[https://mp.weixin.qq.com/s?\_\_biz=MzA5MTkxMDQ4MQ==&mid=2648933190&idx=1&sn=916f539cb1e695948169a358549227d3&chksm=88621b78bf15926e0a94e50a43651dab0ceb14a1fb6b1d8b9b75e38c6d8ac908e31dd2131ded&token=1963100670&lang=zh\_CN&scene=21#wechat\_redirect](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933190&idx=1&sn=916f539cb1e695948169a358549227d3&chksm=88621b78bf15926e0a94e50a43651dab0ceb14a1fb6b1d8b9b75e38c6d8ac908e31dd2131ded&token=1963100670&lang=zh_CN&scene=21" \l "wechat_redirect)

# JUC中的阻塞队列

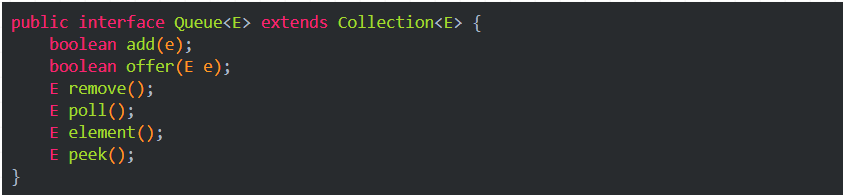
## **本文内容**

* 掌握Queue、BlockingQueue接口中常用的方法
* 介绍6种阻塞队列，及相关场景示例
* 重点掌握4种常用的阻塞队列

## **Queue接口**

队列是一种先进先出（FIFO）的数据结构，java中用Queue接口来表示队列。

Queue接口中定义了6个方法：



每个Queue方法都有两种形式：

（1）如果操作失败则抛出异常，

（2）如果操作失败，则返回特殊值（null或false，具体取决于操作），接口的常规结构如下表所示。

| **操作类型** | **抛出异常** | **返回特殊值** |
| --- | --- | --- |
| 插入 | add(e) | offer(e) |
| 移除 | remove() | poll() |
| 检查 | element() | peek() |

Queue从Collection继承的add方法插入一个元素，除非它违反了队列的容量限制，在这种情况下它会抛出IllegalStateException；offer方法与add不同之处仅在于它通过返回false来表示插入元素失败。

remove和poll方法都移除并返回队列的头部，确切地移除哪个元素是由具体的实现来决定的，仅当队列为空时，remove和poll方法的行为才有所不同，在这些情况下，remove抛出NoSuchElementException，而poll返回null。

element和peek方法返回队列头部的元素，但不移除，它们之间的差异与remove和poll的方式完全相同，如果队列为空，则element抛出NoSuchElementException，而peek返回null。

**队列一般不要插入空元素。**

## **BlockingQueue接口**

BlockingQueue位于juc中，俗称阻塞队列， 阻塞队列首先它是一个队列，继承Queue接口，是队列就会遵循先进先出（FIFO）的原则，又因为它是阻塞的，故与普通的队列有两点区别：

* 当一个线程向队列里面添加数据时，如果队列是满的，那么将阻塞该线程，暂停添加数据
* 当一个线程从队列里面取出数据时，如果队列是空的，那么将阻塞该线程，暂停取出数据

BlockingQueue相关方法：

| **操作类型** | **抛出异常** | **返回特殊值** | **一直阻塞** | **超时退出** |
| --- | --- | --- | --- | --- |
| 插入 | add(e) | offer(e) | put(e) | offer(e,timeuout,unit) |
| 移除 | remove() | poll() | take() | poll(timeout,unit) |
| 检查 | element() | peek() | 不支持 | 不支持 |

重点，再来解释一下，加深印象：

1. 3个可能会有异常的方法，add、remove、element；这3个方法不会阻塞（是说队列满或者空的情况下是否会阻塞）；队列满的情况下，add抛出异常；队列为空情况下，remove、element抛出异常
2. offer、poll、peek 也不会阻塞（是说队列满或者空的情况下是否会阻塞）；队列满的情况下，offer返回false；队列为空的情况下，pool、peek返回null

3、队列满的情况下，调用put方法会导致当前线程阻塞

4、队列为空的情况下，调用take方法会导致当前线程阻塞

5、offer(e,timeuout,unit)，超时之前，插入成功返回true，否者返回false

6、poll(timeout,unit)，超时之前，获取到头部元素并将其移除，返回true，否者返回false

以上一些方法希望大家都记住，方便以后使用。

## **BlockingQueue常见的实现类**

**ArrayBlockingQueue**

基于数组的阻塞队列实现，其内部维护一个定长的数组，用于存储队列元素。线程阻塞的实现是通过ReentrantLock来完成的，数据的插入与取出共用同一个锁，因此ArrayBlockingQueue并不能实现生产、消费同时进行。而且在创建ArrayBlockingQueue时，我们还可以控制对象的内部锁是否采用公平锁，默认采用非公平锁。

**LinkedBlockingQueue**

基于单向链表的阻塞队列实现，在初始化LinkedBlockingQueue的时候可以指定大小，也可以不指定，默认类似一个无限大小的容量（Integer.MAX\_VALUE），不指队列容量大小也是会有风险的，一旦数据生产速度大于消费速度，系统内存将有可能被消耗殆尽，因此要谨慎操作。另外LinkedBlockingQueue中用于阻塞生产者、消费者的锁是两个（锁分离），因此生产与消费是可以同时进行的。

**PriorityBlockingQueue**

一个支持优先级排序的无界阻塞队列，进入队列的元素会按照优先级进行排序

**SynchronousQueue**

同步阻塞队列，SynchronousQueue没有容量，与其他BlockingQueue不同，SynchronousQueue是一个不存储元素的BlockingQueue，每一个put操作必须要等待一个take操作，否则不能继续添加元素，反之亦然。

**DelayQueue**

DelayQueue是一个支持延时获取元素的无界阻塞队列，里面的元素全部都是“可延期”的元素，列头的元素是最先“到期”的元素，如果队列里面没有元素到期，是不能从列头获取元素的，哪怕有元素也不行，也就是说只有在延迟期到时才能够从队列中取元素

**LinkedTransferQueue**

LinkedTransferQueue是基于链表的FIFO无界阻塞队列，它出现在JDK7中，Doug Lea 大神说LinkedTransferQueue是一个聪明的队列，它是ConcurrentLinkedQueue、SynchronousQueue(公平模式下)、无界的LinkedBlockingQueues等的超集，LinkedTransferQueue包含了ConcurrentLinkedQueue、SynchronousQueue、LinkedBlockingQueues三种队列的功能

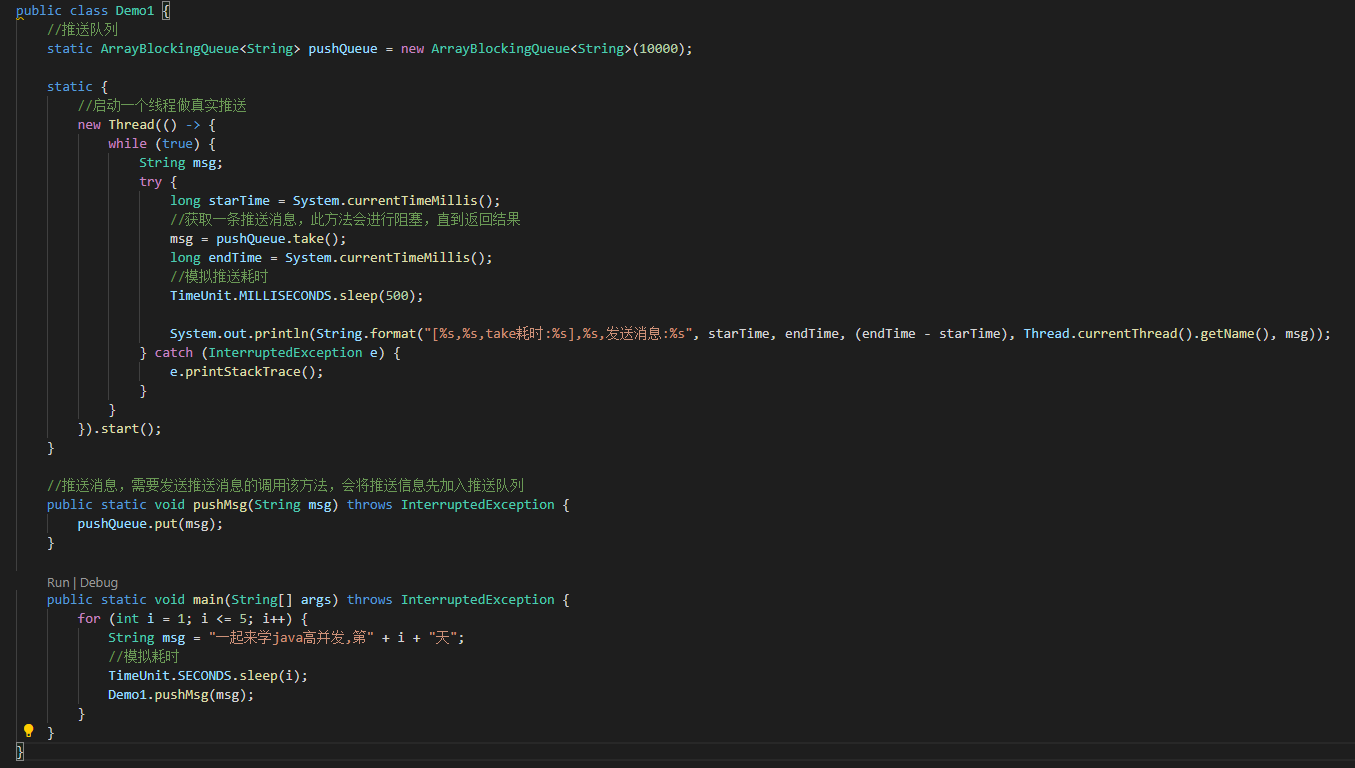
下面我们来介绍每种阻塞队列的使用。

## **ArrayBlockingQueue**

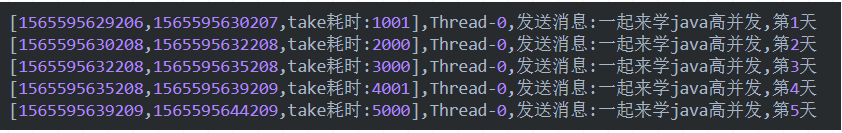
有界阻塞队列，内部使用数组存储元素，有2个常用构造方法：

//capacity表示容量大小，默认内部采用非公平锁  
public ArrayBlockingQueue(int capacity)  
//capacity：容量大小，fair：内部是否是使用公平锁  
public ArrayBlockingQueue(int capacity, boolean fair)

需求：业务系统中有很多地方需要推送通知，由于需要推送的数据太多，我们将需要推送的信息先丢到阻塞队列中，然后开一个线程进行处理真实发送，代码如下：



输出：



代码中我们使用了有界队列ArrayBlockingQueue，创建ArrayBlockingQueue时候需要制定容量大小，调用pushQueue.put将推送信息放入队列中，如果队列已满，此方法会阻塞。代码中在静态块中启动了一个线程，调用pushQueue.take();从队列中获取待推送的信息进行推送处理。

**注意：**ArrayBlockingQueue如果队列容量设置的太小，消费者发送的太快，消费者消费的太慢的情况下，会导致队列空间满，调用put方法会导致发送者线程阻塞，所以注意设置合理的大小，协调好消费者的速度。

## **LinkedBlockingQueue**

内部使用单向链表实现的阻塞队列，3个构造方法：

//默认构造方法，容量大小为Integer.MAX\_VALUE  
public LinkedBlockingQueue();  
//创建指定容量大小的LinkedBlockingQueue  
public LinkedBlockingQueue(int capacity);  
//容量为Integer.MAX\_VALUE,并将传入的集合丢入队列中  
public LinkedBlockingQueue(Collection<? extends E> c);

LinkedBlockingQueue的用法和ArrayBlockingQueue类似，建议使用的时候指定容量，如果不指定容量，插入的太快，移除的太慢，可能会产生OOM。

## **PriorityBlockingQueue**

**无界的优先级**阻塞队列，内部使用数组存储数据，达到容量时，会自动进行扩容，放入的元素会按照优先级进行排序，4个构造方法：

//默认构造方法，默认初始化容量是11  
public PriorityBlockingQueue();  
//指定队列的初始化容量  
public PriorityBlockingQueue(int initialCapacity);  
//指定队列的初始化容量和放入元素的比较器  
public PriorityBlockingQueue(int initialCapacity,Comparator<? super E> comparator);  
//传入集合放入来初始化队列，传入的集合可以实现SortedSet接口或者PriorityQueue接口进行排序，如果没有实现这2个接口，按正常顺序放入队列  
public PriorityBlockingQueue(Collection<? extends E> c);

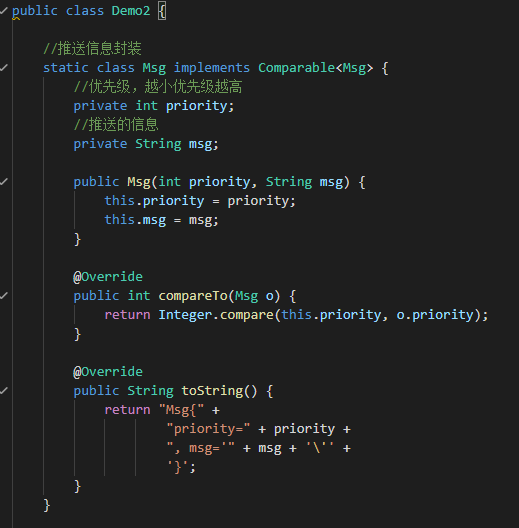
优先级队列放入元素的时候，会进行排序，所以我们需要指定排序规则，有2种方式：

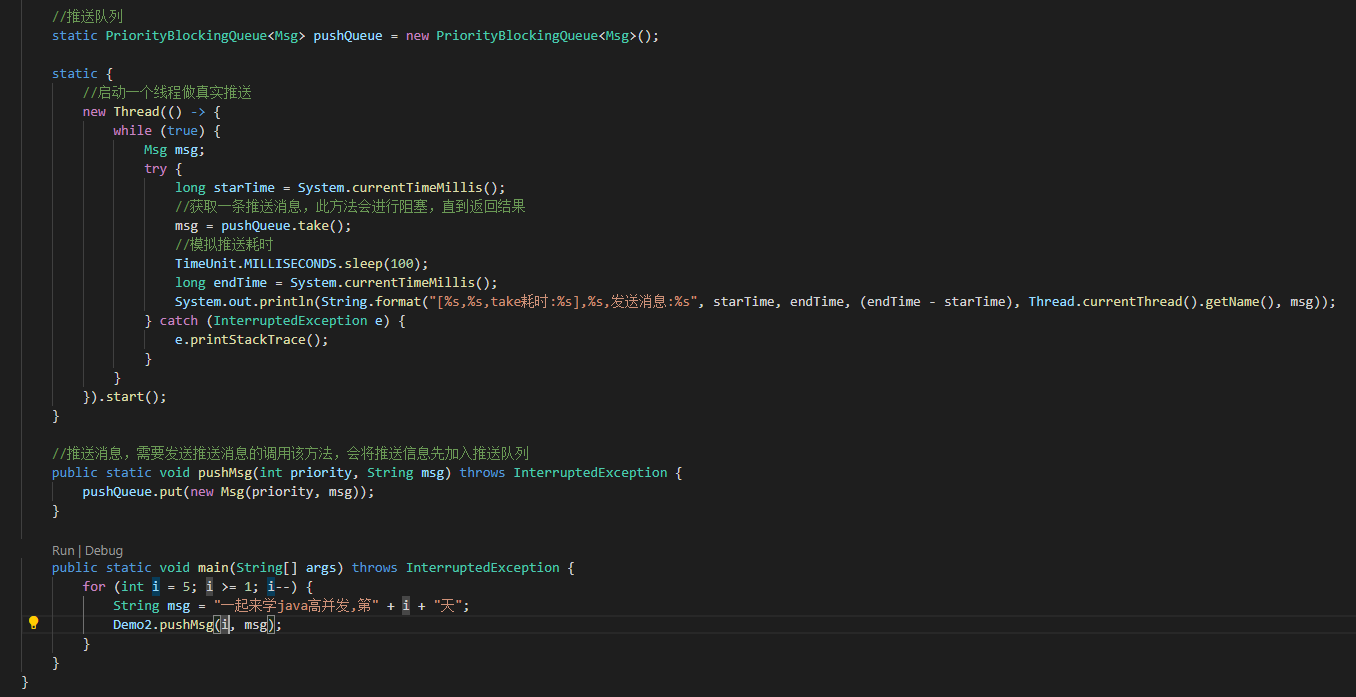
创建PriorityBlockingQueue指定比较器Comparator

放入的元素需要实现Comparable接口

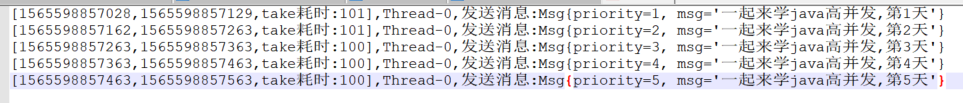
上面2种方式必须选一个，如果2个都有，则走第一个规则排序。

需求：还是上面的推送业务，目前推送是按照放入的先后顺序进行发送的，比如有些公告比较紧急，优先级比较高，需要快点发送，怎么搞？此时PriorityBlockingQueue就派上用场了，代码如下：





输出：



main中放入了5条推送信息，i作为消息的优先级按倒叙放入的，最终输出结果中按照优先级由小到大输出。注意Msg实现了Comparable接口，具有了比较功能。

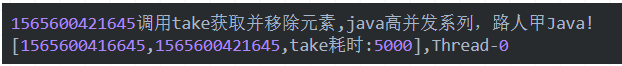
## SynchronousQueue

同步阻塞队列，SynchronousQueue没有容量，与其他BlockingQueue不同，SynchronousQueue是一个不存储元素的BlockingQueue，每一个put操作必须要等待一个take操作，否则不能继续添加元素，反之亦然。SynchronousQueue 在现实中用的不多，线程池中有用到过，Executors.newCachedThreadPool()实现中用到了这个队列，当有任务丢入线程池的时候，如果已创建的工作线程都在忙于处理任务，则会新建一个线程来处理丢入队列的任务。

来个示例代码：



输出：



main方法中启动了一个线程，调用queue.put方法向队列中丢入一条数据，调用的时候产生了阻塞，从输出结果中可以看出，直到take方法被调用时，put方法才从阻塞状态恢复正常。

## **DelayQueue**

DelayQueue是一个支持延时获取元素的无界阻塞队列，里面的元素全部都是“可延期”的元素，列头的元素是最先“到期”的元素，如果队列里面没有元素到期，是不能从列头获取元素的，哪怕有元素也不行，也就是说只有在延迟期到时才能够从队列中取元素。

**需求：**还是推送的业务，有时候我们希望早上9点或者其他指定的时间进行推送，如何实现呢？此时DelayQueue就派上用场了。

我们先看一下DelayQueue类的声明：

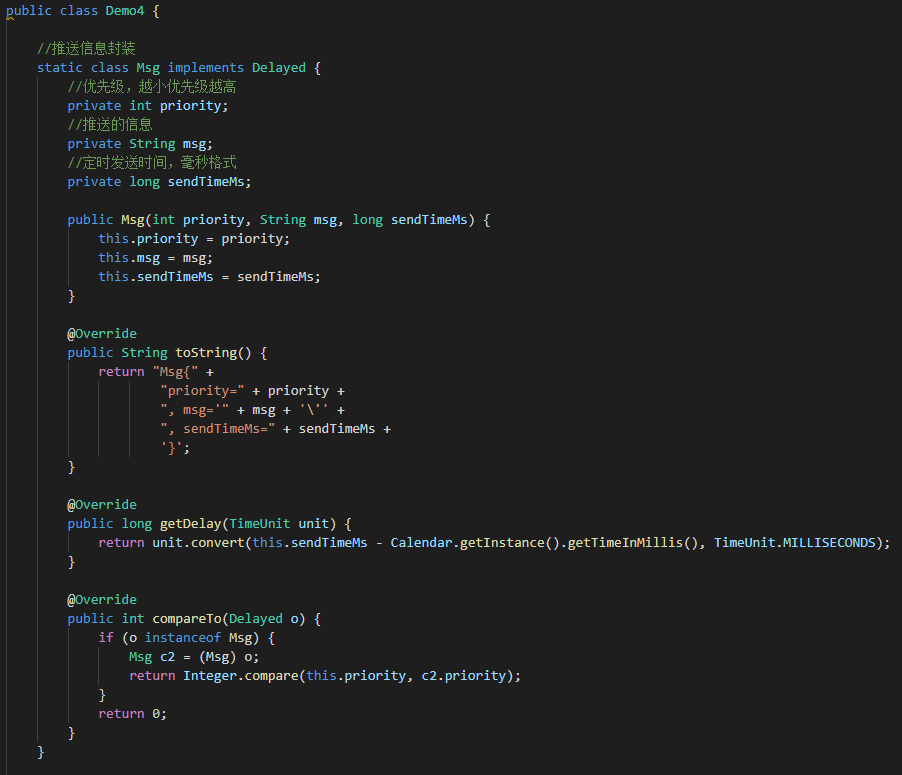
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>  
    implements BlockingQueue<E>

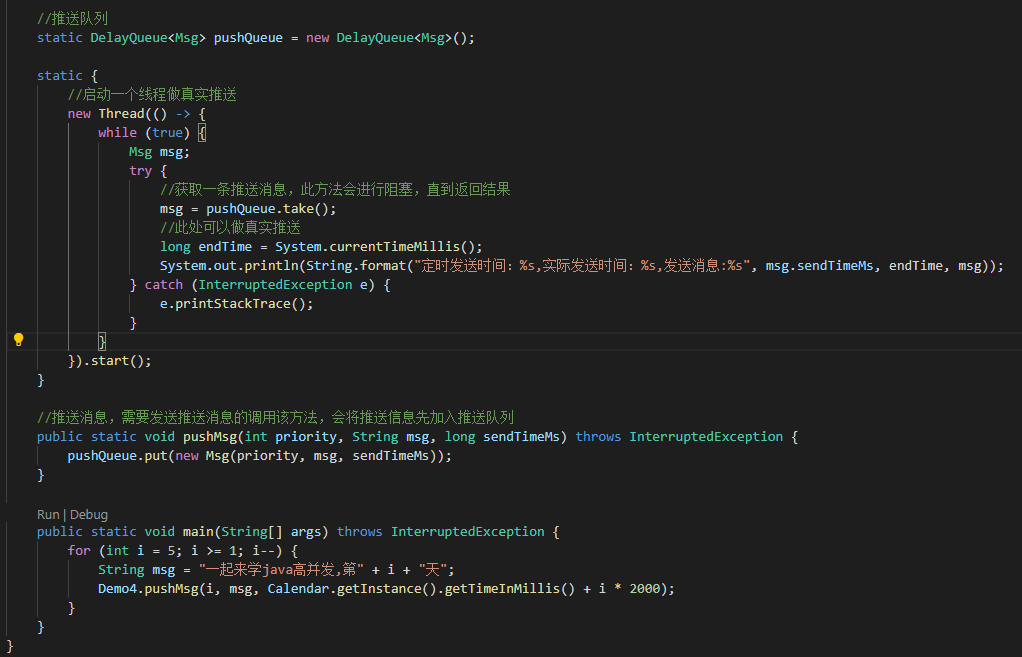
元素E需要实现接口Delayed，我们看一下这个接口的代码：

public interface Delayed extends Comparable<Delayed> {  
    long getDelay(TimeUnit unit);  
}

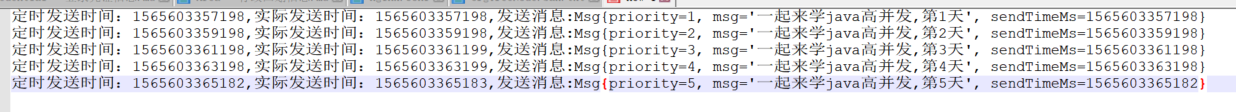
Delayed继承了Comparable接口，这个接口是用来做比较用的，DelayQueue内部使用PriorityQueue来存储数据的，PriorityQueue是一个优先级队列，丢入的数据会进行排序，排序方法调用的是Comparable接口中的方法。下面主要说一下Delayed接口中的getDelay方法：此方法在给定的时间单位内返回与此对象关联的剩余延迟时间。

**对推送我们再做一下处理，让其支持定时发送（定时在将来某个时间也可以说是延迟发送），代码如下：**





输出：

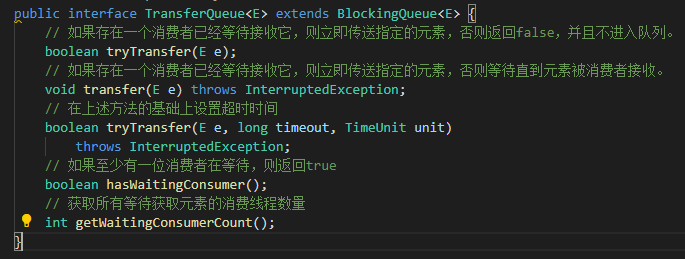


可以看出时间发送时间，和定时发送时间基本一致，代码中Msg需要实现Delayed接口，重点在于getDelay方法，这个方法返回剩余的延迟时间，代码中使用this.sendTimeMs减去当前时间的毫秒格式时间，得到剩余延迟时间。

## **LinkedTransferQueue**

LinkedTransferQueue是一个由链表结构组成的无界阻塞TransferQueue队列。相对于其他阻塞队列，LinkedTransferQueue多了tryTransfer和transfer方法。

LinkedTransferQueue类继承自AbstractQueue抽象类，并且实现了TransferQueue接口：



再看一下上面的这些方法，transfer(E e)方法和SynchronousQueue的put方法类似，都需要等待消费者取走元素，否者一直等待。其他方法和ArrayBlockingQueue、LinkedBlockingQueue中的方法类似。

## **总结**

* 重点需要了解BlockingQueue中的所有方法，以及他们的区别
* 重点掌握ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue、DelayQueue的使用场景
* 需要处理的任务有优先级的，使用PriorityBlockingQueue
* 处理的任务需要延时处理的，使用DelayQueue