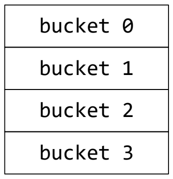
note author: heylichen@qq.com
JDK version: JDK 8

2 个重要参数。



capacity number of buckets in this case: 4
load factor
if entry count>= capacity * loadFactor, add buckets and rehash

类声明

```java
public class HashMap<K,V> extends AbstractMap<K,V>
                                implements Map<K,V>, Cloneable, Serializable
```

0.1 Static utilities

```java
/**
* Computes key.hashCode() and spreads (XORs) higher bits of hash  to lower.  Because the
* table uses power-of-two masking, sets of  hashes that vary only in bits above the current
* mask will  always collide. (Among known examples are sets of Float keys  holding consecutive
* whole numbers in small tables.)  So we  apply a transform that spreads the impact of higher
* bits  downward. There is a tradeoff between speed, utility, and  quality of bit-spreading.
* Because many common sets of hashes  are already reasonably distributed (so don't benefit
* from  spreading), and because we use trees to handle large sets of  collisions in bins,
* we just XOR some shifted bits in the  cheapest possible way to reduce systematic lossage,
* as well as  to incorporate impact of the highest bits that would otherwise  never be used
* in index calculations because of table bounds.
 主要是把高16位右移与低16位XOR。这样在低16位上就有高低位的信息。
*/
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

/**
* Returns x's Class if it is of the form "class C implements
* Comparable<C>", else null. 此注释清晰说明了方法的功能。
*/
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;
}

/**
* Returns k.compareTo(x) if x matches kc (k's screened comparable class), else 0.
*/
```

```
@SuppressWarnings({"rawtypes","unchecked"}) // for cast to Comparable
static int compareComparables(Class<?> kc, Object k, Object x) {
   return (x == null || x.getClass() != kc ? 0 : ((Comparable)k).compareTo(x));
}


/**
* Returns a power of two size for the given target capacity.
*/
static final int tableSizeFor(int cap) {
   int n = cap - 1; //下面分析中的第1步
   n |= n >>> 1;     //下面分析中的第2步
   n |= n >>> 2;
   n |= n >>> 4;
   n |= n >>> 8;
   n |= n >>> 16;
   return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1; //下面分析中的第3步
}
```

我们看一下 tableSizeFor，计算大于等于 cap 的一个 2 的幂。不用加减乘除，只用位运算，追求执行效率。

All (positive) powers of two have exactly 1 bit set; and (power of 2 - 1) has all of the bits set less than the most significant bit. So, we can find the next largest power of two by

1. Subtracting 1
2. Setting all of the less significant bits
3. Adding 1 back

证明上面 3 步的正确性。2 种场景，要么已经是 2 的幂，要么不是。假设最高的不是 0 的位在第 n 位，低 n-1 位如果都是 0，则已经是 2 的幂，经过上面 3 步，结果还是原值。如果低 n-1 位如果不都是 0，减 1，则最高位还在第 n 位，经过低 n-1 位置 1 再加上 1，则会变为最高位在第 n+1 位，低 n 位都是 0 的 2 的幂。符合预期。

代码中的位操作就是实现的第 2 步: 低位全部置为 1。例如 n=cap - 1 = 01010000

```
n |= n >>> 1;
得到          继续                一般地，每次多1位变为1，最多可能要做30次，可以把所有位都置为1.
  01010000       01111000             01xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
| 00101000     | 00111100          => 011xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
= 01111000     = 01111100          => 0111xxxxxxxxxxxxxxxxxxxxxxxxxxxx
                                   => 01111xxxxxxxxxxxxxxxxxxxxxxxxxxx
                                   => 011111xxxxxxxxxxxxxxxxxxxxxxxxxx
                                   ...
                                   => 01111111111111111111111111111111
```

But you might notice something interesting: after the first smear, when shifting by 1, we have the two most significant bits set. So, instead of shifting by 1, we can skip an operation by shifting by 2:

```
01xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
=> 011xxxxxxxxxxxxxxxxxxxxxxxxxxxxx 第1次右移1位再OR, 得到2个1
=> 01111xxxxxxxxxxxxxxxxxxxxxxxxxxx 第2次右移2位再OR, 得到4个1。继续这个方式
=> 011111111xxxxxxxxxxxxxxxxxxxxxxx 第3次右移4位再OR, 得到8个1
=> 01111111111111111xxxxxxxxxxxxxxx 第4次右移8位再OR, 得到16个1
=> 01111111111111111111111111111111 第5次右移16位再OR, 低位全部置为1
```

0.1.1 一些优化代码片段

计算 bucket 的 index

用位运算比 mod 运算效率高

```
  tab[i = (n - 1) & hash]) //tab就是Node<K,V>[] table, n = table.length
  //因为Node<K,V>[] table数组长度始终是2的幂, 因此(n - 1) & hash等价于hash mod n.
```

倾向于访问局部变量而不是实例变量

甚至因为此种做法，很多地方代码可以抽服用方法的，而没有抽。例如 compte, computeIfAbsent 等查找节点的相关代码。为何要这样？可能的原因是这样增加数据访问的空间局部性，提高性能。

```
public boolean containsValue(Object value) {
  Node<K,V>[] tab; V v; //明明可以直接访问实例变量table, 但偏要复制给tab,后续都访问tab
  if ((tab = table) != null && size > 0) { //整个HashMap源码都是此做法 , WHY?
    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;
}
```

下面是搜到的一些相关内容 (http://mail.openjdk.java.net/pipermail/core-libs-dev/2010-May/004165.html)。

Hi, in class String I often see member variables copied to local variables. In java.nio.Buffer I don't see that (e.g. for "position" in nextPutIndex(int nb)). Now I'm wondering.

From JMM (Java-Memory-Model) I learned, that jvm can hold non-volatile variables in a cache for each thread, so e.g. even in CPU register for few ones. From this knowing, I don't understand, why doing the local caching manually in String (and many other classes), instead trusting on the JVM.

Can anybody help me in understanding this ?

-Ulf 下面是回复:

It's a coding style made popular by Doug Lea. It's an extreme optimization that probably isn't necessary; you can expect the JIT to make the same optimizations. (you can try to check the machine code yourself!) Nevertheless, copying to locals produces the smallest bytecode, and for low-level code it's nice to write code that's a little closer to the machine.

Also, optimizations of finals (can cache even across volatile reads) could be better. John Rose is working on that.

For some algorithms in j.u.c, copying to a local is necessary for correctness.

Martin

而 JDK 8 的 HashMap 的第一作者就是 Doug Lea. 个人觉得不是很有必要。

0.2 core logic

0.2.1 static constants

```
/**
 * The default initial capacity - MUST be a power of two.
 */
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

/**
 * The maximum capacity, used if a higher value is implicitly specified  by either of the
 * constructors with arguments.  MUST be a power of two <= 1<<30.
 */
static final int MAXIMUM_CAPACITY = 1 << 30;

/**
 * The load factor used when none specified in constructor.
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f;

/**
 * The bin count threshold for using a tree rather than list for a bin. Bins are converted
 * to trees when adding an element to a bin with at least this many nodes. The value must
 * be greater than 2 and should be at least 8 to mesh with assumptions in tree removal about
 * conversion back to plain bins upon shrinkage.
 */
static final int TREEIFY_THRESHOLD = 8;
```

```
/**
 * The bin count threshold for untreeifying a (split) bin during a
 * resize operation. Should be less than TREEIFY_THRESHOLD, and at
 * most 6 to mesh with shrinkage detection under removal.
 */
static final int UNTREEIFY_THRESHOLD = 6;

/**
 * The smallest table capacity for which bins may be treeified. (Otherwise the table is
 * resized if too many nodes in a bin.) Should be at least 4 * TREEIFY_THRESHOLD to avoid
 * conflicts between resizing and treeification thresholds.
 */
static final int MIN_TREEIFY_CAPACITY = 64;
```

0.2.2 static inner class Node

```
/**
 * Basic hash bin node, used for most entries.  (See below for
 * TreeNode subclass, and in LinkedHashMap for its Entry subclass.)
 */
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;   //hash和key是final的
    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() { //equals和hashCode同时考虑key和value
        return Objects.hashCode(key) ^ Objects.hashCode(value);
    }

    public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

    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;
    }
}
```

0.2.3 fields and construction

```
/**
 * The table, initialized on first use, and resized as necessary. When allocated, length is
 * always a power of two. (We also tolerate length zero in some operations to allow bootstrapping
 * mechanics that are currently not needed.) bucket数组，注意长度永远是2的幂。
 */
transient Node<K,V>[] table;
//Holds cached entrySet(). Note that AbstractMap fields are used for keySet() and values().
transient Set<Map.Entry<K,V>> entrySet;
//The number of key-value mappings contained in this map.
transient int size;
/**
 * The number of times this HashMap has been structurally modified Structural modifications
 * are those that change the number of mappings in the HashMap or otherwise modify its internal
 * structure (e.g., rehash). This field is used to make iterators on Collection-views of
 * the HashMap fail-fast. (See ConcurrentModificationException).
 */
transient int modCount;
/**
 * The next size value at which to resize (capacity * load factor). @serial
 * (The javadoc description is true upon serialization. Additionally, if the table array
 * has not been allocated, this field holds the initial array capacity, or zero signifying
 * DEFAULT_INITIAL_CAPACITY.) 注意此值在初始化时有其他2个含义，并不表示threshold。
   1 初始化时，如果>0，表示initial array capacity，存的是capacity不是threshold。
   2 初始化时，如果==0,表示DEFAULT_INITIAL_CAPACITY，存的是capacity不是threshold。
   3 其他情况存的就是threshold
 */
int threshold;
//The load factor for the hash table. @serial
final float loadFactor;

// Constructs an empty <tt>HashMap</tt> with the specified initial capacity and load factor.
public HashMap(int initialCapacity, float loadFactor) {
   if (initialCapacity < 0)
       throw new IllegalArgumentException("Illegal initial capacity: " + initialCapacity);
   if (initialCapacity > MAXIMUM_CAPACITY) initialCapacity = MAXIMUM_CAPACITY;
   if (loadFactor <= 0 || Float.isNaN(loadFactor))
       throw new IllegalArgumentException("Illegal load factor: " + loadFactor);
   this.loadFactor = loadFactor;
   this.threshold = tableSizeFor(initialCapacity);
}

/**
 * Constructs an empty <tt>HashMap</tt> with the specified initial capacity and the default
 * load factor (0.75).
*/
public HashMap(int initialCapacity) {
   this(initialCapacity, DEFAULT_LOAD_FACTOR);
}

/**
 * Constructs an empty <tt>HashMap</tt> with the default initial capacity (16) and the default
 * load factor (0.75).
*/
public HashMap() { //threshold默认值0，表示capacity是DEFAULT_INITIAL_CAPACITY
   this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}

/**
 * Constructs a new HashMap with the same mappings as the specified Map. The HashMap is
 * created with default load factor (0.75) and an initial capacity sufficient to hold the
 * mappings in the specified Map.
*/
```

```java
public HashMap(Map<? extends K, ? extends V> m) {
   this.loadFactor = DEFAULT_LOAD_FACTOR;
   putMapEntries(m, false);
}

final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {
    int s = m.size();
    if (s > 0) {
        if (table == null) { // pre-size
            float ft = ((float)s / loadFactor) + 1.0F;
            int t = ((ft < (float)MAXIMUM_CAPACITY) ? (int)ft : MAXIMUM_CAPACITY);
            if (t > threshold) threshold = tableSizeFor(t); //初始化实例变量threshold
        }
        else if (s > threshold)
            resize();
        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); //见put章节的详解。有需要会扩容
        }
    }
}
```

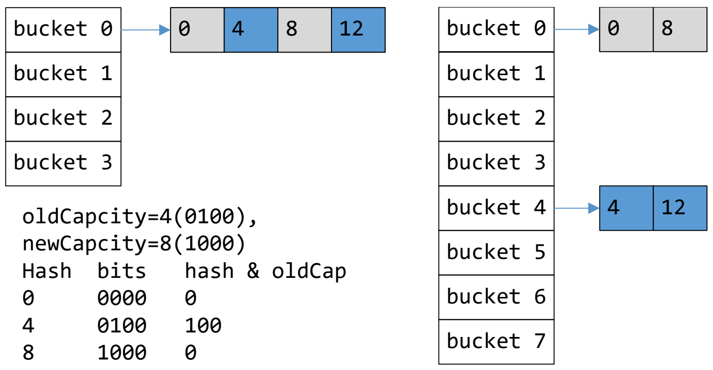
0.2.4 resize

```java
final Node<K,V>[] resize() {
  Node<K,V>[] oldTab = table;
  int oldCap = (oldTab == null) ? 0 : oldTab.length;
  int oldThr = threshold;
  int newCap, newThr = 0;
  if (oldCap > 0) { //case 1 初始化过，oldCap扩容
    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;   // case 2 未初始化，capacity存在threshold
  else {               // zero initial threshold signifies using defaults
    newCap = DEFAULT_INITIAL_CAPACITY; //case 3 未初始化，threshold==0, 使用默认值
    newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
  }
  if (newThr == 0) { //newThr无值, 则计算
    float ft = (float)newCap * loadFactor;
    newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
              (int)ft : Integer.MAX_VALUE);
  }
  threshold = newThr;
  @SuppressWarnings({"rawtypes","unchecked"})
  Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap]; //扩容空间
  table = newTab;
  if (oldTab != null) { //如果原来有buckets, 全部rehash
    for (int j = 0; j < oldCap; ++j) {
      Node<K,V> e;
      if ((e = oldTab[j]) != null) {
        oldTab[j] = null;
        if (e.next == null) //因为newCap是2的幂, e.hash & (newCap - 1)就是算e.hash mod newCap
          newTab[e.hash & (newCap - 1)] = e; //只有1个节点, 直接复制
        else if (e instanceof TreeNode) // 此buket节点过多时会转为红黑树
          ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
        else { // preserve order 有适量节点，rehash, 见下图展开分析
          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;
}
```

rehash 逻辑

得益于 buckets 数组长度都是 2 的幂，且每次扩容，必然是数组长度变为原来的 2 倍，rehash 时，可以采取高效的逻辑。在对一个 bucket 的节点链表 rehash 时，只要遍历节点，对节点 hash 值做位运算，即可完成 rehash。因为原来的属于同一个 bucket 的所有节点，因为扩容后 buket 数量变为原来 2 倍，这些节点扩容后会属于 2 个不同的 bucket。哪个节点属于原 bucket, 哪个节点属于原 bucketIndex+oldCap? 取决于 hash 值 / oldCap 是奇数还是偶数。从下面图示的例子可以理解。

如果不是利用这点，那么 rehash 的过程稍微复杂一些。在 rehash 1 个 bucket 链表时，遍历每个节点，对该节点的 hash 值重算新的 bucketIndex. 然后找到新的 buket 的链表的尾部节点（可能要维护 1 个变量），再做插入到新的尾部节点。这样要对每个新的 bucket 维护 1 个尾部节点变量。因为计算 bucketIndex 可以用位运算，效率还是蛮高的。但复杂性增加在于维护 newCapacity 个 bucket 的尾部节点引用。



```
oldCapcity=4(0100),
newCapcity=8(1000)
Hash  bits   hash & oldCap
0     0000   0
4     0100   100
8     1000   0
12    1100   100
```

0.2.5 put

```
/**
 * Associates the specified value with the specified key in this map. If the map previously
 * contained a mapping for the key, the old value is replaced.
 */
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

/**
 * Implements Map.put and related methods.
 *
 * @param hash hash for key
 * @param key the key
 * @param value the value to put
 * @param onlyIfAbsent if true, don't change existing value
 * @param evict if false, the table is in creation mode.
 * @return previous value, or null if none
 */
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
  Node<K,V>[] tab; Node<K,V> p; int n, i;
  if ((tab = table) == null || (n = tab.length) == 0)
    n = (tab = resize()).length; //如果原来buckets是空, 扩容
  if ((p = tab[i = (n - 1) & hash]) == null)
    tab[i] = newNode(hash, key, value, null); //如果对应bucket链表为空, 直接新建节点
  else {
    Node<K,V> e; K k;
    if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
      e = p; // bucket节点链表, 第1个节点就是对应的key
    else if (p instanceof TreeNode)
      e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value); //bucket节点是红黑树
    else { //遍历bucket节点链表, 找对应key
      for (int binCount = 0; ; ++binCount) {
        if ((e = p.next) == null) { //bucket节点链表没找到对应key
          p.next = newNode(hash, key, value, null); //新建节点,
          if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
              treeifyBin(tab, hash); //bucket节点链表数量达到阈值, 转为红黑树
          break; //新建节点时, 直接退出for, e==null
        }
        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
          break; //找到对应key的节点, e!=null
        p = e;
      }
    }
    if (e != null) { // existing mapping for key
        V oldValue = e.value;
        if (!onlyIfAbsent || oldValue == null) e.value = value;
        afterNodeAccess(e); //用于LinkedHashMap, 此处暂不关心
        return oldValue;
    }
  }
  ++modCount;
  if (++size > threshold) resize(); // entry数量大于阈值(loadFactor指定), 扩容, rehash
  afterNodeInsertion(evict);        //用于LinkedHashMap, 此处暂不关心
  return null;
}
```

0.2.6 get

```
/**
```

```
 * Returns the value to which the specified key is mapped, or {@code null} if this map contains
 * no mapping for the key. More formally, if this map contains a mapping from a key {@code
 * k} to a value {@code v} such that {@code (key==null ? k==null : key.equals(k))}, then
 * this method returns {@code v}; otherwise it returns {@code null}. (There can be at most
 * one such mapping.) A return value of {@code null} does not necessarily indicate that
 * the map contains no mapping for the key; it's also possible that the map explicitly maps
 * the key to {@code null}. The {@link #containsKey containsKey} operation may be used to
 * distinguish these two cases.   @see #put(Object, Object)
 */
public V get(Object key) {
  Node<K,V> e;
  return (e = getNode(hash(key), key)) == null ? null : e.value;
}

// Implements Map.get and related methods.
final Node<K,V> getNode(int hash, Object key) {
  Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
  if ((tab = table) != null && (n = tab.length) > 0 &&
    (first = tab[(n - 1) & hash]) != null) {
    if (first.hash == hash && ((k = first.key) == key || (key != null && key.equals(k))))
      return first; // always check first node
    if ((e = first.next) != null) {
      if (first instanceof TreeNode)
          return ((TreeNode<K,V>)first).getTreeNode(hash, key);
      do {
          if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
              return e;
      } while ((e = e.next) != null);
  }
  return null;
}
```

0.2.7 remove

```
/*
 * Removes the mapping for the specified key from this map if present. @param key key whose
 * mapping is to be removed from the map @return the previous value associated with key,
 * or null if there was no mapping for key. (A null return can also indicate that the map
 * previously associated null with key.)
 */
public V remove(Object key) {
    Node<K,V> e;
    return (e = removeNode(hash(key), key, null, false, true)) == null ?
        null : e.value;
}

/**
* Implements Map.remove and related methods.
*
* @param hash hash for key
* @param key the key
* @param value the value to match if matchValue, else ignored
* @param matchValue if true only remove if value is equal
* @param movable if false do not move other nodes while removing
* @return the node, or null if none
*/
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;
  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; //寻找key等于入参的节点，存在node中
    if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
```

```
          node = p;                           //case 1 目标节点node就是对应bucket首节点
        else if ((e = p.next) != null) { //进入分支时，p是bucket首节点，e=p.next
          if (p instanceof TreeNode)
            node = ((TreeNode<K,V>)p).getTreeNode(hash, key); //case 2 红黑树中找
          else {
            do { //循环时保持e=p.next
              if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k)))) {
                node = e;
                break;
              }
              p = e;        //退出循环时有2种情况，1) 没找到目标节点node==null,
            } while ((e = e.next) != null);     //2) 目标节点为node, node=p.next. case 3
          }
        }
        if (node != null && (!matchValue || (v = node.value) == value ||
                             (value != null && value.equals(v)))) {
          if (node instanceof TreeNode)
              ((TreeNode<K,V>)node).removeTreeNode(this, tab, movable); //case 2
          else if (node == p)
              tab[index] = node.next;                                   //case 1
          else
              p.next = node.next;                                       //case 3
          ++modCount;
          --size;
          afterNodeRemoval(node);
          return node;
        }
      }
      return null;
    }

    public boolean remove(Object key, Object value) {
      return removeNode(hash(key), key, value, true, true) != null;
    }
```

0.2.8 compute and merge

```
@Override
public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction) {
  if (mappingFunction == null)
      throw new NullPointerException();
  int hash = hash(key);
  Node<K,V>[] tab; Node<K,V> first; int n, i;
  int binCount = 0;
  TreeNode<K,V> t = null;
  Node<K,V> old = null;
  if (size > threshold || (tab = table) == null || (n = tab.length) == 0)
      n = (tab = resize()).length; //按需扩容
  if ((first = tab[i = (n - 1) & hash]) != null) {
    // 按key查找对应节点的原值
    if (first instanceof TreeNode)
      old = (t = (TreeNode<K,V>)first).getTreeNode(hash, key);
    else {
      Node<K,V> e = first; K k;
      do {
        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k)))) {
          old = e;
          break;
        }
        ++binCount;
      } while ((e = e.next) != null);
    }
    V oldValue;
```

```
      //找到old, 且old.value有值则直接返回
      if (old != null && (oldValue = old.value) != null) {
        afterNodeAccess(old);
        return oldValue;
      }
    }
    //old==null或者 old!=null,old.value==null
    V v = mappingFunction.apply(key);
    if (v == null) {
        return null;          //key维持无值, 因原值为空, 所以无需操作
    } else if (old != null) { //不管是在红黑树还是在节点链表中, 找到节点node!=null
        old.value = v;        //  old != null且v!=null, 更新key的值
        afterNodeAccess(old);
        return v;
    }
    else if (t != null)       //old==null, bucket首节点为红黑树root
        t.putTreeVal(this, tab, hash, key, v);
    else {                    //old==null, bucket中为普通节点链表
        tab[i] = newNode(hash, key, v, first);
        if (binCount >= TREEIFY_THRESHOLD - 1)
            treeifyBin(tab, hash);
    }
    ++modCount;
    ++size;
    afterNodeInsertion(true);
    return v;
}

public V computeIfPresent(K key,
                        BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
    if (remappingFunction == null)
        throw new NullPointerException();
    Node<K,V> e; V oldValue;
    int hash = hash(key);
    if ((e = getNode(hash, key)) != null && (oldValue = e.value) != null) {
      V v = remappingFunction.apply(key, oldValue);
      if (v != null) {
        e.value = v;
        afterNodeAccess(e);
        return v;
      }
      else
        removeNode(hash, key, null, false, true);
    }
    return null;
}

@Override
public V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
    if (remappingFunction == null)
      throw new NullPointerException();
    int hash = hash(key);
    Node<K,V>[] tab; Node<K,V> first; int n, i;
    int binCount = 0;
    TreeNode<K,V> t = null;
    Node<K,V> old = null;
    if (size > threshold || (tab = table) == null || (n = tab.length) == 0)
      n = (tab = resize()).length; // 按需扩容
    //按key查找对应节点old
    if ((first = tab[i = (n - 1) & hash]) != null) {
      if (first instanceof TreeNode)
        old = (t = (TreeNode<K,V>)first).getTreeNode(hash, key);
      else {
        Node<K,V> e = first; K k;
```

```
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k)))) {
                    old = e;
                    break;
                }
                ++binCount;
            } while ((e = e.next) != null);
        }
    }
    V oldValue = (old == null) ? null : old.value;
    V v = remappingFunction.apply(key, oldValue);
    if (old != null) { //原来有对应节点
      if (v != null) { //计算后有值，更新
          old.value = v;
          afterNodeAccess(old);
      }
      else            //计算后新值为null, remove
          removeNode(hash, key, null, false, true);
    }
    else if (v != null) { //原来old==null, 但新值!=null, 插入新值节点
      if (t != null)
          t.putTreeVal(this, tab, hash, key, v);
      else {
          tab[i] = newNode(hash, key, v, first);
          if (binCount >= TREEIFY_THRESHOLD - 1)
              treeifyBin(tab, hash);
      }
      ++modCount;
      ++size;
      afterNodeInsertion(true);
    }  // else old==null, 新值==null, 维持无值原状
    return v;
}

@Override
public V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction) {
    if (value == null)
        throw new NullPointerException();
    if (remappingFunction == null)
        throw new NullPointerException();
    int hash = hash(key);
    Node<K,V>[] tab; Node<K,V> first; int n, i;
    int binCount = 0;
    TreeNode<K,V> t = null;
    Node<K,V> old = null;
    if (size > threshold || (tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length; //按需扩容
    //按key查找对应节点old
    if ((first = tab[i = (n - 1) & hash]) != null) {
        if (first instanceof TreeNode)
            old = (t = (TreeNode<K,V>)first).getTreeNode(hash, key);
        else {
            Node<K,V> e = first; K k;
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k)))) {
                    old = e;
                    break;
                }
                ++binCount;
            } while ((e = e.next) != null);
        }
    }
```

```
  if (old != null) {
      V v;
      //有原值，计算用原值和入参值计算新值，否则入参作为新值。与compute区别就在于此
      if (old.value != null)
          v = remappingFunction.apply(old.value, value);
      else
          v = value;
      if (v != null) {
          old.value = v;
          afterNodeAccess(old);
      }
      else
          removeNode(hash, key, null, false, true);
      return v;
  }
  if (value != null) {
      if (t != null)
          t.putTreeVal(this, tab, hash, key, value);
      else {
          tab[i] = newNode(hash, key, value, first);
          if (binCount >= TREEIFY_THRESHOLD - 1)
              treeifyBin(tab, hash);
      }
      ++modCount;
      ++size;
      afterNodeInsertion(true);
  }
  return value;
}
```

0.2.9 iterator

```
abstract class HashIterator {
  Node<K,V> next;        // next entry to return
  Node<K,V> current;     // current entry
  int expectedModCount;  // for fast-fail
  int index;             // current slot

  HashIterator() {
    expectedModCount = modCount;
    Node<K,V>[] t = table;
    current = next = null;
    index = 0;
    if (t != null && size > 0) { // advance to first entry 找到第一个不为空的bucket的index
        do {} while (index < t.length && (next = t[index++]) == null);
    }
  }

  public final boolean hasNext() {
      return next != null;
  }

  final Node<K,V> nextNode() {
    Node<K,V>[] t;
    Node<K,V> e = next; // 返回值此时已取好
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
    if (e == null)
        throw new NoSuchElementException();
    //1 保存current, 2 保存next = current.next, 如果不为空，不进入分支
    //3 如果current.next为空，则找到第一个不为空的bucket的首节点保存到next
    if ((next = (current = e).next) == null && (t = table) != null) {
        do {} while (index < t.length && (next = t[index++]) == null);
    }
    return e;
  }

  public final void remove() {
      Node<K,V> p = current;
      if (p == null)
          throw new IllegalStateException();
      if (modCount != expectedModCount)
          throw new ConcurrentModificationException();
      current = null;
      K key = p.key;
      removeNode(hash(key), key, null, false, false);
      expectedModCount = modCount;
  }
}

final class KeyIterator extends HashIterator implements Iterator<K> {
  public final K next() { return nextNode().key; }
}

final class ValueIterator extends HashIterator implements Iterator<V> {
  public final V next() { return nextNode().value; }
}

final class EntryIterator extends HashIterator implements Iterator<Map.Entry<K,V>> {
  public final Map.Entry<K,V> next() { return nextNode(); }
}
```