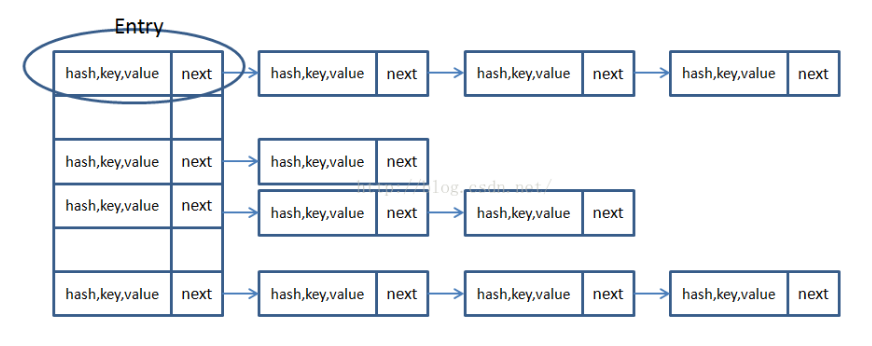
**HashMap底层原理分析**

**首先有一个链表，该链表的每个元素都是数组，当添加一个元素（key-value）时，就首先计算元素key的hash值，以此确定插入数组中的位置，但是可能存在同一hash值的元素已经被放在数组同一位置了，这时就添加到同一hash值的元素的后面，他们在数组的同一位置，但是形成了链表，同一个链表上的Hash值是相同的，所以说数组存放的是链表。而当链表长度太长时，链表就转换为红黑树，这样大大提高了查找的效率。**

  当链表数组的容量超过初始容量的0.75时，再散列将链表数组扩大2倍，把原链表数组的搬移到新的数组中

即HashMap的原理图是：



# 一，JDK1.8中的涉及到的数据结构

## 1，位桶数组

1. **transient** Node<k,v>[] table;//存储（位桶）的数组</k,v>

## 2，数组元素Node<K,V>实现了Entry接口

1. //Node是单向链表，它实现了Map.Entry接口
2. **static** **class** Node<k,v> **implements** Map.Entry<k,v> {
3. **final** **int** hash;
4. **final** K key;
5. V value;
6. Node<k,v> next;
7. //构造函数Hash值 键 值 下一个节点
8. Node(**int** hash, K key, V value, Node<k,v> next) {
9. **this**.hash = hash;
10. **this**.key = key;
11. **this**.value = value;
12. **this**.next = next;
13. }
15. **public** **final** K getKey()        { **return** key; }
16. **public** **final** V getValue()      { **return** value; }
17. **public** **final** String toString() { **return** key + = + value; }
19. **public** **final** **int** hashCode() {
20. **return** Objects.hashCode(key) ^ Objects.hashCode(value);
21. }
23. **public** **final** V setValue(V newValue) {
24. V oldValue = value;
25. value = newValue;
26. **return** oldValue;
27. }
28. //判断两个node是否相等,若key和value都相等，返回true。可以与自身比较为true
29. **public** **final** **boolean** equals(Object o) {
30. **if** (o == **this**)
31. **return** **true**;
32. **if** (o **instanceof** Map.Entry) {
33. Map.Entry<!--?,?--> e = (Map.Entry<!--?,?-->)o;
34. **if** (Objects.equals(key, e.getKey()) &&
35. Objects.equals(value, e.getValue()))
36. **return** **true**;
37. }
38. **return** **false**;
39. }

3，红黑树

**[java]** [view plain](http://blog.csdn.net/tuke_tuke/article/details/51588156) [copy](http://blog.csdn.net/tuke_tuke/article/details/51588156)

1. //红黑树
2. **static** **final** **class** TreeNode<k,v> **extends** LinkedHashMap.Entry<k,v> {
3. TreeNode<k,v> parent;  // 父节点
4. TreeNode<k,v> left; //左子树
5. TreeNode<k,v> right;//右子树
6. TreeNode<k,v> prev;    // needed to unlink next upon deletion
7. **boolean** red;    //颜色属性
8. TreeNode(**int** hash, K key, V val, Node<k,v> next) {
9. **super**(hash, key, val, next);
10. }
12. //返回当前节点的根节点
13. **final** TreeNode<k,v> root() {
14. **for** (TreeNode<k,v> r = **this**, p;;) {
15. **if** ((p = r.parent) == **null**)
16. **return** r;
17. r = p;
18. }
19. }

**二，源码中的数据域**

加载因子（默认0.75）：为什么需要使用加载因子，为什么需要扩容呢？**因为如果填充比很大，说明利用的空间很多，如果一直不进行扩容的话，链表就会越来越长，这样查找的效率很低，因为链表的长度很大（当然最新版本使用了红黑树后会改进很多），扩容之后，将原来链表数组的每一个链表分成奇偶两个子链表分别挂在新链表数组的散列位置，这样就减少了每个链表的长度，增加查找效率**

HashMap本来是以空间换时间，所以填充比没必要太大。但是填充比太小又会导致空间浪费。如果关注内存，填充比可以稍大，如果主要关注查找性能，填充比可以稍小。

**[java]** [view plain](http://blog.csdn.net/tuke_tuke/article/details/51588156) [copy](http://blog.csdn.net/tuke_tuke/article/details/51588156)

1. **public** **class** HashMap<k,v> **extends** AbstractMap<k,v> **implements** Map<k,v>, Cloneable, Serializable {
2. **private** **static** **final** **long** serialVersionUID = 362498820763181265L;
3. **static** **final** **int** DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // aka 16
4. **static** **final** **int** MAXIMUM\_CAPACITY = 1 << 30;//最大容量
5. **static** **final** **float** DEFAULT\_LOAD\_FACTOR = 0.75f;//填充比
6. //当add一个元素到某个位桶，其链表长度达到8时将链表转换为红黑树
7. **static** **final** **int** TREEIFY\_THRESHOLD = 8;
8. **static** **final** **int** UNTREEIFY\_THRESHOLD = 6;
9. **static** **final** **int** MIN\_TREEIFY\_CAPACITY = 64;
10. **transient** Node<k,v>[] table;//存储元素的数组
11. **transient** Set<map.entry<k,v>> entrySet;
12. **transient** **int** size;//存放元素的个数
13. **transient** **int** modCount;//被修改的次数fast-fail机制
14. **int** threshold;//临界值 当实际大小(容量\*填充比)超过临界值时，会进行扩容
15. **final** **float** loadFactor;//填充比（......后面略）

**三，HashMap的构造函数**

HashMap的构造方法有4种，主要涉及到的参数有，指定初始容量，指定填充比和用来初始化的Map

**[java]** [view plain](http://blog.csdn.net/tuke_tuke/article/details/51588156) [copy](http://blog.csdn.net/tuke_tuke/article/details/51588156)

1. //构造函数1
2. **public** HashMap(**int** initialCapacity, **float** loadFactor) {
3. //指定的初始容量非负
4. **if** (initialCapacity < 0)
5. **throw** **new** IllegalArgumentException(Illegal initial capacity:  +
6. initialCapacity);
7. //如果指定的初始容量大于最大容量,置为最大容量
8. **if** (initialCapacity > MAXIMUM\_CAPACITY)
9. initialCapacity = MAXIMUM\_CAPACITY;
10. //填充比为正
11. **if** (loadFactor <= 0 || Float.isNaN(loadFactor))
12. **throw** **new** IllegalArgumentException(Illegal load factor:  +
13. loadFactor);
14. **this**.loadFactor = loadFactor;
15. **this**.threshold = tableSizeFor(initialCapacity);//新的扩容临界值
16. }
18. //构造函数2
19. **public** HashMap(**int** initialCapacity) {
20. **this**(initialCapacity, DEFAULT\_LOAD\_FACTOR);
21. }
23. //构造函数3
24. **public** HashMap() {
25. **this**.loadFactor = DEFAULT\_LOAD\_FACTOR; // all other fields defaulted
26. }
28. //构造函数4用m的元素初始化散列映射
29. **public** HashMap(Map<!--? **extends** K, ? **extends** V--> m) {
30. **this**.loadFactor = DEFAULT\_LOAD\_FACTOR;
31. putMapEntries(m, **false**);
32. }

**四，HashMap的存取机制**

1，HashMap如何getValue值，看源码

**[java]** [view plain](http://blog.csdn.net/tuke_tuke/article/details/51588156) [copy](http://blog.csdn.net/tuke_tuke/article/details/51588156)

1. **public** V get(Object key) {
2. Node<K,V> e;
3. **return** (e = getNode(hash(key), key)) == **null** ? **null** : e.value;
4. }
5. /\*\*
6. \* Implements Map.get and related methods
7. \*
8. \* @param hash hash for key
9. \* @param key the key
10. \* @return the node, or null if none
11. \*/
12. **final** Node<K,V> getNode(**int** hash, Object key) {
13. Node<K,V>[] tab;//Entry对象数组
14. Node<K,V> first,e; //在tab数组中经过散列的第一个位置
15. **int** n;
16. K k;
17. /\*找到插入的第一个Node，方法是hash值和n-1相与，tab[(n - 1) & hash]\*/
18. //也就是说在一条链上的hash值相同的
19. **if** ((tab = table) != **null** && (n = tab.length) > 0 &&(first = tab[(n - 1) & hash]) != **null**) {
20. /\*检查第一个Node是不是要找的Node\*/
21. **if** (first.hash == hash && // always check first node
22. ((k = first.key) == key || (key != **null** && key.equals(k))))//判断条件是hash值要相同，key值要相同
23. **return** first;
24. /\*检查first后面的node\*/
25. **if** ((e = first.next) != **null**) {
26. **if** (first **instanceof** TreeNode)
27. **return** ((TreeNode<K,V>)first).getTreeNode(hash, key);
28. /\*遍历后面的链表，找到key值和hash值都相同的Node\*/
29. **do** {
30. **if** (e.hash == hash &&
31. ((k = e.key) == key || (key != **null** && key.equals(k))))
32. **return** e;
33. } **while** ((e = e.next) != **null**);
34. }
35. }
36. **return** **null**;
37. }

get(key)方法时获取key的hash值，计算hash&(n-1)得到在链表数组中的位置first=tab[hash&(n-1)],先判断first的key是否与参数key相等，不等就遍历后面的链表找到相同的key值返回对应的Value值即可

**2，**HashMap如何put(key，value);看源码

**[java]** [view plain](http://blog.csdn.net/tuke_tuke/article/details/51588156) [copy](http://blog.csdn.net/tuke_tuke/article/details/51588156)

1. **public** V put(K key, V value) {
2. **return** putVal(hash(key), key, value, **false**, **true**);
3. }
4. /\*\*
5. \* Implements Map.put and related methods
6. \*
7. \* @param hash hash for key
8. \* @param key the key
9. \* @param value the value to put
10. \* @param onlyIfAbsent if true, don't change existing value
11. \* @param evict if false, the table is in creation mode.
12. \* @return previous value, or null if none
13. \*/
14. **final** V putVal(**int** hash, K key, V value, **boolean** onlyIfAbsent,
15. **boolean** evict) {
16. Node<K,V>[] tab;
17. Node<K,V> p;
18. **int** n, i;
19. **if** ((tab = table) == **null** || (n = tab.length) == 0)
20. n = (tab = resize()).length;
21. /\*如果table的在（n-1）&hash的值是空，就新建一个节点插入在该位置\*/
22. **if** ((p = tab[i = (n - 1) & hash]) == **null**)
23. tab[i] = newNode(hash, key, value, **null**);
24. /\*表示有冲突,开始处理冲突\*/
25. **else** {
26. Node<K,V> e;
27. K k;
28. /\*检查第一个Node，p是不是要找的值\*/
29. **if** (p.hash == hash &&((k = p.key) == key || (key != **null** && key.equals(k))))
30. e = p;
31. **else** **if** (p **instanceof** TreeNode)
32. e = ((TreeNode<K,V>)p).putTreeVal(**this**, tab, hash, key, value);
33. **else** {
34. **for** (**int** binCount = 0; ; ++binCount) {
35. /\*指针为空就挂在后面\*/
36. **if** ((e = p.next) == **null**) {
37. p.next = newNode(hash, key, value, **null**);
38. //如果冲突的节点数已经达到8个，看是否需要改变冲突节点的存储结构，
39. //treeifyBin首先判断当前hashMap的长度，如果不足64，只进行
40. //resize，扩容table，如果达到64，那么将冲突的存储结构为红黑树
41. **if** (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st
42. treeifyBin(tab, hash);
43. **break**;
44. }
45. /\*如果有相同的key值就结束遍历\*/
46. **if** (e.hash == hash &&((k = e.key) == key || (key != **null** && key.equals(k))))
47. **break**;
48. p = e;
49. }
50. }
51. /\*就是链表上有相同的key值\*/
52. **if** (e != **null**) { // existing mapping for key，就是key的Value存在
53. V oldValue = e.value;
54. **if** (!onlyIfAbsent || oldValue == **null**)
55. e.value = value;
56. afterNodeAccess(e);
57. **return** oldValue;//返回存在的Value值
58. }
59. }
60. ++modCount;
61. /\*如果当前大小大于门限，门限原本是初始容量\*0.75\*/
62. **if** (++size > threshold)
63. resize();//扩容两倍
64. afterNodeInsertion(evict);
65. **return** **null**;
66. }

下面简单说下添加键值对put(key,value)的过程：  
1，判断键值对数组tab[]是否为空或为null，否则以默认大小resize()；  
2，根据键值key计算hash值得到插入的数组索引i，如果tab[i]==null，直接新建节点添加，否则转入3  
3，判断当前数组中处理hash冲突的方式为链表还是红黑树(check第一个节点类型即可),分别处理

**五，HasMap的扩容机制resize();**

构造hash表时，如果不指明初始大小，默认大小为16（即Node数组大小16），如果Node[]数组中的元素达到（填充比\*Node.length）重新调整HashMap大小 变为原来2倍大小,扩容很耗时

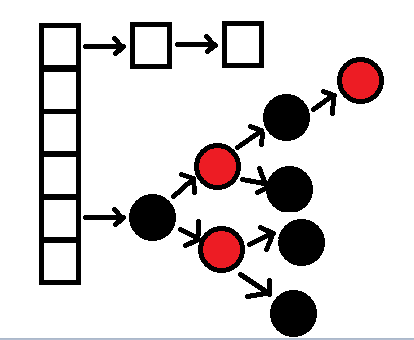
**[java]** [view plain](http://blog.csdn.net/tuke_tuke/article/details/51588156) [copy](http://blog.csdn.net/tuke_tuke/article/details/51588156)

1. /\*\*
2. \* Initializes or doubles table size.  If null, allocates in
3. \* accord with initial capacity target held in field threshold.
4. \* Otherwise, because we are using power-of-two expansion, the
5. \* elements from each bin must either stay at same index, or move
6. \* with a power of two offset in the new table.
7. \*
8. \* @return the table
9. \*/
10. **final** Node<K,V>[] resize() {
11. Node<K,V>[] oldTab = table;
12. **int** oldCap = (oldTab == **null**) ? 0 : oldTab.length;
13. **int** oldThr = threshold;
14. **int** newCap, newThr = 0;
16. /\*如果旧表的长度不是空\*/
17. **if** (oldCap > 0) {
18. **if** (oldCap >= MAXIMUM\_CAPACITY) {
19. threshold = Integer.MAX\_VALUE;
20. **return** oldTab;
21. }
22. /\*把新表的长度设置为旧表长度的两倍，newCap=2\*oldCap\*/
23. **else** **if** ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&
24. oldCap >= DEFAULT\_INITIAL\_CAPACITY)
25. /\*把新表的门限设置为旧表门限的两倍，newThr=oldThr\*2\*/
26. newThr = oldThr << 1; // double threshold
27. }
28. /\*如果旧表的长度的是0，就是说第一次初始化表\*/
29. **else** **if** (oldThr > 0) // initial capacity was placed in threshold
30. newCap = oldThr;
31. **else** {               // zero initial threshold signifies using defaults
32. newCap = DEFAULT\_INITIAL\_CAPACITY;
33. newThr = (**int**)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);
34. }


38. **if** (newThr == 0) {
39. **float** ft = (**float**)newCap \* loadFactor;//新表长度乘以加载因子
40. newThr = (newCap < MAXIMUM\_CAPACITY && ft < (**float**)MAXIMUM\_CAPACITY ?
41. (**int**)ft : Integer.MAX\_VALUE);
42. }
43. threshold = newThr;
44. @SuppressWarnings({"rawtypes","unchecked"})
45. /\*下面开始构造新表，初始化表中的数据\*/
46. Node<K,V>[] newTab = (Node<K,V>[])**new** Node[newCap];
47. table = newTab;//把新表赋值给table
48. **if** (oldTab != **null**) {//原表不是空要把原表中数据移动到新表中
49. /\*遍历原来的旧表\*/
50. **for** (**int** j = 0; j < oldCap; ++j) {
51. Node<K,V> e;
52. **if** ((e = oldTab[j]) != **null**) {
53. oldTab[j] = **null**;
54. **if** (e.next == **null**)//说明这个node没有链表直接放在新表的e.hash & (newCap - 1)位置
55. newTab[e.hash & (newCap - 1)] = e;
56. **else** **if** (e **instanceof** TreeNode)
57. ((TreeNode<K,V>)e).split(**this**, newTab, j, oldCap);
58. /\*如果e后边有链表,到这里表示e后面带着个单链表，需要遍历单链表，将每个结点重\*/
59. **else** { // preserve order保证顺序
60. ////新计算在新表的位置，并进行搬运
61. Node<K,V> loHead = **null**, loTail = **null**;
62. Node<K,V> hiHead = **null**, hiTail = **null**;
63. Node<K,V> next;
65. **do** {
66. next = e.next;//记录下一个结点
67. //新表是旧表的两倍容量，实例上就把单链表拆分为两队，
68. //e.hash&oldCap为偶数一队，e.hash&oldCap为奇数一对
69. **if** ((e.hash & oldCap) == 0) {
70. **if** (loTail == **null**)
71. loHead = e;
72. **else**
73. loTail.next = e;
74. loTail = e;
75. }
76. **else** {
77. **if** (hiTail == **null**)
78. hiHead = e;
79. **else**
80. hiTail.next = e;
81. hiTail = e;
82. }
83. } **while** ((e = next) != **null**);
85. **if** (loTail != **null**) {//lo队不为null，放在新表原位置
86. loTail.next = **null**;
87. newTab[j] = loHead;
88. }
89. **if** (hiTail != **null**) {//hi队不为null，放在新表j+oldCap位置
90. hiTail.next = **null**;
91. newTab[j + oldCap] = hiHead;
92. }
93. }
94. }
95. }
96. }
97. **return** newTab;
98. }

**六，JDK1.8使用红黑树的改进**

在java jdk8中对HashMap的源码进行了优化，在jdk7中，HashMap处理“碰撞”的时候，都是采用链表来存储，当碰撞的结点很多时，查询时间是O（n）。  
在jdk8中，HashMap处理“碰撞”增加了红黑树这种数据结构，当碰撞结点较少时，采用链表存储，当较大时（>8个），采用红黑树（特点是查询时间是O（logn））存储（有一个阀值控制，大于阀值(8个)，将链表存储转换成红黑树存储）



问题分析：

你可能还知道哈希碰撞会对hashMap的性能带来灾难性的影响。如果多个hashCode()的值落到同一个桶内的时候，这些值是存储到一个链表中的。最坏的情况下，所有的key都映射到同一个桶中，这样hashmap就退化成了一个链表——查找时间从O(1)到O(n)。

随着HashMap的大小的增长，get()方法的开销也越来越大。由于所有的记录都在同一个桶里的超长链表内，平均查询一条记录就需要遍历一半的列表。

**JDK1.8HashMap的红黑树是这样解决的**：

         如果某个桶中的记录过大的话（当前是TREEIFY\_THRESHOLD = 8），HashMap会动态的使用一个专门的treemap实现来替换掉它。这样做的结果会更好，是O(logn)，而不是糟糕的O(n)。

        它是如何工作的？前面产生冲突的那些KEY对应的记录只是简单的追加到一个链表后面，这些记录只能通过遍历来进行查找。但是超过这个阈值后HashMap开始将列表升级成一个二叉树，**使用哈希值作为树的分支变量，如果两个哈希值不等，但指向同一个桶的话，较大的那个会插入到右子树里。如果哈希值相等，HashMap希望key值最好是实现了Comparable接口的，这样它可以按照顺序来进行插入**。这对HashMap的key来说并不是必须的，不过如果实现了当然最好。如果没有实现这个接口，在出现严重的哈希碰撞的时候，你就并别指望能获得性能提升了。