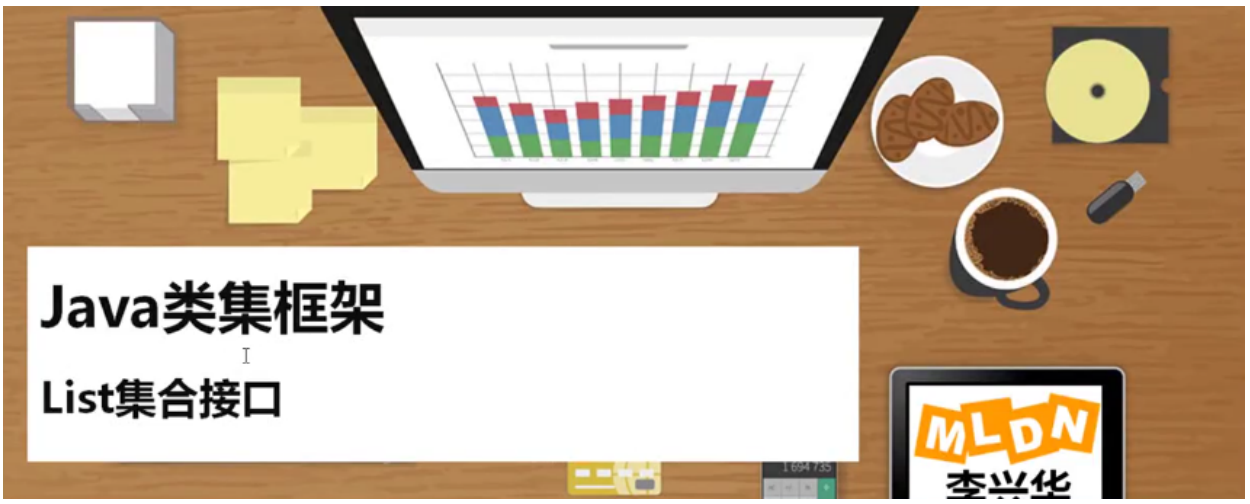


博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

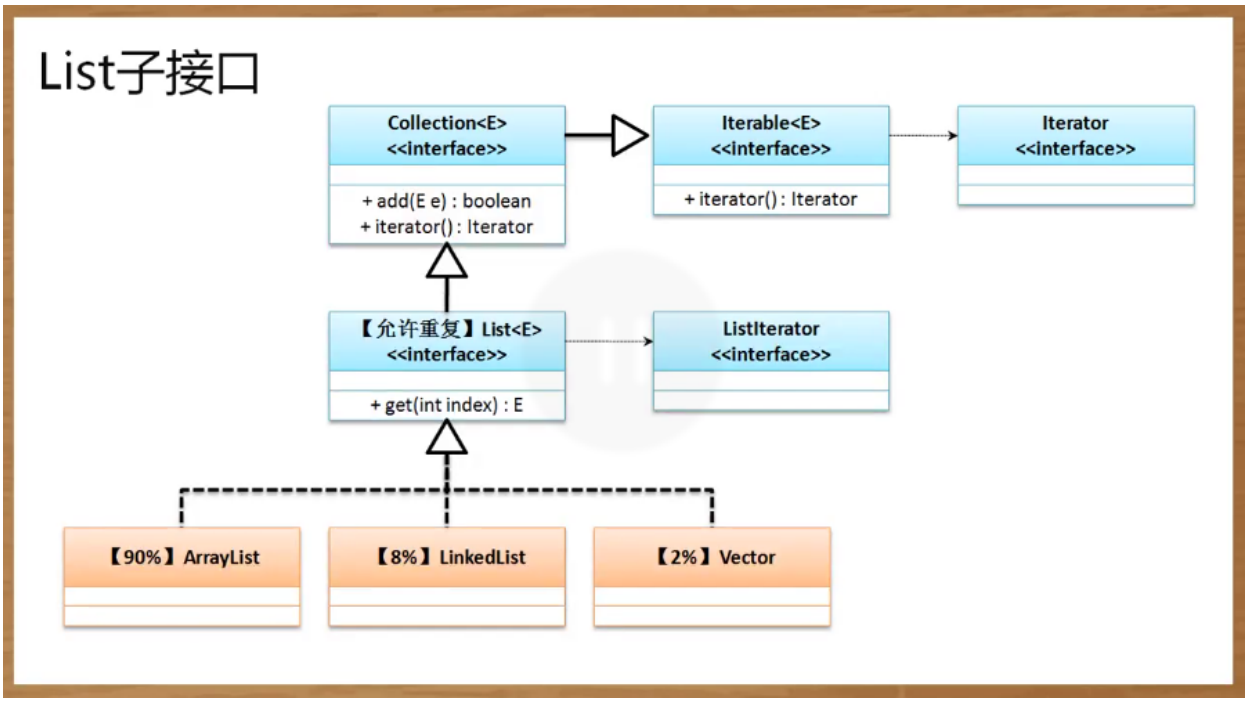
List是Collection子接口，其最大的特点是允许有重复元素数据，该接口的定义如下：

```
public interface List<E> extends Collection<E>
```

但是需要清楚的是List子接口对于Collection接口进行了方法扩充

No	方法名称	类型	描述
01	**public E get(int index)**	**普通**	**获取指定索引上的数据**
02	public E set(int index,E element)	普通	修改指定索引数据
03	public ListIterator<E> listIterator()	普通	返回ListIterator接口对象

但是List本身依然属于一个接口，那么对于接口要想使用则一定要使用子类来完成定义，在List子接口中有三个常用子类：ArrayList、Vector、LinkedList。



从JDK1.9开始List子接口里面追加有一些static方法，以方便用户的处理。

范例：观察List中的静态方法

```
package cn.mldn.demo;
import java.util.List;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		List<String> all = List.of("Hello", "World", "你好", "MLDN", "饿了么？");
		System.out.println(all);
	}
}
```

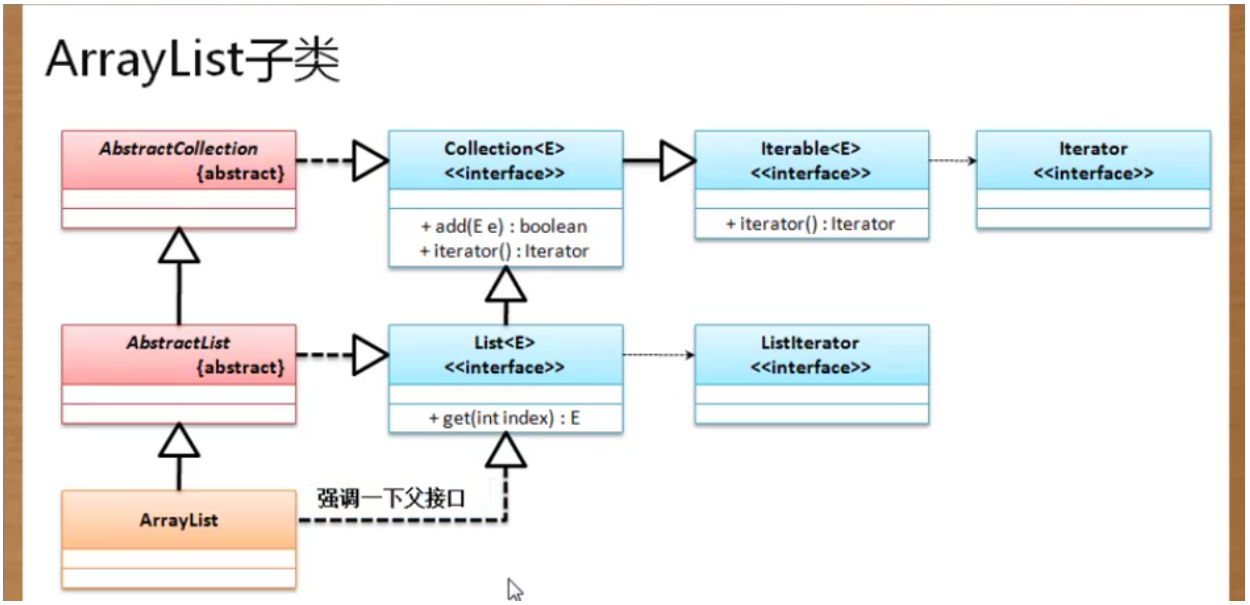
这些操作方法并不是List传统用法，是在新版本之后添加的新功能。

■ArrayList子类

ArrayList是List子接口使用最多的一个子类，但是这个子类在使用的时候也是有前提要求的，所以本次对这个类的相关定义以及源代码组成进行分析在Java里面ArrayList类的定义如下：

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList子类的继承结构如下所示：



范例：使用ArrayList实例化List父接口

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		List<String> all = new ArrayList<String>() ; // 为List父接口进行实例化
		all.add("Hello") ;
		all.add("Hello") ;	// 重复数据
		all.add("World") ;
		all.add("MLDN") ;
		System.out.println(all);
	}
}
```

通过本程序可以发现List存储的特征：

·保存的顺序就是其存储顺序；

·List集合里面允许存在有重复数据；

在以上的程序里面虽然实现了集合的输出，但是这种输出的操作是直接利用了每一个类提供的toString()方法实现的，为了方便的进行输出处理，在JDK1.8之后Iterable父接口之中定义有一个forEach()方法，方法定义如下：

·输出支持：default void forEach(Consumer<? super T> action);

范例：使用forEach()方法输出(不是标准输出)

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		List<String> all = new ArrayList<String>() ; // 为List父接口进行实例化
		all.add("Hello") ;
		all.add("Hello") ;	// 重复数据
		all.add("World") ;
```

```
            all.add("MLDN") ;
            all.forEach((str)->{
                System.out.print(str + "、 ");
            });
        }
}
```

需要注意的是，此种输出并不是在正常开发情况下要考虑的操作形式。

范例：观察List集合的其它操作方法

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
            List<String> all = new ArrayList<String>() ; // 为List父接口进行实例化
            System.out.println("集合是否为空？" + all.isEmpty() + "、集合元素个数：" +
all.size());
            all.add("Hello") ;
            all.add("Hello") ;     // 重复数据
            all.add("World") ;
            all.add("MLDN") ;
            all.remove("Hello") ; // 删除元素
            System.out.println("集合是否为空？" + all.isEmpty() + "、集合元素个数：" +
all.size());
            all.forEach((str)->{
                System.out.print(str + "、 ");
            });
        }
}
```

如果以方法的功能为例，那么ArrayList里面操作支持与之前编写的链表形式是非常相似的，但是它并不是使用链表来实现的，通过类名称实际上就已经可以清楚的发现了，ArrayList应该封装的是一个数组。

ArrayList构造：
public ArrayList()
	public ArrayList() {
	  this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
	}

	private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

ArrayList构造：
public ArrayList(int
initialCapacity)
	public ArrayList(int initialCapacity) {
	  if (initialCapacity > 0) {
	  this.elementData = new Object[initialCapacity];
	  } else if (initialCapacity == 0) {
	  this.elementData = EMPTY_ELEMENTDATA;
	  } else {
	  throw new IllegalArgumentException("Illegal Capacity: "+
	  initialCapacity);
	  }
	}

transient Object[] elementData;	

通过有参的构造方法可以发现，在ArrayList里面所包含的数据实际上就是一个对象数组。如果现在在进行数据追加的时候发现ArrayList集合里面保存的对象数组的长度不够的时候那么会进行新的数组开辟，同时将原始的旧数组内容拷贝到新数组之中，而后数组的开辟操作：

```
private int newCpacity(int minCapacity){
//overflow - conscious code
int oldCapacity = elementData.length;
int newCapacity = oldCapacity + (oldCapacity >> 1);
if(newCapacity - mincapacity < = 0){
        if(elementData = = DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
                return Math.max(DEFAULT_CAPACITY,minCapacity);
        if(minCapacity<0)   //overflow
                throw new OutOfMemoryError();
        return minCapacity;
}
return(newCapacity - MAX_ARRAY_SIZE < = 0) ? newcapacity : hugecapacity(mincapacity);
}
```

如果在实例化ArrayList类对象的时候并没有传递初始化的长度，则默认情况下会使用一个空数组，但是如果在进行数据增加的时候发现数组容量不够了。则会判断当前的增长的容量与默认的容量的大小，使用较大的一个数值进行新的数组开辟，所以可以得出一个结论：

JDK1.9之后	ArrayList默认的构造只会使用默认的空数组，使用的时候才会开辟数组，默认开辟的长度为10
JDK1.9之前	ArrayList默认的构造只会使用默认开辟大小为10的数组

当ArrayList之中保存的容量不足的时候会采用成倍的方式进行增长，原始长度为10，那么下次的增长就是20，以此类推。在使用ArrayList子类的时候一定要估算你的数据量会有多少，如果超过了10个，那么使用有参构造方法进行创建，以避免垃圾数组的空间产生。

■ArrayList保存自定义类

通过之前的分析已经清楚了ArrayList子类的实现原理以及List核心操作，但是在测试的时候使用的是系统提供的String类，这是一个设计非常完善的类，而对于类集而言也可以实现自定义类对象的保存。

范例：实现自定义类对象的保存

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
class Person {
        private String name ;
        private int age ;
```

```
	public Person(String name,int age) {
		this.name = name ;
		this.age = age ;
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj) {
			return true ;
		}
		if (obj == null) {
			return false ;
		}
		if (!(obj instanceof Person)) {
			return false ;
		}
		Person per = (Person) obj ;
		return this.name.equals(per.name) && this.age == per.age ;
	}
	// setter、getter、构造略
	public String toString() {
		return "姓名：" + this.name + "、年龄：" + this.age ;
	}
}
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		List<Person> all = new ArrayList<Person>();
		all.add(new Person("张三",30)) ;
		all.add(new Person("李四",16)) ;
		all.add(new Person("小强",78)) ;
		System.out.println(all.contains(new Person("小强",78)));
		all.remove(new Person("小强",78)) ;
		all.forEach(System.out::println);	// 方法引用代替了消费型的接口
	}
}
```

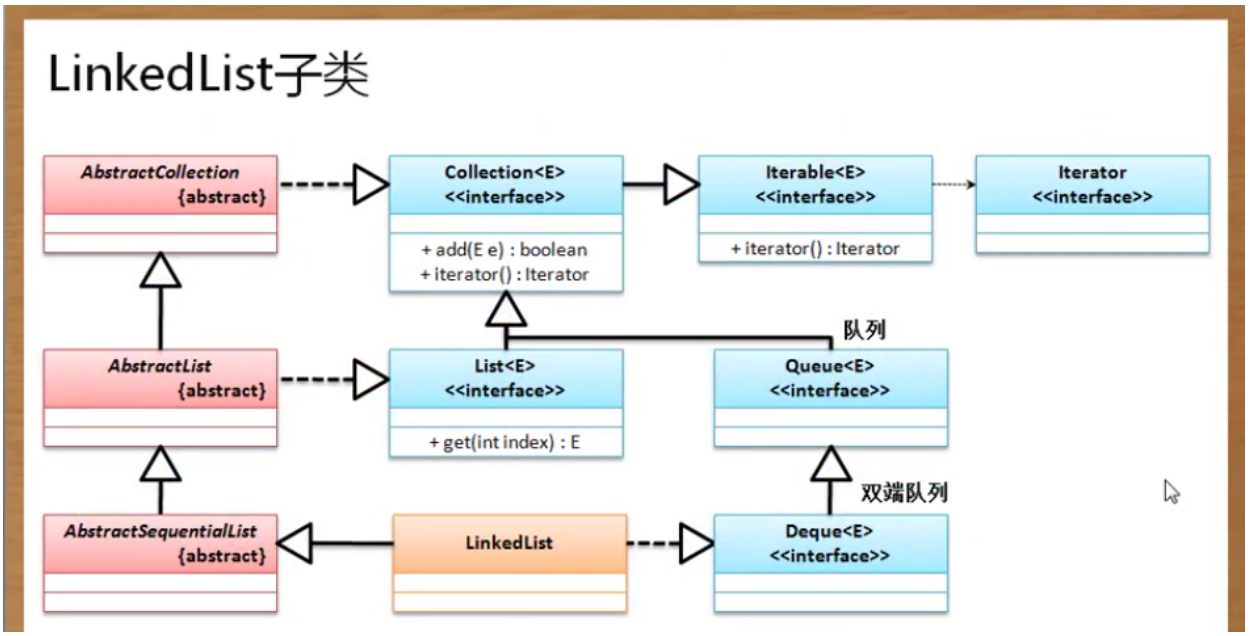
在使用List保存自定义类对象的时候如果需要使用到contains()、remove()方法进行查询与删除处理的时候一定要保证类之中已经成功覆写了equals()方法.

■LinkedList子类

在List接口里面还有另外一个比较常用的子类：LinkedList，这个类通过名称就已经可以发现其特点了：基于链表的实现，那么首先来观察一下LinkedList的定义：

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

LinkedList子类的继承关系如下：



范例：使用LinkedList实现集合操作

```
package cn.mldn.demo;
import java.util.LinkedList;
import java.util.List;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		List<String> all = new LinkedList<String>(); // 为List父接口进行实例化
		all.add("Hello");
		all.add("Hello"); // 重复数据
		all.add("World");
		all.add("MLDN");
		all.forEach(System.out::println);
	}
}
```

如果说现在只是观察程序的功能会发现和ArrayList使用是完全一样的，但是其内部的实现机制是完全不同的，首先观察LinkedList构造方法里面并没有提供有像ArrayList那样的初始化大小的方法，而只是提供有无参构造处理方法：“public LinkedList()”。随后观察add()方法的具体实现。

public boolean add(E e){
　　linkLast(e);
　　return true;
}
	在之前编写自定义链表的时候，是判断了传入数据是否为NULL，如果为NULL则不进行保存，但是在LinkedList里面并没有做这样的处理，而是所有的数据都可以保存，而后此方法调用了LinkLast()方法（在最后一个结点之后追加）

void linkLast(E e) {
　　final Node<E> l = last;
　　final Node<E> newNode = new Node<>(l, e, null);
　　last = newNode;
　　if (l == null)
　　　first = newNode;
　　else
	在LinkedList类里面保存的数据都是利用Node结点进行的封装处理，同时为了提供程序执行性能，每一次都会保存上一个追加的节点（最后一个节点），就可以在增加数据的时候避免递归处理，在增加数据的时候要进行数据保存个数的追加。

```
            l.next = newNode;
        size++;
        modCount++;
    }
```

通过一系列的分析之后就可以出现，LinkedList封装的就是一个链表实现。

面试题：请问ArrayList与LinkedList有什么区别?

·ArrayList是数组实现的集合操作，而LinkedList是链表实现的集合操作；

·在使用List集合中的get() 方法根据索引获取数据时，ArrayList的时间复杂度为“O(1)”，而LinkedList时间福再度为“O(N)”（N为集合的长度）；

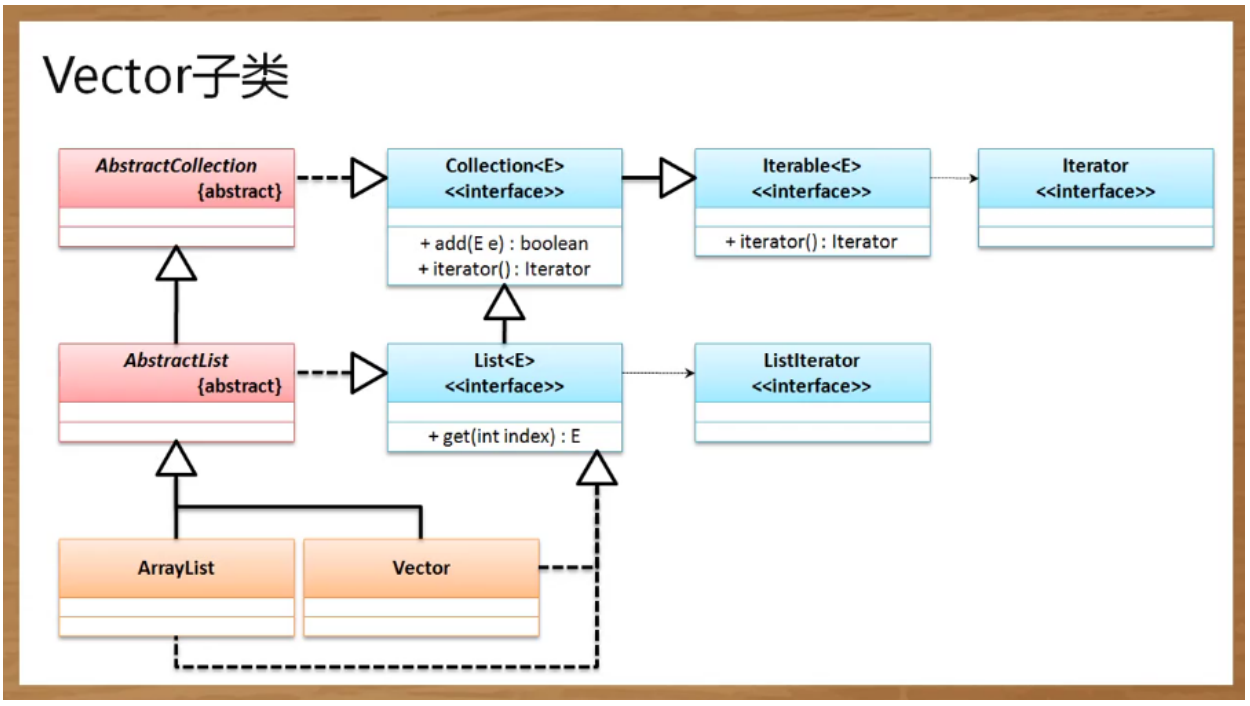
·ArrayList在使用的时候默认的初始化对象数组的大小长度为10，如果空间不足则会采用2倍的形式进行容量的扩充，如果保存大数据量的时候有可能会造成垃圾的产生以及性能的下降，但是这个时候可以使用LinkedList子类保存

■Vector子类

Vector是一个原始古老的程序类，这个类是在JDK1.0的时候就提供的，而后到了JDK1.2的时候由于许多的开发者已经习惯于使用Vector，并且许多的系统类也是基于Vector实现的，考虑到其使用的广泛性，所以类集框架将其保存了下来，并且让其多实现了一个List接口，观察Vector定义结构：

```
public class Vector<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

继承结构与ArrayList是相同的，所以来讲这个类继承结构如下：



范例：Vector类使用

```
package cn.mldn.demo;
```

```
import java.util.List;
import java.util.Vector;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		List<String> all = new Vector<String>(); // 为List父接口进行实例化
		all.add("Hello");
		all.add("Hello"); // 重复数据
		all.add("World");
		all.add("MLDN");
		all.forEach(System.out::println);
	}
}
```

下面可以进一步的观察Vector类实现：

```
public Vector() {
	this(10);
}
```

```
public Vector(int initialCapacity) {
	this(initialCapacity, 0);
}
```

```
public Vector(int initialCapacity, int capacityIncrement) {
	super();
	if (initialCapacity < 0)
		throw new IllegalArgumentException("Illegal Capacity: "+
						initialCapacity);
	this.elementData = new Object[initialCapacity];
	this.capacityIncrement = capacityIncrement;
}
```

Vector类如果使用的是无参构造方法，则一定会默认开辟一个10个长度的速度，而后其余的实现操作与ArrayList是相同的。通过源代码的分析可以发现Vector类之中的操作方法采用的都是synchronized同步处理，而ArrayList并没有进行同步处理，所以Vector类之中的方法在多线程访问的时候属于线程安全的，但是性能不如ArrayList高。