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；

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

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

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

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：

```
D:\jdk-13.0.2-64bit\bin\java.exe --module-path D:\javafx-sdk-11.0.2\lib --add-mo
task is running
false
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.base/java.lang.Thread.start(Thread.java:790)
    at hust.cs.javacourse.ch30.thread.pool.ThreadTest.main(ThreadTest.java:17)

Process finished with exit code 1
```

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。