**同步容器类**

* **同步容器类**

一.为什么会出现同步容器？

　　在Java的集合容器框架中，主要有四大类别：List、Set、Queue、Map。

　　List、Set、Queue接口分别继承了Collection接口，Map本身是一个接口。注意Collection和Map是一个顶层接口，而List、Set、Queue则继承了Collection接口，分别代表数组、集合和队列这三大类容器。像ArrayList、LinkedList都是实现了List接口，HashSet实现了Set接口，而Deque（双向队列，允许在队首、队尾进行入队和出队操作）继承了Queue接口，PriorityQueue实现了Queue接口。另外LinkedList（实际上是双向链表）实现了了Deque接口。像ArrayList、LinkedList、HashMap这些容器都是非线程安全的。如果有多个线程并发地访问这些容器时，就会出现问题。

因此，在编写程序时，必须要求程序员手动地在任何访问到这些容器的地方进行同步处理，这样导致在使用这些容器的时候非常地不方便。所以，Java提供了同步容器供用户使用。

二.Java中的同步容器类

　　在Java中，同步容器主要包括2类：

　　1）Vector、Stack、HashTable

　　2）Collections类中提供的静态工厂方法创建的类

　　Vector实现了List接口，Vector实际上就是一个数组，和ArrayList类似，但是Vector中的方法都是synchronized方法，即进行了同步措施。Stack也是一个同步容器，它的方法也用synchronized进行了同步，它实际上是继承于Vector类。

　　HashTable实现了Map接口，它和HashMap很相似，但是HashTable进行了同步处理，而HashMap没有。

Collections类是一个工具提供类，注意，它和Collection不同，Collection是一个顶层的接口。在Collections类中提供了大量的方法，比如对集合或者容器进行排序、查找等操作。最重要的是，在它里面提供了几个静态工厂方法来创建同步容器类。

三.同步容器的缺陷

　　从同步容器的具体实现源码可知，同步容器中的方法采用了synchronized进行了同步，那么很显然，这必然会影响到执行性能，另外，同步容器就一定是真正地完全线程安全吗？不一定，这个在下面会讲到。

我们首先来看一下传统的非同步容器和同步容器的性能差异，我们以ArrayList和Vector为例：

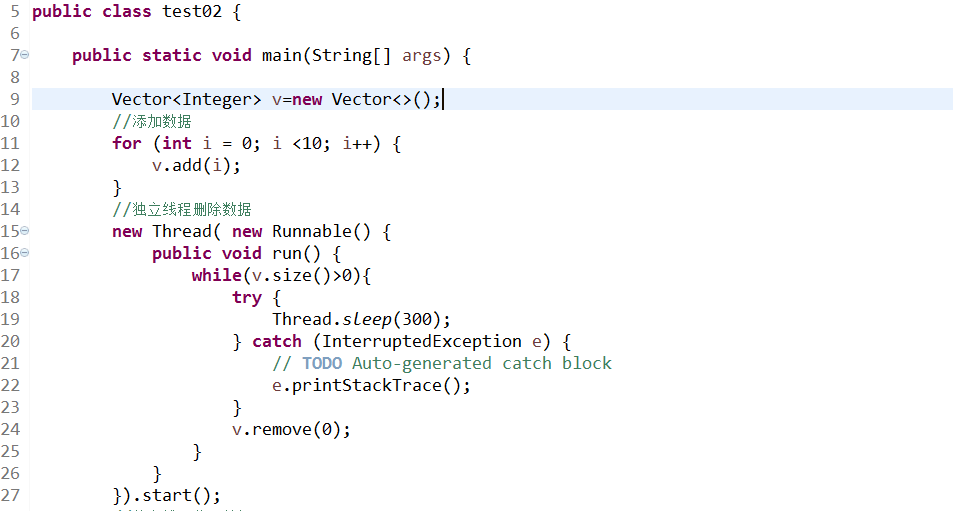
**1.性能问题**

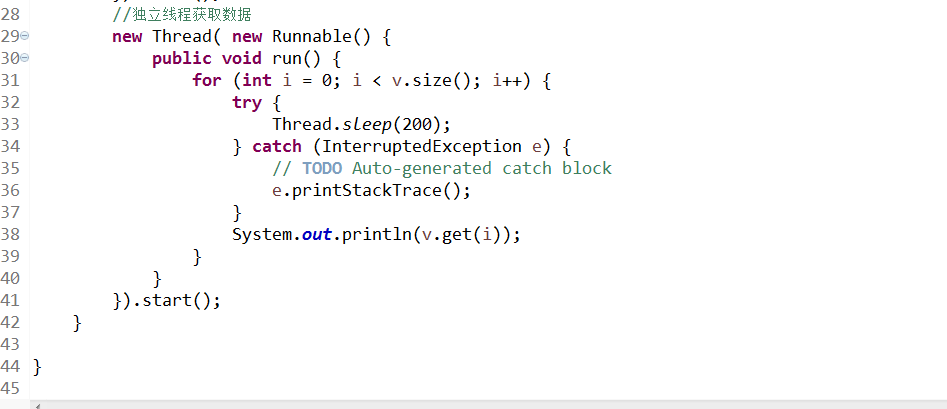
进行同样多的插入操作，Vector的耗时比ArrayList高，这只是其中的一方面性能问题上的反映，另外由于Vector中的add方法和get方法都进行了同步，因此，在有多个线程进行访问时，如果多个线程都只是进行读取操作，那么每个时刻就只能有一个线程进行读取，其他线程便只能等待，这些线程必须竞争同一把锁。

因此为了解决同步容器的性能问题，在Java 1.5中提供了并发容器，位于java.util.concurrent目录下。

**2.同步容器真的是安全的吗？**

也有有人认为Vector中的方法都进行了同步处理，那么一定就是线程安全的，事实上这可不一定。看下面这段代码：





在机器上运行的结果：这段代码报错了：数组下标越界

Vector是线程安全的，为什么还会报这个错？很简单，对于Vector，虽然能保证每一个时刻只能有一个线程访问它，但是不排除这种可能：

当某个线程在某个时刻执行这句时：



假若此时vector的size方法返回的是10，i的值为9，然后另外一个线程执行了这句：



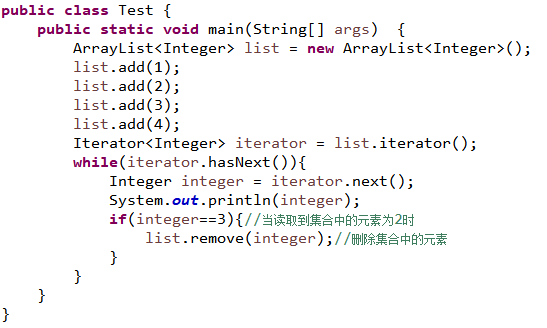
将下标为0的元素删除了。那么通过get方法访问下标为9的元素肯定就会出问题了。因此为了保证线程安全，必须在方法调用端做额外的同步措施,在删除元素的线程和获取元素的线程中加入同一把同步锁

**3. ConcurrentModificationException异常**

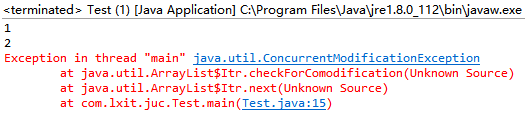
在对Vector等容器并发地进行迭代修改时，会报ConcurrentModificationException异常

1. ConcurrentModificationException异常出现的原因

测试以下代码：



程序运行结果：



从异常信息可以发现，异常出现在checkForComodification()方法中

以上程序代码在删除1，2，4中的其中一个时会出错，唯独删3不会出错，分析原因：

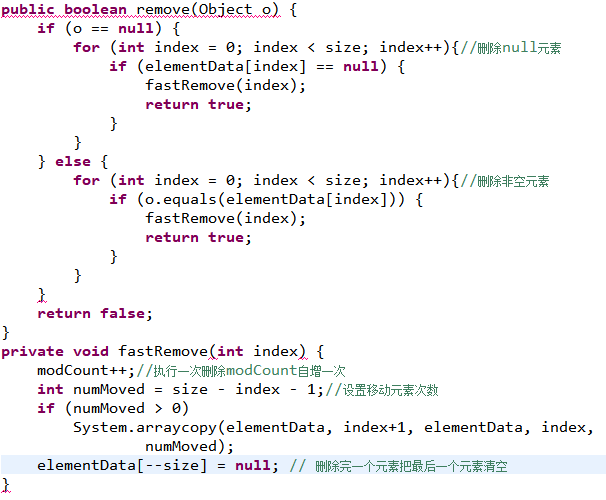
首先看ArrayList的iterator()方法的具体实现，查看源码发现在ArrayList的源码中并没有iterator()这个方法，那么很显然这个方法应该是其父类或者实现的接口中的方法，我们在其父类AbstractList中找到了iterator()方法的具体实现，下面是其实现代码：

C:\Users\ADMINI~1\AppData\Local\Temp\1547562033(1).png

从这段代码可以看出返回的是一个指向Itr类型对象的引用，我们接着看Itr的具体实现，在AbstractList类中找到了Itr类的具体实现，它是AbstractList的一个成员内部类，下面这段代码是Itr类的所有实现：



我们再看一下在ArrayList中的remove()方法做了什么：



通过list中remove方法删除元素最终是调用的fastRemove()方法，在fastRemove()方法中，首先对modCount进行加1操作（因为对集合修改了一次），然后接下来就是删除元素的操作，最后将size进行减1操作，并将引用置为null以方便垃圾收集器进行回收工作。

那么现在我回到问题代码中，在代码删除1，2，4中的其中一个时会出错，唯独删3不会出错，分析：

假如要我要删除的元素是1: 要获取元素的下标：cursor=0, 上一个元素的下标：lastRet=-1, 对ArrayList修改次数的期望值expectedModCount=4，

对List的修改次数modCount=4，因为对ArrayList进行了四次增加元素，size()=4; 开始迭代，首先调用iterator.hasNext()方法，判断cursor!=size();

0!=4,条件成立，然后通过Iterator的next()方法获取元素，在next()方法中会调用checkForComodification()方法，modCount != expectedModCount，些时modCount== expectedModCount=4，条件不成立，不会抛出异常，然后根据cursor的值获取到元素，接着将cursor的值赋给lastRet，并对cursor的值进行加1操作。初始时，cursor为0，lastRet为-1，那么调用一次之后，cursor的值为1，lastRet的值为0。注意此时，modCount为4，expectedModCount也为4。接着往下看，程序中判断当前元素的值是否为1，若为1，则调用list.remove()方法来删除该元素，通过list中的remove可知，remove方法删除元素最终是调用的fastRemove()方法，在fastRemove()方法中，首先对modCount进行加1操作（因为对集合修改了一次），然后接下来就是删除元素的操作，最后将size进行减1操作，并将引用置为null以方便垃圾收集器进行回收工作。那么注意此时各个变量的值：对于iterator，其expectedModCount为4，cursor的值为1，lastRet的值为0。对于list，其modCount为5，size为3。接着看程序代码，执行完删除操作后，继续while循环，调用hasNext方法()判断，由于此时cursor为1，而size为3，那么返回true，所以继续执行while循环，然后继续调用iterator的next()方法：注意，此时要注意next()方法中的第一句：checkForComodification()。如果modCount不等于expectedModCount，则抛出ConcurrentModificationException异常。很显然，此时modCount为5，而expectedModCount为4，因此程序就抛出了ConcurrentModificationException异常。

如果删除元素为2时同理

如果删除元素为3时，删除前道理与删除前两个一致，删除3后各值之间的变化：对于iterator，其expectedModCount为4，cursor的值为3，lastRet的值为2。对于list，其modCount为5，size为3。接着看程序代码，执行完删除操作后，继续while循环，调用hasNext方法()判断，由于此时cursor为3，而size为3，那么返回false，所以while循环结束，所以不会报错

如果删除元素为4时，删除前道理与删除前两个一致，删除4后各值之间的变化：对于iterator，其expectedModCount为4，cursor的值为4，lastRet的值为3。对于list，其modCount为5，size为3。接着看程序代码，执行完删除操作后，继续while循环，调用hasNext方法()判断，由于此时cursor为4，而size为3，那么返回true，所以继续执行while循环，然后继续调用iterator的next()方法：注意，此时要注意next()方法中的第一句：checkForComodification()。如果modCount不等于expectedModCount，则抛出ConcurrentModificationException异常。很显然，此时modCount为5，而expectedModCount为4，因此程序就抛出了ConcurrentModificationException异常。

**到这里，想必大家应该明白为何上述代码会抛出ConcurrentModificationException异常了。关键点就在于：调用list.remove()方法导致modCount和expectedModCount的值不一致。**

1. 在单线程环境下的解决办法

既然知道原因了，那么如何解决呢？其实很简单，细心的朋友可能发现在Itr类中也给出了一个remove()方法：在这个方法中，删除元素实际上调用的就是list.remove()方法，**但是它多了一个操作：expectedModCount = modCount;**因此，在迭代器中如果要删除元素的话，需要调用Itr类的remove方法。将**list.remove改成iterator.remove();**就不会报错了

1. 在多线程环境下的解决方法

上面的解决办法在单线程环境下适用，但是在多线程下适用吗？看下面一个例子：



运行结果中同样会出现会抛出ConcurrenModificationException异常

有可能有人会说ArrayList是非线程安全的容器，换成Vector就没问题了，实际上换成Vector还是会出现这种错误。原因在于，虽然Vector的方法采用了synchronized进行了同步，但是实际上通过Iterator访问的情况下，每个线程里面返回的是不同的iterator，也即是说expectedModCount是每个线程私有。假若此时有2个线程，线程1在进行遍历，线程2在进行修改，那么很有可能导致线程2修改后导致Vector中的modCount自增了，线程2的expectedModCount也自增了，但是线程1的expectedModCount没有自增，此时线程1遍历时就会出现expectedModCount不等于modCount的情况了。

因此一般有2种解决办法：

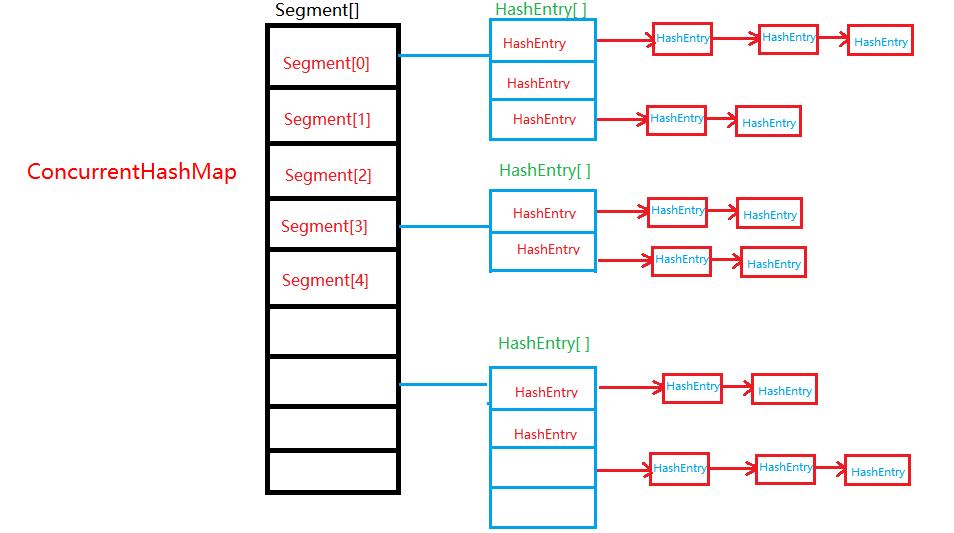
1）在使用iterator迭代的时候使用synchronized或者Lock进行同步；

2）使用并发容器CopyOnWriteArrayList代替ArrayList和Vector。

**ConcurrentHashMap的锁分段技术**

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。



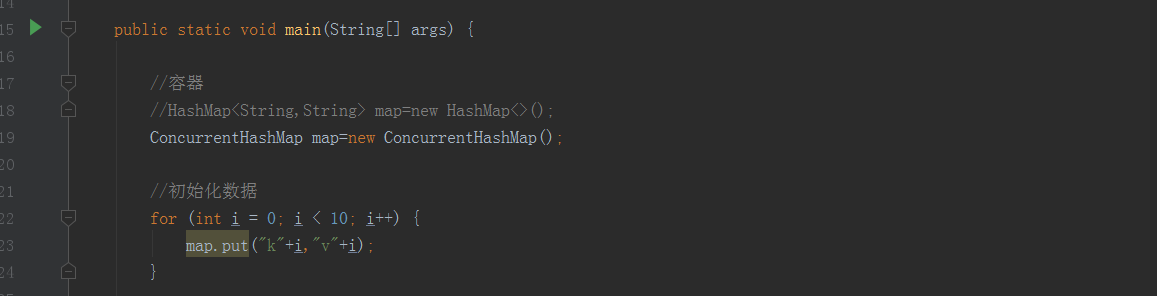
CurrentHashMap的初始化一共有三个参数，一个initialCapacity，表示初始的容量，一个loadFactor，表示负载参数，最后一个是concurrentLevel，代表ConcurrentHashMap内部的Segment的数量，ConcurrentLevel一经指定，不可改变，后续如果ConcurrentHashMap的元素数量增加导致ConrruentHashMap需要扩容，ConcurrentHashMap不会增加Segment的数量，而只会增加Segment中链表数组的容量大小，这样的好处是扩容过程不需要对整个ConcurrentHashMap做rehash，而只需要对Segment里面的元素做一次rehash就可以了。

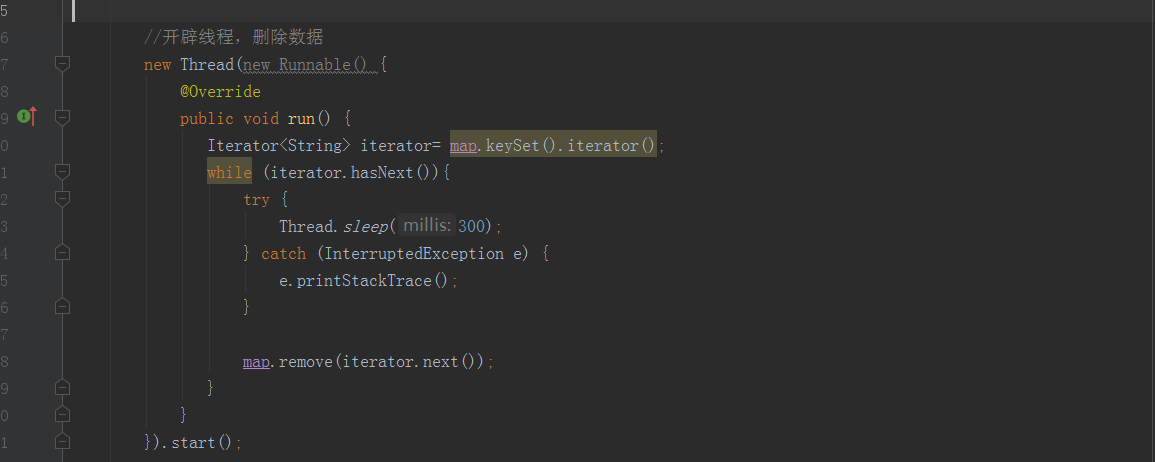
Jdk1.8将HashEntry改成了Node ,丢弃了Segment，改成CAS和synchronized了

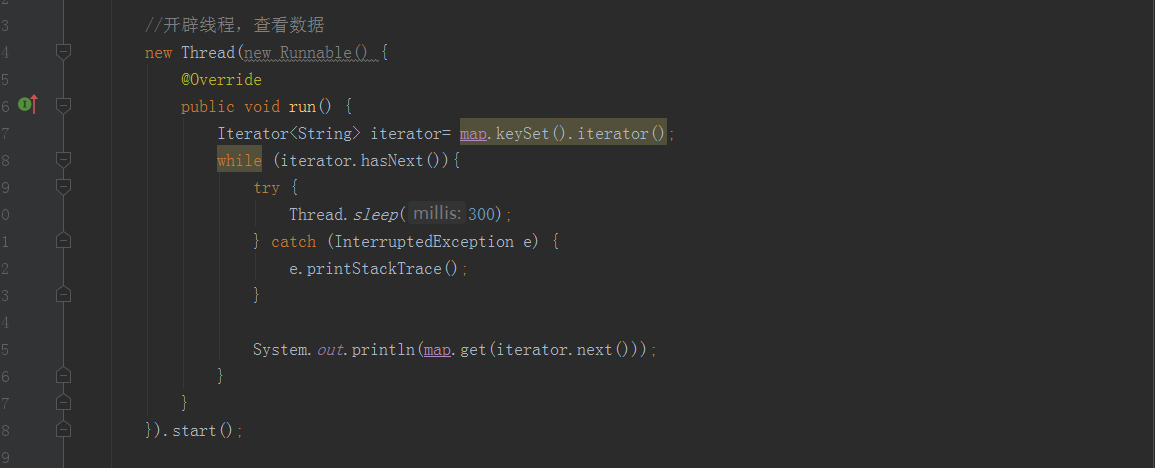
重要特点：

* public V get(Object key)不涉及到锁，也就是说获得对象时没有使用锁；
* put、remove方法要使用锁，但并不一定有锁争用，原因在于ConcurrentHashMap将缓存的变量分到多个Segment，每个Segment上有一个锁，只要多个线程访问的不是一个Segment就没有锁争用，就没有堵塞，各线程用各自的锁，ConcurrentHashMap缺省情况下生成16个Segment，也就是允许16个线程并发的更新而尽量没有锁争用

案例：







**并发容器之CopyOnWriteArrayList**

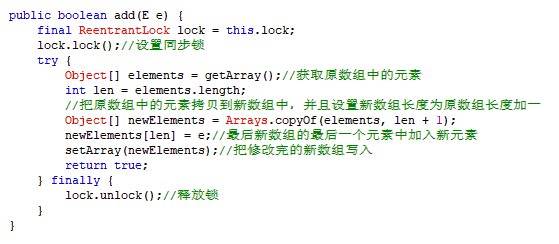
Copy-On-Write简称COW，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容Copy出去形成一个新的内容然后再改，这是一种延时懒惰策略。从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器,它们是CopyOnWriteArrayList和CopyOnWriteArraySet。CopyOnWrite容器非常有用，可以在非常多的并发场景中使用到。

**什么是CopyOnWrite容器**

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

**CopyOnWriteArrayList的实现原理**

在使用CopyOnWriteArrayList之前，我们先阅读其源码了解下它是如何实现的。以下代码是向CopyOnWriteArrayList中add方法的实现（向CopyOnWriteArrayList里添加元素），可以发现在添加的时候是需要加锁的，否则多线程写的时候会Copy出N个副本出来。



重要特点：

* 实现了List接口
* 内部持有一个ReentrantLock lock = new ReentrantLock();
* 底层是用volatile transient声明的数组 array
* 读写分离，写时复制出一个新的数组，完成插入、修改或者移除操作后将新数组赋值给array
* 读的时候不需要加锁，如果读的时候有多个线程正在向CopyOnWriteArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的CopyOnWriteArrayList。

案例：

