https://mp.weixin.qq.com/s/lUdg-Hd2JwWEm\_VKkIrd4w

# HashMap（基于jdk1.8源码分析）

现在的面试当中凡是那些大厂，基本上都会问到一些关于HashMap的问题了，而且这个集合在开发中也经常会使用到。于是花费了大量的时间去研究分析写了这篇文章。本文是基于jdk1.8来分析的。篇幅较长，但是都是循序渐进的。耐心读完相信你会有所收获。

### 一、带着问题分析

这篇文章，希望能解决以下问题。

（1）HashMap的底层数据结构是什么？

（2）HashMap中增删改查操作的底部实现原理是什么？

（3）HashMap是如何实现扩容的？

（4）HashMap是如何解决hash冲突的？

（7）HashMap为什么是非线程安全的？

下面我们就带着这些问题，揭开HashMap的面纱。

### 二、认识HashMap

HashMap最早是在jdk1.2中开始出现的，一直到jdk1.7一直没有太大的变化。但是到了jdk1.8突然进行了一个很大的改动。其中一个最显著的改动就是：

之前jdk1.7的存储结构是数组+链表，到了jdk1.8变成了数组+链表+红黑树。

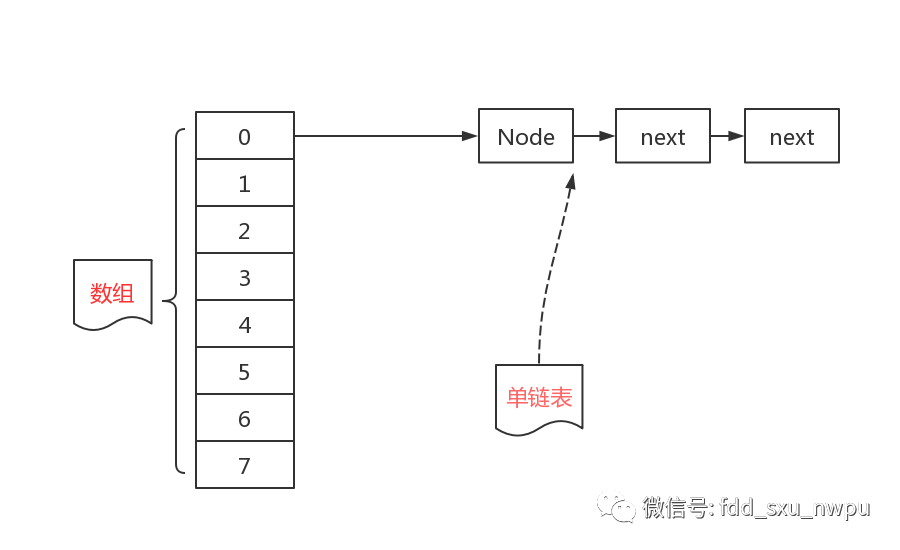
另外，HashMap是非线程安全的，也就是说在多个线程同时对HashMap中的某个元素进行增删改操作的时候，是不能保证数据的一致性的。

下面我们就开始一步一步的分析。

### 三、深入分析HashMap

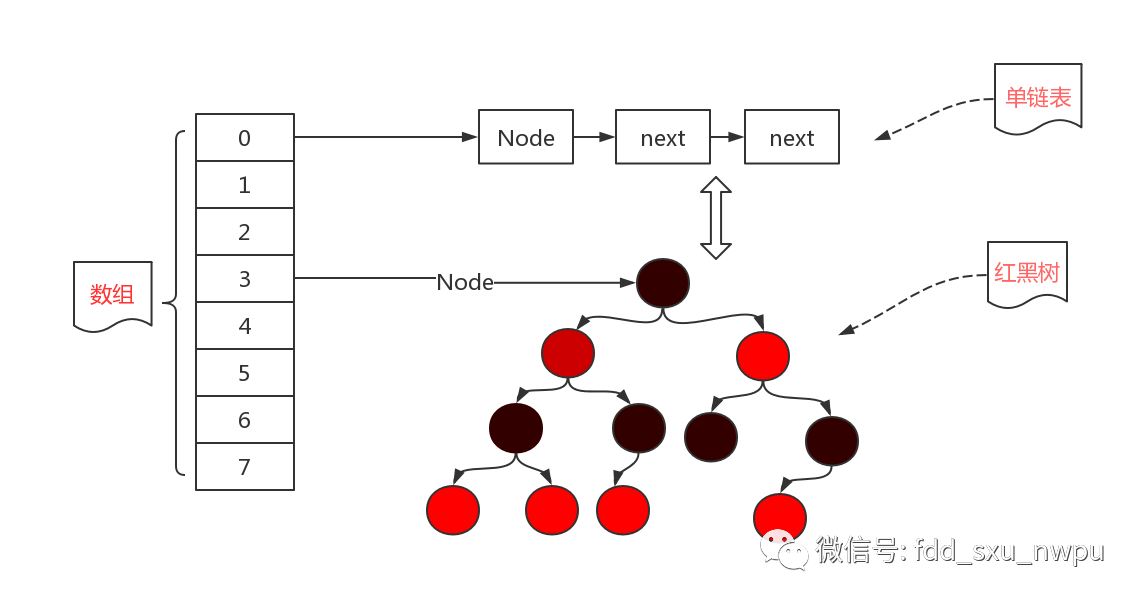
### 1、底层数据结构

为了进行一个对比分析，我们先给出一个jdk1.7的存储结构图



从上图我们可以看到，在jdk1.7中，首先是把元素放在一个个数组里面，后来存放的数据元素越来越多，于是就出现了链表，对于数组中的每一个元素，都可以有一条链表来存储元素。这就是有名的“拉链式”存储方法。

就这样用了几年，后来存储的元素越来越多，链表也越来越长，在查找一个元素时候效率不仅没有提高（链表不适合查找，适合增删），反倒是下降了不少，于是就对这条链表进行了一个改进。如何改进呢？就是把这条链表变成一个适合查找的树形结构，没错就是红黑树。于是HashMap的存储数据结构就变成了下面的这种。



我们会发现优化的部分就是把链表结构变成了红黑树。原来jdk1.7的优点是增删效率高，于是在jdk1.8的时候，不仅仅增删效率高，而且查找效率也提升了。

**注意：不是说变成了红黑树效率就一定提高了，只有在链表的长度不小于8，而且数组的长度不小于64的时候才会将链表转化为红黑树，**

#### ****问题一：什么是红黑树呢？****

红黑树是一个自平衡的二叉查找树，也就是说红黑树的查找效率是非常的高，查找效率会从链表的o(n)降低为o(logn)。如果之前没有了解过红黑树的话，也没关系，你就记住红黑树的查找效率很高就OK了。

#### **问题二：为什么不一下子把整个链表变为红黑树呢？**

这个问题的意思是这样的，就是说我们为什么非要等到链表的长度大于等于8的时候，才转变成红黑树？在这里可以从两方面来解释

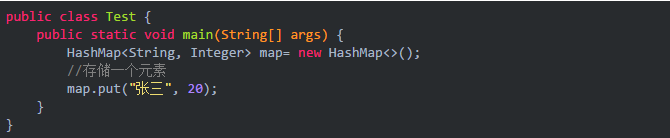
（1）构造红黑树要比构造链表复杂，在链表的节点不多的时候，从整体的性能看来， 数组+链表+红黑树的结构可能不一定比数组+链表的结构性能高。就好比杀鸡焉用牛刀的意思。

（2）HashMap频繁的扩容，会造成底部红黑树不断的进行拆分和重组，这是非常耗时的。因此，也就是链表长度比较长的时候转变成红黑树才会显著提高效率。

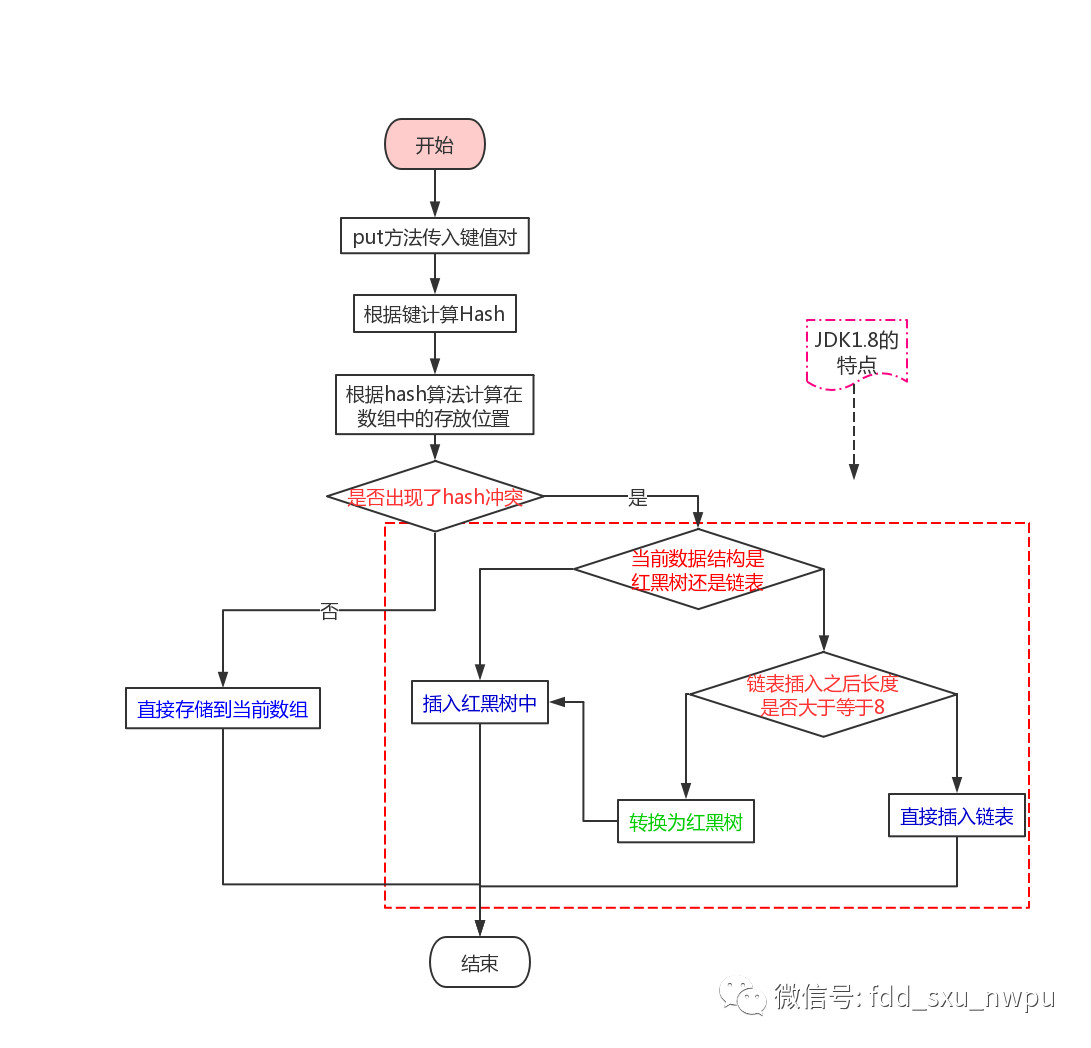
OK，到这里相信我们对hashMap的底层数据结构有了一个认识。现在带着上面的结构图，看一下如何存储一个元素。

### 2、存储元素

我们在存储一个元素的时候，大多是使用下面的这种方式。



在这里HashMap，第一个参数是键，第二个参数是值，合起来叫做键值对。存储的时候只需要调用put方法即可。那底层的实现原理是怎么样的呢？这里还是先给出一个流程图



上面这个流程，不知道你能否看到，红色字迹的是三个判断框，也是转折点，我们使用文字来梳理一下这个流程：

（1）第一步：调用put方法传入键值对

（2）第二步：使用hash算法计算hash值

（3）第三步：根据hash值确定存放的位置，判断是否和其他键值对位置发生了冲突

（4）第四步：若没有发生冲突，直接存放在数组中即可

（5）第五步：若发生了冲突，还要判断此时的数据结构是什么？

（6）第六步：若此时的数据结构是红黑树，那就直接插入红黑树中

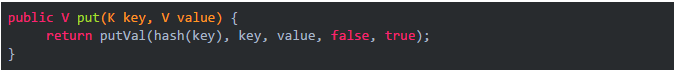
（7）第七步：若此时的数据结构是链表，判断插入之后是否大于等于8

（8）第八步：插入之后大于8了，就要先调整为红黑树，再插入

（9）第九步：插入之后不大于8，那么就直接插入到链表尾部即可。

上面就是插入数据的整个流程，光看流程还不行，我们还需要深入到源码中去看看底部是如何按照这个流程写代码的。

鼠标聚焦在put方法上面，按一下F3，我们就能进入put的源码。来看一下：



也就是说，put方法其实调用的是putVal方法。putVal方法有5个参数：

（1）第一个参数hash：调用了hash方法计算hash值

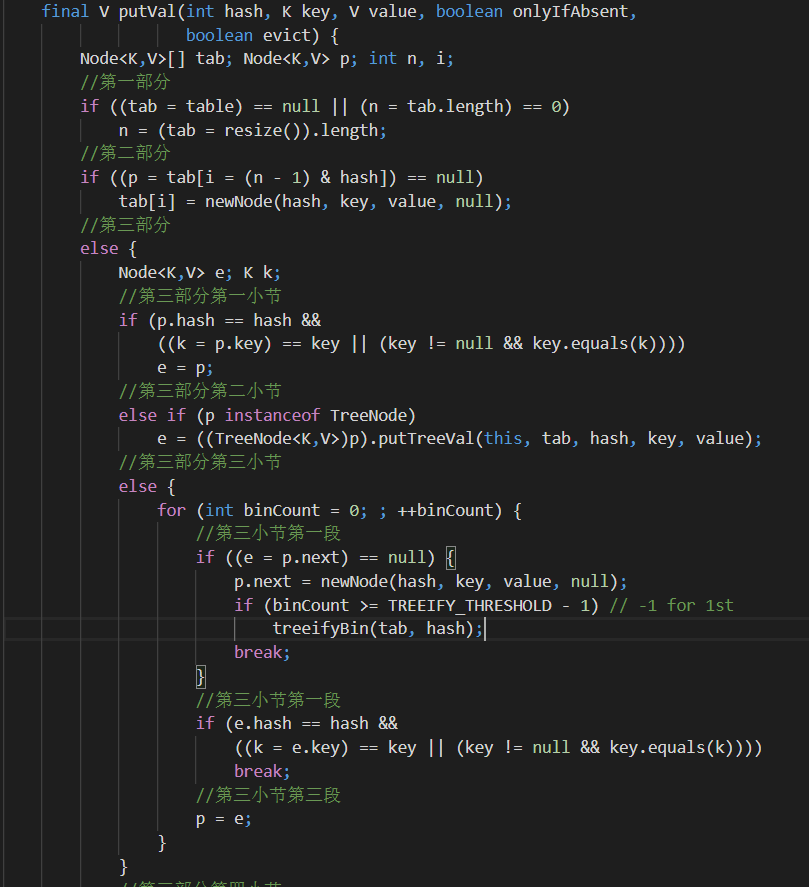
（2）第二个参数key：就是我们传入的key值，也就是例子中的张三

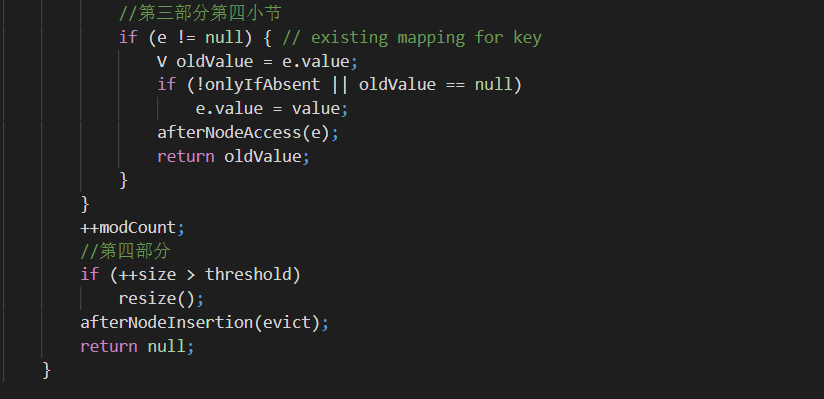
（3）第三个参数value：就是我们传入的value值，也就是例子中的20

（4）第四个参数onlyIfAbsent：也就是当键相同时，不修改已存在的值

（5）第五个参数evict ：如果为false，那么数组就处于创建模式中，所以一般为true。

知道了这5个参数的含义，我们就进入到这个putVal方法中。



乍一看，这代码完全没有读下去的欲望，第一次看的时候真实恶心到想吐，但是结合上一开始画的流程图再来分析，相信就会好很多。我们把代码进行拆分（整体分了四大部分）：

（1）Node[] tab中tab表示的就是数组。Nodep中p表示的就是当前插入的节点

（2）第一部分：



（3）第二部分：



i表示在数组中插入的位置，计算的方式为(n - 1) & hash。在这里需要判断插入的位置是否是冲突的，如果不冲突就直接newNode，插入到数组中即可，这就和流程图中第一个判断框对应了。

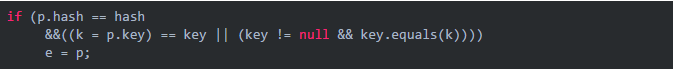
如果插入的hash值冲突了，那就转到第三部分，处理冲突

（4）第三部分：



我们会看到，处理冲突还真是麻烦，好在我们对这一部分又进行了划分

a）第三部分第一小节：



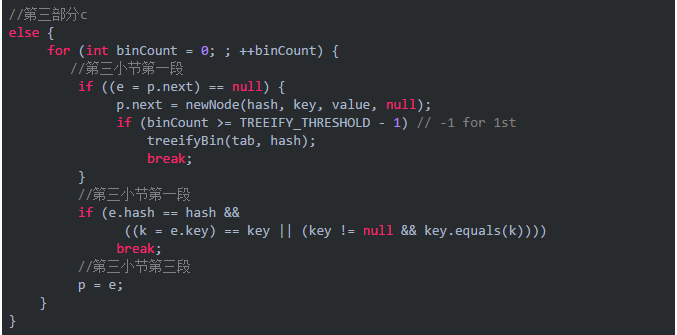
在这里判断table[i]中的元素是否与插入的key一样，若相同那就直接使用插入的值p替换掉旧的值e。

b）第三部分第二小节：



判断插入的数据结构是红黑树还是链表，在这里表示如果是红黑树，那就直接putTreeVal到红黑树中。这就和流程图里面的第二个判断框对应了。

c）第三部分第三小节：



如果数据结构是链表，首先要遍历table数组是否存在，如果不存在直接newNode(hash, key, value, null)。如果存在了直接使用新的value替换掉旧的。

注意一点：不存在并且在链表末尾插入元素的时候，会判断binCount >= TREEIFY\_THRESHOLD - 1。也就是判断当前链表的长度是否大于阈值8，如果大于那就会把当前链表转变成红黑树，方法是treeifyBin。这也就和流程图中第三个判断框对应了。

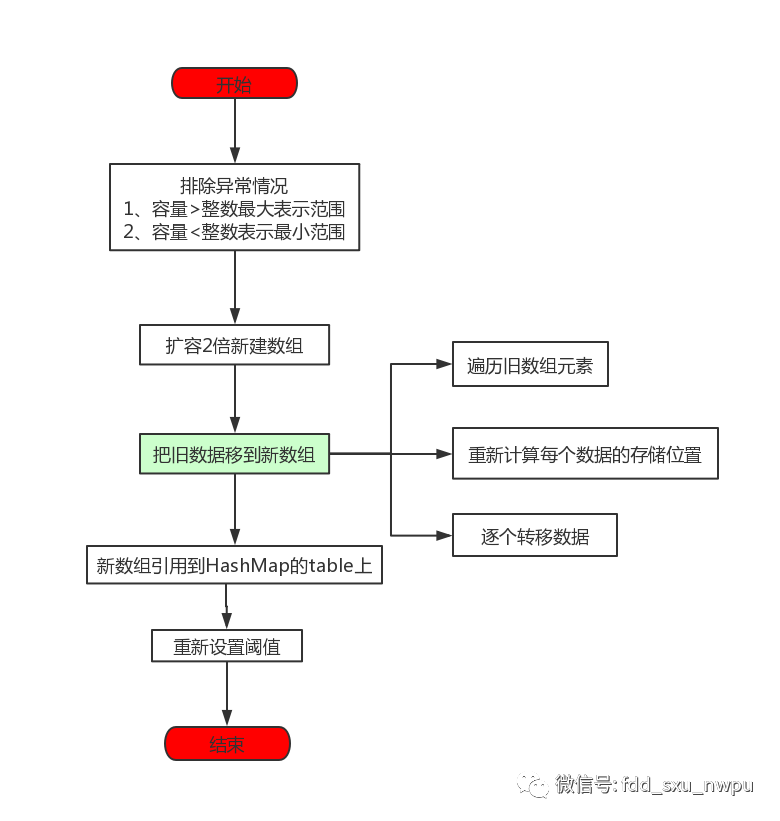
（5）第四部分：



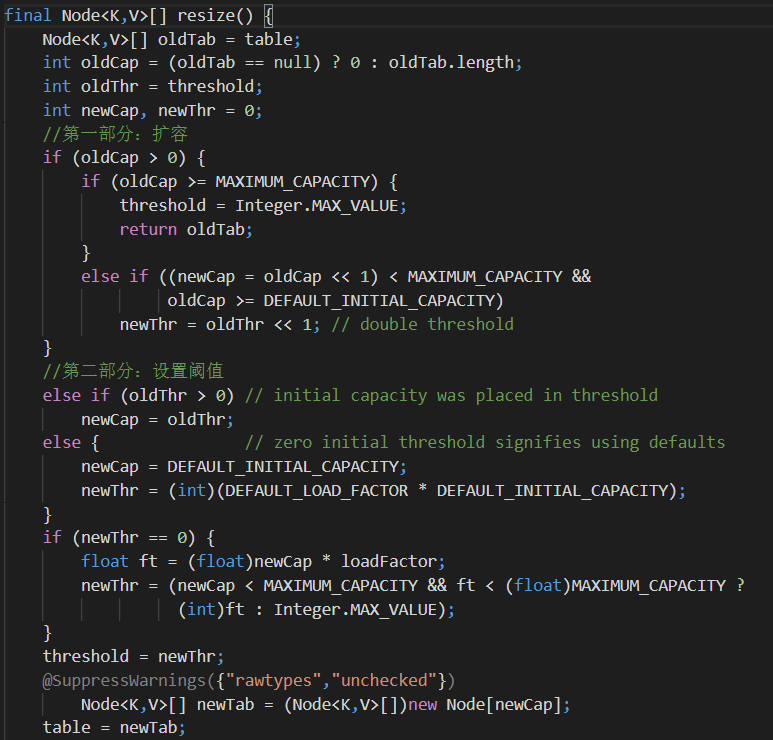
插入成功之后，还要判断一下实际存在的键值对数量size是否大于阈值threshold。如果大于那就开始扩容了。

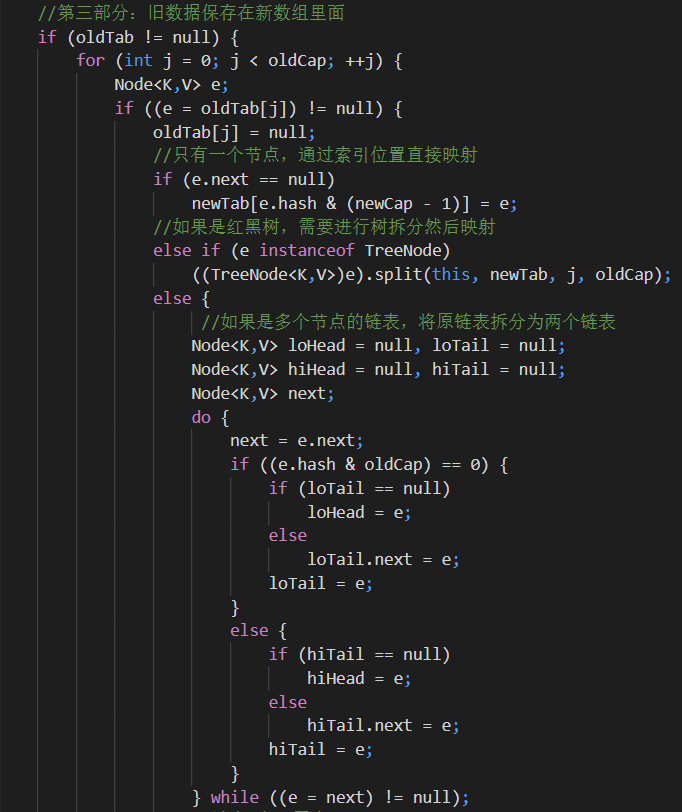
### 3、扩容

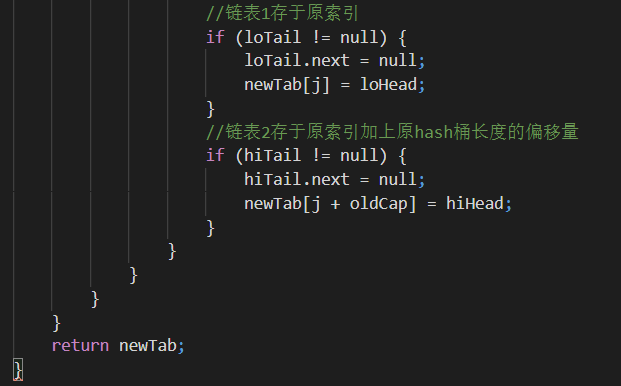
为什么扩容呢？很明显就是当前容量不够，也就是put了太多的元素。为此我们还是先给出一个流程图，再来进行分析。



这个扩容就比较简单了，HaspMap扩容就是就是先计算 新的hash表容量和新的容量阀值，然后初始化一个新的hash表，将旧的键值对重新映射在新的hash表里。如果在旧的hash表里涉及到红黑树，那么在映射到新的hash表中还涉及到红黑树的拆分。整个流程也符合我们正常扩容一个容量的过程，我们根据流程图结合代码来分析：

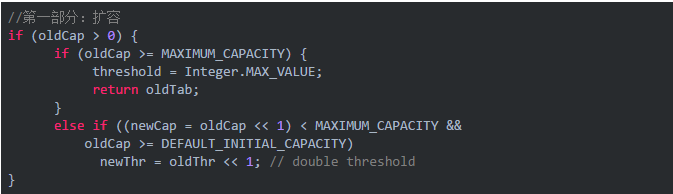






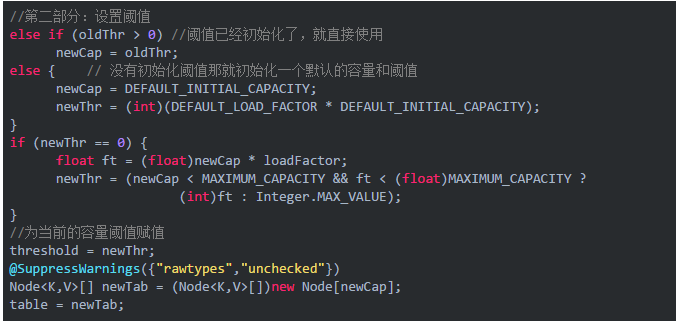
这代码量同样让人恶心，不过我们还是分段来分析：

（1）第一部分：



根据代码也能看明白：首先如果超过了数组的最大容量，那么就直接将阈值设置为整数最大值，然后如果没有超过，那就扩容为原来的2倍，这里要注意是oldThr << 1，移位操作来实现的。

（2）第二部分：



首先第一个else if表示如果阈值已经初始化过了，那就直接使用旧的阈值。然后第二个else表示如果没有初始化，那就初始化一个新的数组容量和新的阈值。

（3）第三部分

第三部分同样也很复杂，就是把旧数据复制到新数组里面。这里面需要注意的有下面几种情况：

A：扩容后，若hash值新增参与运算的位=0，那么元素在扩容后的位置=原始位置

B：扩容后，若hash值新增参与运算的位=1，那么元素在扩容后的位置=原始位置+扩容后的旧位置。

hash值新增参与运算的位是什么呢？我们把hash值转变成二进制数字，新增参与运算的位就是倒数第五位。

这里面有一个非常好的设计理念，扩容后长度为原hash表的2倍，于是把hash表分为两半，分为低位和高位，如果能把原链表的键值对， 一半放在低位，一半放在高位，而且是通过e.hash & oldCap == 0来判断，这个判断有什么优点呢？

举个例子：n = 16，二进制为10000，第5位为1，e.hash & oldCap 是否等于0就取决于e.hash第5 位是0还是1，这就相当于有50%的概率放在新hash表低位，50%的概率放在新hash表高位。

OK，到这一步基本上就算是把扩容这一部分讲完了，还有一个问题没有解决，也就是说存储的原理讲明白了，存储的元素多了如何扩容也明白了，扩容之后出现了地址冲突怎么办呢？