**import** java.util.concurrent.CountDownLatch;

掌握并发编程吧

# CountDownLatch

**CountDownLatch为一个同步辅助工具，让一个或多个线程等待，直到其他的线程执行操作完成。**

　　它的功能可以在绝大部分情况上替代join()方法，甚至在实际运用中比join()方法的用法更灵活。

**操作过程：用CountDownLatch类创建实例，指定需要等待完成点个数。await()方法会阻塞当前线程，直到计数器减为零。每次线程执行调用countDown()方法，就会使计数器减1，直到计数器减为0时，等待的线程继续运行。**CountDownLatch类中的构造器：

# ConcurrentHashMap

**实现原理**

**锁分离 (Lock Stripping)**

ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术。它使用了多个锁来控制对hash表的不同部分进行的修改。ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的hash table，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。

有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。不变性是多线程编程占有很重要的地位，下面还要谈到。

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. /\*\*
2. \* The segments, each of which is a specialized hash table
3. \*/
4. **final Segment<K,V>[] segments;**

**不变(Immutable)和易变(Volatile)**

ConcurrentHashMap完全允许多个读操作并发进行，读操作并不需要加锁。如果使用传统的技术，如HashMap中的实现，如果允许可以在hash链的中间添加或删除元素，读操作不加锁将得到不一致的数据。ConcurrentHashMap实现技术是保证HashEntry几乎是不可变的。HashEntry代表每个hash链中的一个节点，其结构如下所示：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **static final class HashEntry<K,V> {**
2. **final** K key;
3. **final** **int** hash;
4. **volatile** V value;
5. **final** HashEntry<K,V> next;
6. }

可以看到除了value不是final的，其它值都是final的，这意味着不能从hash链的中间或尾部添加或删除节点，因为这需要修改next引用值，所有的节点的修改只能从头部开始。对于put操作，可以一律添加到Hash链的头部。但是对于remove操作，可能需要从中间删除一个节点，这就需要将要删除节点的前面所有节点整个复制一遍，最后一个节点指向要删除结点的下一个结点。这在讲解删除操作时还会详述。为了确保读操作能够看到最新的值，将value设置成volatile，这避免了加锁。

**其它**

为了加快定位段以及段中hash槽的速度，每个段hash槽的的个数都是2^n，这使得通过位运算就可以定位段和段中hash槽的位置。当并发级别为默认值16时，也就是段的个数，hash值的高4位决定分配在哪个段中。但是我们也不要忘记《算法导论》给我们的教训：hash槽的的个数不应该是2^n，这可能导致hash槽分配不均，这需要对hash值重新再hash一次。（这段似乎有点多余了

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/ede461a2878547bda04753006e9d5210/icon_sad.gif

 ）

这是重新hash的算法，还比较复杂，我也懒得去理解了。

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **private static int hash(int h) {**
2. // Spread bits to regularize both segment and index locations,
3. // using variant of single-word Wang/Jenkins hash.
4. h += (h <<  15) ^ 0xffffcd7d;
5. h ^= (h >>> 10);
6. h += (h <<   3);
7. h ^= (h >>>  6);
8. h += (h <<   2) + (h << 14);
9. **return** h ^ (h >>> 16);
10. }

这是定位段的方法：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **final Segment<K,V> segmentFor(int hash) {**
2. **return** segments[(hash >>> segmentShift) & segmentMask];
3. }

**数据结构**

关于Hash表的基础数据结构，这里不想做过多的探讨。Hash表的一个很重要方面就是如何解决hash冲突，ConcurrentHashMap和HashMap使用相同的方式，都是将hash值相同的节点放在一个hash链中。与HashMap不同的是，ConcurrentHashMap使用多个子Hash表，也就是段(Segment)。下面是ConcurrentHashMap的数据成员：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>**
2. **implements** ConcurrentMap<K, V>, Serializable {
3. /\*\*
4. \* Mask value for indexing into segments. The upper bits of a
5. \* key's hash code are used to choose the segment.
6. \*/
7. **final** **int** segmentMask;
9. /\*\*
10. \* Shift value for indexing within segments.
11. \*/
12. **final** **int** segmentShift;
14. /\*\*
15. \* The segments, each of which is a specialized hash table
16. \*/
17. **final** Segment<K,V>[] segments;
18. }

ConcurrentMap的定义又如下:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **public** **interface** ConcurrentMap<K, V> **extends** Map<K, V> {        V putIfAbsent(K key, V value);    **boolean** remove(Object key, Object value);    **boolean** replace(K key, V oldValue, V newValue);        V replace(K key, V value);  } |

其中规定了4个方法。

* V putIfAbsent(K key, V value);   如果没有这个key，则放入这个key-value，返回null，否则返回key对应的value。
* boolean remove(Object key, Object value); 移除key和对应的value，如果key对应的不是value，移除失败
* boolean replace(K key, V oldValue, V newValue); 替代key对应的值，仅当当前值为旧值
* V replace(K key, V value); 替代key对应的值，只要当前有值

所有的成员都是final的，其中segmentMask和segmentShift主要是为了定位段，参见上面的segmentFor方法。

每个Segment相当于一个子Hash表，它的数据成员如下：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **static** **final** **class** Segment<K,V> **extends** ReentrantLock **implements** Serializable {
2. **private static final long serialVersionUID = 2249069246763182397L;**
3. /\*\*
4. \* The number of elements in this segment's region.
5. \*/
6. **transient** **volatile** **int** count;
8. /\*\*
9. \* Number of updates that alter the size of the table. This is
10. \* used during bulk-read methods to make sure they see a
11. \* consistent snapshot: If modCounts change during a traversal
12. \* of segments computing size or checking containsValue, then
13. \* we might have an inconsistent view of state so (usually)
14. \* must retry.
15. \*/
16. **transient** **int** modCount;
18. /\*\*
19. \* The table is rehashed when its size exceeds this threshold.
20. \* (The value of this field is always <tt>(int)(capacity \*
21. \* loadFactor)</tt>.)
22. \*/
23. **transient** **int** threshold;
25. /\*\*
26. \* The per-segment table.
27. \*/
28. **transient** **volatile** HashEntry<K,V>[] table;
30. /\*\*
31. \* The load factor for the hash table.  Even though this value
32. \* is same for all segments, it is replicated to avoid needing
33. \* links to outer object.
34. \* @serial
35. \*/
36. **final** **float** loadFactor;
37. }

count用来统计该段数据的个数，它是volatile，它用来协调修改和读取操作，以保证读取操作能够读取到几乎最新的修改。协调方式是这样的，每次修改操作做了结构上的改变，如增加/删除节点(修改节点的值不算结构上的改变)，都要写count值，每次读取操作开始都要读取count的值。这利用了Java 5中对volatile语义的增强，对同一个volatile变量的写和读存在happens-before关系。modCount统计段结构改变的次数，主要是为了检测对多个段进行遍历过程中某个段是否发生改变，在讲述跨段操作时会还会详述。threashold用来表示需要进行rehash的界限值。table数组存储段中节点，每个数组元素是个hash链，用HashEntry表示。table也是volatile，这使得能够读取到最新的table值而不需要同步。loadFactor表示负载因子。

**实现细节**

**修改操作**

先来看下删除操作remove(key)。

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **public V remove(Object key) {**
2. hash = hash(key.hashCode());
3. **return** segmentFor(hash).remove(key, hash, **null**);
4. }

整个操作是先定位到段，然后委托给段的remove操作。当多个删除操作并发进行时，只要它们所在的段不相同，它们就可以同时进行。下面是Segment的remove方法实现：

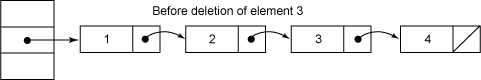
**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

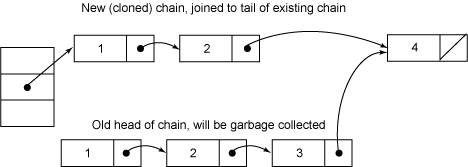
1. V remove(Object key, **int** hash, Object value) {
2. lock();
3. **try** {
4. **int** c = count - 1;
5. HashEntry<K,V>[] tab = table;
6. **int** index = hash & (tab.length - 1);
7. HashEntry<K,V> first = tab[index];
8. HashEntry<K,V> e = first;
9. **while** (e != **null** && (e.hash != hash || !key.equals(e.key)))
10. e = e.next;
12. V oldValue = **null**;
13. **if** (e != **null**) {
14. V v = e.value;
15. **if** (value == **null** || value.equals(v)) {
16. oldValue = v;
17. // All entries following removed node can stay
18. // in list, but all preceding ones need to be
19. // cloned.
20. ++modCount;
21. HashEntry<K,V> newFirst = e.next;
22. **for** (HashEntry<K,V> p = first; p != e; p = p.next)
23. newFirst = **new** HashEntry<K,V>(p.key, p.hash,
24. newFirst, p.value);
25. tab[index] = newFirst;
26. count = c; // write-volatile
27. }
28. }
29. **return** oldValue;
30. } **finally** {
31. unlock();
32. }
33. }

 整个操作是在持有段锁的情况下执行的，空白行之前的行主要是定位到要删除的节点e。接下来，如果不存在这个节点就直接返回null，否则就要将e前面的结点复制一遍，尾结点指向e的下一个结点。e后面的结点不需要复制，它们可以重用。下面是个示意图，我直接从[这个网站](http://www.ibm.com/developerworks/java/library/j-jtp08223/) 上复制的（画这样的图实在是太麻烦了，如果哪位有好的画图工具，可以推荐一下）。

删除元素之前：



删除元素3之后：



第二个图其实有点问题，复制的结点中应该是值为2的结点在前面，值为1的结点在后面，也就是刚好和原来结点顺序相反，还好这不影响我们的讨论。

整个remove实现并不复杂，但是需要注意如下几点。第一，当要删除的结点存在时，删除的最后一步操作要将count的值减一。这必须是最后一步操作，否则读取操作可能看不到之前对段所做的结构性修改。第二，remove执行的开始就将table赋给一个局部变量tab，这是因为table是volatile变量，读写volatile变量的开销很大。编译器也不能对volatile变量的读写做任何优化，直接多次访问非volatile实例变量没有多大影响，编译器会做相应优化。

接下来看put操作，同样地put操作也是委托给段的put方法。下面是段的put方法：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. V put(K key, **int** hash, V value, **boolean** onlyIfAbsent) {
2. lock();
3. **try** {
4. **int** c = count;
5. **if** (c++ > threshold) // ensure capacity
6. rehash();
7. HashEntry<K,V>[] tab = table;
8. **int** index = hash & (tab.length - 1);
9. HashEntry<K,V> first = tab[index];
10. HashEntry<K,V> e = first;
11. **while** (e != **null** && (e.hash != hash || !key.equals(e.key)))
12. e = e.next;
14. V oldValue;
15. **if** (e != **null**) {
16. oldValue = e.value;
17. **if** (!onlyIfAbsent)
18. e.value = value;
19. }
20. **else** {
21. oldValue = **null**;
22. ++modCount;
23. tab[index] = **new** HashEntry<K,V>(key, hash, first, value);
24. count = c; // write-volatile
25. }
26. **return** oldValue;
27. } **finally** {
28. unlock();
29. }
30. }

该方法也是在持有段锁的情况下执行的，首先判断是否需要rehash，需要就先rehash。接着是找是否存在同样一个key的结点，如果存在就直接替换这个结点的值。否则创建一个新的结点并添加到hash链的头部，这时一定要修改modCount和count的值，同样修改count的值一定要放在最后一步。put方法调用了rehash方法，reash方法实现得也很精巧，主要利用了table的大小为2^n，这里就不介绍了。

修改操作还有putAll和replace。putAll就是多次调用put方法，没什么好说的。replace甚至不用做结构上的更改，实现要比put和delete要简单得多，理解了put和delete，理解replace就不在话下了，这里也不介绍了。

**获取操作**

首先看下get操作，同样ConcurrentHashMap的get操作是直接委托给Segment的get方法，直接看Segment的get方法：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. V get(Object key, **int** hash) {
2. **if** (count != 0) { // read-volatile
3. HashEntry<K,V> e = getFirst(hash);
4. **while** (e != **null**) {
5. **if** (e.hash == hash && key.equals(e.key)) {
6. V v = e.value;
7. **if** (v != **null**)
8. **return** v;
9. **return** readValueUnderLock(e); // recheck
10. }
11. e = e.next;
12. }
13. }
14. **return** **null**;
15. }

get操作不需要锁。第一步是访问count变量，这是一个volatile变量，由于所有的修改操作在进行结构修改时都会在最后一步写count变量，通过这种机制保证get操作能够得到几乎最新的结构更新。对于非结构更新，也就是结点值的改变，由于HashEntry的value变量是volatile的，也能保证读取到最新的值。接下来就是对hash链进行遍历找到要获取的结点，如果没有找到，直接访回null。对hash链进行遍历不需要加锁的原因在于链指针next是final的。但是头指针却不是final的，这是通过getFirst(hash)方法返回，也就是存在table数组中的值。这使得getFirst(hash)可能返回过时的头结点，例如，当执行get方法时，刚执行完getFirst(hash)之后，另一个线程执行了删除操作并更新头结点，这就导致get方法中返回的头结点不是最新的。这是可以允许，通过对count变量的协调机制，get能读取到几乎最新的数据，虽然可能不是最新的。要得到最新的数据，只有采用完全的同步。

最后，如果找到了所求的结点，判断它的值如果非空就直接返回，否则在有锁的状态下再读一次。这似乎有些费解，理论上结点的值不可能为空，这是因为put的时候就进行了判断，如果为空就要抛NullPointerException。空值的唯一源头就是HashEntry中的默认值，因为HashEntry中的value不是final的，非同步读取有可能读取到空值。仔细看下put操作的语句：tab[index] = new HashEntry<K,V>(key, hash, first, value)，在这条语句中，HashEntry构造函数中对value的赋值以及对tab[index]的赋值可能被重新排序，这就可能导致结点的值为空。这种情况应当很罕见，一旦发生这种情况，ConcurrentHashMap采取的方式是在持有锁的情况下再读一遍，这能够保证读到最新的值，并且一定不会为空值。

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. V readValueUnderLock(HashEntry<K,V> e) {
2. lock();
3. **try** {
4. **return** e.value;
5. } **finally** {
6. unlock();
7. }
8. }

另一个操作是containsKey，这个实现就要简单得多了，因为它不需要读取值：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **boolean containsKey(Object key, int hash) {**
2. **if** (count != 0) { // read-volatile
3. HashEntry<K,V> e = getFirst(hash);
4. **while** (e != **null**) {
5. **if** (e.hash == hash && key.equals(e.key))
6. **return** **true**;
7. e = e.next;
8. }
9. }
10. **return** **false**;
11. }

**跨段操作**

有些操作需要涉及到多个段，比如说size(), containsValaue()。先来看下size()方法：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **public int size() {**
2. **final** Segment<K,V>[] segments = **this**.segments;
3. **long** sum = 0;
4. **long** check = 0;
5. **int**[] mc = **new** **int**[segments.length];
6. // Try a few times to get accurate count. On failure due to
7. // continuous async changes in table, resort to locking.
8. **for** (**int** k = 0; k < RETRIES\_BEFORE\_LOCK; ++k) {
9. check = 0;
10. sum = 0;
11. **int** mcsum = 0;
12. **for** (**int** i = 0; i < segments.length; ++i) {
13. sum += segments[i].count;
14. mcsum += mc[i] = segments[i].modCount;
15. }
16. **if** (mcsum != 0) {
17. **for** (**int** i = 0; i < segments.length; ++i) {
18. check += segments[i].count;
19. **if** (mc[i] != segments[i].modCount) {
20. check = -1; // force retry
21. **break**;
22. }
23. }
24. }
25. **if** (check == sum)
26. **break**;
27. }
28. **if** (check != sum) { // Resort to locking all segments
29. sum = 0;
30. **for** (**int** i = 0; i < segments.length; ++i)
31. segments[i].lock();
32. **for** (**int** i = 0; i < segments.length; ++i)
33. sum += segments[i].count;
34. **for** (**int** i = 0; i < segments.length; ++i)
35. segments[i].unlock();
36. }
37. **if** (sum > Integer.MAX\_VALUE)
38. **return** Integer.MAX\_VALUE;
39. **else**
40. **return** (**int**)sum;
41. }

size方法主要思路是先在没有锁的情况下对所有段大小求和，如果不能成功（这是因为遍历过程中可能有其它线程正在对已经遍历过的段进行结构性更新），最多执行RETRIES\_BEFORE\_LOCK次，如果还不成功就在持有所有段锁的情况下再对所有段大小求和。在没有锁的情况下主要是利用Segment中的modCount进行检测，在遍历过程中保存每个Segment的modCount，遍历完成之后再检测每个Segment的modCount有没有改变，如果有改变表示有其它线程正在对Segment进行结构性并发更新，需要重新计算。

其实这种方式是存在问题的，在第一个内层for循环中，在这两条语句sum += segments[i].count; mcsum += mc[i] = segments[i].modCount;之间，其它线程可能正在对Segment进行结构性的修改，导致segments[i].count和segments[i].modCount读取的数据并不一致。这可能使size()方法返回任何时候都不曾存在的大小，很奇怪javadoc居然没有明确标出这一点，可能是因为这个时间窗口太小了吧。size()的实现还有一点需要注意，必须要先segments[i].count，才能segments[i].modCount，这是因为segment[i].count是对volatile变量的访问，接下来segments[i].modCount才能得到几乎最新的值（前面我已经说了为什么只是“几乎”了）。这点在containsValue方法中得到了淋漓尽致的展现：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **public boolean containsValue(Object value) {**
2. **if** (value == **null**)
3. **throw** **new** NullPointerException();
5. // See explanation of modCount use above
7. **final** Segment<K,V>[] segments = **this**.segments;
8. **int**[] mc = **new** **int**[segments.length];
10. // Try a few times without locking
11. **for** (**int** k = 0; k < RETRIES\_BEFORE\_LOCK; ++k) {
12. **int** sum = 0;
13. **int** mcsum = 0;
14. **for** (**int** i = 0; i < segments.length; ++i) {
15. **int** c = segments[i].count;
16. mcsum += mc[i] = segments[i].modCount;
17. **if** (segments[i].containsValue(value))
18. **return** **true**;
19. }
20. **boolean** cleanSweep = **true**;
21. **if** (mcsum != 0) {
22. **for** (**int** i = 0; i < segments.length; ++i) {
23. **int** c = segments[i].count;
24. **if** (mc[i] != segments[i].modCount) {
25. cleanSweep = **false**;
26. **break**;
27. }
28. }
29. }
30. **if** (cleanSweep)
31. **return** **false**;
32. }
33. // Resort to locking all segments
34. **for** (**int** i = 0; i < segments.length; ++i)
35. segments[i].lock();
36. **boolean** found = **false**;
37. **try** {
38. **for** (**int** i = 0; i < segments.length; ++i) {
39. **if** (segments[i].containsValue(value)) {
40. found = **true**;
41. **break**;
42. }
43. }
44. } **finally** {
45. **for** (**int** i = 0; i < segments.length; ++i)
46. segments[i].unlock();
47. }
48. **return** found;
49. }

同样注意内层的第一个for循环，里面有语句int c = segments[i].count; 但是c却从来没有被使用过，即使如此，编译器也不能做优化将这条语句去掉，因为存在对volatile变量count的读取，这条语句存在的唯一目的就是保证segments[i].modCount读取到几乎最新的值。关于containsValue方法的其它部分就不分析了，它和size方法差不多。

跨段方法中还有一个isEmpty()方法，其实现比size()方法还要简单，也不介绍了。最后简单地介绍下迭代方法，如keySet(), values(), entrySet()方法，这些方法都返回相应的迭代器，所有迭代器都继承于Hash\_Iterator类(提交时居然提醒我不能包含sh It，只得加了下划线)，里实现了主要的方法。其结构是：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **abstract class Hash\_Iterator{**
2. **int** nextSegmentIndex;
3. **int** nextTableIndex;
4. HashEntry<K,V>[] currentTable;
5. HashEntry<K, V> nextEntry;
6. HashEntry<K, V> lastReturned;
7. }

 nextSegmentIndex是段的索引，nextTableIndex是nextSegmentIndex对应段中中hash链的索引，currentTable是nextSegmentIndex对应段的table。调用next方法时主要是调用了advance方法：

**Java代码**

C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/dd9f6c2c32e84f02997ff7a7103eaf98/icon_star.png

1. **final void advance() {**
2. **if** (nextEntry != **null** && (nextEntry = nextEntry.next) != **null**)
3. **return**;
5. **while** (nextTableIndex >= 0) {
6. **if** ( (nextEntry = currentTable[nextTableIndex--]) != **null**)
7. **return**;
8. }
10. **while** (nextSegmentIndex >= 0) {
11. Segment<K,V> seg = segments[nextSegmentIndex--];
12. **if** (seg.count != 0) {
13. currentTable = seg.table;
14. **for** (**int** j = currentTable.length - 1; j >= 0; --j) {
15. **if** ( (nextEntry = currentTable[j]) != **null**) {
16. nextTableIndex = j - 1;
17. **return**;
18. }
19. }
20. }
21. }
22. }

不想再多介绍了，唯一需要注意的是跳到下一个段时，一定要先读取下一个段的count变量。

这种迭代方式的主要效果是不会抛出ConcurrentModificationException。一旦获取到下一个段的table，也就意味着这个段的头结点在迭代过程中就确定了，在迭代过程中就不能反映对这个段节点并发的删除和添加，对于节点的更新是能够反映的，因为节点的值是一个volatile变量。

测试程序：

**[java]** [view plaincopy](http://blog.csdn.net/xuefeng0707/article/details/40834595)

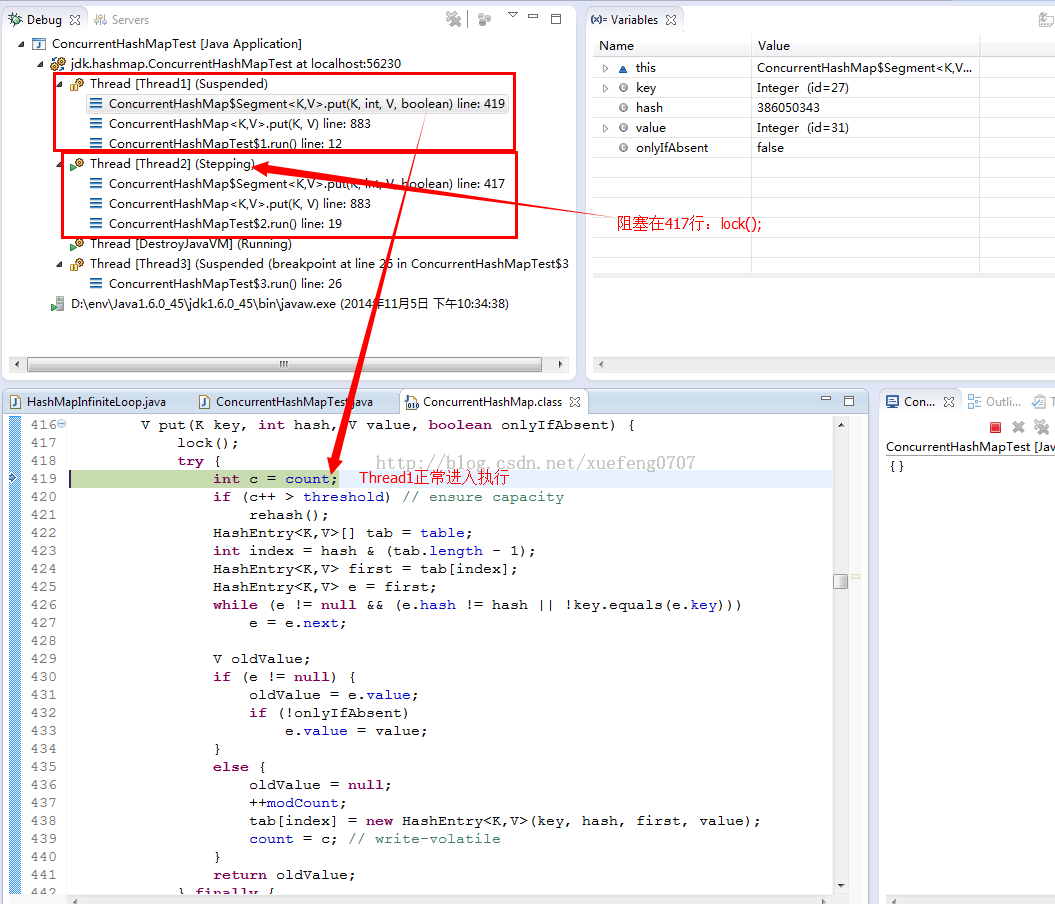
C://Users/10288/AppData/Local/YNote/data/keyguansz@163.com/551e058cd4da45be8a68d25e3ccc9548/code_ico.png

1. **import java.util.concurrent.ConcurrentHashMap;**
3. **public class ConcurrentHashMapTest {**
5. **private** **static** ConcurrentHashMap<Integer, Integer> map = **new** ConcurrentHashMap<Integer, Integer>();
6. **public** **static** **void** main(String[] args) {
7. **new** Thread("Thread1"){
8. @Override
9. **public** **void** run() {
10. map.put(3, 33);
11. }
12. };
14. **new** Thread("Thread2"){
15. @Override
16. **public** **void** run() {
17. map.put(4, 44);
18. }
19. };
21. **new** Thread("Thread3"){
22. @Override
23. **public** **void** run() {
24. map.put(7, 77);
25. }
26. };
27. System.out.println(map);
28. }
29. }

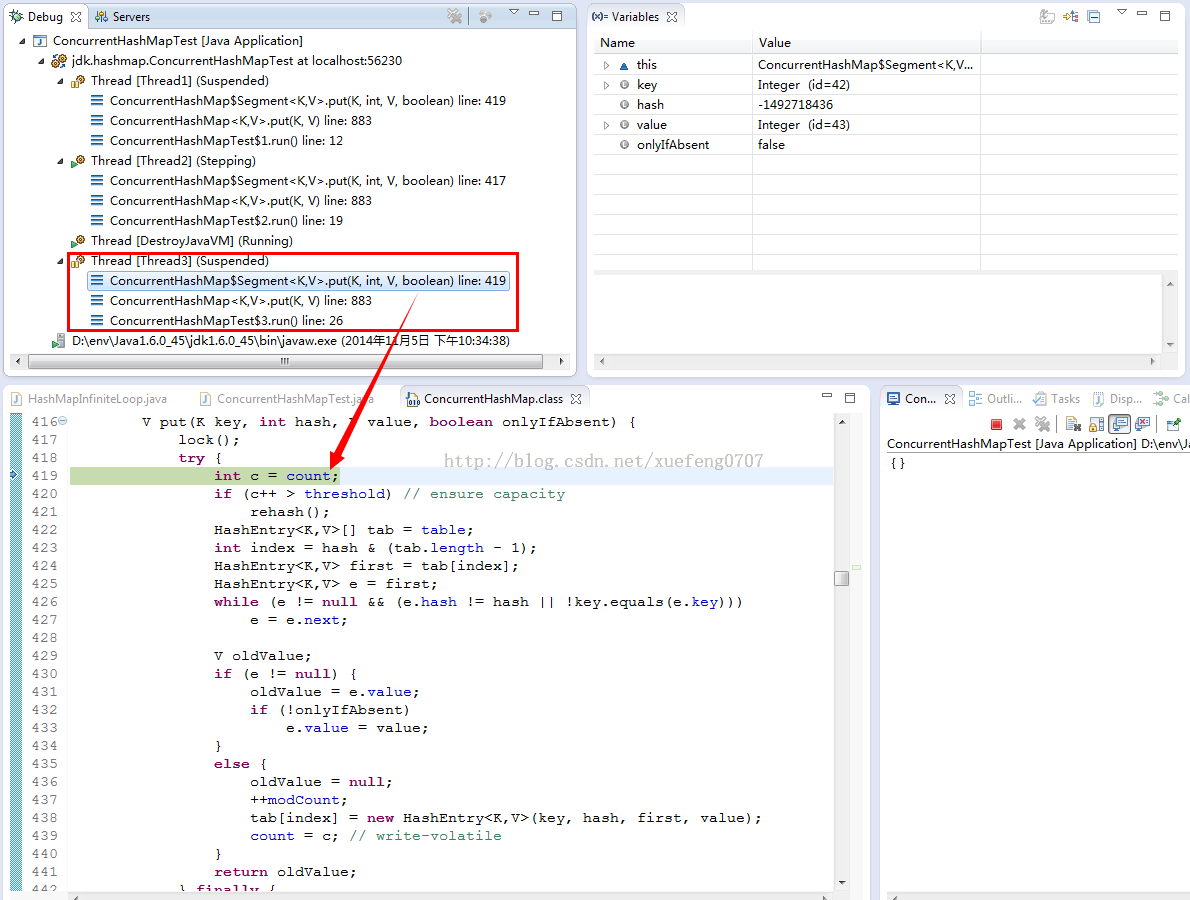
ConcurrentHashMap中默认是把segments初始化为长度为16的数组。

根据ConcurrentHashMap.segmentFor的算法，3、4对应的Segment都是segments[1]，7对应的Segment是segments[12]。

（1）Thread1和Thread2先后进入Segment.put方法时，Thread1会首先获取到锁，可以进入，而Thread2则会阻塞在锁上：



（2）切换到Thread3，也走到Segment.put方法，因为7所存储的Segment和3、4不同，因此，不会阻塞在lock()：



以上就是ConcurrentHashMap的工作机制，通过把整个Map分为N个Segment（类似HashTable），可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。

**结束语**

ConcurrentHashMap是一个支持高并发的高性能的HashMap实现，它支持完全并发的读以及一定程度并发的写。ConcurrentHashMap的实现也是很精巧，充分利用了最新的JMM规范，值得学习，却不值得模仿。最后由于本人水平有限，对大师的作品难免有误解，如果存在，还望大牛们不吝指出。

参考文章：

原文：<http://www.iteye.com/topic/344876>

**1.ConcurrentHashMap源码分析整理【重点参考】**

<http://www.molotang.com/articles/559.html>

2.[HashMap与ConcurrentHashMap的区别](http://blog.csdn.net/xuefeng0707/article/details/40834595)[demo]

<http://blog.csdn.net/xuefeng0707/article/details/40834595>