参考：<http://blog.csdn.net/ns_code/article/details/35787253>

[**【Java集合源码剖析】LinkedList源码剖析**](http://blog.csdn.net/ns_code/article/details/35787253)

**转载请注明出处：**[**http://blog.csdn.net/ns\_code/article/details/35787253**](http://blog.csdn.net/ns_code/article/details/35787253)

**您好，我正在参加CSDN博文大赛，如果您喜欢我的文章，希望您能帮我投一票，谢谢！**

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

**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. }

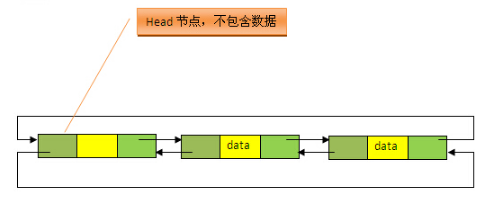
551. // 返回LinkedList的Object[]数组
552. **public** Object[] toArray() {
553. // 新建Object[]数组
554. Object[] result = **new** Object[size];
555. **int** i = 0;
556. // 将链表中所有节点的数据都添加到Object[]数组中
557. **for** (Entry<E> e = header.next; e != header; e = e.next)
558. result[i++] = e.element;
559. **return** 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。

**您好，我正在参加CSDN博文大赛，如果您喜欢我的文章，希望您能帮我投一票，谢谢！**

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