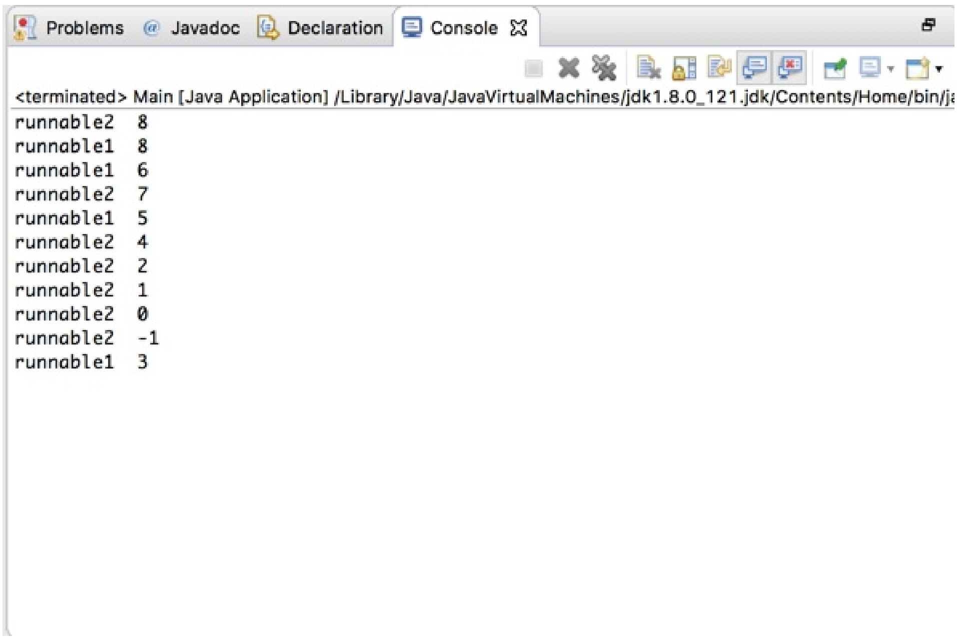
> 自从java1.5以后，官网就推出了Executor这样一个类，这个类，可以维护我们的大量线程在操作临界资源时的稳定性。

先上一段代码吧：

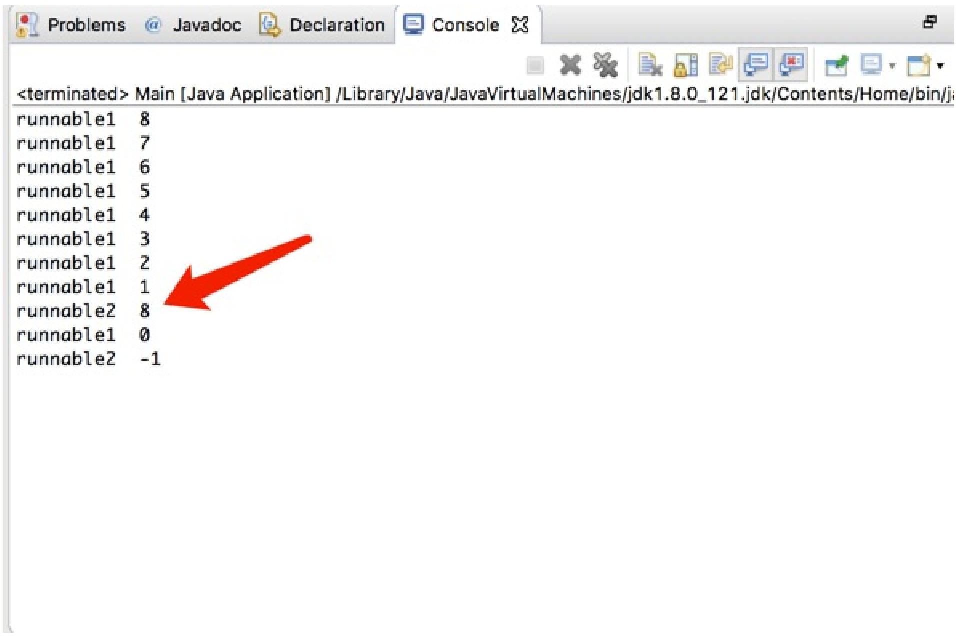
```
TestRunnable.java
public class TestRunnable implements Runnable {
    private String name;

    public TestRunnable(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        while (true) {
            if (Main.Surplus < 0)
                return;
            Main.Surplus--;
            System.out.println(name + "  " + Main.Surplus);
        }
    }
}
```

```
main入口
public static void main(String[] args) {

         TestRunnable runnable = new TestRunnable("runnable1");
         TestRunnable runnable2 = new TestRunnable("runnable2");

         Thread t1 = new Thread(runnable);
         Thread t2 = new Thread(runnable2);

         t1.start();
         t2.start();

    }
```



```
<terminated> Main [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/j
runnable2  8
runnable1  8
runnable1  6
runnable2  7
runnable1  5
runnable2  4
runnable2  2
runnable2  1
runnable2  0
runnable2  -1
runnable1  3
```

这样，我们就看到了，数据肯定是乱了的，当然这个时候我们可以加上一个synchronized的关键字，但是这样也会出现点小问题的

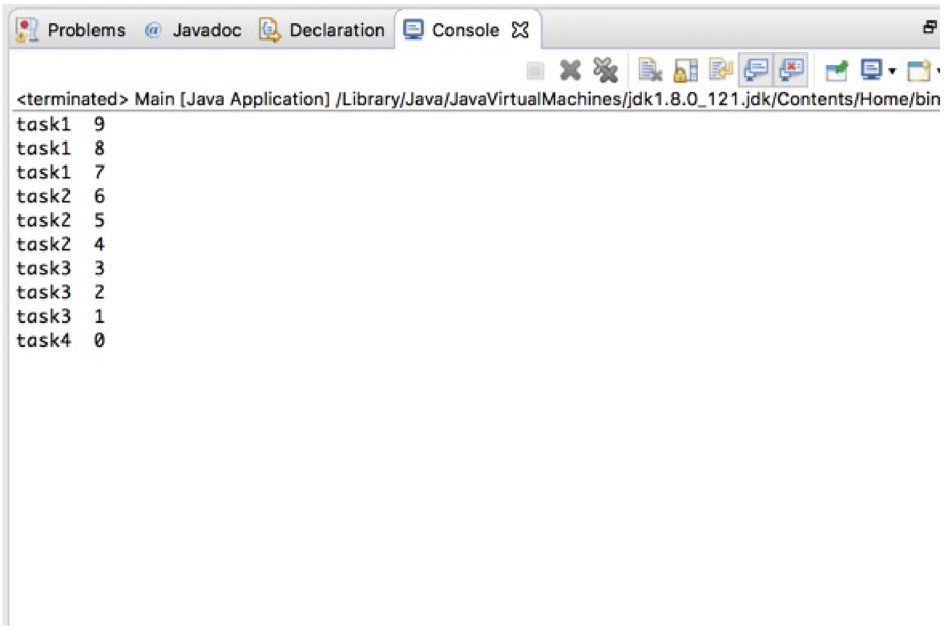


```
<terminated> Main [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/j
runnable1  8
runnable1  7
runnable1  6
runnable1  5
runnable1  4
runnable1  3
runnable1  2
runnable1  1
runnable2  8
runnable1  0
runnable2  -1
```

下面我打算采用一种java内置的线程管理的机制，来解决这个问题，解决这个问题的思路大概就是，我们维护了一个线程池，当有请求操作的时候统统进入线程池，并且我们只开了一个线程，可以让请求顺序执行，顺序调用临界资源，就很安全了。

```java
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Main {
    public static int Surplus = 10;

    private ExecutorService executor = Executors.newSingleThreadExecutor();

    void addTask(Runnable runnable) {
        executor.execute(runnable);
    }

    <V> V addTask(Callable<V> callable) {
        Future<V> submit = executor.submit(callable);
        try {
            return submit.get();
        } catch (InterruptedException e) {
            System.out.println("InterruptedException" + e.toString());
        } catch (ExecutionException e) {
            System.out.println("ExecutionException" + e.toString());
        }
        return null;
    }

    public void testAddTask(String name) {
        addTask(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 3; i++) {
                    if (Main.Surplus <= 0)
                        return;
                    Main.Surplus--;
                    System.out.println(name + "  " + Main.Surplus);
                }

            }
        });
    }

    public void testAddTask2(String name) {
        int count = addTask(new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                for (int i = 0; i < 3; i++) {
                    if (Main.Surplus <= 0)
                        return 0;
                    Main.Surplus--;
                    System.out.println(name + "  " + Main.Surplus);
                }
                return Main.Surplus;
            }
        });

    }

    public void close() {
        executor.shutdown();
    }

    public static void main(String[] args) {
        Main main = new Main();
        main.testAddTask("task1");
        main.testAddTask2("task2");
        main.testAddTask("task3");
        main.testAddTask2("task4");
        main.close();
    }
}
```

在这里，我们定义了两种方法，分别是addTask，具有泛型的addTask，这两种方法实现原理都是一样的，其中一个是有回调的，一个是没有回调的，就看项目需求了吧。



```
<terminated> Main [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin
task1  9
task1  8
task1  7
task2  6
task2  5
task2  4
task3  3
task3  2
task3  1
task4  0
```

然后分别调用这两个方法咯，就可以看到结果是非常有序，且不会混乱的。

当然啊，系统为我们提供这样一个类，肯定不是为了实现这么小的一个功能的，它还有很多功能，我也在进一步的学习中~