Java 集合系列（2）： ArrayList 源码深入解析和使用示例

概要

上一章，我们学习了 Collection 的架构。这一章开始，我们对 Collection 的具体实现类进行讲解 ;首先 ,讲解 List ,而 List 中 ArrayList 又最为常用。因此 ,本章我们讲解 ArrayList。先对 ArrayList 有个整体认识，再学习它的源码，最后再通过例子来学习如何使用它。

1. ArrayList 介绍

ArrayList 简介

ArrayList 是一个数组队列，相当于 动态数组。与 Java 中的数组相比，它的容量能动态增长。它继承于 AbstractList ,实现了 List, RandomAccess, Cloneable, java.io.Serializable 这些接口。

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。稍后，我们会比较 List 的“快速随机访问”和“通过 Iterator 迭代器访问”的效率。

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。

和 Vector 不同，ArrayList 中的操作不是线程安全的！所以，建议在单线程中才使用 ArrayList，而在多线程中可以选择 Vector 或者 CopyOnWriteArrayList。

ArrayList 构造函数

```
// 默认构造函数
ArrayList()
// capacity 是 ArrayList 的默认容量大小。当由于增加数据导致容量不足时，容量会添加上一次容量大小的一半。
ArrayList(int capacity)
```

```
// 创建一个包含 collection 的 ArrayList
ArrayList(Collection<? extends E> collection)
```

ArrayList 的 API

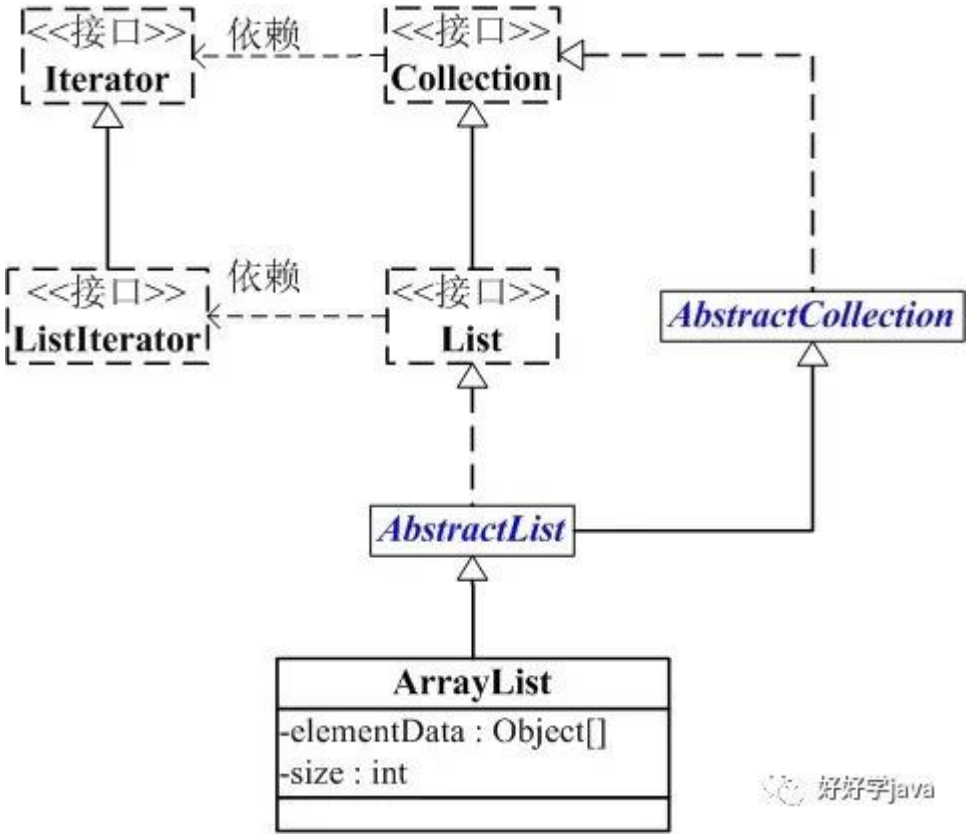
```
// Collection 中定义的 API
boolean            add(E object)
boolean            addAll(Collection<? extends E> collection)
void               clear()
boolean            contains(Object object)
boolean            containsAll(Collection<?> collection)
boolean            equals(Object object)
int                hashCode()
boolean            isEmpty()
Iterator<E>        iterator()
boolean            remove(Object object)
boolean            removeAll(Collection<?> collection)
boolean            retainAll(Collection<?> collection)
int                size()
<T> T[]            toArray(T[] array)
Object[]           toArray()
// AbstractCollection 中定义的 API
void               add(int location, E object)
boolean            addAll(int location, Collection<? extends E> collection)
E                  get(int location)
int                indexOf(Object object)
int                lastIndexOf(Object object)
ListIterator<E>    listIterator(int location)
ListIterator<E>    listIterator()
E                  remove(int location)
E                  set(int location, E object)
List<E>            subList(int start, int end)
// ArrayList 新增的 API
Object             clone()
void               ensureCapacity(int minimumCapacity)
void               trimToSize()
void               removeRange(int fromIndex, int toIndex)
```

2. ArrayList 数据结构

ArrayList 的继承关系

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

ArrayList 与 Collection 关系



ArrayList 包含了两个重要的对象： elementData 和 size。

elementData 是"Object[]类型的数组"，它保存了添加到 ArrayList 中的元素。实际上，elementData 是个动态数组，我们能通过构造函数 ArrayList(int initialCapacity)来执行它的初始容量为 initialCapacity；如果通过不含参数的构造函数 ArrayList()来创建 ArrayList，则 elementData 的容量默认是 10。elementData 数组的大小会根据 ArrayList 容量的增长而动态的增长，具体的增长方式，请参考源码分析中的 ensureCapacity()函数。

size 则是动态数组的实际大小。

3. ArrayList 源码解析(基于 JDK1.6.0_45)

为了更了解 ArrayList 的原理，下面对 ArrayList 源码代码作出分析。ArrayList 是通过数组实现的，源码比较容易理解。

```
package java.util;
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    // 序列版本号
    private static final long serialVersionUID = 8683452581122892189L;
    // 保存 ArrayList 中数据的数组
    private transient Object[] elementData;
    // ArrayList 中实际数据的数量
    private int size;
    // ArrayList 带容量大小的构造函数。
    public ArrayList(int initialCapacity) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        // 新建一个数组
        this.elementData = new Object[initialCapacity];
    }
    // ArrayList 构造函数。默认容量是 10。
    public ArrayList() {
        this(10);
    }
    // 创建一个包含 collection 的 ArrayList
    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);
    }
    // 将当前容量值设为 =实际元素个数
    public void trimToSize() {
        modCount++;
```

```
        int oldCapacity = elementData.length;
        if (size < oldCapacity) {
            elementData = Arrays.copyOf(elementData, size);
        }
    }
    // 确定 ArrarList 的容量。
    // 若 ArrayList 的容量不足以容纳当前的全部元素，设置 新的容量="(原始容量 x3)/2 + 1"
    public void ensureCapacity(int minCapacity) {
        // 将"修改统计数"+1
        modCount++;
        int oldCapacity = elementData.length;
        // 若当前容量不足以容纳当前的元素个数，设置 新的容量="(原始容量 x3)/2 + 1"
        if (minCapacity > oldCapacity) {
            Object oldData[] = elementData;
            int newCapacity = (oldCapacity * 3)/2 + 1;
            if (newCapacity < minCapacity)
                newCapacity = minCapacity;
            elementData = Arrays.copyOf(elementData, newCapacity);
        }
    }
    // 添加元素 e
    public boolean add(E e) {
        // 确定 ArrayList 的容量大小
        ensureCapacity(size + 1);  // Increments modCount!!
        // 添加 e 到 ArrayList 中
        elementData[size++] = e;
        return true;
    }
    // 返回 ArrayList 的实际大小
    public int size() {
        return size;
    }
    // 返回 ArrayList 是否包含 Object(o)
    public boolean contains(Object o) {
        return indexOf(o) >= 0;
    }
    // 返回 ArrayList 是否为空
    public boolean isEmpty() {
        return size == 0;
    }
```

```
// 正向查找，返回元素的索引值
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;
    }
    // 反向查找，返回元素的索引值
    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;
}
// 反向查找(从数组末尾向开始查找)，返回元素(o)的索引值
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;
}
// 返回 ArrayList 的 Object 数组
```

```
    public Object[] toArray() {
        return Arrays.copyOf(elementData, size);
    }
    // 返回 ArrayList 的模板数组。所谓模板数组，即可以将 T 设为任意的数据类型
    public <T> T[] toArray(T[] a) {
        // 若数组 a 的大小 < ArrayList 的元素个数；
        // 则新建一个 T[]数组，数组大小是“ArrayList 的元素个数”，并将“ArrayList”全部拷贝到新数组中
        if (a.length < size)
            return (T[]) Arrays.copyOf(elementData, size, a.getClass());
        // 若数组 a 的大小 >= ArrayList 的元素个数；
        // 则将 ArrayList 的全部元素都拷贝到数组 a 中。
        System.arraycopy(elementData, 0, a, 0, size);
        if (a.length > size)
            a[size] = null;
        return a;
    }
    // 获取 index 位置的元素值
    public E get(int index) {
        RangeCheck(index);
        return (E) elementData[index];
    }
    // 设置 index 位置的值为 element
    public E set(int index, E element) {
        RangeCheck(index);
        E oldValue = (E) elementData[index];
        elementData[index] = element;
        return oldValue;
    }
    // 将 e 添加到 ArrayList 中
    public boolean add(E e) {
        ensureCapacity(size + 1);  // Increments modCount!!
        elementData[size++] = e;
        return true;
    }
    // 将 e 添加到 ArrayList 的指定位置
    public void add(int index, E element) {
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException(
            "Index: "+index+", Size: "+size);
```

```
    ensureCapacity(size+1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
         size - index);
    elementData[index] = element;
    size++;
}
// 删除 ArrayList 指定位置的元素
public E remove(int index) {
    RangeCheck(index);
    modCount++;
    E oldValue = (E) 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;
}
// 删除 ArrayList 的指定元素
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;
}
// 快速删除第 index 个元素
private void fastRemove(int index) {
    modCount++;
    int numMoved = size - index - 1;
    // 从"index+1"开始，用后面的元素替换前面的元素。
```

```
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    // 将最后一个元素设为 null
    elementData[--size] = null; // Let gc do its work
}
// 删除元素
public boolean remove(Object o) {
    if (o == null) {
        for (int index = 0; index < size; index++)
        if (elementData[index] == null) {
            fastRemove(index);
        return true;
        }
    } else {
        // 便利 ArrayList，找到“元素 o”，则删除，并返回 true。
        for (int index = 0; index < size; index++)
        if (o.equals(elementData[index])) {
            fastRemove(index);
        return true;
        }
    }
    return false;
}
// 清空 ArrayList，将全部的元素设为 null
public void clear() {
    modCount++;
    for (int i = 0; i < size; i++)
        elementData[i] = null;
    size = 0;
}
// 将集合 c 追加到 ArrayList 中
public boolean addAll(Collection<? extends E> c) {
    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacity(size + numNew);  // Increments modCount
    System.arraycopy(a, 0, elementData, size, numNew);
    size += numNew;
    return numNew != 0;
}
```

```
// 从 index 位置开始，将集合 c 添加到 ArrayList
public boolean addAll(int index, Collection<? extends E> c) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(
        "Index: " + index + ", Size: " + size);
    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacity(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;
}
// 删除 fromIndex 到 toIndex 之间的全部元素。
protected void removeRange(int fromIndex, int toIndex) {
modCount++;
int numMoved = size - toIndex;
    System.arraycopy(elementData, toIndex, elementData, fromIndex,
                     numMoved);
// Let gc do its work
int newSize = size - (toIndex-fromIndex);
while (size != newSize)
    elementData[--size] = null;
}
private void RangeCheck(int index) {
if (index >= size)
    throw new IndexOutOfBoundsException(
    "Index: "+index+", Size: "+size);
}
// 克隆函数
public Object clone() {
    try {
        ArrayList<E> v = (ArrayList<E>) super.clone();
        // 将当前 ArrayList 的全部元素拷贝到 v 中
        v.elementData = Arrays.copyOf(elementData, size);
        v.modCount = 0;
        return v;
```

```
        } catch (CloneNotSupportedException e) {
            // this shouldn't happen, since we are Cloneable
            throw new InternalError();
        }
    }
    // java.io.Serializable 的写入函数
    // 将 ArrayList 的“容量，所有的元素值”都写入到输出流中
    private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException{
    // Write out element count, and any hidden stuff
    int expectedModCount = modCount;
    s.defaultWriteObject();
        // 写入“数组的容量”
        s.writeInt(elementData.length);
    // 写入“数组的每一个元素”
    for (int i=0; i<size; i++)
            s.writeObject(elementData[i]);
    if (modCount != expectedModCount) {
            throw new ConcurrentModificationException();
        }
    }
    // java.io.Serializable 的读取函数：根据写入方式读出
    // 先将 ArrayList 的“容量”读出，然后将“所有的元素值”读出
    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        // Read in size, and any hidden stuff
        s.defaultReadObject();
        // 从输入流中读取 ArrayList 的“容量”
        int arrayLength = s.readInt();
        Object[] a = elementData = new Object[arrayLength];
        // 从输入流中将“所有的元素值”读出
        for (int i=0; i<size; i++)
            a[i] = s.readObject();
    }
}
```

总结

1. ArrayList 实际上是**通过一个数组去保存数据的。**当我们构造 ArrayList 时；若使用默认构造函数，则 ArrayList 的**默认容量大小是 10。**

2. 当 ArrayList 容量不足以容纳全部元素时，ArrayList 会重新设置容量：**新的容量 =“(原始容量 x3)/2 + 1”**。
3. ArrayList 的克隆函数，即是将全部元素克隆到一个数组中。
4. ArrayList 实现 java.io.Serializable 的方式。当写入到输出流时，先写入“容量”，再依次写入“每一个元素”；当读出输入流时，先读取“容量”，再依次读取“每一个元素”。

4. ArrayList 遍历方式

ArrayList 支持 3 种遍历方式

第一种，**通过迭代器遍历**。即通过 Iterator 去遍历。

```
Integer value = null;
Iterator iter = list.iterator();
while (iter.hasNext()) {
    value = (Integer)iter.next();
}
```

第二种，**随机访问，通过索引值去遍历。**
由于 ArrayList 实现了 RandomAccess 接口，它支持通过索引值去随机访问元素。

```
Integer value = null;
int size = list.size();
for (int i=0; i<size; i++) {
    value = (Integer)list.get(i);
}
```

第三种，**for 循环遍历**。

```
Integer value = null;
for (Integer integ:list) {
    value = integ;
}
```

下面通过一个实例，**比较这 3 种方式的效率**，实例代码(ArrayListRandomAccessTest.java)如下：

```
import java.util.*;
import java.util.concurrent.*;
/*
 * @desc ArrayList 遍历方式和效率的测试程序。
 *
 * @author skywang
```

```
 */
public class ArrayListRandomAccessTest {
    public static void main(String[] args) {
        List list = new ArrayList();
        for (int i=0; i<100000; i++)
            list.add(i);
        //isRandomAccessSupported(list);
        iteratorThroughRandomAccess(list) ;
        iteratorThroughIterator(list) ;
        iteratorThroughFor2(list) ;
    }
    private static void isRandomAccessSupported(List list) {
        if (list instanceof RandomAccess) {
            System.out.println("RandomAccess implemented!");
        } else {
            System.out.println("RandomAccess not implemented!");
        }
    }
    public static void iteratorThroughRandomAccess(List list) {
        long startTime;
        long endTime;
        startTime = System.currentTimeMillis();
        for (int i=0; i<list.size(); i++) {
            list.get(i);
        }
        endTime = System.currentTimeMillis();
        long interval = endTime - startTime;
        System.out.println("iteratorThroughRandomAccess：" + interval+" ms");
    }
    public static void iteratorThroughIterator(List list) {
        long startTime;
        long endTime;
        startTime = System.currentTimeMillis();
        for(Iterator iter = list.iterator(); iter.hasNext(); ) {
            iter.next();
        }
        endTime = System.currentTimeMillis();
        long interval = endTime - startTime;
        System.out.println("iteratorThroughIterator：" + interval+" ms");
    }
```

```
    public static void iteratorThroughFor2(List list) {
        long startTime;
        long endTime;
        startTime = System.currentTimeMillis();
        for(Object obj:list)
            ;
        endTime = System.currentTimeMillis();
        long interval = endTime - startTime;
        System.out.println("iteratorThroughFor2：" + interval+" ms");
    }
}
```

运行结果：

```
iteratorThroughRandomAccess：3 ms
iteratorThroughIterator：8 ms
iteratorThroughFor2：5 ms
```

由此可见，遍历 ArrayList 时，使用**随机访问(即，通过索引序号访问)效率最高**，而使用迭代器的效率最低！

5. toArray()异常

当我们调用 ArrayList 中的 toArray()，可能遇到过抛出“java.lang.ClassCastException”异常的情况。下面我们说说这是怎么回事。

ArrayList 提供了 2 个 toArray()函数:

```
Object[] toArray()
<T> T[] toArray(T[] contents)
```

调用 toArray() 函数会抛出“java.lang.ClassCastException”异常，但是调用 toArray(T[] contents) 能正常返回 T[]。

toArray() 会抛出异常是因为 toArray() 返回的是 Object[] 数组，将 Object[] 转换为其它类型(如如，将 Object[]转换为的 Integer[])则会抛出“java.lang.ClassCastException”异常，因为 Java 不支持向下转型。具体的可以参考前面 ArrayList.java 的源码介绍部分的 toArray()。

解决该问题的办法是调用 T[] toArray(T[] contents)，而不是 Object[] toArray()。

调用 toArray(T[] contents) 返回 T[]的可以通过以下几种方式实现。

```
// toArray(T[] contents)调用方式一
public static Integer[] vectorToArray1(ArrayList<Integer> v) {
    Integer[] newText = new Integer[v.size()];
    v.toArray(newText);
```

```
    return newText;
}
// toArray(T[] contents)调用方式二。最常用！
public static Integer[] vectorToArray2(ArrayList<Integer> v) {
    Integer[] newText = (Integer[])v.toArray(new Integer[0]);
    return newText;
}
// toArray(T[] contents)调用方式三
public static Integer[] vectorToArray3(ArrayList<Integer> v) {
    Integer[] newText = new Integer[v.size()];
    Integer[] newStrings = (Integer[])v.toArray(newText);
    return newStrings;
}
```

6. ArrayList 示例

本文通过一个实例(ArrayListTest.java)，介绍 ArrayList 中常用 API 的用法。

```
import java.util.*;
/*
 * @desc ArrayList 常用 API 的测试程序
 * @author skywang
 * @email kuiwu-wang@163.com
 */
public class ArrayListTest {
    public static void main(String[] args) {
        // 创建 ArrayList
        ArrayList list = new ArrayList();
        // 将“”
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        // 将下面的元素添加到第 1 个位置
        list.add(0, "5");
        // 获取第 1 个元素
        System.out.println("the first element is: "+ list.get(0));
        // 删除“3”
        list.remove("3");
        // 获取 ArrayList 的大小
```

```
        System.out.println("Arraylist size=: "+ list.size());
        // 判断 list 中是否包含"3"
        System.out.println("ArrayList contains 3 is: "+ list.contains(3));
        // 设置第 2 个元素为 10
        list.set(1, "10");
        // 通过 Iterator 遍历 ArrayList
        for(Iterator iter = list.iterator(); iter.hasNext(); ) {
            System.out.println("next is: "+ iter.next());
        }
        // 将 ArrayList 转换为数组
        String[] arr = (String[])list.toArray(new String[0]);
        for (String str:arr)
            System.out.println("str: "+ str);
        // 清空 ArrayList
        list.clear();
        // 判断 ArrayList 是否为空
        System.out.println("ArrayList is empty: "+ list.isEmpty());
    }
}
```

运行结果：

```
the first element is: 5
Arraylist size=: 4
ArrayList contains 3 is: false
next is: 5
next is: 10
next is: 2
next is: 4
str: 5
str: 10
str: 2
str: 4
ArrayList is empty: true
```