ConcurrentHashMap

类的默认常量值：

```
public class ConcurrentHashMap<K,V> extends AbstractMap<K,V>
    implements ConcurrentMap<K,V>, Serializable {
    private static final long serialVersionUID = 7249069246763182397L;
    private static final int MAXIMUM_CAPACITY = 1 << 30;
    private static final int DEFAULT_CAPACITY = 16;
    static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
    /**
     * The default concurrency level for this table. Unused but
     * defined for compatibility with previous versions of this class.
     */
    private static final int DEFAULT_CONCURRENCY_LEVEL = 16;
    private static final float LOAD_FACTOR = 0.75f;
    static final int TREEIFY_THRESHOLD = 8;
    static final int UNTREEIFY_THRESHOLD = 6;
    static final int MIN_TREEIFY_CAPACITY = 64;
    /**
     * Minimum number of rebinnings per transfer step. Ranges are
     * subdivided to allow multiple resizer threads.  This value
     * serves as a lower bound to avoid resizers encountering
     * excessive memory contention.  The value should be at least
     * DEFAULT_CAPACITY.
     */
    private static final int MIN_TRANSFER_STRIDE = 16;
    /**
     * The number of bits used for generation stamp in sizeCtl.
     * Must be at least 6 for 32bit arrays.
     */
    private static int RESIZE_STAMP_BITS = 16;
    /**
     * The maximum number of threads that can help resize.
     * Must fit in 32 - RESIZE_STAMP_BITS bits.
     */
```

```
    private static final int MAX_RESIZERS = (1 << (32 - RESIZE_STAMP_BITS)) - 1;
    /**
     * The bit shift for recording size stamp in sizeCtl.
     */
    private static final int RESIZE_STAMP_SHIFT = 32 - RESIZE_STAMP_BITS;
    /*
     * Encodings for Node hash fields. See above for explanation.
     */
    static final int MOVED     = -1; // hash for forwarding nodes
    static final int TREEBIN   = -2; // hash for roots of trees
    static final int RESERVED  = -3; // hash for transient reservations
    static final int HASH_BITS = 0x7fffffff; // usable bits of normal node hash

    /** Number of CPUS, to place bounds on some sizings */
    static final int NCPU = Runtime.getRuntime().availableProcessors();
```

Node节点：不允许修改value值，HashMap允许

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

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

    public final K getKey()       { return key; }
    public final V getValue()     { return val; }
    public final int hashCode()   { return key.hashCode() ^ val.hashCode(); }
    public final String toString(){ return key + "=" + val; }
    public final V setValue(V value) {// 不允许修改value值，HashMap允许
        throw new UnsupportedOperationException();
    }

    public final boolean equals(Object o) {
        Object k, v, u; Map.Entry<?,?> e;
        return ((o instanceof Map.Entry) &&
                (k = (e = (Map.Entry<?,?>)o).getKey()) != null &&
                (v = e.getValue()) != null &&
                (k == key || k.equals(key)) &&
                (v == (u = val) || v.equals(u)));
    }
    /**
     * Virtualized support for map.get(); overridden in subclasses.
     */
    //增加find方法辅助map.get方法，HashMap中的Node类中没有此方法
    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;
    }
}
```

TreeNode:

```
// Nodes for use in TreeBins
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.
     */
    final TreeNode<K,V> findTreeNode(int h, Object k, Class<?> kc) {
        if (k != null) {//HashMap没有非空判断
            TreeNode<K,V> p = this;
            do  {
                int ph, dir; K pk; TreeNode<K,V> q;
                TreeNode<K,V> pl = p.left, pr = p.right;
                if ((ph = p.hash) > h)
                    p = pl;
                else if (ph < h)
                    p = pr;
                else if ((pk = p.key) == k || (pk != null && k.equals(pk)))
                    return p;
                else if (pl == null)
                    p = pr;
                else if (pr == null)
                    p = pl;
                else if ((kc != null ||
```

```
                         (kc = comparableClassFor(k)) != null) &&
                        (dir = compareComparables(kc, k, pk)) != 0)
                    p = (dir < 0) ? pl : pr;
                else if ((q = pr.findTreeNode(h, k, kc)) != null)
                    return q;
                else
                    p = pl;
            } while (p != null);
        }
        return null;
    }
}
```

TreeBins

TreeBin用于封装维护TreeNode，包含putTreeVal、lookRoot、UNlookRoot、remove、balanceInsetion、balanceDeletion等方法，当链表转树时，用于封装TreeNode，也就是说，ConcurrentHashMap的红黑树存放的时TreeBin，而不是treeNode。

```
static final class TreeBin<K,V> extends Node<K,V> {
    TreeNode<K,V> root;
    volatile TreeNode<K,V> first;
    volatile Thread waiter;
    volatile int lockState;
    // values for lockState
    static final int WRITER = 1; // set while holding write lock
    static final int WAITER = 2; // set when waiting for write lock
    static final int READER = 4; // increment value for setting read lock

    /**
     * Tie-breaking utility for ordering insertions when equal
     * hashCodes and non-comparable. We don't require a total
     * order, just a consistent insertion rule to maintain
     * equivalence across rebalancings. Tie-breaking further than
     * necessary simplifies testing a bit.
     */
    static int tieBreakOrder(Object a, Object b) {
        int d;
        if (a == null || b == null ||
            (d = a.getClass().getName().
             compareTo(b.getClass().getName())) == 0)
            d = (System.identityHashCode(a) <= System.identityHashCode(b) ?
                 -1 : 1);
        return d;
    }

    /**
     * Creates bin with initial set of nodes headed by b.
     */
    TreeBin(TreeNode<K,V> b) {
        super(TREEBIN, null, null, null);
        this.first = b;
        TreeNode<K,V> 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 = tieBreakOrder(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);
    }

    /**
     * Acquires write lock for tree restructuring.
     */
    private final void lockRoot() {
        if (!U.compareAndSwapInt(this, LOCKSTATE, 0, WRITER))
            contendedLock(); // offload to separate method
    }

    /**
     * Releases write lock for tree restructuring.
     */
    private final void unlockRoot() {
        lockState = 0;
```

```
    }

    /**
     * Possibly blocks awaiting root lock.
     */
    private final void contendedLock() {
        boolean waiting = false;
        for (int s;;) {
            if (((s = lockState) & ~WAITER) == 0) {
                if (U.compareAndSwapInt(this, LOCKSTATE, s, WRITER)) {
                    if (waiting)
                        waiter = null;
                    return;
                }
            }
            else if ((s & WAITER) == 0) {
                if (U.compareAndSwapInt(this, LOCKSTATE, s, s | WAITER)) {
                    waiting = true;
                    waiter = Thread.currentThread();
                }
            }
            else if (waiting)
                LockSupport.park(this);
        }
    }

    /**
     * Returns matching node or null if none. Tries to search
     * using tree comparisons from root, but continues linear
     * search when lock not available.
     */
    final Node<K,V> find(int h, Object k) {
        if (k != null) {
            for (Node<K,V> e = first; e != null; ) {
                int s; K ek;
                if (((s = lockState) & (WAITER|WRITER)) != 0) {
                    if (e.hash == h &&
                        ((ek = e.key) == k || (ek != null && k.equals(ek))))
                        return e;
                    e = e.next;
                }
                else if (U.compareAndSwapInt(this, LOCKSTATE, s,
                                             s + READER)) {
                    TreeNode<K,V> r, p;
                    try {
                        p = ((r = root) == null ? null :
                             r.findTreeNode(h, k, null));
                    } finally {
                        Thread w;
                        if (U.getAndAddInt(this, LOCKSTATE, -READER) ==
                            (READER|WAITER) && (w = waiter) != null)
                            LockSupport.unpark(w);
                    }
```

```
                    return p;
                }
            }
        }
        return null;
    }
...
```

ForwardingNode:

```
/**
 * A node inserted at head of bins during transfer operations.
 */
//在transfer操作中，一个节点插入到bins中
static final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }
    Node<K,V> find(int h, Object k) {
        // loop to avoid arbitrarily deep recursion on forwarding nodes
        outer: for (Node<K,V>[] tab = nextTable;;) {
            Node<K,V> e; int n;
            if (k == null || tab == null || (n = tab.length) == 0 ||
                (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;
                if (eh < 0) {
                    if (e instanceof ForwardingNode) {
                        tab = ((ForwardingNode<K,V>)e).nextTable;
                        continue outer;
                    }
                    else
                        return e.find(h, k);
                }
                if ((e = e.next) == null)
                    return null;
            }
        }
    }
}
```

基本属性：

```
transient volatile Node<K,V>[] table;//第一次插入的时候初始化
private transient volatile Node<K,V>[] nextTable;//resize的时候使用，nonnull
/**
```

```
 * Base counter value, used mainly when there is no contention,
 * but also as a fallback during table initialization
 * races. Updated via CAS.
 */
private transient volatile long baseCount;
/**
 * Table initialization and resizing control.  When negative, the
 * table is being initialized or resized: -1 for initialization,
 * else -(1 + the number of active resizing threads).  Otherwise,
 * when table is null, holds the initial table size to use upon
 * creation, or 0 for default. After initialization, holds the
 * next element count value upon which to resize the table.
 */
//负数代表正在进行初始化或扩容操作 ,其中-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，类似于扩容阈值。它的值始终是当前ConcurrentHashMap容量的0.75倍，这与loadfactor是对应的。实际容量>=sizeCtl，则扩容。
private transient volatile int sizeCtl;//table初始化和扩容的控制
/**
 * The next table index (plus one) to split while resizing.
 */
private transient volatile int transferIndex;
/**
 * Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
 */
private transient volatile int cellsBusy;
/**
 * Table of counter cells. When non-null, size is a power of 2.
 */
private transient volatile CounterCell[] counterCells;
```

构造函数

```
//1.
public ConcurrentHashMap() {
    }
//2.
public ConcurrentHashMap(int initialCapacity) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY :
               tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = cap;
}
//3.Creates a new map with the same mappings as the given map
public ConcurrentHashMap(Map<? extends K, ? extends V> m) {
    this.sizeCtl = DEFAULT_CAPACITY;
    putAll(m);
}
//4.
public ConcurrentHashMap(int initialCapacity, float loadFactor) {
    this(initialCapacity, loadFactor, 1);
```

```
}
//5.concurrencyLevel，表示能够同时更新ConccurentHashMap且不产生锁竞争的最大线程数。默认值为16，(即允许16个线程并发可能不会产生竞争)。为了保证并发的性能，我们要很好的估计出concurrencyLevel值，不然要么竞争相当厉害，从而导致线程试图写入当前锁定的段时阻塞。
public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {
    if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    if (initialCapacity < concurrencyLevel)   // Use at least as many bins
        initialCapacity = concurrencyLevel;   // as estimated threads
    long size = (long)(1.0 + (long)initialCapacity / loadFactor);
    int cap = (size >= (long)MAXIMUM_CAPACITY) ?
        MAXIMUM_CAPACITY : tableSizeFor((int)size);
    this.sizeCtl = cap;
}
```

put方法

putVal

将键值对映射到table中，key/value均不能为null

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

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException(); //都不能为null
    int hash = spread(key.hashCode()); //key的计算哈希值
    int binCount = 0; // 用于记录相应链表的长度
    for (Node<K,V>[] tab = table;;) { //死循环，直到插入成功
        Node<K,V> f; int n, i, fh;
        //1.如果没有初始化，则初始化
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        //2.找该 hash 值对应的数组下标，得到第一个节点 f
        //2.1 如果f为null,则直接cas设置，成功则跳出循环，失败，说明有并发操作，进入到下一次循环
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        //2.2 如果f不为null，f节点的hash等于MOVED，代表正在扩容，则帮助其扩容
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);  //帮助其扩容
        //2.3 如果f不为null，f节点的hash不等于MOVED。即f头节点部位null，而且没有在扩容
        else {
            V oldVal = null;
            /////// 锁定头节点f
            synchronized (f) {
                if (tabAt(tab, i) == f) { //避免多线程，需要重新检查头节点是否为f
                    //1.如果是链表节点，hash>=0,因为treebin为-2
                    //先查找链表中是否出现了此key，如果出现，则更新value，并跳出循环，
```

```
                    //否则将节点加入末尾并跳出循环
                    if (fh >= 0) {
                        binCount = 1;// 用于累加，记录链表的长度.1代表第一个节点f。
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            // 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
                            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;
                            // 到了链表的最末端，将这个新值放到链表的最后面并跳出循环
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key,value, null);
                                break;
                            }
                        }
                    }
                    //2.如果是树节点，则调用树的方法进行插入
                    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;
                        }
                    }
                }
            }
            //解锁

            //如果插入成功
            if (binCount != 0) {
                // 如果插入的是链表节点，则要判断下该桶位是否要转化为树。如果是红黑树，则binCount=2。
                // 判断是否要将链表转换为红黑树，临界值和 HashMap 一样，也是 8
                if (binCount >= TREEIFY_THRESHOLD)
                    // 这个方法和 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，
                    // 如果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
                    treeifyBin(tab, i);
                if (oldVal != null) //如果有旧值，则直接替代。无需扩容。
                    return oldVal;
                break;
            }
        }
    }
    addCount(1L, binCount);
    return null;
}
```

```
/*
putVal(K key, V value, boolean onlyIfAbsent)方法干的工作如下：
1、检查key/value是否为空，如果为空，则抛异常，否则进行2
2、进入for死循环，进行3
3、检查table是否初始化了，如果没有，则调用initTable()进行初始化然后进行 2，否则进行4
4、根据key的hash值计算出其应该在table中储存的位置i，取出table[i]的节点用f表示。
    根据f的不同有如下三种情况：
    1）如果table[i]==null(即该位置的节点为空，没有发生碰撞)， 则利用CAS操作直接存储在该位置，如果CAS操作成功则退出死循环。
    2）如果table[i]!=null(即该位置已经有其它节点，发生碰撞)，碰撞处理也有两种情况
        2.1）检查table[i]的节点的hash是否等于MOVED，如果等于，则检测到正在扩容，则帮助其扩容
        2.2）说明table[i]的节点的hash值不等于MOVED，如果table[i]为链表节点，则将此节点插入链表中即可
            如果table[i]为树节点，则将此节点插入树中即可。插入成功后，进行 5
5、如果table[i]的节点是链表节点，则检查table的第i个位置的链表是否需要转化为数，如果需要则调用treeifyBin函数进行转化
*/
```

1、第一步根据给定的key的hash值找到其在table中的位置index。

2、找到位置index后，存储进行就好了。

只是这里的存储有三种情况罢了，第一种：table[index]中没有任何其他元素，即此元素没有发生碰撞，这种情况直接存储就好了哈。第二种，table[i]存储的是一个链表，如果链表不存在key则直接加入到链表尾部即可，如果存在key则更新其对应的value。第三种，table[i]存储的是一个树，则按照树添加节点的方法添加就好。

在putVal函数，出现了如下几个函数

1、casTabAt tabAt 等CAS操作

2、initTable 作用是初始化table数组

3、treeifyBin 作用是将table[i]的链表转化为树

put流程总结：

1. 如果没有初始化，则cas设置sizeCrtl为-1，调用initTable初始化table
2. 死循环：
    1. 若头节点f为null，则cas设置头节点，跳出循环
    2. 若头节点f的哈希值为-1(MOVED)，表示正在扩容，调用helpTransfer(tab, f)帮助扩容
    3. 否则：
        1. 通过synchronized对头节点**加锁**，对链表/红黑树进行插入，然后**解锁**
        2. 如果插入的是链表，节点超过8，则调用treeifyBin(tab, i)
            1. tab<64,则调用tryPresize(n << 1)扩容
            2. 否则转树(**加锁**))
        3. 如果是替换，则直接返回oldValue
3. 调用addCount(1L, binCount)，返回null

cas机制：

```
//获取tab中索引为i的node
static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
    return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
}
// 利用CAS算法设置i位置上的Node节点（将c和table[i]比较，相同则插入v）。
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i, Node<K,V> c, Node<K,V> v) {
    return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}
// 设置节点位置的值，仅在上锁区被调用
static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {
    U.putObjectVolatile(tab, ((long)i << ASHIFT) + ABASE, v);
}
```

初始化表initTable()

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。

```
//用sizeCtl的大小来初始化表
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        //1.如果sizeCtl为负数，则说明已经有其它线程正在进行扩容，即正在初始化或初始化完成，
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        //2.如果CAS成功设置为-1，代表抢到了锁，然后初始化，否则说明其它线程正在初始化或是已经初始化完毕
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {//再一次检查确认是否还没有初始化
                    //如果sc>0,代表原来的sizeCtl的值为传入的初始化值，否则默认16
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    // 将这个数组赋值给 table，table 是 volatile 的
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);//即sc = 0.75n。
                }
            } finally {
                sizeCtl = sc;//sizeCtl = 0.75*Capacity,为扩容门限
            }
            break;
        }
    }
    return tab;//返回初始化后的表
}
```

链表转为树：treeifyBin

该方法的思想也相当的简单，检查下table的长度是否大于等于MIN_TREEIFY_CAPACITY（64），如果不大于，则调用tryPresize方法将table两倍扩容就可以了，就不降链表转化为树了。如果大于，则就将table[i]的链表转化为树。

```
//链表转树：将将数组tab的第index位置的链表转化为 树
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    if (tab != null) {
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)// 容量<64，则table两倍扩容，不转树了
            tryPresize(n << 1);
        //转为树，b 是头结点
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            //加锁
            synchronized (b) {
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    // 下面就是遍历链表，建立一颗红黑树
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)//第一次，设置头节点hd
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    // 将红黑树设置到数组相应位置中，根节点类型为TreeBin
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

扩容

扩容tryPresize

在putAll以及treeifyBin中调用

```
/**Tries to presize table to accommodate the given number of elements.
 * @param size number of elements (doesn't need to be perfectly accurate)*/
// 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
private final void tryPresize(int size) {
    //给定的容量若>=MAXIMUM_CAPACITY的一半，直接扩容到允许的最大值，否则调用tableSizeFor函数扩容
    // c：size 的 1.5 倍，再加 1，再往上取最近的 2 的 n 次方。
    int c = (size >= (MAXIMUM_CAPACITY >>> 1)) ? MAXIMUM_CAPACITY :
        tableSizeFor(size + (size >>> 1) + 1);
    int sc;
    //只有大于等于0才表示该线程可以扩容，具体看sizeCtl的含义
    while ((sc = sizeCtl) >= 0) {
        Node<K,V>[] tab = table; int n;
        //没有被初始化，则初始化
        if (tab == null || (n = tab.length) == 0) {
            n = (sc > c) ? sc : c;
```

```
            // 期间没有其他线程对表操作，则CAS将SIZECTL状态置为-1，表示正在进行初始化
            if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
                try {
                    if (table == tab) { //再一次检查
                        @SuppressWarnings("unchecked")
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                        table = nt;
                        sc = n - (n >>> 2); //无符号右移2位，此即0.75*n
                    }
                } finally {
                    sizeCtl = sc; // 更新扩容阀值
                }
            }
        }
        // 若欲扩容值小于等于原阀值，或现有容量>=最值，什么都不用做了
        else if (c <= sc || n >= MAXIMUM_CAPACITY)
            break;
        // table不为空，且在此期间其他线程未修改table
        else if (tab == table) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                Node<K,V>[] nt;
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                // 2. 用 CAS 将 sizeCtl 加 1，然后执行 transfer 方法
                //    此时 nextTab 不为 null
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            // 1. 将 sizeCtl 设置为 (rs << RESIZE_STAMP_SHIFT) + 2)，结果是一个比较大的负数
            //  当前线程是唯一的或是第一个发起扩容的线程  此时nextTable=null
            else if (U.compareAndSwapInt(this, SIZECTL, sc,(rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
        }
    }
}
```

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 transfer(tab, nt)。

所以，可能的操作就是执行 1 次 transfer(tab, null) + 多次 transfer(tab, nt)，这里怎么结束循环的需要看完 transfer 源码才清楚。

```
/**
 * Returns the stamp bits for resizing a table of size n.当扩容到n时，调用该函数返回一个标志位
 * Must be negative when shifted left by RESIZE_STAMP_SHIFT.
      numberOfLeadingZeros返回n对应32位二进制数左侧0的个数，如9（1001）返回28
      RESIZE_STAMP_BITS=16,
      因此返回值为：(参数n的左侧0的个数)|(2^15)
 */
static final int resizeStamp(int n) {
    return Integer.numberOfLeadingZeros(n) | (1 << (RESIZE_STAMP_BITS - 1));
}
```

数据迁移transfer

transfer()方法为ConcurrentHashMap扩容操作的核心方法。由于ConcurrentHashMap支持多线程扩容，而且也没有进行加锁，所以实现会变得有点儿复杂。整个扩容操作分为两步：

1. 构建一个nextTable，其大小为原来大小的两倍，这个步骤是在单线程环境下完成的
2. 将原来table里面的内容复制到nextTable中，这个步骤是允许多线程操作的，所以性能得到提升，减少了扩容的时间消耗

```
// Moves and/or copies the nodes in each bin to new table.
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    int n = tab.length, stride;
    // 每核处理的量小于16，则强制赋值16??
    // stride 在单核下直接等于 n，多核模式下为 (n>>>3)/NCPU，最小值是 16
    // stride 可以理解为”步长“，有 n 个位置是需要进行迁移的，
    //   将这 n 个任务分为多个任务包，每个任务包有 stride 个任务
    if ( (stride = (NCPU > 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
    // 前面我们说了，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
    // 之后参与迁移的线程调用此方法时，nextTab 不会为 null
    if (nextTab == null) {            // initiating
        try {
            @SuppressWarnings("unchecked")
            //构建nextTable对象,容量为原来的两倍
            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 是 ConcurrentHashMap 中的属性
        nextTable = nextTab;
        // transferIndex也是 ConcurrentHashMap 的属性，用于控制迁移的位置
        transferIndex = n;
    }
    int nextn = nextTab.length;  //next表的长度
    // 用于标志位（fwd的hash值为-1，fwd.nextTable=nextTab）
    // ForwardingNode 翻译过来就是正在被迁移的 Node
    // 这个构造方法会生成一个Node，key、value 和 next 都为 null，关键是 hash 为 MOVED
    // 后面我们会看到，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
//    所以它其实相当于是一个标志。
ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
// 当advance == true时，表明该节点已经处理过了
// advance 指的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 循环，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;
    // 控制 --i ,遍历原hash表中的节点
    // 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
    // 简单理解结局：i 指向了 transferIndex，bound 指向了 transferIndex-stride
    while (advance) {
        int nextIndex, nextBound;
        if (--i >= bound || finishing)
            advance = false;
        // 将 transferIndex 值赋给 nextIndex
        // 这里 transferIndex 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 ((nextIndex = transferIndex) <= 0) {
            i = -1;
            advance = false;
        }
        // 用CAS计算得到的transferIndex
        else if (U.compareAndSwapInt
                 (this, TRANSFERINDEX, nextIndex,
                  nextBound = (nextIndex > stride ?
                               nextIndex - stride : 0))) {
            // 看括号中的代码，nextBound 是这次迁移任务的边界，注意，是从后往前
            bound = nextBound;
            i = nextIndex - 1;
            advance = false;
        }
    }
    if (i < 0 || i >= n || i + n >= nextn) {
        int sc;
        // 所有的迁移操作已经完成
        if (finishing) {
            nextTable = null;
            table = nextTab;   // 将新的 nextTab 赋值给 table 属性，完成迁移
            // 重新计算 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 << 1) - (n >>> 1);
            return;            // 跳出死循环
        }
        // CAS 更扩容阈值，在这里面sizectl值减一，说明新加入一个线程参与到扩容操作
        // 之前我们说过，sizeCtl 在迁移前会设置为 (rs << RESIZE_STAMP_SHIFT) + 2
        // 然后，每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) << RESIZE_STAMP_SHIFT)
                    return;
                // 到这里，说明 (sc - 2) == resizeStamp(n) << RESIZE_STAMP_SHIFT，
                // 也就是说，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);
        // f.hash == -1 表示遍历到了ForwardingNode节点，意味着该节点已经处理过了
        // 这里是控制并发扩容的核心
        else if ((fh = f.hash) == MOVED)
            advance = true; // already processed
        else {
            // 对数组该位置处的结点加锁，开始处理数组该位置处的迁移工作
            synchronized (f) {
                // 节点复制工作
                if (tabAt(tab, i) == f) {
                    Node<K,V> ln, hn;
                    // fh >= 0 ,表示为链表节点
                    if (fh >= 0) {
                        // 构造两个链表  一个是原链表  另一个是原链表的反序排列
                        // 下面这一块和 Java7 中的 ConcurrentHashMap 迁移是差不多的，
                        // 需要将链表一分为二，
                        //   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
                        //   lastRun 之前的节点需要进行克隆，然后分到两个链表中
                        int runBit = fh & n;
                        Node<K,V> lastRun = f;
                        // lastRun指向的节点以及后面的节点所有节点的hash&n都相同（全部都移动 or not）
                        //仔细一看发现，如果没有第一个 for 循环，也是可以工作的，但是，这个 for 循环下来，如果 lastRun 的后面还有比较多的节点，那么这次就是值得的。因为我们只需要克隆 lastRun 前面的节点，后面的一串节点跟着 lastRun 走就是了，不需要做任何操作。
                        //不过比较坏的情况就是每次 lastRun 都是链表的最后一个元素或者很靠后的元素，那么这次遍历就有点浪费了。不过 Doug Lea 也说了，据统计，如果使用默认的阈值，大约只有 1/6 的节点需要克隆。
                        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;
                        }
```

```
            //遍历链表，采用头插法，所以元素的顺序就逆序了，但lastRun节点之后顺序不变
            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nextTable i 位置处插上链表ln
            setTabAt(nextTab, i, ln);
            // 在nextTable i 位置处插上链表hn
            setTabAt(nextTab, i + n, hn);
            // 将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
            setTabAt(tab, i, fwd);
            // advance = true 代表该位置已经迁移完毕,可以执行--i动作，继续遍历节点
            advance = true;
        }
         // 如果是TreeBin，则按照红黑树进行处理，处理逻辑与上面一致
        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;
                }
            }
            // 扩容后树节点个数若<=6，将树转链表
            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);
            setTabAt(tab, i, fwd);
```

```
                    advance = true;
                }
            }
        }
    }
}
```

说到底，transfer 这个方法并没有实现所有的迁移任务，每次调用这个方法只实现了 transferIndex 往前 stride 个位置的迁移工作，其他的需要由外围来控制。

这个时候，再回去仔细看 tryPresize 方法可能就会更加清晰一些了。

上面的源码有点儿长，稍微复杂了一些，在这里我们抛弃它多线程环境，我们从单线程角度来看：

1. 检查nextTable是否为null，如果是，则初始化nextTable，使其容量为table的两倍
2. 死循环遍历节点，知道finished：节点从table复制到nextTable中，支持并发，请思路如下：
    1. 如果节点 f 为null，则插入ForwardingNode（采用Unsafe.comareAndSwapObjectf方法实现），这个是触发并发扩容的关键
    2. 如果f为链表的头节点（fh >= 0），则先构造一个反序链表，然后把他们分别放在nextTable的i和i + n位置，并将ForwardingNode 插入原节点位置，代表已经处理过了
    3. 如果f为TreeBin节点，同样也是构造一个反序 ，同时需要判断是否需要进行unTreeify()操作，并把处理的结果分别插入到nextTable的i 和i+nw位置，并插入ForwardingNode 节点
3. 所有节点复制完成后，则将table指向nextTable，同时更新sizeCtl = nextTable的0.75倍，完成扩容过程

在多线程环境下，ConcurrentHashMap用两点来保证正确性：ForwardingNode和synchronized。当一个线程遍历到的节点如果是ForwardingNode，则继续往后遍历，如果不是，则将该节点加锁，防止其他线程进入，完成后设置ForwardingNode节点，以便要其他线程可以看到该节点已经处理过了，如此交叉进行，高效而又安全。

在put操作时如果发现fh.hash = -1，则表示正在进行扩容操作，则当前线程会协助进行扩容操作。

```
            else if ((fh = f.hash) == MOVED)
                tab = helpTransfer(tab, f);12
```

helpTransfer()方法为协助扩容方法，当调用该方法的时候，nextTable一定已经创建了，所以该方法主要则是进行复制工作。如下：

```
    final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
        Node<K,V>[] nextTab; int sc;
        if (tab != null && (f instanceof ForwardingNode) &&
                (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {
            int rs = resizeStamp(tab.length);
            while (nextTab == nextTable && table == tab &&
                    (sc = sizeCtl) < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                        sc == rs + MAX_RESIZERS || transferIndex <= 0)
                    break;
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
                    transfer(tab, nextTab);
                    break;
```

```
                }
            }
            return nextTab;
        }
        return table;
    }
```

size

ConcurrentHashMap的size()方法返回的是一个不精确的值，因为在进行统计的时候有其他线程正在进行插入和删除操作。

为了更好地统计size，ConcurrentHashMap提供了baseCount、counterCells两个辅助变量和一个CounterCell辅助内部类。

```
public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
            (int)n);
}
```

其实在1.8中，它不推荐size()方法，而是推崇mappingCount()方法，因为元素个数可能超过int？：

```
public long mappingCount() {//返回估计值，因为多线程可能会修改
    long n = sumCount();
    return (n < 0L) ? 0L : n; // ignore transient negative values
}
```

wr

```
/**
 * A padded cell for distributing counts.  Adapted from LongAdder
 * and Striped64.  See their internal docs for explanation.
 */
@sun.misc.Contended static final class CounterCell {
    volatile long value;
    CounterCell(long x) { value = x; }
}
//ConcurrentHashMap中元素个数,但返回的不一定是当前Map的真实元素个数。基于CAS无锁更新
private transient volatile long baseCount;
private transient volatile CounterCell[] counterCells;
```

arg

```
final long sumCount() {
    CounterCell[] as = counterCells; CounterCell a;
    long sum = baseCount;
    if (as != null) {//遍历，所有counter求和
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}
```

在put()方法最后会调用addCount()方法，该方法主要做两件事:1.更新baseCount的值，2.检测是否进行扩容，我们只看更新baseCount部分：

x == 1，如果counterCells == null，则U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)，如果并发竞争比较大可能会导致改过程失败，如果失败则最终会调用fullAddCount()方法。其实为了提高高并发的时候baseCount可见性的失败问题，又避免一直重试，JDK 8 引入了类Striped64,其中LongAdder和DoubleAdder都是基于该类实现的，而CounterCell也是基于Striped64实现的。如果counterCells != null，且uncontended = U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x)也失败了，同样会调用fullAddCount()方法，最后调用sumCount()计算s。

```
/**
 * Adds to count, and if table is too small and not already
 * resizing, initiates transfer. If already resizing, helps
 * perform transfer if work is available.  Rechecks occupancy
 * after a transfer to see if another resize is already needed
 * because resizings are lagging additions.
 *
 * @param x the count to add
 * @param check if <0, don't check resize, if <= 1 only check if uncontended
 */
private final void addCount(long x, int check) {
    CounterCell[] as; long b, s;
    // s = b + x，完成baseCount++操作；
    if ((as = counterCells) != null ||
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        CounterCell a; long v; int m;
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            !(uncontended =
              U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
            //  多线程CAS发生失败时执行
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        s = sumCount();
    }
    // 检查是否进行扩容,check >= 0 :则需要进行扩容操作
```

```
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            //当前线程是唯一的或是第一个发起扩容的线程  此时nextTable=null
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
            s = sumCount();
        }
    }
}
```

get

get 方法从来都是最简单的，这里也不例外：

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
    - 如果该位置为 null，那么直接返回 null 就可以了
    - 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
    - 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，后面我们再介绍 find 方法
    - 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<K,V>[] tab; Node<K,V> e, p; ``int` `n, eh; K ek;``
 ``int` `h = spread(key.hashCode());``    ``if` `((tab = table) != ``null` `&& (n = tab.length)
> ``0` `&&``        ``(e = tabAt(tab, (n - ``1``) & h)) != ``null``) {``        ``// 判断头结点是
否就是我们需要的节点``        ``if` `((eh = e.hash) == h) {``            ``if` `((ek = e.key) ==
key || (ek != ``null` `&& key.equals(ek)))``                ``return` `e.val;``        ``}``
   ``// 如果头结点的 hash 小于 0，说明 正在扩容，或者该位置是红黑树``        ``else` `if` `(eh <
``0``)``            ``// 参考 ForwardingNode.find(int h, Object k) 和 TreeBin.find(int h, Object
k)``            ``return` `(p = e.find(h, key)) != ``null` `? p.val : ``null``;` `        ``//
遍历链表``        ``while` `((e = e.next) != ``null``) {``            ``if` `(e.hash == h &&``
            ``((ek = e.key) == key || (ek != ``null` `&& key.equals(ek))))``
``return` `e.val;``        ``}``    ``}``    ``return` `null``;``}`
```

简单说一句，此方法的大部分内容都很简单，只有正好碰到扩容的情况，ForwardingNode.find(int h, Object k) 稍微复杂一些，不过在了解了数据迁移的过程后，这个也就不难了，所以限于篇幅这里也不展开说了。

接下来是看看正在扩容的情况：

```
static final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }
    Node<K,V> find(int h, Object k) {
        // loop to avoid arbitrarily deep recursion on forwarding nodes
        outer: for (Node<K,V>[] tab = nextTable;;) {
            Node<K,V> e; int n;
            if (k == null || tab == null || (n = tab.length) == 0 ||
                (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;
                if (eh < 0) {
                    if (e instanceof ForwardingNode) {
                        tab = ((ForwardingNode<K,V>)e).nextTable;
                        continue outer;
                    }
                    else
                        return e.find(h, k);
                }
                if ((e = e.next) == null)
                    return null;
            }
        }
    }
}
```

remove

删除时也需要确实扩容完成后才可以操作。

删除时，对头节点加锁删除

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

```
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 ||
            (f = tabAt(tab, i = (n - 1) & hash)) == null)
            break;
        else if ((fh = f.hash) == MOVED)//删除时也需要确实扩容完成后才可以操作。
            tab = helpTransfer(tab, f);
```

```
        else {
            V oldVal = null;
            boolean validated = false;
            synchronized (f) {
                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))) {//cv不为null则替换，否则删除。
                                    oldVal = ev;
                                    if (value != null)
                                        e.val = value;
                                    else if (pred != null)
                                        pred.next = e.next;
                                    else
                                        setTabAt(tab, i, e.next);//没前置节点就是头节点
                                }
                                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;
}
```

参考

http://www.importnew.com/28263.html