在了解HashMap的扩容过程之前，先要了解一些hashmap中的变量。

Node<K,V>:链表节点，包含了key，value、hash、next

table：Node<K,V>类型的数组，里面的元素是链表，用于存放hashmap元素的实体。

size：记录了放入hashmap的元素个数。

loadFactor：负载因子。

threshold：阈值，决定了hashmap何时扩容，以及扩容后的大小，一般等于table大小乘以loadFactor

当我们自定义HashMap初始容量大小时，构造函数并非直接把我们定义的数值当作hashmap容量大小，而是把该数值当作参数调用方法tableSizeFor，然后把返回值作为hashmap的初始容量大小。

```
public HashMap(int initialCapacity, float loadFactor) {
    ...
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}
```

在tableSizeFor方法中会返回一个大于等于当前参数的2的倍数，因此hashmap中的数组的容量大小总是2的倍数。

我们知道，**HashMap的底层数组长度总是2的n次方**，原因是 HashMap 在其构造函数 HashMap(int initialCapacity, float loadFactor) 中作了特别的处理，如下面的代码所示。当底层数组的length为2的n次方时， h&(length - 1) 就相当于对length取模，其效率要比直接取模高得多，这是HashMap在效率上的一个优化。

```
// HashMap 的容量必须是2的幂次方，超过 initialCapacity 的最小 2^n
int capacity = 1;
while (capacity < initialCapacity)
    capacity <<= 1;
```

在上文已经提到过，HashMap 中的数据结构是一个数组链表，我们希望的是元素存放的越均匀越好。最理想的效果是，Entry数组中每个位置都只有一个元素，这样，查询的时候效率最高，不需要遍历单链表，也不需要通过equals去比较Key，而且空间利用率最大。

那如何计算才会分布最均匀呢？正如上一节提到的，HashMap采用了一个分两步走的哈希策略：

- 使用 hash() 方法用于对Key的hashCode进行重新计算，以防止质量低下的hashCode()函数实现。由于hashMap的支撑数组长度总是 2 的倍数，通过右移可以使低位的数据尽量的不同，从而使Key的hash值的分布尽量均匀；
- 使用 indexFor() 方法进行取余运算，以使Entry对象的插入位置尽量分布均匀(下文将专门对此阐述)。

对于取余运算，我们首先想到的是：哈希值%length = bucketIndex。**但当底层数组的length为2的n次方时， h&(length - 1) 就相当于对length取模，而且速度比直接取模快得多，这是HashMap在速度上的一个优化。**除此之外，HashMap 的底层数组长度总是2的n次方的主要原因是什么呢？我们借助于 chenssy 在其博客《java提高篇（二三）——HashMap》 中的关于这个问题的阐述：

这里，我们假设length分别为16(2^4) 和 15，h 分别为 5、6、7。

我们可以看到，当n=15时，6和7的结果一样，即它们位于table的同一个桶中，也就是产生了碰撞，6、7就会在这个桶中形成链表，这样就会导致查询速度降低。诚然这里只分析三个数字不是很多，那么我们再看 h 分别取 0-15时的情况。

h	length-1	h&length-1	
0	14	0000 & 1110 = 0000	0
1	14	0001 & 1110 = 0000	0
2	14	0010 & 1110 = 0010	2
3	14	0011 & 1110 = 0010	2
4	14	0100 & 1110 = 0100	4
5	14	0101 & 1110 = 0100	4
6	14	0110 & 1110 = 0110	6
7	14	0111 & 1110 = 0110	6
8	14	1000 & 1110 = 1000	8
9	14	1001 & 1110 = 1000	8
10	14	1010 & 1110 = 1010	10
11	14	1011 & 1110 = 1010	10
12	14	1100 & 1110 = 1100	12
13	14	1101 & 1110 = 1100	12
14	14	1110 & 1110 = 1110	14
15	14	1111 & 1110 = 1110	14

从上面的图表中我们可以看到，当 length 为15时总共发生了8次碰撞，同时发现空间浪费非常大，因为在 1、3、5、7、9、11、13、15 这八处没有存放数据。这是因为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。这样，空间的减少会导致碰撞几率的进一步增加，从而就会导致查询速度慢。

而当length为16时，length – 1 = 15，即 1111，那么，**在进行低位&运算时，值总是与原来hash值相同，而进行高位运算时，其值等于其低位值。**所以，当 length=2^n 时，不同的hash值发生碰撞的概率比较小，这样就会使得数据在table数组中分布较均匀，查询速度也较快。

因此，总的来说，HashMap 的底层数组长度总是2的n次方的原因有两个，即当 length=2^n 时：

- **不同的hash值发生碰撞的概率比较小，这样就会使得数据在table数组中分布较均匀，空间利用率较高，查询速度也较快；**
- **h&(length - 1) 就相当于对length取模，而且在速度、效率上比直接取模要快得多，即二者是等价不等效的，这是HashMap在速度和效率上的一个优化。**

何时进行扩容，hashmap使用的是懒加载，构造完hashmap对象后，只要不进行put方法插入元素，hashmap并不会去初始化或者扩容table。

当首次调用put方法时，hashmap会发现table为空，然后调用resize方法进行初始化
在put方法后，如果hashmap发现size大于threshold时，则会调用resize方法进行扩容。

扩容：
若threshold（阈值）不为空，table的首次初始化大小为阈值，否则初始化为缺省值16.
当table需要扩容时，扩容后的table大小变为原来的两倍，接下来就是进行扩容后table的调整：
假设扩容前的table大小为2的N次方，有上述put方法解析可知，元素的table索引为其hash值得后N位确定。
那么扩容后得table大小即为2的N+1次方，则其中元素的table索引为其hash值得后N+1位确定，比原来多了一位。
因此：table中的元素只有两种情况：

1. 元素hash值第N+1位为0：不需要进行位置调整。
2. 元素hash值第N+1位为1：调整至原索引的两倍位置。

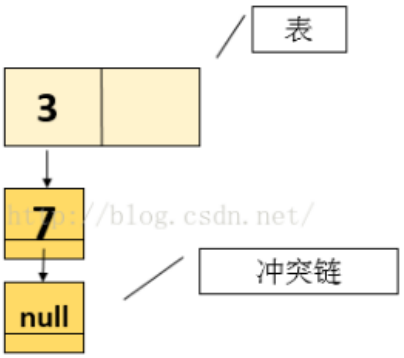
在resize方法中，判断N+1是否为0
若为0，将元素移至新table的原索引处。

若不为0，将元素移至新table的两倍索引处。

例子：HashMap内部的Node数组默认的大小是16，假设有100万个元素，那么最好的情况下每个hash桶里有62500个元素，这时get、put、remove等方法的效率都会很低。为了解决这个问题，HashMap提供了扩容机制，当元素个数达到数组大小loadFactor后会扩大数组的大小，在默认的情况下，数组大小为16，loadFactor为0.75，也就是说当HashMap中的元素超过16/0.75=12时，会把数组大小扩展到2*16=32，并且重新计算每个元素在新数组中的位置。

HashMap在多线程扩容的时候　死循环。

假设放置结果图如下：



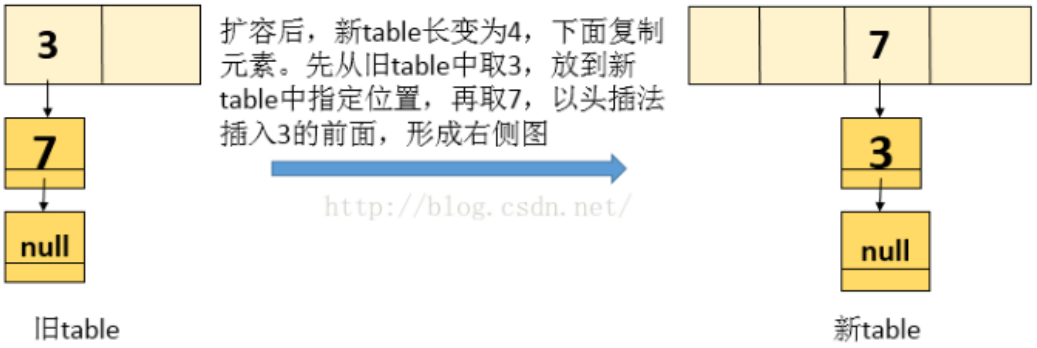
现在有两个线程A和B，都要执行put操作，即向表中添加元素，即线程A和线程B都会看到上面图的状态快照

执行顺序如下：

执行一：　线程A执行到transfer函数中（1）处挂起（transfer函数代码中有标注）。此时在线程A的栈中

```java
e = 3
next = 7
```

执行二：线程B执行 transfer函数中的while循环，即会把原来的table变成新一table（线程B自己的栈中），再写入到内存中。如下图（假设两个元素在新的hash函数下也会映射到同一个位置）



线程B扩容前后的过程

执行三： 线程A解挂，接着执行（看到的仍是旧表），即从transfer代码（1）处接着执行，当前的 e = 3, next = 7, 上面已经描述。

1. 处理元素 3 ， 将 3 放入 线程A自己栈的新table中（新table是处于线程A自己栈中，是线程私有的，不肥线程2的影响），处理3后的图如下：

线程A向新table中添加元素3的过程

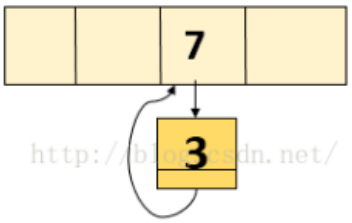
2. 线程A再复制元素 7 ，当前 e = 7 ,而next值由于线程 B 修改了它的引用，所以next 为 3 ，处理后的新表如下图

旧table　　新table

线程A向新table中添加元素7的过程

取7的next时，由于线程B已经修改了7的next（即，7 → 3 → null）,所以取到的next为3，不是null

3. 由于上面取到的next = 3, 接着while循环，即当前处理的结点为3， next就为null ，退出while循环，执行完while循环后，新表中的内容如下图：



新table

4. 当操作完成，执行查找时，会陷入死循环！