请大家阅读文档时，在视图里勾选导航窗格，在左边显示章节目录方便浏览。

## 一、编程题1

下面程序是一个泛型容器Container<T>的定义，该容器是对ArrayList的一个封装，实现了四个公有的方法add、remove、size、get。

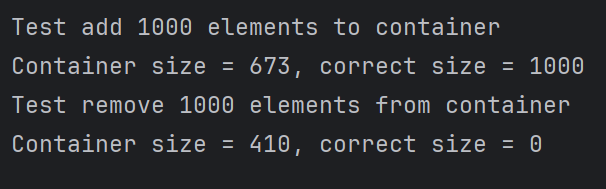
class Container<T>{  
 private List<T> elements = new ArrayList<>();  
  
 */\*\*  
 \* 添加元素  
 \** ***@param*** *e 要添加的元素  
 \*/* public void add(T e){  
 elements.add(e);  
 }  
  
 */\*\*  
 \* 删除指定下标的元素  
 \** ***@param*** *index 指定元素下标  
 \** ***@return*** *被删除的元素  
 \*/* public T remove(int index){  
 return elements.remove(index);  
 }  
  
 */\*\*  
 \* 获取容器里元素的个数  
 \** ***@return*** *元素个数  
 \*/* public int size(){  
 return elements.size();  
 }  
  
 */\*\*  
 \* 获取指定下标的元素  
 \** ***@param*** *index 指定下标  
 \** ***@return*** *指定下标的元素  
 \*/* public T get(int index){  
 return elements.get(index);  
 }  
}

现在用如下程序对泛型容器进行测试。

public class Test {  
 public static void testAdd(){  
 Container<Integer> container = new Container<>();  
 int addLoops = 10; //addTask内的循环次数  
 Runnable addTask = new Runnable() {  
 @Override  
 public void run() {  
 for(int i = 0; i < addLoops; i++){  
 container.add(i);  
 }  
 }  
 };  
  
 int addTaskCount = 100; //addTask线程个数  
 ExecutorService es = Executors.*newCachedThreadPool*();  
 for(int i = 0; i < addTaskCount; i++){  
 es.execute(addTask);  
 }  
  
 es.shutdown();  
 while (!es.isTerminated()){}  
 System.*out*.println("Test add " + (addLoops \* addTaskCount) +   
 " elements to container");  
 System.*out*.println("Container size = " + container.size() +   
 ", correct size = " + (addLoops \* addTaskCount));  
  
 }  
  
 public static void testRemove(){  
 Container<Integer> container = new Container<>();  
 int removeLoops = 10; //removeTask内的循环次数  
 int removeTaskCount = 100; //removeTask线程个数  
  
 //首先添加removeLoops \* removeTask个元素到容器  
 for(int i = 0; i < removeLoops \* removeTaskCount; i++){  
 container.add(i);  
 }  
  
 Runnable removeTask = new Runnable() {  
 @Override  
 public void run() {  
 for(int i = 0 ; i < removeLoops; i++){  
 container.remove(0);  
 }  
 }  
 };  
  
 ExecutorService es = Executors.*newCachedThreadPool*();  
 for(int i = 0; i < removeTaskCount; i++){  
 es.execute(removeTask);  
 }  
  
 es.shutdown();  
 while (!es.isTerminated()){}  
  
 System.*out*.println("Test remove " + (removeLoops \* removeTaskCount) +   
 " elements from container");  
 System.*out*.println("Container size = " + container.size() +   
 ", correct size = 0");  
 }  
  
  
 public static void main(String[] args){  
 *testAdd*();  
 *testRemove*();  
 }  
}

1）请运行上面的测试程序来测试泛型Container是否是线程安全的，请分析ArrayList的add方法与remove方法的源代码，简单分析导致泛型Container不是线程安全的原因；

**线程不安全。**某次运行结果如图



查看ArrayList的add方法源码如下

    private void add(E e, Object[] elementData, int s) {

        if (s == elementData.length)

            elementData = grow();

        elementData[s] = e;

        size = s + 1;

    }

当多个线程并发执行 add() 操作，它们可能同时修改 size 和 elementData 数组，导致线程安全问题，remove方法同理

2）请实现泛型Container<T>的一个线程安全的版本SynchronizedContainer<T>，只需要实现与Container<T>一样的4个公有方法。要求必须用synchronized同步语句块或Lock锁实现、

在每个方法返回值void前添加synchronized

或者在每个方法中使用lock锁，比如

private final Lock lock = new ReentrantLock();

   public void add(T e) {

        lock.lock();

        try {

            elements.add(e);

        } finally {

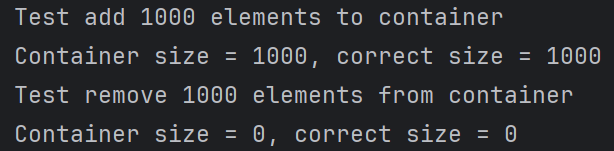
            lock.unlock();

        }

    }

具体代码**见附录**或工程文件的homework.ch30.p1的SynchronizedContainer类

3）用上面同样的测试代码，来测试SynchronizedContainer<T>的线程安全性；

测试结果如图，结果正确

## 二、编程题2

实现一个线程安全的同步队列SyncQueue<T>，模拟多线程环境下的生产者消费者机制，SyncQueue<T>的定义如下：

*/\*\*  
 \* 一个线程安全同步队列，模拟多线程环境下的生产者消费者机制  
 \* 一个生产者线程通过produce方法向队列里产生元素  
 \* 一个消费者线程通过consume方法从队列里消费元素  
 \** ***@param*** <*T*> *元素类型  
 \*/*public class SyncQueue<T> {  
 */\*\*  
 \* 保存队列元素  
 \*/* private ArrayList<T> list = new ArrayList<>();

//*TODO 这里加入需要的数据成员*

*/\*\*  
 \* 生产数据  
 \** ***@param*** *elements 生产出的元素列表，需要将该列表元素放入队列  
 \** ***@throws*** *InterruptedException  
 \*/* public void produce(List<T> elements) {

//*TODO 这里需要实现代码*

}  
  
 */\*\*  
 \* 消费数据  
 \** ***@return*** *从队列中取出的数据  
 \** ***@throws*** *InterruptedException  
 \*/* public List<T> consume(){

//*TODO 这里需要实现代码*  
 }  
}

SyncQueue<T>的测试代码为：创建一个生产者线程不断地向队列生产数据，创建一个消费者线程不断地从队列消费数据，**要求生产出来的数据和消费的数据次序完全一样**。测试代码如下所示：

public class TestSyncQueue {  
 public static void main(String[] args){  
 SyncQueue<Integer> syncQueue = new SyncQueue<>();  
 Runnable produceTask = ()->{  
 while(true){  
 try {  
 List<Integer> list = new ArrayList<>();  
 int elementsCount = (int)(Math.*random*() \* 10) + 1;  
 for(int i = 0; i < elementsCount; i++){  
 int r = (int)(Math.*random*() \* 10) + 1;  
 list.add(r);  
 }  
 syncQueue.produce(list);  
 Thread.*sleep*((int)(Math.*random*() \* 5) + 1);  
 }  
 catch (InterruptedException e) { e.printStackTrace(); }  
 }  
 };  
  
 Runnable consumeTask = ()->{  
 while (true){  
 try{  
 List<Integer> list = syncQueue.consume();  
 Thread.*sleep*((int)(Math.*random*() \* 10) + 1);  
 }  
 catch (InterruptedException e) { e.printStackTrace(); }  
 }  
 };  
  
 ExecutorService es = Executors.*newFixedThreadPool*(2);  
 es.execute(produceTask);  
 es.execute(consumeTask);  
 es.shutdown();  
 while (!es.isTerminated()){}  
 }  
}

要求实现基于二个版本的SyncQueue<T>，第1个版本为类名为SyncQueue1<T>，第1个版本为类名为SyncQueue2<T>，其功能要求分别为：

1）SyncQueue1<T>实现生产者线程和消费者线程**轮流**生产数据和消费数据，即如果队列不为空，则生产者线程必须等到消费者线程将队列里的数据消费完后才能向队列生产数据；如果队列为空，则消费者线程必须等待生产者线程向队列生产数据后才能消费数据。这个版本的测试结果应该如下所示：

Produce elements: 4 5 3 1 9 6 10 6 7 6

Consume elements: 4 5 3 1 9 6 10 6 7 6

Produce elements: 2 5 9 9 7 10

Consume elements: 2 5 9 9 7 10

Produce elements: 8 8 9 2

Consume elements: 8 8 9 2

2）SyncQueue2<T>要求有些区别：即**生产者线程不管队列是否为空，随时可以向队列生产数据**；消费者线程则在队列为空时，必须等待生产者线程向队列生产数据后才能消费数据。这个版本的测试结果应该如下所示：

Produce elements: 7 9 2 7 6 8 9 1 7 3

Produce elements: 7 5 9 8 8

Consume elements: 7 9 2 7 6 8 9 1 7 3 7 5 9 8 8

Produce elements: 6 2 8

Produce elements: 3 2 5 10 3 4 5 1 6

Produce elements: 5 7

Consume elements: 6 2 8 3 2 5 10 3 4 5 1 6 5 7

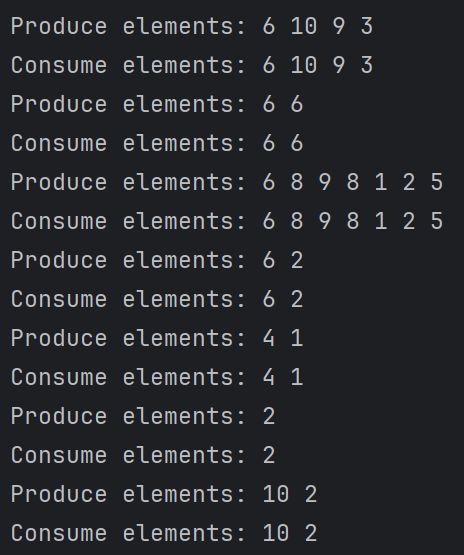
Produce elements: 3

Consume elements: 3

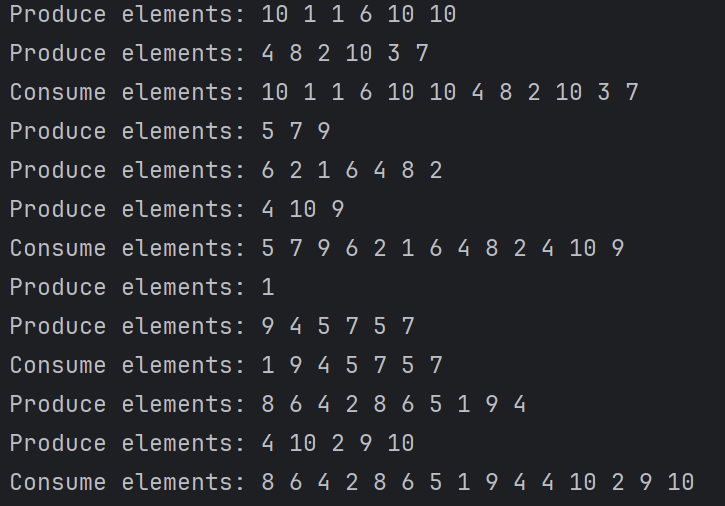
Produce elements: 10 3 4

Consume elements: 10 3 4

两者的运行结果如图，经检查符合题意，具体代码**见附录**或工程文件



**版本1的运行结果**



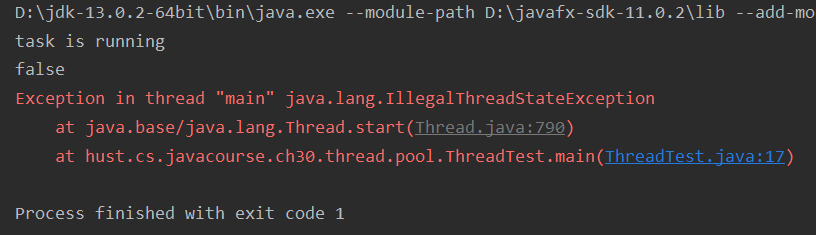
**版本2的运行结果**

## 三、编程题3

我们知道JDK提供了线程池的支持，线程池可以通过重复利用已创建的线程降低线程创建和销毁造成的消耗。但是Thread一旦启动，执行完线程任务后，就不可再次启动。请看下面示例代码：

public class ThreadTest {  
 public static void main(String[] args){  
 Runnable task = ()->{ System.*out*.println("task is running"); };  
 Thread t = new Thread(task);  
 t.start();  
 try { Thread.*sleep*(1000); }  
 catch (InterruptedException e) { e.printStackTrace(); }  
 System.*out*.println(t.isAlive()); //当线程任务结束后，isAlive返回false  
 t.start(); //一旦线程结束，不可再start，否则抛出异常  
 }  
}

上面运行结果如下：



那么JDK线程池是如何做到线程复用的？这是一个值得思考的问题。这里的线程复用指的是一个Thread线程对象一旦启动，可以运行多个线程任务。Thread类的构造函数可以传入Runnable对象，然后在start方法里启动子线程后，会调用传入的Runnable对象的run方法。

Thread类在内部通过实例变量target保存构造函数传入的Runnable方法：

/\* What will be run. \*/  
private Runnable target;

同时Thread类实现了Runnable接口，它的run方法实现为：

@Override  
public void run() {  
 if (target != null) {  
 target.run();  
 }  
}

可以看到，当子线程启动以后，会自动调用Thread的run方法，run方法里检查任务对象target是否为空，如果不为空则执行target.run。因此当target.run方法结束以后，线程执行结束，这个时候是无法复用线程对象的。

因此，要想复用线程对象，必须要让线程的run方法永远不结束，也就是一个while(true)循环，在循环里等待新的线程任务到来，大致的逻辑应该是这样的：

while(true){

等待新任务

执行新任务

}

基于以上思路，请实现一个可复用的线程类ReusableThread，类的定义为：

public class ReusableThread extends Thread{  
 private Runnable runTask = null; //保存接受的线程任务  
 //TODO 加入需要的数据成员  
  
 //只定义不带参数的构造函数  
 public ReusableThread(){  
 super();  
 }  
  
 */\*\*  
 \* 覆盖Thread类的run方法  
 \*/* @Override  
 public void run() {  
 //这里必须是永远不结束的循环  
 }  
  
 */\*\*  
 \* 提交新的任务  
 \** ***@param*** *task 要提交的任务  
 \*/* public void submit(Runnable task){  
  
 }  
}

提示：可以使用条件对象的await/singalAll机制：在run方法里，如果没有线程任务就等待；一旦有线程任务通过submit提交，就唤醒线程执行提交的任务。

实现好ReusableThread类，可用下面的测试代码进行测试：

public static void test3(){  
 Runnable task1 = new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("Thread " + Thread.*currentThread*().getId() +

": is running " + toString());  
 try { Thread.*sleep*(200); }  
 catch (InterruptedException e) { e.printStackTrace(); }  
 }  
 @Override  
 public String toString() {  
 return "task1";  
 }  
 };  
  
 Runnable task2 = new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("Thread " + Thread.*currentThread*().getId() +

" is running " + toString());  
 try { Thread.*sleep*(100); }  
 catch (InterruptedException e) { e.printStackTrace(); }  
 }  
 @Override  
 public String toString() {  
 return "task2";  
 }  
 };  
  
 ReusableThread t =new ReusableThread();  
 t.start(); //主线程启动子线程  
 for(int i = 0; i < 5; i++){  
 t.submit(task1);   
 t.submit(task2);  
 }  
}

上述测试代码的执行结果为：

Thread 13: is running task1

Thread 13 is running task2

Thread 13: is running task1

Thread 13 is running task2

Thread 13: is running task1

Thread 13 is running task2

Thread 13: is running task1

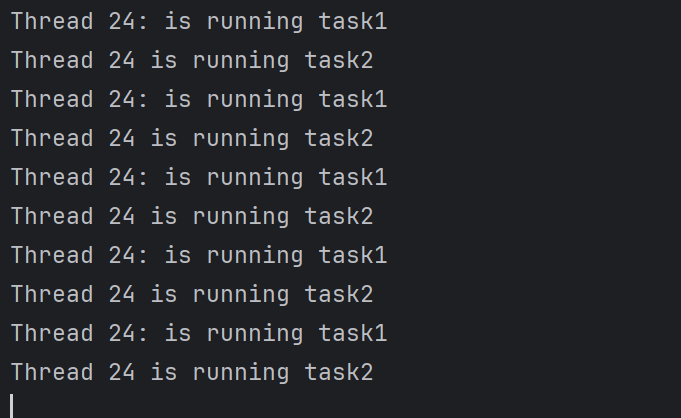
Thread 13 is running task2

Thread 13: is running task1

Thread 13 is running task2

注意由于ReusableThread线程对象会等待新的任务，因此上面的程序永远不会结束。通过这个编程题大家就清楚了在Java里如何复用一个线程对象。这个原理大家清楚后，以后若去看JDK线程池的源代码，就很好理解了。

输出如下，代码见附录或工程文件中homework.ch30.p3包

  
**编程题3的运行结果**

# 30章附录

## 编程题1代码

SynchronizedContainer类

package homework.ch30.p1;

import java.util.ArrayList;

import java.util.List;

public class SynchronizedContainer<T> {

    private List<T> elements = new ArrayList<>();

*/\*\**

     \* 添加元素

     \* @param e 要添加的元素

     \*/

    public synchronized void add(T e) {

        elements.add(e);

    }

*/\*\**

     \* 删除指定下标的元素

     \* @param index 指定元素下标

     \* @return 被删除的元素

     \*/

    public synchronized T remove(int index) {

        return elements.remove(index);

    }

*/\*\**

     \* 获取容器里元素的个数

     \* @return 元素个数

     \*/

    public synchronized int size() {

        return elements.size();

    }

*/\*\**

     \* 获取指定下标的元素

     \* @param index 指定下标

     \* @return 指定下标的元素

     \*/

    public synchronized T get(int index) {

        return elements.get(index);

    }

}

## 编程题2代码

### SyncQueue1类

package homework.ch30.p2;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

*/\*\**

 \* SyncQueue1<T> 是一个线程安全的队列实现，支持生产者和消费者模式。

 \* 与普通队列不同，生产者和消费者线程必须轮流操作：

 \* - 如果队列不为空（即已经生产数据），生产者线程必须等待消费者线程消费完数据后才能生产；

 \* - 如果队列为空（即未生产数据），消费者线程必须等待生产者线程生产数据后才能消费。

 \* @param <T> 队列中存储的数据类型

 \*/

public class SyncQueue1<T> {

*// 使用 ArrayList 存储队列中的数据*

    private final List<T> list = new ArrayList<>();

*// 使用 ReentrantLock 实现线程同步*

    private final Lock lock = new ReentrantLock();

*// Condition 用于实现线程间的等待/通知机制*

    private final Condition notEmpty = lock.newCondition(); *// 队列非空时可消费*

    private final Condition notFull = lock.newCondition();  *// 队列为空时可生产*

*// 标记是否已经生产数据*

    private boolean isProduced = false;

*/\*\**

     \* 生产者方法，向队列中添加一组数据。

     \* 如果队列中已有数据（isProduced 为 true），生产者线程会等待消费者线程消费完数据。

     \*

     \* @param elements 要添加到队列中的数据列表

     \* @throws InterruptedException 如果线程在等待锁时被中断

     \*/

    public void produce(List<T> elements) throws InterruptedException {

        lock.lock(); *// 获取锁*

        try {

*// 如果已经生产数据，等待消费者消费*

            while (isProduced) {

                notFull.await(); *// 释放锁并等待，直到被唤醒*

            }

            list.addAll(elements); *// 将数据添加到队列*

            System.out.print("Produce elements: ");

            for (T e : elements) System.out.print(e + " "); *// 打印生产的数据*

            System.out.println();

            isProduced = true; *// 标记为已生产*

            notEmpty.signal(); *// 通知等待的消费者线程队列非空*

        } finally {

            lock.unlock(); *// 释放锁*

        }

    }

*/\*\**

     \* 消费者方法，从队列中消费数据。

     \* 如果队列为空（isProduced 为 false），消费者线程会等待生产者线程生产数据。

     \*

     \* @return 从队列中消费的数据列表

     \* @throws InterruptedException 如果线程在等待时被中断

     \*/

    public List<T> consume() throws InterruptedException {

        lock.lock(); *// 获取锁*

        try {

*// 如果队列为空，等待生产者生产数据*

            while (!isProduced) {

                notEmpty.await(); *// 释放锁并等待，直到被唤醒*

            }

*// 复制队列中的数据到结果列表*

            List<T> result = new ArrayList<>(list);

            list.clear(); *// 清空队列*

            System.out.print("Consume elements: ");

            for (T e : result) System.out.print(e + " "); *// 打印消费的数据*

            System.out.println();

            isProduced = false; *// 标记为未生产*

            notFull.signal(); *// 通知等待的生产者线程队列为空*

            return result; *// 返回消费的数据*

        } finally {

            lock.unlock(); *// 释放锁*

        }

    }

}

### SyncQueue2类

package homework.ch30.p2;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

*/\*\**

 \* SyncQueue2<T> 是一个线程安全的队列实现，支持生产者和消费者模式。

 \* 与普通队列不同，生产者线程可以随时向队列中添加数据（即使队列为空），

 \* 而消费者线程在队列为空时必须等待，直到生产者线程添加数据后才能消费。

 \*

 \* @param <T> 队列中存储的数据类型

 \*/

public class SyncQueue2<T> {

*// 使用 ArrayList 存储队列中的数据*

    private final List<T> list = new ArrayList<>();

*// 使用 ReentrantLock 实现线程同步*

    private final Lock lock = new ReentrantLock();

*// Condition 用于实现线程间的等待/通知机制*

    private final Condition notEmpty = lock.newCondition(); *// 队列非空时可消费*

*/\*\**

     \* 生产者方法，向队列中添加一组数据。

     \* 生产者线程可以随时调用此方法，不受队列是否为空的影响。

     \*

     \* @param elements 要添加到队列中的数据列表

     \* @throws InterruptedException 如果线程在等待锁时被中断

     \*/

    public void produce(List<T> elements) throws InterruptedException {

        lock.lock(); *// 获取锁*

        try {

            list.addAll(elements); *// 将数据添加到队列*

            System.out.print("Produce elements: ");

            for (T e : elements) System.out.print(e + " "); *// 打印生产的数据*

            System.out.println();

            notEmpty.signal(); *// 通知等待的消费者线程队列非空*

        } finally {

            lock.unlock(); *// 释放锁*

        }

    }

*/\*\**

     \* 消费者方法，从队列中消费数据。

     \* 如果队列为空，消费者线程会等待，直到生产者线程添加数据并通知。

     \*

     \* @return 从队列中消费的数据列表

     \* @throws InterruptedException 如果线程在等待时被中断

     \*/

    public List<T> consume() throws InterruptedException {

        lock.lock(); *// 获取锁*

        try {

*// 如果队列为空，等待生产者通知*

            while (list.isEmpty()) {

                notEmpty.await(); *// 释放锁并等待，直到被唤醒*

            }

*// 复制队列中的数据到结果列表*

            List<T> result = new ArrayList<>(list);

            list.clear(); *// 清空队列*

            System.out.print("Consume elements: ");

            for (T e : result) System.out.print(e + " "); *// 打印消费的数据*

            System.out.println();

            return result; *// 返回消费的数据*

        } finally {

            lock.unlock(); *// 释放锁*

        }

    }

}

## 编程题3代码

package homework.ch30.p3;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

import java.util.concurrent.locks.\*;

import java.util.concurrent.locks.\*;

public class ReusableThread extends Thread {

    private Runnable runTask = null;

    private final Lock lock = new ReentrantLock();

    private final Condition notEmpty = lock.newCondition();  *// 子线程等待任务*

    private final Condition taskDone = lock.newCondition();  *// 主线程等待执行完*

    private boolean hasNewTask = false;

    public ReusableThread() {

        super();

    }

    @Override

    public void run() {

        while (true) {

            lock.lock();

            try {

*// 子线程等待有新任务*

                while (!hasNewTask) {

                    notEmpty.await();

                }

                if (runTask != null) {

                    runTask.run();

                    runTask = null;

                }

                hasNewTask = false;

                taskDone.signal(); *// 通知主线程任务已完成*

            } catch (InterruptedException e) {

                e.printStackTrace();

            } finally {

                lock.unlock();

            }

        }

    }

    public void submit(Runnable task) {

        lock.lock();

        try {

*// 等待旧任务执行完再提交新任务*

            while (hasNewTask) {

                taskDone.await();

            }

            runTask = task;

            hasNewTask = true;

            notEmpty.signal(); *// 通知子线程去执行*

        } catch (InterruptedException e) {

            e.printStackTrace();

        } finally {

            lock.unlock();

        }

    }

}