

# 非阻塞队列ConcurrentLinkedQueue实现分析

2015-11-08 22:34 feiying 0 阅读 108

## 1.非阻塞队列数据结构

BlockingQueue系列，功能为"阻塞"，也就是说，当队列的内容满了之后，调用队列的线程自动进行线程阻塞状态，

这种阻塞背后的实现是使用ReentrantLock来做的；

而对于队列来讲，还有一种需求就是，是否可以通过完全的纯CAS级别的锁，来让队列进行poll，offer都无锁的操作；

这就是无阻塞队列，从单端队列和双端队列来分，一共分为两种不同的类型：

- ConcurrentLinkedDeque.class, 非阻塞双端队列，链表实现
- ConcurrentLinkedQueue.class, 非阻塞队列，链表实现

ConcurrentLinkedDeque是双端入口的队列，它主要在队头和队尾都能进行add/remove等各种操作，而没有任何的限制，

对于整体的类的实现上，除了Node节点保持向前的一个节点，队头需要考虑有add,队尾考了有remove，只是判断上更为复杂，条件分支更多而已，

整个算法几乎和ConcurrentLinkedQueue差不多，因此，这里就分析ConcurrentLinkedQueue即可。

对于ConcurrentLinkedQueue来讲，顾名思义，是由链表来作为存储结构的：

```

private static class Node<E> {
    volatile E item; 节点内容
    volatile Node<E> next; 指向下一个节点的指针

    /**
     * Constructs a new node. Uses relaxed write because item can
     * only be seen after publication via casNext.
     */
    Node(E item) {
        UNSAFE.putObject(this, itemOffset, item);
    }

    boolean casItem(E cmp, E val) {
        return UNSAFE.compareAndSwapObject(this, itemOffset, cmp, val);
    }

    void lazySetNext(Node<E> val) {
        UNSAFE.putOrderedObject(this, nextOffset, val); 懒替换
    }

    boolean casNext(Node<E> cmp, Node<E> val) {
        return UNSAFE.compareAndSwapObject(this, nextOffset, cmp, val);
    }

    // Unsafe mechanics

    private static final sun.misc.Unsafe UNSAFE;
    private static final long itemOffset;
    private static final long nextOffset;

    static {
        try {
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            Class k = Node.class;
            itemOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField("item"));
            nextOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField("next"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
}

```

**CAS元素替换**

**CAS将指针替换**

在CAS锁定的时候，直接通过内存定位属性，所以需要位移

应用服务器技术讨论圈

需要留意的是，Node节点中的CAS方法，几乎每一个方法都是调用Unsafe类中的一些方法来进行CAS加锁操作

操作一共分为2类，一类是CAS进行元素替换，一类是将next指针进行调整；

而一般出队和入队都需要head和tail，ConcurrentLinkedQueue将这两个作为指针进行缓存起来，

```

private transient volatile Node<E> head;

private transient volatile Node<E> tail;

```

而正因为前面Node节点的操作一共分为两类，而出队和入队同时需要用到这两类的内容，

ConcurrentLinkedQueue又不允许方法级别加锁，所以在多线程的情况下，经常会出现tail指针和head指针暂时出现错位的情况，

如下图所示：



这种情况再入队，出队的情况下，都需要进行判断，配合上面的Node节点的CAS方法，因此各种方法的条件分支非常的多，不容易理解；

下面就对出队和入队等相关的关键方法，结合上面的数据结构进行分析；

## 2.add/offer

因为为非阻塞队列，所以add方法和offer方法没有任何的区别，不像阻塞队列那种各种之间二者之间一个是阻塞，一个是阻塞抛异常；

以add方法为例，

```
*/  
public boolean add(E e) {  
    return offer(e);  
}
```

直接调用的就是offer方法，可见重点在offer方法中：

```
*/  
public boolean offer(E e) {  
    checkNotNull(e);  
    final Node<E> newNode = new Node<E>(e);  
  
    for (Node<E> t = tail, p = t;;) {  
        Node<E> q = p.next;  
        if (q == null) {  
            // p is last node  
            if (p.casNext(null, newNode)) {  
                // Successful CAS is the linearization point  
                // for e to become an element of this queue,  
                // and for newNode to become "live".  
                if (p != t) // hop two nodes at a time  
                    casTail(t, newNode); // Failure is OK.  
                return true;  
            }  
            // Lost CAS race to another thread; re-read next  
        }  
        else if (p == q) {  
            // We have fallen off list. If tail is unchanged, it  
            // will also be off-list, in which case we need to  
            // jump to head, from which all live nodes are always  
            // reachable. Else the new tail is a better bet.  
            p = (t != (t = tail)) ? t : head;  
        }  
        else  
            // Check for tail updates after two hops.  
            p = (p != t && t != (t = tail)) ? t : q;  
    }  
}
```

轮询进行插入队列操作，直到插入成功为止；

q为队列尾部，说明现在是插入的好时机，执行casNext进行插入节点； ①

p==q，而q应该是p的next，这说明中间有删除节点了 ②

不是上述两种情况，需要向尾部再通过循环推进一下，将p=q ③

整体思路比较复杂，

首先是判断当前Node节点是否为null，然后创建出来一个newNode节点，之后，开始循环，一直到插入成功为止；

每一次循环中，p从tail指针处开始循环，然后取q为p的next节点：

a.如果是单线程运作的话，因为p是tail，那么q就是null，所以立刻进行第一个分支，直接开始调用p.casNext，将tail的指针

指向newNode，如果成功的话，这里有一个判断：



```

if (p != t) // hop two nodes at a time
    casTail(t, newNode); // Failure is OK.
return true;

private boolean casTail(Node<E> cmp, Node<E> val) {
    return UNSAFE.compareAndSwapObject(t, s, tailOffset, cmp, val);
}

```

p这个时候，应该就是tail了，但是不要忘记，这段代码是在多线程的环境下进行，可能有n个线程正在offer，

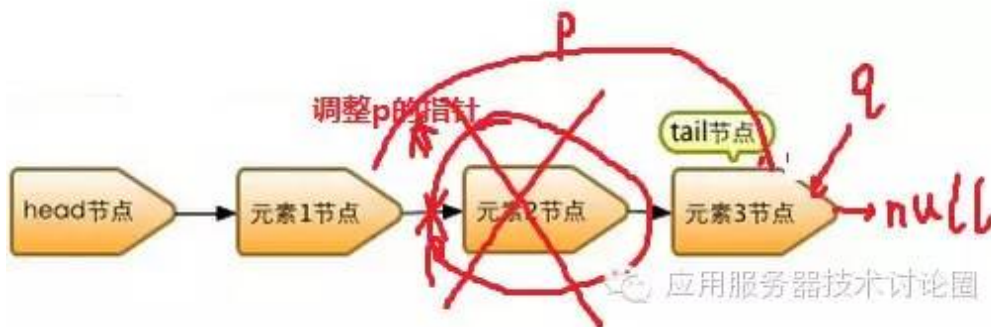
那么虽然你将p的指针指向newNode，但是这个时候其他线程已经抢先将node节点挂到tail上，也就是现在p节点已经不是尾巴节点了，你需要调用casTail方法，重新CAS，将newNode直接挂到末尾；

b.第二个循环的意思是，p==q,也就是进入这个循环，

首先是，q不是null,说明在执行q=p.next这一句话的时候，已经有其它的线程抢先将Node节点挂到末尾了

但是这里又很奇怪，p竟然等于q，说明这种情况是，可能同时队尾的节点被remove掉了，

类似于下面这种情况：



本来元素1节点的p指针是指向元素2的，但是元素2被删除了，这样p只能和q指向同一个队列了，

当这种情况，又进行了一次的判断：

```

// We have fallen off list. If tail is unchanged, it
// will also be off-list, in which case we need to
// jump to head, from which all live nodes are always
// reachable. Else the new tail is unreachable.
p = (t != (t = tail)) ? t : head;

```

先分解一下 (t != (t = tail)) 这段话：

```

Node<E> tmp=t;
t = tail;
if (tmp != t) {
    p=t;
} else {
    p= head;
}

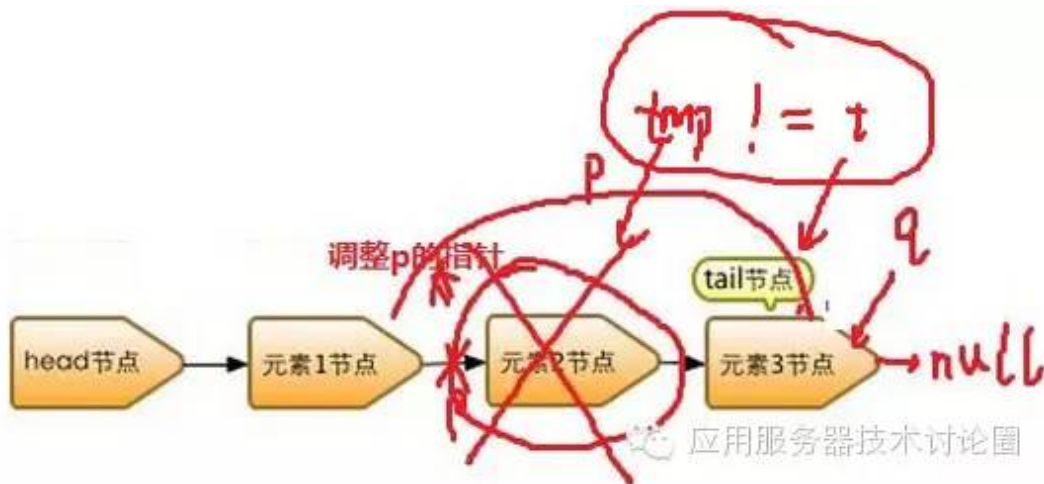
```

下载《开发者大全》

下载 (/download/dev.apk)

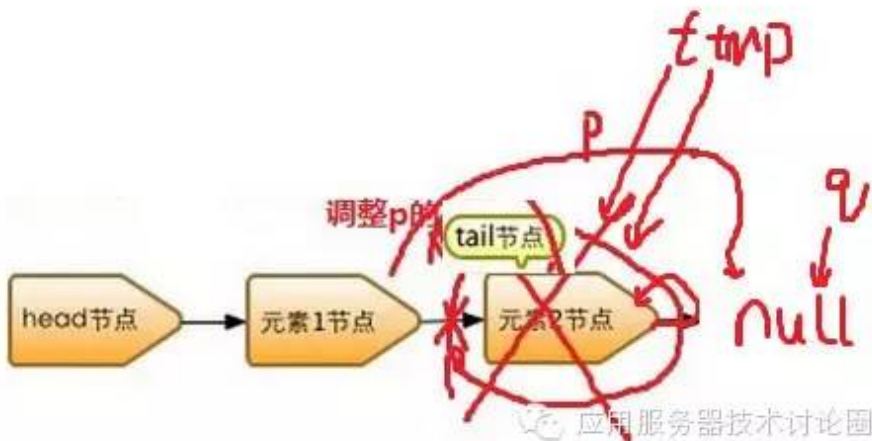
这段代码的上述两句话，tmp引用先指向t，然后tail引用由指向t，

如果tmp不等于t，如下图所示：



说明tmp指向的节点已经脱离了链表，你这个p节点需要赋值为t，再进行下一次循环；

如果tmp==t了，那就说明下图：



当前的末尾节点已经脱离了当前的链表，因此这个p节点指向的是null，

p指针无法继续往后移动了，到头了，

因此这里直接将p节点指向head===》也就是从头来吧，再从头开始遍历吧；

c.第三个循环进入，它说明肯定有其他线程再操作入队，而没有进入第二个循环，说明尾部处的节点也没有删除，

那么就进入下述的代码：

```
// Check for tail updates after two hops.
p = (p != t && t != (t = tail)) ? t : q;
```

可以翻译为：

```

if (p==t) {
    p = q;
} else {
    Node<E> tmp=t;
    t = tail;
    if (tmp==t) {
        p=q;
    } else {
        p=t;
    }
}

```

应用服务器技术讨论圈

如果 $p==t$ ，或者tmp还是tail的话，那么p向前推进一下，这种情况就是其他线程修改了，导致目前操纵的p指针需要后移才行，

否则，现在的p指针不是tail处，因此需要进入下一个循环继续到队尾进行入队操作；

如果发现tmp不是tail的话，p就是尾节点了，不用再推进了，直接进入下一个循环，再去尝试CAS操作；

总结一下，offer的操作非常复杂，需要考虑各种情况的条件判断，代码极难完全读懂，维护起来比较费劲，这也是无锁流程最差的一点了；

### 3.poll/peek/remove

peek是撇看一下，实质上并没有元素的删除和增加，但是从程序的角度来讲，也比较麻烦：

```

public E peek() {
    restartFromHead:
    for (;;) {
        for (Node<E> h = head, p = h, q;;) {
            E item = p.item;
            if (item != null || (q = p.next) == null) {
                updateHead(h, p);
                return item;
            }
            else if (p == q)
                continue restartFromHead;
            else
                p = q;
        }
    }
}

```

应用服务器技术讨论圈

首先也是一个循环，然后也是分成三个情况：

a.item不为null的时候，并没有立刻return返回，而是尝试先updateHead，

这个updateHead传入的参数是head和p，如果是单线程的话，完全不必多次一举，因为head就是p，

但是多线程的话，很可能在执行上面的判断后，head和p就不是一个指向了，因此在返回item之前，一定要再去进行CAS的赋值：

下载《开发者大全》

下载 (/download/dev.apk)



```
/**
 * Try to CAS head to p. If successful, repoint old head to itself
 * as sentinel for succ(), below.
 */
final void updateHead(Node<E> h, Node<E> p) {
    if (h != p && casHead(h, p))
        h.lazySetNext(h);
}
```

 应用服务器技术讨论圈

从casHead的源码上看，也是当表头发生变化的时候，对p节点重新赋值，这样p.item就是最新的了；

b.p==q这种情况，和offer的第二种情况一样，也是head节点被删除了，这时候，需要从头开始进行遍历；

c.如果不是上述的两种情况，还得从头开始遍历，那么索性就把p赋值为q，这样下一个迭代就可以进入上一个判断，迭代遍历了；

poll方法稍微比peek方法要复杂一点：

在peek的基础上，增加了一个分支判断，

也就是首先判断如果item不为null的话，直接进行casItem弹出item元素，

然后当p和head不相等的时候，调用updateHeadCAS更新head指针，

这个时候p这个节点已经脱离链表了，队列头已经变成了q，

直接返回item即可；

如果q为null，那么第二个分支判断说明队列为空，直接返回null

其余两个判断分支和peek一致；

对于remove来说，原理同样是CAS的操作：



```

    */
    Node<E> first() {
        restartFromHead:
        for (;;) {
            for (Node<E> h = head, p = h, q;;) {
                boolean hasItem = (p.item != null);
                if (hasItem || (q = p.next) == null) {
                    updateHead(h, p);
                    return hasItem ? p : null;
                }
                else if (p == q)
                    continue restartFromHead;
                else
                    p = q;
            }
        }
    }

    */
    public boolean remove(Object o) {
        if (o == null) return false;
        Node<E> pred = null;
        for (Node<E> p = first(); p != null; p = succ(p)) {
            E item = p.item;
            if (item != null &&
                o.equals(item) &&
                p.casItem(item, null)) {
                Node<E> next = succ(p);
                if (pred != null && next != null)
                    pred.casNext(p, next);
                return true;
            }
            pred = p;
        }
        return false;
    }

    */
    final Node<E> succ(Node<E> p) {
        Node<E> next = p.next;
        return (p == next) ? head : next;
    }

```

判断，直接一步到位，  
将前后指针链接，对象就删除了

应用服务器技术讨论圈

做一个循环，frist()是找head节点，succ(p)是每一步都找当前节点的下一个节点，一直找到o这个节点，

因为o不一定是队头，所以竞争压力小很多，删除节点一步到位，链接前后指针即可；

其它方法大同小异，例如size方法，就是对链表进行遍历：

```

    */
    public int size() {
        int count = 0;
        for (Node<E> p = first(); p != null; p = succ(p))
            if (p.item != null)
                // Collection.size() spec says to max out
                if (++count == Integer.MAX_VALUE)
                    break;
        return count;
    }

```

应用服务器技术讨论圈

可以分析得出，同样也是调用succ方法找下一个节点，所以，链表的遍历会比单线程来得更为慢一些，

在应用的时候，需要注意，这种size的方式，还不如维持一个外部的AtomicInteger来得好一些；

## 总结：

**ConcurrentLinkedQueue没有长度，是链表存储结构，完全采用CAS锁编写的Queue，代码逻辑比较复杂，分支判断比较多，但在多线程大并发上比LinkedBlockingQueue优势明显！**


下载《开发者大全》

下载 (/download/dev.apk)

×



分享 :

阅读 108  0

应用服务器技术讨论圈 更多文章

东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)

玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)

金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)

GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)

Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢

Android开发技术周报 Issue#4 (/html/430/201510/400351761/1.html)

Facebook如何采集其Android应用性能数据 (/html/209/201602/401955288/1.html)

Twitter是如何构建高性能分布式日志的！ (/html/46/201509/209237310/1.html)

iOS移动开发周报-第10期 (/html/224/201405/200119673/1.html)

与Android Memory谈一场不分手的恋爱（一） (/html/137/201604/402846534/1.html)