我就知道面试官接下来要问我 ConcurrentHashMap 底层原理了

2020-04-27 想当将军... 阅 8 转藏到我的图书馆

文章目录

这篇文章，我打算从以下几个方面来讲。

1）多线程下的 HashMap 有什么问题?

2）怎样保证线程安全，为什么选用 ConcurrentHashMap?

3）ConcurrentHashMap 1.7 源码解析

底层存储结构
常用变量
构造函数
put() 方法
ensureSegment() 方法
scanAndLockForPut() 方法
rehash() 扩容机制
get() 获取元素方法
remove() 方法
size() 方法是怎么统计元素个数的

4）ConcurrentHashMap 1.8 源码解析

put()方法详解
initTable()初始化表
addCount()方法
fullAddCount()方法
transfer()是怎样扩容和迁移元素的
helpTransfer()方法帮助迁移元素

想当将军的螺丝
关注 对话

多线程下 HashMap 有什么问题?

在上一篇文章中，已经讲解了 HashMap 1.7 死循环的成因，也正因为如此，我们才说 HashMap 在多线程下是不安全的。但是，在JDK1.8 的 HashMap 改为采用尾插法，已经不存在死循环的问题了，为什么也会线程不安全呢?

我们以 put 方法为例（1.8），

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
```

假如现在有两个线程都执行到了上图中的划线处。当线程一判断为空之后，CPU 时间片到了，被挂起。线程二也执行到此处判断为空，继续执行下一句，创建了一个新节点，插入到此下标位置。然后，线程一解挂，同样认为此下标的元素为空，因此也创建了一个新节点放在此下标处，因此造成了元素的覆盖。

所以，可以看到不管是 JDK1.7 还是 1.8 的 HashMap 都存在线程安全的问题。那么，在多线程环境下，应该怎样去保证线程安全呢?

怎样保证线程安全，为什么选用 ConcurrentHashMap?

首先，你可能想到，在多线程环境下用 Hashtable 来解决线程安全的问题。这样确实是可以的，但是同样的它也有缺点，我们看下最常用的 put 方法和 get 方法。

```
public synchronized V put(K key, V value) {
    // Make sure the value is not null
    if (value == null) {
        throw new NullPointerException();
    }

    // Makes sure the key is not already in the hashtable.
    Entry<?,?> tab[] = table;
    int hash = key.hashCode();
    int index = (hash & 0x7FFFFFFF) % tab.length;
    /unchecked/
    Entry<K,V> entry = (Entry<K,V>)tab[index];
    for(; entry != null ; entry = entry.next) {
        if ((entry.hash == hash) && entry.key.equals(key)) {
            V old = entry.value;
            entry.value = value;
            return old;
        }
    }

    addEntry(hash, key, value, index);
    return null;
}
```

Hashtable-put

```
@SuppressWarnings("unchecked")
public synchronized V get(Object key) {
    Entry<?,?> tab[] = table;
    int hash = key.hashCode();
    int index = (hash & 0x7FFFFFFF) % tab.length;
    for (Entry<?,?> e = tab[index] ; e != null ; e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            return (V)e.value;
        }
    }
    return null;
}
```

Hatable-get

可以看到，不管是往 map 里边添加元素还是获取元素，都会用 synchronized 关键字加锁。当有多个元素之前存在资源竞争时，只能有一个线程可以获取到锁，操作资源。更不能忍的是，一个简单的读取操作，互相之间又不影响，为什么也不能同时进行呢?

所以，hashtable 的缺点显而易见，它不管是 get 还是 put 操作，都是锁住了整个 table，效率低下，因此 并不适合高并发场景。

也许，你还会想起来一个集合工具类 Collections，生成一个SynchronizedMap。其实，它和 Hashtable 差不多，同样的原因，锁住整张表，效率低下。

```
private static class SynchronizedMap<K,V>
    implements Map<K,V>, Serializable {
    private static final long serialVersionUID = 1978198479659022715L;

    private final Map<K,V> m;     // Backing Map
    final Object      mutex;        // Object on which to synchronize

    SynchronizedMap(Map<K,V> m) {
        this.m = Objects.requireNonNull(m);
        mutex = this;
    }

    SynchronizedMap(Map<K,V> m, Object mutex) {
        this.m = m;
        this.mutex = mutex;
    }

    public int size() { synchronized (mutex) {return m.size();} }
    public boolean isEmpty() { synchronized (mutex) {return m.isEmpty();} }
    public boolean containsKey(Object key) { synchronized (mutex) {return m.co
    public boolean containsValue(Object value) { synchronized (mutex) {return
    public V get(Object key) { synchronized (mutex) {return m.get(key);} }

    public V put(K key, V value) { synchronized (mutex) {return m.put(key, va
    public V remove(Object key) { synchronized (mutex) {return m.remove(key);}
    public void putAll(Map<? extends K, ? extends V> map) { synchronized (mute
    public void clear() { synchronized (mutex) {m.clear();} }
```

所以，思考一下，既然锁住整张表的话，并发效率低下，那我把整张表分成 N 个部分，并使元素尽量均匀的分布到每个部分中，分别给他们加锁，互相之间并不影响，这种方式岂不是更好 。这就是在 JDK1.7 中 ConcurrentHashMap 采用的方案，被叫做锁分段技术，每个部分就是一个 Segment（段）。

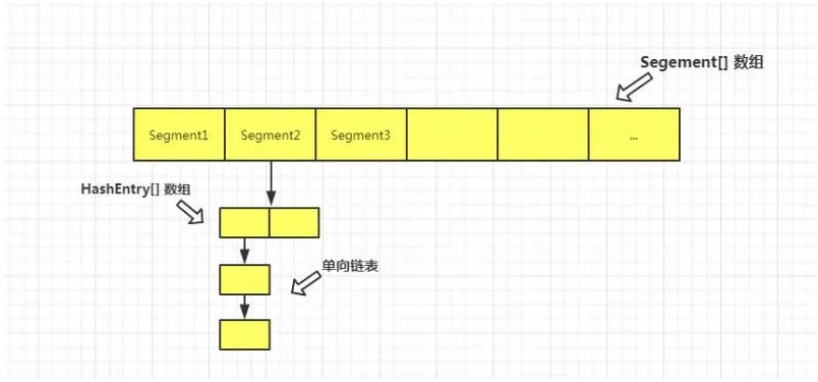
但是，在JDK1.8中，完全重构了，采用的是 Synchronized + CAS ，把锁的粒度进一步降低，而放弃了 Segment 分段。（此时的 Synchronized 已经升级了，效率得到了很大提升，锁升级可以了解一下）

ConcurrentHashMap 1.7 源码解析

我们看下在 JDK1.7中 ConcurrentHashMap 是怎么实现的。墙裂建议，在本文之前了解一下多线程的基本知识，如JMM内存模型，volatile关键字作用，CAS和自旋，ReentranLock 重入锁。

底层存储结构

在 JDK1.7中，本质上还是采用链表+数组的形式存储键值对的。但是，为了提高并发，把原来的整个 table 划分为 n 个 Segment 。所以，从整体来看，它是一个由 Segment 组成的数组。然后，每个 Segment 里边是由 HashEntry 组成的数组，每个 HashEntry之间又可以形成链表。我们可以把每个 Segment 看成是一个小的 HashMap，其内部结构和 HashMap 是一模一样的。



当对某个 Segment 加锁时，如图中 Segment2，并不会影响到其他 Segment 的读写。每个 Segment 内部自己操作自己的数据。这样一来，我们要做的就是尽可能的让元素均匀的分布在不同的 Segment中。最理想的状态是，所有执行的线程操作的元素都是不同的 Segment，这样就可以降低锁的竞争。

废话了这么多，还是来看底层源码吧，因为所有的思想都在代码里体现。借用 Linus的一句话，“No BB . Show me the code ” （改编版，哈哈）

常用变量

先看下 1.7 中常用的变量和内部类都有哪些，这有助于我们了解 ConcurrentHashMap 的整体结构。

```
//默认初始化容量，这个和 HashMap中的容量是一个概念，表示的是整个 Map的容量
static final int DEFAULT_INITIAL_CAPACITY = 16;

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

//默认的并发级别，这个参数决定了 Segment 数组的长度
static final int DEFAULT_CONCURRENCY_LEVEL = 16;

//最大的容量
static final int MAXIMUM_CAPACITY = 1 << 30;

//每个Segment中table数组的最小长度为2，且必须是2的n次幂。
//由于每个Segment是懒加载的，用的时候才会初始化，因此为了避免使用时立即调整大小，设定了最小容量2
static final int MIN_SEGMENT_TABLE_CAPACITY = 2;

//用于限制Segment数量的最大值，必须是2的n次幂
static final int MAX_SEGMENTS = 1 << 16; // slightly conservative

//在size方法和containsValue方法，会优先采用乐观的方式不加锁，直到重试次数达到2，才会对所有Segment加锁
//这个值的设定，是为了避免无限次的重试。后边size方法会详讲怎么实现乐观机制的。
static final int RETRIES_BEFORE_LOCK = 2;

//segment掩码值，用于根据元素的hash值定位所在的 Segment 下标。后边会细讲
final int segmentMask;
```

```
//和 segmentMask 配合使用来定位 Segment 的数组下标，后边讲。
final int segmentShift;

// Segment 组成的数组，每一个 Segment 都可以看做是一个特殊的 HashMap
final Segment<K,V>[] segments;

//Segment 对象，继承自 ReentrantLock 可重入锁。
//其内部的属性和方法和 HashMap 神似，只是多了一些拓展功能。
static final class Segment<K,V> extends ReentrantLock implements Serializab
le {

  //这是在 scanAndLockForPut 方法中用到的一个参数，用于计算最大重试次数
  //获取当前可用的处理器的数量，若大于1，则返回64，否则返回1。
  static final int MAX_SCAN_RETRIES =
    Runtime.getRuntime().availableProcessors() > 1 ? 64 : 1;

  //用于表示每个Segment中的 table，是一个用HashEntry组成的数组。
  transient volatile HashEntry<K,V>[] table;

  //Segment中的元素个数，每个Segment单独计数（下边的几个参数同样的都是单独计数）
  transient int count;

  //每次 table 结构修改时，如put，remove等，此变量都会自增
  transient int modCount;

  //当前Segment扩容的阈值，同HashMap计算方法一样也是容量乘以加载因子
  //需要知道的是，每个Segment都是单独处理扩容的，互相之间不会产生影响
  transient int threshold;

  //加载因子
  final float loadFactor;

  //Segment构造函数
  Segment(float lf, int threshold, HashEntry<K,V>[] tab) {
    this.loadFactor = lf;
    this.threshold = threshold;
    this.table = tab;
  }

  ...
  // put(),remove(),rehash() 方法都在此类定义
}

// HashEntry，存在于每个Segment中，它就类似于HashMap中的Node，用于存储键值对的具
体数据和维护单向链表的关系
static final class HashEntry<K,V> {
  //每个key通过哈希运算后的结果，用的是 Wang/Jenkins hash 的变种算法，此处不细
讲，感兴趣的可自行查阅相关资料
  final int hash;
  final K key;
  //value和next都用 volatile 修饰，用于保证内存可见性和禁止指令重排序
  volatile V value;
  //指向下一个节点
  volatile HashEntry<K,V> next;

  HashEntry(int hash, K key, V value, HashEntry<K,V> next) {
    this.hash = hash;
    this.key = key;
    this.value = value;
    this.next = next;
```

```
  }
}
```

构造函数

ConcurrentHashMap 有五种构造函数，但是最终都会调用同一个构造函数，所以只需要搞明白这一个核心的构造函数就可以了。

PS: 文章注释中 (1)(2)(3) 等序号都是用来方便做标记，不是计算值

```
public ConcurrentHashMap(int initialCapacity,
                                                    float loadFactor, int
  concurrencyLevel) {
  //检验参数是否合法。值得说的是，并发级别一定要大于0，否则就没办法实现分段锁了。
  if (!
(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
    throw new IllegalArgumentException();
  //并发级别不能超过最大值
  if (concurrencyLevel > MAX_SEGMENTS)
    concurrencyLevel = MAX_SEGMENTS;
  // Find power-of-two sizes best matching arguments
  //偏移量，是为了对hash值做位移操作，计算元素所在的Segment下标，put方法详讲
  int sshift = 0;
  //用于设定最终Segment数组的长度，必须是2的n次幂
  int ssize = 1;
  //这里就是计算 sshift 和 ssize 值的过程    (1)
  while (ssize < concurrencyLevel) {
    ++sshift;
    ssize <<= 1;
  }
  this.segmentShift = 32 - sshift;
  //Segment的掩码
  this.segmentMask = ssize - 1;
  if (initialCapacity > MAXIMUM_CAPACITY)
    initialCapacity = MAXIMUM_CAPACITY;
  //c用于辅助计算cap的值      (2)
  int c = initialCapacity / ssize;
  if (c * ssize < initialCapacity)
    ++c;
  // cap 用于确定某个Segment的容量，即Segment中HashEntry数组的长度
  int cap = MIN_SEGMENT_TABLE_CAPACITY;
  //(3)
  while (cap < c)
    cap <<= 1;
  // create segments and segments[0]
  //这里用 loadFactor做为加载因子，cap乘以加载因子作为扩容阈值，创建长度为cap的
HashEntry数组，
  //三个参数，创建一个Segment对象，保存到S0对象中。后边在 ensureSegment 方法会用到
S0作为原型对象去创建对应的Segment。
  Segment<K,V> s0 =
    new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
             (HashEntry<K,V>[])new HashEntry[cap]);
  //创建出长度为 ssize 的一个 Segment数组
  Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
  //把S0存到Segment数组中去。在这里，我们就可以发现，此时只是创建了一个Segment数
组，
  //但是并没有把数组中的每个Segment对象创建出来，仅仅创建了一个Segment用来作为原型
对象。
  UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]
  this.segments = ss;
}
```须是2的n次幂

```
  int  ssize  =  1;
```

上边的注释中留了 (1)(2)(3) 三个地方还没有细说。我们现在假设一组数据，把涉及到的几个变量计算出来，就能明白这些参数的含义了。

```java
//假设调用了默认构造，都用的是默认参数，
即  initialCapacity  和  concurrencyLevel  都是16
//(1)   sshift 和 ssize 值的计算过程为，每次循环，都会把 sshift 自增1，并且 ssize 左移一位，即乘以2，
//直到 ssize 的值大于等于 concurrencyLevel 的值 16。
sshfit=0,1,2,3,4
ssize=1,2,4,8,16
//可以看到，初始他们的值分别是0和1，最终结果是4和16
//sshfit是为了辅助计算segmentShift值，ssize是为了确定Segment数组长度。
//(2)    此时,计算c的值，
c  =  16/16  =  1;
//判断 c * 16 < 16 是否为真，真的话 c 自增1，此处为false，因此 c的值为1不变。
//(3)    此时，由于c为1， cap为2 ，因此判断 cap < c 为false，最终cap为2。
//总结一下，以上三个步骤，最终都是为了确定以下几个关键参数的值，
//确定  segmentShift  ，这个用于后边计算hash值的偏移量，此处即为  32-4=28，
//确定  ssize，必须是一个大于等于  concurrencyLevel  的一个2的n次幂值
//确定  cap，必须是一个大于等于2的一个2的n次幂值
//感兴趣的小伙伴，还可以用另外几组参数来计算上边的参数值，可以加深理解参数的含义。
//例如initialCapacity和concurrencyLevel分别传入10和5，或者传入33和16
```

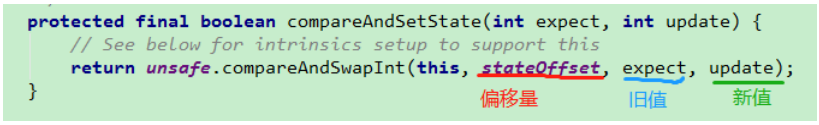
put()方法

put 方法的总体流程是，

通过哈希算法计算出当前 key 的 hash 值
通过这个 hash 值找到它所对应的 Segment 数组的下标
再通过 hash 值计算出它在对应 Segment 的 HashEntry数组 的下标
找到合适的位置插入元素

```java
//这是Map的put方法
public  V  put(K  key,  V  value)  {
  Segment<K,V>  s;
  //不支持value为空
  if  (value  ==  null)
    throw  new  NullPointerException();
  //通过  Wang/Jenkins  算法的一个变种算法，计算出当前key对应的hash值
  int  hash  =  hash(key);
  //上边我们计算出的  segmentShift为28，因此hash值右移28位，说明此时用的是hash的高4位，
  //然后把它和掩码15进行与运算，得到的值一定是一个  0000  ~ 1111 范围内的值，即  0~15 。
  int  j  =  (hash  >>>  segmentShift)  &  segmentMask;
  //这里是用Unsafe类的原子操作找到Segment数组中j下标的  Segment  对象
  if  ((s  =  (Segment<K,V>)UNSAFE.getObject                    //  nonvolatile; recheck
      (segments,  (j  <<  SSHIFT)  +  SBASE))  ==  null)  //    in  ensureSegment
    //初始化j下标的Segment
    s  =  ensureSegment(j);
  //在此Segment中添加元素
  return  s.put(key,  hash,  value,  false);
}
```

上边有一个这样的方法， UNSAFE.getObject (segments, (j << SSHIFT) + SBASE。它是为了通过Unsafe这个类，找到 j 最新的实际值。这个计算 (j << SSHIFT) + SBASE ，在后边非常常见，我们只需要知道它代表的是 j 的一个偏移量，通过偏移量，就可以得到 j 的实际值。可以类比，AQS 中的 CAS 操作。Unsafe中的操作，都需要一个偏移量，看下图，



(j << SSHIFT) + SBASE 就相当于图中的 stateOffset偏移量。只不过图中是 CAS 设置新值，而我们这里是取 j 的最新值。后边很多这样的计算方式，就不赘述了。接着看 s.put 方法，这才是最终确定元素位置的方法。

```
//Segment中的 put 方法
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
  //这里通过tryLock尝试加锁，如果加锁成功，返回null，否则执行 scanAndLockForPut方法
  //这里说明一下，tryLock 和 lock 是 ReentrantLock 中的方法，
  //区别是 tryLock 不会阻塞，抢锁成功就返回true，失败就立马返回false，
  //而 lock 方法是，抢锁成功则返回，失败则会进入同步队列，阻塞等待获取锁。
  HashEntry<K,V> node = tryLock() ? null :
    scanAndLockForPut(key, hash, value);
  V oldValue;
  try {
    //当前Segment的table数组
    HashEntry<K,V>[] tab = table;
    //这里就是通过hash值，与tab数组长度取模，找到其所在HashEntry数组的下标
    int index = (tab.length - 1) & hash;
    //当前下标位置的第一个HashEntry节点
    HashEntry<K,V> first = entryAt(tab, index);
    for (HashEntry<K,V> e = first;;) {
      //如果第一个节点不为空
      if (e != null) {
        K k;
        //并且第一个节点，就是要插入的节点，则替换value值，否则继续向后查找
        if ((k = e.key) == key ||
          (e.hash == hash && key.equals(k))) {
          //替换旧值
          oldValue = e.value;
          if (!onlyIfAbsent) {
            e.value = value;
            ++modCount;
          }
          break;
        }
        e = e.next;
      }
      //说明当前index位置不存在任何节点，此时first为null，
      //或者当前index存在一条链表，并且已经遍历完了还没找到相等的key，此时first就是链表第一个元素
      else {
        //如果node不为空，则直接头插
        if (node != null)
          node.setNext(first);
        //否则，创建一个新的node，并头插
        else
          node = new HashEntry<K,V>(hash, key, value, first);
        int c = count + 1;
        //如果当前Segment中的元素大于阈值，并且tab长度没有超过容量最大值，则扩容
        if (c > threshold && tab.length < MAXIMUM_CAPACITY)
          rehash(node);
        //否则，就把当前node设置为index下标位置新的头结点
        else
          setEntryAt(tab, index, node);
        ++modCount;
        //更新count值
```

```
        count = c;
        //这种情况说明旧值肯定为空
        oldValue = null;
        break;
      }
    }
  } finally {
    //需要注意ReentrantLock必须手动解锁
    unlock();
  }
  //返回旧值
  return oldValue;
}
```

这里说明一下计算 Segment 数组下标和计算 HashEntry 数组下标的不同点：

```
//下边的hash值是通过哈希运算后的hash值，不是hashCode
//计算 Segment 下标
  (hash >>> segmentShift) & segmentMask
  //计算 HashEntry 数组下标
  (tab.length - 1) & hash
```

思考一下，为什么它们的算法不一样呢？计算 Segment 数组下标是用的 hash值高几位（这里以高 4 位为例）和掩码做与运算，而计算 HashEntry 数组下标是直接用的 hash 值和数组长度减1做与运算。

我的理解是，这是为了尽量避免当前 hash 值计算出来的 Segment 数组下标和计算出来的 HashEntry 数组下标趋于相同。简单说，就是为了避免分配到同一个 Segment 中的元素扎堆现象，即避免它们都被分配到同一条链表上，导致链表过长。同时，也是为了减少并发。下面做一个运算，帮助理解一下（假设不用高 4 位运算，而是正常情况都用低位做运算）。

```
//我们以并发级别16，HashEntry数组容量 4 为例，则它们参与运算的掩码分别为 15 和 3
//hash值
0110 1101 0110 1111 0110 1110 0010 0010
//segmentMask = 15 ，标记为 (1)
0000 0000 0000 0000 0000 0000 0000 1111
//tab.length - 1 = 3 ，标记为 (2)
0000 0000 0000 0000 0000 0000 0000 0011
//用 hash 分别和 15 ，3 做与运算，会发现得到的结果是一样，都是十进制 2.
//这表明，当前 hash值被分配到下标为 2 的 Segment 中，同时，被分配到下标为 2 的 HashEntry 数组中
//现在若有另外一个 hash 值 h2，和第一个hash值，高位不同，但是低4位相同，
1010 1101 0110 1111 0110 1110 0010 0010
//我们会发现，最后它也会被分配到下标为 2 的 Segment 和 HashEntry 数组，就会和第一个元素形成链表。
//所以，为了避免这种扎堆现象，让元素尽量均匀分配，就让 hash 的高 4 位和 (1)处做与 运算，而用低位和 (2)处做与运算
//这样计算后，它们所在的Segment下标分别为 6(0110), 10(1010)，即使它们在HashEntry数组中的下标都为 2(0010)，也无所谓
//因为它们并不在一个 Segment 中，也就不会在同一个 HashEntry 数组中，更不会形成链表。
//更重要的是，它们不会有并发，因为在各自不同的 Segment 自己操作自己的加锁解锁，互不影响
```

可能有的小伙伴就会打岔了，那如果两个 hash 值，低位和高位都相同，怎么办呢。如果是这样，我只能说，这个 hash 算法也太烂了吧。（这里的 hash 算法也会尽量避免这种情况，当然只是减少几率，并不能杜绝）

我有个大胆的想法，这里的高低位不同的计算方式，是不是后边 1.8 HashMap 让 hash 高低位做异或运算的引子呢？不得而知。。

put 方法比较简单，只要能看懂 HashMap 中的 put 方法，这里也没问题。主要是它调用的子方法比较复杂，下边一个一个讲解。

ensureSegment()方法

回到 Map的 put 方法，判断 j 下标的 Segment为空后，则需要调用此方法，初始化一个 Segment 对象，以确保拿到的对象一定是不为空的，否则无法执行s.put了。

```
//k为 (hash >>> segmentShift) & segmentMask 算法计算出来的值
private Segment<K,V> ensureSegment(int k) {
  final Segment<K,V>[] ss = this.segments;
  //u代表 k 的偏移量，用于通过 UNSAFE 获取主内存最新的实际 K 值
  long u = (k << SSHIFT) + SBASE; // raw offset
  Segment<K,V> seg;
  //从内存中取到最新的下标位置的 Segment 对象，判断是否为空，(1)
  if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) {
    //之前构造函数说了，s0是作为一个原型对象，用于创建新的 Segment 对象
    Segment<K,V> proto = ss[0]; // use segment 0 as prototype
    //容量
    int cap = proto.table.length;
    //加载因子
    float lf = proto.loadFactor;
    //扩容阈值
    int threshold = (int)(cap * lf);
    //把 Segment 对应的 HashEntry 数组先创建出来
    HashEntry<K,V>[] tab = (HashEntry<K,V>[])new HashEntry[cap];
    //再次检查 K 下标位置的 Segment 是否为空， (2)
    if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))
      == null) { // recheck
      //此处把 Segment 对象创建出来，并赋值给 s，
      Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);
      //循环检查 K 下标位置的 Segment 是否为空， (3)
      //若不为空，则说明有其它线程抢先创建成功，并且已经成功同步到主内存中了，
      //则把它取出来，并返回
      while ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))
             == null) {
        //CAS，若当前下标的Segment对象为空，就把它替换为最新创建出来的 s 对象。
        //若成功，就跳出循环，否则，就一直自旋直到成功，或者 seg 不为空（其他线程成功导致）。
        if (UNSAFE.compareAndSwapObject(ss, u, null, seg = s))
          break;
      }
    }
  }
  return seg;
}
```

可以发现，我标注了上边 (1)(2)(3) 个地方，每次都判断最新的Segment是否为空。可能有的小伙伴就会迷惑，为什么做这么多次判断，我直接去自旋不就好了，反正最后都要自旋的。

我的理解是，在多线程环境下，因为不确定是什么时候会有其它线程 CAS 成功，有可能发生在以上的任意时刻。所以，只要发现一旦内存中的对象已经存在了，则说明已经有其它线程把Segment对象创建好，并CAS成功同步到主内存了。此时，就可以直接返回，而不需要往下执行了。这样做，是为了代码执行效率考虑。

scanAndLockForPut()方法

put 方法第一步抢锁失败之后，就会执行此方法，

```
private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {
  //根据hash值定位到它对应的HashEntry数组的下标位置，并找到链表的第一个节点
```

```
  //注意，这个操作会从主内存中获取到最新的状态，以确保获取到的first是最新值
  HashEntry<K,V> first = entryForHash(this, hash);
  HashEntry<K,V> e = first;
  HashEntry<K,V> node = null;
  //重试次数，初始化为 -1
  int retries = -1; // negative while locating node
  //若抢锁失败，就一直循环，直到成功获取到锁。有三种情况
  while (!tryLock()) {
    HashEntry<K,V> f; // to recheck first below
    //1.若 retries 小于0，
    if (retries < 0) {
      if (e == null) {
        //若 e 节点和 node 都为空，则创建一个 node 节点。这里只是预测性的创建一个
node节点
        if (node == null) // speculatively create node
          node = new HashEntry<K,V>(hash, key, value, null);
        retries = 0;
      }
      //如当前遍历到的 e 节点不为空，则判断它的key是否等于传进来的key，若是则
把 retries 设为0
      else if (key.equals(e.key))
        retries = 0;
      //否则，继续向后遍历节点
      else
        e = e.next;
    }
    //2.若是重试次数超过了最大尝试次数，则调用lock方法加锁。表明不再重试，我下定决
心了一定要获取到锁。
    //要么当前线程可以获取到锁，要么获取不到就去排队等待获取锁。获取成功后，再
break。
    else if (++retries > MAX_SCAN_RETRIES) {
      lock();
      break;
    }
    //3.若 retries 的值为偶数，并且从内存中再次获取到最新的头节点，判断若不等于
first
    //则说明有其他线程修改了当前下标位置的头结点，于是需要更新头结点信息。
    else if ((retries & 1) == 0 &&
          (f = entryForHash(this, hash)) != first) {
      //更新头结点信息，并把重试次数重置为 -1，继续下一次循环，从最新的头结点遍历
当前链表。
      e = first = f; // re-traverse if entry changed
      retries = -1;
    }
  }
  return node;
}
```

这个方法逻辑比较复杂，会一直循环尝试获取锁，若获取成功，则返回。否则的话，每次循环时，都会同时遍历当前链表。若遍历完了一次，还没找到和key相等的节点，就会预先创建一个节点。注意，这里只是预测性的创建一个新节点，也有可能在这之前，就已经获取锁成功了。

同时，当重试次每偶数次时，就会检查一次当前最新的头结点是否被改变。因为若有变化的话，还需要从最新的头结点开始遍历链表。

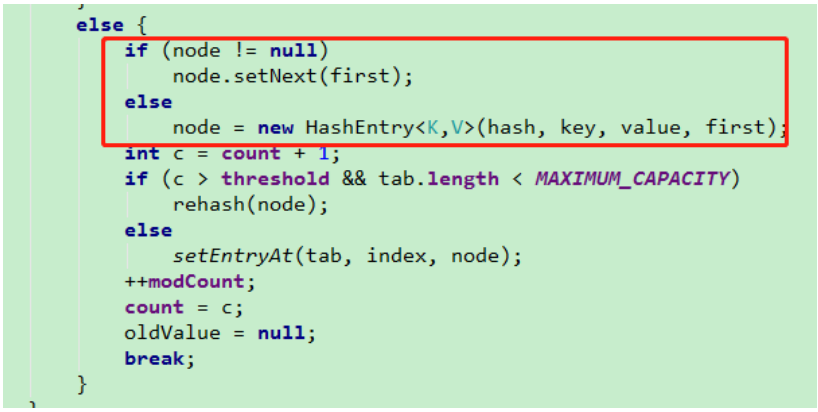
还有一种情况，就是循环次数达到了最大限制，则停止循环，用阻塞的方式去获取锁。这时，也就停止了遍历链表的动作，当前线程也不会再做其他预热(warm up)的事情。

关于为什么预测性的创建新节点，源码中原话是这样的：

Since traversal speed doesn't matter, we might as well help warm up the associated code and accesses as well.

解释一下就是，因为遍历速度无所谓，所以，我们可以预先(**warm up**)做一些相关联代码的准备工作。这里相关联代码，指的就是循环中，在获取锁成功或者调用 lock 方法之前做的这些事情，当然也包括创建新节点。

在put 方法中可以看到，有一句是判断 node 是否为空，若创建了，就直接头插。否则的话，它也会自己创建这个新节点。



```
        else {
            if (node != null)
                node.setNext(first);
            else
                node = new HashEntry<K,V>(hash, key, value, first);
            int c = count + 1;
            if (c > threshold && tab.length < MAXIMUM_CAPACITY)
                rehash(node);
            else
                setEntryAt(tab, index, node);
            ++modCount;
            count = c;
            oldValue = null;
            break;
        }
```

scanAndLockForPut 这个方法可以确保返回时，当前线程一定是获取到锁的状态。

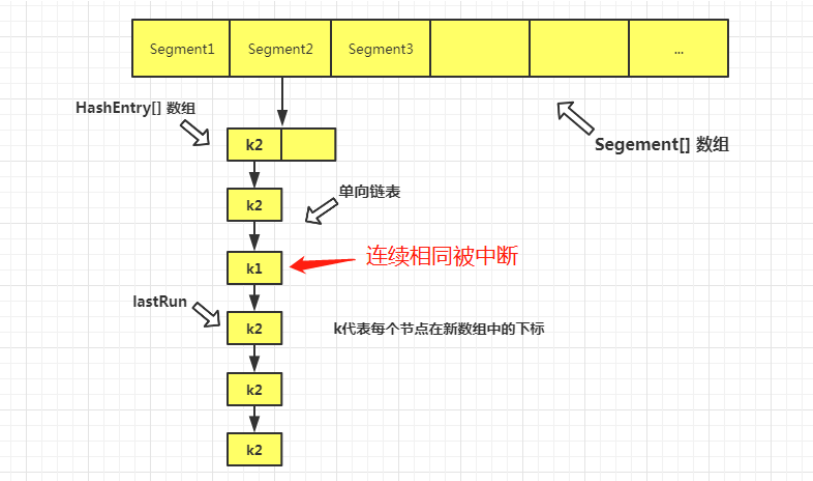
rehash()方法

当 put 方法时，发现元素个数超过了阈值，则会扩容。需要注意的是，每个Segment只管它自己的扩容，互相之间并不影响。换句话说，可以出现这个 Segment的长度为2，另一个Segment的长度为4的情况（只要是2的n次幂）。

```
//node为创建的新节点
private void rehash(HashEntry<K,V> node) {
  //当前Segment中的旧表
  HashEntry<K,V>[] oldTable = table;
  //旧的容量
  int oldCapacity = oldTable.length;
  //新容量为旧容量的2倍
  int newCapacity = oldCapacity << 1;
  //更新新的阈值
  threshold = (int)(newCapacity * loadFactor);
  //用新的容量创建一个新的 HashEntry 数组
  HashEntry<K,V>[] newTable =
    (HashEntry<K,V>[]) new HashEntry[newCapacity];
  //当前的掩码，用于计算节点在新数组中的下标
  int sizeMask = newCapacity - 1;
  //遍历旧表
  for (int i = 0; i < oldCapacity ; i++) {
    HashEntry<K,V> e = oldTable[i];
    //如果e不为空，说明当前链表不为空
    if (e != null) {
      HashEntry<K,V> next = e.next;
      //计算hash值再新数组中的下标位置
      int idx = e.hash & sizeMask;
      //如果e不为空，且它的下一个节点为空，则说明这条链表只有一个节点，
      //直接把这个节点放到新数组的对应下标位置即可
      if (next == null)      //  Single node on list
        newTable[idx] = e;
      //否则，处理当前链表的节点迁移操作
```

```
      else { // Reuse consecutive sequence at same slot
        //记录上一次遍历到的节点
        HashEntry<K,V> lastRun = e;
        //对应上一次遍历到的节点在新数组中的新下标
        int lastIdx = idx;
        for (HashEntry<K,V> last = next;
            last != null;
            last = last.next) {
          //计算当前遍历到的节点的新下标
          int k = last.hash & sizeMask;
          //若 k 不等于 lastIdx，则说明此次遍历到的节点和上次遍历到的节点不在同一个下标位置
          //需要把 lastRun 和 lastIdx 更新为当前遍历到的节点和下标值。
          //若相同，则不处理，继续下一次 for 循环。
          if (k != lastIdx) {
            lastIdx = k;
            lastRun = last;
          }
        }
        //把和 lastRun 节点的下标位置相同的链表最末尾的几个连续的节点放到新数组的对应下标位置
        newTable[lastIdx] = lastRun;
        //再把剩余的节点，复制到新数组
        //从旧数组的头结点开始遍历，直到 lastRun 节点，因为 lastRun节点后边的节点都已经迁移完成了。
        for (HashEntry<K,V> p = e; p != lastRun; p = p.next) {
          V v = p.value;
          int h = p.hash;
          int k = h & sizeMask;
          HashEntry<K,V> n = newTable[k];
          //用的是复制节点信息的方式，并不是把原来的节点直接迁移，区别于lastRun处理方式
          newTable[k] = new HashEntry<K,V>(h, p.key, v, n);
        }
      }
    }
  }
  //所有节点都迁移完成之后，再处理传进来的新的node节点，把它头插到对应的下标位置
  int nodeIndex = node.hash & sizeMask; // add the new node
  //头插node节点
  node.setNext(newTable[nodeIndex]);
  newTable[nodeIndex] = node;
  //更新当前Segment的table信息
  table = newTable;
}
```

上边的迁移过程和 lastRun 和 lastIdx 变量可能不太好理解，我画个图就明白了。以其中一条链表处理方式为例。



从头结点开始向后遍历，找到当前链表的最后几个下标相同的连续的节点。如上图，虽然开头出现了有两个节点的下标都是 k2, 但是中间出现一个不同的下标 k1，打断了下标连续相同，因此从下一个k2，又重新开始算。好在后边三个连续的节点下标都是相同的，因此倒数第三个节点被标记为 lastRun，且变量无变化。

从lastRun节点到尾结点的这部分就可以整体迁移到新数组的对应下标位置了，因为它们的下标都是相同的，可以这样统一处理。

另外从头结点到 lastRun 之前的节点，无法统一处理，只能一个一个去复制了。且注意，这里不是直接迁移，而是复制节点到新的数组，旧的节点会在不久的将来，因为没有引用指向，被 JVM 垃圾回收处理掉。

（不知道为啥这个方法名起为 rehash，其实扩容时 hash 值并没有重新计算，变化的只是它们所在的下标而已。我猜测，可能是，借用了 1.7 HashMap 中的说法吧。。。）

get()

put 方法搞明白了之后，其实 get 方法就很好理解了。也是先定位到 Segment，然后再定位到 HashEntry 。

```
public V get(Object key) {
  Segment<K,V> s; // manually integrate access methods to reduce overhead
  HashEntry<K,V>[] tab;
  //计算hash值
  int h = hash(key);
  //同样的先定位到 key 所在的Segment ，然后从主内存中取出最新的节点
  long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;
  if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
    (tab = s.table) != null) {
    //若Segment不为空，且链表也不为空，则遍历查找节点
    for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile
          (tab, ((long)(((tab.length - 1) & h)) << TSHIFT) + TBASE);
        e != null; e = e.next) {
      K k;
      //找到则返回它的 value 值，否则返回 null
      if ((k = e.key) == key || (e.hash == h && key.equals(k)))
        return e.value;
    }
  }
  return null;
}
```

remove()

remove 方法和 put 方法类似，也不用做过多特殊的介绍，

```
public V remove(Object key) {
  int hash = hash(key);
  //定位到Segment
  Segment<K,V> s = segmentForHash(hash);
  //若 s为空，则返回 null，否则执行 remove
  return s == null ? null : s.remove(key, hash, null);
}

public boolean remove(Object key, Object value) {
  int hash = hash(key);
  Segment<K,V> s;
  return value != null && (s = segmentForHash(hash)) != null &&
    s.remove(key, hash, value) != null;
}

final V remove(Object key, int hash, Object value) {
  //尝试加锁，若失败，则执行 scanAndLock ，此方法和 scanAndLockForPut 方法类似
  if (!tryLock())
    scanAndLock(key, hash);
  V oldValue = null;
  try {
    HashEntry<K,V>[] tab = table;
    int index = (tab.length - 1) & hash;
    //从主内存中获取对应 table 的最新的头结点
    HashEntry<K,V> e = entryAt(tab, index);
    HashEntry<K,V> pred = null;
    while (e != null) {
      K k;
      HashEntry<K,V> next = e.next;
      //匹配到 key
      if ((k = e.key) == key ||
        (e.hash == hash && key.equals(k))) {
        V v = e.value;
        // value 为空，或者 value 也匹配成功
        if (value == null || value == v || value.equals(v)) {
          if (pred == null)
            setEntryAt(tab, index, next);
          else
            pred.setNext(next);
          ++modCount;
          --count;
          oldValue = v;
        }
        break;
      }
      pred = e;
      e = next;
    }
  } finally {
    unlock();
  }
  return oldValue;
}
```

size()

size 方法需要重点说明一下。爱思考的小伙伴可能就会想到，并发情况下，有可能在统计期间，数组元素个数不停的变化，而且，整个表还被分成了 N个 Segment，怎样统计才能保证结果的准确性呢？我们一起来看下吧。

```
public int size() {
  // Try a few times to get accurate count. On failure due to
```

```
  //  continuous  async  changes  in  table,  resort  to  locking.
  //segment数组
  final  Segment<K,V>[]  segments  =  this.segments;
  //统计所有Segment中元素的总个数
  int  size;
  //如果size大小超过32位，则标记为溢出为true
  boolean  overflow;
  //统计每个Segment中的  modcount  之和
  long  sum;
  //上次记录的  sum  值
  long  last  =  0L;
  //重试次数，初始化为  -1
  int  retries  =  -1;
  try  {
    for  (;;)  {
      //如果超过重试次数，则不再重试，而是把所有Segment都加锁，再统计  size
      if  (retries++  ==  RETRIES_BEFORE_LOCK)  {
        for  (int  j  =  0;  j  <  segments.length;  ++j)
          //强制加锁
          ensureSegment(j).lock();  //  force  creation
      }
      sum  =  0L;
      size  =  0;
      overflow  =  false;
      //遍历所有Segment
      for  (int  j  =  0;  j  <  segments.length;  ++j)  {
        Segment<K,V>  seg  =  segmentAt(segments,  j);
        //若当前遍历到的Segment不为空，则统计它的  modCount  和  count  元素个数
        if  (seg  !=  null)  {
          //累加当前Segment的结构修改次数，如put，remove等操作都会影响modCount
          sum  +=  seg.modCount;
          int  c  =  seg.count;
          //若当前Segment的元素个数  c  小于0  或者  size  加上  c  的结果小于0，
则认为溢出
          //因为若超过了  int  最大值，就会返回负数
          if  (c  <  0  ||  (size  +=  c)  <  0)
            overflow  =  true;
        }
      }
      //当此次尝试，统计的  sum  值和上次统计的值相同，则说明这段时间内，
      //并没有任何一个  Segment  的结构发生改变，就可以返回最后的统计结果
      if  (sum  ==  last)
        break;
      //不相等，则说明有  Segment  结构发生了改变，则记录最新的结构变化次数之
和  sum，
      //并赋值给 last，用于下次重试的比较。
      last  =  sum;
    }
  }  finally  {
    //如果超过了指定重试次数，则说明表中的所有Segment都被加锁了，因此需要把它们都
解锁
    if  (retries  >  RETRIES_BEFORE_LOCK)  {
      for  (int  j  =  0;  j  <  segments.length;  ++j)
        segmentAt(segments,  j).unlock();
    }
  }
  //若结果溢出，则返回  int  最大值，否则正常返回  size  值
  return  overflow  ?  Integer.MAX_VALUE  :  size;
}
```

其实源码中前两行的注释也说的非常清楚了。我们先采用乐观的方式，认为在统计 size 的过程中，并没有发生 put， remove 等会改变 Segment 结构的操作。但是，如果发生了，就需

要重试。如果重试2次都不成功(执行三次，第一次不能叫做重试)，就只能强制把所有 Segment 都加锁之后，再统计了，以此来得到准确的结果。

ConcurrentHashMap 1.8 源码分析

需要说明的是，JDK 1.8 的 CHM（ConcurrentHashMap） 实现，完全重构了 1.7 。不再有 Segment 的概念，只是为了兼容 1.7 才申明了一下，并没有用到。因此，不再使用分段锁，而是给数组中的每一个头节点（为了方便，以后都叫桶）都加锁，锁的粒度降低了。并且，用的是 Synchronized 锁。

可能有的小伙伴就有疑惑了，不是都说同步锁是重量级锁吗，这样不是会影响并发效率吗?

确实之前同步锁是一个重量级锁，但是在 JDK1.6 之后进行了各种优化之后，它已经不再那么重了。引入了偏向锁，轻量级锁，以及锁升级的概念，而且，据说在更细粒度的代码层面上，同步锁已经可以媲美 Lock 锁，甚至是赶超了。除此之外，它还有很多优点，这里不再展开了。感兴趣的可以自行查阅同步锁的锁升级过程，以及它和 Lock 锁的区别。

在 1.8 CHM 中，底层存储结构和 1.8 的 HashMap 是一样的，都是数组+链表+红黑树。不同的就是，多了一些并发的处理。

文章开头我们提到了，在 1.8 HashMap 中的线程安全问题，就是因为在多个线程同时操作同一个桶的头结点时，会发生值的覆盖情况。那么，顺着这个思路，我们看一下在 CHM 中它是怎么避免这种情况发生的吧。

PS：由于1.8的 CHM 和 HashMap 结构和基本属性变量，还有初始化逻辑都差不多，只是多了一些并发情况需要用到的参数和内部类，因此，不再单独拎出来介绍。在方法中用到的时候，再详细解释。

put()方法

因此，从 put 方法开始，我们看下，它在插入新元素的时候，是怎么保证线程安全的吧。

```
public V put(K key, V value) {
  return putVal(key, value, false);
}

final V putVal(K key, V value, boolean onlyIfAbsent) {
  //可以看到，在并发情况下，key 和 value 都是不支持为空的。
  if (key == null || value == null) throw new NullPointerException();
  //这里和1.8 HashMap 的hash 方法大同小异，只是多了一个操作，如下
  //( h ^ (h >>> 16)) & HASH_BITS;   HASH_BITS = 0x7fffffff;
  // 0x7fffffff ，二进制为 0111 1111 1111 1111 1111 1111 1111 1111 。
  //所以，hash值除了做了高低位异或运算，还多了一步，保证最高位的 1 个 bit 位总是0。
  //这里，我并没有明白它的意图，仅仅是保证计算出来的hash值不超过 Integer 最大值，且不为负数吗。
  //同 HashMap 的hash 方法对比一下，会发现连源码注释都是相同的，并没有多说明其它的。
  //我个人认为意义不大，因为最后 hash 是为了和 capacity -1 做与运算，
而 capacity 最大值为 1<<30，
  //即 0100 0000 0000 0000 0000 0000 0000 0000 ，减1为 0011 1111 1111 1111 1111 1111 1111 1111。
  //即使 hash 最高位为 1(无所谓0)，也不影响最后的结果，最高位也总会是0.
  int hash = spread(key.hashCode());
  //用来计算当前链表上的元素个数
  int binCount = 0;
  for (Node<K,V>[] tab = table;;) {
    Node<K,V> f; int n, i, fh;
    //如果表为空，则说明还未初始化。
```

```
    if (tab == null || (n = tab.length) == 0)
      //初始化表，只有一个线程可以初始化成功。
      tab = initTable();
    //若表已经初始化，则找到当前 key 所在的桶，并且判断是否为空
    else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
      //若当前桶为空，则通过 CAS 原子操作，把新节点插入到此位置，
      //这保证了只有一个线程可以 CAS 成功，其它线程都会失败。
      if (casTabAt(tab, i, null,
              new Node<K,V>(hash, key, value, null)))
        break;                                  // no lock when adding to
  empty bin
    }
    //若所在桶不为空，则判断节点的 hash 值是否为 MOVED（值是-1）
    else if ((fh = f.hash) == MOVED)
      //若为-1，说明当前数组正在进行扩容，则需要当前线程帮忙迁移数据
      tab = helpTransfer(tab, f);
    else {
      V oldVal = null;
      //这里用加同步锁的方式，来保证线程安全，给桶中第一个节点对象加锁
      synchronized (f) {
        //recheck 一下，保证当前桶的第一个节点无变化，后边很多这样类似的操作，不再
赘述
        if (tabAt(tab, i) == f) {
          //如果hash值大于等于0，说明是正常的链表结构
          if (fh >= 0) {
            binCount = 1;
            //从头结点开始遍历，每遍历一次，binCount计数加1
            for (Node<K,V> e = f;; ++binCount) {
              K ek;
              //如果找到了和当前 key 相同的节点，则用新值替换旧值
              if (e.hash == hash &&
                ((ek = e.key) == key ||
                  (ek != null && key.equals(ek)))) {
                oldVal = e.val;
                if (!onlyIfAbsent)
                  e.val = value;
                break;
              }
              Node<K,V> pred = e;
              //若遍历到了尾结点，则把新节点尾插进去
              if ((e = e.next) == null) {
                pred.next = new Node<K,V>(hash, key,
                                value, null);
                break;
              }
            }
          }
          //否则判断是否是树节点。这里提一下，TreeBin只是头结点对TreeNode的再封装
          else if (f instanceof TreeBin) {
            Node<K,V> p;
            binCount = 2;
            if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                value)) != null) {
              oldVal = p.val;
              if (!onlyIfAbsent)
                p.val = value;
            }
          }
        }
      }
      //注意下，这个判断是在同步锁外部，因为 treeifyBin内部也有同步锁，并不影响
```

```
      if (binCount != 0) {
        //如果节点个数大于等于 8，则转化为红黑树
        if (binCount >= TREEIFY_THRESHOLD)
          treeifyBin(tab, i);
        //把旧节点值返回
        if (oldVal != null)
          return oldVal;
        break;
      }
    }
  }
  //给元素个数加 1，并有可能会触发扩容，比较复杂，稍后细讲
  addCount(1L, binCount);
  return null;
}
```

initTable()方法

先看下当数组为空时，是怎么初始化表的。

```
private final Node<K,V>[] initTable() {
  Node<K,V>[] tab; int sc;
  //循环判断表是否为空，直到初始化成功为止。
  while ((tab = table) == null || tab.length == 0) {
    //sizeCtl 这个值有很多情况，默认值为0，
    //当为 -1 时，说明有其它线程正在对表进行初始化操作
    //当表初始化成功后，又会把它设置为扩容阈值
    //当为一个小于 -1 的负数，用来表示当前有几个线程正在帮助扩容(后边细讲)
    if ((sc = sizeCtl) < 0)
      //若 sc 小于0，其实在这里就是-1，因为此时表是空的，不会发生扩容，sc只能为
正数或者-1
      //因此，当前线程放弃 CPU 时间片，只是自旋。
      Thread.yield(); // lost initialization race; just spin
    //通过 CAS 把 sc 的值设置为-1，表明当前线程正在进行表的初始化，其它失败的
线程就会自旋
    else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
      try {
        //重新检查一下表是否为空
        if ((tab = table) == null || tab.length == 0) {
          //如果sc大于0，则为sc，否则返回默认容量 16。
          //当调用有参构造创建 Map 时，sc的值是大于0的。
          int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
          @SuppressWarnings('unchecked')
          //创建数组
          Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
          table = tab = nt;
          //n减去 1/4 n ，即为 0.75n ，表示扩容阈值
          sc = n - (n >>> 2);
        }
      } finally {
        //更新 sizeCtl 为扩容阈值
        sizeCtl = sc;
      }
      //若当前线程初始化表成功，则跳出循环。其它自旋的线程因为判断数组不为空，也会
停止自旋
      break;
    }
  }
  return tab;
}
```

addCount()方法

若 put 方法元素插入成功之后，则会调用此方法，传入参数为 addCount(1L, binCount)。这个方法的目的很简单，就是把整个 table 的元素个数加 1 。但是，实现比较难。

我们先思考一下，如果让我们自己去实现这样的统计元素个数，怎么实现?

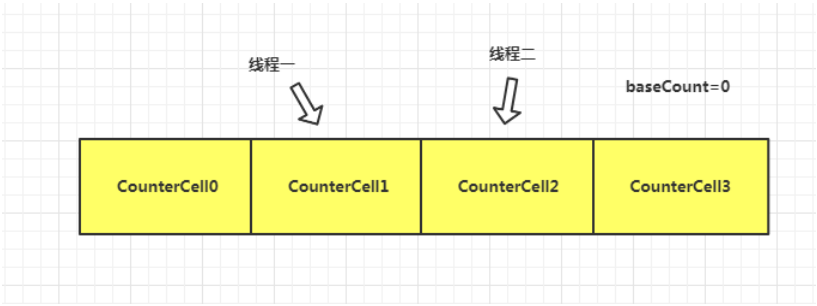
类比 1.8 的 HashMap ，我们可以搞一个 size 变量来存储个数统计。但是，这是在多线程环境下，需要考虑并发的问题。因此，可以把 size 设置为 volatile 的，保证可见性，然后通过 CAS 乐观锁来自增 1。

这样虽然也可以实现。但是，设想一下现在有非常多的线程，都在同一时间操作这个 size 变量，将会造成特别严重的竞争。所以，基于此，这里做了更好的优化。让这些竞争的线程，分散到不同的对象里边，单独操作它自己的数据(计数变量)，用这样的方式尽量降低竞争。到最后需要统计 size 的时候，再把所有对象里边的计数相加就可以了。

上边提到的 size ，在此用 baseCount 表示。分散到的对象用 CounterCell 表示，对象里边的计数变量用 value 表示。注意这里的变量都是 volatile 修饰的。

当需要修改元素数量时，线程会先去 CAS 修改 baseCount 加1，若成功即返回。若失败，则线程被分配到某个 CounterCell ，然后操作 value 加1。若成功，则返回。否则，给当前线程重新分配一个 CounterCell，再尝试给 value 加1。 (这里简略的说，实际更复杂)

CounterCell 会组成一个数组，也会涉及到扩容问题。所以，先画一个示意图帮助理解一下。




```
//线程被分配到的格子
@sun.misc.Contended static final class CounterCell {
  //此格子内记录的 value 值
        volatile long value;
        CounterCell(long x) { value = x; }
}

//用来存储线程和线程生成的随机数的对应关系
static final int getProbe() {
  return UNSAFE.getInt(Thread.currentThread(), PROBE);
}

//  x为1,check代表链表上的元素个数
private final void addCount(long x, int check) {
  CounterCell[] as; long b, s;
  //此处要进入if有两种情况
  //1.数组不为空，说明数组已经被创建好了。
  //2.若数组为空，说明数组还未创建，很有可能竞争的线程非常少，因此就直接 CAS 操作 baseCount
  //若 CAS 成功，则方法跳转到 (2)处，若失败，则需要考虑给当前线程分配一个格子（指CounterCell对象）
  if ((as = counterCells) != null ||
    !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
    CounterCell a; long v; int m;
    //字面意思，是无竞争，这里先标记为 true，表示还没有产生线程竞争
    boolean uncontended = true;
    //这里有三种情况，会进入 fullAddCount 方法
```

```
    //1.若数组为空，进方法 (1)
    //2.ThreadLocalRandom.getProbe() 方法会给当前线程生成一个随机数（可以简单的认为也是一个hash值）
    //然后用随机数与数组长度取模，计算它所在的格子。若当前线程所分配到的格子为空，进方法 (1)。
    //3.若数组不为空，且线程所在格子不为空，则尝试 CAS 修改此格子对应的 value 值加1。
    //若修改成功，则跳转到 (3)，若失败，则把 uncontended 值设为 fasle，说明产生了竞争，然后进方法 (1)
    if (as == null || (m = as.length - 1) < 0 ||
      (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
      !(uncontended =
          U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
      //方法(1)，这个方法的目的是让当前线程一定把 1 加成功。情况更多，更复杂，稍后讲。
      fullAddCount(x, uncontended);
      return;
    }
    //(3)能走到这，说明数组不为空，且修改 baseCount失败，
    //且线程被分配到的格子不为空，且修改 value 成功。
    //但是这里没明白为什么小于等于1，就直接返回了，这里我怀疑之前的方法漏掉了binCount=0的情况。
    //而且此处若返回了，后边怎么判断扩容？（存疑）
    if (check <= 1)
      return;
    //计算总共的元素个数
    s = sumCount();
  }
  //(2)这里用于检查是否需要扩容（下边这部分很多逻辑不懂的话，等后边讲完扩容，再回来看就理解了）
  if (check >= 0) {
    Node<K,V>[] tab, nt; int n, sc;
    //若元素个数达到扩容阈值，且tab不为空，且tab数组长度小于最大容量
    while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
           (n = tab.length) < MAXIMUM_CAPACITY) {
      //这里假设数组长度n就为16，这个方法返回的是一个固定值，用于当做一个扩容的校验标识
      //可以跳转到最后，看详细计算过程，
0000 0000 0000 0000 1000 0000 0001 1011
      int rs = resizeStamp(n);
      //若sc小于0，说明正在扩容
      if (sc < 0) {
            //sc的结构类似这样，1000 0000 0001 1011 0000 0000 0000 0001
        //sc的高16位是数据校验标识，低16位代表当前有几个线程正在帮助扩容,RESIZE_STAMP_SHIFT=16
        //因此判断校验标识是否相等，不相等则退出循环
        //sc == rs + 1,sc == rs + MAX_RESIZERS 这两个应该是用来判断扩容是否已经完成，但是计算方法存疑
        //感兴趣的可以看这个地址，应该是一个 bug ，
        //  https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8214427
        //nextTable=null 说明需要扩容的新数组还未创建完成
        //transferIndex这个参数小于等于0，说明已经不需要其它线程帮助扩容了，
        //但是并不说明已经扩容完成，因为有可能还有线程正在迁移元素。稍后扩容细讲就明白了。
        if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
          sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
          transferIndex <= 0)
          break;
        //到这里说明当前线程可以帮助扩容，因此sc值加一，代表扩容的线程数加1
        if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
          transfer(tab, nt);
```

```
      }
      //当sc大于0，说明sc代表扩容阈值，因此第一次扩容之前肯定走这个分支，用于初始
化新表  nextTable
      //rs<<16
      //1000  0000  0001  1011  0000  0000  0000  0000
      //+2
      //1000  0000  0001  1011  0000  0000  0000  0010
      //这个值，转为十进制就是  -2145714174，用于标识，这是扩容时，初始化新表的状
态，
      //扩容时，需要用到这个参数校验是否所有线程都全部帮助扩容完成。
      else  if  (U.compareAndSwapInt(this,  SIZECTL,  sc,
                      (rs  <<  RESIZE_STAMP_SHIFT)  +  2))
        //扩容，第二个参数代表新表，传入null，则说明是第一次初始化新表(nextTable)
        transfer(tab,  null);
      s  =  sumCount();
    }
  }
}

//计算表中的元素总个数
final  long  sumCount()  {
  CounterCell[]  as  =  counterCells;  CounterCell  a;
  //baseCount，以这个值作为累加基准
  long  sum  =  baseCount;
  if  (as  !=  null)  {
    //遍历  counterCells  数组，得到每个对象中的value值
    for  (int  i  =  0;  i  <  as.length;  ++i)  {
      if  ((a  =  as[i])  !=  null)
        //累加  value  值
        sum  +=  a.value;
    }
  }
  //此时得到的就是元素总个数
  return  sum;
}

//扩容时的校验标识
static  final  int  resizeStamp(int  n)  {
  return  Integer.numberOfLeadingZeros(n)  |  (1  <<  (RESIZE_STAMP_BITS  -  1));
}

//Integer.numberOfLeadingZeros方法的作用是返回  n  的最高位为1的前面的0的个数
//n=16，
0000  0000  0000  0000  0000  0000  0001  0000
//前面有27个0，即27
0000  0000  0000  0000  0000  0000  0001  1011
//RESIZE_STAMP_BITS为16，然后  1<<(16-1)，即  1<<15
0000  0000  0000  0000  1000  0000  0000  0000
//它们做或运算，得到  rs  的值
0000  0000  0000  0000  1000  0000  0001  1011
```

fullAddCount()方法

上边的 addCount 方法还没完，别忘了有可能元素个数加 1 的操作还未成功，就走到 fullAddCount 这个方法了。看方法名，就知道了，全力增加计数值，一定要成功（奥利给）。这个方法和扩容迁移方法是最难的，保持耐心~

```
//传过来的参数分别为  1  ,  false
private  final  void  fullAddCount(long  x,  boolean  wasUncontended)  {
  int  h;
  //如果当前线程的随机数为0，则强制初始化一个值
  if  ((h  =  ThreadLocalRandom.getProbe())  ==  0)  {
```

```
    ThreadLocalRandom.localInit();            //  force  initialization
    h  =  ThreadLocalRandom.getProbe();
    //此时把  wasUncontended  设为true，认为无竞争
    wasUncontended  =  true;
  }
  //用来表示比  contend（竞争）更严重的碰撞，若为true，表示可能需要扩容，以减少碰撞冲突
  boolean  collide  =  false;                                  //  True  if  last  slot  nonempty
  //循环内，外层if判断分三种情况，内层判断又分为六种情况
  for  (;;)  {
    CounterCell[]  as;  CounterCell  a;  int  n;  long  v;
    //1. 若counterCells数组不为空。     建议先看下边的2和3两种情况，再回头看这个。
    if  ((as  =  counterCells)  !=  null  &&  (n  =  as.length)  >  0)  {
      //  (1)  若当前线程所在的格子（CounterCell对象）为空
      if  ((a  =  as[(n  -  1)  &  h])  ==  null)  {
        if  (cellsBusy  ==  0)  {
          //若无锁，则乐观的创建一个 CounterCell 对象。
          CounterCell  r  =  new  CounterCell(x);
          //尝试加锁
          if  (cellsBusy  ==  0  &&
            U.compareAndSwapInt(this,  CELLSBUSY,  0,  1))  {
            boolean  created  =  false;
            //加锁成功后，再  recheck  一下数组是否不为空，且当前格子为空
            try  {
              CounterCell[]  rs;  int  m,  j;
              if  ((rs  =  counterCells)  !=  null  &&
                (m  =  rs.length)  >  0  &&
                rs[j  =  (m  -  1)  &  h]  ==  null)  {
                //把新创建的对象赋值给当前格子
                rs[j]  =  r;
                created  =  true;
              }
            }  finally  {
              //手动释放锁
              cellsBusy  =  0;
            }
            //若当前格子创建成功，且上边的赋值成功，则说明加1成功，退出循环
            if  (created)
              break;
            //否则，继续下次循环
            continue;                      //  Slot  is  now  non-empty
          }
        }
        //若cellsBusy=1，说明有其它线程抢锁成功。或者若抢锁的 CAS 操作失败，都会走到这里，
        //则当前线程需跳转到(9)重新生成随机数，进行下次循环判断。
        collide  =  false;
      }
      /**
      *后边这几种情况，都是数组和当前随机到的格子都不为空的情况。
      *且注意每种情况，若执行成功，且不break，continue，则都会执行(9)，重新生成随机数，进入下次循环判断
      */
      //  (2)  到这，说明当前方法在被调用之前已经  CAS  失败过一次，若不明白可回头看下  addCount  方法，
      //为了减少竞争，则跳转到⑨处重新生成随机数，并把  wasUncontended  设置为  true  ，认为下一次不会产生竞争
      else  if  (!wasUncontended)              //  CAS  already  known  to  fail
        wasUncontended  =  true;           //  Continue  after  rehash
```

```
      // (3) 若 wasUncontended 为 true 无竞争，则尝试一次 CAS。若成功，则结束循环，若失败则判断后边的 (4)(5)(6)。
      else if (U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))
        break;
      // (4) 结合 (6) 一起看，(4)(5)(6)都是 wasUncontended=true，且CAS修改value失败的情况。
      //若数组有变化，或者数组长度大于等于当前CPU的核心数，则把 collide 改为 false
      //因为数组若有变化，说明是由扩容引起的；长度超限，则说明已经无法扩容，只能认为无碰撞。
      //这里很有意思，认真思考一下，当扩容超限后，则会达到一个平衡，即 (4)(5) 反复执行，直到 (3) 中CAS成功，跳出循环。
      else if (counterCells != as || n >= NCPU)
        collide = false;                        // At max size or stale
      // (5) 若数组无变化，且数组长度小于CPU核心数时，且 collide 为 false，就把它改为 true，说明下次循环可能需要扩容
      else if (!collide)
        collide = true;
      // (6) 若数组无变化，且数组长度小于CPU核心数时，且 collide 为 true，说明冲突比较严重，需要扩容了。
      else if (cellsBusy == 0 &&
           U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
        try {
          //recheck
          if (counterCells == as) {// Expand table unless stale
            //创建一个容量为原来两倍的数组
            CounterCell[] rs = new CounterCell[n << 1];
            //转移旧数组的值
            for (int i = 0; i < n; ++i)
              rs[i] = as[i];
            //更新数组
            counterCells = rs;
          }
        } finally {
          cellsBusy = 0;
        }
        //认为扩容后，下次不会产生冲突了，和(4)处逻辑照应
        collide = false;
        //当次扩容后，就不需要重新生成随机数了
        continue;                                   // Retry with expanded table
      }
      // (9)，重新生成一个随机数，进行下一次循环判断
      h = ThreadLocalRandom.advanceProbe(h);
    }
    //2.这里的 cellsBusy 参数非常有意思，是一个volatile的 int值，用来表示自旋锁的标志，
    //可以类比 AQS 中的 state 参数，用来控制锁之间的竞争，并且是独占模式。简化版的AQS。
    //cellsBusy 若为0，说明无锁，线程都可以抢锁，若为1，表示已经有线程拿到了锁，则其它线程不能抢锁。
    else if (cellsBusy == 0 && counterCells == as &&
           U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
      boolean init = false;
      try {
        //这里再重新检测下 counterCells 数组引用是否有变化
        if (counterCells == as) {
          //初始化一个长度为 2 的数组
          CounterCell[] rs = new CounterCell[2];
          //根据当前线程的随机数值，计算下标，只有两个结果 0 或 1，并初始化对象
          rs[h & 1] = new CounterCell(x);
```

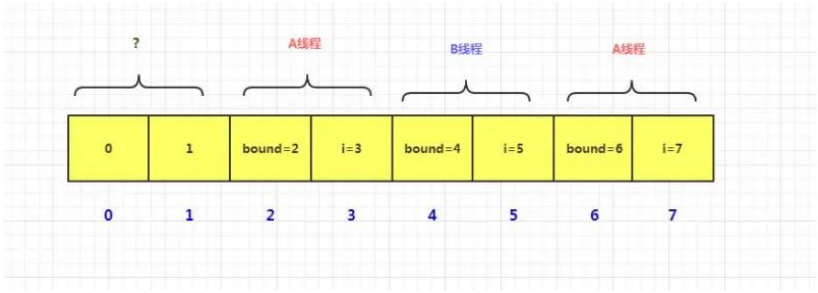
```
            //更新数组引用
            counterCells = rs;
            //初始化成功的标志
            init = true;
          }
        } finally {
          //别忘了，需要手动解锁。
          cellsBusy = 0;
        }
        //若初始化成功，则说明当前加1的操作也已经完成了，则退出整个循环。
        if (init)
          break;
      }
      //3.到这，说明数组为空，且 2 抢锁失败，则尝试直接去修改 baseCount 的值，
      //若成功，也说明加1操作成功，则退出循环。
      else if (U.compareAndSwapLong(this, BASECOUNT, v = baseCount, v + x))
        break;                                                    // Fall back on
    using base
    }
}
```

不得不佩服 Doug Lea 大神，思维这么缜密，如果是我的话，直接一个 CAS 完事。（手动摊手~）

transfer()方法

需要说明的一点是，虽然我们一直在说帮助扩容，其实更准确的说应该是帮助迁移元素。因为扩容的第一次初始化新表（扩容后的新表）这个动作，只能由一个线程完成。其他线程都是在帮助迁移元素到新数组。

这里还是先看下迁移的示意图，帮助理解。



扩容

为了方便，上边以原数组长度 8 为例。在元素迁移的时候，所有线程都遵循从后向前推进的规则，即如图A线程是第一个进来的线程，会从下标为7的位置，开始迁移数据。

而且当前线程迁移时会确定一个范围，限定它此次迁移的数据范围，如图 A 线程只能迁移 bound=6到 i=7 这两个数据。

此时，其它线程就不能迁移这部分数据了，只能继续向前推进，寻找其它可以迁移的数据范围，且每次推进的步长为固定值 stride（此处假设为2）。如图中 B线程发现 A 线程正在迁移 6,7的数据，因此只能向前寻找，然后迁移 bound=4 到 i=5 的这两个数据。

当每个线程迁移完成它的范围内数据时，都会继续向前推进。那什么时候是个头呢?

这就需要维护一个全局的变量 transferIndex，来表示所有线程总共推进到的元素下标位置。如图，线程 A 第一次迁移成功后又向前推进，然后迁移2,3 的数据。此时，若没有其他线程在帮助迁移，则 transferIndex 即为2。

剩余部分等待下一个线程来迁移，或者有任何的 A 和B线程已经迁移完成，也可以推进到这里帮助迁移。直到 transferIndex=0 。（会做一些其他校验来判断是否迁移全部完成，看代码）。

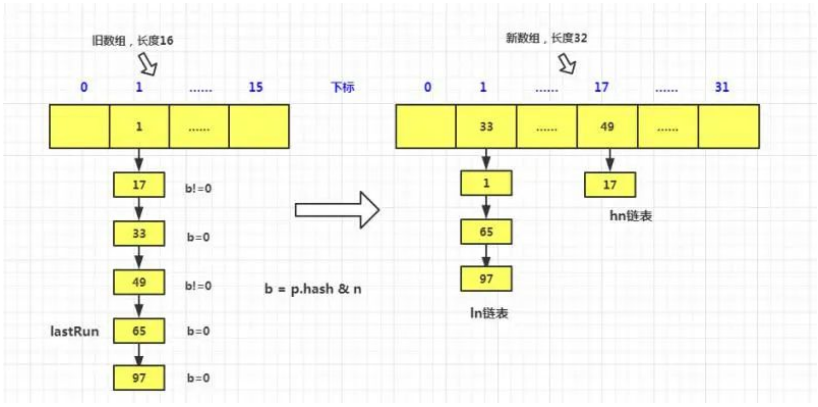
```
//这个类是一个标志，用来代表当前桶（数组中的某个下标位置）的元素已经全部迁移完成
static final class ForwardingNode<K,V> extends Node<K,V> {
  final Node<K,V>[] nextTable;
  ForwardingNode(Node<K,V>[] tab) {
    //把当前桶的头结点的 hash 值设置为 -1，表明已经迁移完成，
    //这个节点中并不存储有效的数据
    super(MOVED, null, null, null);
    this.nextTable = tab;
  }
}

//迁移数据
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
  int n = tab.length, stride;
  //根据当前CPU核心数，确定每次推进的步长，最小值为16.（为了方便我们以2为例）
  if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_S
TRIDE)
    stride = MIN_TRANSFER_STRIDE; // subdivide range
  //从 addCount 方法，只会有一个线程跳转到这里，初始化新数组
  if (nextTab == null) {                        // initiating
    try {
      @SuppressWarnings('unchecked')
      //新数组长度为原数组的两倍
      Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
      nextTab = nt;
    } catch (Throwable ex) {            // try to cope with OOME
      sizeCtl = Integer.MAX_VALUE;
      return;
    }
    //用 nextTable 指代新数组
    nextTable = nextTab;
    //这里就把推进的下标值初始化为原数组长度（以16为例）
    transferIndex = n;
  }
  //新数组长度
  int nextn = nextTab.length;
  //创建一个标志类
  ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
  //是否向前推进的标志
  boolean advance = true;
  //是否所有线程都全部迁移完成的标志
  boolean finishing = false; // to ensure sweep before committing nextTa
b
  //i 代表当前线程正在迁移的桶的下标，bound代表它本次可以迁移的范围下限
  for (int i = 0, bound = 0;;) {
    Node<K,V> f; int fh;
    //需要向前推进
    while (advance) {
      int nextIndex, nextBound;
      //(1) 先看 (3) 。i每次自减 1，直到 bound。若超过bound范围，或者finishing
标志为true，则不用向前推进。
      //若未全部完成迁移，且 i 并未走到 bound，则跳转到 (7)，处理当前桶的元素迁
移。
      if (--i >= bound || finishing)
        advance = false;
      //(2) 每次执行，都会把 transferIndex 最新的值同步给 nextIndex
      //若 transferIndex小于等于0，则说明原数组中的每个桶位置，都有线程在处理迁移
```

了，
//于是，需要跳出while循环，并把 i设为 -1，以跳转到④判断在处理的线程是否已经全部完成。
else if ((nextIndex = transferIndex) <= 0) {
i = -1;
advance = false;
}
//(3) 第一个线程会先走到这里，确定它的数据迁移范围。(2)处会更新 nextIndex为transferIndex 的最新值
//因此第一次 nextIndex=n=16，nextBound代表当次迁移的数据范围下限，减去步长即可，
//所以，第一次时，nextIndex=16，nextBound=16-2=14。后续，每次都会间隔一个步长。
else if (U.compareAndSwapInt
(this, TRANSFERINDEX, nextIndex,
nextBound = (nextIndex > stride ?
nextIndex - stride : 0))) {
//bound代表当次数据迁移下限
bound = nextBound;
//第一次的i为15，因为长度16的数组，最后一个元素的下标为15
i = nextIndex - 1;
//表明不需要向前推进，只有当把当前范围内的数据全部迁移完成后，才可以向前推进
advance = false;
}
}
//(4)
if (i < 0 || i >= n || i + n >= nextn) {
int sc;
//若全部线程迁移完成
if (finishing) {
nextTable = null;
//更新table为新表
table = nextTab;
//扩容阈值改为原来数组长度的 3/2 ，即新长度的 3/4，也就是新数组长度的0.75倍
sizeCtl = (n << 1) - (n >>> 1);
return;
}
//到这，说明当前线程已经完成了自己的所有迁移（无论参与了几次迁移），
//则把 sc 减1，表明参与扩容的线程数减少 1。
if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
//在 addCount 方法最后，我们强调，迁移开始时，会设置 sc=(rs << RESIZE_STAMP_SHIFT) + 2
//每当有一个线程参与迁移，sc 就会加 1，每当有一个线程完成迁移，sc 就会减1。
//因此，这里就是去校验当前 sc 是否和初始值是否相等。相等，则说明全部线程迁移完成。
if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
return;
//只有此处，才会把finishing 设置为true。
finishing = advance = true;
//这里非常有意思，会把 i 从 -1 修改为16，
//目的就是，让 i 再从后向前扫描一遍数组，检查是否所有的桶都已被迁移完成，参看 (6)
i = n; // recheck before commit
}
}
//(5) 若i的位置元素为空，则说明当前桶的元素已经被迁移完成，就把头结点设置为fwd标志。
else if ((f = tabAt(tab, i)) == null)

```
      advance = casTabAt(tab, i, null, fwd);
    //(6) 若当前桶的头结点是 ForwardingNode ，说明迁移完成，则向前推进
    else if ((fh = f.hash) == MOVED)
      advance = true; // already processed
    //(7) 处理当前桶的数据迁移。
    else {
      synchronized (f) {    //给头结点加锁
        if (tabAt(tab, i) == f) {
          Node<K,V> ln, hn;
          //若hash值大于等于0，则说明是普通链表节点
          if (fh >= 0) {
            int runBit = fh & n;
            //这里是 1.7 的 CHM 的 rehash 方法和 1.8 HashMap的 resize 方法的结合
体。
            //会分成两条链表，一条链表和原来的下标相同，另一条链表是原来的下标加数
组长度的位置
            //然后找到 lastRun 节点，从它到尾结点整体迁移。
            //lastRun前边的节点则单个迁移，但是需要注意的是，这里是头插法。
            //另外还有一点和1.7不同，1.7 lastRun前边的节点是复制过去的，而这里是直
接迁移的，没有复制操作。
            //所以，最后会有两条链表，一条链表从 lastRun到尾结点是正序的，而
lastRun之前的元素是倒序的，
            //另外一条链表，从头结点开始就是倒叙的。看下图。
            Node<K,V> lastRun = f;
            for (Node<K,V> p = f.next; p != null; p = p.next) {
              int b = p.hash & n;
              if (b != runBit) {
                runBit = b;
                lastRun = p;
              }
            }
            if (runBit == 0) {
              ln = lastRun;
              hn = null;
            }
            else {
              hn = lastRun;
              ln = null;
            }
            for (Node<K,V> p = f; p != lastRun; p = p.next) {
              int ph = p.hash; K pk = p.key; V pv = p.val;
              if ((ph & n) == 0)
                ln = new Node<K,V>(ph, pk, pv, ln);
              else
                hn = new Node<K,V>(ph, pk, pv, hn);
            }
            setTabAt(nextTab, i, ln);
            setTabAt(nextTab, i + n, hn);
            setTabAt(tab, i, fwd);
            advance = true;
          }
          //树节点
          else if (f instanceof TreeBin) {
            TreeBin<K,V> t = (TreeBin<K,V>)f;
            TreeNode<K,V> lo = null, loTail = null;
            TreeNode<K,V> hi = null, hiTail = null;
            int lc = 0, hc = 0;
            for (Node<K,V> e = t.first; e != null; e = e.next) {
              int h = e.hash;
              TreeNode<K,V> p = new TreeNode<K,V>
                (h, e.key, e.val, null, null);
```

```
            if ((h & n) == 0) {
              if ((p.prev = loTail) == null)
                lo = p;
              else
                loTail.next = p;
              loTail = p;
              ++lc;
            }
            else {
              if ((p.prev = hiTail) == null)
                hi = p;
              else
                hiTail.next = p;
              hiTail = p;
              ++hc;
            }
          }
          ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
            (hc != 0) ? new TreeBin<K,V>(lo) : t;
          hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
            (lc != 0) ? new TreeBin<K,V>(hi) : t;
          setTabAt(nextTab, i, ln);
          setTabAt(nextTab, i + n, hn);
          setTabAt(tab, i, fwd);
          advance = true;
        }
      }
    }
  }
}
```

迁移后的新数组链表方向示意图，以 runBit =0 为例。



helpTransfer()方法

最后再看 put 方法中的这个方法，就比较简单了。

```
final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
  Node<K,V>[] nextTab; int sc;
  //头结点为 ForwardingNode ，并且新数组已经初始化
  if (tab != null && (f instanceof ForwardingNode) &&
    (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {
    int rs = resizeStamp(tab.length);
    while (nextTab == nextTable && table == tab &&
           (sc = sizeCtl) < 0) {
      //若校验标识失败，或者已经扩容完成，或推进下标到头，则退出
      if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
        sc == rs + MAX_RESIZERS || transferIndex <= 0)
        break;
```

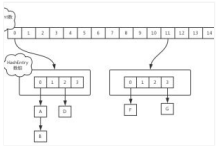
```
        //当前线程需要帮助迁移，sc值加1
        if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
          transfer(tab, nextTab);
          break;
        }
      }
      return nextTab;
    }
    return table;
  }
```

JDK1.8 的 CHM 最主要的逻辑基本上都讲完了，其它方法原理类同。1.8 的 ConcurrentHashMap 实现原理还是比较简单的，但是代码实现比较复杂。相对于 1.7 来说，锁的粒度降低了，效率也提高了。

上一篇：Docker Jenkins Nginx Spring Boot 自动化部署项目

Java7/8中的HashMap和ConcurrentHashMap源码分析，看了都说好

哈希表，均摊复杂度是O(1)，因为第一步通过数组索引找到数组位置是O(1)，然后到链表中查找元素的均摊复杂度是O(size/length)，size为元...

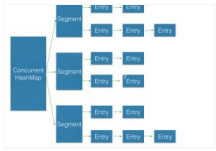


谈谈 ConcurrentHashMap1.7 和 1.8 的不同实现

if (casTabAt(tab, i, null, new Node(hash, key, value, null)))1、初始化时counterCells为空，在并发量很高时，如果存在两个线程同时...

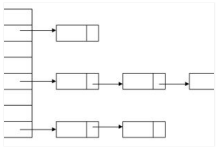
深入分析ConcurrentHashMap

深入分析ConcurrentHashMap深入分析ConcurrentHashMap 术语定义。ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。一个C...



[Java并发包学习八]深度剖析ConcurrentHashMap | winwill2012博客

int h = hash(key);这是普通人能够想到的方案，但是牛逼的作者还有一个更好的Idea：先给3次机会，不lock所有的Segment，遍历所有Segment...



深入剖析ConcurrentHashMap(2)(转，分析得不错)

如果线程b先执行了clear，清空了一部分segment的时候，线程a执行了put且正好把“first”放入了“清空过”的segment中，而把“second”放...