# Java基础

## 1、中间人代理攻击

## static final的含义，各自的含义

## 0、Exception和Error的区别

## 谈谈final、finally和finalize的区别

* final

1. 被final修饰的类不能再派生新的子类，因此一个类不能即被abstract修饰又被final修饰；
2. 被final声明的变量必须在声明时就给定初始值，在以后的使用中只能读取；
3. 被final声明的方法同样只能使用，不能被子类重写。

* finally

在try-catch中使用，无论是否抛出异常都会执行finally中的内容。如果try中有return语句，那么最后也会先执行finally中的内容，再执行return语句。

* finalize

该方法是Object级别的方法，因此所有类都具有该方法。该方法会在本对象被GC之前调用，例如，

public class TestSE {  
 public final int a = 0;  
 public static void main(String[] args){  
 System.*out*.println("begin main...");  
 TestSE testSE = new TestSE();  
 testSE = null;  
 System.*gc*();  
 }  
 public void finalize(){  
 System.*out*.println("finalize method overriden");  
 }  
}

输出结果：

begin main...

finalize method overridden

## 强引用、软引用、弱引用和虚引用

## String、StringBuffer和StringBuilder的区别

## 动态代理是基于什么原理

首先了解设计模式-代理模式。

## int和Integer的区别

## 对比Vector、ArrayList和LinkedList的区别

## Hashtable、HashMap、HashSet和TreeMap的区别

## 如何保证集合是线程安全的？ConcorrentHashMap如何实现高效的线程安全？

## Java提供了哪些IO方式？NIO如何实现多路复用？

## Java有几种文件拷贝方式？哪一种最高效？

## 接口和抽象类有什么区别的？

## 谈谈你知道的设计模式？

代理模式、单例模式(枚举实现单例、双重检查锁)

## 13、对象的实例化过程和顺序

## 重写与重载？

## 14、HashMap不会出现重复的key，因为重复的key后者会覆盖前者。

# Java进阶

## synchronized和ReentrantLock的区别？

## synchronized底层如何实现？什么是锁的升级、降级？

## 一个线程两次调用start()方法会发生什么情况？

## 什么情况下Java程序会产生死锁？如何定位、修复？

## Java并发包提供了哪些并发工具？

## 并发包中的ConcurrentLinkedQueue和LinkedBlockingQueue有什么区别？

## Java并发类库提供的线程池有哪几种？分别有什么特点？

## AtomicInteger底层实现原理是什么？如何在自己的产品代码中应用CAS操作？

## CAS(compare and swap)

CAS有三个操作参数：内存地址A、期望值B、要修改的新值D。

原理：比如想要修改A地址的值，首先将A地址的值进行复制，取名为B。然后获取A地址的现在的值C，如果C与B还相等则证明A地址的值没有被修改过，将修改后的值D替换掉现在A地址的值，这个期间是同步的，如果

CAS的优势：

CAS是CPU指令级的操作，只有一步原子操作，所以非常快。而且CAS避免了请求操作系统来裁定锁的问题，不用麻烦操作系统，直接在CPU内部搞定。

## 请介绍类加载过程，什么是双亲委派模型？

## 有哪些方法可以在运行时动态生成一个Java类？

## 谈谈JVM内存区域的划分，哪些区域可能发生OutOfMemoryError?

## 如何监控和诊断JVM堆内和堆外内存使用？

## Java常见的垃圾收集器有哪些？

## 谈谈你的GC调优思路？

## Java内存模型中的happen-before是什么？

## Java程序运行在Docker等容器环境有什么新问题？

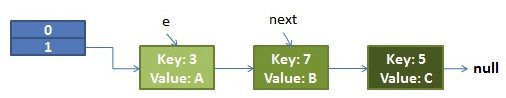
## 你知道JVM JIT的内存机制吗？

## 为什么HashMap在多线程的情况下会有线程安全问题？

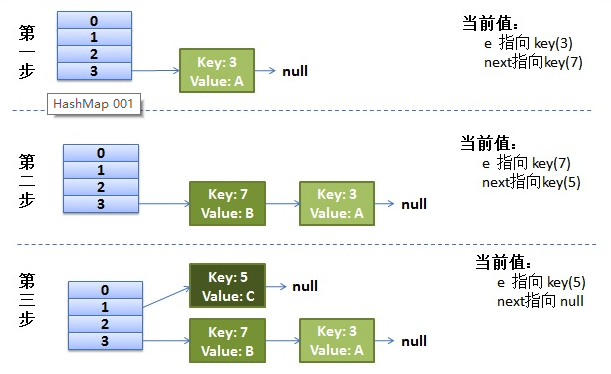
注意：HashMap在JDK1.7的时候有两个线程安全问题，在JDK1.8的时候有一个线程安全问题。在1.7中和1.8中都会出现的线程安全问题是在执行put操作的时候，线程之间的操作有可能相互覆盖，比如A线程获取了一个桶的首节点，这时时间片用完，B线程开始执行，B执行的时候正好在这个桶的首节点中插入了一个键值对，这时B线程执行完成，切换到A线程，A线程此时并不知道里面的值已经存在，所以继续执行就会覆盖B线程之前插入的值。只有在1.8以下才会出现的问题如下：

在讨论这个问题之前先要搞清楚Java的内存模型：Java分为栈内存和堆内存，栈内存主要存储方法的局部变量、对象引用等；堆内存主要存储对象的实例。并且每个方法在被调用的时候都会将方法内的局部变量和对象引用放入调用方法的栈帧中。而且栈是线程私有的，即多个线程调用相同的方法，每个线程的方法栈都会有该方法的相同的一份栈帧。对于对象与对象之间的赋值，比如obj1 = obj2，在内存中的表现是obj1的引用指向堆内存中的obj2的实例。明白了这些之后再来看这个线程安全问题。

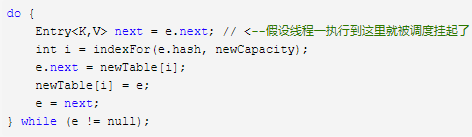
线程安全问题发生在多个线程对同一个HashMap的rehash的过程中。假设新的哈希表的索引是通过key的值mod哈希表的大小获得的。原始的哈希表如下：



那么rehash的过程就如下所示，

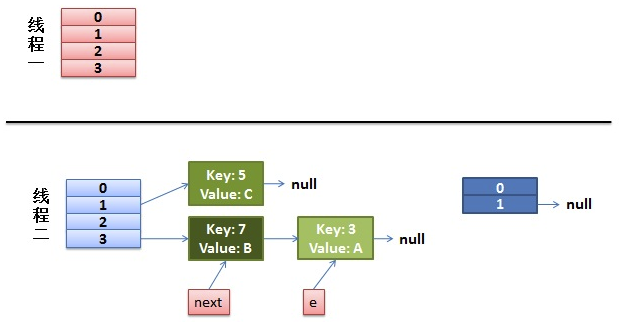


resize()中的transfer()（安全问题主要发生在这个代码中）的源代码如下：

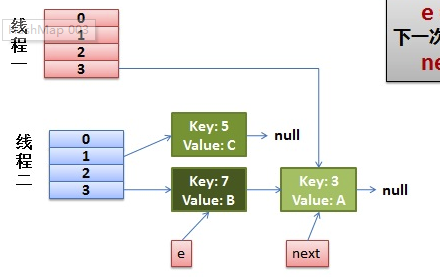


线程二执行完成了。那么线程一、线程二的总体情况如下，线程二图中next和e的指向就是两个线程各自栈帧中的next和e的指向，因为堆内存上的实例并没有变，变得是自己对象引用的指向，注意这点，所以对于线程一的next也是指向key7，e也是指向key3。

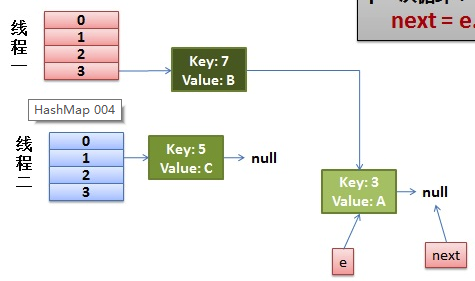
这里也可以看到rehash之后，新哈希表中的链表部分与旧表是逆序的(逆序的原因主要就是因为e.next=newTable[i]; newTable[i]=e;这两句代码)。这里为什么要从头节点开始插入呢，因为从尾节点插入的时间复杂度为O(n)。



接下来线程一继续执行，先执行newTable[i]=e，索引为3的元素指向key3，然后e=next，e指向key7，下一次循环next=e.next，使next指向key3，如图，



接着，e.next=newTable[i]，导致e.next指向key3，然后newTable[i]=e，导致索引为3的元素指向key7，之后e=next，e指向key3，如图，



下面，e.next=newTable[i]，导致key3指向key7，到这里出现了环形链表。于是，在使用get()方法查找key的时候，如果定位到索引为3的元素，而这时恰好查找的key不存在，那么遍历后面的链表就会陷入死循环，导致CPU的使用率暴增，这是一个由多线程引起的问题。

## synchronized的用法总结

<https://blog.csdn.net/javazejian/article/details/72828483>

* 修饰实例方法，给当前实例加锁，进入同步代码前要获得当前实例的锁。这也就意味着，如果一个线程执行该对象的一个synchronized方法，那么其他线程也不能执行该对象的其他synchronized方法，因为这个对象作为一个对象锁已经被占用了。
* 修饰静态方法，给当前类和该类的对象加锁，进入同步代码前要获得当前类的锁。如果一个线程A调用一个对象的非static synchronized方法，而线程B调用这个实例对象所属类的静态synchronized方法，是允许的，
* 修饰代码块，指定加锁对象。可以为对象枷锁，可以为类加锁。比如，为当前对象加锁，访问限制同上。

## volatile关键字的用法总结

<https://www.cnblogs.com/dolphin0520/p/3920373.html>

## 34、为什么重写equal()方法同时要重写hashcode()方法？

首先在Java中hashcode()方法只有在一种情况下有用，那就是用到散列表的时候(HashMap、HashSet、Hashtable等)，因为在这些表中需要使用hashcode()方法来获取hash值，从而计算在散列表中桶的位置。其他地方基本用不到该方法。

只有在涉及到散列表的时候equal()方法才与hashcode()方法有关系。其他地方，二者没有半毛钱关系。因为在HashMap的源码中分析得到，当二者的hash值(根据hashcode()计算得来)相同的时候，二者是会被分到同一个桶中的，之后再使用equal()判断二者是否相同，如果不相同，将后者添加到该桶后面的链表中；如果相同，将原来的key进行覆盖。所以需要先使用hashcode()来判断桶的位置，再使用equal()判断是否添加链表。所以二者在使用到散列表时是有关系的。

如果重写了equal()方法而没有重写hashcode()方法，那么就会发生一种情况就是两者相同但是hash值不同，这就造成分在了不同的桶中，但是二者的值却相同，这违背了HashSet的原则：不存储重复的值。所以重写equal()方法的同时需要重写hashcode()方法。

# Java应用开发扩展

## 36、谈谈乐观锁和悲观锁的原理和应用场景？

## 37、谈谈Spring Bean的生命周期和作用域？

## 38、什么场景下需要用到Netty？对比Java标准NIO类库，你知道Netty如何实现更高性能吗？

## 39、谈谈你了解的常用的分布式ID的设计方案？