

集合源码：ArrayList 深入解析

简介

ArrayList 是 java 集合框架中比较常用的数据结构了。继承自 AbstractList，实现了 List 接口。底层基于数组实现容量大小动态变化。允许 null 的存在。同时还实现了 RandomAccess、Cloneable、Serializable 接口，所以 ArrayList 是支持快速访问、复制、序列化的。

成员变量

ArrayList 底层是基于数组来实现容量大小动态变化的。

```
/**
 * The size of the ArrayList (the number of elements it contains).
 */
private int size; // 实际元素个数
transient Object[] elementData;
```

注意：上面的 size 是指 elementData 中实际有多少个元素，而 elementData.length 为集合容量，表示最多可以容纳多少个元素。

默认初始容量大小为 10;

```
/**
 * Default initial capacity.
 */
private static final int DEFAULT_CAPACITY = 10;
```

这个变量是定义在 AbstractList 中的。记录对 List 操作的次数。主要使用是在 Iterator，是防止在迭代的过程中集合被修改。

```
protected transient int modCount = 0;
```

下面两个变量是用在构造函数里面的

```
/**
 * Shared empty array instance used for empty instances.
 */
private static final Object[] EMPTY_ELEMENTDATA = {};
```

```
/**
 * 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 = {};
```

两个空的数组有什么区别呢？We distinguish this from EMPTY_ELEMENTDATA to know how much to inflate when first element is added. 简单来讲就是第一次添加元素时知道该 elementData 从空的构造函数还是有参构造函数被初始化的。以便确认如何扩容。

构造函数

无参构造函数

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

注意：注释是说构造一个容量大小为 10 的空的 list 集合，但构造函数只是给 elementData 赋值了一个空的数组，其实是在第一次添加元素时容量扩大至 10 的。构造一个初始容量大小为 initialCapacity 的 ArrayList

```
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);
    }
}
```

由以上源码可见：当使用无参构造函数时是把 DEFAULTCAPACITY_EMPTY_ELEMENTDATA 赋值给 elementData。当 initialCapacity 为零时则是把 EMPTY_ELEMENTDATA 赋值给 elementData。当 initialCapacity 大于

零时初始化一个大小为 initialCapacity 的 object 数组并赋值给 elementData。
使用指定 Collection 来构造 ArrayList 的构造函数

```
public ArrayList(Collection
    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;
    }
}
```

将 Collection 转化为数组并赋值给 elementData, 把 elementData 中元素的个数赋值给 size。如果 size 不为零, 则判断 elementData 的 class 类型是否为 Object[], 不是的话则做一次转换。如果 size 为零, 则把 EMPTY_ELEMENTDATA 赋值给 elementData, 相当于 new ArrayList(0)。

主要操作方法解析

add 操作

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    ensureExplicitCapacity(minCapacity);
}
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
    // overflow-conscious code
```

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

由此可见：每次添加元素到集合中时都会先确认下集合容量大小。然后将 size 自增 1。ensureCapacityInternal 函数中判断如果 elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA 就取 DEFAULT_CAPACITY 和 minCapacity 的最大值也就是 10。这就是 EMPTY_ELEMENTDATA 与 DEFAULTCAPACITY_EMPTY_ELEMENTDATA 的区别所在。同时也验证了上面的说法：使用无参构造函数时是在第一次添加元素时初始化容量为 10 的。ensureExplicitCapacity 中对 modCount 自增 1，记录操作次数，然后如果 minCapacity 大于 elementData 的长度，则对集合进行扩容。显然第一次添加元素时 elementData 的长度为零。那我们来看看 grow 函数。

```
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 win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

很简单明了的一个函数，默认将扩容至原来容量的 1.5 倍。但是扩容之后也不一定适用，有可能太小，有可能太大。所以才会有下面两个 if 判断。如果 1.5 倍太小的话，则将我们所需的容量大小赋值给 newCapacity，如果 1.5 倍太大或者我们需要的容量太大，那就直接拿 newCapacity = (minCapacity > MAX_ARRAY_SIZE) ? Integer.MAX_VALUE : MAX_ARRAY_SIZE 来扩容。然后将原数组中的数据复制到大小为 newCapacity 的新数组中，并将新数组赋值给 elementData。

```
public void add(int index, E element) {
    rangeCheckForAdd(index);
    ensureCapacityInternal(size + 1); // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1, size - index);
    elementData[index] = element;
    size++;
}

public boolean addAll(Collection {
```

```
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;
}

public boolean addAll(int index, Collection 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;
}
```

有以上源码可知，`add(int index, E element)`，`addAll(Collection c)`，`addAll(int index, Collection c)` 操作是都是先对集合容量检查，以确保不会数组越界。然后通过 `System.arraycopy()` 方法将旧数组元素拷贝至一个新的数组中去。

remove 操作

```
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;
}

public boolean remove(Object o) {
    if (o == null) {
```

```
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) {
                fastRemove(index);
                return true;
            }
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {
                fastRemove(index);
                return true;
            }
    }
    return false;
}

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
}
```

当我们调用 `remove(int index)` 时，首先会检查 `index` 是否合法，然后再判断要删除的元素是否位于数组的最后一个位置。如果 `index` 不是最后一个，就再次调用 `System.arraycopy()` 方法拷贝数组。说白了就是将从 `index + 1` 开始向后所有的元素都向前挪一个位置。然后将数组的最后一个位置空，`size - 1`。如果 `index` 是最后一个元素那么就直接将数组的最后一个位置空，`size - 1` 即可。当我们调用 `remove(Object o)` 时，会把 `o` 分为是否为空来分别处理。然后对数组做遍历，找到第一个与 `o` 对应的下标 `index`，然后调用 `fastRemove` 方法，删除下标为 `index` 的元素。其实仔细观察 `fastRemove(int index)` 方法和 `remove(int index)` 方法基本全部相同。

get 操作

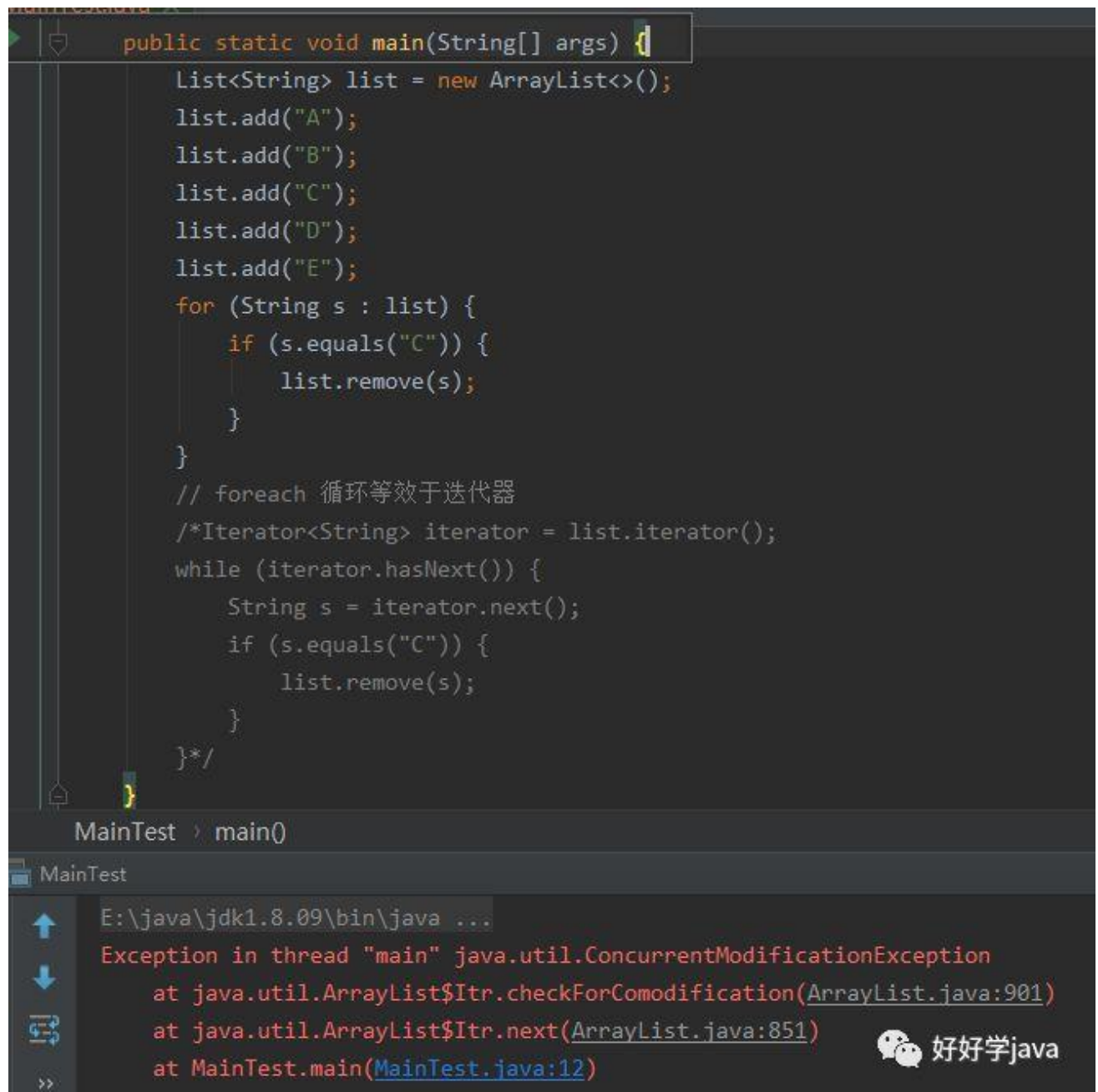
```
public E get(int index) {
    rangeCheck(index);
    return elementData[index];
}
```

由于 `ArrayList` 底层是基于数组实现的，所以获取元素就相当简单了，直接调用数组随

机访问即可。

迭代器 iterator

有使用过集合的都知道，在用 for 遍历集合的时候是不可以对集合进行 remove 操作的，因为 remove 操作会改变集合的大小。从而容易造成结果不准确甚至数组下标越界，更严重者还会抛出 ConcurrentModificationException。



```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("A");
    list.add("B");
    list.add("C");
    list.add("D");
    list.add("E");
    for (String s : list) {
        if (s.equals("C")) {
            list.remove(s);
        }
    }
    // foreach 循环等效于迭代器
    /*Iterator<String> iterator = list.iterator();
    while (iterator.hasNext()) {
        String s = iterator.next();
        if (s.equals("C")) {
            list.remove(s);
        }
    }*/
}
```

MainTest > main()

MainTest

E:\java\jdk1.8.09\bin\java ...

Exception in thread "main" java.util.ConcurrentModificationException
at java.util.ArrayList\$Itr.checkForComodification(ArrayList.java:901)
at java.util.ArrayList\$Itr.next(ArrayList.java:851)
at MainTest.main(MainTest.java:12)

好好学java

foreach 遍历等同于 iterator。为了搞清楚异常原因，我们还必须过一遍源码。

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

原来是直接返回一个 Itr 对象。

```
private class Itr implements Iterator<E> {
    int cursor;          // index of next element to return
    int lastRet = -1;    // index of last element returned; -1 if no such
    int expectedModCount = modCount;
    public boolean hasNext() {
        return cursor != size;
    }
    @SuppressWarnings("unchecked")
    public E next() {
        checkForComodification();
        int i = cursor;
        if (i >= size)
            throw new NoSuchElementException();
        Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new ConcurrentModificationException();
        cursor = i + 1;
        return (E) elementData[lastRet = i];
    }
    public void remove() {
        if (lastRet < 0)
            throw new IllegalStateException();
        checkForComodification();
        try {
            ArrayList.this.remove(lastRet);
            cursor = lastRet;
            lastRet = -1;
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
    final void checkForComodification() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
    }
}
```


}

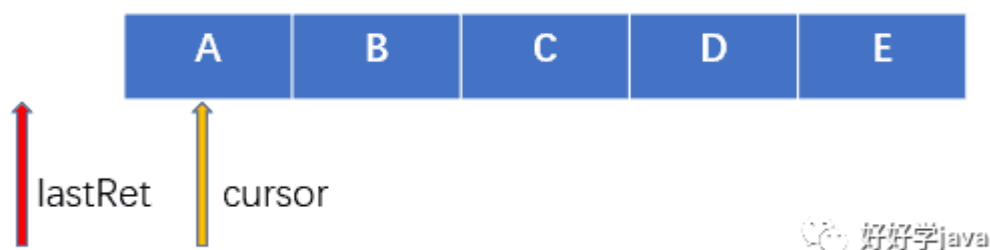
从源码可以看出，ArrayList 定义了一个内部类 Itr 实现了 Iterator 接口。在 Itr 内部有三个成员变量。cursor：代表下一个要访问的元素下标。lastRet：代表上一个要访问的元素下标。expectedModCount：代表对 ArrayList 修改次数的期望值，初始值为 modCount。

Itr 的三个主要函数。

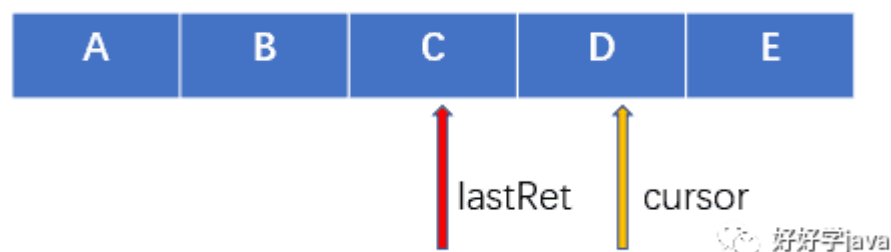
hasNext 实现比较简单，如果下一个元素的下标等于集合的大小，就证明到最后了。

next 方法也不复杂，但很关键。首先判断 expectedModCount 和 modCount 是否相等。然后对 cursor 进行判断，看是否超过集合大小和数组长度。然后将 cursor 赋值给 lastRet，并返回下标为 lastRet 的元素。最后将 cursor 自增 1。开始时，cursor = 0，lastRet = -1；每调用一次 next 方法，cursor 和 lastRet 都会自增 1。

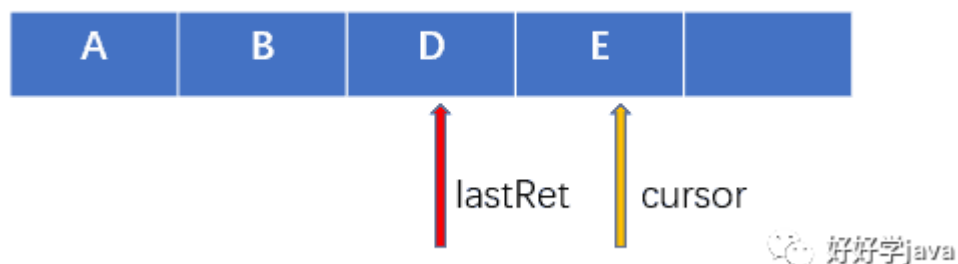
remove 方法首先会判断 lastRet 的值是否小于 0，然后在检查 expectedModCount 和 modCount 是否相等。接下来是关键，直接调用 ArrayList 的 remove 方法删除下标为 lastRet 的元素。然后将 lastRet 赋值给 cursor，将 lastRet 重新赋值为 -1，并将 modCount 重新赋值给 expectedModCount。



下面我们一步一步来分析 Itr 的操作。如图所示，开始时 cursor 指向下标为 0 的元素，lastRet 指向下标为 -1 的元素，也就是 null。每调用一次 next，cursor 和 lastRet 就分别会自增 1。当 next 返回 "C" 时，cursor 和 lastRet 分别为 3 和 2。



此时调用 remove，注意是 ArrayList 的 remove，而不是 Itr 的 remove。会将 D E 两个元素直接往前移动一位，最后一位置空，并且 modCount 会自增 1。从 remove 方法可以看出。



此时 $\text{cursor} = 3$ ， $\text{size} = 4$ ，没有到数组末尾，所以循环继续。来到 `next` 方法，因为上一步的 `remove` 方法对 `modCount` 做了修改，致使 `expectedModCount` 与 `modCount` 不相等，这就是 `ConcurrentModificationException` 异常的原因所在。从例子.png 中也可以看出异常出自 `ArrayList` 中的内部类 `Itr` 中的 `checkForComodification` 方法。

异常的解决：

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String s = iterator.next();
    if (s.equals("C")) {
        iterator.remove();
    }
}
```

好好学java

直接调用 `iterator.remove()` 即可。因为在该方法中增加了 `expectedModCount = modCount` 操作。但是这个 `remove` 方法也有弊端。

- 1、只能进行 `remove` 操作，`add`、`clear` 等 `Itr` 中没有。
- 2、调用 `remove` 之前必须先调用 `next`。因为 `remove` 开始就对 `lastRet` 做了校验。而 `lastRet` 初始化时为 `-1`。
- 3、`next` 之后只可以调用一次 `remove`。因为 `remove` 会将 `lastRet` 重新初始化为 `-1`。

总结

`ArrayList` 底层基于数组实现容量大小动态可变。扩容机制为首先扩容为原始容量的 1.5 倍。如果 1.5 倍太小的话，则将我们所需的容量大小赋值给 `newCapacity`，如果 1.5 倍太大或者我们需要的容量太大，那就直接拿 `newCapacity = (minCapacity > Integer.MAX_VALUE ? Integer.MAX_VALUE : MAX_ARRAY_SIZE)` 来扩容。扩容之后是通过数组的拷贝来确保元素的准确性的，所以尽可能减少扩容操作。`ArrayList` 的最大存储能力：`Integer.MAX_VALUE`。`size` 为集合中存储的元素的个数。`elementData.length` 为数组长度，表示最多可以存储多少个元素。如果需要边遍历边 `remove`，必须使用 `iterator`。

且 remove 之前必须先 next , next 之后只能用一次 remove。

<https://github.com/houwanle>