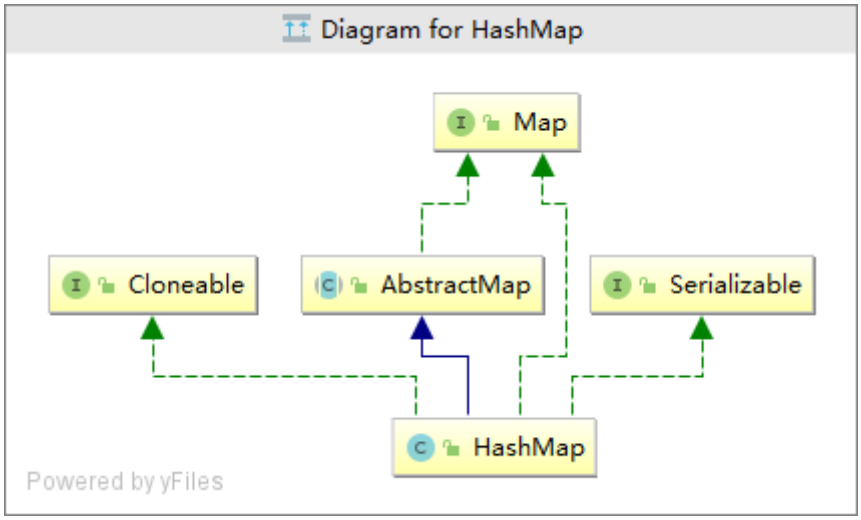
HashMap底层学习

基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了非同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。



HashMap构造方法

```
        Map<Integer, String> map = new HashMap<>();//在JDK1.7之前new HashMap<>()中也是需要声明参数类型的，即new HashMap<Integer, String>()

        //存放值
        map.put(12,"value1");
        map.put(15,"value2");
        map.put(3,"value3");
        map.put(5,"value4");
        map.put(12,"value5");//这里放入跟value1相同的key，发现key并没有被替换，而value被替换了

        System.out.println("Map中的元素：" + map);
        System.out.println("Map中元素的数量：" + map.size());
```

HashMap重要参数

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable { //这里HashMap即是继承了AbstractMap又是实现了Map接口，而在AbstractMap类中也是实现了Map接口，这样就会造成代码的冗余，但是官方并没有修改
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16 ，初始的数组大小，默认为1向左移四位，即16

    static final int MAXIMUM_CAPACITY = 1 << 30; //数组的最大容量，默认为1左移30位

    static final float DEFAULT_LOAD_FACTOR = 0.75f;//负载因子，加载因子，当HashMap的size()到达数组大小*负载因子就会进行扩容操作
```

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

    int threshold; //阈值

    final float loadFactor;


    static class Node<K,V> implements Map.Entry<K,V> {//定义链表节点，由四个属性组成，分别为：哈希码，key,value和下一个节点组成
        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 HashMap() {
        this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted，无参构造，将默认加载因子赋值给加载因子
    }

    public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);//有参构造，可以自定义参数数组初始化大小并将自定义数组大小和负载因子赋值
    }

    public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY; //设置自定义初始化的数组大小不允许超过数组最大大小
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +
                                               loadFactor);
        this.loadFactor = loadFactor;
        this.threshold = tableSizeFor(initialCapacity);
    }

}
```

HashMap的put方法

```
    public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);//调用putVal方法
    }

    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
```

```
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;//定义节点tab和节点p
        if ((tab = table) == null || (n = tab.length) == 0)//判断节点为空
            n = (tab = resize()).length;
        if ((p = tab[i = (n - 1) & hash]) == null)//如果要放置的数组的位置为空，则直接创建一个新节点放置，并且将下一个的节点设为null，i的值为数组大小减一并与哈希码进行与运算，即hash % length -1
            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))))//如果传进来的hash已经存在并且key也相同,将新传进来的节点赋值给临时节点。
                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);//将传进来的值放置在上一个节点的后面：JDK1.8是使用尾插法，JDK1.7使用的是头插法插入链表节点
                        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//如果临时节点不为空。则将新的节点的value赋值给老节点的value而返回老节点的value
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)//如果添加的hashmap的size()大于了阀值，进行扩容操作
            resize();//扩容
        afterNodeInsertion(evict);
        return null;
    }
```

扩容

```
final Node<K,V>[] resize() {//扩容方法
        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) {//如果以前的容量就已经大于了最大容量
                threshold = Integer.MAX_VALUE; //将阀值设置为Integer.MAX.VALUE
                return oldTab;
            }
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY) //将新容量设置为老容量大小左
移一位，即扩大为2倍，且小于最大容量并且老容量大于等于默认容量16
                newThr = oldThr << 1; // double threshold 则将阀值也扩大一倍
        }
        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;
    }
```

问题

1、为什么负载因子默认是0.75

答：负载因子作为1的话：空间利用率得到了很大的满足，但是会很容易在一个数组的同一个下标下产生很长的链表（碰撞）导致查询效率低。

负载因子作为0.5的话碰撞的概率是低了，产生链表的概率也低了，查询效率也高了，但是空间利用率太低。

2、为什么主数组的长度必须为2^n

答：hash & (length -1)等效于hash % length ，而等效的前提是length必须是2的n次幂，用与运算而不用取模的原因是因为位运算的效率高于取余数的效率。

原因2：防止hash冲突，位置冲突，取模的时候可以最大效率使用数组，而如果不用取模就很大概率会产生很长的链表，降低查询效率。