# SQL教程

这是小白的零基础SQL教程。

什么是SQL？简单地说，SQL就是访问和处理关系数据库的计算机标准语言。也就是说，无论用什么编程语言（Java、Python、C++……）编写程序，只要涉及到操作关系数据库，比如，一个电商网站需要把用户和商品信息存入数据库，或者一个手机游戏需要把用户的道具、通关信息存入数据库，都必须通过SQL来完成。

所以，现代程序离不开关系数据库，要使用关系数据库就必须掌握SQL。

在本教程中，你将学到关系数据库的基本概念，如何使用SQL操作数据库，以及一种最流行的开源数据库MySQL的基本安装和使用方法。

## 教程特色：可以在线运行！

你可以在线直接输入并运行SQL，然后观察运行结果。当然，这个在线效果是通过集成了AlaSQL这个JavaScript库实现的，它并不会保存结果，刷新页面，数据库就会恢复到初始状态。

## NoSQL

你可能还听说过NoSQL数据库，也就是非SQL的数据库，包括MongoDB、Cassandra、Dynamo等等，它们都不是关系数据库。有很多人鼓吹现代Web程序已经无需关系数据库了，只需要使用NoSQL就可以。但事实上，SQL数据库从始至终从未被取代过。回顾一下NoSQL的发展历程：

- 1970: NoSQL = We have no SQL
- 1980: NoSQL = Know SQL
- 2000: NoSQL = No SQL!
- 2005: NoSQL = Not only SQL
- 2013: NoSQL = No, SQL!

今天，SQL数据库仍然承担了各种应用程序的核心数据存储，而NoSQL数据库作为SQL数据库的补充，两者不再是二选一的问题，而是主从关系。所以，无论使用哪种编程语言，无论是Web开发、游戏开发还是手机开发，掌握SQL，是所有软件开发人员所必须的。

不要再犹豫了！从现在开始，坚持一周，拿下SQL！

## 关于作者

廖雪峰，十年软件开发经验，业余产品经理，精通Java/Python/Ruby/Visual Basic/Objective C等，对开源框架有深入研究，著有《Spring 2.0核心技术与最佳实践》一书，多个业余开源项目托管在GitHub，欢迎微博交流：

使用窄屏手机的童鞋，请点击左上角“目录”查看教程：


# 关系数据库概述

为什么需要数据库？

因为应用程序需要保存用户的数据，比如Word需要把用户文档保存起来，以便下次继续编辑或者拷贝到另一台电脑。

要保存用户的数据，一个最简单的方法是把用户数据写入文件。例如，要保存一个班级所有学生的信息，可以向文件中写入一个CSV文件：

```
id,name,gender,score
1,小明,M,90
2,小红,F,95
3,小军,M,88
4,小丽,F,88
```

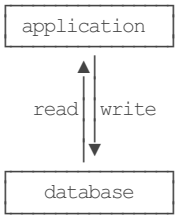
如果要保存学校所有班级的信息，可以写入另一个CSV文件。

但是，随着应用程序的功能越来越复杂，数据量越来越大，如何管理这些数据就成了大问题：

- 读写文件并解析出数据需要大量重复代码；
- 从成千上万的数据中快速查询出指定数据需要复杂的逻辑。

如果每个应用程序都各自写自己的读写数据的代码，一方面效率低，容易出错，另一方面，每个应用程序访问数据的接口都不相同，数据难以复用。

所以，数据库作为一种专门管理数据的软件就出现了。应用程序不需要自己管理数据，而是通过数据库软件提供的接口来读写数据。至于数据本身如何存储到文件，那是数据库软件的事情，应用程序自己并不关心：



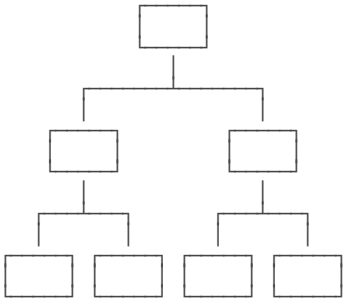

这样一来，编写应用程序的时候，数据读写的功能就被大大地简化了。

## 数据模型

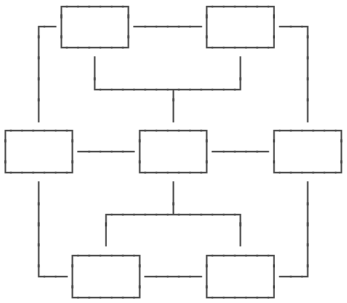
数据库按照数据结构来组织、存储和管理数据，实际上，数据库一共有三种模型：

- 层次模型
- 网状模型
- 关系模型

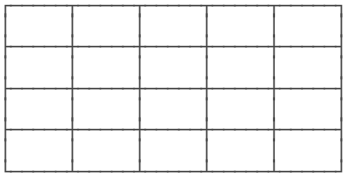
层次模型就是以“上下级”的层次关系来组织数据的一种方式，层次模型的数据结构看起来就像一颗树：



网状模型把每个数据节点和其他很多节点都连接起来，它的数据结构看起来就像很多城市之间的路网：



关系模型把数据看作是一个二维表格，任何数据都可以通过行号+列号来唯一确定，它的数据模型看起来就是一个Excel表：



随着时间的推移和市场竞争，最终，基于关系模型的关系数据库获得了绝对市场份额。

为什么关系数据库获得了最广泛的应用？

因为相比层次模型和网状模型，关系模型理解和使用起来最简单。

关系数据库的关系模型是基于数学理论建立的。我们把域（Domain）定义为一组具有相同数据类型的值的集合，给定一组域D1,D2,...,Dn，它们的笛卡尔集定义为D1×D2×……×Dn={(d1,d2,...,dn)|di∈Di,i=1,2,...,n}， 而D1×D2×……×Dn的子集叫作在域D1,D2,...,Dn上的关系，表示为R(D1,D2,...,Dn)，这里的R表示$#%&^@!&$#;!~%¥%:(……算了，根本讲不明白，大家也不用理解。

基于数学理论的关系模型虽然讲起来挺复杂，但是，基于日常生活的关系模型却十分容易理解。我们以学校班级为例，一个班级的学生就可以用一个表格存起来，并且定义如下：

| ID | 姓名 | 班级ID | 性别 | 年龄 |
|---|---|---|---|---|
| 1 | 小明 | 201 | M | 9 |
| 2 | 小红 | 202 | F | 8 |
| 3 | 小军 | 202 | M | 8 |
| 4 | 小白 | 201 | F | 9 |

其中，班级ID对应着另一个班级表：

| ID | 名称 | 班主任 |
|---|---|---|
| 201 | 二年级一班 | 王老师 |
| 202 | 二年级二班 | 李老师 |

通过给定一个班级名称，可以查到一条班级记录，根据班级ID，又可以查到多条学生记录，这样，二维表之间就通过ID映射建立了“一对多”关系。

## 数据类型

对于一个关系表，除了定义每一列的名称外，还需要定义每一列的数据类型。关系数据库支持的标准数据类型包括数值、字符串、时间等：

| 名称 | 类型 | 说明 |
|---|---|---|

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| INT | 整型 | 4字节整数类型，范围约+/-21亿 |
| BIGINT | 长整型 | 8字节整数类型，范围约+/-922亿亿 |
| REAL | 浮点型 | 4字节浮点数，范围约+/-$10^{38}$ |
| DOUBLE | 浮点型 | 8字节浮点数，范围约+/-$10^{308}$ |
| DECIMAL(M,N) | 高精度小数 | 由用户指定精度的小数，例如，DECIMAL(20,10)表示一共20位，其中小数10位，通常用于财务计算 |
| CHAR(N) | 定长字符串 | 存储指定长度的字符串，例如，CHAR(100)总是存储100个字符的字符串 |
| VARCHAR(N) | 变长字符串 | 存储可变长度的字符串，例如，VARCHAR(100)可以存储0~100个字符的字符串 |
| BOOLEAN | 布尔类型 | 存储True或者False |
| DATE | 日期类型 | 存储日期，例如，2018-06-22 |
| TIME | 时间类型 | 存储时间，例如，12:20:59 |
| DATETIME | 日期和时间类型 | 存储日期+时间，例如，2018-06-22 12:20:59 |

上面的表中列举了最常用的数据类型。很多数据类型还有别名，例如，`REAL`又可以写成`FLOAT(24)`。还有一些不常用的数据类型，例如，`TINYINT`（范围在0~255）。各数据库厂商还会支持特定的数据类型，例如`JSON`。

选择数据类型的时候，要根据业务规则选择合适的类型。通常来说，`BIGINT`能满足整数存储的需求，`VARCHAR(N)`能满足字符串存储的需求，这两种类型是使用最广泛的。

## 主流关系数据库

目前，主流的关系数据库主要分为以下几类：

1. 商用数据库，例如：Oracle，SQL Server，DB2等；
2. 开源数据库，例如：MySQL，PostgreSQL等；
3. 桌面数据库，以微软Access为代表，适合桌面应用程序使用；
4. 嵌入式数据库，以Sqlite为代表，适合手机应用和桌面程序。

# SQL

什么是SQL？SQL是结构化查询语言的缩写，用来访问和操作数据库系统。SQL语句既可以查询数据库中的数据，也可以添加、更新和删除数据库中的数据，还可以对数据库进行管理和维护操作。不同的数据库，都支持SQL，这样，我们通过学习SQL这一种语言，就可以操作各种不同的数据库。

虽然SQL已经被ANSI组织定义为标准，不幸地是，各个不同的数据库对标准的SQL支持不太一致。并且，大部分数据库都在标准的SQL上做了扩展。也就是说，如果只使用标准SQL，理论上所有数据库都可以支持，但如果使用某个特定数据库的扩展SQL，换一个数据库就不能执行了。例如，Oracle把自己扩展的SQL称为`PL/SQL`，Microsoft把自己扩展的SQL称为`T-SQL`。

现实情况是，如果我们只使用标准SQL的核心功能，那么所有数据库通常都可以执行。不常用的SQL功能，不同的数据库支持的程度都不一样。而各个数据库支持的各自扩展的功能，通常我们把它们称之为“方言”。

总的来说，SQL语言定义了这么几种操作数据库的能力：

**DDL：Data Definition Language**

DDL允许用户定义数据，也就是创建表、删除表、修改表结构这些操作。通常，DDL由数据库管理员执行。

**DML：Data Manipulation Language**

DML为用户提供添加、删除、更新数据的能力，这些是应用程序对数据库的日常操作。

**DQL：Data Query Language**

DQL允许用户查询数据，这也是通常最频繁的数据库日常操作。

## 语法特点

SQL语言关键字不区分大小写！！！但是，针对不同的数据库，对于表名和列名，有的数据库区分大小写，有的数据库不区分大小写。同一个数据库，有的在Linux上区分大小写，有的在Windows上不区分大小写。

所以，本教程约定：SQL关键字总是大写，以示突出，表名和列名均使用小写。

### SQL的全称是：

- [ ] Strange Question Language
- [ ] Structured Question Language
- [ ] Structured Query Language

Submit

# 安装MySQL

MySQL是目前应用最广泛的开源关系数据库。MySQL最早是由瑞典的MySQL AB公司开发，该公司在2008年被SUN公司收购，紧接着，SUN公司在2009年被Oracle公司收购，所以MySQL最终就变成了Oracle旗下的产品。

和其他关系数据库有所不同的是，MySQL本身实际上只是一个SQL接口，它的内部还包含了多种数据引擎，常用的包括：

- InnoDB：由Innobase Oy公司开发的一款支持事务的数据库引擎，2006年被Oracle收购；
- MyISAM：MySQL早期集成的默认数据库引擎，不支持事务。

MySQL接口和数据库引擎的关系就好比某某浏览器和浏览器引擎（IE引擎或Webkit引擎）的关系。对用户而言，切换浏览器引擎不影响浏览器界面，切换MySQL引擎不影响自己写的应用程序使用MySQL的接口。

使用MySQL时，不同的表还可以使用不同的数据库引擎。如果你不知道应该采用哪种引擎，记住总是选择InnoDB就好了。

因为MySQL一开始就是开源的，所以基于MySQL的开源版本，又衍生出了各种版本：

### MariaDB

由MySQL的创始人创建的一个开源分支版本，使用XtraDB引擎。

### Aurora

由Amazon改进的一个MySQL版本，专门提供给在AWS托管MySQL用户，号称5倍的性能提升。

### PolarDB

由Alibaba改进的一个MySQL版本，专门提供给在阿里云托管的MySQL用户，号称6倍的性能提升。

而MySQL官方版本又分了好几个版本：

- Community Edition：社区开源版本，免费；
- Standard Edition：标准版；
- Enterprise Edition：企业版；
- Cluster Carrier Grade Edition：集群版。

以上版本的功能依次递增，价格也依次递增。不过，功能增加的主要是监控、集群等管理功能，对于基本的SQL功能是完全一样的。

所以使用MySQL就带来了一个巨大的好处：可以在自己的电脑上安装免费的Community Edition版本，进行学习、开发、测试，部署的时候，可以选择付费的高级版本，或者云服务商提供的兼容版本，而不需要对应用程序本身做改动。

## 安装MySQL

要在Windows或Mac上安装MySQL，首先从MySQL官方网站下载最新的MySQL Community Server版本：

https://dev.mysql.com/downloads/mysql/

选择对应的操作系统版本，下载安装即可。在安装过程中，MySQL会自动创建一个`root`用户，并提示输入`root`口令。

要在Linux上安装MySQL，可以使用发行版的包管理器。例如，Debian和Ubuntu用户可以简单地通过命令`apt-get install mysql-server`安装最新的MySQL版本。

## 运行MySQL

MySQL安装后会自动在后台运行。为了验证MySQL安装是否正确，我们需要通过`mysql`这个命令行程序来连接MySQL服务器。

在命令提示符下输入`mysql -u root -p`，然后输入口令，如果一切正确，就会连接到MySQL服务器，同时提示符变为`mysql>`。

输入`exit`退出MySQL命令行。注意，MySQL服务器仍在后台运行。

# 关系模型

我们已经知道，关系数据库是建立在关系模型上的。而关系模型本质上就是若干个存储数据的二维表，可以把它们看作很多Excel表。

表的每一行称为记录（Record），记录是一个逻辑意义上的数据。

表的每一列称为字段（Column），同一个表的每一行记录都拥有相同的若干字段。

字段定义了数据类型（整型、浮点型、字符串、日期等），以及是否允许为`NULL`。注意`NULL`表示字段数据不存在。一个整型字段如果为`NULL`不表示它的值为`0`，同样的，一个字符串型字段为`NULL`也不表示它的值为空串`''`。

> 通常情况下，字段应该避免允许为NULL。不允许为NULL可以简化查询条件，加快查询速度，也利于应用程序读取数据后无需判断是否为NULL。

和Excel表有所不同的是，关系数据库的表和表之间需要建立“一对多”，“多对一”和“一对一”的关系，这样才能够按照应用程序的逻辑来组织和存储数据。

例如，一个班级表：

| ID | 名称 | 班主任 |
|---|---|---|
| 201 | 二年级一班 | 王老师 |
| 202 | 二年级二班 | 李老师 |

每一行对应着一个班级，而一个班级对应着多个学生，所以班级表和学生表的关系就是“一对多”：

| ID | 姓名 | 班级ID | 性别 | 年龄 |
|---|---|---|---|---|
| 1 | 小明 | 201 | M | 9 |
| 2 | 小红 | 202 | F | 8 |
| 3 | 小军 | 202 | M | 8 |
| 4 | 小白 | 201 | F | 9 |

反过来，如果我们先在学生表中定位了一行记录，例如`ID=1`的小明，要确定他的班级，只需要根据他的“班级ID”对应的值`201`找到班级表中`ID=201`的记录，即二年级一班。所以，学生表和班级表是“多对一”的关系。

如果我们把班级表分拆得细一点，例如，单独创建一个教师表：

| ID | 名称 | 年龄 |
| --- | --- | --- |
| A1 | 王老师 | 26 |
| A2 | 张老师 | 39 |
| A3 | 李老师 | 32 |
| A4 | 赵老师 | 27 |

班级表只存储教师ID：

| ID | 名称 | 班主任ID |
| --- | --- | --- |
| 201 | 二年级一班 | A1 |
| 202 | 二年级二班 | A3 |

这样，一个班级总是对应一个教师，班级表和教师表就是“一对一”关系。

在关系数据库中，关系是通过主键和外键来维护的。我们在后面会分别深入讲解。

# 主键

在关系数据库中，一张表中的每一行数据被称为一条记录。一条记录就是由多个字段组成的。例如，`students`表的两行记录：

| id | class_id | name | gender | score |
| --- | --- | --- | --- | --- |
| 1 | 1 | 小明 | M | 90 |
| 2 | 1 | 小红 | F | 95 |

每一条记录都包含若干定义好的字段。同一个表的所有记录都有相同的字段定义。

对于关系表，有个很重要的约束，就是任意两条记录不能重复。不能重复不是指两条记录不完全相同，而是指能够通过某个字段唯一区分出不同的记录，这个字段被称为主键。

例如，假设我们把`name`字段作为主键，那么通过名字`小明`或`小红`就能唯一确定一条记录。但是，这么设定，就没法存储同名的同学了，因为插入相同主键的两条记录是不被允许的。

对主键的要求，最关键的一点是：记录一旦插入到表中，主键最好不要再修改，因为主键是用来唯一定位记录的，修改了主键，会造成一系列的影响。

由于主键的作用十分重要，如何选取主键会对业务开发产生重要影响。如果我们以学生的身份证号作为主键，似乎能唯一定位记录。然而，身份证号也是一种业务场景，如果身份证号升位了，或者需要变更，作为主键，不得不修改的时候，就会对业务产生严重影响。

所以，选取主键的一个基本原则是：不使用任何业务相关的字段作为主键。

因此，身份证号、手机号、邮箱地址这些看上去可以唯一的字段，均不可用作主键。

作为主键最好是完全业务无关的字段，我们一般把这个字段命名为`id`。常见的可作为`id`字段的类型有：

1. 自增整数类型：数据库会在插入数据时自动为每一条记录分配一个自增整数，这样我们就完全不用担心主键重复，也不用自己预先生成主键；
2. 全局唯一GUID类型：使用一种全局唯一的字符串作为主键，类似`8f55d96b-8acc-4636-8cb8-76bf8abc2f57`。GUID算法通过网卡MAC地址、时间戳和随机数保证任意计算机在任意时间生成的字符串都是不同的，大部分编程语言都内置了GUID算法，可以自己预算出主键。

对于大部分应用来说，通常自增类型的主键就能满足需求。我们在`students`表中定义的主键也是`BIGINT NOT NULL AUTO_INCREMENT`类型。

> 如果使用INT自增类型，那么当一张表的记录数超过2147483647（约21亿）时，会达到上限而出错。使用BIGINT自增类型则可以

> 最多约922亿亿条记录。

### 联合主键

关系数据库实际上还允许通过多个字段唯一标识记录，即两个或更多的字段都设置为主键，这种主键被称为联合主键。

对于联合主键，允许一列有重复，只要不是所有主键列都重复即可：

| id_num | id_type | other columns... |
|---|---|---|
| 1 | A | ... |
| 2 | A | ... |
| 2 | B | ... |

如果我们把上述表的`id_num`和`id_type`这两列作为联合主键，那么上面的3条记录都是允许的，因为没有两列主键组合起来是相同的。

没有必要的情况下，我们尽量不使用联合主键，因为它给关系表带来了复杂度的上升。

### 小结

主键是关系表中记录的唯一标识。主键的选取非常重要：主键不要带有业务含义，而应该使用BIGINT自增或者GUID类型。主键也不应该允许`NULL`。

可以使用多个列作为联合主键，但联合主键并不常用。

## 外键

当我们用主键唯一标识记录时，我们就可以在`students`表中确定任意一个学生的记录：

| id | name | other columns... |
|---|---|---|
| 1 | 小明 | ... |
| 2 | 小红 | ... |

我们还可以在`classes`表中确定任意一个班级记录：

| id | name | other columns... |
|---|---|---|
| 1 | 一班 | ... |
| 2 | 二班 | ... |

但是我们如何确定`students`表的一条记录，例如，`id=1`的小明，属于哪个班级呢？

由于一个班级可以有多个学生，在关系模型中，这两个表的关系可以称为“一对多”，即一个`classes`的记录可以对应多个`students`表的记录。

为了表达这种一对多的关系，我们需要在`students`表中加入一列`class_id`，让它的值与`classes`表的某条记录相对应：

| id | class_id | name | other columns... |
|---|---|---|---|
| 1 | 1 | 小明 | ... |
| 2 | 1 | 小红 | ... |
| 5 | 2 | 小白 | ... |

这样，我们就可以根据`class_id`这个列直接定位出一个`students`表的记录应该对应到`classes`的哪条记录。

例如：

- 小明的`class_id`是`1`，因此，对应的`classes`表的记录是`id=1`的一班；
- 小红的`class_id`是`1`，因此，对应的`classes`表的记录是`id=1`的一班；
- 小白的`class_id`是`2`，因此，对应的`classes`表的记录是`id=2`的二班。

在`students`表中，通过`class_id`的字段，可以把数据与另一张表关联起来，这种列称为`外键`。

外键并不是通过列名实现的，而是通过定义外键约束实现的：

```
ALTER TABLE students
ADD CONSTRAINT fk_class_id
FOREIGN KEY (class_id)
REFERENCES classes (id);
```

其中，外键约束的名称`fk_class_id`可以任意，`FOREIGN KEY (class_id)`指定了`class_id`作为外键，`REFERENCES classes (id)`指定了这个外键将关联到`classes`表的`id`列（即`classes`表的主键）。

通过定义外键约束，关系数据库可以保证无法插入无效的数据。即如果`classes`表不存在`id=99`的记录，`students`表就无法插入`class_id=99`的记录。

由于外键约束会降低数据库的性能，大部分互联网应用程序为了追求速度，并不设置外键约束，而是仅靠应用程序自身来保证逻辑的正确性。这种情况下，`class_id`仅仅是一个普通的列，只是它起到了外键的作用而已。

要删除一个外键约束，也是通过`ALTER TABLE`实现的：

```
ALTER TABLE students
DROP FOREIGN KEY fk_class_id;
```

注意：删除外键约束并没有删除外键这一列。删除列是通过`DROP COLUMN ...`实现的。

## 多对多

通过一个表的外键关联到另一个表，我们可以定义出一对多关系。有些时候，还需要定义“多对多”关系。例如，一个老师可以对应多个班级，一个班级也可以对应多个老师，因此，班级表和老师表存在多对多关系。

多对多关系实际上是通过两个一对多关系实现的，即通过一个中间表，关联两个一对多关系，就形成了多对多关系：

`teachers`表：

| id | name |
|---|---|
| 1 | 张老师 |
| 2 | 王老师 |
| 3 | 李老师 |
| 4 | 赵老师 |

`classes`表：

| id | name |
|---|---|
| 1 | 一班 |
| 2 | 二班 |

中间表`teacher_class`关联两个一对多关系：

| id | teacher_id | class_id |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |

| id | teacher_id | class_id |
|---|---|---|
| 3 | 2 | 1 |
| 4 | 2 | 2 |
| 5 | 3 | 1 |
| 6 | 4 | 2 |

通过中间表`teacher_class`可知`teachers`到`classes`的关系：

- `id=1`的张老师对应`id=1,2`的一班和二班；
- `id=2`的王老师对应`id=1,2`的一班和二班；
- `id=3`的李老师对应`id=1`的一班；
- `id=4`的赵老师对应`id=2`的二班。

同理可知`classes`到`teachers`的关系：

- `id=1`的一班对应`id=1,2,3`的张老师、王老师和李老师；
- `id=2`的二班对应`id=1,2,4`的张老师、王老师和赵老师；

因此，通过中间表，我们就定义了一个“多对多”关系。

### 一对一

一对一关系是指，一个表的记录对应到另一个表的唯一一个记录。

例如，`students`表的每个学生可以有自己的联系方式，如果把联系方式存入另一个表`contacts`，我们就可以得到一个“一对一”关系：

| id | student_id | mobile |
|---|---|---|
| 1 | 1 | 135xxxx6300 |
| 2 | 2 | 138xxxx2209 |
| 3 | 5 | 139xxxx8086 |

有细心的童鞋会问，既然是一对一关系，那为啥不给`students`表增加一个`mobile`列，这样就能合二为一了？

如果业务允许，完全可以把两个表合为一个表。但是，有些时候，如果某个学生没有手机号，那么，`contacts`表就不存在对应的记录。实际上，一对一关系准确地说，是`contacts`表一对一对应`students`表。

还有一些应用会把一个大表拆成两个一对一的表，目的是把经常读取和不经常读取的字段分开，以获得更高的性能。例如，把一个大的用户表分拆为用户基本信息表`user_info`和用户详细信息表`user_profiles`，大部分时候，只需要查询`user_info`表，并不需要查询`user_profiles`表，这样就提高了查询速度。

### 小结

关系数据库通过外键可以实现一对多、多对多和一对一的关系。外键既可以通过数据库来约束，也可以不设置约束，仅依靠应用程序的逻辑来保证。

## 索引

在关系数据库中，如果有上万甚至上亿条记录，在查找记录的时候，想要获得非常快的速度，就需要使用索引。

索引是关系数据库中对某一列或多个列的值进行预排序的数据结构。通过使用索引，可以让数据库系统不必扫描整个表，而是直接定位到符合条件的记录，这样就大大加快了查询速度。

例如，对于`students`表：

| id | class_id | name | gender | score |
|---|---|---|---|---|

| id | class_id | name | gender | score |
| --- | --- | --- | --- | --- |
| 1 | 1 | 小明 | M | 90 |
| 2 | 1 | 小红 | F | 95 |
| 3 | 1 | 小军 | M | 88 |

如果要经常根据`score`列进行查询，就可以对`score`列创建索引：

```
ALTER TABLE students
ADD INDEX idx_score (score);
```

使用`ADD INDEX idx_score (score)`就创建了一个名称为`idx_score`，使用列`score`的索引。索引名称是任意的，索引如果有多列，可以在括号里依次写上，例如：

```
ALTER TABLE students
ADD INDEX idx_name_score (name, score);
```

索引的效率取决于索引列的值是否散列，即该列的值如果越互不相同，那么索引效率越高。反过来，如果记录的列存在大量相同的值，例如`gender`列，大约一半的记录值是`M`，另一半是`F`，因此，对该列创建索引就没有意义。

可以对一张表创建多个索引。索引的优点是提高了查询效率，缺点是在插入、更新和删除记录时，需要同时修改索引，因此，索引越多，插入、更新和删除记录的速度就越慢。

对于主键，关系数据库会自动对其创建主键索引。使用主键索引的效率是最高的，因为主键会保证绝对唯一。

## 唯一索引

在设计关系数据表的时候，看上去唯一的列，例如身份证号、邮箱地址等，因为他们具有业务含义，因此不宜作为主键。

但是，这些列根据业务要求，又具有唯一性约束：即不能出现两条记录存储了同一个身份证号。这个时候，就可以给该列添加一个唯一索引。例如，我们假设`students`表的`name`不能重复：

```
ALTER TABLE students
ADD UNIQUE INDEX uni_name (name);
```

通过`UNIQUE`关键字我们就添加了一个唯一索引。

也可以只对某一列添加一个唯一约束而不创建唯一索引：

```
ALTER TABLE students
ADD CONSTRAINT uni_name UNIQUE (name);
```

这种情况下，`name`列没有索引，但仍然具有唯一性保证。

无论是否创建索引，对于用户和应用程序来说，使用关系数据库不会有任何区别。这里的意思是说，当我们在数据库中查询时，如果有相应的索引可用，数据库系统就会自动使用索引来提高查询效率，如果没有索引，查询也能正常执行，只是速度会变慢。因此，索引可以在使用数据库的过程中逐步优化。

## 小结

通过对数据库表创建索引，可以提高查询速度。

通过创建唯一索引，可以保证某一列的值具有唯一性。

数据库索引对于用户和应用程序来说都是透明的。

# 在线SQL

为了便于在线练习，我们提供了一个在线运行SQL的功能。实际上这是在浏览器页面运行的一个JavaScript编写的内存型SQL数据库AlaSQL。不必运行MySQL等实际的数据库软件，即可在线编写并执行SQL语句。

可以在此测试执行最简单的SQL语句：

```
-- 以双减号开头的是注释
SELECT * FROM students;
```

Run

请注意，在页面加载时，`students`表和`classes`表就自动被创建并填入了若干数据。由于数据只存在于浏览器的内存中，因此，如果修改了数据，重新刷新页面后，数据会重置为初始值。

# 查询数据

在关系数据库中，最常用的操作就是查询。

## 准备数据

为了便于讲解和练习，我们先准备好了一个`students`表和一个`classes`表，它们的结构和数据如下：

`students`表存储了学生信息：

| id | class_id | name | gender | score |
|---|---|---|---|---|
| 1 | 1 | 小明 | M | 90 |
| 2 | 1 | 小红 | F | 95 |
| 3 | 1 | 小军 | M | 88 |
| 4 | 1 | 小米 | F | 73 |
| 5 | 2 | 小白 | F | 81 |
| 6 | 2 | 小兵 | M | 55 |
| 7 | 2 | 小林 | M | 85 |
| 8 | 3 | 小新 | F | 91 |
| 9 | 3 | 小王 | M | 89 |
| 10 | 3 | 小丽 | F | 85 |

`classes`表存储了班级信息：

| id | name |
|---|---|
| 1 | 一班 |

| id | name |
|---|---|
| 2 | 二班 |
| 3 | 三班 |
| 4 | 四班 |

请注意，和`MySQL`的持久化存储不同的是，由于我们使用的是AlaSQL内存数据库，两张表的数据在页面加载时导入，并且只存在于浏览器的内存中，因此，刷新页面后，数据会重置为上述初始值。

## MySQL

如果你想用MySQL练习，可以下载这个SQL脚本，然后在命令行运行：

```
$ mysql -u root -p < init-test-data.sql
```

就可以自动创建`test`数据库，并且在`test`数据库下创建`students`表和`classes`表，以及必要的初始化数据。

和内存数据库不同的是，对MySQL数据库做的所有修改，都会保存下来。如果你希望恢复到初始状态，可以再次运行该脚本。

# 基本查询

要查询数据库表的数据，我们使用如下的SQL语句：

```
SELECT * FROM <表名>
```

假设表名是`students`，要查询`students`表的所有行，我们用如下SQL语句：

```
-- 查询students表的所有数据
SELECT * FROM students;
```

Run

使用`SELECT * FROM students`时，`SELECT`是关键字，表示将要执行一个查询，`*`表示“所有列”，`FROM`表示将要从哪个表查询，本例中是`students`表。

该SQL将查询出`students`表的所有数据。注意：查询结果也是一个二维表，它包含列名和每一行的数据。

要查询`classes`表的所有行，我们用如下SQL语句：

```
-- 查询classes表的所有数据
```

```
SELECT * FROM classes;
```

Run

运行上述SQL语句，观察查询结果。

`SELECT`语句其实并不要求一定要有`FROM`子句。我们来试试下面的`SELECT`语句：

```
-- 计算100+200
SELECT 100+200;
```

Run

上述查询会直接计算出表达式的结果。虽然`SELECT`可以用作计算，但它并不是SQL的强项。但是，不带`FROM`子句的`SELECT`语句有一个有用的用途，就是用来判断当前到数据库的连接是否有效。许多检测工具会执行一条`SELECT 1;`来测试数据库连接。

### 小结

使用SELECT查询的基本语句`SELECT * FROM <表名>`可以查询一个表的所有行和所有列的数据。

SELECT查询的结果是一个二维表。

# 条件查询

使用`SELECT * FROM <表名>`可以查询到一张表的所有记录。但是，很多时候，我们并不希望获得所有记录，而是根据条件选择性地获取指定条件的记录，例如，查询分数在80分以上的学生记录。在一张表有数百万记录的情况下，获取所有记录不仅费时，还费内存和网络带宽。

SELECT语句可以通过`WHERE`条件来设定查询条件，查询结果是满足查询条件的记录。例如，要指定条件“分数在80分或以上的学生”，写成`WHERE`条件就是`SELECT * FROM students WHERE score >= 80`。

其中，`WHERE`关键字后面的`score >= 80`就是条件。`score`是列名，该列存储了学生的成绩，因此，`score >= 80`就筛选出了指定条件的记录：

```
-- 按条件查询students:
SELECT * FROM students WHERE score >= 80;
```

Run

因此，条件查询的语法就是：

```
SELECT * FROM <表名> WHERE <条件表达式>
```

条件表达式可以用`<条件1> AND <条件2>`表达满足条件1并且满足条件2。例如，符合条件“分数在80分或以上”，并且还符合条件“男生”，把这两个条件写出来：

- 条件1：根据score列的数据判断：`score >= 80`；
- 条件2：根据gender列的数据判断：`gender = 'M'`，注意`gender`列存储的是字符串，需要用单引号括起来。

就可以写出`WHERE`条件：`score >= 80 AND gender = 'M'`：

```
-- 按AND条件查询students:
SELECT * FROM students WHERE score >= 80 AND gender = 'M';
```

Run

第二种条件是`<条件1> OR <条件2>`，表示满足条件1或者满足条件2。例如，把上述`AND`查询的两个条件改为`OR`，查询结果就是“分数在80分或以上”或者“男生”，满足任意之一的条件即选出该记录：

```
-- 按OR条件查询students:
```

```
SELECT * FROM students WHERE score >= 80 OR gender = 'M';
```

Run

很显然`OR`条件要比`AND`条件宽松，返回的符合条件的记录也更多。

第三种条件是`NOT <条件>`，表示“不符合该条件”的记录。例如，写一个“不是2班的学生”这个条件，可以先写出“是2班的学生”：`class_id = 2`，再加上`NOT`：`NOT class_id = 2`：

```
-- 按NOT条件查询students:
SELECT * FROM students WHERE NOT class_id = 2;
```

Run

上述`NOT`条件`NOT class_id = 2`其实等价于`class_id <> 2`，因此，`NOT`查询不是很常用。

要组合三个或者更多的条件，就需要用小括号`()`表示如何进行条件运算。例如，编写一个复杂的条件：分数在80以下或者90以上，并且是男生：

```
-- 按多个条件查询students:
SELECT * FROM students WHERE (score < 80 OR score > 90) AND gender = 'M';
```

Run

如果不加括号，条件运算按照`NOT`、`AND`、`OR`的优先级进行，即`NOT`优先级最高，其次是`AND`，最后是`OR`。加上括号可以改变优先

级。

### 常用的条件表达式

| 条件 | 表达式举例1 | 表达式举例2 | 说明 |
| --- | --- | --- | --- |
| 使用=判断相等 | score = 80 | name = 'abc' | 字符串需要用单引号括起来 |
| 使用>判断大于 | score > 80 | name > 'abc' | 字符串比较根据ASCII码，中文字符比较根据数据库设置 |
| 使用>=判断大于或相等 | score >= 80 | name >= 'abc' | |
| 使用<判断小于 | score < 80 | name <= 'abc' | |
| 使用<=判断小于或相等 | score <= 80 | name <= 'abc' | |
| 使用<>判断不相等 | score <> 80 | name <> 'abc' | |
| 使用LIKE判断相似 | name LIKE 'ab%' | name LIKE '%bc%' | %表示任意字符，例如'ab%'将匹配'ab'，'abc'，'abcd' |

查询分数在60分(含)~90分(含)之间的学生可以使用的WHERE语句是：

- [ ] WHERE score >= 60 OR score <= 90
- [ ] WHERE score >= 60 AND score <= 90
- [ ] WHERE score IN (60, 90)
- [ ] WHERE score BETWEEN 60 AND 90
- [ ] WHERE 60 <= score <= 90

Submit

### 小结

通过`WHERE`条件查询，可以筛选出符合指定条件的记录，而不是整个表的所有记录。

# 投影查询

使用`SELECT * FROM <表名> WHERE <条件>`可以选出表中的若干条记录。我们注意到返回的二维表结构和原表是相同的，即结果集的所有列与原表的所有列都一一对应。

如果我们只希望返回某些列的数据，而不是所有列的数据，我们可以用`SELECT 列1, 列2, 列3 FROM ...`，让结果集仅包含指定列。这种操作称为投影查询。

例如，从`students`表中返回`id`、`score`和`name`这三列：

```
-- 使用投影查询
SELECT id, score, name FROM students;
```

Run

这样返回的结果集就只包含了我们指定的列，并且，结果集的列的顺序和原表可以不一样。

使用`SELECT 列1, 列2, 列3 FROM ...`时，还可以给每一列起个别名，这样，结果集的列名就可以与原表的列名不同。它的语法是`SELECT 列1 别名1, 列2 别名2, 列3 别名3 FROM ...`。

例如，以下`SELECT`语句将列名`score`重命名为`points`，而`id`和`name`列名保持不变：

```
-- 使用投影查询，并将列名重命名：
SELECT id, score points, name FROM students;
```

Run

投影查询同样可以接`WHERE`条件，实现复杂的查询：

```
-- 使用投影查询+WHERE条件：
SELECT id, score points, name FROM students WHERE gender = 'M';
```

Run

### 小结

使用`SELECT *`表示查询表的所有列，使用`SELECT 列1, 列2, 列3`则可以仅返回指定列，这种操作称为投影。

`SELECT`语句可以对结果集的列进行重命名。

# 排序

### 排序

我们使用SELECT查询时，细心的读者可能注意到，查询结果集通常是按照`id`排序的，也就是根据主键排序。这也是大部分数据库的做法。如果我们要根据其他条件排序怎么办？可以加上`ORDER BY`子句。例如按照成绩从低到高进行排序：

```
-- 按score从低到高
SELECT id, name, gender, score FROM students ORDER BY score;
```

Run

如果要反过来，按照成绩从高到底排序，我们可以加上`DESC`表示“倒序”：

```
-- 按score从高到低
SELECT id, name, gender, score FROM students ORDER BY score DESC;
```

Run

如果`score`列有相同的数据，要进一步排序，可以继续添加列名。例如，使用`ORDER BY score DESC, gender`表示先按`score`列倒序，如果有相同分数的，再按`gender`列排序：

```
-- 按score, gender排序:
SELECT id, name, gender, score FROM students ORDER BY score DESC, gender;
```

Run

默认的排序规则是`ASC`：“升序”，即从小到大。`ASC`可以省略，即`ORDER BY score ASC`和`ORDER BY score`效果一样。

如果有`WHERE`子句，那么`ORDER BY`子句要放到`WHERE`子句后面。例如，查询一班的学生成绩，并按照倒序排序：

```
-- 带WHERE条件的ORDER BY:
SELECT id, name, gender, score
FROM students
WHERE class_id = 1
ORDER BY score DESC;
```

Run

这样，结果集仅包含符合`WHERE`条件的记录，并按照`ORDER BY`的设定排序。

### 小结

使用`ORDER BY`可以对结果集进行排序；

可以对多列进行升序、倒序排序。

# 分页查询

### 分页

使用SELECT查询时，如果结果集数据量很大，比如几万行数据，放在一个页面显示的话数据量太大，不如分页显示，每次显示100条。

要实现分页功能，实际上就是从结果集中显示第1~100条记录作为第1页，显示第101~200条记录作为第2页，以此类推。

因此，分页实际上就是从结果集中“截取”出第M~N条记录。这个查询可以通过`LIMIT <M> OFFSET <N>`子句实现。我们先把所有学生按照成绩从高到低进行排序：

```
-- 按score从高到低
SELECT id, name, gender, score FROM students ORDER BY score DESC;
```

Run

现在，我们把结果集分页，每页3条记录。要获取第1页的记录，可以使用`LIMIT 3 OFFSET 0`：

```
-- 查询第1页
```

```
SELECT id, name, gender, score
FROM students
ORDER BY score DESC
LIMIT 3 OFFSET 0;
```

Run

上述查询`LIMIT 3 OFFSET 0`表示，对结果集从0号记录开始，最多取3条。注意SQL记录集的索引从0开始。

如果要查询第2页，那么我们只需要“跳过”头3条记录，也就是对结果集从3号记录开始查询，把`OFFSET`设定为3：

```
-- 查询第2页
SELECT id, name, gender, score
FROM students
ORDER BY score DESC
LIMIT 3 OFFSET 3;
```

Run

类似的，查询第3页的时候，`OFFSET`应该设定为6:

```
-- 查询第3页
SELECT id, name, gender, score
FROM students
ORDER BY score DESC
LIMIT 3 OFFSET 6;
```

Run

查询第4页的时候，`OFFSET`应该设定为9:

```
-- 查询第4页
```

```
SELECT id, name, gender, score
FROM students
ORDER BY score DESC
LIMIT 3 OFFSET 9;
```

Run

由于第4页只有1条记录，因此最终结果集按实际数量1显示。`LIMIT 3`表示的意思是“最多3条记录”。

可见，分页查询的关键在于，首先要确定每页需要显示的结果数量`pageSize`（这里是3），然后根据当前页的索引`pageIndex`（从1开始），确定`LIMIT`和`OFFSET`应该设定的值：

- `LIMIT`总是设定为`pageSize`；
- `OFFSET`计算公式为`pageSize * (pageIndex - 1)`。

这样就能正确查询出第N页的记录集。

如果原本记录集一共就10条记录，但我们把`OFFSET`设置为20，会得到什么结果呢？

```
-- OFFSET设定为20
SELECT id, name, gender, score
FROM students
ORDER BY score DESC
LIMIT 3 OFFSET 20;
```

Run

`OFFSET`超过了查询的最大数量并不会报错，而是得到一个空的结果集。

## 注意

`OFFSET`是可选的，如果只写`LIMIT 15`，那么相当于`LIMIT 15 OFFSET 0`。

在MySQL中，`LIMIT 15 OFFSET 30`还可以简写成`LIMIT 30, 15`。

使用`LIMIT <M> OFFSET <N>`分页时，随着`N`越来越大，查询效率也会越来越低。

## 小结

使用`LIMIT <M> OFFSET <N>`可以对结果集进行分页，每次查询返回结果集的一部分；

分页查询需要先确定每页的数量和当前页数，然后确定`LIMIT`和`OFFSET`的值。

思考

在分页查询之前，如何计算一共有几页？

# 聚合查询

如果我们要统计一张表的数据量，例如，想查询`students`表一共有多少条记录，难道必须用`SELECT * FROM students`查出来然后再数一数有多少行吗？

这个方法当然可以，但是比较弱智。对于统计总数、平均数这类计算，SQL提供了专门的聚合函数，使用聚合函数进行查询，就是聚合查询，它可以快速获得结果。

仍然以查询`students`表一共有多少条记录为例，我们可以使用SQL内置的`COUNT()`函数查询：

```
-- 使用聚合查询:
SELECT COUNT(*) FROM students;
```

Run

`COUNT(*)`表示查询所有列的行数，要注意聚合的计算结果虽然是一个数字，但查询的结果仍然是一个二维表，只是这个二维表只有一行一列，并且列名是`COUNT(*)`。

通常，使用聚合查询时，我们应该给列名设置一个别名，便于处理结果：

```
-- 使用聚合查询并设置结果集的列名为num:
SELECT COUNT(*) num FROM students;
```

Run

`COUNT(*)`和`COUNT(id)`实际上是一样的效果。另外注意，聚合查询同样可以使用`WHERE`条件，因此我们可以方便地统计出有多少男生、多少女生、多少80分以上的学生等：

```
-- 使用聚合查询并设置WHERE条件:
```

```
SELECT COUNT(*) boys FROM students WHERE gender = 'M';
```

Run

除了`COUNT()`函数外，SQL还提供了如下聚合函数：

| 函数 | 说明 |
| --- | --- |
| SUM | 计算某一列的合计值，该列必须为数值类型 |
| AVG | 计算某一列的平均值，该列必须为数值类型 |
| MAX | 计算某一列的最大值 |
| MIN | 计算某一列的最小值 |

注意，`MAX()`和`MIN()`函数并不限于数值类型。如果是字符类型，`MAX()`和`MIN()`会返回排序最后和排序最前的字符。

要统计男生的平均成绩，我们用下面的聚合查询：

```
-- 使用聚合查询计算男生平均成绩:
SELECT AVG(score) average FROM students WHERE gender = 'M';
```

Run

要特别注意：如果聚合查询的`WHERE`条件没有匹配到任何行，`COUNT()`会返回0，而`SUM()`、`AVG()`、`MAX()`和`MIN()`会返回`NULL`：

```
-- WHERE条件gender = 'X'匹配不到任何行:
```

```
SELECT AVG(score) average FROM students WHERE gender = 'X';
```

Run

每页3条记录，如何通过聚合查询获得总页数？

- [ ] SELECT COUNT(*) / 3 FROM students;
- [ ] SELECT FLOOR(COUNT(*) / 3) FROM students;
- [ ] SELECT CEILING(COUNT(*) / 3) FROM students;

Submit

## 分组

如果我们要统计一班的学生数量，我们知道，可以用`SELECT COUNT(*) num FROM students WHERE class_id = 1;`。如果要继续统计二班、三班的学生数量，难道必须不断修改`WHERE`条件来执行`SELECT`语句吗？

对于聚合查询，SQL还提供了“分组聚合”的功能。我们观察下面的聚合查询：

```
-- 按class_id分组:
SELECT COUNT(*) num FROM students GROUP BY class_id;
```

Run

执行这个查询，`COUNT()`的结果不再是一个，而是3个，这是因为，`GROUP BY`子句指定了按`class_id`分组，因此，执行该`SELECT`语句时，会把`class_id`相同的列先分组，再分别计算，因此，得到了3行结果。

但是这3行结果分别是哪三个班级的，不好看出来，所以我们可以把`class_id`列也放入结果集中：

```
-- 按class_id分组:
```

```
SELECT class_id, COUNT(*) num FROM students GROUP BY class_id;
```

Run

这下结果集就可以一目了然地看出各个班级的学生人数。我们再试试把`name`放入结果集：

```
-- 按class_id分组:
SELECT name, class_id, COUNT(*) num FROM students GROUP BY class_id;
```

Run

不出意外，执行这条查询我们会得到一个语法错误，因为在任意一个分组中，只有`class_id`都相同，`name`是不同的，SQL引擎不能把多个`name`的值放入一行记录中。因此，聚合查询的列中，只能放入分组的列。

注意：AlaSQL并没有严格执行SQL标准，上述SQL在浏览器可以正常执行，但是在MySQL、Oracle等环境下将报错，请自行在MySQL中测试。

也可以使用多个列进行分组。例如，我们想统计各班的男生和女生人数：

```
-- 按class_id, gender分组:
SELECT class_id, gender, COUNT(*) num FROM students GROUP BY class_id, gender;
```

Run

上述查询结果集一共有6条记录，分别对应各班级的男生和女生人数。

### 练习

请使用一条SELECT查询查出每个班级的平均分：

```
-- 查出每个班级的平均分，结果集应当有3条记录：
SELECT 'TODO';
```

Run

请使用一条SELECT查询查出每个班级男生和女生的平均分：

```
-- 查出每个班级的平均分，结果集应当有6条记录：
SELECT 'TODO';
```

Run

### 小结

使用SQL提供的聚合查询，我们可以方便地计算总数、合计值、平均值、最大值和最小值；

聚合查询也可以添加`WHERE`条件。

## 多表查询

SELECT查询不但可以从一张表查询数据，还可以从多张表同时查询数据。查询多张表的语法是：`SELECT * FROM <表1> <表2>`。

例如，同时从`students`表和`classes`表的“乘积”，即查询数据，可以这么写：

```
-- FROM students, classes:
```

```
SELECT * FROM students, classes;
```

Run

这种一次查询两个表的数据，查询的结果也是一个二维表，它是`students`表和`classes`表的“乘积”，即`students`表的每一行与`classes`表的每一行都两两拼在一起返回。结果集的列数是`students`表和`classes`表的列数之和，行数是`students`表和`classes`表的行数之积。

这种多表查询又称笛卡尔查询，使用笛卡尔查询时要非常小心，由于结果集是目标表的行数乘积，对两个各自有100行记录的表进行笛卡尔查询将返回1万条记录，对两个各自有1万行记录的表进行笛卡尔查询将返回1亿条记录。

你可能还注意到了，上述查询的结果集有两列`id`和两列`name`，两列`id`是因为其中一列是`students`表的`id`，而另一列是`classes`表的`id`，但是在结果集中，不好区分。两列`name`同理

要解决这个问题，我们仍然可以利用投影查询的“设置列的别名”来给两个表各自的`id`和`name`列起别名：

```
-- set alias:
SELECT
    students.id sid,
    students.name,
    students.gender,
    students.score,
    classes.id cid,
    classes.name cname
FROM students, classes;
```

Run

注意，多表查询时，要使用`表名.列名`这样的方式来引用列和设置别名，这样就避免了结果集的列名重复问题。但是，用`表名.列名`这种方式列举两个表的所有列实在是很麻烦，所以SQL还允许给表设置一个别名，让我们在投影查询中引用起来稍微简洁一点：

```
-- set table alias:
```

```
SELECT
    s.id sid,
    s.name,
    s.gender,
    s.score,
    c.id cid,
    c.name cname
FROM students s, classes c;
```

Run

注意到`FROM`子句给表设置别名的语法是`FROM <表名1> <别名1>, <表名2> <别名2>`。这样我们用别名`s`和`c`分别表示`students`表和`classes`表。

多表查询也是可以添加`WHERE`条件的，我们来试试：

```
-- set where clause:
SELECT
    s.id sid,
    s.name,
    s.gender,
    s.score,
    c.id cid,
    c.name cname
FROM students s, classes c
WHERE s.gender = 'M' AND c.id = 1;
```

Run

这个查询的结果集每行记录都满足条件`s.gender = 'M'`和`c.id = 1`。添加`WHERE`条件后结果集的数量大大减少了。

### 小结

使用多表查询可以获取M x N行记录；

多表查询的结果集可能非常巨大，要小心使用。

# 连接查询

连接查询是另一种类型的多表查询。连接查询对多个表进行JOIN运算，简单地说，就是先确定一个主表作为结果集，然后，把其他表的行有选择性地“连接”在主表结果集上。

例如，我们想要选出`students`表的所有学生信息，可以用一条简单的SELECT语句完成：

```
-- 选出所有学生
```

```
SELECT s.id, s.name, s.class_id, s.gender, s.score FROM students s;
```

Run

但是，假设我们希望结果集同时包含所在班级的名称，上面的结果集只有`class_id`列，缺少对应班级的`name`列。

现在问题来了，存放班级名称的`name`列存储在`classes`表中，只有根据`students`表的`class_id`，找到`classes`表对应的行，再取出`name`列，就可以获得班级名称。

这时，连接查询就派上了用场。我们先使用最常用的一种内连接——INNER JOIN来实现：

```
-- 选出所有学生，同时返回班级名称
SELECT s.id, s.name, s.class_id, c.name class_name, s.gender, s.score
FROM students s
INNER JOIN classes c
ON s.class_id = c.id;
```

Run

注意INNER JOIN查询的写法是：

1. 先确定主表，仍然使用`FROM <表1>`的语法；
2. 再确定需要连接的表，使用`INNER JOIN <表2>`的语法；
3. 然后确定连接条件，使用`ON <条件...>`，这里的条件是`s.class_id = c.id`，表示`students`表的`class_id`列与`classes`表的`id`列相同的行需要连接；
4. 可选：加上`WHERE`子句、`ORDER BY`等子句。

使用别名不是必须的，但可以更好地简化查询语句。

那什么是内连接（INNER JOIN）呢？先别着急，有内连接（INNER JOIN）就有外连接（OUTER JOIN）。我们把内连接查询改成外连接查询，看看效果：

```
-- 使用OUTER JOIN
```

```
SELECT s.id, s.name, s.class_id, c.name class_name, s.gender, s.score
FROM students s
RIGHT OUTER JOIN classes c
ON s.class_id = c.id;
```

Run

执行上述RIGHT OUTER JOIN可以看到，和INNER JOIN相比，RIGHT OUTER JOIN多了一行，多出来的一行是“四班”，但是，学生相关的列如`name`、`gender`、`score`都为`NULL`。

这也容易理解，因为根据`ON`条件`s.class_id = c.id`，`classes`表的id=4的行正是“四班”，但是，`students`表中并不存在class_id=4的行。

有RIGHT OUTER JOIN，就有LEFT OUTER JOIN，以及FULL OUTER JOIN。它们的区别是：

INNER JOIN只返回同时存在于两张表的行数据，由于`students`表的`class_id`包含1，2，3，`classes`表的`id`包含1，2，3，4，所以，INNER JOIN根据条件`s.class_id = c.id`返回的结果集仅包含1，2，3。

RIGHT OUTER JOIN返回右表都存在的行。如果某一行仅在右表存在，那么结果集就会以`NULL`填充剩下的字段。

LEFT OUTER JOIN则返回左表都存在的行。如果我们给students表增加一行，并添加class_id=5，由于classes表并不存在id=5的行，所以，LEFT OUTER JOIN的结果会增加一行，对应的`class_name`是`NULL`：

```
-- 先增加一列class_id=5:
INSERT INTO students (class_id, name, gender, score) values (5, '新生', 'M', 88);
-- 使用LEFT OUTER JOIN
```

```
SELECT s.id, s.name, s.class_id, c.name class_name, s.gender, s.score
FROM students s
LEFT OUTER JOIN classes c
ON s.class_id = c.id;
```

Run

最后，我们使用FULL OUTER JOIN，它会把两张表的所有记录全部选择出来，并且，自动把对方不存在的列填充为NULL：

```
-- 使用FULL OUTER JOIN
```

```
SELECT s.id, s.name, s.class_id, c.name class_name, s.gender, s.score
FROM students s
FULL OUTER JOIN classes c
ON s.class_id = c.id;
```

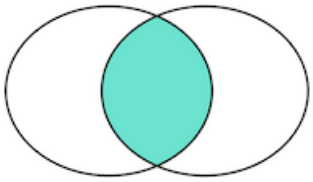
Run

对于这么多种JOIN查询，到底什么使用应该用哪种呢？其实我们用图来表示结果集就一目了然了。

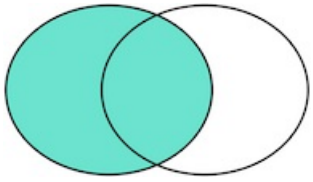
假设查询语句是：

```
SELECT ... FROM tableA ??? JOIN tableB ON tableA.column1 = tableB.column2;
```

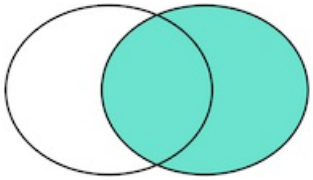
我们把tableA看作左表，把tableB看成右表，那么INNER JOIN是选出两张表都存在的记录：



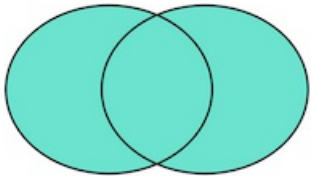
LEFT OUTER JOIN是选出左表存在的记录：



RIGHT OUTER JOIN是选出右表存在的记录：



FULL OUTER JOIN则是选出左右表都存在的记录：



## 小结

JOIN查询需要先确定主表，然后把另一个表的数据“附加”到结果集上；

INNER JOIN是最常用的一种JOIN查询，它的语法是`SELECT ... FROM <表1> INNER JOIN <表2> ON <条件...>`；

JOIN查询仍然可以使用`WHERE`条件和`ORDER BY`排序。

# 修改数据

关系数据库的基本操作就是增删改查，即CRUD：Create、Retrieve、Update、Delete。其中，对于查询，我们已经详细讲述了`SELECT`语句的详细用法。

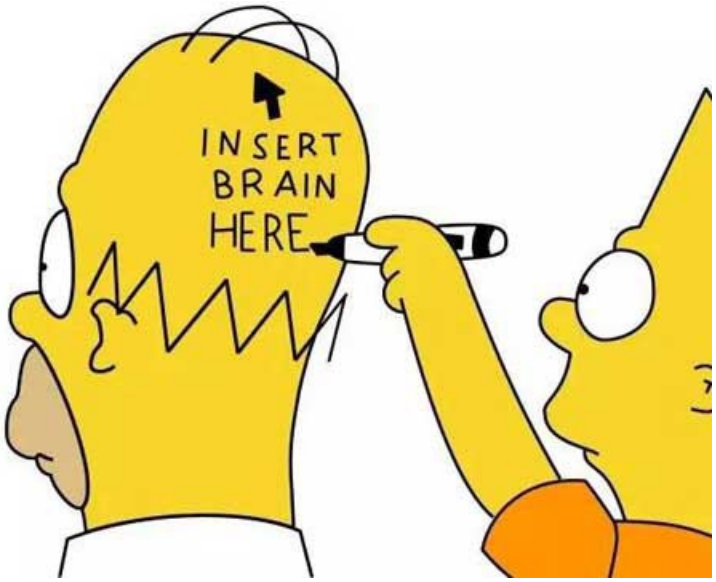
而对于增、删、改，对应的SQL语句分别是：

- INSERT：插入新记录；
- UPDATE：更新已有记录；
- DELETE：删除已有记录。

我们将分别讨论这三种修改数据的语句的使用方法。

# INSERT

当我们需要向数据库表中插入一条新记录时，就必须使用`INSERT`语句。




`INSERT`语句的基本语法是：

```
INSERT INTO <表名> (字段1, 字段2, ...) VALUES (值1, 值2, ...);
```

例如，我们向`students`表插入一条新记录，先列举出需要插入的字段名称，然后在`VALUES`子句中依次写出对应字段的值：

```
-- 添加一条新记录
```

```
INSERT INTO students (class_id, name, gender, score) VALUES (2, '大牛', 'M', 80);
-- 查询并观察结果:
SELECT * FROM students;
```

Run

注意到我们并没有列出`id`字段，也没有列出`id`字段对应的值，这是因为`id`字段是一个自增主键，它的值可以由数据库自己推算出来。此外，如果一个字段有默认值，那么在`INSERT`语句中也可以不出现。

要注意，字段顺序不必和数据库表的字段顺序一致，但值的顺序必须和字段顺序一致。也就是说，可以写`INSERT INTO students (score, gender, name, class_id) ...`，但是对应的`VALUES`就得变成`(80, 'M', '大牛', 2)`。

还可以一次性添加多条记录，只需要在`VALUES`子句中指定多个记录值，每个记录是由`(...)`包含的一组值：

```
-- 一次性添加多条新记录
INSERT INTO students (class_id, name, gender, score) VALUES
  (1, '大宝', 'M', 87),
  (2, '二宝', 'M', 81);

SELECT * FROM students;
```

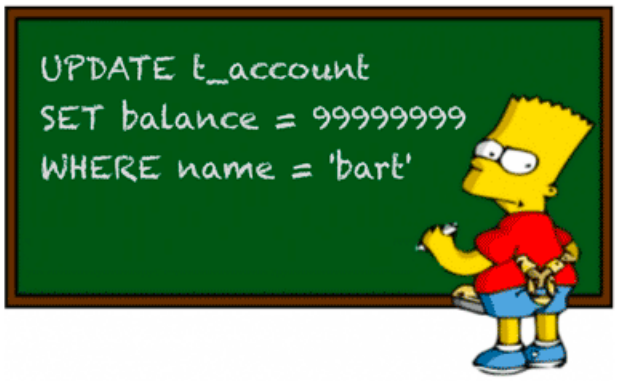
Run

### 小结

使用`INSERT`，我们就可以一次向一个表中插入一条或多条记录。

# UPDATE

如果要更新数据库表中的记录，我们就必须使用`UPDATE`语句。




`UPDATE`语句的基本语法是：

```
UPDATE <表名> SET 字段1=值1, 字段2=值2, ... WHERE ...;
```

例如，我们想更新`students`表`id=1`的记录的`name`和`score`这两个字段，先写出`UPDATE students SET name='大牛', score=66`，然后在`WHERE`子句中写出需要更新的行的筛选条件`id=1`：

```
-- 更新id=1的记录
UPDATE students SET name='大牛', score=66 WHERE id=1;
-- 查询并观察结果:
SELECT * FROM students WHERE id=1;
```

Run

注意到`UPDATE`语句的`WHERE`条件和`SELECT`语句的`WHERE`条件其实是一样的，因此完全可以一次更新多条记录：

```
-- 更新id=5,6,7的记录
UPDATE students SET name='小牛', score=77 WHERE id>=5 AND id<=7;
-- 查询并观察结果:
SELECT * FROM students;
```

Run

在`UPDATE`语句中，更新字段时可以使用表达式。例如，把所有80分以下的同学的成绩加10分：

```
-- 更新score<80的记录
UPDATE students SET score=score+10 WHERE score<80;
-- 查询并观察结果:
SELECT * FROM students;
```

Run

其中，`SET score=score+10`就是给当前行的`score`字段的值加上了10。

如果`WHERE`条件没有匹配到任何记录，`UPDATE`语句不会报错，也不会有任何记录被更新。例如：

```
-- 更新id=999的记录
UPDATE students SET score=100 WHERE id=999;
-- 查询并观察结果:
SELECT * FROM students;
```

Run

最后，要特别小心的是，`UPDATE`语句可以没有`WHERE`条件，例如：

```
UPDATE students SET score=60;
```

这时，整个表的所有记录都会被更新。所以，在执行`UPDATE`语句时要非常小心，最好先用`SELECT`语句来测试`WHERE`条件是否筛选出了期望的记录集，然后再用`UPDATE`更新。

## MySQL

在使用MySQL这类真正的关系数据库时，`UPDATE`语句会返回更新的行数以及`WHERE`条件匹配的行数。

例如，更新`id=1`的记录时：

```
mysql> UPDATE students SET name='大宝' WHERE id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

MySQL会返回`1`，可以从打印的结果`Rows matched: 1 Changed: 1`看到。

当更新`id=999`的记录时：

```
mysql> UPDATE students SET name='大宝' WHERE id=999;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

MySQL会返回`0`，可以从打印的结果`Rows matched: 0 Changed: 0`看到。

### 小结

使用`UPDATE`，我们就可以一次更新表中的一条或多条记录。

# DELETE

如果要删除数据库表中的记录，我们可以使用`DELETE`语句。


`DELETE`语句的基本语法是：

```
DELETE FROM <表名> WHERE ...;
```

例如，我们想删除`students`表中`id=1`的记录，就需要这么写：

```
-- 删除id=1的记录
DELETE FROM students WHERE id=1;
-- 查询并观察结果:
SELECT * FROM students;
```

Run

注意到`DELETE`语句的`WHERE`条件也是用来筛选需要删除的行，因此和`UPDATE`类似，`DELETE`语句也可以一次删除多条记录：

```
-- 删除id=5,6,7的记录
DELETE FROM students WHERE id>=5 AND id<=7;
-- 查询并观察结果:
SELECT * FROM students;
```

Run

如果`WHERE`条件没有匹配到任何记录，`DELETE`语句不会报错，也不会有任何记录被删除。例如：

```
-- 删除id=999的记录
DELETE FROM students WHERE id=999;
-- 查询并观察结果:
SELECT * FROM students;
```

Run

最后，要特别小心的是，和`UPDATE`类似，不带`WHERE`条件的`DELETE`语句会删除整个表的数据：

```
DELETE FROM students;
```

这时，整个表的所有记录都会被删除。所以，在执行`DELETE`语句时也要非常小心，最好先用`SELECT`语句来测试`WHERE`条件是否筛选出了期望的记录集，然后再用`DELETE`删除。

## MySQL

在使用MySQL这类真正的关系数据库时，`DELETE`语句也会返回删除的行数以及`WHERE`条件匹配的行数。

例如，分别执行删除`id=1`和`id=999`的记录：

```
mysql> DELETE FROM students WHERE id=1;
Query OK, 1 row affected (0.01 sec)

mysql> DELETE FROM students WHERE id=999;
Query OK, 0 rows affected (0.01 sec)
```

### 小结

使用`DELETE`，我们就可以一次删除表中的一条或多条记录。

# MySQL

安装完MySQL后，除了MySQL Server，即真正的MySQL服务器外，还附赠一个MySQL Client程序。MySQL Client是一个命令行客户端，可以通过MySQL Client登录MySQL，然后，输入SQL语句并执行。

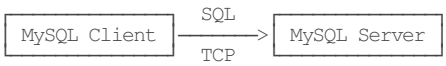
打开命令提示符，输入命令`mysql -u root -p`，提示输入口令。填入MySQL的root口令，如果正确，就连上了MySQL Server，同时提示符变为`mysql>`：

```
Command Prompt                                          - □ x
Microsoft Windows [Version 10.0.0]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\> mysql -u root -p
Enter password: ******

Server version: 5.7
Copyright (c) 2000, 2018, ...
Type 'help;' or '\h' for help.

mysql>
```

输入`exit`断开与MySQL Server的连接并返回到命令提示符。

> MySQL Client的可执行程序是mysql，MySQL Server的可执行程序是mysqld。

MySQL Client和MySQL Server的关系如下：




在MySQL Client中输入的SQL语句通过TCP连接发送到MySQL Server。默认端口号是3306，即如果发送到本机MySQL Server，地址就是`127.0.0.1:3306`。

也可以只安装MySQL Client，然后连接到远程MySQL Server。假设远程MySQL Server的IP地址是`10.0.1.99`，那么就使用`-h`指定IP或域名：

```
mysql -h 10.0.1.99 -u root -p
```

### 小结

命令行程序`mysql`实际上是MySQL客户端，真正的MySQL服务器程序是`mysqld`，在后台运行。

# 管理MySQL

要管理MySQL，可以使用可视化图形界面MySQL Workbench。

MySQL Workbench可以用可视化的方式查询、创建和修改数据库表，但是，归根到底，MySQL Workbench是一个图形客户端，它对MySQL的操作仍然是发送SQL语句并执行。因此，本质上，MySQL Workbench和MySQL Client命令行都是客户端，和MySQL交互，唯一的接口就是SQL。

因此，MySQL提供了大量的SQL语句用于管理。虽然可以使用MySQL Workbench图形界面来直接管理MySQL，但是，很多时候，通过SSH远程连接时，只能使用SQL命令，所以，了解并掌握常用的SQL管理操作是必须的。

## 数据库

在一个运行MySQL的服务器上，实际上可以创建多个数据库（Database）。要列出所有数据库，使用命令：

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| shici              |
| sys                |
| test               |
| school             |
+--------------------+
```

其中，`information_schema`、`mysql`、`performance_schema`和`sys`是系统库，不要去改动它们。其他的是用户创建的数据库。

要创建一个新数据库，使用命令：

```
mysql> CREATE DATABASE test;
Query OK, 1 row affected (0.01 sec)
```

要删除一个数据库，使用命令：

```
mysql> DROP DATABASE test;
Query OK, 0 rows affected (0.01 sec)
```

注意：删除一个数据库将导致该数据库的所有表全部被删除。

对一个数据库进行操作时，要首先将其切换为当前数据库：

```
mysql> USE test;
Database changed
```

## 表

列出当前数据库的所有表，使用命令：

```
mysql> SHOW TABLES;
+---------------------+
| Tables_in_test      |
+---------------------+
| classes             |
| statistics          |
| students            |
| students_of_class1  |
+---------------------+
```

要查看一个表的结构，使用命令：

```
mysql> DESC students;
+----------+--------------+------+-----+---------+----------------+
| Field    | Type         | Null | Key | Default | Extra          |
+----------+--------------+------+-----+---------+----------------+
| id       | bigint(20)   | NO   | PRI | NULL    | auto_increment |
| class_id | bigint(20)   | NO   |     | NULL    |                |
| name     | varchar(100) | NO   |     | NULL    |                |
| gender   | varchar(1)   | NO   |     | NULL    |                |
| score    | int(11)      | NO   |     | NULL    |                |
+----------+--------------+------+-----+---------+----------------+
5 rows in set (0.00 sec)
```

还可以使用以下命令查看创建表的SQL语句：

```
mysql> SHOW CREATE TABLE students;
+----------+-----------------------------------------------------------------+
| students | CREATE TABLE `students` (                                       |
|          |   `id` bigint(20) NOT NULL AUTO_INCREMENT,                      |
|          |   `class_id` bigint(20) NOT NULL,                               |
|          |   `name` varchar(100) NOT NULL,                                 |
|          |   `gender` varchar(1) NOT NULL,                                 |
|          |   `score` int(11) NOT NULL,                                     |
|          |   PRIMARY KEY (`id`)                                            |
|          | ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8           |
+----------+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

创建表使用`CREATE TABLE`语句，而删除表使用`DROP TABLE`语句：

```
mysql> DROP TABLE students;
Query OK, 0 rows affected (0.01 sec)
```

修改表就比较复杂。如果要给`students`表新增一列`birth`，使用：

```
ALTER TABLE students ADD COLUMN birth VARCHAR(10) NOT NULL;
```

要修改`birth`列，例如把列名改为`birthday`，类型改为`VARCHAR(20)`：

```
ALTER TABLE students CHANGE COLUMN birth birthday VARCHAR(20) NOT NULL;
```

要删除列，使用：

```
ALTER TABLE students DROP COLUMN birthday;
```

### 退出MySQL

使用`EXIT`命令退出MySQL：

```
mysql> EXIT
Bye
```

注意`EXIT`仅仅断开了客户端和服务器的连接，MySQL服务器仍然继续运行。

# 实用SQL语句

在编写SQL时，灵活运用一些技巧，可以大大简化程序逻辑。

### 插入或替换

如果我们希望插入一条新记录（INSERT），但如果记录已经存在，就先删除原记录，再插入新记录。此时，可以使用`REPLACE`语句，这样就不必先查询，再决定是否先删除再插入：

```
REPLACE INTO students (id, class_id, name, gender, score) VALUES (1, 1, '小明', 'F', 99);
```

若`id=1`的记录不存在，`REPLACE`语句将插入新记录，否则，当前`id=1`的记录将被删除，然后再插入新记录。

### 插入或更新

如果我们希望插入一条新记录（INSERT），但如果记录已经存在，就更新该记录，此时，可以使用`INSERT INTO ... ON DUPLICATE KEY UPDATE ...`语句：

```
INSERT INTO students (id, class_id, name, gender, score) VALUES (1, 1, '小明', 'F', 99) ON DUPLICATE KEY UPDATE name='小明', gender='F', score=99;
```

若`id=1`的记录不存在，`INSERT`语句将插入新记录，否则，当前`id=1`的记录将被更新，更新的字段由`UPDATE`指定。

### 插入或忽略

如果我们希望插入一条新记录（INSERT），但如果记录已经存在，就啥事也不干直接忽略，此时，可以使用`INSERT IGNORE INTO ...`语句：

```
INSERT IGNORE INTO students (id, class_id, name, gender, score) VALUES (1, 1, '小明', 'F', 99);
```

若`id=1`的记录不存在，`INSERT`语句将插入新记录，否则，不执行任何操作。

### 快照

如果想要对一个表进行快照，即复制一份当前表的数据到一个新表，可以结合`CREATE TABLE`和`SELECT`：

```
-- 对class_id=1的记录进行快照，并存储为新表students_of_class1:
CREATE TABLE students_of_class1 SELECT * FROM students WHERE class_id=1;
```

新创建的表结构和`SELECT`使用的表结构完全一致。

### 写入查询结果集

如果查询结果集需要写入到表中，可以结合`INSERT`和`SELECT`，将`SELECT`语句的结果集直接插入到指定表中。

例如，创建一个统计成绩的表`statistics`，记录各班的平均成绩：

```
CREATE TABLE statistics (
    id BIGINT NOT NULL AUTO_INCREMENT,
    class_id BIGINT NOT NULL,
    average DOUBLE NOT NULL,
    PRIMARY KEY (id)
);
```

然后，我们就可以用一条语句写入各班的平均成绩：

```
INSERT INTO statistics (class_id, average) SELECT class_id, AVG(score) FROM students GROUP BY class_id;
```

确保`INSERT`语句的列和`SELECT`语句的列能一一对应，就可以在`statistics`表中直接保存查询的结果：

```
> SELECT * FROM statistics;
+----+----------+--------------+
| id | class_id | average      |
+----+----------+--------------+
|  1 |        1 |         86.5 |
|  2 |        2 | 73.666666666 |
|  3 |        3 | 88.333333333 |
+----+----------+--------------+
3 rows in set (0.00 sec)
```

### 强制使用指定索引

在查询的时候，数据库系统会自动分析查询语句，并选择一个最合适的索引。但是很多时候，数据库系统的查询优化器并不一定总是能使用最优索引。如果我们知道如何选择索引，可以使用`FORCE INDEX`强制查询使用指定的索引。例如：

```
> SELECT * FROM students FORCE INDEX (idx_class_id) WHERE class_id = 1 ORDER BY id DESC;
```

指定索引的前提是索引`idx_class_id`必须存在。

# 事务

在执行SQL语句的时候，某些业务要求，一系列操作必须全部执行，而不能仅执行一部分。例如，一个转账操作：

```
-- 从id=1的账户给id=2的账户转账100元
-- 第一步：将id=1的A账户余额减去100
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
-- 第二步：将id=2的B账户余额加上100
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
```

这两条SQL语句必须全部执行，或者，由于某些原因，如果第一条语句成功，第二条语句失败，就必须全部撤销。

这种把多条语句作为一个整体进行操作的功能，被称为数据库事务。数据库事务可以确保该事务范围内的所有操作都可以全部成功或者全部失败。如果事务失败，那么效果就和没有执行这些SQL一样，不会对数据库数据有任何改动。

可见，数据库事务具有ACID这4个特性：

- A：Atomic，原子性，将所有SQL作为原子工作单元执行，要么全部执行，要么全部不执行；
- C：Consistent，一致性，事务完成后，所有数据的状态都是一致的，即A账户只要减去了100，B账户则必定加上了100；

- I：Isolation，隔离性，如果有多个事务并发执行，每个事务作出的修改必须与其他事务隔离；
- D：Duration，持久性，即事务完成后，对数据库数据的修改被持久化存储。

对于单条SQL语句，数据库系统自动将其作为一个事务执行，这种事务被称为隐式事务。

要手动把多条SQL语句作为一个事务执行，使用`BEGIN`开启一个事务，使用`COMMIT`提交一个事务，这种事务被称为显式事务，例如，把上述的转账操作作为一个显式事务：

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

很显然多条SQL语句要想作为一个事务执行，就必须使用显式事务。

`COMMIT`是指提交事务，即试图把事务内的所有SQL所做的修改永久保存。如果`COMMIT`语句执行失败了，整个事务也会失败。

有些时候，我们希望主动让事务失败，这时，可以用`ROLLBACK`回滚事务，整个事务会失败：

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
ROLLBACK;
```

数据库事务是由数据库系统保证的，我们只需要根据业务逻辑使用它就可以。

### 隔离级别

对于两个并发执行的事务，如果涉及到操作同一条记录的时候，可能会发生问题。因为并发操作会带来数据的不一致性，包括脏读、不可重复读、幻读等。数据库系统提供了隔离级别来让我们有针对性地选择事务的隔离级别，避免数据不一致的问题。

SQL标准定义了4种隔离级别，分别对应可能出现的数据不一致的情况：

| Isolation Level | 脏读（Dirty Read） | 不可重复读（Non Repeatable Read） | 幻读（Phantom Read） |
|---|---|---|---|
| Read Uncommitted | Yes | Yes | Yes |
| Read Committed | - | Yes | Yes |
| Repeatable Read | - | - | Yes |
| Serializable | - | - | - |

我们会依次介绍4种隔离级别的数据一致性问题。

### 小结

数据库事务具有ACID特性，用来保证多条SQL的全部执行。

# Read Uncommitted

Read Uncommitted是隔离级别最低的一种事务级别。在这种隔离级别下，一个事务会读到另一个事务更新后但未提交的数据，如果另一个事务回滚，那么当前事务读到的数据就是脏数据，这就是脏读（Dirty Read）。

我们来看一个例子。

首先，我们准备好`students`表的数据，该表仅一行记录：

```
mysql> select * from students;
+----+-------+
| id | name  |
+----+-------+
|  1 | Alice |
+----+-------+
1 row in set (0.00 sec)
```

然后，分别开启两个MySQL客户端连接，按顺序依次执行事务A和事务B：

| 时刻 | 事务A | 事务B |
| --- | --- | --- |
| 1 | SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; | SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; |
| 2 | BEGIN; | BEGIN; |
| 3 | UPDATE students SET name = 'Bob' WHERE id = 1; | |
| 4 | | SELECT * FROM students WHERE id = 1; |
| 5 | ROLLBACK; | |
| 6 | | SELECT * FROM students WHERE id = 1; |
| 7 | | COMMIT; |

当事务A执行完第3步时，它更新了`id=1`的记录，但并未提交，而事务B在第4步读取到的数据就是未提交的数据。

随后，事务A在第5步进行了回滚，事务B再次读取`id=1`的记录，发现和上一次读取到的数据不一致，这就是脏读。

可见，在Read Uncommitted隔离级别下，一个事务可能读取到另一个事务更新但未提交的数据，这个数据有可能是脏数据。

# Read Committed

在Read Committed隔离级别下，一个事务可能会遇到不可重复读（Non Repeatable Read）的问题。

不可重复读是指，在一个事务内，多次读同一数据，在这个事务还没有结束时，如果另一个事务恰好修改了这个数据，那么，在第一个事务中，两次读取的数据就可能不一致。

我们仍然先准备好`students`表的数据：

```
mysql> select * from students;
+----+-------+
| id | name  |
+----+-------+
|  1 | Alice |
+----+-------+
1 row in set (0.00 sec)
```

然后，分别开启两个MySQL客户端连接，按顺序依次执行事务A和事务B：

| 时刻 | 事务A | 事务B |
| --- | --- | --- |
| 1 | SET TRANSACTION ISOLATION LEVEL READ COMMITTED; | SET TRANSACTION ISOLATION LEVEL READ COMMITTED; |
| 2 | BEGIN; | BEGIN; |
| 3 | | SELECT * FROM students WHERE id = 1; |
| 4 | UPDATE students SET name = 'Bob' WHERE id = 1; | |
| 5 | COMMIT; | |
| 6 | | SELECT * FROM students WHERE id = 1; |

| 时刻 | 事务A | 事务B |
| --- | --- | --- |
| 7 | | COMMIT; |

当事务B第一次执行第3步的查询时，得到的结果是`Alice`，随后，由于事务A在第4步更新了这条记录并提交，所以，事务B在第6步再次执行同样的查询时，得到的结果就变成了`Bob`，因此，在Read Committed隔离级别下，事务不可重复读同一条记录，因为很可能读到的结果不一致。

# Repeatable Read

在Repeatable Read隔离级别下，一个事务可能会遇到幻读（Phantom Read）的问题。

幻读是指，在一个事务中，第一次查询某条记录，发现没有，但是，当试图更新这条不存在的记录时，竟然能成功，并且，再次读取同一条记录，它就神奇地出现了。

我们仍然先准备好`students`表的数据：

```
mysql> select * from students;
+----+-------+
| id | name  |
+----+-------+
|  1 | Alice |
+----+-------+
1 row in set (0.00 sec)
```

然后，分别开启两个MySQL客户端连接，按顺序依次执行事务A和事务B：

| 时刻 | 事务A | 事务B |
| --- | --- | --- |
| 1 | SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; | SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; |
| 2 | BEGIN; | BEGIN; |
| 3 | | SELECT * FROM students WHERE id = 99; |
| 4 | INSERT INTO students (id, name) VALUES (99, 'Bob'); | |
| 5 | COMMIT; | |
| 6 | | SELECT * FROM students WHERE id = 99; |
| 7 | | UPDATE students SET name = 'Alice' WHERE id = 99; |
| 8 | | SELECT * FROM students WHERE id = 99; |
| 9 | | COMMIT; |

事务B在第3步第一次读取`id=99`的记录时，读到的记录为空，说明不存在`id=99`的记录。随后，事务A在第4步插入了一条`id=99`的记录并提交。事务B在第6步再次读取`id=99`的记录时，读到的记录仍然为空，但是，事务B在第7步试图更新这条不存在的记录时，竟然成功了，并且，事务B在第8步再次读取`id=99`的记录时，记录出现了。

可见，幻读就是没有读到的记录，以为不存在，但其实是可以更新成功的，并且，更新成功后，再次读取，就出现了。

# Serializable

Serializable是最严格的隔离级别。在Serializable隔离级别下，所有事务按照次序依次执行，因此，脏读、不可重复读、幻读都不会出现。

虽然Serializable隔离级别下的事务具有最高的安全性，但是，由于事务是串行执行，所以效率会大大下降，应用程序的性能会急剧降低。如果没有特别重要的情景，一般都不会使用Serializable隔离级别。

## 默认隔离级别

如果没有指定隔离级别，数据库就会使用默认的隔离级别。在MySQL中，如果使用InnoDB，默认的隔离级别是Repeatable Read。

# 开发集成

即将推出，请耐心等待。等待不耐烦的，请关注微博@廖雪峰

# 期末总结

即将推出，请耐心等待。等待不耐烦的，请关注微博@廖雪峰