## **HashMap 线程不安全的典型表现**

## **我们先回顾一下HashMap。HashMap是一个数组链表，当一个key/Value对被加入时，首先会通过Hash算法定位出这个键值对要被放入的桶，然后就把它插到相应桶中。如果这个桶中已经有元素了，那么发生了碰撞，这样会在这个桶中形成一个链表。一般来说，当有数据要插入HashMap时，都会检查容量有没有超过设定的thredhold，如果超过，需要增大HashMap的尺寸，但是这样一来，就需要对整个HashMap里的节点进行重哈希操作。关于HashMap的重哈希操作本文不再详述，读者可以参考《Map 综述（一）：彻头彻尾理解 HashMap》一文。在此，笔者借助陈皓的《疫苗：JAVA HASHMAP的死循环》一文说明HashMap线程不安全的典型表现 —— 死循环。**

HashMap重哈希的关键源码如下：

/\*\*

\* Transfers all entries from current table to newTable.

\*/

void transfer(Entry[] newTable) {

// 将原数组 table 赋给数组 src

Entry[] src = table;

int newCapacity = newTable.length;

// 将数组 src 中的每条链重新添加到 newTable 中

for (int j = 0; j < src.length; j++) {

Entry<K,V> e = src[j];

if (e != null) {

src[j] = null; // src 回收

// 将每条链的每个元素依次添加到 newTable 中相应的桶中

do {

Entry<K,V> next = e.next;

// e.hash指的是 hash(key.hashCode())的返回值;

// 计算在newTable中的位置，注意原来在同一条子链上的元素可能被分配到不同的桶中

int i = indexFor(e.hash, newCapacity);

e.next = newTable[i];

newTable[i] = e;

e = next;

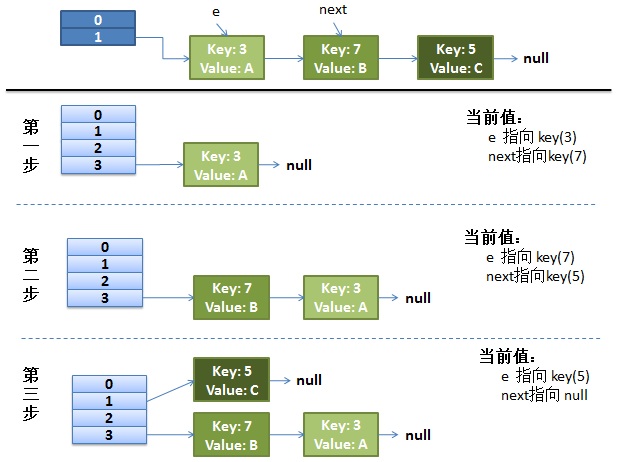
} while (e != null);

}

}

}

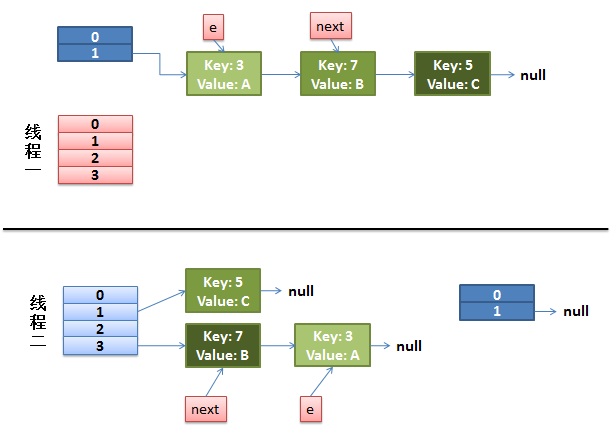
****1、单线程环境下的重哈希过程演示****



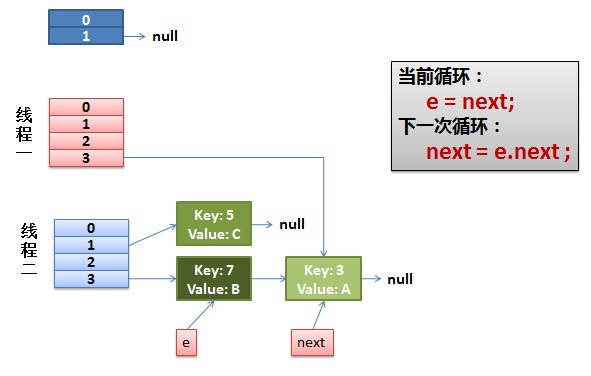
单线程情况下，rehash 不会出现任何问题，如上图所示。假设hash算法就是最简单的 key mod table.length（也就是桶的个数）。最上面的是old hash表，其中的Hash表桶的个数为2， 所以对于 key = 3、7、5 的键值对在 mod 2以后都冲突在table[1]这里了。接下来的三个步骤是，Hash表resize成4，然后对所有的键值对重哈希的过程。

2、多线程环境下的重哈希过程演示

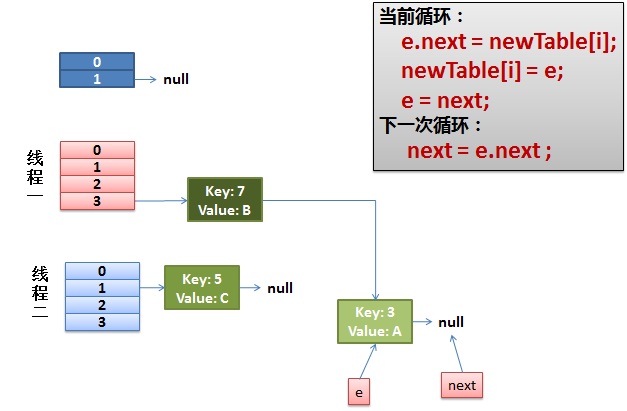
　　假设我们有两个线程，我用红色和浅蓝色标注了一下，被这两个线程共享的资源正是要被重哈希的原来1号桶中的Entry链。我们再回头看一下我们的transfer代码中的这个细节：



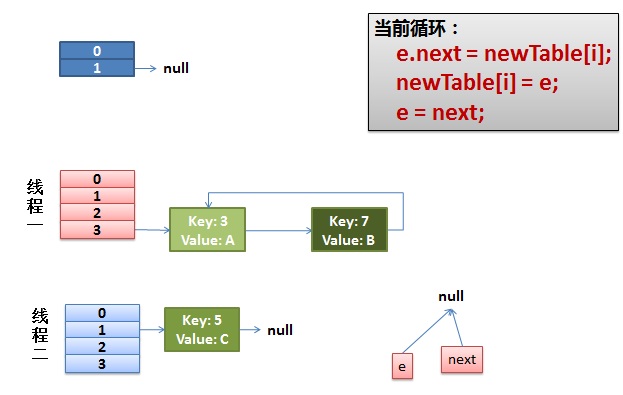
注意，在Thread2重哈希后，Thread1的指针e和指针next分别指向了Thread2重组后的链表(e指向了key(3)，而next指向了key(7))。此时，Thread1被调度回来执行：Thread1先是执行 newTalbe[i] = e;然后是e = next，导致了e指向了key(7)，而下一次循环的next = e.next导致了next指向了key(3)，如下图所示：



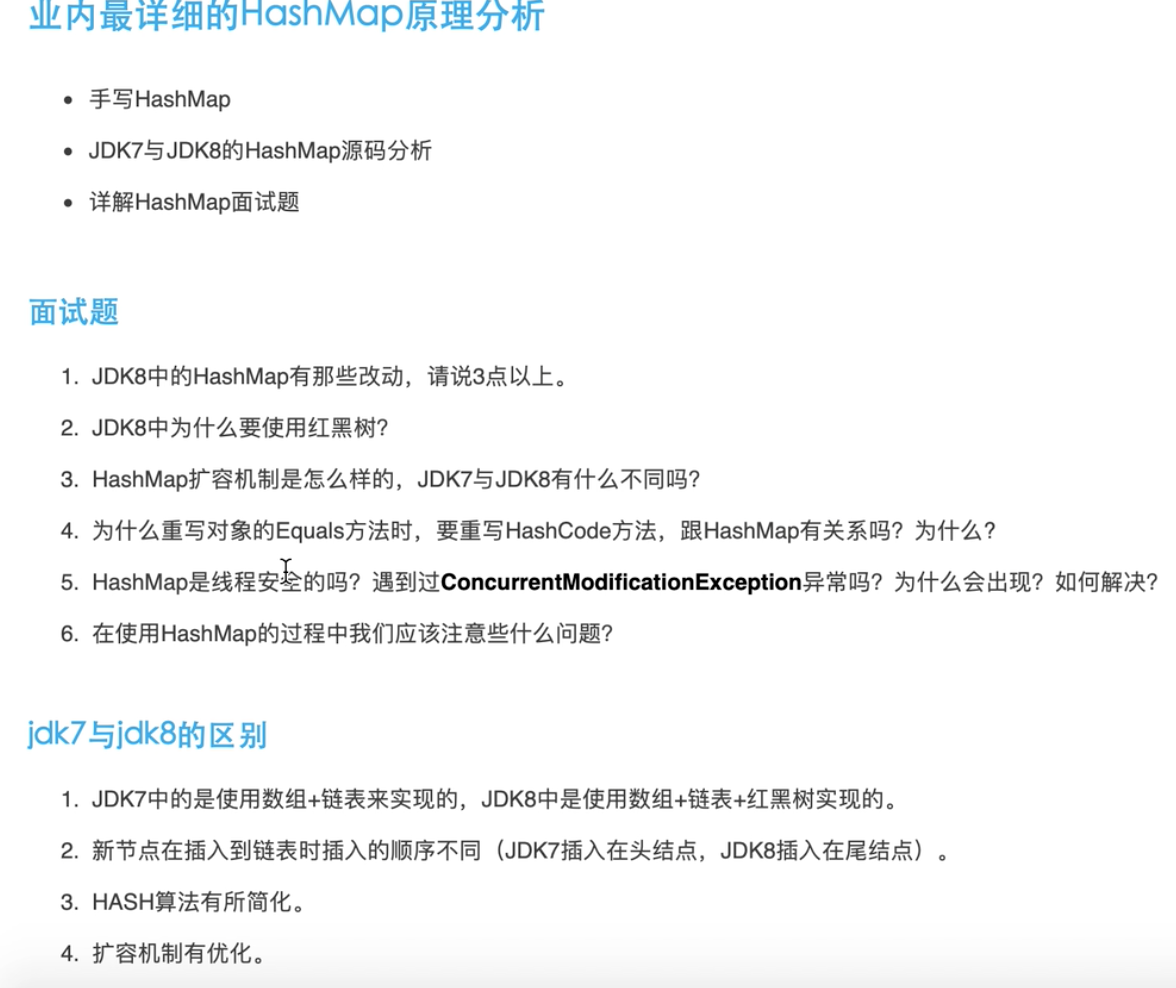
这时，一切安好。Thread1有条不紊的工作着：把key(7)摘下来，放到newTable[i]的第一个，然后把e和next往下移，如下图所示：



在此时，特别需要注意的是，当执行e.next = newTable[i]后，会导致 key(3).next 指向了 key(7)，而此时的key(7).next 已经指向了key(3)，环形链表就这样出现了，如下图所示。于是，当我们的Thread1调用HashMap.get(11)时，悲剧就出现了 —— Infinite Loop。



这是HashMap在并发环境下使用中最为典型的一个问题，就是在HashMap进行扩容重哈希时导致Entry链形成环。一旦Entry链中有环，势必会导致在同一个桶中进行插入、查询、删除等操作时陷入死循环。

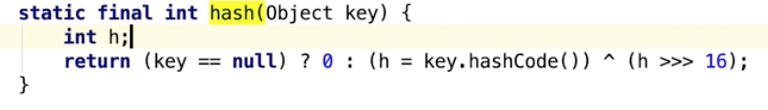


红黑树的查找是是在 完全平衡二叉树 和 链表 之间

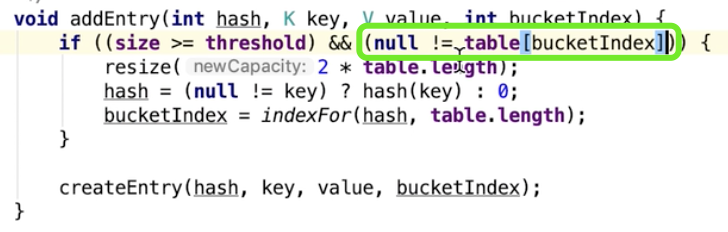
折中 思想

Jdk7和jdk8

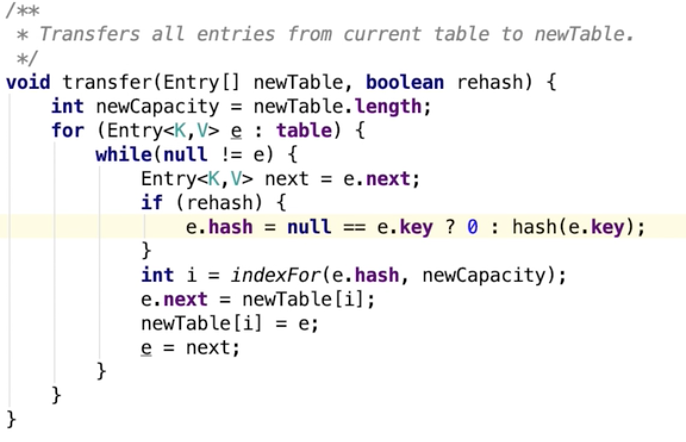
jdk8 简化了，因为用了红黑树



当前插入的数组位置不为空，才扩容

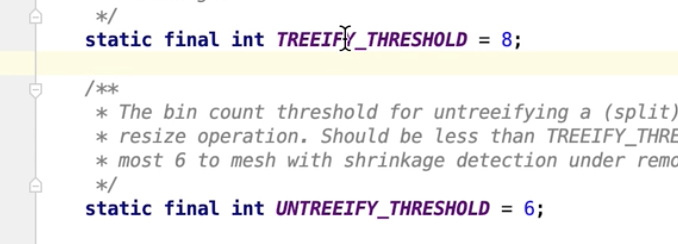


有死锁的风险--jdk8已解决

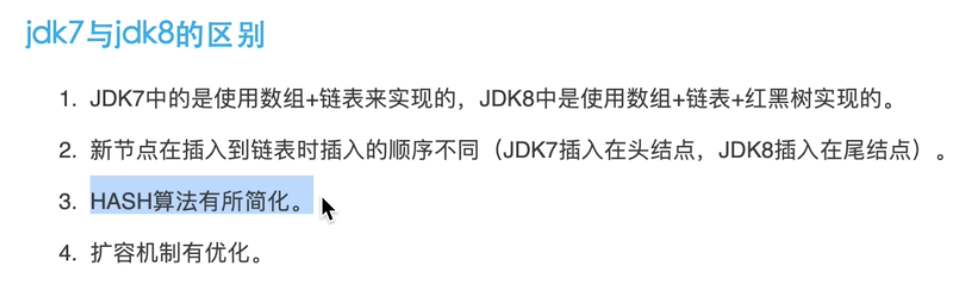


jdk8当链表的元素个数大于等于8转为红黑树

当红黑树的元素个数小于等于6时，转为链表



插入的节点，插在尾结点，因为需要遍历链表，得到结点数目，判断是否要转为红黑树，到了最后，于是就插在了尾结点



扩容机制，避免的死锁的风险

FastFail快速失败，设计的时候就没有按照线程安全来设置

