HashMap为什么是线程不安全

题目标签

学习时长：30分钟

题目难度：一般

知识点标签：并发，线程，数据结构，jdk1.8

解题思路分析

- 先从源码的实现进行分析
- 线程不安全是针对多线程而言，所以需要先来叙述下单线程情况
- 紧跟其后进行一个线程的叙述

HashMap的底层存储结构

HashMap底层是一个Entry数组，一旦发生Hash冲突的的时候，HashMap采用拉链法解决碰撞冲突，Entry内部的变量：

```
final Object key;
Object value;
Entry next;
int hash
```

通过Entry内部的next变量可以知道使用的是链表，这时候我们可以知道，如果多个线程，在某一时刻同时操作HashMap并执行put操作，而有大于两个key的hash值相同，如图中a1、a2，这个时候需要解决碰撞冲突，而解决冲突的办法上面已经说过，对于链表的结构在这里不再赘述，暂且不讨论是从链表头部插入还是从尾部初入，这个时候两个线程如果恰好都取到了对应位置的头结点e1，而最终的结果可想而知，a1、a2两个数据中势必会有一个会丢失。

HashMap 中put方法

```
public Object put(Object obj, Object obj1)
    {
        if(table == EMPTY_TABLE)
            inflateTable(threshold);
        if(obj == null)
            return putForNullKey(obj1);
        int i = hash(obj);
        int j = indexFor(i, table.length);
        for(Entry entry = table[j]; entry != null; entry = entry.next)
        {
            Object obj2;
            if(entry.hash == i && ((obj2 = entry.key) == obj ||
obj.equals(obj2)))
            {
                Object obj3 = entry.value;
```

```
                entry.value = obj1;
                entry.recordAccess(this);
                return obj3;
            }
        }

        modCount++;
        addEntry(i, obj, obj1, j);
        return null;
    }
```

put方法不是同步的，同时调用了addEntry方法：

```
void addEntry(int i, Object obj, Object obj1, int j)
    {
        if(size >= threshold && null != table[j])
        {
            resize(2 * table.length);
            i = null == obj ? 0 : hash(obj);
            j = indexFor(i, table.length);
        }
        createEntry(i, obj, obj1, j);
    }
```

addEntry方法依然不是同步的，所以导致了线程不安全出现伤处问题，其他类似操作不再说明，源码一看便知，下面主要说一下另一个非常重要的知识点，同样也是HashMap非线程安全的原因，我们知道在HashMap存在扩容的情况，对应的方法为HashMap中的resize方法：

HashMap中rezise()方法源码

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
if (oldCap >= MAXIMUM_CAPACITY) {
threshold = Integer.MAX_VALUE;
            return oldTab;
        }
else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
   else {
// 第一次插入值的时候，table=null,所以给数组进行初始化，设置容量，阈值
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
if (newThr == 0) {
float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
```

```
    }
threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                        oldTab[j] = null;
            if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
             else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        //如果异或运算=0，则位置不变
                 if ((e.hash & oldCap) == 0) {
if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
else {
if (hiTail == null)
                                hiHead = e;
                     else
                                hiTail.next = e;
                                hiTail = e;
                    }
                      } while ((e = next) != null);
                      if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                      }
if (hiTail != null) {
                        hiTail.next = null;
              //如果这个数据是在table[2]上，并且此时数组的容量是16.
              //那么新的位置将变成new_index=2+16
                 newTab[j + oldCap] = hiHead;
               }
                 }
            }
        }
    }
return newTab;
}
```

HashMap初始容量大小为16,一般来说，当有数据要插入时，都会检查容量有没有超过设定的thredhold=负载因子*16，如果超过，需要增大Hash表的尺寸，但是这样一来，整个Hash表 里的元素都需要被重算一遍。这叫rehash，这个成本相当的大。

对索引数组中的元素for循环遍历：

对链表上的每一个节点遍历：用 next 取得要转移那个元素的下一个，将 e 转移到新 Hash 表的头部，使用头插法插入节点。

循环2，直到链表节点全部转移

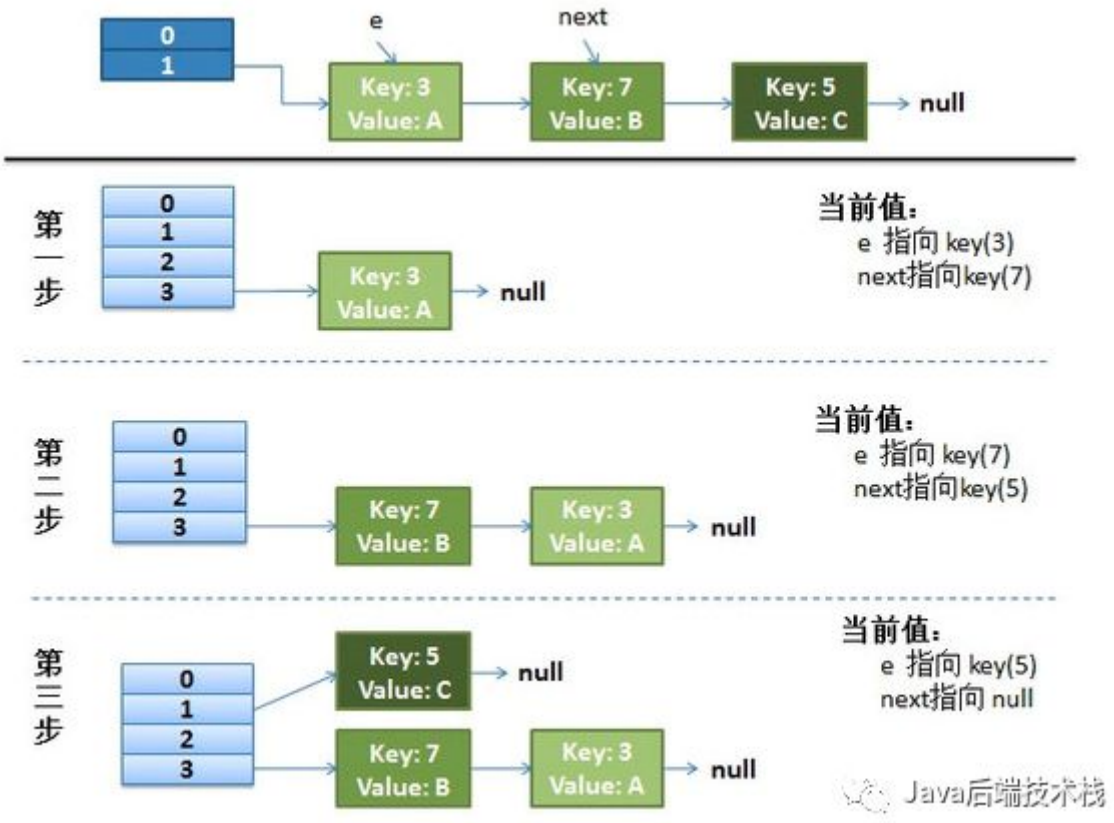
循环1，直到所有索引数组全部转移

经过这几步，我们会发现转移的时候是逆序的。假如转移前链表顺序是1->2->3，那么转移后就会变成3->2->1。

这时候就有点头绪了，死锁问题不就是因为1->2的同时2->1造成的吗？所以，HashMap 的死锁问题就出在这个for循环遍历。

单线程的rezise

- 假设hash算法就是最简单的 key mod table.length（也就是数组的长度）。
- 最上面的是old hash 表，其中的Hash表的 size = 2, 所以 key = 3, 7, 5，在 mod 2以后碰撞发生在table[1]
- 接下来的三个步骤是 Hash表 resize 到4，并将所有的 <key,value> 重新rehash到新 Hash 表的过程

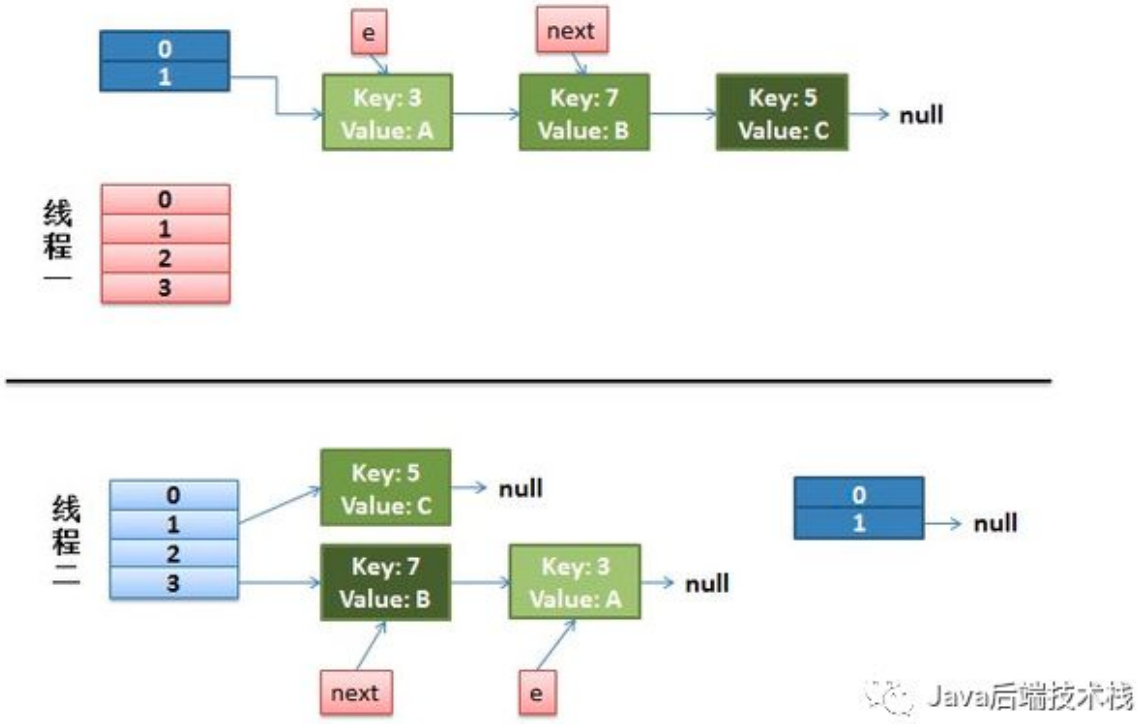


多线程rezise

- 因为是单向链表，如果要转移头指针，一定要保存下一个结点，不然转移后链表就丢了
- e 要插入到链表的头部，所以要先用 e.next 指向新的 Hash 表第一个元素（为什么不加到新链表最后？因为复杂度是 O（N））

- 现在新 Hash 表的头指针仍然指向 e 没转移前的第一个元素，所以需要将新 Hash 表的头指针指向 e
- 转移 e 的下一个结点

现在的状态：



从上面的图我们可以看到，因为线程1的 e 指向了 key(3)，而 next 指向了 key(7)，在线程2 rehash 后，就指向了线程2 rehash 后的链表。

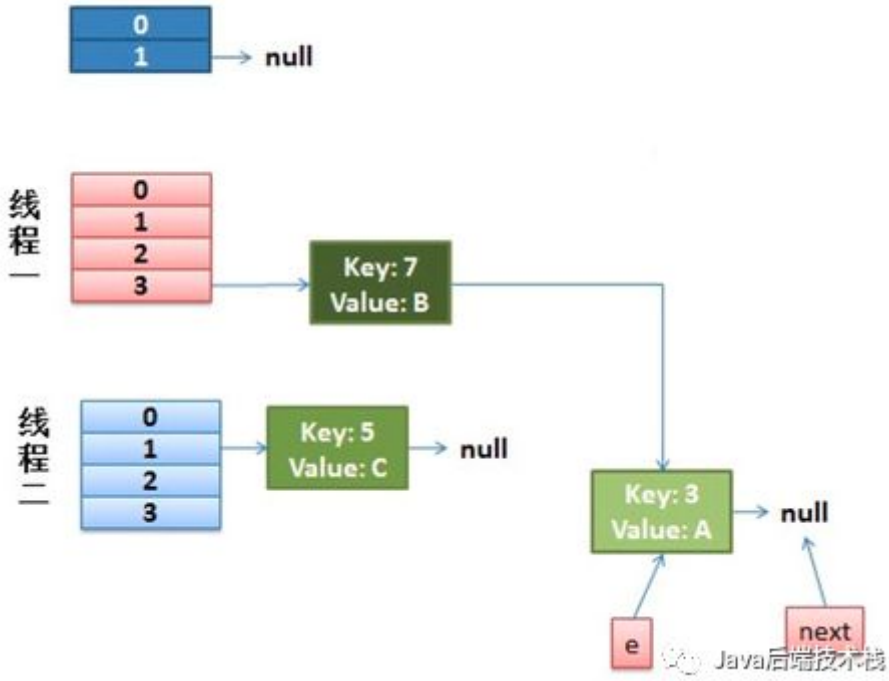
然后线程1被唤醒了：

- 执行e.next = newTable[i]，于是 key(3)的 next 指向了线程1的新 Hash 表，因为新 Hash 表为空，所以e.next = null，
- 执行newTable[i] = e，所以线程1的新 Hash 表第一个元素指向了线程2新 Hash 表的 key(3)。好了，e 处理完毕。
- 执行e = next，将 e 指向 next，所以新的 e 是 key(7)

然后该执行 key(3)的 next 节点 key(7)了：

- 现在的 e 节点是 key(7)，首先执行Entry<K,V> next = e.next,那么 next 就是 key(3)了
- 执行e.next = newTable[i]，于是key(7) 的 next 就成了 key(3)
- 执行newTable[i] = e，那么线程1的新 Hash 表第一个元素变成了 key(7)
- 执行e = next，将 e 指向 next，所以新的 e 是 key(3)

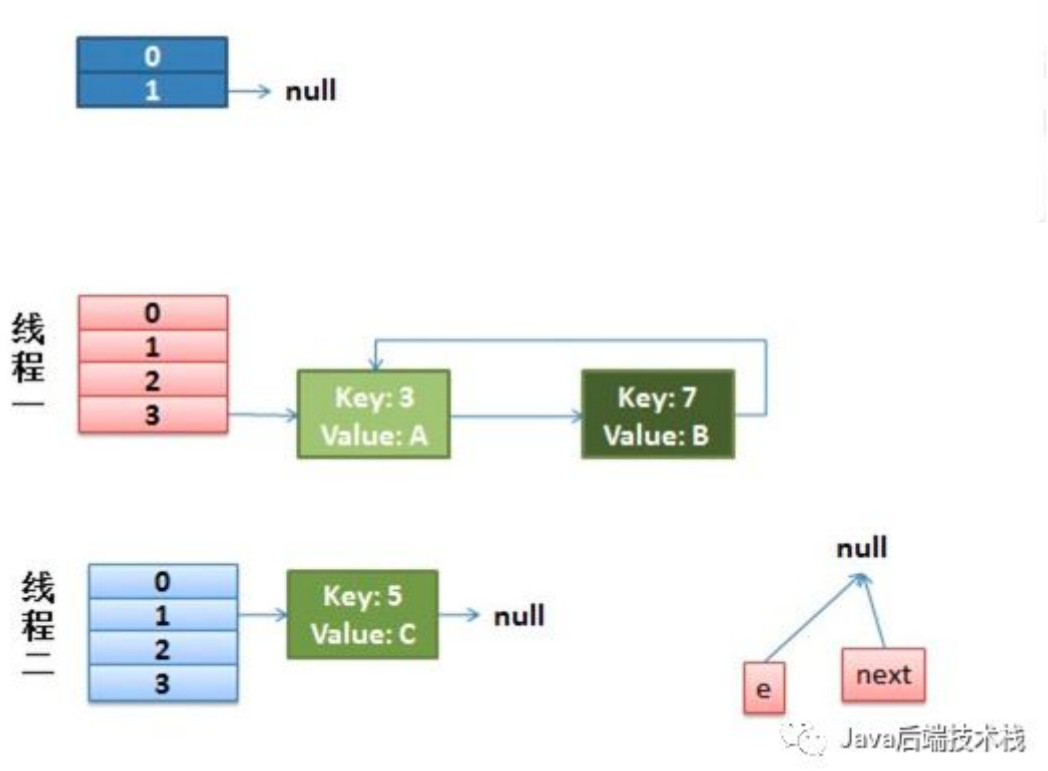
这时候的状态



然后又该执行 key(7)的 next 节点 key(3)了：

- 现在的 e 节点是 key(3)，首先执行Entry<K,V> next = e.next,那么 next 就是 null
- 执行e.next = newTable[i]，于是key(3) 的 next 就成了 key(7)
- 执行newTable[i] = e，那么线程1的新 Hash 表第一个元素变成了 key(3)
- 执行e = next，将 e 指向 next，所以新的 e 是 key(7)

这时候状态为



很明显，环形链表出现了！！当然，现在还没有事情，因为下一个节点是 null，

所以就遍历完成了，等put()的其余过程搞定后，HashMap 的底层实现就是线程1的新 Hash 表了。

如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

使用建议

对有多线程应用的功能，或需要考虑线程安全的功能，建议使用ConcurrentHashMap 类来代替 HashMap

总结

在并发执行put操作时会发生数据覆盖的情况。