

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：2021 级

上机实践成绩：

指导教师：张召

姓名：彭一琄

学号：10215501412

上机实践名称：Java 多线程编程 1

上机实践日期：2023. 3. 24

上机实践编号：4

组号：

上机实践时间：9:50

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、使用环境

- IntelliJ IDEA
- JDK 版本：Java 19

四、实验过程

1. 创建线程

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。

继承 Thread 类并改写 run 方法，让线程每 1000 毫秒打印一次线程 ID，两次打印的中间用空格分隔：

```
class ThreadTest extends Thread {  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                throw new RuntimeException(e);  
            }  
            System.out.print(Thread.currentThread().getId()+" ");  
        }  
    }  
}
```

创建实例对象并运行进程：

```
public class Test {
    public static void main(String[] args) {
        ThreadTest thread1 = new ThreadTest();
        ThreadTest thread2 = new ThreadTest();
        thread1.start();
        thread2.start();
    }
}
```

运行结果：

```
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe" "-javaagent
31 32 31 32 31 32 31 32 31 32 31 32 31 32 31 32 31 32
进程已结束,退出代码0
```

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。

改写battle类，实现runnable接口，实现bo2持续攻击bo1的效果：

```
public class Battle implements Runnable{
    private BattleObject bo1;
    private BattleObject bo2;
    public Battle(BattleObject bo1, BattleObject bo2) {
        this.bo1 = bo1;
        this.bo2 = bo2;
    }

    @Override
    public void run() {
        while(!bo2.isDestoryed()){
            bo1.attackHero(bo2);
        }
    }
}
```

创建battle的实例，并用这个实例作为target来创建线程对象，并启动线程：

```
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;  
        Battle b = new Battle(bo1,bo2);  
        Thread thread1=new Thread(b, name: "线程1");  
        thread1.start();  
    }  
}
```

程序运行结果:

```
Object1 正在攻击 Object2, Object2 的耐久还剩 450.00  
Object1 正在攻击 Object2, Object2 的耐久还剩 400.00  
Object1 正在攻击 Object2, Object2 的耐久还剩 350.00  
Object1 正在攻击 Object2, Object2 的耐久还剩 300.00  
Object1 正在攻击 Object2, Object2 的耐久还剩 250.00  
Object1 正在攻击 Object2, Object2 的耐久还剩 200.00  
Object1 正在攻击 Object2, Object2 的耐久还剩 150.00  
Object1 正在攻击 Object2, Object2 的耐久还剩 100.00  
Object1 正在攻击 Object2, Object2 的耐久还剩 50.00  
Object1 正在攻击 Object2, Object2 的耐久还剩 0.00  
Object2被消灭。
```

2. 线程控制

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。

Join 方法的作用是让主线程等待, 一直到调用 join 方法的线程执行结束为止。Join 方法调用了 wait 方法, 在调用的线程 isAlive 时, 主线程会一直等待。因此, 如下操作让四个进程一定会顺序执行。

```
public static void main(String[] args) throws InterruptedException {  
    ThreadTest03 join = new ThreadTest03();  
    Thread thread1 = new Thread(join, name: "上课铃响");  
    Thread thread2 = new Thread(join, name: "老师上课");  
    Thread thread3 = new Thread(join, name: "下课铃响");  
    Thread thread4 = new Thread(join, name: "老师下课");  
    thread1.start();  
    thread1.join();  
    thread2.start();  
    thread2.join();  
    thread3.start();  
    thread3.join();  
    thread4.start();  
}
```

运行结果:

```
"C:\Program Files  
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。

守护线程 daemon thread 可以在后台运行, 直到所有前台线程运行完毕。

创建助教线程并设置为守护线程:

```
public static void main(String[] args) throws InterruptedException{  
      
    ThreadTest04 assist = new ThreadTest04();  
    Thread thread4=new Thread(assist, name: "助教线程");  
    thread4.setDaemon(true);  
    thread4.start();  
    for(int i = 0; i < 10; i++){  
        Thread.sleep( millis: 1000);  
        System.out.println("同学们正在上课");  
        if(i == 9){  
            System.out.println("同学们下课了");  
        }  
    }  
}
```

程序运行结果：

```
助教在教室的第0秒
助教在教室的第1秒
同学们正在上课
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
助教在教室的第10秒
同学们下课了

进程已结束,退出代码0
```

3. 线程安全初探

Task5: 给出 TestVolatile 类，测试 main 方法，观察运行结果，并尝试分析结果。

当变量 sayHello 加上 volatile 修饰符后，主线程修改了线程 t 的 sayHello 值后，线程 t 结束。其中 while 循环的次数是不确定的，取决于 cpu 执行这些指令的速度。

```
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);
    }
}
```

```
after main func sleeping...
Thread terminated.1788640269
sayHello set to false
```

进程已结束,退出代码0

但是如果把修饰符 `volatile` 去掉,那么主线程更改的值就不会起效果。

```
boolean sayHello = true;
```

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

那么在主线程和 `t` 线程中分别打印出 `sayHello` 的值:

这是因为变量 `sayHello` 存储在内存中，是主线程和 `t` 线程的共享变量，但是由于每个线程都是从工作内存中读取这些变量的值，也就是这些值被每个线程分别缓存。而 `volatile` 保证了内存的可见性，也就是当一个线程修改了某一个共享变量的值，其他线程能够立刻知道该变更。

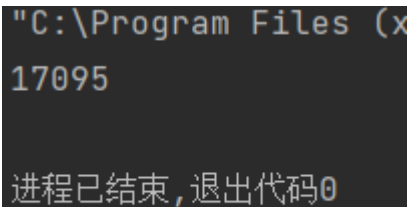
Task6: 给出 PlusMinus 、TestPlus 、Plus 三个类，描述 TestPlus 的 main 方法的运行逻辑，并多次运行，观察输出结果，并尝试分析结果。

PlusMinus 类实现了+1 -1 操作，而 Plus 类可以在线程中运行，它的构造函数需要一

个 PlusMinus 实例，并且对其进行了 10000 次+1 操作。

TestPlus 创建了 PlusMinus 实例，并把初始值设为 0，随后创建了长度为 10 的线程数组，每个线程里面跑一个 Plus 实例，这些 Plus 实例使用的都是同一个 PlusMinus 实例，所以它们同时在对 PlusMinus 做+1 操作。

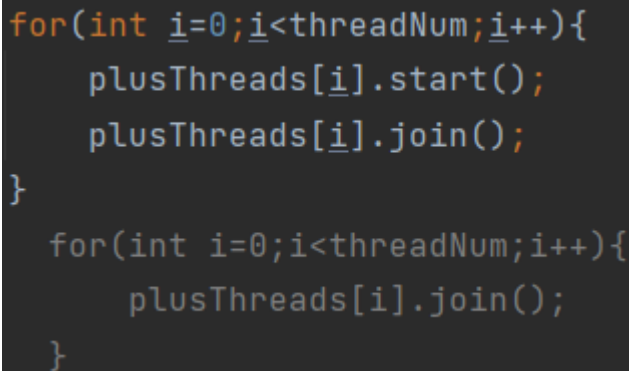
对 TestPlus 进行测试，可以看到运行结果并不是 100000：



```
"C:\Program Files (x86)
17095

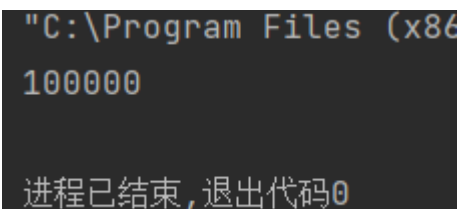
进程已结束,退出代码0
```

去掉 join 方法的循环，在 start 的同时加上 join 方法：



```
for(int i=0;i<threadNum;i++){
    plusThreads[i].start();
    plusThreads[i].join();
}
for(int i=0;i<threadNum;i++){
    plusThreads[i].join();
}
```

运行结果：

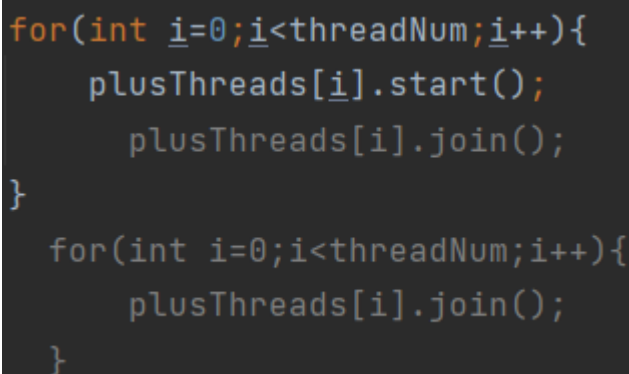


```
"C:\Program Files (x86)
100000

进程已结束,退出代码0
```

此时线程 1-10 是顺序执行的。

如果不使用 join 进行线程同步：



```
for(int i=0;i<threadNum;i++){
    plusThreads[i].start();
    plusThreads[i].join();
}
for(int i=0;i<threadNum;i++){
    plusThreads[i].join();
}
```

运行结果：


```
"C:\Program Files (x86)
1959

进程已结束,退出代码0
```

可以看到实际累加的次数变得更少了。

这是因为 `PlusMinus` 中 `num=num+1` 的操作不具有原子性，这条指令包含 3 个操作：读取 `num` 的值，进行 +1 操作，写入新的值。而在这些线程同时执行的时候，有的线程只执行了读取，还没有 +1，另一个线程就执行读取并 +1，此时第一个线程读取的值就是过期的了，那么这个线程的 +1 操作也是无效的，最终这两个线程的执行结果就是 `num` 只 +1 了一次。

如果在 `PlusMinus` 的方法 `PlusOne` 前面加上修饰符 `synchronized`，就可以使这个操作变得原子性，不可以被打断：

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

最终输出结果：

```
"C:\Program Files (x86)\Java
100000

进程已结束,退出代码0
```

可以得到与线程顺序执行相同的结果。

五、总结

Java 的多线程编程中，在一个新的线程中执行指令有两种方式，分别是继承 `Thread` 类和实现 `Runnable` 接口。继承 `Thread` 类时需要改写 `run` 方法，然后调用实例的 `start` 方法即可。当一个类实现 `Runnable` 接口后，需要在创建实例后把实例作为 `Thread` 类的构造方法的参数传进去，从而实现在进程中执行代码。

Java 的线程控制有三种主要方法，在调用 `join` 方法的线程执行完毕前不允许其他线程执行；在调用 `setDaemon` 方法后，线程结束的标准改为所有前台线程执行完毕，而不需要执行完守护线程本身或后台线程的指令；`yield` 用于正在执行的线程，使线程让出 CPU 资源，回到一个准备抢占 cpu 的线程队列。

Java 并发编程有两个很重要的特性：可见性、原子性。可见性是指多个线程的公共变量可以在被修改时马上通知到其他线程，原子性是指一系列指令在执行时是否会被其他线程打断。其中 `volatile` 可以保证可见性，`synchronized` 可以保证原子性。