

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：2022

上机实践成绩：

指导教师：张召

姓名：朱天祥

学号：10225501461

上机实践名称：java 多线程 2

上机实践日期：23/3/31

上机实践编号：lab5

组号：

上机实践时间：

一、实验目的

1. 熟悉 java 多线程编程
2. 熟悉并掌握线程同步和线程交互

二、实验任务

1. 学习使用 synchronized 关键字
2. 学习使用 wait () , notify () , notifyAll () 方法进行线程交互

三、使用环境

1. IntelliJ IDEA
2. JDK 版本：Java19

四、实验过程

Task1: 使用 synchronized 关键字修改 plus 类的代码

1. 使用锁对象

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

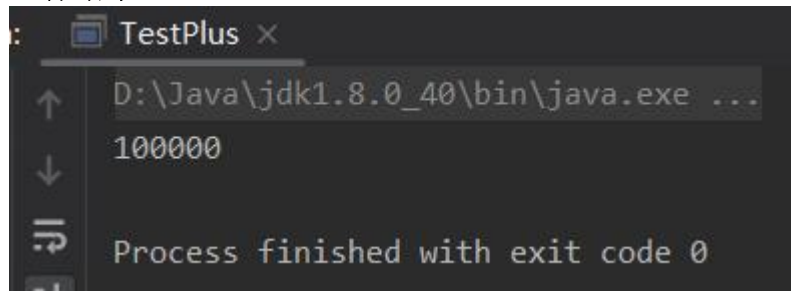
在 plus 线程中将成员变量 plusminus 作为锁对象，各个线程只有一个线程能够得到锁对象，以此实现互斥访问 plusMinus

2. 同步方法

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

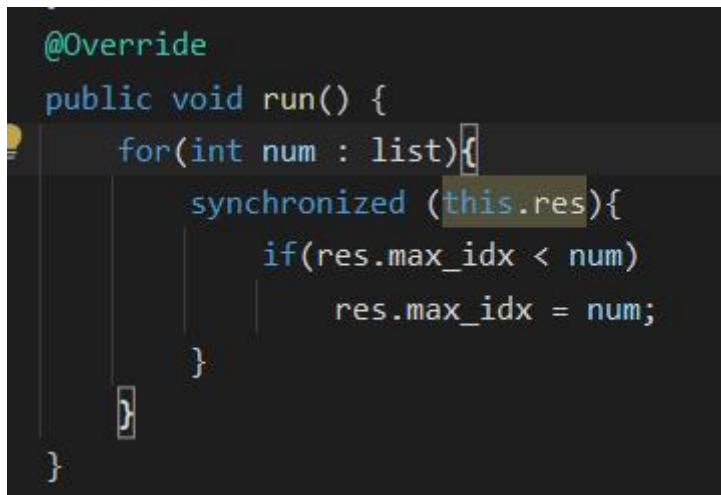
将 `plusminux` 中加 1 和减 1 操作都上锁，使得只有一个线程在操作 `plusMinux`

运行结果：



```
TestPlus x
D:\Java\jdk1.8.0_40\bin\java.exe ...
100000
Process finished with exit code 0
```

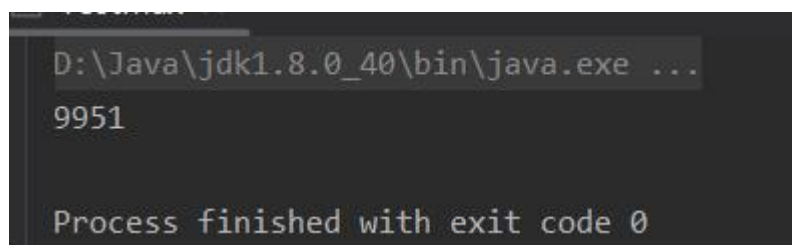
Task2:



```
@Override
public void run() {
    for(int num : list){
        synchronized (this.res){
            if(res.max_idx < num)
                res.max_idx = num;
        }
    }
}
```

值得注意的是不能把 `synchronized` 代码段放在 `if` 中，因为如果一个线程执行完 `if` 即将修改 `max_idx` 值时切换到另一线程，那么就会引起最大值被另一线程覆盖。

运行结果：



```
D:\Java\jdk1.8.0_40\bin\java.exe ...
9951
Process finished with exit code 0
```

Task3: 设计 3 个线程彼此争夺资源造成死锁的场景（基于文档中代码）

```

public static void main(String[] args) throws InterruptedException {
    PlusMinus plusMinus1 = new PlusMinus();
    plusMinus1.num = 1000;
    PlusMinus plusMinus2 = new PlusMinus();
    plusMinus2.num = 1000;
    PlusMinus plusMinus3 = new PlusMinus();
    plusMinus3.num = 1000;
    MyThread thread1 = new MyThread(plusMinus1, plusMinus2, plusMinus3, _tid: 1);
    MyThread thread2 = new MyThread(plusMinus1, plusMinus2, plusMinus3, _tid: 2);
    MyThread thread3 = new MyThread(plusMinus1, plusMinus2, plusMinus3, _tid: 3);
    Thread t1 = new Thread(thread1);
    Thread t2 = new Thread(thread2);
    Thread t3 = new Thread(thread3);
    t1.start();
    t2.start();
    t3.start();
    t1.join();
}

```

类似地创建三个线程。

```

if (tid == 1) {
    synchronized (pm1) {
        System.out.println("thread" + tid + "正在占用 plusMinus1");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("thread" + tid + "试图继续占用 plusMinus2");
        System.out.println("thread" + tid + "等待中...");
        synchronized (pm2) {
            System.out.println("thread" + tid + "成功占用了 plusMinus2");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("thread" + tid + "试图继续占用 plusMinus3");
            System.out.println("thread" + tid + "等待中...");
            synchronized (pm3) {
                System.out.println("thread" + tid + "成功占用了 plusMinus3");
            }
        }
    }
}

```

以上是 run 方法中 id 为 1 的线程的代码，首先争抢 pm1，sleep1 秒后争夺 pm2，而后争夺 pm3。Id 为 2 和 id 为 3 的线程代码是类似的，2 争夺的顺序是 pm2、pm3、pm1，线程 3 争夺的顺序是 pm3、pm1、pm2。

当然构造函数也要做一定的修改

```
MyThread(PlusMinus _pm1, PlusMinus _pm2, PlusMinus _pm3, int _tid) {  
    this.pm1 = _pm1;  
    this.pm2 = _pm2;  
    this.pm3 = _pm3;  
    this.tid = _tid;  
}  
PlusMinus pm1;  
PlusMinus pm2;  
PlusMinus pm3;  
int tid;
```

运行结果:

```
D:\Java\jdk1.8.0_40\bin\java.exe ...  
thread1正在占用 plusMinus1  
thread3正在占用 plusMinus3  
thread2正在占用 plusMinus2  
thread3试图继续占用 plusMinus1  
thread1试图继续占用 plusMinus2  
thread2试图继续占用 plusMinus3  
thread1等待中...  
thread3等待中...  
thread2等待中...
```

可以看到三个线程互相争夺对方手中的资源，最终造成死锁。

Task4:

synchronized 作用:

将 main 线程中唯一的 plusMinuxOne 作为锁对象，实现互斥访问其 num 属性，不会造成加 1 后的数值对减 1 后的数值进行覆盖。

运行结果:

因为太长了我就不把截图放在这里了，因为 plus 的睡眠时间比 minus 长 10 倍，所以运行结果大概就是刚开始 num 从 50 减到 40，然后加 1 到 41 后再从 41 减到 31 以此类推，最后减到 1 后，每 0.1 秒 num 加到 2 后立马又减到 1，如此循环下去。

Cpu 情况:



用任务管理器看的 cpu 利用率的曲线图，刚开始 minus 线程不断减 num 因此频繁占用 cpu，到后面 num 变为 1 后大部分时间都在等待 num 变成 2，因此 cpu 利用率又变低了，并且在持续的占用 cpu，程序终止后 cpu 占用率恢复正常，因此使用 while 循环的方法是可能导致 cpu 被持续占用的，这也就导致了资源的浪费。

Task5:

当有多个 minus 线程后，会有 num 减到负数的现象，其原因是当 num 减到 2 左右的的大于 1 的数值后，第一个线程结束了 while (num == 1) 的循环准备让 num 减 1，而此时切换到另外一个线程也同样结束了 while 循环准备减 1 操作，此时会让 num 减 2，造成 num 变为 0 或负数的情况，从此以后每个 minus 线程都不会收到 while 的约束而不停的让 num 减 1。

解决方案:

```
public void run() {  
    while (true) {  
        if(pmo.num > 1){  
            synchronized (pmo){  
                if(pmo.num > 1)  
                    pmo.minusOne();  
            }  
        }  
        try {  
            Thread.sleep(10);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

修改 minus 线程的代码，用 synchronized 关键字使得 num>1 和减 1 的操作具有原子性，不再会有两个线程同时减 1 的情况。

Task6:

运行结果:

```
D:\Java\jdk1.8.0_40\bin\java.exe ...
t6 add product: product
t5 get product: product
t6 add product: product
t5 get product: product
t5 get product: product
t6 add product: product
t6 add product: product
t5 get product: product
t6 add product: product
t5 get product: product
```

t5 是最后一个 start 的消费者进程，不知道为什么总是最后一个 start 的进程能够获得产品。把 t4 换成最后一个 start 的进程后运行结果如下：

```
D:\Java\jdk1.8.0_40\bin\java.exe ...
t6 add product: product
t4 get product: product
t4 get product: product
t6 add product: product
t6 add product: product
t4 get product: product
```

现在只有 t4 获得产品。

我猜想的原因是 notifyall 底层是把等待队列一个个唤醒的，唤醒后将队列全部清空，第一个被唤醒的线程结束循环得到产品，其余的线程依然在 while 中继续等待，这应该是跟 notifyall 的唤醒顺序有所关联。

更改 while 为 if 后运行结果如下：

```
Exception in thread "Thread-4" Exception in thread "Thread-0" Exception in thread "Thread-1" Exception in thread "Thread-2" java.util.NoSuchElementException
at java.util.LinkedList.removeFirst(LinkedList.java:270)
at java.util.LinkedList.remove(LinkedList.java:685)
at thread.ProductFactory.getProduct(Test.java:108)
at thread.Test$5.run(Test.java:64)
```

报了 noelement 错误，定位到代码中是 arraylist 的 remove 方法报的错，因此应该是 arraylist 为空时调用了 remove 方法导致了该异常，其原因为：notifyall 后等待的几个线程全部被唤醒，因此可能会导致两个线程同时在 if 代码块结束后调用 remove 方法，而 list 中只有一个元素，只可调用一次，这就导致了在 list 为空时调用 remove 方法而报错。

Task6 (optional) :

Notifyall 与 notify 的不同点在于 notifyall 能够唤醒所有等待线程，而 notify 只能唤醒一个线程。那么如果只使用 notify() 可能会导致某些线程，一直处于等待队列中，而永远不会被唤醒并获得执行权。notify() 具有随机唤醒的特点，导致在多条条件的情况下，会导致某些线程永远不会被通知到。稳妥的方式，是使用 notifyAll()，让等待中的线程，都有一次再执行的权利。

Task7

minus 进程:

```
Thread t1 = new Thread() {
    public void run() {
        while (true) {
            synchronized (pmo) {
                while (pmo.num == 1) {
                    try {
                        pmo.wait();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
                pmo.minusOne();
            }
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
```

需要注意 wait 方法一定要在 synchronized 代码段中，不然会报非法监视状态异常，minusOne 方法也一定要在 syn 代码段内，以此保证只有拿到 pmo 锁的线程才能进行减 1 操作，不然长时间运行后可能会有 num 变为负数的情况。

plus 进程:

```
Thread t4 = new Thread() {
    public void run() {
        while (true) {
            synchronized (pmo) {
                pmo.plusOne();
                pmo.notifyAll();
            }
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Cpu 使用情况:



与 task5 不同的是当 num 减到 1 之后, cpu 的占用率没有之前用 while 循环那么高, 节省了大量的 cpu 资源。

五、总结

1. **synchronized** 关键字能够保证互斥访问共享资源, 使得操作具有原子性。
2. 当各个线程互相争夺对方手中的资源时会导致两个线程都无法顺利进行下去, 这个线程称为死锁, 我们需要谨慎使用 **synchronized** 关键字防止出现死锁现象。
3. 调用 **wait** 方法可以使得当前线程放弃某一锁或者说是资源, 等待其他线程 **notify** 后重新获得锁。需要注意的是 **wait** 和 **notify** 一定要在 **synchronized** 关键字代码段或方法中, 如果都没有获得资源就调用 **notify** 或 **wait** 的话会报错。