

# 初探 ArrayList 动态扩容原理

作者: [Gouzhong1223](#)

原文链接: <https://ld246.com/article/1582881797965>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)



## 1.概述

传统的 Java 数组都有一个通病，就是不支持动态扩容，我们在初始化一个数组的时候一旦制定了容量，那么在后面遇到数组越界的时候，就会抛出异常，这对于我们的开发是非常的不方便的，于是就有了对数组进行了封装的容器类：ArrayList。ArrayList 是一种支持动态扩容的容器，**其本质上是对数组进行了封装**，所以，数组所持有的特性 ArrayList 依然持有，比如说随机访问。

## 2.ArrayList初始化

### 2.1 ArrayList 的空参构造如下：

```
/**
 * Constructs an empty list with an initial capacity of ten.
 */
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

这里可以看到，当我们在初始化一个 ArrayList 的时候，成员变量被赋予一个叫做

DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA 的成语变量，下面来看一下 elementData 还有 DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA 具体的含义。

```
/**
 * The array buffer into which the elements of the ArrayList are stored.
 * The capacity of the ArrayList is the length of this array buffer. Any
 * empty ArrayList with elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA
 * will be expanded to DEFAULT_CAPACITY when the first element is added.
 */
transient Object[] elementData; // non-private to simplify nested class access

java.util.ArrayList<E>
transient Object[] elementData

的数组缓冲区存储到其中该ArrayList的元素。该ArrayList的容量是这样的阵列缓冲器的长度。添加的第一个元素时
与elementData中== DEFAULTCAPACITY_EMPTY_ELEMENTDATA任何空的ArrayList将扩大到
DEFAULT_CAPACITY.
默认 < 1.8 >
```

```
/**
 * Shared empty array instance used for default sized empty instances. We
 * distinguish this from EMPTY_ELEMENTDATA to know how much to inflate when
 * first element is added.
 */
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

java.util.ArrayList<E>
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {}

用于默认共享空数组实例尺寸空实例。我们区分从EMPTY_ELEMENTDATA知道有多少在添加第一个元素是放大。
默认 < 1.8 >
```

这个地方就可以很明显的看出来，在初始化一个 ArrayList 的时候并没有指定容量，而是把一个默认空数组赋予 ArrayList 的 elementData，很明显，ArrayList 装在元素的功能其实是 elementData 来成的，所以，刚刚初始化的 ArrayList 数组可以堪称是 size 是 0 的数组。

## 2.2 ArrayList 初始化容量

初始化出来一个空的容器之后，开始尝试着给这个容器添加一些元素，下面，很好奇这个容器是怎么一个容量为零的数组中添加元素的。

现在就往数组中添加一个字符串，在 add 方法执行的地方，打上一个断点

```
18
19 public static void main(String[] args) {
20
21     ArrayList arrayList = new ArrayList();
22
23     arrayList.add("1");
24
25 }
26
27 }
28
```

然后 debug 执行代码，进入 add 方法

```
455 /**
456  * Appends the specified element to the end of this list.
457  *
458  * @param e element to be appended to this list e: "1"
459  * @return <tt>true</tt> (as specified by {@link Collection#add})
460  */
461 public boolean add(E e) { e: "1"
462     ensureCapacityInternal( minCapacity: size + 1); // Increments modCount!!
463     elementData[size++] = e;
464     return true;
465 }
```

从上面的图可以看出来，进入 add 方法之后会进入一个 ensureCapacityInternal 方法中，这个 size 是初始化 ArrayList 默认的容量，是 0，我们进入 ensureCapacityInternal，看一看这里面是怎么执行的

```
230 private void ensureCapacityInternal(int minCapacity) { minCapacity: 1
231     ensureExplicitCapacity( calculateCapacity( elementData, minCapacity)); minCapacity: 1
232 }
```

这里进入 ensureCapacityInternal 方法之后，会调用一个 ensureExplicitCapacity 方法，而这个方法



依赖一个叫做`calculateCapacity`的方法，点进去看，这个里面是怎么执行的

```
223 private static int calculateCapacity(Object[] elementData, int minCapacity) { elementData: Object[0]@488 minCapacity: 1
224     if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) { elementData: Object[0]@488
225         return Math.max(DEFAULT_CAPACITY, minCapacity);
226     }
227     return minCapacity;
228 }
```

这里面有一段代码

```
if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
    return Math.max(DEFAULT_CAPACITY, minCapacity);
}
```

看到这个地方的`DEFAULTCAPACITY_EMPTY_ELEMENTDATA`是不是很眼熟啊，这个不就是最开始初始化 `ArrayList` 的时候赋予的默认数组么？所以，这个方法就是判断这个需要添加元素的数组是否是初始化的数组的，如果是，那就会再做一个比较，看一下怎么比较的，看这个`minCapacity`，翻译过就是最小容量，什么最小容量呢？意思就是说，当前这个`ArrayList`的 `add` 方法执行成功之后，数组一个长度，我们当前的`ArrayList` 是刚刚初始化的，`size` 是 0，如果成功添加这个元素之后，那就是 1 所以`minCapacity`就是 1，和它作比较的`DEFAULT_CAPACITY`呢？

```
112 /**
113     * Default initial capacity.
114     */
115     private static final int DEFAULT_CAPACITY = 10;
```

是 10，这个 10 指代的是初始容量，**注意：初始容量和默认容量是有区别的**。这里会在他们两个数中去一个大的数，然后返回。按照逻辑，这里返回的应该是 10，这个 10 将会被传递给`ensureExplicitCapacity`，我们再进入这个方法里面看一看这是怎么执行的。

```
234 private void ensureExplicitCapacity(int minCapacity) { minCapacity: 10
235     modCount++;
236
237     // overflow-conscious code
238     if (minCapacity - elementData.length > 0)
239         grow(minCapacity);
240 }
```

这个 `modCount` 指的是对这个 `ArrayList` 的操作次数，然后就会进入一个判断：

```
// overflow-conscious code
if (minCapacity - elementData.length > 0)
    grow(minCapacity);
```

这里就会把帮刚刚重新初始化的一个容量和当前 `ArrayList` 的大小做差比较大小，看重新初始化的这容量是不是可以再容下一个元素。因为当前的 `ArrayList` 是刚刚初始化的，里面什么都没有，所以就跳入`grow`方法，并将我们最新初始化的一个容量传递过去，点进去看一下`grow`方法是怎样执行的。

```
250     /**
251     * Increases the capacity to ensure that it can hold at least the
252     * number of elements specified by the minimum capacity argument.
253     *
254     * @param minCapacity the desired minimum capacity
255     */
256     private void grow(int minCapacity) {
257         // overflow-conscious code
258         int oldCapacity = elementData.length;
259         int newCapacity = oldCapacity + (oldCapacity >> 1);
260         if (newCapacity - minCapacity < 0)
261             newCapacity = minCapacity;
262         if (newCapacity - MAX_ARRAY_SIZE > 0)
263             newCapacity = hugeCapacity(minCapacity);
264         // minCapacity is usually close to size, so this is a win:
265         elementData = Arrays.copyOf(elementData, newCapacity);
266     }
```

在这个grow方法中，会做两个判断，说一下执行步骤：

- 首先把这个 当前ArrayList 的底层容器 elementData 里面的长度取出来
- 然后将老容量加上老容量乘以 0.5 倍的数值定义为新容量（这里老容量本来就是 0，乘以 1.5 倍之还是 0）
- 比较新容量和当前重新初始化的一个数组容量（10）的大小，如果新容量小于这个，就把新容量重赋值为重新新初始化的数组容量。
- 然后再判断新容量和MAX\_ARR\_SIZE 的数值大小

```
242     /**
243     * The maximum size of array to allocate.
244     * Some VMs reserve some header words in an array.
245     * Attempts to allocate larger arrays may result in
246     * OutOfMemoryError: Requested array size exceeds VM limit
247     */
248     private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
59     /**
60     * A constant holding the maximum value an {@code int} can
61     * have,  $2^{31}-1$ .
62     */
63     @Native public static final int MAX_VALUE = 0x7fffffff;
```

如果大于这个数，name 就会给新容量重新赋值为一个更大的数

- 然后就会把当前的 elementData 拷贝到一个新的，容量为 10 的一个数组中
- 至此，ArrayList 的第一次初始化容量就完毕了  
这时候就要开始往里面添加元素了

```
461     public boolean add(E e) { e: "1"
462         ensureCapacityInternal( minCapacity: size + 1); // Increments modCount!!
463         elementData[size++] = e; e: "1"
464         return true;
465     }
```

这里 size 是一个后自增的使用方法，添加完成之后，size 就会+1。

### 3.ArrayList 其余时候元素的添加

#### 3.1 添加元素之后不会超过 ArrayList 的容量

当我们再次往里面添加元素的时候，当走到`calculateCapacity`方法的时候，判断为不为新初始化的数时，就会直接返回当前`ArrayList`的 `size+1`。

```
223 @ private static int calculateCapacity(Object[] elementData, int minCapacity) { elementData: Object[10]@507 minCapacity: 2
224   if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) { elementData: Object[10]@507
225     return Math.max(DEFAULT_CAPACITY, minCapacity);
226   }
227   return minCapacity; minCapacity: 2
228 }
```

走到`ensureExplicitCapacity`方法的时候，也就自然**不会触发我们的扩容机制了**

```
234 private void ensureExplicitCapacity(int minCapacity) { minCapacity: 2
235   modCount++;
236
237   // overflow-conscious code
238   if (minCapacity - elementData.length > 0)
239     grow(minCapacity); minCapacity: 2
240 }
241
```

后面的就是添加元素，然后 `size+1` 了。

### 3.2 添加元素之后超过最大容量

比如说，当我们添加第十一个元素的时候，原本的 `ArrayList` 容量为 10，这时候，走到`ensureExplicitCapacity`方法的时候，添加之后的最小容量是大于我们的 `elementData` 的，所以，又会进入到`grow`方法中，在这里，会把当前数组的容量和当前容量的 0.5 倍加起来，赋值给一个新的容量，然后将整个 `elementData` 数组拷贝到这个拥有新容量的数组中，老的数组 gc 周期就会将它处理掉。

## 完结