ConcurrentHashMap

> 本文基于 jdk11，前置知识：[Java HashMap 详解](./HashMap 学习总结.pdf)

插入时逻辑

在 jdk1.7 以前，ConcurrentHashMap 采用分段锁，将一个大的 HashMap 数组分为多个小的段 Segment，每个段也是一个 HashMap 数组，插入时首先计算 key 所属的段，然后对整个段上锁，再执行插入。通过细化敏感资源的思想，ConcurrentHashMap 大大提高了效率，而在 jdk1.8 之后，这种思想更是明显。

在 jdk1.8 后，ConcurrentHashMap 放弃了分段的做法，**通过对数组下标上锁，更细一步的缩小了锁的范围**。

这是如何做到的呢？在 ConcurrentHashMap 中，一个元素要插入（或者删除）到下标 i 的位置，它首先必须要获得 tab[i] 上持有的锁，tab[i] 其实就是一个 Node，如果这是一个链表，tab[i] 的含义是链表的头节点，由于插入采用的是尾插法，因此头节点的位置是不会改变的；如果这是一颗树，由于插入时树的根节点是可能会被调整的，因此 ConcurrentHashMap 用 TreeBin 类（继承 Node）封装了树的根节点，树的根节点可能会变，但 TreeBin 是不会变的，换句话说，就是给整棵树套了层皮。

jdk1.8 后，锁是利用 synchronized 实现的，经过几次优化后，synchronized 的性能已经相当不错了。

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    int hash = spread(key.hashCode());
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh; K fk; V fv;
        // 如果头节点为空，那么利用 CAS 尝试设置头节点
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value)))
                break;                   // no lock when adding to empty
bin
        }
        else {
            V oldVal = null;  // P
```

```
            // 否则对头节点上锁
            synchronized (f) {
                // 上锁之后在此检查，因为在上锁的那一刻是存在并发冲突的，例如在 P
点
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) { // 哈希值大于 0，意味着是链表
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            // 同 hashmap 一样的逻辑
                        }
                    }
                    else if (f instanceof TreeBin) { // 树的操作
                        Node<K,V> p;
                        binCount = 2;
                        (p = ((TreeBin<K,V>)f).putTreeVal(hash,
key,value));
                    }
                }
            }
            if (binCount >= TREEIFY_THRESHOLD)
                treeifyBin(tab, i);  // 同 HashMap 一样的逻辑，树化
            break;
        }
    }
    addCount(1L, binCount);
    return null;
}
```

总结来看，jdk1.8 中 ConcurrentHashMap 是通过 synchronized 对链表头节点或 TreeBin（树的根节点的包装类）上锁，一旦一个线程持有锁后，由于链表头节点是不会改变的，而树根节点通过包装后也是不变的，所以其他线程的插入、删除必须要等待获取链表头节点上的锁，这保证了线程并发安全性。

jdk1.8 后虽然删除了分段锁，但仍然是一个将锁细化的思想，这次将锁细化到了数组中的每一个下标，细化程度应该是更甚于 jdk1.7 的，因此效率也是高于 jdk.17 的。我们从中也可以学习到一点思想，例如下单时我们不必锁住整个方法，可以只对商品 ID 上锁。

大小增加逻辑

ConcurrentHashMap 是如何安全的增加或减少大小的？我们直到单纯的 size++ 和 size– 是不安全的，因为 Java 是基于栈解释而不是基于寄存器的，size++ 首先会把 size 和 1 入栈，然后再取出这两个变量进行相加，这至少需要 3 条命令，并不是一条原语，因此这不是并发安全的。

按理来说，应该可以采用 Atomic 类进行自增和自减，这的确可以，但是 Atomic 底层也是利用 CAS 不断自旋，ConcurrentHashMap 采用了效率更高的一种方式，ConcurrentHashMap 仍然采用细分的思想，创建了 CounterCell 数组，**如果线程对 BASECOUNT 自增或自减失败，线程就会转而操作 CounterCell 数组，将**

CounterCell 中的值增减或减小，如果这还失败，那么会尝试对 CounterCell 扩容，直到到达一个阈值，然后继续循环。

至于选择 CounterCell 数组中的哪一个 CounterCell，这仍然是通过哈希计算出下标，CounterCell 数组的大小也是 2 的幂，如果冲突频繁，也会尝试扩容，所允许的最大大小为 CPU 核心数，这是有原因的，当线程数超过 CPU 核心数后，并行效率其实还不如串行。

说白了，其实就是减少冲突，既然另一个线程在操作这个变量，那么我就去操作其他变量，计算 size 的时候再重新加起来就好了，下面是计算大小的方法：

```
final long sumCount() {
    CounterCell[] cs = counterCells;
    long sum = baseCount;
    if (cs != null) {
        for (CounterCell c : cs)
            if (c != null)
                sum += c.value;
    }
    return sum;
}
```

这种方法还是非常巧妙的，**将需要竞争的资源分散开来，以减小冲突，需要时再重新聚合起来**，果然看源码还是能学到很多牛逼的思想。

```
for (;;) {
    CounterCell[] cs; CounterCell c; int n; long v;
    if ((cs = counterCells) != null && (n = cs.length) > 0) {
        c = cs[(n - 1) & h];
        // 尝试修改值
        if (U.compareAndSetLong(c, CELLVALUE, v = c.value, v + x))
            break;
        // 如果 cs 数组的大小大于等于 CPU 核心数了，停止扩容
        else if (counterCells != cs || n >= NCPU)
            //
        else if (cellsBusy == 0 && U.compareAndSetInt(this, CELLSBUSY, 0,
1)) {
            try {
                // 扩容
                if (counterCells == cs) // Expand table unless stale
                    counterCells = Arrays.copyOf(cs, n << 1);
            } finally {
                cellsBusy = 0;
            }
            collide = false;
            continue;                   // Retry with expanded table
        }
        h = ThreadLocalRandom.advanceProbe(h);
    }
    // 尝试设置 BASECOUNT
```

```
    else if (U.compareAndSetLong(this, BASECOUNT, v = baseCount, v + x))
        break;                          // Fall back on using base
}
```

初始化逻辑

在 jdk1.7 之前，如果过多线程同时开始初始化，可能会导致 CPU飙升，而在 jdk.18 后，缓解了这个问题：

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSetInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2); // sc = n - 1/4 n = 0.75n
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

可以发现，如果 sc 标志 = -1 时，标识正在初始化，jdk1.7 之前会无线自旋，而在当前版本，添加了 Thread.yield() 方法，表示线程愿意降低自己执行的优先度，通过这种方法来缓解 CPU 飙升的问题。

还可以发现，一旦初始化完成后，sc 会被设置为 0.75N，这是扩容时的阈值，**扩容因子默认为 0.75，并且不允许用户修改。**

扩容时逻辑

ConcurrentHashMap 允许多线程同时进行扩容，在插入时，你会看到如下代码：

```
else if ((fh = f.hash) == MOVED)
    tab = helpTransfer(tab, f);
```

一旦 tab[i].hash 为 -1 时，标识此下标正在发生扩容，那么线程将会尝试帮助扩容：

```
final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
    Node<K,V>[] nextTab; int sc;
    nextTab = ((ForwardingNode<K,V>)f).nextTable; // 获取新表
    if (tab != null && (f instanceof ForwardingNode)) {
        int rs = resizeStamp(tab.length); // 获取扩容的唯一标识符
        while (nextTab == nextTable && table == tab && (sc = sizeCtl) <
0) {
            // 如果标识符不相等，标识扩容已完成或已经开始下一轮扩容了
            // sc == rs + MAX_RESIZERS 是个特殊的运算，用于控制帮忙线程最大数
量
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs +
MAX_RESIZERS)
                    break;
            // 尝试将 sc + 1 表示线程增加，成功则开始扩容
            if (U.compareAndSetInt(this, SIZECTL, sc, sc + 1)) {
                transfer(tab, nextTab);
                break;
            }
        }
        return nextTab;
    }
    return table;
}
```

上面代码中的 rs 的值是由当前数组大小唯一标识的，例如标识是从 16 扩容到 32，还是由 32 扩容到 64，并且只有 16 位有效，而 sc 的值是由最先开始扩容的线程设置的，sc 的高 16 位被设置成 rs，不仅如此，**rs << RESIZE_STAMP_SHIFT 一定是负的，这意味着扩容时 sc 一定是负的**：

```
if (U.compareAndSetInt(this, SIZECTL, sc, (rs << RESIZE_STAMP_SHIFT) +
2))
    transfer(tab, null); //RESIZE_STAMP_SHIFT =16
```

这意味着，如果 (sc >>> RESIZE_STAMP_SHIFT) != rs，说明扩容已经完成或已经进入下一轮扩容，信息已过时，应该退出重新循环，这里都是为了保障并发安全。

可以看到每个线程进入帮忙扩容时，都会自增 sc 的值，而 sc 的值最初为 rs << 16 + 2，这意味着 sc 的低 16 位标识当前正在参与扩容的线程数 +1（初始时只有一个线程但 +2 了，多了一个），ConcurrentHashMap 通过这种方式可以控制参与帮忙调整线程的最大数量。

转移的逻辑与 HashMap 是类似的，但由于允许多线程共同扩容，ConcurrentHashMap 必须要是并发安全的。

在 HashMap 中我们有一个很巧妙的结论，下标 i 的元素扩容后要么在 i 处，要么在 i + n 处，这意味两个不同下标之间的元素是不可能有任何冲突的，因此，只要保证不同线程选取不同下标进行扩容就可以了。在 ConcurrentHashMap 中，这通过 volatile 变量 transferIndex 来实现，transferIndex 标识线程开始选取的下标，线程会尝试选取

stride（步长）个下标，那么线程必须要利用 CAS 修改 transferIndex 来标识下一个线程开始的位置，如果 CAS 失败了，线程需要重新尝试，**因为 volatile 只保证了可见性而并没有保证原子性。**

在对下标 i 进行扩容时，线程必须要锁住 tab[i]，锁住的逻辑与插入时类似，这是由于在扩容时，可能还会与插入、查找、删除等操作存在并发冲突，因此必须上锁。但换个角度，如果 tab[i] 还没有线程对其扩容（但是扩容操作已经开始，只是还没轮到 i），此时其他的操作是不会被堵塞的。

sizeCtl

sizeCtl，即 sc 作为大小控制量，在 ConcurrentHashMap 中至关重要，通过上面的分析，来总结一下这个变量不同值对应的不同含义。

- sizeCtl = 0：这标识 tab 还未初始化，需要初始化。
- sizeCtl = -1：这标识 tab 正在初始化，等待的线程需降低自己的执行的优先级，防止 CPU 飙升。
- sizeCtl < 0 && sizeCtl != -1：标识正在扩容，sc 高16位标识扩容标识戳，数组大小不同，则标识戳也不同；低16位表示正在扩容的线程数 + 1。
- sizeCtl > 0：这代表扩容的阈值，ConcurrentHashMap 中，扩容因子固定为 0.75。