HashMap是java使用频率最高的集合类之一。本文将从重要知识点、主要方法源码分析、与其他集合的比较三个方面来探索**JDK1.8**版本的HashMap。本文目录如下



# 重要知识点

## 继承关系

```
public class HashMap<K,V> extends AbstractMap<K,V>implements Map<K,V>,
Cloneable, Serializable
```

可以看到HashMap继承了AbstractMap实现了三个接口Map<K,V>, Cloneable, Serializable。我的理解是继承一个类是将HashMap**分为map类**，而实现接口是表明HashMap有可复制、可序列化的**能力**。

*顺便说一句，不知道大家有没有想过为什么HashMap既然继承了AbstractMap为什么还要实现Map？并且AbstractMap也实现了Map？我看的时候好奇就去网上搜了搜，据ava集合框架的创始人Josh Bloch描述，这样的写法其实是一个失误。在ava集合框架中，类似这样的写法很多。最开始写ava集合框架的时候，他认为这样写，在某些地方可能是有价值的，直到他意识到错了。显然的，JDK的维护者，后来不认为这个小小的失误值得去修改。所以就这样存在下来了。stack overflow上的回答*

## 重要参数

### 静态常量

```
// 初始容量为2^4=16
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;

// 最大容量=2^30,传入容量过大将被这个值替换
static final int MAXIMUM_CAPACITY = 1 << 30;

// 负载因子=0.75，当键值对个数达到>=容量* 负载因子（0.75）会触发resize扩容
static final float DEFAULT_LOAD_FACTOR = 0.75f;

// 树化的阀值=8，当链表长度大于8，且数组长度大于MIN_TREEIFY_CAPACITY，就会转为红黑树
static final int TREEIFY_THRESHOLD = 8;

// 非树化的阀值=6，当resize时候发现链表长度小于6时，从红黑树退化为链表
static final int UNTREEIFY_THRESHOLD = 6;

//最小的树化容量=64， 在要将链表转为红黑树之前，再进行一次判断，若数组容量小于该值，则用resize扩容，放弃转为红黑树
// 意图：在建立Map的初期，放置过多键值对进入同一个数组下标中，而导致不必要的链表->红黑树的转化，此时扩容即可，可有效减少冲突
static final int MIN_TREEIFY_CAPACITY = 64;
```

重点解释一下负载因子，HashMap不是在容量等于size的时候才扩容，而是在快接近size时候就提前扩容。负载因子就是决定提前到多大。负载因子越大表示散列表的装填程度越高，反之越小。
它默认是0.75，也可以在构造函数里自定义。
负载因子越大，散列表的数据越密集，空间利用率越大，key也越容易冲突化为链表/红黑树，查找效率低；
负载因子越小，散列表的数据越稀疏，对空间的利用越浪费，但key也越不容易冲突，查找效率高。系统默认负载因子为0.75，一般情况下我们是无需修改的。

# 数据结构

HashMap是一个映射散列表，它存储的数据是键值对(key-value)。
JDK1.8前采用数组+链表/红黑树， Node<K,V>[] table数组中的每一个Node元素是一个链表的头结点。这样结合数组和链表的优点，查询效率是大O(1)。

构造函数如下：

```
static class Node<K,V> implements Map.Entry<K,V> {{
        final int hash;
        final K key;
        V value;
        Node<K,V> next;
}
```

# 位运算

HashMap的位运算可以说是老*猪带胸罩，一套接一套的。

## 第一套：扰动函数

HashMap不是直接使用key的hashcode，而是要做异或加工。**目的是减少散列冲突,使元素能够更均匀的分布在数组中。**

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

hashcode是一个int类型32位的数，h >>> 16即把hashcode的高16位向右移动到低16位。然后将**hashCode的高16位和低16位异或**，异或混合过后**高16位的特征也掺杂进低16位**，让数字的每一位都参加了散列运算当中。比如,h代表hashcode

```
h:         1111 1111 1111 1111 1111 0000 1110 1010
h>>16:     0000 0000 0000 0000 1111 1111 1111 1111
h=h^h>>16:1111 1111 1111 1111 0000 1111 0001 0101
```

## 第二套：key是如何hash出对应的数组下标？

hash出应的数组下标理所当然的做法就取余hashcode % length，但jdk用了更有效率的**位操作(length - 1) & hash**来代替取余操作。

## 第三套：为什么长度一定要是2的整次幂？

只有当数组长度是2的整次幂的时候，**(length - 1) & hash才可以代替取余操作hash%length** ，毕竟位运算比取余操作效率更高。当长度是2的整次幂时候，比如8的二进制是1000，肉眼看过去是"一个1后面跟着一堆0"，在减一后就变成了0111，肉眼看过去是"前面全是0后面全是1"。再**和hash与运算出的结果不会超过数组长度，因为前面全是0，与的结果还是0。**

比如长度如果是16，h是上面扰动函数算出的hashcode

```
length-1:     0000 0000 0000 0000 0000 0000 0000 1111
h:            1111 1111 1111 1111 0000 1111 0001 0101
(length-1)&h: 0000 0000 0000 0000 0000 0000 0000 0101
```

假设有两个个key，他们的hashcode不同，分别为code1和code2code1和code2分别与"前面全是0后面全是1"二进制相与，结果一定不同。**但是，如果code1和code2分别与一个"后面不一定是1"的二进制相与，结果有可能相同**

# lazy_load

HashMap是延迟加载，即构造函数不负责初始化，而是由resize（ ）扩容承担初始化的责任。

具体过程是：第一次调用put()方法判断数组是否为空，如果为空调用resize（ ）扩容方法初始化后再put（ ）。

# 重要方法源码分析

# put()

put方法主要由putVal方法实现：

```
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HashMap有没有初始化
        if ((tab = table) == null || (n = tab.length) == 0) //jdk源码的风格 在判断
语句赋值
            n = (tab = resize()).length;
    //如果没有产生hash冲突
        if ((p = tab[i = (n - 1) & hash]) == null)
            //直接在数组tab[i = (n - 1) & hash]处新建一个结点
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            //发生了hash冲突，并且key相同，对结点进行更新

            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
//HashMap允许为空，空值是不能直接判断相等的
                e = p;
            //如果结点是树节点，就插入到红黑树中
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
                //否则，则为链表，遍历查找
                for (int binCount = 0; ; ++binCount) {
                    //到链表尾也没有找到就在尾部插入一个新结点。
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        //注意添加之后链表长度若大于8的话，需将链表转为红黑树
                        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))))
                        //找到就跳出去更新结点的值
                        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;
    }
```

## put（ ）可能造成线程不安全的问题

- JDK8之前，并发put下可能造成死循环。原因是多线程下单链表的数据结构被破坏，指向混乱，造成了链表成环。JDK 8中对HashMap做了大量优化，已经不存在这个问题。

- 并发put，有可能造成键值对的丢失，如果两个线程同时读取到当前node，在链表尾部插入，先插入的线程是无效的，会被后面的线程覆盖掉。

## 总结

1. 判断HashMap有没有初始化，并赋值
2. 如果没有产生hash冲突，直接在数组tab[i = (n - 1) & hash]处新建一个结点；
3. 否则，发生了hash冲突，此时key如果和头结点的key相同，找到要更新的结点，直接跳到最后去更新值
4. 否则，如果数组下标中的类型是TreeNode，就插入到红黑树中
5. 如果只是普通的链表，就在链表中查找，找到key相同的结点就跳出，到最后去更新值；到链表尾也没有找到就在尾部插入一个新结点。
6. 判断此时链表长度若大于8的话，还需要将链表转为红黑树（注意在要将链表转为红黑树之前，再进行一次判断，若数组容量小于64，则用resize扩容，放弃转为红黑树）

# get()

get方法主要由getNode方法实现：

```
final Node<K,V> getNode(int hash, Object key) {
        Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    //判断HashMap有没有被初始化
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            //数组下标的链表头就找到key相同的，那么返回链表头的值
            if (first.hash == hash && // always check first node
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;
            if ((e = first.next) != null) {
                //如果数组下标处的类型是TreeNode，就在红黑树中查找
                if (first instanceof TreeNode)
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                //在链表中遍历查找了
                do {
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }
```

## 总结

1. 如果在数组下标的链表头就找到key相同的，那么返回链表头的值
2. 否则如果数组下标处的类型是TreeNode，就在红黑树中查找
3. 否则就是在普通链表中查找了
4. 都找不到就返回null

remove方法的流程大致和get方法类似。

# resize()

扩容方法有这么一句`newCap = oldCap << 1`说明是扩容后数组大小是原数组的两倍。

同时，该方法也承担了首次put值时，**初始化数组**的责任。

这个方法有点长，我将它分为三段分析。

下面三段在源码中是连在一起的一个方法，只是我这里为了逻辑清晰把它分开了。

第一段，准备好新数组，并做对数组的大小的进行判断，如果是初始化数组，基本工作在这一段就完成了。

```
    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) {
            //如果旧数组的长度已经达到最大容量了2^30
            if (oldCap >= MAXIMUM_CAPACITY) {
                //将阈值修改为int的最大值，不进行扩容直接返回旧数组
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }//新长度是新长度的2倍之后新长度小于最大容量+旧长度大于初始化长度16
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY)
                //阈值扩大一倍
                newThr = oldThr << 1; // double threshold
        }//这种情况是指定了初始容量，new HashMap（int initialCapacity），第一次put初始
化的时候
        else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;
        else {//这种情况是没指定初始容量，new HashMap（），第一次put初始化的时候
            // 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;
        //初始化新数组
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        //如果是初始化，到这里就结束啦，直接跳到最后返回table新数组。
        table = newTab;
```

第二段，遍历原数组每一个结点，有三种情况：只有一个头结点、是红黑树、是链表。

```


        if (oldTab != null) {
            //遍历原数组
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                if ((e = oldTab[j]) != null) {
                    oldTab[j] = null;
                    // 如果数组中只有一个元素，即只有一个头结点，重新哈希新下标就可以了
```

```
                    if (e.next == null)
                        newTab[e.hash & (newCap - 1)] = e;
                    //如果是一个树节点
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                    else { //否则就是链表，这种情况复制比较复杂，单独分一段讲
```

第三段，链表的复制比较复杂。

- 旧链表拆分成两个新链表。首先我们要明白，新数组的的长度是旧数组的两倍。也就是说**旧数组的一个下标可以对应新数组的两个下标**。比如就数组的下标是k，新数组的就对应k和k+oldCap两个下标。于是我们准备**两个链表作为新数组的两个下标的结点**，这里我叫这两个链表为A和B。
- 拆分的标准是e.hash & oldCap == 0。这句其实就是**取e的hashcode在长度范围内的最高位**，其实最高位不外乎两种情况，1和0。但是怎么能取到在长度范围内的最高位呢——把它和长度做与就可得到。比如长度是4，与上e的hashcode得最高位为1。

```
hashcode:1111 1111 1111 1111 0000 1111 0001 0101
length:  0000 0000 0000 0000 0000 0000 0001 0000
         0000 0000 0000 0000 0000 0000 0001 0000
```

如果(e.hash & oldCap) 等于0，则该节点在新、旧数组的下标都是k。

如果(e.hash & oldCap) 不等于0，则该节点在新数组的下标是k+oldCap。

```
                    //loHead指向lo链表的头，loTail指向lo链表尾
                        Node<K,V> loHead = null, loTail = null;
                    //hiHead指向hi链表的头，hiTail指向hi链表尾
                        Node<K,V> hiHead = null, hiTail = null;
                        Node<K,V> next;
                        do {
                            next = e.next;
                            //(e.hash & oldCap) == 0)哈希值最高位是0分到链表lo
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            //否则分到链表hi
                            else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                      //如果lo链表不为空，loHead挂到新数组[原下标]处；
                        if (loTail != null) {
                            loTail.next = null;

                            newTab[j] = loHead;
                        }//如果hi链表不为空，hiHead挂到新数组中[原下标+oldCap]处
                        if (hiTail != null) {
                            hiTail.next = null;
```

```
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
    }
    return newTab;
```

## 总结

1. 如果数组未被初始化，就根据初始化值初始化数组
2. 否则新生成一个长度是原来2倍的新数组，把所有元素复制到新数组
3. 如果元素只有一个节点，复制到重新hash()计算的下标
4. 如果是一个树节点，就对树进行复制
5. 如果是链表，则新生成两个链表，一个挂在原下标位置，一个挂在原下标+原长度位置

# 其他相似的集合

## 与HashTable

一、**是否允许为空。**HashMap可以允许存在**一个**为null的key和任意个为null的value，但是HashTable中的key和value都不允许为null。

当HashMap遇到为null的key时，它会调用putForNullKey方法来进行处理。value如果为空则抛出NullPointerException()

```
if (key == null)            return putForNullKey(value);      而当HashTable遇到null时，他会直接抛出NullPointerException异常信息。
if (value == null) {    throw new NullPointerException();
```

二、**是否线程安全。**Hashtable的方法是线程安全的，而HashMap的方法不是。

Hashtable的方法都是用synchronized修饰的，在修改数组时锁住整个Hashtable，**这样的做法效率很低。**

```
public synchronized V put(K key, V value) {...}
public synchronized V put(K key, V value) {...}
```

三、HashTable基于Dictionary类，而HashMap是基于AbstractMap。

四、HashTable直接调用hashcode，而HashMap会经过扰动函数，　而且用与位运算代替了取余。因此**HashTable的长度不用是2的整次幂**

## 与ConcurrentHashMap

ConcurrentHashMap这个类很重要，我会新开一篇博文探究这个类。

## 与HashSet

## HashMap1.7