阻塞队列-SynchronousQueue

简介

参考：https://blog.csdn.net/chenssy/article/details/77371992

> SynchronousQueue是一个不存储元素的队列。每一个put操作必须等待一个take操作，反之亦然。所以其特定是没有容量，不能peek查看，不支持空元素。
>
> 默认情况下，线程的等待唤醒是非公平的，可以设置成公平模式，保证线程是先入先出（先到先得）。通常两种模式的性能差不多，非公平模式可以维持更多的线程，公平模式则支持更高的吞吐量。
>
> 它支持公平访问队列。默认情况下线程采用非公平性策略访问队列。SynchronousQueue类只有两个构造方法：

1. 通过TransferQueue来实现公平，TransferStack实现非公平。

```java
public SynchronousQueue(boolean fair) {
    transferer = fair ? new TransferQueue<E>() : new TransferStack<E>();
}
```

```java
public SynchronousQueue() {
    this(false);
}
```

2.

公共的父类Transferer

```java
abstract static class Transferer<E> {
        /**
         * Performs a put or take.
         *
         * @param e if non-null, the item to be handed to a consumer;
         *          if null, requests that transfer return an item
         *          offered by producer.
         * @param timed if this operation should timeout
         * @param nanos the timeout, in nanoseconds
         * @return if non-null, the item provided or received; if null,
         *         the operation failed due to timeout or interrupt --
         *         the caller can distinguish which of these occurred
         *         by checking Thread.interrupted.
         */
        abstract E transfer(E e, boolean timed, long nanos);
    }
```

3.put，不能插入null。中断返回。向队列中插入元素，直到有消费者消费才返回。

transferer.transfer(e, false, 0) e不为空，为空则变成了take了。

```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    if (transferer.transfer(e, false, 0) == null) {
        Thread.interrupted();
        throw new InterruptedException();
    }
}
```

4.take 检索并返回队列中的头节点，如果没有匹配的，则等待别的线程插入。

transferer.transfer(null, false, 0); e为null

```
/**
 * Retrieves and removes the head of this queue, waiting if necessary
 * for another thread to insert it.
 *
 * @return the head of this queue
 * @throws InterruptedException {@inheritDoc}
 */
public E take() throws InterruptedException {
    E e = transferer.transfer(null, false, 0);
    if (e != null)
        return e;
    Thread.interrupted();
    throw new InterruptedException();
}
```

非公平的实现

```
static final class TransferStack<E> extends Transferer<E> {
        /*
         * This extends Scherer-Scott dual stack algorithm, differing,
         * among other ways, by using "covering" nodes rather than
         * bit-marked pointers: Fulfilling operations push on marker
         * nodes (with FULFILLING bit set in mode) to reserve a spot
         * to match a waiting node.
         */
        static final int REQUEST    = 0; //代表消费者
        static final int DATA       = 1; //代表生产者
        static final int FULFILLING = 2; //正在交易
```

节点

```
static final class SNode {
            volatile SNode next;        // next node in stack
            volatile SNode match;       // the node matched to this
            volatile Thread waiter;     // to control park/unpark
            Object item;                // data; or null for REQUESTs
            int mode;    // 节点模式： REQUEST、DATA、FUFILLING
            // Note: item and mode fields don't need to be volatile
            // since they are always written before, and read after,
            // other volatile/atomic operations.
            SNode(Object item) {
                this.item = item;
            }
```

节点中的transfer方法：

```
/**
 * Puts or takes an item.
 */
@SuppressWarnings("unchecked")
E transfer(E e, boolean timed, long nanos) {
    /*
     * Basic algorithm is to loop trying one of three actions:
     *
     * 1. If apparently empty or already containing nodes of same
     *    mode, try to push node on stack and wait for a match,
     *    returning it, or null if cancelled.
     *
     * 2. If apparently containing node of complementary mode,
     *    try to push a fulfilling node on to stack, match
     *    with corresponding waiting node, pop both from
     *    stack, and return matched item. The matching or
     *    unlinking might not actually be necessary because of
     *    other threads performing action 3:
     *
     * 3. If top of stack already holds another fulfilling node,
     *    help it out by doing its match and/or pop
     *    operations, and then continue. The code for helping
     *    is essentially the same as for fulfilling, except
     *    that it doesn't return the item.
     */

    SNode s = null; // constructed/reused as needed
    int mode = (e == null) ? REQUEST : DATA; //是消费者，还是生产者？（take，put？）

    for (;;) {
        SNode h = head;
        // 栈为空或者当前节点模式与头节点模式一样，将节点压入栈内，等待匹配
        if (h == null || h.mode == mode) {  // empty or same-mode
            // 超时
            if (timed && nanos <= 0) {      // can't wait
                // 节点被取消了，向前推进
                if (h != null && h.isCancelled())
```

```
                    //  重新设置头结点（弹出之前的头结点）
                    casHead(h, h.next);     // pop cancelled node
                else
                    return null;
            // 不超时
            // 生成一个SNode节点，并尝试替换掉头节点head (head -> s)
            } else if (casHead(h, s = snode(s, e, h, mode))) {
                // 自旋，等待线程匹配
                SNode m = awaitFulfill(s, timed, nanos);
                // 返回的m == s 表示该节点被取消了或者超时、中断了
                if (m == s) {               // wait was cancelled
                    clean(s); // 清理节点S，return null
                    return null;
                }
                // 因为通过前面一步将S替换成了head，如果h.next == s ，则表示有其他节点插入到S前面了,变成了head
                // 且该节点就是与节点S匹配的节点
                if ((h = head) != null && h.next == s)
                    // 将s.next节点设置为head，相当于取消节点h、s
                    casHead(h, s.next);     // help s's fulfiller
                // 如果是请求则返回匹配的域，否则返回节点S的域
                return (E) ((mode == REQUEST) ? m.item : s.item);
            }
        // 如果栈不为null，且两者模式不匹配（h != null && h.mode != mode）
        // 说明他们是一队对等匹配的节点，尝试用当前节点s来满足h节点
        } else if (!isFulfilling(h.mode)) { // try to fulfill
            // head 节点已经取消了，向前推进
            if (h.isCancelled())            // already cancelled
                casHead(h, h.next);         // pop and retry
             // 尝试将当前节点打上"正在匹配"的标记，并设置为head
            else if (casHead(h, s=snode(s, e, h, FULFILLING|mode))) {
                // 循环loop
                for (;;) { // loop until matched or waiters disappear
                    // s为当前节点，m是s的next节点，
                    // m节点是s节点的匹配节点
                    SNode m = s.next;       // m is s's match
                    // m == null，其他节点把m节点匹配走了
                    if (m == null) {        // all waiters are gone
                        // 将s弹出
                        casHead(s, null);   // pop fulfill node
                        // 将s置空，下轮循环的时候还会新建
                        s = null;           // use new node next time
                        // 退出该循环，继续主循环
                        break;              // restart main loop
                    }
                    // 获取m的next节点
                    SNode mn = m.next;
                    // 尝试匹配
                    if (m.tryMatch(s)) {
                        // 匹配成功，将s 、 m弹出
                        casHead(s, mn);     // pop both s and m
                        return (E) ((mode == REQUEST) ? m.item : s.item);
                    } else                  // lost match
```

```
                        // 如果没有匹配成功，说明有其他线程已经匹配了，把m移出
                        s.casNext(m, mn);   // help unlink
                }
            }
        // 到这最后一步说明节点正在匹配阶段
        } else {                            // help a fulfiller
            // head 的next的节点，是正在匹配的节点，m 和 h配对
            SNode m = h.next;               // m is h's match
            // m == null 其他线程把m节点抢走了，弹出h节点
            if (m == null)                  // waiter is gone
                casHead(h, null);           // pop fulfilling node
            else {
                SNode mn = m.next;
                if (m.tryMatch(h))          // help match
                    casHead(h, mn);         // pop both h and m
                else                        // lost match
                    h.casNext(m, mn);       // help unlink
            }
        }
    }
}
```

整个处理过程分为三种情况，具体如下：

1. 如果当前栈为空获取节点模式与栈顶模式一样，则尝试将节点加入栈内，同时通过自旋方式等待节点匹配，最后返回匹配的节点或者null（被取消）
2. 如果栈不为空且节点的模式与首节点模式匹配，则尝试将该节点打上FULFILLING标记，然后加入栈中，与相应的节点匹配，成功后将这两个节点弹出栈并返回匹配节点的数据
3. 如果有节点在匹配，那么帮助这个节点完成匹配和出栈操作，然后在主循环中继续执行

公平

队列实现。ransferQueue队列中永远会存在一个 dummy node。

整个transfer的算法如下：

1. 如果队列为null或者尾节点模式与当前节点模式一致，则尝试将节点加入到等待队列中（采用自旋的方式），直到被匹配或、超时或者取消。匹配成功的话要么返回null（producer返回的）要么返回真正传递的值（consumer返回的），如果返回的是node节点本身则表示当前线程超时或者取消了。
2. 如果队列不为null，且队列的节点是当前节点匹配的节点，则进行数据的传递匹配并返回匹配节点的数据
3. 在整个过程中都会检测并帮助其他线程推进