


Java 8 HashMap源码分析

一字马胡 (/u/86c421886c32) [+ 关注](#)
2017.10.22 13:33* 字数 4578 阅读 497 评论 4 喜欢 29
(/u/86c421886c32)

作者：一字马胡 (<http://www.jianshu.com/u/86c421886c32>)
转载标志 【2017-11-03】

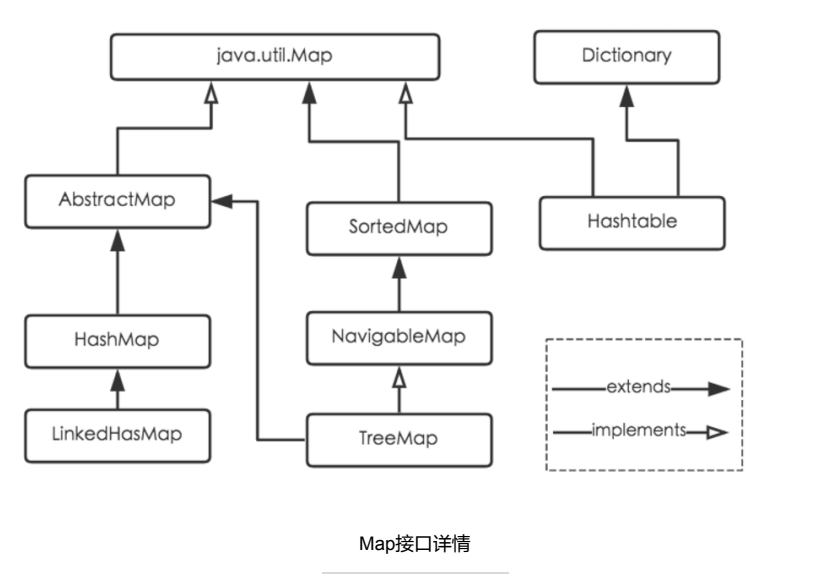
更新日志

日期	更新内容	备注
2017-11-03	添加转载标志	持续更新

导入

HashMap是一种使用最为频繁的<K,V>容器，本文将基于jdk8中HashMap的源码来分析它的实现细节，来探索HashMap是如何为提升效率不断优化设计的，但是，无论HashMap怎么优化怎么高效，都是在单线程环境的前提下，HashMap是不支持并发环境下使用的，因为它线程不安全，至于为什么它线程不安全，可以参考文章为什么HashMap线程不安全 (<http://www.jianshu.com/p/e2f75c8cce01>)，这篇文章详细说明了HashMap为什么不是线程安全的，而且该文章也粗略的分析了一下HashMap的实现细节，但是描述还不太充分，鉴于HashMap的重要性，本文将对HashMap做深度解析，并结合源码分析来深入其内部实现，希望通过分析总结，可以很好的掌握HashMap的特性，以及学习HashMap的精巧设计。

首先，HashMap是一种Map，HashMap仅是一种Map的实现版本，下面的图片展示了java中Map的一些实现版本：



- HashMap：HashMap将根据key的hashCode值来找到存储value的位置，如果hash函数比较完美的话，因为可以很快的找到key对应的value存储的位置，所以具有很高的效率，需要注意的一点是，HashMap因为是基于key的hashCode值来存储value的，所以遍历HashMap不会保证它的顺序和插入时的顺序一致，可以说很大概率这个顺序是不一致的，所以如果需要保持插入顺序，你不可以选择HashMap。还要一点是

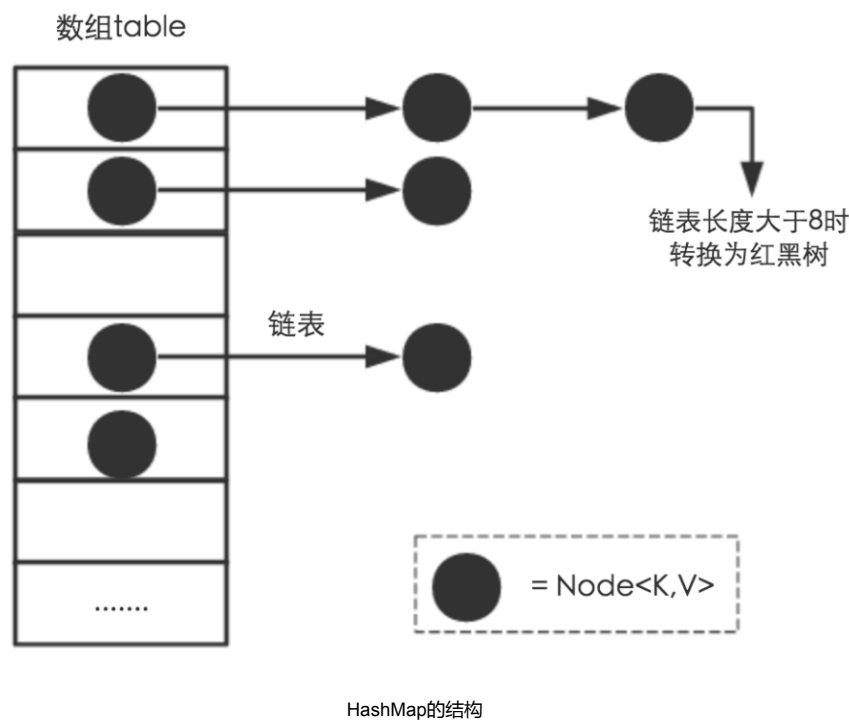
HashMap允许key为null，但是只允许有一个key为null，再次说明，HashMap不是线程安全的，并发环境下你应该首选ConcurrentHashMap，ConcurrentHashMap是一种高效的并发Map，它是线程安全版本的HashMap，至于它的实现细节分析与总结将在其他的文章中进行，本文不对它做分析。

- LinkedHashMap：LinkedHashMap是HashMap的子类，它将保持记录的插入顺序。
- TreeMap：TreeMap实现了SortedMap接口，很明显，他将对插入的记录排序，在遍历TreeMap的时候，得到的是经过排序的记录，所以，如果你需要对插入的记录做排序的话，选择TreeMap，然后指定比较器就可以了。

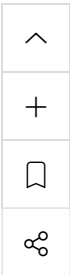
介于篇幅限制，本文仅对HashMap做分析，其他的Map将安排在未来适宜的时刻进行。

HashMap内部结构

首先来看一下HashMap内部结构是什么样子的。通过观察源码，可以发现HashMap在实现上使用了数组+链表+红黑树三种数据结构，可以说在实现上HashMap是比较复杂的，但是这种复杂性带来的收益是很大的，HashMap是一种非常高效的Map，这也是它为什么这么受欢迎的主要原因。下图展示了HashMap的存储结构：



上文中讲到，HashMap是通过计算key的hashCode来找到记录的存储位置的，那因为hash函数不会完美的原因，势必要造成多个记录的key的hashCode一样的情况，上图展示了这种情况，完美情况下，我们希望每一个数组位置上仅有一个记录，但是很多情况下一个数组位置上会落入多个记录，也就是哈希冲突，解决哈希冲突的方法主要有开发地址和链地址，HashMap采用了后者，将hashCode相同的记录放在同一个数组位置上，多个hashCode相同的记录被存储在一条链表上，我们知道，链表上的查询复杂的为O(N)，当这个N很大的时候也就成了瓶颈，所以HashMap在链表的长度大于8的时候就会将链表转换为红黑树这种数据结构，红黑树的查询效率高达O(lgN)，也就是说，复杂度降了一个数量级，完全可以适用于实际生产环境。下面是链表节点数据结构的代码：



```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash; //哈希值, HashMap用这个值来确定记录的位置
    final K key; //记录key
    V value; //记录value
    Node<K,V> next; //链表下一个节点

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

    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;
    }
}
```

下面是上面图中展示的数组：

```
transient Node<K,V>[] table;
```

这个table就是存储数据的数组，上面图中的每个黑色的球是一个Node。下面展示了几个重要的成员变量：



```
/**
 * The number of key-value mappings contained in this map.
 */
transient int size;

/**
 * The next size value at which to resize (capacity * load factor).
 *
 * @serial
 */
// (The javadoc description is true upon serialization.
// Additionally, if the table array has not been allocated, this
// field holds the initial array capacity, or zero signifying
// DEFAULT_INITIAL_CAPACITY.)
int threshold;

/**
 * The load factor for the hash table.
 *
 * @serial
 */
final float loadFactor;

/**
 * The default initial capacity - MUST be a power of two.
 */
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.
 */
static final int MAXIMUM_CAPACITY = 1 << 30;

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

需要注意的一点是，HashMap的哈希桶table的大小必须为2的n次方，也就是说必须为合数，初始大小为16，下文中将会说明为什么一定要是2的n次方。size字段的意思是当前记录数量，loadFactor是负载因子，默认为0.75，而threshold是作为扩容的阈值而存在的，它是由负载因子决定的。下面的方法是返回与给定数值最接近的2的n次方的值：

```
/**
 * Returns a power of two size for the given target capacity.
 */
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

HashMap如何确定记录的table位置

在理解了HashMap的基本存储结构之后，首先来分析一下HashMap是如何确定记录的table位置的。这是至关重要的一步，也是众多HashMap操作的第一步，因为要想找到记录，首先要确定记录在table中的index，然后才能去table的index上的链表或者红黑树里面去寻找记录。下面的方法hash展示了HashMap是如何计算记录的hashCode值的方法：



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

上面的hash方法仅仅是第一步，它只是计算出了hashCode值，但是还可以确定table中的index，接下来的一步需要做的就是根据hashCode来定位index，也就是需要对hashCode取模（ $\text{hashCode} \% \text{length}$ ），length是table的长度，但是我们知道，取模运算是较为复杂的计算，是非常耗时的计算，那有没有方法不通过取模计算而达到取模的效果呢，答案是肯定的，上文中提到，table的长度必然是2的n次方，这点很重要，HashMap通过设定table的长度为2的n次方，在取模的时候就可以通过下面的算法来进行：

```
int index = hashCode & (length - 1)
```

在length总是2的n次方的前提下，上面的算法等效于 $\text{hashCode} \% \text{length}$ ，但是现在通过使用&代替了%，而&的效率要远比%高，为了说明上面的算法是成立的，下面进行试验：

```
hashCode = 8  
length = 4  
  
index = (8 % 4) = 0  
index = 8 & (4-1) = (1000&0011) = 0
```

当然这种证明是没有意义的，更为严谨的证明请参考更多的资料。

HashMap插入元素的过程详解

上面分析了HashMap计算记录在table中的index的方法，下面来分析一下HashMap是如何将一个记录插入到HashMap中去的。也就是HashMap中非常重要的方法put，下面展示了它的实现细节：



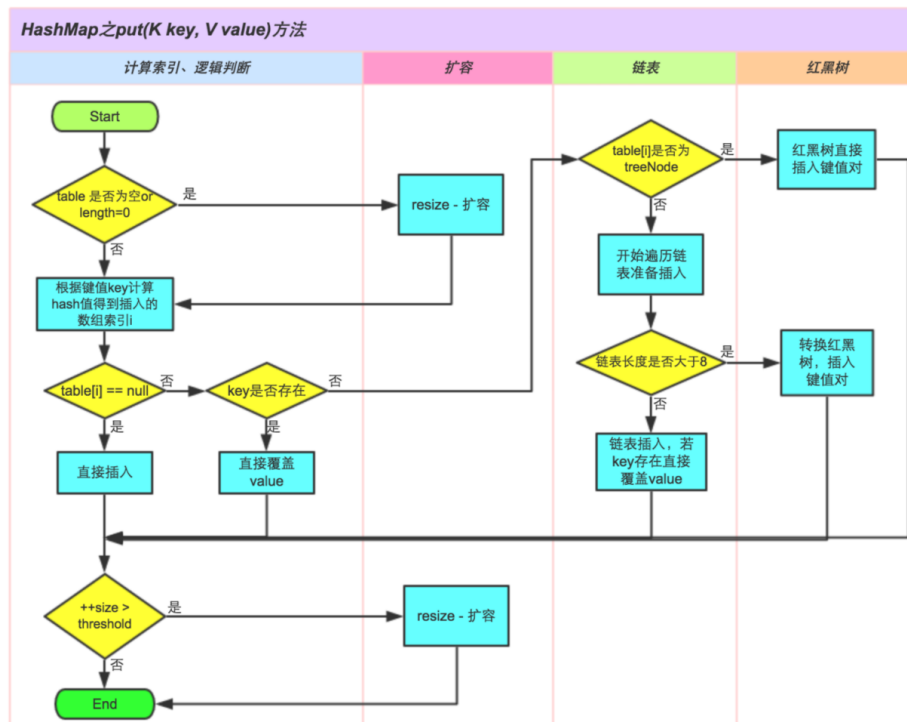
```

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;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    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)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}

```

下面流程图展示了put方法的执行逻辑：



HashMap中put操作的流程

- 首先判断table是否为null或者长度为0，如果是，那么调用方法resize来初始化table，resize的细节将在下文中进行分析，这个方法用来对HashMap的table数组扩容，它将发生在初始化table以及table中的记录数量达到阈值之后。
- 然后计算记录的hashCode，以及根据上文中提到的方法来计算记录在table中的index，如果发现index未知上为null，则调用newNode来创建一个新的链表节点，然后放在table的index位置上，此时表面没有哈希冲突。
- 如果table的index位置不为空，那么说明造成了哈希冲突，这时候如果记录和index位置上的记录相等，则直接覆盖，否则继续判断
- 如果index位置上的节点TreeNode，如果是，那么说明此时的index位置上是一颗红黑树，需要调用putTreeVal方法来将这新的记录插入到红黑树中去。否则走下面的逻辑。
- 如果index位置上的节点类型不是TreeNode，那么说明此位置上的哈希冲突还没有达到阈值，还是一个链表结构，那么就根据插入链表插入新节点的算法来找到合适的位置插入，这里面需要注意的是，新插入的记录会覆盖老的记录，如果这个新的记录是首次插入，那么就会插入到该index位置上链表的最尾部，这里面还需要一次判断，如果插入了新的节点之后达到了阈值，那么就需要调用方法treeifyBin来讲链表转化为红黑树。
- 在插入完成之后，哈希桶中记录的数量是否达到了哈希桶设置的阈值，如果达到了，那么就需要调用方法resize来扩容。

HashMap扩容resize方法详解

上文分析了HashMap的put方法的细节，其中提到，当初始化table以及记录数量达到阈值之时会触发HashMap的扩容，而扩容是通过方法resize来进行的，下面来分析一下resize方法是如何工作的。

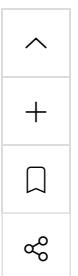


```

final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
            oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else { // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
            (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    ((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 {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        else {
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                    }
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
    }
    return newTab;
}

```

上面展示了resize方法的细节，可以看到扩容的实现时较为复杂的，但是我们知道所谓扩容，就是新申请一个较大容量的数组table，然后将原来的table中的内容都重新计算哈希落到新的数组table中来，然后将老的table释放掉。这里面有两个关键点，一个是新哈希数组的申请以及老哈希数组的释放，另外一个重新计算记录的哈希值以将其插入到新的table中去。首先第一个问题是，扩容会扩大到多少，通过观察上面的代码可以确定，每次扩容都会扩大table的容量为原来的两倍，当然有一个最大值，如果HashMap的容量



已经达到最大值了，那么就不会再进行扩容操作了。第二个问题是HashMap是如何在扩容之后将记录从老的table迁移到新的table中来的。上文中已经提到，table的长度确保是2的n次方，那么有意思的是，每次扩容容量变为原来的两倍，那么一个记录在新table中的位置要么就和原来一样，要么就需要迁移到($\text{oldCap} + \text{index}$)的位置上。下面简单来证明一下这个算法的正确性：

假设原来的table大小为4，那么扩容之后会变为8，那么对于一个元素A来说，如果他的hashCode值为3，那么他在上的位置为 $(3 \& 3) = 3$ ，那么新位置呢？ $(3 \& 7) = 3$ ，这种情况下元素A的index和原来的index是一致的不用变
元素B，他的hashCode值为47，那么在原来table中的位置为 $(47 \& 3) = 3$ ，在新table中的位置为 $(47 \& 7) = 7$ ，是 $(3 + 4)$ ，正好偏移了oldCap个单位。

那么如何快速确定一个记录迁移的位置呢？因为我们的计算方法为： $(\text{hashCode} \& (\text{length} - 1))$ ，而扩容将导致 $(\text{length} - 1)$ 会新增一个1，也就是说，hashCode将会多一位来做判断，如果这个需要新判断的位置上为0，那么index不变，否则变为需要迁移到 $(\text{oldIndex} + \text{oldCap})$ 这个位置上去，下面举个例子吧：

还是上面的两个元素A和B，哈希值分别为3和47，在table长度为4的情况下，因为 $(3) = (11)$ ，所以A和B会有两位获得index，A和B的二进制分别为：

3 : 11
47: 101111

在table的length为4的前提下：

3-> 11 & 11 = 3
47-> 00011 & 101111 = 3

在扩容后，length变为8：

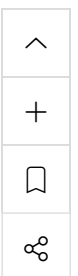
3-> 011 & 111 = 3
47-> 10111 & 01111 = 7

对于3来说，新增的参与运算的位为0，所以index不变，而对于47来说，新增的参与运算的位为1，所以index需要变为 $(\text{index} + \text{oldCap})$

HashMap获取记录操作详解

上面分析了插入记录的操作流程，下面来分析一下HashMap是如何支持获取记录的操作的。我们既然知道了HashMap的结果，就应该大概猜到HashMap需要在我们获取记录的时候要做什么，首先，因为可能会发生哈希冲突，所以我们需要获取的记录可能会存储在一个链表上，也可能存储在一棵红黑树上，这需要实际判断，所以，获取操作首先应该就算记录的hashCode，然后根据hashCode来计算在table中的index，然后判断该数组位置上是一条链表还是一棵红黑树，如果是链表，那么就遍历链表来找到我们需要的记录，否则如果是一棵红黑树，那么就通过遍历这棵红黑树找到我们需要的记录，当然，寻找记录可能会找不到，因为可能我们获取的记录根本就不存在，那么就要返回null暗示用户，当然，HashMap返回null不仅可以代表没有这个记录的信息之外，还可以代表该记录key对应着的value就是null，所以你不能通过HashMap是否返回null来判断HashMap中是否有相应的记录，如果你有类似的需求，你应该调用HashMap的方法：`containsKey`，这个方法将在下文中进行分析。

上面的分析是我们的猜测，下面来看一下HashMap是如何做的，获取元素是通过调用HashMap的get方法来进行的，下面展示了get方法的代码：



```
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;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

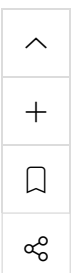
首先会获得当前table的一个快照，然后根据需要查找的记录的key的hashCode来定位到table中的index，如果该位置为null，说明没有记录落到该位置上，也就不存在我们查找的记录，直接返回null。如果该位置不为null，说明至少有一个记录落到该位置上，那么就判断该位置的第一个记录是否使我们查找的记录，如果是则直接返回，否则，根据该index上是一条链表还是一棵红黑树来分别查找我们需要的记录，找到则返回记录，否则返回null。下面来看一下如何判断HashMap中是否有一个记录的方法：

```
public boolean containsKey(Object key) {
    return getNode(hash(key), key) != null;
}
```

这个方法调用了getNode来从table中获得一个Node，返回null，说明不存在该记录，否则存在，containsKey方法和get方法都是通过调用getNode方法来进行的，但是他们的区别在于get方法在判断得到的Node不为null的情况下任然可能返回null，因为Node的value可能为null，所以应该在合适的时候调用合适的方法。

HashMap删除记录详解

现在来看一下HashMap是如何实现删除一个记录的。下面首先展示了相关的代码：



```
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;
    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;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            node = p;
        else if ((e = p.next) != null) {
            if (p instanceof TreeNode)
                node = ((TreeNode<K,V>)p).getTreeNode(hash, key);
            else {
                do {
                    if (e.hash == hash &&
                        ((k = e.key) == key ||
                         (key != null && key.equals(k)))) {
                        node = e;
                        break;
                    }
                    p = e;
                } while ((e = e.next) != null);
            }
        }
        if (node != null && (!matchValue || (v = node.value) == value ||
            (value != null && value.equals(v)))) {
            if (node instanceof TreeNode)
                ((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);
            else if (node == p)
                tab[index] = node.next;
            else
                p.next = node.next;
            ++modCount;
            --size;
            afterNodeRemoval(node);
            return node;
        }
    }
    return null;
}
```

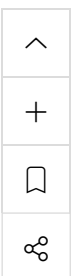
首先，通过记录的hashCode来找到他在table中的index，因为最后需要返回被删除节点的值，所以需要记录被删除的节点。当然记录被删除的节点也是有意义的，比如对于table中的index位置上为一条链表的情况来说，我们只需要记住需要删除的Node，然后真正删除的时候就可以只需要操作该node就可以了，当然对于链表的相关操作详解将在另外的篇章中进行。以及红黑树等高级数据结构的分析总结也会在新的篇章中介绍，目前只需要知道HashMap通过在合适的时候使用不同的数据结构来达到高效的目的就可以了。

HashMap的线程安全详解

本部分的内容请参考为什么HashMap线程不安全

(<http://www.jianshu.com/p/e2f75c8cce01>)，在此不再赘述。大概的意思就是因为是在并发环境下，可能同一时刻有多个线程在操作HashMap，因为HashMap中没有任何措施来保护table，所以在并发环境下多个线程是可以同时操作table的，那么比如在put的时候触发了HashMap扩容，那么在扩容的过程中多个线程的原因可能在某个table的index上会形成一个链表的环，那么此后如果有线程通过get来获取记录的时候，如果刚好这个记录在这个环之后，那么获取记录的线程就会造成死循环，更为具体的分析请参考全文。

本文分析了jdk8中的HashMap，从HashMap是如何计算记录的hashCode的，然后到记录插入操作，以及查询记录操作、删除记录操作等，本文的分析是更像是一种概述，并没有深入到细节中去，比如文中提到了table中的某个index上可能是链表，也可能是一棵红黑树，但是点到为止，并没有详细分析HashMap是如何维护这棵红黑树的，在我看来，分析问题有时候需要联想很多内容，但是一定要有重点，本文的重点是分析HashMap的实现，而HashMap中用到的红黑树只是一种类似工具的内容，况且这涉及到



了一些更为复杂的内容，在这种情况下，如果将多种重要且难以理解的内容柔和在一篇文章中，会造成阅读不顺畅等问题，所以，我的做法是，在每篇文章中尽量只提到一个重要或者难以理解的内容，这样就可以在轻松愉快的前提下快速阅读一篇文章。当然，类似本文中实现的红黑树的原理，实现等分析将一定在另外的篇章中进行分析。

📖 java技术积累 (/nb/16860338) 举报文章 © 著作权归作者所有



一字马胡 (/u/86c421886c32) ♂

写了 90935 字，被 824 人关注，获得了 309 个喜欢 (/u/86c421886c32)




+ 关注

言多必失，行大于言 <https://github.com/pandening>

如果觉得我的文章对您有用，请随意赞赏。您的支持将鼓励我继续创作！


赞赏支持

👍 喜欢 | 29




更多分享

(<http://cwb.assets.jianshu.io/notes/images/1866014>



写下你的评论...

4条评论 只看作者 按喜欢排序 按时间正序 按时间倒序



东风冷雪 (/u/a3ea268aeb60)
2楼 · 2017.10.22 15:32
(/u/a3ea268aeb60)
最不喜欢看java源码了，看着好难受😞

👍 赞

💬 回复

- 一字马胡 (/u/86c421886c32) : @东风冷雪 (/users/a3ea268aeb60) 为啥难受啊？哈哈，看源码要有一个好心态，没必要什么都看懂，能看明白原理就好了，多看几遍有所收获就达到目的了 😊


2017.10.22 16:22  回复
- 东风冷雪 (/u/a3ea268aeb60) : @一字马胡 (/users/86c421886c32) 源码感觉写的太精辟了，泛型什么的，还有借助其他方法，实在。等我在变强些在看吧。。

2017.10.22 17:15  回复
- hfk (/u/b0770aaefa9d) : @东风冷雪 (/users/a3ea268aeb60) 是看了才能变强，而不是变强了才能看懂

2017.10.29 01:04  回复
-  添加新评论

被以下专题收入，发现更多相似内容

+ 收入我的专题






 Java技术 (/c/31b36424c7d2?utm_source=desktop&utm_medium=notes-

^

+

🔖

🔗

-  included-collection) (/c/ac2515ae904e?utm_source=desktop&utm_medium=notes-included-collection)
-  程序员 (/c/NEt52a?utm_source=desktop&utm_medium=notes-included-collection)
-  程序员首页投稿 (/c/89995286335f?utm_source=desktop&utm_medium=notes-included-collection)
-  首页投稿 (/c/bDHhpK?utm_source=desktop&utm_medium=notes-included-collection)
-  创作 (/c/fa5ccc9dd344?utm_source=desktop&utm_medium=notes-included-collection)
-  java (/c/65045af8f9b1?utm_source=desktop&utm_medium=notes-included-collection)
- 展开更多

- 推荐阅读

更多精彩内容 > (/)
- Java CompletableFuture (/p/6ee694cfb54b?utm_ca...** (/p/6ee694cfb54b?utm_campaign=maleskine&utm_content=note&utm_source=recommendation)

作者：一字马胡 转载标志 【2017-11-03】 更新日志 一、java中创建线程的方法在java中有三种方式创建一个线程。 1、继承Thread，重写run方法 2、实... 一字马胡 (/u/86c421886c32?utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)
- Spring Bean 的生命周期 (/p/2b29607b7778?utm_ca...** (/p/2b29607b7778?utm_campaign=maleskine&utm_content=note&utm_source=recommendation)

作者：一字马胡 转载标志 【2017-11-03】 更新日志 Spring Bean 生命周期概述 关于Spring生命周期的资料非常多，内容大同小异，本文就当做是学习笔... 一字马胡 (/u/86c421886c32?utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)
- 我的名字，你的姓氏 (/p/061d1a8db78f?utm_campai...** (/p/061d1a8db78f?utm_campaign=maleskine&utm_content=note&utm_source=recommendation)

1、十六岁那年，我做了一个决定。不再依靠母亲，自己去超市买卫生巾。那时小，只知道胸部前日渐膨胀的生物，它们像两棵嫩芽，在我身体里生根发芽... 凉子菇娘 (/u/af9d9c4db83d?utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)
- 不要不好意思打扮自己 (/p/d4907fbd68a2?utm_camp...** (/p/d4907fbd68a2?utm_campaign=maleskine&utm_content=note&utm_source=recommendation)

1 前段时间刚好看了一篇文章，是关于一个女孩子从小被教育不要爱美的故事，导致这个女孩子长大后心情抑郁,最后自杀了。看到那篇文章后，难免让人深... 我是Aso (/u/7b8915c7f203?utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)
- 鹿晗关晓彤：世界不及你好！ (/p/0d1b834ac73a?utm...** (/p/0d1b834ac73a?utm_campaign=maleskine&utm_content=note&utm_source=recommendation)

01 鹿晗和关晓彤让微博瘫痪了，此时此刻，得知消息的粉丝，心情应该比国庆返程高速路上的车还要堵。鹿晗在微博称：“大家好，给大家介绍一下，这是... 袁曲无闻 (/u/deeea9e09cbc?utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)

+

🔗

🔗