# 线性表

# 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. 数组默认为空
6. HashMap的capacity始终为2的幂
7. 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；
8. 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；
9. 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

### 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

### 重要操作（增删改查）

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。

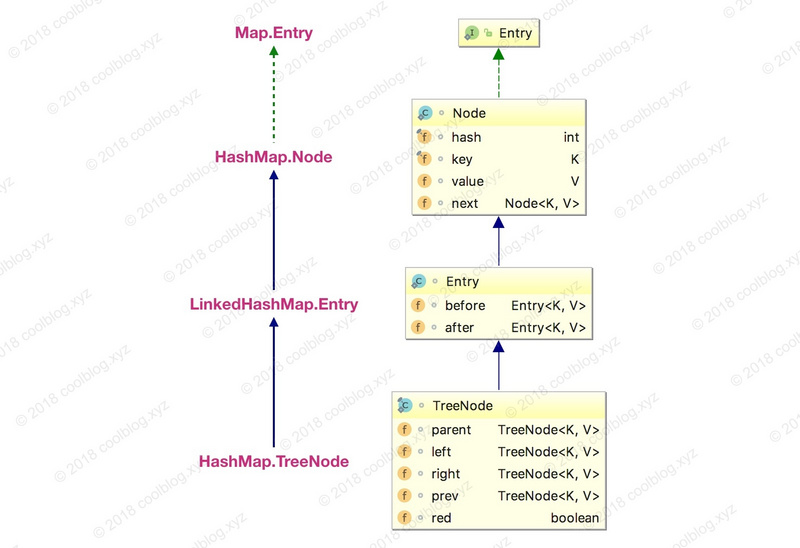
## LinkedHashMap

### 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





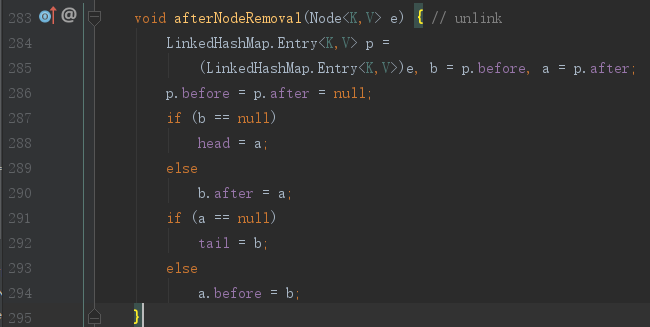
第222~231行创建双向链表，把最新put进来的元素存放在链表末尾。算法分两种情况，如果插入的是第一个元素，此时tail为null，因此需要设置head（226行），否则，将p节点存放在tail后下一个地方，tail指向p。

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。



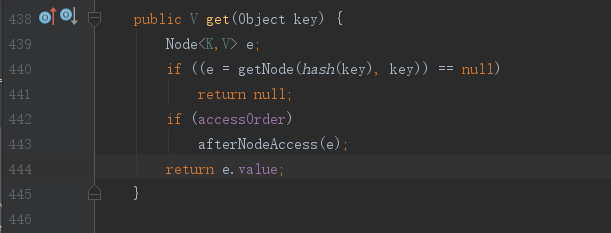
1. Remove操作

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。



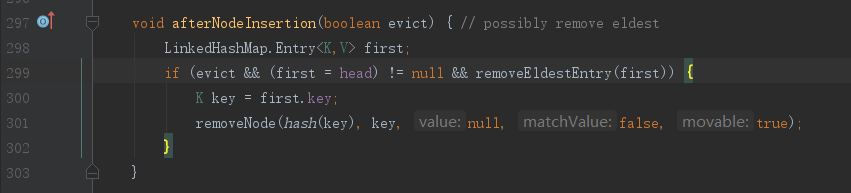
1. Get操作（查）

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）。