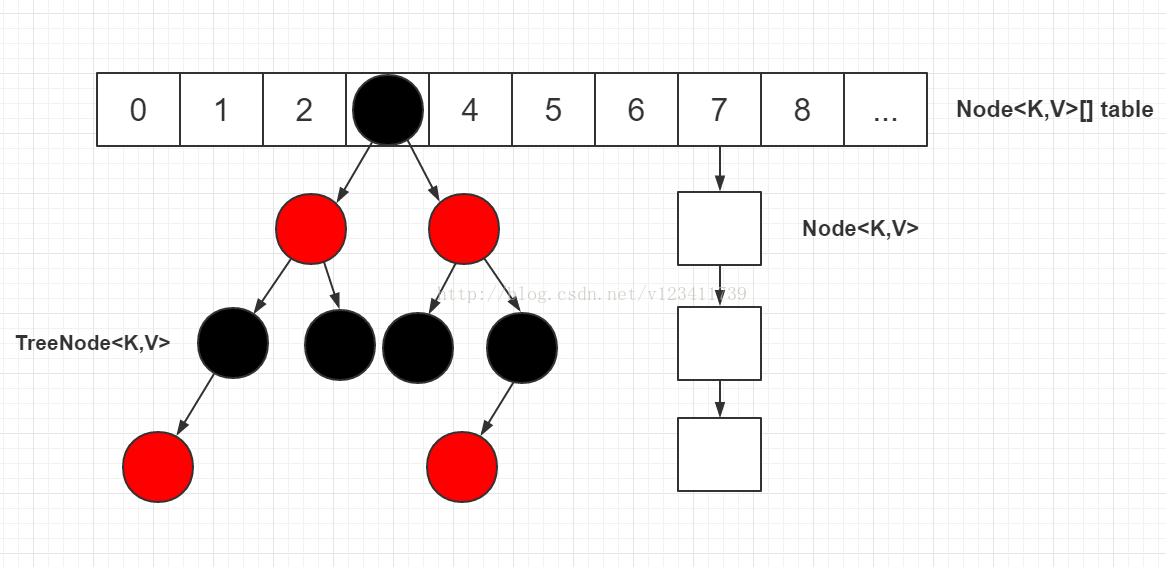
1. 概念

HashMap是一个key-value的键值对集合，key用来存储键值对的键，它是无序不可重复的，value用来存储值。

2. 原理

HashMap的底层原理是一个“链表+数组+红黑树”。横向是一个动态数组，每个数组存储的是一个链表，当链表长度大于8的时候转化为红黑树。



（1）Node节点

一个实现了Map.Entry接口的内部类，用于存储key-value键值对，并且有一个next指针指向下一个对象。

**static class** Node<K,V> **implements** Map.Entry<K,V> {  
 **final int hash**;  
 **final** K **key**;  
 V **value**;  
 Node<K,V> **next**;  
  
 Node(**int** hash, K key, V value, Node<K,V> next) {  
 **this**.**hash** = hash;  
 **this**.**key** = key;  
 **this**.**value** = value;  
 **this**.**next** = next;  
 }  
  
 **public final** K getKey() { **return key**; }  
 **public final** V getValue() { **return value**; }  
 **public final** String toString() { **return key** + **"="** + **value**; }  
  
 **public final int** hashCode() {  
 **return** Objects.*hashCode*(**key**) ^ Objects.*hashCode*(**value**);  
 }  
  
 **public final** V setValue(V newValue) {  
 V oldValue = **value**;  
 **value** = newValue;  
 **return** oldValue;  
 }  
  
 **public final boolean** equals(Object o) {  
 **if** (o == **this**)  
 **return true**;  
 **if** (o **instanceof** Map.Entry) {  
 Map.Entry<?,?> e = (Map.Entry<?,?>)o;  
 **if** (Objects.*equals*(**key**, e.getKey()) &&  
 Objects.*equals*(**value**, e.getValue()))  
 **return true**;  
 }  
 **return false**;  
 }  
}

（2）确定hash桶数组位置

方法一：

**static final int** hash(Object key) {  
 **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);  
}

方法二：

static int indexFor(int h, int length) { //jdk1.7的源码，jdk1.8没有这个方法，但是实现原理一样的

return h & (length-1); //第三步 取模运算

}

（3）put方法

**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;

//判断是否需要重新扩容  
 **if** ((tab = **table**) == **null** || (n = tab.**length**) == 0)  
 n = (tab = resize()).**length**;

//根据hash定位指定位置，查看Node数组的指定位置是否为空  
 **if** ((p = tab[i = (n - 1) & hash]) == **null**)

//如果为空则把元素添加到该位置  
 tab[i] = newNode(hash, key, value, **null**);  
 **else** {  
 Node<K,V> e; K k;

//到这里，已经找到要添加的桶，首先判断头节点是否和要添加的元素相等  
 **if** (p.**hash** == hash &&  
 ((k = p.**key**) == key || (key != **null** && key.equals(k))))  
 e = p;

//如果头节点不想等，那么看该链表是否是红黑树  
 **else if** (p **instanceof** TreeNode)  
 e = ((TreeNode<K,V>)p).putTreeVal(**this**, tab, hash, key, value);  
 **else** {  
 //到这，说明是普通链表，遍历是否有相同的元素，如果有替换，没有添加

**for** (**int** binCount = 0; ; ++binCount) {

//遍历到末尾，直接添加  
 **if** ((e = p.**next**) == **null**) {  
 p.**next** = newNode(hash, key, value, **null**);  
 //判断添加完之后的长度，来确定是否转化为红黑树

**if** (binCount >= ***TREEIFY\_THRESHOLD*** - 1) *// -1 for 1st  
 //转化为红黑树*

treeifyBin(tab, hash);  
 **break**;  
 }

//遇到相同元素，结束循环  
 **if** (e.**hash** == hash &&  
 ((k = e.**key**) == key || (key != **null** && key.equals(k))))  
 **break**;  
 p = e;  
 }  
 }

//如果e不为空，说明遇到相同的元素，用新的value替换旧的value，返回旧的value  
 **if** (e != **null**) { *// existing mapping for key*

V oldValue = e.**value**;  
 **if** (!onlyIfAbsent || oldValue == **null**)  
 e.**value** = value;  
 afterNodeAccess(e);  
 **return** oldValue;  
 }  
 }

//如果不是替换，则会增加一个长度，判断是否需要重新扩容  
 ++**modCount**;  
 **if** (++**size** > **threshold**)  
 resize();  
 afterNodeInsertion(evict);  
 **return null**;  
}

（3）get

**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;

//判断table数组以及指定hash桶是否为空  
 **if** ((tab = **table**) != **null** && (n = tab.**length**) > 0 &&  
 (first = tab[(n - 1) & hash]) != **null**) {

//判断第一个元素是否是要返回的元素  
 **if** (first.**hash** == hash && *// always check first node* ((k = first.**key**) == key || (key != **null** && key.equals(k))))  
 **return** first;  
 **if** ((e = first.**next**) != **null**) {

//判断是否是红黑树  
 **if** (first **instanceof** TreeNode)  
 **return** ((TreeNode<K,V>)first).getTreeNode(hash, key);

//到这说明是普通链表，遍历链表查找  
 **do** {  
 **if** (e.**hash** == hash &&  
 ((k = e.**key**) == key || (key != **null** && key.equals(k))))  
 **return** e;  
 } **while** ((e = e.**next**) != **null**);  
 }  
 }  
 **return null**;  
}

（4）remove

**public** V remove(Object key) {  
 Node<K,V> e;  
 **return** (e = removeNode(*hash*(key), key, **null**, **false**, **true**)) == **null** ?  
 **null** : e.**value**;  
}

**final** Node<K,V> removeNode(**int** hash, Object key, Object value,  
 **boolean** matchValue, **boolean** movable) {  
 Node<K,V>[] tab; Node<K,V> p; **int** n, index;

//首先判断table数组以及指定位置的桶是否为空  
 **if** ((tab = **table**) != **null** && (n = tab.**length**) > 0 &&  
 (p = tab[index = (n - 1) & hash]) != **null**) {  
 Node<K,V> node = **null**, e; K k; V v;

//如果要是出的正是桶的第一个元素  
 **if** (p.**hash** == hash &&  
 ((k = p.**key**) == key || (key != **null** && key.equals(k))))  
 node = p;  
 **else if** ((e = p.**next**) != **null**) {

//如果该链表是一个红黑树  
 **if** (p **instanceof** TreeNode)

//获得要删除的元素  
 node = ((TreeNode<K,V>)p).getTreeNode(hash, key);  
 **else** {

//到这说明是一个普通链表，遍历查找要删除的元素  
 **do** {  
 **if** (e.**hash** == hash &&  
 ((k = e.**key**) == key ||  
 (key != **null** && key.equals(k)))) {  
 node = e;  
 **break**;  
 }

//p始终是e的前一个元素  
 p = e;  
 } **while** ((e = e.**next**) != **null**);  
 }  
 }

//找到要删除的元素  
 **if** (node != **null** && (!matchValue || (v = node.**value**) == value ||  
 (value != **null** && value.equals(v)))) {

//若是红黑树节点，则调用红黑树的删除方法  
 **if** (node **instanceof** TreeNode)  
 ((TreeNode<K,V>)node).removeTreeNode(**this**, tab, movable);

//不然，如果是第一个节点，则让哈希桶指向它的下一个元素作为头  
 **else if** (node == p)  
 tab[index] = node.**next**;

//不是第一个元素，则使p直接指向e的下一个元素来跳过e  
 **else** p.**next** = node.**next**;  
 ++**modCount**;  
 --**size**;  
 afterNodeRemoval(node);  
 **return** node;  
 }  
 }  
 **return null**;  
}