

Java 的多线程编程

陈德创 19030500217

西安电子科技大学

日期：2020 年 5 月 29 日

目录

1	小组名单	2
2	题目	2
3	练习	2
3.1	线程的创建	2
3.2	线程的运行	3
3.3	线程的调度	3
3.3.1	线程优先级	3
3.3.2	线程的基本控制	3
3.3.3	结束线程	4
3.4	线程的同步	5
3.4.1	锁	5
3.4.2	死锁	5
3.4.3	线程的交互	6
3.5	线程的生命周期	6
4	题目分析和程序设计	6
5	执行结果	7
6	个人总结	7
	附录	8
A	生产者——消费者模型代码示例	8

1 小组名单

学号	姓名	工作
19030500217	陈德创	完成程序、练习例程、调试

2 题目

1. PPT 中全部小练习
2. 程序设计，程序模拟电影院多个售票口售票，要求：
 - 1). 电影票顺序出售
 - 2). 程序模拟显示售票的详细过程（如：“窗口 X 出售编号 XXX 电影票”）
 - 3). 三个窗口同时出票
 - 4). 出票间隔采用随机控
 - 5). 不能重复出售相同的电影票

3 练习

3.1 线程的创建

JAVA 中有两种创建线程的方法如下：

1. 构造一个 *Runnable* 接口的实现类并将其作为参数传入 *Thread* 类的构造方法中
2. 继承 *Thread* 类并重写其 *run* 方法

PPT 中 12 页和 14 页分别展示了这两种构建线程的方法，结果如下：

```
1    waiting for run...
2    #1->3
3    #2->3
4    #2->2
5    #1->2
6    #1->1
7    #2->1
8    #2->run!
9    #1->run!
```

当然每次执行的结果可能略有不同。这两种方法各有优劣，第二种方法最大的优点就是简单。而第一种方法更符合 *OOP* 的设计逻辑，因为 *Thread* 类是虚拟 CPU 的封装¹，而且利用接口可以省下来继承位，何乐而不为。

在第 16 页的 PPT 中，我们可以得到如下结果：

```
1    #1
2    #2: do something ...
```

¹JAVA 语言程序设计（第三版） P309

深刻地向我们显示了方法一和方法二的差别，毕竟 JAVA 中的继承位这么珍贵，就没必要在线程上浪费一个。

3.2 线程的运行

线程创建后并不会立即执行，必须在调用 *start* 方法后才可以执行。调用 *start* 方法实质上是讲线程设置为 *Runnable* 状态，真正的执行还是得看虚拟机。*start* 方法会单开一个线程运行 *run* 方法。而直接运行 *run* 方法就是普通的函数调用。

查阅资料 JAVA 线程的设计模式似乎是静态代理模式，可以帮助我们只关心 *run* 方法的实现本身，而不必去理会线程的具体管理。

3.3 线程的调度

3.3.1 线程优先级

我们现在学习的线程更像是一种抢蛋糕，同一时间一个 CPU 只能运行一段代码，不同的线程同时去争夺 CPU 时间片，谁抢到就运行谁。我们无法直接控制 CPU 的行为，不过我们可以给出一些策略，比如线程优先级。

线程有三个预设优先级, $MIN_PRIORITY = 0$, $MAX_PRIORITY = 10$, $NORM_PRIORITY = 5$ ，实际上线程的优先级可以是在 $MIN_PRIORITY \sim MAX_PRIORITY$ 的任何整数，不过也没啥用。线程可以依靠 *setPriority* 方法设置优先级，但是最好在 *start* 方法之前设置。

线程的优先级在不同的虚拟机的作用不同，教科书上说线程是严格按照优先级执行的，但是自己实验和查博客是说优先级只是一种建议，怎么来还是得看虚拟机。如下图：



```
package chapter03;

public class PriorityTest {
    public static void main(String[] args) {
        Thread t1 = new Thread(new RunLoop(), name: "#1");
        t1.setPriority(Thread.MIN_PRIORITY);
        Thread t2 = new Thread(new RunLoop(), name: "#2");
        t2.setPriority(Thread.MAX_PRIORITY);
        t2.start();
        t1.start();
    }
}

class RunLoop implements Runnable{
    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            System.out.println(Thread.currentThread().getName() + " --> " + i);
        }
    }
}
```

Output:

```
#1 --> 0
#2 --> 0
#1 --> 1
#2 --> 1
#1 --> 2
#2 --> 2
#1 --> 3
#2 --> 3
#1 --> 4
#2 --> 4
#1 --> 5
#2 --> 5
#1 --> 6
#2 --> 6
#1 --> 7
#2 --> 7
#1 --> 8
#2 --> 8
#1 --> 9
#2 --> 9
#1 --> 10
```

我们给了 *t1* 最低优先级，*t2* 最高优先级并且让 *t2* 先 *start*，但是似乎 *t1* 并不想等 *t2*，甚至抢先一步。

3.3.2 线程的基本控制

线程基本控制有以下的方法：

1. *currentThread()*: 获得当前线程的引用

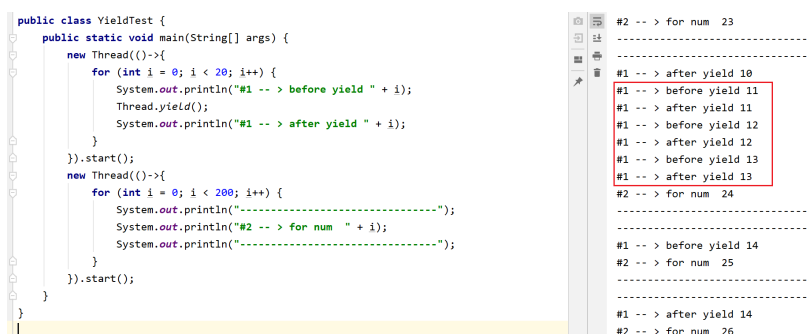
2. `isAlive()`: 测试线程是否或者（线程会在执行完或者抛出异常后死亡）
3. `sleep()`: 令当前线程休眠一段时间，即什么都不做，该方法会让出 CPU 时间片，但是不会交出该线程持有的锁。
4. `join()`: 阻塞当前线程直到目标线程运行结束
5. `yield()`: 让出该线程抢到的 CPU 时间片，并进入 *Runnable* 状态。注意 *yield* 不一定会导致其他线程的执行，因为该线程仍可能抢到时间片。

PPT28 页 结果为:

```
1      t1 awake!
2      t1 join finished
```

如果去掉 *Joiner* 中的 `this.toJoin.join()` 语句，那么显然应该 *Joiner* 先执行完，毕竟 *ToJoin* 睡了这么长时间。但是 `join()` 令其阻塞至 *ToJoin* 执行完，所以它会后执行完。

yield() 方法 为了更好地展示这个方法，我做了如下实验:



```
public class YieldTest {
    public static void main(String[] args) {
        new Thread(()->{
            for (int i = 0; i < 20; i++) {
                System.out.println("#1 -- > before yield " + i);
                Thread.yield();
                System.out.println("#1 -- > after yield " + i);
            }
        }).start();
        new Thread(()->{
            for (int i = 0; i < 200; i++) {
                System.out.println("-----");
                System.out.println("#2 -- > for num " + i);
                System.out.println("-----");
            }
        }).start();
    }
}
```

```
#2 -- > for num 23
-----
#1 -- > after yield 10
#1 -- > before yield 11
#1 -- > after yield 11
#1 -- > before yield 12
#1 -- > after yield 12
#1 -- > before yield 13
#1 -- > after yield 13
#2 -- > for num 24
-----
#1 -- > before yield 14
#2 -- > for num 25
-----
#1 -- > after yield 14
#2 -- > for num 26
```

这个例子很极端，因为第二个线程执行了 200 次而第一个只执行了 20 次。但是我们可以看到红框内的部分，第一个线程在 *yield* 前和 *yield* 后的输出语句挨在了一起，说明线程一再让掉 CPU 时间片后有抢到了。所以不能认为 *yield* 一定会导致其他线程的执行。

3.3.3 结束线程

JDK 提供了 *stop* 方法结束线程，但是已经被废弃。因为 *stop* 内部是通过抛出异常来实现结束线程的，并且释放其所持有的所有锁。因为很多时候需要依靠锁来维护数据的一致性，改方法极易造成数据的不一致。

一般来说我们需要结束的线程应该是有一个永真循环而保持其不会停止，所以我们可以通过设置 *flag* 来判断，即每次循环判断 `flag == true?` 如果不是，则退出。比如 PPT32 页练习。结果为:

```
1      tick ...
2      tick ...
3      quitting Task ...
4      tick ...
5      Tick finished.
```

3.4 线程的同步

多个线程对同共享数据进行操作时，由于多个线程的执行顺序不确定，所以可能导致数据的一致性遭到破坏，甚至可能产生严重后果。比如两个线程对同一个栈进行操作，一个进行压栈，一个进行弹栈。在压栈过程将会有放入数据和栈顶指针移动两个过程。在这放入数据之后指针移动之前另一个线程进行了弹栈操作，那么先然弹出栈顶的元素不是新放入的元素。这将导致数据不一致甚至是新进栈数据的丢失。更严重地可能导致空栈弹栈或者栈元素溢出等可能导致程序崩溃的情况。

3.4.1 锁

为了解决这种情况，JAVA 引入了锁的概念。我们可以利用 *synchronized* 来将一段代码标记为“临界区”并将一个对象设置为锁，同一时间只能有一个线程获得锁。线程获得对象锁后才能执行临界区代码片段（即拥有对该对象的操作权）。此时，其他任何线程无法获得对象锁，也无法执行临界区代码。

我觉得对于需要同一锁的不同线程而言，这更像是把一整块代码“绑”在了一起，要么都执行，要么都不执行。当然实际上不是这样。

3.4.2 死锁

当两个线程互相等待对方持有的锁的时候，死锁就产生了：

“你给我解释一下什么是死锁我就给你 offer”

“你给 offer 我就给你解释什么是死锁”

图 1: 死锁



上图两个线程互相请求对方的锁，产生死锁。JAVA 底层没有对死锁的检验，死锁的避免依赖于程序的编写。死锁的避免可以使用资源排序的方法，在访问多个共享数据对象时，从全局考虑定义一个获得锁的顺序，并在整个程序中都遵守此顺序；释放锁时，要按加锁的反序释放。

3.4.3 线程的交互

线程不仅需要同步数据，其也要进行一定的交互。

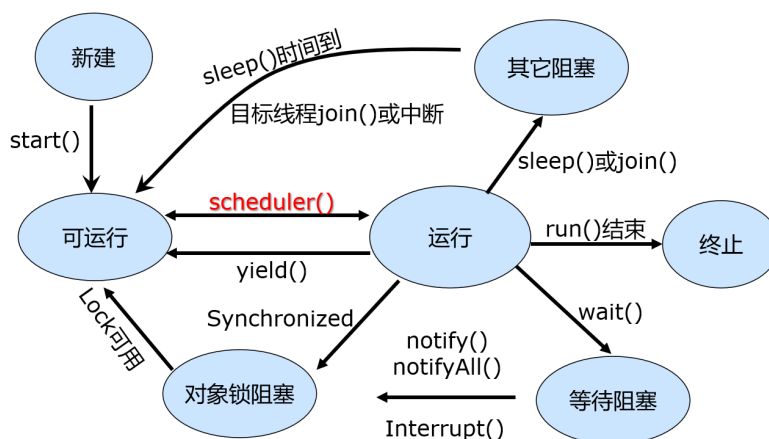
1. `wait()`: 在 `synchronized` 块中调用锁对象的方法将阻塞该线程，释放锁，并进入该对象锁的 `wait pool`，直到被 `notify` 唤醒。
2. `notify()`: 对某一对象调用 `notify` 方法会将该对象 `wait pool` 中的一个线程唤醒，并等待该锁。注意这个唤醒是随机的。
3. `notifyAll()`: 唤醒该对象 `wait pool` 中所有的线程。

一个很经典的模型为“生产者——消费者”模型，我将这段代码放到了附录中。

3.5 线程的生命周期

这个就不多说了，下面内容来自 PPT。

图 2: 线程的生命周期



一般来说一个线程只能跑一次，也就说在一个多次调用某一线程的 `start` 方法是没有用的，并且会抛出 `IllegalThreadStateException` 异常。

4 题目分析和程序设计

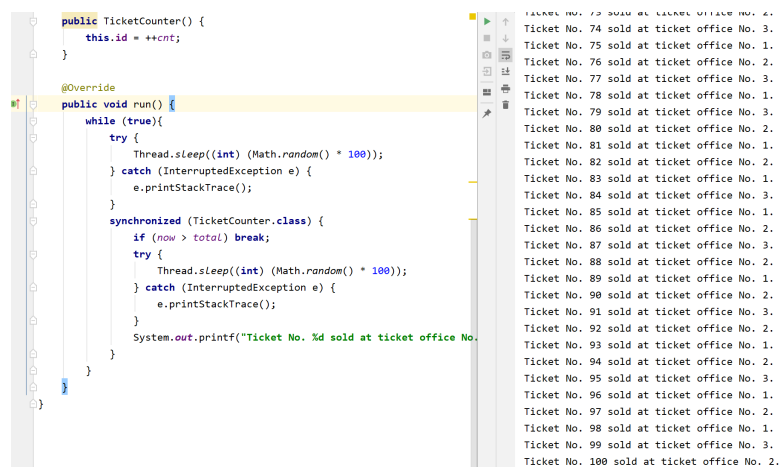
这次还是比较简单的，我们设计一个 `TicketCounter` 类，里面有 `total` 静态属性来表示总的票数，`now` 属性表示当前要售卖的票编号。每次售票即输出售票信息并且讲 `now++`，循环售票，边界条件是 `now > total`。

三个窗口同时售票即开三个线程，为了区分售票口我们可以增加 `id` 属性。随机的时间间隔可以采用 `Thread.sleep((int)(Math.random()*MAX_TIME))` 来模拟，即每次出票睡最多 `MAX_TIME` 的时间。

当然还有最重要的，锁。`synchronized` 应该加在循环语句内，否则在一个线程卖完之前其他的售票口都拿不到锁，这显然不对。对于锁的选择，我们可以直接锁 `TicketCounter.class`。最后要记得 `sleep` 方法应该在 `synchronized` 方法外，因为在 `sleep` 期间线程是拿不到锁的。

5 执行结果

图 3: 执行结果



The screenshot shows an IDE with a Java class named `TicketCounter` on the left and its execution output on the right. The code defines a `run()` method that simulates ticket sales by sleeping for a random duration and then printing the ticket number and office. The output on the right shows 100 tickets sold, alternating between office 1 and office 2.

```
public TicketCounter() {
    this.id = ++cnt;
}

@Override
public void run() {
    while (true) {
        try {
            Thread.sleep((int) (Math.random() * 100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (TicketCounter.class) {
            if (now > total) break;
            try {
                Thread.sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.printf("Ticket No. %d sold at ticket office No. %d\n", id, office);
            id++;
            office = (office + 1) % 2;
        }
    }
}
```

Output:

```
Ticket No. 73 sold at ticket office No. 2.
Ticket No. 74 sold at ticket office No. 3.
Ticket No. 75 sold at ticket office No. 1.
Ticket No. 76 sold at ticket office No. 2.
Ticket No. 77 sold at ticket office No. 3.
Ticket No. 78 sold at ticket office No. 1.
Ticket No. 79 sold at ticket office No. 3.
Ticket No. 80 sold at ticket office No. 2.
Ticket No. 81 sold at ticket office No. 1.
Ticket No. 82 sold at ticket office No. 2.
Ticket No. 83 sold at ticket office No. 1.
Ticket No. 84 sold at ticket office No. 3.
Ticket No. 85 sold at ticket office No. 1.
Ticket No. 86 sold at ticket office No. 2.
Ticket No. 87 sold at ticket office No. 3.
Ticket No. 88 sold at ticket office No. 2.
Ticket No. 89 sold at ticket office No. 1.
Ticket No. 90 sold at ticket office No. 2.
Ticket No. 91 sold at ticket office No. 3.
Ticket No. 92 sold at ticket office No. 2.
Ticket No. 93 sold at ticket office No. 1.
Ticket No. 94 sold at ticket office No. 2.
Ticket No. 95 sold at ticket office No. 3.
Ticket No. 96 sold at ticket office No. 2.
Ticket No. 97 sold at ticket office No. 1.
Ticket No. 98 sold at ticket office No. 3.
Ticket No. 99 sold at ticket office No. 1.
Ticket No. 100 sold at ticket office No. 2.
```

6 个人总结

因为这回 PPT 没有指定具体的尝试哪些，所以就索性直接按照知识点来分来写报告。PPT 中的代码我都看了，而且运行并且尝试更改，这次报告的顺序也是按照 ppt 来的，虽然在这次报告中可能体现不是很明显。

这次应该是除了大作业最后一次 JAVA 报告了，很可惜都要结课了连老师真容都没有见到。不管怎么说，特殊时期，特殊办法。

不知道老师以后还教什么课，希望还有机会选到老师的课。也希望老师保重身体，少熬夜。

附录

A 生产者——消费者模型代码示例

```
1
2 import java.util.Stack;
3
4 public class SyncPCTest {
5     public static void main(String[] args) {
6         SyncStack st = new SyncStack();
7         Producer producer = new Producer(st);
8         Consumer consumer = new Consumer(st);
9         Thread t1 = new Thread(producer);
10        Thread t2 = new Thread(consumer);
11        t2.start();
12        t1.start();
13    }
14
15
16    class Producer implements Runnable{
17
18        private SyncStack st;
19
20        public Producer(SyncStack st) {
21            this.st = st;
22        }
23
24        @Override
25        public void run() {
26            for (int i = 0; i < 200; i++) {
27                try {
28                    Thread.sleep(30);
29                } catch (InterruptedException e) {
30                    e.printStackTrace();
31                }
32                char c = (char) ((int) (Math.random() * 26) + 'A');
33                st.push(c);
34                System.out.printf("P ---> %d \t ---> %c\n", i, c);
35            }
36        }
37    }
38
39    class Consumer implements Runnable{
40
41        private SyncStack st;
42    }
```



```

43     public Consumer(SyncStack st) {
44         this.st = st;
45     }
46
47     @Override
48     public void run() {
49         for (int i = 0; i < 200; i++) {
50             try {
51                 Thread.sleep(30);
52             } catch (InterruptedException e) {
53                 e.printStackTrace();
54             }
55             char c = st.pop();
56             System.out.printf("C ----> %d \t ----> %c\n", i, c);
57         }
58     }
59 }
60
61 class SyncStack{
62     private Stack<Character> st;
63     {
64         st = new Stack<>();
65     }
66
67     public synchronized Character pop(){
68         if (st.empty()) {
69             try {
70                 this.wait();
71             } catch (InterruptedException e) {
72                 e.printStackTrace();
73             }
74         }
75         Character pop = st.pop();
76         return pop;
77     }
78
79     public synchronized void push(Character c){
80         st.push(c);
81         this.notify();
82     }
83
84 }

```