[https://mp.weixin.qq.com/s?\_\_biz=MzIyNDU2ODA4OQ==&mid=2247484006&idx=1&sn=b9c6543948005bbab9315d52766f743b&chksm=e80db410df7a3d06c4168bba2976c5555fd1cd12ed96b1d1594d460eeea091bf1db8ef961cb4&scene=21#wechat\_redirect](https://mp.weixin.qq.com/s?__biz=MzIyNDU2ODA4OQ==&mid=2247484006&idx=1&sn=b9c6543948005bbab9315d52766f743b&chksm=e80db410df7a3d06c4168bba2976c5555fd1cd12ed96b1d1594d460eeea091bf1db8ef961cb4&scene=21" \l "wechat_redirect)

# 你知道为什么HashMap是线程不安全的吗？

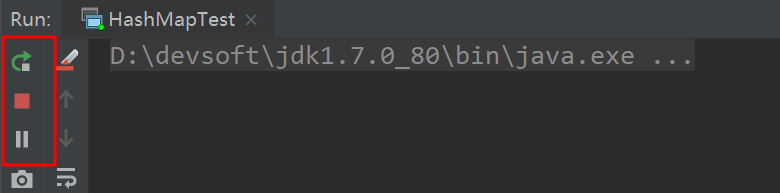
我们都知道HashMap是线程不安全的，在多线程环境中不建议使用，但是其线程不安全主要体现在什么地方呢，本文将对该问题进行解密。

1.jdk1.7中的HashMap

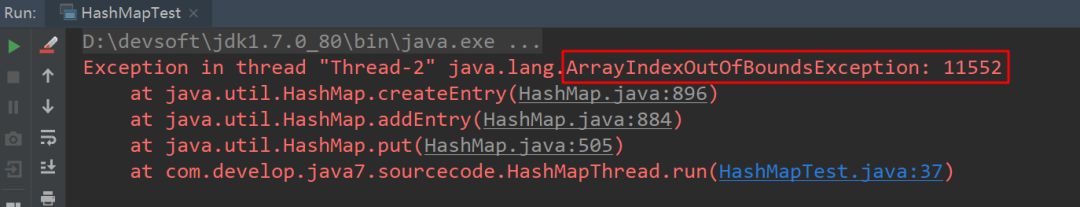
在jdk1.8中对HashMap做了很多优化，这里先分析在jdk1.7中的问题，相信大家都知道在jdk1.7多线程环境下HashMap容易出现死循环，这里我们先用代码来模拟出现死循环的情况：

**public** **class** **HashMapTest** {  
  
    **public** **static** **void** **main**(String[] args) {  
        HashMapThread thread0 = **new** HashMapThread();  
        HashMapThread thread1 = **new** HashMapThread();  
        HashMapThread thread2 = **new** HashMapThread();  
        HashMapThread thread3 = **new** HashMapThread();  
        HashMapThread thread4 = **new** HashMapThread();  
        thread0.start();  
        thread1.start();  
        thread2.start();  
        thread3.start();  
        thread4.start();  
    }  
}  
  
**class** **HashMapThread** **extends** **Thread** {  
    **private** **static** AtomicInteger ai = **new** AtomicInteger();  
    **private** **static** Map<Integer, Integer> map = **new** HashMap<>();  
  
    @Override  
    **public** **void** **run**() {  
        **while** (ai.**get**() < 1000000) {  
            map.put(ai.**get**(), ai.**get**());  
            ai.incrementAndGet();  
        }  
    }  
}

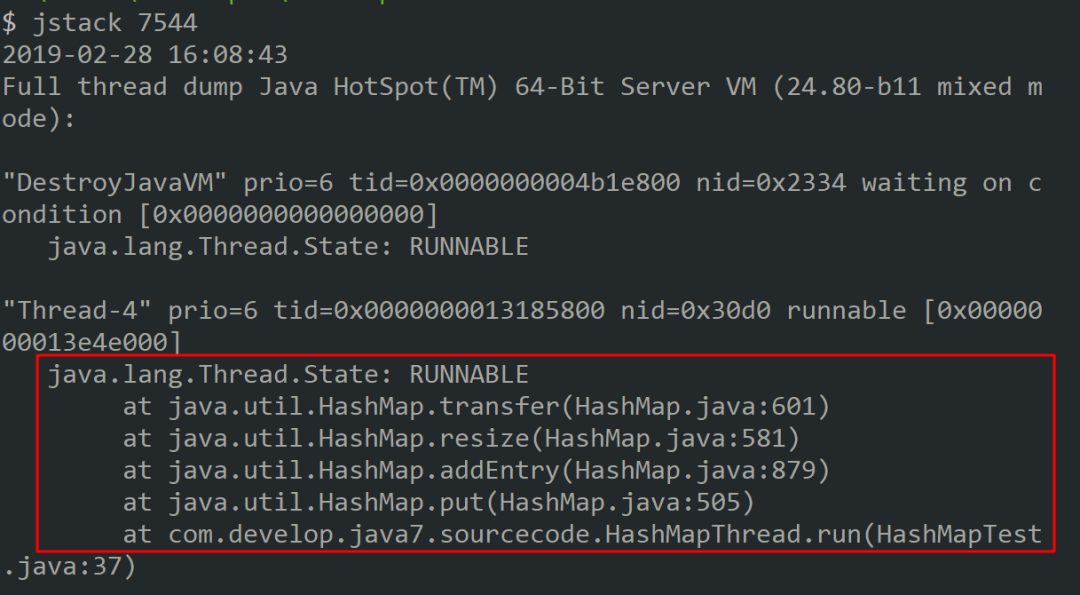
上述代码比较简单，就是开多个线程不断进行put操作，并且HashMap与AtomicInteger都是全局共享的。在多运行几次该代码后，出现如下死循环情形：



其中有几次还会出现数组越界的情况：



这里我们着重分析为什么会出现死循环的情况，通过jps和jstack命名查看死循环情况，结果如下：、



从堆栈信息中可以看到出现死循环的位置，通过该信息可明确知道死循环发生在HashMap的扩容函数中，根源在transfer函数中，jdk1.7中HashMap的transfer函数如下：

void transfer(Entry[] newTable, boolean rehash) {  
        int newCapacity = newTable.length;  
        **for** (Entry<K,V> e : table) {  
            **while**(null != e) {  
                Entry<K,V> **next** = e.**next**;  
                **if** (rehash) {  
                    e.hash = null == e.key ? 0 : hash(e.key);  
                }  
                int i = indexFor(e.hash, newCapacity);  
                e.**next** = newTable[i];  
                newTable[i] = e;  
                e = **next**;  
            }  
        }  
    }

总结下该函数的主要作用：

**在对table进行扩容到newTable后，需要将原来数据转移到newTable中，注意10-12行代码，这里可以看出在转移元素的过程中，使用的是头插法，也就是链表的顺序会翻转，这里也是形成死循环的关键点。**

下面进行详细分析。

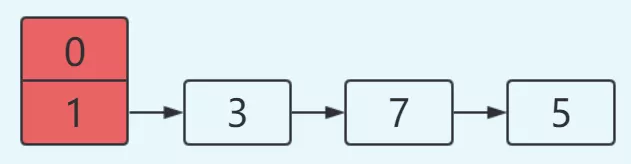
### 1.1 扩容造成死循环分析过程

前提条件：

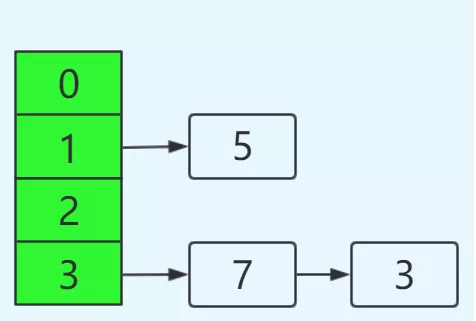
这里假设

1. hash算法为简单的用key mod链表的大小。
2. 最开始hash表size=2，key=3,7,5，则都在table[1]中。
3. 然后进行resize，使size变成4。

未resize前的数据结构如下：

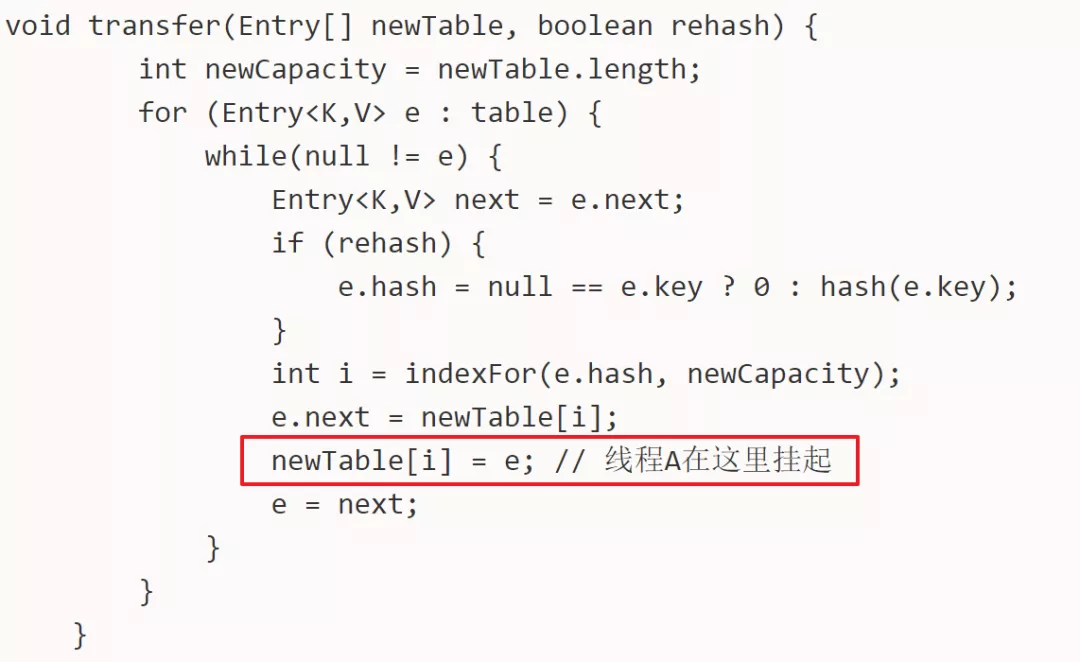


如果在单线程环境下，最后的结果如下：

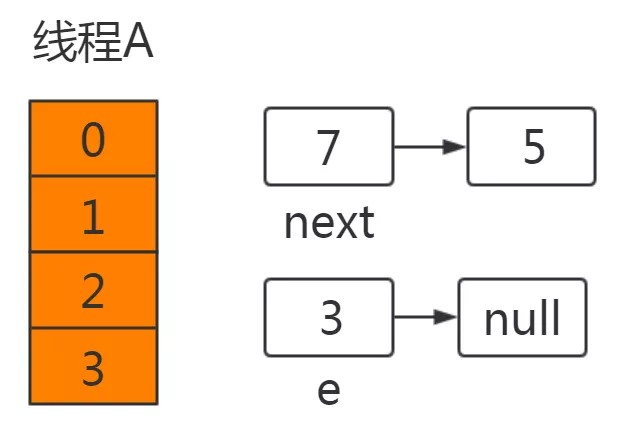


这里的转移过程，不再进行详述，只要理解transfer函数在做什么，其转移过程以及如何对链表进行反转应该不难。

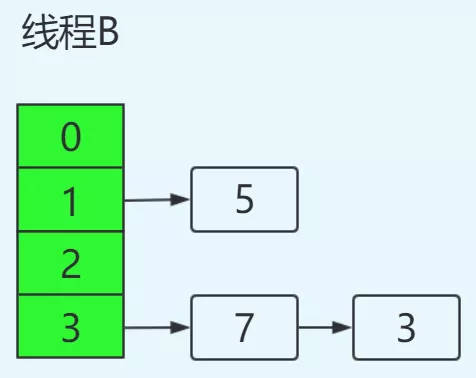
然后在多线程环境下，假设有两个线程A和B都在进行put操作。线程A在执行到transfer函数中第11行代码处挂起，因为该函数在这里分析的地位非常重要，因此再次贴出来。



此时线程A中运行结果如下：



线程A挂起后，此时线程B正常执行，并完成resize操作，结果如下：

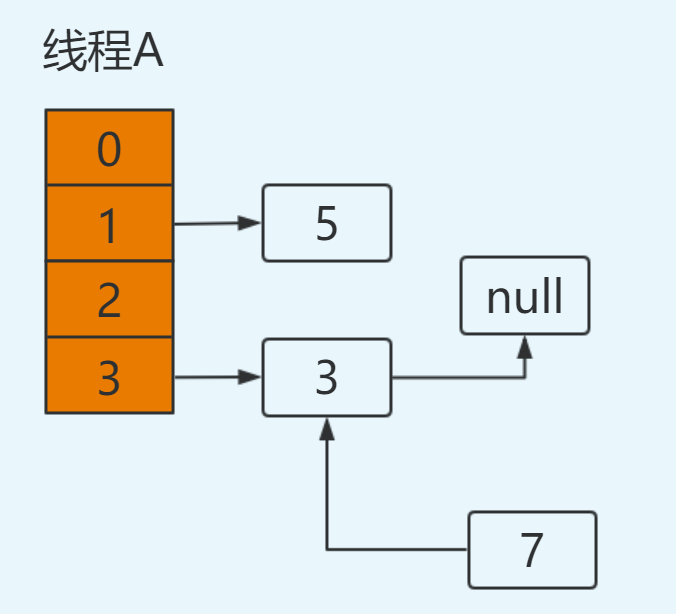


这里需要特别注意的点：**由于线程B已经执行完毕，根据Java内存模型，现在newTable和table中的Entry都是主存中最新值：7.next=3，3.next=null。**

此时切换到线程A上，在线程A挂起时内存中值如下：e=3，next=7，newTable[3]=null，代码执行过程如下：

newTable[3]=e ----> newTable[3]=3  
e=next ----> e=7

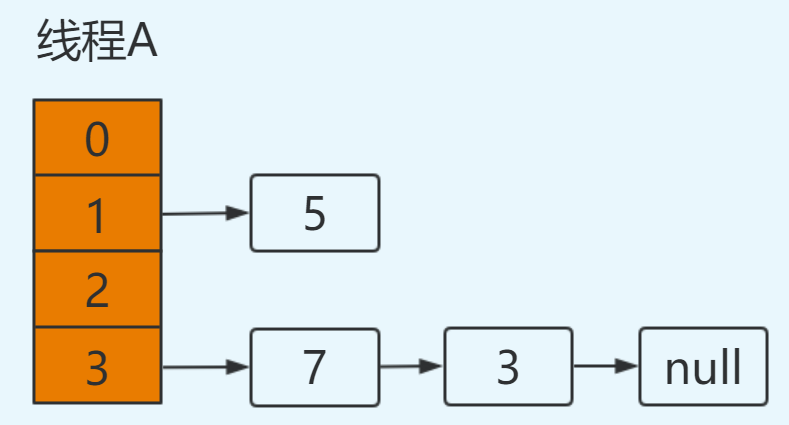
此时结果如下：



继续循环：

e=7  
next=e.next *----> next=3【从主存中取值】*  
e.next=newTable[3] *----> e.next=3【从主存中取值】*  
newTable[3]=e *----> newTable[3]=7*  
e=next *----> e=3*

结果如下：

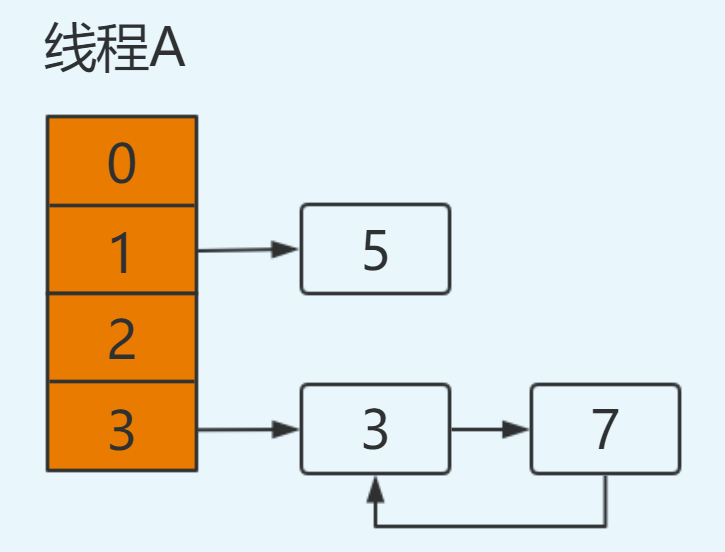


再次进行循环：

e=3  
**next**=e.**next** ----> **next**=null  
e.**next**=newTable[3] ----> e.**next**=7 即：3.**next**=7  
newTable[3]=e ----> newTable[3]=3  
e=**next** ----> e=null

注意此次循环：e.next=7，而在上次循环中7.next=3，出现环形链表，并且此时e=null循环结束。

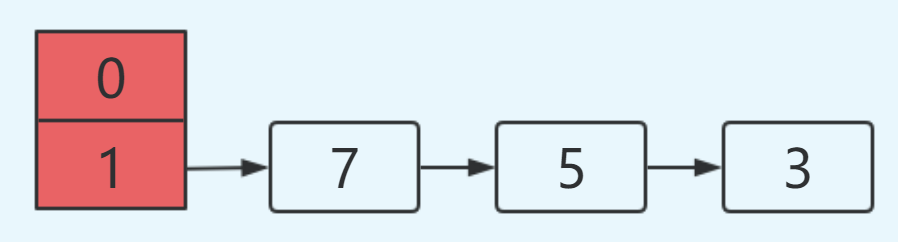
结果如下：



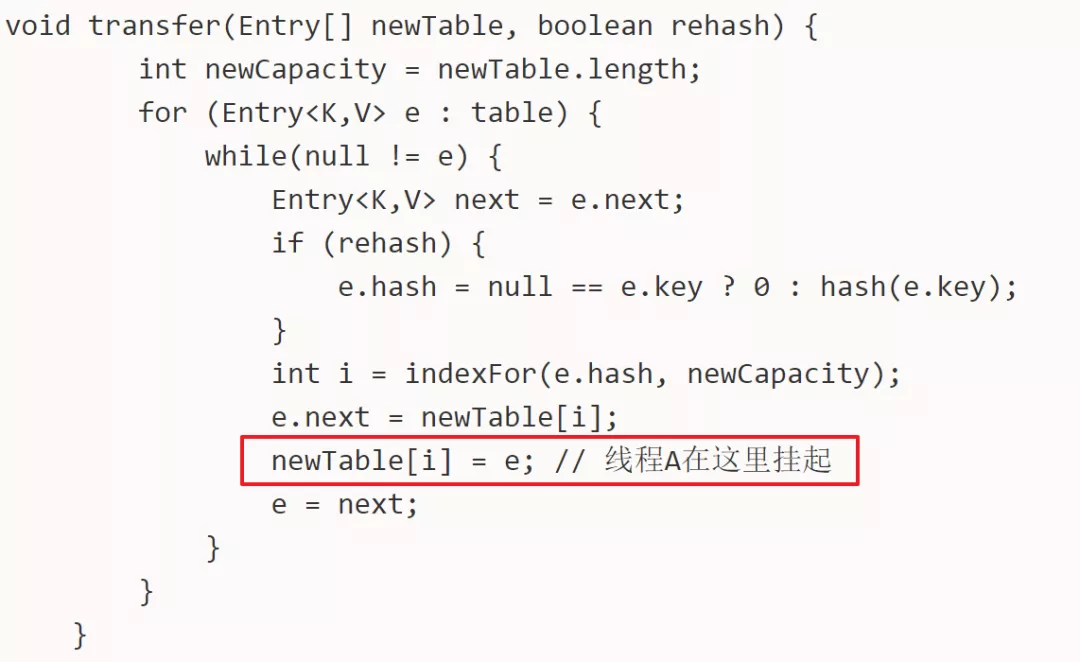
在后续操作中只要涉及轮询hashmap的数据结构，就会在这里发生死循环，造成悲剧。

### 1.2 扩容造成数据丢失分析过程

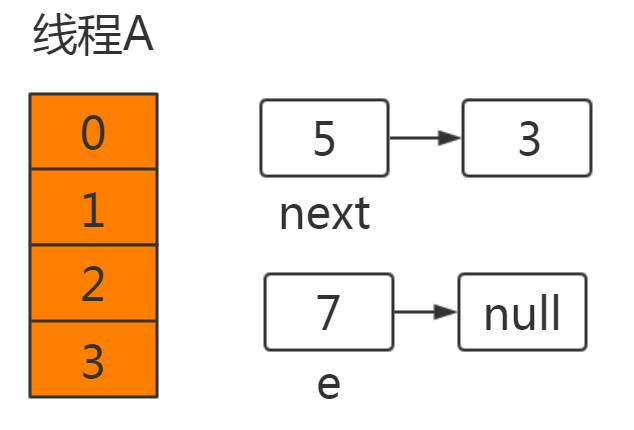
遵照上述分析过程，初始时：



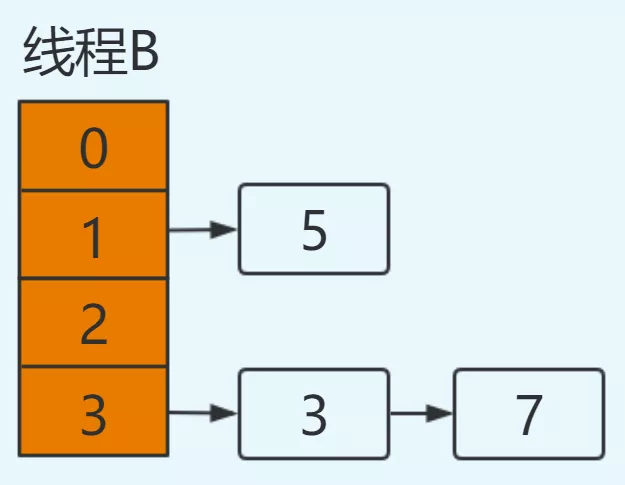
线程A和线程B进行put操作，同样线程A挂起：



此时线程A的运行结果如下：



此时线程B已获得CPU时间片，并完成resize操作：



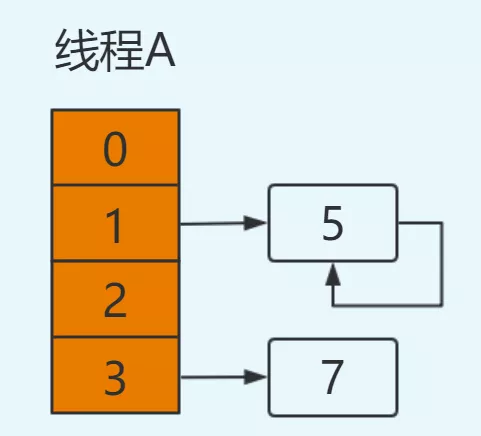
同样注意由于线程B执行完成，newTable和table都为最新值：5.next=null。

此时切换到线程A，在线程A挂起时：e=7，next=5，newTable[3]=null。

执行newtable[i]=e，就将7放在了table[3]的位置，此时next=5。接着进行下一次循环：

e=5  
next=e.next *----> next=null，从主存中取值*  
e.next=newTable[1] *----> e.next=5，从主存中取值*  
newTable[1]=e *----> newTable[1]=5*  
e=next *----> e=null*

将5放置在table[1]位置，此时e=null循环结束，3元素丢失，并形成环形链表。并在后续操作hashmap时造成死循环。



## 2.jdk1.8中HashMap

在jdk1.8中对HashMap进行了优化，在发生hash碰撞，不再采用头插法方式，而是直接插入链表尾部，因此不会出现环形链表的情况，但是在多线程的情况下仍然不安全，这里我们看jdk1.8中HashMap的put操作源码：



这是jdk1.8中HashMap中put操作的主函数， 注意第6行代码，如果没有hash碰撞则会直接插入元素。如果线程A和线程B同时进行put操作，刚好这两条不同的数据hash值一样，并且该位置数据为null，所以这线程A、B都会进入第6行代码中。

假设一种情况，线程A进入后还未进行数据插入时挂起，而线程B正常执行，从而正常插入数据，然后线程A获取CPU时间片，此时线程A不用再进行hash判断了，问题出现：线程A会把线程B插入的数据给覆盖，发生线程不安全。

## 总结

首先HashMap是线程不安全的，其主要体现：

1. 在jdk1.7中，在多线程环境下，扩容时会造成环形链或数据丢失。
2. 在jdk1.8中，在多线程环境下，会发生数据覆盖的情况。