HashMap源码分析

HashMap是一个散列表，存储的内容是一个键值对的映射。即通过唯一的key寻找相应的value。注意：key在整个散列表中是唯一的，但是不同的key对应的value是可以相等的。且key可以为null，value也可以为null。

- 存储key和value的静态类

```
static class Node<K, V> implements Map.Entry<K, V> {
    final int hash;
    final K key; //存储了实际的key
    V value; //存储了实际的value
    Node<K, V> next; //这是一个指针，指向了下一个Node节点，即HashMap通过链地址法（拉链法）来解决hash碰撞的问题。

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

Notes:

1. 该类是实际存储key和value的静态类，其中key的类型是K类型的，value的类型是V类型的。分别存储在该类的属性key和value中。
2. 且**key和value只能是引用数据类型，不能是基本数据类型**。因为要判断key是否相等，需要调用hashcode()和equal()方法。而基本数据类型并没有这两个方法。对于null，默认hash值为0。即null为key的key-value映射对总是存储在table索引下标为0的位置。

- HashMap的构造函数

```
/*
初始化一个HashMap时，此时并没有创建Node<K,V> table 数组，仅仅是对一些变量进行了赋值操作。
*/
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                initialCapacity); //如果小于0，则抛异常
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY; //如果大于MAXIMUM-CAPACITY，则设置成MAXIMUM_CAPACITY == 2^30
    if (loadFactor <= 0 || Float.isNaN(loadFactor))  // 检查loadFactor的合法性
        throw new IllegalArgumentException("Illegal load factor: " +
                loadFactor);
```

```
    this.loadFactor = loadFactor; //显示初始化加载因子
    this.threshold = tableSizeFor(initialCapacity); //对传入的initCapacity进行Size规则化。使得capacity的大小永远为2^n次方。n=0,1,2,...,30
}
```

```
/**
* 分情况讨论（由于构造函数有对initCapacity进行负数或者是最大容量检查，所以此时参数只能是0-MAXIMUM_CAPACITY(2^30)）：
* 原理：总共两个操作，移位、或运算。
*       对于任意的n（非0），由于n为一个非0的数，那么n的二进制表示至少有一位为1。仅仅考虑n的最高位1。
*       Step1：进行 n |= n >>> 1，此时最高位1的右边一位不论之前是0还是1，经过与最高位1进行或运算，得到结果总是1。即这一步，让最高位右边的一位变为1
*              若最高位1已经是最低位（最右边），那么这一步以及接下来的几步n为原值。
*       Step2:进行n |= n >>> 2，经过Step1，已经有两个连续的1，执行Step2，得到4个连续的1。
*       Step3:...依次类推。
*       最终的结果：将最高位的1的右边位均变为1。而我们想要得到的是2^n。那么将该结果加1，得到的是原本最高位的左边位为1，其余位均为0。即为2^n
* 特殊情况：1.如果cap == 0，那么 n = -1,即二进制表示为11111111 11111111 11111111 11111111。仅仅考虑最高位1，经过移位、或操作，最终还是-1。返回结果则是1。
*          2.如果cap == MAXIMUM_CAPACITY，那么进行移位、或操作，就会得到 00111111 11111111 11111111 11111111 那么最终加1后取值为MAXIMUM-CAPACITY
*           3.如果cap == 1，那么移位、或操作之后，n == 0，最终加1后取值为1
* 重要点：
*        1. 关注最高位1。移位与与操作即将最高位1右边的位数全部变为1。
*        2. MAXIMUM_CAPACITY = 2^30
*        3. 2^n那么二进制表示只有一位数为1。
* 方法的本质作用：
*              1.用于找到大于等于initialCapacity的最小的2的幂。比如0的时候为1，1的时候为1，2的时候为2，3的时候为4。
*              2.当initCapacity已经是2的幂的时候，那么直接取这个数。这个由函数内部的cap-1来保证。
*/
static final int tableSizeFor(int cap) {
    int n = cap - 1; //减1的原因是如果此时已经是2^n，那么经过下面操作，n变为了2^(n+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;
}
```

- 重要的属性

```
```

```
transient Node<K, V>[] table; //Node<K,V>类型的数组。即实际存放Node<K,V>的数组，即HashMap本质上是一个数组加链表的组合。
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // 当没有在构造器中传入参数initCapacity时，默认HashMap初始容量16
static final int MAXIMUM_CAPACITY = 1 << 30; // 最大容量2^30
static final float DEFAULT_LOAD_FACTOR = 0.75f; // 默认加载因子，完美，不建议修改
```

- HashMap构造方法

  1. 当指定initCapacity和loadFactor时，在将initCapacity规则化之后，采用该intiCapacity和loadFactor。
  2. 当指定了initCapacity而未指定loadFactor时，采用默认loadFactor。然后调用上面的构造方法，进行初始化。
  3. 当initCapacity和loadFactor均未指定时，则仅仅将默认loadFactor赋值给loadFactor，即this.loadFactor = DEFAULT_LOAD_FACTOR。

- important method分析

  1. hash()

```
static final int hash(Object key) {
    int h;

    /*
    1.对于key为null,则返回的hash值为0。从而插入的时候，会直接放在table索引为0的位置。即第一个位置。即HashMap中的key可以为null。但仅仅只能由一个key为null
    2.不然，调用key的原本的hashcode()方法（该方法可能被重写，也可能是直接继承Object类）得到hash值，将该
    hash值的低16位与高16位进行异或操作，作为最终的低16位。将异或过后的hash值作为最终的hash值返回。
    */
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

  2. put()

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K, V>[] tab;
    Node<K, V> p;
    int n, i;
    //首先判断此时是否已经存在table。即存放Node<K,V>节点的数组。
    if ((tab = table) == null || (n = tab.length) == 0)
        //创建新的数组
        n = (tab = resize()).length;
```

```
    /*
    计算某个key所在数组元素位置的方法
    1.由于n是2^n，所以(n-1)使得低n位值全为1，其余位均为0。当数组在该索引下标下无元素时，则直接放到该位。若有元素，则发生了hash碰撞。
    2.当发生了碰撞，则判断是否是相同的元素。判断流程如下：
        2.1.若hash值不相等，那么直接判断条件结束，不需要在判断后面的条件，从而提高了效率（因为hash值不相等的两个key一定不相等），通过链地址法处理hash碰撞。
        2.2.若hash值相等，那么也有可能不是同一个对象，此时要再次判断是否是同一个对象，判断方法如下：
            2.2.1.若两个对象内存地址相等（==比较的是内存地址），那么肯定是统一对象，直接返回True。不再进行判断，提高效率。
            2.2.2.若两个对象内存地址不相等，那么通过调用equals()方法，若相等，那么返回true。因为在java中，通过equals()方法判断两个对象相等，那么这两个对象肯定相等。之所以把equals()方法放在最后，是为了提高效率。毕竟调用方法来判断比较消耗资源。
            2.2.3.当两个对象相等的时候，直接进行新值替代旧值。当两个对象不相等，那么直接通过链地址法来解决hash碰撞。
    */
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);  // table数组在该下标下的位置为空，那么直接将key-value对放到该位置。
    else {
        Node<K, V> e;
        K k;
        if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
            e = p; // 新值替代旧值。否则若if条件不成立，说明对象不相等。则需要将该key-value也插入。
        else if (p instanceof TreeNode)  // 如果此时头节点（即存放在数组中）为红黑数的节点，那么直接调用函数插入到红黑树中。
            e = ((TreeNode<K, V>) p).putTreeVal(this, tab, hash, key, value);
        else {
            /*
                下面整个流程是处理hash碰撞（链表的情况下）
                流程：
                    1.若此时p.next == null，说明之前未发生hash碰撞，那么直接newNode，然后将p节点指向该新节点。
                    2.如果之前发生了hash碰撞，那么整个for循环会沿着当前链表一直遍历寻找下去，每次都会判断是否是相同的key，若是，则新值代替旧值
                        直到到达链表末尾，若仍未找到相同的key，则直接插到链表尾部。
                    3.每次循环都会更新binCount来进行计数。当整个链表的Size>=TREEIFY_THRESHOLD - 1，则将链表转换成红黑树。
                */
            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)))) // 每次都会判断当前key与待插入的key是否相等。如果找到了话，
那么直接break循环
                    break;
                p = e;
            }
        }
        //当在遍历链表的过程中，如果找到了与待插入的key相同的key，那么进行新
值覆盖旧值的过程。并返回旧值。
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;  // 并且会返回旧值。注意该旧值可能为null。因为
value是可以为null的。
            return oldValue;  // 并且会返回旧值。注意该旧值可能为null。因为
value是可以为null的。
        }
    }
    ++modCount;
    /*
    每次插入一个key-value键值对时，都要更新size（即总的key-value对的个数）。
并检查是否需要进行resize()操作。当size的值大于threshold，则需要resize()。
    而threshold=loadFactor * capacity
    */
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    /*
        返回null可能有两种情况：
        1.待插入的key没有key与之相匹配，那么直接插入这个key-value对，并且直
接返回null
        2.待插入的key有相应的key与之相匹配，那么会进行新值取代旧值，并且将旧
值返回。而此时的旧值正好是null
        */
    return null;
}
```

3. `resize()`

注意事项：在初始化一个`HashMap`即调用其构造方法时，并没有创建table。仅仅时对某些属性进行了舒适化操作。比如给`loadFactor`，`initCapacity`等赋值。所以在第一次进行`put(key, value)`时，会调用该`resise()`方法。

    1. 三种构造方法的区别：

        - 当给定了`initCapacity`和`loadFactor`时，会将`threshold`字段赋值为规则化之后`initCapacity`。第一次调用`resize()`方法，会将该`threshold`的值赋值给`capacity`，即当前初始化容量，并且将新的`threshold`的值赋值为`threahold = capacity * loadFactor`，创建`capacity`大小的`table`，完成初始化，直接返回。

- 当仅仅给定了intCapacity，则采用默认的loadFactor，并且后续操作和上面的一样。因为通过查看源码，发现该构造器内部实际上调用了上一个构造器。
- 当什么都没有指定时，构造器内部仅仅将loadFactor字段赋值为默认值。当调用resize()方法时，初始化capacity为DEFAULT_INITAL_CAPACITY，初始化threshold为DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY。

2. 源码分析

```
//TODO：注意绘制该流程图，以一种简洁清晰的方法表达出来。
final Node<K, V>[] resize() {
    Node<K, V>[] oldTab = table;  // 获取oldTable
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
//oldCap > 0表明不是调用构造方法初始化后，第一次调用put()时执行resize()方法。
    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; //resize操作会将原来的容量扩大为原来的1倍。
    } else if (oldThr > 0)  // 表明传入了initCapacity参数，从而有oldThr，可能传入了loadFactor参数
        newCap = oldThr;  // 将oldThreshold的值当作newCapacity
    else {                  // 表示什么都没有传入，构造方法仅仅将loadFactor的值赋值为默认值0.75f。
        newCap = DEFAULT_INITIAL_CAPACITY; // 赋值初始化容量16
        newThr = (int) (DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY); // 新的threshold
    }
    //该if条件只有在传入了initCapacity参数（可能传入了loadFactor参数，也可能没有，但不管怎样，loadFactor的值总是有的，若传入了loadFactor，则此时loadFactor的值即为传入的值，否则，是DEFAULT_LOAD_FACTOR），且第一次调用put()方法，然后调用resize()方法时，才会成立
    if (newThr == 0) {
        float ft = (float) newCap * loadFactor;  // 新的threshold = newCap * loadFactor，且此时的loadFator可能是传入的loadFactor或者是DEFAULT_LOAD_FACTOR
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float) MAXIMUM_CAPACITY ?
                (int) ft : Integer.MAX_VALUE);
    }
    threshold = newThr; // resize之后更新threshold的值
    @SuppressWarnings({"rawtypes", "unchecked"})
    Node<K, V>[] newTab = (Node<K, V>[]) new Node[newCap]; // 根据newCap创建新的table数组。
    table = newTab;  // 将新的table数组饮用赋值给此时table的值oldTab != null 表明不是调用构造方法后，第一次调用put()时执行resize()方法。即只
```

```
有oldTab != null时才会进行rehash操作
    if (oldTab != null) {
        /*
        分析：由于更新了新的数组，所以所有的元素均需要重新进行散列操作，以下即为rehash()过程
        1.首先遍历table数组的每一个元素，若该元素未发生hash碰撞，即Node属性next为null，那么直接计算新位置。e.hash & (newCap - 1)
        2.若该元素发生了hash碰撞，那么遍历该链表来判断链表的每一个节点在新table中的位置。
            2.1.为了完成该操作，首先设置四个Node<K,V>类型的引用。Node<K, V> loHead = null, loTail = null;Node<K, V> hiHead = null, hiTail = null;
            2.2.do-while循环遍历每一个节点。说明如下：
                2.2.1对于链表的每一个节点，该节点在新table中的位置仅仅有两种可能：
                    证明：由于每次进行扩容操作，新的table的capacity都是原来旧的table容量的2倍。且我们知道，容量总是2^n，所以，newCap = 2 ^ (n+1) while oldCap = 2 ^ n
                    由于进行散列操作时，采用的计算方法为 (table.size - 1) & key.hash。那么在oldTab下，此时table.length == 2^n，进行减1操作后，那么此时值二进制位的0-n位全为1，而其他高位全为0。所以散列位置为key.hash二进制表示的低n位所代表的数。在newTab下，散列位置为
                    key.hash后n+1位所表示的数。由于key.hash还是原来的hash，不会发生变化。
                    所以：
                        1.当该key.hash的第n+1位为0时，那么该key在新的table和旧的table的位置一样
                        2.当该key.hash的第n+1位为1时，那么该key在新的table位置为在旧的table位置加上旧的capacity
        */
        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; //如果该桶并没有发生hash碰撞，那么直接计算得到该元素在新桶的位置
                else if (e instanceof TreeNode)
                    ((TreeNode<K, V>) e).split(this, newTab, j, oldCap); // 如果是树节点，那么另外处理
                else {
                    Node<K, V> loHead = null, loTail = null;
                    Node<K, V> hiHead = null, hiTail = null;
                    Node<K, V> next;
                    do {
                    /*
                    遍历链表的每一个节点元素，通过四个引用，将属于原位置的节点找出来，形成一个新的链表，将属于原位置+原容量的节点找出来，形成新的链表
                    */
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {  // 如果条件成立，说明e.hash的第n+1位(oldCap == 2 ^ n)为0。那么在新的table中，仍被散列到原来的索引下标的位置。
```

```
                                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; //直接将头节点赋值给相应的位置，从而整个属于原位置的节点组成的链表正确插入到table中
                        }
                        if (hiTail != null) {
                            hiTail.next = null;
                            newTab[j + oldCap] = hiHead; //直接将头节点赋值给相应的位置，从而整个属于原位置+原容量的节点组成的链表正确插入到table中
                        }
                    }
                }
            }
        }
        return newTab; //将最终新的table返回
    }
```

3. get()

```
    public V get(Object key) {
        Node<K, V> e;
        return (e = getNode(hash(key), key)) == null ? null : e.value;
    }

    public boolean containsKey(Object key) {
        return getNode(hash(key), key) != null;
    }
        /*
        1.HashMap中，null可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为null。当get()方法返回null值时，
        可能是 HashMap中没有该键，也可能使该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键，
        而应该用containsKey()方法来判断。
        2.注意：该方法返回的时Node节点。
        */
    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) {
        //当条件成立，检查哈希桶。首先检查第一个元素
        if (first.hash == hash &&
                ((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; //当空HashMap或者没有该元素，直接返回null。
}
```

- 关于相等的相关知识

1. 如果两个对象调用`equals()`判断是相等的，那么这两个对象调用`hashcode()`方法的返回值一定相同

```
*If two objects are equal according to the {@code equals(Object)}
*method, then calling the {@code hashCode} method on each of
*the two objects must produce the same integer result.
```

2. 如果两个对象调用`equals()`判断是不相等的，那么这两个对象调用`hashcode()`方法的返回值可能相等，也可能不相等。

3. 如果两个对象调用`hashcode()`方法返回值相同，那么这两个对象调用`equals()`方法判断这两个对象可能相等，也可能不相等。

4. 如果两个对象调用`hashcode()`方法返回值不相等，那么这两个对象一定不相等。因为如果这两个对象相等，那么`hashcode()`返回值一定相等。

5. 总结：

    - java中判断两个对象是否相等的核心方法是`equals()`方法。如果`equals()`方法返回`true`，那么这两个对象一定相等。否则这两个对象肯定不相等。如果相等，那么这两个对象调用`hashcode()`的返回值肯定是相等的。如果不相等，那么`hashcode()`的返回值可能相等，也可能不相等。

- 若某个类没有重写`equals()`方法，那么会继承自`Object`类。而`Object`类中的`equals()`方法比较的是对象内存地址。

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

- `equals()`和`==`的区别：
  - `equals()`不能作用于基本数据类型，只能作用于引用类型的变量，如果没有对`equals()`方法进行重写，则比较的是引用类型的变量所指向的对象的地址；诸如`String`、`Integer`等类对`equals`方法进行了重写的话，比较的是所指向的对象的内容。（当然，前提肯定是先比较内存地址，因为内存地址相等了，那么肯定同一个对象。）
  - 对于`==`，比较的是值是否相等。如果作用于基本数据类型的变量，则直接比较其存储的 “值”是否相等；如果作用于引用类型的变量，则比较的是所指向的对象的地址。