目录

16 ConcurrentHashMap 源码解析和设计思路

更新时间：2019-10-01 20:53:45

> 与有肝胆人共事，从无字句处读书。
>
> ——周恩来

引导语

当我们碰到线程不安全场景下，需要使用 Map 的时候，我们第一个想到的 API 估计就是 ConcurrentHashMap，ConcurrentHashMap 内部封装了锁和各种数据结构来保证访问 Map 是线程安全的，接下来我们一一来看下，和 HashMap 相比，多了哪些数据结构，又是如何保证线程安全的。

1 类注释

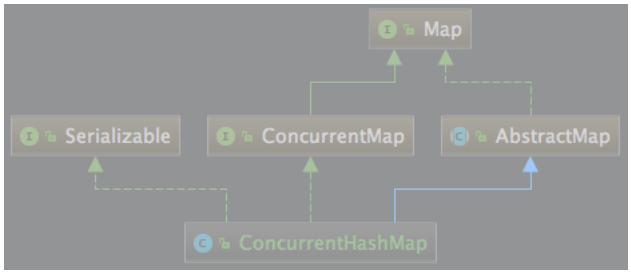
我们从类注释上大概可以得到如下信息：

1. 所有的操作都是线程安全的，我们在使用时，无需再加锁；
2. 多个线程同时进行 put、remove 等操作时并不会阻塞，可以同时进行，和 HashTable 不同，HashTable 在操作时，会锁住整个 Map；
3. 迭代过程中，即使 Map 结构被修改，也不会抛 ConcurrentModificationException 异常；
4. 除了数组 + 链表 + 红黑树的基本结构外，新增了转移节点，是为了保证扩容时的线程安全的节点；
5. 提供了很多 Stream 流式方法，比如说：forEach、search、reduce 等等。

从类注释中，我们可以看出 ConcurrentHashMap 和 HashMap 相比，新增了转移节点的数据结构，至于底层如何实现线程安全，转移节点的具体细节，暂且看不出来，接下来我们细看源码。

2 结构

目录



看 ConcurrentHashMap 源码，我们会发现很多方法和代码和 HashMap 很相似，有的同学可能会问，为什么不继承 HashMap 呢？继承的确是个好办法，但尴尬的是，ConcurrentHashMap 都是在方法中间进行一些加锁操作，也就是说加锁把方法切割了，继承就很难解决这个问题。

ConcurrentHashMap 和 HashMap 两者的相同之处：

1. 数组、链表结构几乎相同，所以底层对数据结构的操作思路是相同的（只是思路相同，底层实现不同）；
2. 都实现了 Map 接口，继承了 AbstractMap 抽象类，所以大多数的方法也都是相同的，HashMap 有的方法，ConcurrentHashMap 几乎都有，所以当我们需要从 HashMap 切换到 ConcurrentHashMap 时，无需关心两者之间的兼容问题。

不同之处：

1. 红黑树结构略有不同，HashMap 的红黑树中的节点叫做 TreeNode，TreeNode 不仅仅有属性，还维护着红黑树的结构，比如说查找，新增等等；ConcurrentHashMap 中红黑树被拆分成两块，TreeNode 仅仅维护的属性和查找功能，新增了 TreeBin，来维护红黑树结构，并负责根节点的加锁和解锁；
2. 新增 ForwardingNode （转移）节点，扩容的时候会使用到，通过使用该节点，来保证扩容时的线程安全。

3 put

ConcurrentHashMap 在 put 方法上的整体思路和 HashMap 相同，但在线程安全方面写了很多保障的代码，我们先来看下大体思路：

1. 如果数组为空，初始化，初始化完成之后，走 2；
2. 计算当前槽点有没有值，没有值的话，cas 创建，失败继续自旋（for 死循环），直到成功，槽点有值的话，走 3；
3. 如果槽点是转移节点(正在扩容)，就会一直自旋等待扩容完成之后再新增，不是转移节点走 4；
4. 槽点有值的，先锁定当前槽点，保证其余线程不能操作，如果是链表，新增值到链表的尾部，如果是红黑树，使用红黑树新增的方法新增；
5. 新增完成之后 check 需不需要扩容，需要的话去扩容。

具体源码如下：

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    //计算hash
```

目录

```
Node<K,V> f; int n, i, fh;
//table是空的，进行初始化
if (tab == null || (n = tab.length) == 0)
    tab = initTable();
//如果当前索引位置没有值，直接创建
else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
    //cas 在 i 位置创建新的元素，当 i 位置是空时，即能创建成功，结束for自循，否则继续自旋
    if (casTabAt(tab, i, null,
                 new Node<K,V>(hash, key, value, null)))
        break;                   // no lock when adding to empty bin
}
//如果当前槽点是转移节点，表示该槽点正在扩容，就会一直等待扩容完成
//转移节点的 hash 值是固定的，都是 MOVED
else if ((fh = f.hash) == MOVED)
    tab = helpTransfer(tab, f);
//槽点上有值的
else {
    V oldVal = null;
    //锁定当前槽点，其余线程不能操作，保证了安全
    synchronized (f) {
        //这里再次判断 i 索引位置的数据没有被修改
        //binCount 被赋值的话，说明走到了修改表的过程里面
        if (tabAt(tab, i) == f) {
            //链表
            if (fh >= 0) {
                binCount = 1;
                for (Node<K,V> e = f;; ++binCount) {
                    K ek;
                    //值有的话，直接返回
                    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;
                    //把新增的元素赋值到链表的最后，退出自旋
                    if ((e = e.next) == null) {
                        pred.next = new Node<K,V>(hash, key,
                                                  value, null);
                        break;
                    }
                }
            }
            //红黑树，这里没有使用 TreeNode,使用的是 TreeBin，TreeNode 只是红黑树的一个
            //TreeBin 持有红黑树的引用，并且会对其加锁，保证其操作的线程安全
            else if (f instanceof TreeBin) {
                Node<K,V> p;
                binCount = 2;
                //满足if的话，把老的值给oldVal
                //在putTreeVal方法里面，在给红黑树重新着色旋转的时候
                //会锁住红黑树的根节点
                if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                      value)) != null) {
                    oldVal = p.val;
                    if (!onlyIfAbsent)
                        p.val = value;
                }
            }
        }
```

```
                // 链表是否需要转化成红黑树
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                //这一步几乎走不到。槽点已经上锁，只有在红黑树或者链表新增失败的时候
                //才会走到这里，这两者新增都是自旋的，几乎不会失败
                break;
            }
        }
    }
    //check 容器是否需要扩容，如果需要去扩容，调用 transfer 方法去扩容
    //如果已经在扩容中了，check有无完成
    addCount(1L, binCount);
    return null;
}
```

源码中都有非常详细的注释，就不解释了，我们重点说一下，ConcurrentHashMap 在 put 过程中，采用了哪些手段来保证线程安全。

3.1 数组初始化时的线程安全

数组初始化时，首先通过自旋来保证一定可以初始化成功，然后通过 CAS 设置 SIZECTL 变量的值，来保证同一时刻只能有一个线程对数组进行初始化，CAS 成功之后，还会再次判断当前数组是否已经初始化完成，如果已经初始化完成，就不会再次初始化，通过自旋 + CAS + 双重 check 等手段保证了数组初始化时的线程安全，源码如下：

```
//初始化 table，通过对 sizeCtl 的变量赋值来保证数组只能被初始化一次
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    //通过自旋保证初始化成功
    while ((tab = table) == null || tab.length == 0) {
        // 小于 0 代表有线程正在初始化，释放当前 CPU 的调度权，重新发起锁的竞争
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        // CAS 赋值保证当前只有一个线程在初始化，-1 代表当前只有一个线程能初始化
        // 保证了数组的初始化的安全性
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                // 很有可能执行到这里的时候，table 已经不为空了，这里是双重 check
                if ((tab = table) == null || tab.length == 0) {
                    // 进行初始化
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

此的为了保证线程安全，做了四处优化：

1. 通过自旋死循环保证一定可以新增成功。

在新增之前，通过 for (Node<K,V>[] tab = table;;) 这样的死循环来保证新增一定可以成功，一旦新增成功，就可以退出当前死循环，新增失败的话，会重复新增的步骤，直到新增成功为止。

2. 当前槽点为空时，通过 CAS 新增。

Java 这里的写法非常严谨，没有在判断槽点为空的情况下直接赋值，因为在判断槽点为空和赋值的瞬间，很有可能槽点已经被其他线程赋值了，所以我们采用 CAS 算法，能够保证槽点为空的情况下赋值成功，如果恰好槽点已经被其他线程赋值，当前 CAS 操作失败，会再次执行 for 自旋，再走槽点有值的 put 流程，这里就是自旋 + CAS 的结合。

3. 当前槽点有值，锁住当前槽点。

put 时，如果当前槽点有值，就是 key 的 hash 冲突的情况，此时槽点上可能是链表或红黑树，我们通过锁住槽点，来保证同一时刻只会有一个线程能对槽点进行修改，截图如下：

4. 红黑树旋转时，锁住红黑树的根节点，保证同一时刻，当前红黑树只能被一个线程旋转，代码截图如下：

通过以上 4 点，保证了在各种情况下的新增（不考虑扩容的情况下），都是线程安全的，通过自旋 + CAS + 锁三大姿势，实现的很巧妙，值得我们借鉴。

3.3 扩容时的线程安全

ConcurrentHashMap 的扩容时机和 HashMap 相同，都是在 put 方法的最后一步检查是否需要扩容，如果需要则进行扩容，但两者扩容的过程完全不同，ConcurrentHashMap 扩容的方法叫做 transfer，从 put 方法的 addCount 方法进去，就能找到 transfer 方法，transfer 方法的主要思路是：

时，把原数组槽点赋值为转移节点；

3. 这时如果有新数据正好需要 put 到此槽点时，发现槽点为转移节点，就会一直等待，所以在扩容完成之前，该槽点对应的数据是不会发生变化的；
4. 从数组的尾部拷贝到头部，每拷贝成功一次，就把原数组中的节点设置成转移节点；
5. 直到所有数组数据都拷贝到新数组时，直接把新数组整个赋值给数组容器，拷贝完成。

关键源码如下：

```
// 扩容主要分 2 步，第一新建新的空数组，第二移动拷贝每个元素到新数组中去
// tab：原数组，nextTab：新数组
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    // 老数组的长度
    int n = tab.length, stride;
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE; // subdivide range
    // 如果新数组为空，初始化，大小为原数组的两倍，n << 1
    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;
    }
    // 新数组的长度
    int nextn = nextTab.length;
    // 代表转移节点，如果原数组上是转移节点，说明该节点正在被扩容
    ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
    boolean advance = true;
    boolean finishing = false; // to ensure sweep before committing nextTab
    // 无限自旋，i 的值会从原数组的最大值开始，慢慢递减到 0
    for (int i = 0, bound = 0;;) {
        Node<K,V> f; int fh;
        while (advance) {
            int nextIndex, nextBound;
            // 结束循环的标志
            if (--i >= bound || finishing)
                advance = false;
            // 已经拷贝完成
            else if ((nextIndex = transferIndex) <= 0) {
                i = -1;
                advance = false;
            }
            // 每次减少 i 的值
            else if (U.compareAndSwapInt
                    (this, TRANSFERINDEX, nextIndex,
                     nextBound = (nextIndex > stride ?
                                  nextIndex - stride : 0))) {
                bound = nextBound;
                i = nextIndex - 1;
                advance = false;
            }
        }
        // if 任意条件满足说明拷贝结束了
        if (i < 0 || i >= n || i + n >= nextn) {
            int sc;
```

目录

```
            // 所以此处直接赋值，没有任何问题。
            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 {
            synchronized (f) {
                // 进行节点的拷贝
                if (tabAt(tab, i) == f) {
                    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;
                        }
                        // 如果节点只有单个数据，直接拷贝，如果是链表，循环多次组成链表拷贝
                        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);
                        }
                        // 在新数组位置上放置拷贝的值
                        setTabAt(nextTab, i, ln);
                        setTabAt(nextTab, i + n, hn);
                        // 在老数组位置上放上 ForwardingNode 节点
                        // put 时，发现是 ForwardingNode 节点，就不会再动这个节点的数据了
                        setTabAt(tab, i, fwd);
                        advance = true;
                    }
                    // 红黑树的拷贝
                    else if (f instanceof TreeBin) {
                        // 红黑树的拷贝工作，同 HashMap 的内容，代码忽略
                        ……
                        // 在老数组位置上放上 ForwardingNode 节点
```

```
            }
        }
    }
}
```

扩容中的关键点，就是如何保证是线程安全的，小结有如下几点：

1. 拷贝槽点时，会把原数组的槽点锁住；
2. 拷贝成功之后，会把原数组的槽点设置成转移节点，这样如果有数据需要 put 到该节点时，发现该槽点是转移节点，会一直等待，直到扩容成功之后，才能继续 put，可以参考 put 方法中的 helpTransfer 方法；
3. 从尾到头进行拷贝，拷贝成功就把原数组的槽点设置成转移节点。
4. 等扩容拷贝都完成之后，直接把新数组的值赋值给数组容器，之前等待 put 的数据才能继续 put。

扩容方法还是很有意思的，通过在原数组上设置转移节点，put 时碰到转移节点时会等待扩容成功之后才能 put 的策略，来保证了整个扩容过程中肯定是线程安全的，因为数组的槽点一旦被设置成转移节点，在没有扩容完成之前，是无法进行操作的。

4 get

ConcurrentHashMap 读的话，就比较简单，先获取数组的下标，然后通过判断数组下标的 key 是否和我们的 key 相等，相等的话直接返回，如果下标的槽点是链表或红黑树的话，分别调用相应的查找数据的方法，整体思路和 HashMap 很像，源码如下：

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    //计算hashcode
    int h = spread(key.hashCode());
    //不是空的数组 && 并且当前索引的槽点数据不是空的
    //否则该key对应的值不存在，返回null
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        //槽点第一个值和key相等，直接返回
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        //如果是红黑树或者转移节点，使用对应的find方法
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        //如果是链表，遍历查找
        while ((e = e.next) != null) {
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

5 总结

方法感兴趣的话，可以尝试去 GitHub 上去查看源码。

15 CopyOnWriteArrayList 源码解析和设计思路

17 并发 List、Map源码面试题

精选留言 15

欢迎在这里发表留言，作者筛选后可公开显示

qq_起风了_90

老师，我在调试代码的时候，存储二个键值对，map.put("a",1); map.put("a",2);在调试源代的时候，运行到这段代码，if (binCount != 0) { if (binCount >= TREEIFY_THRESHOLD) treeifyBin(tab, i); if (oldVal != null) return oldVal; break; } 这时候oldVal的值是1，但是跳出循环后，值怎么变成2了呢？这个过程是怎么变的，看不出来啊

0 回复 5天前

宣告不幸的黑猫

老师，Jdk1.8之前的版本是怎么实现concurrentHashMap的

0 回复 2019-12-06

文贺 回复 **宣告不幸的黑猫**

同学你好，核心思想就是通过分段+加锁。

回复 7天前

鬼魅的程序涌上心头

扩容时会将扩容过后的槽点变为转移节点,那什么时候将转移节点设置成普通节点呢?是不是扩容完成将新数组拷贝成element数组的时候?

1 回复 2019-11-25

风舞炫动 回复 **鬼魅的程序涌上心头**

不会设置了，等全部拷贝完了直接给容器赋值新数组。原来的相当于等着jvm来回收了，已经没啥用了

回复 2019-11-27 18:08:08

weixin_慕工程5089940

老师为什么他已经用sync锁住了槽点了为什么还要用写锁来锁树根啊?

1 回复 2019-11-07

文贺 回复 **weixin_慕工程5089940**

因为槽点上的第一个值，不一定是红黑树的根节点。

回复 2019-11-17 11:06:31

目录

licly

node节点里面的属性为什么要加volatile呢，看网上说是为了保证table数组元素的可见性，但是数组元素赋值是在node赋值后面的，volatile只能保证它前面操作的可见性吧

0 回复 2019-11-06

文贺 回复 **licly**

个人理解，Node.next 在大量自旋操作中被使用，自旋操作相对其他操作来说，是比较耗时的，volatile 可以最大程度的保证自旋时的数据是最新的。

回复 2019-11-17 11:04:30

慕盖茨4571687

该槽点进行扩容时什么意思，不是这个map进行扩容吗?

0 回复 2019-10-28

文贺 回复 **慕盖茨4571687**

//如果当前槽点是转移节点，表示该槽点正在扩容，就会一直等待扩容完成 是这句话么，你说的对，是Map正在扩容，这句话主要想表达的是扩容正好进行到这个槽点了。

回复 2019-10-31 11:34:41

---

licly

老师，指定容量的构造方法ConcurrentHashMap(int initialCapacity)，调用tableSizeFor的时候，为什么要传initialCapacity + (initialCapacity >>> 1) + 1，而不是像hashmap一样直接传入initialCapacity

0 回复 2019-10-16

文贺 回复 **licly**

同学你好，原理应该在 《高效程序的奥秘》3.2 小节，我看了没有看懂，不敢乱说，你有兴趣的话可以看看，目的是提高性能。

回复 2019-10-17 19:37:21

---

qq_现实点_03300102

//如果当前槽点是转移节点，表示该槽点正在扩容，就会一直等待扩容完成 //转移节点的 hash 值是固定的，都是 MOVED else if ((fh = f.hash) == MOVED) tab = helpTransfer(tab, f); 老师,这个是当前槽点正在扩容,线程会帮着一起扩容吧?不是一直等待扩容完成吧?

0 回复 2019-10-15

文贺 回复 **qq_现实点_03300102**

同学你好，transfer 方法里面数据拷贝时，是加锁的，同一时刻，只能一个线程执行数据拷贝，helpTransfer 方法比较复杂，理解起来很困难，所以说等待拷贝会更加容易理解一点，但你说的一点都没错的，方法注释上面写的是帮助。

回复 2019-10-15 20:10:39

---

licly

目录

2)和 resizeStamp(n) << RESIZE_STAMP_SHIFT不相等就表示扩容没结束，相等了就表示扩容结束呢?

0 回复 2019-10-14

文贺 回复 **licly**

这个主要是因为在调用 transfer 之前，都有 U.compareAndSwapInt(this, SIZECTL, sc, (rs < < RESIZE_STAMP_SHIFT) + 2) 的操作，((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT) 正是其逆过程，如果相等，那么扩容就结束了。

回复 2019-10-15 20:53:16

weixin_慕婉清1547377

想努力看懂，但是还是看不懂，怎么办

0 回复 2019-10-11

文贺 回复 **weixin_慕婉清1547377**

理解哈，我第一次看源码的时候和你一样的困惑，源码的逻辑看着都头疼，但我会尝试的反复的看，反复的 debug，特别是 debug 时，看着运行时的数据，理解会容易很多，我相信你也可以的，静下心来，反复看，反复 debug，肯定是可以的。

回复 2019-10-12 18:58:25

licly

扩容的时候，stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE。为什么要用n >>> 8呢?

0 回复 2019-10-10

慕仰0328976 回复 **licly**

我是这么理解的：transfer主要和上一版的优化点就是支持多线程同时扩容，也就是一个线程在扩容的时候，如果有另一个线程进来了不会阻塞会帮助它一起扩容，如果理解了stride的意思的话就会明白，其实作者是先把n分成多组每组的大小是stride，然后每一个线程扩容的时候拿一组自己处理互不干涉，这种时候最好的情况就是这个电脑下所有能用的线程都来扩容效率最高，那就应该分成（NCPU * 每个CPU下的线程数）组，但是因为每个CPU下的可用线程是自己配的，作者也不知道会是多少，所以定了一个普遍的值8，因为右移比除法快所以用右移。

回复 2019-10-12 18:54:36

文贺 回复 **慕仰0328976**

点击展开剩余评论

千学不如一看，千看不如一练