

首先介绍完上图，然后开始针对没集合框架类，进行执行过程的描述。

1.为什么不使用实现Iterator，而使用Iterable？

解：因为如果实现Iterator，必定要实现hasNext以及next方法，也就是必须得定义当前迭代器迭代到哪里的指针，假设为A，但是，如果该集合框架要作为一个方法参数进行传递的时候，到底指针A是指向当前位置，还是重新开始迭代，这会无法预知。因此，实现Iterable，通过实现iterator()方法，new一个迭代器，避免多个迭代器之间相互影响

2.synchronized底层实现原理与应用

**Synchronized主要有三种应用方式：**

修饰实例方法，修饰静态方法，修饰同步代码块，synchronized实现都是基于进入和退出Monitor对象来实现方法同步和代码块同步，但是两者实现细节不一样。任何对象都有个monitor与其关联，一旦一个monitor被持有后，它将处于锁定状态。

**修饰同步代码块的时候**：使用monitorenter和moniterexit指令实现的，线程执行到monitorenter指令的时候，会尝试获取对象的monitor所有权，执行monitorexit的时候又会释放monitor的所有权，**无论这个方法是正常结束还是异常结束。为了保证在方法异常完成时 monitorenter 和 monitorexit 指令依然可以正确配对执行，编译器会自动产生一个异常处理器，这个异常处理器声明可处理所有的异常，它的目的就是用来执行 monitorexit 指令。**monitorenter编译后插入同步代码块的开始位置，monitorexit插入到同步代码块结束的位置，还有另一个种情况就是上面说的，方法正常结束，插入方法结束处，异常结束插入异常处。

**修饰方法的时候**：采用ACC\_SYNCHRONIZED标识，该标识指明了该方法是一个同步方法，JVM通过该ACC\_SYNCHRONIZED访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。此时持有的是当前类实例对象的monitor，直到方法结束后释放对象的monitor

**由于synchronized锁是存储在对象头里面的，因此还得介绍一下对象头**

**非数组类型**采用2字宽，包括mark word（32bit）、class metadata address（32bit）（指向对象类型数据的指针，用于判断对象时哪个类的实例）

**数组类型**采用3字宽，多了一个array length（32bit）

无锁状态：**对象hashcode**25bit，**对象分代年龄**4bit，1bit表示**是否偏向锁**，2bit表示**锁标志位**

轻量级锁：指向栈中锁记录的指针（30bit），锁标志00

重量级锁(普通synchronized锁): 指向互斥量（重量级锁）的指针（30bit），标志10

GC标记：空（30bit），锁标志11

偏向锁： 线程id（23bit）、Epoch（2bit）、对象分代（4bit），是否偏向1（1bit），锁标志01

**Synchronized锁优化**：

**偏向锁**：锁不仅不存在多线程竞争，而且总是由同一线程多次获得，这时候就需要用到偏向锁

①访问同步块

②判断该对象头中是否存储了当前线程的id

--2.1如果有，说明该线程已经获得了偏向锁，不需要进行CAS加锁解锁

--2.2如果没有，使用CAS竞争锁，修改对象头，写入当前线程的id；

③如果再执行过程中，有其他线程来竞争偏向锁，此时就会出发释放偏向锁，暂停原来拥有偏向锁的线程，进行解锁，将线程id置为空，将对象头恢复为无锁状态01；然后恢复线程。当存在竞争的时候，偏向锁就失败了，会升级为**轻量级锁**。

**轻量级锁**：轻量级锁所适应的场景是线程交替执行同步块的场合

①执行同步前，jvm会在当前线程的栈帧中，创建存储锁记录的空间，将对象头中的markword复制到锁记录的栈帧中。

②使用CAS将对象头中的markword替换为指向锁记录的指针

--2.1如果成功，获得轻量级锁。

--2.2如果失败，表明当前锁被其他线程占有，此时会产生自旋锁

自旋锁：通过几个空循环，可能50到100，在若干个空循环之后，继续进行步骤②；此时如果其他线程释放锁了，当前线程获得锁，开始执行同步块；

如果失败了，就只能膨胀为重量锁了，**当前线程阻塞**。

③执行同步块

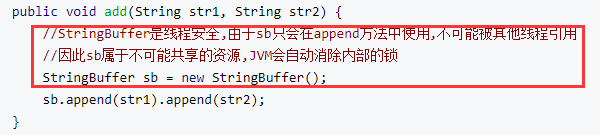
④采用CAS将指向所记录的指针重新替换为原来的对象头状态

--4.1如果成功，释放锁

--4.2如果失败，说明存在重量锁的竞争，释放锁并唤醒阻塞的线程，由重量锁的线程执行同步块。

**锁消除：**

去除不可能存在共享资源竞争的锁，通过何种方式消除没有必要的锁，可以节省毫无意义的请求锁时间。



局部变量不可能存在资源共享，所以进行锁消除

**3.ArrayList执行过程**

**3.1成员变量**

**默认容量**DEFAULT\_CAPACITY 10；**空常量数组**EMPTY\_ELEMENTDATA；**默认空常量数组**DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA；**实际存储数组**Object[] elementData；**元素个数**size；

**3.2构造器**

3.2.1指定初始容量的构造器：

如果初始容量>0，直接elementData =new object[initialCapacity]；

如果等于0；就使用elementData =EMPTY\_ELEMENTDATA；（在add方法里面会区分EMPTY\_ELEMENTDATA与DEFAULT\_EMPTY\_ELEMENTDATA）.如果小于0，就抛出异常

3.2.2无参构造器

ElementData = DEFAULT\_EMPTY\_ELEMENTDATA；

3.2.3使用集合作为参数的构造器

使用c的toArray方法，返回object[]，并赋值给elementData，如果集合长度不为0，检查toArray是否真的返回object[]，如果没有使用Arrays.copyOf，将原来的数组，拷贝为object[]类型的数组；如果集合长度为0，elementData =EMPTY\_ELEMENTDATA；

**3.3获取设定元素**

Get(int)方法，先检查index是否大于等于size，然后获取index下标的元素

Set（int， E）,先检查index是否大于等于size，然后获取旧值，设置新值，返回旧值

**3.4 扩容方法**

**MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8**

grow(int minCapacity)：首先获取原数组的长度，然后newCapacity=old + old >> 1；如果newCap 小于 minCap，就让newCap等于minCap；如果newCap大于MAX\_ARRAY\_SIZE，要么newCap等于Integer..MAX\_VALUE,要么等于MAX\_ARRAY\_SIZE；调用Arrays.copyOf进行原来数组拷贝到新长度的数组中。

**3.5 增加元素**

3.5.1 add(E e)；

初始化数组的时候，size=0；首先调用ensureCapacityInternal(size + 1);确认容量，这个ensureCapacityInternal方法，首先判断elementData是否是DEFAULT\_EMPTY\_ELEMENTDATA，如果是，让minCap = DEFAULT\_CAPACITY与minCap的最大值，由于minCap刚开始为1；所以没有DEFAULT\_CAPACITY=10大；因此在无参构造器的时候，elementData = DEFAULT\_EMPTY\_ELEMENTDATA，就是为扩容时，将elementData容量设置为10；接下来调用ensureExplicitCapacity，如果minCap>数组长度（此时为0）；就进行容量为10的扩容。这也就是EMPTY\_ELEMENTDATA与DEFAULT\_EMPTY\_ELEMENTDATA的区别。如果是EMPTY\_ELEMENTDATA类型，最后扩容的长度就是1；

扩容结束后，就将e放到size的位置，此时size始终小于数组容量；

3.5.2 add(int index, E element)

判断index不能大于size，不能小于0；然后调用ensureCapacityInternal(size + 1);观察是否需要扩容；System.arraycopy将index开始到最后的内容，移动到index+1之后；进行元素的插入、

**3.6删除元素**

3.6.1 remove(int index)

检查index是否合法，保存要删除的元素的值oldVal；然后计算移动元素的步数，size-index-1；观察移动步数是否>0，如果是进行System.arraycopy将index+1后面的元素，移动到index后面，如果<=0，说明index不合法，或者是最后一个元素，不需要移动。然后将最后一个元素设置为null，有助于GC，返回旧值oldVal

3.6.2 remove(Object o)

如果o==null，找到第一个为null的元素下标，然后调用fastRemove(index)；否则找到第一个元素与o的equals方法返回true的元素下标，然后调用fastRemove(index)；

3.6.3 fastRemove(int index)

与remove(index)基本一致，少了有效性检查和没有返回值

**4.LinkedList的执行过程**

**4.1节点Node的定义**（表明LinkedList是双向链表实现的）

静态内部类：E item，指向前一个和后一个的Node引用prev 和next；然后一个构造方法，用于构造element，对item，next和prev进行初始化。

**4.2成员变量**

Size = 0；指向头尾节点的引用first和last；

**4.3构造方法**

4.3.1 LinkedList()无参构造器

什么都不做

4.3.2 LinkedList(Collection<? **extends** E> c)

调用this();然后使用addAll（c）将集合添加进去；

4.3.3 为什么LinkedList没有提供指定大小的构造器？

因为LinkedList采用链表存储，不需要申请连续的空间，并且添加元素很方便，对于指定大小没有必要。

**4.4 查找元素**

4.4.1get(int index):

判断index是否合法，然后调用node(index),返回Node，最后返回node.item；

4.4.2 getFirst（），getLast（）

直接获取first.item和last.item；

4.4.3 indexOf(Object o)

不管o是否为null，遍历，如果为null找到第一个item为null的返回index，如果不为null，找到第一个item与o.equals返回为true的，返回下标

4.4.4 node（index）返回index的节点Node

判断index是在size>>1，前半部分还是后半部分；如果前半部分从first向后遍历；如果后半部分从last向前遍历；找到对应节点；

**4.5 添加元素**

4.5.1 add(E e)，addLast(E e) ，offerLast ，offer(E e)添加到表尾

调用linkLast(e)；linkLast：获取last节点为l，然后创建一个newNode，节点item为e，前驱指向l，后继为null，将newNode设为新的last；如果l为空，说明当前无节点，first指向newNode,如果不为空，last.next指向newNode；

4..5.2 addFirst(E e), offerFirst

调用linkFirst(e)：获取first节点f，创建一个newNode，item为e，前驱null，后继为f，新节点赋值为first。如果f==null，当前无节点，last = newNode，如果不为空，f.prev = newNode；

4.5.3 add(int index, E element)

检查index是否越界，是否属于0-size之间；如果index==size，直接调用linkLast添加到表尾；否则调用linkBefore（element, node(index);

4.5.4 linkBefore(E e, Node<E> succ)在非空节点succ之前插入元素

得到succ的前驱pred，生成一个新节点newNode，前驱指向pred，后继指向succ

如果pred为空，说明succ为first，让first指向newNode，如果不为空，就pred.next指向newNode

4.5.5 addAll(Collection<? **extends** E> c)，将集合插入到列表的末尾

底层调用addAll(size, c);

4.5.6 addAll(int index, Collection<? extends E> c)将集合元素插入到列表中的指定位置

判断越界；将集合c转换为数组a；记录a的长度；如果长度为0，返回false；否则判断index和size是否相等，设定两个节点，pred和succ分别表示当前节点的前驱和当前节点；如果index等于size；succ=null，predlast；如果不等；succ=node(index)，pred=succ.pred;;

变量a数组，将a数组的元素赋值为e，创建节点newNode，前驱是pred，后继为null；

如果pred==null说明succ为first，那就让first指向newNode；否则就pred.next指向newNode；一趟结束后，更新pred为newNode；此时前半部分已经链好，只剩下a的最后一个节点，也就是新的pred，与succ之间的连接；如果succ为空，说明插入的是表尾，只需将last指向pred；如果succ不为空，pred.next = succ,succ.prev = pred;

**4.6删除元素**

4.6.1 remove（）删除第一个节点，底部调用removeFirst；removeFirst调用unlinkFirst，删除第一个节点，removeFirst里面已经进行first的判空了，这里不需要判断

unlinkFirst(Node<E> f)

获取first节点的item，以及next节点赋值为next；将first.item和next置为空；first指向新的next；如果next为空，说明first就是唯一节点，将last指向null即可；如果不为空，next.prev=null；返回item

4.6.2 remove(int index) 底层调用unlink(node(index));

unlink(Node<E> x)默认x不为空；

记录节点x的数据，item记为element，前驱记为prev，后继记为next；

如果前驱为空，first指向next；否则，prev.next = next; x.prev=null;

如果后继为空，last指向prev，否则，next.prev = prev, x.next =null;

将x的item置为空，便于gc，然后返回删除元素的值element；

4.6.3 remove(Object o)删除指定对象o

不管o是否为空，若为空，从前开始遍历找到第一个为item为null的节点，调用unlink（x）进行删除，如果不为空，从前往后遍历直到找到o.equals(x.item)为true的，调用unlink（x）

4.6.4 removeLast()，last不为空调用unlinkLast(Node<E> l)

得到last的前驱节点prev；记录last的值以便返回；将当前last置空，last指向prev；如果prev为空，让first为空，如果不为空，prev.next=null；返回删除元素值

**4.7修改元素**

Set(int index, E element)

找到node（index），然后修改item的值为element

**4.8查询元素**

Get（index），主要依靠node(index)

**4.9遍历元素**

lastReturned上一次返回的节点，nextIndex下一个节点的索引；ListItr(int index)构造指定next索引的迭代器。

hasNext：nextIndex是否小于size；next（）；检查exceptedModcount==modcount，不等就说明在迭代的时候，对链表元素进行改变了，抛出异常；将当前next的值给lastReturned；然后next = next.next; 索引++；返回lastReturned的item值，即在next中，next始终比lastReturned后一个；

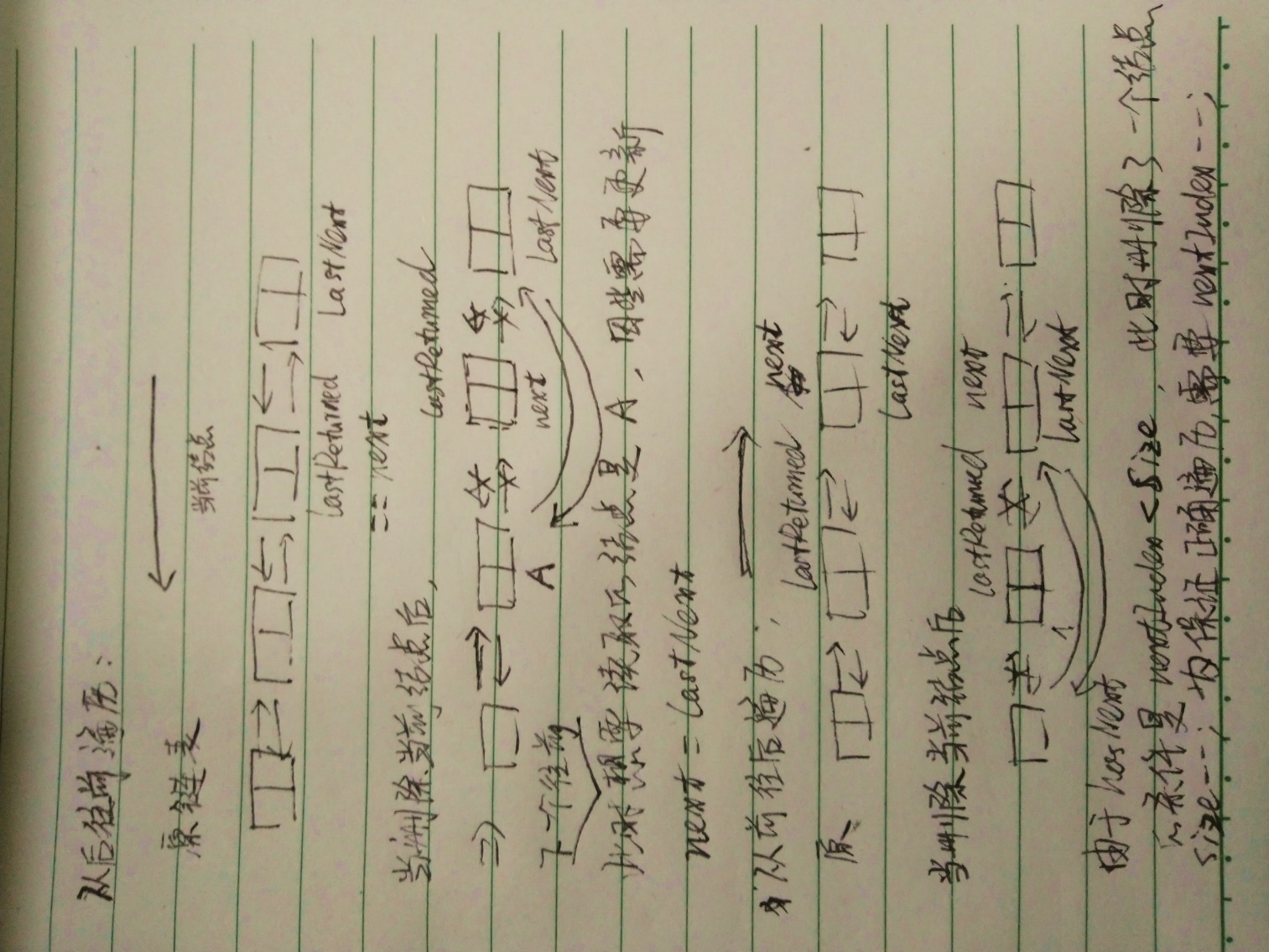
hasPrevious：nextIndex>0; previous()方法：检查exceptedModcount==modcount，lastReturned = next = (next == null) ? last : next.prev;如果next==null已经遍历到表尾，返回lastReturned=next=last；否则就为next.prev；nextIndex--；‘返回lastReturned的item；’这里的lastReturned和next是相等的，和next方法区别开来；

删除函数是重点:

①首先确认exceptedModcount==modcount

②得到当前节点的下一个节点，然后删除当前节点

③判断next == lastReturned



**5.Vector与Stack**

**5.1简介**

通过阅读源码发现Vector和ArrayList基本上是很相似的，只是Vector的增删改查方法上都加了synchronized关键字，所以是线程安全的，但是直接通过synchronized同步机制实现的线程安全会导致效率很低，所以基本上Vector很少被使用，下面就是Vector的增删改查方法和一起辅助方法的源码分析。

**5.2成员变量**

底层数组：elementData；元素个数elementCount；增长容量capacityIncrement

**5.3构造器**

5.3.1 **Vector(int initialCapacity, int capacityIncrement)**指定容量和容量增长的构造器，与ArrayList的区别

创建指定initialCapacity容量的数组；并设定增长容量

5.3.2 Vector(int initialCapacity)指定初始容量的增长容量0，到grow会发现，增长容量为0，扩容两倍，否则扩容oldCapacity + capacityIncrement

5.3.3 Vector()无参构造器，默认初始容量为10，增长容量为0

调用第二个构造器

5.3.4集合构造器Vector(Collection<? extends E> c)，将集合转为数组，如果不是object类型的数组就，就Arrays.copyof拷贝为object类型的数组进行elmentData初始化

**5.4增加元素方法**

5.4.1add(E e);调用ensureCapacityHelper对其进行grow准备，直接把元素放到数组表尾，这个ensureCapacityHelper与ArrayList里面的有些区别，ArrayList首先会调用ensureCapacityInternal里面判断是否是默认的空容量数组，这里没有必要；

5.4.2 add(index , element) 底层调用insertElementAt（element，index）

insertElementAt：判断index是否超过了elementcount；进行扩容准备ensureCapacityHelper

，移动元素，从index开始到结尾，从到index+1开始到后面，移动elementCount-index个元素；在index中插入指定元素；

5.4.3 addAll（index ， collection）将集合插入到指定index位置，集合c转为数组，然后记录集合c的长度，进行扩容准备，计算移动步数，elementCount-index；从index到结尾，移动到index+numNew的位置，移动numMoved个元素；

**5.5扩容方法**

与ArrayList一致；差别在于newCap的计算，如果capIncre>0;按照newCap=old+capIncre；

否则就按照new = 2\* old来扩容；

**5.6删除方法**

5.6.1删除指定位置的元素；remove(**int** index)

判断index的合理性；计算移动的元素个数；如果元素个数大于零；移动元素；将最后一个元素置为空，返回要删除的元素；

5.6.2 removeAll（object o）直接遍历每个元素置为null

5.6.3removeRange（from ，to）首先计算移动的元素有几个numMoved，然后从to下标开始到最后，移动到from下标位置，移动元素个数为numMoved；移动完后，从最后一个元素开始置空，计算to-from值，然后elementcount-（to-from），就是要删除的元素要停止的位置；

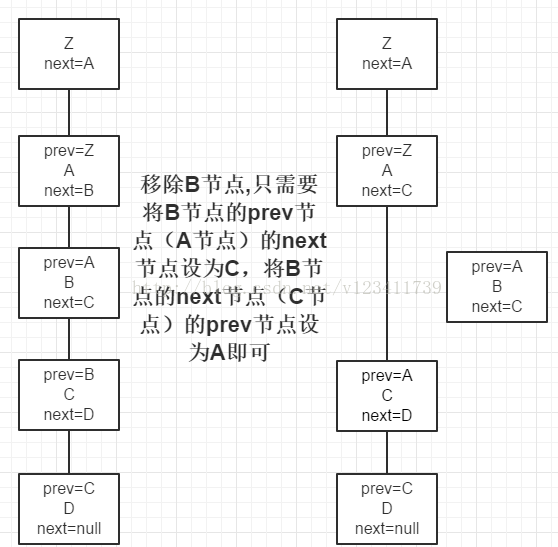
**5.7Stack过程分析**

将Vector的表尾当成是栈顶，然后push调用addElement方法；pop调用removeElementAt方法，都对表尾进行操作

**6.HashMap执行过程**

数组+链表+红黑树

红黑树的根节点不一定是索引位置的头结点；转为红黑树之后，链表的结构还在，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；红黑树结构中，叶子节点也可能有next节点，因为红黑树的结果与链表的结果不影响，红黑树的叶子节点，不一定是链表的最后一个节点。红黑树移除节点或者增加节点的时候会维护链表操作，这只是维持链表结构，但是红黑树的平衡还是需要用另外的方法调整的



引入红黑树的原因，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，红黑树的增删改查的时间复杂度能降到o(logn)

**6.1成员变量**

默认容量DEFAULT\_INITIAL\_CAPACITY 16；最大容量***MAXIMUM\_CAPACITY*** 2^30；默认负载因子 ***DEFAULT\_LOAD\_FACTOR*** 0.75f；链表转红黑树的阈值 ***TREEIFY\_THRESHOLD*** 8，**有可能转为红黑树，前提是满足大于*MIN\_TREEIFY\_CAPACITY64***；扩容时，桶中元素小于该值，转回链表 ***UNTREEIFY\_THRESHOLD*** 6；当哈希表中容量大于这个值时，才能转换为红黑树，哈希桶数组 table ，长度总是2^n，原因见第二点；node节点构成的set集合，entrySet；size 存放元素的个数，threshold扩容阈值，size>threshold时扩容；loadFactor负载因子；

**6.2 Node节点的定义**

继承了Map.entry；Hash，key，value，next

**6.3 TreeNode节点定义**

继承了LinkedHashMap的entry，这里原因暂时没搞懂；

节点含有：parent，left，right，prev，red；

**6.4 构造函数**

6.4.1 HashMap(int initialCapacity, float loadFactor)指定初始容量及装载因子

如果初始容量大于最大值，设为最大容量，装载因子不能小于0或者非浮点数，初始化装载因子和阈值，阈值是最接近initialCapacity的2的n次幂；

6.4.2 HashMap（int），调用HashMap(int initialCapacity, float loadFactor)，loadFactor为默认的loadFactor 0.75f，

6.4.3 无参构造器，只有装因子为0.75f，其他全是默认的；在resize函数中会介绍

newCap = ***DEFAULT\_INITIAL\_CAPACITY***; newThr = (**int**)(***DEFAULT\_LOAD\_FACTOR*** \* ***DEFAULT\_INITIAL\_CAPACITY***);

6.4.4 HashMap(Map<? **extends** K, ? **extends** V> m)，map放入hashmap中，装载因子默认0.75f，m中所有元素调用putMapEntry；

**6.5 put函数及相关方法**

6.5.1 putMapEntries(m, evict）所有集合元素放入hashmap中，evict为false表示初始化时调用该函数，其他时候调用，evict为true；

得到m的size，如果size大于0，并且table==null；使用ft=((float)s / loadFactor) + 1.0F，如果ft<(**float**)***MAXIMUM\_CAPACITY***)取ft，否则取最大值；赋值为t，此时threshold肯定比t小，因此将t作为*tableSizeFor参数，计算之后，得到新的threshold；此处也是保征初始化时，threshold一定是2的n次幂；因为第一次的threshold要给table.length*

如果s>threshold，进行扩容，此情况是table 不是null的时候；接下来遍历m.entryset;调用putval（*hash*(key), key, value, **false**, evict）

6.5.2 putVal（**int** hash, K key, V value, **boolean** onlyIfAbsent, **boolean** evict），onlyIfAbsent为true表示不修改节点值

①若table为初始化，或者长度为0，进行resize扩容并把长度设为n

②根据hash与n-1的按位与结果，确定在table中的插入位置，下标设为i，节点设为p

2.1如果p==null，进行插入操作tab[i] = newNode(hash, key, value, null);

2.2如果不为null，tab[i]不为空，有元素，考虑红黑树和链表：

2.2.1与table中的头结点p进行比较，如果p的hash，key都相等，直接赋值给**e**；

2.2.2如果不想等，判断p节点是不是树节点，如果是，按照红黑树进行插入，返回的是树中与插入节点相同的节点；

2.2.3如果不是红黑树类型，说明是链表类型，开始遍历，将p.next赋值为e，如果e==null，就在这个为null的位置，插入新节点，如果遍历的次数大于等于TREEIFY\_THRESHOLD – 1，可能转为红黑树，调用treeify(tab,hash);如果hash值与key都相等， 退出循环

③来到这一步，表明table，链表，树中已经存在与插入节点有相同hash和key的节点，（可能刚插入的，可能本来就有的），如果e不为null，如果onlyIfAbsent为false或者oldValue为null，就进行覆盖原值；

④返回旧值

6.5.3 putTreeVal(HashMap<K,V> ，红黑树的插入，同时维护链表的属性，即原来的next属性；先得到树的根节点；

①从根节点开始遍历，当前节点为p，先找hash值，如果p.hash值比形参h小，dir = 1表明向右，如果比形参h大，dir=-1表明向左找；如果hash值相等，并且key也相等，返回当前p节点；

②只会执行一次，因为找到了就返回，找不到，在找也没有意义；如果Hash相等key不等，就从p的左右节点进行查找，分别调用左右节点find方法，找到了就返回q；如果不能比较，没有实现接口或者其他，就采用自定义的规则进行比较。

③如果到这一步，上面的条件都没有满足，记录当前访问的节点 p赋值为xp；继续寻找如果dir<=0,将p指向左子树，否则右子树，最后如果遇到p== null，说明该树真的没有key相同的节点，进行插入操作，而插入位置就是作为xp的孩子；得到xp的next节点xpn，新建节点x，next指向xpn；如果dir小于0，xp.left指向x；否则right指向x；维护链表关系xp的next指向x；x的parent以及prev指向xp；返回null，

6.5.4 链表可能转为树结构treeifyBin(Node<K,V>[] tab, **int** hash) {

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

①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，方法结束

②如果满足最小树形限制，进行结构转换，并且头结点不为null；此时头结点赋值为e

③定义head，tail节点

④循环，把e转换为TreeNode，next指向null，赋值为p；如果tl为空，刚建立，树结构为null，hd指向p；否则就在尾节点插入，p的prev执行tl，tl的next指向p，将tl设置为新的p；知道表中的节点全部转换为树节点；但此时结构编程了双向链表结构。只不过节点变为树节点；让tab头节点=head；

⑤进行树型转换调用treeify(tab)

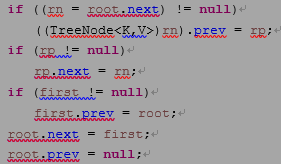
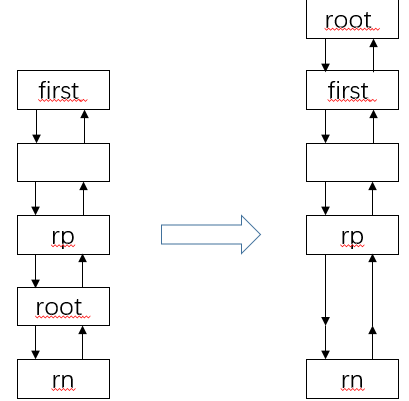
6.5.5链表转为红黑树treeify(Node<K,V>[] tab)

①调用此方法的节点开始遍历，当前节点设为x，定义next为x的next

②将x的左右孩子置为空，

③如果根节点为空，将x置为根节点，x的parent为null，颜色为黑色；

④如果已经存在根节点；，得到x的key和hash，然后遍历root为根的树结构，先根据hash值进行遍历，x小于ph往左找，否则向右找，hash一样之后，根据key来比较；然后把上一次访问的节点记为xp，知道p==null，xp就是p要插入的位置，根据dir的值，作为xp的左右孩子；然后插入调整平衡；如果根节点不在table索引位置头节点，就将其调整到头结点；

**6.6 getNode(int hash, Object key)方法**

通过长度-1&hash值，找到下标index ，然后找到头结点tab[index],判断头节点的hash值和key是否相等，如果相等返回，否则判断树节点还是链表节点，如果是树节点通过查找左右子树，如果是链表节点，遍历链表；

**6.7remove方法**

**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**

6.7.1 removeNode(**int** hash, Object key, Object value,

**boolean** matchValue, **boolean** movable)

①table长度大于0，头结点不为空，将p指向头结点；

②如果p的hash值和key与给定的相等，将node指向p；

③如果不相等；e指向p的next，如果不为null，判断是treeNode还是链表Node

④如果treeNode，，找到hash和key相等的节点，同样赋值给node；

⑤如果是链表的，遍历找到hash和key相等的节点，同样赋值给node，p为node的前一个节点；

⑥如果node不为空，说明key找到了要删除的节点，match为false表明不需要比对value；

⑦如果node是TreeNode，调用树的删除节点；如果node==p，直接头结点指向node.next；如果不是头结点，表明是链表的中的节点，p是前一个节点，直接p的next指向node的next；

6.7.2删除树节点removeTreeNode(HashMap<K,V> map, Node<K,V>[] tab,

**boolean** movable)

首先定义first指向头节点，然后得到当前节点的前驱和后继，如果前驱为空，说明next就是新的first和头结点，如果不为空，前驱指向后继；如果后继不为空则后继的前驱指向前驱，如果first为空，表明没有节点了，结束；得到root节点，然后判断删除后是否红黑树太小，如果太小，变为链表；剩下就是红黑树的调整了，暂不介绍；

**6.8扩容函数**

获得就容量old，保存当前阈值oldThr

分三种情况：

①原来表非空，size>threshold时被调用，

若旧容量已经超过了最大容量，就将**阈值设为Integer的最大值**返回；如果**容量翻倍**后小于最大容量，并且旧容量大于等于初始容量16，**newThr等于oldThr的两倍**；

②table为空时调用，oldCap小于等于0，oldThr大于零，表示用户调用初始容量和负载因子的构造器，此时table为空，oldCap=0，oldThr>0;

newCap就是oldThr；

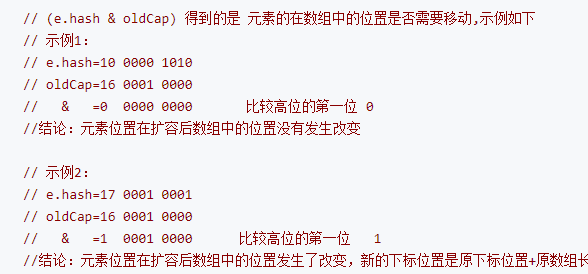
③table为空时被调用，oldCap<=0且oldThr小于等于0；用户调用无参构造器；

NewCap=16，newThr=16.\*0.75f

如果newThr为0；表明第二种，newThr = newCap \* loadFactor;

更新新的阈值，然后用新容量初始化newTab ，table = newTab；---------如果是初始化阶段，已经结束了，

如果oldTab不为空；遍历oldTab；如果只有头结点直接重新计算hash值，并赋值到newTab中，如果不只有头结点，分两种情况，头结点是树节点，用红黑树进行rehash，oldCap即bit，分别建立新树和老树，使用hash和bit进行与操作，如果为0，就放在老树中，如果不为0，就放在新树中，如果老树就将头结点index指向老树的loHead，因为节点少了，判断是否需要转为链表结构，不需要，就进行树形化；如果为新树，就将头结点index + bit指向hihead，判断是否需要转为链表，不需要就进行树形化；如果是链表结构，定义两个链表，老结点链表表明rehash之后还是原来的位置，new结点的链表rehash之后位置改变了；使用hash和oldCap进行与操作，如果为0表明rehash之后还是原来位置，放入old链表，否则位置为原数组长度+原下标位置，放入new链表，；两条链表建好了之后，如果old链表不为空，位置不发生改变，直接将指向old链表的头结点loHead赋值给newTab[j]，然后loTail的next置为空；如果new链表不为空，将指向链表的头结点赋值到newTab[j + oldCap];返回newTable；



这就是为什么要hash与length进行与的原因

既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了

**7.TreeMap执行过程分析**

**7.1简介**

使用TreeMap存储进行排序，我们自定义的类，必须定义自己的比较机制，一种是user类去实现comparable接口，并实现compareTo方法，另一种是mycomparator，写一个类实现comparator接口，实现comparable方法，将mycomparator作为参数传入到TreeMap构造方法中。

**7.2类名及成员变量**

比较器对象comparator，根节点root，集合大小size，静态内部类，用于表示节点类型，实现了Map.entry, 包含key，value，left，right，parent，color；

**7.3构造方法**

7.3.1 无参构造器

默认比较机制，comparator为null

7.3.2 自定义比较器的构造方法，实现comparator接口，作为参数传递

7.3.3构造map作为TreeMap，比较器仍然为空，调用putAll将map加到树中

7.3.4构造已知的sortedMap对象作为TreeMap

比较器使用sortedMap的比较器，调用buildFromSorted对有序集合，进行构造TreeMap；

**7.4查找方法**

7.4.1getEntry方法，如果比较器不为null，调用getEntryUsingComparator查找，如果key为null，抛出异常，如果比较器为null，调用key实现的comparable接口的compareTo方法与红黑树节点进行比较，小于0往左，大于0往右；getEntryUsingComparator思想类似，只不是采用的比较的规则不一样，调用compare方法

**7.5添加操作**

7.5.1putAll方法，参数为map

如果当前红黑树为空，并且map是sortedMap类型，且比较器相等，就会执行buildFromSorted 对有序集合构造红黑树，否则的话循环调用put方法进行添加

7.5.2从有序集合中构建红黑树buildFromSorted，底层调用buildFromSorted(0, 0, size-1, *computeRedLevel*(size), it, str, defaultVal);

其中还得计算完全二叉树的层数；从最后一个节点开始，采用除2减1的方式；计算层数。

有序集合构造红黑树过程：只有最后一层标注为红色，其余层都是黑色，lo = 0，hi=size -1；求出mid=（lo + hi）/2；构造左子树，如果lo小于mid；left = build（level+1，lo，mid – 1）；构造middle节点，key，value进行赋值给middle的key，value，构建middle节点，如果当前level==redLevel，即最后一层，标红；如果left不为空，就left和middle相连；

如果mid<hi，进行右子树的构建，right = build（level + 1，mid +1，hi。。），然后将middle与right相连，返回middle作为根节点

7.5.3 put（key ，value）方法

获取根节点，如果根节点为空，调用compare方法，比较key与key，compare方法，如果比较器为空，就使用compareTo，如果不为空，就是用比较器的compare；比较key与key，目的在于检查key不能为null；将key/value作为新的根节点；否则就是红黑树已经存在的情况：如果比较器不为空，就按照compare方法比较key大小，找到插入位置，否则按照compareTo方法找到插入位置；以key/value创建entry对象e，根据比较结果大于0小于0将e作为parent的左右孩子进行插入；然后调整平衡；

**7.6查找操作**

7.6.1getEntry（key）

如果比较器不为空，优先按照比较器来进行左右查找，否则按照compareTo，规则就是小于0往左，大于零往右，等于0返回；

**7.7 删除操作**

①如果删除节点是有两个孩子，找到后继节点，右子树最左，或者父亲是其父亲的左孩子的节点；然后替换被删除的节点， 进行删除

②如果删除节点只有一个孩子，如果是左孩子，就让待删除的节点的父亲的左孩子指向左孩子，否则指向右孩子，如果删除节点是黑色，进行调整

③如果没有孩子，且删除节点是黑色，进行调整，然后把其父亲的左孩子或右孩子置为null，由待删除的节点确定；

**8. TreeSet**

**8.1 成员**

NavigableMap m用于存放TreeMap，PRESENT用于作为key-value的value值，final不可变类；

**8.2构造方法**

传入无参TreeMap构造器，传入带比较器的TreeMap构造器，传入集合构造器，传入SortedSet构造器，这些构造器都是调用TreeSet(NavigableMap<E, object> m),因此m一定为会被初始化为Treemap类型

**9.LinkedHashMap执行过程**

**9.1概述**

在HashMap的基础上维护了一条双向链表，解决了HashMap不能随时保持遍历顺序和插入顺序一致的问题。LinkedHashMap对访问顺序提供了相关支持，在一些场景下该特性有用，比如缓存。在实现上，LinkedHashMap很多方法直接继承自HashMap，仅为维护双向链表覆写了部分方法；

**9.2特性分析**

①LinkedHashMap和TreeMap都实现了entry的排序，有什么区别：

--TreeMap按照key排序，而LinkedHashMap按照entry插入或者访问顺序排序

--LinkedHashMap保持entry有序方式是调整链表的before，after指针，而treeMap保持entry有序的方式是对tree结构的调整，因此显然LinkedHashMap代价小

②特殊的构造函数LinkedHashMap(int, float,Boolean)

--boolean = true;迭代器顺序遵循LRU原则，最近最少访问的entry会被最先遍历到，这种map结构非常适合构建LRU缓存

③removeEldestEntry（map.entry）

--通过覆写，可以实现：当添加新的映射到map中时，强制自动移除过期的映射。

--过期数据：

----双链表按插入entry排序，则为最早插入双链表的entry

----双链表按访问entry排序，则为最近最少访问的entry

④和hashmap的比较

--增删改查性能比hashmap要差一些，因为要维护双向链表

--迭代器执行时间长短

----LinkedHashMap和size成比例，HashMap和capacity成比例，因此hashmap相对比较费时，以为size<=capacity

⑤三个特殊回调方法

--afterNodeRemoval，删除节点后，双向链表中unlink

--afterNodeInsertion，插入节点后，是否删除eldest节点

--afterNodeAccess，访问节点后，是否调整当前访问节点的顺序

**9.3 属性**

头指针和尾指针，主要用到key、value、next、before、after，一个布尔型成员，表示accessOrder，表明是否按照访问顺序进行；

**9.4构造函数**

比hashmap多了一个设定访问顺序accessOrder的构造器，表明按照访问顺序进行

**9.5增加函数**

并没有重写什么添加函数，只是重写了两个方法，newNode和newTreeNode；这两个方法不在是创建Node了，而是创建LinkedHashMap.entry；然后通过linkNodeLast将新增的节点，链接在链表的尾部。获取到tail，将tail指向当前节点，如果last为null，head指向p，否则，p的before指向last，last的after指向p

**9.6回调函数**

9.6.1afterNodeAccess，当accessOrder为true时，将当前访问的节点e移到表尾，然后将e的before，指向e的after

9.6.2 afterNodeInsertion(**boolean** evict)，evict为false表示处于创建模式，只有在使用map集合作为构造器时，才会被设置为false；

该函数作用是，移除最近最久不访问的节点，即第一个节点；

需要三个条件:①evict为true②first！=null③removeEldestEntry为true，默认为false，如果需要实现删除，则要重写removeEldestEntry方法。

9.6.3将节点从双向链表中移除afterNodeRemoval(Node<K,V> e)

找到当前节点e的before和after，记录下来，然后e的before和after置空，将before指向after即可

**9.7查找**

9.7.1 get方法，在hashMap的基础上加入了访问顺序为true的情况，需要将e节点移到表尾

9.7.2 containsValue：HashMap中，通过先找数组，再找链表的方式，而LinkedHashMap直接通过head和after来寻找元素，效率较高。

9.8 LRU，通过linkedHashmap实现；

继承LinkedHashMap，定义一个缓存容量，调用构造器构造accessOrder为true的LinkedHashMap，设定缓存容量，重写removeEldestEntry方法，size》capacity的时候，进行LRU删除

**10．PriorityQueue执行原理**

通过小顶堆实现，可以用一个完全二叉树表示，保证每次取出的元素都是队列中权值最小的。元素大小可以通过自然排序或者比较器进行

**10.1属性分析**

默认初始容量***DEFAULT\_INITIAL\_CAPACITY*** 11，定义数组queue，元素个数size；比较器comparator

**10.2构造函数**

主要调用核心构造器，PriorityQueue(int initialCap, comparator)；为queue = new Object[]; comparator= comparator；无参构造器，使用默认初始容量11，comparator为null，初始容量构造器，定义初始容量，comparator为null；比较器构造器，初始容量为默认初始容量11，比较器为传入的构造器

10.2.1传入集合框架c的构造器

判断你是不是sortedset，如果是，调用集合框架的初始化方法initElementsFromCollection，并设置比较器；如果是优先队列的类型，就调用优先队列的初始化方法initFromPriorityQueue， 都不是就是普通集合，将比较器设为null，调用集合初始化方法initFromCollection

initElementsFromCollection 该方法先将集合转为数组，判断里面的元素非空，然后因为是有序集合，就直接将赋值给队列即可。

initFromPriorityQueue：如果c是优先队列类型直接赋值给队列。

initFromCollection如果是普通集合类型先调用initElementsFromCollection 方法，然后进行堆调整。

**10.3扩容机制**

扩容机制，当size>=queue.length时，会调用grow进行扩容

如果oldCap小于64，就让oldCap + oldCap +2;否则oldcap + oldCap/2

**10.4 添加元素**

Offer函数：如果size= 0；直接queue[0] = e；否则就插入到最后一个位置，调用siftUp向上调整堆；siftUp调用两个函数，分别用比较器进行判断和不用比较器进行判断，思路一致；

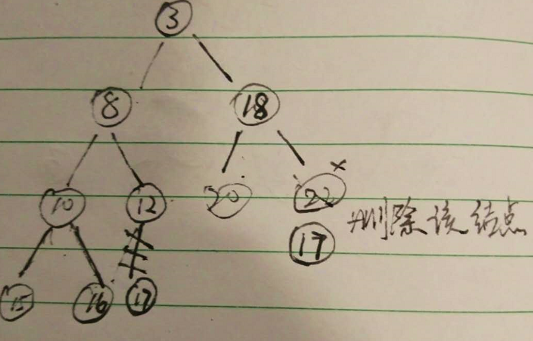
获取父亲节点，和父节点比较大小，如果x比较大，结束，否则将父节点和比较位置的值交换位置，然后父节点的下标赋值给k，k表示当前与父节点比较的位置；知道找到根节点或者父节点小于x的，停止

**10.5移除元素**

Poll方法，记录最后一个节点的值x，然后将其删除，调用siftDown从根节点开始，根节点的下一层开始，k表示当前节点的位置，2\*k +1得到左孩子下标，+1得到右孩子下标，如果左孩子较大，就将右孩子下标给左孩子，记录两者中较小的节点值c，比较c和x的大小如果如果x小于c，结束，否则，将c的值赋值给当前节点位置k的数组元素；k等于当前节点左右孩子较小的那个下标即child；知道找到x大于c的，或者k到达最后一个节点的父节点位置。最后将x的值赋值给queue[k]的位置

删除指定元素：

Remove(object o): 遍历找到o对应的下标，然后调用removeAt(i)方法；得到最后一个元素的值，然后从找到的下标i开始向下调整堆，说明，i为父节点的子树已经调整完毕

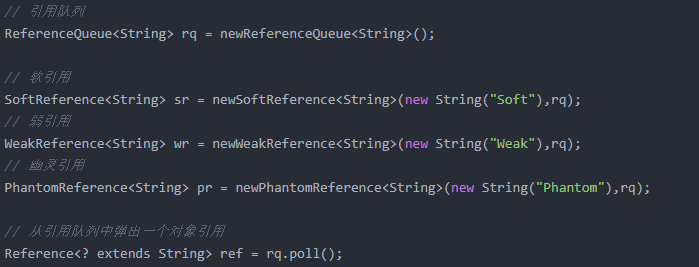


如果此时queue[i] == moved,说明向下调整不需要调整，此时需要向上调整，因为上面可能没有满足小顶堆。

**11.WeakHashMap执行流程**

**11.1java中的引用**

引用类型主要分为4种：**强引用**，永远不会回收被引用的对象，例如new出来的，**软引用**，有用但不是必须的，如果系统内存紧张，可能会被回收；**弱引用**，非必须对象，只能存活到下一次gc发生前；**虚引用**，无法操作对象



当引用的对象将要被JVM回收时，会将其加入到引用队列中。

**11.2 引用对列ReferenceQueue**

如果一个对象a，除了被b所引用之外就没有在别的地方存在引用，那么可以回收a；当a被回收后，我们就需要把b放入引用队列，即引用队列ReferenceQueue就是引用的监听器。

定义一个引用队列queue，定义一个object对象value，定义一个HashMap,key和value都是Object；for循环100次，每次生成1M的byte数组，然后定义一个WeakReference放入bytes和queue，最后将WeakReference，和value一起放入map中；此时如果bytes数组被jvm回收，那么weakReference就会被放入到引用队列中。

(k = (WeakReference<byte[]>) referenceQueue.remove()这个就可发现weakReference确实被放入了referenceQueue中。

**11.3成员变量**

引用队列queue，静态内部类entry，继承了weakReference类，实现了Map.Entry接口

包含value，hash，next，和一个构造器，该构造器调用父类即weakReference的构造器，将Object key，和ReferenceQueue queue与weakReference进行连接；这样一旦key被回收了，当前的entry就会被加入到queue中。

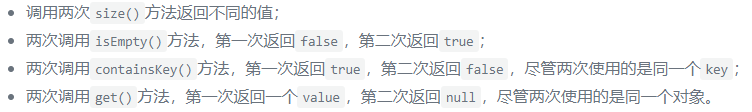
解释一下为什么，对象回收之后，map中对应的K，V也会被移除

expungeStaleEntries 与这个方法有关，GC之后，queue就会存着entry；

循环调用queue.poll方法将其赋值给x；按照先进先出的规则；将得到的x转为Entry<K, V>e，然后根据e.hash值和table长度，计算下标值，知道与e相等的节点，然后进行删除操作；这样就能够将队列中的entry与map中相等的entry一起移除了。

什么时候会调用expungeStaleEntries 方法，获取size的时候，扩容的时候，获取元素get的时候，插入元素put的时候。

WeakHashMap会存在一些问题：



**12. Hashtable执行过程**

**12.1Hashtable与HashMap的区别**

Hashtable与HashMap都是存放键值对，结构是数组加链表，1.8的HashMap多了红黑树；HashMap允许存放一个key为null的键值对，可以存放多个value为null的键值对；Hashtable不允许key为null，value为null的键值对；扩容方式不一样，hashtable中数组默认11，扩容方式2\*old +1，HashMap数组大小默认16，扩容是原来的2倍，hashtable大部分使用synchronized修饰，证明Hashtable是线程安全的。

**12.2成员变量**

每个entry都是单向链表的表头，entry数组 table，count表示表中entry的数量，threshold表示表的阈值，如果count>=threshold，进行扩容；

**12.3 构造器**

**MAX\_ARRAY\_SIZE** 为Integer.MAX\_VALUE-8;

12.3.1初始容量，负载因子，如果初始容量为0，将其置为1，然后new Entry[initialCapacity]；threshold = initialCapacity\*loadFactor与**MAX\_ARRAY\_SIZE** +1后的最小值；

12.3.2指定初始容量构造器；默认使用initialCapacity，0.75f；

12.3.3无参构造器，默认构造11, 0.75f

12.3.4 插入集合构造器；初始容量为 2\*集合长度与11的最大值，0.75f；调用putAll将集合插入；

**12.4查找方法**

Contains：双重循环，先循环tab头结点，再循环链表，

ContainsKey：通过key调用hashcode，然后与32位的0x7FFFFFFF,其实与hashmap的作用一致，避免出现负数，让hash值较为平均分布；找到tab的位置，然后遍历判断key和hash相等的元素；

Get();比containsKey多了一步，返回value值

**12.5扩容方法**

得到oldCapacity，newCapacity = oldCapacity \* 2 + 1；最大只能到MAX\_ARRAY\_SIZE，如果超过了就赋值为MAX\_ARRAY\_SIZE；构造newCap的数组，取threshold=newCap\*loadFactor和MAX\_ARRAY\_SIZE+1的最小值；扩容完成后就进行数组元素的赋值，遍历原来的Map，重新计算index下标，然后往头结点进行插入；每次插入都放在头结点；

**12.6增加元素**

AddEntry，判断count>=threshold，进行rehash操作，然后计算hash值，和下标；依然是头插法，得到头结点，然后new一个entry，key，value是带插入的值，然后next指向头节点，赋值为新的头结点。

Put方法调用AddEntry;注意如下：

此处key和value都不能为null，如果value为null，直接空指针异常，如果key为null，调用key.hashcode也会抛出空指针异常；首先找index，在找该链表下是否有相同key值的，如果有，修改原值，没有就调用addEntry；

PutAll就是调用put方法；

**12.7删除元素**

Remove key找到头节点，头结点设为e，定义一个prev节点，表示上一个访问的节点，一次循环结束后，prev指向e，e指向e的next；如果e的hash和key都相等；prev的next指向e的next或者头结点指向e的next；

Remove key，value；必须key和value都相等才能删除