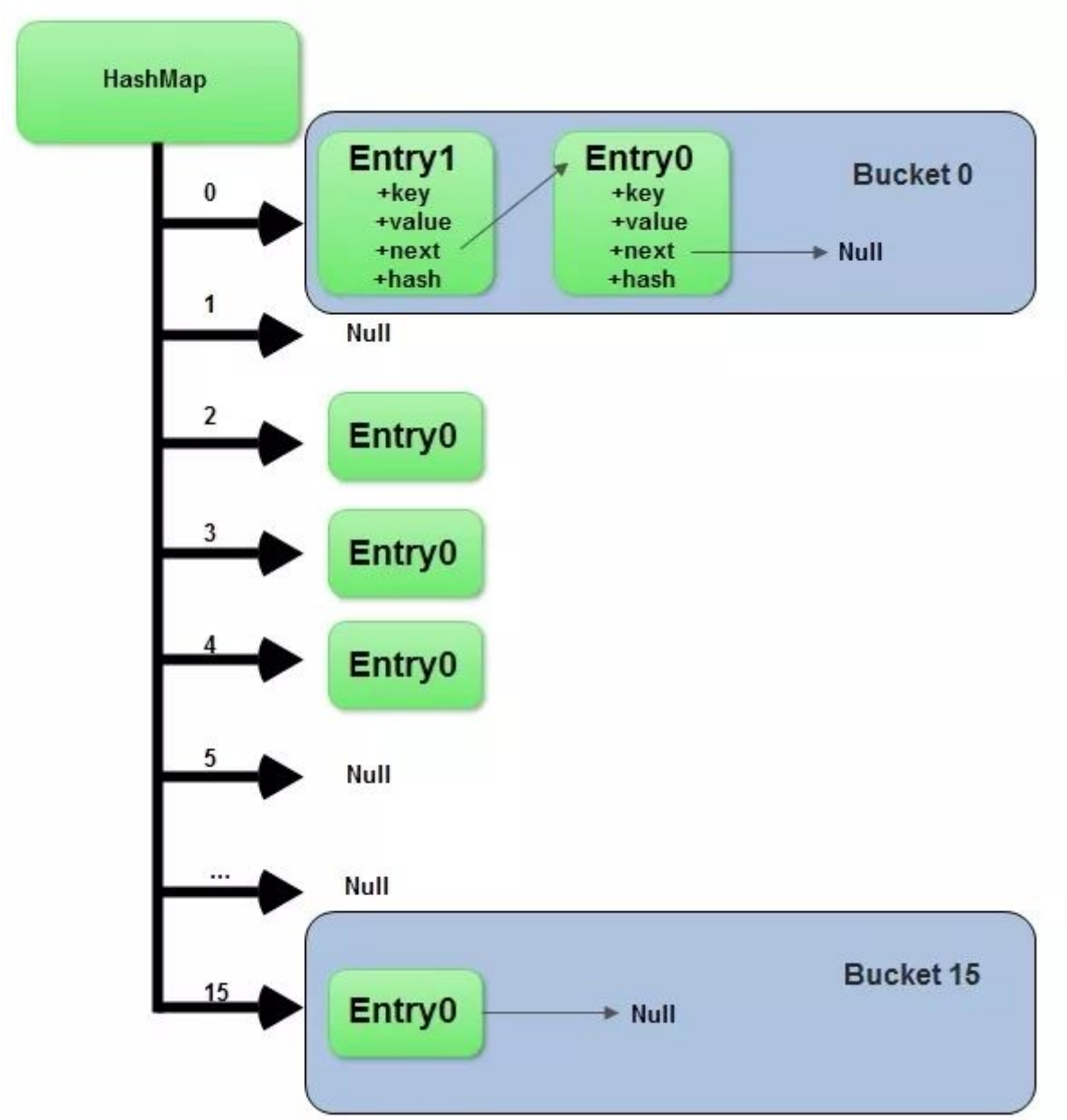
HashMap

HashMap的数据结构



> 0、1是该元素数据索引也是元素的hashCode

HashMap的核心成员

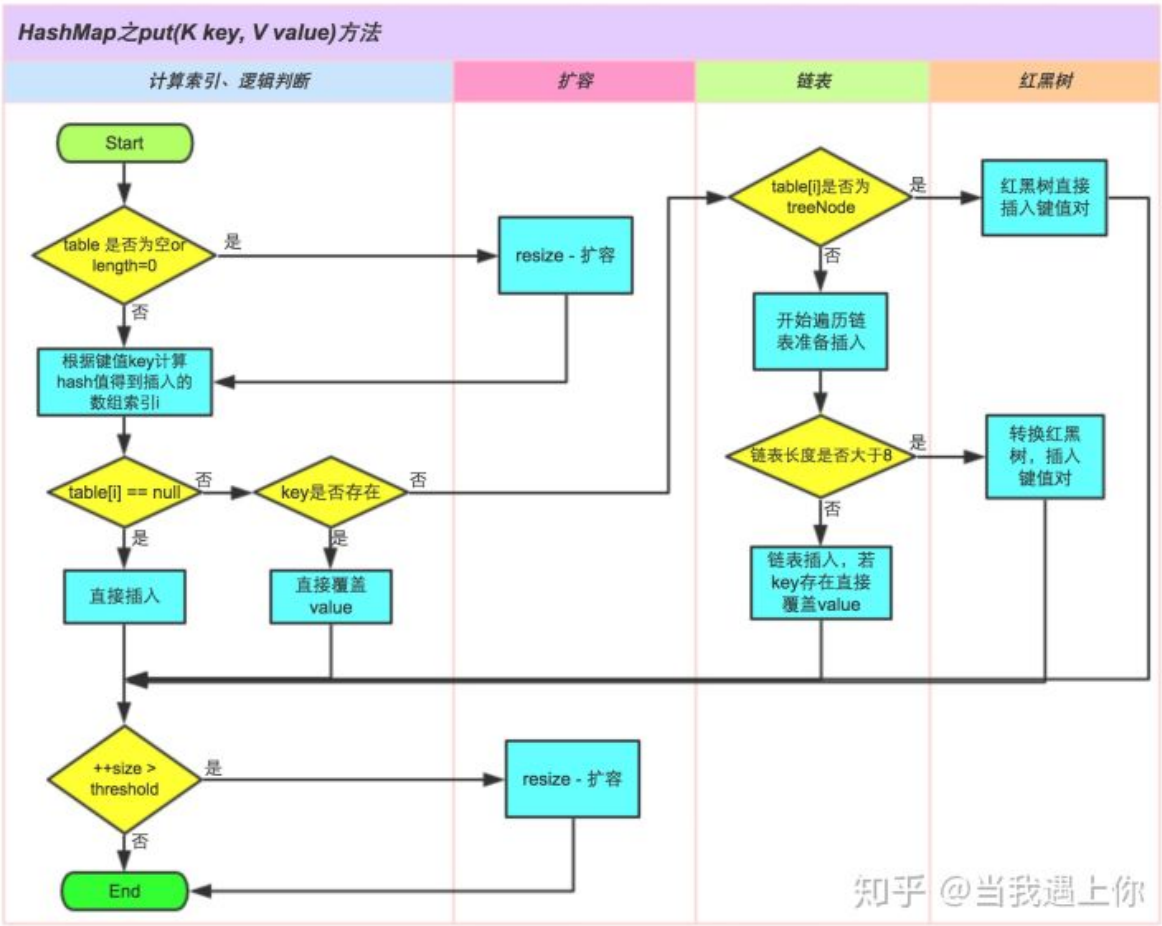
```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {

    // 默认容量，默认为16，必须是2的幂
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;

    // 最大容量，值是2^30
    static final int MAXIMUM_CAPACITY = 1 << 30

    // 装载因子，默认的装载因子是0.75
    static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

```
    // 解决冲突的数据结构由链表转换成树的阈值，默认为8
    static final int TREEIFY_THRESHOLD = 8;

    // 解决冲突的数据结构由树转换成链表的阈值，默认为6
    static final int UNTREEIFY_THRESHOLD = 6;

    /* 当桶中的bin被树化时最小的hash表容量。
     *  如果没有达到这个阈值，即hash表容量小于MIN_TREEIFY_CAPACITY，当桶中bin的数量太多时会执行resize扩容操作。
     *  这个MIN_TREEIFY_CAPACITY的值至少是TREEIFY_THRESHOLD的4倍。
     */
    static final int MIN_TREEIFY_CAPACITY = 64;

    static class Node<K,V> implements Map.Entry<K,V> {
        //...
    }
    // 存储数据的数组
    transient Node<K,V>[] table;

    // 遍历的容器
    transient Set<Map.Entry<K,V>> entrySet;

    // Map中KEY-VALUE的数量
    transient int size;

    /**
     * 结构性变更的次数。
     * 结构性变更是指map的元素数量的变化，比如rehash操作。
     * 用于HashMap快速失败操作，比如在遍历时发生了结构性变更，就会抛出ConcurrentModificationException。
     */
    transient int modCount;

    // 下次resize的操作的size值。
    int threshold;

    // 负载因子，resize后容量的大小会增加现有size * loadFactor
    final float loadFactor;
}
```

hash计算，确定数组索引位置

```
static final int hash(Object key) {   //jdk1.8
     int h;
     // h = key.hashCode() 为第一步 取hashCode值
     // h ^ (h >>> 16)  为第二步 高位参与运算
     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

// 获取元素
tab[i = (n - 1) & hash]
```

Hashmap中put()过程



put()源码如下：

```
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;
        // 判断数组是否为空，长度是否为0，是则进行扩容数组初始化
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        // 通过hash算法找到数组下标得到数组元素，为空则新建
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            // 找到数组元素，hash相等同时key相等，则直接覆盖
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            // 该数组元素在链表长度>8后形成红黑树结构的对象,p为树结构已存在的对象
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
                // 该数组元素hash相等，key不等，同时链表长度<8.进行遍历寻找元素，有就覆盖
无则新建
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        // 新建链表中数据元素，尾插法
                        p.next = newNode(hash, key, value, null);
```

```
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        // 链表长度>=8 结构转为 红黑树
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        // 新值覆盖旧值
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            // onlyIfAbsent默认false
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    // 判断是否需要扩容
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

基本过程如下:

1. 检查数组是否为空，执行resize()扩充；在实例化HashMap时，并不会进行初始化数组。默认长度是16
2. 通过hash值计算数组索引，获取该索引位的首节点。
3. 如果首节点为null（没发生碰撞），则创建新的数组元素，直接添加节点到该索引位(bucket)。
4. 如果首节点不为null（发生碰撞），那么有3种情况
   ① key和首节点的key相同，覆盖old value（保证key的唯一性）；否则执行②或③
   ② 如果首节点是红黑树节点（TreeNode），将键值对添加到红黑树。
   ③ 如果首节点是链表，进行遍历寻找元素，有就覆盖无则新建，将键值对添加到链表。添加之后会判断链表长度是否到达TREEIFY_THRESHOLD - 1（8）这个阈值，“尝试”将链表转换成红黑树。
5. 最后判断当前元素个数是否大于threshold，扩充数组。

HashMap中get()过程

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

    final Node<K,V> getNode(int hash, Object key) {
        Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            // 永远检查第一个node
            if (first.hash == hash && // always check first node
```

```
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;
            if ((e = first.next) != null) {
                if (first instanceof TreeNode)  // 树查找
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                do {
                    if (e.hash == hash &&   // 遍历链表
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }
```

在Hashmap1.8中，无论是存元素还是取元素，都是优先判断bucket上第一个元素是否匹配，而在1.7中则是直接遍历查找。

基本过程如下:

1. 根据key计算hash;
2. 检查数组是否为空，为空返回null;
3. 根据hash计算bucket位置，如果bucket第一个元素是目标元素，直接返回。否则执行4;
4. 如果bucket上元素大于1并且是树结构，则执行树查找。否则执行5;
5. 如果是链表结构，则遍历寻找目标

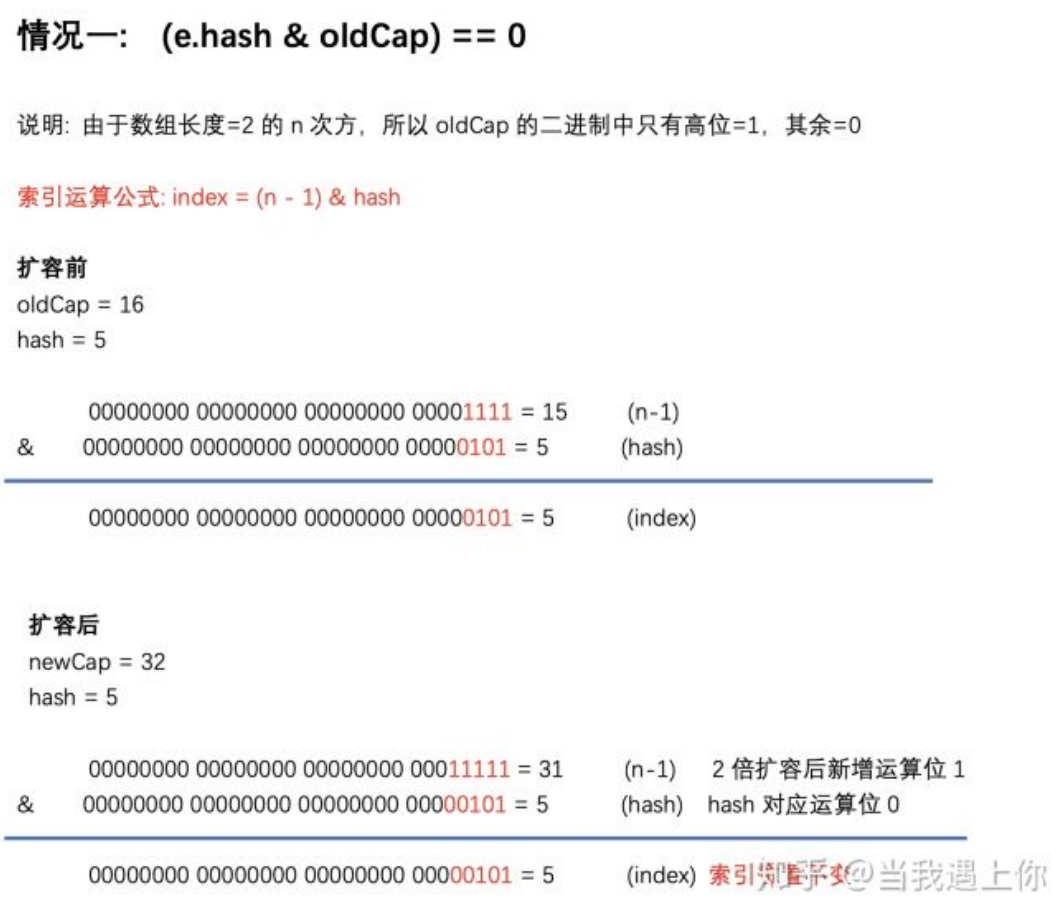
Hashmap中resize()过程

```
final Node<K, V>[] resize() {
        Node<K, V>[] oldTab = table;
        int oldCap = (oldTab == null) ? 0 : oldTab.length;
        int oldThr = threshold;
        int newCap, newThr = 0;
        if (oldCap > 0) {
            // 如果已达到最大容量不在扩容
            if (oldCap >= MAXIMUM_CAPACITY) {
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }
            // 通过位运算扩容到原来的两倍
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                    oldCap >= DEFAULT_INITIAL_CAPACITY)
                newThr = oldThr << 1; // double threshold
        } else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;
        else {               // zero initial threshold signifies using defaults
            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);
        }
        // 新的扩容临界值
        threshold = newThr;
```

```
        @SuppressWarnings({"rawtypes", "unchecked"})
        Node<K, V>[] newTab = (Node<K, V>[]) new Node[newCap];
        table = newTab;
        //如果原来的table等于null，直接返回
        if (oldTab != null) {
            //遍历原来的table，bucket
            for (int j = 0; j < oldCap; ++j) {
                HashMap.Node<K, V> e;//bucket
                if ((e = oldTab[j]) != null) {//如果当前桶位上有元素，不为null
                    oldTab[j] = null;//当前位置置空
                    if (e.next == null)//如果只有一个元素e，e.next为null，直接安排e到
新表新家
                        newTab[e.hash & (newCap - 1)] = e;
                    else if (e instanceof HashMap.TreeNode)//判断e是不是树形节点，也
就是超过8个元素
                        ((HashMap.TreeNode<K, V>) e).split(this, newTab, j,
oldCap);//
                    else { // preserve order
                        HashMap.Node<K, V> loHead = null, loTail = null;//低位的头
结点，尾结点
                        HashMap.Node<K, V> hiHead = null, hiTail = null;//高危的头
结点，尾结点
                        HashMap.Node<K, V> next;//下个节点
                        do {//在这个循环里，依次处理该桶上链表，分裂成高位链和低位链
                            next = e.next;//后一个节点元素
                            //如果元素的hash值，与 原来表的容量 等于0,实际上是把原来在一
个桶位的元素分流
                            //例如e.hash & 10000，值会有0和1两种，等于0的，还是相对于
原表的索引位置
                            //等于1,把他向高位调整
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)//只有第一次进来的时候，loTail为
null
                                    loHead = e;//设置头结点位e
                                else
                                    loTail.next = e;//尾插
                                loTail = e;//低位链指针下移
                            } else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        //while循环完之后，大概会形成两个链表【高位链hiHead--hiTail，低
位链loHead--loTail】,最极端的情况是只有高位链或
                        //只有低位链。拿着这两个链表，插入到对应桶位，入驻新家。
                        //判断低位链中是否有元素
                        if (loTail != null) {
                            loTail.next = null;//干掉低位链尾部的next，因为e的下一个
结点很可能被分到高位，所以我们要干掉这个叛徒
                            newTab[j] = loHead;
                        }
                        //判断高位链中是否有元素
                        if (hiTail != null) {
                            hiTail.next = null;
                            newTab[j + oldCap] = hiHead;
```

```
                        }
                    }
                }
            }
        }
        return newTab;
    }
```

1.8版本中扩容分2种情况，下边用图例来解释。



@当我遇上你

情况二： (e.hash & oldCap) != 0

说明：由于数组长度=2 的 n 次方，所以 oldCap 的二进制中只有高位=1，其余=0

索引运算公式: index = (n - 1) & hash

扩容前

oldCap = 16

hash = 21

```
      00000000 00000000 00000000 00001111 = 15      (n-1)
&     00000000 00000000 00000000 00010101 = 21      (hash)
-----------------------------------------------------------
      00000000 00000000 00000000 00000101 = 5       (index)
```

扩容后

newCap = 32

hash = 21

```
      00000000 00000000 00000000 00011111 = 31      (n-1)    2 倍扩容后新增运算位 1
&     00000000 00000000 00000000 00010101 = 21      (hash)   hash 对应运算位 1
-----------------------------------------------------------
      00000000 00000000 00000000 00010101 = 5 + 16  (index)  索引位置=index+oldCap
```

知乎 @当我遇上你

红黑树

红黑树比较传统的定义是需要满足以下五个特征：
（1）每个节点或者是黑色，或者是红色。
（2）根节点是黑色。
（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]
（4）如果一个节点是红色的，则它的子节点必须是黑色的。
（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。
其特点在于给数的每一个节点加上了颜色属性，在插入的过程中通过颜色变换和节点旋转调平衡。其实博主不是很喜欢上面的定义，还有一种视角就是将它与二三树比较。

红黑树相比avl树，在检索的时候效率其实差不多，都是通过平衡来二分查找。但对于插入删除等操作效率提高很多。红黑树不像avl树一样追求绝对的平衡，他允许局部很少的不完全平衡，这样对于效率影响不大，但省去了很多没有必要的调平衡操作，avl树调平衡有时候代价较大，所以效率不如红黑树，在现在很多地方都是底层都是红黑树的天下啦~