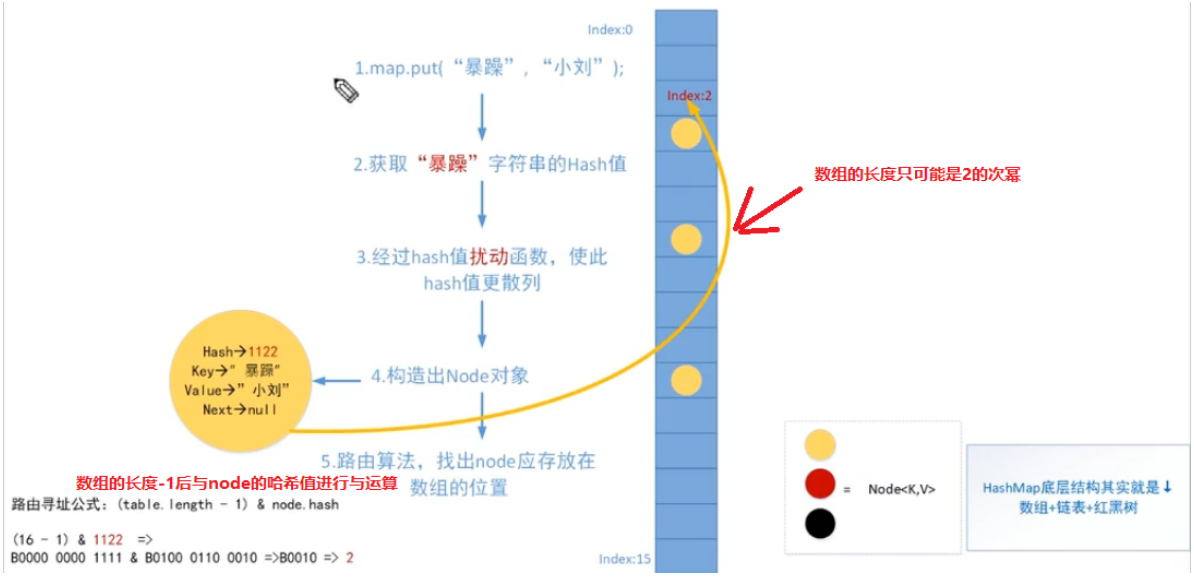
基本属性

- 负载因子是0.75
- 链表转换红黑树的阈值是8，当链表结点大于8的时候就转换为红黑树（转红黑树试桶的最小长度是64）
- 红黑树转链表的阈值是6，当红黑树的节点小于等于6的时候转换为链表
- 初始时默认桶大小为16，构造函数的参数是次方，默认是4，最大为30

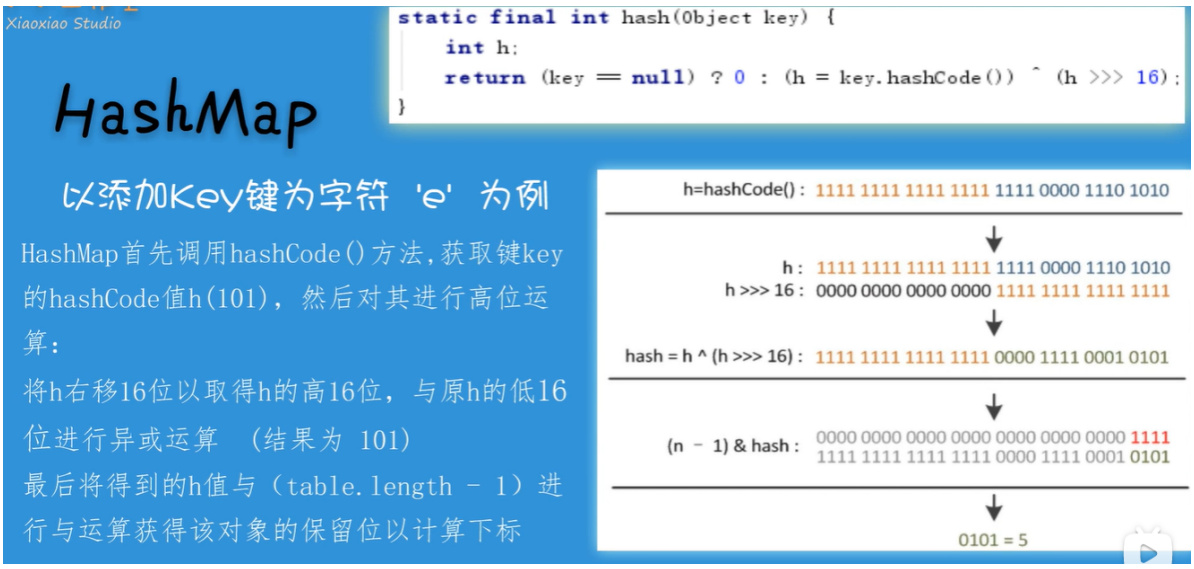
JDK1.8中HashMap的新特性：

- 底层数据结构从“数组+链表”改成“数组+链表+红黑树”，主要是优化了hash i冲突较严重时，链表过长的查找性能:O(n)-> O(logn)
- 计算table初始容量的方式发生了改变，老的方式是从1开始不断向左进行移位运算，直到找到大于等于入参容量的值;新的方式则是通过“5个移位＋或等于运算”来计算。
- 优化了hash 值的计算方式，新的只是简单的让高16位参与了运算。
- 扩容时插入方式从“头插法”改成“尾插法”，避免了并发下的死循环。
- 扩容时计算节点在新表的索引位置方式从“h & (length-1)”改成“hash & oldCap”

put元素



hash方法：



map.put()流程：

1. 构造Node节点：

   首先需要构造一个Node，Node里面放对应的key，value和hash值，hash值是调用hash(key)这个方法获取的。hash方法是这样的：**h = key.hashCode()**， hashCode()这个方法是public native的，是由本地C或C++写的接口；接着h ^ (h >>> 16)，意思是：将h右移16位的高h和原来的低h进行异或运算，最后返回结果。

2. 寻找存放在数组中的位置：

   (table.length - 1) & key.hashCode()来获取到数组的index，源码是这样的：

   ```
   if ((p = tab[i = (n - 1) & hash]) == null)
       tab[i] = newNode(hash, key, value, null);
   ```

   获取到index后直接将新的Node赋值到对应数组位置。

3. 插入（**jdk1.8之前是头插法，之后是尾插**）：

   - 找到数组中对应索引处的链表
   - 如果数组中对应索引处的元素为空，则新创建一个链表，放到这个索引处
   - 如果对应索引处的元素不为空
     - 如果要插入的key和 链表中的第一节点的key相同，那么就把第一个节点赋值给e
     - 如果节点是红黑树：循环遍历链表中的节点，如果在链表中未找到与要插入的节点的key相同的节点，那么插入一个新的节点来存储；插入链表的尾部，如果插入后链表长度大于8则转化为红黑树；

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
    private static final long serialVersionUID = 362498820763181265L;
    transient Node<K,V>[] table;

    transient Set<Map.Entry<K,V>> entrySet;
    /**实际存储的key-value键值对的个数*/
    transient int size;

    transient int modCount;

    /**阈值，当table == {}时，该值为初始容量（初始容量默认为16）；当table被填充了，也就是为table分配内    存空间后，threshold一般为 capacity*loadFactory。HashMap在进行扩容时需要参考threshold，后面会详    细谈到*/
    int threshold;

    /**负载因子，代表了table的填充度有多少，默认是0.75
    加载因子存在的原因，还是因为减缓哈希冲突，如果初始桶为16，等到满16个元素才扩容，某些桶里可能就有不止一个    元素了。所以加载因子默认为0.75，也就是说大小为16的HashMap，到了第13个元素，就会扩容成32。
    */
    final float loadFactor;
    /**HashMap被改变的次数，由于HashMap非线程安全，在对HashMap进行迭代时，
    如果期间其他线程的参与导致HashMap的结构发生变化了（比如put，remove等操作），
    需要抛出异常ConcurrentModificationException*/
    transient int modCount;
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

    static final int MAXIMUM_CAPACITY = 1 << 30;
```

```
    static final float DEFAULT_LOAD_FACTOR = 0.75f;

    static final int TREEIFY_THRESHOLD = 8;//变树阈值，节点数大于8从链表转换为红黑树

    static final int UNTREEIFY_THRESHOLD = 6;//变链表阈值，节点数小于等于6从红黑树转换
为链表

    static final int MIN_TREEIFY_CAPACITY = 64;

    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 HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +
                                               loadFactor);
        this.loadFactor = loadFactor;
        this.threshold = tableSizeFor(initialCapacity);
    }


    public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }


    public HashMap() {
        this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
    }


    public HashMap(Map<? extends K, ? extends V> m) {
        this.loadFactor = DEFAULT_LOAD_FACTOR;
        putMapEntries(m, false);
    }

    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;
        //找到数组中对应索引处的链表
        if ((p = tab[i = (n - 1) & hash]) == null)//将(table.length - 1) &
key.hashCode()；如果数组中对应索引处的元素为空，则新创建一个链表，放到这个索引处

            tab[i] = newNode(hash, key, value, null);

        else {//如果对应索引处的元素不为空
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))//如果要插
入的key和 链表中的第一节点的key相同，那么就把第一个节点赋值给e
                e = p;
            else if (p instanceof TreeNode)//如果节点是红黑树
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
                for (int binCount = 0; ; ++binCount) {//循环遍历链表中的节点
                    if ((e = p.next) == null) {//如果在链表中未找到与要插入的节点的key
相同的节点，那么插入一个新的节点来存储
                        //插入链表的尾部
                        p.next = newNode(hash, key, value, null);
                        //如果插入后链表长度大于8则转化为红黑树
                        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))))//如果找到了这个节点，那么跳出循环
                        break;
                    p = e;
                }
            }
            //判断找到的相同key处的节点，并覆盖这个节点的value
            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;
    }




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

```
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
        //hashCode()这个方法是public native调用本地C或C++写的接口
    }
    //扩容
     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;
    }

}
```

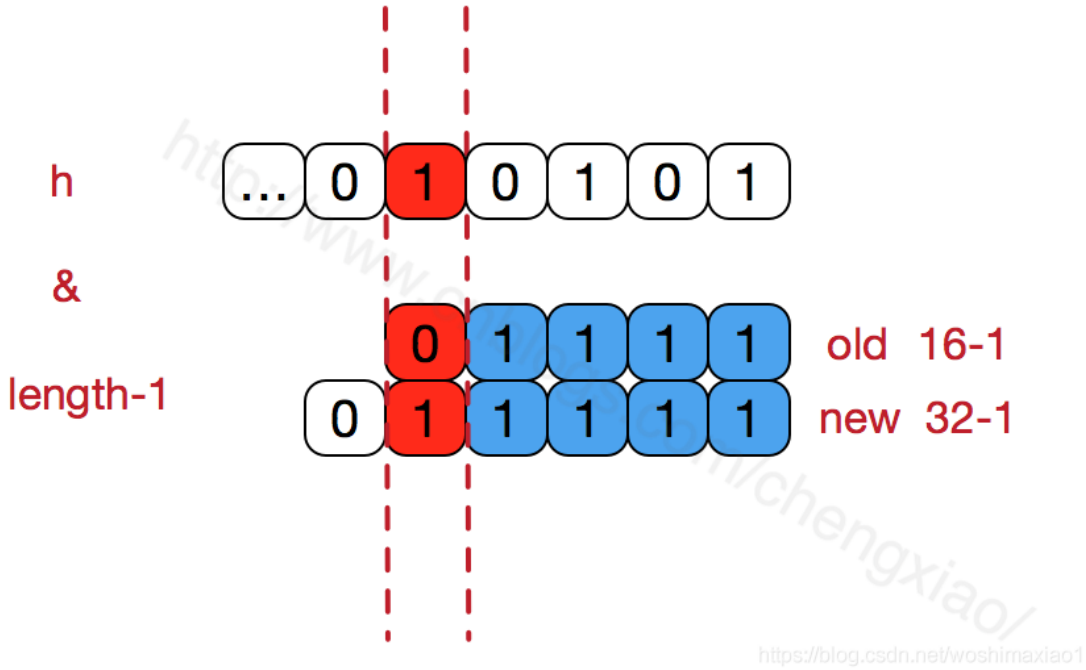
可以看到，如果table长度小于常量MIN_TREEIFY_CAPACITY时，不会变为红黑树，而是调用resize()方法进行扩容。MIN_TREEIFY_CAPACITY的默认值是64。显然HashMap认为，虽然链表长度超过了8，但是table长度太短，只需要扩容然后重新散列一下就可以。

从上面这段代码我们可以看出，在常规构造器中，没有为数组table分配内存空间（有一个入参为指定Map的构造器例外），**而是在执行put操作的时候才真正构建table数组**

后面的代码中可以看到，如果table长度已经达到了64，就会开始变为红黑树，else if中的代码把原来的Node节点变成了TreeNode节点，并且进行了红黑树的转换。

HashMap的长度为什么是2的次幂

降低hash冲突：HashMap的数组长度一定保持2的次幂，比如16的二进制表示为 10000，那么length-1就是15，二进制为01111，同理扩容后的数组长度为32，二进制表示为100000，length-1为31，二进制表示为011111。从下图可以我们也能看到这样会保证低位全为1，而扩容后只有一位差异，也就是多出了最左位的1，这样在通过 h&(length-1)的时候，只要h对应的最左边的那一个差异位为0，就能保证得到的新的数组索引和老数组索引一致(大大减少了之前已经散列良好的老数组的数据位置重新调换)，个人理解。



还有，数组长度保持2的次幂，length-1的低位都为1，会使得获得的数组索引index更加均匀

get（key）方法

```
public V get(Object key) {
        //如果key为null,则直接去table[0]处去检索即可。
        if (key == null)
            return getForNullKey();
        Entry<K,V> entry = getEntry(key);
        return null == entry ? null : entry.getValue();
}
```

get方法通过key值返回对应value，如果key为null，直接去table[0]处检索。我们再看一下getEntry这个方法

```
final Entry<K,V> getEntry(Object key) {

        if (size == 0) {
            return null;
        }
        //通过key的hashcode值计算hash值
        int hash = (key == null) ? 0 : hash(key);
        //indexFor (hash&length-1) 获取最终数组索引，然后遍历链表，通过equals方法比对找出对应记录
        for (Entry<K,V> e = table[indexFor(hash, table.length)];
             e != null;
             e = e.next) {
            Object k;
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                return e;
        }
        return null;
    }
```

可以看出，get方法的实现相对简单，key(hashcode)–>hash–>indexFor–>最终索引位置，找到对应位置table[i]，再查看是否有链表，遍历链表，通过key的equals方法比对查找对应的记录。要注意的是，有人觉得上面在定位到数组位置之后然后遍历链表的时候，e.hash == hash这个判断没必要，仅通过equals判断就可以。其实不然，试想一下，如果传入的key对象重写了equals方法却没有重写hashCode，而恰巧此对象定位到这个数组位置，如果仅仅用equals判断可能是相等的，但其hashCode和当前对象不一致，这种情况，根据Object的hashCode的约定，不能返回当前对象，而应该返回null，后面的例子会做出进一步解释。