HashMap

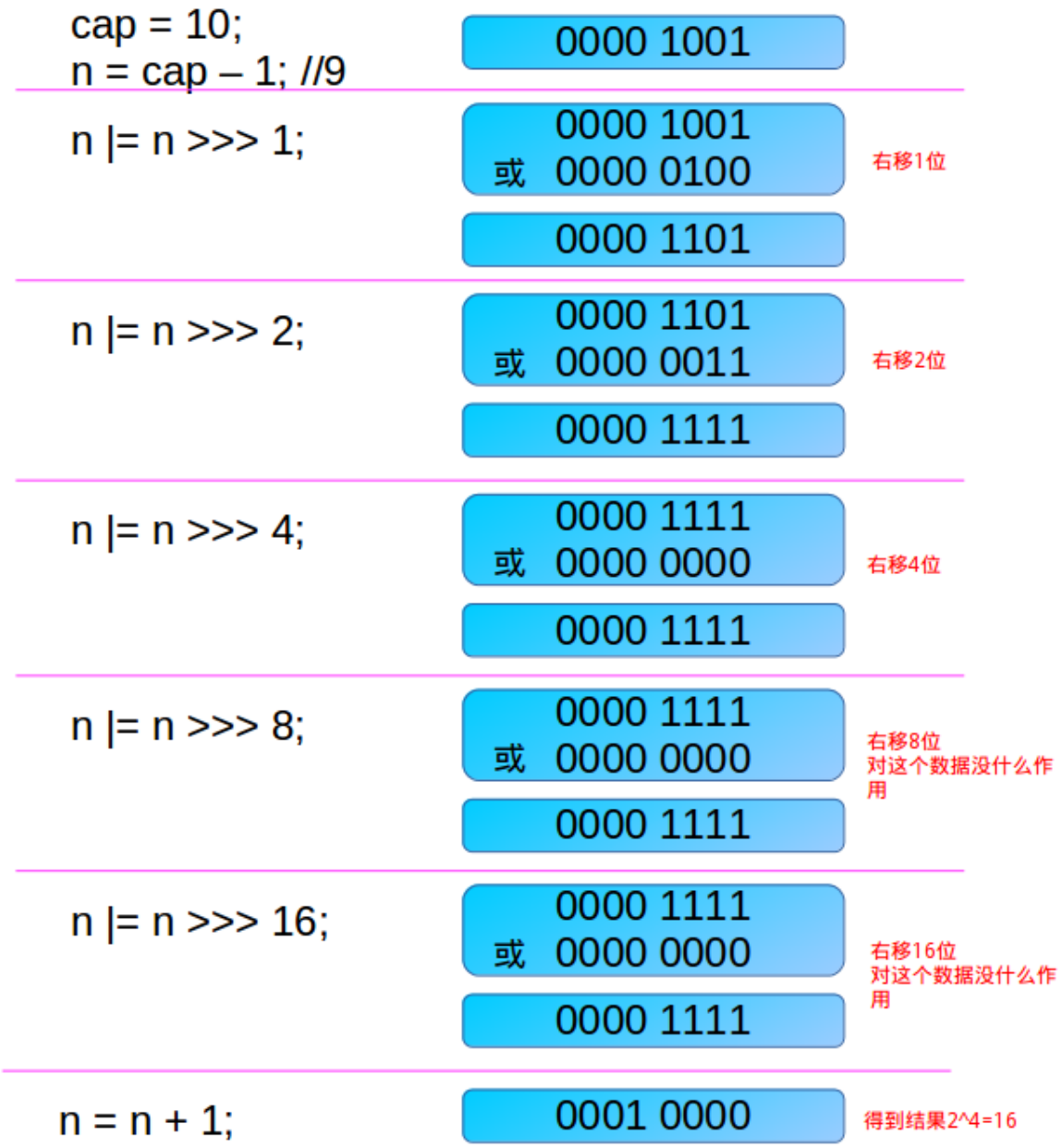
1.0 有参构造方法

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


public HashMap(int initialCapacity, float loadFactor) {
    //安全校验
    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;
    //tableSizeFor() 返回指定容量大小的最小2次幂
    this.threshold = tableSizeFor(initialCapacity);
}
```

```java
/**
 * Returns a power of two size for the given target capacity.
 */
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_CAPACITY :
n + 1;
}
```

- 为什么要进行 n-1操作？
  - 为了防止cap已经是2的幂。如果cap已经是2的幂， 又没有执行这个减1操作，则执行完后面的几条无符号右移操作之后，返回的capacity将是这个cap的2倍

```
//n-1:
0000 0000 0000 0000 0000 0000 0011 0000 //48
0000 0000 0000 0000 0000 0000 0001 1000

>>1: 0000 0000 0000 0000 0000 0000 0011 1000
     0000 0000 0000 0000 0000 0000 0000 1110
>>2: 0000 0000 0000 0000 0000 0000 0011 1110
     0000 0000 0000 0000 0000 0000 0000 0011

>>4: 0000 0000 0000 0000 0000 0000 0011 1111

1+2+4+8+16+32=63

//若不进行n-1:
0000 0000 0000 0000 0000 0000 0010 0000 //32
0000 0000 0000 0000 0000 0000 0001 0000

>>1: 0000 0000 0000 0000 0000 0000 0011 0000
>>2: 0000 0000 0000 0000 0000 0000 0000 1100
     0000 0000 0000 0000 0000 0000 0011 1100
>>4: 0000 0000 0000 0000 0000 0000 0000 0011
     0000 0000 0000 0000 0000 0000 0011 1111  //63
```



1.1 Put()方法

```
// 计算key的hash值
static final int hash(Object key) {
    int h;
    // h=调用Object.hashCode()
    // h ^ h >>> 16

    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

- **为什么要将 key的hashCode与 右移16位后的key 做 异或？**

```
0001 1011 0110 1101  0011 0101 1000 0110
0000 0000 0000 0000  0001 1011 0110 1101

异或:
0001 1011 0110 1101  0010 1110 1110 1011
```

未进行^运算，直接&运算：

```
0001 1011 0110 1101  0011 0101 1000 0110
0000 0000 000 0000  0000  0000 0000 1111
                                0110   6
```

进行^运算后，在进行&运算：

```
0001 1011 0110 1101  0010 1110 1110  1011
0000 0000 000 0000  0000  0000 0000  0110
                                0010   4
```

我们根据Key的哈希值来计算哈希表的索引。如果不做异或运算的话，用&(运算符)运算的话，（二进制运算）只有四位有效、其次会导致计算出来的Hash值相同的很多。若进行异或运算，将key的哈希值高位也做了运算，就可以增加随机性，避免减少Hash冲突。

```
/**
 * The table, initialized on first use, and resized as
 * necessary. When allocated, length is always a power of two.
 * (We also tolerate length zero in some operations to allow
 * bootstrapping mechanics that are currently not needed.)
 */
transient Node<K,V>[] table;
```

```
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) {
    //哈希表
    Node<K,V>[] tab;
    //p 当前节点
    Node<K,V> p;
    //n (num)哈希表的数组长度
    //i (index)tab的索引
    int n, i;

    //校验是否为首次添加
    if ((tab = table) == null || (n = tab.length) == 0)
        //若哈希表(散列表)为空,调用resize()进行初始化哈希表
        n = (tab = resize()).length;

    //hash算法计算 哈希表索引
    //tab[i = (n - 1) & hash]
    //索引为头节点，没有Hash冲突
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else { //存在hash冲突

        // e (element)节点元素
        Node<K,V> e;
        // k (key)key值
        K k;
        //p = tab[i = (n - 1) & hash]
        //哈希表(头节点),比较头节点hash是否和传入对象hash,(key内存地址 |
equals)相等
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
           //相等,赋值给element
            e = p;
        //判断是否为TreeNode节点
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);
        //判断是否为链表
        else {
            //for作用找到key映射的节点
            for (int binCount = 0; ; ++binCount) {
                //if (哈希)链表头结点的下一节点是否为空
                if ((e = p.next) == null) {
```

```
                    //直接放入新节点
                    p.next = newNode(hash, key, value, null);
                    //binCount>= 7直接树化
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for
1st
                        treeifyBin(tab, hash);
                    break;
                }
                //(哈希)链表头结点的下一节点不为空
                //比较该节点hash是否和传入对象hash,key内存地址或
key.equals(key)相等
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null &&
key.equals(k))))
                    break;
                p = e;
            }
        }
        //覆盖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;
}
```

- **tab[i = (n - 1) & hash] 为什么是(n-1) & hash?**

  因为哈希表默认初始容量是16，而要将键值对 放在哈希表(数组)中0-15的位置上，所以就是n-1。

- **put方法() 是如何实现的？：**

  1. 计算key的hashCode值（key的hashCode做了高位 异或 运算）
  2. 散列表若为空时，调用resize()进行初始化
  3. 若散列表不为空，且没有hash冲突，直接将元素加入到散列表中。

4. 若散列表不为空，有hash冲突，会进行三种判断
    1. 判断头节点的key地址相同或equals后内容相同，则新值替换旧值
    2. 判断是否为红黑树结构，是的话就调用树的插入方法。
    3. 如果是链表结构，循环遍历到链表中的空白节点，尾插法进行插入。插入之后判断链表个数是否达到红黑树法的阈值>=7。若在遍历过程中碰到有节点和插入元素的哈希值与key之相等，则覆盖value。

1.2 Get()方法

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

```
/**
 * Implements Map.get and related methods.
 *
 * @param hash hash for key
 * @param key the key
 * @return the node, or null if none
 */
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) {
        if (first.hash == hash && // always check first node
            ((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;
}
```

- **谈一下hashMap中get是如何实现的？**
  1. 计算key的hashCode值（对hashCode高位做 异或 运算）
  2. tab[(n - 1) & hash 获取到链表头节点，如果头节点能找到就直接返回。找不到就在链表或数中进行遍历查找。如果遇到hash冲突就利用key的内存地址或equals方法遍历查找节点。

1.3 resize()方法

```
/**
 * 哈希表 扩容
 *
 * Initializes or doubles table size.  If null, allocates in
 * accord with initial capacity target held in field threshold.
 * Otherwise, because we are using power-of-two expansion, the
 * elements from each bin must either stay at same index, or move
 * with a power of two offset in the new table.
 *
 * @return the table
 */
final Node<K,V>[] resize() {
    //将table赋值给oldTab
    Node<K,V>[] oldTab = table;
    // oldCap=0
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    // oldThr=0
    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
        //第一次初始化哈希表
        //newCap=16
        //newThr=12
        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=12
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})

    //初始化默认容量为16的哈希表
    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;
                    }
                }
            }
        }
    }
    //返回 哈希表
    return newTab;
}
```

1.4 为什么是16？为什么必须是2的幂？如果输入值不是2的幂比如10会怎么样？

- 当n为2的幂次方时，（n-1）& hash 的值是均匀分布的。当n不为2的幂次方时，（n-1）& hash 的值不是是均匀分布的，这样会导致数组的一些位置可能永远不会插入数据，浪费数组的空间，加大hash冲突

当n为2的幂次方时，（ n-1 ）& hash 的值是均匀分布的，我们假设n=16，hash从0开始递增：

当n不为2的幂次方时，（ n-1 ）& hash 的值不是是均匀分布的，我们假设n=15，hash从0开始递增：