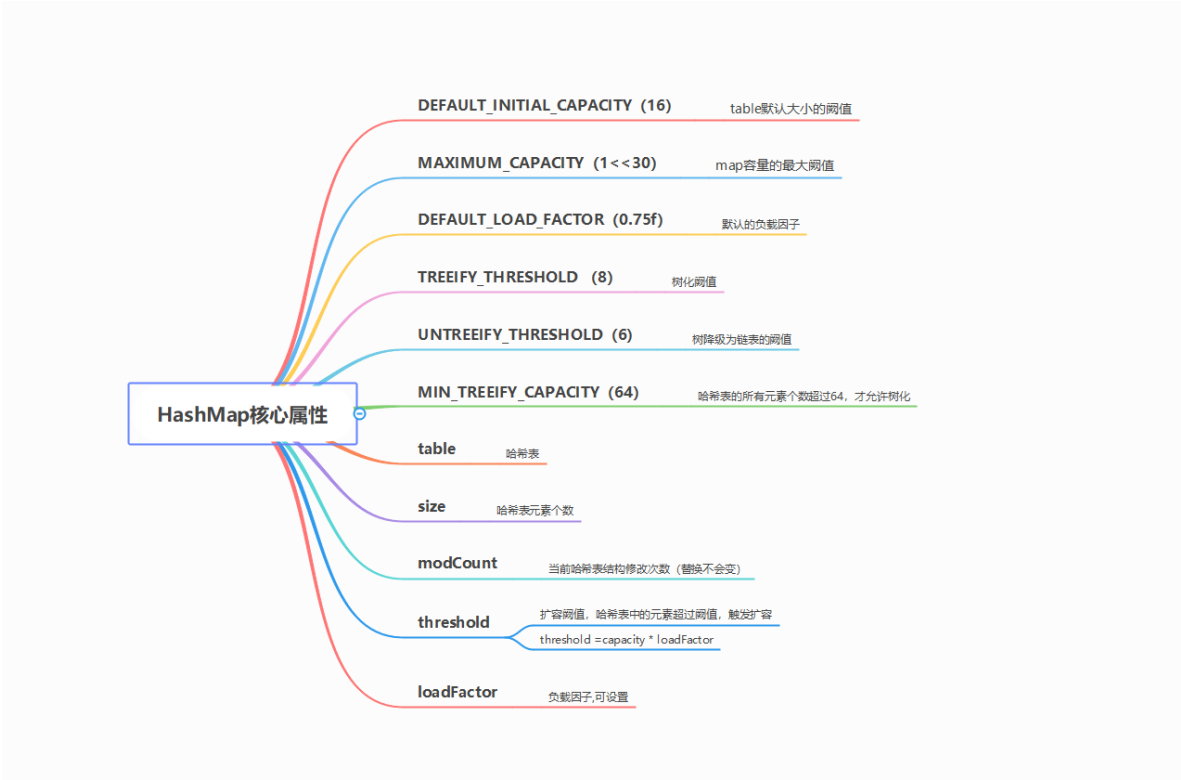
HashMap 源码阅读篇（jdk8）

1. hashmap 核心属性

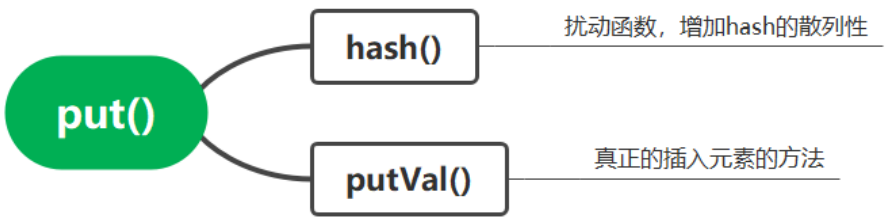
扩容阈值threshold = 负载因子 loadFactor*数组长度capacity



2.put()

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

put（）方法其实只有对两个方法的调用，**hash（）**和 **putVal（）**



扰动函数hash（）

```
    static final int hash(Object key) {  // 增加hash的散列性
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  // 异或运算：相同为0 不同为1
    }
```

分析：

这里的hash方法是传入一个key值，当key为空时，返回0，当key不为空时返回 一个int 型的 整数，这个整数是由key的哈希值h与上h右移16位得来的，这行代码的作用是**增加hash的散列性。**

hash()函数如何增加hash的散列性?

计算出key的哈希值h后，与上h的高16位，这样的好处是什么?

任何一个Object类型的hashCode方法得到的hash值是一个int类型，Java中int型是4*8=32位的，那么如果哈希表长度比较小，根据路由寻址算法hash&(length-1)就会导致hash值只有低位参与了运算，那些低位相同，高位不同的hash值就碰撞了，如：

```
// Hash碰撞示例：
H1: 00000000 00000000 00000000 00000101 & 1111 = 0101
H2: 00000000 11111111 00000000 00000101 & 1111 = 0101
```

当增加了扰动算法后，hash的高16位右移并与原Hash值进行**异或运算**，混合高16位和低16位的值，得到了一个更加散列的低16位的Hash值，如：

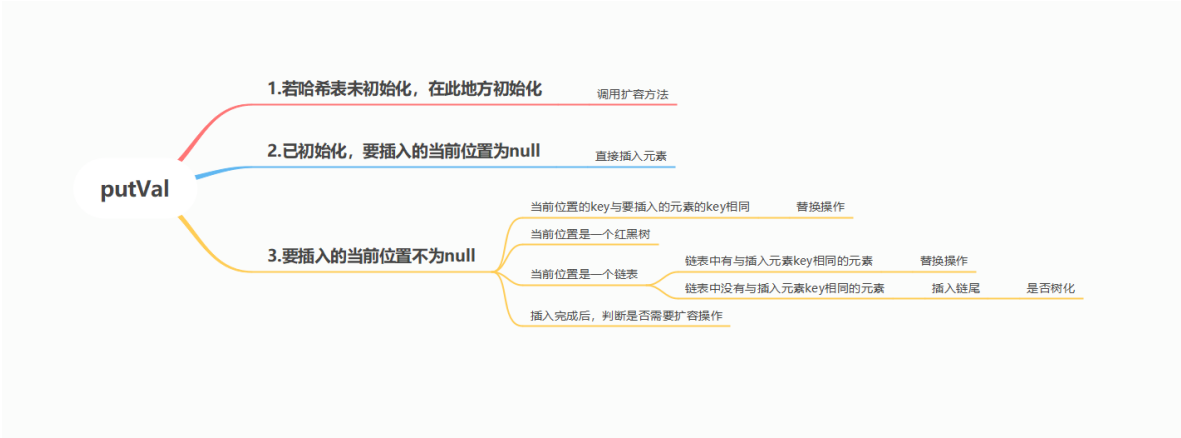
```
00000000 00000000 00000000 00000101 // H1
00000000 00000000 00000000 00000000 // H1 >>> 16
00000000 00000000 00000000 00000101 // hash = H1 ^ (H1 >>> 16) = 5

00000000 11111111 00000000 00000101 // H2
00000000 00000000 00000000 11111111 // H2 >>> 16
00000000 00000000 00000000 11111010 // hash = H2 ^ (H2 >>> 16) = 250
```

最终：

```
// 没有Hash碰撞
index1 = (n - 1) & H1 = (16 - 1) & 5 = 5
index2 = (n - 1) & H2 = (16 - 1) & 250 = 10
```

putVal()

put 操作流程脑图



因为哈希表初始化会占用很大内存，用户可能只是new HashMap 而没有去使用，延迟初始化可以减小内存的使用。

如果哈希表已经初始化了，则根据路由寻址（路由寻址算法：**桶位 = (table.length-1) & hash）**判断位桶的这个位置index是否有元素存在

1. 如果不存在元素，直接存入该位置
2. 如果存在元素，则判断桶位上的元素是否与要插入的相同，如果相同，执行替换操作
3. 如果不同，判断桶位上的元素是链表或者是红黑树。
4. 如果是TreeNode 树形结构，则进行putTreeVal（）树化操作。
5. 否则当前位置时一个链表结构，则遍历链表，查找是否有与key相同的元素，有就执行替换操作
6. 如果没有下一个next节点，则插入链尾，并判断是否需要执行树化方法

树化的条件是什么?

- 当链表的长度>=8且数组长度>=64时，会把链表转化成红黑树。
- 当链表长度>=8，但数组长度<64时，会优先进行扩容，而不是转化成红黑树。
- 当红黑树节点数<=6，自动转化成链表。

为什么需要数组长度到64才会转化红黑树?

当数组长度较短时，如16，链表长度达到8已经是占用了最大限度的50%，意味着负载已经快要达到上限，此时如果转化成红黑树，之后的扩容又会再一次把红黑树拆分平均到新的数组中，这样非但没有带来性能的好处，反而会降低性能。所以在数组长度低于64时，优先进行扩容。

为什么要大于等于8转化为红黑树，而不是7或9?

树节点的比普通节点更大，在链表较短时红黑树并未能明显体现性能优势，反而会浪费空间，在链表较短是采用链表而不是红黑树。在理论数学计算中（装载因子=0.75），链表的长度到达8的概率是百万分之一；把7作为分水岭，大于7转化为红黑树，小于7转化为链表。红黑树的出现是为了在某些极端的情况下，抗住大量的hash冲突，正常情况下使用链表是更加合适的

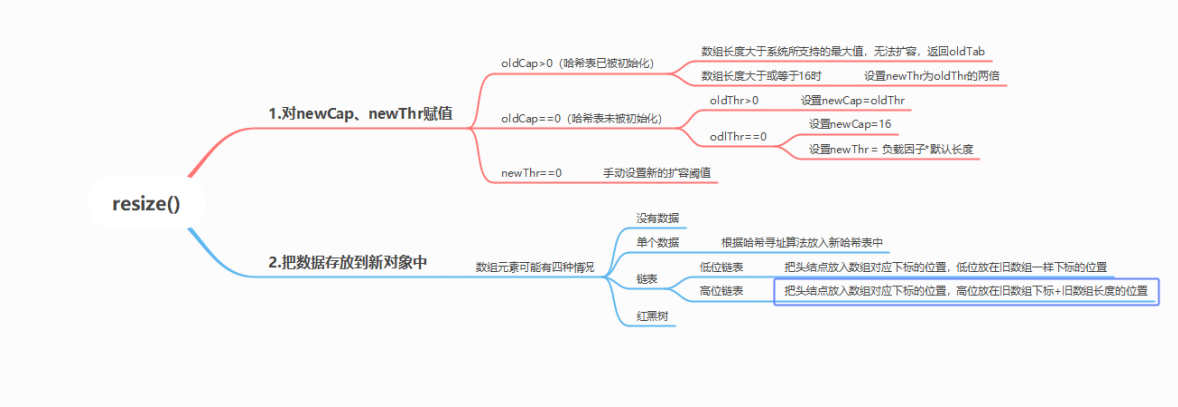
putVal()源码

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
```

```
        //tab 表示当前的哈希表
        //p 表示当前点的节点
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        //这里为tab初始化，即等于HashMap中的table，如果当前哈希表为空或者哈希表的长度为0
        if ((tab = table) == null || (n = tab.length) == 0)
            //调用扩容方法，并且给n赋值。注意：这里是哈希表的初始化，因为哈希表的初始化会占用很大内存，用户可能只是new HashMap 而没有使用，在putVal方法中初始化可以减小内存的使用
            n = (tab = resize()).length;
        //如果哈希表中数据不为空，设置p为当前节点
        //若p当前位置为空，（最简单的情况）
        if ((p = tab[i = (n - 1) & hash]) == null)
            //在当前下标位置赋值，完成插入
            tab[i] = newNode(hash, key, value, null);
        //p当前位置不为空
        //有三种情况：①当前位置的节点与要插入的元素key相同 - ② 红黑树 - ③链表
        else {
            //e 表示与当前插入的值一致的元素，k 表示临时的一个key
            Node<K,V> e; K k;
            //①如果当前节点的元素是否与要插入的元素相同
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                //把此时相同的节点的位置赋值给e，表示后续需要替换操作
                e = p;
            //②红黑树
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            //③链表
            else {
                //遍历链表
                for (int binCount = 0; ; ++binCount) {
                    //遍历到最后一个元素，没有找到与要插入的key相同的元素
                    if ((e = p.next) == null) {
                        //插入到尾节点
                        p.next = newNode(hash, key, value, null);
                        //判断插入当前元素后，检查是否达到树化标准，即判断是否是以下两种情况
                        //1.链表长度大于8，table元素个数超过64
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            //树化
                            treeifyBin(tab, hash);
                        break;
                    }
                    //在链表中找到了与插入元素key相同的node元素
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        //结束遍历，后续要执行替换操作
                        break;
                    //p=p.next
                    p = e;
                }
            }
            //e是与当前插入元素key相同的节点元素，当e不为null时，表示找到了相同key的元素。要执行替换操作
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    //新值替换旧值
                    e.value = value;
```

```
                afterNodeAccess(e);
                return oldValue; //返回旧值
            }
        }
        //插入新数据成功，table的修改值次数+1
        ++modCount;
        //判断长度是否达到扩容阈值
        if (++size > threshold)
            //扩容
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

3. 扩容方法resize()



扩容分析

扩容方法resize()在源码中分为两个部分

- 设置新的数组长度newCap和新的扩容阈值
- 旧数组容量超过阈值(初次默认16)

第一个部分是为了计算出扩容后的数组长度和扩容后的扩容阈值，这里根据为扩容前哈希表不同的情况有不同的计算方法，

第二个部分执行真正的扩容

- 新建一个新的数组
- 复制每个元素到新的数组中去
  （1）桶位元素是单个数据，直接根据路由寻址计算hash值放入新的哈希表中
  （2）桶位元素是一个链表，判断是**高位链表**还是**低位链表**
  高位链表，放到旧数组下标+上旧的数组长度的位置
  低位链表，放在与旧数组下标一样的位置

resize() 源码

```
final Node<K,V>[] resize() {
        //oldTab 表示的是扩容之前的哈希表
```

```
        Node<K,V>[] oldTab = table;
        //oldCap 表示的是扩容之前的数组长度,如果扩容前的数组为空，设置旧的数组长度为0，否则设置为旧哈希表长度
        int oldCap = (oldTab == null) ? 0 : oldTab.length;
        //pldThr 表示的是扩容之前数组的扩容阈值
        int oldThr = threshold;
        //newCap 表示的是新的数组长度，newThr 表示的是新的扩容阈值
        int newCap, newThr = 0;
        //当扩容前的数组长度大于0时，即此时数组正常初始化，里面存有数据
        if (oldCap > 0) {
            //当扩容前的数组大度大于系统支持的最大值时
            if (oldCap >= MAXIMUM_CAPACITY) {
                //设置扩容阈值为int最大值
                threshold = Integer.MAX_VALUE;
                //此时无法扩容，返回扩容前的哈希表
                return oldTab;
            }
            //此时数组长度在系统支持的范围内，设置新的数组长度为旧的数组长度的两倍，并且比较是否小于系统支持的最大值
            //与旧的数组长度是否大于等于16
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY)
                //满足上述两个条件时，新的扩容阈值增加为旧的扩容阈值的两倍
                newThr = oldThr << 1; // double threshold
        }
        //数组长度=0的情况，即哈希表未初始化，且旧的扩容阈值大于0
        //有以下几种情况：new HashMap(initCap,loadFactor),new HashMap(initCap),new HashMap(map)
        else if (oldThr > 0) // initial capacity was placed in threshold
            //设置新的数组长度为旧的数组扩容阈值
            newCap = oldThr;
        //oldCap==0且oldThr==0的情况
        else {               // zero initial threshold signifies using defaults
            //设置新的数组长度为默认值
            newCap = DEFAULT_INITIAL_CAPACITY;
            //使用负载因子计算新的扩容阈值
            newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
        }
        //如果新的扩容阈值等于0时，即旧的数组长度大于0，但不满足上述两种条件，导致newThr没有赋值
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor;
            newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
?
                      (int)ft : Integer.MAX_VALUE);
        }
        //设置扩容阈值为新的扩容阈值
        threshold = newThr;
        @SuppressWarnings({"rawtypes","unchecked"})
                //构造扩容后的哈希表对象
            Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        //设置table为新的哈希表
        table = newTab;
        //旧的哈希表不为空。即里面有数据
        if (oldTab != null) {
            //遍历数组
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
```

```
                if ((e = oldTab[j]) != null) {
                    //对象引用设为空，方便JVM回收
                    oldTab[j] = null;
                    //①这里表示数组中这个位置只有单个元素，不是链表
                    if (e.next == null)
                        //使用哈希寻址算法放入新哈希表中
                        newTab[e.hash & (newCap - 1)] = e;
                    //②此时这个元素是红黑树结构
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                    else { // preserve order
                        //③链表结构
                        //loHead表示低位链表的头结点，loTail表示低位链表的尾节点
                        //hiHead表示高位链表的头结点，hiTail表示高位链表的尾节点
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
                        //链表的下一个指针
                        Node<K,V> next;
                        //循环遍历链表
                        do {
                            //指向下一个节点
                            next = e.next;
                            //判断是否是低位链表
                            if ((e.hash & oldCap) == 0) {
                                //尾节点是否为空
                                if (loTail == null)
                                    //头结点指向e
                                    loHead = e;
                                else
                                    //尾节点下一个指向e
                                    loTail.next = e;
                                loTail = e;
                            }
                            else {
                                //判断高位尾部是否为空
                                if (hiTail == null)
                                    //头结点指向e
                                    hiHead = e;
                                else
                                    //尾节点下一位指向e
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);

                        //判断低位尾节点是否为空
                        if (loTail != null) {
                            //置为空，因为有可能存在其他引用
                            loTail.next = null;
                            //把头结点放入数组对应下标的位置，低位放在旧数组一样下标的位置
                            newTab[j] = loHead;
                        }
                        //判断高位尾节点是否为空
                        if (hiTail != null) {
                            //置为空，因为有可能存在其他引用
                            hiTail.next = null;
                            //把头结点放入数组对应下标的位置，高位放在旧数组下标+旧数组长
度的位置
```

```
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
        return newTab;
    }
```

如何判别低位链表和高位链表的呢?

使用e.hash & oldCap==0 表示是低位链表，否则就是高位链表。

分析：

什么是高位和低位链表?

以下面这张图为例，未扩容的数组长度为16，扩容后的数组长度为32。在未扩容前的哈希表的下标为15的位置中，由**hash&（n-1**）得出1111，因此，在下标15位置上的链表的元素的hash值后四位一定是1111。

那么就会出现两种情况：

- 1111 前面一位是1 即hash值为 ...1 1111
- 1111 前面一位是0 即hash值为 ...0 1111

根据前面一位是1或者0判断链表元素是高位还是低位，从而存放到指定的值。

知道了什么是高位和低位链表，根据 **e.hash & oldCap==0** 就可以判断出是高位还是低位链表，这个又是怎么判断的?

oldCap表示的是扩容前的数组长度，这里oldCap=16=1 0000

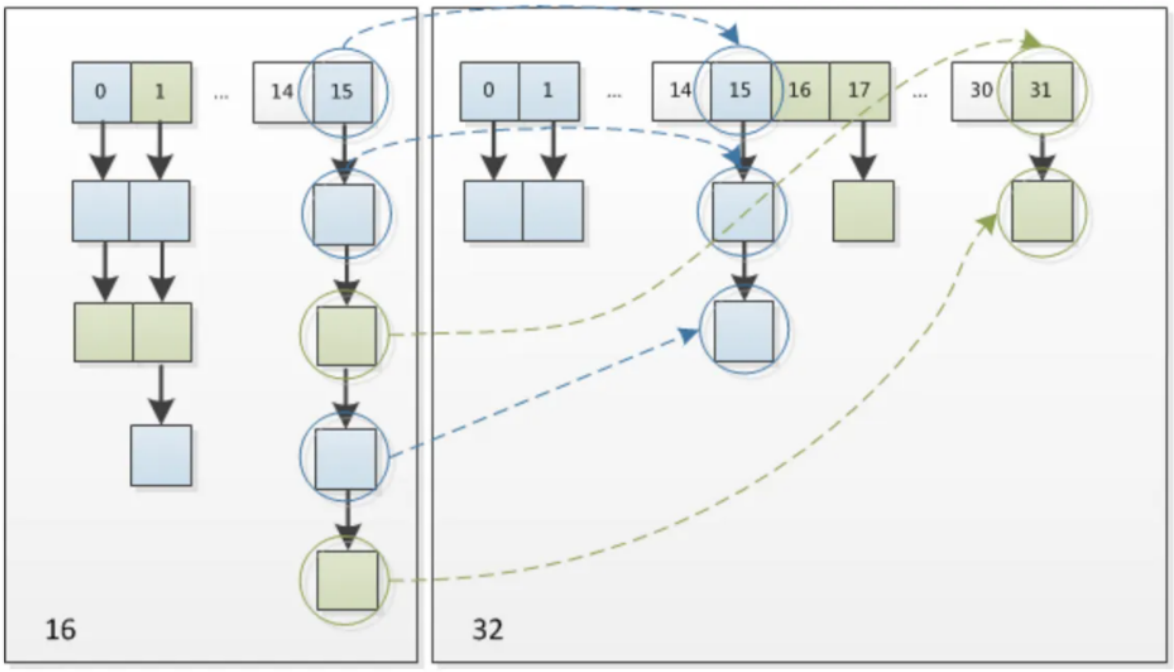
e.hash&oldCap有以下两种情况

（1）高链

```
  1 1111
& 1 0000
= 1 0000 高位是1，存入高链
```

（2）低链

```
  0 1111
& 1 0000
= 0 0000 高位是0，存入低链。
```



阅读原文：https://blog.csdn.net/qq_44830568/article/details/114606828?spm=1001.2014.3001.5501

4. get()

流程

1. 首先根据 hash 方法获取到 key 的 hash 值
2. 然后通过 hash & (length - 1) 的方式获取到 key 所对应的Node数组下标 (length对应数组长度)
3. 首先判断此结点是否为空，是否就是要找的值，是则返回空，否则进入第二个结点。
4. 接着判断第二个结点是否为空，是则返回空，不是则判断此时数据结构是链表还是红黑树
5. 链表结构进行顺序遍历查找操作，每次用 == 符号 和 equals( ) 方法来判断 key 是否相同，满足条件则直接返回该结点。链表遍历完都没有找到则返回空。
6. 红黑树结构执行相应的 getTreeNode( ) 查找操作。

源码分析

```
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;

    //Node数组不为空，数组长度大于0，数组对应下标的Node不为空
    if ((tab = table) != null && (n = tab.length) > 0 &&
        //也是通过 hash & (length - 1) 来替代 hash % length 的
        (first = tab[(n - 1) & hash]) != null) {

        //先和第一个结点比，hash值相等且key不为空，key的第一个结点的key的对象地址和值均相等
        //则返回第一个结点
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
```

```
            return first;
        //如果key和第一个结点不匹配，则看.next是否为空，不为null则继续，为空则返回null
        if ((e = first.next) != null) {
            //如果此时是红黑树的结构，则进行处理getTreeNode()方法搜索key
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            //是链表结构的话就一个一个遍历，直到找到key对应的结点，
            //或者e的下一个结点为null退出循环
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

getTreeNode() 红黑树查找：

```
  final TreeNode<K,V> getTreeNode(int h, Object k) {
            return ((parent != null) ? root() : this).find(h, k, null);
        }
```

```
/**
 * Finds the node starting at root p with the given hash and key.
 * The kc argument caches comparableClassFor(key) upon first use
 * comparing keys.
 */
final TreeNode<K,V> find(int h, Object k, Class<?> kc) {
    //获取当前对象
    TreeNode<K,V> p = this;
    //循环树结构
    do {
        int ph, dir; K pk;
        //获取当前节点的左子节点，右子节点
        TreeNode<K,V> pl = p.left, pr = p.right, q;
        //根据hash值判断，p=左子节点，或右子节点
        if ((ph = p.hash) > h)
            p = pl;
        else if (ph < h)
            p = pr;
        //p的key与之key对比，如果相同，则返回当前对象
        else if ((pk = p.key) == k || (k != null && k.equals(pk)))
            return p;
        //如果左子节点为空，则p=右子节点
        else if (pl == null)
            p = pr;
         //如果右子节点为空，则p=左子节点
        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;
            //嵌套查询，如果找到，则返回该对象
            else if ((q = pr.find(h, k, kc)) != null)
                return q;
            else
                p = pl;
        //循环对象，直到找到，或者循环结束
        } while (p != null);
        return null;
    }
```