链滴

HashMap 在 Jdk1.7 和 1.8 中的实现

作者：wubo8196
原文链接：https://ld246.com/article/1587568495745
来源网站：链滴

一、初窥HashMap

HashMap是应用更广泛的哈希表实现，而且大部分情况下，都能在常数时间性能的情况下进行put和et操作。要掌握HashMap，主要从如下几点来把握：

- jdk1.7中底层是由 **数组（也有叫做“位桶”的）+链表**实现；jdk1.8中底层是由**数组+链表/红黑树**现
- 可以存储null键和null值，线程不安全。在HashMap中，null可以作为键，这样的键只有一个，但以有一个或多个键所对应的值为null。当get()方法返回null值时，即可以表示HashMap中没有该key也可以表示该key所对应的value为null。因此，在HashMap中不能由get()方法来判断HashMap中是存在某个key，应该用containsKey()方法来判断。而在Hashtable中，无论是key还是value都不能为nll。
- 初始size为 **16**，扩容：newsize = oldsize*2，size一定为2的n次幂
- 扩容针对整个Map，每次扩容时，原来数组中的元素依次重新计算存放位置，并重新插入
- 插入元素后才判断该不该扩容，有可能无效扩容（插入后如果扩容，如果没有再次插入，就会产生效扩容）
- 当Map中元素总数超过Entry数组的75%，触发扩容操作，为了减少链表长度，元素分配更均匀
- 1.7中是 **先扩容后插入**新值的，1.8中是**先插值再扩容**

为什么说HashMap是线程不安全的？在接近临界点时，若此时两个或者多个线程进行put操作，都会行resize（扩容）和reHash（为key重新计算所在位置），而reHash在并发的情况下可能会形成链表环总结来说就是在多线程环境下，使用HashMap进行put操作会引起死循环，导致CPU利用率接近100，所以在并发情况下不能使用HashMap。为什么在并发执行put操作会引起死循环？是因为多线程会致HashMap的Entry链表形成环形数据结构，一旦形成环形数据结构，Entry的next节点永远不为空就会产生死循环获取Entry。jdk1.7的情况下，并发扩容时容易形成链表环，此情况在1.8时就好太多多了。因为在1.8中当链表长度大于阈值（默认长度为8）时，链表会被改成树形（红黑树）结构。

二、jdk1.7中HashMap的实现

HashMap底层维护的是数组+链表，我们可以通过一小段源码来看看：

```
/**
 * The default initial capacity - MUST be a power of two.
 *  即 默认初始大小，值为16
 */
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

/**
 * The maximum capacity, used if a higher value is implicitly specified
 * by either of the constructors with arguments.
 * MUST be a power of two <= 1<<30.
 *  即 最大容量，必须为2^30
 */
static final int MAXIMUM_CAPACITY = 1 << 30;

/**
 * The load factor used when none specified in constructor.
 * 负载因子为0.75
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

原文链接：HashMap 在 Jdk1.7 和 1.8 中的实现

```
/**
 * The bin count threshold for using a tree rather than list for a
 * bin.  Bins are converted to trees when adding an element to a
 * bin with at least this many nodes. The value must be greater
 * than 2 and should be at least 8 to mesh with assumptions in
 * tree removal about conversion back to plain bins upon
 * shrinkage.
 * 大致意思就是说hash冲突默认采用单链表存储，当单链表节点个数大于8时，会转化为红黑树存储
 */
static final int TREEIFY_THRESHOLD = 8;

/**
 * The bin count threshold for untreeifying a (split) bin during a
 * resize operation. Should be less than TREEIFY_THRESHOLD, and at
 * most 6 to mesh with shrinkage detection under removal.
 * hash冲突默认采用单链表存储，当单链表节点个数大于8时，会转化
   为红黑树存储。
* 当红黑树中节点少于6时，则转化为单链表存储
 */
static final int UNTREEIFY_THRESHOLD = 6;

/**
 * The smallest table capacity for which bins may be treeified.
 * (Otherwise the table is resized if too many nodes in a bin.)
 * Should be at least 4 * TREEIFY_THRESHOLD to avoid conflicts
 * between resizing and treeification thresholds.
 * hash冲突默认采用单链表存储，当单链表节点个数大于8时，会转化为红黑树存储。
 * 但是有一个前提：要求数组长度大于64，否则不会进行转化
 */
static final int MIN_TREEIFY_CAPACITY = 64;
```

通过以上代码可以看出初始容量（16）、负载因子以及对数组的说明。数组中的每一个元素其实就是**ntry<K,V>[] table**，Map中的key和value就是以Entry的形式存储的。Entry包含四个属性：key、value、hash值和用于单向链表的next。关于Entry<K,V>的具体定义参看如下源码：

```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;
    int hash;

    Entry(int h, K k, V v, Entry<K,V> n) {
        value = v;
        next = n;
        key = k;
        hash = h;
    }

    public final K getKey() {
        return key;
    }

    public final V getValue() {
```

```
        return value;
    }

    public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

    public final boolean equals(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry e = (Map.Entry)o;
        Object k1 = getKey();
        Object k2 = e.getKey();
        if (k1 == k2 || (k1 != null && k1.equals(k2))) {
            Object v1 = getValue();
            Object v2 = e.getValue();
            if (v1 == v2 || (v1 != null && v1.equals(v2)))
                return true;
        }
        return false;
    }

    public final int hashCode() {
        return Objects.hashCode(getKey()) ^ Objects.hashCode(getValue());
    }

    public final String toString() {
        return getKey() + "=" + getValue();
    }

    /**
     * This method is invoked whenever the value in an entry is
     * overwritten by an invocation of put(k,v) for a key k that's already
     * in the HashMap.
     */
    void recordAccess(HashMap<K,V> m) {
    }

    /**
     * This method is invoked whenever the entry is
     * removed from the table.
     */
    void recordRemoval(HashMap<K,V> m) {
    }
}
```

HashMap的初始值要考虑负载因子:

● 哈希冲突：若干Key的哈希值按数组大小取模后，如果落在同一个数组下标上，将组成一条Entry链对Key的查找需要遍历Entry链上的每个元素执行equals()比较。

● 加载因子：为了降低哈希冲突的概率，默认当HashMap中的键值对达到数组大小的75%时，即会发扩容。因此，如果预估容量是100，即需要设定100/0.75 = 134的数组大小。

● 空间换时间：如果希望加快Key查找的时间，还可以进一步降低加载因子，加大初始大小，以降低希冲突的概率。

HashMap和Hashtable都是用hash算法来决定其元素的存储，因此HashMap和Hashtable的hash表含如下属性：

● 容量（capacity）：hash表中桶的数量

● 初始化容量（initial capacity）：创建hash表时桶的数量，HashMap允许在构造器中指定初始化量

● 尺寸（size）：当前hash表中记录的数量

● 负载因子（load factor）：负载因子等于“size/capacity”。负载因子为0，表示空的hash表，0.表示半满的散列表，依此类推。轻负载的散列表具有冲突少、适宜插入与查询的特点（但是使用Iterator迭代元素时比较慢）

除此之外，hash表里还有一个“负载极限”，“负载极限”是一个0～1的数值，“负载极限”决定了ash表的最大填满程度。当hash表中的负载因子达到指定的“负载极限”时，hash表会自动成倍地增容量（桶的数量），并将原有的对象重新分配，放入新的桶内，这称为rehashing。

HashMap和Hashtable的构造器允许指定一个负载极限，HashMap和Hashtable默认的“负载极限为0.75，这表明当该hash表的3/4已经被填满时，hash表会发生rehashing。

“负载极限”的默认值（0.75）是时间和空间成本上的一种折中：

● 较高的“负载极限”可以降低hash表所占用的内存空间，但会增加查询数据的时间开销，而查询是频繁的操作（HashMap的get()与put()方法都要用到查询）

● 较低的“负载极限”会提高查询数据的性能，但会增加hash表所占用的内存开销

程序猿可以根据实际情况来调整“负载极限”值。

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， 该位置就是此象准备往数组中存放的位置。 该计算过程参看如下代码：

```
transient int hashSeed = 0;
final int hash(Object k) {
    int h = hashSeed;
    if (0 != h && k instanceof String) {
        return sun.misc.Hashing.stringHash32((String) k);
    }

    h ^= k.hashCode();

    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}

/**
 * Returns index for hash code h.
 */
static int indexFor(int h, int length) {
    // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";
    return h & (length-1);
}
```

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。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(**拉链法**)。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，所以新值存放在组中，旧值在新值的链表上）。即将新值作为此链表的头节点，为什么要这样操作？据说后插入的Enty被查找的可能性更大（因为get查询的时候会遍历整个链表），此处有待考究，如果有哪位大神知道请留言告知。有一种说法就是链表查找复杂度高，可插入和删除性能高，如果将新值插在末尾，就需先经过一轮遍历，这个时间复杂度高，开销大，如果是插在头结点，省去了遍历的开销，还发挥了链插入性能高的优势。

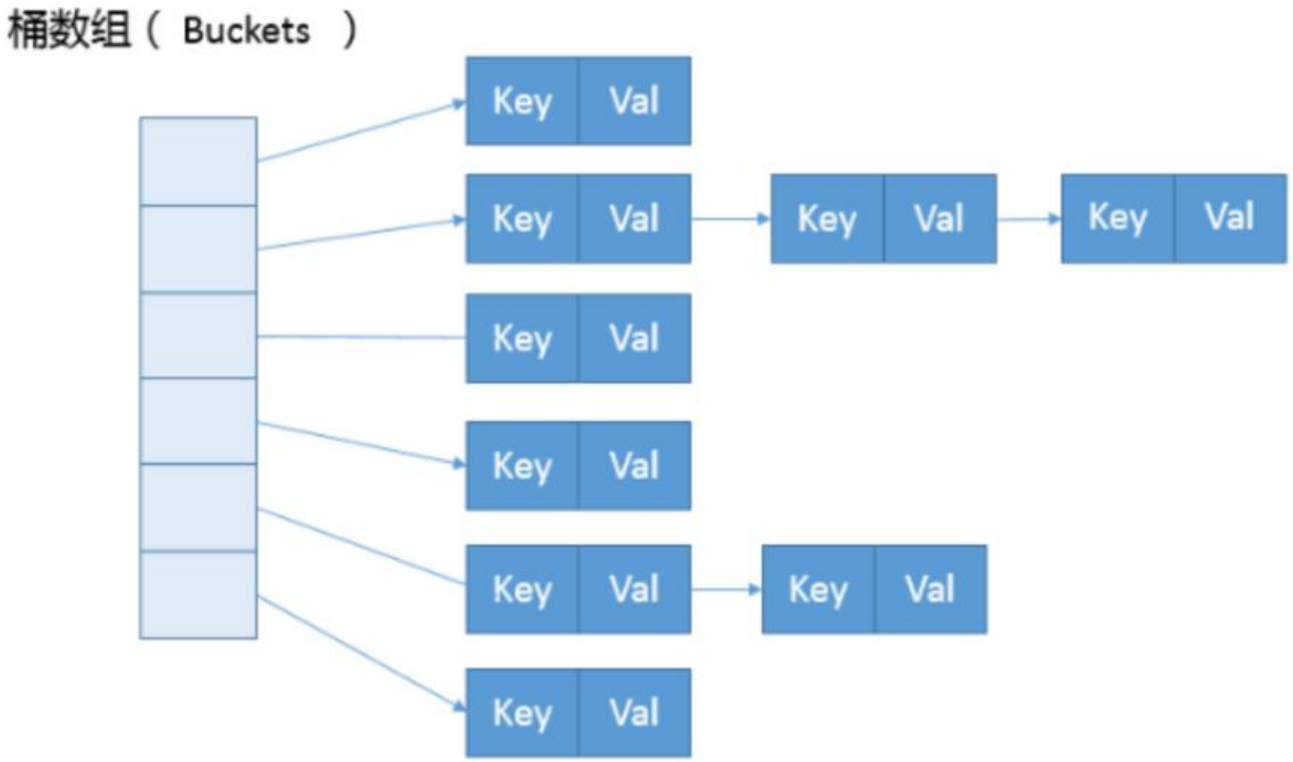
如果该位置没有对象存在，就将此对象直接放进数组当中；如果该位置已经有对象存在了，则顺着此在的对象的链开始寻找(为了判断是否值相同，map不允许<key,value>键值对重复)， 如果此链上对象的话，再去使用 equals方法进行比较，如果对此链上的每个对象的 equals 方法比较都为 false 则将该对象放到数组当中，然后将数组中该位置以前存在的那个对象链接到此对象的后面。

添加节点到链表中：找到数组下标后，会先进行key判重，如果没有重复，就准备将新值放入到链表表头。

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    // 如果当前 HashMap 大小已经达到了阈值，并且新值要插入的数组位置已经有元素了，那么要容
    if ((size >= threshold) && (null != table[bucketIndex])) {
        // 扩容
        resize(2 * table.length);
        // 扩容以后，重新计算 hash 值
        hash = (null != key) ? hash(key) : 0;
        // 重新计算扩容后的新的下标
        bucketIndex = indexFor(hash, table.length);
    }
    // 往下看
    createEntry(hash, key, value, bucketIndex);
}
// 这个很简单，其实就是将新值放到链表的表头，然后 size++
void createEntry(int hash, K key, V value, int bucketIndex) {
    Entry<K,V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<>(hash, key, value, e);
    size++;
}
```

这个方法的主要逻辑就是先判断是否需要扩容，需要带的话先扩容，然后再将这个新的数据插入到扩后的数组的相应位置处的链表的表头。

扩容就是用一个新的大数组替换原来的小数组，并将原来数组中的值迁移到新的数组中。由于是双倍容，迁移过程中，会将原来table[i]中的链表的所有节点，分拆到新的数组的newTable[i]和newTable[+oldLength]位置上。如原来数组长度是16，那么扩容后，原来table[0]处的链表中的所有元素会被配到新数组中newTable[0]和newTable[16]这两个位置。扩容期间，由于会新建一个新的空数组，并用旧的项填充到这个新的数组中去。所以，在这个填充的过程中，如果有线程获取值，很可能会取到ull 值，而不是我们所希望的、原来添加的值。



图中，左边部分即代表哈希表，也称为哈希数组（默认数组大小是16，每对key-value键值对其实是在map的内部类entry里的），**数组的每个元素都是一个单链表的头节点**，跟着的蓝色链表是用来解冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中。

前面说过HashMap的key是允许为null的，当出现这种情况时，会放到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>=threshold（threshold等于“容量*负载因子”）时，会发生扩容。

```
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);
}
```

特别提示：jdk1.7中resize，只有当 size>=threshold并且 table中的那个槽中已经有Entry时，才会生resize。即有可能虽然size>=threshold，但是必须等到相应的槽至少有一个Entry时，才会扩容,可通过上面的代码看到每次resize都会扩大一倍容量（2 * table.length）。

三、jdk1.8中HashMap的实现

在jdk1.8中HashMap的内部结构可以看作是数组(Node<K,V>[] table)和链表的复合结构，数组被分一个个桶（bucket），通过哈希值决定了键值对在这个数组中的寻址（哈希值相同的键值对，则以链形式存储。有一点需要注意，如果链表大小超过阈值（TREEIFY_THRESHOLD,8），图中的链表就会改造为树形（红黑树）结构。

transient Node<K,V>[] table;

Entry的名字变成了Node，原因是和红黑树的实现TreeNode相关联。**1.8与1.7最大的不同就是利用红黑树，即由数组+链表（或红黑树）组成。**

在分析jdk1.7中HashMap的hash冲突时，不知大家是否有个疑问就是万一发生碰撞的节点非常多怎办？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JDK1.8中得到了解决在最坏的情况下，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。

dk1.7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jdk1.8中采用的是桶+链表/红黑树的方式，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，这个链就将转换成红黑树。

jdk1.8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，会被调整成一颗红树（上图中null节点没画）。这就是jdk1.7与jdk1.8中HashMap实现的最大区别。

HashMap根据链地址法（拉链法）来解决冲突，**在jdk1.8中，如果链表长度大于8且节点数组长度大64的时候，就把链表下所有的节点转为红黑树。**

通过分析put方法的源码，可以让这种区别更直观：

```
static final int TREEIFY_THRESHOLD = 8;

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，放在这个位上即可
        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。
```

```
            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;
        }
```

以上代码中的特别之处如下：

```
if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
        treeifyBin(tab, hash);
```

treeifyBin()就是将链表转换成红黑树。

树化操作的过程有点复杂，可以结合源码来看看。将原本的单链表转化为双向链表，再遍历这个双向表转化为红黑树。

```
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    //树形化还有一个要求就是数组长度必须大于等于64，否则继续采用扩容策略
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();
```

```
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        TreeNode<K,V> hd = null, tl = null;//hd指向首节点，tl指向尾节点
        do {
            TreeNode<K,V> p = replacementTreeNode(e, null);//将链表节点转化为红黑树节点
            if (tl == null) // 如果尾节点为空，说明还没有首节点
                hd = p;  // 当前节点作为首节点
            else { // 尾节点不为空，构造一个双向链表结构，将当前节点追加到双向链表的末尾
                p.prev = tl; // 当前树节点的前一个节点指向尾节点
                tl.next = p; // 尾节点的后一个节点指向当前节点
            }
            tl = p; // 把当前节点设为尾节点
        } while ((e = e.next) != null); // 继续遍历单链表
        //将原本的单链表转化为一个节点类型为TreeNode的双向链表
        if ((tab[index] = hd) != null) // 把转换后的双向链表，替换数组原来位置上的单向链表
            hd.treeify(tab); // 将当前双向链表树形化
    }
}
```

大家要特别注意一点，树化有个要求就是数组长度必须大于等于MIN_TREEIFY_CAPACITY（64），则继续采用扩容策略。

总的来说，HashMap默认采用数组+单链表方式存储元素，当元素出现哈希冲突时，会存储到该位置单链表中。但是单链表不会一直增加元素，当元素个数超过8个时，会尝试将单链表转化为红黑树存。但是在转化前，会再判断一次当前数组的长度，只有数组长度大于64才处理。否则，进行扩容操作。

将双向链表转化为红黑树的实现：

```
 final void treeify(Node<K,V>[] tab) {
    TreeNode<K,V> root = null;  // 定义红黑树的根节点
    for (TreeNode<K,V> x = this, next; x != null; x = next) { // 从TreeNode双向链表的头节点开逐个遍历
        next = (TreeNode<K,V>)x.next; // 头节点的后继节点
        x.left = x.right = null;
        if (root == null) {
            x.parent = null;
            x.red = false;
            root = x; // 头节点作为红黑树的根，设置为黑色
        }
        else { // 红黑树存在根节点
            K k = x.key;
            int h = x.hash;
            Class<?> kc = null;
            for (TreeNode<K,V> p = root;;) { // 从根开始遍历整个红黑树
                int dir, ph;
                K pk = p.key;
                if ((ph = p.hash) > h) // 当前红黑树节点p的hash值大于双向链表节点x的哈希值
                    dir = -1;
                else if (ph < h) // 当前红黑树节点的hash值小于双向链表节点x的哈希值
                    dir = 1;
                else if ((kc == null &&
                          (kc = comparableClassFor(k)) == null) ||
                         (dir = compareComparables(kc, k, pk)) == 0) // 当前红黑树节点的hash值等于双链表节点x的哈希值，则如果key值采用比较器一致则比较key值
                    dir = tieBreakOrder(k, pk); //如果key值也一致则比较className和identityHashCode
```

```
                TreeNode<K,V> xp = p;
                if ((p = (dir <= 0) ? p.left : p.right) == null) { // 如果当前红黑树节点p是叶子节点，那双向链表节点x就找到了插入的位置
                    x.parent = xp;
                    if (dir <= 0) //根据dir的值，插入到p的左孩子或者右孩子
                        xp.left = x;
                    else
                        xp.right = x;
                    root = balanceInsertion(root, x); //红黑树中插入元素，需要进行平衡调整(过程和TreeMap调整逻辑一模一样)
                    break;
                }
            }
        }
    }
    //将TreeNode双向链表转化为红黑树结构之后，由于红黑树是基于根节点进行查找，所以必须将黑树的根节点作为数组当前位置的元素
    moveRootToFront(tab, root);
}
```

然后将红黑树的根节点移动端数组的索引所在位置上：

```
static <K,V> void moveRootToFront(Node<K,V>[] tab, TreeNode<K,V> root) {
    int n;
    if (root != null && tab != null && (n = tab.length) > 0) {
        int index = (n - 1) & root.hash; //找到红黑树根节点在数组中的位置
        TreeNode<K,V> first = (TreeNode<K,V>)tab[index]; //获取当前数组中该位置的元素
        if (root != first) { //红黑树根节点不是数组当前位置的元素
            Node<K,V> rn;
            tab[index] = root;
            TreeNode<K,V> rp = root.prev;
            if ((rn = root.next) != null) //将红黑树根节点前后节点相连
                ((TreeNode<K,V>)rn).prev = rp;
            if (rp != null)
                rp.next = rn;
            if (first != null) //将数组当前位置的元素，作为红黑树根节点的后继节点
                first.prev = root;
            root.next = first;
            root.prev = null;
        }
        assert checkInvariants(root);
    }
}
```

putVal方法处理的逻辑比较多，包括初始化、扩容、树化，近乎在这个方法中都能体现，针对源码简讲解下几个关键点：

- 如果Node<K,V>[] table是null，resize方法会负责初始化，即如下代码：

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

- resize方法兼顾两个职责，创建初始存储表格，或者在容量不满足需求的时候，进行扩容（resize
。

在放置新的键值对的过程中，如果发生下面条件，就会发生扩容。

```
if (++size > threshold)
    resize();
```

- 具体键值对在哈希表中的位置（数组index）取决于下面的位运算：

```
i = (n - 1) & hash
```

仔细观察哈希值的源头，会发现它并不是key本身的hashCode，而是来自于HashMap内部的另一个hash方法。为什么这里需要将高位数据移位到低位进行异或运算呢？这是因为有些数据计算出的哈希值异主要在高位，而HashMap里的哈希寻址是忽略容量以上的高位的，那么这种处理就可以有效避免似情况下的哈希碰撞。

在jdk1.8中取消了indefFor()方法，直接用(tab.length-1)&hash，所以看到这个，代表的就是数组的角标。

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

为什么HashMap为什么要树化?

之前在极客时间的专栏里看到过一个解释。本质上这是个安全问题。因为在元素放置过程中，如果一对象哈希冲突，都被放置到同一个桶里，则会形成一个链表，我们知道链表查询是线性的，会严重影存取的性能。而在现实世界，构造哈希冲突的数据并不是非常复杂的事情，恶意代码就可以利用这些据大量与服务器端交互，导致服务器端CPU大量占用，这就构成了哈希碰撞拒绝服务攻击，国内一线联网公司就发生过类似攻击事件。

用哈希碰撞发起拒绝服务攻击(DOS，Denial-Of-Service attack),常见的场景是攻击者可以事先构造量相同哈希值的数据，然后以JSON数据的形式发送给服务器，服务器端在将其构建成为Java对象过中，通常以Hashtable或HashMap等形式存储，哈希碰撞将导致哈希表发生严重退化，算法复杂度能上升一个数据级，进而耗费大量CPU资源。

为什么要将链表中转红黑树的阈值设为8?

我们可以这么来看，当链表长度大于或等于阈值（默认为 8）的时候，如果同时还满足容量大于或等于 MIN_TREEIFY_CAPACITY（默认为 64）的要求，就会把链表转换为红黑树。同样，后续如果由于删或者其他原因调整了大小，当红黑树的节点小于或等于 6 个以后，又会恢复为链表形态。每次遍历一链表，平均查找的时间复杂度是 O(n)，n 是链表的长度。红黑树有和链表不一样的查找性能，由于红树有自平衡的特点，可以防止不平衡情况的发生，所以可以始终将查找的时间复杂度控制在 O(log(n)。最初链表还不是很长，所以可能 O(n) 和 O(log(n)) 的区别不大，但是如果链表越来越长，那么这区别便会有所体现。所以为了提升查找性能，需要把链表转化为红黑树的形式。

还要注意很重要的一点，单个 TreeNode 需要占用的空间大约是普通 Node 的两倍，所以只有当包足够多的 Nodes 时才会转成 TreeNodes，而是否足够多就是由 TREEIFY_THRESHOLD 的值决定的而当桶中节点数由于移除或者 resize 变少后，又会变回普通的链表的形式，以便节省空间。

默认是链表长度达到 8 就转成红黑树，而当长度降到 6 就转换回去，这体现了时间和空间平衡的思想最开始使用链表的时候，空间占用是比较少的，而且由于链表短，所以查询时间也没有太大的问题。是当链表越来越长，需要用红黑树的形式来保证查询的效率。

在理想情况下，链表长度符合泊松分布，各个长度的命中概率依次递减，当长度为 8 的时候，是最理的值。

事实上，链表长度超过 8 就转为红黑树的设计，更多的是为了防止用户自己实现了不好的哈希算法时致链表过长，从而导致查询效率低，而此时转为红黑树更多的是一种保底策略，用来保证极端情况下

询的效率。

通常如果 hash 算法正常的话，那么链表的长度也不会很长，那么红黑树也不会带来明显的查询时间的优势，反而会增加空间负担。所以通常情况下，并没有必要转为红黑树，所以就选择了概率非常小小于千万分之一概率，也就是长度为 8 的概率，把长度 8 作为转化的默认阈值。

如果开发中发现 HashMap 内部出现了红黑树的结构，那可能是我们的哈希算法出了问题，所以需要用合适的hashCode方法，以便减少冲突。

四、分析Hashtable、HashMap、TreeMap的区别

● HashMap是继承自AbstractMap类，而HashTable是继承自Dictionary类。不过它们都同时实现map、Cloneable（可复制）、Serializable（可序列化）这三个接口。存储的内容是基于key-value键值对映射，不能有重复的key，而且一个key只能映射一个value。HashSet底层就是基于HashMap现的。

● Hashtable的key、value都不能为null；HashMap的key、value可以为null，不过只能有一个key null，但可以有多个null的value；TreeMap键、值都不能为null。

● Hashtable、HashMap具有 **无序**特性。TreeMap是利用红黑树实现的（树中的每个节点的值都大于或等于它的左子树中的所有节点的值，并且小于或等于它的右子树中的所有节点的值），实现了SrtMap接口，能够对保存的记录根据键进行排序。所以一般需求排序的情况下首选TreeMap，默认按的升序排序（深度优先搜索），也可以自定义实现Comparator接口实现排序方式。

一般情况下我们选用HashMap，因为HashMap的键值对在取出时是随机的，其依据键的hashCode键的equals方法存取数据，具有很快的访问速度，所以在Map中插入、删除及索引元素时其是效率最的实现。而TreeMap的键值对在取出时是排过序的，所以效率会低点。

TreeMap是基于红黑树的一种提供顺序访问的Map，与HashMap不同的是它的get、put、remove类操作都是o(log(n))的时间复杂度，具体顺序可以由指定的Comparator来决定，或者根据键的自然序来判断。

对HashMap做下总结:

HashMap基于哈希散列表实现，可以实现对数据的读写。**将键值对传递给put方法时，它调用键对的hashCode()方法来计算hashCode，然后找到相应的bucket位置（即数组）来储存值对象。当获对象时，通过键对象的equals()方法找到正确的键值对，然后返回值对象。**HashMap使用链表来解决ash冲突问题，当发生冲突了，对象将会储存在链表的头节点中。HashMap在每个链表节点中储存键对对象，当两个不同的键对象的hashCode相同时，它们会储存在同一个bucket位置的链表中，如果表大小超过阈值（TREEIFY_THRESHOLD,8），链表就会被改造为树形结构。

有个问题要特别声明下:

● HashMap在jdk1.7中采用 **表头插入法**，在扩容时会**改变**链表中元素原本的顺序，以至于在并发场下导致链表成环的问题。

● 在jdk1.8中采用的是 **尾部插入法**，在扩容时会保持链表元素原本的顺序，就不会出现链表成环的题了。

我们可以简单列下HashMap在1.7和1.8之间的变化：

● 1.7中采用数组+链表，1.8采用的是数组+链表/红黑树，即在1.7中链表长度超过一定长度后就改成黑树存储。

● 1.7扩容时需要重新计算哈希值和索引位置，1.8并不重新计算哈希值，巧妙地采用和扩容后容量进&操作来计算新的索引位置。

● 1.7是采用表头插入法插入链表，1.8采用的是尾部插入法。

● 在1.7中采用表头插入法，在扩容时会改变链表中元素原本的顺序，以至于在并发场景下导致链表成

的问题；在1.8中采用尾部插入法，在扩容时会保持链表元素原本的顺序，就不会出现链表成环的问了。

转自: https://yuanrengu.com/2020/ba184259.html