

## MySQL常用命令

登录: `mysql -u root -p`

退出: `exit`

查看已经存在的数据库: `show databases;`

创建数据库: `create database 数据库名;`

切换到要使用的数据库: `use 数据库名;`

查看当前数据库表格: `show tables;`

初始化数据/导入SQL文件: `source 文件路径`

删除数据库: `drop database 数据库名;`

查看表结构: `desc 表名;`

查看当前使用的数据库: `select database();`

查看数据版本: `select version();`

结束一条语句: `\c`

查看建表语句: `show create table 表名;`

## MySQL语句分类

DQL (数据查询语言) : 查询语句, 凡是select语句都是DQL。

DML (数据操作语言) : insert delete update, 对表当中的数据进行增删改。

DDL (数据定义语言) : create drop alter, 对表结构的增删改。

TCL (事务控制语言) : commit提交事务, rollback回滚事务。(TCL中的T是Transaction)

DCL (数据控制语言) : grant授权、revoke撤销权限等。

## MySQL的SQL脚本

当一个文件的扩展名是.sql, 并且该文件中编写了大量的sql语句, 我们称这样的文件为sql脚本。

注意: 直接使用source命令可以执行sql脚本。

sql脚本中的数据量太大的时候, 无法打开, 使用source命令完成初始化。

## MySQL DQL

### 简单查询

语法格式:

`select 字段名1,字段名2,字段名3,... from 表名;`

提示:

- 1、任何一条sql语句以 “;” 结尾。
- 2、sql语句不区分大小写。

例如:

查询员工名字:

`select ENAME from emp;`

查询员工年薪:

`select ENAME,SAL*12 from emp; (字段可以参与运算)`

给查询结果的列重命名:

`select ENAME,SAL*12 as YEARSAL from emp;`

as关键字可以省略:

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

重命名为中文别名：

```
select ENAME,SAL*12 as '年薪' from emp;
```

注意：标准sql语句中要求字符串使用单引号括起来。虽然mysql支持双引号，尽量别用。

查询所有字段：

```
select * from emp; // 实际开发中不建议使用*，效率较低。（会把*转为字段）
```

## 条件查询

语法格式：

```
select
    字段,字段...
from
    表名
where
    条件;
```

执行顺序：先from，然后where，最后select

例如：

查询工资等于5000的员工姓名：

```
select ename from emp where sal=5000;
```

查询SMITH的工资：

```
select sal from emp where ename = 'SMITH'; // 字符串使用单引号括起来。
```

找出工资大于等于3000的员工：

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

找出工资不等于3000的

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

找出工资在1100和3000之间的员工，包括1100和3000

```
select ename,sal from emp where sal>=1100 and sal <=3000;
select ename,sal from emp where sal between 1100 and 3000;
between...and...闭区间
select ename,sal from emp where sal between 3000 and 1100; // 查询不到任何数据
```

注意：between and在使用的时候必须左小右大

between and除了可以使用在数字方面之外，还可以使用在字符串方面。

查询名字首字母在A到D区间

```
select ename from emp where ename between 'A' and 'D'; // 左闭右开
```

找出哪些人津贴为NULL：

在数据库当中NULL不是一个值，代表什么也没有，为空。

空不是一个值，不能用等号衡量。

必须使用 is null或者is not null

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

找出哪些人津贴不为NULL：

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

找出哪些人没有津贴

```
select ename,sal,comm from emp where comm is null or comm=0;
```

找出工作岗位是MANAGER和SALESMAN的员工

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

and和or联合起来用：找出薪资大于1000的并且部门编号是20或30部门的员工。

```
select ename,sal,deptno from emp where sal>1000 and (deptno=20 or deptno=30);
```

注意：当运算符的优先级不确定的时候加小括号。

**条件查询in：in等同于or：**

找出工作岗位是MANAGER和SALESMAN的员工

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

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

找出工资为800或者5000的

```
select ename,job from emp where sal in(800, 5000); // in后面的值不是区间，是具体的值
```

not in:不在这几个值当中

```
select ename,sal from emp where sal not in (800,5000);
```

## 模糊查询like

找出名字当中含有O的（在模糊查询当中，必须掌握两个特殊的符号，一个是%，一个是\_）

%代表任意多个字符，\_代表任意1个字符

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

找出名字中第一个字母是A的

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

找出名字中第二个字母是A的

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

找出名字中有下划线的

```
select name from t_user where name like '%_ %'; (\转义字符)
```

找出名字中最后一个字母是T的

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

## 数据排序

按照工资升序，找出员工名和薪资

```
select ename,sal from emp order by sal; (默认升序)
```

```
select ename,sal from emp order by sal asc; (指定升序)
```

```
select ename,sal from emp order by sal desc; (指定降序)
```

按照工资的降序排列，当工资相同的时候再按照名字的升序排列。

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

注意：越靠前的字段越能起到主导作用。只有当前面的字段无法完成排序的时候，才会启用后面的字段。（前面字段相等时，才会启用后面字段）

可以用字段顺序进行排序

```
select ename,sal from emp order by 2; //2代表第二列
```

找出工作岗位是SALESMAN的员工，并且要求按照薪资的降序排列。

```
select ENAME,JOB,SAL from emp where JOB='SALESMAN' order by SAL desc;
```

## 分组函数

count 计数

sum 求和

avg 平均值

max 最大值

min 最小值

记住：所有的分组函数都是对“某一组”数据进行操作

找出工资总和

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

找出最高工资

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

找出最低工资

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

找出平均工资

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

找出总人数

```
select count(*) from emp;
```

```
select count(ename) from emp;
```

分组函数一共五个。分组函数还有另一个名字：多行处理函数。

多行处理函数的特点：输入多行，最终输出的结果是1行。

分组函数自动忽略NULL。不需要加where comm is not null

count(\*)和count(具体的某个字段)，他们有什么区别？

count(\*)：不是统计某个字段中数据的个数，而是统计总记录条数。（和某个字段无关）

count(comm)：表示统计comm字段中不为NULL的数据总数量。

分组函数也能组合起来用：

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

找出工资高于平均工资的员工

```
select ename,sal from emp where sal > avg(sal); //ERROR 1111 (HY000): Invalid use of group function
```

原因：SQL语句当中有一个语法规则，分组函数不可直接使用在where子句当中。分组函数在group by之后执行，group by是再在where执行之后执行的

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

计算每个员工的年薪

```
select ename,(sal+comm) yearsal from emp;
```

重点：所有数据库都是这样规定的，只要有NULL参与的运算结果一定是NULL。使用ifnull函数：

ifnull(可能为NULL的数据,被当做什么处理)：属于单行处理函数。

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

## group by和having

group by：按照某个字段或者某些字段进行分组。

having：having是对分组之后的数据进行再次过滤。

找出每个工作岗位的最高薪资。

```
select job,max(sal) from emp group by job; (先进行分组，再对每组求最大值)
```

先执行from emp 再group by job 再select

注意：分组函数一般都会和group by联合使用，这也是为什么它被称为分组函数的原因。

并且任何一个分组函数（count sum avg max min）都是在group by语句执行结束之后才会执行的。

当一条sql语句没有group by的话，整张表的数据会自成一组。

## 查询语句执行顺序

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

## 多字段分组查询

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

以上在mysql当中，查询结果是有的，但是结果没有意义，在Oracle数据库当中会报错。语法错误。

Oracle的语法规则比MySQL语法规则严谨。

规则：当一条语句中有group by的话，select后面只能跟分组函数和参与分组的字段。

每个工作岗位的平均薪资

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

## 多个字段联合起来一块分组

找出每个部门不同工作岗位的最高薪资。

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

找出每个部门的最高薪资，要求显示薪资大于2900的数据

```
select max(sal),deptno from emp group by deptno having max(sal) > 2900; // 这种方式效率低。
```

```
select max(sal),deptno from emp where sal > 2900 group by deptno; // 效率较高，建议能够使用where过滤的  
尽量使用where。
```

找出每个部门的平均薪资，要求显示薪资大于2000的数据

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

## 查询结果集的去重

加入distinct

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

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

以上的sql语句是错误的。

记住：distinct只能出现在所有字段的最前面。

distinct出现在最前方表示所有字段联合起来去重

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

统计岗位的数量

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

## 连接查询

在实际开发中，大部分的情况下都不是从单表中查询数据，一般都是多张表联合查询取出最终的结果。

在实际开发中，一般一个业务都会对应多张表，比如：学生和班级，起码两张表。

stuno	stuname	classno	classname
1	zs	1	高三一班
2	ls	1	高三一班
...			

学生和班级信息存储到一张表中，结果就像上面一样，数据会存在大量的重复，导致数据的冗余。

### 连接查询的分类

根据语法出现的年代来划分的话，包括：

SQL92（一些老的DBA可能还在使用这种语法）

SQL99（比较新的语法）

根据表的连接方式来划分，包括：

内连接：

等值连接

非等值连接

自连接

外连接：

左外连接（左连接）

右外连接（右连接）

全连接（很少用）

在表的连接查询方面有一种现象被称为：笛卡尔积现象。（笛卡尔乘积现象）

笛卡尔积现象：当两张表进行连接查询的时候，没有任何条件进行限制，最终的查询结果条数是两张表记录条数的乘积。

避免笛卡尔积现象：加条件进行过滤

注意：避免了笛卡尔积现象，不会减少记录的匹配次数，只不过显示的是有效记录

### 关于表的别名：

select e.ename,d.dname from emp e,dept d;

表的别名优点：

第一：执行效率高

第二：可读性好

emp表

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	1980-12-17	800.00	NULL	20
7499	ALLEN	SALESMAN	7698	1981-02-20	1600.00	300.00	30
7521	WARD	SALESMAN	7698	1981-02-22	1250.00	500.00	30
7566	JONES	MANAGER	7839	1981-04-02	2975.00	NULL	20
7654	MARTIN	SALESMAN	7698	1981-09-28	1250.00	1400.00	30
7698	BLAKE	MANAGER	7839	1981-05-01	2850.00	NULL	30
7782	CLARK	MANAGER	7839	1981-06-09	2450.00	NULL	10
7788	SCOTT	ANALYST	7566	1987-04-19	3000.00	NULL	20
7839	KING	PRESIDENT	NULL	1981-11-17	5000.00	NULL	10
7844	TURNER	SALESMAN	7698	1981-09-08	1500.00	0.00	30
7876	ADAMS	CLERK	7788	1987-05-23	1100.00	NULL	20
7900	JAMES	CLERK	7698	1981-12-03	950.00	NULL	30
7902	FORD	ANALYST	7566	1981-12-03	3000.00	NULL	20
7934	MILLER	CLERK	7782	1982-01-23	1300.00	NULL	10

dept表

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

salgrade表

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

找出每一个员工的部门名称，要求显示员工名和部门名

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

### 内连接之等值连接

最大特点是：条件是等量关系。

查询每个员工的部门名称，要求显示员工名和部门名。

SQL92: `select e.ename,d.dname from emp e,dept d where e.deptno=d.deptno;`

SQL99: `select e.ename,d.dname from emp e join dept d on e.deptno=d.deptno;`

// inner可以省略的，带着inner目的是可读性好一些。

`select e.ename,d.dname from emp e inner join dept d on e.deptno = d.deptno;`

语法：

```
...
A表
join
B表
on
连接条件
where
...
```

SQL99语法结构更清晰一些：表的连接条件和后来的where条件分离。

内连接之非等值连接

特点是：连接条件中的关系是非等量关系。

找出每个员工的工资等级，要求显示员工名、工资、工资等级

```
select
    e.ename,e.sal,s.grade
from
    emp e
join
    salgrade s
on
    e.sal between s.losal and hisal;
```

### 自连接

特点是：一张表看做两张表。自己连接自己。

找出每个员工的上级领导，要求显示员工名和对应的领导名。

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	1980-12-17	800.00	NULL	20
7499	ALLEN	SALESMAN	7698	1981-02-20	1600.00	300.00	30
7521	WARD	SALESMAN	7698	1981-02-22	1250.00	500.00	30
7566	JONES	MANAGER	7839	1981-04-02	2975.00	NULL	20
7654	MARTIN	SALESMAN	7698	1981-09-28	1250.00	1400.00	30
7698	BLAKE	MANAGER	7839	1981-05-01	2850.00	NULL	30
7782	CLARK	MANAGER	7839	1981-06-09	2450.00	NULL	10
7788	SCOTT	ANALYST	7566	1987-04-19	3000.00	NULL	20
7839	KING	PRESIDENT	NULL	1981-11-17	5000.00	NULL	10
7844	TURNER	SALESMAN	7698	1981-09-08	1500.00	0.00	30
7876	ADAMS	CLERK	7788	1987-05-23	1100.00	NULL	20
7900	JAMES	CLERK	7698	1981-12-03	950.00	NULL	30
7902	FORD	ANALYST	7566	1981-12-03	3000.00	NULL	20
7934	MILLER	CLERK	7782	1982-01-23	1300.00	NULL	10

员工的领导编号 = 领导的员工编号

```

select
    a.ename as '员工名', b.ename as '领导名'
from
    emp a
inner join
    emp b
on
    a.mgr=b.empno;

```

## 外连接

内连接：

假设A和B表进行连接，使用内连接的话，凡是A表和B表能够匹配上的记录查询出来，这就是内连接。

AB两张表没有主副之分，两张表是平等的。

外连接：

假设A和B表进行连接，使用外连接的话，AB两张表中有一张表是主表，一张表是副表，主要查询主表中的数据，捎带着查询副表，当副表中的数据没有和主表中的数据匹配上，副表自动模拟出NULL与之匹配。

外连接的分类

左外连接（左连接）：表示左边的这张表是主表。

右外连接（右连接）：表示右边的这张表是主表。

左连接有右连接的写法，右连接也会有对应的左连接的写法。

找出每个员工的上级领导（所有员工必须全部查询出来。）

内连接：

```

select
    a.ename '员工', b.ename '领导'
from
    emp a
join
    emp b
on
    a.mgr = b.empno;

```

外连接：（左外连接/左连接）

```

select
    a.ename '员工', b.ename '领导'
from
    emp a
left join
    emp b

```



```
on
    a.mgr = b.empno;
```

外连接：（右连接）

```
select
    a.ename '员工', b.ename '领导'
from
    emp b
right join
    emp a
on
    a.mgr = b.empno;
```

// outer是可以省略的。

```
select
    a.ename '员工', b.ename '领导'
from
    emp a
left outer join
    emp b
on
    a.mgr = b.empno;
```

外连接最重要的特点是：主表的数据无条件的全部查询出来。

找出哪个部门没有员工？

```
select
    d.*
from
    emp e
right join
    dept d
on
    e.deptno=d.deptno
where
    e.empno is null;
```

### 三张表连接查询

语法：

```
....
    A
join
    B
join
    C
on
    ...
```

注意：A表和B表先进行表连接，连接之后A表继续和C表进行连接

找出每一个员工的部门名称以及工资等级

```
select
```

```

        e.ename,d.dname,s.grade
from
    emp e
join
    dept d
on
    e.deptno=d.deptno
join
    salgrade s
on
    e.sal between s.losal and s.hisal;

```

找出每一个员工的部门名称、工资等级、以及上级领导

```

select
    e.ename '员工',d.dname,s.grade,e1.name '领导'
from
    emp e
join
    dept d
on
    e.deptno=d.deptno
join
    salgrade s
on
    e.sal between s.losal and s.hisal;
join
    left emp e1
on
    e.mgr=e1.empno;

```

## 子查询

select语句当中嵌套select语句，被嵌套的select语句是子查询。

子查询可以出现的位置：

```

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

```

### where中使用子查询

找出高于平均薪资的员工信息

```

select * from emp where sal > (select avg(sal) from emp);

```

### from后面嵌套子查询

找出每个部门平均薪水的等级

```

select
    t.*,s.grade
from
    (select deptno,avg(sal) avg_sal from emp group by deptno) t
join

```

```

        salgrade s
    on
        t.avgsal between s.losal and s.hisal

```

找出每个部门平均的薪水等级

```

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

```

### 在select后面嵌套子查询

找出每个员工所在的部门名称，要求显示员工名和部门名

```

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

```

### union (可以将查询结果集相加)

找出工作岗位是SALESMAN和MANAGER的员工

```

第一种: select ename,job from emp where job = 'MANAGER' or job = 'SALESMAN';
第二种: select ename,job from emp where job in('MANAGER','SALESMAN');
第三种: union
select ename,job from emp where job = 'MANAGER'
union
select ename,job from emp where job = 'SALESMAN';

```

两张不相干的表中的数据拼接在一起显示 (列数要一致)

```

select ename from emp
union
select dname from dept;

```

### limit (重点, 分页查询)

limit是mysql特有的, 其他数据库中没有, 不通用。(Oracle中有一个相同的机制, 叫做rownum)

limit取结果集中的部分数据。

语法机制:

```

limit startIndex, length
startIndex表示起始位置, 从0开始, 0表示第一条数据。
length表示取几个

```

取出工资前5名的员工 (思路: 降序取前5个)

```

select ename,sal from emp order by sal desc limit 0, 5;
select ename,sal from emp order by sal desc limit 5;

```

limit是sql语句最后执行的一个环节:

```
select          5
...
from            1
...
where           2
...
group by        3
...
having          4
...
order by        6
...
limit           7
```

找出工资排名在第4到第9名的员工

```
select ename ,sal from emp order by sal desc limit 3,6;
```

### 通用的标准分页sql

每页显示3条记录:

第1页: 0, 3

第2页: 3, 3

第3页: 6, 3

第4页: 9, 3

第5页: 12, 3

每页显示pageSize条记录:

第pageNo页: (pageNo - 1) \* pageSize, pageSize

pageSize: 每页显示多少条记录

pageNo: 显示第几页

java代码:

```
int pageNo = 2; // 页码是2
```

```
int pageSize = 10; // 每页显示10条
```

```
limit (pageNo - 1) * pageSize, pageSize
```

## MySQL DDL和DML

### 创建表

建表语句的语法格式:

```
create table 表名(
    字段名1 数据类型,
    字段名2 数据类型,
    字段名3 数据类型,
    ....
);
```

### 关于MySQL当中字段的数据类型

int 整数型(java中的int)

bigint 长整型(java中的long)

float 浮点型(java中的float double)

char 定长字符串(String)

varchar 可变长字符串(StringBuffer/StringBuilder)

date	日期类型（对应Java中的java.sql.Date类型）
text	二进制文本类型，如论坛、帖子、题目等
BLOB	二进制大对象（存储图片、视频等流媒体信息） Binary Large Object（对应java中的Object）
CLOB	字符大对象（存储较大文本，比如，可以存储4G的字符串。） Character Large Object（对应java

中的Object)

### char和varchar选择

在实际的开发中，当某个字段中的数据长度不发生改变的时候，是定长的，例如：性别、生日等都是采用char。

当一个字段的数据长度不确定，例如：简介、姓名等都是采用varchar

### BLOB和CLOB类型的使用

clob用于存储大量的文本数据。大字段的操作常常以流的方式处理。

### 表名在数据库当中一般建议以：t\_或者tbl\_开始

创建学生表：

学生信息包括：

学号、姓名、性别、班级编号、生日

学号：bigint

姓名：varchar

性别：char

班级编号：int

生日：char

```
create table t_student(  
    no bigint,  
    name varchar(255),  
    sex char(1),  
    classno varchar(255),  
    birth char(10)  
);
```

### insert插入语句

语法格式：

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

要求：字段的数量和值的数量相同，并且数据类型要对应相同

```
insert into t_student(no,name,sex,classno,birth) values(1,'zhangsan','1','gaosan1ban', '1950-10-12');
```

```
insert into t_student(name,sex,classno,birth,no) values('lisi','1','gaosan1ban', '1950-10-12',2);
```

```
insert into t_student(name) values('wangwu'); // 除name字段之外，剩下的所有字段自动插入NULL。
```

```
drop table if exists t_student; // 当这个表存在的话删除。
```

```
create table t_student(  
    no bigint,  
    name varchar(255),  
    sex char(1) default 1,  
    classno varchar(255),  
    birth char(10)  
);
```

需要注意的地方：

当一条insert语句执行成功之后，表格当中必然会多一行记录。  
即使多的这一行记录当中某些字段是NULL，后期也没有办法在执行insert语句插入数据了，只能使用update进行更新。

字段可以省略不写，但是后面的value对数量和顺序都有要求。

```
insert into t_student values(1,'jack','0','gaosan2ban','1986-10-23');
```

#### 一次插入多行数据

```
insert into t_student  
(no,name,sex,classno,birth)  
values  
(3,'rose','1','gaosi2ban','1952-12-14'),(4,'laotie','1','gaosi2ban','1955-12-14');
```

## 表的复制

语法：

```
create table 表名 as select语句;
```

将查询结果当做表创建出来

#### 将查询结果插入到一张表中

```
insert into dept1 select * from dept;
```

## 修改数据

语法格式：

```
update 表名 set 字段名1=值1,字段名2=值2... where 条件;
```

注意：没有条件整张表数据全部更新。

将部门10的LOC修改为SHANGHAI，将部门名称修改为RENSHIBU

```
update dept set loc = 'SHANGHAI', dname = 'RENSHIBU' where deptno = 10;
```

更新所有记录

```
update dept1 set loc = 'x', dname = 'y';
```

## 删除数据

语法格式：

```
delete from 表名 where 条件;
```

注意：没有条件全部删除。

删除10部门数据

```
delete from dept where deptno=10;
```

删除所有记录

```
delete from dept;
```

删除大表中的数据（重点）

```
truncate table 表名; // 表被截断，不可回滚。永久丢失。
```

删除表

```
drop table 表名; // 通用。
```

```
drop table if exists 表名; // oracle不支持这种写法
```

## 约束(Constraint)

在创建表的时候，可以给表的字段添加相应的约束，添加约束的目的是为了保证表中数据的合法性、有效性、完整性。

常见的约束：

非空约束(not null)：约束的字段不能为NULL

唯一约束(unique)：约束的字段不能重复

主键约束(primary key)：约束的字段既不能为NULL，也不能重复（简称PK）

外键约束(foreign key)：被约束的值只能取被参照字段的值（简称FK）

检查约束(check)：注意Oracle数据库有check约束，但是mysql没有，目前mysql不支持该约束

### 非空约束 not null

```
drop table if exists t_user;
create table t_user(
    id int,
    username varchar(255) not null,
    password varchar(255)
);
insert into t_user(id,password) values(1,'123');
ERROR 1364 (HY000): Field 'username' doesn't have a default value
insert into t_user(id,username,password) values(1,'lisi','123');
```

### 唯一性约束 (unique)

唯一约束修饰的字段具有唯一性，不能重复。但可以为NULL。

给某一列添加unique

```
drop table if exists t_user;
create table t_user(
    id int,
    username varchar(255) unique // 列级约束
);
insert into t_user values(1,'zhangsan');
insert into t_user values(2,'zhangsan');
ERROR 1062 (23000): Duplicate entry 'zhangsan' for key 'username'
```

给两个列或者多个列添加unique

```
drop table if exists t_user;
create table t_user(
    id int,
    usercode varchar(255),
    username varchar(255),
    unique(usercode,username) // 多个字段联合起来添加1个约束unique 【表级约束】
);
insert into t_user values(1,'111','zs');
insert into t_user values(2,'111','ls');
insert into t_user values(3,'222','zs');
select * from t_user;
insert into t_user values(4,'111','zs');
ERROR 1062 (23000): Duplicate entry '111-zs' for key 'usercode'

drop table if exists t_user;
create table t_user(
```

```

        id int,
        usercode varchar(255) unique,
        username varchar(255) unique
    );
insert into t_user values(1,'111','zs');
insert into t_user values(2,'111','ls');
ERROR 1062 (23000): Duplicate entry '111' for key 'usercode'
注意：not null约束只有列级约束。没有表级约束。

```

## 主键约束

给一张表添加主键约束

```

drop table if exists t_user;
create table t_user(
    id int primary key, // 列级约束
    username varchar(255),
    email varchar(255)
);
insert into t_user(id,username,email) values(1,'zs','zs@123.com');
insert into t_user(id,username,email) values(2,'ls','ls@123.com');
insert into t_user(id,username,email) values(3,'ww','ww@123.com');
insert into t_user(id,username,email) values(1,'jack','jack@123.com');
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
insert into t_user(username,email) values('jack','jack@123.com');
ERROR 1364 (HY000): Field 'id' doesn't have a default value
根据以上的测试得出：id是主键，因为添加了主键约束，主键字段中的数据不能为NULL，也不能重复。
主键的特点：不能为NULL，也不能重复。

```

## 主键相关的术语

主键约束：primary key  
 主键字段：id字段添加primary key之后，id叫做主键字段  
 主键值：id字段中的每一个值都是主键值。

## 主键作用

- 表的设计三范式中有要求，第一范式就要求任何一张表都应该有主键。
- 主键的作用：主键值是这行记录在这张表当中的唯一标识。（就像一个人的身份证号码一样。）

## 主键的分类

根据主键字段的字段数量来划分：

单一主键（推荐的，常用的。）

复合主键(多个字段联合起来添加一个主键约束)（复合主键不建议使用，因为复合主键违背三范式。）

根据主键性质来划分：

自然主键：主键值最好就是一个和业务没有任何关系的自然数。（这种方式是推荐的）

业务主键：主键值和系统的业务挂钩，例如：拿着银行卡的卡号做主键，拿着身份证号码作为主键。（不

推荐用）

最好不要拿着和业务挂钩的字段作为主键。因为以后的业务一旦发生改变的时候，主键值可能也需要随着发生变化，但有的时候没有办法变化，因为变化可能会导致主键值重复。

一张表的主键约束只能有1个。

使用表级约束方式定义主键：

```

drop table if exists t_user;
create table t_user(
    id int,

```



```

        username varchar(255),
        primary key(id)
    );
insert into t_user(id,username) values(1,'zs');
insert into t_user(id,username) values(2,'ls');
insert into t_user(id,username) values(3,'ws');
insert into t_user(id,username) values(4,'cs');
insert into t_user(id,username) values(4,'cx');
ERROR 1062 (23000): Duplicate entry '4' for key 'PRIMARY'

```

### mysql提供主键值自增：（非常重要）

```

drop table if exists t_user;
create table t_user(
    id int primary key auto_increment, // id字段自动维护一个自增的数字，从1开始，以1递增。
    username varchar(255)
);

```

提示:Oracle当中也提供了一个自增机制，叫做：序列（sequence）对象。

### 外键约束

关于外键约束的相关术语：

外键约束: foreign key

外键字段：添加有外键约束的字段

外键值：外键字段中的每一个值。

业务背景：

请设计数据库表，用来维护学生和班级的信息

第一种方案：一张表存储所有数据

no(pk)	name	classno	classname
1	zs1	101	高三1班
2	zs2	101	高三1班
3	zs3	102	高三2班
4	zs4	102	高三2班
5	zs5	102	高三2班

缺点：冗余。【不推荐】

第二种方案：两张表（班级表和学生表）

t\_class 班级表

cno(pk)	cname
101	高三1班
102	高三2班

t\_student 学生表

sno(pk)	sname	classno(该字段添加外键约束fk)
1	zs1	101
2	zs2	101
3	zs3	102
4	zs4	102
5	zs5	102

将以上表的建表语句写出来：

t\_student中的classno字段引用t\_class表中的cno字段，此时t\_student表叫做子表。t\_class表叫做父表。

添加外键约束的字段只能取参照字段已有的值

顺序要求：

删除数据的时候，先删除子表，再删除父表。

添加数据的时候，先添加父表，在添加子表。

创建表的时候，先创建父表，再创建子表。

删除表的时候，先删除子表，在删除父表。

注意：

外键值可以为NULL

外键字段引用其他表的某个字段的时候被引用的字段不一定是主键，但至少具有unique约束。

## 存储引擎

完整的建表语句

```
CREATE TABLE `t_x` (  
  `id` int(11) DEFAULT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

注意：在MySQL当中，凡是标识符是可以使用飘号括起来的。最好别用，不通用。

建表的时候可以指定存储引擎，也可以指定字符集。

mysql默认使用的存储引擎是InnoDB方式。

默认采用的字符集是UTF8

存储引擎

存储引擎这个名字只有在mysql中存在。（Oracle中有对应的机制，但是不叫做存储引擎。Oracle中没有特殊的名字，就是“表的存储方式”）

mysql支持很多存储引擎，每一个存储引擎都对应了一种不同的存储方式。

每一个存储引擎都有自己的优缺点，需要在合适的时机选择合适的存储引擎。

常见存储引擎

```
Engine: MyISAM  
Support: YES  
Comment: MyISAM storage engine  
Transactions: NO  
XA: NO  
Savepoints: NO
```

特征：

MyISAM这种存储引擎不支持事务。

MyISAM是mysql最常用的存储引擎，但是这种引擎不是默认的。

MyISAM采用三个文件组织一张表：

xxx.frm（格式文件，存储表结构定义）

xxx.MYD（数据文件，存储表行的内容）

xxx.MYI（索引的文件，存储表的索引）

优点：可被压缩，节省存储空间。并且可以转换为只读表，提高检索效率。

缺点：不支持事务。

```
Engine: InnoDB  
Support: DEFAULT  
Comment: Supports transactions, row-level locking, and foreign keys  
Transactions: YES  
XA: YES  
Savepoints: YES
```

优点：支持事务、行级锁、外键等。这种存储引擎数据的安全得到保障。

表的结构存储在xxx.frm文件中

数据存储



  
这种InnoDB存储引擎在MySQL服务器崩溃之后提供自动恢复机制  
InnoDB支持级联删除和级联更新

```
Engine: MEMORY
Support: YES
Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
XA: NO
Savepoints: NO
```

特征:

在数据库目录中，每个表均以.frm格式的文件表示

表数据及索引被存储在内存中

表级锁机制

不能包含TEXT或BLOB字段

缺点：不支持事务。数据容易丢失。因为所有数据和索引都是存储在内存当中的

优点：查询速度最快

## 事务 (Transaction)

一个事务是一个完整的业务逻辑单元，不可再分。

比如：银行账户转账，从A账户向B账户转账10000.需要执行两条update语句：

```
update t_act set balance = balance - 10000 where actno = 'act-001';
```

```
update t_act set balance = balance + 10000 where actno = 'act-002';
```

以上两条DML语句必须同时成功，或者同时失败，不允许出现一条成功，一条失败。

要想保证以上的两条DML语句同时成功或者同时失败，那么就需要使用数据库的“事务机制”。

和事务相关的语句只有：DML语句。（insert delete update）

因为它们这三个语句都是和数据库表当中的“数据”相关的。

事务的存在是为了保证数据的完整性，安全性。

假设所有的业务都能使用1条DML语句搞定，则不要事务机制

但实际情况不是这样的，通常一个“事儿（事务【业务】）”需要多条DML语句共同联合完成

**保存点**

1执行insert

2执行delete

3执行update

4设置savepoint

5执行delete

6执行update

7执行rollback()

执行回滚操作时回滚到3，而不是全部回滚

**事务理解**

假设一个事儿，需要先执行一条insert，再执行一条update，最后执行一条delete，这个事儿才算完成。

开启事务机制（开始）

执行insert语句--> insert.....（这个执行成功之后，把这个执行记录到数据库的操作历史当中，并不会向文件中保存一条数据，不会真正的修改硬盘上的数据。）

执行update语句---> update...（这个执行也是记录一下历史操作，不会真正的修改硬盘上的数据）

执行delete语句----> delete...（这个执行也是记录一下历史操作【记录到缓存】，不会真正的修改硬盘上的数据）

提交事务或者回滚事务（结束）

提交事务和回滚事务都会清空历史记录。但提交事务会把数据持久到硬盘上

## 事务的特性

事务包括四大特性：ACID

A: 原子性：事务是最小的工作单元，不可再分

C: 一致性：事务必须保证多条DML语句同时成功或者同时失败

I: 隔离性：事务A与事务B之间具有隔离

D: 持久性：指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响

## 关于事务之间的隔离性

事务隔离性存在隔离级别，理论上隔离级别包括4个：

**第一级别**：读未提交（read uncommitted）

对方事务还没有提交，我们当前事务可以读取到对方未提交的数据（在内存中的不稳定的数据）

读未提交存在脏读（Dirty Read）现象：表示读到了脏的数据。

**第二级别**：读已提交（read committed）

对方事务提交之后的数据我方可以读取到

这种隔离级别解决了：脏读现象没有了

读已提交存在的问题是：不可重复读

**第三级别**：可重复读（repeatable read）

这种隔离级别解决了：不可重复读问题。

这种隔离级别存在的问题是：读取到的数据是幻象

**第四级别**：序列化读/串行化读（serializable）

解决了所有问题。

效率低。需要事务排队。

oracle数据库默认的隔离级别是：读已提交

mysql数据库默认的隔离级别是：可重复读

## 演示事务

mysql事务默认情况下是自动提交的。

自动提交：只要执行任意一条DML语句则提交一次

关闭自动提交：start transaction;

## 使用两个事务演示隔离级别

演示第1级别：读未提交

设置事务全局隔离级别 set global transaction isolation level read uncommitted;

查看事务隔离级别 select @@global.tx\_isolation;

```
mysql> use bjpownode;
Database changed
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 1  | zs      |
| 3  | ls      |
+----+-----+
2 rows in set (0.00 sec)
```

```
mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 1  | zs      |
| 3  | ls      |
| 4  | ww      |
+----+-----+
3 rows in set (0.03 sec)
```

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> use bjpownode;
Database changed
mysql> insert into t_user(username) values('ww');
Query OK, 1 row affected (0.00 sec)
```

```
mysql> _
```

演示第2级别：读已提交

set global transaction isolation level read committed;

```
Database changed
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 1  | zs      |
| 3  | ls      |
+----+-----+
2 rows in set (0.00 sec)
```

```
mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 1  | zs      |
| 3  | ls      |
+----+-----+
2 rows in set (0.00 sec)
```

```
mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 1  | zs      |
| 3  | ls      |
| 5  | test    |
+----+-----+
3 rows in set (0.00 sec)
```

```
mysql> use bjpownode;
Database changed
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> insert into t_user(username) values('test');
Query OK, 1 row affected (0.00 sec)
```

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> _
```

演示第3级别：可重复读

set global transaction isolation level repeatable read;

```
mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 1  | zs      |
| 3  | ls      |
| 5  | test    |
+----+-----+
3 rows in set (0.00 sec)
```

```
mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 1  | zs      |
| 3  | ls      |
| 5  | test    |
+----+-----+
3 rows in set (0.00 sec)
```

```
mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 1  | zs      |
| 3  | ls      |
| 5  | test    |
+----+-----+
3 rows in set (0.00 sec)
```

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> delete from t_user;
Query OK, 3 rows affected (0.03 sec)
```

```
mysql> commit;
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> select * from t_user;
Empty set (0.00 sec)
```

```
mysql> insert into t_user(username) values('test01');
Query OK, 1 row affected (0.03 sec)
```

```
mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 6  | test01   |
+----+-----+
1 row in set (0.00 sec)
```

```
mysql> _
```

演示第四级别：串行化读

set global transaction isolation level serializable;

```
mysql> use bjpowernode;
Database changed
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 6  | test01  |
+----+-----+
1 row in set (0.00 sec)

mysql> insert into t_user(username) values('hehe');
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.03 sec)

mysql>
```

```
mysql> use bjpowernode;
Database changed
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t_user;
+----+-----+
| id | username |
+----+-----+
| 6  | test01  |
| 7  | hehe    |
+----+-----+
2 rows in set (29.05 sec)

mysql> commit;
```

## 索引

索引就相当于一本书的目录，通过目录可以快速的找到对应的资源

在数据库方面，查询一张表的时候有两种检索方式：

第一种方式：全表扫描

第二种方式：根据索引检索（效率很高）

### 索引为什么可以提高检索效率原因

其实最根本的原理是缩小了扫描的范围

索引虽然可以提高检索效率，但是不能随意的添加索引，因为索引也是数据库当中的对象，也需要数据库不断的维护，是有维护成本的

比如，表中的数据经常被修改这样就不适合添加索引，因为数据一旦修改，索引需要重新排序，进行维护

添加索引是给某一个字段，或者说某些字段添加索引。

`select ename,sal from emp where ename = 'SMITH';`

当ename字段上没有添加索引的时候，以上sql语句会进行全表扫描，扫描ename字段中所有的值

当ename字段上添加索引的时候，以上sql语句会根据索引扫描，快速定位

### 创建索引对象 删除索引对象

创建索引对象：

`create index 索引名称 on 表名(字段名);`

删除索引对象：

`drop index 索引名称 on 表名;`

### 什么时候考虑给字段添加索引？（满足什么条件）

数据量庞大。（根据客户的需求，根据线上的环境）

该字段很少的DML操作。（因为字段进行修改操作，索引也需要维护）

该字段经常出现在where子句中。（经常根据哪个字段查询）

注意：主键和具有unique约束的字段会自动会添加索引

根据主键查询效率较高。尽量根据主键检索。

### 查看sql语句的执行计划

`explain select ename,sal from emp where sal = 5000;`

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	emp	NULL	ALL	NULL	NULL	NULL	NULL	14	10.00	Using where

type: ALL 表示全表扫描

给薪资sal字段添加索引:

```
create index emp_sal_index on emp(sal);
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	emp	NULL	ref	emp_sal_index	emp_sal_index	9	const	1	100.00	NULL

type:ref 表示索引扫描

## 索引的实现原理

索引底层采用的数据结构是: B + Tree

通过B +Tree缩小扫描范围, 底层索引进行了排序, 分区, 索引会携带数据在表中的“物理地址”, 最终通过索引检索到数据之后, 获取到关联的物理地址, 通过物理地址定位表中的数据, 效率是最高的。

```
select ename from emp where ename = 'SMITH';
```

通过索引转换为:

```
select ename from emp where 物理地址 = 0x3;
```



## 索引的分类

单一索引: 给单个字段添加索引

复合索引: 给多个字段联合起来添加1个索引

主键索引: 主键上会自动添加索引

唯一索引: 有unique约束的字段上会自动添加索引

## 索引什么时候失效

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

模糊查询的时候, 第一个通配符使用的是%, 这个时候索引是失效的。

## 视图(view)

### 视图

站在不同的角度去看到数据。(同一张表的数据, 通过不同的角度去看待)。

### 创建视图 删除视图

```
create view myview as select empno,ename from emp;
```

```
drop view myview;
```

注意: 只有DQL语句才能以视图对象的方式创建出来。

对视图进行增删改查, 会影响到原表数据。(通过视图影响原表数据的, 不是直接操作的原表) 可以对视图进行CRUD操作

```
create table emp_bak as select * from emp;
```

```
create view myview as select empno,ename,sal from emp_bak;
update myview set ename='hehe',sal=1 where empno = 7369; // 通过视图修改原表数据
delete from myview1 where empno = 7369; // 通过视图删除原表数据
```

### 视图的作用

视图可以隐藏表的实现细节。保密级别较高的系统，数据库只对外提供相关的视图，java程序员只对视图对象进行CRUD。

## DBA命令

### 将数据库当中的数据导出

在windows的dos命令窗口中执行：（导出整个库）

```
mysqldump bjpownode>D:\bjpownode.sql -uroot -pmysql5721
```

在windows的dos命令窗口中执行：（导出指定数据库当中的指定表）

```
mysqldump bjpownode emp>D:\bjpownode.sql -uroot -pmysql5721
```

### 导入数据

```
create database bjpownode;
```

```
use bjpownode;
```

```
source D:\bjpownode.sql
```

## 数据库设计三范式（重点内容）

设计表的依据。按照这个三范式设计的表不会出现数据冗余。

### 三范式

**第一范式：**任何一张表都应该有主键，并且每一个字段原子性不可再分

**第二范式：**建立在第一范式的基础之上，所有非主键字段完全依赖主键，不能产生部分依赖

学生编号(PK)	教师编号(PK)	学生姓名	教师姓名
1001	001	张三	王老师
1002	002	李四	赵老师
1003	001	王五	王老师
1001	002	张三	赵老师

学生只依赖两个主键中的学生编号，而不依赖教师编号，产生数据冗余。

**多对多关系解决：三张表，关系表、两个外键**

t\_student 学生表

sno(pk)      sname

-----

1            张三

2            李四

3            王五

t\_teacher 讲师表

tno(pk)      tname

-----

1            王老师

2            张老师

3            李老师

t\_student\_teacher\_relation 学生讲师关系表

id(pk)            sno(fk)            tno(fk)

-----





1	zs	123
2	ls	456

t\_user\_detail 用户详细信息表

id(pk)	realname	tel	userid(fk+unique)....
1	张三	1111111111	2
2	李四	1111415621	1