第二周总结

HashMap 源码分析：
Put 函数
在 map 里放入 key-value 这个键值对

```
public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }
```

实际通过 putVal 函数执行：

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        //1. 如果当前 table 为空，新建默认大小的 table
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
1           //2. 获取当前 key 对应的节点
        if ((p = tab[i = (n - 1) & hash]) == null)
            //3. 如果不存在，新建节点
            tab[i] = newNode(hash, key, value, null);
        else {
            //4. 存在节点
            Node<K,V> e; K k;
            //5. key 的 hash 相同，key 的引用相同或者 key equals，则覆盖
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            //6. 如果当前节点是一个红黑树树节点，则添加树节点
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);
            //7. 不是红黑树节点，也不是相同节点，则表示为链表结构
            else {
                for (int binCount = 0; ; ++binCount) {
                    //8. 找到最后那个节点
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        //9. 如果链表长度超过 8 转成红黑树
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for
1st
                            treeifyBin(tab, hash);
```

```
                        break;
                    }
                    //10.如果链表中有相同的节点，则覆盖
                    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;
                //是否替换掉 value 值
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        //记录修改次数
        ++modCount;
        //是否超过容量，超过需要扩容
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

框 1 中 hash 是 key 通过 hash()方法获取的，i=(n-1) & hash 来算出对应节点在数组的索引位置，如果该位置为空就直接插入

Get 函数

返回传入 key 所对应的 value，如果这个 map 里没有这个 key 的映射对返回 null

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

实际执行函数：

```
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) {
1               if (first.hash == hash && // always check first node
                    ((k = first.key) == key || (key != null && key.equals(k))))
                    return first;

2               if ((e = first.next) != null) {
                    if (first instanceof TreeNode)
                        return ((TreeNode<K,V>)first).getTreeNode(hash, key);

3                   do {
                        if (e.hash == hash &&
                            ((k = e.key) == key || (key != null &&
key.equals(k))))
                            return e;
                    } while ((e = e.next) != null);
                }
            }
            return null;
        }
```

首先这个函数返回基本 map 元素数据结构 Node 类：

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash; //索引值
        final K key;//键值
        V value;//value 值
        Node<K,V> next;//下一个节点

        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
```

框 1 里的代码先判断 Node table 里地第一个节点是否是要找的节点，即判断 first.hash 和 first.key 于传入的参数是否相等

框 2 里得代码判断 first 节点是否是一个 TreeNode（红黑树节点）类型，如果是它是一个红黑树的根节点，执行 getTreeNode(hash, key)方法:

```
final TreeNode<K,V> getTreeNode(int h, Object k) {
        return ((parent != null) ? root() : this).find(h, k, null);
    }
```

这个方法又是通过 find()函数调用执行的：

```
final TreeNode<K,V> find(int h, Object k, Class<?> kc) {
            TreeNode<K,V> p = this;
            do {
                int ph, dir; K pk;
                TreeNode<K,V> pl = p.left, pr = p.right, q;
                if ((ph = p.hash) > h)
                    p = pl;
                else if (ph < h)
                    p = pr;
                else if ((pk = p.key) == k || (k != null && k.equals(pk)))
                    return p;
                else if (pl == null)
                    p = pr;
                else if (pr == null)
                    p = pl;
                else if ((kc != null ||
                          (kc = comparableClassFor(k)) != null) &&
                         (dir = compareComparables(kc, k, pk)) != 0)
                    p = (dir < 0) ? pl : pr;
                else if ((q = pr.find(h, k, kc)) != null)
                    return q;
                else
                    p = pl;
            } while (p != null);
            return null;
        }
```

很明显这个 TreeNode 的 Tree 结构是 BST(红黑树)， 通过与子树根节点的 hash 值（索引值）比较来找到要找的树节点

框 3 中的代码是通过 while ((e = e.next) != null) 查找的普通链表结构的 Node