@来源 HashMap连环18问 (qq.com)

存储结构

HashMap的底层数据结构是什么?

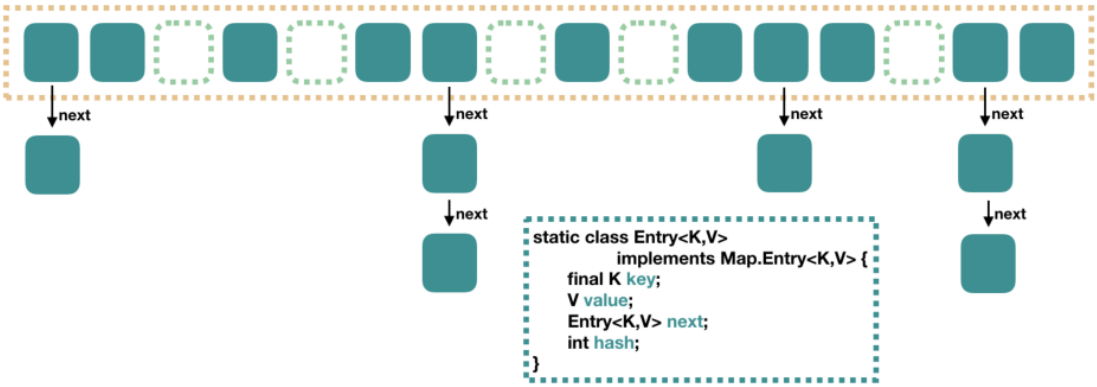
在 JDK1.7 和 JDK1.8 中有所差别：

在 JDK1.7 中，由“数组+链表”组成，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的。

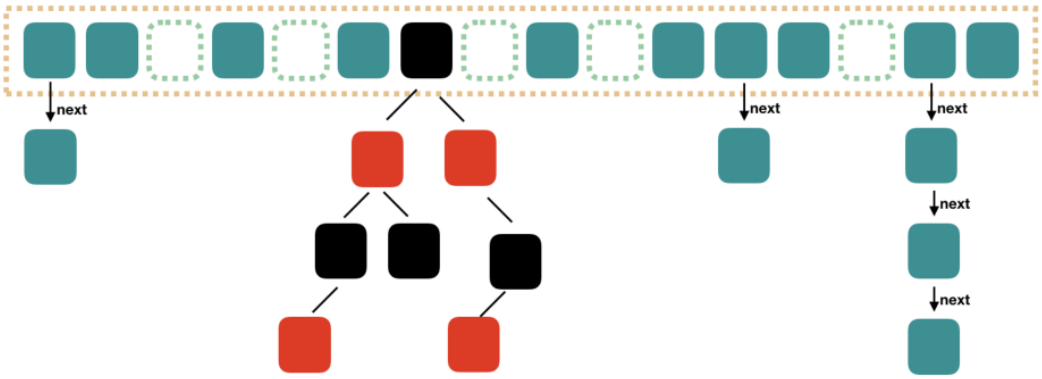
在 JDK1.8 中，由“数组+链表+红黑树”组成。当链表过长，则会严重影响 HashMap 的性能，红黑树搜索时间复杂度是 O(logn)，而链表是糟糕的 O(n)。因此，JDK1.8 对数据结构做了进一步的优化，引入了红黑树，链表和红黑树在达到一定条件会进行转换：

- 当链表长度超过 8 且数据总量大于等于 64 才会转红黑树。
- 将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树，以减少搜索时间。

JDK1.7 HashMap结构



JDK1.8 HashMap结构



为什么在解决hash冲突的时候，不直接用红黑树？而选择先用链表，再转红黑树？

因为红黑树需要进行左旋，右旋，变色这些操作来保持平衡，而单链表不需要。当元素小于 8 个的时候，此时做查询操作，链表结构已经能保证查询性能。当元素大于 8 个的时候， 红黑树搜索时间复杂度是 O(logn)，而链表是 O(n)，此时需要红黑树来加快查询速度，但是新增节点的效率变慢了。

因此，如果一开始就用红黑树结构，元素太少，新增效率又比较慢，无疑这是浪费性能的。

不用红黑树，用二叉查找树可以吗？

可以。但是二叉查找树在特殊情况下会变成一条线性结构（这就跟原来使用链表结构一样了，造成很深的问题），遍历查找会非常慢。

当链表转为红黑树后，什么时候退化为链表？

为6的时候退转为链表。中间有个差值7可以防止链表和树之间频繁的转换。假设一下，如果设计成链表个数超过8则链表转换成树结构，链表个数小于8则树结构转换成链表，如果一个HashMap不停的插入、删除元素，链表个数在8左右徘徊，就会频繁的发生树转链表、链表转树，效率会很低。

为什么链表改为红黑树的阈值是8？

是因为泊松分布，我们来看作者在源码中的注释：

```
Because TreeNodes are about twice the size of regular nodes, we
 use them only when bins contain enough nodes to warrant use
 (see TREEIFY_THRESHOLD). And when they become too small (due
to removal or resizing) they are converted back to plain bins.
 In usages with well-distributed user hashCodes, tree bins are
 rarely used.  Ideally, under random hashCodes, the frequency
of nodes in bins follows a Poisson distribution
 (http://en.wikipedia.org/wiki/Poisson_distribution) with a
 parameter of about 0.5 on average for the default resizing
 threshold of 0.75, although with a large variance because of
 resizing granularity. Ignoring variance, the expected
 occurrences of list size k are (exp(-0.5) pow(0.5, k) /
 factorial(k)). The first values are:
 0:    0.60653066
 1:    0.30326533
 2:    0.07581633
 3:    0.01263606
 4:    0.00157952
 5:    0.00015795
 6:    0.00001316
 7:    0.00000094
 8:    0.00000006
 more: less than 1 in ten million
```

翻译过来大概的意思是：理想情况下使用随机的哈希码，容器中节点分布在 hash 桶中的频率遵循泊松分布，按照泊松分布的计算公式计算出了桶中元素个数和概率的对照表，可以看到链表中元素个数为 8 时的概率已经非常小，再多的就更少了，所以原作者在选择链表元素个数时选择了 8，是根据概率统计而选择的。

字段结构

默认加载因子是多少？为什么是0.75，不是0.6或者0.8?

回答这个问题前，我们来先看下HashMap的默认构造函数：

```
1       int threshold;              // 容纳键值对的最大值
2       final float loadFactor;     // 负载因子
3       int modCount;
4       int size;
```

Node[] table的初始化长度length(默认值是16)，Load factor为负载因子(默认值是0.75)，threshold是HashMap所能容纳键值对的最大值。threshold = length * Load factor。也就是说，在数组定义好长度之后，负载因子越大，所能容纳的键值对个数越多。

默认的loadFactor是0.75，0.75是对空间和时间效率的一个平衡选择，一般不要修改，除非在时间和空间比较特殊的情况下：

- 如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值 。
- 相反，如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1。

我们来追溯下作者在源码中的注释（JDK1.7）：

```
1 As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher
  values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the
  HashMap class, including get and put). The expected number of entries in the map and its load factor should
  be taken into account when setting its initial capacity, so as to minimize the number of rehash operations.
  If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash
  operations will ever occur.
```

翻译过来大概的意思是：作为一般规则，默认负载因子（0.75）在时间和空间成本上提供了很好的折衷。较高的值会降低空间开销，但提高查找成本（体现在大多数的HashMap类的操作，包括get和put）。设置初始大小时，应该考虑预计的entry数在map及其负载系数，并且尽量减少rehash操作的次数。如果初始容量大于最大条目数除以负载因子，rehash操作将不会发生。

索引计算

HashMap中key的存储索引是怎么计算的?

首先根据key的值计算出hashcode的值，然后根据hashcode计算出hash值，最后通过hash&（length-1）计算得到存储的位置。看看源码的实现：

jdk1.7 索引计算

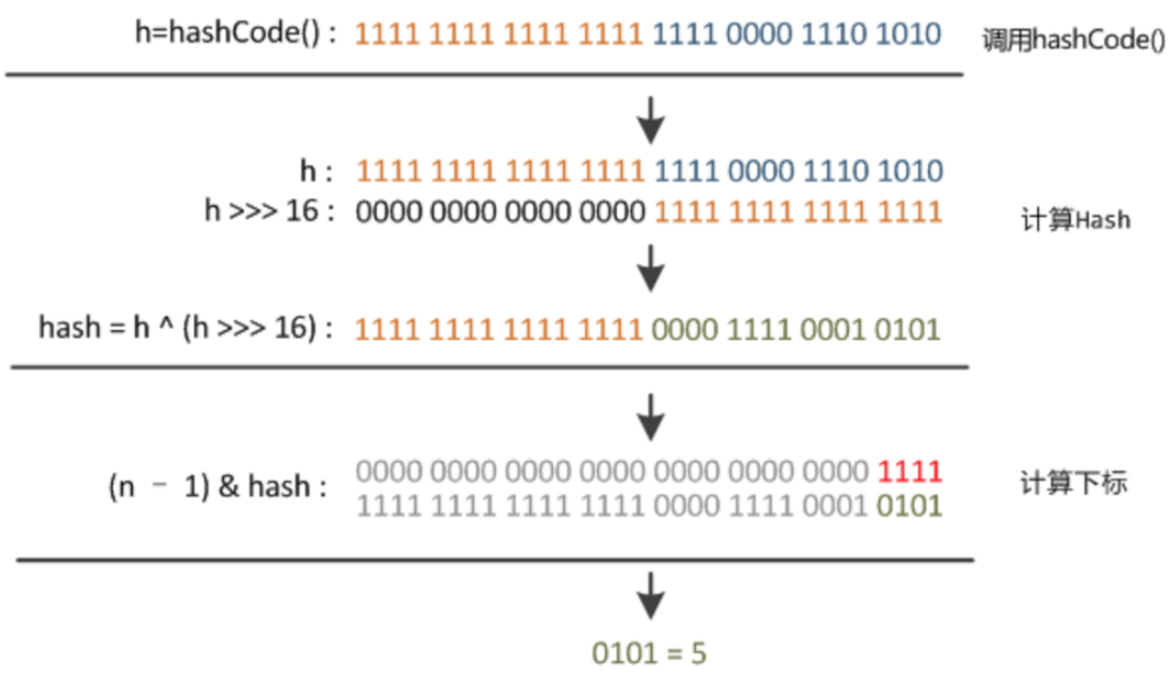
```
// jdk1.7
方法一：
static int hash(int h) {
    int h = hashSeed;
        if (0 != h && k instanceof String) {
            return sun.misc.Hashing.stringHash32((String) k);
        }

    h ^= k.hashCode(); // 为第一步：取hashCode值
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
方法二：
static int indexFor(int h, int length) {  //jdk1.7的源码，jdk1.8没有这个方法，但实现原理一样
    return h & (length-1);  //第三步：取模运算
}
```

jdk1.8 索引计算

```
// jdk1.8
static final int hash(Object key) {
      int h;
      return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
     /*
      h = key.hashCode() 为第一步：取hashCode值
      h ^ (h >>> 16)  为第二步：高位参与运算
     */
}
```

这里的 Hash 算法本质上就是三步：**取key的 hashCode 值、根据 hashcode 计算出hash值、通过取模计算下标**。其中，JDK1.7和1.8的不同之处，就在于第二步。我们来看下详细过程，以JDK1.8为例，n为table的长度：



扩展出以下几个问题

JDK1.8为什么要hashcode异或其右移十六位的值?

因为在JDK 1.7 中扰动了 4 次，计算 hash 值的性能会稍差一点点。从速度、功效、质量来考虑，JDK1.8优化了高位运算的算法，通过hashCode()的高16位异或低16位实现：(h = k.hashCode()) ^ (h >>> 16)。这么做可以在数组 table 的 length 比较小的时候，也能保证考虑到高低Bit都参与到Hash的计算中，同时不会有太大的开销。

为什么hash值要与length-1相与?

- 把 hash 值对数组长度取模运算，模运算的消耗很大，没有位运算快。
- 当 length 总是 2 的n次方时，h& (length-1) 运算等价于对length取模，也就是 h%length，但是 & 比 % 具有更高的效率。

HashMap数组的长度为什么是2的幂次方?

这样做效果上等同于取模，在速度、效率上比直接取模要快得多。除此之外，2 的 N 次幂有助于减少碰撞的几率。如果 length 为2的幂次方，则 length-1 转化为二进制必定是11111……的形式，在与h的二进制与操作效率会非常的快，而且空间不浪费。我们来举个例子，看下图：

length=16			
h	length-1	h & (length - 1)	result
4	15	0100 & 1111 = 0100	4
5	15	0101 & 1111 = 0101	5
6	15	0110 & 1111 = 0110	6
7	15	0111 & 1111 = 0111	7

length=15			
h	length-1	h & (length - 1)	result
4	14	0100 & 1110 = 0100	4
5	14	0101 & 1110 = 0100	4
6	14	0110 & 1110 = 0110	6
7	14	0111 & 1110 = 0110	6

当n=15时，6 和 7 的结果一样，这样表示他们在 table 存储的位置是相同的，也就是产生了碰撞，6、7就会在一个位置形成链表，4和5的结果也是一样，这样就会导致查询速度降低。

如果我们进一步分析，还会发现空间浪费非常大，以 length=15 为例，在 1、3、5、7、9、11、13、15 这八处没有存放数据。因为hash值在与14（即 1110）进行&运算时，得到的结果最后一位永远都是0，即 0001、0011、0101、0111、1001、1011、1101、1111位置处是不可能存储数据的。

补充数组容量计算的小奥秘

HashMap 构造函数允许用户传入的容量不是 2 的 n 次方，因为它可以自动地将传入的容量转换为 2 的 n 次方。会取大于或等于这个数的 且最近的2次幂作为 table 数组的初始容量，使用 **tableSizeFor(int)**方法，如 tableSizeFor(10) = 16（2 的 4 次幂），tableSizeFor(20) = 32（2 的 5 次幂），也就是说 table 数组的长度总是 2 的次幂。JDK1.8 源码如下：

```java
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;
    }
/*
解释：位或( | )
int n = cap - 1;  让cap-1再赋值给n的目的是另找到的目标值大于或等于原值。例如二进制1000，十进制数值为8。如果不对它减1而直接操作，将得到答案10000，即16。显然不是结果。减1后二进制为111，再进行操作则会得到原来的数值1000，即8。
*/
```

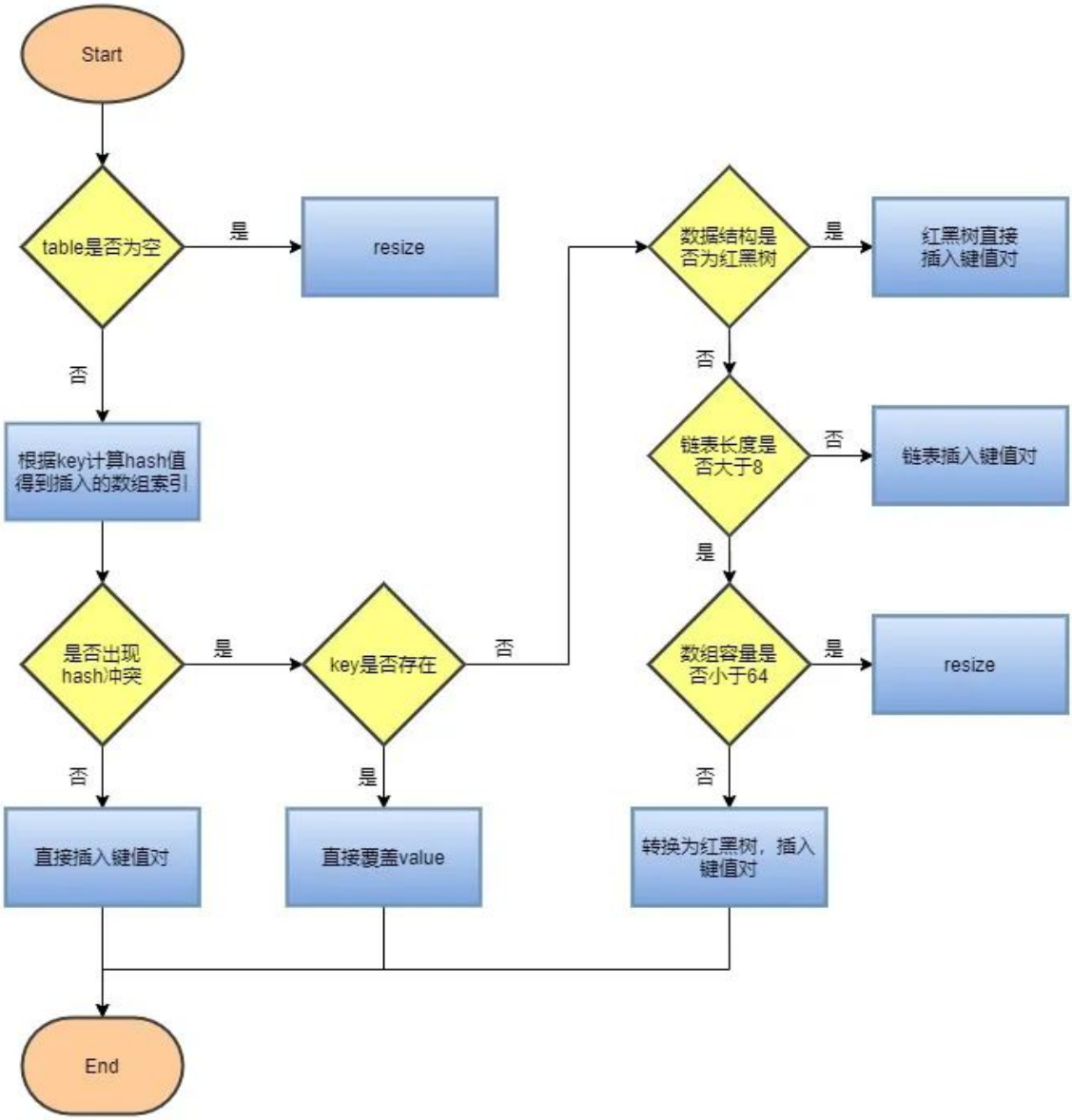
put方法

HashMap的put方法流程?

简要流程如下：

1. 首先根据 key 的值计算 hash 值，找到该元素在数组中存储的下标；
2. 如果数组是空的，则调用 resize 进行初始化；
3. 如果没有哈希冲突直接放在对应的数组下标里；
4. 如果冲突了，且 key 已经存在，就覆盖掉 value；
5. 如果冲突后，发现该节点是红黑树，就将这个节点挂在树上；
6. 如果冲突后是链表，判断该链表是否大于 8 ，如果大于 8 并且数组容量小于 64，就进行扩容；
7. 如果链表长度大于 8 并且数组的容量大于等于 64，则将这个结构转换为红黑树；
8. 否则，链表插入键值对，若 key 存在，就覆盖掉 value。

hashmap之put方法 (JDK1.8)



详细分析，见 JDK1.8 的 put 方法源码:

```java
public V put(K key, V value) {
     // 对key的hashCode()做hash
        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;
        // 步骤1：tab为空则创建
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        // 步骤2：计算index，并对null做处理
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            // 步骤3：节点key存在，直接覆盖value
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            // 步骤4：判断该链为红黑树
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);     // 步骤5：该链为链表
            else {
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        //链表长度大于8转换为红黑树进行处理
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    // key已经存在直接覆盖value
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null &&
key.equals(k))))        break;
                    p = e;
                }
            }
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        // 步骤6：超过最大容量 就扩容
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }

// 第31行treeifyBin方法部分代码
final void treeifyBin(Node<K,V>[] tab, int hash) {
        int n, index; Node<K,V> e;
        // static final int MIN_TREEIFY_CAPACITY = 64;
        // 如果大于8但是数组容量小于64，就进行扩容
        if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
            resize();

    }


```

```
68
```

JDK1.7和1.8的put方法区别是什么?

区别在两处:

解决哈希冲突时，JDK1.7 只使用链表，JDK1.8 使用链表+红黑树，当满足一定条件，链表会转换为红黑树。

链表插入元素时，JDK1.7 使用头插法插入元素，在多线程的环境下有可能导致环形链表的出现，扩容的时候会导致死循环。因此，JDK1.8使用尾插法插入元素，在扩容时会保持链表元素原本的顺序，就不会出现链表成环的问题了，但JDK1.8 的 HashMap 仍然是线程不安全的，具体原因会在另一篇文章分析。

扩容机制

HashMap的扩容方式

Hashmap 在容量超过负载因子所定义的容量之后，就会扩容。Java 里的数组是无法自动扩容的，方法是将 Hashmap 的大小扩大为原来数组的两倍，并将原来的对象放入新的数组中。

那扩容的具体步骤是什么？让我们看看源码。

先来看下 JDK1.7 的代码:

```
1 void resize(int newCapacity) {   //传入新的容量
2         Entry[] oldTable = table;    //引用扩容前的Entry数组
3         int oldCapacity = oldTable.length;
4         if (oldCapacity == MAXIMUM_CAPACITY) {  //扩容前的数组大小如果已经达到最大(2^30)了
5             threshold = Integer.MAX_VALUE; //修改阈值为int的最大值(2^31-1)，这样以后就不会扩容
6 了           return;
7         }
8
9         Entry[] newTable = new Entry[newCapacity];  //初始化一个新的Entry数组
10        transfer(newTable);                         //！！将数据转移到新的Entry数组里
11        table = newTable;                           //HashMap的table属性引用新的Entry数组
12        threshold = (int)(newCapacity * loadFactor);//修改阈值
13    }
```

这里就是使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将原有Entry数组的元素拷贝到新的Entry数组里。

```
1 void transfer(Entry[] newTable) {
2         Entry[] src = table;                   //src引用了旧的Entry数组
3         int newCapacity = newTable.length;
4         for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组
5             Entry<K,V> e = src[j];             //取得旧Entry数组的每个元素
6             if (e != null) {
7                 src[j] = null;//释放旧Entry数组的对象引用（for循环后，旧的Entry数组不再引用任何对
8 象）               do {
9                     Entry<K,V> next = e.next;
10                    int i = indexFor(e.hash, newCapacity); //！！重新计算每个元素在数组中的位置
11                    e.next = newTable[i]; //标记[1]
12                    newTable[i] = e;      //将元素放在数组上
13                    e = next;             //访问下一个Entry链上的元素
14                } while (e != null);
15            }
16        }
17    }
```

newTable[i] 的引用赋给了 e.next，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到 Entry 链的尾部(如果发生了 hash 冲突的话）。

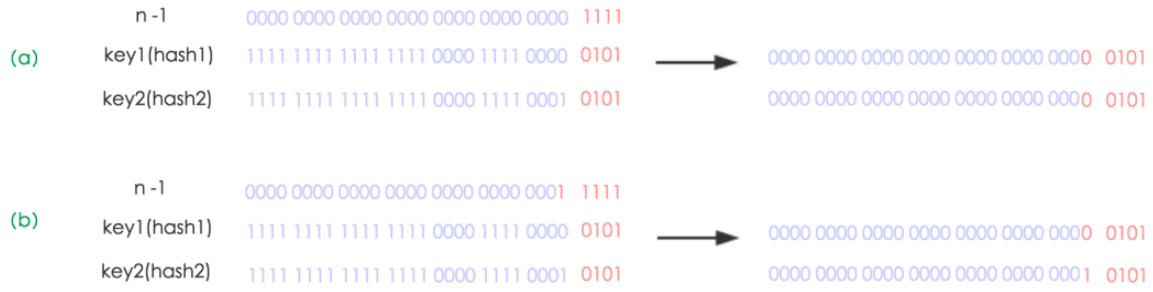
JDK1.8的优化

扩容在JDK1.8中有什么不一样?

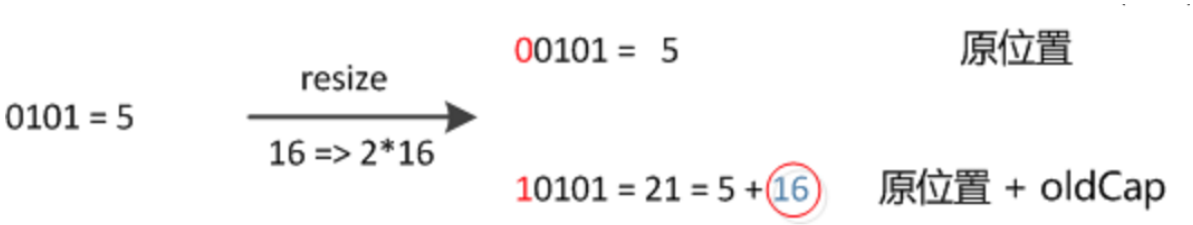
JDK1.8做了两处优化：

1. resize 之后，元素的位置在原来的位置，或者原来的位置 +oldCap (原来哈希表的长度）。不需要像 JDK1.7 的实现那样重新计算hash，只需要看看原来的 hash 值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引 + oldCap ”。这个设计非常的巧妙，省去了重新计算 hash 值的时间。

   如下图所示，n 为 table 的长度，图（a）表示扩容前的 key1 和 key2 两种 key 确定索引位置的示例，图（b）表示扩容后 key1 和key2 两种 key 确定索引位置的示例，其中 hash1 是 key1 对应的哈希与高位运算结果。

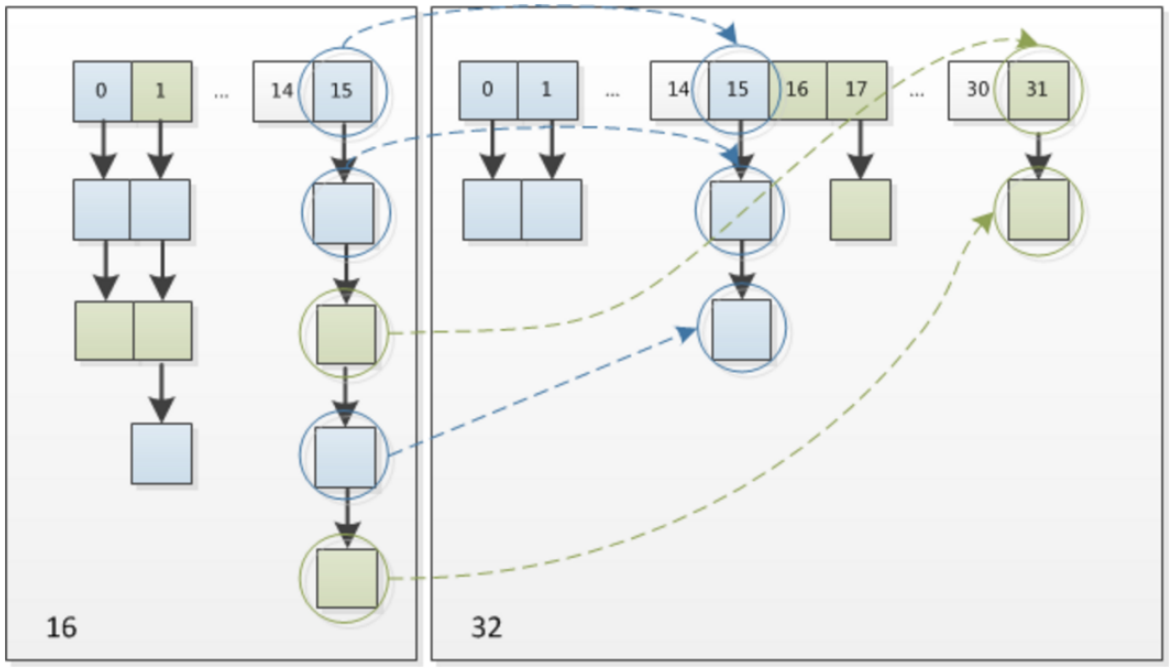


元素在重新计算 hash 之后，因为 n 变为 2倍，那么 n-1 的 mask 范围在高位多 1 bit(红色)，因此新的 index就会发生这样的变化：



2. JDK1.7 中 rehash 的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置（头插法）。JDK1.8 不会倒置，使用尾插法。

下图为 16扩充为 32 的 resize 示意图：



感兴趣的小伙伴可以看下 JDK1.8 的 resize 源码：

```java
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) {
            // 超过最大值就不再扩充了，就只好随你碰撞去吧
            if (oldCap >= MAXIMUM_CAPACITY) {
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }
            // 没超过最大值，就扩充为原来的2倍
            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);
        }
        // 计算新的resize上限
        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) {
            // 把每个bucket都移动到新的buckets中
            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 { // 链表优化重hash的代码块
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
                        Node<K,V> next;
                        do {
                            next = e.next;
                            // 原索引
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            // 原索引+oldCap
                            else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        // 原索引放到bucket里
                        if (loTail != null) {
                            loTail.next = null;
```

```
                        newTab[j] = loHead;
                    }
                    // 原索引+oldCap放到bucket里
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
        return newTab;
    }
```

其他

还知道哪些hash算法?

Hash函数是指把一个大范围映射到一个小范围，目的往往是为了节省空间，使得数据容易保存。比较出名的有MurmurHash、MD4、MD5等等。

key可以作为Null吗?

可以，key 为 Null 的时候，hash算法最后的值以0来计算，也就是放在数组的第一个位置。

```
@Test
public void myTest(){
    HashMap<Integer, Integer> map = new HashMap<>();
    map.put(null,10);
    System.out.println(map.get(null));
    System.out.println(map.get(0));
    map.put(0,20);
    System.out.println(map.get(null));
    System.out.println(map.get(0));
}
```

```
✔ Tests passed: 1 of 1 test – 55 ms
F:\java\jdk\bin\java.exe ...
10
null
10
20
```

一般用什么作为HashMap的key?

一般用Integer、String 这种不可变类当 HashMap 当 key，而且 String 最为常用。

- 因为字符串是不可变的，所以在它创建的时候 hashcode 就被缓存了，不需要重新计算。这就是 HashMap 中的键往往都使用字符串的原因。

- 因为获取对象的时候要用到 equals() 和 hashCode() 方法，那么键对象正确的重写这两个方法是非常重要的,这些类已经很规范的重写了 hashCode() 以及 equals() 方法。

用可变类当HashMap的key有什么问题?

hashcode 可能发生改变，导致 put 进去的值，无法 get 出。如下所示：

hashcode案例

```
1 HashMap<List<String>, Object> changeMap = new HashMap<>();
2 List<String> list = new ArrayList<>();
3 list.add("hello");
4 Object objectValue = new Object();
5 changeMap.put(list, objectValue);
6 System.out.println(changeMap.get(list));
7 list.add("hello world");//hashcode发生了改变
8 System.out.println(changeMap.get(list));
```

输出值如下

hashcode案例续

```
1 java.lang.Object@74a14482
2 null
```