windows cmd 命令行 启动和关闭mysql服务

`net stop` 服务名称
`net start` 服务名称 (MySQL80)

退出MySQL

`exit`

登录MySQL

使用bin 目录下的mysqlL.exe 命令来连接mysql数据库服务器
本地登录 :
显示密码登录
`mysql -urrot -p123456`
隐藏密码登录
`mysql -urrot -p`
`123456`

查看MySQL有哪些数据库

`show databases;`
以分号结尾

查看某个数据库下有那些表

`show tables;`

使用某个数据库

`use 数据库名`

创建数据库

`create database bjpowernode`

导入数据

source 地址

数据库中最基本的单元是表: table

数据库中是以表格的形式表示数据的

因为表比较直观

任何一张表都有行和列:

```
行(row) : 被称为数据/记录
列(column) : 被称为字段
```

每一个字段都有 : 字段名, 数据类型, 约束等属性.

SQL分类

DQL: 数据查询语言(凡是带有select关键字的都是查询语句)

select. . .

DML:

数据操作语言(凡是对表当中的数据进行增删改的都是DML)
insert delete update
insert 增
delete 删
update 改
这个主要是操作表中的数据data.

DDL:

数据定义语言
凡是带有create、drop、 alter的 都是DDL
DDL主要操作的是表的结构。不是表中的数据。
create:新建，等同于增
drop:删除
alter:修改
这个增删改和DML不同，这个主要是对表结构进行操作。

TCL:

是事务控制语言
包括:
事务提交: commit;事务回滚: rollback;

DCL:

是数据控制语言.
例如:授权grant、 撤销权限revoke....

查看表中的数据

`select * from 表名;`

查看表中的结构

`desc 表名;`

查询mysql的版本号：`select version（）;`
查询当前使用的哪个数据库：`select database（）`；也可直接再use一下需要的数据库；
结束一条语句：\c 命令

简单查询

查询一个字段

`select 字段名 from 表名;`
其中select和from都是关键字
字段名和表名是标识符

查询多个字段

使用" , "号隔开
`select 字段名,字段名 from 表名;`

查询多个字段

第一种方式: 把所有字段都写上
`select a,b,c from 表名;`
第二种方式: 可以使用*
`select * from 表名;`
这种方式的缺点:

```
1. 效率低
2. 可读性差
在实际开发中不建议
```

给查询的列起别名

使用as关键字
`原名 as 修改名`
注意: 只是将显示的查询结果列名显示为修改名, 原名是不会变化的
select 语句是永远都不会进行修改操作的(因为只负责查询)
as可以省略

条件查询

select 字段，字段...from 表名 where 条件；
条件（> ,>=,<,<=,<>或者！=，between 小数字 and 大数字 ,）

运算符	说明
=	等于
<> 或 !=	不等于
<	小于
<=	小于等于
>	大于
>=	大于等于
between ... and ...	两个值之间
is null	为null（is not null不为空）
and	并且
or	或者
in	包含，相当于多个or（not in 不在这个范围中）
not	not可以取非，主要用在is或in中
like	like成为模糊查询，支持%或_匹配 %表示任意多个字符 _表示任意一个字符

排序

升序

默认是升序

`order by 字段;` 默认

`order by 字段 asc;`

降序

`order by 字段 desc;`

如果按照某个字段排序，而该字段又相等则再在后面的字段加上，第二个字段名 asc/desc；

例如：select 字段1，字段2，from表名 order by 字段名1 desc，字段名2 asc；（越靠前的字段越能起主导作用，order by后面是最后执行的）

数据处理函数

又叫单行处理函数

特点：一行一行处理，每一行输出一个结果

可以改变每一行的数值

Lower	转换小写
upper	转换大写
substr	取子串 格式:(substr(被截取的字符串,起始下标,截取长度))
length	取长度
trim	去空格
str_to_date	将字符串转换成日期
date_format	格式化日期
format	设置千分位
round	四舍五入
ranf()	生成随机数
ifnull	可以将null转换成具体值 ifnull(字段, 当中什么数)
concat	字符串拼接 concat(字符1, 字符2)

case ... when ... then ... when ... then ... else ... end

```
 select
    ename
    job
    sal as oldsal
       (case job when 'MANAGER' then sal  * 1.1 when 'SALESMAN' then sal*1.5 else sal end) as newsal
    from
    emp;
```

case

case字段 when 数据 then 相应操作 when 数据 then 操作 else end

格式化数字: format(数字, '格式')

select ename , format (sal , '$999,999') as sal form emp;

str_to_date函数可以将字符串转换成日期类型date

语法格式:
str_to_date ('字符串日期' , '日期格式')

mysql的日期格式:

%Y 年
%m 月
%d 日
%h 时
%i 分
%s 秒

date_format

这个函数可以讲日期类型转换成特定格式的做饭吃
select id , name , date_format(brith , ' %m/%d/%Y ') as brith from t_user;

date和datetime两个类型的区别

date是短日期: 只包括年月日信息　默认格式 :
$Y-datetime$是长日期 : 包括年月日时分秒信息默认格式 :Y-%m-%d　%h : %i : %s

在mysql当中获取系统当前时间

new()

分组函数(多行处理函数)

多行处理函数的特点 : 输入多行, 最终输出一行

5个 :
　count 计数
　sum　求和
　avg　平均值
　max　最大值
　min　最小值
注意 :
分组函数在使用的时候必须先进行分组, 然后才能用
如果你没有对数据进行分组, 整张表默认为一组

1. 分组函数自动忽略null
2. count(具体字段) : 表示统计该字段下所有不为null的元素的总数
   count(*) : 表示统计表中的总行数, (只要有一行数据count则++)
   因为每一行记录不可能都为null, 一行数据中有一列不为null, 则这行数据就是有效的
3. 分组函数不能够直接使用在where子句中
4. 所有的分组函数可以组合起来一起用

分组查询

什么是分组查询

在实际的应用中, 可能有这样的需求, 需要先进行分组, 然后对每一组的数据进行操作
这个时候我们需要使用分组查询, 怎么进行分组查询
select
　...
from
　...

group by
    ... ;

计算每个部门的工资和?

计数每个工作岗位的平均薪资?

以上关键字的执行顺序

select
    ...
from
    ...
where
    ...
group by
    ...
order by
    ...

--------执行顺序---------

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

having

使用having可以对分完组之后的数据进一步过滤

having不能单独使用, having不能代替where, having必须和group by 联合使用

优先使用where

找出每个部门最高薪资
按照部门编号分组, 求每一组最大值
`select deptno , max(sal) from emp group by deotno;`

要求显示最高薪资大于3000

```
select
    deptno, max(sal)
from
    emp
group by
    deotno
having
    max(sal) > 3000;
```

优先使用where

先将大于3000的都找出来, 然后再分组

```
select
   deptno, max(sal)
from
   emp
where
   sal > 3000
group by
   deotno;
```

第一阶段总结

```
select
   ...
where
   ...
group by
   ...
having
   ...
order by
   ...
```

执行顺序

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

从某张表查询数据
先经过where条件筛选出有价值的数据
对这些有价值数据进行分组
分组之后可以使用having继续筛选
select查询出来

去除重复记录

distinct 把查询结果取出重复记录
注意 : 原数据不会被修改, 只是查询结果去重

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

distinct出现在两个字段最前方 , 表示两个字段联合起来去重

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

limit 与offset的用法

起始下标默认从0开始
在mysql中一般使用limit来实现分页

分页公式

每页显示paseSize条记录

第pageNo页 : limit (pageNo - 1) * paseSize , paseSize

1. LIMIT后面跟一个参数，表示要提取的数量。
   如： select* from test LIMIT 3 指提取前三条数据，类似sqlServer的top语法。
2. LIMIT后面跟两个参数时，第一个参数是指第几行，第二个参数是取几条数据。
   如： select * from test limit 2,3; 这个SQL是指从第二行的下一行开始向下取3条数据。(即取：3，4，5行的三条数据)
3. LIMIT和OFFSET组合使用时，LIMIT后面只能有一个参数，表示要提取的数量，offset后面的数字则表示第几行。
   如：select * from test LIMIT 3 offset 2; 这个SQL是指从第二行的下一行开始向下取3条数据。(即取：3，4，5行的三条数据)

连接查询

什么是连接查询

从一张表中单独查询, 称为单表查询
从emp表和dept表联合起来查询数据, 从emp表中取出员工名字, 从dept表中去部门名字
这种跨表查询, 多张表联合起来查询数据, 被称为连接查询

连接查询的分类

根据语法的年代分类
  SQL92
  SQL99

根据表连接的方式分类:
  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(不讲)

SQL92语法:

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

SQL99语法:

```
select
```

e.ename , d.dname
from
emp e
join
dept d
on //条件
e.depton = d.depton

子查询

什么是子查询

select 语句中嵌套select语句, 被嵌套的select语句称为子查询

子查询都可以出现在哪里

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

union合并查询结果集

union 注意事项:
union在进行结果合并的时候, 要求两个结果集列数相同

这才是完整的SELECT查询

完整的SELECT查询

```
SELECT DISTINCT column, //AGG_FUNC(column_or_expression), …
FROM mytable
    JOIN another_table
      ON mytable.column = //another_table.column
    WHERE constraint_expression
    GROUP BY column
    HAVING constraint_expression
    ORDER BY column ASC/DESC
    LIMIT count OFFSET COUNT;
```

执行顺序

1. FROM 和 JOINs
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. DISTINCT
7. ORDER BY

8. LIMIT / OFFSET

表的创建

建表的语法格式:

建表属于DDL语句, DDL包括: create drop alter

cerate 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);
cerate 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
);
表名: 建议以t_ 或者tbl_开始, 可读性强 见名知意
字段名: 见名知意

mysql数据类型

常见
varchar
可变长度的字符串
比较智能, 节省空间
会根据实际的数据长度动态分配空间

```
优点: 节省空间
缺点: 需要动态分配空间, 速度慢
```

char
定长字符串
不管实际的数据长度是多少
分配固定长度的空间去存储数据
使用不恰当的时候可能会导致空间的浪费

```
优点 : 不需要动态分配空间, 速度快
缺点 : 使用不恰当可能会导致空间的浪费
```

int
数字中的整数型.等同于java的int
bigint
数字中的长整型.等同于java中的long
float
单精度浮点型数据
double
双精度浮点型数据
date
短日期类型
datetime
长日期类型

clob
字符大对象
最多可以存储4G的字符串
比如: 存储一篇文章, 存储一个说明
超过255个字符都要采用CLOB字符大对象来存储
blob
二进制大对象
专门用来存储图片, 声音, 视频等流媒体数据
往BLOB类型的字符上插入数据的时候, 例如插入一个图片, 视频等
使用IO流

删除表

drop table t_student; //当这张表不存在的时候会报错

drop table if exists t_student; //如果这张表存在的话, 删除

插入数据insert(DML)

语法格式 :
insert into 表名(字段名1, 字段名2, 字段名3...) values(值1, 值2, 值3);
字段名和值要一一对应
数量要对应, 数据类型要对应

注意: inesrt语句但凡是执行成功了, 那么必然会多一条记录
没有给其它字段指定值的话, 默认值是null

使用default修改默认值

```
cerate table 表名(
    字段名1 数据类型 default 默认值,
    字段名2 数据类型,
    字段名3 数据类型
);
```

一次插入多条记录

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

update修改表(DML)

语法格式 表名 set 字段名1 = 值1 , 字段名2 = 值2 , 字段名3 = 值3 .... where 条件;
注意 : 没有条件限制会导致所有数据全部更新

delete删除表中的数据(DML)

语法格式
delete from 表名 where 条件;
注意 : 没有条件限制会导致表的数据全部删除

delete from t_user where id = 2 ;

delect语句删除数据的原理
表中的数据被删除了, 但是这个数据在硬盘上的真实存储空间不会被释放
缺点是: 删除效率比较低
优点是: 支持回滚, 后悔了可以再恢复数据

truncate删除

truncate table 表名
这种删除效率比较高, 表被异常截断, 物理删除
缺点: 不支持回滚
优点: 快速

快速复制表

create table emp2 as select * from emp

复制部分表

create table emp3 as select empno , ename from emp where job = 'MANAGER'

约束

在创建表的时候, 我们可以给表中的字段加上一些约束, 来保证这个这个表中的数据的完整性,有效性
约束的作用 :是为了保证表中的数据有效

约束包括那些

非空约束 : not null
唯一性约束 : unique
主键约束 : primary key
外键约束 : foreign key
检查约束 : check (mysql不支持 , oracle支持)

唯一性约束

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

表示name和email两个字段联合起来唯一

主键约束primary key

主键约束: 就是一种约束
主键字段: 就是字段上添加了主键约束,
主键值: 主键字段的每一个值都叫做: 主键值

什么是主键? 有什么用?
主键值是每一行记录的唯一标识
主键值是每一行记录的身份证号

任意一张表都应该有主键, 没有主键, 表无效

主键的特征: not null + unique (主键值不能是null,同时也不能重复)

怎么给一张表添加主键约束?

```
drop table if exists t_vip;
create table t_vip (
    id int primary key,
    name varchar(255)
);
```

主键只能有一个

自增主键

```
create table t_vip(
    id int primary key auto_increment,
    //auto_increment表示自增, 从一开始, 以一自增
    name varchar(255)
);
```

外键约束

先创建父表, 再创建子表

```
drop table if exists t_student;
drop table if exists t_class;

create table t_class(
    classno int primary key,
    classname varchar(255)
);

create table t_student(
    no int primary key auto_increament;
    name varchar(255),
    cno int,
    foregin key(con) references t_class(classno)
);
```

存储引擎

存储引擎是一个表存储/组织数据的方式

不同的存储引擎, 表存储数据的方式不同

怎么给表添加/指定"存储引擎"

查看存储引擎

```
show create table t_student;
```

可以在建表的时候给表指定存储引擎

在建表的时候可以在最后小括号的 ") " 的右边使用:

ENGINE 来指定存储引擎

default

CHARSET 来指定这张表的字符编码方式

结论：
mysql 默认的存储引擎是 : InnoDB

```
create table t_product(
   id int primary key,
   name varchar(255)
)engine=InnoDB default charset=utf8;
```

查看当前数据库支持的存储引擎

show engines \G

mysql常用的存储引擎

MyISAM存储引擎?

```
它管理的表具有以下特征：
    使用三个文件表示每个表：
        格式文件 — 存储表结构的定义（mytable.frm）
        数据文件 — 存储表行的内容（mytable.MYD）
        索引文件 — 存储表上索引（mytable.MYI）：索引是一本书的目录，缩小扫描范围，提高查询效率的一种机制。
    可被转换为压缩、只读表来节省空间
```

提示一下：
对于一张表来说，只要是主键，
或者加有unique约束的字段上会自动创建索引。

MyISAM存储引擎特点：
可被转换为压缩、只读表来节省空间
这是这种存储引擎的优势！！！！

MyISAM不支持事务机制，安全性低。

InnoDB存储引擎?

InnoDB存储引擎?
这是mysql默认的存储引擎，同时也是一个重量级的存储引擎。
InnoDB支持事务，支持数据库崩溃后自动恢复机制。
InnoDB存储引擎最主要的特点是：非常安全。

它管理的表具有下列主要特征：
– 每个 InnoDB 表在数据库目录中以.frm 格式文件表示
– InnoDB 表空间 tablespace 被用于存储表的内容（表空间是一个逻辑名称。表空间存储数据+索引。）

– 提供一组用来记录事务性活动的日志文件
– 用 COMMIT(提交)、SAVEPOINT 及ROLLBACK(回滚)支持事务处理
– 提供全 ACID 兼容
– 在 MySQL 服务器崩溃后提供自动恢复
– 多版本（MVCC）和行级锁定
– 支持外键及引用的完整性，包括级联删除和更新

InnoDB最大的特点就是支持事务：
以保证数据的安全。效率不是很高，并且也不能压缩，不能转换为只读，
不能很好的节省存储空间。

MEMORY存储引擎

使用 MEMORY 存储引擎的表，其数据存储在内存中，且行的长度固定，
这两个特点使得 MEMORY 存储引擎非常快。

MEMORY 存储引擎管理的表具有下列特征：
– 在数据库目录内，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。（目的就是快，查询快！）
– 表级锁机制。
– 不能包含 TEXT 或 BLOB 字段。

MEMORY 存储引擎以前被称为HEAP 引擎。

MEMORY引擎优点：查询效率是最高的。不需要和硬盘交互。
MEMORY引擎缺点：不安全，关机之后数据消失。因为数据和索引都是在内存当中。

事务

9.1、什么是事务?

一个事务其实就是一个完整的业务逻辑。
是一个最小的工作单元。不可再分。

```
什么是一个完整的业务逻辑？
假设转账，从A账户向B账户中转账10000.
    将A账户的钱减去10000（update语句）
    将B账户的钱加上10000（update语句）
    这就是一个完整的业务逻辑。

    以上的操作是一个最小的工作单元，要么同时成功，要么同时失败，不可再分。
    这两个update语句要求必须同时成功或者同时失败，这样才能保证钱是正确的。
```

9.2、只有DML语句才会有事务这一说，其它语句和事务无关！！！

insert
delete
update
只有以上的三个语句和事务有关系，其它都没有关系。

```
因为 只有以上的三个语句是数据库表中数据进行增、删、改的。
只要你的操作一旦涉及到数据的增、删、改，那么就一定要考虑安全问题。

数据安全第一位！！！
```

9.3、假设所有的业务，只要一条DML语句就能完成，还有必要存在事务机制吗？

```
正是因为做某件事的时候，需要多条DML语句共同联合起来才能完成，
所以需要事务的存在。如果任何一件复杂的事儿都能一条DML语句搞定，
那么事务则没有存在的价值了。

到底什么是事务呢？
    说到底，说到本质上，一个事务其实就是多条DML语句同时成功，或者同时失败！

事务：就是批量的DML语句同时成功，或者同时失败！
```

事务是怎么做到多条DML语句同时成功和同时失败的呢？

InnoDB存储引擎：提供一组用来记录事务性活动的日志文件

```
事务开启了：
insert
insert
insert
delete
update
update
update
事务结束了！

在事务的执行过程中，每一条DML的操作都会记录到“事务性活动的日志文件”中。
在事务的执行过程中，我们可以提交事务，也可以回滚事务。

提交事务？
    清空事务性活动的日志文件，将数据全部彻底持久化到数据库表中。
    提交事务标志着，事务的结束。并且是一种全部成功的结束。

回滚事务？
    将之前所有的DML操作全部撤销，并且清空事务性活动的日志文件
    回滚事务标志着，事务的结束。并且是一种全部失败的结束。
```

怎么提交事务，怎么回滚事务？

提交事务：commit; 语句

回滚事务：rollback; 语句（回滚永远都是只能回滚到上一次的提交点！）

```
事务对应的英语单词是：transaction

测试一下，在mysql当中默认的事务行为是怎样的？
    mysql默认情况下是支持自动提交事务的。（自动提交）
    什么是自动提交？
        每执行一条DML语句，则提交一次！

    这种自动提交实际上是不符合我们的开发习惯，因为一个业务
    通常是需要多条DML语句共同执行才能完成的，为了保证数据
    的安全，必须要求同时成功之后再提交，所以不能执行一条
    就提交一条。
```

怎么将mysql的自动提交机制关闭掉呢?

先执行这个命令：start transaction;

事务包括4个特性?

原子性 （atomicity）

强调事务的不可分割.
事务的原子性是指事务必须是一个原子的操作序列单元。事务中包含的各项操作在一次执行过程中，只允许出现两种状态之一，要么都成功，要么都失败。
任何一项操作都会导致整个事务的失败，同时其它已经被执行的操作都将被撤销并回滚，只有所有的操作全部成功，整个事务才算是成功完成。

一致性 （consistency）

事务的执行的前后数据的完整性保持一致.

事务的一致性是指事务在执行不能破坏数据库数据的完整性和一致性，一个事务在执行之前和执行之后，数据库都必须处以一致性状态。
比如：张三给李四转钱，不可能张三被扣了钱，李四没有加钱。

隔离性 （isolation）

一个事务执行的过程中,不应该受到其他事务的干扰

事务的隔离性是指在并发环境中，并发的事务是互相隔离的，一个事务的执行不能被其它事务干扰。也就是说，不同事物并非操作相同数据时，每个事务都有完整的数据空间。
一个事务内部的操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务是不能互相干扰的。

持久性

事务一旦结束,数据就持久到数据库

事务的持久性是指事务一旦提交后，数据库中的数据必须被永久的保存下来。即使服务器系统崩溃或服务器宕机等故障。只要数据库重新启动，那么一定能够将其恢复到事务成功结束后的状态。

事务的隔离性

A教室和B教室中间有一道墙，这道墙可以很厚，也可以很薄。这就是事务的隔离级别。
这道墙越厚，表示隔离级别就越高。

事务和事务之间的隔离级别有哪些呢？4个级别

读未提交：read uncommitted

（最低的隔离级别）《没有提交就读到了》
什么是读未提交？
　事务A可以读取到事务B未提交的数据。
这种隔离级别存在的问题就是：
　读现象！(Dirty Read)
我们称读到了脏数据。
　这种隔离级别一般都是理论上的，大多数的数据库隔离级别都是二档起步！

读已提交：read committed

《提交之后才能读到》
什么是读已提交？
　事务A只能读取到事务B提交之后的数据。
这种隔离级别解决了什么问题？
　解决了脏读的现象。
这种隔离级别存在什么问题？
　不可重复读取数据。
什么是不可重复读取数据呢？
　在事务开启之后，第一次读到的数据是3条，当前事务还没有
结束，可能第二次再读取的时候，读到的数据是4条，3不等于4
　称为不可重复读取。

这种隔离级别是比较真实的数据，每一次读到的数据是绝对的真实。
　oracle数据库默认的隔离级别是：read committed

可重复读：repeatable read

《提交之后也读不到，永远读取的都是刚开启事务时的数据》
什么是可重复读取？
　事务A开启之后，不管是多久，每一次在事务A中读取到的数据
都是一致的。即使事务B将数据已经修改，并且提交了，事务A
　读取到的数据还是没有发生改变，这就是可重复读。
可重复读解决了什么问题？
　解决了不可重复读取数据。
可重复读存在的问题是什么？
　可以会出现幻影读。
每一次读取到的数据都是幻象。不够真实！

早晨9点开始开启了事务，只要事务不结束，到晚上9点，读到的数据还是那样！
　读到的是假象。不够绝对的真实。

mysql中默认的事务隔离级别就是这个！！！！！！！！！！

序列化/串行化：serializable

（最高的隔离级别）
这是最高隔离级别，效率最低。解决了所有的问题。
　这种隔离级别表示事务排队，不能并发！
synchronized，线程同步（事务同步）
　每一次读取到的数据都是最真实的，并且效率是最低的。

验证各种隔离级别

查看隔离级别：SELECT @@tx_isolation

被测试的表t_user
验证：read uncommited
mysql> set global transaction isolation level read uncommitted;

验证：read commited
mysql> set global transaction isolation level read committed;

验证：repeatable read
mysql> set global transaction isolation level repeatable read;

验证：serializable
mysql> set global transaction isolation level serializable;

索引（index）

什么是索引?

索引是在数据库表的字段上添加的，是为了提高查询效率存在的一种机制。
　一张表的一个字段可以添加一个索引，当然，多个字段联合起来也可以添加索引。
索引相当于一本书的目录，是为了缩小扫描范围而存在的一种机制。

对于一本字典来说，查找某个汉字有两种方式：
　　第一种方式：一页一页挨着找，直到找到为止，这种查找方式属于全字典扫描。
效率比较低。
　　第二种方式：先通过目录（索引）去定位一个大概的位置，然后直接定位到这个
　　位置，做局域性扫描，缩小扫描的范围，快速的查找。这种查找方式属于通过
索引检索，效率较高。

select * from t_user where name = 'jack';

以上的这条SQL语句会去name字段上扫描，为什么?
　　因为查询条件是：name='jack'

如果name字段上没有添加索引（目录），或者说没有给name字段创建索引，
　MySQL会进行全扫描，会将name字段上的每一个值都比对一遍。效率比较低。

MySQL在查询方面主要就是两种方式：
　　第一种方式：全表扫描
　　第二种方式：根据索引检索。

注意：

在实际中，汉语字典前面的目录是排序的，按照a b c d e f....排序，
为什么排序呢？因为只有排序了才会有区间查找这一说！（缩小扫描范围
其实就是扫描某个区间罢了！）

在mysql数据库当中索引也是需要排序的，并且这个所以的排序和TreeSet
数据结构相同。TreeSet（TreeMap）底层是一个自平衡的二叉树！在mysql
当中索引是一个B-Tree数据结构。

遵循左小又大原则存放。采用中序遍历方式遍历取数据。

索引的实现原理?

提醒1：在任何数据库当中主键上都会自动添加索引对象，id字段上自动有索引，
因为id是PK。另外在mysql当中，一个字段上如果有unique约束的话，也会自动
创建索引对象。

提醒2：在任何数据库当中，任何一张表的任何一条记录在硬盘存储上都有
一个硬盘的物理存储编号。

提醒3：在mysql当中，索引是一个单独的对象，不同的存储引擎以不同的形式
存在，在MyISAM存储引擎中，索引存储在一个.MYI文件中。在InnoDB存储引擎中
索引存储在一个逻辑名称叫做tablespace的当中。在MEMORY存储引擎当中索引
被存储在内存当中。不管索引存储在哪里，索引在mysql当中都是一个树的形式
存在。（自平衡二叉树：B-Tree）

在mysql当中，主键上，以及unique字段上都会自动添加索引的！！！！

什么条件下，我们会考虑给字段添加索引呢?
条件1：数据量庞大（到底有多么庞大算庞大，这个需要测试，因为每一个硬件环境不同）
条件2：该字段经常出现在where的后面，以条件的形式存在，也就是说这个字段总是被扫描。
条件3：该字段很少的DML(insert delete update)操作。（因为DML之后，索引需要重新排序。）

```
建议不要随意添加索引，因为索引也是需要维护的，太多的话反而会降低系统的性能。
建议通过主键查询，建议通过unique约束的字段进行查询，效率是比较高的。
```

索引怎么创建？怎么删除？语法是什么?

创建索引：

```
mysql> create index emp_ename_index on emp(ename);
```

给emp表的ename字段添加索引，起名：emp_ename_index

```
删除索引：
    mysql> drop index emp_ename_index on emp;
    将emp表上的emp_ename_index索引对象删除。
```

在mysql当中，怎么查看一个SQL语句是否使用了索引进行检索?

mysql> explain select * from emp where ename = 'KING';

```
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL |   14 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
```

扫描14条记录：说明没有使用索引。type=ALL

mysql> create index emp_ename_index on emp(ename);

```
mysql> explain select * from emp where ename = 'KING';
+----+-------------+-------+------+-----------------+-----------------+---------+-------+------+-------------+
| id | select_type | table | type | possible_keys   | key             | key_len | ref   | rows | Extra       |
+----+-------------+-------+------+-----------------+-----------------+---------+-------+------+-------------+
|  1 | SIMPLE      | emp   | ref  | emp_ename_index | emp_ename_index | 33      | const |    1 | Using where |
+----+-------------+-------+------+-----------------+-----------------+---------+-------+------+-------------+
```

索引有失效的时候，什么时候索引失效呢?

失效的第1种情况：

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

```
    ename上即使添加了索引，也不会走索引，为什么？
        原因是因为模糊匹配当中以“%”开头了！
        尽量避免模糊查询的时候以“%”开始。
        这是一种优化的手段/策略。

    mysql> explain select * from emp where ename like '%T';
    +----+-------------+-------+------+---------------+------+---------+------+------+-------------+
    | id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
    +----+-------------+-------+------+---------------+------+---------+------+------+-------------+
    |  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL |   14 | Using where |
    +----+-------------+-------+------+---------------+------+---------+------+------+-------------+
```

失效的第2种情况：

使用or的时候会失效，如果使用or那么要求or两边的条件字段都要有索引，才会走索引，如果其中一边有一个字段没有索引，那么另一个字段上的索引也会实现。所以这就是为什么不建议使用or的原因。

```
    mysql> explain select * from emp where ename = 'KING' or job = 'MANAGER';
    +----+-------------+-------+------+-----------------+------+---------+------+------+-------------+
```

```
    | id | select_type | table | type | possible_keys   | key  | key_len | ref  | rows | Extra       |
    +----+-------------+-------+------+-----------------+------+---------+------+------+-------------+
    |  1 | SIMPLE      | emp   | ALL  | emp_ename_index | NULL | NULL    | NULL |   14 | Using where |
    +----+-------------+-------+------+-----------------+------+---------+------+------+-------------+
```

失效的第3种情况：

使用复合索引的时候，没有使用左侧的列查找，索引失效

什么是复合索引？

两个字段，或者更多的字段联合起来添加一个索引，叫做复合索引。

```
create index emp_job_sal_index on emp(job,sal);

mysql> explain select * from emp where job = 'MANAGER';
+----+-------------+-------+------+-------------------+-------------------+---------+-------+------+-------------+
| id | select_type | table | type | possible_keys     | key               | key_len | ref   | rows | Extra       |
+----+-------------+-------+------+-------------------+-------------------+---------+-------+------+-------------+
|  1 | SIMPLE      | emp   | ref  | emp_job_sal_index | emp_job_sal_index | 30      | const |    3 | Using where |
+----+-------------+-------+------+-------------------+-------------------+---------+-------+------+-------------+

mysql> explain select * from emp where sal = 800;
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL |   14 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
```

失效的第4种情况：

在where当中索引列参加了运算，索引失效。

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

explain select * from emp where sal = 800;
+----+-------------+-------+------+---------------+---------------+---------+-------+------+-------------+
| id | select_type | table | type | possible_keys | key           | key_len | ref   | rows | Extra       |
+----+-------------+-------+------+---------------+---------------+---------+-------+------+-------------+
|  1 | SIMPLE      | emp   | ref  | emp_sal_index | emp_sal_index | 9       | const |    1 | Using where |
+----+-------------+-------+------+---------------+---------------+---------+-------+------+-------------+
```

```
mysql> explain select * from emp where sal+1 = 800;
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL |   14 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
```

失效的第5种情况：

在where当中索引列使用了函数

```
explain select * from emp where lower(ename) = 'smith';
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL |   14 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
```

索引是各种数据库进行优化的重要手段。

优化的时候优先考虑的因素就是索引。

索引在数据库当中分了很多类?
单一索引：一个字段上添加索引。
复合索引：两个字段或者更多的字段上添加索引。

主键索引：主键上添加索引。
唯一性索引：具有unique约束的字段上添加索引。
.....

注意：唯一性比较弱的字段上添加索引用处不大。

视图(view)

什么是视图?

view:站在不同的角度去看待同一份数据。

怎么创建视图对象？怎么删除视图对象?

表复制：

```
mysql> create table dept2 as select * from dept;
```

```
dept2表中的数据：
mysql> select * from dept2;
```

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

创建视图对象：

```
    create view dept2_view as select * from dept2;
```

删除视图对象：

```
    drop view dept2_view;
```

注意：只有DQL语句才能以view的形式创建。

```
    create view view_name as 这里的语句必须是DQL语句;
```

用视图做什么?

我们可以面向视图对象进行增删改查，对视图对象的增删改查，会导致原表被操作！（视图的特点：通过对视图的操作，会影响到原表数据。）

```
//面向视图查询
select * from dept2_view;

// 面向视图插入
insert into dept2_view(deptno,dname,loc) values(60,'SALES', 'BEIJING');

// 查询原表数据
mysql> select * from dept2;
+--------+------------+----------+
| DEPTNO | DNAME      | LOC      |
+--------+------------+----------+
|     10 | ACCOUNTING | NEW YORK |
|     20 | RESEARCH   | DALLAS   |
|     30 | SALES      | CHICAGO  |
|     40 | OPERATIONS | BOSTON   |
|     60 | SALES      | BEIJING  |
+--------+------------+----------+

// 面向视图删除
mysql> delete from dept2_view;

// 查询原表数据
mysql> select * from dept2;
Empty set (0.00 sec)
```

```
// 创建视图对象
create view
    emp_dept_view
as
    select
        e.ename,e.sal,d.dname
```

```
    from
        emp e
    join
        dept d
    on
        e.deptno = d.deptno;

// 查询视图对象
mysql> select * from emp_dept_view;
+--------+---------+------------+
| ename  | sal     | dname      |
+--------+---------+------------+
| CLARK  | 2450.00 | ACCOUNTING |
| KING   | 5000.00 | ACCOUNTING |
| MILLER | 1300.00 | ACCOUNTING |
| SMITH  |  800.00 | RESEARCH   |
| JONES  | 2975.00 | RESEARCH   |
| SCOTT  | 3000.00 | RESEARCH   |
| ADAMS  | 1100.00 | RESEARCH   |
| FORD   | 3000.00 | RESEARCH   |
| ALLEN  | 1600.00 | SALES      |
| WARD   | 1250.00 | SALES      |
| MARTIN | 1250.00 | SALES      |
| BLAKE  | 2850.00 | SALES      |
| TURNER | 1500.00 | SALES      |
| JAMES  |  950.00 | SALES      |
+--------+---------+------------+

// 面向视图更新
update emp_dept_view set sal = 1000 where dname = 'ACCOUNTING';

// 原表数据被更新
mysql> select * from emp;
+-------+--------+-----------+------+------------+---------+---------+------
--+
| 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 | 1000.00 |    NULL |
10 |
|  7788 | SCOTT  | ANALYST   | 7566 | 1987-04-19 | 3000.00 |    NULL |
20 |
```

```
|  7839 | KING   | PRESIDENT | NULL | 1981-11-17 | 1000.00 |    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 | 1000.00 |    NULL |
10 |
+-------+--------+-----------+------+------------+---------+---------+------
--+
```

视图对象在实际开发中到底有什么用?

《方便，简化开发，利于维护》

```
create view
        emp_dept_view
    as
        select
            e.ename,e.sal,d.dname
        from
            emp e
        join
            dept d
        on
            e.deptno = d.deptno;
```

假设有一条非常复杂的SQL语句，而这条SQL语句需要在不同的位置上反复使用。
每一次使用这个sql语句的时候都需要重新编写，很长，很麻烦，怎么办?
可以把这条复杂的SQL语句以视图对象的形式新建。
在需要编写这条SQL语句的位置直接使用视图对象，可以大大简化开发。
并且利于后期的维护，因为修改的时候也只需要修改一个位置就行，只需要修改视图对象所映射的SQL语句。

我们以后面向视图开发的时候，使用视图的时候可以像使用table一样。
可以对视图进行增删改查等操作。视图不是在内存当中，视图对象也是存储在硬盘上的，不会消失。

```
再提醒一下：
    视图对应的语句只能是DQL语句。
    但是视图对象创建完成之后，可以对视图进行增删改查等操作。

小插曲：
    增删改查，又叫做：CRUD。
    CRUD是在公司中程序员之间沟通的术语。一般我们很少说增删改查。
    一般都说CRUD。

    C:Create（增）
    R:Retrive（查：检索）
    U:Update（改）
    D:Delete（删）
```

DBA常用命令?

重点掌握：

数据的导入和导出（数据的备份）

其它命令了解一下即可。（这个培训日志文档留着，以后忘了，可以打开文档复制粘贴。）

```
数据导出？
    注意：在windows的dos命令窗口中：
        mysqldump bjpowernode>D:\bjpowernode.sql -uroot -p123456

    可以导出指定的表吗？
        mysqldump bjpowernode emp>D:\bjpowernode.sql -uroot -p123456

数据导入？
    注意：需要先登录到mysql数据库服务器上。
    然后创建数据库：create database bjpowernode;
    使用数据库：use bjpowernode
    然后初始化数据库：source D:\bjpowernode.sql
```

数据库设计三范式

什么是数据库设计范式?

数据库表的设计依据。教你怎么进行数据库表的设计。

数据库设计范式共有?

3个。

第一范式：要求任何一张表必须有主键，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，要求所有非主键字段完全依赖主键，
不要产生部分依赖。

第三范式：建立在第二范式的基础之上，要求所有非主键字段直接依赖主键，
不要产生传递依赖。

声明：三范式是面试官经常问的，所以一定要熟记在心！

设计数据库表的时候，按照以上的范式进行，可以避免表中数据的冗余，空间的浪费。
```

第一范式

最核心，最重要的范式，所有表的设计都需要满足。
必须有主键，并且每一个字段都是原子性不可再分。

```
学生编号 学生姓名 联系方式
-----------------------------------------
1001         张三       zs@gmail.com,1359999999
1002         李四       ls@gmail.com,13699999999
1001         王五       ww@163.net,13488888888

以上是学生表，满足第一范式吗？
     不满足，第一：没有主键。第二：联系方式可以分为邮箱地址和电话

学生编号(pk) 学生姓名    邮箱地址             联系电话
-----------------------------------------------------
1001                张三       zs@gmail.com    1359999999
1002                李四       ls@gmail.com    13699999999
1003                王五       ww@163.net
```

第二范式：

建立在第一范式的基础之上，
要求所有非主键字段必须完全依赖主键，不要产生部分依赖。

```
学生编号 学生姓名 教师编号 教师姓名
-----------------------------------------------------
1001             张三       001      王老师
1002             李四       002      赵老师
1003             王五       001      王老师
1001             张三       002      赵老师

这张表描述了学生和老师的关系：（1个学生可能有多个老师，1个老师有多个学生）
这是非常典型的：多对多关系！

分析以上的表是否满足第一范式？
     不满足第一范式。

怎么满足第一范式呢？修改

学生编号+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

学生编号 教师编号，两个字段联合做主键，复合主键（PK: 学生编号+教师编号）
经过修改之后，以上的表满足了第一范式。但是满足第二范式吗？
    不满足，“张三”依赖1001，“王老师”依赖001，显然产生了部分依赖。
    产生部分依赖有什么缺点？
        数据冗余了。空间浪费了。“张三”重复了，“王老师”重复了。

为了让以上的表满足第二范式，你需要这样设计：
    使用三张表来表示多对多的关系！！！！
    学生表
    学生编号(pk)          学生名字
    ------------------------------------
    1001                    张三
    1002                    李四
    1003                    王五

    教师表
    教师编号(pk)          教师姓名
    ---------------------------------------
    001                   王老师
    002                   赵老师

    学生教师关系表
    id(pk)          学生编号(fk)            教师编号(fk)
    ------------------------------------------------------
    1                       1001                        001
    2                       1002                        002
    3                       1003                        001
    4                       1001                        002
```

```
背口诀：
    多对多怎么设计？
        多对多，三张表，关系表两个外键！！！！！！！！！！！！！！
```

第三范式

第三范式建立在第二范式的基础之上

要求所有非主键字典必须直接依赖主键，不要产生传递依赖。

```
学生编号（PK） 学生姓名 班级编号   班级名称
-----------------------------------------------------------
    1001                张三        01          一年一班
    1002                李四        02          一年二班
    1003                王五        03          一年三班
    1004                赵六        03          一年三班

以上表的设计是描述：班级和学生的关系。很显然是1对多关系！
一个教室中有多个学生。

分析以上表是否满足第一范式？
    满足第一范式，有主键。
```

```
分析以上表是否满足第二范式？
    满足第二范式，因为主键不是复合主键，没有产生部分依赖。主键是单一主键。

分析以上表是否满足第三范式？
    第三范式要求：不要产生传递依赖！
    一年一班依赖01，01依赖1001，产生了传递依赖。
    不符合第三范式的要求。产生了数据的冗余。

那么应该怎么设计一对多呢？

    班级表：一
    班级编号(pk)                    班级名称
    ----------------------------------------
    01                                  一年一班
    02                                  一年二班
    03                                  一年三班

    学生表：多

    学生编号（PK） 学生姓名 班级编号(fk)
    ----------------------------------------
    1001                张三            01
    1002                李四            02
    1003                王五            03
    1004                赵六            03

    背口诀：
        一对多，
```

总结表的设计?

一对多:

一对多，两张表，多的表加外键！！！！！！！！！！！！

```
多对多：
    多对多，三张表，关系表两个外键！！！！！！！！！！！！！！！

一对一：
    一对一放到一张表中不就行了吗？为啥还要拆分表？
    在实际的开发中，可能存在一张表字段太多，太庞大。这个时候要拆分表。
    一对一怎么设计？
        没有拆分表之前：一张表
            t_user
            id      login_name      login_pwd       real_name       email       address.....
            ----------------------------------------------------------------------------
            1           zhangsan        123             张三            zhangsan@xxx
            2           lisi            123             李四            lisi@xxx
            ...

        这种庞大的表建议拆分为两张：
```

```
            t_login 登录信息表
            id(pk)      login_name      login_pwd
            ---------------------------------
            1               zhangsan        123
            2               lisi            123

            t_user 用户详细信息表
            id(pk)      real_name       email               address........
login_id(fk+unique)
            ------------------------------------------------------------------
-------------------------
            100         张三              zhangsan@xxx
1
            200         李四              lisi@xxx
2
```

```
            口诀：一对一，外键唯一！！！！！！！！！！
```

嘱咐一句话：

数据库设计三范式是理论上的。

```
实践和理论有的时候有偏差。

最终的目的都是为了满足客户的需求，有的时候会拿冗余换执行速度。

因为在sql当中，表和表之间连接次数越多，效率越低。（笛卡尔积）

有的时候可能会存在冗余，但是为了减少表的连接次数，这样做也是合理的，
并且对于开发人员来说，sql语句的编写难度也会降低。

面试的时候把这句话说上：他就不会认为你是初级程序员了！
```