

ImportNew

- [首页](#)
- [所有文章](#)
- [资讯](#)
- [Web](#)
- [架构](#)
- [基础技术](#)
- [书籍](#)
- [教程](#)
- [Java小组](#)
- [工具资源](#)

- 导航条 - ▾

ConcurrentHashMap总结

2016/10/18 | 分类: [基础技术](#) | [4 条评论](#) | 标签: [ConcurrentHashMap](#), [并发](#)

分享到:

123 原文出处: [Hosee](#)

并发编程实践中, ConcurrentHashMap是一个经常被使用的数据结构, 相比于Hashtable以及Collections.synchronizedMap(), ConcurrentHashMap在线程安全的基础上提供了更好的写并发能力, 但同时降低了对读一致性的要求 (这点好像CAP理论啊 $O(n^2)$)。ConcurrentHashMap的设计与实现非常精巧, 大量的利用了volatile, final, CAS等lock-free技术来减少锁竞争对于性能的影响, 无论对于Java并发编程的学习还是Java内存模型的理解, ConcurrentHashMap的设计以及源码都值得非常仔细的阅读与揣摩。

这篇日志记录了自己对ConcurrentHashMap的一些总结, 由于JDK6, 7, 8中实现都不同, 需要分开阐述在不同版本中的ConcurrentHashMap。

之前已经在[ConcurrentHashMap原理分析](#)中解释了ConcurrentHashMap的原理, 主要是从代码的角度来阐述是源码是如何写的, 本文仍然从源码出发, 挑选个人觉得重要的点 (会用红色标注) 再次进行回顾, 以及阐述ConcurrentHashMap的一些注意点。

1. JDK6与JDK7中的实现

1.1 设计思路

ConcurrentHashMap采用了分段锁的设计, 只有在同一个分段内才存在竞态关系, 不同的分段锁之间没有锁竞争。相比于对整个Map加锁的设计, 分段锁大大的提高了高并发环境下的处理能力。但同时, 由于不是对整个Map加锁, 导致一些需要扫描整个Map的方法 (如size(), containsValue()) 需要使用特殊的实现, 另外一些方法 (如clear()) 甚至放弃了对一致性的要求

(ConcurrentHashMap是弱一致性的，具体请查看[ConcurrentHashMap能完全替代HashTable吗?](#))。

ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap（[JDK7与JDK8中HashMap的实现](#)）的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock）。ConcurrentHashMap中的HashEntry相对于HashMap中的Entry有一定的差异性：HashEntry中的value以及next都被volatile修饰，这样在多线程读写过程中能够保持它们的可见性，代码如下：

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

1.2 并发度 (Concurrency Level)

并发度可以理解为程序运行时能够同时更新ConcurrentHashMap且不产生锁竞争的最大线程数，实际上就是ConcurrentHashMap中的分段锁个数，即Segment[]的数组长度。ConcurrentHashMap默认的并发度为16，但用户也可以在构造函数中设置并发度。当用户设置并发度时，ConcurrentHashMap会使用大于等于该值的最小2幂指数作为实际并发度（假如用户设置并发度为17，实际并发度则为32）。运行时通过将key的高n位（ $n = 32 - \text{segmentShift}$ ）和并发度减1（segmentMask）做位与运算定位到所在的Segment。segmentShift与segmentMask都是在构造过程中根据concurrency level被相应的计算出来。

如果并发度设置的过小，会带来严重的锁竞争问题；如果并发度设置的过大，原本位于同一个Segment内的访问会扩散到不同的Segment中，CPU cache命中率会下降，从而引起程序性能下降。（文档的说法是根据你并发的线程数量决定，太多会导性能降低）



1.3 创建分段锁

和JDK6不同，JDK7中除了第一个Segment之外，剩余的Segments采用的是延迟初始化的机制：每次put之前都需要检查key对应的Segment是否为null，如果是则调用ensureSegment()以确保对应的Segment被创建。

ensureSegment可能在并发环境下被调用，但与想象中不同，ensureSegment并未使用锁来控制竞争，而是使用了Unsafe对象的getObjectVolatile()提供的原子读语义结合CAS来确保Segment创建的原子性。代码段如下：

```
1 if ((seg = (Segment<K,V>) UNSAFE.getObjectVolatile(ss, u))  
2     == null) { // recheck  
3     Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);  
4     while ((seg = (Segment<K,V>) UNSAFE.getObjectVolatile(ss, u))  
5           == null) {  
6         if (UNSAFE.compareAndSwapObject(ss, u, null, seg = s))  
7             break;  
8     }  
9 }
```

1.4 put/putIfAbsent/putAll

和JDK6一样，ConcurrentHashMap的put方法被代理到了对应的Segment（定位Segment的原理之前已经描述过）中。与JDK6不同的是，JDK7版本的ConcurrentHashMap在获得Segment锁的过程中，做了一定的优化 – 在真正申请锁之前，put方法会通过tryLock()方法尝试获得锁，在尝试获得锁

的过程中会对对应hashcode的链表进行遍历，如果遍历完毕仍然找不到与key相同的HashEntry节点，则为后续的put操作提前创建一个HashEntry。当tryLock一定次数后仍无法获得锁，则通过lock申请锁。

需要注意的是，由于在并发环境下，其他线程的put，rehash或者remove操作可能会导致链表头结点的变化，因此在过程中需要进行检查，如果头结点发生变化则重新对表进行遍历。而如果其他线程引起了链表中的某个节点被删除，即使该变化因为是非原子写操作（删除节点后链接后续节点调用的是Unsafe.putOrderedObject()，该方法不提供原子写语义）可能导致当前线程无法观察到，但因为不影响遍历的正确性所以忽略不计。


之所以在获取锁的过程中对整个链表进行遍历，主要目的是希望遍历的链表被CPU cache所缓存，为后续实际put过程中的链表遍历操作提升性能。

在获得锁之后，Segment对链表进行遍历，如果某个HashEntry节点具有相同的key，则更新该HashEntry的value值，否则新建一个HashEntry节点，将它设置为链表的新head节点并将原头节点设为新head的下一个节点。新建过程中如果节点总数（含新建的HashEntry）超过threshold，则调用rehash()方法对Segment进行扩容，最后将新建HashEntry写入到数组中。

put方法中，链接新节点的下一个节点（HashEntry.setNext()）以及将链表写入到数组中

（setEntryAt()）都是通过Unsafe的putOrderedObject()方法来实现，这里并未使用具有原子写语义的putObjectVolatile()的原因是：JMM会保证获得锁到释放锁之间所有对象的状态更新都会在锁被释放之后更新到主存，从而保证这些变更对其他线程是可见的。

1.5 rehash

相对于HashMap的resize，ConcurrentHashMap的rehash原理类似，但是Doug Lea为rehash做了一定的优化，避免让所有的节点都进行复制操作：于扩容是基于2的幂指来操作，假设扩容前某HashEntry对应到Segment中数组的index为i，数组的容量为capacity，那么扩容后该HashEntry对应到新数组中的index只可能为i或者i+capacity，因此大多数HashEntry节点在扩容前后index可以保持不变。基于此，rehash方法中会定位第一个后续所有节点在扩容后index都保持不变的节点，然后将这个节点之前的所有节点重排即可。这部分代码如下：

```

1  private void rehash(HashEntry<K,V> node) {
2      HashEntry<K,V>[] oldTable = table;
3      int oldCapacity = oldTable.length;
4      int newCapacity = oldCapacity << 1;
5      threshold = (int)(newCapacity * loadFactor);
6      HashEntry<K,V>[] newTable =
7          (HashEntry<K,V>[]) new HashEntry[newCapacity];
8      int sizeMask = newCapacity - 1;
9      for (int i = 0; i < oldCapacity ; i++) {
10         HashEntry<K,V> e = oldTable[i];
11         if (e != null) {
12             HashEntry<K,V> next = e.next;
13             int idx = e.hash & sizeMask;
14             if (next == null) // Single node on list
15                 newTable[idx] = e;
16             else { // Reuse consecutive sequence at same slot
17                 HashEntry<K,V> lastRun = e;
18                 int lastIdx = idx;
19                 for (HashEntry<K,V> last = next;
20                     last != null;
21                     last = last.next) {
22                     int k = last.hash & sizeMask;
23                     if (k != lastIdx) {
24                         lastIdx = k;
25                         lastRun = last;
26                     }
27                 }
28                 newTable[lastIdx] = lastRun;

```

```

29         // Clone remaining nodes
30         for (HashEntry<K,V> p = e; p != lastRun; p = p.next) {
31             V v = p.value;
32             int h = p.hash;
33             int k = h & sizeMask;
34             HashEntry<K,V> n = newTable[k];
35             newTable[k] = new HashEntry<K,V>(h, p.key, v, n);
36         }
37     }
38 }
39 }
40 int nodeIndex = node.hash & sizeMask; // add the new node
41 node.setNext(newTable[nodeIndex]);
42 newTable[nodeIndex] = node;
43 table = newTable;
44 }

```

1.6 remove

和put类似，remove在真正获得锁之前，也会对链表进行遍历以提高缓存命中率。

1.7 get与containsKey

get与containsKey两个方法几乎完全一致：他们都没有使用锁，而是通过Unsafe对象的getObjectVolatile()方法提供的原子读语义，来获得Segment以及对应的链表，然后对链表遍历判断是否存在key相同的节点以及获得该节点的value。但由于遍历过程中其他线程可能对链表结构做了调整，因此get和containsKey返回的可能是过时的数据，这一点是ConcurrentHashMap在弱一致性上的体现。如果要求强一致性，那么必须使用Collections.synchronizedMap()方法。

1.8 size、containsValue

这些方法都是基于整个ConcurrentHashMap来进行操作的，他们的原理也基本类似：首先不加锁循环执行以下操作：循环所有的Segment（通过Unsafe的getObjectVolatile()以保证原子读语义），获得对应的值以及所有Segment的modcount之和。如果连续两次所有Segment的modcount和相等，则过程中没有发生其他线程修改ConcurrentHashMap的情况，返回获得的值。

当循环次数超过预定义的值时，这时需要对所有的Segment依次进行加锁，获取返回值后再依次解锁。值得注意的是，加锁过程中要强制创建所有的Segment，否则容易出现其他线程创建Segment并进行put，remove等操作。代码如下：

```

1  for(int j =0; j < segments.length; ++j)
2
3  ensureSegment(j).lock();// force creation

```

一般来说，应该避免在多线程环境下使用size和containsValue方法。

注1：modcount在put, replace, remove以及clear等方法中都会被修改。

注2：对于containsValue方法来说，如果在循环过程中发现匹配value的HashEntry，则直接返回true。

最后，与HashMap不同的是，ConcurrentHashMap并不允许key或者value为null，按照Doug Lea的说法，这么设计的原因是在ConcurrentHashMap中，一旦value出现null，则代表HashEntry的key/value没有映射完成就被其他线程所见，需要特殊处理。在JDK6中，get方法的实现中就有一段对HashEntry.value == null的防御性判断。但Doug Lea也承认实际运行过程中，这种情况似乎不可能发生（参考：<http://cs.oswego.edu/pipermail/concurrency-interest/2011-March/007799.html>）。

2. JDK8中的实现

ConcurrentHashMap在JDK8中进行了巨大改动，很需要通过源码来再次学习下Doug Lea的实现方法。


它摒弃了Segment（锁段）的概念，而是启用了一种全新的方式实现，利用CAS算法。它沿用了与它同时期的HashMap版本的思想，底层依然由“数组”+链表+红黑树的方式思想([JDK7与JDK8中HashMap的实现](#))，但是为了做到并发，又增加了很多辅助的类，例如TreeBin，Traverser等对象内部类。

2.1 重要的属性

首先来看几个重要的属性，与HashMap相同的就不再介绍了，这里重点解释一下sizeCtl这个属性。可以说它是ConcurrentHashMap中出镜率很高的一个属性，因为它是一个控制标识符，在不同的地方有不同用途，而且它的取值不同，也代表不同的含义。

- 负数代表正在进行初始化或扩容操作
- -1代表正在初始化
- -N 表示有N-1个线程正在进行扩容操作
- 正数或0代表hash表还没有被初始化，这个数值表示初始化或下一次进行扩容的大小，这一点类似于扩容阈值的概念。后面可以看到，它的值始终是当前ConcurrentHashMap容量的0.75倍，这与loadfactor是对应的。

```

1  /**
2   * 盛装Node元素的数组 它的大小是2的整数次幂 
3   * Size is always a power of two. Accessed directly by iterators.
4   */
5   transient volatile Node<K,V>[] table;
6
7   /**
8   * Table initialization and resizing control. When negative, the
9   * table is being initialized or resized: -1 for initialization,
10  * else -(1 + the number of active resizing threads). Otherwise,
11  * when table is null, holds the initial table size to use upon
12  * creation, or 0 for default. After initialization, holds the
13  * next element count value upon which to resize the table.
14  * hash表初始化或扩容时的一个控制位标识量。
15  * 负数代表正在进行初始化或扩容操作
16  * -1代表正在初始化
17  * -N 表示有N-1个线程正在进行扩容操作
18  * 正数或0代表hash表还没有被初始化，这个数值表示初始化或下一次进行扩容的大小
19
20  */
21  private transient volatile int sizeCtl;
22  // 以下两个是用来控制扩容的时候 单线程进入的变量
23  /**
24   * The number of bits used for generation stamp in sizeCtl.
25   * Must be at least 6 for 32bit arrays.
26   */
27  private static int RESIZE_STAMP_BITS = 16;
28  /**
29   * The bit shift for recording size stamp in sizeCtl.
30   */
31  private static final int RESIZE_STAMP_SHIFT = 32 - RESIZE_STAMP_BITS;
32
33  /**
34   * Encodings for Node hash fields. See above for explanation.
35   */
36  static final int MOVED = -1; // hash值是-1, 表示这是一个forwardNode节点
37  static final int TREEBIN = -2; // hash值是-2 表示这时一个TreeBin节点

```

2.2 重要的类

2.2.1 Node

Node是最核心的内部类，它包装了key-value键值对，所有插入ConcurrentHashMap的数据都包装在这里面。它与HashMap中的定义很相似，但是有一些差别它对value和next属性设置了volatile同步锁(与JDK7的Segment相同)，它不允许调用setValue方法直接改变Node的value域，它增加了find方法辅助map.get()方法。

2.2.2 TreeNode

TreeNode是树节点类，另外一个核心的数据结构。当链表长度过长的时候，会转换为TreeNode。但是与HashMap不相同的是，它并不是直接转换为红黑树，而是把这些结点包装成TreeNode放在TreeBin对象中，由TreeBin完成对红黑树的包装。而且TreeNode在ConcurrentHashMap集成自Node类，而并非HashMap中的集成自LinkedHashMap.Entry<K,V>类，也就是说TreeNode带有next指针，这样做的目的是方便基于TreeBin的访问。

2.2.3 TreeBin

这个类并不负责包装用户的key、value信息，而是包装的很多TreeNode节点。它代替了TreeNode的根节点，也就是说在实际的ConcurrentHashMap“数组”中，存放的是TreeBin对象，而不是TreeNode对象，这是与HashMap的区别。另外这个类还带有了读写锁。

这里仅贴出它的构造方法。可以看到在构造TreeBin节点时，仅仅指定了它的hash值为TREEBIN常量，这也就是个标识为。同时也看到我们熟悉的红黑树构造方法



2.2.4 ForwardingNode

一个用于连接两个table的节点类。它包含一个nextTable指针，用于指向下一张表。而且这个节点的key value next指针全部为null，它的hash值为-1。这里面定义的find的方法是从nextTable里进行查询节点，而不是以自身为头节点进行查找。

```
1  /**
2   * A node inserted at head of bins during transfer operations.
3   */
4   static final class ForwardingNode<K,V> extends Node<K,V> {
5       final Node<K,V>[] nextTable;
6       ForwardingNode(Node<K,V>[] tab) {
7           super(MOVED, null, null, null);
8           this.nextTable = tab;
9       }
10
11      Node<K,V> find(int h, Object k) {
12          // loop to avoid arbitrarily deep recursion on forwarding nodes
13          outer: for (Node<K,V>[] tab = nextTable;;) {
14              Node<K,V> e; int n;
15              if (k == null || tab == null || (n = tab.length) == 0 ||
16                  (e = tabAt(tab, (n - 1) & h)) == null)
17                  return null;
18              for (;;) {
19                  int eh; K ek;
20                  if ((eh = e.hash) == h &&
21                      ((ek = e.key) == k || (ek != null && k.equals(ek))))
22                      return e;
23                  if (eh < 0) {
24                      if (e instanceof ForwardingNode) {
25                          tab = ((ForwardingNode<K,V>)e).nextTable;
26                          continue outer;
27                      }
28                      else
```



```

29         return e.find(h, k);
30     }
31     if ((e = e.next) == null)
32         return null;
33     }
34 }
35 }
36 }

```

2.3 Unsafe与CAS

在ConcurrentHashMap中，随处可以看到U，大量使用了U.compareAndSwapXXX的方法，这个方法是利用一个CAS算法实现无锁化的修改值的操作，他可以大大降低锁代理的性能消耗。这个算法的基本思想就是不断地去比较当前内存中的变量值与你指定的一个变量值是否相等，如果相等，则接受你指定的修改的值，否则拒绝你的操作。因为当前线程中的值已经不是最新的值，你的修改很可能会覆盖掉其他线程修改的结果。这一点与乐观锁，SVN的思想是比较类似的。

2.3.1 unsafe静态块

unsafe代码块控制了一些属性的修改工作，比如最常用的SIZECTL。在这一版本的concurrentHashMap中，大量应用来的CAS方法进行变量、属性的修改工作。利用CAS进行无锁操作，可以大大提高性能。

```

1  private static final sun.misc.Unsafe U;
2  private static final long SIZECTL;
3  private static final long TRANSFERINDEX;
4  private static final long BASECOUNT;
5  private static final long CELLSBUSY;
6  private static final long CELLVALUE;
7  private static final long ABASE;
8  private static final int ASHIFT;
9
10 static {
11     try {
12         U = sun.misc.Unsafe.getUnsafe();
13         Class<?> k = ConcurrentHashMap.class;
14         SIZECTL = U.objectFieldOffset
15             (k.getDeclaredField("sizeCtl"));
16         TRANSFERINDEX = U.objectFieldOffset
17             (k.getDeclaredField("transferIndex"));
18         BASECOUNT = U.objectFieldOffset
19             (k.getDeclaredField("baseCount"));
20         CELLSBUSY = U.objectFieldOffset
21             (k.getDeclaredField("cellsBusy"));
22         Class<?> ck = CounterCell.class;
23         CELLVALUE = U.objectFieldOffset
24             (ck.getDeclaredField("value"));
25         Class<?> ak = Node[].class;
26         ABASE = U.arrayBaseOffset(ak);
27         int scale = U.arrayIndexScale(ak);
28         if ((scale & (scale - 1)) != 0)
29             throw new Error("data type scale not a power of two");
30         ASHIFT = 31 - Integer.numberOfLeadingZeros(scale);
31     } catch (Exception e) {
32         throw new Error(e);
33     }
34 }

```

2.3.2 三个核心方法

ConcurrentHashMap定义了三个原子操作，用于对指定位置的节点进行操作。正是这些原子操作保证了ConcurrentHashMap的线程安全。

```

1  //获得在i位置上的Node节点
2  static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
3      return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
4  }

```

```

5 //利用CAS算法设置i位置上的Node节点。之所以能实现并发是因为他指定了原来这个节点的值是多少
6 //在CAS算法中,会比较内存中的值与你指定的这个值是否相等,如果相等才接受你的修改,否则拒绝你的修改
7 //因此当前线程中的值并不是最新的值,这种修改可能会覆盖掉其他线程的修改结果 有点类似于SVN
8 static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
9                                     Node<K,V> c, Node<K,V> v) {
10     return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
11 }
12 //利用volatile方法设置节点位置的值
13 static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {
14     U.putObjectVolatile(tab, ((long)i << ASHIFT) + ABASE, v);
15 }

```

2.4 初始化方法initTable

对于ConcurrentHashMap来说,调用它的构造方法仅仅是设置了一些参数而已。而整个table的初始化是在向ConcurrentHashMap中插入元素的时候发生的。如调用put、computeIfAbsent、compute、merge等方法的时候,调用时机是检查table==null。

初始化方法主要应用了关键属性sizeCtl 如果这个值 < 0, 表示其他线程正在进行初始化,就放弃这个操作。在这也可以看出ConcurrentHashMap的初始化只能由一个线程完成。如果获得了初始化权限,就用CAS方法将sizeCtl置为-1,防止其他线程进入。初始化数组后,将sizeCtl的值改为 $0.75 * n$ 。

```

1 /**
2  * Initializes table, using the size recorded in sizeCtl.
3  */
4 private final Node<K,V>[] initTable() {
5     Node<K,V>[] tab; int sc;
6     while ((tab = table) == null || tab.length == 0) {
7         //sizeCtl表示有其他线程正在进行初始化操作,把线程挂起。对于table的初始化工作,只能有一个
8         if ((sc = sizeCtl) < 0)
9             Thread.yield(); // lost initialization race; just spin
10        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) { //利用CAS方法把sizeCtl的值置
11            try {
12                if ((tab = table) == null || tab.length == 0) {
13                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
14                    @SuppressWarnings("unchecked")
15                    Node<K,V>[] nt = (Node<K,V>[]) new Node<?,?>[n];
16                    table = tab = nt;
17                    sc = n - (n >>> 2); //相当于0.75*n 设置一个扩容的阈值
18                }
19            } finally {
20                sizeCtl = sc;
21            }
22            break;
23        }
24    }
25    return tab;
26 }

```

2.5 扩容方法 transfer

当ConcurrentHashMap容量不足的时候,需要对table进行扩容。这个方法的基本思想跟HashMap是很像的,但是由于它是支持并发扩容的,所以要复杂的多。原因是它支持多线程进行扩容操作,而并没有加锁。我想这样做的目的不仅仅是为了满足concurrent的要求,而是希望利用并发处理去减少扩容带来的时间影响。因为在扩容的时候,总是会涉及到从一个“数组”到另一个“数组”拷贝的操作,如果这个操作能够并发进行,那真真是极好的了。

整个扩容操作分为两个部分

- 第一部分是构建一个nextTable,它的容量是原来的两倍,这个操作是单线程完成的。这个单线程的保证是通过RESIZE_STAMP_SHIFT这个常量经过一次运算来保证的,这个地方在后面会

有提到；

- 第二个部分就是将原来table中的元素复制到nextTable中，这里允许多线程进行操作。

先来看一下单线程是如何完成的：

它的大体思想就是遍历、复制的过程。首先根据运算得到需要遍历的次数i，然后利用tabAt方法获得i位置的元素：

- 如果这个位置为空，就在原table中的i位置放入forwardNode节点，这个也是触发并发扩容的关键点；
- 如果这个位置是Node节点（fh>=0），如果它是一个链表的头节点，就构造一个反序链表，把他们分别放在nextTable的i和i+n的位置上
- 如果这个位置是TreeBin节点（fh<0），也做一个反序处理，并且判断是否需要untreefi，把处理的结果分别放在nextTable的i和i+n的位置上
- 遍历过所有的节点以后就完成了复制工作，这时让nextTable作为新的table，并且更新sizeCtl为新容量的0.75倍，完成扩容。

再看一下多线程是如何完成的：

在代码的69行有一个判断，如果遍历到的节点是forward节点，就向后继续遍历，再加上给节点上锁的机制，就完成了多线程的控制。多线程遍历节点，处理了一个节点，就把对应点的值set为forward，另一个线程看到forward，就向后遍历。这样交叉就完成了复制工作。而且还很好的解决了线程安全的问题。这个方法的设计实在是让我膜拜。



```

1  /**
2   * 一个过渡的table表 只有在扩容的时候才会使用
3   */
4   private transient volatile Node<K,V>[] nextTable;
5
6   /**
7   * Moves and/or copies the nodes in each bin to new table. See
8   * above for explanation.
9   */
10  private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
11      int n = tab.length, stride;
12      if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
13          stride = MIN_TRANSFER_STRIDE; // subdivide range
14      if (nextTab == null) {           // initiating
15          try {
16              @SuppressWarnings("unchecked")
17              Node<K,V>[] nt = (Node<K,V>[]) new Node<?,?>[n << 1]; //构造一个nextTable对象 它
18              nextTab = nt;
19          } catch (Throwable ex) {      // try to cope with OOME
20              sizeCtl = Integer.MAX_VALUE;
21              return;
22          }
23          nextTable = nextTab;
24          transferIndex = n;
25      }
26      int nextn = nextTab.length;
27      ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab); //构造一个连节点指针 用于标志
28      boolean advance = true; //并发扩容的关键属性 如果等于true 说明这个节点已经处理过
29      boolean finishing = false; // to ensure sweep before committing nextTab
30      for (int i = 0, bound = 0;;) {
31          Node<K,V> f; int fh;
32          //这个while循环体的作用就是在控制i-- 通过i--可以依次遍历原hash表中的节点
33          while (advance) {

```

```

34     int nextIndex, nextBound;
35     if (--i >= bound || finishing)
36         advance = false;
37     else if ((nextIndex = transferIndex) <= 0) {
38         i = -1;
39         advance = false;
40     }
41     else if (U.compareAndSwapInt
42         (this, TRANSFERINDEX, nextIndex,
43          nextBound = (nextIndex > stride ?
44                      nextIndex - stride : 0))) {
45         bound = nextBound;
46         i = nextIndex - 1;
47         advance = false;
48     }
49 }
50 if (i < 0 || i >= n || i + n >= nextn) {
51     int sc;
52     if (finishing) {
53         //如果所有的节点都已经完成复制工作 就把nextTable赋值给table 清空临时对象nextTab
54         nextTable = null;
55         table = nextTab;
56         sizeCtl = (n << 1) - (n >>> 1); //扩容阈值设置为原来容量的1.5倍 依然相当于现在
57         return;
58     }
59     //利用CAS方法更新这个扩容阈值, 在这里面sizeCtl值减一, 说明新加入一个线程参与到扩容操作
60     if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
61         if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
62             return;
63         finishing = advance = true;
64         i = n; // recheck before commit
65     }
66 }
67 //如果遍历到的节点为空 则放入ForwardingNode指针
68 else if ((f = tabAt(tab, i)) == null)
69     advance = casTabAt(tab, i, null, fwd);
70 //如果遍历到ForwardingNode节点 说明这个点已经被处理过了 直接跳过 这里是控制并发扩容的核心
71 else if ((fh = f.hash) == MOVED)
72     advance = true; // already processed
73 else {
74     //节点上锁
75     synchronized (f) {
76         if (tabAt(tab, i) == f) {
77             Node<K,V> ln, hn;
78             //如果fh>=0 证明这是一个Node节点
79             if (fh >= 0) {
80                 int runBit = fh & n;
81                 //以下的部分在完成的工作是构造两个链表 一个是原链表 另一个是原链表的反序
82                 Node<K,V> lastRun = f;
83                 for (Node<K,V> p = f.next; p != null; p = p.next) {
84                     int b = p.hash & n;
85                     if (b != runBit) {
86                         runBit = b;
87                         lastRun = p;
88                     }
89                 }
90                 if (runBit == 0) {
91                     ln = lastRun;
92                     hn = null;
93                 }
94                 else {
95                     hn = lastRun;
96                     ln = null;
97                 }
98                 for (Node<K,V> p = f; p != lastRun; p = p.next) {
99                     int ph = p.hash; K pk = p.key; V pv = p.val;
100                     if ((ph & n) == 0)
101                         ln = new Node<K,V>(ph, pk, pv, ln);
102                     else
103                         hn = new Node<K,V>(ph, pk, pv, hn);
104                 }
105                 //在nextTable的i位置上插入一个链表
106                 setTabAt(nextTab, i, ln);
107                 //在nextTable的i+n的位置上插入另一个链表
108                 setTabAt(nextTab, i + n, hn);
109                 //在table的i位置上插入forwardNode节点 表示已经处理过该节点
110                 setTabAt(tab, i, fwd);
111                 //设置advance为true 返回到上面的while循环中 就可以执行i--操作
112                 advance = true;
113             }

```

```

114 //对TreeBin对象进行处理 与上面的过程类似
115 else if (f instanceof TreeBin) {
116     TreeBin<K,V> t = (TreeBin<K,V>)f;
117     TreeNode<K,V> lo = null, loTail = null;
118     TreeNode<K,V> hi = null, hiTail = null;
119     int lc = 0, hc = 0;
120     //构造正序和反序两个链表
121     for (Node<K,V> e = t.first; e != null; e = e.next) {
122         int h = e.hash;
123         TreeNode<K,V> p = new TreeNode<K,V>
124             (h, e.key, e.val, null, null);
125         if ((h & n) == 0) {
126             if ((p.prev = loTail) == null)
127                 lo = p;
128             else
129                 loTail.next = p;
130             loTail = p;
131             ++lc;
132         }
133         else {
134             if ((p.prev = hiTail) == null)
135                 hi = p;
136             else
137                 hiTail.next = p;
138             hiTail = p;
139             ++hc;
140         }
141     }
142     //如果扩容后已经不再需要tree的结构 反向转换为链表结构
143     ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
144         (hc != 0) ? new TreeBin<K,V>(lo) : t;
145     hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
146         (lc != 0) ? new TreeBin<K,V>(hi) : t;
147     //在nextTable的i位置上插入一个链表
148     setTabAt(nextTab, i, ln);
149     //在nextTable的i+n的位置上插入另一个链表
150     setTabAt(nextTab, i + n, hn);
151     //在table的i位置上插入forwardNode节点 表示已经处理过该节点
152     setTabAt(tab, i, fwd);
153     //设置advance为true 返回到上面的while循环中 就可以执行i--操作
154     advance = true;
155 }
156 }
157 }
158 }
159 }
160 }

```



2.6 Put方法

前面的所有的介绍其实都为这个方法做铺垫。ConcurrentHashMap最常用的就是put和get两个方法。现在来介绍put方法，这个put方法依然沿用HashMap的put方法的思想，根据hash值计算这个新插入的点在table中的位置i，如果i位置是空的，直接放进去，否则进行判断，如果i位置是树节点，按照树的方式插入新的节点，否则把i插入到链表的末尾。ConcurrentHashMap中依然沿用这个思想，有一个最重要的不同点就是ConcurrentHashMap不允许key或value为null值。另外由于涉及到多线程，put方法就要复杂一点。在多线程中可能有以下两个情况

1. 如果一个或多个线程正在对ConcurrentHashMap进行扩容操作，当前线程也要进入扩容的操作中。这个扩容的操作之所以能被检测到，是因为transfer方法中在空结点上插入forward节点，如果检测到需要插入的位置被forward节点占有，就帮助进行扩容；
2. 如果检测到要插入的节点是非空且不是forward节点，就对这个节点加锁，这样就保证了线程安全。尽管这个有一些影响效率，但是还是会比hashTable的synchronized要好得多。

整体流程就是首先定义不允许key或value为null的情况放入 对于每一个放入的值，首先利用spread方法对key的hashCode进行一次hash计算，由此来确定这个值在table中的位置。

如果这个位置是空的，那么直接放入，而且不需要加锁操作。

如果这个位置存在结点，说明发生了hash碰撞，首先判断这个结点的类型。如果是链表节点 (fh>0) ,则得到的结点就是hash值相同的节点组成的链表的头节点。需要依次向后遍历确定这个新加入的值所在位置。如果遇到hash值与key值都与新加入节点是一致的情况，则只需要更新value值即可。否则依次向后遍历，直到链表尾插入这个结点。如果加入这个节点以后链表长度大于8，就把这个链表转换成红黑树。如果这个节点的类型已经是树节点的话，直接调用树节点的插入方法进行插入新的值。

```

1  public V put(K key, V value) {
2      return putVal(key, value, false);
3  }
4
5  /** Implementation for put and putIfAbsent */
6  final V putVal(K key, V value, boolean onlyIfAbsent) {
7      //不允许 key或value为null
8      if (key == null || value == null) throw new NullPointerException();
9      //计算hash值
10     int hash = spread(key.hashCode());
11     int binCount = 0;
12     //死循环 何时插入成功 何时跳出
13     for (Node<K,V>[] tab = table;;) {
14         Node<K,V> f; int n, i, fh;
15         //如果table为空的话，初始化table
16         if (tab == null || (n = tab.length) == 0)
17             tab = initTable();
18         //根据hash值计算出在table里面的位置
19         else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
20             //如果这个位置没有值，直接放进去，不需要加锁
21             if (casTabAt(tab, i, null,
22                 new Node<K,V>(hash, key, value, null)))
23                 break; // no lock when adding to empty bin
24         }
25         //当遇到表连接点时，需要进行整合表的操作
26         else if ((fh = f.hash) == MOVED)
27             tab = helpTransfer(tab, f);
28         else {
29             V oldVal = null;
30             //结点上锁 这里的结点可以理解为hash值相同组成的链表的头结点
31             synchronized (f) {
32                 if (tabAt(tab, i) == f) {
33                     //fh 0 说明这个节点是一个链表的节点 不是树的节点
34                     if (fh >= 0) {
35                         binCount = 1;
36                         //在这里遍历链表所有的结点
37                         for (Node<K,V> e = f;; ++binCount) {
38                             K ek;
39                             //如果hash值和key值相同 则修改对应结点的value值
40                             if (e.hash == hash &&
41                                 ((ek = e.key) == key ||
42                                 (ek != null && key.equals(ek)))) {
43                                 oldVal = e.val;
44                                 if (!onlyIfAbsent)
45                                     e.val = value;
46                                 break;
47                             }
48                             Node<K,V> pred = e;
49                             //如果遍历到了最后一个结点，那么就证明新的节点需要插入 就把它插入在链表
50                             if ((e = e.next) == null) {
51                                 pred.next = new Node<K,V>(hash, key,
52                                     value, null);
53                                 break;
54                             }
55                         }
56                     }
57                     //如果这个节点是树节点，就按照树的方式插入值
58                     else if (f instanceof TreeBin) {
59                         Node<K,V> p;
60                         binCount = 2;
61                         if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
62                             value)) != null) {
63                             oldVal = p.val;
64                             if (!onlyIfAbsent)
65                                 p.val = value;
66                         }
67                     }
68                 }
69             }
70         }
71     }
72     return oldVal;
73 }

```

```

70         if (binCount != 0) {
71             //如果链表长度已经达到临界值8 就需要把链表转换为树结构
72             if (binCount >= TREEIFY_THRESHOLD)
73                 treeifyBin(tab, i);
74             if (oldVal != null)
75                 return oldVal;
76             break;
77         }
78     }
79 }
80 //将当前ConcurrentHashMap的元素数量+1
81 addCount(1L, binCount);
82 return null;
83 }

```

我们可以发现JDK8中的实现也是锁分离的思想，只是锁住的是一个Node，而不是JDK7中的Segment，而锁住Node之前的操作是无锁的并且也是线程安全的，建立在之前提到的3个原子操作上。

2.6.1 helpTransfer方法

这是一个协助扩容的方法。这个方法被调用的时候，当前ConcurrentHashMap一定已经有了nextTable对象，首先拿到这个nextTable对象，调用transfer方法。回看上面的transfer方法可以看到，当本线程进入扩容方法的时候会直接进入复制阶段。

2.6.2 treeifyBin方法

这个方法用于将过长的链表转换为TreeBin对象。但是他并不是直接转换，而是进行一次容量判断，如果容量没有达到转换的要求，直接进行扩容操作并返回；如果满足条件才链表的结构抓换为TreeBin，这与HashMap不同的是，它并没有把TreeNode直接放入红黑树，而是利用了TreeBin这个小容器来封装所有的TreeNode。



2.7 get方法

get方法比较简单，给定一个key来确定value的时候，必须满足两个条件 key相同 hash值相同，对于节点可能在链表或树上的情况，需要分别去查找。

```

1  public V get(Object key) {
2      Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
3      //计算hash值
4      int h = spread(key.hashCode());
5      //根据hash值确定节点位置
6      if ((tab = table) != null && (n = tab.length) > 0 &&
7          (e = tabAt(tab, (n - 1) & h)) != null) {
8          //如果搜索到的节点key与传入的key相同且不为null,直接返回这个节点
9          if ((eh = e.hash) == h) {
10             if ((ek = e.key) == key || (ek != null && key.equals(ek)))
11                 return e.val;
12             }
13         //如果eh<0 说明这个节点在树上 直接寻找
14         else if (eh < 0)
15             return (p = e.find(h, key)) != null ? p.val : null;
16         //否则遍历链表 找到对应的值并返回
17         while ((e = e.next) != null) {
18             if (e.hash == h &&
19                 ((ek = e.key) == key || (ek != null && key.equals(ek))))
20                 return e.val;
21         }
22     }
23     return null;
24 }

```

2.8 Size相关的方法

对于ConcurrentHashMap来说，这个table里到底装了多少东西其实是个不确定的数量，因为不可能在调用size()方法的时候像GC的“stop the world”一样让其他线程都停下来让你去统计，因此只能说这个数量是个估计值。对于这个估计值，ConcurrentHashMap也是大费周章才计算出来的。

2.8.1 辅助定义

为了统计元素个数，ConcurrentHashMap定义了一些变量和一个内部类

```

1  /**
2   * A padded cell for distributing counts. Adapted from LongAdder
3   * and Striped64. See their internal docs for explanation.
4   */
5   @sun.misc.Contended static final class CounterCell {
6       volatile long value;
7       CounterCell(long x) { value = x; }
8   }
9
10  /**
11   *
12   * 实际上保存的是hashmap中的元素个数 利用CAS锁进行更新
13   * 但它并不用返回当前hashmap的元素个数
14   */
15
16   */
17   private transient volatile long baseCount;
18   /**
19   * Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
20   */
21   private transient volatile int cellsBusy;
22
23   /**
24   * Table of counter cells. When non-null, size is a power of 2.
25   */
26   private transient volatile CounterCell[] counterCells;

```

2.8.2 mappingCount与Size方法



mappingCount与size方法的类似 从Java工程师给出的注释来看，应该使用mappingCount代替size方法 两个方法都没有直接返回basecount 而是统计一次这个值，而这个值其实也是一个大概的数值，因此可能在统计的时候有其他线程正在执行插入或删除操作。

```

1  public int size() {
2      long n = sumCount();
3      return ((n < 0L) ? 0 :
4              (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
5              (int)n);
6  }
7
8  /**
9   * Returns the number of mappings. This method should be used
10   * instead of {@link #size} because a ConcurrentHashMap may
11   * contain more mappings than can be represented as an int. The
12   * value returned is an estimate; the actual count may differ if
13   * there are concurrent insertions or removals.
14   *
15   * @return the number of mappings
16   * @since 1.8
17   */
18   public long mappingCount() {
19       long n = sumCount();
20       return (n < 0L) ? 0L : n; // ignore transient negative values
21   }
22
23   final long sumCount() {
24       CounterCell[] as = counterCells; CounterCell a;
25       long sum = baseCount;
26       if (as != null) {
27           for (int i = 0; i < as.length; ++i) {
28               if ((a = as[i]) != null)
29                   sum += a.value; //所有counter的值求和
30           }
31       }
32   }

```



```

31     return sum;
32 }

```

2.8.3 addCount方法

在put方法结尾处调用了addCount方法，把当前ConcurrentHashMap的元素个数+1这个方法一共做了两件事,更新baseCount的值，检测是否进行扩容。

```

1  private final void addCount(long x, int check) {
2      CounterCell[] as; long b, s;
3      //利用CAS方法更新baseCount的值
4      if ((as = counterCells) != null ||
5          !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
6          CounterCell a; long v; int m;
7          boolean uncontended = true;
8          if (as == null || (m = as.length - 1) < 0 ||
9              (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
10             !(uncontended =
11                 U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
12              fullAddCount(x, uncontended);
13              return;
14          }
15          if (check <= 1)
16              return;
17          s = sumCount();
18      }
19      //如果check值大于等于0 则需要检验是否需要扩容操作
20      if (check >= 0) {
21          Node<K,V>[] tab, nt; int n, sc;
22          while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
23              (n = tab.length) < MAXIMUM_CAPACITY) {
24              int rs = resizeStamp(n);
25              //
26              if (sc < 0) {
27                  if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
28                      sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
29                      transferIndex <= 0)
30                      break;
31                  //如果已经有其他线程在执行扩容操作
32                  if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
33                      transfer(tab, nt);
34              }
35              //当前线程是唯一的或是第一个发起扩容的线程 此时nextTable=null
36              else if (U.compareAndSwapInt(this, SIZECTL, sc,
37                  (rs << RESIZE_STAMP_SHIFT) + 2))
38                  transfer(tab, null);
39              s = sumCount();
40          }
41      }
42  }

```

总结

JDK6,7中的ConcurrentHashMap主要使用Segment来实现减小锁粒度，把HashMap分割成若干个Segment，在put的时候需要锁住Segment，get时候不加锁，使用volatile来保证可见性，当要统计全局时（比如size），首先会尝试多次计算modcount来确定，这几次尝试中，是否有其他线程进行了修改操作，如果没有，则直接返回size。如果有，则需要依次锁住所有的Segment来计算。

jdk7中ConcurrentHashMap中，当长度过长碰撞会很频繁，链表的增改删查操作都会消耗很长的时间，影响性能,所以jdk8 中完全重写了concurrentHashMap,代码量从原来的1000多行变成了 6000多行，实现上也和原来的分段式存储有很大的区别。

主要设计上的变化有以下几点：

1. 不采用segment而采用node，锁住node来实现减小锁粒度。
2. 设计了MOVED状态 当resize的过程中 线程2还在put数据，线程2会帮助resize。