Java 8 中 ConcurrentHashMap工作原理的要点分析

nullzx 2018-03-22 原文

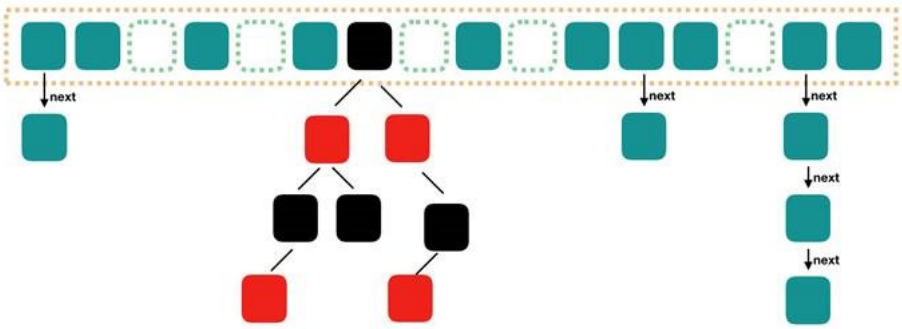
简介：

本文主要介绍Java8中的并发容器ConcurrentHashMap的工作原理，和其它文章不同的是，本文重点分析了对不同线程的各类并发操作如get,put,remove之间是如何同步的，以及这些操作和扩容操作之间同步可能出现的各种情况。由于源代码的分析肯定会有所纰漏，希望大家积极指出错误。

欢迎探讨，如有错误敬请指正

如需转载，请注明出处 http://www.cnblogs.com/nullzx/

1.Java8中 ConcurrentHashMap的结构



图片来源（ http://www.importnew.com/28263.html ）

我们将数组称之为表，将数组中每个链表或红黑树称之为桶，将**数组中的**每个结点称之为槽，也就是说“槽”存储了链表的头结点或者红黑树的根结点。源代码中用内部类Node表示链表中的每个结点。

```
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        volatile V val;
        volatile Node<K,V> next;

        //省略其它代码

        Node<K,V> find(int h, Object k) {
            Node<K,V> e = this;
            if (k != null) {
                do {
```

```
                    K ek;
                    if (e.hash == h &&
                        ((ek = e.key) == k || (ek != null && k.equals(ek))))
                        return e;
                } while ((e = e.next) != null);
            }
            return null;
        }
    }
```

每个结点都有一个hash属性它表示了Node对象的哈希值，这个哈希值实际上是key.hashCode()经过spread函数进一步散列后的值（后面的内容有spread函数源代码）。特别需要注意的是val和next属性都是用volatile修饰的。

而有关红黑树的内容不在本文的讨论范围之内，有兴趣的同学可以参考我的另外三篇有关红黑树的技术博客。

从2-3-4树到红黑树（上）

从2-3-4树到红黑树（中）

从2-3-4树到红黑树（下）

2. 表初始化长度与负载因子的含义

构造函数

```
public ConcurrentHashMap(int initialCapacity, float loadFactor)
```

2.1 表的长度

实际上表的长度必须为2的整数次幂。该类内部会用大于等于initialCapacity的最小2的整数次幂作为长度。假设你构造ConcurrentHashMap对象时传递的initialCapacity的值是21，那么实际上表的长度是32。一般教科书上设计哈希表时，会将表的长度设置为较大的质数，而这里将表的长度设置成2的整数次幂，我认为有以下两点原因：

1）在教科书中我们是通过

（Node对象的hash属性值）%表长度

来定位槽的位置。这样做的前提是我们假设求余运算是很快就可以完成的，但实际上CPU可能需要很多条指令才能实现求余操作。如果槽的长度正好的2的整数次幂，那么我们就可以通过下面的方式计算槽的位置，这和上面的计算方式等价，但位与运算明显要快于求余运算。

（Node对象的hash属性值）&（表长度-1）

2）在多线程扩容的时，这样的长度设置可以避免在扩容时对新表加锁，从而加快ConcurrentHashMap的扩容速度。关于扩容的细节问题，后面会进行讲述。

2.2 负载因子的含义

默认负载因子为0.75。我们假设表的长度为100（当然，实际上不可能是这个值，这里只是为了方便分析）。那么我们最多存储75个结点就要扩容（注意并不是占用75个槽以后才会扩容）。所以负载因子是对查询效率和存储空间平衡关系的表示。

3.减少Key的冲突

```
    static final int HASH_BITS = 0x7fffffff;
    static final int spread(int h) {
        return (h ^ (h >>> 16)) & HASH_BITS;
    }
```

通过key确定槽的位置时，如果我们直接使用

key.hashCode() &（表长度-1）

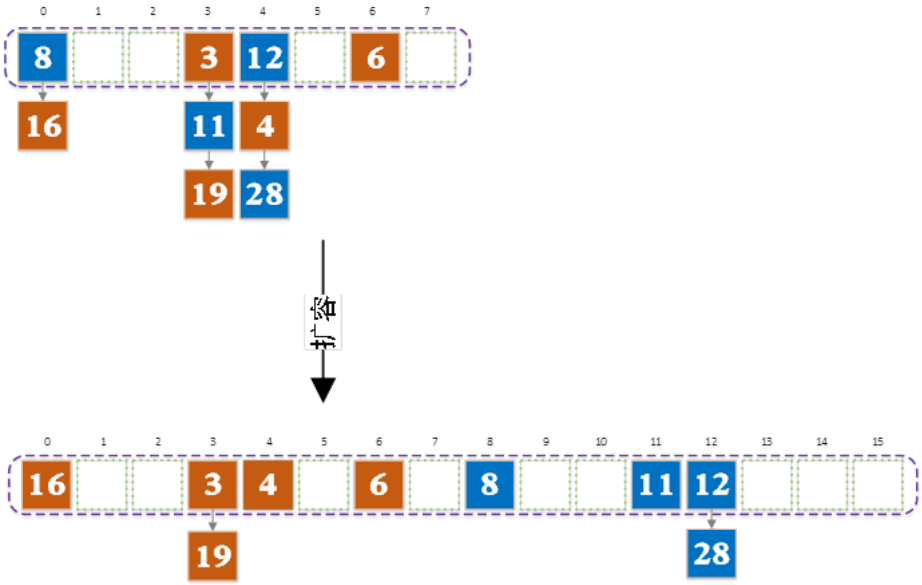
那么我们实际上只使用了key.hashCode()的低若干位信息，高位不起作用。所以为了key更加的分散，减少冲突，在实际定位槽的位置时，我们会将key.hashCode()再进行spread一下，充分使用key.hashCode()的高16位信息。而spread后的哈希值会存储在结点的hash属性中。

如果通过上述方法，仍然还存在较多的key冲突,那么就会导致同一个槽中聚集了较多结点，Java8中就会将这个长的链表转化为一颗以key表示大小的红黑树，以减少查询时间。默认情况下链表长度大于8就会被转化成红黑树。

4. 扩容操作

这里我们先不考虑并发问题，先说说基本的扩容操作，当put操作完成后，都要统计当前ConcurrentHashMap中结点的个数（显然结点个数不是一个准确值，只能是一个估计值）。如果结点个数大于设定的阀值（表的长度*负载因子），就要进行扩容操作，以提高查询效率。

前面我们说过表的长度是2的整数次幂，扩容时我们让表的长度翻倍，所以扩容后的新表长度也必然是2的整数次幂。我们这里假设旧表的长度是8（实际上代码中表的最小长度也是16，这样假设是为了画图方便），图中的数字表示结点的hash值。



从图中我们可以看出，扩容后表的长度变成了16。我们现在要对比观察扩容前后每个结点的位置，显然可以得到一个有意思的结论：**每个结点在扩容后要么留在了新表原来的位置上，要么去了新表“原位置+8”的位置上，而8就是旧表的长度。**比如扩容前3号槽有[3，11,19]结点，扩容后[3,19]结点依然留在了原3号位置，而节点[11]去了“原位置3 + 8 = 11”的位置。计算新表中槽的位置有很巧妙的方法，有兴趣的同学可以参照transfer函数的源代码。

扩容长度翻倍并，且扩容后长度仍然是2的整数次幂的特性在多线程扩容有很大的优势。**原表中不同桶上的结点，在新表上一定不会分配到相同位置的槽上。**我们可以让不同线程负责原表不同位置的桶中所有结点的迁移，这样两个线程的迁移操作是不会相互干扰的。

比如我们可以让一个线程负责原表中3号桶中所有结点的迁移，另一个线程负责原表中4号桶所有结点的迁移。原表中3号位置上的结点只能迁移到新表3号位置或11号位置上，绝对不会映射到其它位置上。而4号位置上的结点只能迁移到新表4号位置或12号位置上，所以在迁移结点的过程中，两个线程就不必在新表的对应槽上加锁了。

5. 几个重要方法的源代码分析

5.1 get方法

```
1.      public V get(Object key) {
2.          Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
3.          int h = spread(key.hashCode());
4.          if ((tab = table) != null && (n = tab.length) > 0 &&
5.              (e = tabAt(tab, (n - 1) & h)) != null) {
6.              if ((eh = e.hash) == h) {
7.                  if ((ek = e.key) == key || (ek != null && key.equals(ek)))
8.                      return e.val;
9.              }
10.             //说明这个槽迁移已完成 或者 说明槽中是红黑树的根
11.             else if (eh < 0)
12.                 return (p = e.find(h, key)) != null ? p.val : null;
13.             while ((e = e.next) != null) {
```

```
                if (e.hash == h &&
                    ((ek = e.key) == key || (ek != null && key.equals(ek))))
                    return e.val;
            }
        }
        return null;
    }
```

通过源代码发现，整个get操作都没有加锁，也没有用 CAS操作，那么get方法是怎么保证线程安全的呢？现在先不回答这个问题，不过我们应该注意get方法中头结点hash值小于0的情况（即eh < 0）的情况，结合后面的扩容操作进行解释。

5.2 put方法

```
    public V put(K key, V value) {
        return putVal(key, value, false);
    }
```

put方法实际上调用了putVal方法

```
    final V putVal(K key, V value, boolean onlyIfAbsent) {
        if (key == null || value == null) throw new NullPointerException();
        int hash = spread(key.hashCode());
        int binCount = 0;
        for (Node<K,V>[] tab = table;;) {
            Node<K,V> f; int n, i, fh;
            if (tab == null || (n = tab.length) == 0)
                tab = initTable();
            //每次循环都会重新计算槽的位置，因为在扩容完成后会使用新表，槽的位置可能会发生改变
            else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
              //槽为空，先尝试用CAS方式进行添加结点
                if (casTabAt(tab, i, null,
                             new Node<K,V>(hash, key, value, null)))
                    break;                   // no lock when adding to empty bin
            }
            //当前线程先帮助迁移,迁移完成后在新表中进行put
            else if ((fh = f.hash) == MOVED)
                tab = helpTransfer(tab, f);
            else {
                V oldVal = null;
              //加锁操作，防止其它线程对此桶同时进行put,remove,transfer操作
                synchronized (f) {
```

```
                    //头结点发生改变，就说明当前链表（或红黑树）的头节点已不是f了
                    //可能被前面的线程remove掉了或者，需要重新对链表新的头节点加锁
```

```
                    //或者头节点变成了ForwardingNode类型的结点，说明所有结点已迁移到新表上了，满足fh ==
MOVED
```

```
                    if (tabAt(tab, i) == f) {
                        //ForwordingNode的hash值为-1
                        //链表结点的hash值 >= 0
                        if (fh >= 0) {
                            binCount = 1;
                            for (Node<K,V> e = f;; ++binCount) {
                                K ek;
                                if (e.hash == hash &&
                                    ((ek = e.key) == key ||
                                     (ek != null && key.equals(ek)))) {
                                    oldVal = e.val;
                                    if (!onlyIfAbsent)
                                        e.val = value;
                                    break;
                                }
                                Node<K,V> pred = e;
                                //如果遍历到了最后一个结点，
                                //那么就说明需要插入新的结点,就把新结点插入在链表尾部
                                if ((e = e.next) == null) {
                                    pred.next = new Node<K,V>(hash, key,
                                                              value, null);
                                    break;
```

```
                                }
                            }
                        }
                        //红黑树的根结点的hash值为-2
                        else if (f instanceof TreeBin) {
                            Node<K,V> p;
                            binCount = 2;
                            if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                                  value)) != null) {
                                oldVal = p.val;
                                if (!onlyIfAbsent)
                                    p.val = value;
                            }
                        }
                    }
                }
                if (binCount != 0) {
                    if (binCount >= TREEIFY_THRESHOLD)
                        treeifyBin(tab, i);
                    if (oldVal != null)
                        return oldVal;
                    break;
                }
            }
        }
        //内部判断是否需要扩容
        addCount(1L, binCount);
        return null;
    }
```

put方法做了以下几点事情：

1）如果没有初始化就先调用initTable()方法来进行初始化过程2）如果没有hash冲突就尝试CAS方式插入

3）如果还在进行扩容操作就先帮助其它线程进一起行扩容

4）如果存在hash冲突，就加锁来保证put操作的线程安全。

有意思的是，ConcurrentHashMap中并没有使用ReentrantLock，而是直接使用了synchronized关键字对槽加锁。个人猜测，这样做的原因是避免创建过多的锁对象。如果桶的长度是1024（别问我为啥是这个值，我只是考虑到了它是2的整数次幂，如果你联想到了其它不宜公开讨论的内容，请告诉我地址），那么我们就需要在每个桶的位置上分配一把锁，也就要1024把锁，考虑到每次扩容后都还要重新创建所有的锁对象，这显然是不划算的。

添加结点操作完成后会调用addCount方法，在这个方法中会去判断是否需要扩容操作。如果容量超过阀值了，就由这个线程发起扩容操作。如果已经处于扩容状态（sizeCtl < -1），根据剩余迁移的数据和已参加到扩容中的线程数来判断是否需要当前线程来帮助扩容。

5.3 remove方法

```
    public V remove(Object key) {
        return replaceNode(key, null, null);
    }
```

实际上调用了replaceNode方法

```
    final V replaceNode(Object key, V value, Object cv) {
        int hash = spread(key.hashCode());
        for (Node<K,V>[] tab = table;;) {
            Node<K,V> f; int n, i, fh;
            if (tab == null || (n = tab.length) == 0 ||
                /*每次循环都会重新计算槽的位置，因为在扩容完成后会使用新表，槽的位置可能会发生改变*/
                (f = tabAt(tab, i = (n - 1) & hash)) == null)
                break;
            //如果有线程正在扩容，先帮助它一起扩容，然后在新表中进行put操作
            else if ((fh = f.hash) == MOVED)
                tab = helpTransfer(tab, f);
            else {
                V oldVal = null;
                boolean validated = false;
                //加锁操作，防止其它线程对此桶同时进行put,remove,transfer操作
```

```
                synchronized (f) {
                  //头结点发生改变，就说明当前链表（或红黑树）的头节点已不是f了
                    //可能被前面的线程remove掉了或者，需要重新对链表新的头节点加锁
```

```
                  //或者头节点变成了ForwardingNode类型的结点，说明所有结点已迁移到新表上了，满足fh ==
MOVED
                    if (tabAt(tab, i) == f) {
                        if (fh >= 0) {
                            validated = true;
                            for (Node<K,V> e = f, pred = null;;) {
                                K ek;
                                if (e.hash == hash &&
                                    ((ek = e.key) == key ||
                                     (ek != null && key.equals(ek)))) {
                                    V ev = e.val;
                                    if (cv == null || cv == ev ||
                                        (ev != null && cv.equals(ev))) {
                                        oldVal = ev;
                                        if (value != null)
                                            e.val = value;
                                        else if (pred != null)
                                            pred.next = e.next;
                                        else
                                            setTabAt(tab, i, e.next);
                                    }
                                    break;
                                }
                                pred = e;
                                if ((e = e.next) == null)
                                    break;
                            }
                        }
                        else if (f instanceof TreeBin) {
                            validated = true;
                            TreeBin<K,V> t = (TreeBin<K,V>)f;
                            TreeNode<K,V> r, p;
                            if ((r = t.root) != null &&
                                (p = r.findTreeNode(hash, key, null)) != null) {
                                V pv = p.val;
                                if (cv == null || cv == pv ||
                                    (pv != null && cv.equals(pv))) {
                                    oldVal = pv;
                                    if (value != null)
                                        p.val = value;
                                    else if (t.removeTreeNode(p))
                                        setTabAt(tab, i, untreeify(t.first));
                                }
                            }
                        }
                    }
                }
                if (validated) {
                    if (oldVal != null) {
                        if (value == null)
                            addCount(-1L, -1);
                        return oldVal;
                    }
                    break;
                }
            }
        }
        return null;
    }
```

5.4 ForwardingNode类

```
    static final class ForwardingNode<K,V> extends Node<K,V> {
        //新表的引用
        final Node<K,V>[] nextTable;
        ForwardingNode(Node<K,V>[] tab) {
            super(MOVED, null, null, null);
            this.nextTable = tab;
        }

        //进行get操作的线程若发现槽中的节点为ForwordingNode类型
```

```
        //说明槽中节点迁移已完成，会调用ForwordingNode的find方法在新表中进行查找
        Node<K,V> find(int h, Object k) {
            // loop to avoid arbitrarily deep recursion on forwarding nodes
           //从新表中查询
            outer: for (Node<K,V>[] tab = nextTable;;) {
              //n表示新表的长度
                Node<K,V> e; int n;
                if (k == null || tab == null || (n = tab.length) == 0 ||
                    //重新在新表中定位
                    (e = tabAt(tab, (n - 1) & h)) == null)
                    return null;
                for (;;) {
                    int eh; K ek;
                    if ((eh = e.hash) == h &&
                        ((ek = e.key) == k || (ek != null && k.equals(ek))))
                        return e;
                  //继续递归查询？这里没看懂
                    if (eh < 0) {
                        if (e instanceof ForwardingNode) {
                            tab = ((ForwardingNode<K,V>)e).nextTable;
                            continue outer;
                        }
                        else
                            return e.find(h, k);
                    }
                    //下一个
                    if ((e = e.next) == null)
                        return null;
                }
            }
        }
    }
```

ForwardingNode类继承了Node类，所以ForwardingNode对象也是Node类型对象，所以它也可以放到表中。

ForwardingNode在扩容中使用。每一个ForwardingNode对象都包含扩容后的表的引用（nextTable）。 ForwardingNode对象的key，value，next属性值全部为null，它的hash值为-1（注意小于0哦，可以去看看get方法中对应的部分了）。

ForwardingNode对象中**也**定义了find的方法,它是从扩容后的新表中查询结点，而不是以自身为头结点进行查找。

5.5 扩容方法

```
    private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
        int n = tab.length, stride;
        if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
            stride = MIN_TRANSFER_STRIDE; // subdivide range
        if (nextTab == null) {            // initiating
            try {
                @SuppressWarnings("unchecked")
                Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
                nextTab = nt;
            } catch (Throwable ex) {      // try to cope with OOME
                sizeCtl = Integer.MAX_VALUE;
                return;
            }
            nextTable = nextTab;
            transferIndex = n;
        }
        int nextn = nextTab.length;
        ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
        boolean advance = true;
        boolean finishing = false; // to ensure sweep before committing nextTab
        for (int i = 0, bound = 0;;) {
            Node<K,V> f; int fh;
            while (advance) {
                int nextIndex, nextBound;
                if (--i >= bound || finishing)
                    advance = false;
                else if ((nextIndex = transferIndex) <= 0) {
                    i = -1;
                    advance = false;
                }
                else if (U.compareAndSwapInt
                         (this, TRANSFERINDEX, nextIndex,
```

```
                    nextBound = (nextIndex > stride ?
                                 nextIndex - stride : 0))) {
                bound = nextBound;
                i = nextIndex - 1;
                advance = false;
            }
        }
        if (i < 0 || i >= n || i + n >= nextn) {
            int sc;
            if (finishing) {
                nextTable = null;
                table = nextTab;
                sizeCtl = (n << 1) - (n >>> 1);
                return;
            }
            if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
                if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                    return;
                finishing = advance = true;
                i = n; // recheck before commit
            }
        }
        else if ((f = tabAt(tab, i)) == null)
            advance = casTabAt(tab, i, null, fwd);
        else if ((fh = f.hash) == MOVED)
            advance = true; // already processed
        else {
          //头结点加锁，防止其它线程此时对该桶进行put和remove操作
            synchronized (f) {
              //和put及remove操作判断头结点是否改变的原理类似
                if (tabAt(tab, i) == f) {
                    // fh >= 0 表示链表
                    Node<K,V> ln, hn;
                    if (fh >= 0) {
                        int runBit = fh & n;
                        Node<K,V> lastRun = f;
                        for (Node<K,V> p = f.next; p != null; p = p.next) {
                            int b = p.hash & n;
                            if (b != runBit) {
                                runBit = b;
                                lastRun = p;
                            }
                        }
                        if (runBit == 0) {
                            ln = lastRun;
                            hn = null;
                        }
                        else {
                            hn = lastRun;
                            ln = null;
                        }
                       //按新表中槽的位置分为两部分
                         //注意新表中的节点都是新建的，而不是修改原的结点的next指针
                         //这样做是为了同其它线程的get方法并发时能get正确的结果
                        for (Node<K,V> p = f; p != lastRun; p = p.next) {
                            int ph = p.hash; K pk = p.key; V pv = p.val;
                            if ((ph & n) == 0)
                                ln = new Node<K,V>(ph, pk, pv, ln);
                            else
                                hn = new Node<K,V>(ph, pk, pv, hn);
                        }
                        setTabAt(nextTab, i, ln);
                        setTabAt(nextTab, i + n, hn);
                        //将头结点设置为fwd
                        setTabAt(tab, i, fwd);
                        advance = true;
                    }
                  //表示红黑树
                    else if (f instanceof TreeBin) {
                        TreeBin<K,V> t = (TreeBin<K,V>)f;
                        TreeNode<K,V> lo = null, loTail = null;
                        TreeNode<K,V> hi = null, hiTail = null;
                        int lc = 0, hc = 0;
                        for (Node<K,V> e = t.first; e != null; e = e.next) {
                            int h = e.hash;
                            TreeNode<K,V> p = new TreeNode<K,V>
                                (h, e.key, e.val, null, null);
                            if ((h & n) == 0) {
```

```
                                    if ((p.prev = loTail) == null)
                                        lo = p;
                                    else
                                        loTail.next = p;
                                    loTail = p;
                                    ++lc;
                                }
                                else {
                                    if ((p.prev = hiTail) == null)
                                        hi = p;
                                    else
                                        hiTail.next = p;
                                    hiTail = p;
                                    ++hc;
                                }
                            }
                            ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
                                (hc != 0) ? new TreeBin<K,V>(lo) : t;
                            hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
                                (lc != 0) ? new TreeBin<K,V>(hi) : t;
                            setTabAt(nextTab, i, ln);
                            setTabAt(nextTab, i + n, hn);
                            //将头结点设置为fwd
                            setTabAt(tab, i, fwd);
                            advance = true;
                        }
                    }
                }
            }
        }
    }
```

整个扩容操作分为两个部分

1）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。

2）是将原来table中的结点迁移到nextTable中，这里允许多线程进行操作。

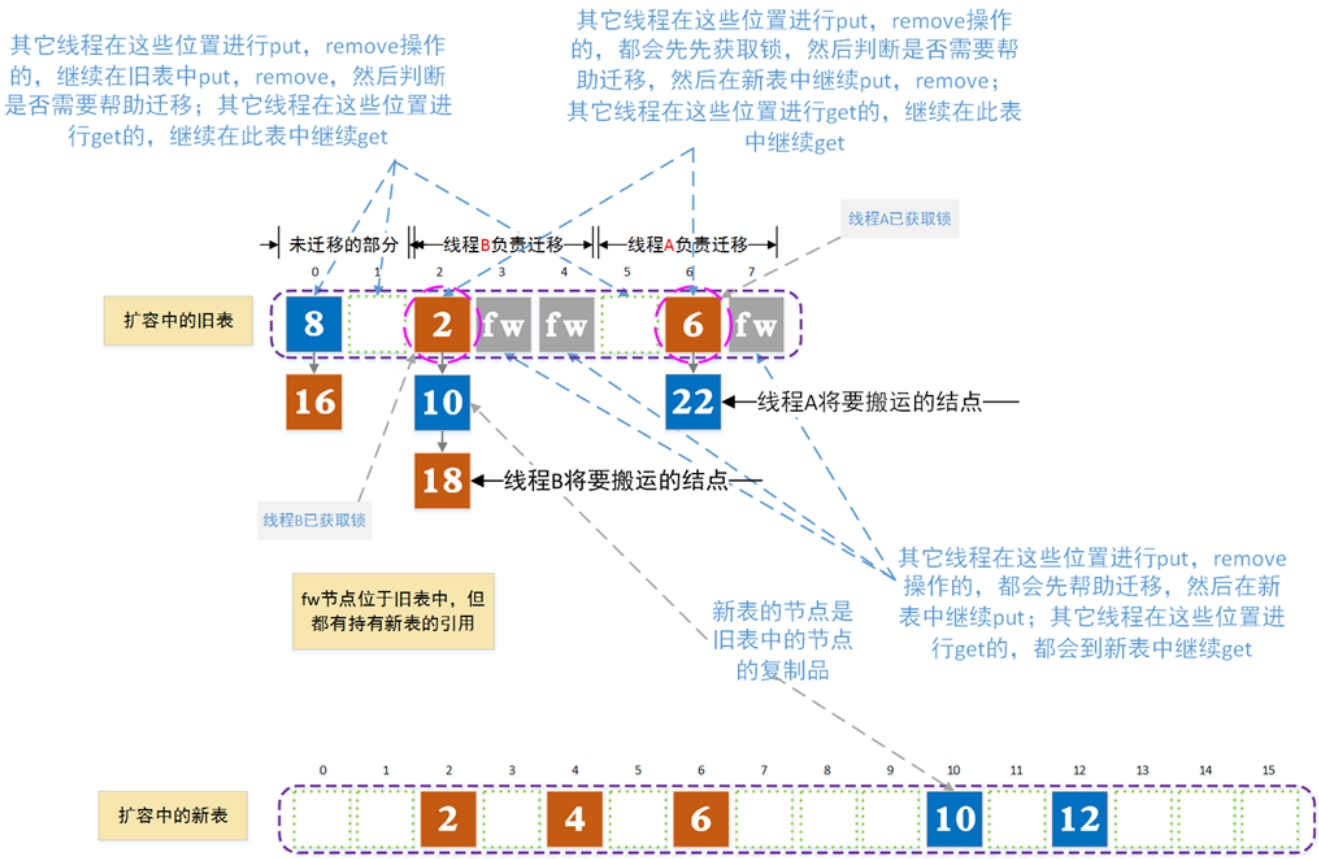
在每个位置扩容时，会对头结点加锁，避免其它线程在该位置进行put及remove操作，这个位置扩容结束时会将头结点设置成ForwardingNode，然后释放锁。ForwardingNode结点中包含新表的引用，ForwardingNode结点的hash属性的值为-1，next属性的值为null。原表中头结点为null时同样被设置成ForwardingNode结点。

多线程迁移的过程不是一个线程处理一个位置，而是一个线程处理多个连续的多个位置。在ConcurrentHashMap类中还定义下面属性值，开始扩容时这个值表示了旧表的长度，也就是说搬运工作是从旧表的末尾开始的。

```
1.  private transient volatile int transferIndex;
```

transfer函数中定义一个局部变量**stride**，它表示了每个线程的一次迁移处理的桶的个数，当一个线程处理完成后 transferIndex就自减一个stride，那么下一个线程就应该从transerIndex – stride处开始，往前处理stride个桶，以此类推完成协作。为什么要设计成从旧表的后部开始往头部的方向搬运呢？个人猜想是搬运结束的时候条件是统一的，只是写代码技巧吧。当然怎么确定整个旧表上的内容全部都迁移了，还需要读更多的源代码，这里就不作分析了。

6.并发问题的分析



上图表示了扩容操作过程中旧表和新表之间的一种可能的状态,在图中fw表示ForwordingNode类型结点，数字表示Node类型结点（上图中的扩容过程和前面论述过的“4.扩容操作”章节中的的扩容过程不是同一个过程）。现在我们就通过以下几种情况解释上图所表达的意思。首先，多个线程在同一个位置上的get操作时显然不需要同步，所以这种情况不需要讨论，我们来讨论剩下几种情况。

6.1初化的同步问题

表长度的分配并不是在构造函数中进行的，而是在put方法中进行的，也就是说这实际上是个懒汉模式。但是如果多个线程同时进行表长度的空间分配，显然是非线程安全的。所以只能有一个线程来进行创建表，其它线程会等待创建完成。ConcurrentHashMap类中设定一个volatile变量sizeCtl

```
1.  private transient volatile int sizeCtl;
```

然后通过CAS方法去修改它，如果有其它线程发现sieCtl为-1

```
1.  U.compareAndSwapInt(this, SIZECTL, sc, -1)
```

就表示已经有线程正在创建表了，那么当前线程就会放弃CPU使用权（调用Thread.yield()方法），等待分初始化完成后继续进行put操作。否则当前线程尝试将siezeCtl修改为-1,若成功，就由当前线程来创建表。

6.2 put方法和remove方法之间的同步问题

在表的同一个槽上，一个线程调用put方法和另一个线程调用put方法是互斥的；在表的同一个槽上，一个线程调用remove方法和另一个线程调用remove方法也是互斥的；在表的同一个槽上，一个线程调用remove方法和另一个线程调用put方法也是互斥的。这些互斥操作在代码中都是通过锁来保证的。

6.3 put(或remove)方法和get方法的同步问题

实际上是不需要同步，先到先得。这主要由于Node定义中value和next都定义成了volatile类型。一个线程能否get到另一个线程刚刚put（或remove）的值，这主要由两个线程当前访问的结点所处的位置决定的。

6.4 get方法和扩容操作的同步问题

可以分成两种情况讨论

1）该位置的头结点是Node类型对象，直接get，即使这个桶正在进行迁移，在get方法未完成前，迁移完已成（槽被设置成了ForwordingNode对象），也没关系，并不影响get的结果，因为get线程仍然持有旧链表的引用，可以从当前结点位置访问到所有的后续结点，原因是新表中的节点是通过复制旧表中的结点得到的，所以新表的结点的next不会影响旧表中对应结点的next值。当get方法结束后，旧链表就不可达了，会被垃圾回收线程回收。

2）该位置的头结点是ForwordingNode类型对象（头结点的hash值 == -1），头结点是ForwordingNode类型的对象，调用该对象的find方法，在新表中查找。

所以无论哪种情况，都能get到正确的值。

6.5 put(或remove)方法和扩容操作的同步问题

同样可以分为两种情况讨论：

1）该位置的头结点是Node类型对象，put操作就走正常路线，先将Node对象放入到旧表中，然后调用addCount方法，判断是否需要帮助扩容。

2）该位置的头结点是ForwordingNode类型对象，那就会先帮助扩容，然后在新表中进行put操作。

7.参考内容

[1] Java7/8 中的 HashMap 和 ConcurrentHashMap 全解析

[2] java-并发-ConcurrentHashMap高并发机制-jdk1.8

[3] 探索jdk8之ConcurrentHashMap 的实现机制

Java 8 中 ConcurrentHashMap工作原理的要点分析的更多相关文章

1. Java8 中 ConcurrentHashMap工作原理的要点分析
   简介: 本文主要介绍Java8中的并发容器ConcurrentHashMap的工作原理,和其它文章不同的是,本文重点分析了不同线程的各类并发操作如get,put,remove之间是如何同步的,以及这些 ...
2. Java类加载器的工作原理
   Java类加载器的作用就是在运行时加载类.Java类加载器基于三个机制:委托.可见性和单一性.委托机制是指将加载一个类的请求交给父类加载 器,如果这个父类加载器不能够找到或者加载这个类,那么再加载它. ...
3. Java 详解 JVM 工作原理和流程
   Java 详解 JVM 工作原理和流程 作为一名Java使用者,掌握JVM的体系结构也是必须的.说起Java,人们首先想到的是Java编程语言,然而事实上,Java是一种技术,它由四方面组成:Java ...
4. Java 连接池的工作原理(转)
   原文:Java 连接池的工作原理 什么是连接? 连接,是我们的编程语言与数据库交互的一种方式.我们经常会听到这么一句话“数据库连接很昂贵“. 有人接受这种说法,却不知道它的真正含义.因此,下面我将解释 ...
5. 【转】Java学习---HashMap的工作原理
   [原文]https://www.toutiao.com/i6592560649652404744/ HashMap的工作原理是近年来常见的Java面试题.几乎每个Java程序员都知道HashMap,都 ...
6. 【JAVA】 Java 连接池的工作原理

什么是连接? 连接,是我们的编程语言与数据库交互的一种方式.我们经常会听到这么一句话"数据库连接很昂贵". 有人接受这种说法,却不知道它的真正含义.因此,下面我将解释它 ...

7. Java 7 和 Java 8 中的 HashMap原理解析
   HashMap 可能是面试的时候必问的题目了,面试官为什么都偏爱拿这个问应聘者?因为 HashMap 它的设计结构和原理比较有意思,它既可以考初学者对 Java 集合的了解又可以深度的发现应聘者的数据 ...
8. (转)Java 详解 JVM 工作原理和流程
   作为一名Java使用者,掌握JVM的体系结构也是必须的.说起Java,人们首先想到的是Java编程语言,然而事实上,Java是一种技术,它由四方面组成:Java编程语言.Java类文件格式.Java虚 ...
9. Spring MVC中DispatcherServlet工作原理探究
   转:http://blog.csdn.net/zhouyuqwert/article/details/6853730 下面类图将主要的类及方法抽离出来,以便查看方便,根据类的结构来说明整个请求是如何工 ...

随机推荐

1. mir [20161220]
   最近玩backmir,查询了一些资料,突然领悟到原来各个地方的boss攻击和防御都有一定的上限,而相对应的,玩家也有攻击和防御,只要玩家的攻防能对付boss的攻防,就可以无伤打boss. 小时候玩热血 ...
2. py 抓取中文网址
3. BestCoder Round #73
   这场比赛打完后可以找何神玩了orz(orz)* T1Rikka with Chess 嘿嘿嘿.输出n/2+m/2即可. 我能说我智商捉鸡想了4min吗? T2Rikka with Graph 由于N个 ...
4. sleep函数——Gevent源码分析
   gevent是一个异步I/O框架,当遇到I/O操作的时候,会自动切换任务,从而能异步地完成I/O操作 但是在测试的情况下,可以使用sleep函数来让gevent进行任务切换.示例如下: import ...
5. 【原创】java中的父进程子进程 —— 坑爹的java Runtime.getRuntime().exec
   最近有一个需求,需要用一个java进程启动多个子进程来完成并发任务.由于必须给用户完成任务的反馈,所以需要父进程记录子进程的生命周期. exec方法返回一个Process对象,在当前进程内调用该对象的 ...
6. IE8下div中2个button仅仅显示一个
   IE8下div中2个button仅仅显示一个,代码例如以下: <div id="adviceType" style="display: none;" &g ...
7. C#中DBNull.Value和Null的用法和区别
   DBNull.Value,, 是适用于向数据库的表中插入空值.而 null,是指在程序中表示空引用. 或者对象为空.就是没有实例化. row[column]的值为DBNull.Value的话,至少说明 ...
8. extjs 跨域 ajax.request
   https://www.cnblogs.com/yuzhongwusan/p/3677955.html https://stackoverflow.com/questions/25727306/req ...
9. java入门-day02
   变量和数据类型 Java是强类型语言.数据在计算之前一定要有确定的类型 基本数据类型; byte /short /int /long/(分别占1-4字节) float(4字节,精度6-7位) ...
10. 2018-11-06 Visual Studio Code插件-英汉词典初版发布
    VS插件市场地址: 英汉词典 - Visual Studio Marketplace 开源在: program-in-chinese/vscode_english_chinese_dictionary ...