参考：<http://blog.csdn.net/ns_code/article/details/35793865>

[**【Java集合源码剖析】Vector源码剖析**](http://blog.csdn.net/ns_code/article/details/35793865)

**转载请注明出处：[http://blog.csdn.net/ns\_code/article/details/35793865](http://blog.csdn.net/ns_code/article/details/35793865" \t "_blank)**

**Vector简介**

    Vector也是基于数组实现的，是一个动态数组，其容量能自动增长。

    Vector是JDK1.0引入了，它的很多实现方法都加入了同步语句，因此是线程安全的（其实也只是相对安全，有些时候还是要加入同步语句来保证线程的安全），可以用于多线程环境。

    Vector实现了Serializable接口，因此它支持串行化，实现了Cloneable接口，能被克隆，实现了RandomAccess接口，支持快速随机访问。

**Vector源码剖析**

    Vector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/35793865) [copy](http://blog.csdn.net/ns_code/article/details/35793865)

1. **package** java.util;
3. **public** **class** Vector<E>
4. **extends** AbstractList<E>
5. **implements** List<E>, RandomAccess, Cloneable, java.io.Serializable
6. {
8. // 保存Vector中数据的数组
9. **protected** Object[] elementData;
11. // 实际数据的数量
12. **protected** **int** elementCount;
14. // 容量增长系数
15. **protected** **int** capacityIncrement;
17. // 序列化id
18. **private** **static** **final** **long** serialVersionUID = -2767605614048989439L;
20. // Vector构造函数。默认容量是10。
21. **public** Vector() {
22. **this**(10);
23. }
25. // 指定Vector容量大小的构造函数
26. **public** Vector(**int** initialCapacity) {
27. **this**(initialCapacity, 0);
28. }
30. // 指定Vector"容量大小"和"增长系数"的构造函数
31. **public** Vector(**int** initialCapacity, **int** capacityIncrement) {
32. **super**();
33. **if** (initialCapacity < 0)
34. **throw** **new** IllegalArgumentException("Illegal Capacity: "+
35. initialCapacity);
36. // 新建一个数组，数组容量是initialCapacity
37. **this**.elementData = **new** Object[initialCapacity];
38. // 设置容量增长系数
39. **this**.capacityIncrement = capacityIncrement;
40. }
42. // 指定集合的Vector构造函数。
43. **public** Vector(Collection<? **extends** E> c) {
44. // 获取“集合(c)”的数组，并将其赋值给elementData
45. elementData = c.toArray();
46. // 设置数组长度
47. elementCount = elementData.length;
48. // c.toArray might (incorrectly) not return Object[] (see 6260652)
49. **if** (elementData.getClass() != Object[].**class**)
50. elementData = Arrays.copyOf(elementData, elementCount, Object[].**class**);
51. }
53. // 将数组Vector的全部元素都拷贝到数组anArray中
54. //vector之所以是线程安全的是因为它的方法都加上了synchronized关键字，即对方
55. //法都做了同步，这样虽然解决了线程安全的问题，但是非常的消耗性能
56. **public** **synchronized** **void** copyInto(Object[] anArray) {
57. System.arraycopy(elementData, 0, anArray, 0, elementCount);
58. }
60. // 将当前容量值设为 =实际元素个数
61. **public** **synchronized** **void** trimToSize() {
62. modCount++;
63. **int** oldCapacity = elementData.length;
64. **if** (elementCount < oldCapacity) {
65. elementData = Arrays.copyOf(elementData, elementCount);
66. }
67. }
69. // 确认“Vector容量”的帮助函数
70. **private** **void** ensureCapacityHelper(**int** minCapacity) {
71. **int** oldCapacity = elementData.length;
72. // 当Vector的容量不足以容纳当前的全部元素，增加容量大小。
73. // 若 容量增量系数>0(即capacityIncrement>0)，则将容量增大当capacityIncrement
74. // 否则，将容量增大一倍。
75. **if** (minCapacity > oldCapacity) {
76. Object[] oldData = elementData;
77. **int** newCapacity = (capacityIncrement > 0) ?
78. (oldCapacity + capacityIncrement) : (oldCapacity \* 2);
79. **if** (newCapacity < minCapacity) {
80. newCapacity = minCapacity;
81. }
82. elementData = Arrays.copyOf(elementData, newCapacity);
83. }
84. }
86. // 确定Vector的容量。
87. **public** **synchronized** **void** ensureCapacity(**int** minCapacity) {
88. // 将Vector的改变统计数+1
89. modCount++;
90. ensureCapacityHelper(minCapacity);
91. }
93. // 设置容量值为 newSize
94. **public** **synchronized** **void** setSize(**int** newSize) {
95. modCount++;
96. **if** (newSize > elementCount) {
97. // 若 "newSize 大于 Vector容量"，则调整Vector的大小。
98. ensureCapacityHelper(newSize);
99. } **else** {
100. // 若 "newSize 小于/等于 Vector容量"，则将newSize位置开始的元素都设置为null
101. **for** (**int** i = newSize ; i < elementCount ; i++) {
102. elementData[i] = **null**;
103. }
104. }
105. elementCount = newSize;
106. }
108. // 返回“Vector的总的容量”
109. **public** **synchronized** **int** capacity() {
110. **return** elementData.length;
111. }
113. // 返回“Vector的实际大小”，即Vector中元素个数
114. **public** **synchronized** **int** size() {
115. **return** elementCount;
116. }
118. // 判断Vector是否为空
119. **public** **synchronized** **boolean** isEmpty() {
120. **return** elementCount == 0;
121. }
123. // 返回“Vector中全部元素对应的Enumeration”
124. **public** Enumeration<E> elements() {
125. // 通过匿名类实现Enumeration
126. **return** **new** Enumeration<E>() {
127. **int** count = 0;
129. // 是否存在下一个元素
130. **public** **boolean** hasMoreElements() {
131. **return** count < elementCount;
132. }
134. // 获取下一个元素
135. **public** E nextElement() {
136. **synchronized** (Vector.**this**) {
137. **if** (count < elementCount) {
138. **return** (E)elementData[count++];
139. }
140. }
141. **throw** **new** NoSuchElementException("Vector Enumeration");
142. }
143. };
144. }
146. // 返回Vector中是否包含对象(o)
147. **public** **boolean** contains(Object o) {
148. **return** indexOf(o, 0) >= 0;
149. }

152. // 从index位置开始向后查找元素(o)。
153. // 若找到，则返回元素的索引值；否则，返回-1
154. **public** **synchronized** **int** indexOf(Object o, **int** index) {
155. **if** (o == **null**) {
156. // 若查找元素为null，则正向找出null元素，并返回它对应的序号
157. **for** (**int** i = index ; i < elementCount ; i++)
158. **if** (elementData[i]==**null**)
159. **return** i;
160. } **else** {
161. // 若查找元素不为null，则正向找出该元素，并返回它对应的序号
162. **for** (**int** i = index ; i < elementCount ; i++)
163. **if** (o.equals(elementData[i]))
164. **return** i;
165. }
166. **return** -1;
167. }
169. // 查找并返回元素(o)在Vector中的索引值
170. **public** **int** indexOf(Object o) {
171. **return** indexOf(o, 0);
172. }
174. // 从后向前查找元素(o)。并返回元素的索引
175. **public** **synchronized** **int** lastIndexOf(Object o) {
176. **return** lastIndexOf(o, elementCount-1);
177. }
179. // 从后向前查找元素(o)。开始位置是从前向后的第index个数；
180. // 若找到，则返回元素的“索引值”；否则，返回-1。
181. **public** **synchronized** **int** lastIndexOf(Object o, **int** index) {
182. **if** (index >= elementCount)
183. **throw** **new** IndexOutOfBoundsException(index + " >= "+ elementCount);
185. **if** (o == **null**) {
186. // 若查找元素为null，则反向找出null元素，并返回它对应的序号
187. **for** (**int** i = index; i >= 0; i--)
188. **if** (elementData[i]==**null**)
189. **return** i;
190. } **else** {
191. // 若查找元素不为null，则反向找出该元素，并返回它对应的序号
192. **for** (**int** i = index; i >= 0; i--)
193. **if** (o.equals(elementData[i]))
194. **return** i;
195. }
196. **return** -1;
197. }
199. // 返回Vector中index位置的元素。
200. // 若index越界，则抛出异常
201. **public** **synchronized** E elementAt(**int** index) {
202. **if** (index >= elementCount) {
203. **throw** **new** ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
204. }
206. **return** (E)elementData[index];
207. }
209. // 获取Vector中的第一个元素。
210. // 若失败，则抛出异常！
211. **public** **synchronized** E firstElement() {
212. **if** (elementCount == 0) {
213. **throw** **new** NoSuchElementException();
214. }
215. **return** (E)elementData[0];
216. }
218. // 获取Vector中的最后一个元素。
219. // 若失败，则抛出异常！
220. **public** **synchronized** E lastElement() {
221. **if** (elementCount == 0) {
222. **throw** **new** NoSuchElementException();
223. }
224. **return** (E)elementData[elementCount - 1];
225. }
227. // 设置index位置的元素值为obj
228. **public** **synchronized** **void** setElementAt(E obj, **int** index) {
229. **if** (index >= elementCount) {
230. **throw** **new** ArrayIndexOutOfBoundsException(index + " >= " +
231. elementCount);
232. }
233. elementData[index] = obj;
234. }
236. // 删除index位置的元素，涉及到数组的拷贝，数组元素的移动
237. **public** **synchronized** **void** removeElementAt(**int** index) {
238. modCount++;
239. **if** (index >= elementCount) {
240. **throw** **new** ArrayIndexOutOfBoundsException(index + " >= " +
241. elementCount);
242. } **else** **if** (index < 0) {
243. **throw** **new** ArrayIndexOutOfBoundsException(index);
244. }
246. **int** j = elementCount - index - 1;
247. **if** (j > 0) {
248. System.arraycopy(elementData, index + 1, elementData, index, j);
249. }
250. elementCount--;
251. elementData[elementCount] = **null**; /\* to let gc do its work \*/
252. }
254. // 在index位置处插入元素(obj)
255. **public** **synchronized** **void** insertElementAt(E obj, **int** index) {
256. modCount++;
257. **if** (index > elementCount) {
258. **throw** **new** ArrayIndexOutOfBoundsException(index
259. + " > " + elementCount);
260. }
261. ensureCapacityHelper(elementCount + 1);
262. System.arraycopy(elementData, index, elementData, index + 1, elementCount - index);
263. elementData[index] = obj;
264. elementCount++;
265. }
267. // 将“元素obj”添加到Vector末尾
268. **public** **synchronized** **void** addElement(E obj) {
269. modCount++;
270. ensureCapacityHelper(elementCount + 1);
271. elementData[elementCount++] = obj;
272. }
274. // 在Vector中查找并删除元素obj。
275. // 成功的话，返回true；否则，返回false。
276. **public** **synchronized** **boolean** removeElement(Object obj) {
277. modCount++;
278. **int** i = indexOf(obj);
279. **if** (i >= 0) {
280. removeElementAt(i);
281. **return** **true**;
282. }
283. **return** **false**;
284. }
286. // 删除Vector中的全部元素
287. **public** **synchronized** **void** removeAllElements() {
288. modCount++;
289. // 将Vector中的全部元素设为null
290. **for** (**int** i = 0; i < elementCount; i++)
291. elementData[i] = **null**;
293. elementCount = 0;
294. }
296. // 克隆函数
297. **public** **synchronized** Object clone() {
298. **try** {
299. Vector<E> v = (Vector<E>) **super**.clone();
300. // 将当前Vector的全部元素拷贝到v中
301. v.elementData = Arrays.copyOf(elementData, elementCount);
302. v.modCount = 0;
303. **return** v;
304. } **catch** (CloneNotSupportedException e) {
305. // this shouldn't happen, since we are Cloneable
306. **throw** **new** InternalError();
307. }
308. }
310. // 返回Object数组
311. **public** **synchronized** Object[] toArray() {
312. **return** Arrays.copyOf(elementData, elementCount);
313. }
315. // 返回Vector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
316. **public** **synchronized** <T> T[] toArray(T[] a) {
317. // 若数组a的大小 < Vector的元素个数；
318. // 则新建一个T[]数组，数组大小是“Vector的元素个数”，并将“Vector”全部拷贝到新数组中
319. **if** (a.length < elementCount)
320. **return** (T[]) Arrays.copyOf(elementData, elementCount, a.getClass());
322. // 若数组a的大小 >= Vector的元素个数；
323. // 则将Vector的全部元素都拷贝到数组a中。
324. System.arraycopy(elementData, 0, a, 0, elementCount);
326. **if** (a.length > elementCount)
327. a[elementCount] = **null**;
329. **return** a;
330. }
332. // 获取index位置的元素
333. **public** **synchronized** E get(**int** index) {
334. **if** (index >= elementCount)
335. **throw** **new** ArrayIndexOutOfBoundsException(index);
337. **return** (E)elementData[index];
338. }
340. // 设置index位置的值为element。并返回index位置的原始值
341. **public** **synchronized** E set(**int** index, E element) {
342. **if** (index >= elementCount)
343. **throw** **new** ArrayIndexOutOfBoundsException(index);
345. Object oldValue = elementData[index];
346. elementData[index] = element;
347. **return** (E)oldValue;
348. }
350. // 将“元素e”添加到Vector最后。
351. **public** **synchronized** **boolean** add(E e) {
352. modCount++;
353. ensureCapacityHelper(elementCount + 1);
354. elementData[elementCount++] = e;
355. **return** **true**;
356. }
358. // 删除Vector中的元素o
359. **public** **boolean** remove(Object o) {
360. **return** removeElement(o);
361. }
363. // 在index位置添加元素element
364. **public** **void** add(**int** index, E element) {
365. insertElementAt(element, index);
366. }
368. // 删除index位置的元素，并返回index位置的原始值
369. **public** **synchronized** E remove(**int** index) {
370. modCount++;
371. **if** (index >= elementCount)
372. **throw** **new** ArrayIndexOutOfBoundsException(index);
373. Object oldValue = elementData[index];
375. **int** numMoved = elementCount - index - 1;
376. **if** (numMoved > 0)
377. System.arraycopy(elementData, index+1, elementData, index,
378. numMoved);
379. elementData[--elementCount] = **null**; // Let gc do its work
381. **return** (E)oldValue;
382. }
384. // 清空Vector
385. **public** **void** clear() {
386. removeAllElements();
387. }
389. // 返回Vector是否包含集合c
390. **public** **synchronized** **boolean** containsAll(Collection<?> c) {
391. **return** **super**.containsAll(c);
392. }
394. // 将集合c添加到Vector中
395. **public** **synchronized** **boolean** addAll(Collection<? **extends** E> c) {
396. modCount++;
397. Object[] a = c.toArray();
398. **int** numNew = a.length;
399. ensureCapacityHelper(elementCount + numNew);
400. // 将集合c的全部元素拷贝到数组elementData中
401. System.arraycopy(a, 0, elementData, elementCount, numNew);
402. elementCount += numNew;
403. **return** numNew != 0;
404. }
406. // 删除集合c的全部元素
407. **public** **synchronized** **boolean** removeAll(Collection<?> c) {
408. **return** **super**.removeAll(c);
409. }
411. // 删除“非集合c中的元素”
412. **public** **synchronized** **boolean** retainAll(Collection<?> c)  {
413. **return** **super**.retainAll(c);
414. }
416. // 从index位置开始，将集合c添加到Vector中
417. **public** **synchronized** **boolean** addAll(**int** index, Collection<? **extends** E> c) {
418. modCount++;
419. **if** (index < 0 || index > elementCount)
420. **throw** **new** ArrayIndexOutOfBoundsException(index);
422. Object[] a = c.toArray();
423. **int** numNew = a.length;
424. ensureCapacityHelper(elementCount + numNew);
426. **int** numMoved = elementCount - index;
427. **if** (numMoved > 0)
428. System.arraycopy(elementData, index, elementData, index + numNew, numMoved);
430. System.arraycopy(a, 0, elementData, index, numNew);
431. elementCount += numNew;
432. **return** numNew != 0;
433. }
435. // 返回两个对象是否相等
436. **public** **synchronized** **boolean** equals(Object o) {
437. **return** **super**.equals(o);
438. }
440. // 计算哈希值
441. **public** **synchronized** **int** hashCode() {
442. **return** **super**.hashCode();
443. }
445. // 调用父类的toString()
446. **public** **synchronized** String toString() {
447. **return** **super**.toString();
448. }
450. // 获取Vector中fromIndex(包括)到toIndex(不包括)的子集
451. **public** **synchronized** List<E> subList(**int** fromIndex, **int** toIndex) {
452. **return** Collections.synchronizedList(**super**.subList(fromIndex, toIndex), **this**);
453. }
455. // 删除Vector中fromIndex到toIndex的元素
456. **protected** **synchronized** **void** removeRange(**int** fromIndex, **int** toIndex) {
457. modCount++;
458. **int** numMoved = elementCount - toIndex;
459. System.arraycopy(elementData, toIndex, elementData, fromIndex,
460. numMoved);
462. // Let gc do its work
463. **int** newElementCount = elementCount - (toIndex-fromIndex);
464. **while** (elementCount != newElementCount)
465. elementData[--elementCount] = **null**;
466. }
468. // java.io.Serializable的写入函数
469. **private** **synchronized** **void** writeObject(java.io.ObjectOutputStream s)
470. **throws** java.io.IOException {
471. s.defaultWriteObject();
472. }
473. }

**几点总结**

    Vector的源码实现总体与ArrayList类似，关于Vector的源码，给出如下几点总结：

    1、Vector有四个不同的构造方法。无参构造方法的容量为默认值10，仅包含容量的构造方法则将容量增长量（从源码中可以看出容量增长量的作用，第二点也会对容量增长量详细说）明置为0。

    2、注意扩充容量的方法ensureCapacityHelper。与ArrayList相同，Vector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，就先看构造方法中传入的容量增长量参数CapacityIncrement是否为0，如果不为0，就设置新的容量为旧容量加上容量增长量，如果为0，就设置新的容量为旧的容量的2倍，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），而后同样用Arrays.copyof()方法将元素拷贝到新的数组。

    3、很多方法都加入了synchronized同步语句，来保证线程安全。

    4、同样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，Vector中也允许元素为null。

    5、其他很多地方都与ArrayList实现大同小异，Vector现在已经基本不再使用。