MySQL学习笔记【基础篇】

> 从前曾经学过一下`mysql`的基础内容，不过由于当时没有认真学导致会的东西太少，现根据一个教程视频的学习【此内容对应基础篇的`P1-P178`】，完成了这个笔记，主要涉及`mysql`的基础知识。

一、数据库相关概念

1、`DB`：数据库，保存一组有组织的数据的容器
2、`DBMS`：数据库管理系统，又称为数据库软件（产品），用于管理DB中的数据
3、`SQL`:结构化查询语言，用于和DBMS通信的语言

二、数据库的好处

1.持久化数据到本地

2.可以实现结构化查询，方便管理

三、数据库存储数据的特点

1、将数据放到表中，表再放到库中
2、一个数据库中可以有多个表，每个表都有一个的名字，用来标识自己。表名具有唯一性。
3、表具有一些特性，这些特性定义了数据在表中如何存储，类似java中 “类”的设计。
4、表由列组成，我们也称为字段。所有表都是由一个或多个列组成的，每一列类似java 中的”属性”
5、表中的数据是按行存储的，每一行类似于java中的“对象”。

四、MySQL服务的启动和停止

方式一：计算机——右击管理——服务
方式二：通过管理员身份运行，输入

```
net start 服务名（启动服务）
net stop 服务名（停止服务）
```

服务名是根据你当初的设定决定的，如果你忘了，可以参考这个链接。

你的服务名是`MySQL57`

五、mysql服务的登录和退出

方式一：通过mysql自带的客户端
只限于root用户

方式二：通过windows自带的客户端
登录：
`mysql 【-h主机名 -P端口号 】-u用户名 -p密码`

退出：
`exit或ctrl+C`

六、MySQL的常用命令

1.查看当前所有的数据库

`show databases;`

2.打开指定的库

`use 库名`

3.查看当前库的所有表

`show tables;`

4.查看其它库的所有表

`show tables from 库名;`

5.创建表

```
create table 表名(

    列名 列类型,
    列名 列类型,
    。。。
);
```

6.查看表结构

`desc 表名;`

7.查看服务器的版本

方式一：登录到mysql服务端

`select version();`

方式二：没有登录到mysql服务端（在cmd控制台上输入）

`mysql --version`

或

`mysql --V`

七、MySQL的语法规范

1.不区分大小写,但建议关键字大写，表名、列名小写

2.每条命令最好用分号结尾

3.每条命令根据需要，可以进行缩进或换行

4.注释

单行注释：`#注释文字`

单行注释：`-- 注释文字`

多行注释：`/* 注释文字  */`

八、SQL的语言分类

```
DQL（Data Query Language）：数据查询语言
    select
DML(Data Manipulate Language):数据操作语言
    insert 、update、delete
DDL（Data Define Languge）：数据定义语言
    create、drop、alter
TCL（Transaction Control Language）：事务控制语言
    commit、rollback
```

九、DQL语言的学习

基础查询

语法：

```
SELECT 查询列表 FROM 表名;
```

注意：

- 查询列表可以是：表中的字段、常量值、表达式、函数。
- 查询的结果是一个虚拟的表格。
- 常量值可以用**单引号双引号**引上，**字段值**可以用(键盘上数字1左边那个键的符号引上)，注意二者区别。一般情况下可以不用引，当字段与关键字重名或者是特殊符号时引上比较好。

举例：

1.查询表中的单个字段

```
SELECT last_name FROM employees;
```

2.查询表中的多个字段

```
SELECT last_name, salary, email FROM employees;
```

3.查询表中的所有字段

```
SELECT * FROM employees;
```

4.查询常量值

```
SELECT 100;
```

```
SELECT 'john';
```

5.查询表达式

```
SELECT 100%98;
```

6.查询函数

```
SELECT VERSION();
```

7.起别名

方式一：

```
SELECT 100%98 as 结果;
```

```
SELECT last_name AS 姓,first_name AS 名 FROM employees;
```

起别名好处：

- 便于理解。
- 如果要查询的字段有重名的情况，可以使用别名区别开来。

方式二：

```
SELECT last_name 姓,first_name 名 FROM employees;
```

8.去重

案例：查询员工表中涉及到的所有部门编号。

`SELECT DISTINCT department_id FROM employees;`

9.+号的作用

注意mysql中+号只有一个功能，充当运算符，不能连接字符串。

```
select 100+90;如果两个操作数均为数值型，则做加法运算。
select '123' + 90;  若其中一方为字符型，则试图将字符型数值转化成数值型
                        如果转换成功，则继续做加法运算；
                        否则，则将字符型转换成0
//select '123'+90的结果为213
select 'john'+90;       //结果为90

select null+10;         只要其中一方为null，则结果肯定为null
```

案例：查询员工`名`和`姓`连接成一个字段，并显示为`姓名`。

既然+号不能起到连接字段的作用，我们可以利用`concat`函数：

`SELECT CONCAT(last_name,first_name) AS 姓名 FROM employees;`

括号内的参数可以为多个。

10.concat函数

功能：拼接字符

`select concat(字符1，字符2，字符3，...);`

11.ifnull函数

功能：判断某字段是否为null，如果为null返回指定的值，否则返回原来的值。

`select ifnull(commission_pct,0) from employees;`

12.isnull函数

功能：判断该字段或表达式的值是否为null，是返回1，否则返回0

条件查询

语法：

`select 查询列表 from 表名 where 条件;`

条件的分类：

- 条件表达式
    - 示例：`salary>10000`
    - 条件运算符：`> < >= <= = !=`
- 逻辑表达式
    - 示例：salary>10000 && salary<20000
    - 逻辑运算符：
        - and（&&）:两个条件如果同时成立，结果为true，否则为false；
        - or(||)：两个条件只要有一个成立，结果为true，否则为false；
        - not(!)：如果条件成立，则not后为false，否则为true；
- 模糊查询
    - 运算符

- like
    - 一般和通配符搭配使用。
    - 通配符有：
        - `%` 表示任意多个字符，包含0个字符
        - `_` 表示任意单个字符
- between and
    - 注意它包含临界值
    - 两个临界值不要随意调换顺序
- in
    - 用于判断某字段的值是否属于in列表中的某一项
    - in列表的值类型必须统一或兼容,比如'123'可以转换成123
    - 不支持通配符
- is null
    - `=`或者`<>`不能用于判断null值
    - is null或is not null可以判断null值

- **实例：**
    - 查询员工名中包含字符a的员工信息
        - `SELECT * FROM employees WHERE last_name LIKE '%a%';`
    - 查询员工名中第三个字符为e,第五个字符为a的员工名和工资(第几个就在前面画几个下划线，注意不要忘记末尾要加个百分号，表示后面还有0个或多个字符)
        - `SELECT last_name, salary FROM employees WHERE last_name LIKE '__n_l%';`
    - 查询员工名中第二个字符为`_`的员工名(由于通配符中有下划线，这就需要用到转义字符或者用escape说明，比如下面那个说明`$`符后面的下划线不作为通配符)
        - `SELECT * FROM employees WHERE last_name LIKE '_\_%';`
        - `SELECT * FROM employees WHERE last_name LIKE '_$_%' ESCAPE '$';` **（推荐使用第二种方法）**
    - 查询员工编号在100到120之间的员工信息
        - `SELECT * FROM employees WHERE employee_id BETWEEN 100 AND 120`
    - 查询员工的工种编号是`IT_PROG、AD_VP、AD_PRES`中的员工名和工种编号
        - `SELECT last_name, job_id FROM employees WHERE job_id IN ('IT_PROG','AD_VP','AD_PRES');`
    - 查询没有奖金的员工名和奖金率
        - `SELECT last_name, commission_pct FROM employees WHERE commission_pct IS NULL;`
    - 查询有奖金的员工名和奖金率
        - `SELECT last_name, commission_pct FROM employees WHERE commission_pct IS NOT NULL;`

安全等于 `<=>`

- 比如

    - `SELECT last_name, commission_pct FROM employees WHERE commission_pct <=> NULL;`
    - `SELECT last_name, commission_pct FROM employees WHERE salary <=> 12000;`
- 不过它用的较少，可读性差

`is null` 与 `<=>`的比较：

- `is null`:仅仅可以判断NULL值。可读性较高。
- `<=>` :该可以判断NULL值，又可以判断普通的数值。可读性较低

阶段测试：

```
查询员工号为176的员工的姓名和部门号和年薪：
SELECT last_name,department_id,salary*12*(1+IFNULL(commission_pct, 0)) AS 年薪 FROM employees;
```

排序查询

语法：

- `select 查询列表 from 表名 【where 筛选条件】 order by 排序列表 【asc | desc】;`
- 注意上面的asc表示升序，desc表示降序。如果不写，默认是升序，不过前提是你写了 `order by`。
- order by子句中可以支持单个字段、多个字段、表达式、函数、别名。
- order by子句一般放在查询语句的最后面，不过limit子句除外。

案例：

1.查询员工信息，要求工资从高到低排序。

`SELECT * FROM employees ORDER BY salary DESC;`

2.查询部门编号>=90的员工信息，要求按入职时间的先后进行排序。

`SELECT * FROM employees WHERE department_id >= 90 ORDER BY hiredate ASC;`

3.【按表达式排序】按年薪的高低显示员工的信息和年薪。

`SELECT *,salary*12*(1+IFNULL(commission_pct,0)) 年薪 FROM employees ORDER BY salary*12*(1+IFNULL(commission_pct,0)) DESC;`

4.【按别名排序】按年薪的高低显示员工的信息和年薪。

`SELECT *,salary*12*(1+IFNULL(commission_pct,0)) 年薪 FROM employees ORDER BY 年薪 DESC;`

5.按姓名的长度显示员工的员工和工资【按函数排序】

`SELECT LENGTH(last_name)字节长度,last_name,salary FROM employees ORDER BY 字节长度 DESC;`

6.查询员工信息，要求先按工资升序排序，如果一样的话再按员工编号降序排序【按多个字段排序】

`SELECT * FROM employees ORDER BY salary ASC,employee_id DESC;`

常见函数

单行函数

1.字符函数

(1)length：获取参数值的字节个数

```
SELECT LENGTH('john');
```

(2)concat：拼接字符串

```
SELECT CONCAT(last_name,'_',first_name) FROM employees;
```

(3)upper、lower

小案例：将表中姓大写，名小写，然后拼接。

```
SELECT CONCAT(UPPER(last_name),LOWER(first_name)) 姓名 FROM employees;
```

(4)substr、substring

注意：索引从1开始。

截取从指定索引处后面的所有字符：

```
SELECT SUBSTR('李莫愁爱上了陆展元',7) out_put;
```

截取从指定索引处指定字符长度的字符：

```
SELECT SUBSTR('李莫愁爱上了陆展元',1,3) out_put;
```

小案例：

将姓名中首字符大写，其他字符小写然后用_拼接，显示出来。

```
SELECT CONCAT(UPPER(SUBSTR(last_name,1,1)),'_',LOWER(SUBSTR(last_name,2))) out_put FROM employees;
```

(5)instr：返回子串第一次出现的索引，如果找不到则返回0

```
SELECT INSTR('杨不悔爱上了殷六侠','殷八侠') AS out_put ;
```

(6)trim

```
SELECT LENGTH(TRIM('       张存山          ')) AS out_put;
```

```
SELECT TRIM('a' FROM 'aaa张aaaaaa翠aaaaaaaa山') out_put;
```

(7)lpad：用指定的字符实现左填充指定长度

```
SELECT LPAD('因素是',10,'*') AS out_put;
```

(8)rpad：用指定的字符实现右填充指定长度

```
SELECT RPAD('订单',12,'abb') AS out_put;
```

(9)replace

```
SELECT REPLACE('张无忌爱上了周芷若周芷若周芷若周芷若','周芷若','赵敏')AS out_put;
```

2.数学函数

(1)round：　四舍五入

```
SELECT ROUND(1.65);
```

```
SELECT ROUND(1.567,2);
```

(2)ceil：向上取整

```
SELECT CEIL(1.52);
```

(3)floor：向下取整

```
SELECT FLOOR(9.99);
```

(4)truncate：截断

```
SELECT TRUNCATE(1.69,1);
```

(5)mod：取余

```
SELECT MOD(-10,-3);
```

3.日期函数

(1)now：返回当前系统日期

```
SELECT NOW();
```

(2)curdate：返回当前系统日期，不包含时间

```
SELECT CURDATE();
```

(3)curtime：返回当前时间，不包含日期

```
SELECT CURTIME();
```

(4)获取指定的部分，年、月、日、小时、分钟、秒。

```
SELECT YEAR(NOW()) 年;
```

```
SELECT MONTH(NOW()) 月;
```

```
SELECT MONTHNAME(NOW()) 月;
```

举个例子：获取employees表中的时间中的年。

```
SELECT YEAR(hiredate) 年 FROM employees;
```

(5)str_to_date：将字符通过指定的格式转换成日期。

语法：

前一个为日期字符串，后一个为日期格式，日期格式可以从下列表中挑选。

序号	格式符	功能
1	%Y	四位的年份
2	%y	2位的年份
3	%m	月份（01,02...11,12）
4	%c	月份（1,2,...11,12）
5	%d	日（01,02,...）
6	%H	小时（24小时制）
7	%h	小时（12小时制）
8	%i	分钟（00,01...59）
9	%s	秒（00,01,...59）

举个例子：

```
SELECT STR_TO_DATE('1998-3-2','%Y-%c-%d') AS out_put;
```

小案例:查询入职日期为`4-3 1992`的员工信息

```
SELECT * FROM employees WHERE hiredate = STR_TO_DATE('4-3 1992', '%c-%d %Y');
```

(6)date_format：将日期转换成字符

```
SELECT DATE_FORMAT(NOW(),'%y年%m月%d日') AS out_put;
```

小案例：查询有奖金的员工名和入职日期(xx月/xx日　xx年)

```
SELECT last_name,DATE_FORMAT(hiredate,'%m月/%d日 %y年') FROM employees WHERE commission_pct IS NOT NULL;
```

4.流程控制函数

(1)if函数：if else的效果

```
SELECT IF(10>5, '大', '小');
```

```
SELECT last_name, commission_pct,IF(commission_pct IS NULL, '没奖金','有奖金') 备注 FROM employees;
```

(2)case函数的使用一：switch case的效果

语法：

```
case 要判断的字段或表达式
when 常量1 then 要显示的值1或语句1            //注意：如果要显示语句，则需要在语句后面加分号
when 常量2 then 要显示的值2或语句2
...
else 要显示的值n或语句n
end
```

案例：

查询员工的工资，要求

部门号=30，显示的工资为1.1倍；

部门号=40，显示的工资为1.2倍；

部门号=50，显示的工资为1.3倍；

其他部门，显示的工资为原工资。

```
SELECT salary 原始工资,department_id,
CASE department_id
WHEN 30 THEN salary*1.1
WHEN 40 THEN salary*1.2
WHEN 50 THEN salary*1.3
ELSE salary
END AS 新工资
FROM employees;
```

(3)case函数的使用二：类似 多重if

语法：

```
case
when 条件1 then 显示的值1或语句1                    //注意：如果要显示语句，则需要在语句后面加分号
when 条件2 then 要显示的值2或语句2
...
else 要显示的值n或语句n
end
```

案例：

查询员工的工资的情况，

如果工资>20000，显示A级别；

如果工资>15000，显示B级别；

如果工资>10000，显示C级别；

否则，显示D级别；

```
SELECT salary,
CASE
WHEN salary > 20000 THEN 'A'
WHEN salary > 15000 THEN 'B'
WHEN salary > 10000 THEN 'C'
ELSE 'D'
END AS 工资级别
FROM employees;
```

分组函数

功能：用作统计使用，又称为聚合函数或统计函数或组函数。

分类：

- sum　求和
- avg 平均值
- max　最大值

- min 最小值
- count 计算个数。

简单使用：

`SELECT SUM(salary) FROM employees;`

`SELECT AVG(salary) FROM employees;`

`SELECT COUNT(salary) FROM employees;`

`SELECT SUM(salary)和,AVG(salary)平均,MAX(salary)最高,MIN(salary)最低,COUNT(salary)个数 FROM employees;`

特点：

- sum和avg一般用于处理数值型
- max、min、count可以处理任何数据类型
- count的参数可以支持：字段、`*`、【常量值，一般放1】，不过建议使用`count(*)`
    - `SELECT COUNT(*) FROM employees;`
    - 查询部门编号为90的员工个数
        - `SELECT COUNT(*) FROM employees WHERE department_id = 90;`
- 它们都忽略null值
- 可以和distinct搭配实现去重的运算
    - `SELECT SUM(DISTINCT salary),SUM(salary) FROM employees;`
    - `SELECT COUNT(DISTINCT salary), COUNT(salary) FROM employees;`
- 和分组函数一同查询的字段要求是group by后的字段。

分组查询

语法：

`select 分组函数,列(要求出现在group by的后面) from 表 【where 筛选条件】group by 分组的列表 【order by子句】`

注意：

查询列表必须特殊，要求是分组函数和group by后出现的字段

简单的分组查询：

- 案例1：查询每个工种的最高工资。
    - `SELECT MAX(salary),job_id FROM employees GROUP BY job_id;`
- 案例2：查询每个位置上的部门个数。
    - `SELECT COUNT(*),location_id FROM departments GROUP BY location_id;`

【进阶】添加分组前的筛选条件：

- 案例1：查询邮箱中包含a字符的，每个部门的平均工资。
    - `SELECT AVG(salary),department_id FROM employees WHERE email LIKE '%a%' GROUP BY department_id;`
- 案例2：查询有奖金的每个领导手下员工的最高工资。
    - `SELECT MAX(salary),manager_id FROM employees WHERE commission_pct IS NOT NULL GROUP BY manager_id;`

【再进阶】添加分组后的筛选条件(用到了HAVING)

- 案例1：查询哪个部门的员工个数`>2`。
    - `SELECT COUNT(*),department_id FROM employees GROUP BY department_id HAVING COUNT(*)>2;`
- 案例2：查询每个工种有奖金的员工的最高工资>12000的工种编号和最高工资。
    - 思路
        - ①查询每个工种有奖金的的员工的最高工资；
        - ②根据①的结果继续筛选，最高工资>12000
    - `SELECT MAX(salary),job_id FROM employees WHERE commission_pct IS NOT NULL GROUP BY job_id HAVING MAX(salary)>12000;`
- 案例3：查询领导编号>102的每个领导手下的最低工资>5000的领导编号是哪个，以及其最低工资。
    - ```
      SELECT MIN(salary),manager_id
      FROM employees
      WHERE manager_id > 102
      GROUP BY manager_id
      HAVING MIN(salary)>5000;
      ```

按表达式或函数分组

- 案例：按员工姓名的长度分组，查询每一组的员工个数，筛选员工个数>5的有哪些。
    - ①查询每个长度的员工个数
        - ```
          SELECT COUNT(*),LENGTH(last_name) len_name
          FROM employees
          GROUP BY LENGTH(last_name);
          ```
    - ②添加筛选条件
        - ```
          SELECT COUNT(*),LENGTH(last_name) len_name
          FROM employees
          GROUP BY LENGTH(last_name)
          HAVING COUNT(*)>5;

          -- 或者这样写：

          SELECT COUNT(*) cnt,LENGTH(last_name) len_name
          FROM employees
          GROUP BY len_name
          HAVING cnt>5;

          -- 发现group by支持别名,按where不支持
          ```

按多个字段进行分组

- 案例：查询每个部门每个工种的员工的平均工资。
    - ```
      SELECT AVG(salary),department_id,job_id
      FROM employees
      GROUP BY department_id, job_id;
      ```

添加排序

- 案例：查询每个部门每个工种的员工的平均工资，并且按平均工资的高低显示。
    - ```
      SELECT AVG(salary),department_id,job_id
      FROM employees
      GROUP BY department_id, job_id
      ORDER BY AVG(salary) DESC;
      ```

特点：

- 分组查询中的筛选条件分为两类
    - 分组前筛选：其筛选源是原始表，放在group by子句的前面，用到了where关键字。
    - 分组后筛选：其筛选表是分组后的结果集合，放在group by子句的后面，用到了having关键字。
- 分组函数做条件时肯定是放在having子句后面
- 能用分组前筛选的，就优先考虑分组前筛选
- group by子句支持单个字段分组，多个字段分组（多个字段之间用逗号隔开，没有顺序要求)
- 也可以添加排序(排序放在整个分组查询的最后)

连接查询

含义：

又成为多表查询，当查询的字段来自多个表时，就会用到连接查询。

分类【按年代分类】：

- sql92标准：仅仅支持内连接
- sql99标准：支持内连接+外连接(左外和右外)+交叉连接

分类【按功能分类】：

- 内连接
    - 等值连接
    - 非等值连接
    - 自连接
- 外连接
    - 左外连接
    - 右外连接
    - 全外连接
- 交叉连接

sql92标准

1.等值连接

案例1：查询女神名和对应的男神名。

```
SELECT NAME,boyName
FROM boys, beauty
WHERE beauty.`boyfriend_id`=boys.id;
```
```

案例2：查询员工名和对应的部门名。

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

案例3：查询员工名、工种号、工种名。

```
-- 当emplyees,jobs表中都有job_id字段时，需要指明它来自哪个表，否则会执行错误。
SELECT last_name, employees.job_id, job_title
FROM employees,jobs
WHERE employees.`job_id`=jobs.`job_id`;
```

我们也可以给表起别名，用不用as都可以，如下：

```
-- as可省略
SELECT last_name, e.job_id, job_title
FROM employees AS e,jobs AS j
WHERE e.`job_id`=j.`job_id`;
```

注意：如果为表起了别名，则查询的字段不能使用原来的表名去限定。

案例4：查询有奖金的员工名、部门名。

```
SELECT last_name,department_name,commission_pct
FROM employees e,departments d
WHERE e.`department_id`=d.`department_id`
AND e.`commission_pct` IS NOT NULL;
```

案例5：查询城市中第二个字符为o的部门名和城市名。

```
SELECT department_name,city
FROM departments d,locations l
WHERE d.`location_id`=l.`location_id`
AND l.`city` LIKE '_o%';
```

案例6：查询每个城市的部门个数。

```
SELECT city,COUNT(*) 个数
FROM locations l, departments d
WHERE l.`location_id`=d.`location_id`
GROUP BY city;
```

案例7：查询有奖金的每个部门的部门名和部门的领导编号和该部门的最低工资。

```
SELECT department_name,e.manager_id,MIN(salary)
FROM employees e, departments d
WHERE e.`department_id`=d.`department_id`
AND e.`commission_pct` IS NOT NULL
GROUP BY d.`department_name`, d.`manager_id`;
```

案例8：查询每个工种的工种名和员工的个数，并且按员工个数降序。

```sql
SELECT job_title,COUNT(*)
FROM employees e, jobs j
WHERE e.`job_id`=j.`job_id`
GROUP BY j.`job_title`
ORDER BY COUNT(*) DESC;
```

案例9：查询员工名、部门名和所在的城市。

```sql
SELECT last_name,department_name,city
FROM employees e, departments d, locations l
WHERE e.`department_id`=d.`department_id`
AND d.`location_id`=l.`location_id`;
```

总结一下等值连接的特点：

- 多表等值连接的结果为多表的交集部分；
- n表连接，至少需要n-1个连接条件；
- 多表的顺序没有要求；
- 一般需要为表起别名；
- 可以搭配前面介绍的所有子句，比如排序、分组、筛选。

2.非等值连接

案例：查询员工的工资和工资级别。

```sql
SELECT salary,grade_level
FROM employees e, job_grades g
WHERE salary BETWEEN g.`lowest_sal` AND g.`highest_sal`;
```

3.自连接

案例：查询员工名和上级的名字。

```sql
-- 可以把employees看成两个表e，m，e是员工表，m是上级表。
SELECT e.employee_id ,e.last_name,m.employee_id,m.last_name
FROM employees e, employees m
WHERE e.manager_id = m.employee_id;
```

sql99标准

语法

```
select 字段，...
from 表1 别名
【inner|left outer|right outer|cross】join 表2 别名 on  连接条件
【inner|left outer|right outer|cross】join 表3 on  连接条件
【where 筛选条件】
【group by 分组字段】
【having 分组后的筛选条件】
【order by 排序的字段或表达式】
```

分类

- 内连接：inner
    - 等值连接
    - 非等值连接
    - 自连接
- 外连接
    - 左外：left 【outer】
    - 右外：right【outer】
    - 全外：full【outer】
- 交叉连接：cross

内连接

语法

```
select 查询列表
from 表1 别名
inner join 表2 别名
on 连接条件;
```

1.等值连接

特点：

- 可以添加排序、分组、筛选；
- inner可以省略；
- 筛选条件放在where后面，连接条件放在on后面，提高分离性，便于阅读；
- inner join连接和sql92语法中的等值连接效果是一样的，都是查询多表的交集。

案例1：查询员工名、部门名

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

案例2：查询员工名和对应的部门名。（筛选）

```
SELECT last_name,job_title
FROM employees e
INNER JOIN jobs j
ON e.`job_id`=j.`job_id`
WHERE e.`last_name` LIKE '%e%';
```

案例3：查询部门个数>3的城市名和部门个数。（添加分组+筛选）

```sql
-- 根据要求，查询部门个数>3的城市名和部门个数，也就是说先按照城市名进行分组，然后再用部门个数>3这个条件进行筛选（这就用到了having）

SELECT city,COUNT(*) 部门个数
FROM departments d
INNER JOIN locations l
ON d.`location_id`=l.`location_id`
GROUP BY city
HAVING COUNT(*)>3;
```

案例4：查询部门的员工个数>3的部门名和员工个数，并按个数降序。（添加排序）

```sql
-- ①查询每个部门的员工个数
select count(*),department_name
from employees
INNER JOIN departments d
ON e.`department_id`=d.`department_id`
group by department_name
-- ②在①的结果上选员工数>3的记录，并排序
SELECT department_name 部门名,COUNT(*)
FROM employees e
INNER JOIN departments d
ON e.`department_id`=d.`department_id`
GROUP BY d.`department_id`
HAVING COUNT(*)>3
ORDER BY COUNT(*) DESC;
```

案例5：查询员工名、部门名、工种名，并按部门名降序。（三表连接）

```sql
SELECT last_name,department_name,job_title
FROM employees e
INNER JOIN departments d ON e.`department_id`=d.`department_id`
INNER JOIN jobs j ON e.`job_id`=j.`job_id`
ORDER BY department_name DESC;
```

2.非等值连接

案例1：查询员工的工资级别。

```sql
SELECT salary,grade_level
FROM employees e
JOIN job_grades g
ON e.`salary` BETWEEN g.`lowest_sal` AND g.`highest_sal`;
```

案例2：查询每个工资级别的个数>20的，并且按工资级别降序排序。

```sql
SELECT COUNT(*) 个数,grade_level
FROM employees e
JOIN job_grades g
ON e.`salary` BETWEEN g.`lowest_sal` AND g.`highest_sal`
GROUP BY grade_level
HAVING COUNT(*)>20
ORDER BY grade_level DESC;
```

3.自连接

案例1：查询员工的名字、上级的名字。

```
SELECT e.last_name 员工名,m.last_name 上级名
FROM employees e
JOIN employees m
ON e.`manager_id`=m.`employee_id`;
```

案例2：查询姓名中包含字符k的员工的名字，以及对应的上级的名字。

```
SELECT e.last_name 员工名,m.last_name 上级名
FROM employees e
JOIN employees m
ON e.`manager_id`=m.`employee_id`
WHERE e.`last_name` LIKE '%k%';
```

外连接(这里没有介绍全外连接)

应用场景：用于查询一个表中有，另一个表中没有的记录。

特点：

- 外连接的查询结果为`主表`中的所有记录，如果`从表`中有和它匹配的，则显示匹配的值；若没有，则显示null。即：外连接查询结果=内连接结果+`主表`中有而`从表`中没有的记录。
- 左外连接中，left join左边的是主表；右外连接中，right join 右边的是主表。

引入：查询男朋友不在男神表的女神名。

```
SELECT b.name 女神名
FROM beauty b
LEFT OUTER JOIN boys bo
ON b.`boyfriend_id`=bo.`id`
WHERE bo.`id` IS NULL;
```

或者

```
SELECT b.name 女神名
FROM boys bo
RIGHT OUTER JOIN beauty b
ON b.`boyfriend_id`=bo.`id`
WHERE bo.`id` IS NULL;
```

案例：查询哪个部门没有员工。

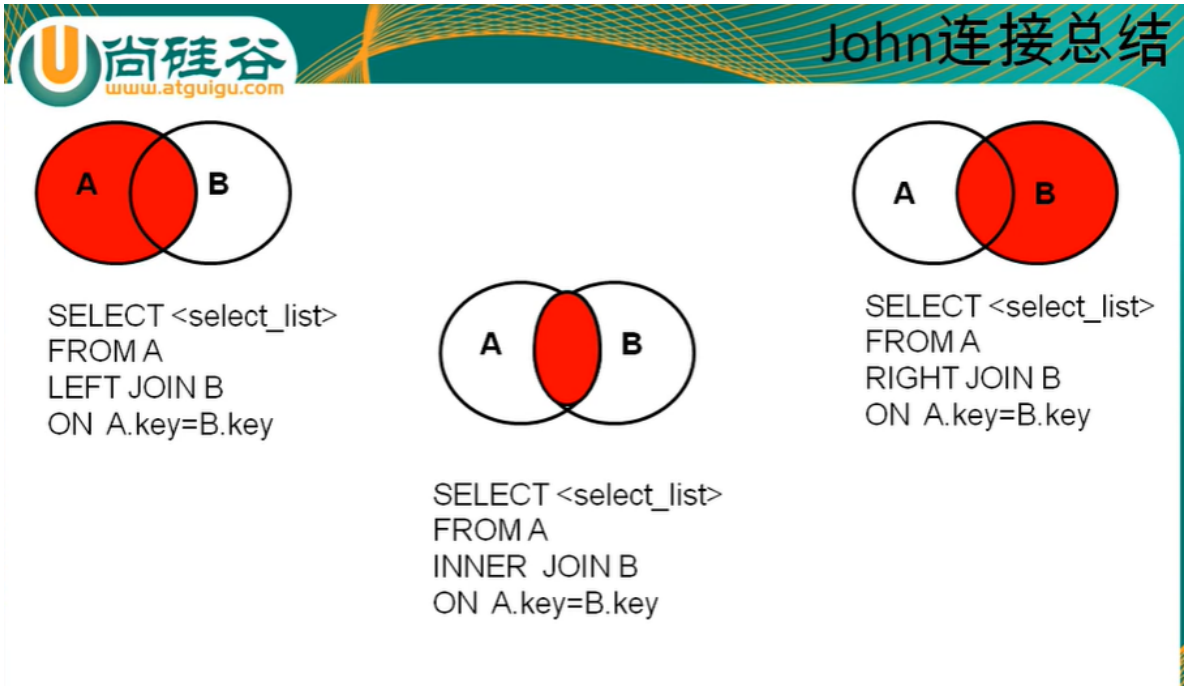
```
SELECT d.*,e.employee_id
FROM departments d
LEFT OUTER JOIN employees e
ON d.`department_id`=e.`department_id`
WHERE e.`employee_id` IS NULL;
```

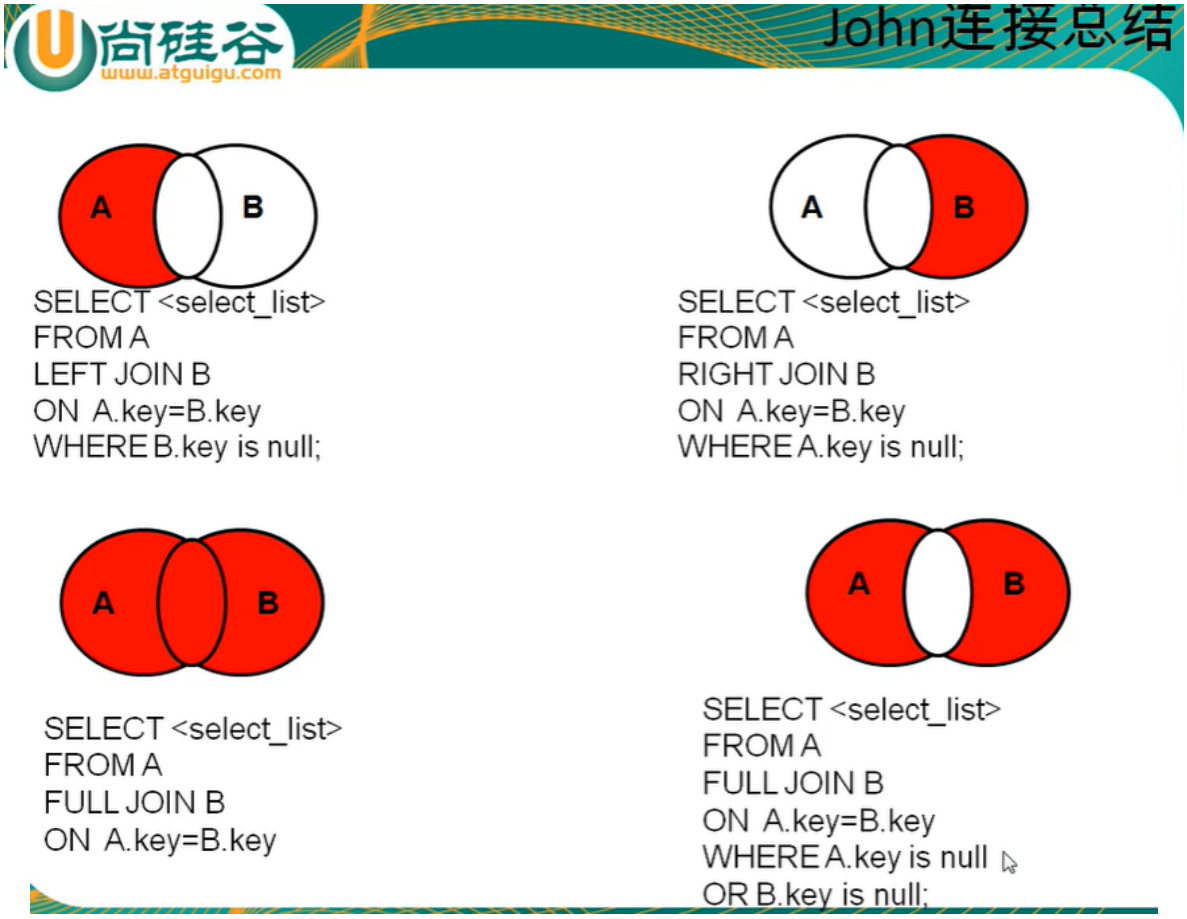
交叉连接

就是使用sql99标准实现两个表的笛卡尔乘积。如：

```sql
SELECT b.*,bo.*
FROM beauty b
CROSS JOIN boys bo;
```

join连接总结





综合案例

1.查询编号>3的女神的男朋友信息，如果有则列出详细，如果没有，用NULL填充。

```sql
-- 根据要求，查询女神的男朋友信息，如果没有则用NULL填充，从这里可以看出女神表beauty为主表，男神表boys为从表。
--下面使用左外连接 left左面的为主表
SELECT b.id,b.`name`,bo.*
FROM beauty b
LEFT OUTER JOIN boys bo
ON b.`boyfriend_id`=bo.`id`
WHERE b.`id`>3;
```

2.查询哪个城市没有部门。

```sql
-- 从要求中可以推断出locations表为主表，departments表为从表，因为有些部门没有和城市匹配。
-- 下面使用右外连接 right右面的为主表
SELECT city
FROM departments d
RIGHT OUTER JOIN locations l
ON d.`location_id`=l.`location_id`
WHERE d.`department_id` IS NULL;
```

3.查询部门名为SAL或IT的员工信息。

```sql
-- 题目要求查询部门名为指定内容的员工信息，考虑到可能会出现有些部门没有员工，为了查到全，所以这里采用外连接，不采用内连接。部门表departments为主表，员工表employees为从表
-- 下面使用左外连接
SELECT e.*,d.department_name
FROM departments d
LEFT OUTER JOIN employees e
ON d.`department_id`=e.`department_id`
WHERE d.`department_name` IN('SAL','IT');
```

子查询

含义:

出现在其他语句中的select语句，称为子查询或内查询。

外部的查询语句，成为主查询或外查询。

分类:

- 按子查询出现的位置分类:
    - select后面
        - 仅仅支持标量子查询
    - from后面
        - 支持表子查询
    - **where或having后面**
        - 支持**标量子查询、列子查询**
        - 也支持行子查询(不过用的较少)
    - exists后面（相关子查询）
        - 支持表子查询
- 按结果集的行列数不同分类:

- 标量子查询（结果集只有一行一列）
- 列子查询（结果集只有一列多行）
- 行子查询（结果集有一行多列，也支持多列多行）
- 表子查询（结果集一般为多行多列，也就是说具有上面三个子查询的特性）

where或having后面

特点：

- 子查询放在小括号内
- 子查询一般放在条件的右侧
- 标量子查询，一般搭配着单行操作符使用。
  - 单行操作符比如：`> < >= <= = <>`
- 列子查询，一般搭配着多行操作符使用
  - 多行操作符比如：`IN、ANY/SOME、ALL`
- 子查询的执行优先于主查询的执行，也就是说主查询的条件用到了子查询的结果。

where或having后面的标量子查询（也称为单行子查询）使用

案例1：谁的工资比Abel高?

```
-- ①：查询Abel的工资
SELECT salary
FROM employees
WHERE last_name='Abel';
-- ②：查询员工的信息，满足salary>①的结果
SELECT *
FROM employees
WHERE salary > (
    SELECT salary
    FROM employees
    WHERE last_name='Abel'
);
```

案例2：返回job_id与141号员工相同，salary比143号员工多的员工的姓名、job_id和工资。

```
-- ①查询141号员工的job_id
SELECT job_id
FROM employees
WHERE employee_id = 141;
-- ②查询143号员工的salary
SELECT salary
FROM employees
WHERE employee_id = 143;
-- ③查询员工的信息，要求job_id=①并且salary>②
SELECT last_name,job_id,salary
FROM employees
WHERE job_id=(
    SELECT job_id
    FROM employees
    WHERE employee_id = 141
) AND salary>(
    SELECT salary
    FROM employees
    WHERE employee_id = 143
```

```
);
```

案例3：返回公司工资最少的员工的last_name、job_id和salary

```
-- ①查询公司的最低工资
SELECT MIN(salary)
FROM employees;
-- ②查询员工的last_name、job_id和salary,要求salary=①
SELECT last_name,job_id,salary
FROM employees
WHERE salary=(
    SELECT MIN(salary)
    FROM employees
);
```

案例4：查询最低工资大于50号部门最低工资的部门id和其最低工资。

```
-- ①:查询50号部门的最低工资
SELECT MIN(salary)
FROM employees
WHERE department_id = 50;
-- ②:查询每个部门的最低工资
SELECT department_id, MIN(salary)
FROM employees
GROUP BY department_id;
-- ③:在②的基础上筛选，满足min(salary)>①
SELECT department_id, MIN(salary)
FROM employees
GROUP BY department_id
HAVING MIN(salary) > (
    SELECT MIN(salary)
    FROM employees
    WHERE department_id = 50
);
```

非法使用标量子查询情况，比如

```
SELECT department_id, MIN(salary)
FROM employees
GROUP BY department_id
HAVING MIN(salary) > (
    SELECT salary
    FROM employees
    WHERE department_id = 50
);


-- 上面这个是非法使用，原因是子查询的结果不是单行单列
```

where或having后面的列子查询（也称为多行子查询）使用

案例1：返回location_id是1400或1700的部门中的所有员工姓名。

```
-- 1.先查询location_id是1400或1700的部门编号
SELECT DISTINCT department_id
```

```
FROM departments
WHERE location_id = 1400
OR location_id = 1700;
-- 2.查询员工姓名，要求部门号是1列表中的某一个
SELECT last_name
FROM employees
WHERE department_id IN(
    SELECT DISTINCT department_id
    FROM departments
    WHERE location_id = 1400
    OR location_id = 1700
);
```

案例2：返回其他工种中比`job_id`为`IT_PROG`工种任意一个工资低的员工的员工号、姓名、`job_id`以及salary。

```
-- 1.查询job_id=IT_PROG部门的任一工资
SELECT DISTINCT salary
FROM employees
WHERE job_id = 'IT_PROG';
-- 2.查询员工号、姓名、job_id以及salary，并且salary<any(1中的任意一个)
SELECT last_name,employee_id,job_id,salary
FROM employees
WHERE salary < ANY(
    SELECT DISTINCT salary
    FROM employees
    WHERE job_id = 'IT_PROG'
) AND job_id<>'IT_PROG';
-- 或者这么写
SELECT last_name,employee_id,job_id,salary
FROM employees
WHERE salary < (
    SELECT DISTINCT MAX(salary)
    FROM employees
    WHERE job_id = 'IT_PROG'
) AND job_id<>'IT_PROG';
```

案例3：返回其他工种中比`job_id`为`IT_PROG`工种所有工资都低的员工的员工号、姓名、`job_id`以及salary。

```
SELECT last_name,employee_id,job_id,salary
FROM employees
WHERE salary < ALL(
    SELECT DISTINCT salary
    FROM employees
    WHERE job_id = 'IT_PROG'
) AND job_id<>'IT_PROG';
-- 或者这么写
SELECT last_name,employee_id,job_id,salary
FROM employees
WHERE salary < (
    SELECT DISTINCT MIN(salary)
    FROM employees
    WHERE job_id = 'IT_PROG'
) AND job_id<>'IT_PROG';
```

where或having后面的行子查询（结果集为一行多列或多列多行)使用

案例：查询员工编号最小并且工资最高的员工信息。

```sql
-- 1.查询最小的员工编号
SELECT MIN(employee_id)
FROM employees;
-- 2.查询最高工资
SELECT MAX(salary)
FROM employees;
-- 3.查询员工信息
SELECT *
FROM employees
WHERE employee_id = (
    SELECT MIN(employee_id)
    FROM employees
)AND salary=(
    SELECT MAX(salary)
    FROM employees
);


-- 用行子查询解决
SELECT *
FROM employees
WHERE (employee_id,salary)=(
    SELECT MIN(employee_id),MAX(salary)
    FROM employees
);
```

select后面

案例1：查询每个部门的员工个数。

```sql
SELECT d.*,(
    SELECT COUNT(*)
    FROM employees e
    WHERE e.department_id  = d.`department_id`
) 个数
FROM departments d;
```

案例2：查询员工号=102的部门名。

```sql
SELECT(
    SELECT department_name
    FROM departments d
    INNER JOIN employees e
    ON d.department_id = e.department_id
    WHERE e.employee_id = 102
) 部门名;
```

from后面

案例：查询每个部门的平均工资的工资等级。

```sql
-- 1.先查询每个部门的平均工资
```

```
SELECT AVG(salary),department_id
FROM employees
GROUP BY department_id;
-- 2.连接1的结果集何job_grades表，筛选条件是平均工资在lowest_sal和highest_sal之间。
SELECT ag_dep.*,g.`grade_level`
FROM (
    SELECT AVG(salary) ag,department_id
    FROM employees
    GROUP BY department_id
)ag_dep
INNER JOIN job_grades g
ON ag_dep.ag BETWEEN lowest_sal AND highest_sal;


-- 发现from将子查询结果充当一张表，要求必须起别名
```

exists后面（相关子查询）

语法：

`exists(完整的查询语句);`

最终结果只有两种情况，1或0。

案例1：查询有员工的部门名。

```
SELECT department_name
FROM departments d
WHERE EXISTS(
    SELECT *
    FROM employees e
    WHERE d.department_id = e.`department_id`
);


 -- 或使用in解决
SELECT department_name
FROM departments d
WHERE d.`department_id` IN(
    SELECT department_id
    FROM employees
);
```

案例2：查询没有女朋友的男神信息。

```
-- 使用in
SELECT bo.*
FROM boys bo
WHERE bo.id NOT IN(
    SELECT boyfriend_id
    FROM beauty
);

-- 使用exists
SELECT bo.*
FROM boys bo
WHERE NOT EXISTS(
```

```
    SELECT boyfriend_id
    FROM beauty b
    WHERE b.`boyfriend_id`=bo.`id`
);
```

练习

案例1：查询和`Zlotkey`相同部门的员工姓名和工资。

```
-- 1.查询Zlotkey的部门
SELECT department_id
FROM employees
WHERE last_name = 'Zlotkey';

-- 2.查询部门号等于1.的结果的姓名和工资
SELECT last_name, salary
FROM employees
WHERE department_id = (
    SELECT department_id
    FROM employees
    WHERE last_name = 'Zlotkey'
);
```

案例2：查询工资比公司平均工资高的员工的员工号，姓名和工资。

```
-- 1.查询公司的平均工资
SELECT AVG(salary)
FROM employees;
-- 2.查询工资比1.的结果高的员工的员工号，姓名和工资
SELECT employee_id,last_name,salary
FROM employees
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
);
```

案例3：查询各部门中工资比本部门平均工资高的员工的员工号，姓名和工资。

```
-- 首先需要读懂题意，问的是各部门中工资大于本部门平均工资的员工的信息，意思就是筛选处每个部门大于自身部门的平均工资的员工

-- 1.查询各部门的平均工资
SELECT AVG(salary),department_id
FROM employees
GROUP BY department_id;
-- 2.连接1.的结果集和employees表，进行筛选
SELECT employee_id,last_name,salary, e.department_id
FROM employees e
INNER JOIN(
    SELECT AVG(salary) ag,department_id
    FROM employees
    GROUP BY department_id
)ag_dep
ON e.`department_id`=ag_dep.department_id
WHERE salary > ag_dep.ag;
```

案例4：查询和姓名中包含字母u的员工在相同部门的员工的员工号和姓名。

```sql
-- 1.查询姓名中包含字母u的员工的部门号
SELECT DISTINCT department_id
FROM employees
WHERE last_name LIKE '%u%';
-- 2.查询部门号=1.的结果集的任意一个员工的员工号和姓名
SELECT last_name,employee_id
FROM employees
WHERE department_id IN(
    SELECT DISTINCT department_id
    FROM employees
    WHERE last_name LIKE '%u%'
);
```

案例5：查询在部门的location_id为1700的部门工作的员工的员工号。

```sql
-- 1.查询location_id为1700的部门
SELECT DISTINCT department_id
FROM departments
WHERE location_id = 1700;
-- 2.查询部门号=1.的结果集中的任意一个的员工号。
SELECT employee_id
FROM employees
WHERE department_id = ANY(
    SELECT DISTINCT department_id
    FROM departments
    WHERE location_id = 1700
);
-- 按照这里的题意，any换成in也行
```

案例6：查询管理者是K_ing的员工姓名和工资。

```sql
-- 1.查询姓名为K_ing的员工编号
SELECT employee_id
FROM employees
WHERE last_name = 'K_ing';
-- 2.查询哪个员工的manager_id = 1.的结果集的任意一个
SELECT last_name,salary
FROM employees
WHERE manager_id IN(
    SELECT employee_id
    FROM employees
    WHERE last_name = 'K_ing'
);
```

案例7：查询工资最高的员工的姓名，要求first_name和last_name显示为一列，列名为姓，名。

```sql
-- 1.查询最高工资
SELECT MAX(salary)
FROM employees;
-- 2.查询工资=1的结果集的姓、名
SELECT CONCAT(first_name,last_name) "姓名"
FROM employees
WHERE salary=(
    SELECT MAX(salary)
    FROM employees
);
```

经典案例

案例1：查询工资最低的员工信息: last_name, salary 。

```sql
-- 一、查询最低工资
SELECT MIN(salary)
FROM employees;
-- 二、查询last_name,salary，要求salary=一、的结果
SELECT last_name,salary
FROM employees
WHERE salary=(
    SELECT MIN(salary)
    FROM employees
);
```

案例2：查询平均工资最低的部门信息 。

```sql
-- 一、查询各部门的平均工资
SELECT AVG(salary),department_id
FROM employees
GROUP BY department_id;
-- 二、查询一、结果中的最低平均工资
SELECT MIN(ag)
FROM (
    SELECT AVG(salary) ag,department_id
    FROM employees
    GROUP BY department_id
) ag_dep;
-- 三、查询哪个部门的平均工资=二、的结果
SELECT AVG(salary),department_id
FROM employees
GROUP BY department_id
HAVING AVG(salary)=(
    SELECT MIN(ag)
    FROM (
        SELECT AVG(salary) ag,department_id
        FROM employees
        GROUP BY department_id
    ) ag_dep
);
-- 四、查询部门信息
SELECT d.*
FROM departments d
WHERE d.`department_id`=(
    SELECT department_id
```

```
    FROM employees
    GROUP BY department_id
    HAVING AVG(salary)=(
        SELECT MIN(ag)
        FROM (
            SELECT AVG(salary) ag,department_id
            FROM employees
            GROUP BY department_id
        ) ag_dep
    )
);
```

或者

```
-- 一、求出最低平均工资的部门编号
SELECT department_id
FROM employees
GROUP BY department_id
order by avg(salary)
limit 1;
-- 二、查询部门信息
SELECT *
FROM departments
WHERE department_id=(
    SELECT department_id
    FROM employees
    GROUP BY department_id
    ORDER BY AVG(salary)
    LIMIT 1
);
```

案例3：查询平均工资最低的部门信息和该部门的平均工资 。

```
-- 一、查询各部门的平均工资
SELECT AVG(salary),department_id
FROM employees
GROUP BY department_id;
-- 二、求出最低平均工资的部门编号
SELECT AVG(salary),department_id
FROM employees
GROUP BY department_id
order by avg(salary)
limit 1;
-- 三、查询部门信息(这里用到了表子查询)
SELECT d.*
FROM departments d
JOIN (
    SELECT AVG(salary),department_id
    FROM employees
    GROUP BY department_id
    ORDER BY AVG(salary)
    LIMIT 1
)ag_dep
ON d.`department_id`=ag_dep.department_id;
```

案例4：查询平均工资最高的 job 信息 。

```sql
-- 一、查询每个job的平均工资
SELECT AVG(salary),job_id
FROM employees
GROUP BY job_id
ORDER BY AVG(salary) DESC
LIMIT 1;
-- 二、查询job信息
SELECT *
FROM jobs
WHERE jobs.`job_id` = (
    SELECT job_id
    FROM employees
    GROUP BY job_id
    ORDER BY AVG(salary) DESC
    LIMIT 1
);
```

案例5：查询平均工资高于公司平均工资的部门有哪些?

```sql
-- 一、查询公司的平均工资
SELECT AVG(salary)
FROM employees;
-- 二、查询每个部门的平均工资
SELECT AVG(salary)
FROM employees
GROUP BY department_id;
-- 三、筛选二、的结果集，满足平均工资>一的结果

```

案例6：查询出公司中所有 manager 的详细信息。

```sql
-- 一、查询所有manager的员工编号
SELECT DISTINCT manager_id
FROM employees;
-- 二、查询详细信息，满足employee_id=1的结果
SELECT *
FROM employees
WHERE employee_id = ANY(
    SELECT DISTINCT manager_id
    FROM employees
);
```

案例7：各个部门中 最高工资中最低的那个部门的 最低工资是多少 。

```sql
-- 一、查询各部门的最高工资
SELECT MAX(salary)
FROM employees
GROUP BY department_id;
-- 二、找一、结果集中最低的那个
SELECT MAX(salary)
FROM employees
GROUP BY department_id
ORDER BY MAX(salary)
LIMIT 1;
-- 三、查询哪个部门的最高工资等于二、结果，

```

```sql
SELECT MIN(salary),department_id
FROM employees
WHERE department_id = (
    SELECT department_id
    FROM employees
    GROUP BY department_id
    ORDER BY MAX(salary)
    LIMIT 1
);
```

案例8：查询平均工资最高的部门的 manager 的详细信息: last_name, department_id, email, salary 。

```sql
-- 一、找到平均工资最高的部门编号
SELECT department_id
FROM employees
GROUP BY department_id
ORDER BY AVG(salary) DESC
LIMIT 1;
-- 二、将employees和departments连接查询，筛选条件是一、的结果
SELECT last_name,d.department_id,email,salary
FROM employees e
INNER JOIN departments d
ON d.`manager_id` = e.`employee_id`
WHERE d.`department_id`=(
    SELECT department_id
    FROM employees
    GROUP BY department_id
    ORDER BY AVG(salary) DESC
    LIMIT 1
);
```

分页查询

应用场景：当要显示的数据，一页显示不全时，需要分页提交sql请求。

语法：

```sql
select 查询列表
from 表
【join type join 表2
on 连接条件
where 筛选条件
group by 分组字段
having 分组后的筛选
order by 排序的字段】
limit offset,size;

-- offset表示要显示条目的起始索引（起始索引从0开始）
-- size表示要显示的条目个数
```

```
--执行顺序：from表先走，再inner join on,再去筛选，再去group by，再去having，再走select，然后order by，最后执行limit
```

特点：

- limit语句放在查询语句的最后；
- 在web开发的分页显示中会用到下面的公式：
    - 要显示的页数page，每页的条目数size
    - ```
      select 查询列表
      from 表
      limit (page-1)*size,size;
      ```

案例1：查询前5条员工信息。

```
SELECT * FROM employees LIMIT 0,5;
```

案例2：查询第11条到第25条员工信息。

```
SELECT * FROM employees LIMIT 10,15;
```

案例3：查询有奖金的员工信息，并且工资较高的前10名显示出来。

```
SELECT * FROM employees WHERE commission_pct IS NOT NULL ORDER BY salary DESC LIMIT 10;
```

union联合查询

union联合，合并：将多条查询语句的结果合成一个结果。

语法：

```
查询语句1
union
查询语句2
union
...
```

引入的案例：查询部门编号>90或邮箱中包含a的员工信息。

```
SELECT * FROM employees WHERE email LIKE '%a%'
UNION
SELECT * FROM employees WHERE department_id>90;
```

应用场景：

要查询的结果来自于多个表，且多个表没有直接的连接关系，但查询的信息一致（表示列字段意义差不多）时。

特点：

- 要求多条查询语句的查询列数是一致的。
- 要求多条查询语句的查询的每一列的类型和顺序最好一致。
- union关键字默认去重，如果使用union all，就可以包含重复项。
```

十、DML语言的学习

插入语句

语法:

```
-- 方式一、
insert into 表名(字段名，...)
values(值1，...);

-- 方式二、
insert into 表名
set 列名=值,列名=值,...

-- 方式一支持插入多行数据，方式二不支持
-- 方式一支持子查询，方式二不支持
```

特点:

- 字段类型和值类型一致或兼容，而且一一对应;
- 可以为空的字段，可以不用插入值，或用null填充;
- 不可以为空的字段，必须插入值;
- 字段个数和值的个数必须一致;
- 字段可以省略，但默认所有字段，并且顺序和表中的存储顺序一致;

修改语句

语法:

```
-- 修改单表语法:
update 表名 set 字段=新值,字段=新值
【where 条件】

-- 修改多表语法【补充】:
-- sql92语法
update 表1 别名1,表2 别名2
set 字段=新值，字段=新值
where 连接条件
and 筛选条件
--sql99语法
update 表1 别名
inner|left|right join 表2 别名
on 连接条件
set 列=值,...
where 筛选条件
```

修改多表记录案例:

案例1：修改张无忌的女朋友的手机号为114。

```sql
UPDATE boys bo
INNER JOIN beauty b
ON bo.`id`=b.`boyfriend_id`
SET b.`phone`='114'
WHERE bo.`boyName`='张无忌';
```

案例2：修改没有男朋友的女神的男朋友编号为2号。

```sql
UPDATE boys bo
RIGHT JOIN beauty b
ON bo.`id`=b.`boyfriend_id`
SET b.`boyfriend_id`=2
WHERE bo.`id` IS NULL;
```

删除语句

语法：

```sql
--方式一：delete语句
-- 单表删除
delete from 表名 【where 筛选条件】
-- 不加where的时候会清空该表

-- 多表删除【补充】
--sql92语法
delete 别名1，别名2
from 表1 别名1，表2 别名2
where 连接条件
and 筛选条件;
--sql99语法
delete 表1的别名,表2的别名
from 表1 别名
inner|left|right join 表2 别名
where 筛选条件;

--方式二：truncate语句（不允许加where）
truncate table 表名
--执行它后会清空该表
```

两种方式的区别：

- truncate不能加where条件，而delete可以加where条件
- truncate的效率高一丢丢
- truncate 删除带自增长的列的表后，如果再插入数据，数据从1开始；delete 删除带自增长列的表后，如果再插入数据，数据从上一次的断点处开始
- truncate删除不能回滚，delete删除可以回滚

多表的删除案例：

- 案例：删除张无忌的女朋友信息。
    - ```sql
      DELETE b
      FROM beauty b
      INNER JOIN boys bo ON b.`boyfriend_id`=bo.`id`
      WHERE bo.`boyName`='张无忌';
      ```
      ```

- 案例：删除黄晓明的信息以及他女朋友的信息。
    - ```
      DELETE b,bo
      FROM beauty b
      INNER JOIN boys bo ON b.`boyfriend_id`=bo.`id`
      WHERE bo.`boyName`='黄晓明';
      ```

十一、DDL语言的学习

也称为数据定义语言，包括库和表的管理。

- 库的管理
    - 创建
    - 修改
    - 删除
- 表的管理
    - 创建
    - 修改
    - 删除

创建：create；

修改：alter；

删除：drop。

库的管理

1.库的创建

```
create database [if not exists]库名;
```

案例：创建库Books。

```
-- 加IF NOT EXISTS是为了避免当存在那个库时再创建会报错
CREATE DATABASE IF NOT EXISTS books;
```

2.库的修改

更改库的字符集

```
ALTER DATABASE books CHARACTER SET utf8;
```

3.库的删除

```
-- 加if exists是为了避免删除不存在的库报错
drop database [if exists] 库名
```

案例：删除库Books。
```

```sql
DROP DATABASE IF EXISTS books;
```

表的管理

1.表的创建

语法:

```sql
create table 表名(
    列名 列的类型【(长度)约束】,
    列名 列的类型【(长度)约束】,
    列名 列的类型【(长度)约束】,
    ...
    列名 列的类型【(长度)约束】
);
```

案例：创建表Book。

```sql
-- 加IF NOT EXISTS是为了增强容错性
CREATE TABLE IF NOT EXISTS book(
    id INT,#编号
    bName VARCHAR(20),#图书名
    price DOUBLE,#价格
    authorId INT,#作者
    publishDate DATETIME #出版日期
);
```

案例：创建表author。

```sql
CREATE TABLE author(
    id INT,#编号
    au_name VARCHAR(20),
    nation VARCHAR(10) #国籍
);
```

2.表的修改

语法:

```sql
alter table 表名 add|drop|modify|change column 字段名【字段类型 约束】;
```

作用范围:

- 可以修改列名
    - 将表book的publishdate字段修改成pubDate，注意后面还要加上类型。
    - ```sql
      ALTER TABLE book CHANGE COLUMN publishdate pubDate DATETIME;
      ```
- 修改列的类型或约束
    - ```sql
      -- 注意它不区分大小写
      ALTER TABLE book MODIFY COLUMN pubdate TIMESTAMP;
      ```

- 添加新列
    - ```
      -- 注意类型也要写上
      ALTER TABLE author ADD COLUMN annual DOUBLE;
      ```
- 删除列
    - ```
      ALTER TABLE author DROP COLUMN annual;
      ```
- 修改表名
    - ```
      ALTER TABLE author RENAME TO book_author;
      ```

3.表的删除

```
-- 加IF EXISTS是为了容错性
DROP TABLE IF EXISTS book_author;
```

表的复制

1.仅仅复制表的结构

```
-- 创建一个copy表，复制author表的结构
CREATE TABLE copy LIKE author;
```

2.复制表的结构+数据

```
CREATE TABLE copy2
SELECT * FROM author;
```

只复制部分数据:

```
CREATE TABLE copy3
SELECT id,au_name
FROM author
WHERE nation='中国';
```

仅仅复制某些字段(不复制数据):

```
CREATE TABLE copy4
SELECT id,au_name
FROM author
WHERE 1=2;
-- 或
CREATE TABLE copy4
SELECT id,au_name
FROM author
WHERE 0;
```

常见的数据类型
```

- 数值型
    - 整型
    - 小数
        - 定点数
        - 浮点数
- 字符型
    - 较短的文本：char、varchar
    - 较长的文本：text、blob(较长的二进制数据)
- 日期型

1.整型

- **整型**

整数类型	字节	范围
Tinyint	1	有符号：-128~127 无符号：0~255
Smallint	2	有符号：-32768~32767 无符号：0~65535
Mediumint	3	有符号：-8388608~8388607 无符号：0~1677215 (好吧，反正很大，不用记住)
Int、integer	4	有符号：- 2147483648~2147483647 无符号：0~4294967295 (好吧，反正很大，不用记住)
Bigint	8	有符号： -9223372036854775808 ~9223372036854775807 无符号：0~ 92233720368547758O7*2+1 (好吧，反正很大，不用记住)

特点：

- 如果不设置无符号还是有符号，默认时有符号，如果想设置无符号，则需要添加unsigned 关键字。
- 如果插入的数值超出了整型的范围，会报out of range异常，并且插入临界值。
- 如果不设置长度，会有默认的长度。
- 长度代表了显示的最大宽度，如果不够会用0在左边填充，但在创建表时需要搭配zerofill使用。如：
    - ```
      CREATE TABLE tab_int(
          t1 INT(7) ZEROFILL,
          t2 INT(7) ZEROFILL
      );
      -- 但这样就不能插入负数了
      ```

案例：如何设置无符号和有符号。
      ```

```sql
CREATE TABLE tab_int(
    t1 INT,
    t2 INT UNSIGNED
);
```

2.小数

- 小数

浮点数类型	字节	范围
float	4	±1.75494351E-38~±3.402823466E+38 (好吧，反正很大，不用记住)
double	8	±2.2250738585072014E-308~ ±1.7976931348623157E+308 (好吧，反正很大，不用记住)
定点数类型	字节	范围
DEC(M,D) DECIMAL(M,D)	M+2	最大取值范围与double相同，给定decimal的有效取值范围由M和D决定

浮点型:

`float(M,D)`

`double(M,D)`

定点型:

`dec(M,D)`

`decimal(M,D)`

特点:

- M代表整数部位+小数部位的长度
- D代表小数部位的长度
- 如果超出范围，则插入临界值
- 如果没有特殊要求的话，M和D可以省略
- 如果是decimal，则M默认为0，D默认为0
- 如果是float和double，则会根据插入的数值的精度来决定精度.
- 定点型的精确度较高，如果要求插入数值的精度较高比如货币运算，则用定点型的。

原则:

所选择的类型越简单越好，能保存数值的类型越小越好。

3.字符型

- 较短的文本：char、varchar

- 较长的文本：text、blob(较长的二进制数据)
- 其他：
    - binary和varbinary用于保存较短的二进制
    - enum用于保存枚举
    - set用于保存集合

char和varchar类型

说明：用来保存MySQL中较短的字符串。

字符串类型	最多字符数	描述及存储需求
char(M)	M	M为0~255之间的整数
varchar(M)	M	M为0~65535之间的整数

char代表固定长度的字符，varchar代表可变长度的字符。可以和char、string类比。

4.日期型

日期和时间类型	字节	最小值	最大值
date	4	1000-01-01	9999-12-31
datetime	8	1000-01-01 00:00:00	9999-12-31 23:59:59
timestamp	4	19700101080001	2038年的某个时刻
time	3	-838:59:59	838:59:59
year	1	1901	2155

常见约束

约束的含义：

一种限制，用于限制表中的数据，为了保证表中的数据的准确和可靠性。

分类：

- NOT NULL：非空，用于保证该字段的值不能为空，比如姓名、学号等；
- DEFAULT：默认，用于保证该字段有默认值，比如性别；
- PRIMARY KEY：主键，用于保证该字段的值具有唯一性，并且非空，比如学号、员工编号等；
- UNIQUE：唯一，用于保证该字段的值具有唯一性，可以为空，比如座位号；

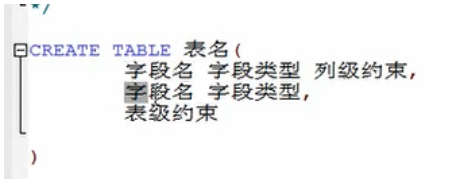
- CHECK：检查约束【mysql中不支持】
- FOREIGN KEY：外键，用于限制两个表的关系，用于保证该字段的值必须来自于主表的关联列的值。
    - 注意是在从表中添加外键约束，用于引用主表中某列的值。比如学生表的专业编号，员工表的部门编号，员工表的工种编号。
    - 从表的外键列的类型要求和主表中对应的列的类型一致。名称无要求。
    - 主表的关联列必须是一个key(一般是主键、唯一键)
    - 插入数据时，先插入主表，再插入从表的数据。删除数据时先删除从表，再删除主表。

添加约束的时机：

- 创建表时
- 修改表时

约束的添加分类：

- 列级约束
    - 六大约束语法上都支持，但外键约束没有效果；
- 表级约束
    - 除了非空(NOT NULL)、默认(DEFAULT)，其他的都支持；



创建表时添加约束

1.添加列级约束

语法：

- 直接在字段名和类型后面追加约束类型即可。
- 只支持默认、非空、主键、唯一。

```
CREATE TABLE major(
    id INT PRIMARY KEY,
    majorName VARCHAR(20)
);

--尽管这里写了外键约束，但由于在列级约束中外键约束没有效果，故这里没有效果。
--这个check在mysql不支持
CREATE TABLE stuinfo(
    id INT PRIMARY KEY,#主键
    stuName VARCHAR(20) NOT NULL,#非空约束
    gender CHAR(1) CHECK(gender='男' OR gender='女'),#检查约束
    seat INT UNIQUE,#唯一约束
    age INT DEFAULT 18,#默认约束
    majorId INT REFERENCES major(id)#外键约束
);

-- 查看stuinfo表中所有的索引，包括主键、外键、唯一
SHOW INDEX FROM stuinfo;
```

2.添加表级约束

语法：

在各个字段的最下面

`【constraint 约束名】 约束类型(字段名)`

```
DROP TABLE IF EXISTS stuinfo;
CREATE TABLE stuinfo(
    id INT,
    stuname VARCHAR(20),
    gender CHAR(1),
    seat INT,
    age INT,
    majorid INT,
    CONSTRAINT pk PRIMARY KEY(id),#主键
    CONSTRAINT uq UNIQUE(seat),#唯一键
    CONSTRAINT ck CHECK(genger = '男' OR gender = '女'),#检查约束
    CONSTRAINT fk_stuinfo_major FOREIGN KEY(majorid) REFERENCES major(id)#外键

);

-- 或者这样写

DROP TABLE IF EXISTS stuinfo;
CREATE TABLE stuinfo(
    id INT,
    stuname VARCHAR(20),
    gender CHAR(1),
    seat INT,
    age INT,
    majorid INT,
    PRIMARY KEY(id),#主键
    UNIQUE(seat),#唯一键
    CHECK(genger = '男' OR gender = '女'),#检查约束
    FOREIGN KEY(majorid) REFERENCES major(id)#外键
);
```

3.通用的写法：

也就是说可以用列级约束的那样子写的就用列级约束写，否则就用表级约束去写。

```
CREATE TABLE IF NOT EXISTS stuinfo(
    id INT PRIMARY KEY,
    studname VARCHAR(20) NOT NULL,
    sex CHAR(1),
    age INT DEFAULT 18,
    seat INT UNIQUE,
    majorid INT,
    CONSTRAINT fk_stuinfo_major FOREIGN KEY(majorid) REFERENCES major(id)
);
```

修改表时添加约束

语法

```
-- 1. 添加列级约束
alter table 表名 modify column 字段名 字段类型 新约束
-- 2. 添加表级约束
alter table 表名 add 【constraint 约束名】 约束类型(字段名) 【外键的引用】
```

1.添加非空约束

```
-- 这里为了举例，先重新创建一个表
DROP TABLE IF EXISTS stuinfo;
CREATE TABLE stuinfo(
    id INT,
    stuname VARCHAR(20),
    gender CHAR(1),
    seat INT,
    age INT,
    majorid INT
);

-- 将stuname字段添加非空约束
ALTER TABLE stuinfo MODIFY COLUMN stuname VARCHAR(20) NOT NULL;
```

2.添加默认约束

```
-- 将age字段添加默认约束
ALTER TABLE stuinfo MODIFY COLUMN age INT DEFAULT 18;
```

3.添加主键

```
-- 将id字段添加主键约束
ALTER TABLE stuinfo MODIFY COLUMN id INT PRIMARY KEY;
--或者这么写
ALTER TABLE stuinfo ADD PRIMARY KEY(id);
```

4.添加唯一

```
-- 将seat字段添加唯一
ALTER TABLE stuinfo MODIFY COLUMN seat INT UNIQUE;
-- 或者这么写
ALTER TABLE stuinfo ADD UNIQUE(seat);
```

5.添加外键

```
-- 为majorid字段添加外键约束，与major表中的id字段关联
ALTER TABLE stuinfo ADD CONSTRAINT fk_stuinfo_major FOREIGN KEY(majorid)
REFERENCES major(id);
```

修改表时删除约束

1.删除非空约束

```
ALTER TABLE stuinfo MODIFY COLUMN stuname VARCHAR(20) NULL;
```

2.删除默认约束

```
ALTER TABLE stuinfo MODIFY COLUMN age INT;
```

3.删除主键

```
ALTER TABLE stuinfo DROP PRIMARY KEY;
```

4.删除唯一

```
ALTER TABLE stuinfo DROP INDEX seat;
-- index后面要写什么需要先执行show index from stuinfo 查看对应的Key_name是什么,填到index后面
```

5.删除外键

```
ALTER TABLE stuinfo DROP FOREIGN KEY fk_stuinfo_major;
-- foreign key 后面要写什么需要先执行show index from stuinfo 查看对应的Key_name是什么,填到key后面
```

标识列

又称为自增长列，可以不用手动的添加值，系统提供默认的序列值。

特点

- 标识列必须和主键搭配吗？------> 不一定，但要求是一个key，比如主键、唯一、外键;
- 一个表中可以有几个标识列？-------> 至多一个;
- 标识列的类型必须是数值类型;
- 标识列可以通过`SET auto_increment_increment=3;`设置步长。也可以通过手动插入值来设置起始值。

创建表时设置标识列

```
DROP TABLE IF EXISTS tab_identity;
CREATE TABLE tab_identity(
    id INT PRIMARY KEY AUTO_INCREMENT,      -- 看AUTO_INCREMENT
    NAME VARCHAR(20)
);

INSERT INTO tab_identity VALUES(NULL,'join');
INSERT INTO tab_identity(NAME) VALUES('tom');

SELECT * FROM tab_identity;
```

修改表时设置标识列

```sql
DROP TABLE IF EXISTS tab_identity;
CREATE TABLE tab_identity(
    id INT,
    NAME VARCHAR(20)
);

ALTER TABLE tab_identity MODIFY COLUMN id INT PRIMARY KEY AUTO_INCREMENT;
```

修改表时删除标识列

```sql
ALTER TABLE tab_identity MODIFY COLUMN id INT;
```

十二、TCL语言的学习

相关概念介绍

Transaction Control Language：事务控制语言。

事务：

一个或一组sql语句组成一个执行单元，这个执行单元要么全部执行，要么全部不执行。

通过一组逻辑操作单元（一组DML——sql语句），将数据从一种状态切换到另外一种状态

案例：转账。

```
初始时
张三丰 1000
郭襄   1000
update 表 set 张三丰的余额=500 where name='张三丰'
update 表 set 郭襄的余额=1500 where name='郭襄'
```

事务的ACID(acid)属性：

- 原子性：要么都执行，要么都回滚；
- 一致性：保证数据的状态操作前和操作后保持一致；
- 隔离性：多个事务同时操作相同数据库的同一个数据时，一个事务的执行不受另外一个事务的干扰；
- 持久性：一个事务一旦提交，则数据将持久化到本地，除非其他事务对其进行修改；

事务的分类：

- 隐式事务：没有明显的开启和结束事务的标志。
    - 比如insert、update、delete语句本身就是一个事务
- 显式事务：具有明显的开启和结束事务的标志。
    - 前提：必须先设置自动提交功能(`autocommit`)为禁用。`SET autocommit=0;`不过只针对当前的事务有效。可以先执行`SHOW VARIABLES LIKE '%autocommit%';`查看一下是否已经被禁用。
    - 步骤一：开启事务
        - ```sql
          SET autocommit=0;
          START TRANSACTION;  #可选
          ```
          ```

- 步骤二：编写事务的一组逻辑操作单元（多条sql语句[比如select、insert、update、delete]）
    - ```
      语句1;
      语句2;
      ...
      ```
- 步骤三：结束事务
    - ```
      commit; #提交事务
      rollback;#回滚事务
      savepoint 节点名;  #设置保存点，联想一下断点
      ```

演示事务的使用步骤

- 准备表并插入数据：
    - ```
      DROP TABLE IF EXISTS account;

      CREATE TABLE account(
      id INT PRIMARY KEY AUTO_INCREMENT,
      username VARCHAR(20),
      balance DOUBLE
      );

      INSERT INTO account(username,balance)
      VALUES ('张无忌',1000),('郭襄',1000);
      ```
- 开启事务
    - ```
      SET autocommit = 0;
      START TRANSACTION;  #可选
      ```
- 编写一组事务的语句
    - ```
      UPDATE account SET balance = 500 WHERE username='张无忌';
      UPDATE account SET balance = 1500 WHERE username='郭襄';
      ```
- 结束事务
    - ```
      COMMIT;
      --如果写rollback放在commit之前，或者不写commit，会导致数据不会发生改变
      ```

delete 和 truncate 在事务使用时的区别：

- 演示delete(支持回滚)
    - ```
      SET autocommit = 0;
      START TRANSACTION;
      DELETE FROM account;
      ROLLBACK;
      -- 回滚后发现没有删除
      ```
- 演示truncate(不支持回滚)
      ```

```
SET autocommit = 0;
START TRANSACTION;
TRUNCATE TABLE account;
ROLLBACK;
```

事务的隔离级别

事务并发问题如何发生?

当多个事务同时操作同一个数据库的相同数据时

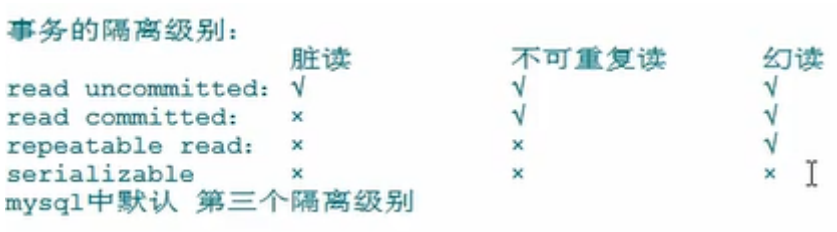
事务的并发问题有哪些?

- 脏读：一个事务读取了其他事务还没有提交的数据，读到的是其他事务“更新”的数据;
- 不可重复读：同一个事务中，多次读取到的数据不一致;
- 幻读：一个事务读取了其他事务还没有提交的数据，只是读到的是 其他事务“插入”的数据。

如何避免事务的并发问题?

通过设置事务的隔离级别来避免，隔离级别有

- `READ UNCOMMITTED`(读未提交数据)允许事务读取未被其他事务提交的变更，脏读、不可重复读和幻读的问题都会出现。
- `READ COMMITTED`(读已提交数据) 只允许事务读取已经被其他事务提交的变更，可以避免脏读，但不可重复读和幻读仍然可能出现。
- `REPEATABLE READ`(可重复读) 确保事务可以多次从一个字段中读取相同的值，在这个事务持续期间，禁止其他事务对这个字段进行更新，可以避免脏读、不可重复读和一部分幻读，但幻读的问题依旧存在。
- `SERIALIZABLE`(串行化) 确保事务可以从一个表中读取相同的行，在这个事务持续期间，禁止其他事务对该表执行插入、更新和删除操作，所有并发问题都可以避免，但性能十分低， 可以避免脏读、不可重复读和幻读。



演示事务的隔离级别

```
-- 用cmd命令窗口(管理员模式)操作

-- 查看当前的隔离级别
select @@tx_isolation;

-- 设置隔离级别
set session transaction isolation level 隔离级别名称;

-- 操作数据库
use 数据库名;

-- 开启事务
SET autocommit = 0;

-- 编写sql语句
```

```
16
17  -- 结束事务
18  -- rollback; 写rollback或commit，根据具体情况决定
19  commit;
```

演示`savepoint`的使用

```
1  SET autocommit = 0;
2  START TRANSACTION;       #在客户端中一般使用这条，在控制台里不需要
3
4  DELETE FROM account WHERE id = 1;
5  SAVEPOINT a; #设置保存点
6  DELETE FROM account WHERE id = 3;
7  ROLLBACK TO a;  #回滚到保存点,这样就导致id=1的数据删除，id=3的数据没有删除
8
9  SELECT * FROM account;
```

十三、视图

含义：

虚拟表，和普通表一样使用。是mysql5.1版本中出现的新特性，是通过表动态生成的数据。它只保存sql逻辑，不保存查询结果。

案例：查询姓张的学生名和专业名。

```
1  -- 以前是这样写
2  SELECT stuname,majorName
3  FROM stuinfo s
4  INNER JOIN major m
5  ON s.`majorid`=m.`id`
6  WHERE s.`stuname` LIKE '张%';
7
8  -- 学了视图后可以将主要部分封装起来
9  CREATE VIEW v1
10 AS
11 SELECT stuname,majorName
12 FROM stuinfo s
13 INNER JOIN major m
14 ON s.`majorid`=m.`id`;
15 --再去使用它
16 SELECT * FROM v1 WHERE stuname LIKE '张%';
```

视图和表的区别

	创建语法的关键字	占用物理空间	使用
视图	`create view`	不占用，仅仅保存的是sql逻辑	增删改查，一般不能增删改
表	`create table`	占用	增删改查

视图的创建

语法：

```
CREATE VIEW  视图名
AS
查询语句;
```

案例1：查询姓名中包含a字符的员工名、部门名和工种信息。

```
-- 创建视图
CREATE VIEW myv1
AS
SELECT last_name,department_name,job_title
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_id
INNER JOIN jobs j ON e.job_id = j.job_id;
-- 使用
SELECT * FROM myv1 WHERE last_name LIKE '%a%';
```

案例2：查询各部门的平均工资级别。

```
# 创建视图，查看每个部门的平均工资
CREATE VIEW myv2
AS
SELECT AVG(salary) ag, department_id
FROM employees
GROUP BY department_id;
# 使用
SELECT myv2.`ag`, g.grade_level
FROM myv2
JOIN job_grades g
ON myv2.`ag` BETWEEN g.`lowest_sal` AND g.`highest_sal`;
```

案例3：查询平均工资最低的部门信息。

```
#前面已经创建过视图myv2，这里就不需要再创建

# 使用
SELECT * FROM myv2 ORDER BY ag LIMIT 1;
```

案例4：查询平均工资最低的部门名和工资。

```
# 创建视图myv3,将包含视图myv2的sql操作再次封装
CREATE VIEW myv3
AS
SELECT * FROM myv2 ORDER BY ag LIMIT 1;
# 使用
SELECT d.*, m.ag
FROM myv3 m
JOIN departments d
ON m.`department_id`=d.`department_id`;
```

视图的修改

语法：

```
-- 方式一
creat or replace view 视图名
as
查询语句;

-- 方式二
alter view 视图名
as
查询语句;
```

视图的删除

语法：

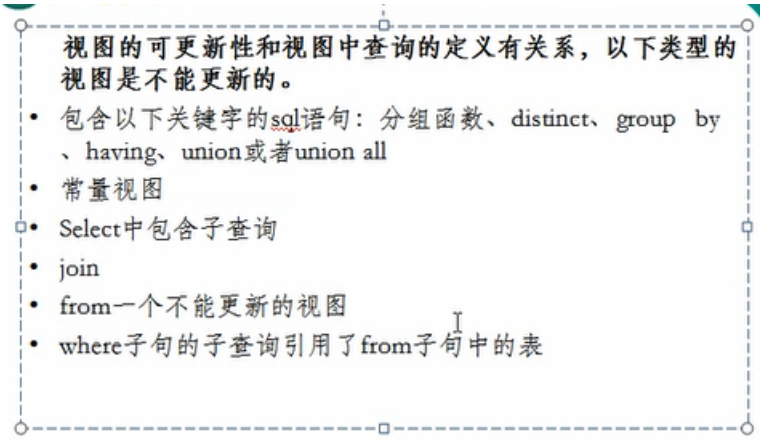
```
drop view 视图名,视图名,...;
```

视图的查看

```
-- 这里说的查看是指查看它的结构
desc 视图名;
```

视图的更新

注意这里的更新是指更改视图中的数据。



其余的就可以，语法和普通表的一样。更新包括插入、删除、修改。

案例

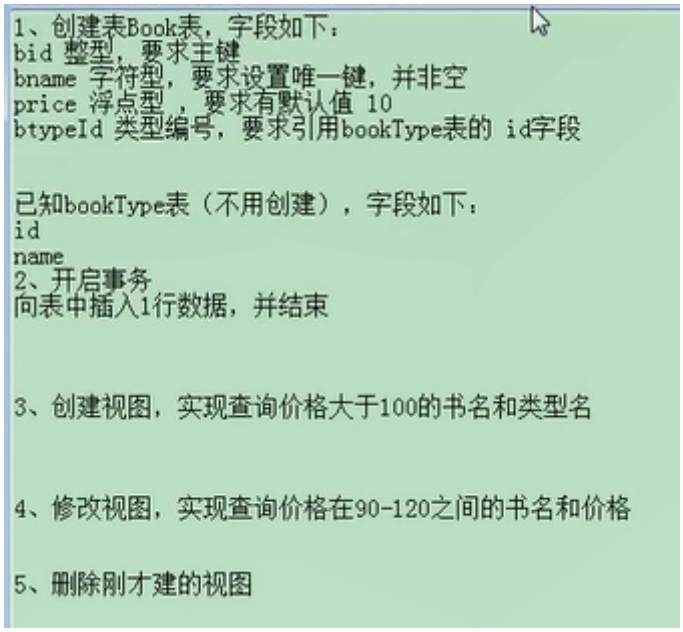
案例1：创建视图`emp_v1`，要求查询电话号码以`011`开头的员工姓名和工资、邮箱。

```
CREATE OR REPLACE VIEW emp_v1
AS
SELECT last_name,salary,email
FROM employees e
WHERE phone_number LIKE '011%';

SELECT * FROM emp_v1;
```

案例2：创建视图`emp_v2`，要求查询部门的最高工资高于12000的部门信息。

```sql
CREATE OR REPLACE VIEW emp_v2
AS
SELECT MAX(salary) mx, department_id
FROM employees
GROUP BY department_id
HAVING mx > 12000;

SELECT d.*, e.mx
FROM emp_v2 e
INNER JOIN departments d
ON d.`department_id` = e.department_id;
```

小测试



1.

```sql
CREATE TABLE bookType(
    id INT PRIMARY KEY,
    NAME VARCHAR(20)
);

CREATE TABLE Book(
    bid INT PRIMARY KEY,
    bname VARCHAR(32) UNIQUE NOT NULL,
    price DOUBLE DEFAULT 10,
    bypeId INT,
    CONSTRAINT fk_book_booktype FOREIGN KEY(bypeId) REFERENCES bookType(id)
);
```

2.

```
SET autocommit = 0;
START TRANSACTION;
#先插入主表中数据
INSERT INTO bookType VALUES
(2333, '水浒');
#再插入从表中数据
INSERT INTO Book VALUES
(1, '张飞', 100000, 2333);
COMMIT;
```

3.

```
CREATE VIEW myv1
AS
SELECT bname, NAME
FROM book bo
INNER JOIN booktype b
ON bo.bypeId=b.id
WHERE price > 100;

SELECT * FROM myv1;
```

4.

```
ALTER VIEW myv1
AS
SELECT bname, price
FROM book
WHERE price BETWEEN 90 AND 120;

SELECT * FROM myv1;
```

5.

```
DROP VIEW myv1;
```

十四、变量

分类

- 系统变量（按照作用范围划分）
    - 全局变量
        - 作用域：服务器每次启动将为所有的全局变量赋初始值。针对于所有的会话(连接)有效，但是不能跨重启，就是重启前的设置，在重启后就恢复默认。
    - 会话变量
        - 作用域：仅仅针对于当前会话(连接)有效。
- 自定义变量（按照作用范围划分）
    - 用户变量
        - 作用域：针对于当前会话(连接)有效，同于会话变量的作用域。应用在任何地方，也就是begin end里面或者begin end外面。

- 局部变量
  - 作用域：仅仅在定义它的begin end中有效。应用在begin end中的第一句话。

系统变量

说明：

变量由系统提供，不是用户定义，属于服务器层面。

使用的语法：

- 1.查看所有的系统变量
  ```
  -- 查看全局变量
  SHOW GLOBAL VARIABLES;
  -- 查看会话变量(session可以不写,默认是session)
  SHOW SESSION VARIABLES;
  ```
- 2.查看满足条件的部门系统变量
  ```
  SHOW GLOBAL VARIABLES LIKE '%char%';
  ```
- 3.查看指定的某个系统变量的值
  ```
  -- 如果这样写，默认是从会话变量中查询
  SELECT @@系统变量名;
  -- 如果想从全局变量中找
  SELECT @@global.系统变量名;
  ```
- 4.为某个系统变量复制
  ```
  set 系统变量名 = 值;
  set global 系统变量名 = 值;

  set @@global.系统变量名=值;
  set @@session.系统变量名=值;
  ```
- 注意：如果是全局级别，则需要加global，如果是会话级别，则可以加，也可以不加session。不写默认是session。

全局变量的演示

1.查看所有的全局变量：

```
SHOW GLOBAL VARIABLES;
```

2.查看部分的全局变量：

```
-- 查看包含char的全局变量
SHOW GLOBAL VARIABLES LIKE '%char%';
```

3.查看指定的全局变量的值：

```
-- 查看autocommit的值
SELECT @@global.autocommit;
-- 查看隔离级别
SELECT @@global.tx_isolation;
```

4.为某个指定的全局变量赋值

```
SET @@global.autocommit=0;
```

会话变量的演示

1.查看所有的会话变量

```
SHOW SESSION VARIABLES;
-- 或者
SHOW VARIABLES;
```

2.查看部分的会话变量

```
SHOW SESSION VARIABLES LIKE '%char%';
-- 或者
SHOW VARIABLES LIKE '%char%';
```

3.查看指定的会话变量的值

```
SELECT @@tx_isolation;
-- 或者
SELECT @@session.tx_isolation;
```

4.为某个会话变量赋值

```
SET @@session.tx_isolation = 'read-uncommitted';
-- 或者
SET SESSION tx_isolation = 'read-committed';
```

自定义变量

说明：

变量是由用户自定义的，不是由系统提供的。

使用步骤：

```
声明
赋值
使用(查看、比较、运算等)
```

用户变量的演示

1.声明并初始化

```sql
-- 下面三种方式都可以
SET @用户变量名=值;
SET @用户变量名:=值;
SELECT @用户变量名:=值;
```

2.赋值(更新用户变量的值)

```sql
-- 方式一：通过set或select
SET @用户变量名=值;
SET @用户变量名:=值;
SELECT @用户变量名:=值;
-- 方式二：通过select into
SELECT 字段 INTO @变量名
FROM 表;

-- 例如：查看employees表中的数量
SELECT COUNT(*) INTO @count
FROM employees;
```

3.使用(查看用户变量的值)

```sql
SELECT @用户变量名;

-- 例如：
SELECT @count;
```

局部变量的演示

1.声明

```sql
-- 只声明
DECLARE 变量名 类型;
-- 声明并初始化
DECLARE 变量名 类型 DEFAULT 值;
```

2.赋值

```sql
-- 方式一：通过set或select
SET 局部变量名=值;
SET 局部变量名:=值;
SELECT @局部变量名:=值;
-- 方式二：通过select into
SELECT 字段 INTO 局部变量名
FROM 表;
```

3.使用

```sql
SELECT 局部变量名;
```

对比用户变量和局部变量

	作用域	定义和使用的位置	语法
用户变量	当前会话	当前会话中的任何地方	必须加上@符号，不用限定类型
局部变量	begin end中	只能在begin end中，且为第一句话	一般不加@符号，需要限定类型

案例：声明两个变量并赋初值，求和，并打印

1.用用户变量来做

```
SET @m = 1;
SET @n = 2;
SET @sum = @m + @n;
SELECT @sum;
```

2.用局部变量来做

```

```

十五、存储过程和函数

存储过程

含义：

一组经过预先编译的sql语句的集合，理解成批处理语句。

好处：

- 提高了sql语句的重用性，减少了开发程序员的压力；
- 简化操作；
- 减少了编译次数并且减少了和数据库服务器的连接次数，提高了效率；

语法：

1.创建存储过程语法

```
CREATE PROCEDURE 存储过程名(参数列表)
BEGIN
    存储过程体(一组合法的SQL语句)
END
```

注意：

1.参数列表包含三部分，分别是参数模式、参数名、参数类型。

举例：

`in stuname varchar(20)`

参数模式：

- `IN`：该参数可以作为输入，也就是说该参数需要调用方传入值；
- `OUT`：该参数可以作为输出，也就是该参数可以作为返回值；

- `INOUT`：该参数既可以作为输入，又可以作为输出，也就是该参数既可以传入值，又可以返回值。

2.如果存储过程体仅仅只有一句话，`BEGIN END`可以省略。

3.存储过程体中的每条sql语句的结尾都要加分号。存储过程的结尾可以使用`DELIMITER`重新设置。

语法：

```
DELIMITER 结束标记
-- 如 DELIMITER $
```

2.调用存储过程语法

```
CALL 存储过程名(实参列表);
```

3.删除存储过程语法

```
DROP PROCEDURE 存储过程名
```

举例：

```
DROP PROCEDURE test_pro3;
```

4.查看存储过程的信息

```
SHOW CREATE PROCEDURE test_pro3;
```

空参列表

案例：

插入到`admin`表中五条记录。

```
-- 注意在cmd命令行窗口管理员模式下进行
DELIMITER $
CREATE PROCEDURE myp1()
BEGIN
    INSERT INTO admin(username, `password`)
    VALUES('join1', '0000'),
    ('lily', '0000'),
    ('tom', '0000'),
    ('zz', '0000'),
    ('dd', '0000');
END $

# 调用
CALL myp1() $
```

创建带in模式参数的存储过程

案例1：

创建存储过程实现，根据女神名，查询对应的男神信息。

```
DELIMITER $          #确定$符号为结束标记
CREATE PROCEDURE myp2(IN beautyName VARCHAR(20))
BEGIN
    SELECT bo.*
    FROM boys bo
    RIGHT JOIN beauty b ON bo.id = b.boyfriend_id
    WHERE b.name = beautyName;

END $

# 调用
CALL myp2('刘岩') $
```

案例2：

创建存储过程实现，用户是否登录成功。

```
DELIMITER $
CREATE PROCEDURE myp3(IN username VARCHAR(20), IN PASSWORD VARCHAR(20))
BEGIN
    DECLARE result INT DEFAULT 0; #变量声明并初始化

    SELECT COUNT(*) INTO result #变量赋值
    FROM admin a
    WHERE a.username = username
    AND a.password = PASSWORD;

    SELECT IF(result>0, '成功', '失败');    #变量使用
END $
#调用
CALL myp3('john', '1000') $
```

创建带out模式的存储过程

案例1：

根据女神名，返回对应的男神名。

```
DELIMITER $
CREATE PROCEDURE myp4(IN beautyName VARCHAR(20), OUT boyName VARCHAR(20))
BEGIN
    SELECT bo.boyName INTO boyName
    FROM boys bo
    INNER JOIN beauty b ON bo.id = b.boyfriend_id
    WHERE b.name = beautyName;


END $

SET @bName$     #声明用户变量
#调用
CALL myp4('张飞',@bName)$

SELECT @bName$
```

案例2：

根据女神名，返回对应的男神名和男神魅力值。

```
DELIMITER $
CREATE PROCEDURE myp5(IN beautyName VARCHAR(20), OUT boyName VARCHAR(20),
OUT userCP INT)
BEGIN
    SELECT bo.boyName, bo.userCP INTO boyName, userCP
    FROM boys bo
    INNER JOIN beauty b ON bo.id = b.boyfriend_id
    WHERE b.name = beautyName;
END $
#调用
CALL myp5('张飞',@bName, @usercp) $
SELECT @bName,@usercp$
```

创建带`inout`模式的存储过程

案例：

传入a和b两个值，最终a和b都翻倍并返回。

```
DELIMITER $
CREATE PROCEDURE myp6(INOUT a INT, INOUT b INT)
BEGIN
    SET a = a * 2;  #这里a和b都是局部变量，设置值时不用加@
    SET b = b * 2;
END $

-- 调用的时候需要先创建两个变量
SET @a = 10$
SET @b = 20$
CALL myp6(@a, @b)$
SELECT @a, @b$
```

案例

案例1：创建存储过程实现传入用户名和密码，插入到`admin`表中。

```
CREATE PROCEDURE test_pro1(IN username VARCHAR(20), IN loginPwd VARCHAR(20))
BEGIN
    INSERT INTO admin(admin.username, PASSWORD)
    VALUES(username,loginPwd);
END $
#调用
CALL test_pro1('张飞','123123')$
```

案例2：创建存储过程或函数实现传入女神编号，返回女神名称和女神电话。

```
DELIMITER $
CREATE PROCEDURE test_pro2(IN id INT, OUT gname VARCHAR(20), OUT phone
VARCHAR(20))
BEGIN
    SELECT b.`name`, b.`phone` INTO gname, phone
    FROM beauty b
    WHERE b.`id` = id;
END $
#调用
CALL test_pro2(1, @a, @b)$
SELECT @a, @b$
```

案例3：创建存储过程或函数实现传入两个女神生日，返回大小。

```
DELIMITER $
CREATE PROCEDURE test_pro3(IN birth1 DATETIME, IN birth2 DATETIME, OUT result
INT)
BEGIN
    SELECT DATEDIFF(birth1,birth2) INTO result;
END $

CALL test_pro3('1998-1-1',NOW(),@result)$
SELECT @result $
```

案例4：创建存储过程或函数实现传入一个日期，格式化成xx年xx月xx日并返回。

```
DELIMITER $
CREATE PROCEDURE test_pro4(IN mydate DATETIME,OUT strDate VARCHAR(50))
BEGIN
    SELECT DATE_FORMAT(mydate, '%y年%m月%d日') INTO strDate;
END $

CALL test_pro4(NOW(), @str)$
SELECT @str $
```

案例5：创建存储过程或函数实现传入`女神名称`，返回：`女神 and 男神`格式的字符串

```
DELIMITER $
CREATE PROCEDURE test_pro5(IN beautyName VARCHAR(50), OUT str VARCHAR(100))
BEGIN
    SELECT CONCAT(beautyName, ' and ', IFNULL(boyName, 'null')) INTO str
    FROM boys bo
    RIGHT JOIN beauty b ON b.boyfriend_id = bo.id
    WHERE b.name = beautyName;
END $

CALL test_pro5('xx', @str) $
SELECT @str $
```

案例6：创建存储过程或函数，根据传入的条目数和起始索引，查询beauty表中的记录。

```
DELIMITER $
CREATE PROCEDURE test_pro6(IN size INT, IN startIndex INT)
BEGIN
    SELECT * FROM beauty LIMIT startIndex, size;
END$
CALL test_pro6(3, 5) $
```

函数

含义：

与存储过程一样。

与存储过程的区别：

存储过程可以有0个返回，也可以有多个返回， 适合做批量插入、批量更新；而函数必须且只能有1个返回，适合做处理数据后返回一个结果。

语法：

1.创建函数语法

```
CREATE FUNCTION 函数名(参数列表) RETURNS 返回类型
BEGIN
    函数体
END
```

注意：

- 参数列表包含两个部分，分别是参数名、参数类型；
- 函数体：肯定有return语句。如果没有return语句放在函数体的后面也不会报错，但不建议。应写成`return 值;`
- 当函数体中只有一句话，则可以省略begin end；
- 使用delimiter语句设置结束标记；

2.调用函数语法

```
SELECT 函数名(实参列表)
```

案例演示

1.无参有返回

案例：返回公司的员工个数。

```
DELIMITER $
CREATE FUNCTION myf1() RETURNS INT
BEGIN
    DECLARE c INT DEFAULT 0;     #定义局部变量
    SELECT COUNT(*) INTO c       #为变量赋值
    FROM employees;

    RETURN c;           #返回值
END$

SELECT myf1()$
```

2.有参有返回

案例1：根据员工名，返回他的工资。

```
DELIMITER $
CREATE FUNCTION myf3(empName VARCHAR(20)) RETURNS DOUBLE
BEGIN
    SET @sal = 0;        #定义用户变量
    SELECT salary INTO @sal      #赋值
    FROM employees
    WHERE last_name = empName;

    RETURN @sal;                 #返回值
END $

SELECT myf3('Olson')$
```

案例2：根据部门名，返回该部门的平均工资。

```
DELIMITER $
CREATE FUNCTION myf4(deptName VARCHAR(20)) RETURNS DOUBLE
BEGIN
    DECLARE sal DOUBLE ;
    SELECT AVG(salary) INTO sal
    FROM employees e
    JOIN departments d ON e.department_id = d.department_id
    WHERE d.department_name = deptName;
    RETURN sal;
END $

SELECT myf4('IT')$
```

3.查看函数的信息

```
SHOW CREATE FUNCTION myf4;
```

4.删除函数

```
DROP FUNCTION myf4;
```

十六、流程控制结构

分类

- 顺序结构：程序从上往下依次执行。
- 分支结构：程序从两条或多条路径中选择一条去执行。
- 循环结构：程序在满足一定条件的基础上，重复执行一段代码。

分支结构

1.`if`函数

功能：

实现简单的双分支。

语法：

```
SELECT IF(表达式1,表达式2,表达式3);
--执行顺序
如果表达式1成立，则if函数会返回表达式2的值，否则会返回表达式3的值。
```

应用场合：

任何地方。

2.`case`结构

功能：

情况1：类似于Java中的switch语句，一般用于实现等值判断。

情况2：类似于Java中的多重if语句，一般用于实现区间判断。

语法：

```
-- 情况1
CASE 变量|表达式|字段
WHEN 要判断的值 THEN 返回的值1
WHEN 要判断的值 THEN 返回的值2
WHEN 要判断的值 THEN 返回的值3
...
ELSE 要返回的值n
END
-- 情况2
CASE
WHEN 要判断的条件1 THEN 返回的值1
WHEN 要判断的条件2 THEN 返回的值2
WHEN 要判断的条件3 THEN 返回的值3
...
ELSE 要返回的值n
END
```

特点：

- 可以作为表达式，嵌套在其他语句中使用，可以放在任何地方，begin end中或begin end外面；也可以作为独立的语句去使用，但这样只能放在begin end中。
- 如果when中值满足或者条件成立，则执行对应的then后面的语句，并且结束case；如果都不满足，则执行else中的语句或值。
- else可以省略，如果else省略了，并且所有when条件都不满足，则返回NULL。

案例：

创建一个存储过程，根据传入的成绩，来去显示等级，比如传入的成绩在[90-100]中间，则显示A；[80-90)之间，则显示B；[60-80)之间，显示C；否则，显示D；

```
DELIMITER $
CREATE PROCEDURE test_case(IN score INT)
BEGIN
    CASE
    WHEN score>=90 AND score<=100 THEN SELECT 'A';
    WHEN score>=80 THEN SELECT 'B';
    WHEN score>=60 THEN SELECT 'C';
    ELSE SELECT 'D';
    END CASE;
END $

CALL test_case(88)$
```

3. `if`结构

功能：

实现多重分支。

语法：

```
IF 条件1 THEN 语句1;
ELSEIF 条件2 THEN 语句2;
...
【ELSE 语句n;】      -- 这一句可以省略
END IF;
```

应用场合：

只能应用在begin end中。

案例：

创建一个存储过程，根据传入的成绩，来去显示等级，比如传入的成绩在[90-100]中间，则返回A；[80-90)之间，则返回B；[60-80)之间，返回C；否则，返回D；

```
DELIMITER $
CREATE FUNCTION test_if(score INT)RETURNS CHAR(1)
BEGIN

    IF score >= 90 AND score <= 100 THEN RETURN 'A';
    ELSEIF score >= 80 THEN RETURN 'B';
    ELSEIF score >= 60 THEN RETURN 'C';
    ELSE RETURN 'D';
    END IF;
END $

SELECT test_if(88)$
```

循环结构

分类：

- while
- loop
- repeat

循环控制：

iterate类似于continue，继续，结束本次循环，继续下一次；

leave类似于break，跳出，结束当前所在的循环。

语法：

1.`while`

```
【标签:】WHILE 循环条件  DO
    循环体
END WHILE 【标签】;

-- 如果想要加入循环控制，则需要写标签。
```

2.`loop`

```
【标签:】loop
    循环体
end loop 【标签】;
-- 如果想要加入循环控制，则需要写标签。
-- 可以用来模拟简单的死循环
```

3.`repeat`

```
【标签:】repeat
    循环体;
until 结束循环的条件
end repeat【标签】;
-- 如果想要加入循环控制，则需要写标签。
```

案例演示

案例1：批量插入，根据次数插入到`admin`表中多条记录。(不添加循环控制语句)

```
DELIMITER $
CREATE PROCEDURE pro_while1(IN insertCount INT)
BEGIN
    DECLARE i INT DEFAULT 1;
    WHILE i <= insertCount DO
        INSERT INTO admin(username,PASSWORD) VALUES(CONCAT('hh',i),'adb');
        SET i = i + 1;
    END WHILE;
END $

CALL pro_while1(3)$
```

案例2：批量插入，根据次数插入到`admin`表中多条记录，如果次数>20则停止。(使用循环控制语句)

```
DELIMITER $
CREATE PROCEDURE pro_while2(IN insertCount INT)
BEGIN
    DECLARE i INT DEFAULT 1;
    a:WHILE i <= insertCount DO
        INSERT INTO admin(username,PASSWORD) VALUES(CONCAT('hh',i),'adb');
        IF i >= 20 THEN LEAVE a;
        END IF;
        SET i = i + 1;
    END WHILE a;
END $

CALL pro_while2(32)$
```

案例3：批量插入，根据次数插入到`admin`表中多条记录，直插入偶数次。(添加iterate语句)

```
DELIMITER $
CREATE PROCEDURE pro_while3(IN insertCount INT)
BEGIN
    DECLARE i INT DEFAULT 0;
    a:WHILE i <= insertCount DO
        SET i = i + 1;
        IF MOD(i,2) != 0 THEN ITERATE a;
        END IF;
        INSERT INTO admin(username,PASSWORD) VALUES(CONCAT('hh',i),'adb');

    END WHILE a;
END $

CALL pro_while3(32)$
```

案例4：已知表`stringcontent`，其中字段`id 自增长`、`content varchar(20)`，向该表插入指定个数的，随机的字符串。

```
-- 创建表
CREATE TABLE stringcontent(
    id INT PRIMARY KEY AUTO_INCREMENT,
    content VARCHAR(20)
);
-- 插入数据事务
DELIMITER $
CREATE PROCEDURE test_randstr_insert(IN insertCount INT)
BEGIN
    DECLARE i INT DEFAULT 1;         #定义一个循环变量i,表示插入次数
    DECLARE str VARCHAR(26) DEFAULT 'abcdefghijklmnopqrstuvwxyz';
    DECLARE startIndex INT DEFAULT 1;    #代表起始索引长度
    DECLARE len INT DEFAULT 1;       #代表截取的字符的长度
    WHILE i <= insertCount DO
        SET len = FLOOR(RAND()*(20-startIndex+1)+1);         #产生一个随机的整数，代表截取长度，1- (26-startIndex+1)
        SET startIndex = FLOOR(RAND()*26+1);             #产生一个随机的整数，代表起始索引1-26
        INSERT INTO stringcontent(content)VALUES(SUBSTR(str,startIndex,len));
        SET i = i + 1;           #循环变量更新
    END WHILE;
```

```
20
21 END $
22
23 CALL test_randstr_insert(14)$
```

小结：