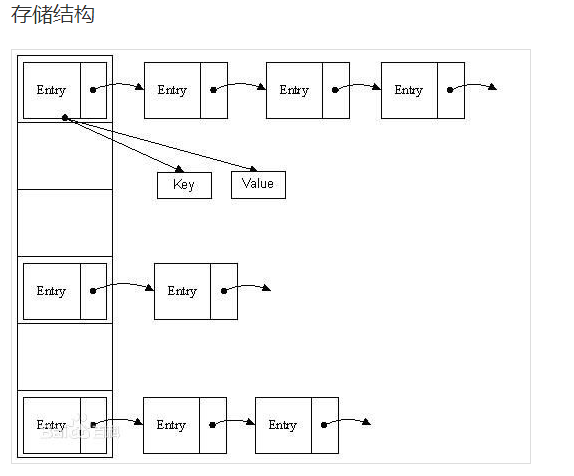
1.7的HashMap设计



1 1.8的HashMap设计

1.1 整体概览

HashMap采用的是**数组+链表+红黑树**的形式。

数组是可以扩容的，链表也是转化为红黑树的，这2种方式都可以承载更多的数据。

用户可以设置的参数：初始总容量默认16，默认的加载因子0.75

初始的数组个数默认是16 容量X加载因子=阈值(如第一次扩容16\*0.75=12个)

一旦目前容量超过该阈值，则执行扩容操作。

什么时候扩容？

* 1 当前容量（数组+链表的总个数，也就是**键值对**）超过阈值
* 2 当链表中元素个数超过默认设定（8个），当数组的大小还未超过64的时候，此时进行数组的扩容**（扩容完会重新分桶）**，如果超过则将链表转化成红黑树。

什么时候链表转化为红黑树？（上面已经提到了）

* 当数组大小已经超过64并且链表中的元素个数超过默认设定（8个）时，将链表转化为红黑树。

**目前形象的表示数组中的一个元素称为一个桶。**

## 1.2 put过程

* 根据key计算出hash值
* hash值&（数组长度-1）得到所在数组的index
  + 如果该index位置的Node元素不存在，则直接创建一个新的Node
  + 如果该index位置的Node元素是TreeNode类型即红黑树类型了，则直接按照红黑树的插入方式进行插入
  + 如果该index位置的Node元素是非TreeNode类型则，则按照链表的形式进行插入操作

链表插入操作完成后，判断是否超过阈值TREEIFY\_THRESHOLD（默认是8），超过则要么数组扩容**（看是否超过64.）**要么链表转化成红黑树

* 判断当前总容量是否超出阈值，如果超出则执行扩容。

1.3 扩容过程

按照2倍扩容的方式，那么就需要将之前的所有元素全部重新按照2倍桶的长度重新计算所在桶。

你能谈谈他的容量问题吗？

我：hashmap的初始容量是16，内部有一个负载因子初始为0.75，当数组中元素达到初始值\*负载因子时，就会进行扩容。方法是使用一个新的数组，**数组大小为原来的两倍。**当然负载因子的大小是可调的，当我们有足够的内存而且非常看重时间的时候，可以将负载因子减小一点，反之可以增加负载因子。

那为什么初始容量是16？

* 我：在计算数组索引位置的时候, hash值&（数组长度-1）得到所在数组的index

采用两次has，h第一次是调用hashcode()方法得到h，第二次是用h&(length-1)。

当length为偶数的时候(length-1)得到的二进制的最后一位为1，&运算后即能得到奇数又能得到偶数。而length为奇数仅仅能得到偶数，这样浪费了一半空间。

这里扩容为啥是2倍？

因为2倍的话，更加容易计算他们所在的桶，并且各自不会相互干扰。

如原桶长度是4，现在桶长度是8，那么

* 桶0中的元素会被分到桶0和桶4中
* 桶1中的元素会被分到桶1和桶5中
* 桶2中的元素会被分到桶2和桶6中
* 桶3中的元素会被分到桶3和桶7中

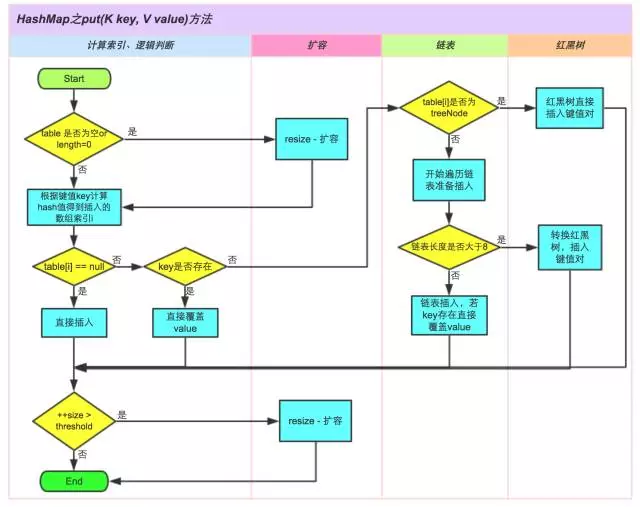
为啥是这样呢？

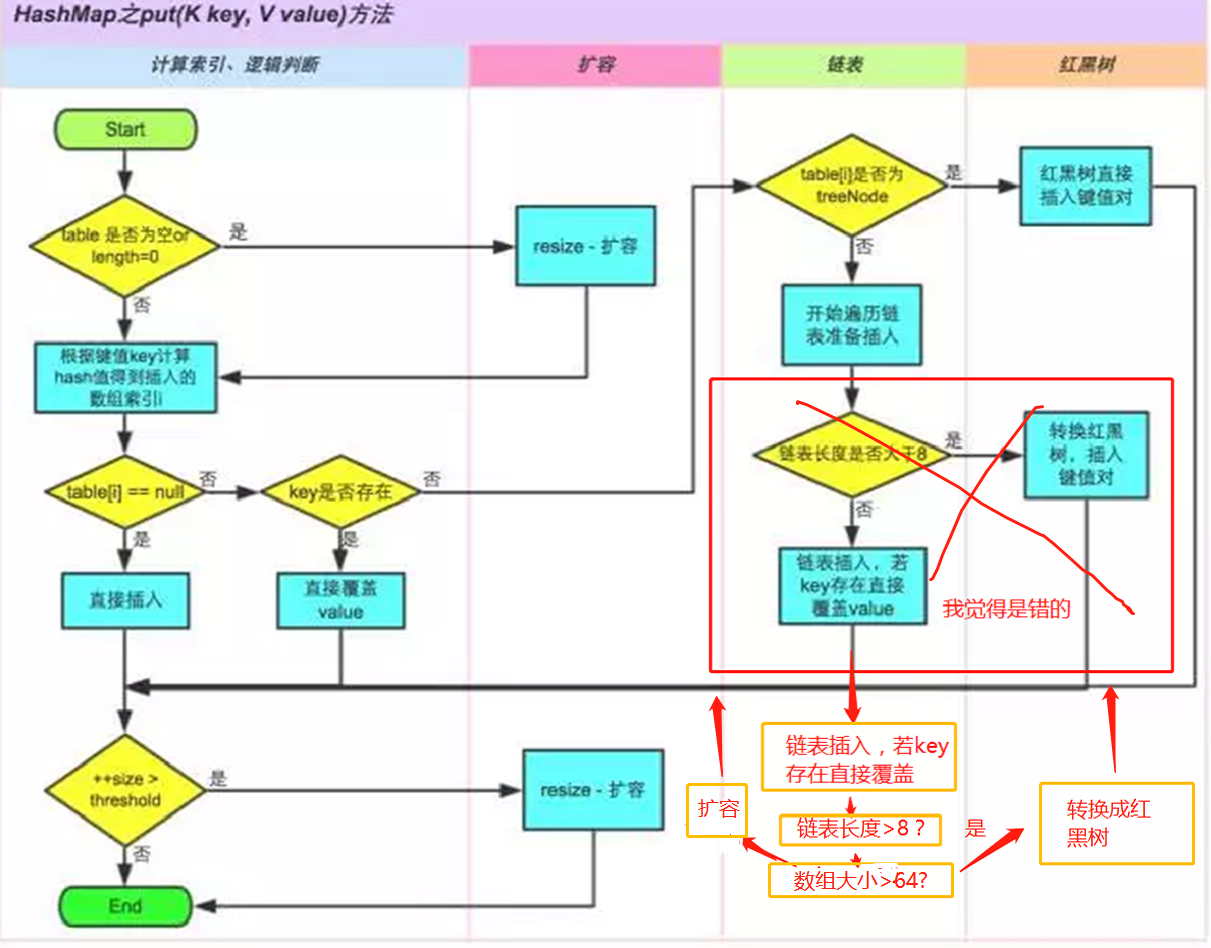
桶0中的元素的hash值后2位必然是00，这些hash值可以根据后3位000或者100分成2类数据（0，4一类，8，12一类）。他们分别&（8-1）即&111,则后3位为000的在桶0中，后3位为100的必然在桶4中。其他同理，也就是说桶4和桶0重新瓜分了原来桶0中的元素。

**如果换成其他倍数，那么瓜分就比较混乱了。**

这样在瓜分这些数据的时候，只需要先把这些数据分类，如上述桶0中分成000和100 2类，然后直接构成新的链表，分类完毕后，直接将新的链表挂在对应的桶下即可

当length为合数［二的幂方］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，这样保证了索引能均匀分布。

综合来说取16是为了增加桶数组的利用率和减少hash碰撞，还有提升扩容时分配元素的方便性。



1.4 get过程

get过程比较简单

* 根据key计算出hash值
* hash值&（数组长度-1）得到所在数组的index
  + 如果要找的key就是上述数组index位置的元素，直接返回该元素的值
  + 如果该数组index位置元素是TreeNode类型，则按照红黑树的查询方式来进行查找
  + 如果该数组index位置元素非TreeNode类型，则按照链表的方式来进行遍历查询
* 在多线程的环境下，使用hashmap会出现什么情况，怎么解决呢？
* hashmap是非线程安全的，可以使用ConcurrentHashMap。

# 2 1.7的ConcurrentHashMap设计

ConcurrentHashMap是线程安全，通过分段锁的方式提高了并发度。分段Segment是一开始就确定的了，后期不能再进行扩容的。

其中的段Segment继承了重入锁ReentrantLock，有了锁的功能，同时含有类似HashMap中的数组加链表结构**（这里没有使用红黑树）**

虽然Segment的个数是不能扩容的，但是单个Segment里面的数组是可以扩容的。

jdk1.7中采用Segment + HashEntry的方式进行实现，结构如下：

## http://upload-images.jianshu.io/upload_images/2184951-af57d9d50ae9f547.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240

## 2.1 整体概览

## ConcurrentHashMap有3个参数：

* initialCapacity：初始总容量，默认16
* loadFactor：加载因子，默认0.75
* concurrencyLevel：并发级别，默认16

segment的个数即ssize（段数）

取大于等于并发级别的最小的2的幂次。如concurrencyLevel=16，那么sszie=16,如concurrencyLevel=10，那么ssize=16

单个segment的初始容量cap（容量/段数，如果有余数则向上+1）

c=initialCapacity/ssize,并且可能需要+1。如15/7=2，那么c要取3，如16/8=2，那么c取2，c可能是一个任意值，那么同上述一样，cap取的值就是大于等于c的最下2的幂次。最小值要求是2

单个segment的阈值threshold（cap\*loadFactor）

如2\*0.75=1.5。

* 所以默认情况下，
* initialCapacity：初始总容量，默认16
* loadFactor：加载因子，默认0.75
* concurrencyLevel：并发级别，默认16

segment的个数sszie=16,每个segment的初始容量cap=2（最少为2），单个segment的阈值threshold=1

## 2.2 put过程

* 首先根据key计算出一个hash值，找到对应的Segment
* 调用Segment的lock方法，为后面的put操作加锁
* 根据key计算出hash值，找到Segment中数组中对应index的链表，并将该数据放置到该链表中
* 判断当前Segment包含元素的数量大于阈值，则Segment进行扩容

2.4 get过程

* 根据key计算出对应的segment
* 再根据key计算出对应segment中数组的index
* 最终遍历上述index位置的链表，查找出对应的key的value

#### 2.5size实现

因为ConcurrentHashMap是可以并发插入数据的，所以在准确计算元素时存在一定的难度，一般的思路是统计每个Segment对象中的元素个数，然后进行累加，但是这种方式计算出来的结果并不一样的准确的，因为在计算后面几个Segment的元素个数时，已经计算过的Segment同时可能有数据的插入或则删除，在1.7的实现中，采用了如下方式：

先采用不加锁的方式，连续计算元素的个数，最多计算3次：  
1、如果前后两次计算结果相同，则说明计算出来的元素个数是准确的；  
2、如果前后两次计算结果都不同，则给每个Segment进行加锁（限制无法插入或者删除数据），再计算一次元素的个数；

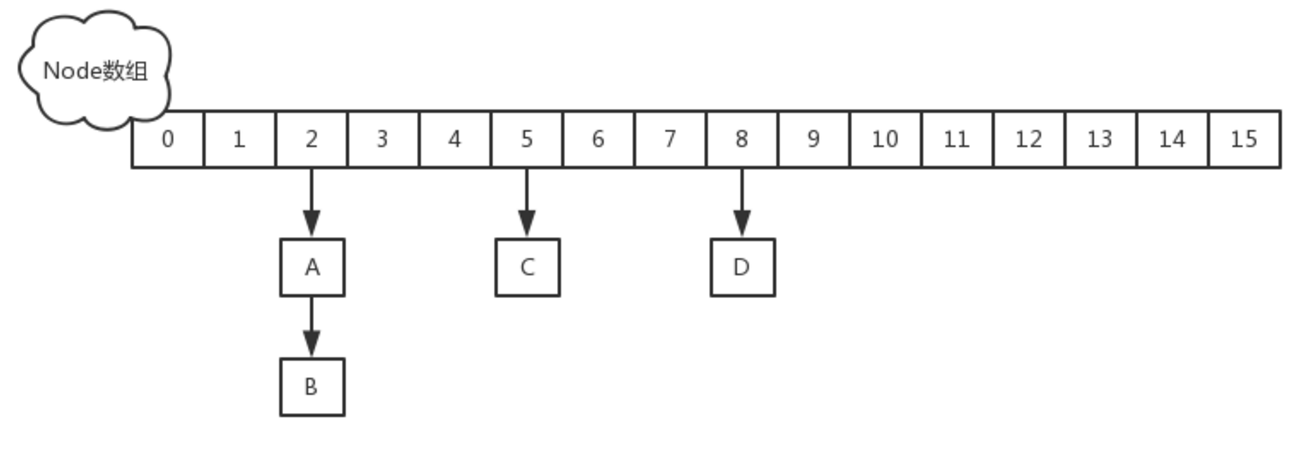
# 3 1.8的ConcurrentHashMap设计

1.8的ConcurrentHashMap摒弃了1.7的segment（锁段）设计，而是启用了一种全新的方式实现,**利用CAS算法**,在1.8HashMap的基础上实现了线程安全的版本，即也是采用**数组+链表+红黑树**的形式。

数组可以扩容，链表可以转化为红黑树

看完ConcurrentHashMap整个类的源码，给自己的感觉就是和HashMap的实现基本一模一样，当有修改操作时借助了synchronized来对table[i]进行锁定保证了线程安全以及使用了CAS来保证原子性操作，其它的基本一致.

例如：ConcurrentHashMap的get(int key)方法的实现思路为：根据key的hash值找到其在table所对应的位置i,然后在table[i]位置所存储的链表(或者是树)进行查找是否有键为key的节点，如果有，则返回节点对应的value，否则返回null。思路是不是很熟悉，是不是和HashMap中该方法的思路一样。所以，如果你也在看ConcurrentHashMap的源码，不要害怕，思路还是原来的思路，只是多了些许东西罢了。



## 3.1 整体概览(JDK1.8)

sizeCtl最重要的属性之一，看源码之前，这个属性表示什么意思，一定要记住。

**private transient volatile int sizeCtl;//控制标识符//(  transient是Java语言的关键字，用来表示一个域不是该对象串行化的一部分。当一个对象被串行化的时候，transient型变量的值不包括在串行化的表示中，然而非transient型的变量是被包括进去的。  )**

**sizeCtl**是控制标识符，不同的值表示不同的意义。

* 负数代表正在进行**初始化或扩容**操作 ,其中-1代表正在初始化 ,-N 表示有N-1个线程正在进行扩容操作
* 正数或0代表hash表还没有被初始化，这个数值表示初始化或下一次进行扩容的大小，类似于扩容阈值。它的值始终是当前ConcurrentHashMap容量的0.75倍，表示阈值，实际容量>=sizeCtl，则扩容。

**改进一：取消segments字段，直接采用transient volatile HashEntry<K,V>[] table保存数据，采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。**

**改进二：将原先table数组＋单向链表的数据结构，变更为table数组＋单向链表＋红黑树的结构。对于hash表来说，最核心的能力在于将key hash之后能均匀的分布在数组中。如果hash之后散列的很均匀，那么table数组中的每个队列长度主要为0或者1。但实际情况并非总是如此理想，虽然ConcurrentHashMap类默认的加载因子为0.75，但是在数据量过大或者运气不佳的情况下，还是会存在一些队列长度过长的情况，如果还是采用单向列表方式，那么查询某个节点的时间复杂度为O(n)；因此，对于个数超过8(默认值)的列表，jdk1.8中采用了红黑树的结构，那么查询的时间复杂度可以降低到O(logN)，可以改进性能。**

为了说明以上2个改动，看一下put操作是如何实现的。

其实在我看来，就是两处改进：

1.在未初始化的时候，采用**CAS**初始化，这样可以避免多个key相同同时 初始化导致某些先来的键值对丢失。

2.在检测到容器在扩容的时候，不做任何的操作，等待新表再进一步操作。这样既可以解决多个线程同时导致扩容所带来的链表死循环问题；也可以解决put的同时扩容，旧表在赋值给新表后，另外线程在get到null值。

3.最后在每个真正put的操作都加锁，可以避免由于同时put导致某个位置重叠之类的问题。

3.2 Put 过程

final V putVal(K key, V value, boolean onlyIfAbsent) {

if (key == null || value == null) throw new NullPointerException();

int hash = spread(key.hashCode());

int binCount = 0;

for (Node<K,V>[] tab = table;;) {

**Node<K,V> f;** int n, i, fh;

// **如果table为空或者长度为0，初始化**；否则，根据hash值计算得到**数组索引i**，如果tab[i]为空，直接新建节点Node即可（此处用cas写入新的数值，大概就是是null则新建Node）。注：**tab[i]实质为链表或者红黑树的首节点。**

if (tab == null || (n = tab.length) == 0)

tab = initTable();

else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {

**if (casTabAt(tab, i, null,**

**new Node<K,V>(hash, key, value, null)))**

**break; // no lock w**hen adding to empty bin

}

// 如果tab[i]不为空并且hash值为MOVED，说明该链表正在进行transfer操作，返回扩容完成后的table。（**sizeCtl**是控制标识符，此处**MOVED=-1**）//在hashmap有可能会因为取得旧表而get null，在这儿解决了 。因为这儿是发现扩容就等待新表再做下一步的操作**(表是整个旧表，而锁只是锁住某一部分。所以这儿要等待新表)。**

else if ((fh = f.hash) == MOVED)

tab = helpTransfer(tab, f);

else {

V oldVal = null;

// 针对**首个节点进行加锁操作**，而不是segment，进一步减少线程冲突。1.7是整个segment加锁，一个segment又有很多个数组节点。

**synchronized (f)** {

if (tabAt(tab, i) == f) {

if (fh >= 0) {

binCount = 1;

for (Node<K,V> e = f;; ++binCount) {

K ek;

// 如果在链表中找到值为key的节点e，直接设置e.val = value即可。

if (e.hash == hash &&

((ek = e.key) == key ||

(ek != null && key.equals(ek)))) {

oldVal = e.val;

if (!onlyIfAbsent)

e.val = value;

break;

}

// 如果没有找到值为key的节点，直接新建Node并加入链表即可。

Node<K,V> pred = e;

if ((e = e.next) == null) {

pred.next = new Node<K,V>(hash, key,

value, null);

break;

}

}

}

// 如果首节点为TreeBin类型，说明为红黑树结构，执行putTreeVal操作。

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;

}

}

}

}

if (binCount != 0) {

// 如果节点数>＝8，那么转换链表结构为红黑树结构。

if (binCount >= TREEIFY\_THRESHOLD)

treeifyBin(tab, i);

if (oldVal != null)

return oldVal;

break;

}

}

}

// 计数增加1，有可能触发transfer操作(扩容)。

addCount(1L, binCount);

return null;

}

## 当执行 put 方法插入数据时，根据key的hash值，在 Node 数组中找到相应的位置，实现如下： 　　1. 如果相应位置的 Node 还未初始化，则通过CAS插入相应的数据； 　　2. 如果相应位置的 Node 不为空，且当前该节点不处于移动状态，则对该节点加 synchronized 锁，如果该节点的 hash 不小于0，则 遍历 链表更新节点或插入新节点； 　　3. 如果该节点是 TreeBin 类型的节点，说明是红黑树结构，则通过 putTreeVal 方法往红黑树中插入节点； 　　4. 如果 binCount 不为0，说明 put 操作对数据产生了影响，如果当前链表的个数达到8个，则通过 treeifyBin 方法转化为红黑树，如果 oldVal 不为空，说明是一次更新操作，没有对元素个数产生影响，则直接返回旧值； 　　5. 如果插入的是一个新节点，则执行 addCount() 方法尝试更新元素个数 baseCount ；

## 3.3 get过程

1. 根据k计算出hash值，首先定位到table[]中的i。
2. 若table[i]存在，则继续查找。
3. 首先比较链表头部，如果是则返回。
4. 然后如果为红黑树，查找树。
5. 最后再循环链表查找。

## 3.4扩容过程

一旦链表中的元素个数超过了8个，那么可以执行数组扩容或者链表转为红黑树，这里依据的策略跟HashMap依据的策略是一致的。

当数组长度还未达到64个时，优先数组的扩容，否则选择链表转为红黑树。

#### 3.5 size实现

　1.8中使用一个**volatile 类型的变量** baseCount 记录元素的个数，当插入新数据或则删除数据时，会**通过 addCount() 方法更新 baseCount** ，实现如下：

在size()方法和mappingCount方法中都出现了sumCount()方法，因此，我们也顺便看一下。

/\* ---------------- Counter support -------------- \*/

/\*\*

\* A padded cell for distributing counts. Adapted from LongAdder

\* and Striped64. See their internal docs for explanation.

\*/

@sun.misc.Contended static final class CounterCell {

volatile long value;

CounterCell(long x) { value = x; }

}

// Table of counter cells. When non-null, size is a power of 2

private transient volatile CounterCell[] counterCells;

//ConcurrentHashMap中元素个数,基于CAS无锁更新,但返回的不一定是当前Map的真实元素个数。

private transient volatile long baseCount;

final long sumCount() {

CounterCell[] as = counterCells; CounterCell a;

long sum = baseCount;

if (as != null) {

for (int i = 0; i < as.length; ++i) {

if ((a = as[i]) != null)

sum += a.value;

}

}

return sum;

}

　所以在1.8中的 size 实现比1.7简单多，因为元素个数保存 *baseCount 中*，部分元素的变化个数保存在 **CounterCell 数组**中，实现如下：  
  
　　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执行 CAS 修改 baseCount 值，则失败的线程会继续执行方法体中的逻辑，使用 CounterCell 记录元素个数的变化；  
　　2. 如果 CounterCell 数组 counterCells 为空，调用 fullAddCount() 方法进行初始化，并插入对应的记录数，通过 CAS 设置**cellsBusy**字段，只有设置成功的线程才能初始化 CounterCell 数组，**（自旋锁 （锁定通过 CAS） 在调整大小和/或创建 CounterCells 时使用。 在CounterCell类更新value中会使用，功能类似显示锁和内置锁，性能更好）**  
　　3. 如果通过 CAS 设置cellsBusy字段失败的话，则继续尝试通过 CAS 修改 baseCount 字段，如果修改 baseCount 字段成功的话，就退出循环，否则继续**循环插入 CounterCell 对象；**　通过**累加 baseCount 和 CounterCell 数组中的数量，即可得到元素的总个数；**

# CAS

　　要实现无锁（lock-free）的非阻塞算法有多种实现方法，其中 CAS（比较与交换，Compare and swap） 是一种有名的无锁算法。  
　　**CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做**。