阻塞队列

阻塞队列支持阻塞的插入和移除

支持阻塞的插入方法：当队列满时，队列会阻塞插入元素的线程，直到队列不满

支持阻塞的移除方法：当队列为空时，获取元素的线程会等待队列非空

阻塞队列就是生产者用来存放元素、消费者用来获取元素的容器

1）java中的阻塞队列（jdk7）：

ArrayBlockingQueue:一个由数组结构组成的有界阻塞队列，创建队列对象时必须制定容量大小，并且可以指定公平与非公平性，默认情况下非公平（公平性是指阻塞的线程可以按照阻塞的先后顺序访问队列）

LinkedBlockingQueue:一个由链表结构组成的有界阻塞队列，创建时不指定容量大小，默认大小为Integer.MAX_VALUE

PriorityBlockingQueue:一个支持优先级排序的无界阻塞队列.（容量没有上限）

DelayQueue:基于PriorityQueue，一种延时阻塞队列，只有当其指定的延迟时间到了，才能够从队列中获取到元素，无界队列（往队列插入的操作永远不会阻塞）

2）阻塞队列中的方法

方法/处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove()	poll()	take()	poll(time,unit)
检查方法	element	peek()	不可用	不可用

抛出异常：当队列满时再插入元素，会抛出IllegalStateException异常，当队列空时，从队列中获取元素会抛出NoSuchException异常

返回特殊值：当往队列插入元素时会返回插入是否成功，成功返回true;如果是移除方法，从队列里取出一个元素，没有则返回null

一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程直到队列可用或者响应中断退出。当队列空时，如果消费者线程从队列里take元素，队列会阻塞消费者线程直到队列不为空

超时退出：当队列满时，如果往队列里插入元素，则阻塞线程一段时间，如果超过了指定时间，生产者线程就会退出

3）阻塞队列的实现原理

ArrayBlockingQueue:

1.使用通知模式实现

put方法实现：

```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    final E[] items = this.items;
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        try {
            while (count == items.length)
                notFull.await();
        } catch (InterruptedException ie) {
            notFull.signal(); // propagate to non-interrupted thread
            throw ie;
        }
        insert(e);
    } finally {
        lock.unlock();
    }
}
```

首先获取锁，获取的为可中断锁，然后判读队列是否满，满的话调用notFull.await()进行等待

当被其他线程唤醒时，通过insert(e)方法插入元素最后解锁

```
1  private void insert(E x) {
2      items[putIndex] = x;
3      putIndex = inc(putIndex);
4      ++count;
5      notEmpty.signal();
6  }
```

take方法实现：

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        try {
            while (count == 0)
                notEmpty.await();
        } catch (InterruptedException ie) {
            notEmpty.signal(); // propagate to non-interrupted thread
            throw ie;
        }
        E x = extract();
        return x;
    } finally {
        lock.unlock();
    }
}
```

take方法应用的是notEmpty信号

```
private E extract() {
    final E[] items = this.items;
    E x = items[takeIndex];
    items[takeIndex] = null;
    takeIndex = inc(takeIndex);
    --count;
    notFull.signal();
    return x;
}
```

4）阻塞队列的应用场景

a.生产者消费者场景

使用wait和notify

```
public class Test {
    private int queueSize = 10;
    private PriorityQueue<Integer> queue = new PriorityQueue<Integer>
(queueSize);

    public static void main(String[] args)  {
        Test test = new Test();
        Producer producer = test.new Producer();
        Consumer consumer = test.new Consumer();

        producer.start();
        consumer.start();
    }

    class Consumer extends Thread{
```

```
        @Override
        public void run() {
            consume();
        }

        private void consume() {
            while(true){
                synchronized (queue) {
                    while(queue.size() == 0){
                        try {
                            System.out.println("队列空，等待数据");
                            queue.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                            queue.notify();
                        }
                    }
                    queue.poll();           //每次移走队首元素
                    queue.notify();
                    System.out.println("从队列取走一个元素，队列剩
余"+queue.size()+"个元素");
                }
            }
        }
    }

    class Producer extends Thread{

        @Override
        public void run() {
            produce();
        }

        private void produce() {
            while(true){
                synchronized (queue) {
                    while(queue.size() == queueSize){
                        try {
                            System.out.println("队列满，等待有空余空间");
                            queue.wait();
```

```
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                            queue.notify();
                        }
                    }
                    queue.offer(1);          //每次插入一个元素
                    queue.notify();
                    System.out.println("向队列取中插入一个元素，队列剩余空间："+
(queueSize-queue.size()));
                }
            }
        }
    }
}
```

使用阻塞队列

```
public class Test {
    private int queueSize = 10;
    private ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>
(queueSize);

    public static void main(String[] args)  {
        Test test = new Test();
        Producer producer = test.new Producer();
        Consumer consumer = test.new Consumer();

        producer.start();
        consumer.start();
    }

    class Consumer extends Thread{

        @Override
        public void run() {
            consume();
        }

        private void consume() {
            while(true){
                try {
                    queue.take();
```

```
                    System.out.println("从队列取走一个元素，队列剩
余"+queue.size()+"个元素");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    class Producer extends Thread{

        @Override
        public void run() {
            produce();
        }

        private void produce() {
            while(true){
                try {
                    queue.put(1);
                    System.out.println("向队列取中插入一个元素，队列剩余空间："+
(queueSize-queue.size()));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

b.socket客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。

c.DelayQueue应用场景：

缓存系统的设计:可以用DelayQueue保存缓存元素的有效期，使用一个线程循环查询DelayQueue,一旦能从DelayQueue中获取元素时，表示缓存有效期到了

定时任务调度：使用DelayQueue保存当天将会执行的任务和执行时间，一旦从DelayQueue中获取到任务就开始执行