# 华东师范大学数据科学与工程学院实验报告

| | | |
|---|---|---|
| 课程名称:计算机网络与编程 | 年级:22级 | 上机实践成绩: |
| 指导教师:张召 | 姓名:郭夏辉 | 学号:10211900416 |
| 上机实践名称:Java多线程编程1 | 上机实践日期:2023年3月24日 | 上机实践编号:No.04 |
| 组号:1-416 | 上机实践时间:2023年3月24日 | |

## 一、实验目的

- 熟悉Java多线程编程
- 熟悉并掌握线程创建和线程间交互

## 二、实验任务

- 熟悉创建线程方法，继承线程类，实现Runnable接口 (匿名类不涉及)
- 使用 `sleep()` 、`join()` 、 `yield()` 方法对线程进行控制
- 初步接触多线程编程

## 三、实验环境

- Intellij IDEA 2022.3.2
- JDK 19

## 四、实验过程

### task1

使用继承Thread类的方式，编写ThreadTest类，改写 `run()` 方法，方法逻辑为每隔1秒打印 `Thread.currentThread().getId()` ，循环10次。实例化两个 `ThreadTest` 对象，并调用 `start()` 方法，代码及运行结果附在实验报告中。

设计思路

这个题目还是比较基础的，实现起来关键在于重写`run()`函数。有一个很基础的问题，就是在自己写的线程中的`main()`函数里面一定既要有自定义的线程类的实例化，还不能忘了关键的`start()`。而且，Thread类的`sleep()`方法休眠时间是以毫秒计的，这个要留意。

### ThreadTest类

很容易写出代码，如下所示。为了捕获线程运行时被中断的情况，我还加了一个额外的判定。

```
public class ThreadTest extends Thread {
    @Override
    public void run(){
        int seconds=0;
        while(seconds<10){
            try{
                Thread.sleep(1000);
            }catch(InterruptedException e){
                e.printStackTrace();
                System.out.println("Thread has been interrupted.");
            }
            ++seconds;
            System.out.print("距离最初,时间过去了"+seconds+"秒 ");
            System.out.println(Thread.currentThread().getId());
        }

    }

    public static void main(String [] argv){
        ThreadTest mythread1 = new ThreadTest();
        ThreadTest mythread2 = new ThreadTest();
        mythread1.start();
        mythread2.start();
    }
}
```

结果

```
距离最初,时间过去了1秒 25
距离最初,时间过去了1秒 24
距离最初,时间过去了2秒 24
距离最初,时间过去了2秒 25
距离最初,时间过去了3秒 24
距离最初,时间过去了3秒 25
```

```
距离最初,时间过去了4秒 25
距离最初,时间过去了4秒 24
距离最初,时间过去了5秒 25
距离最初,时间过去了5秒 24
距离最初,时间过去了6秒 25
距离最初,时间过去了6秒 24
距离最初,时间过去了7秒 25
距离最初,时间过去了7秒 24
距离最初,时间过去了8秒 距离最初,时间过去了8秒 24
25
距离最初,时间过去了9秒 25
距离最初,时间过去了9秒 24
距离最初,时间过去了10秒 24
距离最初,时间过去了10秒 25
```

可以看到两个线程并发地运行情况，在第8秒时有些特殊，在某个线程第一个输出之后自己的第二个线程并没有来运行，而是另一个线程的第一个进行输出。毕竟这两个输出并不是一个“原子操作”，如果没有阻塞可能有出乎意料的情况。

## task2

给出以下 `BattleObject` 、 `Battle` 、 `TestBattle` 类，请改写 `Battle` 类，实现 `Runnable` 接口， `run()` 方法逻辑为让 `bo1` 调用 `attackHero(bo2)` ，直到 `bo2` 的状态为 `isDestoryed()` ，请完成代码后使用 `TestBattle` 进行测试，将实现代码段及运行结果附在实验报告中。

### 设计思路

这个题目最开始我还是不太明白，在查阅了一些资料后才知道怎么做。首先就是自己并不需要专门开一个文件来定义 `Runnable` 接口然后再在 `Battle` 类中实现它，因为这个接口本身就是Java所自带的。其次就是我想要让这个Battle的过程更加真实一些，就两个对象之间互相攻击，而不是只有一个攻击另外一个。这个我实现的方法是设置了两个线程，然后每个线程代表一方对另一方的攻击（两个线程的攻击方和被攻击方相反）。

### **BattleObject**类

这个类还是比较完善的，直接用题目中的即可。注意要实现我的双方互相攻击，就要加一个特判，即自身已经 `isDestoryed` 了就不能发动攻击了。

```
public class BattleObject {
    public String name;
    public float hp;
    public int attack;
    public void attackHero(BattleObject bo) {
```

```
        try {
// 每次攻击暂停
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if(isDestoryed()){
            System.out.printf("%s can't attack %s due to death.",name,bo.name);
            return;
        }
        bo.hp -= attack;
        System.out.printf("%s 正在攻击 %s, %s 的耐久还剩 %.2f\n", name,
                bo.name, bo.name, bo.hp);
        if (bo.isDestoryed())
            System.out.println(bo.name + "被消灭。");
    }
    public boolean isDestoryed() {
        return 0 >= hp;
    }
}
```

### Battle类

这个重写`run()`方法即可，还是注意要特判。

```
public class Battle implements Runnable {
    private BattleObject bo1;
    private BattleObject bo2;
    public Battle(BattleObject bo1, BattleObject bo2) {
        this.bo1 = bo1;
        this.bo2 = bo2;
    }
    // TODO
    @Override
    public void run(){
        while(!bo2.isDestoryed()){
            bo1.attackHero(bo2);
            if(bo1.isDestoryed()){
                break;
            }
        }
    }

}
```

### TestBattle类

这个按照我的想法，设置两个线程分别开始运行即可。

```
public class TestBattle {
    public static void main(String[] args) {
        BattleObject bo1 = new BattleObject();
        bo1.name = "Object1";
        bo1.hp = 600;
        bo1.attack = 50;
        BattleObject bo2 = new BattleObject();
        bo2.name = "Object2";
        bo2.hp = 500;
        bo2.attack = 40;
        // TODO
        /*
        Battle battle = new Battle(bo1,bo2);
        Thread thread1 = new Thread(battle,"Thread-1");
        thread1.start();
        */

        // Extend
        Battle battle1 = new Battle(bo1,bo2);
        Battle battle2 = new Battle(bo2,bo1);
        Thread thread1 = new Thread(battle1,"Thread-1");
        Thread thread2 = new Thread(battle2,"Thread-2");
        thread1.start();
        thread2.start();
    }
}
```

结果

```
Object1 正在攻击 Object2, Object2 的耐久还剩 450.00
Object2 正在攻击 Object1, Object1 的耐久还剩 560.00
Object1 正在攻击 Object2, Object2 的耐久还剩 400.00
Object2 正在攻击 Object1, Object1 的耐久还剩 520.00
Object1 正在攻击 Object2, Object2 的耐久还剩 350.00
Object2 正在攻击 Object1, Object1 的耐久还剩 480.00
Object1 正在攻击 Object2, Object2 的耐久还剩 300.00
Object2 正在攻击 Object1, Object1 的耐久还剩 440.00
Object1 正在攻击 Object2, Object2 的耐久还剩 250.00
Object2 正在攻击 Object1, Object1 的耐久还剩 400.00
Object1 正在攻击 Object2, Object2 的耐久还剩 200.00
Object2 正在攻击 Object1, Object1 的耐久还剩 360.00
```

```
Object1 正在攻击 Object2, Object2 的耐久还剩 150.00
Object2 正在攻击 Object1, Object1 的耐久还剩 320.00
Object1 正在攻击 Object2, Object2 的耐久还剩 100.00
Object2 正在攻击 Object1, Object1 的耐久还剩 280.00
Object1 正在攻击 Object2, Object2 的耐久还剩 50.00
Object2 正在攻击 Object1, Object1 的耐久还剩 240.00
Object1 正在攻击 Object2, Object2 的耐久还剩 0.00
Object2被消灭。
Object2 can't attack Object1 due to death.
```

## task3

完善代码，用join方法实现正常的逻辑，并将关键代码和结果写到实验报告中。

```
public class ThreadTest03 implements Runnable{
    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) throws InterruptedException {
        ThreadTest03 join = new ThreadTest03();
        Thread thread1 = new Thread(join, "上课铃响");
        Thread thread2 = new Thread(join, "老师上课");
        Thread thread3 = new Thread(join, "下课铃响");
        Thread thread4 = new Thread(join, "老师下课");
        // TODO
    }
}
```

### 设计思路

我的思路可能比较保守一些，但是应该是正确的。`join()`使当前线程暂停执行，直到被调用`join()`方法的线程执行完毕后才恢复当前线程的运行。可以这样想，在某个进程刚刚开始的时候，便有一个工作用的主线程了，然后这个问题中的四个操作(上课铃响,老师上课,下课铃响,老师下课)分别用一个线程来做。逻辑上，这四个线程的执行应该是有先后顺序的，我严格地让主线程等待某个线程执行完毕后才去执行下一个，这样可以实现四个线程按顺序执行。还有一个值得注意的地方就是`join()`之前一定要有对应线程的`start()`。

### **ThreadTest03**类

代码补全的内容

```
        thread1.start();
        thread1.join();
        thread2.start();
        thread2.join();
        thread3.start();
        thread3.join();
        thread4.start();
        thread4.join();
```

完整代码

```
public class ThreadTest03 implements Runnable{
    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) throws InterruptedException {
        ThreadTest03 join = new ThreadTest03();
        Thread thread1 = new Thread(join, "上课铃响");
        Thread thread2 = new Thread(join, "老师上课");
        Thread thread3 = new Thread(join, "下课铃响");
        Thread thread4 = new Thread(join, "老师下课");
        // TODO
        thread1.start();
        thread1.join();
        thread2.start();
        thread2.join();
        thread3.start();
        thread3.join();
        thread4.start();
        thread4.join();
    }
}
```

结果

```
上课铃响
老师上课
下课铃响
老师下课
```

## task4

完善代码，将助教线程设置为守护线程，当同学们下课时，助教线程自动结束。并将关键代码和结果写到实验报告中。

```
public class ThreadTest04 implements Runnable{
    @Override
    public void run(){
        int worktime = 0;
        while(true){
            System.out.println("助教在教室的第"+ worktime +"秒");
            try{
                Thread.currentThread().sleep(1000);
            }catch (InterruptedException e){
                e.printStackTrace();
            }
            worktime ++;
        }
    }
    public static void main(String[] args) throws InterruptedException{
        // TODO
        for(int i = 0; i < 10; i++){
            thread.sleep(1000);
            System.out.println("同学们正在上课");
            if(i == 9){
                System.out.println("同学们下课了");
            }
        }
    }
}
```

### 设计思路

在这个问题中，助教线程可以认为是上课线程的守护线程，然后只需要正常地在开始运行上课线程之前设置守护线程即可。

### 完整代码

```
public class ThreadTest04 implements Runnable{
    @Override
    public void run(){
        int worktime = 0;
        while(true){
```

```
            System.out.println("助教在教室的第"+ worktime +"秒");
            try{
                Thread.currentThread().sleep(1000);
            }catch (InterruptedException e){
                e.printStackTrace();
            }
            worktime ++;
        }
    }
    public static void main(String[] args) throws InterruptedException{
        // TODO
        ThreadTest04 classThread = new ThreadTest04();
        Thread thread = new Thread(classThread,"助教");
        thread.setDaemon(true);
        thread.start();

        for(int i = 0; i < 10; i++){
            thread.sleep(1000);
            System.out.println("同学们正在上课");
            if(i == 9){
                System.out.println("同学们下课了");
            }
        }
    }
}
```

结果

```
助教在教室的第0秒
同学们正在上课
助教在教室的第1秒
同学们正在上课
助教在教室的第2秒
同学们正在上课
助教在教室的第3秒
同学们正在上课
助教在教室的第4秒
同学们正在上课
助教在教室的第5秒
同学们正在上课
助教在教室的第6秒
同学们正在上课
助教在教室的第7秒
同学们正在上课
```

```
助教在教室的第8秒
同学们正在上课
助教在教室的第9秒
同学们正在上课
同学们下课了
```

通过结果，可以看到Java对守护线程的回收机制。守护线程会在没有用户线程运行时自动结束。当所有用户线程都结束时，JVM就会结束守护线程，然后退出程序。

这里结合操作系统中的所学知识，我觉得守护线程通常执行辅助性的任务，但它们不能持有任何会导致用户线程阻塞的锁。否则守护线程在JVM关闭时会被强制中断，这些锁可能永远无法释放，从而导致程序死锁。

## task5

给出 `TestVolatile`类，测试 main 方法，观察运行结果，并尝试分析结果。

### TestVolatile类

```java
// if variable is not volatile, this example may not be terminated
// but this behaviour may differ on some machines
class TestVolatile extends Thread{
    //volatile
    // volatile boolean sayHello = true;
    boolean sayHello = true;
    public void run() {
        long count=0;
        while (sayHello) {
            count++;
        }
        System.out.println("Thread terminated." + count);
    }
    public static void main(String[] args) throws InterruptedException {
        TestVolatile t = new TestVolatile();
        t.start();
        Thread.sleep(1000);
        System.out.println("after main func sleeping...");
        t.sayHello = false;
        t.join();
        System.out.println("sayHello set to " + t.sayHello);
    }
}
```

### 结果

```
after main func sleeping...
```

然后程序就卡了/捂脸。

### 分析与修改

首先来看一下这个程序正常情况下在干什么。`TestVolatile`类包含了一个`sayHello`变量，重写的`run()`方法在不停地循环直到`sayHello`变成了false，最后输出结果。`main()`方法中创建并启动了一个线程，然后主线程会睡眠1秒钟，然后将新线程的`sayHello`值设为false,等待其结束并输出sayHello。

在这个过程中，线程有可能将`sayHello`变量的值缓存至寄存器中，而不是放到主存中。不同线程虽然共享相同的虚拟地址空间，即某个线程对主存的修改其他线程是能观察到的；但是不同线程的寄存器是不一样的，某个线程对寄存器的修改其他线程无法观察地到，这样就出现了程序的错误。

只用这样修改即可。

```
    volatile boolean sayHello = true;
```

在java中（其实C语言也是），如果用volatile来修饰某个变量，那么它的值会被强制刷新到主存中而不是寄存器中。这样做使得一个线程修改了它的值时，其他线程能够立即观察到，保证了变量在多线程之间的可见性和一致性。

修改后的结果如下所示。

```
after main func sleeping...
Thread terminated.2845132701
sayHello set to false
```

可以看到主线程休眠的那一秒中新线程累加了2845132701次，还是很惊人的。

## task6

给出 `PlusMinus` 、 `TestPlus` 、 `Plus` 三个类，描述 `TestPlus` 的 `main` 方法的运行逻辑，并多次运行，观察输出结果，并尝试分析结果。

### PlusMinus类

这个类还是十分简单的。`plusOne`方法会将num加1，`minusOne`方法会将num减1，`printNum`方法会返回num的当前值。

```
public class PlusMinus {
    public int num;
    public void plusOne(){
        num = num + 1;
    }
    public void minusOne(){
        num = num - 1;
    }
    public int printNum(){
        return num;
    }
}
```

### TestPlus类

这个是程序的主体，透过这个我可以看到这个程序具体要干什么。它创建了一个`PlusMinus`对象，然后创建了10个`Plus`线程，并启动这些线程进行计算。每个Plus线程会执行10000次plusOne方法，将num加上10000。最后，主线程等待所有Plus线程执行完毕，并输出num的最终值。

```
public class TestPlus {
    public static void main(String[] args) throws InterruptedException {
        PlusMinus plusMinus = new PlusMinus();
        plusMinus.num = 0;
        int threadNum = 10;
        Thread[] plusThreads = new Thread[threadNum];
        for(int i=0;i<threadNum;i++){
            plusThreads[i] = new Plus(plusMinus);
        }
        for(int i=0;i<threadNum;i++){
            plusThreads[i].start();
        }
        for(int i=0;i<threadNum;i++){
            plusThreads[i].join();
        }
        System.out.println(plusMinus.printNum());
    }
}
```

### Plus类

这个类也是比较简单的。每次执行run方法时，会调用`PlusMinus`的`plusOne()`方法，将num加上10000。

```
class Plus extends Thread {
    Plus(PlusMinus pm) {
        this.plusMinus = pm;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            plusMinus.plusOne();
        }
    }

    PlusMinus plusMinus;
}
```

## 结果

第一次运行

```
21120
```

第二次运行

```
22865
```

第三次运行

```
20469
```

可以看到不仅每次都没有达到预期的结果(100000),而且每次竟然运行的结果还不一样。

## 分析与修改

`PlusMinus`对象中的num变量是被多个线程共享的，会出现线程竞争问题。根据操作系统的所学知识，线程竞争有三种具体的形式：原子性问题(原子性:一个操作不可分割、不可中断)、可见性问题(可见性:一个线程对共享变量的修改能够被其他线程及时地观察)和有序性问题(有序性:程序执行的顺序和预期的顺序一致)。在这个问题中，碰到的是原子性问题。

无论是什么问题，解决线程竞争问题的一般途径还是采用同步机制来保证线程之间的同步和互斥。在Java中，synchronized是一种同步机制，专门用来解决多线程并发执行时的数据竞争问题。方法被synchronized修饰后，同一时刻只有一个线程能执行该方法，其他线程会被阻塞，直到当前线程执行完毕，释放锁之后才能继续访问。所以我主要是修改`PlusMinus`的`plusOne()`方法，因为这个问题中只用了`plusOne()`。

```
    public synchronized void plusOne(){
        num = num + 1;
    }
```

修改后的结果如预期一样。

```
100000
```

还有最后一个问题，这个能不能去只用volatile修饰`PlusMinus`中num变量的类型呢？经过分析和测试，是不行的。首先，自加操作并不是一个原子操作，仅仅用volatile修饰并未解决原子性问题。其次，task5中之所以添加volatile修饰就能正常，是因为`TestVolatile`碰到的是可见性问题，而不是原子性问题。

## 五、总结

线程是复杂的，但是在实际应用中却有着很广泛的用途。比如在加载web网页时，往往这个网页上有很多的图像资源，如果只在一个线程中加载， IO 密集型操作很容易引起页面的卡顿。但是在多线程条件下，每个线程都负责加载一部分，并发地执行时，可以显著提升页面的响应速度。

通过本次实验，从的Java多线程编程入手，我了解了相关的基本操作，更透过遇到的几个问题和正在学习的操作系统知识联系了起来。最大化线程潜力的核心在于消除其中可能存在的诸如线程竞争这样的问题。不断地摸索和总结让我有了很大的收获，愿我未来能最大化多线程的价值。