# 税友集团

## 税友集团面试题

### 1.简单介绍实践的项目，然后根据项目进行提问

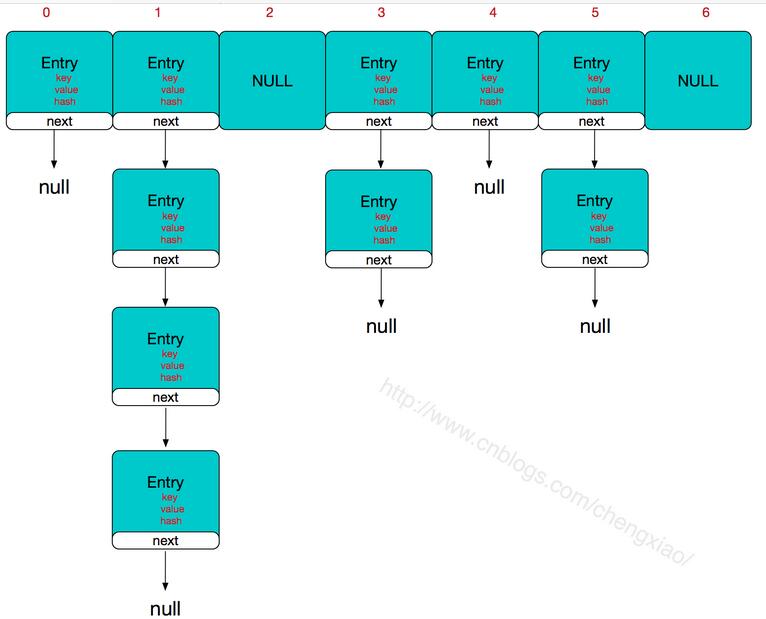
### 2.如何判断数据库中你所加的索引有没有起作用？

创建索引是为了加快查询效率，减少对磁盘的操作。一般可以采取查看相同sql语句的执行时间（但实际上数据库自身会进行一些优化，也可能是其它索引导致的，因此这个说法不合理），使用Oracle数据库时只需要创建索引，oracle会自动管理相应的内容，所以在开发过程中没有太多地去关注这部分的内容。在mysql数据库中可以使用explain关键字实现，explain显示了MySQL如何使用索引来处理select语句以及连接表，它可以帮助选择更好的索引和写出更优化的查询语句。

### 3.你常用的集合有哪些，是在什么情况下用的？

结合实际的情况对常用集合进行阐述，HashMap：在不考虑并发的时候用，ArrayList：在查询次数比较多的时候，ConcurrentHashMap：高并发的时候使用

### 4.简述一下HashMap的原理，为什么key不能重复？



HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了**解决哈希冲突**而存在的，如果定位到的数组位置不含链表（当前entry的next指向null）,那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，HashMap中的链表出现越少，性能才会越好。

HashMap有4个构造器，其他构造器如果用户没有传入initialCapacity 和loadFactor这两个参数，会使用默认值

initialCapacity默认为16，loadFactory默认为0.75

初始值为16，扩容因子为0.75，当满足扩容条件之后会自动扩容一倍

HashMap允许存储null作为键值，但只能有一个键值为null

get方法通过key值返回对应value，如果key为null，直接去table[0]处检索。

尽管我们在进行get和put操作的时候，使用的key从逻辑上讲是等值的（通过equals比较是相等的），但由于没有重写hashCode方法，所以put操作时，key(hashcode1)-->hash-->indexFor-->最终索引位置 ，而通过key取出value的时候 key(hashcode1)-->hash-->indexFor-->最终索引位置，由于hashcode1不等于hashcode2，导致没有定位到一个数组位置而返回逻辑上错误的值null（也有可能碰巧定位到一个数组位置，但是也会判断其entry的hash值是否相等，上面get方法中有提到。）

在重写equals的方法的时候，必须注意重写hashCode方法，同时还要保证通过equals判断相等的两个对象，调用hashCode方法要返回同样的整数值。而如果equals判断不相等的两个对象，其hashCode可以相同（只不过会发生哈希冲突，应尽量避免）

### 5.简述一下ConcurrentHashMap的原理及其具体实现

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是 ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在 ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。

我回答ConcurrentHashMap引入了分割的概念，不再是和Hashtable一样对整个map加锁，而是对其中的一部分加锁。问到具体实现答不上来。。。。。

### 6.开发使用的jdk版本是多少，jdk1.8的新特性？

jdk1.8学习的时候了解到Lambda表达式（涉及匿名内部类的相关概念），使用Lambda表达式替代匿名内部类的创建，但必须满足的是实现的必须为函数式接口。（函数式接口指的是只有一个抽象方法的接口）

### 7.为什么HashMap要使用红黑树？

由于没有深入研究所以答不上来，然后面试官一步步引导，问了链表查询的复杂度是多少，我回答是O(n)，红黑树的复杂度是多少，我回答是O(logn)，追问了我好几次你确定吗，搞得我怀疑人生，根据复杂度给我解释了红黑树的优点，从复杂度可以得出红黑树具有高效检索的优点，另外还说了尽管通过了链表来解决了碰撞，如果所有的都在一条链表上的情况出现，内存的使用率较低，红黑树可以很好的提高内存的使用率。

HashMap在里面就是链表加上红黑树的一种结构，这样利用了链表对内存的使用率以及红黑树的高效检索，是一种很happy的数据结构。

### 8.HashMap除了可以通过链表来解决hash碰撞还有什么办法？

### 9.项目中有没有遇到高并发的情况，是如何处理的？

多应用服务器  
F5负载均衡  
缓存/数据库:读写分离,切片（设置有效时间随机有效降低同一时间数据库的负载）

ps -ef |grep nginx 查看相关的进程信息

nginx是一款自由的、开源的、高性能的HTTP服务器和反向代理服务器；同时也是一个IMAP、POP3、SMTP代理服务器；nginx可以作为一个HTTP服务器进行网站的发布处理，另外nginx可以作为反向代理进行负载均衡的实现。

### 10.如何保证两个数据库的数据保持一致性？

### 11.聊一下同步和异步的区别？

通过实际生活中的案例分析（排队买早饭......）

### 12.doubber、redis、mq相关内容

### 13.最后和我说了一下他们公司对实习生的安排以及还有什么要问的吗？

a.刚刚在整个面试的过程中也意识到自己有很多的不足之处，我想了解一下，在学习之余如何去更好地提升自己，希望能够给我一些建议

感觉自己在课堂上学习的理论知识太过于局限，很难切合实际运用到项目中。整个面试的过程和自己想象中的不太一样。

b.近几天也了解了贵公司的一些信息，好像在深圳、广州都有设置分公司对吧，对于人员的调剂是怎么考虑的（好比如如果之后有事想回深圳或者广州工作有没有这个机会？）

c.员工薪资福利、待遇上来说都是挺不错的，如果真正入职后有怎样的提升机会呢？或者是之后如果想转除开发之外的其他职位是怎么安排？

## 税友集团笔试题

### 1.为什么匿名内部类参数必须为final类型？

**局部变量的生命周期和匿名内部类对象的生命周期的不一致性**

在jdk1.8之前要求被局部内部类、匿名内部类访问的局部变量必须是final修饰的，从jdk1.8之后取消了这个限制，系统会自动添加一个final修饰

匿名内部类不能访问外部类方法中的局部变量，除非变量被声明为final类型。原因是编译程序实现上的困难：**内部类对象的生命周期会超过局部变量的生命周期。**

局部变量的生命周期：当该方法被调用时，该方法中的局部变量在栈中被创建，当方法调用结束时，退栈，这些局部变量全部死亡。而内部类对象生命周期与其它类一样：自创建一个匿名内部类对象，系统为该对象分配内存，直到没有引用变量指向分配给该对象的内存，它才会死亡(被JVM垃圾回收)。所以完全可能出现的一种情况是：成员方法已调用结束，局部变量已死亡，但匿名内部类的对象仍然活着。

如果匿名内部类的对象访问了同一个方法中的局部变量，就要求只要匿名内部类对象还活着，那么栈中的那些它要所访问的局部变量就不能“死亡”。

解决方法：匿名内部类对象可以访问同一个方法中被定义为final类型的局部变量。定义为final后，编译程序的实现方法：对于匿名内部类对象要访问的所有final类型局部变量，都拷贝成为该对象中的一个数据成员。这样，即使栈中局部变量已死亡，但被定义为final类型的局部变量的值永远不变，因而匿名内部类对象在局部变量死亡后，照样可以访问final类型的局部变量，因为它自己拷贝了一份，且与原局部变量的值始终一致。

### 2.将金额数值转化为对应的字符串

### 输出某个字符串的最大非重复子串

### 4.自增运算符

