http://www.importnew.com/22011.html

同时可以参考：http://blog.csdn.net/zjkc050818/article/details/78462354

HashMap的原理以及如何实现，之前在[JDK7与JDK8中HashMap的实现](http://my.oschina.net/hosee/blog/618953)中已经说明了。

那么，为什么说HashMap是线程不安全的呢？它在多线程环境下，会发生什么情况呢？

1. resize死循环

我们都知道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，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。

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | void resize(int newCapacity) {          Entry[] oldTable = table;          int oldCapacity = oldTable.length;          if (oldCapacity == MAXIMUM\_CAPACITY) {              threshold = Integer.MAX\_VALUE;              return;          }            Entry[] newTable = new Entry[newCapacity];          transfer(newTable, initHashSeedAsNeeded(newCapacity));          table = newTable;          threshold = (int)Math.min(newCapacity \* loadFactor, MAXIMUM\_CAPACITY + 1);  } | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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;              }          }  } |

大概看下transfer：

1. 对索引数组中的元素遍历
2. 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。
3. 循环2，直到链表节点全部转移
4. 循环1，直到所有索引数组全部转移

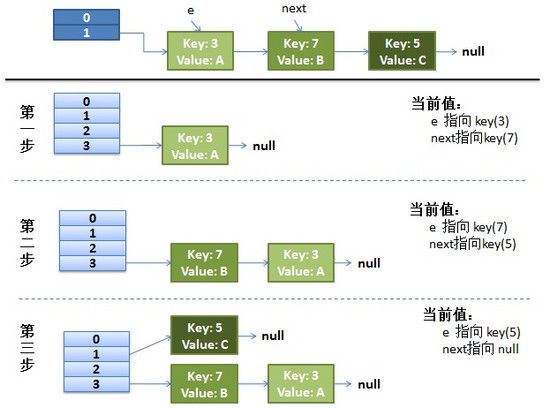
经过这几步，我们会发现转移的时候是逆序的。假如转移前链表顺序是1->2->3，那么转移后就会变成3->2->1。这时候就有点头绪了，死锁问题不就是因为1->2的同时2->1造成的吗？所以，HashMap 的死锁问题就出在这个transfer()函数上。

1. 1.1 单线程 rehash 详细演示

单线程情况下，rehash 不会出现任何问题：

* 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

如图所示：

[](http://www.importnew.com/22011.html/1-29)

1. 1.2 多线程 rehash 详细演示

为了思路更清晰，我们只将关键代码展示出来

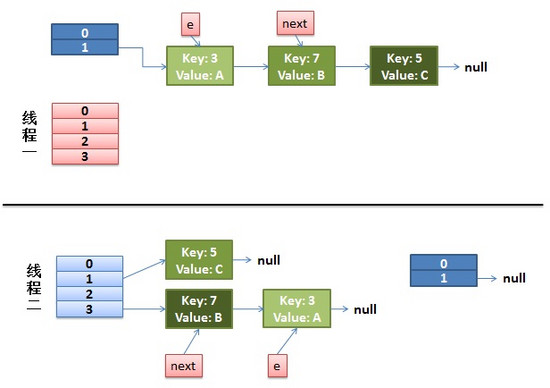
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | while(null != e) {      Entry<K,V> next = e.next;      e.next = newTable[i];      newTable[i] = e;      e = next;  } |

1. Entry<K,V> next = e.next;——因为是单链表，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
2. e.next = newTable[i];——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））
3. newTable[i] = e;——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
4. e = next——转移 e 的下一个结点

假设这里有两个线程同时执行了put()操作，并进入了transfer()环节

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | while(null != e) {      Entry<K,V> next = e.next; //线程1执行到这里被调度挂起了      e.next = newTable[i];      newTable[i] = e;      e = next;  } |

那么现在的状态为：

[](http://www.importnew.com/22011.html/2-24)

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。

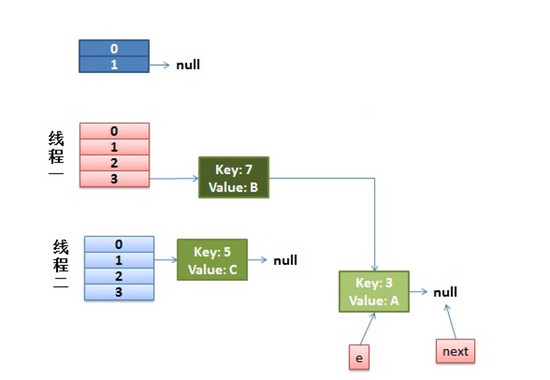
然后线程1被唤醒了：

1. 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，
2. 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。
3. 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:

1. 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
2. 执行e.next = newTable[i]，于是key(7) 的 next 就成了 key(3)
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)
4. 执行e = next，将 e 指向 next，所以新的 e 是 key(3)

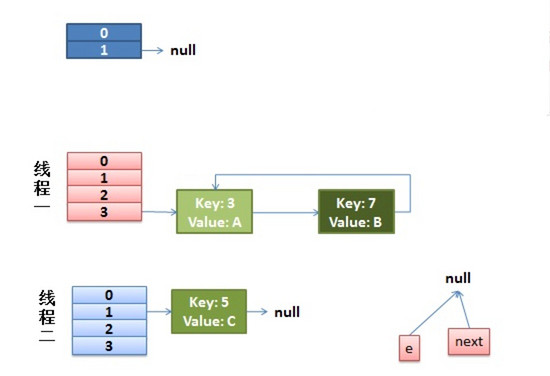
这时候的状态图为：

[](http://www.importnew.com/22011.html/3-19)

然后又该执行 key(7)的 next 节点 key(3)了：

1. 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
2. 执行e.next = newTable[i]，于是key(3) 的 next 就成了 key(7)
3. 执行newTable[i] = e，那么线程1的新 Hash 表第一个元素变成了 key(3)
4. 执行e = next，将 e 指向 next，所以新的 e 是 key(7)

这时候的状态如图所示：

[](http://www.importnew.com/22011.html/4-18)

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transfer()就完成了，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。

2. fail-fast

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。

这个异常意在提醒开发者及早意识到线程安全问题，具体原因请查看[ConcurrentModificationException的原因以及解决措施](http://my.oschina.net/hosee/blog/612718)

顺便再记录一个HashMap的问题：

**为什么String, Interger这样的wrapper类适合作为键？** String, Interger这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。如果你可以仅仅通过将某个field声明成final就能保证hashCode是不变的，那么请这么做吧。因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能。