mysql入门，晋级，存储过程，mysql面试题，老杜mysql数据库全套视频教程精讲_bilibili

> 学习目标:
> 把视频里的**所有视频都学会**，通俗简单易懂。这个阶段**一定多要敲代码。多写sql**，学习目标就是**会写sql语句**，**会使用mysql**。
>
> 课程评价:
> 非常硬核的一门课程,没有过度进阶的内容，这个阶段也不需要学习非常高阶的东西，会使用是最重要的。

第一章.数据库概述

什么是数据库

- 数据库是一门独立的学科，只要是做软件开发的，数据库都要学。
- 数据库（电子化的文件柜）是“**按照数据结构来组织**、存储和管理数据的仓库”。是一个**长期存储在计算机内**的、有组织的、可共享的、统一管理的大量数据的集合。　（数据存储在文件中，所以DB本质上是文件）
- 它的**存储空间很大**，可以存放百万条、千万条、上亿条数据。
- 数据库并不是随意地将数据进行存放，是**有一定的规则的**，否则查询的效率会很低。
- 当今世界是一个充满着数据的互联网世界，充斥着大量的数据。即这个互联网世界就是数据世界。数据的来源有很多，比如出行记录、消费记录、浏览的网页、发送的消息等等。除了文本类型的数据，图像、音乐、声音都是数据。
- 数据库对应的英文单词是DataBase，简称DB。

数据库类型

- **关系型数据库**
  - 关系型数据库是依据关系模型来创建的数据库。所谓关系模型就是“**一对一、一对多、多对多”等关系模型**，**关系模型就是指二维表格模型**，因而**一个关系型数据库就是由二维表及其之间的联系组成的一个数据组织。**
  - 关系型数据可以很好地存储一些关系模型的数据，比如一个老师对应多个学生的数据（“多对多”），一本书对应多个作者（“一对多”），一本书对应一个出版日期（“一对一”）。
  - 关系模型包括**数据结构**（数据存储的问题，二维表）、**操作指令集合**（SQL语句）、**完整性约束**(表内数据约束、表与表之间的约束)。
- **非关系型数据库（NoSQL）**

- NoSQL，泛指非关系型的数据库。随着互联网web2.0网站的兴起，传统的关系数据库在处理web2.0网站，特别是超大规模和高并发的SNS类型的web2.0纯动态网站已经显得力不从心，出现了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。
- NoSQL数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，特别是大数据应用难题。NoSQL最常见的解释是“non-relational”，“Not Only SQL”也被很多人接受。
- NoSQL仅仅是一个概念，泛指非关系型的数据库，区别于关系数据库，**它们不保证关系数据的ACID特性。**NoSQL是一项全新的数据库革命性运动，其拥护者们提倡运用非关系型的数据存储，相对于铺天盖地的关系型数据库运用，这一概念无疑是一种全新的思维的注入。
- NoSQL有如下优点：**易扩展**，NoSQL数据库种类繁多，但是一个**共同的特点都是去掉关系数据库的关系型特性。数据之间无关系**，这样就非常容易扩展。无形之间也在架构的层面上带来了可扩展的能力。大数据量，**高性能**，NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。这**得益于它的无关系性，数据库的结构简单。**

数据库管理系统

- 数据库管理系统（Database Management System，简称DBMS）是为管理数据库而设计的电脑**软件系统**，一般具有存储、截取、安全保障、备份等基础功能。
- 数据库管理系统是数据库系统的核心组成部分，主要完成对数据库的操作与管理功能，实现数据库对象的创建、数据库存储数据的查询、添加、修改与删除操作和数据库的用户管理、权限管理等。
- 常见的数据库管理系统有：MySQL、Oracle、DB2、MS SQL Server、SQLite、PostgreSQL、Sybase等。

什么是SQL

- 结构化查询语言（**Structured Query Language**）简称SQL，是一种特殊目的的**编程语言**，是**一种数据库查询和程序设计语言**，用于存取数据以及查询、更新和管理关系数据库系统。
- 结构化查询语言是**高级的非过程化编程语言**，允许用户在高层数据结构上工作。它不要求用户指定对数据的存放方法，也不需要用户了解具体的数据存放方式，所以**具有完全不同底层结构的不同数据库系统，可以使用相同的结构化查询语言**作为数据输入与管理的接口。结构化查询语言语句可以嵌套，这使它具有极大的灵活性和强大的功能。

- SQL的分类
  - DQL
    - 数据查询语言（**Data Query** Language, DQL）是SQL语言中，**负责进行数据查询而不会对数据本身进行修改**的语句，这是最基本的SQL语句。保留字`SELECT`是DQL（也是所有SQL）用得最多的动词，其他DQL常用的保留字有`FROM`，`WHERE`，`GROUP BY`，`HAVING`和`ORDER BY`。这些DQL保留字常与其他类型的SQL语句一起使用。
  - DDL
    - 数据定义语言 (**Data Definition** Language, DDL) 是SQL语言集中，**负责数据结构定义与数据库对象定义**的语言，由`CREATE`、`ALTER`与`DROP`三个语法所组成，最早是由 Codasyl (Conference on Data Systems Languages) 数据模型开始，现在被纳入 SQL 指令中作为其中一个子集。

      > 改的是表结构。跟表数据无关。
  - DML
    - 数据操纵语言（**Data Manipulation** Language, DML）是SQL语言中，**负责对数据库对象运行数据访问工作**的指令集，以`INSERT`、`UPDATE`、`DELETE`三种指令为核心，分别代表插入、更新与删除。
  - DCL
    - 数据控制语言 (**Data Control** Language) 在SQL语言中，是一种**可对数据访问权进行控制**的指令，它可以控制特定用户账户对数据表、查看表、预存程序、用户自定义函数等数据库对象的控制权。由`GRANT`和`REVOKE`两个指令组成。DCL以控制用户的访问权限为主，GRANT为授权语句，对应的REVOKE是撤销授权语句。
  - TPL
    - 数据事务管理语言（**Transaction Processing** Language）它的语句能**确保被DML语句影响的表的所有行及时得以更新**。TPL语句包括`BEGIN TRANSACTION`，`COMMIT`和`ROLLBACK`。
  - CCL
    - 指针控制语言（**Cursor Control** Language），它的语句，像`DECLARE CURSOR`，`FETCH INTO`和`UPDATE WHERE CURRENT`用于对一个或多个表单独行的操作。
- DBMS、SQL、DB之间的关系
  - DBMS通过执行 SQL 来操作 DB 中的数据。

- 注意：不同的数据库管理系统使用的具体SQL语句的语法会有细微不同！尽管SQL是一个标准化的语言，但不同的数据库系统对标准的实现程度不同。有些系统可能更严格地遵循标准，而有些则可能有自己的扩展。

第二章.MySQL安装

MySQL概述

- MySQL是一个**关系型数据库管理系统**，由瑞典MySQL AB公司开发，MySQL AB公司被Sun公司收购，Sun公司又被Oracle公司收购，目前属于Oracle公司。
- MySQL是目前最流行的关系型数据库管理系统，在WEB应用方面MySQL是最好的RDBMS应用软件之一。 国内淘宝网站就使用的是MySQL集群。
- MySQL特点
    - MySQL有开源版本和收费版本，你使用开源版本是不收费的。
    - MySQL支持大型数据库，可以处理上千万记录的大型数据库。
    - MySQL**使用标准的SQL数据库语言**形式。
    - MySQL在**很多系统上面都支持。**
    - MySQL**对Java，C都有很好的支持**，当然其他的语言也支持比如Python、PHP。
    - MySQL是可以定制的，采用了GPL协议，你可以修改源码来开发自己的MySQL系统。

MySQL的下载

第一步：打开MySQL官网https://www.mysql.com/

第二步：点击"DOWNLOADS"

第三步：当前页继续下拉，直到找到链接：MySQL Community(GPL) Downloads

第四步：点击链接，进入页面，其中“MySQL Community Server”是解压版mysql，“MySQL Installer for Windows”是安装版。

MySQL安装与配置

启动MySQL服务：`net start mysql80`，注意start后面是mysql服务的名称

停止mysql服务的命令：`net stop mysql`

登录mysql：输入`mysql -uroot -p`，然后回车，输入密码，然后回车

修改MySQL的root账户密码：ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY '新密码';

登录MySQL

本地登录：

如果mysql的服务是启动的，打开dos命令窗口，输入：mysql -uroot -p，回车，然后输入root账户的密码。

```
mysql -u"用户名" -p
```

> 解释“mysql -uroot -p”：
> mysql是一个命令，在bin目录下，对应的命令文件是mysql.exe，如果将bin目录配置到环境变量path中，才可以在以上位置使用该命令。
> -uroot 表示登录的用户是root，u实际上是user单词的首字母。
> -p 表示登录时使用密码，p实际上是password单词的首字母。

也可以将密码以明文的形式写到-p后面，但这样做可能会导致你的密码泄露。

远程登录：

假设mysql安装在A机器上，现在你要在B机器上连接mysql数据库，此时需要使用远程登录，远程登录时加上远程机器的ip地址即可：

```
mysql -u"用户名" -h"IP地址" -p"密码"
```

> -h中的h实际上是host单词的首字母。在-h后面的是远程计算机的ip地址。
> 127.0.0.1是计算机默认的本机IP地址。
> 127.0.0.1又可以写作：localhost，他们是等效的。

注意：mysql默认情况下root账户是不支持远程登录的，其实这是一种安全策略，为了保护root账户的安全。如果希望root账户支持远程登录，这是需要进行设置的。

mysql8 开放root账户远程登录权限（危险动作）

- 第一步：先在本地使用root账户登录mysql

- 第二步：use mysql;
- 第三步：update user set host = '%' where user = 'root';
- 第四步：flush privileges;

第三章.初始化数据

MySQL命令行基本命令

1. 列出当前数据库管理系统中有哪些数据库。

```
show databases;             #  注意最后的分号 ！
```

2. 创建数据库，起名bjpowernode。

```
create database bjpowernode;
```

3. 使用bjpowernode数据库。

```
use bjpowernode;
```

4. 查看当前用的是哪个数据库。

```
select database();
```

5. 查看当前数据库中有哪些表。

```
show tables;
```

6. 删除数据库bjpowernode。

```
drop database bjpowernode;
```

7. 退出mysql
    1. exit
    2. quit
    3. ctrl + c
8. 查看当前mysql版本

```sql
select version();
```

还可以使用mysql.exe命令来查看版本信息（在没有登录mysql之前使用）：mysql --version

数据库表(table)的概述

name	age	gender
张三	20	男
李四	22	女

- 以上就是数据库表格的直观展示形式。
- 表是数据库存储数据的基本单元，数据库存储数据的时候，是将数据存储在表对象当中的。
- > 为什么将数据存储在表中呢？　因为表存储数据非常直观。
- 任何一张表都有行和列：
  - 行（Row）：记录（一行就是一条数据）
  - 列（Column）：字段（name字段、age字段、gender字段）
- 每个字段包含以下属性：
  - 字段名：name、age、gender都是字段的名字
  - 字段的数据类型：每个字段都有数据类型，比如：字符类型、数字类型、日期类型
  - 字段的数据长度：每个字段有可能会有长度的限制
  - 字段的约束：比如某些字段要求该字段下的数据不能重复、不能为空等，用来保证表格中数据合法有效

初始化测试数据

为了方便后面内容的学习，提前准备了表以及表中的测试数据，以下是建表并且初始化数据的sql脚本

```sql
DROP TABLE IF EXISTS EMP;
DROP TABLE IF EXISTS DEPT;
DROP TABLE IF EXISTS SALGRADE;

```

```sql
CREATE TABLE DEPT(DEPTNO int(2) not null ,
    DNAME VARCHAR(14) ,
    LOC VARCHAR(13),
    primary key (DEPTNO)
);
CREATE TABLE EMP(EMPNO int(4)  not null ,
    ENAME VARCHAR(10),
    JOB VARCHAR(9),
    MGR INT(4),
    HIREDATE DATE  DEFAULT NULL,
    SAL DOUBLE(7,2),
    COMM DOUBLE(7,2),
    primary key (EMPNO),
    DEPTNO INT(2)
);

CREATE TABLE SALGRADE( GRADE INT,
    LOSAL INT,
    HISAL INT
);

INSERT INTO DEPT ( DEPTNO, DNAME, LOC ) VALUES ( 10, 'ACCOUNTING', 'NEW YORK');
INSERT INTO DEPT ( DEPTNO, DNAME, LOC ) VALUES ( 20, 'RESEARCH', 'DALLAS');
INSERT INTO DEPT ( DEPTNO, DNAME, LOC ) VALUES ( 30, 'SALES', 'CHICAGO');
INSERT INTO DEPT ( DEPTNO, DNAME, LOC ) VALUES ( 40, 'OPERATIONS', 'BOSTON');

INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7369, 'SMITH', 'CLERK', 7902,  '1980-12-17', 800, NULL, 20);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7499, 'ALLEN', 'SALESMAN', 7698,  '1981-02-20', 1600, 300, 30);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7521, 'WARD', 'SALESMAN', 7698,  '1981-02-22', 1250, 500, 30);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7566, 'JONES', 'MANAGER', 7839,  '1981-04-02', 2975, NULL, 20);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7654, 'MARTIN', 'SALESMAN', 7698,  '1981-09-28', 1250, 1400, 30);
```

```sql
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7698, 'BLAKE', 'MANAGER', 7839, '1981-05-01', 2850, NULL, 30);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7782, 'CLARK', 'MANAGER', 7839, '1981-06-09', 2450, NULL, 10);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7788, 'SCOTT', 'ANALYST', 7566, '1987-04-19', 3000, NULL, 20);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7839, 'KING', 'PRESIDENT', NULL, '1981-11-17', 5000, NULL, 10);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7844, 'TURNER', 'SALESMAN', 7698, '1981-09-08', 1500, 0, 30);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7876, 'ADAMS', 'CLERK', 7788,  '1987-05-23', 1100, NULL, 20);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7900, 'JAMES', 'CLERK', 7698,  '1981-12-03', 950, NULL, 30);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7902, 'FORD', 'ANALYST', 7566,  '1981-12-03', 3000, NULL, 20);
INSERT INTO EMP ( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,DEPTNO ) VALUES ( 7934, 'MILLER', 'CLERK', 7782,  '1982-01-23', 1300, NULL, 10);

INSERT INTO SALGRADE ( GRADE, LOSAL, HISAL ) VALUES ( 1, 700, 1200);
INSERT INTO SALGRADE ( GRADE, LOSAL, HISAL ) VALUES ( 2, 1201, 1400);
INSERT INTO SALGRADE ( GRADE, LOSAL, HISAL ) VALUES ( 3, 1401, 2000);
INSERT INTO SALGRADE ( GRADE, LOSAL, HISAL ) VALUES ( 4, 2001, 3000);
INSERT INTO SALGRADE ( GRADE, LOSAL, HISAL ) VALUES ( 5, 3001, 9999);
commit;
```

- 什么是sql脚本：文件名是.sql，并且该文件中编写了大量的SQL语句，执行sql脚本程序就相当于批量执行SQL语句。
- 入职的时候，项目一般都是进展了一部分，多数情况下你进项目组的时候数据库的表以及数据都是有的，项目经理第一天可能会给你一个较大的sql脚本文件，你需要执行这个脚本文件来初始化你的本地数据库。（当然，也有可能数据库是共享的。）

初始化步骤：

- 创建文件：bjpowernode.sql，把以上SQL语句全部复制到sql脚本文件中。
- 执行SQL脚本文件，初始化数据库
    - 第一步：命令窗口登录mysql
    - 第二步：创建数据库bjpowernode（如果之前已经创建就不需要再创建了）：`create database bjpowernode;`
    - 第三步：使用数据库bjpowernode：`use bjpowernode;`
    - 第四步：source命令执行sql脚本，注意：source命令后面是sql脚本文件的绝对路径。 `source E:\bjpowernode.sql`
    - 第五步：查看是否初始化成功，执行：`show tables;`
    - 使用其他的mysql客户端工具也可以执行sql脚本，比如navicat。 使用source命令执行sql脚本的优点：**可支持大文件**(没有文件大小限制，但客户端工具可能有限制)。

测试数据介绍

- emp：员工信息
- dept：部门信息
- salgrade：工资等级信息

查看表结构：desc或describe，语法格式：`desc或describe + 表名;`

```
1 desc emp;
```

> 通过以上结果展示的不是表中的数据，而是表的结构。

- Field是字段名
- Type是这个字段的数据类型
- Null是这个字段是否允许为空
- Key是这个字段是否为主键或外键
- Default是这个字段的默认值

对以上表结构进行解释说明：

- emp表
    - empno：员工编号，int类型（整数），不能为空，主键（主键后期学习约束时会进行说明）

    - ename：员工姓名，varchar类型（字符串）
    - job：工作岗位，varchar类型
    - mgr：上级领导编号，int类型
    - hiredate：雇佣日期，date类型（日期类型）
    - sal：月薪，double类型（带有浮点的数字）
    - comm：提成(commission)，double类型
    - deptno：部门编号，int类型
- dept表
    - deptno：部门编号，int类型，主键
    - dname：部门名称，varchar类型
    - loc：位置，varchar类型
- salgrade表
    - grade：等级，int类型
    - losal：最低工资，int类型
    - hisal：最高工资，int类型

查询dept表中的全部数据：`select * from dept;`

第四章.查询DQL

简单查询

查询是SQL语言的核心，用于表达SQL查询的select查询命令是功能最强也最为复杂的SQL语句，它的作用就是从数据库中检索数据，并将查询结果返回给用户。 select语句由：select子句(查询内容)、from子句(查询对象)、where子句(查询条件)、order by子句(排序方式)、group by子句(分组方式)等组成。查询语句属于SQL语句中的DQL语句，是所有SQL语句中最为复杂也最重要的语句，所以必须掌握。

查一个字段

查询一个字段说的是：一个表有多列，查询其中的一列。

语法格式：`select 字段名 from 表名;`

- select和from是关键字，不能随便写
- **一条SQL语句必须以";"结尾**
- **对于SQL语句来说，不区分大小写 (包括字段名与表名)，可以大小混合**

- 字段名和表名属于标识符，按照表的实际情况填写，不知道字段名的，可以使用desc命令查看表结构

案例1：查询公司中所有员工编号

```
select empno from emp;
```

案例2：查询公司中所有员工姓名

```
SELECT ENAME FROM EMP;
```

在mysql命令行客户端中，sql语句没有分号是不会执行的，末尾加上";"就执行了。

假设一个SQL语句在书写过程中出错了，怎么终止这条SQL呢？ 在结尾用 '\c' 。

```
select
ename
from                # 可以回车分行，遇到 ";" 才会执行语句。可通过"\c"来终止
\c
```

查多个字段

查询多个字段时，在字段名和字段名之间添加 "," 即可。

语法格式：`select 字段名1,字段名2,字段名3 from 表名;`

案例1：查询员工编号以及员工姓名。

```
select empno, ename from emp;
```

> 字段的前后顺序无所谓（只是显示结果列的时候顺序变了).
>
> 也可以写成 `select ename, empno from emp;`

查所有字段

查询所有字段的可以将每个字段都列出来查询，也可以采用"*"来代表所有字段。

案例1：查询员工的所有信息

```
select * from emp;
```

案例2：查询所有部门信息

```
select * from dept;
```

采用 `*` 进行查询存在的缺点：

- `select * from dept;` 在执行的时候**会被解析为** `select DEPTNO, DNAME, LOC from dept;` 再执行，所以这种**效率方面弱**一些。
- 采用 `*` 的**可读性较差**，通过 `*` 很难看出都有哪些具体的字段。

什么时候使用 `*` ？

- 这个SQL语句不在项目编码中使用，如果平时自己想快速查看表中所有数据的话，这种写法还是很实用的。

查询时字段可参与数学运算

在进行查询操作的时候，字段是可以参与数学运算的，例如加减乘除等。

案例1：查询每个员工的月薪

```
select ename, sal from emp;
```

案例2：查询每个员工的年薪（月薪 * 12）

```
select ename, sal * 12 from emp;
```

查询时字段可起别名

给查询结果的列名进行重命名：

as关键字

- 使用as关键字

```
select ename, sal * 12 as yearsal from emp;
```

> 通过as关键字起别名后，查询结果列显示yearsal，可读性增强。

省略as关键字

- 其实as关键字可以省略，只要使用空格即可

```
select ename, sal * 12 yearsal from emp;
```

如果别名中有空格

```
select ename, sal * 12 year sal from emp;
```

可以看出，执行报错了，说语法有问题。分析一下：SQL语句编译器在检查该语句的时候，在year后面遇到了空格，会继续找from关键字，但year后面不是from关键字，所以编译器报错了。 怎么解决？如果别名中有空格的话，可以将这个别名**使用双引号或者单引号**将其括起来。

```
select ename, sal * 12 "year sal" from emp;
select ename, sal * 12 'year sal' from emp;
```

在mysql中，字符串既可以使用双引号也可以使用单引号，但还是**建议使用单引号**，因为单引号属于标准SQL。

> 双引号在别的数据库如Oracle可能失效。

别名中有中文

- 如果别名采用中文呢?

```
select ename, sal * 12 年薪 from emp;
```

别名是中文是可以的，但是对于低版本的mysql来说会报错，需要添加双引号或单引号。

条件查询

通常在进行查询操作的时候，都是查询符合某些条件的数据，很少将表中所有数据都取出来。怎么取出表的部分数据？需要在查询语句中添加条件进行数据的过滤。常见的过滤条件如下：

条件	说明
=	等于

条件	说明
<> 或 !=	不等于
>=	大于等于
<=	小于等于
>	大于
<	小于
between...and...	等同于 >= and <=
is null	为空
is not null	不为空
<=>	安全等于（可读性差，很少使用了）。
and 或 &&	并且
or 或 ||	或者
in	在指定的值当中
not in	不在指定的值当中
exists	
not exists	
like	模糊查询

条件查询语法格式

```
select
  ...
from
  ...
where
  过滤条件;
```

过滤条件放在where子句当中，以上语句的执行顺序是：
第一步：**先执行from**
第二步：**再通过where**条件过滤
第三步：最后执行select，查询并将结果展示到控制台

等于、不等于

等于 =

判断等量关系，支持多种数据类型，比如：数字、字符串、日期等。

案例1：查询月薪3000的员工编号及姓名

```
select
  empno,ename
from
  emp
where
  sal = 3000;
```

案例2：查询员工FORD的岗位及月薪

```
select
    job, sal
from
    emp
where
    ename = 'FORD';
```

存储在表emp中的员工姓名是FORD，全部大写，如果在查询的时候，写成全部小写会怎样呢?

```
select
    job, sal
from
    emp
where
    ename = 'ford';
```

> 注意：通过测试发现，即使写成小写ford，也是可以查询到结果的，**不过这里需要注意的是：在Oracle数据库当中是查询不到数据的，Oracle的语法要比MySQL的语法严谨。对于SQL语句本身来说是不区分大小写的，但是对于表中真实存储的数据，大写A和小写a还是不一样的，这一点Oracle做的很好。MySQL的语法更随性。另外在Oracle当中，字符串是必须使用单引号括起来的，但在MySQL当中，字符串可以使用单引号，也可以使用双引号**，如下：

```
select
    job, sal
from
  emp
where
  ename = "FORD";
```

案例3：查询岗位是MANAGER的员工编号及姓名

```
select
  empno, ename
from
  emp
where
  job = 'MANAGER';
```

不等于 <> 或 !=

判断非等量关系，支持字符串、数字、日期类型等。不等号有两种写法，第一种 <>（推荐这种写法），第二种 !=，第二种写法和Java程序中的不等号相同，第一种写法比较诡异，不过也很好理解，比如<>3，表示小于3、大于3，就是不等于3。

案例1：查询工资不是3000的员工编号、姓名、薪资

```
select
  empno,ename,sal
from
  emp
where
  sal <> 3000;
```

案例2：查询工作岗位不是MANAGER的员工姓名和岗位

```sql
select
  ename,job
from
    emp
where
    job <> 'MANAGER';
```

大于(等于)、小于(等于)

大于 >

案例：找出薪资大于3000的员工姓名、薪资

```sql
select
  ename, sal
from
  emp
where
  sal > 3000;
```

大于等于 >=

案例：找出薪资大于等于3000的员工姓名、薪资

```sql
select
  ename, sal
from
  emp
where
  sal >= 3000;
```

小于 <

案例：找出薪资小于3000的员工姓名、薪资

```sql
select
  ename, sal
from
  emp
where
  sal < 3000;
```

小于等于 <=

案例：找出薪资小于等于3000的员工姓名、薪资

```
select
  ename, sal
from
  emp
where
  sal <= 3000;
```

and

and表示并且，还有另一种写法：&&

案例：找出薪资大于等于3000并且小于等于5000的员工姓名、薪资。

```
select
  ename,sal
from
  emp
where
  sal >= 3000 and sal <= 5000;
```

or

or表示或者，还有另一种写法：||

案例：找出工作岗位是MANAGER和SALESMAN的员工姓名、工作岗位

```
select
  ename, job
from
  emp
where
  job = 'MANAGER' or job = 'SALESMAN';
```

> 注意：这个题目描述中有这样一句话：MANAGER和SALESMAN，有的同学一看到“和”，就直接使用“and”了，因为“和”对应的英文单词是“and”，如果是这样的话，就大错特错了，因为and表示并且，使用and表示工作岗位既是MANAGER又是SALESMAN的员工，这样的员工是不存在的，因为每一个员工只有一个岗

> 位，不可能同时从事两个岗位。

and和or的优先级问题

and和or同时出现时，**and优先级较高，会先执行**；如果希望or先执行，这个时候需要给or条件添加小括号。另外，以后**遇到不确定的优先级时，可以通过添加小括号**的方式来解决。对于优先级问题没必要记忆。

案例：找出薪资小于1500，并且部门编号是20或30的员工姓名、薪资、部门编号。
先来看一下错误写法：

```
select
  ename,sal,deptno
from
  emp
where
  sal < 1500 and deptno = 20 or deptno = 30;
```

认真解读题意得知：薪资小于1500是一个大前提，要找出的是薪资小于1500的，满足这个条件的前提下，再找部门编号是20或30的，显然以上的运行结果中出现了薪资为1600的，为什么1600的会出现呢？这是因为“sal < 1500 and deptno = 20”结合在一起了，“depnto = 30”成了一个独立的条件。会导致部门编号为30的所有员工全部查询出来。我们应该让“deptno = 20 or deptno = 30”结合在一起，正确写法如下：

```
select
  ename,sal,deptno
from
  emp
where
  sal < 1500 and (deptno = 20 or deptno = 30);
```

between...and...

between...and...等同于`>= and <=` ，做区间判断的，**闭区间，包含左右两个边界值。**它支持数字、日期、字符串等数据类型。

between...and...在使用时**一定是左小右大。**左大右小时无法查询到数据。

between...and... 和 >= and <= 只是在写法结构上有区别，执行原理和效率方面没有区别。

案例：找出薪资在1600到3000的员工姓名、薪资

```sql
select
  ename,sal
from
  emp
where
    sal between 1600 and 3000;
```

↑等价于：

```sql
select ename, sal from emp where sal >= 1600 and sal<=3000;
```

采用左大右小的方式：

```sql
select
  ename,sal
from
  emp
where
    sal between 3000 and 1600;
```

> 没有查询到任何数据，所以在使用的时候一定要注意：**左小右大。**

支持字符与日期格式：

```sql
select ename from emp where ename between 'a' and 'z';
select * from emp where hiredate between '1981-09-08' and
'1982-01-23';
```

is null、is not null

判断某个数据是否为null，不能使用等号，只能使用 is null
判断某个数据是否不为null，不能使用不等号，只能使用 is not null
在**数据库中null不是一个值，不能用等号和不等号衡量**，null代表什么也没有，没有数据，没有值

is null

案例1：找出津贴为空的员工姓名、薪资、津贴。

```sql
select
  ename,sal,comm
from
  emp
where
  comm is null;
```

is not null

案例2：找出津贴不为空的员工姓名、薪资、津贴

```sql
select
  ename,sal,comm
from
  emp
where
  comm is not null;
```

安全等于（仅作了解）

<=> 安全等于，用的很少，因为它的缺点是可读性差，了解即可。
<=> 安全等于可作为普通运算符=，安全等于除了可以达到等号=的效果之外，还可以使用`comm<=>null`代替`comm is null`。

使用`!(comm<=>null)`代替`comm is not null`

案例1：找出薪资3000的员工姓名、岗位

```sql
select
  ename,job
from
  emp
where
  sal <=> 3000;
```

in、not in (不是区间)

in (与or替换)

`job in('MANAGER','SALESMAN','CLERK')`等同于`job = 'MANAGER' or job = 'SALESMAN' or job = 'CLERK'`
`sal in(1600, 3000, 5000)`等同于`sal = 1600 or sal = 3000 or sal = 5000`

in 后面有一个小括号，小括号当中有多个值，值和值之间采用逗号隔开。

sal in(1500, 5000)，需要注意的是：这个并不是说薪资在1500到5000之间，**in不代表区间**，表示sal是1500的和sal是5000的。

案例1：找出工作岗位是MANAGER和SALESMAN的员工姓名、薪资、工作岗位
第一种：使用or

```
select
  ename,sal,job
from
  emp
where
  job = 'MANAGER' or job = 'SALESMAN';
```

第二种：使用in

```
select
  ename,sal,job
from
  emp
where
  job in('MANAGER', 'SALESMAN');
```

案例2：找出薪资是1500/1600/3000的员工姓名、工作岗位

```
select
  ename,job
from
  emp
where
  sal in(1500, 1600, 3000);
```

not in (与and替换)

`job not in('MANAGER','SALESMAN')` 等同于 `job <> 'MANAGER' and job <> 'SALESMAN'`
`sal not in(1600, 5000)` 等同于 `sal <> 1600 and sal <> 5000`

案例：找出工作岗位不是MANAGER和SALESMAN的员工姓名、工作岗位
第一种：使用and

```
select
  ename,job
from
  emp
where
  job <> 'MANAGER' and job <> 'SALESMAN';
```

第二种：使用not in

```
select
  ename,job
from
  emp
where
  job not in('MANAGER', 'SALESMAN');
```

in、not in 与 NULL

通过表中数据观察到，有4个员工的津贴不为NULL，剩下10个员工的津贴都是NULL。
写这样一条SQL语句：

```
select * from emp where comm in(NULL, 300);
```

以上执行结果只有一条记录：

首先要知道**in的执行原理实际上是采用=和or的方式**，也就是说，以上SQL语句实际上是：

```
select * from emp where comm = NULL or comm = 300;
```

其中NULL不能用等号=进行判断，所以`comm = NULL`结果是`false`，然而中间使用的是or，所以`comm = NULL`被忽略了。所以查询结果就以上一条数据。

通过以上的测试得知：**in是自动忽略NULL的（小括号里有null也不会做null的相等判断）。**

再写这样一条SQL语句：

```
select * from emp where comm not in(NULL, 300);
```

以上的执行结果没有查到任何数据：

首先要知道**not in的执行原理实际上是采用<>和and的方式**，也就是说，以上SQL语句实际上是：

```
select * from emp where comm <> NULL and comm <> 300;
```

其中NULL的判断不能使用<>，所以comm <> NULL结果是false，由于后面是and，and表示并且，comm <> NULL已经是false了，所以and右边的就没必要运算了，comm <> NULL and comm <> 300的整体运算结果就是false。所以查询不到任何数据。

通过以上测试得知，**not in是不会自动忽略NULL的**，所以在**使用not in的时候小括号里一定要提前去除掉NULL。**

in 和 or 的效率比拼

在MySQL当中，如何统计一个SQL语句的执行时长?

- 可以使用这个命令：`show profiles;` 这个命令可以查看在mysql中执行的所有SQL以及命令的耗费时长。
- show profiles; 是在mysql5.0.37之后添加的。
- 如何开启时长统计功能：set profiling = 1;
- 查看时长统计功能是否开启：`show variables like '%pro%';`
- 查看每条SQL的耗时：show profiles;
- 查看其中某条SQL耗时明细：show profile for query query_id;
- 查看最新一条SQL的耗时明细：show profile;
- 查看cpu，io等信息：`show profile block io, cpu for query query_id;`

or的效率为O(n)，而in的效率为O(log n), 当n越大的时候效率相差越明显（**也就是说数据量越大的时候，in的效率越高**）。以下是测试过程：
第一步，创建测试表，并生成测试数据，测试数据为1000万条记录。数据库中关闭了query cache，因此数据库缓存不会对查询造成影响。具体的代码如下：

```
#创建测试的test表
DROP TABLE IF EXISTS test;
CREATE TABLE test(
    ID INT(10) NOT NULL,
    `Name` VARCHAR(20) DEFAULT '' NOT NULL,
    PRIMARY KEY( ID )
)ENGINE=INNODB DEFAULT CHARSET utf8;

#创建生成测试数据的存储过程
DROP PROCEDURE IF EXISTS pre_test;
DELIMITER //
CREATE PROCEDURE pre_test()
BEGIN
DECLARE i INT DEFAULT 0;
SET autocommit = 0;
WHILE i<10000000 DO
INSERT INTO test ( ID,`Name` ) VALUES( i, CONCAT( 'Carl', i ) );
SET i = i+1;
IF i%2000 = 0 THEN
COMMIT;
END IF;
END WHILE;
END; //
DELIMITER ;

#执行存储过程生成测试数据
CALL pre_test();
```

以上SQL看不懂没关系，先执行它，进行数据初始化准备工作。
第二步：分三种情况进行测试，分别是：
第1种情况：in和or所在列为主键的情形。
第2种情况：in和or所在列创建有索引的情形。
第3种情况：in和or所在列没有索引的情形。
每种情况又采用不同的in和or的数量进行测试。由于测试语句的数据量有4种情况，这里就称为A组、B组、C组、D组，其中A组为3个值，B组为150个值，C组为300个值，D组为1000个值。具体的测试语句如下：

```
#A组
#in和or中有3条数据的情况
```

```
SELECT * FROM test WHERE id IN (1,23,48);
SELECT * FROM test WHERE id =1 OR id=23 OR id=48;

#B组   (省略大部分数据)
#in和or中有150条数据的情况
SELECT * FROM test WHERE id IN
(59617932,98114476,89047409,26968186,56586105,35488201,53251989,18182139,71164231,57655852,7948544,60658339,50758185,66667117,34771253,68699137,27877290,44275282,1585444,71219424,90937482,83928635,24588528,81933207,9607562,12013895,84640278);
SELECT * FROM test WHERE id=59617932 OR id=98114476 OR id=89047409 OR id=26968186 OR id=56586105 OR id=35488201 OR id=53251989 OR id=18182139 ;


#C组   (省略大部分数据)
#in和or中有300条数据的情况
SELECT * FROM test WHERE id IN
(37092877,94859722,74276090,8763830,38727241,95732954,93414819,55070016,3591352,73857925,92290525,15210159,83905516,54934589,83004136,31442143,6060569,22209206,27649629,11464943,77822402,28714780,10058522,62252663,13751461,38997875,47320577,64507359,47566746);

SELECT * FROM test WHERE id=37092877 OR id=94859722 OR id=74276090 OR id=8763830 OR id=38727241 OR id=95732954 OR id=93414819 OR id=55070016 OR id=3591352 OR id=73857925 OR id=92290525 OR id=15210159 OR id=83905516 OR id=54934589 OR id=83004136 OR id=31442143 OR id=47566746;


#D组  (省略大部分数据)
#in和or中有1000条数据的情况
SELECT * FROM test WHERE id IN
(93674701,9720356,31732184,53855095,33144472,71864888,27541768,27238726,83648428,12942332,26918445,19781953,81861032,74800064,12286132,6624397,64942581,70512799,46356598,88292448,87069909,38175756,98121997,62570414,15900806,51527968,89092372,80842203,53772848,78871524,3608561,85909562,41702172,61800503,57877634,93407278,30824340,13159046,49055339,73058078,983603,73571456,51694978,75136628,82716874,83551181,7964224,47505945,92695321,15885152,79282709,18572099,27392970,9463847);

```

```
SELECT * FROM test WHERE id=93674701 OR id=9720356 OR
id=31732184 OR id=53855095 OR id=33144472 OR id=71864888 OR
id=27541768 OR id=27238726 OR id=83648428 OR id=12942332 OR
id=26918445 OR id=19781953 OR id=81861032 OR id=74800064 OR
id=12286132 OR id=6624397 OR id=64942581 OR id=70512799 OR
id=46356598 OR id=88292448 OR id=87069909 OR id=38175756 OR
id=98121997 OR id=62570414 OR id=15900806 OR id=51527968 OR
id=89092372 OR id=8084203 OR id=53772848 OR id=78871524 OR
id=3608561 OR id=85909562 OR id=41702172 OR id=61800503 OR
id=57877634 OR id=93407278 OR id=30824340 OR id=9463847;
```

测试结果如下：
第一种情况，ID列为主键的情况，4组测试执行计划一样，执行的时间也基本没有区别。

第二种情况，ID列为一般索引的情况，4组测试执行计划一样，执行的时间也基本没有区别。

第三种情况，ID列没有索引的情况，4组测试执行计划一样，执行的时间有很大的区别。
A组or和in的执行时间： or的执行时间为：5.016s　　in的执行时间为：5.071s
B组or和in的执行时间： or的执行时间为：1min 02s　　in的执行时间为：5.018s
C组or和in的执行时间： or的执行时间为：1min 55s　　in的执行时间为：5.018s
D组or和in的执行时间： or的执行时间为：6min 17s　　in的执行时间为：5.057s

结论：从上面的测试结果，可以看出**如果in和or所在列有索引或者主键的话，or和in没啥差别**，执行计划和执行时间都几乎一样。**如果in和or所在列没有索引的话，性能差别就很大了。**在**没有索引的情况下，随着in或者or后面的数据量越多，in的效率不会有太大的下降，但是or会随着记录越多的话性能下降非常厉害**，从第三种测试情况中可以很明显地看出了，基本上是指数级增长。

因此在给in和or的效率下定义的时候，应该再加上一个条件，就是所在的列是否有索引或者是否是主键。如果有索引或者主键性能没啥差别，如果没有索引，性能差别不是一点点，in 的效率会远高于 or！

模糊查询like

模糊查询又被称为模糊匹配，在实际开发中使用较多，比如：查询公司中所有姓张的，查询岗位中带有经理两个字的职位等，这些都需要使用模糊查询。

模糊查询的语法格式如下：

```
select .. from .. where 字段 like '通配符表达式';
```

在模糊查询中，通配符主要包括两个：一个是`%`，一个是下划线`_`。其中`%`**代表任意多个 (0到多个)**字符。下划线`_`**代表任意一个**字符。

案例1：查询员工名字以'S'开始的员工姓名

```
select ename from emp where ename like 'S%';
```

案例2：查询员工名字以'T'结尾的员工姓名

```
select ename from emp where ename like '%T';
```

案例3：查询员工名字中含有'O'的员工姓名

```
select ename from emp where ename like '%O%';
```

案例4：查询员工名字中第二个字母是'A'的员工姓名

```
select ename from emp where ename like '_A%';
```

案例5：查询学员名字中含有下划线的。
执行以下SQL语句，先准备测试数据：

```
drop table if exists student;
create table student(
  id int,
  name varchar(255)
);
insert into student(id,name) values(1, 'susan');
insert into student(id,name) values(2, 'lucy');
insert into student(id,name) values(3, 'jack_son');
select * from student;
```

查询学员名字中**含有下划线**的，执行以下SQL试试：

```
select * from student where name like '%_%';
```

以上SQL将所有数据全部显示了，显然这个查询结果不是我们想要的；因为下划线代表任意单个字符，如果你想让这个下划线变成一个普通的下划线字符，就要**使用转义字符**了，在**mysql当中转义字符是**`\`，这个和Java语言中的转义字符是一样的：

```
select * from student where name like '%\_%';
```

排序操作

排序操作很常用，比如查询学员成绩，按照成绩降序排列。

排序的SQL语法：

```
select .. from .. order by 字段 asc/desc    # 不写asc或desc，则默认采用升序(asc)
```

单一字段升序

查询员工的编号、姓名、薪资，按照薪资升序排列。

```
select empno,ename,sal from emp order by sal asc;
```

单一字段降序

查询员工的编号、姓名、薪资，按照薪资降序排列。

```
select empno,ename,sal from emp order by sal desc;
```

默认采用升序

查询员工的编号、姓名、薪资，按照薪资升序排列。

```
select empno,ename,sal from emp order by sal;
```

多个字段排序

查询员工的编号、姓名、薪资，按照薪资升序排列，如果薪资相同的，再按照姓名升序排列。

```
select empno,ename,sal from emp order by sal asc, ename asc;
```

> 如果按照某个字段排序的结果出现相同的值，那么这些值的顺序通常是不确定的。它们的顺序取决于数据库内部的实现和数据的物理存储方式；除非又指定相同后按其他字段排序。

可用不建议的写法： `order by` 后用列的编号（编号从1开始）

```
select empno,ename,sal from emp order by 3 asc, 2 asc;  # 因为在查询中，sal是第3列，ename是第2列
```

where和order by的位置

找出岗位是MANAGER的员工姓名和薪资，按照薪资升序排列。

```
select ename,sal from emp where job = 'MANAGER' order by sal asc;
```

> 通过这个例子想告诉大家：**from先执行，where之后执行，再是select，最后是order by语句。**

```
select ename,job,sal*12 as yearsal from emp where job in('MANAGER','SALESMAN') order by yearsal desc;
```

distinct去重

查询工作岗位

```
select job from emp;
```

可以看到工作岗位中有重复的记录，如何在显示的时候去除重复记录呢?

在字段前添加`distinct`关键字。

```
select distinct job from emp;
```

注意：这个去重**只是将显示的结果去重，原表数据不会被更改。**

接下来测试一下，在distinct关键字前添加其它字段是否可以?

```
select ename, distinct job from emp;
#ERROR 1064(42000): You have an error in your SQL syntax;
```

> ename是14条记录，distinct job是5条记录，不能同时显示。

报错了，通过测试得知:

distinct只能出现在所有字段的最前面。当distinct出现后，后面多个字段一定是联合去重的 (多个字段一块查出的结果再去重)。

找出公司中不同部门的不同工作岗位。

```
select distinct deptno, job from emp;
```

数据处理函数

关于select语句，`select 字段名 from 表名; 这里的字段名可以看做“变量”，select后面既然可以跟变量，那么也可以跟常量：

```
select 100;   # 常量
select "zhangsan";  # 常量 (字符串字面量)
# 查询结果都是其本身
```

```
select ename;   # 这里ename是变量，但会报错，因为找不到该字段
# ERROR 1054(42S22):Unknown column 'ename' in 'field list'
```

通过以上测试得知，**select后面既可以跟变量，又可以跟常量。**

最后一条SQL的ename没有添加单引号，它会被当做某个表的字段名，因为没有这个字段所以报错。

字符串相关

转大写upper和ucase

```
# 查询所有员工名字，以大写形式展现
select upper(ename) as ename from emp;
```

还有一个和upper函数功能相同的函数ucase，也可以转大写，了解一下即可：

```
# 查询所有员工姓名，以大写形式展现
select ucase(ename) as ename from emp;
```

用途示例：查询员工Smith的岗位、薪资（假如你不知道数据库表中的人名是大写、小写还是大小写混合）

```
select ename, job, sal from emp where upper(ename) = 'SMITH';
```

转小写lower和lcase

```
# 查询员工姓名，以小写形式展现
select lower(ename) as ename from emp;
select lcase(ename) as ename from emp;
```

截取字符串substr

语法：`substr('被截取的字符串', 起始下标, 截取长度)`

有两种写法：

- 第一种：`substr('被截取的字符串', 起始下标, 截取长度)`
- 第二种：`substr('被截取的字符串', 起始下标)` ，当第三个参数“截取长度”缺失时，截取到字符串末尾

注意：**起始下标从1开始**，不是从0开始。（**1表示从左侧开始的第一个**位置，**-1表示从右侧开始的第一个**位置）

练习：找出员工名字中第二个字母是A的

```
select ename from emp where substr(ename, 2, 1) = 'A';
```

```
select substr('abcdef', -1, 1);   # 结果是 f
select substr('abcdef', -1, 2);   # 结果还是 f    【截的时候永远往右截】
select substr('abcdef', -2, 2);   # 结果是 ef
```

获取字符串长度length

length() 统计的是字节的长度。

```
select length('你好123');  # 结果是 7
```

> 注意：一个汉字是2个字节长度。

获取字符的个数char_length

```
select char_length('你好123');  # 结果是 5
```

字符串拼接

语法：`concat('字符串1', '字符串2', '字符串3'....)`

拼接的字符串数量没有限制。

```
select concat('abc', 'def', 'xyz') as str;
```

注意：在MySQL 8 之前，双竖线 `||` 也是可以完成字符串拼接的。但在MySQL 8 之后，`||` 只作为逻辑运算符，不能再进行字符串拼接

```
select 'abc' || 'def' || 'xyz';    # MySQL 8 之前可用
```

MySQL 8 之后，`||` 只作为"或者"运算符。

例如：找出工资高于3000或者低于900的员工姓名和薪资：

```
select ename, sal from emp where sal > 3000 || sal < 900;
```

MySQL中可以使用 `+` 进行字符串的拼接吗？不可以。**在MySQL 中 `+` 只作加法运算**，在进行加法运算时，会将加号两边的数据尽最大的努力转换成数字再求和，**如果无法转换成数字，最终运算结果通通是0。**

```
select 'abc' + 'def';  # 结果是 0
```

去除字符串前后空白trim

```
select concat('test', trim('    a  b  c    '), 'def');   # 结果是 testa  b  cdef
```

默认是去除前后空白 (中间空白不管)，也可以去除指定的前缀后缀，例如：

去除前置0

```
select trim(leading '0' from '000111000');
```

去除后置0

```
select trim(trailing '0' from '000111000');
```

前置0和后置0全部去除

```
select trim(both '0' from '000111000');
```

数字相关

rand()和rand(x)

rand()生成**0到1的随机浮点数。**

```
select rand();  # 0.18175401369273106
```

rand(x)生成0到1的随机浮点数，通过指定整数x来确定每次获取到相同的浮点值。

```
select rand(7);
```

> 里面的 x 作为 seed，不同电脑MySQL中的 rand(7) 的结果都一样，是 0.9065021936842261

round(x)和round(x,y)四舍五入

round(x) 四舍五入，保留整数位，舍去所有小数

```
select round(9.754);   # 结果是 10
select round(9.454);   # 结果是 9
```

round(x,y) 四舍五入，保留y位小数

```
select round(9.454, 1);  # 结果是 9.5
select round(9.454, 0);  # 结果是 9
select round(14.3, -1);  # 结果是 10   [相当于保留十位，对个位四舍五入]
```

truncate(x, y)舍去

truncate(x, y) 保留 y 位小数，其他位直接舍去，不四舍五入。

```
select truncate(9.999, 2);  # 结果是 9.99
```

> 以上SQL表示保留两位小数，剩下的全部舍去。

ceil与floor

数字处理函数除了以上的之外，还有ceil和floor函数：

- ceil函数：返回大于或等于数值x的最小整数 【向上取整】
- floor函数：返回小于或等于数值x的最大整数 【向下取整】

```
select ceil(5.3);   # 6
select floor(5.7821);  # 5
```

空处理

`ifnull(x, y)`，空处理函数，表示当 x 为`NULL`时，将 x 当做 y 处理。

ifnull(comm, 0)，表示如果员工的津贴是NULL时当做0处理。

在SQL语句中，凡是有NULL参与的数学运算，最终的计算结果都是NULL：

> 这一规则在多种数据库系统中都是通用的，包括但不限于 MySQL、PostgreSQL、SQL Server、Oracle 和 SQLite 等

```
select null + 1;  # 结果是 NULL
select null*10;   # NULL
```

需求：查询每个员工的年薪。 (年薪 = (月薪 + 津贴) * 12个月。注意：有的员工津贴comm是NULL。)

```
select ename, (sal + ifnull(comm, 0)) * 12 as yearsal from
emp;
```

日期和时间相关函数

获取当前日期和时间

now()和sysdate()的区别：

- now()：获取的是**执行select语句**的时刻。
- sysdate()：获取的是**执行sysdate()函数的时刻。**

```
select now();    # 2024-09-16 21:25:39
select sysdate();
```

```
# 测试时刻不一样：
select now(), sleep(2), sysdate();   #  2024-09-16 21:25:39
0  2024-09-16 21:25:41
```

获取当前日期

获取当前日期有三种写法，掌握任意一种即可：

- curdate()
- current_date()
- current_date

```
select curdate();       #  2024-09-16
select current_date();
select current_date;
```

获取当前时间

获取当前时间有三种写法，掌握其中一种即可：

- curtime()
- current_time()
- current_time

```
select curtime();       #   21:31:06
select current_time();
select current_time;
```

获取单独的年、月、日、时、分、秒

```
select year(now());    # 2024    [year()参数里要传一个日期类型的数据]
select month(now());   # 9
select day(now());     # 16
select hour(now());    # 21
select minute(now());  # 34
select second(now());  # 27
```

注意：这些函数在使用的时候，需要传递一个日期参数给它，它可以获取到你给定的这个日期相关的年、月、日、时、分、秒的信息。

一次性提取一个给定日期的“年月日”部分，可以使用date()函数，例如：

```
select date(now());    # 2024-09-16
```

一次性提取一个给定日期的“时分秒”部分，可以使用time()函数，例如：

```
select time(now());    # 21:36:09
```

date_add函数

date_add函数的作用：给指定的日期添加间隔的时间，从而得到一个新的日期。

date_add函数的语法格式：`date_add(日期, interval expr 单位)`

详细解释一下这个函数的相关参数：

- 日期：一个日期类型的数据
- interval：关键字，翻译为“间隔”，固定写法
- expr：指定具体的间隔量，一般是一个数字。**也可以为负数，如果为负数，效果和date_sub函数相同。**
- 单位：
  - year：年
  - month：月
  - day：日
  - hour：时
  - minute：分

- second：秒
- microsecond：微秒（1秒等于1000毫秒，1毫秒等于1000微秒）
- week：周
- quarter：季度（四分之一年，即三个月）

例如：

```
select date_add('2023-01-03', interval 3 day);    # 2023-01-06
```

> 以'2023-01-03'为基准，间隔3天之后的日期：'2023-01-06'

```
select date_add('2023-01-03', interval 3 month);   # 2023-04-
03
```

> 以'2023-01-03'为基准，间隔3个月之后的日期：'2023-04-03'

```
select date_add('2010-10-11 10:10:10', interval 1 hour);
```

date_sub()

```
select date_sub('2010-04-01', interval 3 month);  # 2010-01-01
select date_add('2022-10-01 10:10:10', interval -1
microsecond);  # 2022-10-01 10:10:09.999999
```

另外，单位也可以采用复合型单位，例如：

- SECOND_MICROSECOND
- MINUTE_MICROSECOND
- MINUTE_SECOND：几分几秒之后
- HOUR_MICROSECOND
- HOUR_SECOND
- HOUR_MINUTE：几小时几分之后
- DAY_MICROSECOND
- DAY_SECOND
- DAY_MINUTE
- DAY_HOUR：几天几小时之后
- YEAR_MONTH：几年几个月之后

```
select date_add('2022-10-01 10:10:10', interval '3,2' day_hour);  # 2022-10-04 12:10:10
```

> '3,2' 表示3天 加 2个小时 之后。 '3,2' 和 day_hour 是对应的。

date_format日期格式化函数

将日期转换成具有某种格式的日期字符串，通常用在查询操作当中。（**date类型转换成char类型**）

语法格式：`date_format(日期, '日期格式')`

该函数有两个参数：

- 第一个参数：日期。这个参数就是即将要被格式化的日期。类型是date类型。
- 第二个参数：指定要格式化的格式字符串。
  - %Y：四位年份
  - %y：两位年份
  - %m：月份（1..12）
  - %d：日（1..30）
  - %H：小时（0..23）
  - %i：分（0..59）
  - %s：秒（0..59）

例如：获取当前系统时间，让其以这个格式展示：2000-10-11 20:15:30

```
select date_format(now(), '%Y-%m-%d %H:%i:%s');    # 2024-09-16 22:20:20
```

注意：在mysql当中，默认的日期格式就是：`%Y-%m-%d %H:%i:%s`，所以当你直接输出日期数据的时候，会自动转换成该格式的字符串：

```
select now();   # 2024-09-16 22:20:20      [这里是个隐式转换，底层还是将获取到的Date类型转成字符串显示]
```

```
select ename, sal, date_format(hiredate, '%y/%m/%d') as hiredate from emp;
```

str_to_date函数

该函数的作用是将char类型的日期字符串转换成日期类型date，通常使用在插入和修改操作当中。（**char类型转换成date类型**）

假设有一个学生表t_student，学生有一个生日的字段，类型是date类型：

```
drop table if exists t_student;
create table t_student(
  name varchar(255),
  birth date
);
desc t_student;
```

我们要给这个表插入一条数据：姓名zhangsan，生日85年10月1日，执行以下insert语句：

```
insert into t_student(name,birth) values('zhangsan',
'10/01/1985');
# ERROR 1292(22007): Incorrect date value:
```

错误原因：日期值不正确。意思是：birth字段需要一个日期，给的字符串'10/01/1985' 识别不了。这种情况下，我们就可以使用str_to_date函数进行类型转换：

```
insert into t_student(name,birth) values('zhangsan',
str_to_date('10/01/1985', '%m/%d/%Y'));
```

当然，如果你提供的日期字符串格式能够被mysql解析，str_to_date函数是可以省略的，底层会自动调用该函数进行类型转换：

```
insert into t_student(name,birth) values('zhangsan', '1985-10-
01');
insert into t_student(name,birth) values('zhangsan', '85-10-
01');
insert into t_student(name,birth) values('zhangsan',
'85/10/01');
insert into t_student(name,birth) values('zhangsan',
'1985/10/01');
```

> 如果日期格式符合以上的几种格式，mysql都会**自动进行类型转换**的。

dayofweek、dayofmonth、dayofyear函数

- dayofweek：一周中的第几天（1~7），**周日是1**，周一是2，...，周六是7。
- dayofmonth：一个月中的第几天（1~31）
- dayofyear：一年中的第几天（1~366）

```
select dayofweek('2024-09-15');  # 1  （周日是1）
select dayofmonth(now());
select dayofyear(now());
```

last_day函数

获取给定**日期所在月的最后一天的日期**：

```
select last_day('2024-09-16');   # 2024-09-30
```

datediff函数

计算两个日期之间所差天数：

```
select datediff('1970-02-01 20:10:30', '1970-01-01');  # 31
```

> 时分秒不算，只计算日期部分相差的天数。

timediff函数

计算两个日期所差时间，例如日期1和日期2所差10:20:30，表示差10小时20分钟30秒。

```
select timediff('1970-01-02 20:10:30', '1970-01-01 20:09:30');
 # 24:01:00
```

if函数

```
# 如果条件为TRUE则返回“YES”，如果条件为FALSE则返回“NO”：
SELECT IF(500<1000, "YES", "NO");
```

例如：如果工资高于3000，则输出1，反之则输出0

```
select ename, if(sal > 3000, 1, 0) from emp;
```

例如：如果名字是SMITH的，工资上调10%，其他员工工资正常显示。

```
select ename,if(ename='SMITH', sal * 1.1, sal) as sal from
emp;
```

例如：工作岗位是MANAGER的工资上调10%，是SALESMAN的工资上调20%，其他岗位工资正常。

```
select ename, job, if(job='MANAGER', sal*1.1,
if(job='SALESMAN', sal*1.2, sal)) as newsal from emp;
```

上面这个需求也可以使用：`case.. when.. then.. when.. then.. else.. end`来完成：

```
select ename, job,
case job
when 'MANAGER' then sal*1.1
when 'SALESMAN' then sal*1.2
else sal
end
as sal
from emp;
```

cast函数

cast函数用于将**值从一种数据类型转换为表达式中指定的另一种数据类型**

语法：`cast(值 as 数据类型)`

例如：cast('2020-10-11' as date)，表示将字符串'2020-10-11'转换成日期date类型。

在使用cast函数时，可用的数据类型包括：

- date：日期类型
- time：时间类型
- datetime：日期时间类型
- signed：有符号的int类型（有符号指的是正数负数）
- char：定长字符串类型
- decimal(x, y)：浮点型 [x表示精度，是有效数字中允许的总位数（包括小数点两侧的位数）；y表示标度，是小数点后的位数（四舍五入）]

```
select cast('2020-10-11 20:15:30' as date);   #  2020-10-11
select cast('2020-10-11 20:15:30' as time);   #  20:15:30
select cast('2020-10-11 20:15:30' as datetime);  # 2020-10-11 20:15:30
select cast('-5.3'as signed);    # -5
select cast('5.3'as signed);   # 5
select cast(123.456 as char(3));  # 123
select cast(123.456 as char(4));  # 123.
select cast('000123.456' as decimal(5,1));   # 123.5   [1位小数(四舍五入)，总共最多允许5位，现在有4位也满足]
```

加密函数

md5函数，可以将给定的字符串经过md5算法进行加密处理，字符串经过加密之后会生成一个固定长度128bit位的字符串，md5加密之后的密文是不可逆的：（但通过碰撞攻击等也有可能破解）

```
select md5('powernode');   # 88cdef88bcdb8b2e33fb7b4f076cf803  [以十六进制显示]
```

分组函数

> 分组函数又可叫“多行处理函数”： 多个输入对应一个输出。

分组函数的执行原则：**先分组，然后对每一组数据执行分组函数。**如果**没有分组语句group by的话，整张表的数据自成一组。**

分组函数包括五个：

- max：最大值
- min：最小值

- avg：平均值
- sum：求和
- count：计数

max

找出员工的最高薪资

```
select max(sal) from emp;
```

min

找出员工的最低工资

```
select min(sal) from emp;
```

avg

计算员工的平均薪资

```
select avg(sal) from emp;
```

sum

计算员工的工资和

```
select sum(sal) from emp;
```

计算员工的津贴之和

```
select sum(comm) from emp;
```

重点：所有的**分组函数都是自动忽略NULL**的。

count

统计员工人数

```
select count(ename) from emp;    # 14
select count(*) from emp;    #  14
select count(1) from emp;    #  14
```

> `count(*)` 和 `count(1)` 的效果一样，统计该组中总记录行数。

- `count(*)`：　　统计这张表中一共有多少行记录。
- `count(某个字段)`：统计这个字段中不为NULL的个数。

count(ename)统计的是这个ename字段中不为NULL个数的总和。

例如：count(comm) 结果是 4，而不是14

```
select count(comm) from emp;   # 4
```

例如：统计岗位数量

```
select count(distinct job) from emp;   # 5
```

分组函数组合使用

```
select count(*), max(sal), min(sal), avg(sal), sum(sal) from
emp;
```

分组函数注意事项

分组函数不能直接使用在where子句当中

```
select ename,job from emp where sal > avg(sal);
# ERROR 1111 (HYooo): Invalid use of group function

```

报错原因：分组函数的执行顺序在where执行之后，所以不能正常运行。

分组查询

group by

按照某个字段分组，或者按照某些字段联合分组。

注意：**group by的执行是在where之后执行。**

语法：

- `group by 字段`
- `group by 字段1,字段2,字段3....`

找出每个岗位的平均薪资

```
select job, avg(sal) from emp group by job;  # avg()是对分好的每一组进行计算
```

找出每个部门最高工资

```
select deptno, max(sal) from emp group by deptno;
```

找出每个部门不同岗位的平均薪资

```
select deptno, job, avg(sal) from emp group by deptno,job;
```

注意：**当select语句中有group by的话，select后面只能跟分组函数或参加分组的字段。**（其他的不允许写）

```
select ename, deptno, avg(sal) from emp group by deptno;  # 这个SQL执行后会报错
        # ename有14条数据 , avg(sal) 有 3条数据，对不上
# 可以写成 ↓
select deptno, avg(sal) from emp group by deptno;
select avg(sal) from emp group by deptno;
```

having

having写在group by的后面，当你对分组之后的数据不满意，可以继续**通过having对分组之后的数据进行过滤。** （having语句必须在出现有group by语句后才能写）

where的过滤是在分组前进行过滤。

使用原则：**尽量在where中过滤**，实在不行，再使用having。**越早过滤效率越高。**

找出除20部分之外，其它部门的平均薪资。

```
select deptno,avg(sal) from emp where deptno<>20 group by deptno;    # 建议
select deptno,avg(sal) from emp group by deptno having deptno <> 20; # 不建议
```

查询每个部门平均薪资，找出平均薪资高于2000的。

```
select deptno,avg(sal) from emp group by deptno having avg(sal) > 2000;
```

组内排序

substring_index函数的使用： 截取到第y次出现 x 的位置的子串

```
select substring_index('http://www.baidu.com' ,'.',1);  # http://www
select substring_index('http://www.baidu.com' ,'.',2);  # http://www.baidu
```

group_concat函数的使用： 分组拼接 （自动用 `,` 连接）

```
select group_concat(empno order by sal desc) from emp group by job;
/*
  group_concat (empno order by sal desc)
----------------------------------------
7902, 7788
7934, 7876, 7900, 7369
7566, 7698, 7782
7839
7499, 7844, 7654,7521
*/
```

练习：找出每个工作岗位的工资排名在前两名的。

```
select substring_index(group_concat(empno order by sal desc),
',', 2) from emp group by job;
```

单表的DQL语句执行顺序

```
select ...       # 5
from ...    # 1
where ...    # 2
group by ...  # 3
having ...     # 4
order by ...       # 6
```

连接查询

什么是连接查询?

1. 从一张表中查询数据称为单表查询。
2. 从**两张或更多张表中联合查询数据**称为**多表查询，又叫做连接查询。**
3. 什么时候需要使用连接查询?

   比如这样的需求：员工表中有员工姓名，部门表中有部门名字，要求查询每个员工所在的部门名字，这个时候就需要连接查询。

连接查询的分类

1. 根据语法出现的年代进行分类：
    1. SQL92（这种语法很少用，可以不用学。）
    2. SQL99（我们主要学习这种语法。）
2. 根据连接方式的不同进行分类：
    1. 内连接
        1. 等值连接
        2. 非等值连接
        3. 自连接
    2. 外连接

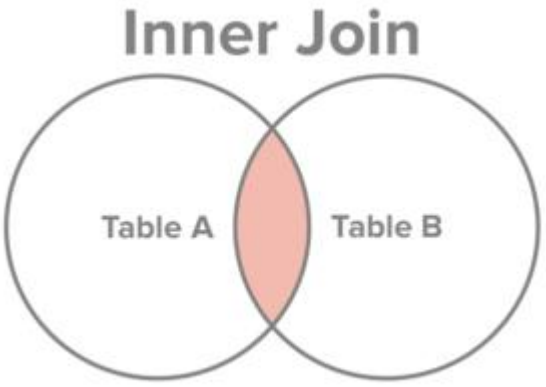
    1. 左外连接（左连接）
    2. 右外连接（右连接）
3. 全连接　[MySQL 8 不支持全连接，但可以模拟全连接]

笛卡尔积现象

1. 笛卡尔积现象：当**两张表进行连接查询时，如果没有任何条件进行过滤，最终的查询结果条数是两张表条数的乘积。**为了避免笛卡尔积现象的发生，需要添加条件进行筛选过滤。
2. 需要注意：**添加条件之后，虽然避免了笛卡尔积现象，但是匹配的次数没有减少。**
3. 为了SQL语句的可读性，**为了执行效率，建议给表起别名。**

内连接

什么叫内连接？ 交集



两个表的交集部分，满足条件的记录才会出现在结果集中。

等值连接

连接时，条件为等量关系。

案例：查询每个员工所在的部门名称，要求显示员工名、部门名。

```sql
select
    e.ename, d.dname     # 表别名.字段名
from
    emp e         # e 是表emp的别名
inner join
    dept d        # d 是表dept的别名
on
    e.deptno = d.deptno;     # 两张表连接的等值条件
```

注意：inner可以省略。

```sql
SELECT
    e.ename,d.dname
FROM
    emp e
JOIN
    dept d
ON
    e.deptno = d.deptno
WHERE
    sal > 2000;
```

非等值连接

连接时，条件是非等量关系。

案例：查询每个员工的工资等级，要求显示员工名、工资、工资等级。

```sql
select
    e.ename, e.sal, s.grade
from
    emp e
join
    salgrade s
on
    e.sal between s.losal and s.hisal;   # 每一行记录内 sal 与 当前行的 losal、hisal 作比较
```

自连接

连接时，**一张表看做两张表，自己和自己进行连接。**

案例：找出每个员工的直属领导，要求显示员工名、领导名。

```
select
    e.ename 员工名, l.ename 领导名
from
    emp e
join
    emp l    # 同一张表取不同的别名
on
    e.mgr = l.empno;
```

> 把等量关系思考清楚：连接条件是：e.mgr = l.empno（员工的领导编号 = 领导的员工编号）
>
> 员工KING因为其mgr是NULL，所以不会展现在结果里。

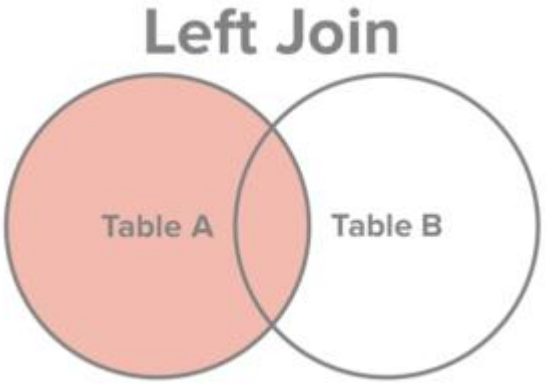
外连接

什么叫外连接?

内连接是满足条件的记录查询出来。也就是两张表的交集。

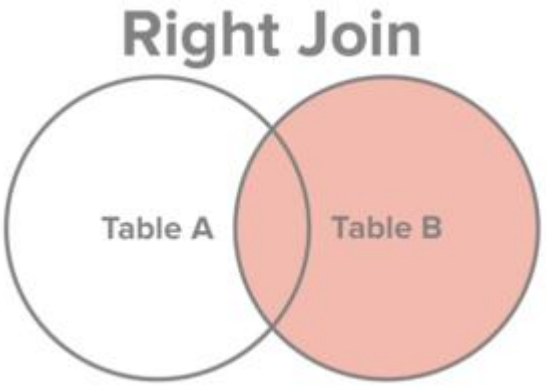
外连接是除了满足条件的记录查询出来，再将其中一张表的记录全部查询出来，另一张表如果没有与之匹配的记录，自动模拟出NULL与其匹配。

左外连接：



- 把左边表全部展现的，就是左外连接；
- 把右边表全部展现的，就是右外连接；

右外连接：



左外连接（左连接）

案例：查询所有部门信息，并且找出每个部门下的员工。

```
select
  d.*, e.ename    # d.* 是展示d表全部字段
from
  dept d    # 左
left outer join
  emp e     # 右
on
  d.deptno = e.deptno;

/*
    DEPTNO    DNAME          LOC      ename
-------------------------------------------
    10        ACCOUNTING     NEW YORK    MILLER
    10        ACCOUNTING     NEW YORK    KING
    10        ACCOUNTING     NEW YORK    CLARK
    20        RESEARCH       DALLAS      FORD
    20        RESEARCH       DALLAS      ADAMS
    30        SALES          CHICAGO     JAMES
    30        SALES          CHICAGO     TURNER
    40        OPERATIONS     BOSTON      NULL        （右表没对应的，右表里内容就是NULL）
*/
```

注意：outer 可以省略。 （区分内外连接的关键字是有没有 `left / right`）

任何一个左连接都可以写作右连接，反之右连接也可写成左连接。

右外连接（右连接）

还是上面的案例，可以写作右连接。

```
select
  d.*, e.ename
from
  emp e       # 左
right outer join
  dept d      # 右 (右表内容全部保留，如果左表没对应的就是NULL)
on
  d.deptno = e.deptno;
```

案例：找出所有员工的上级领导，要求显示员工名和领导名。

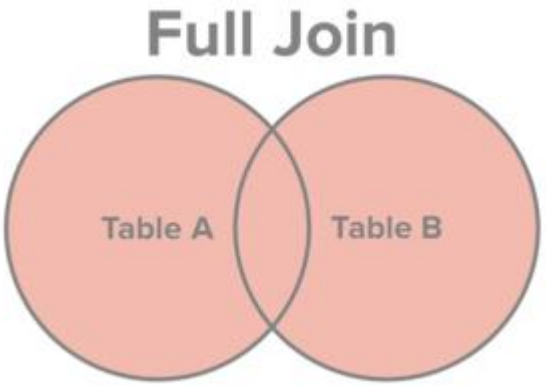
```
select
  e.ename 员工名, l.ename 领导名
from
  emp e
left join
  emp l
on
  e.mgr = l.empno;
```

```
select
  e.ename 员工名,l.ename 领导名
from
  emp l
right join
  emp e
on
  e.mgr = l.empno;
```

全连接

什么是全连接?

MySQL不支持`full join`语法 （但可用其他语法模拟全连接的效果）。Oracle数据库支持。



把两张表数据全部查询出来，如果没有匹配的记录，各自为对方模拟出NULL进行匹配。

例如：客户表：t_customer

```
  cid   cname
-----------------
   1   zhangsan
   2   lisi
   3   wangwu
```

订单表：t_order

```
   oid    price   cid
 ----------------------
   100     3400    1
   200     6600    2
   300     9900    NULL
```

案例：查询所有的客户和订单。

```
select
   c.*, o.*
from
   t_customer c
full join
   t_order o
on
   c.cid = o.cid;
```

多张表连接

三张表甚至更多张表如何进行表连接？ 接着 `join..on..join..on..` 的语法。

案例：找出每个员工的部门，并且要求显示每个员工的薪资等级。

```
select
   e.ename,d.dname,s.grade
from
   emp e
join
   dept d
on
   e.deptno = d.deptno   # 找出每个员工的部门
join
   salgrade s
on
   e.sal between s.losal and s.hisal;  # 每个员工的薪资等级
```

子查询

什么是子查询?

1. **select语句中嵌套select语句**就叫做子查询。
2. select语句可以嵌套在哪里?
    - where后面、from后面、select后面都是可以的。

```
select ..(select)..
from ..(select)..
where ..(select)..
```

where后面使用子查询

案例：找出高于平均薪资的员工姓名和薪资。

错误的示范：

```
select ename,sal from emp where sal > avg(sal);
```

错误原因：where后面不能直接使用分组函数。

可以使用子查询：

```
select ename,sal from emp where sal > (select avg(sal) from emp);    # 子查询用括号括起来 (带小括号的子查询是先执行的)
```

from后面使用子查询

小窍门：**from 后面的子查询可以看做一张临时表。**

案例：找出每个部门的平均工资的等级。

第一步：先找出每个部门平均工资。

```
select deptno, avg(sal) avgsal from emp group by deptno;
```

第二步：将以上查询结果当做临时表 t，t 表和salgrade 表进行连接查询。条件：`t.avgsal between s.losal and s.hisal`

```
select t.*, s.grade from (select deptno, avg(sal) avgsal from emp group by deptno) t join salgrade s on t.avgsal between s.losal and s.hisal;
```

select后面使用子查询

案例：查询每个员工名及所在的部门名。

```
select e.ename, (select d.dname from dept d where e.deptno = d.deptno) as dname from emp e;
```

exists、not exists

在 MySQL 数据库中，**EXISTS 用于检查子查询的查询结果行数是否大于0。如果子查询的查询结果行数大于0，则 EXISTS 条件为真。**（即存在查询结果则是true。）

主要应用场景：

- EXISTS 可以与 SELECT、UPDATE、DELETE 一起使用，用于检查另一个查询是否返回任何行；

- EXISTS 可以用于验证条件子句中的表达式是否存在；
- EXISTS **常用于子查询条件过滤**，例如查询有订单的用户等。

```
# 测试数据
drop table if exists t_customer;
drop table if exists t_order;

create table t_customer(
  customer_id int,
  customer_name varchar(32)
);

create table t_order(
  order_id int,
  order_price decimal(5,1),
  customer_id int
);

insert into t_customer(customer_id,customer_name)
values(1,'zhangsan');
insert into t_customer(customer_id,customer_name)
values(2,'lisi');
insert into t_customer(customer_id,customer_name)
values(3,'wangwu');

insert into t_order(order_id, order_price, customer_id)
values(10, 1000.0, 1);
insert into t_order(order_id, order_price, customer_id)
values(20, 2000.0, 1);
insert into t_order(order_id, order_price, customer_id)
values(30, 3000.0, 2);
insert into t_order(order_id, order_price, customer_id)
values(40, 4000.0, 2);

commit;
select * from t_customer;
select * from t_order;
```

现在来看一个简单的案例：假设要查询先前有过订单的顾客，而订单信息保存在 t_order 表中，顾客信息保存在 t_customer 表中。可以使用以下 sql 语句：

```
select * from t_customer c where exists(select * from t_order
o where o.customer_id = c.customer_id);
```

在这个查询语句中，子查询用于检查是否有订单与每个客户相关联。如果子查询返回至少一行，则表示该顾客已经下过订单，并返回此客户的所有信息，否则该顾客将不被包含在结果中。

以下是这个查询语句的执行过程：

1. 首先查询表 t_customer 中的所有顾客信息（以下简称为顾客表）；
2. 对于顾客表中的**每一行，都执行一次子查询**，**子查询查询该顾客有没有订单**，如果有，则在结果集中保留该顾客信息；如果没有，则将该顾客排除；
3. 最终返回有订单顾客的所有信息。

```
# 用 IN 的写法：
select * from t_customer where customer_id in(select distinct customer_id from t_order);
```

除了 EXISTS，也可以使用 NOT EXISTS 条件从 SELECT、UPDATE、DELETE 语句中获取子查询的返回结果。**NOT EXISTS 用于检查一个子查询是否返回任何行**，如果**没有行返回，那么 NOT EXISTS 将返回 true。**

例如，想要查找所有没有下过订单的顾客，可以使用以下语句：

```
select * from t_customer c where not exists(select * from t_order o where o.customer_id=c.customer_id);
```

在这个查询语句中，如果没有任何与顾客相关联的订单，则 NOT EXISTS 子查询将返回一个空结果集，这时候 WHERE 条件为 true，并将返回所有顾客信息。如果顾客有订单，则 NOT EXISTS 子查询的结果集将不为空，WHERE 条件为 false，则不会返回该顾客的信息。

总之，无论是 EXISTS 还是 NOT EXISTS，都是非常有用的 SQL 工具。可以通过它们来**结合子查询来动态过滤查询结果**，使 SQL 查询变得更加灵活和高效。

in和exists区别

IN 和 EXISTS 都是用于关系型数据库查询的操作符。不同之处在于：

1. **IN 操作符是根据指定列表中的值**来判断是否满足条件，而 **EXISTS 操作符则是根据子查询的结果是否有返回记录集**来判断。
2. **EXISTS 操作符通常比 IN 操作符更快**，尤其是在子查询返回记录数很大的情况下。因为 EXISTS 只需要判断是否存在符合条件的记录，而 IN 操作符需要比对整个列表，因此执行效率相对较低。

3. **IN 操作符可同时匹配多个值**，而 **EXISTS 只能匹配一组条件。**

下面是一个简单的示例，用于演示 IN 和 EXISTS 之间的区别。

假设有两个表 orders 和 products，orders 表中记录了订单信息，products 表中记录了商品信息。现在想查询所有“手机”和“平板电脑”这两种商品中，至少有一笔订单销售了 $1000 以上的商品：

使用 IN 操作符：

```
SELECT *
FROM products
WHERE product_name IN ('手机', '平板电脑')
AND product_id IN (
  SELECT product_id
  FROM orders
  WHERE order_amount > 1000
);
```

使用 EXISTS 操作符：

```
SELECT *
FROM products
WHERE product_name IN ('手机', '平板电脑')
AND EXISTS (
  SELECT *
  FROM orders
  WHERE orders.product_id = products.product_id
  AND order_amount > 1000
);
```

总之，IN 和 EXISTS 都是用于条件过滤的操作符，但其实现方式和性能特点都不同，需要根据具体情况进行选择和使用。

union&union all

不管是 union 还是 union all 都可以**将两个查询结果集进行合并。**

union 会对合并之后的查询结果集进行**去重**操作。

union all 是直接将查询结果集合并，**不进行去重**操作。

（union all和union**都可以完成的话，优先选择union all**，union all因为**不需要去重，所以效率高**一些。）

案例：查询工作岗位是MANAGER 和 SALESMAN的员工。

```
select ename,sal from emp where job='MANAGER'
union all
select ename,sal from emp where job='SALESMAN';
```

以上案例采用or也可以完成，那 or 和 union all 有什么区别?

考虑**走索引优化之类的选择 union all，其它选择 or。** （使用 OR 会导致索引失效）

```
# 用 OR 的写法:
SELECT ename, sal FROM emp WHERE job = 'MANAGER' OR job =
'SALESMAN';
```

注意：两个结果集**合并时，查询的列数量要相同**：

> Oracle数据库里还要求数据类型一致，但MySQL不做要求。

```
select ename, sal from emp   # 查询两列
union
select ename, sal,job from emp;  # 查询三列
# ERROR 1222 (21000): The used SELECT statements have a
different number of columns
```

```
select ename from emp
union
select dname from dept;   # 可以的
```

limit

limit作用：**查询第 x 条到第 y 条的记录。**通常是因为表中数据量太大，需要分页显示。

limit语法格式： （开始下标从 0 开始） 【 limit 语句 写在最末尾 】

```
limit 开始下标, 长度
```

案例：查询员工表前5条记录

```
select ename,sal from emp limit 0, 5;  # 从第一个记录开始取，一共取5条
```

如果下标是从0开始，可以简写为：

```
select ename,sal from emp limit 5;
```

案例：查询工资排名在前5名的员工（limit是在order by执行之后才会执行的）

```
select ename,sal from emp order by sal desc limit 5;
```

- 通用的分页 SQL 语句：(假设知道每页展示的条数和总分页数量，找出第 N 页的记录)

假设每页显示3条记录：pageSize = 3

```
#第1页：
limit 0, 3
#第2页：
limit 3, 3
#第3页：
limit 6, 3

#第pageNo页：
limit (pageNo - 1) * pageSize, pageSize
```

35道DQL练习题

060-35道DQL练手题123*哔哩哔哩*bilibili

第五章.表相关的操作

创建表

语法格式：

```
create table 表名(
  字段名1 数据类型,
  字段名2 数据类型,
  ......
  字段名n-1 数据类型,
  字段名n 数据类型          # 最后一个字段名结尾没有逗号','
);
```

例如：创建学生表

```
create table t_student(
  no int,
  name varchar(10),     # 可变长度字符串
  gender char(1) default '男'    # 定长字符串。   插入没有指定插入值时系统会插入默认值
);                                          # 不使用default指定的话，默认值就是 NULL
```

插入数据

语法格式：

```
insert into 表名(字段名1, 字段名2, 字段名3,......) values (值1,值2,值3,......);
```

字段名和值要**一一对应**。类型要一一对应，数量要一一对应。

字段名也可以省略，如果字段名省略就表示把所有字段名都写上去了，并且顺序和建表时的顺序相同。

如果只写了部分字段名和对应值，则剩下的字段会赋默认值。

例如：

```
insert into t_user(no, name) values(3, 'Lucy');   # 字段名及值可部分省略，结果就是赋的默认值
```

```
insert into t_user values(3, 'Mary', 'F');   # 省略全部字段名，那就要一一对应，值部分不能有省的
```

删除表

语法格式：

```
drop table 表名;      # 表名不存在时会报错
```

或者

```
drop table if exists 表名;     # 判断是否存在这个表，如果存在则删除，表不存在也不会报错。（避免不存在时的报错）
```

> Oracle数据库里不支持 `if exists` 的语法。

注意： 删表时谨慎点，删除没有语句直接恢复的 (只能采用回滚或备份等方式)。

MySQL数据类型

> 不同数据库系统之间的数据类型并不完全相同，要注意看具体文档。

数据类型（data_type）是指**系统中所允许的数据的类型。**数据库中的每个列都应该有适当的数据类型，用于限制或允许该列中存储的数据。

例如，列中存储的为数字，则相应的数据类型应该为数值类型。

如果使用错误的数据类型可能会严重影响应用程序的功能和性能，所以在设计表时，应该特别重视数据列所用的数据类型。更改包含数据的列不是一件小事，这样做可能会导致数据丢失。因此，在**创建表时必须为每个列设置正确的数据类型和长度。**

MySQL 的数据类型可以分为整数类型、浮点数类型、定点数类型、日期和时间类型、字符串类型、二进制类型等。

整数类型

tinyint：1个字节（微小整数）
smallint：2个字节（小整数）
mediumint：3个字节（中等大小的整数）
int（integer）：4个字节（普通大小整数）
bigint：8个字节（大整数）

浮点数类型

float：4个字节，单精度（**最多5位小数**）【超过5位小数会进行四舍五入到5位】

double：8个字节，双精度（最多16位小数）

> double与float也可以写成：`double(3, 2)`此时就规定了3位有效数字，2位小数

定点数类型

decimal：定点数类型。　　底层**实际上采用字符串的形式存储数字。**

语法：`decimal(precision, scale)`　　（支持的有效数字最多65个，小数位最多30个）

例如：decimal(3, 2) 表示最多3个有效数字，最多2个小数。即从-9.99 ~ 9.99 。　10其实是10.00，为四位有效数字，所以存不进去。

> 有效数字会忽略前导零。

```
create table t_data_type02(a decimal(3,2));
insert into t_data_type02(a) values(10);   # ERROR 1264(22003):Out of range value for column'a' at row1
insert into t_data_type02(a) values(1.23); # √
insert into t_data_type02(a) values(1.123456789);  # 会四舍五入到1.12
```

日期和时间类型

year：1个字节，只存储年，格式YYYY
time：3个字节，只存储时间，格式HH:MM:SS / HHMMSS
date：3个字节，只存储年月日，格式：YYYY-MM-DD
datetime：8个字节，存储 **年月日+时分秒**，格式：YYYY-MM-DD HH:MM:SS（从公元1000年 ~ 公元9999年）
timestamp：4个字节，存储年月日+时分秒，格式：YYYY-MM-DD HH:MM:SS（从公元1980年 ~ 公元2040年）或者格式为 YYYYMMDDHHMMSS　（ 采用这种格式不需要使用单引号，当然你使用单引号也可以）

```
# timestamp 类型
insert into t_data_type(a) values(20001010101010);   # 2000.10.10 10:10:10
insert into t_data_type(a) values('20001010101010');   # 带单引号也可以
insert into t_data_type(a) values(20100807);  # 忽略 时分秒 也可以  (则时分秒默认按0填入)
select * from t_data_type;
/*    a
   2000-10-10 10:10:10
   2000-10-10 10:10:10
   2010-08-07 00:00:00
*/
INSERT INTO t_data_type(a) VALUES ('2023-09-21');
```

字符串类型

char

char(m)：m长度是 **0~255**个字符。
固定长度字符串，在定义时指定字符串列长。当保存时，在右侧填充空格以达到指定的长度。m表示列的长度，范围是 0～255 个字符。
例如，CHAR(4) 定义了一个固定长度的字符串列，包含的字符个数最大为 4。当插入的字符长度大于4，则报错（除非超过4个长度之后都是空格字符，则空格字符会自动被删除用来保证插入的成功）。

varchar

varchar(m)：m 长度是 **0~16383**个字符
长度可变的字符串。varchar 的最大实际长度由最长的行的大小和使用的字符集确定，而**实际占用的空间为字符串的实际长度加 1。**
例如，varchar(50) 定义了一个最大长度为 50 的字符串，如果插入的字符串只有 10 个字符，则实际存储的字符串为 10 个字符**和一个字符串结束字符。** varchar 在值保存和检索时尾部的空格仍保留。

插入值	CHAR(4)	存储需求	VARCHAR(4)	存储需求 (实际长度加一)
`''`	`''`	4字节	`''`	1字节

插入值	CHAR(4)	存储需求	VARCHAR(4)	存储需求 (实际长度加一)
`'ab'`	`'ab '`	4字节	`'ab'`	3字节
`'abc'`	`'abc '`	4字节	`'abc'`	4字节
`'abcd'`	`'abcd'`	4字节	`'abcd'`	5字节
`'abcd '`	`'abcd'` (末尾多出的两空格会删掉)	4字节	`'abcd '`	7字节

> Char 比 Varchar效率高，但可能浪费空间。 Char对应到Java里就是String不可变字符串。

text

text类型： 适合存储长文本，也是字符串文本。

- tinytext 表示长度为 255字符的 TEXT 列。
- **text 表示长度为 65535字符**的 TEXT 列。
- mediumtext 表示长度为 16777215字符的 TEXT 列。 （一千六百万字符）
- longtext 表示长度为 4294967295 或 4GB 字符的 TEXT 列。 （四十二亿字符）

enum

enum类型：

- 语法：<字段名> enum('值1','值2',...)
- 该字段**插入值时，只能是指定的枚举值。**

```
create table t_data_type08(
  season enum('SPRING','SUMMER')
);
insert into t_data_type08(season) values('SPRING');  # 插入值只能是被指定的范围其中的值
insert into t_data_type08(season) values('spring');  # 可以插入表中重复的值，且MySQL中这里不区分大小写

insert into t_data_type08(season) values('AUTUMN'); # ERROR 1265 (01000): Data truncated for column'season' at row 1
insert into t_data_type08(season) values('SPRING,SUMMER'); # ERROR 1265 (01000): Data truncated for column 'a' at row 1
```

set

set类型：

- 语法：<字段名> set('值1','值2','值3',...) 注意：**一次插入的值里不可重复。**
- 该字段**插入值时，只能是指定的值。** 插入值可以同时多个。

```
create table t_data_type09(
  a set('abc','xyz')
);
insert into t_data_type09(a) values('abc');
insert into t_data_type09(a) values('abc');
insert into t_data_type09(a) values('abc,xyz');  # 插入可以多个
insert into t_data_type09(a) values('abc, xyz');  # 不要有空格，会报错 ERROR 1265 (01000): Data truncated for column
insert into t_data_type09(a) values('abc,abc');  # 重复值只会插入 abc 一个

/*   a
    abc
    abc
    abc,xyz
    abc
*/
```

二进制类型 (blob)

BLOB类型：二进制大对象 (Binary Large OBject)，**可以存储图片、声音、视频等**文件。

- blob：小的，最大长度65535个字节
- mediumblob：中等的，最大长度16777215个字节
- longblob：大的，最大长度4GB的字节

> 不过一般要存储视频之类都是把视频托管到文件服务器，留下URL字符串存在数据库表中。

增删改表结构DDL

> 一般对表更改做的少，且使用Navicat等也可实现。重点度不如DQL、DML

创建一个表

```
create table t_student(
  no bigint,
  name varchar(255),
  age int comment '年龄'   # comment '注释'  给对应字段添加注释。
);
```

```
# 查看已有表的所有字段注释
show full columns from t_student;
/*  Field | Type     | Collation        | Null | Key | Default | Extra |       Privileges               | Comment
    no      bigint      NULL              YES           NULL           select,insert,update,references
   name   varchar(255) utf8mb4_0900_ai_ci  YES           NULL           select,insert,update,references
    age      int         NULL              YES           NULL           select,insert,update,references |   AGE
*/
```

查看建表语句

查看当初建立这张表时使用的建表语句，如果有`comment`注释也会显示。

```
show create table 表名;
```

查看基本的表结构：

```
desc 表名
describe 表名
```

> MySQL中 标识符 可以用 反引号 (backtick) “ ` ” 括起来，例如：
>
> ```
> create table `t_product` (`pname` varchar(255));
> ```

修改表名

```
alter table 表名 rename 新表名;
```

新增字段

```
alter table 表名 add 字段名 数据类型;    # 新增的字段不设默认值那就是NULL
alter table 表名 add 字段名 数据类型 default 默认值;
```

修改字段名和数据类型

```
alter table 表名 change 旧字段名 新字段名 新数据类型;
ALTER TABLE users CHANGE username user_name VARCHAR(100) NOT NULL DEFAULT 'default_user'; # 设置默认值或更改NULL约束
```

> 更改数据类型时，确保新类型与现有数据兼容。防止出现数据丢失或转换错误。

修改字段数据类型

```
alter table 表名 modify column 字段名 数据类型;
```

- `MODIFY COLUMN`关键字用于更改现有列的数据类型，但不允许更改列名。

删除字段

```
alter table 表名 drop 字段名;
```

DML语句

当对表中的数据进行增删改的时候，使用DML语句（数据操纵语言）。主要包括：insert、delete、update (这三条都支持事务)。

insert 增

语法格式：

```
insert into 表名(字段名1,字段名2,字段名3,...) values(值1,值2,值3,...);   # 要插入的值的顺序、数量 与 写出来的字段对应
```

表名后面的小括号当中的字段名如果省略掉，表示自动将所有字段都列出来了，并且字段的顺序和建表时的顺序一致。
一般为了可读性强，建议把字段名写上。

```
insert into 表名 values(值1,值2,值3,...);      # 此时值必须全部对应表中所有字段及顺序，值不可有省略的
```

```
insert into t_user(no, name, gender) values(3, 'Jerry', 'M');
insert into t_user(no, name) values(4, 'Merry');   # 此时这条记录的gender字段值就是默认值

insert into t_user(no, name) values(5);  # Error, 插入值的数量要对应
insert into t_user values(6, 'Mark');    # Error, 如果省略字段名，插入值的数量和顺序要与表中全部对应，不得省略
```

一次批量插入多条记录：`insert into 表名(字段名1,字段名2,...) values(值1,值2,...),(值1,值2,...),...;`

```
insert into t_stu(no,name,age) values(1,'jack',20),(2,'lucy',30),(3,'mark',25);
```

delete 删

语法格式：

```
# 将所有记录全部删除
delete from 表名;

# 删除符合条件的记录
delete from 表名 where 条件;

delete from t_user where name='lisi';
```

以上的删除属于DML的方式删除，**这种删除的数据是可以通过事务回滚的方式重新恢复**的，但是**删除的效率较低。**（这种删除是支持事务的）

另外还有一种删除表中数据的方式，但是这种方式不支持事务，不可以回滚，删了之后数据是永远也找不回来了。这种删除叫做：**表被截断。**

```
truncate table 表名;       # 删除效率高，但不可恢复
```

注意：这个语句**删除效率非常高**，巨大的表，瞬间干掉所有数据。但**不可恢复。**

> `TRUNCATE` 不属于DML语句，不支持事务。

update 改

语法格式：

```
update 表名 set 字段名1=值1, 字段名2=值2, 字段名3=值3 where 条件;
# 不加where条件就是全部记录都更新
```

约束constraint

创建表时，可以**给表的字段添加约束，可以保证数据的完整性、有效性。**比如大家上网注册用户时常见的：”用户名不能为空“等提示信息。

约束通常包括：

- 非空约束：not null

- 检查约束：check　　[MySQL 8 之后才支持， 而Oracle数据库是一直支持的]
- 唯一性约束：unique
- 主键约束：primary key　　(简称 “PK”)
- 外键约束：foreign key　　(简称 “FK”)

非空约束 not null

语法格式：

```
create table t_stu(
  no int,
  name varchar(255) not null,
  age int
);
```

要求name字段不能为空。插入数据时如果没有给name指定值，则会报错。

```
insert into t_stu(no,age) values(2,30);    # 不给有非空约束的name字段赋值就会报错，除非有默认值
# ERROR 1364 (HY000): Field'name' doesn't have a default value

create table t_stu(
  no int,
  name varchar(255) not null default 'Jack',    # 设置默认值就不会报错了
  age int
);
```

检查约束 check()

```
create table t_stu(
  no int,
  name varchar(255),
  age int,
  check(age > 18)
);
```

```
insert into t_stu(no, name,age) values(1,'jack',20);  # 正常

insert into t_stu(no,name,age) values(2,'tom',18);
# ERROR 3819(HY000):Check constraint't_stu_chk_1'is violated.
```

唯一性约束 unique

语法格式：

```
create table t_stu(
  id int,
  name varchar(255),
  email varchar(255) unique
);
```

email字段设置为唯一性，唯一性的字段值是可以为NULL的，为NULL时就可以都是NULL。其他情况都不能重复。

> 数据库理论中，`NULL` 表示一个未知的值或缺失的信息，它们不被视为实际的数据值，因此不适用于唯一性约束的规则；SQL 标准规定，`NULL` 值在比较时是“不可比较”的，这意味着 `NULL` 值不能与其他 `NULL` 值或任何非 `NULL` 值进行比较。因此，数据库系统通常允许多个 `NULL` 值存在于具有唯一性约束的列中，因为它们无法确定这些 `NULL` 值是否相等。

```
insert into t_stu(id,name,email)
values(2,'lucy','lucy@123.com');

insert into t_stu(id,name,email)
values(3,'Tom','lucy@123.com');   # 不能重复
# ERROR 1062 (23000): Duplicate entry'1ucy@123.com' for
key't_user.email'

insert into t_stu(id,name) values (10, 'zhangsan');
insert into t_stu(id,name) values (10, 'zhangsan');   # 对于为
unique的email字段是NULL，此时可以重复，即
/*      id        name      email
        2         lucy     lucy@123.com
        10      zhangsan     NULL
        10      zhangsan     NULL
*/
```

```
create table t_stu(
  id int,
  name varchar(255),
  email varchar(255) unique not null   # 非空约束与唯一性约束联合使用
);
```

以上**在字段后面添加的约束，叫做列级约束。**

添加约束还有另一种方式：**表级约束**：在所有字段写完后再进行约束

```
create table t_stu(
  no int,
  name varchar(255),
  email varchar(255),
  unique(email)     # 表级约束
);
```

使用**表级约束可以为多个字段添加联合的约束**唯一。

```
create table t_user(
  id int,
  name varchar(255),
  email varchar(255),
  unique(name,email)    # 表示 name与email两个字段加起来一块不能重复
);
```

```
insert into t_user(id, name,email) values(1,'jack','abc@123.com');
insert into t_user(id, name,email) values(2,'jack','xyz@123.com');  # 虽然 name相同，但都正常添加

insert into t_user(id, name,email) values(3,'jack','xyz@123.com'); # ERROR 1062(23000): Duplicate entry 'jack-xyz@123.com' for key 'user.name'
```

> 注意：如果使用列级约束 email varchar(255) unique, name varchar(255) unique 意思就是既要name不相同，又要保证 email不相同。互相是独立的约束。

创建表级约束时可以给约束起名字，将来可以**通过约束的名字来删除约束**：（创建时不起名的话，系统会自动给约束起名）

`constraint 约束名 约束`

```
create table t_user(
  no int,
  name varchar(255),
  email varchar(255),
  constraint t_stu_name_email_unique unique(name,email)
);
```

```
# 删除唯一性约束
ALTER TABLE 表名 DROP INDEX 唯一性约束名;
ALTER TABLE 表名 DROP KEY 唯一性约束名;
```

所有的约束都存储在一个系统表当中：table_constraints。这个系统表在这个数据库当中：information_schema

```
# 查看约束
use information_schema;
select constraint_name from table_constraints where
table_name='t_user';
```

主键约束 PK

1. 主键：primary key，简称PK
2. **主键约束的字段不能为NULL，并且不能重复。**
3. **任何一张表都应该有主键**，没有主键的表可以视为无效表。　主键约束的字段叫主键字段，字段里的值叫主键值。
4. **主键值是这行记录的唯一标识。**在数据库表中**即使两条数据一模一样，但由于主键值不同，也认为是两条完全不同的数据。**
5. 一张表只能有一个主键，不管这个主键是单一主键还是复合主键，都只能有一个主键。
6. 主键分类：
    1. 根据字段数量分类：
        1. 单一主键（**1个字段作为主键**） ==> 建议的
        2. 复合主键（**2个或2个以上的字段作为主键**）
    2. 根据业务分类：
        1. 自然主键（主键和任何业务都无关，只是一个单纯的自然数据）===> 建议的

2. 业务主键（主键和业务挂钩，例如：银行卡账号作为主键） （不建议，因为业务有可能变化）

7. 单一主键（建议使用这种）

```
create table t_student(
  id bigint primary key,    # 列级约束主键
  sno varchar(255) unique,
  sname varchar(255) not null
)
```

```
create table t_student(
  id bigint,
  sno varchar(255) unique,
  sname varchar(255) not null,
  primary key(id)    # 表级约束主键
)
```

8. 复合主键（很少用，了解）（设计范式里不推荐使用复合主键）

> 这种情况下，虽然单个列的值可能不是唯一的，但是列的组合值必须唯一。复合主键要求所有组成主键的字段**都不能包含** `NULL` **值。** 此时通常不会说单个字段是主键，而是说这些字段共同构成了主键。

```
create table t_user(
  no int,
  name varchar(255),
  age int,
  primary key(no,name)  # 只能使用表级约束
);
```

9. 主键自增：既然主键值是一个自然的数字，**MySQL 为主键值提供了一种自增机制**，不需要我们程序员维护，MySQL 自动维护该字段

```
create table t_vip(
  no int primary key auto_increment,   # 这样主键值会自动生成，插入数据时不用管主键字段
  name varchar(255)
);
```

```
insert into t_vip(name) values('zhangsan');  # 此时数据就不用去管主键字段的值。

# 这时不要采用省略字段名的写法，如 insert into t_vip values('zhangsan'); 会报错，因为不知道no的赋值。
```

> 具体设计中也不一定靠主键自增。 且其他数据库中不一定有主键自增。
>
> MySQL 中，如果表的主键没有设置为自增（AUTO_INCREMENT）且没有设置默认值，那么在插入数据时，你需要显式地为每条记录指定一个主键值。

外键约束 FK

有这样一个需求：要求设计表，能够存储学生以及学校信息。

1. 第一种方案：一张表

id (主键)	name	age	sno	sname
1	张三	20	1	北区一中
2	李四	21	1	北区一中
3	王五	19	2	南区二中
4	赵六	18	2	南区二中

这种方式会导致数据冗余，浪费空间。（这里学校的信息重复存了，产生冗余）

2. 第二种方案：两张表：一张存储学生，一张存储学校

t_school 表

sno (主键)	sname
1	北区一中
2	南区二中

t_student 表

id (主键)	name	age	sno (外键)
1	张三	20	1

id (主键)	name	age	sno (外键)
2	李四	21	1
3	王五	19	2
4	赵六	18	2

如果采用以上两张表存储数据，**对于学生表来说，sno这个字段的值是不能随便填**的，这个**sno是学校编号**，必须**要求这个字段中的值来自学校表的sno。**

为了达到要求，此时就**必须要给t_student表的sno字段添加外键约束**了。

2. 外键约束：foreign key，简称FK。
3. **添加了外键约束的字段中的数据必须来自其他字段，不能随便填。**
4. 假设**给a字段添加了外键约束，要求a字段中的数据必须来自b字段，b字段不一定是主键，但至少要有唯一性。**
5. 外键约束可以给单个字段添加，叫做**单一外键。**

   也可以给多个字段联合添加，叫做**复合外键。**复合外键很少用。
6. **a表如果引用b表**中的数据，可以**把b表叫做父表，把a表叫做子表。** 【注意顺序】
   1. 创建表时，**先创建父表，再创建子**表。
   2. 插入数据时，**先插入父表，在插入子**表。
   3. 删除数据时，**先删除子表，再删除父**表。
   4. 删除表时，先删除子表，再删除父表。
   5. 如果需要更新子表中的外键值以引用父表中的另一个记录，首先确保父表中存在要引用的新记录。
7. 如何添加外键： `foreign key(字段名) references 父表名(字段名)`

```
create table t_school(
  sno int primary key,
  sname varchar(255)
);
create table t_student(
  no int primary key auto_increment,
  name varchar(255),
  age int,
  school_no int,
  constraint t_school_sno_fk foreign key(school_no)
references t_school(sno)
);
```

> MySQL 中，如果表的主键没有设置为自增（AUTO_INCREMENT）且没有设置默认值，那么在插入数据时，你需要显式地为每条记录指定一个主键值。

```
# 假设外键school_no只能是1或2，此时想插入3的数据就会报错
insert into t_student (name, age, school_no) values('wangwu', 20, 3);
# ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails
```

8. **级联删除**

创建子表时，外键可以添加：on delete cascade，这样在**删除父表数据时，受引用影响的子表会级联删除。** 【 谨慎使用 】

没有设置级联删除约束的情况下，先删父表数据会报错。 只能先删完子表再删父表。

```
create table t_student(
  no int primary key auto_increment,
  name varchar(255),
  age int,
  sno int,
  constraint t_school_sno_fk foreign key(sno) references t_school(sno) on delete cascade
);
```

除了通过建表的时候建约束，还可以修改表的约束：

```
# 删除约束
alert table t_student drop foreign key t_student_sno_fk;
# 添加约束
alert table t_student add constraint t_student_sno_fk foreign key(sno) references t_school(sno) on delete cascade;
```

9. 级联更新

创建子表时，外键约束添加：on update cascade ， 这样 **修改父表中被外键引用的字段值时，子表的外键数据值会跟着改变。**

```
create table t_student(
  no int primary key auto_increment,
  name varchar(255),
  age int,
  sno int,
  constraint t_school_sno_fk foreign key(sno) references
t_school(sno) on update cascade
);
```

10. 级联置空

创建子表时，可以在外键约束添加：on delete set null， 这样 **删除父表记录时，子表中所有引用该父表记录的外键字段被置为NULL。**但子表记录仍然保留。

```
create table t_student(
  no int primary key auto_increment,
  name varchar(255),
  age int,
  sno int,
  constraint t_school_sno_fk foreign key(sno) references
t_school(sno) on delete set null
);
```

- 避免循环依赖：指在设计数据库的表关系时，防止出现两个或多个表通过外键约束相互引用，形成一个闭环的情况。这种闭环会导致数据完整性问题，使得数据的插入、更新和删除操作变得复杂和困难。

假设有三个表：A、B和C。表之间的关系如下：

- 表A有一个外键引用表B的主键。
- 表B有一个外键引用表C的主键。
- 表C有一个外键引用表A的主键。

这种设计就构成了一个循环依赖：A → B → C → A。在这种情况下，如果尝试插入、更新或删除这些表中的任何数据，都可能导致外键约束冲突，因为数据库管理系统需要同时处理多个相互依赖的外键约束。

循环依赖的问题在于：

1. **数据操作复杂**：当尝试修改数据时，可能需要多个步骤来确保所有相关的表都满足外键约束，这会增加操作的复杂性。
2. **性能影响**：数据库系统需要检查多个表的外键约束，这可能会增加查询和更新操作的开销。

3. **数据不一致**：在并发操作或事务处理中，循环依赖可能导致数据不一致，因为可能难以预测和维护数据之间的正确关系。
4. **难以维护**：循环依赖使得数据库的维护和理解变得更加困难，特别是在大型和复杂的数据库系统中。

为了避免循环依赖，可以采取以下措施：

- **重新设计数据模型**：重新考虑表之间的关系，确保没有循环依赖。可能需要重新设计数据模型，以消除循环关系。
- **使用其他方法维护数据完整性**：而不是依赖外键约束，可以考虑使用触发器、应用程序逻辑或其他数据库约束（如唯一约束或检查约束）来维护数据完整性。
- **限制外键的使用**：在某些情况下，可能不需要使用外键来维护表之间的关系，可以考虑使用其他方法，如存储关联数据的数组或XML列。
- **分层数据设计**：将数据关系设计成层次结构，而不是相互交叉的关系，这样可以避免循环依赖。

在设计数据库时，仔细规划表之间的关系是非常重要的，以确保数据的一致性和系统的可维护性。

第六章.数据库设计三范式

什么是数据库设计三范式?

是数据库表设计的原则依据，以后设计数据库表要尽量遵守这三个范式（不是必须要遵守，某些情况下有特殊）。这三范式教你怎么设计数据库表**有效**，并且**节省空间。**

三范式

第一范式：**任何一张表都应该有主键，每个字段是原子性的不能再分。** 【第一范式必须遵守，别的可以不保证】

- 以下表的设计不符合第一范式：无主键，并且联系方式可拆分。

sno	sname	contact
1001	zhangsan	zs@qq.com,135123456
1002	lisi	ls@qq.com,181234567

sno	sname	contact
1001	wangwu	ww@qq.com,192123489

- 应该这样设计：

sno	sname	email	phone
1001	zhangsan	zs@qq.com	135123456
1002	lisi	ls@qq.com	181234567
1003	wangwu	ww@qq.com	192123489

第二范式 (2-NF)：建立在第一范式基础上的，另外要求**所有非主键字段完全依赖主键，不能产生部分依赖。**

- 以下表存储了学生和老师的信息：

sno (PK)	tno (PK)	sname	tname
1001	1	张三	王老师
1002	2	李四	赵老师
1003	1	王五	王老师
1001	2	张三	赵老师

虽然符合第一范式，但是违背了第二范式，学生姓名、老师姓名都产生了部分依赖（“张三”值只由复合主键中sno一个字段决定）。导致数据冗余。

所以复合主键不建议用，容易产生部分依赖。

> 部分依赖是指在一个复合主键（由多个属性组成的主键）的情况下，一个非主属性依赖于主键的一部分，而不是整个主键

- 以下这种设计方式就是符合第二范式的：

> 多对多（一个学生对多个老师，一个老师对多个学生）三张表，关系表中加外键。

学生表

学生编号 (PK)	学生姓名
1001	张三
1002	李四
1003	王五

老师表

教师编号 (PK)	教师姓名
1	王老师
2	赵老师

学生-老师关系表

id (PK)	学生编号 (FK)	教师编号 (FK)
1	1001	1
2	1002	2
3	1003	1
4	1001	2

第三范式 (3-NF)：建立在第二范式基础上的，**非主键字段不能传递依赖于主键字段。**

> 传递依赖是指一个属性依赖于另一个非主属性的情况；意味着存在一个属性依赖于另一个非主属性，而这个非主属性本身又依赖于一个主属性。

- 以下设计方式就是违背第三范式的

学生编号 (PK)	sname	cno	cname
1001	张三	1	一年一班
1002	李四	2	一年二班
1003	王五	3	一年三班
1004	赵六	3	一年三班

以上虽然满足第二范式，但因为产生了传递依赖（cname依赖cno，cno又依赖主键），导致班级名称冗余。

- 以下这种方式就是符合第三范式的：

> 一对多（一个学生对一个班级，一个班级对多个学生），两张表，多的表加外键

sno (PK)	sname	cno (FK)
1001	张三	1
1002	李四	2
1003	王五	3
1004	赵六	3

班级表

cno (PK)	cname
1	一年一班
2	一年二班
3	一年三班

一对多怎么设计

口诀：一对多（一个A对多个B，一个B对一个A）两张表（A表、B表），多的表中加外键。

例子见第三范式的。

多对多怎么设计

口诀：多对多（一个A对多个B，一个B对多个A）三张表 （A表、B表、A-B关系表），关系表中加外键。

例子见第二范式的。

一对一怎么设计

两种方案：

1. 第一种：主键共享。

例如，一张丈夫表，一张妻子表：

husband_id (PK)	sname
1	张三
2	李四
3	王五

wife_id (PK + FK)	wname
1	小花
2	小兰
3	小红

> 为什么不直接放在一张表里？ 实际中比如用户信息存储，把用户名与密码单独放一张表，剩下一张表存用户名、用户简介等其他一长串用户信息。

2. 第二种：外键唯一。 [这种方案居多。 一对一其实是特殊的一对多，对外键加个unique约束就行]

husband_id (PK)	h_name
1	张三
2	李四
3	王五

wife_id (PK)	w_name	husband_id (FK + unique)
1	小花	1
2	小兰	2

wife_id (PK)	w_name	husband_id (FK + unique)
3	小红	3

注意：最终设计以满足客户需求为原则，有的时候会拿空间换速度。

第七章.视图 (view)

1. **只能将select语句创建为视图。** （视图可看作对select查询语句的包装）
2. 创建视图： 【创建时 as 后面只能是 DQL语句】

```
# ↓ 直接创建视图 ，如果数据库中已有同名视图，创建就会报错。
create view v_emp as select e.ename, d.dname from emp e join
dept d on e.deptno = d.deptno;

# ↓ (创建或替换) 检查视图是否已经存在，如果存在，它会替换原有的视图定义；
如果不存在，它会创建一个新的视图
create or replace view v_emp as select e.ename,d.dname from
emp e join dept d on e.deptno = d.deptno;
```

```
# 面向视图查询
select * from v_emp;
select e.ename from v_emp;

# 隐藏原始字段名
create view myview02 as select empno a,ename b,sal c from emp;
select empno from myview02;
# ERROR 1054 (42522): Unknown column 'empno' in 'field list'
select a, b, c from myview2 # 只能使用视图中的字段名来查
```

3. 视图作用
    1. 如果开发中有**一条非常复杂的SQL，而这个SQL在多处使用**，会给开发和维护带来成本。**使用视图可以降低开发和维护的成本。**
    2. 视图可以**隐藏原始表的字段名、原始表名。** 可用于保密开发。
4. 修改视图： （如果视图不存在会报错）

```
alter view v_emp as select e.ename,d.dname,d.deptno from emp e
join dept d on e.deptno = d.deptno;
```

> ALTER VIEW 更多用于修改视图的属性，而 CREATE OR REPLACE VIEW 通常用于完全替换视图的查询定义。
>
> 如果你确定视图已经存在并且只需要修改它的定义，使用 `ALTER VIEW`。如果你不确定视图是否存在，或者你想要确保视图的定义是最新的，即使需要替换现有的视图，那么使用 `CREATE OR REPLACE VIEW`

5. 删除视图

```
drop view if exists v_emp;
```

6. **对视图增删改**（DML：insert delete update）**可以影响到原表数据。**

第八章.事务 (Transaction)

事务概述

1. **事务是一个最小的工作单元。**在数据库当中，**事务表示一件完整的事。** 事务（Transaction）是指**一组原子性的数据库操作，这些操作要么全部成功，要么全部失败。**
2. 一个业务的完成可能需要多条DML语句共同配合才能完成，例如转账业务，需要执行两条DML语句，先更新“张三”账户的余额，再更新“李四”账户的余额，为了保证转账业务不出现问题，就必须保证要么同时成功，要么同时失败。

   怎么保证同时成功或者同时失败呢？ 就需要使用事务机制。
3. 用了事务机制之后，在同一个事务当中，多条DML语句会同时成功，或者同时失败，不会出现一部分成功，一部分失败的现象。
4. **事务只针对DML语句有效**：因为**只有这三个语句是改变表中数据**的。
   1. insert
   2. delete
   3. update

事务四大特性：ACID

1. **原子性（Atomicity）**：是指**事务包含的所有操作要么全部成功，要么同时失败。**
2. **一致性（Consistency）**：是指**事务开始前，和事务完成后，数据应该是一致的。**例如张三和李四的钱加起来是5000，中间不管进行过多少次的转账操作(update)，总量5000是不会变的。这就是事务的一致性。

3. **隔离性（Isolation）**：隔离性是当多个用户并发访问数据库时，比如操作同一张表时，**数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。**
4. **持久性（Durability）**：持久性是指**一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的**，即便是在**数据库系统遇到故障的情况下也不会丢失提交事务的操作。

> 有日志记录等机制保证持久性。

演示MySQL事务

MySQL默认情况下采用的事务机制是：自动提交。所谓自动提交就是只要执行一条DML语句则提交一次。

```
# 自动提交，即执行这一条语句，就会自动开启事务，然后执行事务的过程，执行完后提交。只要提交事务，事务就结束并持久化到硬盘了
insert into t_user(id,name) values (1, 'jack');
```

关闭事务的自动提交：

- 在dos命令窗口中开启MySQL事务：`start transaction;` 或者：`begin;`
- 回滚事务：`rollback;`　（把之前的历史操作清除掉，回到原点）
- 提交事务：`commit;`　　（所有操作成功了）

只要执行以上的 rollback 或者 commit ，事务都会结束。

```
start transaction;  # 开启事务　（不用这条语句就会是自动提交机制）
# ...若干条 DML 语句 ...
commit;    或者   rollback;    # 事务结束
```

```
start transaction;    # 开启事务
select * from t_user;
/* | id | name |
   |  1 | jack |    */
insert into t_user values(2, 'lisi');
insert into t_user values(3, 'wangwu');
rollback;     # 事务结束，事务开始后的操作都回滚

select * from t_user;
/* | id | name |
   |  1 | jack |    */
```

事务隔离级别

隔离级别	脏读现象	不可重复读	幻读
读未提交 (read uncommitted)	**存在**	存在	存在
读提交 (read committed)	不存在	**存在**	存在
可重复读 (repeatable read)	不存在	不存在	**存在**
串行化 (serializable)	不存在	不存在	不存在

隔离级别从低到高排序：读未提交 < 读提交 < 可重复读 < 串行化

不同隔离级别会存在不同的现象，现象按照严重性从高到低排序：脏读 > 不可重复读 > 幻读

> 实际中，读未提交 和 串行化 级别都很少设置。

查看与设置隔离级别

> Oracle数据库的默认隔离级别是第二档：读提交。

mysql默认的隔离级别：可重复读（REPEATABLE READ）。

- 查看当前会话的隔离级别：select @@transaction_isolation;　　[局部的，两个'@'符号不要省略]
- 查看全局的隔离级别：select @@gobal.transaction_isolation;

设置事务隔离级别：

- 会话级：set session transaction isolation level 隔离级别;

- 全局级：set global transaction isolation level 隔离级别;

```
# 设置会话级-读提交
set session transaction isolation level read committed;
```

> 会话（Session）是指用户与数据库服务器之间的一个交互周期，通常从用户连接到数据库服务器开始，直到用户断开连接结束.
>
> 简单理解就是控制台打开一个黑窗口并登录mysql就是一个会话，再打开一个黑窗口并登录就是另一个会话。

不同现象

1.脏读

(Dirty Read）指的是**一个事务读取了另一个事务尚未提交的数据**，即**读取了另一个事务中的脏数据（Dirty Data）**。在此情况下，**如果另一个事务回滚了或者修改了这些数据**，那么**读取这些脏数据的事务所处理的数据就是不准确**的。

2.不可重复读

(Non-Repeatable Read）指在**一个事务内，多次读取同一个数据行，得到的结果可能是不一样**的。这是**由于其他事务对数据行做出了修改操作，导致数据的不一致**性。

3.幻读

(Phantom Read) 指在**事务执行过程中，前后两次相同的查询条件得到的结果集不一致**，可能会**变多或变少**。

隔离级别

读未提交

（READ UNCOMMITTED）

A事务与B事务，**A事务可以读取到B事务未提交的数据。**这是最低的隔离级别。几乎两个事务之间没有隔离。这种隔离级别是一种理论层面的，在实际的数据库产品中，没有从这个级别起步的。

当事务隔离级别是读未提交时，三种现象都存在：脏读，不可重复读，幻读。

我们可以开启两个DOS命令窗口，模拟两个事务，演示一下这种隔离级别。三种现象中最严重的是脏读，只需要演示脏读问题即可，因为存在脏读的话，就一定存在不可重复读和幻读问题。

将全局事务隔离级别设置为：READ UNCOMMITTED

```
set global transaction isolation level read uncommitted;
```

开启两个DOS命令窗口来模拟两个事务：A事务与B事务。

A事务	B事务
mysql> use powernode	
	mysql> use powernode
mysql> start transaction;	
	mysql> start transaction;
mysql> select * from a;	
id : 1 2 3	
	mysql> insert into a values(4);
mysql> select * from a;	
id : 1 2 3 4	

通过以上测试，可以看到，A事务读取到了B事务还没有提交的数据。这种现象就是脏读。

读提交

（READ COMMITTED）

A事务与B事务，**A事务可以读取到B事务提交之后的数据。**Oracle数据库默认的就是这种隔离级别。

将数据库的全局事务隔离级别设置为读提交：READ COMMITTED

```
set global transaction isolation level read committed;
```

演示：

A事务	B事务
mysql> use powernode	

A事务	B事务
	mysql> use powernode
mysql> start transaction;	
	mysql> start transaction;
mysql> select * from a;	
id : 1 2 3	
	mysql> insert into a values(4);
mysql> select * from a;	
id : 1 2 3	
	mysql> commit;
mysql> select * from a;	
id : 1 2 3 4	

通过以上测试看出，A事务只能读取到B事务提交之后的数据。这种隔离级别解决了脏读问题，但肯定是存在不可重复读和幻读问题。因为只要事务B进行了增删改操作之后并提交了，事务A读取到的数据肯定是不同的。即：不可重复读和幻读都存在。

可重复读

（REPEATABLE READ）

这个隔离级别是MySQL数据库默认的。

A事务和B事务，A事务开启后，读取了某一条记录，然后**B事务对这条记录进行修改并提交，A事务读取到的还是修改前的数据。**这种隔离级别称为可重复读。

将数据库全局隔离级别修改为可重复读：

```
set global transaction isolation level repeatable read;
```

演示：

A事务	B事务
mysql> use powernode	

A事务	B事务
	mysql> use powernode
mysql> start transaction;	
	mysql> start transaction;
mysql> select empno,ename,sal from emp where empno=7369;	
empno : 7369 ename : 史密斯 sal 800.00	
	mysql> update emp set ename='SMITH',sal=8000 where empno=7369;
	mysql> commit;
mysql> select empno,ename,sal from emp where empno=7369;	
empno : 7369 ename : 史密斯 sal 800.00	

通过以上测试得知：当事务隔离级别设置为可重复读时，避免了不可重复读问题。

在MySQL当中，当事务隔离级别设置为可重复读时，能够避免幻读问题吗？ 见以下测试：

事务A	事务B
mysql> use powernode	
	mysql> use powernode
mysql> start transaction;	
	mysql> start transaction;
mysql> select * from a;	
id : 1 2 3 4	
	mysql> insert into a values(5);
	mysql> commit;

事务A	事务B
mysql> select * from a;	
id : 1 2 3 4	

通过以上测试得知：**当事务隔离级别设置为可重复读时，也避免了幻读问题。是完全避免了幻读问题吗？并不是。** 见以下测试：

事务A	事务B
mysql> use powernode	
	mysql> use powernode
mysql> start transaction;	
	mysql> start transaction;
mysql> select * from a; （快照读）	
id : 1	
2	
3	
4	
5	
	mysql> insert into a values(6);
	mysql> commit;
mysql> select * from a **for update;** （当前读）	
id : 1	
2	
3	
4	
5	

事务A	事务B
6 [使用当前读时幻读现象发生，快照读不会]	

通过以上测试得知：**当事务隔离级别设置为可重复读，MySQL会尽最大努力避免幻读问题，但这种隔离级别无法完全避免幻读问题。**

串行化

（SERIALIZABLE）

这种隔离级别最高，避免了所有的问题，**缺点是效率低**，因为这种隔离级别会**导致事务排队处理，不支持并发。**

设置数据库全局隔离级别为串行化：

```
set global transaction isolation level serializable;
```

演示：

事务A	事务B
mysql> use powernode	
	mysql> use powernode
mysql> start transaction;	
	mysql> start transaction;
mysql> select * from a;	
id : 1 2 3 4 5 6	
mysql> insert into a values(7);	
	mysql> select * from a;
mysql> select * from a;	
mysql> commit;	
	id : 1 2 3 4 5 6 7

通过以上测试得知：当事务隔离级别设置为串行化时，事务只能排队执行，不支持并发。

可重复读的幻读问题

MySQL默认的隔离级别可重复读，在很大程度上避免了幻读问题（并不能完全解决），那么它是如何解决幻读问题的呢，解决方案包括两种：

- 针对**快照读** Snapshot Read（普通 select 语句），是**通过 MVCC (Multi-Version Concurrency Control) 方式解决了幻读**，因为可重复读隔离级别下，事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，即使中途有其他事务插入了一条数据，是查询不出来这条数据的，所以就很好的避免了幻读问题。
- 针对**当前读** Current Read（select ... **for update** 等语句），是**通过 next-key lock（记录锁+间隙锁）方式解决了幻读**，因为当执行 select ... for update 语句的时候，会加上 next-key lock，如果有其他事务在 next-key lock 锁范围内插入了一条记录，那么这个插入语句就会被阻塞，无法成功插入，所以就很好的避免了幻读问题。

快照读如何解决幻读?

什么是快照读？**普通的select语句都是采用的快照读。**即在**整个事务的处理过程中，执行相同的一个select语句时，每次都是读取的快照。**（快照指的是固定的某个时刻的数据，就像现实世界中的拍照一样）。即，当事务隔离级别是可重复读，并且执行的select语句是一个普通的select语句时，都会采用快照读的方式读取数据，底层实现原理是：

- 底层由 MVCC（Multi-Version Concurrency Control，多版本并发控制）实现，实现的方式是**开始事务后，在执行第一个查询语句后，会创建一个 Read View，后续的查询语句利用这个 Read View**，通过这个 Read View 就可以在 undo log 版本链找到事务开始时的数据，所以事务过程中每次查询的数据都是一样的，**即使中途有其他事务插入了新纪录，是查询不出来这条数据**的，所以就很好的避免了幻读问题。

演示：

事务A	**事务B**
mysql> use powernode	
	mysql> use powernode
mysql> start transaction;	
	mysql> start transaction;

事务A	事务B
mysql> select * from a; //快照读	
id : 1 2 3 4	
	mysql> insert into a values(5);
	mysql> commit;
mysql> select * from a; //快照读	
id : 1 2 3 4	

当前读如何解决幻读?

当前读，即**每一次都读取最新的数据。**当前读包括：update、delete、insert、select...for update。因为增删改的时候都要基于最新的数据进行增删改。

> update、delete、insert 这几个DML语句执行前，MySQL都会默认进行一次当前读（底层会实现，表面看不见）。

而select...for update原理是：**对查询范围内的数据进行加锁，不允许其它事务对这个范围内的数据进行增删改。**也就是说**这个select语句范围内的数据是不允许并发的，只能排队执行**，从而避免幻读问题。

select...for update加的锁叫做：**next-key lock。**可以称其为：间隙锁 (gap lock)+ 记录锁 (Record Lock)**。间隙锁用来保证在锁定的范围内不允许insert操作。记录锁用来保证在锁定的范围内不允许delete和update操作。**

假如有这样的数据：

```
| id  |
|  1  |
|  2  |
|  4  |
```

SQL语句是这样写的：

```
select * from a where id between 2 and 4 for update;   # 当前读
```

那么id在 [2, 4] 区间的所有记录行被锁定，不能插入3 （3正好满足2-4的范围）是通过间隙锁来保证的。不能修改或删除2和4 是通过记录锁来搞定的。

演示：

事务A	事务B
mysql> use powernode	
	mysql> use powernode
mysql> start transaction;	
	mysql> start transaction;
mysql> select * from a where id between 2 and 4 for update; // 当前读	
	mysql> insert into a values(3); 卡住了，待A事务提交后才执行

出现幻读的两种情况

在**同一个事务处理过程中，如果前后两次都采用快照读，或者都采用当前读，则不会出现幻读问题。**如果第一次使用快照读，后面使用了当前读，则会出现幻读问题。

第一种产生幻读场景

A事务与B事务。在A事务中第一次查询使用快照读，B事务插入数据。然后在A事务中第二次查询使用当前读。则会产生幻读现象。
演示：

事务A	事务B
mysql> use powernode	
	mysql> use powernode
mysql> start transaction;	
	mysql> start transaction;
mysql> select * from a;	
id : 1 2 4	
	mysql> insert into a values(5);
	mysql> commit;

事务A	事务B
mysql> select * from a for update; // 产生了幻读	
id : 1 2 4 5	

第二种产生幻读场景

事务A与事务B，在事务A中**第一次查询使用快照读**，在事务B中插入一条数据，然后在**事务A中更新**事务B插入的那条记录，最后在事务A中**再次使用快照读。**则会发生幻读现象。

事务A	事务B
mysql> use powernode	
	mysql> use powernode
mysql> start transaction;	
	mysql> start transaction;
mysql> select * from a;	
id : 1 2 4 5	
	mysql> insert into a values(6);
	mysql> commit;
mysql> update a set id=100 where id=6; //主要是因为这个SQL语句的执行触发了当前读	
mysql> select * from a; // 产生了幻读	
id : 1 2 4 5 100	

总结可重复读幻读问题

MySQL的**可重复读隔离级别**（默认隔离级），根据不同的查询方式，分别提出了避免幻读的方案：

- 针对**快照读**（普通 select 语句），是通过 **MVCC** 方式解决了幻读。
- 针对**当前读**（select ... for update 等语句），是通过 **next-key lock**（记录锁+间隙锁）方式解决了幻读。

举例两个发生幻读场景的例子:

- 第一个例子：对于快照读， MVCC 并不能完全避免幻读现象。因为当事务 A 更新了一条事务 B 插入的记录，那么事务 A 前后两次查询的记录条目就不一样了，所以就发生幻读。
- 第二个例子：对于当前读，如果事务开启后，并没有执行当前读，而是先快照读，然后这期间如果其他事务插入了一条记录，那么事务后续使用当前读进行查询的时候，就会发现两次查询的记录条目就不一样了，所以就发生幻读。

所以，MySQL 可重复读隔离级别**并没有彻底解决幻读，只是很大程度上避免了幻读现象**的发生。

要避免这类特殊场景下发生幻读的现象的话，就是**尽量在开启事务之后，马上执行 select ... for update 这类当前读的语句**，因为它会对记录加 next-key lock，**从而避免其他事务插入一条新记录。**

> 直接用串行化会影响并发，一般不用。

第九章.DBA命令

> DBA (Database Administrator) 是负责管理和维护数据库系统的专业人员，他们负责确保数据库的安全性、可靠性和高性能运行。 以下命令在开发中较少使用，使用时可查看文档。

新建用户

> 以root身份登录时，mysql中有个数据库就叫 `mysql` ， 在 `use mysql;`后查看表，有个表名叫 `user`的里面存储着当前MySQL用户信息。

创建一个用户名为java1，密码设置为123的本地用户：

```
mysql -uroot -p123456     # 先以root登录，然后直接创建用户
create user 'java1'@'localhost' identified by '123';   # 该用户只能在本机使用
```

创建一个用户名为java2，密码设置为123的外网用户：

```
create user 'java2'@'%' identified by '123';        # 使用该用户可以在远程电脑登录并通过ip访问MySQL
```

```
mysql -ujava2 -p123 -h192.168.1.100
```

采用以上方式新建的用户没有任何权限：自带数据库也只能看到两个：

```
mysql -ujava1 -p123;
show databases;
/*  information_schema
    performance_schema  */
```

使用root用户查看系统中当前用户有哪些?

```
# 以root登录后直接：
select user, host from mysql.user;   # 数据库名.表名
# 或者先切换到系统数据库
use mysql;
select user, host from user;
```

给用户授权

授权语法：

```
grant [权限1，权限2...] on 库名.表名 to '用户名'@'主机名/IP地址';
```

给本地用户授权：grant [权限1，权限2...] on 库名.表名 to '用户名'@'localhost';

给外网用户授权：grant [权限1，权限2...] on 库名.表名 to '用户名'@'%';

所有权限：all privileges

细粒度权限：select、insert、delete、update、alter、create、drop、index(索引)、usage(登录权限)……

库名可以使用`*`，它代表所有数据库

表名可以采用`*`，它代表所有表

也可以提供具体的数据库和表，例如：powernode.emp （表示powernode数据库的emp表）

```
# 将所有库所有表的查询权限赋予本地用户java1
grant select,insert,delete,update,create on *.* to
'java1'@'localhost';

# 将powernode库中所有表的所有权限赋予本地用户java1
grant all privileges on powernode.* to 'java1'@'localhost';

flush privileges;   # 刷新权限
```

授权后必须刷新权限，才能生效.

如何查看某个用户拥有哪些权限?

```
show grants for 'java1'@'localhost'
show grants for 'java2'@'%'
```

with grant option：提供给其他用户授权的能力。

```
# with grant option的作用是：java2用户也可以给其他用户授权了。
grant select,insert,delete,update,reload on *.* to 'java2'@'%'
with grant option;
```

撤销用户权限

```
revoke 权限 on 数据库名.表名 from '用户'@'IP地址';
```

```
# 撤销本地用户java1的insert、update、delete权限
revoke insert, update, delete on powernode.* from
'java1'@'localhost'

# 撤销外网用户java2的insert权限
revoke insert on powernode.* from 'java2'@'%'

flush privileges;
```

撤销权限后也需要刷新权限。

注意：**撤销权限时 “数据库名.表名” 不能随便写，要求和授权语句时的 “数据库名.表名” 一致。**

修改用户的密码

具有管理用户权限的用户才能修改密码，例如root账户可以修改其他账户的密码：

```
# 本地用户修改密码
alter user 'java1'@'localhost' identified by '456';

# 外网用户修改密码
alter user 'java2'@'%' identified by '456';

flush privileges;
```

修改密码后，也需要刷新权限才能生效.

> 以上是MySQL 8 版本以后修改用户密码的方式。

修改用户名

```
rename user '原始用户名'@'localhost' to '新用户名'@'localhost';
rename user '原始用户名'@'localhost' to '新用户名'@'%';
rename user '原始用户名'@'%' to '新用户名'@'%';
rename user '原始用户名'@'%' to '新用户名'@'localhost';
```

```
rename user 'java1'@'localhost' to 'java11'@'localhost';
rename user 'java11'@'localhost' to 'java123'@'%';

# 记得刷新权限
flush privileges;
```

删除用户

```
drop user 'java123'@'localhost';
drop user 'java2'@'%';

flush privileges;
```

数据备份

- 导出数据（请在登录mysql数据库之前进行）

```
# 导出powernode这个数据库中所有的表
mysqldump powernode > e:/powernode.sql -uroot -p1234 --default-character-set=utf8

# 导出powernode中emp表的数据
mysqldump powernode emp > e:/powernode.sql -uroot -p1234 --default-character-set=utf8
```

- 导入数据第一种方式：（请在登录mysql之前进行）

```
# 先在登录mysql状态下新建一个数据库
create database powernode;
# 然后退出登录
exit;
# 在登录mysql之前执行以下命令
mysql powernode < e:/powernode.sql -uroot -p1234 --default-character-set=utf8
```

- 导入数据第二种方式：（请在登录mysql之后操作）

```
create  database powernode;
use powernode;
source d:/powernode.sql
```

第十章.客户端工具-Navicat

对于后端开发人员来说，一个好的MySQL客户端工具可以大大提升开发效率。目前企业中使用多的有：

- Navicat
- SQLyog
- MySQL Workbench　（MySQL自带安装）
- Datagrip　（JetBrain系列）

> 需要一个轻量级的数据库管理工具，且已使用 IntelliJ IDEA Ultimate 版本，那么内置的数据库管理工具可能已足够使用。

> 但如果是一个数据库管理员或需要进行更复杂的数据库操作，DataGrip是更合适的选择，因为它提供了更专业和全面的工具集

注意：像Navicat这些并不是数据库，只是一个客户端工具，方便你操作数据库中的数据。数据库并不在这。

MySQL的port是3306

使用Navicat：

新建connection，这里以本机演示：

Connection Name: localhost
Host: localhost
Port: 3306
User Name: root
Password: 123456

- Query：

  写的许多sql语句保存在 `.sql` 文件中，每个 `.sql` 脚本文件称作一个Query（查询）

- Backups（备份）：

  右键 new backup，选择要保存的数据；备份以Objects的形式存在。 通过 对象(Objects)可以提取 SQL。

  保存好的备份文件可以 `extract sql`，提取出 `.sql` 文件。

- 想再执行上述.sql文件：

  对其中一个数据库右键 execute SQL file，然后传入之前备份的 `.sql` 文件。

- 对现有表直接导出 .sql 文件：

  选择一个数据库，右键 Dump SQL File 。 （Dump - 转储）

- 新建表：

  对一个数据库右键 new table， （Name - 字段名， Type - 类型， VIRTUAL - 虚拟列，Key - 主键， Comment - 注释），

  上面点击 Add Field来添加字段。

  > 选中主键行，在下方框选 Auto Increment来使主键自动自增

  然后 ctrl + s保存。

- 设置外键：

从 Fields 切换到 Foreign Keys，里面的name是外键约束名，fields-给本表的哪个字段添加外键，Referenced Table - 外键要引用的哪张表，Referenced Fields - 要引用的外表字段。

- 查看表结构：对表右键 Design Table
- 对表插入数据：对表双击打开即可看到表内数据，下方有 + ， - 符号 来增删数据。√(确认)，×(取消)，刷新符号。
- 查询（Query）：

  首先 new query，然后可以选择在连接的哪个数据库上进行；记得 ctrl + s 保存成对象形式(实际以`.sql`文件存储)。创建的每个查询对象就是一个sql脚本文件。 一般就是在查询对象中编写 SQL 语句。（一个文件中可以写很多条SQL语句）

  写好后 点击 run 即可执行。 如果写了多条 select 语句，可以选中部分然后点击 run selected 来执行部分语句。 查询结果会分开展示。 点击 Beautify SQL 来格式化美化。

  > Navicat 的查询编辑器是一个多功能工具，它支持执行 DDL、DML、TCL 和 DCL等 SQL 语句。

```
/* 多行注释
*/
# 单行注释
```

- 新建数据库：

  对以IP或本机的连接服务器 右键 new database，（Character set - 字符集，通常选utf8mb4； Collation - 校对，通常选utf8mb4_general_ci）

  > `utf8mb4` 是 `UTF-8` 编码的超集，它支持最多 4 个字节的字符，这使得它能够存储任何 Unicode 字符。（`mb4` 指 “most bytes 4”，意味着最多使用 4 个字节来表示一个字符）
  >
  > `utf8mb4_general_ci` 表示使用 `utf8mb4` 字符集进行字符串存储，并且在校对（比较和排序）时不区分大小写

第十一章.企业真题

115-企业真题第一题*哔哩哔哩*bilibili

窗口函数

MySQL 8.0及以上版本中支持如下常用的窗口函数：

1. ROW_NUMBER()：排名函数，返回当前结果集中每个行的行号；
2. RANK()：排名函数，计算分组结果中的排名，相同的行排名相同且没有空缺，下一个行排名跳过空缺；
3. DENSE_RANK()：排名函数，计算分组结果中的排名，相同的行排名相同，排名连续，没有空缺；
4. NTILE()：将分组结果等分为指定的组数，计算每组的大小；
5. LAG()：返回分组内前一行的值；
6. LEAD()：返回分组内后一行的值；
7. FIRST_VALUE()：返回分组内第一个值；
8. LAST_VALUE()：返回分组内最后一个值；
9. AVG()、SUM()、COUNT()、MIN()、MAX()：聚合函数，可以配合OVER()进行窗口操作。

需要注意的是，MySQL的窗口函数和其他DBMS中的窗口函数相比较，可能略有不同，需要根据MySQL的文档进行使用。

第十二章.存储过程

什么是存储过程?

存储过程 （Stored Procedure）可称为**过程化SQL语言**，是在普通SQL语句的基础上增加了编程语言的特点，**把数据操作语句(DML)和查询语句(DQL)组织在过程化代码中**，通过**逻辑判断、循环等操作**实现复杂计算的程序语言。

换句话说，**存储过程其实就是数据库内置的一种编程语言，这种编程语言也有自己的变量、if语句、循环语句等。**在一个存储过程中可以将多条SQL语句以逻辑代码的方式将其串联起来，**执行这个存储过程就是将这些SQL语句按照一定的逻辑去执行**，所以**一个存储过程也可以看做是一组为了完成特定功能的SQL 语句集。**

每一个存储过程都是一个数据库对象，就像 table(表) 和 view(视图) 一样，**存储在数据库当中，一次编译永久有效。**并且**每一个存储过程都有自己的名字。客户端程序通过存储过程的名字来调用存储过程。**

在**数据量特别庞大的情况下利用存储过程**能达到倍速的**效率提升。**

> 不同数据库系统的存储过程在概念上是相似的，但它们在语法、功能和可用的编程语言方面可能不同。

存储过程的优点和缺点

优点：速度快。

- 降低了**应用服务器**和**数据库服务器**之间网络通讯的开销。尤其在数据量庞大的情况下效果显著。

缺点：移植性差。编写难度大。维护性差。

- 每一个数据库都有自己的存储过程的语法规则，这种语法规则不是通用的。一旦使用了存储过程，则数据库产品很难更换，例如：编写了mysql的存储过程，这段代码只能在mysql中运行，无法在oracle数据库中运行。
- 对于数据库存储过程这种语法来说，没有专业的IDE工具（集成开发环境），所以编码速度较低。自然维护的成本也会较高。

> 在实际开发中，存储过程还是很少使用的。只有在系统遇到了性能瓶颈，在进行优化的时候，对于大数量的应用来说，可以考虑使用一些。

第一个存储过程

存储过程的创建

```
create procedure p1()   /* p1是存储过程名 */
begin
    /* begin end里面写 SQL 语句集 */
    select empno,ename from emp;
end;
```

> 在navicat中执行后会以对象的形式存在 “Functions”中，可以把存储过程看作一种函数。

存储过程的调用

```
call p1();
```

> 直接在navicat里new query，然后在里面写这些语句即可。

存储过程的查看

查看创建存储过程的语句：

```
show create procedure p1;
```

通过系统表`information_schema.ROUTINES`查看存储过程的详细信息：（Routine - 例行程序）

```
select * from information_schema.routines where routine_name = 'p1';
```

> information_schema.ROUTINES 是 MySQL 数据库中一个系统表，存储了所有存储过程、函数、触发器的状态信息，包括名称、返回值类型、参数、创建时间、修改时间等。

information_schema.ROUTINES 表中的一些重要的列包括：

- SPECIFIC_NAME：存储过程的具体名称，包括该存储过程的名字，参数列表。
- ROUTINE_SCHEMA：存储过程所在的数据库名称。
- ROUTINE_NAME：存储过程的名称。
- ROUTINE_TYPE：PROCEDURE - 存储过程，FUNCTION - 函数，TRIGGER - 触发器。
- ROUTINE_DEFINITION：存储过程的创建定义语句。
- CREATED：存储过程的创建时间。
- LAST_ALTERED：存储过程的最后修改时间。
- DATA_TYPE：存储过程的返回值类型、参数类型等。

存储过程的删除

```
drop procedure if exists p1;
```

delimiter命令

在 MySQL 中，`delimiter` 命令用于改变 MySQL 解释语句的定界符。MySQL 默认使用分号 `;` 作为语句的定界符。而使用 `delimiter` 命令可以将分号 `;` 更改为其他字符，从而可以在 SQL 语句中使用分号 `;`。

> 适用于在DOS命令窗口中进行编写。

例如，使用DOS命令窗口想要创建一个存储过程。在存储过程中通常会包括多条 SQL 语句，而这些语句都需要以分号 `;` 结尾。但默认情况下，执行到第一条语句的分号 `;` 后，MySQL 就会停止解释，导致后续的语句无法执行。解决方式就是使用 `delimiter` 命令将分号改为其他字符，使分号 `;` 不再是语句定界符。例如：

```
delimiter //       /* 让MySQL在读到//(双斜杠)后才执行  */

CREATE PROCEDURE my_proc ()
BEGIN
    SELECT * FROM my_table;
    INSERT INTO my_table (col1, col2) VALUES ('value1',
'value2');
END;  //

delimiter ;
```

> 在这个例子中，我们使用 `delimiter //` 命令将定界符改为两个斜线 `//`。在存储过程中，以分号 `;` 结尾的语句不再被解释为语句的结束。而使用 `delimiter ;` 可以将分号恢复为语句定界符。

总之，`delimiter` 命令可以改变 MySQL 数据库系统中 SQL 查询语句的分隔符，从而可使一条 SQL 查询语句包含多个 SQL 语句。

MySQL的变量

mysql中的变量包括：系统变量、用户变量、局部变量。

系统变量

MySQL 系统变量是指**在 MySQL 服务器运行时控制其行为的参数**。这些变量可以被设置为特定的值来改变服务器的默认设置，以满足不同的需求。

MySQL 系统变量可以具有全局（global）或会话（session）作用域。

- **全局（global）**作用域是指**对所有连接和所有数据库都适用**；
- **会话（session）**作用域是指**只对当前连接和当前数据库**适用。

> 用一个用户名和密码登入，就是一个连接 connection。

1.查看系统变量

```
show [global|session] variables;            /*  直接查看系统变量 */

show [global|session] variables like '';    /*  模糊查找变量名的方式   */

select @@[global|session.]系统变量名;        /* 知道具体的变量名，直接查具体的。(两个@符号是固定写法，不要省略)  */
```

注意：没有指定session或global时，**默认是session。**

2.设置系统变量

```
set [global | session] 系统变量名 = 值;

set @@[global | session.]系统变量名 = 值;
```

注意：无论是全局设置还是会话设置，**当mysql服务重启之后，之前配置都会失效**（比如`net stop mysql`）。可以通过修改MySQL根目录下的**my.ini配置文件达到永久修改的效果。**（my.ini是MySQL数据库默认的系统级配置文件，默认是不存在的，需要新建，并参考一些资料进行配置）

> windows系统是my.ini
> linux系统是my.cnf

my.ini文件通常放在mysql安装的根目录下。这个文件通常是不存在的，可以新建；新建后例如提供以下配置：

```
[mysqld]
autocommit=0
```

这种配置就表示永久性关闭自动提交机制。（不建议这样做）

用户变量

用户自定义的变量。只在当前会话有效（用户名密码登入后exit，再次用户名密码登入就失效没有了）。所有的用户变量以‘@‘开始。

1.给用户变量赋值： 可以用`= 值`，也可以用`:= 值` （推荐用`:=`，因为在MySQL中 = 还有判断相等的意思）

```
set @name = 'jackson';
set @age := 30;
set @gender := '男', @addr := '北京大兴区';

select @email := 'jackson@123.com';  /* 定义并给变量赋值，同时直接读取值  */
select sal into @sal from emp where ename ='SMITH';   /* 查询出SIMITH的sal值并赋给@sal (动态地赋值) */
```

2.读取用户变量的值：

```
select @name, @age, @gender, @addr, @email, @sal;
```

注意：**mysql中用户变量不需要声明。**不需要类型限制，**直接赋值**就行。如果没有声明变量，直接读取该变量，返回null

局部变量

在存储过程中可以使用局部变量。**使用declare声明。只在存储过程begin和end之间有效。** 一个会话中可以创建多个存储过程。

1.变量的声明：　　(变量名不需要加 @ 符号)

```
declare 变量名 数据类型 [default ...];
```

变量的数据类型就是表字段的数据类型，例如：int、bigint、char、varchar、date、time、datetime等。

注意：declare通常出现在begin end之间的开始部分。

2.变量的赋值：

```
set 变量名 = 值;
set 变量名 := 值;
select 字段名 into 变量名 from 表名 ...;
```

例如：以下程序演示局部变量的声明、赋值、读取：

```
create PROCEDURE p2()
begin
    /*声明局部变量用来存储员工总人数*/
    declare emp_count int default 0;
    /*声明局部变量用来存储smith的工资*/
```

```
    declare sal double(10,2) default 0.0;

    /*给变量赋值 (动态)*/
    select count(*) into emp_count from emp;
    /*给变量赋值 (静态)*/
    set sal := 5000.0;

    /*读取变量的值*/
    select emp_count;
    /*读取变量的值*/
    select sal;
end;
```

```
call p2();   /* 调用存储过程 */
```

if语句

语法格式:

```
if 条件 then
......
elseif 条件 then
......
elseif 条件 then
......
else
......
end if;
```

案例：员工月薪sal，超过10000的属于“高收入”，6000到10000的属于“中收入”，少于6000的属于“低收入”。

```
create procedure p3(               )
begin
    declare sal int default 5000;
    declare grade varchar(20);

    if sal > 10000 then
    set grade := '高收入';
    elseif sal >= 6000 then
    set grade := '中收入';
    else
    set grade := '低收入';
```

```
    end if;

    select grade;
end;
```

```
call p3();
```

存储过程的参数

存储过程的参数包括三种形式： （这些参数都是变量）

- `in 变量名 数据类型`：入参（未指定时，默认是in）
- `out 变量名`：出参
- `inout 变量名`：既是入参，又是出参

```
// 以Java举例来理解出入参
public int sum(int a int b){  // a、b变量就是入参 in
    int c = a + b;
     return c;  // c变量就是出参 out
}
```

案例：员工月薪sal，超过10000的属于“高收入”，6000到10000的属于“中收入”，少于6000的属于“低收入”。

```
/*
in修饰的变量是入参，用来接收调用者传过来的数据。
out修饰的变量是出参，用来保存整个存储过程的执行结果。
inout修饰的变量既是入参，又是出参。
*/
create procedure p4(in sal int, out grade varchar(20))
                /* in 变量名 数据类型,       */
begin
    if sal > 10000 then
        set grade := '高收入';
    elseif sal >= 6000 then
        set grade := '中收入';
    else
        set grade := '低收入';
    end if;
end;
```

```
call p4(5000, @grade);  /* 调用存储过程 */
/* 因为在存储过程外面只能访问到用户变量，所以执行结果的参数是用户变量  */
select @grade;
```

案例：将传入的工资sal上调10%

```
create procedure p5(inout sal int)
begin
    set sal := sal * 1.1;
end;
```

```
set @sal := 10000;
call p5(@sal);
select @sal;
```

case语句

语法格式：

```
/*  值匹配  */
case 值
    when 值1 then
    ......
    when 值2 then
    ......
    when 值3 then
    ......
    else
    ......
end case;
```

```
/*  判断条件   */
case
    when 条件1 then
    ......
    when 条件2 then
    ......
    when 条件3 then
    ......
    else
    ......
end case;
```

案例：根据不同月份，输出不同的季节。3 4 5月份春季。6 7 8月份夏季。9 10 11月份秋季。12 1 2 冬季。其他非法。

```
create procedure mypro(in month int, out result varchar(100))
begin
    case month
        when 3 then set result := '春季';
        when 4 then set result := '春季';
        when 5 then set result := '春季';
        when 6 then set result := '夏季';
        when 7 then set result := '夏季';
        when 8 then set result := '夏季';
        when 9 then set result := '秋季';
        when 10 then set result := '秋季';
        when 11 then set result := '秋季';
        when 12 then set result := '冬季';
        when 1 then set result := '冬季';
        when 2 then set result := '冬季';
        else set result := '非法月份';
    end case;
end;
```

```
create procedure mypro(in month int, out result varchar(100))
begin
    case
        when month = 3 or month = 4 or month = 5 then
            set result := '春季';
        when  month = 6 or month = 7 or month = 8  then
            set result := '夏季';
        when  month = 9 or month = 10 or month = 11  then
            set result := '秋季';
        when  month = 12 or month = 1 or month = 2  then
            set result := '冬季';
```

```
        else
            set result := '非法月份';
    end case;
end;
```

```
call mypro(9, @season);
select @season;
```

while循环

语法格式：

```
/*  条件成立就执行循环  */
while 条件 do
    循环体;
end while;
```

案例：传入一个数字n，计算1~n中所有偶数的和。

```
create procedure mypro(in n int)
begin
    declare sum int default 0;   /*  局部变量  */
    while n > 0 do
        if n % 2 = 0 then
            set sum := sum + n;
        end if;
        set n := n - 1;
    end while;
    select sum;
end;
```

```
call mypro(10);
```

repeat循环

语法格式：

```
/*  条件成立就结束循环  */
repeat
    循环体;
    until 条件    /*  注意：这里不要用分号';' 做结尾 */
end repeat;
```

注意：**条件成立时结束循环。**

案例：传入一个数字n，计算1~n中所有偶数的和。

```
create procedure mypro(in n int, out sum int)
begin
    set sum := 0;
    repeat
        if n % 2 = 0 then
          set sum := sum + n;
        end if;
        set n := n - 1;
        until n <= 0     /*  没有分号';'  */
    end repeat;
end;
```

```
call mypro(10, @sum);
select @sum;
```

loop循环

语法格式：

```
create procedure mypro()
begin
    declare i int default 0;
    /*  可以给循环起名  */
  mylp:loop    /*  loop开始，end loop结束 */
        set i := i + 1;
        if i = 5 then
            leave mylp;  /* 用 `leave 循环名` 来结束循环  */
        end if;
        select i;
    end loop;
end;
```

```
create procedure mypro()
begin
    declare i int default 0;
  mylp:loop
        set i := i + 1;
        if i = 5 then
            iterate mylp;   /* iterate 类似continue，表示结束本次循环，下面代码不再执行；直接跳入下一次的循环  */
        end if;
        if i = 10 then
          leave mylp;
        end if;
        select i;
    end loop;
end;
```

> leave 与 iterate 的区别就类似于常规编程语言中的`break`与`continue`。

游标cursor

游标（cursor）**可以理解为一个指向结果集中某条记录的指针，允许程序逐一访问结果集中的每条记录，并对其进行逐行操作和处理。**

使用游标时，需要在存储过程或函数中定义一个游标变量，并通过`DECLARE`语句进行声明和初始化。然后，使用`OPEN`语句打开游标，使用`FETCH`语句逐行获取游标指向的记录，并进行处理。最后，使用`CLOSE`语句关闭游标，释放相关资源。

游标可以大大地提高数据库查询的灵活性和效率。

1.声明游标的语法：

```
declare 游标名称 cursor for 查询语句;
```

2.打开游标的语法：

```
open 游标名称;
```

3.通过游标取数据并存放到变量里的语法：

```
fetch 游标名称 into 变量[,变量,变量.....]
```

4.关闭游标的语法：

```
close 游标名称;
```

案例：从dept表查询部门编号和部门名，创建一张新表dept2，将查询结果插入到新表中。

```
drop procedure if exists mypro;

create procedure mypro()
begin
    /*  注意：局部变量的声明需要在游标声明之前完成   */
    declare no int;
    declare name varchar(100);

    /*  声明游标：dept_cursor游标指向了 select deptno,dname from dept的结果集 */
    declare dept_cursor cursor for select deptno,dname from dept;

    drop table if exists dept2;
    create table dept2(
        no int primary key,
        name varchar(100)
    );

    open dept_cursor;   /*  打开游标  */

    while true do
        fetch dept_cursor into no, name;    /* 通过游标取数据，并存放到变量 no、name里 */
        /* 执行一次fetch，游标就会自动下移一行，指向新的记录  */
        insert into dept2(no,name) values(no,name);
    end while;

    close dept_cursor;    /* 关闭游标，释放资源  */
end;

call mypro();
```

执行结果：

```
1329-No data -zero rowsfetched,selected,orprocessed
```

出现了异常：异常信息中显示没有数据了。这是因为while true循环导致的。

不过虽然出现了异常，但是表创建成功了，数据也插入成功了：

```
no   |  name
10   |  ACCOUNTING
20   |  RESEARCH
30   |  SALES
40   |  OPERATIONS
50   |  销售部
```

注意：声明局部变量和声明游标有顺序要求，局部变量的声明需要在游标声明之前完成。

捕捉异常并处理

语法格式：

```
DECLARE handler_name HANDLER FOR condition_value
action_statement
        /* 可填值 ↑ */           /* 可填值 ↑ */    /* 可填值 ↑ */
```

1. handler_name 表示异常处理程序的名称，取值选择包括：
   - `CONTINUE`：发生异常后，程序不会终止，**会正常执行后续的过程。**(捕捉) [类似Java的try..catch...]
   - `EXIT`：发生异常后，**终止存储过程的执行。**（上抛） [类似Java的throws]
2. condition_value 是指捕获的异常，取值选择包括：
   - `SQLSTATE sqlstate_value`，例如：`SQLSTATE '02000'`（sqlstate_value 就是SQL状态码）
   - `SQLWARNING`，代表所有01开头的SQLSTATE
   - `NOT FOUND`，代表所有02开头的SQLSTATE
   - `SQLEXCEPTION`，代表除了01和02开头的所有SQLSTATE
3. action_statement 是指异常发生时要执行的SQL语句，例如：CLOSE cursor_name

案例：给之前的游标添加异常处理机制

```
drop procedure if exists mypro;

create procedure mypro()
```

```
begin

    declare no int;
    declare name varchar(100);
    declare dept_cursor cursor for select deptno,dname from
dept;

    /* 通常在这个位置进行异常的处理  */
    /*  如果发生的异常是NOT FOUND则关闭游标，并且退出存储过程  */
    declare exit handler for not found close dept_cursor;

    drop table if exists dept2;
    create table dept2(
        no int primary key,
        name varchar(100)
    );

    open dept_cursor;

    while true do
        fetch dept_cursor into no, name;
        insert into dept2(no,name) values(no,name);
    end while;

    close dept_cursor;
end;

call mypro();
```

存储函数

存储函数(Stored Function)：**带返回值的存储过程。**参数只允许是in（但不能显式的写in）。**没有out，也没有inout。**

> 存储函数、存储过程都是数据库对象。

语法格式：

```
CREATE FUNCTION 存储函数名(参数列表) RETURNS 数据类型 [特征]
BEGIN
    --函数体
    RETURN ...;
END;
```

“特征”的可取重要值如下：

- `deterministic`：用该特征**标记该函数为确定性函数。**这是一种**优化策略**，这种情况下整个**函数体的执行就会省略**了，**直接返回之前缓存的结果，来提高函数的执行效率。**

  > 什么是确定性函数？ **每次调用函数时传同一个参数的时候，返回值都是固定的。**

- `no sql`：用该特征**标记该函数执行过程中不会查询数据库**，如果确实没有查询语句建议使用。告诉 MySQL **优化器不需要考虑使用查询缓存和优化器缓存来优化**这个函数，这样就可以**避免不必要的查询消耗产生，从而提高性能。**
- `reads sql data`：用该特征**标记该函数会进行查询操作**，**告诉 MySQL 优化器**这个函数需要查询数据库的数据，**可以使用查询缓存来缓存结果，从而提高查询性能**；同时 MySQL 还会**针对该函数的查询进行优化器缓存处理。**

案例：计算1 ~ n的所有偶数之和

```
-- 删除函数
drop function if exists sum_fun;

-- 创建函数
create function sum_fun(n int) returns int deterministic
begin
    declare result int default 0;  /* 定义局部变量 */
    while n > 0 do
        if n % 2 = 0 then
            set result := result + n;
        end if;
        set n := n - 1;
    end while;
    return result;   /*  返回值  */
end;

-- 调用函数
set @result = sum_fun(100);  /* 使用用户变量接收函数返回值  */
select @result;
```

触发器 (Trigger)

MySQL 触发器（Trigger）是一种数据库对象，**它是与表相关联的特殊程序** (不需要手动调用)。它**可以在特定的数据操作**（例如插入（INSERT）、更新（UPDATE）或删除（DELETE））**触发时自动执行。**MySQL 触发器**使数据库开发人员能够在数据的不同状态之间维护一致性和完整性，并且可以为特定的数据库表自动执行操作。**

触发器的作用主要有以下几个方面：

1. 强制实施业务规则：触发器可以帮助**确保数据表中的业务规则得到强制执行**，例如检查插入或更新的数据是否符合某些规则。
2. **数据审计**：触发器可以声明在执行**数据修改时自动记日志或审计数据变化**的操作，使数据对数据库管理员和 SQL 审计人员更易于追踪和审计。
3. 执行特定业务操作：触发器可以**自动执行特定的业务操作**，例如计算数据行的总数、计算平均值或总和等。

MySQL 触发器分为两种类型: BEFORE 和 AFTER。

- BEFORE 触发器在执行 INSERT、UPDATE、DELETE (增删改) 语句**之前执行。**
- AFTER 触发器在执行 INSERT、UPDATE、DELETE (增删改) 语句**之后执行。**

创建触发器的语法如下：

```
CREATE TRIGGER trigger_name
BEFORE/AFTER INSERT/UPDATE/DELETE ON table_name FOR EACH ROW
BEGIN
-- 触发器要执行的 SQL 语句
END;
```

其中：

- trigger_name：触发器的名称
- BEFORE/AFTER：触发器的类型，可以是 BEFORE 或者 AFTER
- INSERT/UPDATE/DELETE：触发器所监控的 DML 调用类型
- table_name：触发器所绑定的表名
- FOR EACH ROW：行级触发器，修改一行数据触发一次。表示触发器在每行受到 DML 的影响之后都会执行。

  > 如果不写FOR EACH ROW 就默认是 语句级触发器；即不管修改多少行数据，只执行一次。

- 触发器执行的 SQL 语句：该语句会在触发器被触发时执行

注意：触发器是一种高级的数据库功能，只有在必要的情况下才应该使用，例如在需要实施强制性业务规则时。**过多的触发器和复杂的触发器逻辑可能会影响查询性能和扩展性。**

关于触发器的NEW和OLD关键字：

在 MySQL 触发器中，NEW 和 OLD 是两个特殊的关键字，用于引用在触发器中受到修改的行的新值和旧值。具体而言：

- NEW：在触发 INSERT 或 UPDATE 操作期间，**NEW 用于引用将要插入或更新到表中的新行的值。**
- OLD：在触发 UPDATE 或 DELETE 操作期间，**OLD 用于引用**更新或删除**之前在表中的旧行的值。**

通俗的讲，NEW 是指触发器执行的操作所要插入或更新到当前行中的新数据；而 OLD 则是指当前行在触发器执行前原本的数据。

在MySQL 触发器中，NEW 和 OLD 使用方法是相似的。在触发器中，可以像引用表的其他列一样引用 NEW 和 OLD。例如，可以使用 OLD.column_name 从旧行中引用列值，也可以使用 NEW.column_name 从新行中引用列值。

示例：假设有一个名为 my_table 的表，其中包含一个名为 quantity 的列。当在该表上执行 UPDATE 操作时，以下触发器会将旧值 OLD.quantity 累加到新值 NEW.quantity 中：

```
CREATE TRIGGER my_trigger
BEFORE UPDATE ON my_table
FOR EACH ROW      /*  这里没分号';' */
BEGIN
    SET NEW.quantity = NEW.quantity + OLD.quantity;
END;
```

> 在此触发器中，OLD.quantity 引用原始行的 quantity 值（旧值），而 NEW.quantity 引用更新行的 quantity 值（新值）。在触发器执行期间，数据行的 quantity 值将设置为旧值加上新值。

注意：在使用 NEW 和 OLD 时，需要根据 DML 操作的类型进行判断，以确定哪个关键字表示新值，哪个关键字则表示旧值。

案例：当我们对dept表中的数据进行insert delete update的时候，请将这些操作记录到日志表当中，日志表如下：

```
drop table if exists oper_log;

create table oper_log(
  id bigint primary key auto_increment,
  table_name varchar(100) not null comment '操作的哪张表',
  oper_type varchar(100) not null comment '操作类型包括insert delete update',
  oper_time datetime not null comment '操作时间',
  oper_id bigint not null comment '操作的那行记录的id',
  oper_desc text comment '操作描述'
);
```

触发器1：向dept表中插入数据时，记录日志

```
create trigger dept_trigger_insert
after insert on dept
for each row
begin
    insert into oper_log(id,table_name,oper_type,oper_time,oper_id,oper_desc) values
(null,'dept','insert',now(),new.deptno,concat('插入数据:deptno=', new.deptno, ',dname=', new.dname,',loc=', new.loc));
     /*   concat()做字符串拼接   */
end;
```

1.查看触发器：

```
show triggers;
```

2.删除触发器：

```
drop trigger if exists dept_trigger_insert;
```

触发器2：修改dept表中数据时，记录日志

```
create trigger dept_trigger_update
after update on dept
for each row     /*  行级触发器  */
begin
    insert into
oper_log(id,table_name,oper_type,oper_time,oper_id,oper_desc)
values
(null,'dept','update',now(),new.deptno,concat('更新前：deptno=',
old.deptno, ',dname=', old.dname,',loc=', old.loc,
                                                ',更新后：
deptno=', new.deptno, ',dname=', new.dname,',loc=', new.loc));
end;
```

更新一条记录：

```
update dept set loc = '北京' where deptno = 60;
```

注意：更新一条记录则对应一条日志。如果一次更新3条记录，那么日志表中插入3条记录。（因为采用的行级触发器）

触发器3：删除dept表中数据时，记录日志

```
create trigger dept_trigger_delete
after delete on dept
for each row
begin
    insert into
oper_log(id,table_name,oper_type,oper_time,oper_id,oper_desc)
values
(null,'dept','delete',now(),old.deptno,concat('删除了数据：
deptno=', old.deptno, ',dname=', old.dname,',loc=', old.loc));
end;
```

删除一条记录：

```
delete from dept where deptno = 60;
```

第十三章.存储引擎

存储引擎概述

存储引擎（Storage Engine）**决定了数据在磁盘上的存储方式和访问方式。**不同的存储引擎实现了**不同的存储和检索算法**，因此它们在处理和管理数据的方式上存在差异。

MySQL常见的存储引擎包括 InnoDB、MyISAM、Memory、Archive等。**每个存储引擎都有自己的特点和适用场景。**

例如，

- InnoDB引擎**支持事务和行级锁定**，适用于**需要高并发读写**的应用；（for update可以看作行级锁）
- MyISAM引擎不支持事务，但**适用于读操作较多**的应用；
- Memory引擎数据全部存储在内存中，适用于对**读写速度要求很高**的应用等等。（不支持事务，且如果系统宕机后重启数据就丢失）

选择适合的存储引擎可以提高MySQL的性能和效率，并且根据应用需求来合理选择存储引擎可以提供更好的数据管理和查询功能。

MySQL支持哪些存储引擎

使用 `show engines \G;` 命令可以查看所有的存储引擎：

```
*************************** 1. row ***************************
      Engine: MEMORY
     Support: YES
     Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 2. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
```

```
*************************** 3. row ***************************
      Engine: CSV
     Support: YES
     Comment: CSV storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 4. row ***************************
      Engine: FEDERATED
     Support: NO       // no 就表示当前版本的mysql并不支持这个引擎
     Comment: Federated MySQL storage engine
Transactions: NULL
          XA: NULL
  Savepoints: NULL
*************************** 5. row ***************************
      Engine: PERFORMANCE_SCHEMA
     Support: YES
     Comment: Performance Schema
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 6. row ***************************
      Engine: MyISAM
     Support: YES
     Comment: MyISAM storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 7. row ***************************
      Engine: InnoDB
     Support: DEFAULT
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
*************************** 8. row ***************************
      Engine: ndbinfo
     Support: NO
     Comment: MySQL Cluster system information storage engine
Transactions: NULL
```

```
           XA: NULL
  Savepoints: NULL
*************************** 9. row ***************************
      Engine: BLACKHOLE
     Support: YES
     Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
           XA: NO
  Savepoints: NO
*************************** 10. row ***************************
      Engine: ARCHIVE
     Support: YES
     Comment: Archive storage engine
Transactions: NO
           XA: NO
  Savepoints: NO
*************************** 11. row ***************************
      Engine: ndbcluster
     Support: NO
     Comment: Clustered, fault-tolerant tables
Transactions: NULL
           XA: NULL
  Savepoints: NULL
```

`Support`是`Yes`的表示支持该存储引擎。当前MySQL的版本是`8.0.33`

MySQL 8 默认的存储引擎是：`InnoDB`

指定和修改存储引擎

指定存储引擎

在MySQL中，你可以在创建表时指定使用的存储引擎。通过在CREATE TABLE语句中使用`ENGINE`关键字，你可以指定要使用的存储引擎。

以下是指定存储引擎的示例：

```sql
CREATE TABLE my_table (column1 INT, column2 VARCHAR(50))
ENGINE = InnoDB;
```

在这个例子中，我们创建了一个名为my_table的表，并指定了使用InnoDB存储引擎。

如果你不显式指定存储引擎，MySQL将使用默认的存储引擎。默认情况下，MySQL 8的默认存储引擎是InnoDB。

修改存储引擎

在MySQL中，你可以通过ALTER TABLE语句修改表的存储引擎。下面是修改存储引擎的示例：

```
ALTER TABLE my_table ENGINE = MyISAM;
```

在这个例子中，我们使用ALTER TABLE语句将my_table表的存储引擎修改为MyISAM。

注意，在修改存储引擎之前，需要考虑以下几点：

1. 修改存储引擎可能需要执行复制表的操作，因此可能会造成数据的丢失或不可用。确保在执行修改之前备份你的数据。
2. 不是所有的存储引擎都支持相同的功能。要确保你选择的新存储引擎支持你应用程序所需的功能。
3. 修改表的存储引擎可能会影响到现有的应用程序和查询。确保在修改之前评估和测试所有的影响。
4. ALTER TABLE语句可能需要适当的权限才能执行。确保你拥有足够的权限来执行修改存储引擎的操作。

总而言之，**修改存储引擎需要谨慎进行，且需要考虑到可能的影响和风险。**建议在进行修改之前**进行适当的测试和备份。**

常用的存储引擎及适用场景

在实际开发中，以下存储引擎是比较常用的：（了解，要知道有这些引擎)

1. InnoDB：
    1. MySQL**默认的事务型存储引擎**
    2. **支持ACID事务**
    3. 具有**较好的并发性能和数据完整性**
    4. 支持**行级锁定。**
    5. 适用于大多数应用场景，**尤其是需要事务支持**的应用。

2. MyISAM：
    1. 是MySQL早期版本中常用的存储引擎
    2. 支持全文索引和表级锁定
    3. 不支持事务
    4. 由于其简单性和高性能，在某些特定的应用场景中会得到广泛应用，如**读密集**的应用。
3. MEMORY：
    1. 称为HEAP，是将**表存储在内存中**的存储引擎
    2. 具有**非常高的读写性能**，但**数据会在服务器重启时丢失。**
    3. 适用于**需要快速读写的临时数据集、缓存和临时表**等场景。
4. CSV：
    1. 将**数据以纯文本格式存储**的存储引擎
    2. 适用于**需要处理和导入/导出CSV格式数据**的场景。
5. ARCHIVE： /ˈɑːkaɪv/
    1. 将**数据高效地进行压缩和存储**的存储引擎
    2. 适用于**需要长期存储大量历史数据且不经常查询**的场景。

第十四章.索引 (index)

什么是索引

索引是**一种能够提高检索（查询）效率的提前排好序的数据结构。**例如：书的目录就是一种索引机制。**索引是解决SQL慢查询**的一种方式。　　在数据库中以对象的形式存在。

> MySQL中有很多对象：
>
> - table对象
> - view对象，
> - index对象
> - procedure存储过程对象
> - function函数对象
> - trigger触发器对象
> - constraint约束对象

索引的创建和删除

主键会自动添加索引

主键字段会自动添加索引，不需要程序员干涉，主键字段上的索引被称为：`主键索引`

unique约束的字段自动添加索引

unique约束的字段也会自动添加索引，不需要程序员干涉，这种字段上添加的索引称为：`唯一索引`

给指定的字段添加索引

1.建表时添加索引：

```
CREATE TABLE emp (
    /* ...  */
    name varchar(255),
    /* ...  */
    INDEX idx_name (name)  /* index 索引名 (要索引的表中字段)  */
);

```

2.如果表已经创建好了，后期给字段添加索引

```
ALTER TABLE emp ADD INDEX idx_name (name);
```

3.也可以这样添加索引：

```
create index idx_name on emp(name);
```

删除指定字段上的索引

```
ALTER TABLE emp DROP INDEX idx_name;
```

查看某张表上添加了哪些索引

```
show index from 表名;
```

索引的分类

不同的`存储引擎`有不同的索引类型和实现：

- 按照数据结构分类：
  - **B+树 索引**（mysql的InnoDB存储引擎采用的就是这种索引）采用 B+树 的数据结构
  - Hash 索引（仅 `memory` 存储引擎支持）：采用 哈希表 的数据结构
- **按照物理存储分类**：
  - 聚集索引：**索引和表中数据在一起**，数据存储的时候就是**按照索引顺序存储**的。**一张表只能有一个聚集索引。**
  - 非聚集索引：索引和表中数据是分开的，**索引是独立于表空间**的，一张表可以有多个非聚集索引。
- 按照字段特性分类：
  - 主键索引（primary key）
  - 唯一索引（unique）
  - 普通索引（index）
  - 全文索引（fulltext：仅 `InnoDB` 和 `MyISAM` 存储引擎支持）
- 按照字段个数分类：
  - 单列索引
  - 联合索引（也叫复合索引、组合索引）

MySQL索引采用的B+树数据结构

常见的树相关的数据结构包括：

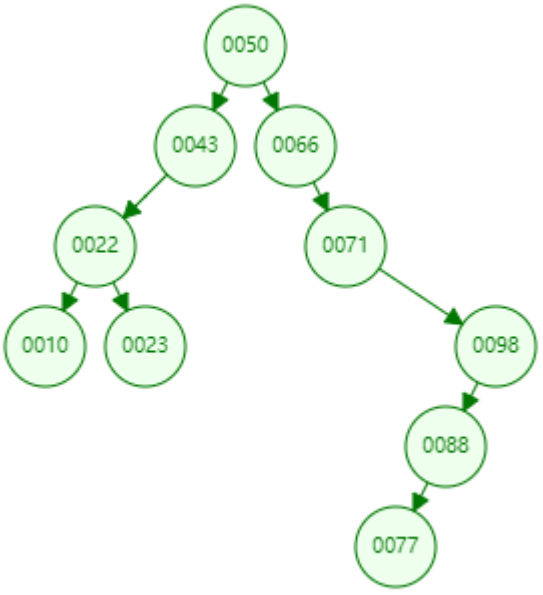
- 二叉树
- 红黑树
- B树
- B+树

区别：**树的高度不同。**树的**高度越低，性能越高。**这是因为**每一个节点都是一次 I/O**

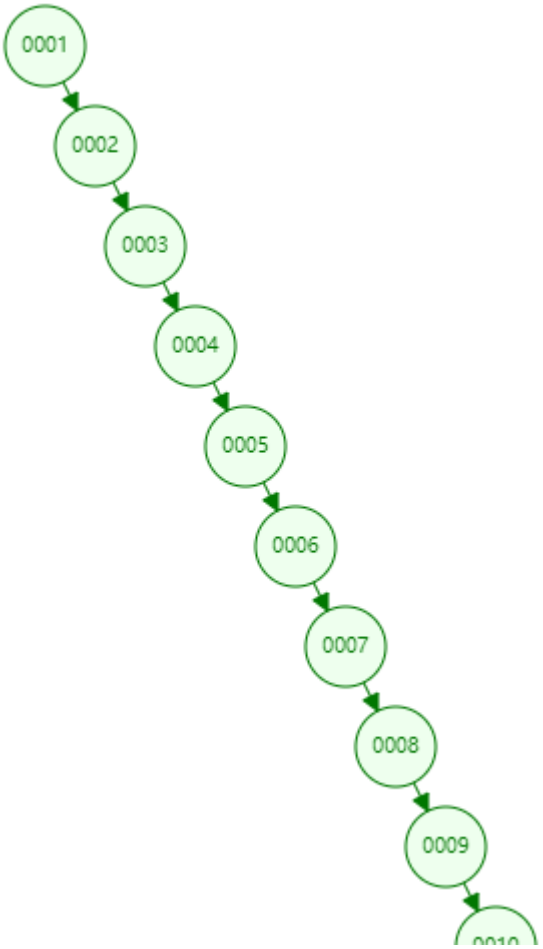
二叉树

如果不给id字段添加索引，默认进行全表扫描，假设查询id=10的数据，那至少要进行10次磁盘IO。效率低。可以给id字段添加索引，假设该索引使用了二叉树这种数据结构，这时IO次数显著减少，效率显著提升了。（默认左小右大，按照中序遍历[Inorder Traversal -左根右]就可以得到从小到大的数据）

> 推荐一个数据结构可视化网站Data Structure Visualizations：
> https://www.cs.usfca.edu/~galles/visualization/Algorithms.html



但是MySQL并没有直接使用这种普通的二叉树，因为这种普通二叉树在`数据极端`（如刚好原始数据按从小到大排列）的情况下，出现树的形状全在一边，类似链表的情况，此时查找效率等同于链表查询O(n)，查找效率极低。

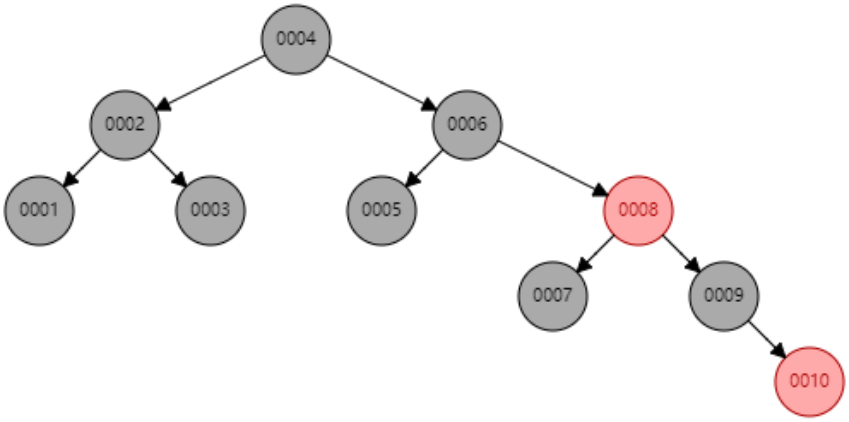


红黑树（自平衡二叉树）

通过自旋平衡规则进行旋转，子节点会自动分叉为2个分支，从而减少树的高度，当**数据有序插入时比二叉树数据检索性能更好。**

但是如果数据量庞大，例如500万条数据，也会导致树的高度很高，磁盘IO次数仍然很多，查询效率也会比较低。

因此MySQL并没有使用红黑树这种数据结构作为索引。



B Trees（B树）

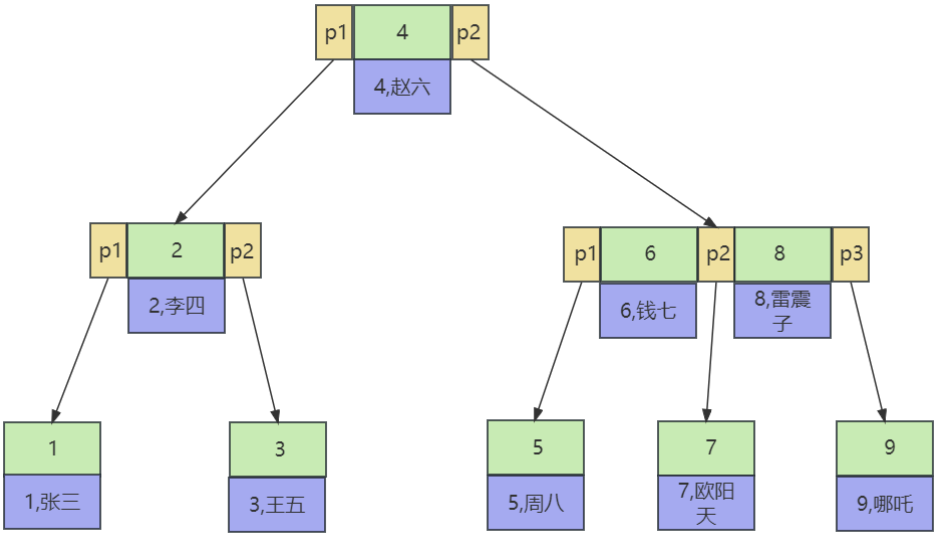
B Trees首先是一个**自平衡**的。
B Trees每个节点下的子节点数量 > 2。
B Trees每个节点中也不是存储单个数据，可以存储多个数据。
B Trees又称为`平衡多路查找树`。

B Trees分支的数量不是2，是大于2，具体是多少个分支，由`阶`决定。例如：

- 3阶的B Trees，一个节点下最多有3个子节点，每个节点中最多有2个数据。
- 4阶的B Trees，一个节点下最多有4个子节点，每个节点中最多有3个数据。
- 5阶（5, 4）
- 6阶（6, 5）
- ....
- 16阶（16, 15）**【MySQL采用了16阶】**

采用B Trees，会发现相同的数据量，**B Tree 树的高度更低**。磁盘IO次数更少。

在B Trees中，每个节点不仅存储了`索引值`，还存储该索引值对应的`数据行`。
并且每个节点中的p1 p2 p3是指向下一个节点的指针。



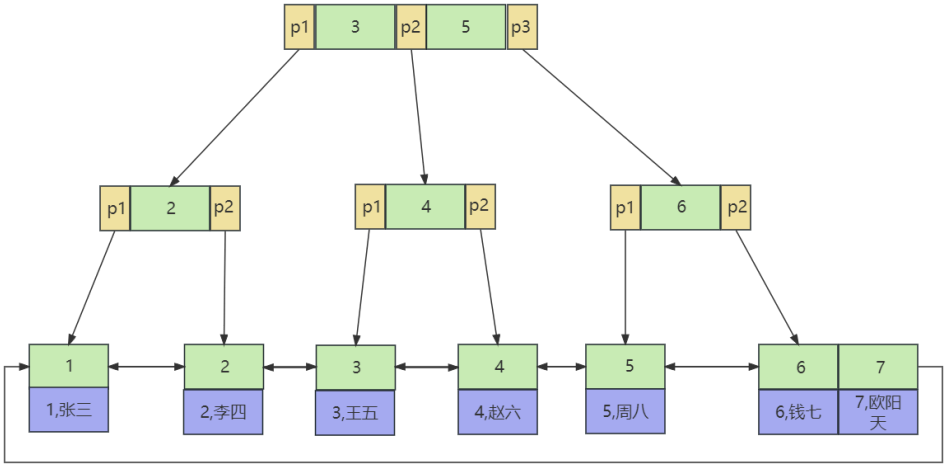
B Trees数据结构存在的**缺点是：不适合做区间查找，对于区间查找效率较低。**假设要查id在[3~7]之间的，需要查找的是3,4,5,6,7。那么查这每个索引值都需要从根节点开始（不能直接从3到5）。

因此MySQL使用了B+ Trees解决了这个问题。

B+ Trees（B+ 树）

B+ Trees 相较于 B Trees改进了哪些?

- B+树**将数据都存储在叶子节点**中。并且**叶子节点之间使用双向链表连接**，这样很**适合范围查询。**
- B+树的**非叶子节点上只有索引值，没有数据**，所以非叶子节点可以存储更多的索引值，这样**让B+树更矮更胖，提高检索效率。**



> 注意叶子节点之间使用双向链表连接

经典面试题

①mysql为什么选择B+树作为索引的数据结构，而不是B树？

1. **非叶子节点上可以存储更多的键值，阶数可以更大**，更矮更胖，磁盘IO次数少，数据查询效率高。
2. 所有数据都是**有序存储在叶子节点上，让范围查找，分组查找效率更高。**
3. **数据页之间、数据记录之间采用链表链接**，让升序降序更加方便操作。

②如果一张表没有主键索引，那还会创建B+树吗？

当**一张表没有主键索引时，默认会使用一个隐藏的内置的聚集索引**（clustered index）。**这个聚集索引**是基于表的物理存储顺序构建的，**通常是使用B+树来实现**的。

其他索引及相关调优

Hash索引

支持Hash索引的存储引擎有：

- InnoDB（不支持手动创建Hash索引，系统会自动维护一个`自适应的Hash索引`）
  - 对于InnoDB来说，即使手动指定了某字段采用Hash索引，最终`show index from 表名`的时候，还是`BTREE`。
- Memory（支持Hash索引）

Hash索引底层的数据结构就是哈希表。一个数组，数组中每个元素是链表。和java中HashMap一样。哈希表中每个元素都是key value结构。key存储`索引值`，value存储`行指针`。

```
/* 查询时  */
select * from t_user where name = '孙行者';
```

检索原理：假设 name='孙行者'。通过哈希算法将'孙行者'转换为数组下标，通过下标找链表，在链表上遍历找到孙行者的行指针。

注意：不同的字符串，经过哈希算法得到的数组下标可能相同，这叫做哈希碰撞/哈希冲突。【不过，好的哈希算法应该具有很低的碰撞概率。常用的哈希算法如MD5、SHA-1、SHA-256等都被设计为尽可能减少碰撞的发生。】

Hash索引优缺点：

- 优点：**只能用在等值比较中，效率很高。**例如：name='孙悟空'
- 缺点：**不支持排序，不支持范围查找。**

聚集索引和非聚集索引

按照数据的物理存储方式不同，可以将索引分为聚集索引（聚簇索引）和非聚集索引（非聚簇索引）。

存储引擎是**InnoDB的，主键上的索引属于聚集索引。**
存储引擎是MyISAM的，任意字段上的索引都是非聚集索引。

InnoDB的物理存储方式：当创建一张表t_user，并使用InnoDB存储引擎时，会在硬盘上生成这样一个文件：

- t_user.ibd （InnoDB data表索引 + 数据）
- t_user.frm （存储表结构信息）

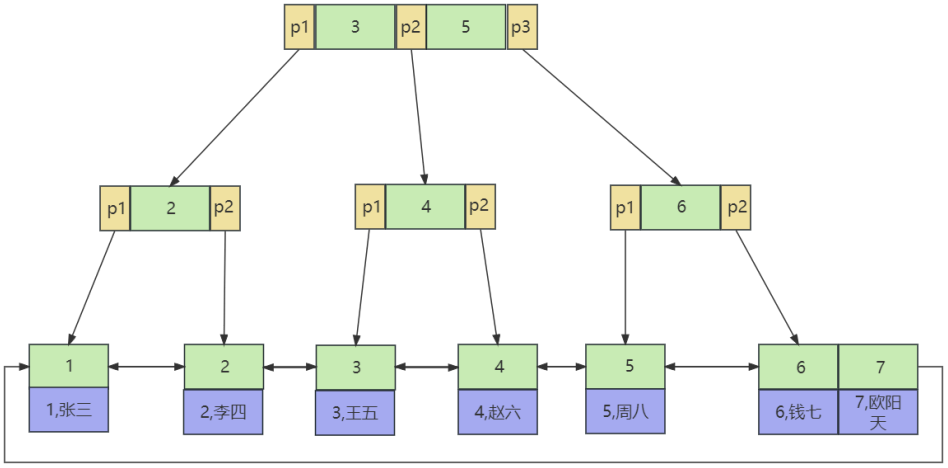
MyISAM的物理存储方式：当创建一张表t_user，并使用MyISAM存储引擎时，会在硬盘上生成这样一个文件：

- t_user.MYD （表数据）
- t_user.MYI （表索引）
- t_user.frm （表结构）

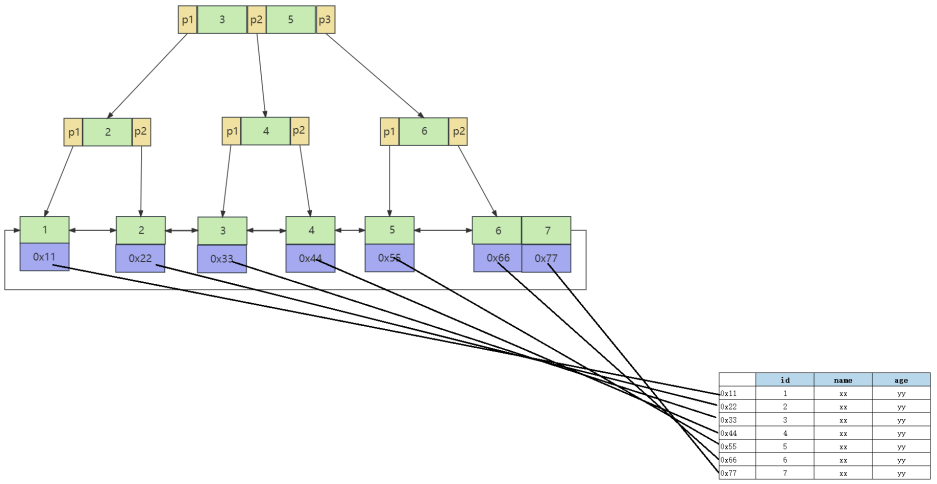
注意：从MySQL8.0开始，不再生成frm文件了，引入了数据字典，用数据字典（两个系统表）来统一存储表结构信息，例如：

- information_schema.TABLES （表包含了数据库中所有表的信息，例如表名、数据库名、引擎类型等）
- information_schema.COLUMNS（表包含了数据库中所有表的列信息，例如列名、数据类型、默认值等）

聚集索引的原理图：（B+树，叶子节点上存储了索引值 + 数据）



非聚集索引的原理图：（B+树，叶子节点上存储了索引值 + 行指针）（行指针再指向单独存数据的地方）



聚集索引的优点和缺点：

1. 优点：聚集索引将数据存储在索引树的叶子节点上。可以减少一次查询，因为查询索引树的同时可以获取数据。
2. 缺点：对数据进行修改或删除时需要更新索引树，会增加系统的开销。

二级索引

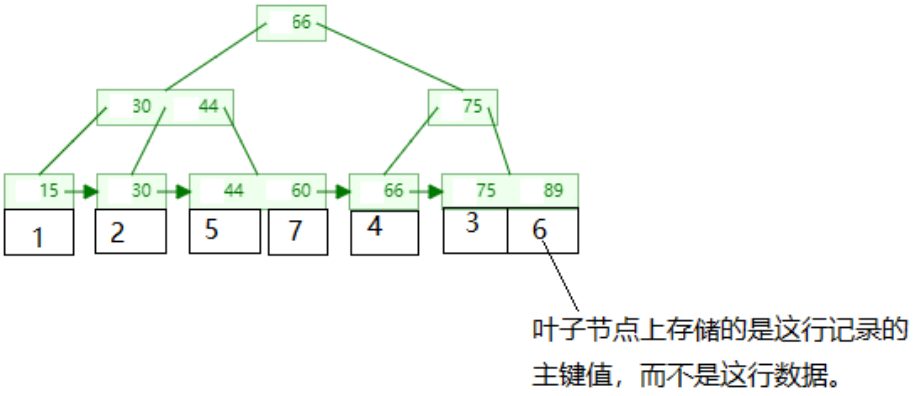
二级索引也属于非聚集索引。也有人把二级索引称为辅助索引。

假设有表t_user，id是主键，age是非主键；在age字段上添加的索引称为二级索引。（**所有非主键索引都是二级索引**）

二级索引的数据结构：

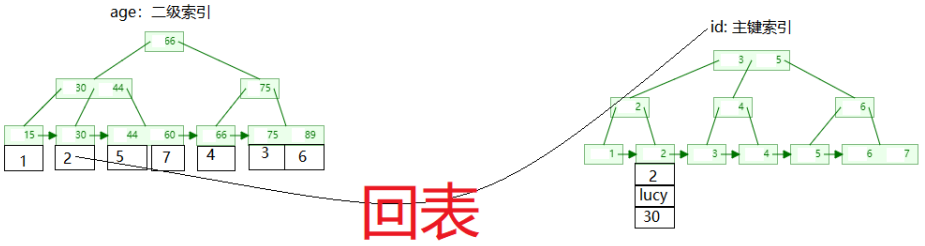
id	name	age
1	jack	15
2	lucy	30
3	tom	75
4	jerry	66
5	james	44
6	sea	89
7	sa	60

age二级索引的数据结构



二级索引的查询原理：

```
select * from t_user where age = 30;
/* 因为查的*需要所有数据，实际上底层还会调用语句：  select * from t_user where id = 2;   也就发生了“回表”现象   */
```



避免“回表【回到原数据表进行查询】”是提高SQL执行效率的手段。例如：`select id from t_user where age = 30;`这样的语句是不需要回表的。 不过要查`select id, name from t_user where age = 30;`时也还是会“回表”。

所以尽可能避免随意使用`select *`，会导致回表而效率降低。

覆盖索引

覆盖索引（Covering Index），是**指某个查询语句可以通过索引的覆盖来完成，而不需要"回表"查询真实数据。**其中的覆盖指的是在执行查询语句时，**查询需要的所有列都可以从索引中提取到**，而不需要再去查询实际数据行获取查询所需数据。

当使用覆盖索引时，MySQL可以直接通过索引，也就是索引上的数据来获取所需的结果，而不必再去查找表中的数据。这样可以显著提高查询性能。

假设有一个用户表（user）包含以下列：id, username, email, age。

常见的查询是根据用户名查询用户的邮箱。如果为了提高这个查询的性能，可以创建一个覆盖索引，包含（username, email）这两列。

创建覆盖索引的SQL语句可以如下：

```
CREATE INDEX idx_user_username_email ON user (username, email);    /* 叶子节点上存储的是主键值 */
```

当执行以下查询时：

```
SELECT email FROM user WHERE username = 'lucy';
```

MySQL可以直接使用覆盖索引（idx_user_username_email）来获取查询结果，而不必再去查找用户表中的数据。这样可以减少磁盘I/O并提高查询效率。而如果没有覆盖索引，MySQL会先使用索引（username）来找到匹配的行，然后再回表查询获取邮箱，这个过程会增加更多的磁盘I/O和查询时间。

值得注意的是，**覆盖索引的创建需要考虑查询的字段选择。如果查询需要的字段较多，可能需要创建包含更多列的覆盖索引**，以满足完全覆盖查询的需要。

覆盖索引具有以下优点：

1. 提高查询性能：覆盖索引能够满足查询的所有需求，同时不需要访问表中的实际数据行，从而可以提高查询性能。这是因为DBMS可以直接使用索引来执行查询，而不需要从磁盘读取实际的数据行。
2. 减少磁盘和内存访问次数：当使用覆盖索引时，DBMS不需要访问实际的数据行。这样可以减少磁盘和内存访问次数，从而提高查询性能。
3. 减少网络传输：由于在覆盖索引中可以存储所有查询所需的列，因此可以减少数据的网络传输次数，从而提高查询的性能。
4. 可以降低系统开销：在高压力的数据库系统中，使用覆盖索引可以减少系统开销，从而提高系统的可靠性和可维护性。

覆盖索引的缺点包括：

1. 需要更多的内存：覆盖索引需要存储查询所需的所有列，因此需要更多的内存来存储索引。在大型数据库系统中，这可能会成为一项挑战。
2. 会使索引变得庞大：当索引中包含了许多列时，它们可能会使索引变得非常庞大，从而影响查询性能，并且可能会占用大量的磁盘空间。
3. 只有在查询中包含了索引列时才能使用：只有当查询中包含了所有的索引列时才能使用覆盖索引。如果查询中包含了其他列，DBMS仍然需要访问实际的数据行，并且无法使用覆盖索引提高查询性能。

索引下推

索引下推（**Index Condition Pushdown**）是一种 MySQL 中的**优化方法**，它可以**将查询中的过滤条件下推到索引层级中处理**，从而**减少回表次数，优化查询性能。**

具体来说，在使用索引下推时，**MySQL 会在索引的叶节点层级执行查询的过滤条件，过滤掉无用的索引记录**，**仅返回符合条件的记录的主键，这样就可以避免查询时回表读取表格的数据行**，从而缩短了整个查询过程的时间。

假设有以下表结构：表名：users

id	name	age	city
1	John	25	New York
2	Alice	30	London
3	Bob	40	Paris
4	Olivia	35	Berlin
5	Michael	28	Sydney

现在我们创建了一个多列索引：（**索引下推通常是基于多列索引的**）

```
ALTER TABLE users ADD INDEX idx_name_city_age (name, city, age);
```

假设我们要查询年龄大于30岁，并且所在城市是"London"的用户，假设只给age字段添加了索引，它就不会使用索引下推。传统的查询优化器会将所有满足年龄大于30岁的记录读入内存，然后再根据城市进行筛选。

使用索引下推优化后，在**索引范围扫描的过程中，优化器会判断只有在城市列为"London"的情况下，才会将满足年龄大于30岁的记录加载到内存中。**这样就可以**避免不必要的IO和数据传输，提高查询性能。**

具体的查询语句可以是：

```
SELECT * FROM users WHERE age > 30 AND city = 'London';
```

在执行这个查询时，优化器会使用索引下推技术，先根据索引范围扫描找到所有满足条件的记录，然后再回到原数据表中获取完整的行数据，最终返回结果。

通过使用索引下推优化技术，可以减少不必要的数据加载和IO操作，提高查询性能。

在**一般情况下，索引下推是MySQL优化器自动处理的，并不需要程序员进行干预。**

MySQL优化器会根据查询条件和索引的定义，自动决定是否使用索引下推优化技术。当条件满足索引下推的使用场景时，优化器会自动选择使用索引下推。这个决策是根据优化器的统计信息和查询的成本估算来进行的。

单列索引（单一索引）

单列索引（Single-column Index）是指**对数据库表中的某一列或属性进行索引创建，对该列进行快速查找和排序操作。**单列索引可以加快查询速度，提高数据库的性能。

举个例子，假设有一个学生表（student），其中有以下几个列：学生编号（student_id）、姓名（name）、年龄（age）和性别（gender）。

如果针对学生表的学生编号（student_id）列创建了单列索引，那么可以快速地根据学生编号进行查询或排序操作。例如：

```
SELECT * FROM student WHERE student_id = 123456;
```

由于对学生编号列建立了单列索引，所以数据库可以直接通过索引快速定位到具有学生编号123456的那一行记录，从而加快查询速度。

复合索引（组合索引）

复合索引（Compound Index）也称为多列索引（Multi-Column Index），是指对数据库表中多个列进行索引创建。

与单列索引不同，**复合索引可以包含多个列。这样可以将多个列的值组合起来作为索引的键**，以提高多列条件查询的效率。

举例，假设有一个订单表（Order），其中包含以下几个列：订单编号（OrderID）、客户编号（CustomerID）、订单日期（OrderDate）和订单金额（OrderAmount）。

如果为订单表的客户编号和订单日期这两列创建复合索引（CustomerID, OrderDate），那么可以在查询时同时根据客户编号和订单日期来快速定位到匹配的记录。

例如，查询客户编号为123456且订单日期为2021-01-01的订单信息：

```
SELECT * FROM Order WHERE CustomerID = 123456 AND OrderDate = '2021-01-01';
    /* 注意 条件中 CustomerID与OrderDate的顺序刚好与创建复合索引时的（CustomerID, OrderDate）一致  */
```

由于为客户编号和订单日期创建了复合索引，数据库可以使用这个索引来快速定位到符合条件的记录，从而加快查询速度。复合索引的使用能够提高多列条件查询的效率，但需要注意的是，复合索引的创建和维护可能会增加索引的存储空间和对于写操作的影响。

相对于单列索引，复合索引有以下几个优势：

1. 减少索引的数量：**复合索引可以包含多个列，因此可以减少索引的数量，减少索引的存储空间和维护成本。**
2. 提高查询性能：当查询条件中涉及到复合索引的多个列时，数据库可以使用复合索引进行快速定位和过滤，从而提高查询性能。
3. **覆盖查询**：如果复合索引包含了所有查询需要的列，那么数据库可以直接使用索引中的数据，而不需要再进行表的读取，从而提高查询性能。
4. 排序和分组：由于复合索引包含多个列，因此可以用于排序和分组操作，从而提高排序和分组的性能。

索引的优缺点

索引是数据库中一种重要的数据结构，用于加速数据的检索和查询操作。它的优点和缺点如下：

优点：

1. **提高查询性能**：通过创建索引，可以大大减少数据库查询的数据量，从而提升查询的速度。
2. **加速排序**：当查询需要按照某个字段进行排序时，索引可以加速排序的过程，提高排序的效率。
3. **减少磁盘IO**：索引可以减少磁盘IO的次数，这对于磁盘读写速度较低的场景，尤其重要。

缺点：

1. 占据**额外的存储空间**：索引需要占据额外的存储空间，特别是在大型数据库系统中，索引可能占据较大的空间。
2. **增删改操作的性能损耗**：每次对数据表进行插入、更新、删除等操作时，**需要更新索引**，会导致操作的性能降低。
3. **资源消耗较大**：**索引需要占用内存和CPU资源，特别是在大规模并发访问**的情况下，可能对系统的性能产生影响。

何时用索引

在以下情况下建议使用索引：

1. **频繁执行查询**操作**的字段**：如果这些字段经常被查询（where后面的字段），使用索引可以提高查询的性能，减少查询的时间。
2. **大表**：当表的数据量较大时，使用索引可以快速定位到所需的数据，提高查询效率。
3. **需要排序或者分组**的字段：在对字段进行排序或者分组操作时，索引可以减少排序或者分组的时间。
4. **外键关联的字段**：在进行表之间的**关联查询**时，使用索引可以加快关联查询的速度。

在以下情况下不建议使用索引：

1. **频繁执行更新**操作的表：如果表经常被更新数据，使用索引可能会降低更新操作的性能，因为每次更新都需要维护索引。
2. 小表：对于数据量较小的表，使用索引可能并不会带来明显的性能提升，反而会占用额外的存储空间。
3. 对于**唯一性很差的字段**，一般不建议添加索引。当一个字段的唯一性很差时，查询操作基本上需要扫描整个表的大部分数据。如果为这样的字段创建索引，索引的大小可能会比数据本身还大，导致索引的存储空间占用过高，同时也会导致查询操作的性能下降。

总之，索引需要根据具体情况进行使用和权衡，需要考虑到表的大小、查询频率、更新频率以及业务需求等因素。

第十五章.MySQL优化

MySQL优化手段

MySQL数据库的优化手段通常包括但不限于：

- **SQL查询优化**：这是**最低成本的优化**手段，通过优化查询语句、适当添加索引等方式进行。并且**效果显著。**
- 库表结构优化：通过规范化设计、优化索引和数据类型等方式进行库表结构优化，需要对数据库结构进行调整和改进
- 系统配置优化：根据硬件和操作系统的特点，调整最大连接数、内存管理、IO调度等参数
- 硬件优化：升级硬盘、增加内存容量、升级处理器等硬件方面的投入，需要购买和替换硬件设备，成本较高

> 主要掌握：SQL查询优化

SQL性能分析工具

查看数据库整体情况

通过以下命令可以查看当前数据库在SQL语句执行方面的整体情况：

```
show global status like 'Com_select';
/* ↑ 在MySQL中用于显示自服务器启动以来执行的SELECT查询的总数。这个计数器包括所有类型的查询操作，不仅仅是SELECT语句，还包括其他需要查询数据库的操作，比如登录MySQL时验证密码的操作也会被计入Com_select的计数中 */
show global status like 'Com_insert';
show global status like 'Com_delete';
show global status like 'Com_update';

show global status like 'Com_______';
```

这些**结果反映了从 MySQL 服务器启动到当前时刻，执行的各种SQL语句总数。**对于 MySQL 性能优化来说，通过查看 `Com_select` 的值可以了解 SELECT 查询在整个 MySQL 服务期间所占比例的情况：

- 如果 `Com_select` 次数过高，可能说明查询表中的每条记录都会返回过多的字段。
- 如果 `Com_select` 次数很少，同时insert或delete或update的次数很高，可能说明服务器运行的应用程序过于依赖写入操作和少量读取操作。

总之，通过查看 `Com_select` 的值，可以了解 MySQL 服务器的长期执行情况，并在优化查询性能时，帮助了解 MySQL 的性能瓶颈。

慢查询日志

慢查询日志**文件可以将查询较慢的DQL语句记录下来**，便于**定位需要调优的select语句。**

通过以下命令查看慢查询日志功能是否开启：

```
/* 先登录MySQL，并选定其中一个database   */
show variables like 'slow_query_log';     /* 功能默认是关闭的，即显示为 OFF   */
```

慢查询日志功能默认是关闭的。请修改my.ini文件来开启慢查询日志功能，在my.ini的[mysqld]后面添加如下配置：

```
# my.ini
[mysqld]
slow_query_log=1        # slow_query_log=1表示开启慢查询日志功能（这是永久修改的，服务重启后仍开启）
long_query_time=3     # long_query_time=3表示：只要SELECT语句的执行耗时超过3秒则将其记录到慢查询日志中
```

重启mysql服务。再次查看是否开启慢查询日志功能。

此时尝试执行一条时长超过3秒的select语句：

```
select empno,ename,sleep(4) from emp where ename='smith';
```

慢查询日志文件默认存储在：MySQL根目录的 \data 目录下，默认的名字是：`计算机名-slow.log`

通过该文件可以清晰的看到哪些DQL语句属于慢查询。

show profiles

通过show profiles可以**查看一个SQL语句在执行过程中具体的耗时**情况。帮助**更好的定位问题**所在。

1.查看当前数据库是否支持 profile操作：

```
select @@have_profiling;
```

2.查看 profiling 开关是否打开：（如果支持 profile操作）

```
select @@profiling;
```

> 注意：navicat for mysql 是默认开启profiling开关的。 DOS命令窗口中输入该查询命令会显示profiling是关闭的。

3.将 profiling 开关打开：

```
set profiling = 1;
```

4.可以执行多条DQL语句，然后使用 show profiles; 来**查看当前数据库中执行过的每个SELECT语句的耗时**情况。

```
select empno,ename from emp;
select empno,ename from emp where empno=7369;
select count(*) from emp;
show profiles;
```

```
Query_ID   |   Duration (耗时)   |   Query
1          |      0.00021900     |  select @@profiling
2          |      0.01063050     |  select empno,ename from
emp
3          |      0.00038350     |  select empno,ename from
emp where empno=7369
4          |      0.00565100     |  select count(*) from emp
```

5.**查看某个SQL语句在执行过程**中，每个阶段的耗时情况：

```
show profile for query 4;     /*  选择Query_ID，这里就是查询上面的
select count(*) from emp语句情况  */
```

6.想**查看执行过程中cpu的情况**，可以执行以下命令：

```
show profile cpu for query 4;
```

explain

explain命令可以**查看一个DQL语句的执行计划，根据执行计划可以做出相应的优化措施。**提高执行效率。

```
/* 语法：  explain DQL语句  */
explain select * from emp where empno=7369;
```

id

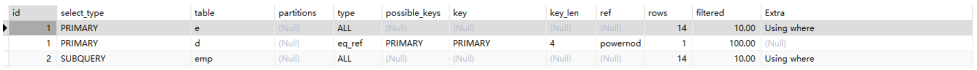
id**反映出一条select语句执行顺序，id越大优先级越高。id相同则按照自上而下**的顺序执行。

```
explain select e.ename,d.dname from emp e join dept d on
e.deptno=d.deptno join salgrade s on e.sal between s.losal and
s.hisal;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	d	(Null)	ALL	PRIMARY	(Null)	(Null)	(Null)	4	100.00	(Null)
1	SIMPLE	e	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	14	10.00	Using where; Using join buffer (hash join)
1	SIMPLE	s	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	5	20.00	Using where; Using join buffer (hash join)

由于id相同，反映出三张表在执行顺序上属于平等关系，执行时采用，先d，再e，最后s。

```
explain select e.ename,d.dname from emp e join dept d on
e.deptno=d.deptno where e.sal=(select sal from emp where
ename='ford');
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	e	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	14	10.00	Using where
1	PRIMARY	d	(Null)	eq_ref	PRIMARY	PRIMARY	4	powernod	1	100.00	(Null)
2	SUBQUERY	emp	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	14	10.00	Using where

反映出，先执行子查询emp表，然后让e和d做表连接。

select_type

反映了mysql查询语句的类型。常用值包括：

- SIMPLE：表示查询中不包含 子查询或UNION 操作。这种查询通常包括一个表或是最多一个联接（JOIN）
- PRIMARY：表示当前查询是一个主查询。（主要的查询）
- UNION：表示查询中包含UNION操作
- SUBQUERY：子查询
- DERIVED：派生表（表示查询语句出现在from后面）

table

反映了这个查询操作的是哪个表。

type

反映了查询表中数据时的访问类型，常见的值：

1. NULL：**效率最高，一般不可能优化到这个级别**，只有查询时没有查询表的时候，访问类型是NULL。例如：`select 1;`
2. system：通常访问系统表的时候，访问类型是system。一般也很难优化到这个程序。
3. const：**根据主键或者唯一性索引查询**，索引值是常量值时。 `explain select * from emp where empno=7369;` (empno有唯一性)
4. eq_ref：根据主键或者唯一性索引查询。索引值不是常量值。
5. ref：使用了非唯一的索引进行查询。
6. range：使用了索引，扫描了索引树的一部分。
7. index：表示用了索引，但是也需要遍历整个索引树。
8. all：**全表扫描**

效率最高的是NULL，效率最低的是all，从上到下，从高到低。 （一般提升到3、4级别就行了）

possible_keys

这个查询可能会用到的索引

key

实际用到的索引

key_len

反映索引中在查询中使用的列所占的总字节数。

rows

查询扫描的预估计行数。

Extra

给出了与查询相关的额外信息和说明。这些额外信息可以帮助更好地理解查询执行的过程。

索引优化

一个字段加索引与不加索引的效率是有很大差别的。

最左前缀原则

假设有这样一张表：

```
create table t_customer(
    id int primary key auto_increment,
    name varchar(255),
    age int,
    gender char(1),
    email varchar(255)
);
```

添加了这些数据：

```
insert into t_customer values(null, 'zhangsan', 20, 'M', 'zhangsan@123.com'); /* 在 INSERT 语句中为 AUTO_INCREMENT 字段指定 NULL 时，MySQL 会自动为该字段生成一个唯一的自增值*/
insert into t_customer values(null, 'lisi', 22, 'M', 'lisi@123.com');
insert into t_customer values(null, 'wangwu', 18, 'F', 'wangwu@123.com');
insert into t_customer values(null, 'zhaoliu', 22, 'F', 'zhaoliu@123.com');
insert into t_customer values(null, 'jack', 30, 'M', 'jack@123.com');
```

添加了这样的复合索引：（最左边是name字段）

```
create index idx_name_age_gender on t_customer(name,age,gender);
```

最左前缀原则：当查询语句条件中包含了这个复合索引最左边的列 name 时，此时复合索引才会起作用。生效后可能使用完全索引，可能部分使用索引。

即：where中条件必须出现 name 的参与。

验证1：

```
explain select * from t_customer where name='zhangsan' and
age=20 and gender='M';
```

验证结果：完全使用了索引 （索引长度为1033个字节）

验证2：

```
explain select * from t_customer where name='zhangsan' and
age=20;
```

验证结果：使用了部分索引 （key_len 变少了，索引长度为1028个字节，说明gender在索引中占用5个字节）

验证3：

```
explain select * from t_customer where name='zhangsan';
```

验证结果：使用了部分索引 （索引长度为1023个字节，说明age在索引中占用5个字节）

验证4：

```
explain select * from t_customer where age=20 and gender='M'
and name='zhangsan';
```

验证结果：完全使用了索引 （只要name出现在条件里就行，不需要保证最靠左）

验证5：

```
explain select * from t_customer where gender='M' and age=20;
```

验证结果：没有使用任何索引

验证5：

```
explain select * from t_customer where gender='M' and
name='zhangsan';
```

验证结果：使用了部分索引 （索引长度为1023个字节，说明只用到了name的索引，没用到gender的，因为复合索引中间断开了导致性能折损：要么三个都存在，要么按顺序的name, age ）

范围查询时，在"范围条件"右侧的列索引会失效：

验证：

```
explain select * from t_customer where name='zhangsan' and age
> 20 and gender='M';  /* gender在范围条件右侧 */
```

验证结果：name和age列索引生效。gender列索引无效。

怎么解决？建议范围查找时范围条件带上“=”，这样可以让索引完全生效：

```
explain select * from t_customer where name='zhangsan' and age
>= 20 and gender='M';
```

索引失效情况

有这样一张表：

```
create table t_emp(
  id int primary key auto_increment,
  name varchar(255),
  sal int,
  age char(2)
);
```

有这样一些数据：

```
insert into t_emp values(null, '张三', 5000,'20');
insert into t_emp values(null, '张飞', 4000,'30');
insert into t_emp values(null, '李飞', 6000,'40');
```

有这样一些索引：

```
create index idx_t_emp_name on t_emp(name);
create index idx_t_emp_sal on t_emp(sal);
create index idx_t_emp_age on t_emp(age);
```

以下的索引失效要尽可能避免。

索引列参加了运算，索引失效

```
explain select * from t_emp where sal > 5000;
```

验证结果：使用了索引

```
explain select * from t_emp where sal*10 > 50000;
```

验证结果：索引失效 （索引列sal参与了运算）

索引列进行模糊查询时以 % 开始的，索引失效

```
explain select * from t_emp where name like '张%';
```

验证结果：索引有效

```
explain select * from t_emp where name like '%飞';
```

验证结果：索引失效 （模糊查询以 % 开始）

索引列是字符串类型，但查询时省略了单引号，索引失效

```
explain select * from t_emp where age='20';
```

验证结果：索引有效

```
explain select * from t_emp where age=20;
```

验证结果：索引失效

查询条件中有or，只要有未添加索引的字段，索引失效

```
explain select * from t_emp where name='张三' or sal=5000;
```

验证结果：使用了索引

将t_emp表sal字段上的索引删除：

```
alter table t_emp drop index idx_t_emp_sal;
```

再次验证：

```
explain select * from t_emp where name='张三' or sal=5000;
```

验证结果：索引失效 （sal是未索引字段）

当查询的符合条件的记录在表中占比较大，索引失效

复制一张新表：emp2

```
create table emp2 as select * from emp;
show index from emp2;  /* 只是单纯复制了表的数据，表结构索引等并未设置 */
```

给sal添加索引：

```
alter table emp2 add index idx_emp2_sal(sal);
```

验证1：

```
explain select * from emp2 where sal > 800;
```

验证结果：索引失效（sal > 800的记录在表中占比很大，所以用的全表扫描；因为用索引还会有回表的操作）

验证2：

```
explain select * from emp2 where sal > 2000;
```

验证结果：使用索引

关于is null和is not null的索引失效问题

给emp2的comm字段添加一个索引：

```
create index idx_emp2_comm on emp2(comm);
```

将emp2表的comm字段值全部更新为NULL：

```
update emp2 set comm=null;
```

验证此时条件使用is null是否走索引：

```
explain select * from emp2 where comm is null;
```

验证结果：索引失效 （因为此时 comm is null 的记录占全表的占比很大，所以全表扫描）

验证此时条件使用is not null是否走索引：

```
explain select * from emp2 where comm is not null;
```

验证结果：使用索引

> 结论：走索引还是不走索引，和数据分布有很大关系，如果**符合条件的记录占比较大，会考虑使用全表扫描，而放弃走索引。**

指定索引

当**一个字段上既有单列索引，又有复合索引时**，可以通过以下的SQL提示**来要求该SQL语句执行时采用哪个索引**：

- use index(索引名称)：建议使用该索引，**只是建议，底层MySQL会根据实际效率来考虑是否使用你推荐的索引。**
- ignore index(索引名称)：忽略使用该索引
- force index(索引名称)：强行使用该索引

查看 t_customer 表上的索引：

```
show index from t_customer;
```

可以看到name age gender三列添加了一个复合索引。

现在给name字段添加一个单列索引：

```
create index idx_name on t_customer(name);
```

看看以下的语句默认使用了哪个索引：

```
explain select * from t_customer where name='zhangsan';
```

验证得知，默认使用了复合索引。

如何建议使用单列索引idx_name：

```
explain select * from t_customer use index(idx_name) where
name='zhangsan';
```

如何忽略使用符合索引 idx_name_age_gender：

```
explain select * from t_customer ignore
index(idx_name_age_gender) where name='zhangsan';
```

如何强行使用单列索引idx_name：

```
explain select * from t_customer force index(idx_name) where
name='zhangsan';
```

覆盖索引

覆盖索引强调的是：**在select后面写字段的时候，这些字段尽可能是索引所覆盖的字段**，这样**可以避免回表查询。**

尽可能避免使用`select *`，因为select * 很容易导致回表查询。（本质是：**能在索引上检索的，就不要再次回表查询了。**）

例如：有一张表 emp3，其中 ename,job添加了联合索引：idx_emp3_ename_job，以下这个select语句就不会回表：

```
drop table if exists emp3;
create table emp3 as select * from emp;
alter table emp3 add constraint emp3_pk primary key(empno);
create index idx_emp3_ename_job on emp3(ename,job);

explain select empno,ename,job from emp3 where ename='KING';
 /* Extra中写的Using index  */
```

如果查询语句要查找的列没有在索引中，则会回表查询，例如：

```
explain select empno,ename,job,sal from emp3 where
ename='KING';  /* Extra中为空白 （sal没在索引中）  */
```

面试题：t_user表字段如下：id,name,password,realname,birth,email。表中数据量500万条，请针对以下SQL语句给出优化方案：

```
select id,name,realname from t_user where name='鲁智深';  /* 注意 id 本身有主键索引，realname也需要查询  */
```

如果只给name添加索引，底层会进行大量的回表查询，效率较低，**建议给name和realname两个字段添加联合索引**，虽然查询的时候是部分使用索引，但这样大大减少回表操作，提高查询效率。

前缀索引

如果一个字段类型是varchar或text字段，**字段中存储的是文本或者大文本，直接对这种长文本创建索引，会让索引体积很大**，怎么优化呢?

可以**将字符串的前几个字符截取下来当做索引来创建。**这种索引**被称为前缀索引**，例如：

```
drop table if exists emp4;
create table emp4 as select * from emp;

create index idx_emp4_ename_2 on emp4(ename(2));  /* 将emp4表中ename字段的前2个字符创建到索引当中 */
```

使用前缀索引时，需要**通过以下计算效率的公式来确定使用前几个字符作为索引**：

```
select count(distinct substring(ename, 1, 前几个字符)) / count(*) from 表名;
```

以上查询结果越接近1，表示索引的效果越好。（原理：**做索引值的话，索引值越具有唯一性，效率越高**）

选择单列索引和复合索引

当**查询语句的条件中有多个条件，建议将这几个列创建为复合索引**，因为创建单列索引很容易回表查询。

例如分别给emp5表ename，job添加两个单列索引：

```
create table emp5 as select * from emp;
alter table emp5 add constraint emp5_pk primary key(empno);

create index idx_emp5_ename on emp5(ename);
create index idx_emp5_job on emp5(job);
```

执行以下查询语句：

```
explain select empno,ename,job from emp5 where ename='SMITH'
and job='CLERK';
/* 实际只使用了ename的索引。  Extra中是Using where（表示有回表操作）
[Using index不发生回表]  */
```

ename和job都出现在查询条件中，可以给emp6表的ename和job创建一个复合索引：

```
create table emp6 as select * from emp;
alter table emp6 add constraint emp6_pk primary key(empno);

create index idx_emp6_ename_job on emp6(ename,job);
explain select empno,ename,job from emp6 where ename='SMITH'
and job='CLERK';
```

对于以上查询语句，使用复合索引避免了回表，因此这种情况下还是建议使用复合索引。

注意：**创建索引时应考虑最左前缀原则，主字段并且具有很强唯一性的字段建议排在第一位**，例如：

```
create index idx_emp_ename_job on emp(ename,job);
```

和以下方式对比：

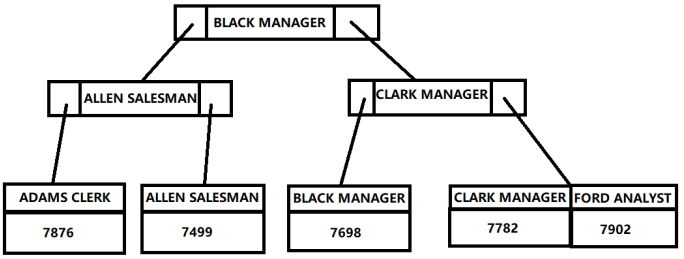
```
create index idx_emp_job_ename on emp(job,ename);
```

由于ename是主字段，并且ename具有很好的唯一性，建议将ename列放在最左边。因此这两种创建复合索引的方式，建议采用第一种。

复合索引底层原理：

> 复合索引底层排序是按照靠左边的字段升序排，相同再看下一个字段。

empno	ename	job
7876	ADAMS	CLERK
7499	ALLEN	SALESMAN
7698	BLAKE	MANAGER
7782	CLARK	MANAGER
7902	FORD	ANALYST
7900	JAMES	CLERK
7566	JONES	MANAGER
7839	KING	PRESIDENT
7654	MARTIN	SALESMAN
7934	MILLER	CLERK
7788	SCOTT	ANALYST
7369	SMITH	CLERK
7844	TURNER	SALESMAN
7521	WARD	SALESMAN



索引创建原则

1. 需要**表数据量庞大**才会考虑创建索引，通常超过百万条数据。（才几千条数据根本不需要索引）
2. **经常出现在where，order by，group by后面的字段建议添加**索引。
3. 创建索引的**字段尽量具有很强的唯一性。**
4. 如果**字段存储文本**，内容较大，一定**要创建前缀索引。**
5. **尽量使用复合索引**来避免回表查询，使用单列索引容易回表查询。
6. 如果**一个字段中的数据不会为NULL，建议建表时添加not null约束**，这样**优化器就知道使用哪个索引列更加有效。**
7. **不要创建太多索引**，当对**数据进行增删改**的时候，**索引需要重新排序。**
8. 如果很少的查询，**经常的增删改不建议加索引。**

SQL优化

order by的优化

准备数据：

```
drop table if exists workers;
create table workers(
  id int primary key auto_increment,
  name varchar(255),
  age int,
  sal int
);
insert into workers values(null, '孙悟空', 500, 50000);
insert into workers values(null, '猪八戒', 300, 40000);
insert into workers values(null, '沙和尚', 600, 40000);
insert into workers values(null, '白骨精', 600, 10000);
```

explain查看一个带有order by的语句时，Extra列会显示：using index 或者 using filesort，区别是什么？

- using index：表示**使用索引**，因为**索引是提前排好序的。效率很高。**
- using filesort：表示**使用文件排序**，这就表示没有走索引，对表中数据进行排序，**排序时将硬盘的数据读取到内存当中**，在**内存当中排好序。**这个**效率是低的，应避免。**

此时name没有添加索引，如果根据name进行排序的话：

```
explain select id,name from workers order by name;  /* Extra中是using filesort ，  效率较低  */
```

给name添加索引再根据name排序：

```
create index idx_workers_name on workers(name);
explain select id,name from workers order by name;   /* Extra中是 Using index，效率提升  */
```

如果要过age和sal两个字段进行排序，最好给age和sal两个字段添加复合索引，不添加复合索引时：
按照age升序排，如果age相同则按照sal升序

```
explain select id,age,sal from workers order by age,sal;  /* using filesort  */
```

给age和sal添加复合索引，再按照age升序排，如果age相同则按照sal升序：：

```
create index idx_workers_age_sal on workers(age, sal);
explain select id,age,sal from workers order by age,sal;  /* Using index  */
```

在B+树上叶子结点上的所有数据默认是按照升序排列的，如果按照age降序，如果age相同则按照sal降序，会走索引吗？

```
explain select id,age,sal from workers order by age desc,sal desc;  /* Extra中是Backward index scan; Using index */
```

这样效率也是很高的，因为B+树叶子结点之间采用的是双向指针。可以从左向右（升序），也可以从右向左（降序）。

如果一个升序，一个降序会怎样呢？

```
explain select id,age,sal from workers order by age asc, sal desc;   /* Using index; Using filesort  */
```

可见age使用了索引，但是sal没有使用索引。怎么办呢?

可以针对这种排序情况创建对应的索引来解决:

```
create index idx_workers_ageasc_saldesc on workers(age asc, sal desc);  /* 建索引时就可以指定升序降序，就会根据升降序要求来创建对应的B+树 */
explain select id,age,sal from workers order by age asc, sal desc;
```

order by也遵循最左前缀法则:

```
explain select id,age,sal from workers order by sal;     /* Using index; Using filesort  */
/*  写成 select id,age,sal from workers order by age, sal;  才会完全使用索引  */
```

未使用覆盖索引会怎样?

```
explain select * from workers order by age,sal;     /*  Using filesort  */
```

通过测试得知，排序也要尽量使用覆盖索引。

order by 优化原则总结:

1. 排序也要遵循最左前缀法则。
2. 使用覆盖索引。
3. 针对不同的排序规则，创建不同索引。（如果所有字段都是升序，或者所有字段都是降序，则不需要创建新的索引）
4. **如果无法避免filesort，要注意排序缓存的大小**，默认缓存大小256KB，可以修改系统变量 sort_buffer_size：

```
show variables like 'sort_buffer_size';
```

group by优化

创建empx表：

```
create table empx as select * from emp;
```

job字段上没有索引，根据job进行分组，查看每个工作岗位有多少人：

```
select job,count(*) from empx group by job;
```

看是否走索引：

```
explain select job,count(*) from empx group by job;  /*  Extra 中是Using temporary （使用了临时表，效率较低） */
```

给job添加索引：

```
create index idx_empx_job on empx(job);
explain select job,count(*) from empx group by job;   /* Using index  */
```

group by也需要遵守最左前缀法则：

给deptno和sal添加复合索引

```
create index idx_empx_deptno_sal on empx(deptno, sal);   /*  最左前缀是deptno  */
```

根据deptno分组，查看每个部门人数：

```
explain select deptno,count(*) from empx group by deptno;  /* Using index （因为deptno是复合索引中最左边的字段）*/
```

根据sal分组，查看每个工资有多少人：

```
explain select sal, count(*) from empx group by sal;  /* Using index; Using temporary 使用了临时表，效率较低 */
```

如果将deptno（复合索引的最左列）添加到where条件中：

```
explain select sal, count(*) from empx where deptno=10 group by sal;  /* Using index */
```

效率有提升的，这说明，group by确实也遵循最左前缀法则。（where中使用了最左列）

limit优化

当**数据量特别庞大时，取数据时，越往后效率越低**，怎么提升？

```
select * from t_user order by id limit 1000000,2;  /*  0.59 sec */
select * from t_user order by id limit 5000000,2;  /*  2.65 sec */
select * from t_user order by id limit 9000000,2;  /*  4.57 sec */
```

MySQL官方给出的解决方案是：**使用覆盖索引+子查询的形式**来提升效率。

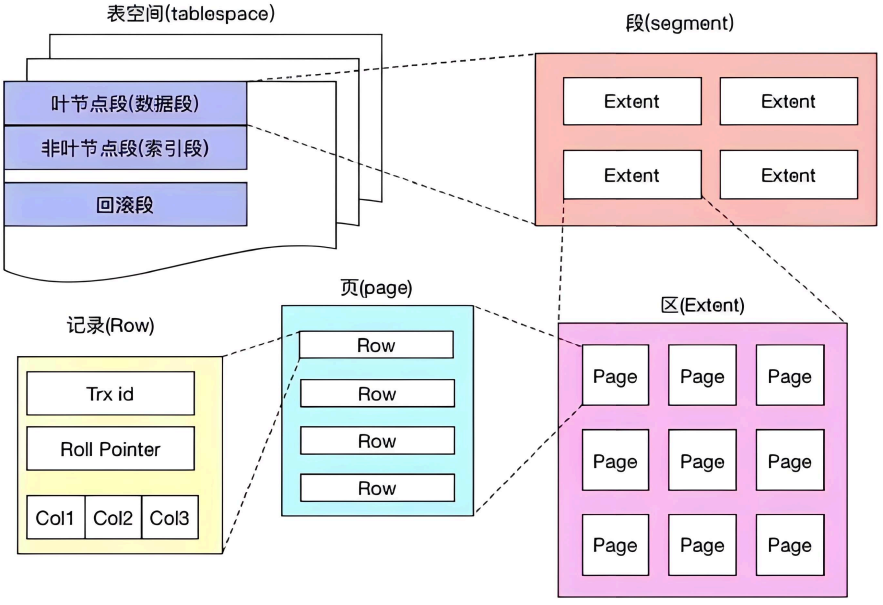
```
select id from t_user order by id limit 9000000, 2;  /* 覆盖索引（id是主键索引） */  /* 3.08 sec */
/* 要取数据的话就把 ↑ 用覆盖索引得到的临时表与原表做连接，再取对应的数据 */
select u.* from t_user u join (select id from t_user order by id limit 9000000,2) t on u.id = t.id;  /* 3.09 sec */
```

主键优化

主键设计原则：

1. **主键值不要太长**，二级索引叶子结点上存储的是主键值，主键值太长，容易导致索引占用空间较大。
2. **尽量使用auto_increment生成主键。**尽量**不要使用uuid做主键，因为uuid不是顺序插入。**
3. **最好不要使用业务主键**，因为业务的变化会导致主键值的频繁修改，**主键值不建议修改**，因为主键值修改，聚集索引一定会重新排序。
4. 在**插入数据时，主键值最好是顺序插入**，不要乱序插入，因为**乱序插入可能会导致B+树叶子结点频繁的进行页分裂与页合并**操作，效率较低。

1. 主键值对应聚集索引，插入主键值如果是乱序的，B+树叶子结点需要不断的重新排序，重排过程中还会频繁涉及到**页分裂和页合并**的操作，效率较低。
2. B+树上的每个节点都存储在页（page）中。一个页面中存储一个节点。
3. MySQL的InnoDB存储引擎一个页可以存储16KB的数据。
4. 如果主键值不是顺序插入的话，会导致频繁的页分裂和页合并。在**一个B+树中，页分裂和页合并是树的自动调整机制**的一部分。当一个页已经满了，再插入一个新的关键字时就会触发页分裂操作，将页中的关键字分配到两个新的页中，同时调整树的结构。相反，当一个页中的关键字数量下降到一个阈值以下时，就会触发页合并操作，将两个相邻的页合并成一个新的页。**如果主键值是随机的、不是顺序插入的，那么页的利用率会降低，页分裂和页合并的次数就会增加。**由于页的分裂和合并是比较耗时的操作，频繁的分裂和合并会降低数据库系统的性能。因此，为了优化B+树的性能，可以将主键值设计成顺序插入的，这样可以减少页的分裂和合并的次数，提高B+树的性能。在实际应用中，如果对主键值的顺序性能要求不是特别高，也可以采用一些技术手段来减少页分裂和合并，例如B+树分裂时采用"延迟分裂"技术，或者通过调整页的大小和节点的大小等方式来优化B+树的性能。



insert优化

insert优化原则：

- **批量插入**：数据量较大时，不要一条一条插入，可以批量插入，当然，**建议一次插入数据不超过1000条**

```
insert into t_user(id,name,age) values (1,'jack',20),
(2,'lucy',30), (3,'timi',22);
```

- mysql默认是自动提交事务，只要执行一条DML语句就自动提交一次，因此，**当插入大量数据时，建议手动开启事务和手动提交事务。不建议使用数据库事务自动提交机制。**
- **主键值建议采用顺序插入**，顺序插入比乱序插入效率高。
- **超大数据量插入(一次插入百万条记录)可以考虑使用mysql提供的load指令**，load指令可以将csv文件中的数据批量导入到数据库表当中，并且效率很高，过程如下：
  - 第一步：登录mysql时指定参数

```
mysql --local-infile -uroot -p1234
```

- 第二步：开启local_infile功能

```
set global local_infile = 1;
```

- 第三步：执行load指令

```
use powernode;
create table t_temp(
  id int primary key,
  name varchar(255),
  password varchar(255),
  birth char(10),
  email varchar(255)
);

load data local infile 'D:\\powernode\\resources\\t_temp-
100w.csv' into table t_temp fields terminated by ',' lines
terminated by '\n';
```

文件中的数据如下：

```
# 每一列之间用 ',' 隔开      每一行用 '\n' 隔开
1,zhangsan,123,2010-10-11,zhangsan@1234.com
2,zhangsan,123,2010-10-11,zhangsan@1234.com
3,zhangsan,123,2010-10-11,zhangsan@1234.com
4,zhangsan,123,2010-10-11,zhangsan@1234.com
5,zhangsan,123,2010-10-11,zhangsan@1234.com
```

count(*)优化

分组函数count的使用方式：

- count(主键)
  - 原理：将每个主键值取出，累加
- count(常量值)
  - 原理：获取到每个常量值，累加
- count(字段)
  - 原理：取出字段的每个值，判断是否为NULL，不为NULL则累加。
- count(*)
  - 原理：**不用取值**，底层mysql做了优化，**直接统计总行数**，**效率最高。**

```
select count(*) from t_user;   /* 虽然行数达百万条时也要4sec，但与其他count(主键、字段、常量)等相比效率是最高的 */
```

结论：**如果要统计一张表中数据的总行数，建议使用 count(*)**

注意：

- 对于**InnoDB存储引擎**来说，**count计数的实现原理就是将表中每一条记录取出，然后累加。**如果想真正提高效率，可以自己使用额外的其他程序来实现，例如每向表中插入一条记录时，在redis数据库中维护一个总行数，这样获取总行数的时候，直接从redis中获取即可，这样效率是最高的。
- **对于MyISAM存储引擎**来说，当**一个select语句没有where条件时，获取总行数效率是极高的**，不需要统计，因为MyISAM存储引擎**维护了一个单独的总行数。**

update优化

当存储引擎是InnoDB时，表的行级锁是针对索引列添加的锁，如果索引失效了，或者不是索引列时，会提升为表级锁。

> 尽量不要无故去提升表的锁级别。 能保持在行级锁就不要上升到表级锁。

什么是行级锁？A事务和B事务，开启A事务后，通过A事务修改表中某条记录，修改后，在A事务未提交的前提下，B事务去修改同一条记录时，无法继续，直到A事务提交，B事务才可以继续。

因此，为了更新的效率，**建议update语句中where条件中的字段是添加索引的。**

有一张表：t_fruit

```
create table t_fruit(
  id int primary key auto_increment,
  name varchar(255)
);
insert into t_fruit values(null, '苹果');
insert into t_fruit values(null, '香蕉');
insert into t_fruit values(null, '橘子');
```

开启A事务和B事务，演示行级锁：

A事务	B事务
登录账号	
	登录账号
start transaction;	
	start transaction;
update t_fruit set name='火龙果' where id = 1;	
QueryOK, 1 row affected (0.01 sec)	
	update t_fruit set name='榴莲' where id = 1; 卡住
commit; (事务A结束之后，事务B继续执行)	
	QueryOK,1rowaffected(25.641 sec)

A事务	B事务
	commit;

当然，如果更新的不是同一行数据，事务A和事务B可以并发：

事务A	事务B
登录	
	登录
start transaction;	
	start transaction;
update t_fruit set name='西瓜' where id = 1;	
QueryOK, 1 row affected(0.00sec)	
	update t_fruit set name='梨' where id=2;
	QueryOK, 1 row affected(0.00sec)

行级锁是对索引列加锁，以上更新语句的where条件是id，id是主键，当然有索引，所以使用了行级锁。

如果索引失效，或者字段上没有索引，则会升级为表级锁：

事务A	事务B
登录	
	登录
start transaction;	
	start transaction;
update t_fruit set name='苹果' where name = '西瓜';	
QueryOK, 1 row affected(0.00sec)	
	update t_fruit set name='桃子' where name = '橘子';

事务A	事务B
	（事务B 卡住）
commit;	
	QueryOK, 1 row affected(10.00sec)