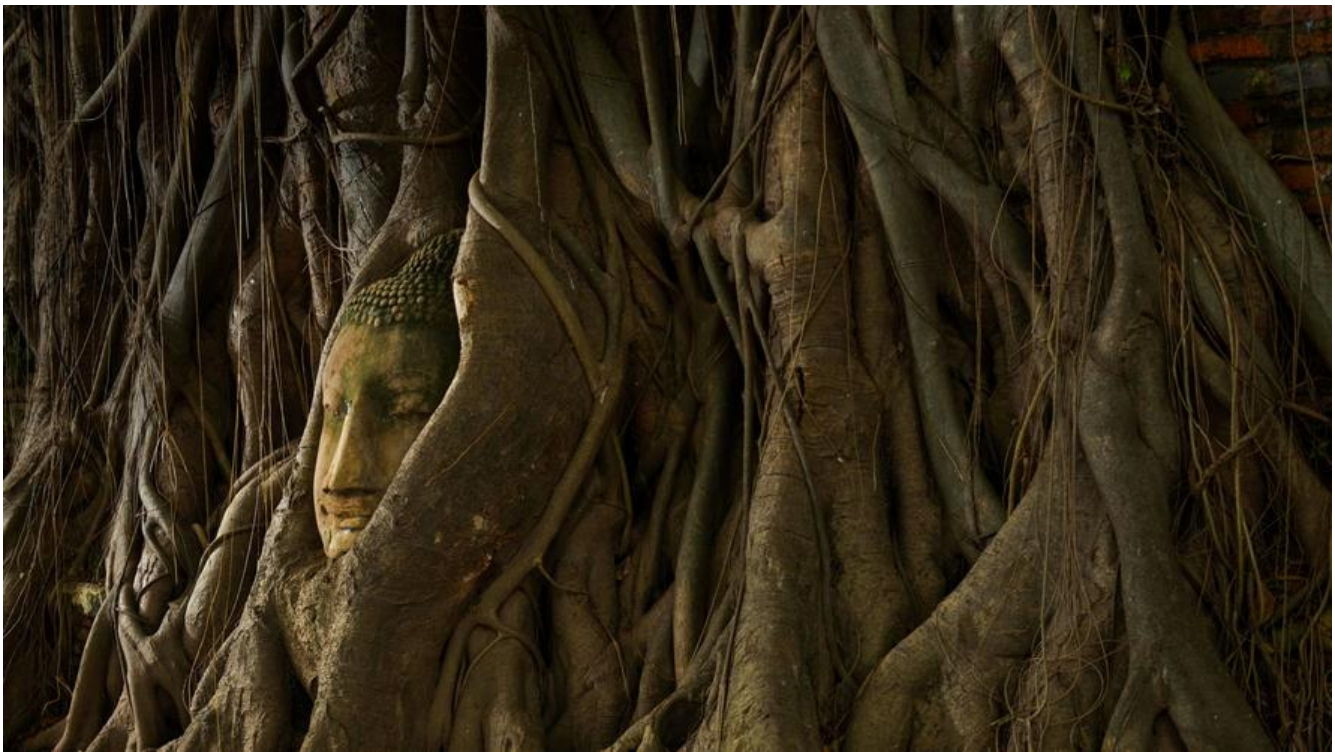
HashMap 常用函数源码解读

作者：18380422102
原文链接：https://ld246.com/article/1534846223483
来源网站：链滴



1.常量信息

```
//默认容量16，必须是2的幂
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
//最大容量
static final int MAXIMUM_CAPACITY = 1 << 30;
//负载因子 即已使用容量超过75%就会触发扩容
static final float DEFAULT_LOAD_FACTOR = 0.75f;
//节点链表长度达到8，会触发由链表变为红黑树的操作
static final int TREEIFY_THRESHOLD = 8;
//红黑树的节点被删除后长度小于6，会触发红黑树变链表的操作
static final int UNTREEIFY_THRESHOLD = 6;
//链表变红黑树要满足容器容量大于64，否则会选择扩容而不是变为红黑树
static final int MIN_TREEIFY_CAPACITY = 64;
```

2.put方法

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node[] tab; Node p; int n, i;
//如果容器为空，则先调用扩容方法，进行一个初始化容器的操作
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
// 如果容器中相同hash的位置为空，则在此位置新建一个节点
// hashMap中获取元素位置的方式为元素hash值与容量为模，即hash除容量的余数
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node e; K k;
//如果该位置不为空，且key值引用或内容相同，则替换元素
```

```
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
//如果节点是树节点，则执行树节点插入方法
        else if (p instanceof TreeNode)
            e = ((TreeNode)p).putTreeVal(this, tab, hash, key, value);
        else {
//如果节点是链表节点，则先遍历链表
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
//插入节点后容量满足TREEIFY_THRESHOLD ，则将该条链表变为树形结构
                    if (binCount >= TREEIFY_THRESHOLD - 1)
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
//如果链表节点的key和输入的key相同，则替换元素
                    break;
                p = e;
            }
        }
//替换元素
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

3.扩容方法

```
final Node[] resize() {
    Node[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
// 新的容量是旧的容量的两倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
```

```
// 扩容门限也相应的提升为之前的两倍
            newThr = oldThr << 1; // double threshold
        }
        else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;
        else {               // zero initial threshold signifies using defaults
//首次扩容，容量初始化为16，门限初始化为16*0.75
            newCap = DEFAULT_INITIAL_CAPACITY;
            newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
        }
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor;
            newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                      (int)ft : Integer.MAX_VALUE);
        }
        threshold = newThr;
//开始向新的数组复制数据
        @SuppressWarnings({"rawtypes","unchecked"})
            Node[] newTab = (Node[])new Node[newCap];
        table = newTab;
        if (oldTab != null) {
            for (int j = 0; j < oldCap; ++j) {
                Node e;
                if ((e = oldTab[j]) != null) {
                    oldTab[j] = null;
                    if (e.next == null)
//单节点直接在新数组的对应的位置放上数据
                        newTab[e.hash & (newCap - 1)] = e;
                    else if (e instanceof TreeNode)
//树形链表使用树形方法拆分链表
                        ((TreeNode)e).split(this, newTab, j, oldCap);
                    else { // preserve order
//链表结构则将链表拆分为两条，一条在当前位置，一条在当前位置+之前容量的位置
                        Node loHead = null, loTail = null;
                        Node hiHead = null, hiTail = null;
                        Node next;
                        do {
                            next = e.next;
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        if (loTail != null) {
```

```
                    loTail.next = null;
//高位为0，放在原位置
                    newTab[j] = loHead;
                }
                if (hiTail != null) {
                    hiTail.next = null;
//高位为1，放在原位置偏移oldCap的位置
                    newTab[j + oldCap] = hiHead;
                }
            }
        }
    }
    return newTab;
}
```

详细解释一下hashmap获取key存放位置的方法，和扩容后拆分链表的方法