1. **概述**

ArrayList是**基于数组实现**的，是一个**动态数组**，其**容量能自动增长**，类似于C语言中的动态申请内存，动态增长内存。

ArrayList**不是线程安全**的，只能用在单线程环境下，多线程环境下可以考虑用Collections.synchronizedList(List l)函数返回一个线程安全的ArrayList类，也可以使用concurrent并发包下的CopyOnWriteArrayList类。

ArrayList实现了Serializable接口，因此它**支持序列化**，能够通过序列化传输，实现了RandomAccess接口，**支持快速随机访问**，实际上就是通过下标序号进行快速访问，实现了Cloneable接口，**能被克隆**。

每个ArrayList实例都有一个容量，该容量是指用来存储列表元素的数组的大小。它总是至少等于列表的大小。随着向ArrayList中不断添加元素，其容量也自动增长。自动组增长会带来数据向新数组的重新拷贝，因此，如果可预知数据量的多少，可在构造ArrayList时指定其容量。在添加大量元素前，应用程序也可以使用ensureCapacity操作来增加ArrayList实例的容量，这样可以减少递增式再分配的数量。

注意，此实现不是同步的。如果多个线程同时访问一个ArrayList实例，而其中至少一个线程从结构上修改了列表，那么它必须保持外部同步。

1. **实现**

对于ArrayList而言，它实现List接口、底层使用数组保存所有元素。其操作基本上时对数组的操作。

1. **私有属性：**

**transient** Object[] **elementData**; *// non-private to simplify nested class access***private int size**;

elementData存储ArrayList内的元素，size表示它包含的元素的数量。

**关键字transient**。Java的serialization提供了一种持久化对象实例的机制。当持久化对象时，可能有一个特殊的对象数据成员，我们不想用serialization机制来保存它。为了在一个特定对象的一个域上关闭serialization，可以在这个域前加上关键字transient。**被标记为transient的属性在对象被序列化的时候不会被保存**。

1. **构造方法：**

ArrayList提供了三种方式的构造器，可以**构造一个默认初始容量为10的空列表**、**构造一个指定初始容量的空列表**以及**构造一个包含指定collection的元素的列表**，这些元素按照该collection的迭代器返回它们的顺序排列的。

*// 指定初始容量的空列表*

**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);  
 }  
}

*// 无参构造函数。默认容量为10*

**public** ArrayList() {  
 **this**.**elementData** = ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***;  
}

*// 包含collection的元素的列表*

**public** ArrayList(Collection<? **extends** E> c) {  
 **elementData** = c.toArray();  
 **if** ((**size** = **elementData**.**length**) != 0) {  
 *// c.toArray might (incorrectly) not return Object[] (see 6260652)* **if** (**elementData**.getClass() != Object[].**class**)  
 **elementData** = Arrays.*copyOf*(**elementData**, **size**, Object[].**class**);  
 } **else** {  
 *// replace with empty array.* **this**.**elementData** = ***EMPTY\_ELEMENTDATA***;  
 }  
}

1. **元素存储：**

ArrayList提供了5种添加元素的方法。

set(int index, E element)

add(E e)

add(int index, E element)

addAll(Collection<? extends E> c)

addAll(int index, Collection<? extends E> c)

*// 用指定的元素替代此列表中指定位置上的元素，并返回以前位于该位置上的元素***public** E set(**int** index, E element) {  
 rangeCheck(index);  
  
 E oldValue = elementData(index);  
 **elementData**[index] = element;  
 **return** oldValue;  
}

*// 将指定的元素添加到此列表的尾部*

**public boolean** add(E e) {  
 *//确保数组有足够的空间来存储对象e*

ensureCapacityInternal(**size** + 1); *// Increments modCount!!*

*//将对象e存放到数组的末尾* **elementData**[**size**++] = e;  
 **return true**;  
}

*// 将指定的元素插入此列表中的指定位置*

*// 如果当前位置有元素，则向右移动当前位于该位置的元素以及所有后续元素（索引+1）*

**public void** add(**int** index, E element) {  
 rangeCheckForAdd(index);  
  
 ensureCapacityInternal(**size** + 1); *// 如果数组长度不足，将进行扩容*

*// 将元素从index位置开始、长度为size-index的元素，*

*//拷贝到从下标为index+1位置开始的新的elementData数组中。*

*//即将当前位于该位置的元素以及所有后续元素右移一个位置* System.*arraycopy*(**elementData**, index, **elementData**, index + 1,  
 **size** - index);  
 **elementData**[index] = element;  
 **size**++;  
}

*// 按照指定collection的迭代器所返回的元素顺序，*

*//将该collection中的所有元素添加到此列表的尾部。*

**public boolean** addAll(Collection<? **extends** E> c) {  
 Object[] a = c.toArray();  
 **int** numNew = a.**length**;  
 ensureCapacityInternal(**size** + numNew); *// Increments modCount* System.*arraycopy*(a, 0, **elementData**, **size**, numNew);  
 **size** += numNew;  
 **return** numNew != 0;  
}

*// 从指定位置开始，将指定collection中的所有元素插入到此列表中。*

**public boolean** addAll(**int** index, Collection<? **extends** E> c) {  
 rangeCheckForAdd(index);  
  
 Object[] a = c.toArray();  
 **int** numNew = a.**length**;  
 ensureCapacityInternal(**size** + numNew); *// Increments modCount* **int** numMoved = **size** - index;  
 **if** (numMoved > 0)  
 System.*arraycopy*(**elementData**, index, **elementData**, index + numNew,  
 numMoved);  
  
 System.*arraycopy*(a, 0, **elementData**, index, numNew);  
 **size** += numNew;  
 **return** numNew != 0;  
}

ArrayList是基于数组实现的，属性中也看到了数组，具体是怎么实现的呢？比如就这个添加元素的方法，如果数组大，则在将某个位置的值设为指定元素即可，如果数组容量不够呢？

答：add(E e)中先调用了ensureCapacity(size+1)方法，之后将元素的索引给elementData[size]，而后size自增。例如初次添加时，size为0，add将elementData[0]赋值为e，然后size设置为1（类似执行以下两条语句elementData[0]=e;size=1）。将元素的索引赋给elementData[size]不是会出现数组越界的情况吗？这里关键就在ensureCapacity(size+1)中了。

1. **元素读取：**

*// 返回此列表中指定位置上的元素*

**public** E get(**int** index) {  
 rangeCheck(index);  
  
 **return** elementData(index);  
}

1. **元素删除：**

ArrayList提供了根据下标对象两种方式的删除功能。

*// 移除此列表中指定位置的元素*

**public** E remove(**int** index) {  
 rangeCheck(index);  
  
 **modCount**++;  
 E oldValue = elementData(index);  
  
 **int** numMoved = **size** - index - 1;  
 **if** (numMoved > 0)  
 System.*arraycopy*(**elementData**, index+1, **elementData**, index,  
 numMoved);  
 **elementData**[--**size**] = **null**; *// clear to let GC do its work* **return** oldValue;  
}

首先是检查范围，修改modCount，保留将要被移除的元素，将移除位置之后的元素向前挪动一个位置，将list末尾元素置空（null），返回被移除的元素。

*// 移除此列表中首次出现的指定元素（如果存在）。这是因为ArrayList中允许存放重复的元素*

**public boolean** remove(Object o) {

*//由于ArrayList中允许存放null，因此下面通过两种情况分别处理*  
 **if** (o == **null**) {  
 **for** (**int** index = 0; index < **size**; index++)  
 **if** (**elementData**[index] == **null**) {

*//类似remove(int index)，移除列表中指定位置上的元素*   
 fastRemove(index);  
 **return true**;  
 }  
 } **else** {  
 **for** (**int** index = 0; index < **size**; index++)  
 **if** (o.equals(**elementData**[index])) {  
 fastRemove(index);  
 **return true**;  
 }  
 }  
 **return false**;  
}

首先通过代码可以看到，当移除成功后返回true，否则返回false。通过遍历element寻找是否存在传入对象，一旦找到就调用fastRemove(index)移除对象。为什么找到了元素就知道了index，不通过remove(index)来移除元素呢？因为fastRemove跳过了判断边界的处理，因为找到元素就相当于确定了index不会超过边界，而且fastRemove并不返回被移除的元素。下面是fastRemove的代码，基本和remove(int index)一致。

**private void** fastRemove(**int** index) {  
 **modCount**++;  
 **int** numMoved = **size** - index - 1;  
 **if** (numMoved > 0)  
 System.*arraycopy*(**elementData**, index+1, **elementData**, index,  
 numMoved);  
 **elementData**[--**size**] = **null**; *// clear to let GC do its work*}

**protected void** removeRange(**int** fromIndex, **int** toIndex) {  
 **modCount**++;  
 **int** numMoved = **size** - toIndex;  
 System.*arraycopy*(**elementData**, toIndex, **elementData**, fromIndex,  
 numMoved);  
  
 *// clear to let GC do its work* **int** newSize = **size** - (toIndex-fromIndex);  
 **for** (**int** i = newSize; i < **size**; i++) {  
 **elementData**[i] = **null**;  
 }  
 **size** = newSize;  
}

此执行过程是将elementData从toIndex位置开始的元素向前移动到fromIndex，然后将toIndex位置之后的元素全部置空顺便修改size。

这个方法是protected，即受保护的方法，为什么定义为protected呢？

答：

1. **调整数组容量：**

从上面介绍的向ArrayList中存储元素的代码中，我们看到，每当向数组中添加元素时，都要去检查添加后元素的个数是否会超出当前数组的长度，如果超出，数组将会进行扩容，以满足添加数据的需求。数组扩容通过一个公开的方法ensureCapacity(int minCapacity)来实现。在时机添加大量元素前，我也可以使用ensureCapacity来手动增加ArrayList实例的容量，以减少递增式再分配的数量。

**public void** ensureCapacity(**int** minCapacity) {  
 **int** minExpand = (**elementData** != ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***)? 0: ***DEFAULT\_CAPACITY***;  
  
 **if** (minCapacity > minExpand) {  
 ensureExplicitCapacity(minCapacity);  
 }  
}

从上述代码中可以看出，数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长大约是其原容量的1.5倍。这种操作的代价是很高的，因此在实际使用时，应尽量避免数组容量的扩张。当我们可预知要保存的元素多少时，要在构造ArrayList实例时，就指定其容量，以避免数组扩容的发生。或者格局实际需求，通过调用ensureCapacity方法来手动增加ArrayList实例的容量。

1. **转为静态数组toArray**

第一个，**调用Arrays.copyOf将返回一个数组**，**数组内容是size个elementData的元素**，即拷贝elementData从0至size-1位置的元素到新数组并返回。

**public** Object[] toArray() {  
 **return** Arrays.*copyOf*(**elementData**, **size**);  
}

第二个，如果**传入数组的长度小于size**，**返回一个新的数组，大小为size，类型与传入数组相同**。所传入数组长度与size**相等**，**则将elementData复制到传入数组中并返回传入的数组**。若传入数组长度**大于**size，**除了复制elementData外，还将把返回数组的第size个元素置为空**。

**public** <T> T[] toArray(T[] a) {  
 **if** (a.**length** < **size**)  
 *// Make a new array of a's runtime type, but my contents:* **return** (T[]) Arrays.*copyOf*(**elementData**, **size**, a.getClass());  
 System.*arraycopy*(**elementData**, 0, a, 0, **size**);  
 **if** (a.**length** > **size**)  
 a[**size**] = **null**;  
 **return** a;  
}

**Fail-Fast机制：**

ArrayList也采用了快速失败机制，通过记录modCount参数来实现。在面对并发的修改时，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险。

**重要总结：**

1、注意其三个不同的构造方法。**无参构造方法构造的ArrayList的容量默认为10**，**带有Collection参数的构造方法，将Collection转化为数组赋给ArrayList的实现数组elementData**。

2、注意扩充容量的方法ensureCapacity。ArrayList在每次增加元素（可能是1个，也可能是一组）时，都要调用该方法来确保足够的容量。当容量不足以容纳当前的元素个数时，就设置新的容量为旧的容量的1.5倍加1，如果设置后的新容量还不够，则直接新容量设置为传入的参数（也就是所需的容量），而后用Arrays.copyOf()方法将元素拷贝到新的数组（详见下面的第3点）。从中可以看出，当容量不够时，每次增加元素，都要将原来的元素拷贝到一个新的数组中，非常之耗时，也因此**建议在事先能确定元素数量的情况下，才使用ArrayList，否则建议使用LinkedList**。

3、ArrayList的实现中大量地调用了Arrays.copyOf()和System.arraycopy()方法。我们有必要对这两个方法的实现做下深入的了解。

首先来看Arrays.copyOf()方法。它有很多个重载的方法，但实现思路都是一样的，我们来看泛型版本的源码：

public static <T> T[] copyOf(T[] original, int newLength) {

return (T[]) copyOf(original, newLength, original.getClass());

}

很明显调用了另一个copyOf方法，该方法有三个参数，最后一个参数指明要转换的数据的类型，其源码如下：

public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType) {

T[] copy = ((Object)newType == (Object)Object[].class)

? (T[]) new Object[newLength]

: (T[]) Array.newInstance(newType.getComponentType(), newLength);

System.arraycopy(original, 0, copy, 0,

Math.min(original.length, newLength));

return copy;

}

这里可以很明显地看出，该方法实际上是在其内部又创建了一个长度为newlength的数组，调用System.arraycopy()方法，将原来数组中的元素复制到了新的数组中。

下面来看System.arraycopy()方法。该方法被标记了native，调用了系统的C/C++代码，在JDK中是看不到的，但在openJDK中可以看到其源码。该函数实际上最终调用了C语言的memmove()函数，因此它可以保证同一个数组内元素的正确复制和移动，比一般的复制方法的实现效率要高很多，很适合用来批量处理数组。Java强烈推荐在复制大量数组元素时用该方法，以取得更高的效率。

4、ArrayList基于数组实现，可以通过下标索引直接查找到指定位置的元素，因此**查找效率高，但每次插入或删除元素，就要大量地移动元素，插入删除元素的效率低**。

5、在查找给定元素索引值等的方法中，源码都将该元素的值分为null和不为null两种情况处理，ArrayList中允许元素为null。