HashMap

> 基于 jdk11

构造函数

HashMap 支持传入一个初始容量以及扩容因子作为构造参数，有意思的是 **HashMap 并不会以我们传入的初始容量作为大小，而是会将第一个大于等于初始容量的 2 的幂作为 Map 的大小**，HashMap 的大小永远都是 2 的幂，这有许多好处：

1. 令 N 为 2 的幂，那么 $H \ % \ N \iff H \ & \ (N-1) $，位运算比求余运算快得多。
2. 令 N 为 2 的幂，当 $H \ % \ N=k$ 时，有 $ H \ % \ 2N = k \ 或 \ H \ % \ 2N = k + N$，也就是说当 Map 扩容时，元素扩容后的位置要么就在原下标，要么就在原下标+N的位置，两个不同下标的元素扩容时绝不会发生冲突，这十分方便。

扩容因子其实就是扩容的阈值，当Map实际元素个数达到 $设定容量 \times 扩容因子$ 时，将会触发扩容操作，默认的扩容因子是 0.75，这个值是官方推荐的，不至于过大导致扩容难以被触发，如果一直不扩容，那么 Map 中发生哈希冲突的可能性就会越大；也不至于过小导致扩容频繁发生。

插入时逻辑

HashMap 中，元素的哈希值是这样产生的：

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

Map 中计算哈希值时，让 key 的 hashCode 与其本身的高 16 位进行亦或得出，这样做的原因**是由于在计算索引采用的公式为 $hash & (N-1)$，当 N 较小时，例如 N = 8，此时与 N - 1 操作将仅仅采用 hash 的低三位作为下标，而没有利用到 hash 的高**

位，基于这个原因，HashMap 在设计 hash 时，让哈希的高位在低位上也能有所体现，这样做可以有效减少冲突。

在 jdk1.8 后，当链表长度超过阈值时，就会转换成红黑树，这个阈值在 HashMap 中是 8（第九个元素到来时）：

```
for (int binCount = 0; ; ++binCount) {
    // 遍历到尾部，没找到，新建节点插入
    if ((e = p.next) == null) {
        p.next = newNode(hash, key, value, null);
        // TREEIFY_THRESHOLD = 8，大于等于 8 个节点就树化
        if (binCount >= TREEIFY_THRESHOLD - 1)
            treeifyBin(tab, hash);
        break;
    }
    // 如果找到了就退出，直接更新
    if (e.hash == hash && ((k = e.key) == key || (key != null &&
key.equals(k))))
        break;
    p = e;
}
```

在树化的方法中，例如将原来的链表的 Node 转换成 TreeNode，有意思的是 HashMap 中 TreeNode 是继承至 Node 的，因此 next 指向依然是不变的，并且在 TreeNode 中，**还增加了一个 prev 字段，这意味着一颗红黑树中还存在一个双向链表。**

> 管不得 Map 这么耗内存。

```
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        // 获取tab[i]
        TreeNode<K,V> hd = null, tl = null;
        do {
            // 将每个节点转换成 TreeNode
            TreeNode<K,V> p = replacementTreeNode(e, null);
            if (tl == null)
                hd = p;
            else {
                p.prev = tl; // 链上前一个节点
                tl.next = p;
            }
            tl = p;
        } while ((e = e.next) != null);
        if ((tab[index] = hd) != null)
            hd.treeify(tab);    // 遍历元素插入红黑树
    }
}
```

注意到 tab.length < MIN_TREEIFY_CAPACITY 判断，当哈希表的大小小于 64 时，此时不会树化而是优先扩容，扩容也能有效缓解哈希冲突的情况。

因此树化的条件是：**链表元素大于 8 个，并且哈希表大小大于等于 64**

可以发现 HashMap 在插入链表时采用的是尾插法，而如果已经转换成树了，HashMap 也有对应的分支：

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

现在，就是红黑树插入的逻辑了。

红黑树的插入

回顾一下红黑树的知识点：

- 根节点一定是黑的
- 红色节点的孩子一定是黑的
- 从一个节点到任意一个叶子节点不同路径上的黑色节点数目都是一样的

如果采用自底向上的插入方法，我们首先要定位待插入的位置，然后向上平衡。

为了保证第三点要素，新插入的节点必须是红色的，然后考虑所有的情况：

- 父节点为空：此时自身为根节点，变为黑色插入。
- 父节点为黑色：自身为红色节点，不会破坏红黑树的性质，直接插入。
- 父节点为红色（这意味着祖父节点必定是黑色的）：
  - 叔叔节点为空或者为黑色：此时需要旋转，并且父节点和祖父节点都需要变色。
  - 叔叔节点为红色：将叔叔节点和父节点变为黑色，祖父节点变为红色，颜色翻转，由于祖父节点变为红色，因此祖父节点可能不平衡，对祖父节点继续向上循环。

瞅瞅 HashMap 中的源码，首先是查找，查找就是简单的二叉查找树的查找，一个问题是对于 Object 的 key，要如何去比较大小呢？在树化的逻辑中，任何元素都是不同的，也就是说一定具有可比性，HashMap 的做法是：

- 首先比较哈希值。
- 哈希值相等的话，如果类型实现了 Comparable 的接口话，调用 Comparable 比较。
- 如果还相等，比较它们的类名（字符串比较）。
- 如果还想等，则调用 `System.identityHashCode` 方法比较它们地址大小。这不可能相等，此方法类似于 C 语言的取址符，不同对象的地址一定是不同的！

一旦找到对应的位置后，则开始向上平衡：

```
static <K,V> TreeNode<K,V> balanceInsertion(TreeNode<K,V> root,
TreeNode<K,V> x) {
    // 待插入节点默认红色
    x.red = true;
    for (TreeNode<K,V> xp, xpp, xppl, xppr;;) {
        // 父节点空，自己作为根，直接返回
        if ((xp = x.parent) == null) {
            x.red = false;
            return x;
        }
        // 父节点黑色，直接返回
        else if (!xp.red)
            return root;
        xppl = xpp.left;
        // 那么父节点为红色
        // 父节点作为左节点，需要一次 L型旋转(右旋)
        if (xp == (xppl)) {
            // 父节点是左，叔叔节点就是右了
            // 如果叔叔节点是红的，那么颜色翻转，x = 祖父节点，继续循环
            if ((xppr = xpp.right) != null && xppr.red) {
                xppr.red = false;
                xp.red = false;
                xpp.red = true;
                x = xpp;
            }
            // 否则要旋转
            else {
                // 如果 x 在右边，那么就是 LR 型，需要先进行一个 R型旋转（左旋）
                if (x == xp.right) {
                    root = rotateLeft(root, x = xp);
                    xpp = (xp = x.parent) == null ? null : xp.parent;
                }
                // 然后来一次 L 型旋转，父节点和祖父节点变色，这里作了健壮性判断
                if (xp != null) {
                    xp.red = false;
                    if (xpp != null) {
                        xpp.red = true;
                        root = rotateRight(root, xpp);
                    }
                }
            }
        }
        else {
            // 这里就是 R 型了，一样分析
        }
    }
}
```

扩容时逻辑

当数组实际大小达到一个阈值时，将触发扩容：

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

threshold 其实就是设定的大小乘以扩容因子：

```
newThr = oldThr << 1;
if (newThr == 0) {
    float ft = (float)newCap * loadFactor;
    newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
              (int)ft : Integer.MAX_VALUE);
}
threshold = newThr;
```

上面这段代码逻辑是在 resize 函数中的，也就是说 threshold 其实等于新数组大小乘扩容因子，而新数组大小为原来的两倍。

扩容时其实就是新建一个两倍与旧数组大小的新数组，然后将旧数组中的元素全部拷贝进新数组。前面讲过，由于如果元素原来的下标为 i，那么在新数组中该元素的下标要么为 i，要么为 i + n，这意味着，原先一个桶中的一系列元素可以可以分为两组，一组在 i 中，一组在 i + n 中：

```
for (int j = 0; j < oldCap; ++j) {
    Node<K,V> e;
    if ((e = oldTab[j]) != null) {
        oldTab[j] = null;
        else if (e instanceof TreeNode)
            // 执行树的逻辑
            ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
        else { // preserve order
            Node<K,V> loHead = null, loTail = null;
            Node<K,V> hiHead = null, hiTail = null;
            Node<K,V> next;
            do {
                // 将元素分为两组，一组 low，一组 hi
            } while ((e = next) != null);
            if (loTail != null) {
                loTail.next = null;
                newTab[j] = loHead;  // low 链表放在 j 处
            }
            if (hiTail != null) {
                hiTail.next = null;
                newTab[j + oldCap] = hiHead; // hi 链表放在 j + n 处
            }
        }
    }
}
```

在 jdk1.7 中，**扩容时是一个一个元素使用头插法扩容的，在多线程情况下可能会产生死锁，jdk1.8 中是先得出两个链表，然后直接接入，从而解决这个 BUG。**

对树扩容的逻辑是类似的，因为之前说过，一个树中蕴含着一个链表，因此逻辑也是类似的，不过树中有一些额外的逻辑：

```
if (lc <= UNTREEIFY_THRESHOLD)
    tab[index] = loHead.untreeify(map);
// UNTREEIFY_THRESHOLD = 6
```

一旦数目小于等于 6 个，树就会退化成链表。

删除时逻辑

查找时逻辑就不赘述了，删除时主要看看树退化的条件：

```
if (root == null || (movable && (root.right == null || (rl = root.left)
== null
             || rl.left == null))) {
    tab[index] = first.untreeify(map);  // too small
    return;
}
```

移除树元素退化时并不是以 6 个为基准退化，这与上诉不同，最少时，可以为 4 个。