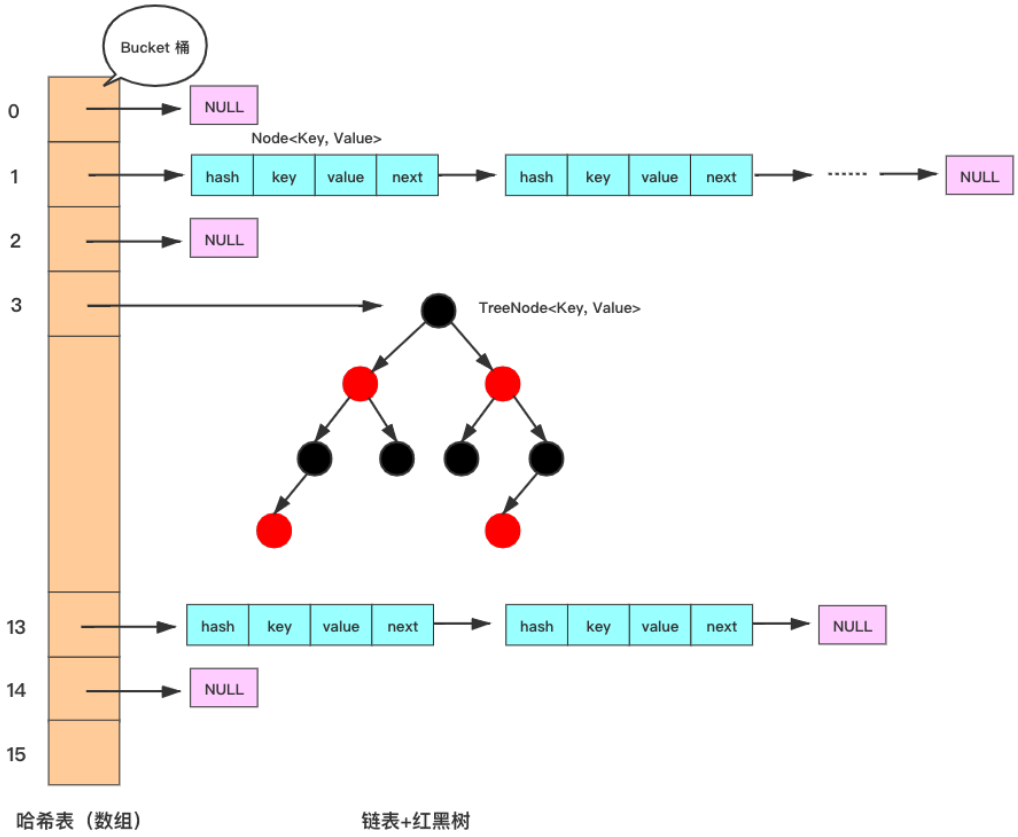
> 在《HashMap（JDK7）》中详细解析了HashMap在JDK7中的实现原理，主要围绕其put、get、resize、transfer等方法，本文将继续解析HashMap在JDK8中的具体实现，首先也将从put、get、resize等方法出发，着重解析HashMap在JDK7和JDK8的具体区别，最后回答并解析一些常见的HashMap问题。

一、HashMap在JDK8中的结构

HashMap在JDK7或者JDK8中采用的基本存储结构都是**数组+链表**形式，可能有人会提出疑问，HashMap在JDK8中不是**数组+链表+红黑树**吗？本文的回答**是**。至于为什么JDK8在一定条件下将链表转换为红黑树，我相信很多人都会回答：**为了提高查询效率**。基本答案可以说是这样的，JDK7中的HashMap随着Entry节点增多，哈希碰撞的概率在慢慢变大，这就直接导致哈希表中的单链表越来越长，这就大大降低了HashMap的查询能力，且时间复杂度可能会退化到O(n)。针对这种情况。JDK8做出了优化，就是在一定的条件下，链表会被转换为红黑树，提升查询小。



在上面的示意图可以看出，与JDK7的最大区别就是哈希表中不仅有链表，还可能存在红黑树这种结构。在这提出两个问题：

- 何时链表会转换成红黑树?
- 为什么需要转换为红黑树?

二、HashMap在JDK8中的具体实现

2.1 理解HashMap的成员变量

JDK8中的HashMap也有多个成员属性，如下所示：

```
1 // 哈希表默认的初始化容量
2 static final int DEFAULT_INITIAL_CAPAC
3 // 哈希表最大的容量
4 static final int MAXIMUM_CAPACITY = 1
5 // 默认的负载因子
6 static final float DEFAULT_LOAD_FACTOR
```

```
// 链表可能转换为红黑树的基本阀值（链表长度>
static final int TREEIFY_THRESHOLD = 8
// 哈希表扩容后，如果发现红黑树节点数小于6，
 static final int UNTREEIFY_THRESHOLD
 // 链表转换为红黑树的另一个条件，哈希表长度
 static final int MIN_TREEIFY_CAPACITY
 // 哈希表
 transient Node<K,V>[] table;
 // Node<K,V>的Set集合
 transient Set<Map.Entry<K,V>> entrySe
 // Node<K,V>节点个数
 transient int size;
 // map内元素的个数的修改次数
 transient int modCount;
 // 扩容阀值，当size >= threshold时候，有
 int threshold;
 // 自定义负载因子
 final float loadFactor;
```

由于《HashMap（JDK7）》已经详细介绍了上述部分成员属性，这里仅仅介绍一些JDK8特有的属性：

- TREEIFY_THRESHOLD：链表转换为红黑树的阀值，当链表长度大于等于8的时候，链表可能会被转换为红黑树，这里之所以说是可能，是因为还要满足另外一个条件：哈希表长度大于等于64，否则哈希表会尝试扩容。
- UNTREEIFY_THRESHOLD：红黑树退化成链表的阀值，当红黑树节点小于等于6的时候，红黑树会转换成普通的链表。
- MIN_TREEIFY_CAPACITY：链表转换为红黑树的第二个条件，哈希表长度大于等于64的时候，且链表长度达到8才会转换为红黑树，否则将会扩容。

在JDK7中，Key和Value的存储是利用Entry节点，JDK8中使用的是Node节点，前者是JDK7中HashMap的内部类，实现了Map.Entry接口，后者是JDK8中HashMap的内部类，实现的也是Map.Entry接口，两者的成员属性也是一致的，具体代码比较如下：

- JDK7中Entry节点

```
static class Entry<K,V> implements Map
    final K key;
    V value;
    Entry<K,V> next;
    int hash;
    // 后续代码省略
}
```

- JDK8中的Node节点

```
static class Node<K,V> implements Map.
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
}
```

两者其实是一致的，这里不做过多解释。

2.2 理解HashMap的构造方法

相比于JDK7，JDK8的HashMap的构造方法和JDK7几乎一致，这里需要说一点区别，JDK7中的HashMap的构造方法如下所示：

```
// 该构造方法对初始化容量和负载因子进行了一个
// 然后将传入的负载因子复制给了loadFactor成
// 将初始化容量赋值给了扩容阀值（扩容临界数值
public HashMap(int initialCapacity, fl
    if (initialCapacity < 0)
        throw new IllegalArgumentExcep

    if (initialCapacity > MAXIMUM_CAPA
        initialCapacity = MAXIMUM_CAPA
     if (loadFactor <= 0 || Float.isNa
         throw new IllegalArgumentExce
```

```
12
13
14      this.loadFactor = loadFactor;
15      threshold = initialCapacity;
16      init();
17  }
```

JDK8中钙构造方法如下所示：

```
1  public HashMap(int initialCapacity, fl
2      if (initialCapacity < 0)
3          throw new IllegalArgumentExcep
4
5      if (initialCapacity > MAXIMUM_CAPA
6          initialCapacity = MAXIMUM_CAPA
7      if (loadFactor <= 0 || Float.isNaN
8          throw new IllegalArgumentExcep
9
10     this.loadFactor = loadFactor;
11     this.threshold = tableSizeFor(ini
12 }
```

相比较发现，JDK8中使用了tableSizeFor()方法，其实就是计算出了最接近initialCapacity的且大于initialCapacity的2的N次幂的值，比如传入的初始化容量为27，那么最接近27且大于27的2的N次幂是32，此时N = 5，而在JDK7中是第一次put中完成的，当然对于threshold的值，未初始化的时候都是承载的是initialCapacity，后续都会重新计算为 capacity * loadFactor。

2.3 理解HashMap的put方法

相比于JDK7，JDK8的put方法貌似要复杂很多，咋一眼看上去有点惶恐，不过没关系，我们一起一行一行来分析，基本上也能啃下这块硬骨头。

```
1  public V put(K key, V value) {
2      return putVal(hash(key), key,
```

```
}

/**
* 参数解析:
* onlyIfAbsent: 如果为true, 那么将不
* evict: 该参数用于LinkedHashMap, 这
*/
final V putVal(int hash, K key,
               boolean evict) {
    // tab是该方法中内部数组引用, p是
    Node<K,V>[] tab; Node<K,V> p
    // 第一次put的时候, table未初始
    if ((tab = table) == null ||
        // 这里实现扩容, 具体逻辑稍后
        n = (tab = resize()).len
    // 获取指定key的对应下标的首节点
    if ((p = tab[i = (n - 1) & h
        tab[i] = newNode(hash, k
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key
            // p此时指向的是不为nul
            // 如果首节点的key和要存
            e = p;
        else if (p instanceof Tr
            // 如果首节点是红黑树的,
            e = ((TreeNode<K,V>)
        else {
```

```
32              // 能走到这里，说明首节
33              // 开始需要遍历链表，如
34              // 如果不存在，则在末端
35              // 尝试转换成红黑树。注
36              // 当哈希表长度是否到达
37              for (int binCount =
38                  // 当binCount =
39                  // 其他情况及链表中
40                  if ((e = p.next)
41                      // 新建一个Noc
42                      p.next = new
43                      // 判断遍历次
44                      if (binCount
45                          // “尝试”
46                          treeifyB
47                      break;
48                  }
49                  // 只要没走到上面哪
50                  if (e.hash == ha
51                      ((k = e.key)
52                      break;
53                  // 将正在遍历的节点
54                  p = e;
55              }
56          }
57          // 首节点或者链表中替换旧值为
58          if (e != null) { // exis
59              V oldValue = e.value
60              if (!onlyIfAbsent ||
```

```
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            // 如果满足扩容条件，旧扩容
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

分析上面的代码，put基本流程可以总结如下：

第一步：检查哈希表是否为空，如果为空，就进行扩容。

第二步：通过key的hash值以及哈希表长度来确定当前key在哈希表中的索引下标，并获取到同一下标下的链表首节点或者红黑树的根节点。

第三步：图个获取到首节点（或根节点）为null，说明当前哈希表的位置上没有任何链表或者红黑树，此时将key和value封装成Node节点对象存储到首节点位置。

第四步：如果获取到首节点（或根节点）不为null，说明当前哈希表的位置上有链表或者红黑树，这是进一步判断当前key是否和首节点的Node中的key一致，如果一致，则将首节点的值替换成新值，否则进行下一步。

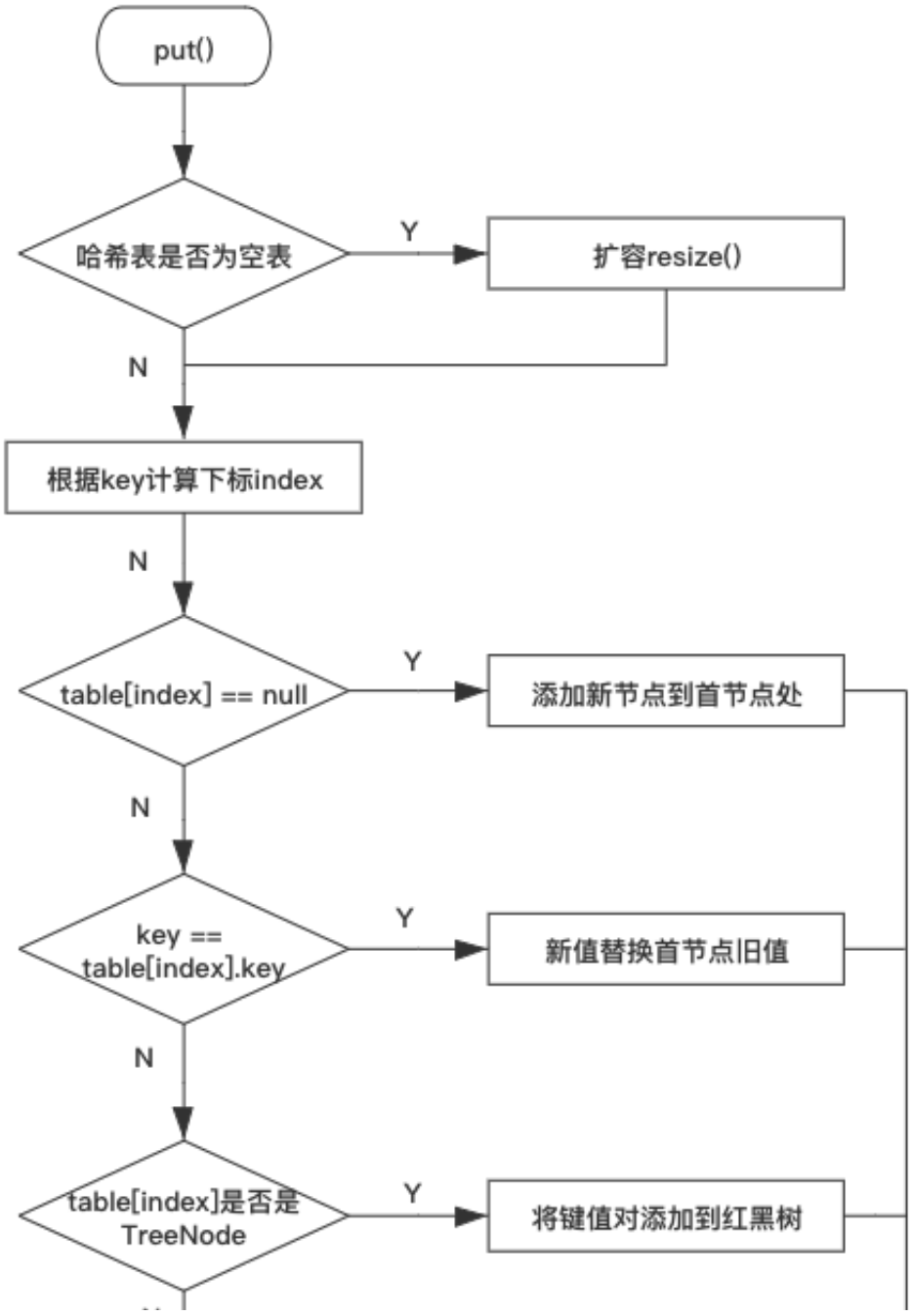
第五步：如果当前需要存储的key和首节点的key不一致，那么进一步判断当前节点是否为TreeNode类型，也就是红黑树类型，如果是红黑树类型，则做树节点插入，否则进行第六步。

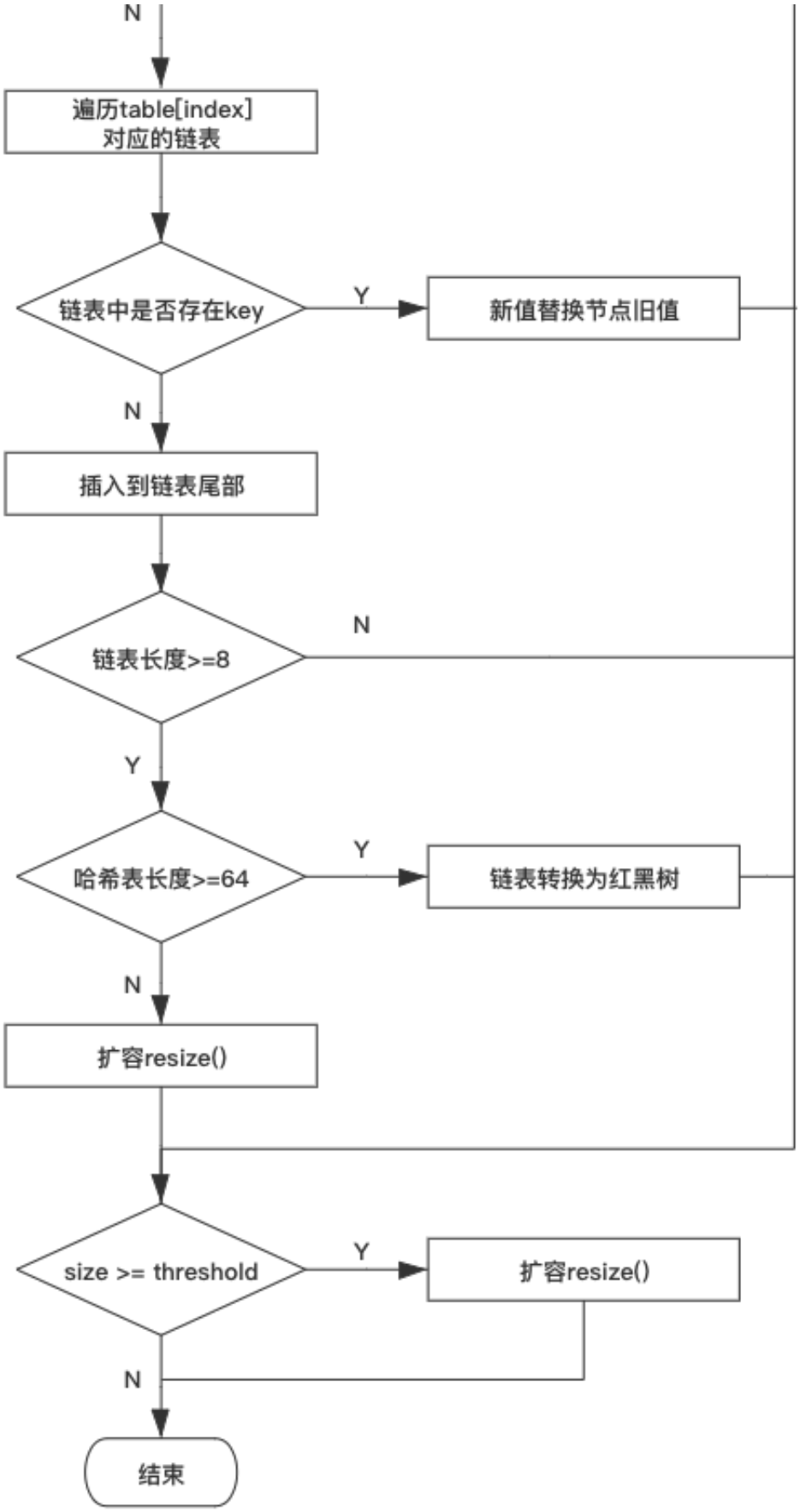
第六步：如果需要存储的key和首节点的key不一致，且首节点不是TreeNode类型，那么就到这第六步，第六步主要就是遍历链表，如果链表中包含相同key的Node对象，那么就做值的替换，否则将新建Node对象并插入到链表的结尾处。完成结尾处插入之后，就会根据条件`binCount >= 7`来判断是否尝试将链表转换为红黑树，这里之所以是遍历次数大于等于7，这是因为从链表的第二个节点开始的。

第七步：最后判断K-V对数是否超过了扩容阀值threshold，超过了则扩容。

特别说明：上述第六步之所以说是尝试转换成红黑树，这是因为在转换逻辑里面对哈希表的长度还进了校验，只有哈希表长度大于等于64才转换，否则进行扩容。这里面的原理将在转换红给树的代码里面详细分析。

为了方便理解put的整个流程，将上述文字描述转换为流程图：




N
遍历table[index]
对应的链表
链表中是否存在key
Y
新值替换节点旧值
N
插入到链表尾部
链表长度>=8
N
Y
哈希表长度>=64
Y
链表转换为红黑树
N
扩容resize()
size >= threshold
Y
扩容resize()
N
结束

从上面的流程图就可以回答文章开始的第一个疑问：何时链表会转换成红给树？转换成红黑树条件是发生**哈希冲突**，且新节点下标所在位置有链表，且新节点必须插入到链表尾部且插入后长度正好达到8，并且哈希表长度要大于等于64，这些条件都满足才会有链表转换成红黑树的操作。如果没有发生哈希冲突，就不会转换，也就是新节点直接存如哈希表桶内。如果新节点替换了首节点或其他节点，那么也不会发生转换。如果插入到链表尾部，且链表长度达到8，且哈希表长度达到64才会发生转换操作。

在put过程中会有将链表转换为红黑树的过程，具体转换代码如下所示：

```
final void treeifyBin(Node<K,V>[]
    int n, index; Node<K,V> e;
    // 在这里判断是否满足扩容条件，如果
    if (tab == null || (n = tab.l
        resize();
    else if ((e = tab[index = (n
        // 到这里开始遍历链表
        TreeNode<K,V> hd = null,
        do {
            // 将链表中的节点Node类
            TreeNode<K,V> p = re
            if (tl == null)
                hd = p;
            else {
                p.prev = tl;
                tl.next = p;
            }
            tl = p;
        } while ((e = e.next) !=
        // TreeNode链表转换成为红黑
```

```
21            if ((tab[index] = hd) !=
22                hd.treeify(tab);
23        }
24    }
```

这里暂不分析链表是如何转换成为红黑树的，红黑树这种数据结构其内容还是比较繁琐的，要求读者具有红黑树数据基础，过多分析将影响本文对HashMap的实现原理分析，这里不过多阐述。

那么这里回答一下第二个疑问：为什么需要转换为红黑树?
HashMap在JDK8之后引入了红黑树的概念，表示若哈希表的桶中链表元素超过8 时（默认哈希表长度不小于64），会自动转化为红给树；若桶中元素小于等于6时，树结构还原成链表形式。红给树的平均查找长度时log(n)，长度为8，查找长度为log(8)=3，链表的平均查找长度为n/2，当长度为8时，平均查找长度为8/2=4，这才有转换成树的必要；
链表长度如果时小于等于6，6/2=3，虽然速度也很快的，但是转化为树结构和生成树的时间并不会太短，以7和8作为平衡点是因为，中间有个差值7可以防止链表和树之间频繁的转换。假设：如果设计成链表个数超过8则联合转换成树结构，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，链表个数载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。
概括起来就是：链表，如果元素小于8个，查询成本高，新值成本低；红给树，如果元素大于8个，查询成本低，新值成本高。

2.4 理解HashMap的resize方法

前一篇文章分析过HashMap载JDK7中的扩容条件：

- K-V对数大于等于扩容阀值，也就是 `size >= threshold` ，并且put过程中要发生哈希碰撞（key为null的碰撞除外），也就是说要存放Entry对象的桶以及存在链表了呢，这个时候才会扩容，否则仅满足`size >= threshold` 是不会发生扩容的。
- put键为null的K-V对俄时候永远不会发生扩容。

那么在本文中分析了HashMap在JDK8中的put方法，基本可以总结出HashMap在JDK8中的扩容条件

- 哈希表为null或者长度为0。
- 哈希表中存储的K-V对数超过了threshold。
- 链表中长度超过了8，并且哈希表长度到达63，此时也会发生扩容。

对比JDK7和JDK8，HashMap的扩容条件也有些差异，这也是两个JDK版本中HashMap的区别之一了。
再者，HashMap在JDK7和JDK8中的扩容机制其实也是有区别的，在JDK8中，HashMap的扩容机制有了改进，设计的非常巧妙，避免了JDK7中的“再哈希”，提高了扩容性能。接下来，我们将使用示意图的形式将展示HashMap在JDK7和JDK8中的扩容基本算法，方便理解两个版本之间的差异。

2.4.1 重新理解JDK7 HashMap的resize方法

首先我们将上一篇文章对resize方法（JDK7）进行分析的源码注释拷贝过来，扩容代码如下：

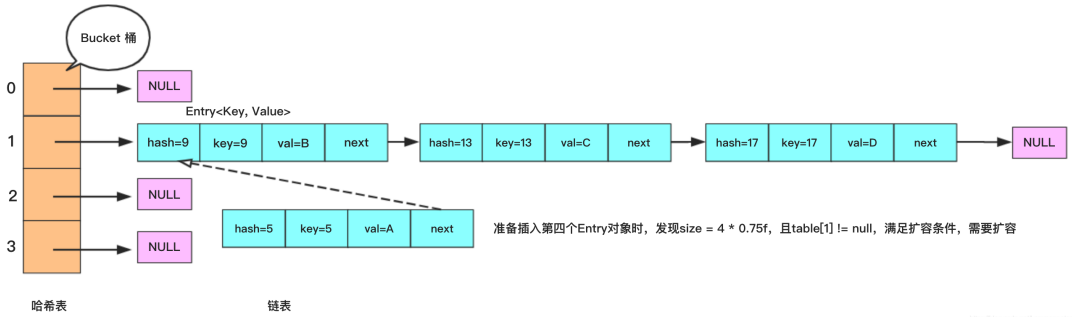
```
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.le
    // 如果老数组的容量达到了最大，那么
    if (oldCapacity == MAXIMUM_CA
        threshold = Integer.MAX_V
        return;
    }

     // 创建一个新的哈希表，容量是原来
     Entry[] newTable = new Entry
     // 将重新计算所有Entry对象的下标
     transfer(newTable, initHashS
     table = newTable;
```

```
15      // 重新计算新的扩容阈值threshol
16      threshold = (int)Math.min(ne
17  }
18
19  void transfer(Entry[] newTable,
20      int newCapacity = newTable.l
21      // 遍历老的table，遍历到每一个bu
22      for (Entry<K,V> e : table) {
23          while(null != e) {
24              Entry<K,V> next = e.
25              // 重新计算hash
26              if (rehash) {
27                  e.hash = null ==
28              }
29              // 重新计算下标
30              int i = indexFor(e.h
31              // 头节点插入链表
32              e.next = newTable[i]
33              newTable[i] = e;
34              // 继续原链表的下一个节
35              e = next;
36          }
37      }
38  }
```

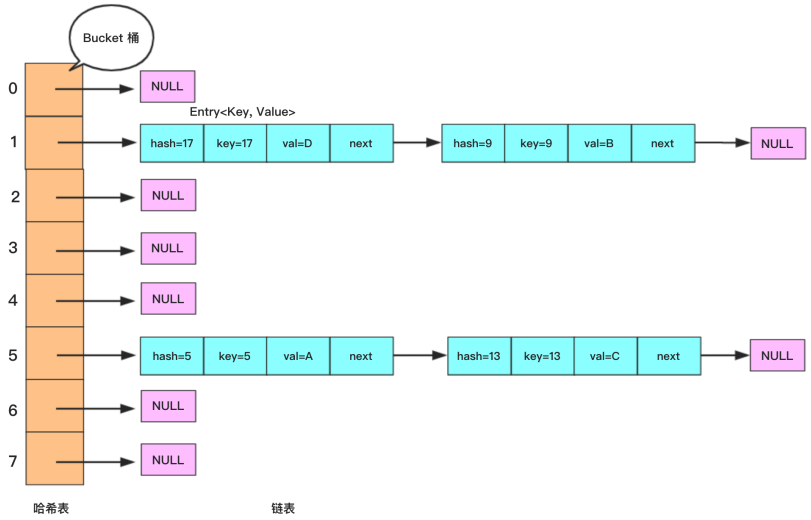
上面的代码展示了HashMap在JDK7中的扩容和数据迁移，基本过程时先创建一个容量时原理数组容量的2倍哈希表，然后遍历老哈希表，将每个桶内链表都遍历一遍，然后对每个key重新进行hash计算，然后将其插入到新的哈希表中，直到所有的Entry对象都转移到了新的哈希表中为止。

这里我们一起举例子来展示一下HashMap在JDK7中的具体扩容过程。这里**假设**

获取下标的算法key的hash值对哈希表长度进行取模运算（也可以时对 lenght -1 进行取模运算，这里为李方便直接对长度取模运算），且所有的可以都是有正整数来表示（因为正整数的hash值就是正整数本身）。假设哈希表长度为4，有4个Entry对象，他们key分别是5，9，13，17，他们的值可以是任何同一类型的值，这里使用大写字母A，B，C，D来表示。4个Entry插入顺序按照key来排序就是17，13，9，5，按照本例中假设的下标算法，这四个Entry对象都会发生了哈希冲突，且都位于下标为1的bucket桶中，如下图所示：



JDK中向链表中添加节点是采用的头插入法，哈希表长度为4，他的 threshold = 3 ，当添加第四个节点的时候，发现达到了扩容的条件，那么就需要进行扩容，首先是新疆一个容量为8的哈希表，然后将对老哈希表中所有节点再哈希，重新求取所有节点的下标，这里需要注意的是，整个扩容过程是在准备添加 key =5 的Entry节点的时候触发的，那么首先将老哈希表中所有的Entry节点搬迁到新哈希表中，最后再添加 key = 5 的Entry节点到新哈希表中。在新的哈希表中插入Entry节点的顺序就变成了9，13，17，5，最后插入的结果如下图所示：



观察上图可以看出，当原链表中的节点key再出出现哈希冲突的时候，比如会出现**倒挂**现象，这里的key=17和key=9的节点就出现了倒挂现象，这也就是

HashMap内元素是无序的原因了，读者可以细细体会。

2.4.2 重新理解JDK8 HashMap的resize方法

对比JDK7中的HashMap，其实JDK8中，HashMap的扩容机制实现的原理也是类似的，也是遍历链表或者红给树来完成数据的迁移。只不过两者之间的区别是后者不再重现对所有节点进行**再哈希**运算了，而是设计了一个更为巧妙的方式来确定节点的下标，那么到底是如何确定的呢？我们一同通过阅读源码来一探究竟。

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    // 计算老哈希表的容量，如果老哈希表
    int oldCap = (oldTab == null)
    // threshold的值有点特殊，当初始
    // 否则就是capacity * loadFact
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        // 能进入到这里，说明不是初始
        if (oldCap >= MAXIMUM_CA
            threshold = Integer.
            return oldTab;
        }
        else if ((newCap = oldCa
                 oldCap >= DEFAU
            // 新哈希表长度扩容到原
            newThr = oldThr << 1
     }
     else if (oldThr > 0) // init
        // 能进入到这里，说明oldCap
        // 且这个值必然是2的N次幂，
```

```
23          newCap = oldThr;
24      else {                  // zero
25          // 如果上述条件都不满足，那么
26          newCap = DEFAULT_INITIAL
27          newThr = (int)(DEFAULT_L
28      }
29      // 下面的if是为零计算扩容阀值，因
30      if (newThr == 0) {
31          float ft = (float)newCap
32          newThr = (newCap < MAXIM
33                    (int)ft : Inte
34      }
35      // 到这里就完成了扩容后的容量和扩
36      threshold = newThr;
37      // 创建新的哈希表，容量为newCap,
38      @SuppressWarnings({"rawtypes
39      Node<K,V>[] newTab = (Node<K
40      // 将扩容后的哈希表赋值给table
41      table = newTab;
42      if (oldTab != null) {
43          // 能进入这里，说明不是初始化
44          for (int j = 0; j < oldC
45              Node<K,V> e;
46              if ((e = oldTab[j])
47                  // 将桶内节点设置为
48                  oldTab[j] = null
49                  // 如果链表只有一个
50                  if (e.next == nu
51                      newTab[e.has
```

```
52                  // 如果该节点是红黑
53                  else if (e insta
54                      ((TreeNode<K
55                  else { // preser
56                      // 说明是链表
57                      // “lo”前缀代
58                      // loHead是头
59                      Node<K,V> lo
60                      Node<K,V> hi
61                      Node<K,V> ne
62                      do {
63                          next = e
64                          // 从if-e
65                          // 通过条
66                          // 不满足
67                          if ((e.h
68                              if (
69
70                              else
71
72                              loTa
73                          }
74                          else {
75                              if (
76
77                              else
78
79                              hiTa
80                          }
```

```
                    } while ((e
                    // 将loHead链
                    if (loTail !
                        loTail.n
                        newTab[j
                    }
                    // 将hiHead链
                    if (hiTail !
                        hiTail.n
                        newTab[j
                    }
                }
            }
        }
        return newTab;
    }
```

其实读者对上面的代码读起来并不是很难，逻辑也很清晰，但是对于如何将一个链表拆分为两个链表可能会存在疑问，不知思绪在何方！其实光从链表分割的判断依据`(e.hash & oldCap) == 0`很难直接看清楚是如何判断，那么接下来我们就从判断条件出发，使用图文的形式将问题说清楚。

举个例子来说明一下：

我们假设有一个哈希表，表的容量为默认初始容量15，那么`length - 1 = 15`，15使用二进制表示就是：`0000 0000 0000 0000 0000 0000 0000 1111`，而任何key计算出来的hash值就可以使用二进制来表示 那么 `(length - 1) & hash`，其实就是取hash值的最低四位，因为15的二进制前28位都是0，我们假设有四个K-V对，如下表所示：

键值对	键哈希值（二进制）	&运算后下标二进制	下标
{5 : A}	0000 0000 0000 0000 0000 0000 0000 0101	0101	5
{21 : B}	0000 0000 0000 0000 0000 0000 0001 0101	0101	5
{37 : C}	0000 0000 0000 0000 0000 0000 0010 0101	0101	5
{53 : D}	0000 0000 0000 0000 0000 0000 0011 0101	0101	5

正整数的hash值就是其本身，转换成二进制后对 length - 1进行&运算，其实就是最后的二进制结果都是0101，所以下标都是5。也就是说这四个K-V组成的Node节点都在一个bucker内，且组成了单链表。我们假设需要对当前哈希表进行扩容，那么扩容后的容量就是32，那么哥哥节点新的索引值就是(32 - 1) & hash，而31的二进制表示位0000 0000 0000 0000 0000 0000 0001 1111，其实就是取hash值的最低5位，因为31的二进制最低5位位11111，那么对hash值取最低五位其实就无非两种情况：最后四位和hash值一致第五位要么是1，要么是0，这么一来，上面四个键值对的key计算的下标如下表所示：

键值对	键哈希值（二进制）	&运算后下标二进制	下标
{5 : A}	0000 0000 0000 0000 0000 0000 0000 0101	0 0101	5
{21 : B}	0000 0000 0000 0000 0000 0000 0001 0101	1 0101	21
{37 : C}	0000 0000 0000 0000 0000 0000 0010 0101	0 0101	5
{53 : D}	0000 0000 0000 0000 0000 0000 0011 0101	1 0101	21

从上表中也可以看出来，00101和原理一样，下标依旧是5，而10101其实就是 10101 = 00101 + 10000 = 5 + 16 = j + oldCap ，其中j 就是节点在老哈希表中的下标。故虽然数组大小扩大了一倍，但是同一个key在新旧哈希表中对应的下标缺存在一定联系：要么一致，要么相差个oldCap。

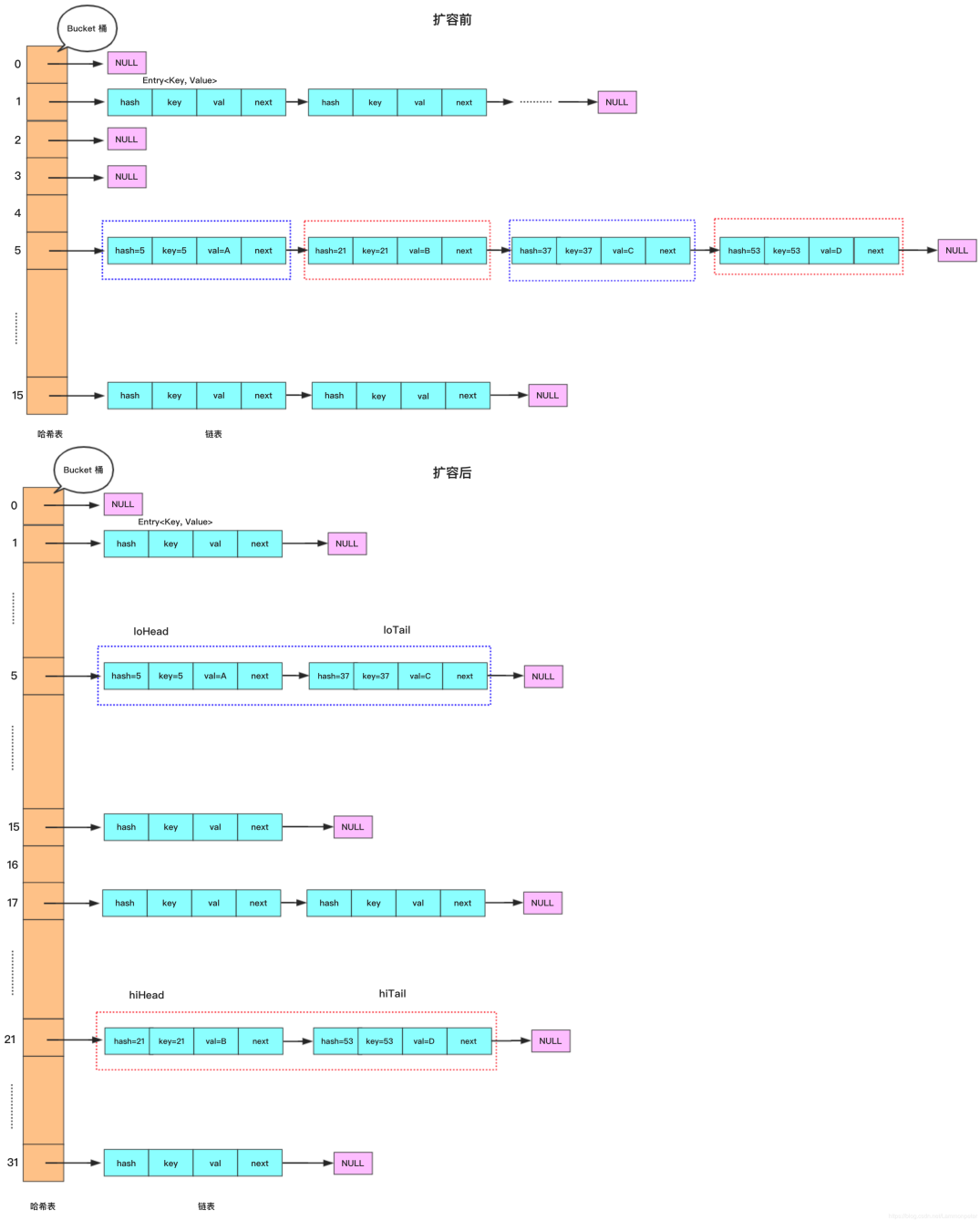
基于这个结论，那么我们只需要看新值的高位是0或者是1即可，这里假设是从16扩容到32的，那么看到第五位即可，其实就是 hash & 10000即可，也就是：

```
hash & 0000 0000 0000 0000 0000 0
```

上式等价于：

```
hash & oldCap
```

所以这个等式的结果要么是0，要么就是16，也就是新增的高位（第五位）要么是0，要么是1，当位0 的时候，那么NOde节点下标不变，当位1的时候，Node节点下为 `j + oldCap`，这里也就是`5 + 16 = 21`，扩容结果如下图所示：



到这里，就基本完成了对HashMap在两个版本中扩容机制的补充说明，通过上面图文的详细讲解，读者肯定对整个扩容机制有了一个更加深刻的认识。

2.5 理解HashMap的get方法

JDK7和JDK8的get方法其实原理上没有多大区别，仅仅是JDK8中多了红黑树的节点获取，其他的基本一致，这里贴出源码，逐行对源码进行注释分析，至于流程，可以参考《HashMap（JDK7）》中对get方法的流程图描述，或者自行画出流程图。这里就不再贴出流程图。

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

final Node<K,V> getNode(int hash,
    // tab:内部数组 first: 索引位首节
    Node<K,V>[] tab; Node<K,V> fi
    // 数组不为null 数组长度大于0 索引
     if ((tab = table) != null &&
         (first = tab[(n - 1) & h
         // 如果索引位首节点的hash =
         if (first.hash == hash &
             ((k = first.key) ==
             // 返回索引位首节点（值对
             return first;
         if ((e = first.next) !=
             // 如果是红黑树则到红黑树
             if (first instanceof
                 return ((TreeNod
             do {
                 // 遍历链表，查询key
                 if (e.hash == ha
                     ((k = e.key)
```

```
                    return e;
                } while ((e = e.next
            }
        }
        return null;
    }
```

把上述代码总结起来就是以下几个步骤：

1. 检查哈希表数组是否为null和索引位首节点（bucket的第一个节点）是否为null
2. 如果索引节点的hash==key的hash 或 key 和索引节点的k相同则直接返回（bucket的第一个节点）
3. 如果首节点是红黑树则到红给树查找key相同的节点
4. 不是首节点，也不是红黑树，那么就开始遍历链表，获取与key相同键的节点
5. 如果都没找到就返回null

三、几个关于HashMap的常见问题

3.1 都是HashMap是线程不安全的，那么到底不安全在哪?

HashMap在多线程环境下，存在数据覆盖的问题。这里以JDK7位代表举一个put的例子，线程1和线程2同时对哈希表中的某个索引位置put一个Entry节点，线程1获取到指定索引位置的头节点，线程2页同时获取到了指定位置的头节点，因此两个线程都同时创建一个新的Entry对象存到指定的索引位置上，并将新的Entry节点的next属性指向老的头节点，这就会产生数据覆盖的问题，假设线程2先完成，那么线程1就会直接覆盖掉线程2插入的头节点。

3.2 HashMap的扩容死锁是如何造成的?

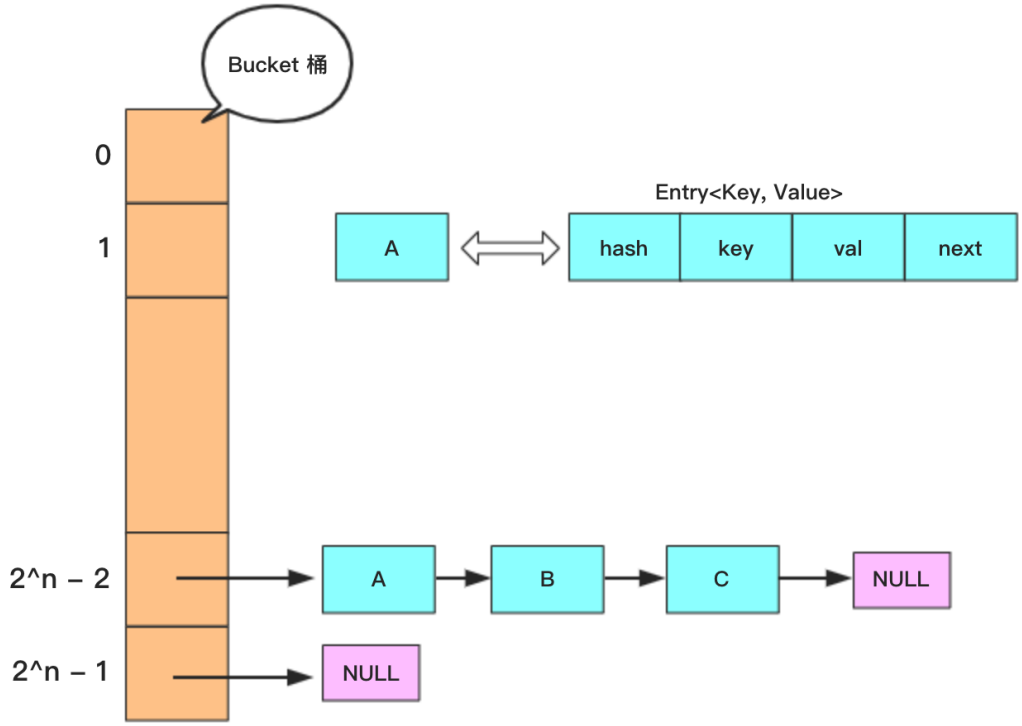
HashMap的扩容死锁问题发生在JDK7及以前版本中，这是因为JDK7级以前版本在扩容后的数据迁移采用的头插入法，JDK8以后版本进行了优化，采用的是尾

部插入法及两队链表，避免了该问题。现在就以JDK的HashMap为例，来分析一下是如何造成扩容死锁的。在JDK7中，数据的迁移是在如下代码中完成的：

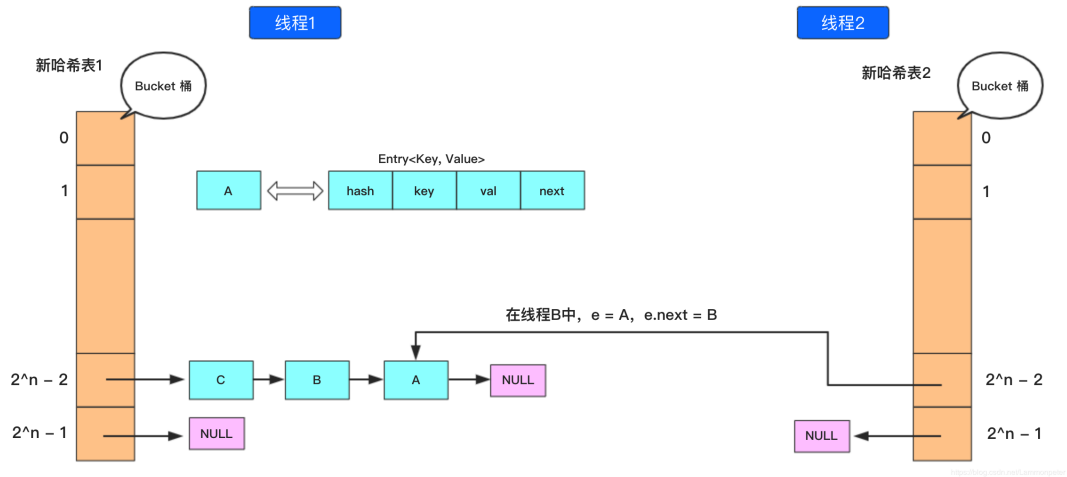
```
void transfer(Entry[] newTable, b
    int newCapacity = newTable.le
    // 遍历老的table，遍历到每一个buc
    for (Entry<K,V> e : table) {
        while(null != e) {
            Entry<K,V> next = e.n
            // 重新计算hash
            if (rehash) {
                e.hash = null ==
             }
             // 重新计算下标
             int i = indexFor(e.h
             // 头节点插入链表
             e.next = newTable[i]
             newTable[i] = e;
             // 继续原链表的下一个节
             e = next;
         }
     }
}
```

其实就是重新确定链表中的节点的索引下标，然后将其插入到对应的下标的链表内，JDK7采用的是头节点插入，其实这就给了多线程环境下产生扩容死锁机会。接下来我们使用图来说明这一现象。

原哈希表如下图所示，位了排版方便，不再统一画出hash、key、value、next等属性，统一使用一个方格表示这个四个属性。

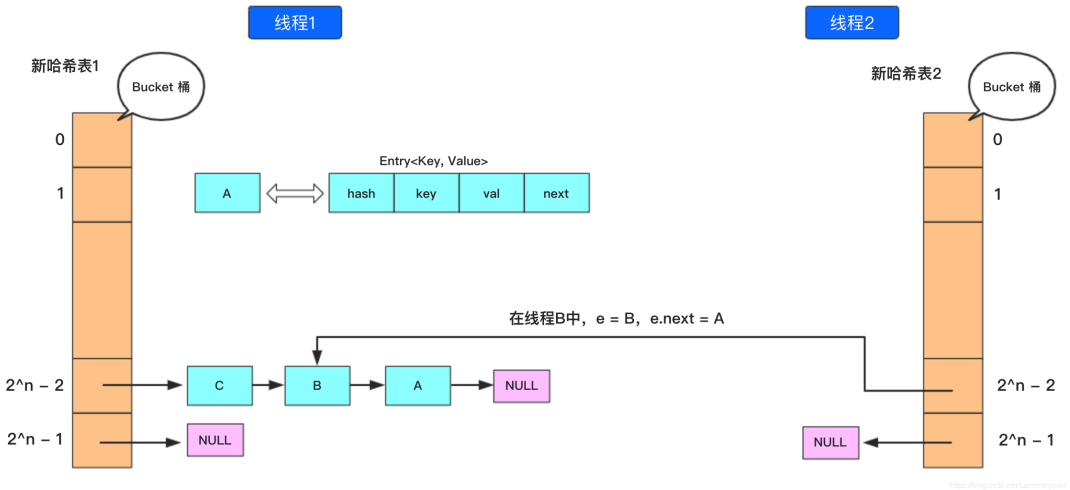


线程1和线程2各自创建了一个新的哈希表，假设在线程1已经做完扩容操作后，线程2才开始扩容。此时对于线程2来说，当前节点e指向A节点，下一个节点e.next仍然指向B节点，而此时在线程1的链表中，已经是C->B->A的顺序。按照头插法，线程2中的哈希表的bucket指向A节点，此时A节点成为线程B中链表的头节点，如下图所示：

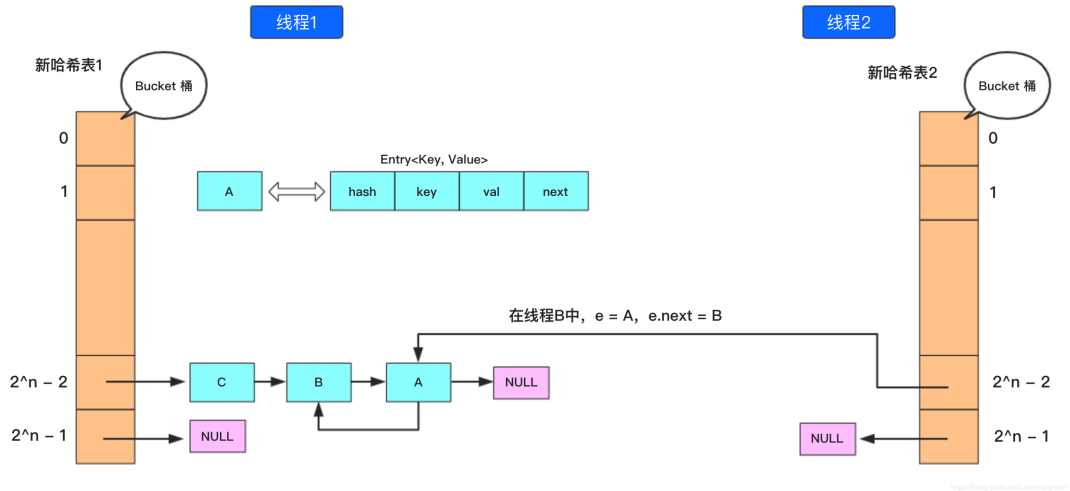


A节点成为线程2中链表的头节点后，下一个节点e.next为B节点。很显然，下一个节点满足 e.next != null，那么当前节点e就变成了B节点，下一个节点e.next变为A节点。继续执行头插法，将B变为链表的头节点，同时next指针执行旧的头

节点A，如下图：



此时，下一个节点e.next为A节点，不为null，继续头插法，指针后裔，那么当前节点就成了A节点，下一个节点为null。将A节点作为线程2链表中的头节点，并将next指针指向原来的旧头节点B，如下图所示：



此时已经形成环链表，A和B节点中互有对方引用，这也是HashMap线程不安全的一种表现。

3.3 如何规避HashMap的线程不安全问题?

HashMap是非线程安全的，可以使用Collections.SynchronizedMap() 来包装HashMap，使其具备线程安全的特性。多线程环境下，可以使用ConcrrrentHashMap来代替HashMap，ConcurrentHashMap的用法和HashMap基本一致。