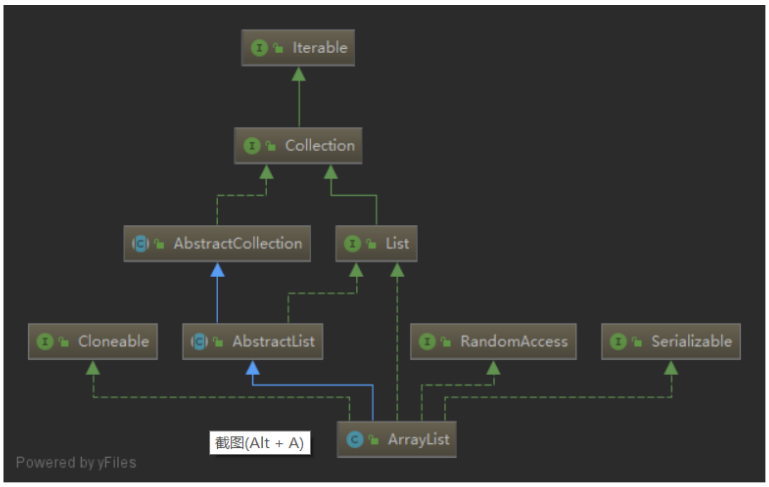
## 简介

ArrayList是一种以数组实现的List，与数组相比，它具有动态扩展的能力，因此也可称之为动态数组。

## 继承体系




ArrayList实现了List, RandomAccess, Cloneable, java.io.Serializable等接口。

ArrayList实现了List，提供了基础的添加、删除、遍历等操作。

ArrayList实现了RandomAccess，提供了随机访问的能力。

ArrayList实现了Cloneable，可以被克隆。

ArrayList实现了Serializable，可以被序列化。

## 源码解析

### 属性

```
1.     /**
2.      * 默认容量
3.      */
4.     private static final int DEFAULT_CAPACITY = 10;
5.
6.     /**
7.      * 空数组，如果传入的容量为0时使用
8.      */
9.     private static final Object[] EMPTY_ELEMENTDATA = {};
10.
11.    /**
12.     * 空数组，传传入容量时使用，添加第一个元素的时候会重新初始为默认容量大小
13.     */
14.    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
15.
16.    /**
17.     * 存储元素的数组
18.     */
19.    transient Object[] elementData; // non-private to simplify nested class access
20.
21.    /**
22.     * 集合中元素的个数
23.     */
24.    private int size;
25.
```

（1）DEFAULT_CAPACITY

默认容量为10，也就是通过new ArrayList()创建时的默认容量。1.8之前，1.8之后先创建0长度的数组，添加第一个元素后再创建10长度数组

（2）EMPTY_ELEMENTDATA

空的数组，这种是通过new ArrayList(0)创建时用的是这个空数组。

（3）DEFAULTCAPACITY_EMPTY_ELEMENTDATA

也是空数组，这种是通过new ArrayList()创建时用的是这个空数组，与EMPTY_ELEMENTDATA的区别是在添加第一个元素时使用这个空数组的会初始化为DEFAULT_CAPACITY（10）个元素。

（4）elementData

真正存放元素的地方，使用transient是为了不序列化这个字段。

至于没有使用private修饰，后面注释是写的“为了简化嵌套类的访问”，但是楼主实测加了private嵌套类一样可以访问。

*private表示是类私有的属性，只要是在这个类内部都可以访问，嵌套类或者内部类也是在类的内部，所以也可以访问类的私有成员。*

（5）size

真正存储元素的个数，而不是elementData数组的长度。

## ArrayList(int initialCapacity)构造方法

传入初始容量，如果大于0就初始化elementData为对应大小，如果等于0就使用EMPTY_ELEMENTDATA空数组，如果小于0抛出异常。

```
1.    public ArrayList(int initialCapacity) {
2.        if (initialCapacity > 0) {
3.            // 如果传入的初始容量大于0，就新建一个数组存储元素
4.            this.elementData = new Object[initialCapacity];
5.        } else if (initialCapacity == 0) {
6.            // 如果传入的初始容量等于0，使用空数组EMPTY_ELEMENTDATA
7.            this.elementData = EMPTY_ELEMENTDATA;
8.        } else {
9.            // 如果传入的初始容量小于0，抛出异常
10.           throw new IllegalArgumentException("Illegal Capacity: " + initialCapacity);
11.       }
12.   }
13.
```

## ArrayList()构造方法

不传初始容量，初始化为DEFAULTCAPACITY_EMPTY_ELEMENTDATA空数组，会在添加第一个元素的时候扩容为默认的大小，即10。

```
1.    public ArrayList() {
2.        // 如果没有传入初始容量，则使用空数组DEFAULTCAPACITY_EMPTY_ELEMENTDATA
3.        // 使用这个数组是在添加第一个元素的时候会扩容到默认大小10
4.        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
5.    }
6.
```

## ArrayList 构造方法

传入集合并初始化elementData，这里会使用拷贝把传入集合的元素拷贝到elementData数组中，如果元素个数为0，则初始化为EMPTY_ELEMENTDATA空数组。

```
    /**
    * 把传入集合的元素初始化到ArrayList中
    */
    public ArrayList(Collection<? extends E> c) {
        // 集合转数组
        elementData = c.toArray();
        if ((size = elementData.length) != 0) {
            // 检查c.toArray()返回的是不是Object[]类型，如果不是，重新拷贝成Object[].class类型
            if (elementData.getClass() != Object[].class)
                elementData = Arrays.copyOf(elementData, size, Object[].class);
        } else {
            // 如果c的空集合，则初始化为空数组EMPTY_ELEMENTDATA
            this.elementData = EMPTY_ELEMENTDATA;
        }
    }

```

为什么 `c.toArray();` 返回的有可能不是Object[]类型呢？请看下面的代码：

```
    public class ArrayTest {
        public static void main(String[] args) {
            Father[] fathers = new Son[]{};
            // 打印结果为class [Lcom.coolcoding.code.Son;
            System.out.println(fathers.getClass());

            List<String> strList = new MyList();
            // 打印结果为class [Ljava.lang.String;
            System.out.println(strList.toArray().getClass());
        }
    }

    class Father {}

    class Son extends Father {}

    class MyList extends ArrayList<String> {
        /**
         * 子类重写父类的方法，返回值可以不一样
         * 但这里只能用数组类型，换成Object就不行
         * 应该算是java本身的bug
         */
        @Override
        public String[] toArray() {
            // 为了方便举例直接写死
            return new String[]{"1", "2", "3"};
        }
    }

```

## add(E e)方法

添加元素到末尾，平均时间复杂度为O(1)。

```
    public boolean add(E e) {
        // 检查是否需要扩容
        ensureCapacityInternal(size + 1);
        // 把元素插入到最后一位
        elementData[size++] = e;
        return true;
    }

    private void ensureCapacityInternal(int minCapacity) {
        ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
    }

    private static int calculateCapacity(Object[] elementData, int minCapacity) {
        // 如果是空数组DEFAULTCAPACITY_EMPTY_ELEMENTDATA，就初始化为默认大小10
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            return Math.max(DEFAULT_CAPACITY, minCapacity);
        }
        return minCapacity;
    }

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

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

```

    private void grow(int minCapacity) {
        int oldCapacity = elementData.length;
        // 新容量为旧容量的1.5倍
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        // 如果新容量发现比需要的容量还小，则以需要的容量为准
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        // 如果新容量已经超过最大容量了，则使用最大容量
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        // 以新容量拷贝出来一个新数组
        elementData = Arrays.copyOf(elementData, newCapacity);
    }

```

（1）检查是否需要扩容；

（2）如果elementData等于DEFAULTCAPACITY_EMPTY_ELEMENTDATA则初始化容量大小为DEFAULT_CAPACITY；

（3）新容量是老容量的1.5倍（oldCapacity + (oldCapacity >> 1)），如果加了这么多容量发现比需要的容量还小，则以需要的容量为准；如果大于最大值，则以最大值为准

（4）创建新容量的数组并把老数组拷贝到新数组；

## add(int index, E element)方法

添加元素到指定位置，平均时间复杂度为O(n)。

```
    public void add(int index, E element) {
        // 检查是否越界
        rangeCheckForAdd(index);
        // 检查是否需要扩容
        ensureCapacityInternal(size + 1);
        // 将inex及其之后的元素往后挪一位，则index位置处就空出来了
        System.arraycopy(elementData, index, elementData, index + 1,
                         size - index);
        // 将元素插入到index的位置
        elementData[index] = element;
        // 大小增1
        size++;
    }

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

```

（1）检查索引是否越界；

（2）检查是否需要扩容；

（3）把插入索引位置后的元素都往后挪一位；

（4）在插入索引位置放置插入的元素；

（5）大小加1；

## addAll 方法

求两个集合的并集。

```
    /**
    * 将集合c中所有元素添加到当前ArrayList中
    */
    public boolean addAll(Collection<? extends E> c) {
        // 将集合c转为数组
        Object[] a = c.toArray();
        int numNew = a.length;
        // 检查是否需要扩容
        ensureCapacityInternal(size + numNew);
        // 将c中元素全部拷贝到数组的最后
        System.arraycopy(a, 0, elementData, size, numNew);
        // 大小增加c的大小
        size += numNew;
        // 如果c不为空就返回true，否则返回false
        return numNew != 0;
    }

```

（1）拷贝c中的元素到数组a中；

（2）检查是否需要扩容；

（3）把数组a中的元素拷贝到elementData的尾部；

## get(int index)方法

获取指定索引位置的元素，时间复杂度为O(1)。

```
    public E get(int index) {
        // 检查是否越界
        rangeCheck(index);
        // 返回数组index位置的元素
        return elementData(index);
    }

    private void rangeCheck(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }

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

```

（1）检查索引是否越界，这里只检查是否越上界，如果越上界抛出IndexOutOfBoundsException异常，如果越下界抛出的是ArrayIndexOutOfBoundsException异常。

（2）返回索引位置处的元素；

## remove(int index)方法

删除指定索引位置的元素，时间复杂度为O(n)。

```
    public E remove(int index) {
        // 检查是否越界
        rangeCheck(index);

        modCount++;
        // 获取index位置的元素
        E oldValue = elementData(index);

        // 如果index不是最后一位，则将index之后的元素往前挪一位
        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index, numMoved);

        // 将最后一个元素删除，帮助GC
        elementData[--size] = null; // clear to let GC do its work

        // 返回旧值
        return oldValue;
    }

```

（1）检查索引是否越界；

（2）获取指定索引位置的元素；

（3）如果删除的不是最后一位，则其它元素往前移一位；

（4）将最后一位置为null，方便GC回收；

（5）返回删除的元素。

*可以看到，ArrayList删除元素的时候并没有缩容。*

## remove(Object o)方法

删除指定元素值的元素，时间复杂度为O(n)。

```
1.     public boolean remove(Object o) {
2.         if (o == null) {
3.             // 遍历整个数组，找到元素第一次出现的位置，并将其快速删除
4.             for (int index = 0; index < size; index++)
5.                 // 如果要删除的元素为null，则以null进行比较，使用==
6.                 if (elementData[index] == null) {
7.                     fastRemove(index);
8.                     return true;
9.                 }
10.         } else {
11.             // 遍历整个数组，找到元素第一次出现的位置，并将其快速删除
12.             for (int index = 0; index < size; index++)
13.                 // 如果要删除的元素不为null，则进行比较，使用equals()方法
14.                 if (o.equals(elementData[index])) {
15.                     fastRemove(index);
16.                     return true;
17.                 }
18.         }
19.         return false;
20.     }
21.
22.     private void fastRemove(int index) {
23.         // 少了一个越界的检查，遍历后的，不会越界
24.         modCount++;
25.         // 如果index不是最后一位，则将index之后的元素往前挪一位
26.         int numMoved = size - index - 1;
27.         if (numMoved > 0)
28.             System.arraycopy(elementData, index+1, elementData, index, numMoved);
29.         // 将最后一个元素删除，帮助GC
30.         elementData[--size] = null; // clear to let GC do its work
31.     }
32.
```

（1）找到第一个等于指定元素值的元素；

（2）快速删除；

*fastRemove(int index)相对于remove(int index)少了检查索引越界的操作，可见jdk将性能优化到极致。*

## retainAll方法

求两个集合的交集。

```java
    public boolean retainAll(Collection<?> c) {
        // 集合c不能为null
        Objects.requireNonNull(c);
        // 调用批量删除方法，这时complement传入true，表示删除不包含在c中的元素
        return batchRemove(c, true);
    }

    /**
    * 批量删除元素
    * complement为true表示删除c中不包含的元素
    * complement为false表示删除c中包含的元素
    */
    private boolean batchRemove(Collection<?> c, boolean complement) {
        final Object[] elementData = this.elementData;
        // 使用读写两个指针同时遍历数组
        // 读指针每次自增1，写指针放入元素的时候才加1
        // 这样不需要额外的空间，只需要在原有的数组上操作就可以了
        int r = 0, w = 0;
        boolean modified = false;
        try {
            // 遍历整个数组，如果c中包含该元素，则把该元素放到写指针的位置（以complement为准）
            for (; r < size; r++)
                if (c.contains(elementData[r]) == complement)
                    elementData[w++] = elementData[r];
        } finally {
            // 正常来说r最后是等于size的，除非c.contains()抛出了异常
            if (r != size) {
                // 如果c.contains()抛出了异常，则把未读的元素都拷贝到写指针之后
                System.arraycopy(elementData, r,
                                 elementData, w,
                                 size - r);
                w += size - r;
            }
            if (w != size) {
                // 将写指针之后的元素置为空，帮助GC
                for (int i = w; i < size; i++)
                    elementData[i] = null;
                modCount += size - w;
                // 新大小等于写指针的位置（因为每写一次写指针就加1，所以新大小正好等于写指针的位置）
                size = w;
                modified = true;
            }
        }
        // 有修改返回true
        return modified;
    }

```

（1）遍历elementData数组；

（2）如果元素在c中，则把这个元素添加到elementData数组的w位置并将w位置往后移一位；

（3）遍历完之后，w之前的元素都是两者共有的，w之后（包含）的元素不是两者共有的；

（4）将w之后（包含）的元素置为null，方便GC回收；

## removeAll

求两个集合的单方向差集，只保留当前集合中不在c中的元素，不保留在c中不在当前集体中的元素。

---

```java
    public boolean removeAll(Collection<?> c) {
        // 集合c不能为空
        Objects.requireNonNull(c);
        // 同样调用批量删除方法，这时complement传入false，表示删除包含在c中的元素
        return batchRemove(c, false);
    }

```

与retainAll(Collection<?> c)方法类似，只是这里保留的是不在c中的元素。

System.copyArray()//数组复制
Arrays.copy()//数组复制

modCount++;
批量删除元素用到读写双指针

## 总结

（1）ArrayList内部使用数组存储元素，当数组长度不够时进行扩容，每次加一半的空间，ArrayList不会进行缩容；

（2）ArrayList支持随机访问，通过索引访问元素极快，时间复杂度为O(1)；

（3）ArrayList添加元素到尾部极快，平均时间复杂度为O(1)；

（4）ArrayList添加元素到中间比较慢，因为要搬移元素，平均时间复杂度为O(n)；

（5）ArrayList从尾部删除元素极快，时间复杂度为O(1)；

（6）ArrayList从中间删除元素比较慢，因为要搬移元素，平均时间复杂度为O(n)；

（7）ArrayList支持求并集，调用addAll(Collection<? extends E> c)方法即可；

（8）ArrayList支持求交集，调用retainAll(Collection<? extends E> c)方法即可；

（7）ArrayList支持求单向差集，调用removeAll(Collection<? extends E> c)方法即可；

迭代时删除元素必须使用迭代器的方法删除，否则会出错
原因：迭代器是按位置遍历，然后操作的，假如用List提供的方法删除掉元素，则集合元素个数变了，位置也变了，接下去迭代器操作的集合就不是之前的集合了，会出现异常

modCount变量就是每次集合增、改、删操作就会加1，程序做remove或者next等会先检查expectedModCount与modCount是否相等，如果不能抛异常

获取迭代器是会先拿int expectedModCount = modCount;做next、remove会先检查这两个值是否相等，所以迭代时删除元素必须使用迭代器的方法删除

批量删除，读写指针

## 彩蛋

*elementData设置成了transient，那ArrayList是怎么把元素序列化的呢？*//元素一个一个写入和读取

```java
    private void writeObject(java.io.ObjectOutputStream s)
            throws java.io.IOException{
        // 防止序列化期间有修改
        int expectedModCount = modCount;
        // 写出非transient非static属性（会写出size属性）
        s.defaultWriteObject();

        // 写出元素个数
        s.writeInt(size);

        // 依次写出元素
        for (int i=0; i<size; i++) {
            s.writeObject(elementData[i]);
        }

        // 如果有修改，抛出异常
        if (modCount != expectedModCount) {
            throw new ConcurrentModificationException();
        }
    }

    private void readObject(java.io.ObjectInputStream s)
            throws java.io.IOException, ClassNotFoundException {
        // 声明为空数组
        elementData = EMPTY_ELEMENTDATA;

```

```
        // 读入非transient非static属性（会读取size属性）
        s.defaultReadObject();

        // 读入元素个数，没什么用，只是因为写出的时候写了size属性，读的时候也要按顺序来读
        s.readInt();

        if (size > 0) {
            // 计算容量
            int capacity = calculateCapacity(elementData, size);
            SharedSecrets.getJavaOISAccess().checkArray(s, Object[].class, capacity);
            // 检查是否需要扩容
            ensureCapacityInternal(size);

            Object[] a = elementData;
            // 依次读取元素到数组中
            for (int i=0; i<size; i++) {
                a[i] = s.readObject();
            }
        }
    }

```

查看writeObject()方法可知，先调用s.defaultWriteObject()方法，再把size写入到流中，再把元素一个一个的写入到流中。

一般地，只要实现了Serializable接口即可自动序列化，writeObject()和readObject()是为了自己控制序列化的方式，这两个方法必须声明为private，在java.io.ObjectStreamClass#getPrivateMethod()方法中通过反射获取到writeObject()这个方法。

在ArrayList的writeObject()方法中先调用了s.defaultWriteObject()方法，这个方法是写入非static非transient的属性，在ArrayList中也就是size属性。同样地，在readObject()方法中先调用了s.defaultReadObject()方法解析出了size属性。

elementData定义为transient的优势，自己根据size序列化真实的元素，而不是根据数组的长度序列化元素，减少了空间占用。