## 问题

（1）SynchronousQueue的实现方式?

（2）SynchronousQueue真的是无缓冲的吗?

（3）SynchronousQueue在高并发情景下会有什么问题?

## 简介

SynchronousQueue是java并发包下无缓冲阻塞队列，它用来在两个线程之间移交元素，但是它有个很大的问题，你知道是什么吗？请看下面的分析。

## 源码分析

### 主要属性

```
    // CPU的数量
    static final int NCPUS = Runtime.getRuntime().availableProcessors();
    // 有超时的情况自旋多少次，当CPU数量小于2的时候不自旋
    static final int maxTimedSpins = (NCPUS < 2) ? 0 : 32;
    // 没有超时的情况自旋多少次
    static final int maxUntimedSpins = maxTimedSpins * 16;
    // 针对有超时的情况，自旋了多少次后，如果剩余时间大于1000纳秒就使用带时间的LockSupport.parkNanos()这个方法
    static final long spinForTimeoutThreshold = 1000L;
    // 传输器，即两个线程交换元素使用的东西
    private transient volatile Transferer<E> transferer;
```

```

```

通过属性我们可以Get到两个点：

（1）这个阻塞队列里面是会自旋的；

（2）它使用了一个叫做transferer的东西来交换元素；

### 主要内部类

```
    // Transferer抽象类，主要定义了一个transfer方法用来传输元素
    abstract static class Transferer<E> {
        abstract E transfer(E e, boolean timed, long nanos);
    }
    // 以栈方式实现的Transferer
    static final class TransferStack<E> extends Transferer<E> {
        // 栈中节点的几种类型：
        // 1. 消费者（请求数据的）
        static final int REQUEST    = 0;
        // 2. 生产者（提供数据的）
        static final int DATA       = 1;
        // 3. 二者正在撮合中
        static final int FULFILLING = 2;

        // 栈中的节点
        static final class SNode {
            // 下一个节点
            volatile SNode next;        // next node in stack
            // 匹配者
            volatile SNode match;       // the node matched to this
            // 等待着的线程
```

```
            volatile Thread waiter;     // to control park/unpark
            // 元素
            Object item;                // data; or null for REQUESTs
            // 模式，也就是节点的类型，是消费者，是生产者，还是正在撮合中
            int mode;
        }
        // 栈的头节点
        volatile SNode head;
    }
    // 以队列方式实现的Transferer
    static final class TransferQueue<E> extends Transferer<E> {
        // 队列中的节点
        static final class QNode {
            // 下一个节点
            volatile QNode next;          // next node in queue
            // 存储的元素
            volatile Object item;         // CAS'ed to or from null
            // 等待着的线程
            volatile Thread waiter;       // to control park/unpark
            // 是否是数据节点
            final boolean isData;
        }

        // 队列的头节点
        transient volatile QNode head;
        // 队列的尾节点
        transient volatile QNode tail;
    }

```

(1) 定义了一个抽象类Transferer，里面定义了一个传输元素的方法；

(2) 有两种传输元素的方法，一种是栈，一种是队列；

(3) 栈的特点是后进先出，队列的特点是先进行出；

(4) 栈只需要保存一个头节点就可以了，因为存取元素都是操作头节点；

(5) 队列需要保存一个头节点一个尾节点，因为存元素操作尾节点，取元素操作头节点；

(6) 每个节点中保存着存储的元素、等待着的线程，以及下一个节点；

(7) 栈和队列两种方式有什么不同呢？请看下面的分析。

## 主要构造方法

```
    public SynchronousQueue() {
        // 默认非公平模式
        this(false);
    }

    public SynchronousQueue(boolean fair) {
        // 如果是公平模式就使用队列，如果是非公平模式就使用栈
        transferer = fair ? new TransferQueue<E>() : new TransferStack<E>();
    }

```

(1) 默认使用非公平模式，也就是栈结构；

(2) 公平模式使用队列，非公平模式使用栈；

## 入队

我们这里主要介绍以栈方式实现的传输模式，以put(E e)方法为例。

```
    public void put(E e) throws InterruptedException {
        // 元素不可为空
        if (e == null) throw new NullPointerException();
        // 直接调用传输器的transfer()方法
        // 三个参数分别是：传输的元素，是否需要超时，超时的时间
        if (transferer.transfer(e, false, 0) == null) {
            // 如果传输失败，直接让线程中断并抛出中断异常
            Thread.interrupted();
            throw new InterruptedException();
        }
    }

```

调用transferer的transfer()方法，传入元素e，说明是生产者

## 出队

我们这里主要介绍以栈方式实现的传输模式，以take()方法为例。

```
    public E take() throws InterruptedException {
        // 直接调用传输器的transfer()方法
        // 三个参数分别是：null，是否需要超时，超时的时间
        // 第一个参数为null表示是消费者，要取元素
        E e = transferer.transfer(null, false, 0);
        // 如果取到了元素就返回
        if (e != null)
            return e;
        // 否则让线程中断并抛出中断异常
        Thread.interrupted();
        throw new InterruptedException();
    }

```

调用transferer的transfer()方法，传入null，说明是消费者。

## transfer()方法

transfer()方法同时实现了取元素和放元素的功能，下面我再来看看这个transfer()方法里究竟干了什么。

```
    // TransferStack.transfer()方法
    E transfer(E e, boolean timed, long nanos) {
        SNode s = null; // constructed/reused as needed
        // 根据e是否为null决定是生产者还是消费者
        int mode = (e == null) ? REQUEST : DATA;
        // 自旋+CAS，熟悉的套路，熟悉的味道
        for (;;) {
            // 栈顶元素
            SNode h = head;
            // 栈顶没有元素，或者栈顶元素跟当前元素是一个模式的
            // 也就是都是生产者节点或者都是消费者节点
            if (h == null || h.mode == mode) {  // empty or same-mode
                // 如果有超时而且已到期
                if (timed && nanos <= 0) {      // can't wait
                    // 如果头节点不为空且是取消状态
                    if (h != null && h.isCancelled())
                        // 就把头节点弹出，并进入下一次循环
                        casHead(h, h.next);     // pop cancelled node
                    else
                        // 否则，直接返回null（超时返回null）
                        return null;
                } else if (casHead(h, s = snode(s, e, h, mode))) {
```

```
                    // 入栈成功（因为是模式相同的，所以只能入栈）
                    // 调用awaitFulfill()方法自旋+阻塞当前入栈的线程并等待被匹配到
                    SNode m = awaitFulfill(s, timed, nanos);
                    // 如果m等于s，说明取消了，那么就把它清除掉，并返回null
                    if (m == s) {               // wait was cancelled
                        clean(s);
                        // 被取消了返回null
                        return null;
                    }

                    // 到这里说明匹配到元素了
                    // 因为从awaitFulfill()里面出来要不被取消了要不就匹配到了

                    // 如果头节点不为空，并且头节点的下一个节点是s
                    // 就把头节点换成s的下一个节点
                    // 也就是把h和s都弹出了
                    // 也就是把栈顶两个元素都弹出了
                    if ((h = head) != null && h.next == s)
                        casHead(h, s.next);     // help s's fulfiller
                    // 根据当前节点的模式判断返回m还是s中的值
                    return (E) ((mode == REQUEST) ? m.item : s.item);
                }
            } else if (!isFulfilling(h.mode)) { // try to fulfill
                // 到这里说明头节点和当前节点模式不一样
                // 如果头节点不是正在撮合中

                // 如果头节点已经取消了，就把它弹出栈
                if (h.isCancelled())            // already cancelled
                    casHead(h, h.next);         // pop and retry
                else if (casHead(h, s=snode(s, e, h, FULFILLING|mode))) {
                    // 头节点没有在撮合中，就让当前节点先入队，再让他们尝试匹配
                    // 且s成为了新的头节点，它的状态是正在撮合中
```

```
                    for (;;) { // loop until matched or waiters disappear
                        SNode m = s.next;       // m is s's match
                        // 如果m为null，说明除了s节点外的节点都被其它线程先一步撮合掉了
                        // 就清空栈并跳出内部循环，到外部循环再重新入栈判断
                        if (m == null) {        // all waiters are gone
                            casHead(s, null);   // pop fulfill node
                            s = null;           // use new node next time
                            break;              // restart main loop
                        }
                        SNode mn = m.next;
                        // 如果m和s尝试撮合成功，就弹出栈顶的两个元素m和s
                        if (m.tryMatch(s)) {
                            casHead(s, mn);     // pop both s and m
                            // 返回撮合结果
                            return (E) ((mode == REQUEST) ? m.item : s.item);
                        } else                  // lost match
                            // 尝试撮合失败，说明m已经先一步被其它线程撮合了
                            // 就协助清除它
                            s.casNext(m, mn);   // help unlink
                    }
                }
            } else {                            // help a fulfiller
                // 到这里说明当前节点和头节点模式不一样
                // 且头节点是正在撮合中

                SNode m = h.next;               // m is h's match
                if (m == null)                  // waiter is gone
                    // 如果m为null，说明m已经被其它线程先一步撮合了
                    casHead(h, null);           // pop fulfilling node
                else {
                    SNode mn = m.next;
                    // 协助匹配，如果m和s尝试撮合成功，就弹出栈顶的两个元素m和s
```

```
                    if (m.tryMatch(h))          // help match
                        // 将栈顶的两个元素弹出后，再让s重新入栈
                        casHead(h, mn);         // pop both h and m
                    else                        // lost match
                        // 尝试撮合失败，说明m已经先一步被其它线程撮合了
                        // 就协助清除它
                        h.casNext(m, mn);       // help unlink
                }
            }
        }
    }

    // 三个参数：需要等待的节点，是否需要超时，超时时间
    SNode awaitFulfill(SNode s, boolean timed, long nanos) {
        // 到期时间
        final long deadline = timed ? System.nanoTime() + nanos : 0L;
        // 当前线程
        Thread w = Thread.currentThread();
        // 自旋次数
        int spins = (shouldSpin(s) ?
                     (timed ? maxTimedSpins : maxUntimedSpins) : 0);
        for (;;) {
            // 当前线程中断了，尝试清除s
            if (w.isInterrupted())
                s.tryCancel();

            // 检查s是否匹配到了元素m（有可能是其它线程的m匹配到当前线程的s）
            SNode m = s.match;
            // 如果匹配到了，直接返回m
            if (m != null)
                return m;

            // 如果需要超时
```

```
            if (timed) {
                // 检查超时时间如果小于0了，尝试清除s
                nanos = deadline - System.nanoTime();
                if (nanos <= 0L) {
                    s.tryCancel();
                    continue;
                }
            }
            if (spins > 0)
                // 如果还有自旋次数，自旋次数减一，并进入下一次自旋
                spins = shouldSpin(s) ? (spins-1) : 0;

            // 后面的elseif都是自旋次数没有了
            else if (s.waiter == null)
                // 如果s的waiter为null，把当前线程注入进去，并进入下一次自旋
                s.waiter = w; // establish waiter so can park next iter
            else if (!timed)
                // 如果不允许超时，直接阻塞，并等待被其它线程唤醒，唤醒后继续自旋并查看是否匹配到了元素
                LockSupport.park(this);
            else if (nanos > spinForTimeoutThreshold)
                // 如果允许超时且还有剩余时间，就阻塞相应时间
                LockSupport.parkNanos(this, nanos);
        }
    }

        // SNode里面的方向，调用者m是s的下一个节点
        // 这时候m节点的线程应该是阻塞状态的
        boolean tryMatch(SNode s) {
            // 如果m还没有匹配者，就把s作为它的匹配者
            if (match == null &&
                UNSAFE.compareAndSwapObject(this, matchOffset, null, s)) {
                Thread w = waiter;
```

```
                if (w != null) {    // waiters need at most one unpark
                    waiter = null;
                    // 唤醒m中的线程，两者匹配完毕
                    LockSupport.unpark(w);
                }
                // 匹配到了返回true
                return true;
            }
            // 可能其它线程先一步匹配了m，返回其是否是s
            return match == s;
        }

```

整个逻辑比较复杂，这里为了简单起见，屏蔽掉多线程处理的细节，只描述正常业务场景下的逻辑：

（1）如果栈中没有元素，或者栈顶元素跟将要入栈的元素模式一样，就入栈；

（2）入栈后自旋等待一会看有没有其它线程匹配到它，自旋完了还没匹配到元素就阻塞等待；

（3）阻塞等待被唤醒了说明其它线程匹配到了当前的元素，就返回匹配到的元素；

（4）如果两者模式不一样，且头节点没有在匹配中，就拿当前节点跟它匹配，匹配成功了就返回匹配到的元素；

（5）如果两者模式不一样，且头节点正在匹配中，当前线程就协助去匹配，匹配完成了再让当前节点重新入栈重新匹配；

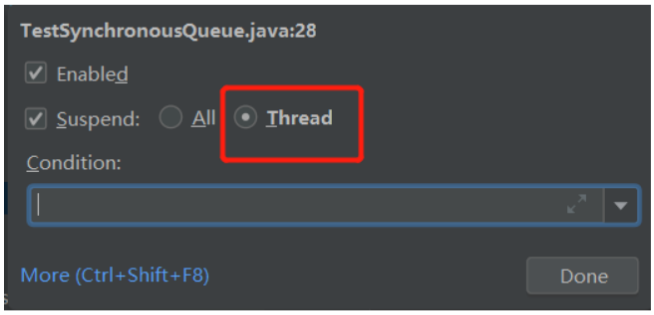
如果直接阅读这部分代码还是比较困难的，建议写个测试用例，打个断点一步一步跟踪调试。

下面是我的测试用例，可以参考下，在IDEA中可以让断点只阻塞线程：

```
    public class TestSynchronousQueue {
        public static void main(String[] args) throws InterruptedException {
            SynchronousQueue<Integer> queue = new SynchronousQueue<>(false);

```

```
            new Thread(()->{
                try {
                    queue.put(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();


            Thread.sleep(500);
            System.out.println(queue.take());
        }
    }

```

修改断点只阻塞线程的方法，右击断点，选择Thread：




## 交给你了

上面的源码分析都是基于Stack的方式来分析的，那么队列是怎么动作的呢？很简单哦，测试用例中的false改成true就可以了，这就交给你了。

## 总结

（1）SynchronousQueue是java里的无缓冲队列，用于在两个线程之间直接移交元素；

（2）SynchronousQueue有两种实现方式，一种是公平（队列）方式，一种是非公平（栈）方式；

（3）栈方式中的节点有三种模式：生产者、消费者、正在匹配中；

（4）栈方式的大致思路是如果栈顶元素跟自己一样的模式就入栈并等待被匹配，否则就匹配，匹配到了就返回；

（5）队列方式的大致思路是……不告诉你^^（两者的逻辑差别还是挺大的）

## 彩蛋

（1）SynchronousQueue真的是无缓冲的队列吗？

通过源码分析，我们可以发现其实SynchronousQueue内部或者使用栈或者使用队列来存储包含线程和元素值的节点，如果同一个模式的节点过多的话，它们都会存储进来，且都会阻塞着，所以，严格上来说，SynchronousQueue并不能算是一个无缓冲队列。

（2）SynchronousQueue有什么缺点呢？

试想一下，如果有多个生产者，但只有一个消费者，如果消费者处理不过来，是不是生产者都会阻塞起来？反之亦然。

这是一件很危险的事，所以，SynchronousQueue一般用于生产、消费的速度大致相当的情况，这样才不会导致系统中过多的线程处于阻塞状态。