重命名表

### 4.2.1 添加字段

为表添加一个字段的命令如下：

```
ALTER TABLE table_name  
ADD COLUMN column_name data_type column_constraint;
```

添加字段与创建表时的字段选项相同，包含字段名称、字段类型以及可选的约束。

假设我们已经创建了一个产品表 `products`：

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);
```

通过以下语句为产品表增加一个新的字段 `description`：

```
ALTER TABLE products ADD COLUMN description text;
```

对于表中已有的数据，新增加的列将会使用默认值进行填充；如果没有指定 `DEFAULT` 值，使用空值填充。

添加字段时还可以定义约束。不过需要注意的是，如果表中已经存在数据，新增字段的默认值有可能会违反指定的约束。例如：

```
-- 插入一行数据
testdb=> INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese',
9.99);
INSERT 0 1

testdb=> ALTER TABLE products ADD COLUMN notes text not null;
ERROR: column "notes" contains null values
```

以上语句出错的原因在于新增的字段 **notes** 存在非空约束，但是对于已有的数据该字段的值为空。解决的方法有两个：添加约束的同时指定一个默认值；添加字段时不指定约束，将所有数据的字段值手动填充（**UPDATE**）之后，再添加约束。

以下语句为新增的字段指定了一个默认值：

```
testdb=> ALTER TABLE products ADD COLUMN notes text DEFAULT 'new product' not
null;
ALTER TABLE

testdb=> SELECT * FROM products;
 product_no | name | price | description | notes
-----+-----+-----+-----+-----
          1 | Cheese | 9.99 | | new product
(1 row)
```

## 4.2.2 删除字段

删除一个字段的语句如下：

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

我们将产品表中的 **notes** 字段删除：

```
testdb=> \d products;
               Table "public.products"
   Column   | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
 product_no | integer |           | not null |
 name       | text    |           |          |
 price      | numeric |           |          |
 description | text    |           |          |
 notes      | text    |           | not null | 'new product'::text
Indexes:
    "products_pkey" PRIMARY KEY, btree (product_no)
Check constraints:
    "products_description_check" CHECK (description <> ''::text)

testdb=> ALTER TABLE products DROP COLUMN notes;
ALTER TABLE
testdb=> \d products;
               Table "public.products"
   Column   | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
 product_no | integer |           | not null |
 name       | text    |           |          |
 price      | numeric |           |          |
```

```

description | text      |          |          |
Indexes:
    "products_pkey" PRIMARY KEY, btree (product_no)

```

删除字段后，相应的数据也会自动删除。同时，该字段上的索引或约束（`products_description_check`）也会同时被删除。但是，如果该字段被其他对象（例如外键引用、视图、存储过程等）引用，无法直接删除。

```

testdb=> ALTER TABLE departments DROP COLUMN department_id;
ERROR:  cannot drop column department_id of table departments because other
objects depend on it
DETAIL:  constraint emp_dept_fk on table employees depends on column
department_id of table departments
HINT:  Use DROP ... CASCADE to drop the dependent objects too.

```

由于 `departments` 表的 `department_id` 是 `employees` 表的外键引用列，无法直接删除该字段。通过提示可以看出，在 `DROP` 的最后加上 `CASCADE` 选项即可级联删除依赖的对象。

```

testdb=> ALTER TABLE departments DROP COLUMN department_id CASCADE;
NOTICE: drop cascades to constraint emp_dept_fk on table employees
ALTER TABLE

```

字段 `department_id` 被删除，同时 `employees` 表中的外键也被级联删除。

### 4.2.3 添加约束

添加约束时通常使用表级约束语法：

```
ALTER TABLE table_name ADD table_constraint;
```

其中，`table_constraint` 可以参考前文。

以下是为产品表 `products` 增加约束的一些示例：

```

testdb=> ALTER TABLE products
        ADD CONSTRAINT products_price_min CHECK (price > 0);
ALTER TABLE
testdb=> ALTER TABLE products ADD CONSTRAINT products_name_uk UNIQUE (name);
ALTER TABLE
testdb=> \d products;
               Table "public.products"
   Column    | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
product_no  | integer |           | not null |
name        | text   |           |          |
price       | numeric |           |          |
description | text   |           |          |
Indexes:
    "products_pkey" PRIMARY KEY, btree (product_no)
    "products_name_uk" UNIQUE CONSTRAINT, btree (name)
Check constraints:
    "products_price_min" CHECK (price > 0::numeric)

```

对于非空约束（`NOT NULL`），可以使用以下语法：

```
ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;
```

我们将产品表的 `name` 字段设置为非空：

```
testdb=> \d products;
               Table "public.products"
  Column      | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
product_no   | integer |           | not null |
name         | text    |           | not null |
price        | numeric |           |          |
description  | text    |           |          |
Indexes:
    "products_pkey" PRIMARY KEY, btree (product_no)
    "products_name_uk" UNIQUE CONSTRAINT, btree (name)
Check constraints:
    "products_price_min" CHECK (price > 0::numeric)
```

添加约束时，系统会检验已有数据是否满足条件，如果不满足将会添加失败。

#### 4.2.4 删除约束

删除约束通常需要知道它的名称，可以通过 `psql` 工具的 `\d table_name` 命令查看表的约束。

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ];
```

**RESTRICT** 是默认值，如果存在其他依赖于该约束的对象，需要使用 **CASCADE** 执行级联删除。例如，外键约束依赖于被引用字段上的唯一约束或主键约束。

我们删除产品表 `name` 字段上的唯一约束：

```
testdb=> ALTER TABLE products DROP CONSTRAINT products_name_uk;
ALTER TABLE
testdb=> \d products;
               Table "public.products"
  Column      | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
product_no   | integer |           | not null |
name         | text    |           | not null |
price        | numeric |           |          |
description  | text    |           |          |
Indexes:
    "products_pkey" PRIMARY KEY, btree (product_no)
Check constraints:
    "products_price_min" CHECK (price > 0::numeric)
```

删除非空约束也需要使用单独的语法：

```
ALTER TABLE table_name ALTER COLUMN column_name DROP NOT NULL;
```

以下语句将会删除产品表 `name` 字段上的非空约束：

```
testdb=> ALTER TABLE products ALTER COLUMN name DROP NOT NULL;
ALTER TABLE
testdb=> \d products;
               Table "public.products"
  Column      | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
product_no   | integer |           | not null |
name         | text    |           |          |
price        | numeric |           |          |
description  | text    |           |          |
```



```
Indexes:
    "products_pkey" PRIMARY KEY, btree (product_no)
Check constraints:
    "products_price_min" CHECK (price > 0::numeric)
```

## 4.2.5 修改字段默认值

如果想要为某个字段设置或者修改默认值，可以使用以下语句：

```
ALTER TABLE table_name ALTER COLUMN column_name SET DEFAULT value;
```

我们将为产品表的价格设置一个默认值：

```
testdb=> ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
ALTER TABLE
testdb=> \d products;
           Table "public.products"
  Column      | Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
product_no   | integer   |           | not null |
name         | text      |           |          |
price        | numeric   |           |          | 7.77
description  | text      |           |          |
Indexes:
    "products_pkey" PRIMARY KEY, btree (product_no)
Check constraints:
    "products_price_min" CHECK (price > 0::numeric)
```

同样，可以删除已有的默认值：

```
ALTER TABLE table_name ALTER COLUMN column_name DROP DEFAULT;
```

以下语句可以删除产品表中的价格默认值：

```
testdb=> ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
ALTER TABLE
testdb=> \d products;
           Table "public.products"
  Column      | Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
product_no   | integer   |           | not null |
name         | text      |           |          |
price        | numeric   |           |          |
description  | text      |           |          |
Indexes:
    "products_pkey" PRIMARY KEY, btree (product_no)
Check constraints:
    "products_price_min" CHECK (price > 0::numeric)
```

删除字段的默认值相当于将它设置为空值（NULL）。

## 4.2.6 修改字段数据类型

通常来说，可以将字段的数据类型修改为兼容的类型。

```
ALTER TABLE table_name ALTER COLUMN column_name TYPE new_data_type;
```

以下语句将产品表的 `price` 字段的类型修改为 `numeric(10,2)`：

```
testdb=> ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
ALTER TABLE
testdb=> \d products;
```

```
           Table "public.products"
  Column      |      Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
product_no   | integer        |           | not null |
name         | text           |           |          |
price        | numeric(10,2)  |           |          |
description  | text           |           |          |
Indexes:
    "products_pkey" PRIMARY KEY, btree (product_no)
Check constraints:
    "products_price_min" CHECK (price > 0::numeric)
```

因为已有的数据能够隐式转换为新的数据类型，上面的语句能够执行成功。如果无法执行隐式转换（例如将字符串‘1’转换为数字1），可以使用 **USING** 执行显式转换。

```
ALTER TABLE table_name ALTER COLUMN column_name TYPE new_data_type USING
expression;
```

我们先为产品表增加一个字符串类型的字段 **level**，然后将其修改为整数类型。

```
testdb=> ALTER TABLE products ADD COLUMN level VARCHAR(10);
ALTER TABLE
testdb=> ALTER TABLE products ALTER COLUMN level TYPE INTEGER;
ERROR: column "level" cannot be cast automatically to type integer
HINT: You might need to specify "USING level::integer".

testdb=> ALTER TABLE products ALTER COLUMN level TYPE INTEGER USING
level::integer;
ALTER TABLE
```

## 4.2.7 重命名字段

使用以下语句可以修改表中字段的名称：

```
ALTER TABLE table_name
RENAME COLUMN column_name TO new_column_name;
```

我们将产品表的字段 **product\_no** 改名为 **product\_number**：

```
ALTER TABLE products
RENAME COLUMN product_no TO product_number;
```

## 4.2.8 重命名表

如果需要修改表的名称，可以使用以下语句：

```
ALTER TABLE table_name
RENAME TO new_name;
```

例如，将产品表的名称改为 **items** 的命令如下：

```
ALTER TABLE products
RENAME TO items;
```

关于修改表的完整功能参考官方文档中的 [ALTER TABLE](#) 语句。

## 4.3 删除表

删除表可以使用 [DROP TABLE](#) 语句：

```
DROP TABLE [ IF EXISTS ] name [ CASCADE | RESTRICT ];
```

其中，`name` 表示要删除的表；如果使用了 `IF EXISTS`，删除一个不存在的表不会产生错误，而是显示一个信息。

以下语句将会删除表 `emp1`：

```
testdb=> DROP TABLE emp1;  
DROP TABLE
```

如果被删除的表存在依赖于它的视图或外键约束，需要指定 `CASCADE` 选项执行级联删除。

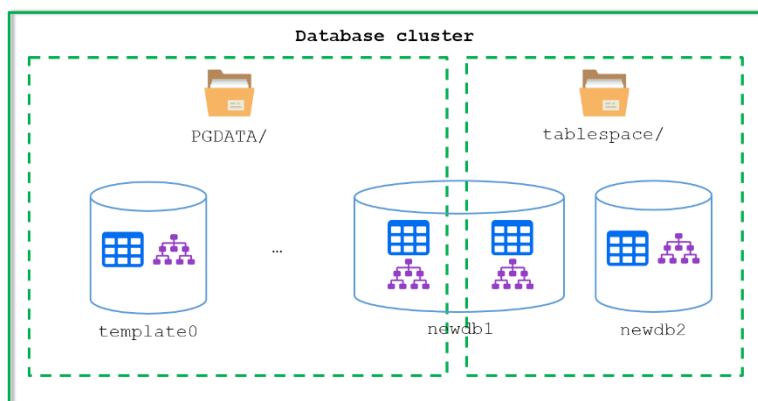
## 第 5 章 管理表空间


当我们使用 `CREATE TABLE` 语句创建一个新的表时, PostgreSQL 帮我们完成了所有的操作, 包括更新系统表中的信息, 在文件系统中生成相应的文件。那么 PostgreSQL 怎么知道应该在什么位置创建系统文件呢? 或者当我们查询表的数据时, 它如何知道数据文件所在的物理位置呢?

在 PostgreSQL 中, 它是通过表空间 (Tablespaces) 来实现逻辑对象 (表、索引等) 与物理文件之间的映射。创建数据库或者数据表 (包括索引) 的时候, 可以为其指定一个表空间 (tablespace)。表空间决定了这些对象在文件系统中的存储路径。现在我们来学习一些关于表空间的知识。

### 5.1 基本概念

在 PostgreSQL 中, 表空间 (tablespace) 表示数据文件的存放目录, 这些数据文件代表了数据库的对象, 例如表或索引。当我们访问表时, 系统通过它所在的表空间定位到对应数据文件所在的位置。



 PostgreSQL 中的表空间与其他数据库系统不太一样, 它更偏向于一个物理上的概念。

表空间的引入为 PostgreSQL 的管理带来了以下好处:

- 如果数据库集群所在的初始磁盘分区或磁盘卷的空间不足, 又无法进行扩展, 可以在其他分区上创建一个新的表空间以供使用。
- 管理员可以根据数据库对象的使用统计优化系统的性能。例如, 可以将访问频繁的索引存放到一个快速且可靠的磁盘上, 比如昂贵的固态硬盘。与此同时, 将很少使用或者对性能要求不高的归档数据表存储到廉价的低速磁盘上。

PostgreSQL 在集群初始化时将所有的数据文件和配置文件存储到它的数据目录中, 通常是环境变量 `PGDATA` 的值。默认创建了两个表空间:

- `pg_default`, `template1` 和 `template0` 默认的表空间, 也是创建其他数据库时的默认表空间; 对应的目录为 `PGDATA/base`。
- `pg_global`, 用于存储一些集群级别的共享系统表 (system catalogs), 例如 `pg_database`、`pg_control`; 对应的目录为 `PGDATA/global`。

初始安装后，使用 `psql` 查询默认创建的表空间：

```
postgres=# \db
              List of tablespaces
   Name   | Owner   | Location
-----+-----+-----
 pg_default | postgres |
 pg_global  | postgres |
(2 rows)
```

同时也可以通过操作系统命令查看相应的目录：

```
[postgres@localhost ~]$ ls -l /var/lib/pgsql/15/data/
total 68
drwx-----. 6 postgres postgres 46 12月 3 16:11 base
-rw-----. 1 postgres postgres 30 12月 3 15:39 current_logfiles
drwx-----. 2 postgres postgres 4096 12月 3 16:11 global
drwx-----. 2 postgres postgres 58 12月 3 15:39 log
drwx-----. 2 postgres postgres 6 11月 8 22:04 pg_commit_ts
drwx-----. 2 postgres postgres 6 11月 8 22:04 pg_dynshmem
-rw-----. 1 postgres postgres 4577 11月 8 22:04 pg_hba.conf
-rw-----. 1 postgres postgres 1636 11月 8 22:04 pg_ident.conf
drwx-----. 4 postgres postgres 68 12月 3 16:14 pg_logical
drwx-----. 4 postgres postgres 36 11月 8 22:04 pg_multixact
drwx-----. 2 postgres postgres 6 11月 8 22:04 pg_notify
drwx-----. 2 postgres postgres 6 11月 8 22:04 pg_replslot
drwx-----. 2 postgres postgres 6 11月 8 22:04 pg_serial
drwx-----. 2 postgres postgres 6 11月 8 22:04 pg_snapshots
drwx-----. 2 postgres postgres 6 12月 3 15:39 pg_stat
drwx-----. 2 postgres postgres 6 11月 8 22:04 pg_stat_tmp
drwx-----. 2 postgres postgres 18 11月 8 22:04 pg_subtrans
drwx-----. 2 postgres postgres 6 11月 8 22:04 pg_tblspc
drwx-----. 2 postgres postgres 6 11月 8 22:04 pg_twophase
-rw-----. 1 postgres postgres 3 11月 8 22:04 PG_VERSION
drwx-----. 3 postgres postgres 60 11月 8 22:04 pg_wal
drwx-----. 2 postgres postgres 18 11月 8 22:04 pg_xact
-rw-----. 1 postgres postgres 88 11月 8 22:04 postgresql.auto.conf
-rw-----. 1 postgres postgres 29459 11月 8 22:04 postgresql.conf
-rw-----. 1 postgres postgres 58 12月 3 15:39 postmaster.opts
-rw-----. 1 postgres postgres 102 12月 3 15:39 postmaster.pid
```

其中的 `base` 和 `global` 目录分别对应表空间 `pg_default` 和 `pg_global`。关于这些文件和目录的具体介绍，可以参考[官方文档](#)。

## 5.2 创建表空间

创建新的表空间使用 [CREATE TABLESPACE](#) 语句：

```
CREATE TABLESPACE tablespace_name
OWNER user_name
LOCATION 'directory';
```

表空间的名称不能以 `pg_` 开头，它们是系统表空间的保留名称；`LOCATION` 参数必须指定

绝对路径名，指定的目录必须是一个已经存在的空目录，PostgreSQL 操作系统用户（postgres）必须是该目录的拥有者，以便能够进行文件的读写。

接下来，我们使用目录/var/lib/pgsql/创建一个新的表空间 app\_tbs。先创建所需的目录：

```
[root@localhost ~]# su - postgres
Last login: Wed Jan  9 09:29:19 EST 2019 on pts/0
[postgres@localhost ~]$ mkdir /var/lib/pgsql/app_tbs
[postgres@localhost ~]$ ll
total 0
drwx-----. 4 postgres postgres 51 Jan  4 16:56 15
drwxr-xr-x. 2 postgres postgres  6 Jan 12 15:39 app_tbs
```

注意目录的所有者和权限。然后使用具有 CREATEDB 权限的用户创建表空间，此处我们使用 postgres 执行以下操作：

```
postgres=# CREATE TABLESPACE app_tbs LOCATION '/var/lib/pgsql/app_tbs';
CREATE TABLESPACE
postgres=# \db
          List of tablespaces
   Name   | Owner   | Location
-----+-----+-----
 app_tbs  | postgres | /var/lib/pgsql/app_tbs
 pg_default | postgres |
 pg_global | postgres |
(3 rows)

postgres=# SELECT oid,* FROM pg_tablespace;
   oid | spcname   | spcowner | spcacl | spcoptions
-----+-----+-----+-----+-----
  1663 | pg_default |      10 |        |
  1664 | pg_global  |      10 |        |
 16470 | app_tbs   |      10 |        |
(3 rows)
```

我们查看一下操作系统中的变化：

```
[postgres@localhost ~]$ ls -l /var/lib/pgsql/app_tbs/
total 0
drwx-----. 2 postgres postgres 6 12月  3 16:19 PG_15_202209061
```

在表空间对应的目录中，创建一个特定版本的子目录（PG\_‘Major version’\_‘Catalogue version number’）。

与此同时，在数据目录下的 pg\_tblspc 子目录中，创建了一个指向表空间目录的符号链接，名称为表空间的 OID（16470）：

```
[postgres@localhost ~]$ ls -l /var/lib/pgsql/15/data/pg_tblspc/
total 0
lrwxrwxrwx. 1 postgres postgres 22 Jan 12 15:42 16470 -> /var/lib/pgsql/app_tbs
```

默认情况下，执行 CREATE TABLESPACE 语句的用户为该表空间的拥有者，也可以使用 OWNER 选项指定拥有者。

对于普通用户，需要授予表空间上的对象创建权限才能使用该表空间。我们为用户 tony 授予表空间 app\_tbs 上的使用权限：

```

postgres=# GRANT CREATE ON TABLESPACE app_tbs TO tony;
GRANT
postgres=# \db+

```

List of tablespaces					
Name	Owner	Location	Access privileges	Options	
Size	Description				
app_tbs	postgres	/var/lib/pgsql/app_tbs	postgres=C/postgres+		
0 bytes			tony=C/postgres		
pg_default	postgres				46 MB
pg_global	postgres				574 kB
(3 rows)					

使用 **tony** 用户连接到数据库 **testdb**，然后在表空间 **app\_tbs** 中创建一个新的数据表 **t**：

```


testdb=> CREATE TABLE t(id int) tablespace app_tbs;
CREATE TABLE

testdb=> SELECT * FROM pg_tables WHERE tablename='t';

```

schemaname	tablename	tableowner	tablespace	hasindexes	hasrules	hastriggers	rowsecurity
public	t	tony	app_tbs	f	f	f	f

(1 row)

 PostgreSQL 支持在 **CREATE DATABASE**、**CREATE TABLE**、**CREATE INDEX** 以及 **ADD CONSTRAINT** 语句中指定 **tablespace\_name** 选项，覆盖默认的表空间（**pg\_default**）。也可以使用相应的 **ALTER ...** 语句将对象从一个表空间移到另一个表空间。

如果不想每次创建对象时手动指定表空间，可以使用配置参数 **default\_tablespace**：

```

testdb=> SET default_tablespace = app_tbs;
SET
testdb=> CREATE TABLE t1(id int);
CREATE TABLE

testdb=> SELECT * FROM pg_tables WHERE tablename='t1';

```

schemaname	tablename	tableowner	tablespace	hasindexes	hasrules	hastriggers	rowsecurity
public	t1	tony	app_tbs	f	f	f	f

(1 row)

对于临时表和索引，使用配置参数 **temp\_tablespaces** 进行控制，参考官方文档。

## 5.3 修改表空间

如果需要修改表空间的定义，可以使用 [ALTER TABLESPACE](#) 语句：

```
ALTER TABLESPACE name RENAME TO new_name;

ALTER TABLESPACE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER };

ALTER TABLESPACE name SET ( tablespace_option = value [, ... ] );
ALTER TABLESPACE name RESET ( tablespace_option [, ... ] );
```

第一个语句用于表空间的重命名；第二个语句用于修改表空间的拥有者；最后两个语句用于设置表空间的参数。

我们将表空间 `app_tbs` 重命名为 `tony_tbs`：

```
postgres=# \db
              List of tablespaces
   Name   | Owner   | Location
-----+-----+-----
 app_tbs  | postgres | /var/lib/pgsql/app_tbs
 pg_default | postgres |
 pg_global | postgres |
(3 rows)

postgres=# ALTER TABLESPACE app_tbs RENAME TO tony_tbs;
ALTER TABLESPACE
postgres=# \db
              List of tablespaces
   Name   | Owner   | Location
-----+-----+-----
 pg_default | postgres |
 pg_global | postgres |
 tony_tbs  | postgres | /var/lib/pgsql/app_tbs
(3 rows)
```

只有表空间的拥有者或超级用户才能修改表空间的定义。

接下来将表空间 `tony_tbs` 的拥有者修改为 `tony`：

```
postgres=# ALTER TABLESPACE tony_tbs OWNER TO tony;
ALTER TABLESPACE
postgres=# \db
              List of tablespaces
   Name   | Owner   | Location
-----+-----+-----
 pg_default | postgres |
 pg_global | postgres |
 tony_tbs  | tony    | /var/lib/pgsql/app_tbs
(3 rows)
```

PostgreSQL 支持设置的表空间参数包括 [seq\\_page\\_cost](#)、[random\\_page\\_cost](#) 以及 [effective\\_io\\_concurrency](#)。它们用于查询计划器选择执行计划时的代价评估。

目前，PostgreSQL 还不支持使用语句修改表空间的存储路径。但是，可以通过手动的方式



移动表空间的位置:

1. 停止 PostgreSQL 服务器进程;
  2. 移动文件系统中的数据文件位置;
  3. 修改 PGDATA/pg\_tblspc 目录中的符号链接文件 (需要提前获取文件名), 指向新的目录;
  4. 启动 PostgreSQL 服务器进程。
- 首先, 停止 PostgreSQL 服务器进程:

```
[postgres@localhost ~]$ whoami
postgres
[postgres@localhost ~]$ /usr/pgsql-15/bin/pg_ctl stop
waiting for server to shut down.... done
server stopped
```

然后, 将操作系统中的/var/lib/pgsql/app\_tbs/目录移动到/var/lib/pgsql/tony\_tbs:

```
[postgres@localhost ~]$ mv /var/lib/pgsql/app_tbs/ /var/lib/pgsql/tony_tbs
```

更新符号链接文件, 执行新的目录:

```
[postgres@localhost ~]$ ln -snf /var/lib/pgsql/tony_tbs
/var/lib/pgsql/15/data/pg_tblspc/16470
[postgres@localhost ~]$ ls /var/lib/pgsql/15/data/pg_tblspc/16470 -l
lrwxrwxrwx. 1 postgres postgres 23 Jan 14 20:21
/var/lib/pgsql/15/data/pg_tblspc/16470 -> /var/lib/pgsql/tony_tbs
```

最后, 重新启动 PostgreSQL 服务器进程:

```
[postgres@localhost ~]$ /usr/pgsql-15/bin/pg_ctl start
waiting for server to start....2019-01-14 20:23:43.628 EST [20994] LOG:
listening on IPv4 address "192.168.56.101", port 5432
2022-11-14 20:23:43.632 EST [20994] LOG: listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2022-11-14 20:23:43.639 EST [20994] LOG: listening on Unix socket
"/tmp/.s.PGSQL.5432"
2022-11-14 20:23:43.661 EST [20994] LOG: redirecting log output to logging
collector process
2022-11-14 20:23:43.661 EST [20994] HINT: Future log output will appear in
directory "log".
done
server started
```

确认是否修改成功:

```
postgres=# \db
          List of tablespaces
  Name   | Owner   | Location
-----+-----+-----
pg_default | postgres |
pg_global | postgres |
tony_tbs  | tony    | /var/lib/pgsql/tony_tbs
(3 rows)
```

⚠ 在 PostgreSQL 9.1 及更早的版本中, 还需要用新的目录更新系统表 pg\_tablespace; 否则, pg\_dump 将继续使用旧的表空间位置。

## 5.4 删除表空间

对于不再需要的表空间，可以使用 [DROP TABLESPACE](#) 语句进行删除：

```
DROP TABLESPACE [ IF EXISTS ] name;
```

**IF EXISTS** 可以避免删除不存在的表空间时产生错误信息。

只有表空间的拥有者或超级用户能够删除表空间。删除表空间之前需要确保其中不存在任何数据库对象，否则无法删除。

```
testdb=> DROP TABLESPACE tony_tbs;
ERROR:  tablespace "tony_tbs" is not empty
```

无法删除表空间 **tony\_tbs** 是因为数据库 **testdb** 中存在使用该表空间创建的对象。

```
testdb=> SELECT ts.spcname,
testdb->         cl.relname
testdb->   FROM pg_class cl
testdb->   JOIN pg_tablespace ts ON cl.reltablespace = ts.oid
testdb->  WHERE ts.spcname = 'tony_tbs';
 spcname | relname
-----+-----
 tony_tbs | t
 tony_tbs | t1
(2 rows)
```

可以将这些对象删除，或者移动到其他表空间中，然后再删除表空间：

```
testdb=> DROP TABLE t, t1;
DROP TABLE
testdb=> DROP TABLESPACE tony_tbs;
DROP TABLESPACE
testdb=> \db
      List of tablespaces
   Name   | Owner   | Location
-----+-----+-----
 pg_default | postgres |
 pg_global  | postgres |
(2 rows)
```

其他数据库中也可能存在依赖于被删除表空间的对象，同样需要先进行处理，才能删除表空间。

删除表空间时，同时会删除文件系统中对应的表空间子目录。

关于表空间，我们先了解这么多。接下来我们探讨一下如何防止因系统崩溃、硬件故障或者用户错误可能带来的数据丢失，也就是 PostgreSQL 数据库的备份与恢复。

## 第 6 章 备份与恢复

### 6.1 基本概念

服务器系统错误、硬件故障或者人为失误都可能导致数据的丢失或损坏。因此，备份和恢复对于数据库的高可用性至关重要。数据库管理员应该根据业务的需求制定合适的备份策略，并提前演练各种故障情况下的恢复过程，做到有备无患。

在升级 PostgreSQL 版本之前，通常也需要先进行数据库的备份。另外，备份也可以用于主从复制结构中的从节点初始化。

**备份（backup）**是通过某种方式（物理复制或者逻辑导出）将数据库的文件或结构和数据拷贝到其他位置进行存储。

**还原（restore）**是一种不完全的恢复，使用备份的文件将数据库恢复到执行备份时的状态。备份时间点之后的数据变更无法通过还原进行恢复。

**恢复（recovery）**通常是先使用物理备份文件进行还原，然后再应用备份时间点到故障点之间的日志文件（WAL），将数据库恢复到最新状态。

#### 6.1.1 备份类型

根据备份的方式和内容不同，可以进行以下分类。

##### 6.1.1.1 物理备份与逻辑备份

**物理备份（Physical Backup）**就是直接复制数据库相关的文件。通常来说，物理备份比逻辑备份更快，但是占用的空间也更大。PostgreSQL 支持在线和离线的物理备份。实际环境中应该以物理备份为主。

**逻辑备份（Logical Backup）**就是将数据库的结构和数据导出为 SQL 文件，还原时通过文件中的 SQL 语句和命令重建数据库并恢复数据。逻辑备份通常需要更多的备份和还原时间。逻辑备份可以作为物理备份的补充，或者用于测试目的的数据导入导出。

##### 6.1.1.2 在线备份与离线备份

**在线备份（Online Backup）**是指 PostgreSQL 服务器处于启动状态时的备份，也称为热备份（Hot Backup）。由于逻辑备份需要连接到数据库进行操作，因此逻辑备份只能是在线备份。

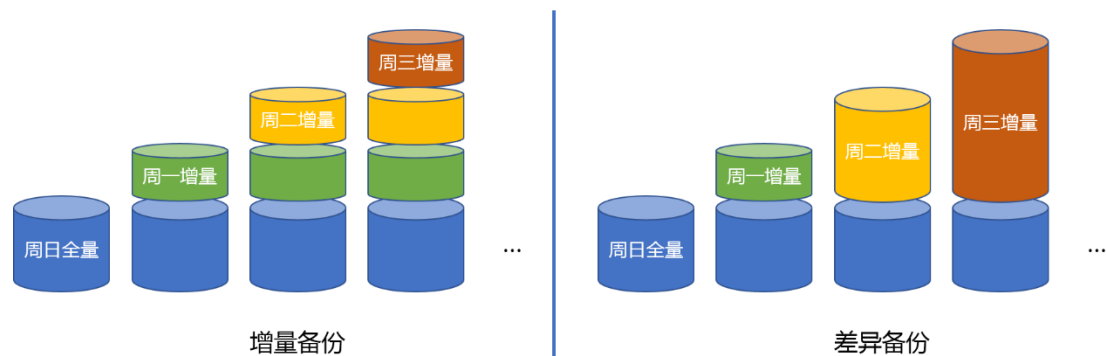
**离线备份（Offline Backup）**是指 PostgreSQL 服务器处于关闭状态时的备份，也称为冷备份（Cold Backup）。离线状态只能执行数据库的物理备份，即复制数据库文件。

##### 6.1.1.3 全量备份与增量备份

**全量备份（Full Backup）**就是备份所有的数据库文件，执行一次完整的 PostgreSQL 数据库

集群备份。这种方式需要备份的内容较多，备份时较慢，但是恢复速度更快。

**增量备份 (Incremental Backup)** 就是备份上一次备份（任何类型）之后改变的文件。另外，**差异备份 (Differential Backup)** 是针对上一次完全备份后发生变化的所有文件进行备份。增量备份每次备份的数据量较小，但是恢复时需要基于全量备份，并依次恢复增量部分，时间较长。差异备份位于两者之间。

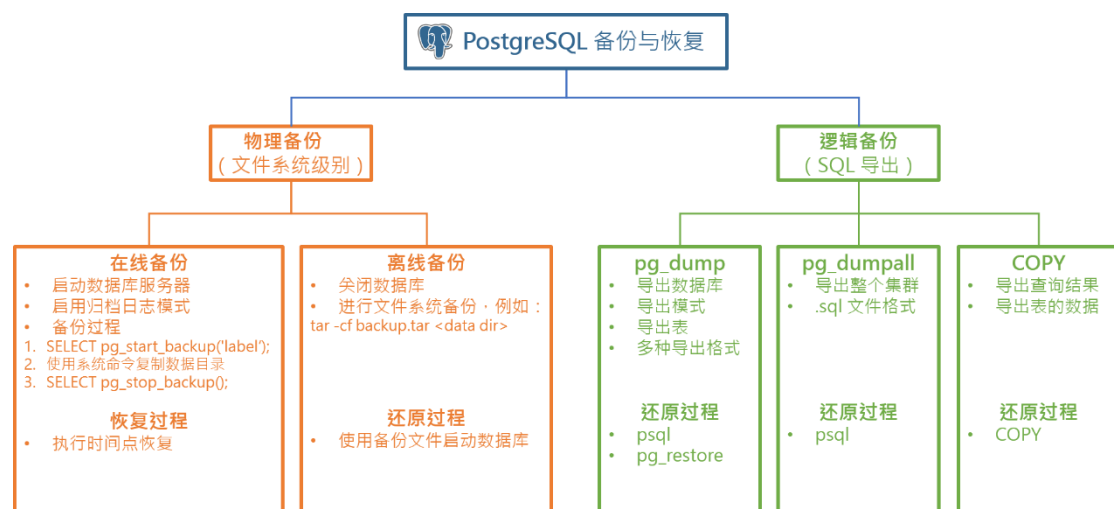


在 PostgreSQL 中通过一个基准备份 (Base Backup)，加上不断备份的事务日志文件 (WAL) 来达到增量备份的效果。

在实际环境中可以结合全量备份和增量/差异备份，以平衡备份和恢复所需的存储空间和时间。

## 6.1.2 备份恢复工具

PostgreSQL 为不同的备份和恢复类型提供了相应的工具和方法。



- **pg\_dump**，逻辑备份工具，支持单个数据库（可以指定模式、表）的导出，可以选择导出的格式。
- **pg\_dumpall**，逻辑备份工具，用于导出整个数据库集群，包括公用的全局对象。
- **pg\_basebackup**，物理备份工具，为数据库集群创建一个基准备份。它也可以用于时间点恢复 (point-in-time recovery) 的基准备份，或者设置基于日志传输或流复制的从节点的初始化。
- **psql**，PostgreSQL 交互式命令行工具，也可以用于导入逻辑备份产生的 SQL 文件。

- [pg\\_restore](#)，逻辑还原工具，用于还原 `pg_dump` 导出的归档格式的备份文件。
- [COPY](#)，PostgreSQL 专有的 SQL 语句，将表中的数据复制到文件，或者将文件中的数据复制到表中。

此外，还可以通过第三方工具执行备份与恢复操作。

- pgAdmin（开源）
- Barman（开源）
- pg\_probackup（开源）
- pgBackRest（开源）
- BART（商业）

接下来，我们介绍如何利用 PostgreSQL 自带的工具和命令执行备份与恢复操作。

## 6.2 备份与恢复

### 6.2.1 逻辑备份与还原

执行逻辑备份时，PostgreSQL 服务器必须已经启动，备份工具（例如 `pg_dump`）通过建立数据库连接，从数据库中查询出相应的结构信息和数据，并生成备份文件。针对不同的备份格式，PostgreSQL 提供了配套的还原工具。

#### 6.2.1.1 备份单个数据库

PostgreSQL 提供了备份单个数据库的工具 `pg_dump`。它支持三种文件格式：

- **plain**，文本格式，输出一个纯文本形式的 SQL 脚本，默认值。还原时直接使用 `psql` 工具导入。
- **custom**，自定义格式，输出一个自定义格式的归档文件，还原时使用 `pg_restore` 工具。与目录导出格式结合使用时，提供了最灵活的输出格式，它允许在恢复时手动选择和排序已归档的项。这种格式在默认情况还会进行文件的压缩。
- **directory**，目录格式，输出一个目录格式的归档，还原时使用 `pg_restore` 工具。这种格式将会创建一个目录，为每个导出的表和大对象都创建一个文件，另外再创建一个内容目录文件，该文件使用自定义格式存储关于导出对象的描述。这种格式在默认情况还会进行文件的压缩，并且支持并行导出。
- **tar**，打包格式，输出一个 `tar` 格式的归档，还原时使用 `pg_restore` 工具。这种格式与目录格式兼容，解压一个 `tar` 格式的归档将会产生一个目录格式的归档。但是，`tar` 格式不支持压缩。另外，在使用 `tar` 格式归档进行还原时，表数据项的相对顺序不能进行改动。

接下来我们看一些示例。首先，使用 `pg_dump` 文本格式导出数据库 `testdb`：

```
[postgres@localhost ~]$ whoami
postgres
[postgres@localhost ~]$ pg_dump testdb > testdb.sql
```

以上命名将会创建一个 sql 文件 `testdb.sql`，其中包含了重建数据库 `testdb` 中对象所需的脚本，包括表中的数据。

关于 `pg_dump` 工具的各种选项，可以参考[官方文档](#)。

对于 `sql` 文件格式的备份，还原时直接使用 `psql` 导入相关文件即可：

```
[postgres@localhost ~]$ psql newdb -f testdb.sql
SET
SET
SET
...
```

以上命令将 `testdb.sql` 中的内容还原到数据库 `newdb` 中。

关于 `psql` 工具的使用可以参考[官方文档](#)。

`pg_dump` 和 `psql` 支持的读写管道功能使得我们可以直接将数据库从一个服务器导出到另一个服务器，例如：

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

接下来，使用自定义格式导出数据库 `testdb`：

```
[postgres@localhost ~]$ pg_dump -Fc testdb -f testdb.dmp
```

其中，`-Fc` 指定自定义格式，`-f` 指定导出的文件名。这种格式的备份，还原时需要使用 PostgreSQL 提供的 `pg_restore` 工具。

```
[postgres@localhost ~]$ pg_restore -d newdb testdb.dmp
```

其中，`-d` 表示还原到数据库 `newdb` 中。

关于 `pg_restore` 工具的使用和选项，可以参考[官方文档](#)。

继续使用目录格式导出数据库 `testdb`：

```
[postgres@localhost ~]$ pg_dump -Fd testdb -f testdb_dir
```

其中，`-Fd` 指定目录格式，`-f` 指定导出的文件目录。这种格式的备份，还原时需要使用 PostgreSQL 提供的 `pg_restore` 工具。

```
[postgres@localhost ~]$ pg_restore -d newdb testdb_dir
```

使用 `tar` 格式导出数据库 `testdb`：

```
[postgres@localhost ~]$ pg_dump -Ft testdb -f testdb.tar
```

其中，`-Ft` 指定 `tar` 格式，`-f` 指定导出的文件名。这种格式的备份，还原时需要使用 PostgreSQL 提供的 `pg_restore` 工具。

```
[postgres@localhost ~]$ pg_restore -d newdb testdb.tar
```

#### 6.2.1.2 备份整个集群

`pg_dump` 每次只导出一个数据库，而且它不会导出角色或表空间（属于集群范围）相关的信息。为此，PostgreSQL 还提供了导出数据库集群的 `pg_dumpall` 工具。它会针对集群中的每个数据库调用 `pg_dump` 来完成导出工作，同时还导出所有数据库公用的全局对象（`pg_dump` 不保存这些对象），包括数据库用户和组、表空间以及所有数据库的访问权限等属性。

我们使用 `pg_dumpall` 导出整个数据库集群：

```
[postgres@localhost ~]$ pg_dumpall -f cluster.sql
```

关于 `pg_dumpall` 工具的各种选项和使用方法，可以参考[官方文档](#)。

因为 `pg_dumpall` 从所有数据库中读取表，所以需要以一个超级用户的身份连接以便生成完整的导出操作。同样，还原时也需要超级用户特权执行备份的脚本，这样才能增加用户和组以及创建数据库。

`pg_dumpall` 导出 `sql` 文件格式的备份，还原时直接使用 `psql` 导入相关文件即可。

```
[postgres@localhost ~]$ psql -f cluster.sql postgres
```

以上命令通过 `psql` 工具连接到数据库 `postgres`，然后运行 `cluster.sql` 文件还原之前的备份。

### 6.2.1.3 导出表数据

如果只需要导出指定表中的数据（例如为了测试而迁移数据），PostgreSQL 至少提供了以下两种方法：

- 使用 `pg_dump` 中导出表的选项 `-t` 和 `-T`；
- 使用 `COPY` 命令复制数据

首先使用 `pg_dump` 导出数据库 `testdb` 中以 `emp` 开头的表（排除表 `employees`）中的数据：

```
[postgres@localhost ~]$ pg_dump -a -t 'emp*' -T employees testdb > testdb_table.sql
```

其中，`-a` 表示只导出数据（不包含结构），`-t` 指定要导出的表，`-T` 表示排除的表。导出时也可以指定其他的导出格式，并且采用相应的方式进行数据导入，参考上文。

另外，使用 `COPY` 命令可以导出单个表中的数据或查询结果集：

```
postgres=# \c testdb;
You are now connected to database "testdb" AS user "postgres".
testdb=# COPY products to '/var/lib/pgsql/products.dat';
COPY 1
```

`COPY` 支持不同的写入/读取文件格式：`text`、`csv` 或者 `binary`。默认是 `text`。

对于 `COPY` 导出的文件，同样使用 `COPY` 命令进行导入。

```
testdb=# COPY products FROM '/var/lib/pgsql/products.dat';
```

以上语句将指定文件中的数据导入（追加）表 `products` 中。

关于 `COPY` 命令的使用，可以参考[官方文档](#)。

### 6.2.1.4 备份大型数据库

对于大型数据库，在执行备份和恢复操作时，可能会面临两个问题：备份文件过大，或者操作时间过长。

对于文件过大的问题，`pg_dump` 和 `pg_dumpall` 支持将内容写到标准输出，结合操作系统提供的一些工具，可以进行以下处理：

导出压缩格式的文件，以下命令利用 `gzip` 工具对导出文件进行压缩。

```
[postgres@localhost ~]$ time pg_dumpall > cluster.sql

real    2m18.435s
user    0m32.315s
sys     0m49.526s
[postgres@localhost ~]$ time pg_dumpall | gzip > cluster.sql.gz

real    6m20.461s
user    6m38.507s
sys     0m47.674s

[postgres@localhost ~]$ ls -lh
total 23G
drwx----- 6 postgres postgres 81 Dec 5 11:18 11
drwx----- 2 postgres postgres 6 Dec 8 2017 backups
-rw-r--r-- 1 postgres postgres 22G Mar 6 04:00 cluster.sql
-rw-r--r-- 1 postgres postgres 1.8G Mar 6 04:33 cluster.sql.gz
drwx----- 2 postgres postgres 6 Dec 8 2017 data
```

可以看出，压缩需要占用更多的时间，但是导出的备份文件更小。

还原时先使用 `gunzip` 解压再导入：

```
[postgres@localhost ~]$ gunzip -c cluster.sql.gz | psql newdb
```

**分割导出的文件**，利用操作系统提供的工具，例如 `split`，将导出的文件分割成多个小的文件：

```
[postgres@localhost ~]$ pg_dumpall | split -b 1G - cluster
```

以上命令使用 `pg_dumpall` 导出整个数据库集群，并且利用 `split` 将备份文件分割为多个 1 GB 的文件（`clusteraa`、`clusterab` 等）。还原时导入所有的备份文件即可：

```
[postgres@localhost ~]$ cat cluster* | psql newdb
```

`COPY` 命令也提供了类似的压缩功能：

```
COPY products TO PROGRAM 'gzip > /var/lib/pgsql/products.dat.gz';
```

还原时也一样：

```
COPY products FROM PROGRAM 'gunzip < /var/lib/pgsql/products.dat.gz';
```

**pg\_dump 自定义格式导出**，如果 PostgreSQL 所在的系统上安装了 `zlib` 压缩库，导出自定义格式时将会对输出文件进行压缩。这种方式和使用 `gzip` 压缩的效果类似，但是它还有一个优点就是还原时可以选择指定的表。

对于超大型的数据库，可以将 `split` 与其他两种方法配合使用。

**pg\_dump 并行导出**，并行导出可以同时备份多个表。使用 `-j` 参数控制并发数，并行导出只支持“目录”导出格式。

```
[postgres@localhost ~]$ time pg_dump -F d -f out.dir testdb

real    10m22.901s
user    10m13.333s
sys     0m8.830s

[postgres@localhost ~]$ time pg_dump -j 8 -F d -f out.dir testdb
real    2m31.607s
```



```
user    2m27.522s
sys     0m4.010s
```

还原时也可以使用 `pg_restore -j` 指定并发导入。它只能用于自定义或目录格式的导出文件，无论这些归档是否由 `pg_dump -j` 创建。

## 6.2.2 物理备份与恢复

### 6.2.2.1 离线备份

离线备份，也称为冷备份，通过文件系统级别的复制备份数据库（/data 目录）。这种备份方式要求关闭 PostgreSQL 实例服务，以便创建一个一致的数据备份。PostgreSQL 可以通过软链接的方式将 `pg_wal`（`pg_xlog`）和 `pg_tblspce` 存储到其他的挂载设备。

执行以下步骤创建一个 PostgreSQL 冷备份，备份数据中包含了软链接指向的数据内容：

1. 关闭 PostgreSQL 服务：

```
[postgres@localhost ~]$ pg_ctl stop -D /var/lib/pgsql/15/data/
```

2. 执行操作系统命令，拷贝数据目录：

```
[postgres@localhost ~]$ tar czf backup.tar.gz /var/lib/pgsql/15/data
```

也可以使用 `cp` 或者 `rsync` 等命令复制数据目录。

3. 完成备份之后，重新启动 PostgreSQL 服务。

```
[postgres@localhost ~]$ pg_ctl start -D /var/lib/pgsql/15/data/
```

离线备份只能备份整个数据库集群，还原时也一样。执行还原时可以直接使用备份文件启动 PostgreSQL 服务：

1. 关闭 PostgreSQL 服务（也可以启动另一个实例服务管理备份的数据目录）：

```
[postgres@localhost ~]$ pg_ctl stop -D /var/lib/pgsql/15/data/
```

2. 使用备份的数据目录启动实例服务：

```
[postgres@localhost ~]$ pwd
```

```
/var/lib/pgsql
```

```
[postgres@localhost ~]$ tar xzf backup.tar.gz
```

```
[postgres@localhost ~]$ /usr/pgsql-15/bin/pg_ctl start -D ./var/lib/pgsql/15/data/
```

使用冷备份执行的还原只能恢复到备份点时的状态，备份点到故障点之间的任何修改都不会恢复。另外，还原时不会重新创建软链接，而是直接使用备份时的生成数据目录。

### 6.2.2.2 在线备份

对于前面介绍的逻辑备份和物理冷备份，都是基于一个时间点的备份，还原时只能恢复到该时间点。这些备份的一个主要缺点就是无法恢复备份点之后的数据变更。为了实现完全的数据恢复，确保不丢失任何变更和数据，PostgreSQL 提供了在线增量备份的功能。

PostgreSQL 对于数据的修改，都会写入一个称为预写式日志（WAL）的文件中，该文件位于数据目录的 `pg_wal`（`pg_xlog`）子目录中。当系统出现故障需要恢复时，可以通过重做最后一次检查点（checkpoint）以来的 WAL 日志执行数据库的恢复。基于这个机制，我们可以先创建

一个文件级别的完全备份，然后不断备份生成的 WAL 文件，达到增量备份的效果。这些备份操作都可以在线执行，因此对于无法接受停机的业务系统至关重要。

预写式日志（Write-Ahead Logging）是实现可靠性和数据完整性的标准方法，同时还能减少磁盘 IO 的操作，提高数据库的性能。可以参考[官方文档](#)。

## 归档日志模式

对于在线备份而言，数据库集群处于运行状态，而且必须使用归档日志模式（Archive Log）运行，这样才能不断备份（归档）生成的 WAL 日志文件。因此，我们首先来设置 WAL 归档。

启用 PostgreSQL 的 WAL 归档需要在 postgresql.conf 文件中配置三个参数：

- [wal\\_level](#)，决定写入 WAL 的信息量。可选值为 minimal、replica 以及 logical，默认值为 replica。启用 WAL 归档需要设置为 replica 或更高配置。在 PostgreSQL 9.6 之前的版本中，还允许设置为 archive 和 hot\_standby。仍然可以设置这两个值，但它们直接映射到 replica。
- [archive\\_mode](#)，是否启动日志归档。默认值为 off。如果设置为 on 或者 always，可以通过设置下面的归档命令参数 archive\_command 执行已完成的 WAL 段文件的归档操作。
- [archive\\_command](#)，执行日志归档操作的脚本命令。例如操作系统的 cp 命令。

修改 postgresql.conf 文件，添加以下内容：

```
wal_level = replica
archive_mode = on
archive_command = 'test ! -f /var/lib/pgsql/wal_archive/%f && cp %p
/var/lib/pgsql/wal_archive/%f'
archive_timeout = 300
```

在 archive\_command 中使用了两个占位符，%p 代表要归档的日志文件路径名称（相对于当前工作目录，即数据目录的文件名），%f 代表要归档的日志文件名（不包含路径）。如果需要在命令中使用 % 符号，可以使用两个连写的 %% 表示。以上命令会在执行时生成许多类似下面的实际操作：

```
test ! -f /var/lib/pgsql/wal_archive/000000010000000000000001 && cp
pg_wal/000000010000000000000001
/var/lib/pgsql/wal_archive/000000010000000000000001
```

我们还需要提前创建好 /var/lib/pgsql/wal\_archive 目录，并且确保 PostgreSQL 实例的管理用户（默认为 postgres）具有该目录的读写权限。

PostgreSQL 只会针对已经完成的 WAL 段进行归档操作，而 archive\_timeout 确保了即使某个 WAL 日志长时间没有填满（业务不繁忙），最多 5 分钟之后，它也会被归档。需要注意的是，未完成的 WAL 大小也是一样大；因此，过小的 archive\_timeout 可能会导致生成大量的 WAL 文件。

以上参数的修改需要重新启动 PostgreSQL 实例服务：

```
[postgres@localhost ~]$ pg_ctl restart
```

每过 5 分钟（archive\_timeout），就会产生一次 WAL 段文件的切换，同时对该文件进行归档。也可以使 [pg\\_switch\\_wal](#) 函数执行手动切换。

```
postgres=# SELECT * FROM pg_switch_wal();
pg_switch_wal
-----
0/40001C8
(1 row)
```

PosgreSQL 提供了一个视图 `pg_stat_archiver`, 用于查看 WAL 归档状态:

```

postgres=# SELECT * FROM pg_stat_archiver;
 archived_count | last_archived_wal | last_archived_time |
failed_count | last_failed_wal | last_failed_time |
stats_rese
t
-----+-----+-----
--+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
-----
4 | 00000001000000000000000000000004 | 2019-03-25 22:36:16.462352-04 |
0 | | | 2019-01-04 16:57:12.471428-05
(1 row)

```

至此，我们完成了 WAL 归档的设置和验证。

下一步就是执行基准备份了，基准备份是一个全量备份。执行基准备份最简单的方法就是使用 `pg_basebackup` 工具，它也可以用于流复制结构中的从节点初始化，我们在下一章进行介绍。这里我们使用另外一种创建备份的方式，即调用两个系统函数：~~`pg_start_backup()` 和 `pg_stop_backup()`~~。

首先，调用 `pg_start_backup()` 函数开始一个备份：

```
postgres=# SELECT pg_start_backup('basebackup20190328');
pg_start_backup
-----
0/2E000060
(1 row)
```

其中的参数‘basebackup20190328’是本次备份的标识。该函数会在集群的数据目录中创建一个名称为 backup\_label 的备份标签文件，记录本次备份的相关信息，包括起始时间和备份标识。另外，如果存在自定义的表空间，还会创建一个表空间映射文件，名称为 tablespace\_map，记录 pg\_tblspc/中的表空间符号链接的信息。这两个文件对于备份的完整性至关重要。

这种备份方式是排他性的备份，当前备份执行的过程中不允许运行其他备份。在 PostgreSQL 9.6 之后，支持非排他性的备份方式，参考[官方文档](#)。

执行完以上命令之后, PostgreSQL 会进入备份模式, 然后就可以使用操作系统命令(cp、tar、rsync 等)复制数据目录:

```

[postgres@localhost ~]$ cp -r /var/lib/pgsql/15/data/
/var/lib/pgsql/14/backups/basebackup20190328/
[postgres@localhost ~]$ ls /var/lib/pgsql/14/backups/basebackup20190328/
backup_label      global            pg_dynshmem      pg_logical        pg_replslot
pg_stat           pg_tblspc        pg_wal           postgresql.conf   tablespace_map
base              log              pg_hba.conf      pg_multixact      pg_serial
pg_stat_tmp       pg_twophase       pg_xact          postmaster.opts
current_logfiles  pg_commit_ts     pg_ident.conf    pg_notify         pg_snapshots
pg_subtrans       PG_VERSION       postgresql.auto.conf postmaster.pid

```

此时的文件备份可能不是一致性备份，因为备份的同时某些文件可能会被修改。但是这些不一致性可以通过 WAL 日志重放进行恢复，就像发生系统故障时的恢复一样。

最后，调用 `pg_stop_backup()` 函数结束备份：

```
postgres=# SELECT pg_stop_backup();
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
pg_stop_backup
-----
0/2E000168
(1 row)
```

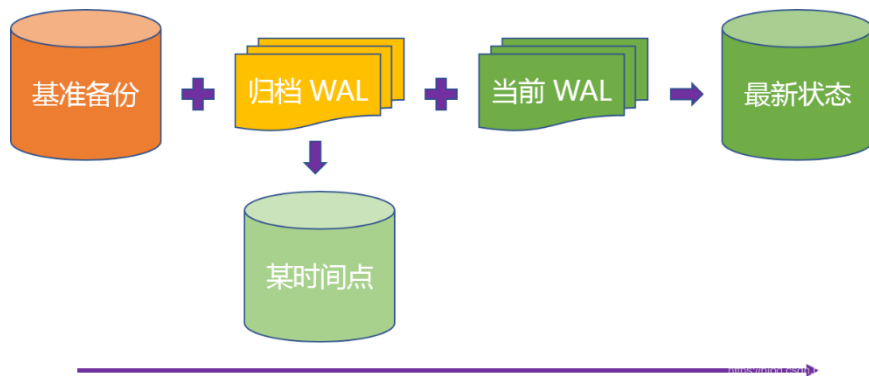
该函数将会终止备份模式，并且执行一次 WAL 段文件切换。切换文件是为了准备归档备份过程中写入过的最后一个 WAL 段。该函数返回了一些关于此次备份的信息。另外，PostgreSQL 还会在 `pg_wal`（`pg_xlog`）目录中生成一个 `'wal-segement-number.backup'` 文件（`000000010000000000000002E.00000060.backup`），其中存储了本次备份的历史信息。

一旦完成最后的 WAL 段文件归档，备份过程就结束了。`pg_stop_backup` 返回的结果就是组成一个完整备份所需的最后一个段文件。如果启用了 `archive_mode`，`pg_stop_backup` 将会等待最后一个 WAL 段文件归档后才返回。

PostgreSQL 没有提供备份目录（`backup catalog`）的功能，用于记录这些备份历史信息。但是某些第三方工具（例如 `Barman`）提供了这些高级功能。

### 6.2.2.3 时间点恢复

有了基准备份和连续的 WAL 归档，再加上当前使用的 WAL 段文件，PostgreSQL 可以恢复到最新的状态。不仅如此，我们还可以通过 WAL 回放到上次备份以来的任意时间点，也就是时间点恢复（`Point-in-Time Recovery`）。



接下来我们基于上面的备份执行一次恢复操作。开始之前创建一个测试表：

```
postgres=# \c testdb
You are now connected to database "testdb" AS user "postgres".
testdb=# create table t1(id int,v varchar(10));
CREATE TABLE
testdb=# INSERT INTO t1(id, v) values(1, 'pg');
INSERT 0 1
```

这个表是在备份之后创建的，只有恢复到最新状态才能看到。

首先，我们模仿系统故障，直接关闭服务器进程，并且删除数据目录中的文件（需要保留 `pg_wal` 子目录，因为完全恢复需要使用当前还未归档的 WAL 文件）。

```
[postgres@localhost ~]$ ps -ef|grep /usr/pgsql-15/bin/postgres
postgres 6471 32086 0 02:46 pts/1 00:00:00 grep --color=auto
/usr/pgsql-15/bin/postgres
postgres 32313 1 0 Mar25 pts/1 00:00:16 /usr/pgsql-15/bin/postgres -D
/var/lib/pgsql/15/data/
[postgres@localhost ~]$ kill -9 32313
[postgres@localhost ~]$ ps -ef|grep postgres
postgres 7058 32086 0 02:48 pts/1 00:00:00 ps -ef
postgres 7059 32086 0 02:48 pts/1 00:00:00 grep --color=auto postgres
root 32083 7880 0 Mar25 pts/1 00:00:00 su - postgres
postgres 32086 32083 0 Mar25 pts/1 00:00:01 -bash

[postgres@localhost ~]$ mkdir /var/lib/pgsql/15/data_old
[postgres@localhost ~]$ mv /var/lib/pgsql/15/data/*
/var/lib/pgsql/15/data_old
```

如果此时尝试重启 PostgreSQL 服务进程，提示目录不是数据库集群目录：

```
[postgres@localhost ~]$ pg_ctl start
pg_ctl: directory "/var/lib/pgsql/15/data" is not a database cluster directory
```

还原前面执行的基准备份，需要注意的是使用数据库的拥有者（`postgres`）执行操作。如果使用了自定义的表空间，确认 `pg_tblspc` 子目录中的符号链接正确还原：

```
[postgres@localhost ~]$ cp -R /var/lib/pgsql/14/backups/basebackup20190328/* /var/lib/pgsql/15/data/
[postgres@localhost ~]$ ls /var/lib/pgsql/15/data/pg_tblspc/ -l
total 0
lrwxrwxrwx. 1 postgres postgres 23 Mar 28 03:21 16558 ->
/var/lib/pgsql/tony_tbs
```

删除还原的 `pg_wal` 子目录中的文件，这些文件是在执行基准备份时复制生成的，对于恢复没有作用。

```
[postgres@localhost ~]$ rm -rf /var/lib/pgsql/15/data/pg_wal/*
```

将前面保留的未归档的 WAL 段文件复制到 `pg_wal` 子目录（最好使用复制，而不是移动文件，这样的话在恢复失败后还可以重新尝试）。

```
[postgres@localhost ~]$ cp -R /var/lib/pgsql/15/data_old/pg_wal/*
/var/lib/pgsql/15/data/pg_wal/
```

在 `postgresql.conf` 文件中配置 WAL 恢复相关的参数：

```
restore_command = 'cp /var/lib/pgsql/wal_archive/%f %p'
#recovery_target_time = "2019-3-28 12:05 GMT"
```

`restore_command` 用于指定还原归档 WAL 文件的命令，基本上与 `archive_command` 的设置相反。`recovery_target_time` 用于设置恢复目标，即恢复到哪个时间点。PostgreSQL 支持多种恢复目标的设置，默认为恢复到最新的状态。详细的参数介绍可以参考[官方文档](#)。

另外，数据目录中创建一个文件 `recovery.signal`，内容为空。该文件的作用就是告知 PostgreSQL 服务器需要执行恢复操作。

为了防止恢复期间的用户连接，可以临时修改 `pg_hba.conf` 文件，拒绝远程客户端的连接请

求，等到恢复成功时再修改回来。

最后，重启 PostgreSQL 服务器进程：

```
[postgres@localhost ~]$ pg_ctl start
waiting for server to start....2019-03-28 22:21:26.231 EDT [31505] LOG:
listening on IPv4 address "192.168.56.101", port 5432
2019-03-28 22:21:26.234 EDT [31505] LOG: listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2019-03-28 22:21:26.238 EDT [31505] LOG: listening on Unix socket
"/tmp/.s.PGSQL.5432"
2019-03-28 22:21:26.252 EDT [31505] LOG: redirecting log output to logging
collector process
2019-03-28 22:21:26.252 EDT [31505] HINT: Future log output will appear in
directory "log".
done
server started
```

启动服务器时，会使用恢复模式，并且读取归档的 WAL 文件以及未归档的 WAL 文件。恢复完成后，服务器会删除 `recovery.signal` 文件，并将 `backup_label` 文件重命名为 `backup_label.old`。这样可以防止再次启动时进入恢复模式。最后，数据库将会进入正常操作模式。

我们验证一下数据库中的表 `t1` 是否成功恢复：

```
testdb=# SELECT * FROM t1;
 id | v
----+-----
  1 | pg
(1 rows)
```

如果在恢复之前限制了客户端的连接，现在可以还原 `pg_hba.conf` 文件中的配置。

除此之外，PostgreSQL 还支持时间线（Timelines）恢复，具体内容可以参考[官方文档](#)。

前面提到，基于连续的 WAL 归档技术，如果在另外一台服务器上使用相同的基准备份启动数据库集群，并且不断应用这些归档日志，可以构建一个主从复制的体系结构。这种结构可以提供数据库集群级别的高可用性（HA），PostgreSQL 9.0 引入的流复制支持从节点上的只读操作（read-only），可以承担部分查询的负载。

## 第 7 章 简单查询

我们对于数据库中数据的常见操作，可以简称为**增删改查**（CRUD，Create、Retrieve、Update、Delete）。其中，使用最多，也最复杂的功能当属数据查询。根据 SQL 标准，查询语句使用 **SELECT** 关键字表示。

本书使用的示例表和数据可以关注公众号【SQL 编程思想】获取下载链接。

### 7.1 单表查询

我们先从简单查询开始，来看一个示例。

```
SELECT first_name,  
       last_name  
FROM employees;  
first_name |last_name |  
-----|-----|  
Steven    |King      |  
Neena     |Kochhar   |  
Lex       |De Haan   |  
Alexander |Hunold    |  
Bruce     |Ernst     |  
...
```

基本上不用解释，我们都能知道以上查询语句将会返回员工表（**employees**）中的名字（**first\_name**）和姓氏（**last\_name**）。

**SELECT** 之后是要返回的信息，比如字段名或表达式，多个值使用逗号分隔；**FROM** 表示要查询哪个表；分号表示查询语句结束。

SQL 语句不分区大小写，但是通常将关键字（**SELECT**、**FROM** 等）进行大写，其他内容使用小写，便于阅读。

下面的查询通过一个表达式计算员工的年度薪水：

```
SELECT first_name,  
       last_name,  
       salary * 12 AS annual_income  
FROM employees;  
first_name |last_name |annual_income|  
-----|-----|-----|  
Steven    |King      | 288000.00|  
Neena     |Kochhar   | 204000.00|  
Lex       |De Haan   | 204000.00|  
Alexander |Hunold    | 108000.00|  
Bruce     |Ernst     | 72000.00|  
...
```

其中，**AS** 关键字用于为查询的结果指定一个别名，可以省略。

如果想要查询某个表的全部字段，可以列出所有的字段名称，也可以使用星号（\*）表示：

```

SELECT *
FROM employees;
employee_id|first_name |last_name  |email      |phone_number      |hire_date
|job_id     |salary    |commission_pct|manager_id|department_id|
-----|-----|-----|-----|-----|-----
-|-----|-----|-----|-----|-----|
          100|Steven          |King              |SKING              |515.123.4567
|2003-06-17|AD_PREST      |24000.00|              |          90|
          101|Neena          |Kochhar           |NKOCHHAR|515.123.4568
|2005-09-21|AD_VP         |17000.00|              |100|          90|
          102|Lex            |De Haan           |LDEHAAN  |515.123.4569
|2001-01-13|AD_VP         |17000.00|              |100|          90|
...

```

在实际项目中，应该避免使用 **SELECT \***，因为应用程序可能并不需要全部的字段，而且表结构可能会发生改变，明确指定的字段名称可以减少不确定性。

## 7.2 无表查询

有的时候，我们可能会遇到这样的查询语句：

```

SELECT 2 + 3;
?column?|
-----|
          5|

```

也就是省略了 **FROM** 子句的查询，这是 PostgreSQL 的扩展语法。这种查询通常用于返回系统信息，或者当作计算器使用。需要注意的是，并非所有的关系数据库都支持这种写法，因此它并不具有可移植性。

## 7.3 消除重复结果

现实生活中，存在许多名字相同的人。在执行数据库查询时，也可能会返回重复的值，例如，以下语句查询员工表中的部门编号：

```

SELECT department_id
FROM employees;
department_id
-----
          90
          90
          90
          100
          100
          100
          ...
(107 rows)

```

由于一个部门可以存在多个员工，查询结果中包含了大量重复的数据。如果想要知道员工表



中存在多少个不同的部门编号，需要针对以上结果进行去重操作。SQL 提供了消除查询结果重复值的 **DISTINCT** 关键字。例如：

```
SELECT DISTINCT department_id
FROM employees;
department_id
-----
          70
          80
          20
          10

          90
         100
         110
          30
          50
          40
          60

(12 rows)
```

查询结果只有不重复的 12 条数据。**DISTINCT** 也可以针对多个字段进行去重操作，例如：

```
SELECT DISTINCT first_name,
                last_name
FROM employees;
```

表示查询姓和名的组合不重复的数据。

## 7.4 使用注释

在 PostgreSQL 中，以两个连字符（--）开始，直到这一行结束的内容表示注释：

```
-- 这是标准 SQL 注释方式
SELECT DISTINCT
    first_name
FROM employees;
```

注释的内容会在语法分析之前替换成空格，因此不会被服务器执行。另外，PostgreSQL 还支持 C 语言风格的注释方法（/\* ... \*/）。例如：

```
SELECT DISTINCT
    first_name
    /* 这是一个多行注释，
       DISTINCT 表示排除重复值
       /* 这是一个嵌套的注释 */
    */
FROM employees;
```

PostgreSQL 中的这种注释支持嵌套。

## 第 8 章 查询条件

上一章中我们介绍了如何使用 **SELECT** 和 **FROM** 关键字查询表中的全部数据。如果只想返回满足某些条件的数据，比如某个部门或者某些职位的员工、姓名以特定字符串开头的员工等，需要使用 SQL 中的过滤条件，也就是 **WHERE** 子句。

### 8.1 WHERE 子句

**WHERE** 子句的语法如下：

```
SELECT column1, column2, ...  
  FROM table  
 WHERE conditions;
```

**WHERE** 子句位于 **FROM** 之后，用于指定一个或者多个逻辑条件，用于过滤返回的结果。满足条件的行将会返回，否则将被忽略。**PostgreSQL** 提供了各种运算符和函数，用于构造逻辑条件。

先看一个简单的示例，以下语句返回了薪水为 10000 的员工。

```
SELECT first_name,  
       last_name,  
       salary  
  FROM employees  
 WHERE salary = 10000;  
first_name | last_name | salary  
-----+-----+-----  
Peter      | Tucker   | 10000.00  
Janette    | King      | 10000.00  
Harrison   | Bloom     | 10000.00  
Hermann    | Baer      | 10000.00  
(4 rows)
```

此处的等于号(=)是一个比较运算符，因此只有薪水等于 10000 的数据应用该运算符之后的结果为真值(True)，返回结果中只包含这些数据。

**PostgreSQL** 提供了以下各种比较运算符。

运算符	描述	示例
=	等于	manager_id = 100
!=	不等于	department_id != 50
<>	不等于	job_id <> 'SA_REP'
>	大于	salary > 10000
>=	大于等于	hire_date >= '2007-01-01'
<	小于	salary < 15000

<=	小于等于	employee_id <= 123
BETWEEN	位于范围之内	salary BETWEEN 10000 AND 15000
IN	属于列表之中	job_id IN ('AC_MGR', 'HR_REP', 'IT_PROG')

以上这些运算符的作用都比较明显，不做详细介绍。需要注意的是 **BETWEEN** 包含了两端的值，等价于>=加上<=。

```
SELECT first_name,
       last_name,
       salary
FROM employees
WHERE salary BETWEEN 11000 AND 12000;
first_name | last_name | salary
-----+-----+-----
Den        | Raphaely  | 11000.00
Alberto    | Errazuriz | 12000.00
Gerald     | Cambrault | 11000.00
Lisa       | Ozer      | 11500.00
Ellen      | Abel      | 11000.00
(5 rows)
```

## 8.2 模式匹配

PostgreSQL 支持各种字符串模式匹配的功能。最简单的方式就是使用 **LIKE** 运算符，以下查询返回了姓氏（last\_name）以“Kin”开头的员工。

```
SELECT first_name,
       last_name
FROM employees
WHERE last_name LIKE 'Kin%';
first_name | last_name
-----+-----
Steven     | King
Janette    | King
(2 rows)
```

其中的百分号（%）可以匹配零个或者多个任意字符；另外，下划线（\_）可以匹配一个任意字符。例如：

- “%en”匹配以“en”结束的字符串；
- “%en%”匹配包含“en”的字符串；
- “B\_g”匹配“Big”、“Bug”等。

如果字符串中存在这两个通配符（%或\_），可以在它们前面加上一个反斜杠（\）进行转义。

```
SELECT 1 WHERE 'this is 25%' LIKE '%25\%';
?column?
-----
1
```

```
(1 row)
```

也可以通过 **ESCAPE** 子句指定其他的转义字符。

```
SELECT 1 WHERE 'this is 25%' LIKE '%25@%' ESCAPE '@';
?column?
-----
      1
(1 row)
```

另外，**NOT LIKE** 运算符匹配与 **LIKE** 相反的结果。

```
SELECT first_name,
       last_name
FROM employees
WHERE last_name NOT LIKE 'Kin%';
first_name | last_name
-----+-----
Neena      | Kochhar
Lex        | De Haan
Nancy      | Greenberg
Daniel     | Faviet
John       | Chen
.....
(105 rows)
```

**LIKE** 运算符区分大小写，PostgreSQL 同时还提供了不区分大小写的 **ILIKE** 运算符。

```
SELECT first_name,
       last_name
FROM employees
WHERE last_name LIKE 'kin%';
first_name | last_name
-----+-----
(0 rows)

SELECT first_name,
       last_name
FROM employees
WHERE last_name ILIKE 'kin%';
first_name | last_name
-----+-----
Steven     | King
Janette    | King
(2 rows)
```

PostgreSQL 还提供了更加复杂的正则表达式匹配等功能，详细内容可以参考[官方文档](#)。

## 8.3 空值判断

根据 SQL 标准，空值使用 **NULL** 表示。空值是一个特殊值，代表了未知数据。如果使用常规的比较运算符与 **NULL** 进行比较，总是返回空值。

```
NULL = 0; -- 结果为空值
```

```
NULL = NULL; -- 结果为空值
NULL != NULL; -- 结果为空值
```

如果在查询条件中使用这种方式，将不会返回任何结果。因此，对于 **NULL** 值的比较，需要使用特殊的运算符：**IS NULL**。

```
SELECT first_name,
       last_name,
       department_id
FROM employees
WHERE department_id IS NULL;
first_name | last_name | department_id
-----+-----+-----
Kimberely  | Grant    | 
(1 row)
```

以上查询返回了部门编号为空的员工，意味着该员工还没有被分配到任何部门。

**IS NOT NULL** 执行与 **IS NULL** 相反的操作，返回值不为空的数据。

## 8.4 复杂条件

**WHERE** 子句可以包含多个条件，使用逻辑运算符（**AND**、**OR**、**NOT**）将它们进行组合，并根据最终的逻辑值进行过滤。

**AND**（逻辑与）运算符的逻辑真值表如下：

	<b>TRUE</b>	<b>FALSE</b>	<b>NULL</b>
<b>TRUE</b>	TRUE	FALSE	NULL
<b>FALSE</b>	FALSE	FALSE	FALSE
<b>NULL</b>	NULL	FALSE	NULL

对于 **AND** 运算符，只有当它两边的结果都为真时，最终结果才为真；否则最终结果为假，不返回结果。

以下查询返回薪水为 10000，并且姓氏为“King”的员工：

```
SELECT first_name,
       last_name,
       salary
FROM employees
WHERE salary = 10000 AND last_name = 'King';
first_name | last_name | salary
-----+-----+-----
Janette    | King      | 10000.00
(1 row)
```

**OR**（逻辑或）运算符的逻辑真值表如下：

	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

OR 逻辑或运算符只要有一个条件为真，结果就为真。以下查询返回薪水为 10000，或者姓氏为“King”的员工：

```
SELECT first_name,
       last_name,
       salary
FROM employees
WHERE salary = 10000 OR last_name = 'King';
first_name | last_name | salary
-----+-----+-----
Steven     | King      | 24000.00
Peter      | Tucker   | 10000.00
Janette    | King      | 10000.00
Harrison   | Bloom     | 10000.00
Hermann    | Baer      | 10000.00
(5 rows)
```

对于逻辑运算符 AND 和 OR，需要注意的是，它们使用短路运算。也就是说，只要前面的表达式能够决定最终的结果，不进行后面的计算。这样能够提高运算效率。因此，以下语句不会产生除零错误：

```
SELECT 1 WHERE 1 = 0 AND 1/0 = 1;
SELECT 1 WHERE 1 = 1 OR 1/0 = 1;
```

还需要注意的一个问题是，当我们组合 AND 和 OR 运算符时，AND 运算符优先级更高，总是先执行。

```
SELECT first_name,
       last_name,
       salary
FROM employees
WHERE salary = 10000 OR salary = 24000 AND last_name = 'King';
first_name | last_name | salary
-----+-----+-----
Steven     | King      | 24000.00
Peter      | Tucker   | 10000.00
Janette    | King      | 10000.00
Harrison   | Bloom     | 10000.00
Hermann    | Baer      | 10000.00
(5 rows)
```

由于 AND 优先级高，查询返回的是薪水为 24000 并且姓氏为“King”的员工，或者薪水为 10000 的员工。如果相要返回姓氏为“King”，并且薪水为 10000 或 24000 的员工，可以使用括号修改优先级：

```
SELECT first_name,
       last_name,
       salary
```

```
FROM employees
WHERE (salary = 10000 OR salary = 24000)
AND last_name = 'King';
first_name | last_name | salary
-----+-----+-----
Steven      | King      | 24000.00
Janette     | King      | 10000.00
(2 rows)
```

NOT（逻辑非）运算符用于取反操作，它的逻辑真值表如下：

	NOT
TRUE	FALSE
FALSE	TRUE
NULL	NULL

注意，对于未知的 NULL 值，经过 NOT 处理之后仍然是未知值。

除此之外，NOT 还可以结合前面介绍的运算符一起使用：

- NOT BETWEEN，位于范围之外
- NOT IN，不在列表之中
- NOT LIKE，不匹配模式
- NOT IS NULL，不为空，等价于 IS NOT NULL

最后，当查询条件包含复杂逻辑时，它们的运算优先级从高到低排列如下：

条件运算符	描述
=, !=, <>, <, <=, >, >=	比较运算
IS [NOT] NULL, [NOT] LIKE, [NOT] BETWEEN, [NOT] IN, [NOT] EXISTS	比较运算
NOT	逻辑否定
AND	逻辑与

使用括号可以调整多个运算符之间的优先级。

## 第 9 章 排序显示

当我们使用 `SELECT` 语句查询表中的数据时, PostgreSQL 不确保按照一定的顺序返回结果。如果相要将查询结果按照某些规则进行排序显示, 例如按照薪水从高到低, 或者按照入职时间的先后进行排序, 需要使用 `ORDER BY` 子句。

### 9.1 单列排序

单列排序是指按照某个字段或者表达式进行排序, 用法如下:

```
SELECT column1, column2, ...
   FROM table
  ORDER BY column1 [ASC | DESC];
```

`ORDER BY` 表示按照某个字段进行排序, `ASC` 表示升序排序 (Ascending), `DESC` 表示降序排序 (Descending), 默认值为 `ASC`。

以下查询返回部门编号为 60 的员工, 并且按照薪水从高到低进行排序显示:

```
SELECT first_name,
       last_name,
       salary
   FROM employees
  WHERE department_id = 60
 ORDER BY salary DESC;
 first_name | last_name | salary
-----+-----+-----
 Alexander | Hunold    | 9000.00
 Bruce     | Ernst     | 6000.00
 David     | Austin    | 4800.00
 Valli     | Pataballa | 4800.00
 Diana     | Lorentz   | 4200.00
(5 rows)
```

从结果可以看到, 对于第 3 行和第 4 行, 它们的薪水值都是 4800。那么这两行应该如何排序显示呢? 答案是不确定。如果想要解决这个问题, 需要使用多列排序。

### 9.2 多列排序

对于单列排序, 有可能存在多个数据值相同的情况。此时, 可以再指定其他的排序字段进行处理。

```
SELECT column1, column2, ...
   FROM table
  ORDER BY column1 ASC, column2 DESC, ...;
```

首先基于第一个排序字段进行排序, 对于可能存在的相同值, 再基于第二个字段进行排序, 依此类推。



以下查询返回部门编号为 60 的员工,并且按照薪水从高到低进行排序显示,如果薪水相同,再按照名字 (first\_name) 降序排列:

```
SELECT first_name,
       last_name,
       salary
FROM employees
WHERE department_id = 60
ORDER BY salary DESC, first_name DESC;
first_name | last_name | salary
-----+-----+-----
Alexander  | Hunold    | 9000.00
Bruce       | Ernst     | 6000.00
Valli       | Pataballa | 4800.00
David       | Austin    | 4800.00
Diana       | Lorentz   | 4200.00
(5 rows)
```

此时,“Valli Pataballa”排在了“David Austin”的前面。

ORDER BY 后的排序字段可以是 SELECT 列表中没有的字段。以下语句返回了员工的姓名和薪水,按照入职先后进行显示:

```
SELECT first_name,
       last_name,
       salary
FROM employees
ORDER BY hire_date;
```

除了在 ORDER BY 后指定字段名或者表达式之外,也可以简单的使用它们在 SELECT 列表中出现的顺序来表示:

```
SELECT first_name,
       last_name,
       salary
FROM employees
ORDER BY 1, 3;
```

以上语句表示先按照第 1 个字段 (first\_name) 进行排序,再按照第 3 个字段 (salary) 进行排序。

另外,PostgreSQL 对于字符类型的数据进行排序时不区分大小写,“CAT”和“cat”顺序相同。

```
CREATE TABLE tbl_char(c1 varchar(10));
INSERT INTO tbl_char VALUES('CAT'), ('cat'), ('dog');

SELECT *
FROM tbl_char
ORDER BY c1;
c1
-----
cat
CAT
dog
(3 rows)
```

## 9.3 空值排序

在 SQL 中，空值是一个特殊的值，使用 NULL 表示。如果排序的字段中存在空值时，应该如何处理呢？先看一个示例：

```
SELECT first_name,
       last_name,
       commission_pct
FROM employees
WHERE first_name = 'Peter'
ORDER BY commission_pct;
```

first_name	last_name	commission_pct
Peter	Hall	0.25
Peter	Tucker	0.30
Peter	Vargas	

(3 rows)

以上查询按照佣金百分比（`commission_pct`）进行升序显示。对于“Peter Vargas”，由于他没有佣金提成，相应的值为空，PostgreSQL 默认将他排在了最后。

PostgreSQL 支持使用 `NULLS FIRST`（空值排在最前）和 `NULLS LAST`（空值排在最后）指定空值的排序位置；升序排序时默认为 `NULLS LAST`，降序排序时默认为 `NULLS FIRST`。

我们修改上面的示例，将“Peter Vargas”排在最前，但仍然按照佣金百分比进行升序显示：

```
SELECT first_name,
       last_name,
       commission_pct
FROM employees
WHERE first_name = 'Peter'
ORDER BY commission_pct NULLS FIRST;
```

first_name	last_name	commission_pct
Peter	Vargas	
Peter	Hall	0.25
Peter	Tucker	0.30

(3 rows)

## 第 10 章 限定结果数量

查询语句的结果可能包含成百上千行数据，但是前端显示时也许只需要其中的小部分，例如 TOP-N 排行榜；或者为了便于查看，每次只显示一定数量的结果，例如分页功能。为了处理这类应用，SQL 提供了标准的 **FETCH** 和 **OFFSET** 子句。另外，PostgreSQL 还实现了扩展的 **LIMIT** 语法。

### 10.1 Top-N 查询

这类查询通常是为了找出排名中的前 N 个记录，例如以下语句查询薪水最高的前 10 名员工，使用 **FETCH** 语法：

```
SELECT first_name, last_name, salary
FROM employees
ORDER BY salary DESC
FETCH FIRST 10 ROWS ONLY;
```

first_name	last_name	salary
Steven	King	24000.00
Lex	De Haan	17000.00
Neena	Kochhar	17000.00
John	Russell	14000.00
Karen	Partners	13500.00
Michael	Hartstein	13000.00
Shelley	Higgins	12008.00
Nancy	Greenberg	12008.00
Alberto	Errazuriz	12000.00
Lisa	Ozer	11500.00

其中，**FIRST** 也可以写成 **NEXT**，**ROWS** 也可以写成 **ROW**。结果返回了排序之后的前 10 条记录。

使用 **LIMIT** 语法也可以实现相同的功能：

```
SELECT first_name, last_name, salary
FROM employees
ORDER BY salary DESC
LIMIT 10;
```

### 10.2 分页查询

许多应用都支持分页显示的功能，即每页显示一定数量的记录（例如 10 行、20 行等），同时提供类似上一页和下一页的导航。使用 SQL 实现这种功能需要引入另一个子句：**OFFSET**。

假设我们的应用提供了分页显示，每页显示 10 条记录。现在用户点击了下一页，需要显示第 11 到第 20 条记录。使用标准 SQL 语法实现如下：

```

SELECT first_name, last_name, salary
  FROM employees
 ORDER BY salary DESC
OFFSET 10 ROWS
  FETCH FIRST 10 ROWS ONLY;
first_name|last_name|salary |
-----|-----|-----|
Gerald    |Cambrault|11000.00|
Den       |Raphaely |11000.00|
Ellen     |Abel     |11000.00|
Clara     |Vishney  |10500.00|
Eleni     |Zlotkey  |10500.00|
Harrison  |Bloom    |10000.00|
Peter     |Tucker   |10000.00|
Janette   |King     |10000.00|
Hermann   |Baer     |10000.00|
Tayler    |Fox      | 9600.00|

```

**OFFSET** 表示先忽略掉多少行数据，然后再返回后面的结果。**ROWS** 也可以写成 **ROW**。对于应用程序而言，只需要传入不同的 **OFFSET** 偏移量和 **FETCH** 数量，就可以在结果中任意导航。

使用 **LIMIT** 加上 **OFFSET** 同样可以实现分页效果：

```

SELECT first_name, last_name, salary
  FROM employees
 ORDER BY salary DESC
LIMIT 10 OFFSET 10;

```

## 10.3 注意事项

我们先看一下完整的 **FETCH** 和 **LIMIT** 语法：

```

SELECT column1, column2, ...
  FROM table
 [WHERE conditions]
 [ORDER BY column1 ASC, column2 DESC, ...]
 [OFFSET m {ROW | ROWS}]
 [FETCH { FIRST | NEXT } [ num_rows ] { ROW | ROWS } ONLY];

SELECT column1, column2, ...
  FROM table
 [WHERE conditions]
 [ORDER BY column1 ASC, column2 DESC, ...]
 [LIMIT { num_rows | ALL } ]
 [OFFSET m {ROW | ROWS}];

```

在使用以上功能时需要注意以下问题：

- **FETCH** 是标准 SQL 语法，**LIMIT** 是 PostgreSQL 扩展语法。
- 如果没有指定 **ORDER BY**，限定数量之前并没有进行排序，是一个随意的结果。
- **OFFSET** 偏移量必须为 0 或者正整数。默认为 0，NULL 等价于 0。
- **FETCH** 限定的数量必须为 0 或者正整数。默认为 1，NULL 等价于不限定数量。

- LIMIT 限定的数量必须为 0 或者正整数, 没有默认值。ALL 或者 NULL 表示不限定数量。
- 随着 OFFSET 的增加, 查询的性能会越来越差。因为服务器需要计算更多的偏移量, 即使这些数据不需要被返回前端。

## 第 11 章 分组汇总

### 11.1 聚合函数

聚合函数（aggregate function）针对一组数据行进行运算，并且返回单个结果。PostgreSQL 支持以下常见的聚合函数：

- **AVG** - 计算一组值的平均值。
- **COUNT** - 统计一组值的数量。
- **MAX** - 计算一组值的最大值。
- **MIN** - 计算一组值的最小值。
- **SUM** - 计算一组值的和值。
- **STRING\_AGG** - 连接一组字符串。

以下示例分别返回了 IT 部门所有员工的平均薪水、员工总数、最高薪水、最低薪水、以及薪水总计：

```
SELECT AVG(salary),
       COUNT(*),
       MAX(salary),
       MIN(salary),
       SUM(salary)
FROM employees
WHERE department_id = 60;
```

avg	count	max	min	sum
5760.0000000000000000	5	9000.00	4200.00	28800.00

关于聚合函数，需要注意两点：

- 函数参数前添加 **DISTINCT** 关键字，可以在计算时排除重复值。
- 忽略参数中的 **NULL** 值。

来看以下查询：

```
SELECT COUNT(*),
       COUNT(DISTINCT salary),
       COUNT(commission_pct)
FROM employees
WHERE department_id = 60;
```

count	count	count
5	4	0

其中，**COUNT(\*)**返回了该部门员工的总数（5），**COUNT(DISTINCT salary)**返回了薪水不相同的员工数量（4），**COUNT(commission\_pct)**返回了佣金百分比不为空值的数量（0），该部门员工都没有佣金提成。

另外, PostgreSQL 为聚合函数提供了一个 **FILTER** 扩展选项, 可以用于汇总满足特定条件的数据。例如:

```
SELECT COUNT(*) FILTER (WHERE salary >= 10000) high_sal,
       COUNT(*) FILTER (WHERE salary < 10000) low_sal
FROM employees;

high_sal|low_sal|
-----|-----|
      19|      88|
```

其中, **FILTER** 选项可以指定一个 **WHERE** 条件, 只有满足条件的数据才会进行汇总。因此, 示例中的第一个 **COUNT** 函数返回了月薪大于等于 10000 的员工数量, 第二个 **COUNT** 函数返回了月薪小于 10000 的员工数量。

以下示例使用 **STRING\_AGG** 函数将 IT 部门员工的名字使用分号进行分隔, 按照薪水从高到低排序后连接成一个字符串:

```
SELECT STRING_AGG(first_name, ';' ORDER BY salary DESC)
FROM employees
WHERE department_id = 60;

string_agg
-----|
Alexander;Bruce;David;Valli;Diana|
```

更多的聚合函数可以参考[官方文档](#)。

## 11.2 分组聚合

我们已经获得了 IT 部门的一些汇总信息, 如果还需要知道其他部门的相关信息, 可以多次运行相同的查询 (修改查询条件中的部门编号)。但是这种明显过于复杂, 不适合实际应用。SQL 为此提供了 **GROUP BY** 子句, 它用于将数据分成多个组, 然后使用聚合函数对每个组进行汇总。

举例来说, 如果我们想要知道每个部门内所有员工的平均薪水、员工总数、最高薪水、最低薪水、以及薪水总计, 可以使用以下查询语句:

```
SELECT department_id,
       AVG(salary),
       COUNT(*),
       MAX(salary),
       MIN(salary),
       SUM(salary)
FROM employees
GROUP BY department_id
ORDER BY department_id;

department_id|avg
-----|-----|count|max
-----|-----|-----|-----|min
-----|-----|-----|-----|sum
-----|-----|-----|-----|-----|
      10| 4400.0000000000000000|1| 4400.00| 4400.00| 4400.00|
```

20	9500.0000000000000000	2	13000.00	6000.00	19000.00
30	4150.0000000000000000	6	11000.00	2500.00	24900.00
40	6500.0000000000000000	1	6500.00	6500.00	6500.00
50	3475.5555555555555556	45	8200.00	2100.00	156400.00
60	5760.0000000000000000	5	9000.00	4200.00	28800.00
70	10000.0000000000000000	1	10000.00	10000.00	10000.00
80	8955.8823529411764706	34	14000.00	6100.00	304500.00
90	19333.333333333333333	3	24000.00	17000.00	58000.00
100	8601.3333333333333333	6	12008.00	6900.00	51608.00
110	10154.0000000000000000	2	12008.00	8300.00	20308.00
	7000.0000000000000000	1	7000.00	7000.00	7000.00

查询执行时，首先根据 **GROUP BY** 子句中的列（**department\_id**）进行分组，然后使用聚合函数汇总组内的数据。最后一条数据是针对部门编号字段为空的数据进行的分组汇总，**GROUP BY** 将所有的 **NULL** 分为一组。

**GROUP BY** 并不一定需要与聚合函数一起使用，例如：

```
SELECT department_id
FROM employees
GROUP BY department_id
ORDER BY department_id;
```

```
department_id|
-----|
10|
20|
30|
40|
50|
60|
70|
80|
90|
100|
110|
|
```

查询的结果就是不同的部门编号分组，这种查询的结果与 **DISTINCT** 效果相同：

```
SELECT DISTINCT department_id
FROM employees
ORDER BY department_id;
```

**GROUP BY** 不仅可以按照一个字段进行分组，也可以使用多个字段将数据分成更多的组。例如，以下查询将员工按照不同的部门和职位组合进行分组，然后进行汇总：

```
SELECT department_id,
       job_id,
       AVG(salary),
       COUNT(*),
       MAX(salary),
       MIN(salary),
       SUM(salary)
FROM employees
GROUP BY department_id, job_id
ORDER BY department_id, job_id;
```



department_id	job_id	avg	count	max	min	sum
10	AD_ASST	4400.000000	1	4400.00	4400.00	4400.00
20	MK_MAN	13000.000000	1	13000.00	13000.00	13000.00
20	MK_REP	6000.000000	1	6000.00	6000.00	6000.00
30	PU_CLERK	2780.000000	5	3100.00	2500.00	13900.00
30	PU_MAN	11000.000000	1	11000.00	11000.00	11000.00
40	HR_REP	6500.000000	1	6500.00	6500.00	6500.00
50	SH_CLERK	3215.000000	20	4200.00	2500.00	64300.00
50	ST_CLERK	2785.000000	20	3600.00	2100.00	55700.00
50	ST_MAN	7280.000000	5	8200.00	5800.00	36400.00
60	IT_PROG	5760.000000	5	9000.00	4200.00	28800.00
...						

使用了 **GROUP BY** 子句进行分组操作之后需要注意一点，就是 **SELECT** 列表中只能出现分组字段或者聚合函数，不能再出现表中的其他字段。下面是一个错误的示例：

```
SELECT department_id,
       job_id,
       AVG(salary),
       COUNT(*),
       MAX(salary),
       MIN(salary),
       SUM(salary)
FROM employees
GROUP BY department_id;
```

SQL 错误 [42803]: ERROR: column "employees.job\_id" must appear in the GROUP BY clause or be used in an aggregate function  
Position: 31

错误的原因在于 **job\_id** 既不是分组的条件，也不是聚合函数。查询要求按照部门进行分组汇总，但是每个部门存在多个不同的职位，数据库无法知道需要显示哪个职位编号。

## 11.3 分组过滤

当我们需要针对分组汇总后的数据再次进行过滤时，例如找出平均薪水值大于 10000 的部门，直观的想法就是在 **WHERE** 子句中增加一个过滤条件，例如：

```
SELECT department_id,
       AVG(salary),
       COUNT(*),
       MAX(salary),
       MIN(salary),
       SUM(salary)
FROM employees
WHERE AVG(salary) > 10000
GROUP BY department_id
ORDER BY department_id;
```

SQL 错误 [42803]: ERROR: aggregate functions are not allowed in WHERE

不过查询并没有返回期望的结果，而是出现了一个错误：**WHERE** 子句中不允许出现聚合函数。因为在 **SQL** 询中，如果同时存在 **WHERE** 子句和 **GROUP BY** 子句，**WHERE** 子句在 **GROUP BY** 子句之前执行。因此，**WHERE** 子句无法对分组后的结果进行过滤。

**WHERE** 子句执行时还没有进行分组计算，它只能基于分组之前的数据进行过滤。如果需要对分组后的结果进行过滤，需要使用 **HAVING** 子句。以上查询的正确写法如下：

```
SELECT department_id,
       AVG(salary),
       COUNT(*),
       MAX(salary),
       MIN(salary),
       SUM(salary)
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 10000
ORDER BY department_id;
```

department_id	avg	count	max	min	sum
90	19333.333333333333	3	24000.00	17000.00	58000.00
110	10154.000000000000	2	12008.00	8300.00	20308.00

**HAVING** 出现在 **GROUP BY** 之后，也在它之后执行，因此能够使用聚合函数进行过滤。

我们可以同时使用 **WHERE** 子句进行数据行的过滤，使用 **HAVING** 进行分组结果的过滤。以下示例用于查找哪些部门中薪水大于 10000 的员工数量多于 2 个：

```
SELECT department_id,
       COUNT(*) AS headcount
FROM employees
WHERE salary > 10000
GROUP BY department_id
HAVING COUNT(*) > 2;
```

department_id	headcount
80	8
90	3

查询时，首先通过 **WHERE** 子句找出薪水大于 10000 的所有员工；然后，按照部门编号进行分组，计算每个组内的员工数量；最后使用 **HAVING** 子句过滤员工数量多于 2 个人的部门。

## 11.4 高级选项

PostgreSQL 除了支持基本的 **GROUP BY** 分组操作之外，还支持 3 种高级的分组选项：**GROUPING SETS**、**ROLLUP** 以及 **CUBE**。

### 11.4.1 GROUPING SETS 选项

GROUPING SETS 是 GROUP BY 的扩展选项，用于指定自定义的分组集。举例来说，以下是一个销售数据表：

```
CREATE TABLE sales (  
    item VARCHAR(10),  
    year VARCHAR(4),  
    quantity INT  
);  
  
INSERT INTO sales VALUES('apple', '2018', 800);  
INSERT INTO sales VALUES('apple', '2018', 1000);  
INSERT INTO sales VALUES('banana', '2018', 500);  
INSERT INTO sales VALUES('banana', '2018', 600);  
INSERT INTO sales VALUES('apple', '2019', 1200);  
INSERT INTO sales VALUES('banana', '2019', 1800);
```

按照产品（item）和年度（year）进行分组汇总时，所有可能的 4 种分组集包括：

- 按照产品和年度的组合进行分组；
- 按照产品进行分组；
- 按照年度进行分组；
- 所有数据分为一组。

可以通过以下多个查询获取所有分组集的分组结果：

```
-- 按照产品和年度的组合进行分组  
SELECT item, year, SUM(quantity)  
FROM sales  
GROUP BY item, year;  
  
item |year|sum |  
-----|----|----|  
banana|2019|1800|  
apple |2019|1200|  
apple |2018|1800|  
banana|2018|1100|  
  
-- 按照产品进行分组  
SELECT item, NULL AS year, SUM(quantity)  
FROM sales  
GROUP BY item;  
  
item |year|sum |  
-----|----|----|  
banana|    |2900|  
apple |    |3000|  
  
-- 按照年度进行分组  
SELECT NULL AS item, year, SUM(quantity)  
FROM sales  
GROUP BY year;  
  
item|year|sum |  
----|----|----|
```

```

    |2018|2900|
    |2019|3000|

-- 所有数据分为一组
SELECT NULL AS item, NULL AS year, SUM(quantity)
   FROM sales;

item|year|sum |
----|----|----|
    |    |5900|

```

在后续的文章中我们会介绍如何使用集合运算符（**UNION ALL**）将 4 个查询结果合并到一起。但是这种方法存在一些问题：首先，查询语句比较冗长，查询的次数随着分组字段的增加呈指数增长；其次，多次查询意味着需要多次扫描同一张表，存在性能上的问题。

**GROUPING SETS** 是 **GROUP BY** 的扩展选项，能够为这种查询需求提供更加简单有效的解决方法。我们使用分组集改写上面的示例：

```

SELECT item, year, SUM(quantity)
   FROM sales
  GROUP BY GROUPING SETS (
    (item, year),
    (item),
    (year),
    ()
  );

item |year|sum |
-----|----|----|
    |    |5900|
banana|2019|1800|
apple |2019|1200|
apple |2018|1800|
banana|2018|1100|
banana|    |2900|
apple |    |3000|
    |2018|2900|
    |2019|3000|

```

**GROUPING SETS** 选项用于定义分组集，每个分组集都需要包含在单独的括号中，空白的括号（**()**）表示将所有数据当作一个组处理。查询的结果等于前文 4 个查询的合并结果，但是语句更少，可读性更强；而且 PostgreSQL 执行时只需要扫描一次销售表，性能更加优化。

另外，默认的 **GROUP BY** 使用由所有分组字段构成的一个分组集，本示例中为 **((item, year))**。

## 11.4.2 CUBE 选项

随着分组字段的增加，即使通过 **GROUPING SETS** 列出所有可能的分组方式也会显得比较麻烦。设想一下使用 4 个字段进行分组统计的场景，所有可能的分组集共计有 16 个。这种情况下编写查询语句仍然很复杂，为此 PostgreSQL 提供了简写形式的 **GROUPING SETS**：**CUBE** 和 **ROLLUP**。

**CUBE** 表示所有可能的分组集，例如：

```
CUBE ( c1, c2, c3 )
```

等价于：

```
GROUPING SETS (
  ( c1, c2, c3 ),
  ( c1, c2 ),
  ( c1, c3 ),
  ( c2, c3 ),
  ( c1 ),
  ( c2 ),
  ( c3 ),
  ( )
)
```

因此，我们可以进一步将上面的示例改写如下：

```
SELECT item, year, SUM(quantity)
FROM sales
GROUP BY CUBE (item,year);
```

```
item |year|sum |
-----|----|----|
      |    |5900|
banana|2019|1800|
apple |2019|1200|
apple |2018|1800|
banana|2018|1100|
banana|    |2900|
apple |    |3000|
      |2018|2900|
      |2019|3000|
```

### 11.4.3 ROLLUP 选项

**GROUPING SETS** 第二种简写形式就是 **ROLLUP**，用于生成按照层级进行汇总的结果，类似于财务报表中的小计、合计和总计。例如：

```
ROLLUP ( c1, c2, c3 )
```

等价于：

```
GROUPING SETS (
  ( c1, c2, c3 ),
  ( e1, e2 ),
  ( e1 ),
  ( )
)
```

以下查询返回按照产品和年度组合进行统计的销量小计，加上按照产品进行统计的销量合计，再加上所有销量的总计：

```
SELECT item, year, SUM(quantity)
FROM sales
GROUP BY ROLLUP (item,year);
```

```
item |year|sum |
```

```

-----|----|----|
      |      |5900|
banana|2019|1800|
apple |2019|1200|
apple |2018|1800|
banana|2018|1100|
banana|      |2900|
apple |      |3000|

```

查看结果时，需要根据每个字段上的空值进行判断。比如第一行的产品和年度都为空，因此它是所有销量的总计。为了便于查看，可以将空值进行转换显示：

```

SELECT coalesce(item, '所有产品') AS "产品",
       coalesce(year, '所有年度') AS "年度",
       SUM(quantity) AS "销量"
FROM sales
GROUP BY ROLLUP (item, year);

```

```

产品    |年度    |销量    |
-----|-----|----|
所有产品|所有年度|5900|
banana |2019   |1800|
apple  |2019   |1200|
apple  |2018   |1800|
banana |2018   |1100|
banana |所有年度|2900|
apple  |所有年度|3000|

```

**COALESCE** 函数返回第一个非空的参数值。

可以根据需要返回按照某些组合进行统计的结果，以下查询返回按照产品和年度组合进行统计的销量小计，加上按照产品进行统计的销量合计：

```

SELECT coalesce(item, '所有产品') AS "产品",
       coalesce(year, '所有年度') AS "年度",
       SUM(quantity) AS "销量"
FROM sales
GROUP BY item, ROLLUP (year);

```

```

产品    |年度    |销量    |
-----|-----|----|
banana |2019   |1800|
apple  |2019   |1200|
apple  |2018   |1800|
banana |2018   |1100|
banana |所有年度|2900|
apple  |所有年度|3000|

```

对于 **CUBE** 和 **ROLLUP** 而言，每个元素可以是单独的字段或表达式，也可以是使用括号包含的列表。如果是括号中的列表，产生分组集时它们必须作为一个整体。例如：

```

CUBE ( (c1, c2), (c3, c4) )

```

等价于：

```

GROUPING SETS (

```

```

    ( c1, c2, c3, c4 ),
    ( c1, c2 ),
    ( c3, c4 ),
    ( )
)

```

因为 c1 和 c2 是一个整体, c3 和 c4 是一个整体。

同样:

```
ROLLUP ( c1, (c2, c3), c4 )
```

等价于:

```

GROUPING SETS (
    ( c1, c2, c3, c4 ),
    ( c1, c2, c3 ),
    ( c1 ),
    ( )
)

```

#### 11.4.4 GROUPING 函数

虽然有时候可以通过空值来判断数据是不是某个字段上的汇总,比如说按照年度进行统计的结果在字段 year 上的值为空。但是情况并非总是如此,考虑以下示例:

```

-- 未知产品在 2018 年的销量为 5000
INSERT INTO sales VALUES(NULL, '2018', 5000);

SELECT coalesce(item, '所有产品') AS "产品",
       coalesce(year, '所有年度') AS "年度",
       SUM(quantity) AS "销量"
FROM sales
GROUP BY ROLLUP (item,year);

```

查询结果如下:

	ABC 产品	ABC 年度	123 销量
1	所有产品	所有年度	10,900
2	banana	2019	1,800
3	apple	2019	1,200
4	apple	2018	1,800
5	所有产品	2018	5,000
6	banana	2018	1,100
7	所有产品	所有年度	5,000
8	banana	所有年度	2,900
9	apple	所有年度	3,000

其中第 5 行和第 7 行的显示存在问题,它们分别应该是未知产品在 2018 年的销量小计和所有年度的销量合计。问题的关键在于无法区分是分组产生的 NULL 还是源数据中的 NULL。为了解决这个问题,PostgreSQL 提供了一个分组函数: GROUPING。

以下查询显示了 GROUPING 函数的结果:

```
SELECT item AS "产品",
       year AS "年度",
       SUM(quantity) AS "销量",
       GROUPING(item),
       GROUPING(year),
       GROUPING(item, year)
FROM sales
GROUP BY ROLLUP (item, year);
```

返回的结果如下：

产品	年度	销量	grouping	grouping	grouping
		5900	1	1	3
banana	2019	1800	0	0	0
apple	2019	1200	0	0	0
apple	2018	1800	0	0	0
banana	2018	1100	0	0	0
banana		2900	0	1	1
apple		3000	0	1	1

**GROUPING** 函数如果只有一个参数，返回整数 0 或者 1。如果某个统计结果使用的分组集包含了函数中的参数字段，该函数返回 0，否则返回 1。比如说，第 1 行数据是所有产品所有年度的统计（分组集为空），所以 **GROUPING(item)**和 **GROUPING(year)**结果都是 1；第 7 行数据是未知产品所有年度的统计(分组集为(item, )),所以 **GROUPING(item)**结果为 0, **GROUPING(year)**结果为 1。

**GROUPING** 函数如果包含多个参数，针对每个参数返回整数 0 或者 1，然后将它们按照二进制数值连接到一起。比如说，第 1 行数据中的 **GROUPING(item, year)**结果等于 **GROUPING(item)**和 **GROUPING(year)**结果的二进制数值连接，也就是 3（二进制的 11）。

通过使用 **GROUPING** 函数，我们可以正确显示分组中的 NULL 值和源数据中的 NULL 值：

```
SELECT CASE GROUPING(item) WHEN 1 THEN '所有产品' ELSE item END AS "产品",
       CASE GROUPING(year) WHEN 1 THEN '所有年度' ELSE year END AS "年度",
       SUM(quantity) AS "销量"
FROM sales
GROUP BY ROLLUP (item, year);
```

该查询的结果如下：

产品	年度	销量
所有产品	所有年度	5900
banana	2019	1800
apple	2019	1200
apple	2018	1800
banana	2018	1100
banana	所有年度	2900
apple	所有年度	3000



## 第 12 章 多表连接

在关系型数据库中，通常将不同的实体和它们之间的联系存储到多个表中。比如员工的个人信息存储在 `employees` 表中，而与部门相关的信息存储在 `departments` 表中，同时 `employees` 表中存在一个外键字段（`department_id`），引用了 `departments` 表的主键（`department_id`）。

当我们想要查看员工的信息时，通常只需要查询员工表；但是如果想要同时查看员工的个人信息以及他/她所在的部门信息，就需要同时查询 `employees` 和 `departments` 表中的信息。此时，我们需要使用连接查询。连接查询（JOIN）基于两个表中的连接字段将数据行拼接到一起，可以同时返回两个表中的相关数据。

PostgreSQL 支持各种类型的 SQL 连接查询：

- 内连接（INNER JOIN）
- 左外连接（LEFT OUTER JOIN）
- 右外连接（RIGHT OUTER JOIN）
- 全外连接（FULL OUTER JOIN）
- 交叉连接（CROSS JOIN）
- 自然连接（NATURAL JOIN）
- 自连接（Self Join）

其中，左外连接、右外连接以及全外连接统称为外连接（OUTER JOIN）。

### 12.1 内连接

内连接用于返回两个表中匹配的数据行，使用关键字 `INNER JOIN` 表示，也可以简写成 `JOIN`；以下是内连接的示意图（基于两个表的 `id` 进行连接）：



其中，`id = 1` 和 `id = 3` 是两个表中匹配（`table1.id = table2.id`）的数据，因此内连接返回了 2 行记录。以下是一个内连接查询的示例：

```
SELECT d.department_id,  
       e.department_id,  
       d.department_name,  
       e.first_name,  
       e.last_name  
FROM employees e  
JOIN departments d  
  ON e.department_id = d.department_id;
```

其中，JOIN 表示内连接，ON 表示连接条件。另外，SELECT 列表中的字段名加上了表名限定，例如 d.department\_id，这是因为两个表中都存在部门编号，必须明确指定需要显示哪个表中的字段。不过，如果某个字段只存在于一个表中，可以省略表名，例如 first\_name。

该查询的结果如下（显示部分内容）：

123 department_id	123 department_id	ABC department_name	ABC first_name	ABC last_name
90	90	Executive	Steven	King
90	90	Executive	Neena	Kochhar
90	90	Executive	Lex	De Haan
100	100	Finance	Nancy	Greenberg
100	100	Finance	Daniel	Faviet
100	100	Finance	John	Chen
100	100	Finance	Ismael	Sciarra
100	100	Finance	Jose Manuel	Urman
100	100	Finance	Luis	Popp
30	30	Purchasing	Den	Raphaely

建议在多表查询中，总是加上表名限定，明确字段的来源。

对于内连接而言，也可以使用 FROM 和 WHERE 表示：

```
SELECT d.department_id,
       e.department_id,
       d.department_name,
       e.first_name,
       e.last_name
FROM employees e, departments d
WHERE e.department_id = d.department_id;
```

在这种语法中，多个表在 FROM 子句中使用逗号进行分割，连接条件使用 WHERE 子句表示。实际上，在 SQL 历史中定义了两种多表连接的语法：

- ANSI SQL/86 标准使用 FROM 和 WHERE 关键字指定表的连接条件。
- ANSI SQL/92 标准使用 JOIN 和 ON 关键字指定表的连接条件。

推荐使用 JOIN 和 ON，它们的语义更清晰，更符合 SQL 的声明性。当 WHERE 子句中包含多个查询条件，又用于指定表的连接关系时，显得比较混乱。

## 12.2 左/右外连接

左外连接返回左表中所有的数据行；对于右表，如果没有匹配的数据，显示为空值。左外连接使用关键字 LEFT OUTER JOIN 表示，也可以简写成 LEFT JOIN。左外连接参考以下示意图（基于两个表的 id 进行连接）：

id	name	id	price		id	name	price
1	apple	1	9.5	➔	1	apple	9.5
2	banana	3	6.0		2	banana	
3	pear	5	8.8		3	pear	6.0
table1		table2			table1 LEFT OUTER JOIN table2		

查询首先返回左表中的全部数据（id 等于 1、2、3）。由于 id = 2 在 table2 中不存在对应的数据，对于 table2 中的字段返回空值。

由于某些部门刚刚成立,可能还没有员工,因此前面的内连接查询不会显示这些部门的信息。如果想要在连接查询中返回这些部门的信息,需要使用左外连接:

```
SELECT d.department_id,
       e.department_id,
       d.department_name,
       e.first_name,
       e.last_name
FROM departments d
LEFT JOIN employees e
ON e.department_id = d.department_id;
```

该查询的结果如下（显示部分内容）：

123 department_id	123 department_id	ABC department_name	ABC first_name	ABC last_name
120	[NULL]	Treasury	[NULL]	[NULL]
270	[NULL]	Payroll	[NULL]	[NULL]
230	[NULL]	IT Helpdesk	[NULL]	[NULL]
160	[NULL]	Benefits	[NULL]	[NULL]
250	[NULL]	Retail Sales	[NULL]	[NULL]
140	[NULL]	Control And Credit	[NULL]	[NULL]
170	[NULL]	Manufacturing	[NULL]	[NULL]
180	[NULL]	Construction	[NULL]	[NULL]
190	[NULL]	Contracting	[NULL]	[NULL]
200	[NULL]	Operations	[NULL]	[NULL]

右外连接返回右表中所有的数据行；对于左表，如果没有匹配的数据，显示为空值。右外连接使用关键字 **RIGHT OUTER JOIN** 表示，也可以简写成 **RIGHT JOIN**。也就是说：

```
table1 RIGHT JOIN table2
```

等价于：

```
table2 LEFT JOIN table1
```

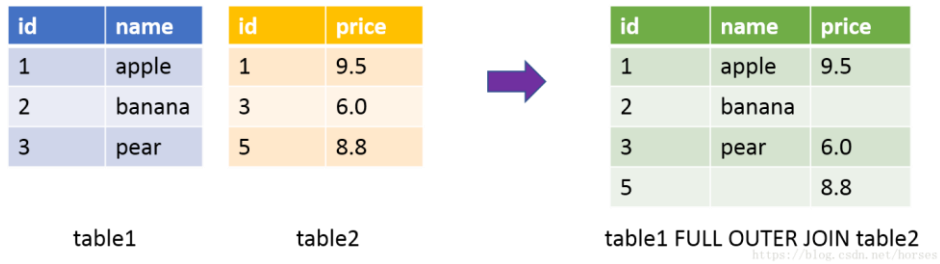
因此，上面的查询也可以使用右外连接来表示：

```
SELECT d.department_id,
       e.department_id,
       d.department_name,
       e.first_name,
       e.last_name
FROM departments d
RIGHT JOIN employees e
ON d.department_id = e.department_id;
```

## 12.3 全外连接

全外连接等效于左外连接加上右外连接，返回左表和右表中所有的数据行。全外连接使用关键字 **FULL OUTER JOIN** 表示，也可以简写成 **FULL JOIN**。

全外连接的示意图如下（基于两个表的 id 进行连接）：



查询首先返回两个表中匹配的数据（id 等于 1 和 3），对于 table1 中的 id = 2，table2 中的对应字段（price）显示为空，对于 table2 中的 id = 5，对应的 table1 中的字段（name）显示为空。

以下查询将员工表和部门表进行全外连接，连接字段为部门编号：

```
SELECT d.department_id,
       e.department_id,
       d.department_name,
       e.first_name,
       e.last_name
FROM departments d
FULL JOIN employees e
  ON d.department_id = e.department_id
WHERE e.employee_id IN (176, 177, 178)
   OR d.department_id IN (110, 120, 130);
```

为了显示方便，使用 **WHERE** 条件过滤掉了大部分的结果。

	123 department_id 🔍	123 department_id 🔍	ABC department_name 🔍	ABC first_name 🔍	ABC last_name 🔍
1	80	80	Sales	Jonathon	Taylor
2	80	80	Sales	Jack	Livingston
3	[NULL]	[NULL]	[NULL]	Kimberely	Grant
4	110	110	Accounting	Shelley	Higgins
5	110	110	Accounting	William	Gietz
6	120	[NULL]	Treasury	[NULL]	[NULL]
7	130	[NULL]	Corporate Tax	[NULL]	[NULL]

<https://tonydong.blog.csdn.net>

查询结果不但包含了没有员工的部门，同时还存在一个没有部门的员工。

对于**外连接**，需要注意 **WHERE** 条件和 **ON** 条件之间的差异：**ON** 条件是针对连接之前的数据进行过滤，**WHERE** 是针对连接之后的数据进行过滤，同一个条件放在不同的子句中可能会导致不同的结果。

以下示例将部门表与员工表进行左外连接查询，并且在 ON 子句中指定了多个条件：

```
SELECT d.department_id,  
       e.department_id,  
       d.department_name,  
       e.first_name,  
       e.last_name  
FROM departments d  
LEFT JOIN employees e  
  ON d.department_id = e.department_id AND e.employee_id = 0;
```

ON 子句指定了一个不存在的员工（e.employee\_id = 0），因此员工表不会返回任何数据。但是由于查询指定的是左外连接，仍然会返回部门信息，查询结果如下图所示（显示部分内容）。

	123 department_id 🔼🔼	123 department_id 🔼🔼	ABC department_name 🔼🔼	ABC first_name 🔼🔼	ABC last_name 🔼🔼
1	10	[NULL]	Administration	[NULL]	[NULL]
2	20	[NULL]	Marketing	[NULL]	[NULL]
3	30	[NULL]	Purchasing	[NULL]	[NULL]
4	40	[NULL]	Human Resources	[NULL]	[NULL]
5	50	[NULL]	Shipping	[NULL]	[NULL]
6	60	[NULL]	IT	[NULL]	[NULL]
7	70	[NULL]	Public Relations	[NULL]	[NULL]
8	80	[NULL]	Sales	[NULL]	[NULL]
9	90	[NULL]	Executive	[NULL]	[NULL]
10	100	[NULL]	Finance	[NULL]	[NULL]

对于相同的查询条件，使用 WHERE 子句的示例如下：

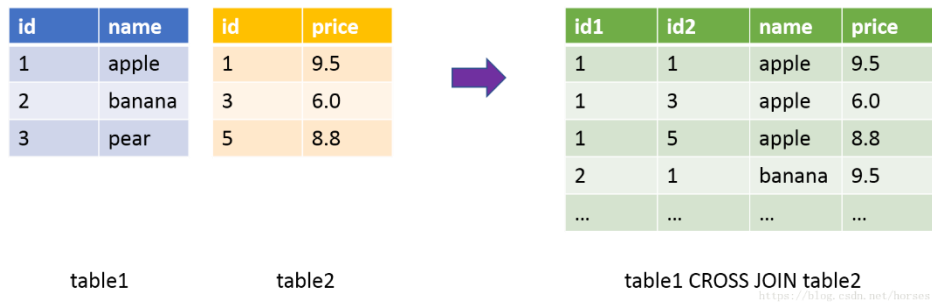
```
SELECT d.department_id,  
       e.department_id,  
       d.department_name,  
       e.first_name,  
       e.last_name  
FROM departments d  
LEFT JOIN employees e  
  ON d.department_id = e.department_id  
WHERE e.employee_id = 0;
```

查询结果没有返回任何数据，因为左连接产生的结果经过 WHERE 条件（e.employee\_id = 0）过滤之后没有任何满足的数据。

## 12.4 交叉连接

当连接查询没有指定任何连接条件时，就称为交叉连接。交叉连接使用关键字 CROSS JOIN 表示，也称为笛卡尔积（Cartesian product）。

两个表的笛卡尔积相当于一个表的所有行和另一个表的所有行两两组合，结果数量为两个表的行数相乘。假如第一个表有 100 行，第二个表有 200 行，它们的交叉连接将会产生 100 × 200 = 20000 行结果。交叉连接的示意图如下（基于两个表的 id 进行连接）：



以下查询通过笛卡儿积返回九九乘法表：

```
SELECT v || '*' || h || '=' || v*h
FROM generate_series(1,9) v
CROSS JOIN generate_series(1,9) h;
```

查询的结果如下图所示。

```
?column?|
-----+
1*1=1   |
1*2=2   |
1*3=3   |
1*4=4   |
1*5=5   |
1*6=6   |
1*7=7   |
1*8=8   |
1*9=9   |
2*1=2   |
...
```

上面的交叉连接也可以使用以下等效写法：

```
SELECT v || '*' || h || '=' || v*h
FROM generate_series(1,9) v, generate_series(1,9) h;

SELECT v || '*' || h || '=' || v*h
FROM generate_series(1,9) v
JOIN generate_series(1,9) h ON TRUE;
```

## 12.5 自然连接

对于连接查询，如果满足以下条件，可以使用 USING 替代 ON 子句，简化连接条件的输入：

- 连接条件是等值连接，即  $t1.col1 = t2.col1$ ；
- 两个表中的列必须同名同类型，即  $t1.col1$  和  $t2.col1$  的类型相同。

由于 employees 表和 departments 表中的 department\_id 字段名称和类型都相同，可以使用 USING 简写前文中的连接查询：

```
SELECT d.department_id,
       e.department_id,
```

```

        d.department_name,
        e.first_name,
        e.last_name
FROM employees e
JOIN departments d
USING (department_id);

```

USING 条件中的字段不需要指定表名，它是公共的字段。

如果 USING 子句中包含了两个表中所有的这种同名同类型字段，可以使用更加简单的自然连接（NATURAL JOIN）表示。例如，employees 表和 departments 表拥有 2 个同名同类型字段：department\_id 和 manager\_id，如果基于这 2 个字段进行等值连接，可以使用自然连接：

```

SELECT d.department_id,
       d.department_name,
       e.first_name,
       e.last_name
FROM departments d
NATURAL JOIN employees e;

```

查询的结果如下图所示（显示部分内容）。

	123 department_id	ABC department_name	ABC first_name	ABC last_name
1	90	Executive	Neena	Kochhar
2	90	Executive	Lex	De Haan
3	100	Finance	Daniel	Faviet
4	100	Finance	John	Chen
5	100	Finance	Ismael	Sciarra
6	100	Finance	Jose Manuel	Urman
7	100	Finance	Luis	Popp
8	30	Purchasing	Alexander	Khoo
9	30	Purchasing	Shelli	Baida
10	30	Purchasing	Sigal	Tobias

查询返回的员工满足以下条件：他/她的经理也是他/她所在部门的经理。

## 12.6 自连接

连接（Self Join）是指连接操作符的两边都是同一个表，即把一个表和自己进行连接。自连接本质上并没有什么特殊之处，主要用于处理那些对自己进行了外键引用的表。

例如，员工表中的经理字段（manager\_id）是一个外键列，指向了员工表自身的员工编号字段（employee\_id）。如果要显示员工姓名以及他们经理的姓名，可以通过自连接实现：

```

SELECT e.first_name||', '||e.last_name AS employee_name,
       m.first_name||', '||m.last_name AS manager_name
FROM employees m
JOIN employees e
ON m.employee_id = e.manager_id;

```

由于查询多次使用了同一个表（employees），必须为它们指定不同的表别名。查询的结果如下图所示（显示部分内容）。

ABC employee_name	ABC manager_name
Neena, Kochhar	Steven, King
Lex, De Haan	Steven, King
Nancy, Greenberg	Neena, Kochhar
Daniel, Faviot	Nancy, Greenberg
John, Chen	Nancy, Greenberg
Ismael, Sciarra	Nancy, Greenberg
Jose Manuel, Urman	Nancy, Greenberg
Luis, Popp	Nancy, Greenberg
Den, Raphaely	Steven, King
Alexander, Khoo	Den, Raphaely

如果还需要知道员工的职位信息，比如职位名称，可以在连接查询中加上 jobs 表。以下是三个表连接查询的示例：

```
SELECT d.department_name,
       e.first_name||', '||e.last_name AS employee_name,
       j.job_title
FROM departments d
JOIN employees e ON d.department_id = e.department_id
JOIN jobs j ON j.job_id = e.job_id;
```

查询结果如下（显示部分内容）：

ABC department_name	ABC employee_name	ABC job_title
Executive	Steven, King	President
Executive	Neena, Kochhar	Administration Vice President
Executive	Lex, De Haan	Administration Vice President
Finance	Nancy, Greenberg	Finance Manager
Finance	Daniel, Faviot	Accountant
Finance	John, Chen	Accountant
Finance	Ismael, Sciarra	Accountant
Finance	Jose Manuel, Urman	Accountant
Finance	Luis, Popp	Accountant
Purchasing	Den, Raphaely	Purchasing Manager



## 第 13 章 CASE 条件表达式

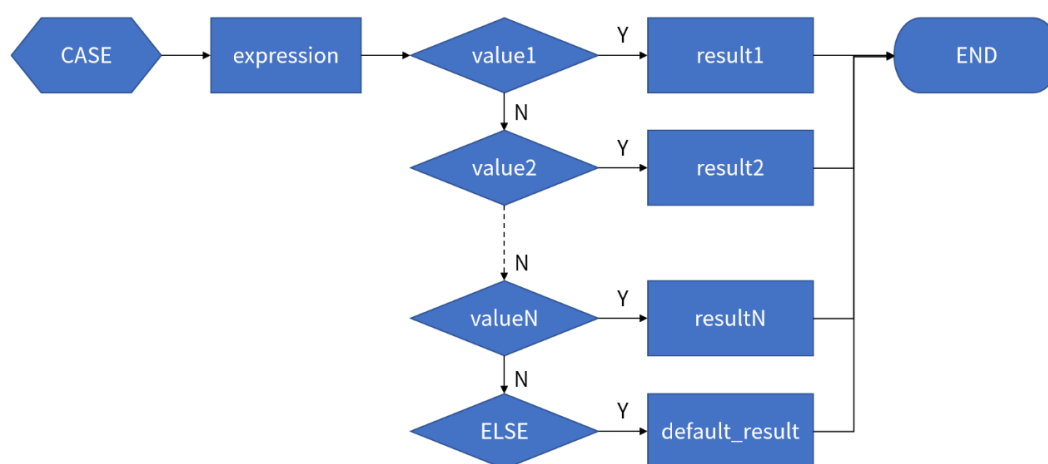
CASE 表达式的作用就是为 SQL 语句增加类似于 IF-THEN-ELSE 的逻辑处理功能，可以根据不同的条件返回不同的结果。PostgreSQL 支持两种形式的条件表达式：**简单 CASE 表达式**和**搜索 CASE 表达式**。另外，为了方便空值处理，PostgreSQL 还提供了两个缩写形式的 CASE 表达式（函数）：**NULLIF** 和 **COALESCE**。

### 13.1 简单 CASE 表达式

简单 CASE 表达式的语法如下：

```
CASE expression
  WHEN value1 THEN result1
  WHEN value2 THEN result2
  [...]
  [ELSE default_result]
END;
```

表达式的计算过程如下图所示。



首先计算表达式（`expression`）的值，然后依次与 **WHEN** 列表中的值（`value1`, `value2`, ...）进行比较，找到第一个匹配的值，然后返回对应 **THEN** 列表中的结果（`result1`, `result2`, ...）；如果没有找到匹配的值，返回 **ELSE** 中的默认值；如果没有指定 **ELSE**，返回 **NULL**。

下面的查询使用简单 CASE 表达式统计每个部门的人数，并且转换为列的方式显示：

```
SELECT SUM(CASE department_id WHEN 10 THEN 1 ELSE 0 END) AS dept_10_count,
       SUM(CASE department_id WHEN 20 THEN 1 ELSE 0 END) AS dept_20_count,
       SUM(CASE department_id WHEN 30 THEN 1 ELSE 0 END) AS dept_30_count
FROM employees;
```

dept_10_count	dept_20_count	dept_30_count
1	2	6

需要注意的是每个分支的结果必须具有相同的数据类型，否则会产生类型错误。例如，以下

示例对于不同条件返回的数据类型不一致：

```
SELECT first_name,  
       last_name,  
       CASE department_id  
         WHEN 10 THEN 'Administration'  
         WHEN 20 THEN 20  
         WHEN 30 THEN 'Purchasing'  
         ELSE 'Others' END AS department_name  
FROM employees;
```

```
SQL 错误 [22P02]: ERROR: invalid input syntax for integer: "Others"  
Position: 183
```

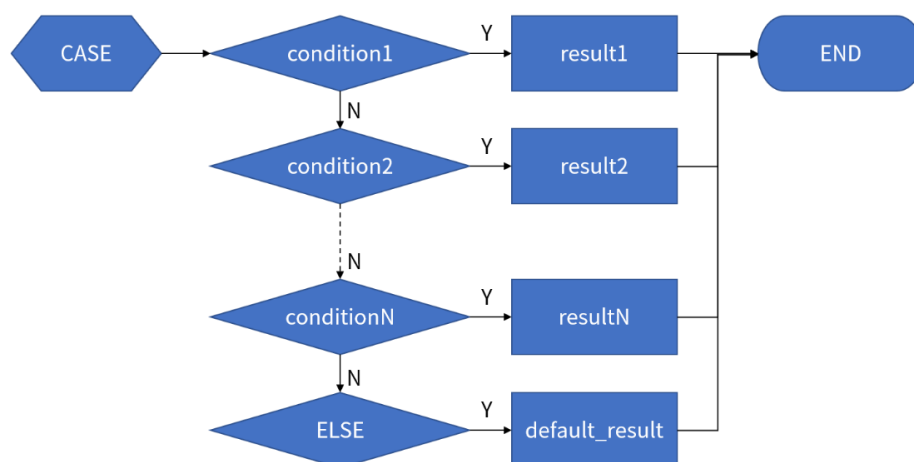
简单 **CASE** 表达式在进行计算的时候，使用的是等值比较(=)，能够支持简单的逻辑处理。如果想要基于更加复杂的条件进行判断，例如根据某个列的取值范围返回不同的信息，或者判断表达式的值是否为空，都需要使用更加强大的搜索 **CASE** 表达式。

## 13.2 搜索 CASE 表达式

搜索 **CASE** 表达式的语法如下：

```
CASE  
  WHEN condition1 THEN result1  
  WHEN condition2 THEN result2  
  ...  
  [ELSE default_result]  
END
```

表达式的计算过程如下图所示。



按照顺序依次计算 **WHEN** 子句中的条件（**condition1**, **condition2**, ...），找到第一个结果为真的分支，返回相应的结果；如果没有任何条件为真，返回 **ELSE** 中的默认值；如果此时没有指定 **ELSE**，返回空值。

搜索 **CASE** 表达式可以在 **WHEN** 子句中构造复杂的条件，完成各种逻辑处理。首先，所有

的简单 CASE 表达式都可以替换称等价的搜索 CASE 表达式。我们将前面的示例改写如下：

```
SELECT SUM(CASE WHEN department_id = 10 THEN 1 ELSE 0 END) AS dept_10_count,  
       SUM(CASE WHEN department_id = 20 THEN 1 ELSE 0 END) AS dept_20_count,  
       SUM(CASE WHEN department_id = 30 THEN 1 ELSE 0 END) AS dept_30_count  
FROM employees;
```

以下示例根据薪水的范围将员工的收入分为高中低三个档次：

```
SELECT e.first_name,  
       e.last_name,  
       e.salary,  
       CASE  
         WHEN e.salary < 5000 THEN '低'  
         WHEN e.salary < 15000 THEN '中'  
         ELSE '高'  
       END AS salary_level  
FROM employees e;
```

first_name	last_name	salary	salary_level
Steven	King	24000.00	高
Neena	Kochhar	17000.00	高
Lex	De Haan	17000.00	高
Nancy	Greenberg	12008.00	中
Daniel	Faviet	9000.00	中
John	Chen	8200.00	中
Ismael	Sciarra	7700.00	中
Jose Manuel	Urman	7800.00	中
Luis	Popp	6900.00	中
Den	Raphaely	11000.00	中
Alexander	Khoo	3100.00	低
Shelli	Baida	2900.00	低
...			

如果薪水低于 5000，满足第一个 WHEN 子句的条件，返回“低”；否则进入第二个 WHEN 子句，如果小于 15000（同时大于等于 5000），返回“中”；否则进入 ELSE 子句，返回“高”。

既然是表达式，CASE 表达式除了可以用于 SELECT 列表，也可以出现在其他 SQL 子句中，例如 WHERE 条件子句、GROUP BY 分组子句、ORDER BY 排序子句等。以下示例除了将薪水显示为三个档次，同时还按照档次和名字进行排序：

```
SELECT e.first_name,  
       e.last_name,  
       e.salary,  
       CASE  
         WHEN e.salary < 5000 THEN '低'  
         WHEN e.salary < 15000 THEN '中'  
         ELSE '高'  
       END AS salary_level  
FROM employees e  
ORDER BY CASE  
         WHEN e.salary < 5000 THEN 3  
         WHEN e.salary < 15000 THEN 2  
         ELSE 1  
       END, first_name;
```

first_name	last_name	salary	salary_level
Lex	De Haan	17000.00	高
Neena	Kochhar	17000.00	高
Steven	King	24000.00	高
Adam	Fripp	8200.00	中
Alberto	Errazuriz	12000.00	中
Alexander	Hunold	9000.00	中
Allan	McEwen	9000.00	中
Alyssa	Hutton	8800.00	中
Amit	Banda	6200.00	中
Bruce	Ernst	6000.00	中
Charles	Johnson	6200.00	中
Christopher	Olsen	8000.00	中
Clara	Vishney	10500.00	中
...			

### 13.3 缩写函数

除了以上两种形式的 CASE 表达式之外，PostgreSQL 还提供了两个与 NULL 相关的缩写 CASE 表达式（函数）：NULLIF 和 COALESCE。

NULLIF 函数的用法如下：

```
NULLIF(expression_1, expression_2)
```

NULLIF 函数包含 2 个参数，如果第一个参数等于第二个参数，返回 NULL；否则，返回第一个参数的值。它可以使用等价的 CASE 表达式表示为：

```
CASE
  WHEN expression_1 = expression_2 THEN NULL
  ELSE expression_1
END
```

以下示例说明了 NULLIF 函数的效果：

```
SELECT NULLIF(1, 1), NULLIF('A', 'B');
```

```
nullif|nullif|
-----|-----|
      |A      |
```

NULLIF 函数的一个常见用途是防止除零错误：

```
SELECT 1 / 0; -- 除零错误

SELECT 1 / NULLIF(0, 0);
```

COALESCE 函数的语法如下：

```
COALESCE(expression_1, expression_2, expression_3, ...)
```

COALESCE 函数接受多个参数，并且返回第一个非空的参数值；如果所有参数都为空值，

返回 NULL 值。它可以使用等价的 CASE 表达式表示为：

```
CASE
  WHEN expression_1 IS NOT NULL THEN expression_1
  WHEN expression_2 IS NOT NULL THEN expression_2
  WHEN expression_3 IS NOT NULL THEN expression_3
  ...
END
```

以下示例将佣金比率为空的数据显示为 0：

```
SELECT e.first_name,
       e.last_name,
       e.commission_pct,
       COALESCE(e.commission_pct, 0)
FROM employees e;
```

first_name	last_name	commission_pct	coalesce
Steven	King		0
Neena	Kochhar		0
Lex	De Haan		0
...			
John	Russell	0.40	0.40
Karen	Partners	0.30	0.30
Alberto	Errazuriz	0.30	0.30
Gerald	Cambrault	0.30	0.30
Eleni	Zlotkey	0.20	0.20
Peter	Tucker	0.30	0.30
...			

# 第 14 篇 常用函数

函数（function）是一些预定义好的代码模块，可以将输入进行计算和处理，最终输出一个结果值。PostgreSQL 函数可以分为两类：**标量函数**（scalar function）和**聚合函数**（aggregation function）。标量函数针对每个输入都会返回相应的结果，聚合函数针对一组输入汇总出一个结果。在第 11 篇中已经介绍了几个常见的聚合函数。

本篇主要介绍 PostgreSQL 提供的标量函数。为了便于学习，可以将常见的系统函数分为以下类别：数学函数、字符函数、日期时间函数以及类型转换函数。除了可以使用这些系统内置的函数之外，PostgreSQL 也支持创建自定义的函数（UDF）。

## 14.1 数学函数

数学函数和运算符用于执行算术运算，输入和输出通常都是数字类型。

### 14.1.1 算术运算符

PostgreSQL 支持以下算术运算符：

运算符	描述	示例	结果
+	加法	2 + 3	5
-	减法	2 - 3	-1
*	乘法	2 * 3	6
/	整除	5 / 2	2
%	模除（求余）	5 % 4	1
^	求幂（左边为底数，右边为指数）	2.0 ^ 3.0	8
/	平方根	/ 25.0	5
/	立方根	/ 27.0	3
!	阶乘	5 !	120
!!	阶乘（前置运算符）	!! 5	120
@	绝对值	@ -5.0	5
&	按位与	91 & 15	11
	按位或	32   3	35
#	按位异或	17 # 5	20

~	按位非	~1	-2
<<	按位左移	1 << 4	16
>>	按位右移	8 >> 2	2

其中，按位运算只对整型数字类型有效；左移 N 位相当于乘以 2 的 N 次方，右移 N 位相当于除以 2 的 N 次方。

### 14.1.2 绝对值

**abs(x)**函数用于计算 x 的绝对值。例如：

```
SELECT abs(-17.4);
| abs |
|-----|
| 17.4 |
```

### 14.1.3 取整函数

**ceil(dp)/ceiling(dp)**函数用于计算大于或等于 dp 的最小整数；**floor(dp)**函数用于计算小于或等于 dp 的最大整数；**round(dp)**函数四舍五入为整数；**trunc(dp)**函数向零取整。

```
SELECT ceil(-42.8), floor(-42.8), round(12.45), trunc(12.8);
| ceil | floor | round | trunc |
|-----|-----|-----|-----|
| -42  | -43   | 12    | 12    |
```

另外，**round(dp, s)**函数四舍五入到 s 位小数；**trunc(dp, s)**函数截断到 s 位小数。

### 14.1.4 乘方与开方

**power(a, b)**函数计算 a 的 b 次方；**sqrt(dp)**函数计算 dp 的平方根；**cbirt(dp)**函数计算 dp 的立方根。

```
SELECT power(2, 3), sqrt(4), cbirt(27);
| power | sqrt | cbirt |
|-----|-----|-----|
| 8     | 2    | 3.0000000000000004 |
```

### 14.1.5 指数与对数

**exp(dp)**函数计算以自然常数 e 为底的指数，**ln(dp)**函数计算以自然常数 e 为底数的对数，**log(dp)/log10(dp)**函数计算以 10 为底的对数，**log(b, x)**函数计算以 b 为底的对数。

```
SELECT exp(1.0), ln(2.718281828459045), log(100), log(2.0, 16.0);
| exp | ln | log | log |
|-----|-----|-----|-----|
| 2.718281828459045 | 0.9999999999999999 | 2 | 4 |
```

### 14.1.6 整数商和余数

**div(y, x)**函数计算 y 除以 x 的整数商，**mod(y, x)**函数计算 y 除以 x 的余数。

```
SELECT div(9,4), mod(9,4);
| div | mod |
|-----|-----|
| 2 | 1 |
```

### 14.1.7 弧度与角度

**degrees(dp)**函数用于将弧度转为角度，**radians(dp)**函数用于将角度转弧度。

```
SELECT degrees(1.57), radians(90.0);
| degrees | radians |
|-----|-----|
| 89.95437383553924 | 1.5707963267948966 |
```

### 14.1.8 常量 $\pi$

**pi()**函数用于返回常量“ $\pi$ ”的值。

```
SELECT pi();
| pi |
|-----|
| 3.141592653589793 |
```

### 14.1.9 符号函数

**sign(dp)**函数返回参数的正负号，可能的结果为-1、0、+1。

```
SELECT sign(-8.4);
| sign |
|-----|
| -1 |
```

### 14.1.10 生成随机数

PostgreSQL 提供了用于返回一个随机数的函数 **random()**。

```
SELECT random();
| random |
|-----|
| 0.07772749848663807 |
```

**random()**返回一个大于等于 0 小于 1 的随机数，类型为双精度浮点数。

**setseed(dp)**函数可以为随后一次运行的 **random()**函数设置种子数，范围： $-1.0 \leq dp \leq 1.0$ 。

```
SELECT setseed(0);
SELECT random();
| random |
|-----|
| 0.8401877167634666 |
```



相同的种子可以得到相同的随机数，用于重现结果。

## 14.2 字符函数

### 14.2.1 字符串连接

**concat(str, ...)**函数用于连接字符串，并且忽略其中的 NULL 参数；**concat\_ws(sep, str, ...)**函数使用指定分隔符 **sep** 连接字符串。

```
SELECT concat(2, NULL, 22), concat_ws(' and ', 2, NULL, 22);
| concat | concat_ws |
|-----|-----|
|      22 | 2 and 22 |
```

两个竖杠 (||) 也可以用于连接字符串，但是 NULL 参数将会返回 NULL。

```
SELECT 'Post' || 'greSQL', 'Post' || NULL || 'greSQL';
| ?column? | ?column? |
|-----|-----|
| PostgreSQL | (null) |
```

### 14.2.2 字符与编码

**ascii(string)**函数返回第一个字符的 ASCII 码。对于 UTF8 返回 Unicode 码；对于其他多字节编码，参数必须是一个 ASCII 字符。

```
SELECT ascii('x');
| ascii |
|-----|
|    120 |
```

**chr(int)**函数返回编码对应的字符。对于 UTF8，参数指定的是 Unicode 码；对于其他多字节编码，参数必须对应一个 ASCII 字符。参数不允许为 0（空字符），因为 text 数据类型不能存储空字符。

```
SELECT chr(120);
| chr |
|-----|
|    x |
```

### 14.2.3 字符串长度

**bit\_length(string)**函数用于计算字符串包含的比特数；**length(string)**、**char\_length(string)**、**character\_length(string)**函数计算字符串包含的字符数；**octet\_length(string)**函数计算字符串包含的字节数。

```
SELECT bit_length('jose'), length('jose'), octet_length('jose');
| bit_length | length | octet_length |
|-----|-----|-----|
|          32 |      4 |            4 |
```

## 14.2.4 大小写转换

**lower(string)**函数将字符串转换为小写形式，**upper(string)**函数将字符串转换为大写形式，**initcap(string)**函数将每个单词的首字母大写，其他字母小写。

```
SELECT lower('TOM'), upper('tom'), initcap('hi THOMAS');
| lower | upper | initcap |
|-----|-----|-----|
| tom   | TOM   | Hi Thomas |
```

## 14.2.5 子串查找与替换

**substring(string [FROM] [for])**函数用于提取从位置 FROM 开始的 for 个字符子串，位置从 1 开始计算。**substr(string, FROM [, count])**的作用相同。

```
SELECT substring('Thomas' FROM 2 for 3), substr('Thomas',2, 3);
| substring | substr |
|-----|-----|
| hom      | hom    |
```

**left(str, n)**函数返回字符串左边的 n 个字符。如果 n 为负数，返回除了最后|n|个字符之外的所有字符。

**right(str, n)**函数返回字符串右边的 n 个字符。如果 n 为负数，返回除了左边|n|个字符之外的字符。

```
SELECT left('abcde', 2), right('abcde', 2);
| left | right |
|-----|-----|
| ab   | de    |
```

**substring(string FROM pattern)**函数提取匹配 POSIX 正则表达式的子串。

**substring(string FROM pattern for escape)**函数提取匹配 SQL 正则表达式的子串。

```
SELECT substring('Thomas' FROM '...$'), substring('Thomas' FROM '%#"o_a#"'
for '#');
| substring | substring |
|-----|-----|
| mas      | oma      |
```

**regexp\_match(string, pattern [, flags])**函数返回匹配 POSIX 正则表达式的第一个子串。

```
SELECT regexp_match('foobarbequebaz', '(bar) (beque)');
| regexp_match |
|-----|
| {bar,beque} |
```

**regexp\_matches(string, pattern [, flags])**函数返回匹配 POSIX 正则表达式的所有子串，结果是一个集合。

```
SELECT regexp_matches('foobarbequebaz', 'ba.', 'g');
| regexp_matches |
|-----|
| bar           |
| baz           |
```

**position(substring in string)**返回子串的位置；**strpos(string, substring)**函数的作用相同，但是参数顺序相反。

```
SELECT position('om' in 'Thomas'), strpos('Thomas', 'om');
| position | strpos |
|-----|-----|
|          3 |          3 |
```

**starts\_with(string, prefix)**函数判断 string 是否以 prefix 开头，如果是则返回 true；否则返回 false。

```
SELECT starts_with('alphabet', 'alph');
| starts_with |
|-----|
| true       |
```

**replace(string, FROM, to)**函数将字符串 string 中的 FROM 子串替换为 to 子串；**regexp\_replace(string, pattern, replacement [, flags])**函数将字符串 string 中匹配 POSIX 正则表达式 pattern 的子串替换为 replacement。

```
SELECT replace('abcdefabcdef', 'cd', 'XX'), regexp_replace('Thomas',
'.[mN]a.', 'M');
| replace | regexp_replace |
|-----|-----|
| abXXefabXXef | ThM |
```

**translate(string, FROM, to)**函数将字符串 string 中出现在 FROM 中的字符串替换成 to 中相应位置的字符。如果 FROM 长度大于 to，在 to 中没有对应值的字符将被删除。

```
SELECT translate('12345', '143', 'ax');
| translate |
|-----|
| a2x5     |
```

**overlay(string placing substring FROM [for])**函数使用 substring 覆盖字符串 string 中从 FROM 开始的 for 个字符。

```
SELECT overlay('Txxxxas' placing 'hom' FROM 2 for 4);
| overlay |
|-----|
| Thomas |
```

## 14.2.6 截断与填充

**trim([leading | trailing | both] [characters] FROM string)**函数从字符串的开头（leading）、结尾（trailing）或者两端（both）删除由指定字符 characters（默认为空格）组成的最长子串；**trim([leading | trailing | both] [FROM] string [, characters])**函数的作用相同。

```
SELECT trim(both 'xyz' FROM 'yxTomxx');
| btrim |
|-----|
| Tom   |
```

**btrim(string [, characters])**函数的作用与上面 trim 函数的 both 选项相同；**ltrim(string [, characters])**与上面 trim 函数的 leading 选项相同；**rtrim(string [, characters])**函数上面 trim 函数的

**trailing** 选项相同。

```
SELECT btrim('yxTomxx', 'xyz'), ltrim('yxTomxx', 'xyz'), rtrim('yxTomxx', 'xyz');
| btrim | ltrim | rtrim |
|-----|-----|-----|
| Tom   | Tomxx | yxTom |
```

**lpad(string, length [, fill ])**函数在 **string** 左侧使用 **fill** 中的字符（默认空格）进行填充，直到长度为 **length**。如果 **string** 长度大于 **length**，从右侧截断到长度 **length**。

**rpadd(string, length [, fill ])**函数在 **string** 右侧使用 **fill** 中的字符（默认空格）进行填充，直到长度为 **length**。如果 **string** 长度大于 **length**，从右侧截断到长度 **length**。

**repeat(string, number)**函数将字符串 **string** 重复 **N** 次。

```
SELECT lpad('hi', 5, 'xy'), rpadd('hi', 5, 'xy'), repeat('Pg', 4);
| lpad | rpadd | repeat |
|-----|-----|-----|
| xyxhi | hixyx | PgPgPgPg |
```

## 14.2.7 字符串格式化

**format(formatstr , formatarg)**用于对字符串格式化，类似于 C 语言中的 **sprintf** 函数。

```
SELECT format('Hello %s, %1$s', 'World');
| format |
|-----|
| Hello World, World |
```

 关于 **format** 函数的格式化参数可以参考[官方文档](#)。

## 14.2.8 MD5 值

**md5(string)**函数用于返回十六进制格式的 MD5 值。

```
SELECT md5('abc');
| md5 |
|-----|
| 900150983cd24fb0d6963f7d28e17f72 |
```

## 14.2.9 字符串拆分

**regexp\_split\_to\_table(string, pattern[, flags])**函数用于拆分字符串，使用 POSIX 正则表达式作为分隔符。函数的返回类型是 **text** 集合。

```
SELECT regexp_split_to_table('hello world', '\s+');
| regexp_split_to_table |
|-----|
| hello |
| world |
```


**split\_part(string, delimiter, field)**函数使用 **delimiter** 拆分字符串，并返回指定项（从 1 开始计数）。

```
SELECT split_part('abc~@~def~@~ghi', '~@~', 2);
| split_part |
|-----|
|      def  |
```

14.2.10 字符串反转

**reverse(str)**函数用于将字符串反转。

```
SELECT reverse('上海自来水');
| reverse |
|-----|
| 水来自海上|
```

 更多字符函数可以参考[官方文档](#)。

14.3 日期时间函数

PostgreSQL 提供了以下日期和时间运算的算术运算符。

运算符	示例	结果
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (days)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'

*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

PostgreSQL 还提供了大量用于日期和时间数据处理的函数。

### 14.3.1 计算时间间隔

**age(timestamp, timestamp)**函数用于计算两个时间点之间的间隔，**age(timestamp)**函数用于计算当前日期的凌晨 12 点到该时间点之间的间隔。

```
SELECT age(timestamp '2020-12-31', timestamp '2020-01-01'), age(timestamp '2020-01-01');
 age          | age          |
-----|-----|
11 mons 30 days|2 mons 2 days|
```

2020 年 12 月 31 日到 2020 年 01 月 01 日之间有 11 个月零 30 天；今天（2020 年 03 月 03 日）距离 2020 年 01 月 01 日已经过了 2 个月零 2 天。

### 14.3.2 获取时间中的信息

**date\_part(text, timestamp)**和 **extract(field FROM timestamp)**函数用于获取日期时间中的某一部分，例如年份、月份、小时等；**date\_part(text, interval)**和 **extract(field FROM interval)**函数用于获取时间间隔中的某一部分。

```
SELECT date_part('year', timestamp '2020-03-03 20:38:40'), extract(year FROM
timestamp '2020-03-03 20:38:40'),
       date_part('month', interval '1 years 5 months'), extract(month FROM
interval '1 years 5 months');
date_part|date_part|date_part|date_part|
-----|-----|-----|-----|
2020|2020|5|5|
```

通过返回字段的标题可以看出，**extract** 函数实际上也是调用了 **date\_part** 函数，只是参数方式不同。这两个函数支持获取的信息包括：

- century, 世纪；
- day, 对于 timestamp, 返回月份中的第几天；对于 interval, 返回天数；
- decade, 年份除以 10；
- dow, 星期天（0）到星期六（6）；
- doy, 一年中的第几天，（1 - 365/366）；
- epoch, 对于 timestamp WITH time zone, 返回 1970-01-01 00:00:00 UTC 到该时间的秒数；对于 date 和 timestamp, 返回本地时间的 1970-01-01 00:00:00 到该时间的秒数；对于 interval, 返回以秒数表示的该时间间隔；
- hour, 小时（1 - 23）；
- isodow, ISO 8601 标准中的星期一（1）到星期天（7）；

- isoyear, ISO 8601 标准定义的日期所在的年份。每年从包含 1 月 4 日的星期一开始, 2017 年 01 月 01 日属于 2016 年;
- microseconds, 微秒, 包含秒和小数秒在内的数字乘以 1000000;
- millennium, 千年;
- milliseconds, 毫秒, 包含秒和小数秒在内的数字乘以 1000;
- minute, 分钟, (0 - 59);
- month, 月份;
- quarter, 季度, (1 - 4);
- second, 秒数, 包含小数秒;
- timezone, UTC 时区, 单位为秒;
- timezone\_hour, UTC 时区中的小时部分;
- timezone\_minute, UTC 时区中的分钟部分;
- week, ISO 8601 标准中的星期几, 每年从第一个星期四所在的一周开始;
- year, 年份。

### 14.3.3 截断日期/时间

**date\_trunc(field, source [, time\_zone ])**函数用于将 timestamp、timestamp WITH time zone、date、time 或者 interval 数据截断到指定的精度。

```
SELECT date_trunc('year', timestamp '2020-03-03 20:38:40'),
       date_trunc('day',      timestampz      '2020-03-03      20:38:40+00',
'Asia/Shanghai'),
       date_trunc('hour', interval '2 days 3 hours 40 minutes');
date_trunc      |date_trunc      |date_trunc      |
-----|-----|-----|
2020-01-01 00:00:00|2020-03-04 00:00:00|2 days 03:00:00|
```

date\_trunc 函数支持以下截断精度:

- microseconds
- milliseconds
- second
- minute
- hour
- day
- week
- month
- quarter
- year
- decade
- century
- millennium

### 14.3.4 创建日期/时间

**make\_date(year int, month int, day int)**函数用于创建一个日期:

```
SELECT make_date(2020, 03, 15);
make_date |
-----|
2020-03-15|
```

**make\_interval(years int DEFAULT 0, months int DEFAULT 0, weeks int DEFAULT 0, days int DEFAULT 0, hours int DEFAULT 0, mins int DEFAULT 0, secs double precision DEFAULT 0.0)**函数通过指定年、月、日等信息创建一个时间间隔。

```
SELECT make_interval(days => 1, hours => 5);
make_interval |
-----|
1 day 05:00:00|
```

**make\_time(hour int, min int, sec double precision)**函数通过指定小时、分钟和秒数创建一个时间。

```
SELECT make_time(1, 2, 30.5);
make_time |
-----|
01:02:30.5|
```

**make\_timestamp(year int, month int, day int, hour int, min int, sec double precision)** 函数通过指定年、月、日、时、分、秒创建一个时间戳。

```
SELECT make_timestamp(2020, 3, 15, 8, 20, 23.5);
make_timestamp |
-----|
2020-03-15 08:20:23.5|
```

**make\_timestamptz(year int, month int, day int, hour int, min int, sec double precision, [ timezone text ])**函数通过指定年、月、日、时、分、秒创建一个带时区的时间戳。如果没有指定时区，使用当前时区。

```
SELECT make_timestamptz(2020, 3, 15, 8, 20, 23.5);
make_timestamptz |
-----|
2020-03-15 08:20:23.5+08|
```

**to\_timestamp(double precision)**函数将 Unix 时间戳（自从 1970-01-01 00:00:00+00 以来的秒数）转换为 PostgreSQL 时间戳数据。

```
SELECT to_timestamp(1583152349);
to_timestamp |
-----|
2020-03-02 20:32:29+08|
```

### 14.3.5 获取系统时间

PostgreSQL 提供了大量用于获取系统当前日期和时间的函数，例如 `current_date`、`current_time`、`current_timestamp`、`clock_timestamp()`、`localtimestamp`、`now()`、`statement_timestamp()`等；同时还支持延迟语句执行的 `pg_sleep()`等函数，具体可以参考[这篇文章](#)。




### 14.3.6 时区转换

**AT TIME ZONE** 运算符用于将 timestamp without time zone、timestamp WITH time zone 以及 time WITH time zone 转换为指定时区中的时间。

```
SELECT TIMESTAMP '2020-03-03 20:38:40' AT TIME ZONE 'Asia/Shanghai',
       TIMESTAMP WITH TIME ZONE '2020-03-03 20:38:40-05:00' AT TIME ZONE
'Asia/Shanghai',
       TIME WITH TIME ZONE '20:38:40-05:00' AT TIME ZONE 'Asia/Shanghai';
timezone           |timezone           |timezone           |
-----|-----|-----|
2020-03-03 20:38:40+08|2020-03-04 09:38:40|09:38:40+08|
```

**timezone(zone, timestamp)**函数等价于 SQL 标准中的 *timestamp AT TIME ZONE zone*。

 还有一些关于日期时间的函数，可以参考[官方文档](#)。

## 14.4 类型转换函数

类型转换函数用于将数据从一种类型转换为另一种类型。

### 14.4.1 CAST 函数

**CAST ( expr AS data\_type )**函数用于将 expr 转换为 data\_type 数据类型；PostgreSQL 类型转换运算符 (::) 也可以实现相同的功能。例如：

```
SELECT CAST ('15' AS INTEGER), '2020-03-15'::DATE;
int4|date      |
----|-----|
15|2020-03-15|
```

如果数据无法转换为指定的类型，将会返回错误：

```
SELECT CAST ('A15' AS INTEGER);
SQL 错误 [22P02]: 错误: 无效的类型 integer 输入语法: "A15"
位置: 14
```

### 14.4.2 to\_date 函数

**to\_date(string, format)**函数用于将字符串 string 按照 format 格式转换为日期类型。

```
SELECT to_date('2020/03/15', 'YYYY/MM/DD');
to_date |
-----|
2020-03-15|
```

其中，YYYY 代表四位数的年；MM 代表两位数的月；DD 代表两位数的日。更多的格式选项可以参考[官方文档](#)。

### 14.4.3 to\_timestamp 函数

**to\_timestamp(string, format)**函数用于将字符串 string 按照 format 格式转换为 timestamp

WITH time zone 类型。

```
SELECT to_timestamp('2020-03-15 19:08:00.678', 'YYYY-MM-DD HH24:MI:SS.MS');
to_timestamp |
-----|
2020-03-15 19:08:00.678+08|
```

其中，HH24 表示 24 小时制的小时；MI 表示分钟；SS 表示秒数；MS 表示毫秒数。

#### 14.4.4 to\_char 函数

**to\_char(expre, format)**函数用于将 timestamp、interval、integer、double precision 或者 numeric 类型的值转换为指定格式的字符串。

```
SELECT to_char(current_timestamp, 'HH24:MI:SS'),
       to_char(interval '5h 12m 30s', 'HH12:MI:SS'),
       to_char(-125.8, '999D99');
to_char |to_char |to_char|
-----|-----|-----|
21:30:22|05:12:30|-125.80|
```

其中，格式中的 9 代表数字位；D 代表小数点。关于数字的格式化选项可以参考[官方文档](#)。

#### 14.4.5 to\_number 函数

**to\_number(string, format)**函数用于将字符串转换为数字。

```
SELECT to_number('¥125.8', 'L999D9');
to_number|
-----|
125.8|
```

其中，格式字符串中的 L 表示本地货币符号。

#### 14.4.6 隐式类型转换

除了显式使用类型转换函数或运算符之外，很多时候 PostgreSQL 会自动执行数据类型的隐式转换。例如：

```
SELECT 1+'2', 'todo: '||current_timestamp;
?column?|?column? |
-----|-----|
3|todo:2020-03-09 21:49:49.370621+08|
```

## 第 15 章 子查询

子查询（Subquery）是指嵌套在其他 SELECT、INSERT、UPDATE 以及 DELETE 语句中的查询语句。

子查询的作用与多表连接查询有点类似，也是为了从多个关联的表中返回或者过滤数据。例如，我们想知道哪些员工的月薪大于平均月薪，可以通过子查询实现：

```
SELECT e.first_name, e.last_name, e.salary
FROM employees e
WHERE salary > (SELECT avg(salary) FROM employees);
first_name |last_name |salary |
-----|-----|-----|
Steven      |King      |24000.00|
Neena       |Kochhar   |17000.00|
Lex         |De Haan   |17000.00|
...
```

其中，WHERE 子句中使用了一个子查询，用于计算平均月薪。PostgreSQL 在执行以上语句时，先执行子查询返回平均月薪；然后将该值传递给外查询使用。

子查询必须位于括号中，也称为内查询，包含子查询的查询语句被称为外查询。除了 WHERE 子句之外，其他子句中也可以使用子查询，例如 SELECT 列表、FROM 子句等。

### 15.1 派生表

FROM 子句中的子查询被称为派生表（Derived table），语法如下：

```
SELECT column1, column2, ...
FROM (subquery) AS table_alias;
```

其中子查询相当于创建了一个临时表 table\_alias。以下语句用于获取每个部门的总月薪：

```
SELECT d.department_name,
       ds.sum_salary
FROM departments d
JOIN (SELECT department_id,
              SUM(salary) AS sum_salary
      FROM employees
      group by department_id) ds
ON (d.department_id = ds.department_id);
department_name |sum_salary|
-----|-----|
Administration |   4400.00|
Marketing       |  19000.00|
Purchasing      |  24900.00|
...
```

其中，子查询返回了部门编号和部门月薪合计；然后再和 departments 表进行连接查询。

## 15.2 IN 操作符

如果 **WHERE** 子查询返回多个记录，可以使用 **IN** 操作符进行条件过滤：

```
SELECT d.department_id,  
       d.department_name  
FROM departments d  
WHERE d.department_id in (SELECT department_id FROM employees WHERE  
hire_date >= date '2008-01-01');  
department_id|department_name|  
-----|-----|  
50|Shipping|  
80|Sales|
```

以上查询返回了存在 2008 年 01 月 01 日以后入职员工的部门。如果想要返回包含该日期之前入职的员工的部门，可以使用 **NOT IN** 操作符。

除了 **IN** 之外，还有一些其他进行类似过滤的操作符。

## 15.3 ALL 操作符

**ALL** 操作符与比较运算符一起使用，可以将一个值与子查询返回的列表进行比较：

```
SELECT first_name, last_name, salary  
FROM employees  
WHERE salary > all (SELECT salary FROM employees WHERE department_id = 80);  
first_name|last_name|salary|  
-----|-----|-----|  
Steven|King|24000.00|  
Neena|Kochhar|17000.00|  
Lex|De Haan|17000.00|
```

以上语句返回了月薪比销售部门（**department\_id = 80**）**所有员工**都高的员工。

其他比较运算符也可以与 **ALL** 进行组合，例如 **salary < ALL** 表示月薪比销售部门所有员工都低的员工。

## 15.4 ANY 操作符

**ANY** 操作符和 **ALL** 操作符使用方法类似，只是效果不同：

```
SELECT first_name, last_name, salary  
FROM employees  
WHERE salary > any (SELECT salary FROM employees WHERE department_id = 80);  
first_name|last_name|salary|  
-----|-----|-----|  
Steven|King|24000.00|
```

```
Neena      |Kochhar    |17000.00|
Lex        |De Haan    |17000.00|
Alexander  |Hunold     | 9000.00|
...
```

以上语句返回了月薪比销售部门（`department_id = 80`）任何员工高的员工。

`ANY` 也可以和其他比较运算符一起使用，例如 `= ANY` 实际上和 `IN` 的作用相同。

另外，`SOME` 和 `ANY` 是同义词。


## 15.5 关联子查询

以上所有示例中的子查询都可以独立运行，因为它们没有使用到外部查询中的信息。还有另一类子查询，它们会引用外部查询中的列，因而与外部查询产生关联，被称为关联子查询。

以下语句返回月薪大于所在部门平均月薪的员工：

```
SELECT first_name, last_name, salary
FROM employees o
WHERE o.salary > (SELECT avg(salary) FROM employees i WHERE i.department_id
= o.department_id);
```

我们可以看到，子查询中使用了外查询的字段（`o.department_id`）。对于外部查询中的每个员工，运行子查询返回他/她所在部门的平均月薪，然后传递给外部查询进行判断。

 关联子查询对于外查询中的每一行都会运行一次（数据库可能会对此进行优化），而非关联子查询在整个查询运行时只会执行一次。

以下语句在 `SELECT` 列表中使用关联子查询，返回每个部门的总月薪，和上文第一个示例相同：

```
SELECT d.department_name,
       (SELECT SUM(salary)
        FROM employees e
        WHERE e.department_id = d.department_id) AS sum_salary
FROM departments d
ORDER BY d.department_name;
```

## 15.6 横向子查询

一般来说，子查询只能引用外查询中的字段，而不能使用同一层级中其他表中的字段。例如：

```
-- Error case
SELECT d.department_name,
       t.avg_salary
FROM departments d
JOIN (SELECT avg(e.salary) AS avg_salary
      FROM employees e
```

```
WHERE e.department_id = d.department_id) t;  
SQL Error [42601]: ERROR: syntax error at end of input  
Position: 209
```

以上语句在 **JOIN** 中引用了左侧 **departments** 表中的字段，产生了语法错误。为此，我们需要使用横向子查询（**LATERAL subquery**）。通过增加 **LATERAL** 关键字，子查询可以引用左侧表中的列：

```
SELECT d.department_name,  
       t.sum_salary  
FROM departments d  
CROSS JOIN LATERAL (SELECT sum(e.salary) AS sum_salary  
                     FROM employees e  
                     WHERE e.department_id = d.department_id) t;
```

以上语句同样返回了每个部门的名称和总月薪。

## 15.7 EXISTS 操作符

**EXISTS** 操作符用于检查子查询结果的存在性。如果子查询返回任何结果，**EXISTS** 返回 **True**；否则，返回 **False**。

```
SELECT d.department_id,  
       d.department_name  
FROM departments d  
WHERE exists (SELECT 1 FROM employees WHERE department_id = d.department_id  
and hire_date >= date '2008-01-01');
```

以上示例返回了存在 2008 年 01 月 01 日以后入职员工的部门，与上文中的 **IN** 操作符示例相同。

**NOT EXISTS** 操作符执行相反的操作，即子查询不返回任何结果，**NOT EXISTS** 返回 **True**；否则，返回 **False**。

**[NOT] IN** 用于检查某个值是否属于（=）子查询的结果列表，**[NOT] EXISTS** 只检查子查询结果的存在性。如果子查询的结果中存在 **NULL**，**NOT EXISTS** 结果为 **True**；但是，**NOT IN** 结果为 **False**，因为 **NOT (X = NULL)** 的结果为 **NULL**。例如：

```
SELECT d.department_id,  
       d.department_name  
FROM departments d  
WHERE not exists (SELECT 1 FROM employees WHERE department_id =  
d.department_id);
```

以上语句查找没有任何员工的部门，结果返回了 16 条记录。如果使用 **NOT IN** 操作符：

```
SELECT d.department_id,  
       d.department_name  
FROM departments d  
WHERE d.department_id not in (SELECT department_id FROM employees);
```

查询没有返回任何结果，因为有一个员工不属于任何部门，导致子查询的结果中包含 **NULL**：

```
SELECT first_name, last_name, department_id
FROM employees
```

```
WHERE department_id is null;
```

```
first_name|last_name|department_id|
```

```
-----|-----|-----|
```

```
Kimberely |Grant      |          |
```

## 第 16 章 集合运算

数据库中的表（table）本质上就是由行（row）组成的集合。因此，PostgreSQL 同样支持集合论中的集合操作，包括并集（UNION）、交集（INTERSECT）和差集（EXCEPT）：

- UNION 操作符用于将两个查询结果合并成一个结果集，返回出现在第一个查询或者出现在第二个查询中的数据；
- INTERSECT 操作符用于返回两个查询结果中的共同部分，即同时出现在第一个查询结果和第二个查询结果中的数据；
- EXCEPT 操作符用于返回出现在第一个查询结果中，但不在第二个查询结果中的数据。

这三个操作符的作用如下图所示：

<table><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>2</td><td></td><td></td><td></td></tr></table>	1				1				2				<table><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>2</td><td></td><td></td><td></td></tr></table>	1				1				2				<table><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>2</td><td></td><td></td><td></td></tr></table>	1				1				2			
1																																						
1																																						
2																																						
1																																						
1																																						
2																																						
1																																						
1																																						
2																																						
<table><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>3</td><td></td><td></td><td></td></tr></table>	1				1				3				<table><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>3</td><td></td><td></td><td></td></tr></table>	1				1				3				<table><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>3</td><td></td><td></td><td></td></tr></table>	1				1				3			
1																																						
1																																						
3																																						
1																																						
1																																						
3																																						
1																																						
1																																						
3																																						
<b>t1 UNION t2</b>	<b>t1 INTERSECT t2</b>	<b>t1 EXCEPT t2</b>																																				
<table><tr><td>1</td><td></td><td></td><td></td></tr><tr><td>2</td><td></td><td></td><td></td></tr><tr><td>3</td><td></td><td></td><td></td></tr></table>	1				2				3				<table><tr><td>1</td><td></td><td></td><td></td></tr></table>	1				<table><tr><td>2</td><td></td><td></td><td></td></tr></table>	2																			
1																																						
2																																						
3																																						
1																																						
2																																						

集合操作符要求参与运算的两个查询结果具有相同数量的列，以及对列的类型必须匹配或兼容。

### 16.1 UNION

UNION 操作符用于将两个查询结果合并成一个结果集，返回出现在第一个查询或者出现在第二个查询中的数据：

```
SELECT column1, column2
FROM table1
UNION [DISTINCT | ALL]
SELECT col1, col2
FROM table2;
```

其中，DISTINCT 表示将合并后的结果集进行去重；ALL 表示保留结果集中的重复记录；如果省略，默认为 DISTINCT。例如：

```
SELECT * FROM (values(1),(2)) t1(n)
union
SELECT * FROM (values(1),(3)) t2(n);
n|
--|
1|
```



```

2|
3|

SELECT * FROM (values(1),(2)) t1(n)
union all
SELECT * FROM (values(1),(3)) t2(n);
n|
-|
1|
2|
1|
3|

```

第一个查询结果中只有一个数字 1，第二个查询结果中保留了重复的数字 1。

## 16.2 INTERSECT

**INTERSECT** 操作符用于返回两个查询结果中的共同部分，即同时出现在第一个查询结果和第二个查询结果中的数据：

```

SELECT column1, column2
FROM table1
INTERSECT [DISTINCT | ALL]
SELECT col1, col2
FROM table2;

```

其中，**DISTINCT** 表示将合并后的结果集进行去重；**ALL** 表示保留结果集中的重复记录；如果省略，默认为 **DISTINCT**。例如：

```

SELECT * FROM (values(1),(2)) t1(n)
intersect
SELECT * FROM (values(1),(3)) t2(n);
n|
-|
1|

SELECT * FROM (values(1),(1),(2)) t1(n)
intersect all
SELECT * FROM (values(1),(3)) t2(n);
n|
-|
1|

SELECT * FROM (values(1),(1),(2)) t1(n)
intersect all
SELECT * FROM (values(1),(1),(3)) t2(n);
n|
-|
1|
1|

```

第一个查询结果中只有一个数字 1；第二个查询虽然使用了 **ALL** 选项，结果也只有一个 1；第三个查询结果中有两个 1。

## 16.3 EXCEPT

**EXCEPT** 操作符用于返回出现在第一个查询结果中，但不在第二个查询结果中的数据：

```
SELECT column1, column2
  FROM table1
EXCEPT [DISTINCT | ALL]
SELECT col1, col2
  FROM table2;
```

其中，**DISTINCT** 表示将合并后的结果集进行去重；**ALL** 表示保留结果集中的重复记录；如果省略，默认为 **DISTINCT**。例如：

```
SELECT * FROM (values(1),(1),(2)) t1(n)
EXCEPT
SELECT * FROM (values(1),(3)) t2(n);
n|
-|
2|

SELECT * FROM (values(1),(1),(2)) t1(n)
EXCEPT ALL
SELECT * FROM (values(1),(3)) t2(n);
n|
-|
1|
2|
```

第一个查询结果中没有数字 1；第二个查询结果中保留了一个数字 1。

## 16.4 分组与排序

对于分组操作，集合操作符中的每个查询都可以包含一个 **GROUP BY**，不过它们只针对各自进行分组；如果想要对最终结果进行分组，需要在外层嵌套一个 **SELECT** 语句：

```
SELECT n, count(*) FROM (
  SELECT * FROM (values(1),(2)) t1(n)
  UNION ALL
  SELECT * FROM (values(1),(3)) t2(n)) t
GROUP BY n;
n|count|
-|-----|
1|    2|
2|    1|
3|    1|
```

如果要对集合运算的数据进行排序，需要将 **ORDER BY** 子句写在最后；集合操作符中的第一个查询中不能出现排序操作：

```
SELECT * FROM (values(1),(2)) t1(n)
ORDER BY n
```

```

UNION ALL
SELECT * FROM (values(1),(3)) t2(n);
SQL Error [42601]: ERROR: syntax error at or near "union"
    Position: 50

SELECT * FROM (values(1),(2)) t1(n)
UNION ALL
SELECT * FROM (values(1),(3)) t2(n)
ORDER BY n;
n|
-|
1|
1|
2|
3|

```

## 16.5 集合操作优先级

PostgreSQL 支持同时使用多个集合操作符，此时我们需要注意它们的优先级：

```

SELECT column1, column2
    FROM table1
    UNION [DISTINCT | ALL]
SELECT col1, col2
    FROM table2
INTERSECT [DISTINCT | ALL]
SELECT c1, c2
    FROM table3;

```

多个集合操作符使用以下执行顺序：

- 相同的集合操作符按照从左至右的顺序执行；
- **INTERSECT** 的优先级高于 **UNION** 和 **EXCEPT**；
- 使用括号可以修改集合操作的执行顺序。

以下示例使用了两个 **UNION** 操作符，其中一个增加了 **ALL** 选项：

```

SELECT * FROM (values(1)) t1(n)
UNION ALL
SELECT * FROM (values(1)) t2(n)
UNION
SELECT * FROM (values(1)) t3(n);
n|
-|
1|

```

查询最终的结果只有一个数字 1，因为最后的 **UNION** 去除了重复的数据。

以下示例使用了两个不同的集合操作符：

```

SELECT * FROM (values(1)) t1(n)
UNION ALL
SELECT * FROM (values(1)) t2(n)

```

```
INTERSECT
SELECT * FROM (values(1)) t3(n);
n|
-|
1|
1|
```

查询最终的结果包含了两个数字 1，因为 INTERSECT 先执行，最后的 UNION ALL 保留了重复的数据。

我们最后看一个使用括号的示例：

```
(
SELECT * FROM (values(1)) t1(n)
UNION ALL
SELECT * FROM (values(1)) t2(n)
)
INTERSECT
SELECT * FROM (values(1)) t3(n);
n|
-|
1|
```

## 第 17 篇 通用表表达式

通用表表达式（Common Table Expression、CTE）是一个临时的查询结果或者临时表，可以在其他 **SELECT**、**INSERT**、**UPDATE** 以及 **DELETE** 语句中使用。通用表表达式只在当前语句中有效，类似于子查询。

使用 CTE 的主要好处包括：

- 提高复杂查询的可读性。CTE 可以将复杂查询模块化，组织成容易理解的结构。
- 支持递归查询。CTE 通过引用自身实现递归，可以方便地处理层次结构数据和图数据。

### 17.1 简单 CTE

通用表表达式的定义如下：

```
WITH cte_name (col1, col2, ...) AS (  
    cte_query_definition  
)  
sql_statement;
```

其中，

- **WITH** 表示定义 CTE，因此 CTE 也称为 **WITH** 查询；
- **cte\_name** 指定了 CTE 的名称，后面是可选的字段名；
- 括号内是 CTE 的内容，可以是 **SELECT** 语句，也可以是 **INSERT**、**UPDATE**、**DELETE** 语句；
- **sql\_statement** 是主查询语句，可以引用前面定义的 CTE。该语句同样可以是 **SELECT**、**INSERT**、**UPDATE** 或者 **DELETE**。

PostgreSQL 中的 CTE 通常用于简化复杂的连接查询或子查询。例如：

```
WITH department_avg(department_id, avg_salary) AS (  
    SELECT department_id,  
           AVG(salary) AS avg_salary  
    FROM employees  
    GROUP BY department_id  
)  
SELECT d.department_name,  
       da.avg_salary  
FROM departments d  
JOIN department_avg da  
    ON (d.department_id = da.department_id)  
ORDER BY d.department_name;  
department_name |avg_salary      |  
-----|-----|  
Accounting      |10154.0000000000000000|  
Administration | 4400.0000000000000000|  
Executive       | 19333.333333333333333|  
...
```

首先，我们定义了一个名为 `department_avg` 的 CTE，表示每个部门的平均月薪；然后和 `departments` 表进行连接查询。虽然用其他方式也可以实现相同的功能，但是 CTE 让代码显得更加清晰易懂。

一个 **WITH** 关键字可以定义多个 CTE，而且后面的 CTE 可以引用前面的 CTE。例如：

```
WITH cte1(n) AS (  
    SELECT 1  
) ,  
cte2(m) AS (  
    SELECT n+1 FROM cte1  
)  
SELECT *  
FROM cte1, cte2;  
n|m|  
-|-|  
1|2|
```

以上示例中定义了两个 CTE，其中 `cte2` 引用了 `cte1`。最后的查询使用两者进行连接查询。

## 17.2 递归 CTE

递归 CTE 允许在它的定义中进行自引用，理论上来说可以实现任何复杂的计算功能，最常用的场景就是遍历层次结构的数据和图结构数据。

```
WITH RECURSIVE cte_name AS(  
    cte_query_initial -- 初始化部分  
    UNION [ALL]  
    cte_query_iterative -- 递归部分  
) SELECT * FROM cte_name;
```

其中，

- **RECURSIVE** 表示递归；
- `cte_query_initial` 是初始化查询，用于创建初始结果集；
- `cte_query_iterative` 是递归部分，可以引用 `cte_name`；
- 如果递归查询无法从上一次迭代中返回更多的数据，将会终止递归并返回结果。

一个经典的递归 CTE 案例就是生成数字序列：

```
WITH RECURSIVE t(n) AS (  
    VALUES (1)  
    UNION ALL  
    SELECT n+1 FROM t WHERE n < 10  
)  
SELECT n FROM t;  
n |  
--|  
1 |  
2 |  
3 |  
4 |
```

```
5|
6|
7|
8|
9|
10|
```

以上语句执行过程如下：

- 执行 CTE 中的初始化查询，生成一行数据（1）；
- 第一次执行递归查询，判断  $n < 10$  成立，生成一行数据 2（ $n+1$ ）；
- 重复执行递归查询，生成更多的数据；直到  $n = 10$  终止；此时临时表 t 中包含 10 条数据；
- 执行主查询，返回所有的数据。

**注意**，如果没有指定终止条件，上面的查询将会进入死循环。

接下来我们看一个更实用的案例，通过递归 CTE 遍历组织结构。

```
WITH RECURSIVE employee_path (employee_id, employee_name, path) AS
(
    SELECT employee_id, CONCAT(first_name, ',', last_name), CONCAT(first_name,
    ', ', last_name) AS path
    FROM employees
    WHERE manager_id IS NULL
    UNION ALL
    SELECT e.employee_id, CONCAT(e.first_name, ',', e.last_name),
    CONCAT(ep.path, '->', e.first_name, ',', e.last_name)
    FROM employee_path ep
    JOIN employees e ON ep.employee_id = e.manager_id
)
SELECT employee_name, path
FROM employee_path
ORDER BY employee_id;
employee_name |path
-----|-----
Steven,King |Steven,King
Neena,Kochhar |Steven,King->Neena,Kochhar
Lex,De Haan |Steven,King->Lex,De Haan
Alexander,Hunold |Steven,King->Lex,De Haan->Alexander,
Bruce,Ernst |Steven,King->Lex,De Haan->Alexander,Hunold->Bruce,Ernst
David,Austin |Steven,King->Lex,De Haan->Alexander,Hunold->David,Austin
...
```

其中，初始化查询语句返回了公司最高层的领导（`manager_id IS NULL`），也就是“Steven,King”；递归查询将员工表的 `manager_id` 与已有结果集中的 `employee_id` 关联，获取每个员工的下一级员工，直到无法找到新的数据；`path` 字段存储了每个员工从上至下的管理路径。

当然，我们也可以对组织结构从下至上进行遍历。更多关于递归 CTE 的实际应用场景，可以参考[这篇文章](#)。

## 17.3 DML 语句与 CTE

除了 SELECT 语句之外，INSERT、UPDATE 或者 DELETE 语句也可以与 CTE 一起使用。我们可以在 CTE 中使用 DML 语句，也可以将 CTE 用于 DML 语句。

如果在 CTE 中使用 DML 语句，我们可以将数据修改操作影响的结果作为一个临时表，然后在其他语句中使用。例如：

```
-- 创建一个员工历史表
CREATE TABLE employees_history
AS SELECT * FROM employees WHERE 1 = 0;

WITH deletes AS (
    DELETE FROM employees
    WHERE employee_id = 206
    RETURNING *
)
INSERT INTO employees_history
SELECT * FROM DELETES;

SELECT employee_id, first_name, last_name
FROM employees_history;
employee_id|first_name|last_name|
-----|-----|-----|
          206|William  |Gietz    |
```

我们首先创建了一个记录员工历史信息的历史表 `employees_history`；然后使用 DELETE 语句定义了一个 CTE，RETURNING \* 返回了被删除的数据，构成了结果集 `deletes`；然后使用 INSERT 语句记录被删除的员工信息。

接下来我们将该员工添加回员工表：

```
WITH inserts AS (
    INSERT INTO employees
    VALUES
    (206,'William','Gietz','WGIETZ','515.123.8181','2002-06-07','AC_ACCOUNT',880
    0.00,NULL,205,110)
    RETURNING *
)
INSERT INTO employees_history
SELECT * FROM inserts;
```

除了插入数据到 `employees` 表之外，我们还利用 CTE 在表 `employees_history` 中增加了一条历史记录，现在该表中有两条数据。

CTE 中的 UPDATE 语句有些不同，因为更新的数据分为更新之前的状态和更新之后的状态。例如：

```
DELETE FROM employees_history;-- 清除历史记录

WITH updates AS (
    UPDATE employees
    SET salary = salary + 500
```



```

        WHERE employee_id = 206
        RETURNING *
    )
    INSERT INTO employees_history
    SELECT * FROM employees WHERE employee_id = 206;

    SELECT employee_id, salary FROM employees_history;
    employee_id|salary |
    -----|-----|
              206|8300.00|

```

**returning** 在 CTE 中, **UPDATE** 语句修改了一个员工的月薪; 但是为了记录修改之前的数据, 我们插入 **employees\_history** 的数据仍然来自 **employees** 表。因为在一个语句中, 所有的操作都在一个事务中, 所以主查询中的 **employees** 是修改之前的状态。

如果想要获取更新之后的数据, 直接使用 **updates** 即可:

```

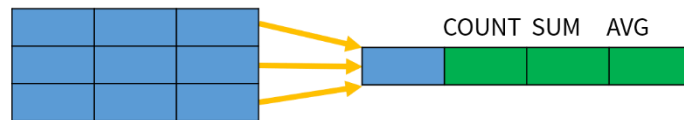
WITH updates AS (
    UPDATE employees
        set salary = salary - 500
    WHERE employee_id = 206
    RETURNING *
)
SELECT employee_id, first_name, last_name, salary
FROM updates;
employee_id|first_name|last_name|salary |
-----|-----|-----|-----|
          206|William  |Gietz    |8300.00|

```

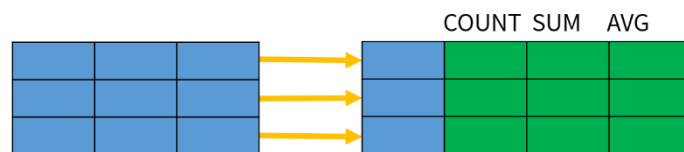
## 第 18 章 窗口函数

在第 11 章中我们学习了常见的聚合函数，包括 AVG、COUNT、MAX、MIN、SUM 以及 STRING\_AGG。聚合函数的作用是针对一组数据进行运算，并且返回一条汇总结果。

除了聚合函数之外，SQL 还定义了许多专门用于数据分析的窗口函数（Window Function）。不过，窗口函数不是将一组数据汇总为单个结果，而是针对每一行数据，基于和它相关的一组数据计算出一个结果。下图演示了聚合函数和窗口函数的区别：



聚合函数



窗口函数

以下示例分别将 AVG、COUNT、SUM 作为聚合函数和窗口函数，计算员工的平均月薪、人数总和以及月薪总和：

```
SELECT AVG(salary), COUNT(*), SUM(salary)
  FROM employees;
avg          |count|sum          |
-----|-----|-----|
6461.8317757009345794| 107|691416.00|

SELECT employee_id,first_name, last_name, AVG(salary) OVER (), COUNT(*) OVER
(), SUM(salary) OVER ()
  FROM employees;
employee_id|first_name |last_name |avg          |count|sum          |
-----|-----|-----|-----|-----|-----|
100|Steven   |King      |6461.8317757009345794| 107|691416.00|
101|Neena    |Kochhar   |6461.8317757009345794| 107|691416.00|
102|Lex      |De Haan   |6461.8317757009345794| 107|691416.00|
...
```

聚合函数通常也可以作为窗口函数，区别在于后者包含了 OVER 关键字；空括号表示将所有数据作为整体进行分析，所以得到的数值和聚合函数一样。显然，窗口函数为每一个员工都返回了一个结果。

### 18.1 窗口函数的定义

窗口函数的定义如下：

```

window_function ( expression, ... ) OVER (
    PARTITION BY ...
    ORDER BY ...
    frame_clause
)

```

其中 `window_function` 是窗口函数的名称；`expression` 是函数参数，有些函数不需要参数；`OVER` 子句包含三个选项：分区（`PARTITION BY`）、排序（`ORDER BY`）以及窗口大小（`frame_clause`）。

### 18.1.1 分区选项（PARTITION BY）

`PARTITION BY` 选项用于定义分区，作用类似于 `GROUP BY` 的分组。如果指定了分区选项，窗口函数将会分别针对每个分区单独进行分析；如果省略分区选项，所有的数据作为一个整体进行分析，上文中的示例就是如此。

以下语句按照部门进行分组，分析每个部门的平均月薪：

```

SELECT first_name, last_name, department_id, salary, AVG(salary) OVER
(PARTITION BY department_id)
FROM employees
ORDER BY department_id;

```

first_name	last_name	department_id	salary	avg
Jennifer	Whalen	10	4400.00	4400.0000000000000000
Pat	Fay	20	6000.00	9500.0000000000000000
Michael	Hartstein	20	13000.00	9500.0000000000000000
Shelli	Baida	30	2900.00	4150.0000000000000000
Karen	Colmenares	30	2500.00	4150.0000000000000000
Den	Raphaely	30	11000.00	4150.0000000000000000
...				

部门 10 只有一个员工，平均月薪就是她自己的月薪 4400；部门 20 有两个员工，平均月薪等于  $(6000 + 13000)/2 = 9500$ ；其他数据依次类推。

### 18.1.2 排序选项（ORDER BY）

`ORDER BY` 选项用于指定分区内的排序方式，通常用于数据的排名分析。以下示例用于计算每个员工在部门内的入职顺序：

```

SELECT first_name, last_name, department_id, hire_date,
       RANK() OVER (PARTITION BY department_id ORDER BY hire_date)
FROM employees
ORDER BY department_id;

```

first_name	last_name	department_id	hire_date	rank
Jennifer	Whalen	10	2003-09-17	1
Michael	Hartstein	20	2004-02-17	1
Pat	Fay	20	2005-08-17	2
Den	Raphaely	30	2002-12-07	1
Alexander	Khoo	30	2003-05-18	2
Sigal	Tobias	30	2005-07-24	3
...				

其中，`PARTITION BY` 选项表示按照部门进行分区；`ORDER BY` 选项指定在部门内按照入

职先后进行排序；RANK 函数用于计算名次，下文将会进行介绍。

部门 10 只有一个员工，Jennifer 就是第一个入职的员工；部门 20 有两个员工，Michael（2004-02-17）比 Pat（2005-08-17）入职更早；其他数据依次类推。

 ORDER BY 子句同样支持 NULLS FIRST 和 NULLS LAST 选项，用于指定空值的排序顺序。默认为 NULLS LAST。

### 18.1.3 窗口选项（frame\_clause）

frame\_clause 选项用于在**当前分区**内指定一个计算窗口。指定了窗口之后，分析函数不再基于分区进行计算，而是基于窗口内的数据进行计算。以下示例用于计算每个产品当当前月份的累计销量（[示例数据](#)）：

```
SELECT product AS "产品", ym "年月", amount "销量",
       SUM(amount) OVER (PARTITION BY product ORDER BY ym ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW)
  FROM sales_monthly
 ORDER BY product, ym;
```

产品	年月	销量	sum
桔子	201801	10154.00	10154.00
桔子	201802	10183.00	20337.00
桔子	201803	10245.00	30582.00
桔子	201804	10325.00	40907.00
桔子	201805	10465.00	51372.00
桔子	201806	10505.00	61877.00

其中，PARTITION BY 选项表示按照产品进行分区；ORDER BY 选项表示按照日期进行排序；窗口子句 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW 指定窗口从当前分区的第一行开始到当前行结束；因此 SUM 函数计算的是产品累计到当前月份为止的销量合计。

具体来说，窗口大小的常用选项如下：

```
{ ROWS | RANGE } frame_start
{ ROWS | RANGE } BETWEEN frame_start AND frame_end
```

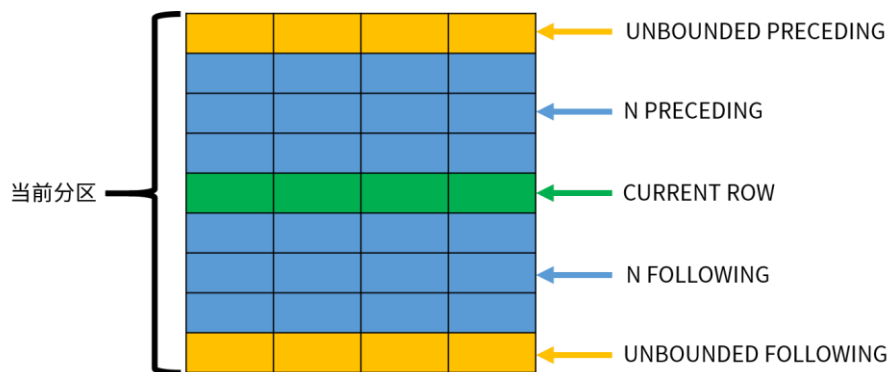
其中，ROWS 表示以行为单位计算窗口的偏移量，RANGE 表示以数值（例如 30 分钟）为单位计算窗口的偏移量。其中，frame\_start 用于定义窗口的起始位置，可以指定以下内容之一：

- UNBOUNDED PRECEDING，窗口从分区的第一行开始，默认值；
- N PRECEDING，窗口从当前行之前的第 N 行或者数值开始；
- CURRENT ROW，窗口从当前行开始。

frame\_end 用于定义窗口的结束位置，可以指定以下内容之一：

- CURRENT ROW，窗口到当前行结束，默认值；
- N FOLLOWING，窗口到当前行之后的第 N 行或者数值结束；
- UNBOUNDED FOLLOWING，窗口到分区的最后一行结束。

下图可以方便我们理解这些选项的含义：



**CURRENT ROW** 表示当前正在处理的行，其他的行可以使用相对当前行的位置表示。需要注意，窗口的大小不会超出当前分区的范围。

PostgreSQL 还提供了更多复杂的窗口选项，可以参考[官方文档](#)。

常见的窗口函数可以分为以下几类：聚合窗口函数、排名窗口函数以及取值窗口函数。

## 18.2 聚合窗口函数

常用的聚合函数，例如 **AVG**、**SUM**、**COUNT** 等，也可以作为窗口函数使用。上文我们已经列举了一些聚合窗口函数的示例，再来看一个使用 **AVG** 函数计算移动平均值的例子：

```
SELECT saledate, amount, avg(amount) OVER (ORDER BY saledate ROWS BETWEEN 1
PRECEDING AND 1 FOLLOWING)
FROM sales_data
WHERE product = '桔子' AND channel = '淘宝';
saledate |amount |avg
-----|-----|-----
2019-01-01|1864.00|1893.5000000000000000
2019-01-02|1923.00|1505.3333333333333333
2019-01-03| 729.00|1066.3333333333333333
2019-01-04| 547.00| 966.6666666666666667
2019-01-05|1624.00|1272.0000000000000000
2019-01-06|1645.00|1332.0000000000000000
...
```

该语句返回了“桔子”在“淘宝”上的销量，以及每一天和它前后一天（共 3 天）的平均销量。

移动平均值通常用于处理时间序列的数据。例如，厂房的温度检测器获取了每秒钟的温度，我们可以使用以下窗口计算前五分钟内的平均温度：

```
avg(temperature) OVER (ORDER BY ts RANGE BETWEEN interval '5 minute' PRECEDING
AND CURRENT ROW)
```

## 18.3 排名窗口函数

排名窗口函数用于对数据进行分组排名。常见的排名窗口函数包括：

- **ROW\_NUMBER**，为分区中的每行数据分配一个序列号，序列号从 1 开始分配。
- **RANK**，计算每行数据在其分区中的名次；如果存在名次相同的数据，后续的排名将会产生跳跃。
- **DENSE\_RANK**，计算每行数据在其分区中的名次；即使存在名次相同的数据，后续的排名也是连续的值。
- **PERCENT\_RANK**，以百分比的形式显示每行数据在其分区中的名次；如果存在名次相同的数据，后续的排名将会产生跳跃。
- **CUME\_DIST**，计算每行数据在其分区内的累积分布，也就是该行数据及其之前的数据的比率；取值范围大于 0 并且小于等于 1。
- **NTILE**，将分区内的数据分为 N 等份，为每行数据计算其所在的位置。

排名窗口函数不支持动态的窗口大小（`frame_clause`），而是以当前分区作为分析的窗口。

以下示例按照部门为单位，计算员工的月薪排名：

```
SELECT d.department_name "部门名称", concat(e.first_name, ',' , e.last_name)
"姓名", e.salary "月薪",
       ROW_NUMBER() OVER (PARTITION BY e.department_id ORDER BY e.salary DESC)
AS "row_number",
       RANK() OVER (PARTITION BY e.department_id ORDER BY e.salary DESC) AS
"rank",
       DENSE_RANK() OVER (PARTITION BY e.department_id ORDER BY e.salary DESC)
AS "dense_rank",
       PERCENT_RANK() OVER (PARTITION BY e.department_id ORDER BY e.salary DESC)
AS "percent_rank"
FROM employees e
JOIN departments d ON (e.department_id = d.department_id)
WHERE d.department_name in ('IT', 'Purchasing')
ORDER BY 1, 4;
```

部门名称	姓名	月薪	row_number	rank	dense_rank	percent_rank
IT	Alexander,Hunold	9000.00	1	1	1	0
IT	Bruce,Ernst	6000.00	2	2	2	0.25
IT	Valli,Pataballa	4800.00	3	3	3	0.5
IT	David,Austin	4800.00	4	3	3	0.5
IT	Diana,Lorentz	4200.00	5	5	4	1
Purchasing	Den,Raphaely	11000.00	1	1	1	0
Purchasing	Alexander,Khoo	3100.00	2	2	2	0.2
Purchasing	Shelli,Baida	2900.00	3	3	3	0.4
Purchasing	Sigal,Tobias	2800.00	4	4	4	0.6
Purchasing	Guy,Himuro	2600.00	5	5	5	0.8
Purchasing	Karen,Colmenares	2500.00	6	6	6	1

**ROW\_NUMBER** 函数为每个员工分配了一个连续的数字编号，可以看作是一种排名。IT 部门的“Valli,Pataballa”和“David,Austin”的月薪相同，但是编号不同：

**RANK** 函数为每个员工指定了一个名次，IT 部门的“Valli,Pataballa”和“David,Austin”的名次

都是 3；而在他们之后的“Diana,Lorentz”的名次为 5，产生了跳跃；


DENSE\_RANK 函数为每个员工指定了一个名次,IT 部门的“Valli,Pataballa”和“David,Austin”的名次都是 3；在他们之后的“Diana,Lorentz”的名次为 4，名次是连续值；

PERCENT\_RANK 函数按照百分比指定名次，取值位于 0 到 1 之间。其中“Diana,Lorentz”的百分比排名为 1，也产生了跳跃。

以上示例中 4 个窗口函数的 OVER 子句完全相同，此时可以采用一种更简单的写法：

```
SELECT d.department_name "部门名称", concat(e.first_name, ', ' , e.last_name)
"姓名", e.salary "月薪",
       ROW_NUMBER() OVER w AS "row_number",
       RANK() OVER w AS "rank",
       DENSE_RANK() w AS "dense_rank",
       PERCENT_RANK() w AS "percent_rank"
FROM employees e
JOIN departments d ON (e.department_id = d.department_id)
WHERE d.department_name in ('IT', 'Purchasing')
WINDOW w AS (PARTITION BY e.department_id ORDER BY e.salary DESC)
ORDER BY 1, 4;
```

其中，WINDOW 定义了一个窗口变量 w，然后在窗口函数的 OVER 子句中使用了该变量；这样可以简化函数的输入。

 窗口函数在 GROUP BY 分组、聚合函数以及 HAVING 过滤之后运行。如果多个窗口函数拥有相同的 PARTITION BY 和 ORDER BY 选项，它们会在遍历数据时一起进行计算，也就是说它们读取输入数据的顺序完全一致。

以下语句演示了 CUME\_DIST 和 NTILE 函数的作用：

```
SELECT concat(first_name, ', ' , last_name) "姓名", hire_date AS "入职日期",
       CUME_DIST() OVER (ORDER BY hire_date) AS "累积占比",
       NTILE(100) OVER (ORDER BY hire_date) AS "相对位置"
FROM employees;
```

姓名	入职日期	累积占比	相对位置
Lex,De Haan	2001-01-13	0.009345794392523364	1
Hermann,Baer	2002-06-07	0.04672897196261682	1
Shelley,Higgins	2002-06-07	0.04672897196261682	2
William,Gietz	2002-06-07	0.04672897196261682	2
Susan,Mavris	2002-06-07	0.04672897196261682	3
Daniel,Faviet	2002-08-16	0.056074766355140186	3
...			

其中，CUME\_DIST 函数显示 2001-01-13 以及之前入职的员工大概有 0.9%（1/107）；NTILE(100)函数表明前 1%入职的员工有“Lex,De Haan”和“Hermann,Baer”，由于员工总数为 107，所以不是完全准确。

## 18.4 取值窗口函数

取值窗口函数用于返回指定位置上的数据。常见的取值窗口函数包括：

- **FIRST\_VALUE**, 返回窗口内第一行的数据。
- **LAST\_VALUE**, 返回窗口内最后一行的数据。
- **NTH\_VALUE**, 返回窗口内第 N 行的数据。
- **LAG**, 返回分区中当前行之前的第 N 行的数据。
- **LEAD**, 返回分区中当前行之后的第 N 行的数据。

其中，LAG 和 LEAD 函数不支持动态的窗口大小（frame\_clause），而是以当前分区作为分析的窗口。

以下语句使用 **FIRST\_VALUE**、**LAST\_VALUE** 以及 **NTH** 函数分别获取每个部门内部月薪最高、月薪最低以及月薪第三高的员工：

```

SELECT department_id, first_name, last_name, salary,
       FIRST_VALUE(salary) OVER w,
       LAST_VALUE(salary) OVER w,
       NTH_VALUE(salary, 3) OVER w
FROM employees
WINDOW w AS (PARTITION BY department_id ORDER BY salary desc ROWS BETWEEN
UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
ORDER BY department_id, salary DESC;
department_id|first_name|last_name|salary|first_value|last_value|nth_valu
e|
-----|-----|-----|-----|-----|-----|-----
---|
10|Jennifer |Whalen    | 4400.00| 4400.00| 4400.00|
10|Michael  |Hartstein|13000.00|13000.00| 6000.00|
10|Pat       |Fay      | 6000.00|13000.00| 6000.00|
10|Den       |Raphaely |11000.00|11000.00| 2500.00| 2900.00|
10|Alexander|Khoo     | 3100.00|11000.00| 2500.00| 2900.00|
10|Shelli   |Baida    | 2900.00|11000.00| 2500.00| 2900.00|
10|Sigal    |Tobias   | 2800.00|11000.00| 2500.00| 2900.00|
10|Guy      |Himuro   | 2600.00|11000.00| 2500.00| 2900.00|
10|Karen    |Colmenares| 2500.00|11000.00| 2500.00| 2900.00|
...

```

以上三个函数的默认窗口是从当前分区的第一行到当前行，所以我们在 **OVER** 子句中将窗口设置为整个分区。

LAG 和 LEAD 函数同样用于计算销量数据的环比/同比增长。例如，以下语句统计不同产品每个月的环比增长率：

```
WITH sales_monthly AS (
    SELECT product, to_char(saledate, 'YYYYMM') ym, sum(amount) sum_amount
    FROM sales_data
    GROUP BY product, to_char(saledate, 'YYYYMM')
)
SELECT product AS "产品", ym "年月", sum_amount "销量",
       (sum_amount - LAG(sum_amount, 1) OVER (PARTITION BY product ORDER BY
ym)) /
       LAG(sum_amount, 1) OVER (PARTITION BY product ORDER BY ym) * 100 AS "
环比增长率(%)"
FROM sales_monthly
ORDER BY product, ym;
产品|年月|销量|环比增长率(%)|
```



```
----|-----|-----|-----|
桔子|201901|126083.00|          |
桔子|201902|119417.00|-5.28699348841636065100|
桔子|201903|147290.00|23.34089786211343443600|
桔子|201904|147848.00| 0.37884445651435942700|
桔子|201905|182417.00|23.38144580920945836300|
桔子|201906|186206.00| 2.07710904137224052600|
...
```

首先，创建一个通用表表达式 `sales_monthly`，得到了不同产品每个月的销量汇总；`LAG(sum_amount, 1)`表示获取上一期的销量；当前月份的销量减去上个月的销量，再除以上个月的销量，就是环比增长率。

## 第 19 章 DML 语句

本篇介绍如何对表中的数据进行修改操作，包括插入数据的 `INSERT` 语句、更新数据的 `UPDATE` 语句、删除数据的 `DELETE` 语句，以及合并数据的 `INSERT ON CONFLICT` 语句。

我们首先创建两个示例表：

```
CREATE TABLE dept (  
    department_id int NOT NULL,  
    department_name varchar(30) NOT NULL,  
    CONSTRAINT dept_pkey PRIMARY KEY (department_id)  
);  
  
CREATE TABLE emp (  
    employee_id int NOT NULL,  
    first_name varchar(20) NULL,  
    last_name varchar(25) NOT NULL,  
    hire_date date not null default current_date,  
    salary numeric(8,2) NULL,  
    manager_id int NULL,  
    department_id int NULL,  
    CONSTRAINT emp_pkey PRIMARY KEY (employee_id),  
    CONSTRAINT fk_emp_dept FOREIGN KEY (department_id) REFERENCES  
dept(department_id) ON DELETE CASCADE,  
    CONSTRAINT fk_emp_manager FOREIGN KEY (manager_id) REFERENCES  
emp(employee_id)  
);
```

关于创建表的 `CREATE TABLE` 语句，可以参考第 4 章。

### 19.1 插入数据

PostgreSQL 提供了 `INSERT` 语句，可以用于插入一行或者多行数据。

#### 19.1.1 插入单行数据

`INSERT` 语句的简单形式如下：

```
INSERT INTO table_name(column1, column2, ...)  
VALUES (value1, value2, ...);
```

其中，`value1` 是 `column1` 的值，`value2` 是 `column2` 的值。例如：

```
INSERT INTO dept(department_id, department_name) VALUES ( 10,  
'Administration');  
  
SELECT * FROM dept;  
department_id|department_name|  
-----|-----|  
10|Administration |
```

如果 `VALUES` 列表为所有字段都指定了值，并且按照表的字段顺序出现，可以省略表名后

的字段列表。因此，我们也可以使用以下插入语句：

```
INSERT INTO dept VALUES ( 20, 'Marketing');

SELECT * FROM dept;
department_id|department_name|
-----|-----|
          10|Administration |
          20|Marketing      |
```

指定字段的值也可以使用 **DEFAULT**，表示使用定义字段时的默认值；如果没有指定默认值使用 **NULL**。

### 19.1.2 插入多行数据

PostgreSQL 中的 **INSERT** 语句支持一次插入多行数据，在 **VALUES** 之后使用逗号进行分隔。例如：

```
INSERT INTO emp
VALUES (200, 'Jennifer', 'Whalen', '2020-01-01', 4400.00, NULL, 10),
      (201, 'Michael', 'Hartstein', '2020-02-02', 13000.00, NULL, 20),
      (202, 'Pat', 'Fay', default, 6000.00, 201, 20);

SELECT * FROM emp;
employee_id|first_name|last_name|hire_date|salary|
manager_id|department_id|
-----|-----|-----|-----|-----|-----|-----|
----|
          200|Jennifer |Whalen  |2020-01-01| 4400.00|      |      10|
          201|Michael  |Hartstein|2020-02-02|13000.00|      |      20|
          202|Pat      |Fay      |2020-04-14| 6000.00|    201|      20|
```

以上语句一次增加了 3 名员工信息，日期可以使用字符形式的字面值（‘2020-01-01’），**default** 表示使用默认的当前日期。

### 19.1.3 复制数据

**INSERT INTO SELECT** 语句可以将一个查询语句的结果插入表中。例如：

```
create table emp1 (like emp);

INSERT INTO emp1
SELECT * FROM emp
WHERE department_id = 20;

SELECT * FROM emp1;
employee_id|first_name|last_name|hire_date|salary|
manager_id|department_id|
-----|-----|-----|-----|-----|-----|-----|
----|
          201|Michael  |Hartstein|2020-02-02|13000.00|      |      20|
          202|Pat      |Fay      |2020-04-14| 6000.00|    201|      20|
```

我们首先基于 **emp** 创建了一个新表 **emp1**，然后通过查询语句将 **emp** 中的部分数据复制到 **emp1** 中。

### 19.1.4 返回插入的数据

PostgreSQL 对 SQL 标准进行了扩展，可以在 INSERT 语句之后使用 RETURNING 返回插入的数据值。例如：

```
INSERT INTO dept
values (30, 'Purchasing')
RETURNING department_id;
department_id|
-----|
              30|
```

以上语句除了插入一条数据到 dept 表中之外，同时还返回了该数据的 department\_id。

## 19.2 更新数据

### 19.2.1 单表更新

PostgreSQL 使用 UPDATE 语句更新表中已有的数据，基本的语法如下：

```
UPDATE table_name
    SET column1 = value1,
        column2 = value2,
        ...
WHERE conditions;
```

其中，WHERE 决定了需要更新的数据行，只有满足条件的数据才会更新；如果省略 WHERE 条件，将会更新表中的所有数据，需要谨慎使用。

以下语句将编号为 200 的员工从原部门调动到 Marketing，并且涨薪 1000：

```
UPDATE emp
    SET salary = salary + 1000,
        department_id = 20
WHERE employee_id = 200;
```

### 19.2.2 跨表更新

除了以上形式的更新语句之外，PostgreSQL 还支持通过关联其他表中的数据进行更新。以下语句利用 emp 中的数据更新 emp1 表：

```
UPDATE emp1
    SET salary = emp.salary,
        department_id = emp.department_id,
        manager_id = emp.manager_id
FROM emp
WHERE emp1.employee_id = emp.employee_id;
```

我们使用 FROM 子句访问 emp 中的数据，并且在 WHERE 子句中指定了两个表的关联条件。这种语句与多表连接查询（JOIN）类似，有时候也称为多表连接更新（UPDATE JOIN）。

### 19.2.3 返回更新后的数据

PostgreSQL 同样对 UPDATE 语句进行了扩展,支持使用 RETURNING 返回更新后的数据值。例如:

```
UPDATE emp
set salary = salary + 1000,
    department_id = 20
WHERE employee_id = 200
RETURNING first_name, last_name, salary;
first_name|last_name|salary |
-----|-----|-----|
Jennifer  |Whalen   |6400.00|
```

我们再次更新编号为 200 的员工的信息,并且返回了更新之后的记录。

## 19.3 删除数据

### 19.3.1 单表删除

删除数据可以使用 DELETE 语句:

```
DELETE FROM table_name
WHERE conditions;
```

同样,只有满足 WHERE 条件的数据才会被删除;如果省略,将会删除表中所有的数据。

以下语句用于删除 emp1 中员工编号为 201 的数据:

```
DELETE
FROM emp1
WHERE employee_id = 201;
```

如果没有编号为 201 的记录,不会删除任何数据。

### 19.3.2 跨表删除

PostgreSQL 同样支持通过关联其他表进行数据删除。以下语句利用 emp 表删除 emp1 表中的数据:

```
DELETE
FROM emp1
USING emp
WHERE emp1.employee_id = emp.employee_id;
```

注意,跨表删除使用 USING 关键字引用其他的表,而不是 JOIN。以上语句了 emp1 中员工编号存在于 emp 表中的数据,等价于以下子查询实现:

```
DELETE
FROM emp1
WHERE emp1.employee_id in (SELECT employee_id FROM emp);
```

### 19.3.3 返回被删除的数据

PostgreSQL 中的 `DELETE` 语句也可以使用 `RETURNING` 返回被删除的数据。例如：

```
-- 先插入一些数据
INSERT INTO emp1
SELECT * FROM emp
WHERE department_id = 20;

DELETE
FROM emp1
RETURNING *;
employee_id|first_name|last_name|hire_date|salary|manager_id|department_id|
-----|-----|-----|-----|-----|-----|-----
-|
          201|Michael  |Hartstein|2020-02-02|13000.00|          |      20|
          202|Pat      |Fay      |2020-04-14| 6000.00|         201|      20|
          200|Jennifer |Whalen   |2020-01-01| 6400.00|          |      20|
```

我们先从 `emp` 复制了一些数据到 `emp1` 中，然后删除所有数据并且返回这些记录。

## 19.4 合并数据

### 19.4.1 MERGE 语句

SQL 标准中定义了一个用于合并数据的 `MERGE` 语句，可以基于指定条件执行插入、更新或者删除操作。PostgreSQL 15 实现了 `MERGE` 语句，基本语法如下：

```
MERGE INTO target
USING source
ON join_condition
{ WHEN MATCHED THEN { UPDATE | DELETE | DO NOTHING } |
  WHEN NOT MATCHED THEN { INSERT | DO NOTHING } }
;
```

其中 `target` 是合并操作的目标表；`source` 是合并数据的来源，可以是表名或者查询语句；`ON` 子句是判断源数据在目标表中是否存在的条件；`WHEN MATCHED THEN` 分支指定了数据匹配（已经存在）时执行的操作；`WHEN NOT MATCHED THEN` 分支指定了数据不存在时的操作。

我们创建一个示例表 `account`：

```
CREATE TABLE account (
    id INTEGER PRIMARY KEY,
    balance NUMERIC NOT NULL,
    status VARCHAR(1) NOT NULL CHECK (status IN ('Y', 'N'))
);
```

使用以下语句为 `account` 表新增一条记录：

```
MERGE INTO account a
USING (VALUES(1, 0, 'Y')) s(id, balance, status)
ON a.id = s.id
```

```

WHEN MATCHED THEN
    UPDATE SET balance = s.balance, status = s.status
WHEN NOT MATCHED THEN
    INSERT (id, balance, status)
    VALUES (s.id, s.balance, s.status);

SELECT * FROM account;
id|balance|status|
--|-----|-----|
1|      0|Y      |

```

由于 id 等于 1 的记录不存在，以上语句将会执行 **WHEN NOT MATCHED THEN** 分支，插入一条新的记录。

接下来我们将插入源数据中的 **balance** 修改为 100，再次执行 **MERGE** 语句：

```

MERGE INTO account a
USING (VALUES(1, 100, 'Y')) s(id, balance, status)
ON a.id = s.id
WHEN MATCHED THEN
    UPDATE SET balance = s.balance, status = s.status
WHEN NOT MATCHED THEN
    INSERT (id, balance, status)
    VALUES (s.id, s.balance, s.status);

SELECT * FROM account;
id|balance|status|
--|-----|-----|
1|     100|Y      |

```

以上语句将会执行 **WHEN MATCHED THEN** 分支，更新 **account** 表中 id 等于 1 的记录。

最后，我们在 **MERGE** 语句中增加一个分支，用于删除数据：

```

MERGE INTO account a
USING (VALUES(1, 100, 'N')) s(id, balance, status)
ON a.id = s.id
WHEN MATCHED AND s.status = 'N' THEN
    DELETE
WHEN MATCHED THEN
    UPDATE SET balance = s.balance, status = s.status
WHEN NOT MATCHED THEN
    INSERT (id, balance, status)
    VALUES (s.id, s.balance, s.status);

SELECT * FROM account;
id|balance|status|
--|-----|-----|

```

语句中的 **WHEN MATCHED AND s.status = 'N' THEN** 表示如果源数据存在，并且源数据中的状态为 N，则删除目标表中的对应记录。因此，最后的查询语句没有返回结果。

## 19.4.2 INSERT ON CONFLICT 语句

对于 PostgreSQL 14 以及更早版本，可以通过 **INSERT INTO ... ON CONFLICT...** 实现数据合并的功能。

```
INSERT INTO table_name(column1, column2, ...)
{VALUES (value1, value2, ...) | SELECT ...}
ON CONFLICT conflict_target conflict_action;
```

其中，`conflict_target` 是判断数据是否已经存在的条件：

- `( { index_column_name | ( index_expression ) } )`，基于某个具有索引的字段或者表达式进行判断；
- `ON CONSTRAINT constraint_name`，基于某个唯一约束进行判断。

`conflict_action` 表示冲突时采取的操作：

- `DO NOTHING`，如果数据已经存在，不做任何操作；
- `DO UPDATE SET`，如果数据已经存在，更新该数据；可以使用 `WHERE` 子句进一步限制需要更新的数据。

这种语句通过为 `INSERT` 语句增加 `ON CONFLICT` 选项，组合了 `INSERT` 和 `UPDATE` 语句的功能，因此也被称为 `UPSERT` 语句。

`emp` 表中已经存在编号为 200 的员工，如果我们再次插入该编号将会提示主键冲突：

```
INSERT INTO emp
values (200, 'Jennifer', 'Whalen', '2020-01-01', 4400.00, NULL, 10)
SQL Error [23505]: ERROR: duplicate key value violates unique constraint
"emp_pkey"
Detail: Key (employee_id)=(200) already exists.
```

此时，我们可以增加冲突处理，从而避免语句出错：

```
INSERT INTO emp
values (200, 'Jennifer', 'Whalen', '2020-01-01', 4400.00, NULL, 10)
on conflict (employee_id)
do nothing;
```

以上语句基于 `employee_id` 字段是否重复进行判断，冲突时不做任何处理。

另一种处理冲突的方式就是进行数据更新：

```
SELECT department_id
FROM emp
WHERE employee_id = 200;
department_id|
-----|
20|

INSERT INTO emp
values (200, 'Jennifer', 'Whalen', '2020-01-01', 4400.00, NULL, 10)
on conflict on constraint emp_pkey
do update
set first_name = EXCLUDED.first_name,
    last_name = EXCLUDED.last_name,
    hire_date = EXCLUDED.hire_date,
    salary = EXCLUDED.salary,
    manager_id = EXCLUDED.manager_id,
    department_id = EXCLUDED.department_id;

SELECT *
```



```

FROM emp
WHERE employee_id = 200;
employee_id|first_name|last_name|hire_date|salary|manager_id|department_i
d|
-----|-----|-----|-----|-----|-----|-----
-|
                200|Jennifer  |Whalen   |2020-01-01|4400.00|          |10|

```

该员工的部门编号在前面被修改为 20；我们通过主键约束 `emp_pkey` 进行重复数据的判断，然后更新该员工的数据；`EXCLUDED` 是一个特殊的表，代表了原本应该插入的数据行；最终该员工的部门编号被更新为 10。

## 第 20 章 事务与并发控制

### 20.1 数据库事务

数据库事务是由一个或者多个操作组成的工作单元。一个经典事务示例就是银行账户之间的转账，它由发起方的扣款操作和接收方入账操作组成，两者必须都成功或者都失败。

数据库中的事务具有原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）以及持久性（Durability），也就是 ACID 属性：

- **原子性**保证事务中的操作要么全部成功，要么全部失败，不会只成功一部分。比如从 A 账户转出 1000 元到 B 账户，如果从 A 账户减去 1000 元成功执行，但是没有往 B 账户增加 1000 元，意味着客户将会损失 1000 元。用数据库中的术语来说，这种情况导致了数据库的不一致性。
- **一致性**确保了数据修改的有效性，并且遵循一定的业务规则；例如，上面的银行转账事务中如果一个账户扣款成功，但是另一个账户加钱失败，那么就会出现数据不一致（此时需要回滚已经执行的扣款操作）。另外，数据库还必须保证满足完整性约束，比如账户扣款之后不能出现余额为负数（可以在余额字段上添加检查约束）。
- **隔离性**决定了并发事务之间的可见性和相互影响程度。例如，账户 A 向账户 B 转账的过程中，账户 B 查询的余额应该是转账之前的数目；如果多人同时向账户 B 转账，结果也应该保持一致性，就像依次转账的结果一样。SQL 标准定义了 4 种不同的隔离级别，具体参考下文。
- **持久性**确保已经提交的事务必须永久生效，即使发生断电、系统崩溃等故障，数据库都不会丢失数据。对于 PostgreSQL 而言，使用的是预写式日志（WAL）的机制实现事务的持久性。

### 20.2 事务控制语句

我们先来介绍一下 PostgreSQL 提供的事务控制语句，执行以下命令创建示例表：

```
CREATE TABLE accounts(  
    id serial PRIMARY KEY,  
    user_name varchar(50),  
    balance numeric(10,4)  
);  
  
ALTER TABLE accounts ADD CONSTRAINT bal_check CHECK(balance >= 0);
```

accounts 是一个简化的账户表，主要包含用户名和余额信息。我们为该表插入一条记录：

```
INSERT INTO accounts(user_name, balance)  
values ('UserA', 6000);
```

```
SELECT * FROM accounts;
id|user_name|balance |
--|-----|-----|
1|UserA    |6000.0000|
```

默认情况下, PostgreSQL 自动为以上 INSERT 语句开始一个事务, 执行插入操作之后自动提交该事务。

不过, 我们也可以手动控制事务的开始和提交。例如:

```
begin;

INSERT INTO accounts(user_name, balance)
values ('UserB', 0);

SELECT * FROM accounts;
id|user_name|balance |
--|-----|-----|
1|UserA    |6000.0000|
2|UserB    | 0.0000|
```

其中, BEGIN 用于开始一个新的事务, PostgreSQL 中也可以使用 BEGIN WORK 或者 BEGIN TRANSACTION 开始事务; 然后插入一条记录, 查询显示了两条记录。

如果此时打开另一个数据库连接, 查询 accounts 表只能看到一条记录。因为上面的事务还没有提交, 事务的隔离性使得我们无法看到其他事务未提交的修改。

我们将上面的事务进行提交:

```
commit;
```

COMMIT 用于提交事务, 也可以使用 COMMIT WORK 或者 COMMIT TRANSACTION。此时, 其他事务就能看到用户 UserB 的记录了。

事务除了可以提交之外, 也可以被回滚。我们演示一下如何回滚事务:

```
begin;

INSERT INTO accounts(user_name, balance)
values ('UserC', 2000);

SELECT * FROM accounts;
id|user_name|balance |
--|-----|-----|
1|UserA    |6000.0000|
2|UserB    | 0.0000|
3|UserC    |2000.0000|
```

开始事务之后, 我们又新增了一个账户没有提交; 此时可以回滚该事务:

```
rollback;

SELECT * FROM accounts;
id|user_name|balance |
--|-----|-----|
1|UserA    |6000.0000|
2|UserB    | 0.0000|
```

ROLLBACK 用于回滚当前事务，也可以使用 ROLLBACK WORK 或者 ROLLBACK TRANSACTION。回滚之后，事务中的数据修改都会被撤销，账户 UserC 并没有创建成功。

还有一个与事务控制相关的语句：SAVEPOINT，用于在事务中定义保存点。例如：

```
begin;

INSERT INTO accounts(user_name, balance)
values ('UserC', 2000);

savepoint sv1;

INSERT INTO accounts(user_name, balance)
values ('UserD', 0);

rollback to sv1;

commit;

SELECT * FROM accounts;
id|user_name|balance |
--|-----|-----|
1|UserA    |6000.0000|
2|UserB    |  0.0000|
4|UserC    |2000.0000|
```

开始一个事务之后，先插入账户 UserC，然后定义了保存点 sv1；接着插入账户 UserD，然后回滚到保存点 sv1；此时账户 UserD 被撤销，账户 UserC 仍然存在；最后提交事务。

## 20.3 并发与隔离

PostgreSQL 支持多用户并发访问，并且保证多个用户同时访问相同的数据时不会造成数据的不一致性。当多个用户同时访问相同的数据时，如果不进行任何隔离控制，可能导致以下问题：

- **脏读**（dirty read），一个事务能够读取其他事务未提交的修改。例如，B 的初始余额为 0；A 向 B 转账 1000 元但没有提交；此时 B 能够看到 A 转过来的 1000 元，并且成功取款 1000 元；然后 A 取消了转账；银行损失了 1000 元。
- **不可重复读**（nonrepeatable read），一个事务读取某个记录后，再次读取该记录时数据发生了改变（被其他事务修改并提交）。例如，B 查询初始余额为 1000，取款 1000；同时 A 向 B 转账 1000 元并且提交；B 再次查询发现余额还是 1000 元，以为取款机出错了（当然，通过查询流水记录可以发现真相；数据库的状态仍然是一致的）。
- **幻读**（phantom read），一个事务按照某个条件查询一些数据后，再次执行相同查询时结果的数量发生了变化（另一个事务增加或者删除了某些数据并且完成提交）。幻读和非重复读有点类似，都是由于其他事务修改数据导致的结果变化。
- **更新丢失**（lost update），第一类：当两个事务更新相同的数据时，第一个事务被提交，然后第二个事务被撤销；那么第一个事务的更新也会被撤销（所有隔离级别都不允许发生这种情况）。第二类：当两个事务同时读取某一记录，然后分别进行修改提交；就会造成先提交的事务的修改丢失。

为了解决并发问题，SQL 标准定义了 4 种不同的事务隔离级别（从低到高）：

隔离级别	脏读	不可重复读	幻读	更新丢失
Read Uncommitted	可能，但 PostgreSQL 不会	可能	可能	可能
Read Committed	不可能	可能	可能	可能
Repeatable Read	不可能	不可能	可能，但 PostgreSQL 不会	不可能
Serializable	不可能	不可能	不可能	不可能

事务的隔离级别从低到高依次为：

- **Read Uncommitted（读未提交）**：最低的隔离级别，实际上就是不隔离，任何事务都可以看到其他事务未提交的修改；该级别可能产生各种并发异常。不过，PostgreSQL 消除了 Read Uncommitted 级别时的脏读，因为它的实现等同于 Read Committed。
- **Read Committed（读已提交）**：一个事务只能看到其他事务已经提交的数据，解决了脏读问题，但是存在不可重复读、幻读和第二类更新丢失问题。这是 PostgreSQL 的默认隔离级别。
- **Repeated Read（可重复读）**：一个事务对于同某个数据的读取结果不变，即使其他事务对该数据进行了修改并提交；不过如果其他事务删除了该记录，则无法再查询到数据（幻读）。SQL 标准中的可重复读可能出现幻读，但是 PostgreSQL 在可重复读级别消除了幻读。
- **Serializable（可串行化）**：最高的隔离级别，事务串行化执行，没有并发。

只有 Serializable 真正实现了事务的完全隔离，但是不支持并发的数据库系统应用场景非常有限。因此，需要对并发性能和隔离性进行平衡，大多数数据库（包括 PostgreSQL）的默认隔离级别为 Read Committed；此时，可以避免脏读，同时拥有不错的并发性能。

下面我们来演示一下 Read Committed 隔离级别下的并发事务处理，使 SHOW 命令可以查看当前的隔离级别：

```
show transaction_isolation;
transaction_isolation|
-----|
read committed      |
```

如果需要修改当前事务的隔离级别，可以在事务的最开始执行 SET TRANSACTION 命令：

```
begin;
SET TRANSACTION ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ
COMMITTED | READ UNCOMMITTED };
...
```

下表演示了 PostgreSQL 默认级别（READ COMMITTED）时不会发生脏读，但是存在不可重复读、幻读和更新丢失问题：

事务 1	事务 2
begin; SELECT balance	

FROM accounts WHERE id = 1; -- 返回 6000	
	begin; UPDATE accounts set balance = balance + 1000 WHERE id = 1; SELECT balance FROM accounts WHERE id = 1; -- 返回 7000
SELECT balance FROM accounts WHERE id = 1; -- 仍然返回 6000，没有脏读	
	commit; -- 提交事务
SELECT balance FROM accounts WHERE id = 1; -- 此时返回 7000，出现不可重复读	
SELECT * FROM accounts WHERE id=4; -- 返回 UserC	begin; DELETE FROM accounts WHERE id = 4; commit; -- 删除 UserC 并提交事务
SELECT * FROM accounts WHERE id=4; -- 没有结果，出现幻读	
SELECT balance FROM accounts WHERE id = 1; -- 此时返回 7000	
	begin; SELECT balance FROM accounts WHERE id = 1; -- 此时返回 7000
UPDATE accounts set balance = 6000 WHERE id = 1; -- 更新为 6000	
	UPDATE accounts set balance = 8000 WHERE id = 1; -- 等待事务 1 提交
commit;	
	commt;

<pre>SELECT balance FROM accounts WHERE id = 1; -- 返回 8000，而不是自己修改成的 6000，更新丢失</pre>	
--	--

在以上过程中，PostgreSQL 使用了锁加 MVCC（Multiversion Concurrency Control）技术来实现数据的隔离和一致性。简单来说，MVCC 就是保留每次数据修改之前的旧版本，根据隔离级别决定读取哪个版本的数据。这种实现的最大好处就是读操作永远不会阻塞写操作、写操作永远不会阻塞读操作。

如果一个事务已经修改某个数据而且未提交，则另一个事务不允许同时修改该数据（必须等待），写操作一定是相互阻塞的，需要按照顺序执行。

对于业务开发人员来说，我们一般使用 PostgreSQL 的默认隔离级别，因为它的 MVCC 实现消除了大部分的锁等待问题。虽然此时可能产生不可重复读、幻读和更新丢失，但是并不会导致数据的不一致性，因为这些都是其他事务的正常操作。

## 第 21 章 索引与优化

索引（Index）可以用于提高数据库的查询性能；但是索引也需要进行读写，同时还会占用更多的存储空间；因此了解并适当利用索引对于数据库的优化至关重要。本篇我们就来介绍如何高效地使用 PostgreSQL 索引。

### 21.1 索引简介

假如存在以下数据表：

```
CREATE TABLE test (  
    id integer,  
    name text  
);  
  
INSERT INTO test  
SELECT v, 'val:' || v FROM generate_series(1, 10000000) v;
```

我们经常需要使用类似以下的查询返回结果：

```
SELECT name FROM test WHERE id = 10000;
```

如果没有索引，数据库需要扫描整个表才能找到相应的数据。利用 **EXPLAIN** 命令可以看到数据库的执行计划，也就是 PostgreSQL 执行 SQL 语句的具体步骤：

```
explain analyze  
SELECT name FROM test WHERE id = 10000;  
QUERY PLAN  
-----  
Gather          (cost=1000.00..107137.70   rows=1   width=11)   (actual  
time=50.266..12082.777 rows=1 loops=1)   |  
  Workers Planned: 2   |  
  Workers Launched: 2   |  
  -> Parallel Seq Scan on test (cost=0.00..106137.60 rows=1 width=11)  
(actual time=7674.992..11553.964 rows=0 loops=3) |  
    Filter: (id = 10000)   |  
    Rows Removed by Filter: 3333333   |  
Planning Time: 16.480 ms   |  
Execution Time: 12093.016 ms   |
```

**Parallel Seq Scan** 表示并行顺序扫描，执行消耗了 12s；由于表中有包含大量数据，而查询只返回一行数据，显然这种方法效率很低。

 关于执行计划的更多信息，可以参考[这篇文章](#)。

如果在 **id** 列上存在索引，则可以通过索引快速找到匹配的结果。我们先创建一个索引：

```
CREATE INDEX test_id_index ON test (id);
```

创建索引需要消耗一定的时间。然后再次查看数据库的执行计划：

```
explain analyze  
SELECT name FROM test WHERE id = 10000;
```



```

QUERY PLAN
-----
Index Scan using test_id_index on test (cost=0.43..8.45 rows=1 width=11)
(actual time=20.410..20.412 rows=1 loops=1)
    Index Cond: (id = 10000)
Planning Time: 14.989 ms
Execution Time: 20.521 ms

```

**Index Scan** 表示索引扫描，执行消耗了 20ms；这种方式类似于图书最后的關鍵字索引，读者可以相对快速地浏览索引并翻到适当的页面，而不必阅读整本书来找到感兴趣的内容。

索引不仅仅能够优化查询语句，某些包含 **WHERE** 条件的 **UPDATE**、**DELETE** 语句也可以利用索引提高性能，因为修改数据的前提是找到数据。

此外，索引也可以用于优化连接查询，基于连接条件中的字段创建索引可以提高连接查询的性能。索引还能优化分组或者排序操作，因为索引自身是按照顺序进行组织存储的。

另一方面，系统维护索引需要付出一定的代价，从而增加数据修改操作的负担。所以，我们需要合理创建索引，一般只为经常使用到的字段创建索引。就像图书一样，不可能为书中的每个关键字都创建一个索引。

## 21.2 索引类型

PostgreSQL 提高了多种索引类型：B-树、哈希、GiST、SP-GiST、GIN 以及 BRIN 索引。每种索引基于不同的存储结构和算法，用于优化不同类型的查询。默认情况下，PostgreSQL 创建 B-树索引，因为它适合大部分情况下的查询。

### 21.2.1 B-树索引

B-树是一个自平衡树(self-balancing tree)，按照顺序存储数据，支持对数时间复杂度( $O(\log N)$ ) 的搜索、插入、删除和顺序访问。

举例来说，假如 100 条数据时需要 1 次磁盘 I/O，也就是说  $N$  等于 100；10000 条数据时只需要 2 次 I/O，1 亿条数据时只需要 4 次 I/O。

对于索引列上的以下比较运算符，PostgreSQL 优化器都会考虑使用 B-树索引：

- <
- <=
- =
- >=
- BETWEEN
- IN
- IS NULL
- IS NOT NULL

另外，如果模式匹配运算符 **LIKE** 和 **~** 中模式的开头不是通配符，优化器也可以使用 B-树索引，例如：

```
col LIKE 'foo%'
col ~ '^foo'
```

对于不区分大小的 **ILIKE** 和 **~\*** 运算符，如果匹配的模式以非字母的字符（不受大小写转换影响）开头，也可以使用 **B-树索引**。

**B-树索引**还可以用于优化排序操作，例如：

```
SELECT col1, col2
FROM t
WHERE col1 BETWEEN 100 AND 200
ORDER BY col1;
```

**col1** 上的索引不仅能够优化查询条件，也可以避免额外的排序操作；因为基于该索引访问时本身就是按照排序返回结果。

### 21.2.2 哈希索引

哈希索引（**Hash index**）只能用于简单的等值查找（**=**），也就是说索引字段被用于等号条件判断。因为对数据进行哈希运算之后不再保留原来的大小关系。

创建哈希索引需要使用 **HASH** 关键字：

```
CREATE INDEX index_name
ON table_name USING HASH (column_name);
```

**CREATE INDEX** 语句用于创建索引，**USING** 子句指定索引的类型，具体参考下文。

### 21.2.3 GiST 索引

**GiST** 代表通用搜索树（**Generalized Search Tree**），**GiST** 索引单个索引类型，而是一种支持不同索引策略的框架。**GiST** 索引常见的用途包括几何数据的索引和全文搜索。**GiST** 索引也可以用于优化“最近邻”搜索，例如：

```
SELECT *
FROM places
ORDER BY location <-> point '(101,456)'
LIMIT 10;
```

该语句用于查找距离某个目标地点最近的 10 个地方。

### 21.2.4 SP-GiST 索引

**SP-GiST** 代表空间分区 **GiST**，主要用于 **GIS**、多媒体、电话路由以及 **IP** 路由等数据的索引。

与 **GiST** 类似，**SP-GiST** 也支持“最近邻”搜索。

### 21.2.5 GIN 索引

**GIN** 代表广义倒排索引（**generalized inverted indexes**），主要用于单个字段中包含多个值的数据，例如 **hstore**、**array**、**jsonb** 以及 **range** 数据类型。一个倒排索引为每个元素值都创建一个单独的索引项，可以有效地查询某个特定元素值是否存在。**Google**、**百度** 这种搜索引擎利用的就是倒排索引。

## 21.2.6 BRIN 索引

BRIN 代表块区间索引 (block range indexes)，存储了连续物理范围区间内的数据摘要信息。BRIN 也相比 B-树索引要小很多，维护也更容易。对于不进行水平分区就无法使用 B-树索引的超大型表，可以考虑 BRIN。

BRIN 通常用于具有线性排序顺序的字段，例如订单表的创建日期。

关于这些索引类型的更多信息，可以参考[官方文档](#)。

## 21.3 创建索引

PostgreSQL 使用 CREATE INDEX 语句创建新的索引：

```
CREATE INDEX index_name ON table_name
[USING method]
(column_name [ASC | DESC] [NULLS FIRST | NULLS LAST]);
```

其中：

- index\_name 是索引的名称，table\_name 是表的名称；
- method 表示索引的类型，例如 btree、hash、gist、spgist、gin 或者 brin。默认为 btree；
- column\_name 是字段名，ASC 表示升序排序（默认值），DESC 表示降序索引；
- NULLS FIRST 和 NULLS LAST 表示索引中空值的排列顺序，升序索引时默认为 NULLS LAST，降序索引时默认为 NULLS FIRST。

如果我们经常使用 name 字段作为查询条件，可以为 test 表创建以下索引：

```
CREATE INDEX test_name_index ON test (name);
```

创建索引之后，优化器会自动选择是否使用索引，例如：

```
explain analyze
SELECT * FROM test WHERE name IS NULL;
QUERY PLAN
-----|
Index Scan using test_name_index on test  (cost=0.43..5.77 rows=1 width=15)
(actual time=0.036..0.037 rows=0 loops=1)|
    Index Cond: (name IS NULL)           |
Planning Time: 1.067 ms                  |
Execution Time: 0.048 ms                  |
```

基于索引字段的 IS NULL 运算符同样可以利用索引进行优化。

### 21.3.1 唯一索引

在创建索引时，可以使用 UNIQUE 关键字指定唯一索引：

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name [ASC | DESC] [NULLS FIRST | NULLS LAST]);
```

唯一索引可以用于实现唯一约束，PostgreSQL 目前只支持 B-树类型的唯一索引。多个 NULL 被看作是不同的值，因此唯一索引字段可以存在多个空值。

 对于主键和唯一约束，PostgreSQL 会自动创建一个唯一索引，从而确保唯一性。

### 21.3.2 多列索引

PostgreSQL 支持基于多个字段的索引，也就是多列索引（复合索引）。默认情况下，一个多列索引最多可以使用 32 个字段。只有 B-树、GIST、GIN 和 BRIN 索引支持多列索引。

```
CREATE [UNIQUE] INDEX index_name ON table_name
[USING method]
(column1 [ASC | DESC] [NULLS FIRST | NULLS LAST], ...);
```

对于多列索引，应该将最常作为查询条件使用的字段放在左边，较少使用的字段放在右边。例如，基于 (c1, c2, c3) 创建的索引可以优化以下查询：

```
WHERE c1 = v1 and c2 = v2 and c3 = v3;
WHERE c1 = v1 and c2 = v2;
WHERE c1 = v1;
```

但是以下查询无法使用该索引：

```
WHERE c2 = v2;
WHERE c3 = v3;
WHERE c2 = v2 and c3 = v3;
```

对于多列唯一索引，字段的组合值不能重复；但是如果某个字段是空值，其他字段可以出现重复值。

### 21.3.3 函数索引

函数索引，也叫表达式索引，是指基于某个函数或者表达式的值创建的索引。PostgreSQL 中创建函数索引的语法如下：

```
CREATE [UNIQUE] INDEX index_name
ON table_name (expression);
```

expression 是基于字段的表达式或者函数。

以下查询在 name 字段上使用了 upper 函数：

```
explain analyze
SELECT * FROM test WHERE upper(name) = 'VAL:10000';
QUERY PLAN
-----|
Gather          (cost=1000.00..122556.19   rows=50001   width=15)   (actual
time=18.629..7310.422 rows=1 loops=1)   |
  Workers Planned: 2                               |
  Workers Launched: 2                              |
  -> Parallel Seq Scan on test  (cost=0.00..116556.09 rows=20834 width=15)
(actual time=4746.266..7171.452 rows=0 loops=3)   |
    Filter: (upper(name) = 'VAL:10000'::text)      |
    Rows Removed by Filter: 3333333               |
Planning Time: 0.100 ms                           |
Execution Time: 7310.444 ms                        |
```

虽然 name 字段上存在索引 test\_name\_index，但是函数会导致优化器无法使用该索引。为了优化这种不区分大小写的查询语句，可以基于 name 字段创建一个函数索引：

```
drop index test_name_index;
create index test_name_index on test(upper(name));
```

再次查看该语句的执行计划：

```
explain analyze
SELECT * FROM test WHERE upper(name) = 'VAL:10000';
QUERY PLAN
-----|
Bitmap Heap Scan on test  (cost=1159.93..57095.47 rows=50000 width=15)
(actual time=17.046..17.047 rows=1 loops=1)
  Recheck Cond: (upper(name) = 'VAL:10000'::text)
  Heap Blocks: exact=1
  -> Bitmap Index Scan on test_name_index  (cost=0.00..1147.43 rows=50000
width=0) (actual time=17.032..17.032 rows=1 loops=1)
    Index Cond: (upper(name) = 'VAL:10000'::text)
Planning Time: 1.985 ms
Execution Time: 17.080 ms
```

函数索引的维护成本比较高，因为插入和更新时都需要进行函数计算。

### 21.3.4 部分索引

部分索引（**partial index**）是只针对表中部分数据行创建的索引，通过一个 **WHERE** 子句指定需要索引的行。例如，对于订单表 **orders**，绝大部的订单都处于完成状态；我们只需要针对未完成的订单进行查询跟踪，可以创建一个部分索引：

```
create table orders(order_id int primary key, order_ts timestamp, finished
boolean);

create index orders_unfinished_index
on orders (order_id)
WHERE finished is not true;
```

该索引只包含了未完成的订单 **id**，比直接基于 **finished** 字段创建的索引小很多。它可以用于优化未完成订单的查询：

```
explain analyze
SELECT order_id
FROM orders
WHERE finished is not true;
QUERY PLAN
-----|
Bitmap Heap Scan on orders  (cost=4.38..24.33 rows=995 width=4) (actual
time=0.010..0.010 rows=0 loops=1)
  Recheck Cond: (finished IS NOT TRUE)
  -> Bitmap Index Scan on orders_unfinished_index  (cost=0.00..4.13 rows=995
width=0) (actual time=0.004..0.004 rows=0 loops=1)
Planning Time: 0.130 ms
Execution Time: 0.049 ms
```

### 21.3.5 覆盖索引

PostgreSQL 中的索引都属于二级索引，意味着索引和数据是分开存储的。因此通过索引查找数据即需要访问索引，又需要访问表，而表的访问是随机 I/O。

为了解决这个性能问题，PostgreSQL 支持 **Index-Only Scan**，只需要访问索引的数据就能获得需要的结果，而不需要再次访问表中的数据。例如：

```
CREATE TABLE t (a int, b int, c int);
CREATE UNIQUE INDEX idx_t_ab ON t USING btree (a, b) INCLUDE (c);
```

以上语句基于字段 **a** 和 **b** 创建了多列索引，同时利用 **INCLUDE** 在索引的叶子节点存储了字段 **c** 的值。以下查询可以利用 **Index-Only Scan**：

```
explain analyze
SELECT a, b, c
FROM t
WHERE a = 100 and b = 200;
QUERY PLAN
-----|
Index Only Scan using idx_t_ab on t  (cost=0.15..8.17 rows=1 width=12) (actual
time=0.007..0.007 rows=0 loops=1)
  Index Cond: ((a = 100) AND (b = 200))
  Heap Fetches: 0
Planning Time: 0.078 ms
Execution Time: 0.021 ms
```

以上查询只返回索引字段（**a**、**b**）和覆盖的字段（**c**），可以仅通过扫描索引即可返回结果。

B-树索引支持 **Index-Only Scan**，GiST 和 SP-GiST 索引支持某些运算符的 **Index-Only Scan**，其他索引不支持这种方式。

## 21.4 查看索引

PostgreSQL 提供了一个关于索引的视图 **pg\_indexes**，可以用于查看索引的信息：

```
SELECT * FROM pg_indexes WHERE tablename = 'test';
schemaname|tablename|indexname      |tablespace|indexdef
-----|-----|-----|-----|-----|
Public    |test     |test_id_index  |          |CREATE INDEX test_id_index ON
public.test USING btree (id)
public    |test     |test_name_index|          |CREATE INDEX test_name_index
ON public.test USING btree (upper(name))
```

该视图包含的字段依次为：模式名、表名、索引名、表空间以及索引的定义语句。

psql 客户端可以使用 **\d table\_name** 命令查看表的结构，包括表中的索引信息。

## 21.5 维护索引

PostgreSQL 提供了一些修改和重建索引的方法：

```
ALTER INDEX index_name RENAME TO new_name;
ALTER INDEX index_name SET TABLESPACE tablespace_name;
```

```
REINDEX [ ( VERBOSE ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } index_name;
```

两个 **ALTER INDEX** 语句分别用于重命名索引和移动索引到其他表空间；**REINDEX** 用于重建索引数据，支持不同级别的索引重建。

另外，索引被创建之后，系统会在修改数据的同时自动更新索引。不过，我们需要定期执行 **ANALYZE** 命令更新数据库的统计信息，以便优化器能够合理使用索引。

## 21.6 删除索引

如果需要删除一个已有的索引，可以使用以下命令：

```
DROP INDEX index_name [ CASCADE | RESTRICT ];
```

**CASCADE** 表示级联删除其他依赖该索引的对象；**RESTRICT** 表示如果存在依赖于该索引的对象，将会拒绝删除操作。默认为 **RESTRICT**。

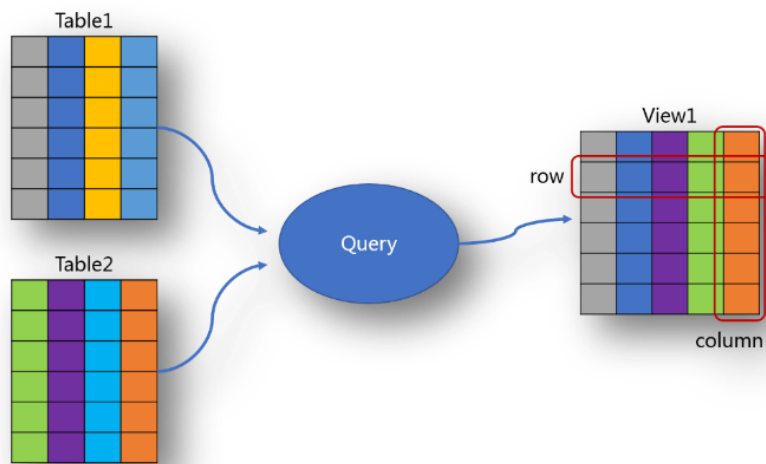
我们可以使用以下语句删除 **test** 上的索引：

```
DROP INDEX test_id_index, test_name_index;
```

## 第 22 章 视图

### 22.1 视图概述

视图（View）本质上是一个存储在数据库中的查询语句。视图本身不包含数据，也被称为虚拟表。我们在创建视图时给它指定了一个名称，然后可以像表一样对其进行查询。



<https://blog.csdn.net/horses>

合理使用的视图可以给我们带来以下好处：

- 替代复杂查询，减少复杂性。将复杂的查询语句定义为视图，然后使用视图进行查询，可以隐藏具体的实现；
- 提供一致性接口，实现业务规则。在视图的定义中增加业务逻辑，对外提供统一的接口；当底层表结构发生变化时，只需要修改视图接口，而不需要修改外部应用，可以简化代码的维护并减少错误；
- 控制对于表的访问，提高安全性。通过视图为用户提供数据访问，而不是直接访问表；同时可以限制允许访问某些敏感信息，例如身份证号、工资等。

### 22.2 创建视图

PostgreSQL 使用 CREATE VIEW 语句创建视图：

```
CREATE VIEW view_name AS query;
```

其中，view\_name 是视图的名称；AS 之后是视图的查询语句，可以是简单查询或者复杂的查询。以下语句创建了一个包含员工详细信息的视图：



```
CREATE VIEW emp_details_view
AS SELECT
    e.employee_id,
    e.job_id,
    e.manager_id,
    e.department_id,
    d.location_id,
    e.first_name,
    e.last_name,
    e.salary,
    e.commission_pct,
    d.department_name,
    j.job_title
FROM employees e
JOIN departments d ON (e.department_id = d.department_id)
JOIN jobs j ON (j.job_id = e.job_id);
```

该视图使用了连接查询从 3 个表中获取信息。

接下来，我们可以直接从视图中查询数据，不需要每次编写复杂的连接查询：

```
SELECT * FROM emp_details_view
WHERE department_name = 'IT';
```

该语句返回了 IT 部门的员工信息。

## 22.3 修改视图

如果需要修改视图定义中的查询，可以使用 CREATE OR REPLACE 语句：

```
CREATE OR REPLACE VIEW view_name
AS
query
```

PostgreSQL 目前只支持追加视图定义中的字段，不支持减少字段或者修改字段的名称或顺序。例如，我们可以为视图 emp\_details\_view 增加一个字段 hire\_date：

```
CREATE OR REPLACE VIEW emp_details_view
AS SELECT
    e.employee_id,
    e.job_id,
    e.manager_id,
    e.department_id,
    d.location_id,
    e.first_name,
    e.last_name,
    e.salary,
    e.commission_pct,
    d.department_name,
    j.job_title,
    e.hire_date
FROM employees e
JOIN departments d ON (e.department_id = d.department_id)
JOIN jobs j ON (j.job_id = e.job_id);
```

另外，PostgreSQL 还提供了 ALTER VIEW 语句修改视图的属性。例如以下语句用于修改视图的名称：

```
ALTER VIEW emp_details_view RENAME TO emp_info_view;
```

该语句将视图 emp\_details\_view 重命名为 emp\_info\_view。

ALTER VIEW 语句还提供了其他的修改功能，例如设置字段的默认值、修改视图所属的模式等，具体可以参考[官方文档](#)。

## 22.4 删除视图

使用 DROP VIEW 语句删除一个已有的视图：

```
DROP VIEW [ IF EXISTS ] name [ CASCADE | RESTRICT ];
```

其中，IF EXISTS 可以避免删除一个不存在的视图时产生错误；CASCADE 表示级联删除依赖于该视图的对象；RESTRICT 表示如果存在依赖对象则提示错误信息，这是默认值。

我们将视图 emp\_info\_view 删除：

```
DROP VIEW emp_info_view;
```

## 22.5 递归视图

第 17 章介绍了通用表表达式（CTE），包括使用 RECURSIVE 关键字实现递归查询。与此类似，视图的定义中也可以使用 RECURSIVE：

```
CREATE RECURSIVE VIEW view_name (column_names) AS query;
```

需要注意的是，递归视图需要指定字段的名称 column\_names。以上语句实际上等价于：

```
CREATE VIEW view_name AS  
WITH RECURSIVE cte_name (column_names) AS (query)  
SELECT column_names FROM cte_name;
```

我们将第 17 章中通过递归 CTE 遍历组织结构的查询语句定义为一个递归视图：

```
CREATE RECURSIVE VIEW employee_path(employee_id, employee_name, path) AS  
    SELECT employee_id, CONCAT(first_name, ',', last_name), CONCAT(first_name,  
'', last_name) AS path  
    FROM employees  
    WHERE manager_id IS NULL  
    UNION ALL  
    SELECT e.employee_id, CONCAT(e.first_name, ',', e.last_name),  
CONCAT(ep.path, '->', e.first_name, ',', e.last_name)  
    FROM employee_path ep  
    JOIN employees e ON ep.employee_id = e.manager_id;
```

## 22.6 可更新视图

如果一个视图满足以下条件：

- 视图定义的 **FROM** 子句中只包含一个表或者可更新视图；
- 视图定义的最顶层查询语句中不包含以下子句：**GROUP BY**、**HAVING**、**LIMIT**、**OFFSET**、**DISTINCT**、**WITH**、**UNION**、**INTERSECT** 以及 **EXCEPT**；
- **SELECT** 列表中不包含窗口函数、集合函数或者聚合函数（例如 **SUM**、**COUNT**、**AVG** 等）。

那么该视图被称为可更新视图（**updatable view**），意味着我们可以对其执行 **INSERT**、**UPDATE** 以及 **DELETE** 语句，PostgreSQL 会将这些操作转换为对底层表的操作。

我们创建一个视图 `employees_it`：

```
CREATE VIEW employees_it AS
SELECT employee_id,
       first_name,
       last_name,
       email,
       phone_number,
       hire_date,
       job_id,
       manager_id,
       department_id
FROM employees
WHERE department_id = 60;
```

视图 `employees_it` 只包含了 IT 部门的员工信息，并且不包含员工的月薪字段。该视图目前只能查询到 5 条数据：

```
SELECT employee_id, first_name, last_name FROM employees_it;
employee_id|first_name|last_name|
-----|-----|-----|
      103|Alexander |Hunold   |
      104|Bruce     |Ernst    |
      105|David     |Austin   |
      106|Valli     |Pataballa|
      107|Diana     |Lorentz  |
```

我们通过视图 `employees_it` 为 `employees` 表增加一个员工：

```
INSERT INTO employees_it
(employee_id, first_name, last_name, email, phone_number, hire_date, job_id,
manager_id, department_id)
VALUES (207, 'Tony', 'Dong', 'DONG', '590.423.5568', '2020-05-06', 'IT_PROG',
103, 60);

SELECT employee_id, first_name, last_name FROM employees_it;
employee_id|first_name|last_name|
-----|-----|-----|
      103|Alexander |Hunold   |
      104|Bruce     |Ernst    |
      105|David     |Austin   |
```

106 Valli	Pataballa
107 Diana	Lorentz
207 Tony	Dong

需要注意，不在视图定义中的字段不能通过视图进行修改。以下语句尝试通过视图 `employees_it` 修改员工的月薪：

```
UPDATE employees_it
SET salary = 5000
WHERE employee_id = 207;
SQL Error [42703]: ERROR: column "salary" of relation "employees_it" does not exist
Position: 26
```

不在视图定义中的字段不能通过视图进行操作。

我们将新增的员工从 `employees` 表中删除：

```
DELETE FROM employees_it
WHERE employee_id = 207;
```

## 22.6.1 WITH CHECK OPTION

我们再次通过视图 `employees_it` 为 `employees` 表增加一个员工，此时将部门编号改为 80：

```
INSERT INTO employees_it
(employee_id, first_name, last_name, email, phone_number, hire_date, job_id,
manager_id, department_id)
VALUES(207, 'Tony', 'Dong', 'DONG', '590.423.5568', '2020-05-06', 'IT_PROG',
149, 80);

SELECT employee_id,first_name, last_name
FROM employees_it
WHERE employee_id = 207;
employee_id|first_name|last_name|
-----|-----|-----|

SELECT employee_id,first_name, last_name
FROM employees
WHERE employee_id = 207;
employee_id|first_name|last_name|
-----|-----|-----|
207|Tony      |Dong      |
```

新增加的员工不属于 IT 部门，增加之后无法通过视图查询，但是可以通过 `employees` 表查询。

为了防止通过视图插入或者修改视图不可见的的数据，可以使用 `WITH CHECK OPTION` 选项：

```
CREATE OR REPLACE VIEW employees_it AS
SELECT employee_id,
       first_name,
       last_name,
       email,
       phone_number,
       hire_date,
       job_id,
```

```
    manager_id,  
    department_id  
FROM employees  
WHERE department_id = 60  
WITH CHECK OPTION;
```

再次通过视图 `employees_it` 为 `employees` 表增加一个员工：

```
DELETE FROM employees  
WHERE employee_id = 207;  
  
INSERT INTO employees_it  
    (employee_id, first_name, last_name, email, phone_number, hire_date, job_id,  
manager_id, department_id)  
VALUES (207, 'Tony', 'Dong', 'DONG', '590.423.5568', '2020-05-06', 'IT_PROG',  
149, 80);  
SQL Error [44000]: ERROR: new row violates check option for view "employees_it"  
    Detail: Failing row contains (207, Tony, Dong, DONG, 590.423.5568, 2020-05-06,  
IT_PROG, null, null, 149, 80).
```

执行结果显示违反检查选项，无法插入数据。

`WITH CASCADED CHECK OPTION` 选项会对视图以及它所依赖的其他视图进行级联检查；  
`WITH LOCAL CHECK OPTION` 选项只对当前视图进行检查。默认为 `CASCADED`。

## 第 23 章 PL/pgSQL 存储过程

### 23.1 概述

除了标准 SQL 语句之外, PostgreSQL 还支持使用各种过程语言(例如 PL/pgSQL、C、PL/Tcl、PL/Python、PL/Perl、PL/Java 等 ) 创建复杂的过程和函数, 称为存储过程 (Stored Procedure) 和自定义函数 (User-Defined Function)。存储过程支持许多过程元素, 例如控制结构、循环和复杂的计算。

使用存储过程带来的好处包括:

- 减少应用和数据库之间的网络传输。所有的 SQL 语句都存储在数据库服务器中, 应用程序只需要发送函数调用并获取除了结果, 避免了发送多个 SQL 语句并等待结果。
- 提高应用的性能。因为自定义函数和存储过程进行了预编译并存储在数据库服务器中。
- 可重用性。存储过程和函数的功能可以被多个应用同时使用。

当然, 使用存储过程也可能带来一些问题:

- 导致软件开发缓慢。因为存储过程需要单独学习, 而且很多开发人员并不具备这种技能。
- 不易进行版本管理和代码调试。
- 不同数据库管理系统之间无法移植, 语法存在较大的差异。

本文主要介绍 PL/pgSQL 存储过程, 它和 Oracle PL/SQL 非常类似, 是 PostgreSQL 默认支持的存储过程。使用 PL/pgSQL 的原因包括:

- PL/pgSQL 简单易学, 无论是否具有编程基础都能够很快学会。
- PL/pgSQL 是 PostgreSQL 默认支持的过程语言, PL/pgSQL 开发的自定义函数可以和内置函数一样使用。
- PL/pgSQL 提高了许多强大的功能, 例如游标, 可以实现复杂的函数。

### 23.2 PL/pgSQL 代码块结构

PL/pgSQL 是一种块状语言, 因此存储过程和函数以代码块的形式进行组织。以下是一个 PL/pgSQL 代码块的定义:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements;
    ...
END [ label ];
```

其中, label 是一个可选的代码块标签, 可以用于 EXIT 语句退出指定的代码块, 或者限定

变量的名称；**DECLARE** 是一个可选的声明部分，用于定义变量；**BEGIN** 和 **END** 之间是代码主体，也就是主要的功能代码；所有的语句都使用分号（**;**）结束，**END** 之后的分号表示代码块结束。

以下是一个简单的代码块示例：

```
DO $$
DECLARE
    name text;
BEGIN
    name := 'PL/pgSQL';
    RAISE NOTICE 'Hello %!', name;
END $$;
```

以上是一个匿名块，与此相对的是命名块（也就是存储过程和函数）。其中，**DO** 语句用于执行匿名块；我们定义了一个字符串变量 **name**，然后给它赋值并输出一个信息；**RAISE NOTICE** 用于输出通知消息。

**\$\$**用于替换单引号（**'**），因为 **PL/pgSQL** 代码主体必须是字符串文本，意味着代码中所有的单引号都必须转义（重复写两次）。对于上面的示例，需要写成以下形式：

```
DO
'DECLARE
    name text;
BEGIN
    name := 'PL/pgSQL';
    RAISE NOTICE 'Hello %!', name;
END ';
```

显然这种写法很不方便，因此 **PL/pgSQL** 提供了 **\$\$** 避免单引号问题。我们经常还会遇到其他形式的符号，例如 **\$function\$** 或者 **\$procedure\$**，作用也是一样。

在 **psql** 客户端运行以上代码的结果如下：

```
postgres=# DO $$
postgres$# DECLARE
postgres$#   name text;
postgres$# BEGIN
postgres$#   name := 'PL/pgSQL';
postgres$#   RAISE NOTICE 'Hello %!', name;
postgres$# END $$;

NOTICE:  Hello PL/pgSQL!
```

### 23.2.1 嵌套子块

**PL/pgSQL** 支持代码块的嵌套，也就是将一个代码块嵌入其他代码块的主体中。被嵌套的代码块被称为子块（**subblock**），包含子块的代码块被称为外部块（**subblock**）。子块可以将代码进行逻辑上的拆分，子块中可以定义与外部块重名的变量，而且在子块内拥有更高的优先级。例如：

```
DO $$
<<outer_block>>
DECLARE
```

```

    name text;
BEGIN
    name := 'outer_block';
    RAISE NOTICE 'This is %', name;

    DECLARE
        name text := 'sub_block';
    BEGIN
        RAISE NOTICE 'This is %', name;
        RAISE NOTICE 'The name FROM the outer block is %', outer_block.name;
    END;

    RAISE NOTICE 'This is %', name;

END outer_block $$;

```

首先，外部块中定义了一个变量 `name`，值为“outer\_block”，输出该变量的值；然后在子块中定义了同名的变量，值为“sub\_block”，输出该变量的值，并且通过代码块标签输出了外部块的变量值；最后再次输出该变量的值。以上代码执行的输出结果如下：

```

NOTICE: This is outer_block
NOTICE: This is sub_block
NOTICE: The name FROM the outer block is outer_block
NOTICE: This is outer_block

```

## 23.3 声明与赋值

与其他编程语言类似，PL/pgSQL 支持定义变量和常量。

### 23.3.1 变量

变量是一个有意义的名字，代表了内存中的某个位置。变量总是属于某个数据类型，变量的值可以在运行时被修改。

在使用变量之前，需要在代码的声明部分进行声明：

```
variable_name data_type [ NOT NULL ] [ { DEFAULT | := | = } expression ];
```

其中，`variable_name` 是变量的名称，通常需要指定一个有意义的名称；`data_type` 是变量的类型，可以是任何 SQL 数据类型；如果指定了 `NOT NULL`，必须使用后面的表达式为变量指定初始值。

以下是一些变量声明的示例：

```

user_id integer;
quantity numeric(5) DEFAULT 0;
url varchar := 'http://mysite.com';

```

除了基本的 SQL 数据类型之外，PL/pgSQL 还支持基于表的字段或行或者其他变量定义变量：

```

myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;

```



```
amount quantity%TYPE;
```

`myrow` 是一个行类型的变量,可以存储查询语句返回的数据行(数据行的结构要和 `tablename` 相同); `myfield` 的数据类型取决于 `tablename.columnname` 字段的定义; `amount` 和 `quantity` 的类型一致。

与行类型变量类似的还有记录类型变量,例如:

```
arow RECORD;
```

记录类型的变量没有预定义的结构,只有当变量被赋值时才确定,而且可以在运行时被改变。记录类型的变量可以用于任意查询语句或者 `FOR` 循环变量。

除此之外, `PL/pgSQL` 还可以使用 `ALIAS` 定义一个变量别名:

```
newname ALIAS FOR oldname;
```

此时, `newname` 和 `oldname` 代表了相同的对象。

### 23.3.2 常量

如果在定义变量时指定了 `CONSTANT` 关键字,意味着定义的是常量。常量的值需要在声明时初始化,并且不能修改。

以下示例通过定义常量 `PI` 计算圆的面积:

```
DO $$
DECLARE
    PI CONSTANT NUMERIC := 3.14159265;
    radius NUMERIC;
BEGIN
    radius := 1.0;
    RAISE NOTICE 'The area is %', PI * radius * radius;
END $$;

NOTICE: The area is 3.1415926500
```

常量可以用于避免魔数 (magic number), 提高代码的可读性; 也可以减少代码的维护工作, 所有使用常量的代码都会随着常量值的修改而同步, 不需要修改多个硬编码的数据值。

## 23.4 控制结构

### 23.4.1 IF 语句

`IF` 语句可以基于条件选择性执行操作, `PL/pgSQL` 提供了三种形式的 `IF` 语句。

- `IF ... THEN ... END IF`
- `IF ... THEN ... ELSE ... END IF`
- `IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF`

首先, 最简单的 `IF` 语句如下:

```
IF boolean-expression THEN
```

```
statements
END IF;
```

如果表达式 **boolean-expression** 的值为真，执行 **THEN** 之后的语句；否则，忽略这些语句。  
例如：

```
DO $$
BEGIN
  IF 2 > 3 THEN
    RAISE NOTICE '2 大于 3';
  END IF;

  IF 2 < 3 THEN
    RAISE NOTICE '2 小于 3';
  END IF;
END $$;

NOTICE:  2 小于 3
```

第二种 **IF** 语句的语法如下：

```
IF boolean-expression THEN
  statements
ELSE
  other-statements
END IF;
```

如果表达式 **boolean-expression** 的值为真，执行 **THEN** 之后的语句；否则，执行 **ELSE** 之后的语句。例如：

```
DO $$
BEGIN
  IF 2 > 3 THEN
    RAISE NOTICE '2 大于 3';
  ELSE
    RAISE NOTICE '2 小于 3';
  END IF;
END $$;

NOTICE:  2 小于 3
```

第三种 **IF** 语句支持多个条件分支：

```
IF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements ]
[ ELSIF boolean-expression THEN
  statements ]
...
[ ELSE
  statements ]
END IF;
```

依次判断条件中的表达式，如果某个条件为真，执行相应的语句；如果所有条件都为假，执行 **ELSE** 后面的语句；如果没有 **ELSE** 就什么都不执行。例如：

```
DO $$
```

```


DECLARE
    i integer := 3;
    j integer := 3;
BEGIN
    IF i > j THEN
        RAISE NOTICE 'i 大于 j';
    ELSIF i < j THEN
        RAISE NOTICE 'i 小于 j';
    ELSE
        RAISE NOTICE 'i 等于 j';
    END IF;
END $$;

NOTICE: i 等于 j
DO

```

### 23.4.2 CASE 语句

除了 IF 语句之外，PostgreSQL 还提供了 CASE 语句，同样可以根据不同的条件执行不同的分支语句。CASE 语句分为两种：简单 CASE 语句和搜索 CASE 语句。

 CASE 语句和第 13 章中介绍的 CASE 表达式不是一个概念，CASE 表达式是一个 SQL 表达式。

简单 CASE 语句的结构如下：

```

CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
        statements
    [ WHEN expression [, expression [ ... ]] THEN
        statements
    ... ]
    [ ELSE
        statements ]
END CASE;

```

首先，计算 search-expression 的值；然后依次和 WHEN 中的表达式进行等值比较；如果找到了相等的值，执行相应的 statements；后续的分不再进行判断；如果没有匹配的值，执行 ELSE 语句；如果此时没有 ELSE，将会抛出 CASE\_NOT\_FOUND 异常。

例如：

```

DO $$
DECLARE
    i integer := 3;
BEGIN
    CASE i
        WHEN 1, 2 THEN
            RAISE NOTICE 'one or two';
        WHEN 3, 4 THEN
            RAISE NOTICE 'three or four';
        ELSE
            RAISE NOTICE 'other value';
        END CASE;
END $$;

```

```
NOTICE: three or four
```

简单 **CASE** 语句只能进行简单的等值比较，搜索 **CASE** 语句可以实现更复杂的控制逻辑：

```
CASE
    WHEN boolean-expression THEN
        statements
    [ WHEN boolean-expression THEN
        statements
    ... ]
    [ ELSE
        statements ]
END CASE;
```

依次判断每个 **WHEN** 之后的表达式，如果为真则执行相应的语句；后续的分支不再进行判断；如果没有匹配的值，执行 **ELSE** 语句；如果此时没有 **ELSE**，将会抛出 **CASE\_NOT\_FOUND** 异常。例如：

```
DO $$
DECLARE
    i integer := 3;
BEGIN
    CASE
        WHEN i BETWEEN 0 AND 10 THEN
            RAISE NOTICE 'value is between zero and ten';
        WHEN i BETWEEN 11 AND 20 THEN
            RAISE NOTICE 'value is between eleven and twenty';
        ELSE
            RAISE NOTICE 'other value';
        END CASE;
END $$;
```

搜索 **CASE** 表达式可以构造任意复杂的判断逻辑，实现 **IF** 语句的各种功能。

### 23.4.3 循环语句

PostgreSQL 提供 4 种循环执行命令的语句：**LOOP**、**WHILE**、**FOR** 和 **FOREACH** 循环，以及循环控制的 **EXIT** 和 **CONTINUE** 语句。

首先，**LOOP** 用于定义一个无限循环语句：

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

一般需要使用 **EXIT** 或者 **RETURN** 语句退出循环，**label** 可以用于 **EXIT** 或者 **CONTINUE** 语句退出或者跳到执行的嵌套循环中。例如：

```
DO $$
DECLARE
    i integer := 0;
BEGIN
    LOOP
        EXIT WHEN i = 5;
        i := i + 1;
    
```

```
    RAISE NOTICE 'Loop: %', i;
END LOOP;
END $$;
```

```
NOTICE: Loop: 1
NOTICE: Loop: 2
NOTICE: Loop: 3
NOTICE: Loop: 4
NOTICE: Loop: 5
```

其中，EXIT 语句用于退出循环。完整的 EXIT 语句如下：

```
EXIT [ label ] [ WHEN boolean-expression ];
```

另一个控制循环的语句是 CONTINUE：

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

CONTINUE 表示忽略后面的语句，直接进入下一次循环。例如：

```
DO $$
DECLARE
    i integer := 0;
BEGIN
    LOOP
        EXIT WHEN i = 10;
        i := i + 1;
        CONTINUE WHEN mod(i, 2) = 1;
        RAISE NOTICE 'Loop: %', i;
    END LOOP;
END $$;
```

```
NOTICE: Loop: 2
NOTICE: Loop: 4
NOTICE: Loop: 6
NOTICE: Loop: 8
NOTICE: Loop: 10
```

当变量 i 为奇数时，直接进入下一次循环，不会打印出变量的值。

WHILE 循环的语法如下：

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

当表达式 boolean-expression 的值为真时，循环执行其中的语句；然后重新计算表达式的值，当表达式的值假时退出循环。例如：

```
DO $$
DECLARE
    i integer := 0;
BEGIN
    WHILE i < 5 LOOP
        i := i + 1;
        RAISE NOTICE 'Loop: %', i;
    END LOOP;
END $$;
```

```
NOTICE: Loop: 1
NOTICE: Loop: 2
NOTICE: Loop: 3
NOTICE: Loop: 4
NOTICE: Loop: 5
```

**FOR** 循环可以用于遍历一个整数范围或者查询结果集，遍历整数范围的语法如下：

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

**FOR** 循环默认从小到大进行遍历，**REVERSE** 表示从大到小遍历；**BY** 用于指定每次的增量，默认为 1。例如：

```
DO $$
BEGIN
    FOR i IN 1..5 BY 2 LOOP
        RAISE NOTICE 'Loop: %', i;
    END LOOP;
END $$;

NOTICE: Loop: 1
NOTICE: Loop: 3
NOTICE: Loop: 5
```

变量 **i** 不需要提前定义，可以在 **FOR** 循环内部使用。

遍历查询结果集的 **FOR** 循环如下：

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

其中，**target** 可以是一个 **RECORD** 变量、行变量或者逗号分隔的标量列表。在循环中，**target** 代表了每次遍历的行数据。例如：

```
DO $$
DECLARE
    emp record;
BEGIN
    FOR emp IN (SELECT * FROM employees LIMIT 5) LOOP
        RAISE NOTICE 'Loop: %,%', emp.first_name, emp.last_name;
    END LOOP;
END $$;

NOTICE: Loop: Steven,King
NOTICE: Loop: Neena,Kochhar
NOTICE: Loop: Lex,De Haan
NOTICE: Loop: Alexander,Hunold
NOTICE: Loop: Bruce,Ernst
```

**FOREACH** 循环与 **FOR** 循环类似，只不过变量的是一个数组：

```
[ <<label>> ]
```

```
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

如果没有指定 **SLICE** 或者指定 **SLICE 0**, **FOREACH** 将会变量数组中的每个元素。例如:

```
DO $$
DECLARE
    x int;
BEGIN
    FOREACH x IN ARRAY (ARRAY[[1,2,3],[4,5,6]])
    LOOP
        RAISE NOTICE 'x = %', x;
    END LOOP;
END $$;

NOTICE: x = 1
NOTICE: x = 2
NOTICE: x = 3
NOTICE: x = 4
NOTICE: x = 5
NOTICE: x = 6
```

如果指定了一个正整数的 **SLICE**, **FOREACH** 将会变量数组的切片; **SLICE** 不能大于数组的维度。例如:

```
DO $$
DECLARE
    x int[];
BEGIN
    FOREACH x SLICE 1 IN ARRAY (ARRAY[[1,2,3],[4,5,6]])
    LOOP
        RAISE NOTICE 'row = %', x;
    END LOOP;
END $$;

NOTICE: row = {1,2,3}
NOTICE: row = {4,5,6}
```

以上示例通过 **FOREACH** 语句遍历了数组的一维切片。

## 23.5 游标

**PL/pgSQL** 游标允许我们封装一个查询, 然后每次处理结果集中的一条记录。游标可以将大结果集拆分成许多小的记录, 避免内存溢出; 另外, 我们可以定义一个返回游标引用的函数, 然后调用程序可以基于这个引用处理返回的结果集。

使用游标的步骤大体如下:

1. 声明游标变量;
2. 打开游标;
3. 从游标中获取结果;

4. 判断是否存在更多结果。如果存在，执行第 3 步；否则，执行第 5 步；
5. 关闭游标。

我们直接通过一个示例演示使用游标的过程：

```
DO $$
DECLARE
    rec_emp RECORD;
    cur_emp CURSOR(p_deptid INTEGER) FOR
        SELECT first_name, last_name, hire_date
        FROM employees
        WHERE department_id = p_deptid;
BEGIN
    -- 打开游标
    OPEN cur_emp(60);

    LOOP
        -- 获取游标中的记录
        FETCH cur_emp INTO rec_emp;
        -- 没有找到更多数据时退出循环
        EXIT WHEN NOT FOUND;

        RAISE NOTICE '%,% hired at:%' , rec_emp.first_name, rec_emp.last_name,
rec_emp.hire_date;
    END LOOP;

    -- Close the cursor
    CLOSE cur_emp;
END $$;

NOTICE: Alexander,Hunold hired at:2006-01-03
NOTICE: Bruce,Ernst hired at:2007-05-21
NOTICE: David,Austin hired at:2005-06-25
NOTICE: Valli,Pataballa hired at:2006-02-05
NOTICE: Diana,Lorentz hired at:2007-02-07
```

首先，声明了一个游标 `cur_emp`，并且绑定了一个查询语句，通过一个参数 `p_deptid` 获取指定部门的员工；然后使用 `OPEN` 打开游标；接着在循环中使用 `FETCH` 语句获取游标中的记录，如果没有找到更多数据退出循环语句；变量 `rec_emp` 用于存储游标中的记录；最后使用 `CLOSE` 语句关闭游标，释放资源。

游标是 PL/pgSQL 中的一个强大的数据处理功能，更多的使用方法可以参考[官方文档](#)。

## 23.6 错误处理

### 23.6.1 报告错误和信息

PL/pgSQL 提供了 `RAISE` 语句，用于打印消息或者抛出错误：

```
RAISE level format;
```

不同的 `level` 代表了错误的不同严重级别，包括：



- DEBUG
- LOG
- NOTICE
- INFO
- WARNING
- EXCEPTION

在上文示例中,我们经常使用 **NOTICE** 输出一些信息。如果不指定 **level**,默认为 **EXCEPTION**,将会抛出异常并且终止代码运行。

**format** 是一个用于提供信息内容的字符串,可以使用百分号(%)占位符接收参数的值,两个连写的百分号(%%)表示输出百分号自身。

以下是一些 **RAISE** 语句示例:

```
DO $$
BEGIN
    RAISE DEBUG 'This is a debug text.';
    RAISE INFO 'This is an information.';
    RAISE LOG 'This is a log.';
    RAISE WARNING 'This is a warning at %', now();
    RAISE NOTICE 'This is a notice %';
END $$;

INFO: This is an information.
WARNING: This is a warning at 2020-05-16 11:27:06.138569+08
NOTICE: This is a notice %
```

从结果可以看出,并非所有的消息都会打印到客户端和服务器日志中。这个可以通过配置参数 [client\\_min\\_messages](#) 和 [log\\_min\\_messages](#) 进行设置。

对于 **EXCEPTION** 级别的错误,可以支持额外的选项:

```
RAISE [ EXCEPTION ] format USING option = expression [, ... ];
RAISE [ EXCEPTION ] condition_name USING option = expression [, ... ];
RAISE [ EXCEPTION ] SQLSTATE 'sqlstate' USING option = expression [, ... ];
RAISE [ EXCEPTION ] USING option = expression [, ... ];
```

其中, **option** 可以是以下选项:

- **MESSAGE**, 设置错误消息。如果 **RAISE** 语句中已经包含了 **format** 字符串,不能再使用该选项。
- **DETAIL**, 指定错误详细信息。
- **HINT**, 设置一个提示信息。
- **ERRCODE**, 指定一个错误码(**SQLSTATE**)。可以是文档中的条件名称或者五个字符组成的 **SQLSTATE** 代码。
- **COLUMN**、**CONSTRAINT**、**DATATYPE**、**TABLE**、**SCHEMA**, 返回相关对象的名称。

以下是一些示例:

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
    USING HINT = 'Please check your user ID';

RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
```

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';

RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

### 23.6.2 检查断言


PL/pgSQL 提供了 ASSERT 语句，用于调试存储过程和函数：

```
ASSERT condition [ , message ];
```

其中，condition 是一个布尔表达式；如果它的结果为真，ASSERT 通过；如果结果为假或者 NULL，将会抛出 ASSERT\_FAILURE 异常。message 用于提供额外的错误信息，默认为“assertion failed”。例如：

```
DO $$
DECLARE
    i integer := 1;
BEGIN
    ASSERT i = 0, 'i 的初始值应该为 0!';
END $$;

ERROR: i 的初始值应该为 0!
CONTEXT: PL/pgSQL function inline_code_block line 5 at ASSERT
```

 注意，ASSERT 只适用于代码调试；输出错误信息使用 RAISE 语句。

### 23.6.3 捕获异常

默认情况下，PL/pgSQL 遇到错误时会终止代码执行，同时撤销事务。我们也可以在代码块中使用 EXCEPTION 捕获错误并继续事务：

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
        handler_statements
    ... ]
END;
```

如果代码执行出错，程序将会进入 EXCEPTION 模块；依次匹配 condition，找到第一个匹配的分支并执行相应的 handler\_statements；如果没有找到任何匹配的分支，继续抛出错误。

以下是一个除零错误的示例：

```
DO $$
DECLARE
    i integer := 1;
BEGIN
    i := i / 0;
EXCEPTION
```

```

    WHEN division_by_zero THEN
        RAISE NOTICE '除零错误!';
    WHEN OTHERS THEN
        RAISE NOTICE '其他错误!';
END $$;

```

NOTICE: 除零错误!

**OTHERS** 用于捕获未指定的错误类型。

PL/pgSQL 还提供了捕获详细错误信息的 **GET STACKED DIAGNOSTICS** 语句，具体可以参考[官方文档](#)。

## 23.7 自定义函数

要创建一个自定义的 PL/pgSQL 函数，可以使用 **CREATE FUNCTION** 语句：

```

CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
[, ...] ] )
    RETURNS rettype
AS $$
DECLARE
    declarations
BEGIN
    statements;
    ...
END; $$
LANGUAGE plpgsql;

```

**CREATE** 表示创建函数，**OR REPLACE** 表示替换函数定义；**name** 是函数名；括号内是参数，多个参数使用逗号分隔；**argmode** 可以是 **IN**（输入）、**OUT**（输出）、**INOUT**（输入输出）或者 **VARIADIC**（数量可变），默认为 **IN**；**argname** 是参数名称；**argtype** 是参数的类型；**default\_expr** 是参数的默认值；**rettype** 是返回数据的类型；**AS** 后面是函数的定义，和上文中的匿名块相同；最后，**LANGUAGE** 指定函数实现的语言，也可以是其他过程语言。

以下示例创建一个函数 **get\_emp\_count**，用于返回指定部门中的员工数量：

```

CREATE OR REPLACE FUNCTION get_emp_count(p_deptid integer)
    RETURNS integer
AS $$
DECLARE
    ln_count integer;
BEGIN
    SELECT count(*) into ln_count
    FROM employees
    WHERE department_id = p_deptid;

    return ln_count;
END; $$
LANGUAGE plpgsql;

```

创建该函数之后，可以像内置函数一样在 SQL 语句中进行调用：

```
SELECT department_id, department_name, get_emp_count(department_id)
FROM departments d;
department_id|department_name      |get_emp_count|
-----|-----|-----|
          10|Administration      |          1|
          20|Marketing             |          2|
          30|Purchasing           |          6|
...
```

PL/pgSQL 函数支持重载 (Overloading)，也就是相同的函数名具有不同的函数参数。例如，以下语句创建一个重载的函数 `get_emp_count`，返回指定部门指定日期之后入职的员工数量：

```
CREATE OR REPLACE FUNCTION get_emp_count(p_deptid integer, p_hiredate date)
RETURNS integer
AS $$
DECLARE
    ln_count integer;
BEGIN
    SELECT count(*) into ln_count
    FROM employees
    WHERE department_id = p_deptid and hire_date >= p_hiredate;

    return ln_count;
END; $$
LANGUAGE plpgsql;
```

查询每个部门 2005 年之后入职的员工数量：

```
SELECT
department_id, department_name, get_emp_count(department_id), get_emp_count(department_id, '2005-01-01')
FROM departments d;
department_id|department_name      |get_emp_count|get_emp_count|
-----|-----|-----|-----|
          10|Administration      |          1|          0|
          20|Marketing             |          2|          1|
          30|Purchasing           |          6|          4|
...
```

我们再来看一个 **VARIADIC** 参数的示例：

```
CREATE OR REPLACE FUNCTION sum_num(
    VARIADIC nums numeric[])
RETURNS numeric
AS $$
DECLARE ln_total numeric;
BEGIN
    SELECT SUM(nums[i]) INTO ln_total
    FROM generate_subscripts(nums, 1) t(i);

    RETURN ln_total;
END; $$
LANGUAGE plpgsql;
```

参数 `nums` 是一个数组，可以传入任意多个参数；然后计算它们的和值。例如：

```
SELECT sum_num(1,2), sum_num(1,2,3);
sum_num|sum_num|
-----|-----|
      3|       6|
```

如果函数不需要返回结果，可以返回 `void` 类型；或者直接使用存储过程。

## 23.8 存储过程

PostgreSQL 11 增加了存储过程，使用 `CREATE PROCEDURE` 语句创建：

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
[, ...] ] )
AS $$
DECLARE
    declarations
BEGIN
    statements;
    ...
END; $$
LANGUAGE plpgsql;
```

存储过程的定义和函数主要的区别在于没有返回值，其他内容都类似。以下示例创建了一个存储过程 `update_emp`，用于修改员工的信息：

```
CREATE OR REPLACE PROCEDURE update_emp(
    p_empid in integer,
    p_salary in numeric,
    p_phone in varchar)
AS $$
BEGIN
    UPDATE employees
    set salary = p_salary,
        phone_number = p_phone
    WHERE employee_id = p_empid;
END; $$
LANGUAGE plpgsql;
```

调用存储过程使用 `CALL` 语句：

```
call update_emp(100, 25000, '515.123.4560');
```

### 23.8.1 事务管理

在存储过程内部，可以使用 `COMMIT` 或者 `ROLLBACK` 语句提交或者回滚事务。例如：

```
create table test(a int);

CREATE PROCEDURE transaction_test()
LANGUAGE plpgsql
AS $$
BEGIN
    FOR i IN 0..9 LOOP
```

```
        INSERT INTO test (a) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END
$$;

CALL transaction_test();
SELECT * FROM test;
a|
-|
0|
2|
4|
6|
8|
```

只有偶数才会被最终提交。

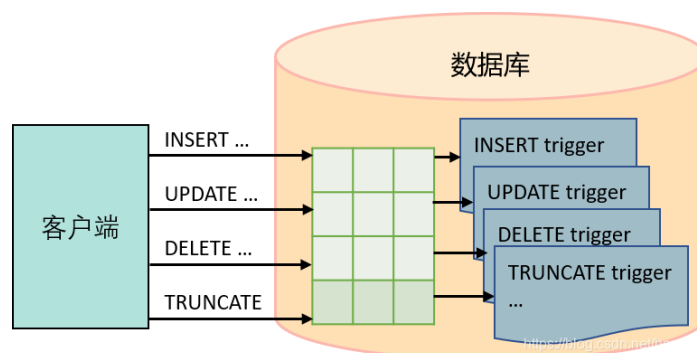
## 第 24 章 触发器

上一章我们介绍了如何利用 PL/pgSQL 过程语言实现存储过程和自定义函数。除此之外，PostgreSQL 自定义函数还可以用于实现另一种功能：**触发器**。

### 24.1 触发器概述

PostgreSQL 触发器 (trigger) 是一种特殊的函数，当某个数据变更事件 (INSERT、UPDATE、DELETE 或者 TRUNCATE 语句) 或者数据库事件 (DDL 语句) 发生时自动执行，而不是由用户或者应用程序进行调用。

基于某个表或者视图数据变更的触发器被称为**数据变更触发器** (DML 触发器)，基于数据库事件的触发器被称为**事件触发器** (DDL 触发器)。一般我们更多使用的是数据变更触发器。



对于数据变更触发器，PostgreSQL 支持两种级别的触发方式：**行级 (row-level) 触发器**和**语句级 (statement-level) 触发器**。这两者的区别在于触发的时机和触发次数。例如，对于一个影响 20 行数据的 UPDATE 语句，行级触发器将会触发 20 次，而语句级触发器只会触发 1 次。

触发器可以在事件发生之前 (BEFORE) 或者之后 (AFTER) 触发。如果在事件之前触发，它可以跳过针对当前行的修改，甚至修改被更新或插入的数据；如果在事件之后触发，触发器可以获得所有的变更结果。INSTEAD OF 触发器可以用于替换数据变更的操作，但只能基于视图定义。

下表列出了 PostgreSQL 中支持的各种触发器：

触发时机	触发事件	行级触发器	语句级触发器
BEFORE	INSERT、UPDATE、DELETE	表和外部表	表、视图和外部表
BEFORE	TRUNCATE	--	表
AFTER	INSERT、UPDATE、DELETE	表和外部表	表、视图和外部表
AFTER	TRUNCATE	--	表
INSTEAD OF	INSERT、UPDATE、DELETE	视图	--

INSTEAD OF	TRUNCATE	--	--
------------	----------	----	----

触发器对于多应用共享的数据库而言非常有用，可以将跨应用的功能存储在数据库中，当表中的数据发生任何变化时都会自动执行触发器的操作。例如，可以用触发器实现数据修改的历史审计，而不需要各种应用程序实现任何相关的逻辑。

另外，触发器还可以用于实现复杂的数据完整性和业务规则。例如，在非业务时间不允许修改用户的信息。

但是另一方面，触发器可能带来的问题就是在不清楚它们的存在和逻辑时可能会影响数据修改的结果和性能。

## 24.2 创建触发器

PostgreSQL 触发器的创建分为两步：

1. 使用 `CREATE FUNCTION` 语句创建一个触发器函数；
2. 使用 `CREATE TRIGGER` 语句将该函数与表进行关联。

首先，创建一个触发器函数：

```
CREATE [ OR REPLACE ] FUNCTION trigger_function ()
    RETURNS trigger
AS $$
DECLARE
    declarations
BEGIN
    statements;
    ...
END; $$
LANGUAGE plpgsql;
```

触发器函数与普通函数的区别在于它没有参数，并且返回类型为 `trigger`；触发器函数也可以使用其他过程语言，本书只涉及 `PL/pgSQL`。在触发器函数的内部，系统自动创建了许多特殊的变量：

- **NEW**，类型为 `RECORD`，代表了行级触发器 `INSERT`、`UPDATE` 操作之后的新数据行。  
对于 `DELETE` 操作或者语句级触发器而言，该变量为 `null`；
- **OLD**，类型为 `RECORD`，代表了行级触发器 `UPDATE`、`DELETE` 操作之前的旧数据行。  
对于 `INSERT` 操作或者语句级触发器而言，该变量为 `null`；
- **TG\_NAME**，触发器的名称；
- **TG\_WHEN**，触发的时机，例如 `BEFORE`、`AFTER` 或者 `INSTEAD OF`；
- **TG\_LEVEL**，触发器的级别，`ROW` 或者 `STATEMENT`；
- **TG\_OP**，触发的操作，`INSERT`、`UPDATE`、`DELETE` 或者 `TRUNCATE`；
- **TG\_RELID**，触发器所在表的 `oid`；
- **TG\_TABLE\_NAME**，触发器所在表的名称；
- **TG\_TABLE\_SCHEMA**，触发器所在表的模式；
- **TG\_NARGS**，创建触发器时传递给触发器函数的参数个数；



- **TG\_ARGV[]**, 创建触发器时传递给触发器函数的具体参数, 下标从 0 开始。非法的下标 (小于 0 或者大于等于 **tg\_nargs**) 将会返回空值。

然后, 我们使用 **CREATE TRIGGER** 语句创建一个触发器:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} {event [OR ...]}
ON table_name
[FOR [EACH] {ROW | STATEMENT}]
[WHEN ( condition ) ]
EXECUTE FUNCTION trigger_function;
```

其中, **event** 可以是 **INSERT**、**UPDATE**、**DELETE** 或者 **TRUNCATE**, **UPDATE** 支持特定字段 (**UPDATE OF col1, col2**) 的更新操作; 触发器可以在事件之前 (**BEFORE**) 或者之后 (**AFTER**) 触发, **INSTEAD OF** 只能用于替代视图上的 **INSERT**、**UPDATE** 或者 **DELETE** 操作; **FOR EACH ROW** 表示行级触发器, **FOR EACH STATEMENT** 表示语句级触发器; **WHEN** 用于指定一个额外的触发条件, 满足条件才会真正支持触发器函数。

接下来我们通过触发器来实现记录员工的信息变更历史, 首先创建一个历史记录表 **employees\_history**:

```
create table employees_history (
    id serial primary key,
    employee_id int null,
    first_name varchar(20) null,
    last_name varchar(25) null,
    email varchar(25) null,
    phone_number varchar(20) null,
    hire_date date null,
    job_id varchar(10) null,
    salary numeric(8,2) null,
    commission_pct numeric(2,2) null,
    manager_id int null,
    department_id int null,
    action_type varchar(10) not null,
    change_dt timestamp not null
);
```

然后定义一个触发器函数 **track\_employees\_change**:

```
create or replace function track_employees_change()
returns trigger as
$$
begin
    if tg_op = 'INSERT' then
        INSERT INTO employees_history(employee_id, first_name, last_name, email,
phone_number, hire_date, job_id, salary, commission_pct, manager_id,
department_id, action_type, change_dt)
        values(new.employee_id, new.first_name, new.last_name, new.email,
new.phone_number, new.hire_date, new.job_id, new.salary, new.commission_pct,
new.manager_id, new.department_id, 'INSERT', current_timestamp);
    elseif tg_op = 'UPDATE' then
        INSERT INTO employees_history(employee_id, first_name, last_name, email,
phone_number, hire_date, job_id, salary, commission_pct, manager_id,
department_id, action_type, change_dt)
        values(old.employee_id, old.first_name, old.last_name, old.email,
```

```

old.phone_number, old.hire_date, old.job_id, old.salary, old.commission_pct,
old.manager_id, old.department_id, 'UPDATE', current_timestamp);
    elsif tg_op = 'DELETE' then
        INSERT INTO employees_history(employee_id, first_name, last_name, email,
phone_number, hire_date, job_id, salary, commission_pct, manager_id,
department_id, action_type, change_dt)
            values(old.employee_id, old.first_name, old.last_name, old.email,
old.phone_number, old.hire_date, old.job_id, old.salary, old.commission_pct,
old.manager_id, old.department_id, 'DELETE', current_timestamp);
    end if;

    return new;
end; $$
language plpgsql;

```

该函数根据不同的操作记录了相应的历史信息、操作类型和操作时间。

最后创建一个触发器 `trg_employees_change`，将该函数与 `employees` 进行关联：

```

create trigger trg_employees_change
before insert or UPDATE or DELETE
on employees
for each row
execute function track_employees_change();

```

至此，我们完成了触发器的创建。接下来进行一些数据测试：

```

INSERT INTO employees(employee_id, first_name, last_name, email, phone_number,
hire_date, job_id, salary, commission_pct, manager_id, department_id)
values(207, 'Tony', 'Dong', 'TonyDong', '01066665678', '2020-05-25',
'IT_PROG', 6000, null, 103, 60);

SELECT * FROM employees_history;

```

id	employee_id	first_name	last_name	email	phone_number	hire_date	job_id	salary	commission_pct	manager_id	department_id	action_type	change_dt
1	207	Tony	Dong	TonyDong	01066665678	2020-05-25	IT_PROG	6000.00		103	60	INSERT	2020-05-25 15:45:17

我们往 `employees` 中插入一条记录之后，`employees_history` 记录了这一操作历史；对于 `UPDATE` 和 `DELETE` 操作也是如此。

## 24.3 管理触发器

PostgreSQL 提供了 `ALTER TRIGGER` 语句，用于修改触发器：

```

ALTER TRIGGER name ON table_name RENAME TO new_name;


```

这种方式目前只支持修改触发器的名称，修改触发器函数的方法和修改普通函数相同。

PostgreSQL 还支持触发器的禁用和启用：

```
ALTER TABLE table_name
{ENABLE | DISABLE} TRIGGER {trigger_name | ALL | USER};
```

默认创建的触发器处于启用状态；我们也可以使用以上语句禁用或者启用某个触发器、某个表上的所有触发器（ALL）或用户触发器（不包括内部生成的约束触发器，例如用于外键约束或延迟唯一性约束以及排除约束的触发器）。

 视图 `information_schema.triggers` 中存储了关于触发器的信息。

## 24.4 删除触发器

被禁用的触发器仍然存在，只是不会被触发；如果想要删除触发器，可以使用 `DROP TRIGGER` 语句：

```
DROP TRIGGER [IF EXISTS] trigger_name
ON table_name [RESTRICT | CASCADE];
```

`IF EXISTS` 可以避免触发器不存在时的错误提示；`CASCADE` 表示级联删除依赖于该触发器的对象，`RESTRICT` 表示如果存在依赖于该触发器的对象返回错误，默认为 `RESTRICT`。

我们将 `employees` 表上的触发器 `trg_employees_change` 删除：

```
drop trigger trg_employees_change on employees;
```

虽然删除了触发器，但是触发器函数 `track_employees_change` 仍然存在。

## 24.5 事件触发器

除了数据变更触发器之外，PostgreSQL 还提供了另一种触发器：事件触发器。事件触发器主要用于捕获全局的 DDL 事件，目前支持 `ddl_command_start`、`ddl_command_end`、`table_rewrite` 和 `sql_drop`，这些事件支持的完整语句可以参考[官方列表](#)。

对于事件触发器的函数而言，同样预定义了两个变量：

- **TG\_EVENT**，触发事件；
- **TG\_TAG**，触发语句。

对于事件触发器，首先也需要创建一个函数，返回类型为 `event_trigger`。例如：

```
create or replace function abort_any_command()
returns event_trigger
AS $$
begin
    if (user != 'postgres') then
        raise exception 'command % is disabled', tg_tag;
    end if;
end; $$
```

```
language plpgsql;
```

以上函数判断当前操作用户是否为超级用户（postgres），如果不是则不允许执行任何 DDL 语句。

接下来使用 CREATE EVENT TRIGGER 语句创建事件触发器：

```
create event trigger abort_ddl on ddl_command_start
execute function abort_any_command();
```

此时，如果使用非 postgres 用户执行 DDL 语句时将会返回错误：

```
hrdb=# SELECT user;
 user
-----
 tony
(1 row)

hrdb=# create table t(id int);
ERROR:  command CREATE TABLE is disabled
CONTEXT:  PL/pgSQL function abort_any_command() line 4 at RAISE
```

ALTER EVENT TRIGGER 语句可以启用/禁用事件触发器或者修改触发器的名称等：

```
ALTER EVENT TRIGGER name DISABLE;
ALTER EVENT TRIGGER name ENABLE;
ALTER EVENT TRIGGER name RENAME TO new_name;
```

DROP EVENT TRIGGER 语句可以用于删除事件触发器：

```
DROP EVENT TRIGGER [ IF EXISTS ] name [ CASCADE | RESTRICT ];
```

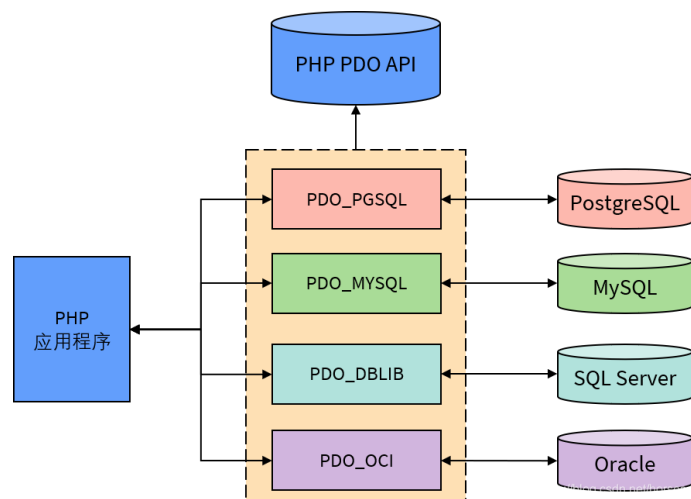
我们将事件触发器 abort\_ddl 删除：

```
DROP EVENT TRIGGER abort_ddl;
```

## 第 25 章 PHP 访问 PostgreSQL

[PHP](#) (Hypertext Preprocessor) 是一种流行的通用脚本语言，具有快速、灵活和实用等特性，特别适合于 Web 应用和网站开发。

[PHP 数据对象](#) (PDO) 定义了一个统一的 API 接口，用于在 PHP 程序中访问关系型数据库。不同数据库可以基于该标准实现特定的驱动程序，也可以实现专有的扩展功能。通过 PHP PDO 连接数据库的示意图如下：



本篇我们来介绍如何利用 PHP 数据对象接口连接和操作 PostgreSQL 数据库，包括创建和删除表、执行数据的 CRUD 操作等。

✍ 除此之外，在 PHP 中访问 PostgreSQL 数据库的另一种方法就是使用原生的 [PostgreSQL 扩展](#)。

### 25.1 连接数据库

如果使用 [LAMP](#)、[WAPP](#)、[MAPP](#) 工具栈安装集成开发环境，默认已经启用了 PDO\_PGSQL 驱动；如果是单独安装的 PHP 环境，需要在 php.ini 配置文件中增加该驱动。可以增加或者去掉下面两行配置项前面的注释符号（；）：

```
extension_dir = "php-install-path/ext"
extension=pdo_pgsql
```

首先在 web 根目录创建一个数据库的配置文件 db.ini：

```
host=192.168.56.104
port=5432
database=hrdb
user=tony
password=tony
```

将以上内容替换成你的数据库信息。然后创建一个用于连接数据库的文件 connection.php：

```

<?php
/**
 * 数据库连接
 */
class Connection {

    /**
     * Connection
     * @var type
     */
    private static $conn;

    /**
     * 连接数据库并返回一个 PDO 对象实例
     * @return PDO
     * @throws Exception
     */
    public function connect() {

        // 读取数据库配置参数文件
        $params = parse_ini_file('db.ini');
        if ($params === false) {
            throw new Exception("读取数据库配置参数文件错误！");
        }
        // connect to the postgresql database
        $conStr
sprintf("pgsql:host=%s;port=%d;dbname=%s;user=%s;password=%s",
        $params['host'],
        $params['port'],
        $params['database'],
        $params['user'],
        $params['password']);

        $pdo = new PDO($conStr);
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        return $pdo;
    }

    /**
     * 返回连接对象实例
     * @return type
     */
    public static function get() {
        if (null === static::$conn) {
            static::$conn = new static();
        }

        return static::$conn;
    }
}

```

我们创建了一个 **Connection** 类，用于连接数据库；其中 **connect()**方法通过读取配置文件，创建并返回了一个 **PDO** 对象。

接下来创建一个测试连接的文件 `test_connection.php`:

```
<?php

require 'connection.php';

try {
    Connection::get()->connect();
    echo '成功连接 PostgreSQL 数据库服务器!';
} catch (PDOException $e) {
    echo $e->getMessage();
}
```

通过 `require` 引用前面的 `connection.php`, 然后调用静态方法 `connect()` 获取连接。在浏览器中输入该文件的访问地址, 页面显示以下信息表示连接数据库成功:

```
成功连接 PostgreSQL 数据库服务器!
```

## 25.2 创建和删除表

PDO 对象的 `exec()` 方法可以用于执行 SQL 命令, 创建各种数据库对象。

我们先创建一个新的代码文件 `create_table.php`:

```
<?php

require 'connection.php';

try {
    $pdo = Connection::get()->connect();
    echo '成功连接 PostgreSQL 数据库服务器! <br>';

    $sql = 'create table if not exists users (
        id serial primary key,
        name character varying(10) not null unique,
        created_at timestamp not null
    );';

    $pdo->exec($sql);
    echo '成功创建表 users!';

} catch (PDOException $e) {
    echo $e->getMessage();
}
```

在浏览器中打开该文件地址, 返回以下信息:

```
成功连接 PostgreSQL 数据库服务器!
成功创建表 users!
```

此时, PostgreSQL 数据库中多了一个 `users` 表。在完成本篇的测试内容之后, 我们可以将 `$sql` 的内容换成 `DROP TABLE` 语句, 从而删除 `users` 表。

## 25.3 插入数据

在 PHP 应用中插入数据到 PostgreSQL 表中包含了以下几个步骤：

1. 创建一个新的 PDO 实例，通过 `connect()` 方法连接到数据库；
2. 构造一个 INSERT 语句，可以通过占位符（例如: `param1`）传递参数；
3. 调用 PDO 对象的 `prepare()` 方法返回一个预编译语句对象 `PDOStatement`；
4. 调用 `PDOStatement` 对象的 `bindValue()` 方法为参数传递数值；
5. 最后，调用 `PDOStatement` 对象的 `execute()` 方法执行 INSERT 语句。

我们创建一个新的 PHP 文件 `insert_user.php`：

```
<?php

require 'connection.php';

try {
    $pdo = Connection::get()->connect();
    echo '成功连接 PostgreSQL 数据库服务器! <br>';

    // 预编译插入语句
    $sql = 'INSERT INTO users(name, created_at) VALUES(:name,:createdAt)';
    $stmt = $pdo->prepare($sql);

    // 绑定参数值
    $name = 'tony';
    $createdAt = '2020-06-03 11:30:16';
    $stmt->bindValue(':name', $name);
    $stmt->bindValue(':createdAt', $createdAt);

    // 执行插入操作
    $stmt->execute();

    // 返回 id
    $id = $pdo->lastInsertId('users_id_seq');
    echo '插入数据成功, 用户 id: ' . $id . '<br>';

    // 绑定参数值
    $name = 'david';
    $createdAt = '2020-06-01 20:11:11';
    $stmt->bindValue(':name', $name);
    $stmt->bindValue(':createdAt', $createdAt);

    // 执行插入操作
    $stmt->execute();

    // 返回 id
    $id = $pdo->lastInsertId('users_id_seq');
    echo '插入数据成功, 用户 id: ' . $id . '<br>';

} catch (PDOException $e) {
    echo $e->getMessage();
}
```



我们执行了两次插入操作，并且通过 PDO 对象的 `lastInsertId()` 方法返回了插入数据的 `id`；`users_id_seq` 是 `users` 表中自增字段 `id` 对应的序列名称。

在浏览器中打开以上文件地址，返回信息如下：

```
成功连接 PostgreSQL 数据库服务器！
插入数据成功，用户 id: 1
插入数据成功，用户 id: 2
```

如果想要在页面显示 `users` 表中的数据，需要执行查询操作。

## 25.4 查询数据

在 PHP 应用中查询表中的数据包含了以下几个步骤：

1. 创建一个新的 PDO 实例，通过 `connect()` 方法连接到数据库；
2. 调用 PDO 对象的 `query()` 方法，传入一个查询语句文本，返回一个 `PDOStatement` 对象；
3. 调用 `PDOStatement` 对象的 `fetch()` 方法从查询结果中返回下一行数据。该方法的 `fetch_style` 参数决定了返回结果的方式。

下面我们创建一个新的文件 `get_users.php`，查询并显示用户信息：

```
<?php

require 'connection.php';

try {
    $pdo = Connection::get()->connect();

    // 执行查询语句
    $stmt = $pdo->query('SELECT id, name, created_at FROM users');
    $users = [];
    while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
        $users[] = [
            'id' => $row['id'],
            'name' => $row['name'],
            'createdAt' => $row['created_at']
        ];
    }

} catch (PDOException $e) {
    echo $e->getMessage();
}

?>

<!DOCTYPE html>
<html>
    <head>
        <style>
            table {
                font-family: arial, sans-serif;
                border-collapse: collapse;
                width: 100%;
```

```

    }

    td, th {
        border: 1px solid #dddddd;
        text-align: left;
        padding: 8px;
    }

    tr:nth-child(even) {
        background-color: #dddddd;
    }
</style>
<title>查询用户数据</title>
</head>
<body>
<h2>用户列表</h2>
<table>
    <thead>
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>CreatedAt</th>
        </tr>
    </thead>
    <tbody>
        <?php foreach ($users AS $user) : ?>
            <tr>
                <td><?php echo htmlspecialchars($user['id']) ?></td>
                <td><?php echo htmlspecialchars($user['name']); ?></td>
                <td><?php echo htmlspecialchars($user['createdAt']); ?></td>
            </tr>
        <?php endforeach; ?>
    </tbody>
</table>
</body>
</html>

```

首先，通过 `fetch()` 方法返回用户信息；参数 `PDO::FETCH_ASSOC` 表示返回一个数组，下标是字段的名称。然后在 **HTML** 中通过一个表格显示用户信息，返回的页面如下：

ID	Name	CreatedAt
1	tony	2020-06-03 11:30:16
2	david	2020-06-01 20:11:11

<https://blog.csdn.net/horses>

## 25.5 修改数据

修改数据的步骤和插入数据类似，只是将 `INSERT` 语句改成了 `UPDATE` 语句。我们创建一个新的文件 `update_user.php`：

```
<?php

require 'connection.php';

try {
    $pdo = Connection::get()->connect();

    // 预编译更新语句
    $sql = 'UPDATE users '
        . 'SET name = :name '
        . 'WHERE id = :id';
    $stmt = $pdo->prepare($sql);

    // 绑定参数值
    $name = 'tom';
    $id = 1;
    $stmt->bindValue(':name', $name);
    $stmt->bindValue(':id', $id);

    // 执行更新操作
    $stmt->execute();

    // 返回更新的行数
    $rowCount = $stmt->rowCount();
    echo '更新数据成功，更新记录数： ' . $rowCount . '<br>';

} catch (PDOException $e) {
    echo $e->getMessage();
}
```

在代码的最后通过 `PDOStatement` 对象的 `rowCount()`方法返回更新的记录数。在浏览器中打开以上文件地址，返回信息如下：

```
更新数据成功，更新记录数： 1
```

我们也可以通过 `get_users.php` 查看数据的变化。

## 25.6 删除数据

通过 PHP 删除数据的步骤和更新数据几乎相同。我们创建一个新的文件 `delete_user.php`：

```
<?php

require 'connection.php';

try {
```

```

    $pdo = Connection::get()->connect();

    // 预编译删除语句
    $sql = 'DELETE FROM users '
        . 'WHERE id = :id';
    $stmt = $pdo->prepare($sql);

    // 绑定参数值
    $id = 1;
    $stmt->bindValue(':id', $id);

    // 执行删除操作
    $stmt->execute();

    // 返回删除的行数
    $rowCount = $stmt->rowCount();
    echo '删除数据成功, 删除记录数: ' . $rowCount . '<br>';

} catch (PDOException $e) {
    echo $e->getMessage();
}

```

在浏览器中打开以上文件地址，返回信息如下：

```
删除数据成功, 删除记录数: 1
```

如果再次执行 `get_users.php`，用户列表中只返回一个记录。

## 25.7 管理事务

默认情况下，PostgreSQL 使用自动提交方式；也就是对于任何 SQL 语句，都会自动执行一次 **COMMIT** 操作。因此，前面的示例中我们没有进行任何的事务控制。

不过，在 PHP 应用中可以手动控制事务的提交和回滚。通过 PDO 对象的 `beginTransaction()`、`commit()` 和 `rollback()` 方法开始、提交和回滚一个事务。

我们创建一个新的文件 `transacion.php`：

```

<?php

require 'connection.php';

try {
    $pdo = Connection::get()->connect();
    echo '成功连接 PostgreSQL 数据库服务器! <br>';

    $pdo->beginTransaction();

    // 预编译插入语句
    $sql = 'INSERT INTO users(name, created_at) VALUES(:name,:createdAt)';
    $stmt = $pdo->prepare($sql);

    // 绑定参数值

```

```

$name = 'bob';
$createdAt = '2020-06-04 22:00:00';
$stmt->bindValue(':name', $name);
$stmt->bindValue(':createdAt', $createdAt);

// 执行插入操作
$stmt->execute();

// 返回 id
$id = $pdo->lastInsertId('users_id_seq');
echo '插入数据成功, 用户 id: ' . $id . '<br>';

// 预编译更新语句
$sql = 'UPDATE users '
      . 'SET name = :name '
      . 'WHERE id = :id';
$stmt = $pdo->prepare($sql);

// 绑定参数值
$name = 'david';
$stmt->bindValue(':name', $name);
$stmt->bindValue(':id', $id);

// 执行更新操作
$stmt->execute();

// 返回更新的行数
$rowCount = $stmt->rowCount();
echo '更新数据成功, 更新记录数: ' . $rowCount . '<br>';

$stmt->commit();
} catch (PDOException $e) {
    $pdo->rollBack();
    echo '执行操作失败, 回滚事务! ' . '<br>';
    echo $e->getMessage();
}

```

首先, 通过 `beginTransaction()` 方法开始一个事务; 然后分别执行插入和更新操作, 成功后提交事务; 如果出现异常, 回滚事务。执行该文件返回以下信息:

```

成功连接 PostgreSQL 数据库服务器!
插入数据成功, 用户 id: 4
执行操作失败, 回滚事务!
SQLSTATE[23505]: Unique violation: 7 ERROR: duplicate key value violates
unique constraint "users_name_key" DETAIL: Key (name)=(david) already exists.

```

错误消息显示 `name` 字段违反了唯一约束, 因为 `david` 已经存在。此时, 整个事务都被回滚, 成功插入的记录也不会存在。

## 25.8 调用存储过程

最后, 我们介绍一下如何在 PHP 中调用 PostgreSQL 存储过程和函数。我们在 PostgreSQL

创建一个存储过程 `add_user`:

```
CREATE OR REPLACE PROCEDURE add_user(pv_name varchar, pd_created_at
timestamp)
AS $$
BEGIN
    INSERT INTO users(name, created_at)
    values (pv_name, pd_created_at);
END; $$
LANGUAGE plpgsql;
```

`add_user` 用于增加一个用户。然后通过 **PHP** 调用该存储过程，创建一个新的文件 `stored_procedure.php`:

```
<?php

require 'connection.php';

try {
    $pdo = Connection::get()->connect();

    // 预编译语句
    $sql = 'call add_user(:name,:createdAt)';
    $stmt = $pdo->prepare($sql);

    // 绑定参数值
    $name = 'anne';
    $createdAt = '2020-06-04 08:08:08';
    $stmt->bindValue(':name', $name);
    $stmt->bindValue(':createdAt', $createdAt);

    // 执行操作
    $stmt->execute();

    // 返回 id
    $id = $pdo->lastInsertId('users_id_seq');
    echo '插入数据成功, 用户 id: ' . $id . '<br>';

} catch (PDOException $e) {
    echo $e->getMessage();
}
```

我们使用 `call` 命令调用存储过程，执行插入操作。在浏览器中输入该文件返回以下信息：

```
插入数据成功, 用户 id: 6
```

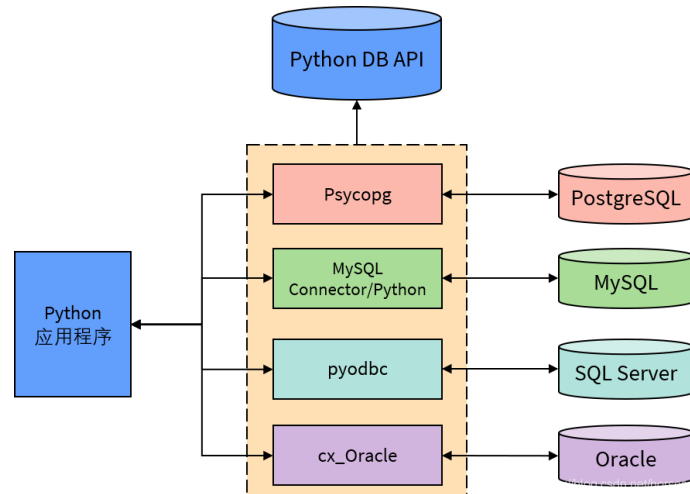
此时 `users` 表中增加了一条记录。

如果调用的是函数，可以通过 `fetch()`、`fetchColumn()`和 `fetchAll()`方法获取函数返回值。具体可以参考 [PDO 使用手册](#)。

## 第 26 章 Python 访问 PostgreSQL

[Python](#) 是一种高级、通用的解释型编程语言，以其优雅、准确、简单的语言特性，在云计算、Web 开发、自动化运维、数据科学以及机器学习等人工智能领域获得了广泛应用。

Python 定义了连接和操作数据库的标准接口 Python DB API。不同的数据库在此基础上实现了特定的驱动，这些驱动都实现了标准接口。



对于 PostgreSQL 数据库，最常见的 Python 驱动程序就是 [psycopg](#)，它完全实现了 [Python DB-API 2.0](#) 接口规范。接下来我们介绍如何通过 psycopg 连接和操作 PostgreSQL 数据库。

### 26.1 连接数据库

首先，我们需要安装 Python 和 psycopg 驱动。Python 可以通过官方网站下载，安装之后可以通过以下命令查看版本信息：

```
PS C:\Users\dongx> python -V
Python 3.10.3
```

然后通过 pip 安装最新的 psycopg：

```
PS C:\Users\dongx> pip install --upgrade psycopg2
Collecting psycopg2
  Using cached psycopg2-2.8.5-cp38-cp38-win_amd64.whl (1.1 MB)
Installing collected packages: psycopg2
Successfully installed psycopg2-2.8.5
```

为了方便开发，我们安装一个 IDE（集成开发环境）：JetBrains 出品的 Pycharm Community Edition。在 Pycharm 中新建一个项目 PythonPostgreSQL，然后创建一个数据库连接的配置文件 dbconfig.ini，添加以下内容：

```
[postgresql]
host = 192.168.56.104
port = 5432
database = hrdb
```

```
user = tony
password = tony
```

配置文件中存储了数据库的连接信息：主机、端口、数据库、用户以及密码；我们需要按照自己的环境进行配置。

然后，新建一个测试数据库连接的 Python 文件 `postgresql_connection.py`：

```
# 导入 psycopg2 模块和 Error 对象
import psycopg2
from psycopg2 import DatabaseError
from configparser import ConfigParser

def read_db_config(filename='dbconfig.ini', section='postgresql'):
    """ 读取数据库配置文件，返回一个字典对象 """
    # 创建解析器，读取配置文件
    parser = ConfigParser()
    parser.read(filename)

    # 获取 postgresql 部分的配置
    db = {}
    if parser.has_section(section):
        items = parser.items(section)
        for item in items:
            db[item[0]] = item[1]
    else:
        raise Exception('文件 {1} 中未找到 {0} 配置信息！'.format(section,
filename))

    return db

db_config = read_db_config()
connection = None

try:
    # 使用 psycopg2.connect 方法连接 PostgreSQL 数据库
    connection = psycopg2.connect(**db_config)

    # 创建一个游标
    cur = connection.cursor()

    # 获取 PostgreSQL 版本号
    cur.execute('SELECT version()')
    db_version = cur.fetchone()

    # 输出 PostgreSQL 版本
    print("连接成功，PostgreSQL 服务器版本: ", db_version)`在这里插入代码片`

    # 关闭游标
    cur.close()
except (Exception, DatabaseError) AS e:
    print("连接 PostgreSQL 失败: ", e)
finally:
    # 释放数据库连接
    if connection is not None:
```



```
connection.close()
print("PostgreSQL 数据库连接已关闭。")
```

- 首先，我们导入了 `psycopg2` 驱动和解析配置文件的 `configparser` 模块；
- 然后，创建一个读取配置文件的 `read_db_config` 函数；
- 接下来调用 `psycopg2.connect` 函数创建一个新的数据库连接；
- 然后通过连接对象的 `cursor` 函数创建一个新的游标，并且执行查询语句返回数据库的版本；
- 在此之后，调用游标对象的 `fetchone()` 方法获取返回结果并打印信息；
- 最后，调用 `close()` 方法关闭游标资源和数据库连接对象。

执行以上脚本，返回的信息如下：

```
连接成功，PostgreSQL 服务器版本： ('PostgreSQL 12.3 on x86_64-pc-linux-gnu,
compiled by gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-39), 64-bit',)
PostgreSQL 数据库连接已关闭。
```

## 26.2 创建和删除表

建立数据库连接之后，通过执行 `CREATE TABLE` 和 `DROP TABLE` 语句可以创建和删除数据表。我们创建一个新的 Python 文件 `postgresql_ddl.py`：

```
# 导入 psycopg2 模块和 Error 对象
import psycopg2
from psycopg2 import DatabaseError
from configparser import ConfigParser

def read_db_config(filename='dbconfig.ini', section='postgresql'):
    """ 读取数据库配置文件，返回一个字典对象
    """
    # 创建解析器，读取配置文件
    parser = ConfigParser()
    parser.read(filename)

    # 获取 postgresql 部分的配置
    db = {}
    if parser.has_section(section):
        items = parser.items(section)
        for item in items:
            db[item[0]] = item[1]
    else:
        raise Exception('文件 {1} 中未找到 {0} 配置信息！'.format(section,
filename))

    return db

db_config = read_db_config()
connection = None

try:
    # 使用 psycopg2.connect 方法连接 PostgreSQL 数据库
```

```

connection = psycopg2.connect(**db_config)

# 创建一个游标
cur = connection.cursor()

# 定义 SQL 语句
sql = """ create table users (
            id serial primary key,
            name character varying(10) not null unique,
            created_at timestamp not null
        ) """

# 执行 SQL 命令
cur.execute(sql)

# 关闭游标
cur.close()

# 提交事务
connection.commit()
print("操作成功! ")
except (Exception, DatabaseError) AS e:
    print("操作失败: ", e)
finally:
    # 释放数据库连接
    if connection is not None:
        connection.close()
        print("PostgreSQL 数据库连接已关闭。")

```

同样是先连接数据库；然后利用游标对象的 `execute()` 方法执行 SQL 命令创建表；`commit` 方法用于提交事务修改，如果不执行该操作不会真正创建表，因为 `psycopg2` 连接 PostgreSQL 默认不会自动提交（`autocommit`）。执行该脚本的结果如下：

```

操作成功！
PostgreSQL 数据库连接已关闭。

```

如果 `user` 表已经存在，将会返回以下错误：

```

操作失败: relation "users" already exists

```

我们可以将文件中的 `sql` 语句修改成“`drop table users`”，删除 `users` 表并重建创建。

## 26.3 插入数据

PostgreSQL 使用 `INSERT` 语句插入数据；Python 中游标对象的 `execute()` 方法用于执行 SQL 语句，该方法可以接收参数实现预编译语句。

```

# 导入 psycopg2 模块和 Error 对象
import psycopg2
from psycopg2 import DatabaseError
from configparser import ConfigParser

def read_db_config(filename='dbconfig.ini', section='postgresql'):

```

```

""" 读取数据库配置文件，返回一个字典对象
"""
# 创建解析器，读取配置文件
parser = ConfigParser()
parser.read(filename)

# 获取 postgresql 部分的配置
db = {}
if parser.has_section(section):
    items = parser.items(section)
    for item in items:
        db[item[0]] = item[1]
else:
    raise Exception('文件 {1} 中未找到 {0} 配置信息!'.format(section,
filename))

return db

db_config = read_db_config()
connection = None

try:
    # 使用 psycopg2.connect 方法连接 PostgreSQL 数据库
    connection = psycopg2.connect(**db_config)

    # 创建一个游标
    cur = connection.cursor()

    # 定义 SQL 语句
    sql = """ INSERT INTO users(name, created_at)
                values (%s, %s) RETURNING id
            """

    # 执行 SQL 命令
    cur.execute(sql, ('tony', '2020-06-08 11:00:00'))

    # 获取 id
    id = cur.fetchone()[0]

    # 提交事务
    connection.commit()
    print("操作成功! 用户 id: ", id)

    # 关闭游标
    cur.close()
except (Exception, DatabaseError) AS e:
    print("操作失败: ", e)
finally:
    # 释放数据库连接
    if connection is not None:
        connection.close()
        print("PostgreSQL 数据库连接已关闭。")

```

sql 变量中的百分号(%)是占位符，这些占位符的值在 execute()方法中进行替换；游标对象的 fetchone 方法用于返回一行结果，这里用于获取 RETURNING id 返回的用户 id。执行以上

脚本返回的结果如下：

```
操作成功！ 用户 id: 1
PostgreSQL 数据库连接已关闭。
```

如果想要查看插入 `users` 表中的数据，可以执行查询操作。

## 26.4 查询数据

游标对象提供了三种获取返回结果的方法：`fetchone()`方法获取下一行数据，`fetchmany(size=cursor.arraysize)`方法获取下一组数据行，`fetchall()`方法返回全部数据行。

我们创建一个新的文件 `postgresql_query.py`：

```
# 导入 psycopg2 模块和 Error 对象
import psycopg2
from psycopg2 import DatabaseError
from configparser import ConfigParser

def read_db_config(filename='dbconfig.ini', section='postgresql'):
    """ 读取数据库配置文件，返回一个字典对象 """
    # 创建解析器，读取配置文件
    parser = ConfigParser()
    parser.read(filename)

    # 获取 postgresql 部分的配置
    db = {}
    if parser.has_section(section):
        items = parser.items(section)
        for item in items:
            db[item[0]] = item[1]
    else:
        raise Exception('文件 {1} 中未找到 {0} 配置信息！'.format(section,
filename))

    return db

db_config = read_db_config()
connection = None

try:
    # 使用 psycopg2.connect 方法连接 PostgreSQL 数据库
    connection = psycopg2.connect(**db_config)

    # 创建一个游标
    cur = connection.cursor()

    # 定义 SQL 语句
    sql = """ SELECT id, name, created_at
                FROM users
            """
```

```

# 执行 SQL 命令
cur.execute(sql)
print("用户数量: ", cur.rowcount)

# 获取结果
user = cur.fetchone()
while user is not None:
    print(user)
    user = cur.fetchone()

# 关闭游标
cur.close()
except (Exception, DatabaseError) AS e:
    print("操作失败: ", e)
finally:
    # 释放数据库连接
    if connection is not None:
        connection.close()

```

游标对象的 `rowcount` 属性代表了返回的数据行数，`fetchone()`方法返回一行数据或者 `None`，`while` 循环用于遍历和打印查询结果。由于 `users` 表中目前只有一行数据，执行以上文件的结果如下：

```

用户数量:  1
(1, 'tony', datetime.datetime(2020, 6, 8, 11, 0))

```

## 26.5 修改数据

修改数据的流程与插入数据相同，只是需要将 `INSERT` 语句替换成 `UPDATE` 语句。我们创建一个新的文件 `postgresql_update.py`：

```

# 导入 psycopg2 模块和 Error 对象
import psycopg2
from psycopg2 import DatabaseError
from configparser import ConfigParser

def read_db_config(filename='dbconfig.ini', section='postgresql'):
    """ 读取数据库配置文件，返回一个字典对象 """
    # 创建解析器，读取配置文件
    parser = ConfigParser()
    parser.read(filename)

    # 获取 postgresql 部分的配置
    db = {}
    if parser.has_section(section):
        items = parser.items(section)
        for item in items:
            db[item[0]] = item[1]
    else:
        raise Exception('文件 {1} 中未找到 {0} 配置信息！'.format(section,

```

```

filename))

    return db

db_config = read_db_config()
connection = None

try:
    # 使用 psycopg2.connect 方法连接 PostgreSQL 数据库
    connection = psycopg2.connect(**db_config)

    # 创建一个游标
    cur = connection.cursor()

    # 定义 SQL 语句
    sql = """ UPDATE users
                set name = %s
                WHERE id = %s
            """

    # 执行 SQL 命令
    cur.execute(sql, ('tom', 1))

    # 获取 id
    rows = cur.rowcount

    # 提交事务
    connection.commit()
    print("操作成功! 更新行数: ", rows)

    # 再次查询数据
    sql = """ SELECT id, name, created_at
                FROM users WHERE id = 1
            """

    cur.execute(sql)
    user = cur.fetchone()
    print(user)

    # 关闭游标
    cur.close()
except (Exception, DatabaseError) AS e:
    print("操作失败: ", e)
finally:
    # 释放数据库连接
    if connection is not None:
        connection.close()

```

更新数据之后，再次执行了查询语句，返回更新后的用户信息。执行该文件的结果如下：

```

操作成功! 更新行数:  1
(1, 'tom', datetime.datetime(2020, 6, 8, 11, 0))

```

## 26.6 删除数据

将 UPDATE 语句替换成 DELETE 语句，就可以删除表中的数据。我们创建一个新的文件 postgresql\_delete.py:

```
# 导入 psycopg2 模块和 Error 对象
import psycopg2
from psycopg2 import DatabaseError
from configparser import ConfigParser

def read_db_config(filename='dbconfig.ini', section='postgresql'):
    """ 读取数据库配置文件，返回一个字典对象
    """
    # 创建解析器，读取配置文件
    parser = ConfigParser()
    parser.read(filename)

    # 获取 postgresql 部分的配置
    db = {}
    if parser.has_section(section):
        items = parser.items(section)
        for item in items:
            db[item[0]] = item[1]
    else:
        raise Exception('文件 {1} 中未找到 {0} 配置信息!'.format(section,
filename))

    return db

db_config = read_db_config()
connection = None

try:
    # 使用 psycopg2.connect 方法连接 PostgreSQL 数据库
    connection = psycopg2.connect(**db_config)

    # 创建一个游标
    cur = connection.cursor()

    # 定义 SQL 语句
    sql = """ DELETE FROM users WHERE id = %s
    """

    # 执行 SQL 命令
    cur.execute(sql, (1,))
    rows = cur.rowcount

    # 提交事务
    connection.commit()
    print("操作成功! 删除行数: ", rows)

    # 关闭游标
    cur.close()
```

```

except (Exception, DatabaseError) AS e:
    print("操作失败: ", e)
finally:
    # 释放数据库连接
    if connection is not None:
        connection.close()

```

执行该文件，返回以下结果：

```
操作成功！ 删除行数：1
```

## 26.7 管理事务

在前面的示例中，需要使用 `connection.commit()` 提交对 PostgreSQL 数据库执行的修改，这是因为 `psycopg2` 默认没有打开自动提交功能。我们也可以利用连接对象的 `autocommit` 属性设置是否自动提交。

将上文中的 `postgresql_insert.py` 修改如下：

```

# 导入 psycopg2 模块和 Error 对象
import psycopg2
from psycopg2 import DatabaseError
from configparser import ConfigParser

def read_db_config(filename='dbconfig.ini', section='postgresql'):
    """ 读取数据库配置文件，返回一个字典对象 """
    # 创建解析器，读取配置文件
    parser = ConfigParser()
    parser.read(filename)

    # 获取 postgresql 部分的配置
    db = {}
    if parser.has_section(section):
        items = parser.items(section)
        for item in items:
            db[item[0]] = item[1]
    else:
        raise Exception('文件 {1} 中未找到 {0} 配置信息！'.format(section,
filename))

    return db

db_config = read_db_config()
connection = None

try:
    # 使用 psycopg2.connect 方法连接 PostgreSQL 数据库
    connection = psycopg2.connect(**db_config)

    # 打印和设置自动提交
    print('默认 autocommit: ', connection.autocommit)

```



```

connection.autocommit = True
print('新的 autocommit: ', connection.autocommit)

# 创建一个游标
cur = connection.cursor()

# 定义 SQL 语句
sql = """ INSERT INTO users(name, created_at)
          values (%s, %s) RETURNING id
          """

# 执行 SQL 命令
cur.execute(sql, ('tony', '2020-06-08 11:00:00'))

# 获取 id
id = cur.fetchone()[0]

print("操作成功! 用户 id: ", id)

# 关闭游标
cur.close()
except (Exception, DatabaseError) AS e:
    print("操作失败: ", e)
finally:
    # 释放数据库连接
    if connection is not None:
        connection.close()
        print("PostgreSQL 数据库连接已关闭。")

```

通过 `connection.autocommit` 设置了自动提交, 所以 `INSERT` 语句插入数据之后不需要再执行 `commit` 操作。

```

默认 autocommit: False
新的 autocommit: True
操作成功! 用户 id: 2
PostgreSQL 数据库连接已关闭。

```

如果一个事务中包含多个数据库操作, 还是应该在事务的最后统一执行提交, 并且在异常处理部分通过连接对象的 `rollback()` 方法回滚部分完成的事务。

另一种管理事务的方法是使用 `with` 语句, 这样可以避免手动的资源管理和事务操作。我们创建一个新的文件 `postgresql_transaction.py`:

```

# 导入 psycopg2 模块和 Error 对象
import psycopg2
from psycopg2 import DatabaseError
from configparser import ConfigParser

def read_db_config(filename='dbconfig.ini', section='postgresql'):
    """ 读取数据库配置文件, 返回一个字典对象
    """
    # 创建解析器, 读取配置文件
    parser = ConfigParser()
    parser.read(filename)

    # 获取 postgresql 部分的配置

```

```

    db = {}
    if parser.has_section(section):
        items = parser.items(section)
        for item in items:
            db[item[0]] = item[1]
    else:
        raise Exception('文件 {1} 中未找到 {0} 配置信息!'.format(section,
filename))

    return db

db_config = read_db_config()
connection = None

try:
    # 使用 with 语句管理事务
    with psycopg2.connect(**db_config) AS connection:

        # 创建一个游标
        with connection.cursor() AS cur:

            # 插入数据
            sql = """ INSERT INTO users(name, created_at)
                        values (%s, %s)
                    """
            cur.execute(sql, ('Jason', '2020-06-08 15:30:00'))

            # 更新数据
            sql = """ UPDATE users
                        set created_at = %s
                        WHERE name = %s
                    """
            cur.execute(sql, ('2020-06-08 16:00:00', 'tony'))

            sql = """ SELECT id, name, created_at
                        FROM users
                    """

            # 查询数据
            cur.execute(sql)

            # 获取结果
            user = cur.fetchone()
            while user is not None:
                print(user)
                user = cur.fetchone()
except (Exception, DatabaseError) AS e:
    print("操作失败: ", e)
finally:
    # 释放数据库连接
    if connection is not None:
        connection.close()

```

整个事务包含插入数据、更新数据以及查询数据三个操作。

执行该脚本的结果如下：

```
(3, 'Jason', datetime.datetime(2020, 6, 8, 15, 30))
(2, 'tony', datetime.datetime(2020, 6, 8, 16, 0))
```

## 26.8 调用存储函数

游标对象的 `callproc()` 方法可以用于执行存储函数。我们先创建一个返回用户数量的函数 `get_user_count`:

```
CREATE OR REPLACE FUNCTION get_user_count()
RETURNS INT
AS $$
DECLARE
ln_count INT;
BEGIN
    SELECT count(*) INTO ln_count
    FROM users;

    RETURN ln_count;
END; $$
LANGUAGE plpgsql;
```

接下来创建一个新的文件 `postgresql_func.py`:

```
# 导入 psycopg2 模块和 Error 对象
import psycopg2
from psycopg2 import DatabaseError
from configparser import ConfigParser

def read_db_config(filename='dbconfig.ini', section='postgresql'):
    """ 读取数据库配置文件，返回一个字典对象
    """
    # 创建解析器，读取配置文件
    parser = ConfigParser()
    parser.read(filename)

    # 获取 postgresql 部分的配置
    db = {}
    if parser.has_section(section):
        items = parser.items(section)
        for item in items:
            db[item[0]] = item[1]
    else:
        raise Exception('文件 {1} 中未找到 {0} 配置信息!'.format(section,
filename))

    return db

db_config = read_db_config()
connection = None

try:
    # 使用 WITH 语句管理事务
    with psycopg2.connect(**db_config) AS connection:
```

```
# 创建一个游标
with connection.cursor() AS cur:

    # 调用存储函数
    cur.callproc('get_user_count')

    row = cur.fetchone()[0]
    print('用户总数: ', row)
except (Exception, DatabaseError) AS e:
    print("操作失败: ", e)
finally:
    # 释放数据库连接
    if connection is not None:
        connection.close()
```

`callproc()`方法调用存储函数也可以写成以下等价的形式:

```
cur.execute('SELECT * FROM get_user_count()')
```

执行以上脚本返回的结果如下:

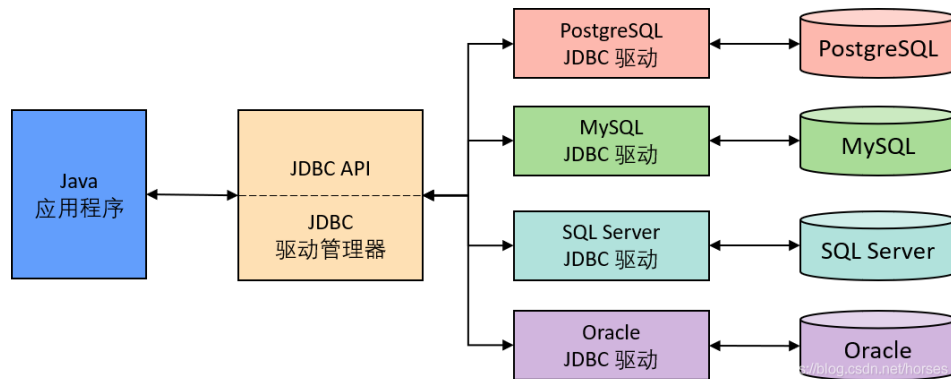
```
用户总数:  2
```

`callproc()`方法不支持存储过程, 可以使用 `execute()`方法调用 PostgreSQL 中的 `CALL` 命令执行存储过程。

更多关于 Psycpg 接口的使用和配置, 可以参考 [Psycpg 文档](#)。

## 第 27 章 Java 访问 PostgreSQL

JDBC (Java Database Connectivity) 是 Java 语言中用于访问数据库的应用程序接口 (API)。JDBC 提供了操作关系数据库的标准方法, 属于 Java Standard Edition 平台的一部分。以下是 Java 应用程序通过 JDBC 访问数据库的示意图:



其中各个模块的作用如下:

- Java 应用程序由开发人员编码完成, 用于实现业务处理的逻辑和流程;
- JDBC API 提供了统一的接口和驱动管理, 实现了应用程序和 JDBC 驱动隔离。同一套应用代码只需要切换驱动程序就可以支持不同的数据库;
- JDBC 驱动实现了 JDBC API 中定义的接口, 用于与不同的数据库进行交互;
- 数据库提供数据的存储管理和访问控制。

本章介绍如何通过 JDBC 连接 PostgreSQL 数据库并执行各种数据操作。

### 27.1 连接数据库

首先, 我们需要准备开发环境。开发环境分为三个部分: Java JDK、PostgreSQL 数据库以及 JDBC 驱动。

开发 Java 应用需要使用 JDK (Java Development Kit); 使用 `java -version` 命令检查是否已经安装 JDK, 要求至少 JDK 1.8.0 版本以上。

```
C:\Users\dongx>java -version
java version "13.0.1" 2019-10-15
Java(TM) SE Runtime Environment (build 13.0.1+9)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)
```

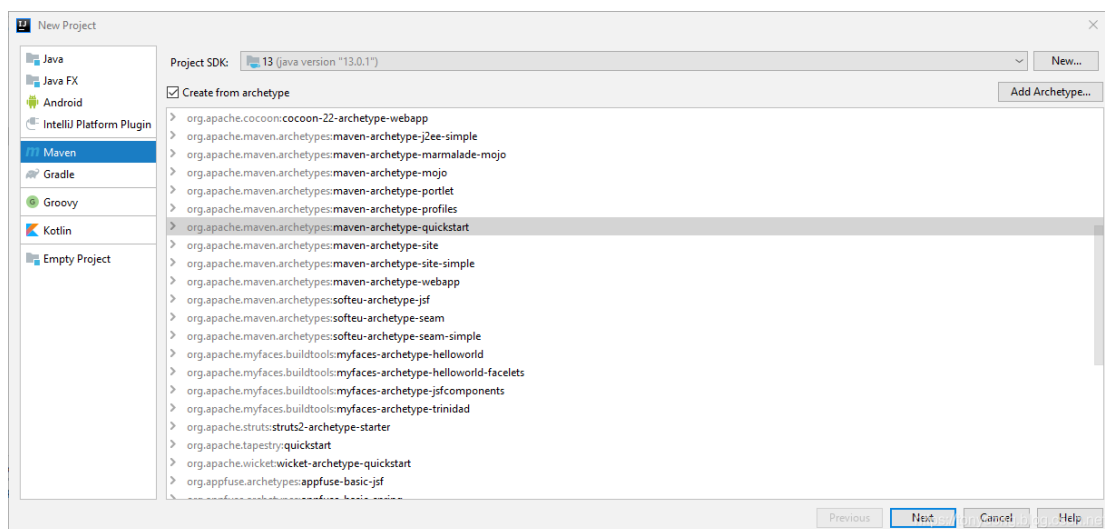
如果显示无法识别以上命令, 表示没有安装 JDK。可以通过 Oracle 官方 [Java SE JDK 地址](#) 下载并安装。安装完成之后, 还需要设置环境变量 (Windows):

- 右键单击“我的电脑”, 然后选择“属性”。在“高级”选项卡上, 选择“环境变量”, 然后新建环境变量“JAVA\_HOME”, 变量值为 JDK 的安装目录, 例如“C:\Program Files\Java\jdk-13.0.1”。

- 编辑环境变量“Path”，将以下内容追加到最后：%JAVA\_HOME%\bin\。

PostgreSQL 数据库的安装可以参考第 1 章。

最后是安装 PostgreSQL 的 JDBC 驱动。为了方便程序开发，推荐安装一个 IDE（集成开发环境），我们使用 JetBrains 出品的 [IntelliJ IDEA](#) 社区版。IntelliJ IDEA 可以使用 Maven 管理包的依赖，我们创建一个新的 Maven 项目（基于项目模板）：



然后在项目的 pom.xml 文件的<dependencies>节点中添加以下内容：

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.2.14</version>
</dependency>
```

Maven 会自动下载 PostgreSQL JDBC 驱动并且进行配置。

我们在项目目录中创建一个数据库的连接配置文件 db.properties，内容如下：

```
url=jdbc:postgresql://192.168.56.104:3306/hrdb
user=tony
password=123456
```

其中，url 中指定了数据库的 IP 地址、端口以及目标数据库；user 和 password 分别为连接数据库的用户和密码。

然后将项目默认创建的 App.java 文件修改如下：

```
package org.example;

// 导入 JDBC 和 IO 包
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;

public class App
{
    public static void main( String[] args )
```

```

    {
        String url = null;
        String user = null;
        String password = null;

        // 读取数据库连接配置文件
        try (FileInputStream file = new FileInputStream("db.properties")) {

            Properties pros = new Properties();
            pros.load(file);
            url = pros.getProperty("url");
            user = pros.getProperty("user");
            password = pros.getProperty("password");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

        // 建立数据库连接
        try (Connection conn = DriverManager.getConnection(url, user,
password)) {
            System.out.println("连接 PostgreSQL 数据库成功!");
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

首先,通过一个 `FileInputStream` 对象读取数据库的连接配置文件;然后使用 `try-with-resources` 方式建立数据库连接,打印信息。运行该程序的结果如下:

```
连接 PostgreSQL 数据库成功!
```

## 27.2 创建和删除表

通过 JDBC 连接数据库并执行 DDL 语句的过程如下:

1. 利用 `DriverManager` 类的 `getConnection()` 方法获取一个 `Connection` 连接对象;
2. 使用连接对象的 `createStatement()` 方法创建一个 `Statement` 语句对象;
3. 利用语句对象的 `execute()` 方法执行 SQL 语句;
4. 释放 `Statement` 以及 `Connection` 对象资源。

创建一个新的 Java 文件 `PostgreSQLDDL.java`, 内容如下:

```

package org.example;

// 导入 JDBC 和 IO 包
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;

public class PostgreSQLDDL {

```

```

public static void main(String[] args )
{
    String url = null;
    String user = null;
    String password = null;
    String sql_str = "create table users (" +
        " id serial primary key," +
        " name character varying(10) not null unique," +
        " created_at timestamp not null" +
        ")";

    // 读取数据库连接配置文件
    try (FileInputStream file = new FileInputStream("db.properties")) {

        Properties p = new Properties();
        p.load(file);
        url = p.getProperty("url");
        user = p.getProperty("user");
        password = p.getProperty("password");
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }

    // 建立数据库连接, 创建查询语句, 并且执行语句
    try (Connection conn = DriverManager.getConnection(url, user,
password);
        Statement ps = conn.createStatement()) {

        // 执行 SQL 语句
        ps.execute(sql_str);
        System.out.println("成功创建 users 表!");

    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}

```

首先,通过一个 `FileInputStream` 对象读取数据库的连接配置文件;然后使用 `try-with-resources` 方式建立数据库连接, 创建查询语句, 并且执行语句创建 `users` 表。运行该程序的结果如下:

成功创建 users 表!

如果 `users` 表已经存在, 将会提示以下错误:

ERROR: relation "users" already exists

此时可以将 `sql_str` 的内容修改 `drop table users` 删除并重新创建 `users` 表。

## 27.3 插入数据

通过 JDBC 连接数据库并执行插入操作的过程如下:



1. 利用 `DriverManager` 类的 `getConnection()`方法获取一个 `Connection` 连接对象;
2. 使用连接对象的 `createStatement()`方法创建一个 `Statement` 或者 `PreparedStatement` 语句对象;
3. 利用语句对象的 `execute()`或者 `executeBatch()`方法执行 `INSERT` 语句;
4. 释放 `Statement` 以及 `Connection` 对象资源。

创建一个新的 Java 文件 `PostgreSQLInsert.java`, 内容如下:

```
package org.example;

// 导入 JDBC 和 IO 包
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;
import java.sql.Timestamp;

public class PostgreSQLInsert {
    public static void main(String[] args )
    {
        String url = null;
        String user = null;
        String password = null;
        String sql_str = "INSERT INTO users(name, created_at) values(?, ?)";

        // 读取数据库连接配置文件
        try (FileInputStream file = new FileInputStream("db.properties")) {

            Properties p = new Properties();
            p.load(file);
            url = p.getProperty("url");
            user = p.getProperty("user");
            password = p.getProperty("password");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

        // 建立数据库连接, 创建预编译语句, 并且执行语句
        try (Connection conn = DriverManager.getConnection(url, user,
password);
            PreparedStatement ps = conn.prepareStatement(sql_str)) {

            // 设置输入参数
            ps.setString(1, "tony");
            ps.setTimestamp(2, new Timestamp(System.currentTimeMillis()));
            ps.addBatch();

            ps.setString(1, "david");
            ps.setTimestamp(2, new Timestamp(System.currentTimeMillis()));
            ps.addBatch();


            // 执行批量插入操作
            ps.executeBatch();
            System.out.println("插入数据成功!");
        }
    }
}
```

```

        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

其中，`sql_str` 中的两个问号(?)表示两个占位符，它们的值会在运行时进行替换；然后使用 `prepareStatement()` 方法创建了一个预编译的 SQL 语句，在执行该语句之前使用 `setString()` 和 `setTimestamp()` 方法替换占位符的值，并且使用 `addBatch()` 添加批量操作；最后执行 `executeBatch()` 方法进行批量插入操作。

 预编译语句可以避免 SQL 语句的重复编译，使用不同的参数多次运行语句可以提高效率，并且能够预防 SQL 注入。

该程序的执行结果如下：

插入数据成功！

如果想要查看被插入的数据，需要执行查询语句。

## 27.4 查询数据

通过 JDBC 连接数据库并执行查询语句的过程如下：

1. 利用 `DriverManager` 类的 `getConnection()` 方法获取一个 `Connection` 连接对象；
2. 使用连接对象的 `createStatement()` 方法创建一个 `Statement` 或者 `PreparedStatement` 语句对象；
3. 利用语句对象的 `executeQuery()` 方法执行 SQL 语句或者存储过程，返回一个 `ResultSet` 结果集对象；
4. 遍历结果集，获取并处理查询结果；
5. 释放 `ResultSet`、`Statement` 以及 `Connection` 对象资源。

新建一个 Java 文件 `PostgreSQLQuery.java`，内容如下：

```

package org.example;

// 导入 JDBC 和 IO 包
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;

public class PostgreSQLQuery {
    public static void main( String[] args )
    {
        String url = null;
        String user = null;
        String password = null;
        String sql_str = "SELECT id, name, created_at FROM users";

        // 读取数据库连接配置文件

```

```

        try (FileInputStream file = new FileInputStream("db.properties")) {

            Properties pros = new Properties();
            pros.load(file);
            url = pros.getProperty("url");
            user = pros.getProperty("user");
            password = pros.getProperty("password");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

        // 建立数据库连接, 创建查询语句, 并且执行语句
        try (Connection conn = DriverManager.getConnection(url, user,
password);

            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql_str)) {

            // 处理查询结果集
            while (rs.next()) {
                System.out.println(rs.getInt("id") + "\t" +
                    rs.getString("name") + "\t" +
                    rs.getTimestamp("created_at"));
            }

        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

首先, 通过一个 `FileInputStream` 对象读取数据库的连接配置文件; 然后使用 `try-with-resources` 方式建立数据库连接, 创建查询语句, 并且执行该语句; 最后使用一个 `while` 循环获取查询结果集, `rs.next()` 获取结果中的下一条记录; `rs.getInt()`、`rs.getString()` 和 `rs.getTimestamp()` 分别用于获取记录中的整数、字符串以及时间戳字段。

该程序的输出结果如下:

```

1    tony    2020-06-15 15:36:13.562
2    david   2020-06-15 15:36:13.563

```

## 27.5 修改数据

我们可以通过 `PreparedStatement` 语句对象 `executeUpdate()` 方法执行更新语句。创建一个新的源文件 `PostgreSQLUpdate.java`:

```

package org.example;

// 导入 JDBC 和 IO 包
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;

```

```

public class PostgreSQLUpdate {
    public static void main(String[] args )
    {
        String url = null;
        String user = null;
        String password = null;
        int affectedrows = 0;
        String sql_str = "UPDATE users " +
            "set name = ? " +
            "WHERE id = ?";

        // 读取数据库连接配置文件
        try (FileInputStream file = new FileInputStream("db.properties")) {

            Properties p = new Properties();
            p.load(file);
            url = p.getProperty("url");
            user = p.getProperty("user");
            password = p.getProperty("password");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

        // 建立数据库连接，创建更新语句，并且执行语句
        try (Connection conn = DriverManager.getConnection(url, user,
password);
            PreparedStatement ps = conn.prepareStatement(sql_str)) {

            // 设置输入参数
            ps.setString(1, "tom");
            ps.setInt(2, 1);

            // 执行更新操作
            affectedrows = ps.executeUpdate();
            System.out.println(String.format("更新行数: %d", affectedrows));

        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

其中，sql\_str 中的两个问号(?)表示两个占位符，它们的值会在运行时进行替换；然后使用 prepareStatement()方法创建了一个 PreparedStatement 预编译的 SQL 语句，在执行该语句之前使用 setInt()和 setString()方法替换占位符的值；最后执行 executeBatch()方法进行更新操作，返回被更新的行数。

运行以上程序的输出结果如下：

```
更新行数: 1
```

我们再次运行上文中的查询示例，确认修改后的结果：

```

2  david    2020-06-15 15:36:13.563
1  tom      2020-06-15 15:36:13.562

```

## 27.6 删除数据

删除操作和插入、更新操作类似，只需要将 SQL 语句替换成 DELETE 即可。我们创建一个新的 Java 文件 PostgreSQLDelete.java:

```
package org.example;

// 导入 JDBC 和 IO 包
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;

public class PostgreSQLDelete {
    public static void main(String[] args )
    {
        String url = null;
        String user = null;
        String password = null;
        int affectedrows = 0;
        String sql_str = "DELETE FROM users WHERE id = ?";

        // 读取数据库连接配置文件
        try (FileInputStream file = new FileInputStream("db.properties")) {

            Properties p = new Properties();
            p.load(file);
            url = p.getProperty("url");
            user = p.getProperty("user");
            password = p.getProperty("password");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

        // 建立数据库连接，创建查询语句，并且执行语句
        try (Connection conn = DriverManager.getConnection(url, user, password);
            PreparedStatement ps = conn.prepareStatement(sql_str)) {

            // 设置输入参数
            ps.setInt(1, 1);

            // 执行更新操作
            affectedrows = ps.executeUpdate();
            System.out.println(String.format("删除行数: %d", affectedrows));

        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

执行该程序输出的结果如下:

## 27.7 处理事务

默认情况下，JDBC 连接 PostgreSQL 时使用自动提交模式，意味着每个 SQL 语句都会自动执行事务的提交操作。如果我们想要在一个事务中执行多条 SQL 语句，需要禁用连接对象的自动提交属性，并且手动执行 COMMIT 或者 ROLLBACK 操作。

我们创建一个新的 Java 文件 PostgreSQLTransaction.java:

```
package org.example;

// 导入 JDBC 和 IO 包
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;
import java.sql.Timestamp;

public class PostgreSQLTransaction {
    public static void main(String[] args )
    {
        String url = null;
        String user = null;
        String password = null;
        String sql_str = "INSERT INTO users(name, created_at) values(?, ?)";
        String sql_str2 = "UPDATE users set name = ? WHERE id = ?";

        // 读取数据库连接配置文件
        try (FileInputStream file = new FileInputStream("db.properties")) {

            Properties p = new Properties();
            p.load(file);
            url = p.getProperty("url");
            user = p.getProperty("user");
            password = p.getProperty("password");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

        // 建立数据库连接，创建查询语句，并且执行语句
        try (Connection conn = DriverManager.getConnection(url, user,
password)) {

            //设置手动提交
            conn.setAutoCommit(false);

            try (PreparedStatement ps = conn.prepareStatement(sql_str);
                PreparedStatement ps2 = conn.prepareStatement(sql_str2)) {

                // 设置输入参数并执行语句
                ps.setString(1, "anne");
```

```

        ps.setTimestamp(2,
Timestamp(System.currentTimeMillis()));
        ps.executeUpdate();

        ps2.setString(1, "anne");
        ps2.setInt(2, 2);
        ps2.executeUpdate();

        // 提交事务
        conn.commit();
        System.out.println("事务提交成功!");
    } catch (SQLException e) {
        // 回滚事务
        conn.rollback();
        System.out.println(e.getMessage());
        System.out.println("回滚事务!");
    }
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}
}

```

创建连接之后，使用 `setAutoCommit()` 方法禁用自动提交；然后分别执行插入语句和更新语句，并提交事务；在异常处理中回滚事务并打印错误消息。

由于更新语句将 `name` 设置为重复值，因此该程序返回以下错误：

```

ERROR: duplicate key value violates unique constraint "users_name_key"
Detail: Key (name)=(tony) already exists.
回滚事务!

```

此时，插入语句也会被回滚；所以并不会创建 `anne`。

## 27.8 调用存储过程

利用 JDBC 中的 `CallableStatement` 对象可以调用 PostgreSQL 存储过程和函数。首先创建一个存储过程 `add_user`：

```

CREATE OR REPLACE PROCEDURE add_user(pv_name varchar, pd_created_at
timestamp)
AS $$
BEGIN
    INSERT INTO users(name, created_at)
    VALUES (pv_name, pd_created_at);
END; $$
LANGUAGE plpgsql;

```

`CallableStatement` 对象默认只支持存储函数调用；为了支持存储过程，需要将连接属性 `escapeSyntaxCallMode` 设置为 `callIfNoReturn`。我们将 `db.properties` 文件修改如下：

```

url=jdbc:postgresql://192.168.56.104:5432/hrdb?escapeSyntaxCallMode=callIfNoReturn
user=tony

```

```
password=tony
```

然后创建一个新的文件 PostgreSQLSP.java:

```
package org.example;

// 导入 JDBC 和 IO 包
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;

public class PostgreSQLSP {
    public static void main( String[] args )
    {
        String url = null;
        String user = null;
        String password = null;


        // 读取数据库连接配置文件
        try (FileInputStream file = new FileInputStream("db.properties")) {

            Properties pros = new Properties();
            pros.load(file);
            url = pros.getProperty("url");
            user = pros.getProperty("user");
            password = pros.getProperty("password");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

        // 建立数据库连接, 调用存储过程
        try (Connection conn = DriverManager.getConnection(url, user,
password);
            CallableStatement stmt = conn.prepareCall("{call
add_user( ?, ? )}");
            ) {
            stmt.setString(1, "anne");
            stmt.setTimestamp(2, new Timestamp(System.currentTimeMillis()));
            stmt.execute();
            System.out.println("调用存储过程成功!");
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

其中, `**{ call add_user( ?, ? ) }` 中的问号是占位符, 表示一个输入参数; `prepareCall()` 方法返回一个 `CallableStatement` 对象, 代表调用存储过程的语句; 设置输入参数后执行存储过程。运行该程序输出以下内容:

```
调用存储过程成功!
```

 除了以上内容之外, JDBC 还提供了许多功能, 例如连接池、事务管理以及负载均衡等, 具体可以参考 [PostgreSQL JDBC 文档](#)。

在实际的应用开发中, 我们不需要直接调用这些底层 JDBC 接口, 而是可以利用成熟的框架,



例如 Mybatis、Hibernate 和 Spring JDBC。这些框架为我们处理了所有的低层细节，包括连接管理、事务控制以及异常处理等；当然，这些框架最后还是调用了 JDBC 接口。

# 附录 A SQL 完整性约束

## A.1 完整性约束

为了维护数据的完整性和一致性，或者为了实现业务需求，SQL 标准定义了完整性约束。以下是常用的 6 种完整性约束：

- **非空约束**（NOT NULL），用于确保字段不会出现空值。例如学生信息表中，学生的姓名、出生日期、性别等一定要有数据。
- **唯一约束**（UNIQUE），用于确保字段中的值不会重复。例如每个学生的身份证、手机号等需要唯一。
- **主键约束**（Primary Key），用于唯一标识表中的每一行数据。例如学生信息表中，学号通常作为主键。主键字段不能为空并且唯一，每个表可以有且只能有一个主键。
- **外键约束**（Foreign Key），用于建立两个表之间的参照完整性。例如学生属于班级，学生信息表中的班级字段是一个外键，引用了班级表的主键。对于外键引用，被引用的数据必须存在；学生不可能属于一个不存在的班级。
- **检查约束**（CHECK）可以定义更多的业务规则。例如，性别的取值只能为“男”或“女”，用户名必须大写等；
- **默认值**（DEFAULT）用于为字段提供默认的数据。例如，玩家注册时的级别默认为一级。

其中，主键代表的是实体完整性；外键定义的是参照完整性；其他属于用户定义的完整性（也称为域完整性）。

SQL 支持在创建表的时候定义约束，或者为已有的表增加新的约束：

```
CREATE TABLE table_name
(
    column_1 data_type column_constraint,
    column_2 data_type,
    ...,
    table_constraint
);

ALTER TABLE table_name ADD CONSTRAINT constraint_desc;
```


其中，column\_constraint 位于字段的定义之后，被称为列级约束；table\_constraint 位于所有字段之后，被称为表级约束。

各种主流数据库对于 SQL 完整性约束的支持如下：

数据库/约束	NOT NULL	UNIQUE	PRIMARY KEY	FOREIGN KEY	CHECK	DEFAULT
Oracle	✓	✓	✓	✓	✓	✓
MySQL	✓	✓	✓	✓*	✓*	✓

SQL Server	✓	✓	✓	✓	✓	✓
PostgreSQL	✓	✓	✓	✓	✓	✓
SQLite	✓	✓	✓	✓	✓	✓

\* MySQL 8.0 开始支持检查约束，InnoDB 和 NDB 存储引擎支持外键约束。

 当我们定义了约束之后，数据库管理系统会在 INSERT、UPDATE、DELETE 等数据修改操作时，或者提交事务时检查数据是否满足完整性约束条件；如果发现用户的操作违反了完整性约束，数据库可能会拒绝执行该操作，或者级联执行其他的修改操作。

虽然以上数据库都提供了 6 种完整性约束的支持，但是在实现和语法上存在一些微小的差异，接下来我们具体讨论一下各种约束。

## A.2 非空约束

定义了 NOT NULL 约束的字段数据不能为空，例如：

```
CREATE TABLE t_nn(
    id INT NOT NULL,
    c1 VARCHAR(10)
);

-- Oracle、MySQL
ALTER TABLE t_nn MODIFY c1 VARCHAR(10) NOT NULL;

-- SQL Server
ALTER TABLE t_nn ALTER COLUMN c1 VARCHAR(10) NOT NULL;

-- PostgreSQL
ALTER TABLE t_nn ALTER COLUMN c1 SET NOT NULL;


-- SQLite 不支持修改字段的约束
```

其中，id 在创建表时指定了非空约束；c1 字段通过 ALTER TABLE 语句增加了非空约束，注意不同数据库的语法实现。接下来我们插入一些数据：

```
INSERT INTO t_nn(id, c1) values (1, 'sql');
INSERT INTO t_nn(id, c1) values (2, null);
SQL Error [1048] [23000]: Column 'c1' cannot be null
```

数据库中的空值（NULL）是一个特殊的值，通常用于表示缺失值或者不适用的值。空值与数字 0 并不相同；空值与空字符串（"）也不相同，但是 Oracle 中的空值与空字符串等价。因此，以下语句在 Oracle 中执行出错，但在其他数据库中执行成功：

```
-- Oracle 空值与空字符串
INSERT INTO t_nn(id, c1) values (2, '');
SQL Error [1400] [23000]: ORA-01400: cannot insert NULL into
("TONY"."T_NN"."C1")
```

 处理空值时需要特别小心，具体可以参考[这篇文章](#)。

## A.3 唯一约束

唯一约束字段中的值不能重复，但是可以存在多个空值。例如：

```
CREATE TABLE t_unique(  
    id INT UNIQUE,  
    c1 INT,  
    c2 INT,  
    CONSTRAINT uk_t_unique UNIQUE (c1, c2)  
);
```

其中，`id` 在创建表时指定了字段级别的唯一约束；`c1` 和 `c2` 字段指定了表级的唯一约束。在我们指定唯一约束时，数据库会自动创建一个唯一索引来实现该功能。接下来我们插入一些重复的数据：

```
INSERT INTO t_unique(id, c1, c2) VALUES (1, 1, 1);  
INSERT INTO t_unique(id, c1, c2) VALUES (NULL, 2, 2);  
INSERT INTO t_unique(id, c1, c2) VALUES (NULL, 3, 3);  
  
-- SQL Server 唯一约束中只允许一个 NULL 值  
SQL Error [2627] [23000]: Violation of UNIQUE KEY constraint  
'UQ__t_unique__3213E83E85135D71'. Cannot insert duplicate key in object  
'dbo.t_unique'. The duplicate key value is (<NULL>).
```

以上语句为 `id` 字段插入了 2 个空值；SQL Server 唯一约束中只允许一个 NULL（也就是 NULL 和 NULL 相同），提示错误；其他数据库可以执行成功。

我们再看一下多字段的复合唯一约束中部分字段数据为空的情况：

```
INSERT INTO t_unique(id, c1, c2) VALUES (2, 1, NULL);  
INSERT INTO t_unique(id, c1, c2) VALUES (3, 1, NULL);  
  
-- Oracle  
SQL Error [1] [23000]: ORA-00001: unique constraint (TONY.UK_T_UNIQUE)  
violated  
  
-- SQL Server  
SQL Error [2627] [23000]: Violation of UNIQUE KEY constraint 'uk_t_unique'.  
Cannot insert duplicate key in object 'dbo.t_unique'. The duplicate key value  
is (1, <NULL>).
```

以上语句为 `c2` 字段插入了 2 个空值；Oracle 和 SQL Server 唯一约束中如果某个字段不为空，其他字段只允许一个 NULL（也就是 NULL 和 NULL 相同）；其他数据库可以执行成功。

还有一种情况，就是复合唯一约束中的所有字段都为空：

```
INSERT INTO t_unique(id, c1, c2) VALUES (4, NULL, NULL);  
INSERT INTO t_unique(id, c1, c2) VALUES (5, NULL, NULL);  
  
-- SQL Server  
SQL Error [2627] [23000]: Violation of UNIQUE KEY constraint 'uk_t_unique'.  
Cannot insert duplicate key in object 'dbo.t_unique'. The duplicate key value  
is (<NULL>, <NULL>).
```

只有 SQL Server 执行出错，也就是说：

- SQL Server 会索引 NULL 值，所以唯一索引只能有一个 NULL 值。
- Oracle 索引中如果部分字段为空，会索引其他不为空的字段；如果所有字段都为空，不会建立索引。
- MySQL、PostgreSQL、SQLite 不会索引 NULL 值，所以唯一索引可以有多个值。

我们也可在创建表之后增加唯一约束或者唯一索引：

```
CREATE TABLE t_unique(
    id INT UNIQUE,
    c1 INT,
    c2 INT
);

-- Oracle、MySQL、SQL Server、PostgreSQL
ALTER TABLE t_unique ADD CONSTRAINT uk_t_unique UNIQUE (c1, c2);

-- 所有数据库，包括 SQLite
CREATE UNIQUE INDEX uk_t_unique ON t_unique (c1, c2);
```

SQLite 不支持创建表之后再增加约束，可以使用唯一索引替代。

## A.4 主键约束

**主键（PRIMARY KEY）**是表中用于唯一地标识每行记录的字段，构成主键的所有字段都不能为空（NOT NULL）并且唯一（UNIQUE）。一个表只能有一个主键。主键可能是一个或多个字段，多个字段的主键被称为复合主键。

如果主键由单个字段构成，可以定义为列级约束或者表级约束。例如：

```
CREATE TABLE t_primary1(id INT NOT NULL PRIMARY KEY,
    c1 INT);

CREATE TABLE t_primary2(id INT NOT NULL,
    c1 INT,
    CONSTRAINT pk2 PRIMARY KEY(id));
```

t\_primary1 的 id 字段定义了主键约束，使用系统生成的主键名；t\_primary2 的 id 字段定义了主键约束，使用自定义的主键名 pk2。如果是多列主键，只能在表级进行定义：

```
CREATE TABLE t_primary3(id INT NOT NULL,
    c1 INT NOT NULL,
    CONSTRAINT pk3 PRIMARY KEY(id, c1));
```

⚠ MySQL 中的主键约束忽略用户指定的名称，使用固定的名称 PRIMARY。

另外，我们也可以使用 ALTER TABLE 语句为已有的表增加一个主键约束：

```
CREATE TABLE t_primary4(id INT NOT NULL,
    c1 INT);
ALTER TABLE t_primary4 ADD CONSTRAINT pk4 PRIMARY KEY (id);
```

⚠ SQLite 不支持这种增加主键约束的方法。

数据库通常会为主键字段创建一个唯一索引，用于确保主键字段值的唯一性。因此，下面的第二个 `INSERT` 语句违反了主键约束：

```
INSERT INTO t_primary1(id, c1) values (1, 100);
INSERT INTO t_primary1(id, c1) values (1, 200);
SQL 错误 [1062] [23000]: Duplicate entry '1' for key 't_primary1.PRIMARY'
```

## A.5 外键约束

外键约束用于建立两个关系表之间的参照引用，通常是一个表中的字段引用另一个表中的主键字段。例如，员工属于部门；因此员工表中的部门字段可以创建外键，引用部门表中的主键。例如：

```
CREATE TABLE dept
( department_id    INTEGER NOT NULL PRIMARY KEY
, department_name  CHARACTER VARYING(30) NOT NULL
) ;

CREATE TABLE emp
( employee_id      INTEGER NOT NULL PRIMARY KEY
, first_name       CHARACTER VARYING(20)
, last_name        CHARACTER VARYING(25) NOT NULL
, salary           NUMERIC(8,2)
, manager_id       INTEGER
, department_id    INTEGER
, CONSTRAINT       fk_emp_dept
                   FOREIGN KEY (department_id)
                   REFERENCES dept(department_id)
) ;
```

外键约束中被引用的表称为父表（`dept`），外键所在的表称为子表（`emp`）。我们再为 `emp` 表增加一个外键：

```
ALTER TABLE emp
ADD CONSTRAINT fk_emp_manager
           FOREIGN KEY (manager_id)
           REFERENCES emp(employee_id)
;

```

 SQLite 不支持这种增加主键约束的方法。

外键约束 `fk_emp_manager` 引用了 `emp` 表自身，用于维护员工和经理之间的联系。如果 `emp` 中已经存在数据，必须满足该外键约束的条件，否则无法添加该约束。

外键约束可以维护数据的参照完整性，员工不会属于一个不存在的部门，例如：

```
INSERT INTO dept VALUES (1, '办公室');

-- SQLite
-- PRAGMA foreign_keys = ON;
INSERT INTO emp VALUES (100, '大', '刘', 50000, NULL, 1);
INSERT INTO emp VALUES (101, '三', '张', 30000, 1, 2);
```

```
SQL Error [2291] [23000]: ORA-02291: integrity constraint (TONY.FK_EMP_DEPT)
violated - parent key not found
```

我们首先创建了一个部门，然后插入两个员工的数据；由于第二个员工的部门(department\_id = 2)不存在，违反了外键约束，插入失败。

⚠ 如果是 SQLite，需要在编译时启用了外键约束支持，并且需要执行 `PRAGMA foreign_keys = ON;` 命令，具体信息可以参考[官方文档](#)。

此时，如果我们删除 dept 表中的记录：

```
DELETE
FROM dept
WHERE department_id = 1;
SQL Error [2292] [23000]: ORA-02292: integrity constraint (TONY.FK_EMP_DEPT)
violated - child record found
```

由于 emp 表中存在部门编号为 1 的员工，删除该部门的信息会破坏数据的完整性，因此执行失败。如果我们将 dept 表中的部门编号从 1 修改为其他编号，同样会违法外键约束。

显然，我们需要有一种能够支持这些数据级联操作的方式。SQL 为此提供了可选的外键级联操作选项：

```
CONSTRAINT constraint_name
FOREIGN KEY (column_name)
REFERENCES parent_name(column_name)
ON DELETE [NO ACTION|RESTRICT|CASCADE|SET NULL|SET DEFAULT]
ON UPDATE [NO ACTION|RESTRICT|CASCADE|SET NULL|SET DEFAULT];
```

其中：

- **NO ACTION** 表示如果父表上的 DELETE 或者 UPDATE 操作违反外键约束，返回错误；在事务提交（COMMIT）时检查。
- **RESTRICT** 表示如果父表上的 DELETE 或者 UPDATE 操作违反外键约束，返回错误；在语句执行时立即检查。
- **CASCADE** 表示如果父表上执行 DELETE 或者 UPDATE 操作，级联删除或者更新子表上的记录。
- **SET NULL**，如果父表上执行 DELETE 或者 UPDATE 操作，将子表中的外键字段设置为 NULL。
- **SET DEFAULT**，如果父表上执行 DELETE 或者 UPDATE 操作，将子表中的外键字段设置为默认值。

如果没有指定级联选项，默认为 NO ACTION。

数据库\级联操作	ON UPDATE	ON DELETE
Oracle	NO ACTION	NO ACTION CASCADE SET NULL
MySQL	NO ACTION RESTRICT CASCADE	NO ACTION RESTRICT CASCADE

	SET NULL	SET NULL
SQL Server	NO ACTION CASCADE SET NULL SET DEFAULT	NO ACTION CASCADE SET NULL SET DEFAULT
PostgreSQL	NO ACTION RESTRICT CASCADE SET NULL SET DEFAULT	NO ACTION RESTRICT CASCADE SET NULL SET DEFAULT
SQLite	NO ACTION RESTRICT CASCADE SET NULL SET DEFAULT	NO ACTION RESTRICT CASCADE SET NULL SET DEFAULT

Oracle 不支持任何外键的级联更新操作;MySQL 中的 NO ACTION 和 RESTRICT 效果相同, 都是在语句执行时立即检查。

我们删除 emp 表上的外键约束 fk\_emp\_dept, 然后创建一个支持级联删除的约束:

```
ALTER TABLE emp DROP CONSTRAINT fk_emp_dept;

ALTER TABLE emp
ADD CONSTRAINT fk_emp_dept
    FOREIGN KEY (department_id)
    REFERENCES dept(department_id)
    ON DELETE CASCADE;
```

⚠ SQLite 不支持删除外键约束, 只能重新创建 emp 表。

接下来我们可以删除 dept 表中的数据, 同时 emp 表中的记录也会被级联删除。

```
DELETE
FROM dept
WHERE department_id = 1;
```

## A.6 检查约束

检查约束指定了一个类似于 WHERE 子句中的条件, 条件中可以使用一个或者多个字段, 每一行数据都必须满足这个条件。不过与 WHERE 条件不同的是, 如果检查的结果是 NULL, 不违反检查约束。例如:

```
CREATE TABLE t_check(
    id INT PRIMARY KEY,
    c1 INT CHECK (c1 IS NOT NULL),
```




```

c2 VARCHAR(10),
c3 INT,
c4 INT,
CONSTRAINT check_c2 CHECK (c2 IN ('START', 'CLOSE'))
);

ALTER TABLE t_check
ADD CONSTRAINT check_c3c4 CHECK ( c3 > c4 );

```

首先，c1 字段上定义了一个列级检查约束，这也是实现非空约束的一种方式；c2 字段上定义了一个表级检查约束，确保取值只能是列表中的值；最后，通过 ALTER TABLE 语句增加了一个检查约束，确保 c3 的值大于 c4，这种引用了多个字段的约束只能是表级约束。

 SQLite 不支持 ALTER TABLE 语句增加约束，可以在创建表时进行定义。

然后我们插入一些数据：

```

INSERT INTO t_check(id, c1, c2, c3, c4) VALUES (1, 1, 'START', 20, 19);

INSERT INTO t_check(id, c1, c2, c3, c4) VALUES (2, NULL, 'START', 20, 19);
SQL Error [3819] [HY000]: Check constraint 't_check_chk_1' is violated.

INSERT INTO t_check(id, c1, c2, c3, c4) VALUES (2, 2, 'PROC', 20, 19);
SQL Error [3819] [HY000]: Check constraint 'check_c2' is violated.

INSERT INTO t_check(id, c1, c2, c3, c4) VALUES (2, 2, 'START', 20, 20);
SQL Error [3819] [HY000]: Check constraint 'check_c3c4' is violated.

```

第一条数据没有违反任何约束；第二条数据 c1 字段的数据为空，违反了非空检查约束；第三条数据违反了 c2 字段上的检查约束；第四条数据 c3 没有大于 c4。

如果插入的数据为空，不会违反检查约束。下面数据中的 c4 为空，可以插入成功：

```

INSERT INTO t_check(id, c1, c2, c3, c4) VALUES (2, 2, 'START', 20, NULL);

SELECT * FROM t_check;
id|c1|c2   |c3|c4|
--|--|-----|--|--|
1| 1|START|20|19|
2| 2|START|20|  |

```

## A.7 默认值

默认值（DEFAULT）用于为字段提供默认的数据。如果用户插入时没有提供数据，使用该默认值。如果没有指定字段的默认值，默认为 NULL。

```

DROP TABLE t_default;

CREATE TABLE t_default(
  id INT PRIMARY KEY,
  c1 INT DEFAULT 0 NOT NULL,
  c2 INT
);

```

```

-- Oracle、MySQL
ALTER TABLE t_default MODIFY C2 INT DEFAULT 100;

-- SQL Server
ALTER TABLE t_default ADD DEFAULT 100 FOR c2;

-- PostgreSQL
ALTER TABLE t_default ALTER COLUMN c2 SET DEFAULT 100;

-- SQLite 不支持修改字段约束

```

其中，c1 字段定义了默认值 0；c2 字段通过 ALTER TABLE 语句定义了默认值 100。接下来测试一下数据插入：

```

INSERT INTO t_default(id) VALUES (1);
SELECT * FROM t_default;
id|c1|c2 |
--|--|---|
1| 0|100|

```

## A.8 其他约束

除了以上常用的完整性约束之外，SQL 还可以通过其他方式实现数据的约束：

- **字段类型**，定义字段的数据类型实际上也是一种约束，属于域约束。例如，INT 类型的字段只能存储整数。不过，SQLite 使用动态类型，不受此限制。
- **断言（Assertion）**，与检查约束类似，但是支持更加宽泛的约束。例如，限制每个部门中最多包含 N 个员工。目前很少有数据库实现了断言。
- **触发器（Trigger）**，预定义存储的 SQL 语句，当用户对表中的数据执行操作时自动触发。触发器可以用于进行复杂的数据检查和控制。