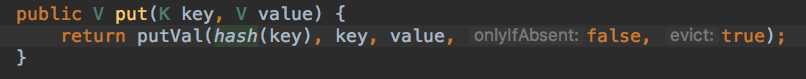
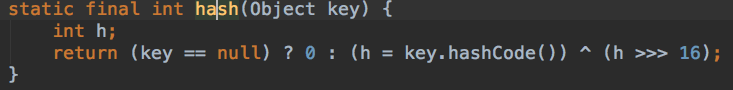
HashMap源码分析

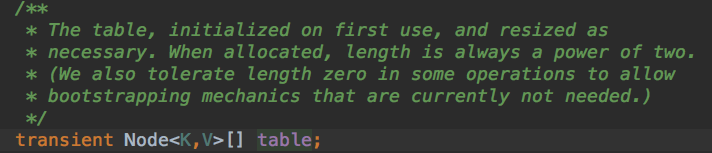
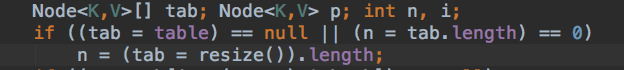
一、总述

HashMap是工作中经常使用的数据结构，也是面试中经常问到的知识点，掌握HashMap的工作原理，帮助更好的使用HashMap，并在面试中获得更多机会，下面我们从源码层面分析HashMap的工作原理。

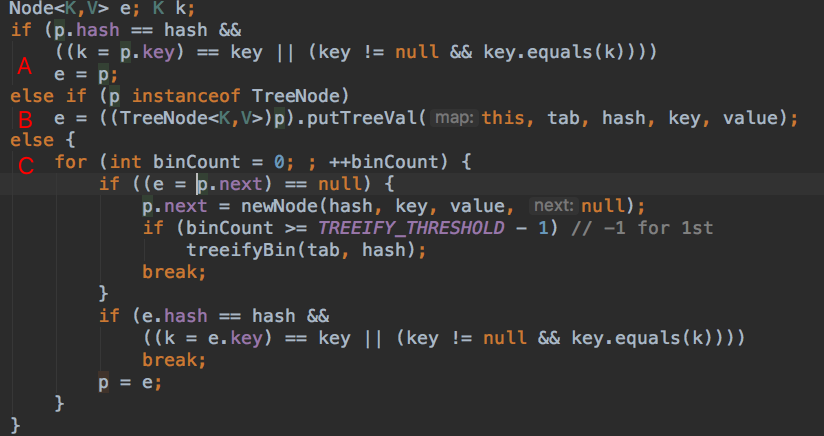
二、Put()

对单一功能的源码分析不仅会涉及到当前功能代码，还会辐射到相关的代码，如果能将一个主流程分析清楚，其他主流程也会更容易。

 进入put方法，其中hash()调用了key本身的hashCode方法，产生int，并将hashCode的高16位和低16位相互异或。原因是哈希表长度为2的幂次，因此hashCode高位值在计算元素放置的位点（index calculating）时被忽略，只有低于或等于哈希表长度掩码（高位全0，低位全1）的位才生效，因此要让高位参与决策，又要权衡时间、效果，异或是较好的选择。

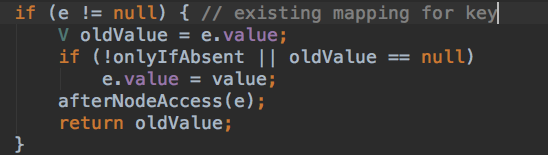
这里也可以看出，HashMap支持null作为key。

进入putVal方法，该方法存放KV，并返回被覆盖的value（如果key存在，否则返回null）。HashMap的本质是Node数组，Node中包含hash、key、value、next。本质上Node是链表元素。注释中写明，table在第一次使用时初始化，并在需要的时候扩容。另外不仅是HashMap中，JDK的很多代码中都会将成员变量先拷贝一份为局部变量，例如tab = table，原因是局部变量存放在栈中，成员变量存放在堆中，访问栈会更快。当初次访问或哈希表大小为0时进行扩容，即首次使用。扩容过程后续会分析【分支一】。

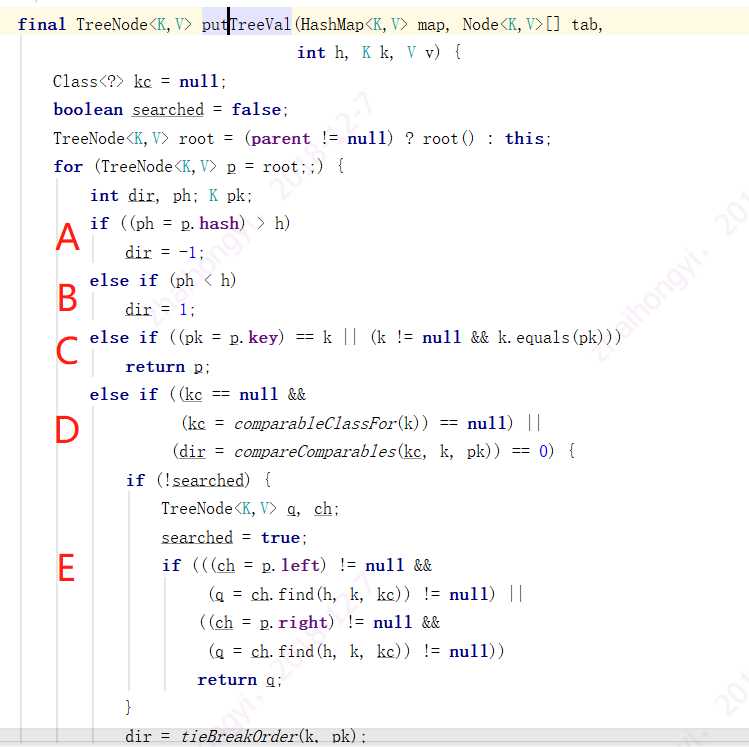
n - 1为哈希掩码，高位全0，低位全1，与操作，只保留hash的低位。如果tab的位点为空，则放入一个Node。否则p不等于null。

如果位点不是null，总会检查p，即头结点是否与插入的key相同，判断条件时hash相等且key相等。如果头结点不同会检查后续链表节点和树的子节点。注意此时并没有完成覆盖，而是将老节点暂存在e中。这就是代码A，接着看代码C，对于HashMap的源码分析，我们总是先读链表，后读红黑树。【分支二】

需要注意的是，key不相同但hash相同的多个key，不会覆盖，而是保留。所以如果hashCode方法设计不够随机，更容易产生hash冲突。只有key相同，hash相同时才会覆盖。判断key相同的方法是引用的同一个对象或符合equals方法。

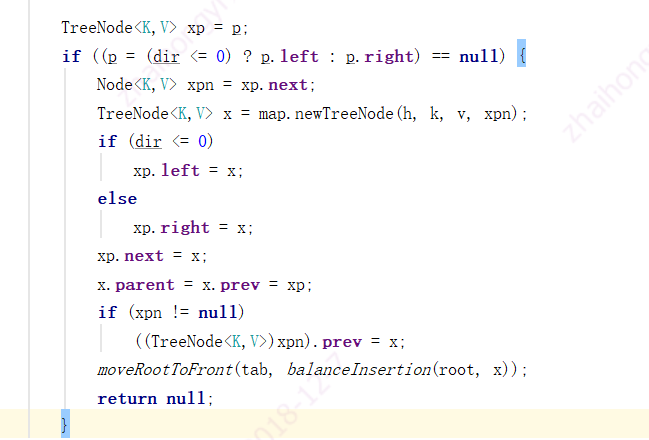
同样继续遍历链表，如果已经到达尾部，则将元素插入链尾。遍历过程中也会对比key是否相同，判断条件跟A相同。如果有相同的说明需要覆盖，同样暂存在e中。

当e不是null，即哈希表中已经存在key，则替换value，返回老值。接着递增modCount，并判断是否需要扩容。个人认为代码A没必要，可以合并到遍历链表中。接着回到【分支二】。



在一颗树种插入的本质是找到一个节点为新节点的父节点，和一个方向，插入为左孩子还是右孩子。红黑树是一颗二叉搜索平衡树，查找时利用了二叉搜索树折半查找的特性。虽然这些Node hash冲突（位点相同），但hash值并不一定相同，首先可以根据hash值的大小判断位置，使用了二叉搜索树的特性（如果hash>Node.hash则向右子树寻找【代码A】，如果hash<Node.hash则向左子树寻找【代码B】），如果hash相同则会判断key是否相同，如果两者都相同即已经存在这个key【代码C】。如果不相同，则会尝试调用compareTo方法，比较两者大小。前提是Key的类型实现了Compareable接口。如果没实现Compareable接口，或compareTo返回了0，即无法比较两个对象的大小【代码D】，会继续在以当前节点为根节点的子树上搜索是否包含重复key，即ch.find(h, k, ck)，即遍历子树对比每个元素。但HashMap只会搜索一次（原因不明），如果搜索不到或已经搜索过了，会用更快速的方法判断方向，该方法就是tieBreakOrder(Object a, Object b)，它使用System.identityHashCode(obj)，会根据内存位置计算一个int，但并不简单返回内存位置。使用identityHashCode目的是如果对象重写了hashCode方法，identityHashCode可以避免调用重写的hashCode方法，直接调用Object类的hashCode方法。经过tieBreakOrder，一定会找出方向。【代码E】

可以得出一个结论，当hash相同时HashMap用compareTo比较两个对象，如果Key的类型能有效的比较出对象大小，会帮助HashMap查找，进而提高性能。否则HashMap不明确寻找方向会遍历子树节点。这也是读源码的额外收获。



找到插入的位置，即节点p，插入左孩子还是右孩子右dir决定。可以看到，当在树中查找时如果当前节点前进方向的孩子为null时，就会插入这个位置。由于红黑树是由链表转换而来，且TreeNode继承了Node，TreeNode不仅包含树的指针（parent、left、right、prev），还包含链表的指针（next）,prev的确是TreeNode的属性【分支三】。所以创建树节点时，除了对本身的指针赋值意外，还要维护链式next、prev指针，跟向双向链表中插入元素一样。插入后通过左旋、右旋平衡树【分支四】，平衡后树的根节点可能会改变，再重置为新的根节点。【分支二结束】

put操作的分析就结束了，留下了三个分支：

分支一：如何扩容

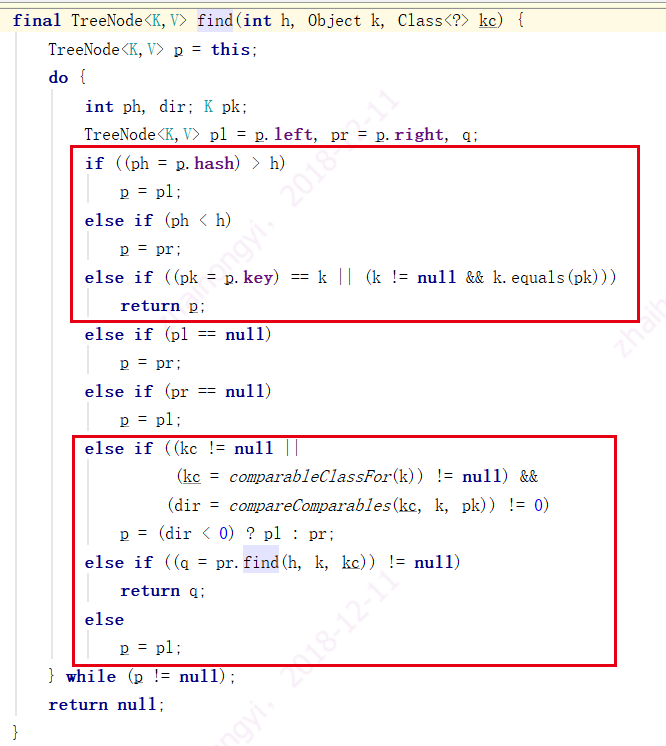
分支三：TreeNode中next、prev指针的使用

分支四：红黑树如何实现平衡

三、Get()

Put()方法找到插入节点，Get()方法查找特定节点。首先计算hash，检查头结点，如果是链表则一次向后遍历查找，如果是树节点会利用二叉搜索树的特性和compareTo方法选择方向，找到节点。

主要分析一下树节点的查找，第一个红框，根据hash的大小决定方向，利用二叉搜索树的特性。第二个红框，当hash相同时使用且Key类型实现了Comparable接口，则调用compareTo方法。这时也可以理解dir使用-1,1也是为了适配compareTo的返回值。如果hash相同但没实现Comparable或compareTo返回了0，即无法比较两个对象，HashMap会先递归的向右查找，如果查找到则返回，否则向左查找。这里也能说明Key对象实现compareTo方法的重要性，能减少树查找的深度，提高查找效率。

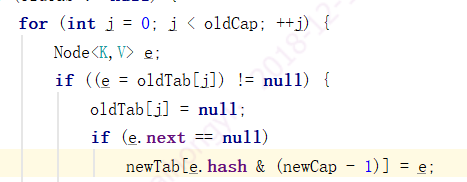


1. resize()

回到分支一，如何扩容。当第一次使用或表中键值对数量超过threshold，都会触发扩容。原则上新表容量是原表容量的两倍，成倍扩容的优势是rehash后的点要么在原地，要么在原位 + oldCap位置上，无论对单节点、链表和树节点都有这个结论，这时理解HashMap扩容非常重要的特性。但如果新表长度已经超过最大值2 ^ 30，会扩容并设置threshold为Integer.MAX\_VALUE；原表大小超过最大值时，则不会扩容。

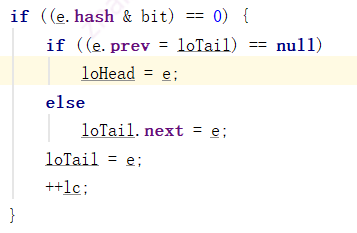
总的来说，扩容时会创建一个新表，然后将原表的数据依次rehash放入新表。

遍历旧表，如果节点只是单一节点（next == null），则直接rehash一下，放入新表。rehash使用了e.hash & (newCap - 1)，而不是取模。这也利用了2的幂次的优势。2的幂次 - 1的二进制高位全0，低位全1，与操作跟取模结果相同，且与操作比取模更快。

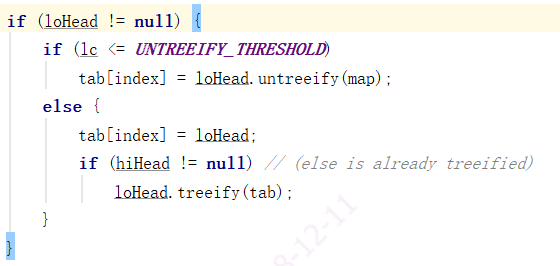


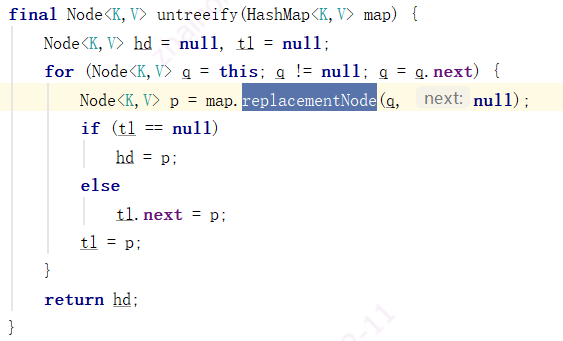
接着如果是链表，会被分成两个链表，一个留在原地，一个放在原位 + oldCap位置上。HashMap分别创建两个链表的头结点和尾节点，尾节点用于插入，最终将头结点放入hash表中。最好能平均分成两个链表，减少遍历的长度，HashMap用了一个巧妙的方法，只判断hash值的某一位是0还是1，对于hash值可以认为是随机int，每一位都是随机的，同样也使用了与操作(e.hash & oldCap) == 0。oldCap是2的幂次，所以只有一位是1，完美解决问题。这也是2的幂次的优势。新链表的创建、放入新表的操作略。

如果是树节点，HashMap会拆分为两个链表（虽然声明是TreeNode，但只利用了链表特性），原理跟链表较相似，但还有不同。前面说过TreeNode包含了next节点，因为本身树就是由链表转化而来，且保留了next指针，所以可以通过next遍历一个树。rehash放在那颗树根链表的判断方法相同。遍历过程中记录两个链表的元素数量。



如果拆分的链表的元素数量小于等于6个时，注意并不是8个，应该转化为链表，将TreeNode转化为Node，去掉多余的指针，再创建新的链式结构。否则应该重新创建一棵树。注释中already treeified的意思是如果另一颗树为空，当前这颗树就是原树，无需树化。而树化的过程就是遍历链表，创建树。跟Put()方法很相似。代码省略。





分支一结束。

1. remove()

5.1 remove(key)

删除某个key对应的键值对，删除操作第一步是查找，跟get方法相同。接着删除这个节点。node是要删除的节点，先判断是否匹配value，或value是否相等，如果是单一元素，或删除链表头部会进入node==p，如果删除链表其他节点则执行p.next = node.next，p节点是node的前驱节点。如果是TreeNode，执行removeTreeNode()。

