

《手册》第 11-12 页对 `ArrayList` 的 `subList` 和 `Arrays.asList()` 进行了如下描述：

【强制】`ArrayList` 的 `subList` 结果不可强转成 `ArrayList`，否则会抛出 `ClassCastException` 异常，即 `java.util.RandomAccessSubList cannot be cast to java.util.ArrayList`。

【强制】在 `SubList` 场景中，高度注意对原集合元素的增加或删除，均会导致子列表的遍历、增加、删除产生 `ConcurrentModificationException` 异常。

【强制】使用工具类 `Arrays.asList()` 把数组转换成集合时，不能使用其修改集合相关的方法，它的 `add/remove/clear` 方法会抛出 `UnsupportedOperationException` 异常。

那么我们思考下面几个问题：

- 《手册》为什么要这么规定？
- 这对我们编码又有什么启发呢？

这些都是本节重点解答的问题。

通过前面章节的学习，相信很多人已经对通过使用类图、阅读源码和源码的注释等来学习方法已经轻车熟路了。

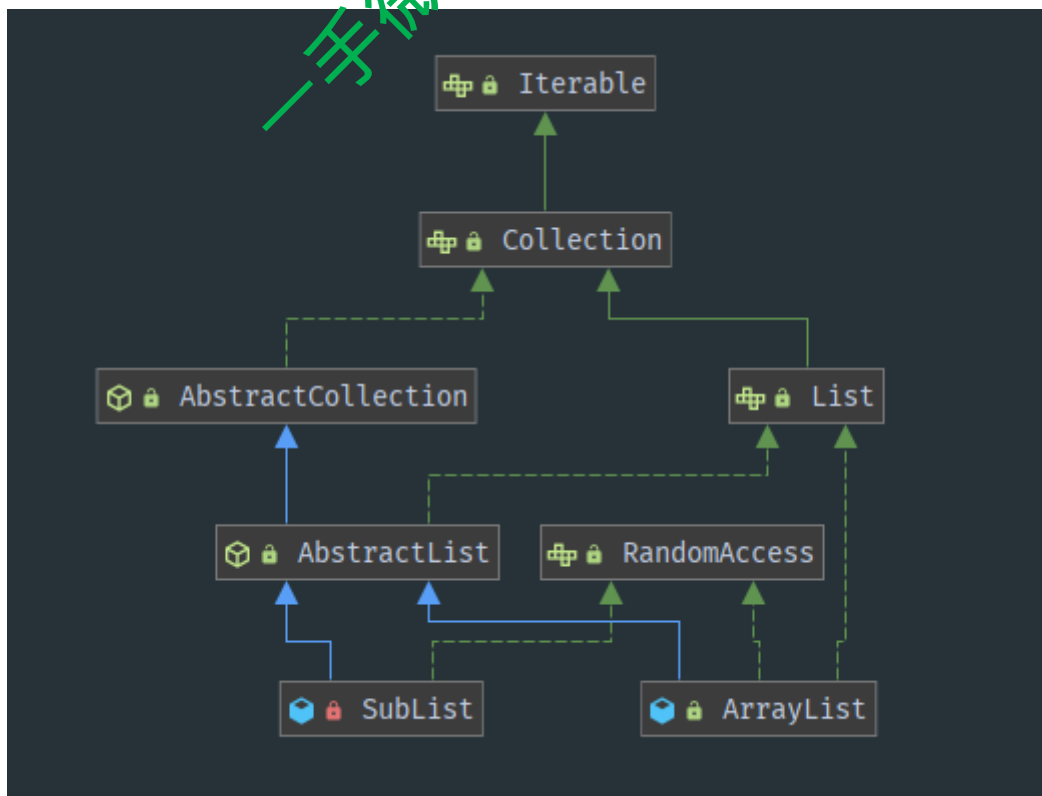
下面我们根据本节话题继续实战。

2.1 ArrayList 的 subList 分析

2.1.1 类图法

通过 IDEA 的提供的类图工具，我们可以查看该类的继承体系。

具体步骤：在 `SubList` 类中右键，选择“Diagrams”->“Show Diagram”。



可以看到 `SubList` 和 `ArrayList` 的继承体系非常类似，都实现了 `RandomAccess` 接口 继承自 `AbstractList`。

`SubList` 和 `ArrayList` 并没有继承关系，因此“`ArrayList` 的 `SubList` 并不能强转为 `ArrayList`”。

通过类图我们对 `SubList` 有了一个整体的了解，这将为我们进步学习打下很好的基础。

2.2.2 DEMO 和调试大法

如果想学习某个特性，最好的方法之一就是写一个小段 DEMO 来观察分析。

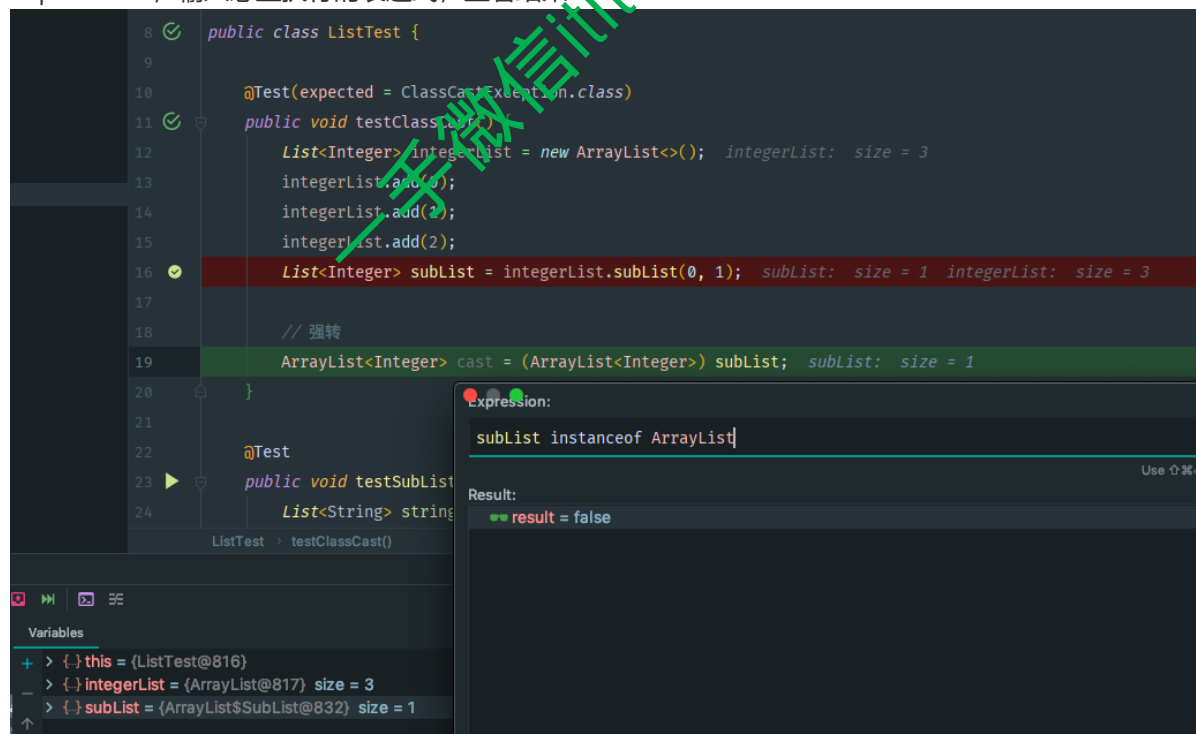
因此我们下面，写一个简单的测试代码片段来验证转换异常问题：

```
@Test(expected = ClassCastException.class)
public void testClassCast() {
    List<Integer> integerList = new ArrayList<>();
    integerList.add(0);
    integerList.add(1);
    integerList.add(2);
    List<Integer> subList = integerList.subList(0, 1);

    ArrayList<Integer> cast = (ArrayList<Integer>) subList;
}
```

我们还可以使用调试的表达式功能来验证我们的想法。

在调试界面的“Variables”窗口选择想研究的对象，如 `subList`，然后右键选择“Evaluate Expression”，输入想查执行的表达式，查看结果：



从上面的表达式的结果也可以清晰地看出，`subList` 并不是 `ArrayList` 类型的实例。

我们写一个代码片段来验证功能：

```
@Test
public void testSubList() {
    List<String> stringList = new ArrayList<>();
}
```

```

        stringList.add("赵");
        stringList.add("钱");
        stringList.add("孙");
        stringList.add("李");
        stringList.add("周");
        stringList.add("吴");
        stringList.add("郑");
        stringList.add("王");

        List<String> subList = stringList.subList(2, 4);
        System.out.println("子列表: " + subList.toString());
        System.out.println("子列表长度: " + subList.size());

        subList.set(1, "慕容");
        System.out.println("子列表: " + subList.toString());
        System.out.println("原始列表: " + stringList.toString());
    }

```

输出结果为:

```

子列表: [孙, 李]
子列表长度: 2
子列表: [孙, 慕容]
原始列表: [赵, 钱, 孙, 慕容, 周, 吴, 郑, 王]

```

可以观察到, 对子列表的修改最终对原始列表产生了影响。

那么为啥修改子序列的索引为 1 的值影响的是原始列表的第 4 个元素呢? 后面将进行分析和解读。

2.1.3 源码分析

java.util.ArrayList#subList 源码:

```

public List<E> subList(int fromIndex, int toIndex) {
    subListRangeCheck(fromIndex, toIndex, size);
    return new SubList(this, 0, fromIndex, toIndex);
}

```

通过源码可以看到该方法主要有两个核心逻辑: 一个是检查索引的范围, 一个是构造子列表对象。

通注释我们可以学到核心知识点:

该方法返回本列表中 fromIndex (包含) 和 toIndex (不包含) 之间的**元素视图**。如果两个索引相等会返回一个空列表。

如果需要对 list 的某个范围的元素进行操作, 可以用 subList, 如:

```
list.subList(from, to).clear();
```

任何对子列表的操作最终都会反映到原列表中。

我们查看函数 java.util.ArrayList.SubList#set 源码:

```

public E set(int index, E e) {
    rangeCheck(index);
    checkForComodification();
    E oldValue = ArrayList.this.elementData(offset + index);
    ArrayList.this.elementData[offset + index] = e;
    return oldValue;
}

```

可以看到替换值的时候，获取索引是通过 `offset + index` 计算得来的。

这里的 `java.util.ArrayList#elementData` 即为原始列表存储元素的数组。

```

SubList(AbstractList<E> parent,
        int offset, int fromIndex, int toIndex) {
    this.parent = parent;
    this.parentOffset = fromIndex;
    this.offset = offset + fromIndex;
    this.size = toIndex - fromIndex;
    this.modCount = ArrayList.this.modCount;
}

```

通过子列表的构造函数我们知道，这里的偏移量 (`offset`) 的值为 `fromIndex` 参数。

因此上小节提到的：** 为啥子序列的索引为 1 的值影响的是原始列表的第 4 个元素呢？ ** 的问题就不言自明了。

另外在 `SubList` 的构造函数中，会将 `ArrayList` 的 `modCount` 赋值给 `SubList` 的 `modCount`。

我们再回到规约中规定：

【强制】在 `subList` 场景中，高度注意对原集合元素的增加或删除，均会导致子列表的遍历、增加、删除产生 `ConcurrentModificationException` 异常。

我们看 `java.util.ArrayList#add(E)` 的源码：

```

public boolean add(E e) {
    ensureCapacityInternal(size + 1);
    elementData[size++] = e;
    return true;
}

```

可以发现新增元素和删除元素，都会对 `modCount` 进行修改。

我们再看 `SubList` 的核心的函数，如 `java.util.ArrayList.SubList#get` 和 `java.util.ArrayList.SubList#size`：

```
public E get(int index) {
    rangeCheck(index);
    checkForComodification();
    return ArrayList.this.elementData(offset + index);
}

public int size() {
    checkForComodification();
    return this.size;
}
```

都会进行修改检查：

```
java.util.ArrayList.SubList#checkForComodification
```

```
private void checkForComodification() {
    if (ArrayList.this.modCount != this.modCount)
        throw new ConcurrentModificationException();
}
```

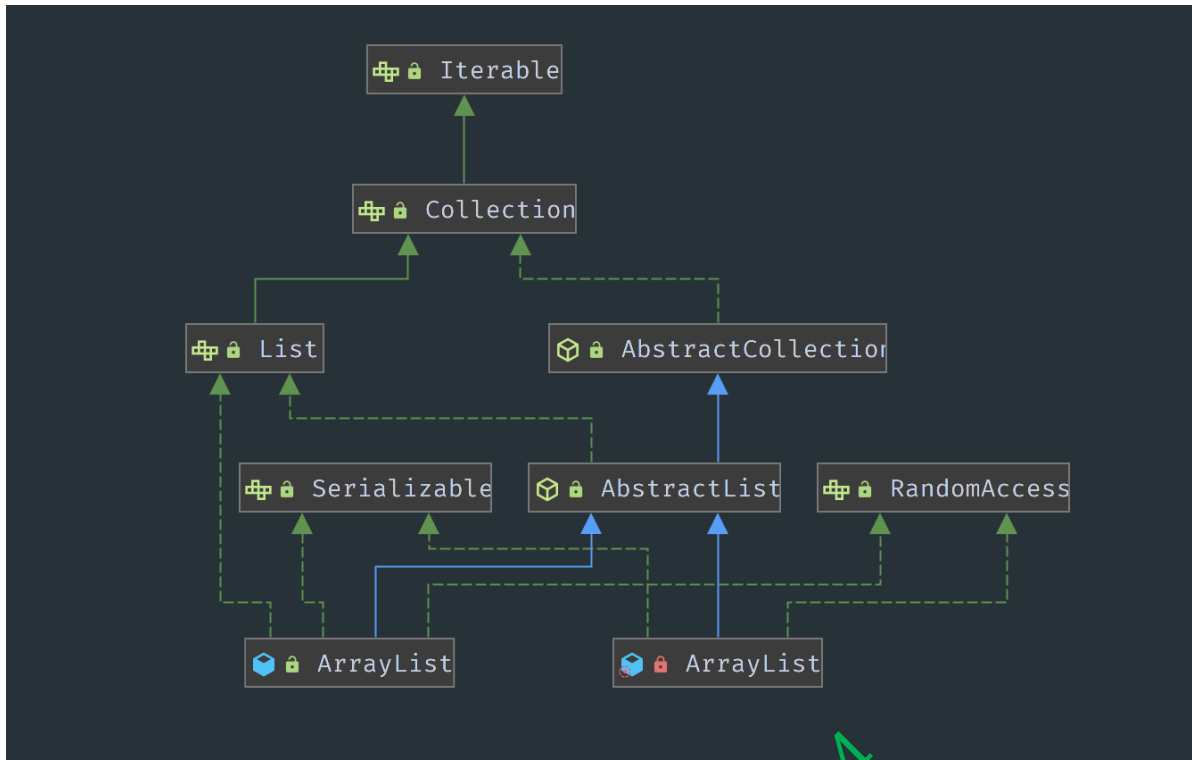
而从上面的 `SubList` 的构造函数我们可以看到，`SubList` 复制了 `ArrayList` 的 `modCount`，因此对原函数的新增或删除都会导致 `ArrayList` 的 `modCount` 的变化。而子列表的遍历、增加、删除时又会检查创建 `SubList` 时的 `modCount` 是否一致，显然此时两者会不一致，导致抛出 `ConcurrentModificationException` (并发修改异常)。

至此上面约定的原因我们也非常明白了。

2.2 Arrays.asList () 分析

2.2.1 类图法

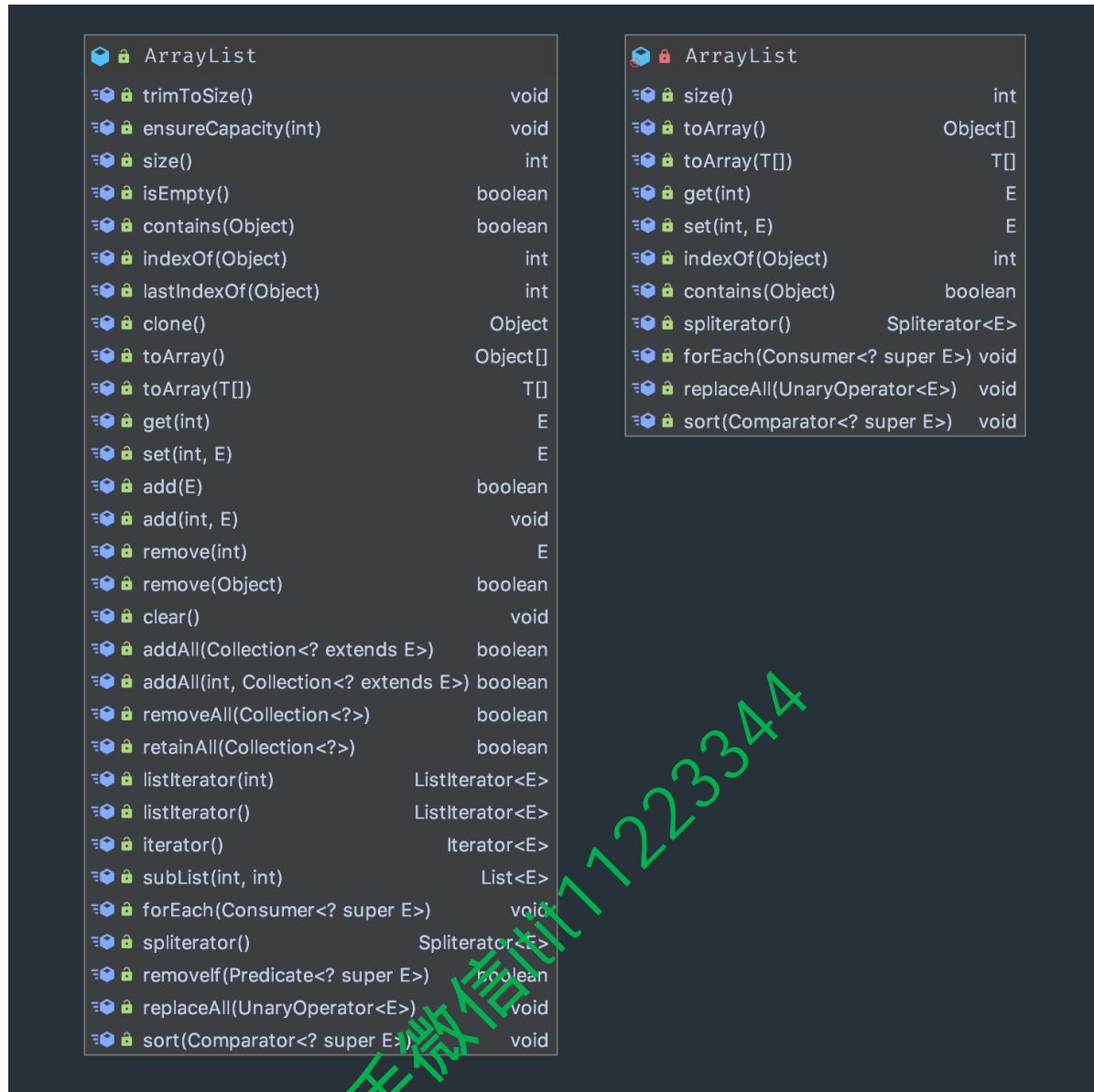
和前面一样，查看类图来了解 `Arrays.asList()` 的返回类型。



发现该 `java.util.Arrays.ArrayList` (右侧) 和 `java.util.ArrayList` (左侧)，的继承体系非常相似，继承自 `java.util.AbstractList`。

一手微信it171223344

我们打开左上角的“Method”功能，对比两者的主要函数的异同：



我们可以清楚地发现，`java.util.Arrays.ArrayList` (右侧) 并没有像左侧一样重写 `add`、`remove` 函数。

2.2.2 源码大法

接下来我们分析 `Arrays.asList()` 的源码：

```
@SafeVarargs
@SuppressWarnings("varargs")
public static <T> List<T> asList(T... a) {
    return new ArrayList<>(a);
}
```

通过注释我们可以得到下面的要点：

返回基于特定数组的**定长列表**。

该方法扮演数组到集合的桥梁。

该方法也提供了包含多个元素的定长列表的方法：

```
List stooges = Arrays.asList("Larry", "Moe", "Curly");
```

可看出此方法的功能是为了返回**定长的列表**。

这里的“定长列表”的描述非常重要，这也就解释了为什么不支持增加和删除元素的原因。

结合前面的类图，我们去查看 `AbstractList` 的 `add` 和 `remove` 相关函数：

```
java.util.AbstractList#add(int, E)
```

```
public void add(int index, E element) {  
    throw new UnsupportedOperationException();  
}
```

```
java.util.AbstractList#remove
```

```
public E remove(int index) {  
    throw new UnsupportedOperationException();  
}
```

可知如果子类不重写这两个函数，就会抛出 `UnsupportedOperationException`（不支持的操作异常）。

我们再看看 `java.util.AbstractList#clear` 的源码：

```
public void clear() {  
    removeRange(0, size())  
}
```

通过注释可知 如果没有重写 `remove(int index)` 或 `removeRange(int fromIndex, int toIndex)` 同样也会抛出 `UnsupportedOperationException`。

在 Java 的学习过程中，大多数人都是通过看视频，读博客，搜索引擎搜索，买书等来学习知识。

但是很多资料都是告诉你结论，但这样容易浮于表面，知其然而不知其所以然。而源码、官方文档等才是权威的知识。

希望从现在开始学习和开发中能够偶尔到感兴趣的类中查看源码，这样学的更快，更扎实。通过进入源码中自主研究，这样印象更加深刻，掌握的程度更深。

我们同样发现学习的手段并非只有一种，往往多种研究方式结合起来效果最好。

本文通过类图分析、源码分析以及 DEMO 和调试的方式对 `ArrayList` 的 `SubList` 问题和 `Arrays` 的 `asList` 进行分析。并根据分析阐述了对我们学习的启发。

本节的要点：

1. `ArrayList` 内部类 `SubList` 和 `ArrayList` 没有继承关系，因此无法将其强转为 `ArrayList`。

2. `ArrayList` 的 `subList` 构造时传入 `ArrayList` 的 `modCount`，因此对原列表的修改将会导致子列表的遍历、增加、删除产生 `ConcurrentModificationException` 异常。
3. `Arrays.asList()` 函数是提供通过数组构造定长集合的功能，该函数提供数组到集合的桥梁。

下一节我们将讲述添加注释的正确姿势。

《手册》第 11 页 集合处理章节有这么一条规定：

【强制】不要在 `foreach` 循环里进行元素的 `remove/add` 操作。`remove` 元素请使用 `Iterator` 方式，如果并发操作，需要对 `Iterator` 对象加锁。

那么问题来了，为什么“不要在 `foreach` 循环里进行元素的 `remove/add` 操作。`remove` 元素请使用 `Iterator` 方式”？

请大家结合前面和本小节所学的内容自己实际动手研究一下。

```
}
```

一手微信itit1223344