链滴

集合类源码解析之 ArrayList

作者：jilinwula

原文链接：https://ld246.com/article/1634887780712

来源网站：链滴

上在一篇中我们已经介绍过了ArrayList集合类是List接口的实现类，所以它会默认具有List接口的相特性。所以在这里我们就可以说ArrayList是一个能够保证元素的插入顺序并且可以保存重复元素的集类。除了上述的特性外，ArrayList和其它集合类相比还可以保存null元素到集合类中（并不是所有的合类都支持此功能）。ArrayList集合类底层是通过动态数组的方式实现的。动态数组的意思是说Array ist的底层数组大小是可以动态改变的。我们知道在Java中数组的大小是不可以改变的，也就是说如果组初始化成功，那么在使用时就一定是这么大的数组了。如果在使用时超过了数组的最大索引时，那虚拟机就会抛出异常。既然Java中数组的大小是不可改变的，那么ArrayList底层是怎么实现动态数组能的呢。

- **初始化**

其实在ArrayList底层数组也是不可以改变的，底层动态数组的实现逻辑是通过重新创建新数组的方式实现的。也就是说它底层处理的逻辑是当ArrayList发现底层数组的大小已经超过了数组默认初始化大时，就会创建一个新数组，然后把原数组中的数据拷贝到新数组中，然后操作这个新数组使用。如果发现新创建的数组大小还是不够我们存储时，继续重复上面的逻辑。所以我们在使用ArrayList集合类，是不用考虑底层数组的大小的。下面我们通过ArrayList的源码来证明我们刚刚所说的动态数组的实逻辑。

```
private static final long serialVersionUID = 8683452581122892189L;

private static final int DEFAULT_CAPACITY = 10;

private static final Object[] EMPTY_ELEMENTDATA = {};

private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

transient Object[] elementData;

private int size;
```

上面代码是ArrayList源码中申明的所有静态变量或实例变量，在我们后面分析源码时会用到这些变量所以我们要知道这些变量具体的数据类型是什么，好方便我们在分析源码时更好的理解。

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

上面的方法是ArrayList的构造方法，这个方法只实现了一个功能就是将elementData数组设置为一个数组，也可以理解为将ArrayList集合中的底层数组清空。但这时elementData数组并没有被初始化，就是说在我们使用ArrayList集合类是，即使我们创建了ArrayList对象，底层的数组也不会执行初始化作。那ArrayList的底层数组到底是在什么时候才会初始化呢?我们继续看下面的源码。

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1);
    elementData[size++] = e;
    return true;
}
```

上述代码是ArrayList集合类中的添加方法，虽然我们现在还不知道ensureCapacityInternal()方法具的作用是什么，但我们简单分析可知，这段代码执行完后，就会把当前元素添加到ArrayList底层数组第0个索引位置，并且返回true。下面我们按照方法的调用顺序来看一下ensureCapacityInternal()方中的源码。

```
private void ensureCapacityInternal(int minCapacity) {
```

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

    ensureExplicitCapacity(minCapacity);
}
```

这个方法中我们看到有一个if语句，if语句的判断逻辑是：ArrayList中底层数组如果是一个空数组那就执行if语句中的代码。if语句中的代码逻辑是：比较静态变量的值与方法的参数值的大小。方法的参值其实就是集合中已经存储的元素个数然后执行加1后的值。然后将最大的那个值赋给方法参数。代执行到这里使我们知道，if语句中的代码只会执行一次，并且仅当ArrayList中的底层数组必须是空数时，也就是没有被初始化时才会执行。不管if语句是不是执行，方法都会调用ensureExplicitCapacity(方法，唯一的区别就是方法参数大小的问题。并且在此时我们发现现在的ArrayList中底层数组还是没被初始化呢？别着急，我们继续分析下面的方法源码。

```
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

这个方法比较简单主要的功能就是执行if语句。if语句的条件是：比较当前方法参数与ArrayList中底数组的大小。我们知道方法参数一定是一个大于0的数，并且我们知道数组现在还没有被初始化，所调用数组中的length属性时，结果一定是0。所以上述if语句一定会执行，也就是说一定会执行grow(方法。下面我们看一下grow()方法中的源码。

```
private void grow(int minCapacity) {
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

上面的代码貌似看点有复杂，我们暂时不用全部考虑，我只看最后一条代码即可，方法在最后调用了Arrays.copyOf()方法，我们知道该方法的作用是返回一个新数组，并将当前数组的内容拷贝到新数组中并设置新数组的初始化大小。由此可见，这段代码就是ArrayList集合类底层数组的实例化方法。只是不是按照我们传统的编码方法直接执行初始化，而是直接调用了数组的初始化工具类实现的。

- **什么时候创建新数组**

现在我们已经知道了ArrayList数组的底层实现逻辑，那我们现在就会有个疑惑，到底ArrayList中的元个数达到什么数量时，才会重新创建新的数组来存储元素？通过上面的分析，我们知道，ArrayList数的默认模式初始化大小是10。也就是说如果ArrayList中底层数组如果不创建新数组的话，那么此集合多能存储的元素大小就是10。下面我们假如正在第10次调用ArrayList中的add()方法，看来一下此时码中参数的变化。

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // 因为是第10次调用，所以此时size=9
    elementData[size++] = e;
    return true;
}
```

```
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    // 因为数组已经初始化了，所以不会在执行if语句中的代码，因为在add()方法中执行了+1运算，以此时虽然集合中的元素个数是9，但此方法的参数值却为10
    ensureExplicitCapacity(minCapacity);
}

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
    // 按照上述的分析此方法的参数值应该为10 因为数组初始化时的默认大小是10 所以.length返回值也是10 所以if条件是不成产的。所以并不会创建新数组
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

按照上面分析，我们可以得到结论是在ArrayList中，只有当前添加的元素刚好超过了底层数组中的大时，才会创建新的数组来存储元素。

- **新数组的大小**

接下来，我们接着分析，底层每创建一个新数组时和上一个数组相比，这个新数组的大小会比前一个组大多少。按照上面的分析，我们知道，新数组的大小是通过grow()方法计算出来的。下面我们详细析一下grow()方法。我们暂时假设我们正在向ArrayList中添加第11个元素，也就是说现在ArrayList的元素个数已经是10个，底层数组已经达到了最大容量，如果继续添加新元素时，势必会触发grow()法。这里我们省略掉其它方法，只考虑grow()方法中参数的变化。

```
private void grow(int minCapacity) { //  按照上面的分析此时方法的参数值应该为11
    int oldCapacity = elementData.length; // 因为原数组已经达到了最大容量，所以此时为10
    int newCapacity = oldCapacity + (oldCapacity >> 1); // 按照上面的分析，那我们很容易计算来，这个新数组的大小就是15
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

为了验证我们上面的分析，下面我们通过调试源码的方式，用debug来查看一下当我们向ArrayList中加第11个元素时，参数的变化是不是我们上述分析的那样。

```
public class ArrayListTest {
    @Test
    public void test() {
        ArrayList arrayList = new ArrayList();
        for (int i = 1; i <= 11; i++) {
            arrayList.add(i);
        }
    }
}
```

```
private void grow(int minCapacity) {  minCapacity: 11
    // overflow-conscious code
    int oldCapacity = elementData.length;  oldCapacity (slot_2): 10
    int newCapacity = oldCapacity + (oldCapacity >> 1);  newCapacity (slot_3): 15  oldCapacity (slot_2): 10
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);  minCapacity: 11
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);  newCapacity (slot_3): 15
}
```

newCapacity = 15

按照我们上述的分析，我们可以得出结论，数组每重新创建一次，大小就为原数组大小的1.5倍。因为>1操作相当于等于原数值的一半。

● 注意事项

通过上面的源码分析，我们已经知道了ArrayList中所有的底层实现逻辑， 现在我们将谈谈，在我们经知道了，上述底层实现时，怎么在开发时能够更好的使用ArrayList集合类。

● 我们分析源码时知道，ArrayList动态数组的底层实现逻辑是通过创建新数组来实现的。所以我们在发时，如果事先知道存储量很大时，那我们就可以将ArrayList中底层数组的默认初始化大小设置的大点，这样就可以减少创建新数组，所造成的性能损失，进而提高程序的运行效率。我们可以调用ArrayList中构造方法来设置默认初始化的大小。

```
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);
    }
}
```

● 我们在分析源码时还发现，所有的方法中都没有添加任何的同步代码。所以可以说ArrayList集合类是线程安全的。如果在多线程开发时如果想使用此集合类，那我们要特别注意，必须要添加额外的代来保证ArrayList的同步。主要方式就是采用同步代码块。当然后除了此方法外，我们还可以使用下面方法来保证ArrayList的同步。

```
List arrayList = Collections.synchronizedList(new ArrayList());
```

● 我们知道数组的检索数度是非常快的，所以我们在使用时，如果是要处理检索任务时，那么我们把据存储在ArrayList中是很合适的，因为它可以帮助我们快速的查找到我们想要的元素。但在更新时，不一定主要分为两种情况。如果我们更新的是数组中中间的元素，那们处理时性能则会比较差，因为们知道数组中的内存必须是连续的，所以底层处理时，会把这个元素后面的元素依次前移一位，所以造成一些不必要的操作，损失性能。但如果我们要更新的是数组中的最后一个元素时，则ArrayList的理性能则会非常快，因为ArrayList的特性是检索快， 所以会很快查找到该元素，然后将该元素删除但又因为是最后一个元素，所以不会执行前移操作，所以性能较高。