# Java知识点 手写HashMap

## 结构

HashMap是数组+链表（在JDK1.8中，如果链表长度大于8，则改为红黑树）。

如图：



HashMap里存的都是Node对象，Node的组成如图：



## 背景知识点

### 如何通过key，获取存储位置

首先通过key的hashCode值，然后无符号右移运算、按位异或运算。

#### 无符号右移

例：10>>>2=2

10的二进制1010

右移2位就是讲后两位去掉，前面补0，即

1010变成了0010，也就是2

同样的，在HashMap中，是对key的hashCode值>>>16，就是将hashCode的二进制后面16位去除，前面补0。

比如2987074的二进制就是1011011001010001000010

那么2987074>>>16就是去掉后面红色的16位，并在前面补0，即

0000000000000000101101

这个值就是45。

#### 按位异或

两个二进制相比较，相同位中，数字相同则结果为0，不同则结果为1。

例：1100^1010=0110

换算成十进制就是12^10=6

#### 与运算

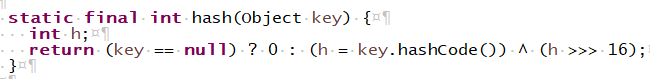
两个二进制相比较，相同位中，有一个0，则结果为0，否则为1

例：1100&1010=1000

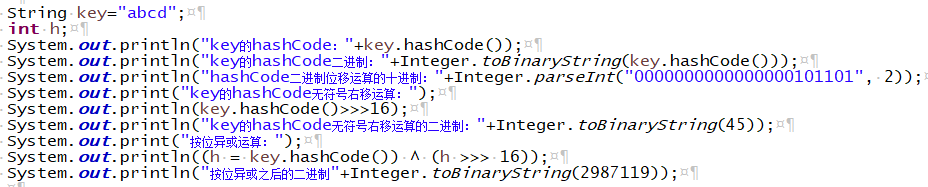
换算成十进制就是12&10=8。

#### hash()

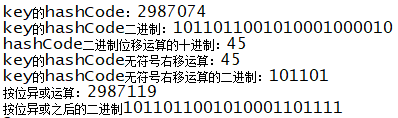
HashMap的hash方法如图



详细分析一下这几个运算



结果：



先获取了字符串abcd的hashCode为2987074

二进制为：1011011001010001000010

无符号右移为：0000000000000000101101

右移之后的十进制为：45

将两个二进制进行按位异或运算：1011011001010001101111

十进制就是2987119。

在HashMap中，如果key是null，则hash()之后的值是0，如果不是null，则本例中Key”abcd”的hash()就是2987119。

#### Key分组

我们知道Key是存在HashMap中的数组里的，具体存在数组的哪个位置是很重要的问题。试想一下，HashMap里有个容量为100的数组，我们往里put了1000个键值对。

最好的情况肯定数组每个元素对应的链表放10个键值对，这样查找起来最简单。

如果Key分配不好，把1000个键值对都放在了1个链表中，这查起来就太慢了。

分配的好：



分配的不好：



图二就是散列冲突的太严重了：所有Key经过散列之后获取的地址都一致。

为解决散列冲突，HashMap采用了链地址法，即：

**(length - 1) & hash**

因为在HashMap中，length总是2的N次方，所以以上方法等同于对length取模，即h%length，只是&比%效率高。

#### 总结

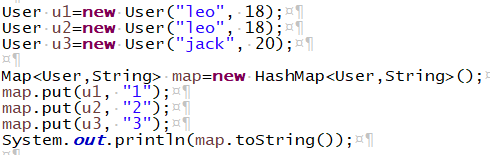
HashMap通过hash()方法（hashCode，无符号右移，按位异或）获取的值，进行除留余数法，将key尽可能均匀地分配到数组中。

## 对象做Key

有的面试官会问“对象能不能做Key”，答案是肯定的。

一般我们都是用int、String做Key，String就是对象，所以对象肯定能做Key。

但是，事情不是那么简单滴，看代码：



打印出的map是：



我们可以看到对象u1和u2在内容上是完全一致的，put进map里，按道理内容一样的Key在Map里只能有一个，可是在结果中却有俩。

为什么会出现这种情况？原因肯定是HashMap认为u1和u2是不一致的。

为什么不一致，结合上一篇文章对hash()这个方法的分析，肯定是u1和u2的hashCode不一致。



结果：



Object自带的hashCode()是一个本地方法。

如果要解决此问题，就必须重写自定义对象的hashCode()方法以及equals()方法，以达到相同内容的对象可以返回相同的hashCode。

common-lang包有两个类：HashCodeBuilder、EqualsBuilder可以帮助我们实现，具体不再赘述。

另外再看一个例子：



结果：



是不是很诡异？相同内容的两个对象，hashCode不同；不同的两个对象hashCode却是相同的。

如果这两个String做Key，会不会后者把前者冲突掉呢？答案是不会的，因为步骤是先比较两个对象的hashCode，如果hashCode不一样，那这两个对象肯定不一样——自定义对象必须重写hashCode方法。如果hashCode一样，是不能说明两个对象一致，就再需要equals()一下，这样才能分辨两个对象是否一致。

## modCount

这个字段在ArrayList、HashMap里都有，就是记录被修改、删除的次数。

为啥要有这个字段，记录这个次数有什么用？

我们知道ArrayList、HashMap是线程不安全的，线程安全的Vector、HashTable又太耗性能，我们还是希望能用线程不安全的ArrayList、HashMap，但是线程安全这个问题总得解决吧。这里采取了乐观锁的办法，modCount就是那个版本号。

当进行迭代的时候，先把modCount赋值给expectedModCount（期待中的modCount数值），每次迭代都会把modCount与expectedModCount对比一下。如果HashMap被修改了，modCount增加，与expectedModCount不一致就直接抛出异常ConcurrentModificationException。

## 扩容方法

先解释几个字段：

table，就是存链表头结点的数组。

threshold，容量的阈值，

loadFactor，负载因子，为0.75。

阈值就是容量\*负载因子，一旦我们put的键值对长度超过了阈值，就需要扩容，扩容之后是原容量的2倍。

MAXIMUM\_CAPACITY，最大容量，1左移30位，就是1073741824，二进制是1000000000000000000000000000000。

如果原table的长度大于这个值，则直接将threshold设置为Integer的最大值。因为是最大值了，以后也就不用扩容了。

扩容的步骤：

首先要确定新table的容量，一般情况下都是将新容量，新阈值\*2。

然后就是将原table里的内容复制到新table里。就是先遍历原数组，然后在里面遍历每个元素的链表，将所有链表都复制到新的数组里。

## put方法步骤

我们虽然都是用put(key,value)这个方法，但其内部主要是使用putVal()方法。

介绍一下putVal()的5个参数：

hash，hash值，就是通过这个值确定键值对放在数组哪个位置，公式是**(数组长度-1)&hash**。

key、value，没啥说的。

onlyIfAbsent，布尔值，如果是true，key值不存在的情况下才往里put，如果存在的话就不执行put操作，这样就不会修改原来的值了。

evict，布尔值，如果是false，则代表这是一个创建的操作。

说一下主要的步骤：

1. 如果table数组为空或长度为0，则执行初始化操作。
2. 根据hash值确定键值对要放的位置，如果这个位置上没有元素，则直接放进去。

另外，看过hash()代码我们应该知道，如果key是null，返回的hash就是0，所以key为null的键值对肯定放在table[0]这个位置。

再多说一下HashTable在put的代码里，先判断value，如果为null就抛出异常。然后在计算hash的时候直接用key的hashCode，可想而知，如果key为null是一定报异常的。

ConcurrentHashMap在put的代码里，首先就判断KV，如果二者有一个为空就抛异常。

所以HashTable、ConcurrentHashMap是不允许KV是null的。

1. 如果位置上有元素，则需要做详细的处理，这一段代码是HashMap的主代码。在JDK1.8中，如果链表的长度大于8，则将其转化为红黑树，这个回头写一篇讲讲，本例还是拿链表来说。

3.1、先判断链表的第一个键值对，如果新旧两个键值对的hash相同，并且Key相同（两个key==，或者在key不为null的情况下，新旧键值对的key的equals()是true），则说明新旧键值对的Key是相同的，直接替换。这部分的判断操作，可以看“对象做Key”这一节的说明。

3.2、如果第一个不匹配，就往下遍历链表，执行与上一步一样的判断Key的步骤，如果相同，把这个Node提出来，跳出循环，把Value替换成新值；如果不同就往下遍历，直到最后一个，都没有的话，就链表尾端新建一个。

3.3、最后如果size超过阈值，执行扩容操作。

## get方法步骤

了解了put方法，基本上也就很容易理解get方法的步骤了。最重要的还是判断key，如果新旧两个键值对的hash相同，并且Key相同（两个key==，或者在key不为null的情况下，新旧键值对的key的equals()是true），则说明这就是我们要获取的键值对。

步骤依然是定位所在数组的位置，获取第一个链表元素进行key校验，不同的话往下遍历链表进行key校验。

## remove方法步骤

remove是调用了removeNode()方法。步骤主要是两大块，第一步就是通过Key，先把要删除的元素找出来，这个跟get步骤基本一致，不再赘述。

第二步是删除。这里有两个地方要说一下，对应Key的Node已经取出来了，删不删还是有两种选择的，一种是只要Key对上了，不管Value是什么，都删；还有一种是，不仅Key要对上，Value也要对上，才执行删除操作。

因为是链表，所以删起来很简单方便。先获取链表第一个元素，如果这俩是一个，则直接把数组table此索引下的元素改为链表的next元素即可。

table[索引]=原Node.next;

原链表：



如果要删除的元素是第一个A，则：



Node的组成在第一篇里有描述。

在第一步找要删除的Node中，有一个临时的Node，在代码里是p，这个是存放要删除Node的前一个Node，在下面就用得上了。

如果第一个不是，则直接把P这个Node的next指向要删除Node的next。

即p.next=node.next;

原链表：



如果要删除的元素是C，则直接把B的next（原来指向C）改为C的next（就是D）：



这么看就清晰了。