# 并发容器概念

## 概述

实际上从java5，java6开始，我们主键拥有了比较好用的并发容器，在此之前呢是没有的，这些好用的并发容器，大大提高了我们在并发情况下能力，这里的容器呢是广义的，包括我们的Map，List，Set，Queue啊，这些都包含进来，他们本身都是可以用来存储东西的，我们就一起来看下一共有哪些常用的并发容器？

## 常用的并发容器

### ConcurrentHashMap：线程安全的HashMap

HashMap我们会用的非常多啊，它是一个Map，那Map会分为Key，value，但是本身这个HashMap他不是一个线程安全的，所以当我们需要多线程同时操作的时间，需要用到我们的ConcurrentHashMap。

### CopyOnWriteArrayList：线程安全的List

List我们用的也是非常多了，我们平时用的实现一般是ArrayList，但是同样，ArrayList也是线程不安全的，所以就利用我们的CopyOnWriteArrayList，这是一个线程安全的，在多线程得情况下，不会出现并发安全问题

### 3、BlockingQueue：这是一个接口，表示阻塞队列，非常适合用于作为数据共享的通道

### 4、ConcurrentLinkedQueue：高效的非阻塞并发队列，使用链表实现。可以看做一个线程安全的LinkedList。

我们可以把他理解为是一个线程安全的LinkedList，关于List我们有ArrayList，也有LinkedList，他们的实现一个是数组，一个是链表，那ConcurrentLinkedQueue我们就可以把它认为是一个线程安全的链表实现的List，

### 5、ConcurrentSkipListMap：是一个Map，使用跳表的数据结构进行快速查找

# 2、趣说集合的历史 -- 古老和过时的同步容器

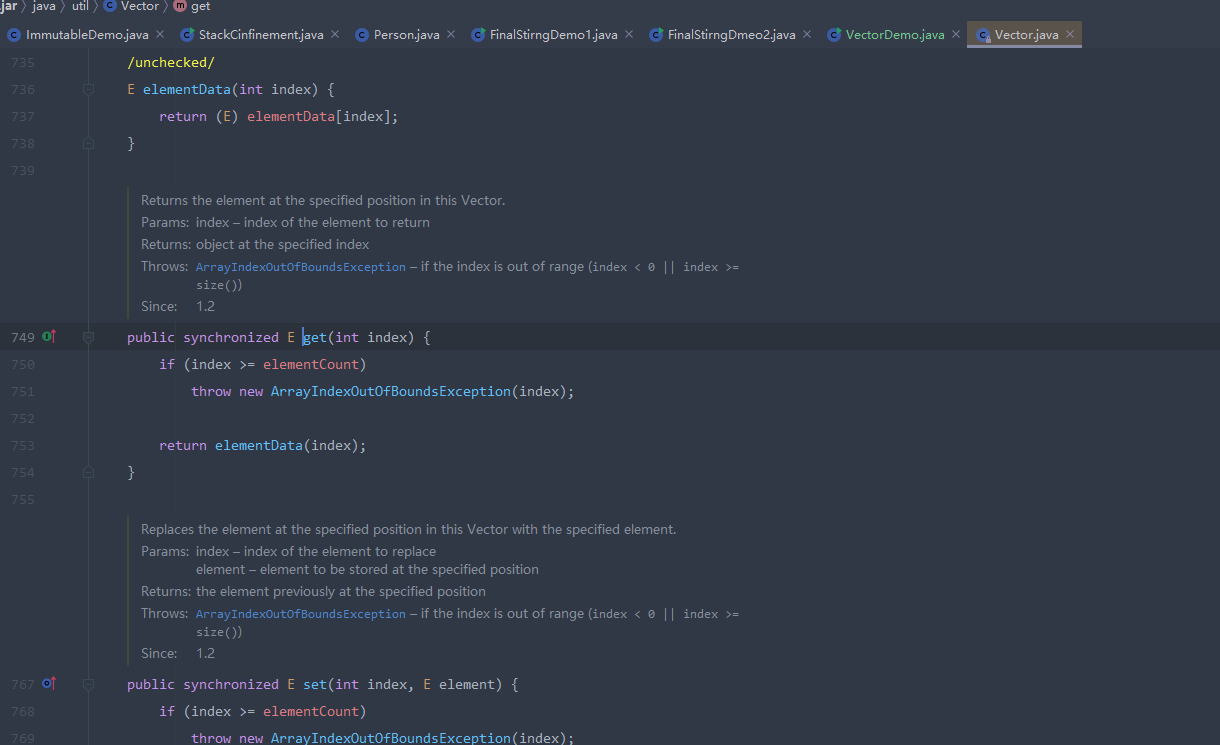
## 1、Vector和Hashtable(早期并发容器)

### 1、Vector

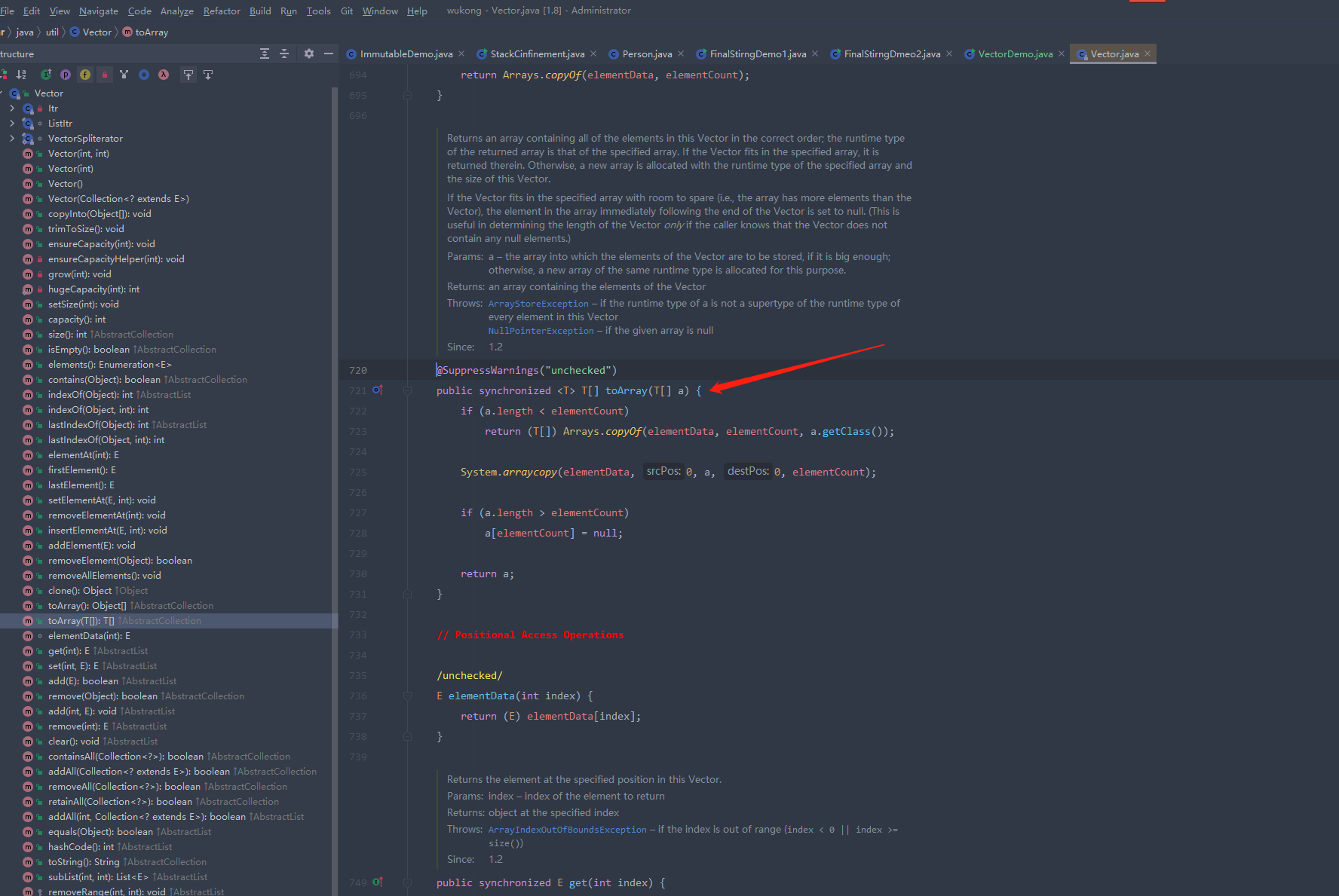
他们是JDK早期的一部分，他们的目标时提供一个线程安全的集合类，不过呢随着我们JDK的发展，他已经不能满足当今的需求了，比如说他们的性能不够好，并且他们对复合操作支持的不够好，比如说有其它线程并发的修改这个容器里面的内容的时候，他们可能会抛出异常，所以它存在这样那样的缺陷。

我们刚才提到他最大的缺陷是性能问题，所以我们就来看下为什么性能不好。

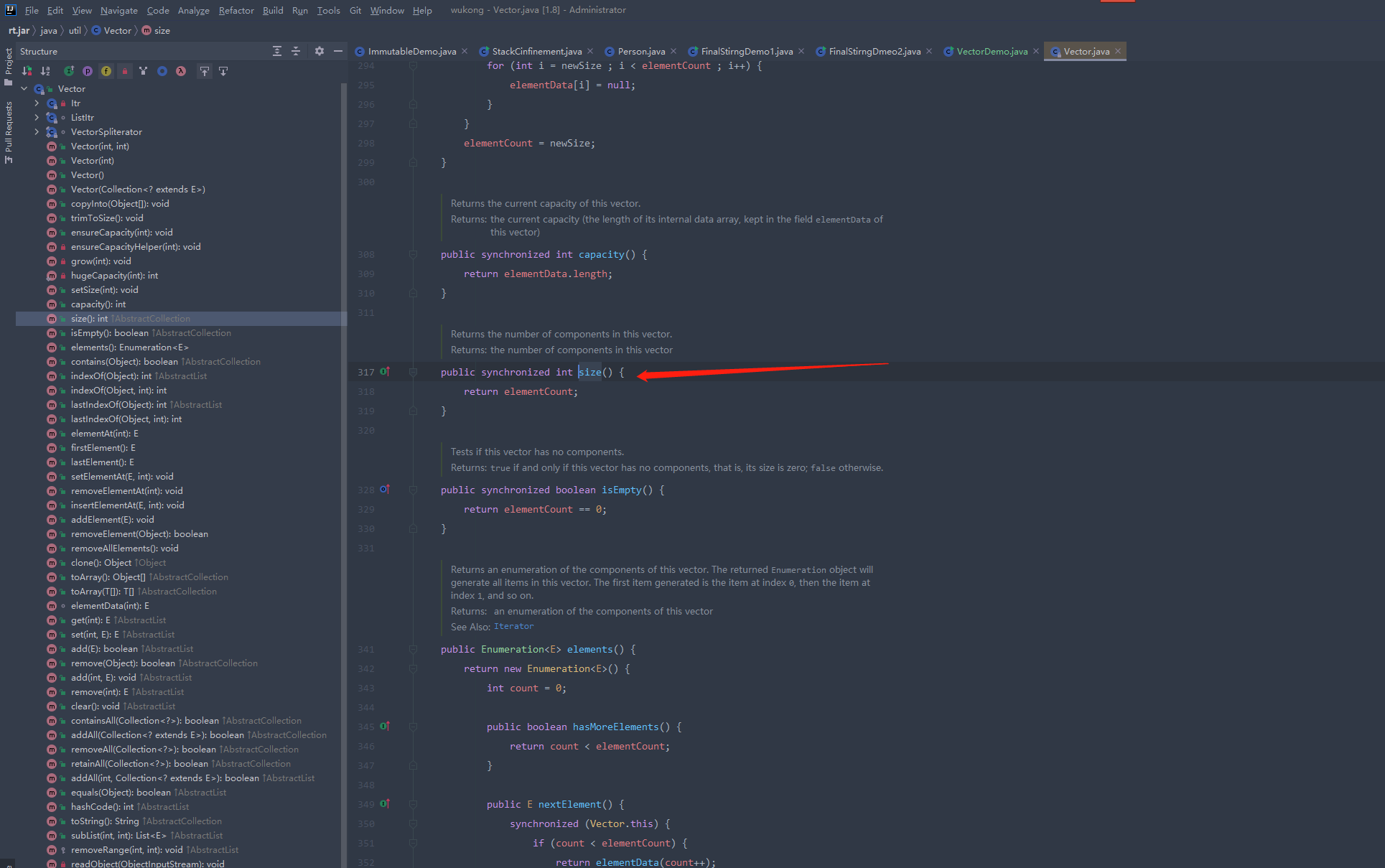
首先我们来看下Vector的get方法实现



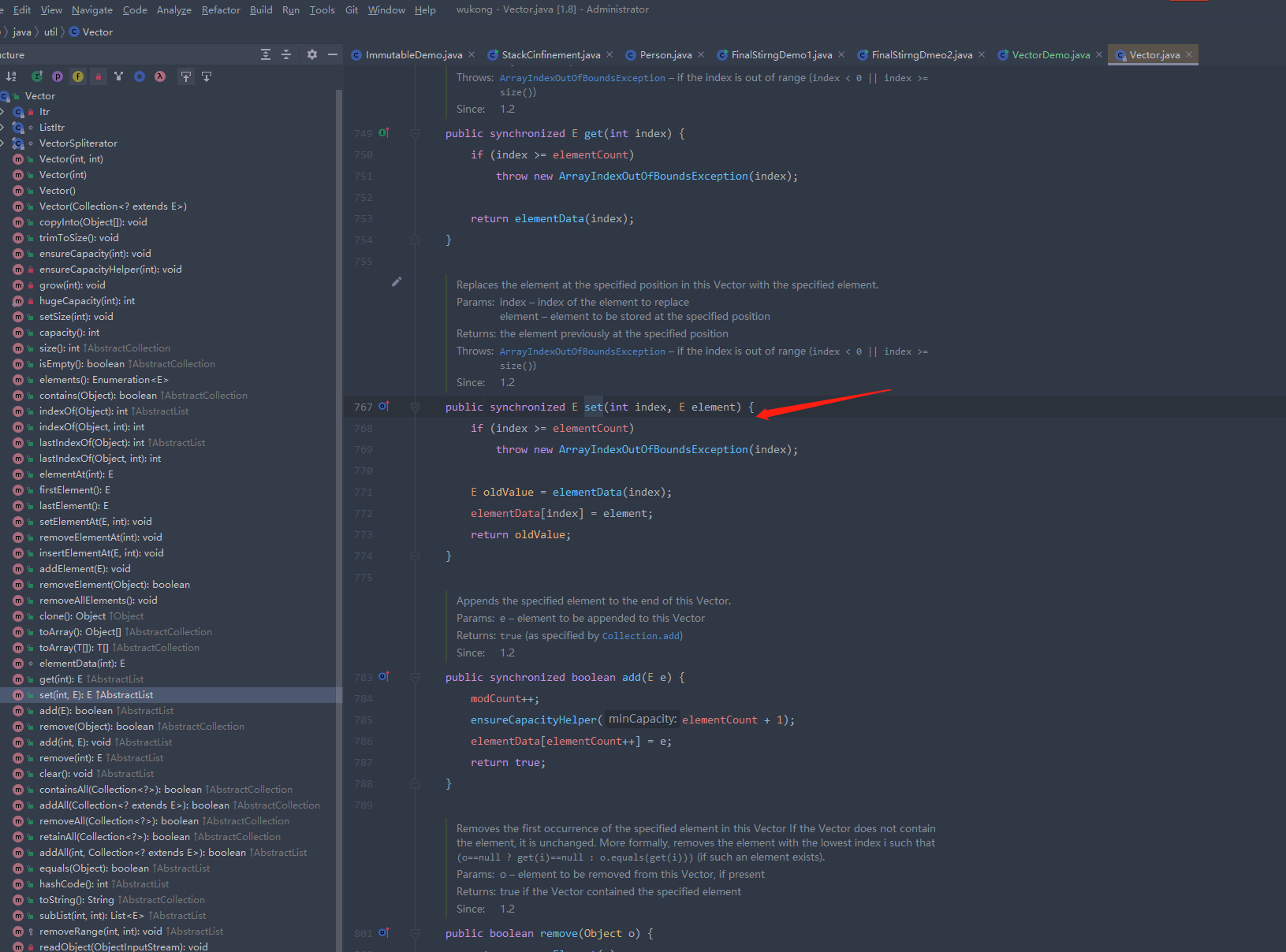
首先呢，get方法是被synchronized修饰的，而且他这里的synchronized是修饰在方法上的，也就意味着如果在Victor这个类中的其他方法上也有synchronized的话，他们是不能在同一时刻多个线程同时执行，所以我们再来看下这个类中的其他方法。



比如说这个toArray()也是被synchronized修饰。



这边的size方法也是被synchronized修饰。



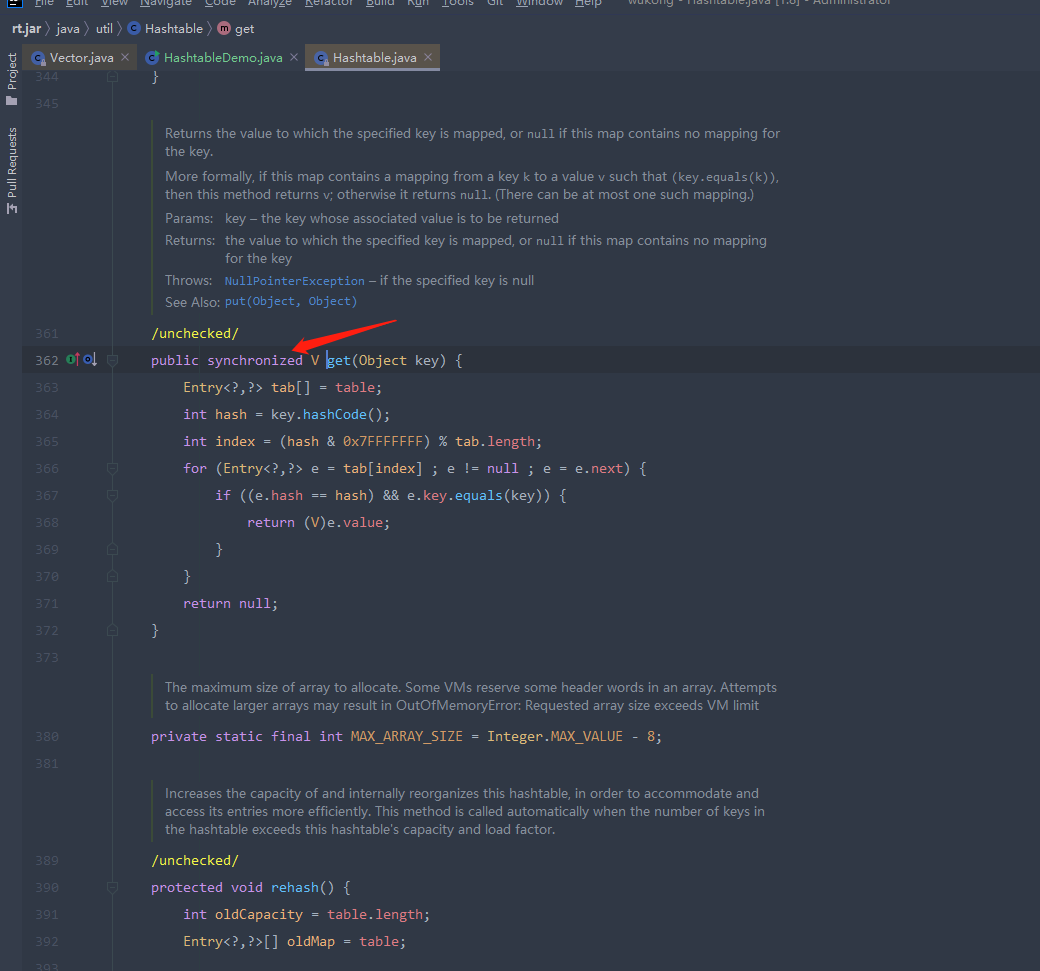
set方法也是被synchronized修饰，由于synchronized这个性质啊，它导致在并发量大的时候呢，没有办法做到性能特别好，所以你看整个这个类方法特别多，这么几十个方法，如果说有很多方法都是被synchronized保护的话，那么这个类虽然是线程安全的，但是呢在多线程下，他的性能也不好，所以啊，这就是这个类他的特点。

### 2、Hashtable

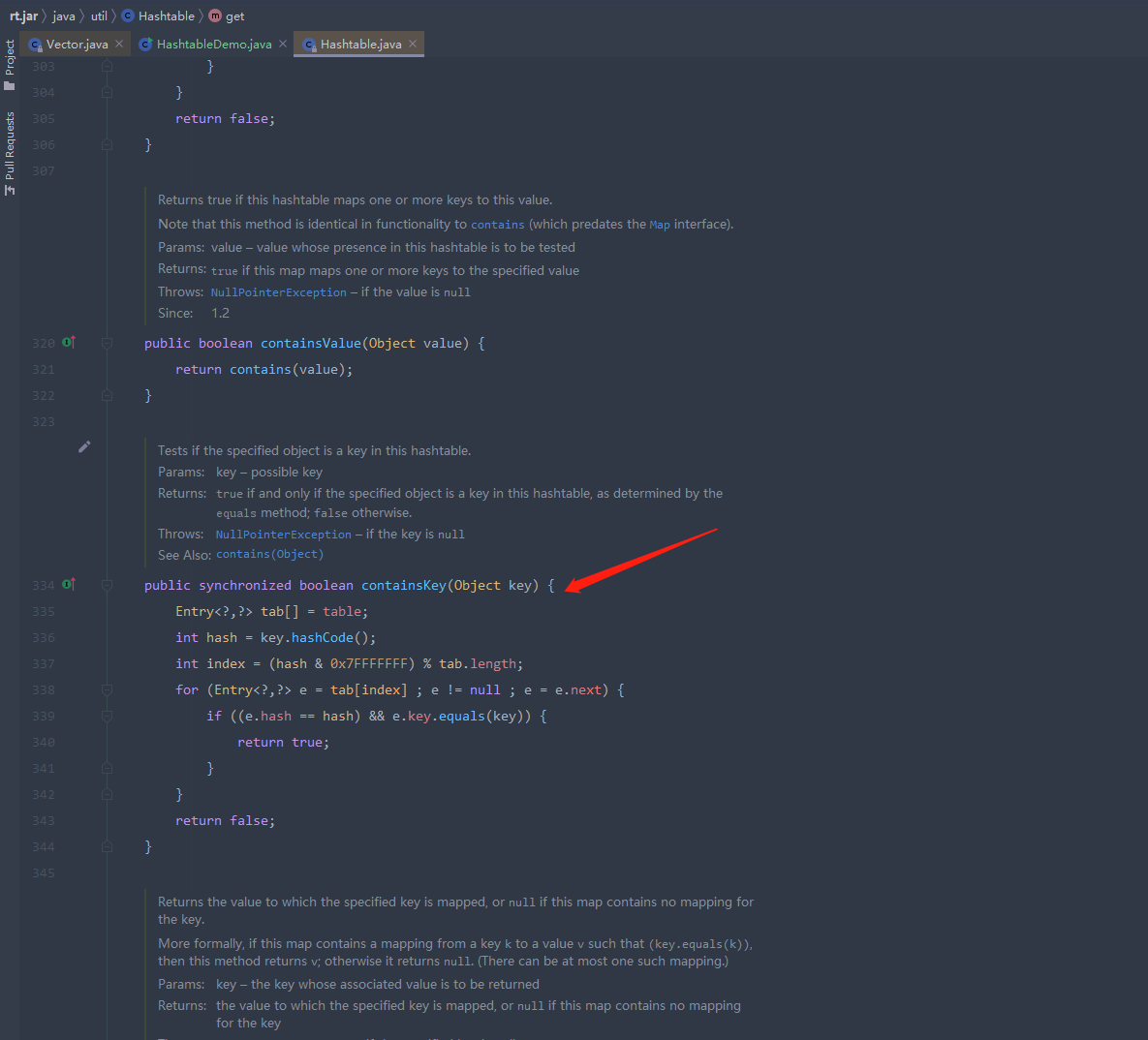
我们把Vector理解为ArrayList，那Hashtable你可以理解为就是一个线程安全的HashMap，所以啊，我们就来看一下Hashtable。

我们来看下为什么Hashtable也是性能不高的，我们来看下它的源码。

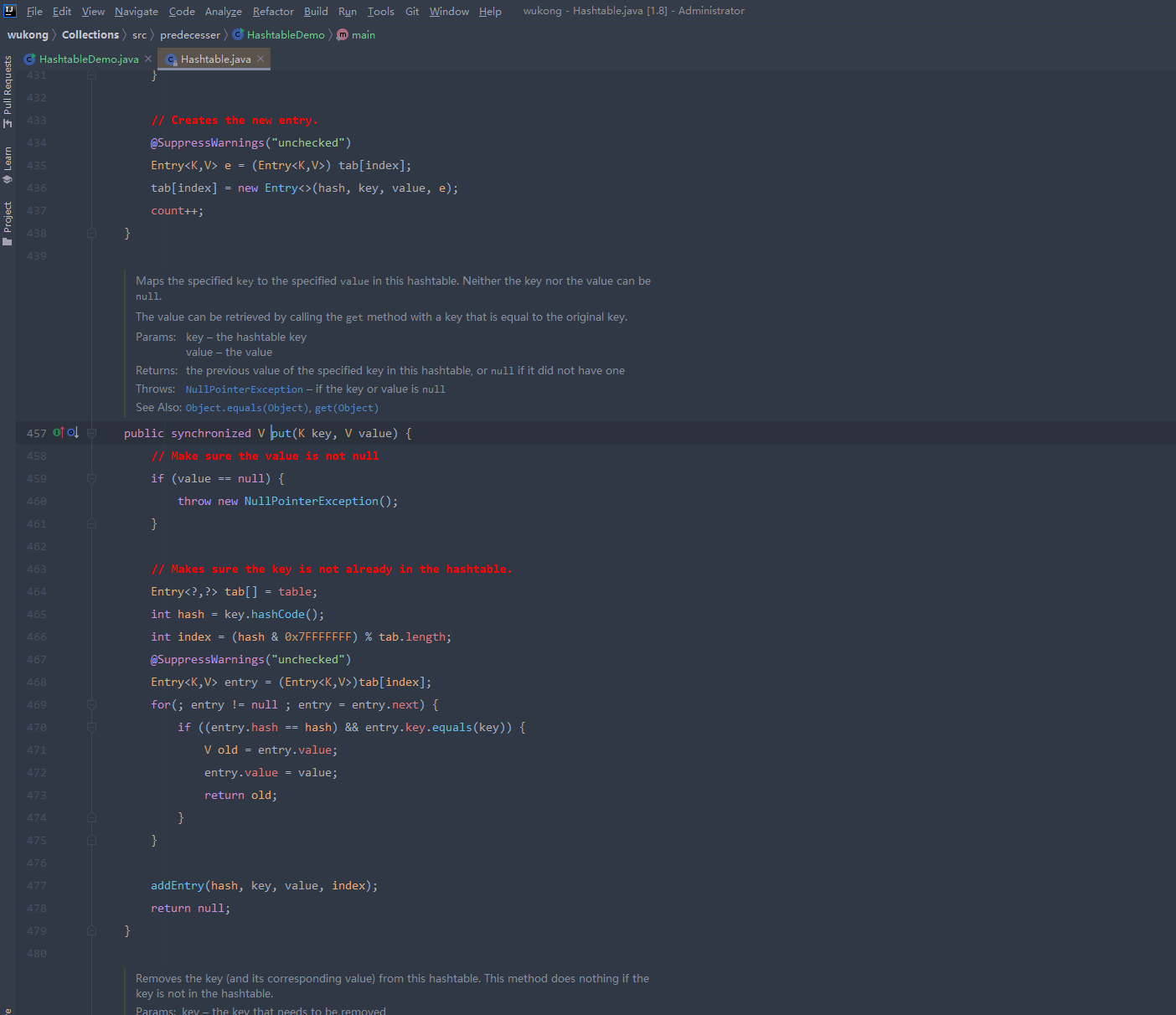
get()源码



这又是一个用synchronized修饰的一个方法，我们再来看下其他的方法也是一样的。



我们使用put()方法也是被synchronized修饰的



这已经充分地表明了Vector和Hashtable其实是一起的，他们的原理呢也是一样的，都是说把这个类中的所有方法上直接加上synchronized，简单明了，要是让我们来设计呢，一开始世界设计成这样，其实也不错，

### 总结

Vector和Hashtable最主要的问题就是并发性能力差

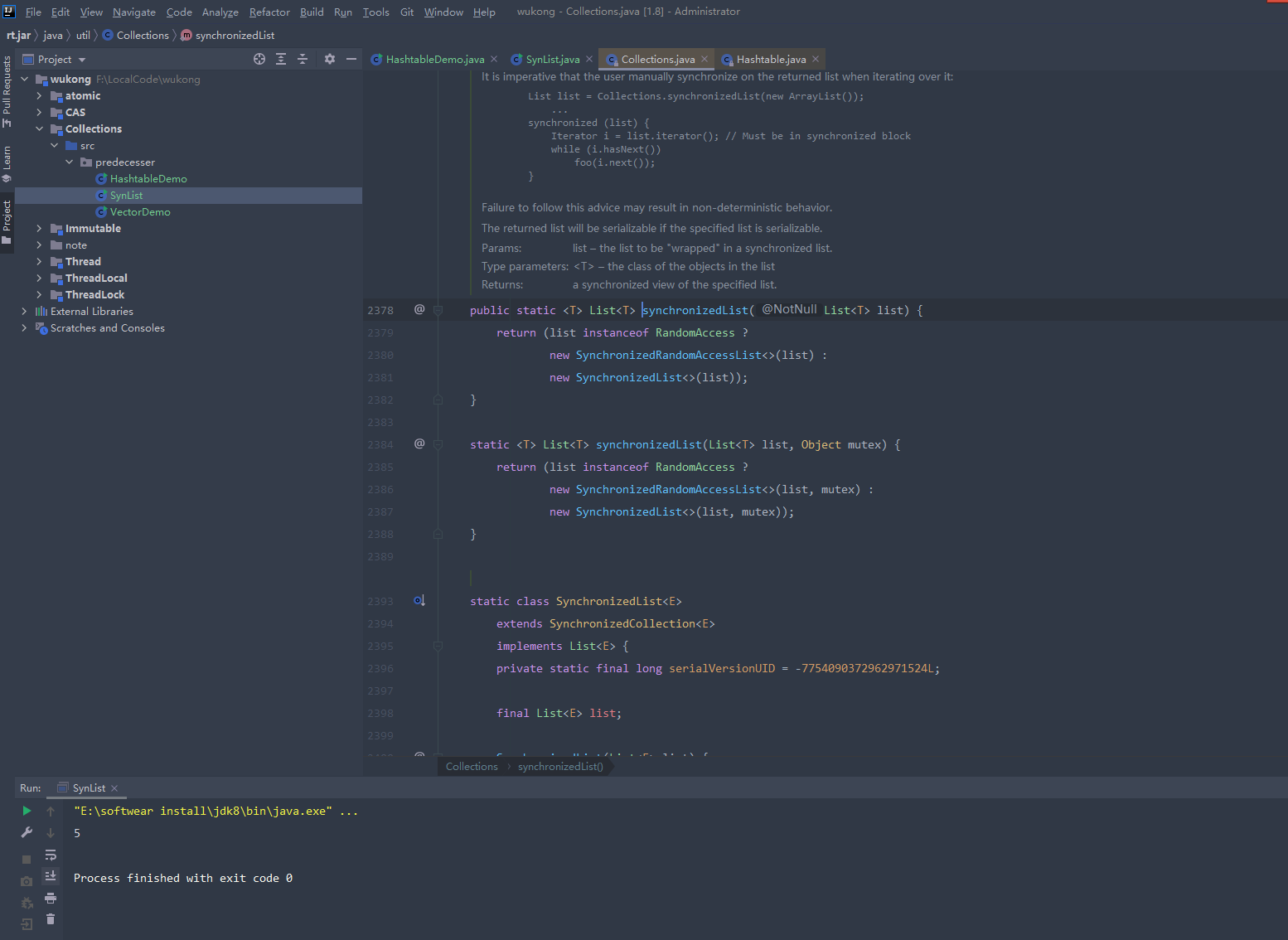
## HashMap和ArrayList(更新迭代数据容器)

我们知道HashMap和ArrayList他们是线程不安全的，但是他们有升级版

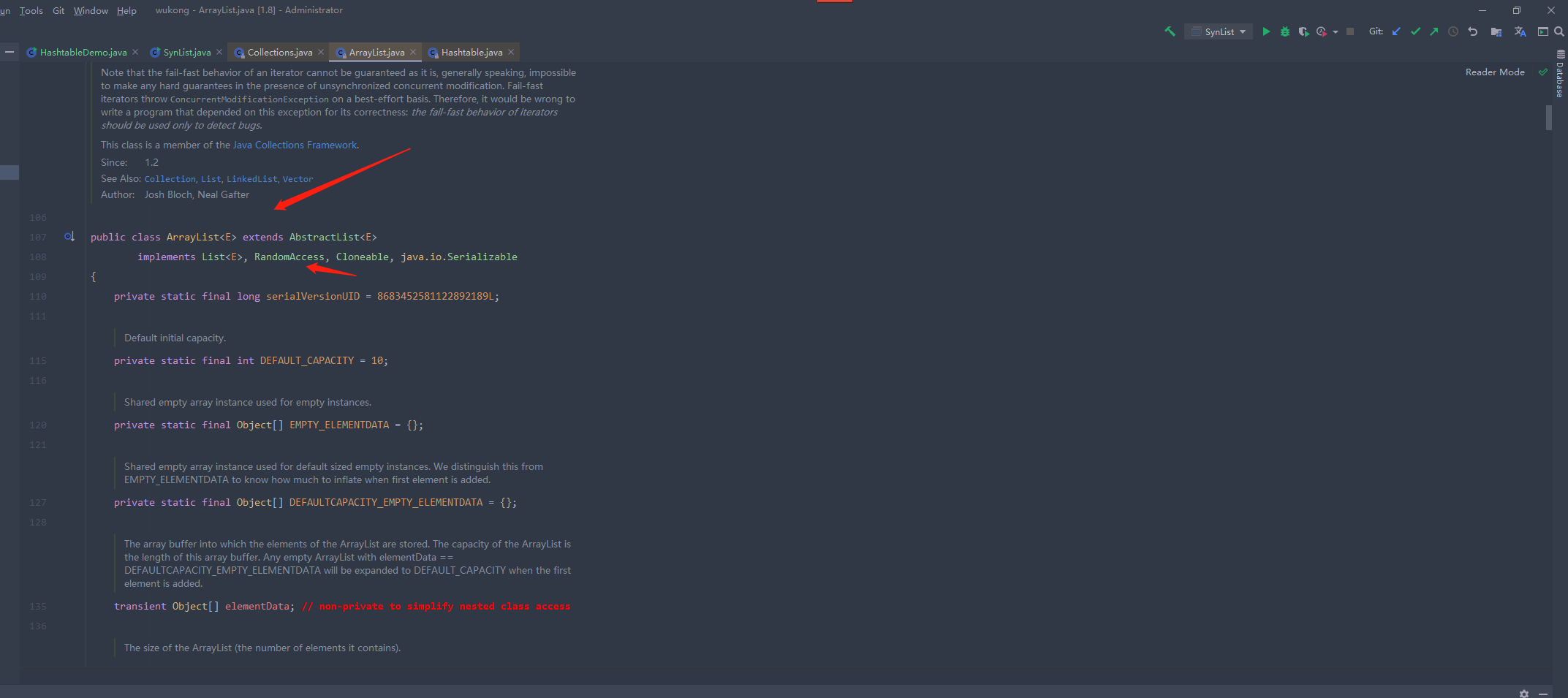
## 虽然这两个类不是线程安全的，但是可以用Collections.synchronizedList(new ArrayList<E>())和Collections.synchronizedMap(new HashMap<K,V>())使之变成线程安全的。

## 源码分析

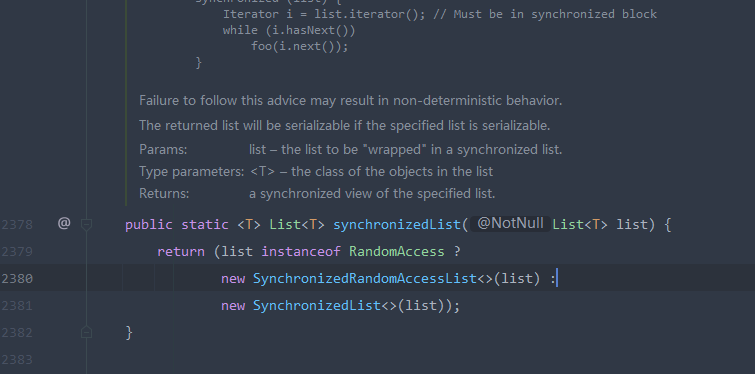
### Collections.synchronizedList(new ArrayList<E>())



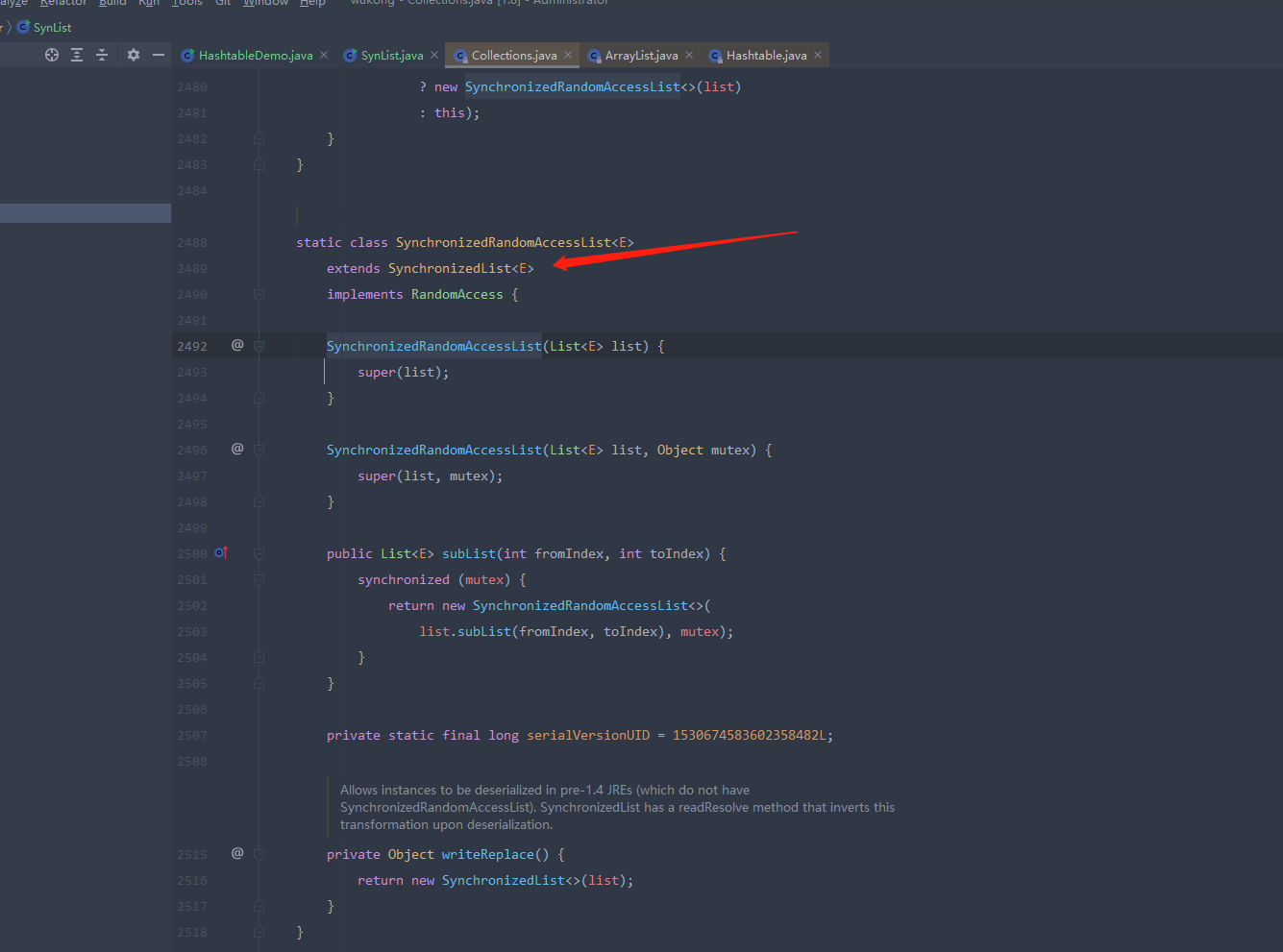
他通过判断你传入的List是不是RandomAccess来决定是不是返回new SynchronizedRandomAccessList<>(list)或者SynchronizedList<>(list)，那我们的ArrayList是符合RandomAccess的



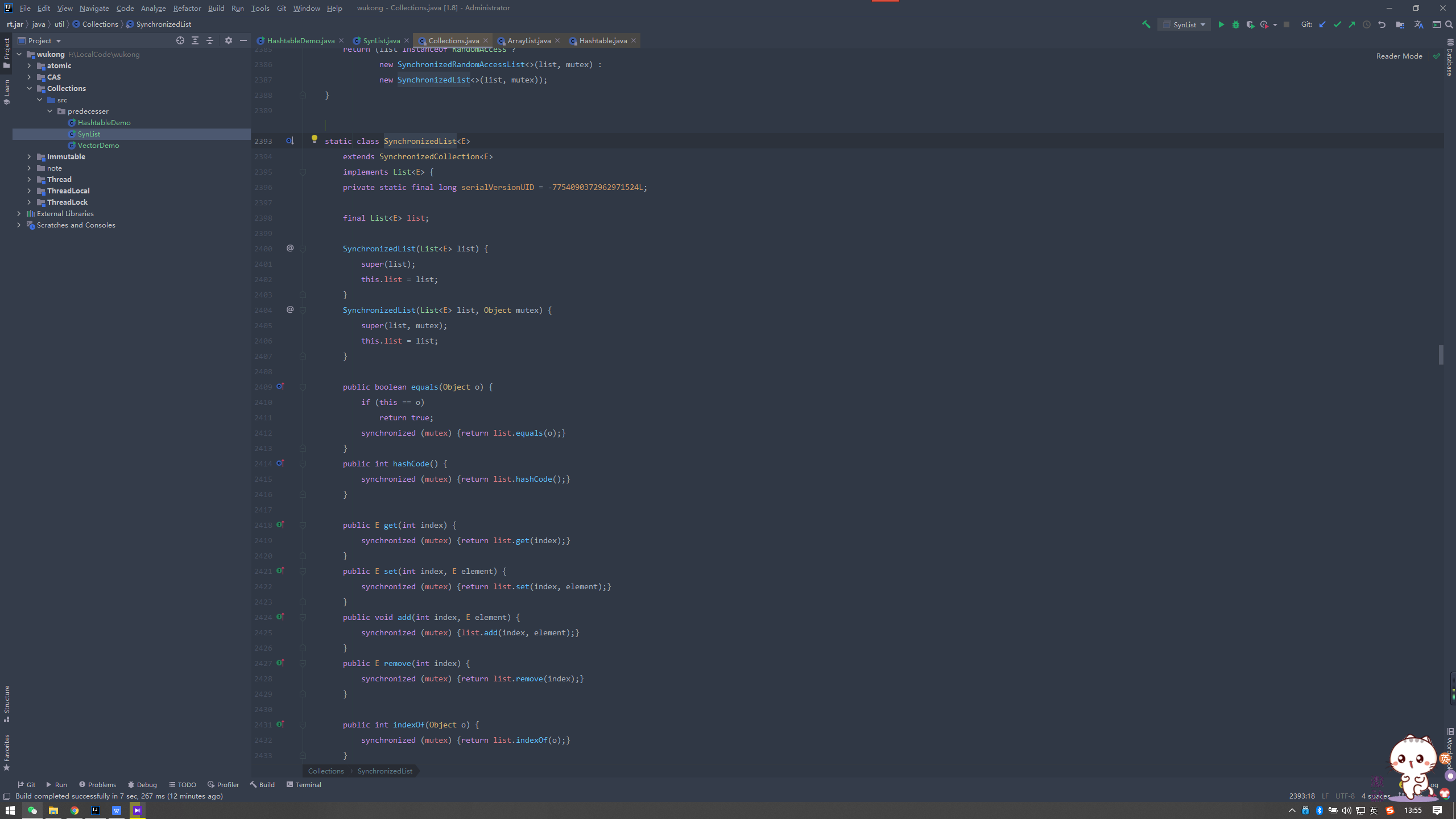
我们的ArrayList是实现了RandomAccess接口的，这个就是随机访问接口，就是说他能不能跳着访问，所以在下面我们就可以看出



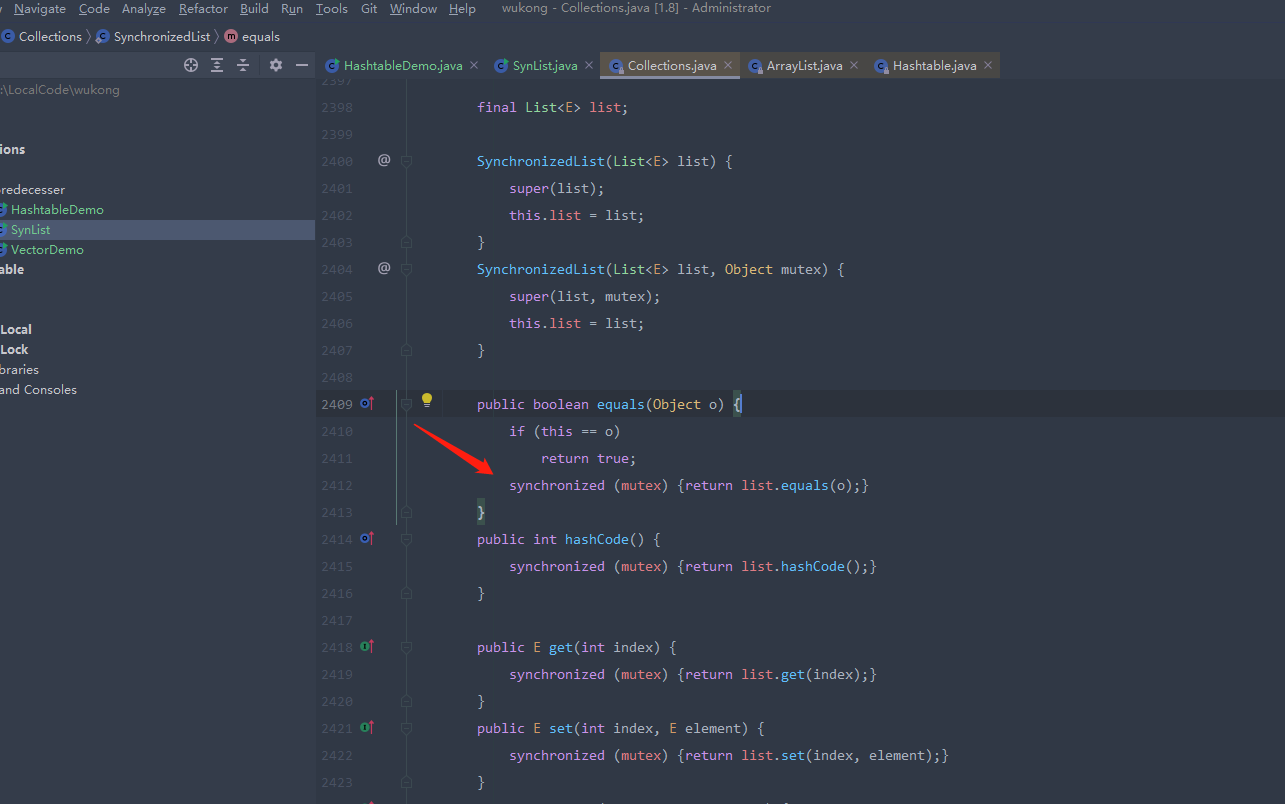
这个方法返回的就是new SynchronizedRandomAccessList<>(list)，所以我们进入到这个类



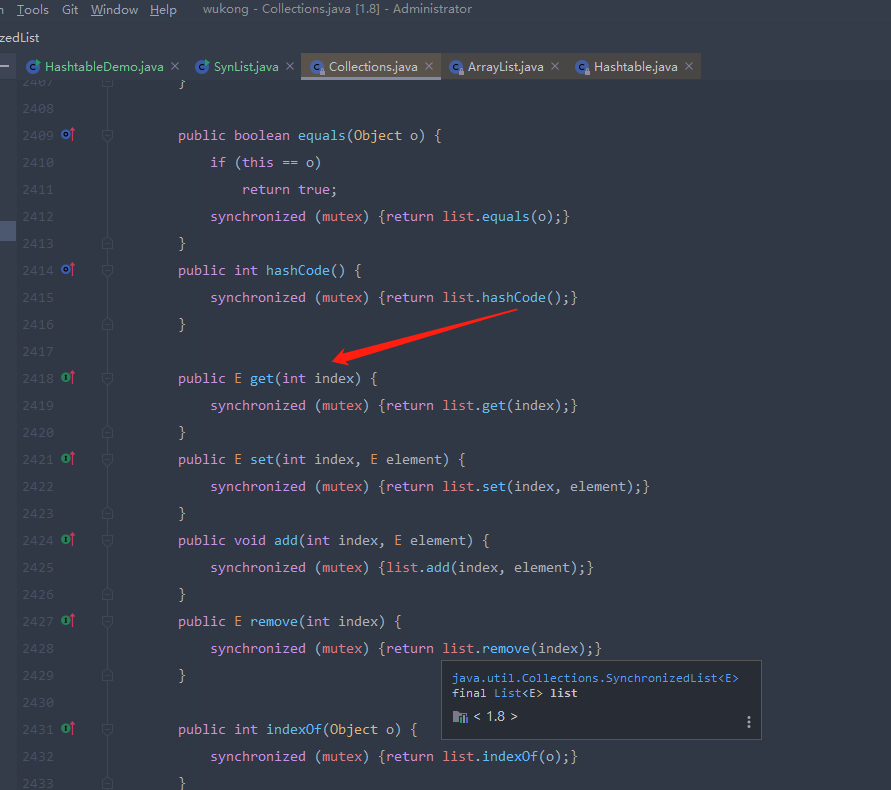
我们看到这个类又继承了SynchronizedList<E>，我们再往里面点



你就会发现，他和我们之前实现的原理是有一些不同的。它在各个方法的里面用的是同步代码块的方式



你比如说是get()



这个synchronized不是修饰在整个方法上，它是利用同步代码块里面放一个对象，同样其他的set()，add()，remove()都是用的同样的方法。并且对象用的是同一个。所以啊他一样式可以保证线程安全的，虽然它的synchronized不是加在了方法上，但是他从性能上来讲，也没有什么提高，所以呢我们使用的Collections.synchronizedList(newArrayList<E>())，这种策略啊，其实和我们上一种策略Vector和Hashtable中的线程安全实现，并没有比上一种策略高到那里去，所以相对于前面两种相对而言不是特别好的迭代之后，我们就可以进入到比较不错的实现：ConcurrentHashMap 和CopyOnWriteArrayList

## ConcurrentHashMap和CopyOnWriteArrayList

### 取代同步的HashMap和同步的ArrayList(时代巨轮滚滚向前)

### 绝大多数并发情况下，ConcurrentHashMap和CopyOnWriteArrayList的性能都更好。

唯一例外的是，如果一个List，他经常被修改，那么用Collections.synchronizedList，所包装出来的这个List，他的性能会优于我们的CopyOnWriteArrayList，这是因为我们的CopyOnWriteArrayList它适合的是读多写少的场景，每次写入呢都会完整的复制整个列表，所以比较消耗资源，但是我们的ConcurrentHashMap他几乎没有例外，他在任何场景下都比我们前面的Vector，包括同步的HashMap他的性能要更高

# 3、ConcurrentHashMap(重点、面试常考)

## 1、磨刀不误砍柴工：Map介绍

## 2、为什么需要ConcurrentHashMap

## 3、九层之台，起于累土，罗马不是一天建成的：HashMap的分析

## 4、JDK1.7的ConcurrentHashMap实现和分析

## 5、JDK1.8的ConcurrentHashMap的实现和源码分析

## 6、对比JDK1.7和1.8的优缺点，为什么要把1.7的结构改成1.8的结构？

## 7、组合操作：ConcurrentHashMap也不是线程安全的？

## 8、实际生产案例分享

# 4、CopyOnWriteArrayList

# 5、并发队列Queue(阻塞队列、非阻塞队列)

# 6、各并发容器总结