探究 HashMap 扩容机制

基于 Java1.7 hashmap 源码解析

以下是 HashMap 源码里面的一些关键成员变量以及知识点。在后面的源码解析中会遇到，所以我们有必要先了解下。

---

initialCapacity：初始容量。指的是 HashMap 集合初始化的时候自身的容量。可以在构造方法中指定；如果不指定的话，**总容量默认值是 16** 。需要注意的是初始容量**必须是 2 的幂次方**。

size：当前 HashMap 中已经存储着的**键值对数量**，即 HashMap.size()

loadFactor：加载因子。所谓的加载因子就是 HashMap (**当前的容量/总容量**) 到达一定值的时候，HashMap 会实施扩容。加载因子也可以通过构造方法中指定，**默认的值是 0.75** 。举个例子，假设有一个 HashMap 的初始容量为 16 ，那么扩容的阀值就是 **0.75 * 16 = 12** 。也就是说，在你打算存入第 13 个值的时候，HashMap 会先执行扩容。

threshold：扩容阀值。即 扩容阀值 = HashMap 总容量 * 加载因子。当前 HashMap 的容量大于或等于扩容阀值的时候就会去执行扩容。**扩容的容量为当前 HashMap 总容量的两倍**。比如，当前 HashMap 的总容量为 16 ，那么扩容之后为 32 。

table：**Entry 数组**。我们都知道 HashMap 内部存储 key/value 是通过 Entry 这个介质来实现的。而 table 就是 Entry 数组。

源码解析：

```
/**
 * 返回数组下标
 */
static int indexFor (int h, int length) {
      return h & (length-1);        // 通过 hashcode 和 length 计算 index
}
```

比如：hashcode=30，length=16，那么 length-1=15

第一步：先转成二进制数：30=11110B，15=01111B

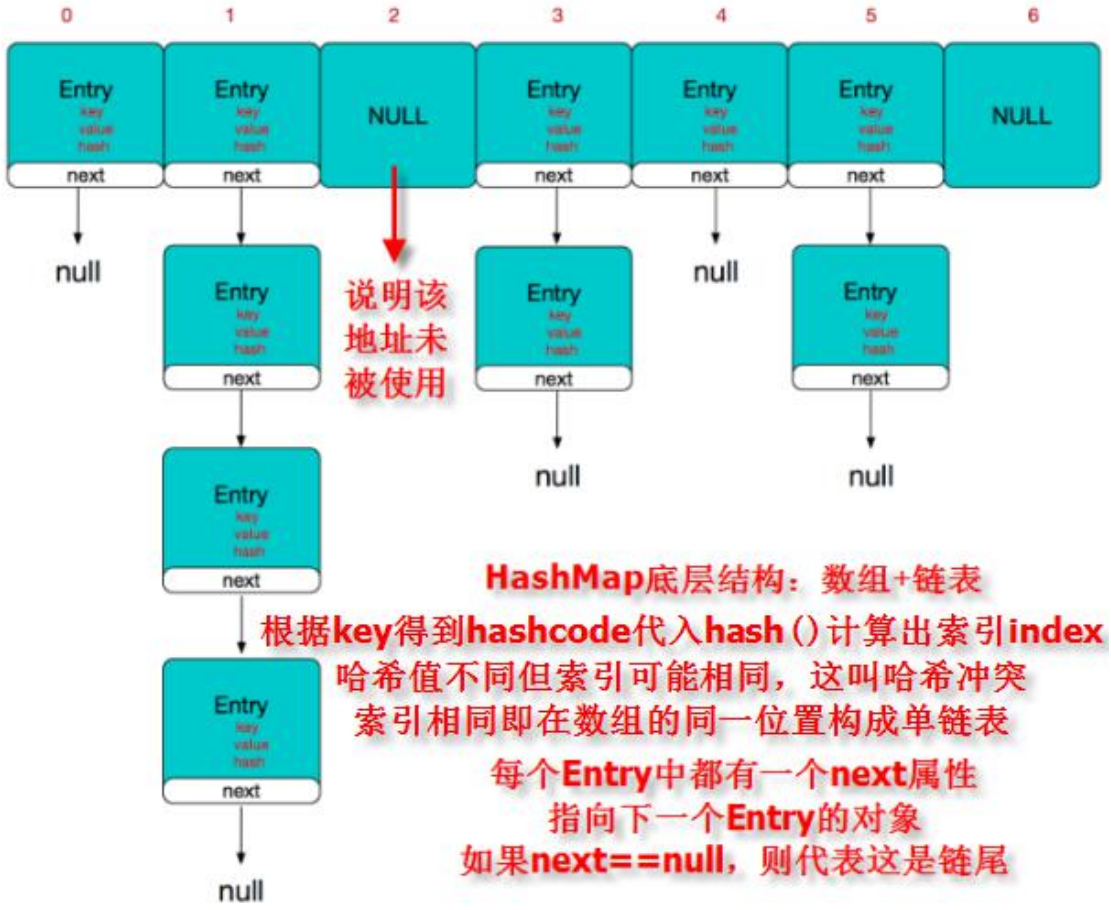
第二步：位运算：逻辑与

1 1 1 1 0

0 1 1 1 1

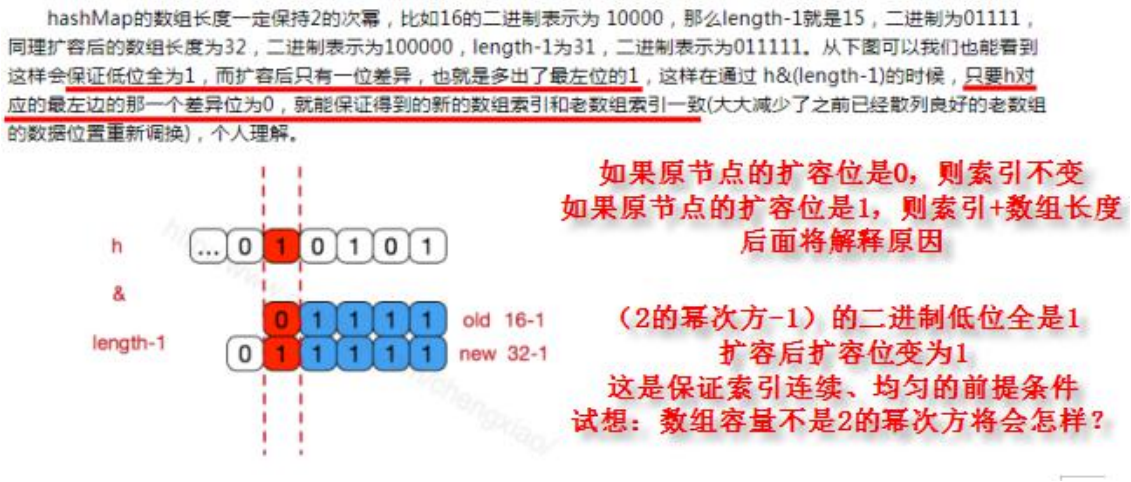
0 1 1 1 0 -> 14 -> 这是 hashcode=30 所对应的数组下标

在 Java 1.7 中，HashMap 的实现方法是**数组 + 链表**的形式。上面的 table 就是数组，而**数组中的每个元素，都是链表的第一个结点**。即如下图所示：



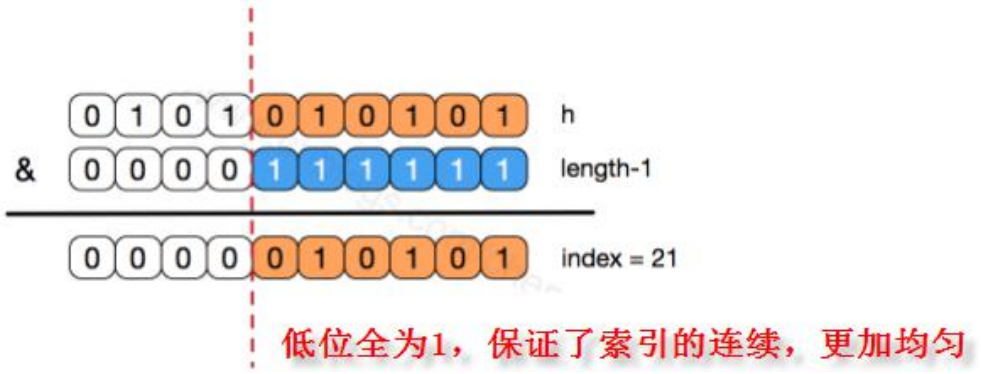
面试题：为什么 HashMap 的数组容量必须为 2 的幂次方？

答：**Root Cause：（2 的幂次方-1）的二进制低位全为 1，哈希算法中正是利用了 hashcode 和 table.length-1 的与运算，得到[0,length-1]区间的索引**

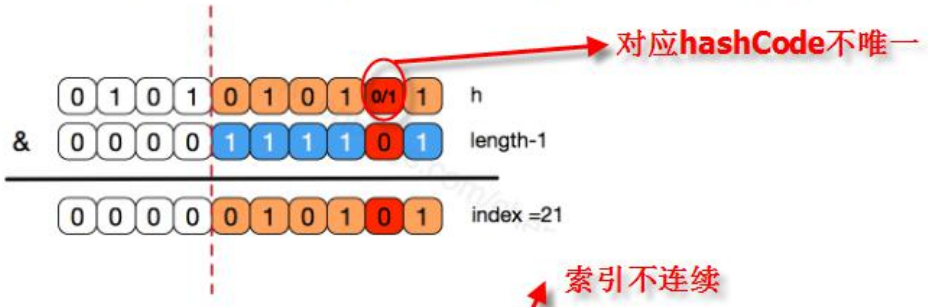


1. 保证扩容前后**索引一致**，有 50%几率分散索引到原索引+oldcap。
2. **索引连续、均匀**。避免数组的某个位置不能被映射，浪费储存空间。
3. **哈希值低位唯一**。因为索引的 0 位可以映射 0/1，导致不能一一映射。

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，比如：



我们看到，上面的&运算，高位是不会对结果产生影响的（hash函数采用各种位运算可能也是为了使得低位更加散列），我们只关注低位bit，如果低位全部为1，那么对于h低位部分来说，任何一位的变化都会对结果产生影响，也就是说，要得到index=21这个存储位置，h的低位只有这一种组合。这也是数组长度设计为必须为2的次幂的原因。



如果不是2的次幂，也就是低位不是全为1此时，要使得index=21，h的低位部分不再具有唯一性了，哈希冲突的几率会变的更大，同时，index对应的这个bit位无论如何不会等于1了，而对应的那些数组位置也就被白白浪费了。

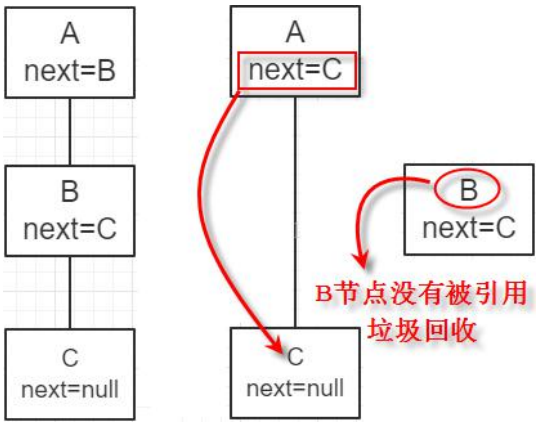
基于 Java1.8 hashmap 源码解析

JDK 1.8 对 HashMap 进行了比较大的优化，底层实现由之前的“数组+链表”改为“**数组+链表/红黑树**”，当链表节点大于 8 时会转为红黑树，红黑树节点小于 6 时会转为链表。请看源码：

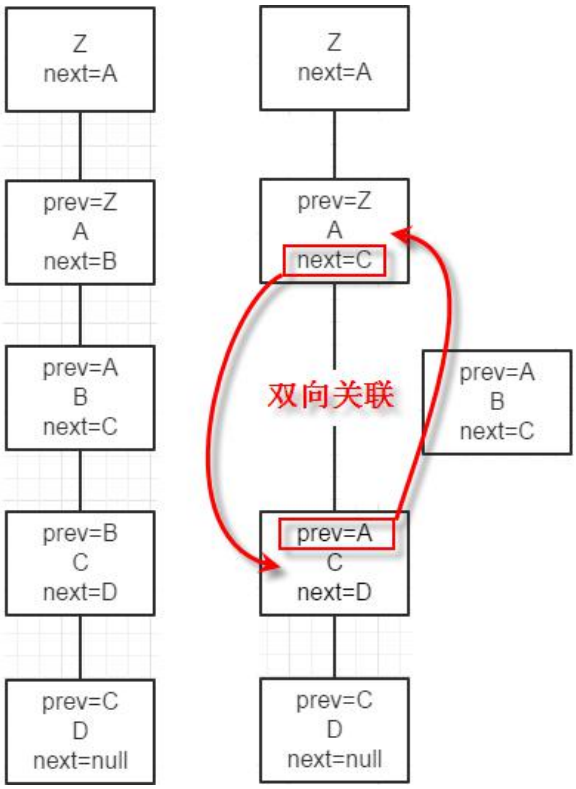
```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;    // 默认容量 16
static final int MAXIMUM_CAPACITY = 1 << 30;    // 最大容量 2 的 30 次方
static final float DEFAULT_LOAD_FACTOR = 0.75f; // 默认负载因子 0.75
static final int TREEIFY_THRESHOLD = 8; // 如果哈希函数不合理，即使扩容也无法减少链表的长度，于是设计当链表长度大于 8 时，转换成红黑树
static final int UNTREEIFY_THRESHOLD = 6;    // 在哈希表扩容时，如果发现链表长度小于 6，则会由树重新退化为链表
static final int MIN_TREEIFY_CAPACITY = 64;    // 转红黑树时，table 的最小长度为 64
```

先了解以下几个点，有利于更好的理解 HashMap 的源码：

1. **头节点**指的是 table 表上索引位置的节点，也就是链表的头节点。
2. **根结点**指的是红黑树最上面的那个节点，也就是没有父节点的节点。
3. **红黑树的根结点不一定是索引位置的头结点**。
4. **转为红黑树节点后，链表的结构还存在**，通过 next 属性维持，红黑树节点在进行操作时都会维护链表的结构，并不是转为红黑树节点，链表结构就不存在了。
5. **在红黑树上，叶子节点也可能有 next 节点**，因为红黑树的结构跟链表的结构是互不影响的，不会因为是叶子节点就说该节点已经没有 next 节点。
6. 源码中一些变量定义：如果定义了一个节点 p，则 pl 为 p 的左节点，pr 为 p 的右节点，pp 为 p 的父节点，ph 为 p 的 hash 值，pk 为 p 的 key 值，kc 为 key 的类等等。
7. 链表中移除一个节点只需如下图操作，其他操作同理：



8. 红黑树在维护链表结构时，移除一个节点只需如下图操作，其他操作同理：（注：此处只是红黑树维护链表结构的操作，**红黑树还需要单独进行红黑树的移除**或者其他操作。）



9. 源码中进行红黑树的查找时，会反复用到以下两条规则：1）如果目标节点的 hash 值小于 p 节点的 hash 值，则向 p 节点的左边遍历；否则向 p 节点的右边遍历。2）如果目标节点的 key 值小于 p 节点的 key 值，则向 p 节点的左边遍历；否则向 p 节点的右边遍历。这两条规则是利用了红黑树的特性（左节点<根结点<右节点）。

10. 源码中进行红黑树的查找时，会用 dir（direction）来表示向左还是向右查找，dir 存储的值是目标节点的 hash/key 与 p 节点的 hash/key 的比较结果。

源码解析：

```
// 红黑树节点，继承自 LinkedHashMap.Entry
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    TreeNode<K,V> parent;  // red-black tree links
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;    // needed to unlink next upon deletion
    boolean red;
}
```

```
// 基本 hash 节点，继承自 Map.Entry
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;
        }
    }
```

不管**增加、删除、查找键值对**，**定位到哈希桶数组的位置**都是很关键的第一步。前面说过 HashMap 的数据结构是“数组+链表/红黑树”的结合，所以我们当然希望这个 HashMap 里面的元素位置尽量分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用 hash 算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，不用遍历链表/红黑树，大大优化了查询的效率。HashMap 定位数组索引位置，直接决定了 hash 方法的离散性能。下面是定位哈希桶数组的源码：

```
// 代码 1
static final int hash(Object key) {        // 计算 key 的 hash 值
    int h;
    // 1.先拿到 key 的 hashCode 值; 2.将 hashCode 的高 16 位参与运算
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
// 代码 2
int n = table.length;
int index = (n - 1) & hash;          // 将(table.length - 1) 与 hash 值进行&运算
```

源码解析：

(h = key.hashCode()) ^ (h >>> 16)

第一步：调用 hashCode()方法计算出 hash 的值，这个值是 32 位的；

第二步：赋值给 h；

第三步：计算 h>>>16

>>>是无符号右移运算符

相当于最高位不是符号位，也就是说，hash 值右移 16 位，不管符号

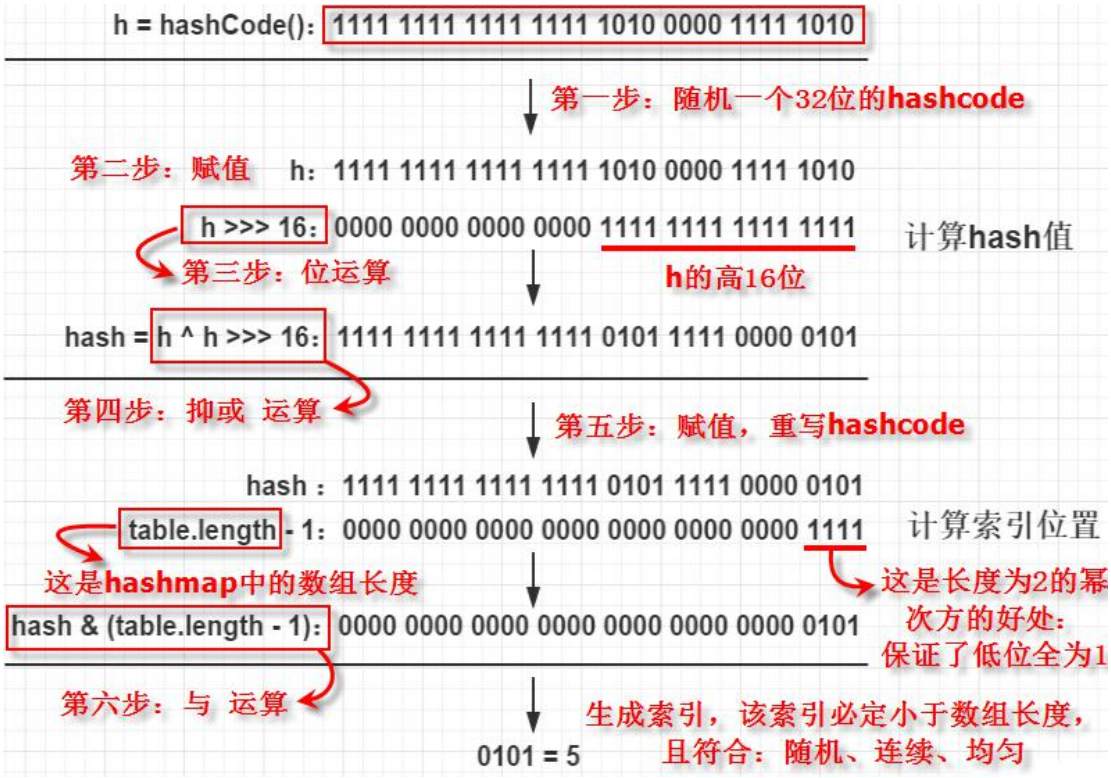
即 h>>>16 就是取 h 的高 16 位；

第四步：h 与 h 的高 16 位进行 **抑或** 运算（相同位为 0，不同位为 1）

第五步：重写 hash，这么做的目的是**让随机数的高 16 位参与运算**

第六步：让 hash 和数组长度-1 做 **与** 运算，结果必定在[0, length-1]区间

可以得出结论：该算法的前提是**桶（数组长度）必须为 2 的幂次方**

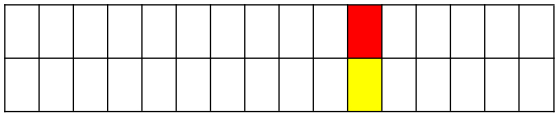

h = hashCode(): 1111 1111 1111 1111 1010 0000 1111 1010
第一步：随机一个32位的hashcode
第二步：赋值
h：1111 1111 1111 1111 1010 0000 1111 1010
h >>> 16：0000 0000 0000 0000 1111 1111 1111 1111
计算hash值
第三步：位运算
h的高16位
hash = h ^ h >>> 16：1111 1111 1111 1111 0101 1111 0000 0101
第四步：抑或 运算
第五步：赋值，重写hashcode
hash ：1111 1111 1111 1111 0101 1111 0000 0101
table.length - 1：0000 0000 0000 0000 0000 0000 0000 1111
计算索引位置
这是hashmap中的数组长度
这是长度为2的幂次方的好处：保证了低位全为1
hash & (table.length - 1)：0000 0000 0000 0000 0000 0000 0000 0101
第六步：与 运算
生成索引，该索引必定小于数组长度，且符合：随机、连续、均匀
0101 = 5

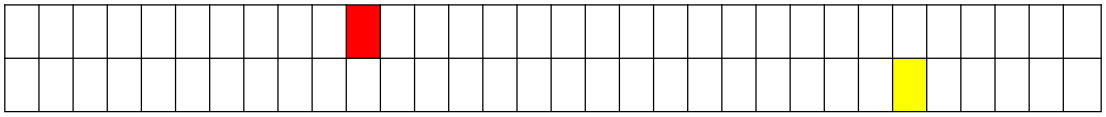
面试题：扩容后，节点重 hash 为什么只可能分布在**原索引位置**与**原索引+oldCap 位置**？

答：如图所示

红：节点 1，hashcode = 0000 0000 0000 0000 0000 1111 0000 1010，index = 10

黄：节点 2，hashcode = 0000 0000 0000 0000 0000 1111 0001 1010，index = 10





计算 1：

老表容量-1：0000 0000 0000 0000 0000 0000 0000 **1111**

节点 1：　0000 0000 0000 0000 0000 1111 0000 **1010**　　0000 0000 0000 0000 0000 0000 0000 **1010**

节点 2：　0000 0000 0000 0000 0000 1111 0001 **1010**　->　0000 0000 0000 0000 0000 0000 0000 **1010**

在oldTable中，节点1和节点2的hashcode不同，但是计算的索引却是相同的
这意味着这两个节点在一个桶中

计算 2：

新表容量-1：0000 0000 0000 0000 0000 0000 000**1** **1111**

节点 1：　0000 0000 0000 0000 0000 1111 0000 **1010**　　0000 0000 0000 0000 0000 0000 000**0** **1010**

节点 2：　0000 0000 0000 0000 0000 1111 0001 **1010**　->　0000 0000 0000 0000 0000 0000 000**1** **1010**

在newTable中，由于低4位（从左数第5位）扩容后为1，所以计算的索引是不同的
且节点2的位置和节点1的位置正好相差10000，即oldTable.length

计算 3：

老表容量:　0000 0000 0000 0000 0000 0000 000**1** 0000

节点 1：　0000 0000 0000 0000 0000 1111 000**0** 1010　　0000 0000 0000 0000 0000 0000 000**0** 0000

节点 2：　0000 0000 0000 0000 0000 1111 000**1** 1010　->　0000 0000 0000 0000 0000 0000 000**1** 0000

既然扩容后只有1位可能影响索引的值，那么优化后的算法如下：
如果扩容位节点1和节点2的值相同，返回相同索引
如果值不同，则值为0的返回原索引，值为1的返回原索引+原数组长度