



[回到首页](#)

## 芋道源码 —— 知识星球

我是一段不羁的公告！

记得给芬芳这 3 个项目加油，添加一个 STAR 噢。

<https://github.com/YunaiV/SpringBoot-Labs>

<https://github.com/YunaiV/onemall>

<https://github.com/YunaiV/ruoyi-vue-pro>

2019-12-01

[JDK](#)

## 精尽 JDK 源码解析 —— 集合（一）数组 ArrayList

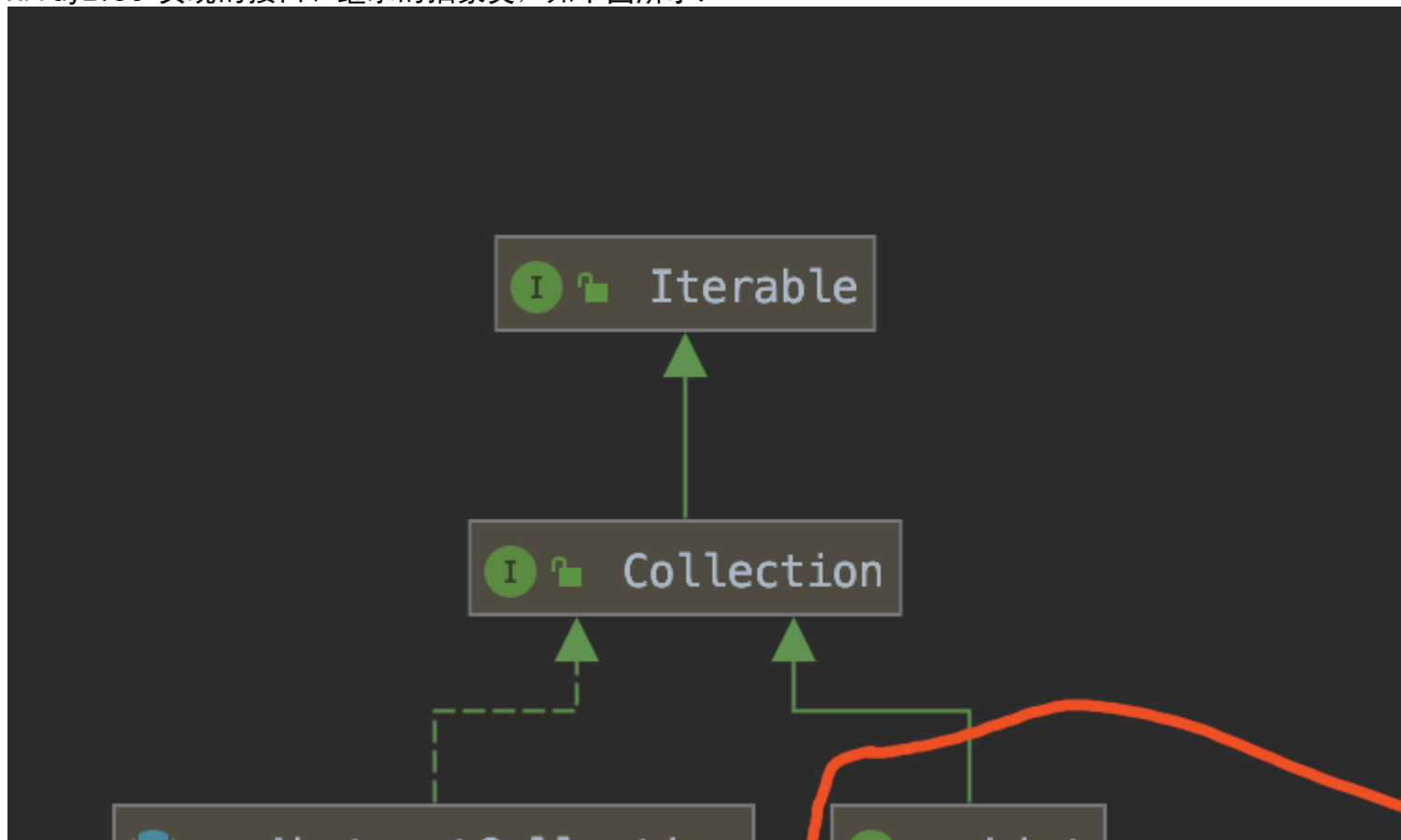
### 1. 概述

ArrayList，基于 `array` 数组实现的，支持自动扩容的动态数组。相比数组来说，因为其支持自动扩容的特性，成为我们日常开发中，最常用的集合类，没有之一。

在前些年，实习或初级工程师的面试，可能最爱问的就是 ArrayList 和 LinkedList 的区别与使用场景。不过貌似，现在问的已经不多了，因为现在信息非常发达，这种常规面试题已经无法区分能力了。当然即使如此，也不妨碍我们拿它开刀，毕竟是咱的“老朋友”。

### 2. 类图

ArrayList 实现的接口、继承的抽象类，如下图所示：



实现了 4 个接口，分别是：

[java.util.List](#) 接口，提供数组的添加、删除、修改、迭代遍历等操作。

[java.util.RandomAccess](#) 接口，表示 ArrayList 支持快速的随机访问。

关于 RandomAccess 标记接口，我们这里先不展开，胖友可以自行阅读 [《RandomAccess 这个空架子有何用？》](#) 文章。

[java.io.Serializable](#) 接口，表示 ArrayList 支持序列化的功能。

关于 Serializable 标记接口，我们这里先不展开，胖友可以自行阅读 [《Java 序列化与反序列化》](#) 文章。

[java.lang.Cloneable](#) 接口，表示 ArrayList 支持克隆。

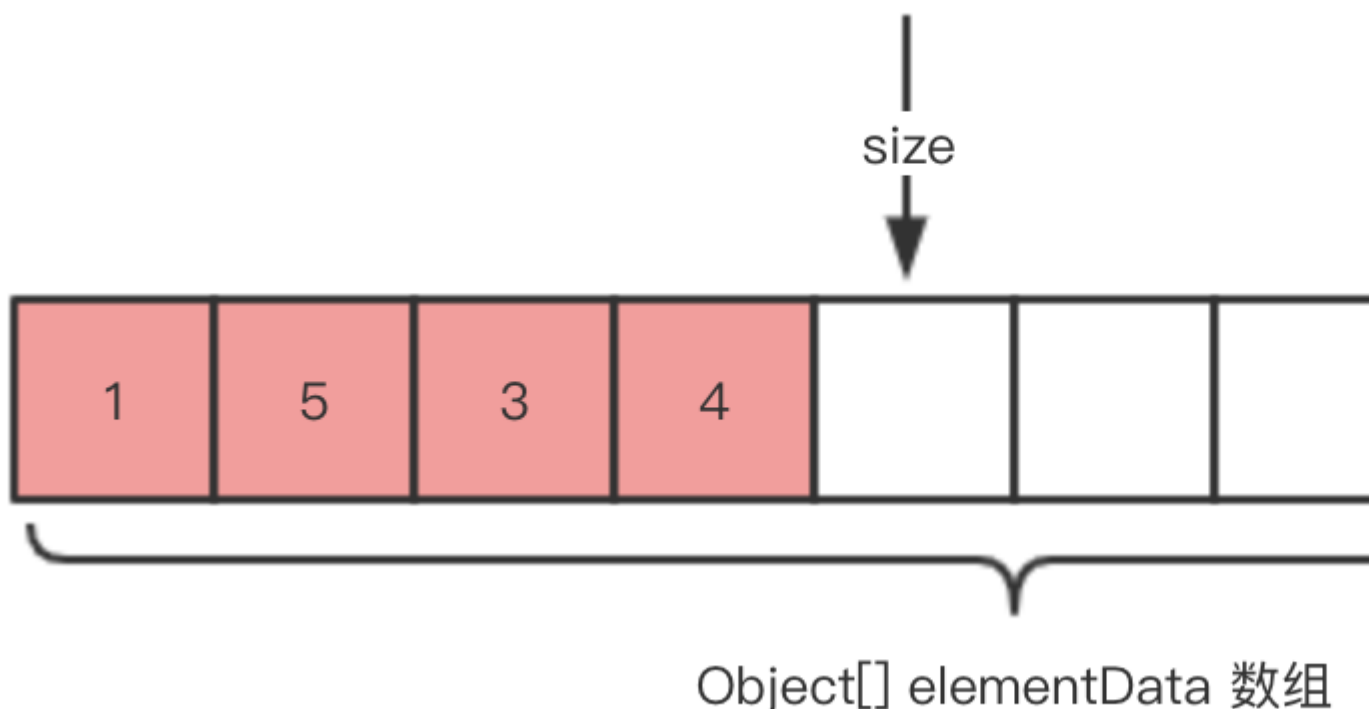
关于 Cloneable 标记接口，我们这里先不展开，胖友可以自行阅读 [《在 Java 中为什么实现了 Cloneable 接口，就能调用 Object 的 clone 方法？》](#) 讨论，特别是 R 大的回答。

继承了 [java.util.AbstractList](#) 抽象类，而 AbstractList 提供了 List 接口的骨架实现，大幅度的减少了实现迭代遍历相关操作的代码。可能这样表述有点抽象，胖友点到 [java.util.AbstractList](#) 抽象类中看看，例如说 `#iterator()`、`#indexOf(Object o)` 等方法。

不过实际上，在下面中我们会看到，ArrayList 大量重写了 AbstractList 提供的方法实现。所以，AbstractList 对于 ArrayList 意义不大，更多的是 AbstractList 其它子类享受了这个福利。

### 3. 属性

ArrayList 的属性很少，仅仅 2 个。如下图所示：



`elementData` 属性：元素数组。其中，图中红色空格代表我们已经添加元素，白色空格代表我们并未使用。

`size` 属性：数组大小。注意，`size` 代表的是 `ArrayList` 已使用 `elementData` 的元素的数量，对于开发者看到的 `#size()` 也是该大小。并且，当我们添加新的元素时，恰好其就是元素添加到 `elementData` 的位置（下标）。当然，我们知道 `ArrayList` 真正的大小是 `elementData` 的大小。

对应代码如下：

```
// ArrayList.java

/**
 * 元素数组。
 *
 * 当添加新的元素时，如果该数组不够，会创建新数组，并将原数组的元素拷贝到新数组。之后，将该变量指向新数组。
 *
 * 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 不使用 private 修复，方便内嵌类的访问。

/**
 * 已使用的数组大小
 *
 * The size of the ArrayList (the number of elements it contains).
 *
 * @serial
 */
private int size;
```

## 4. 构造方法

`ArrayList` 一共有三个构造方法，我们分别来看看。

### ① `#ArrayList(int initialCapacity)`

`#ArrayList(int initialCapacity)` 构造方法，根据传入的初始化容量，创建 `ArrayList` 数组。如果我们在使用时，如果预先指到数组大小，一定要使用该构造方法，可以避免数组扩容提升性能，同时也是合理使用内存。代码如下：

```
// ArrayList.java

/**
 * 共享的空数组对象。
 *
 * 在 {@link #ArrayList(int)} 或 {@link #ArrayList(Collection)} 构造方法中，
 * 如果传入的初始化大小或者集合大小为 0 时，将 {@link #elementData} 指向它。
 *
 * Shared empty array instance used for empty instances.
 */
private static final Object[] EMPTY_ELEMENTDATA = {};

public ArrayList(int initialCapacity) {
```

```

// 初始化容量大于 0 时，创建 Object 数组
if (initialCapacity > 0) {
    this.elementData = new Object[initialCapacity];
// 初始化容量等于 0 时，使用 EMPTY_ELEMENTDATA 对象
} else if (initialCapacity == 0) {
    this.elementData = EMPTY_ELEMENTDATA;
// 初始化容量小于 0 时，抛出 IllegalArgumentException 异常
} else {
    throw new IllegalArgumentException("Illegal Capacity: "+
                                     initialCapacity);
}
}

```

比较特殊的是，如果初始化容量为 0 时，使用 `EMPTY_ELEMENTDATA` 空数组。在添加元素的时候，会进行扩容创建需要的数组。

## ② `#ArrayList(Collection<? extends E> c)`

`#ArrayList(Collection<? extends E> c)` 构造方法，使用传入的 `c` 集合，作为 `ArrayList` 的 `elementData`。代码如下：

```
// ArrayList.java
```

```

public ArrayList(Collection<? extends E> c) {
    // 将 c 转换成 Object 数组
    elementData = c.toArray();
    // 如果数组大小大于 0
    if ((size = elementData.length) != 0) {
        // defend against c.toArray (incorrectly) not returning Object[]
        // (see e.g. https://bugs.openjdk.java.net/browse/JDK-6260652)
        // <X> 如果集合元素不是 Object[] 类型，则会创建新的 Object[] 数组，并将 elementData 赋值到其中，最后赋值给 el
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    // 如果数组大小等于 0，则使用 EMPTY_ELEMENTDATA。
    } else {
        // replace with empty array.
        this.elementData = EMPTY_ELEMENTDATA;
    }
}
}

```

比较让人费解的是，在 `<X>` 处的代码。它是用于解决 [JDK-6260652](https://bugs.openjdk.java.net/browse/JDK-6260652) 的 Bug。它在 JDK9 中被解决，也就是说，JDK8 还会存在该问题。

我们来看一段能够触发 [JDK-6260652](https://bugs.openjdk.java.net/browse/JDK-6260652) 的测试代码，然后分别在 JDK8 和 JDK13 下执行。代码如下：

```
// ArrayListTest.java
```

```

public static void test02() {
    List<Integer> list = Arrays.asList(1, 2, 3);
    Object[] array = list.toArray(); // JDK8 返回 Integer[] 数组，JDK9+ 返回 Object[] 数组。
    System.out.println("array className : " + array.getClass().getSimpleName());

    // 此处，在 JDK8 和 JDK9+ 表现不同，前者会报 ArrayStoreException 异常，后者不会。
    array[0] = new Object();
}

```

}

JDK8 执行如下图所示:

```
// 需要在 JDK8 版本执行, 因为 JDK9 已经修复该问题
public static void test02() {
    List<Integer> list = Arrays.asList(1, 2);
    Object[] array = list.toArray(); // 实际
    System.out.println("array className : "

    // 此处, 在 JDK8 和 JDK9+ 表现不同, 前者会抛
    array[0] = new Object();
}
}
```

ArrayListTest > test02()

ArrayListTest ×

/Library/Java/JavaVirtualMachine/jdk-8.0.144

objc[94611]: Class JavaLaunchHelper is implemented in


.jdk/Contents/Home/jre/lib/instrument.dylib

array className : Integer[]

Exception in thread "main" java.lang.ArrayStoreException

at util.ArrayListTest.test02(ArrayListTest.java:14)

at util.ArrayListTest.main(ArrayListTest.java:14)



JDK13 执行如下图所示:

```
// 需要在 JDK8 版本执行, 因为 JDK9 已经修复该问题
public static void test02() {
```

在 JDK8 中，返回的实际是 `Integer []` 数组，那么我们将 `Object` 对象设置到其中，肯定是会报错的。具体怎么修复的，看 [JDK-6260652](#) 的最末尾一段。

### ③ `#ArrayList()`

无参数构造方法 `#ArrayList()` 构造方法，也是我们使用最多的构造方法。代码如下：

```
// ArrayList.java

/**
 * 默认初始化容量
 *
 * Default initial capacity.
 */
private static final int DEFAULT_CAPACITY = 10;

/**
 * 共享的空数组对象，用于 {@link #ArrayList()} 构造方法。
 *
 * 通过使用该静态变量，和 {@link #EMPTY_ELEMENTDATA} 区分开来，在第一次添加元素时。
 *
 * 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 = {};

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

在我们学习 `ArrayList` 的时候，一直被灌输了一个概念，在未设置初始化容量时，`ArrayList` 默认大小为 10。但是此处，我们可以看到初始化为 `DEFAULTCAPACITY_EMPTY_ELEMENTDATA` 这个空数组。这是为什么呢？`ArrayList` 考虑到节省内存，一些使用场景下仅仅是创建了 `ArrayList` 对象，实际并未使用。所以，`ArrayList` 优化成初始化是个空数组，在首次添加元素时，才真正初始化为容量为 10 的数组。

那么为什么单独声明了 `DEFAULTCAPACITY_EMPTY_ELEMENTDATA` 空数组，而不直接使用 `EMPTY_ELEMENTDATA` 呢？在下文中，我们会看到 `DEFAULTCAPACITY_EMPTY_ELEMENTDATA` 首次扩容为 10，而 `EMPTY_ELEMENTDATA` 按照 1.5 倍扩容从 0 开始而不是 10。两者的起点不同，嘿嘿。

## 5. 添加单个元素

`#add(E e)` 方法，顺序添加单个元素到数组。代码如下：

```
// ArrayList.java

@Override
public boolean add(E e) {
    // <1> 增加数组修改次数
    modCount++;
    // 添加元素
```

```

        add(e, elementData, size);
        // 返回添加成功
        return true;
    }

    private void add(E e, Object[] elementData, int s) {
        // <2> 如果容量不够，进行扩容
        if (s == elementData.length)
            elementData = grow();
        // <3> 设置到末尾
        elementData[s] = e;
        // <4> 数量大小加一
        size = s + 1;
    }

```

<1> 处，增加数组修改次数 `modCount`。在父类 `AbstractList` 上，定义了 `modCount` 属性，用于记录数组修改次数。

<2> 处，如果元素添加的位置就超过末尾（数组下标是从 0 开始，而数组大小比最大下标大 1），说明数组容量不够，需要进行扩容，那么就需要调用 `#grow()` 方法，进行扩容。稍后我们在 [「6. 数组扩容」](#) 小节来讲。

<3> 处，设置到末尾。

<4> 处，数量大小加一。

总体流程上来说，抛开扩容功能，和我们日常往 `[]` 数组里添加元素是一样的。

看懂这个方法后，胖友自己来看看 `#add(int index, E element)` 方法，插入单个元素到指定位置。代码如下：

```

// ArrayList.java

public void add(int index, E element) {
    // 校验位置是否在数组范围内
    rangeCheckForAdd(index);
    // 增加数组修改次数
    modCount++;
    // 如果数组大小不够，进行扩容
    final int s;
    Object[] elementData;
    if ((s = size) == (elementData = this.elementData).length)
        elementData = grow();
    // 将 index + 1 位置开始的元素，进行往后挪
    System.arraycopy(elementData, index,
                     elementData, index + 1,
                     s - index);
    // 设置到指定位置
    elementData[index] = element;
    // 数组大小加一
    size = s + 1;
}

private void rangeCheckForAdd(int index) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}

```

## 6. 数组扩容

`#grow()` 方法，扩容数组，并返回它。整个的扩容过程，首先创建一个新的更大的数组，一般是 1.5 倍大小（为什么说是一般呢，稍后我们会看到，会有一些小细节），然后将原数组复制到新数组中，最后返回新数组。代码如下：

```
// ArrayList.java

private Object[] grow() {
    // <1>
    return grow(size + 1);
}

private Object[] grow(int minCapacity) {
    int oldCapacity = elementData.length;
    // <2> 如果原容量大于 0，或者数组不是 DEFAULTCAPACITY_EMPTY_ELEMENTDATA 时，计算新的数组大小，并创建扩容
    if (oldCapacity > 0 || elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        int newCapacity = ArraysSupport.newLength(oldCapacity,
            minCapacity - oldCapacity, /* minimum growth */
            oldCapacity >> 1           /* preferred growth */);
        return elementData = Arrays.copyOf(elementData, newCapacity);
    // <3> 如果是 DEFAULTCAPACITY_EMPTY_ELEMENTDATA 数组，直接创建新的数组即可。
    } else {
        return elementData = new Object[Math.max(DEFAULT_CAPACITY, minCapacity)];
    }
}
```

<1> 处，调用 `#grow(int minCapacity)` 方法，要求扩容后至少比原有大 1。因为是最小扩容的要求，实际是允许比它大。

<2> 处，如果原容量大于 0 时，又或者数组不是 `DEFAULTCAPACITY_EMPTY_ELEMENTDATA` 时，则计算新的数组大小，并创建扩容。

- `ArraysSupport#newLength(int oldLength, int minGrowth, int prefGrowth)` 方法，计算新的数组大小。简单来说，结果就是 `Math.max(minGrowth, prefGrowth) + oldLength`，按照 `minGrowth` 和 `prefGrowth` 取大的。
- 一般情况下，从 `oldCapacity >> 1` 可以看处，是 1.5 倍扩容。但是会有两个特殊情况：  
1) 初始化数组要求大小为 0 的时候，`0 >> 1` 时（`>> 1` 为右移操作，相当于除以 2）还是 0，此时使用 `minCapacity` 传入的 1。  
2) 在下文中，我们会看到添加多个元素，此时传入的 `minCapacity` 不再仅仅加 1，而是扩容到 `elementData` 数组恰好可以添加下多个元素，而该数量可能会超过当前 `ArrayList` 0.5 倍的容量。

<3> 处，如果是 `DEFAULTCAPACITY_EMPTY_ELEMENTDATA` 数组，直接创建新的数组即可。思考下，如果无参构造方法使用 `EMPTY_ELEMENTDATA` 的话，无法实现该效果了。

既然有数组扩容方法，那么是否有缩容方法呢？在 `#trimToSize()` 方法中，会创建大小恰好够用的新数组，并将原数组复制到其中。代码如下：

```
// ArrayList.java

public void trimToSize() {
    // 增加修改次数
    modCount++;
    // 如果有多余的空间，则进行缩容
    if (size < elementData.length) {
        elementData = (size == 0)

```



```

        ? EMPTY_ELEMENTDATA // 大小为 0 时，直接使用 EMPTY_ELEMENTDATA
        : Arrays.copyOf(elementData, size); // 大小大于 0，则创建大小为 size 的新数组，将原数组复制到其中。
    }
}

```

同时，提供 `#ensureCapacity(int minCapacity)` 方法，保证 `elementData` 数组容量至少有 `minCapacity`。代码如下：

```

// ArrayList.java

public void ensureCapacity(int minCapacity) {
    if (minCapacity > elementData.length // 如果 minCapacity 大于数组的容量
        && !(elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA
            && minCapacity <= DEFAULT_CAPACITY)) { // 如果 elementData 是 DEFAULTCAPACITY_EMPTY_ELEMENTDATA 的时候，
                                                    // 需要最低 minCapacity 容量大于 DEFAULT_CAPACITY，因为实际上容量

        // 数组修改次数加一
        modCount++;
        // 扩容
        grow(minCapacity);
    }
}

```

比较简单，我们可以将这个理解成主动扩容。

## 7. 添加多个元素

`#addAll(Collection<? extends E> c)` 方法，批量添加多个元素。在我们明确知道会添加多个元素时，推荐使用该方法而不是添加单个元素，避免可能多次扩容。代码如下：

```

// ArrayList.java

public boolean addAll(Collection<? extends E> c) {
    // 转成 a 数组
    Object[] a = c.toArray();
    // 增加修改次数
    modCount++;
    // 如果 a 数组大小为 0，返回 ArrayList 数组无变化
    int numNew = a.length;
    if (numNew == 0)
        return false;
    // <1> 如果 elementData 剩余的空间不够，则进行扩容。要求扩容的大小，至于能够装下 a 数组。
    Object[] elementData;
    final int s;
    if (numNew > (elementData = this.elementData).length - (s = size))
        elementData = grow(s + numNew);
    // <2> 将 a 复制到 elementData 从 s 开始位置
    System.arraycopy(a, 0, elementData, s, numNew);
    // 数组大小加 numNew
    size = s + numNew;
    return true;
}

```

<1> 处，如果 `elementData` 剩余的空间不足，则进行扩容。要求扩容的大小，至于能够装下 `a`

数组。当然，在 [「6. 数组扩容」](#) 的小节，我们已经看到，如果要求扩容的空间太小，则扩容 1.5 倍。

<2> 处，将 `a` 复制到 `elementData` 从 `s` 开始位置。

总的看下来，就是 `#add(E e)` 方法的批量版本，优势就正如我们在本节开头说的，避免可能多次扩容。

看懂这个方法后，胖友自己来看看 `#addAll(int index, Collection<? extends E> c)` 方法，从指定位置开始插入多个元素。代码如下：

```
// ArrayList.java

public boolean addAll(int index, Collection<? extends E> c) {
    // 校验位置是否在数组范围内
    rangeCheckForAdd(index);

    // 转成 a 数组
    Object[] a = c.toArray();
    // 增加数组修改次数
    modCount++;
    // 如果 a 数组大小为 0，返回 ArrayList 数组无变化
    int numNew = a.length;
    if (numNew == 0)
        return false;
    // 如果 elementData 剩余的空间不够，则进行扩容。要求扩容的大小，至于能够装下 a 数组。
    Object[] elementData;
    final int s;
    if (numNew > (elementData = this.elementData).length - (s = size))
        elementData = grow(s + numNew);

    // 【差异点】如果 index 开始的位置已经被占用，将它们后移
    int numMoved = s - index;
    if (numMoved > 0)
        System.arraycopy(elementData, index,
                           elementData, index + numNew,
                           numMoved);

    // 将 a 复制到 elementData 从 s 开始位置
    System.arraycopy(a, 0, elementData, index, numNew);
    // 数组大小加 numNew
    size = s + numNew;
    return true;
}
```

重点看【差异点】部分。

## 8. 移除单个元素

`#remove(int index)` 方法，移除指定位置的元素，并返回该位置的原元素。代码如下：

```
// ArrayList.java

public E remove(int index) {
    // 校验 index 不要超过 size
    Objects.checkIndex(index, size);
```

```

final Object[] es = elementData;

// 记录该位置的原值
@SuppressWarnings("unchecked") E oldValue = (E) es[index];
// <X>快速移除
fastRemove(es, index);

// 返回该位置的原值
return oldValue;
}

```

重点是 <X> 处，调用 #fastRemove(Object[] es, int i) 方法，快速移除。代码如下

```

// ArrayList.java

private void fastRemove(Object[] es, int i) {
    // 增加数组修改次数
    modCount++;
    // <Y>如果 i 不是移除最末尾的元素，则将 i + 1 位置的数组往前挪
    final int newSize;
    if ((newSize = size - 1) > i) // -1 的原因是，size 是从 1 开始，而数组下标是从 0 开始。
        System.arraycopy(es, i + 1, es, i, newSize - i);
    // 将新的末尾置为 null，帮助 GC
    es[size = newSize] = null;
}

```

- <Y> 处，看起来比较复杂，胖友按照“如果 i 不是移除最末尾的元素，则将 i + 1 位置的数组往前挪”来理解，就很好懂了。

#remove(Object o) 方法，移除首个为 o 的元素，并返回是否移除到。代码如下：

```

// ArrayList.java

public boolean remove(Object o) {
    final Object[] es = elementData;
    final int size = this.size;
    // <Z> 寻找首个为 o 的位置
    int i = 0;
    found: {
        if (o == null) { // o 为 null 的情况
            for (; i < size; i++)
                if (es[i] == null)
                    break found;
        } else { // o 非 null 的情况
            for (; i < size; i++)
                if (o.equals(es[i]))
                    break found;
        }
        // 如果没找到，返回 false
        return false;
    }
    // 快速移除
    fastRemove(es, i);
    // 找到了，返回 true
    return true;
}

```

和 `#remove(int index)` 差不多，就是在 `<Z>` 处，改成获得首个为 `o` 的位置，之后就调用 `#fastRemove(Object[] es, int i)` 方法，快速移除即可。

## 9. 移除多个元素

我们先来看 `#removeRange(int fromIndex, int toIndex)` 方法，批量移除 `[fromIndex, toIndex)` 的多个元素，注意不包括 `toIndex` 的元素噢。代码如下：

```
// ArrayList.java

protected void removeRange(int fromIndex, int toIndex) {
    // 范围不正确，抛出 IndexOutOfBoundsException 异常
    if (fromIndex > toIndex) {
        throw new IndexOutOfBoundsException(
            outOfBoundsMsg(fromIndex, toIndex));
    }
    // 增加数组修改次数
    modCount++;
    // <X> 移除 [fromIndex, toIndex) 的多个元素
    shiftTailOverGap(elementData, fromIndex, toIndex);
}

private static String outOfBoundsMsg(int fromIndex, int toIndex) {
    return "From Index: " + fromIndex + " > To Index: " + toIndex;
}
```

`<X>` 处，调用 `#shiftTailOverGap(Object[] es, int lo, int hi)` 方法，移除 `[fromIndex, toIndex)` 的多个元素。代码如下：

```
// ArrayList.java

private void shiftTailOverGap(Object[] es, int lo, int hi) {
    // 将 es 从 hi 位置开始的元素，移到 lo 位置开始。
    System.arraycopy(es, hi, es, lo, size - hi);
    // 将从 [size - hi + lo, size) 的元素置空，因为已经被挪到前面了。
    for (int to = size, i = (size - hi - lo); i < to; i++)
        es[i] = null;
}
```

- 和 `#fastRemove(Object[] es, int i)` 方法一样的套路，先挪后置 `null`。
- 有一点要注意，`ArrayList` 特别喜欢把多行代码写成一行。所以，可能会有胖又会有疑惑，貌似这里没有修改数组的大小 `size` 啊？答案在 `i = (size - hi - lo)`，简直到精简到难懂。

`#removeAll(Collection<?> c)` 方法，批量移除指定的多个元素。实现逻辑比较简单，但是看起来会比较绕。简单来说，通过两个变量 `w`（写入位置）和 `r`（读取位置），按照 `r` 顺序遍历数组（`elementData`），如果不存在于指定的多个元素中，则写入到 `elementData` 的 `w` 位置，然后 `w` 位置 + 1，跳到下一个写入位置。通过这样的方式，实现将不存在 `elementData` 覆盖写到 `w` 位置。可能理解起来有点绕，当然看代码也会有点绕绕，嘿嘿。代码如下：

```
// ArrayList.java
```

```
public boolean removeAll(Collection<?> c) {
    return batchRemove(c, false, 0, size);
}
```

调用 `#batchRemove(Collection<?> c, boolean complement, final int from, final int end)` 方法，批量移除指定的多个元素。代码如下：

```
// ArrayList.java

boolean batchRemove(Collection<?> c, boolean complement, final int from, final int end) {
    // 校验 c 非 null 。
    Objects.requireNonNull(c);
    final Object[] es = elementData;
    int r;
    // Optimize for initial run of survivors
    // <1> 优化，顺序遍历 elementData 数组，找到第一个不符合 complement ，然后结束遍历。
    for (r = from; r < end; r++) {
        // <1.1> 遍历到尾，都没不符合条件的，直接返回 false 。
        if (r == end)
            return false;
        // <1.2> 如果包含结果不符合 complement 时，结束
        if (c.contains(es[r]) != complement)
            break;
    }
    // <2> 设置开始写入 w 为 r ，注意不是 r++ 。
    // r++ 后，用于读取下一个位置的元素。因为通过上的优化循环，我们已经 es[r] 是不符合条件的。
    int w = r++;
    try {
        // <3> 继续遍历 elementData 数组，如何符合条件，则进行移除
        for (Object e; r < end; r++)
            if (c.contains(e = es[r]) == complement) // 判断符合条件
                es[w++] = e; // 移除的方式，通过将当前值 e 写入到 w 位置，然后 w 跳到下一个位置。
    } catch (Throwable ex) {
        // Preserve behavioral compatibility with AbstractCollection,
        // even if c.contains() throws.
        // <4> 如果 contains 方法发生异常，则将 es 从 r 位置的数据写入到 es 从 w 开始的位置
        System.arraycopy(es, r, es, w, end - r);
        w += end - r;
        // 继续抛出异常
        throw ex;
    } finally { // <5>
        // 增加数组修改次数
        modCount += end - w;
        // 将数组 [w, end) 位置赋值为 null 。
        shiftTailOverGap(es, w, end);
    }
    return true;
}
```

- 不要慌，我们先一起看下每一小块的逻辑。然后，胖友自己调试下，妥妥的就明白了。
- complement 参数，翻译过来是“补足”的意思。怎么理解呢？表示如果 elementData 元素在 c 集合中时，是否保留。
  - 如果 complement 为 false 时，表示在集合中，就不保留，这显然符合 `#removeAll(Collection<?> c)` 方法要移除的意图。
  - 如果 complement 为 true 时，表示在集合中，就暴露，这符合我们后面会看到的 `#retainAll(Collection<?> c)` 方法要求交集的意图。
- <1>

处，首先我们要知道这是一个基于 Optimize 优化的目的。我们是希望先判断是否 elementData 没有任何一个符合 c 的，这样就无需进行执行对应的移除逻辑。但是，我们又希望能够避免重复遍历，于是就有了这样一块的逻辑。总的来说，这块逻辑的目的是，优化，顺序遍历 elementData 数组，找到第一个不符合 complement，然后结束遍历。

- <1.1> 处，遍历到尾，都没不符合条件的，直接返回 false。也就是说，丫根就不需要进行移除的逻辑。
- <1.2> 处，如果包含结果不符合 complement 时，结束循环。可能有点难理解，我们来举个例子。假设 elementData 是 [1, 2, 3, 1] 时，c 是 [2] 时，那么在遍历第 0 个元素 1 时，则 `c.contains(es[r]) != complement => false != false` 不符合，所以继续缓存；然后，在遍历第 1 个元素 2 时，`c.contains(es[r]) != complement => true != false` 符合，所以结束循环。此时，我们便找到了第一个需要移除的元素的位置。当然，移除不是在这里执行，我们继续往下看。 淡定~
- <2> 处，设置开始写入 w 为 r，注意不是 r++。这样，我们后续在循环 elementData 数组，就会从 w 开始写入。并且此时，r 也跳到了下一个位置，这样间接我们可以发现，w 位置的元素已经被“跳过”了。
- <3> 处，继续遍历 elementData 数组，如何符合条件，则进行移除。可能有点难理解，我们继续上述例子。遍历第 2 个元素 3 时候，`c.contains(es[r]) == complement => false == false` 符合，所以将 3 写入到 w 位置，同时 w 指向下一个位置；遍历第三个元素 1 时候，`c.contains(es[r]) == complement => true == false` 不符合，所以不进行任何操作。
- <4> 处，如果 contains 方法发生异常，则将 es 从 r 位置的数据写入到 es 从 w 开始的位置。这样，保证我们剩余未遍历到的元素，能够挪到从 w 开始的位置，避免多出来一些元素。
- <5> 处，是不是很熟悉，将数组 [w, end) 位置赋值为 null。
- 还是那句话，如果觉得绕，多调试，可以手绘点图，辅助理解下哈。

#retainAll(Collection<?> c) 方法，求 elementData 数组和指定多个元素的交集。简单来说，恰好和 #removeAll(Collection<?> c) 相反，移除不在 c 中的元素。代码如下：

```
// ArrayList.java

public boolean retainAll(Collection<?> c) {
    return batchRemove(c, true, 0, size);
}
```

试着按照芳芳上面解释的，自己走一波。

## 10. 查找单个元素

#indexOf(Object o) 方法，查找首个为指定元素的位置。代码如下：

```
// ArrayList.java

public int indexOf(Object o) {
    return indexOfRange(o, 0, size);
}

int indexOfRange(Object o, int start, int end) {
    Object[] es = elementData;
    // o 为 null 的情况
    if (o == null) {
        for (int i = start; i < end; i++) {
            if (es[i] == null) {
```

```

        return i;
    }
}
// o 非 null 的情况
} else {
    for (int i = start; i < end; i++) {
        if (o.equals(es[i])) {
            return i;
        }
    }
}
// 找不到, 返回 -1
return -1;
}

```

而 `#contains(Object o)` 方法, 就是基于该方法实现。代码如下:

```

// ArrayList.java

public boolean contains(Object o) {
    return indexOf(o) >= 0;
}

```

有时我们需要查找最后一个为指定元素的位置, 所以会使用到 `#lastIndexOf(Object o)` 方法。代码如下:

```

// ArrayList.java

public int lastIndexOf(Object o) {
    return lastIndexOfRange(o, 0, size);
}

int lastIndexOfRange(Object o, int start, int end) {
    Object[] es = elementData;
    // o 为 null 的情况
    if (o == null) {
        for (int i = end - 1; i >= start; i--) { // 倒序
            if (es[i] == null) {
                return i;
            }
        }
    }
    // o 非 null 的情况
    } else {
        for (int i = end - 1; i >= start; i--) { // 倒序
            if (o.equals(es[i])) {
                return i;
            }
        }
    }
    // 找不到, 返回 -1
    return -1;
}

```

## 11. 获得指定位置的元素

`#get(int index)` 方法，获得指定位置的元素。代码如下：

```
// ArrayList.java

public E get(int index) {
    // 校验 index 不要超过 size
    Objects.checkIndex(index, size);
    // 获得 index 位置的元素
    return elementData(index);
}

E elementData(int index) {
    return (E) elementData[index];
}
```

随机访问 `index` 位置的元素，时间复杂度为  $O(1)$ 。

## 12. 设置指定位置的元素

`#set(int index, E element)` 方法，设置指定位置的元素。代码如下：

```
// ArrayList.java

public E set(int index, E element) {
    // 校验 index 不要超过 size
    Objects.checkIndex(index, size);
    // 获得 index 位置的原元素
    E oldValue = elementData(index);
    // 修改 index 位置为新元素
    elementData[index] = element;
    // 返回 index 位置的原元素
    return oldValue;
}
```

## 13. 转换成数组

`#toArray()` 方法，将 `ArrayList` 转换成 `[]` 数组。代码如下：

```
// ArrayList.java

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

// Arrays.java

public static <T> T[] copyOf(T[] original, int newLength) {
    return (T[]) copyOf(original, newLength, original.getClass());
}
```



```
}
```

注意，返回的是 `Object[]` 类型噢。

实际场景下，我们可能想要指定 `T` 泛型的数组，那么我们就需要使用到 `#toArray(T[] a)` 方法。代码如下：

```
// ArrayList.java

public <T> T[] toArray(T[] a) {
    // <1> 如果传入的数组小于 size 大小，则直接复制一个新数组返回
    if (a.length < size)
        // Make a new array of a's runtime type, but my contents:
        return (T[]) Arrays.copyOf(elementData, size, a.getClass());
    // <2> 将 elementData 复制到 a 中
    System.arraycopy(elementData, 0, a, 0, size);
    // <2.1> 如果传入的数组大于 size 大小，则将 size 赋值为 null
    if (a.length > size)
        a[size] = null;
    // <2.2> 返回 a
    return a;
}
```

分成 2 个情况，根据传入的 `a` 数组是否足够大。

<1> 处，如果传入的数组小于 `size` 大小，则直接复制一个新数组返回。一般情况下，我们不会这么干。

<2> 处，将 `elementData` 复制到 `a` 中。

- <2.1> 处，如果传入的数组大于 `size` 大小，则将 `size` 位置赋值为 `null`。额，有点没搞懂这个有啥目的。

- <2.2> 处，返回传入的 `a`。很稳。

考虑到 <1> 处，可能会返回一个新数组，所以即使 <2> 返回的就是 `a` 数组，最好使用还是按照 `a = list.toArray(a)`。

## 14. 求哈希值

`#hashCode()` 方法，求 `ArrayList` 的哈希值。代码如下：

```
// ArrayList.java

public int hashCode() {
    // 获得当前的数组修改次数
    int expectedModCount = modCount;
    // 计算哈希值
    int hash = hashCodeRange(0, size);
    // 如果修改次数发生改变，则抛出 ConcurrentModificationException 异常
    checkForComodification(expectedModCount);
    return hash;
}

int hashCodeRange(int from, int to) {
    final Object[] es = elementData;
    // 如果 to 超过大小，则抛出 ConcurrentModificationException 异常
```

```

        if (to > es.length) {
            throw new ConcurrentModificationException();
        }
        // 遍历每个元素，* 31 求哈希。
        int hashCode = 1;
        for (int i = from; i < to; i++) {
            Object e = es[i];
            hashCode = 31 * hashCode + (e == null ? 0 : e.hashCode());
        }
        return hashCode;
    }
}

```

可能胖友会好奇，为什么使用 31 作为乘子呢？可以看看 [《科普：为什么 String hashCode 方法选择数字 31 作为乘子》](#)。

## 15. 判断相等

`#equals(Object o)` 方法，判断是否相等。代码如下：

```

// ArrayList.java

public boolean equals(Object o) {
    // 如果是自己，直接返回相等
    if (o == this) {
        return true;
    }

    // 如果不为 List 类型，直接不相等
    if (!(o instanceof List)) {
        return false;
    }

    // 获得当前的数组修改次数
    final int expectedModCount = modCount;
    // ArrayList can be subclassed and given arbitrary behavior, but we can
    // still deal with the common case where o is ArrayList precisely
    // <X> 根据不同类型，调用不同比对的方法。主要考虑 ArrayList 可以直接使用其 elementData 属性，性能更优。
    boolean equal = (o.getClass() == ArrayList.class)
        ? equalsArrayList((ArrayList<?>) o)
        : equalsRange((List<?>) o, 0, size);

    // 如果修改次数发生改变，则抛出 ConcurrentModificationException 异常
    checkForComodification(expectedModCount);
    return equal;
}

```

可能第一眼让胖友比较费解的是，为什么根据类型是否为 `ArrayList`，调用了两个不同的方法去比对呢？因为普通的 `List`，我们只能使用 `Iterator` 进行迭代，相比 `ArrayList` 的 `elementData` 属性遍历，性能会略低一些。处处是细节哈。

这两个方法的代码如下，已经添加详细注释。代码如下：

```

// ArrayList.java

```

```

boolean equalsRange(List<?> other, int from, int to) {
    // 如果 to 大于 es 大小, 说明说明发生改变, 抛出 ConcurrentModificationException 异常
    final Object[] es = elementData;
    if (to > es.length) {
        throw new ConcurrentModificationException();
    }
    // 通过迭代器遍历 other, 然后逐个元素对比
    var oit = other.iterator();
    for (; from < to; from++) {
        // 如果 oit 没有下一个, 或者元素不相等, 返回 false 不匹配
        if (!oit.hasNext() || !Objects.equals(es[from], oit.next())) {
            return false;
        }
    }
    // 通过 oit 是否遍历完。实现大小是否相等的效果。
    return !oit.hasNext();
}

private boolean equalsArrayList(ArrayList<?> other) {
    // 获得 other 数组修改次数
    final int otherModCount = other.modCount;
    final int s = size;
    boolean equal;
    // 判断数组大小是否相等
    if (equal = (s == other.size)) {
        final Object[] otherEs = other.elementData;
        final Object[] es = elementData;
        // 如果 s 大于 es 或者 otherEs 的长度, 说明发生改变, 抛出 ConcurrentModificationException 异常
        if (s > es.length || s > otherEs.length) {
            throw new ConcurrentModificationException();
        }
        // 遍历, 逐个比较每个元素是否相等
        for (int i = 0; i < s; i++) {
            if (!Objects.equals(es[i], otherEs[i])) {
                equal = false;
                break; // 如果不相等, 则 break
            }
        }
    }
    // 如果 other 修改次数发生改变, 则抛出 ConcurrentModificationException 异常
    other.checkForComodification(otherModCount);
    return equal;
}

```

## 16. 清空数组

`#clear()` 方法, 清空数组。代码如下:

```

// ArrayList.java

public void clear() {
    // 获得当前的数组修改次数
    modCount++;
    // 遍历数组, 倒序设置为 null
    final Object[] es = elementData;

```

```

        for (int to = size, i = size = 0; i < to; i++)
            es[i] = null;
    }

```

## 17. 序列化数组

`#writeObject(java.io.ObjectOutputStream s)` 方法，实现 `ArrayList` 的序列化。代码如下：

```

// ArrayList.java

@java.io.Serial
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    // Write out element count, and any hidden stuff
    // 获得当前的数组修改次数
    int expectedModCount = modCount;

    // <1> 写入非静态属性、非 transient 属性
    s.defaultWriteObject();

    // Write out size as capacity for behavioral compatibility with clone()
    // <2> 写入 size，主要为了与 clone 方法的兼容
    s.writeInt(size);

    // Write out all elements in the proper order.
    // <3> 逐个写入 elementData 数组的元素
    for (int i = 0; i < size; i++) {
        s.writeObject(elementData[i]);
    }

    // 如果 other 修改次数发生改变，则抛出 ConcurrentModificationException 异常
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}

```

<1> 处，调用 `ObjectOutputStream#defaultWriteObject()` 方法，写入非静态属性、非 `transient` 属性。可能有些胖友不了解 Java 的序列化相关的知识，可以看看 [《Serializable 原理》](#) 文章。

<2> 处，写入 `size`，主要为了与 `clone` 方法的兼容。不过芴芴也觉得挺奇怪的，明明在 <1> 处，已经写入了 `size`，这里怎么还来这么一出呢？各种翻查资料，暂时只看到 [《源码分析：ArrayList 的 writeObject 方法中的实现是否多此一举？》](#) 有个讨论。

<3> 吹，逐个写入 `elementData` 元素的数组。我们回过来看下 `elementData` 的定义，它是一个 `transient` 修饰的属性。为什么呢？因为 `elementData` 数组，并不一定是全满的，而可能是扩容的时候有一定的预留，如果直接序列化，会有很多空间的浪费，所以只序列化从 `[0, size)` 的元素，减少空间的占用。

## 18. 反序列化数组

`#readObject(java.io.ObjectInputStream s)` 方法，反序列化数组。代码如下：

```

// ArrayList.java

```

```

@java.io.Serial
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {

    // Read in size, and any hidden stuff
    // 读取非静态属性、非 transient 属性
    s.defaultReadObject();

    // Read in capacity
    // 读取 size，不过忽略不用
    s.readInt(); // ignored

    if (size > 0) {
        // like clone(), allocate array based upon size not capacity
        SharedSecrets.getJavaObjectInputStreamAccess().checkArray(s, Object[].class, size); // 不知道作甚，哈哈哈。
        // 创建 elements 数组
        Object[] elements = new Object[size];

        // Read in all elements in the proper order.
        // 逐个读取
        for (int i = 0; i < size; i++) {
            elements[i] = s.readObject();
        }

        // 赋值给 elementData
        elementData = elements;
    } else if (size == 0) {
        // 如果 size 是 0，则直接使用空数组
        elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new java.io.InvalidObjectException("Invalid size: " + size);
    }
}

```

和序列化的过程，恰好相反（哈哈哈，不然还想咋样），一眼就看的明白。

## 19. 克隆

#clone() 方法，克隆 ArrayList 对象。代码如下：

```

// ArrayList.java

public Object clone() {
    try {
        // 调用父类，进行克隆
        ArrayList<?> v = (ArrayList<?>) super.clone();
        // 拷贝一个新的数组
        v.elementData = Arrays.copyOf(elementData, size);
        // 设置数组修改次数为 0
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError(e);
    }
}

```

```
}
```

注意，`elementData` 是重新拷贝出来的新的数组，避免和原数组共享。

## 20. 创建子数组

`#subList(int fromIndex, int toIndex)` 方法，创建 `ArrayList` 的子数组。代码如下：

```
// ArrayList.java

public List<E> subList(int fromIndex, int toIndex) {
    subListRangeCheck(fromIndex, toIndex, size);
    return new SubList<>(this, fromIndex, toIndex);
}

private static class SubList<E> extends AbstractList<E> implements RandomAccess {

    /**
     * 根 ArrayList
     */
    private final ArrayList<E> root;
    /**
     * 父 SubList
     */
    private final SubList<E> parent;
    /**
     * 起始位置
     */
    private final int offset;
    /**
     * 大小
     */
    private int size;

    // ... 省略代码
}
```

实际使用时，一定要注意，`SubList` 不是一个只读数组，而是和根数组 `root` 共享相同的 `elementData` 数组，只是说限制了 `[fromIndex, toIndex)` 的范围。这块的源码，并不复杂，所以这里也就不展开了。一般情况下，我们也不需要了解它的源码，嘿嘿。

## 21. 创建 Iterator 迭代器

`#iterator()` 方法，创建迭代器。一般情况下，我们使用迭代器遍历 `ArrayList`、`LinkedList` 等等 `List` 的实现类。代码如下：

```
// ArrayList.java

public Iterator<E> iterator() {
    return new Itr();
}
```

```
}
```

创建 `Itr` 迭代器。`Itr` 实现 [java.util.Iterator](#) 接口，是 `ArrayList` 的内部类。虽然说 `AbstractList` 也提供了一个 `Itr` 的实现，但是 `ArrayList` 为了更好的性能，所以自己实现了，在其类上也有注释 “An optimized version of `AbstractList.Itr`”。

`Itr` 一共有 3 个属性，如下：

```
// ArrayList.java#Itr

/**
 * 下一个访问元素的位置，从下标 0 开始。
 */
int cursor;          // index of next element to return
/**
 * 上一次访问元素的位置。
 *
 * 1. 初始化为 -1，表示无上一个访问的元素
 * 2. 遍历到下一个元素时，lastRet 会指向当前元素，而 cursor 会指向下一个元素。这样，如果我们要实现 remove 方法，移除当前元素时，lastRet 会指向当前元素，而 cursor 会指向下一个元素。这样，如果我们要实现 remove 方法，移除当前元素时，lastRet 会指向当前元素，而 cursor 会指向下一个元素。这样，如果我们要实现 remove 方法，移除当前元素时，lastRet 会指向当前元素，而 cursor 会指向下一个元素。
 * 3. 移除元素时，设置为 -1，表示最后访问的元素不存在了，都被移除咧。
 */
int lastRet = -1; // index of last element returned; -1 if no such
/**
 * 创建迭代器时，数组修改次数。
 *
 * 在迭代过程中，如果数组发生了变化，会抛出 ConcurrentModificationException 异常。
 */
int expectedModCount = modCount;

// prevent creating a synthetic constructor
Itr() {}
```

每个属性，胖友自己看看注释噢。

下面，让我们来看看 `Itr` 对 `Iterator` 的 4 个实现方法。

`#hasNext()` 方法，判断是否还可以继续迭代。代码如下：

```
// ArrayList.java#Itr

public boolean hasNext() {
    return cursor != size;
}
```

`cursor` 如果等于 `size`，说明已经到数组末尾，无法继续迭代了。

`#next()` 方法，下一个元素。代码如下：

```
// ArrayList.java#Itr

public E next() {
    // 校验是否数组发生了变化
    checkForComodification();
```

```

// 判断如果超过 size 范围, 抛出 NoSuchElementException 异常
int i = cursor; // <1> i 记录当前 cursor 的位置
if (i >= size)
    throw new NoSuchElementException();
// 判断如果超过 elementData 大小, 说明可能被修改了, 抛出 ConcurrentModificationException 异常
Object[] elementData = ArrayList.this.elementData;
if (i >= elementData.length)
    throw new ConcurrentModificationException();
// <2> cursor 指向下一个位置
cursor = i + 1;
// <3> 返回当前位置的元素
return (E) elementData[lastRet = i]; // <4> 此处, 会将 lastRet 指向当前位置
}

final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}

```

<1> 处, 记录当前 cursor 的位置。因为我们当前返回的就是要求 cursor 位置的元素。

<2> 处, cursor 指向下一个位置。

<3> 处, 返回当前位置的元素。同时在 <4> 处, 会将 lastRet 指向当前位置。

#remove() 方法, 移除当前元素。代码如下:

```

// ArrayList.java#ltr

public void remove() {
    // 如果 lastRet 小于 0, 说明没有指向任何元素, 抛出 IllegalStateException 异常
    if (lastRet < 0)
        throw new IllegalStateException();
    // 校验是否数组发生了变化
    checkForComodification();

    try {
        // <1> 移除 lastRet 位置的元素
        ArrayList.this.remove(lastRet);
        // <2> cursor 指向 lastRet 位置, 因为被移了, 所以需要后退下
        cursor = lastRet;
        // <3> lastRet 标记为 -1, 因为当前元素被移除了
        lastRet = -1;
        // <4> 记录新的数组的修改次数
        expectedModCount = modCount;
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
}

```

<1> 处, 调用 #remove(int index) 方法, 移除 lastRet 位置的元素。所以, 如果要注意, 如果移除元素比较前面, 会将后面位置的往前挪, 即复制, 可能比较消耗性能。

<2> 处, cursor 指向 lastRet 位置, 因为被移了, 所以需要后退下。

<3> 处, lastRet 标记为 -1, 因为当前元素被移除了。

<4> 处, 记录新的数组的修改次数。因为此处修改了数组, 如果不修改下, 后续迭代肯定会报错。

#forEachRemaining(Consumer<? super E> action) 方法, 消费剩余未迭代的元素。代码如下:



```
// ArrayList.java#ltr

@Override
public void forEachRemaining(Consumer<? super E> action) {
    // 要求 action 非空
    Objects.requireNonNull(action);
    // 获得当前数组大小
    final int size = ArrayList.this.size;
    // 记录 i 指向 cursor
    int i = cursor;
    if (i < size) {
        // 判断如果超过 elementData 大小, 说明可能被修改了, 抛出 ConcurrentModificationException 异常
        final Object[] es = elementData;
        if (i >= es.length)
            throw new ConcurrentModificationException();
        // 逐个处理
        for (; i < size && modCount == expectedModCount; i++)
            action.accept(elementAt(es, i));
        // update once at end to reduce heap write traffic
        // 更新 cursor 和 lastRet 的指向
        cursor = i;
        lastRet = i - 1;
        // 校验是否数组发生了变化
        checkForComodification();
    }
}
```

比较简单, 胖友自己瞅瞅。貌似平时这个方法用的不是很多。

## 22. 创建 ListIterator 迭代器

芳芳: 可能一些胖友不了解 ListIterator 迭代器, 因为平时使用不多。可以先去看看 [《Java 集合框架之 Iterator 和 ListIterator》](#)。简单来说, ListIterator 是为 List 设计的, 功能更强大的 Iterator 迭代器。

#listIterator(...) 方法, 创建 ListIterator 迭代器。代码如下:

```
// ArrayList.java

public ListIterator<E> listIterator(int index) {
    rangeCheckForAdd(index);
    return new ListItr(index);
}

public ListIterator<E> listIterator() {
    return new ListItr(0);
}
```

创建 ListItr 迭代器。ListItr 实现 [java.util.ListIterator](#) 接口, 是 ArrayList 的内部类。虽然说 AbstractList 也提供了一个 ListItr 的实现, 但是 ArrayList 为了更好的性能, 所以自己实现了, 在其类上也有注释 “An optimized version of AbstractList.ListItr”。

ListItr 直接继承 Itr 类, 无自定义的属性。代码如下:

```
// ArrayList.java#ListItr
```

```
ListItr(int index) {  
    super();  
    cursor = index;  
}
```

可以手动设置指定的位置开始迭代。

因为 ListItr 的实现代码比较简单，我们就不逐个来看了，直接贴加了注释的代码。代码如下：

```
// ArrayList.java#ListItr
```

```
/**
```

```
 * @return 是否有前一个
```

```
 */
```

```
public boolean hasPrevious() {  
    return cursor != 0;  
}
```

```
/**
```

```
 * @return 下一个位置
```

```
 */
```

```
public int nextIndex() {  
    return cursor;  
}
```

```
/**
```

```
 * @return 前一个位置
```

```
 */
```

```
public int previousIndex() {  
    return cursor - 1;  
}
```

```
/**
```

```
 * @return 前一个元素
```

```
 */
```

```
@SuppressWarnings("unchecked")
```

```
public E previous() {
```

```
    // 校验是否数组发生了变化
```

```
    checkForComodification();
```

```
    // 判断如果小于 0，抛出 NoSuchElementException 异常
```

```
    int i = cursor - 1;
```

```
    if (i < 0)
```

```
        throw new NoSuchElementException();
```

```
    // 判断如果超过 elementData 大小，说明可能被修改了，抛出 ConcurrentModificationException 异常
```

```
    Object[] elementData = ArrayList.this.elementData;
```

```
    if (i >= elementData.length)
```

```
        throw new ConcurrentModificationException();
```

```
    // cursor 指向上一个位置
```

```
    cursor = i;
```

```
    // 返回当前位置的元素
```

```
    return (E) elementData[lastRet = i]; // 此处，会将 lastRet 指向当前位置
```

```
}
```

```
/**
```

```
 * 设置当前元素
```

```
 *
```

```

    * @param e 设置的元素
    */
    public void set(E e) {
        // 如果 lastRet 无指向, 抛出 IllegalStateException 异常
        if (lastRet < 0)
            throw new IllegalStateException();
        // 校验是否数组发生了变化
        checkForComodification();

        try {
            // 设置
            ArrayList.this.set(lastRet, e);
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }

    /**
     * 添加元素到当前位置
     *
     * @param e 添加的元素
     */
    public void add(E e) {
        // 校验是否数组发生了变化
        checkForComodification();

        try {
            // 添加元素到当前位置
            int i = cursor;
            ArrayList.this.add(i, e);
            // cursor 指向下一个位置, 因为当前位置添加了新的元素, 所以需要后挪
            cursor = i + 1;
            // lastRet 标记为 -1, 因为当前元素并未访问
            lastRet = -1;
            // 记录新的数组的修改次数
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
}

```

## 666. 彩蛋

咳咳咳, 比想象中的长的多的一篇文章。并且实际上, 我们还有几个 ArrayList 的方法的解析没有写, 如下:

```

#splitIterator()
#removeIf(Predicate<? super E> filter)
#replaceAll(UnaryOperator<E> operator)
#sort(Comparator<? super E> c)
#forEach(Consumer<? super E> action)

```

哈哈, 也是比较简单的方法, 胖友自己可以解决一波的哈。就当, 课后作业?! 嘿嘿。

下面, 我们来对 ArrayList 做一个简单的小结:

ArrayList 是基于 `array` 数组实现的 List 实现类，支持在数组容量不够时，一般按照 1.5 倍自动扩容。同时，它支持手动扩容、手动缩容。

ArrayList 随机访问时间复杂度是  $O(1)$ ，查找指定元素的平均时间复杂度是  $O(n)$ 。

可能胖友对时间复杂度的计算方式不是很了解，可以看看 [《算法复杂度分析（上）：分析算法运行时，时间资源及空间资源的消耗》](#) 和 [《算法复杂度分析（下）：最好、最坏、平均、均摊等时间复杂度概述》](#) 两文。

ArrayList 移除指定位置的元素的最好时间复杂度是  $O(1)$ ，最坏时间复杂度是  $O(n)$ ，平均时间复杂度是  $O(n)$ 。

最好时间复杂度发生在末尾移除的情况。

ArrayList 移除指定元素的时间复杂度是  $O(n)$ 。

因为首先需要进行查询，然后在使用移除指定位置的元素，无论怎么计算，都需要  $O(n)$  的时间复杂度。

ArrayList 添加元素的最好时间复杂度是  $O(1)$ ，最坏时间复杂度是  $O(n)$ ，平均时间复杂度是  $O(n)$ 。

最好时间复杂度发生在末尾添加的情况。

结尾在抛个拓展，在 Redis String 的数据结构，实现方式是类似 Java ArrayList 的方式，感兴趣的胖友可以自己去瞅瞅。

## 文章目录

1. [1. 1. 概述](#)
2. [2. 2. 类图](#)
3. [3. 3. 属性](#)
4. [4. 4. 构造方法](#)
5. [5. 5. 添加单个元素](#)
6. [6. 6. 数组扩容](#)
7. [7. 7. 添加多个元素](#)
8. [8. 8. 移除单个元素](#)
9. [9. 9. 移除多个元素](#)
10. [10. 10. 查找单个元素](#)
11. [11. 11. 获得指定位置的元素](#)
12. [12. 12. 设置指定位置的元素](#)
13. [13. 13. 转换成数组](#)
14. [14. 14. 求哈希值](#)
15. [15. 15. 判断相等](#)
16. [16. 16. 清空数组](#)
17. [17. 17. 序列化数组](#)
18. [18. 18. 反序列化数组](#)
19. [19. 19. 克隆](#)
20. [20. 20. 创建子数组](#)
21. [21. 21. 创建 Iterator 迭代器](#)
22. [22. 22. 创建 ListIterator 迭代器](#)
23. [23. 666. 彩蛋](#)

[返回首页](#)