通过索引进行优化

想要了解索引的优化方式，必须要对索引的底层原理有所了解

hash 是等值查询，需要根据指定的index 才能拿到 index对应的链表

存储引擎是memory的时候，数据结构是 hash

MyISAM和InnoDB存储索引的数据结构是B+树

Mysql 在读取数据的时候有一个磁盘预读的概念，是一个页的整数倍，一次默认读取16K（1k是1024字节），相等于4页（操作系统中的4k对齐 一页）

mysql索引是b+树实现的

b树——>b+树（所有数据放在叶子节点）

▼ **索引基本知识**

▼ 索引的优点

a、大大减少了服务器需要扫描的数据量

b、帮助服务器避免排序和临时表（索引已经排好序了）

c、将随机io变成顺序io

▼ 索引的用处

1、快速查找匹配WHERE子句的行

2、从consideration中消除行,如果可以在多个索引之间进行选择，mysql通常会使用找到最少行的索引

3、如果表具有多列索引，则优化器可以使用索引的任何最左前缀来查找行

4、当有表连接的时候，从其他表检索行数据

5、查找特定索引列的min或max值

6、如果排序或分组时在可用索引的最左前缀上完成的，则对表进行排序和分组

7、在某些情况下，可以优化查询以检索值而无需查询数据行

▼ 索引的分类

聚集索引：其叶子节点存储的是记录（只能有一个，一般是主键索引）

普通索引：其叶子节点存储的是主键值

1、主键索引

2、唯一索引

主键索引和唯一索引的效率差不多，唯一索引值可以为空，主键索引值不能为空，空值得比较会慢一点

3、普通索引

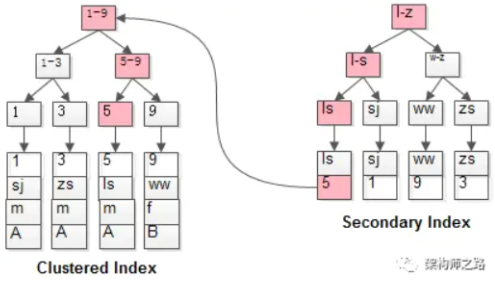
4、全文索引（一般是针对varchar或者text字段的）

5、组合索引

▼ 面试技术名词

1、回表（普通索引）才会有回表的情况

根据普通索引找到子节点中存储的主键，然后根据这个主键再到聚集索引定位记录



2、覆盖索引

MySQL官网，类似的说法出现在explain查询计划优化章节，即explain的输出结果Extra字段为Using index时，能够触发索引覆盖。

如：select id,name from user where name='a'; id 主键索引 name普通索引

由于name是普通索引，普通索引的叶子节点保存的是主键索引，正好这个地方只是查找了 id 和 name ，所以无须回表，效率较高，这种叫索引覆盖

又如：select id,name,sex from user where name='shenjian';

由于 sex 在 普通索引结构表中找不到，必须要到聚集索引表中通过id 获取sex，效率较低，必须回表查找

3、最左匹配

如：建了一个（name,age,sex） 的符合索引

where name = '' 或 name='' and age = 11 是走索引的 where age=12 and sex = 1 是不走索引的（因为索引结构是左节点小于右节点，是有顺序的）

index （a,b,c）为例建立这样的索引相当于建立了索引a、ab、abc三个索引。就是最左优先

4、索引下推（组合索引） ————— 先在索引层面做一次判断，进行一次过滤

索引下推是mysql 5.6 添加的，用于优化查询

如果一个组合作引(name,age,sex)

如果 能在 存储引擎中 判断age和sex 就是索引下推，如果在mysql服务层过滤age 和sex 就不是下推

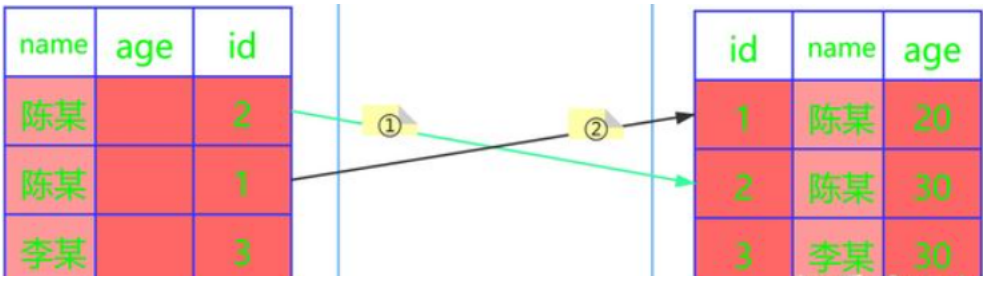
select * from table where name = 'a' and age = 12 and sex =1 ==⇒ 索引下推

select * from table where name = 'a' 不是

如：SELECT * from user where name like '陈%' and age=20 =======》 index(name,age)

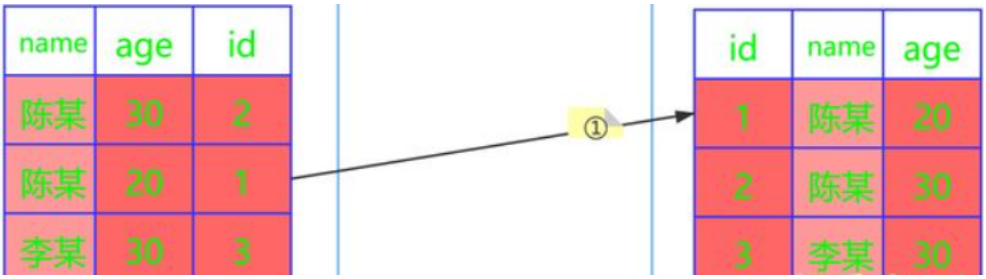
mysql 5.6 之前

会忽略age这个字段，直接通过name查询，查询到了两条记录，id分别是2和1，然后通过 2和1再做一次回表查询



mysql5.6之后

innodb并没有忽略age这个字段，而是在索引内部就判断了age是否等于20，对于不等于20的记录直接跳过，这样在（name,age）这棵索引树中只匹配到了一个记录，然后通过只匹配到的这一条进行回表查询



▼ 索引匹配方式

▼ 全值匹配

全值匹配指的是和索引中的所有列进行匹配 index_sap(name,age,pos)

explain select * from staffs where name = 'July' and age = '23' and pos = 'dev';

rows 只是预估值

```
mysql> explain select * from staffs where name = 'July' and age = '23' and pos = 'dev';
+----+-------------+--------+------------+------+---------------+---------+---------+-------------------+------+----------+-------+
| id | select_type | table  | partitions | type | possible_keys | key     | key_len | ref               | rows | filtered | Extra |
+----+-------------+--------+------------+------+---------------+---------+---------+-------------------+------+----------+-------+
|  1 | SIMPLE      | staffs | NULL       | ref  | idx_nap       | idx_nap | 140     | const,const,const |    1 |   100.00 | NULL  |
+----+-------------+--------+------------+------+---------------+---------+---------+-------------------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

▼ 匹配最左前缀

根据组合索引列顺序 从左前后匹配 只匹配前面的几列

explain select * from staffs where name = 'July' and age = '23';

explain select * from staffs where name = 'July';

▼ 匹配列前缀

可以匹配某一列的值的开头部分

explain select * from staffs where name like 'J%'; ——用到了索引

explain select * from staffs where name like '%y'; ——不会用到索引

所以模糊查询的时候 %模糊值得前面，索引列是用不到的———一个优化的点

▼ 匹配范围值

可以查找某一个范围的数据

explain select * from staffs where name > 'Mary'; —— 会用到索引

▼ 精确匹配某一列并范围匹配另外一列

可以查询第一列的全部和第二列的部分

explain select * from staffs where name = 'July' and age > 25;

▼ 只访问索引的查询

查询的时候只需要访问索引，不需要访问数据行，本质上就是覆盖索引

explain select name,age,pos from staffs where name = 'July' and age = 25 and pos = 'dev';

如果 执行计划中的 Extra 的结果是 Using index 说明就是索引覆盖

▼ **哈希索引**

1、基于哈希表的实现，只有精确匹配索引所有列的查询才有效

2、在mysql中，只有memory的存储引擎显式支持哈希索引

3、哈希索引自身只需存储对应的hash值，所以索引的结构十分紧凑，这让哈希索引查找的速度非常快

4、哈希索引的限制

1、哈希索引只包含哈希值和行指针，而不存储字段值，索引不能使用索引中的值来避免读取行

2、哈希索引数据并不是按照索引值顺序存储的，所以无法进行排序

3、哈希索引不支持部分列匹配查找，哈希索引是使用索引列的全部内容来计算哈希值

4、哈希索引支持等值比较查询，也不支持任何范围查询

5、访问哈希索引的数据非常快，除非有很多哈希冲突，当出现哈希冲突的时候，存储引擎必须遍历链表中的所有行指针，逐行进行比较，直到找到所有符合条件的行

6、哈希冲突比较多的话，维护的代价也会很高，链表会很长

5、案例———可以当缓存用，因为没有持久化，在内存中，效率高

当需要存储大量的URL，并且根据URL进行搜索查找，如果使用B+树，存储的内容就会很大

select id from url where url=""

也可以利用将url使用CRC32做哈希，可以使用以下查询方式：

select id fom url where url="" and url_crc=CRC32("")

此查询性能较高原因是使用体积很小的索引来完成查找

▼ 组合索引——考虑顺序和空间问题

- 当包含多个列作为索引，需要注意的是正确的顺序依赖于该索引的查询，同时需要考虑如何更好的满足排序和分组的需要
- 案例，建立组合索引a,b,c不同SQL语句使用索引情况

语句	索引是否发挥作用
where a=3	是，只使用了a
where a=3 and b=5	是，使用了a,b
where a=3 and b=5 and c=4	是，使用了a,b,c
where b=3 or where c=4	否
where a=3 and c=4	是，仅使用了a
where a=3 and b>10 and c=7	是，使用了a,b
where a=3 and b like '%xx%' and c=7	使用了a
xx%	使用了 a,b

倒数第三行，如果 b 是一个范围，后面列则索引不生效

▼ 聚簇索引与非聚簇索引

- 聚簇索引：不是单独的索引类型，而是一种数据存储方式，指的是数据行跟相邻的键值紧凑的存储在一起

  优点：

  1、可以把相关数据保存在一起

  2、数据访问更快，因为索引和数据保存在同一个树中

  3、使用覆盖索引扫描的查询可以直接使用页节点中的主键值

  缺点：

  1、聚簇数据最大限度地提高了IO密集型应用的性能，如果数据全部在内存，那么聚簇索引就没有什么优势

  2、插入速度严重依赖于插入顺序，按照主键的顺序插入是最快的方式

  如果不是顺序，会产生页分裂/页合并过程，浪费额外的资源

  3、更新聚簇索引列的代价很高，因为会强制将每个被更新的行移动到新的位置

4、基于聚簇索引的表在插入新行，或者主键被更新导致需要移动行的时候，可能面临页分裂的问题

5、聚簇索引可能导致全表扫描变慢，尤其是行比较稀疏，或者由于页分裂导致数据存储不连续的时候

- 非聚簇索引：数据文件跟索引文件分开存放

▼ 覆盖索引

- 基本介绍

  1、如果一个索引包含所有需要查询的字段的值，我们称之为覆盖索引

  2、不是所有类型的索引都可以称为覆盖索引，覆盖索引必须要存储索引列的值

  3、不同的存储实现覆盖索引的方式不同，不是所有的引擎都支持覆盖索引，memory不支持覆盖索引

- 优势

  1、索引条目通常远小于数据行大小，如果只需要读取索引，那么mysql就会极大的较少数据访问量（IO）

  2、因为索引是按照列值顺序存储的，所以对于IO密集型的范围查询会比随机从磁盘读取每一行数据的IO要少的多

  3、一些存储引擎如MYISAM在内存中只缓存索引，数据则依赖于操作系统来缓存，因此要访问数据需要一次系统调用，这可能会导致严重的性能问题

- 案例演示

  1、当发起一个被索引覆盖的查询时，在explain的extra列可以看到using index的信息，此时就使用了覆盖索引

```
mysql> explain select store_id,film_id from inventory\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: inventory
   partitions: NULL
         type: index
possible_keys: NULL
          key: idx_store_id_film_id
      key_len: 3
          ref: NULL
         rows: 4581
     filtered: 100.00
        Extra: Using index
1 row in set, 1 warning (0.01 sec)
```

  2、在大多数存储引擎中，覆盖索引只能覆盖那些只访问索引中部分列的查询。不过，可以进一步的进行优化，可以使用innodb的二级索引来覆盖查询。

  例如：actor使用innodb存储引擎，并在last_name字段又二级索引，虽然该索引的列不包括主键actor_id，但也能够用于对actor_id做覆盖查询

```
mysql> explain select actor_id,last_name from actor where last_name='HOPPER'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
   partitions: NULL
         type: ref
possible_keys: idx_actor_last_name
          key: idx_actor_last_name
      key_len: 137
          ref: const
         rows: 2
     filtered: 100.00
        Extra: Using index
1 row in set, 1 warning (0.00 sec)
```

▼ 优化小细节

- 当使用索引列进行查询的时候尽量不要使用表达式，把计算放到业务层而不是数据库层

  select actor_id from actor where actor_id=4;

~~select actor_id from actor where actor_id+1=5;~~

- 尽量使用主键查询，而不是其他索引，因此主键查询不会触发回表查询
- 使用前缀索引

  有时候需要索引很长的字符串，这会让索引变的大且慢，通常情况下可以使用某个列开始的部分字符串，这样大大的节约索引空间，从而提高索引效率，但这会降低索引的选择性，索引的选择性是指不重复的索引值和数据表记录总数的比值，范围从1/#T到1之间。索引的选择性越高则查询效率越高，因为选择性更高的索引可以让mysql在查找的时候过滤掉更多的行。

  一般情况下某个列前缀的选择性也是足够高的，足以满足查询的性能，但是对应BLOB,TEXT,VARCHAR类型的列，必须要使用前缀索引，因为mysql不允许索引这些列的完整长度，使用该方法的诀窍在于要选择足够长的前缀以保证较高的选择性，通过又不能太长。

  ▼ 案例展示

```
--创建数据表
create table citydemo(city varchar(50) not null);
insert into citydemo(city) select city from city;

--重复执行5次下面的sql语句
insert into citydemo(city) select city from citydemo;   ----- 目的是产生好多重复数据

--更新城市表的名称
update citydemo set city=(select city from city order by rand() limit 1);

--查找最常见的城市列表，发现每个值都出现45-65次，
select count(*) as cnt,city from citydemo group by city order by cnt desc limit 10;

--查找最频繁出现的城市前缀，先从3个前缀字母开始，发现比原来出现的次数更多，可以分别截取多个字符查看城市出现的次数
select count(*) as cnt,left(city,3) as pref from citydemo group by pref order by cnt desc limit 10;
select count(*) as cnt,left(city,7) as pref from citydemo group by pref order by cnt desc limit 10;
--此时前缀的选择性接近于完整列的选择性

--还可以通过另外一种方式来计算完整列的选择性，可以看到当前缀长度到达7之后，再增加前缀长度，选择性提升的幅度已经很小了
select count(distinct left(city,3))/count(*) as sel3,
count(distinct left(city,4))/count(*) as sel4,
count(distinct left(city,5))/count(*) as sel5,
count(distinct left(city,6))/count(*) as sel6,
count(distinct left(city,7))/count(*) as sel7,
count(distinct left(city,8))/count(*) as sel8
from citydemo;

--计算完成之后可以创建前缀索引
alter table citydemo add key(city(7));

--注意：前缀索引是一种能使索引更小更快的有效方法，但是也包含缺点：mysql无法使用前缀索引做order by 和 group by。
```

- 使用索引扫描来排序————— 因为索引已经排好序了

  mysql有两种方式可以生成有序的结果：通过排序操作或者按索引顺序扫描，如果explain出来的type列的值为index,则说明mysql使用了索引扫描来做排序

  扫描索引本身是很快的，因为只需要从一条索引记录移动到紧接着的下一条记录。但如果索引不能覆盖查询所需的全部列，那么就不得不每扫描一条索引记录就得回表查询一次对应的行，这基本都是随机IO，因此按索引顺序读取数据的速度通常要比顺序地全表扫描慢

  mysql可以使用同一个索引即满足排序，又用于查找行，如果可能的话，设计索引时应该尽可能地同时满足这两种任务。

  只有当索引的列顺序和order by子句的顺序完全一致，并且所有列的排序方式都一样时，mysql才能够使用索引来对结果进行排序，如果查询需要关联多张表，则只有当orderby子句引用的字段全部为第一张表时，才能使用索引做排序。order by子句和查找型查询的限制是一样的，需要满足索引的最左前缀的要求，否则，mysql都需要执行顺序操作，而无法利用索引排序

  ▼ 案例演示

```
--sakila数据库中rental表在rental_date,inventory_id,customer_id上有rental_date的索引
--使用rental_date索引为下面的查询做排序
explain select rental_id,staff_id from rental where rental_date='2005-05-25' order by inventory_id,customer_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
```

```
        table: rental
   partitions: NULL
         type: ref
possible_keys: rental_date
          key: rental_date
      key_len: 5
          ref: const
         rows: 1
     filtered: 100.00
        Extra: Using index condition
1 row in set, 1 warning (0.00 sec)
--order by子句不满足索引的最左前缀的要求，也可以用于查询排序，这是因为所以你的第一列被指定为一个常数

--该查询为索引的第一列提供了常量条件，而使用第二列进行排序，将两个列组合在一起，就形成了索引的最左前缀
explain select rental_id,staff_id from rental where rental_date='2005-05-25' order by inventory_id desc\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: ref
possible_keys: rental_date
          key: rental_date
      key_len: 5
          ref: const
         rows: 1
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)

--下面的查询不会利用索引
explain select rental_id,staff_id from rental where rental_date>'2005-05-25' order by rental_date,inventory_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: ALL
possible_keys: rental_date
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16005
     filtered: 50.00
        Extra: Using where; Using filesort            --  如果前面是一个范围  则组合索引失效，不走索引

--该查询使用了两中不同的排序方向，但是索引列都是正序排序的
explain select rental_id,staff_id from rental where rental_date='2005-05-25' order by inventory_id desc,customer_id asc\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: ALL
possible_keys: rental_date
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16005
     filtered: 50.00
        Extra: Using where; Using filesort           --  如果一个升序，一个降序   组合索引不知道怎么排   不走索引
1 row in set, 1 warning (0.00 sec)

--该查询中引用了一个不再索引中的列
explain select rental_id,staff_id from rental where rental_date='2005-05-25' order by inventory_id,staff_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: ALL
possible_keys: rental_date
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16005
     filtered: 50.00
        Extra: Using where; Using filesort
1 row in set, 1 warning (0.00 sec)
```

```
---  总结：如果使用排序的时候，where 条件和  order by 可以组成一个组合索引 ，
---        where 不是范围查找   oderby  排序条件一直都是asc 或者都是desc   则会使用索引排序
```

- union all,in,or都能够使用索引，但是推荐使用in—— 执行计划基本一样，效率也差不多，但是in更好一点

  explain select * from actor where actor_id = 1 union all select * from actor where actor_id = 2;

  explain select * from actor where actor_id in (1,2);

  explain select * from actor where actor_id = 1 or actor_id =2;

- 范围列可以用到索引

  范围条件是：<、<=、>、>=、between

  范围列可以用到索引，但是范围列后面的列无法用到索引，索引最多用于一个范围列

- 强制类型转换会全表扫描

  create table user(id int,name varchar(10),phone varchar(11));

  alter table user add index idx_1(phone);

  explain select * from user where phone=13800001234;———— 不触发索引

  explain select * from user where phone='13800001234';———— 触发索引

- 更新十分频繁，数据区分度不高的字段上不宜建立索引

  更新会变更B+树，更新频繁的字段建议索引会大大降低数据库性能

  类似于性别这类区分不大的属性，建立索引是没有意义的，不能有效的过滤数据

  一般区分度在80%以上的时候就可以建立索引，区分度可以使用 count(distinct(列名))/count(*) 来计算

- 创建索引的列，不允许为null，可能会得到不符合预期的结果
- 当需要进行表连接的时候，最好不要超过三张表，因为需要join的字段，数据类型必须一致
- 能使用limit的时候尽量使用limit
- 单表索引建议控制在5个以内
- 单索引字段数不允许超过5个（组合索引）

  因为索引有一个最左匹配原则，列太多的话查找太费事，还有就是浪费空间

- 创建索引的时候应该避免以下错误概念

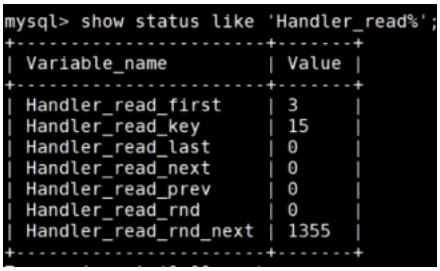
  索引越多越好

  过早优化，在不了解系统的情况下进行优化

▼ 索引监控

https://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html

命令：show status like 'Handler_read%';



参数解析：

Handler_read_first：读取索引第一个条目的次数，读取根节点一共读取了多少次

Handler_read_key：通过index获取数据的次数

Handler_read_last：读取索引最后一个条目的次数

Handler_read_next：通过索引读取下一条数据的次数

Handler_read_prev：通过索引读取上一条数据的次数

Handler_read_rnd：从固定位置读取数据的次数

Handler_read_rnd_next：从数据节点读取下一条数据的次数

注意：应该主要关注 Handler_read_key Handler_read_rnd_next ，如果这两个值越大越好，越小说明索引使用率很差

▼ 简单案例

预先准备好数据

```
SET FOREIGN_KEY_CHECKS=0;
DROP TABLE IF EXISTS `itdragon_order_list`;
CREATE TABLE `itdragon_order_list` (
  `id` bigint(11) NOT NULL AUTO_INCREMENT COMMENT '主键id，默认自增长',
  `transaction_id` varchar(150) DEFAULT NULL COMMENT '交易号',
  `gross` double DEFAULT NULL COMMENT '毛收入(RMB)',
  `net` double DEFAULT NULL COMMENT '净收入(RMB)',
  `stock_id` int(11) DEFAULT NULL COMMENT '发货仓库',
  `order_status` int(11) DEFAULT NULL COMMENT '订单状态',
  `descript` varchar(255) DEFAULT NULL COMMENT '客服备注',
  `finance_descript` varchar(255) DEFAULT NULL COMMENT '财务备注',
  `create_type` varchar(100) DEFAULT NULL COMMENT '创建类型',
  `order_level` int(11) DEFAULT NULL COMMENT '订单级别',
  `input_user` varchar(20) DEFAULT NULL COMMENT '录入人',
  `input_date` varchar(20) DEFAULT NULL COMMENT '录入时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=10003 DEFAULT CHARSET=utf8;

INSERT INTO itdragon_order_list VALUES ('10000', '81X97310V32236260E', '6.6', '6.13', '1', '10', 'ok', 'ok', 'auto', '1', 'itdragon', '
INSERT INTO itdragon_order_list VALUES ('10001', '61525478BB371361Q', '18.88', '18.79', '1', '10', 'ok', 'ok', 'auto', '1', 'itdragon',
INSERT INTO itdragon_order_list VALUES ('10002', '5RT64180WE555861V', '20.18', '20.17', '1', '10', 'ok', 'ok', 'auto', '1', 'itdragon',
```

逐步开始进行优化：

第一个案例：

```
select * from itdragon_order_list where transaction_id = "81X97310V32236260E";
--通过查看执行计划发现type=all,需要进行全表扫描
explain select * from itdragon_order_list where transaction_id = "81X97310V32236260E";

--优化一、为transaction_id创建唯一索引
 create unique index idx_order_transaID on itdragon_order_list (transaction_id);
--当创建索引之后，唯一索引对应的type是const，通过索引一次就可以找到结果，普通索引对应的type是ref，表示非唯一性索引赛秒，找到值还要进行扫描，直到将索引文件扫描
 explain select * from itdragon_order_list where transaction_id = "81X97310V32236260E";

 --优化二、使用覆盖索引，查询的结果变成 transaction_id,当extra出现using index,表示使用了覆盖索引
 explain select transaction_id from itdragon_order_list where transaction_id = "81X97310V32236260E";
```

第二个案例

```
--创建复合索引
create index idx_order_levelDate on itdragon_order_list (order_level,input_date);

--创建索引之后发现跟没有创建索引一样，都是全表扫描，都是文件排序
explain select * from itdragon_order_list order by order_level,input_date;

--可以使用force index强制指定索引
explain select * from itdragon_order_list force index(idx_order_levelDate) order by order_level,input_date;
--其实给订单排序意义不大，给订单级别添加索引意义也不大，因此可以先确定order_level的值，然后再给input_date排序
explain select * from itdragon_order_list where order_level=3 order by input_date;
```