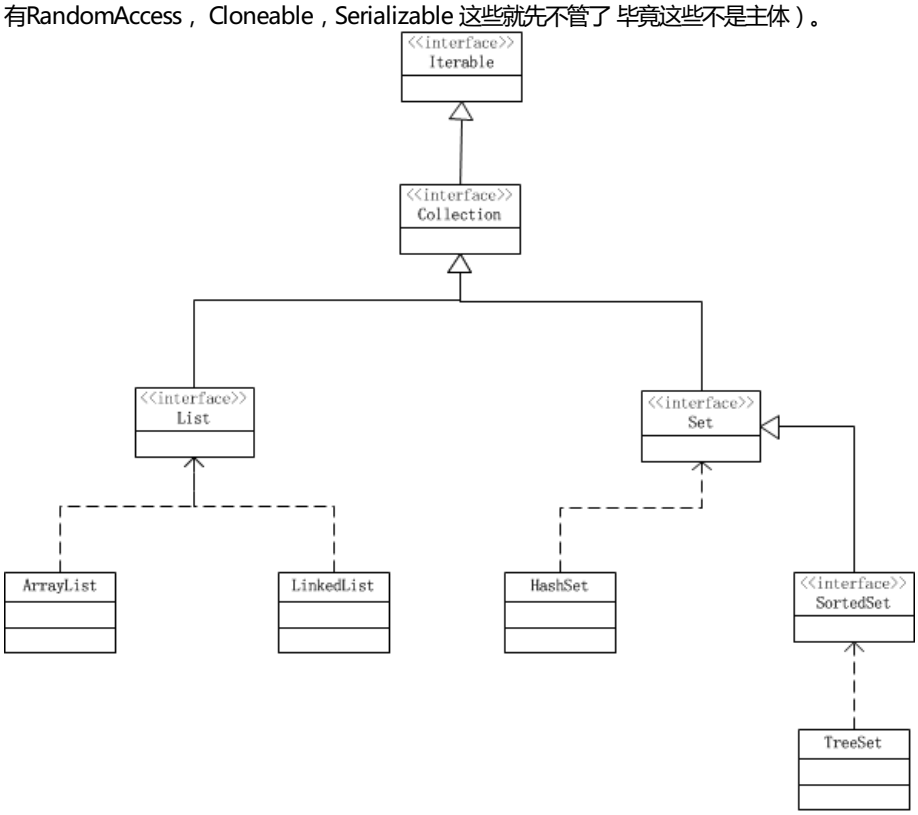
JAVA Collection 常用集合 源码解析

我觉得学习任何一个东西，首先你要有一个整体的架构印象，知道它大概是个什么东西，你要学的东西又在这个架构中的哪一部分，这样先总后分，学习起来就觉得很有意思了。
下面先附上Collection集合的总体架构图（PS：这并不是完整的类图，至于那些为骨骼架构而设计的AbstractCollection之类的抽象类还有RandomAccess， Cloneable，Serializable 这些就先不管了 毕竟这些不是主体）。



（Collection集合核心类图）

从这个核心类图上我们可以看到 最上层的是Iterable接口类，它就只是定义了一个迭代的接口，里面只有一个迭代的函数，这也是为什么所有的集合类都有迭代函数的原因。
Collection类中定义了一些基本的集体函数，比如增加 add()、删除 remove()、求大小 size() 等函数。在Collection接口后有了分支：List和Set。分支的依据就是根据集合中的元素是否可以重复。List是允许元素重复的集合，Set是不允许元素重复的集合。然后在List后又有了分支，这里的分支依据则是根据它们的存储结构来进行划分。ArrayList是顺序存储结构 LinkedList是链式存储结构。Set则按是否有序分为HashSet和SortedSet。其中SortedSet是一个接口，它的实现类为TreeSet。 下面我就根据这些实现类的源码 从构造方法、增加、删除这三个方面来介绍。

1、ArrayList 它的特点是支持快速查找与更新，连续存储。

构造函数：

```
public ArrayList(int initialCapacity) {
 super();
      if (initialCapacity < 0)
         throw new IllegalArgumentException("Illegal Capacity: "+
                                            initialCapacity);
 this.elementData = new Object[initialCapacity];
   }

 public ArrayList() {
 this(10);
   }

 public ArrayList(Collection<? extends E> c) {
 elementData = c.toArray();
 size = elementData.length;
 // c.toArray might (incorrectly) not return Object[] (see 6260652)
 if (elementData.getClass() != Object[].class)
    elementData = Arrays.copyOf(elementData, size, Object[].class);
   }
```

ArrayList一共提供了3个构造函数 ArrayList(int initialCapacity) 根据用户输入的数组大小进行初始化 ArrayList() 用户没有输入数组大小则调用 ArrayList(int initialCapacity)构造方法 并默认大小为10。所以说，JAVA中你不给定数组大小，系统也还是会根据默认大小去分配一个初始空间。ArrayList(Collection<? extends E> c) 根据已有的集体进行初始化。看了这3个构造函数时，我就建议大家以后在进行数组的初始化时还是根据实际要用的大小去调用 ArrayList(int initialCapacity) 初始化数组 这样可以减少扩容的次数，提升效率。

增加函数：

```
public boolean add(E e) {
 ensureCapacity(size + 1);  // Increments modCount!!
 elementData[size++] = e;
 return true;
   }

public void add(int index, E element) {
 if (index > size || index < 0)
    throw new IndexOutOfBoundsException(
 "Index: "+index+", Size: "+size);
 ensureCapacity(size+1);  // Increments modCount!!
 System.arraycopy(elementData, index, elementData, index + 1,
   size - index);
 elementData[index] = element;
 size++;
   }

 public void ensureCapacity(int minCapacity) {
 modCount++;
 int oldCapacity = elementData.length;
 if (minCapacity > oldCapacity) {
    Object oldData[] = elementData;
    int newCapacity = (oldCapacity * 3)/2 + 1;
       if (newCapacity < minCapacity)
 newCapacity = minCapacity;
         // minCapacity is usually close to size, so this is a win:
         elementData = Arrays.copyOf(elementData, newCapacity);
 }
   }
```

ArrayList中有两种增加元素的方法 add(E e) 方法是直接在最后加元素，这种最快、效率也是最高。 add(int index, E element)方法 是在数组中的某一个位置插入元素，这种方法需要移动元素，效率当然相对也就低一些。所以建议如果没有特别要求还是直接用 add(E e) 方法直接进行元素的添加。当然，不管你用哪哪种方法都会进行一个容量的判断，判断当前数组是否有足够的空间来存储新元素。在 ensureCapacity(int minCapacity) 这个函数中，我们可以看到，ArrayList数组在扩容时是把存储空间扩为原来的1.5倍，然后再把原来的元素复制过来。所以，一个好的初始容量是可以减少扩容的次数的。

删除函数：

```
 public E remove(int index) {
 RangeCheck(index);

 modCount++;
 E oldValue = (E) elementData[index];

 int numMoved = size - index - 1;
 if (numMoved > 0)
    System.arraycopy(elementData, index+1, elementData, index,
       numMoved);
 elementData[--size] = null; // Let gc do its work

 return oldValue;
   }

public boolean remove(Object o) {
 if (o == null) {
         for (int index = 0; index < size; index++)
  if (elementData[index] == null) {
     fastRemove(index);
     return true;
  }
 } else {
    for (int index = 0; index < size; index++)
  if (o.equals(elementData[index])) {
     fastRemove(index);
     return true;
  }
      }
 return false;
   }
  private void fastRemove(int index) {
```

```
        modCount++;
        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--size] = null; // Let gc do its work
    }
```

ArrayList中有两种删除元素的方法 remove(int index) 根据索引进行元素的删除，remove(Object o) 根据数组元素进行删除，但细看就知道第二种 remove(Object o) 方法是先通过循环遍历得到元素对应的索引，然后根据索引再进行删除。所以说删除方法 首选 remove(int index) 。同时，在 remove(Object o) 在进行删除时是进行了一个分类的处理，即对传进来的对象要进行判断，看是否为 NULL 以避免空指针错误，这是我们在写程序时要特别注意的地方，有对象就要考虑是否会为NULL 进行分类处理。API中很多地方都是这样进行处理的。 另外我们也看到了在ArrayList中进行元素的删除时是要进行元素移动的，且越靠前的元素，删除时要移动的元素就越多，所以如果是删除或插入频繁的数组，就不太适合用ArrayList来进行存储 。

2、LinkedList 链式存储结构，插入和删除较快。

构造函数：

```
 public LinkedList() {
        header.next = header.previous = header;
    }
 public LinkedList(Collection<? extends E> c) {
 this();
 addAll(c);
    }
```

LinkedList中提供两种初始化链表的方法 LinkedList() 初始化一个空链表 LinkedList(Collection<? extends E> c) 根据一个已知集合去初始化链表。这里没什么好说的。

增加方法：

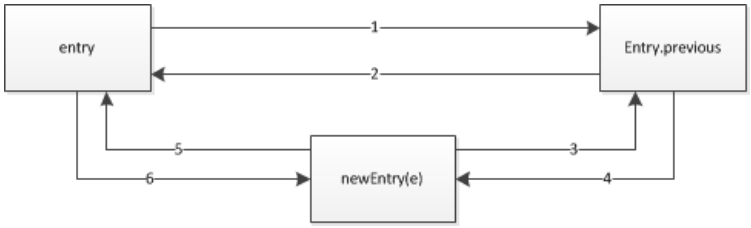
```
public boolean add(E e) {
 addBefore(e, header);
        return true;
    }

 public void add(int index, E element) {
        addBefore(element, (index==size ? header : entry(index)));
    }

 public void addFirst(E e) {
 addBefore(e, header.next);
    }

 public void addLast(E e) {
 addBefore(e, header);
    }
   private Entry<E> addBefore(E e, Entry<E> entry) {
     Entry<E> newEntry = new Entry<E>(e, entry, entry.previous);
     newEntry.previous.next = newEntry;
     newEntry.next.previous = newEntry;
     size++;
     modCount++;
     return newEntry;
    }
```

LinkedList中一共包含 add(E e) 增加元素e　add(int index, E element)在索引index处增加元素e　addFirst(E e) 增加表头元素 addLast(E e) 增加表尾元素 通过比较函数实现 可以发现 所有的增加函数 其内部都是通过调用 addBefore(E e, Entry<E> entry) 函数来进行实现的，本质上是一质的，只是不同包装而已。下面我们就来分析一下 addBefore(E e, Entry<E> entry) 这个函数。

分析之前我们必须知道 Entry 这个东西是什么东东

```
 private static class Entry<E> {
     E element;
     Entry<E> next;
     Entry<E> previous;
     Entry(E element, Entry<E> next, Entry<E> previous) {
         this.element = element;
         this.next = next;
         this.previous = previous;
     }
    }
```

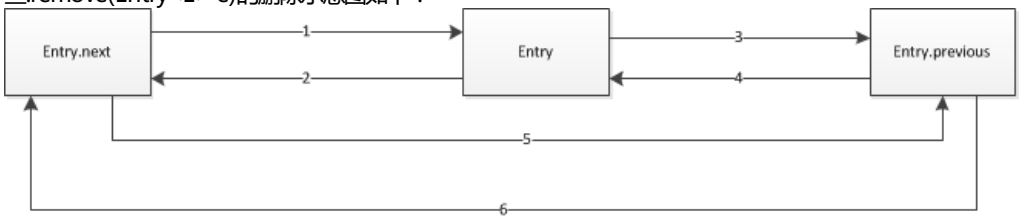
Entry就是一个内部类，它的作用就相当于以前我们数据结构中学过的结点，用来纪录当前元素的前一个结点，后一个结点以及当前的结点值。正是利用了这种存储结构从而使它可以进行快速的插入与删除。



知道它的结构以后呢 我们就根据上面这个图来解析 addBefore(E e, Entry<E> entry) 这个函数的操作。
首先第一步： Entry<E> newEntry = new Entry<E>(e, entry, entry.previous); 根据Entry的构造 函数 这一步会产生 3、5 这两个箭头
第二步：newEntry.previous.next = newEntry; 会把箭头2 变成箭头4
第三步： newEntry.next.previous = newEntry; 会把箭头1 变成箭头6
这样就成功的将新元素new Entry插入到了entry元素的前面。
这里插入比较特殊的就是 addBefore(e, header); 即在头结点之前插入，其实也就是插入链表的尾部。从LinkedList的插入方法中也可以看出它插入元素时的时间复杂度为O(1) 速度很快。
删除函数：

```
public E remove(int index) {
      return remove(entry(index));
   }

public boolean remove(Object o) {
      if (o==null) {
         for (Entry<E> e = header.next; e != header; e = e.next) {
            if (e.element==null) {
               remove(e);
               return true;
            }
         }
      } else {
         for (Entry<E> e = header.next; e != header; e = e.next) {
            if (o.equals(e.element)) {
               remove(e);
               return true;
            }
         }
      }
      return false;
   }

 private E remove(Entry<E> e) {
 if (e == header)
    throw new NoSuchElementException();
      E result = e.element;
 e.previous.next = e.next;
 e.next.previous = e.previous;
      e.next = e.previous = null;
      e.element = null;
 size--;
 modCount++;
      return result;
   }
```

remove(int index)根据索引来进行删除 remove(Object o)根据元素来进行删除 这两种方法的本质都是调用remove(Entry <E> e)来进行删除，为什么呢 因为在LinkedList中元素的存储结构就是Entry，所有的元素都是存在一个Entry对象数组中，两种方法的区别在于remove(Object o)是要先遍历链表得到元素对应的Entry对象，然后再删除。所以，如果知道索引还是直接用索引删除快一些.remove(Entry<E> e)的删除示意图如下：



原本是只有1、2、3、4这 4个箭头的。删除一共分三步走
第一步：e.previous.next = e.next; 使得箭头6产生
第二步： e.next.previous = e.previous; 使得箭头5产生
第三步： e.next = e.previous = null; e.element = null;使得箭头1、2、3、4 消失 元素清空 等垃圾回收器回收。
从这里我们可以看到 在Entry对象数组中删除元素的时间复杂度为 O(1) 所以相比如ArrayList数组来说 删除的效率还是会高出一些。
所以对二插入和删除频繁的数组来说 建议用LinkedList数组。
3、HashSet HashSet是一种无序、不重复的集合。

构造函数：

```
 public HashSet() {
 map = new HashMap<E,Object>();
   }
public HashSet(Collection<? extends E> c) {
 map = new HashMap<E,Object>(Math.max((int) (c.size()/.75f) + 1, 16));
 addAll(c);
   }
```

HashSet的构造函数一共是提供了两种形式。HashSet() 无参数构造 直接无参数构造一个HashMap对象。 HashSet(Collection<? extends E> c)则是根据集合元素初始化HashSet。

增加函数：

```
   public boolean add(E e) {
 return map.put(e, PRESENT)==null;
   }
```

删除函数：

```
   public boolean remove(Object o) {
 return map.remove(o)==PRESENT;
   }
```

从上可以，HashSet的内部操作实际上都是HashMap的操作，所以具体的 留到下回分析Map系集合时再细讲。