*摘要: 本文主要介绍HashMap在JDK7和JDK8中的实现*

## JDK7中的HashMap

HashMap底层维护一个数组，数组中的每一项都是一个Entry

transient Entry<K,V>[] table;

我们向 HashMap 中所放置的对象实际上是存储在该数组当中；

而Map中的key，value则以Entry的形式存放在数组中

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

int hash;

而这个Entry应该放在数组的哪一个位置上（这个位置通常称为位桶或者hash桶，即hash值相同的Entry会放在同一位置，用链表相连），是通过key的hashCode来计算的。

final int hash(Object k) {

int h = 0;

h ^= k.hashCode();

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

通过hash计算出来的值将会使用indexFor方法找到它应该所在的table下标：

static int indexFor(int h, int length) {

return h & (length-1);

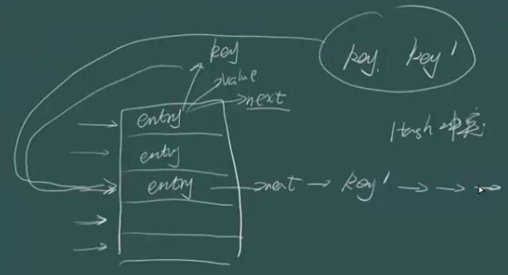
}

这个方法其实相当于对table.length取模。

当两个key通过hashCode计算相同时，则发生了hash冲突(碰撞)，HashMap解决hash冲突的方式是用链表。

当发生hash冲突时，则将存放在数组中的Entry设置为新值的next（这里要注意的是，比如A和B都hash后都映射到下标i中，之前已经有A了，当map.put(B)时，将B放到下标i中，A则为B的next，所以新值存放在数组中，旧值在新值的链表上）

示意图：



所以当hash冲突很多时，HashMap退化成链表。

总结一下map.put后的过程：

当向 HashMap 中 put 一对键值时，它会根据 key的 hashCode 值计算出一个位置， 该位置就是此对象准备往数组中存放的位置。

如果该位置没有对象存在，就将此对象直接放进数组当中；如果该位置已经有对象存在了，则顺着此存在的对象的链开始寻找(为了判断是否是否值相同，map不允许<key,value>键值对重复)， 如果此链上有对象的话，再去使用 equals方法进行比较，如果对此链上的每个对象的 equals 方法比较都为 false，则将该对象放到数组当中，然后将数组中该位置以前存在的那个对象链接到此对象的后面。

值得注意的是，当key为null时，都放到table[0]中

private V putForNullKey(V value) {

for (Entry<K,V> e = table[0]; e != null; e = e.next) {

if (e.key == null) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(0, null, value, 0);

return null;

}

当size大于threshold时，会发生扩容。 threshold等于capacity\*load factor

void addEntry(int hash, K key, V value, int bucketIndex) {

if ((size >= threshold) && (null != table[bucketIndex])) {

resize(2 \* table.length);

hash = (null != key) ? hash(key) : 0;

bucketIndex = indexFor(hash, table.length);

}

createEntry(hash, key, value, bucketIndex);

}

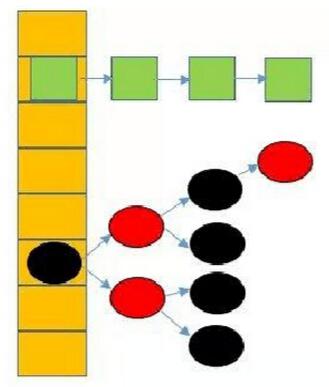
jdk7中resize，只有当 size>=threshold并且 table中的那个槽中已经有Entry时，才会发生resize。即有可能虽然size>=threshold，但是必须等到每个槽都至少有一个Entry时，才会扩容。还有注意每次resize都会扩大一倍容量

## JDK8中的HashMap

一直到JDK7为止，HashMap的结构都是这么简单，基于一个数组以及多个链表的实现，hash值冲突的时候，就将对应节点以链表的形式存储。

这样子的HashMap性能上就抱有一定疑问，如果说成百上千个节点在hash时发生碰撞，存储一个链表中，那么如果要查找其中一个节点，那就不可避免的花费O(N)的查找时间，这将是多么大的性能损失。这个问题终于在JDK8中得到了解决。再最坏的情况下，链表查找的时间复杂度为O(n),而红黑树一直是O(logn),这样会提高HashMap的效率。

JDK7中HashMap采用的是位桶+链表的方式，即我们常说的散列链表的方式，而JDK8中采用的是位桶+链表/红黑树（有关红黑树请查看[红黑树](http://my.oschina.net/hosee/blog/618828" \t "https://my.oschina.net/hosee/blog/_blank)）的方式，也是非线程安全的。当某个位桶的链表的长度达到某个阀值的时候，这个链表就将转换成红黑树。



JDK8中，当同一个hash值的节点数不小于8时，将不再以单链表的形式存储了，会被调整成一颗红黑树（上图中null节点没画）。这就是JDK7与JDK8中HashMap实现的最大区别。

接下来，我们来看下JDK8中HashMap的源码实现。

JDK中Entry的名字变成了Node，原因是和红黑树的实现TreeNode相关联。

transient Node<K,V>[] table;

当冲突节点数不小于8-1时，转换成红黑树。

static final int TREEIFY\_THRESHOLD = 8;

以put方法在JDK8中有了很大的改变

public V put(K key, V value) {

return putVal(hash(key), key, value, false, true);

}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,

boolean evict) {

Node<K,V>[] tab;

Node<K,V> p;

int n, i;

//如果当前map中无数据，执行resize方法。并且返回n

if ((tab = table) == null || (n = tab.length) == 0)

n = (tab = resize()).length;

//如果要插入的键值对要存放的这个位置刚好没有元素，那么把他封装成Node对象，放在这个位置上就完事了

if ((p = tab[i = (n - 1) & hash]) == null)

tab[i] = newNode(hash, key, value, null);

//否则的话，说明这上面有元素

else {

Node<K,V> e; K k;

//如果这个元素的key与要插入的一样，那么就替换一下，也完事。

if (p.hash == hash &&

((k = p.key) == key || (key != null && key.equals(k))))

e = p;

//1.如果当前节点是TreeNode类型的数据，执行putTreeVal方法

else if (p instanceof TreeNode)

e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

else {

//还是遍历这条链子上的数据，跟jdk7没什么区别

for (int binCount = 0; ; ++binCount) {

if ((e = p.next) == null) {

p.next = newNode(hash, key, value, null);

//2.完成了操作后多做了一件事情，判断，并且可能执行treeifyBin方法

if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st

treeifyBin(tab, hash);

break;

}

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

break;

p = e;

}

}

if (e != null) { // existing mapping for key

V oldValue = e.value;

if (!onlyIfAbsent || oldValue == null) //true || --

e.value = value;

//3.

afterNodeAccess(e);

return oldValue;

}

}

++modCount;

//判断阈值，决定是否扩容

if (++size > threshold)

resize();

//4.

afterNodeInsertion(evict);

return null;

}

treeifyBin()就是将链表转换成红黑树。

之前的indefFor()方法消失 了，直接用(tab.length-1)&hash，所以看到这个，代表的就是数组的下角标。

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

## Reference：

1. http://www.tuicool.com/articles/Yruqiye

2. http://wenku.baidu.com/link?url=qRXqFTKcObVZATjznA97yNw8zMdsxNsX20sLAyn40YmUqF43QVf\_yIPB97U33qMT36mtDaEzzuBHev5zCzr1jfJ2SZHjufV4LdEVzGHZ2T3

3. https://segmentfault.com/a/1190000003617333

4. http://blog.csdn.net/q291611265/article/details/46797557