25 经典并发容器，多线程面试必备—深入解析ConcurrentHashMap下

更新时间：2019-11-21 10:45:02

> 青年是学习智慧的时期，中年是付诸实践的时期。
>
> —— 卢梭

通过上一节的学习，我们了解了 ConcurrentHashMap 的核心 hash 算法实现。本节我们将继续学习 put 相关的几个方法以及 get 方法。

上节提到对哈希表的加载是在第一次 put 操作时进行的，put 方法中相关的代码如下：

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

那么接下来我们就来看看 initTable 方法，如何创建哈希表。

1、initTable 源码分析

initTable 是初始化 table 的方法。内部考虑了多线程的并发安全。我们直接看 initTable 的代码：

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
     //如果sizeCtl<0,那么有其他线程正在创建table，所以本线程让出CPU的执行权。直到table创建完成，while循环跳出。if中同时还把sizeCtl的值赋值给了sc。
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
      //以CAS方式修改sizeCtl为-1，表示本线程已经开始创建table的工作。
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
              //再次确认是否table还是空的
                if ((tab = table) == null || tab.length == 0) {
                  //如果sc有值，那么使用sc的值作为table的size，否则使用默认值16
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                  //sc被设置为table大小的3/4
                    sc = n - (n >>> 2);
                }
            } finally {
              //sizeCtl被设置为table大小的3/4
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

里面有个关键的值 sizeCtl，这个值有多个含义。

1、-1 代表有线程正在创建 table；

2、-N 代表有 N-1 个线程正在复制 table；

3、在 table 被初始化前，代表根据构造函数传入的值计算出的应被初始化的大小；

4、在 table 被初始化后，则被设置为 table 大小 的 75%，代表 table 的容量（数组容量）。

initTable 中使用到 1 和 4，2 和 3 在其它方法中会有使用。下面我们可以先看下 ConcurrentHashMap 的构造方法，里面会使用上面的 3 。

2、ConcurrentHashMap 构造函数源码分析

ConcurrentHashMap 带容量参数的构造函数源码如下：

```
public ConcurrentHashMap(int initialCapacity) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
  //如果传入的初始化容量值超过最大容量的一半，那么sizeCtl会被设置为最大容量。
  //否则通过tableSizeFor方法就算出一个2的n次方数值作为size
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY :
               tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = cap;
}
```

这是一个有参数的构造方法。如果你对未来存储的数据量有预估，我们可以指定哈希表的大小，避免频繁的扩容操作。tableSizeFor 这个方法确保了哈希表的大小永远都是 2 的 n 次方。这里我们回想一下上一节的内容，如果 size 不是 2 的 n 次方，那么 hash 算法计算的下标发生的碰撞概率会大大增加。因此通过 tableSizeFor 方法确保了返回大于传入参数的最小 2 的 n 次方。注意这里传入的参数不是 initialCapacity，而是 initialCapacity 的 1.5 倍 + 1。这样做是为了保证在默认 75% 的负载因子下，能够足够容纳 initialCapacity 数量的元素。讲到这里你一定好奇 tableSizeFor 是如何实现向上取得最接近入参 2 的 n 次方的。下面我们来看 tableSizeFor 源代码：

```
private static final int tableSizeFor(int c) {
    int n = c - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

依旧是二进制按位操作，这样一顿操作后，得到的数值就是大于 c 的最小 2 的 n 次。我们推演下过程，假设 c 是 9：

1、int n = 9 - 1

n=8

2、n |= n >>> 1

n=1000

n >>> 1=0100

两个值按位或后

n=1100

3、n |= n >>> 2

n=1100

n >>> 2=0011

n=1111

到这里可以看出规律来了。如果 c 足够大，使得 n 很大，那么运算到 n |= n >>> 16 时，n 的 32 位都为 1。

总结一下这一段逻辑，其实就是**把 n 有数值的 bit 位全部置为 1**。这样就得到了一个肯定大于等于 n 的值。我们再看最后一行代码，最终返回的是 n+1，那么一个所有位都是 1 的二进制数字，+1 后得到的就是一个 2 的 n 次方数值。

关于 ConcurrentHashMap (int initialCapacity) 构造函数的分析我们总结下：

1、构造函数中并不会初始化哈希表；

2、构造函数中仅设置哈希表大小的变量 sizeCtl；

3、initialCapacity 并不是哈希表大小；

4、哈希表大小为 initialCapacity*1.5+1 后，向上取最小的 2 的 n 次方。如果超过最大容量一半，那么就是最大容量。

3、Put 方法中，保存 key/value 源码分析

前面我们还一直围绕在哈希表的创建在做讲解。接下来我们分析真正往哈希表存储数据的逻辑，我们先进行下回顾：

```
else {
  V oldVal = null;
  synchronized (f) {
    //再次确认该位置的值是否已经发生了变化
    if (tabAt(tab, i) == f) {
      //fh大于0，表示该位置存储的还是链表
      if (fh >= 0) {
        binCount = 1;
        //遍历链表
        for (Node<K,V> e = f;; ++binCount) {
          K ek;
          //如果存在一样hash值的node，那么根据onlyIfAbsent的值选择覆盖value或者不覆盖
          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;
          //如果找到最后一个元素，也没有找到相同hash的node，那么生成新的node存储key/value，作为尾节点放入链表。
          if ((e = e.next) == null) {
            pred.next = new Node<K,V>(hash, key,
                                      value, null);
            break;
          }
        }
      }
      //下面的逻辑处理链表已经转为红黑树时的key/value保存
      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;
        }
      }
    }
  }
```

这段代码主逻辑如下：

第一种情况：hash 值映射哈希表对应位置存储的是链表：

1、遍历 hash 值映射位置的链表；

2、如果存在同样 hash 值的 node，那么根据要求选择覆盖或者不覆盖；

3、如果不存在同样 hash 值的 node，那么创建新的 node 用来保存 key/value，并且放在链表尾部。

第二种情况：hash 值映射哈希表对应位置存储的是红黑树：

通过 TreeBin 对象的 putTreeVal 方法保存 key/value

以上逻辑还是比较清晰和简单。我们继续往下看，保存完 key/value 后，其实并没有结束 put 操作，而是进行了扩容的操作，代码如下：

```
if (binCount != 0) {
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}
```

binCount 是用来记录链表保存 node 的数量的，可以看到当其大于 TREEIFY_THRESHOLD，也就是 8 的时候进行扩容。

4、扩容源码分析

首先我们要理解为什么 Map 需要扩容，这是因为我们采用哈希表存储数据，当固定大小的哈希表存储数据越来越多时，链表长度会越来越长，这会造成 put 和 get 的性能下降。此时我们希望哈希表中多一些桶位，预防链表继续堆积的更长。接下来我们分析 treeifyBin 方法代码，这个代码中会选择是把此时保存数据所在的链表转为红黑树，还是对整个哈希表扩容。

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    if (tab != null) {
      //如果哈希表长度小于64，那么选择扩大哈希表的大小，而不是把链表转为红黑树
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            tryPresize(n << 1);
      //将哈希表中index位置的链表转为红黑树
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            synchronized (b) {
              //下面逻辑将node链表转化为TreeNode链表
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                  //TreeBin代表红黑树，将TreeBin保存在哈希表的index位置
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
```

我们再重点看一下 tryPresize，此方法中实现了对数组的扩容，传入的参数 size 是原来哈希表大小的一倍。我们假定原来哈希表大小为 16，那么传入的 size 值为 32，以此数值作为例子来分析源代码。注意 while 中第一个 if 此时不会进入，但为了讲解代码我也在注释中一并讲解了，大家看的时候在这个分支中不要以 size=16 作为前提来分析。

```
//size为32
//sizeCtl为原大小16的3/4，也就是12
private final void tryPresize(int size) {
  //根据tableSizeFor计算出满足要求的哈希表大小，对齐为2的n次方。c被赋值为64，这是扩容的上限，扩容一般都是扩容为原来的2倍，这里c值为了处理一些特殊的情况，确保扩容能够正常退出。
   int c = (size >= (MAXIMUM_CAPACITY >>> 1)) ? MAXIMUM_CAPACITY :
      tableSizeFor(size + (size >>> 1) + 1);
   int sc;
 //此时sc和sizeCtl均为12，进入while循环
   while ((sc = sizeCtl) >= 0) {
      Node<K,V>[] tab = table; int n;
    //这里处理的table还未初始化的逻辑，这是由于putAll操作不调用initTable，而是直接调用tryPresize
      if (tab == null || (n = tab.length) == 0) {
       //putAll第一次调用时，假设putAll进来的map只有一个元素，那么size传入1，计算出c为2.而sc和sizeCtl都为0，因此n=2
         n = (sc > c) ? sc : c;
         if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
               if (table == tab) {
                  @SuppressWarnings("unchecked")
                  Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                  table = nt;
                 //经过计算sc=2
                  sc = n - (n >>> 2);
               }
            } finally {
              //sizeCtl设置为2.第二次循环时，因为sc和c相等，都为2，进入下面的else if分支，结束while循环。
               sizeCtl = sc;
            }
         }
      }
    //扩容已经达到C值，结束扩容
      else if (c <= sc || n >= MAXIMUM_CAPACITY)
         break;
    //table已经存在，那么就对已有table进行扩容
      else if (tab == table) {
         int rs = resizeStamp(n);
        //sc小于0，说明别的线程正在扩容，本线程协助扩容
         if (sc < 0) {
            Node<K,V>[] nt;
           //判断是否扩容的线程达到上限，如果达到上限，退出
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
               sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
               transferIndex <= 0)
               break;
          //未达上限，参与扩容，更新sizeCtl值。transfer方法负责把当前哈希表数据移入新的哈希表。
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
               transfer(tab, nt);
         }
        //本线程为第一个扩容线程，transfer第二个参数传入null，代表需要新建扩容后的哈希表
         else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                      (rs << RESIZE_STAMP_SHIFT) + 2))
            transfer(tab, null);
      }
   }
}
```

扩容方法 transfer 中会创建新的哈希表，关键代码如下：

```
int n = tab.length, stride;
......
Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
```

n<<1 得到的数值为 2n，也就是说每次都是扩容到原来 2 倍，这样保证了哈希表的大小始终为 2 的 n 次方。

扩容的核心代码到这里就分析完了，扩容相关代码还有很多，不过主要的核心思想我们能理解就可以了。

讲到这里我们再回一下 put 方法中最后有如下一行代码：

```
addCount(1L, binCount);
```

这行代码其实是对哈希表保存的元素数量进行计数。同时根据当前保存状况，判断是否进行扩容。你可能会问，在添加元素的过程中不是已经执行了扩容的逻辑了吗？没错，不过上面的扩容逻辑是链表过长引起的。而 addCount 方法中会判断哈希表是否超过 75% 的位置已经被使用，从而触发扩容。扩容的逻辑是基本一致的。

5、get 方法源码分析

本节和前一节耗费了大量笔墨分析 put 的源代码。put 的源代码比较复杂，其实 put 方法的复杂是为了 get 服务，以提高 get 的效率。相比较 put 方法而言，get 方法就简单多了。我们直接看源代码：

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
  //获取key值的hash值
    int h = spread(key.hashCode());
  //这个if判断中做了如下几件事情：
  //1、哈希表是否存在
  //2、哈希表是否保存了数据，同时取得哈希表length
  //3、哈希表中hash值映射位置保存的对象不为null，并取出给e，e为链表头节点
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
      //如果e的hash值和传入key的hash值相等
        if ((eh = e.hash) == h) {
          //如果e的key和传入的key引用相同，或者key eaquals ek。那么返回e的value。
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
      //如果头节点的hash<0,有两种情况
      //1、hash=-1，正在扩容，该节点为ForwardingNode，通过find方法在nextTable中查找
      //2、hash=-2，该节点为TreeBin，链表已经转为了红黑树。同样通过TreeBin的find方法查找。
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
      //以上两种条件不满足，说明hash映射位置保存的还是链表头节点，但是和传入key值不同。那么遍历链表查找即可。
        while ((e = e.next) != null) {
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

6、总结

通过两小节的学习，我们把 ConcurrentHashMap 中的主要源代码学习完成了，由于篇幅有限，还有很多更细节的地方没有讲解。如果想继续研究的话，建议把 Node、TreeNode 相关结构看一下。对算法感兴趣的话，可以看一下红黑树转化的过程。

ConcurrentHashMap 中，通过大量的 CAS 操作加上 Synchronized 来确保线程安全。对 ConcurrentHashMap 的学习我们把重点放在哈希算法和扩容上，面试的时候是考察的重点。

}

24 经典并发容器，多线程面试必备—深入解析 ConcurrentHashMap上

26不让我进门，我就在门口一直等！—BlockingQueue和 ArrayBlockingQueue