Java8 ConcurrentHashMap 源码解析

作者：yangyujiao
原文链接：https://ld246.com/article/1548220865009
来源网站：链滴

最近组内技术分享，我分到讲解 ConcurrentHashMap。
结合网上看到的一些资料，整理了点东西，分享一下。
主要参考下面文章：
https://www.javadoop.com/post/hashmap#%E5%88%9D%E5%A7%8B%E5%8C%96
https://www.cnblogs.com/nullzx/p/8647220.html
<h3 id="有解释错误的地方-欢迎指摘-">有解释错误的地方，欢迎指摘。</h3>
<hr>
<h2 id="Unsafe里的CAS-操作相关">Unsafe 里的 CAS 操作相关</h2>
首先介绍的 cas，主要是因为 8 的 ConcurrentHashMap 主要用的就是 cas。
CAS(compare-and-swap 比较交换)操作。CAS 是一种低级别的、细粒度的技术,它允许多个线更新一个内存位置,同时能够检测其他线程的冲突并进行恢复。它是许多高性能并发算法的基础。 CAS 是一些 CPU 直接支持的指令，操作都封装在 java 不公开的类库中,sun.misc.Unsafe。此类包含了对子操作的封装,具体用本地代码实现。本地的 C 代码直接利用到了硬件上的原子操作，在 Java 中无锁作 CAS 基于以下 3 个方法实现。
<pre><code class="highlight-chroma">//第一个参数o为给定对象，offset为对象内存的偏移量，通过这个偏移量迅速定位字段并设置或取该字段的值，
//expected表示期值，x表示要设置的值，下面3个方法都通过CAS原子指令执行操作。
public final native boolean compareAndSwapObject(Object o, long offset,Object expected, Object x);

public final native boolean compareAndSwapInt(Object o, long offset,int expected,int x);

public final native boolean compareAndSwapLong(Object o, long offset,long expected,long x);

</code></pre>
挂起与恢复
将一个线程进行挂起是通过 park 方法实现的，调用 park 后，线程将一直阻塞直到超时或者中等条件出现。unpark 可以终止一个挂起的线程，使其恢复正常。Java 对线程的挂起操作被封装在 LockSupport 类中，LockSupport 类中有各种版本 pack 方法，其底层实现最终还是使用 Unsafe.park()方法和 Unsafe.unpark()方法
<h2 id="ConcurrentHashMap">ConcurrentHashMap</h2>
<img src="https://ld246.com/images/img-loading.svg" alt="imagepng" data-src="https://b3logfile.com/file/2019/01/image-22fa02ff.png?imageView2/2/interlace/1/format/jpg">

<h3 id="1-重要参数及初始化">1.重要参数及初始化</h3>
桶的树化阈值：即 链表转成红黑树的阈值，在存储数据时，当链表长度 > 该值时，则将链表换成红黑树
<pre><code class="highlight-chroma">static final int TREEIFY_THRESHOLD = 8;
</code></pre>
桶的链表还原阈值：即 红黑树转为链表的阈值，当在扩容（resize（））时（此时 HashMap 的据存储位置会重新计算），在重新计算存储位置后，当原有的红黑树内数量 < 6 时，则将 红黑树

换成链表
<pre><code class="highlight-chroma"><span class="highlight
cl">static final int UNTREEIFY_THRESHOLD = 6;
</code></pre>
最小树形化容量阈值：即 当哈希表中的容量 > 该值时，才允许树形化链表 （即 将链表 转换
红黑树）
否则，若桶内元素太多时，则直接扩容，而不是树形化
为了避免进行扩容、树形化选择的冲突，这个值不能小于 4 * TREEIFY_THRESHOLD
<pre><code class="highlight-chroma"><span class="highlight
cl">static final int MIN_TREEIFY_CAPACITY = 64;
</code></pre>
默认加载因子
<pre><code class="highlight-chroma"><span class="highlight
cl">private static final float LOAD_FACTOR = 0.75f;
</code></pre>
<pre><code class="highlight-chroma"><span class="highlight
cl">static final int MOVED   = -1; // hash值是-1，表示这是一个forwardNode节点
static final int TRE
BIN   = -2; // hash值是-2 表示这时一个TreeBin节点
</code></pre>
和 HashMap 中的语义一样，代表整个哈希表。
<pre><code class="highlight-chroma"><span class="highlight
cl">transient volatile Node<K,V>[] table;
</code></pre>
这是一个连接表，用于哈希表扩容，扩容完成后会被重置为 null。
<pre><code class="highlight-chroma"><span class="highlight
cl">/**
* The next table to
use; non-null only while resizing.
*/
private transient v
latile Node<K,V>[] nextTable;
</code></pre>
该属性保存着整个哈希表中存储的所有的结点的个数总和，有点类似于 HashMap 的 size 属性

<pre><code class="highlight-chroma"><span class="highlight
cl">private transient volatile long baseCount;
</code></pre>
这是一个重要的属性，无论是初始化哈希表，还是扩容 rehash 的过程，都是需要依赖这个关键
性的。该属性有以下几种取值：

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，类似于
容阈值。它的值始终是当前 ConcurrentHashMap 容量的 0.75 倍，这与 loadfactor 是对应的。实
容量 >=sizeCtl，则扩容

<pre><code class="highlight-chroma"><span class="highlight
cl">private transient volatile int sizeCtl;
</code></pre>
线程迁移 bin 的起始位置，CAS(transferIndex)成功者可迁移 transferIndex 前置 stride 个 bin
见 transfer）
<pre><code class="highlight-chroma"><span class="highlight

cl">private transient volatile int transferIndex;
</code></pre>
<h4 id="1-1-初始化">1.1 初始化</h4>
<pre><code class="highlight-chroma"><span class="highlight
cl">// 这构造函数里，什么都不干
public Concurren
HashMap() {
}

public Concurren
HashMap(int initialCapacity) {
    if (initialCapacit
 < 0)
        throw new Ill
galArgumentException();
    int cap = ((initia
Capacity >= (MAXIMUM_CAPACITY >>> 1)) ?
                MAXIM
M_CAPACITY :
                tableSize
or(initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = ca
;
}

public Concurren
HashMap(int initialCapacity, float loadFactor) {
        this(initialCa
acity, loadFactor, 1);
    }
</code></pre>
通过提供初始容量，计算了 sizeCtl，sizeCtl = 【(1.5 * initialCapacity + 1)，然后向上取最近的 2 的 n 次方】。如 initialCapacity 为 10，那么得到 sizeCtl 为 16，如果 initialCapacity 为 11，得到 sizeCtl 为 32。
<h4 id="1-2-重要的内部类">1.2 重要的内部类</h4>
<h5 id="Node">Node</h5>
<pre><code class="highlight-chroma"><span class="highlight
cl">static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        volatile V val;
// Java8增加volatile，保证可见性
        volatile Nod
<K,V> next;

        Node(inthash
 K key, V val, Node<K,V> next) {
            this.hash =
hash;
            this.key =
ey;
            this.val = v
l;
            this.next =

```
next;
        }

        public final K
getKey()       { return key; }
        public final V
getValue()     { return val; }
        // HashMap
用Objects.hashCode()，最终也是调用Object.hashCode()；效果一样
        public final in
 hashCode()   { returnkey.hashCode() ^ val.hashCode(); }
        public final St
ing toString(){ returnkey + "=" + val; }
        public final V
setValue(V value) { // 不允许修改value值，HashMap允许
            throw new
UnsupportedOperationException();
        }
        // HashMap
用if (o == this)，且嵌套if；concurrent使用&amp;&amp;
        public final b
olean equals(Object o) {
            Object k, v,
u; Map.Entry&lt;?,?&gt; e;
            return ((oi
stanceof Map.Entry) &amp;&amp;
                    (k = (
 = (Map.Entry&lt;?,?&gt;)o).getKey()) != null &amp;&amp;
                    (v = e
getValue()) != null &amp;&amp;
                    (k ==
key || k.equals(key)) &amp;&amp;
                    (v ==
u = val) || v.equals(u)));
        }

        /**
         * Virtualized
upport for map.get(); overridden in subclasses.
         */
        Node&lt;K,V
gt; find(inth, Object k) { // 增加find方法辅助get方法
            Node&lt;K
V&gt; e = this;
            if (k != null
 {
                do {
                    K ek;
                    if (e.h
sh == h &amp;&amp;
                        ((ek
= e.key) == k || (ek != null &amp;&amp; k.equals(ek))))
                        ret
rne;
```

            } while (e = e.next) != null);
        }
        return null;
    }
}
</code></pre>

这个 Node 内部类与 HashMap 中定义的 Node 类很相似，但是有一些差别
它对 value 和 next 属性设置了 volatile 同步锁
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

<h5 id="TreeNode">TreeNode</h5>
<pre><code class="highlight-chroma">// Nodes for use in TreeBins，链表>8，才可能转为TreeNode.
// HashMap的TreeNode继承至LinkedHashMap.Entry；而这里继承至自己实现的Node，将带有next指针，便于treebi访问。
    static final class TreeNode<K,V> extends Node<K,V> {
        TreeNode<K,V> parent; // red-black tree links
        TreeNode<K,V> left;
        TreeNode<K,V> right;
        TreeNode<K,V> prev;   // needed to unlink next upon deletion
        boolean red;

        TreeNode(int hash, K key, V val, Node<K,V> next,
                 TreeNode<K,V> parent) {
            super(hash key, val, next);
            this.parent = parent;
        }

        Node<K,V> find(int h, Object k) {
            return findTreeNode(h, k, null);
        }

        /**
         * Returns the TreeNode (or null if not found) for the given key
         * starting at given root.
         */ // 查找hash为h，key为k的节点
</code></pre>

```
</span></span><span class="highlight-line"><span class="highlight-cl">        final TreeNo
e&lt;K,V&gt; findTreeNode(int h, Object k, Class&lt;?&gt; kc) {
</span></span><span class="highlight-line"><span class="highlight-cl">            if (k != null
{ // 比HMap增加判空
</span></span><span class="highlight-line"><span class="highlight-cl">                TreeNo
e&lt;K,V&gt; p = this;
</span></span><span class="highlight-line"><span class="highlight-cl">                do  {
</span></span><span class="highlight-line"><span class="highlight-cl">                    intph,
dir; K pk; TreeNode&lt;K,V&gt; q;
</span></span><span class="highlight-line"><span class="highlight-cl">                    Tree
ode&lt;K,V&gt; pl = p.left, pr = p.right;
</span></span><span class="highlight-line"><span class="highlight-cl">                    if ((ph
= p.hash) &gt; h)
</span></span><span class="highlight-line"><span class="highlight-cl">                        p =
pl;
</span></span><span class="highlight-line"><span class="highlight-cl">                    elseif
ph &lt; h)
</span></span><span class="highlight-line"><span class="highlight-cl">                        p =
pr;
</span></span><span class="highlight-line"><span class="highlight-cl">                    elseif
(pk = p.key) == k || (pk != null &amp;amp;&amp;amp; k.equals(pk)))
</span></span><span class="highlight-line"><span class="highlight-cl">                        ret
rnp;
</span></span><span class="highlight-line"><span class="highlight-cl">                    elseif
pl == null)
</span></span><span class="highlight-line"><span class="highlight-cl">                        p =
pr;
</span></span><span class="highlight-line"><span class="highlight-cl">                    elseif
pr == null)
</span></span><span class="highlight-line"><span class="highlight-cl">                        p =
pl;
</span></span><span class="highlight-line"><span class="highlight-cl">                    elseif
(kc != null ||
</span></span><span class="highlight-line"><span class="highlight-cl">
kc = comparableClassFor(k)) != null) &amp;amp;&amp;amp;
</span></span><span class="highlight-line"><span class="highlight-cl">
dir = compareComparables(kc, k, pk)) != 0)
</span></span><span class="highlight-line"><span class="highlight-cl">                        p =
(dir &lt; 0) ? pl : pr;
</span></span><span class="highlight-line"><span class="highlight-cl">                    elseif
(q = pr.findTreeNode(h, k, kc)) != null)
</span></span><span class="highlight-line"><span class="highlight-cl">                        ret
rnq;
</span></span><span class="highlight-line"><span class="highlight-cl">                    else
</span></span><span class="highlight-line"><span class="highlight-cl">                        p =
pl;
</span></span><span class="highlight-line"><span class="highlight-cl">                } while (
 != null);
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">            return null;
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span></code></pre>
```

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 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 nex 指针，这样做的目的是方便基于 TreeBin 的访问
<h5 id="TreeBin">TreeBin</h5>
<pre><code class="highlight-chroma">TreeBin(TreeNode<K,V> b) {
    super(TREEBIN, null, null, null);//hash值为常量TREEBIN=-2,表示roots of trees
    this.first = b;
    TreeNode<K,> r = null;
    for (TreeNode<K,V> x = b, next; x != null; x = next) {
        next = (TreeNode<K,V>)x.next;
        x.left = x.right = null;
        if (r == null) {
            x.parent = null;
            x.red = false;
            r = x;
        }
        else {
            K k = x.key;
            int h = x.hash;
            Class<?> kc = null;
            for (TreeNode<K,V> p = r;;) {
                int dir, ph;
                K pk = p.key;
                if ((ph = p.hash) > h)
                    dir = -1;
                else if (ph < h)
                    dir = 1;
                else if ((kc == null &&
                        (kc = comparableClassFor(k)) == null) ||
                    (dir = compareComparables(kc, k, pk)) == 0)
                    dir = t
</code></pre>

```
eBreakOrder(k, pk);
                    TreeNode<K,V> xp = p;
                    if ((p = (dir <= 0) ? p.left : p.right) == null) {
                        x.parent = xp;
                        if (dir <= 0)
                            xp.left = x;
                        else
                            xp.right = x;
                        r = balanceInsertion(r, x);
                        break;
                    }
                }
            }
        }
        this.root = r;
        assert checkInvariants(root);
    }
```


TreeBin 用于封装维护 TreeNode，包含 putTreeVal、lookRoot、UNlookRoot、remove、balanceInsetion、balanceDeletion 等方法。
这里只分析其构造函数，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
当链表转树时，用于封装 TreeNode，也就是说，ConcurrentHashMap 的红黑树存放的是 TreeBin，而不是 treeNode。

<h5 id="ForwardingNode">ForwardingNode</h5>

```
// A node inserted at head of bins during transfer operations.连接两个table
// 并不是我们传统包含key-value的节点，只是一个标志节点，并且指向nextTable，提供find方法而已。生命周期：仅活于扩容操作且bin不为null时，一定会出现在每个bin的首位。
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); // 此节点hash=-1，key、value、next均为null
        this.nextTable = tab;
    }
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">    //进行get操作
线程若发现槽中的节点为ForwordingNode类型
</span></span><span class="highlight-line"><span class="highlight-cl">    //说明该桶中所
结点已迁移完成，会调用ForwordingNode的find方法在新表中进行查找
</span></span><span class="highlight-line"><span class="highlight-cl">    Node&amp;lt;K,V&amp;gt
 find(int h, Object k) {
</span></span><span class="highlight-line"><span class="highlight-cl">        // 查nextTabl
节点，outer避免深度递归
</span></span><span class="highlight-line"><span class="highlight-cl">        outer: for (N
de&amp;lt;K,V&amp;gt;[] tab = nextTable;;) {
</span></span><span class="highlight-line"><span class="highlight-cl">            // n表示新
的长度
</span></span><span class="highlight-line"><span class="highlight-cl">            Node&amp;lt;K
V&amp;gt; e; int n;
</span></span><span class="highlight-line"><span class="highlight-cl">            if (k == nul
 || tab == null || (n = tab.length) == 0 ||
</span></span><span class="highlight-line"><span class="highlight-cl">                (e = tab
t(tab, (n - 1) &amp;amp; h)) == null)
</span></span><span class="highlight-line"><span class="highlight-cl">                return n
ll;
</span></span><span class="highlight-line"><span class="highlight-cl">            for (;;) { //
AS算法多和死循环搭配！直到查到或null
</span></span><span class="highlight-line"><span class="highlight-cl">                int eh; K
ek;
</span></span><span class="highlight-line"><span class="highlight-cl">                if ((eh =
e.hash) == h &amp;amp;&amp;amp;
</span></span><span class="highlight-line"><span class="highlight-cl">                    ((ek =
e.key) == k || (ek != null &amp;amp;&amp;amp; k.equals(ek))))
</span></span><span class="highlight-line"><span class="highlight-cl">                    return
e;
</span></span><span class="highlight-line"><span class="highlight-cl">                if (eh &lt
 0) {
</span></span><span class="highlight-line"><span class="highlight-cl">                    if (e i
stanceof ForwardingNode) {
</span></span><span class="highlight-line"><span class="highlight-cl">                        tab
= ((ForwardingNode&amp;lt;K,V&amp;gt;)e).nextTable;
</span></span><span class="highlight-line"><span class="highlight-cl">                        con
inue outer;
</span></span><span class="highlight-line"><span class="highlight-cl">                    }
</span></span><span class="highlight-line"><span class="highlight-cl">                    else
</span></span><span class="highlight-line"><span class="highlight-cl">                        ret
rn e.find(h, k);
</span></span><span class="highlight-line"><span class="highlight-cl">                }
</span></span><span class="highlight-line"><span class="highlight-cl">                if ((e = e
next) == null)
</span></span><span class="highlight-line"><span class="highlight-cl">                    return
null;
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span></code></pre>
<p>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
```

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
<h4 id="1-3-三个核心方法">1.3 三个核心方法</h4>
<pre><code class="highlight-chroma"> @SuppressWarnings("unchecked")
    //获得在i位置的Node节点
    static final <K V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
        return (Nod <K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
    }
    //利用CAS算法置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
    //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
    //因为当前线程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的思是比较类似的
    static final <K V> boolean casTabAt(Node<K,V>[] tab, int i,
                        Node<K,V> c, Node<K,V> v) {
        return U.co pareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
    }
    //利用volatile法设置节点位置的值
    static final <K V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {
        U.putObject olatile(tab, ((long)i << ASHIFT) + ABASE, v);
    }

</code></pre>
<h3 id="2--put-过程分析">2. put 过程分析</h3>
<h4 id="2-1-初始化数组-initTable">2.1 初始化数组：initTable</h4>
主要就是初始化一个合适大小的数组，然后会设置 sizeCtl。
初始化方法中的并发问题是通过对 sizeCtl 进行一个 CAS 操作来控制的。
该方法的核心思想就是，只允许一个线程对表进行初始化，如果不巧有其他线程进来了，那么会其他线程交出 CPU 等待下次系统调度。这样，保证了表同时只会被一个线程初始化。
<pre><code class="highlight-chroma">private final Node<K,V>[] initTable() {
    Node<K,V&gt [] tab; int sc;
    while ((tab = ta le) == null || tab.length == 0) {
        // sizeCtl 小零说明已经有线程正在进行初始化操作
        // 当前线程该放弃 CPU 的使用
        if ((sc = sizeC l) < 0)
            Thread.yie

```
d(); // lost initialization race; just spin
            // CAS 一下将 sizeCtl 设置为 -1，代表抢到了锁
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    // sc 大于零说明容量已经初始化了，否则使用默认 DEFAULT_CAPACITY 默认初始容量是 16
                    int n = (sc &gt; 0) ? sc : DEFAULT_CAPACITY;
                    // 初化数组，长度为 16 或初始化时提供的长度
                    Node&lt;K,V&gt;[] nt = (Node&lt;K,V&gt;[])new Node&lt;?,?&gt;[n];
                    // 将个数组赋值给 table，table 是 volatile 的
                    table = tab = nt;
                    // 如 n 为 16 的话，那么这里 sc = 12
                    // 其就是 0.75 * n
                    sc = n - (n &gt;&gt;&gt; 2);
                }
            } finally {
                // 设置 sizeCtl 为 sc，我们就当是 12 吧
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

2.2 链表转红黑树: treeifyBin

treeifyBin 不一定就会进行红黑树转换，也可能是仅仅做数组扩容

```
private final void treeifyBin(Node&lt;K,V&gt;[] tab, int index) {
    Node&lt;K,V&gt; b; int n, sc;
    if (tab != null) {
        // MIN_TREEIFY_CAPACITY 为 64
        // 所以，如数组长度小于 64 的时候，其实也就是 32 或者 16 或者更小的时候，会进行数组扩容
        if ((n = tab.length) &lt; MIN_TREEIFY_CAPACITY)
            // 后面我再详细分析这个方法
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">            tryPresize(
 &amp;lt;&amp;lt; 1);
</span></span><span class="highlight-line"><span class="highlight-cl">        // b 是头结点
</span></span><span class="highlight-line"><span class="highlight-cl">        else if ((b = t
bAt(tab, index)) != null &amp;amp;&amp;amp; b.hash &amp;gt;= 0) {
</span></span><span class="highlight-line"><span class="highlight-cl">            // 加锁
</span></span><span class="highlight-line"><span class="highlight-cl">            synchroniz
d (b) {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">                if (tabAt
tab, index) == b) {
</span></span><span class="highlight-line"><span class="highlight-cl">                    // 下
就是遍历链表，建立一颗红黑树
</span></span><span class="highlight-line"><span class="highlight-cl">                    Tree
ode&amp;lt;K,V&amp;gt; hd = null, tl = null;
</span></span><span class="highlight-line"><span class="highlight-cl">                    for (
ode&amp;lt;K,V&amp;gt; e = b; e != null; e = e.next) {
</span></span><span class="highlight-line"><span class="highlight-cl">                        Tre
Node&amp;lt;K,V&amp;gt; p =
</span></span><span class="highlight-line"><span class="highlight-cl">
ew TreeNode&amp;lt;K,V&amp;gt;(e.hash, e.key, e.val,
</span></span><span class="highlight-line"><span class="highlight-cl">
            null, null);
</span></span><span class="highlight-line"><span class="highlight-cl">                        if (
p.prev = tl) == null)
</span></span><span class="highlight-line"><span class="highlight-cl">
d = p;
</span></span><span class="highlight-line"><span class="highlight-cl">                        els

</span></span><span class="highlight-line"><span class="highlight-cl">                            tl
next = p;
</span></span><span class="highlight-line"><span class="highlight-cl">                        tl =
p;
</span></span><span class="highlight-line"><span class="highlight-cl">                    }
</span></span><span class="highlight-line"><span class="highlight-cl">                    // 将
黑树设置到数组相应位置中
</span></span><span class="highlight-line"><span class="highlight-cl">                    setTa
At(tab, index, new TreeBin&amp;lt;K,V&amp;gt;(hd));
</span></span><span class="highlight-line"><span class="highlight-cl">                }
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h4 id="2-3-扩容-tryPresize">2.3 扩容：tryPresize</h4>
<p>扩容也是做翻倍扩容的，扩容后数组容量为原来的 2 倍</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">// 首先要说明的是，方法参数 size 传进来的时候就已经翻了倍了
</span></span><span class="highlight-line"><span class="highlight-cl">private final void t
yPresize(int size) {
</span></span><span class="highlight-line"><span class="highlight-cl">    // c：size 的 1.5
倍，再加 1，再往上取最近的 2 的 n 次方。
</span></span><span class="highlight-line"><span class="highlight-cl">    int c = (size &gt
= (MAXIMUM_CAPACITY &gt;&gt;&gt; 1)) ? MAXIMUM_CAPACITY :
```

```
        tableSizeFor(s
ze + (size &gt;&gt;&gt; 1) + 1);
    int sc;
    while ((sc = siz
Ctl) &gt;= 0) {
        Node&lt;K,V
gt;[] tab = table; int n;

        // 这个 if 分
和之前说的初始化数组的代码基本上是一样的，在这里，我们可以不用管这块代码
        if (tab == nul
 || (n = tab.length) == 0) {
            n = (sc &gt
 c) ? sc : c;
            if (U.comp
reAndSwapInt(this, SIZECTL, sc, -1)) {
                try {
                    if (tab
e == tab) {
                        @
uppressWarnings("unchecked")
                        No
e&lt;K,V&gt;[] nt = (Node&lt;K,V&gt;[])new Node&lt;?,?&gt;[n];
                        tab
e = nt;
                        sc
 n - (n &gt;&gt;&gt; 2); // 0.75 * n
                    }
                } finally {
                    sizeCt
 = sc;
                }
            }
        }
        else if (c &lt;
 sc || n &gt;= MAXIMUM_CAPACITY)
            break;
        else if (tab =
 table) {
            int rs = res
zeStamp(n);
            if (sc &lt; 0
 {
                Node&lt
K,V&gt;[] nt;
                //RESIZE
STAMP_SHIFT=16,MAX_RESIZERS=2^15-1
                if ((sc &
t;&gt;&gt; RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc ==
rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transf
rIndex &lt;= 0)
                    break;
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">                // 2. 用
AS 将 sizeCtl 加 1，然后执行 transfer 方法
</span></span><span class="highlight-line"><span class="highlight-cl">                //    此时
nextTab 不为 null
</span></span><span class="highlight-line"><span class="highlight-cl">                if (U.co
pareAndSwapInt(this, SIZECTL, sc, sc + 1))
</span></span><span class="highlight-line"><span class="highlight-cl">                    transf
r(tab, nt);
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">            // 1. 将 siz
Ctl 设置为 (rs &lt;&lt; RESIZE_STAMP_SHIFT) + 2)
</span></span><span class="highlight-line"><span class="highlight-cl">            //   计算出
结果是一个比较大的负数
</span></span><span class="highlight-line"><span class="highlight-cl">            //  调用 tra
sfer 方法，此时 nextTab 参数为 null
</span></span><span class="highlight-line"><span class="highlight-cl">            else if (U.c
mpareAndSwapInt(this, SIZECTL, sc,
</span></span><span class="highlight-line"><span class="highlight-cl">
                    (rs &lt;&lt; RESIZE_STAMP_SHIFT) + 2))
</span></span><span class="highlight-line"><span class="highlight-cl">                transfer(
ab, null);
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h4 id="2-4-数据迁移-transfer">2.4 数据迁移：transfer</h4>
<p>我们在 putVal 方法中遍历整个 hash 表的桶结点，如果遇到 hash 值等于 MOVED，说明已经
线程正在扩容 rehash 操作，整体上还未完成，不过我们要插入的桶的位置已经完成了所有节点的迁
。</p>
<p>由于检测到当前哈希表正在扩容，于是让当前线程去协助扩容。</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">final Node&lt;K,V&gt;[] helpTransfer(Node&lt;K,V&gt;[] tab, Node&lt;K,V&gt; f) {
</span></span><span class="highlight-line"><span class="highlight-cl">    Node&lt;K,V&gt
[] nextTab; int sc;
</span></span><span class="highlight-line"><span class="highlight-cl">    if (tab != null &
mp;&amp; (f instanceof ForwardingNode) &amp;&amp;
</span></span><span class="highlight-line"><span class="highlight-cl">        (nextTab = ((
orwardingNode&lt;K,V&gt;)f).nextTable) != null) {
</span></span><span class="highlight-line"><span class="highlight-cl">        //返回一个 16
位长度的扩容校验标识
</span></span><span class="highlight-line"><span class="highlight-cl">        int rs = resiz
Stamp(tab.length);
</span></span><span class="highlight-line"><span class="highlight-cl">        while (nextT
b == nextTable &amp;&amp; table == tab &amp;&amp;
</span></span><span class="highlight-line"><span class="highlight-cl">            (sc = siz
Ctl) &lt; 0) {
</span></span><span class="highlight-line"><span class="highlight-cl">            //sizeCtl
果处于扩容状态的话
</span></span><span class="highlight-line"><span class="highlight-cl">            //前 16 位
数据校验标识，后 16 位是当前正在扩容的线程总数
</span></span><span class="highlight-line"><span class="highlight-cl">            //这里判
校验标识是否相等，如果校验符不等或者扩容操作已经完成了，直接退出循环，不用协助它们扩容了
</span></span><span class="highlight-line"><span class="highlight-cl">            // 如果 siz
Ctl 无符号右移  16 不等于 rs  （ sc前 16 位如果不等于标识符，则标识符变化了）
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">            // 或者 sizCtl == rs + 1  （扩容结束了，不再有线程进行扩容）（默认第一个线程设置 sc ==rs 左移 16 位 + ，当第一个线程结束扩容了，就会将 sc 减一。这个时候，sc 就等于 rs + 1）
</span></span><span class="highlight-line"><span class="highlight-cl">            // 或者 sizCtl == rs + 65535  （如果达到最大帮助线程的数量，即 65535）
</span></span><span class="highlight-line"><span class="highlight-cl">            // 或者转下标正在调整 （扩容结束）
</span></span><span class="highlight-line"><span class="highlight-cl">            // 结束循，返回 table
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">            if ((sc &gt;&gt;&gt; RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
</span></span><span class="highlight-line"><span class="highlight-cl">                sc == rs + MAX_RESIZERS || transferIndex &lt;= 0)
</span></span><span class="highlight-line"><span class="highlight-cl">                break;
</span></span><span class="highlight-line"><span class="highlight-cl">            //否则调用 ransfer 帮助它们进行扩容
</span></span><span class="highlight-line"><span class="highlight-cl">            //sc + 1 识增加了一个线程进行扩容
</span></span><span class="highlight-line"><span class="highlight-cl">            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
</span></span><span class="highlight-line"><span class="highlight-cl">                transfer(ab, nextTab);
</span></span><span class="highlight-line"><span class="highlight-cl">                break;
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        return nextTb;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    return table;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>阅读源码之前，先要理解并发操作的机制。原数组长度为 n，所以我们有 n 个迁移任务，让每个程每次负责一个小任务是最简单的，每做完一个任务再检测是否有其他没做完的任务，帮助迁移就可了，而 Doug Lea 使用了一个 stride，简单理解就是步长，每个线程每次负责迁移其中的一部分，如次迁移 16 个小任务。所以，我们就需要一个全局的调度者来安排哪个线程执行哪几个任务，这个就属性 transferIndex 的作用。</p>
<p>第一个发起数据迁移的线程会将 transferIndex 指向原数组最后的位置，然后从后往前的 stride 任务属于第一个线程，然后将 transferIndex 指向新的位置，再往前的 stride 个任务属于第二个线程依此类推。当然，这里说的第二个线程不是真的一定指代了第二个线程，也可以是同一个线程，这个者应该能理解吧。其实就是将一个大的迁移任务分为了一个个任务包。</p>
<p>ConcurrentHashMap 无锁多线程扩容，减少扩容时的时间消耗。<br>
transfer 扩容操作：单线程构建两倍容量的 nextTable；允许多线程复制原 table 元素到 nextTable
</p>
<ol>
<li>为每个内核均分任务，并保证其不小于 16；</li>
<li>若 nextTab 为 null，则初始化其为原 table 的 2 倍；</li>
<li>死循环遍历，直到 finishing。</li>
</ol>
<ul>
<li>节点为空，则插入 ForwardingNode；</li>
<li>链表节点（fh&gt;=0），分别插入 nextTable 的 i 和 i+n 的位置；</li>
<li>TreeBin 节点（fh&lt;0），判断是否需要 untreefi，分别插入 nextTable 的 i 和 i+n 的位置；</l>
```

finishing 时，nextTab 赋给 table，更新 sizeCtl 为新容量的 0.75 倍 ，完成扩容。

<p>这个方法的核心在于 sizeCtl 值的操作，首先将其设置为一个负数，然后执行 transfer(tab, null)
再下一个循环将 sizeCtl 加 1，并执行 transfer(tab, nt)，之后可能是继续 sizeCtl 加 1，并执行 trans
er(tab, nt)。</p>
<p>所以，可能的操作就是执行 1 次 transfer(tab, null) + 多次 transfer(tab, nt)，这里怎么结束循
的需要看完 transfer 源码才清楚。</p>
<p>以上说的都是单线程，多线程又是如何实现的呢？</p>
<p>遍历到 ForwardingNode 节点((fh = f.hash) == MOVED)，说明此节点被处理过了，直接跳过
这是控制并发扩容的核心。由于给节点上了锁，只允许当前线程完成此节点的操作，处理完毕后，将
应值设为 ForwardingNode（fwd），其他线程看到 forward，直接向后遍历。如此便完成了多线程
复制工作，也解决了线程安全问题。</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
   int n = tab.leng
h, stride;

   // stride 在单
下直接等于 n，多核模式下为 (n>>>3)/NCPU，最小值是 16
   // stride 可以
解为”步长“，有 n 个位置是需要进行迁移的，
   //   将这 n 个任
分为多个任务包，每个任务包有 stride 个任务
   if ((stride = (NC
U > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
   stride = MIN
TRANSFER_STRIDE; // subdivide range

   // 如果 nextTab
为 null，先进行一次初始化
   //   前面我们说
，外围会保证第一个发起迁移的线程调用此方法时，参数 nextTab 为 null
   //   之后参与
移的线程调用此方法时，nextTab 不会为 null
   if (nextTab ==
ull) {
   try {
   // 容量翻倍
   Node<K
V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
   nextTab =
nt;
   } catch (Thro
able ex) {   // try to cope with OOME
   sizeCtl = In
eger.MAX_VALUE;
   return;
   }
   // nextTable
是 ConcurrentHashMap 中的属性
   nextTable =
extTab;
<span class="highlight-line"><span class="highlight-cl">   // transferIn
ex 也是 ConcurrentHashMap 的属性，用于控制迁移的位置

```
                transferIndex = n;
            }

            int nextn = nextTab.length;

            // ForwardingNode 翻译过来就是正在被迁移的 Node
            // 这个构造方会生成一个Node，key、value 和 next 都为 null，关键是 hash 为 MOVED
            // 后面我们会到，原数组中位置 i 处的节点完成迁移工作后，
            //    就会将位置 i 处设置为这个 ForwardingNode，用来告诉其他线程该位置已经处理过了
            //    所以它其实当于是一个标志。
            ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);


            // advance 指是做完了一个位置的迁移工作，可以准备做下一个位置的了
            boolean advance = true;
            boolean finishing = false; // to ensure sweep before committing nextTab

            /*
             * 下面这个 for 环，最难理解的在前面，而要看懂它们，应该先看懂后面的，然后再倒回来看
             *
             */

            // i 是位置索引 bound 是边界，注意是从后往前
            for (int i = 0, bound = 0;;) {
                Node<K,V> f; int fh;

                // 下面这个 while 真的是不好理解
                // advance 为 true 表示可以进行下一个位置的迁移了
                //   简单理解局：i 指向了 transferIndex，bound 指向了 transferIndex-stride
                while (advance) {
                    int nextIndex, nextBound;
                    if (--i > bound || finishing)
                        advance
```

```
= false;
                                                                    
                    // 将 trans
erIndex 值赋给 nextIndex
                    // 这里 tra
sferIndex 一旦小于等于 0，说明原数组的所有位置都有相应的线程去处理了
                    else if ((ne
tIndex = transferIndex) &lt;= 0) {
                        i = -1;
                        advance
= false;
                    }
                    else if (U.c
mpareAndSwapInt
                            (this,
TRANSFERINDEX, nextIndex,
                             next
ound = (nextIndex &gt; stride ?
                                 
    nextIndex - stride : 0))) {
                        // 看括
中的代码，nextBound 是这次迁移任务的边界，注意，是从后往前
                        bound =
nextBound;
                        i = next
ndex - 1;
                        advance
= false;
                    }
                }
                if (i &lt; 0 || i
&gt;= n || i + n &gt;= nextn) {
                    int sc;
                    if (finishing
 {
                        // 所有
迁移操作已经完成
                        nextTab
e = null;
                        // 将新的
nextTab 赋值给 table 属性，完成迁移
                        table =
extTab;
                        // 重新
算 sizeCtl：n 是原数组长度，所以 sizeCtl 得出的值将是新数组长度的 0.75 倍
                        sizeCtl =
(n &lt;&lt; 1) - (n &gt;&gt;&gt; 1);
                        return;
                    }
                    
                    // 之前我
说过，sizeCtl 在迁移前会设置为 (rs &lt;&lt; RESIZE_STAMP_SHIFT) + 2
                    // 然后，
有一个线程参与迁移就会将 sizeCtl 加 1，
```

```
                // 这里使用 CAS 操作对 sizeCtl 进行减 1，代表做完了属于自己的任务
                if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
                    // 任务结束，方法退出
                    if ((sc - 2) != resizeStamp(n) &lt;&lt; RESIZE_STAMP_SHIFT)
                        return

                    // 到这，说明 (sc - 2) == resizeStamp(n) &lt;&lt; RESIZE_STAMP_SHIFT，
                    // 也就说，所有的迁移任务都做完了，也就会进入到上面的 if(finishing){} 分支了
                    finishing = advance = true;
                    i = n; // recheck before commit
                }
            }
            // 如果位置 i 处是空的，没有任何节点，那么放入刚刚初始化的 ForwardingNode ”空节点“
            else if ((f = tabAt(tab, i)) == null)
                advance = casTabAt(tab, i, null, fwd);
            // 该位置处一个 ForwardingNode，代表该位置已经迁移过了
            else if ((fh = f.hash) == MOVED)
                advance = true; // already processed
            else {
                // 对数组位置处的结点加锁，开始处理数组该位置处的迁移工作
                synchronized (f) {
                    if (tabAt(tab, i) == f) {
                        Node&lt;K,V&gt; ln, hn;
                        // 头点的 hash 大于 0，说明是链表的 Node 节点
                        if (fh &gt;= 0) {
                            // 面这一块和 Java7 中的 ConcurrentHashMap 迁移是差不多的，
                            // 要将链表一分为二，
                            // 找到原链表中的 lastRun，然后 lastRun 及其之后的节点是一起进行迁移的
                            // astRun 之前的节点需要进行克隆，然后分到两个链表中
```

```
                int unBit = fh &amp; n;
                Node&lt;K,V&gt; lastRun = f;
                for (Node&lt;K,V&gt; p = f.next; p != null; p = p.next) {
                    int b = p.hash &amp; n;
                    if (b != runBit) {
                        runBit = b;
                        lastRun = p;
                    }
                }
                if (unBit == 0) {
                    l = lastRun;
                    n = null;
                }
                else {
                    n = lastRun;
                    l = null;
                }
                for (Node&lt;K,V&gt; p = f; p != lastRun; p = p.next) {
                    int ph = p.hash; K pk = p.key; V pv = p.val;
                    if ((ph &amp; n) == 0)
                        ln = new Node&lt;K,V&gt;(ph, pk, pv, ln);
                    else
                        hn = new Node&lt;K,V&gt;(ph, pk, pv, hn);
                }
                //中的一个链表放在新数组的位置 i
                setAbAt(nextTab, i, ln);
                //一个链表放在新数组的位置 i+n
                setAbAt(nextTab, i + n, hn);
                //原数组该位置处设置为 fwd，代表该位置已经处理完毕，
                //
```

其他线程一旦看到该位置的 hash 值为 MOVED，就不会进行迁移了
```
                    setAbAt(tab, i, fwd);
                    //dvance 设置为 true，代表该位置已经迁移完毕
                    advance = true;
                }
                else if (f instanceof TreeBin) {
                    //黑树的迁移
                    TreeBin<K,V> t = (TreeBin<K,V>)f;
                    TreeNode<K,V> lo = null, loTail = null;
                    TreeNode<K,V> hi = null, hiTail = null;
                    int c = 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 &amp; 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;
</span></span><span class="highlight-line"><span class="highlight-cl">
 ++hc;
</span></span><span class="highlight-line"><span class="highlight-cl">                    }
</span></span><span class="highlight-line"><span class="highlight-cl">                }
</span></span><span class="highlight-line"><span class="highlight-cl">                //
果一分为二后，节点数&lt;= 6，那么将红黑树转换回链表
</span></span><span class="highlight-line"><span class="highlight-cl">                ln
 (lc &lt;= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
</span></span><span class="highlight-line"><span class="highlight-cl">                    (
c != 0) ? new TreeBin&lt;K,V&gt;(lo) : t;
</span></span><span class="highlight-line"><span class="highlight-cl">                hn
 = (hc &lt;= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
</span></span><span class="highlight-line"><span class="highlight-cl">                    (
c != 0) ? new TreeBin&lt;K,V&gt;(hi) : t;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">                //
 ln 放置在新数组的位置 i
</span></span><span class="highlight-line"><span class="highlight-cl">                set
abAt(nextTab, i, ln);
</span></span><span class="highlight-line"><span class="highlight-cl">                //
 hn 放置在新数组的位置 i+n
</span></span><span class="highlight-line"><span class="highlight-cl">                set
abAt(nextTab, i + n, hn);
</span></span><span class="highlight-line"><span class="highlight-cl">                //
原数组该位置处设置为 fwd，代表该位置已经处理完毕，
</span></span><span class="highlight-line"><span class="highlight-cl">                //
其他线程一旦看到该位置的 hash 值为 MOVED，就不会进行迁移了
</span></span><span class="highlight-line"><span class="highlight-cl">                set
abAt(tab, i, fwd);
</span></span><span class="highlight-line"><span class="highlight-cl">                //
dvance 设置为 true，代表该位置已经迁移完毕
</span></span><span class="highlight-line"><span class="highlight-cl">                ad
ance = true;
</span></span><span class="highlight-line"><span class="highlight-cl">                }
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h4 id="2-5-put方法">2.5 put 方法</h4>
<p>在多线程中可能有以下两个情况</p>
<ol>
<li>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，如检测到需要插入的位置被 forward 节点占有，就帮助进行扩容；</li>
<li>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。尽管这个有一些影响效率，但是还是会比 hashTable 的 synchronized 要好得多。</li>
</ol>
<p>整体流程</p>
<ol>
<li>校验 key value 值，都不能是 null。这点和 HashMap 不同。</li>
<li>得到 key 的 hash 值。</li>
```

死循环并更新 tab 变量的值。
如果容器没有初始化，则初始化。调用 initTable 方法。该方法通过一个变量 + CAS 来控制并发
稍后我们分析源码。
根据 hash 值找到数组下标，如果对应的位置为空，就创建一个 Node 对象用 CAS 方式添加到
器。并跳出循环。
如果 hash 冲突，也就是对应的位置不为 null，则判断该槽是否被扩容了（-1 表示被扩容了），
果被扩容了，返回新的数组。
如果 hash 冲突 且 hash 值不是 -1，表示没有被扩容。则进行链表操作或者红黑树操作，注意，
里的 f 头节点被锁住了，保证了同时只有一个线程修改链表。防止出现链表成环。
和 HashMap 一样，如果链表树超过 8，则修改链表为红黑树。
将数组加 1（CAS 方式），如果需要扩容，则调用 transfer 方法进行移动和重新散列，该方法
，如果是槽中只有单个节点，则使用 CAS 直接插入，如果不是，则使用 synchronized 进行同步，
止并发成环。

<pre><code class="highlight-chroma"><span class="highlight
cl">public V put(K key, V value) {
    return putVal(k
y, value, false);
}

final V putVal(K k
y, V value, boolean onlyIfAbsent) {
    //对传入的参数
行合法性判断 不允许 key或value为null
    if (key == null ||
value == null) throw new NullPointerException();
    // 计算键所对
的 hash 值
    int hash = spre
d(key.hashCode());
    // 用于记录相
链表的长度
    int binCount =
;
    for (Node<K,
>[] tab = table;;) {
        Node<K,V
gt; f; int n, i, fh;
        // 如果数组"
"，进行数组初始化
        if (tab == nul
|| (n = tab.length) == 0)
            // 初始化
组，后面会详细介绍
            tab = initT
ble();

        // 找该 hash
值对应的数组下标，得到第一个节点 f
        else if ((f = t
bAt(tab, i = (n - 1) & hash)) == null) {
            // 如果数
该位置为空，
            //   用一次

```
CAS 操作将这个新值放入其中即可，这个 put 操作差不多就结束了，可以拉到最后面了
</span></span><span class="highlight-line"><span class="highlight-cl">            //
果 CAS 失败，那就是有并发操作，进到下一个循环就好了
</span></span><span class="highlight-line"><span class="highlight-cl">            if (casTabA
(tab, i, null,
</span></span><span class="highlight-line"><span class="highlight-cl">                    n
w Node&amp;lt;K,V&amp;gt;(hash, key, value, null)))
</span></span><span class="highlight-line"><span class="highlight-cl">                break;
            // no lock when adding to empty bin
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        // hash 居然
以等于 MOVED(MOVED = -1；hash for forwarding nodes)，这个需要到后面才能看明白，不过从
字上也能猜到，肯定是因为在扩容
</span></span><span class="highlight-line"><span class="highlight-cl">        else if ((fh = f
hash) == MOVED)
</span></span><span class="highlight-line"><span class="highlight-cl">            // 帮助数
迁移，这个等到看完数据迁移部分的介绍后，再理解这个就很简单了
</span></span><span class="highlight-line"><span class="highlight-cl">            tab = help
ransfer(tab, f);
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        else { // 到
里就是说，f 是该位置的头结点，而且不为空
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">            V oldVal =
null;
</span></span><span class="highlight-line"><span class="highlight-cl">            // 获取数
该位置的头结点的监视器锁
</span></span><span class="highlight-line"><span class="highlight-cl">            synchroniz
d (f) {
</span></span><span class="highlight-line"><span class="highlight-cl">                if (tabAt
tab, i) == f) {
</span></span><span class="highlight-line"><span class="highlight-cl">                    if (fh
gt;= 0) { // 头结点的 hash 值大于 0，说明是链表
</span></span><span class="highlight-line"><span class="highlight-cl">                        //
于累加，记录链表的长度
</span></span><span class="highlight-line"><span class="highlight-cl">                        bin
ount = 1;
</span></span><span class="highlight-line"><span class="highlight-cl">                        //
历链表
</span></span><span class="highlight-line"><span class="highlight-cl">                        for
Node&amp;lt;K,V&amp;gt; e = f;; ++binCount) {
</span></span><span class="highlight-line"><span class="highlight-cl">                            K
ek;
</span></span><span class="highlight-line"><span class="highlight-cl">                            /
 如果发现了"相等"的 key，判断是否要进行值覆盖，然后也就可以 break 了
</span></span><span class="highlight-line"><span class="highlight-cl">                            if
(e.hash == hash &amp;amp;&amp;amp;
</span></span><span class="highlight-line"><span class="highlight-cl">
 ((ek = e.key) == key ||
</span></span><span class="highlight-line"><span class="highlight-cl">
  (ek != null &amp;amp;&amp;amp; key.equals(ek)))) {
</span></span><span class="highlight-line"><span class="highlight-cl">
 oldVal = e.val;
</span></span><span class="highlight-line"><span class="highlight-cl">
```

```
 if (!onlyIfAbsent) //仅putIfAbsent()方法中onlyIfAbsent为true
</span></span><span class="highlight-line"><span class="highlight-cl">
    e.val = value; //putIfAbsent()包含key则返回get，否则put并返回
</span></span><span class="highlight-line"><span class="highlight-cl">
 break;
</span></span><span class="highlight-line"><span class="highlight-cl">                    }
</span></span><span class="highlight-line"><span class="highlight-cl">                    /
 到了链表的最末端，将这个新值放到链表的最后面
</span></span><span class="highlight-line"><span class="highlight-cl">
ode&lt;K,V&gt; pred = e;
</span></span><span class="highlight-line"><span class="highlight-cl">                    if
((e = e.next) == null) {
</span></span><span class="highlight-line"><span class="highlight-cl">
 pred.next = new Node&lt;K,V&gt;(hash, key,
</span></span><span class="highlight-line"><span class="highlight-cl">
                        value, null);
</span></span><span class="highlight-line"><span class="highlight-cl">
 break;
</span></span><span class="highlight-line"><span class="highlight-cl">                    }
</span></span><span class="highlight-line"><span class="highlight-cl">                }
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">            else if
(f instanceof TreeBin) { // 红黑树
</span></span><span class="highlight-line"><span class="highlight-cl">                No
e&lt;K,V&gt; p;
</span></span><span class="highlight-line"><span class="highlight-cl">                bin
ount = 2;
</span></span><span class="highlight-line"><span class="highlight-cl">                //
用红黑树的插值方法插入新节点
</span></span><span class="highlight-line"><span class="highlight-cl">                if (
p = ((TreeBin&lt;K,V&gt;)f).putTreeVal(hash, key,
</span></span><span class="highlight-line"><span class="highlight-cl">
                    value)) != null) {
</span></span><span class="highlight-line"><span class="highlight-cl">                    o
dVal = p.val;
</span></span><span class="highlight-line"><span class="highlight-cl">                    if
(!onlyIfAbsent)
</span></span><span class="highlight-line"><span class="highlight-cl">
 p.val = value;
</span></span><span class="highlight-line"><span class="highlight-cl">                }
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    //binCount
!= 0 说明向链表或者红黑树中添加或修改一个节点成功
</span></span><span class="highlight-line"><span class="highlight-cl">    //binCount
 == 0 说明 put 操作将一个新节点添加成为某个桶的首节点
</span></span><span class="highlight-line"><span class="highlight-cl">    if (binCoun
 != 0) {
</span></span><span class="highlight-line"><span class="highlight-cl">        // 判断
否要将链表转换为红黑树，临界值和 HashMap 一样，也是 8
</span></span><span class="highlight-line"><span class="highlight-cl">        if (binC
unt &gt;= TREEIFY_THRESHOLD)
```

```
                // 这方法和 HashMap 中稍微有一点点不同，那就是它不是一定会进行红黑树转换，
                // 如当前数组的长度小于 64，那么会选择进行数组扩容，而不是转换为红黑树
                // 体源码我们就不看了，扩容部分后面说
                treeifBin(tab, i);
            if (oldVal != null)
                return oldVal;
            break;
        }
    }
    // CAS 式更新baseCount，并判断是否需要扩容
    addCount(1L, bnCount);
    //程序走到这一说明此次 put 操作是一个添加操作，否则早就 return 返回了
    return null;
}
```

3.get 过程分析

1. 计算 hash 值
2. 根据 hash 值找到数组对应位置: (n - 1) & h
3. 根据该位置处结点性质进行相应查找

- 3.1 如果该位置为 null，那么直接返回 null 就可以了
- 3.2 如果该位置处的节点刚好就是我们需要的，返回该节点的值即可
- 3.3 如果该位置节点的 hash 值小于 0，说明正在扩容，或者是红黑树，利用 Node 中的 find 方
- 3.4 如果以上 3 条都不满足，那就是链表，进行遍历比对即可

```
public V get(Object key) {
    Node&lt;K,V&gt;[] tab; Node&lt;K,V&gt; e, p; int n, eh; K ek;
    int h = spread(key.hashCode());
    if ((tab = table) = null &amp;&amp; (n = tab.length) &gt; 0 &amp;&amp;
        (e = tabAt(ta, (n - 1) &amp; h)) != null) {
        // 判断头结是否就是我们需要的节点
        if ((eh = e.hah) == h) {
            if ((ek = e.key) == key || (ek != null &amp;&amp; key.equals(ek)))
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">                return e.
al;
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">            // 如果头结
的 hash 小于 0，说明 正在扩容，或者该位置是红黑树
</span></span><span class="highlight-line"><span class="highlight-cl">            else if (eh &lt;
0)
</span></span><span class="highlight-line"><span class="highlight-cl">                // 参考 Fo
wardingNode.find(int h, Object k) 和 TreeBin.find(int h, Object k)
</span></span><span class="highlight-line"><span class="highlight-cl">                return (p =
e.find(h, key)) != null ? p.val : null;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">            // 遍历链表
</span></span><span class="highlight-line"><span class="highlight-cl">            while ((e = e.
ext) != null) {
</span></span><span class="highlight-line"><span class="highlight-cl">                if (e.hash
= h &amp;amp;&amp;amp;
</span></span><span class="highlight-line"><span class="highlight-cl">                    ((ek = e.
ey) == key || (ek != null &amp;amp;&amp;amp; key.equals(ek))))
</span></span><span class="highlight-line"><span class="highlight-cl">                    return e.
al;
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        return null;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h3 id="看完源码-通过一张图直观展示一下扩容操作">看完源码，通过一张图直观展示一下扩容操
</h3>
<p>看懂上面了，下面就不用看了：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="imagepng" data-src="https:
/b3logfile.com/file/2019/01/image-3063e7b6.png?imageView2/2/interlace/1/format/jpg"></
>
<p><b> 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
便），图中的数字表示结点的 hash 值。</b></p>
<p>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
，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]
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
妙的方法，有兴趣的同学可以参照 transfer 函数的源代码。</p>
<p>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
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
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。</p>
<p>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
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
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。</p>
```