

# 书写高质量SQL的30条建议

---

## 1、查询SQL尽量不要使用select \*, 而是select具体字段。

反例子：

```
1 select * from employee;
```

正例子：

```
1 select id, name from employee;
```

理由：

- 只取需要的字段，节省资源、减少网络开销。
- select \* 进行查询时，很可能就不会使用到覆盖索引了，就会造成回表查询。

## 2、如果知道查询结果只有一条或者只要最大/最小一条记录，建议用limit 1

假设现在有employee员工表，要找出一个名字叫jay的人。

```
1 CREATE TABLE `employee` (  
2   `id` int(11) NOT NULL,  
3   `name` varchar(255) DEFAULT NULL,  
4   `age` int(11) DEFAULT NULL,  
5   `date` datetime DEFAULT NULL,  
6   `sex` int(1) DEFAULT NULL,  
7   PRIMARY KEY (`id`)  
8 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

反例：

```
1 select id, name from employee where name='jay'
```

正例

```
1 select id, name from employee where name='jay' limit 1;
```

理由：

- 加上limit 1后,只要找到了对应的一条记录,就不会继续向下扫描了,效率将会大大提高。
- 当然，如果name是唯一索引的话，是不必要加上limit 1了，因为limit的存在主要就是为了防止全表扫描，从而提高性能,如果一个语句本身可以预知不用全表扫描，有没有limit，性能的差别并不大。

### 3、应尽量避免在where子句中使用or来连接条件

新建一个user表，它有一个普通索引userId，表结构如下：

```
1 CREATE TABLE `user` (  
2   `id` int(11) NOT NULL AUTO_INCREMENT,  
3   `userId` int(11) NOT NULL,  
4   `age` int(11) NOT NULL,  
5   `name` varchar(255) NOT NULL,  
6   PRIMARY KEY (`id`),  
7   KEY `idx_userId` (`userId`)  
8 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

假设现在需要查询userid为1或者年龄为18岁的用户，很容易有以下sql

反例:

```
1 select * from user where userid=1 or age =18
```

正例:

```
1 //使用union all  
2 select * from user where userid=1  
3 union all  
4 select * from user where age = 18  
5  
6 //或者分开两条sql写:  
7 select * from user where userid=1  
8 select * from user where age = 18
```

理由:

- 使用or可能会使索引失效，从而全表扫描。

对于or+没有索引的age这种情况，假设它走了userId的索引，但是走到age查询条件时，它还得全表扫描，也就是需要三步过程：全表扫描+索引扫描+合并 如果它一开始就走全表扫描，直接一遍扫描就完事。mysql是有优化器的，处于效率与成本考虑，遇到or条件，索引可能失效，看起来也合情合理。

### 4、优化limit分页

我们日常做分页需求时，一般会用 limit 实现，但是当偏移量特别大的时候，查询效率就变得低下。

反例:

```
1 select id, name, age from employee limit 10000, 10
```

正例:

```

1 //方案一：返回上次查询的最大记录(偏移量)
2 select id, name from employee where id>10000 limit 10.
3
4 //方案二: order by + 索引
5 select id, name from employee order by id limit 10000, 10
6
7 //方案三: 在业务允许的情况下限制页数:

```

理由:

- 当偏移量最大的时候，查询效率就会越低，因为Mysql并非是跳过偏移量直接去取后面的数据，而是先把偏移量+要取的条数，然后再把前面偏移量这一段的数据抛弃掉再返回的。
- 如果使用优化方案一，返回上次最大查询记录（偏移量），这样可以跳过偏移量，效率提升不少。
- 方案二使用order by+索引，也是可以提高查询效率的。
- 方案三的话，建议跟业务讨论，有没有必要查这么后的分页啦。因为绝大多数用户都不会往后翻太多页。

## 5、优化你的like语句

日常开发中，如果用到模糊关键字查询，很容易想到like，但是like很可能让你的索引失效。

反例:

```

1 select userId, name from user where userId like '%123';

```

正例:

```

1 select userId, name from user where userId like '123%';

```

理由:

- 把%放前面，并不走索引，如下:

```

1 explain select * from user where userid like '%123';

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	1	100	Using where

- 把%放关键字后面，还是会走索引的。如下:

```

1 explain select * from user where userid like '123%';

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	range	idx_userid	idx_userid	98	(Null)	1	100	Using index condition

## 6、使用where条件限定要查询的数据，避免返回多余的行

假设业务场景是这样：查询某个用户是否是会员。曾经看过老的实现代码是这样。。。

反例：

```
1 List<Long> userIds = sqlMap.queryList("select userId from user where isVip=1");
2 boolean isVip = userIds.contains(userId);
```

正例：

```
1 Long userId = sqlMap.queryObject("select userId from user where userId='userId' and
  isVip='1' ")
2 boolean isVip = userId != null;
```

理由：

- 需要什么数据，就去查什么数据，避免返回不必要的数据，节省开销。

## 7、尽量避免在索引列上使用mysql的内置函数

业务需求：查询最近七天内登陆过的用户(假设loginTime加了索引)

反例：

```
1 select userId,loginTime from loginuser where Date_ADD(loginTime,Interval 7 DAY)
  >=now();
```

正例：

```
1 explain select userId,loginTime from loginuser where loginTime >=
  Date_ADD(NOW(),INTERVAL - 7 DAY);
```

理由：

- 索引列上使用mysql的内置函数，索引失效

1 explain select userId,loginTime from loginuser where Date\_ADD(loginTime,Interval 7 DAY) >=now();

信息	结果 1	剖析	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	loginuser	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	1	100	Using where

img

- 如果索引列不加内置函数，索引还是会走的。

```
1 explain select userId,loginTime from loginuser where loginTime >= Date_ADD(NOW(),INTERVAL - 7 DAY);
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	loginuser	(Null)	range	idx_login_time	idx_log5		(Null)	1	100	Using index cor

## 8、应尽量避免在 where 子句中对字段进行表达式操作，这将导致系统放弃使用索引而进行全表扫

反例：

```
1 select * from user where age-1 =10;
```

正例：

```
1 select * from user where age =11;
```

理由：

- 虽然age加了索引，但是因为对它进行运算，索引直接迷路了。。。

```
1 explain select * from user where age-1=10;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	1	100	Using where

## 9、Inner join、left join、right join，优先使用Inner join，如果是left join，左边表结果尽量小

- Inner join 内连接，在两张表进行连接查询时，只保留两张表中完全匹配的结果集
- left join 在两张表进行连接查询时，会返回左表所有的行，即使在右表中没有匹配的记录。
- right join 在两张表进行连接查询时，会返回右表所有的行，即使在左表中没有匹配的记录。

都满足SQL需求的前提下，推荐优先使用Inner join（内连接），如果要使用left join，左边表数据结果尽量小，如果有条件的尽量放到左边处理。

反例：

```
1 select * from tab1 t1 left join tab2 t2 on t1.size = t2.size where t1.id>2;
```

正例：

```
1 select * from (select * from tab1 where id >2) t1 left join tab2 t2 on t1.size = t2.size;
```

理由：

- 如果inner join是等值连接，或许返回的行数比较少，所以性能相对会好一点。
- 同理，使用了左连接，左边表数据结果尽量小，条件尽量放到左边处理，意味着返回的行数可能比较少。

## 10、应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

反例：

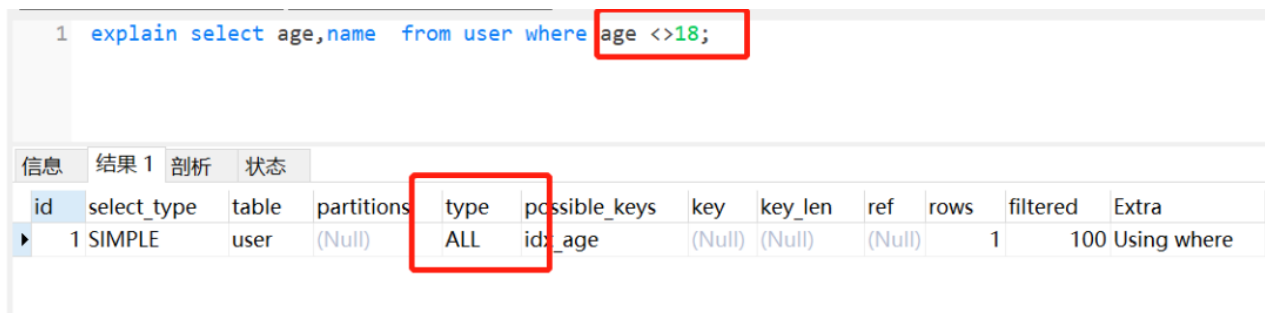
```
1 select age,name from user where age <>18;
```

正例：

```
1 //可以考虑分开两条sql写
2 select age,name from user where age <18;
3 select age,name from user where age >18;
```

理由：

- 使用!=和<>很可能会让索引失效



The screenshot shows the SQL query: `1 explain select age,name from user where age <>18;`. Below the query, the EXPLAIN output is displayed in a table format. The 'type' column shows 'ALL', indicating a full table scan. The 'possible\_keys' column shows 'idx\_age', which is not being used. The 'Extra' column shows '100 Using where', indicating that 100 rows were filtered out by the where clause.

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ALL	idx_age	(Null)	(Null)	(Null)	1	100	Using where

img

## 11、使用联合索引时，注意索引列的顺序，一般遵循最左匹配原则。

表结构：（有一个联合索引idx\_userid\_age，userId在前，age在后）

```
1 CREATE TABLE `user` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
3   `userId` int(11) NOT NULL,
4   `age` int(11) DEFAULT NULL,
5   `name` varchar(255) NOT NULL,
6   PRIMARY KEY (`id`),
7   KEY `idx_userid_age` (`userId`,`age`) USING BTREE
8 ) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

反例：

```
1 select * from user where age = 10;
```

```
1 explain select * from user where age =10 ;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	1	100	Using where

img

正例：

```
1 //符合最左匹配原则
2 select * from user where userid=10 and age =10;
3 //符合最左匹配原则
4 select * from user where userid =10;
```

```
1 explain select * from user where userid =10 and age=10
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ref	idx_userid_age	idx_userid_age	9	const,	1	100	(Null)

```
1 explain select * from user where userid =10 ;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ref	idx_userid_age	idx_userid_age	4	const	1	100	(Null)

理由：

- 当我们创建一个联合索引的时候，如(k1,k2,k3)，相当于创建了（k1）、(k1,k2)和(k1,k2,k3)三个索引，这就是最左匹配原则。
- 联合索引不满足最左原则，索引一般会失效，但是这个还跟Mysql优化器有关的。

**12、对查询进行优化，应考虑在 where 及 order by 涉及的列上建立索引，尽量避免全表扫描。**

反例：

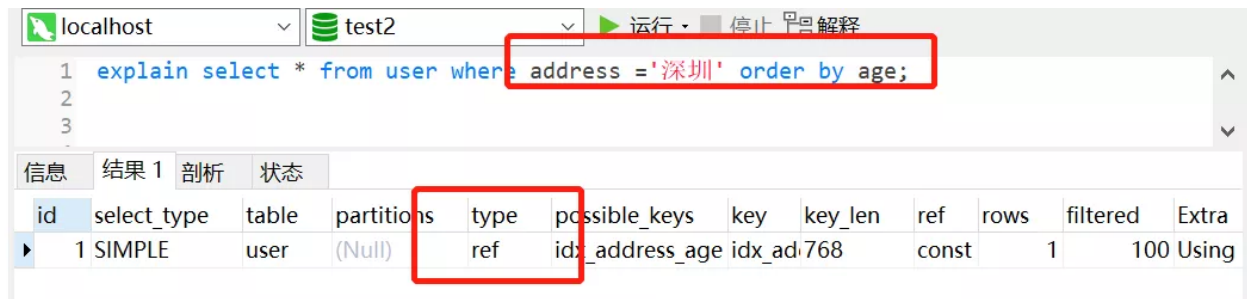
```
1 select * from user where address = '深圳' order by age ;
```

```
1 explain select * from user where address = '深圳' order by age;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	1	100	Using

正例：

```
1 添加索引
2 alter table user add index idx_address_age (address,age)
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ref	idx_address_age	idx_ad	768	const	1	100	Using

img

### 13、如果插入数据过多，考虑批量插入。

反例：

```
1 for(User u :list){
2     INSERT into user(name,age) values(#name#,#age#)
3 }
```

正例：

```
1 //一次500批量插入，分批进行
2 insert into user(name,age) values
3 <foreach collection="list" item="item" index="index" separator=",">
4     ({item.name},{item.age})
5 </foreach>
```

理由：

- 批量插入性能好，更加省时间

打个比喻:假如你需要搬一万块砖到楼顶,你有一个电梯,电梯一次可以放适量的砖（最多放500）,你可以选择一次运送一块砖,也可以一次运送500,你觉得哪个时间消耗大？

### 14、在适当的时候，使用覆盖索引。

覆盖索引能够使得你的SQL语句不需要回表，仅仅访问索引就能够得到所有需要的数据，大大提高了查询效率。

反例：

```
1 // like模糊查询，不走索引了
2 select * from user where userid like '%123%'
```



```
1 explain select * from user where userid like '%123%'
```

信息	结果 1	剖析	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	1	100	Using where

正例：

```
1 //id为主键，那么为普通索引，即覆盖索引登场了。
2 select id,name from user where userid like '%123%';
```

```
1 explain select userid,id from user where userid like '%123%';
```

信息	结果 1	剖析	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	index	(Null)	idx_userid	98	(Null)	1	100	Using where; Using index

img

## 15、慎用distinct关键字

distinct 关键字一般用来过滤重复记录，以返回不重复的记录。在查询一个字段或者很少字段的情况下使用时，给查询带来优化效果。但是在字段很多的时候使用，却会大大降低查询效率。

反例：

```
1 SELECT DISTINCT * from user;
```

正例：

```
1 select DISTINCT name from user;
```

理由：

- 带distinct的语句cpu时间和占用时间都高于不带distinct的语句。因为当查询很多字段时，如果使用distinct，数据库引擎就会对数据进行比较，过滤掉重复数据，然而这个比较，过滤的过程会占用系统资源，cpu时间。

## 16、删除冗余和重复索引

反例：

```
1 KEY `idx_userId` (`userId`)
2 KEY `idx_userId_age` (`userId`,`age`)
```

正例：

```

1 //删除userId索引，因为组合索引（A，B）相当于创建了（A）和（A，B）索引
2 KEY `idx_userId_age` (`userId`,`age`)

```

理由：

- 重复的索引需要维护，并且优化器在优化查询的时候也需要逐个地进行考虑，这会影响性能的。

## 17、如果数据量较大，优化你的修改/删除语句。

避免同时修改或删除过多数据，因为会造成cpu利用率过高，从而影响别人对数据库的访问。

反例：

```

1 //一次删除10万或者100万+?
2 delete from user where id <100000;
3 //或者采用单一循环操作，效率低，时间漫长
4 for (User user: list) {
5     delete from user;
6 }

```

正例：

```

1 //分批进行删除，如每次500
2 delete user where id<500
3 delete product where id>=500 and id<1000;

```

理由：

- 一次性删除太多数据，可能会有lock wait timeout exceed的错误，所以建议分批操作。

## 18、where子句中考虑使用默认值代替null。

反例：

```

1 select * from user where age is not null;

```

```

1
2 explain select * from user where age is not null ;
3

```

信息	结果 1	剖析	状态									
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	user	(Null)	ALL	idx_age	(Null)	(Null)	(Null)	1	100	Using where	

正例：

```

1 //设置0为默认值
2 select * from user where age>0;

```

```

1
2 explain select * from user where age > 0 ;
3

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	range	idx_age	idx_age	4	(Null)	1	100	Using index cond

理由：

- 并不是说使用了is null 或者 is not null 就会不走索引了，这个跟mysql版本以及查询成本都有关。

如果mysql优化器发现，走索引比不走索引成本还要高，肯定会放弃索引，这些条件 `!=`, `>is null`, `is not null` 经常被认为让索引失效，其实是因为一般情况下，查询的成本高，优化器自动放弃的。

- 如果把null值，换成默认值，很多时候让走索引成为可能，同时，表达意思会相对清晰一点。

## 19、不要有超过5个以上的表连接

- 连表越多，编译的时间和开销也就越大。
- 把连接表拆开成较小的几个执行，可读性更高。
- 如果一定需要连接很多表才能得到数据，那么意味着糟糕的设计了。

## 20、exist & in的合理利用

假设表A表示某企业的员工表，表B表示部门表，查询所有部门的所有员工，很容易有以下SQL:

```
1 select * from A where deptId in (select deptId from B);
```

这样写等价于：

先查询部门表B

```
select deptId from B
```

再由部门deptId，查询A的员工

```
select * from A where A.deptId = B.deptId
```

可以抽象成这样的循环：

```

1 List<> resultSet ;
2 for(int i=0;i<B.length;i++) {
3     for(int j=0;j<A.length;j++) {
4         if(A[i].id==B[j].id) {
5             resultSet.add(A[i]);
6             break;
7         }
8     }
9 }

```

显然，除了使用in，我们也可以用exists实现一样的查询功能，如下：

```
1 select * from A where exists (select 1 from B where A.deptId = B.deptId);
```

因为exists查询的理解就是，先执行主查询，获得数据后，再放到子查询中做条件验证，根据验证结果（true或者false），来决定主查询的数据结果是否得意保留。

那么，这样写就等价于：

```
select * from A,先从A表做循环
```

```
select * from B where A.deptId = B.deptId,再从B表做循环.
```

同理，可以抽象成这样一个循环：

```
1 List<> resultSet ;
2 for(int i=0;i<A.length;i++) {
3     for(int j=0;j<B.length;j++) {
4         if(A[i].deptId==B[j].deptId) {
5             resultSet.add(A[i]);
6             break;
7         }
8     }
9 }
```

数据库最费劲的就是跟程序链接释放。假设链接了两次，每次做上百万次的数据集查询，查完就走，这样就只做了两次；相反建立了上百万次链接，申请链接释放反复重复，这样系统就受不了了。即mysql优化原则，就是小表驱动大表，小的数据集驱动大的数据集，从而让性能更优。

因此，我们要选择最外层循环小的，也就是，如果B的数据量小于A，适合使用in，如果B的数据量大于A，即适合选择exist。

## 21、尽量用 union all 替换 union

如果检索结果中不会有重复的记录，推荐union all 替换 union。

反例：

```
1 select * from user where userid=1
2 union
3 select * from user where age = 10
```

正例：

```
1 select * from user where userid=1
2 union all
3 select * from user where age = 10
```

理由：

- 如果使用union，不管检索结果有没有重复，都会尝试进行合并，然后在输出最终结果前进行排序。如果已知检索结果没有重复记录，使用union all代替union，这样会提高效率。

## 22、索引不宜太多，一般5个以内。

- 索引并不是越多越好，索引虽然提高了查询的效率，但是也降低了插入和更新的效率。
- insert或update时有可能会重建索引，所以建索引需要慎重考虑，视具体情况而定。
- 一个表的索引数最好不要超过5个，若太多需要考虑一些索引是否没有存在的必要。

## 23、尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型

反例：

```
1 king_id` varchar (20) NOT NULL COMMENT '守护者Id'
```

正例：

```
1 `king_id` int(11) NOT NULL COMMENT '守护者Id'`
```

理由：

- 相对于数字型字段，字符型会降低查询和连接的性能，并会增加存储开销。

## 24、索引不适合建在有大量重复数据的字段上，如性别这类型数据库字段。

因为SQL优化器是根据表中数据量来进行查询优化的，如果索引列有大量重复数据，Mysql查询优化器推算发现不走索引的成本更低，很可能就放弃索引了。

## 25、尽量避免向客户端返回过多数据量。

假设业务需求是，用户请求查看自己最近一年观看过的直播数据。

反例：

```
1 //一次性查询所有数据回来
2 select * from LivingInfo where watchId =userId and watchTime >=
  Date_sub(now(),Interval 1 Y)
```

正例：

```
1 //分页查询
2 select * from LivingInfo where watchId =userId and watchTime>=
  Date_sub(now(),Interval 1 Y) limit offset, pageSize
3
4 //如果是前端分页，可以先查询前两百条记录，因为一般用户应该也不会往下翻太多页，
5 select * from LivingInfo where watchId =userId and watchTime>=
  Date_sub(now(),Interval 1 Y) limit 200 ;
```

26、当在SQL语句中连接多个表时,请使用表的别名,并把别名前缀于每一列上,这样语义更加清晰。

反例:

```
1 select * from A inner
2 join B on A.deptId = B.deptId;
```

正例:

```
1 select member.name,deptment.deptName from A member inner
2 join B deptment on member.deptId = deptment.deptId;
```

27、尽可能使用varchar/nvarchar代替 char/nchar。

反例:

```
1 `deptName` char(100) DEFAULT NULL COMMENT '部门名称'
```

正例:

```
1 `deptName` varchar(100) DEFAULT NULL COMMENT '部门名称'
```

理由:

- 因为首先变长字段存储空间小,可以节省存储空间。
- 其次对于查询来说,在一个相对较小的字段内搜索,效率更高。

28、为了提高group by 语句的效率,可以在执行到该语句前,把不需要的记录过滤掉。

反例:

```
1 select job, avg (salary) from employee group by job having job ='president'
2 or job = 'managent'
```

正例:

```
1 select job, avg (salary) from employee where job ='president'
2 or job = 'managent' group by job;
```

29、如何字段类型是字符串, where时一定用引号括起来,否则索引失效

反例:

```
1 select * from user where userid =123;
```

```
1 explain select * from user where userid=123
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ALL	idx_userid	(Null)	(Null)	(Null)	1	100	Using where

正例：

```
1 select * from user where userid = '123';
```

```
1 explain select * from user where userid='123';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ref	idx_userid	idx_userid	98	const	1	100	(Null)

理由：

- 为什么第一条语句未加单引号就不走索引了呢？这是因为不加单引号时，是字符串跟数字的比较，它们类型不匹配，MySQL会做隐式的类型转换，把它们转换为浮点数再做比较。

### 30、使用explain 分析你SQL的计划

日常开发写SQL的时候，尽量养成一个习惯吧。用explain分析一下你写的SQL，尤其是走不走索引这一块。

```
1 explain select * from user where userid =10086 or age =18;
```

```
1 explain select * from user where userid =10086 or age =18;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	user	(Null)	ALL	idx_age	(Null)	(Null)	(Null)	1	100	Using where