

## 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 2

上机实践日期：2023.3.31

上机实践编号：5

组号：

上机实践时间：9:50

### 一、实验目的

熟悉 Java 多线程编程

熟悉并掌握线程同步和线程交互

### 二、实验任务

学习使用synchronized 关键字

学习使用wait() 、 notify() 、 notifyAll() 方法进行线程交互

### 三、使用环境

IntelliJ IDEA

JDK 版本: Java 19

### 四、实验过程

#### 1.线程同步

##### 1.1synchronized 关键字

Task1: 对 Lab4 的 3.2 中给出的 PlusMinus 、 TestPlus 、 Plus 代码，使用synchronized 关键字进行修改，使用两种不同的修改方式，使得 num 值不出现线程处理不同步的问题，将实现代码段及运行结果附在实验报告中。

锁对象方法：

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

修饰整个方法，同步方法：

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

代码运行结果:

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

进程已结束,退出代码0
```

Task2: 给出以下 TestMax 、 MyThread 、 Res 代码，使用synchronized 关键字在 TODO 处进行修改，实现最后打印出的 res.max\_idx 的值是所有 MyThread 对象的 list 中保存的数的最大值，将实现代码段及运行结果附在实验报告中。

在进程运行过程中，使用 synchronized 修饰三个线程公用的 res 对象，这样，循环 100 个数组元素的过程将顺序进行，打印出的 index 会从 0 排到 100，重复 3 次。

```
public void run() {
    synchronized (res){
        for(int i=0;i<100;i++){
            if(res.max_idx<list.get(i)){
                try {
                    Thread.sleep(20);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                res.max_idx= list.get(i);
            }
            System.out.println(i);
        }
    }
}
```

同时也可以把 synchronized 加在 if 代码块外部，可以保证 if 比较和赋值两个操作顺序进行不被其他线程打断。

```
public void run() {
    for(int i=0;i<100;i++){
        synchronized (res){
            if(res.max_idx<list.get(i)){
                try {
                    Thread.sleep(20);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                res.max_idx= list.get(i);
            }
            System.out.println(i);
        }
    }
}
```

此外，如果将 synchronized 修饰符中的对象写为当前对象 this，不会起到效果，因为此时加

锁的对象是 3 个进程，而这三个进程没有试图互相占用的行为，所以比较操作和赋值操作仍然会被打断。

代码运行结果：

```
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe"
9951
|
进程已结束,退出代码0
```

## 1.2 死锁

Task3: 设计 3 个线程彼此死锁的场景并编写代码(可基于上述代码或自己编写)，将实现代码段及运行结果附在实验报告中。

修改 MyThread 的构造方法，加入参数 PlusMinus3。

```
MyThread2(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;
```

在 main 函数中定义 plusMinus3，开启第 3 个线程，id 为 3，每个线程使用 3 个资源。

```
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;
    MyThread2 thread1 = new MyThread2(plusMinus1, plusMinus2, plusMinus3, _tid: 1);
    MyThread2 thread2 = new MyThread2(plusMinus1, plusMinus2, plusMinus3, _tid: 2);
    MyThread2 thread3 = new MyThread2(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();
    t2.join();
    t3.join();
}
```

线程 1 占用 pm2，线程 2 占用 pm3，线程 3 占用 pm1，随后再分别请求占用 pm3，pm1，pm2，由于此时线程 1 在要求线程 2 的资源，线程 2 在要求线程 3 的资源，线程 3 在要求线程 1 的资源，因此形成等待回路，实现了线程锁死。

```

else if(tid==3){
    synchronized (pm1){
        System.out.println("thread" + tid + "正在占用 plusMinus1");
        try {
            Thread.sleep( millis: 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");
        }
    }
}

```

程序运行结果：

```

"C:\Program Files (x86)\Java\jdk-19\bin\java.exe
thread1正在占用 plusMinus2
thread3正在占用 plusMinus1
thread2正在占用 plusMinus3
thread3试图继续占用 plusMinus2
thread3等待中...
thread1试图继续占用 plusMinus3
thread2试图继续占用 plusMinus1
thread1等待中...
thread2等待中...

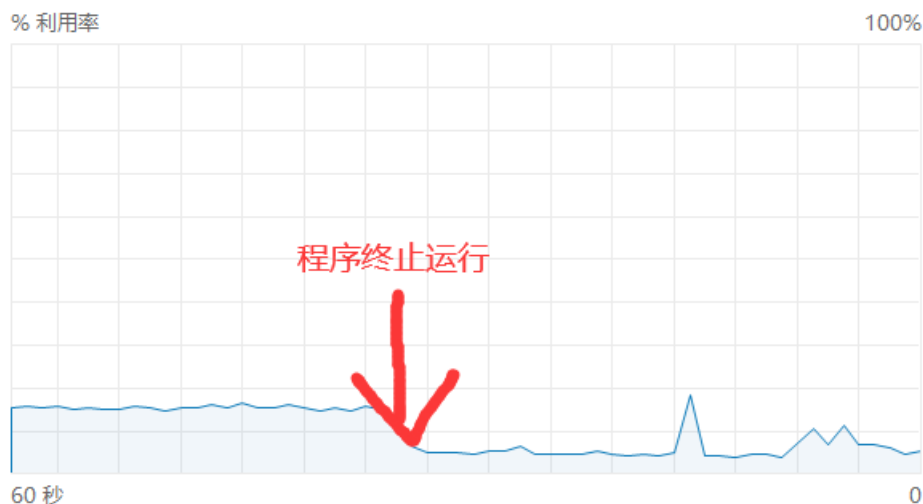
```

## 2.线程交互

Task4: 首先阐述 synchronized 在实例方法上的作用，然后运行本代码段，同时打开检测 cpu 的工具，观察 cpu 的使用情况，将实验结果和 cpu 使用情况截图附在实验报告中。

Synchronized 在此处保证了实例方法 plusOne 和 minusOne 都在获取到当前对象后再执行，而且指令是连续执行，具有原子性，不被其他线程打断。如果没有 Synchronized，那么可能在 num-1 之后，又被另一线程+1，此时 print 出的 num 值不变，无法体现线程对 num 的操作；另外+与-操作也不是原子性，包含取值、对值+1 或-1、将新值写入到磁盘里三个步骤，所以也可能被打断。

# CPU 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz



利用率	速度	基准速度:	2.30 GHz	
5%	3.22 GHz	插槽:	1	
进程	线程	句柄	内核:	8
281	4676	142571	逻辑处理器:	16
			虚拟化:	已启用
正常运行时间			L1 缓存:	640 KB
7:01:36:23			L2 缓存:	10.0 MB
			L3 缓存:	24.0 MB

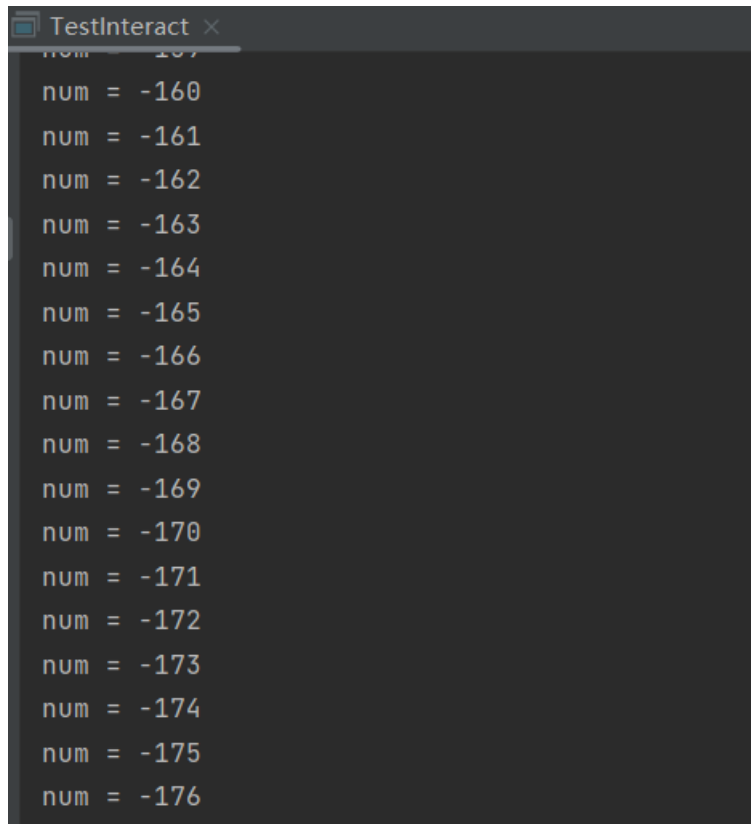
可以看到 cpu 使用率显著增加了，线程数量也增加了  
程序运行结果：

```
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
```

可以看到，线程 1 给 num-1 的同时线程 2 给 num+1，而如果 num 的值=1，此时线程 1 会进入 while 循环，等待线程 2 给 num+1 后再执行-1 操作。这使得 num 的值始终在 1 和 2 之间徘徊。

Task5: 在 Task4 基础上增加若干减一操作线程，运行久一点，观察有没有发生错误。若有，请分析错误原因，给出解决代码。

加入了线程 t3，行为与 t1 相同，给 num-1，此时会发生 num 输出负数的情况。这是因为当 num 的值为 1 时，t3 和 t1 执行，会卡在 while 循环中；随后 t2 再执行，给 num+1，此时 t1 和 t3 同时从 while 循环里放出来，给 num 进行了两次-1 操作，就导致 num 从 2 变到了 0，此时 while 循环就再也不能约束 t1 和 t3 的-1 操作了，程序输出的负数绝对值越来越大：



```
TestInteract x
num = -160
num = -161
num = -162
num = -163
num = -164
num = -165
num = -166
num = -167
num = -168
num = -169
num = -170
num = -171
num = -172
num = -173
num = -174
num = -175
num = -176
```

起初，试图把 while 循环改为 if 判断操作：



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

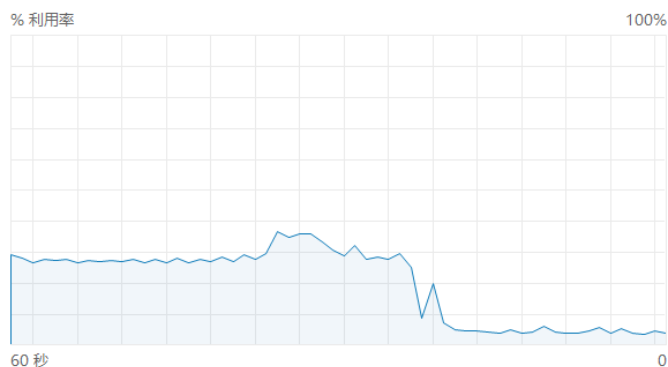
但是这无法排出数个-1 线程同时运行到 if 判断处，然后 t2 给 num+1，这数个-1 线程可能会一个个通过条件判断，然后再依次-1，就导致 num 变为负数。所以只能使判断与-1 操作一气呵成，具有原子性：

```
for(int i=0;i<10;i++){
    ta[i]=new Thread(){
        public void run() {
            while (true) {
                synchronized (pmo) {
                    if (pmo.num == 1) {
                        continue;
                    }
                    pmo.minusOne();
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    };
}
```

这次加入 10 个线程也不会减到负数了，因为 `synchronized` 保证了-1 线程的条件判断和-1 操作之间不会被打断，杜绝了给值为 1 的 `num` 再-1 的错误。

可以看到这种解决方法，占用了较大的 `cpu` 使用率。

## CPU 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz



利用率	速度	基准速度:	2.30 GHz
4%	2.26 GHz	插槽:	1
进程	线程	内核:	8
279	4705	逻辑处理器:	16
正常运行时间	句柄	虚拟化:	已启用
7:03:23:07	141185	L1 缓存:	640 KB
		L2 缓存:	10.0 MB
		L3 缓存:	24.0 MB

### 2.1 使用 `wait` 和 `notify`

Task6: 在以下代码中加入若干获取 `product` 的线程，并运行截图；之后将

`while(productQueue.isEmpty())`修改为 `if (productQueue.isEmpty())`，并观察运行结果，如发生

错误，试分析原因。

加入了大小为 10 的获取 product 的线程组，在获取 product 时输出自己的序号：

```
Thread[] ta = new Thread[10];
for(int i=0;i<10;i++){
    int finalI = i;
    ta[i]=new Thread(){
        public void run() {
            while (true) {
                try {
                    String s = pf.getProduct();
                    System.out.println("t1-"+ finalI + " get product: " + s);
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    };
}
```

代码运行结果，发现只有第一个创建的线程能得到 product：

```
t1 get product: product
t2 add product: product
t1-0 get product: product
t2 add product: product
t1 get product: product
t2 add product: product
t1-0 get product: product
t2 add product: product
t1 get product: product
t2 add product: product
t1-0 get product: product
t2 add product: product
t1 get product: product
```

将 while 改成 if，发生了如下错误：

```
Test (1) x
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe" "-javaagent:D:\Program Files\J
Exception in thread "Thread-0" Exception in thread "Thread-10" Exception in thre
    at java.base/java.util.LinkedList.removeFirst(LinkedList.java:274)
    at java.base/java.util.LinkedList.remove(LinkedList.java:689)
    at ProductFactory.getProduct(Test.java:69)
    at Test$2.run(Test.java:26)
Exception in thread "Thread-7" java.util.NoSuchElementException Create breakpoint
    at java.base/java.util.LinkedList.removeFirst(LinkedList.java:274)
    at java.base/java.util.LinkedList.remove(LinkedList.java:689)
    at ProductFactory.getProduct(Test.java:69)
    at Test$2.run(Test.java:26)
Exception in thread "Thread-6" java.util.NoSuchElementException Create breakpoint
    at java.base/java.util.LinkedList.removeFirst(LinkedList.java:274)
    at java.base/java.util.LinkedList.remove(LinkedList.java:689)
    at ProductFactory.getProduct(Test.java:69)
    at Test$2.run(Test.java:26)
```



这是因为如果换成 if，在加入了新的 product 之后，之前 wait 阻塞的线程被 notify 了，但是不再进行条件判断，全部都去执行下一行取走 product 的指令，导致数组越界。如果换回 while 循环的话，每次线程被唤醒都会再进行一次条件判断，如果数组空了就继续 wait，因此不会出错。

**Bonus Task1 (optional):** 可以修改以下代码逻辑，试说明如果不使用 notifyAll() 而是使用 notify()，在哪些情况下可能出错？

在调用 notify 时，只有一个等待线程会被唤醒而且它不能保证哪个线程会被唤醒，这取决于线程调度器。而在调用 notifyAll 方法时，等待该锁的所有线程都会被唤醒，但在执行剩余代码之前，所有被唤醒的线程在 while 循环中争夺锁定，如果没有争夺成功，线程则会重置等待条件。

在 while 改成 if 的情况下，如果使用 notify，会发生如下错误：

```

Test (1) x
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe" "-javaagent:D:\Program Files
线程32进入等待
线程41进入等待
线程40进入等待
线程39进入等待
线程38进入等待
t2 add product: product
t1 get product: product
线程37进入等待
线程36进入等待
线程35进入等待
线程34进入等待
线程33进入等待
Exception in thread "Thread-1" java.util.NoSuchElementException Create breakpoint
  at java.base/java.util.LinkedList.removeFirst(LinkedList.java:274)
  at java.base/java.util.LinkedList.remove(LinkedList.java:689)
  at ProductFactory.getProduct(Test.java:73)
  at Test$2.run(Test.java:26)
线程31进入等待
t2 add product: product
t1-9 get product: product
  
```

在 wait 之前我让每个线程打印出了它们的编号，可以看到根据编号，报错的线程是数组中第 1 个，也就是线程 32。因此，在 t2 加入 product 时唤醒的线程是 32，但是此时还有许多线程没有进入 wait 状态，于是 t1 取走了 product，但是 32 不会再一次进行条件判断，于是就再一次取走 product，导致了数组越界的错误。在所有线程都进入等待状态后，product 增加时线程调度器轮流唤醒每一个线程，线程之间不会发生同时被唤醒的情况，因此不再发生错误。

综上所述，发生错误的条件是 1.线程还未全部进入等待状态 2.有别的线程在 product 产生时被唤醒的线程之前夺取到了 product。

**Task7:** 在 Task5 的基础上，使用 wait 和 notify 修改代码，达到一致的代码逻辑，同时打开检测 cpu 的工具，观察 cpu 的使用情况，将实验结果和 cpu 使用情况截图附在实验报告中。

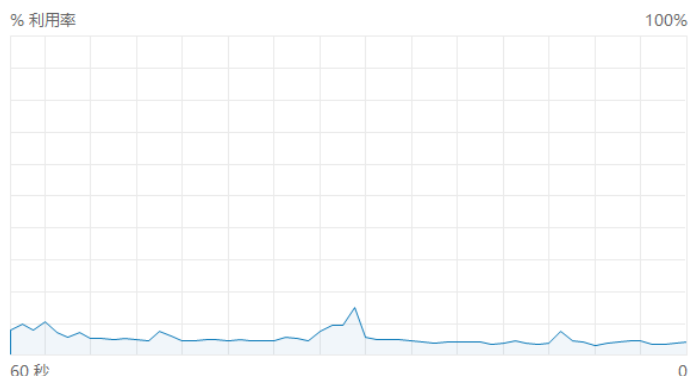
```
class PlusMinusOne {
    volatile int num;
    public void plusOne() {
        synchronized (this) {
            this.num = this.num + 1;
            printNum();
            this.notifyAll();
        }
    }
    public void minusOne() throws InterruptedException {
        synchronized (this) {
            while (num==1) {
                this.wait();
            }
            this.num = this.num - 1;
            printNum();
        }
    }
}
```

与 task5 等同，在 num+1 时提示所有线程开始；而在-1 之前判断是否=1，如果不能-1 则让线程 wait。输出结果如下图：

```
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
```

而且，使用 notify+wait 进行线程通信，可以减少很多 cpu 占用率，代码运行时与不运行时几乎看不出区别。下图中尖锐处为代码停止运行时。

## CPU 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz



利用率	速度	基准速度:	2.30 GHz
4%	1.11 GHz	插槽:	1
进程	线程	内核:	8
284	4852	逻辑处理器:	16
句柄		虚拟化:	已启用
144383		L1 缓存:	640 KB
正常运行时间		L2 缓存:	10.0 MB
7:05:27:12		L3 缓存:	24.0 MB

### 五、总结

Java 的多线程编程中，线程同步是很重要的问题。在线程中建立起执行顺序的串行关系，可以防止连续做同一件事的几条指令被其他线程打断，而线程互斥可以防止线程取用正在被其他线程占用的公共资源。

线程同步通过 `synchronized` 关键字来实现，有两种方式，可以加在方法上，表示这个方法以原子性的性质执行里面的几条指令；也可以在括号里指出公用的资源名称，这可以利用线程互斥的原理，占用相应的资源，让其他想要取用该资源的线程强制进入等待状态。

线程死锁是在每个线程占有一部分资源时，取用其他线程正在占用着的资源，当形成等待链的时候，就会发生死锁现象，此时每个线程都在等待另一个线程的执行结果，而等待环导致这些线程都没法继续执行下去。

在多个线程对同一公共资源进行操作的时候，如果使用 `synchronized` 锁住每一个操作的代码块，会消耗很多 `cpu` 资源，此时可以使用线程交互的 `wait` 和 `notify` 方法，使不满足执行条件的线程进入 `wait` 状态，直到满足这个条件时通知它们。