http://coolshell.cn/articles/9606.html

在淘宝内网里看到同事发了贴说了一个CPU被100%的线上故障，并且这个事发生了很多次，原因是在Java语言在并发情况下使用HashMap造成Race Condition，从而导致死循环。这个事情我4、5年前也经历过，本来觉得没什么好写的，因为Java的HashMap是非线程安全的，所以在并发下必然出现问题。但是，我发现近几年，很多人都经历过这个事（在网上查“HashMap Infinite Loop”可以看到很多人都在说这个事）所以，觉得这个是个普遍问题，需要写篇疫苗文章说一下这个事，并且给大家看看一个完美的“Race Condition”是怎么形成的。

问题的症状

从前我们的Java代码因为一些原因使用了HashMap这个东西，但是当时的程序是单线程的，一切都没有问题。后来，我们的程序性能有问题，所以需要变成多线程的，于是，变成多线程后到了线上，发现程序经常占了100%的CPU，查看堆栈，你会发现程序都Hang在了HashMap.get()这个方法上了，重启程序后问题消失。但是过段时间又会来。而且，这个问题在测试环境里可能很难重现。

我们简单的看一下我们自己的代码，我们就知道HashMap被多个线程操作。而Java的文档说HashMap是非线程安全的，应该用ConcurrentHashMap。

但是在这里我们可以来研究一下原因。

Hash表数据结构

我需要简单地说一下HashMap这个经典的数据结构。

HashMap通常会用一个指针数组（假设为table[]）来做分散所有的key，当一个key被加入时，会通过Hash算法通过key算出这个数组的下标i，然后就把这个<key, value>插到table[i]中，如果有两个不同的key被算在了同一个i，那么就叫冲突，又叫碰撞，这样会在

table[i]上形成一个链表。

我们知道，如果table[]的尺寸很小，比如只有2个，如果要放进10个keys的话，那么碰撞非常频繁，于是一个O(1)的查找算法，就变成了链表遍历，性能变成了O(n)，这是Hash表的缺陷（可参看《Hash Collision DoS 问题》）。

所以，Hash表的尺寸和容量非常的重要。一般来说，Hash表这个容器当有数据要插入时，都会检查容量有没有超过设定的 thredhold，如果超过，需要增大Hash表的尺寸，但是这样一来，整个Hash表里的无素都需要被重算一遍。这叫rehash，这个成本相当的大。

相信大家对这个基础知识已经很熟悉了。

HashMap的rehash源代码

下面，我们来看一下Java的HashMap的源代码。

Put一个Key,Value对到Hash表中：

```
public V put(K key, V value)
{
    ......
    //算Hash值
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    //如果该key已被插入，则替换掉旧的value （链接操作）
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key ||
key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    //该key不存在，需要增加一个结点
```

```
    addEntry(hash, key, value, i);
    return null;
}
```

检查容量是否超标

```
void addEntry(int hash, K key, V value, int bucketIndex)
{
    Entry<K,V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    //查看当前的size是否超过了我们设定的阈值threshold，如果超过，需要resize
    if (size++ >= threshold)
        resize(2 * table.length);
}
```

新建一个更大尺寸的hash表，然后把数据从老的Hash表中迁移到新的Hash表中。

```
void resize(int newCapacity)
{
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    ......
    //创建一个新的Hash Table
    Entry[] newTable = new Entry[newCapacity];
    //将Old Hash Table上的数据迁移到New Hash Table上
    transfer(newTable);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);
}
```

迁移的源代码，注意高亮处：

```
void transfer(Entry[] newTable)
{
    Entry[] src = table;
    int newCapacity = newTable.length;
    //下面这段代码的意思是：
    //  从OldTable里摘一个元素出来，然后放到NewTable中
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
```

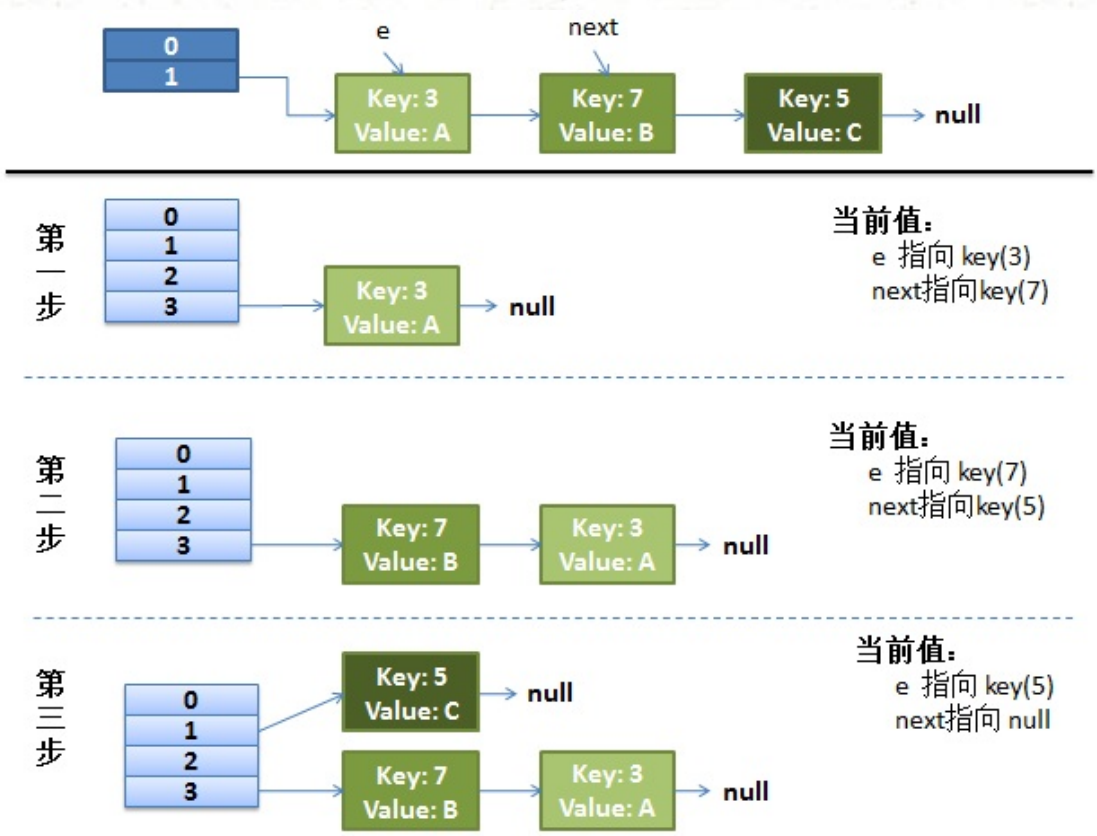
```
                src[j] = null;
                do {
                    Entry<K,V> next = e.next;
                    int i = indexFor(e.hash, newCapacity);
                    e.next = newTable[i];
                    newTable[i] = e;
                    e = next;
                } while (e != null);
            }
        }
    }
```

好了，这个代码算是比较正常的。而且没有什么问题。

正常的ReHash的过程

画了个图做了个演示。

- 我假设了我们的hash算法就是简单的用key mod 一下表的大小（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，然后所有的 <key,value> 重新rehash的过程



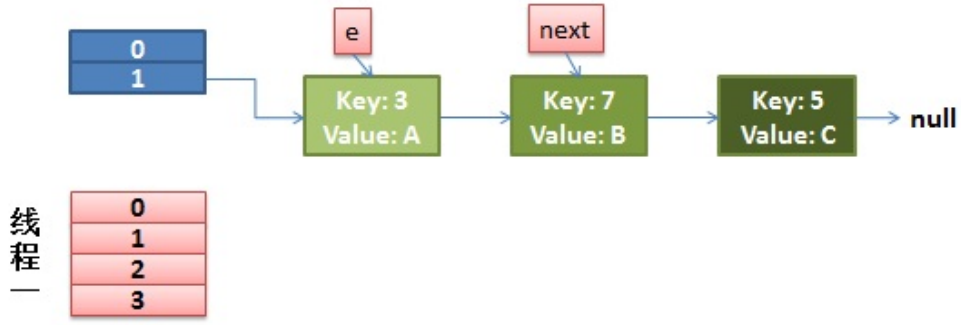
并发下的Rehash

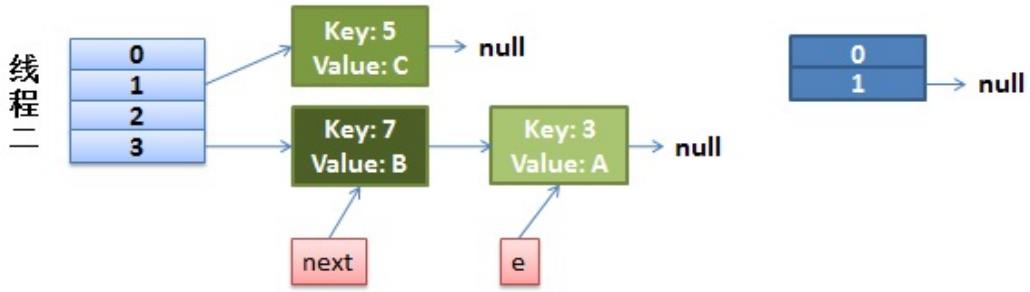
1）假设我们有两个线程。我用红色和浅蓝色标注了一下。

我们再回头看一下我们的 transfer代码中的这个细节：

```
do {
    Entry<K,V> next = e.next; // <--假设线程一执行到这里就被调度挂起了
    int i = indexFor(e.hash, newCapacity);
    e.next = newTable[i];
    newTable[i] = e;
    e = next;
} while (e != null);
```

而我们的线程二执行完成了。于是我们有下面的这个样子。

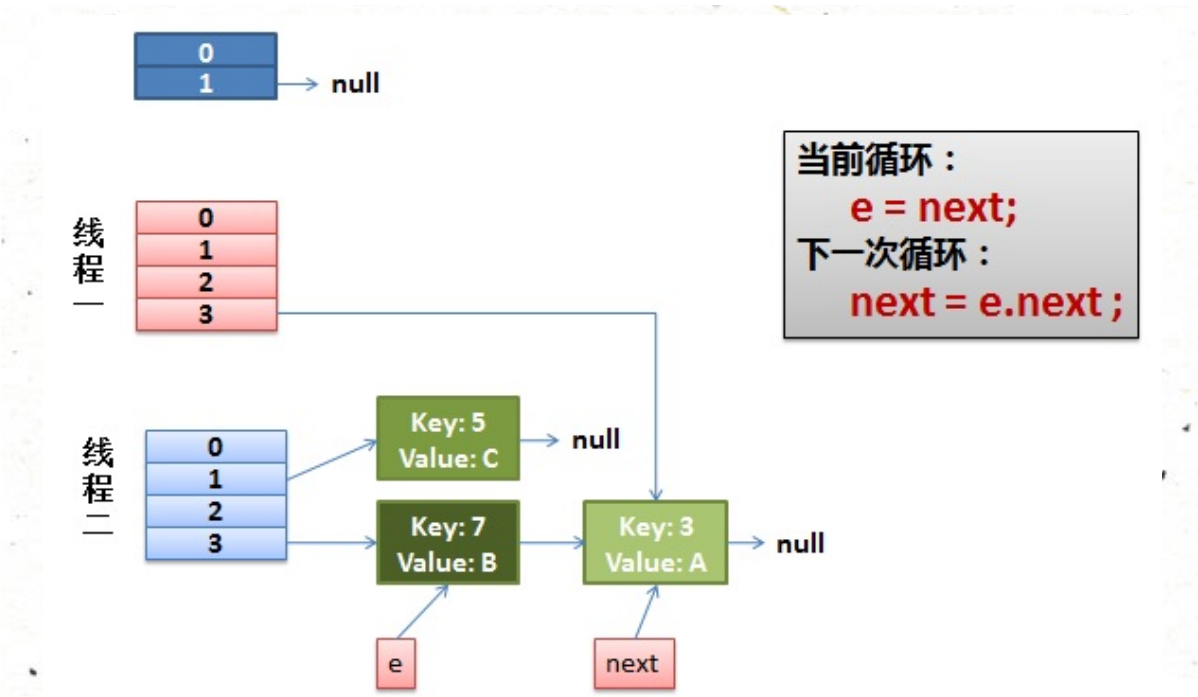




注意，因为**Thread1**的 **e** 指向了**key(3)**，而**next**指向了**key(7)**，其在线程二**rehash**后，指向了线程二重组后的链表。我们可以看到链表的顺序被反转后。

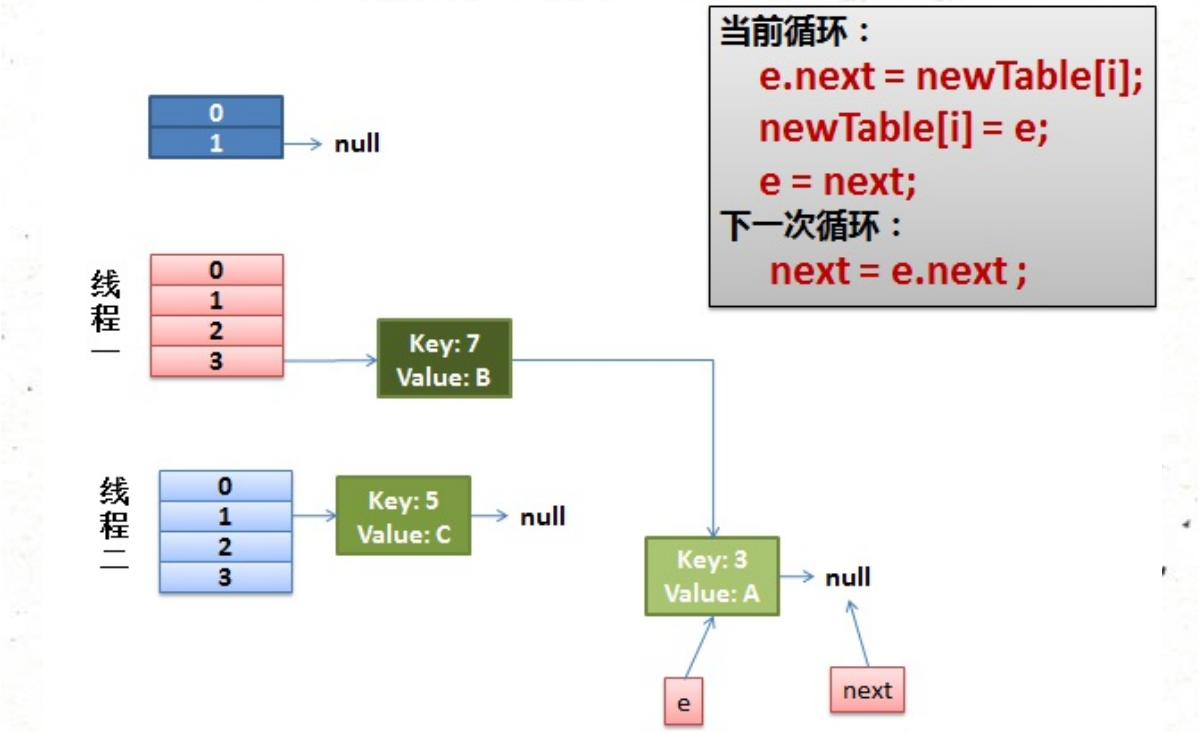
2）线程一被调度回来执行。

- 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)**



3）一切安好。

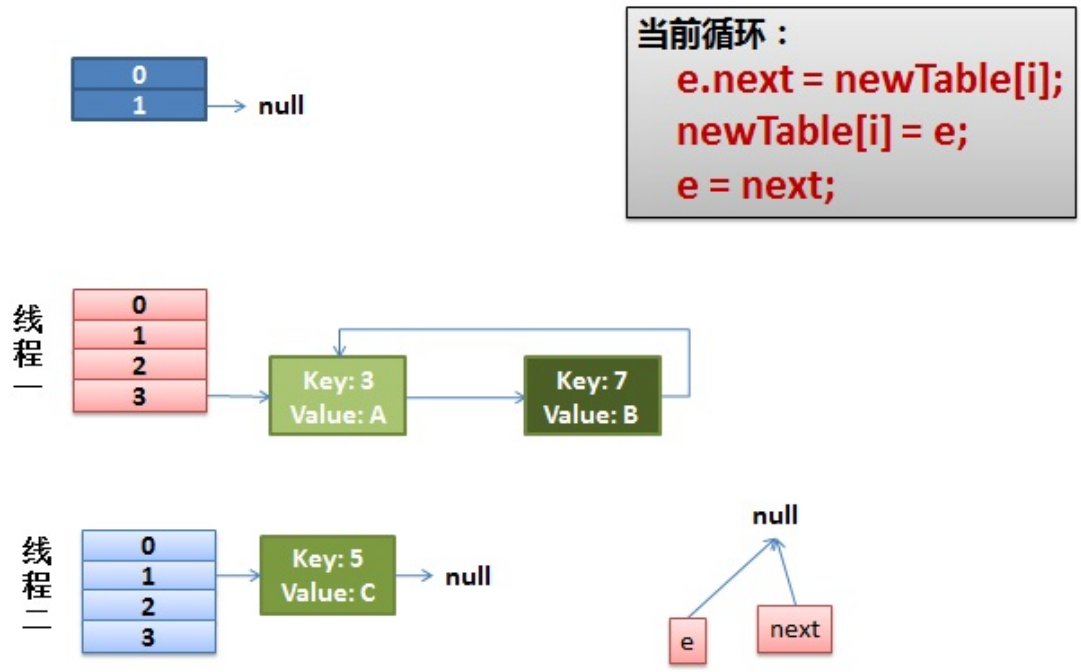
线程一接着工作。把key(7)摘下来，放到newTable[i]的第一个，然后把e和next往下移。



4）环形链接出现。

e.next = newTable[i] 导致 key(3).next 指向了 key(7)

注意：此时的key(7).next 已经指向了key(3)， 环形链表就这样出现了。



于是，当我们的线程一调用到，HashTable.get(11)时，悲剧就出现了——Infinite Loop。