本系列教程参考的是jdk 1.7 ，jdk 1.8的实现与1.7的有些许区别

参考：<http://blog.csdn.net/ns_code/article/details/36034955>

[**【Java集合源码剖析】HashMap源码剖析**](http://blog.csdn.net/ns_code/article/details/36034955)

[**http://blog.csdn.net/ns\_code/article/details/36034955**](http://blog.csdn.net/ns_code/article/details/36034955)

**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)

**HashMap简介**

    HashMap是基于哈希表（me：hash表，就是散列表）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。（me：hashmap 是由数组和单链表实现）

    HashMap是非线程安全的（me：在put时可能导致线程非安全），只适用于单线程环境下，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（me：某些字段使用了transit关键字修饰从而在串行化是忽略相关的字段），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)

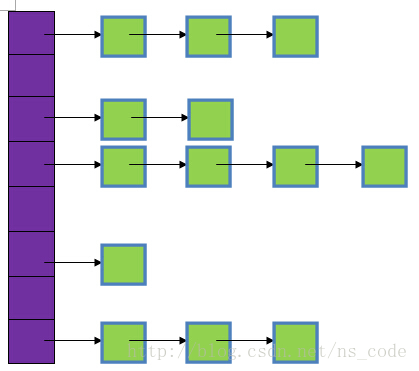
1. **package** java.util;
2. **import** java.io.\*;
4. **public** **class** HashMap<K,V>
5. **extends** AbstractMap<K,V>
6. **implements** Map<K,V>, Cloneable, Serializable
7. {
9. // 默认的初始容量（容量为HashMap中槽的数目）是16，且实际容量必须是2的整数次幂。
10. **static** **final** **int** DEFAULT\_INITIAL\_CAPACITY = 16;
12. // 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. //当发现冲突即碰撞的时候会使用equals来比较
21. **transient** Entry[] table;
23. // HashMap中添加的元素的数量，也可以说是已经用的槽的数量，如果没有冲突的话，
24. // 如果出现冲突且所有的槽都满的话那么槽的数量和数据的数量是不同的
25. //但是size真实返回值是槽的数量，按照常理应该返回集合中元素的数量
26. **transient** **int** size;
28. // 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）
29. **int** threshold;
31. // 加载因子实际大小，如果没有指定的话就使用默认的加载因子0.75，final变量
32. //会提供初始话构造方法
33. **final** **float** loadFactor;
35. // HashMap被改变的次数
36. **transient** **volatile** **int** modCount;
38. // 指定“容量大小”和“加载因子”的构造函数
39. **public** HashMap(**int** initialCapacity, **float** loadFactor) {
40. **if** (initialCapacity < 0)
41. **throw** **new** IllegalArgumentException("Illegal initial capacity: " +
42. initialCapacity);
43. // HashMap的最大容量只能是MAXIMUM\_CAPACITY
44. **if** (initialCapacity > MAXIMUM\_CAPACITY)
45. initialCapacity = MAXIMUM\_CAPACITY;
46. //加载因此不能小于0
47. **if** (loadFactor <= 0 || Float.isNaN(loadFactor))
48. **throw** **new** IllegalArgumentException("Illegal load factor: " +
49. loadFactor);
51. // 找出“大于initialCapacity”的最小的2的幂
52. **int** capacity = 1;
53. **while** (capacity < initialCapacity)
54. capacity <<= 1;
56. // 设置“加载因子”
57. **this**.loadFactor = loadFactor;
58. // 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。
59. threshold = (**int**)(capacity \* loadFactor);
60. // 创建Entry数组，用来保存数据
61. table = **new** Entry[capacity];
62. init();
63. }

66. // 指定“容量大小”的构造函数
67. **public** HashMap(**int** initialCapacity) {
68. **this**(initialCapacity, DEFAULT\_LOAD\_FACTOR);
69. }
71. // 默认构造函数。
72. **public** HashMap() {
73. // 设置“加载因子”为默认加载因子0.75
74. **this**.loadFactor = DEFAULT\_LOAD\_FACTOR;
75. // 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。
76. //  阈值就是槽的容量\*装在因子；当达到一定值的时候就需要扩容，以免真需要的时候出现问题
77. threshold = (**int**)(DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR);
78. // 创建Entry数组，用来保存数据
79. table = **new** Entry[DEFAULT\_INITIAL\_CAPACITY];
80. init();
81. }
83. // 包含“子Map”的构造函数
84. **public** HashMap(Map<? **extends** K, ? **extends** V> m) {
85. **this**(Math.max((**int**) (m.size() / DEFAULT\_LOAD\_FACTOR) + 1,
86. DEFAULT\_INITIAL\_CAPACITY), DEFAULT\_LOAD\_FACTOR);
87. // 将m中的全部元素逐个添加到HashMap中
88. putAllForCreate(m);
89. }
91. //求hash值的方法，重新计算hash值
92. **static** **int** hash(**int** h) {
93. h ^= (h >>> 20) ^ (h >>> 12);
94. **return** h ^ (h >>> 7) ^ (h >>> 4);
95. }
97. // 返回h在数组中的索引值，这里用&代替取模，旨在提升效率
98. // h & (length-1)保证返回值的小于length
99. **static** **int** indexFor(**int** h, **int** length) {
100. **return** h & (length-1);
101. }
103. **public** **int** size() {//返回的实际存储的数据的个数，也可以说是有效的槽的个数
104. //当不存在冲突的时候，如果出现冲突且所有的槽都满的话槽的个数小于实际存储的数据的格式；源码实现返回的是有效的的数量，但是按照常理应该返回集合中元素的数量
105. **return** size;
106. }
108. **public** **boolean** isEmpty() {
109. **return** size == 0;
110. }
112. // 获取key对应的value
113. **public** V get(Object key) {
114. **if** (key == **null**)
115. **return** getForNullKey();
116. // 获取key的hash值
117. **int** hash = hash(key.hashCode());
118. // 在“该hash值对应的链表”上查找“键值等于key”的元素
119. **for** (Entry<K,V> e = table[indexFor(hash, table.length)];
120. e != **null**;
121. e = e.next) {
122. Object k;
123. //判断key是否相同，Entry存储了key的hash,key和value以及next
124. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k)))
125. **return** e.value;
126. }
127. //没找到则返回null
128. **return** **null**;
129. }
131. // 获取“key为null”的元素的值
132. // 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！
133. **private** V getForNullKey() {
134. **for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {
135. **if** (e.key == **null**)
136. **return** e.value;
137. }
138. **return** **null**;
139. }
141. // HashMap是否包含key
142. **public** **boolean** containsKey(Object key) {
143. **return** getEntry(key) != **null**;
144. }
146. // 返回“键为key”的键值对
147. **final** Entry<K,V> getEntry(Object key) {
148. // 获取哈希值
149. // 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
150. **int** hash = (key == **null**) ? 0 : hash(key.hashCode());
151. // 在“该hash值对应的链表”上查找“键值等于key”的元素
152. **for** (Entry<K,V> e = table[indexFor(hash, table.length)];
153. e != **null**;
154. e = e.next) {
155. Object k;
156. //判断元素，即判断Entry是否真实相同
157. **if** (e.hash == hash &&
158. ((k = e.key) == key || (key != **null** && key.equals(k))))
159. **return** e;
160. }
161. **return** **null**;
162. }
164. // 将“key-value”添加到HashMap中
165. //该方法会返回旧值，如果键冲突的话
166. **public** V put(K key, V value) {
167. // 若“key为null”，则将该键值对添加到table[0]中。
168. **if** (key == **null**)
169. **return** putForNullKey(value);
170. // 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。
171. **int** hash = hash(key.hashCode());
172. **int** i = indexFor(hash, table.length);
173. **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {
174. Object k;
175. // 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！
176. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
177. V oldValue = e.value;
178. e.value = value;
179. e.recordAccess(**this**);
180. **return** oldValue;
181. }
182. }
184. // 若“该key”对应的键值对不存在，则将“key-value”添加到table中
185. modCount++;
186. //将key-value添加到table[i]处
187. addEntry(hash, key, value, i);
188. **return** **null**;
189. }
191. // putForNullKey()的作用是将“key为null”键值对添加到table[0]位置
192. **private** V putForNullKey(V value) {
193. **for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {
194. **if** (e.key == **null**) {
195. V oldValue = e.value;
196. e.value = value;
197. e.recordAccess(**this**);
198. **return** oldValue;
199. }
200. }
201. // 如果没有存在key为null的键值对，则直接添加到table[0]处!
202. modCount++;
203. addEntry(0, **null**, value, 0);
204. **return** **null**;
205. }
207. // 创建HashMap对应的“添加方法”，
208. // 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
209. // 而put()是对外提供的往HashMap中添加元素的方法。
210. **private** **void** putForCreate(K key, V value) {
211. **int** hash = (key == **null**) ? 0 : hash(key.hashCode());
212. **int** i = indexFor(hash, table.length);
214. // 若该HashMap表中存在“键值等于key”的元素，则替换该元素的value值
215. **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {
216. Object k;
217. **if** (e.hash == hash &&
218. ((k = e.key) == key || (key != **null** && key.equals(k)))) {
219. e.value = value;
220. **return**;
221. }
222. }
224. // 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
225. createEntry(hash, key, value, i);
226. }
228. // 将“m”中的全部元素都添加到HashMap中。
229. // 该方法被内部的构造HashMap的方法所调用。
230. **private** **void** putAllForCreate(Map<? **extends** K, ? **extends** V> m) {
231. // 利用迭代器将元素逐个添加到HashMap中
232. **for** (Iterator<? **extends** Map.Entry<? **extends** K, ? **extends** V>> i = m.entrySet().iterator(); i.hasNext(); ) {
233. Map.Entry<? **extends** K, ? **extends** V> e = i.next();
234. putForCreate(e.getKey(), e.getValue());
235. }
236. }
238. // 重新调整HashMap的大小，newCapacity是调整后的容量
239. **void** resize(**int** newCapacity) {
240. Entry[] oldTable = table;
241. **int** oldCapacity = oldTable.length;
242. //如果就容量已经达到了最大值，则不能再扩容，直接返回
243. **if** (oldCapacity == MAXIMUM\_CAPACITY) {
244. threshold = Integer.MAX\_VALUE;
245. **return**;
246. }
248. // 新建一个HashMap，将“旧HashMap”的全部元素添加到“新HashMap”中，
249. // 然后，将“新HashMap”赋值给“旧HashMap”。
250. Entry[] newTable = **new** Entry[newCapacity];
251. transfer(newTable);
252. table = newTable;
253. threshold = (**int**)(newCapacity \* loadFactor);
254. }
256. // 将HashMap中的全部元素都添加到newTable中
257. **void** transfer(Entry[] newTable) {
258. Entry[] src = table;
259. **int** newCapacity = newTable.length;
260. **for** (**int** j = 0; j < src.length; j++) {
261. Entry<K,V> e = src[j];
262. **if** (e != **null**) {
263. src[j] = **null**;
264. **do** {
265. Entry<K,V> next = e.next;
266. **int** i = indexFor(e.hash, newCapacity);
267. e.next = newTable[i];
268. newTable[i] = e;
269. e = next;
270. } **while** (e != **null**);
271. }
272. }
273. }
275. // 将"m"的全部元素都添加到HashMap中
276. **public** **void** putAll(Map<? **extends** K, ? **extends** V> m) {
277. // 有效性判断
278. **int** numKeysToBeAdded = m.size();
279. **if** (numKeysToBeAdded == 0)
280. **return**;
282. // 计算容量是否足够，
283. // 若“当前阀值容量 < 需要的容量”，则将容量x2。
284. **if** (numKeysToBeAdded > threshold) {
285. **int** targetCapacity = (**int**)(numKeysToBeAdded / loadFactor + 1);
286. **if** (targetCapacity > MAXIMUM\_CAPACITY)
287. targetCapacity = MAXIMUM\_CAPACITY;
288. **int** newCapacity = table.length;
289. **while** (newCapacity < targetCapacity)
290. newCapacity <<= 1;    //相当于\*2，但是使用位运算更加的快速
291. **if** (newCapacity > table.length)
292. resize(newCapacity);    //将table的容量重新设置，并将原table
293. //的元素拷贝到新的table
294. }
296. // 通过迭代器，将“m”中的元素逐个添加到HashMap中。
297. **for** (Iterator<? **extends** Map.Entry<? **extends** K, ? **extends** V>> i = m.entrySet().iterator(); i.hasNext(); ) {
298. Map.Entry<? **extends** K, ? **extends** V> e = i.next();
299. put(e.getKey(), e.getValue());
300. }
301. }
303. // 删除“键为key”元素
304. **public** V remove(Object key) {
305. Entry<K,V> e = removeEntryForKey(key);
306. **return** (e == **null** ? **null** : e.value);
307. }
309. // 删除“键为key”的元素
310. **final** Entry<K,V> removeEntryForKey(Object key) {
311. // 获取哈希值。若key为null，则哈希值为0；否则调用hash()进行计算
312. **int** hash = (key == **null**) ? 0 : hash(key.hashCode());
313. **int** i = indexFor(hash, table.length);
314. Entry<K,V> prev = table[i];
315. Entry<K,V> e = prev;
317. // 删除链表中“键为key”的元素
318. // 本质是“删除单向链表中的节点”
319. **while** (e != **null**) {
320. Entry<K,V> next = e.next;
321. Object k;
322. **if** (e.hash == hash &&
323. ((k = e.key) == key || (key != **null** && key.equals(k)))) {
324. modCount++;
325. size--;
326. **if** (prev == e)
327. table[i] = next;
328. **else**
329. prev.next = next;
330. e.recordRemoval(**this**);
331. **return** e;
332. }
333. prev = e;
334. e = next;
335. }
337. **return** e;
338. }
340. // 删除“键值对”
341. **final** Entry<K,V> removeMapping(Object o) {
342. **if** (!(o **instanceof** Map.Entry))
343. **return** **null**;
345. Map.Entry<K,V> entry = (Map.Entry<K,V>) o;
346. Object key = entry.getKey();
347. **int** hash = (key == **null**) ? 0 : hash(key.hashCode());
348. **int** i = indexFor(hash, table.length);
349. Entry<K,V> prev = table[i];
350. Entry<K,V> e = prev;
352. // 删除链表中的“键值对e”
353. // 本质是“删除单向链表中的节点”
354. **while** (e != **null**) {
355. Entry<K,V> next = e.next;
356. **if** (e.hash == hash && e.equals(entry)) {
357. modCount++;
358. size--;
359. **if** (prev == e)
360. table[i] = next;
361. **else**
362. prev.next = next;
363. e.recordRemoval(**this**);
364. **return** e;
365. }
366. prev = e;
367. e = next;
368. }
370. **return** e;
371. }
373. // 清空HashMap，将所有的元素设为null(me：而非将某些数据做个删除的标记，而是
374. 会真实的遍历)
375. **public** **void** clear() {
376. modCount++;
377. Entry[] tab = table;
378. **for** (**int** i = 0; i < tab.length; i++)
379. tab[i] = **null**;
380. size = 0;
381. }
383. // 是否包含“值为value”的元素（me：这个查找value的过程可能会使用到
384. 链表的查询，比较key是否equals）
385. //遍历槽然后遍历链表，双层遍历
386. **public** **boolean** containsValue(Object value) {
387. // 若“value为null”，则调用containsNullValue()查找
388. **if** (value == **null**)
389. **return** containsNullValue();
391. // 若“value不为null”，则查找HashMap中是否有值为value的节点。
392. Entry[] tab = table;
393. **for** (**int** i = 0; i < tab.length ; i++)
394. **for** (Entry e = tab[i] ; e != **null** ; e = e.next)
395. **if** (value.equals(e.value))
396. **return** **true**;
397. **return** **false**;
398. }
400. // 是否包含null值
401. **private** **boolean** containsNullValue() {
402. Entry[] tab = table;
403. **for** (**int** i = 0; i < tab.length ; i++)
404. **for** (Entry e = tab[i] ; e != **null** ; e = e.next)
405. **if** (e.value == **null**)
406. **return** **true**;
407. **return** **false**;
408. }
410. // 克隆一个HashMap，并返回Object对象，深复制
411. **public** Object clone() {
412. HashMap<K,V> result = **null**;
413. **try** {
414. result = (HashMap<K,V>)**super**.clone();
415. } **catch** (CloneNotSupportedException e) {
416. // assert false;
417. }
418. result.table = **new** Entry[table.length];
419. result.entrySet = **null**;
420. result.modCount = 0;
421. result.size = 0;
422. result.init();
423. // 调用putAllForCreate()将全部元素添加到HashMap中
424. result.putAllForCreate(**this**);
426. **return** result;
427. }
429. // Entry是单向链表。
430. // 它是 “HashMap链式存储法”对应的链表。
431. // 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Object o), hashCode()这些函数
432. **static** **class** Entry<K,V> **implements** Map.Entry<K,V> {
433. **final** K key;
434. V value;
435. // 指向下一个节点
436. Entry<K,V> next;
437. **final** **int** hash;
439. // 构造函数。
440. // 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"
441. Entry(**int** h, K k, V v, Entry<K,V> n) {
442. value = v;
443. next = n;
444. key = k;
445. hash = h;
446. }
448. **public** **final** K getKey() {
449. **return** key;
450. }
452. **public** **final** V getValue() {
453. **return** value;
454. }
456. **public** **final** V setValue(V newValue) {
457. V oldValue = value;
458. value = newValue;
459. **return** oldValue;
460. }
462. // 判断两个Entry是否相等
463. // 若两个Entry的“key”和“value”都相等，则返回true。
464. // 否则，返回false
465. **public** **final** **boolean** equals(Object o) {
466. **if** (!(o **instanceof** Map.Entry))
467. **return** **false**;
468. Map.Entry e = (Map.Entry)o;
469. Object k1 = getKey();
470. Object k2 = e.getKey();
471. **if** (k1 == k2 || (k1 != **null** && k1.equals(k2))) {
472. Object v1 = getValue();
473. Object v2 = e.getValue();
474. **if** (v1 == v2 || (v1 != **null** && v1.equals(v2)))
475. **return** **true**;
476. }
477. **return** **false**;
478. }
480. // 实现hashCode()
481. **public** **final** **int** hashCode() {
482. **return** (key==**null**   ? 0 : key.hashCode()) ^
483. (value==**null** ? 0 : value.hashCode());
484. }
486. **public** **final** String toString() {
487. **return** getKey() + "=" + getValue();
488. }
490. // 当向HashMap中添加元素时，绘调用recordAccess()。
491. // 这里不做任何处理
492. **void** recordAccess(HashMap<K,V> m) {
493. }
495. // 当从HashMap中删除元素时，绘调用recordRemoval()。
496. // 这里不做任何处理
497. **void** recordRemoval(HashMap<K,V> m) {
498. }
499. }
501. // 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。
502. **void** addEntry(**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. // 若HashMap的实际大小 不小于 “阈值”，则调整HashMap的大小
509. **if** (size++ >= threshold)
510. resize(2 \* table.length);
511. }
513. // 创建Entry。将“key-value”插入指定位置。
514. **void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {
515. // 保存“bucketIndex”位置的值到“e”中
516. Entry<K,V> e = table[bucketIndex];
517. // 设置“bucketIndex”位置的元素为“新Entry”，
518. // 设置“e”为“新Entry的下一个节点”
519. table[bucketIndex] = **new** Entry<K,V>(hash, key, value, e);
520. size++;
521. }
523. // HashIterator是HashMap迭代器的抽象出来的父类，实现了公共了函数。
524. // 它包含“key迭代器(KeyIterator)”、“Value迭代器(ValueIterator)”和“Entry迭代器(EntryIterator)”3个子类。
525. **private** **abstract** **class** HashIterator<E> **implements** Iterator<E> {
526. // 下一个元素
527. Entry<K,V> next;
528. // expectedModCount用于实现fast-fail机制。
529. **int** expectedModCount;
530. // 当前索引
531. **int** index;
532. // 当前元素
533. Entry<K,V> current;
535. HashIterator() {
536. expectedModCount = modCount;
537. **if** (size > 0) { // advance to first entry
538. Entry[] t = table;
539. // 将next指向table中第一个不为null的元素。
540. // 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。
541. **while** (index < t.length && (next = t[index++]) == **null**)
542. ;
543. }
544. }
546. **public** **final** **boolean** hasNext() {
547. **return** next != **null**;
548. }
550. // 获取下一个元素
551. **final** Entry<K,V> nextEntry() {
552. **if** (modCount != expectedModCount)
553. **throw** **new** ConcurrentModificationException();
554. Entry<K,V> e = next;
555. **if** (e == **null**)
556. **throw** **new** NoSuchElementException();
558. // 注意！！！
559. // 一个Entry就是一个单向链表
560. // 若该Entry的下一个节点不为空，就将next指向下一个节点;
561. // 否则，将next指向下一个链表(也是下一个Entry)的不为null的节点。
562. **if** ((next = e.next) == **null**) {
563. Entry[] t = table;
564. **while** (index < t.length && (next = t[index++]) == **null**)
565. ;
566. }
567. current = e;
568. **return** e;
569. }
571. // 删除当前元素
572. **public** **void** remove() {
573. **if** (current == **null**)
574. **throw** **new** IllegalStateException();
575. **if** (modCount != expectedModCount)
576. **throw** **new** ConcurrentModificationException();
577. Object k = current.key;
578. current = **null**;
579. HashMap.**this**.removeEntryForKey(k);
580. expectedModCount = modCount;
581. }
583. }
585. // value的迭代器
586. **private** **final** **class** ValueIterator **extends** HashIterator<V> {
587. **public** V next() {
588. **return** nextEntry().value;
589. }
590. }
592. // key的迭代器
593. **private** **final** **class** KeyIterator **extends** HashIterator<K> {
594. **public** K next() {
595. **return** nextEntry().getKey();
596. }
597. }
599. // Entry的迭代器
600. **private** **final** **class** EntryIterator **extends** HashIterator<Map.Entry<K,V>> {
601. **public** Map.Entry<K,V> next() {
602. **return** nextEntry();
603. }
604. }
606. // 返回一个“key迭代器”
607. Iterator<K> newKeyIterator()   {
608. **return** **new** KeyIterator();
609. }
610. // 返回一个“value迭代器”
611. Iterator<V> newValueIterator()   {
612. **return** **new** ValueIterator();
613. }
614. // 返回一个“entry迭代器”
615. Iterator<Map.Entry<K,V>> newEntryIterator()   {
616. **return** **new** EntryIterator();
617. }
619. // HashMap的Entry对应的集合
620. **private** **transient** Set<Map.Entry<K,V>> entrySet = **null**;
622. // 返回“key的集合”，实际上返回一个“KeySet对象”
623. **public** Set<K> keySet() {
624. Set<K> ks = keySet;
625. **return** (ks != **null** ? ks : (keySet = **new** KeySet()));
626. }
628. // Key对应的集合
629. // KeySet继承于AbstractSet，说明该集合中没有重复的Key。
630. **private** **final** **class** KeySet **extends** AbstractSet<K> {
631. **public** Iterator<K> iterator() {
632. **return** newKeyIterator();
633. }
634. **public** **int** size() {
635. **return** size;
636. }
637. **public** **boolean** contains(Object o) {
638. **return** containsKey(o);
639. }
640. **public** **boolean** remove(Object o) {
641. **return** HashMap.**this**.removeEntryForKey(o) != **null**;
642. }
643. **public** **void** clear() {
644. HashMap.**this**.clear();
645. }
646. }
648. // 返回“value集合”，实际上返回的是一个Values对象
649. **public** Collection<V> values() {
650. Collection<V> vs = values;
651. **return** (vs != **null** ? vs : (values = **new** Values()));
652. }
654. // “value集合”
655. // Values继承于AbstractCollection，不同于“KeySet继承于AbstractSet”，
656. // Values中的元素能够重复。因为不同的key可以指向相同的value。
657. **private** **final** **class** Values **extends** AbstractCollection<V> {
658. **public** Iterator<V> iterator() {
659. **return** newValueIterator();
660. }
661. **public** **int** size() {
662. **return** size;
663. }
664. **public** **boolean** contains(Object o) {
665. **return** containsValue(o);
666. }
667. **public** **void** clear() {
668. HashMap.**this**.clear();
669. }
670. }
672. // 返回“HashMap的Entry集合”
673. **public** Set<Map.Entry<K,V>> entrySet() {
674. **return** entrySet0();
675. }
677. // 返回“HashMap的Entry集合”，它实际是返回一个EntrySet对象
678. **private** Set<Map.Entry<K,V>> entrySet0() {
679. Set<Map.Entry<K,V>> es = entrySet;
680. **return** es != **null** ? es : (entrySet = **new** EntrySet());
681. }
683. // EntrySet对应的集合
684. // EntrySet继承于AbstractSet，说明该集合中没有重复的EntrySet。
685. **private** **final** **class** EntrySet **extends** AbstractSet<Map.Entry<K,V>> {
686. **public** Iterator<Map.Entry<K,V>> iterator() {
687. **return** newEntryIterator();
688. }
689. **public** **boolean** contains(Object o) {
690. **if** (!(o **instanceof** Map.Entry))
691. **return** **false**;
692. Map.Entry<K,V> e = (Map.Entry<K,V>) o;
693. Entry<K,V> candidate = getEntry(e.getKey());
694. **return** candidate != **null** && candidate.equals(e);
695. }
696. **public** **boolean** remove(Object o) {
697. **return** removeMapping(o) != **null**;
698. }
699. **public** **int** size() {
700. **return** size;
701. }
702. **public** **void** clear() {
703. HashMap.**this**.clear();
704. }
705. }
707. // java.io.Serializable的写入函数
708. // 将HashMap的“总的容量，实际容量，所有的Entry”都写入到输出流中
709. **private** **void** writeObject(java.io.ObjectOutputStream s)
710. **throws** IOException
711. {
712. Iterator<Map.Entry<K,V>> i =
713. (size > 0) ? entrySet0().iterator() : **null**;
715. // Write out the threshold, loadfactor, and any hidden stuff
716. s.defaultWriteObject();
718. // Write out number of buckets
719. s.writeInt(table.length);
721. // Write out size (number of Mappings)
722. s.writeInt(size);
724. // Write out keys and values (alternating)
725. **if** (i != **null**) {
726. **while** (i.hasNext()) {
727. Map.Entry<K,V> e = i.next();
728. s.writeObject(e.getKey());
729. s.writeObject(e.getValue());
730. }
731. }
732. }

735. **private** **static** **final** **long** serialVersionUID = 362498820763181265L;
737. // java.io.Serializable的读取函数：根据写入方式读出
738. // 将HashMap的“总的容量，实际容量，所有的Entry”依次读出
739. **private** **void** readObject(java.io.ObjectInputStream s)
740. **throws** IOException, ClassNotFoundException
741. {
742. // Read in the threshold, loadfactor, and any hidden stuff
743. s.defaultReadObject();
745. // Read in number of buckets and allocate the bucket array;
746. **int** numBuckets = s.readInt();
747. table = **new** Entry[numBuckets];
749. init();  // Give subclass a chance to do its thing.
751. // Read in size (number of Mappings)
752. **int** size = s.readInt();
754. // Read the keys and values, and put the mappings in the HashMap
755. **for** (**int** i=0; i<size; i++) {
756. K key = (K) s.readObject();
757. V value = (V) s.readObject();
758. putForCreate(key, value);
759. }
760. }
762. // 返回“HashMap总的容量”
763. **int**   capacity()     { **return** table.length; }
764. // 返回“HashMap的加载因子”
765. **float** loadFactor()   { **return** loadFactor;   }
766. }

**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。

**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)