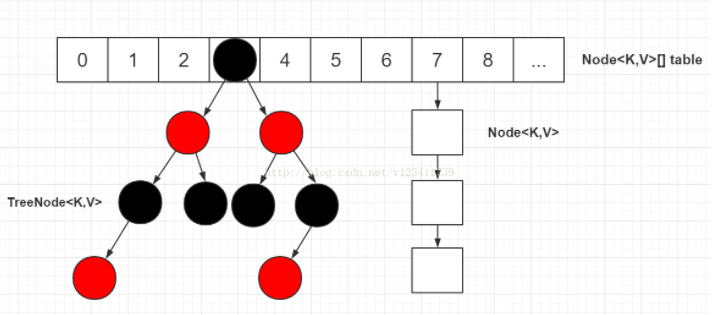
基于JDK1.8；

# 概述

Jdk1.8对hashmap进行了较大的优化，底层实现由之前的数组+链表，改为了数组+链表+红黑树，jdk1.8的hashmap的数据结构如下，当链表节点较少仍然以链表形式存在，当链表节点较多（大于8）会变为红黑树。



注意点：

①头结点指的是table表上索引位置的节点，就是链表头结点，即table数组上的元素存放头结点

②根节点，红黑树最上面的节点，就是没有父节点的节点

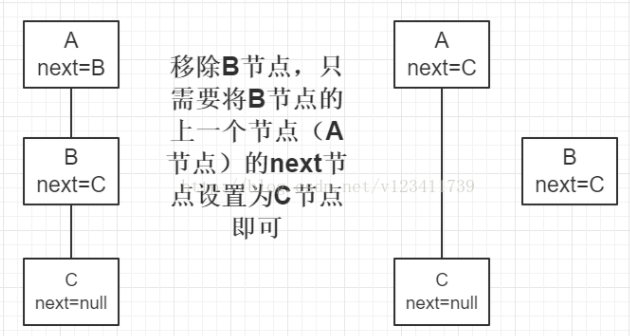
③红黑树的根节点不一定是索引位置的头结点，也就是不一定在table上

④转为红黑树后，链表的结构依然还在，通过next属性维持，红黑树节点进行操作时**都会维护链表的结构**

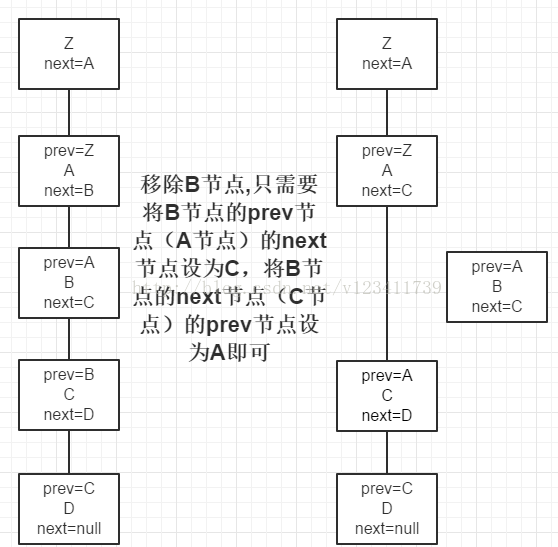
⑤红黑树上，叶子节点也可能有next节点，因为红黑树的结构跟链表的结构是互不影响的，不会因为是叶子节点就说该节点没有next节点，链表的结构依然存在

⑥源码中的一些变量的定义：如果定义了一个p，则pl表示p的做节点，pr表示p的右节点，pp表示p的父节点，ph表示p的hash值，pk表示p的key值，

⑦链表中移除一个节点只需如下操作



⑧红黑树在维护链表结构时，移除一个节点只需如下操作（红黑树中增加了prev属性）其他操作同理，此处只是维护链表操作，不包括红黑树的操作



⑨源码中进行红黑树的查找时，会反复用到一下两条规则，1.如果目标hash值小于p的hash值，则往p的左边遍历，否则往右遍历2.如果目标节点的key小于p的key，则向p的左边遍历；否则向p的右边遍历，这两天规则是利用红黑树的特性（二叉查找树）

⑩红黑树查找时，会用dir表示想左还是向右，dir存储的是目标节点的hash/key与p节点的hash/key的比较结果

引入红黑树原因：提高hashmap的性能，解决发生哈希值相同后，链表过长导致的索引效率低的问题，利用红黑树快速增删改查的特点将相关操作的时间复杂度从O（n）降到O（logn）

# 源码分析

## 基本属性的设定

|  |
| --- |
| **static** **final** **int** ***DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4; **默认容量16**  **static** **final** **int** ***MAXIMUM\_CAPACITY*** = 1 << 30;**最大容量**  **static** **final** **float** ***DEFAULT\_LOAD\_FACTOR*** = 0.75f;**默认负载0.75**  **static** **final** **int** ***TREEIFY\_THRESHOLD*** = 8;**链表节点转换红黑树节点的阈值，有可能转为红黑树，前提是满足大于*MIN\_TREEIFY\_CAPACITY***  **static** **final** **int** ***UNTREEIFY\_THRESHOLD*** = 6; **当扩容时，桶中元素个数小于这个值，就会把树形的桶元素 还原（切分）为链表结构**  **static** **final** **int** ***MIN\_TREEIFY\_CAPACITY*** = 64; **当哈希表中的容量大于这个值时，表中的桶才能进行树形化**  **transient** Node<K,V>[] **table;哈希桶数组，长度总是2的n次方**  **transient** Set<Map.Entry<K,V>> entrySet；**有hashmap中node节点构成的set集合**  **transient** **int** size;**存放元素的个数**  **transient** **int** modCount;**扩容更改map结构计数器**  **int** threshold;**实际size超过阈值时，进行扩容**  **final** **float** loadFactor;**装载因子** |

## Node节点定义

|  |
| --- |
| **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;  }  **public** **final** K getKey() { **return** key; }  **public** **final** V getValue() { **return** value; }  **public** **final** String toString() { **return** key + "=" + value; }  **public** **final** **int** hashCode() {  **return** Objects.*hashCode*(key) ^ Objects.*hashCode*(value);  }  **public** **final** V setValue(V newValue) {  V oldValue = value;  value = newValue;  **return** oldValue;  }  **Node的判断相等，首先判断==即引用地址判断，一样返回true，否则观察是否是map.entry类型，不是返回fasle，如果是，转换为map。Entry，惊醒key和value的对比，如果一样，返回true**  **public** **final** **boolean** equals(Object o) {  **if** (o == **this**)  **return** **true**;  **if** (o **instanceof** Map.Entry) {  Map.Entry<?,?> e = (Map.Entry<?,?>)o;  **if** (Objects.*equals*(key, e.getKey()) &&  Objects.*equals*(value, e.getValue()))  **return** **true**;  }  **return** **false**;  }  } |

## TreeNode

|  |
| --- |
| **static** **final** **class** TreeNode<K,V> **extends** LinkedHashMap.Entry<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) {  **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;  }  } |

## 定位哈希桶数组索引的位置：

|  |
| --- |
| **确定hash值**  **static** **final** **int** hash(Object key) {  **int** h;  **return** (key == **null**) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  }  // 代码2将hash值与数组长度按位求与  int n = tab.length;  // 将(tab.length - 1) 与 hash值进行&运算  int index = (n - 1) & hash; |
| **本质上三步：**  **①拿到key的hashcode**  **②hashcode高位参与运算，重新计算hash值**  **③将计算的hash值与（table.length-1）进行&运算**  **X mod 2^n = x&(2^n -1)表示x是否能被2^n整除，**  **而hashmap底层数组长度总是2的n次方，原因如上，并且取模运算为“h mod table.length”，对应上面的公式，可以得到该运算等同于“h & (table.length - 1)”。这是HashMap在速度上的优化，因为&比%具有更高的效率。**  **另外优化高位运算，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，并且不会有太大的开销。** |

## 比较类型以及相同类型比较大小

|  |
| --- |
| **如果x实现了Comparable接口，就返回c的类型，否则返回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**;  }  **如果x不为空，且类型和k的类型一样，返回k与x的比较结果**  **static** **int** compareComparables(Class<?> kc, Object k, Object x) {  **return** (x == **null** || x.getClass() != kc ? 0 :  ((Comparable)k).compareTo(x));  } |

## Table的容量设置，这里就体现了长度为2的n次方

|  |
| --- |
| **返回一个比给定整数大且最接近该数的2的幂次方整数**  **static** **final** **int** tableSizeFor(**int** cap) {  **int** n = cap - 1;  n |= n >>> 1;  n |= n >>> 2;  n |= n >>> 4;  n |= n >>> 8;  n |= n >>> 16;  **return** (n < 0) ? 1 : (n >= ***MAXIMUM\_CAPACITY***) ? ***MAXIMUM\_CAPACITY*** : n + 1;  }  **只找最高位，最高位到第零位之间最多全1，不会再多出来了**  为什么cap-1  这是为了防止,cap已经是2的幂。如果cap已经是2的幂, 又没有执行这个减1操作,则执行完后面的几条无符号右移操作之后,返回的capacity将是这个cap的2倍。  如果n这时为0了(经过了cap-1之后),则经过后面的几次无符号右移依然是0,最后返回的capacity是1(最后有个n+1的操作)。以下只讨论n不等于0的情况。  第一次右移  n |= n >>> 1;  由于n不等于0,则n的二进制表示中总会有一bit为1,这时考虑最高位的1。通过无符号右移1位,则将最高位的1右移了1位,再做或操作,使得n的二进制表示中与最高位的1紧邻的右边一位也为1,如000011xxxxxx。  第二次右移  n |= n >>> 2;  这个n已经经过了n |= n >>> 1;操作。假设此时n为000011xxxxxx ,则n无符号右移两位,会将最高位两个连续的1右移两位,然后再与原来的n做或操作,这样n的二进制表示的高位中会有4个连续的1。如00001111xxxxxx 。  第三次右移  n |= n >>> 4;  这次把已经有的高位中的连续的4个1,右移4位,再做或操作,这样n的二进制表示的高位中会有8个连续的1。如00001111 1111xxxxxx 。  以此类推  注意,容量最大也就是32bit的正数,因此最后n |= n >>> 16;,最多也就32个1。但是这时已经大于了MAXIMUM\_CAPACITY,因为:  return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;  所以取值到MAXIMUM\_CAPACITY |

## Hashmap的构造函数

|  |
| --- |
| **指定初始容量及装载因子**  **public** HashMap(**int** initialCapacity, **float** loadFactor) {  初始容量小于0抛异常  **if** (initialCapacity < 0)  **throw** **new** IllegalArgumentException("Illegal initial capacity: " + initialCapacity);  初始容量大于最大容量，则为最大容量  **if** (initialCapacity > ***MAXIMUM\_CAPACITY***)  initialCapacity = ***MAXIMUM\_CAPACITY***;  装载因子不能小于等于0或者非浮点型数字，否则抛异常  **if** (loadFactor <= 0 || Float.*isNaN*(loadFactor))  **throw** **new** IllegalArgumentException("Illegal load factor: " + loadFactor);  **初始化装载因子**  **this**.loadFactor = loadFactor;  **初始化阈值**  **this**.threshold = *tableSizeFor*(initialCapacity);  }  **调用hashmap（int ，float）构造函数**  **public** HashMap(**int** initialCapacity) {  **this**(initialCapacity, ***DEFAULT\_LOAD\_FACTOR***);  }  **//所有属性都采用默认值**  **public** HashMap() {  **this**.loadFactor = ***DEFAULT\_LOAD\_FACTOR***; // all other fields defaulted  }  **将map放入hashmap中，装载因子依然是默认，将m中所有元素调用putMapEntry函数，后面进行介绍**  **public** HashMap(Map<? **extends** K, ? **extends** V> m) {  **this**.loadFactor = ***DEFAULT\_LOAD\_FACTOR***;  putMapEntries(m, **false**);  } |

# Hashmap重要函数

## Put函数及相关方法

|  |
| --- |
| 将指定节点key，value放入hashmap中，实际调用putval  **public** V put(K key, V value) {  **return** putVal(*hash*(key), key, value, **false**, **true**);  }  **将所有map集合放入map中，调用putmapentries**  **public** **void** putAll(Map<? **extends** K, ? **extends** V> m) {  putMapEntries(m, **true**);  }  **所有集合元素放入hashmap中**  **final** **void** putMapEntries(Map<? **extends** K, ? **extends** V> m, **boolean** evict) {  **int** s = m.size();  **if** (s > 0) {  **if** (table == **null**) { **判断table是否初始化**  根据带插入的map的size，loadFactor计算hashmap的阈值容量  **float** ft = ((**float**)s / loadFactor) + 1.0F;  **int** t = ((ft < (**float**)***MAXIMUM\_CAPACITY***) ?  (**int**)ft : ***MAXIMUM\_CAPACITY***);  如果t比阈值大，更新阈值  **if** (t > threshold)  threshold = *tableSizeFor*(t);  }  **如果带插入的map的size大于阈值容量，先进性resize**  **else** **if** (s > threshold)  resize();  **循环进行插入，核心还是putval，下面讲**  **for** (Map.Entry<? **extends** K, ? **extends** V> e : m.entrySet()) {  K key = e.getKey();  V value = e.getValue();  putVal(*hash*(key), key, value, **false**, evict);  }  }  }  **此方法调用了resize，tablesizefor，hash方法**  **上面的put方法实际上都是调用该函数进行处理的**  onlyIfAbsent为true表示不修改节点值  evict为false表明，在创建的时候调用这个函数，即构造器中使用的  **final** V putVal(**int** hash, K key, V value, **boolean** onlyIfAbsent,  **boolean** evict) {  Node<K,V>[] tab; Node<K,V> p; **int** n, i;  **①若table未初始化或者长度为0，进行resize扩容并把长度设为n**  **if** ((tab = table) == **null** || (n = tab.length) == 0)  n = (tab = resize()).length;  **②根据hash值与n-1的与结果，确定在table中的插入位置，下标设为i，节点设为p，**  **------如果该p==null，进性插入操作**  **if** ((p = tab[i = (n - 1) & hash]) == **null**)  tab[i] = newNode(hash, key, value, **null**);  **------如果不为null，说明tab[i]不为空，有元素，进行以下操作，考虑链表和红黑树**  **else** {  Node<K,V> e; K k;  -----------**与table中的首元素进行比较，如果hash，key都相等，说明带插入和第一个元素相等，直接赋值**  **if** (p.hash == hash &&  ((k = p.key) == key || (key != **null** && key.equals(k))))  --------------**相等直接赋值给e**  e = p;  --------------**否则判断是否是红黑树节点类型，如果是，按照红黑树进行插入操作，返回的是树中与插入节点相同的节点**  **else** **if** (p **instanceof** TreeNode)  e = ((TreeNode<K,V>)p).putTreeVal(**this**, tab, hash, key, value);  ---------------**不是红黑树类型，说明是链表类型**  **else** {  -----------------**在链表最末端插入元素**  **for** (**int** binCount = 0; ; ++binCount) {  --------------------**遍历到最末端，直到p的next为null，并将其赋值给e，进行插入操作**  **if** ((e = p.next) == **null**) {  p.next = newNode(hash, key, value, **null**);  --------------------**如果数量达到树形化的阈值，就转为红黑树**  **if** (binCount >= ***TREEIFY\_THRESHOLD*** - 1) **// -1 for 1st**  treeifyBin(tab, hash);  **break**;  }  --------------------**如果hash，key都相等，是，不进行插入**  **if** (e.hash == hash &&  ((k = e.key) == key || (key != **null** && key.equals(k))))  **break**;  --------------------与 e = p.next结合，表示指向下一个节点，用于遍历到表尾  p = e;  }  }  -----------**表示table、链表、树中已存在与插入吃hash、key相等的节点**  **if** (e != **null**) { // existing mapping for key  -------------记录表中节点的value值  V oldValue = e.value;  --------------如果onlyIfAbsent**为false或者oldvalue=null，就进行覆盖值**  **if** (!onlyIfAbsent || oldValue == **null**)  e.value = value;  --------------插入后进行回调，空函数，用户根据需要覆盖  afterNodeAccess(e);  **return** oldValue;  }  }  ++modCount;  -----------**插入后是否超过阈值，如果是调用resize**  **if** (++size > threshold)  resize();  --------------插入后进行回调，空函数，用户根据需要覆盖  afterNodeInsertion(evict);  **return** **null**;  } |
| **红黑树的插入，同时维护链表属性，即原来的next属性**  **返回值表示找到了key hash值都相等的节点**  **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，索引位置的头结点不一定红黑树的根节点，如果当前节点的父亲不为空，root函数一层一层往上，直到找到根节点，当前节点父亲为空，则当前节点就是根节点**  TreeNode<K,V> root = (parent != **null**) ? root() : **this**;  **根节点赋值为p，开始遍历**  **for** (TreeNode<K,V> p = root;;) {  **int** dir, ph; K pk;  **---传入的hash值小于p的hash值，dir=-1，表明向左**  **if** ((ph = p.hash) > h)  dir = -1;  **---传入的hash值小于p的hash值，dir=-1，表明向右**  **else** **if** (ph < h)  dir = 1;  ---**如果相等，比较key，此时如果p的key与传入的key相等，即p为目标节点，返回**  **else** **if** ((pk = p.key) == k || (k != **null** && k.equals(pk)))  **return** p;  **------------如果hash相等，key不等，并且k没有实现comparable即不可比较的，或者pk和kc不是同个类型等条件**  **else** **if** ((kc == **null** &&  (kc = *comparableClassFor*(k)) == **null**) ||  (dir = *compareComparables*(kc, k, pk)) == 0) {  -------------------**只会执行一次，在searched为false的时候，以当前节点为根的整个树上搜索是否存在带插入节点**  **if** (!searched) {  TreeNode<K,V> q, ch;  searched = **true**;  -------------------**从p节点的左节点和右节点调用find进行查找，如果查找到目标节点就返回目标节点**  **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大小，用来决定向左向右查找，小于零向左，大于零向右**  dir = *tieBreakOrder*(k, pk);  }  ------------**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**  TreeNode<K,V> xp = p;  ---------------**根据dir，对p进行左右子树的查找**  **if** ((p = (dir <= 0) ? p.left : p.right) == **null**) {  ---------------**能进来表示已经找到了带插入节点的位置x，xp为带插入的节点的父节点，注意treenode节点中既存在树状关系，也存在链表关系，**  Node<K,V> xpn = xp.next;**得到xp的next节点**  **----------------创建新节点，xpn为x的next节点，将x插入到xp与xpn之间**  TreeNode<K,V> x = map.newTreeNode(h, k, v, xpn);  **if** (dir <= 0)**表明x是xp左孩子**  xp.left = x;  **else 表明x是xp的右孩子**  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xp**  **-----------------如果xpn不为空，就将其prev 设为x**  **if** (xpn != **null**)  ((TreeNode<K,V>)xpn).prev = x;  ----------------**插入后进行平衡调整**  *moveRootToFront*(tab, *balanceInsertion*(root, x));  **return** **null**;  }  }  }  **①查找当前红黑树的根节点，将根节点赋值为p，开始查找**  **②如果传入的hash小于p的hash，赋值dir为-1，代表p的左边查找树，如果大于p的hash，将dir赋值为1，查找p的右边，如果等于，并且传入的key与p的key相等，为目标节点，返回p**  **③如果k所属的类没有实现comparable接口，或者k，p节点key比较后相等：第一次会从p的左右节点分别调用find方法，如果找到目标节点就返回；如果不是第一次或者找不到目标节点，就不执行，调用tieBreakORder**  **④如果dir<= 0,向左节点查找，否则向右节点查找，如果无法继续查找，即p==null，说明该位置p节点的上一个访问的节点xp就是要插入的节点的父节点，即新增的x节点的父节点。**  **⑤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节点可能是叶子节点，没有左节点的节点，没有右节点的节点，不可能左右都有，不然还能继续找下去，即使是叶子节点，也会有next节点，红黑树和链表结构互不影响，不会因为是叶子节点，就没有next，next属性用来维护链表结构的），根据dir的值决定x放在xp节点的左右节点，将xp的next改为x，x的parent和prev设为xp，如果xp的next不为空，就让xpn的prev设置为x。**  **⑥平衡调整（见下文）**  **用于不可比较或者hashcode相同时进行比较的方法，只是一个插入规则**  **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;  }  ------------------------------------------------------------------  **把容器里的元素变为树结构，当链表长度大于等于treeifythreshold，就会将容器里的元素变为树结构** |
| **final** **void** treeifyBin(Node<K,V>[] tab, **int** hash) {  **int** n, index; Node<K,V> e;  **如果元素数组为空或者数组长度小于最小树形化限制，就不转换，进行扩容**  **if** (tab == **null** || (n = tab.length) < ***MIN\_TREEIFY\_CAPACITY***)  resize();  **如果元素数组长度大于等于最小树形化限制，进行结构转换**  **先找到当前节点的所在table的索引位置，得到头结点，**  **else** **if** ((e = tab[index = (n - 1) & hash]) != **null**) {  **如果头结点不为空进行以下步骤**  TreeNode<K,V> hd = **null**, tl = **null**;**定义首尾节点head，tail**  **do** {  ----------**将该节点转换为树节点**  TreeNode<K,V> p = replacementTreeNode(e, **null**);  -------------**如果尾节点为空，说明刚建立，还没根节点，将首节点（根节点）指向p**  **if** (tl == **null**)  hd = p;  **else** {**否则在尾节点后面插入p**  p.prev = tl;  tl.next = p;  }  tl = p;**将p设为新的尾节点**  } **while** ((e = e.next) != **null**);直到**表中的链表节点，都转化为树节点**  **----------到目前为止，只是把node变为treenode，把单向链表变为双向链表**  **if** ((tab[index] = hd) != **null**)**把转换的双向链表替换原来的单链表**  **接下来才是转换树的操作**  hd.treeify(tab);  }  }  ---------------------------------------------------------  **将链表转换为红黑树**  **final** **void** treeify(Node<K,V>[] tab) {  TreeNode<K,V> root = **null**;  **遍历链表，x指向当前节点，next指向下一节点**  **for** (TreeNode<K,V> x = **this**, next; x != **null**; x = next) {  next = (TreeNode<K,V>)x.next;**指向下一个节点**  x.left = x.right = **null**;设置x的左右孩子为空  **if** (root == **null**) {**如果没有根节点，x的父亲为空，x颜色为黑色，将x设为根节点**  x.parent = **null**;  x.red = **false**;  root = x;  }  **如果已经存在根节点**  **else** {  K k = x.key;**取得当前链表的节点的key**  **int** h = x.hash;**取得节点的hash**  Class<?> kc = **null**;  **----------如果当前节点不是根节点，则从根节点开始找到该节点的位置，根据hash值**  **for** (TreeNode<K,V> p = root;;) {  **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)  ------------------------如果没有实现comparable接口，或者x节点的key和pk的k相等，则使用定义的规则来比较x节点和p节点的大小。  dir = *tieBreakOrder*(k, pk);  **将p节点赋值给xp，表明xp为p节点的父亲，即上一个访问的节点，用于标注插入位置为xp的后面**  TreeNode<K,V> xp = p;  **if** ((p = (dir <= 0) ? p.left : p.right) == **null**) {  -------------------说明已经找到了在哪里插入，就是在xp后面插入  x.parent = xp;  **if** (dir <= 0)  xp.left = x;  **else**  xp.right = x;  ------------------------插入平衡调整  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**  }  总结：  **①从调用此方法的节点作为起点，开始遍历，如果此时root不存在，将此节点设为root节点标记为黑色**  **②如果当前节点不是根节点，则从根节点开始查找属于该节点的位置**  **③如果x的hash小于p的hash，从p的左边查找，否则从p的右边查找**  **④如果x的key没有实现comparable，或者x的key和p的key相等，使用tiebreakorder方法，得到dir的值**  **⑤如果dir小于0，向左查找p=p.left，否则向右查找，如果无法继续，说明p的位置就是x的位置，xp就是x的父亲**  **⑥根据dir的值决定放左右节点。**  **⑦调用moveRootToFront方法将root调整到索引位置的头结点** |
| **如果当前索引位置的头结点不是root，则将root的上一个节点和下一个节点进行关联，将root放到头结点的位置，原头结点放到root的next节点**  **static** <K,V> **void** moveRootToFront(Node<K,V>[] tab, TreeNode<K,V> root) {  **int** n;  **如果root不为空，tab不为空，且tab长度大于0，执行以下操作**  **if** (root != **null** && tab != **null** && (n = tab.length) > 0) {  ------找到root节点所在的table的索引的位置，index  **int** index = (n - 1) & root.hash;  ------将root索引位置对应的tab的头结点强转为TreeNode，赋值为first  TreeNode<K,V> first = (TreeNode<K,V>)tab[index];  ------如果root不等于头结点first，执行以下操作  **if** (root != first) {  -------------将root设为tab中的头结点，然后将root原本的前后节点关联起来，rn.prev=rp;rp.next = rn;,如果first不为空，将first的prev执行root，root的next指向prev  Node<K,V> rn;  tab[index] = root;  TreeNode<K,V> rp = root.prev;  **if** ((rn = root.next) != **null**)  ((TreeNode<K,V>)rn).prev = rp;  **if** (rp != **null**)  rp.next = rn;  **if** (first != **null**)  first.prev = root;  root.next = first;  root.prev = **null**;  }  --------------检查树是否正常  **assert** *checkInvariants*(root);  }  } |

# Get方法

|  |
| --- |
| **通过key获取value，如果找不到，返回null**  **public** V get(Object key) {  Node<K,V> e;  **return** (e = getNode(*hash*(key), key)) == **null** ? **null** : e.value;  }  核心方法通过hash值和key值得到节点值  **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，长度大于零，hash对应的索引位置头结点不为空，执行以下操作**  **if** ((tab = table) != **null** && (n = tab.length) > 0 &&  (first = tab[(n - 1) & hash]) != **null**) {  **如果头结点first的hash和（key相等或者key不为null且key.equals为true，则表明头结点就是要找的目标节点**  **if** (first.hash == hash && // always check first node  ((k = first.key) == key || (key != **null** && key.equals(k))))  **return** first;  **如果不是，向下遍历**  **if** ((e = first.next) != **null**) {  **判断是否是treenode，是的话，调用getTreeNode，后面分析**  **if** (first **instanceof** TreeNode)  **return** ((TreeNode<K,V>)first).getTreeNode(hash, key);  **do** {  **不是treeNode就是链表形式，找到hash相等并且key相等或者key的equal也相等的结点，返回；**  **if** (e.hash == hash &&  ((k = e.key) == key || (key != **null** && key.equals(k))))  **return** e;  } **while** ((e = e.next) != **null**);  }  }  **return** **null**;**找不到就返回null**  }  **实际调用find方法，如果不是根节点，就找到根节点，是的话，直接调用find找**  **final** TreeNode<K,V> getTreeNode(**int** h, Object k) {  **return** ((parent != **null**) ? root() : **this**).find(h, k, **null**);  }  **哪个结点调用的，this就是谁，从this开始找**  **final** TreeNode<K,V> find(**int** h, Object k, Class<?> kc) {  TreeNode<K,V> p = **this**;  **do** {  **int** ph, dir; K pk;  TreeNode<K,V> pl = p.left, pr = p.right, q;  **if** ((ph = p.hash) > h)  p = pl;  **else** **if** (ph < h)  p = pr;  -------------**p的hash值ph小于h向左，否则向右**  **else** **if** ((pk = p.key) == k || (k != **null** && k.equals(pk)))---**此处表明hash值相等，只要key==或者equals为真就返回p**  **return** p;  **else** **if** (pl == **null**)  p = pr;**如果p左节点为空，从右遍历**  **else** **if** (pr == **null**)  p = pl;**右节点为空，从左遍历**  **else** **if** ((kc != **null** ||  (kc = *comparableClassFor*(k)) != **null**) &&  (dir = *compareComparables*(kc, k, pk)) != 0)  ---------------**如果kc不为空或者（k实现了comparable，进行k与pk的比较），如果dir小于0，p=pl，否则p=pr**  p = (dir < 0) ? pl : pr;  --------------**该情况表明key没有实现comparable接口，直接执行右边遍历**  **else** **if** ((q = pr.find(h, k, kc)) != **null**)  **return** q;  -------------**该情况表明上一个右边遍历没有找到，直接向左遍历**  **else**  p = pl;  } **while** (p != **null**);  **return** **null**;  } |

# Remove方法

|  |
| --- |
| **public** V remove(Object key) {  Node<K,V> e;  **return** (e = removeNode(*hash*(key), key, **null**, **false**, **true**)) == **null** ?  **null** : e.value;  }  **public** **boolean** remove(Object key, Object value) {  **return** removeNode(*hash*(key), key, value, **true**, **true**) != **null**;  }  底层都是调用removeNode方法进行删除操作  **matchValue 如果为true，则当key对应的键值对的值equals(value)为true时才删除；否则不关心value的值**  **movable 删除后是否移动节点，如果为false，则不移动**  **final** Node<K,V> removeNode(**int** hash, Object key, Object value,  **boolean** matchValue, **boolean** movable) {  Node<K,V>[] tab; Node<K,V> p; **int** n, index;  **Table不为空，长度大于零，hash对应的索引位置头结点不为空，p指向头结点，执行以下操作**  **if** ((tab = table) != **null** && (n = tab.length) > 0 &&  (p = tab[index = (n - 1) & hash]) != **null**) {  Node<K,V> node = **null**, e; K k; V v;  **P的hash值与参数中的hash相等，key==成立或者equals成立，将node指向p**  **if** (p.hash == hash &&  ((k = p.key) == key || (key != **null** && key.equals(k))))  node = p;  **else** **if** ((e = p.next) != **null**) {  **如果不相等，e指向p的nex，如果不为null，判断p是不是treenode，是的话按照getTreeNode进行查找**  **if** (p **instanceof** TreeNode)  node = ((TreeNode<K,V>)p).getTreeNode(hash, key);  **否则按照链表方式进行查找**  **else** {  **do** {  ------------------**向下遍历直到找到hash和key相等的，赋值为node**  **if** (e.hash == hash &&  ((k = e.key) == key ||  (key != **null** && key.equals(k)))) {  node = e;  **break**;  }  p = e;  } **while** ((e = e.next) != **null**);  }  }  --**如果node不为空，说明根据key找到了要删除的节点，matchvalue为false表明不需要对比值或者需要对比value，并value值相等，可以执行删除**  **if** (node != **null** && (!matchValue || (v = node.value) == value ||  (value != **null** && value.equals(v)))) {  ------------**如果是treenode，就用removeTreeNode删除**  **if** (node **instanceof** TreeNode)  ((TreeNode<K,V>)node).removeTreeNode(**this**, tab, movable);  **else** **if** (node == p)**node指向头结点，直接头结点指向node的next**  tab[index] = node.next;  **else如果不是头结点，此时p表示node的父节点，因此将p的next指向node的next**  p.next = node.next;  ++modCount;  --size;  afterNodeRemoval(node);  **return** node;  }  }  **return** **null**;  }  **final** **void** removeTreeNode(HashMap<K,V> map, Node<K,V>[] tab,  **boolean** movable) {  **int** n;  **if** (tab == **null** || (n = tab.length) == 0)  **return**;**table为空或length为0直接结束**  **----------此处的hash为node的hash值，找到node对应的头结点**  **int** index = (n - 1) & hash;  ----------**头结点赋值为first，node的next赋值为succ，node的prev赋值为pred**  TreeNode<K,V> first = (TreeNode<K,V>)tab[index], root = first, rl;  TreeNode<K,V> succ = (TreeNode<K,V>)next, pred = prev;  ---------**如果pred为null说明，node为头结点，让头结点和first指向succ即可**  **if** (pred == **null**)  tab[index] = first = succ;  **else否则pred的next指向succ**  pred.next = succ;  **if** (succ != **null**)如果succ不等于null，则succ的prev指向pred  succ.prev = pred;  **if** (first == **null**)**如果此处first为空，表明没有结点了，结束**  **return**;  ----------**如果root父亲不为空，说明头结点不是root，则将root赋值为根节点**  **if** (root.parent != **null**)  root = root.root();  ----------**通过root来判断红黑树是否太小，如果是，调用untreeify方法转为链表返回（转回链表后，无须进行下面的代码，直接return）**  **if** (root == **null** || root.right == **null** ||  (rl = root.left) == **null** || rl.left == **null**) {  tab[index] = first.untreeify(map); // too small  **return**;  }  **链表处理到这里就结束了**  ----------------------------------------------------------------  **This是要被删除的节点node；p赋值为node，pl为node左孩子，pr为node的右孩子**  TreeNode<K,V> p = **this**, pl = left, pr = right, replacement;  ----------**删除节点有左右孩子，此处采用找右子树最小，即右子树的最左**  **if** (pl != **null** && pr != **null**) {  TreeNode<K,V> s = pr, sl;  **while** ((sl = s.left) != **null**) // find successor  s = sl;  --------------**交换s，p的颜色**  **boolean** c = s.red; s.red = p.red; p.red = c; // swap colors  -------------第一次第二次调整所有代码都是为了p与s节点的调换，第三次为了找出replacement覆盖掉p  TreeNode<K,V> sr = s.right;  TreeNode<K,V> pp = p.parent;  --------------**第一次调整**  **if** (s == pr) { **p的右孩子就是s，且为叶子节点，或者s没有左孩子**  p.parent = s;  s.right = p;  }  **else** {**s不是p的直接孩子**  TreeNode<K,V> sp = s.parent;  **if** ((p.parent = sp) != **null**) {  **if** (s == sp.left)  sp.left = p;  **else**  sp.right = p;  }  **if** ((s.right = pr) != **null**)  pr.parent = s;  }    -------------**第二次调整**  p.left = **null**;  **if** ((p.right = sr) != **null**)  sr.parent = p;  **if** ((s.left = pl) != **null**)  pl.parent = s;  **if** ((s.parent = pp) == **null**)  root = s;  **else** **if** (p == pp.left)  pp.left = s;  **else**  pp.right = s;  **if** (sr != **null**)  replacement = sr;  **else**  replacement = p;  }  **else** **if** (pl != **null**)  replacement = pl;  **else** **if** (pr != **null**)  replacement = pr;  **else**  replacement = p;    ----------**第三次调整**  **if** (replacement != p) {  TreeNode<K,V> pp = replacement.parent = p.parent;  **if** (pp == **null**)  root = replacement;  **else** **if** (p == pp.left)  pp.left = replacement;  **else**  pp.right = replacement;  p.left = p.right = p.parent = **null**;  }    **上面的步骤等价于将s，p调换位置，然后删除p，为什么不考虑将s拷贝到p，然后删除s，这是个疑惑**  -------------------------------------------------------------------  **只有黑色才需要调整平衡**  TreeNode<K,V> r = p.red ? root : *balanceDeletion*(root, replacement);  -----------**如果p是叶子节点，只需要简单将p移除，如果p的parent不为空，p是左孩子，那么p的parent的左孩子为空，反之右孩子为空**  **if** (replacement == p) { // detach  TreeNode<K,V> pp = p.parent;  p.parent = **null**;  **if** (pp != **null**) {  **if** (p == pp.left)  pp.left = **null**;  **else** **if** (p == pp.right)  pp.right = **null**;  }  }  **if** (movable)  *moveRootToFront*(tab, r);  }  **解释1：为什么sr是replacement的首选，p为备选？**  解析：首先我们看sr是什么？从代码中可以看到sr第一次被赋值时，是在s节点进行了向左穷遍历结束后，因此此时s节点是没有左节点的，sr即为s节点的右节点。而从上面的三次调整我们知道，p节点已经跟s节点进行了位置调换，所以此时sr其实是p节点的右节点，并且p节点没有左节点，因此要移除p节点，只需要将p节点的右节点sr覆盖掉p节点即可，因此sr是replacement的首选，如果sr为空，则代表p节点为叶子节点，此时将p节点清空即可。 |

# Replace函数

|  |
| --- |
| **根据key和旧的value，查找匹配节点，进行替换**  @Override  **public** **boolean** replace(K key, V oldValue, V newValue) {  Node<K,V> e; V v;  **if** ((e = getNode(*hash*(key), key)) != **null** &&  ((v = e.value) == oldValue || (v != **null** && v.equals(oldValue)))) {  e.value = newValue;  afterNodeAccess(e);  **return** **true**;  }  **return** **false**;  }  **根据key和key的hash值，调用getNode，得到节点e，比较value与oldvalue是否相等，相等就用新值替换旧值**  **根据key查询匹配进行替换value**  @Override  **public** V replace(K key, V value) {  Node<K,V> e;  **if** ((e = getNode(*hash*(key), key)) != **null**) {  V oldValue = e.value;  e.value = value;  afterNodeAccess(e);  **return** oldValue;  }  **return** **null**;  } |

# 扩容函数resize

|  |
| --- |
| **final** Node<K,V>[] resize() {  Node<K,V>[] oldTab = table;**保存当前table**  **------保存当前table的容量**  **int** oldCap = (oldTab == **null**) ? 0 : oldTab.length;  **int** oldThr = threshold;**保存当前阈值**  **int** newCap, newThr = 0;**初始化新的容量和阈值**  **--------第一种：resize在size>threshold时被调用，原来表非空**  **oldThr=oldcap\*loadfactor**  **if** (oldCap > 0) {  ---------**若旧的容量已经超过了最大容量，就将阈值设定为**Integer.***MAX\_VALUE，返回***  **if** (oldCap >= ***MAXIMUM\_CAPACITY***) {  threshold = Integer.***MAX\_VALUE***;  **return** oldTab;  }  ----------**若容量翻倍小于最大容量，并且旧容量大于等于默认初始容量16，扩容两倍**  **else** **if** ((newCap = oldCap << 1) < ***MAXIMUM\_CAPACITY*** &&  oldCap >= ***DEFAULT\_INITIAL\_CAPACITY***)  newThr = oldThr << 1; // double threshold  }  -------**第二种：此处在table为空时调用，oldcap小于等于0，且oldTHr大于零，表示**  **用户创建了HashMap(int initialCapacity, float loadFactor) 或 HashMap(int initialCapacity) 或 HashMap(Map<? extends K, ? extends V> m)导致oldtab为空，oldcap为0，oldthr为用户指定的值**  **else** **if** (oldThr > 0) // initial capacity was placed in threshold  newCap = oldThr;  -------**第三种函数在table为空时被调用，oldcap小于等于0，oldthr等于0，用户**  **调用hashmap（）构造函数，所有参数采用默认值oldcap=0，oldtab=null，oldthr=0**  **else** {  newCap = ***DEFAULT\_INITIAL\_CAPACITY***;**16**  newThr = (**int**)(***DEFAULT\_LOAD\_FACTOR*** \* ***DEFAULT\_INITIAL\_CAPACITY***);**16\*0.75**  }  --------**新阈值为0，表明第二种，此时使用newcap\*loadfactor**  **if** (newThr == 0) {  **float** ft = (**float**)newCap \* loadFactor;  newThr = (newCap < ***MAXIMUM\_CAPACITY*** && ft < (**float**)***MAXIMUM\_CAPACITY*** ?  (**int**)ft : Integer.***MAX\_VALUE***);  }  -------变化阈值为新阈值  threshold = newThr;  @SuppressWarnings({"rawtypes","unchecked"})  ----------初始化table  Node<K,V>[] newTab = (Node<K,V>[])**new** Node[newCap];  table = newTab;  **if** (oldTab != **null**) {  -------**把oldtab的节点，rehash 到newTab**  **for** (**int** j = 0; j < oldCap; ++j) {  Node<K,V> e;  **if** ((e = oldTab[j]) != **null**) {  oldTab[j] = **null**;  ---------------**如果节点是单个节点，直接重新计算index，并赋值**  **if** (e.next == **null**)  newTab[e.hash & (newCap - 1)] = e;  ------------------**如果第treenode，要进行红黑树的rehash操作**  **else** **if** (e **instanceof** TreeNode)  ((TreeNode<K,V>)e).split(**this**, newTab, j, oldCap);  -----------------链表的rehash操作  **else** { // preserve order  --------------------lo表示old，hi表示new  Node<K,V> loHead = **null**, loTail = **null**;  Node<K,V> hiHead = **null**, hiTail = **null**;  Node<K,V> next;  **do** {  next = e.next;  -----------------------**此处不是hash&cap-1，而是hash&cap，得到的是元素在数组中是否需要移动**  **if** ((e.hash & oldCap) == 0) {  -------------------------**tail为空表明该节点是第一节点，lohead指向e，否则**  **Tail。Next指向e，把e作为新的lotail**  **if** (loTail == **null**)  loHead = e;  **else**  loTail.next = e;  loTail = e;  }  -------------------**如果结果不为零**  **else** {  ----------------------**如果hitail==null，说明e为第一个节点，hihead指向他**  **if** (hiTail == **null**)  hiHead = e;  **else**  **------------------------否则就hital。Next指向e，将e作为新的hitail**  hiTail.next = e;  hiTail = e;  }  } **while** ((e = next) != **null**);  -------------------**循环结束后，应该有两条链，lohead和hihead链**  **if** (loTail != **null**) {  -----------------------**如果lotail不为空，说明该链有元素，此时将该链直接接到newtab的j的位置，即不用进行移动，原来该链就是j的位置**  loTail.next = **null**;  newTab[j] = loHead;  }  ------------------------**如果hitail不为空，表明该链有元素，需要重新移动移动规则为原来下标j+原始容量**  **if** (hiTail != **null**) {  hiTail.next = **null**;  newTab[j + oldCap] = hiHead;  }  }  }  }  }  **return** newTab;  } |

 **什么时候扩容：**通过HashMap源码可以看到是在put操作时，即向容器中添加元素时，判断当前容器中元素的个数是否达到阈值（当前数组长度乘以加载因子的值）的时候，就要自动扩容了。

 **扩容(resize)：**其实就是重新计算容量；而这个扩容是计算出所需容器的大小之后重新定义一个新的容器，将原来容器中的元素放入其中。  
**经过rehash之后，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置**

# 其他方法

|  |
| --- |
| **根据key，得到hash值，然后找到节点，为空返回false**  **public** **boolean** containsKey(Object key) {  **return** getNode(*hash*(key), key) != **null**;  }  -----**for循环，外层循环遍历tab，内层循环遍历链表**  **public** **boolean** containsValue(Object value) {  Node<K,V>[] tab; V v;  **if** ((tab = table) != **null** && size > 0) {  **for** (**int** i = 0; i < tab.length; ++i) {  **for** (Node<K,V> e = tab[i]; e != **null**; e = e.next) {  **if** ((v = e.value) == value ||  (value != **null** && value.equals(v)))  **return** **true**;  }  }  }  **return** **false**;  }  -------**如果entryset为空返回new entryset，如果不为空返回es**  **public** Set<Map.Entry<K,V>> entrySet() {  Set<Map.Entry<K,V>> es;  **return** (es = entrySet) == **null** ? (entrySet = **new** EntrySet()) : es;  } |

# ****HashMap和Hashtable的区别：****

1. HashMap允许key和value为null，Hashtable不允许。
2. HashMap的默认初始容量为16，Hashtable为11。
3. HashMap的扩容为原来的2倍，Hashtable的扩容为原来的2倍加1。
4. HashMap是非线程安全的，Hashtable是线程安全的。
5. HashMap的hash值重新计算过，Hashtable直接使用hashCode。
6. HashMap去掉了Hashtable中的contains方法。
7. HashMap继承自AbstractMap类，Hashtable继承自Dictionary类

# HashSet源码分析

**基本属性**

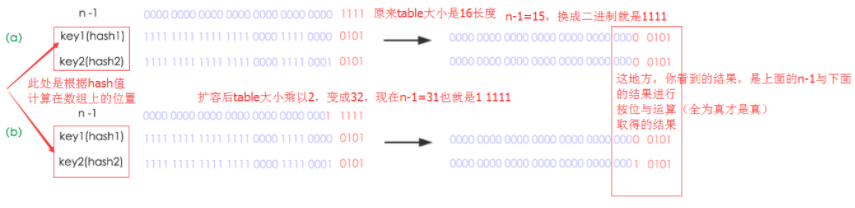
|  |
| --- |
| **static** **final** **long** ***serialVersionUID*** = -5024744406713321676L;  保存一个hashmap的变量，key为E，value为Object类型  **private** **transient** HashMap<E,Object> map;  不可变的Object类型的PRESENT  **private** **static** **final** Object ***PRESENT*** = **new** Object(); |

**构造器**

|  |
| --- |
| **无参构造器，构造一个**<E,Object>的hashmap，初始化map  **public** HashSet() {  map = **new** HashMap<>();  }  构造指定容量的hashmap，c的长度/0.75 + 1，并将集合添加进去，调用继承abstractCollection父类的addAll方法  **public** HashSet(Collection<? **extends** E> c) {  map = **new** HashMap<>(Math.*max*((**int**) (c.size()/.75f) + 1, 16));  addAll(c);  }  构造指定容量，指定负载因子的hashmap  **public** HashSet(**int** initialCapacity, **float** loadFactor) {  map = **new** HashMap<>(initialCapacity, loadFactor);  }  构造指定容量的hashmap  **public** HashSet(**int** initialCapacity) {  map = **new** HashMap<>(initialCapacity);  }  构造一个指定容量和指定负载因子的linkedHashmap  HashSet(**int** initialCapacity, **float** loadFactor, **boolean** dummy) {  map = **new** LinkedHashMap<>(initialCapacity, loadFactor);  } |

其余方法基本调用hashmap的方法

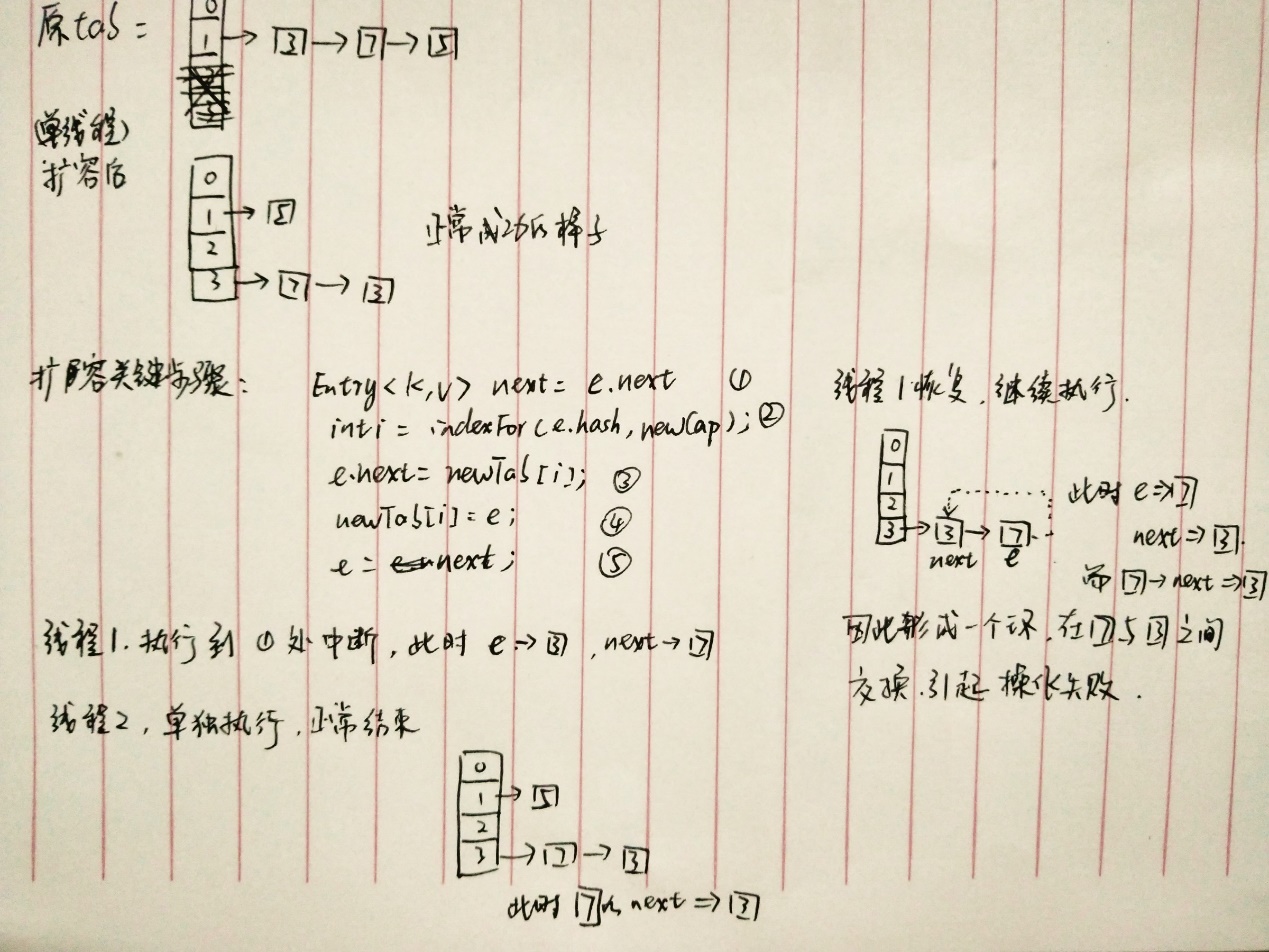
**HashMap扩容中，hash & oldCap的作用，观察扩容前和扩容后下标的变化**



原来的0101和10101在length=16的时候，通过hash&length-1的方法，计算出来都是0101；但是在扩容后即length=32时，hash&length -1 的方法计算出来为0101和10101；这就说明扩容后有些数据需要移动到原来index+oldcap的位置；为了观察0101和10101的区别发现只有最高位是0和1的区别；为了避免进行重复的hash&length -1计算，此时采用了hash&oldCap的方法，将10101和10000；也就是取最高位的值；来判断是否需要移动位置，节省时间。

**1.7的hashMap扩容成环问题**

|  |
| --- |
| void transfer(Entry[] newTable)  {  Entry[] src = table;  int newCapacity = newTable.length;  //下面这段代码的意思是：  // 从OldTable里摘一个元素出来，然后放到NewTable中  for (int j = 0; j < src.length; j++) {  Entry<K,V> e = src[j];  if (e != null) {  src[j] = null;  do {  Entry<K,V> next = e.next;  int i = indexFor(e.hash, newCapacity);  e.next = newTable[i];  newTable[i] = e;  e = next;  } while (e != null);  }  }  } |



上述的while循环中，线程2对newTab的操作是会对线程1产生影响的

**在1.8中，具体如下**：

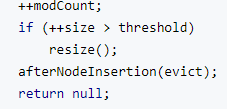
|  |
| --- |
| Node<K,V> loHead = **null**, loTail = **null**;  Node<K,V> hiHead = **null**, hiTail = **null**;  Node<K,V> next;  **do** {  next = e.next;  **if** ((e.hash & oldCap) == 0) {  **if** (loTail == **null**)  loHead = e;  **else**  loTail.next = e;  loTail = e;  }  **else** {  **if** (hiTail == **null**)  hiHead = e;  **else**  hiTail.next = e;  hiTail = e;  }  } **while** ((e = next) != **null**);  **if** (loTail != **null**) {  loTail.next = **null**;  newTab[j] = loHead;  }  **if** (hiTail != **null**) {  hiTail.next = **null**;  newTab[j + oldCap] = hiHead;  } |

线程1执行到next=e.next，中断；然后线程2开始执行。

当线程2执行完毕后，此时的hiHead与loHead属于局部变量，也就是线程2对其做的操作对线程1没有影响，并且该段代码在循环体中没有对newTab进行操作。

线程1此时恢复执行，hiHead与loHead都恢复成空，即重新执行一遍循环体，在进行newTable的赋值，会覆盖掉上次的结果，不过结果是一样的。

但是1.8也存在不安全的问题，在进行**添加元素和删除元素**的时候，如果该元素不存在，会进行size++；



此时如果两个线程同时执行，按理说size=3；但实际上会出现2；线程之间只对自己的副本进行操作，所以会引起这个问题。