问题

（1）LinkedTransferQueue是什么东东?

（2）LinkedTransferQueue是怎么实现阻塞队列的?

（3）LinkedTransferQueue是怎么控制并发安全的?

（4）LinkedTransferQueue与SynchronousQueue有什么异同?

简介

LinkedTransferQueue是LinkedBlockingQueue、SynchronousQueue（公平模式）、ConcurrentLinkedQueue三者的集合体，它综合了这三者的方法，并且提供了更加高效的实现方式。

继承体系

LinkedTransferQueue实现了TransferQueue接口，而TransferQueue接口是继承自BlockingQueue的，所以LinkedTransferQueue也是一个阻塞队列。

TransferQueue接口中定义了以下几个方法：

```
1.     // 尝试移交元素
2.     boolean tryTransfer(E e);
3.     // 移交元素
4.     void transfer(E e) throws InterruptedException;
5.     // 尝试移交元素（有超时时间）
6.     boolean tryTransfer(E e, long timeout, TimeUnit unit)
7.         throws InterruptedException;
8.     // 判断是否有消费者
9.     boolean hasWaitingConsumer();
10.    // 查看消费者的数量
11.    int getWaitingConsumerCount();
12.
```

主要是定义了三个移交元素的方法，有阻塞的，有不阻塞的，有超时的。

存储结构

LinkedTransferQueue使用了一个叫做 `dual data structure` 的数据结构，或者叫做 `dual queue` ，译为双重数据结构或者双重队列。

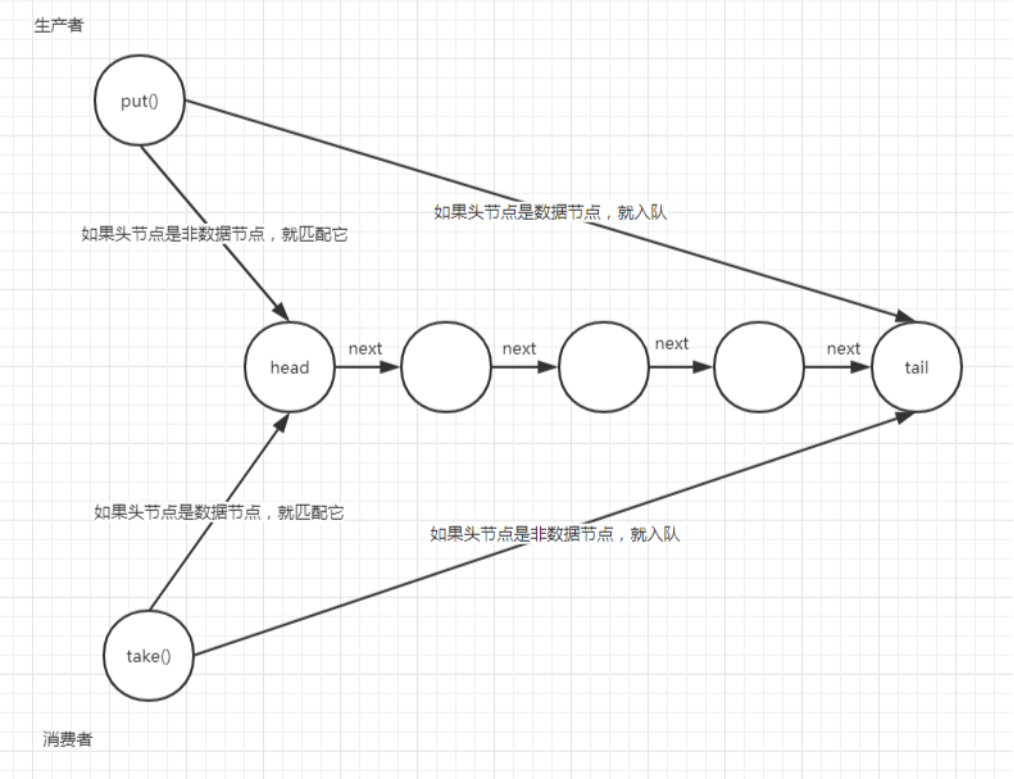
双重队列是什么意思呢?

放取元素使用同一个队列，队列中的节点具有两种模式，一种是数据节点，一种是非数据节点。

放元素时先跟队列头节点对比，如果头节点是非数据节点，就让他们匹配，如果头节点是数据节点，就生成一个数据节点放在队列尾端（入队）。

取元素时也是先跟队列头节点对比，如果头节点是数据节点，就让他们匹配，如果头节点是非数据节点，就生成一个非数据节点放在队列尾端（入队）。

用图形来表示就是下面这样：



不管是放元素还是取元素，都先跟头节点对比，如果二者模式不一样就匹配它们，如果二者模式一样，就入队。

源码分析

主要属性

```
    // 头节点
    transient volatile Node head;
    // 尾节点
    private transient volatile Node tail;
    // 放取元素的几种方式：
    // 立即返回，用于非超时的poll()和tryTransfer()方法中
    private static final int NOW   = 0; // for untimed poll, tryTransfer
    // 异步，不会阻塞，用于放元素时，因为内部使用无界单链表存储元素，不会阻塞放元素的过程
    private static final int ASYNC = 1; // for offer, put, add
    // 同步，调用的时候如果没有匹配到会阻塞直到匹配到为止
    private static final int SYNC  = 2; // for transfer, take
    // 超时，用于有超时的poll()和tryTransfer()方法中
    private static final int TIMED = 3; // for timed poll, tryTransfer

```

主要内部类

```
    static final class Node {
        // 是否是数据节点（也就标识了是生产者还是消费者）
        final boolean isData;   // false if this is a request node
```

```
        // 元素的值
        volatile Object item;    // initially non-null if isData; CASed to match
        // 下一个节点
        volatile Node next;
        // 持有元素的线程
        volatile Thread waiter; // null until waiting
    }

```

典型的单链表结构，内部除了存储元素的值和下一个节点的指针外，还包含了是否为数据节点和持有元素的线程。

内部通过isData区分是生产者还是消费者。

主要构造方法

```
    public LinkedTransferQueue() {
    }

    public LinkedTransferQueue(Collection<? extends E> c) {
        this();
        addAll(c);
    }

```

只有这两个构造方法，且没有初始容量，所以是无界的一个阻塞队列。

入队

四个方法都是一样的，使用异步的方式调用xfer()方法，传入的参数都一模一样。

```
    public void put(E e) {
        // 异步模式，不会阻塞，不会超时
        // 因为是放元素，单链表存储，会一直往后加
        xfer(e, true, ASYNC, 0);
    }

    public boolean offer(E e, long timeout, TimeUnit unit) {
        xfer(e, true, ASYNC, 0);
        return true;
    }

    public boolean offer(E e) {
        xfer(e, true, ASYNC, 0);
        return true;
    }

    public boolean add(E e) {
        xfer(e, true, ASYNC, 0);
        return true;
    }

```

xfer(E e, boolean haveData, int how, long nanos)的参数分别是：

（1）e表示元素；

（2）haveData表示是否是数据节点，

（3）how表示放取元素的方式，上面提到的四种，NOW、ASYNC、SYNC、TIMED；

（4）nanos表示超时时间；

出队

出队的四个方法也是直接或间接的调用xfer()方法，放取元素的方式和超时规则略微不同，本质没有大的区别。

```
    public E remove() {
        E x = poll();
        if (x != null)
            return x;
        else
            throw new NoSuchElementException();
    }
    public E take() throws InterruptedException {
        // 同步模式，会阻塞直到取到元素
        E e = xfer(null, false, SYNC, 0);
        if (e != null)
            return e;
        Thread.interrupted();
        throw new InterruptedException();
    }

    public E poll(long timeout, TimeUnit unit) throws InterruptedException {
        // 有超时时间
        E e = xfer(null, false, TIMED, unit.toNanos(timeout));
        if (e != null || !Thread.interrupted())
            return e;
        throw new InterruptedException();
    }

    public E poll() {
        // 立即返回，没取到元素返回null
        return xfer(null, false, NOW, 0);
    }

```

取元素就各有各的玩法了，有同步的，有超时的，有立即返回的。

移交元素的方法

```
    public boolean tryTransfer(E e) {
        // 立即返回
        return xfer(e, true, NOW, 0) == null;
    }

    public void transfer(E e) throws InterruptedException {
        // 同步模式
        if (xfer(e, true, SYNC, 0) != null) {
            Thread.interrupted(); // failure possible only due to interrupt
```

```
            throw new InterruptedException();
        }
    }

    public boolean tryTransfer(E e, long timeout, TimeUnit unit)
        throws InterruptedException {
        // 有超时时间
        if (xfer(e, true, TIMED, unit.toNanos(timeout)) == null)
            return true;
        if (!Thread.interrupted())
            return false;
        throw new InterruptedException();
    }

```

请注意第二个参数，都是true，也就是这三个方法其实也是放元素的方法。

这里xfer()方法的几种模式到底有什么区别呢？请看下面的分析。

神奇的xfer()方法

```
    private E xfer(E e, boolean haveData, int how, long nanos) {
        // 不允许放入空元素
        if (haveData && (e == null))
            throw new NullPointerException();
        Node s = null;                        // the node to append, if needed
        // 外层循环，自旋，失败就重试
        retry:
        for (;;) {                            // restart on append race

            // 下面这个for循环用于控制匹配的过程
            // 同一时刻队列中只会存储一种类型的节点
            // 从头节点开始尝试匹配，如果头节点被其它线程先一步匹配了
            // 就再尝试其下一个，直到匹配到为止，或者到队列中没有元素为止

            for (Node h = head, p = h; p != null;) { // find & match first node
                // p节点的模式
                boolean isData = p.isData;
                // p节点的值
                Object item = p.item;
                // p没有被匹配到
                if (item != p && (item != null) == isData) { // unmatched
                    // 如果两者模式一样，则不能匹配，跳出循环后尝试入队
                    if (isData == haveData)   // can't match
                        break;
                    // 如果两者模式不一样，则尝试匹配
                    // 把p的值设置为e（如果是取元素则e是null，如果是放元素则e是元素值）
                    if (p.casItem(item, e)) { // match
                        // 匹配成功
                        // for里面的逻辑比较复杂，用于控制多线程同时放取元素时出现竞争的情况的
                        // 看不懂可以直接跳过
                        for (Node q = p; q != h;) {
                            // 进入到这里可能是头节点已经被匹配，然后p会变成h的下一个节点
                            Node n = q.next;  // update by 2 unless singleton
                            // 如果head还没变，就把它更新成新的节点
                            // 并把它删除（forgetNext()会把它的next设为自己，也就是从单链表中删除了）
                            // 这时为什么要把head设为n呢？因为到这里了，肯定head本身已经被匹配掉了
```

```
                            // 这时为什么要把head设为n呢？因为到这里了，肯定head本身已经被匹配掉了
                            // 而上面的p.casItem()又成功了，说明p也被当前这个元素给匹配掉了
                            // 所以需要把它们俩都出队列，让其它线程可以从真正的头开始，不用重复检查了
                            if (head == h && casHead(h, n == null ? q : n)) {
                                h.forgetNext();
                                break;
                            }                 // advance and retry
                            // 如果新的头节点为空，或者其next为空，或者其next未匹配，就重试
                            if ((h = head)   == null ||
                                (q = h.next) == null || !q.isMatched())
                                break;        // unless slack < 2
                        }
                        // 唤醒p中等待的线程
                        LockSupport.unpark(p.waiter);
                        // 并返回匹配到的元素
                        return LinkedTransferQueue.<E>cast(item);
                    }
                }
                // p已经被匹配了或者尝试匹配的时候失败了
                // 也就是其它线程先一步匹配了p
                // 这时候又分两种情况，p的next还没来得及修改，p的next指向了自己
                // 如果p的next已经指向了自己，就重新取head重试，否则就取其next重试
                Node n = p.next;
                p = (p != n) ? n : (h = head); // Use head if p offlist
            }

            // 到这里肯定是队列中存储的节点类型和自己一样
            // 或者队列中没有元素了
            // 就入队（不管放元素还是取元素都得入队）
```

```
            // 入队又分成四种情况：
            // NOW，立即返回，没有匹配到立即返回，不做入队操作
            // ASYNC，异步，元素入队但当前线程不会阻塞（相当于无界LinkedBlockingQueue的元素入队）
            // SYNC，同步，元素入队后当前线程阻塞，等待被匹配到
            // TIMED，有超时，元素入队后等待一段时间被匹配，时间到了还没匹配到就返回元素本身

            // 如果不是立即返回
            if (how != NOW) {                 // No matches available
                // 新建s节点
                if (s == null)
                    s = new Node(e, haveData);
                // 尝试入队
                Node pred = tryAppend(s, haveData);
                // 入队失败，重试
                if (pred == null)
                    continue retry;           // lost race vs opposite mode
                // 如果不是异步（同步或者有超时）
                // 就等待被匹配
                if (how != ASYNC)
                    return awaitMatch(s, pred, e, (how == TIMED), nanos);
            }
            return e; // not waiting
        }
    }

    private Node tryAppend(Node s, boolean haveData) {
        // 从tail开始遍历，把s放到链表尾端
        for (Node t = tail, p = t;;) {        // move p to last node and append
            Node n, u;                        // temps for reads of next & tail
            // 如果首尾都是null，说明链表中还没有元素
```

```
            if (p == null && (p = head) == null) {
                // 就让首节点指向s
                // 注意，这里插入第一个元素的时候tail指针并没有指向s
                if (casHead(null, s))
                    return s;                 // initialize
            }
            else if (p.cannotPrecede(haveData))
                // 如果p无法处理，则返回null
                // 这里无法处理的意思是，p和s节点的类型不一样，不允许s入队
                // 比如，其它线程先入队了一个数据节点，这时候要入队一个非数据节点，就不允许，
                // 队列中所有的元素都要保证是同一种类型的节点
                // 返回null后外面的方法会重新尝试匹配重新入队等
                return null;                  // lost race vs opposite mode
            else if ((n = p.next) != null)    // not last; keep traversing
                // 如果p的next不为空，说明不是最后一个节点
                // 则让p重新指向最后一个节点
                p = p != t && t != (u = tail) ? (t = u) : // stale tail
                    (p != n) ? n : null;      // restart if off list
            else if (!p.casNext(null, s))
                // 如果CAS更新s为p的next失败
                // 则说明有其它线程先一步更新到p的next了
                // 就让p指向p的next，重新尝试让s入队
                p = p.next;                   // re-read on CAS failure
            else {
                // 到这里说明s成功入队了
                // 如果p不等于t，就更新tail指针
                // 还记得上面插入第一个元素时tail指针并没有指向新元素吗？
                // 这里就是用来更新tail指针的
                if (p != t) {                 // update if slack now >= 2
                    while ((tail != t || !casTail(t, s)) &&
```

```
                           (t = tail)   != null &&
                           (s = t.next) != null && // advance and retry
                           (s = s.next) != null && s != t);
                }
                // 返回p，即s的前一个元素
                return p;
            }
        }
    }

    private E awaitMatch(Node s, Node pred, E e, boolean timed, long nanos) {
        // 如果是有超时的，计算其超时时间
        final long deadline = timed ? System.nanoTime() + nanos : 0L;
        // 当前线程
        Thread w = Thread.currentThread();
        // 自旋次数
        int spins = -1; // initialized after first item and cancel checks
        // 随机数，随机让一些自旋的线程让出CPU
        ThreadLocalRandom randomYields = null; // bound if needed

        for (;;) {
            Object item = s.item;
            // 如果s元素的值不等于e，说明它被匹配到了
            if (item != e) {                  // matched
                // assert item != s;
                // 把s的item更新为s本身
                // 并把s中的waiter置为空
                s.forgetContents();           // avoid garbage
                // 返回匹配到的元素
                return LinkedTransferQueue.<E>cast(item);
```

```
            }
            // 如果当前线程中断了，或者有超时的到期了
            // 就更新s的元素值指向s本身
            if ((w.isInterrupted() || (timed && nanos <= 0)) &&
                    s.casItem(e, s)) {        // cancel
                // 尝试解除s与其前一个节点的关系
                // 也就是删除s节点
                unsplice(pred, s);
                // 返回元素的值本身，说明没匹配到
                return e;
            }

            // 如果自旋次数小于0，就计算自旋次数
            if (spins < 0) {                  // establish spins at/near front
                // spinsFor()计算自旋次数
                // 如果前面有节点未被匹配就返回0
                // 如果前面有节点且正在匹配中就返回一定的次数，等待
                if ((spins = spinsFor(pred, s.isData)) > 0)
                    // 初始化随机数
                    randomYields = ThreadLocalRandom.current();
            }
            else if (spins > 0) {             // spin
                // 还有自旋次数就减1
                --spins;
                // 并随机让出CPU
                if (randomYields.nextInt(CHAINED_SPINS) == 0)
                    Thread.yield();           // occasionally yield
            }
            else if (s.waiter == null) {
```

```
                // 更新s的waiter为当前线程
                s.waiter = w;                 // request unpark then recheck
            }
            else if (timed) {
                // 如果有超时，计算超时时间，并阻塞一定时间
                nanos = deadline - System.nanoTime();
                if (nanos > 0L)
                    LockSupport.parkNanos(this, nanos);
            }
            else {
                // 不是超时的，直接阻塞，等待被唤醒
                // 唤醒后进入下一次循环，走第一个if的逻辑就返回匹配的元素了
                LockSupport.park(this);
            }
        }
    }

```

这三个方法里的内容特别复杂，很大一部分代码都是在控制线程安全，各种CAS，我们这里简单描述一下大致的逻辑：

（1）来了一个元素，我们先查看队列头的节点，是否与这个元素的模式一样；

（2）如果模式不一样，就尝试让他们匹配，如果头节点被别的线程先匹配走了，就尝试与头节点的下一个节点匹配，如此一直往后，直到匹配到或到链表尾为止；

（3）如果模式一样，或者到链表尾了，就尝试入队；

（4）入队的时候有可能链表尾修改了，那就尾指针后移，再重新尝试入队，依此往复；

（5）入队成功了，就自旋或阻塞，阻塞了就等待被其它线程匹配到并唤醒；

（6）唤醒之后进入下一次循环就匹配到元素了，返回匹配到的元素；

（7）是否需要入队及阻塞有四种情况：

```
    a）NOW，立即返回，没有匹配到立即返回，不做入队操作

        对应的方法有：poll()、tryTransfer(e)

    b）ASYNC，异步，元素入队但当前线程不会阻塞（相当于无界LinkedBlockingQueue的元素入队）

        对应的方法有：add(e)、offer(e)、put(e)、offer(e, timeout, unit)

    c）SYNC，同步，元素入队后当前线程阻塞，等待被匹配到

        对应的方法有：take()、transfer(e)

    d）TIMED，有超时，元素入队后等待一段时间被匹配，时间到了还没匹配到就返回元素本身

        对应的方法有：poll(timeout, unit)、tryTransfer(e, timeout, unit)

```

总结

（1）LinkedTransferQueue可以看作LinkedBlockingQueue、SynchronousQueue（公平模式）、ConcurrentLinkedQueue三者的集合体；

（2）LinkedTransferQueue的实现方式是使用一种叫做 `双重队列` 的数据结构；

（3）不管是取元素还是放元素都会入队；

（4）先尝试跟头节点比较，如果二者模式不一样，就匹配它们，组成CP，然后返回对方的值；

（5）如果二者模式一样，就入队，并自旋或阻塞等待被唤醒；

（6）至于是否入队及阻塞有四种模式，NOW、ASYNC、SYNC、TIMED；

（7）LinkedTransferQueue全程都没有使用synchronized、重入锁等比较重的锁，基本是通过 自旋+CAS 实现；

（8）对于入队之后，先自旋一定次数后再调用LockSupport.park()或LockSupport.parkNanos阻塞；

彩蛋

LinkedTransferQueue与SynchronousQueue（公平模式）有什么异同呢?

（1）在java8中两者的实现方式基本一致，都是使用的双重队列；

（2）前者完全实现了后者，但比后者更灵活；

（3）后者不管放元素还是取元素，如果没有可匹配的元素，所在的线程都会阻塞；

（4）前者可以自己控制放元素是否需要阻塞线程，比如使用四个添加元素的方法就不会阻塞线程，只入队元素，使用transfer()会阻塞线程；

（5）取元素两者基本一样，都会阻塞等待有新的元素进入被匹配到；