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：**2020 级　　**上机实践成绩：**

指导教师：张召　　**姓名：**张熙翔　　**学号：**10205501427

上机实践名称：Java 多线程编程　　**上机实践日期：**2022/4/10

一、实验目的

- 熟悉 Java 多线程编程
- 熟悉并掌握线程创建、线程控制
- 熟悉并掌握线程同步、线程交互

二、实验任务

- 使用常用的两种方式创建线程
- 使用 join()、yield()等方法对线程进行控制
- 学习使用 synchronized 关键字进行线程同步
- 学习使用 wait()和 notify()方法进行线程交互

三、使用环境

- Intellij IDEA 2020.3.2
- JDK 11.0.6

四、实验过程

Week6_task1:

改写类中的 run()方法，将每个线程的 ID 也打印出来。

代码：

```
package Multithreading;

public class ThreadTest01 extends Thread{
    public void run() {
        System.out.println(Thread.currentThread().getName()+ " 线程 ID 是:
"+Thread.currentThread().getId());}
    public static void main(String[] args){
        ThreadTest01 mthread1 = new ThreadTest01();
        ThreadTest01 mthread2 = new ThreadTest01();
        ThreadTest01 mthread3 = new ThreadTest01();
        mthread1.start();
        mthread2.start();
        mthread3.start();
    }
}
```

运行结果：

```
Thread-2 线程ID是：18
Thread-1 线程ID是：17
Thread-0 线程ID是：16
```

Week6_task2:

通过实现 Runnable 接口的方式编写两个线程，一个线程负责打印字母，另一个线程负责打印数字，两个线程同时进行打印，要求打印出来的结果形式为 a1b23c456d7891…z…（数字 1-9 循环 2 次）。

代码：

```
package Multithreading;

class ThreadTest02 implements Runnable{
    public void run() {
        synchronized (ThreadTest02.class){
            int num=1;
            int cnt=0;
            for (int i = 1; i < 27; i++) {
                if (Thread.currentThread().getName().equals("打印字母")) {
                    System.out.print((char)('a'+i-1));
                    ThreadTest02.class.notifyAll();
                    try {
                        ThreadTest02.class.wait(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                } else {
                    for(int j=0;j<i;j++){
                        if(cnt==2){
                            break;
                        }
                        System.out.print(num);
                        num++;
                        if(num==10){
                            num=1;
                            cnt++;
                        }
                    }
                    ThreadTest02.class.notifyAll();
                    try {
                        ThreadTest02.class.wait(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
```

```
    public static void main(String[] args) {
        ThreadTest02 mt = new ThreadTest02();
        Thread t1 = new Thread(mt,"打印字母");
        Thread t2 = new Thread(mt,"打印数字");
        t1.start();
        t2.start();
    }
}
```

运行结果：

```
1a23b456c7891d23456e789fghijklmnopqrstuvwxyz
进程已结束,退出代码0
```

Week6_task3:

完善代码，用 join 方法实现正常的逻辑。

代码：

```
package Multithreading;

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");

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

运行结果：

```
上课铃响
老师上课
下课铃响
老师下课

进程已结束,退出代码0
```

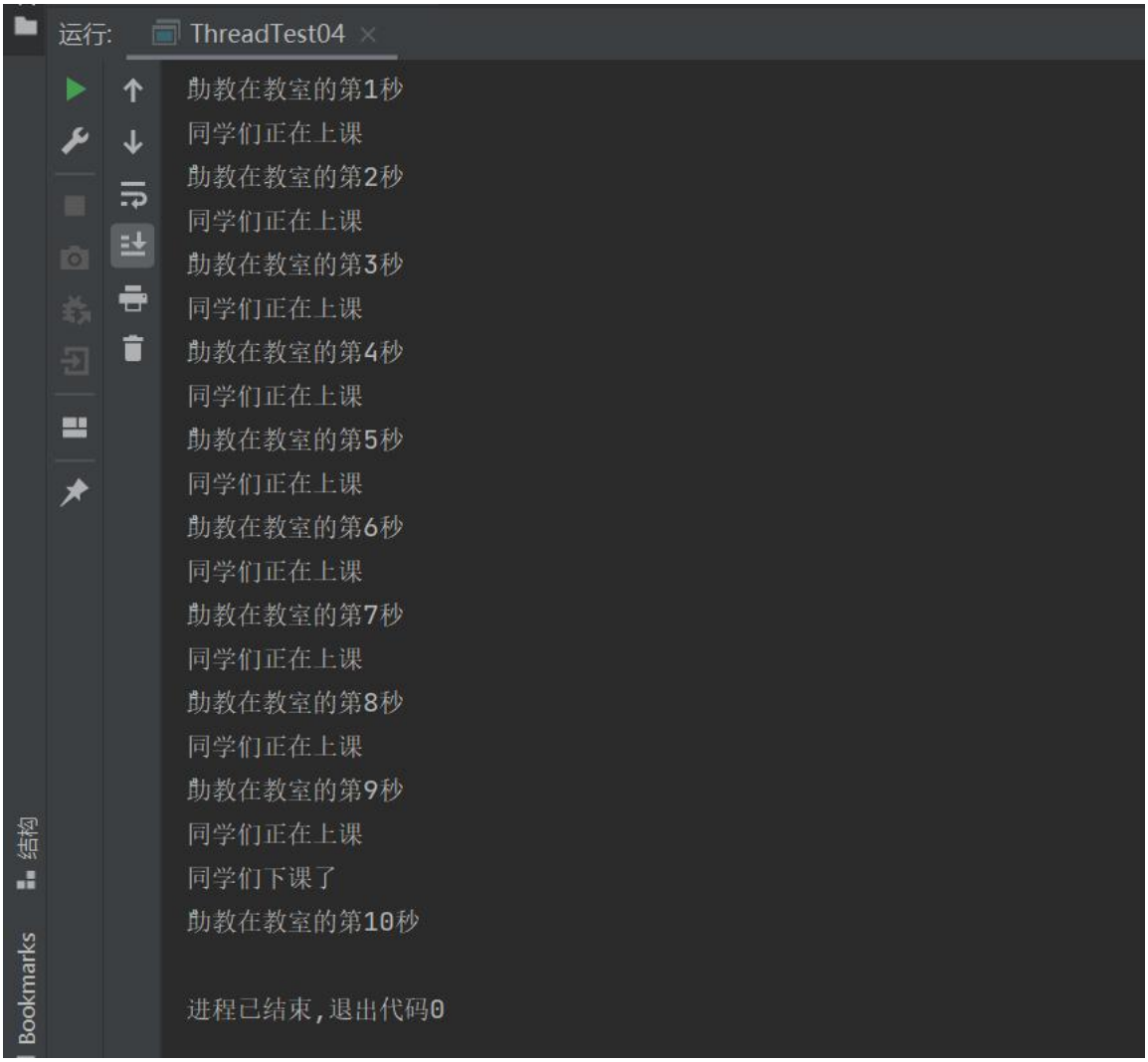
Week6_task4:

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。

代码：

```
package Multithreading;

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{
        ThreadTest04 inClassroom = new ThreadTest04();
        Thread thread = new Thread(inClassroom,"👨‍🏫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");
            }
        }}
}
```

运行结果：



Week6_task5:

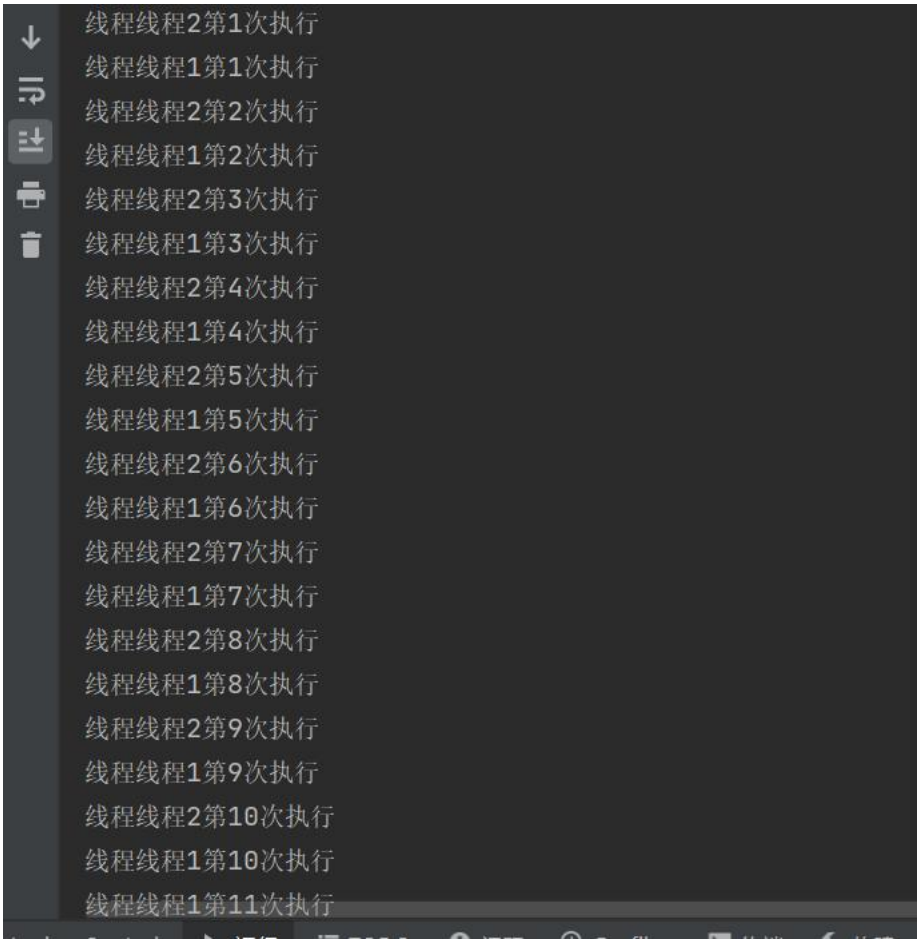
完善代码，将两个线程设置为不同的优先级，并将第一个线程设置为让步状态。总结线程让步的特点。

代码：

```
package Multithreading;

class MyRunnable01 implements Runnable{
    @Override
    public void run(){
        for(int i = 1; i < 100; i++){
            System.out.println("线程"+ Thread.currentThread().getName()
                    +"第" + i + "次执行");
            Thread.yield();
        }}
}
```

```java
class MyRunnable02 implements Runnable{
    @Override
    public void run(){
        for(int i = 1; i < 100; i++){
            System.out.println("线程"+ Thread.currentThread().getName()
                    +"第" + i + "次执行");
        }}
}

public class ThreadTest05 {
    public static void main(String[] args) throws InterruptedException{
        Thread t1 = new Thread(new MyRunnable01(),"线程 1");
        Thread t2 = new Thread(new MyRunnable02(),"线程 2");
        t1.setPriority(10);
        t2.setPriority(1);
        t1.start();
        t2.start();
    }
}
```

运行结果：



线程让步的特点：

线程让步是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。

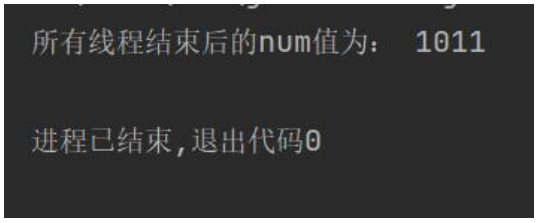
因此，使用 yield()的目的是让相同优先级的线程之间能适当的轮转执行。

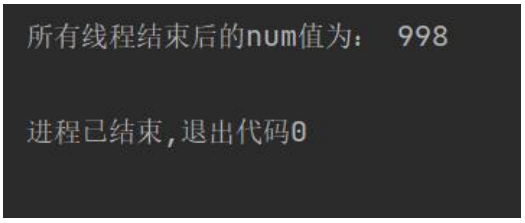
但是，实际中无法保证 yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。

即 yield()从未导致线程转到等待/睡眠/阻塞状态。在大多数情况下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。

Week7_task1:

运行测试类中的 main 函数，将 num 可能得到的两种不同的值截图。





Week7_task2:

换一种写法创建线程。

代码：

```
package Multithreading;

import static java.lang.Thread.sleep;

public class PlusMinus {
    public volatile int num;
    public void plusOne(){
        num = num + 1;
    }
    public void minusOne(){
        num = num - 1;
    }
    public int printNum(){
        return num;
    }}
class TestPlusMinus {
    public static void main(String[] args){
        PlusMinus plusMinus = new PlusMinus();
        plusMinus.num = 1000;
        int threadNum = 1000;
        Thread[] plusThreads = new Thread[threadNum];
        Thread[] minusThreads = new Thread[threadNum];
        Thread thread1 = null;
        Thread thread2 = null;
        for(int i = 0; i < threadNum; i++){
            thread1 = new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                plusMinus.plusOne();
            }},"加法" );
            thread2 = new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    plusMinus.minusOne();
                }},"减法" );
            thread1.start();
            thread2.start();
            plusThreads[i] = thread1;
            minusThreads[i] = thread2;
        }
        for(Thread thread:plusThreads){
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }}
        for(Thread thread:minusThreads){try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }}
        System.out.println("所有线程结束后的 num 值为:  " +
plusMinus.printNum());
    }}
```

Week7_task3:

使用 synchronized 关键字修改 TestPlusMinus 测试类，使得 num 值不出现不同步的问题。

代码：

```
package Multithreading;

public class PlusMinus {
    public volatile int num;
    public void plusOne(){
        num = num + 1;
```

```
    }
    public void minusOne(){
        num = num - 1;
    }
    public int printNum(){
        return num;
    }}
class TestPlusMinus {
    public static void main(String[] args){
        PlusMinus plusMinus = new PlusMinus();
        plusMinus.num = 1000;
        int threadNum = 1000;
        Thread[] plusThreads = new Thread[threadNum];
        Thread[] minusThreads = new Thread[threadNum];
        for(int i = 0; i < threadNum; i++){
            Thread thread1 = new Thread(){
                @Override
                public void run(){
                    try {
                        sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    synchronized (PlusMinus.class){
                    plusMinus.plusOne();}
                }};
            Thread thread2 = new Thread(){
                @Override
                public void run(){
                    try {
                        sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    synchronized (PlusMinus.class){
                    plusMinus.minusOne();}
                }};
            thread1.start();
            thread2.start();
            plusThreads[i] = thread1;
            minusThreads[i] = thread2;
        }
        for(Thread thread:plusThreads){
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }}
        for(Thread thread:minusThreads){try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }}
        System.out.println("所有线程结束后的 num 值为:  " +
plusMinus.printNum());
    }}
```

Week7_task4:

不修改 TestPlusMinus 测试类，使用 synchronized 关键字修改 PlusMinus 基础类，使得 num 值不出现不同步的问题。

代码：

```
package Multithreading;

public class PlusMinus {
    public volatile int num;
    public synchronized void plusOne(){
        num = num + 1;
    }
    public synchronized void minusOne(){
        num = num - 1;
    }
    public synchronized int printNum(){
        return num;}
    }
class TestPlusMinus {
    public static void main(String[] args){
        PlusMinus plusMinus = new PlusMinus();
        plusMinus.num = 1000;
        int threadNum = 1000;
        Thread[] plusThreads = new Thread[threadNum];
        Thread[] minusThreads = new Thread[threadNum];
        for(int i = 0; i < threadNum; i++){
            Thread thread1 = new Thread(){
                @Override
                public void run(){
                    try {
                        sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    plusMinus.plusOne();}
                };
            Thread thread2 = new Thread(){
                @Override
                public void run(){
                    try {
                        sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    plusMinus.minusOne();}
                };
            thread1.start();
            thread2.start();
            plusThreads[i] = thread1;
            minusThreads[i] = thread2;
        }
        for(Thread thread:plusThreads){
            try {
                thread.join();
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }}
        for(Thread thread:minusThreads){try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }}
        System.out.println("所有线程结束后的 num 值为： " +
plusMinus.printNum());
    }}
```

Week7_task5:

设计三个线程发生死锁的场景并编写代码。

场景：

多个线程分别占用对方需要的同步资源不放弃，都在等在对方放弃自己所需要的同步资源，程序会停在出现死锁的位置不再执行。

```
thread1 正在占用 plusMinus1
thread2 正在占用 plusMinus2
thread3 正在占用 plusMinus3
thread1 试图继续占用 plusMinus2
thread2 试图继续占用 plusMinus1
thread3 试图继续占用 plusMinus1
```

代码：

```
package Multithreading;

public class DeadLock {
    public static void main(String[] args){
        PlusMinus plusMinus1 = new PlusMinus();
        plusMinus1.num = 1000;
        PlusMinus plusMinus2 = new PlusMinus();
        plusMinus2.num = 1000;
        PlusMinus plusMinus3 = new PlusMinus();
        plusMinus3.num = 1000;

        Thread thread1 = new Thread(){
            @Override
            public void run(){
                synchronized (plusMinus1){
                    System.out.println("thread1 正在占用 plusMinus1");
                    try {
```

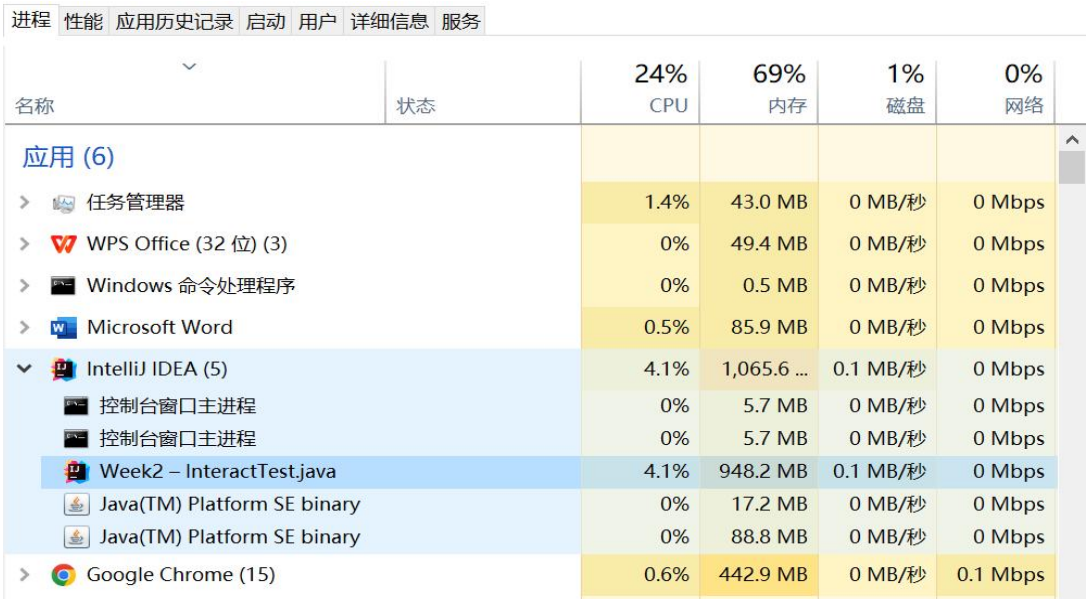
```
                    sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("thread1 试图继续占用 plusMinus2");
                synchronized (plusMinus2){
                    System.out.println("thread1 成功占用 plusMinus2 ");
                }
                System.out.println("thread1 试图继续占用 plusMinus3");
                synchronized (plusMinus3){
                    System.out.println("thread1 成功占用 plusMinus3 ");
                }
            }}
        };
        thread1.start();
        Thread thread2 = new Thread(){@Override
        public void run(){
            synchronized (plusMinus2){
                System.out.println("thread2 正在占用 plusMinus2");
                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("thread2 试图继续占用 plusMinus1");
                synchronized (plusMinus1){
                    System.out.println("thread2 成功占用 plusMinus1 ");
                }
                System.out.println("thread2 plusMinus3");
                synchronized (plusMinus3){
                    System.out.println("thread2 成功占用 plusMinus3 ");
                }
            }}};
        thread2.start();
        Thread thread3 = new Thread(){@Override
        public void run(){
            synchronized (plusMinus3){
                System.out.println("thread3 正在占用 plusMinus3");
                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("thread3 试图继续占用 plusMinus1");
                synchronized (plusMinus1){
                    System.out.println("thread3 成功占用 plusMinus1 ");
                }
                System.out.println("thread3 试图继续占用 plusMinus2");
                synchronized (plusMinus2){
                    System.out.println("thread3 成功占用 plusMinus2 ");
                }
            }}};
        thread3.start();
    }
}
```

Week7_task6:

阐述 synchronized 关键字在实例方法上的作用，运行本代码，观察 CPU 的使用情况。

作用：锁定了整个方法时的内容。在进入实例方法前，线程必须获得当前对象实例的锁。当两个并发线程(thread1 和 thread2)访问同一个对象(plusMinus)中的实例方法时，在同一时刻只能有一个线程得到执行，另一个线程受阻塞。

CPU 使用情况：



Week6_task7:

在 task6 的基础上增加若干减 1 操作线程，运行久一点，观察有没有发生错误。若有，请分析错误原因，并给出解决方法。

有错误。错误：num 会出现负数。因为存在多个减一线程，假设 n 个减一线程在同一时刻读取 num 值，而此时 num 值小于 n，则进程结束后 num 会为负数。

解决方法：对于 n 个减法线程，其中 n-1 个线程在 num==1 成立后直接 break；另一个减法线程在 num==1 后将 num 重新赋值 n，continue 循环。

代码：

```
Thread thread1 = new Thread(){
        @Override
        public void run(){
            while (true){
                if(plusMinus.num <= 1){
                    plusMinus.num = 2;
                    continue;
                    //break;
                }
                plusMinus.minusOne();
```

```
                try {
                    sleep(10);//减法线程 10ms
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }}
        }};
    thread1.start();
```

```
Thread thread3 = new Thread(){
            @Override
            public void run(){
                while (true){
                    if(plusMinus.num <= 1){
                        //continue;
                        break;
                    }
                    plusMinus.minusOne();
                    try {
                        sleep(10);//减法线程 10ms
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }}
            }};
        thread3.start();
```

Week7_task8:

将 PlusMinus02 类中的 volatile 关键字删除，观察实验结果变化，将实验结果附在实验报告中并分析 volatile 关键字作用。

```
num = 11
num = 10
num = 9
num = 8
num = 9
num = 8
num = 7
num = 6
num = 5
num = 4
num = 3
num = 4
num = 3
num = 2
num = 1
num = 2
num = 3
num = 4
num = 5
num = 6
```

无 volatile 关键字

```
num = 11
num = 10
num = 11
num = 10
num = 9
num = 8
num = 7
num = 6
num = 5
num = 4
num = 5
num = 4
num = 3
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
```

有 volatile 关键字

一个线程对一个 volatile 变量的修改，对于其它线程来说是可见的，即线程每次获取 volatile 变量的值都是最新的。当一个变量被 volatile 修饰时，任何线程对它的写操作都会立即刷新到主内存中，并且会强制让缓存了该变量的线程中的数据清空，必须从主内存重新读取最新数据。

Week7_task9:

使用 wait 和 notify 修改下面的代码段，使之也达到 4.2 开头中的功能，并将修改后的完整代码段和实验结果以及 CPU 占用情况截图附在实验报告中。

代码：

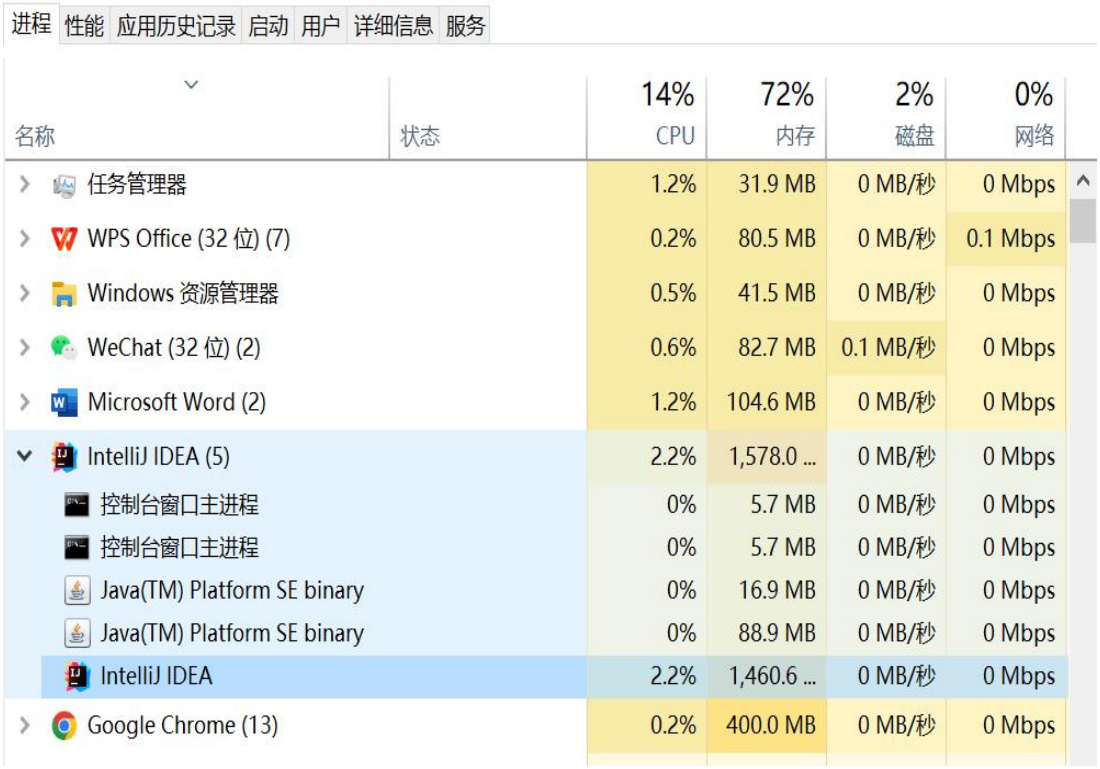
```
package Multithreading;

class PlusMinus02 {
    public volatile int num;
    public void plusOne(){
        synchronized (this){
            num = num + 1;
            printNum();
        }}
    public void minusOne() {
        synchronized (this) {
            num = num - 1;
            printNum();
        }}
    public void printNum(){
        System.out.println("num = " + num);
    }
}

public class InteractTest {
    public static void main(String[] args){
        PlusMinus02 plusMinus = new PlusMinus02();
        plusMinus.num = 1000;
        Object obj= new Object();
        Thread thread1 = new Thread(){
            @Override
            public void run(){
                while (true){
                    synchronized(obj){
                    plusMinus.minusOne();
                    if(plusMinus.num==1){
                        try {
                            obj.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    }
                    try {
                        sleep(10);
                    } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }
            }}};
        thread1.start();
        Thread thread2 = new Thread(){
            @Override
            public void run(){
                while (true){
                    synchronized(obj){
                        plusMinus.plusOne();
                        obj.notifyAll();
                    }
                    try { sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }}};
        thread2.start();
    }
}
```

运行结果：

```
num = 10
num = 11
num = 10
num = 9
num = 8
num = 7
num = 6
num = 5
num = 6
num = 5
num = 4
num = 3
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
```

CPU 使用情况：



Week7_task10:

编写多线程程序，模拟车站三个窗口同时卖票，包括售票（可能存在购买多张的情况），退票（可能存在退多张的情况）和新进票，要求有余票时必须出售，无票时不能出售，购票时若无足量余票可选择继续等待或离开。

代码：

```
package Multithreading;
import static java.lang.Thread.sleep;
import java.util.Scanner;

class MyRunner2 implements Runnable{
    Object obj= new Object();
    Scanner s=new Scanner(System.in);
    public int max=10;
    @Override
    public void run() {
        // TODO Auto-generated method stub
        while(true){
            synchronized(obj){
                System.out.println("请输入操作序号：1.购票 2.退票 3.新进票");
                int doingwhat=s.nextInt();
                System.out.println("请输入操作票数");
                int ticket_num=s.nextInt();
                if(doingwhat==1){//购票
                    if(ticket_num<=max){
```

```
                    System.out.println("窗口
"+Thread.currentThread().getName()+"出售"+ticket_num+"张电影票");
                    this.max=this.max-ticket_num;
                }
                else {
                    System.out.println("票量只有"+max+"张，请等待");
                    try {
                        obj.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
            if(doingwhat==2) {//退票
                this.max=this.max+ticket_num;
                try {
                    sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                obj.notifyAll();
            }
            if(doingwhat==3) {//退票
                this.max=this.max+ticket_num;
                try {
                    sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
               obj.notifyAll();
            }
        }
        try {
            sleep(10);
        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
    }
}
}

public class Ticket{
    public static void main(String[] args) {
        MyRunner2 r=new MyRunner2();
        Thread t1=new Thread(r,"1");
        Thread t2=new Thread(r,"2");
        Thread t3=new Thread(r,"3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

运行结果：

```
请输入操作序号：1.购票 2.退票 3.新进票
1
请输入操作票数
10
窗口1出售10张电影票
请输入操作序号：1.购票 2.退票 3.新进票
1
请输入操作票数
1
票量只有0张，请等待
请输入操作序号：1.购票 2.退票 3.新进票
2
请输入操作票数
3
请输入操作序号：1.购票 2.退票 3.新进票
1
请输入操作票数
1
窗口1出售1张电影票
```