# 集合

以下所有代码都来自于JDK1.8.

## 概述

Collection层级关系，本例子以JDK1.8为Demo。

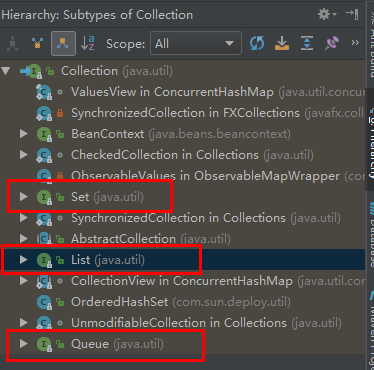


Figure 1 collection集合类包含关系

重点掌握Set，List，Queue这三个具体集合。

## Set层级关系

Set主要是TreeSet、HashSet，LinkedHashSet

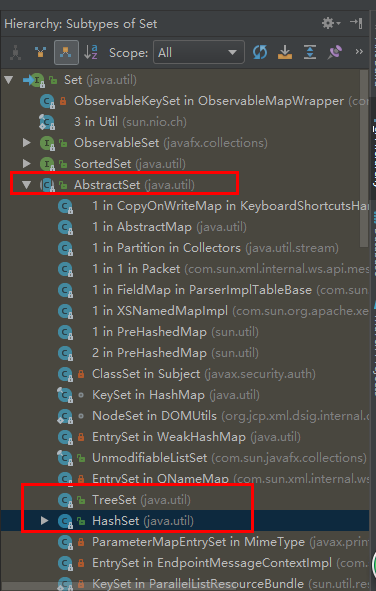
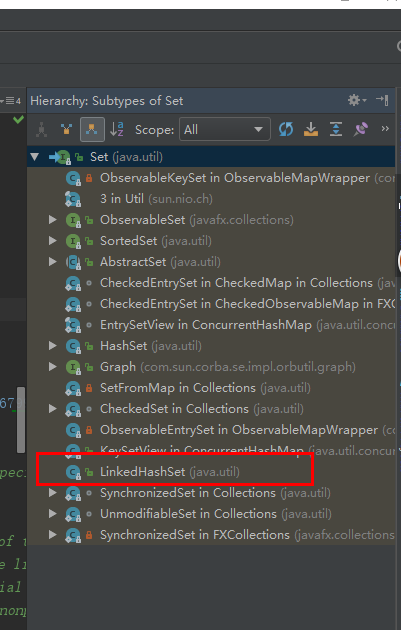


Figure 2 Set接口实现



## List层级关系

List接口具体实现主要学习ArrayList，Vector、LinkedList、CopyOnWriteArrayList

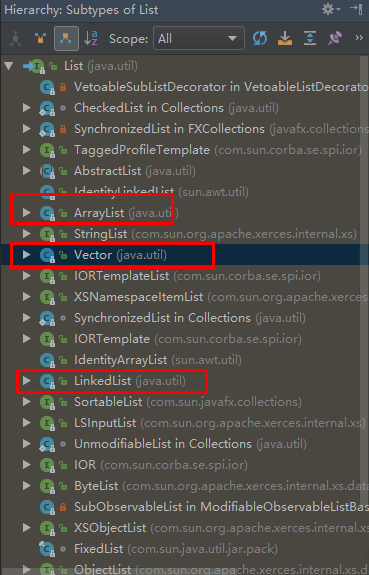


Figure 3 List接口实现

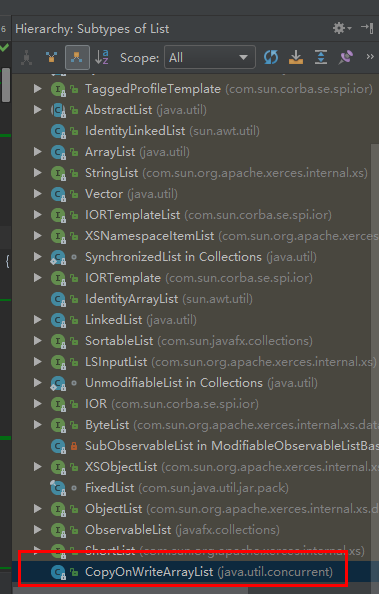


Figure 4 List接口实现

## Queue层级关系

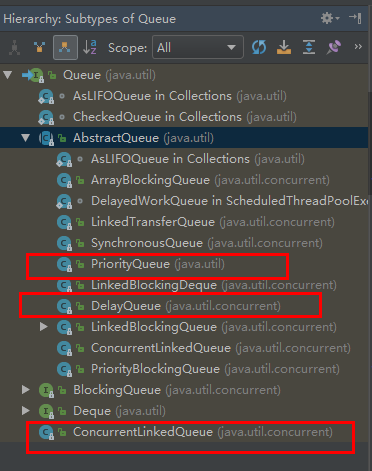


Figure 5 Queue接口实现

# Set结构

在Java集合中Set和Map几乎是一对（成对出现），TreeSet是通过TreeMap实现，HashSet是通过HashMap实现。

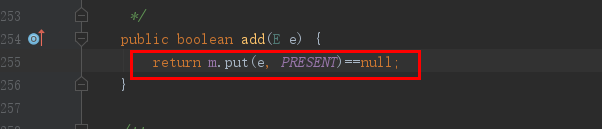
## TreeSet

### 概述

1. TreeSet内部包含一个TreeMap对象m，所有元素存被当做TreeMap的Key，TreeMap的value是一个常对象PRESENT
2. 几乎所有的操作都是通过m进行操作，TreeSet等同于m的代理；
3. 所有元素都是有序的（因为红黑树是有序的）

### 重要操作

Add操作



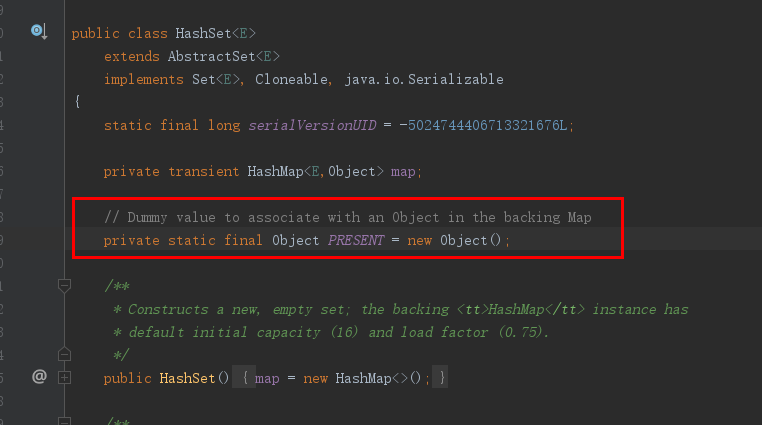
## HashSet

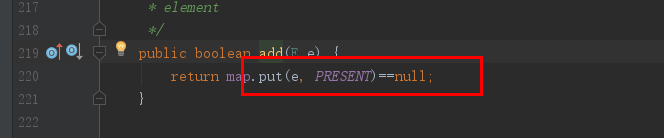
### 概述

1. HashSet是HashMap的代理（如果传递给HashSet构造方法中包含dummy参数，则创建的是LinkedHashMap对象），所有的数据都存放在HashMap对象。
2. 客户端在使用HashSet时，存放的元素会被作为HashMap的key存放，HashMap的value会存放一个固定Object对象，该对象时final；
3. 由第二点可得出HashSet不能存放相同元素，因为HashMap的key不能重复

### 重要操作

Add操作





Map.Put之后需要和null判断就是判断是否成功put。

## LinkedHashSet

### 概述

1. 实现了HashSet类；
2. 所有的构造方法都调用HashSet的Constructor，并带有dummy参数（参考2.2.1）

# 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）
4. 扩容：newCapacity = oldCapacity \* 1.5

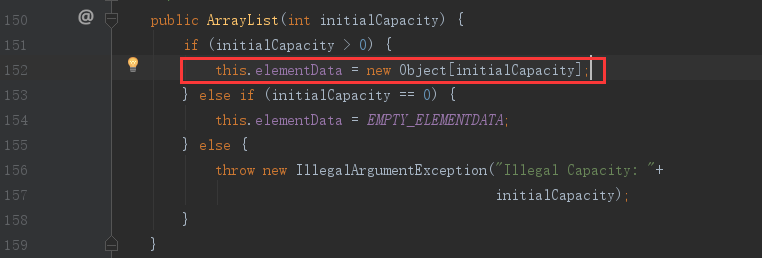
### 构造方法

1. 默认构造

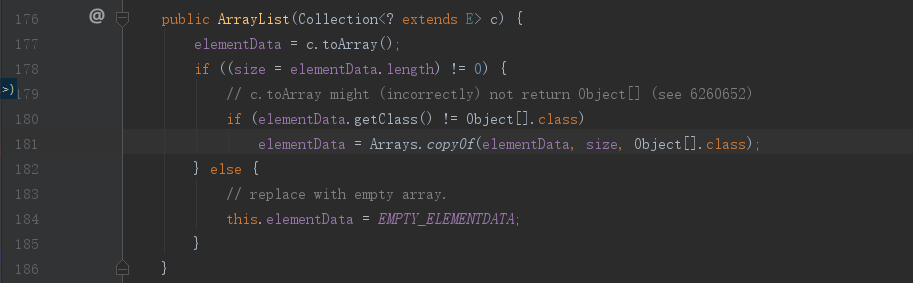


通过代码看出，如果没有设置size，则默认构造产生的是空数组。

1. 设置数组大小



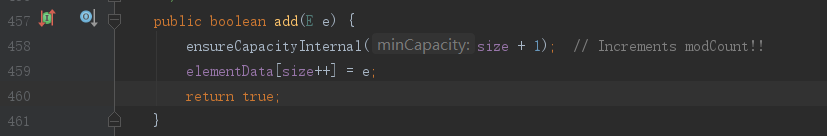
1. 拷贝构造



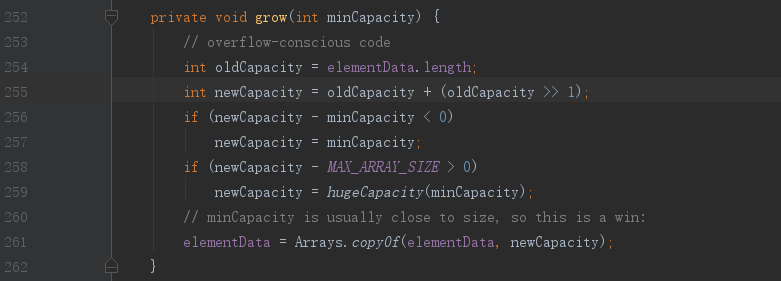
### 重要操作（增删改查）

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

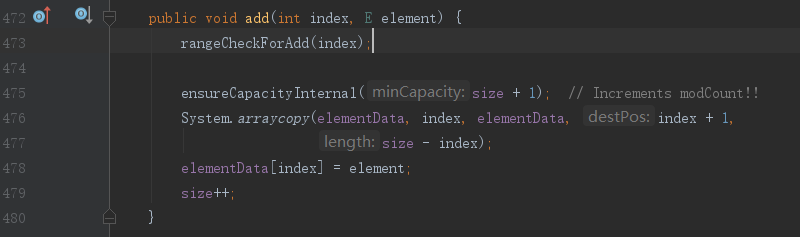


其中ensureCapacityInternal确保将元素存放在末尾不会越界，如果要越界，则调用grow方法将数组长度增大，增大为原来capacity的1.5倍，代码如下



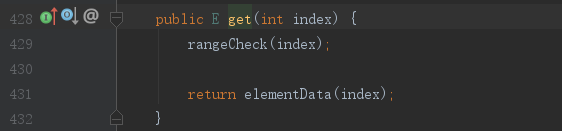
第二种add的思路和第一种类似，首先也要判断index是否越界；

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）。

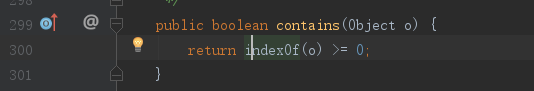


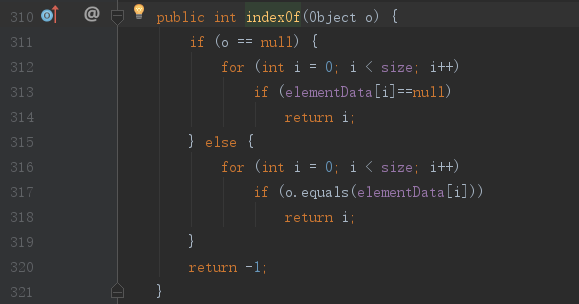
1. Get操作/Contains操作

Get操作是通过index获取；



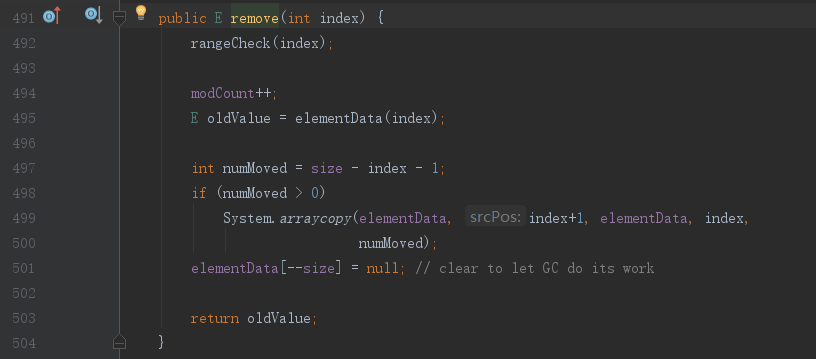
Contain操作通过元素判断是在存在：



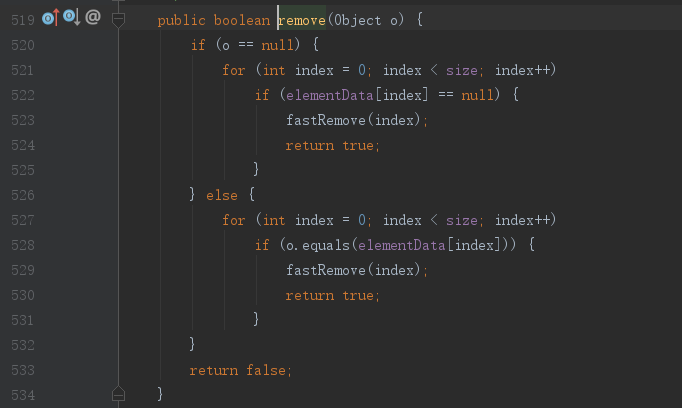


1. Remove操作

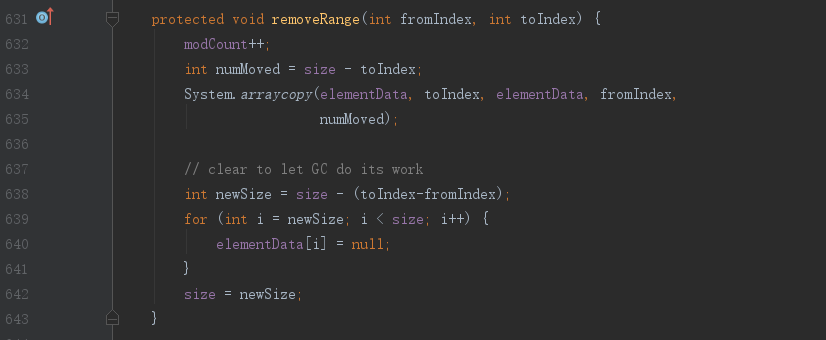
对应的，remove操作也有两种形式，一种是传递index，一种是传递元素



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。

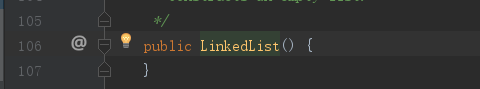
## LinkedList

### 概述

1. 双向链表，实现了Deque接口；
2. 带“Linked”的数据结构都是双向的；
3. 非线程安全的；

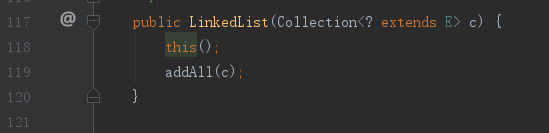
### 构造方法

1. 默认构造方法



无任何操作。

1. 拷贝构造方法



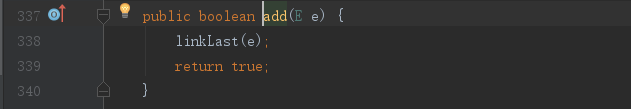
将集合c中的所有元素添加到链表中

### 重要操作

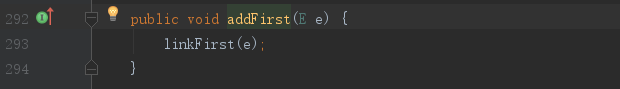
1. Add操作

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。

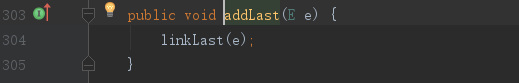
默认情况



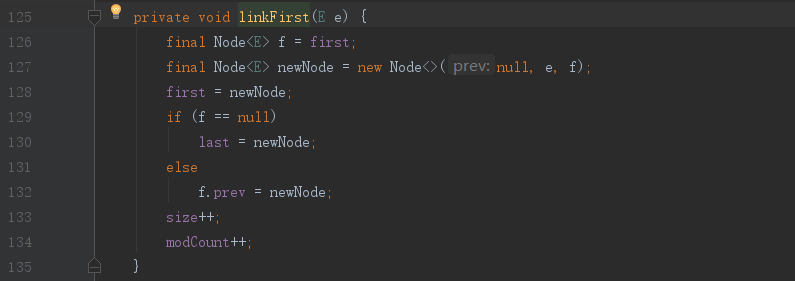
在头部插入

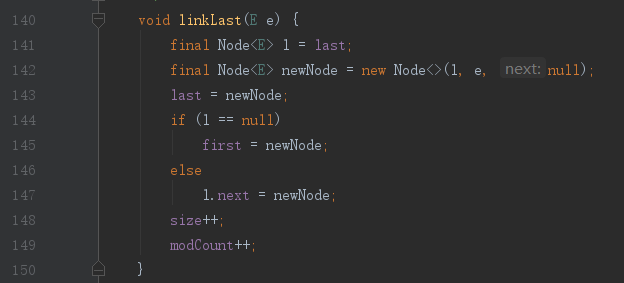


在尾部插入



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

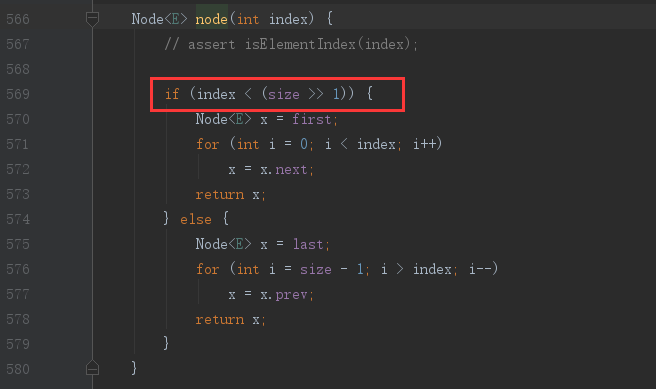




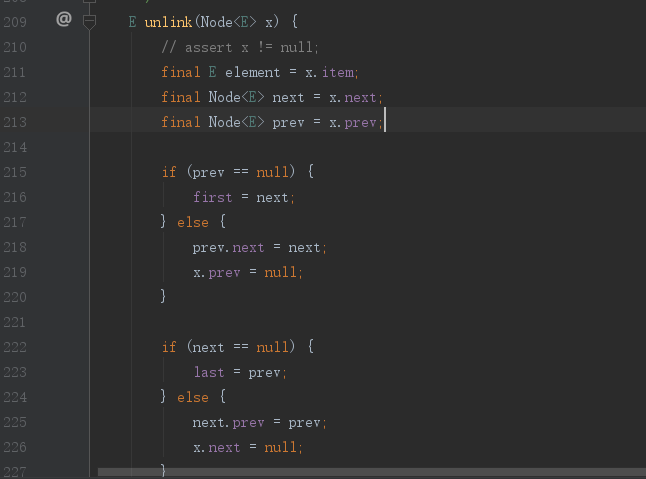
1. Remove操作

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。

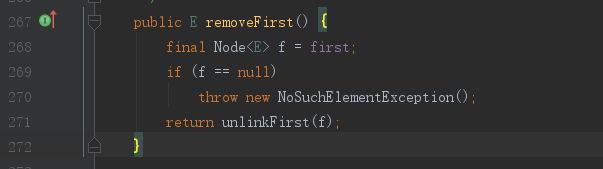
定位代码：

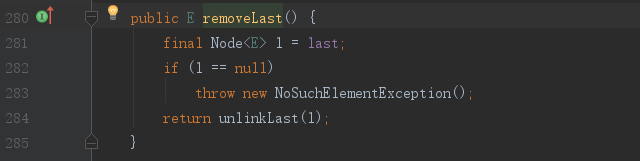


删除代码（常规思路）：

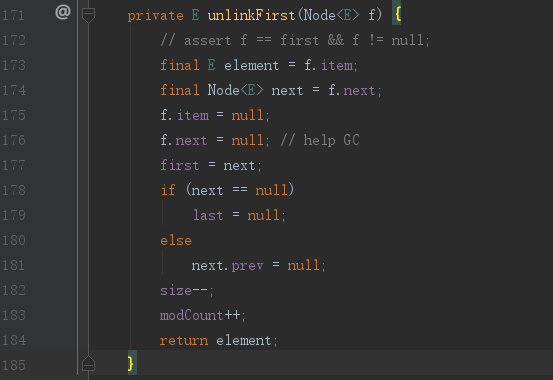


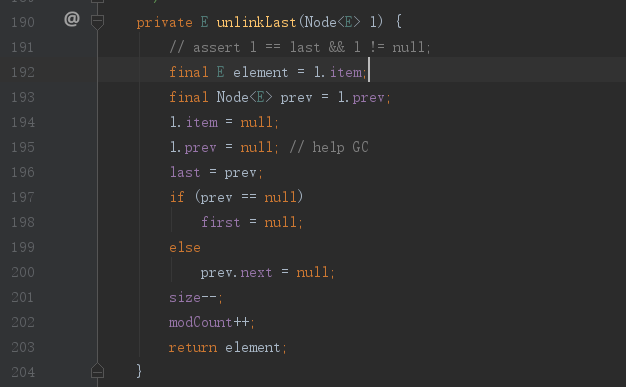
极端情况，删除头部或者尾部节点（也就是上面代码index=0，或则index=size-1）；





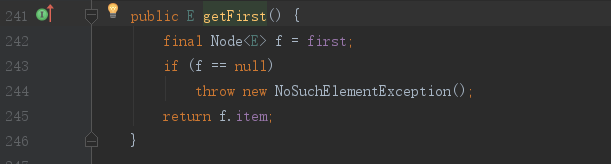
算法的难点是要考虑到删除节点后first，last为null的情况

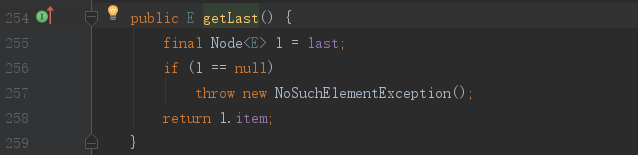


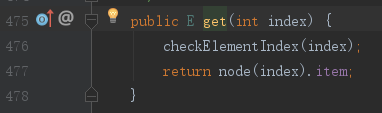


1. Get/Contains操作

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。







## Vector

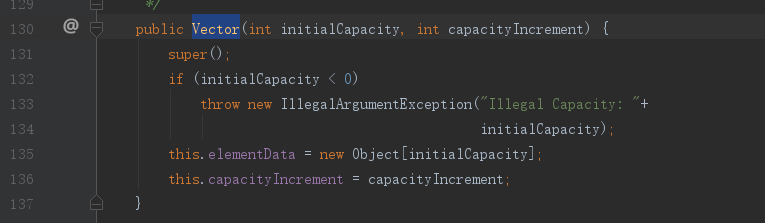
### 概述

1. 线程安全，通过synchronized关键字进行同步
2. capacityIncrement，指定Vector扩容方式，如果capacityIncrement小于等于0，则每次扩容时变为原来的两倍，否则capacity增加capacityIncrement；
3. public类型的方法都是synchronized的

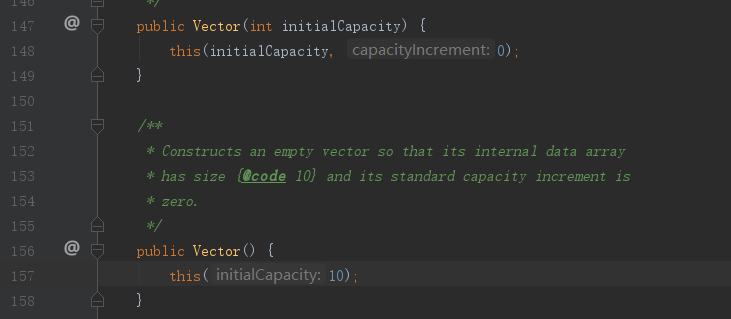
### 构造方法

1. 传参capacity、capacityIncrement

构造时则分配内存。

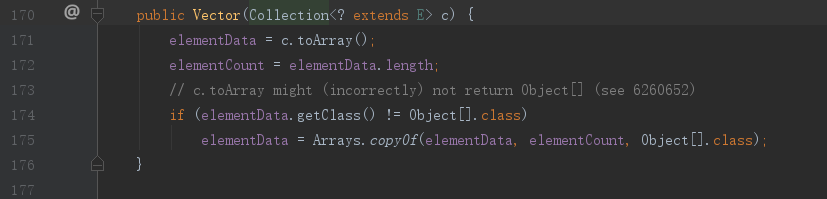


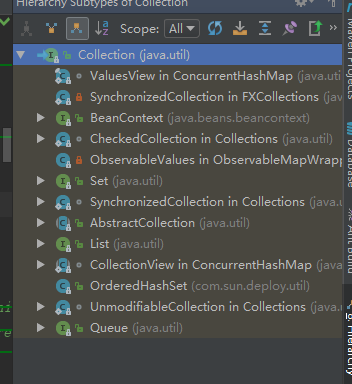
1. 默认构造大小



1. 拷贝构造

注意拷贝构造的参数类型，Collection，亦可以作为Collection的对象类型必须继承自Collection





### 重要操作

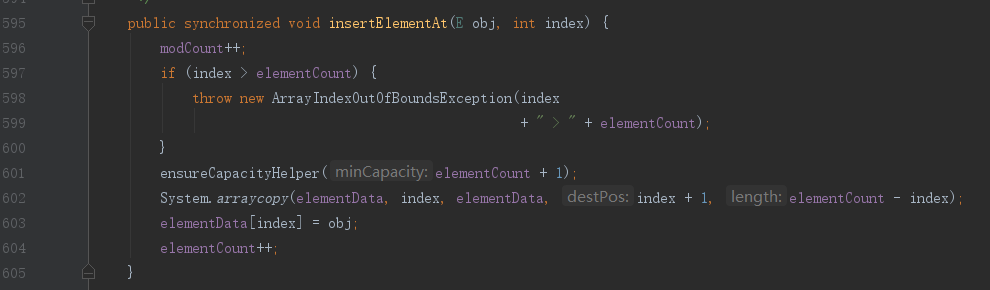
1. 删除操作

需要将删除位置到数组末尾的元素整体移位。



1. 插入操作

需要判断index是否大于size；如果满足插入条件，需要将元素做整体移位；



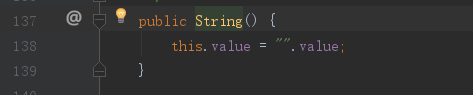
## String

### 概述

1. 重点了解其构造方法。
2. 字符数组是一个final数组，所有对String的值的修改都需要重新申请数组；
3. 所有有关string的操作都是返回新的对象，而不是对原有对象的引用；

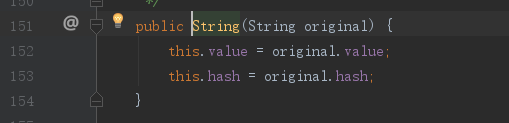
### 构造方法

1. 默认构造



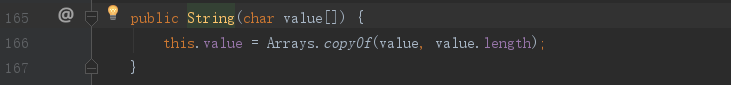
1. 拷贝构造

新的String和original的value指向同一数组地址，因此这个构造方法不推荐使用；

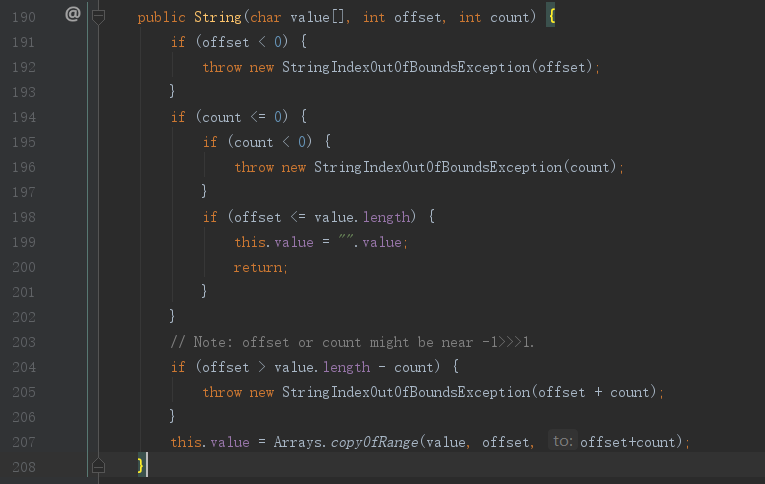


1. Char数组的构造

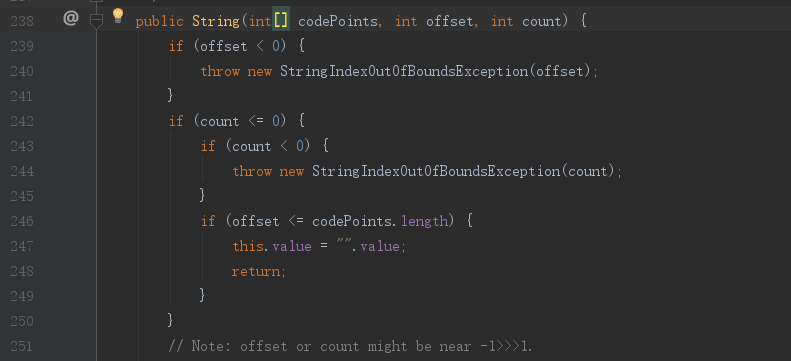
系统申请新的空间存放char数组中的值



1. Char数组构造，指定字符个数



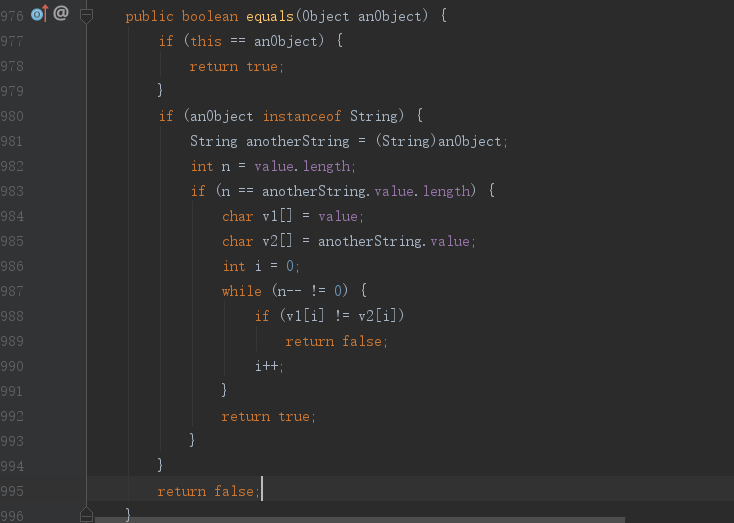
1. Int数组，存放每个字符的ascii码



### 重要操作

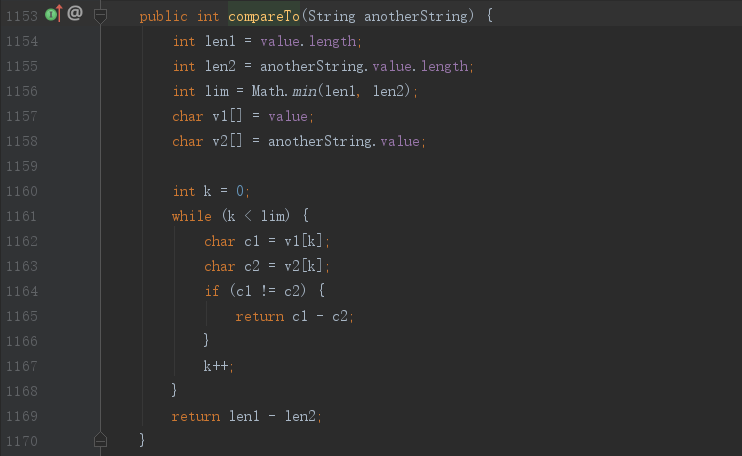
1. equal操作

逐个字符比较

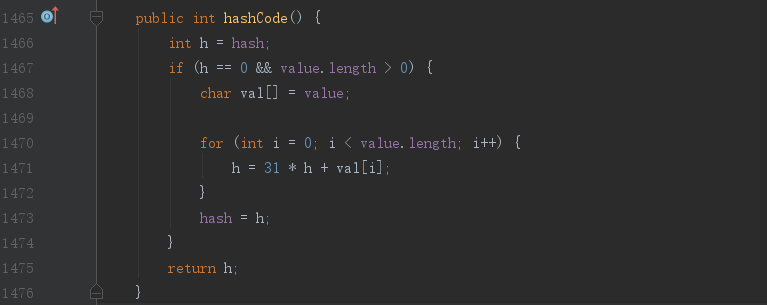


1. comparTo比较字典序

最终通过返回值与0做比较

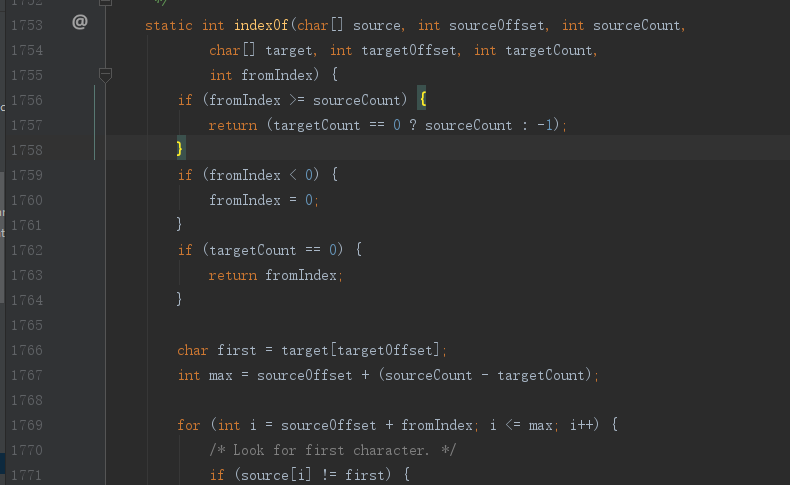


1. hashcode



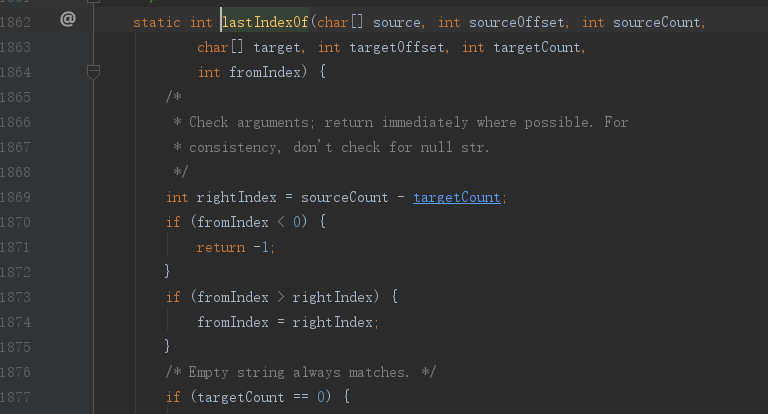
1. indexof计算target串在source串中的起始下标

此处并没有使用KMP算法，而是使用嵌套循环。



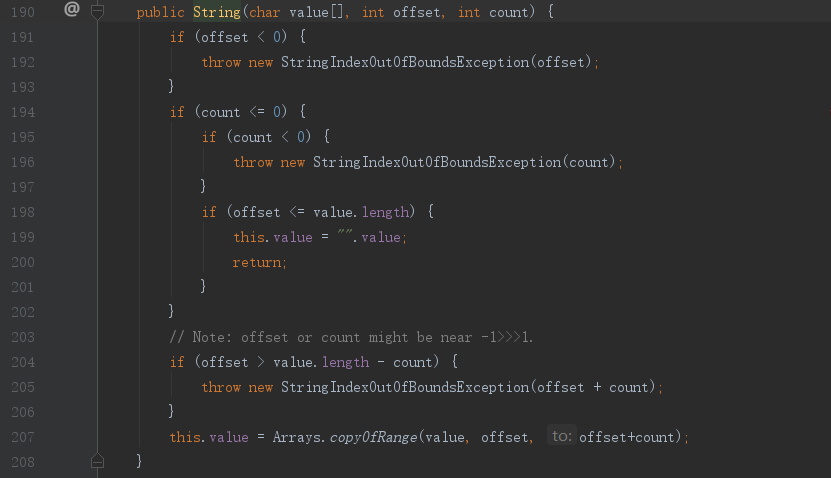
1. lastindexof计算最后一次出现的下标

同样的使用嵌套循环。

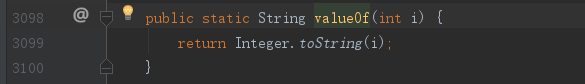


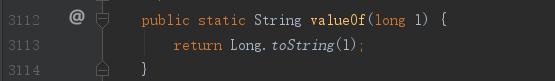
1. substring

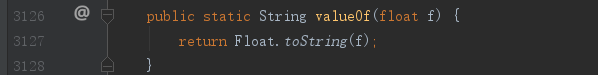
所有的substring最终都会转换为调用如下的构造方法

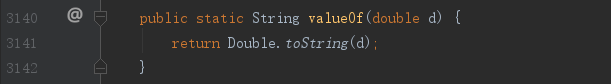


1. valueof，将int，long，float、double这几个基本类型的值转换为string时，通过对应的包装类的toString方法来获取



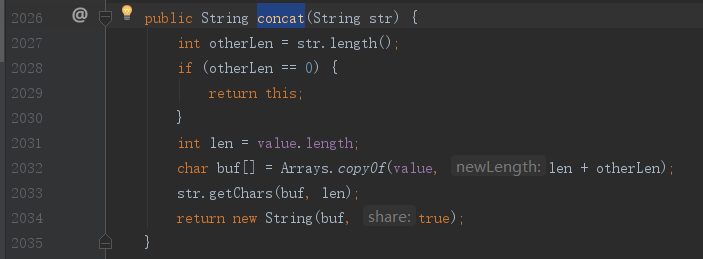




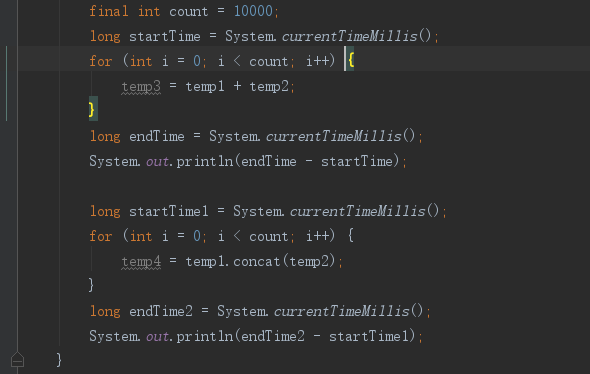


1. concat，将两个string拼接在一起

如果需要将两个string拼接在一起，用这个方法比用加法更快，详细可以具体查看两种操作编译之后的代码(class文件)；

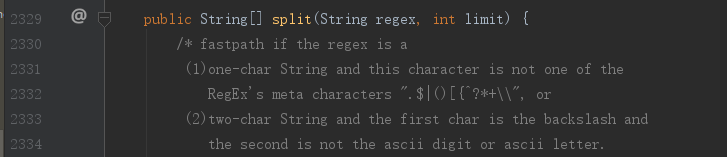


为了对比两种方法的执行时间可以作如下测试：

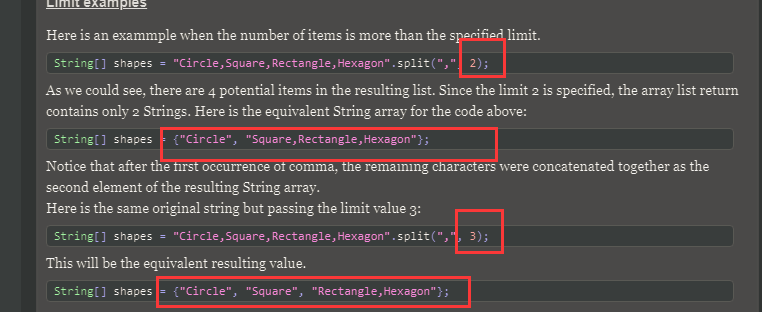


1. Split操作

重点需要理解的是limit参数，limit 参数控制模式应用的次数，因此影响结果数组的长度。如果该限制 n 大于 0，则模式将被最多应用 n - 1 次，数组的长度将不会大于 n，而且数组的最后项将包含超出最后匹配的定界符的所有输入。



举例



## StringBuilder

### 概述

1. StringBuilder继承自AbstractStringBuilder，几乎所有的方法都直接调用父类中已经实现过的方法；
2. StringBuilder和StringBuffer的区别是，StringBuilder中的方法没有synchronized关键字；

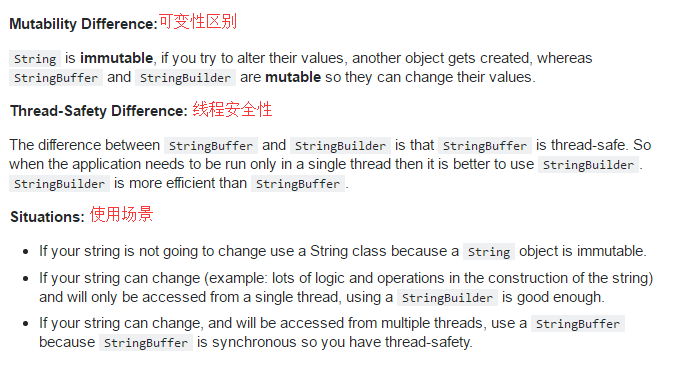
### 构造方法

### 重要操作

## StringBuffer

### 概述

1. StringBuffer继承自AbstractStringBuilder，几乎所有的方法都直接调用父类中已经实现过的方法，只不过在StringBuffer中的方法使用了关键字synchronized
2. StringBuffer类是final类型的，不能再被继承
3. String、StringBuffer、StringBuilder区别



### 构造方法

### 重要操作

# Map系列

## Map接口

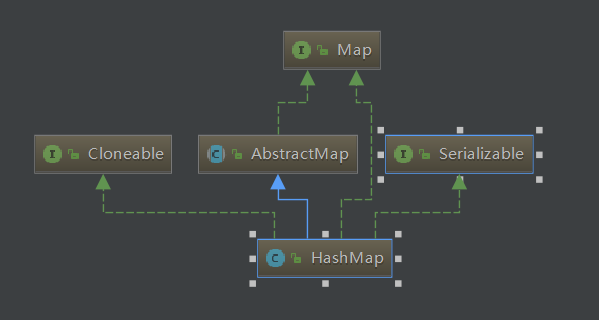
定义一系列的Method，便于其他map类实现。

### 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时通过取模计算hash&（capacity-1）的值；---这种方式在扩容时，同一个桶中的元素只会分为两类
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. 多线程场景下的死循环，参考

https://cloud.tencent.com/developer/article/1120823

1. 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
2. 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

在Java中，即使构造方法为空，如果field有初始化都需要执行默认操作。

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

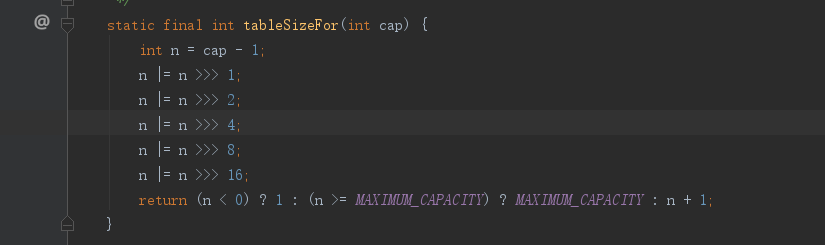


第510行，如果拷贝的map的size大于默认大小，则扩容，扩容的思想就是将内存扩大为原来的两倍，同时把原来的数据从旧地址，“复制”到新地址，“复制”的时候需要重新计算hashcode并计算新的index（详见10）

### 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；

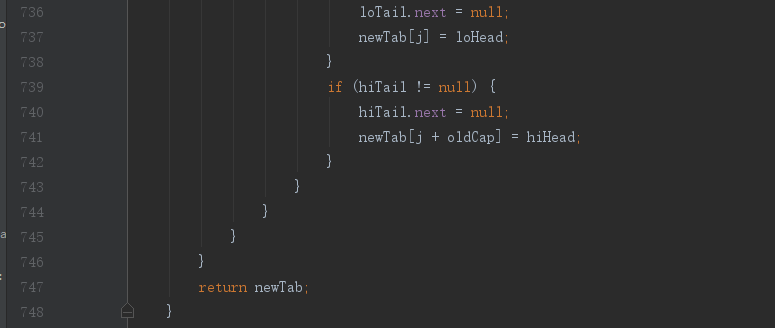
……

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）。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。

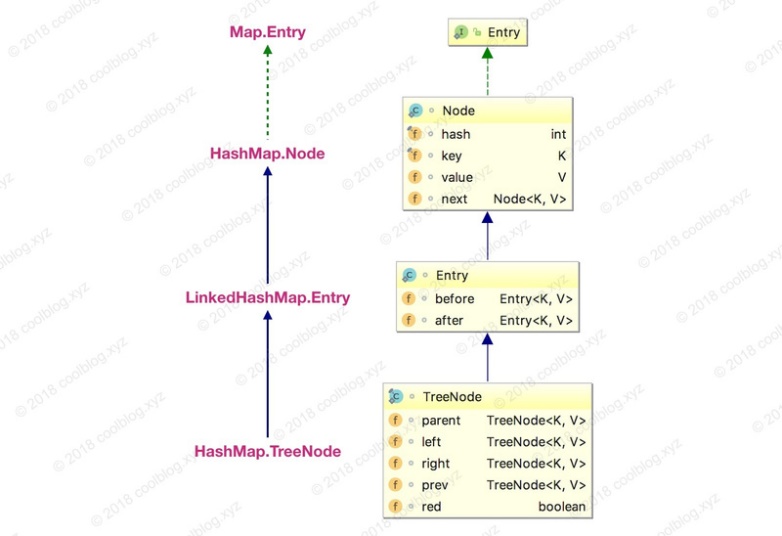
## LinkedHashMap

### 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. 同理，由于继承自HashMap，因此Key可以为null；
4. 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；
5. 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）
6. 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）
7. 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





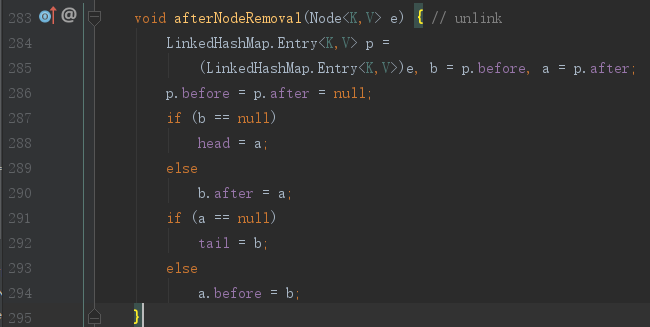
第222~231行创建双向链表，把最新put进来的元素存放在链表末尾。算法分两种情况，如果插入的是第一个元素，此时tail为null，因此需要设置head（226行），否则，将p节点存放在tail后下一个地方，tail指向p。

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。



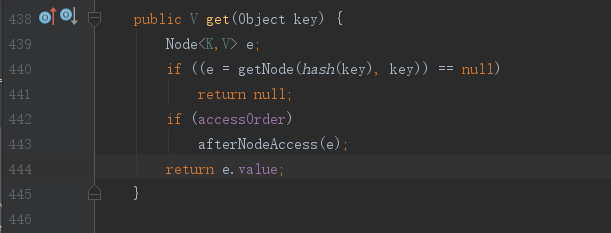
1. Remove操作

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。



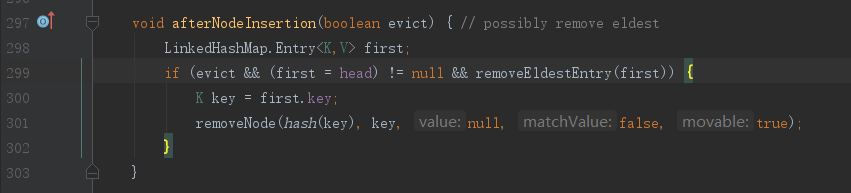
1. Get操作（查）

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. Key是通过Compartor比较器进行比较
3. Key不能为Null，不然无法使用比较器Compartor
4. 非线程安全
5. 理解TreeMap的插入和删除操作，重点是理解红黑树算法，其他操作都是为了提升性能以空间换时间做的一些优化
6. 红黑树的插入和删除操作对应fixAfterInsertion、deleteEntry

### 构造方法

### 重要操作

## WeakHashMap

### 概述

### 构造方法

### 重要操作

## ConcurrentHashMap

参考：https://juejin.im/entry/59fc786d518825297f3fa968

### 概述

* Jdk1.8之后去除了Segment，为了解决向下兼容定义了一个空的Segment
* 增加CAS算法解决多线程同步问题；
* Key和value都不允许为null；

### 构造方法

### 重要操作

所有的容器最重要的两个操作就是“进”容器和“出”容器，围绕这两个操作会增加一些辅助操作，因此理解这些重要操作可以从“进”容器和“出”容器开始逐个解读。

#### Hash值

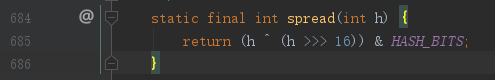


Figure concurrentHashMap的hash计算算法

将key的hashcode的高16位和低16位进行异或操作并存放在低16位，高16位不变，然后再去h的低31位作为index。低31位是因为concurrenthashmap的最大capacity是2的30次方；

#### 初始化table

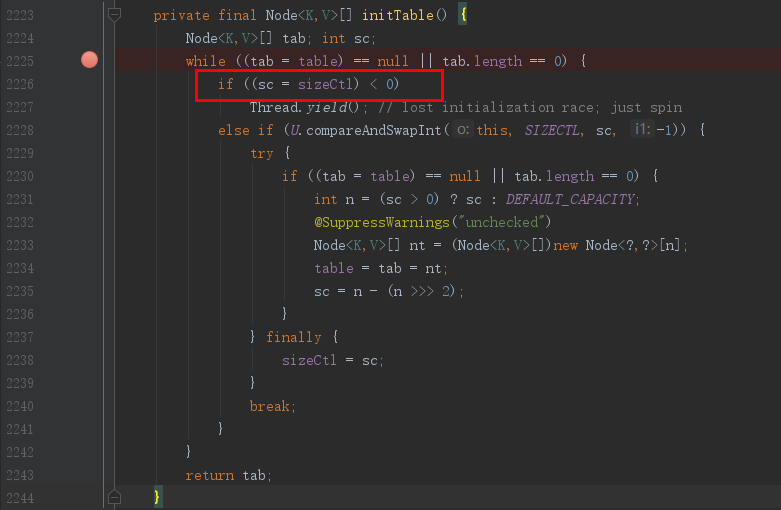


Figure concurrenthashmap初始化过程

ConcurrentHashMap的初始化过程是通过sizeCtl变量进行控制，只有获取到sizeCtl变量的线程才能执行else代码块中的代码。有关sizeCtl变量的解释如下：

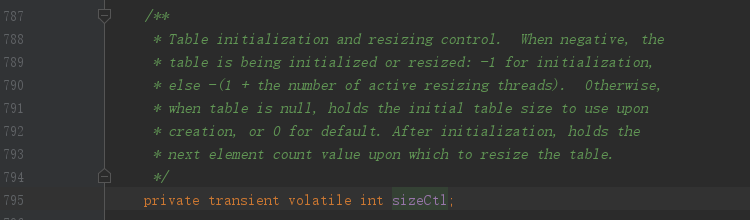


Figure sizeCtl变量注释

* SizeCtl为负数时，表示该ConcurrentHashMap正在被初始化或者扩容；
* 第一次初始化时sizeCtl被设置为-1；
* -（n+1）表示有n个线程正在扩容；
* SizeCtl变量是volatile类型

Figure 7中第2228行保证只能有一个线程执行成功。

#### 扩容

#### Get操作

#### Put操作

# 包装类（封装类）

本章节学习Integer、Double、Float等包装类，这些类都继承自Number类，实现了Comparable接口。首先要了解Number类。主要理解：Java 编译器把原始类型自动转换为封装类的过程称为自动装箱（autoboxing），相当于调用包装类的valueof方法。举例说明：

源码：



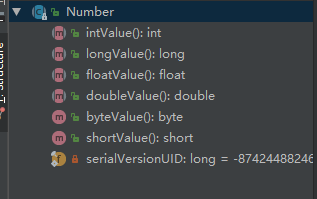
编译之后的代码：



## Number

### 概述

1. 抽象类；
2. 定义了多种抽象方法；



## Integer

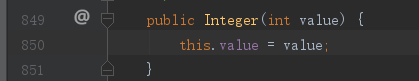
Integer、Double、Long、Float、Boolean等包装类的逻辑基本一致，理解其中一个就可以。

### 概述

1. 内存占用四个字节
2. 最大值0x7fffffff
3. 最小值0x80000000
4. 内部子类IntegerCache是Integer的缓存类，有利于节省内存和提高性能，详细见重要操作
5. 具有一个int类型属性
6. Type属性等价于int.class

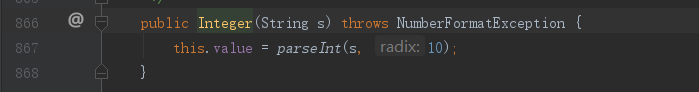
### 构造方法

1. int类型参数的构造



1. String类型参数的构造

ParseInt的radix参数表示进制数



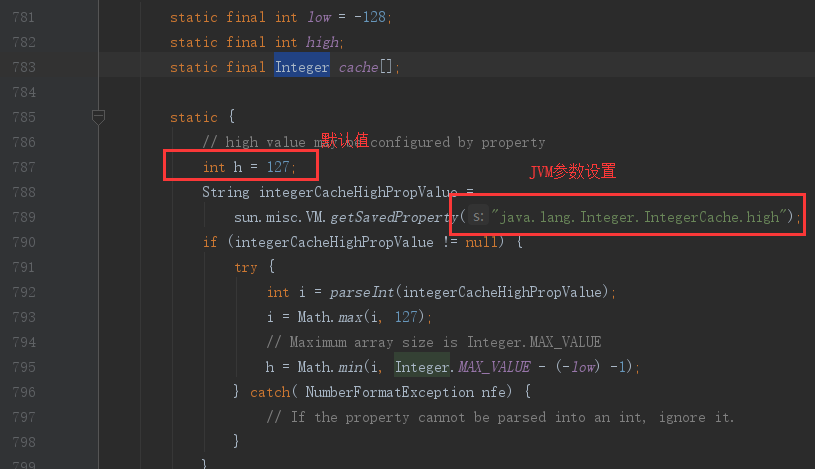
1. 额

### 重要操作

1. 子类IntegerCache

默认情况下，Integer类在加载的时候，会执行IntegerCache中的静态方法，生成[-128,127]之间的所有Integer类对象，并存放在一个静态数组内。

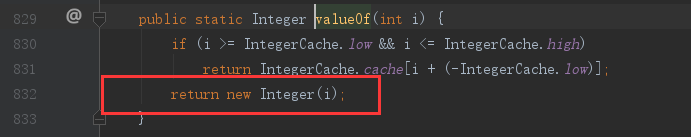
IntegerCache类不能显示被实例化，因为其构造方法是private。



因此，如果在创建[-128,127]之间的Integer时，都是引用同一个Integer对象。IntegerCache中的low和high指定了这个数组区间。有了这个缓存，则可以解释如下执行：

大于127，则会调用valueof中的new操作。

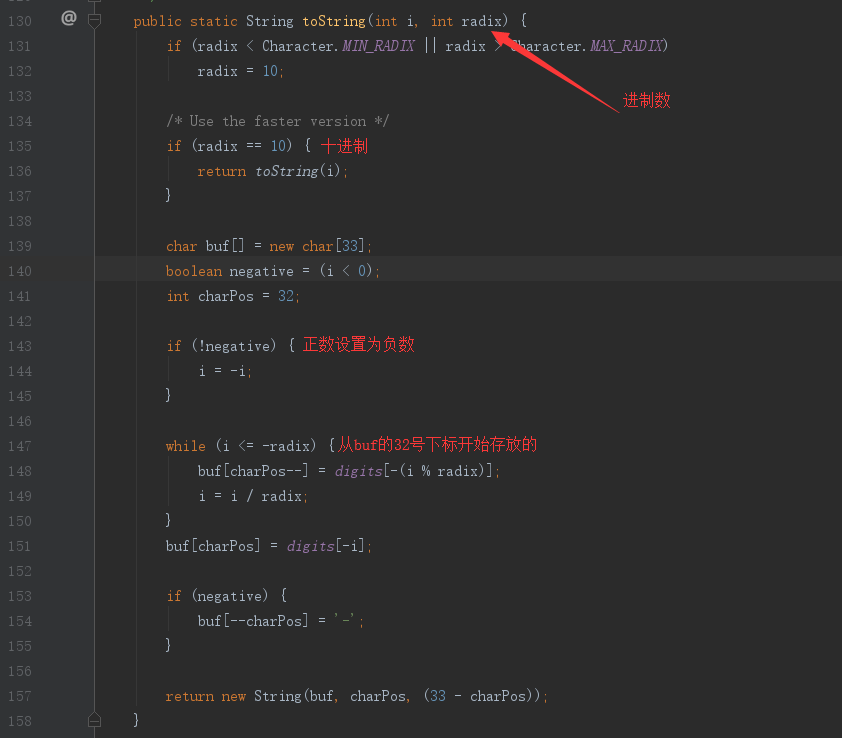




1. toString操作

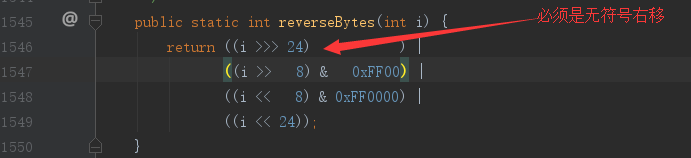
参数radix表示需要转换为string时的进制数，但是radix必须大于等于Character.MIN\_RADIX（值为2），小于等于Character.MAX\_RADIX（值为36）。

此处需要掌握将十进制数转换为任意进制的算法。在此算法中也不可以不用转换为负数来进行转换。



1. reverseByte操作

将一个int的四个字节进行逆转，逆转通过移位的方式进行实现。



# Concurrent包

这一部分的内容需要结合《深入浅出Java Concurrency》和jdk源码一起学习。

参考资料：

<http://www.blogjava.net/xylz/archive/2010/07/08/325587.html>

# CopyOnWrite容器

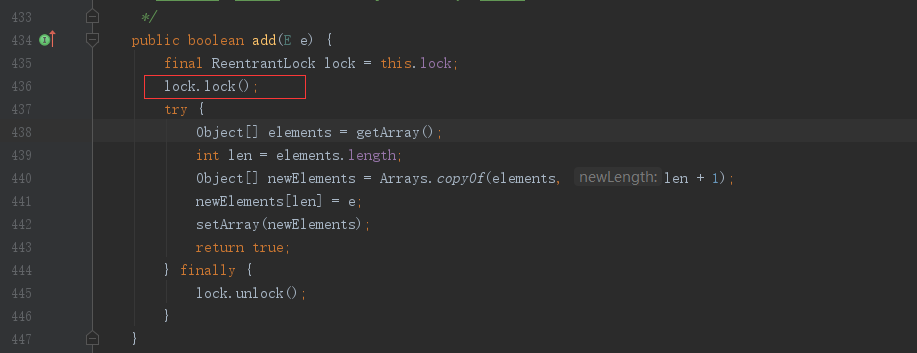
CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器（复制操作时浅拷贝），然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

## CopyOnWriteArrayList

所有对CopyOnWriteArrayList的“写”操作（包括add、remove等操作）都会事先获取ReentrantLock锁，然后再进行操作；所有的“读”操作运行多线程并发执行；

### 重要操作

Add操作



Remove操作

