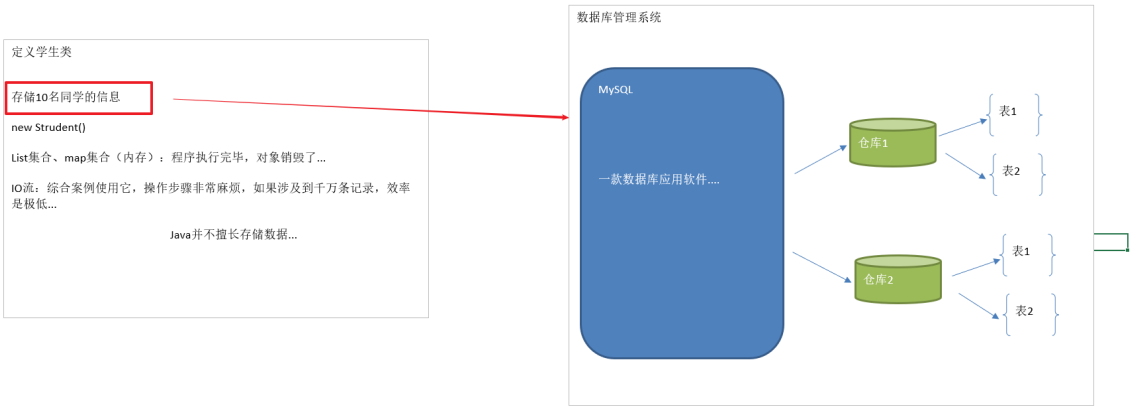
MySQL基础入门

1.数据库的基本概念

```
* 存储数据的仓库,本质上就是存储数据的文件系统，方便我们管理数据。
```

2.数据库管理系统

```
* 数据库管理系统（DataBase Management System，DBMS）：指一种操作和管理数据库的大型软件。

* 数据库管理系统-> mysql软件 -> 多个仓库 --> 多张表 --> 多条记录（数据）
```



3.实体(Java类)与表关系

```
* 一个实体类对应一张表
* 一个实例对象对应一条表记录
* 对象和数据产生关系映射【ORM：Object Relational Mapping】
```

4. 常见的关系型数据库

```
1. MYSQL：开源免费的数据库，小型的数据库.已经被Oracle收购了.MySQL6.x版本（社区、商业）也开始收费。

2. Oracle：收费的大型数据库，Oracle公司的产品。Oracle收购SUN公司，收购MYSQL。

3. DB2：IBM公司的数据库产品,收费的。常应用在银行系统中。

4. OceanBase：阿里的大型数据库，性能已超越Oracle全球第一

----------------------------------------------------------

5. SQLServer：MicroSoft 公司收费的中型的数据库。C#、.net等语言常使用。

```

```
6. SyBase：已经淡出历史舞台。提供了一个非常专业数据建模的工具PowerDesigner。

7. SQLite：嵌入式的小型数据库，应用在手机端。
```

5.MySQL软件的常见操作命令

```
1. 登录
    a.dos命令窗口
        1）本地登录
            mysql -u用户名 -p密码
        2）远程登录
            mysql -h主机ip地址 -u用户名 -p密码
    b.图形化界面（sqlyog）

2. 退出
    exit
    quit
```

6.SQL的基本概念

- **结构化查询语言（Structured Query Language）**

> 通过sql指令，可以实现对数据的增删改查
>
> 【CRUD】：create 创建、retrieve（read）检索、update 修改、delete 删除

- **SQL是一套标准，所有的数据库厂商都实现了此标准；但是各自厂商在此标准上增加了特有的语句，这部分内容我们称为方言。**

```
- 我们在学习sql时，需要关注每一家数据库厂商特有的方言
- 例如：MySQL方言注释 #
```

- **SQL书写规范**

```
1. sql语句可以单行或多行书写，最后以分号结尾

2. sql语句（在windows平台）不去分大小写，建议关键字大写
    SELECT * FROM student;

3. 注释
    单行
        -- 所有数据库厂商支持的注释
        #  mysql特有的方言
    多行
        /*  注释内容 */
```

- **SQL分类**

```
1. DDL(Data Definition Language)数据定义语言
        用来定义数据库对象：数据库，表，列等。关键字：create,drop,alter等

2. DML(Data Manipulation Language)数据操作语言
        用来对数据库中表的数据进行增删改。关键字：insert,delete, update等

3. DQL(Data Query Language) 数据查询语言
        用来查询数据库中表的记录(数据)。关键字：select, where等

-------------------------------------------------

4. DCL(Data Control Language)数据控制语言
        用来定义数据库的访问权限和安全级别，及创建用户。关键字：grant,revoke等

5. TCL(Transaction Control Language) 事务控制语言
        用于控制数据库的事务操作，关键字; commit,savepoint,rollback等
```

7.SQL基础操作

1.DDL(Data Definition Language)

- 用来定义数据库对象：数据库，表，列等
  - **关键字:CREATE,DROP,ALTER等**

数据库 库的相关操作:

C:创建数据库

```
1.直接创建数据库
-- 语法:
     CREATE DATABASE 数据库名
-- 实例:
     CREATE DATABASE mydatabase;

1.创建数据库的同时并指定字符集
-- 语法:
     CREATE DATABASE 数据库名 CHARSET 字符集;
-- 实例:
     CREATE DATABASE mydatabase CHARSET utf8;
```

R:查询

```
1. 查看所有数据库
    语法:
        SHOW DATABASES;

2. 查看建库语句
    语法:
        SHOW CREATE DATABASE 数据库名;
    实例:
        SHOW CREATE DATABASE mydatabase;
```

U:修改

```
1. 修改数据库字符集
    语法：
        ALTER DATABASE 数据库名 CHARSET 新字符集;
    实例：
        ALTER DATABASE mydatabase CHARSET gbk;
```

D:删除数据库

```
1. 直接删除数据库
    语法：
        DROP DATABASE 数据库名;
    实例：
        DROP DATABASE mydatabase;
2.若存在 则删除
    语法：
        DROP DATABASE  IF EXISTS 数据库名;
    实例：
        DROP DATABASE  IF EXISTS mydatabase;
```

使用数据库

```
1. 进入/切换到某一个指定数据库
    语法：
        USE 数据库名;
    实例：
        USER mydatabase;

2. 查看当前所在数据库
    语法：
        SELECT DATABASE();
```

数据库表的相关操作

- 创建表，显示表，修改表，删除表操作

C:创建表

```
1.创建表语法:
    CREATE TABLE 表名(
    列名(也称为字段名) 数据类型 [COMMENT 说明],              -- 注意使用的是"," 进行分隔
    列名(也称为字段名) 数据类型 [COMMENT 说明],
    列名(也称为字段名) 数据类型 [COMMENT 说明],
    ...                                        -- 最后一列没有","
    )
实例:
    CREATE TABLE t_test(
    id int,
    name varchar(20),
    age int
    )
```

- 常用的字段数据类型

```
int：整型
float：单精度浮点型
double：双精度浮点型
decimal：大数据的浮点型（保留精准度）
            decimal(m,n) 指定范围
            m:总长度
            n：小数位长度
        例如：
            decimal(5,2)
                999.99 最大值
                0.01   最小值
varchar：字符串
        varchar(n) 指定字符串最大的长度 n：1~65535
            包含字母、符号、汉字
text：文本型
date：日期型 （年月日）
datetime：日期时间型 （年月日时分秒）
```

- 克隆表

```
创建新表时可以克隆旧表的数据结构（字段和类型） 不会复制数据
    语法：
        CREATE TABLE 新表名 LIKE 旧表名;
    实例：
        CREATE TABLE t_test2 LIKE t_test;
```

R:查询表(与表内数据无关)

```
1. 查看当前数据库所有表
    语法：
        SHOW TABLES;      -- 与所有数据库的查看语法一致

2. 查看建表语句
    语法：
        SHOW CREATE TABLE 表名;
    实例：
```

```
        SHOW CREATE TABLE t_test;

3. 查看表结构
    语法：
        DESC 表名;
    实例：
        DESC t_test;
```

U:修改表(表结构字段相关与表内数据无关)

```
-- 1. 添加一列
    语法：
        ALTER TABLE 表名 ADD 列名 数据类型;
    实例：
        ALTER TABLE t_test ADD hobby VARCHAR(200);

-- 2. 修改列数据类型
    语法：
        ALTER TABLE 表名 MODIFY 列名 数据类型;
    实例：
        ALTER TABLE t_test MODIFY hobby VARCHAR(300);

-- 3. 修改列名
    语法：
        ALTER TABLE 表名 CHANGE 旧列名 新列名 数据类型;
    实例：
        ALTER TABLE t_test CHANGE hobby hobbies VARCHAR(20);

-- 4. 删除指定列
    语法：
        ALTER TABLE 表名 DROP 列名;
    实例：
        ALTER TABLE 表名 DROP 列名;

-- 5. 修改表字符集
    语法：
        ALTER TABLE 表名 CHARSET 字符集;
    实例：
        ALTER TABLE t_test CHARSET gbk;

---------------------------

6. 修改表名
    语法：
        RENAME TABLE 旧表名 TO 新表名;
    实例：
        RENAME TABLE t_test TO test;

```

D: 删除数据表

```
1. 直接删除表
    语法：
        DROP TABLE 表名;
    实例：
        DROP TABLE test;

2.若存在 则删除
    语法：
        DROP TABLE IF EXISTS 表名;
    实例：
        DROP TABLE  IF EXISTS test;
```

2.DML(Data Manipulation Language)

- 用来对数据库中**表的数据进行增删改。**关键字：insert,delete, update等

添加表数据

```
1. 语法：
        -- 往指定列中添加数据 值与一一列对应
        INSERT INTO 表名(列名1,列名2,....) VALUES(值1,值2,...);

        -- 向所有列添加数据
        -- 要求：值插入的顺序跟表结构一致 每一列都需要指定数据
        -- 查看：desc 表名
        INSERT INTO 表名 VALUES(值1,值2,....);


2. 注意：
    -- 列名和值要求类型要求对应
    -- 字符串类型数据可以使用单双引，推荐单引号
    -- 字符串可以插入一切（任意）类型，MySQL底层实现隐式转换
    -- 如果添加的字段与关键字冲突了，可以使用反引号(漂号)包裹起来 消除关键字识别
        例如：`name` `order` `user`

3. 练习
        INSERT INTO t_test2 ( id, NAME) VALUES (1,'陈平安');
        INSERT INTO t_test2  VALUES (4,'马苦玄',15);

4. 补充
    同时添加多条记录
        INSERT INTO t_test2  VALUES (5,'李胜昔',20),(6,'崔城',88);
```

- **蠕虫复制【了解】**

```
1. 作用：将一张表的记录，快速复制到另外一张表

2. 应用场景：数据迁移，开发中很少使用...

3. 需求：创建一张stu新表，表结构需要跟student一张，实现数据的迁移

4. 步骤：
    1.克隆表
        CREATE TABLE new_test LIKE t_test2;
    2.蠕虫复制
        INSERT INTO new_test SELECT * FROM t_test2;
```

修改数据

```
1. 语法：
        UPDATE 表名 SET 列名1=新值1,列名2=新值2 ... [WHERE条件]

2. 解释
        [] 里面的内容可写可不写,不写条件则修改表中全部记录

3. 实例：
        UPDATE new_test SET age=18 ;    -- 表中所有age字段的值都变为18;

        UPDATE new_test SET age=18 WHERE NAME='陈平安';  -- 仅陈平安age修改为
15;
```

删除数据

```
1. 语法：
        DELETE FROM 表名 [WHERE条件]

2. 解释
        [] 里面的内容可写可不写,,不写条件则删除表中全部记录

3. 实例
        DELETE FROM t_test2;    /*【开发中杜绝使用...】*/

        DELETE FROM new_test WHERE age=15;


4. 摧毁表，重构表【了解】
    先把这张表删除，再创建相同结构的新表   -- **注意**:表中数据将会被清空
        TRUNCATE TABLE 表名;
        实例:
        TRUNCATE TABLE new_test;
```

- **小结**

> - 新增记录：insert into 表名
> - 修改记录：update 表名

- 删除记录：delete from 表名

DQL(Data Query Language)

1.简单查询

- 数据准备

```
-- 创建表
CREATE TABLE student1(
    id INT,
    NAME VARCHAR(20),
    chinese DOUBLE,
    english DOUBLE,
    math DOUBLE
);
-- 插入记录
INSERT INTO student1(id,NAME,chinese,english,math) VALUES(1,'tom',89,78,90);
INSERT INTO student1(id,NAME,chinese,english,math)
VALUES(2,'jack',67,98,56);
INSERT INTO student1(id,NAME,chinese,english,math)
VALUES(3,'jerry',87,78,77);
INSERT INTO student1(id,NAME,chinese,english,math)
VALUES(4,'lucy',88,NULL,90);
INSERT INTO student1(id,NAME,chinese,english,math)
VALUES(5,'james',82,84,77);
INSERT INTO student1(id,NAME,chinese,english,math)
VALUES(6,'jack',55,85,45);
INSERT INTO student1(id,NAME,chinese,english,math) VALUES(7,'tom',89,65,30);
```

- **基础查询语法**

```
1. 语法：
        SELECT * FROM 表名;      -- 查询表中所有数据
        SELECT 列名1,列名2.... FROM 表名; -- 查询表中指定字段数据

2. 去重关键字
        SELECT DISTINCT 列名 FROM 表名;
        -- 注意：多列去重，要求内容完全一致 即只可去除查询结果中完全相同的表记录

3. 在查询语句中进行运算，不会影响原表中的数据..
        例:
        SELECT math+10 FROM student1;

4.NULL 值参数学运算结果还是 NULL
    IFNULL() 函数
        IFNULL(列名,默认值) 如果该有字段为null，就可以指定你自定义的默认值

    例: SELECT NAME ,IFNULL(english,0) FROM student1;  -- 若english= null 查
询时替换为0


5. 设置别名
        SELECT 列名 [AS] 列别名 FROM 表名 [AS] 表别名;  -- AS 可以省略不写
```

```
        -- 多列查询 使用"," 隔开即可
        例:
        SELECT NAME n ,IFNULL(english,0) e FROM student1

```

- **实例**

```
-- 查询表中所有学生的信息
SELECT * FROM student1;

-- 查询表中所有学生的姓名和对应的语文成绩
SELECT `name`,chinese FROM student1;

-- 查询表中学生姓名（去重）
SELECT `name` FROM student1;
SELECT DISTINCT `name` FROM student1;
SELECT DISTINCT `name`,chinese FROM student1;

-- 在所有学生数学分数上加10分特长分
SELECT `name`,math+10 FROM student1;

-- 统计每个学生的总分
SELECT IFNULL(english,0) FROM student1;
SELECT `name`,chinese+IFNULL(english,0)+math FROM student1;


-- 使用别名表示学生总分
SELECT `name` AS 姓名 ,chinese+IFNULL(english,0)+math AS 总分 FROM student1;

SELECT s.name FROM student1 AS s;
```

2.单表条件查询

- 数据准备

```
-- 创建表
CREATE TABLE student2 (
  id int,
  name varchar(20),
  age int,
  sex varchar(5),
  address varchar(100),
  math int,
  english int
);
-- 插入记录
INSERT INTO student2(id,NAME,age,sex,address,math,english) VALUES
(1,'马云',55,'男','杭州',66,78),
(2,'马化腾',45,'女','深圳',98,87),
(3,'马景涛',55,'男','香港',56,77),
(4,'柳岩',20,'女','湖南',76,65),
(5,'柳青',20,'男','湖南',86,NULL),
(6,'刘德华',57,'男','香港',99,99),
```

```
(7,'马德',22,'女','香港',99,99),
(8,'德玛西亚',18,'男','南京',56,65);
```

- **语法**

```
1. 语法
        SELECT ... FROM 表名 WHERE 条件;

2. 关系（比较）运算符
        = , != , > , < , >= , <=

3. 逻辑运算符
        &&  and（条件同时成立）
        ||  or（条件满足一个）
        !   not（条件取反）

4. IN关键字（某一列，查询多个值）
        SELECT ... FROM 表名 WHERE 列名 IN (值1,值2,值3....);

5.  BETWEEN关键字（范围查询）
        SELECT ... FROM 表名 WHERE 列名 BETWEEN 较小的值 AND 较大的值;

6. NULL值判断
    IS NULL     查询NULL值数据
    IS NOT NULL 查询非NULL值数据

7. LIKE关键字（模糊匹配）
        SELECT ... FROM 表名 WHERE 列名 LIKE '通配符字符串';
        _ 单个任意字符
        % 多个任意字符
```

- **实例**

```
# 关系运算符 ===============================================

-- 查询math分数大于80分的学生
SELECT * FROM student2 WHERE math > 80;

-- 查询english分数小于或等于80分的学生
SELECT * FROM student2 WHERE english <= 80;

-- 查询age等于20岁的学生
SELECT * FROM student2 WHERE age = 20;

-- 查询age不等于20岁的学生
SELECT * FROM student2 WHERE age != 20;

# 逻辑运算符 ===============================================

-- 查询age大于35 且 性别为男的学生(两个条件同时满足)
SELECT * FROM student2 WHERE age > 35 AND sex = '男';

-- 查询age大于35 或 性别为男的学生(两个条件其中一个满足)
SELECT * FROM student2 WHERE age > 35 OR sex = '男';

-- 查询id是1或3或5的学生
```

```sql
SELECT * FROM student2 WHERE id = 1 OR id = 3 OR id = 5;
-- in关键字

-- 再次查询id是1或3或5的学生
SELECT * FROM student2 WHERE id IN (1,3,5);

-- 查询id不是1或3或5的学生
SELECT * FROM student2 WHERE id NOT IN (1,3,5);

-- 查询english成绩大于等于77，且小于等于87的学生
SELECT * FROM student2 WHERE english >=77 AND english <=87;

SELECT * FROM student2 WHERE english BETWEEN 77 AND 87;

-- 查询英语成绩为null的学生
SELECT * FROM student2 WHERE english IS NULL;

-- 查询英语成绩为非null的学生
SELECT * FROM student2 WHERE english IS NOT NULL;


# like模糊匹配 ========================================

-- 查询姓马的学生
SELECT * FROM student2 WHERE  `name` LIKE '马%';

-- 查询姓名中包含'德'字的学生
SELECT * FROM student2 WHERE  `name` LIKE '%德%';

-- 查询姓马，且姓名有三个字的学生
SELECT * FROM student2 WHERE  `name` LIKE '马__';
```

MySQL查询&约束&多表

1.DQL高级查询

数据准备

```sql
-- DQL语句 单表查询
create database web19;

use web19;

-- 创建表
CREATE TABLE student (
  id int,
  name varchar(20),
  age int,
  sex varchar(5),
  address varchar(100),
  math int,
  english int
);
```

```sql
-- 插入记录
INSERT INTO student(id,NAME,age,sex,address,math,english) VALUES
(1,'马云',55,'男','杭州',66,78),
(2,'马化腾',45,'女','深圳',98,87),
(3,'马景涛',55,'男','香港',56,77),
(4,'柳岩',20,'女','湖南',76,65),
(5,'柳青',20,'男','湖南',86,NULL),
(6,'刘德华',57,'男','香港',99,99),
(7,'马德',22,'女','香港',99,99),
(8,'德玛西亚',18,'男','南京',56,65),
(9,'唐僧',25,'男','长安',87,78),
(10,'孙悟空',18,'男','花果山',100,66),
(11,'猪八戒',22,'男','高老庄',58,78),
(12,'沙僧',50,'男','流沙河',77,88),
(13,'白骨精',22,'女','白虎岭',66,66),
(14,'蜘蛛精',23,'女','盘丝洞',88,88);
```

1.排序查询

- **语法**

```sql
1. 语法
        SELECT ... FROM 表名 ORDER BY 排序列 [ASC |DESC],排序列 [ASC |DESC]
            ASC 升序（默认值）
            DESC 降序

2. 注意
        /*多字段（列）排序，后面的排序结构是在前面排序的基础之上*/

      -- 例如第一个条件是根据学生成绩降序排序 第二个条件根据年龄排序 成绩相同的 按照年龄升序排序

实例:
-- 查询所有数据,使用总分数降序排序
SELECT NAME,IFNULL(math,0)+IFNULL(english,0) `sum` FROM student
-- 查询所有数据,在总分降序排序的基础上，如果分数相同再以年龄升序排序
SELECT * FROM student  ORDER BY IFNULL(math,0)+IFNULL(english,0) DESC,age ASC
```

2.聚合(分组)函数

- **语法**

```sql
1. 语法
        COUNT(*)：统计个数
        MAX(列名)：最大值
        MIN(列名)：最小值
        SUM(列名)：求和
        AVG(列名)：平均值

实例:
-- 查询学生总数（null值处理）
SELECT COUNT(*) FROM student;
```

```sql
-- 注: COUNT(*) 会计数 某行数据中 有任意列 数据不为null 的行

-- 查询年龄大于40的总数
SELECT COUNT(*) FROM student WHERE age > 40

-- 查询数学成绩总分
SELECT SUM(math) FROM student

-- 查询数学成绩的平均分
SELECT AVG(math) FROM student

-- 查询数学成绩的最高分
SELECT MAX(math) FROM student

-- 查询学生数学成绩最低分
SELECT MIN(math) FROM student
```

3. 分组查询

- **语法**

```sql
1. 语法
        SELECT 分组列 FROM 表名 GROUP by 分组列 HAVING 分组后筛选条件

注意： WHERE 和 HAVING 区别
        WHERE 在分组前进行条件筛选(会在 GROUP BY 之前执行)，不支持聚合函数
        HAVING在分组后进行条件筛选，支持聚合函数

实例:
-- 统计男生
SELECT COUNT(*) FROM student WHERE sex = '男';

-- 统计女
SELECT COUNT(*) FROM student WHERE sex = '女';

# 分组
-- 统计男生女生各多少人，在同一条sql实现...
SELECT COUNT(*) FROM student GROUP BY sex;

-- 查询年龄大于25岁的人,按性别分组,统计每组的人数
SELECT COUNT(*) FROM student GROUP BY sex HAVING age > 25;

-- 查询年龄大于25岁的人,按性别分组,统计每组的人数,并只显示性别人数大于2的数据
-- 错误写法: WHERE 后不可以跟聚合函数
SELECT sex , COUNT(*) FROM student WHERE COUNT(*) > 2 AND age > 25 GROUP BY
sex;

-- 正确写法
SELECT sex , COUNT(*) FROM student WHERE  age > 25 GROUP BY sex HAVING
COUNT(*) > 2;
```

4. 分页查询(此处是mysql方言)

- **语法**

```
1. 语法
        SELECT ... FROM 表名 LIMIT 开始索引，每页显示个数;

2. 索引特点
        索引起始值是0；可以省略，默认值也是0

3. 分页索引公式
        开始索引 = (当前页码-1) × 每页个数

实例:
-- 查询学生表中数据，显示前6条
SELECT * FROM student LIMIT 0,6;
SELECT * FROM student LIMIT 6;

-- 查询学生表中数据，从第三条开始显示，显示6条
SELECT * FROM student LIMIT 2,6;

-- 模拟百度分页，一页显示5条
-- 第一页
SELECT * FROM student LIMIT 0,5;

-- 第二页
SELECT * FROM student LIMIT 5,5;

-- 第三页
SELECT * FROM student LIMIT 10,5;
```

2. 数据库约束

- **约束的作用**

```
* 对表中的数据进行限定,保证数据的正确性、有效性和完整性
```

- **约束的分类**

```
1. PRIMARY KEY : 主键约束 一个表中只允许由一个主键 主键字段的值具有非空和唯一的限定要求

2. UNIQUE:       唯一约束  存在唯一约束的字段的值在同一个表中只能出现一次

3. NOT NULL:     非空约束  不允许存在非空约束的值的字段值为null

4. DEFAULT:      默认值    限定某一列的默认值，再没有指定的情况下所有列的默认值为null

5. FOREGIN KEY:  外键约束  限定 两张表之间的表关系
```

1.主键约束

- 作用：限定某一列的值非空且唯一， 主键就是表中记录的唯一标识（类似身份证号码）

```
1. 语法
 * 创建表:
        CREATE TABLE 表名(
        id INT PRIMARY KEY,
            ...
            ...
        )
 * 已有表: ALTER TABLE 表名 add PRIMARY KEY(主键列)

2. 主键的特点: 一张表中只允许存在最多一个主键约束, 但可以设置联合主键(多个字段共同组成主键)

3.自增器 (通常搭配主键使用 前提是主键列的数据能够自增 如int类型)
* 创建表:
        CREATE TABLE 表名(
        id INT PRIMARY KEY AUTO_INCREMENT,
            ...
            ...
            )
* 指定自增器的起始值: ALTER TABLE 表名 AUTO_INCREMENT = 起始值;

4. 删除主键约束:
        ALTER TABLE 表名 DROP PRIMARY KEY
        -- 注意: 当该主键有自增器,那么不能直接删除主键约束的功能
           解释: 只有拥有主键约束的字段才可以使用自增器

实例:
-- 给student表添加主键约束
ALTER TABLE student add primary key(id);

-- 创建表指定主键约束
CREATE TABLE stu1(
    id INT PRIMARY KEY,
    `name` VARCHAR(20)
)

-- 测试

INSERT INTO stu1 VALUES(1,'jack');

-- Duplicate entry '1' for key 'PRIMARY' 错误：主键不能重复
INSERT INTO stu1 VALUES(1,'lucy');

-- Column 'id' cannot be null 错误：主键不能为空
INSERT INTO stu1 VALUES(NULL,'lucy');

-- 指stu1表，name字段也设置为主键约束
-- Multiple primary key defined 错误：主键定义多值
ALTER TABLE stu1 ADD PRIMARY KEY(`name`);

-- 创建联合主键表
CREATE TABLE stu2(
    id INT,
```

```
    `name` VARCHAR(5),
    PRIMARY KEY(id,name)  -- 联合主键 有id和那么组合成的值在表中唯一
    )
-- 测试
INSERT INTO stu2 VALUES(1,'陈平安');

INSERT INTO stu2 VALUES(2,'顾璨');

INSERT INTO stu2 VALUES(3,'隐官');

-- Duplicate entry '1-陈平安' for key 'PRIMARY' 错误
INSERT INTO stu2 VALUES(1,'陈平安')


-- 创建主键约束与自增器
CREATE TABLE stu4(
    id INT PRIMARY KEY AUTO_INCREMENT,
    `name` VARCHAR(20)
    )

-- 测试
INSERT INTO stu4 VALUES(1,'陈平安');

INSERT INTO stu4 VALUES(null,'顾璨');

-- 删除表内所有数据
DELETE FROM stu4;

-- 插入数据
INSERT INTO stu4 VALUES(NULL,'陈平安'); -- 此时id为 3
-- 重构表
TRUNCATE TABLE stu4; -- 删除表 然后重构表

-- 插入数据
INSERT INTO stu4 VALUES(NULL,'陈平安');  -- 此时id为 1

-- 移除自增器(修改id类型为 int 即可)
ALTER TABLE stu4 MODIFY id INT;

INSERT INTO stu4 VALUES(NULL,'陈平安') -- 报错 Column 'id' cannot be null 说明自增器已移除
-- 删除主键约束
ALTER TABLE stu4 DROP PRIMARY KEY;

INSERT INTO stu4 VALUES(1,'陈平安');  -- 添加重复id 成功 说明主键约束已移除

```

2. 唯一约束

- 作用: 限定某一列的值不能够出现重复 不能限定null(对插入为值null时无效 可以出现多个null值)

```
语法:
    CREATE TABLE 表名(
        列名 数据类型 UNIQUE,
        ...
```

```
        ...
    )

实例:

CREATE TABLE stu5(
 id INT PRIMARY KEY AUTO_INCREMENT,
 `name` VARCHAR(32) UNIQUE
);

-- 测试
INSERT INTO stu5 VALUES(1,'jack');

-- Duplicate entry 'jack' for key 'name' 错误:用户名不能重复
INSERT INTO stu5 VALUES(2,'jack');

INSERT INTO stu5 VALUES(3,NULL);

INSERT INTO stu5 VALUES(4,NULL); -- 成功
```

3.非空约束

- 作用: 限定某一列的值不能为null

```
1. 创建表时指定字段为非空约束
        CREATE TABLE 表名(
            列名 数据类型 NOT NULL, -- 非空约束
            列名 数据类型 UNIQUE NOT NULL -- (唯一+非空) 约束
        );

实例:
CREATE TABLE stu6(
    id INT PRIMARY KEY AUTO_INCREMENT,
    `name` VARCHAR(10)  UNIQUE NOT NULL
    )
-- 测试
INSERT INTO stu6 VALUES(1,'jack');

-- 错误 Duplicate entry 'jack' for key 'name'
INSERT INTO stu6 VALUES(2,'jack');

-- 错误 Column 'name' cannot be null
INSERT INTO stu6 VALUES(3,NULL);

```

注:唯一 + 非空 != 主键 一张表只能有一个主键 但是唯一非空列可以设置多个

4.默认值

- 作用:限定某一列的默认值,在没有指定的情况下 默认为null

```
1. 创建表时指定默认字段
CREATE TABLE 表名(
```

```
    列名 数据类型 DEFAULT 默认值,
    ...
    ...
    )

实例:
CREATE TABLE stu7(
    id INT PRIMARY KEY AUTO_INCREMENT,
    `name` VARCHAR(32) DEFAULT NULL,
    sex VARCHAR(5) DEFAULT '男'
    )

INSERT INTO stu7(id,`name`) VALUES(1,'小张'); -- 插入后 sex字段会默认赋值 男

INSERT INTO stu7(id,`name`,sex) VALUES(2,'小宋','女'); -- 赋值后 sex字段值 会编程所赋的值

INSERT INTO stu7(id,`name`,sex) VALUES(3,'小李',NULL); -- 赋值为NULL ,sex字段的值也会变为 NULL

INSERT INTO stu7(id,`name`) VALUES(4,'小王');
```

3.多表关系

- 通过某中途径(外键约束) 使得表与表之间产生关联
- 常见多表关系

```
1. 一对多
    应用场景:
        班级和学生、部门和员工
    解释:
        一个班级内有多名学生，多名学生属于某一个班级

2. 多对多
    应用场景:
        老师和学生、学生和课程
    解释:
        一名老师可以教授多名同学，一名同学可以被多个老师教学

3. 一对一
    应用场景:
        公民和身份证号、公司和注册地
    解释:
        一个公民只能有一个身份证号，一个身份证号对应一个公民
```

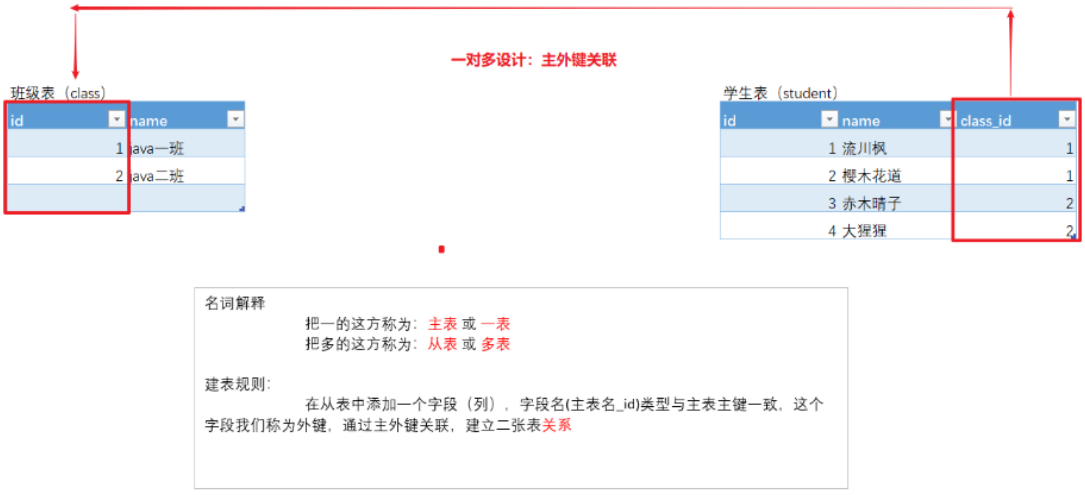
外键约束

- 作用: 限定两张表有关系的数据,保证数据的正确性,有效性和完整性

```
1. 在从表中添加外键约束
    a）创建表
    CREATE TABLE 表名(
            外键列 数据类型,
            [CONSTRAINT] [约束名] FOREIGN KEY(外键列) REFERENCES 主表(主键列)
        );
    b）已有表
    ALTER TABLE 表名 ADD [CONSTRAINT] [约束名] FOREIGN KEY(外键) REFERENCES 主表(主键);

2. 特点
    1）主表不能删除 从表 已引入的数据
    2）从表不能添加 主表 未拥有的数据
    3）先删除从表数据 再删除主表数据
    4）先添加主表数据 再添加从表数据
    5）外键约束字段可以为空但不能错

3. 补充在企业开发时，我们现在很少使用外键约束，影响数据库性能..
    我们是java工程师，可以在java中进行校验判断...

4. 删除外键约束
        ALTER TABLE 表名 DROP FOREIGN KEY 约束名;
```

1.一对多

```
* 举例: 班级和学生
```

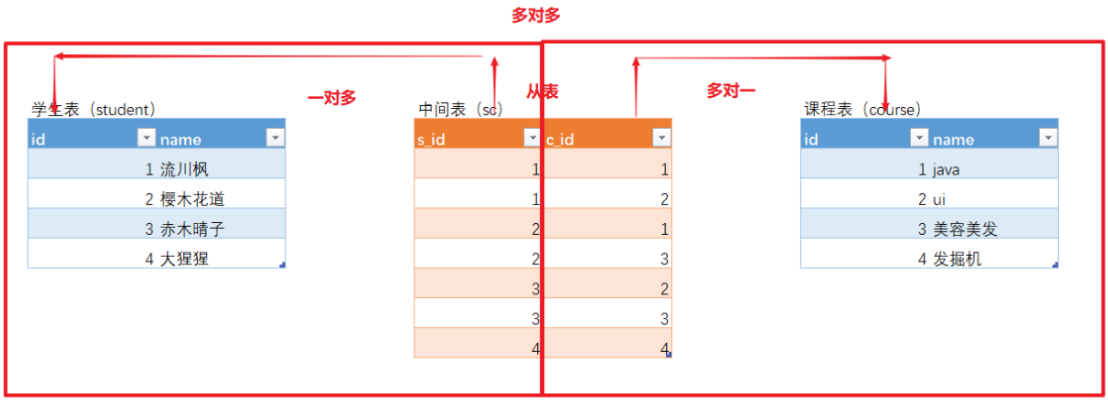



```
-- 创建数据库
CREATE DATABASE web19_pro;
USE web19_pro;

-- 一对多


```

```sql
-- 主表（班级）
CREATE TABLE class(
  id INT PRIMARY KEY AUTO_INCREMENT,
  `name` VARCHAR(32)
);

INSERT INTO class VALUES(1,'java一班');
INSERT INTO class VALUES(2,'java二班');

-- 从表（学生）
CREATE TABLE student(
   id INT PRIMARY KEY AUTO_INCREMENT,
   `name` VARCHAR(32),
   class_id INT
);

INSERT INTO student VALUES(1,'流川枫',1);
INSERT INTO student VALUES(2,'樱木花道',1);
INSERT INTO student VALUES(3,'赤木晴子',2);
INSERT INTO student VALUES(4,'大猩猩',2);

-- 给学生表外键添加约束
ALTER TABLE student ADD CONSTRAINT class_id_fk FOREIGN KEY(class_id)
REFERENCES class(id);

-- 删除外键约束
ALTER TABLE student DROP FOREIGN KEY class_id_fk;
```

2. 多对多

```
* 举例 : 学生和课程
```



建表原则：由二个一对多组成

创建第三张表（从表），又称为中间表，在中间表中添加二个外键字段分别指向各自的主键，通常我们会把这个外键作为联合主键

```sql
-- 多对多
```

```sql
-- 主表（学生，上午已创建，可以直接使用...）

-- 主表（课程）
CREATE TABLE course(
 id INT PRIMARY KEY AUTO_INCREMENT,
 `name` VARCHAR(32)
);
INSERT INTO course VALUES(1,'java');
INSERT INTO course VALUES(2,'ui');
INSERT INTO course VALUES(3,'美容美发');
INSERT INTO course VALUES(4,'挖掘机');

-- 从表（中间表）
CREATE TABLE sc(
 s_id INT,
 c_id INT,
 PRIMARY KEY (s_id,c_id)
);
INSERT INTO sc VALUES(1,1);
INSERT INTO sc VALUES(1,2);
INSERT INTO sc VALUES(2,1);
INSERT INTO sc VALUES(2,3);
INSERT INTO sc VALUES(3,2);
INSERT INTO sc VALUES(3,3);
INSERT INTO sc VALUES(4,4);

-- 测试联合主键
-- Duplicate entry '1-1' for key 'PRIMARY'
INSERT INTO sc VALUES(1,1);

-- 中间表添加外键约束
ALTER TABLE sc ADD FOREIGN KEY(s_id) REFERENCES student(id);
ALTER TABLE sc ADD FOREIGN KEY(c_id) REFERENCES course(id);

-- 流川枫选修一个不存在的课程
INSERT INTO sc VALUES(1,10);
```

3. 一对一

- 实际开发过程中使用较少 因为可以将这种一对一的关系设置在一张表中进行简化

```
* 商品详情
```

```sql
-- 商品（主表）
CREATE TABLE product(
 id INT PRIMARY KEY AUTO_INCREMENT,
 `name` VARCHAR(32)
);
INSERT INTO product VALUES(1,'华为p40');
INSERT INTO product VALUES(2,'一加8');
INSERT INTO product VALUES(3,'小米10');

-- 详情（从表）
CREATE TABLE `desc`(
  id INT PRIMARY KEY AUTO_INCREMENT,
   `name` VARCHAR(32),
   FOREIGN KEY(id) REFERENCES product(id)
);
INSERT INTO `desc` VALUES(1,'华为xxx');
INSERT INTO `desc` VALUES(2,'一加xxx');
INSERT INTO `desc` VALUES(3,'小米xxx');

INSERT INTO `desc` VALUES(4,'三星xxx'); -- 插入失败
```

多表查询

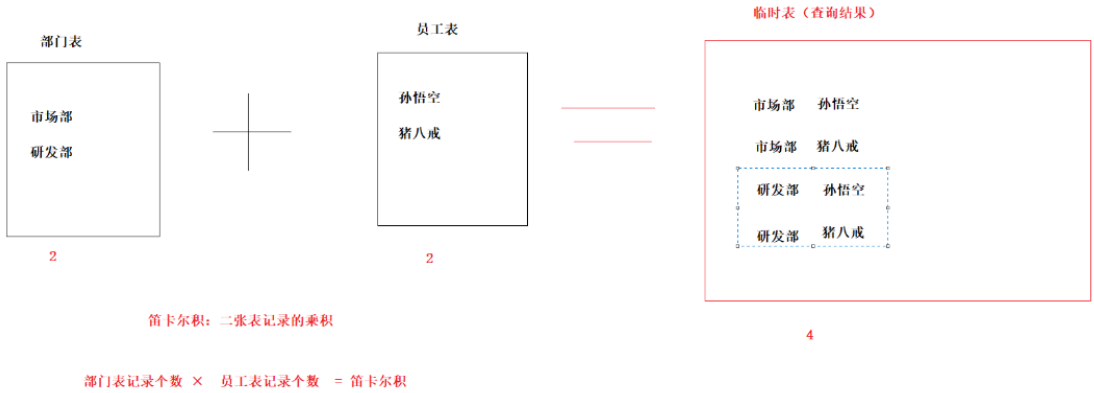
- **数据准备**

```sql
-- 多表查询
CREATE DATABASE web20;
USE web20;
-- 创建部门表（主表）
CREATE TABLE dept (
  id INT PRIMARY KEY AUTO_INCREMENT,
  NAME VARCHAR(20)
);

INSERT INTO dept (NAME) VALUES ('开发部'),('市场部'),('财务部'),('销售部');

-- 创建员工表（从表）
CREATE TABLE emp (
  id INT PRIMARY KEY AUTO_INCREMENT,
  NAME VARCHAR(10),
  gender CHAR(1),   -- 性别（sex）
  salary DOUBLE,   -- 工资
  join_date DATE,  -- 入职日期
  dept_id INT -- 外键字段
);

INSERT INTO emp(NAME,gender,salary,join_date,dept_id) VALUES('孙悟空','男',7200,'2013-02-24',1);
INSERT INTO emp(NAME,gender,salary,join_date,dept_id) VALUES('猪八戒','男',3600,'2010-12-02',2);
INSERT INTO emp(NAME,gender,salary,join_date,dept_id) VALUES('唐僧','男',9000,'2008-08-08',2);
INSERT INTO emp(NAME,gender,salary,join_date,dept_id) VALUES('白骨精','女',5000,'2015-10-07',3);
```

```
INSERT INTO emp(NAME,gender,salary,join_date,dept_id) VALUES('蜘蛛精','女',4500,'2011-03-14',1);
INSERT INTO emp(NAME,gender,salary,join_date,dept_id) VALUES('沙僧','男',6666,'2017-03-04',null);
```

1.笛卡尔积(了解)

```
* 功能
        多张表记录进行乘积组合，这种现象称为笛卡尔积（交叉连接）
* 语法
        SELECT ... FROM 左表,右表;
```




```
-- 笛卡尔积
SELECT * FROM dept,emp;

SELECT COUNT(*) FROM dept,emp;
```

2. 内连接

```
作用:
    使用左表的记录去匹配右表的记录,显示符合条件的部分(只能显示两张表的交集)

语法:
    1. 隐式内连接
        SELECT ... FROM 左表,右表 WHERE 连接条件;
    2.显示内连接(推荐)
        SELECT ... FROM 左表[INNER] JOIN 右表 ON 连接条件 [WHERE] [业务条件];

-- 显示内连接与隐式内连接的区别
    相对而言，隐式连接好理解好书写，语法简单，担心的点较少。
    但是显式连接可以减少字段的扫描，有更快的执行速度。这种速度优势在3张或更多表连接时比较明显

-- ON 后面一般只跟连接条件(主要指两张表的关联信息) 多表联查（两张表及以上）WHERE 条件拼接在最后
```

实例:

```sql
-- 查询开发部的部门信息
-- 隐式
SELECT * FROM dept d, emp e WHERE d.id = e.dept_id AND d.name='开发部';

-- 显示
SELECT * FROM dept d JOIN emp e ON d.id = e.dept_id AND d.name='开发部';

-- 查询唐僧的 id，姓名，性别，工资和所在部门名称

SELECT e.id, e.name, e.gender, e.salary, d.name FROM dept d JOIN emp e ON d.id=e.dept_id AND e.name='唐僧'; -- 可以查询到结果 但不建议写 尽量不在ON后面跟业务条件

SELECT e.id,e.name,e.gender,e.salary,d.name FROM emp e INNER JOIN dept d ON e.dept_id = d.id WHERE e.name ='唐僧'; -- 正确语法
```

3.外连接

- 通常用于查询多表时 展示某张表的全部信息

```
1. 左外连接【推荐】
    功能
        展示左表全部，再去匹配右表，若符合条件显示数据，不符合显示null
    语法
        SELECT ... FROM 左表 LEFT [OUTER] JOIN 右表 ON 连接条件 [WHERE] [业务条件];

2. 右外连接
    功能
        展示右表全部，再去匹配左表，若符合条件显示数据，不符合显示null
    语法
        SELECT ... FROM 左表 LEFT [OUTER] JOIN 右表 ON 连接条件 [WHERE] [业务条件];

右外连接是相对于左的 选则一种即可 两者区别是 数据的显示 写在前面的会在左边展示(不影响数据)
```

实例:

```sql
# 左外连接
-- 查询所有员工信息及对应的部门名称
SELECT e.*, d.name FROM emp e LEFT JOIN dept d ON d.id = e.dept_id ;
-- 查询所有部门及对应的员工信息
SELECT * FROM dept d LEFT JOIN emp e ON d.id = e.dept_id;

# 右外连接
-- 查询所有部门及对应的员工信息
SELECT * FROM emp e RIGHT JOIN dept d ON d.id = e.dept_id;
```

4.子查询(嵌套)

```
* 功能
        一条select查询结果，作为另一条select语法的一部分
* 语法
    1）查询结果单值（使用=）
        SELECT MAX(salary) FROM emp;
    2）查询结果单列（使用in）
        SELECT salary FROM emp;
    3）查询结果多列
        SELECT * FROM emp;

* 规律
    子查询结果单列或单值，一般作为条件在where后面使用
        SELECT ... FROM 表名 WHERE 字段 IN | = (子查询);

    子查结果为多列，一般作为虚拟表在from后面使用
        SELECT ... FROM (子查询) [AS] 表别名;  -- 多列表必须有别名
```

id	NAME	gender	salary	join_date	dept_id
1	孙悟空	男	7200	2013-02-24	1
4	白骨精	女	5000	2015-10-07	3
5	蜘蛛精	女	4500	2011-03-14	1
6	沙僧	男	6666	2017-03-04	(NULL)

id	NAME
1	开发部
2	市场部
3	财务部
4	销售部
(Auto)	NULL)

- 实例

```sql
-- 子查询
# 子查询结果为单值
-- 1 查询工资最高的员工是谁？
SELECT * FROM emp WHERE salary = (SELECT MAX(salary) FROM emp);

-- 2 查询工资小于平均工资的员工有哪些？
SELECT * FROM emp WHERE salary < (SELECT AVG(salary) FROM emp);

# 子查询结果为单列多行
-- 1 查询工资大于5000的员工，来自于哪些部门的名字
SELECT NAME FROM dept WHERE id IN (SELECT dept_id FROM emp WHERE salary > 5000);

-- 2 查询开发部与财务部所有的员工信息
SELECT * FROM emp WHERE dept_id IN (SELECT id FROM dept WHERE name in ('开发部','财务部'));

# 子查询结果为多列多行
-- 1 查询出2011年以后入职的员工信息，包括部门名称
SELECT * FROM (SELECT * FROM emp WHERE join_date >='2011-01-01') e JOIN dept d ON d.id = e.dept_id;

```

5. 自关联查询

```
作用:
 将一张表作为两张表来查 使用表中字段有从属关系的表 如部门表 子部门 父部门, 菜单表 子菜单 父菜单
```

- 数据准备

```sql
CREATE TABLE menu(
id INT PRIMARY KEY AUTO_INCREMENT,
`name` VARCHAR(20),
parentMenuId INT
)

INSERT  INTO `menu`(`id`,`name`,`parentMenuId`) VALUES
(1,'工作台',NULL),
(2,'仪表盘管理',1),
(3,'会员管理',NULL),
(4,'会员档案',3),
(5,'体检上传',3),
(6,'会员统计',3),
(7,'预约管理',NULL),
(8,'预约列表',7),
(9,'预约设置',7),
(10,'套餐管理',7),
(11,'检查组管理',7),
(12,'检查项管理',7),
(13,'健康评估',NULL),
(14,'中医体质辨识',13),
(15,'统计分析',NULL),
(16,'会员数量',15),
(17,'系统设置',NULL),
(18,'菜单管理',17),
(19,'权限管理',17),
(20,'角色管理',17),
(21,'用户管理',17),
(22,'套餐占比',15),
(23,'运营数据',15),(24,'地址管理',1);

```

- 语法

```sql
SELECT ... FROM 实体表名 别名1,实体表名 别名2 [WHERE] [条件]


实例:
-- 查询'会员管理'下的子菜单信息
SELECT * FROM menu m1, menu m2 WHERE m1.parentMenuId = m2.id AND m2.name='会员管理';
```

6. 多表查询案例

- 数据准备

```sql
-- 多表案例
create database web20_pro;
use web20_pro;
-- 部门表
CREATE TABLE dept (
  id INT PRIMARY KEY auto_increment, -- 部门id
  dname VARCHAR(50), -- 部门名称
  loc VARCHAR(50) -- 部门位置
);

-- 添加4个部门
INSERT INTO dept(id,dname,loc) VALUES
(10,'教研部','北京'),
(20,'学工部','上海'),
(30,'销售部','广州'),
(40,'财务部','深圳');

-- 职务表
CREATE TABLE job (
  id INT PRIMARY KEY,
  jname VARCHAR(20), -- 职务名称
  description VARCHAR(50) -- 职务描述
);

-- 添加4个职务
INSERT INTO job (id, jname, description) VALUES
(1, '董事长', '管理整个公司，接单'),
(2, '经理', '管理部门员工'),
(3, '销售员', '向客人推销产品'),
(4, '文员', '使用办公软件');

-- 员工表
CREATE TABLE emp (
  id INT PRIMARY KEY, -- 员工id
  ename VARCHAR(50), -- 员工姓名
  job_id INT, -- 职务id  外键
  mgr INT , -- 上级领导（自关联）
  joindate DATE, -- 入职日期
  salary DECIMAL(7,2), -- 工资 99999.99
  bonus DECIMAL(7,2), -- 奖金 99999.99
  dept_id INT, -- 所在部门编号  外键
  CONSTRAINT emp_jobid_ref_job_id_fk FOREIGN KEY (job_id) REFERENCES job
(id),
  CONSTRAINT emp_deptid_ref_dept_id_fk FOREIGN KEY (dept_id) REFERENCES dept
(id)
);

-- 添加员工
INSERT INTO emp(id,ename,job_id,mgr,joindate,salary,bonus,dept_id) VALUES
(1001,'孙悟空',4,1004,'2000-12-17','8000.00',NULL,20),
(1002,'卢俊义',3,1006,'2001-02-20','16000.00','3000.00',30),
(1003,'林冲',3,1006,'2001-02-22','12500.00','5000.00',30),
(1004,'唐僧',2,1009,'2001-04-02','29750.00',NULL,20),
```

```
(1005,'李逵',4,1006,'2001-09-28','12500.00','14000.00',30),
(1006,'宋江',2,1009,'2001-05-01','28500.00',NULL,30),
(1007,'刘备',2,1009,'2001-09-01','24500.00',NULL,10),
(1008,'猪八戒',4,1004,'2007-04-19','30000.00',NULL,20),
(1009,'罗贯中',1,NULL,'2001-11-17','50000.00',NULL,10),
(1010,'吴用',3,1006,'2001-09-08','15000.00','0.00',30),
(1011,'沙僧',4,1004,'2007-05-23','11000.00',NULL,20),
(1012,'李逵',4,1006,'2001-12-03','9500.00',NULL,30),
(1013,'小白龙',4,1004,'2001-12-03','30000.00',NULL,20),
(1014,'关羽',4,1007,'2002-01-23','13000.00',NULL,10);

-- 工资等级表
CREATE TABLE salarygrade (
  grade INT PRIMARY KEY,  -- 等级
  losalary INT, -- 最低工资
  hisalary INT -- 最高工资
);

-- 添加5个工资等级
INSERT INTO salarygrade(grade,losalary,hisalary) VALUES
(1,7000,12000),
(2,12010,14000),
(3,14010,20000),
(4,20010,30000),
(5,30010,99990);
```

- 条件查询

```
-- 1 查询所有员工信息。显示员工编号，员工姓名，工资，职务名称，职务描述
SELECT e.id,e.ename,e.salary,j.jname,j.description FROM emp e INNER JOIN job j ON e.job_id = j.id;

-- 2 查询所有员工信息。显示员工编号，员工姓名，工资，职务名称，职务描述，部门名称，部门位置
SELECT e.id,e.ename,e.salary,j.jname,j.description,d.dname,d.loc FROM emp e INNER JOIN job j ON e.job_id = j.id INNER JOIN dept d ON e.dept_id = d.id;

-- 3 查询所有员工信息。显示员工姓名，工资，职务名称，职务描述，部门名称，部门位置，工资等级
SELECT e.ename,e.salary,j.jname,j.description,d.dname,d.loc,s.grade FROM emp e INNER JOIN job j ON e.job_id = j.id INNER JOIN dept d ON e.dept_id = d.id INNER JOIN salarygrade s WHERE e.salary BETWEEN s.losalary AND s.hisalary;

-- 4 查询经理的信息。显示员工姓名，工资，职务名称，职务描述，部门名称，部门位置，工资等级
SELECT e.ename,e.salary,j.jname,j.description,d.dname,d.loc,s.grade FROM emp e INNER JOIN job j ON e.job_id = j.id  INNER JOIN dept d ON e.dept_id = d.id INNER JOIN salarygrade s ON e.salary BETWEEN s.losalary AND s.hisalary WHERE j.jname='经理';

-- 5 查询出部门编号、部门名称、部门位置、部门人数
  SELECT d.id,d.dname,d.loc,COUNT(e.id) FROM dept d LEFT JOIN emp e ON d.id = e.dept_id GROUP BY d.dname;


-- 6 查询每个员工的名称及其上级领导的名称（表自关联查询）
SELECT e.ename,m.ename FROM emp e INNER JOIN emp m ON e.mgr = m.id;
```

函数

数字函数

```
-- 获取x的绝对值
ABS(x)
例:
SELECT ABS(-1);   -- 结果: -1

-- 获取x的二进制（OCT返回八进制，HEX返回十六进制）
BIN(x)
例:
SELECT BIN(2);    -- 结果: 10

-- 向上取整
CEILING(x)
例:
SELECT CEILING(0.6);  -- 结果: 1
SELECT CEILING(-0.6); -- 结果: 0

-- 向下取整
FLOOR(x)
例:
SELECT FLOOR(0.6);  -- 结果: 0
SELECT FLOOR(-0.6); -- 结果: -1

-- 获取e（自然对数的底）的x次方
EXP(x)
例:
SELECT EXP(0);        -- 结果: 1

-- 获取集合中最大的值
GREATEST(x1,x2,...,xn)
例:
SELECT GREATEST(1,1.5,2,2.1,4);  -- 结果4

-- 获取集合中的最小值
LEAST(x1,x2,...,xn)
例:
SELECT LEAST(1,1.5,2,2.1,4);  -- 结果1

-- 获取x的自然对数(以e为底)
LN(x)
例:
SELECT LN(1=); -- 结果 0;

-- 获取以x为底y的对数
LOG(x,y)
例:
SELECT LOG(2,4); -- 结果 2

-- 获取x/y的模（余数）
MOD(x,y)
例:
SELECT MOD(9,4); -- 结果 1

```

```sql
-- 获取圆周率
PI()
例:
SELECT PI();  -- 结果 3.141593

-- 获取 0-1 范围内的随机数 可指定参数,参数可以生产一个固定的数(参数相当于某个随机数的唯一标识)
RAND()
例:
SELECT RAND();  -- 结果: 0-1之间的随机数
SELECT RAND(1); -- 结果: 一个固定的随机数 不会随着运行次数 改变值

-- 获取x四舍五入的结果 y用于指定保留的小数的位数
ROUND(x,y)
例:
SELECT ROUND(5.3523,10); -- 结果: 5.3523000000

-- 获取代表数字x的符号(正负数)的值
SIGN(x)                          -- 返回
例:
SELECT SIGN(10);  -- 结果: 正数为1
SELECT SIGN(0);   -- 零:  0
SELECT SIGN(-1);  -- 负数: -1

-- 获取一个数的平方根
SQRT(x)
例:
SELECT SQRT(4);   -- 结果: 2

-- 获取一个数x 通过y指定保留小数点后保留位数(保留位数不会自动进行四舍五入)
TRUNCATE(x,y)
例:
SELECT TRUNCATE(4.256,2); -- 结果: 4.25
```

聚合函数

```sql
-- col 表示列名

-- 获取指定列的平均值
AVG(col)
例:
SELECT AVG(money) FROM account;  -- 结果: 1000

-- 获取指定列中非NULL值的个数
COUNT(col)
例:
SELECT COUNT(money) FROM account; -- 结果: 2

 -- 获取指定列的最小值
MIN(col)
例:
SELECT MIN(money) FROM account; -- 结果: 600

-- 获取指定列的最大值
MAX(col)                         -- 返回指定列的最大值
```

```
例:
SELECT MAX(money) FROM account; -- 结果: 1400

-- 获取指定列所有的数值之和
例:
SUM(col)
SELECT SUM(money) FROM account; -- 结果: 200

-- 获取某列值 组合(拼接)而成结果
例:
GROUP_CONCAT(col)
SELECT GROUP_CONCAT(money) FROM account; -- 结果: 1400,600
```

字符串函数

```
ASCII(char)            -- 返回字符的ASCII码值

BIT_LENGTH(str)        -- 返回字符串的比特长度

CONCAT(s1,s2...,sn)    -- 将s1,s2...,sn连接成字符串

CONCAT_WS(sep,s1,s2...,sn) -- 将s1,s2...,sn连接成字符串，并用sep字符间隔

INSERT(str,x,y,instr) -- 将字符串str从第x位置开始，y个字符长的子串替换为字符串
instr，返回结果

FIND_IN_SET(str,list) -- 分析逗号分隔的list列表，如果发现str，返回str在list中的位置

LCASE(str)或LOWER(str) -- 返回将字符串str中所有字符改变为小写后的结果

LEFT(str,x)             -- 返回字符串str中最左边的x个字符

LENGTH(s)               -- 返回字符串str中的字符数

LTRIM(str)              -- 从字符串str中切掉开头的空格

POSITION(substr,str)    -- 返回子串substr在字符串str中第一次出现的位置

QUOTE(str)              -- 用反斜杠转义str中的单引号

REPEAT(str,srchstr,rplcstr) -- 返回字符串str重复x次的结果

REVERSE(str)                -- 返回颠倒字符串str的结果

RIGHT(str,x)                -- 返回字符串str中最右边的x个字符

RTRIM(str)                  -- 返回字符串str尾部的空格

STRCMP(s1,s2)               -- 比较字符串s1和s2

TRIM(str)                   -- 去除字符串首部和尾部的所有空格

UCASE(str)                  -- 或UPPER(str) 返回将字符串str中所有字符转变为大写后
的结果
```

日期时间函数

```
CURDATE()或CURRENT_DATE()       -- 返回当前的日期

CURTIME()或CURRENT_TIME()       -- 返回当前的时间

DATE_ADD(date,INTERVAL int keyword) -- 返回日期date加上间隔时间int的结果(int必须按照关键字进行格式化),如: SELECTDATE_ADD(CURRENT_DATE,INTERVAL 6 MONTH);

DATE_FORMAT(date,fmt)                -- 依照指定的fmt格式格式化日期date值

DATE_SUB(date,INTERVAL int keyword) -- 返回日期date加上间隔时间int的结果(int必须按照关键字进行格式化),如: SELECTDATE_SUB(CURRENT_DATE,INTERVAL 6 MONTH);

DAYOFWEEK(date)     -- 返回date所代表的一星期中的第几天(1~7)

DAYOFMONTH(date)  -- 返回date是一个月的第几天(1~31)

DAYOFYEAR(date)    -- 返回date是一年的第几天(1~366)

DAYNAME(date)      -- 返回date的星期名，如: SELECT DAYNAME(CURRENT_DATE);

FROM_UNIXTIME(ts,fmt)  -- 根据指定的fmt格式，格式化UNIX时间戳ts

HOUR(time)          -- 返回time的小时值(0~23)

MINUTE(time)        -- 返回time的分钟值(0~59)

MONTH(date)         -- 返回date的月份值(1~12)

MONTHNAME(date)    -- 返回date的月份名，如: SELECT MONTHNAME(CURRENT_DATE);

NOW()               -- 返回当前的日期和时间

QUARTER(date)       -- 返回date在一年中的季度(1~4)，如SELECT QUARTER(CURRENT_DATE);

WEEK(date)          -- 返回日期date为一年中第几周(0~53)

YEAR(date)          -- 返回日期date的年份(1000~9999)

示例:
获取当前系统时间: SELECT FROM_UNIXTIME(UNIX_TIMESTAMP());

SELECT EXTRACT(YEAR_MONTH FROM CURRENT_DATE);

SELECT EXTRACT(DAY_SECOND FROM CURRENT_DATE);

SELECT EXTRACT(HOUR_MINUTE FROM CURRENT_DATE);

返回两个日期值之间的差值(月数): SELECT PERIOD_DIFF(200302,199802);

在Mysql中计算年龄:

```

```
SELECT DATE_FORMAT(FROM_DAYS(TO_DAYS(NOW())-TO_DAYS(birthday)),'%Y')+0 AS age FROM employee;

这样，如果Brithday是未来的年月日的话，计算结果为0。

下面的SQL语句计算员工的绝对年龄，即当Birthday是未来的日期时，将得到负值。

SELECT DATE_FORMAT(NOW(), '%Y') - DATE_FORMAT(birthday, '%Y') - (DATE_FORMAT(NOW(), '00-%m-%d') <DATE_FORMAT(birthday, '00-%m-%d')) AS age from employee
```

加密函数

```
AES_ENCRYPT(str,key)   -- 返回用密钥key对字符串str利用高级加密标准算法加密后的结果，调用AES_ENCRYPT的结果是一个二进制字符串，以BLOB类型存储

AES_DECRYPT(str,key)   -- 返回用密钥key对字符串str利用高级加密标准算法解密后的结果

DECODE(str,key)        -- 使用key作为密钥解密加密字符串str

ENCRYPT(str,salt)      -- 使用UNIXcrypt()函数，用关键词salt(一个可以惟一确定口令的字符串，就像钥匙一样)加密字符串str

ENCODE(str,key)        -- 使用key作为密钥加密字符串str，调用ENCODE()的结果是一个二进制字符串，它以BLOB类型存储

MD5()                  -- 计算字符串str的MD5校验和

PASSWORD(str)          -- 返回字符串str的加密版本，这个加密过程是不可逆转的，和UNIX密码加密过程使用不同的算法。

SHA()                  -- 计算字符串str的安全散列算法(SHA)校验和

示例：

SELECT ENCRYPT('root','salt');

SELECT ENCODE('xufeng','key');

SELECT DECODE(ENCODE('xufeng','key'),'key');#加解密放在一起

SELECT AES_ENCRYPT('root','key');

SELECT AES_DECRYPT(AES_ENCRYPT('root','key'),'key');

SELECT MD5('123456');

SELECT SHA('123456');
```

控制流函数

- 控制流函数可以用于实现条件操作，这些函数可以实现SQL的条件逻辑，允许开发者将一些应用程序业务逻辑转换到数据库后台

CASE函数

```
-- 用法1 case 和 WHEN 之间无内容
SELECT 字段... ,  -- 注意 "," 不能缺失
CASE
WHEN 条件1
THEN result1  -- WHEN 之后的条件为 true 则返回当前WHEN对应的THEN之后的数据 WHEN-THEN成对使用
WHEN 条件2
THEN result2
...
WHEN 条件n
THEN resultn
ELSE          -- 若WHEN之后的所有条件都不满足 则返回 ELSE后的数据
defultResult
END -- case函数结束标识
FROM 表名 [WHERE].............

例:
SELECT
money ,
CASE
WHEN money > 1000
THEN '工资尚可'
WHEN money > 500 AND money <= 1000
THEN '要加油了'
ELSE
 '工资不行'

DEFAULT
'我裂开了'
END
FROM account
;

-- 用法2 case 与 WHEN之间有条件
CASE [test]
WHEN[val1]
THEN [result]
...
ELSE [default]
END  如果test和valN相等，则返回resultN，否则返回default

例:
SELECT CASE 5 WHEN 3 THEN 'A' WHEN 2 THEN 'B' ELSE 'C' END;  -- 结果 C
```

IF函数

```
-- 如果test是真，返回t；否则返回f
IF(test,t,f)
例:
SELECT IF(3<0,1,2); -- 结果: 2
```

IFNULL函数

```
-- 如果arg1非null 则返回arg1 否则返回arg2
IFNULL(arg1,arg2)
例:
SELECT IFNULL(NULL,'aaa'); -- 结果: aaa
SELECT IFNULL('AAA','BBB'); -- 结果:  AAA
```

NULLIF函数

```
-- 如果arg1=arg2返回NULL；否则返回arg1
NULLIF(arg1,arg2)
例:
SELECT NULLIF('A','A'); -- 结果:  NULL
SELECT NULLIF('A','B'); -- 结果:  A
```

格式化函数

```
DATE_FORMAT(date,fmt)   -- 依照字符串fmt格式化日期date值

FORMAT(x,y)             -- 把x格式化为以逗号隔开的数字序列，y是结果的小数位数

INET_ATON(ip)           -- 返回IP地址的数字表示

INET_NTOA(num)          -- 返回数字所代表的IP地址

TIME_FORMAT(time,fmt)   -- 依照字符串fmt格式化时间time值

其中最简单的是FORMAT()函数，它可以把大的数值格式化为以逗号间隔的易读的序列。

示例:

SELECT FORMAT(34234.34323432,3);

SELECT DATE_FORMAT(NOW(),'%W,%D %M %Y %r');

SELECT DATE_FORMAT(NOW(),'%Y-%m-%d');

SELECT DATE_FORMAT(19990330,'%Y-%m-%d');

SELECT DATE_FORMAT(NOW(),'%h:%i %p');

SELECT INET_ATON('10.122.89.47');
```

```
26
27 SELECT INET_NTOA(175790383);
```

类型转换函数

```
1  为了进行数据类型转化，MySQL提供了CAST()函数，它可以把一个值转化为指定的数据类型。类型有：
   BINARY,CHAR,DATE,TIME,DATETIME,SIGNED,UNSIGNED
2
3  示例：
4
5  SELECT CAST(NOW() AS SIGNED INTEGER),CURDATE()+0;
6
7  SELECT 'f'=BINARY 'F','f'=CAST('F' AS BINARY);
```

系统信息函数

```
1  -- 返回当前数据库名
2  DATABASE()
3
4  -- 将表达式expr重复运行count次
5  BENCHMARK(count,expr)   `,,,,,,,,,,,,,,,,,
6
7  -- 返回当前客户的连接ID
8  CONNECTION_ID()
9
10 -- 返回最后一个SELECT查询进行检索的总行数
11 FOUND_ROWS()
12
13 -- 返回当前登陆用户名
14 USER()或SYSTEM_USER()
15
16 --  返回MySQL服务器的版本
17 VERSION()
18
```

事务安全 TCL

事务概述

```
1  一段多步骤的业务操作：
2  ** 要么全部成功 要么全部失败
```

事务操作

语法

```sql
-- 开启事务
BEGIN;

-- 提交事务
COMMIT;

-- 回滚事务
ROLLBACK;
```

示例(转账)

- **数据准备**

```sql
-- 创建数据表
CREATE TABLE account (  -- 账户表
    id INT PRIMARY KEY AUTO_INCREMENT,
    `name` VARCHAR(32),
    money DOUBLE
);

-- 添加数据
INSERT INTO account (`name`, money) VALUES ('许幻山', 1000), ('林有有', 1000);
```

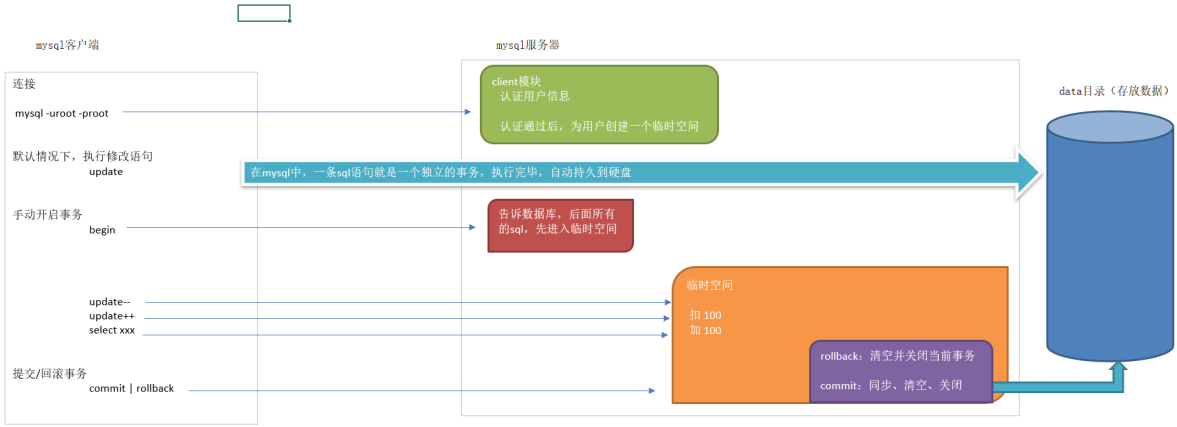
- **示例**

```sql
-- 开启事务
BEGIN;

-- 转账
UPDATE account SET money= money -200 WHERE `name`='许幻山';

-- 收款
UPDATE account SET money= money +200 WHERE `name`='林有有';

--(成功)提交  失败则 使用ROLLBACK回滚
COMMIT;

```

- **注意**

```sql
-- 默认情况下，mysql中一条sql语句，就是一个独立的事务

-- 查看mysql是否开启自动提交
SHOW VARIABLES LIKE 'autocommit';

-- 临时关闭自动提交 (关闭后 则需要进行手动提交)
SET autocommit = off;

-- 例
-- 许幻山 扣100
UPDATE account SET money = money - 100 WHERE `name` = '许幻山';
-- 手动提交/回滚
COMMIT| ROLLBACK;
```

事务的工作原理



回滚点

- 回滚点的作用: 当事务开启后，一部分sql执行成功，添加一个保存点，后续操作报错了，回滚到保存点，保证之前的操作可以成功提交

```sql
1. 设置保存点
SAVEPOINT 回滚点名称;

2.代码报错，回滚到保存点
ROLLBACK TO 保存点名称;
```

- 示例

```sql
-- 开启事务
BEGIN;
-- 转账
UPDATE account SET money= money -200 WHERE `name`='许幻山';
-- 收款
UPDATE account SET money= money +200 WHERE `name`='林有有';
-- 设置回滚点
SAVEPOINT back;
-- 机器故障
-- 回滚事务到回滚点
```

```
ROLLBACK TO back;
-- 提交事务
COMMIT;

```

事务特性(ACID)

```
** 原子性: A atomicity
        如果一个包含多个步骤的业务操作，被事务管理，那么这些操作要么同时成功，要么同时失败

** 一致性：C consistency
        事务在执行前后，数据完整性必须保持一致。

** 隔离性：I isolation【重点】
        多个事务，相互独立，互补干扰 即一个事务中的数据操作不能受其他事务的影响

** 持久性：D durability
        事务一旦提交，同步到磁盘，即使数据库出现了也不会产生影响
```

事务的隔离性

- 多个事务之间隔离的，相互独立的。但是如果多个事务操作同一批数据，则会引发一些问题(脏读 不可重复读 幻读)，可以通过设置数据库的隔离级别来解决这些问题
- 常见问题概念介绍

```
1. 脏读【实际环境中，绝对不允许发生...】
        事务A读取到事务B更新的数据，然后事务B回滚，事务A读取的这部分就是脏数据...
        事务A读取了事务B未提交的数据，就称为脏读

2. 不可重复读
        事务A中多次读取同一个数据，在此期间事务B对数据做了修改并提交，事务A多次读取同一个数据，出现了不一致
        事务A读取到事务B更新后并提交的数据，就称为不可重复读

3. 幻读
        事务A和事务B同时添加一条记录，发现id为10可以用(即id为10的数据不存在)，此时事务A插入id10并保存提交，事务B插入数据出现错误
```

MySQL的隔离级别

级别	名字	隔离级别	脏读	不可重复读	幻读	数据库默认隔离级别
1	读未提交	read uncommitted	是	是	是	
2	读已提交	read committed	否	是	是	Oracle和SQL Server
3	可重复读	repeatable read	否	否	是	MySQL
4	串行化	serializable	否	否	否	

- 安全: 4>3>2>1
- 性能: 1>2>3>4
- 通常使用: 2或3

MySQL高级

索引

索引概述

索引（index）是帮助MySQL高效获取数据的数据结构（有序）。在数据之外，数据库系统还维护者满足特定查找算法的数据结构这些数据结构以某种方式引用（指向）数据,这样就可以在这些数据结构上实现高级查找算法，这种数据结构就是索引。简而言之 索引本质上就是一种数据结构

索引的优缺点

- 优点 :
  - 提高数据检索的效率，降低数据库的IO成本
  - 通过索引列对数据进行排序，降低数据排序的成本，降低CPU的消耗
- 缺点:
  - 索引本身也是一张表，该表中保存了主键与索引字段，并指向实体类的记录，所以索引列本身也会有一定的空间占用
  - 降低了更新表的速度，如对表进行INSERT、UPDATE、DELETE。因为更新表时，MySQL 不仅要保存数据，还要保存一下索引文件每次更新添加了索引列的字段，都会对更新所带来的键值变化后的索引信息进行调整。

索引数据结构

- 索引是在MySQL的**存储引擎层**中实现的，而不是在服务器层实现的。所以每种存储引擎的索引都不一定完全相同，也不是所有的存储引擎都支持所有的索引类型的。MySQL目前提供了以下4种索引:

```
1. BTREE 索引 ： 最常见的索引类型，大部分索引都支持 B 树索引。

2. HASH 索引：只有Memory引擎支持 ， 使用场景简单 。

3. R-tree 索引（空间索引）：空间索引是MyISAM引擎的一个特殊索引类型，主要用于地理空间数据类型，通常使用较少

4. Full-text （全文索引） ：全文索引也是MyISAM的一个特殊索引类型，主要用于全文索引，InnoDB从Mysql5.6版本开始支持全文索引。
```

- **注意:**如果没有特别指明，都是指B+树（多路搜索树，并不一定是二叉的）结构组织的索引。其中聚集索引、复合索引、前缀索引、唯一索引默认都是使用 B+tree 索引，统称为 索引。

BTREE索引简介

BTree又叫多路平衡搜索树，一颗m叉的BTree特性如下：

- 树中每个节点最多包含m子节点。
- 除根节点与叶子节点外，每个节点至少有[ceil(m/2)]个子节点。
- 若根节点不是叶子节点，则至少有两个子节点。
- 所有的叶子节点都在同一层。
- 每个非叶子节点由n个key与n+1个指针组成，其中[ceil(m/2)-1] <= n <= m-1

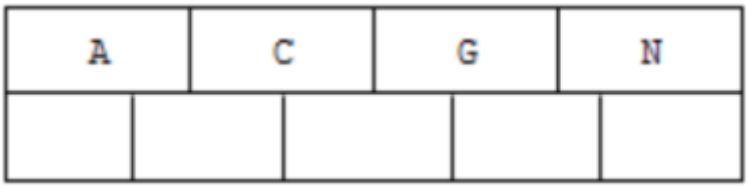
以5叉BTree为例，key的数量：公式推导[ceil(m/2)-1] <= n <= m-1。所以 2 <= n <=4 。当n>4时，中间节点向上分裂到父节点，中间节点两边的数据 以中间节点为界限 在两边进行分裂

- **以5叉查找树为例:**

  插入 C N G A H E K Q M F W L T Z D P R X Y S 这些数据 每个字符代表一条数据 从左到右依次插入

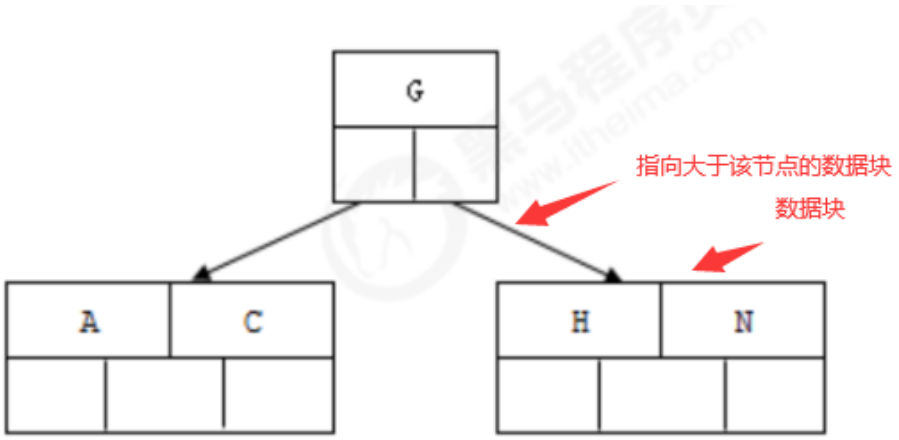
```
1. 计算非叶子节点存储的 key 范围 ： ceil(5/2) - 1 =< n <= 5-1 即: 2< n < 4
由此可知 非叶子节点最多保存 4个key 超过四个 中间节点就需要想上分裂成为父节点
```

- **演变过程:**
  - 1. 插入前四个字母 C N G A

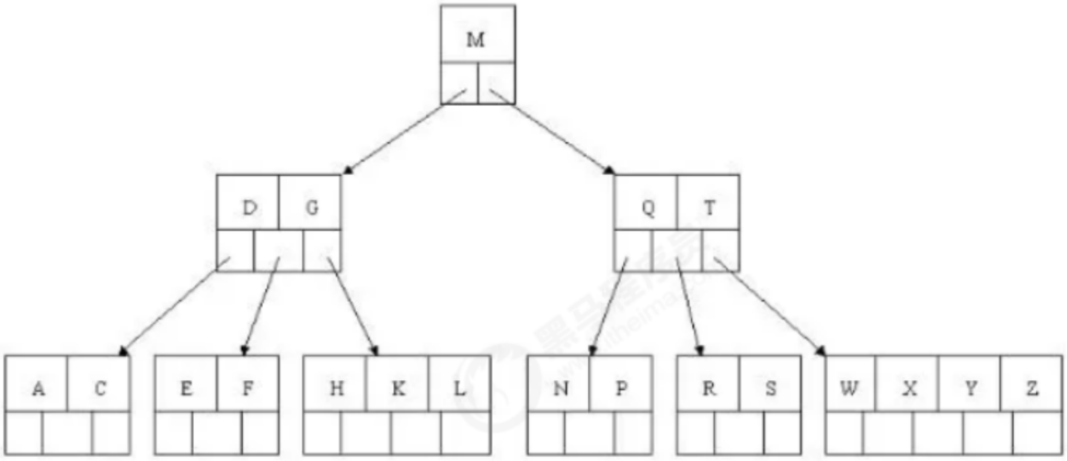



```
1. 首先插入C 此时C为根节点
2. 插入N时 N与C进行比较 比较后 插入在C的右侧(多叉查找树的每个节点可以保存多个key 5叉树非叶子节点的key范围为2-4 )
3. 以此类推 CNGA 几个key的插入如上图所示 每个key下方对应的为指针数 每个指针执行不同的数据块 以C为例 C的左侧指向比C小的数据块 右侧指向比C大的数据块
```

- 2. 插入H, 插入后 节点 key的数目大于4 中间元素就需要进行向上分裂成为父节点,如下所示:



- 3. 随着 元素的插入 循环上述过程 每次插入 先与顶层节点比较 比较的目的是找到对应的可插入数据块 若插入的数据块插入数据后 key 的数量大于该多路查找树的 key值数目的阈值 则该数据块对应的中间元素向上分裂 成为父节点(父节点是由节点分裂而来的 元素插入插入在子节点中 父节点起到的作用主要是 找到对应的子节点数据块) 最终结果如下:

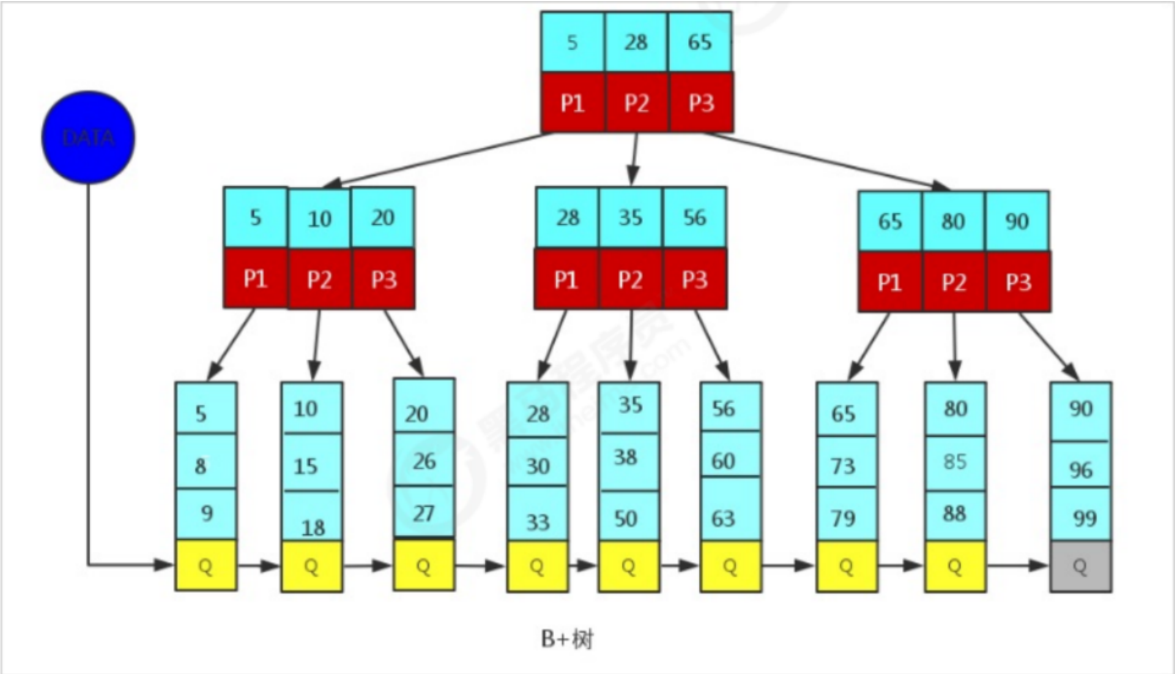



```
BTREE相对于二叉树  查询效率更高  降低了IO次数(树的高度降低了)
```

B+Tree简介

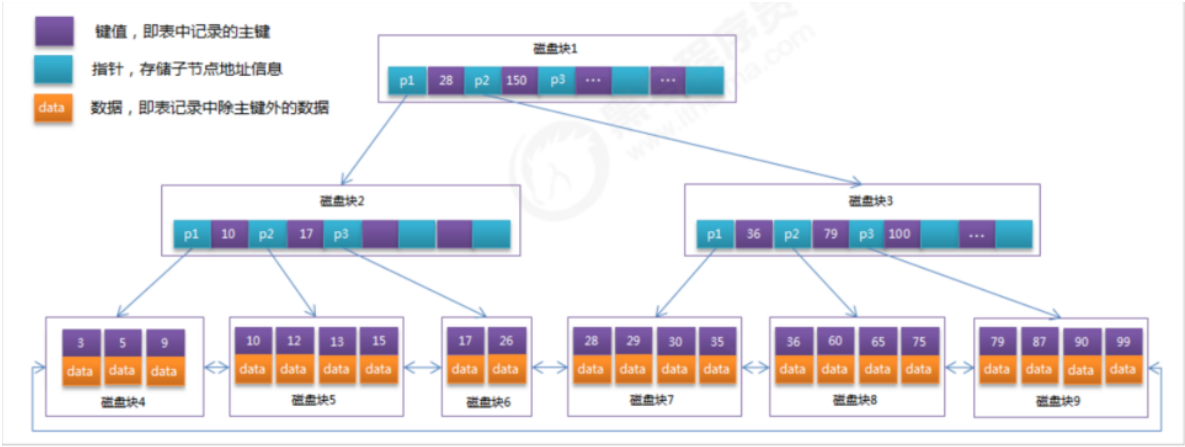
B+Tree为BTree的变种，B+Tree与BTree的区别为：

- n叉B+Tree最多含有n个key，而BTree最多含有n-1个key。
- B+Tree的叶子节点保存所有的key信息，依key大小顺序排列。
- 所有的非叶子节点都可以看作是key的索引部分。



B+树

- B+Tree只有叶子节点保存key信息，查询任何key都要从root走到叶子。所以B+Tree的查询效率更加稳定
- MySql索引数据结构对经典的B+Tree进行了优化。在原B+Tree的基础上，增加一个指向相邻叶子节点的链表指针，就形成了带有顺序指针的B+Tree，提高区间访问的性能。



索引分类

```
1） 单值索引 ：即一个索引只包含单个列，一个表可以有多个单列索引

2） 唯一索引 ：索引列的值必须唯一，但允许有空值

3） 复合索引 ：即一个索引包含多个列
```

索引的相关操作

- **创建索引**

```
CREATE [NIQUE | FULLTEXT | SPATIAL]  -- 索引类型可不指定
INDEX 索引名
ON 表名(需要创建索引的列,....); -- 可以选择创建多个作为复合索引
```

- **查看索引**

```
SHOW INDEX FROM 表名;
```

- **删除索引**

```
DROP INDEX 索引名 ON 表名;
```

- **ALTER命令操作索引**

```
ALTER TABLE 表名 ADD PRIMARY KEY(列名); -- 添加主键索引 (当表中某个列被指定为主键时主键索引也会同时创建)

ALTER TABLE 表名 ADD UNIQUE 索引名(列名); --创建唯一索引,创建后该字段对应的数据值必须是唯一的（NULL除外，NULL可能会出现多次）

ALTER TABLE 表名 ADD INDEX  索引名(列名); -- 添加普通索引

ALTER TABLE 表名 ADD FULLTEXT 索引名(列名); -- 创建全文索引

```

索引设计原则

```
1. 索引字段的选择，最佳候选列应当从where子句的条件中提取，如果where子句中的组合比较多，那么应当挑选最常用、过滤效果最好的列的组合。

2. 使用唯一索引，区分度越高，使用索引的效率越高。

3. 索引可以有效的提升查询数据的效率，但索引数量不是多多益善，索引越多，维护索引的代价越高数据修改的性能消耗也会更高



4. 使用短索引，索引创建之后也是使用硬盘来存储的，因此提升索引访问的I/O效率，也可以提升总体的访问效率。假如构成索引的字段总长度比较短，那么在给定大小的存储块内可以存储更多的索引值，相应的可以有效的提升MySQL访问索引的I/O效率。


5. 利用最左前缀，N个列组合而成的组合索引，那么相当于是创建了N个索引，如果查询时where子句中使用了组成该索引的前几个字段，那么这条查询SQL可以利用组合索引来提升查询效率。

例:
创建复合索引:
CREATE INDEX idx_name_email_status ON tb_seller(NAME,email,STATUS);
就相当于
对name 创建索引 ;
对name , email 创建了索引 ;
对name , email, status 创建了索引 ;
```

```
21
```

索引的使用

- 索引是数据库优化最常用也是最重要的手段之一, 通过索引通常可以解决大多数的MySQL的性能优化问题

环境准备

避免索引失效的方案

- **①全值匹配:** 对索引中所有列都指定具体值 该情况下索引生效 执行效率高

```
1.全值匹配 ，

```

- **②最左前缀法则:** 如果索引了多列，要遵守最左前缀法则。指的是查询从索引的最左前列开始，并且不跳过索引中的列

```

```

查看索引的使用情况

- 查询指令:

```
SHOW STATUS LIKE 'Handler_read%';
SHOW GLOBAL STATUS LIKE 'Handler_read%';
```

- 查询结果:
- aaa
- 结果分析

```
1.Handler_read_first：索引中第一条被读的次数。如果较高，表示服务器正执行大量全索引扫描（这个值越低  越好）

2.Handler_read_key：如果索引正在工作，这个值代表一个行被索引值读的次数，如果值越低，表示索引得到的  性能改善不高，因为索引不经常使用（这个值越高越好）

3.Handler_read_next ：按照键顺序读下一行的请求数。如果你用范围约束或如果执行索引扫描来查询索引列，  该值增加

4.Handler_read_prev：按照键顺序读前一行的请求数。该读方法主要用于优化ORDER BY ... DESC。

5.Handler_read_rnd ：根据固定位置读一行的请求数。如果你正执行大量查询并需要对结果进行排序该值较高。  你可能使用了大量需要MySQL扫描整个表的查询或你的连接没有正确使用键。这个值较高，意味着运行效率低，应  该建立索引来补救

6.Handler_read_rnd_next：在数据文件中读下一行的请求数。如果你正进行大量的表扫描，该值较高。通常说  明你的表索引不正确或写入的查询没有利用索引
```

存储过程

```
* 存储过程和函数是事先经过编译(创建时即确定功能)并存储在数据库中的一段 SQL 语句的集合，调用存储过程和函数可以简化应用开发人员的很多工作，减少数据在数据库和应用服务器之间的传输，可以提高数据处理的效率

* 存储过程和函数的区别在于 函数必须有返回值，而存储过程没有。

* 简而言之 存储过程和存储函数的区别点可以理解为 存储过程是一个没有返回值的存储函数
```

存储过程的CRD

创建存储过程

```
DELIMITER $    -- 声明分隔符为"$"(可任选) 默认情况下使用的是";" 分隔符的作用是告知MySQL sql语句书写完成

CREATE PROCEDURE 存储过程名()
BEGIN          -- 表示SQL语句起点
    SQL语句1; -- 这里的";" 主要用于分隔多条SQL
    SQL语句2;
    ...
END $         -- 表示SQL语句书写完成

DELIMITER $; -- 将分隔符 重新修改为";" (非必须 但是不修改的话 后续分隔符都只能使用 $)

-- 例:
DELIMITER $  -- 声明分隔符
-- 存储过程(下面的内容才是真正的存储过程 分隔符的声明只是为了解决 SQL分隔问题)
CREATE PROCEDURE test1()
BEGIN
SELECT '存储过程语句1';
SELECT '存储过程语句2';
END $

DELIMITER ;
```

- 存储过程创建完成之后会在 mysql库中的proc表中自动添加一条创建记录

调用存储过程

```
CALL 存储过程名();

例:
CALL test1();
```

查看存储过程

```
-- 查询某个数据库中创建的所有的存储过程名
SELECT `name` FROM mysql.proc WHERE db= '要查询的数据库名';
例:
SELECT `name` FROM mysql.`proc` WHERE db='web20_pro';

--查询所有存储过程状态信息
SHOW PROCEDURE STATUS;

-- 查询某个存储过程的定义
SHOW CREATE PROCEDURE 数据库名.存储过程名 ;
例:
SHOW CREATE PROCEDURE web20_pro.`test1`;
```

删除存储过程

```
DROP PROCEDURE [IF EXISTS] 存储过程名;
例:
DROP PROCEDURE IF EXISTS test1;
```

存储过程语法

- 存储过程是支持编程的，因此可以使用变量，表达式，控制结构， 来完成比较复杂的功能

变量声明与赋值

- DECLARE关键字可以声明变量 变量作用范围只能在该变量所属的 BEGIN...END区块内

1.定义变量同时赋默认值

```
DECLARE 变量名1,变量名2.... 变量类型 [DEFAULT 默认值]; -- 可以同时声明多个同类型变量并赋予相同的默认值
例:
DELIMITER $             -- 修改分隔符为 "$"
CREATE PROCEDURE var() -- 定义存储过程 var
BEGIN                   -- 存储过程内容起始边界声明
DECLARE var1,var2 INT DEFAULT 5; -- 声明变量 var1 并赋予默认值5
SELECT var1,var2;               -- 查看变量值
END $                   -- 存储过程内容结束边界声明

DELIMITER ;             -- 修改分隔符为 ";"

-- 调用存储过程:
CALL var();             -- 结果 5,5;
```

2.定义变量 然后通过 SET命令 赋值

```
SET 变量名 = 值,变量名=值.....;
例:
DELIMITER $                             -- 修改分隔符为 "$"
CREATE PROCEDURE var2()                 -- 定义存储过程 var2
BEGIN                                   -- 存储过程内容起始边界声明
DECLARE var3,var4 VARCHAR(30);          -- 声明变量 var3
SET var3='通过set指令给var3赋值',
    var4='通过set指令给var4赋值';      -- 对变量var3和var4 进行赋值
SELECT var3,var4;                       -- 查看变量内容
END $                                   -- 存储过程内容结束边界声明
DELIMITER ;                             -- 修改分隔符为 ";"
-- 调用存储过程
CALL var2();                -- 通过set指令给var3赋值 通过set指令给var4赋值
```

3.通过select ... into 方式对变量进行赋值操作

```
SELECT 字段1,字段2... INTO 变量1,变量2 FROM 表名 [where 条件];  --注意 查询到的记录只能有一条才能完成赋值 且数据类型需要对应 变量名不能与字段名一致(一致会赋值为null)

例:
DELIMITER $                     -- 修改分隔符为 "$"
CREATE PROCEDURE var3()         -- 定义存储过程 var3
BEGIN                           -- 存储过程内容起始边界声明
DECLARE `name1` VARCHAR(10); -- 声明字符类型变量name1
DECLARE money1 INT;             -- 声明整数类型变量money1
SELECT `name`,money INTO `name1`,money1 FROM account WHERE id=1; -- 给变量赋值
SELECT `name1`,money1;          -- 查看变量
END $                           -- 存储过程内容结束边界声明
DELIMITER ;                     -- 修改分隔符为 ";"

-- 调用存储过程
CALL var3();    -- 许幻山 1200
```

if条件判断

语法结构

```
IF 判断条件 THEN 条件成立执行语句
[ELSEIF 判断条件 THEN 条件成立执行语句]
[ELSE 以上条件都不满足的执行语句]
END IF;
```

示例

```
需求:
根据定义的身高变量，判定当前身高的所属的身材类型
180 及以上 ----------> 身材高挑
170 - 180 ---------> 标准身材
170 以下 ----------> 一般身材

```

```
存储过程:
DELIMITER $
CREATE PROCEDURE test()
BEGIN
DECLARE height INT DEFAULT 175;
DECLARE description VARCHAR(50);
IF HEIGHT >= 180              -- if语句开始声明
THEN
SET description = '身材高挑';
ELSEIF height >= 170 AND height < 180
THEN
SET description = '标准身材';
ELSE
SET description = '一般身材';
END IF;                        -- if语句结束声明
SELECT description;
END $
DELIMITER ;

-- 调用存储过程
CALL test();                  -- 标准身材
```

参数传递

语法结构

```
CREATE PROCEDURE 存储过程名([IN/OUT/INOUT] 参数名 参数类型)

IN:    -- 表示参数可以作为传入参数
OUT:   -- 表示该参数可以作为输出即作为返回值
INOUT: -- 表示既可以作为输入参数，也可以作为输出参数
```

示例:

```
需求: 根据定义的身高变量，判定当前身高的所属的身材类型

存储过程:

DELIMITER $
CREATE PROCEDURE parameter(IN height INT, OUT description VARCHAR(50))
BEGIN
IF height >= -- if语句开始声明 变量声明来自于参数不需要在重复声明(重复声明 可能导致传递的参数值被覆                    盖)
THEN
SET description = '身材高挑';
ELSEIF height >= 170 AND height < 180
THEN
SET description = '标准身材';
ELSE
SET description = '一般身材';
END IF;                        -- if语句结束声明
END $
DELIMITER ;

-- 调用存储过程
```

```
CALL parameter(181,@description);  -- @符号表示用一个会话变量来接收 不使用的话无法获取输出结                                                    果
SELECT @description1; -- 身材高挑
```

- **拓展:** @description 这种变量要在变量名称前面加上“@”符号，叫做用户会话变量，作用域为整个会话

  两个"@@" 修饰系统变量 适用于整个系统

case结构

case的两种使用方式

- **方式一**: CASE 后跟有VALUE

```
CASE case_value                  -- case_value为一个具体的值
WHEN value1 THEN 执行语句1;       -- 当case_value=value1时 执行语句1
[WHEN value1 THEN 执行语句2;]     -- 当case_value=value2时 执行语句2
...
[ELSE 执行语句;]                  -- 当case_value与以上value都不匹配时 执行对应语句
END CASE;                        -- case语句结束声明

```

- **方式二**: CASE后直接跟WHEN

```
CASE                             -- case语句起始声明
WHEN 条件1 THEN 执行语句1;        -- 当条件1为true时 执行语句1
[WHEN 条件2 THEN执行语句2;]       -- 当条件2为true时 执行语句2
...
[ELSE 执行语句;]                  -- 当以上条件都为false时 执行对应语句
END CASE;                        -- case语句结束声明
```

- **注意:** 不要忘记执行语句之后的";"

示例

```
-- 需求:给定一个月份, 然后计算出所在的季度

DELIMITER $
CREATE PROCEDURE quarter_calc(IN MONTH INT)
BEGIN
DECLARE result VARCHAR(10);
CASE
WHEN MONTH >= 1 AND MONTH <= 3 THEN
SET result = '第一季度';
WHEN MONTH >= 4 AND MONTH <= 6 THEN
SET result = '第二季度';
WHEN MONTH >= 7 AND MONTH <= 9 THEN
SET result = '第三季度';
ELSE
SET result = '第四季度';
END CASE;
SELECT CONCAT('您输入的月份为:',MONTH,' 该月份对应的的季度为: ',result) AS content;
```

```
END $
DELIMITER

-- 调用存储过程
CALL quarter_calc(11); -- 您输入的月份为:11 该月份对应的的季度为: 第四季度
```

while循环

语法结构

```
WHILE 条件 DO       -- 条件为true 执行下列执行语句
执行语句1;
执行语句2;
......
END WHILE;          -- while循环结束声明
```

示例

```
需求: 计算1加到n的值

DELIMITER $
CREATE PROCEDURE calc(IN n INT)
BEGIN
DECLARE num INT DEFAULT 1;
DECLARE result INT DEFAULT 0;
WHILE num <= n DO               -- 循环条件
SET result = result + num;      -- 累加赋值
SET num = num + 1;              -- 条件值+1
END WHILE;                      -- 结束循环
SELECT result;
END $
DELIMITER ;

-- 调用存储过程
CALL CALC(10); -- 55
```

repeat结构

- 有条件的循环控制语句, 当满足条件的时候退出循环

语法结构

```
REPEAT          -- REPEAT语句起始声明
 执行语句1;      -- 循环语句
 执行语句2;
UNTIL 条件      -- 循环终止条件 注意条件后面没有";"
END REPEAT;    -- REPEAT语句结束声明
```

示例

```
需求: 计算从1加到n的值

DELIMITER $
CREATE PROCEDURE repeat_calc(IN n INT)
BEGIN
DECLARE num INT DEFAULT 1;              -- 定义初始值循环条件
DECLARE result INT DEFAULT 0;           -- 定义变量接收累加结果
REPEAT                                  -- REAPEAT语句起始声明
SET result = result + num;
SET num = num + 1;
UNTIL NUM > n                           -- 当n>10 时停止循环
END REPEAT;
SELECT CONCAT('1到',n,'进行累加的结果为:',result) AS 结果;
END $
DELIMITER ;

-- 调用存储过程
CALL repeat_calc(10);  -- 1到10进行累加的结果为:55
```

loop语句

- LOOP 实现简单的循环，退出循环的条件需要使用其他的语句定义，通常可以使用 LEAVE 语句实现;单独使用LOOP的话 执行的循环语句不会退出

语法结构

```
[LOOP语句别名:] LOOP          -- LOOP语句起始声明
    执行语句1;
    执行语句2;
END LOOP [LOOP语句别名];

```

示例

```
需求 : 循环查询

DELIMITER $
CREATE PROCEDURE loop_cal(IN num INT)
BEGIN
ins: LOOP                   -- LOOP起始声明 别名定义为 ins
SELECT num;
END LOOP ins;               -- LOOP结束声明
END $
DELIMITER ;

-- 调用存储过程
CALL loop_cal(10);  -- mysql进入死循环 无法退出 无法进行其他操作 需要强制结束进程
```

leave语句

- 用来从标注的流程构造中退出，通常与LOOP循环一起使用,用于退出循环

示例

```
需求 : 求出 1-n的数累加结果

DELIMITER $
CREATE PROCEDURE loopWithLeave(IN n INT)
BEGIN
DECLARE result INT DEFAULT 0;    -- 声明变量接收累加结果
DECLARE num INT DEFAULT n;       -- 声明变量记录传入参数的初始值
ins: LOOP                        -- loop语句起始声明 别名定义为ins
SET result = result + n;
SET n = n - 1;
IF n <= 0 THEN                   -- if语句 当n<=0时执行对应语句体
LEAVE ins;                       -- 使用LEAVE 退出循环
END IF;                          -- if语句结束声明
END LOOP ins;                    -- loop语句结束声明
SELECT CONCAT('1到',num,'的累加结果为:',result) AS 结果;
END $
DELIMITER ;

-- 调用存储过程
CALL loopWithLeave(10); -- 1到10的累加结果为:55

```

游标

- 游标(也称为光标)是用来存储查询结果集的数据类型，在存储过程和函数中可以使用游标对结果集进行循环的处理

游标语法结构

```
-- 声明游标
DECLARE 游标名 CURSOR 查询语句;

-- OPEN游标(启用游标)
OPEN 游标名;

-- FETCH游标(通过FETCH指令可以获取由表中的单行数据 通常通过FETCH将获取到的数据存储到变量中 FECTH指令会从上往下依次回去数据,相当于指针 当获取到最后一行数据后 若继续向下获取 会出现错误[],通常可以使用LEAVE指令或句柄(MySql中的事件)来终止FETCH指令)
FETCH 游标名 INTO 变量1,变量2...;

-- CLOSE 游标
CLOSE 游标名;
```

示例

```
-- 方式1 通过句柄控制 FETCH指命令

DELIMITER $
CREATE PROCEDURE cursor_test1()
BEGIN
DECLARE id INT;              -- 声明变量 需要与查询结果中数据的类型对应
DECLARE `name` VARCHAR(20);
DECLARE money INT;
DECLARE hasData INT DEFAULT 1;    -- 声明变量 用于退出repeat循环
DECLARE test CURSOR  FOR SELECT * FROM account; -- 定义游标
DECLARE EXIT HANDLER FOR NOT FOUND SET hasData = 0; -- 声明句柄 需要写在游标下方
OPEN test;  -- 开启游标
REPEAT
FETCH test INTO id,`name`,money;
SELECT CONCAT('id为: ',id,' name为: ',`name`,' money为: ',money);
UNTIL hasData = 0  -- 当hasData=0时 退出循环
END REPEAT;
CLOSE test; -- 关闭游标
END $
DELIMITER ;

-- 调用存储过程
CALL cursor_test1(); -- 1 许幻山 1200
                     -- 2 林有有 800


-- 方式1 通过循环条件(变量)直接控制 FETCH指命令
DELIMITER $
CREATE PROCEDURE cursor_test2()
BEGIN
DECLARE id INT;
DECLARE `name` VARCHAR(20);
DECLARE money INT;
DECLARE hasData INT DEFAULT 1;
DECLARE nums INT;                                    -- 声明变量用于控制循环退出
DECLARE test CURSOR  FOR SELECT * FROM account;  -- 声明游标
OPEN test;                                           -- 开启游标
SELECT COUNT(*) c INTO nums FROM account;        -- 给条件变量赋值 需要写在游标声明下方 不然报错
REPEAT                                               -- REPEAT循环起始声明
FETCH test INTO id,`name`,money;
SELECT CONCAT('id为: ',id,' name为: ',`name`,' money为: ',money);
SET nums = nums - 1;
UNTIL nums = 0                                       -- 当num为0时 退出循环
END REPEAT;                                          -- REPEAT循环结束声明
CLOSE test;                                          -- 关闭游标
END $
DELIMITER ;

-- 调用存储过程
CALL cursor_test2(); -- 1 许幻山 1200
                     -- 2 林有有 800

```

- **句柄:** 当某些事件触发时执行的指令

存储函数

```
预先经过编译(创建时即确定功能)并存储在数据库中的一段 SQL 语句的集合,必须有返回值
```

语法结构

```
CREATE FUNCTION 函数名([参数 参数类型 ...])    -- 可传递多个参数
RETURNS 返回值类型                              -- 注意此处没有";"
BEGIN
    语句1;
    语句2;
    ...
    RETURN 返回值;                              -- 必须有返回值
END
```

示例

```
需求: 定义一个存储函数 获取满足条件的数据的总记录数

DELIMITER $
CREATE FUNCTION getCountByCondition(num INT)
RETURNS INT                 -- 指定函数返回值类型
BEGIN
DECLARE counts INT;         -- 声明变量 存储返回值
SELECT COUNT(*) INTO counts FROM account WHERE id = num; -- 给变量赋值
RETURN counts;              -- 返回数据
END $
DELIMITER ;

-- 调用存储函数
SELECT getCountByCondition(3); -- 0
SELECT getCountByCondition(1); -- 1
SELECT getCountByCondition(2); -- 1
```

触发器

触发器简介

```
* 触发器概念:
触发器是与表有关的数据库对象，可以在进行insert/update/delete等操作之前或之后，触发并执行触发器中定义的SQL语句集合。

* 触发器的作用:
(1)可以协助应用在数据库端确保数据的完整性 , 日志记录 , 数据校验等操作 。

(2)可以使用别名 OLD 和 NEW 来引用触发器中发生变化的记录内容。目前触发器只支持行级触发。
```

触发器类型

触发器类型	NEWH和OLD的使用
INSERT型(插入)触发器	NEW表示将要或者已经新增的数据
UPDATE型(更新)触发器	OLD 表示>修改之前修改之前的数据 , NEW 表示将要或已经修改后的数据
DELETE(删除)型触发器	OLD 表示将要或者已经删除的数据

- **注意:**
  - INSERT通常用来获取新增后的数据 例如id自增长时 获取新增数据的id ,因此只有NEW引用
  - UPDATE通常需要对修改前后的数据进行对比 需要获取修改之前和修改之后的数据 ,因此有OLD和NEW两种引用
  - DELETE可以获取的是将要或已经删除的数据 因此只有OLD引用

触发器操作语法

```
CREATE TIRIGGER 触发器名
BEFORE/AFTER INSERT/UPDATE/DELETE          -- BEFORE/AFTER 表示触发时机 例如:AFTER INSERT 表示新增数据之后触发触发器
ON 操作表表名                               -- 表示对那张表进行操作后会触发触发器
FOR EACH ROW                               -- 表示行级触发器 对每行对应操作都生效
BEGIN                                      -- 触发器执行语句起始声明
    语句体;                                -- 触发器触发后对应的执行语句
END;                                       -- 触发器执行语句结束声明
```

- **数据准备**

```
-- 创建account操作日志表
CREATE TABLE account_log(
id INT(11) PRIMARY KEY AUTO_INCREMENT COMMENT '日志id',
operation VARCHAR(20) NOT NULL COMMENT '操作类型:insert/update/delete',
operate_time DATETIME NOT NULL COMMENT '操作时间',
operate_id  INT(11) NOT NULL COMMENT '操作表的ID',
operate_params VARCHAR(500) COMMENT '操作参数'
)
```

创建触发器

1.创建INSERT触发器

- **创建触发器**:完成插入数据时的日志记录

```
DELIMITER $
CREATE TRIGGER account_logs_insert_trigger  -- 创建触发器
AFTER INSERT                                -- 设定执行时间为 插入之后执行
ON account                                  -- 指定触发器对应的数据表
FOR EACH ROW                                -- 行级触发器
BEGIN                                       -- 触发器执行语句起始声明
INSERT INTO account_log
(id,operation,operate_time,operate_id,operate_params) VALUES
(NULL,'INSERT',NOW(),NEW.id,CONCAT('插入后id:',NEW.id,' name: ',NEW.name,'
money: ',NEW.money));                              -- account表中数据插入后的执
行语句
END $
DELIMITER ;
```

- 插入数据

```
-- 插入数据
INSERT INTO account VALUES(6,'陈平安',2000) ;

-- 查看日志表
SELECT * FROM account_log; -- 1 insert 2020-12-27 14:09:29 6 插入后id:6 name:
陈平安 money: 2000

```

2.创建UPDATE触发器

- **创建触发器:**完成更新数据时的日志记录

```
DELIMITER $
CREATE TRIGGER account_log_update_trigger   -- 创建触发器
AFTER UPDATE                                -- 指定触发时机为数据修改后触发
ON account                                  -- 指定触发器对应的数据表
FOR EACH ROW                                -- 行级触发器
BEGIN                                       -- 触发器执行语句起始声明
INSERT INTO account_log(id,operation,operate_time,operate_id,operate_params)
VALUES(
NULL,'UPDATE',NOW(),NEW.id,CONCAT('数据修改前: id为: ',OLD.id,' name为:
',OLD.name,' money为: ',OLD.money,'数据修改后 id为: ',NEW.id,' name为:
',NEW.name,' money为: ',NEW.money));                              -- account
表中数据修改后account表后的执行语句
END $
DELIMITER ;
```

- 修改数据

```
-- 修改数据
UPDATE account SET money = 2999 WHERE NAME = '宁姚';

-- 查询日志
SELECT * FROM account_log; -- 3 UPDATE  2020-12-27 14:26:18 7   数据修改前: id
为: 7 name为: 宁姚 money为: 3000数据修改后 id为: 7 name为: 宁姚 money为: 2999
```

3.创建DELETE触发器

- **创建触发器:**完成数据删除时的日志记录

```
DELIMITER $
CREATE TRIGGER account_log_delete_trigger
AFTER DELETE
ON account
FOR EACH ROW
BEGIN
INSERT INTO account_log
(id,operation,operate_time,operate_id,operate_params) VALUES
(NULL,'DELETE',NOW(),OLD.id,CONCAT('删除的数据为 :',OLD.id,' name:
',OLD.name,' money: ',OLD.money));
END $
DELIMITER ;
```

- 删除数据

```
-- 删除数据
DELETE FROM account WHERE id = 1;

-- 查询日志
SELECT * FROM account_log; -- 4 DELETE  2020-12-27 14:52:30 1   删除的数据为 :1
name: 许幻山 money: 1200
```

删除触发器

```
DROP TRIGGER [数据库名.]触发器名    -- 不写数据库名. 默认当前数据库

-- 示例
DROP TRIGGER web20_pro.account_log_delete_trigger;
```

查看触发器

```
SHOW TRIGGERS ;  --查看触发器的状态、语法等信息。
```

存储引擎

存储引擎概述

- 存储引擎是MySQL中的一个概念,是存储数据，建立索引，更新查询数据等等技术的实现方式.存储引擎是基于表而不是基于库的,所以存储引擎也可以被称为表类型
- 可以通过SHOW ENGINES ;命令查询当前数据库支持的存储引擎类型 ;创建表时,若不指定存储引擎,那么就会采用系统默认的存储引擎(5.5版本之前:MyISAM 5.5版本之后:InnoDB)
- 查看默认引擎指令: SHOW VARIABLES LIKE '%storage_engine%' ;

常用引擎及其特性简介

InnoDB:

- InnoDB存储引擎提供了具有提交、回滚、崩溃恢复能力的事务安全。但是对比MyISAM的存储引擎，InnoDB写的处理效率差一些，并且会占用更多的磁盘空间以保留数据和索引。

InnoDB特性

```
InnoDB存储引擎是Mysql5.5版本之后的默认引擎的默认存储引擎,特点如下:

* 存储限制为 : 64TB
* 支持事务
* 支持表锁(适用于高并发场景)
* 支持Btree索引 全文索引(5.6版本之后支持) 集群索引 数据索引 索引缓存
* 支持外键( 在创建外键的时候， 要求父表必须有对应的索引 ， 子表在创建外键的时候， 也会自动的创建对应的索引)

```

InnoDB存储表和索引的两种方式

- ①使用共享表空间存储， 这种方式创建的表的表结构保存在.frm文件中， 数据和索引保存在 innodb_data_home_dir 和 innodb_data_fifile_path定义的表空间中，可以是多个文件。
- ②使用多表空间存储， 这种方式创建的表的表结构仍然存在 .frm 文件中，但是每个表的数据和索引单独保存在.ibd 中。

MyISAM:

- MyISAM 不支持事务、也不支持外键，其优势是访问的速度快，对事务的完整性没有要求或者以SELECT、INSERT为主的应用基本上都可以使用这个引擎来创建表 。

MyISAM特性

```
MyISAM存储引擎是Mysql5.5版本之前的默认引擎的默认存储引擎,特点如下:

* 存储限制为 :
* 不支持事务
* 不支持行锁
* 支持Btree索引  索引缓存 数据可压缩

```

MyISAM文件存储方式

```
每个MyISAM在磁盘上存储成3个文件，其文件名都和表名相同.拓展名分别是 :
.frm (存储表定义);
.MYD(MYData , 存储数据);
.MYI(MYIndex , 存储索引);
```

存储引擎的选择

InnoDB :

```
如果应用对事务的完整性有比较高的要求，在并发条件下要求数据的一致性，数据操作除了插入和查询意外，还包含很多的更新、删除操作，那么InnoDB存储引擎是比较合适的选择。InnoDB存储引擎除了有效的降低由于删除和更新导致的锁定， 还可以确保事务的完整提交和回滚，对于类似于计费系统或者财务系统等对数据准确性要求比较高的系统，InnoDB是最合适的选择。
```

MyISAM:

```
MyISAM ： 如果应用是以读操作和插入操作为主，只有很少的更新和删除操作，并且对事务的完整性、并发性要求不是很高，那么选择这个存储引擎是非常合适的
```

数据库锁

数据库锁概述

```
锁是计算机协调多个进程或线程并发访问某一资源的机制（避免争抢）。
在数据库中，除传统的计算资源（如 CPU、RAM、I/O 等）的争用以外，数据也是一种供许多用户共享的资源。如
何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的
一个重要因素。从这个角度来说，锁对数据库而言显得尤其重要，也更加复杂。
```

锁分类

从对数据操作的粒度分：

1） 表锁：操作时，会锁定整个表。

2） 行锁：操作时，会锁定当前操作行。

从对数据操作的类型分:

1） 读锁（共享锁）：针对同一份数据，多个读操作可以同时进行而不会互相影响。

2） 写锁（排它锁）：当前操作没有完成之前，它会阻断其他写锁和读锁。

SQL优化

SQL优化步骤

1. 查看SQL执行频率

- MySQL 客户端连接成功后:可以通过 SHOW [session|global] STATUS 命令可以提供服务器状态信息。
- SHOW [session|global] STATUS 可以根据需要加上参数session或者global来显示不同级别的统计结果

session 级仅统计当前会话 global 级自数据库上次启动至今的统计结果 如果不写，默认使用参数是session。

查看当前session的统计信息

```
-- 查询所有引擎的操作统计
SHOW STATUS LIKE 'Com________'     -- 模糊查询(后续有七个'_');

-- 结果:
| Variable_name | Value |
+---------------+-------+
| Com_binlog    | 0     |
| Com_commit    | 0     |
| Com_delete    | 0     |
| Com_insert    | 0     |
| Com_repair    | 0     |
| Com_revoke    | 0     |
| Com_select    | 1     |
| Com_signal    | 0     |
| Com_update    | 0     |
| Com_xa_end    | 0     |
+---------------+-------+

-- 查询InnoDB引擎的行数操作统计
SHOW STATUS LIKE 'Innodb_rows_%';
-- 结果:
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| Innodb_rows_deleted  | 0     |
| Innodb_rows_inserted | 0     |
| Innodb_rows_read     | 0     |
| Innodb_rows_updated  | 0     |
+----------------------+-------+
```

常用参数及含义

参数	含义
Com_select	执行 SELECT操作的次数，一次查询计数累加 1
Com_insert	执行 INSERT 操作的次数，对于批量插入的 INSERT 操作，只累加一次。
Com_update	执行 UPDATE 操作的次数,每次操作,计数累加1
Com_delete	执行 DELETE操作的次数,每次操作,计数累加1
Innodb_rows_read	SELECT查询返回的行数
Innodb_rows_inserted	执行 INSERT 操作插入的行数
Innodb_rows_updated	执行 UPDATE 操作删除的行数
Innodb_rows_deleted	执行 DELETE 操作删除的行数
Connections	试图连接 MySQL 服务器的次数
Uptime	服务器工作时间
Slow_queries	慢查询的次数

2.定位低效执行SQL

方式一(慢查询日志):

- 通过慢查询日志定位执行效率较低的 SQL 语句

方式二(SHOW PROCESSLIST):

- 慢查询日志在查询结束以后才纪录，所以在应用反映执行效率出现问题的时候查询慢查询日志并不能定位问题，可以使用SHOW PROCESSLIST命令查看当前MySQL在进行的线程，包括线程的状态、是否锁表等，可以实时地查看 SQL 的执行情况，同时对一些锁表操作进行优化。

```
+----+------+-----------------+-----------+---------+------+-------+-----------------+
| Id | User | Host            | db        | Command | Time | State | Info            |
+----+------+-----------------+-----------+---------+------+-------+-----------------+
|  1 | root | localhost:53014 | NULL      | Query   |    0 | NULL  | show processlist |
|  2 | root | localhost:53033 | web20_pro | Sleep   |   17 |       | NULL            |
|  3 | root | localhost:53036 | NULL      | Sleep   |   20 |       | NULL            |
|  4 | root | localhost:53044 | web20_pro | Sleep   |   12 |       | NULL            |
|  5 | root | localhost:53052 | NULL      | Sleep   |   16 |       | NULL            |
+----+------+-----------------+-----------+---------+------+-------+-----------------+

```

- 查询解析

```
1. id列，用户登录mysql时，系统分配的"connection_id"，可以使用函数connection_id()查看

2. user列，显示当前用户。如果不是root，这个命令就只显示用户权限范围的sql语句

3. host列，显示这个语句是从哪个ip的哪个端口上发的，可以用来跟踪出现问题语句的用户

4. db列，显示这个进程目前连接的是哪个数据库

5. command列，显示当前连接的执行的命令，一般取值为休眠（sleep），查询（query），连接（connect等

6. time列，显示这个状态持续的时间，单位是秒

7. state列，显示使用当前连接的sql语句的状态 state描述的是语句执行中的某一个状态.一 个sql语句，以查询为例:可能需要经过copying to tmp table、sorting result、sending data等状态 才可以完成

8. info列，显示这个sql语句，是判断问题语句的一个重要依据

```

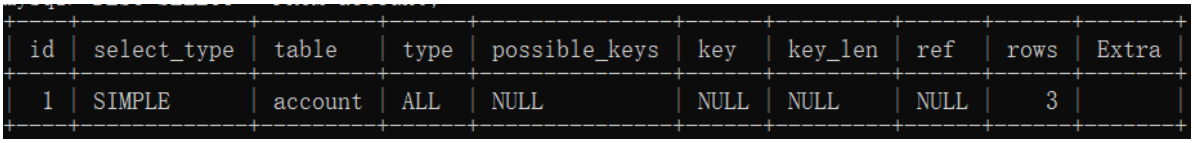
3.使用 explain指令分析执行计划

- 通过以上步骤定位到执行效率低下的SQL语句之后,可以使用EXPLAIN或者DESC命令来获取MySQL中SELECT语句的执行信息,包括执行过程中表连接和连接顺序

示例

```
EXPLAIN SELECT * FROM account;
或
DESC SELECT * FROM account;

```

结果



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	account	ALL	NULL	NULL	NULL	NULL	3	

字段分析

- **id**

```
id 字段值是 SELECT查询的序列号，是一组数字，表示的是查询中执行 SELECT 子句(子查询) 或者是操作表的顺序
常见规则如下:

1. id 相同则表的加载顺序由上至下进行加载

2. id 不同时 id值越大 加载优先级越高 越先被执行 -- 例如子查询中 字句的优先级会高于外层查询

3. 当一组查询中 既有id相同的又有不同的 ,id相同的可以认为是一组 从上往下执行 ,不同组中 id值越大的组 越先执行
```

- **select_type**

```
select_type表示SELECT的类型,常见取值如下:

1.SIMPLE: 表示简单的查询 查询中不包含子查询或者UNION

2.PRIMARY: 查询中若包含任何复杂的子查询,最外层的查询标记为该标识

3.SUBQUERY: 在SELECT或者WHERE列表中包含了子查询

4.DERIVED: 在FROM列表中包含的子查询,被标记为DERIVED(衍生)MYSQL会递归执行这些子查询,将结果放到临时表中

5.UNION: 若第二个SELECT出现在UNION之后，则标记为UNION ； 若UNION包含在FROM子句的子查询中，外层SELECT将被标记为 ： DERIVED

6.UNIONRESULT: 从UNION表获取结果的SELECT
```

- **table**

```
标识展示的这一行数据是关于哪一张表的
```

- **type**

```
type 显示的是访问类型，是较为重要的一个指标,常见取值如下:

1.NULL: MySQL 不访问任何表、索引直接返回结果.    -- 例如: EXPLAIN SELECT DATABASE();

2.system: 表中总计只有一行记录(等于系统表),属于const类型的特例， 一般情况下不会出现

3.const: 表示通过索引一次就找到了,const用于比较primary key 或者 unique 索引,因为只匹配一行数据 所以查询很快. 如果将主键置于where列表中,MySQL就能把该查询转换成为一个常量,const将与"主键"或"唯一"索引的所有部分与常量值进行比较

4.eq_ref: 类似ref，区别在于使用的是唯一索引，使用主键的关联查询，关联查询出的记录只有一条。常见于主键或唯一索引扫描
```

```
5.ref: 非唯一性索引扫描，返回匹配某个单独值的所有行。本质上也是一种索引访问，返回所有匹配某个单独值的所有行（匹配数据有多个）

6.range: 只检索给定返回的行，使用一个索引来选择行。 where 之后出现 between ， < , > , in 等操作

7.index: index 与 ALL的区别为 index 类型只是遍历了索引树， 通常比ALL 快， ALL 是遍历数据文件

8.all: 将遍历全表以找到匹配的行

-- 通常查询需要保证至少达到 range级别,最好为ref
```

- **possible_keys**

```
显示可能应用在这张表的索引， 一个或多个
```

- **key**

```
实际使用的索引， 如果为NULL， 则没有使用索引
```

- **key_len**

```
表示索引中使用的字节数， 该值为索引字段最大可能长度，并非实际使用长度，在不损失精确性的前提下， 长度越短越好
```

- **rows**

```
表示扫描数据的行数
```

- **extra**

```
展示其他的额外的执行计划信息,例如:

1.using filesort: 说明mysql会对数据使用一个外部的索引排序，而不是按照表内的索引顺序进行读取，称为“文件排序”， 效率低

2.using temporary: 使用了临时表保存中间结果，MySQL在对查询结果排序时使用临时表;常见于order by 和 group by； 效率低

3.using index: 表示相应的select操作使用了覆盖索引， 避免访问表的数据行，效率相对可观

```

SQL优化实例

数据导入优化

环境准备

- 创建数据表

```
CREATE TABLE `tb_user` (
`id` int(11) NOT NULL AUTO_INCREMENT,
    `username` varchar(45) NOT NULL,
    `password` varchar(96) NOT NULL,
    `name` varchar(45) NOT NULL,
    `birthday` datetime DEFAULT NULL,
    `sex` char(1) DEFAULT NULL,
    `email` varchar(45) DEFAULT NULL,
    `phone` varchar(45) DEFAULT NULL,
    `qq` varchar(32) DEFAULT NULL,
    `status` varchar(32) NOT NULL COMMENT '用户状态',
    `create_time` datetime NOT NULL,
    `update_time` datetime DEFAULT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `unique_user_username` (`username`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 ;
```

批量数据插入的优化方式

```
** 1.主键顺序插入
    InnoDB类型的表是按照主键的顺序保存的，所以将导入的数据按照主键的顺序排列，可以有效的提高导入数据的效率

** 2.关闭唯一性校验
    在导入数据前执行 SET UNIQUE_CHECKS=0，关闭唯一性校验，一般来说导入的数据都是合法数据,关闭唯一性校验之后可以降低导入性能消耗 从而提高导入的效率 在导入结束后执行SET UNIQUE_CHECKS=1，恢复唯一性校验即可。

** 3.手动提交事务
    在导入前执行 SET AUTOCOMMIT=0，关闭自动提交，导入结束后再执行 SET AUTOCOMMIT=1，打开自动提交，提高导入的效率

```

实例

- **数据导入指令简介**

```
-- 数据导入指令
LOAD DATA LOCAL INFILE '文件绝对路径' INTO TABLE 表名 FIELDS TERMINATED BY ','
LINES   TERMINATED BY '\n';

-- 解析: 从本地导入数据到某张表 字段之间以","分割 每条SQL 语句以 "\n"分割.

-- 导入的数据文本格式示例
1,"username1","ZPAIFXVC","name1","1957-01-23
20:22:02","1","0sq1rgktly@0355.net" ,"13301787682","SJVGNDTDWE","0","2019-04-
20 22:37:15","2019-04-20 22:37:15"
```

- **数据导入**

```
-- 1.主键顺序插入
LOAD DATA LOCAL INFILE 'C:\\Users\\60558\\Desktop\\sql\\sql1.log' INTO TABLE
tb_user FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n';
耗时: 15.041 sec

-- 清空数据表
TRUNCATE TABLE tb_user;

-- 2.主键无序插入
LOAD DATA LOCAL INFILE 'C:\\Users\\60558\\Desktop\\sql\\sql2.log' INTO TABLE
tb_user FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n';
耗时:  4 min 13 sec

-- 其余两种测试: 略
```

INSERT语句优化

```
-- 1.当一次插入的数据为多行时 尽量使用多值插入的insert语句,缩减客户端与数据库之间的连接、
关闭等消耗,使得效率比分开执行的单个insert语句快
INSERT INTO 表名 VALUES(value1,value2,...),(value1,value2,...),
(value1,value2,...)...;

-- 2.在事务中 尽量保证数据根据主键有序插入
例如:
INSERT INTO tb_test VALUES(1,'Tom');
INSERT INTO tb_test VALUES(2,'Cat');
INSERT INTO tb_test VALUES(3,'Jerry');
INSERT INTO tb_test VALUES(4,'Tim');
INSERT INTO tb_test VALUES(5,'Rose');
```

ORDER BY语句优化

环境准备

```
CREATE TABLE `emp` (
 `id` INT(11) NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(100) NOT NULL,
  `age` INT(3) NOT NULL,
   `salary` INT(11) DEFAULT NULL,
```

```
    PRIMARY KEY (`id`)
) ENGINE=INNODB DEFAULT CHARSET=utf8mb4;
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('1','Tom','25','2300');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('2','Jerry','30','3500');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('3','Luci','25','2800');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('4','Jay','36','3500');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('5','Tom2','21','2200');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('6','Jerry2','31','3300');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('7','Luci2','26','2700');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('8','Jay2','33','3500');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('9','Tom3','23','2400');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('10','Jerry3','32','3100');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('11','Luci3','26','2900');
INSERT INTO `emp` (`id`, `name`, `age`, `salary`)
VALUES('12','Jay3','37','4500');
CREATE INDEX idx_emp_age_salary ON emp(age,salary);

```

排序查询

- **单字段排序**

```
EXPLAIN SELECT * FROM emp ORDER BY age ASC;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	emp	ALL	NULL	NULL	NULL	NULL	12	Using filesort

```
EXPLAIN SELECT * FROM emp ORDER BY age DESC;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	emp	ALL	NULL	NULL	NULL	NULL	12	Using filesort

```
EXPLAIN SELECT id FROM emp ORDER BY age ASC;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	emp	index	NULL	idx_emp_age_salary	9	NULL	12	Using index

- **多字段排序**

```
EXPLAIN SELECT id FROM emp ORDER BY age,salary ASC;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	emp	index	NULL	idx_emp_age_salary	9	NULL	12	Using index

```
EXPLAIN SELECT id FROM emp ORDER BY salary , age ASC;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	emp	index	NULL	idx_emp_age_salary	9	NULL	12	Using index; Using filesort

排序的两种方式

- **FileSort排序**: 对返回的数据进行排序(先返回数据 然后在再对数据进行排序),所有不是通过索引直接返回排序结果的排序都叫 FileSort 排序
- **Using Index排序**:通过有序索引 顺序扫描直接返回有序数据,不需要额外排序 操作效率较高

优化方式

```
** 尽量减少额外的排序，通过索引直接返回有序数据

途径: where 条件和Order by 使用相同的索引，并且Order By 的顺序和索引顺序相同;当涉及到多条件排序时尽量保证排序条件一致,都为升序或降序
```

- **MySQL排序算法简介**

```
**双路排序(两次扫描算法):
    首先根据条件取出排序字段和行指针信息，然后在排序区sort buffer 中排序，如果sort buffer不够，则在临时表 temporary table 中存储排序结果。完成排序之后，再根据行指针回表读取记录，该操作可能会导致大量随机I/O操作。

**单路排序(一次扫描算法):
    一次性取出满足条件的所有字段，然后在排序区 sort buffer 中排序后直接输出结果集。排序时内存开销较大，但是排序效率比两次扫描算法要高。

* MySQL 通过比较系统变量 max_length_for_sort_data 的大小和Query语句取出的字段(列数)总大小， 来判定是否那种排序算法，如果max_length_for_sort_data 更大，那么使用第二种优化之后的算法；否则使用第一种。
可以适当提高 sort_buffer_size 和 max_length_for_sort_data 系统变量，来增大排序区的大小，提高排序的效率。
```

- **调优方法**

```
-- 查看当前 max_length_for_sort_data的大小
SHOW VARIABLES LIKE 'max_length_for_sort_data'; -- 默认为 1024 字节

-- 设置 max_length_for_sort_data的大小
SET [ GLOBAL | SESSION] max_length_for_sort_data = 整数  -- 值的设置范围 4~8388608
```

GROUP BY 语句优化

```
GROUP BY 实际上也同样会进行排序操作，而且与 ORDER BY 相比， GROUP BY 主要只是多了排序之后的分
组操作。当然，如果在分组的时候还使用了其他的一些聚合函数，那么还需要一些聚合函数的计算。所以，在
GROUP BY 的实现过程中，与 ORDER BY 一样也可以利用到索引。
如果查询包含 GROUP BY 但是用户想要避免排序结果的消耗， 则可以执行ORDER BY NULL 禁止排序。
```

示例

- **未禁止排序**

```
-- 删除索引
DROP INDEX idx_emp_age_salary ON emp;
-- 查询
EXPLAIN SELECT age,COUNT(*) FROM emp GROUP BY age;
```

```
+----+-------------+-------+------+---------------+------+---------+------+------+----------------------------------------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra                                        |
+----+-------------+-------+------+---------------+------+---------+------+------+----------------------------------------------+
|  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL |   12 | Using temporary; Using filesort              |
+----+-------------+-------+------+---------------+------+---------+------+------+----------------------------------------------+
```

- **禁止排序**

```
EXPLAIN SELECT age,COUNT(*) FROM emp GROUP BY age ORDER BY NULL;
```

```
+----+-------------+-------+------+---------------+------+---------+------+------+-----------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra           |
+----+-------------+-------+------+---------------+------+---------+------+------+-----------------+
|  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL |   12 | Using temporary |
+----+-------------+-------+------+---------------+------+---------+------+------+-----------------+
```

- **创建索引**

```
-- 建立索引
CREATE INDEX idx_emp_age_salary ON emp(age,salary);
-- 查询
EXPLAIN SELECT age,COUNT(*) FROM emp GROUP BY age ORDER BY NULL;
```

```
+----+-------------+-------+-------+---------------+-------------------+---------+------+------+-------------+
| id | select_type | table | type  | possible_keys | key               | key_len | ref  | rows | Extra       |
+----+-------------+-------+-------+---------------+-------------------+---------+------+------+-------------+
|  1 | SIMPLE      | emp   | index | NULL          | idx_emp_age_salary | 9       | NULL |   12 | Using index |
+----+-------------+-------+-------+---------------+-------------------+---------+------+------+-------------+
```

嵌套查询优化

```
通常选择的方式是将一些可以转换为连接查询的子查询 转换连接查询,这样MySQL就不需要在内存中创建临时表来完成这个逻辑上需要两个步骤的查询工作
```

示例

```
EXPLAIN SELECT * FROM t_user WHERE id IN (SELECT user_id FROM user_role );
```

```
+----+--------------------+-----------+-----------------+---------------+---------------+---------+------+------+-------------+
| id | select_type        | table     | type            | possible_keys | key           | key_len | ref  | rows | Extra       |
+----+--------------------+-----------+-----------------+---------------+---------------+---------+------+------+-------------+
|  1 | PRIMARY            | t_user    | ALL             | NULL          | NULL          | NULL    | NULL |    6 | Using where |
|  2 | DEPENDENT SUBQUERY | user_role | index_subquery | fk_ur_user_id | fk_ur_user_id | 99      | func |    1 | Using index; Using where |
+----+--------------------+-----------+-----------------+---------------+---------------+---------+------+------+-------------+
```

```
EXPLAIN SELECT * FROM t_user u , user_role ur WHERE u.id = ur.user_id;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	u	ALL	PRIMARY	NULL	NULL	NULL	6	
1	SIMPLE	ur	ref	fk_ur_user_id	fk_ur_user_id	99	web20_pro.u.id	1	Using where

OR条件优化

```
对于包含OR的查询子句，如果要利用索引，则OR之间的每个条件列都必须用到索引 即条件列都必须有索引，而且不能使用到复合索引(复合索引 or 拆分后 会失效)； 如果没有索引，则应该考虑增加索引。
```

示例

- **查看表emp中的索引**

```
SHOW INDEX FROM emp;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
emp	0	PRIMARY	1	id	A	12	NULL	NULL		BTREE		
emp	1	idx_emp_age_salary	1	age	A	12	NULL	NULL		BTREE		
emp	1	idx_emp_age_salary	2	salary	A	12	NULL	NULL	YES	BTREE		

- **or查询**

```
EXPLAIN SELECT * FROM emp WHERE id = 1 OR age = 30;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	emp	index_merge	PRIMARY,idx_emp_age_salary	idx_emp_age_salary,PRIMARY	4,4	NULL	2	Using sort_union(idx_emp_age_salary,PRIMARY); Using where

- **使用UNION优化**

```
 EXPLAIN SELECT * FROM emp WHERE id = 1 UNION SELECT * FROM emp WHERE id =10;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	emp	const	PRIMARY	PRIMARY	4	const	1	
2	UNION	emp	const	PRIMARY	PRIMARY	4	const	1	
NULL	UNION RESULT	<union1,2>	ALL	NULL	NULL	NULL	NULL	NULL	

分页查询优化

- 一般分页查询时，通过创建覆盖索引能够比较好地提高性能。一个常见又非常头疼的问题就是 limit 2000000,10 ，此时需要MySQL排序前2000010 记录，仅仅返回2000000 - 2000010 的记录，其他记录丢弃，查询排序的代价非常大 。

优化思路1

- 在索引上完成排序分页操作，最后根据主键关联回原表查询所需要的其他列内容

```
EXPLAIN SELECT * FROM tb_item a ,(SELECT id FROM tb_item ORDER BY id LIMIT 2000000,10) b WHERE a.id = b.id;
```

优化思路2

- 该方案适用于主键自增的表，可以把Limit 查询转换成某个位置的查询

```
EXPLAIN SELECT * FROM tb_item WHERE id > 2000000 LIMIT 10 ;
```

使用SQL提示

- SQL提示，是优化数据库的一个重要手段，简单来说，就是在SQL语句中加入一些人为的提示来达到优化操作的目的。

USE INDEX

- 在查询语句中表名的后面，添加 use index 来提供希望MySQL去参考的索引列表，就可以让MySQL不再考虑其他可用的索引。

```
EXPLAIN SELECT * FROM 表名 USE INDEX(索引名) [WHERE 条件];
示例:
EXPLAIN SELECT * FROM ACCOUNT USE INDEX(index_id) WHERE money > 1000;
```

IGNORE INDEX

- 想让MySQL忽略一个或者多个索引，则可以使用 ignore index 作为 hint 。

```
EXPLAIN SELECT * FROM 表名 IGNORE INDEX(索引名) [WHERE 条件];
示例:
EXPLAIN SELECT * FROM ACCOUNT IGNORE INDEX(index_id) WHERE money > 1000;
```

FORCE INDEX

- 强制MySQL使用一个特定的索引，可在查询中使用 force index 作为hint

```
EXPLAIN SELECT * FROM 表名 FORCE INDEX(索引名) [WHERE 条件];
示例:
EXPLAIN SELECT * FROM ACCOUNT FORCE INDEX(index_id) WHERE money > 1000;
```

系统性能优化

应用优化

减少对MySQL的访问

1. 避免对数据进行重复检索

```
在编写应用代码时，需要能够理清对数据库的访问逻辑。能够一次连接就获取到结果的，就不用两次连接，这样可
以大大减少对数据库无用的重复请求。
```

2. 增加cache层

```
在应用中，我们可以在应用中增加 缓存 层来达到减轻数据库负担的目的。缓存层有很多种，也有很多实现方式，
只要能达到降低数据库的负担又能满足应用需求就可以。
因此可以部分数据从数据库中抽取出来放到应用端以文本方式存储， 或者使用框架(Mybatis, Hibernate)提供的一级缓存/二级缓存，或者使用redis数据库来缓存数据 。
```

负载均衡

1.利用MySQL复制分流查询

```
通过MySQL的主从复制，实现读写分离，使增删改操作走主节点，查询操作走从节点，从而可以降低单台服务器的
读写压力。
```

2. 采用分布式数据库架构

```
分布式数据库架构适合大数据量、负载高的情况，它有良好的拓展性和高可用性。通过在多台服务器之间分布数
据，可以实现在多台服务器之间的负载均衡，提高访问效率。
```

内存优化

内存优化原则

```
1） 将尽量多的内存分配给MySQL做缓存，但要给操作系统和其他程序预留足够内存。

2） MyISAM 存储引擎的数据文件读取依赖于操作系统自身的IO缓存，因此，如果有MyISAM表，就要预留更多的
内存给操作系统做IO缓存。

3） 排序区、连接区等缓存是分配给每个数据库会话（session）专用的，其默认值的设置要根据最大连接数合理
分配，如果设置太大，不但浪费资源，而且在并发连接较高时会导致物理内存耗尽。
```

MyISAM 内存优化

- myisam存储引擎使用 key_buffffer 缓存索引块，加速myisam索引的读写速度。对于myisam表的数据块，mysql没有特别的缓存机制，完全依赖于操作系统的IO缓存。

key_buffer_size

- key_buffffer_size决定MyISAM索引块缓存区的大小，直接影响到MyISAM表的存取效率。可以在MySQL参数文件中设置key_buffffer_size的值，对于一般MyISAM数据库，建议至少将1/4可用内存分配给key_buffffer_size。

  在/usr/my.cnf 中做如下配置：

```
key_buffer_size=512M
```

read_rnd_buffer_size

- 如果需要经常顺序扫描myisam表，可以通过增大read_buffffer_size的值来改善性能。但需要注意的是

  read_buffffer_size是每个session独占的，如果默认值设置太大，就会造成内存浪费。

read_rnd_buffer_size

- 对于需要做排序的myisam表的查询，如带有order by子句的sql，适当增加 read_rnd_buffffer_size 的值，可以改善此类的sql性能。但需要注意的是 read_rnd_buffffer_size 是每个session独占的，如果默认值设置太大，就会造成内存浪费。

InnoDB 内存优化

- innodb用一块内存区做IO缓存池，该缓存池不仅用来缓存innodb的索引块，而且也用来缓存innodb的数据块。

innodb_buffffer_pool_size

- 该变量决定了 innodb 存储引擎表数据和索引数据的最大缓存区大小。在保证操作系统及其他程序有足够内存可用的情况下，innodb_buffer_pool_size 的值越大，缓存命中率越高，访问InnoDB表需要的磁盘I/O 就越少，性能也就越高。

```
innodb_buffer_pool_size=512M
```

innodb_log_buffffer_size

- 决定了innodb重做日志缓存的大小，对于可能产生大量更新记录的大事务，增加 innodb_log_buffffer_size的大小，可以避免innodb在事务提交前就执行不必要的日志写入磁盘操作。

```
innodb_log_buffer_size=10M
```

并发参数调整

- 从实现上来说，MySQL Server 是多线程结构，包括后台线程和客户服务线程。多线程可以有效利用服务器资源，提高数据库的并发性能。在Mysql中，控制并发连接和线程的主要参数包括 max_connections、back_log、thread_cache_size、table_open_cahce。

max_connections

采用max_connections 控制允许连接到MySQL数据库的最大数量，默认值是 151。如果状态变量

connection_errors_max_connections 不为零，并且一直增长，则说明不断有连接请求因数据库连接数已达到允

许最大值而失败，这是可以考虑增大max_connections 的值。

Mysql 最大可支持的连接数，取决于很多因素，包括给定操作系统平台的线程库的质量、内存大小、每个连接的负荷、CPU的处理速度，期望的响应时间等。在Linux 平台下，性能好的服务器，支持 500-1000 个连接不是难事，需要根据服务器性能进行评估设定。

back_log

back_log 参数控制MySQL监听TCP端口时设置的积压请求栈大小。如果MySql的连接数达到 max_connections时，新来的请求将会被存在堆栈中，以等待某一连接释放资源，该堆栈的数量即 back_log，如果等待连接的数量超过back_log，将不被授予连接资源，将会报错。5.6.6 版本之前默认值为 50， 之后的版本默认为 50 +（max_connections / 5）， 但最大不超过900。如果需要数据库在较短的时间内处理大量连接请求， 可以考虑适当增大back_log 的值。

table_open_cache

该参数用来控制所有SQL语句执行线程可打开表缓存的数量， 而在执行SQL语句时，每一个SQL执行线程至少要打

开 1 个表缓存。该参数的值应该根据设置的最大连接数 max_connections 以及每个连接执行关联查询中涉及的表

的最大数量来设定： max_connections x N；

thread_cache_size

为了加快连接数据库的速度，MySQL 会缓存一定数量的客户服务线程以备重用，通过参数 thread_cache_size 可

控制 MySQL 缓存客户服务线程的数量。

innodb_lock_wait_timeout

该参数是用来设置InnoDB 事务等待行锁的时间，默认值是50ms， 可以根据需要进行动态设置。对于需要快速反

馈的业务系统来说，可以将行锁的等待时间调小，以避免事务长时间挂起； 对于后台运行的批量处理程序来说，

可以将行锁的等待时间调大， 以避免发生大的回滚操作。