<https://www.jianshu.com/p/8b372f3a195d>

这次主要是分析下HashMap的工作原理，为什么我会拿这个东西出来分析，原因很简单，以前我面试的时候，偶尔问起HashMap，99%的程序员都知道HashMap，基本都会用Hashmap，这其中不仅仅包括刚毕业的大学生，也包括已经工作5年，甚至是10年的程序员。HashMap涉及的知识远远不止put和get那么简单。本次的分析希望对于面试的人起码对于面试官的问题有所应付

\*\* 一、先来回忆下我的面试过程\*\*

\*\* 问：“你用过HashMap，你能跟我说说它吗？”\*\*

\*\* 答：\*\*“当然用过，HashMap是一种<key,value>的存储结构，能够快速将key的数据put方式存储起来，然后很快的通过get取出来”，然后说“HashMap不是线程安全的，  
HashTable是线程安全的，通过synchronized实现的。HashMap取值非常快”等等。这个时候说明他已经很熟练使用HashMap的工具了。

问：“你知道HashMap 在put和get的时候是怎么工作的吗？”

答：“HashMap是通过key计算出Hash值，然后将这个Hash值映射到对象的引用上，get的时候先计算key的hash值，然后找到对象”。这个时候已经显得不自信了。

问：“HashMap的key为什么一般用字符串比较多，能用其他对象，或者自定义的对象吗？为什么？”

答：“这个没研究过，一般习惯用String。”

问：“你刚才提到HashMap不是线程安全的，你怎么理解线程安全。原理是什么？几种方式避免线程安全的问题。”

答：“线程安全就是多个线程去访问的时候，会对对象造成不是预期的结果，一般要加锁才能线程安全。”

其实，问了以上那些问题，我基本能判定这个程序员的基本功了，一般技术中等，接下来的问题没必要问了。

从我的个人角度来看，HashMap的面试问题能够考察面试者的线程问题、Java内存模型问题、线程可见与不可变问题、Hash计算问题、链表结构问题、二进制的&、|、<<、>>等问题。所以一个HashMap就能考验一个人的技术功底了。

1. 什么时候会使用HashMap？他有什么特点？

是基于Map接口的实现，存储键值对时，它可以接收null的键值，是非同步的，HashMap存储着Entry(hash, key, value, next)对象。

2. 你知道HashMap的工作原理吗？

通过hash的方法，通过put和get存储和获取对象。存储对象时，我们将K/V传给put方法时，它调用hashCode计算hash从而得到bucket位置，进一步存储，HashMap会根据当前bucket的占用情况自动调整容量(超过Load Facotr则resize为原来的2倍)。获取对象时，我们将K传给get，它调用hashCode计算hash从而得到bucket位置，并进一步调用equals()方法确定键值对。如果发生碰撞的时候，Hashmap通过链表将产生碰撞冲突的元素组织起来，在Java 8中，如果一个bucket中碰撞冲突的元素超过某个限制(默认是8)，则使用红黑树来替换链表，从而提高速度。

3. 你知道get和put的原理吗？equals()和hashCode()的都有什么作用？

通过对key的hashCode()进行hashing，并计算下标( n-1 & hash)，从而获得buckets的位置。如果产生碰撞，则利用key.equals()方法去链表或树中去查找对应的节点

4. 你知道hash的实现吗？为什么要这样实现？

在Java 1.8的实现中，是通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在bucket的n比较小的时候，也能保证考虑到高低bit都参与到hash的计算中，同时不会有太大的开销。

5. 如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？

如果超过了负载因子(默认0.75)，则会重新resize一个原来长度两倍的HashMap，并且重新调用hash方法。

6. HashMap 的 key 为什么一般用字符串比较多，能用其他对象，或者自定义的对象吗？为什么？

能用其他对象，必须是immutable的，但是自实现的类必须Override两个方法：equals()和hashCode()。否则会调用默认的Object类的对应方法。

← Code

HashMap的key一般用字符串,能用其他对象吗？

作者：Java蚂蚁 / 公众号： 发布时间：2018-09-03

面试官：我们都知道在使用HashMap时，经常使用String类型或者Integer类型作为key，那我们可以使用自定义的对象吗？

应聘者：可以，能用其他对象，必须是immutable的，但是自实现的类必须Override两个方法：equals()和hashCode()。因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象；

面试官：那为什么String类型天生就可以做为键值呢？

应聘者：因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了；

可变对象

可变对象是指创建后自身状态能改变的对象。换句话说，可变对象是该对象在创建后它的哈希值可能被改变。

immutable不可变类：所谓的不可变类是指这个类的实例一旦创建完成后，就不能改变其成员变量值。如JDK内部自带的很多不可变类：Interger、Long和String等。

可变类：相对于不可变类，可变类创建实例后可以改变其成员变量值，开发中创建的大部分类都属于可变类。

String, Interger这样的类作为HashMap的键是再适合不过了，而且String最为常用。

　　因为String对象是不可变的，而且已经重写了equals()和hashCode()方法了。

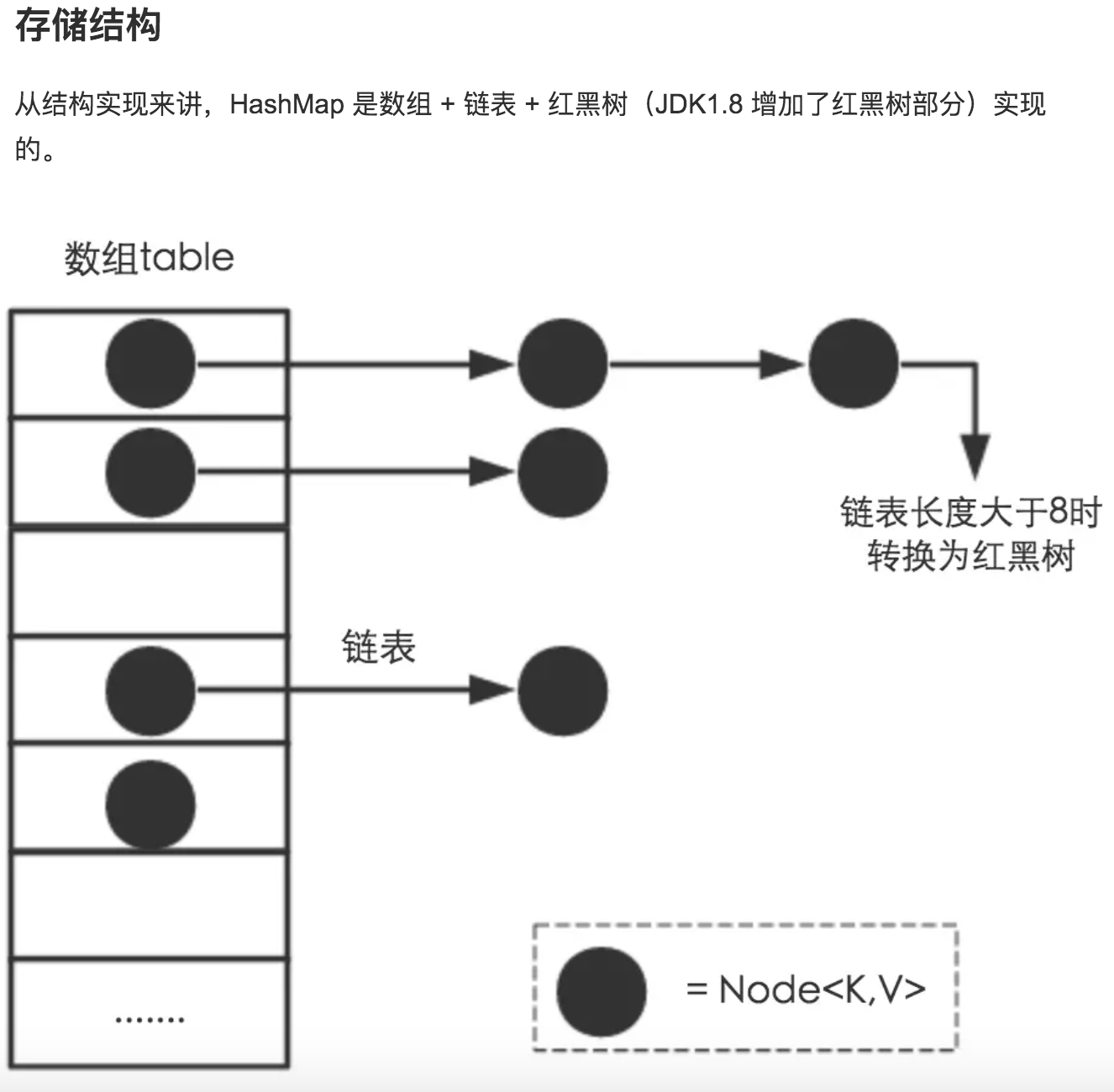
　　1.不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。

　　注：String的不可变性可以看这篇文章《【java基础】浅析String》。

　　2.因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能。

HashMap 最多只允许一条记录的键为 null，允许多条记录的值为 null。

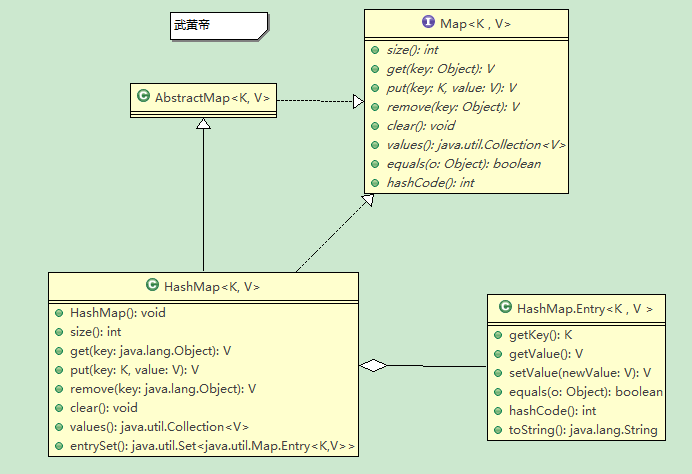
HashMap 非线程安全，即任一时刻可以有多个线程同时写 HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections 的 synchronizedMap 方法使 HashMap 具有线程安全的能力，或者使用 ConcurrentHashMap。

  
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

二、概念分析

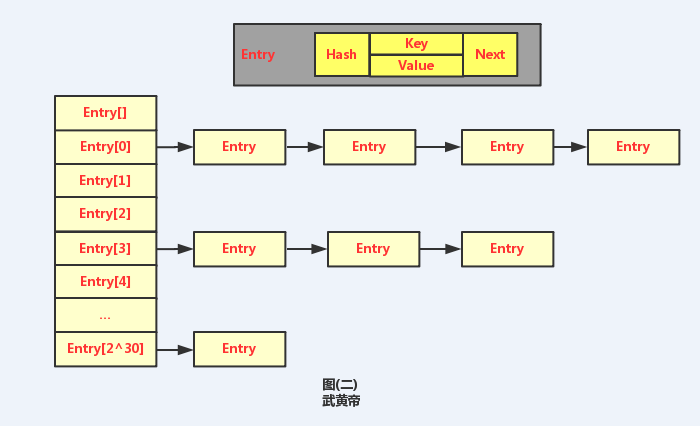
1、HashMap的类图结构

　此处的类图是根据JDK1.6版本画出来的。如下图1:



2、HashMap存储结构

\*\* \*\*HashMap的使用那么简单，那么问题来了，它是怎么存储的，他的存储结构是怎样的，很多程序员都不知道，其实当你put和get的时候，稍稍往前一步，你看到就是它的真面目。其实简单的说HashMap的存储结构是由数组和链表共同完成的。如图：



从上图可以看出HashMap是Y轴方向是数组，X轴方向就是链表的存储方式。大家都知道数组的存储方式在内存的地址是连续的，大小固定，一旦分配不能被其他引用占用。它的特点是查询快，时间复杂度是O(1)，插入和删除的操作比较慢，时间复杂度是O(n)，链表的存储方式是非连续的，大小不固定，特点与数组相反，插入和删除快，查询速度慢。HashMap可以说是一种折中的方案吧。

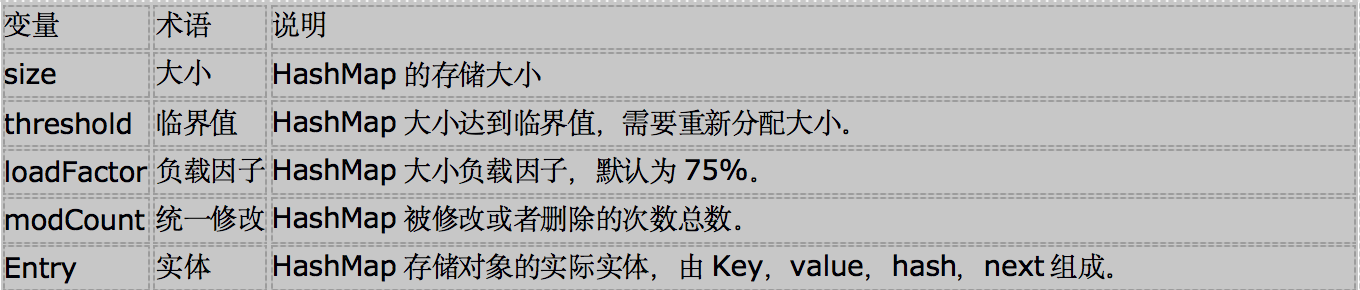
3、HashMap基本原理

1、首先判断Key是否为Null，如果为null，直接查找Enrty[0]，如果不是Null，先计算Key的HashCode，然后经过二次Hash。得到Hash值，这里的Hash特征值是一个int值。

2、根据Hash值，要找到对应的数组啊，所以对Entry[]的长度length求余，得到的就是Entry数组的index。

3、找到对应的数组，就是找到了所在的链表，然后按照链表的操作对Value进行插入、删除和查询操作。

4、HashMap概念介绍



5、HashMap初始化

默认情况下，大多数人都调用new HashMap()来初始化的，我在这里分析new HashMap(int initialCapacity, float loadFactor)的构造函数，代码如下：



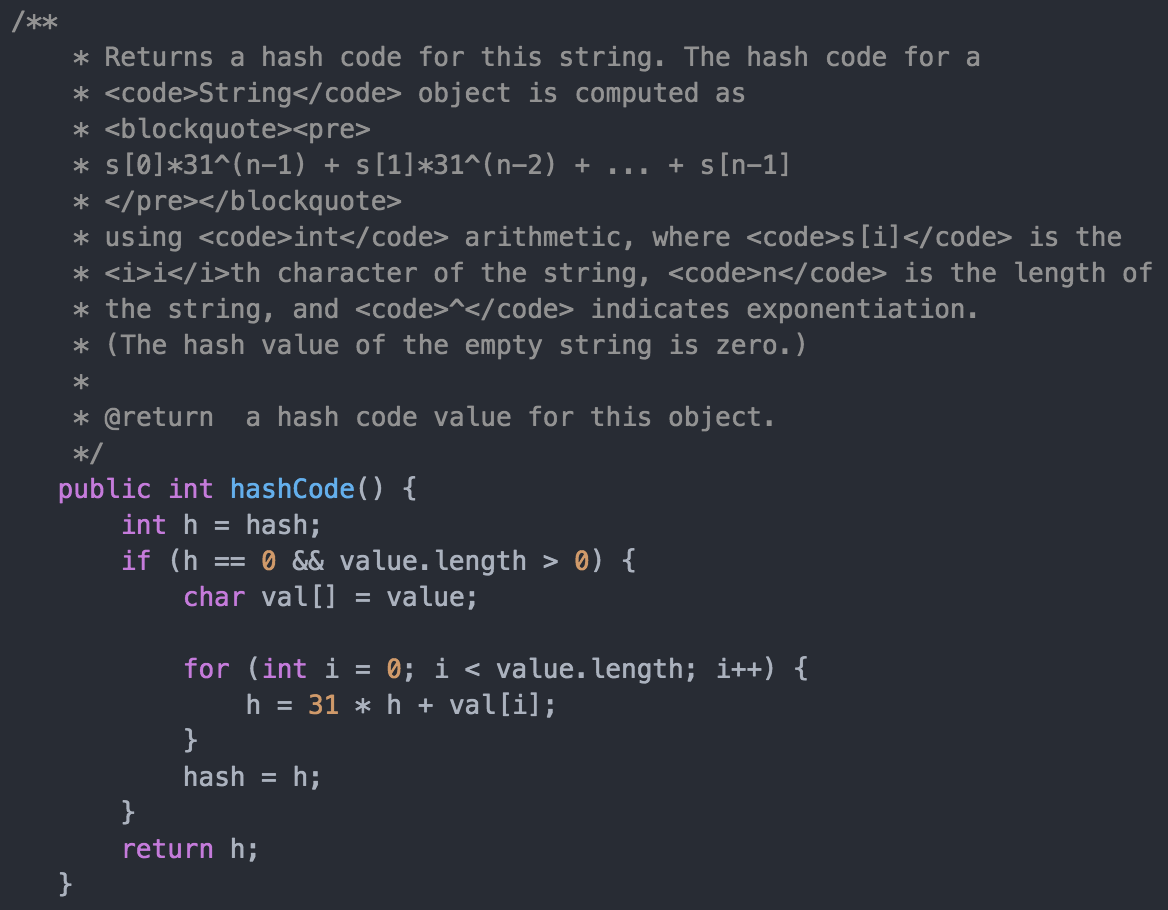
由上面的代码可以看出，初始化的时候需要知道初始化的容量大小，因为在后面要通过按位与的Hash算法计算Entry数组的索引，那么要求Entry的数组长度是2的N次方。

6、HashMap中的Hash计算和碰撞问题

HashMap的hash计算时先计算hashCode(),然后进行二次hash。代码如下：



先不忙着学习HashMap的Hash算法，先来看看JDK的String的Hash算法。代码如下：

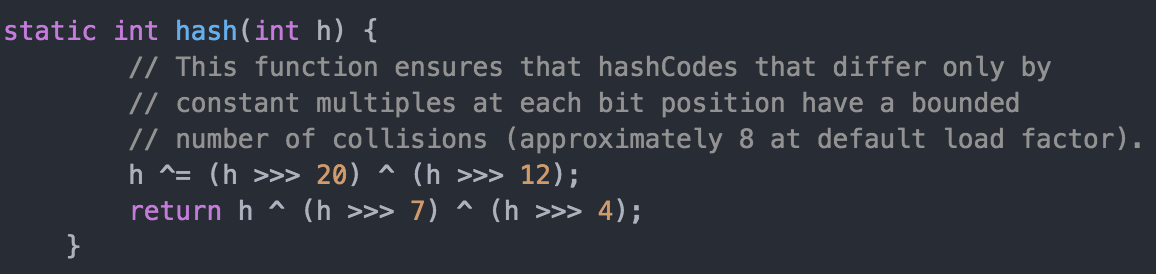


从JDK的API可以看出，它的算法等式就是s[0]31^(n-1) + s[1]31^(n-2) + ... + s[n-1]，其中s[i]就是索引为i的字符，n为字符串的长度。这里为什么有一个固定常量31呢，关于这个31的讨论很多，基本就是优化的数字，主要参考Joshua Bloch's [Effective Ja](https://link.jianshu.com?t=http://rads.stackoverflow.com/amzn/click/0321356683" \t "/Users/wjw/Documents\\x/_blank)va的引用如下：

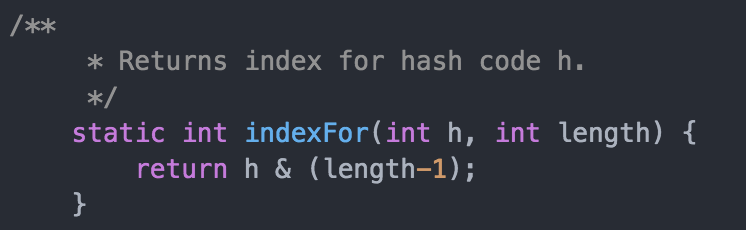
The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, as multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance: 31 \* i == (i << 5) - i. Modern VMs do this sort of optimization automatically.

大体意思是说选择31是因为它是一个奇素数，如果它做乘法溢出的时候，信息会丢失，而且当和2做乘法的时候相当于移位，在使用它的时候优点还是不清楚，但是它已经成为了传统的选择，31的一个很好的特性就是做乘法的时候可以被移位和减法代替的时候有更好的性能体现。例如31i相当于是i左移5位减去i，即31i == (i<<5)-i。现代的虚拟内存系统都使用这种自动优化。

现在进入正题，HashMap为什么还要做二次hash呢? 代码如下：



回答这个问题之前，我们先来看看HashMap是怎么通过Hash查找数组的索引的。



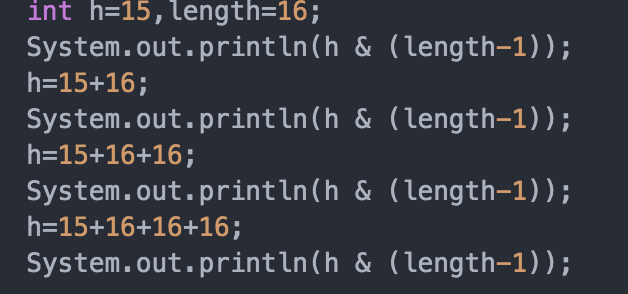
其中h是hash值，length是数组的长度，这个按位与的算法其实就是h%length求余，一般什么情况下利用该算法，典型的分组。例如怎么将100个数分组16组中，就是这个意思。应用非常广泛。

可以看到这个函数大概的作用就是：高16bit不变，低16bit和高16bit做了一个异或。其中代码注释是这样写的：

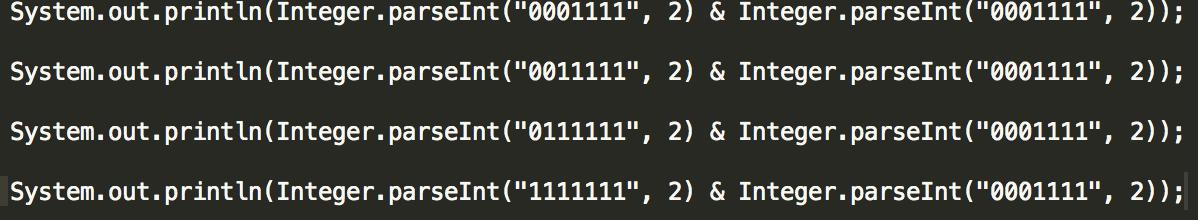
因此，设计者想了一个顾全大局的方法(综合考虑了速度、作用、质量)，就是把高16bit和低16bit异或了一下。设计者还解释到因为现在大多数的hashCode的分布已经很不错了，就算是发生了碰撞也用O(logn)的tree去做了。仅仅异或一下，既减少了系统的开销，也不会造成的因为高位没有参与下标的计算(table长度比较小时)，从而引起的碰撞。

如果还是产生了频繁的碰撞，会发生什么问题呢？作者注释说，他们使用树来处理频繁的碰撞(we use trees to handle large sets of collisions in bins)，在JEP-180中，描述了这个问题：

既然知道了分组的原理了，那我们看看几个例子，代码如下：

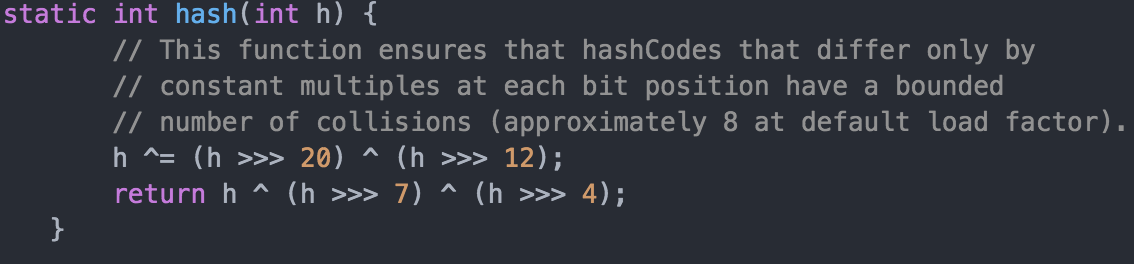


运行结果都是15，为什么呢?我们换算成二进制来看看。



这里你就发现了，在做按位与操作的时候，后面的始终是低位在做计算，高位不参与计算，因为高位都是0。这样导致的结果就是只要是低位是一样的，高位无论是什么，最后结果是一样的，如果这样依赖，hash碰撞始终在一个数组上，导致这个数组开始的链表无限长，那么在查询的时候就速度很慢，又怎么算得上高性能的啊。所以hashmap必须解决这样的问题，尽量让key尽可能均匀的分配到数组上去。避免造成Hash堆积。

回到正题，HashMap怎么处理这个问题，怎么做的二次Hash。



这里就是解决Hash的的冲突的函数，解决Hash的冲突有以下几种方法：

1. 开放定址法

线性探测再散列，二次探测再散列，伪随机探测再散列）

　2. 再哈希法

3. 链地址法

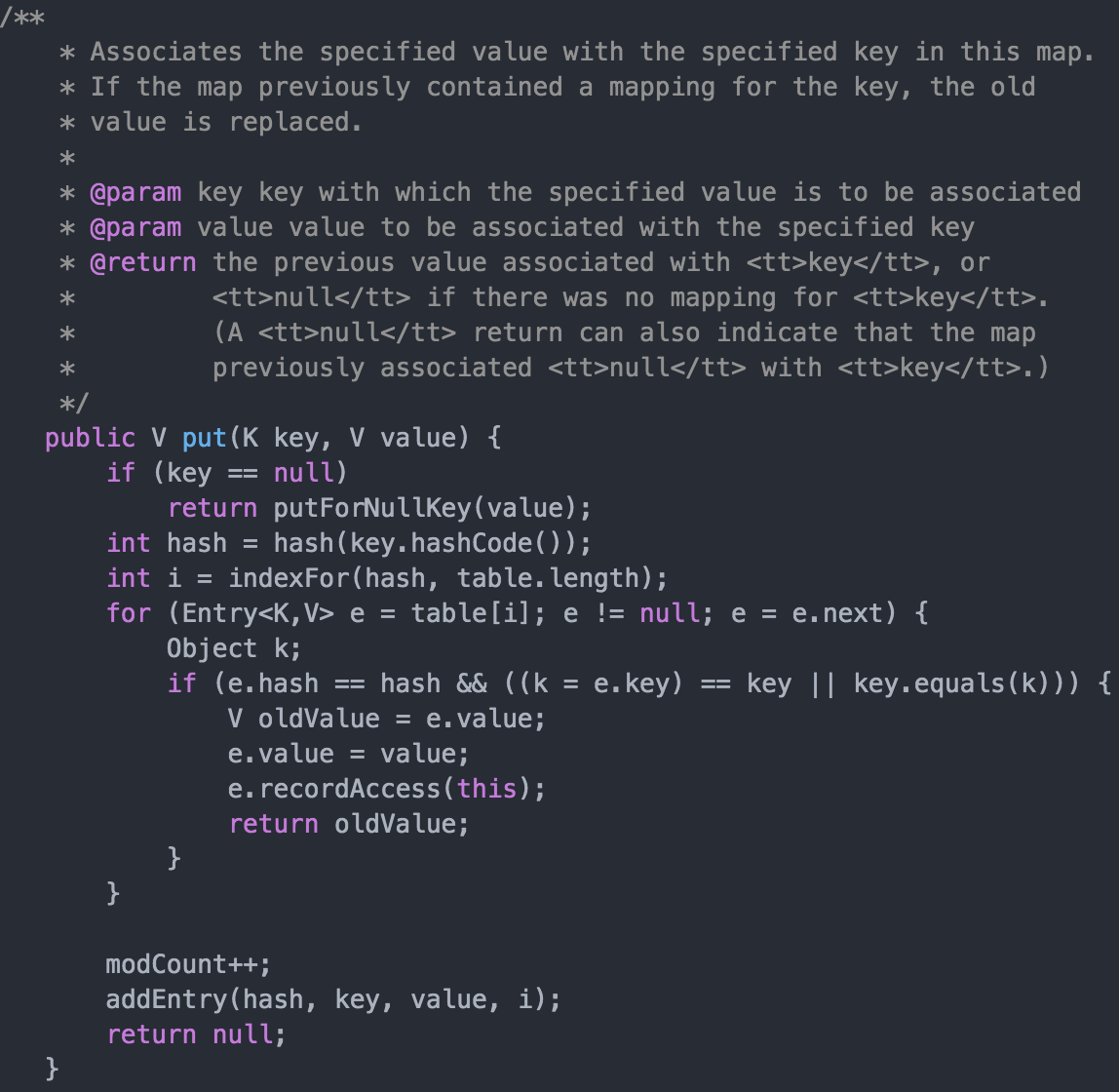
4. 建立一 公共溢出区

而HashMap采用的是链地址法，这几种方法在以后的博客会有单独介绍，这里就不做介绍了。

7、HashMap的put()解析

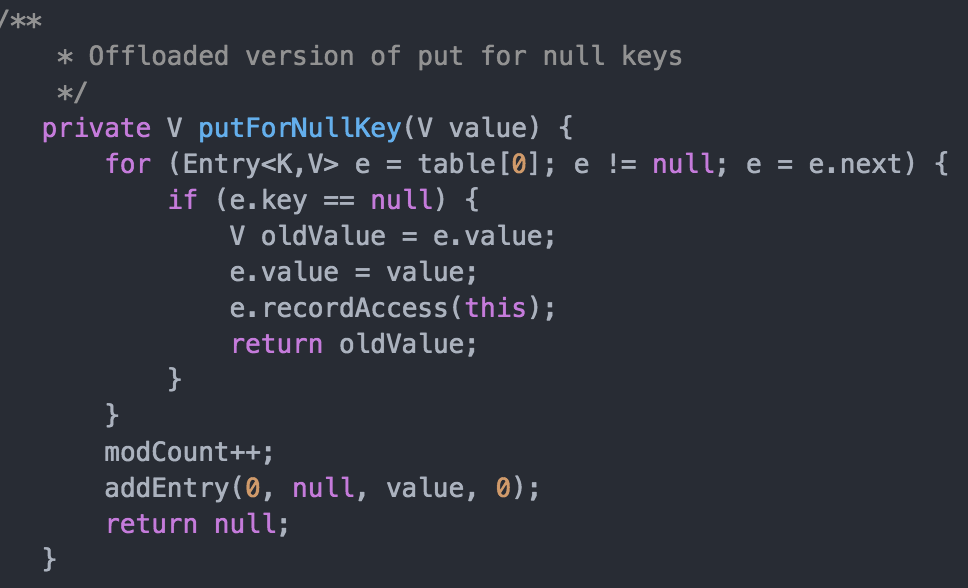
以上说了一些基本概念，下面该进入主题了，HashMap怎么存储一个对象的，代码如下：

系作者获得授权并注明出处。



从代码可以看出，步骤如下：

(1) 首先判断key是否为null，如果是null，就单独调用putForNullKey(value)处理。代码如下：



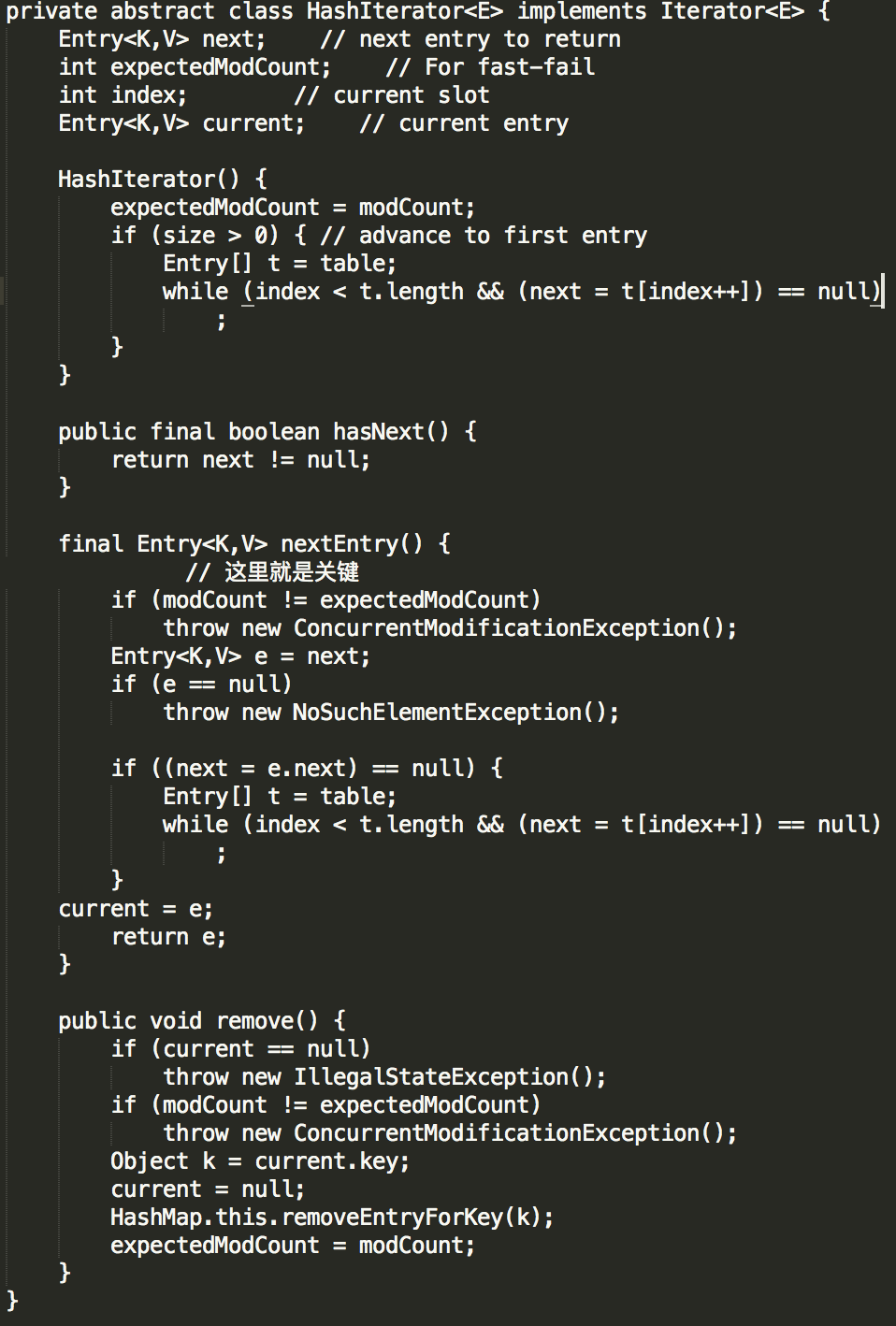
从代码可以看出，如果key为null的值，默认就存储到table[0]开头的链表了。然后遍历table[0]的链表的每个节点Entry，如果发现其中存在节点Entry的key为null，就替换新的value，然后返回旧的value，如果没发现key等于null的节点Entry，就增加新的节点。

(2) 计算key的hashcode，再用计算的结果二次hash，通过indexFor(hash, table.length);找到Entry数组的索引i。

(3) 然后遍历以table[i]为头节点的链表，如果发现有节点的hash，key都相同的节点时，就替换为新的value，然后返回旧的value。

(4) modCount是干嘛的啊? 让我来为你解答。众所周知，HashMap不是线程安全的，但在某些容错能力较好的应用中，如果你不想仅仅因为1%的可能性而去承受hashTable的同步开销，HashMap使用了Fail-Fast机制来处理这个问题，你会发现modCount在源码中是这样声明的。

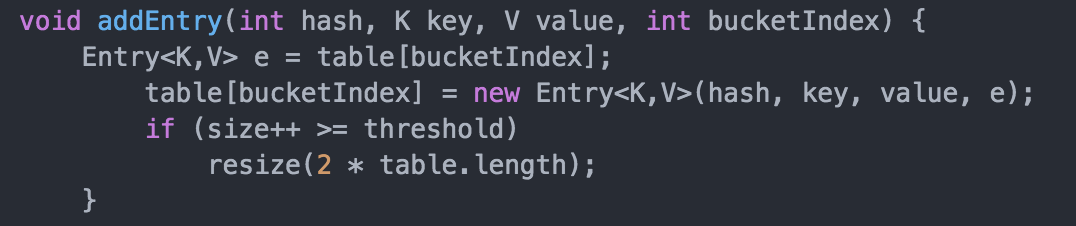
volatile关键字声明了modCount，代表了多线程环境下访问modCount，根据JVM规范，只要modCount改变了，其他线程将读到最新的值。其实在Hashmap中modCount只是在迭代的时候起到关键作用。



使用Iterator开始迭代时，会将modCount的赋值给expectedModCount，在迭代过程中，通过每次比较两者是否相等来判断HashMap是否在内部或被其它线程修改，如果modCount和expectedModCount值不一样，证明有其他线程在修改HashMap的结构，会抛出异常。

所以HashMap的put、remove等操作都有modCount++的计算。

(5) 如果没有找到key的hash相同的节点，就增加新的节点addEntry(),代码如下：

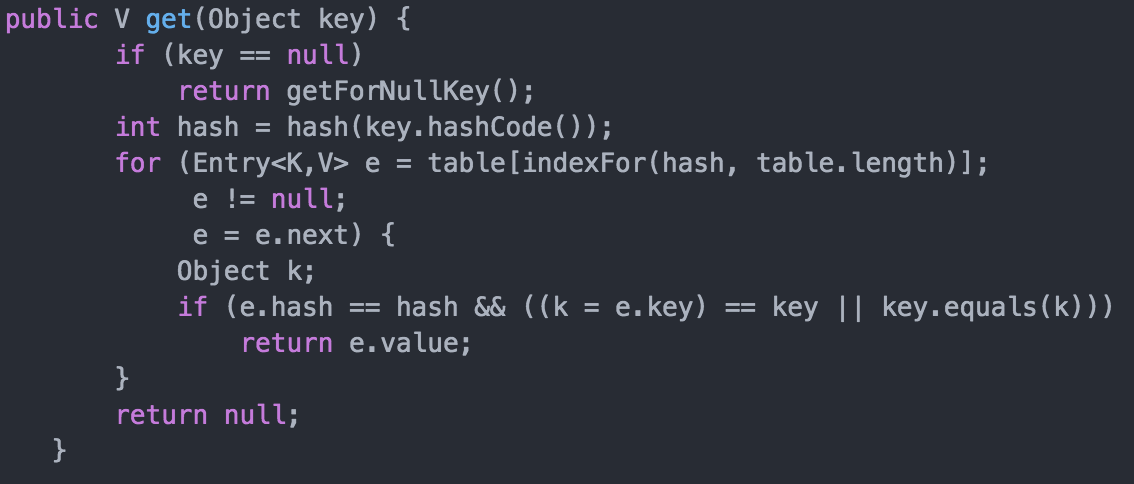


这里增加节点的时候取巧了，每个新添加的节点都增加到头节点，然后新的头节点的next指向旧的老节点。

(6) 如果HashMap大小超过临界值，就要重新设置大小，扩容，见第9节内容。

8、HashMap的get()解析

理解上面的put，get就很好理解了。代码如下：



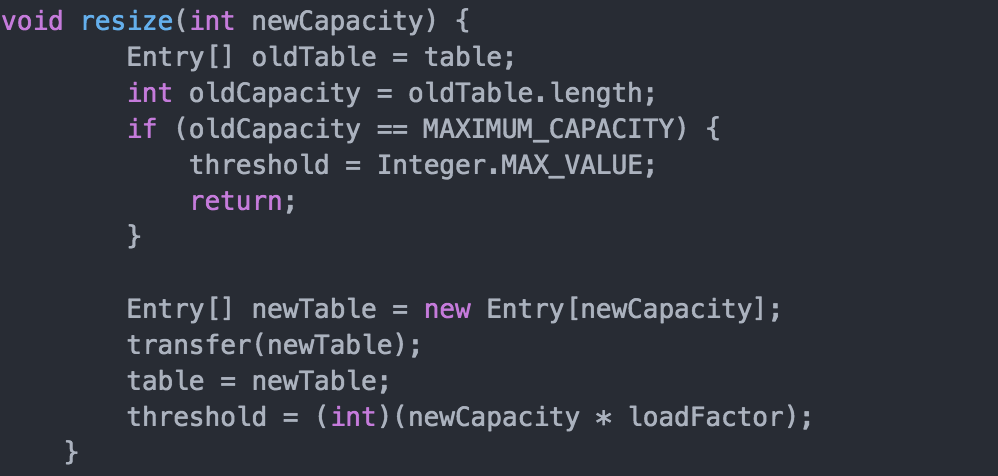
别看这段代码，它带来的问题是巨大的，千万记住,HashMap是非线程安全的，所以这里的循环会导致死循环的。为什么呢?当你查找一个key的hash存在的时候，进入了循环，恰恰这个时候，另外一个线程将这个Entry删除了，那么你就一直因为找不到Entry而出现死循环，最后导致的结果就是代码效率很低，CPU特别高。一定记住。

9、HashMap的size()解析

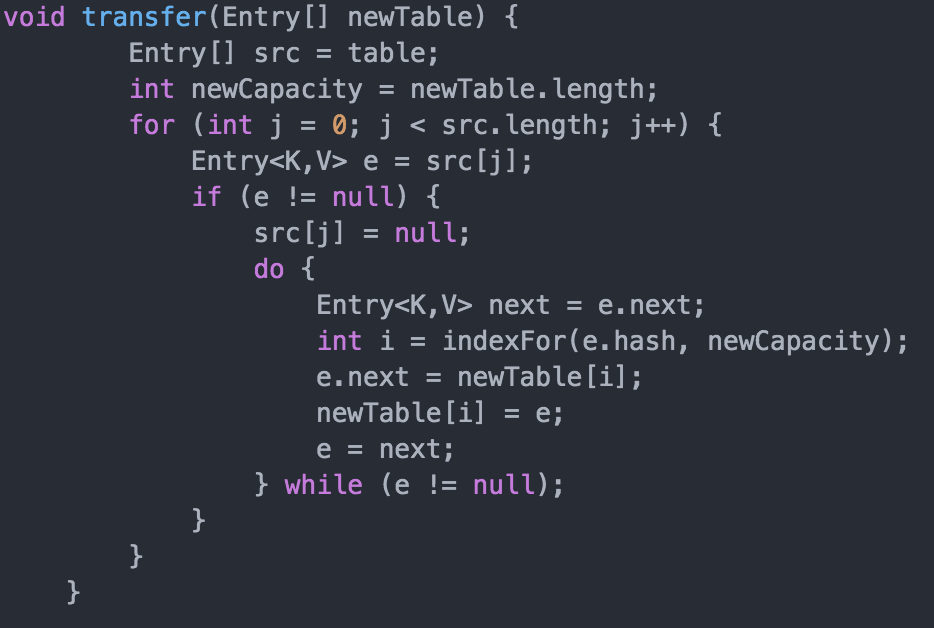
HashMap的大小很简单，不是实时计算的，而是每次新增加Entry的时候，size就递增。删除的时候就递减。空间换时间的做法。因为它不是线程安全的。完全可以这么做。效力高。

9、HashMap的reSize()解析

当HashMap的大小超过临界值的时候，就需要扩充HashMap的容量了。代码如下：



从代码可以看出，如果大小超过最大容量就返回。否则就new 一个新的Entry数组，长度为旧的Entry数组长度的两倍。然后将旧的Entry[]复制到新的Entry[].代码如下：



在复制的时候数组的索引int i = indexFor(e.hash, newCapacity);重新参与计算。

至此，HashMap还有一些迭代器的代码，这里不一一做介绍了，在JDK1.7版本中HashMap也做了一些升级，具体有Hash因子的参与。

今天差不多完成了HashMap的源码解析，下一步将会分析ConcurrencyHashMap的源码。ConcurrencyHashMap弥补了HashMap线程不安全、HashTable性能低的缺失。是目前高性能的线程安全的HashMap类。

很晚了，希望对大家有所帮助，晚安。