HashMap,ConcurrentHashMap原理及源码

Jdk1.8版本

HashMap1.8中，底层实现由之前的 **数组+链表** 改为了 **数组+链表+红黑树**

HashMap里面是一个数组

**transient** Node<K,V>[] **table**;

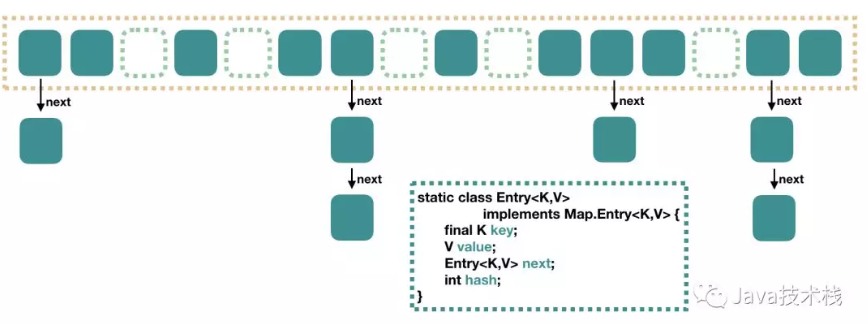
数组中的每一个元素是一个单向链表 （Node是单项链表,next是链表中的节点entry）

**static class** Node<K,V> **implements** Map.Entry<K,V> {

**final int hash**;  
 **final** K **key**;  
 V **value**;  
 Node<K,V> **next**;

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

}



上图中，每一个绿色的实体是嵌套类entry的实例

Entry中包含四个属性：key,value,hash,和用于单项链表的next

**TreeNode：红黑树节点**

**static final class** TreeNode<K,V> **extends** LinkedHashMap.Entry<K,V> {  
 TreeNode<K,V> **parent**; *// 父节点* 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) {  
 **super**(hash, key, val, next);  
 }  
  
 */\*\*  
 \* Returns root of tree containing this node.  
 \*/* **final** TreeNode<K,V> root() {  
 **for** (TreeNode<K,V> r = **this**, p;;) {  
 **if** ((p = r.**parent**) == **null**)  
 **return** r;  
 r = p;  
 }  
 }

HashMap的默认初始容量为1<<4 就是**16**

**static final int *DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4; *// aka 16*

*HashMap的最大容量 1<<30*

**static final int *MAXIMUM\_CAPACITY*** = 1 << 30;

HashMap的负载因子 loadFactor **0.75f**

**static final float *DEFAULT\_LOAD\_FACTOR*** = 0.75f;

HashMap entry数组中链表转为红黑树的阈值 **treeify threshold >=8**

**static final int *TREEIFY\_THRESHOLD*** = 8;

*HashMap entry数组 中由红黑树转为链表的阈值* ***untreeify threshold <=6***

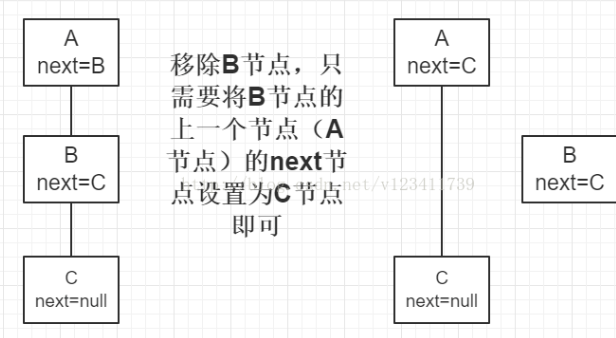
**static final int *UNTREEIFY\_THRESHOLD*** = 6;

HashMap 转红黑树时 ，table的最小长度

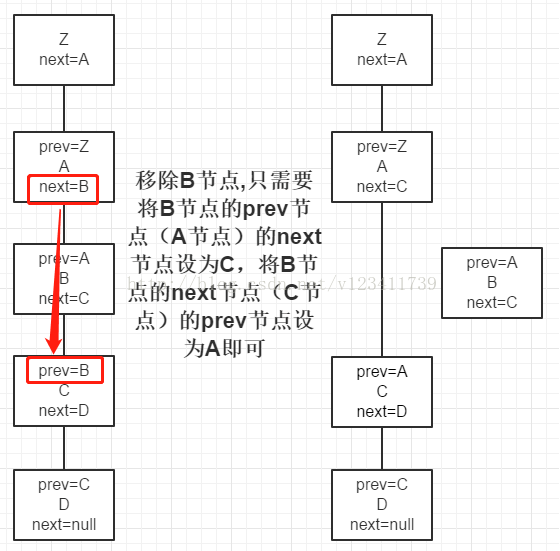
**static final int *MIN\_TREEIFY\_CAPACITY*** = 64;

转为红黑树节点后，链表的结构还在，通过next属性维持，红黑树节点在进行操作的时候都会维护链表的结构，并不是转为红黑树后，链表结构就不存在了

链表中一处一个节点秩序要做如图下的操作



红黑树在维护链表结构时，移除一个节点秩序做如下图操作



源码中进行红黑树查找时，会反复用到两条规则

1. 若目标节点的hash值小于p节点的hash值，则向p节点的左边遍历；

反之，向p节点的右边遍历

1. 若目标节点的key值小于p节点的key值，则向p节点的左边遍历

反之，向p节点的右边遍历

这两条规则利用了**红黑树的特性： 左节点 < 根节点 < 右节点**

**定位哈希桶数组索引的位置**

不管增删查，定位到哈希桶数组的位置都是很关键的一步。

HashMap的数据结构是 数组+链表+红黑树 的结合，我们希望这个hashMap中的元素位置进料量分布的均匀一些，尽量使每隔位置只有一个元素。

这样当用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们想要的，

省去了遍历红黑树的过程，这样就大大的优化了查询效率

**HashMap定位数组的索引位置，直接决定了hash方法的离散性能**

**下面是定位哈希桶数组的源码：**

**// 代码1**

**static final int hash(Object key) { // 计算key的hash值**

**int h;**

**// 1.先拿到key的hashCode值; 2.将hashCode的高16位参与运算**

**return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);**

**}**

**// 代码2**

**int n = tab.length;**

**// 将(tab.length - 1) 与 hash值进行&运算**

**int index = (n - 1) & hash; //&：位运算符 全1为1，否则为0 &&：逻辑运算符**

**上面代码分为三步：**

1. **算出key的hashCode值**
2. **将hashCode值的高位参与运算，重新计算hash值**
3. **将计算出来的hash值与 table.length - 1 进行 &运算**

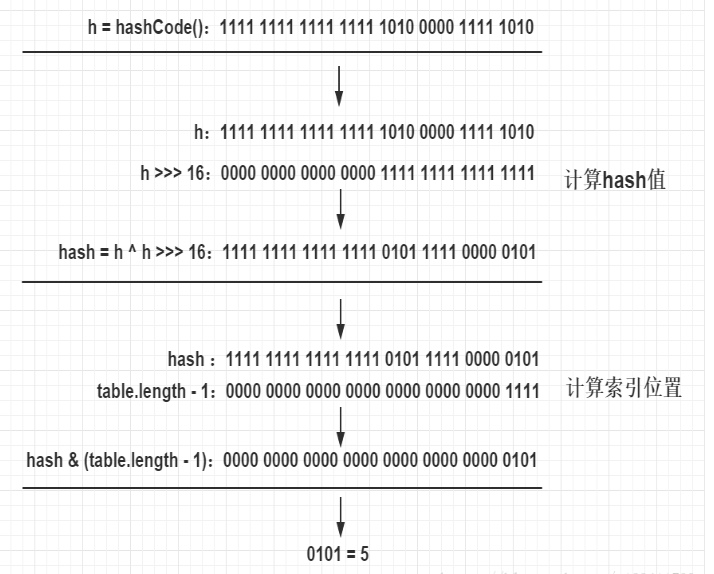
**方法解读：**

对于任意给定的对象，只要它的hashCode()返回值相同，那么计算得到的hash值也是相同的。我们首先想到的就是把hash值与table长度进行取模运算，这样元素的分布相对来说比较均匀。

但是取模运算消耗还是比较大的，在jdk1.8之后，换成了上面的**代码2**的位与运算来代替取模运算。这个方法非常巧妙，它通过(table.length - 1) & h 来得到该对象的索引位置，这个优化是基于以下公式：x mod 2^n = x & (2^n - 1)。HashMap底层数组的长度总是2的n次方，并且取模运算为 h mod table.length，对应上面的公式，可以得到该公式等同于 h & (table.length -1) 。这是HashMap在速度上的优化，因为& 比 % 具有更高的效率。

并且还有话了高位运算的算法，将hashCode的高16位与hashCode进行异或运算，主要是为了在table的length较小的时候，让高位也参与运算，并且不会又太大的开销。

下面的例子，table.length位16



**Get()方法**

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;

**// table不为空 && table长度大于0 && table索引位置(根据hash值计算出)不为空**

if ((tab = table) != null && (n = tab.length) > 0 &&

(first = tab[(n - 1) & hash]) != null) {

if (first.hash == hash && // always check first node

((k = first.key) == key || (key != null && key.equals(k))))

return first; **// first的key等于传入的key则返回first对象**

if ((e = first.next) != null) { **// 向下遍历**

if (first instanceof TreeNode) // 判断是否为TreeNode

**// 如果是红黑树节点，则调用红黑树的查找目标节点方法getTreeNode**

return ((TreeNode<K,V>)first).getTreeNode(hash, key);

**// 走到这代表节点为链表节点**

do { **// 向下遍历链表, 直至找到节点的key和传入的key相等时,返回该节点**

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

return e;

} while ((e = e.next) != null);

}

}

return null; **// 找不到符合的返回空**

**}**

**方法解读：**

1. 先对table进行校验，是否为空，长度是否大于0 (tab = table) != null && (n = tab.length) > 0
2. Table.length - 1 和hash值进行位与运算，得出在table上的索引位置，将该索引位置上的节点赋值给first节点，校验该索引位置是否为空 --> (first = tab[(n - 1) & hash]) != null)
3. 检查first节点的hash值和 hash(key)一样，若一样，则first即为目标节点，直接返回first节点

(first.hash == hash && // always check first node

((k = first.key) == key || (key != null && key.equals(k))))

1. 若first的next节点不为空则继续遍历 if ((e = first.next) != null)
2. 判断first节点是否位treeNode节点，是的话调用 **getTreeNode()**方法查找目标节点

if (first instanceof TreeNode)

Return ((TreeNode<K,V>)first).getTreeNode(hash, key);

1. 若first节点不上TreeNode节点，则调用普通的遍历链表方法查找目标节点
2. 若查不到 return null；

**GetTreeNode()方法**：

final TreeNode<K,V> getTreeNode(int h, Object k) {

// 使用根结点调用find方法

return ((parent != null) ? root() : this).find(h, k, null);

}

1. 找到调用此方法的节点的树的根节点
2. 使用该树的根节点调用**find()**方法

**Find()方法**

/\*\*

\* 从调用此方法的结点开始查找, 通过hash值和key找到对应的节点

\* 此处是红黑树的遍历, 红黑树是特殊的自平衡二叉查找树

\* 平衡二叉查找树的特点：左节点<根节点<右节点

\*/

final TreeNode<K,V> find(int h, Object k, Class<?> kc) {

TreeNode<K,V> p = this; **// this为调用此方法的节点**

do {

int ph, dir; K pk;

TreeNode<K,V> pl = p.left, pr = p.right, q;

if ((ph = p.hash) > h) // 传入的hash值小于p节点的hash值, 则往p节点的左边遍历

p = pl; // p赋值为p节点的左节点

else if (ph < h) // 传入的hash值大于p节点的hash值, 则往p节点的右边遍历

p = pr; // p赋值为p节点的右节点

// 传入的hash值和key值等于p节点的hash值和key值,则p节点为目标节点,返回p节点

else if ((pk = p.key) == k || (k != null && k.equals(pk)))

return p;

else if (pl == null) // p节点的左节点为空则将向右遍历

p = pr;

else if (pr == null) // p节点的右节点为空则向左遍历

p = pl;

else if ((kc != null ||

// 如果传入的key(k)所属的类实现了Comparable接口,则将传入的key跟p节点的key比较

(kc = comparableClassFor(k)) != null) && // 此行不为空代表k实现了Comparable

(dir = compareComparables(kc, k, pk)) != 0)//k<pk则dir<0, k>pk则dir>0

p = (dir < 0) ? pl : pr; // k < pk则向左遍历(p赋值为p的左节点), 否则向右遍历

// 代码走到此处, 代表key所属类没有实现Comparable, 直接指定向p的右边遍历

else if ((q = pr.find(h, k, kc)) != null)

return q;

else// 代码走到此处代表上一个向右遍历（pr.find(h, k, kc)）为空, 因此直接向左遍历

p = pl;

} while (p != null);

return null;

}

1. 将p节点赋值位调用此方法的节点 TreeNode<K,V> p = this;
2. 若传入的hash值小于p节点的hash值，则在p节点的左边遍历

if ((ph = p.hash) > h)

P = pl;

1. 若传入的hash值大于p节点的hash值，则在p节点的右边遍历

else if (ph < h)

p = pr;

1. 若传入的hash值等于p节点的hash值，且传入的key值和p节点的key值相等，则该p节点即为目标节点， 返回p节点
2. 若p节点的左边为空则向右遍历，反之向左遍历
3. 若果传入的key（参数变量k） 所属的类实现了comparable接口（kc不为空，**comparableClassFor()**方法见下 面），则将传入的key跟p节点的key进行比较（kc实现了comparable接口，因此通过kc的比较方法进行比 较），将比较结果赋值给dir，若dir < 0 则代表 k < pk ,则向p节点的左边遍历(pl) ；反之，向p节点的右边 遍历（pr）
4. 代码走到此处，说明key所属类没有实现comparable接口，隐藏直接制定向p的右边遍历，若能找到目标节点则返回
5. 右边遍历没有找到目标节点，此步向左边遍历
6. 以上都没有找到，return null;

## ****comparableClassFor方法****

/\*\*

\* Returns x's Class if it is of the form "class C implements

\* Comparable<C>", else null.

\*/

static Class<?> comparableClassFor(Object x) {

if (x instanceof Comparable) {

Class<?> c; Type[] ts, as; Type t; ParameterizedType p;

if ((c = x.getClass()) == String.class) // bypass checks

return c;

if ((ts = c.getGenericInterfaces()) != null) {

for (int i = 0; i < ts.length; ++i) {

if (((t = ts[i]) instanceof ParameterizedType) &&

((p = (ParameterizedType)t).getRawType() ==

Comparable.class) &&

(as = p.getActualTypeArguments()) != null &&

as.length == 1 && as[0] == c) // type arg is c

return c;

}

}

}

return null;

}

若果x实现了comparable接口，则返回x的class

# ****put方法****

public V put(K key, V value) {

return putVal(hash(key), key, value, false, true);

}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,

boolean evict) {

Node<K,V>[] tab; Node<K,V> p; int n, i;

**// table是否为空或者length等于0, 如果是则调用resize方法进行初始化**

if ((tab = table) == null || (n = tab.length) == 0)

n = (tab = resize()).length;

**// 通过hash值计算索引位置, 如果table表该索引位置节点为空则新增一个**

if ((p = tab[i = (n - 1) & hash]) == null**)// 将索引位置的头节点赋值给p**

tab[i] = newNode(hash, key, value, null);

else {  **// table表该索引位置不为空**

Node<K,V> e; K k;

if (p.hash == hash && **// 判断p节点的hash值和key值是否跟传入的hash值和key值相等**

((k = p.key) == key || (key != null && key.equals(k))))

e = p;  **// 如果相等, 则p节点即为要查找的目标节点，赋值给e**

**// 判断p节点是否为TreeNode, 如果是则调用红黑树的putTreeVal方法查找目标节点**

else if (p instanceof TreeNode)

e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

else { **// 走到这代表p节点为普通链表节点**

for (int binCount = 0; ; ++binCount) { // 遍历此链表, binCount用于统计节点数

if ((e = p.next) == null) { // p.next为空代表不存在目标节点则新增一个节点插入链表尾部

p.next = newNode(hash, key, value, null);

// 计算节点是否超过8个, 减一是因为循环是从p节点的下一个节点开始的

if (binCount >= TREEIFY\_THRESHOLD - 1)

treeifyBin(tab, hash);// 如果超过8个，调用treeifyBin方法将该链表转换为红黑树

break;

}

if (e.hash == hash && // e节点的hash值和key值都与传入的相等, 则e即为目标节点,跳出循环

((k = e.key) == key || (key != null && key.equals(k))))

break;

p = e; // 将p指向下一个节点

}

}

// e不为空则代表根据传入的hash值和key值查找到了节点,将该节点的value覆盖,返回oldValue

if (e != null) {

V oldValue = e.value;

if (!onlyIfAbsent || oldValue == null)

e.value = value;

afterNodeAccess(e); // 用于LinkedHashMap

return oldValue;

}

}

++modCount;

if (++size > threshold) // 插入节点后超过阈值则进行扩容

resize();

afterNodeInsertion(evict); // 用于LinkedHashMap

return null;

}

方法解析：

1. 判断table是否为空或者length等于0，如果是则调用resize()方法初始化
2. 通过hash值计算索引位置，将改索引位置的头节点赋值给p节点，若改索引位置节点为空则使用传入的参数新增一个节点并放在改索引位置
3. 判断p节点的key和hash值是否跟传入的相等，若相等，则p节点就是要查找的目标节点，将p节点赋值给e节点
4. 若p节点不是目标节点，则判断p节点是否位TreeNode，若是，则调用红黑树的**putTreeVal()**查找目标节点
5. 此步表示p节点位普通节点，调用普通的链表方法进行查找，并定义binCount来统计改链表的节点数
6. 若p的next节点为空时，则代表找不到目标节点，则新增一个节点并插入链表尾部，并校验节点数是否超过8个，若超过则调用treeifyBin()方法，将链表节点转为红黑树节点
7. 若遍历的e节点存在hash值和key值都与传入的相同，则e节点即为目标节点，break;
8. 若e节点不为空，则代表目标节点存在，使用传入的value覆盖该节点的value，并返回oldValue
9. 若插入节点后，节点数超过了阈值，则调用resize()方法进行扩容

## **putTreeVal方法**

/\*\*

\* Tree version of putVal.

\* 红黑树插入会同时维护原来的链表属性, 即原来的next属性

\*/

final TreeNode<K,V> putTreeVal(HashMap<K,V> map, Node<K,V>[] tab,

int h, K k, V v) {

Class<?> kc = null;

boolean searched = false;

// 查找根节点, 索引位置的头节点并不一定为红黑树的根结点

TreeNode<K,V> root = (parent != null) ? root() : this;

for (TreeNode<K,V> p = root;;) { // 将根节点赋值给p, 开始遍历

int dir, ph; K pk;

if ((ph = p.hash) > h) // 如果传入的hash值小于p节点的hash值

dir = -1; // 则将dir赋值为-1, 代表向p的左边查找树

else if (ph < h) // 如果传入的hash值大于p节点的hash值,

dir = 1; // 则将dir赋值为1, 代表向p的右边查找树

// 如果传入的hash值和key值等于p节点的hash值和key值, 则p节点即为目标节点, 返回p节点

else if ((pk = p.key) == k || (k != null && k.equals(pk)))

return p;

// 如果k所属的类没有实现Comparable接口 或者 k和p节点的key相等

else if ((kc == null &&

(kc = comparableClassFor(k)) == null) ||

(dir = compareComparables(kc, k, pk)) == 0) {

if (!searched) { // 第一次符合条件, 该方法只有第一次才执行

TreeNode<K,V> q, ch;

searched = true;

// 从p节点的左节点和右节点分别调用find方法进行查找, 如果查找到目标节点则返回

if (((ch = p.left) != null &&

(q = ch.find(h, k, kc)) != null) ||

((ch = p.right) != null &&

(q = ch.find(h, k, kc)) != null))

return q;

}

// 否则使用定义的一套规则来比较k和p节点的key的大小, 用来决定向左还是向右查找

dir = tieBreakOrder(k, pk); // dir<0则代表k<pk，则向p左边查找；反之亦然

}

TreeNode<K,V> xp = p; // xp赋值为x的父节点,中间变量,用于下面给x的父节点赋值

// dir<=0则向p左边查找,否则向p右边查找,如果为null,则代表该位置即为x的目标位置

if ((p = (dir <= 0) ? p.left : p.right) == null) {

// 走进来代表已经找到x的位置，只需将x放到该位置即可

Node<K,V> xpn = xp.next; // xp的next节点

// 创建新的节点, 其中x的next节点为xpn, 即将x节点插入xp与xpn之间

TreeNode<K,V> x = map.newTreeNode(h, k, v, xpn);

if (dir <= 0) // 如果时dir <= 0, 则代表x节点为xp的左节点

xp.left = x;

else // 如果时dir> 0, 则代表x节点为xp的右节点

xp.right = x;

xp.next = x; // 将xp的next节点设置为x

x.parent = x.prev = xp; // 将x的parent和prev节点设置为xp

// 如果xpn不为空,则将xpn的prev节点设置为x节点,与上文的x节点的next节点对应

if (xpn != null)

((TreeNode<K,V>)xpn).prev = x;

moveRootToFront(tab, balanceInsertion(root, x)); // 进行红黑树的插入平衡调整

return null;

}

}

}

1.查找当前红黑树的根结点，将根结点赋值给p节点，开始进行查找

2.如果传入的hash值小于p节点的hash值，将dir赋值为-1，代表向p的左边查找树

3.如果传入的hash值大于p节点的hash值， 将dir赋值为1，代表向p的右边查找树

4.如果传入的hash值等于p节点的hash值，并且传入的key值跟p节点的key值相等, 则该p节点即为目标节点，返回p节点

5.如果k所属的类没有实现Comparable接口，或者k和p节点的key使用compareTo方法比较相等：第一次会从p节点的左节点和右节点分别调用find方法（见上文代码块2）进行查找，如果查找到目标节点则返回；如果不是第一次或者调用find方法没有找到目标节点，则调用tieBreakOrder方法（见下文代码块5）比较k和p节点的key值的大小，以决定向树的左节点还是右节点查找。

6.如果dir <= 0则向左节点查找（p赋值为p.left，并进行下一次循环），否则向右节点查找，如果已经无法继续查找（p赋值后为null），则代表该位置即为x的目标位置，另外变量xp用来记录查找的最后一个节点，即下文新增的x节点的父节点。

7.以传入的hash、key、value参数和xp节点的next节点为参数，构建x节点（注意：xp节点在此处可能是叶子节点、没有左节点的节点、没有右节点的节点三种情况，即使它是叶子节点，它也可能有next节点，红黑树的结构跟链表的结构是互不影响的，不会因为某个节点是叶子节点就说它没有next节点，红黑树在进行操作时会同时维护红黑树结构和链表结构，next属性就是用来维护链表结构的），根据dir的值决定x决定放在xp节点的左节点还是右节点，将xp的next节点设为x，将x的parent和prev节点设为xp，如果原xp的next节点（xpn）不为空, 则将该节点的prev节点设置为x节点, 与上面的将x节点的next节点设置为xpn对应。

8.进行红黑树的插入平衡调整，见文末的解释2。

## ****tieBreakOrder方法****

// 用于不可比较或者hashCode相同时进行比较的方法, 只是一个一致的插入规则，用来维护重定位的等价性。

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;

}

定义一套规则，用于极端情况比较两个参数的大小

## ****treeifyBin方法****

final void treeifyBin(Node<K,V>[] tab, int hash) {

int n, index; Node<K,V> e;

// table为空或者table的长度小于64, 进行扩容

if (tab == null || (n = tab.length) < MIN\_TREEIFY\_CAPACITY)

resize();

// 根据hash值计算索引值, 遍历该索引位置的链表

else if ((e = tab[index = (n - 1) & hash]) != null) {

TreeNode<K,V> hd = null, tl = null;

do {

TreeNode<K,V> p = replacementTreeNode(e, null); // 链表节点转红黑树节点

if (tl == null) // tl为空代表为第一次循环

hd = p; // 头结点

else {

p.prev = tl; // 当前节点的prev属性设为上一个节点

tl.next = p; // 上一个节点的next属性设置为当前节点

}

tl = p; // tl赋值为p, 在下一次循环中作为上一个节点

} while ((e = e.next) != null); // e指向下一个节点

// 将table该索引位置赋值为新转的TreeNode的头节点

if ((tab[index] = hd) != null)

hd.treeify(tab); // 以头结点为根结点, 构建红黑树

}

}

1.校验table是否为空，如果长度小于64，则调用resize方法（见下文resize方法）进行扩容。

2.根据hash值计算索引值，将该索引位置的节点赋值给e节点，从e节点开始遍历该索引位置的链表。

3.调用replacementTreeNode方法（该方法就一行代码，直接返回一个新建的TreeNode）将链表节点转为红黑树节点， 将头结点赋值给hd节点，每次遍历结束将p节点赋值给tl，用于在下一次循环中作为上一个节点进行一些链表的 关联操作（p.prev = tl 和 tl.next = p）。

4.将table该索引位置赋值为新转的TreeNode的头节点hd，如果该节点不为空，则以hd为根结点，调用treeify方法（见下文代码块7）构建红黑树。

## ****treeify方法****

**final void treeify(Node<K,V>[] tab) { // 构建红黑树**

**TreeNode<K,V> root = null;**

**for (TreeNode<K,V> x = this, next; x != null; x = next) {// this即为调用此方法的TreeNode**

**next = (TreeNode<K,V>)x.next; // next赋值为x的下个节点**

**x.left = x.right = null; // 将x的左右节点设置为空**

**if (root == null) { // 如果还没有根结点, 则将x设置为根结点**

**x.parent = null; // 根结点没有父节点**

**x.red = false; // 根结点必须为黑色**

**root = x; // 将x设置为根结点**

**}**

**else {**

**K k = x.key; // k赋值为x的key**

**int h = x.hash; // h赋值为x的hash值**

**Class<?> kc = null;**

**// 如果当前节点x不是根结点, 则从根节点开始查找属于该节点的位置**

**for (TreeNode<K,V> p = root;;) {**

**int dir, ph;**

**K pk = p.key;**

**if ((ph = p.hash) > h) // 如果x节点的hash值小于p节点的hash值**

**dir = -1; // 则将dir赋值为-1, 代表向p的左边查找**

**else if (ph < h) // 与上面相反, 如果x节点的hash值大于p节点的hash值**

**dir = 1; // 则将dir赋值为1, 代表向p的右边查找**

**// 走到这代表x的hash值和p的hash值相等，则比较key值**

**else if ((kc == null && // 如果k没有实现Comparable接口 或者 x节点的key和p节点的key相等**

**(kc = comparableClassFor(k)) == null) ||**

**(dir = compareComparables(kc, k, pk)) == 0)**

**// 使用定义的一套规则来比较x节点和p节点的大小，用来决定向左还是向右查找**

**dir = tieBreakOrder(k, pk);**

**TreeNode<K,V> xp = p; // xp赋值为x的父节点,中间变量用于下面给x的父节点赋值**

**// dir<=0则向p左边查找,否则向p右边查找,如果为null,则代表该位置即为x的目标位置**

**if ((p = (dir <= 0) ? p.left : p.right) == null) {**

**x.parent = xp; // x的父节点即为最后一次遍历的p节点**

**if (dir <= 0) // 如果时dir <= 0, 则代表x节点为父节点的左节点**

**xp.left = x;**

**else // 如果时dir > 0, 则代表x节点为父节点的右节点**

**xp.right = x;**

**// 进行红黑树的插入平衡(通过左旋、右旋和改变节点颜色来保证当前树符合红黑树的要求)**

**root = balanceInsertion(root, x);**

**break;**

**}**

**}**

**}**

**}**

**moveRootToFront(tab, root); // 如果root节点不在table索引位置的头结点, 则将其调整为头结点**

**}**

1. 从调用此方法的节点作为起点，开始进行遍历，并将此节点设为root节点，标记为黑色（x.red = false）。
2. 如果当前节点不是根结点，则从根节点开始查找属于该节点的位置（该段代码跟之前的代码块2和代码块4的查找代码类似）。
3. 如果x节点(将要插入红黑树的节点)的hash值小于p节点(当前遍历到的红黑树节点)的hash值，则向p节点的左边查找
4. 与3相反，如果x节点的hash值大于p节点的hash值，则向p节点的右边查找。
5. 如果x的key没有实现Comparable接口，或者x节点的key和p节点的key相等，使用tieBreakOrder方法（见上文代码块5）来比较x节点和p节点的大小，以决定向左还是向右查找（dir <= 0向左，否则向右）。
6. 如果dir <= 0则向左节点查找（p赋值为p.left，并进行下一次循环），否则向右节点查找，如果已经无法继续查找（p赋值后为null），则代表该位置即为x的目标位置，另外变量xp用来记录最后一个节点，即为下文新增的x节点的父节点。
7. 将x的父节点设置为xp，根据dir的值决定x决定放在xp节点的左节点还是右节点，最后进行红黑树的插入平衡调整。
8. 调用moveRootToFront方法（见下文代码块8）将root节点调整到索引位置的头结点。

# ****resize方法****

1. final Node<K,V>[] resize() {
2. Node<K,V>[] oldTab = table;
3. int oldCap = (oldTab == null) ? 0 : oldTab.length;
4. int oldThr = threshold;
5. int newCap, newThr = 0;
6. if (oldCap > 0) { // 老table不为空
7. if (oldCap >= MAXIMUM\_CAPACITY) { // 老table的容量超过最大容量值
8. threshold = Integer.MAX\_VALUE; // 设置阈值为Integer.MAX\_VALUE
9. return oldTab;
10. }
11. // 将新容量赋值为老容量\*2，如果新容量<最大容量并且老容量>=16, 则将新阈值设置为原来的两倍
12. else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&
13. oldCap >= DEFAULT\_INITIAL\_CAPACITY)
14. newThr = oldThr << 1; // double threshold
15. }
16. else if (oldThr > 0) // 老表的容量为0, 老表的阈值大于0, 是因为初始容量被放入阈值
17. newCap = oldThr; // 则将新表的容量设置为老表的阈值
18. else { // 老表的容量为0, 老表的阈值为0, 则为空表，设置默认容量和阈值
19. newCap = DEFAULT\_INITIAL\_CAPACITY;
20. newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);
21. }
22. if (newThr == 0) { // 如果新阈值为空, 则通过新的容量\*负载因子获得新阈值
23. float ft = (float)newCap \* loadFactor;
24. newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?
25. (int)ft : Integer.MAX\_VALUE);
26. }
27. threshold = newThr; // 将当前阈值赋值为刚计算出来的新的阈值
28. @SuppressWarnings({"rawtypes","unchecked"})
29. // 定义新表,容量为刚计算出来的新容量
30. Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
31. table = newTab; // 将当前的表赋值为新定义的表
32. if (oldTab != null) { // 如果老表不为空, 则需遍历将节点赋值给新表
33. for (int j = 0; j < oldCap; ++j) {
34. Node<K,V> e;
35. if ((e = oldTab[j]) != null) { // 将索引值为j的老表头节点赋值给e
36. oldTab[j] = null; // 将老表的节点设置为空, 以便垃圾收集器回收空间
37. // 如果e.next为空, 则代表老表的该位置只有1个节点,
38. // 通过hash值计算新表的索引位置, 直接将该节点放在该位置
39. if (e.next == null)
40. newTab[e.hash & (newCap - 1)] = e;
41. else if (e instanceof TreeNode)
42. // 调用treeNode的hash分布(跟下面最后一个else的内容几乎相同)
43. ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
44. else { // preserve order
45. Node<K,V> loHead = null, loTail = null; // 存储跟原索引位置相同的节点
46. Node<K,V> hiHead = null, hiTail = null; // 存储索引位置为:原索引+oldCap的节点
47. Node<K,V> next;
48. do {
49. next = e.next;
50. //如果e的hash值与老表的容量进行与运算为0,则扩容后的索引位置跟老表的索引位置一样
51. if ((e.hash & oldCap) == 0) {
52. if (loTail == null) // 如果loTail为空, 代表该节点为第一个节点
53. loHead = e; // 则将loHead赋值为第一个节点
54. else
55. loTail.next = e; // 否则将节点添加在loTail后面
56. loTail = e; // 并将loTail赋值为新增的节点
57. }
58. //如果e的hash值与老表的容量进行与运算为1,则扩容后的索引位置为:老表的索引位置＋oldCap
59. else {
60. if (hiTail == null) // 如果hiTail为空, 代表该节点为第一个节点
61. hiHead = e; // 则将hiHead赋值为第一个节点
62. else
63. hiTail.next = e; // 否则将节点添加在hiTail后面
64. hiTail = e; // 并将hiTail赋值为新增的节点
65. }
66. } while ((e = next) != null);
67. if (loTail != null) {
68. loTail.next = null; // 最后一个节点的next设为空
69. newTab[j] = loHead; // 将原索引位置的节点设置为对应的头结点
70. }
71. if (hiTail != null) {
72. hiTail.next = null; // 最后一个节点的next设为空
73. newTab[j + oldCap] = hiHead; // 将索引位置为原索引+oldCap的节点设置为对应的头结点
74. }
75. }
76. }
77. }
78. }
79. return newTab;
80. }
81. 如果老表的容量大于0，判断老表的容量是否超过最大容量值：如果超过则将阈值设置为Integer.MAX\_VALUE，并直接返回老表（此时oldCap \* 2比Integer.MAX\_VALUE大，因此无法进行重新分布，只是单纯的将阈值扩容到最大）；将新表的容量赋值为老表的容量\*2，如果新容量小于最大容量并且老容量不小于16，则直接将新的阈值设置为原来的两倍。
82. 如果老表的容量为0，老表的阈值大于0，这种情况是传了容量的new方法创建的空表，将新表的容量设置为老表的阈值（这种情况发生在新创建的HashMap第一次put时，该HashMap初始化的时候传了初始容量，由于HashMap并没有capacity变量来存放容量值，因此传进来的初始容量是存放在threshold变量上（查看HashMap(int initialCapacity, float loadFactor)方法），因此此时老表的threshold的值就是我们要新创建的HashMap的capacity，所以将新表的容量设置为**老表的阈值**。
83. 如果老表的容量为0，老表的阈值为0，这种情况是没有传容量的new方法创建的空表，将阈值和容量设置为默认值。
84. 如果新表的阈值为空，则通过新的容量 \* 负载因子获得阈值（这种情况是初始化的时候传了初始容量，跟第2点相同情况，或者初始容量设置的太小导致老表的容量没有超过16导致的）。
85. 将当前阈值设置为刚计算出来的新的阈值，定义新表，容量为刚计算出来的新容量，将当前的表设置为新定义的表。
86. 如果老表不为空，则需遍历所有节点，将节点赋值给新表。
87. 将老表上索引为j的头结点赋值给e节点，并将老表上索引为j的节点设置为空。
88. 如果e的next节点为空，则代表老表的该位置只有1个节点，通过hash值计算新表的索引位置，直接将该节点放在新表的该位置上。
89. 如果e的next节点不为空，并且e为TreeNode，则调用split方法（见下文代码块10）进行hash分布。
90. 如果e的next节点不为空，并且e为普通的链表节点，则进行普通的hash分布
91. 如果e的hash值与老表的容量（为一串只有1个为2的二进制数，例如16为0000 0000 0001 0000）进行位与运算为0，则说明e节点扩容后的索引位置跟老表的索引位置一样（见例子1），进行链表拼接操作：如果loTail为空，代表该节点为第一个节点，则将loHead赋值为该节点；否则将节点添加在loTail后面，并将loTail赋值为新增的节点。
92. 如果e的hash值与老表的容量（为一串只有1个为2的二进制数，例如16为0000 0000 0001 0000）进行位与运算为1，则说明e节点扩容后的索引位置为：老表的索引位置＋oldCap（见例子1），进行链表拼接操作：如果hiTail为空，代表该节点为第一个节点，则将hiHead赋值为该节点；否则将节点添加在hiTail后面，并将hiTail赋值为新增的节点。
93. 老表节点重新hash分布在新表结束后，如果loTail不为空（说明老表的数据有分布到新表上原索引位置的节点），则将最后一个节点的next设为空，并将新表上原索引位置的节点设置为对应的头结点；如果hiTail不为空（说明老表的数据有分布到新表上原索引+oldCap位置的节点），则将最后一个节点的next设为空，并将新表上索引位置为原索引+oldCap的节点设置为对应的头结点。
94. 返回新表。

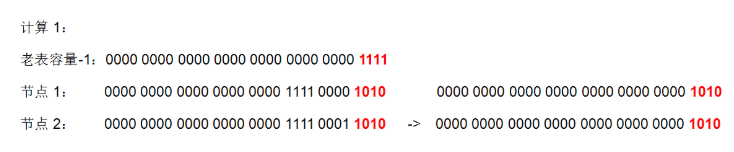
**问题1：resize()后，节点重hash为什么只可能分布在原索引位置与原索引+oldCap位置？**

扩容代码中，使用e节点的hash值跟oldCap进行位与运算，一次决定将节点分布到院索引位置或者原索引+oldCap位置上，why？

假设老表的容量位16，oldCap=16,则新表的容量位16\*2=32，假设节点1的hash值

为0000 0000 0000 0000 0000 1111 0000 1010

节点2的hash值为 0000 0000 0000 0000 0000 1111 0001 1010，则节点1和节点2在老表的索引位置计算如图

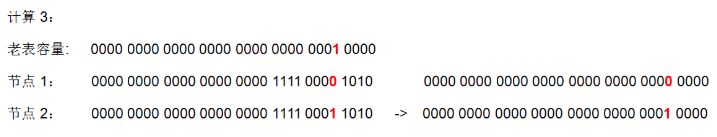


由于老表的长度限制，节点1和节点2的索引位置只取决于节点hash只的最后4位。

再看计算2



计算2位信标的索引计算，若果两个节点在老表的索引位置相同，则新表的索引位置只取决于节点hash值倒数第5位的值，而此位置的值刚好是老表的容量值16，此时节点在新表的索引位置只有两种情况：原索引位置和原索引位置+oldCap位置（此例子中即为10和10+16=26）.由于结果只取决于节点hash值的倒数第5位，而此位置的值刚好位老表的容量值16，所以此时新表的索引位置的计算可以替换为计算3



直接使用节点的hash值与老表的容量16进行位与运算，如果结果为0则该节点在新表的索引位置为原索引位置，否则该节点在新表的索引位置为原索引+oldCap位置

**关于红黑树的平衡调整？**

答：红黑树的操作涉及的操作比较复杂，三言两语无法说清。有兴趣的可以去单独学习，本文由于篇幅关系暂不详细介绍红黑树的具体操作，在这简单的介绍：红黑树是一种自平衡二叉树，拥有优秀的查询和插入/删除性能，广泛应用于关联数组。

对比AVL树，AVL要求每个结点的左右子树的高度之差的绝对值（平衡因子）最多为1，而红黑树通过适当的放低该条件（红黑树限制从根到叶子的最长的可能路径不多于最短的可能路径的两倍长，结果是这个树大致上是平衡的），以此来减少插入/删除时的平衡调整耗时，从而获取更好的性能，而这虽然会导致红黑树的查询会比AVL稍慢，但相比插入/删除时获取的时间，这个付出在大多数情况下显然是值得的。

在HashMap中的应用：HashMap在进行插入和删除时有可能会触发红黑树的插入平衡调整（balanceInsertion方法）或删除平衡调整（balanceDeletion ）方法，调整的方式主要有以下手段：左旋转（rotateLeft方法）、右旋转（rotateRight方法）、改变节点颜色（x.red = false、x.red = true），进行调整的原因是为了维持红黑树的数据结构。