参考：<http://blog.csdn.net/ns_code/article/details/36191279>

[**【Java集合源码剖析】Hashtable源码剖析**](http://blog.csdn.net/ns_code/article/details/36191279)

、**转载请注明出处：**[**http://blog.csdn.net/ns\_code/article/details/36191279**](http://blog.csdn.net/ns_code/article/details/36191279)

**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（me：那是因为两者计算索引的算法不同）。

    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。