# HashMap实现原理（jdk1.6.0\_13）

HashMap是数组加链表结构，数组是靠table属性来维护的，数组中存放的是Entry类型。链表是靠Entry内部类来维护的，链表的头节点存放在table数组中。

# 1.说明

先看一下jdk api文档中类的说明。其中说的已经很清楚了HashMap是允许null键与null值的。



HashMap是线程不安全的。



HashMap不能保证元素的顺序，并且不能保证元素的顺序一直保持不变，因为HashMap中存储的Entry的数量达到一定的阀值，HashMap会自动调整Entry数组的大小，并且将元素转移到新的数组中。



# 常量

## 2.1默认初始容量与加载因子





下文构造器章节会讲到两个常量的用法。

## 2.2Entry数组最大容量



Entry数组的最大容量，Entry数组的长度不得超过这个值。

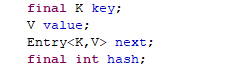
# 属性

HashMap有以下几个属性：

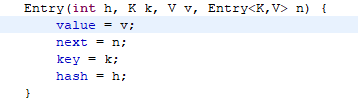
（1）Entry数组



Entry是HashMap中的内部类，key-value映射就是存储在这个内部类中，它有四个属性key，value，hash还有next，next指向下一节点。



Entry只有一个构造器。



（2）HashMap存储的元素的数量



（3）HashMap自动调整大小的阀值，当size超过这个值之后，HashMap会触发resize操作，重新调整table数组的大小。threshold=capacity\*load factor



（4）加载因子



（5）HashMap结构上修改的次数，当讲到迭代器时在谈这个属性

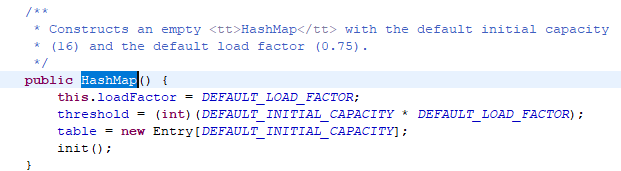


# 构造器

HashMap一共有四个构造器，下面展开来谈一下。

## 4.1默认构造器

看一下源码：



说明上已经说的很清楚了，使用默认的初始容量和默认的加载因子创建一个空的HashMap，这里用到了前面讲两个的常量DEFAULT\_INITIAL\_CAPACITY和DEFAULT\_LOAD\_FACTOR。init是一个模板方法。



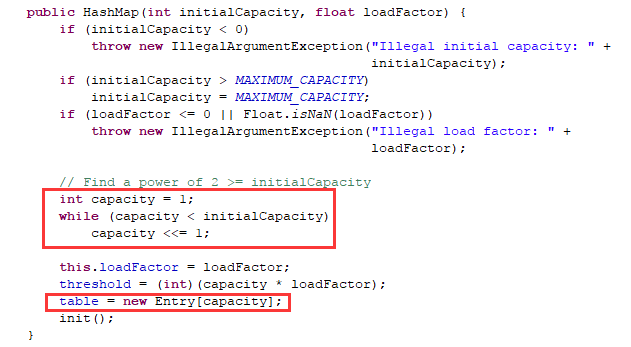
## 4.2HashMap(int)构造器



调用了HashMap(int,float)构造器，下面来看一个这个构造器。

## 4.3HashMap(int, float)构造器

贴一下源码：

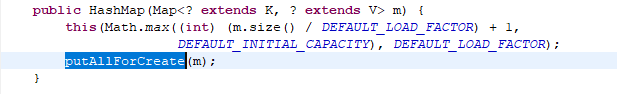


这段代码很好理解，注意红框圈起来的部分。无论你给定的数组初始容量是多少，最后初始化数组容量的一定是一个大于你给定的数组初始容量并且是2的次方的数。这里这样做的原因是减少hash的冲突，有兴趣的可一自行百度一下。

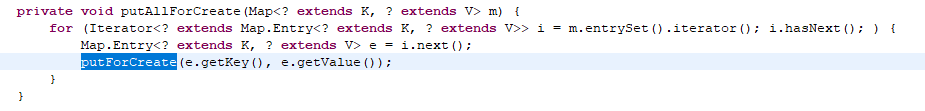
## 4.4HashMap(Map<? extends K, ? entends V>)构造器

用一个给定的Map初始化HashMap。

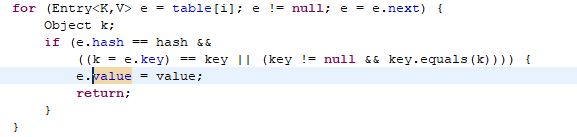
（1）首先调用HashMap(int, float)构造器初始化HashMap，然后调用putAllForCreate方法，接下来看一下putAllForCreate方法。



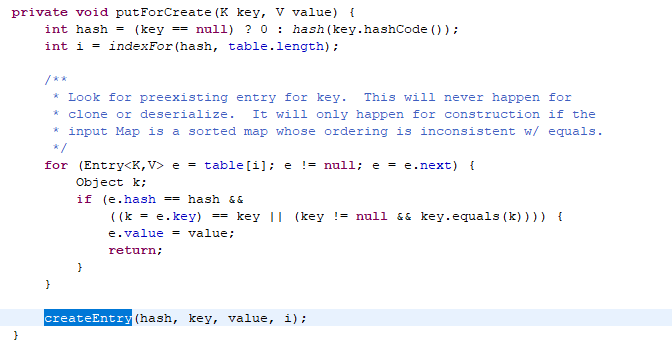
（2）putAllForCreate方法遍历Map所有的Entry元素，调用putForCreate方法。



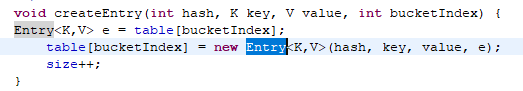
1. putForCreate方法首先根据key获取键的hash值并调用indexFor方法获取当前key应该存放在数组中的位置i，如果table[i]不为null，则遍历table[i]；为null，调用createEntry方法。以下是遍历table[i]的代码：



如果当前的key有存储到链表中， 则更新value值并return，否则调用createEntry方法。

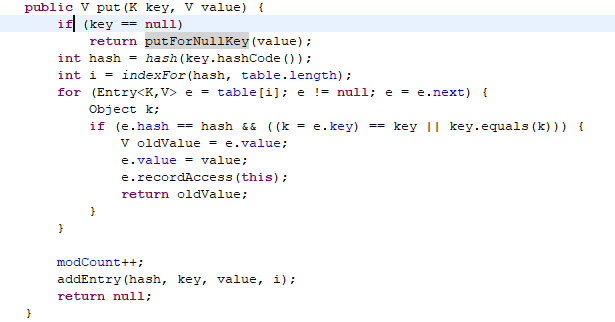


（4）createEntry方法创建了一个新的Entry对象将table[i]指向它并将next指向原table[i]，然后修改size。

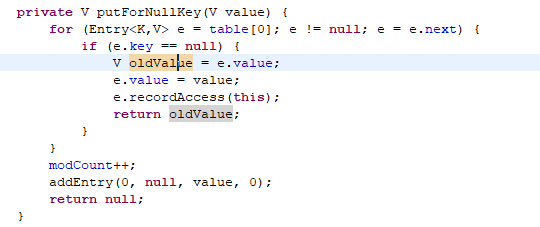


# 基本方法

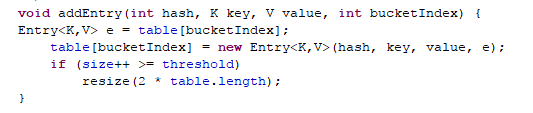
## 5.1put



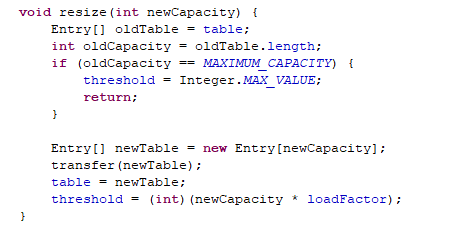
put首先判断key是否为null，为null调用putForNullKey方法，这个方法做了什么操作呢？下面来看一下：



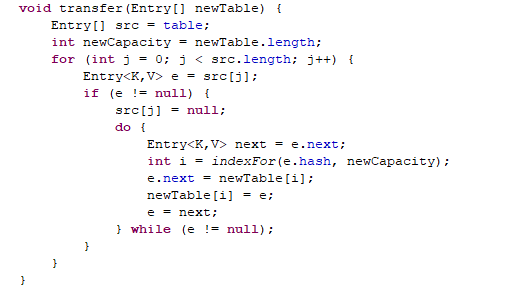
遍历table[0]，查找key为null的元素节点，找到则更新value，跳出方法，否则修改modCount，调用addEntry方法。这里可以看出，HashMap的实现默认是将null键存储在table[0]中的。modCount之后在谈，recordAccess方法在HashMap$Entry默认是空实现，在LinkedHashMap$Entry中得到了具体实现。



addEntry与上面讲的createEntry方法差不多，只不过在addEntry方法中当size超过阀值之后，会用当前数组容量的两倍来扩容HashMap。下面看下扩容算法。



当当前数组长度等于最大容量设置的时候，将阀值设置为Integer最大值并跳出方法；小于最大容量设置的时候，新建一个Entry数组，并调用transfer方法将当前数组中的元素转移到新数组中，并将table指向新数组，重新设置阀值。



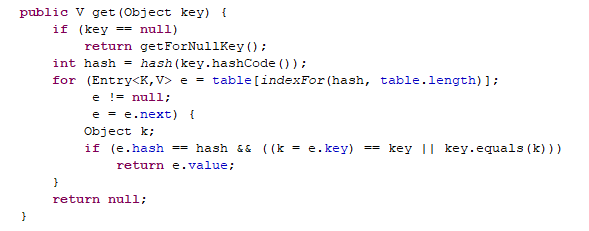
我们在回到put方法中，如果key不为null，通过hash算法计算key的hash值，通过indexFor获取key所应存放的数组的位置i。



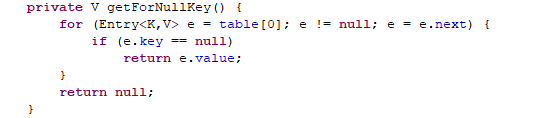
然后遍历table[i]，当前key如果存在就更新value否则调用addEntry。

## 5.2get

get方法相对简单。

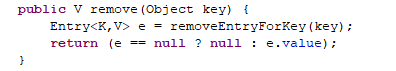


首先判断key是否为null，key为null则调用getForNullKey方法。

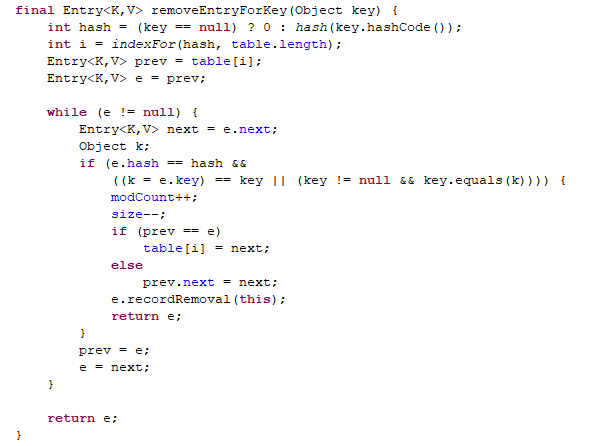


遍历table[0]，找到key为null的Entry并返回value否则返回null。回到get方法中。key不为null时，计算key的hash值，调用indexFor方法获取当前key所存储在数组中的位置，遍历，找到返回value否则返回null。

## 5.3remove



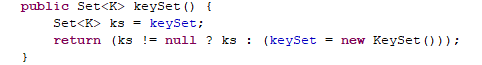
我们来看一下removeEntryForKey方法：

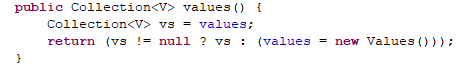


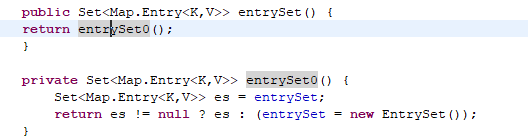
首先计算table数组的下标，找对对应的存储位置。prev表示当前元素的前驱，e表示当前元素，这里e=prev因为头节点没有前驱所以头节点的前驱为本身。然后遍历整个链表，next记录当前元素的后继，如果当前元素是所要查找的节点，修改modCount与size，然后判断当前元素是否为头节点，如果是头节点，则将table[i]指向后继，否则将前驱指向后继。如果当前元素不是所要查找的节点，则将prev指向当前节点，e指向当前节点的后继。如果没有找到，e最终指向的是最后一个节点的后继，即为null。这里考察到了链表的删除操作。

# HashMap的遍历

HashMap提供了三种遍历的方法keySet，values，entrySet，这三种方法都有相似之处。







KeySet、Values、EntrySet是三个内部类，KeySet与EntrySet实现了AbstractSet，Values实现了AbstractCollection，AbstractSet实现了AbstractCollection，而AbstractCollection实现了Collection。我们都知道Collection是继承自Iterable，现在我们主要来看看这三个内部类中实现的iterator方法。

KeySet：



Valuse：

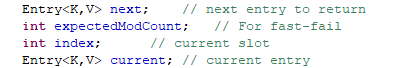


EntrySet：

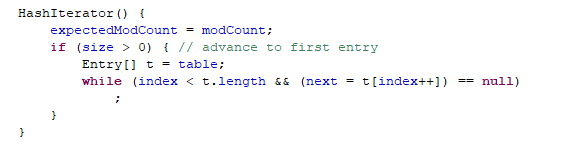


其中KeyIterator、ValueIterator与EntryIterator都实现了HashMapIteratro。现在HashMapIterator才是我们关注的重点，我们主要看一下HashMapIterator的属性，构造器与nextEntry方法。

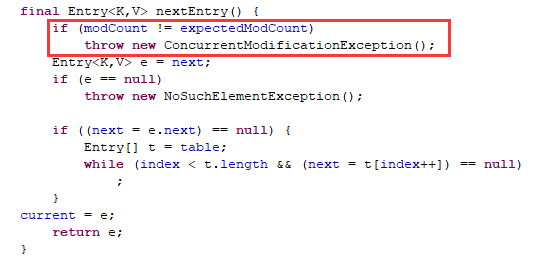
先看一下属性：注释说明的很清楚了，这里就不再赘述。



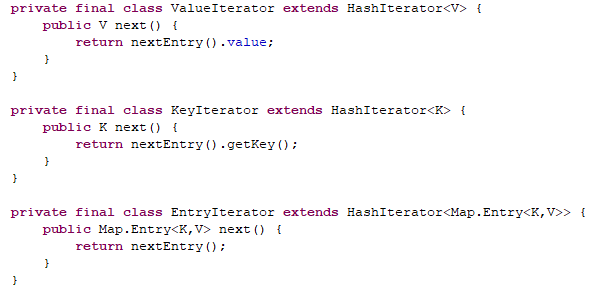
构造器，这里做得工作是找到第一个元素。



nextEntry方法，查找下一个元素。注意红框圈起来的部分，这里就用到了前文提到的modCount属性，expectedModCount是在构造器中初始化的，这里判断modCount与expectedModCount是否相等，如果不相等就抛出ConcurrentModificationException异常，这样为的是在遍历HashMap的过程中，防止有修改HashMap结构的情况，如果在遍历的过程中增加了元素，有可能新增加的元素遍历不到。这里就是快速失败，这里不在展开来讨论。

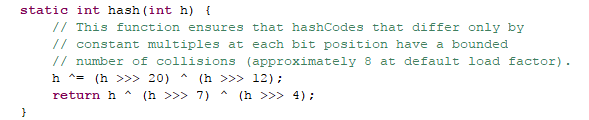


只所以讲nextEntry方法是因为在KeyIterator、ValueIterator与EntryIterator中next方法用到了。



# hash函数





先计算key的hashCode，然后做移位异或操作，最后&length-1，之所以做移位操作是为了让高低位都能得到充分的运算，不至于当遇到一组等差数列时，在与length-1做&操作时，总是计算得到固定的那几个位置，减少碰撞。

数组的长度为什么总是2的次方？因为长度为2的次方时候，后几位都是1，能够充分保证每一位都能参与运算。例如：当数组长度为15时候15-1为1110，最后一位为0，0&任意数都为0，将导致最后一位一直为0，0001、0011、0101、0111、1001、1011、1101这几个位置将永远不会有元素，造成空间的浪费。

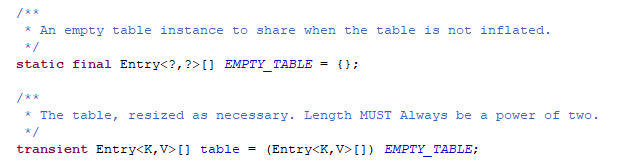
# HashMap实现原理（jdk1.7.0\_67）

我们来看一下在jdk1.7中HashMap所做得优化。

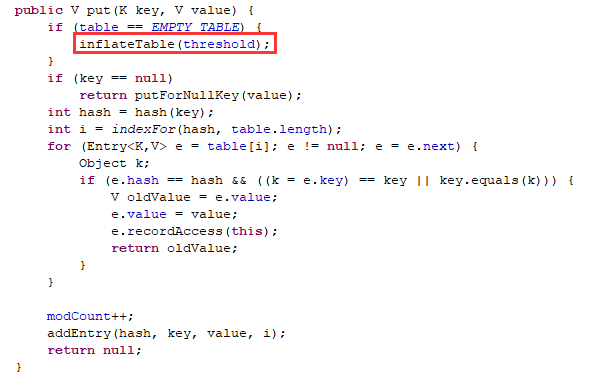
# jdk1.7的优化

## 8.1延迟table数组的初始化

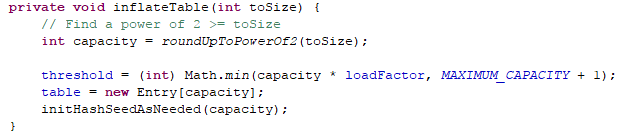
在HashMap1.7的实现中默认的将table属性指向了一个空的不变的Entry数组对象，如下所示：



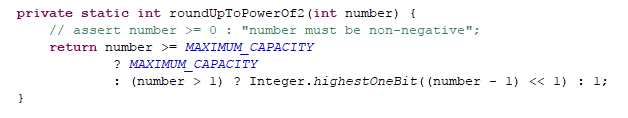
在1.6的实现中，table数组是在构造器中创建的，然而在1.7中table数组的创建放在了put方法中。



inflateTable这个方法做了两件事情，一是创建了table数组，二是生成了hashSeed，hashSeed主要是用来优化hash算法的，下节在讲hash算法的优化。



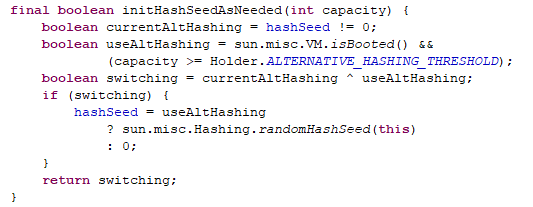
roundUpToPowerOf2这个方法的字面意思已经很明确了，找到大于等于最接近toSize的2的幂数。在这个方法中调用了Integer中的静态方法highestOneBit，highestOneBit的目的是保留最高位的1，将其余位置零。



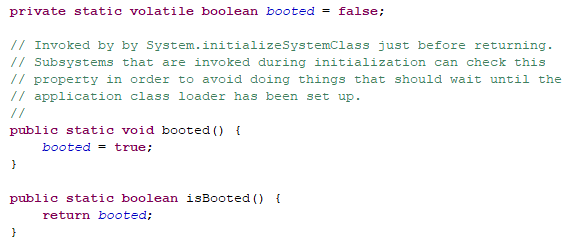
这里为什么使用number-1而不是number呢？因为当number正好为2的次幂时，如果使用number而不是number-1那么number << 1之后将原来的number扩大了两倍，最后得到的结果也是预计结果扩大两倍之后的。例如：number=16时，使用number-1，(16-1) << 1是11110，然后保留最高位，其余位置零，最后得到的结果为10000=16；使用number，最后得到的是32。

接下来看一下生成hashSeed的方法。

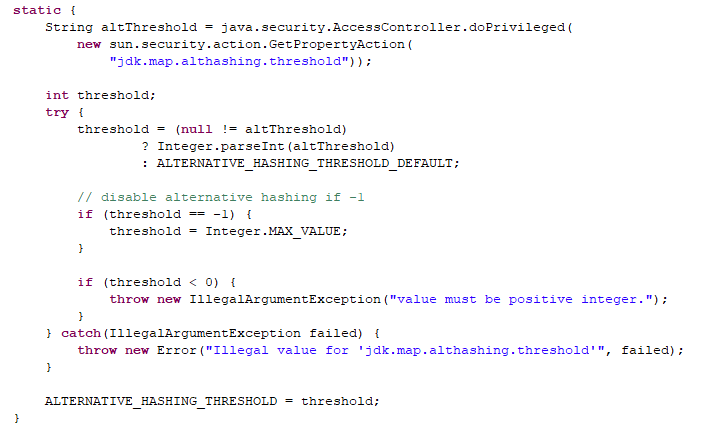




首先currentAltHashing = false。然后useAltHashing = isBooted() && capacity >= Holder.ALTERNATIVE\_HASHING\_THRESHOLD，我们先看一下isBooted()方法：



isBooted()返回booted，booted默认为false，但是在System.initializeSystemCLass()这个方法中调用了booted重新设置了booted值为true。再看一下capacity >= Holder.ALTERNATIVE\_HASHING\_THRESHOLD，在Holder静态代码块中初始化了ALTERNATIVE\_HASHING\_THRESHOLD。



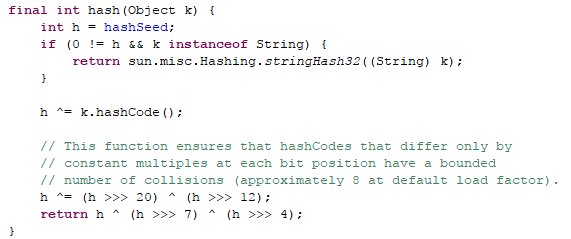
altThreshold是取得系统的环境变量jdk.map.althashing.threshold的值，默认为null，从而threshold的值为ALTERNATIVE\_HASHING\_THRESHOLD\_DEFAULT：



所以ALTERNATIVE\_HASHING\_THRESHOLD = threshold = Integer.MAX\_VALUE。最后回到initHashSeedAsNeeded，capacity >= Holder.ALTERNATIVE\_HASHING\_THRESHOLD为false，所以switching为false，hashSeed为0。

我理解的这个方法其实是在数组容量大于某个给定的阀值（jdk.map.althashing.threshold）的时候，使用hashSeed来优化hash算法。这里优化只针对String类型的key。

## 8.2hash算法的优化



这里hash算法的优化只针对String类型的key，前提的hashSeed不为0。