# 线性表

## ArrayList

### 概述

1. 底层通过数组实现；
2. 数组大小size无需是2的幂；
3. 删除和添加操作都会造成数组内的元素批量移动（删除末尾元素或者在末尾添加元素除外）

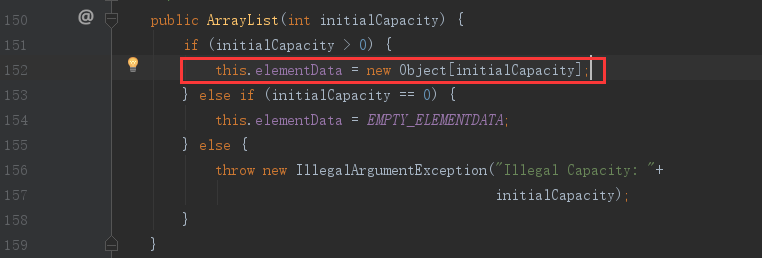
### 构造方法

1. 默认构造

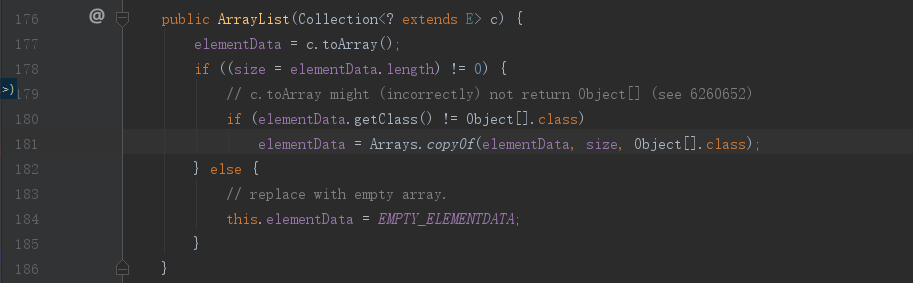


通过代码看出，如果没有设置size，则默认构造产生的是空数组。

1. 设置数组大小



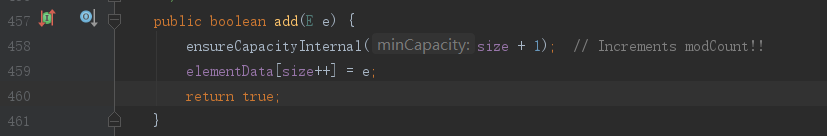
1. 拷贝构造



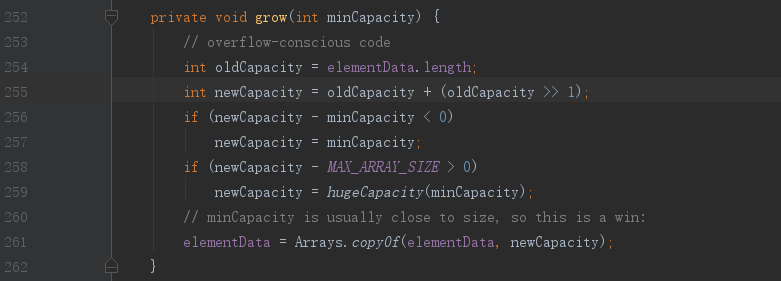
### 重要操作（增删改查）

1. Add操作

Add操作分两种，一种是传递一个元素，直接add到数组末尾；一种是传递需要add元素的index

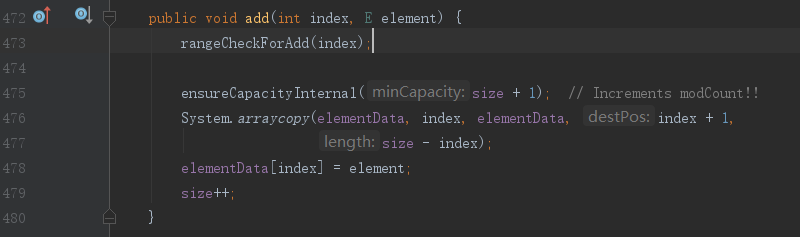


其中ensureCapacityInternal确保将元素存放在末尾不会越界，如果要越界，则调用grow方法将数组长度增大，增大为原来capacity的1.5倍，代码如下



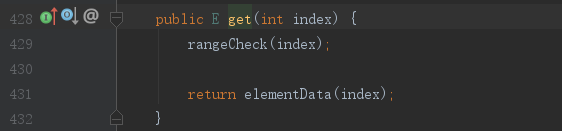
第二种add的思路和第一种类似，首先也要判断index是否越界；

值得注意的是，这种方式可能会造成批量元素往后移动（第476行）。

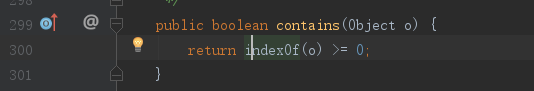


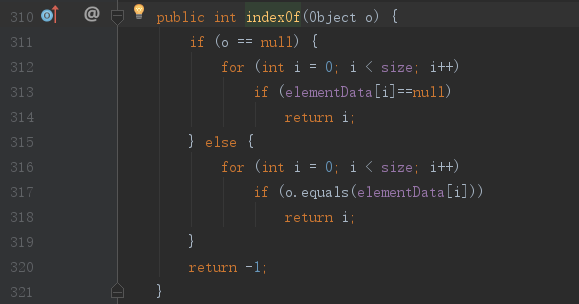
1. Get操作/Contains操作

Get操作是通过index获取；



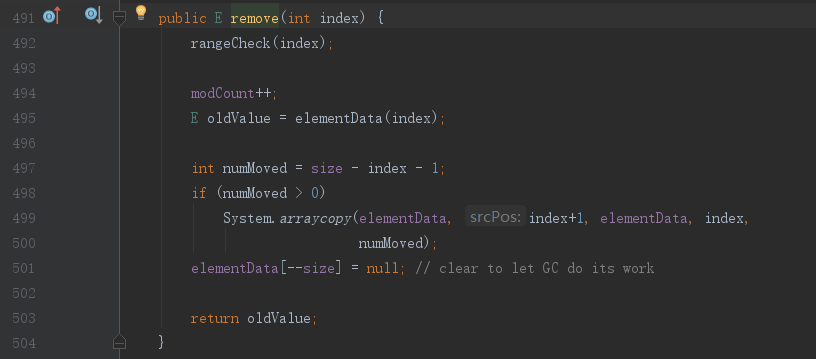
Contain操作通过元素判断是在存在：



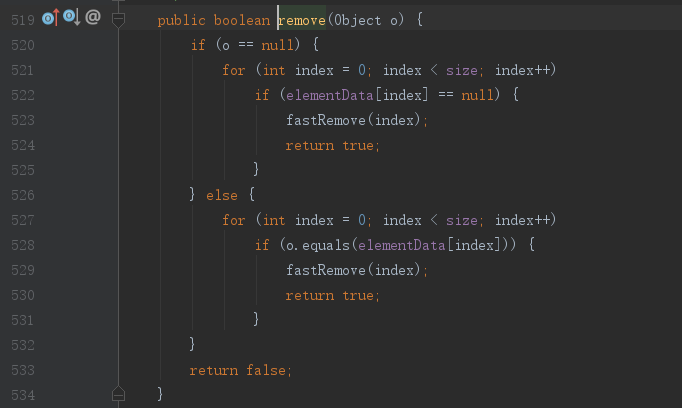


1. Remove操作

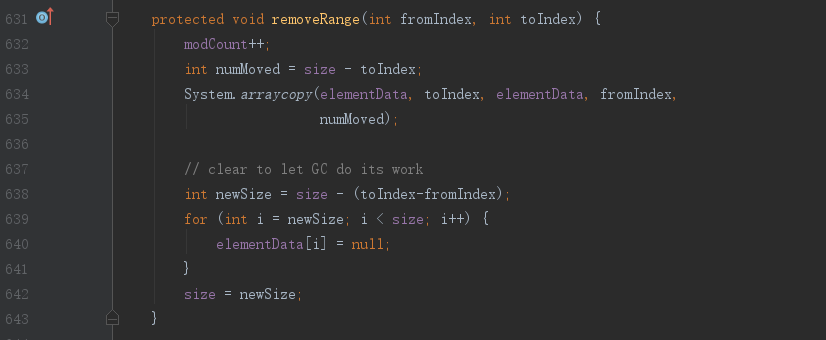
对应的，remove操作也有两种形式，一种是传递index，一种是传递元素



删除指定元素，通过代码可以看出，如果存在多个与被删除value相同的元素，只删除最靠近数组头部的元素即可，并不会将所有相同元素删除。



第三种就是删除给定范围的元素



该方法没有做越界处理。

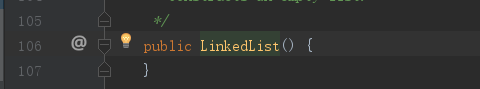
## LinkedList

### 概述

1. 双向链表；

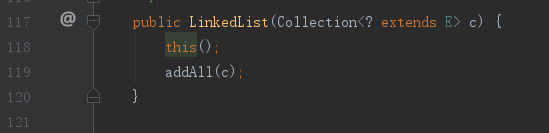
### 构造方法

1. 默认构造方法



无任何操作。

1. 拷贝构造方法



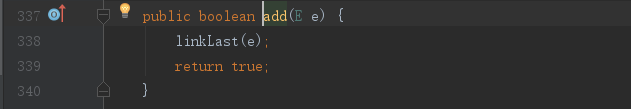
将集合c中的所有元素添加到链表中

### 重要操作

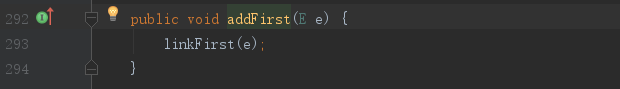
1. Add操作

Add操作分为两种类型，一种是在头部插入节点，一种是在尾部插入节点。默认情况是在尾部插入节点。

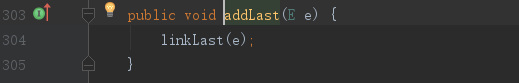
默认情况



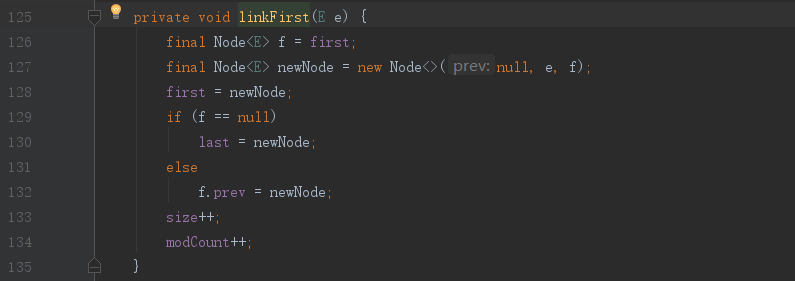
在头部插入

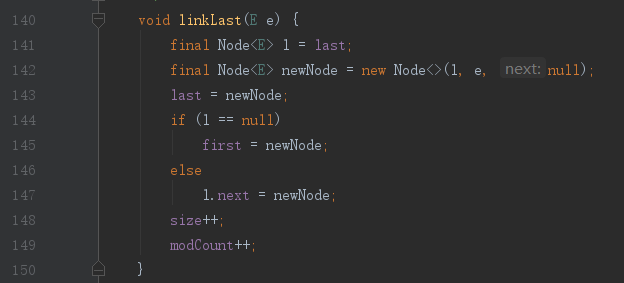


在尾部插入



重点是linkFirst和linkLast方法，方法思路都是常规算法，无需多讲。

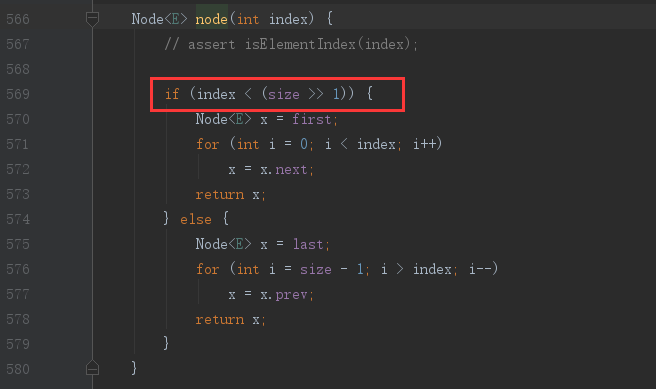




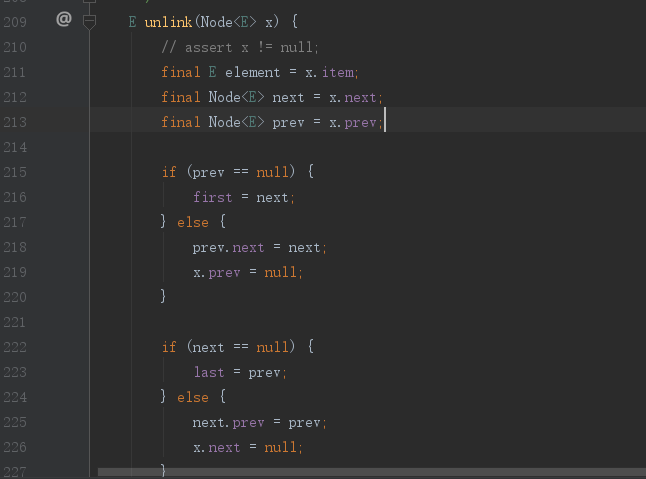
1. Remove操作

根据下标删除对应节点。首先第一步是定位到index对应的节点，此处有个好的思路就是如果index大于当前节点数size的一半，则从后往前遍历，否则从前往后遍历。

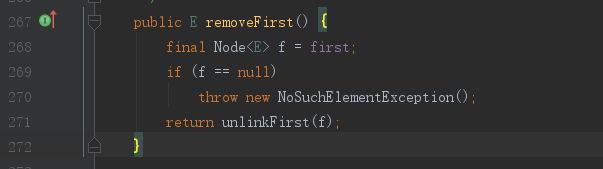
定位代码：

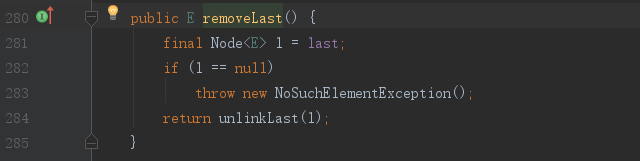


删除代码（常规思路）：

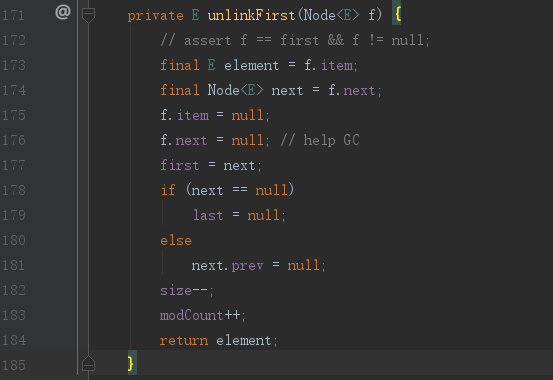


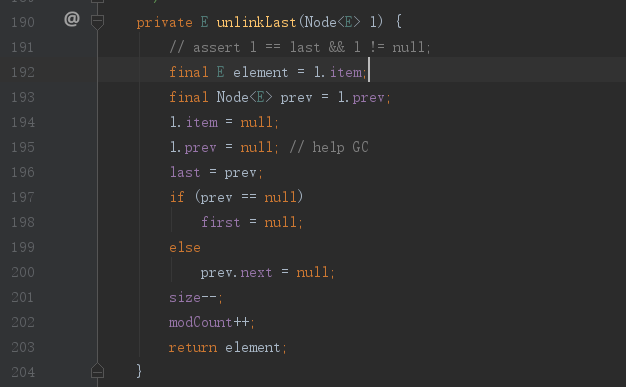
极端情况，删除头部或者尾部节点（也就是上面代码index=0，或则index=size-1）；





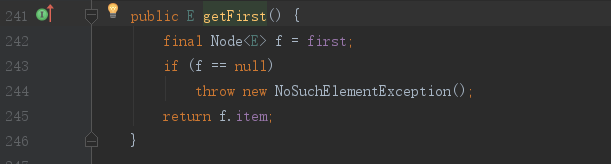
算法的难点是要考虑到删除节点后first，last为null的情况

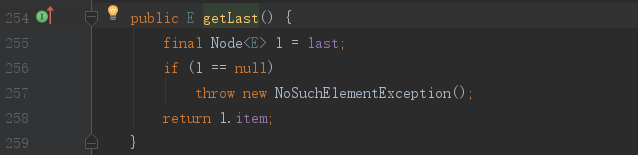


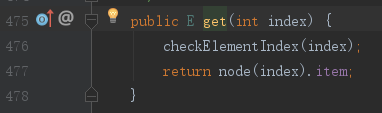


1. Get/Contains操作

Get操作也对应三种情况，获取头部节点，获取尾部节点和获取链表中部节点。







# Map系列

## Map接口

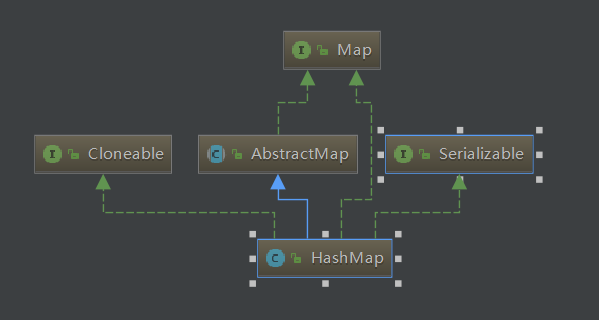
定义一系列的Method，便于其他map类实现。

### 子接口 Entry

用于描述一个<key,value>对。

## HashMap

以下代码截图都来自于jdk1.8



### 概述

1. 底层利用数组存放每一个<key,value>对(Entry)；
2. 默认capacity为2^4=16；
3. 最大capacity为2^30；
4. 默认负载因子0.75
5. Threshold = capacity \* loadfactor，如果size大于Threshold将进行扩容；
6. 数组默认为空
7. HashMap的capacity始终为2的幂
8. 通过hash计算桶的index时不是通过取模，而是计算hash&（capacity-1）的值；
9. 链表转红黑树的阈值为8，也就是一个“桶”中的Entry个数大于8之后，将由原来的链表结构转为红黑树结构；
10. 反之，红黑树转链表的阈值为6，也就是如果红黑树的节点减少到6时，则转换为链表结构；
11. 如果存放的键值对中的key是null，则hash之后的index为0，也就是说HashMap运行存放key为null的键值对，同时value也可以为null
12. 如果事先知道map所需要的桶的个数N，在HashMap初始化时，可以设置其capacity为N/0.75，这样可以防止HashMap扩容带来的性能开销；

### 节点元素

#### Node

1. 链表节点；
2. 继承自Map.Entry<K,V>；

### 构造方法

1. 传递初始capacity和负载因子



第456行计算大于等于initialCapacity并且为2的幂的数（这个方法）。其他构造方式都是调用该构造方法。



通过构造方法可以得知，在new一个HashMap对象时，并没有为table分配内存，只是设置相应的capacity，负载因子等参数。真正分配内存的是在put操作中。

1. 拷贝构造

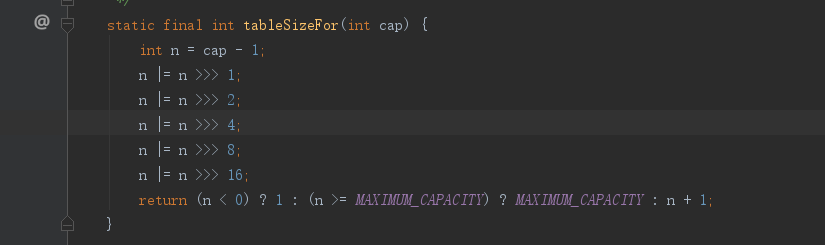


第510行，如果拷贝的map的size大于默认大小，则扩容，扩容的思想就是将内存扩大为原来的两倍，同时把原来的数据从旧地址，“复制”到新地址，“复制”的时候需要重新计算hashcode并计算新的index（详见10）

### 重要操作（增删改查）

1. 计算初始capacity

如果用户在创建HashMap时，capacity的初始值cap不是2的幂，需要计算不小于cap且是2的幂的数。



第一步，弄清楚无符号右移运算符（>>>）;

第二步，n|=n>>>1等价于n=n|(n>>>1)，运算的结果是将n的二进制数中最高的两位变为1；

第三步，n|=n>>>1将n的二进制的最高四位数设置为1；

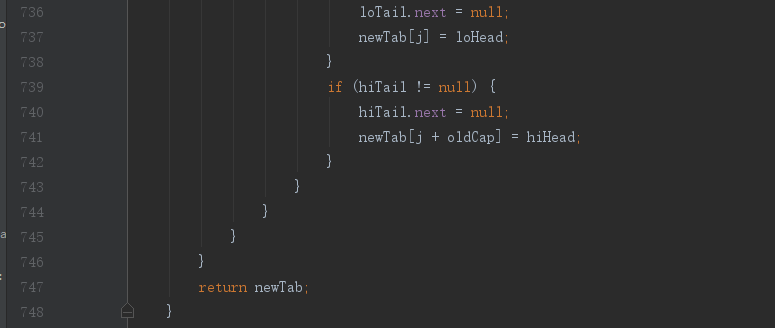
……

最后一步，n的二进制全部为1（1的个数和初始n的有效位一样），所以n+1即为不小于cap且是2的幂；

1. Resize操作（扩容操作）







扩容思路，677~704行都是准备阶段，判断是否能扩容，703行申请新的内存空间进行扩容：

第一步，如果当前HashMap已经达到Max\_capacity，无法扩容，直接返回原有table；

第二步，扩容完成后将原来的entry计算新的index，拷贝到新的空间（一个for循环中嵌套while循环）；如果当前桶中元素是树形结构（红黑树），则进行拆分（713行）；如果是链表结构，则遍历每个阶段重新计算每个节点的位置；

在718~734行中用了一个小技巧，此处并不是逐个节点计算新的index并放入新的newTab中，而是将其分为两类（假设oldcap的二进制表示中1所在位为a），一类是节点的hash值的二进制表示中，位a为0，一类是位a为1，这两类节点通过新的计算，同类型的节点的新index是一样（一样的原因是HashMap在计算index 时，index=hash&（capacity-1），也就是同一个桶中所有entry的hash值的二进制表示中的低位都一样）。几举个简单的例子，假设oldcap=4，则index=3的桶中所有节点的hash值二进制表示中，最后两位都为1，则存在hash为11,111，1011,1111（二进制表示）等；由于新的newcap=4\*2=8，则在新的tab中，11，1011（因为11&7=1011&7……）是同一类，111,1111是同一类，也是从右往左第三位上面为0的是一类，为1的是一类，对应代码720行）；

第713行的拆分桶718~734行的思路类似，先将树结构拆分，同一个桶中的节点拆分为两个链表，如果某一个链表的节点数小于等于阈值UNTREEIFY\_THRESHOLD，则直接并入到新的newtab，如果大于阈值UNTREEIFY\_THRESHOLD，则将链表转换为红黑树结构再并入新的newtab

1. Get操作（查）



第一步计算hash，第二步定位到index对应的“桶”，如果第一个元素是需要get的对象则直接返回，如果不是，则根据桶中对象的存储结构查找，如果是红黑树，则调用getTreeNode，否则遍历链表查找

1. Put操作（增、改）

Put操作时，如果需要put的键值对的key已经存在，则将旧值替换为新值，并返回旧值（两种存储方式都采用这种方式）。如果不存在，则根据存储结构put，如果是链表，则存放在链尾，同时判断增加元素的节点数大于链表转红黑树的阈值，大于则跳转；如果存储结构是红黑树，则按照红黑树的算法插入到指定位置。

通过阅读Put操作的代码可以得知，Put操作其实对应数据结构的“增、改”操作，因为如果存在一样的key，则用新的value覆盖旧的value。





基本思路：首先判断table是否为空，如果为空则申请指定的capacity大小的空间，然后判断index对应的桶是否为空，如果为空则直接存放key,value对，否则判断该桶中第一个元素是否和需要put的键值对的key一样，如果一样，则将原来的value修改为新的value，同时返回旧的value（第652行），如果桶中第一个元素不一样，则根据此时的存储结构做不同的put操作，如果是红黑树存储，则调用putTreeVal，否则遍历链表并插入到尾部。如果index对应的桶中没有key，则返回的是null

1. Remove操作（删）





818~837行定位到需要remove的节点。

第840行表示如果删除的链表头部节点，第842表示删除非头部节点，此时p指向node的父节点。

1. Clear操作



## HashTable

### 概述



示意图

1. Hashtable的capacity不需要为2的幂
2. New一个hashtable对象时则为其分配内存；
3. Hashtable所有的操作前都有synchronized关键字，因此hashtable是线程安全的；
4. 桶中的所有节点都通过单链表结构存储；
5. Hashtable的key和value均不能为null

### 构造方法

1. 传递capacity和负载因子



通过第190行可知，构造hashtable时则为其分配内存。

1. 拷贝构造



### 重要操作（增删改查）

1. 查询是否包含某个value（查）



单链表遍历算法

1. 查询是否包含某个key（key）



1. 获取某个键值对（查）



1. 扩容



扩容是指将内存大小扩大为原来的2倍，同时重新计算hash值并存放在新的内存中。第407~416行为原有的键值对重新存放在新的内存，链表插入操作是在链表头部插入。

1. Put操作（增改）







疑问：为什么第458行没有判断key是否为null

468~474判断是否已经存在相同的key，如果存在则用新的value替换旧value，同时返回旧的value。

第435行，new一个Entry对象，同时设置其next为原来的链表头部。

1. Remove操作



单链表删除节点操作，思想很简单，双指针同步移动。

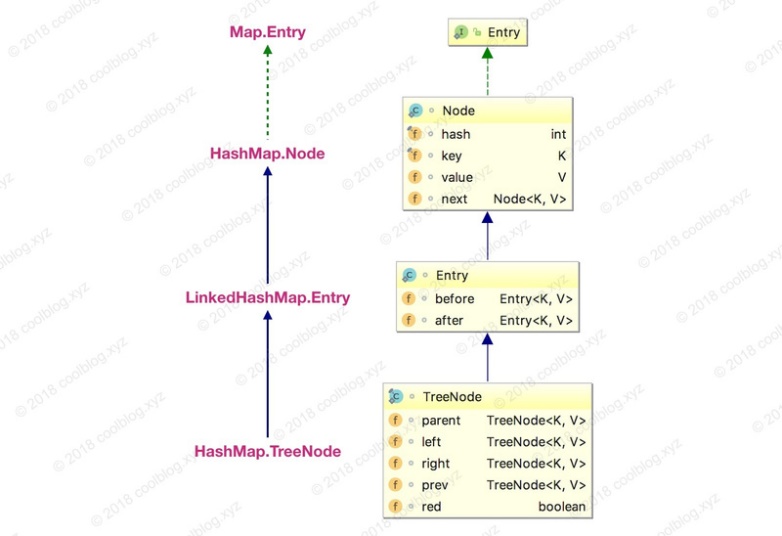
## LinkedHashMap

### 概述



示意图

1. 继承自HashMap，在HashMap原有的结构基础上为每个节点增加前驱before和后继指针after；
2. 同理，LinkedHashMap也是非线程安全的；
3. 原有的HashMap结构没有改变，通过before和after指针维持双向链表的逻辑关系；
4. 标志位accessOrder，默认值为false；如果accessOrder的值为false，则表示按照元素插入顺序维持双向链表；如果accessOrder值为true，则按照访问顺序遍历（也就是最近最少使用算法LRU）
5. 双向队列通过head和tail指针指向其头部和尾部，同时如果某个节点被访问了，则将该节点“移到”双向队列的tail指针指向的节点（这里的移到只是修改前驱和后继指针的指向，并没有改变物理地址）
6. LinkedHashMap和HashMap中节点间的继承关系，注意HashMap中的treenode继承自linkedHashMap中的entry，在使用HashMap时，节点较多时则用红黑树结构存放，而treenode中多余的两个before和after引用则是多余的。



1. 参考; https://segmentfault.com/a/1190000012964859

### 构造方法

1. 默认构造



1. 传递capacity参数

如上图第358行

1. 传递capacity和loadfactor参数

如上图346行

### 重要操作（增删改查）

1. 双向链表的创建过程（增，改）

LinkedHashMap继承自HashMap，没有重新put操作，因此调用put方法时执行HashMap中的put方法代码，但是override了newNode方法，因此在new一个节点时，不同结构调用不同的newNode方法





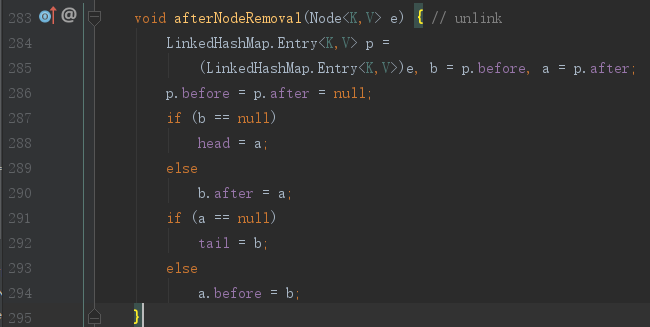
第222~231行创建双向链表，把最新put进来的元素存放在链表末尾。算法分两种情况，如果插入的是第一个元素，此时tail为null，因此需要设置head（226行），否则，将p节点存放在tail后下一个地方，tail指向p。

如果put操作的key已经存在，同时accessOrder=true，除了将旧的value替换为新value之外，还需要将这个节点置换到双向链表的末尾，表明该元素最近被访问了。



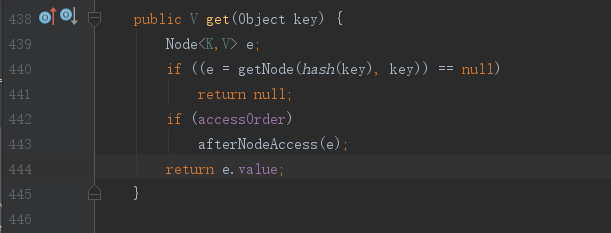
1. Remove操作

执行remove操作，除了原有的HashMap操作外，需要修改双向链表的指向问题。



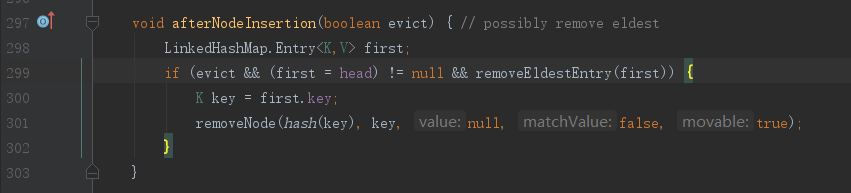
1. Get操作（查）

Get操作override了HashMap中的get操作。



如果accessOrder为true，则需要调整双向链表的顺序（afterNodeAccess方法），将最近访问的节点移到tail处。

1. afterNodeInsertion操作



这个方法在一般情况下并不会被执行，如果override了removeEldestEntry方法，可以实现LRU算法（最近最少使用算法）。也就是删除head指向的节点。比如实现一个新的class继承自linkedhashmap，并且设置双向链表的最大节点数，如果节点数大于该值，则删除最少被使用的节点（head指向的节点）。

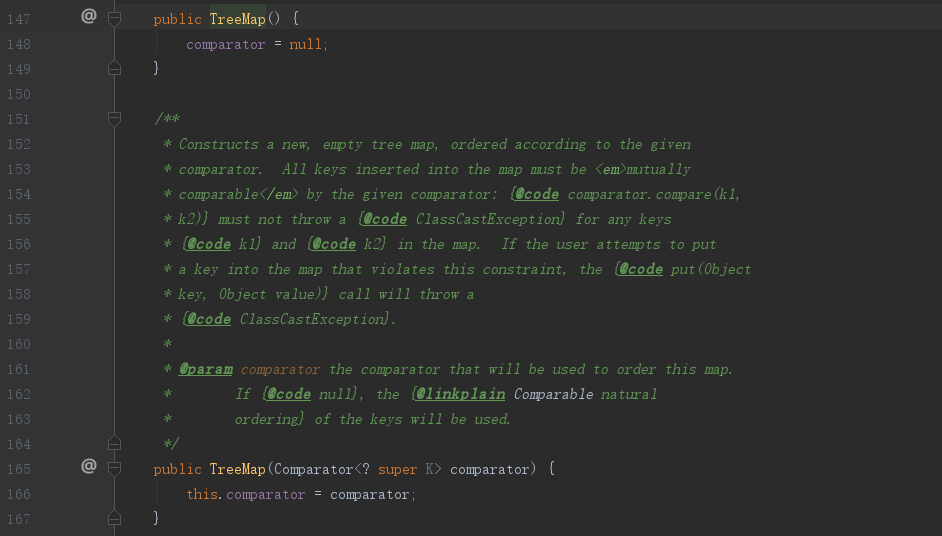
## Treemap

### 概述

1. 底层实现基于红黑树；
2. 入口是root指向的根节点；
3. Key不能为null

### 构造方法

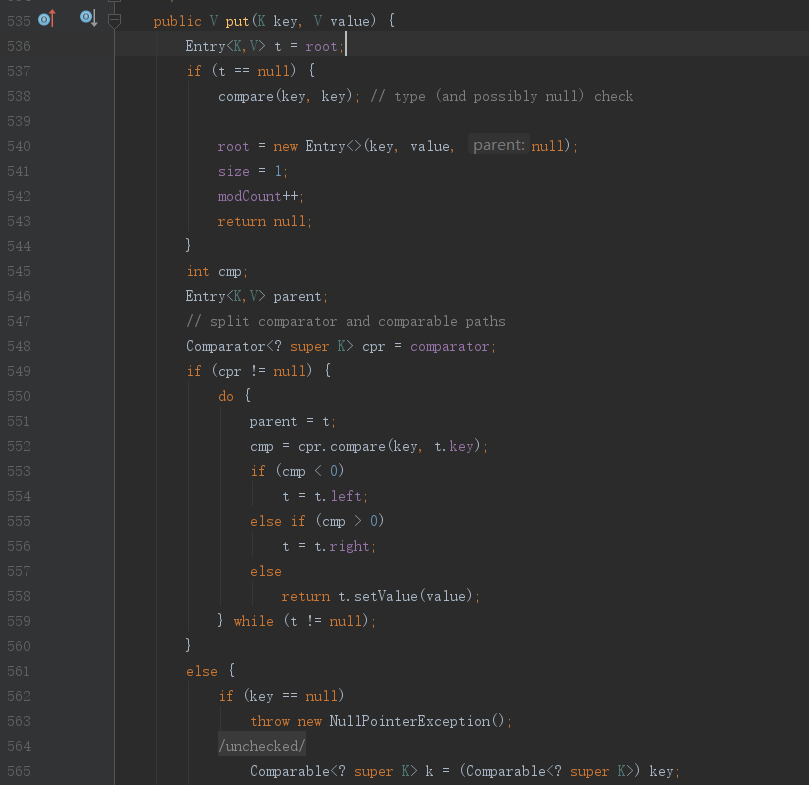
初次构造对象时，无需分配大量内存，只有在put操作时new一个Entry即可，因此构造方法如下。

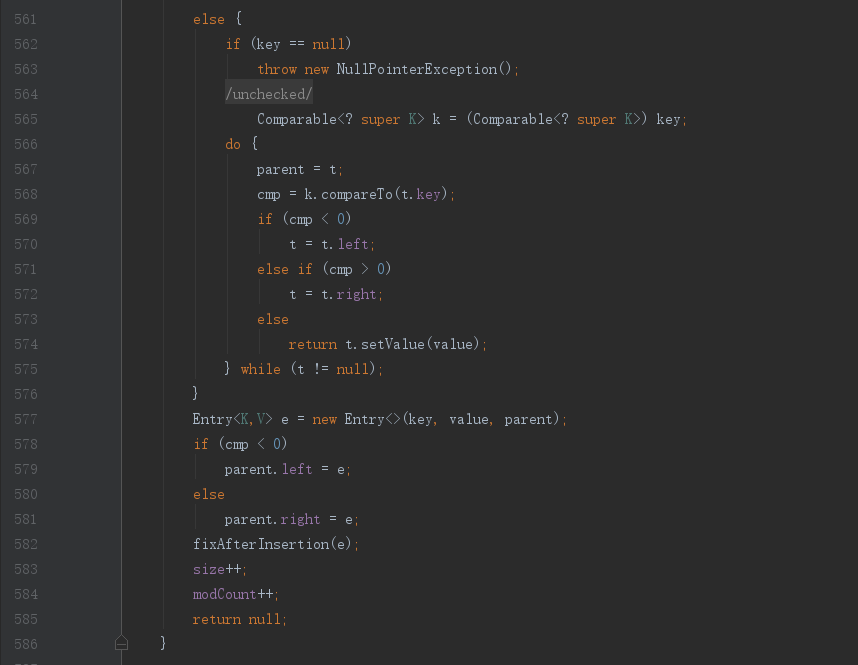


其中compartor用于判断各个节点的大小关系。

### 重要操作

1. Put操作（增、改）





思路是：

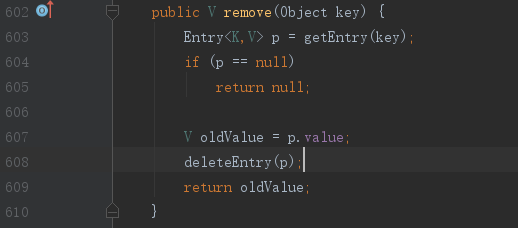
首先判断root是否为空，为空则new一个节点然后直接返回；

然后，第548~560行，比较算子compartor不为空，定位需要插入的节点的位置（搜索树的特点是左子树节点小于根节点，根节点小于右子树节点），如果需要插入的key已经存在，则替换旧的value，否则需要插入的节点是变量parent指向节点的子节点；

第578~581，根据cmp的大小判断新插入的节点e（新创建的节点默认为黑色Black）应该位于parent的左子树还是右子树；

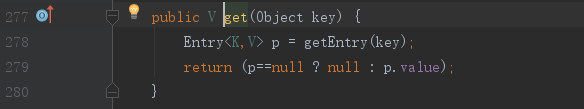
插入节点后需要对红黑树进行旋转，使之符合红黑树条件；（有关红黑树的调整算法参考维基百科）

1. Remove操作

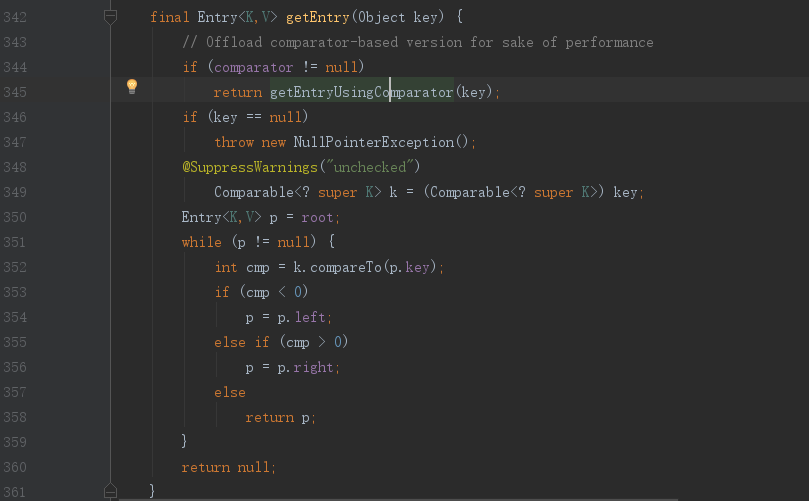


根据key找到对应的节点，然后将其删除；

1. Get操作



重点是getEntry方法



算法思路和平衡二叉树的查找算法一样。