Java集合学习手册（6）：Java ArrayList

一、概述

ArrayList可以理解为动态数组，就是Array的复杂版本。与Java中的数组相比，它的容量能动态增长。ArrayList是List接口的可变数组的实现。实现了所有可选列表操作，并允许包括 null 在内的所有元素。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。（此类大致上等同于 Vector 类，除了此类是不同步的。）

每个ArrayList实例都有一个容量，该容量是指用来存储列表元素的数组的大小。它总是至少等于列表的大小。随着向ArrayList中不断添加元素，其容量也自动增长。自动增长会带来数据向新数组的重新拷贝，因此，如果可预知数据量的多少，可在构造ArrayList时指定其容量。在添加大量元素前，应用程序也可以使用ensureCapacity操作来增加ArrayList实例的容量，这可以减少递增式再分配的数量。

注意，此实现不是同步的。如果多个线程同时访问一个ArrayList实例，而其中至少一个线程从结构上修改了列表，那么它必须保持外部同步。（结构上的修改是指任何添加或删除一个或多个元素的操作，或者显式调整底层数组的大小；仅仅设置元素的值不是结构上的修改。）

我们先学习了解其内部的实现原理，才能更好的理解其应用。

二、ArrayList的实现

对于ArrayList而言，它实现List接口、底层使用数组保存所有元素。其操作基本上是对数组的操作。下面我们来分析ArrayList的源代码：

2.1 List接口

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
}
```

ArrayList继承了AbstractList，实现了List。它是一个数组队列，提供了相关的添加、删除、修改、遍历等功能。

ArrayList实现了RandmoAccess接口，即提供了随机访问功能。RandmoAccess是java中用来被

List实现，为List提供快速访问功能的。在ArrayList中，我们即可以通过元素的序号快速获取元素对象；这就是快速随机访问。

ArrayList实现了Cloneable接口，即覆盖了函数clone()，能被克隆。

ArrayList实现java.io.Serializable接口，这意味着ArrayList支持序列化，能通过序列化去传输。

2.2 底层使用数组实现

```
/**
* The array buffer into which the elements of the ArrayList are stored.
* The capacity of the ArrayList is the length of this array buffer.
*/
private transient Object[] elementData;
```

2.3 构造方法

```
/**
 * Constructs an empty list with an initial capacity of ten.
 */
public ArrayList() {
    this(10);
}
/**
 * Constructs an empty list with the specified initial capacity.
 *
 * @param  initialCapacity  the initial capacity of the list
 * @throws IllegalArgumentException if the specified initial capacity
 *         is negative
 */
public ArrayList(int initialCapacity) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    this.elementData = new Object[initialCapacity];
}

/**
 * Constructs a list containing the elements of the specified
 * collection, in the order they are returned by the collection's
 * iterator.
 *
 * @param c the collection whose elements are to be placed into this list
```

```
     * @throws NullPointerException if the specified collection is null
     */
    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        size = elementData.length;
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    }
```

ArrayList提供了三种方式的构造器：

- public ArrayList()：可以构造一个默认初始容量为10的空列表；
- public ArrayList(int initialCapacity)：构造一个指定初始容量的空列表；
- public ArrayList(Collection<? extends E> c)：构造一个包含指定collection的元素的列表，这些元素按照该collection的迭代器返回它们的顺序排列的。

2.4 存储

ArrayList提供了set(int index, E element)、add(E e)、add(int index, E element)、addAll(Collection<? extends E> c)、addAll(int index, Collection<? extends E> c)这些添加元素的方法。下面我们一一讲解：

- set(int index, E element)：该方法首先调用rangeCheck(index)来校验index变量是否超出数组范围，超出则抛出异常。而后，取出原index位置的值，并且将新的element放入Index位置，返回oldValue。

```
    /**
     * Replaces the element at the specified position in this list with
     * the specified element.
     *
     * @param index index of the element to replace
     * @param element element to be stored at the specified position
     * @return the element previously at the specified position
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    // 用指定的元素替代此列表中指定位置上的元素，并返回以前位于该位置上的元素。
    public E set(int index, E element) {
        rangeCheck(index);

        E oldValue = elementData(index);
        elementData[index] = element;
        return oldValue;
    }
```

```
/**
  * Checks if the given index is in range.  If not, throws an appropriate
  * runtime exception.  This method does *not* check if the index is
  * negative: It is always used immediately prior to an array access,
  * which throws an ArrayIndexOutOfBoundsException if index is negative.
  */
  private void rangeCheck(int index) {
    if (index >= size)
    throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
  }
```

- add(E e)：该方法是将指定的元素添加到列表的尾部。当容量不足时，会调用grow方法增长容量。

```
/**
 * Appends the specified element to the end of this list.
 *
 * @param e element to be appended to this list
 * @return <tt>true</tt> (as specified by {@link Collection#add})
 */
 // 将指定的元素添加到此列表的尾部。
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
private void ensureCapacityInternal(int minCapacity) {
    modCount++;
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
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);
}
```

- add(int index, E element)：在index位置插入element。

```
/**
 * Inserts the specified element at the specified position in this
 * list. Shifts the element currently at that position (if any) and
 * any subsequent elements to the right (adds one to their indices).
 *
 * @param index index at which the specified element is to be inserted
 * @param element element to be inserted
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
 // 将指定的元素插入此列表中的指定位置。
// 如果当前位置有元素，则向右移动当前位于该位置的元素以及所有后续元素（将其索引加1）。
public void add(int index, E element) {
    rangeCheckForAdd(index);
    // 如果数组长度不足，将进行扩容。
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    // 将 elementData中从Index位置开始、长度为size-index的元素，
    // 拷贝到从下标为index+1位置开始的新的elementData数组中。
    // 即将当前位于该位置的元素以及所有后续元素右移一个位置。
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

- addAll(Collection<? extends E> c)和addAll(int index, Collection<? extends E> c)：将特定Collection中的元素添加到Arraylist末尾。

```
/**
     * Appends all of the elements in the specified collection to the end of
     * this list, in the order that they are returned by the
     * specified collection's Iterator.  The behavior of this operation is
     * undefined if the specified collection is modified while the operation
     * is in progress.  (This implies that the behavior of this call is
     * undefined if the specified collection is this list, and this
     * list is nonempty.)
     *
     * @param c collection containing elements to be added to this list
     * @return <tt>true</tt> if this list changed as a result of the call
     * @throws NullPointerException if the specified collection is null
     */
     // 按照指定collection的迭代器所返回的元素顺序，将该collection中的所有元素添加到此列表的尾部。
    public boolean addAll(Collection<? extends E> c) {
        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;
    }

    /**
     * Inserts all of the elements in the specified collection into this
     * list, starting at the specified position.  Shifts the element
     * currently at that position (if any) and any subsequent elements to
     * the right (increases their indices).  The new elements will appear
     * in the list in the order that they are returned by the
     * specified collection's iterator.
     *
     * @param index index at which to insert the first element from the
     *              specified collection
     * @param c collection containing elements to be added to this list
     * @return <tt>true</tt> if this list changed as a result of the call
     * @throws IndexOutOfBoundsException {@inheritDoc}
     * @throws NullPointerException if the specified collection is null
     */
     // 从指定的位置开始，将指定collection中的所有元素插入到此列表中。
    public boolean addAll(int index, Collection<? extends E> 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;
    }
```

在ArrayList的存储方法，其核心本质是在数组的某个位置将元素添加进入。但其中又会涉及到关于数组容量不够而增长等因素。

2.5 读取

这个方法就比较简单了，ArrayList能够支持随机访问的原因也是很显然的，因为它内部的数据结构是数组，而数组本身就是支持随机访问。该方法首先会判断输入的index值是否越界，然后将数组的index位置的元素返回即可。

```
/**
* Returns the element at the specified position in this list.
*
* @param  index index of the element to return
* @return the element at the specified position in this list
* @throws IndexOutOfBoundsException {@inheritDoc}
*/
// 返回此列表中指定位置上的元素。
public E get(int index) {
    rangeCheck(index);
    return (E) elementData[index];
}
private void rangeCheck(int index) {
    if (index >= size)
    throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

2.6 删除

ArrayList提供了根据下标或者指定对象两种方式的删除功能。需要注意的是该方法的返回值并不相同，如下：

- 根据下标删除：

```
// 移除此列表中指定位置上的元素。
    /**
     * Removes the element at the specified position in this list.
     * Shifts any subsequent elements to the left (subtracts one from their
     * indices).
     *
     * @param index the index of the element to be removed
     * @return the element that was removed from the list
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    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; // Let gc do its work
```

```
        return oldValue;
    }
```

- 指定对象删除：

```
/**
     * Removes the first occurrence of the specified element from this list,
     * if it is present.  If the list does not contain the element, it is
     * unchanged.  More formally, removes the element with the lowest index
     * <tt>i</tt> such that
     * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>
     * (if such an element exists).  Returns <tt>true</tt> if this list
     * contained the specified element (or equivalently, if this list
     * changed as a result of the call).
     *
     * @param o element to be removed from this list, if present
     * @return <tt>true</tt> if this list contained the specified element
     */
     // 移除此列表中首次出现的指定元素（如果存在）。这是应为ArrayList中允许存放重复的元素
。
    public boolean remove(Object o) {
    // 由于ArrayList中允许存放null，因此下面通过两种情况来分别处理。
        if (o == null) {
            for (int index = 0; index < size; index++)
                if (elementData[index] == null) {
                // 类似remove(int index)，移除列表中指定位置上的元素。
                    fastRemove(index);
                    return true;
                }
        } else {
            for (int index = 0; index < size; index++)
                if (o.equals(elementData[index])) {
                    fastRemove(index);
                    return true;
                }
        }
        return false;
    }
```

注意：从数组中移除元素的操作，也会导致被移除的元素以后的所有元素的向左移动一个位置。

2.7 调整数组容量

从上面介绍的向ArrayList中存储元素的代码中，我们看到，每当向数组中添加元素时，都要去检

查添加后元素的个数是否会超出当前数组的长度，如果超出，数组将会进行扩容，以满足添加数据的需求。数组扩容有两个方法，其中开发者可以通过一个public的方法ensureCapacity(int minCapacity)来增加ArrayList的容量，而在存储元素等操作过程中，如果遇到容量不足，会调用priavte方法private void ensureCapacityInternal(int minCapacity)实现。

```
public void ensureCapacity(int minCapacity) {
    if (minCapacity > 0)
        ensureCapacityInternal(minCapacity);
}

private void ensureCapacityInternal(int minCapacity) {
    modCount++;
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
/**
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
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);
}
```

从上述代码中可以看出，数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长大约是其原容量的1.5倍（从int newCapacity = oldCapacity + (oldCapacity >> 1)这行代码得出）。这种操作的代价是很高的，因此在实际使用时，我们应该尽量避免数组容量的扩张。当我们可预知要保存的元素的多少时，要在构造ArrayList实例时，就指定其容量，以避免数组扩容的发生。或者根据实际需求，通过调用ensureCapacity方法来手动增加ArrayList实例的容量。

ArrayList还给我们提供了将底层数组的容量调整为当前列表保存的实际元素的大小的功能。它可以通过trimToSize方法来实现。代码如下：

```
public void trimToSize() {
    modCount++;
    int oldCapacity = elementData.length;
    if (size < oldCapacity) {
        elementData = Arrays.copyOf(elementData, size);
    }
}
```

2.8 Fail-Fast机制

ArrayList也采用了快速失败的机制，通过记录modCount参数来实现。在面对并发的修改时，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险。关于Fail-Fast的更详细的介绍，在之前HashMap中已经提到。