**1.Vector源码解析**

**1.1.简介**

通过阅读源码发现Vector和ArrayList基本上是很相似的，只是Vector的增删改查方法上都加了synchronized关键字，所以是线程安全的，但是直接通过synchronized同步机制实现的线程安全会导致效率很低，所以基本上Vector很少被使用，下面就是Vector的增删改查方法和一起辅助方法的源码分析。

**1.2.成员变量**

|  |
| --- |
| **底层数组**  **protected** Object[] elementData;  数组元素的个数  **protected** **int** elementCount;  扩容大小，每次扩容增长的容量，默认为原来的两倍  **protected** **int** capacityIncrement; |

**1.3.构造器**

|  |
| --- |
| **设定初始容量和容量增长的构造器**  **public** Vector(**int** initialCapacity, **int** capacityIncrement) {  **super**();  **if** (initialCapacity < 0)  **throw** **new** IllegalArgumentException("Illegal Capacity: "+  initialCapacity);  **this**.elementData = **new** Object[initialCapacity];  **this**.capacityIncrement = capacityIncrement;  }  指定初始容量的构造器，增长容量0，到grow会发现，增长容量为0，扩容两倍，否则扩容oldCapacity + capacityIncrement  **public** Vector(**int** initialCapacity) {  **this**(initialCapacity, 0);  }  无参构造器，默认初始化10，增长容量为0，调用初始化容量的构造器  **public** Vector() {  **this**(10);  }  集合构造器，将集合转成数组，如果不是object[]，就自己构建object[]类型  **public** Vector(Collection<? **extends** E> c) {  elementData = c.toArray();  elementCount = elementData.length;  // c.toArray might (incorrectly) not return Object[] (see 6260652)  **if** (elementData.getClass() != Object[].**class**)  elementData = Arrays.*copyOf*(elementData, elementCount, Object[].**class**);  } |

**1.4.增加元素方法**

|  |
| --- |
| **调用**ensureCapacityHelper对其进行grow准备，然后直接把元素放到数组表尾  **public** **synchronized** **boolean** add(E e) {  modCount++;  ensureCapacityHelper(elementCount + 1);  elementData[elementCount++] = e;  **return** **true**;  }  **如果形参>数组的元素长度，调用grow**  **private** **void** ensureCapacityHelper(**int** minCapacity) {  // overflow-conscious code  **if** (minCapacity - elementData.length > 0)  grow(minCapacity);  }  **--------------------------------------------------------------**  **底层调用**insertElementAt方法进行添加  **public** **void** add(**int** index, E element) {  insertElementAt(element, index);  }  判断索引值是否超多了元素的个数，如果超过，抛出越界异常；然后确认是否扩容，对插入位置之后的的元素进行移动，，然后将带插入的obj，插入到index 位置  **public** **synchronized** **void** insertElementAt(E obj, **int** index) {  modCount++;  **if** (index > elementCount) {  **throw** **new** ArrayIndexOutOfBoundsException(index  + " > " + elementCount);  }  ensureCapacityHelper(elementCount + 1);  System.*arraycopy*(elementData, index, elementData, index + 1, elementCount - index);  elementData[index] = obj;  elementCount++;  }  -------------------------------------------------------------------  **public** **synchronized** **boolean** addAll(**int** index, Collection<? **extends** E> c) {  modCount++;  **if** (index < 0 || index > elementCount)  **throw** **new** ArrayIndexOutOfBoundsException(index);  将集合转换成数组，记录数组的长度，判断是否需要扩容，计算需要移动的位置，如果移动位置大于0，就index之后的数组元素，进行移动。  Object[] a = c.toArray();  **int** numNew = a.length;  ensureCapacityHelper(elementCount + numNew);  **int** numMoved = elementCount - index;  **if** (numMoved > 0)  System.*arraycopy*(elementData, index, elementData, index + numNew,  numMoved);  System.*arraycopy*(a, 0, elementData, index, numNew);  elementCount += numNew;  **return** numNew != 0;  }  将集合转为数组，进行扩容判断后，直接加到表尾  **public** **synchronized** **boolean** addAll(Collection<? **extends** E> c) {  modCount++;  Object[] a = c.toArray();  **int** numNew = a.length;  ensureCapacityHelper(elementCount + numNew);  从a数组的0下标开始拷贝，目标位置为elmentdata数组的elementCount，拷贝容量为numNew  System.*arraycopy*(a, 0, elementData, elementCount, numNew);  elementCount += numNew;  **return** numNew != 0;  }  与add（e）类似  **public** **synchronized** **void** addElement(E obj) {  modCount++;  ensureCapacityHelper(elementCount + 1);  elementData[elementCount++] = obj;  } |

**1.5.扩容方法**

|  |
| --- |
| private static final int MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8;  得到old容量，如果增长容量大于0，new容量为old+增长容量，如果不大于0，new容量为2\*old，如果新容量比形参小，就让其等于形参容量，如果new容量比最大***MAX\_ARRAY\_SIZE大，就调用huge（形参），要么取Integer的MaxValue，要么就为形参容量，然后进行数组拷贝***  **private** **void** grow(**int** minCapacity) {  // overflow-conscious code  **int** oldCapacity = elementData.length;  **int** newCapacity = oldCapacity + ((capacityIncrement > 0) ?  capacityIncrement : oldCapacity);  **if** (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  **if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0)  newCapacity = *hugeCapacity*(minCapacity);  elementData = Arrays.*copyOf*(elementData, newCapacity);  }  **private** **static** **int** hugeCapacity(**int** minCapacity) {  **if** (minCapacity < 0) // overflow  **throw** **new** OutOfMemoryError();  **return** (minCapacity > ***MAX\_ARRAY\_SIZE***) ?  Integer.***MAX\_VALUE*** :  ***MAX\_ARRAY\_SIZE***;  } |

**1.6.删除方法**

|  |
| --- |
| 得到删除的元素的值，然后计算移动步数，进行数组元素移动  **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** **boolean** remove(Object o) {  **return** removeElement(o);  }  遍历一遍设置null  **public** **synchronized** **void** removeAllElements() {  modCount++;  // Let gc do its work  **for** (**int** i = 0; i < elementCount; i++)  elementData[i] = **null**;  elementCount = 0;  }  **public** **synchronized** **boolean** removeElement(Object obj) {  modCount++;  **int** i = indexOf(obj);  **if** (i >= 0) {  removeElementAt(i);  **return** **true**;  }  **return** **false**;  }  从0开始找  **public** **int** indexOf(Object o) {  **return** indexOf(o, 0);  }  从index开始找  **public** **synchronized** **int** indexOf(Object o, **int** index) {  **if** (o == **null**) {  **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;  }  --------------------------------------  与remove（index）类似，一个有返回值，一个没有  **public** **synchronized** **void** removeElementAt(**int** index) {  modCount++;  **if** (index >= elementCount) {  **throw** **new** ArrayIndexOutOfBoundsException(index + " >= " +  elementCount);  }  **else** **if** (index < 0) {  **throw** **new** ArrayIndexOutOfBoundsException(index);  }  **int** j = elementCount - index - 1;  **if** (j > 0) {  System.*arraycopy*(elementData, index + 1, elementData, index, j);  }  elementCount--;  elementData[elementCount] = **null**; /\* to let gc do its work \*/  }  左闭右开，计算移动步数，长度-toIndex，然后将后面的要删除的个数，实际为toIndex-fromIndex，置为空  **protected** **synchronized** **void** removeRange(**int** fromIndex, **int** toIndex) {  modCount++;  **int** numMoved = elementCount - toIndex;  System.*arraycopy*(elementData, toIndex, elementData, fromIndex, numMoved);  // Let gc do its work  **int** newElementCount = elementCount - (toIndex-fromIndex);  **while** (elementCount != newElementCount)  elementData[--elementCount] = **null**;  } |

**1.7.get个方法**

|  |
| --- |
| 基本都是调用elementData方法；  **public** **synchronized** E get(**int** index) {  **if** (index >= elementCount)  **throw** **new** ArrayIndexOutOfBoundsException(index);  **return** elementData(index);  }  获取第一个元素  **public** **synchronized** E firstElement() {  **if** (elementCount == 0) {  **throw** **new** NoSuchElementException();  }  **return** elementData(0);  }  获取最后一个元素  **public** **synchronized** E lastElement() {  **if** (elementCount == 0) {  **throw** **new** NoSuchElementException();  }  **return** elementData(elementCount - 1);  } |

**1.8.set方法**

|  |
| --- |
| **判断index的值是否超过数组长度；获取旧值，然后设置新值，返回旧值**  **public** **synchronized** E set(**int** index, E element) {  **if** (index >= elementCount)  **throw** **new** ArrayIndexOutOfBoundsException(index);  E oldValue = elementData(index);  elementData[index] = element;  **return** oldValue;  }  判断长度问题，然后直接设置新值  **public** **synchronized** **void** setElementAt(E obj, **int** index) {  **if** (index >= elementCount) {  **throw** **new** ArrayIndexOutOfBoundsException(index + " >= " +  elementCount);  }  elementData[index] = obj;  } |

**2.Stack源码解析**

|  |
| --- |
| 由于stack继承了Vector，基本所有方法都是沿用Vector，只包含了一些栈的特别方法，底层依然调用父类的方法  空构造器，就是调用Vector无参构造器  **public** Stack() {  }  调用父类addElement方法，从前往后插入  **public** E push(E item) {  addElement(item);  **return** item;  }  调用peek方法，返回数组最后一个元素，然后调用父类removeElementAt，将最后一个元素删除  **public** **synchronized** E pop() {  E obj;  **int** len = size();  obj = peek();  removeElementAt(len - 1);  **return** obj;  }  返回数组最后一个元素  **public** **synchronized** E peek() {  **int** len = size();  **if** (len == 0)  **throw** **new** EmptyStackException();  **return** elementAt(len - 1);  }  **public** **boolean** empty() {  **return** size() == 0;  }  调用lastIndexOf，从后面开始找，直到找到了，返回下标size-i  **public** **synchronized** **int** search(Object o) {  **int** i = lastIndexOf(o);  **if** (i >= 0) {  **return** size() - i;  }  **return** -1;  } |