深入 java8 的集合 1：ArrayList 的实现原理

一、概述

一上来，先来看看源码中的这一段注释，我们可以从中提取到一些关键信息：

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)
从这段注释中，我们可以得知 ArrayList 是一个动态数组，实现了 List 接口以及 list 相关的所有方法，它允许所有元素的插入，包括 null。另外，ArrayList 和 Vector 除了线程不同步之外，大致相等。

二、属性

```
//默认容量的大小

private static final int DEFAULT_CAPACITY = 10;



//空数组常量

private static final Object[] EMPTY_ELEMENTDATA = {};



//默认的空数组常量

private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMEN
TDATA = {};
```

```
//存放元素的数组，从这可以发现 ArrayList 的底层实现就是一个 Object 数组
transient Object[] elementData;


//数组中包含的元素个数
private int size;


//数组的最大上限
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
```

ArrayList 的属性非常少，就只有这些。其中最重要的莫过于 elementData 了，ArrayList 所有的方法都是建立在 elementData 之上。接下来，我们就来看一下一些主要的方法吧。

三、方法

1、构造方法

```
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);
```

```
    }

}


public ArrayList() {

    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;

}
```

从构造方法中我们可以看见，默认情况下，elementData 是一个大小为 0 的空数组，当我们指定了初始大小的时候，elementData 的初始大小就变成了我们所指定的初始大小了。

2、get 方法

```
public E get(int index) {

    rangeCheck(index);

    return elementData(index);

}


private void rangeCheck(int index) {

    if (index >= size)

        throw new IndexOutOfBoundsException(outOfBoundsMsg
(index));

}


E elementData(int index) {

    return (E) elementData[index];
```

```
}
```

因为 ArrayList 是采用数组结构来存储的，所以它的 get 方法非常简单，先是判断一下有没有越界，之后就可以直接通过数组下标来获取元素了，所以 get 的时间复杂度是 O(1)。

3、add 方法

```
public boolean add(E e) {

    ensureCapacityInternal(size + 1);  // Increments modCou
nt!!

    elementData[size++] = e;

    return true;

}



public void add(int index, E element) {

    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1);  // Increments modCou
nt!!

    //调用一个 native 的复制方法，把 index 位置开始的元素都往后挪一位

    System.arraycopy(elementData, index, elementData, inde
x + 1, size - index);

    elementData[index] = element;

    size++;

}
```

```
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacit
y);
    }
    ensureExplicitCapacity(minCapacity);
}

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

ArrayList 的 add 方法也很好理解，在插入元素之前，它会先检查是否需要扩容，然后再把元素添加到数组中最后一个元素的后面。在 ensureCapacityInternal 方法中，我们可以看见，如果当 elementData 为空数组时，它会使用默认的大小去扩容。所以说，通过无参构造方法来创建 ArrayList 时，它的大小其实是为 0 的，只有在使用到的时候，才会通过 grow 方法去创建一个大小为 10 的数组。

第一个 add 方法的复杂度为 O(1)，虽然有时候会涉及到扩容的操作，但是扩容的次数是非常少的，所以这一部分的时间可以忽略不计。如果使用的是带指定下标的 add 方法，则复杂度为 O(n)，因为涉及到对数组中元素的移动，这一操作是非常耗时的。

4、set 方法

```
public E set(int index, E element) {
    rangeCheck(index);
    E oldValue = elementData(index);
```

```
        elementData[index] = element;

        return oldValue;

    }
```

set 方法的作用是把下标为 index 的元素替换成 element，跟 get 非常类似，所以就不在赘述了，时间复杂度度为 O(1)。

5、remove 方法

```
    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 wo
rk

        return oldValue;

    }
```

remove 方法与 add 带指定下标的方法非常类似，也是调用系统的 arraycopy 方法来移动元素，时间复杂度为 O(n)。

6、grow 方法

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

grow 方法是在数组进行扩容的时候用到的，从中我们可以看见，ArrayList 每次扩容都是扩 1.5 倍，然后调用 Arrays 类的 copyOf 方法，把元素重新拷贝到一个新的数组中去。

7、size 方法

```
public int size() {
    return size;
}
```

size 方法非常简单，它是直接返回 size 的值，也就是返回数组中元素的个数，时间复杂度为 O(1)。这里要注意一下，返回的并不是数组的实际大小。

8、indexOf 方法和 lastIndexOf

```
public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

public int lastIndexOf(Object o) {
    if (o == null) {
        for (int i = size-1; i >= 0; i--)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = size-1; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
```

```
        return -1;

    }
```

indexOf 方法的作用是返回第一个等于给定元素的值的下标。它是通过遍历比较数组中每个元素的值来查找的，所以它的时间复杂度是 O(n)。

lastIndexOf 的原理跟 indexOf 一样，而它仅仅是从后往前找起罢了。

四、Vector

本来是想把 Vector 当成一章来讲的，结果看了一下源码之后发现并没有什么好讲，因为很多方法都跟 ArrayList 一样，只是多加了个 synchronized 来保证线程安全罢了。如果照着 ArrayList 的方式再将一次就显得没意思了，所以只把 Vector 与 ArrayList 的不同点提一下就可以了。

Vector 比 ArrayList 多了一个属性：

```
    protected int capacityIncrement;
```

这个属性是在扩容的时候用到的，它表示每次扩容只扩 capacityIncrement 个空间就足够了。该属性可以通过构造方法给它赋值。先来看一下构造方法：

```
    public Vector(int initialCapacity, int capacityIncrement)
    {

        super();

        if (initialCapacity < 0)

            throw new IllegalArgumentException("Illegal Capacit
    y: "+initialCapacity);

        this.elementData = new Object[initialCapacity];

        this.capacityIncrement = capacityIncrement;

    }



    public Vector(int initialCapacity) {

        this(initialCapacity, 0);
```

```
    }


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

从构造方法中，我们可以看出 Vector 的默认大小也是 10，而且它在初始化的时候就已经创建了数组了，这点跟 ArrayList 不一样。再来看一下 grow 方法：

```
    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);
    }
```

从 grow 方法中我们可以发现，newCapacity 默认情况下是两倍的 oldCapacity，而当指定了 capacityIncrement 的值之后，newCapacity 变成了 oldCapacity+capacityIncrement。

五、总结

1、ArrayList 创建时的大小为 0；当加入第一个元素时，进行第一次扩容时，默认容量大小为 10。
2、ArrayList 每次扩容都以当前数组大小的 1.5 倍去扩容。
3、Vector 创建时的默认大小为 10。

4、Vector 每次扩容都以当前数组大小的 2 倍去扩容。当指定了 capacityIncrement 之后，每次扩容仅在原先基础上增加 capacityIncrement 个单位空间。
5、ArrayList 和 Vector 的 add、get、size 方法的复杂度都为 O(1)，remove 方法的复杂度为 O(n)。
6、ArrayList 是非线程安全的，Vector 是线程安全的。