         我们经常使用subString方法来对String对象进行分割处理，同时我们也可以使用subList、subMap、subSet来对List、Map、Set进行分割处理，但是这个分割存在某些瑕疵。

**一、subList返回仅仅只是一个视图**

        首先我们先看如下实例：

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. **public** **static** **void** main(String[] args) {
2. List<Integer> list1 = **new** ArrayList<Integer>();
3. list1.add(1);
4. list1.add(2);
6. //通过构造函数新建一个包含list1的列表 list2
7. List<Integer> list2 = **new** ArrayList<Integer>(list1);
9. //通过subList生成一个与list1一样的列表 list3
10. List<Integer> list3 = list1.subList(0, list1.size());
12. //修改list3
13. list3.add(3);
15. System.out.println("list1 == list2：" + list1.equals(list2));
16. System.out.println("list1 == list3：" + list1.equals(list3));
17. }

        这个例子非常简单，无非就是通过构造函数、subList重新生成一个与list1一样的list，然后修改list3，最后比较list1 == list2?、list1 == list3?。按照我们常规的思路应该是这样的：因为list3通过add新增了一个元素，那么它肯定与list1不等，而list2是通过list1构造出来的，所以应该相等，所以结果应该是：

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. list1 == list2：**true**
2. list1 == list3: **false**

        首先我们先不论结果的正确与否，我们先看subList的源码：

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. **public** List<E> subList(**int** fromIndex, **int** toIndex) {
2. subListRangeCheck(fromIndex, toIndex, size);
3. **return** **new** SubList(**this**, 0, fromIndex, toIndex);
4. }

        subListRangeCheck方式是判断fromIndex、toIndex是否合法，如果合法就直接返回一个subList对象，**注意在产生该new该对象的时候传递了一个参数 this ，该参数非常重要，因为他代表着原始list。**

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. /\*\*
2. \* 继承AbstractList类，实现RandomAccess接口
3. \*/
4. **private** **class** SubList **extends** AbstractList<E> **implements** RandomAccess {
5. **private** **final** AbstractList<E> parent;    //列表
6. **private** **final** **int** parentOffset;
7. **private** **final** **int** offset;
8. **int** size;
10. //构造函数
11. SubList(AbstractList<E> parent,
12. **int** offset, **int** fromIndex, **int** toIndex) {
13. **this**.parent = parent;
14. **this**.parentOffset = fromIndex;
15. **this**.offset = offset + fromIndex;
16. **this**.size = toIndex - fromIndex;
17. **this**.modCount = ArrayList.**this**.modCount;
18. }
20. //set方法
21. **public** E set(**int** index, E e) {
22. rangeCheck(index);
23. checkForComodification();
24. E oldValue = ArrayList.**this**.elementData(offset + index);
25. ArrayList.**this**.elementData[offset + index] = e;
26. **return** oldValue;
27. }
29. //get方法
30. **public** E get(**int** index) {
31. rangeCheck(index);
32. checkForComodification();
33. **return** ArrayList.**this**.elementData(offset + index);
34. }
36. //add方法
37. **public** **void** add(**int** index, E e) {
38. rangeCheckForAdd(index);
39. checkForComodification();
40. parent.add(parentOffset + index, e);
41. **this**.modCount = parent.modCount;
42. **this**.size++;
43. }
45. //remove方法
46. **public** E remove(**int** index) {
47. rangeCheck(index);
48. checkForComodification();
49. E result = parent.remove(parentOffset + index);
50. **this**.modCount = parent.modCount;
51. **this**.size--;
52. **return** result;
53. }
54. }

        该SubLsit是ArrayList的内部类，它与ArrayList一样，都是继承AbstractList和实现RandomAccess接口。同时也提供了get、set、add、remove等list常用的方法。但是它的构造函数有点特殊，在该构造函数中有两个地方需要注意：

        1、this.parent = parent;而parent就是在前面传递过来的list，也就是说this.parent就是原始list的引用。

        2、this.offset = offset + fromIndex;this.parentOffset = fromIndex;。同时在构造函数中它甚至将modCount（fail-fast机制）传递过来了。

        我们再看get方法，在get方法中return ArrayList.this.elementData(offset + index);这段代码可以清晰表明get所返回就是原列表offset + index位置的元素。同样的道理还有add方法里面的：

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. parent.add(parentOffset + index, e);
2. **this**.modCount = parent.modCount;

        remove方法里面的

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. E result = parent.remove(parentOffset + index);
2. **this**.modCount = parent.modCount;

        诚然，到了这里我们可以判断subList返回的SubList同样也是AbstractList的子类，同时它的方法如get、set、add、remove等都是在原列表上面做操作，它并没有像subString一样生成一个新的对象。所以subList返回的只是原列表的一个视图，它所有的操作最终都会作用在原列表上。

        那么从这里的分析我们可以得出上面的结果应该恰恰与我们上面的答案相反：

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. list1 == list2：**false**
2. list1 == list3：**true**

**Java细节（3.1）：subList返回的只是原列表的一个视图，它所有的操作最终都会作用在原列表上**

**二、subList生成子列表后，不要试图去操作原列表**

        从上面我们知道subList生成的子列表只是原列表的一个视图而已，如果我们操作子列表它产生的作用都会在原列表上面表现，但是如果我们操作原列表会产生什么情况呢？

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. **public** **static** **void** main(String[] args) {
2. List<Integer> list1 = **new** ArrayList<Integer>();
3. list1.add(1);
4. list1.add(2);
6. //通过subList生成一个与list1一样的列表 list3
7. List<Integer> list3 = list1.subList(0, list1.size());
8. //修改list3
9. list1.add(3);
11. System.out.println("list1'size：" + list1.size());
12. System.out.println("list3'size：" + list3.size());
13. }

        该实例如果不产生意外，那么他们两个list的大小都应该都是3，但是偏偏事与愿违，事实上我们得到的结果是这样的：

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. list1'size：3
2. Exception in thread "main" java.util.ConcurrentModificationException
3. at java.util.ArrayList$SubList.checkForComodification(Unknown Source)
4. at java.util.ArrayList$SubList.size(Unknown Source)
5. at com.chenssy.test.arrayList.SubListTest.main(SubListTest.java:17)

        list1正常输出，但是list3就抛出ConcurrentModificationException异常，看过我另一篇博客的同仁肯定对这个异常非常，fail-fast？不错就是fail-fast机制，在fail-fast机制中，LZ花了很多力气来讲述这个异常，所以这里LZ就不对这个异常多讲了（更多请点这里：[Java提高篇（三四）—–fail-fast机制](http://cmsblogs.com/?p=1220)）。我们再看size方法：

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. **public** **int** size() {
2. checkForComodification();
3. **return** **this**.size;
4. }

        size方法首先会通过checkForComodification验证，然后再返回this.size。

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. **private** **void** checkForComodification() {
2. **if** (ArrayList.**this**.modCount != **this**.modCount)
3. **throw** **new** ConcurrentModificationException();
4. }

        该方法表明当原列表的modCount与this.modCount不相等时就会抛出ConcurrentModificationException。同时我们知道modCount 在new的过程中 “继承”了原列表modCount，只有在修改该列表（子列表）时才会修改该值（先表现在原列表后作用于子列表）。而在该实例中我们是操作原列表，原列表的modCount当然不会反应在子列表的modCount上啦，所以才会抛出该异常。

        对于子列表视图，它是动态生成的，生成之后就不要操作原列表了，否则必然都导致视图的不稳定而抛出异常。最好的办法就是将原列表设置为只读状态，要操作就操作子列表：

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. //通过subList生成一个与list1一样的列表 list3
2. List<Integer> list3 = list1.subList(0, list1.size());
4. //对list1设置为只读状态
5. list1 = Collections.unmodifiableList(list1);

**Java细节（3.2）：生成子列表后，不要试图去操作原列表，否则会造成子列表的不稳定而产生异常**

**三、推荐使用subList处理局部列表**

        在开发过程中我们一定会遇到这样一个问题：获取一堆数据后，需要删除某段数据。例如，有一个列表存在1000条记录，我们需要删除100-200位置处的数据，可能我们会这样处理：

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. **for**(**int** i = 0 ; i < list1.size() ; i++){
2. **if**(i >= 100 && i <= 200){
3. list1.remove(i);
4. /\*
5. \* 当然这段代码存在问题，list remove之后后面的元素会填充上来，
6. \* 所以需要对i进行简单的处理，当然这个不是这里讨论的问题。
7. \*/
8. }
9. }

        这个应该是我们大部分人的处理方式吧，其实还有更好的方法，利用subList。在前面LZ已经讲过，子列表的操作都会反映在原列表上。所以下面一行代码全部搞定：

**[java]** [view plain](http://blog.csdn.net/chenssy/article/details/44102915) [copy](http://blog.csdn.net/chenssy/article/details/44102915)

[print?](http://blog.csdn.net/chenssy/article/details/44102915)[在CODE上查看代码片](https://code.csdn.net/snippets/613474)

1. list1.subList(100, 200).clear();

简单而不失华丽！！！！！

**参考资料：编写高质量代码：改善Java程序的151个建议**