1. 默认HashMap构造函数

|  |
| --- |
| public HashMap() {  this.loadFactor = ***DEFAULT\_LOAD\_FACTOR***; // 初始化对象的加载因子为0.75 } |

1. 调用put(key,value)

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

1. final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict)详解

|  |
| --- |
| 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:存放HashMap中的key-value键值对，第1次set值时对它进行初始化，必要时会进行resize，它的长度永远是2的整数次幂\*/  if ((tab = table) == null || (n = tab.length) == 0) //第1次调用时table为null，所以会执行resize()方法，resize方法在下面解释  n = (tab = resize()).length;  //下面(n - 1) & hash的作用是根据key的hash值和table的长度-1进行与运算，找到一个从0到table的长度-1的索引值  if ((p = tab[i = (n - 1) & hash]) == null)   //如果根据索引找到的对象为null则新建一个Node存到Node数组的该索引位置，注意新节点的next指针为null  tab[i] = newNode(hash, key, value, null);  else {  //如果根据索引找到的对象不是null，说明在Node数组的该索引位置已经有对象p了  Node<K,V> e; K k;  if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))  //如果节点p中的hash值和将要set的key的hash值相同，并且p中的key值和将要set的key值相同时：说明将要set的key值已经存在，此时将此key对应的新值覆盖其对应的旧值，并返回旧值  e = p;  else if (p instanceof TreeNode)  //如果节点p是一个TreeNode，说明：p是一个用双向链表表示的红黑树结构  e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  else {  //如果节点p中的key值和将要set的key值不相同，并且p也不是一个链表结构，说明：p只是一个单独的Node节点。那么就找到table在这个索引上的最后一个Node，并将新set的key-value这个Node挂在table在这个索引上的最后那个节点上  for (int binCount = 0; ; ++binCount) {  if ((e = p.next) == null) {  p.next = newNode(hash, key, value, null);  if (binCount >= ***TREEIFY\_THRESHOLD*** - 1)   //如果table上某个索引的Node链表长度达到8时，则需要将Node链表转为用TreeNode表示的红黑树结构  treeifyBin(tab, hash);  break;  }  if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))  break;  p = e;  }  }  if (e != null) { // existing mapping for key  V oldValue = e.value;  if (!onlyIfAbsent || oldValue == null)  e.value = value;  afterNodeAccess(e);  return oldValue;  }  }  ++modCount;  if (++size > threshold) //当HashMap中的元素数量大于threadhold时，就需要对table进行重新扩容，扩容后的容量是扩容前容量的2倍  resize();  afterNodeInsertion(evict);  return null; } |

1. final Node<K,V>[] resize()详解

|  |
| --- |
| final Node<K,V>[] resize() {  Node<K,V>[] oldTab = table; //第1次调用时table为null  int oldCap = (oldTab == null) ? 0 : oldTab.length; //oldCap为旧table的长度，第1次调用时为0  int oldThr = threshold; //threadhold：其值为0.75\*当前容量，当HashMap的元素数量超过threadhold时，需要对hashmap进行扩容，它是扩容临界点   int newCap, newThr = 0;  if (oldCap > 0) { //第1次调用时oldCap为0  if (oldCap >= ***MAXIMUM\_CAPACITY***) {  threshold = Integer.***MAX\_VALUE***;  return oldTab;  }  else if ((newCap = oldCap << 1) < ***MAXIMUM\_CAPACITY*** &&  oldCap >= ***DEFAULT\_INITIAL\_CAPACITY***)  newThr = oldThr << 1; // double threshold  }  else if (oldThr > 0) // initial capacity was placed in threshold  newCap = oldThr;  else { //第1次调用时执行该处，默认容量为16，默认下一次扩展后容量为16\*0.75=12   newCap = ***DEFAULT\_INITIAL\_CAPACITY***;  newThr = (int)(***DEFAULT\_LOAD\_FACTOR*** \* ***DEFAULT\_INITIAL\_CAPACITY***);  }  if (newThr == 0) {  float ft = (float)newCap \* loadFactor;  newThr = (newCap < ***MAXIMUM\_CAPACITY*** && ft < (float)***MAXIMUM\_CAPACITY*** ?  (int)ft : Integer.***MAX\_VALUE***);  }  threshold = newThr; //第1次调用时threadhold值为12  @SuppressWarnings({"rawtypes","unchecked"})  Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap]; //第1次调用时table的容量为16  table = newTab;  if (oldTab != null) { //第1次调用时不执行此处  for (int j = 0; j < oldCap; ++j) {  Node<K,V> e;  if ((e = oldTab[j]) != null) {  oldTab[j] = null;  if (e.next == null)  newTab[e.hash & (newCap - 1)] = e;  else if (e instanceof TreeNode)  //如果节点是一个TreeNode表示的红黑树，则将节点的hash值与oldCap进行与操作，将红黑树分成两个链表分别挂载到新table的j的索引位置和j+oldCap的索引位置上，在拆分过程中如果两个链表的长度达到8则将其转换为红黑树结构  ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  else { // preserve order  //如果节点是一个Node链表结构，则将节点的hash值与oldCap进行与操作，将链表分成两个链表分别挂载到新table的j的索引位置和j+oldCap的索引位置上  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;  }  }  }  }  }  return newTab; } |

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

|  |
| --- |
| //将table该所为位置的Node链表替换为用TreeNode组成的双线链表，并将双向链表转为红黑树 final void treeifyBin(Node<K,V>[] tab, int hash) {   int n, index; Node<K,V> e;  if (tab == null || (n = tab.length) < ***MIN\_TREEIFY\_CAPACITY***)  resize();  else if ((e = tab[index = (n - 1) & hash]) != null) {  TreeNode<K,V> hd = null, tl = null;  //下面的循环把table在该索引位置的Node链表循环转换为TreeNode双向链表  do {  TreeNode<K,V> p = replacementTreeNode(e, null); //把Node转换为TreeNode链表  if (tl == null) //把table在该索引位置的第1个Node转换为TreeNode后，再将其这设为head节点  hd = p;  else { //把table在该索引位置的其他（非第1个）Node转换为TreeNode后，顺次挂在第1个节点后面，这是个双向链表  p.prev = tl;  tl.next = p;  }  tl = p;  } while ((e = e.next) != null);  if ((tab[index] = hd) != null)  hd.treeify(tab); //将TreeNode双向链表转为红黑树  } } |

1. final void treeify(Node<K,V>[] tab)详解

|  |
| --- |
| final void treeify(Node<K,V>[] tab) {   TreeNode<K,V> root = null;  for (TreeNode<K,V> x = this, next; x != null; x = next) {  next = (TreeNode<K,V>)x.next;  x.left = x.right = null;  if (root == null) {  x.parent = null;  x.red = false;  root = x;  }  else {  K k = x.key;  int h = x.hash;  Class<?> kc = null;  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)  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;  root = balanceInsertion(root, x);  break;  }  }  }  }  moveRootToFront(tab, root); } |