1. **源码初解析**

*/\*\*  
 \* Map初始容量（位运算速度快，且写为16照样在计算机中会转换为二进制）  
 \*/*static final int *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4;  
  
*/\*\*  
 \* Map最大容量  
 \*/*static final int *MAXIMUM\_CAPACITY* = 1 << 30;  
  
*/\*\*  
 \* 装载/扩容因子  
 \*/*static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f;  
  
*/\*\*  
 \* 链表转树阈值  
 \*/*static final int *TREEIFY\_THRESHOLD* = 8;  
  
*/\*\*  
 \* 树转链表阈值  
 \*/*static final int *UNTREEIFY\_THRESHOLD* = 6;  
  
*/\*\*  
 \* 节点类  
 \** ***@param*** <*K*>  
 *\** ***@param*** <*V*>  
 *\*/*static class Node<K,V> implements Map.Entry<K,V> {  
 */\*\*  
 \* 该节点索引  
 \*/* final int hash;  
 final K key;  
 V value;  
 */\*\*  
 \* 指向下一节点  
 \*/* HashMap.Node<K, V> next;  
 ...  
}  
  
*/\*\*  
 \* put方法  
 \** ***@param*** *key  
 \** ***@param*** *value  
 \** ***@return*** *\*/*public V put(K key, V value) {  
 return putVal(*hash*(key), key, value, false, true);  
}  
  
*/\*\*  
 \* 获取key的hash值  
 \* 高16位 ^ 低16位保证了数字的充分使用  
 \* \* @param key  
 \** ***@return*** *\*/*static final int hash(Object key) {  
 int h;  
 return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}  
  
*/\*\*  
 \* put方法核心  
 \** ***@param*** *hash 说白了这里就是key的存放索引  
 \** ***@param*** *key key  
 \** ***@param*** *value value  
 \** ***@param*** *onlyIfAbsent 目前未知  
 \** ***@param*** *evict 目前未知  
 \** ***@return*** *\*/*final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  
 boolean evict) {  
 */\*\*  
 \* 用来存储当前表的临时变量  
 \*/* HashMap.Node<K,V>[] tab;  
 */\*\*  
 \* 节点临时变量，存储某索引下首节点  
 \*/* HashMap.Node<K,V> p;  
 */\*\*  
 \* n：数组长度  
 \* i：数组索引  
 \*/* int n, i;  
 */\*\*  
 \* 如果表为空或者数组长度为0，进行扩容  
 \*/* if ((tab = table) == null || (n = tab.length) == 0)  
 n = (tab = resize()).length;  
 */\*\*  
 \* 如果在i索引上的节点为null，则此节点是此索引上的第一个节点  
 \*/* if ((p = tab[i = (n - 1) & hash]) == null)  
 tab[i] = newNode(hash, key, value, null);  
 */\*\*  
 \* 否则说明此索引上已经存在节点  
 \*/* else {  
 */\*\*  
 \* 用来存储首节点的临时变量  
 \*/* HashMap.Node<K,V> e;  
 */\*\*  
 \* 用来存储某个节点的key  
 \*/* K k;  
 */\*\*  
 \* 如果首节点hash值和此key的hash值一致（索引一致）并且两者相同（地址一致） || key不为空且两者equals相等，  
 \* 则说明需要put的节点与首节点相同，直接覆盖即可  
 \*/* if (p.hash == hash &&  
 ((k = p.key) == key || (key != null && key.equals(k))))  
 e = p;  
 */\*\*  
 \* 否则如果首节点是树节点，说明此处为一棵树，已超出8这个阈值，则将此节点以某种方式放入树种  
 \*/* else if (p instanceof HashMap.TreeNode)  
 e = ((HashMap.TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  
 */\*\*  
 \* 否则说明此处为一链表，且长度 < 8，则遍历该链表  
 \*/* else {  
 for (int binCount = 0; ; ++binCount) {  
 */\*\*  
 \* e：p的下一个节点  
 \* 若p的下一个节点为null，则说明p为最后一个节点，  
 \* 则说明此节点不存在与该链表上，为一个新节点，以某方式加入到此链表上  
 \*/* if ((e = p.next) == null) {  
 p.next = newNode(hash, key, value, null);  
 */\*\*  
 \* 如果到达阈值，则需将链表转换为树结构  
 \*/* if (binCount >= *TREEIFY\_THRESHOLD* - 1) // -1 for 1st  
 treeifyBin(tab, hash);  
 break;  
 }  
 */\*\*  
 \* 如果遇到相同的  
 \*/* if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 break;  
 p = e;  
 }  
 }  
 */\*\*  
 \* 如果e不为空，说明覆盖或是新节点  
 \* 则返回其value值  
 \*/* if (e != null) { // existing mapping for key  
 V oldValue = e.value;  
 if (!onlyIfAbsent || oldValue == null)  
 e.value = value;  
 afterNodeAccess(e);  
 return oldValue;  
 }  
 }  
 */\*\*  
 \* modCount++  
 \*/* ++modCount;  
 */\*\*  
 \* 如果尺寸超过阈值，也需要进行扩容  
 \*/* if (++size > threshold)  
 resize();  
 afterNodeInsertion(evict);  
 return null;  
}