回想四年前，我在学习mysql的索引这块的时候，老师在讲索引的时候，是像下面这么说的

**索引就像一本书的目录。而当用户通过索引查找数据时，就好比用户通过目录查询某章节的某个知识点。这样就帮助用户有效地提高了查找速度。所以，使用索引可以有效地提高数据库系统的整体性能。**

嗯，这么说其实也对。但是呢，大家看完这种说法，其实可能还是觉得太抽象了！因此呢，我还想再深入的细说一下，所以就有了此文！  
需要说明的是，我说的内容只在Mysql的Innodb引擎中是成立的。在Sql Server、oracle、Mysql的Mysiam引擎中的正确性，不一定成立！  
OK，废话不多说，开始啰嗦!

## **正文**

### **索引的科普**

先引进聚簇索引和非聚簇索引的概念！  
我们平时在使用的Mysql中，使用下述语句

CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index\_name

[USING index\_type]

ON tbl\_name (index\_col\_name,...)

index\_col\_name:

col\_name [(length)] [ASC | DESC]

创建的索引，如复合索引、前缀索引、唯一索引，都是属于非聚簇索引，在有的书籍中，又将其称为辅助索引(secondary index)。在后文中，我们称其为非聚簇索引，其数据结构为B+树。

那么，这个聚簇索引，在Mysql中是没有语句来另外生成的。在Innodb中，Mysql中的数据是按照主键的顺序来存放的。那么聚簇索引就是按照每张表的主键来构造一颗B+树，叶子节点存放的就是整张表的行数据。由于表里的数据只能按照一颗B+树排序，因此一张表只能有一个聚簇索引。

在Innodb中，聚簇索引默认就是主键索引。  
这个时候，机智的读者，应该要问我

**如果我的表没建主键呢？**

回答是，如果没有主键，则按照下列规则来建聚簇索引

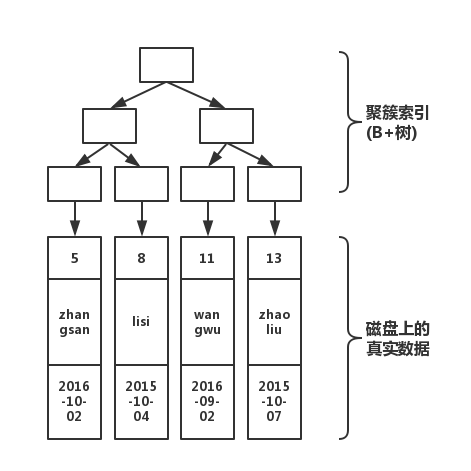
* 没有主键时，会用一个唯一且不为空的索引列做为主键，成为此表的聚簇索引
* 如果没有这样的索引，InnoDB会隐式定义一个主键来作为聚簇索引。

ps:大家还记得，自增主键和uuid作为主键的区别么？由于主键使用了聚簇索引，如果主键是自增id，，那么对应的数据一定也是相邻地存放在磁盘上的，写入性能比较高。如果是uuid的形式，频繁的插入会使innodb频繁地移动磁盘块，写入性能就比较低了。

### **索引原理介绍**

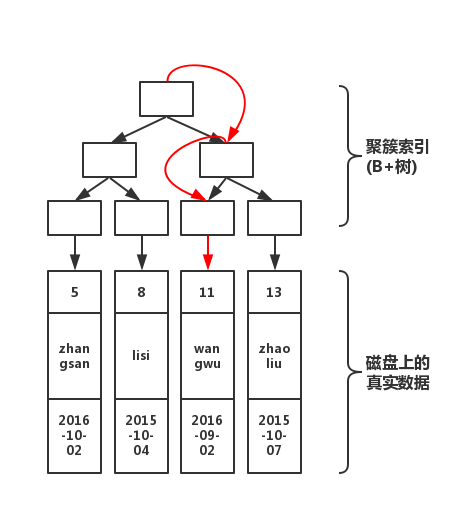
先来一张带主键的表，如下所示，pId是主键

| **pId** | **name** | **birthday** |
| --- | --- | --- |
| 5 | zhangsan | 2016-10-02 |
| 8 | lisi | 2015-10-04 |
| 11 | wangwu | 2016-09-02 |
| 13 | zhaoliu | 2015-10-07 |

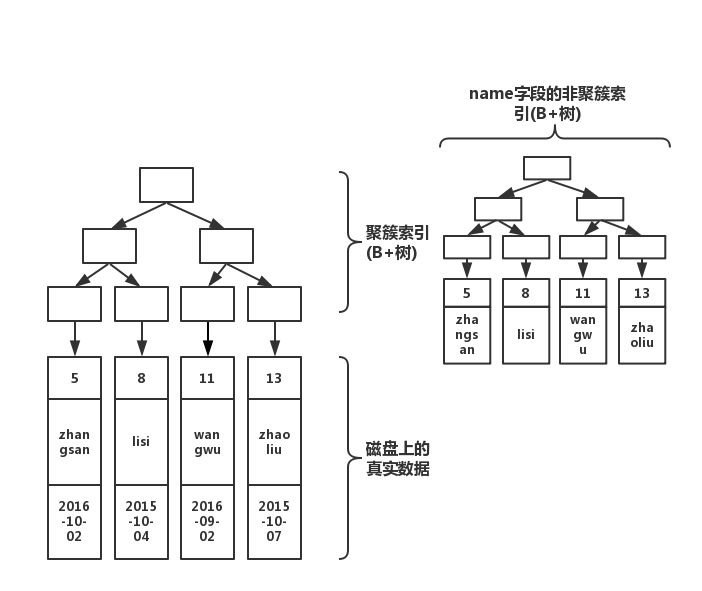
画出该表的结构图如下  


如上图所示，分为上下两个部分，上半部分是由主键形成的B+树，下半部分就是磁盘上真实的数据！那么，当我们， 执行下面的语句

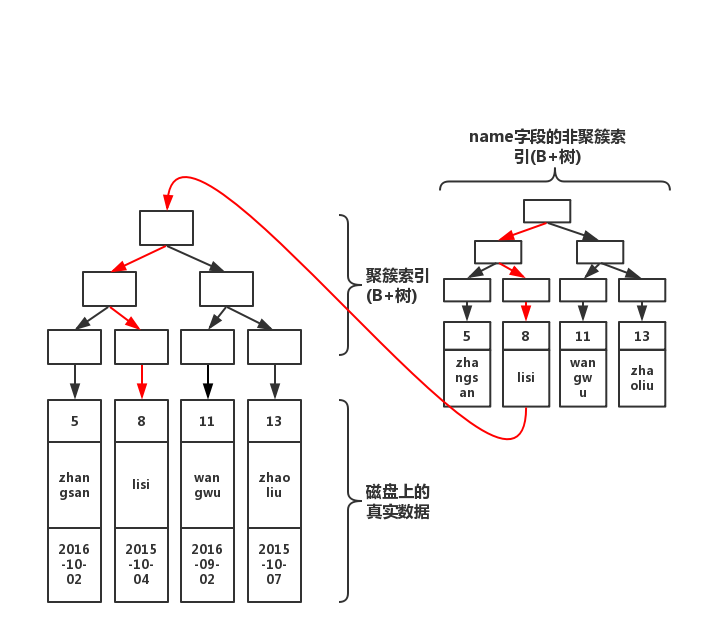
select \* from table where pId='11'

那么，执行过程如下  
  
如上图所示，从根开始，经过3次查找，就可以找到真实数据。如果不使用索引，那就要在磁盘上，进行逐行扫描，直到找到数据位置。显然，使用索引速度会快。但是在写入数据的时候，需要维护这颗B+树的结构，因此写入性能会下降！  
OK，接下来引入非聚簇索引!我们执行下面的语句

create index index\_name on table(name);

此时结构图如下所示  
  
大家注意看，会根据你的索引字段生成一颗新的B+树。因此， 我们每加一个索引，就会增加表的体积， 占用磁盘存储空间。然而，**注意看叶子节点**，非聚簇索引的叶子节点并不是真实数据，它的叶子节点依然是索引节点，存放的是该索引字段的值以及对应的主键索引(聚簇索引)。  
如果我们执行下列语句

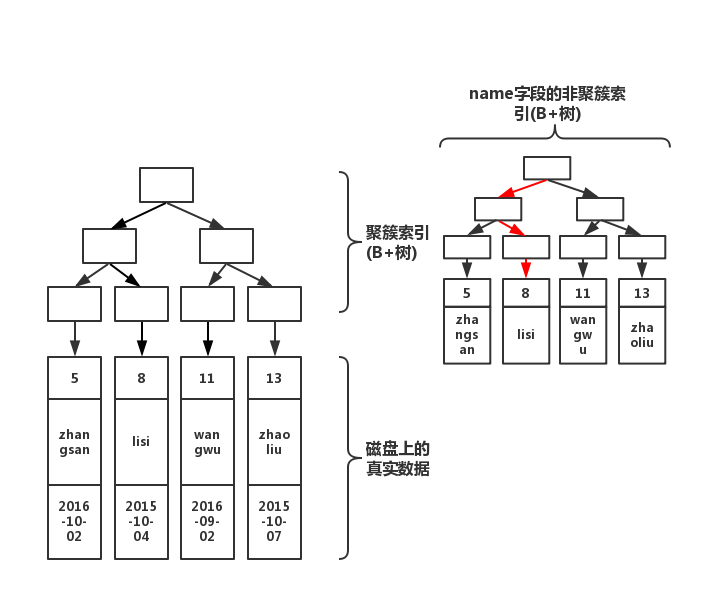
select \* from table where name='lisi'

此时结构图如下所示  
  
通过上图红线可以看出，先从非聚簇索引树开始查找，然后找到聚簇索引(id)后。根据聚簇索引(id)，在聚簇索引的B+树上，找到完整的数据！  
那

**什么情况不去聚簇索引树上查询呢？**

还记得我们的非聚簇索引树上存着该索引字段的值么。如果，此时我们执行下面的语句

select name from table where name='lisi'

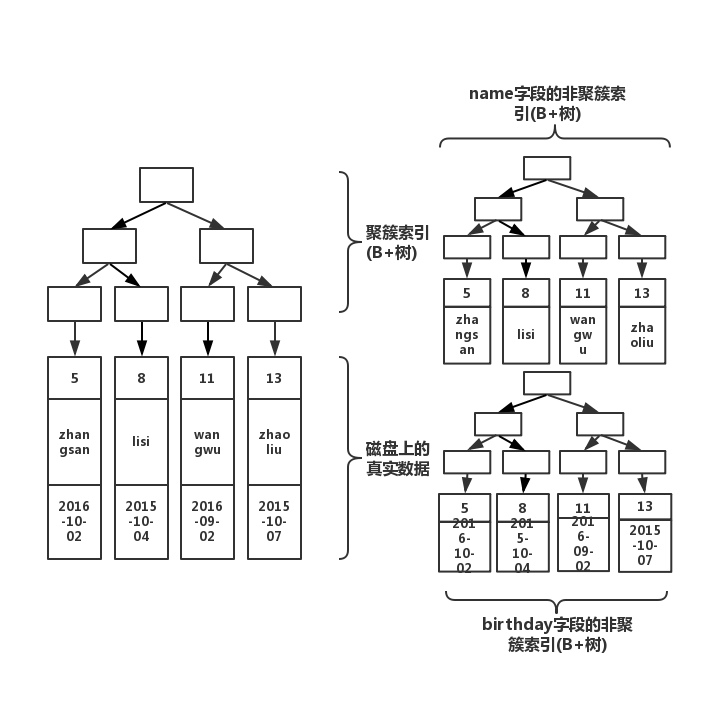
此时结构图如下  
  
如上图红线所示，如果在非聚簇索引树上找到了想要的值，就不会去聚簇索引树上查询。还记得，博主在[《select的正确姿势》](https://www.cnblogs.com/rjzheng/p/9902911.html)提到的索引问题么：

**当执行select col from table where col = ?，col上有索引的时候，效率比执行select \* from table where col = ? 速度快好几倍！**

看完上面的图，你应该对这句话有更深层的理解了。

那么这个时候，我们执行了下述语句，又会发生什么呢？

create index index\_birthday on table(birthday);

此时结构图如下  
  
看到了么，多加一个索引，就会多生成一颗非聚簇索引树。因此，很多文章才说，索引不能乱加。因为，有几个索引，就有几颗非聚簇索引树！你在做插入操作的时候，需要同时维护这几颗树的变化！因此，如果索引太多，插入性能就会下降!

## **总结**

讲到这里，大家应该清楚的明白索引的原理了！可能细节方面还不够严谨，但是我觉得一个研发，理解到这里可以了，够用了，毕竟我们也不是专业的DBA。

回想四年前，我在学习mysql的索引这块的时候，老师在讲索引的时候，是像下面这么说的

**索引就像一本书的目录。而当用户通过索引查找数据时，就好比用户通过目录查询某章节的某个知识点。这样就帮助用户有效地提高了查找速度。所以，使用索引可以有效地提高数据库系统的整体性能。**

嗯，这么说其实也对。但是呢，大家看完这种说法，其实可能还是觉得太抽象了！因此呢，我还想再深入的细说一下，所以就有了此文！  
需要说明的是，我说的内容只在Mysql的Innodb引擎中是成立的。在Sql Server、oracle、Mysql的Mysiam引擎中的正确性，不一定成立！  
OK，废话不多说，开始啰嗦!

## **正文**

### **索引的科普**

先引进聚簇索引和非聚簇索引的概念！  
我们平时在使用的Mysql中，使用下述语句

CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index\_name

[USING index\_type]

ON tbl\_name (index\_col\_name,...)

index\_col\_name:

col\_name [(length)] [ASC | DESC]

创建的索引，如复合索引、前缀索引、唯一索引，都是属于非聚簇索引，在有的书籍中，又将其称为辅助索引(secondary index)。在后文中，我们称其为非聚簇索引，其数据结构为B+树。

那么，这个聚簇索引，在Mysql中是没有语句来另外生成的。在Innodb中，Mysql中的数据是按照主键的顺序来存放的。那么聚簇索引就是按照每张表的主键来构造一颗B+树，叶子节点存放的就是整张表的行数据。由于表里的数据只能按照一颗B+树排序，因此一张表只能有一个聚簇索引。

在Innodb中，聚簇索引默认就是主键索引。  
这个时候，机智的读者，应该要问我

**如果我的表没建主键呢？**

回答是，如果没有主键，则按照下列规则来建聚簇索引

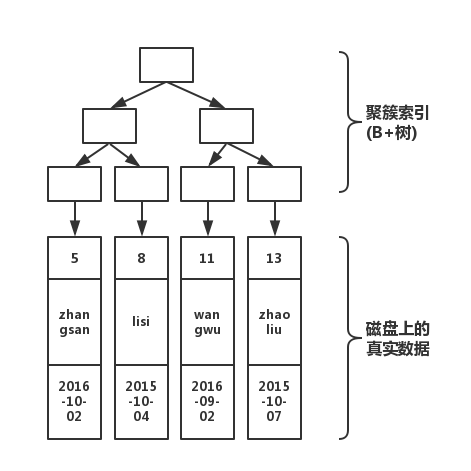
* 没有主键时，会用一个唯一且不为空的索引列做为主键，成为此表的聚簇索引
* 如果没有这样的索引，InnoDB会隐式定义一个主键来作为聚簇索引。

ps:大家还记得，自增主键和uuid作为主键的区别么？由于主键使用了聚簇索引，如果主键是自增id，，那么对应的数据一定也是相邻地存放在磁盘上的，写入性能比较高。如果是uuid的形式，频繁的插入会使innodb频繁地移动磁盘块，写入性能就比较低了。

### **索引原理介绍**

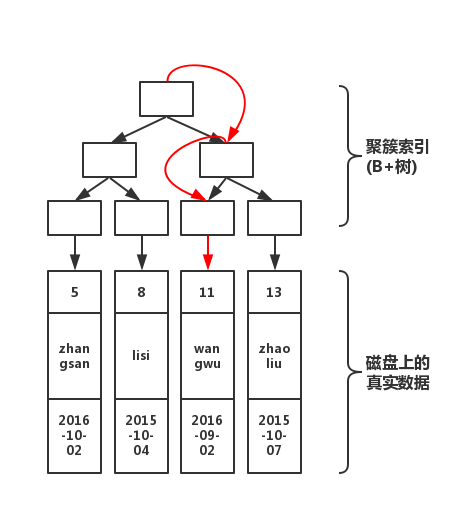
先来一张带主键的表，如下所示，pId是主键

| **pId** | **name** | **birthday** |
| --- | --- | --- |
| 5 | zhangsan | 2016-10-02 |
| 8 | lisi | 2015-10-04 |
| 11 | wangwu | 2016-09-02 |
| 13 | zhaoliu | 2015-10-07 |

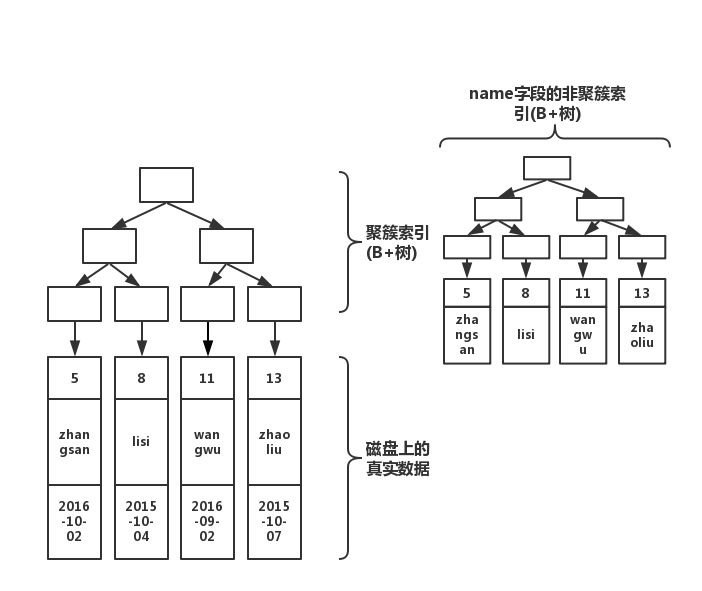
画出该表的结构图如下  


如上图所示，分为上下两个部分，上半部分是由主键形成的B+树，下半部分就是磁盘上真实的数据！那么，当我们， 执行下面的语句

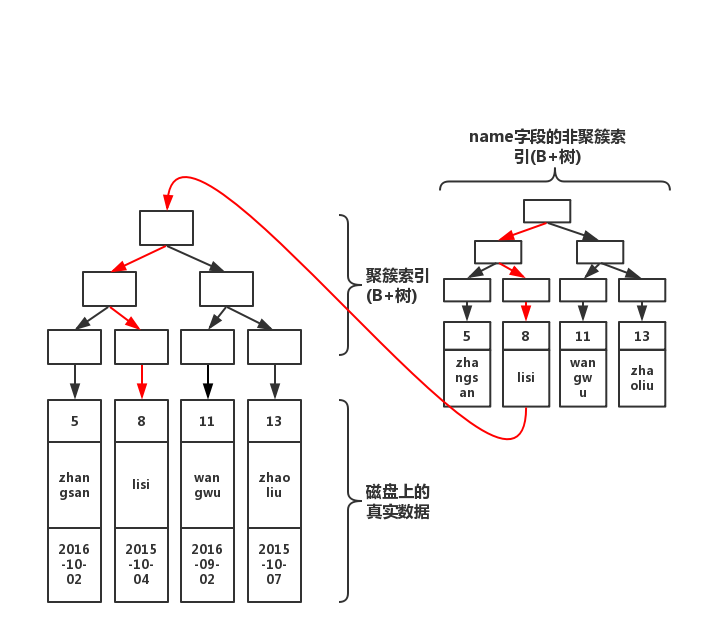
select \* from table where pId='11'

那么，执行过程如下  
  
如上图所示，从根开始，经过3次查找，就可以找到真实数据。如果不使用索引，那就要在磁盘上，进行逐行扫描，直到找到数据位置。显然，使用索引速度会快。但是在写入数据的时候，需要维护这颗B+树的结构，因此写入性能会下降！  
OK，接下来引入非聚簇索引!我们执行下面的语句

create index index\_name on table(name);

此时结构图如下所示  
  
大家注意看，会根据你的索引字段生成一颗新的B+树。因此， 我们每加一个索引，就会增加表的体积， 占用磁盘存储空间。然而，**注意看叶子节点**，非聚簇索引的叶子节点并不是真实数据，它的叶子节点依然是索引节点，存放的是该索引字段的值以及对应的主键索引(聚簇索引)。  
如果我们执行下列语句

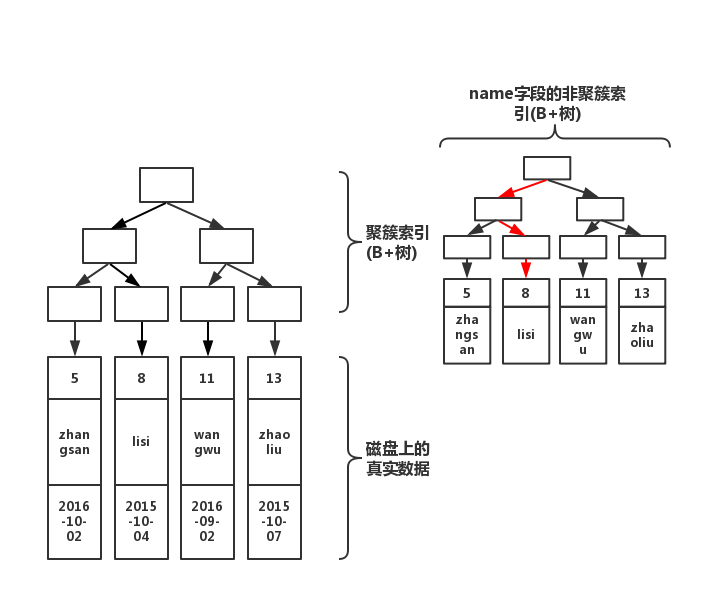
select \* from table where name='lisi'

此时结构图如下所示  
  
通过上图红线可以看出，先从非聚簇索引树开始查找，然后找到聚簇索引后。根据聚簇索引，在聚簇索引的B+树上，找到完整的数据！  
那

**什么情况不去聚簇索引树上查询呢？**

还记得我们的非聚簇索引树上存着该索引字段的值么。如果，此时我们执行下面的语句

select name from table where name='lisi'

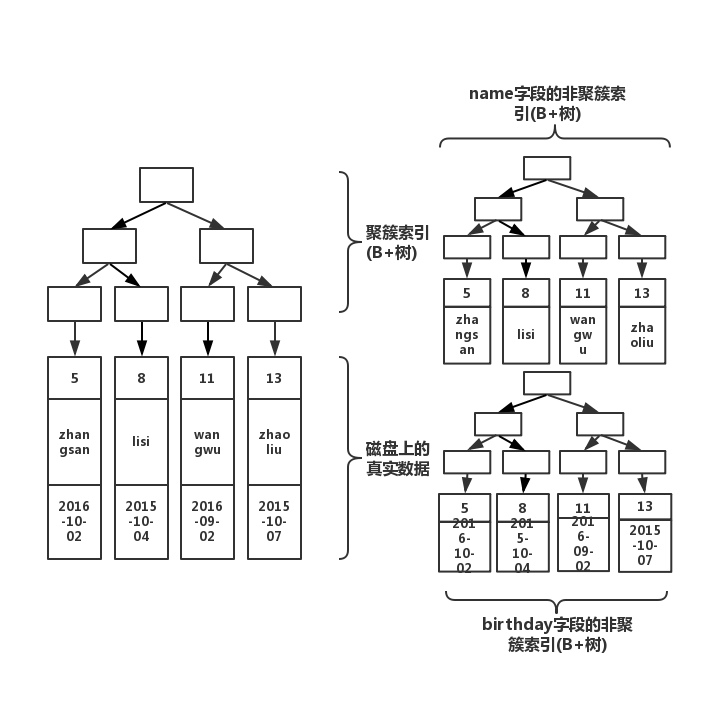
此时结构图如下  
  
如上图红线所示，如果在非聚簇索引树上找到了想要的值，就不会去聚簇索引树上查询。还记得，博主在[《select的正确姿势》](https://www.cnblogs.com/rjzheng/p/9902911.html)提到的索引问题么：

**当执行select col from table where col = ?，col上有索引的时候，效率比执行select \* from table where col = ? 速度快好几倍！**

看完上面的图，你应该对这句话有更深层的理解了。

那么这个时候，我们执行了下述语句，又会发生什么呢？

create index index\_birthday on table(birthday);

此时结构图如下  
  
看到了么，多加一个索引，就会多生成一颗非聚簇索引树。因此，很多文章才说，索引不能乱加。因为，有几个索引，就有几颗非聚簇索引树！你在做插入操作的时候，需要同时维护这几颗树的变化！因此，如果索引太多，插入性能就会下降!

## **总结**

讲到这里，大家应该清楚的明白索引的原理了！可能细节方面还不够严谨，但是我觉得一个研发，理解到这里可以了，够用了，毕竟我们也不是专业的DBA。