HashMap源码解读

# JDK1.7

## HashMap

### 1.1.1、概述

HashMap的主干是一个Entry数组。Entry是HashMap的基本组成单元，每一个Entry包含一个key-value键值对。

### 1.1.2、实现原理

HashMap由数组+链表组成的，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，如果定位到的数组位置不含链表（当前entry的next指向null）,那么查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，HashMap中的链表出现越少，性能才会越好。

HashMap数组每一个元素的初始值都是Null。

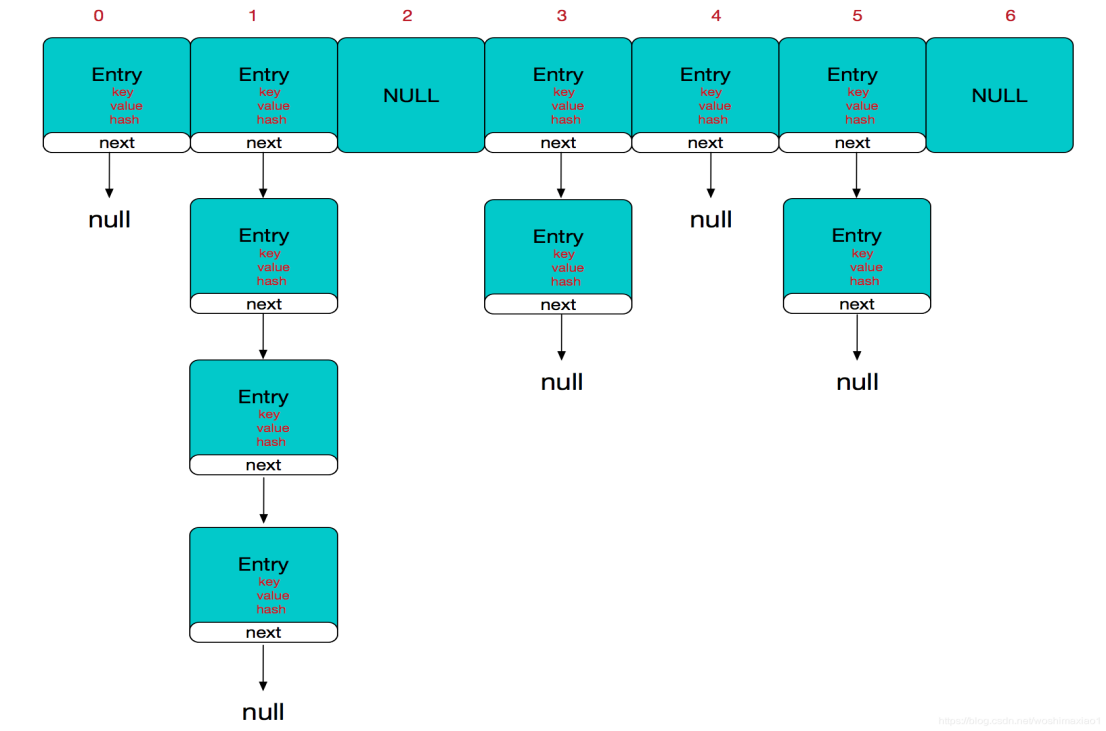


Entry是HashMap中的一个静态内部类

**static class** Entry***<***K,V***>* implements** Map.Entry***<***K,V***> {* final** K **key**;  
 V **value**;  
 Entry***<***K,V***>* next**;  
 **int hash**;

Entry***(*int** h, K k, V v, Entry***<***K,V***>*** n***) {* value** = v;  
 **next** = n;  
 **key** = k;  
 **hash** = h;  
 ***}* public final** K getKey***() {* return key**;  
 ***}* public final** V getValue***() {* return value**;  
 ***}* public final** V setValue***(***V newValue***) {*** V oldValue = **value**;  
 **value** = newValue;  
 **return** oldValue;  
 ***}* public final boolean** equals***(***Object o***) {* if *(***!***(***o **instanceof** Map.Entry***))* return false**;  
 Map.Entry e = ***(***Map.Entry***)***o;  
 Object k1 = getKey***()***;  
 Object k2 = e.getKey***()***;  
 **if *(***k1 == k2 || ***(***k1 != **null** && k1.equals***(***k2***))) {*** Object v1 = getValue***()***;  
 Object v2 = e.getValue***()***;  
 **if *(***v1 == v2 || ***(***v1 != **null** && v1.equals***(***v2***)))* return true**;  
 ***}* return false**;  
 ***}***

### 1.1.3、总体结构



### 1.1.4、HashMap初始化

**public static void** main***(***String***[]*** args***) {*** Map***<***Object, Object***>*** map = **new** HashMap***<>()***;  
 map.put***(*"1"**, **"zhansan"*)***;  
 map.put***(*"2"**, **"lisi"*)***;  
***}***

#### 1、默认值

**static final int *DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4; // 默认容量为16，容量为2的次幂数

**static final int *MAXIMUM\_CAPACITY*** = 1 << 30;**static final float *DEFAULT\_LOAD\_FACTOR*** = 0.75f;

**transient** Entry***<***K,V***>[]* table** = ***(***Entry***<***K,V***>[]) EMPTY\_TABLE***;

#### 构造初始化

**public** HashMap***() {* this*(DEFAULT\_INITIAL\_CAPACITY***, ***DEFAULT\_LOAD\_FACTOR)***;  
***}***

**public** HashMap***(*int** initialCapacity***) {* this*(***initialCapacity, ***DEFAULT\_LOAD\_FACTOR)***;  
***}***

**public** HashMap***(*int** initialCapacity, **float** loadFactor***) {* if *(***initialCapacity < 0***)* throw new** IllegalArgumentException***(*"Illegal initial capacity: "** +  
 initialCapacity***)***;  
 **if *(***initialCapacity > ***MAXIMUM\_CAPACITY)*** initialCapacity = ***MAXIMUM\_CAPACITY***;  
 **if *(***loadFactor <= 0 || Float.*isNaN****(***loadFactor***))* throw new** IllegalArgumentException***(*"Illegal load factor: "** +  
 loadFactor***)***;  
  
 **this**.**loadFactor** = loadFactor;  
 **threshold** = initialCapacity;  
 init***()***;  
***}***

#### put方法

**public** V put***(***K key, V value***) {* if *(*table** == ***EMPTY\_TABLE) {//判断数组是否为空*** inflateTable***(*threshold*)***;  
 ***}* if *(***key == **null*)* return** putForNullKey***(***value***)***;  
 **int** hash = hash***(***key***)***;  
 **int** i = *indexFor****(***hash, **table**.**length*)***;  
 **for *(***Entry***<***K,V***>*** e = **table*[***i***]***; e != **null**; e = e.**next*) {*** Object k;  
 **if *(***e.**hash** == hash && ***((***k = e.**key*)*** == key || key.equals***(***k***))) {*** V oldValue = e.**value**;  
 e.**value** = value;  
 e.recordAccess***(*this*)***;  
 **return** oldValue;  
 ***}  
 }* modCount**++;  
 addEntry***(***hash, key, value, i***)***;  
 **return null**;  
***}***

**3.1、数组为空时，构造数组**

**private void** inflateTable***(*int** toSize***) {*** *// Find a power of 2 >= toSize* **int** capacity = *roundUpToPowerOf2****(***toSize***)***;   
  
 **threshold** = ***(*int*)*** Math.*min****(***capacity \* **loadFactor**, ***MAXIMUM\_CAPACITY*** + 1***)***;  
 **table** = **new** Entry***[***capacity***]***;  
 initHashSeedAsNeeded***(***capacity***)***;  
***}***

计算数组的容量大小 capacity

计算出一个大于等于number的2的指数次幂的值

Hashmap容量为2的次幂，保证其二进制数据只有一个bit位是1其他位置是0；

减1后，保证低四位都是1111，最终计算角标时，去取决于hashcode的低四位

**private static int** roundUpToPowerOf2***(*int** number***) {*** *// assert number >= 0 : "number must be non-negative";* **return** number >= ***MAXIMUM\_CAPACITY*** ? ***MAXIMUM\_CAPACITY*** : ***(***number > 1***)*** ? Integer.*highestOneBit****((***number - 1***)*** << 1***)*** : 1;  
***}***

**public static int** highestOneBit***(*int** i***) {*** *// HD, Figure 3-1* i |= ***(***i >> 1***)***;  
 i |= ***(***i >> 2***)***;  
 i |= ***(***i >> 4***)***;  
 i |= ***(***i >> 8***)***;  
 i |= ***(***i >> 16***)***;  
 **return** i - ***(***i >>> 1***)***;  
***}***

Number=16为例进行计算

16 的二进制 0001 0000

Number-1二进制 0000 1111

***((***number - 1***)*** << 1***)***

左移1位 >> 0001 1110

右移1位 >> 0000 1111

或运算 | 0001 1111

右移2位 >> 0000 0111

或运算 | 0001 1111

右移4位 >> 0000 0001

或运算 | 0001 1111

后续结果都是 0001 1111

***(***i >>> 1***)***

无符号右移动 0000 1111

i - ***(***i >>> 1***)***

相减结果 0001 0000

Int类型每个数字占4个字节，每个字节8位，32，移动31次

**final boolean** initHashSeedAsNeeded***(*int** capacity***) {* boolean** currentAltHashing = **hashSeed** != 0;  
 **boolean** useAltHashing = sun.misc.VM.*isBooted****()*** &&  
 ***(***capacity >= Holder.***ALTERNATIVE\_HASHING\_THRESHOLD)***;  
 **boolean** switching = currentAltHashing ^ useAltHashing;  
 **if *(***switching***) {* hashSeed** = useAltHashing  
 ? sun.misc.Hashing.*randomHashSeed****(*this*)*** : 0;  
 ***}* return** switching;  
***}***

异或运算符是用符号“^”表示的，其运算规律是：  
两个操作数的位中，相同则结果为0，不同则结果为1

**3.2、key为null时，存储在数组脚标为0的位置**

**private** V putForNullKey***(***V value***) {* for *(***Entry***<***K,V***>*** e = **table*[***0***]***; e != **null**; e = e.**next*) {* if *(***e.**key** == **null*) {*** // 已经存在key为null的数据，直接替换V oldValue = e.**value**;  
 e.**value** = value;  
 e.recordAccess***(*this*)***;  
 **return** oldValue;  
 ***}  
 }* modCount**++;  
 addEntry***(***0, **null**, value, 0***)***;  
 **return null**;  
***}***

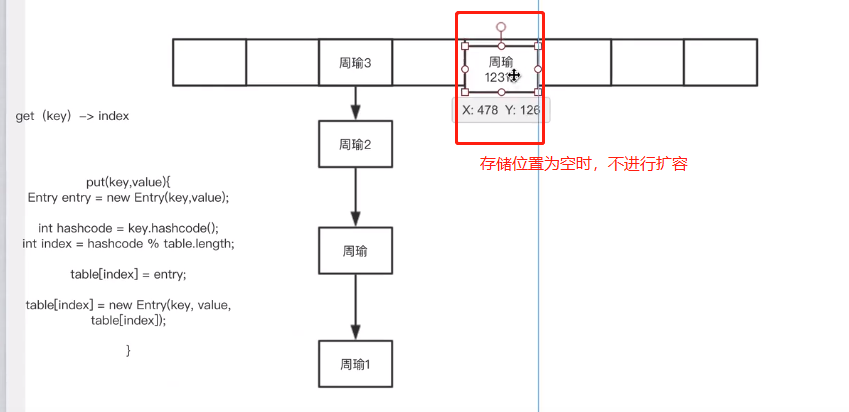
保存数据

**void** addEntry***(*int** hash, K key, V value, **int** bucketIndex***) {* if *((*size** >= **threshold*)*** && ***(*null** != **table*[***bucketIndex***])) {*** resize***(***2 \* **table**.**length*)***;  
 hash = ***(*null** != key***)*** ? hash***(***key***)*** : 0;  
 bucketIndex = *indexFor****(***hash, **table**.**length*)***;  
 ***}*** createEntry***(***hash, key, value, bucketIndex***)***;  
***}***

1. 衡量HashMap是否进行Resize的条件如：

***(*size** >= **threshold*)*** && ***(*null** != **table*[***bucketIndex***])***

如果存储的数据数已大于阈值，同时要存储的这个数据的index所对应的table位置上已有数据，则进行扩容



1. 影响发生Resize的因素有两个：

1.Capacity

HashMap的当前长度。HashMap的长度是2的幂。

2.LoadFactor

HashMap负载因子，默认值为0.75f。

1. 扩容

**void** resize***(*int** newCapacity***) {*** Entry***[]*** oldTable = **table**;  
 **int** oldCapacity = oldTable.**length**;  
 **if *(***oldCapacity == ***MAXIMUM\_CAPACITY) {* threshold** = Integer.***MAX\_VALUE***;  
 **return**;  
 ***}*** Entry***[]*** newTable = **new** Entry***[***newCapacity***]***;  
 transfer***(***newTable, initHashSeedAsNeeded***(***newCapacity***))***;  
 **table** = newTable;  
 **threshold** = ***(*int*)***Math.*min****(***newCapacity \* **loadFactor**, ***MAXIMUM\_CAPACITY*** + 1***)***;  
***}***

1. 创建一个新的Entry空数组，长度是原数组的2倍。
2. 计算并转移数据到新数组
3. 新数组替换旧数组

// 数组转移

**void** transfer***(***Entry***[]*** newTable, **boolean** rehash***) {* int** newCapacity = newTable.**length**;  
 **for *(***Entry***<***K,V***>*** e : **table*) {* while*(*null** != e***) {*** Entry***<***K,V***>*** next = e.**next**;  
 **if *(***rehash***) {*** e.**hash** = **null** == e.**key** ? 0 : hash***(***e.**key*)***;  
 ***}* int** i = *indexFor****(***e.**hash**, newCapacity***)***;  
 e.**next** = newTable***[***i***]***;  
 newTable***[***i***]*** = e;  
 e = next;  
 ***}  
 }  
}***

// 计算数组脚标

&运算的效率要远高于%

**static int** indexFor***(*int** h, **int** length***) {*** *// assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";* **return** h & ***(***length-1***)***;  
***}***

Hashmap容量为2的次幂，保证其二进制数据只有一个bit位是1其他位置是0；

减1后，保证低四位都是1111，最终计算角标时，去取决于hashcode的低四位

与运算符用符号“&”表示，其使用规律如下：  
两个操作数中位都为1，结果才为1，否则结果为0

以length=16,h=10为例进行计算

16 的二进制 0001 0000

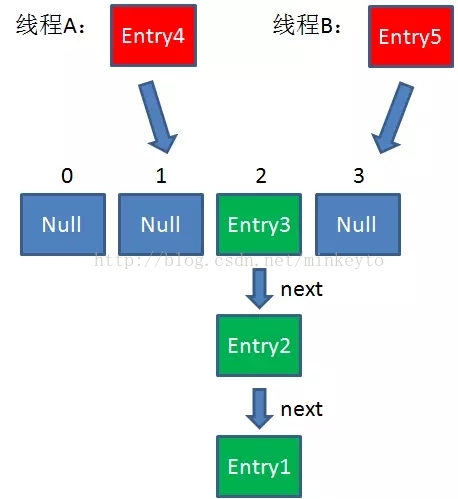
15 的二进制 0000 1111

31 的二进制 0001 1111

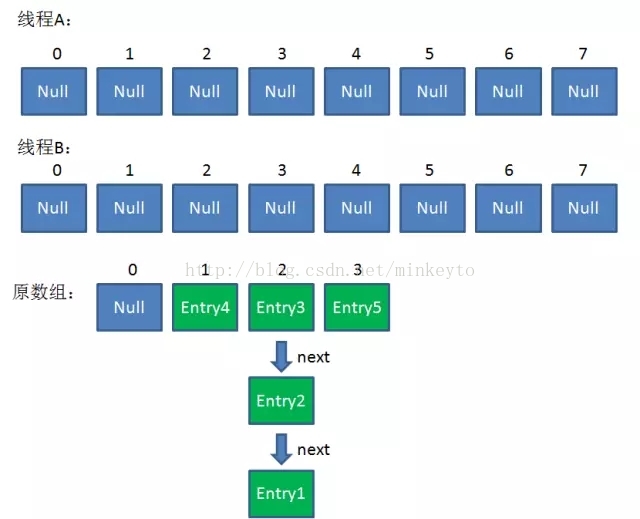
&运算 0000 1111

#### hashMap在并发条件下扩容

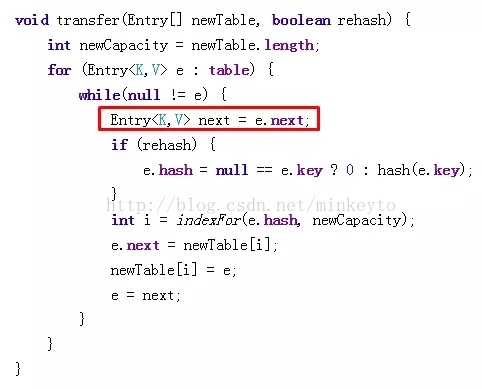
假设一个HashMap已经到了Resize的临界点。此时有两个线程A和B，在同一时刻对HashMap进行Put操作：



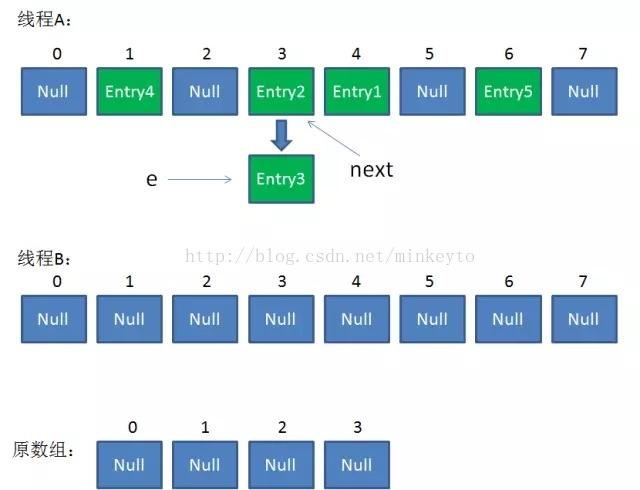
此时达到Resize条件，两个线程各自进行Rezie的第一步，也就是扩容：



这时候，两个线程都走到了reHash的步骤。让我们回顾一下ReHash的代码：



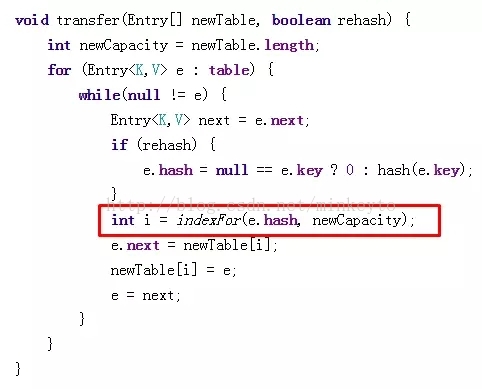
假如此时线程B遍历到Entry3对象，刚执行完红框里的这行代码，线程就被挂起。   
对于线程B来说 ： e = Entry3 next =Entry2   
这时候线程A畅通无阻地进行着Rehash，当ReHash完成后，结果如下（图中的e和next，代表线程B的两个引用）：



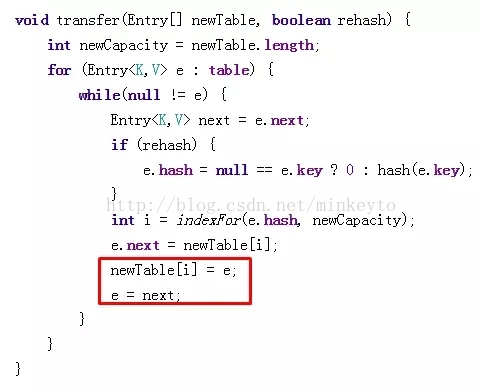
直到这一步，看起来没什么毛病。接下来线程B恢复，继续执行属于它自己的ReHash。线程B刚才的状态是：

e = Entry3

next = Entry2



当执行到上面这一行时，显然 i = 3，因为刚才线程A对于Entry3的hash结果也是3。

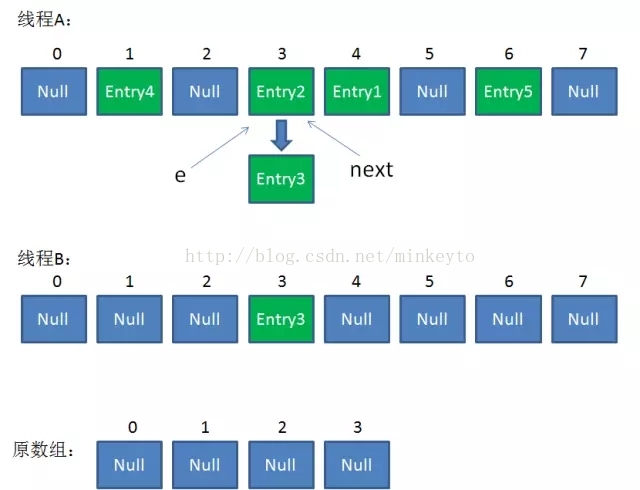


我们继续执行到这两行，Entry3放入了线程B的数组下标为3的位置，并且e指向了Entry2。此时e和next的指向如下：

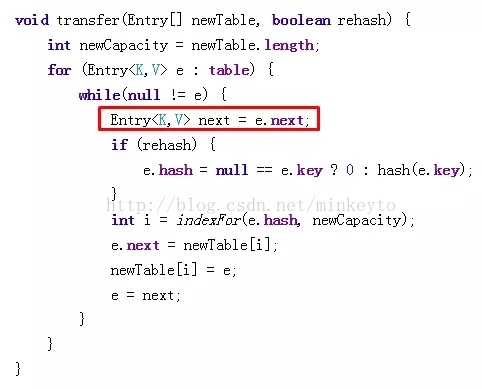
e = Entry2

next = Entry2

整体情况如图所示：



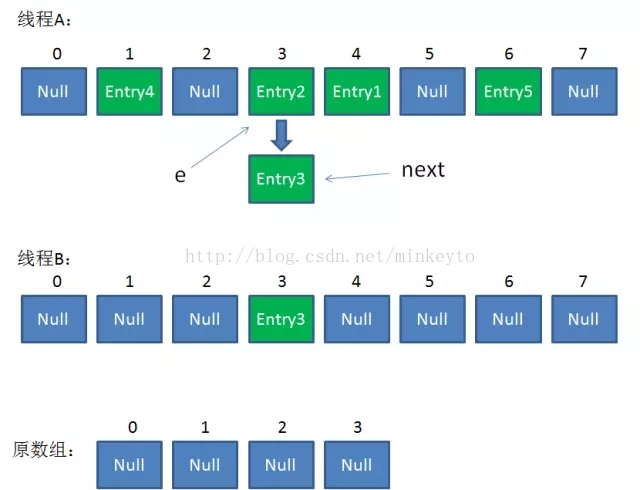
接着是新一轮循环，又执行到红框内的代码行：



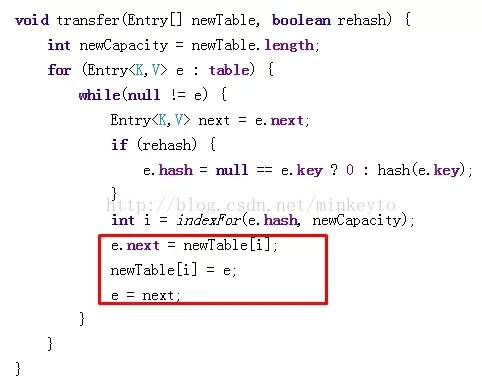
e = Entry2

next = Entry3

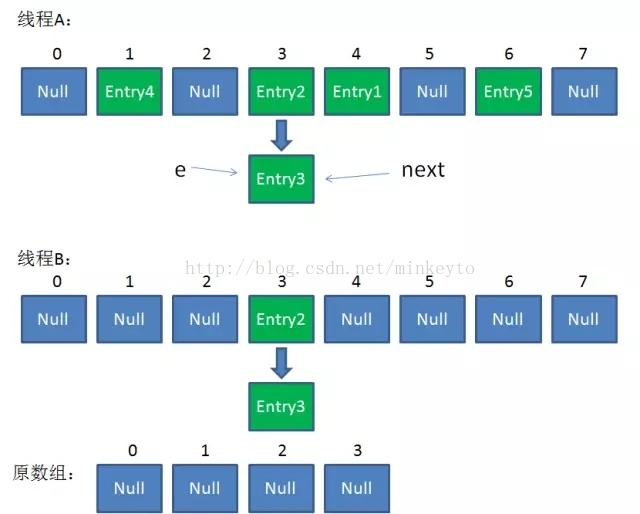
整体情况如图所示：



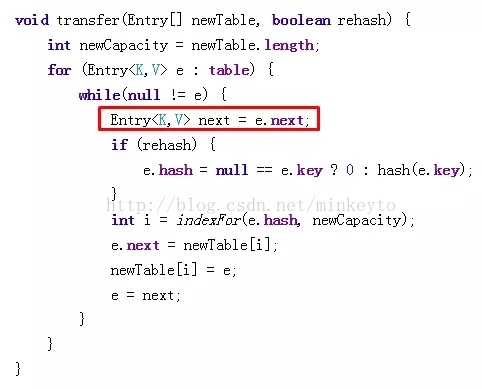
接下来执行下面的三行，用头插法把Entry2插入到了线程B的数组的头结点：



整体情况如图所示：



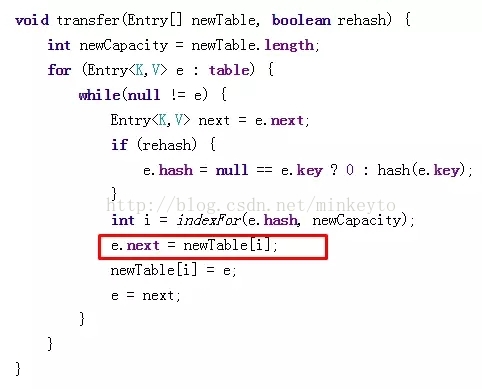
第三次循环开始，又执行到红框的代码：



e = Entry3

next = Entry3.next = null

最后一步，当我们执行下面这一行的时候，见证奇迹的时刻来临了：



newTable[i] = Entry2

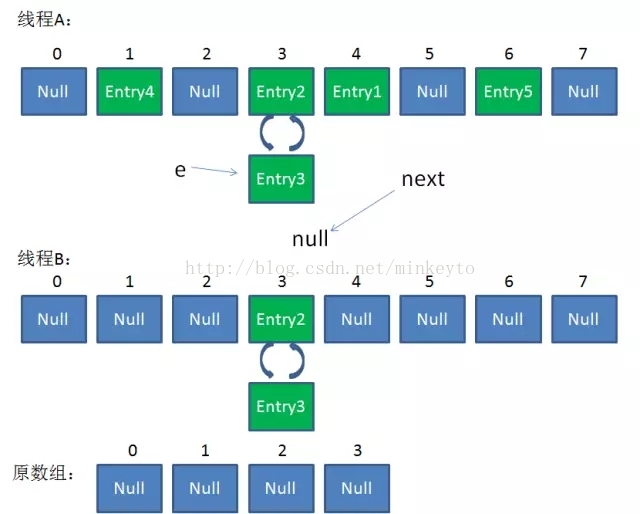
e = Entry3

Entry2.next = Entry3

Entry3.next = Entry2

链表出现了环形！

整体情况如图所示：



此时，问题还没有直接产生。当调用Get查找一个不存在的Key，而这个Key的Hash结果恰好等于3的时候，由于位置3带有环形链表，所以程序将会进入死循环

#### get方法

**public** V get***(***Object key***) {*** Segment***<***K,V***>*** s; *// manually integrate access methods to reduce overhead* HashEntry***<***K,V***>[]*** tab;  
 **int** h = hash***(***key***)***;  
 **long** u = ***(((***h >>> **segmentShift*)*** & **segmentMask*)*** << ***SSHIFT)*** + ***SBASE***;  
 **if *((***s = ***(***Segment***<***K,V***>)UNSAFE***.getObjectVolatile***(*segments**, u***))*** != **null** &&  
 ***(***tab = s.**table*)*** != **null*) {* 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;  
 **if *((***k = e.**key*)*** == key || ***(***e.**hash** == h && key.equals***(***k***)))* return** e.**value**;  
 ***}  
 }* return null**;  
***}***

### 1.1.5、ConcurrentHashMap初始化

**public static void** main***(***String***[]*** args***) {*** ConcurrentHashMap***<***Object, Object***>*** map = **new** ConcurrentHashMap***<>()***;  
 map.put***(*"1"**, **"2"*)***;  
 map.put***(*"1"**, **"3"*)***;  
 map.get***(*"1"*)***;  
***}***

#### 默认值

**static final int *DEFAULT\_INITIAL\_CAPACITY*** = 16;**static final float *DEFAULT\_LOAD\_FACTOR*** = 0.75f;**static final int *DEFAULT\_CONCURRENCY\_LEVEL*** = 16; //并发级别**static final int *MAXIMUM\_CAPACITY*** = 1 << 30;**static final int *MIN\_SEGMENT\_TABLE\_CAPACITY*** = 2; **static final int *MAX\_SEGMENTS*** = 1 << 16; *// slightly conservative***static final int *RETRIES\_BEFORE\_LOCK*** = 2;

#### 2、构造初始化

**public** ConcurrentHashMap***() {* this*(DEFAULT\_INITIAL\_CAPACITY***, ***DEFAULT\_LOAD\_FACTOR***, ***DEFAULT\_CONCURRENCY\_LEVEL)***;  
***}***

**public** ConcurrentHashMap***(*int** initialCapacity***) {* this*(***initialCapacity, ***DEFAULT\_LOAD\_FACTOR***, ***DEFAULT\_CONCURRENCY\_LEVEL)***;  
***}***

**public** ConcurrentHashMap***(***Map***<***? **extends** K, ? **extends** V***>*** m***) {* this*(***Math.*max****((*int*) (***m.size***()*** / ***DEFAULT\_LOAD\_FACTOR)*** + 1,  
 ***DEFAULT\_INITIAL\_CAPACITY)***,  
 ***DEFAULT\_LOAD\_FACTOR***, ***DEFAULT\_CONCURRENCY\_LEVEL)***;  
 putAll***(***m***)***;  
***}***

**public** ConcurrentHashMap***(*int** initialCapacity,  
 **float** loadFactor, **int** concurrencyLevel***) {* if *(***!***(***loadFactor > 0***)*** || initialCapacity < 0 || concurrencyLevel <= 0***)* throw new** IllegalArgumentException***()***;  
 **if *(***concurrencyLevel > ***MAX\_SEGMENTS)*** concurrencyLevel = ***MAX\_SEGMENTS***;  
 *// Find power-of-two sizes best matching arguments* **int** sshift = 0;  
 **int** ssize = 1;  
 **while *(***ssize < concurrencyLevel***) {*** ++sshift;  
 ssize <<= 1;  
 ***}* this**.**segmentShift** = 32 - sshift;  
 **this**.**segmentMask** = ssize - 1;  
 **if *(***initialCapacity > ***MAXIMUM\_CAPACITY)*** initialCapacity = ***MAXIMUM\_CAPACITY***;  
 **int** c = initialCapacity / ssize;  
 **if *(***c \* ssize < initialCapacity***)*** ++c;  
 **int** cap = ***MIN\_SEGMENT\_TABLE\_CAPACITY***;  
 **while *(***cap < c***)*** cap <<= 1;  
 *// create segments and segments[0]* Segment***<***K,V***>*** s0 =  
 **new** Segment***<***K,V***>(***loadFactor, ***(*int*)(***cap \* loadFactor***)***,  
 ***(***HashEntry***<***K,V***>[])*new** HashEntry***[***cap***])***;  
 Segment***<***K,V***>[]*** ss = ***(***Segment***<***K,V***>[])*new** Segment***[***ssize***]***;  
 ***UNSAFE***.putOrderedObject***(***ss, ***SBASE***, s0***)***; *// ordered write of segments[0]* **this**.**segments** = ss;  
***}***

#### 3、put方法

**public** V put***(***K key, V value***) {*** Segment***<***K,V***>*** s;  
 **if *(***value == **null*)* throw new** NullPointerException***()***;  
 **int** hash = hash***(***key***)***;  
 **int** j = ***(***hash >>> **segmentShift*)*** & **segmentMask**;  
 **if *((***s = ***(***Segment***<***K,V***>)UNSAFE***.getObject *// nonvolatile; recheck* ***(*segments**, ***(***j << ***SSHIFT)*** + ***SBASE))*** == **null*)*** *// in ensureSegment* s = ensureSegment***(***j***)***;  
 **return** s.put***(***key, hash, value, **false*)***;  
***}***

**static final class** Segment***<***K,V***>* extends** ReentrantLock **implements** Serializable ***{*** **private static final long *serialVersionUID*** = 2249069246763182397L;  
**static final int *MAX\_SCAN\_RETRIES*** =  
 Runtime.*getRuntime****()***.availableProcessors***()*** > 1 ? 64 : 1;**transient volatile** HashEntry***<***K,V***>[]* table**;  
  
 **transient int count**;  
**transient int modCount**;  
**transient int threshold**;  
**final float loadFactor**;  
  
 Segment***(*float** lf, **int** threshold, HashEntry***<***K,V***>[]*** tab***) {* this**.**loadFactor** = lf;  
 **this**.**threshold** = threshold;  
 **this**.**table** = tab;  
 ***}* final** V put***(***K key, **int** hash, V value, **boolean** onlyIfAbsent***) {*** HashEntry***<***K,V***>*** node = tryLock***()*** ? **null** :  
 scanAndLockForPut***(***key, hash, value***)***;  
 V oldValue;  
 **try *{*** HashEntry***<***K,V***>[]*** tab = **table**;  
 **int** index = ***(***tab.**length** - 1***)*** & hash;  
 HashEntry***<***K,V***>*** first = *entryAt****(***tab, index***)***;  
 **for *(***HashEntry***<***K,V***>*** e = first;;***) {* if *(***e != **null*) {*** K k;  
 **if *((***k = e.**key*)*** == key ||  
 ***(***e.**hash** == hash && key.equals***(***k***))) {*** oldValue = e.**value**;  
 **if *(***!onlyIfAbsent***) {*** e.**value** = value;  
 ++**modCount**;  
 ***}* break**;  
 ***}*** e = e.**next**;  
 ***}* 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**;  
 ***}  
 }  
 }* finally *{*** unlock***()***;  
 ***}* return** oldValue;  
 ***}***

**private int** hash***(***Object k***) {* int** h = **hashSeed**;  
  
 **if *((***0 != h***)*** && ***(***k **instanceof** String***)) {* return** sun.misc.Hashing.*stringHash32****((***String***)*** k***)***;  
 ***}*** h ^= k.hashCode***()***;  
  
 *// Spread bits to regularize both segment and index locations,  
 // using variant of single-word Wang/Jenkins hash.* h += ***(***h << 15***)*** ^ 0xffffcd7d;  
 h ^= ***(***h >>> 10***)***;  
 h += ***(***h << 3***)***;  
 h ^= ***(***h >>> 6***)***;  
 h += ***(***h << 2***)*** + ***(***h << 14***)***;  
 **return** h ^ ***(***h >>> 16***)***;  
***}***

**private transient final int hashSeed** = *randomHashSeed****(*this*)***;  
  
**private static int** randomHashSeed***(***ConcurrentHashMap instance***) {* if *(***sun.misc.VM.*isBooted****()*** && Holder.***ALTERNATIVE\_HASHING) {* return** sun.misc.Hashing.*randomHashSeed****(***instance***)***;  
 ***}* return** 0;  
***}***