深入 java8 的集合 3：HashMap 的实现原理

一、概述

二话不说，一上来就点开源码，发现里面有一段介绍如下：

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.
翻译一下大概就是在说，这个哈希表是基于 Map 接口的实现的，它允许 null 值和 null 键，它不是线程同步的，同时也不保证有序。

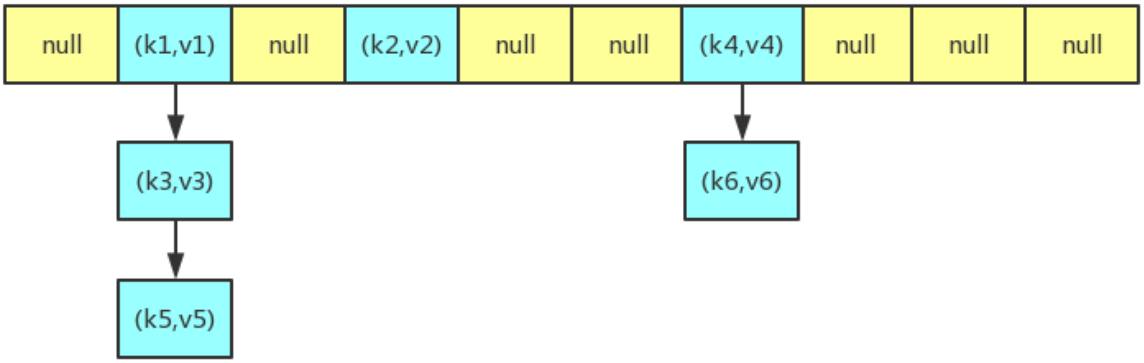
This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the “capacity” of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it’s very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.
不要急，我真的不是来翻译的。再来看看这一段，讲的是 Map 的这种实现方式为 get（取）和 put（存）带来了比较好的性能。但是如果涉及到大量的遍历操作的话，就尽量不要把 capacity 设置得太高（或 load factor 设置得太低），否则会严重降低遍历的效率。

影响 HashMap 性能的两个重要参数：“initial capacity”（初始化容量）和”load factor“（负载因子）。简单来说，容量就是哈希表桶的个数，负载因子就是键值对个数与哈希表长度的一个比值，当比值超过负载因子之后，HashMap 就会进行 rehash 操作来进行扩容。

HashMap 的大致结构如下图所示，其中哈希表是一个数组，我们经常把数组中的每一个节点称为一个桶，哈希表中的每个节点都用来存储一个键值对。在插入元素时，如果发生冲突（即多个键值对映射到同一个桶上）的话，就会通过链表的形式来解决冲突。因为一个桶上可能存在多个键值对，所以在查找的时候，会先通过 key 的

哈希值先定位到桶，再遍历桶上的所有键值对，找出 key 相等的键值对，从而来获取 value。



（因为我这里主要介绍的是 Java 对 HashMap 的实现，而不是 hash 算法，所以如果对哈希算法不了解，建议先去学习一下！）

二、属性

再来看看 HashMap 类中包含了哪些重要的属性，这对下面介绍 HashMap 方法的实现有一定的参考意义。

```
//默认的初始容量为 16

static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;



//最大的容量上限为 2^30

static final int MAXIMUM_CAPACITY = 1 << 30;



//默认的负载因子为 0.75

static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

```
//变成树型结构的临界值为 8
static final int TREEIFY_THRESHOLD = 8;

//恢复链式结构的临界值为 6
static final int UNTREEIFY_THRESHOLD = 6;

//哈希表
transient Node<K,V>[] table;

//哈希表中键值对的个数
transient int size;

//哈希表被修改的次数
transient int modCount;

//它是通过 capacity*load factor 计算出来的，当 size 到达这个值时，就会进行扩容操作
int threshold;

//负载因子
```

```
final float loadFactor;



//当哈希表的大小超过这个阈值，才会把链式结构转化成树型结构，否则仅采取扩容来尝试减少冲突

static final int MIN_TREEIFY_CAPACITY = 64;
```

下面是 Node 类的定义，它是 HashMap 中的一个静态内部类，哈希表中的每一个节点都是 Node 类型。我们可以看到，Node 类中有 4 个属性，其中除了 key 和 value 之外，还有 hash 和 next 两个属性。hash 是用来存储 key 的哈希值的，next 是在构建链表时用来指向后继节点的。

```
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 + "=" + valu
e; }


    public final int hashCode() {
        return Objects.hashCode(key) ^ Objects.hashCode(val
ue);
    }


    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;
    }
```

```
}
```

三、方法

1、get 方法

```
    //get 方法主要调用的是 getNode 方法，所以重点要看 getNode 方法的实现
    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;
        //如果哈希表不为空 && key 对应的桶上不为空
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            //是否直接命中
            if (first.hash == hash && // always check first node
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;
```

```
            //判断是否有后续节点
            if ((e = first.next) != null) {
                //如果当前的桶是采用红黑树处理冲突，则调用红黑树的 get 方法去获取节点
                if (first instanceof TreeNode)
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                //不是红黑树的话，那就是传统的链式结构了，通过循环的方法判断链中是否存在该 key
                do {
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }
```

实现步骤大致如下：
1、通过 hash 值获取该 key 映射到的桶。
2、桶上的 key 就是要查找的 key，则直接命中。
3、桶上的 key 不是要查找的 key，则查看后续节点：
（1）如果后续节点是树节点，通过调用树的方法查找该 key。
（2）如果后续节点是链式节点，则通过循环遍历链查找该 key。

2、put 方法

```
    //put 方法的具体实现也是在 putVal 方法中，所以我们重点看下面的 putVal 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;
        //如果哈希表为空，则先创建一个哈希表
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        //如果当前桶没有碰撞冲突，则直接把键值对插入，完事
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            //如果桶上节点的 key 与当前 key 重复，那你就是我要找的节点了
```

```
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equ
als(k))))
                e = p;
            //如果是采用红黑树的方式处理冲突，则通过红黑树的 putTree
Val 方法去插入这个键值对
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab,
hash, key, value);
            //否则就是传统的链式结构
            else {
                //采用循环遍历的方式，判断链中是否有重复的 key
                for (int binCount = 0; ; ++binCount) {
                    //到了链尾还没找到重复的 key，则说明 HashMap 没有
包含该键
                    if ((e = p.next) == null) {

                        //创建一个新节点插入到尾部
                        p.next = newNode(hash, key, value, nul
l);

                        //如果链的长度大于 TREEIFY_THRESHOLD 这个
临界值，则把链变为红黑树
```

```
                    if (binCount >= TREEIFY_THRESHOLD - 1)
 // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                //找到了重复的 key
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null &&
key.equals(k))))
                    break;
                p = e;
            }
        }
        //这里表示在上面的操作中找到了重复的键，所以这里把该键的
值替换为新值
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
```

```
        //判断是否需要进行扩容
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

put 方法比较复杂，实现步骤大致如下：
1、先通过 hash 值计算出 key 映射到哪个桶。
2、如果桶上没有碰撞冲突，则直接插入。
3、如果出现碰撞冲突了，则需要处理冲突：
（1）如果该桶使用红黑树处理冲突，则调用红黑树的方法插入。
（2）否则采用传统的链式方法插入。如果链的长度到达临界值，则把链转变为红黑树。
4、如果桶中存在重复的键，则为该键替换新值。
5、如果 size 大于阈值，则进行扩容。

3、remove 方法

理解了 put 方法之后，remove 已经没什么难度了，所以重复的内容就不再做详细介绍了。

```
//remove 方法的具体实现在 removeNode 方法中，所以我们重点看下面的 removeNode 方法
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 va
lue,
                           boolean matchValue, boolean movabl
e) {
    Node<K,V>[] tab; Node<K,V> p; int n, index;
    //如果当前 key 映射到的桶不为空
    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;
        //如果桶上的节点就是要找的 key，则直接命中
        if (p.hash == hash && ((k = p.key) == key || (key !=
null && key.equals(k))))
            node = p;
        else if ((e = p.next) != null) {
            //如果是以红黑树处理冲突，则构建一个树节点
            if (p instanceof TreeNode)
                node = ((TreeNode<K,V>)p).getTreeNode(hash,
key);
            //如果是以链式的方式处理冲突，则通过遍历链表来寻找节点
            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);
            }
        }
        //比对找到的 key 的 value 跟要删除的是否匹配
        if (node != null && (!matchValue || (v = node.value)
 == value ||
                             (value != null && value.equals
(v)))) {
            //通过调用红黑树的方法来删除节点
            if (node instanceof TreeNode)
                ((TreeNode<K,V>)node).removeTreeNode(this, t
ab, movable);
            //使用链表的操作来删除节点
            else if (node == p)
                tab[index] = node.next;
            else
                p.next = node.next;
            ++modCount;
```

```
                --size;
                afterNodeRemoval(node);
                return node;
            }
        }
        return null;
    }
```

4、hash 方法

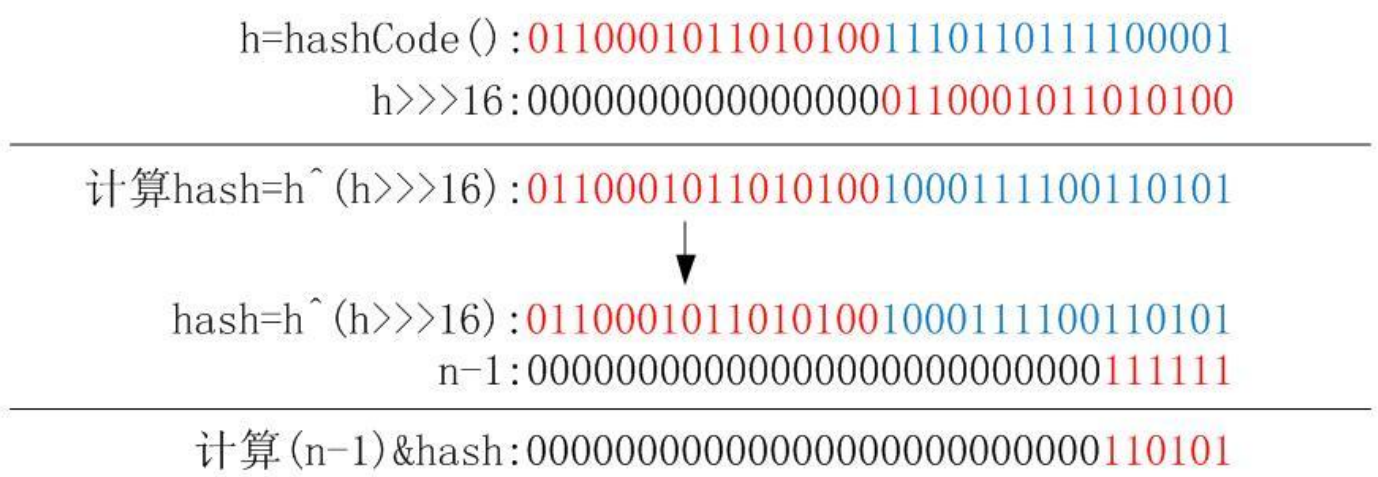
在 get 方法和 put 方法中都需要先计算 key 映射到哪个桶上，然后才进行之后的操作，计算的主要代码如下：

```
(n - 1) & hash
```

上面代码中的 n 指的是哈希表的大小，hash 指的是 key 的哈希值，hash 是通过下面这个方法计算出来的，采用了二次哈希的方式，其中 key 的 hashCode 方法是一个 native 方法：

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

这个 hash 方法先通过 key 的 hashCode 方法获取一个哈希值，再拿这个哈希值与它的高 16 位的哈希值做一个异或操作来得到最后的哈希值，计算过程可以参考下图。为啥要这样做呢？注释中是这样解释的：如果当 n 很小，假设为 64 的话，那么 n-1 即为 63（0x111111），这样的值跟 hashCode()直接做与操作，实际上只使用了哈希值的后 6 位。如果当哈希值的高位变化很大，低位变化很小，这样就很容易造成冲突了，所以这里把高低位都利用起来，从而解决了这个问题。



正是因为与的这个操作，决定了 HashMap 的大小只能是 2 的幂次方，想一想，如果不是 2 的幂次方，会发生什么事情？即使你在创建 HashMap 的时候指定了初始大小，HashMap 在构建的时候也会调用下面这个方法来调整大小：

```
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_C
APACITY : n + 1;

}
```

这个方法的作用看起来可能不是很直观，它的实际作用就是把 cap 变成第一个大于等于 2 的幂次方的数。例如，16 还是 16，13 就会调整为 16，17 就会调整为 32。

5、resize 方法

HashMap 在进行扩容时，使用的 rehash 方式非常巧妙，因为每次扩容都是翻倍，与原来计算（n-1）&hash 的结果相比，只是多了一个 bit 位，所以节点要么就在原来的位置，要么就被分配到“原位置+旧容量”这个位置。

例如，原来的容量为 32，那么应该拿 hash 跟 31（0x11111）做与操作；在扩容扩到了 64 的容量之后，应该拿 hash 跟 63（0x111111）做与操作。新容量跟原来相比只是多了一个 bit 位，假设原来的位置在 23，那么当新增的那个 bit 位的计算结果为 0 时，那么该节点还是在 23；相反，计算结果为 1 时，则该节点会被分配到 23+31 的桶上。

正是因为这样巧妙的 rehash 方式，保证了 rehash 之后每个桶上的节点数必定小于等于原来桶上的节点数，即保证了 rehash 之后不会出现更严重的冲突。

```
final Node<K,V>[] resize() {

    Node<K,V>[] oldTab = table;

    int oldCap = (oldTab == null) ? 0 : oldTab.length;

    int oldThr = threshold;

    int newCap, newThr = 0;

    //计算扩容后的大小

    if (oldCap > 0) {

        //如果当前容量超过最大容量，则无法进行扩容

        if (oldCap >= MAXIMUM_CAPACITY) {

            threshold = Integer.MAX_VALUE;

            return oldTab;

        }

        //没超过最大值则扩为原来的两倍

        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 t
hreshold

        newCap = oldThr;

    else {              // zero initial threshold signifies
using defaults

        newCap = DEFAULT_INITIAL_CAPACITY;

        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIA
L_CAPACITY);

    }

    if (newThr == 0) {

        float ft = (float)newCap * loadFactor;

        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)M
AXIMUM_CAPACITY ?

                  (int)ft : Integer.MAX_VALUE);

    }

    //新的 resize 阈值

    threshold = newThr;

    //创建新的哈希表

    @SuppressWarnings({"rawtypes","unchecked"})

        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];

    table = newTab;

    if (oldTab != null) {
```

```
    //遍历旧哈希表的每个桶，重新计算桶里元素的新位置
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                //如果桶上只有一个键值对，则直接插入
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                //如果是通过红黑树来处理冲突的，则调用相关方法把树分离开
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                //如果采用链式处理冲突
                else { // preserve order
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    //通过上面讲的方法来计算节点的新位置
                    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;

    }
```

在这里有一个需要注意的地方，有些文章指出当哈希表的**桶占用**超过阈值时就进行扩容，这是不对的；实际上是当哈希表中的**键值对个数**超过阈值时，才进行扩容的。

四、总结

通过红黑树的方式来处理哈希冲突是我第一次看见！学过哈希，学过红黑树，就是从来没想到两个可以结合到一起这么用！

按照原来的拉链法来解决冲突，如果一个桶上的冲突很严重的话，是会导致哈希表的效率降低至 O（n），而通过红黑树的方式，可以把效率改进至 O（logn）。相比链式结构的节点，树型结构的节点会占用比较多的空间，所以这是一种以空间换时间的改进方式。