39-45行 注释：主要介绍了HashMap和HashTable的区别，即HashMap允许null作为键、值，而且HashMap不是线程安全的。并且HashMap中的元素不是有序的，特别的，也不保证随着时间推移，这个map中存储的顺序不发生改变。

解析：**hashmap是线程不安全的，键、值都允许null的存在，map中的元素不保证有序，随着时间推移可能还会发生变化。**

47-54行 注释：HashMap的get、put方法在能够正确将元素散列在桶中的情况下拥有常数级别的表现（即碰撞次数不过多）。而迭代器在Map中的表现情况依赖于初始化的容量与桶的数量+键值对的值的比例，即装载因子。因此，如果希望迭代器的表现良好，则不能够设置过大的初始容量。

我们知道HashMap的方法中，默认的装载因子大小是0.75，初始容量大小是16（旁边还有一句注释，必须是2的次方）

解析：**默认装载因子是0.75，默认初始容量是16**

56-65行 注释：就像上面提到的，影响HashMap性能的主要有两个因素，一个是初始容量的大小，一个是装载因子的大小。当HashMap中键值对的数量超过当前容量大小\*装载因子的时候，整个表会重新进行hash一次，并进行扩容（接近于原容量的两倍）。

解析：**每次resize的大小是将原容量扩充接近1倍。**

67-76行 注释： 介绍了采用0.75作为默认装载因子的意义。太高的装载因子会减少多余的空间，但是对查找性能很不友好。因此，最好是在初始化HashMap的时候根据它的键值对量对初始容量和装载因子进行相应的调整，确保让整个hashmap重新hash的次数最小化（即提高性能）。如果初始容量\*装载因子>键值对的数量，那么重新散列的情况就不会发生。

解析：定

78-85行 注释：如果Hashmap中的大多数键值对都已经有序了，那么给与它一个充分大的容量会比小的好（肯定啊。。。小了又要rehash）。如果说map中存放了大量的重复的键，会影响map的效率。为了改善影响，那么会在键之间采用比较来改善这种情况。

解析：**hashmap的散列方法采用的是拉链法，如果有大量重复主键的话，必然会导致碰撞的大量发生。**

87-94行 注释：介绍了hashmap的一个特点：线程不安全。如果有多个线程对Map进行增加/删除元素的话，需要在外部对map对象加锁。这个通常是由一些对象封装了相应的映射关系来完成的。

解析：**hashmap是线程不安全的，如果有并发操作需要在外部加上互斥锁。**

96-100行 注释：接着上面的继续讲，如果不存在这种封装了的对象，那么hashmap需要在初始化之初使用一个包装类：Collections.synchronizedMap

操作像这样：Map m = Collections.synchronizedMap(new HashMap(...));

解析：**一种使线程不安全的hashmap变得线程安全的操作。**

102-109行 注释：这里介绍了一下集合类迭代器的一个共同特性：快速失败。即当迭代器创建之后，所属的集合有任何结构上的变化（比如增加/删除元素，那种改变已有键值对的不算），都会抛出一个名为：ConcurrentModificationException的异常。除非是使用迭代器本身安全的remove()方法。

解析：迭代器本身有两个元素，一个ModCount，一个ExceptModCount，每次移动迭代器指针都会检查这两个的值是否相等，如果不等就会抛出快速失败的异常，而迭代器本身的人remove()方法会同步修改这两个值，则不会抛出异常。这个是所有集合类迭代器的共同特征。

110-117行 注释：**迭代器的快速失败特征是不可靠的**，应该说任何依赖于不同步情况下的并发修改都是难以保证正确性的。因此，迭代器的快速失败特征应该只被用于检查bug出在哪里，而不是保证程序的正确性。

145-154行 注释：hashmap虽然在很多实现的性能表现地像哈希表元素存储在桶里，但在map中桶的数量过多时，桶节点会转化成二叉树节点（红黑树）我们知道红黑树是平衡树的一种，它在数据很大时的查找性能很好。当然我们知道，方法大多数还是为一般情况下准备的，因此这种检验桶是否变成了树节点会一定程度上影响性能。

解析：HashMap在桶（一个单链表，因为采用的是**拉链法**）的**键簇**（即链表中的元素数目）较大时（这个值是8），**桶节点会变成树节点（红黑树）**，用于提高它的查找性能。但同时这种检验也难免会一定程度上影响平时的性能。（最好的查找性能是一个桶里只有一个元素，因为桶里存放的是发生碰撞的）

156-172行 注释：当桶里的节点全部变成树节点的时候，它们会主要通过比较hashcode变得有序。如果两个键之间都是实现了Comparable接口的（比如常用的原始数据类型+包装类+String都是实现了的），那么它们会通过compareTo()方法进行比较。虽然说这样对于本来hashcode就唯一/已经有序的键值对会比较浪费时间，但是对于hashcode()方法错误的分配以及很多键共享同一个hashcode的情况，这么做是值得的。

解析：对于树节点中的数据，**hashmap主要采用比较Hashcode的方法，如果键的类型实现了Comparable接口的方法，那么就会采用compareTo()使它有序**。我们知道，hashcode相等，不一定equals()，equals()不一定==。

174-186行 注释：因为树节点占用的空间比较大，近似普通节点的2倍，所以只有当空间足够的情况下才会进行转变（>=8），在小于6的时候又会转变回去。如果hashcode方法分配良好，在理想情况下，桶中转变为树节点的概率应当服从泊松分布。

解析：**转化为树节点的阈值是8，退化的阈值是6.树节点占用空间较大。**

199-202行 注释：根节点有时候可能不在树中（比如迭代器的remove方法），不过可以通过TreeNode.root()方法恢复。

204-209行 注释：所有适用的内部方法接收哈希码作为一个参数（通常来自公有方法），来避免对键哈希码的重新计算。许多内部方法也接收一个标签参数，通常是当前的表，在resize的时候也能是新的或旧的表。

解析：接收哈希码作为参数，避免重复计算。提高性能。

211-218行 注释：当哈希桶成树型/非树型以及分隔时，我们会保持它们处在同一种遍历顺序当中，并且一定程度上（较小）减轻迭代器操作的负担。在使用比较和插入的时候，为了保持总体的跨平衡有序，我们把键的类型和独有的hashcode作为连接桥梁。

220-226行 注释：桶是朴素桶还是红黑树桶的使用和转换，由于子类LinkedHashMap的存在而变得更加的复杂。hashmap中的一些由添加/删除触发得到钩子方法允许LinkedHashMap在其他情况下保留独立的内部特性。（笔者找到的钩子方法中的一个是void reinitialize()方法）它们也需要map对象实例通过一些实例方法创建新的节点

(

228-229行 注释：并发编程使用类似于SSA的风格，可以减少错误。（SSA：笔者查了一下，这是一种应用在JVM中的对代码的编译方法）

以上是hashmap的一些特性，接下来是源码的逐行分析阅读

232-235行 常量定义：默认的初始容量=16

237-242行 常量定义：hashmap的最大容量：1073741824（2^30） 必须是2的次方，如果在构造方法中隐式指定了更大的值，那么只会采用这个值。

244-247行 常量定义：默认装载因子=0.75

249-257行 常量定义：hash桶从链式存储转变成红黑树存储的阈值=8

解析：这个值必须>=2，并且至少为8,因为要应对节点删除导致又变回朴素桶的情况

259-264行 常量定义：hash桶由树桶转换为朴素桶的阈值=6

解析：在键簇小于这个值的时候，会在执行resize方法的时候变回朴素桶。

267-272行 常量定义：桶在可能转变为树桶的情况下，最小的容量=64

解析：这个值必须是（249）的四倍的倍数，树桶会占用更大的空间，如果小于这个值，会使resize方法和“树化”方法冲突。

274-开始 Node类定义：用于朴素键值对的类

1. hashCode()方法：将key的哈希码与value的哈希码做异或运算得到。
2. setValue(V newValue)：返回修改前的（旧的）Value值

解析：和大多数类一样 没有太特别的方法。

320-339行 终态静态方法：int hash(Object key)

解析：将key的hash值分为高位（前16位）和低位，将高位/低位做XOR（异或）运算。减少碰撞的可能性（比如Float型的哈希码就可能是连续的整数）

341-362行 静态方法：static Class<?> comparableClassFor(Object x) {}

解析：实现156-172行的效果。通过反射拿到参数的继承接口/类名，如果实现了Compareable接口，则返回参数的类型，否则返回Null.

364-372行 终态静态方法：static int compareComparables(Class<?> kc, Object k, Object x)

解析：实现156-172行提及的效果。对两个类型的参数进行比较（已经检查过K的类型为实现过Comparable接口）。

返回：如果X匹配参数KC的类型，返回compareTo方法的结果，否则返回0.

377-385行 终态静态方法：static final int tableSizeFor(int cap)

解析：对参数进行运算，将参数\*2，如果>=规定的最大容量（237-242），将返回最大容量值，而不是\*2后的值

返回值：2\*cap，或者是MAXIMUM\_CAPACITY（1<<30)

415行 字段：transient int modCount;

解析：每个集合类都有这么一个字段属性。用于记录这个集合更新的次数（比如map中键值对的数量发生改变，或者一些内部变化，比如rehash）。主要用于检验迭代器的快速失败情况。

446行 构造方法： public HashMap(int initialCapacity, float loadFactor)

解析：提供两个参数，分别用于赋值初始容量和装载因子。对两个值进行常规的检查，如果初始容量大于MAXIMUM\_CAPACITY,则赋值为MAXIMUM\_CAPACITY。装载因子不可为非数字或负数、0

487行 构造方法：public HashMap(Map<? extends K, ? extends V> m)

**[java]** [view plain](http://blog.csdn.net/a2370889_com/article/details/78882919) [copy](http://blog.csdn.net/a2370889_com/article/details/78882919)

1. **final** **void** putMapEntries(Map<? **extends** K, ? **extends** V> m, **boolean** evict) {
2. **int** s = m.size();
3. **if** (s > 0) {
4. **if** (table == **null**) { // pre-size
5. **float** ft = ((**float**)s / loadFactor) + 1.0F;
6. **int** t = ((ft < (**float**)MAXIMUM\_CAPACITY) ?
7. (**int**)ft : MAXIMUM\_CAPACITY);
8. **if** (t > threshold)
9. threshold = tableSizeFor(t);
10. }
11. **else** **if** (s > threshold)
12. resize();
13. **for** (Map.Entry<? **extends** K, ? **extends** V> e : m.entrySet()) {
14. K key = e.getKey();
15. V value = e.getValue();
16. putVal(hash(key), key, value, **false**, evict);
17. }
18. }
19. }

解析：将Map类型的参数内的所有键值对信息转换为Hashmap类型的，使用默认的装载因子0.75，并实现了Map.putAll()方法。

如果接收的对象已经被初始化了，那么直接判定

是否大于门槛值（当前容量\*装载因子），大于则resize()扩容;

如果没被初始化，那么table数组是为null，这时候获取当前需要转换map的能容纳的最小容量t（m.size()/0.75+1）,如果t>hashmap的门槛值threshold，那么将hashmap的门槛值置为map的两倍。因为如果threshold小于t的话，在putVal()的过程中会需要马上调用resize()方法，造成不必要的性能损失。

在完成以后，实现map的Put方法(putVal())，将键值对放入对应的hashmap实例中。

554行 实例方法:public V get(Object key)

**[java]** [view plain](http://blog.csdn.net/a2370889_com/article/details/78882919) [copy](http://blog.csdn.net/a2370889_com/article/details/78882919)

1. **public** V get(Object key) {
2. Node<K,V> e;
3. **return** (e = getNode(hash(key), key)) == **null** ? **null** : e.value;
4. }

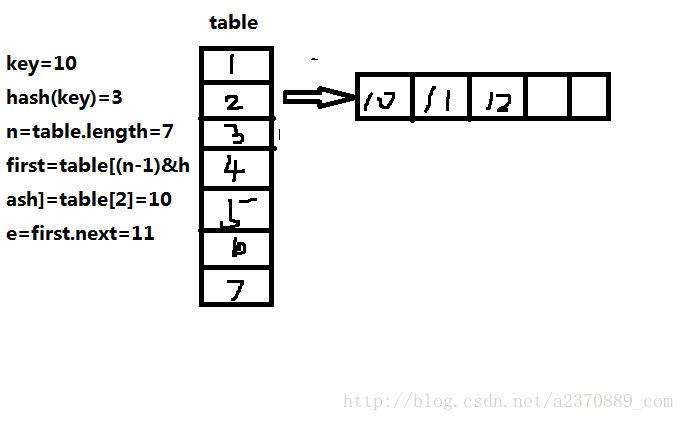
解析：即普通的获取map中的键值对映射的值，不过注意，因为hashmap允许键、值为null，所以get方法返回Null并不能作为不存在相对应键的判定，而是应该采用containsKey()方法来达到这个效果。

566行 终态方法：final Node<K,V> getNode(int hash, Object key)

**[java]** [view plain](http://blog.csdn.net/a2370889_com/article/details/78882919) [copy](http://blog.csdn.net/a2370889_com/article/details/78882919)

1. **final** Node<K,V> getNode(**int** hash, Object key) {
2. Node<K,V>[] tab; Node<K,V> first, e; **int** n; K k;
3. **if** ((tab = table) != **null** && (n = tab.length) > 0 &&
4. (first = tab[(n - 1) & hash]) != **null**) {
5. **if** (first.hash == hash && // always check first node
6. ((k = first.key) == key || (key != **null** && key.equals(k))))
7. **return** first;
8. **if** ((e = first.next) != **null**) {
9. **if** (first **instanceof** TreeNode)
10. **return** ((TreeNode<K,V>)first).getTreeNode(hash, key);
11. **do** {
12. **if** (e.hash == hash &&
13. ((k = e.key) == key || (key != **null** && key.equals(k))))
14. **return** e;
15. } **while** ((e = e.next) != **null**);
16. }
17. }
18. **return** **null**;
19. }

解析：该方法是HashMap的核心方法之一。

table是HashMap采用拉链法的数组，里面存放一条链表，链表内都是hash值相同，造成碰撞的元素。  
first是对应的key值对应的hash值，在table数组中对应hash值位置链表的第一个元素。如果要找的key就是这个first，则将该first键值对返回。

否则将first指向下一个链表元素节点，注意这时候可能链表并不是单链表，而是二叉链表即红黑树情况，所以要检查一下是否节点是TreeNode的实例。

如果是，则采用TreeNode的getTreeNode方法获取相应的键值对并返回。

否则，检查当前节点e的hash值，以及元素是否与查找本身==，如果相等，则返回当前节点键值对e,否则继续遍历链表。

经历过以上步骤如果仍没有返回，则没找到对应的键值对，返回Null。

594行 实例方法:

**[java]** [view plain](http://blog.csdn.net/a2370889_com/article/details/78882919) [copy](http://blog.csdn.net/a2370889_com/article/details/78882919)

1. **public** **boolean** containsKey(Object key) {
2. **return** getNode(hash(key), key) != **null**;
3. }

解析：本方法采用566行实例方法，这也是为什么containsKey能判断键是否存在。因为getNode不会返回Null，除非查找的元素不存在。

624行 终态方法 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,boolean exvict)

**[java]** [view plain](http://blog.csdn.net/a2370889_com/article/details/78882919) [copy](http://blog.csdn.net/a2370889_com/article/details/78882919)

1. **final** V putVal(**int** hash, K key, V value, **boolean** onlyIfAbsent,
2. **boolean** evict) {
3. Node<K,V>[] tab; Node<K,V> p; **int** n, i;
4. **if** ((tab = table) == **null** || (n = tab.length) == 0)
5. n = (tab = resize()).length;
6. **if** ((p = tab[i = (n - 1) & hash]) == **null**)
7. tab[i] = newNode(hash, key, value, **null**);
8. **else** {
9. Node<K,V> e; K k;
10. **if** (p.hash == hash &&
11. ((k = p.key) == key || (key != **null** && key.equals(k))))
12. e = p;
13. **else** **if** (p **instanceof** TreeNode)
14. e = ((TreeNode<K,V>)p).putTreeVal(**this**, tab, hash, key, value);
15. **else** {
16. **for** (**int** binCount = 0; ; ++binCount) {
17. **if** ((e = p.next) == **null**) {
18. p.next = newNode(hash, key, value, **null**);
19. **if** (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st
20. treeifyBin(tab, hash);
21. **break**;
22. }
23. **if** (e.hash == hash &&
24. ((k = e.key) == key || (key != **null** && key.equals(k))))
25. **break**;
26. p = e;
27. }
28. }
29. <span style="color:#ff0000;">**if** (e != **null**) { // existing mapping for key
30. V oldValue = e.value;
31. **if** (!onlyIfAbsent || oldValue == **null**)
32. e.value = value;
33. afterNodeAccess(e);
34. **return** oldValue;
35. }</span>
36. }
37. ++modCount;
38. **if** (++size > threshold)
39. resize();
40. afterNodeInsertion(evict);
41. **return** **null**;
42. }

解析：这是HashMap中最重要的方法之一。接收四个参数，其中后两个布尔型，一个代表在所对应键的值已经存在的情况下，是否执行覆盖;一个代表hashMap的table字段是否处于创建状态（前面提到，table仅在初始化时执行一次(不是终态是因为包括resize操作)）。而常用的Put()方法，调用时是**采取覆盖操作，不创建table.**

（1）和getNode方法类似，首先根据key的哈希码找到对应拉链数组的索引。

（2）如果对应的桶是红黑树桶，那么执行树节点的PutTreeVal，这里暂时不做介绍；否则在本方法内执行Put操作。

（3）定义一个int型bitCount，用于计算链表的长度，当链表长度达到treefy的门槛值(**8-1=7**，因为是执行Put操作后就会变成8，而前面已经提到过，**树化操作会在put/remove方法结束后执行**)时，桶将从朴素桶转变为红黑树桶（执行treeifyBin(tab, hash)操作）

（4）对朴素桶进行单链表的顺序查找，如果找到则跳出循环。

（5）跳出循环后进行判断，e(即查找指针p的next)是否为空。我们知道，这个for循环只有两种方式跳出，一种是e=p.next==null，即链表中没有找到所查找的键;另外一种是e==key，即找到了被查找的键。所以执行分支语句，**如果本身就存在这个键，那么将新值替换掉旧值，同时将旧值返回（标红处的语句实现了这个操作）**

（6）这个时候函数的主要功能put已经完成，接下来是一些hashmap本身属性的操作。比如++modcount，同步迭代器属性。对size进行判断，是否需要resize()操作，这些都是在插入操作（没有在上面返回证明键值对是插入的，而不是更新的）完成后进行的，因为改变hashmap本身的属性。

（7）后面执行一次LinkedHashMap继承的afterNodeInsertion方法，不过不会执行，因为传入的是false，该方法可能在插入后执行删除头结点的操作。1

676行 终态方法： final Node<K,V>[] resize()

**[java]** [view plain](http://blog.csdn.net/a2370889_com/article/details/78882919) [copy](http://blog.csdn.net/a2370889_com/article/details/78882919)

1. **final** Node<K,V>[] resize() {
2. Node<K,V>[] oldTab = table;
3. **int** oldCap = (oldTab == **null**) ? 0 : oldTab.length;
4. **int** oldThr = threshold;
5. **int** newCap, newThr = 0;
6. **if** (oldCap > 0) {
7. **if** (oldCap >= MAXIMUM\_CAPACITY) {
8. threshold = Integer.MAX\_VALUE;
9. **return** oldTab;
10. }
11. **else** **if** ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&
12. oldCap >= DEFAULT\_INITIAL\_CAPACITY)
13. newThr = oldThr << 1; // double threshold
14. }
15. **else** **if** (oldThr > 0) // initial capacity was placed in threshold
16. newCap = oldThr;
17. **else** {               // zero initial threshold signifies using defaults
18. newCap = DEFAULT\_INITIAL\_CAPACITY;
19. newThr = (**int**)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);
20. }
21. **if** (newThr == 0) {
22. **float** ft = (**float**)newCap \* loadFactor;
23. newThr = (newCap < MAXIMUM\_CAPACITY && ft < (**float**)MAXIMUM\_CAPACITY ?
24. (**int**)ft : Integer.MAX\_VALUE);
25. }
26. threshold = newThr;
27. @SuppressWarnings({"rawtypes","unchecked"})
28. Node<K,V>[] newTab = (Node<K,V>[])**new** Node[newCap];
29. table = newTab;
30. **if** (oldTab != **null**) {
31. **for** (**int** j = 0; j < oldCap; ++j) {
32. Node<K,V> e;
33. **if** ((e = oldTab[j]) != **null**) {
34. oldTab[j] = **null**;
35. **if** (e.next == **null**)
36. newTab[e.hash & (newCap - 1)] = e;
37. **else** **if** (e **instanceof** TreeNode)
38. ((TreeNode<K,V>)e).split(**this**, newTab, j, oldCap);
39. **else** { // preserve order
40. Node<K,V> loHead = **null**, loTail = **null**;
41. Node<K,V> hiHead = **null**, hiTail = **null**;
42. Node<K,V> next;
43. **do** {
44. next = e.next;

**[java]** [view plain](http://blog.csdn.net/a2370889_com/article/details/78882919) [copy](http://blog.csdn.net/a2370889_com/article/details/78882919)

1. <span style="white-space:pre;">         </span>//留在原数组索引位置
2. **if** ((e.hash & oldCap) == 0) {
3. **if** (loTail == **null**)
4. loHead = e;
5. **else**
6. loTail.next = e;
7. loTail = e;
8. }

**[java]** [view plain](http://blog.csdn.net/a2370889_com/article/details/78882919) [copy](http://blog.csdn.net/a2370889_com/article/details/78882919)

1. <span style="white-space:pre;">         </span>//挪到新位置，原索引+oldCap（n）
2. **else** {
3. **if** (hiTail == **null**)
4. hiHead = e;
5. **else**
6. hiTail.next = e;
7. hiTail = e;
8. }
9. } **while** ((e = next) != **null**);
10. **if** (loTail != **null**) {
11. loTail.next = **null**;
12. newTab[j] = loHead;
13. }
14. **if** (hiTail != **null**) {
15. hiTail.next = **null**;
16. newTab[j + oldCap] = hiHead;
17. }
18. }
19. }
20. }
21. }
22. **return** newTab;
23. }

解析：这是HashMap中核心方法之一，用于拓展map中键值对数组的大小（拉链法的数组）。如果还没有被初始化，那么table数组会被初始化为默认大小（16），否则会拓展为原大小的**两倍**。链表中的元素，因为是产生“碰撞”而进去的，所以会有相同的table索引值（table[(table.length-1)&hash]）;或者会分散在**偏移量为2的乘方**的新的table数组中

（1）如果扩展前的长度oldCap为正数，则对它进行判断：如果已经到了hashmap数组最大长度Max\_Value，那么将门槛值threshold置为Int型最大数，避免下次扩展操作，并返回原数组；如果oldCap小于最大长度值，则将oldCap扩大一倍并赋值给newCap。newCap<=最大长度且oldCap>=默认值16时，设定新门槛值newThr为旧门槛值的一倍；如果其中一个不满足，newThr仍旧为0。

（2）else if (oldThr > 0) 此时已经出了 oldCap>0的条件，所以我们知道，这个if判断还有个隐藏条件是oldCap==0，因为如果oldCap小于0，早在构造函数的时候已经抛出了异常。可以改写为（oldCap==0&&oldThr>0） newCap=oldThr.  将旧门槛值赋给了新数组长度

（3）else 此时的条件是：oldCap==0且oldThr==0  这时将默认容量和默认装载因子赋给新的

（4）if (newThr == 0)  为什么newThr会是0呢。回到（1）中，如果旧容量oldCap扩容两倍后大于等于最大长度或旧门槛值oldThr<16，都会使newThr没被赋值，还是初始的默认值。所以这会儿就要进行判断了，到底是哪个原因。后面计算了新的门槛值，满足条件（newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ）则新门槛值newThr=newCap\*loadFactor，否则也置为Int型最大值。

诶，为什么这里**只判断了其中一个条件**呢？笔者猜测是这个原因：如果oldCap<默认值16，那么可以知道必然是调用了有参数的构造方法指定了初始容量大小，而有参构造方法(int initCap,float factor)里面有一行（**456行**） threshold=tableSizefor(initCap),这个方法返回的是传入参数（即初始容量）的两倍。我们知道，默认装载因子不过0.75而已，即便容量翻一倍，也就1.5倍（原容量）大小的门槛值，而这里设定的已经是2倍了，newThr(1.5\*oldCap)<oldThr(2\*oldCap)自然不用管它。到这里resize方法关于hashmap本身属性值改变就完成。

（5）这时就要开始对原数组的数据“搬家”到新数组了，由于指针膨胀的原因，所以要重新计算一次原数组中数据对应到新数组中hash的索引。

（6）

**[java]** [view plain](http://blog.csdn.net/a2370889_com/article/details/78882919) [copy](http://blog.csdn.net/a2370889_com/article/details/78882919)

1. **if** (e.next == **null**)
2. newTab[e.hash & (newCap - 1)] = e;
3. **else** **if** (e **instanceof** TreeNode)
4. ((TreeNode<K,V>)e).split(**this**, newTab, j, oldCap);

呐，可以看到，如果链表里面只有一个节点（即没有发生碰撞），那肯定是朴素桶了，直接重新计算索引值搬家即可；否则，发生了碰撞，判断当前节点是不是树节点，如果是，执行树节点的分散方法。(注：本方法只会被resize方法调用)

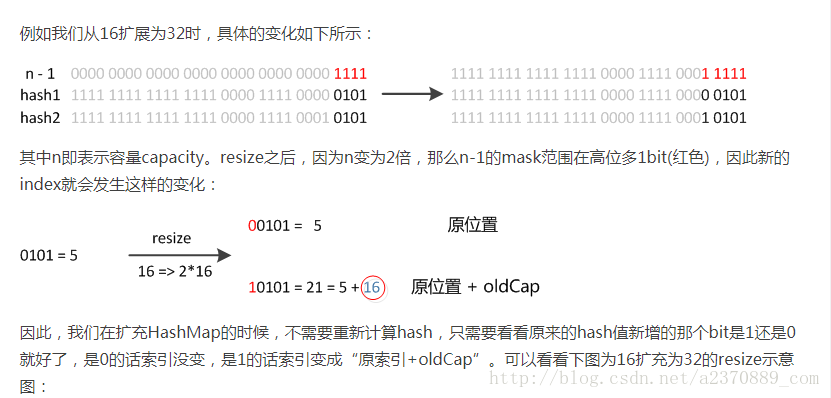
（7）720行：  if ((e.hash & oldCap) == 0)

体现智商的时候来了，很明显- - 笔者的智商不是特别够，还好有很多大佬珠玉在前，这篇博文这行介绍的非常好：[点击打开链接](http://blog.csdn.net/fan2012huan/article/details/51233540) 同时，这个地方也是1.8与1.7差别最大的地方之一，hashmap省去了1.7的rehash操作，提高了性能，同时也降低了碰撞的可能性。

笔者就简单的在这里讲一讲他的思想，这个方法的目的是为了避免重新计算hash值

我们知道，hashmap扩充是2的乘方，而求索引是table[hash&(n-1)]，其中n是数组的长度，hash是键的哈希码。我们知道，hash和n都是int型的，32位长度。而n变成了2n，实际上就是高一位的值变了，可能是0，也可能是1。那么(hash&n-1)的值实际上就是要么没变（还是原索引）;要么变了（原索引+oldCap）。

而**为什么是e.hash&oldCap**(其实就是n，而不是n-1了)。根据上面的话我们知道，实际上就是判断新增加的高位到底是0，还是1.如果是0，则留在原位置不动；如果是1，则需要挪到+oldCap处。而hashmap的Capacity一直保持为2的幂次，所以可以理解为n比n-1向前进了一位，即只针对n处，判断高位是0还是1，而分散到相应的数组位置。  如果还没懂的同学可以再看看这一篇文章：<https://www.zhihu.com/question/28365219>



这也是为什么前面注释会提到说链表中的元素会被均匀的分散到偏移量为2的乘方的新数组中，因为要么是留在元数组中，要么是原索引+oldCap(扩大了oldCap的一倍，即\*2)。同时我们可以近似的看做，e.hash相对高位的0、1表现是随机的，这样就更加减小了碰撞的可能性，散列的更加均匀。

（8）

然后根据前面的这个判断，将单链表中元素分为了两串lo(不需要移动的)、hi(需要移动的)。

定义了两个变量head/tail，Head是串单链表的头部，tail是串的尾部，用tail将链表串接起来。

用尾插法将 会处于新数组中的元素用尾插法串联起来，当达到末尾的时候，执行操作，将单链表放入新数组相应的索引位置。整个resize()方法结束。