参考：<http://blog.csdn.net/ns_code/article/details/35568011>

[**【Java集合源码剖析】ArrayList源码剖析**](http://blog.csdn.net/ns_code/article/details/35568011)

**转载请注明出处：**[**http://blog.csdn.net/ns\_code/article/details/35568011**](http://blog.csdn.net/ns_code/article/details/35568011)

**本篇博文参加了CSDN博文大赛，如果您觉得这篇博文不错，希望您能帮我投一票，谢谢！**

**投票地址：**[**http://vote.blog.csdn.net/Article/Details?articleid=35568011**](http://vote.blog.csdn.net/Article/Details?articleid=35568011)

**ArrayList简介**

    ArrayList是基于数组实现的，是一个动态数组，其容量能自动增长，类似于C语言中的动态申请内存，动态增长内存。

    ArrayList不是线程安全的，只能用在单线程环境下，多线程环境下可以考虑用Collections.synchronizedList(List l)函数返回一个线程安全的ArrayList类，也可以使用concurrent并发包下的CopyOnWriteArrayList类。

    ArrayList实现了Serializable接口，因此它支持序列化，能够通过序列化传输，实现了RandomAccess接口，支持快速随机访问，实际上就是通过下标序号进行快速访问，实现了Cloneable接口，能被克隆。

**ArrayList源码剖析**

    ArrayList的源码如下（加入了比较详细的注释）：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/35568011) [copy](http://blog.csdn.net/ns_code/article/details/35568011)

1. **package** java.util;
3. **public** **class** ArrayList<E> **extends** AbstractList<E>
4. **implements** List<E>, RandomAccess, Cloneable, java.io.Serializable
5. {
6. // 序列版本号
7. **private** **static** **final** **long** serialVersionUID = 8683452581122892189L;
9. // ArrayList基于该数组实现，用该数组保存数据
10. **private** **transient** Object[] elementData;
12. // ArrayList中实际数据的数量
13. **private** **int** size;
15. // ArrayList带容量大小的构造函数。
16. **public** ArrayList(**int** initialCapacity) {
17. **super**();
18. **if** (initialCapacity < 0)
19. **throw** **new** IllegalArgumentException("Illegal Capacity: "+
20. initialCapacity);
21. // 新建一个数组
22. **this**.elementData = **new** Object[initialCapacity];
23. }
25. // ArrayList无参构造函数。默认容量是10。
26. **public** ArrayList() {
27. **this**(10);
28. }
30. // 创建一个包含collection的ArrayList
31. //使用了Arrays的copy方法来拷贝数组的元素
32. **public** ArrayList(Collection<? **extends** E> c) {
33. elementData = c.toArray();
34. size = elementData.length;
35. **if** (elementData.getClass() != Object[].**class**)
36. elementData = Arrays.copyOf(elementData, size, Object[].**class**);
37. }

40. // 将当前容量值设为实际元素个数
41. **public** **void** trimToSize() {
42. modCount++;
43. **int** oldCapacity = elementData.length;
44. **if** (size < oldCapacity) {
45. elementData = Arrays.copyOf(elementData, size);
46. }
47. }

50. // 确定ArrarList的容量。
51. // 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”
52. **public** **void** ensureCapacity(**int** minCapacity) {
53. // 将“修改统计数”+1，该变量主要是用来实现fail-fast机制的
54. modCount++;
55. **int** oldCapacity = elementData.length;
56. // 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”，即扩容1.5倍
57. **if** (minCapacity > oldCapacity) {
58. Object oldData[] = elementData;
59. **int** newCapacity = (oldCapacity \* 3)/2 + 1;
60. //如果还不够，则直接将minCapacity设置为当前容量
61. **if** (newCapacity < minCapacity)
62. newCapacity = minCapacity;
63. elementData = Arrays.copyOf(elementData, newCapacity);
64. }
65. }
67. // 添加元素e
68. **public** **boolean** add(E e) {
69. // 确定ArrayList的容量大小
70. ensureCapacity(size + 1);  // Increments modCount!!
71. // 添加e到ArrayList中
72. elementData[size++] = e;
73. **return** **true**;
74. }
76. // 返回ArrayList的实际大小
77. **public** **int** size() {
78. **return** size;
79. }
81. // ArrayList是否包含Object(o)
82. **public** **boolean** contains(Object o) {
83. **return** indexOf(o) >= 0;
84. }
86. //返回ArrayList是否为空
87. **public** **boolean** isEmpty() {
88. **return** size == 0;
89. }
91. // 正向查找，返回元素的索引值
92. **public** **int** indexOf(Object o) {
93. **if** (o == **null**) {
94. **for** (**int** i = 0; i < size; i++)
95. **if** (elementData[i]==**null**)
96. **return** i;
97. } **else** {
98. **for** (**int** i = 0; i < size; i++)
99. **if** (o.equals(elementData[i]))
100. **return** i;
101. }
102. **return** -1;
103. }
105. // 反向查找，返回元素的索引值
106. **public** **int** lastIndexOf(Object o) {
107. **if** (o == **null**) {
108. **for** (**int** i = size-1; i >= 0; i--)
109. **if** (elementData[i]==**null**)
110. **return** i;
111. } **else** {
112. **for** (**int** i = size-1; i >= 0; i--)
113. **if** (o.equals(elementData[i]))
114. **return** i;
115. }
116. **return** -1;
117. }
119. // 返回ArrayList的Object数组
120. **public** Object[] toArray() {
121. **return** Arrays.copyOf(elementData, size);
122. }
124. // 返回ArrayList元素组成的数组
125. **public** <T> T[] toArray(T[] a) {
126. // 若数组a的大小 < ArrayList的元素个数；
127. // 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
128. **if** (a.length < size)
129. **return** (T[]) Arrays.copyOf(elementData, size, a.getClass());
131. // 若数组a的大小 >= ArrayList的元素个数；
132. // 则将ArrayList的全部元素都拷贝到数组a中。
133. System.arraycopy(elementData, 0, a, 0, size);
134. **if** (a.length > size)
135. a[size] = **null**;
136. **return** a;
137. }
139. // 获取index位置的元素值
140. **public** E get(**int** index) {
141. RangeCheck(index);
143. **return** (E) elementData[index];
144. }
146. // 设置index位置的值为element
147. //该方法返回旧值
148. **public** E set(**int** index, E element) {
149. RangeCheck(index);
151. E oldValue = (E) elementData[index];
152. elementData[index] = element;
153. **return** oldValue;
154. }
156. // 将e添加到ArrayList中，添加到当前最后
157. **public** **boolean** add(E e) {
158. ensureCapacity(size + 1);  // Increments modCount!!
159. elementData[size++] = e;
160. **return** **true**;
161. }
163. // 将e添加到ArrayList的指定位置
164. //涉及到数组元素的移动
165. **public** **void** add(**int** index, E element) {
166. **if** (index > size || index < 0)
167. **throw** **new** IndexOutOfBoundsException(
168. "Index: "+index+", Size: "+size);
170. ensureCapacity(size+1);  // Increments modCount!!
171. System.arraycopy(elementData, index, elementData, index + 1,
172. size - index);
173. elementData[index] = element;
174. size++;
175. }
177. // 删除ArrayList指定位置的元素
178. //涉及到数组元素的移动
179. **public** E remove(**int** index) {
180. RangeCheck(index);
182. modCount++;
183. E oldValue = (E) elementData[index];
185. **int** numMoved = size - index - 1;
186. **if** (numMoved > 0)
187. System.arraycopy(elementData, index+1, elementData, index,
188. numMoved);
189. elementData[--size] = **null**; // Let gc do its work
191. **return** oldValue;
192. }
194. // 删除ArrayList的指定元素，仅仅删除正序第一个
195. **public** **boolean** remove(Object o) {
196. **if** (o == **null**) {
197. **for** (**int** index = 0; index < size; index++)
198. **if** (elementData[index] == **null**) {
199. fastRemove(index);
200. **return** **true**;
201. }
202. } **else** {
203. **for** (**int** index = 0; index < size; index++)
204. **if** (o.equals(elementData[index])) {
205. fastRemove(index);
206. **return** **true**;
207. }
208. }
209. **return** **false**;
210. }

213. // 快速删除第index个元素
214. **private** **void** fastRemove(**int** index) {
215. modCount++;
216. **int** numMoved = size - index - 1;
217. // 从"index+1"开始，用后面的元素替换前面的元素。
218. **if** (numMoved > 0)
219. System.arraycopy(elementData, index+1, elementData, index,
220. numMoved);
221. // 将最后一个元素设为null
222. elementData[--size] = **null**; // Let gc do its work
223. }
225. // 删除元素
226. **public** **boolean** remove(Object o) {
227. **if** (o == **null**) {
228. **for** (**int** index = 0; index < size; index++)
229. **if** (elementData[index] == **null**) {
230. fastRemove(index);
231. **return** **true**;
232. }
233. } **else** {
234. // ArrayList，找到“元素o”，则删除，并返回true。
235. **for** (**int** index = 0; index < size; index++)
236. **if** (o.equals(elementData[index])) {
237. fastRemove(index);
238. **return** **true**;
239. }
240. }
241. **return** **false**;
242. }
244. // 清空ArrayList，将全部的元素设为null
245. **public** **void** clear() {
246. modCount++;
248. **for** (**int** i = 0; i < size; i++)
249. elementData[i] = **null**;
251. size = 0;
252. }
254. // 将集合c追加到ArrayList中
255. **public** **boolean** addAll(Collection<? **extends** E> c) {
256. Object[] a = c.toArray();
257. **int** numNew = a.length;
258. ensureCapacity(size + numNew);  // Increments modCount
259. System.arraycopy(a, 0, elementData, size, numNew);
260. size += numNew;
261. **return** numNew != 0;
262. }
264. // 从index位置开始，将集合c添加到ArrayList
265. **public** **boolean** addAll(**int** index, Collection<? **extends** E> c) {
266. **if** (index > size || index < 0)
267. **throw** **new** IndexOutOfBoundsException(
268. "Index: " + index + ", Size: " + size);
270. Object[] a = c.toArray();
271. **int** numNew = a.length;
272. ensureCapacity(size + numNew);  // Increments modCount
274. **int** numMoved = size - index;
275. **if** (numMoved > 0)
276. System.arraycopy(elementData, index, elementData, index + numNew,
277. numMoved);
279. System.arraycopy(a, 0, elementData, index, numNew);
280. size += numNew;
281. **return** numNew != 0;
282. }
284. // 删除fromIndex到toIndex之间的全部元素。
285. **protected** **void** removeRange(**int** fromIndex, **int** toIndex) {
286. modCount++;
287. **int** numMoved = size - toIndex;
288. System.arraycopy(elementData, toIndex, elementData, fromIndex,
289. numMoved);
291. // Let gc do its work
292. **int** newSize = size - (toIndex-fromIndex);
293. **while** (size != newSize)
294. elementData[--size] = **null**;
295. }
297. **private** **void** RangeCheck(**int** index) {
298. **if** (index >= size)
299. **throw** **new** IndexOutOfBoundsException(
300. "Index: "+index+", Size: "+size);
301. }

304. // 克隆函数
305. **public** Object clone() {
306. **try** {
307. ArrayList<E> v = (ArrayList<E>) **super**.clone();
308. // 将当前ArrayList的全部元素拷贝到v中
309. v.elementData = Arrays.copyOf(elementData, size);
310. v.modCount = 0;
311. **return** v;
312. } **catch** (CloneNotSupportedException e) {
313. // this shouldn't happen, since we are Cloneable
314. **throw** **new** InternalError();
315. }
316. }

319. // java.io.Serializable的写入函数
320. // 将ArrayList的“容量，所有的元素值”都写入到输出流中
321. **private** **void** writeObject(java.io.ObjectOutputStream s)
322. **throws** java.io.IOException{
323. // Write out element count, and any hidden stuff
324. **int** expectedModCount = modCount;
325. s.defaultWriteObject();
327. // 写入“数组的容量”
328. s.writeInt(elementData.length);
330. // 写入“数组的每一个元素”
331. **for** (**int** i=0; i<size; i++)
332. s.writeObject(elementData[i]);
334. **if** (modCount != expectedModCount) {
335. **throw** **new** ConcurrentModificationException();
336. }
338. }

341. // java.io.Serializable的读取函数：根据写入方式读出
342. // 先将ArrayList的“容量”读出，然后将“所有的元素值”读出
343. **private** **void** readObject(java.io.ObjectInputStream s)
344. **throws** java.io.IOException, ClassNotFoundException {
345. // Read in size, and any hidden stuff
346. s.defaultReadObject();
348. // 从输入流中读取ArrayList的“容量”
349. **int** arrayLength = s.readInt();
350. Object[] a = elementData = **new** Object[arrayLength];
352. // 从输入流中将“所有的元素值”读出
353. **for** (**int** i=0; i<size; i++)
354. a[i] = s.readObject();
355. }
356. }

**几点总结**

    关于ArrayList的源码，给出几点比较重要的总结：

    1、注意其三个不同的构造方法。无参构造方法构造的ArrayList的容量默认为10，带有Collection参数的构造方法，将Collection转化为数组赋给ArrayList的实现数组elementData。

    2、注意扩充容量的方法ensureCapacity。ArrayList在每次增加元素（可能是1个，也可能是一组）时，都要调用该方法来确保足够的容量。当容量不足以容纳当前的元素个数时，就设置新的容量为旧的容量的1.5倍加1，如果设置后的新容量还不够，则直接新容量设置为传入的参数（也就是所需的容量），而后用Arrays.copyof()方法将元素拷贝到新的数组（详见下面的第3点）。从中可以看出，当容量不够时，每次增加元素，都要将原来的元素拷贝到一个新的数组中，非常之耗时，也因此建议在事先能确定元素数量或者元素数量很少的情况下，才使用ArrayList，否则建议使用LinkedList。

    3、ArrayList的实现中大量地调用了Arrays.copyof()和System.arraycopy()方法。我们有必要对这两个方法的实现做下深入的了解。

    首先来看Arrays.copyof()方法。它有很多个重载的方法，但实现思路都是一样的，我们来看泛型版本的源码：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/35568011) [copy](http://blog.csdn.net/ns_code/article/details/35568011)

1. **public** **static** <T> T[] copyOf(T[] original, **int** newLength) {
2. **return** (T[]) copyOf(original, newLength, original.getClass());
3. }

    很明显调用了另一个copyof方法，该方法有三个参数，最后一个参数指明要转换的数据的类型，其源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/35568011) [copy](http://blog.csdn.net/ns_code/article/details/35568011)

1. **public** **static** <T,U> T[] copyOf(U[] original, **int** newLength, Class<? **extends** T[]> newType) {
2. T[] copy = ((Object)newType == (Object)Object[].**class**)
3. ? (T[]) **new** Object[newLength]
4. : (T[]) Array.newInstance(newType.getComponentType(), newLength);
5. System.arraycopy(original, 0, copy, 0,
6. Math.min(original.length, newLength));
7. **return** copy;
8. }

    这里可以很明显地看出，该方法实际上是在其内部又创建了一个长度为newlength的数组，调用System.arraycopy()方法，将原来数组中的元素复制到了新的数组中。

    下面来看System.arraycopy()方法。该方法被标记了native，调用了系统的C/C++代码，在JDK中是看不到的，但在openJDK中可以看到其源码。该函数实际上最终调用了C语言的memmove()函数，因此它可以保证同一个数组内元素的正确复制和移动，比一般的复制方法的实现效率要高很多，很适合用来批量处理数组。Java强烈推荐在复制大量数组元素时用该方法，以取得更高的效率。

    4、注意ArrayList的两个转化为静态数组的toArray方法。

    第一个，Object[] toArray()方法。该方法有可能会抛出java.lang.ClassCastException异常，如果直接用向下转型的方法，将整个ArrayList集合转变为指定类型的Array数组，便会抛出该异常，而如果转化为Array数组时不向下转型，而是将每个元素向下转型，则不会抛出该异常，显然对数组中的元素一个个进行向下转型，效率不高，且不太方便。

    第二个，<T> T[] toArray(T[] a)方法。该方法可以直接将ArrayList转换得到的Array进行整体向下转型（转型其实是在该方法的源码中实现的），且从该方法的源码中可以看出，参数a的大小不足时，内部会调用Arrays.copyOf方法，该方法内部创建一个新的数组返回，因此对该方法的常用形式如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/35568011) [copy](http://blog.csdn.net/ns_code/article/details/35568011)

1. **public** **static** Integer[] vectorToArray2(ArrayList<Integer> v) {
2. Integer[] newText = (Integer[])v.toArray(**new** Integer[0]);
3. **return** newText;
4. }

     5、ArrayList基于数组实现，可以通过下标索引直接查找到指定位置的元素，因此查找效率高，但每次插入或删除元素，就要大量地移动元素，插入删除元素的效率低；相反，LinkedList由链表实现，插入和删除元素的效率高，但是查找效率低，且要保存下一元素的引用浪费空间。

    6、在查找给定元素索引值等的方法中，源码都将该元素的值分为null和不为null两种情况处理，ArrayList中允许元素为null。

**本篇博文参加了CSDN博文大赛，如果您觉得这篇博文不错，希望您能帮我投一票，谢谢！**

**投票地址：**[**http://vote.blog.csdn.net/Article/Details?articleid=35568011**](http://vote.blog.csdn.net/Article/Details?articleid=35568011)