<https://mp.weixin.qq.com/s/vxPhQh4Nig7q9dGkLkz-9A>

# 一个List，怎么这么多坑？我太南了

List 可谓是我们经常使用的集合类之一，几乎所有业务代码都离不开 List。既然天天在用，那就没准就会踩中这几个 List 常见坑。

今天我们就来总结这些常见的坑在哪里，捞自己一手，防止后续同学再继续踩坑。

本文涉及知识点如下：



## **ArrayList 这是李逵，还是李鬼？**

以前实习的时候，写过这样一段简单代码，通过 Arrays#asList 将数组转化为 List 集合。



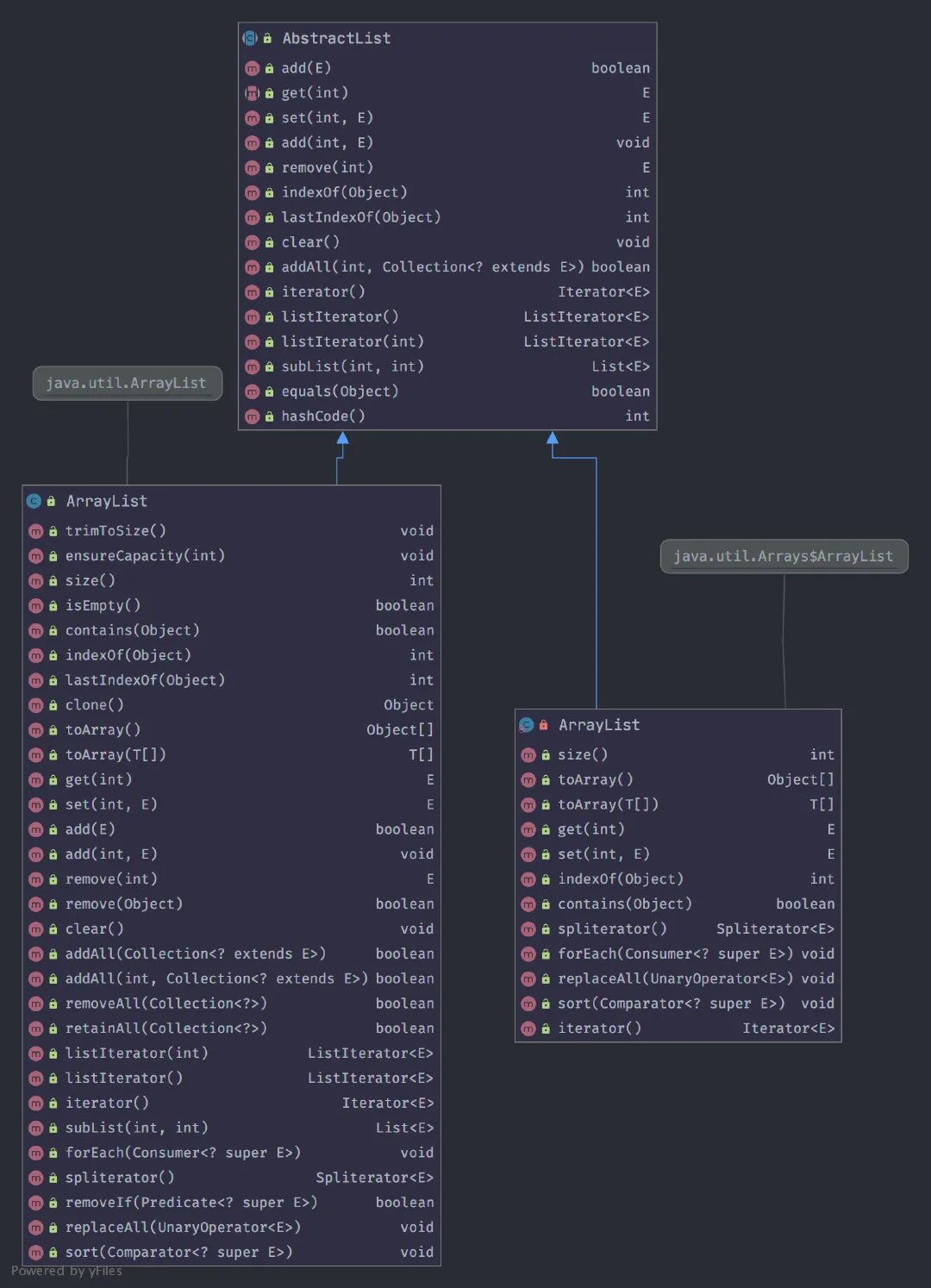
这段代码表面看起来没有任何问题，编译也能通过，但是真正测试运行的时候将会在第 4 行抛出  UnsupportedOperationException。

刚开始很不解，Arrays#asList 返回明明也是一个 ArrayList，为什么添加一个元素就会报错？这以后还能好好新增元素吗？



最后通过 Debug 才发现这个Arrays#asList 返回的 ArrayList 其实是个**李鬼**，仅仅只是 Arrays 一个内部类，并非真正的 java.util.ArrayList。

通过 IDEA，生成这两个的类图，如下：



从上图我们发现，add/remove 等方法实际都来自 AbstractList，而 java.util.Arrays$ArrayList 并没有重写父类的方法。而父类方法恰恰都会抛出 UnsupportedOperationException。



这就是为什么这个李鬼  ArrayList 不支持的增删的实际原因。

## **你用你的新 List，为什么却还互相影响**

李鬼 ArrayList 除了不支持增删操作这个坑以外，还存在另外一个大坑，改动内部元素将会同步影响原数组。



输出结果：

arrays:[modify\_1, modify\_2, 3]  
list:[modify\_1, modify\_2, 3]

从日志输出可以看到，不管我们是修改原数组，还是新 List 集合，两者都会互相影响。

查看 java.util.Arrays$ArrayList 实现，我们可以发现底层实际使用了原始数组。



知道了实际原因，修复的办法也很简单，套一层 ArrayList 呗！

List<String> list = **new** ArrayList<>(Arrays.asList(arrays));

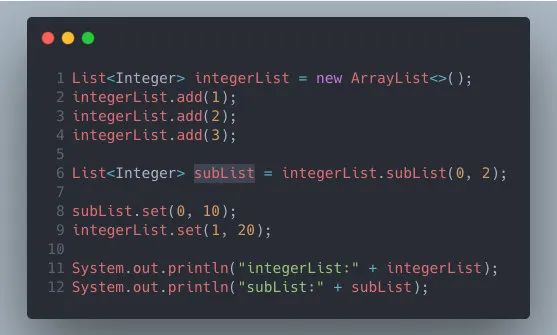
不过这么写感觉十分繁琐，推荐使用 **Guava Lists** 提供的方法。

List<String> list = Lists.newArrayList(arrays);

通过上面两种方式，我们将新的 List 集合与原始数组解耦，不再互相影响，同时由于此时还是真正的 ArrayList，不用担心 add/remove报错了。

除了 Arrays#asList产生新集合与原始数组互相影响之外，JDK 另一个方法 List#subList 生成新集合也会与原始 List 互相影响。

我们来看一个例子：



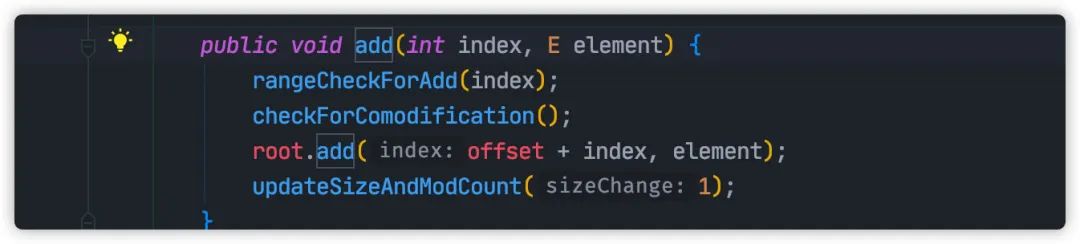
日志输出结果：

integerList:[10, 20, 3]  
subList:[10, 20]

查看  List#subList  实现方式，可以发现这个 SubList 内部有一个 parent 字段保存保存最原始 List 。

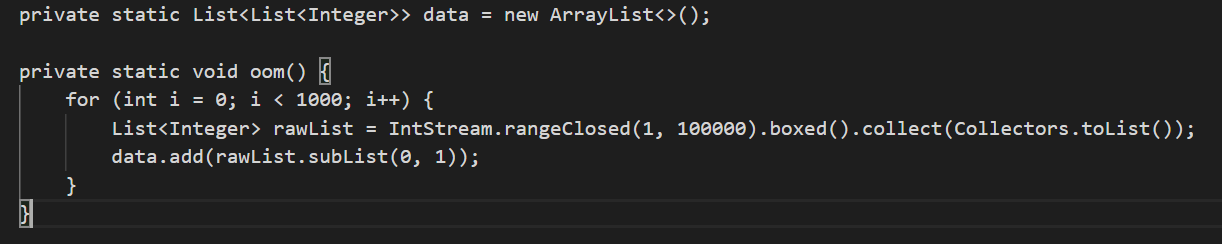


所有外部读写动作看起来是在操作 SubList ，实际上底层动作却都发生在原始 List 中，比如 add 方法：



另外由于 SubList 实际上还在引用原始 List，业务开发中，如果不注意，很可能产生 **OOM** 问题。

以下例子来自于极客时间：**Java业务开发常见错误100例**



data 看起来最终保存的只是 1000 个具有 1 个元素的 List，不会占用很大空间。但是程序很快就会 **OOM**。

**OOM** 的原因正是因为每个 SubList 都强引用个一个 10 万个元素的原始 List，导致 GC 无法回收。

这里修复的办法也很简单，跟上面一样，也来个套娃呗，加一层 ArrayList 。

## **不可变集合，说好不变，你怎么就变了**

为了防止 List 集合被误操作，我们可以使用 Collections#unmodifiableList 生成一个不可变（**immutable**）集合，进行防御性编程。

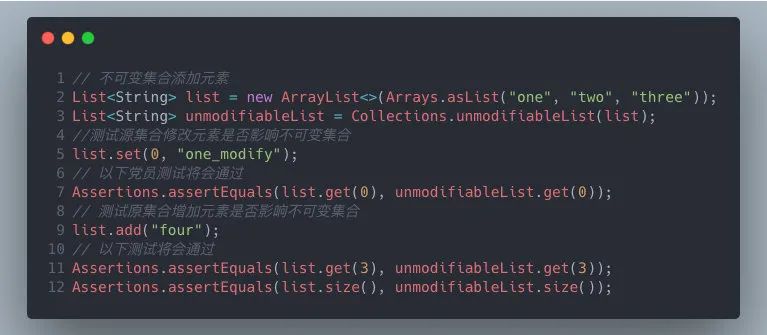
这个不可变集合只能被读取，不能做任何修改，包括增加，删除，修改，从而保护不可变集合的安全。



上面最后三行写操作都将会抛出 UnsupportedOperationException 异常

但是你以为这样就安全了吗？

如果有谁不小心改动原始 List，你就会发现这个不可变集合，竟然就变了。。。



上面单元测试结果将会全部通过，这就代表 Collections#unmodifiableList 产生不可变集合将会被原始 List 所影响。

查看  Collections#unmodifiableList  底层实现方法：



可以看到这跟上面 SubList 其实是同一个问题，新集合底层实际使用了**原始 List**。

由于不可变集合所有修改操作都会报错，所以不可变集合不会产生任何改动，所以并不影响的原始集合。但是防过来，却不行，原始 List 随时都有可能被改动，从而影响不可变集合。

可以使用如下两种方式防止这个情况。

**使用 JDK9 List#of 方法。**

List<String> list = **new** ArrayList<>(Arrays.asList("one", "two", "three"));  
List<String> unmodifiableList = List.of(list.toArray(**new** String[]{}));

**使用 Guava immutable list**

List<String> list = **new** ArrayList<>(Arrays.asList("one", "two", "three"));  
List<String> unmodifiableList = ImmutableList.copyOf(list);

相比而言 Guava 方式比较清爽，使用也比较简单，推荐使用 Guava 这种方式生成不可变集合。

## **foreach 增加/删除元素大坑**

先来看一段代码：



上面的代码我们使用 foreach 方式遍历 List 集合，如果符合条件，将会从集合中删除改元素。

这个程序编译正常，但是运行时，程序将会发生异常，日志如下：



可以看到程序最终错误是由 ArrayList$Itr.next 处的代码抛出，但是代码中我们并没有调用该方法，为什么会这样?

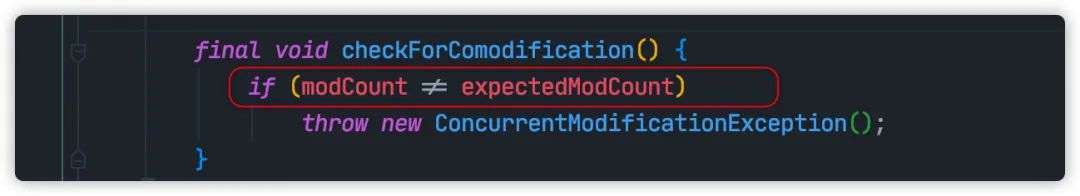
实际是因为 foreach 这种方式实际上 Java 给我们提供的一种语法糖，编译之后将会变为另一种方式。

我们将上面的代码产生 class 文件反编译来看下最后代码长的啥样。



可以看到 foreach 这种方式实际就是 Iterator 迭代器实现方式，这就是为什么 foreach 被遍历的类需要实现 Iterator接口的原因。

接着我们来看下抛出异常方法：



expectedModCount 来源于 list#iterator 方法：



也就是说刚开始遍历循环的时候 expectedModCount==modCount，下面我们来看下 modCount。

modCount 来源于 ArrayList 的父类  AbstractList，可以用来记录 List 集合被修改的次数。



ArrayList#remove 之后将会使 modCount 加一，expectedModCount与 modCount将会不相等，这就导致迭代器遍历时将会抛错。



modCount 计数操作将会交子类自己操作，ArrayList 每次修改操作（增、删）都会使 modCount 加 1。但是如 CopyOnWriteArrayList 并不会使用 modCount 计数。

所以 CopyOnWriteArrayList 使用 foreach 删除是安全的，但是还是建议使用如下两种删除元素，统一操作。

修复的办法有两种：

**使用 Iterator#remove 删除元素**



**JDK1.8 List#removeIf**

****

推荐使用 JDK1.8 这种方式，简洁明了。

**思考**

如果我将上面 foreach 代码判断条件简单修改一下：

****

运行这段代码，可以发现这段代码又不会报错了，有没有很意外？

## **总结**

第一，我们不要先入为主，想当然就认为 Arrays.asList 和 List.subList 就是一个普通，独立的 ArrayList。

如果没办法，使用了 Arrays.asList 和 List.subList ，返回给其他方法的时候，一定要记得再套一层真正的 java.util.ArrayList。

第二 JDK 的提供的不可变集合实际非常笨重，并且低效，还不安全，所以推荐使用 Guava 不可变集合代替。

最后，切记，不要随便在 foreach增加/删除元素。