HashMap是咱们JAVA程序员使用得最频繁的数据结构，今天就基于jdk1.8来研究一下HashMap的底层实现。

在探讨hashMap前先说一下，两种常见的数据结构，数组与链表。

数组：

数组具有遍历快，增删慢的特点。数组在堆中是一块连续的存储空间，遍历时数组的首地址是知道的（首地址=首地址+元素字节数 * 下标），所以遍历快（数组遍历的时间复杂度为O(1)）；增删慢是因为，当在中间插入或删除元素时，会造成该元素后面所有元素地址的改变，所以增删慢（增删的时间复杂度为O(n)）。

链表：

链表具有增删快，遍历慢的特点。链表中各元素的内存空间是不连续的，一个节点至少包含节点数据与后继节点的引用，所以在插入删除时，只需修改该位置的前驱节点与后继节点即可，链表在插入删除时的时间复杂度为O(1)。但是在遍历时，get(n)元素时，需要从第一个开始，依次拿到后面元素的地址，进行遍历，直到遍历到第n个元素（时间复杂度为O(n)），所以效率极低。

HashMap:

Hash表是一个数组+链表的结构，这种结构能够保证在遍历与增删的过程中，如果不产生hash碰撞，仅需一次定位就可完成，时间复杂度能保证在O(1)。 在jdk1.7中，只是单纯的数组+链表的结构，但是如果散列表中的hash碰撞过多时，会造成效率的降低，所以在JKD1.8中对这种情况进行了控制，当一个hash值上的链表长度大于8时，该节点上的数据就不再以链表进行存储，而是转成了一个红黑树。

hash碰撞：

hash是指，两个元素通过hash函数计算出的值是一样的，是同一个存储地址。当后面的元素要插入到这个地址时，发现已经被占用了，这时候就产生了hash冲突

hash冲突的解决方法：

开放定址法(查询产生冲突的地址的下一个地址是否被占用，直到寻找到空的地址)，再散列法，链地址法等。hashmap采用的就是链地址法，jdk1.7中，当冲突时，在冲突的地址上生成一个链表，将冲突的元素的key，通过equals进行比较，相同即覆盖，不同则添加到链表上，此时如果链表过长，效率就会大大降低，查找和添加操作的时间复杂度都为O(n)；但是在jdk1.8中如果链表长度大于8，链表就会转化为红黑树，时间复杂度也降为了O(logn)，性能得到了很大的优化。

下面通过源码分析一下，HashMap的底层实现

首先，hashMap的主干是一个Node数组（jdk1.7及之前为Entry数组）每一个Node包含一个key与value的键值对，与一个next

next指向下一个node，hashMap由多个Node对象组成。

Node是HhaspMap中的一个静态内部类：

```
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 final K getKey()        { return key; }
        public final V getValue()      { return value; }
        public final String toString() { return key + "=" + value; }

        public final int hashCode() {
            return Objects.hashCode(key) ^ Objects.hashCode(value);
        }

        public final V setValue(V newValue) {
            V oldValue = value;
            value = newValue;
            return oldValue;
        }

        public final boolean equals(Object o) {
            if (o == this)
                return true;
            if (o instanceof Map.Entry) {
                Map.Entry<?,?> e = (Map.Entry<?,?>)o;
                if (Objects.equals(key, e.getKey()) &&
                    Objects.equals(value, e.getValue()))
                    return true;
            }
            return false;
        }
    }
```

再看下hashMap中几个重要的字段：

```
//默认初始容量为16，0000 0001 右移4位 0001 0000为16，主干数组的初始容量为16，而且这个数组
//必须是2的倍数(后面说为什么是2的倍数)
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

//最大容量为int的最大值除2
static final int MAXIMUM_CAPACITY = 1 << 30;

//默认加载因子为0.75
static final float DEFAULT_LOAD_FACTOR = 0.75f;

//阈值，如果主干数组上的链表的长度大于8，链表转化为红黑树
 static final int TREEIFY_THRESHOLD = 8;

//hash表扩容后，如果发现某一个红黑树的长度小于6，则会重新退化为链表
 static final int UNTREEIFY_THRESHOLD = 6;

//当hashmap容量大于64时，链表才能转成红黑树
 static final int MIN_TREEIFY_CAPACITY = 64;

//临界值=主干数组容量*负载因子
int threshold;
```

hashMap的构造方法：

```
 //initialCapacity为初始容量，loadFactor为负载因子
public HashMap(int initialCapacity, float loadFactor) {
        //初始容量小于0，抛出非法数据异常
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        //初始容量最大为MAXIMUM_CAPACITY
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        //负载因子必须大于0，并且是合法数字
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +
                                               loadFactor);

        this.loadFactor = loadFactor;
        //将初始容量转成2次幂
        this.threshold = tableSizeFor(initialCapacity);
    }

    //tableSizeFor的作用就是，如果传入A，当A大于0，小于定义的最大容量时，
  //   如果A是2次幂则返回A，否则将A转化为一个比A大且差距最小的2次幂。
    //例如传入7返回8，传入8返回8，传入9返回16
  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;
    }


    //调用上面的构造方法，自定义初始容量，负载因子为默认的0.75
 public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }


    //默认构造方法，负载因子为0.75，初始容量为DEFAULT_INITIAL_CAPACITY=16，初始容量在第一
次put时才会初始化
 public HashMap() {
        this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
    }


    //传入一个MAP集合的构造方法
 public HashMap(Map<? extends K, ? extends V> m) {
        this.loadFactor = DEFAULT_LOAD_FACTOR;
        putMapEntries(m, false);
    }
```

一次put经历的过程:

1. 首先执行put方法;
2. 然后将key传入hash方法，计算其对应的hash值：
3. 此处如果传入的int类型的值：①向一个Object类型赋值一个int的值时，会将int值自动封箱为Integer。②integer类型的hashcode都是他自身的值，即h=key；h >>> 16为无符号右移16位，低位挤走，高位补0；^ 为按位异或，即转成二进制后，相异为1，相同为0，由此可发现，当传入的值小于 2的16次方-1 时，调用这个方法返回的值，都是自身的值。
4. 最后再执行putVal方法：

```
//onlyIfAbsent是true的话，不要改变现有的值
//evict为true的话，表处于创建模式
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
//如果主干上的table为空，长度为0，调用resize方法，调整table的长度（resize方法在下图中）
        if ((tab = table) == null || (n = tab.length) == 0)
            /* 这里调用resize，其实就是第一次put时，对数组进行初始化。
               如果是默认构造方法会执行resize中的这几句话：
               newCap = DEFAULT_INITIAL_CAPACITY;  新的容量等于默认值16
               newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);

               threshold = newThr;   临界值等于16*0.75
               Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
               table = newTab; 将新的node数组赋值给table，然后return newTab

                如果是自定义的构造方法则会执行resize中的：
                int oldThr = threshold;
                newCap = oldThr;   新的容量等于threshold，这里的threshold都是2的倍数，
原因在
                于传入的数都经过tableSizeFor方法，返回了一个新值，上面解释过
                float ft = (float)newCap * loadFactor;
                newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
                (int)ft : Integer.MAX_VALUE);
                 threshold = newThr; 新的临界值等于 (int)(新的容量*负载因子)
                Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
                table = newTab; return newTab;
            */
            n = (tab = resize()).length;  //将调用resize后构造的数组的长度赋值给n
        if ((p = tab[i = (n - 1) & hash]) == null) //将数组长度与计算得到的hash值比较
            tab[i] = newNode(hash, key, value, null);//位置为空，将i位置上赋值一个
node对象
        else {  //位置不为空
            Node<K,V> e; K k;
            if (p.hash == hash &&  // 如果这个位置的old节点与new节点的key完全相同
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;              // 则e=p
            else if (p instanceof TreeNode) // 如果p已经是树节点的一个实例，既这里已经
是树了
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {  //p与新节点既不完全相同，p也不是treenode的实例
                for (int binCount = 0; ; ++binCount) {  //一个死循环
                    if ((e = p.next) == null) {   //e=p.next,如果p的next指向为null
                        p.next = newNode(hash, key, value, null);  //指向一个新的节
点
                        if (binCount >= TREEIFY_THRESHOLD - 1) // 如果链表长度大于等
于8
                            treeifyBin(tab, hash);  //将链表转为红黑树
```

```
                        break;
                    }
        if (e.hash == hash &&  //如果遍历过程中链表中的元素与新添加的元素完全相同，则跳出循环
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e; //将p中的next赋值给p,即将链表中的下一个node赋值给p,
                           //继续循环遍历链表中的元素
                }
            }
            if (e != null) { //这个判断中代码作用为：如果添加的元素产生了hash冲突，那么调用
                                //put方法时，会将他在链表中他的上一个元素的值返回
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)  //判断条件成立的话，将oldvalue替换
                //为newvalue，返回oldvalue；不成立则不替换，然后返回oldvalue
                    e.value = value;
                afterNodeAccess(e);  //这个方法在后面说
                return oldValue;
            }
        }
        ++modCount;  //记录修改次数
        if (++size > threshold)   //如果元素数量大于临界值，则进行扩容
            resize();   //下面说
        afterNodeInsertion(evict);
        return null;
    }
```

resize的源码面试点详解:

- 扩容机制?
- 单元素如何散列到新的数组中?
- 链表中的元素如何散列到新的数组中?
- 红黑树中的元素如何散列到新的数组中?

```
//上图中说了默认构造方法与自定义构造方法第一次执行resize的过程，这里再说一下扩容的过程
 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) {  //扩容肯定执行这个分支
            if (oldCap >= MAXIMUM_CAPACITY) {   //当容量超过最大值时，临界值设置为int最大值
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY) //扩容容量为2倍，临界值为2倍
                newThr = oldThr << 1;
        }
        else if (oldThr > 0) // 不执行
            newCap = oldThr;
        else {                 // 不执行
            newCap = DEFAULT_INITIAL_CAPACITY;
            newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
```

```
        }
        if (newThr == 0) {  // 不执行
            float ft = (float)newCap * loadFactor;
            newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
?
                      (int)ft : Integer.MAX_VALUE);
        }
        threshold = newThr;     //将新的临界值赋值赋值给threshold
        @SuppressWarnings({"rawtypes","unchecked"})
            Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        table = newTab;    //新的数组赋值给table

        //扩容后，重新计算元素新的位置
        if (oldTab != null) {    //原数组
            for (int j = 0; j < oldCap; ++j) {    //通过原容量遍历原数组
                Node<K,V> e;
                if ((e = oldTab[j]) != null) {    //判断node是否为空，将j位置上的节点
                //保存到e,然后将oldTab置为空，这里为什么要把他置为空呢，置为空有什么好处
吗？？
                //难道是吧oldTab变为一个空数组，便于垃圾回收？？ 这里不是很清楚
                    oldTab[j] = null;
                    if (e.next == null)           //判断node上是否有链表
                        newTab[e.hash & (newCap - 1)] = e; //无链表，确定元素存放位
置，
//扩容前的元素地址为 (oldCap - 1) & e.hash ,所以这里的新的地址只有两种可能，一是地址不变，
//二是变为 老位置+oldCap
                    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;


/* 这里如果判断成立，那么该元素的地址在新的数组中就不会改变。因为oldCap的最高位的1，在e.hash对
应的位上为0，所以扩容后得到的地址是一样的，位置不会改变 ，在后面的代码的执行中会放到loHead中
去，最后赋值给newTab[j]；
如果判断不成立，那么该元素的地址变为 原下标位置+oldCap，也就是lodCap最高位的1，在e.hash对应的
位置上也为1，所以扩容后的地址改变了，在后面的代码中会放到hiHead中，最后赋值给newTab[j +
oldCap]
             举个栗子来说一下上面的两种情况：
            设：oldCap=16 二进制为：0001 0000
                oldCap-1=15 二进制为：0000 1111
                e1.hash=10 二进制为：0000 1010
                e2.hash=26 二进制为：0101 1010
            e1在扩容前的位置为：e1.hash & oldCap-1  结果为：0000 1010
            e2在扩容前的位置为：e2.hash & oldCap-1  结果为：0000 1010
            结果相同，所以e1和e2在扩容前在同一个链表上，这是扩容之前的状态。

    现在扩容后，需要重新计算元素的位置，在扩容前的链表中计算地址的方式为e.hash & oldCap-1
    那么在扩容后应该也这么计算呀，扩容后的容量为oldCap*2=32 0010 0000 newCap=32，新的计算
    方式应该为
    e1.hash & newCap-1
    即：0000 1010 & 0001 1111
    结果为0000 1010与扩容前的位置完全一样。
    e2.hash & newCap-1
    即：0101 1010 & 0001 1111
    结果为0001 1010,为扩容前位置+oldCap。
```

```
    而这里却没有e.hash & newCap-1 而是 e.hash & oldCap，其实这两个是等效的，都是判断倒数第五位
    是0，还是1。如果是0，则位置不变，是1则位置改变为扩容前位置+oldCap。
            再来分析下loTail loHead这两个的执行过程（假设(e.hash & oldCap) == 0成立）：
            第一次执行：
            e指向oldTab[j]所指向的node对象，即e指向该位置上链表的第一个元素
            loTail为空,所以loHead指向与e相同的node对象，然后loTail也指向了同一个node对象。
            最后，在判断条件e指向next，就是指向oldTab链表中的第二个元素
            第二次执行：
            lotail不为null，所以lotail.next指向e，这里其实是lotail指向的node对象的next指向e，
            也可以说是，loHead的next指向了e，就是指向了oldTab链表中第二个元素。此时loHead指向
            的node变成了一个长度为2的链表。然后lotail=e也就是指向了链表中第二个元素的地址。
            第三次执行：
            与第二次执行类似，loHead上的链表长度变为3，又增加了一个node，loTail指向新增的node
               ......
            hiTail与hiHead的执行过程与以上相同，这里就不再做解释了。
            由此可以看出，loHead是用来保存新链表上的头元素的，loTail是用来保存尾元素的，直到遍
            历完链表。    这是(e.hash & oldCap) == 0成立的时候。
            (e.hash & oldCap) == 0不成立的情况也相同，其实就是把oldCap遍历成两个新的链表，
            通过loHead和hiHead来保存链表的头结点，然后将两个头结点放到newTab[j]与
            newTab[j+oldCap]上面去
*/
                                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;    //尾节点的next设置为空
                                newTab[j] = loHead;
                            }
                            if (hiTail != null) {
                                hiTail.next = null;    //尾节点的next设置为空
                                newTab[j + oldCap] = hiHead;
                            }
                        }
                    }
                }
            }
            return newTab;
```

```
    }
```