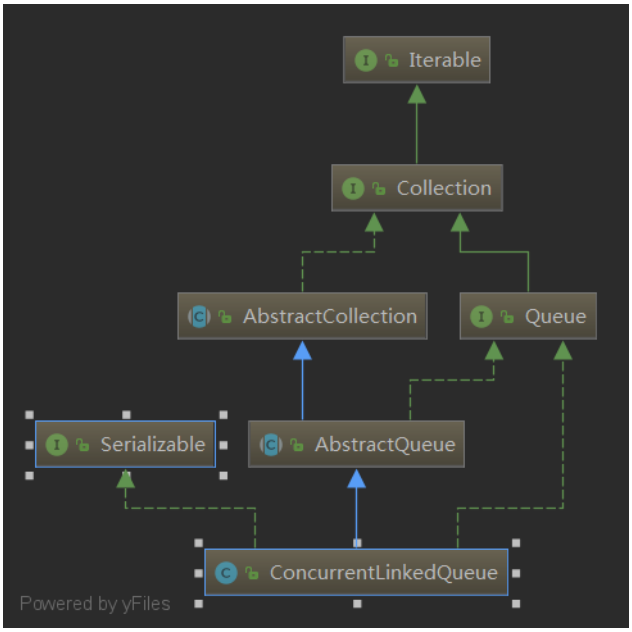
问题

（1）ConcurrentLinkedQueue是阻塞队列吗?

（2）ConcurrentLinkedQueue如何保证并发安全?

（3）ConcurrentLinkedQueue能用于线程池吗?

简介



ConcurrentLinkedQueue只实现了Queue接口，并没有实现BlockingQueue接口，所以它不是阻塞队列，也不能用于线程池中，但是它是线程安全的，可用于多线程环境中。

那么，它的线程安全又是如何实现的呢？让我们一起来瞧一瞧。

源码分析

主要属性

```
    // 链表头节点
    private transient volatile Node<E> head;
    // 链表尾节点
    private transient volatile Node<E> tail;

```

就这两个主要属性，一个头节点，一个尾节点。

主要内部类

```
    private static class Node<E> {
        volatile E item;
        volatile Node<E> next;
    }

```

典型的单链表结构，非常纯粹。

主要构造方法

```
    public ConcurrentLinkedQueue() {
        // 初始化头尾节点
        head = tail = new Node<E>(null);
    }

    public ConcurrentLinkedQueue(Collection<? extends E> c) {
        Node<E> h = null, t = null;
        // 遍历c，并把它元素全部添加到单链表中
        for (E e : c) {
            checkNotNull(e);
            Node<E> newNode = new Node<E>(e);
            if (h == null)
                h = t = newNode;
            else {
                t.lazySetNext(newNode);
                t = newNode;
            }
        }
        if (h == null)
            h = t = new Node<E>(null);
        head = h;
        tail = t;
    }

```

这两个构造方法也很简单，可以看到这是一个无界的单链表实现的队列。

入队

因为它不是阻塞队列，所以只有两个入队的方法，add(e)和offer(e)。

因为是无界队列，所以add(e)方法也不用抛出异常了。

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

    public boolean offer(E e) {
        // 不能添加空元素
        checkNotNull(e);
        // 新节点
        final Node<E> newNode = new Node<E>(e);

        // 入队到链表尾
        for (Node<E> t = tail, p = t;;) {
            Node<E> q = p.next;
            // 如果没有next，说明到链表尾部了，就入队
            if (q == null) {
                // CAS更新p的next为新节点
                // 如果成功了，就返回true
                // 如果不成功就重新取next重新尝试
                if (p.casNext(null, newNode)) {
                    // 如果p不等于t，说明有其它线程先一步更新tail
                    // 也就不会走到q==null这个分支了
                    // p取到的可能是t后面的值
```

```
                    // 把tail原子更新为新节点
                    if (p != t) // hop two nodes at a time
                        casTail(t, newNode);  // Failure is OK.
                    // 返回入队成功
                    return true;
                }
            }
            else if (p == q)
                // 如果p的next等于p，说明p已经被删除了（已经出队了）
                // 重新设置p的值
                p = (t != (t = tail)) ? t : head;
            else
                // t后面还有值，重新设置p的值
                p = (p != t && t != (t = tail)) ? t : q;
        }
    }

```

入队整个流程还是比较清晰的，这里有个前提是出队时会把出队的那个节点的next设置为节点本身。

（1）定位到链表尾部，尝试把新节点到后面；

（2）如果尾部变化了，则重新获取尾部，再重试；

出队

因为它不是阻塞队列，所以只有两个出队的方法，remove()和poll()。

```
    public E remove() {
        E x = poll();
        if (x != null)
            return x;
        else
            throw new NoSuchElementException();
    }

    public E poll() {
        restartFromHead:
        for (;;) {
            // 尝试弹出链表的头节点
            for (Node<E> h = head, p = h, q;;) {
                E item = p.item;
                // 如果节点的值不为空，并且将其更新为null成功了
                if (item != null && p.casItem(item, null)) {
                    // 如果头节点变了，则不会走到这个分支
                    // 会先走下面的分支拿到新的头节点
                    // 这时候p就不等于h了，就更新头节点
                    // 在updateHead()中会把head更新为新节点
                    // 并让head的next指向其自己
                    if (p != h) // hop two nodes at a time
                        updateHead(h, ((q = p.next) != null) ? q : p);
                    // 上面的casItem()成功，就可以返回出队的元素了
                    return item;
                }
                // 下面三个分支说明头节点变了
                // 且p的item肯定为null
                else if ((q = p.next) == null) {
                    // 如果p的next为空，说明队列中没有元素了
```

```
                    // 更新h为p，也就是空元素的节点
                    updateHead(h, p);
                    // 返回null
                    return null;
                }
                else if (p == q)
                    // 如果p等于p的next，说明p已经出队了，重试
                    continue restartFromHead;
                else
                    // 将p设置为p的next
                    p = q;
            }
        }
    }
    // 更新头节点的方法
    final void updateHead(Node<E> h, Node<E> p) {
        // 原子更新h为p成功后，延迟更新h的next为它自己
        // 这里用延迟更新是安全的，因为head节点已经变了
        // 只要入队出队的时候检查head有没有变化就行了，跟它的next关系不大
        if (h != p && casHead(h, p))
            h.lazySetNext(h);
    }

```

出队的整个逻辑也是比较清晰的：

（1）定位到头节点，尝试更新其值为null；

（2）如果成功了，就成功出队；

（3）如果失败或者头节点变化了，就重新寻找头节点，并重试；

（4）整个出队过程没有一点阻塞相关的代码，所以出队的时候不会阻塞线程，没找到元素就返回null；

总结

（1）ConcurrentLinkedQueue不是阻塞队列；

（2）ConcurrentLinkedQueue不能用在线程池中；

（3）ConcurrentLinkedQueue使用（CAS+自旋）更新头尾节点控制出队入队操作；

彩蛋

ConcurrentLinkedQueue与LinkedBlockingQueue对比？

（1）两者都是线程安全的队列；

（2）两者都可以实现取元素时队列为空直接返回null，后者的poll()方法可以实现此功能；

（3）前者全程无锁，后者全部都是使用重入锁控制的；

（4）前者效率较高，后者效率较低；

（5）前者无法实现如果队列为空等待元素到来的操作；

（6）前者是非阻塞队列，后者是阻塞队列；

（7）前者无法用在线程池中，后者可以；