愚公要移山
西北工业大学 计算机技术硕士在读

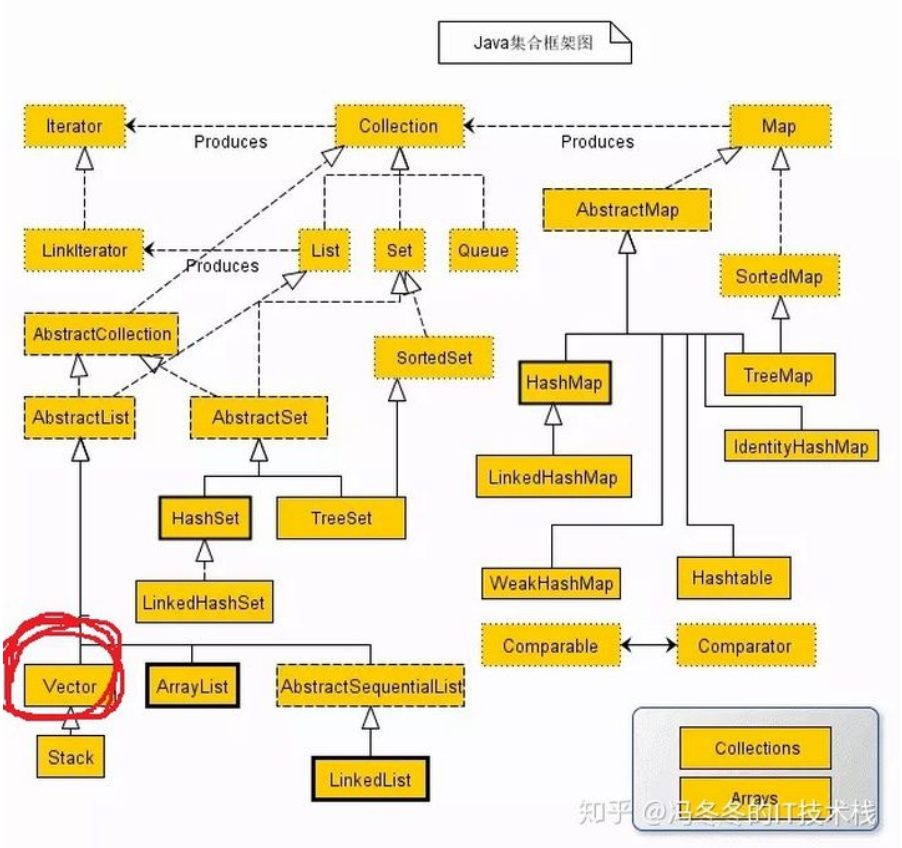
这篇文章开始介绍Vector。他和ArrayList有一些相似,其内部都是通过一个容量能够动态增长的数组来实现的。不同点是Vector是线程安全的。因为其内部有很多同步代码快来保证线程安全。为此，这篇文章，也会通过从源码的角度来分析一下Vector，并和ArrayList等其他集合容器进行一个对比分析。

OK，开始今天的文章。

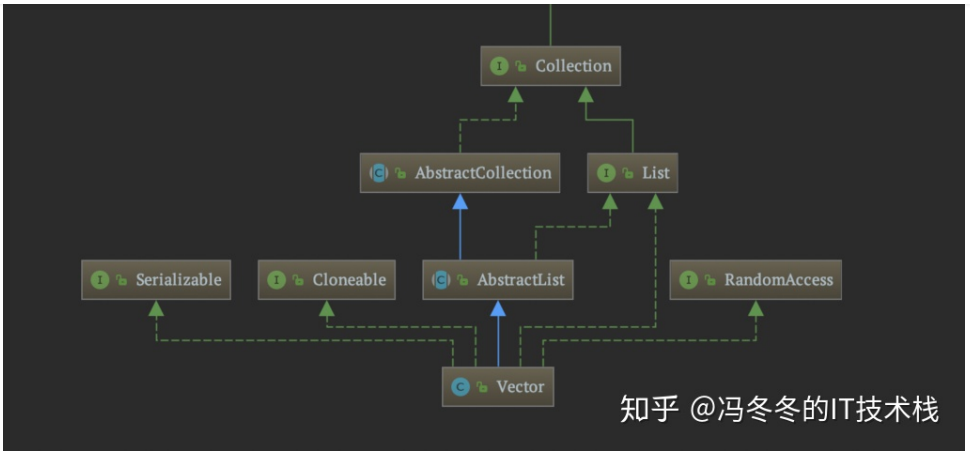
一、认识Vector

Vector可以实现可增长的对象数组。与数组一样，它包含可以使用整数索引进行访问的组件。不过，Vector的大小是可以增加或者减小的，以便适应创建Vector后进行添加或者删除操作。

为此我们先看一下Vector在整个java集合体系中的位置



从上面这张图我们也会发现Vector和ArrayList是出于一个等级上面的，继承关系也和ArrayList一样。不过从宏观上只能看到在整个体系中的位置，现在我们从Vector来看看他的继承关系。



现在我们就根据这张图来进行一个分析，Vector继承于AbstractList，实现了List、RandomAccess、Cloneable、 Serializable等接口。

（1）Vector 继承了AbstractList，实现了List接口。

（2）Vector实现了RandmoAccess接口，即提供了随机访问功能。

（3）Vector 实现了Cloneable接口，即实现克隆功能。

（4）Vector 实现Serializable接口，表示支持序列化。

Vector实现的这些接口，表示会有这样的能力。但是还有一点，就是Vector是线程安全的。下面我们从源码的角度来分析一下Vector是如何实现这些接口和保持线程安全的特性的。

二、源码分析Vector

（1）构造方法

Vector的构造方法一共有四个，因为四个都比较重要，所以在这里就给出四个

第一个： 创建一个空的Vector，并且指定了Vector的初始容量为10

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

第二个：创建一个空的Vector，并且指定了Vector的初始容量

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

成员属性;

1、vector存储对象的数组容器elementData
protected Object[] elementData;

2、elementData数组中存储的实际元素个数
protected int elementCount;

3、扩容规则：扩容时的增长系数，如果capacityIncrement大于0则扩容时候容量=旧数组大小+capacityIncrement，若capacityIncrement<=0则，扩容量=旧数组容量的两倍

protected int capacityIncrement;

第三个：创建一个空的Vector，并且指定了Vector的初始容量和扩容时的增长系数

```
//initialCapacity：初始容量
//capacityIncrement：扩容时的增长系数
public Vector(int initialCapacity, int capacityIncrement) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
    //这句话表明，其底层就是通过数组来建立的
    this.elementData = new Object[initialCapacity];
    this.capacityIncrement = capacityIncrement;
}
```

```
public Vector(Collection<? extends E> c) {
    elementData = c.toArray();
    elementCount = elementData.length;
    if (elementData.getClass() != Object[].class)
        elementData = Arrays.copyOf(elementData, elementCount, Object[].class);
}
```

第四个需要解释一下，首先是把其他集合转化为数组，然后复制粘贴到Vector里面。

（2）增加元素

增加元素有两个主要的方法，第一个是在Vector尾部追加，第二个是在指定位置插入元素。

第一个：在Vector尾部追加元素

```
public synchronized boolean add(E e) {
    modCount++;
    //判断容量大小：若能装下就直接放进来，装不下那就扩容
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}
```

我们再进来看一下ensureCapacityHelper(elementCount + 1)是如何实现的。

```
private void ensureCapacityHelper(int minCapacity) {
    //追加一个元素后，当前的容量是minCapacity
    // 若minCapacity > elementData原始的容量，则要按照minCapacity进行扩容
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

现在相当于真正扩容的方法是grow方法，别着急我们再进来看看。

扩容规则：原始数组大小+capacityIncrement扩容量

```
private void grow(int minCapacity) {
    //第一步：获取elementData的原始容量
    int oldCapacity = elementData.length;
    //第二步：apacityIncrement：表示需要新增加的数量，如果大于0，那就扩充这么多，如果不大于0，
    int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                     capacityIncrement : oldCapacity);
    //第三步：若进行扩容后，capacity仍然小，则新容量改为实际需要minCapacity的大小
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    //第四步：如果新数组的长度比虚拟机能够提供给数组的最大存储空间大，
    // 则将新数组长度更改为最大正数：Integer.MAX_VALUE
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    //第五步：按照新的容量newCapacity创建一个新数组，然后再将原数组中的内容copy到新数组中
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

在这一步我们扩容的时候首先就要排除一些异常的情况，首先就是capacityIncrement（需要增加的数量）是否大于0，如果大于0直接增加这么多。然后发现增加了上面那些还不够那就扩充为实际需要minCapacity的大小。最后发现还不够，就只能扩充到虚拟机能表示的数字最大值了。

第二个：在指定位置增加元素

这个就比较简单了。我们直接看源码就能看明白

```
}
public synchronized void insertElementAt(E obj, int index) {
    //第一步：fail-fast机制
    modCount++;
    //第二步：判断index下标的合法性
    if (index > elementCount) {
        throw new ArrayIndexOutOfBoundsException(index+ " > " + elementCount);
    }
    //第三步：判断容量大小
    ensureCapacityHelper(elementCount + 1);
    //第四步：数组拷贝，将index到末尾的元素拷贝到index + 1到末尾的位置，将index的位置留出来
    System.arraycopy(elementData, index, elementData, index + 1, elementCount - index)
    elementData[index] = obj;
    elementCount++;
}
```

（3）删除元素

删除元素时候同样也有两种方法，第一个根据元素值来删除，第二个根据下表来删除元素。

第一个：根据元素值来删除元素

```
public boolean remove(Object o) {
    return removeElement(o);
}
```

我们发现删除元素其实是调用了removeElement()方法来删除元素的，没关系不要嫌麻烦，进入这个方法内部看一下。

```
public synchronized boolean removeElement(Object obj) {
    //第一步：fail-fast机制
    modCount++;
    //第二步：查找元素obj在数组中的下标
    int i = indexOf(obj);
    //第三步:若下标不小于0：说明Vector容器中含有这个元素
    if (i >= 0) {
        //第四步：调用removeElementAt(int)方法删除元素
        removeElementAt(i);
        return true;
    }
    return false;
}
```

到了这一步，我们又发现，执行删除操作的还不是removeElement()方法，而是removeElementAt(i)，我们再进入这个方法看看。

```
public synchronized void removeElementAt(int index) {
    //第一步：fail-fast机制
    modCount++;
    //第二步：index下标合法性检验
    if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " +elementCount);
    }
    else if (index < 0) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    //第三步：要移动的元素个数
    int j = elementCount - index - 1;
    if (j > 0) {
```

```
    }
    elementCount--;
    elementData[elementCount] = null;
}
```

到了这个方法我们其实可以分析一下，要删除元素要移动大量的元素，时间效率肯定是不好的。毕竟Vector是通过数组来实现的，而不是通过链表。

第二个：删除指定位置的元素

删除指定位置的元素就比较简单了，我们到指定的位置进行删除就好了，但是同样需要把后面的元素进行移位。

```
public synchronized E remove(int index) {
    //第一步：fail-fast机制
    modCount++;
    //第二步：index下标合法性检验
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    //获取旧的元素值
    E oldValue = elementData(index);
    //第三步：计算需要移动的元素个数
    int numMoved = elementCount - index - 1;
    //第四步：将元素向前移动
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index, numMoved);
    elementData[--elementCount] = null;
    return oldValue;
}
```

（3）更改元素

更改元素我们就先看一个吧。这个在大部分场景下一般不用（大部分，根据自己业务来定）。

```
public synchronized void setElementAt(E obj, int index) {
        if (index >= elementCount) {
            throw new ArrayIndexOutOfBoundsException(index + " >= " +  elementCount);
        }
        elementData[index] = obj;
    }
```

（4）查找元素

查找元素我们给出三个，第一个查询Vector容器中是否包含某个元素，第二个查询第一次出现的指定元素的索引，第三个最后一次出现的指定元素的索引。

第一个：查询Vector容器中是否包含某个元素

```
public boolean contains(Object o) {
    return indexOf(o, 0) >= 0;
}
```

我们发现，查询Vector是否包含某个元素时候，其实是调用了第二个方法，那我们直接就看第二个

第二个：查询第一次出现的指定元素的索引

```
    if (o == null) {
        //从index处一个一个搜索
        for (int i = index ; i < elementCount ; i++)
            if (elementData[i]==null)
               return i;
    } else {
        for (int i = index ; i < elementCount ; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

第三个：查询最后一次出现的指定元素的索引

```
public synchronized int lastIndexOf(Object o, int index) {
    if (index >= elementCount)
        throw new IndexOutOfBoundsException(index + " >= "+ elementCount);
    if (o == null) {
        //从index处正向搜索，默认从索引为elementCount-1处开始
       for (int i = index; i >= 0; i--)
           if (elementData[i]==null)
                return i;
    } else {
        for (int i = index; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

小结：

1）线程安全：

从上面的构造方法还有增删改查的操作其实我们都发现了，都有这么一个synchronized关键字，就是这个关键字为Vector容器提供了一个安全机制，保证了线程安全。

2）构造方法：

Vector实际上是通过一个数组去保存数据的。当我们构造Vecotr时；使用默认构造函数，默认容量大小是10。

3）增加元素：扩容规则

当Vector容量不足以容纳全部元素时，Vector的容量会增加。若容量增加系数 大于0，则将容量的值增加“容量增加系数”；否则，将容量大小增加一倍。

4）克隆：

Vector的克隆函数，即是将全部元素克隆到一个数组中。

（5）遍历

不过到这可还没结束，还有重要的一点我们还没说，那就是遍历。其实在我之前的文章介绍ArrayList时候已经提到过了，遍历方式也就那么几种，既然Vector是基于数组实现的，那么遍历方式肯定也是随机访问最快。在这里代码演示几个：

```
    System.err.print(string);
}
//第二种
t.forEach(new Consumer<String>() {
    @Override
    public void accept(String t) {
        System.out.print(t);
    }
});
  第三种：效率最高  数组按照序号访问
for (int i = 0; i < t.size(); i++) {
    System.out.print(t.get(i));
}
//第四种
Iterator<String> it = t.iterator();
while (it.hasNext()) {
    String string = (String) it.next();
    System.err.print(string);
}
//第五种
Enumeration<String> enume = t.elements();
while(enume.hasMoreElements()){
     System.out.print(enume.nextElement().toString());
}
```

三、Vector与其他容器的区别

源码看完了，对于Vector的实现，我相信你也基本上明白其内部实现了，下面就看看他和别的容器的区别，在文章一开始我们就提到了Vector其实基本上和ArrayList一样的，下面对比分下一下：

ArrayList是线程非安全的，这很明显，因为ArrayList中所有的方法都不是同步的，在并发下一定会出现线程安全问题。另一个方法就是Vector，它是ArrayList的线程安全版本，其实现90%和ArrayList都完全一样，区别在于：

1、Vector是线程安全的，ArrayList是线程非安全的

2、Vector可以指定增长因子，如果该增长因子指定了，那么扩容的时候会每次新的数组大小会在原数组的大小基础上加上增长因子；如果不指定增长因子，那么就给原数组大小*2，源代码是这样的：

```
int newCapacity = oldCapacity + ((capacityIncrement > 0) ? capacityIncrement : oldCapa
```

OK，今天的文章就分享到这里，如有问题还请批评指正。

> 欢迎关注微信公众号：java的架构师技术栈，回复关键字可获取计算机系列各种教程资源。包含java基础、java进阶、java工具、java框架、java架构师、python、android、微信小程序、数据库、前端、神经网络、机器学习等等各个方面的教程资源。

面试官问我同步容器（如Vector）的所有操作一定是线程安全的吗？我懵了！ (https://www.hollischuang.com/archives/3935)

2019-08-25 分类：Java (https://www.hollischuang.com/archives/category/java) / 并发编程 (https://www.hollischuang.com/archives/category/java/%e5%b9%b6%e5%8f%91%e7%bc%96%e7%a8%8b) 阅读(3707) 评论(0)

为了方便编写出线程安全的程序，Java里面提供了一些线程安全类和并发工具，比如：同步容器、并发容器、阻塞队列等。

最常见的同步容器就是Vector和Hashtable了，那么，同步容器的所有操作都是线程安全的吗？

这个问题不知道你有没有想过，本文就来深入分析一下这个问题，一个很容易被忽略的问题。

Java中的同步容器

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

本文拿相对简单的Vecotr来举例，我们先来看下Vector中几个重要方法的源码：

```
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}

public synchronized E remove(int index) {
    modCount++;
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    E oldValue = elementData(index);

    int numMoved = elementCount - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--elementCount] = null; // Let gc do its work

    return oldValue;
}

public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    return elementData(index);
}
```

可以看到，Vector这样的同步容器的所有公有方法全都是synchronized的，也就是说，我们可以在多线程场景中放心的使用单独这些方法，因为这些方法本身的确是线程安全的。

但是，请注意上面这句话中，有一个比较关键的词：单独

```
public Object remove(Vector v){
  return v.removeLast();
} 单独容器操作线程安全
```

因为，虽然同步容器的所有方法都加了锁，但是对这些容器的复合操作无法保证其线程安全性。需要客户端通过主动加锁来保证。

简单举一个例子，我们定义如下删除Vector中最后一个元素方法：

对Vector复合操作无法保证安全性

```
public Object deleteLast(Vector v){
    int lastIndex  = v.size()-1;
    v.remove(lastIndex);
}
```

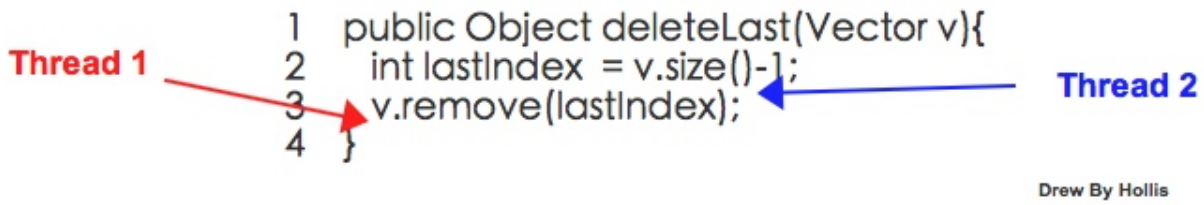
上面这个方法是一个复合方法，包括size()和remove()，乍一看上去好像并没有什么问题，无论是size()方法还是remove()方法都是线程安全的，那么整个deleteLast方法应该也是线程安全的。

但是时，如果多线程调用该方法的过程中，remove方法有可能抛出ArrayIndexOutOfBoundsException。

```
Exception in thread "Thread-1" java.lang.ArrayIndexOutOfBoundsException: Array index out of range: 879
    at java.util.Vector.remove(Vector.java:834)
    at com.hollis.Test.deleteLast(EncodeTest.java:40)
    at com.hollis.Test$2.run(EncodeTest.java:28)
    at java.lang.Thread.run(Thread.java:748)
```

我们上面贴了remove的源码，我们可以分析得出：当index >= elementCount时，会抛出ArrayIndexOutOfBoundsException，也就是说，当当前索引值不再有效的时候，将会抛出这个异常。

因为removeLast方法，有可能被多个线程同时执行，当线程2通过index()获得索引值为10，在尝试通过remove()删除该索引位置的元素之前，线程1把该索引位置的值删除掉了，这时线程一在执行时便会抛出异常。



为了避免出现类似问题，可以尝试加锁：

显示加锁保证复合操作线程安全

```
public void deleteLast() {
    synchronized (v) {
        int index = v.size() - 1;
        v.remove(index);
    }
}
```

如上，我们在deleteLast中，对v进行加锁，即可保证同一时刻，不会有其他线程删除掉v中的元素。

另外，如果以下代码会被多线程执行时，也要特别注意：

```
for (int i = 0; i < v.size(); i++) {
    v.remove(i);
}
```

由于，不同线程在同一时间操作同一个Vector，其中包括删除操作，那么就同样有可能发生线程安全问题。所以，在使用同步容器的时候，如果涉及到多个线程同时执行删除操作，就要考虑下是否需要加锁。

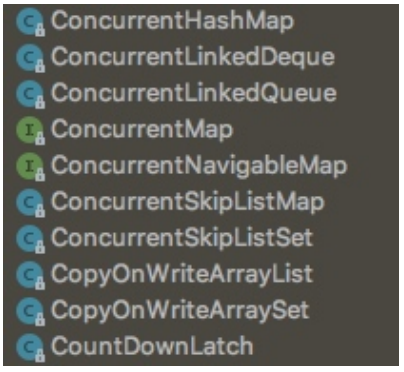
同步容器的问题

前面说过了，同步容器直接保证耽搁操作的线程安全性，但是无法保证复合操作的线程安全，遇到这种情况时，必须要通过主动加锁的方式来实现。

而且，除此之外，同步容易由于对其所有方法都加了锁，这就导致多个线程访问同一个容器的时候，只能进行顺序访问，即使是不同的操作，也要排队，如get和add要排队执行。这就大大的降低了容器的并发能力。

并发容器

针对前文提到的同步容器存在的并发度低问题，从Java5开始，java.util.concurent包下，提供了大量支持高效并发的访问的集合类，我们称之为并发容器。



针对前文提到的同步容器的复合操作的问题，一般在Map中发生的比较多，所以在ConcurrentHashMap中增加了对常用复合操作的支持，比如”若没有则添加”：putIfAbsent()，替换：replace()。这2个操作都是原子操作，可以保证线程安全。

另外，并发包中的CopyOnWriteArrayList和CopyOnWriteArraySet是Copy-On-Write的两种实现。

Copy-On-Write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。

CopyOnWriteArrayList中add/remove等写方法是需要加锁的，而读方法是没有加锁的。

这样做的好处是我们可以对CopyOnWrite容器进行并发的读，当然，这里读到的数据可能不是最新的。因为写时复制的思想是通过延时更新的策略来实现数据的最终一致性的，并非强一致性。

但是，作为代替Vector的CopyOnWriteArrayList并没有解决同步容器的复合操作的线程安全性问题。

总结

本文介绍了同步容器和并发容器。

同步容器是通过加锁实现线程安全的，并且只能保证单独的操作是线程安全的，无法保证复合操作的线程安全性。并且同步容器的读和写操作之间会互相阻塞。

并发容器是Java 5中提供的，主要用来代替同步容器。有更好的并发能力。而且其中的ConcurrentHashMap定义了线程安全的复合操作。

在多线程场景中，如果使用并发容器，一定要注意复合操作的线程安全问题。必要时候要主动加锁。

在并发场景中，建议直接使用java.util.concurent包中提供的容器类，如果需要复合操作时，建议使用有些容器自身提供的复合方法。

Vector的子类Stack：数组实现的栈结构

在Java中Stack类表示后进先出（LIFO）的对象堆栈，是用Vector实现的Stack,栈的入栈、出栈结构都是通过对Vector内的elementData数组尾部增删元素实现的。

```
public class Stack<E> extends Vector<E> {
    构造器
    public Stack() {
    }
empty() 测试堆栈是否为空。

peek() 查看堆栈顶部的对象，但不从堆栈中移除它。

pop() 移除堆栈顶部的对象，并作为此函数的值返回该对象。（弹出Vector内部的数组尾部）

push(E item) 把项压入堆栈顶部。（压入Vector内部的数组尾部）

search(Object o) 返回对象在堆栈中的位置，以 1 为基数
}
}

/**
 *  push函数：将元素存入栈顶
 */
public E push(E item) {
    // 将元素存入栈顶。就是添加到Vector内部的数组的尾部
    // addElement()的实现在Vector.java中
    addElement(item);

    return item;
}
public synchronized void addElement(E obj) { //Vector的addElement方法
        modCount++;
        ensureCapacityHelper(elementCount + 1);
        elementData[elementCount++] = obj;
}
```

```
    /**
     * pop函数：返回栈顶元素，并将其从栈(Vector内部的数组elementData)中删除
     */
    public synchronized E pop() {
        E    obj;
        int    len = size();
        obj = peek();//返回栈顶元素（数组尾部元素）return elementData(arrayLen - 1);
        // 删除栈顶元素，removeElementAt()的实现在Vector.java中
        removeElementAt(len - 1); elementData[--elementCount]
        return obj;
    }

    /**
     * peek函数：返回栈顶元素，不执行删除操作
     */
    public synchronized E peek() {
        int    len = size();

        if (len == 0)
            throw new EmptyStackException();
        // 返回栈顶元素，elementAt()具体实现在Vector.java中
        return elementAt(len - 1);
    }

    /**
     * 栈是否为空
     */
    public boolean empty() {
        return size() == 0;
    }

    /**
     * 查找“元素o”在栈中的位置：由栈底向栈顶方向数
     */
    public synchronized int search(Object o) {
        // 获取元素索引，elementAt()具体实现在Vector.java中
        int i = lastIndexOf(o);

        if (i >= 0) {
            return size() - i;
        }
        return -1;
    }
```