ConcurrentHashMap理解分析

一、key-value都不能为空

- CHM的key和value都不允许为空。

> 注意：CHM表示ConcurrentHashMap的缩写，后文ConcurrentHashMap简称CHM。

- 为什么由源码可知

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    //这里说明了key和value是都不允许为空
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode());
    ......其他源码省略
}
```

> 但是HashMap他是允许一个key=null。

二、key的hash算法规则

- 见源码java.util.concurrent.ConcurrentHashMap#putVal

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    //key的hash算法规则
    int hash = spread(key.hashCode());
    ......其他源码省略
}
```

- 见源码java.util.concurrent.ConcurrentHashMap#spread

```
static final int HASH_BITS = 0x7fffffff = -2147483649;//最小整数
static final int spread(int h) {
    return (h ^ (h >>> 16)) & HASH_BITS;
    //key的HashCode异或上HashCode右移16位再与上最小的负整数，这里与HashMap有点不一样
    //不一样的是HashMap只用了(h ^ (h >>> 16))异或算法。
}
```

三、initTable(sizeCtl)数组的初始化

- 见源码java.util.concurrent.ConcurrentHashMap#initTable

```
//volatile为了保证可见性、有序性。
private transient volatile int sizeCtl;
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
```

```
            //sizeCtl==-1表示数组正在初始化
            //sizeCtl==0表示还没有初始化
            //sizeCtl>0表示下次扩容的大小
            //sizeCtl < 0 && sizeCtl != -1表示有多个线程在辅助扩容
            if ((sc = sizeCtl) < 0)
                Thread.yield(); //表示扩容中让出当前线程占用的cpu时间片
            else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
                //抢到初始化数组的权限将sc=SIZECTL=-1
                try {
                    if ((tab = table) == null || tab.length == 0) {
                        int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                        @SuppressWarnings("unchecked")
                        //数组初始化的长度是16。
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                        table = tab = nt;
                        //设置下一次扩容的大小为原来的0.75.
                        sc = n - (n >>> 2);
                    }
                } finally {
                    //设置下次扩容的大小
                    sizeCtl = sc;
                }
                break;
            }
        }
        return tab;
    }
```

1、自旋扩容如果sizeCtl<0表示正在初始化或者在扩容。Thread.*yield*(); 让出自己cpu执行权限。

2、抢到初始化数组资格就CAS执行**U**.compareAndSwapInt(**this**, **SIZECTL**, sc, -1)。

this：当前对象

SIZECTL：当前对象的属性内存地址

sc：当前的扩容的值

-1：更新后的值

3、数组初始化的长度是16。

四、tabAt数组槽位算法

- 见源码（java.util.concurrent.ConcurrentHashMap#putVal）

```
//通sizeCtl一样volatile保证了内存中table属性的可见性、有序性。
transient volatile Node<K,V>[] table;

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;
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
```

```
        //如何获取数组中的索引位置
        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 when adding to empty bin
        }
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        ......//省略
}
```

> 1、槽位算法 tabAt(tab, i = (n - 1) & hash)。
>
> (n - 1) & hash==hash%n 这里相当于算出来的hash取模于数组的长度length
>
> 2、如果当前槽位数为空则直接cas插入当前的node。
>
> (*casTabAt*(tab, i, **null**,**new** Node<K,V>(hash, key, value, **null**))

五、addCount统计CHM中node的个数（性能、安全）

- 见源码（java.util.concurrent.ConcurrentHashMap#addCount）

```
//在前期并发不高的时候会采用cas递增BASECOUNT，后续要是并发较大的时候就不会使用该方式。
private static final long BASECOUNT;
private final void addCount(long x, int check) {
    //x表示当前新增的数量
    //check>=0会触发是否需要检查扩容
    CounterCell[] as; long b, s;
    if ((as = counterCells) != null ||
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        //cas成功则用cas叠加basecount。
        //cas失败则用CounterCell数组来计数->类似负载均衡。
        CounterCell a; long v; int m;
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            !(uncontended =
              U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        s = sumCount();
    }
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
```

```
                    transfer(tab, nt);
            }
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
            s = sumCount();
        }
    }
}
```

- 见源码（java.util.concurrent.ConcurrentHashMap#sumCount）

```
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;
}
//这里可以看的出来CHM最终的size大小也是通过循环遍历累加起来的：baseCount+as[i].value
```

六、CounterCell【】负载计数策略

- CounterCell[]数组初始化的长度为：length=2。
- h=ThreadLocalRandom.*getProbe*()；//获取一个随机数，用于计算计数数组的槽位。ThreadLocalRandom安全产生不唯一的随机数。
- 根据随机数h计算CounterCell[]计数的槽位：index=h&(n-1)，n为计数数组的长度。
- CounterCell[]数组扩容默认为原来的2倍。
- 最后计数sum=baseCount+CounterCell[i].value。

七、transfer 扩容阶段

- 源码(java.util.concurrent.ConcurrentHashMap#helpTransfer)

```
final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
    Node<K,V>[] nextTab; int sc;
    if (tab != null && (f instanceof ForwardingNode) &&
        (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {
        //扩容戳算法
        int rs = resizeStamp(tab.length);
        while (nextTab == nextTable && table == tab &&
               (sc = sizeCtl) < 0) {
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                sc == rs + MAX_RESIZERS || transferIndex <= 0)
                break;
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
                //扩容操作，抢到扩容资格
                transfer(tab, nextTab);
                break;
```

```
                }
            }
            return nextTab;
        }
        return table;
    }
```

> 1、当数组中的节点数量超过原来的数组长度的0.75length，则会触发扩容。
>
> 2、如果当前正处于扩容阶段，则当前线程会加入并且协助扩容。
>
> 3、如果当前没有在扩容，则直接触发扩容操作。

- java.util.concurrent.ConcurrentHashMap#transfer

```
private final void** transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {

 **int** n = tab.**length**, stride;

//将 (n>>>3 相当于 n/8) 然后除以 CPU 核心数。如果得到的结果小于 16，那么就使用 16

// 这里的目的是让每个 CPU 处理的桶一样多，避免出现转移任务不均匀的现象，如果桶较少

的话，默认一个 CPU（一个线程）处理 16 个桶，也就是长度为 16 的时候，扩容的时候只会有一

个线程来扩容

 **if** ((stride = (**NCPU** > 1) ? (n >>> 3) / **NCPU** : n) <

**MIN_TRANSFER_STRIDE**)

 stride = **MIN_TRANSFER_STRIDE**; *// subdivide range*

*//nextTab* *未初始化，**nextTab* *是用来扩容的* *node* *数组*

 **if** (nextTab == **null**) { *// initiating*

 **try** {

 @SuppressWarnings(**"unchecked"**)

//新建一个 n<<1 原始 table 大小的 nextTab,也就是 32

 Node<K,V>[] nt = (Node<K,V>[])**new** Node<?,?>[n << 1];

 nextTab = nt;//赋值给 nextTab

 } **catch** (Throwable ex) { *// try to cope with OOME*

 **sizeCtl** = Integer.**MAX_VALUE**; //扩容失败，sizeCtl 使用 int 的最大值

 **return**;

 }

 **nextTable** = nextTab; //更新成员变量

 **transferIndex** = n;//更新转移下标，表示转移时的下标
```

```
 }
 **int** nextn = nextTab.**length**;//新的 tab 的长度
// 创建一个 fwd 节点，表示一个正在被迁移的 Node，并且它的 hash 值为-1(MOVED)，也
就是前面我们在讲 putval 方法的时候，会有一个判断 MOVED 的逻辑。它的作用是用来占位，表示
原数组中位置 i 处的节点完成迁移以后，就会在 i 位置设置一个 fwd 来告诉其他线程这个位置已经
处理过了，具体后续还会在讲
 ForwardingNode<K,V> fwd = **new** ForwardingNode<K,V>(nextTab);
// 首次推进为 true，如果等于 true，说明需要再次推进一个下标（i--），反之，如果是
false，那么就不能推进下标，需要将当前的下标处理完毕才能继续推进
 **boolean** advance = **true**;咕泡学院-做技术人的指路明灯，职场生涯的精神导师
//判断是否已经扩容完成，完成就 return，退出循环
 **boolean** finishing = **false**; *// to ensure sweep before committing*
*nextTab*
*通过* *for* *自循环处理每个槽位中的链表元素，默认* *advace* *为真，通过* *CAS* *设置*
*transferIndex* *属性值，并初始化* *i* *和* *bound* *值，**i* *指当前处理的槽位序号，
**bound* *指需要处理*
*的槽位边界，先处理槽位* *15* *的节点；*
 **for** (**int** i = 0, bound = 0;;) {
// 这个循环使用 CAS 不断尝试为当前线程分配任务
 // 直到分配成功或任务队列已经被全部分配完毕
 // 如果当前线程已经被分配过 bucket 区域
 // 那么会通过--i 指向下一个待处理 bucket 然后退出该循环
 Node<K,V> f; **int** fh;
 **while** (advance) {
 **int** nextIndex, nextBound;
//--i 表示下一个待处理的 bucket，如果它>=bound,表示当前线程已经分配过
bucket 区域
 **if** (--i >= bound || finishing)
 advance = **false**;
```

```
 **else if** ((nextIndex = **transferIndex**) <= 0) {//表示所有 bucket 已经
被分配完毕
 i = -1;
 advance = **false**;
 }
//通过 cas 来修改 **TRANSFERINDEX**,为当前线程分配任务，处理的节点区间为
(nextBound,nextIndex)->(0,15)
 **else if** (**U**.compareAndSwapInt
 (**this**, **TRANSFERINDEX**, nextIndex,
 nextBound = (nextIndex > stride ?
 nextIndex - stride : 0))) {
 bound = nextBound;//0
 i = nextIndex - 1;//15
 advance = **false**;
 }
 }
//i<0 说明已经遍历完旧的数组，也就是当前线程已经处理完所有负责的 bucket
 **if** (i < 0 || i >= n || i + n >= nextn) {
 **int** sc;
 **if** (finishing) {//如果完成了扩容
 **nextTable** = **null**;//删除成员变量
 **table** = nextTab;//更新 table 数组
 **sizeCtl** = (n << 1) - (n >>> 1);//更新阈值(32*0.75=24)
 **return**;
 }
// sizeCtl 在迁移前会设置为 (rs << RESIZE_STAMP_SHIFT) + 2 **(****详细介**
**绍点击这里****)**
// 然后，每增加一个线程参与迁移就会将 sizeCtl 加 1，
// 这里使用 CAS 操作对 sizeCtl 的低 16 位进行减 1，代表做完了属于自己的任
```

```
务

 **if** (**U**.compareAndSwapInt(**this**, **SIZECTL**, sc = **sizeCtl**, sc -
1)) {

第一个扩容的线程，执行 transfer 方法之前，会设置 sizeCtl =

(resizeStamp(n) << RESIZE_STAMP_SHIFT) + 2)

后续帮其扩容的线程，执行 transfer 方法之前，会设置 sizeCtl = sizeCtl+1

每一个退出 transfer 的方法的线程，退出之前，会设置 sizeCtl = sizeCtl-1

那么最后一个线程退出时：必然有

sc == (resizeStamp(n) << RESIZE_STAMP_SHIFT) + 2)，即 (sc - 2)

== resizeStamp(n) << RESIZE_STAMP_SHIFT

// 如果 sc - 2 不等于标识符左移 16 位。如果他们相等了，说明没有线程在

帮助他们扩容了。也就是说，扩容结束了。

 **if** ((sc - 2) != *resizeStamp*(n) << **RESIZE_STAMP_SHIFT**)

 **return**;

// 如果相等，扩容结束了，更新 finising 变量

 finishing = advance = **true**;

// 再次循环检查一下整张表

 i = n; *// recheck before commit*咕泡学院-做技术人的指路明灯，职场生涯的精神导师

 }

 }

// 如果位置 i 处是空的，没有任何节点，那么放入刚刚初始化的 ForwardingNode ”空节点“

 **else if** ((f = *tabAt*(tab, i)) == **null**)

 advance = *casTabAt*(tab, i, **null**, fwd);

//表示该位置已经完成了迁移，也就是如果线程 A 已经处理过这个节点，那么线程 B 处理这个节点

时，hash 值一定为 MOVED

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

 advance = **true**; *// already processed*

 }

}
```

> 高低位链表迁移算法
>
> 低位：如果扩容后h&(n-1)与扩容前的h&(n-1)相同我们把它称之为低位。保留原位不挪动。
>
> 高位：如果扩容后h&(n-1)与扩容前的h&(n-1)不同我们把它称之为高位。迁移到扩容后增加的槽位索引位置上。比如说key=6在原来槽位2位置上且当前数组长度为16，那么如果当前扩容后数组的长度为32增加了16个槽位，那么key=6将会被transfer到原来槽位2+16=18的槽位上。这样就大大提高了扩容数据迁移的效率，直接定位到迁移后的槽位，不需要重新计算hash,极大的提高了效率。
>
> 假设扩容前数组的长度为16，扩容后的数组长度为32
> 假设两个同一个槽位链上的两个key的hash值分别为9和20.
> 根据tabAt(h&(n-1))
> key1=9&(16-1)=9&15=9 相当于9%16=9
> 9 = 0000 1001
> 15 = 0000 1111
> 0000 1001 = 9
>
> key2=20&(16-1)=20&15=5 相当于20%16=4
> 20 = 0001 0100
> 15 = 0000 1111
> 0000 0100 = 4

八、resizeStamp扩容戳

- resizeStamp 用来生成一个和扩容有关的扩容戳。
- 见源码（java.util.concurrent.ConcurrentHashMap#resizeStamp）

```
private static int RESIZE_STAMP_BITS = 16;
//n表示当前数组的长度
static final int resizeStamp(int n) {
    //Integer.numberOfLeadingZeros(n)表示获取一个数高位所有的不为0的数
    //1 << (RESIZE_STAMP_BITS - 1)
    return Integer.numberOfLeadingZeros(n) | (1 << (RESIZE_STAMP_BITS - 1));
}
```

- 高低位算法

```
举例：当n=16
Integer.numberOfLeadingZeros(16)=27=          0000 0000 0000 0000 0000 0000
0001 1011
(1 << (RESIZE_STAMP_BITS - 1))=1<<15=32768 = 0000 0000 0000 0000 1000 0000
0000 0000
                                  或运算结果 = 0000 0000 0000 0000 1000 0000
0001 1011
```

1. 首先在 CHM 中是支持并发扩容的，也就是说如果当前的数组需要进行扩容操作，可以由多个线程来共同负责.

2. 可以保证每次扩容都生成唯一的生成戳，每次新的扩容，都有一个不同的 n，这个生成戳就是根据 n 来计算出来的一个数字，n 不同，这个数字也不同.

3. 高位保存扩容标记，低位保存一起扩容的线程数.

九、sizeCtl 扩容退出机制

在扩容操作 transfer 的第 2414 行，代码如下

```
if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
      if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
       return;
       finishing = advance = true;
      i = n; // recheck before commit
}
```

每存在一个线程执行完扩容操作，就通过 cas 执行 sc-1。

接着判断(sc-2) !=resizeStamp(n) << RESIZE_STAMP_SHIFT ; 如果相等，表示当前为整个扩

容操作的 最后一个线程，那么意味着整个扩容操作就结束了；如果不想等，说明还得继续

这么做的目的，一方面是防止不同扩容之间出现相同的 sizeCtl，另外一方面，还可以避免

sizeCtl 的 ABA 问题导致的扩容重叠的情况

十、红黑树

- 红黑树转变的规则

  > 判断链表的长度是否已经达到临界值 8. 如果达到了临界值，这个时候会根据当前数组的长度
  >
  > 来决定是扩容还是将链表转化为红黑树。也就是说如果当前数组的长度小于 64，就会先扩容。
  >
  > 否则，会把当前链表转化为红黑树。

- 为什么采用红黑树

  红黑树和AVL树都是**最常用的平衡二叉搜索树**，它们的查找、删除、修改都是O(lgn) time

  AVL树和红黑树有几点比较和区别：
  （1）AVL树是更加严格的平衡，因此可以提供更快的查找速度，一般读取查找密集型任务，适用AVL树。
  （2）红黑树更适合于插入修改密集型任务。
  （3）通常，AVL树的旋转比红黑树的旋转更加难以平衡和调试。

  总结：
  （1）AVL以及红黑树是高度平衡的树数据结构。它们非常相似，真正的区别在于在任何添加/删除操作时完成的旋转操作次数。
  （2）两种实现都缩放为a O(lg N)，其中N是叶子的数量，但实际上AVL树在查找密集型任务上更快：利用更好的平衡，树遍历平均更短。另一方面，插入和删除方面，AVL树速度较慢：需要更高的旋转次数才能在修改时正确地重新平衡数据结构。
  （3）在AVL树中，从根到任何叶子的最短路径和最长路径之间的差异最多为1。在红黑树中，差异可以是2倍。
  （4）两个都给O（log n）查找，但平衡AVL树可能需要O（log n）旋转，而红黑树将需要最多两次旋转使其达到平衡（尽管可能需要检查O（log n）节点以确定旋转的位置）。旋转本身是O（1）操作，因为你只是移动指针。

十一、为什么数组长度是2的幂次方

- 1、从tabAt(h&(n-1))满足h&(n-1)=h%n从而位运算代替模运算提高效率。
- 2、从h=hashCode^(hashCode>>>16),实现高低位运算概率，减少hash冲突。
- 3、2的幂次方和不是2的幂次方的二进制的有效长度要长，也是间接减少了hash碰撞的概率。
- 4、提高扩容数据迁移的效率。
- 5、能保证 索引值 肯定在 capacity 中，不会超出数组长度。