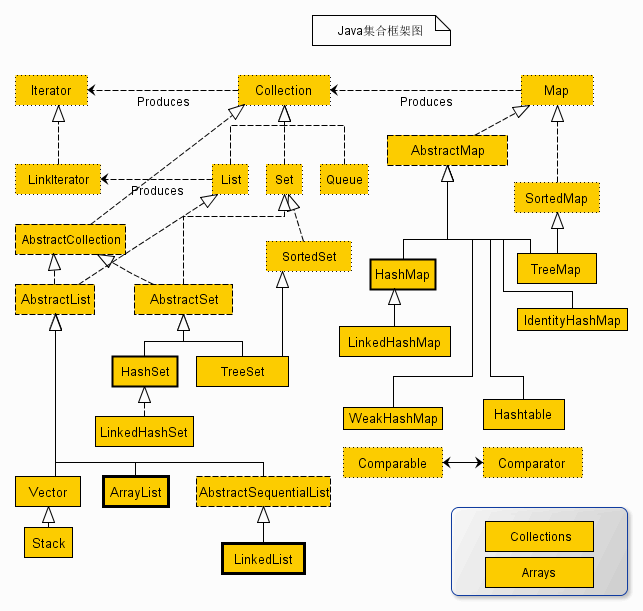
**Java中的集合**



　上述类图中，**实线边框**的是**实现类**，比如ArrayList，LinkedList，HashMap等，**折线边框**的是**抽象类**，比如AbstractCollection，AbstractList，AbstractMap等，而**点线边框**的是**接口**，比如Collection，Iterator，List等。

**3.比较**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 是否有序 | 是否允许元素重复 |
| Collection | | 否 | 是 |
| List | | 是 | 是 |
| Set | AbstractSet | 否 | 否 |
|  | HashSet |
|  | TreeSet | 是（用二叉排序树） |
| Map | AbstractMap | 否 | 使用key-value来映射和存储数据，key必须唯一，value可以重复 |
|  | HashMap |
|  | TreeMap | 是（用二叉排序树） |

# HashMap简介

    HashMap是基于哈希表实现的，每一个元素是一个key-value对，其内部通过单链表解决冲突问题，容量不足（超过了阀值）时，同样会自动增长。

    HashMap是非线程安全的，只是用于单线程环境下，多线程环境下可以采用concurrent并发包下的concurrentHashMap。

    HashMap 实现了Serializable接口，因此它支持序列化，实现了Cloneable接口，能被克隆。

# HashMap源码剖析

    HashMap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/36034955) [copy](http://blog.csdn.net/ns_code/article/details/36034955)

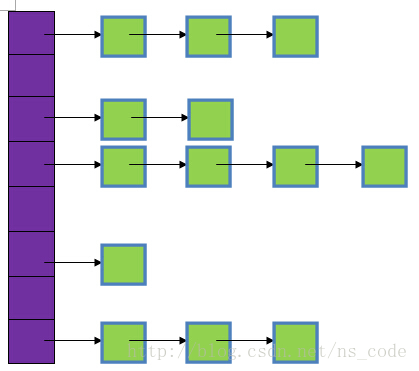
1. **package** java.util;
2. **import** java.io.\*;
4. **public** **class** HashMap<K,V>
5. **extends** AbstractMap<K,V>
6. **implements** Map<K,V>, Cloneable, Serializable
7. {
9. // 默认的初始容量（容量为HashMap中槽的数目）是16，且实际容量必须是2的整数次幂。
10. **static** **final** **int** DEFAULT\_INITIAL\_CAPACITY = 16;
12. // 最大容量（必须是2的幂且小于2的30次方，传入容量过大将被这个值替换）
13. **static** **final** **int** MAXIMUM\_CAPACITY = 1 << 30;
15. // 默认加载因子为0.75
16. **static** **final** **float** DEFAULT\_LOAD\_FACTOR = 0.75f;
18. // 存储数据的Entry数组，长度是2的幂。
19. // HashMap采用链表法解决冲突，每一个Entry本质上是一个单向链表
20. **transient** Entry[] table;
22. // HashMap的底层数组中已用槽的数量
23. **transient** **int** size;
25. // HashMap的阈值，用于判断是否需要调整HashMap的容量（threshold = 容量\*加载因子）
26. **int** threshold;
28. // 加载因子实际大小
29. **final** **float** loadFactor;
31. // HashMap被改变的次数
32. **transient** **volatile** **int** modCount;
34. // 指定“容量大小”和“加载因子”的构造函数
35. **public** HashMap(**int** initialCapacity, **float** loadFactor) {
36. **if** (initialCapacity < 0)
37. **throw** **new** IllegalArgumentException("Illegal initial capacity: " +
38. initialCapacity);
39. // HashMap的最大容量只能是MAXIMUM\_CAPACITY
40. **if** (initialCapacity > MAXIMUM\_CAPACITY)
41. initialCapacity = MAXIMUM\_CAPACITY;
42. //加载因此不能小于0
43. **if** (loadFactor <= 0 || Float.isNaN(loadFactor))
44. **throw** **new** IllegalArgumentException("Illegal load factor: " +
45. loadFactor);
47. // 找出“大于initialCapacity”的最小的2的幂
48. **int** capacity = 1;
49. **while** (capacity < initialCapacity)
50. capacity <<= 1;
52. // 设置“加载因子”
53. **this**.loadFactor = loadFactor;
54. // 设置“HashMap阈值”，当HashMap中存储数据的数量达到threshold时，就需要将HashMap的容量加倍。
55. threshold = (**int**)(capacity \* loadFactor);
56. // 创建Entry数组，用来保存数据
57. table = **new** Entry[capacity];
58. init();
59. }

62. // 指定“容量大小”的构造函数
63. **public** HashMap(**int** initialCapacity) {
64. **this**(initialCapacity, DEFAULT\_LOAD\_FACTOR);
65. }
67. // 默认构造函数。
68. **public** HashMap() {
69. // 设置“加载因子”为默认加载因子0.75
70. **this**.loadFactor = DEFAULT\_LOAD\_FACTOR;
71. // 设置“HashMap阈值”，当HashMap中存储数据的数量达到threshold时，就需要将HashMap的容量加倍。
72. threshold = (**int**)(DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR);
73. // 创建Entry数组，用来保存数据
74. table = **new** Entry[DEFAULT\_INITIAL\_CAPACITY];
75. init();
76. }
78. // 包含“子Map”的构造函数
79. **public** HashMap(Map<? **extends** K, ? **extends** V> m) {
80. **this**(Math.max((**int**) (m.size() / DEFAULT\_LOAD\_FACTOR) + 1,
81. DEFAULT\_INITIAL\_CAPACITY), DEFAULT\_LOAD\_FACTOR);
82. // 将m中的全部元素逐个添加到HashMap中
83. putAllForCreate(m);
84. }
86. //求hash值的方法，重新计算hash值
87. **static** **int** hash(**int** h) {
88. h ^= (h >>> 20) ^ (h >>> 12);
89. **return** h ^ (h >>> 7) ^ (h >>> 4);
90. }
92. // 返回h在数组中的索引值，这里用&代替取模，旨在提升效率
93. // h & (length-1)保证返回值的小于length
94. **static** **int** indexFor(**int** h, **int** length) {
95. **return** h & (length-1);
96. }
98. **public** **int** size() {
99. **return** size;
100. }
102. **public** **boolean** isEmpty() {
103. **return** size == 0;
104. }
106. // 获取key对应的value
107. **public** V get(Object key) {
108. **if** (key == **null**)
109. **return** getForNullKey();
110. // 获取key的hash值
111. **int** hash = hash(key.hashCode());
112. // 在“该hash值对应的链表”上查找“键值等于key”的元素
113. **for** (Entry<K,V> e = table[indexFor(hash, table.length)];
114. e != **null**;
115. e = e.next) {
116. Object k;
117. //判断key是否相同
118. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k)))
119. **return** e.value;
120. }
121. //没找到则返回null
122. **return** **null**;
123. }
125. // 获取“key为null”的元素的值
126. // HashMap将“key为null”的元素存储在table[0]位置，但不一定是该链表的第一个位置！
127. **private** V getForNullKey() {
128. **for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {
129. **if** (e.key == **null**)
130. **return** e.value;
131. }
132. **return** **null**;
133. }
135. // HashMap是否包含key
136. **public** **boolean** containsKey(Object key) {
137. **return** getEntry(key) != **null**;
138. }
140. // 返回“键为key”的键值对
141. **final** Entry<K,V> getEntry(Object key) {
142. // 获取哈希值
143. // HashMap将“key为null”的元素存储在table[0]位置，“key不为null”的则调用hash()计算哈希值
144. **int** hash = (key == **null**) ? 0 : hash(key.hashCode());
145. // 在“该hash值对应的链表”上查找“键值等于key”的元素
146. **for** (Entry<K,V> e = table[indexFor(hash, table.length)];
147. e != **null**;
148. e = e.next) {
149. Object k;
150. **if** (e.hash == hash &&
151. ((k = e.key) == key || (key != **null** && key.equals(k))))
152. **return** e;
153. }
154. **return** **null**;
155. }
157. // 将“key-value”添加到HashMap中
158. **public** V put(K key, V value) {
159. // 若“key为null”，则将该键值对添加到table[0]中。
160. **if** (key == **null**)
161. **return** putForNullKey(value);
162. // 若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。
163. **int** hash = hash(key.hashCode());
164. **int** i = indexFor(hash, table.length);
165. **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {
166. Object k;
167. // 若“该key”对应的键值对已经存在，则用新的value取代旧的value。然后退出！
168. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
169. V oldValue = e.value;
170. e.value = value;
171. e.recordAccess(**this**);
172. **return** oldValue;
173. }
174. }
176. // 若“该key”对应的键值对不存在，则将“key-value”添加到table中
177. modCount++;
178. //将key-value添加到table[i]处
179. addEntry(hash, key, value, i);
180. **return** **null**;
181. }
183. // putForNullKey()的作用是将“key为null”键值对添加到table[0]位置
184. **private** V putForNullKey(V value) {
185. **for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {
186. **if** (e.key == **null**) {
187. V oldValue = e.value;
188. e.value = value;
189. e.recordAccess(**this**);
190. **return** oldValue;
191. }
192. }
193. // 如果没有存在key为null的键值对，则直接题阿见到table[0]处!
194. modCount++;
195. addEntry(0, **null**, value, 0);
196. **return** **null**;
197. }
199. // 创建HashMap对应的“添加方法”，
200. // 它和put()不同。putForCreate()是内部方法，它被构造函数等调用，用来创建HashMap
201. // 而put()是对外提供的往HashMap中添加元素的方法。
202. **private** **void** putForCreate(K key, V value) {
203. **int** hash = (key == **null**) ? 0 : hash(key.hashCode());
204. **int** i = indexFor(hash, table.length);
206. // 若该HashMap表中存在“键值等于key”的元素，则替换该元素的value值
207. **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {
208. Object k;
209. **if** (e.hash == hash &&
210. ((k = e.key) == key || (key != **null** && key.equals(k)))) {
211. e.value = value;
212. **return**;
213. }
214. }
216. // 若该HashMap表中不存在“键值等于key”的元素，则将该key-value添加到HashMap中
217. createEntry(hash, key, value, i);
218. }
220. // 将“m”中的全部元素都添加到HashMap中。
221. // 该方法被内部的构造HashMap的方法所调用。
222. **private** **void** putAllForCreate(Map<? **extends** K, ? **extends** V> m) {
223. // 利用迭代器将元素逐个添加到HashMap中
224. **for** (Iterator<? **extends** Map.Entry<? **extends** K, ? **extends** V>> i = m.entrySet().iterator(); i.hasNext(); ) {
225. Map.Entry<? **extends** K, ? **extends** V> e = i.next();
226. putForCreate(e.getKey(), e.getValue());
227. }
228. }
230. // 重新调整HashMap的大小，newCapacity是调整后的容量
231. **void** resize(**int** newCapacity) {
232. Entry[] oldTable = table;
233. **int** oldCapacity = oldTable.length;
234. //如果就容量已经达到了最大值，则不能再扩容，直接返回
235. **if** (oldCapacity == MAXIMUM\_CAPACITY) {
236. threshold = Integer.MAX\_VALUE;
237. **return**;
238. }
240. // 新建一个HashMap，将“旧HashMap”的全部元素添加到“新HashMap”中，
241. // 然后，将“新HashMap”赋值给“旧HashMap”。
242. Entry[] newTable = **new** Entry[newCapacity];
243. transfer(newTable);
244. table = newTable;
245. threshold = (**int**)(newCapacity \* loadFactor);
246. }
248. // 将HashMap中的全部元素都添加到newTable中
249. **void** transfer(Entry[] newTable) {
250. Entry[] src = table;
251. **int** newCapacity = newTable.length;
252. **for** (**int** j = 0; j < src.length; j++) {
253. Entry<K,V> e = src[j];
254. **if** (e != **null**) {
255. src[j] = **null**;
256. **do** {
257. Entry<K,V> next = e.next;
258. **int** i = indexFor(e.hash, newCapacity);
259. e.next = newTable[i];
260. newTable[i] = e;
261. e = next;
262. } **while** (e != **null**);
263. }
264. }
265. }
267. // 将"m"的全部元素都添加到HashMap中
268. **public** **void** putAll(Map<? **extends** K, ? **extends** V> m) {
269. // 有效性判断
270. **int** numKeysToBeAdded = m.size();
271. **if** (numKeysToBeAdded == 0)
272. **return**;
274. // 计算容量是否足够，
275. // 若“当前阀值容量 < 需要的容量”，则将容量x2。
276. **if** (numKeysToBeAdded > threshold) {
277. **int** targetCapacity = (**int**)(numKeysToBeAdded / loadFactor + 1);
278. **if** (targetCapacity > MAXIMUM\_CAPACITY)
279. targetCapacity = MAXIMUM\_CAPACITY;
280. **int** newCapacity = table.length;
281. **while** (newCapacity < targetCapacity)
282. newCapacity <<= 1;
283. **if** (newCapacity > table.length)
284. resize(newCapacity);
285. }
287. // 通过迭代器，将“m”中的元素逐个添加到HashMap中。
288. **for** (Iterator<? **extends** Map.Entry<? **extends** K, ? **extends** V>> i = m.entrySet().iterator(); i.hasNext(); ) {
289. Map.Entry<? **extends** K, ? **extends** V> e = i.next();
290. put(e.getKey(), e.getValue());
291. }
292. }
294. // 删除“键为key”元素
295. **public** V remove(Object key) {
296. Entry<K,V> e = removeEntryForKey(key);
297. **return** (e == **null** ? **null** : e.value);
298. }
300. // 删除“键为key”的元素
301. **final** Entry<K,V> removeEntryForKey(Object key) {
302. // 获取哈希值。若key为null，则哈希值为0；否则调用hash()进行计算
303. **int** hash = (key == **null**) ? 0 : hash(key.hashCode());
304. **int** i = indexFor(hash, table.length);
305. Entry<K,V> prev = table[i];
306. Entry<K,V> e = prev;
308. // 删除链表中“键为key”的元素
309. // 本质是“删除单向链表中的节点”
310. **while** (e != **null**) {
311. Entry<K,V> next = e.next;
312. Object k;
313. **if** (e.hash == hash &&
314. ((k = e.key) == key || (key != **null** && key.equals(k)))) {
315. modCount++;
316. size--;
317. **if** (prev == e)
318. table[i] = next;
319. **else**
320. prev.next = next;
321. e.recordRemoval(**this**);
322. **return** e;
323. }
324. prev = e;
325. e = next;
326. }
328. **return** e;
329. }
331. // 删除“键值对”
332. **final** Entry<K,V> removeMapping(Object o) {
333. **if** (!(o **instanceof** Map.Entry))
334. **return** **null**;
336. Map.Entry<K,V> entry = (Map.Entry<K,V>) o;
337. Object key = entry.getKey();
338. **int** hash = (key == **null**) ? 0 : hash(key.hashCode());
339. **int** i = indexFor(hash, table.length);
340. Entry<K,V> prev = table[i];
341. Entry<K,V> e = prev;
343. // 删除链表中的“键值对e”
344. // 本质是“删除单向链表中的节点”
345. **while** (e != **null**) {
346. Entry<K,V> next = e.next;
347. **if** (e.hash == hash && e.equals(entry)) {
348. modCount++;
349. size--;
350. **if** (prev == e)
351. table[i] = next;
352. **else**
353. prev.next = next;
354. e.recordRemoval(**this**);
355. **return** e;
356. }
357. prev = e;
358. e = next;
359. }
361. **return** e;
362. }
364. // 清空HashMap，将所有的元素设为null
365. **public** **void** clear() {
366. modCount++;
367. Entry[] tab = table;
368. **for** (**int** i = 0; i < tab.length; i++)
369. tab[i] = **null**;
370. size = 0;
371. }
373. // 是否包含“值为value”的元素
374. **public** **boolean** containsValue(Object value) {
375. // 若“value为null”，则调用containsNullValue()查找
376. **if** (value == **null**)
377. **return** containsNullValue();
379. // 若“value不为null”，则查找HashMap中是否有值为value的节点。
380. Entry[] tab = table;
381. **for** (**int** i = 0; i < tab.length ; i++)
382. **for** (Entry e = tab[i] ; e != **null** ; e = e.next)
383. **if** (value.equals(e.value))
384. **return** **true**;
385. **return** **false**;
386. }
388. // 是否包含null值
389. **private** **boolean** containsNullValue() {
390. Entry[] tab = table;
391. **for** (**int** i = 0; i < tab.length ; i++)
392. **for** (Entry e = tab[i] ; e != **null** ; e = e.next)
393. **if** (e.value == **null**)
394. **return** **true**;
395. **return** **false**;
396. }
398. // 克隆一个HashMap，并返回Object对象
399. **public** Object clone() {
400. HashMap<K,V> result = **null**;
401. **try** {
402. result = (HashMap<K,V>)**super**.clone();
403. } **catch** (CloneNotSupportedException e) {
404. // assert false;
405. }
406. result.table = **new** Entry[table.length];
407. result.entrySet = **null**;
408. result.modCount = 0;
409. result.size = 0;
410. result.init();
411. // 调用putAllForCreate()将全部元素添加到HashMap中
412. result.putAllForCreate(**this**);
414. **return** result;
415. }
417. // Entry是单向链表。
418. // 它是 “HashMap链式存储法”对应的链表。
419. // 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Object o), hashCode()这些函数
420. **static** **class** Entry<K,V> **implements** Map.Entry<K,V> {
421. **final** K key;
422. V value;
423. // 指向下一个节点
424. Entry<K,V> next;
425. **final** **int** hash;
427. // 构造函数。
428. // 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"
429. Entry(**int** h, K k, V v, Entry<K,V> n) {
430. value = v;
431. next = n;
432. key = k;
433. hash = h;
434. }
436. **public** **final** K getKey() {
437. **return** key;
438. }
440. **public** **final** V getValue() {
441. **return** value;
442. }
444. **public** **final** V setValue(V newValue) {
445. V oldValue = value;
446. value = newValue;
447. **return** oldValue;
448. }
450. // 判断两个Entry是否相等
451. // 若两个Entry的“key”和“value”都相等，则返回true。
452. // 否则，返回false
453. **public** **final** **boolean** equals(Object o) {
454. **if** (!(o **instanceof** Map.Entry))
455. **return** **false**;
456. Map.Entry e = (Map.Entry)o;
457. Object k1 = getKey();
458. Object k2 = e.getKey();
459. **if** (k1 == k2 || (k1 != **null** && k1.equals(k2))) {
460. Object v1 = getValue();
461. Object v2 = e.getValue();
462. **if** (v1 == v2 || (v1 != **null** && v1.equals(v2)))
463. **return** **true**;
464. }
465. **return** **false**;
466. }
468. // 实现hashCode()
469. **public** **final** **int** hashCode() {
470. **return** (key==**null**   ? 0 : key.hashCode()) ^
471. (value==**null** ? 0 : value.hashCode());
472. }
474. **public** **final** String toString() {
475. **return** getKey() + "=" + getValue();
476. }
478. // 当向HashMap中添加元素时，绘调用recordAccess()。
479. // 这里不做任何处理
480. **void** recordAccess(HashMap<K,V> m) {
481. }
483. // 当从HashMap中删除元素时，绘调用recordRemoval()。
484. // 这里不做任何处理
485. **void** recordRemoval(HashMap<K,V> m) {
486. }
487. }
489. // 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。
490. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
491. // 保存“bucketIndex”位置的值到“e”中
492. Entry<K,V> e = table[bucketIndex];
493. // 设置“bucketIndex”位置的元素为“新Entry”，
494. // 设置“e”为“新Entry的下一个节点”
495. table[bucketIndex] = **new** Entry<K,V>(hash, key, value, e);
496. // 若HashMap的实际大小 不小于 “阈值”，则调整HashMap的大小
497. **if** (size++ >= threshold)
498. resize(2 \* table.length);
499. }
501. // 创建Entry。将“key-value”插入指定位置。
502. **void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {
503. // 保存“bucketIndex”位置的值到“e”中
504. Entry<K,V> e = table[bucketIndex];
505. // 设置“bucketIndex”位置的元素为“新Entry”，
506. // 设置“e”为“新Entry的下一个节点”
507. table[bucketIndex] = **new** Entry<K,V>(hash, key, value, e);
508. size++;
509. }
511. // HashIterator是HashMap迭代器的抽象出来的父类，实现了公共了函数。
512. // 它包含“key迭代器(KeyIterator)”、“Value迭代器(ValueIterator)”和“Entry迭代器(EntryIterator)”3个子类。
513. **private** **abstract** **class** HashIterator<E> **implements** Iterator<E> {
514. // 下一个元素
515. Entry<K,V> next;
516. // expectedModCount用于实现fast-fail机制。
517. **int** expectedModCount;
518. // 当前索引
519. **int** index;
520. // 当前元素
521. Entry<K,V> current;
523. HashIterator() {
524. expectedModCount = modCount;
525. **if** (size > 0) { // advance to first entry
526. Entry[] t = table;
527. // 将next指向table中第一个不为null的元素。
528. // 这里利用了index的初始值为0，从0开始依次向后遍历，直到找到不为null的元素就退出循环。
529. **while** (index < t.length && (next = t[index++]) == **null**)
530. ;
531. }
532. }
534. **public** **final** **boolean** hasNext() {
535. **return** next != **null**;
536. }
538. // 获取下一个元素
539. **final** Entry<K,V> nextEntry() {
540. **if** (modCount != expectedModCount)
541. **throw** **new** ConcurrentModificationException();
542. Entry<K,V> e = next;
543. **if** (e == **null**)
544. **throw** **new** NoSuchElementException();
546. // 注意！！！
547. // 一个Entry就是一个单向链表
548. // 若该Entry的下一个节点不为空，就将next指向下一个节点;
549. // 否则，将next指向下一个链表(也是下一个Entry)的不为null的节点。
550. **if** ((next = e.next) == **null**) {
551. Entry[] t = table;
552. **while** (index < t.length && (next = t[index++]) == **null**)
553. ;
554. }
555. current = e;
556. **return** e;
557. }
559. // 删除当前元素
560. **public** **void** remove() {
561. **if** (current == **null**)
562. **throw** **new** IllegalStateException();
563. **if** (modCount != expectedModCount)
564. **throw** **new** ConcurrentModificationException();
565. Object k = current.key;
566. current = **null**;
567. HashMap.**this**.removeEntryForKey(k);
568. expectedModCount = modCount;
569. }
571. }
573. // value的迭代器
574. **private** **final** **class** ValueIterator **extends** HashIterator<V> {
575. **public** V next() {
576. **return** nextEntry().value;
577. }
578. }
580. // key的迭代器
581. **private** **final** **class** KeyIterator **extends** HashIterator<K> {
582. **public** K next() {
583. **return** nextEntry().getKey();
584. }
585. }
587. // Entry的迭代器
588. **private** **final** **class** EntryIterator **extends** HashIterator<Map.Entry<K,V>> {
589. **public** Map.Entry<K,V> next() {
590. **return** nextEntry();
591. }
592. }
594. // 返回一个“key迭代器”
595. Iterator<K> newKeyIterator()   {
596. **return** **new** KeyIterator();
597. }
598. // 返回一个“value迭代器”
599. Iterator<V> newValueIterator()   {
600. **return** **new** ValueIterator();
601. }
602. // 返回一个“entry迭代器”
603. Iterator<Map.Entry<K,V>> newEntryIterator()   {
604. **return** **new** EntryIterator();
605. }
607. // HashMap的Entry对应的集合
608. **private** **transient** Set<Map.Entry<K,V>> entrySet = **null**;
610. // 返回“key的集合”，实际上返回一个“KeySet对象”
611. **public** Set<K> keySet() {
612. Set<K> ks = keySet;
613. **return** (ks != **null** ? ks : (keySet = **new** KeySet()));
614. }
616. // Key对应的集合
617. // KeySet继承于AbstractSet，说明该集合中没有重复的Key。
618. **private** **final** **class** KeySet **extends** AbstractSet<K> {
619. **public** Iterator<K> iterator() {
620. **return** newKeyIterator();
621. }
622. **public** **int** size() {
623. **return** size;
624. }
625. **public** **boolean** contains(Object o) {
626. **return** containsKey(o);
627. }
628. **public** **boolean** remove(Object o) {
629. **return** HashMap.**this**.removeEntryForKey(o) != **null**;
630. }
631. **public** **void** clear() {
632. HashMap.**this**.clear();
633. }
634. }
636. // 返回“value集合”，实际上返回的是一个Values对象
637. **public** Collection<V> values() {
638. Collection<V> vs = values;
639. **return** (vs != **null** ? vs : (values = **new** Values()));
640. }
642. // “value集合”
643. // Values继承于AbstractCollection，不同于“KeySet继承于AbstractSet”，
644. // Values中的元素能够重复。因为不同的key可以指向相同的value。
645. **private** **final** **class** Values **extends** AbstractCollection<V> {
646. **public** Iterator<V> iterator() {
647. **return** newValueIterator();
648. }
649. **public** **int** size() {
650. **return** size;
651. }
652. **public** **boolean** contains(Object o) {
653. **return** containsValue(o);
654. }
655. **public** **void** clear() {
656. HashMap.**this**.clear();
657. }
658. }
660. // 返回“HashMap的Entry集合”
661. **public** Set<Map.Entry<K,V>> entrySet() {
662. **return** entrySet0();
663. }
665. // 返回“HashMap的Entry集合”，它实际是返回一个EntrySet对象
666. **private** Set<Map.Entry<K,V>> entrySet0() {
667. Set<Map.Entry<K,V>> es = entrySet;
668. **return** es != **null** ? es : (entrySet = **new** EntrySet());
669. }
671. // EntrySet对应的集合
672. // EntrySet继承于AbstractSet，说明该集合中没有重复的EntrySet。
673. **private** **final** **class** EntrySet **extends** AbstractSet<Map.Entry<K,V>> {
674. **public** Iterator<Map.Entry<K,V>> iterator() {
675. **return** newEntryIterator();
676. }
677. **public** **boolean** contains(Object o) {
678. **if** (!(o **instanceof** Map.Entry))
679. **return** **false**;
680. Map.Entry<K,V> e = (Map.Entry<K,V>) o;
681. Entry<K,V> candidate = getEntry(e.getKey());
682. **return** candidate != **null** && candidate.equals(e);
683. }
684. **public** **boolean** remove(Object o) {
685. **return** removeMapping(o) != **null**;
686. }
687. **public** **int** size() {
688. **return** size;
689. }
690. **public** **void** clear() {
691. HashMap.**this**.clear();
692. }
693. }
695. // java.io.Serializable的写入函数
696. // 将HashMap的“总的容量，实际容量，所有的Entry”都写入到输出流中
697. **private** **void** writeObject(java.io.ObjectOutputStream s)
698. **throws** IOException
699. {
700. Iterator<Map.Entry<K,V>> i =
701. (size > 0) ? entrySet0().iterator() : **null**;
703. // Write out the threshold, loadfactor, and any hidden stuff
704. s.defaultWriteObject();
706. // Write out number of buckets
707. s.writeInt(table.length);
709. // Write out size (number of Mappings)
710. s.writeInt(size);
712. // Write out keys and values (alternating)
713. **if** (i != **null**) {
714. **while** (i.hasNext()) {
715. Map.Entry<K,V> e = i.next();
716. s.writeObject(e.getKey());
717. s.writeObject(e.getValue());
718. }
719. }
720. }

723. **private** **static** **final** **long** serialVersionUID = 362498820763181265L;
725. // java.io.Serializable的读取函数：根据写入方式读出
726. // 将HashMap的“总的容量，实际容量，所有的Entry”依次读出
727. **private** **void** readObject(java.io.ObjectInputStream s)
728. **throws** IOException, ClassNotFoundException
729. {
730. // Read in the threshold, loadfactor, and any hidden stuff
731. s.defaultReadObject();
733. // Read in number of buckets and allocate the bucket array;
734. **int** numBuckets = s.readInt();
735. table = **new** Entry[numBuckets];
737. init();  // Give subclass a chance to do its thing.
739. // Read in size (number of Mappings)
740. **int** size = s.readInt();
742. // Read the keys and values, and put the mappings in the HashMap
743. **for** (**int** i=0; i<size; i++) {
744. K key = (K) s.readObject();
745. V value = (V) s.readObject();
746. putForCreate(key, value);
747. }
748. }
750. // 返回“HashMap总的容量”
751. **int**   capacity()     { **return** table.length; }
752. // 返回“HashMap的加载因子”
753. **float** loadFactor()   { **return** loadFactor;   }
754. }

# 几点总结

    1、首先要清楚HashMap的存储结构，如下图所示：



    图中，紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中。

    2、首先看链表中节点的数据结构：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36034955) [copy](http://blog.csdn.net/ns_code/article/details/36034955)

1. // Entry是单向链表。
2. // 它是 “HashMap链式存储法”对应的链表。
3. // 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Object o), hashCode()这些函数
4. **static** **class** Entry<K,V> **implements** Map.Entry<K,V> {
5. **final** K key;
6. V value;
7. // 指向下一个节点
8. Entry<K,V> next;
9. **final** **int** hash;
11. // 构造函数。
12. // 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"
13. Entry(**int** h, K k, V v, Entry<K,V> n) {
14. value = v;
15. next = n;
16. key = k;
17. hash = h;
18. }
20. **public** **final** K getKey() {
21. **return** key;
22. }
24. **public** **final** V getValue() {
25. **return** value;
26. }
28. **public** **final** V setValue(V newValue) {
29. V oldValue = value;
30. value = newValue;
31. **return** oldValue;
32. }
34. // 判断两个Entry是否相等
35. // 若两个Entry的“key”和“value”都相等，则返回true。
36. // 否则，返回false
37. **public** **final** **boolean** equals(Object o) {
38. **if** (!(o **instanceof** Map.Entry))
39. **return** **false**;
40. Map.Entry e = (Map.Entry)o;
41. Object k1 = getKey();
42. Object k2 = e.getKey();
43. **if** (k1 == k2 || (k1 != **null** && k1.equals(k2))) {
44. Object v1 = getValue();
45. Object v2 = e.getValue();
46. **if** (v1 == v2 || (v1 != **null** && v1.equals(v2)))
47. **return** **true**;
48. }
49. **return** **false**;
50. }
52. // 实现hashCode()
53. **public** **final** **int** hashCode() {
54. **return** (key==**null**   ? 0 : key.hashCode()) ^
55. (value==**null** ? 0 : value.hashCode());
56. }
58. **public** **final** String toString() {
59. **return** getKey() + "=" + getValue();
60. }
62. // 当向HashMap中添加元素时，绘调用recordAccess()。
63. // 这里不做任何处理
64. **void** recordAccess(HashMap<K,V> m) {
65. }
67. // 当从HashMap中删除元素时，绘调用recordRemoval()。
68. // 这里不做任何处理
69. **void** recordRemoval(HashMap<K,V> m) {
70. }
71. }

    它的结构元素除了key、value、hash外，还有next，next指向下一个节点。另外，这里覆写了equals和hashCode方法来保证键值对的独一无二。

    3、HashMap共有四个构造方法。构造方法中提到了两个很重要的参数：初始容量和加载因子。这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中槽的数量（即哈希数组的长度），初始容量是创建哈希表时的容量（从构造函数中可以看出，如果不指明，则默认为16），加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 resize 操作（即扩容）。

    下面说下加载因子，如果加载因子越大，对空间的利用更充分，但是查找效率会降低（链表长度会越来越长）；如果加载因子太小，那么表中的数据将过于稀疏（很多空间还没用，就开始扩容了），对空间造成严重浪费。如果我们在构造方法中不指定，则系统默认加载因子为0.75，这是一个比较理想的值，一般情况下我们是无需修改的。

    另外，无论我们指定的容量为多少，构造方法都会将实际容量设为不小于指定容量的2的次方的一个数，且最大值不能超过2的30次方

    4、HashMap中key和value都允许为null。

    5、要重点分析下HashMap中用的最多的两个方法put和get。先从比较简单的get方法着手，源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36034955) [copy](http://blog.csdn.net/ns_code/article/details/36034955)

1. // 获取key对应的value
2. **public** V get(Object key) {
3. **if** (key == **null**)
4. **return** getForNullKey();
5. // 获取key的hash值
6. **int** hash = hash(key.hashCode());
7. // 在“该hash值对应的链表”上查找“键值等于key”的元素
8. **for** (Entry<K,V> e = table[indexFor(hash, table.length)];
9. e != **null**;
10. e = e.next) {
11. Object k;
12. /判断key是否相同
13. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k)))
14. **return** e.value;
15. }
16. 没找到则返回**null**
17. **return** **null**;
18. }
20. // 获取“key为null”的元素的值
21. // HashMap将“key为null”的元素存储在table[0]位置，但不一定是该链表的第一个位置！
22. **private** V getForNullKey() {
23. **for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {
24. **if** (e.key == **null**)
25. **return** e.value;
26. }
27. **return** **null**;
28. }

    首先，如果key为null，则直接从哈希表的第一个位置table[0]对应的链表上查找。记住，key为null的键值对永远都放在以table[0]为头结点的链表中，当然不一定是存放在头结点table[0]中。

    如果key不为null，则先求的key的hash值，根据hash值找到在table中的索引，在该索引对应的单链表中查找是否有键值对的key与目标key相等，有就返回对应的value，没有则返回null。

    put方法稍微复杂些，代码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36034955) [copy](http://blog.csdn.net/ns_code/article/details/36034955)

1. // 将“key-value”添加到HashMap中
2. **public** V put(K key, V value) {
3. // 若“key为null”，则将该键值对添加到table[0]中。
4. **if** (key == **null**)
5. **return** putForNullKey(value);
6. // 若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。
7. **int** hash = hash(key.hashCode());
8. **int** i = indexFor(hash, table.length);
9. **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {
10. Object k;
11. // 若“该key”对应的键值对已经存在，则用新的value取代旧的value。然后退出！
12. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
13. V oldValue = e.value;
14. e.value = value;
15. e.recordAccess(**this**);
16. **return** oldValue;
17. }
18. }
20. // 若“该key”对应的键值对不存在，则将“key-value”添加到table中
21. modCount++;
22. //将key-value添加到table[i]处
23. addEntry(hash, key, value, i);
24. **return** **null**;
25. }

    如果key为null，则将其添加到table[0]对应的链表中，putForNullKey的源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36034955) [copy](http://blog.csdn.net/ns_code/article/details/36034955)

1. // putForNullKey()的作用是将“key为null”键值对添加到table[0]位置
2. **private** V putForNullKey(V value) {
3. **for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {
4. **if** (e.key == **null**) {
5. V oldValue = e.value;
6. e.value = value;
7. e.recordAccess(**this**);
8. **return** oldValue;
9. }
10. }
11. // 如果没有存在key为null的键值对，则直接题阿见到table[0]处!
12. modCount++;
13. addEntry(0, **null**, value, 0);
14. **return** **null**;
15. }

    如果key不为null，则同样先求出key的hash值，根据hash值得出在table中的索引，而后遍历对应的单链表，如果单链表中存在与目标key相等的键值对，则将新的value覆盖旧的value，比将旧的value返回，如果找不到与目标key相等的键值对，或者该单链表为空，则将该键值对插入到改单链表的头结点位置（每次新插入的节点都是放在头结点的位置），该操作是有addEntry方法实现的，它的源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36034955) [copy](http://blog.csdn.net/ns_code/article/details/36034955)

1. // 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。
2. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
3. // 保存“bucketIndex”位置的值到“e”中
4. Entry<K,V> e = table[bucketIndex];
5. // 设置“bucketIndex”位置的元素为“新Entry”，
6. // 设置“e”为“新Entry的下一个节点”
7. table[bucketIndex] = **new** Entry<K,V>(hash, key, value, e);
8. // 若HashMap的实际大小 不小于 “阈值”，则调整HashMap的大小
9. **if** (size++ >= threshold)
10. resize(2 \* table.length);
11. }

    注意这里倒数第三行的构造方法，将key-value键值对赋给table[bucketIndex]，并将其next指向元素e，这便将key-value放到了头结点中，并将之前的头结点接在了它的后面。该方法也说明，每次put键值对的时候，总是将新的该键值对放在table[bucketIndex]处（即头结点处）。

    两外注意最后两行代码，每次加入键值对时，都要判断当前已用的槽的数目是否大于等于阀值（容量\*加载因子），如果大于等于，则进行扩容，将容量扩为原来容量的2倍。

    6、关于扩容。上面我们看到了扩容的方法，resize方法，它的源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36034955) [copy](http://blog.csdn.net/ns_code/article/details/36034955)

1. // 重新调整HashMap的大小，newCapacity是调整后的单位
2. **void** resize(**int** newCapacity) {
3. Entry[] oldTable = table;
4. **int** oldCapacity = oldTable.length;
5. **if** (oldCapacity == MAXIMUM\_CAPACITY) {
6. threshold = Integer.MAX\_VALUE;
7. **return**;
8. }
10. // 新建一个HashMap，将“旧HashMap”的全部元素添加到“新HashMap”中，
11. // 然后，将“新HashMap”赋值给“旧HashMap”。
12. Entry[] newTable = **new** Entry[newCapacity];
13. transfer(newTable);
14. table = newTable;
15. threshold = (**int**)(newCapacity \* loadFactor);
16. }

    很明显，是新建了一个HashMap的底层数组，而后调用transfer方法，将就HashMap的全部元素添加到新的HashMap中（要重新计算元素在新的数组中的索引位置）。transfer方法的源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36034955) [copy](http://blog.csdn.net/ns_code/article/details/36034955)

1. // 将HashMap中的全部元素都添加到newTable中
2. **void** transfer(Entry[] newTable) {
3. Entry[] src = table;
4. **int** newCapacity = newTable.length;
5. **for** (**int** j = 0; j < src.length; j++) {
6. Entry<K,V> e = src[j];
7. **if** (e != **null**) {
8. src[j] = **null**;
9. **do** {
10. Entry<K,V> next = e.next;
11. **int** i = indexFor(e.hash, newCapacity);
12. e.next = newTable[i];
13. newTable[i] = e;
14. e = next;
15. } **while** (e != **null**);
16. }
17. }
18. }

    很明显，扩容是一个相当耗时的操作，因为它需要重新计算这些元素在新的数组中的位置并进行复制处理。因此，我们在用HashMap的时，最好能提前预估下HashMap中元素的个数，这样有助于提高HashMap的性能。

    7、注意containsKey方法和containsValue方法。前者直接可以通过key的哈希值将搜索范围定位到指定索引对应的链表，而后者要对哈希数组的每个链表进行搜索。

    8、我们重点来分析下求hash值和索引值的方法，这两个方法便是HashMap设计的最为核心的部分，二者结合能保证哈希表中的元素尽可能均匀地散列。

    计算哈希值的方法如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36034955) [copy](http://blog.csdn.net/ns_code/article/details/36034955)

1. **static** **int** hash(**int** h) {
2. h ^= (h >>> 20) ^ (h >>> 12);
3. **return** h ^ (h >>> 7) ^ (h >>> 4);
4. }

    它只是一个数学公式，IDK这样设计对hash值的计算，自然有它的好处，至于为什么这样设计，我们这里不去追究，只要明白一点，用的位的操作使hash值的计算效率很高。

    由hash值找到对应索引的方法如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36034955) [copy](http://blog.csdn.net/ns_code/article/details/36034955)

1. **static** **int** indexFor(**int** h, **int** length) {
2. **return** h & (length-1);
3. }

    这个我们要重点说下，我们一般对哈希表的散列很自然地会想到用hash值对length取模（即除法散列法），Hashtable中也是这样实现的，这种方法基本能保证元素在哈希表中散列的比较均匀，但取模会用到除法运算，效率很低，HashMap中则通过h&(length-1)的方法来代替取模，同样实现了均匀的散列，但效率要高很多，这也是HashMap对Hashtable的一个改进。

    接下来，我们分析下为什么哈希表的容量一定要是2的整数次幂。首先，length为2的整数次幂的话，h&(length-1)就相当于对length取模，这样便保证了散列的均匀，同时也提升了效率；其次，length为2的整数次幂的话，为偶数，这样length-1为奇数，奇数的最后一位是1，这样便保证了h&(length-1)的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性，而如果length为奇数的话，很明显length-1为偶数，它的最后一位是0，这样h&(length-1)的最后一位肯定为0，即只能为偶数，这样任何hash值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间，因此，length取2的整数次幂，是为了使不同hash值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。

# LinkedHashMap简介

    LinkedHashMap是HashMap的子类，与HashMap有着同样的存储结构，但它加入了一个双向链表的头结点，将所有put到LinkedHashmap的节点一一串成了一个双向循环链表，因此它保留了节点插入的顺序，可以使节点的输出顺序与输入顺序相同。

    LinkedHashMap可以用来实现LRU算法（这会在下面的源码中进行分析）。

    LinkedHashMap同样是非线程安全的，只在单线程环境下使用。

# LinkedHashMap源码剖析

    LinkedHashMap源码如下（加入了详细的注释）：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/37867985) [copy](http://blog.csdn.net/ns_code/article/details/37867985)

1. **package** java.util;
2. **import** java.io.\*;

5. **public** **class** LinkedHashMap<K,V>
6. **extends** HashMap<K,V>
7. **implements** Map<K,V>
8. {
10. **private** **static** **final** **long** serialVersionUID = 3801124242820219131L;
12. //双向循环链表的头结点，整个LinkedHa只哟shMap中只有一个header，
13. //它将哈希表中所有的Entry贯穿起来，header中不保存key-value对，只保存前后节点的引用
14. **private** **transient** Entry<K,V> header;
16. //双向链表中元素排序规则的标志位。
17. //accessOrder为false，表示按插入顺序排序
18. //accessOrder为true，表示按访问顺序排序
19. **private** **final** **boolean** accessOrder;
21. //调用HashMap的构造方法来构造底层的数组
22. **public** LinkedHashMap(**int** initialCapacity, **float** loadFactor) {
23. **super**(initialCapacity, loadFactor);
24. accessOrder = **false**;    //链表中的元素默认按照插入顺序排序
25. }
27. //加载因子取默认的0.75f
28. **public** LinkedHashMap(**int** initialCapacity) {
29. **super**(initialCapacity);
30. accessOrder = **false**;
31. }
33. //加载因子取默认的0.75f，容量取默认的16
34. **public** LinkedHashMap() {
35. **super**();
36. accessOrder = **false**;
37. }
39. //含有子Map的构造方法，同样调用HashMap的对应的构造方法
40. **public** LinkedHashMap(Map<? **extends** K, ? **extends** V> m) {
41. **super**(m);
42. accessOrder = **false**;
43. }
45. //该构造方法可以指定链表中的元素排序的规则
46. **public** LinkedHashMap(**int** initialCapacity,**float** loadFactor,**boolean** accessOrder) {
47. **super**(initialCapacity, loadFactor);
48. **this**.accessOrder = accessOrder;
49. }
51. //覆写父类的init()方法（HashMap中的init方法为空），
52. //该方法在父类的构造方法和Clone、readObject中在插入元素前被调用，
53. //初始化一个空的双向循环链表，头结点中不保存数据，头结点的下一个节点才开始保存数据。
54. **void** init() {
55. header = **new** Entry<K,V>(-1, **null**, **null**, **null**);
56. header.before = header.after = header;
57. }

60. //覆写HashMap中的transfer方法，它在父类的resize方法中被调用，
61. //扩容后，将key-value对重新映射到新的newTable中
62. //覆写该方法的目的是为了提高复制的效率，
63. //这里充分利用双向循环链表的特点进行迭代，不用对底层的数组进行for循环。
64. **void** transfer(HashMap.Entry[] newTable) {
65. **int** newCapacity = newTable.length;
66. **for** (Entry<K,V> e = header.after; e != header; e = e.after) {
67. **int** index = indexFor(e.hash, newCapacity);
68. e.next = newTable[index];
69. newTable[index] = e;
70. }
71. }

74. //覆写HashMap中的containsValue方法，
75. //覆写该方法的目的同样是为了提高查询的效率，
76. //利用双向循环链表的特点进行查询，少了对数组的外层for循环
77. **public** **boolean** containsValue(Object value) {
78. // Overridden to take advantage of faster iterator
79. **if** (value==**null**) {
80. **for** (Entry e = header.after; e != header; e = e.after)
81. **if** (e.value==**null**)
82. **return** **true**;
83. } **else** {
84. **for** (Entry e = header.after; e != header; e = e.after)
85. **if** (value.equals(e.value))
86. **return** **true**;
87. }
88. **return** **false**;
89. }

92. //覆写HashMap中的get方法，通过getEntry方法获取Entry对象。
93. //注意这里的recordAccess方法，
94. //如果链表中元素的排序规则是按照插入的先后顺序排序的话，该方法什么也不做，
95. //如果链表中元素的排序规则是按照访问的先后顺序排序的话，则将e移到链表的末尾处。
96. **public** V get(Object key) {
97. Entry<K,V> e = (Entry<K,V>)getEntry(key);
98. **if** (e == **null**)
99. **return** **null**;
100. e.recordAccess(**this**);
101. **return** e.value;
102. }
104. //清空HashMap，并将双向链表还原为只有头结点的空链表
105. **public** **void** clear() {
106. **super**.clear();
107. header.before = header.after = header;
108. }
110. //Enty的数据结构，多了两个指向前后节点的引用
111. **private** **static** **class** Entry<K,V> **extends** HashMap.Entry<K,V> {
112. // These fields comprise the doubly linked list used for iteration.
113. Entry<K,V> before, after;
115. //调用父类的构造方法
116. Entry(**int** hash, K key, V value, HashMap.Entry<K,V> next) {
117. **super**(hash, key, value, next);
118. }
120. //双向循环链表中，删除当前的Entry
121. **private** **void** remove() {
122. before.after = after;
123. after.before = before;
124. }
126. //双向循环立链表中，将当前的Entry插入到existingEntry的前面
127. **private** **void** addBefore(Entry<K,V> existingEntry) {
128. after  = existingEntry;
129. before = existingEntry.before;
130. before.after = **this**;
131. after.before = **this**;
132. }

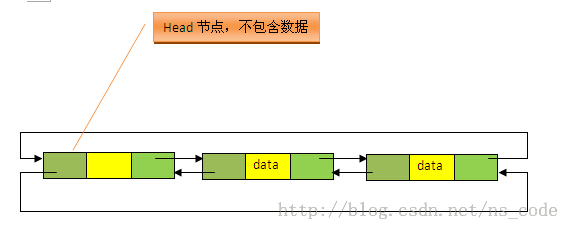
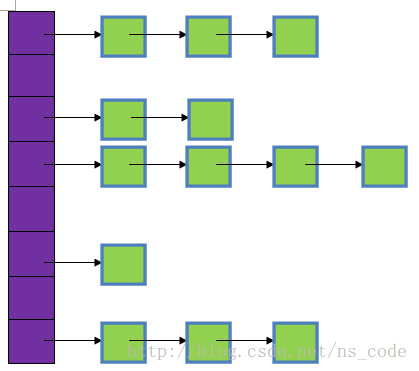
135. //覆写HashMap中的recordAccess方法（HashMap中该方法为空），
136. //当调用父类的put方法，在发现插入的key已经存在时，会调用该方法，
137. //调用LinkedHashmap覆写的get方法时，也会调用到该方法，
138. //该方法提供了LRU算法的实现，它将最近使用的Entry放到双向循环链表的尾部，
139. //accessOrder为true时，get方法会调用recordAccess方法
140. //put方法在覆盖key-value对时也会调用recordAccess方法
141. //它们导致Entry最近使用，因此将其移到双向链表的末尾
142. **void** recordAccess(HashMap<K,V> m) {
143. LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
144. //如果链表中元素按照访问顺序排序，则将当前访问的Entry移到双向循环链表的尾部，
145. //如果是按照插入的先后顺序排序，则不做任何事情。
146. **if** (lm.accessOrder) {
147. lm.modCount++;
148. //移除当前访问的Entry
149. remove();
150. //将当前访问的Entry插入到链表的尾部
151. addBefore(lm.header);
152. }
153. }
155. **void** recordRemoval(HashMap<K,V> m) {
156. remove();
157. }
158. }
160. //迭代器
161. **private** **abstract** **class** LinkedHashIterator<T> **implements** Iterator<T> {
162. Entry<K,V> nextEntry    = header.after;
163. Entry<K,V> lastReturned = **null**;
165. /\*\*
166. \* The modCount value that the iterator believes that the backing
167. \* List should have.  If this expectation is violated, the iterator
168. \* has detected concurrent modification.
169. \*/
170. **int** expectedModCount = modCount;
172. **public** **boolean** hasNext() {
173. **return** nextEntry != header;
174. }
176. **public** **void** remove() {
177. **if** (lastReturned == **null**)
178. **throw** **new** IllegalStateException();
179. **if** (modCount != expectedModCount)
180. **throw** **new** ConcurrentModificationException();
182. LinkedHashMap.**this**.remove(lastReturned.key);
183. lastReturned = **null**;
184. expectedModCount = modCount;
185. }
187. //从head的下一个节点开始迭代
188. Entry<K,V> nextEntry() {
189. **if** (modCount != expectedModCount)
190. **throw** **new** ConcurrentModificationException();
191. **if** (nextEntry == header)
192. **throw** **new** NoSuchElementException();
194. Entry<K,V> e = lastReturned = nextEntry;
195. nextEntry = e.after;
196. **return** e;
197. }
198. }
200. //key迭代器
201. **private** **class** KeyIterator **extends** LinkedHashIterator<K> {
202. **public** K next() { **return** nextEntry().getKey(); }
203. }
205. //value迭代器
206. **private** **class** ValueIterator **extends** LinkedHashIterator<V> {
207. **public** V next() { **return** nextEntry().value; }
208. }
210. //Entry迭代器
211. **private** **class** EntryIterator **extends** LinkedHashIterator<Map.Entry<K,V>> {
212. **public** Map.Entry<K,V> next() { **return** nextEntry(); }
213. }
215. // These Overrides alter the behavior of superclass view iterator() methods
216. Iterator<K> newKeyIterator()   { **return** **new** KeyIterator();   }
217. Iterator<V> newValueIterator() { **return** **new** ValueIterator(); }
218. Iterator<Map.Entry<K,V>> newEntryIterator() { **return** **new** EntryIterator(); }

221. //覆写HashMap中的addEntry方法，LinkedHashmap并没有覆写HashMap中的put方法，
222. //而是覆写了put方法所调用的addEntry方法和recordAccess方法，
223. //put方法在插入的key已存在的情况下，会调用recordAccess方法，
224. //在插入的key不存在的情况下，要调用addEntry插入新的Entry
225. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
226. //创建新的Entry，并插入到LinkedHashMap中
227. createEntry(hash, key, value, bucketIndex);
229. //双向链表的第一个有效节点（header后的那个节点）为近期最少使用的节点
230. Entry<K,V> eldest = header.after;
231. //如果有必要，则删除掉该近期最少使用的节点，
232. //这要看对removeEldestEntry的覆写,由于默认为false，因此默认是不做任何处理的。
233. **if** (removeEldestEntry(eldest)) {
234. removeEntryForKey(eldest.key);
235. } **else** {
236. //扩容到原来的2倍
237. **if** (size >= threshold)
238. resize(2 \* table.length);
239. }
240. }
242. **void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {
243. //创建新的Entry，并将其插入到数组对应槽的单链表的头结点处，这点与HashMap中相同
244. HashMap.Entry<K,V> old = table[bucketIndex];
245. Entry<K,V> e = **new** Entry<K,V>(hash, key, value, old);
246. table[bucketIndex] = e;
247. //每次插入Entry时，都将其移到双向链表的尾部，
248. //这便会按照Entry插入LinkedHashMap的先后顺序来迭代元素，
249. //同时，新put进来的Entry是最近访问的Entry，把其放在链表末尾 ，符合LRU算法的实现
250. e.addBefore(header);
251. size++;
252. }
254. //该方法是用来被覆写的，一般如果用LinkedHashmap实现LRU算法，就要覆写该方法，
255. //比如可以将该方法覆写为如果设定的内存已满，则返回true，这样当再次向LinkedHashMap中put
256. //Entry时，在调用的addEntry方法中便会将近期最少使用的节点删除掉（header后的那个节点）。
257. **protected** **boolean** removeEldestEntry(Map.Entry<K,V> eldest) {
258. **return** **false**;
259. }
260. }

# 几点总结

    关于LinkedHashMap的源码，给出以下几点比较重要的总结：

    1、从源码中可以看出，LinkedHashMap中加入了一个head头结点，将所有插入到该LinkedHashMap中的Entry按照插入的先后顺序依次加入到以head为头结点的双向循环链表的尾部。



    实际上就是HashMap和LinkedList两个集合类的存储结构的结合。在LinkedHashMapMap中，所有put进来的Entry都保存在如第一个图所示的哈希表中，但它又额外定义了一个以head为头结点的空的双向循环链表，每次put进来Entry，除了将其保存到对哈希表中对应的位置上外，还要将其插入到双向循环链表的尾部。

    2、LinkedHashMap由于继承自HashMap，因此它具有HashMap的所有特性，同样允许key和value为null。

    3、注意源码中的accessOrder标志位，当它false时，表示双向链表中的元素按照Entry插入LinkedHashMap到中的先后顺序排序，即每次put到LinkedHashMap中的Entry都放在双向链表的尾部，这样遍历双向链表时，Entry的输出顺序便和插入的顺序一致，这也是默认的双向链表的存储顺序；当它为true时，表示双向链表中的元素按照访问的先后顺序排列，可以看到，虽然Entry插入链表的顺序依然是按照其put到LinkedHashMap中的顺序，但put和get方法均有调用recordAccess方法（put方法在key相同，覆盖原有的Entry的情况下调用recordAccess方法），该方法判断accessOrder是否为true，如果是，则将当前访问的Entry（put进来的Entry或get出来的Entry）移到双向链表的尾部（key不相同时，put新Entry时，会调用addEntry，它会调用creatEntry，该方法同样将新插入的元素放入到双向链表的尾部，既符合插入的先后顺序，又符合访问的先后顺序，因为这时该Entry也被访问了），否则，什么也不做。

    4、注意构造方法，前四个构造方法都将accessOrder设为false，说明默认是按照插入顺序排序的，而第五个构造方法可以自定义传入的accessOrder的值，因此可以指定双向循环链表中元素的排序规则，一般要用LinkedHashMap实现LRU算法，就要用该构造方法，将accessOrder置为true。

    5、LinkedHashMap并没有覆写HashMap中的put方法，而是覆写了put方法中调用的addEntry方法和recordAccess方法，我们回过头来再看下HashMap的put方法：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/37867985) [copy](http://blog.csdn.net/ns_code/article/details/37867985)

1. // 将“key-value”添加到HashMap中
2. **public** V put(K key, V value) {
3. // 若“key为null”，则将该键值对添加到table[0]中。
4. **if** (key == **null**)
5. **return** putForNullKey(value);
6. // 若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。
7. **int** hash = hash(key.hashCode());
8. **int** i = indexFor(hash, table.length);
9. **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {
10. Object k;
11. // 若“该key”对应的键值对已经存在，则用新的value取代旧的value。然后退出！
12. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
13. V oldValue = e.value;
14. e.value = value;
15. e.recordAccess(**this**);
16. **return** oldValue;
17. }
18. }
20. // 若“该key”对应的键值对不存在，则将“key-value”添加到table中
21. modCount++;
22. //将key-value添加到table[i]处
23. addEntry(hash, key, value, i);
24. **return** **null**;
25. }

    当要put进来的Entry的key在哈希表中已经在存在时，会调用recordAccess方法，当该key不存在时，则会调用addEntry方法将新的Entry插入到对应槽的单链表的头部。

    我们先来看recordAccess方法：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/37867985) [copy](http://blog.csdn.net/ns_code/article/details/37867985)

1. //覆写HashMap中的recordAccess方法（HashMap中该方法为空），
2. //当调用父类的put方法，在发现插入的key已经存在时，会调用该方法，
3. //调用LinkedHashmap覆写的get方法时，也会调用到该方法，
4. //该方法提供了LRU算法的实现，它将最近使用的Entry放到双向循环链表的尾部，
5. //accessOrder为true时，get方法会调用recordAccess方法
6. //put方法在覆盖key-value对时也会调用recordAccess方法
7. //它们导致Entry最近使用，因此将其移到双向链表的末尾
8. **void** recordAccess(HashMap<K,V> m) {
9. LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
10. //如果链表中元素按照访问顺序排序，则将当前访问的Entry移到双向循环链表的尾部，
11. //如果是按照插入的先后顺序排序，则不做任何事情。
12. **if** (lm.accessOrder) {
13. lm.modCount++;
14. //移除当前访问的Entry
15. remove();
16. //将当前访问的Entry插入到链表的尾部
17. addBefore(lm.header);
18. }
19. }

    该方法会判断accessOrder是否为true，如果为true，它会将当前访问的Entry（在这里指put进来的Entry）移动到双向循环链表的尾部，从而实现双向链表中的元素按照访问顺序来排序（最近访问的Entry放到链表的最后，这样多次下来，前面就是最近没有被访问的元素，在实现、LRU算法时，当双向链表中的节点数达到最大值时，将前面的元素删去即可，因为前面的元素是最近最少使用的），否则什么也不做。  
    再来看addEntry方法：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/37867985) [copy](http://blog.csdn.net/ns_code/article/details/37867985)

1. //覆写HashMap中的addEntry方法，LinkedHashmap并没有覆写HashMap中的put方法，
2. //而是覆写了put方法所调用的addEntry方法和recordAccess方法，
3. //put方法在插入的key已存在的情况下，会调用recordAccess方法，
4. //在插入的key不存在的情况下，要调用addEntry插入新的Entry
5. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
6. //创建新的Entry，并插入到LinkedHashMap中
7. createEntry(hash, key, value, bucketIndex);
9. //双向链表的第一个有效节点（header后的那个节点）为近期最少使用的节点
10. Entry<K,V> eldest = header.after;
11. //如果有必要，则删除掉该近期最少使用的节点，
12. //这要看对removeEldestEntry的覆写,由于默认为false，因此默认是不做任何处理的。
13. **if** (removeEldestEntry(eldest)) {
14. removeEntryForKey(eldest.key);
15. } **else** {
16. //扩容到原来的2倍
17. **if** (size >= threshold)
18. resize(2 \* table.length);
19. }
20. }
22. **void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {
23. //创建新的Entry，并将其插入到数组对应槽的单链表的头结点处，这点与HashMap中相同
24. HashMap.Entry<K,V> old = table[bucketIndex];
25. Entry<K,V> e = **new** Entry<K,V>(hash, key, value, old);
26. table[bucketIndex] = e;
27. //每次插入Entry时，都将其移到双向链表的尾部，
28. //这便会按照Entry插入LinkedHashMap的先后顺序来迭代元素，
29. //同时，新put进来的Entry是最近访问的Entry，把其放在链表末尾 ，符合LRU算法的实现
30. e.addBefore(header);
31. size++;
32. }

    同样是将新的Entry插入到table中对应槽所对应单链表的头结点中，但可以看出，在createEntry中，同样把新put进来的Entry插入到了双向链表的尾部，从插入顺序的层面来说，新的Entry插入到双向链表的尾部，可以实现按照插入的先后顺序来迭代Entry，而从访问顺序的层面来说，新put进来的Entry又是最近访问的Entry，也应该将其放在双向链表的尾部。

    上面还有个removeEldestEntry方法，该方法如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/37867985) [copy](http://blog.csdn.net/ns_code/article/details/37867985)

1. //该方法是用来被覆写的，一般如果用LinkedHashmap实现LRU算法，就要覆写该方法，
2. //比如可以将该方法覆写为如果设定的内存已满，则返回true，这样当再次向LinkedHashMap中put
3. //Entry时，在调用的addEntry方法中便会将近期最少使用的节点删除掉（header后的那个节点）。
4. **protected** **boolean** removeEldestEntry(Map.Entry<K,V> eldest) {
5. **return** **false**;
6. }
7. }

    该方法默认返回false，我们一般在用LinkedHashMap实现LRU算法时，要覆写该方法，一般的实现是，当设定的内存（这里指节点个数）达到最大值时，返回true，这样put新的Entry（该Entry的key在哈希表中没有已经存在）时，就会调用removeEntryForKey方法，将最近最少使用的节点删除（head后面的那个节点，实际上是最近没有使用）。  
    6、LinkedHashMap覆写了HashMap的get方法：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/37867985) [copy](http://blog.csdn.net/ns_code/article/details/37867985)

1. //覆写HashMap中的get方法，通过getEntry方法获取Entry对象。
2. //注意这里的recordAccess方法，
3. //如果链表中元素的排序规则是按照插入的先后顺序排序的话，该方法什么也不做，
4. //如果链表中元素的排序规则是按照访问的先后顺序排序的话，则将e移到链表的末尾处。
5. **public** V get(Object key) {
6. Entry<K,V> e = (Entry<K,V>)getEntry(key);
7. **if** (e == **null**)
8. **return** **null**;
9. e.recordAccess(**this**);
10. **return** e.value;
11. }

    先取得Entry，如果不为null，一样调用recordAccess方法，上面已经说得很清楚，这里不在多解释了。

7、最后说说LinkedHashMap是如何实现LRU的。首先，当accessOrder为true时，才会开启按访问顺序排序的模式，才能用来实现LRU算法。我们可以看到，无论是put方法还是get方法，都会导致目标Entry成为最近访问的Entry，因此便把该Entry加入到了双向链表的末尾（get方法通过调用recordAccess方法来实现，put方法在覆盖已有key的情况下，也是通过调用recordAccess方法来实现，在插入新的Entry时，则是通过createEntry中的addBefore方法来实现），这样便把最近使用了的Entry放入到了双向链表的后面，多次操作后，双向链表前面的Entry便是最近没有使用的，这样当节点个数满的时候，删除的最前面的Entry(head后面的那个Entry)便是最近最少使用的Entry。

# ****Hashtable简介****

    Hashtable同样是基于哈希表实现的，同样每个元素是一个key-value对，其内部也是通过单链表解决冲突问题，容量不足（超过了阀值）时，同样会自动增长。

    Hashtable也是JDK1.0引入的类，是线程安全的，能用于多线程环境中。

    Hashtable同样实现了Serializable接口，它支持序列化，实现了Cloneable接口，能被克隆。

# HashTable源码剖析

    Hashtable的源码的很多实现都与HashMap差不多，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/36191279) [copy](http://blog.csdn.net/ns_code/article/details/36191279)

1. **package** java.util;
2. **import** java.io.\*;
4. **public** **class** Hashtable<K,V>
5. **extends** Dictionary<K,V>
6. **implements** Map<K,V>, Cloneable, java.io.Serializable {
8. // 保存key-value的数组。
9. // Hashtable同样采用单链表解决冲突，每一个Entry本质上是一个单向链表
10. **private** **transient** Entry[] table;
12. // Hashtable中键值对的数量
13. **private** **transient** **int** count;
15. // 阈值，用于判断是否需要调整Hashtable的容量（threshold = 容量\*加载因子）
16. **private** **int** threshold;
18. // 加载因子
19. **private** **float** loadFactor;
21. // Hashtable被改变的次数，用于fail-fast机制的实现
22. **private** **transient** **int** modCount = 0;
24. // 序列版本号
25. **private** **static** **final** **long** serialVersionUID = 1421746759512286392L;
27. // 指定“容量大小”和“加载因子”的构造函数
28. **public** Hashtable(**int** initialCapacity, **float** loadFactor) {
29. **if** (initialCapacity < 0)
30. **throw** **new** IllegalArgumentException("Illegal Capacity: "+
31. initialCapacity);
32. **if** (loadFactor <= 0 || Float.isNaN(loadFactor))
33. **throw** **new** IllegalArgumentException("Illegal Load: "+loadFactor);
35. **if** (initialCapacity==0)
36. initialCapacity = 1;
37. **this**.loadFactor = loadFactor;
38. table = **new** Entry[initialCapacity];
39. threshold = (**int**)(initialCapacity \* loadFactor);
40. }
42. // 指定“容量大小”的构造函数
43. **public** Hashtable(**int** initialCapacity) {
44. **this**(initialCapacity, 0.75f);
45. }
47. // 默认构造函数。
48. **public** Hashtable() {
49. // 默认构造函数，指定的容量大小是11；加载因子是0.75
50. **this**(11, 0.75f);
51. }
53. // 包含“子Map”的构造函数
54. **public** Hashtable(Map<? **extends** K, ? **extends** V> t) {
55. **this**(Math.max(2\*t.size(), 11), 0.75f);
56. // 将“子Map”的全部元素都添加到Hashtable中
57. putAll(t);
58. }
60. **public** **synchronized** **int** size() {
61. **return** count;
62. }
64. **public** **synchronized** **boolean** isEmpty() {
65. **return** count == 0;
66. }
68. // 返回“所有key”的枚举对象
69. **public** **synchronized** Enumeration<K> keys() {
70. **return** **this**.<K>getEnumeration(KEYS);
71. }
73. // 返回“所有value”的枚举对象
74. **public** **synchronized** Enumeration<V> elements() {
75. **return** **this**.<V>getEnumeration(VALUES);
76. }
78. // 判断Hashtable是否包含“值(value)”
79. **public** **synchronized** **boolean** contains(Object value) {
80. //注意，Hashtable中的value不能是null，
81. // 若是null的话，抛出异常!
82. **if** (value == **null**) {
83. **throw** **new** NullPointerException();
84. }
86. // 从后向前遍历table数组中的元素(Entry)
87. // 对于每个Entry(单向链表)，逐个遍历，判断节点的值是否等于value
88. Entry tab[] = table;
89. **for** (**int** i = tab.length ; i-- > 0 ;) {
90. **for** (Entry<K,V> e = tab[i] ; e != **null** ; e = e.next) {
91. **if** (e.value.equals(value)) {
92. **return** **true**;
93. }
94. }
95. }
96. **return** **false**;
97. }
99. **public** **boolean** containsValue(Object value) {
100. **return** contains(value);
101. }
103. // 判断Hashtable是否包含key
104. **public** **synchronized** **boolean** containsKey(Object key) {
105. Entry tab[] = table;
106. //计算hash值，直接用key的hashCode代替
107. **int** hash = key.hashCode();
108. // 计算在数组中的索引值
109. **int** index = (hash & 0x7FFFFFFF) % tab.length;
110. // 找到“key对应的Entry(链表)”，然后在链表中找出“哈希值”和“键值”与key都相等的元素
111. **for** (Entry<K,V> e = tab[index] ; e != **null** ; e = e.next) {
112. **if** ((e.hash == hash) && e.key.equals(key)) {
113. **return** **true**;
114. }
115. }
116. **return** **false**;
117. }
119. // 返回key对应的value，没有的话返回null
120. **public** **synchronized** V get(Object key) {
121. Entry tab[] = table;
122. **int** hash = key.hashCode();
123. // 计算索引值，
124. **int** index = (hash & 0x7FFFFFFF) % tab.length;
125. // 找到“key对应的Entry(链表)”，然后在链表中找出“哈希值”和“键值”与key都相等的元素
126. **for** (Entry<K,V> e = tab[index] ; e != **null** ; e = e.next) {
127. **if** ((e.hash == hash) && e.key.equals(key)) {
128. **return** e.value;
129. }
130. }
131. **return** **null**;
132. }
134. // 调整Hashtable的长度，将长度变成原来的2倍+1
135. **protected** **void** rehash() {
136. **int** oldCapacity = table.length;
137. Entry[] oldMap = table;
139. //创建新容量大小的Entry数组
140. **int** newCapacity = oldCapacity \* 2 + 1;
141. Entry[] newMap = **new** Entry[newCapacity];
143. modCount++;
144. threshold = (**int**)(newCapacity \* loadFactor);
145. table = newMap;
147. //将“旧的Hashtable”中的元素复制到“新的Hashtable”中
148. **for** (**int** i = oldCapacity ; i-- > 0 ;) {
149. **for** (Entry<K,V> old = oldMap[i] ; old != **null** ; ) {
150. Entry<K,V> e = old;
151. old = old.next;
152. //重新计算index
153. **int** index = (e.hash & 0x7FFFFFFF) % newCapacity;
154. e.next = newMap[index];
155. newMap[index] = e;
156. }
157. }
158. }
160. // 将“key-value”添加到Hashtable中
161. **public** **synchronized** V put(K key, V value) {
162. // Hashtable中不能插入value为null的元素！！！
163. **if** (value == **null**) {
164. **throw** **new** NullPointerException();
165. }
167. // 若“Hashtable中已存在键为key的键值对”，
168. // 则用“新的value”替换“旧的value”
169. Entry tab[] = table;
170. **int** hash = key.hashCode();
171. **int** index = (hash & 0x7FFFFFFF) % tab.length;
172. **for** (Entry<K,V> e = tab[index] ; e != **null** ; e = e.next) {
173. **if** ((e.hash == hash) && e.key.equals(key)) {
174. V old = e.value;
175. e.value = value;
176. **return** old;
177. }
178. }
180. // 若“Hashtable中不存在键为key的键值对”，
181. // 将“修改统计数”+1
182. modCount++;
183. //  若“Hashtable实际容量” > “阈值”(阈值=总的容量 \* 加载因子)
184. //  则调整Hashtable的大小
185. **if** (count >= threshold) {
186. rehash();
188. tab = table;
189. index = (hash & 0x7FFFFFFF) % tab.length;
190. }
192. //将新的key-value对插入到tab[index]处（即链表的头结点）
193. Entry<K,V> e = tab[index];
194. tab[index] = **new** Entry<K,V>(hash, key, value, e);
195. count++;
196. **return** **null**;
197. }
199. // 删除Hashtable中键为key的元素
200. **public** **synchronized** V remove(Object key) {
201. Entry tab[] = table;
202. **int** hash = key.hashCode();
203. **int** index = (hash & 0x7FFFFFFF) % tab.length;
205. //从table[index]链表中找出要删除的节点，并删除该节点。
206. //因为是单链表，因此要保留带删节点的前一个节点，才能有效地删除节点
207. **for** (Entry<K,V> e = tab[index], prev = **null** ; e != **null** ; prev = e, e = e.next) {
208. **if** ((e.hash == hash) && e.key.equals(key)) {
209. modCount++;
210. **if** (prev != **null**) {
211. prev.next = e.next;
212. } **else** {
213. tab[index] = e.next;
214. }
215. count--;
216. V oldValue = e.value;
217. e.value = **null**;
218. **return** oldValue;
219. }
220. }
221. **return** **null**;
222. }
224. // 将“Map(t)”的中全部元素逐一添加到Hashtable中
225. **public** **synchronized** **void** putAll(Map<? **extends** K, ? **extends** V> t) {
226. **for** (Map.Entry<? **extends** K, ? **extends** V> e : t.entrySet())
227. put(e.getKey(), e.getValue());
228. }
230. // 清空Hashtable
231. // 将Hashtable的table数组的值全部设为null
232. **public** **synchronized** **void** clear() {
233. Entry tab[] = table;
234. modCount++;
235. **for** (**int** index = tab.length; --index >= 0; )
236. tab[index] = **null**;
237. count = 0;
238. }
240. // 克隆一个Hashtable，并以Object的形式返回。
241. **public** **synchronized** Object clone() {
242. **try** {
243. Hashtable<K,V> t = (Hashtable<K,V>) **super**.clone();
244. t.table = **new** Entry[table.length];
245. **for** (**int** i = table.length ; i-- > 0 ; ) {
246. t.table[i] = (table[i] != **null**)
247. ? (Entry<K,V>) table[i].clone() : **null**;
248. }
249. t.keySet = **null**;
250. t.entrySet = **null**;
251. t.values = **null**;
252. t.modCount = 0;
253. **return** t;
254. } **catch** (CloneNotSupportedException e) {
255. **throw** **new** InternalError();
256. }
257. }
259. **public** **synchronized** String toString() {
260. **int** max = size() - 1;
261. **if** (max == -1)
262. **return** "{}";
264. StringBuilder sb = **new** StringBuilder();
265. Iterator<Map.Entry<K,V>> it = entrySet().iterator();
267. sb.append('{');
268. **for** (**int** i = 0; ; i++) {
269. Map.Entry<K,V> e = it.next();
270. K key = e.getKey();
271. V value = e.getValue();
272. sb.append(key   == **this** ? "(this Map)" : key.toString());
273. sb.append('=');
274. sb.append(value == **this** ? "(this Map)" : value.toString());
276. **if** (i == max)
277. **return** sb.append('}').toString();
278. sb.append(", ");
279. }
280. }
282. // 获取Hashtable的枚举类对象
283. // 若Hashtable的实际大小为0,则返回“空枚举类”对象；
284. // 否则，返回正常的Enumerator的对象。
285. **private** <T> Enumeration<T> getEnumeration(**int** type) {
286. **if** (count == 0) {
287. **return** (Enumeration<T>)emptyEnumerator;
288. } **else** {
289. **return** **new** Enumerator<T>(type, **false**);
290. }
291. }
293. // 获取Hashtable的迭代器
294. // 若Hashtable的实际大小为0,则返回“空迭代器”对象；
295. // 否则，返回正常的Enumerator的对象。(Enumerator实现了迭代器和枚举两个接口)
296. **private** <T> Iterator<T> getIterator(**int** type) {
297. **if** (count == 0) {
298. **return** (Iterator<T>) emptyIterator;
299. } **else** {
300. **return** **new** Enumerator<T>(type, **true**);
301. }
302. }
304. // Hashtable的“key的集合”。它是一个Set，没有重复元素
305. **private** **transient** **volatile** Set<K> keySet = **null**;
306. // Hashtable的“key-value的集合”。它是一个Set，没有重复元素
307. **private** **transient** **volatile** Set<Map.Entry<K,V>> entrySet = **null**;
308. // Hashtable的“key-value的集合”。它是一个Collection，可以有重复元素
309. **private** **transient** **volatile** Collection<V> values = **null**;
311. // 返回一个被synchronizedSet封装后的KeySet对象
312. // synchronizedSet封装的目的是对KeySet的所有方法都添加synchronized，实现多线程同步
313. **public** Set<K> keySet() {
314. **if** (keySet == **null**)
315. keySet = Collections.synchronizedSet(**new** KeySet(), **this**);
316. **return** keySet;
317. }
319. // Hashtable的Key的Set集合。
320. // KeySet继承于AbstractSet，所以，KeySet中的元素没有重复的。
321. **private** **class** KeySet **extends** AbstractSet<K> {
322. **public** Iterator<K> iterator() {
323. **return** getIterator(KEYS);
324. }
325. **public** **int** size() {
326. **return** count;
327. }
328. **public** **boolean** contains(Object o) {
329. **return** containsKey(o);
330. }
331. **public** **boolean** remove(Object o) {
332. **return** Hashtable.**this**.remove(o) != **null**;
333. }
334. **public** **void** clear() {
335. Hashtable.**this**.clear();
336. }
337. }
339. // 返回一个被synchronizedSet封装后的EntrySet对象
340. // synchronizedSet封装的目的是对EntrySet的所有方法都添加synchronized，实现多线程同步
341. **public** Set<Map.Entry<K,V>> entrySet() {
342. **if** (entrySet==**null**)
343. entrySet = Collections.synchronizedSet(**new** EntrySet(), **this**);
344. **return** entrySet;
345. }
347. // Hashtable的Entry的Set集合。
348. // EntrySet继承于AbstractSet，所以，EntrySet中的元素没有重复的。
349. **private** **class** EntrySet **extends** AbstractSet<Map.Entry<K,V>> {
350. **public** Iterator<Map.Entry<K,V>> iterator() {
351. **return** getIterator(ENTRIES);
352. }
354. **public** **boolean** add(Map.Entry<K,V> o) {
355. **return** **super**.add(o);
356. }
358. // 查找EntrySet中是否包含Object(0)
359. // 首先，在table中找到o对应的Entry链表
360. // 然后，查找Entry链表中是否存在Object
361. **public** **boolean** contains(Object o) {
362. **if** (!(o **instanceof** Map.Entry))
363. **return** **false**;
364. Map.Entry entry = (Map.Entry)o;
365. Object key = entry.getKey();
366. Entry[] tab = table;
367. **int** hash = key.hashCode();
368. **int** index = (hash & 0x7FFFFFFF) % tab.length;
370. **for** (Entry e = tab[index]; e != **null**; e = e.next)
371. **if** (e.hash==hash && e.equals(entry))
372. **return** **true**;
373. **return** **false**;
374. }
376. // 删除元素Object(0)
377. // 首先，在table中找到o对应的Entry链表
378. // 然后，删除链表中的元素Object
379. **public** **boolean** remove(Object o) {
380. **if** (!(o **instanceof** Map.Entry))
381. **return** **false**;
382. Map.Entry<K,V> entry = (Map.Entry<K,V>) o;
383. K key = entry.getKey();
384. Entry[] tab = table;
385. **int** hash = key.hashCode();
386. **int** index = (hash & 0x7FFFFFFF) % tab.length;
388. **for** (Entry<K,V> e = tab[index], prev = **null**; e != **null**;
389. prev = e, e = e.next) {
390. **if** (e.hash==hash && e.equals(entry)) {
391. modCount++;
392. **if** (prev != **null**)
393. prev.next = e.next;
394. **else**
395. tab[index] = e.next;
397. count--;
398. e.value = **null**;
399. **return** **true**;
400. }
401. }
402. **return** **false**;
403. }
405. **public** **int** size() {
406. **return** count;
407. }
409. **public** **void** clear() {
410. Hashtable.**this**.clear();
411. }
412. }
414. // 返回一个被synchronizedCollection封装后的ValueCollection对象
415. // synchronizedCollection封装的目的是对ValueCollection的所有方法都添加synchronized，实现多线程同步
416. **public** Collection<V> values() {
417. **if** (values==**null**)
418. values = Collections.synchronizedCollection(**new** ValueCollection(),
419. **this**);
420. **return** values;
421. }
423. // Hashtable的value的Collection集合。
424. // ValueCollection继承于AbstractCollection，所以，ValueCollection中的元素可以重复的。
425. **private** **class** ValueCollection **extends** AbstractCollection<V> {
426. **public** Iterator<V> iterator() {
427. **return** getIterator(VALUES);
428. }
429. **public** **int** size() {
430. **return** count;
431. }
432. **public** **boolean** contains(Object o) {
433. **return** containsValue(o);
434. }
435. **public** **void** clear() {
436. Hashtable.**this**.clear();
437. }
438. }
440. // 重新equals()函数
441. // 若两个Hashtable的所有key-value键值对都相等，则判断它们两个相等
442. **public** **synchronized** **boolean** equals(Object o) {
443. **if** (o == **this**)
444. **return** **true**;
446. **if** (!(o **instanceof** Map))
447. **return** **false**;
448. Map<K,V> t = (Map<K,V>) o;
449. **if** (t.size() != size())
450. **return** **false**;
452. **try** {
453. // 通过迭代器依次取出当前Hashtable的key-value键值对
454. // 并判断该键值对，存在于Hashtable中。
455. // 若不存在，则立即返回false；否则，遍历完“当前Hashtable”并返回true。
456. Iterator<Map.Entry<K,V>> i = entrySet().iterator();
457. **while** (i.hasNext()) {
458. Map.Entry<K,V> e = i.next();
459. K key = e.getKey();
460. V value = e.getValue();
461. **if** (value == **null**) {
462. **if** (!(t.get(key)==**null** && t.containsKey(key)))
463. **return** **false**;
464. } **else** {
465. **if** (!value.equals(t.get(key)))
466. **return** **false**;
467. }
468. }
469. } **catch** (ClassCastException unused)   {
470. **return** **false**;
471. } **catch** (NullPointerException unused) {
472. **return** **false**;
473. }
475. **return** **true**;
476. }
478. // 计算Entry的hashCode
479. // 若 Hashtable的实际大小为0 或者 加载因子<0，则返回0。
480. // 否则，返回“Hashtable中的每个Entry的key和value的异或值 的总和”。
481. **public** **synchronized** **int** hashCode() {
482. **int** h = 0;
483. **if** (count == 0 || loadFactor < 0)
484. **return** h;  // Returns zero
486. loadFactor = -loadFactor;  // Mark hashCode computation in progress
487. Entry[] tab = table;
488. **for** (**int** i = 0; i < tab.length; i++)
489. **for** (Entry e = tab[i]; e != **null**; e = e.next)
490. h += e.key.hashCode() ^ e.value.hashCode();
491. loadFactor = -loadFactor;  // Mark hashCode computation complete
493. **return** h;
494. }
496. // java.io.Serializable的写入函数
497. // 将Hashtable的“总的容量，实际容量，所有的Entry”都写入到输出流中
498. **private** **synchronized** **void** writeObject(java.io.ObjectOutputStream s)
499. **throws** IOException
500. {
501. // Write out the length, threshold, loadfactor
502. s.defaultWriteObject();
504. // Write out length, count of elements and then the key/value objects
505. s.writeInt(table.length);
506. s.writeInt(count);
507. **for** (**int** index = table.length-1; index >= 0; index--) {
508. Entry entry = table[index];
510. **while** (entry != **null**) {
511. s.writeObject(entry.key);
512. s.writeObject(entry.value);
513. entry = entry.next;
514. }
515. }
516. }
518. // java.io.Serializable的读取函数：根据写入方式读出
519. // 将Hashtable的“总的容量，实际容量，所有的Entry”依次读出
520. **private** **void** readObject(java.io.ObjectInputStream s)
521. **throws** IOException, ClassNotFoundException
522. {
523. // Read in the length, threshold, and loadfactor
524. s.defaultReadObject();
526. // Read the original length of the array and number of elements
527. **int** origlength = s.readInt();
528. **int** elements = s.readInt();
530. // Compute new size with a bit of room 5% to grow but
531. // no larger than the original size.  Make the length
532. // odd if it's large enough, this helps distribute the entries.
533. // Guard against the length ending up zero, that's not valid.
534. **int** length = (**int**)(elements \* loadFactor) + (elements / 20) + 3;
535. **if** (length > elements && (length & 1) == 0)
536. length--;
537. **if** (origlength > 0 && length > origlength)
538. length = origlength;
540. Entry[] table = **new** Entry[length];
541. count = 0;
543. // Read the number of elements and then all the key/value objects
544. **for** (; elements > 0; elements--) {
545. K key = (K)s.readObject();
546. V value = (V)s.readObject();
547. // synch could be eliminated for performance
548. reconstitutionPut(table, key, value);
549. }
550. **this**.table = table;
551. }
553. **private** **void** reconstitutionPut(Entry[] tab, K key, V value)
554. **throws** StreamCorruptedException
555. {
556. **if** (value == **null**) {
557. **throw** **new** java.io.StreamCorruptedException();
558. }
559. // Makes sure the key is not already in the hashtable.
560. // This should not happen in deserialized version.
561. **int** hash = key.hashCode();
562. **int** index = (hash & 0x7FFFFFFF) % tab.length;
563. **for** (Entry<K,V> e = tab[index] ; e != **null** ; e = e.next) {
564. **if** ((e.hash == hash) && e.key.equals(key)) {
565. **throw** **new** java.io.StreamCorruptedException();
566. }
567. }
568. // Creates the new entry.
569. Entry<K,V> e = tab[index];
570. tab[index] = **new** Entry<K,V>(hash, key, value, e);
571. count++;
572. }
574. // Hashtable的Entry节点，它本质上是一个单向链表。
575. // 也因此，我们才能推断出Hashtable是由拉链法实现的散列表
576. **private** **static** **class** Entry<K,V> **implements** Map.Entry<K,V> {
577. // 哈希值
578. **int** hash;
579. K key;
580. V value;
581. // 指向的下一个Entry，即链表的下一个节点
582. Entry<K,V> next;
584. // 构造函数
585. **protected** Entry(**int** hash, K key, V value, Entry<K,V> next) {
586. **this**.hash = hash;
587. **this**.key = key;
588. **this**.value = value;
589. **this**.next = next;
590. }
592. **protected** Object clone() {
593. **return** **new** Entry<K,V>(hash, key, value,
594. (next==**null** ? **null** : (Entry<K,V>) next.clone()));
595. }
597. **public** K getKey() {
598. **return** key;
599. }
601. **public** V getValue() {
602. **return** value;
603. }
605. // 设置value。若value是null，则抛出异常。
606. **public** V setValue(V value) {
607. **if** (value == **null**)
608. **throw** **new** NullPointerException();
610. V oldValue = **this**.value;
611. **this**.value = value;
612. **return** oldValue;
613. }
615. // 覆盖equals()方法，判断两个Entry是否相等。
616. // 若两个Entry的key和value都相等，则认为它们相等。
617. **public** **boolean** equals(Object o) {
618. **if** (!(o **instanceof** Map.Entry))
619. **return** **false**;
620. Map.Entry e = (Map.Entry)o;
622. **return** (key==**null** ? e.getKey()==**null** : key.equals(e.getKey())) &&
623. (value==**null** ? e.getValue()==**null** : value.equals(e.getValue()));
624. }
626. **public** **int** hashCode() {
627. **return** hash ^ (value==**null** ? 0 : value.hashCode());
628. }
630. **public** String toString() {
631. **return** key.toString()+"="+value.toString();
632. }
633. }
635. **private** **static** **final** **int** KEYS = 0;
636. **private** **static** **final** **int** VALUES = 1;
637. **private** **static** **final** **int** ENTRIES = 2;
639. // Enumerator的作用是提供了“通过elements()遍历Hashtable的接口” 和 “通过entrySet()遍历Hashtable的接口”。
640. **private** **class** Enumerator<T> **implements** Enumeration<T>, Iterator<T> {
641. // 指向Hashtable的table
642. Entry[] table = Hashtable.**this**.table;
643. // Hashtable的总的大小
644. **int** index = table.length;
645. Entry<K,V> entry = **null**;
646. Entry<K,V> lastReturned = **null**;
647. **int** type;
649. // Enumerator是 “迭代器(Iterator)” 还是 “枚举类(Enumeration)”的标志
650. // iterator为true，表示它是迭代器；否则，是枚举类。
651. **boolean** iterator;
653. // 在将Enumerator当作迭代器使用时会用到，用来实现fail-fast机制。
654. **protected** **int** expectedModCount = modCount;
656. Enumerator(**int** type, **boolean** iterator) {
657. **this**.type = type;
658. **this**.iterator = iterator;
659. }
661. // 从遍历table的数组的末尾向前查找，直到找到不为null的Entry。
662. **public** **boolean** hasMoreElements() {
663. Entry<K,V> e = entry;
664. **int** i = index;
665. Entry[] t = table;
666. /\* Use locals for faster loop iteration \*/
667. **while** (e == **null** && i > 0) {
668. e = t[--i];
669. }
670. entry = e;
671. index = i;
672. **return** e != **null**;
673. }
675. // 获取下一个元素
676. // 注意：从hasMoreElements() 和nextElement() 可以看出“Hashtable的elements()遍历方式”
677. // 首先，从后向前的遍历table数组。table数组的每个节点都是一个单向链表(Entry)。
678. // 然后，依次向后遍历单向链表Entry。
679. **public** T nextElement() {
680. Entry<K,V> et = entry;
681. **int** i = index;
682. Entry[] t = table;
683. /\* Use locals for faster loop iteration \*/
684. **while** (et == **null** && i > 0) {
685. et = t[--i];
686. }
687. entry = et;
688. index = i;
689. **if** (et != **null**) {
690. Entry<K,V> e = lastReturned = entry;
691. entry = e.next;
692. **return** type == KEYS ? (T)e.key : (type == VALUES ? (T)e.value : (T)e);
693. }
694. **throw** **new** NoSuchElementException("Hashtable Enumerator");
695. }
697. // 迭代器Iterator的判断是否存在下一个元素
698. // 实际上，它是调用的hasMoreElements()
699. **public** **boolean** hasNext() {
700. **return** hasMoreElements();
701. }
703. // 迭代器获取下一个元素
704. // 实际上，它是调用的nextElement()
705. **public** T next() {
706. **if** (modCount != expectedModCount)
707. **throw** **new** ConcurrentModificationException();
708. **return** nextElement();
709. }
711. // 迭代器的remove()接口。
712. // 首先，它在table数组中找出要删除元素所在的Entry，
713. // 然后，删除单向链表Entry中的元素。
714. **public** **void** remove() {
715. **if** (!iterator)
716. **throw** **new** UnsupportedOperationException();
717. **if** (lastReturned == **null**)
718. **throw** **new** IllegalStateException("Hashtable Enumerator");
719. **if** (modCount != expectedModCount)
720. **throw** **new** ConcurrentModificationException();
722. **synchronized**(Hashtable.**this**) {
723. Entry[] tab = Hashtable.**this**.table;
724. **int** index = (lastReturned.hash & 0x7FFFFFFF) % tab.length;
726. **for** (Entry<K,V> e = tab[index], prev = **null**; e != **null**;
727. prev = e, e = e.next) {
728. **if** (e == lastReturned) {
729. modCount++;
730. expectedModCount++;
731. **if** (prev == **null**)
732. tab[index] = e.next;
733. **else**
734. prev.next = e.next;
735. count--;
736. lastReturned = **null**;
737. **return**;
738. }
739. }
740. **throw** **new** ConcurrentModificationException();
741. }
742. }
743. }

746. **private** **static** Enumeration emptyEnumerator = **new** EmptyEnumerator();
747. **private** **static** Iterator emptyIterator = **new** EmptyIterator();
749. // 空枚举类
750. // 当Hashtable的实际大小为0；此时，又要通过Enumeration遍历Hashtable时，返回的是“空枚举类”的对象。
751. **private** **static** **class** EmptyEnumerator **implements** Enumeration<Object> {
753. EmptyEnumerator() {
754. }
756. // 空枚举类的hasMoreElements() 始终返回false
757. **public** **boolean** hasMoreElements() {
758. **return** **false**;
759. }
761. // 空枚举类的nextElement() 抛出异常
762. **public** Object nextElement() {
763. **throw** **new** NoSuchElementException("Hashtable Enumerator");
764. }
765. }

768. // 空迭代器
769. // 当Hashtable的实际大小为0；此时，又要通过迭代器遍历Hashtable时，返回的是“空迭代器”的对象。
770. **private** **static** **class** EmptyIterator **implements** Iterator<Object> {
772. EmptyIterator() {
773. }
775. **public** **boolean** hasNext() {
776. **return** **false**;
777. }
779. **public** Object next() {
780. **throw** **new** NoSuchElementException("Hashtable Iterator");
781. }
783. **public** **void** remove() {
784. **throw** **new** IllegalStateException("Hashtable Iterator");
785. }
787. }
788. }

# 几点总结

    针对Hashtable，我们同样给出几点比较重要的总结，但要结合与HashMap的比较来总结。

    1、二者的存储结构和解决冲突的方法都是相同的。

    2、HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

    3、Hashtable中key和value都不允许为null，而HashMap中key和value都允许为null（key只能有一个为null，而value则可以有多个为null）。但是如果在Hashtable中有类似put(null,null)的操作，编译同样可以通过，因为key和value都是Object类型，但运行时会抛出NullPointerException异常，这是JDK的规范规定的。我们来看下ContainsKey方法和ContainsValue的源码：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36191279) [copy](http://blog.csdn.net/ns_code/article/details/36191279)

1. // 判断Hashtable是否包含“值(value)”
2. **public** **synchronized** **boolean** contains(Object value) {
3. //注意，Hashtable中的value不能是null，
4. // 若是null的话，抛出异常!
5. **if** (value == **null**) {
6. **throw** **new** NullPointerException();
7. }
9. // 从后向前遍历table数组中的元素(Entry)
10. // 对于每个Entry(单向链表)，逐个遍历，判断节点的值是否等于value
11. Entry tab[] = table;
12. **for** (**int** i = tab.length ; i-- > 0 ;) {
13. **for** (Entry<K,V> e = tab[i] ; e != **null** ; e = e.next) {
14. **if** (e.value.equals(value)) {
15. **return** **true**;
16. }
17. }
18. }
19. **return** **false**;
20. }
22. **public** **boolean** containsValue(Object value) {
23. **return** contains(value);
24. }
26. // 判断Hashtable是否包含key
27. **public** **synchronized** **boolean** containsKey(Object key) {
28. Entry tab[] = table;
29. /计算hash值，直接用key的hashCode代替
30. **int** hash = key.hashCode();
31. // 计算在数组中的索引值
32. **int** index = (hash & 0x7FFFFFFF) % tab.length;
33. // 找到“key对应的Entry(链表)”，然后在链表中找出“哈希值”和“键值”与key都相等的元素
34. **for** (Entry<K,V> e = tab[index] ; e != **null** ; e = e.next) {
35. **if** ((e.hash == hash) && e.key.equals(key)) {
36. **return** **true**;
37. }
38. }
39. **return** **false**;
40. }

    很明显，如果value为null，会直接抛出NullPointerException异常，但源码中并没有对key是否为null判断，有点小不解！不过NullPointerException属于RuntimeException异常，是可以由JVM自动抛出的，也许对key的值在JVM中有所限制吧。

    4、Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。  
    5、Hashtable计算hash值，直接用key的hashCode()，而HashMap重新计算了key的hash值，Hashtable在求hash值对应的位置索引时，用取模运算，而HashMap在求位置索引时，则用与运算，且这里一般先用hash&0x7FFFFFFF后，再对length取模，&0x7FFFFFFF的目的是为了将负的hash值转化为正值，因为hash值有可能为负数，而&0x7FFFFFFF后，只有符号外改变，而后面的位都不变。

# 红黑树简介

    TreeMap是基于红黑树实现的，这里只对红黑树做个简单的介绍，红黑树是一种特殊的二叉排序树，关于二叉排序树，参见：<http://blog.csdn.net/ns_code/article/details/19823463>，红黑树通过一些限制，使其不会出现二叉树排序树中极端的一边倒的情况，相对二叉排序树而言，这自然提高了查询的效率。

    二叉排序树的基本性质如下：

    1、每个节点都只能是红色或者黑色

    2、根节点是黑色

    3、每个叶节点（NIL节点，空节点）是黑色的。

    4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。

    5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

    正是这些性质的限制，使得红黑树中任一节点到其子孙叶子节点的最长路径不会长于最短路径的2倍，因此它是一种接近平衡的二叉树。

    说到红黑树，自然不免要和AVL树对比一番。相比较而言，AVL树是严格的平衡二叉树，而红黑树不算严格意义上的平衡二叉树，只是接近平衡，不会让树的高度如BST极端情况那样等于节点的个数。其实能用到红黑树的地方，也都可以用AVL树来实现，但红黑树的应用却非常广泛，而AVL树则很少被使用。在执行插入、删除操作时，AVL树需要调整的次数一般要比红黑树多（红黑树的旋转调整最多只需三次），效率相对较低，且红黑树的统计性能较AVL树要好，当然AVL树在查询效率上可能更胜一筹，但实际上也高不了多少。

红黑树的插入删除操作很简单，就是单纯的二叉排序树的插入删除操作。红黑树被认为比较变态的地方自然在于插入删除后对红黑树的调整操作（旋转和着色），主要是情况分的很多，限于篇幅及博主的熟悉程度优先，这里不打算详细介绍插入删除后调整红黑树的各种情况及其实现，我们有个宏观上的了解即可，如须详细了解，参见算法导论或一些相关的资料。

# TreeMap源码剖析

## 存储结构

    TreeMap的排序是基于对key的排序实现的，它的每一个Entry代表红黑树的一个节点，Entry的数据结构如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **static** **final** **class** Entry<K,V> **implements** Map.Entry<K,V> {
2. // 键
3. K key;
4. // 值
5. V value;
6. // 左孩子
7. Entry<K,V> left = **null**;
8. // 右孩子
9. Entry<K,V> right = **null**;
10. // 父节点
11. Entry<K,V> parent;
12. // 当前节点颜色
13. **boolean** color = BLACK;
15. // 构造函数
16. Entry(K key, V value, Entry<K,V> parent) {
17. **this**.key = key;
18. **this**.value = value;
19. **this**.parent = parent;
20. }
22. 。。。。。
23. }

## 构造方法

    先来看下TreeMap的构造方法。TreeMap一共有4个构造方法。

    1、无参构造方法

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** TreeMap() {
2. comparator = **null**;
3. }

    采用无参构造方法，不指定比较器，这时候，排序的实现要依赖key.compareTo()方法，因此key必须实现Comparable接口，并覆写其中的compareTo方法。

    2、带有比较器的构造方法

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** TreeMap(Comparator<? **super** K> comparator) {
2. **this**.comparator = comparator;
3. }

    采用带比较器的构造方法，这时候，排序依赖该比较器，key可以不用实现Comparable接口。

    3、带Map的构造方法

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** TreeMap(Map<? **extends** K, ? **extends** V> m) {
2. comparator = **null**;
3. putAll(m);
4. }

    该构造方法同样不指定比较器，调用putAll方法将Map中的所有元素加入到TreeMap中。putAll的源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. // 将map中的全部节点添加到TreeMap中
2. **public** **void** putAll(Map<? **extends** K, ? **extends** V> map) {
3. // 获取map的大小
4. **int** mapSize = map.size();
5. // 如果TreeMap的大小是0,且map的大小不是0,且map是已排序的“key-value对”
6. **if** (size==0 && mapSize!=0 && map **instanceof** SortedMap) {
7. Comparator c = ((SortedMap)map).comparator();
8. // 如果TreeMap和map的比较器相等；
9. // 则将map的元素全部拷贝到TreeMap中，然后返回！
10. **if** (c == comparator || (c != **null** && c.equals(comparator))) {
11. ++modCount;
12. **try** {
13. buildFromSorted(mapSize, map.entrySet().iterator(),
14. **null**, **null**);
15. } **catch** (java.io.IOException cannotHappen) {
16. } **catch** (ClassNotFoundException cannotHappen) {
17. }
18. **return**;
19. }
20. }
21. // 调用AbstractMap中的putAll();
22. // AbstractMap中的putAll()又会调用到TreeMap的put()
23. **super**.putAll(map);
24. }

    显然，如果Map里的元素是排好序的，就调用buildFromSorted方法来拷贝Map中的元素，这在下一个构造方法中会重点提及，而如果Map中的元素不是排好序的，就调用AbstractMap的putAll(map)方法，该方法源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** **void** putAll(Map<? **extends** K, ? **extends** V> m) {
2. **for** (Map.Entry<? **extends** K, ? **extends** V> e : m.entrySet())
3. put(e.getKey(), e.getValue());
4. }

    很明显它是将Map中的元素一个个put（插入）到TreeMap中的，主要因为Map中的元素是无序存放的，因此要一个个插入到红黑树中，使其有序存放，并满足红黑树的性质。

    4、带有SortedMap的构造方法

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** TreeMap(SortedMap<K, ? **extends** V> m) {
2. comparator = m.comparator();
3. **try** {
4. buildFromSorted(m.size(), m.entrySet().iterator(), **null**, **null**);
5. } **catch** (java.io.IOException cannotHappen) {
6. } **catch** (ClassNotFoundException cannotHappen) {
7. }
8. }

    首先将比较器指定为m的比较器，这取决于生成m时调用构造方法是否传入了指定的构造器，而后调用buildFromSorted方法，将SortedMap中的元素插入到TreeMap中，由于SortedMap中的元素师有序的，实际上它是根据SortedMap创建的TreeMap，将SortedMap中对应的元素添加到TreeMap中。

## 插入删除

    插入操作即对应TreeMap的put方法，put操作实际上只需按照二叉排序树的插入步骤来操作即可，插入到指定位置后，再做调整，使其保持红黑树的特性。put源码的实现：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** V put(K key, V value) {
2. Entry<K,V> t = root;
3. // 若红黑树为空，则插入根节点
4. **if** (t == **null**) {
5. // TBD:
6. // 5045147: (coll) Adding null to an empty TreeSet should
7. // throw NullPointerException
8. //
9. // compare(key, key); // type check
10. root = **new** Entry<K,V>(key, value, **null**);
11. size = 1;
12. modCount++;
13. **return** **null**;
14. }
15. **int** cmp;
16. Entry<K,V> parent;
17. // split comparator and comparable paths
18. Comparator<? **super** K> cpr = comparator;
19. // 找出(key, value)在二叉排序树中的插入位置。
20. // 红黑树是以key来进行排序的，所以这里以key来进行查找。
21. **if** (cpr != **null**) {
22. **do** {
23. parent = t;
24. cmp = cpr.compare(key, t.key);
25. **if** (cmp < 0)
26. t = t.left;
27. **else** **if** (cmp > 0)
28. t = t.right;
29. **else**
30. **return** t.setValue(value);
31. } **while** (t != **null**);
32. }
33. **else** {
34. **if** (key == **null**)
35. **throw** **new** NullPointerException();
36. Comparable<? **super** K> k = (Comparable<? **super** K>) key;
37. **do** {
38. parent = t;
39. cmp = k.compareTo(t.key);
40. **if** (cmp < 0)
41. t = t.left;
42. **else** **if** (cmp > 0)
43. t = t.right;
44. **else**
45. **return** t.setValue(value);
46. } **while** (t != **null**);
47. }
48. // 为（key-value）新建节点
49. Entry<K,V> e = **new** Entry<K,V>(key, value, parent);
50. **if** (cmp < 0)
51. parent.left = e;
52. **else**
53. parent.right = e;
54. // 插入新的节点后，调用fixAfterInsertion调整红黑树。
55. fixAfterInsertion(e);
56. size++;
57. modCount++;
58. **return** **null**;
59. }

    这里的fixAfterInsertion便是节点插入后对树进行调整的方法，这里不做介绍。  
    删除操作及对应TreeMap的deleteEntry方法，deleteEntry方法同样也只需按照二叉排序树的操作步骤实现即可，删除指定节点后，再对树进行调整即可。deleteEntry方法的实现源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. // 删除“红黑树的节点p”
2. **private** **void** deleteEntry(Entry<K,V> p) {
3. modCount++;
4. size--;
6. **if** (p.left != **null** && p.right != **null**) {
7. Entry<K,V> s = successor (p);
8. p.key = s.key;
9. p.value = s.value;
10. p = s;
11. }
13. Entry<K,V> replacement = (p.left != **null** ? p.left : p.right);
15. **if** (replacement != **null**) {
16. replacement.parent = p.parent;
17. **if** (p.parent == **null**)
18. root = replacement;
19. **else** **if** (p == p.parent.left)
20. p.parent.left  = replacement;
21. **else**
22. p.parent.right = replacement;
24. p.left = p.right = p.parent = **null**;
26. **if** (p.color == BLACK)
27. fixAfterDeletion(replacement);
28. } **else** **if** (p.parent == **null**) {
29. root = **null**;
30. } **else** {
31. **if** (p.color == BLACK)
32. fixAfterDeletion(p);
34. **if** (p.parent != **null**) {
35. **if** (p == p.parent.left)
36. p.parent.left = **null**;
37. **else** **if** (p == p.parent.right)
38. p.parent.right = **null**;
39. p.parent = **null**;
40. }
41. }
42. }

    后面的fixAfterDeletion方法便是节点删除后对树进行调整的方法，这里不做介绍。

    其他很多方法这里不再一一介绍。

# 几点总结

    本文对TreeMap的分析较前几篇文章有些浅尝辄止，TreeMap用的没有HashMap那么多，我们有个宏观上的把我和比较即可。

    1、TreeMap是根据key进行排序的，它的排序和定位需要依赖比较器或覆写Comparable接口，也因此不需要key覆写hashCode方法和equals方法，就可以排除掉重复的key，而HashMap的key则需要通过覆写hashCode方法和equals方法来确保没有重复的key。

    2、TreeMap的查询、插入、删除效率均没有HashMap高，一般只有要对key排序时才使用TreeMap。

3、TreeMap的key不能为null，而HashMap的key可以为null。

# 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没有丝线Serializable接口，因此它不支持序列化，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. // Vector的序列版本号
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. **public** **synchronized** **void** copyInto(Object[] anArray) {
55. System.arraycopy(elementData, 0, anArray, 0, elementCount);
56. }
58. // 将当前容量值设为 =实际元素个数
59. **public** **synchronized** **void** trimToSize() {
60. modCount++;
61. **int** oldCapacity = elementData.length;
62. **if** (elementCount < oldCapacity) {
63. elementData = Arrays.copyOf(elementData, elementCount);
64. }
65. }
67. // 确认“Vector容量”的帮助函数
68. **private** **void** ensureCapacityHelper(**int** minCapacity) {
69. **int** oldCapacity = elementData.length;
70. // 当Vector的容量不足以容纳当前的全部元素，增加容量大小。
71. // 若 容量增量系数>0(即capacityIncrement>0)，则将容量增大当capacityIncrement
72. // 否则，将容量增大一倍。
73. **if** (minCapacity > oldCapacity) {
74. Object[] oldData = elementData;
75. **int** newCapacity = (capacityIncrement > 0) ?
76. (oldCapacity + capacityIncrement) : (oldCapacity \* 2);
77. **if** (newCapacity < minCapacity) {
78. newCapacity = minCapacity;
79. }
80. elementData = Arrays.copyOf(elementData, newCapacity);
81. }
82. }
84. // 确定Vector的容量。
85. **public** **synchronized** **void** ensureCapacity(**int** minCapacity) {
86. // 将Vector的改变统计数+1
87. modCount++;
88. ensureCapacityHelper(minCapacity);
89. }
91. // 设置容量值为 newSize
92. **public** **synchronized** **void** setSize(**int** newSize) {
93. modCount++;
94. **if** (newSize > elementCount) {
95. // 若 "newSize 大于 Vector容量"，则调整Vector的大小。
96. ensureCapacityHelper(newSize);
97. } **else** {
98. // 若 "newSize 小于/等于 Vector容量"，则将newSize位置开始的元素都设置为null
99. **for** (**int** i = newSize ; i < elementCount ; i++) {
100. elementData[i] = **null**;
101. }
102. }
103. elementCount = newSize;
104. }
106. // 返回“Vector的总的容量”
107. **public** **synchronized** **int** capacity() {
108. **return** elementData.length;
109. }
111. // 返回“Vector的实际大小”，即Vector中元素个数
112. **public** **synchronized** **int** size() {
113. **return** elementCount;
114. }
116. // 判断Vector是否为空
117. **public** **synchronized** **boolean** isEmpty() {
118. **return** elementCount == 0;
119. }
121. // 返回“Vector中全部元素对应的Enumeration”
122. **public** Enumeration<E> elements() {
123. // 通过匿名类实现Enumeration
124. **return** **new** Enumeration<E>() {
125. **int** count = 0;
127. // 是否存在下一个元素
128. **public** **boolean** hasMoreElements() {
129. **return** count < elementCount;
130. }
132. // 获取下一个元素
133. **public** E nextElement() {
134. **synchronized** (Vector.**this**) {
135. **if** (count < elementCount) {
136. **return** (E)elementData[count++];
137. }
138. }
139. **throw** **new** NoSuchElementException("Vector Enumeration");
140. }
141. };
142. }
144. // 返回Vector中是否包含对象(o)
145. **public** **boolean** contains(Object o) {
146. **return** indexOf(o, 0) >= 0;
147. }

150. // 从index位置开始向后查找元素(o)。
151. // 若找到，则返回元素的索引值；否则，返回-1
152. **public** **synchronized** **int** indexOf(Object o, **int** index) {
153. **if** (o == **null**) {
154. // 若查找元素为null，则正向找出null元素，并返回它对应的序号
155. **for** (**int** i = index ; i < elementCount ; i++)
156. **if** (elementData[i]==**null**)
157. **return** i;
158. } **else** {
159. // 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
160. **for** (**int** i = index ; i < elementCount ; i++)
161. **if** (o.equals(elementData[i]))
162. **return** i;
163. }
164. **return** -1;
165. }
167. // 查找并返回元素(o)在Vector中的索引值
168. **public** **int** indexOf(Object o) {
169. **return** indexOf(o, 0);
170. }
172. // 从后向前查找元素(o)。并返回元素的索引
173. **public** **synchronized** **int** lastIndexOf(Object o) {
174. **return** lastIndexOf(o, elementCount-1);
175. }
177. // 从后向前查找元素(o)。开始位置是从前向后的第index个数；
178. // 若找到，则返回元素的“索引值”；否则，返回-1。
179. **public** **synchronized** **int** lastIndexOf(Object o, **int** index) {
180. **if** (index >= elementCount)
181. **throw** **new** IndexOutOfBoundsException(index + " >= "+ elementCount);
183. **if** (o == **null**) {
184. // 若查找元素为null，则反向找出null元素，并返回它对应的序号
185. **for** (**int** i = index; i >= 0; i--)
186. **if** (elementData[i]==**null**)
187. **return** i;
188. } **else** {
189. // 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
190. **for** (**int** i = index; i >= 0; i--)
191. **if** (o.equals(elementData[i]))
192. **return** i;
193. }
194. **return** -1;
195. }
197. // 返回Vector中index位置的元素。
198. // 若index月结，则抛出异常
199. **public** **synchronized** E elementAt(**int** index) {
200. **if** (index >= elementCount) {
201. **throw** **new** ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
202. }
204. **return** (E)elementData[index];
205. }
207. // 获取Vector中的第一个元素。
208. // 若失败，则抛出异常！
209. **public** **synchronized** E firstElement() {
210. **if** (elementCount == 0) {
211. **throw** **new** NoSuchElementException();
212. }
213. **return** (E)elementData[0];
214. }
216. // 获取Vector中的最后一个元素。
217. // 若失败，则抛出异常！
218. **public** **synchronized** E lastElement() {
219. **if** (elementCount == 0) {
220. **throw** **new** NoSuchElementException();
221. }
222. **return** (E)elementData[elementCount - 1];
223. }
225. // 设置index位置的元素值为obj
226. **public** **synchronized** **void** setElementAt(E obj, **int** index) {
227. **if** (index >= elementCount) {
228. **throw** **new** ArrayIndexOutOfBoundsException(index + " >= " +
229. elementCount);
230. }
231. elementData[index] = obj;
232. }
234. // 删除index位置的元素
235. **public** **synchronized** **void** removeElementAt(**int** index) {
236. modCount++;
237. **if** (index >= elementCount) {
238. **throw** **new** ArrayIndexOutOfBoundsException(index + " >= " +
239. elementCount);
240. } **else** **if** (index < 0) {
241. **throw** **new** ArrayIndexOutOfBoundsException(index);
242. }
244. **int** j = elementCount - index - 1;
245. **if** (j > 0) {
246. System.arraycopy(elementData, index + 1, elementData, index, j);
247. }
248. elementCount--;
249. elementData[elementCount] = **null**; /\* to let gc do its work \*/
250. }
252. // 在index位置处插入元素(obj)
253. **public** **synchronized** **void** insertElementAt(E obj, **int** index) {
254. modCount++;
255. **if** (index > elementCount) {
256. **throw** **new** ArrayIndexOutOfBoundsException(index
257. + " > " + elementCount);
258. }
259. ensureCapacityHelper(elementCount + 1);
260. System.arraycopy(elementData, index, elementData, index + 1, elementCount - index);
261. elementData[index] = obj;
262. elementCount++;
263. }
265. // 将“元素obj”添加到Vector末尾
266. **public** **synchronized** **void** addElement(E obj) {
267. modCount++;
268. ensureCapacityHelper(elementCount + 1);
269. elementData[elementCount++] = obj;
270. }
272. // 在Vector中查找并删除元素obj。
273. // 成功的话，返回true；否则，返回false。
274. **public** **synchronized** **boolean** removeElement(Object obj) {
275. modCount++;
276. **int** i = indexOf(obj);
277. **if** (i >= 0) {
278. removeElementAt(i);
279. **return** **true**;
280. }
281. **return** **false**;
282. }
284. // 删除Vector中的全部元素
285. **public** **synchronized** **void** removeAllElements() {
286. modCount++;
287. // 将Vector中的全部元素设为null
288. **for** (**int** i = 0; i < elementCount; i++)
289. elementData[i] = **null**;
291. elementCount = 0;
292. }
294. // 克隆函数
295. **public** **synchronized** Object clone() {
296. **try** {
297. Vector<E> v = (Vector<E>) **super**.clone();
298. // 将当前Vector的全部元素拷贝到v中
299. v.elementData = Arrays.copyOf(elementData, elementCount);
300. v.modCount = 0;
301. **return** v;
302. } **catch** (CloneNotSupportedException e) {
303. // this shouldn't happen, since we are Cloneable
304. **throw** **new** InternalError();
305. }
306. }
308. // 返回Object数组
309. **public** **synchronized** Object[] toArray() {
310. **return** Arrays.copyOf(elementData, elementCount);
311. }
313. // 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
314. **public** **synchronized** <T> T[] toArray(T[] a) {
315. // 若数组a的大小 < Vector的元素个数；
316. // 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
317. **if** (a.length < elementCount)
318. **return** (T[]) Arrays.copyOf(elementData, elementCount, a.getClass());
320. // 若数组a的大小 >= Vector的元素个数；
321. // 则将Vector的全部元素都拷贝到数组a中。
322. System.arraycopy(elementData, 0, a, 0, elementCount);
324. **if** (a.length > elementCount)
325. a[elementCount] = **null**;
327. **return** a;
328. }
330. // 获取index位置的元素
331. **public** **synchronized** E get(**int** index) {
332. **if** (index >= elementCount)
333. **throw** **new** ArrayIndexOutOfBoundsException(index);
335. **return** (E)elementData[index];
336. }
338. // 设置index位置的值为element。并返回index位置的原始值
339. **public** **synchronized** E set(**int** index, E element) {
340. **if** (index >= elementCount)
341. **throw** **new** ArrayIndexOutOfBoundsException(index);
343. Object oldValue = elementData[index];
344. elementData[index] = element;
345. **return** (E)oldValue;
346. }
348. // 将“元素e”添加到Vector最后。
349. **public** **synchronized** **boolean** add(E e) {
350. modCount++;
351. ensureCapacityHelper(elementCount + 1);
352. elementData[elementCount++] = e;
353. **return** **true**;
354. }
356. // 删除Vector中的元素o
357. **public** **boolean** remove(Object o) {
358. **return** removeElement(o);
359. }
361. // 在index位置添加元素element
362. **public** **void** add(**int** index, E element) {
363. insertElementAt(element, index);
364. }
366. // 删除index位置的元素，并返回index位置的原始值
367. **public** **synchronized** E remove(**int** index) {
368. modCount++;
369. **if** (index >= elementCount)
370. **throw** **new** ArrayIndexOutOfBoundsException(index);
371. Object oldValue = elementData[index];
373. **int** numMoved = elementCount - index - 1;
374. **if** (numMoved > 0)
375. System.arraycopy(elementData, index+1, elementData, index,
376. numMoved);
377. elementData[--elementCount] = **null**; // Let gc do its work
379. **return** (E)oldValue;
380. }
382. // 清空Vector
383. **public** **void** clear() {
384. removeAllElements();
385. }
387. // 返回Vector是否包含集合c
388. **public** **synchronized** **boolean** containsAll(Collection<?> c) {
389. **return** **super**.containsAll(c);
390. }
392. // 将集合c添加到Vector中
393. **public** **synchronized** **boolean** addAll(Collection<? **extends** E> c) {
394. modCount++;
395. Object[] a = c.toArray();
396. **int** numNew = a.length;
397. ensureCapacityHelper(elementCount + numNew);
398. // 将集合c的全部元素拷贝到数组elementData中
399. System.arraycopy(a, 0, elementData, elementCount, numNew);
400. elementCount += numNew;
401. **return** numNew != 0;
402. }
404. // 删除集合c的全部元素
405. **public** **synchronized** **boolean** removeAll(Collection<?> c) {
406. **return** **super**.removeAll(c);
407. }
409. // 删除“非集合c中的元素”
410. **public** **synchronized** **boolean** retainAll(Collection<?> c)  {
411. **return** **super**.retainAll(c);
412. }
414. // 从index位置开始，将集合c添加到Vector中
415. **public** **synchronized** **boolean** addAll(**int** index, Collection<? **extends** E> c) {
416. modCount++;
417. **if** (index < 0 || index > elementCount)
418. **throw** **new** ArrayIndexOutOfBoundsException(index);
420. Object[] a = c.toArray();
421. **int** numNew = a.length;
422. ensureCapacityHelper(elementCount + numNew);
424. **int** numMoved = elementCount - index;
425. **if** (numMoved > 0)
426. System.arraycopy(elementData, index, elementData, index + numNew, numMoved);
428. System.arraycopy(a, 0, elementData, index, numNew);
429. elementCount += numNew;
430. **return** numNew != 0;
431. }
433. // 返回两个对象是否相等
434. **public** **synchronized** **boolean** equals(Object o) {
435. **return** **super**.equals(o);
436. }
438. // 计算哈希值
439. **public** **synchronized** **int** hashCode() {
440. **return** **super**.hashCode();
441. }
443. // 调用父类的toString()
444. **public** **synchronized** String toString() {
445. **return** **super**.toString();
446. }
448. // 获取Vector中fromIndex(包括)到toIndex(不包括)的子集
449. **public** **synchronized** List<E> subList(**int** fromIndex, **int** toIndex) {
450. **return** Collections.synchronizedList(**super**.subList(fromIndex, toIndex), **this**);
451. }
453. // 删除Vector中fromIndex到toIndex的元素
454. **protected** **synchronized** **void** removeRange(**int** fromIndex, **int** toIndex) {
455. modCount++;
456. **int** numMoved = elementCount - toIndex;
457. System.arraycopy(elementData, toIndex, elementData, fromIndex,
458. numMoved);
460. // Let gc do its work
461. **int** newElementCount = elementCount - (toIndex-fromIndex);
462. **while** (elementCount != newElementCount)
463. elementData[--elementCount] = **null**;
464. }
466. // java.io.Serializable的写入函数
467. **private** **synchronized** **void** writeObject(java.io.ObjectOutputStream s)
468. **throws** java.io.IOException {
469. s.defaultWriteObject();
470. }
471. }

# 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，就设置新的容量为就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。

注：对TreeSet和HashSet的源码不再进行剖析，二者分别是基于TreeMap和HashMap实现的，只是对应的节点中只有key，而没有value，因此对TreeMap和HashMap比较了解的话，对TreeSet和HashSet的理解就会非常容易。

# LinkedList简介

    LinkedList是基于双向循环链表（从源码中可以很容易看出）实现的，除了可以当做链表来操作外，它还可以当做栈、队列和双端队列来使用。

    LinkedList同样是非线程安全的，只在单线程下适合使用。

    LinkedList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Cloneable接口，能被克隆。

# LinkedList源码剖析

    LinkedList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/35787253) [copy](http://blog.csdn.net/ns_code/article/details/35787253)

1. **package** java.util;
3. **public** **class** LinkedList<E>
4. **extends** AbstractSequentialList<E>
5. **implements** List<E>, Deque<E>, Cloneable, java.io.Serializable
6. {
7. // 链表的表头，表头不包含任何数据。Entry是个链表类数据结构。
8. **private** **transient** Entry<E> header = **new** Entry<E>(**null**, **null**, **null**);
10. // LinkedList中元素个数
11. **private** **transient** **int** size = 0;
13. // 默认构造函数：创建一个空的链表
14. **public** LinkedList() {
15. header.next = header.previous = header;
16. }
18. // 包含“集合”的构造函数:创建一个包含“集合”的LinkedList
19. **public** LinkedList(Collection<? **extends** E> c) {
20. **this**();
21. addAll(c);
22. }
24. // 获取LinkedList的第一个元素
25. **public** E getFirst() {
26. **if** (size==0)
27. **throw** **new** NoSuchElementException();
29. // 链表的表头header中不包含数据。
30. // 这里返回header所指下一个节点所包含的数据。
31. **return** header.next.element;
32. }
34. // 获取LinkedList的最后一个元素
35. **public** E getLast()  {
36. **if** (size==0)
37. **throw** **new** NoSuchElementException();
39. // 由于LinkedList是双向链表；而表头header不包含数据。
40. // 因而，这里返回表头header的前一个节点所包含的数据。
41. **return** header.previous.element;
42. }
44. // 删除LinkedList的第一个元素
45. **public** E removeFirst() {
46. **return** remove(header.next);
47. }
49. // 删除LinkedList的最后一个元素
50. **public** E removeLast() {
51. **return** remove(header.previous);
52. }
54. // 将元素添加到LinkedList的起始位置
55. **public** **void** addFirst(E e) {
56. addBefore(e, header.next);
57. }
59. // 将元素添加到LinkedList的结束位置
60. **public** **void** addLast(E e) {
61. addBefore(e, header);
62. }
64. // 判断LinkedList是否包含元素(o)
65. **public** **boolean** contains(Object o) {
66. **return** indexOf(o) != -1;
67. }
69. // 返回LinkedList的大小
70. **public** **int** size() {
71. **return** size;
72. }
74. // 将元素(E)添加到LinkedList中
75. **public** **boolean** add(E e) {
76. // 将节点(节点数据是e)添加到表头(header)之前。
77. // 即，将节点添加到双向链表的末端。
78. addBefore(e, header);
79. **return** **true**;
80. }
82. // 从LinkedList中删除元素(o)
83. // 从链表开始查找，如存在元素(o)则删除该元素并返回true；
84. // 否则，返回false。
85. **public** **boolean** remove(Object o) {
86. **if** (o==**null**) {
87. // 若o为null的删除情况
88. **for** (Entry<E> e = header.next; e != header; e = e.next) {
89. **if** (e.element==**null**) {
90. remove(e);
91. **return** **true**;
92. }
93. }
94. } **else** {
95. // 若o不为null的删除情况
96. **for** (Entry<E> e = header.next; e != header; e = e.next) {
97. **if** (o.equals(e.element)) {
98. remove(e);
99. **return** **true**;
100. }
101. }
102. }
103. **return** **false**;
104. }
106. // 将“集合(c)”添加到LinkedList中。
107. // 实际上，是从双向链表的末尾开始，将“集合(c)”添加到双向链表中。
108. **public** **boolean** addAll(Collection<? **extends** E> c) {
109. **return** addAll(size, c);
110. }
112. // 从双向链表的index开始，将“集合(c)”添加到双向链表中。
113. **public** **boolean** addAll(**int** index, Collection<? **extends** E> c) {
114. **if** (index < 0 || index > size)
115. **throw** **new** IndexOutOfBoundsException("Index: "+index+
116. ", Size: "+size);
117. Object[] a = c.toArray();
118. // 获取集合的长度
119. **int** numNew = a.length;
120. **if** (numNew==0)
121. **return** **false**;
122. modCount++;
124. // 设置“当前要插入节点的后一个节点”
125. Entry<E> successor = (index==size ? header : entry(index));
126. // 设置“当前要插入节点的前一个节点”
127. Entry<E> predecessor = successor.previous;
128. // 将集合(c)全部插入双向链表中
129. **for** (**int** i=0; i<numNew; i++) {
130. Entry<E> e = **new** Entry<E>((E)a[i], successor, predecessor);
131. predecessor.next = e;
132. predecessor = e;
133. }
134. successor.previous = predecessor;
136. // 调整LinkedList的实际大小
137. size += numNew;
138. **return** **true**;
139. }
141. // 清空双向链表
142. **public** **void** clear() {
143. Entry<E> e = header.next;
144. // 从表头开始，逐个向后遍历；对遍历到的节点执行一下操作：
145. // (01) 设置前一个节点为null
146. // (02) 设置当前节点的内容为null
147. // (03) 设置后一个节点为“新的当前节点”
148. **while** (e != header) {
149. Entry<E> next = e.next;
150. e.next = e.previous = **null**;
151. e.element = **null**;
152. e = next;
153. }
154. header.next = header.previous = header;
155. // 设置大小为0
156. size = 0;
157. modCount++;
158. }
160. // 返回LinkedList指定位置的元素
161. **public** E get(**int** index) {
162. **return** entry(index).element;
163. }
165. // 设置index位置对应的节点的值为element
166. **public** E set(**int** index, E element) {
167. Entry<E> e = entry(index);
168. E oldVal = e.element;
169. e.element = element;
170. **return** oldVal;
171. }
173. // 在index前添加节点，且节点的值为element
174. **public** **void** add(**int** index, E element) {
175. addBefore(element, (index==size ? header : entry(index)));
176. }
178. // 删除index位置的节点
179. **public** E remove(**int** index) {
180. **return** remove(entry(index));
181. }
183. // 获取双向链表中指定位置的节点
184. **private** Entry<E> entry(**int** index) {
185. **if** (index < 0 || index >= size)
186. **throw** **new** IndexOutOfBoundsException("Index: "+index+
187. ", Size: "+size);
188. Entry<E> e = header;
189. // 获取index处的节点。
190. // 若index < 双向链表长度的1/2,则从前先后查找;
191. // 否则，从后向前查找。
192. **if** (index < (size >> 1)) {
193. **for** (**int** i = 0; i <= index; i++)
194. e = e.next;
195. } **else** {
196. **for** (**int** i = size; i > index; i--)
197. e = e.previous;
198. }
199. **return** e;
200. }
202. // 从前向后查找，返回“值为对象(o)的节点对应的索引”
203. // 不存在就返回-1
204. **public** **int** indexOf(Object o) {
205. **int** index = 0;
206. **if** (o==**null**) {
207. **for** (Entry e = header.next; e != header; e = e.next) {
208. **if** (e.element==**null**)
209. **return** index;
210. index++;
211. }
212. } **else** {
213. **for** (Entry e = header.next; e != header; e = e.next) {
214. **if** (o.equals(e.element))
215. **return** index;
216. index++;
217. }
218. }
219. **return** -1;
220. }
222. // 从后向前查找，返回“值为对象(o)的节点对应的索引”
223. // 不存在就返回-1
224. **public** **int** lastIndexOf(Object o) {
225. **int** index = size;
226. **if** (o==**null**) {
227. **for** (Entry e = header.previous; e != header; e = e.previous) {
228. index--;
229. **if** (e.element==**null**)
230. **return** index;
231. }
232. } **else** {
233. **for** (Entry e = header.previous; e != header; e = e.previous) {
234. index--;
235. **if** (o.equals(e.element))
236. **return** index;
237. }
238. }
239. **return** -1;
240. }
242. // 返回第一个节点
243. // 若LinkedList的大小为0,则返回null
244. **public** E peek() {
245. **if** (size==0)
246. **return** **null**;
247. **return** getFirst();
248. }
250. // 返回第一个节点
251. // 若LinkedList的大小为0,则抛出异常
252. **public** E element() {
253. **return** getFirst();
254. }
256. // 删除并返回第一个节点
257. // 若LinkedList的大小为0,则返回null
258. **public** E poll() {
259. **if** (size==0)
260. **return** **null**;
261. **return** removeFirst();
262. }
264. // 将e添加双向链表末尾
265. **public** **boolean** offer(E e) {
266. **return** add(e);
267. }
269. // 将e添加双向链表开头
270. **public** **boolean** offerFirst(E e) {
271. addFirst(e);
272. **return** **true**;
273. }
275. // 将e添加双向链表末尾
276. **public** **boolean** offerLast(E e) {
277. addLast(e);
278. **return** **true**;
279. }
281. // 返回第一个节点
282. // 若LinkedList的大小为0,则返回null
283. **public** E peekFirst() {
284. **if** (size==0)
285. **return** **null**;
286. **return** getFirst();
287. }
289. // 返回最后一个节点
290. // 若LinkedList的大小为0,则返回null
291. **public** E peekLast() {
292. **if** (size==0)
293. **return** **null**;
294. **return** getLast();
295. }
297. // 删除并返回第一个节点
298. // 若LinkedList的大小为0,则返回null
299. **public** E pollFirst() {
300. **if** (size==0)
301. **return** **null**;
302. **return** removeFirst();
303. }
305. // 删除并返回最后一个节点
306. // 若LinkedList的大小为0,则返回null
307. **public** E pollLast() {
308. **if** (size==0)
309. **return** **null**;
310. **return** removeLast();
311. }
313. // 将e插入到双向链表开头
314. **public** **void** push(E e) {
315. addFirst(e);
316. }
318. // 删除并返回第一个节点
319. **public** E pop() {
320. **return** removeFirst();
321. }
323. // 从LinkedList开始向后查找，删除第一个值为元素(o)的节点
324. // 从链表开始查找，如存在节点的值为元素(o)的节点，则删除该节点
325. **public** **boolean** removeFirstOccurrence(Object o) {
326. **return** remove(o);
327. }
329. // 从LinkedList末尾向前查找，删除第一个值为元素(o)的节点
330. // 从链表开始查找，如存在节点的值为元素(o)的节点，则删除该节点
331. **public** **boolean** removeLastOccurrence(Object o) {
332. **if** (o==**null**) {
333. **for** (Entry<E> e = header.previous; e != header; e = e.previous) {
334. **if** (e.element==**null**) {
335. remove(e);
336. **return** **true**;
337. }
338. }
339. } **else** {
340. **for** (Entry<E> e = header.previous; e != header; e = e.previous) {
341. **if** (o.equals(e.element)) {
342. remove(e);
343. **return** **true**;
344. }
345. }
346. }
347. **return** **false**;
348. }
350. // 返回“index到末尾的全部节点”对应的ListIterator对象(List迭代器)
351. **public** ListIterator<E> listIterator(**int** index) {
352. **return** **new** ListItr(index);
353. }
355. // List迭代器
356. **private** **class** ListItr **implements** ListIterator<E> {
357. // 上一次返回的节点
358. **private** Entry<E> lastReturned = header;
359. // 下一个节点
360. **private** Entry<E> next;
361. // 下一个节点对应的索引值
362. **private** **int** nextIndex;
363. // 期望的改变计数。用来实现fail-fast机制。
364. **private** **int** expectedModCount = modCount;
366. // 构造函数。
367. // 从index位置开始进行迭代
368. ListItr(**int** index) {
369. // index的有效性处理
370. **if** (index < 0 || index > size)
371. **throw** **new** IndexOutOfBoundsException("Index: "+index+ ", Size: "+size);
372. // 若 “index 小于 ‘双向链表长度的一半’”，则从第一个元素开始往后查找；
373. // 否则，从最后一个元素往前查找。
374. **if** (index < (size >> 1)) {
375. next = header.next;
376. **for** (nextIndex=0; nextIndex<index; nextIndex++)
377. next = next.next;
378. } **else** {
379. next = header;
380. **for** (nextIndex=size; nextIndex>index; nextIndex--)
381. next = next.previous;
382. }
383. }
385. // 是否存在下一个元素
386. **public** **boolean** hasNext() {
387. // 通过元素索引是否等于“双向链表大小”来判断是否达到最后。
388. **return** nextIndex != size;
389. }
391. // 获取下一个元素
392. **public** E next() {
393. checkForComodification();
394. **if** (nextIndex == size)
395. **throw** **new** NoSuchElementException();
397. lastReturned = next;
398. // next指向链表的下一个元素
399. next = next.next;
400. nextIndex++;
401. **return** lastReturned.element;
402. }
404. // 是否存在上一个元素
405. **public** **boolean** hasPrevious() {
406. // 通过元素索引是否等于0，来判断是否达到开头。
407. **return** nextIndex != 0;
408. }
410. // 获取上一个元素
411. **public** E previous() {
412. **if** (nextIndex == 0)
413. **throw** **new** NoSuchElementException();
415. // next指向链表的上一个元素
416. lastReturned = next = next.previous;
417. nextIndex--;
418. checkForComodification();
419. **return** lastReturned.element;
420. }
422. // 获取下一个元素的索引
423. **public** **int** nextIndex() {
424. **return** nextIndex;
425. }
427. // 获取上一个元素的索引
428. **public** **int** previousIndex() {
429. **return** nextIndex-1;
430. }
432. // 删除当前元素。
433. // 删除双向链表中的当前节点
434. **public** **void** remove() {
435. checkForComodification();
436. Entry<E> lastNext = lastReturned.next;
437. **try** {
438. LinkedList.**this**.remove(lastReturned);
439. } **catch** (NoSuchElementException e) {
440. **throw** **new** IllegalStateException();
441. }
442. **if** (next==lastReturned)
443. next = lastNext;
444. **else**
445. nextIndex--;
446. lastReturned = header;
447. expectedModCount++;
448. }
450. // 设置当前节点为e
451. **public** **void** set(E e) {
452. **if** (lastReturned == header)
453. **throw** **new** IllegalStateException();
454. checkForComodification();
455. lastReturned.element = e;
456. }
458. // 将e添加到当前节点的前面
459. **public** **void** add(E e) {
460. checkForComodification();
461. lastReturned = header;
462. addBefore(e, next);
463. nextIndex++;
464. expectedModCount++;
465. }
467. // 判断 “modCount和expectedModCount是否相等”，依次来实现fail-fast机制。
468. **final** **void** checkForComodification() {
469. **if** (modCount != expectedModCount)
470. **throw** **new** ConcurrentModificationException();
471. }
472. }
474. // 双向链表的节点所对应的数据结构。
475. // 包含3部分：上一节点，下一节点，当前节点值。
476. **private** **static** **class** Entry<E> {
477. // 当前节点所包含的值
478. E element;
479. // 下一个节点
480. Entry<E> next;
481. // 上一个节点
482. Entry<E> previous;
484. /\*\*
485. \* 链表节点的构造函数。
486. \* 参数说明：
487. \*   element  —— 节点所包含的数据
488. \*   next      —— 下一个节点
489. \*   previous —— 上一个节点
490. \*/
491. Entry(E element, Entry<E> next, Entry<E> previous) {
492. **this**.element = element;
493. **this**.next = next;
494. **this**.previous = previous;
495. }
496. }
498. // 将节点(节点数据是e)添加到entry节点之前。
499. **private** Entry<E> addBefore(E e, Entry<E> entry) {
500. // 新建节点newEntry，将newEntry插入到节点e之前；并且设置newEntry的数据是e
501. Entry<E> newEntry = **new** Entry<E>(e, entry, entry.previous);
502. newEntry.previous.next = newEntry;
503. newEntry.next.previous = newEntry;
504. // 修改LinkedList大小
505. size++;
506. // 修改LinkedList的修改统计数：用来实现fail-fast机制。
507. modCount++;
508. **return** newEntry;
509. }
511. // 将节点从链表中删除
512. **private** E remove(Entry<E> e) {
513. **if** (e == header)
514. **throw** **new** NoSuchElementException();
516. E result = e.element;
517. e.previous.next = e.next;
518. e.next.previous = e.previous;
519. e.next = e.previous = **null**;
520. e.element = **null**;
521. size--;
522. modCount++;
523. **return** result;
524. }
526. // 反向迭代器
527. **public** Iterator<E> descendingIterator() {
528. **return** **new** DescendingIterator();
529. }
531. // 反向迭代器实现类。
532. **private** **class** DescendingIterator **implements** Iterator {
533. **final** ListItr itr = **new** ListItr(size());
534. // 反向迭代器是否下一个元素。
535. // 实际上是判断双向链表的当前节点是否达到开头
536. **public** **boolean** hasNext() {
537. **return** itr.hasPrevious();
538. }
539. // 反向迭代器获取下一个元素。
540. // 实际上是获取双向链表的前一个节点
541. **public** E next() {
542. **return** itr.previous();
543. }
544. // 删除当前节点
545. **public** **void** remove() {
546. itr.remove();
547. }
548. }

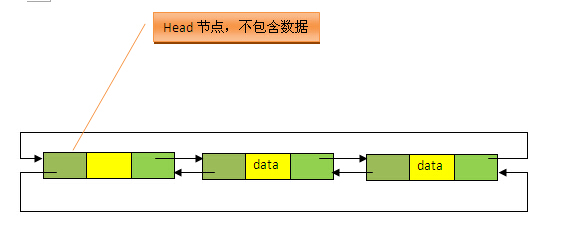
551. // 返回LinkedList的Object[]数组
552. **public** Object[] toArray() {
553. // 新建Object[]数组
554. Object[] result = **new** Object[size];
555. **int** i = 0;
556. // 将链表中所有节点的数据都添加到Object[]数组中
557. **for** (Entry<E> e = header.next; e != header; e = e.next)
558. result[i++] = e.element;
559. **return** result;
560. }
562. // 返回LinkedList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
563. **public** <T> T[] toArray(T[] a) {
564. // 若数组a的大小 < LinkedList的元素个数(意味着数组a不能容纳LinkedList中全部元素)
565. // 则新建一个T[]数组，T[]的大小为LinkedList大小，并将该T[]赋值给a。
566. **if** (a.length < size)
567. a = (T[])java.lang.reflect.Array.newInstance(
568. a.getClass().getComponentType(), size);
569. // 将链表中所有节点的数据都添加到数组a中
570. **int** i = 0;
571. Object[] result = a;
572. **for** (Entry<E> e = header.next; e != header; e = e.next)
573. result[i++] = e.element;
575. **if** (a.length > size)
576. a[size] = **null**;
578. **return** a;
579. }

582. // 克隆函数。返回LinkedList的克隆对象。
583. **public** Object clone() {
584. LinkedList<E> clone = **null**;
585. // 克隆一个LinkedList克隆对象
586. **try** {
587. clone = (LinkedList<E>) **super**.clone();
588. } **catch** (CloneNotSupportedException e) {
589. **throw** **new** InternalError();
590. }
592. // 新建LinkedList表头节点
593. clone.header = **new** Entry<E>(**null**, **null**, **null**);
594. clone.header.next = clone.header.previous = clone.header;
595. clone.size = 0;
596. clone.modCount = 0;
598. // 将链表中所有节点的数据都添加到克隆对象中
599. **for** (Entry<E> e = header.next; e != header; e = e.next)
600. clone.add(e.element);
602. **return** clone;
603. }
605. // java.io.Serializable的写入函数
606. // 将LinkedList的“容量，所有的元素值”都写入到输出流中
607. **private** **void** writeObject(java.io.ObjectOutputStream s)
608. **throws** java.io.IOException {
609. // Write out any hidden serialization magic
610. s.defaultWriteObject();
612. // 写入“容量”
613. s.writeInt(size);
615. // 将链表中所有节点的数据都写入到输出流中
616. **for** (Entry e = header.next; e != header; e = e.next)
617. s.writeObject(e.element);
618. }
620. // java.io.Serializable的读取函数：根据写入方式反向读出
621. // 先将LinkedList的“容量”读出，然后将“所有的元素值”读出
622. **private** **void** readObject(java.io.ObjectInputStream s)
623. **throws** java.io.IOException, ClassNotFoundException {
624. // Read in any hidden serialization magic
625. s.defaultReadObject();
627. // 从输入流中读取“容量”
628. **int** size = s.readInt();
630. // 新建链表表头节点
631. header = **new** Entry<E>(**null**, **null**, **null**);
632. header.next = header.previous = header;
634. // 从输入流中将“所有的元素值”并逐个添加到链表中
635. **for** (**int** i=0; i<size; i++)
636. addBefore((E)s.readObject(), header);
637. }
639. }

# 几点总结

    关于LinkedList的源码，给出几点比较重要的总结：

    1、从源码中很明显可以看出，LinkedList的实现是基于双向循环链表的，且头结点中不存放数据，如下图;



    2、注意两个不同的构造方法。无参构造方法直接建立一个仅包含head节点的空链表，包含Collection的构造方法，先调用无参构造方法建立一个空链表，而后将Collection中的数据加入到链表的尾部后面。

    3、在查找和删除某元素时，源码中都划分为该元素为null和不为null两种情况来处理，LinkedList中允许元素为null。

    4、LinkedList是基于链表实现的，因此不存在容量不足的问题，所以这里没有扩容的方法。

    5、注意源码中的Entry<E> entry(**int** index)方法。该方法返回双向链表中指定位置处的节点，而链表中是没有下标索引的，要指定位置出的元素，就要遍历该链表，从源码的实现中，我们看到这里有一个加速动作。源码中先将index与长度size的一半比较，如果index<size/2，就只从位置0往后遍历到位置index处，而如果index>size/2，就只从位置size往前遍历到位置index处。这样可以减少一部分不必要的遍历，从而提高一定的效率（实际上效率还是很低）。

    6、注意链表类对应的数据结构Entry。如下;

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/35787253) [copy](http://blog.csdn.net/ns_code/article/details/35787253)

1. // 双向链表的节点所对应的数据结构。
2. // 包含3部分：上一节点，下一节点，当前节点值。
3. **private** **static** **class** Entry<E> {
4. // 当前节点所包含的值
5. E element;
6. // 下一个节点
7. Entry<E> next;
8. // 上一个节点
9. Entry<E> previous;
11. /\*\*
12. \* 链表节点的构造函数。
13. \* 参数说明：
14. \*   element  —— 节点所包含的数据
15. \*   next      —— 下一个节点
16. \*   previous —— 上一个节点
17. \*/
18. Entry(E element, Entry<E> next, Entry<E> previous) {
19. **this**.element = element;
20. **this**.next = next;
21. **this**.previous = previous;
22. }
23. }

7、LinkedList是基于链表实现的，因此插入删除效率高，查找效率低（虽然有一个加速动作）。  
    8、要注意源码中还实现了栈和队列的操作方法，因此也可以作为栈、队列和双端队列来使用。

# 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. **public** ArrayList(Collection<? **extends** E> c) {
32. elementData = c.toArray();
33. size = elementData.length;
34. **if** (elementData.getClass() != Object[].**class**)
35. elementData = Arrays.copyOf(elementData, size, Object[].**class**);
36. }

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

49. // 确定ArrarList的容量。
50. // 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”
51. **public** **void** ensureCapacity(**int** minCapacity) {
52. // 将“修改统计数”+1，该变量主要是用来实现fail-fast机制的
53. modCount++;
54. **int** oldCapacity = elementData.length;
55. // 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”
56. **if** (minCapacity > oldCapacity) {
57. Object oldData[] = elementData;
58. **int** newCapacity = (oldCapacity \* 3)/2 + 1;
59. //如果还不够，则直接将minCapacity设置为当前容量
60. **if** (newCapacity < minCapacity)
61. newCapacity = minCapacity;
62. elementData = Arrays.copyOf(elementData, newCapacity);
63. }
64. }
66. // 添加元素e
67. **public** **boolean** add(E e) {
68. // 确定ArrayList的容量大小
69. ensureCapacity(size + 1);  // Increments modCount!!
70. // 添加e到ArrayList中
71. elementData[size++] = e;
72. **return** **true**;
73. }
75. // 返回ArrayList的实际大小
76. **public** **int** size() {
77. **return** size;
78. }
80. // ArrayList是否包含Object(o)
81. **public** **boolean** contains(Object o) {
82. **return** indexOf(o) >= 0;
83. }
85. //返回ArrayList是否为空
86. **public** **boolean** isEmpty() {
87. **return** size == 0;
88. }
90. // 正向查找，返回元素的索引值
91. **public** **int** indexOf(Object o) {
92. **if** (o == **null**) {
93. **for** (**int** i = 0; i < size; i++)
94. **if** (elementData[i]==**null**)
95. **return** i;
96. } **else** {
97. **for** (**int** i = 0; i < size; i++)
98. **if** (o.equals(elementData[i]))
99. **return** i;
100. }
101. **return** -1;
102. }
104. // 反向查找，返回元素的索引值
105. **public** **int** lastIndexOf(Object o) {
106. **if** (o == **null**) {
107. **for** (**int** i = size-1; i >= 0; i--)
108. **if** (elementData[i]==**null**)
109. **return** i;
110. } **else** {
111. **for** (**int** i = size-1; i >= 0; i--)
112. **if** (o.equals(elementData[i]))
113. **return** i;
114. }
115. **return** -1;
116. }
118. // 反向查找(从数组末尾向开始查找)，返回元素(o)的索引值
119. **public** **int** lastIndexOf(Object o) {
120. **if** (o == **null**) {
121. **for** (**int** i = size-1; i >= 0; i--)
122. **if** (elementData[i]==**null**)
123. **return** i;
124. } **else** {
125. **for** (**int** i = size-1; i >= 0; i--)
126. **if** (o.equals(elementData[i]))
127. **return** i;
128. }
129. **return** -1;
130. }

133. // 返回ArrayList的Object数组
134. **public** Object[] toArray() {
135. **return** Arrays.copyOf(elementData, size);
136. }
138. // 返回ArrayList元素组成的数组
139. **public** <T> T[] toArray(T[] a) {
140. // 若数组a的大小 < ArrayList的元素个数；
141. // 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
142. **if** (a.length < size)
143. **return** (T[]) Arrays.copyOf(elementData, size, a.getClass());
145. // 若数组a的大小 >= ArrayList的元素个数；
146. // 则将ArrayList的全部元素都拷贝到数组a中。
147. System.arraycopy(elementData, 0, a, 0, size);
148. **if** (a.length > size)
149. a[size] = **null**;
150. **return** a;
151. }
153. // 获取index位置的元素值
154. **public** E get(**int** index) {
155. RangeCheck(index);
157. **return** (E) elementData[index];
158. }
160. // 设置index位置的值为element
161. **public** E set(**int** index, E element) {
162. RangeCheck(index);
164. E oldValue = (E) elementData[index];
165. elementData[index] = element;
166. **return** oldValue;
167. }
169. // 将e添加到ArrayList中
170. **public** **boolean** add(E e) {
171. ensureCapacity(size + 1);  // Increments modCount!!
172. elementData[size++] = e;
173. **return** **true**;
174. }
176. // 将e添加到ArrayList的指定位置
177. **public** **void** add(**int** index, E element) {
178. **if** (index > size || index < 0)
179. **throw** **new** IndexOutOfBoundsException(
180. "Index: "+index+", Size: "+size);
182. ensureCapacity(size+1);  // Increments modCount!!
183. System.arraycopy(elementData, index, elementData, index + 1,
184. size - index);
185. elementData[index] = element;
186. size++;
187. }
189. // 删除ArrayList指定位置的元素
190. **public** E remove(**int** index) {
191. RangeCheck(index);
193. modCount++;
194. E oldValue = (E) elementData[index];
196. **int** numMoved = size - index - 1;
197. **if** (numMoved > 0)
198. System.arraycopy(elementData, index+1, elementData, index,
199. numMoved);
200. elementData[--size] = **null**; // Let gc do its work
202. **return** oldValue;
203. }
205. // 删除ArrayList的指定元素
206. **public** **boolean** remove(Object o) {
207. **if** (o == **null**) {
208. **for** (**int** index = 0; index < size; index++)
209. **if** (elementData[index] == **null**) {
210. fastRemove(index);
211. **return** **true**;
212. }
213. } **else** {
214. **for** (**int** index = 0; index < size; index++)
215. **if** (o.equals(elementData[index])) {
216. fastRemove(index);
217. **return** **true**;
218. }
219. }
220. **return** **false**;
221. }

224. // 快速删除第index个元素
225. **private** **void** fastRemove(**int** index) {
226. modCount++;
227. **int** numMoved = size - index - 1;
228. // 从"index+1"开始，用后面的元素替换前面的元素。
229. **if** (numMoved > 0)
230. System.arraycopy(elementData, index+1, elementData, index,
231. numMoved);
232. // 将最后一个元素设为null
233. elementData[--size] = **null**; // Let gc do its work
234. }
236. // 删除元素
237. **public** **boolean** remove(Object o) {
238. **if** (o == **null**) {
239. **for** (**int** index = 0; index < size; index++)
240. **if** (elementData[index] == **null**) {
241. fastRemove(index);
242. **return** **true**;
243. }
244. } **else** {
245. // 便利ArrayList，找到“元素o”，则删除，并返回true。
246. **for** (**int** index = 0; index < size; index++)
247. **if** (o.equals(elementData[index])) {
248. fastRemove(index);
249. **return** **true**;
250. }
251. }
252. **return** **false**;
253. }
255. // 清空ArrayList，将全部的元素设为null
256. **public** **void** clear() {
257. modCount++;
259. **for** (**int** i = 0; i < size; i++)
260. elementData[i] = **null**;
262. size = 0;
263. }
265. // 将集合c追加到ArrayList中
266. **public** **boolean** addAll(Collection<? **extends** E> c) {
267. Object[] a = c.toArray();
268. **int** numNew = a.length;
269. ensureCapacity(size + numNew);  // Increments modCount
270. System.arraycopy(a, 0, elementData, size, numNew);
271. size += numNew;
272. **return** numNew != 0;
273. }
275. // 从index位置开始，将集合c添加到ArrayList
276. **public** **boolean** addAll(**int** index, Collection<? **extends** E> c) {
277. **if** (index > size || index < 0)
278. **throw** **new** IndexOutOfBoundsException(
279. "Index: " + index + ", Size: " + size);
281. Object[] a = c.toArray();
282. **int** numNew = a.length;
283. ensureCapacity(size + numNew);  // Increments modCount
285. **int** numMoved = size - index;
286. **if** (numMoved > 0)
287. System.arraycopy(elementData, index, elementData, index + numNew,
288. numMoved);
290. System.arraycopy(a, 0, elementData, index, numNew);
291. size += numNew;
292. **return** numNew != 0;
293. }
295. // 删除fromIndex到toIndex之间的全部元素。
296. **protected** **void** removeRange(**int** fromIndex, **int** toIndex) {
297. modCount++;
298. **int** numMoved = size - toIndex;
299. System.arraycopy(elementData, toIndex, elementData, fromIndex,
300. numMoved);
302. // Let gc do its work
303. **int** newSize = size - (toIndex-fromIndex);
304. **while** (size != newSize)
305. elementData[--size] = **null**;
306. }
308. **private** **void** RangeCheck(**int** index) {
309. **if** (index >= size)
310. **throw** **new** IndexOutOfBoundsException(
311. "Index: "+index+", Size: "+size);
312. }

315. // 克隆函数
316. **public** Object clone() {
317. **try** {
318. ArrayList<E> v = (ArrayList<E>) **super**.clone();
319. // 将当前ArrayList的全部元素拷贝到v中
320. v.elementData = Arrays.copyOf(elementData, size);
321. v.modCount = 0;
322. **return** v;
323. } **catch** (CloneNotSupportedException e) {
324. // this shouldn't happen, since we are Cloneable
325. **throw** **new** InternalError();
326. }
327. }

330. // java.io.Serializable的写入函数
331. // 将ArrayList的“容量，所有的元素值”都写入到输出流中
332. **private** **void** writeObject(java.io.ObjectOutputStream s)
333. **throws** java.io.IOException{
334. // Write out element count, and any hidden stuff
335. **int** expectedModCount = modCount;
336. s.defaultWriteObject();
338. // 写入“数组的容量”
339. s.writeInt(elementData.length);
341. // 写入“数组的每一个元素”
342. **for** (**int** i=0; i<size; i++)
343. s.writeObject(elementData[i]);
345. **if** (modCount != expectedModCount) {
346. **throw** **new** ConcurrentModificationException();
347. }
349. }

352. // java.io.Serializable的读取函数：根据写入方式读出
353. // 先将ArrayList的“容量”读出，然后将“所有的元素值”读出
354. **private** **void** readObject(java.io.ObjectInputStream s)
355. **throws** java.io.IOException, ClassNotFoundException {
356. // Read in size, and any hidden stuff
357. s.defaultReadObject();
359. // 从输入流中读取ArrayList的“容量”
360. **int** arrayLength = s.readInt();
361. Object[] a = elementData = **new** Object[arrayLength];
363. // 从输入流中将“所有的元素值”读出
364. **for** (**int** i=0; i<size; i++)
365. a[i] = s.readObject();
366. }
367. }

# 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. }

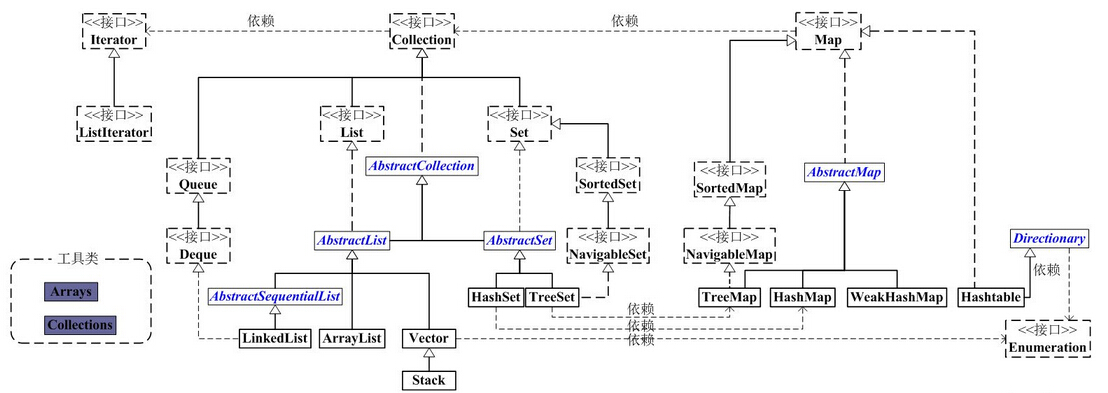
     5、ArrayList基于数组实现，可以通过下标索引直接查找到指定位置的元素，因此查找效率高，但每次插入或删除元素，就要大量地移动元素，插入删除元素的效率低。

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。

[Java集合框架](http://blog.csdn.net/ns_code/article/details/35564663)

  Java集合工具包位于Java.util包下，包含了很多常用的数据结构，如数组、链表、栈、队列、集合、哈希表等。学习Java集合框架下大致可以分为如下五个部分：List列表、Set集合、Map映射、迭代器（Iterator、Enumeration）、工具类（Arrays、Collections）。

    Java集合类的整体框架如下：



    从上图中可以看出，集合类主要分为两大类：Collection和Map。

    Collection是List、Set等集合高度抽象出来的接口，它包含了这些集合的基本操作，它主要又分为两大部分：List和Set。

    List接口通常表示一个列表（数组、队列、链表、栈等），其中的元素可以重复，常用实现类为ArrayList和LinkedList，另外还有不常用的Vector。另外，LinkedList还是实现了Queue接口，因此也可以作为队列使用。

    Set接口通常表示一个集合，其中的元素不允许重复（通过hashcode和equals函数保证），常用实现类有HashSet和TreeSet，HashSet是通过Map中的HashMap实现的，而TreeSet是通过Map中的TreeMap实现的。另外，TreeSet还实现了SortedSet接口，因此是有序的集合（集合中的元素要实现Comparable接口，并覆写Compartor函数才行）。

    我们看到，抽象类AbstractCollection、AbstractList和AbstractSet分别实现了Collection、List和Set接口，这就是在Java集合框架中用的很多的适配器设计模式，用这些抽象类去实现接口，在抽象类中实现接口中的若干或全部方法，这样下面的一些类只需直接继承该抽象类，并实现自己需要的方法即可，而不用实现接口中的全部抽象方法。

    Map是一个映射接口，其中的每个元素都是一个key-value键值对，同样抽象类AbstractMap通过适配器模式实现了Map接口中的大部分函数，TreeMap、HashMap、WeakHashMap等实现类都通过继承AbstractMap来实现，另外，不常用的HashTable直接实现了Map接口，它和Vector都是JDK1.0就引入的集合类。

    Iterator是遍历集合的迭代器（不能遍历Map，只用来遍历Collection），Collection的实现类都实现了iterator()函数，它返回一个Iterator对象，用来遍历集合，ListIterator则专门用来遍历List。而Enumeration则是JDK1.0时引入的，作用与Iterator相同，但它的功能比Iterator要少，它只能再Hashtable、Vector和Stack中使用。

    Arrays和Collections是用来操作数组、集合的两个工具类，例如在ArrayList和Vector中大量调用了Arrays.Copyof()方法，而Collections中有很多静态方法可以返回各集合类的synchronized版本，即线程安全的版本，当然了，如果要用线程安全的结合类，首选Concurrent并发包下的对应的集合类。