参考：<http://blog.csdn.net/ns_code/article/details/37867985>

[**【Java集合源码剖析】LinkedHashmap源码剖析**](http://blog.csdn.net/ns_code/article/details/37867985)

目录[(?)[+]](http://blog.csdn.net/ns_code/article/details/37867985)

**转载请注明出处：**[**http://blog.csdn.net/ns\_code/article/details/37867985**](http://blog.csdn.net/ns_code/article/details/37867985)

    前言：有网友建议分析下LinkedHashMap的源码，于是花了一晚上时间研究了下，分享出此文（这个系列的最后一篇博文了），希望大家相互学习。LinkedHashMap的源码理解起来也不难（当然，要建立在对HashMap源码有较好理解的基础上）。

**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LinkedHashMap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。