

华东师范大学数据科学与工程学院上机实验报告

课程名称：计算机网络与编程	年级：2022 级	上机实践成绩：
指导教师：张召	姓名：李芳	学号：10214602404
上机实践名称：Java 多线程编程		上机实践日期：2024.03.22
上机实践编号：04	组号：	上机实践时间：

一、Java 多线程编程_1 题目要求及功能实现情况

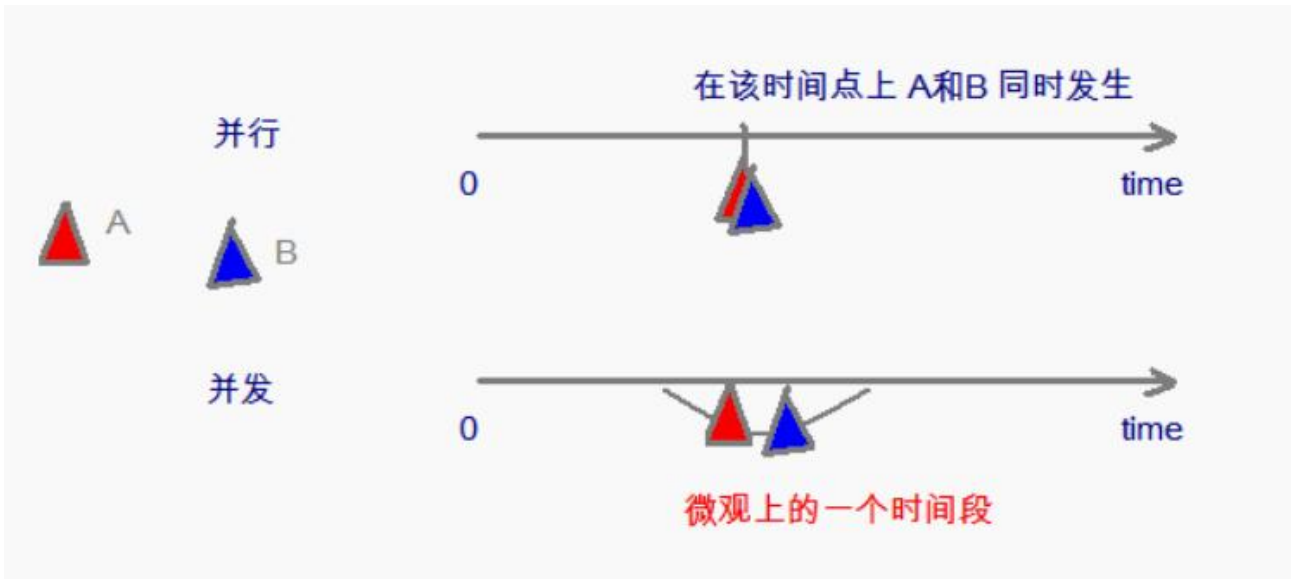
Task1: 改写类中的 run() 方法，将每个线程的 ID 也打印出来，代码及运行结果写到实验报告中。

储备知识：

一、并发与并行

- **并发**：指两个或多个事件在**同一个时间段内**发生。
- **并行**：指两个或多个事件在**同一时刻**发生（同时发生）。
- **注意事项**：并发指的是在一段时间内宏观上有多个程序同时运行。这在单 CPU 系统中，每一时刻只能有一道程序执行，即微观上这些程序是分时的交替运行，只不过是给人的感觉是同时运行，那是因为分时交替运行的时间是非常短的；在多个 CPU 系统中，则这些可以并发执行的程序便可以分配到多个处理器上（CPU），实现多任务并行执行，即利用每个处理器来处理一个可以并发执行的程序，这样多个程序便可以同时执行。

图解：



二、线程与进程

- **进程**：是指一个内存中运行的应用程序，每个进程都有一个独立的内存空间，一个应用程序可以同时运行多个进程；进程也是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序即是一个进程从创建、运行到消亡的过程。
- **线程**：线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。
- **一个程序运行后至少有一个进程，一个进程中可以包含多个线程。**

操作：

- **查看电脑当前任务进程：**

任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

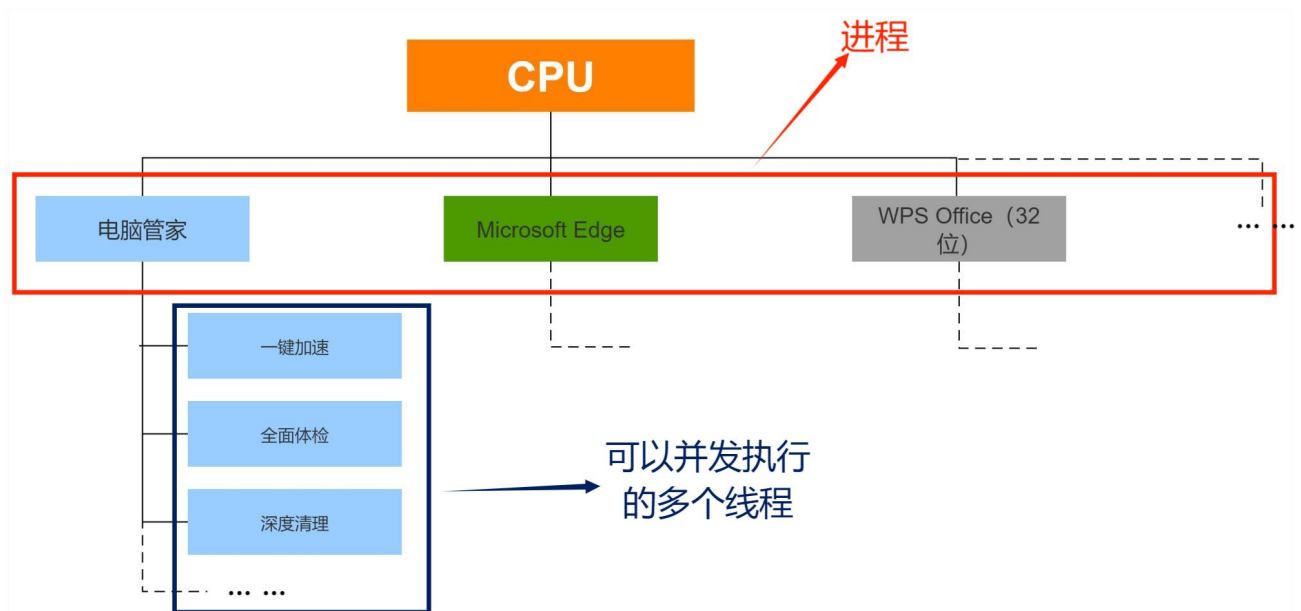
名称	状态	CPU	内存	磁盘	网络
应用 (5)					
Microsoft Edge (20)		0.2%	752.9 MB	0 MB/秒	0 Mbps
WPS Office (32 位) (7)		1.2%	323.9 MB	0 MB/秒	0 Mbps
电脑管家 (7)		3.2%	179.6 MB	0.1 MB/秒	0 Mbps
截图和草图 (2)		0.6%	15.8 MB	0 MB/秒	0 Mbps
任务管理器		0.6%	28.1 MB	0 MB/秒	0 Mbps
后台进程 (122)					
AggregatorHost		0%	0.4 MB	0 MB/秒	0 Mbps
Alibaba PC Safe Service (32 位)		0%	11.4 MB	0 MB/秒	0 Mbps
Antimalware Service Executa...		1.8%	170.5 MB	0.1 MB/秒	0 Mbps
Application Frame Host		2.2%	10.1 MB	0 MB/秒	0 Mbps
BasicService		0%	4.9 MB	0 MB/秒	0 Mbps
COM Surrogate		0%	0.1 MB	0 MB/秒	0 Mbps

简略信息(D) 结束任务(E)

- **查看某一程序线程：**



CPU、进程、线程关系示例图：



三、线程调度

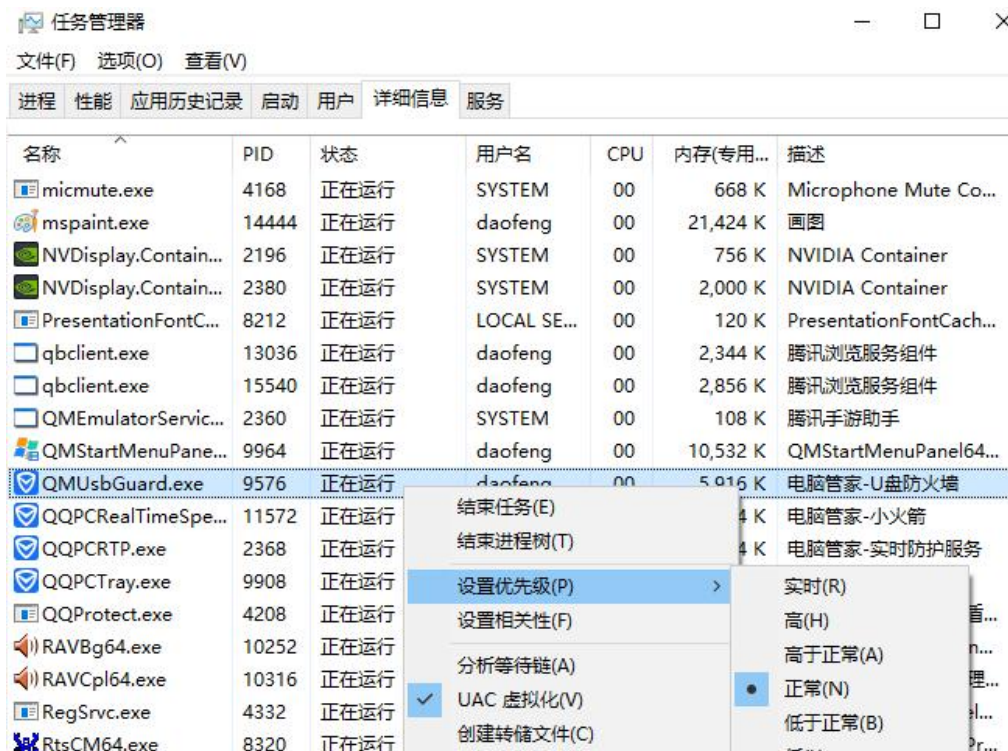
- **分时调度：**所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间。
- **抢占式调度：**优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机选择一个(线程随机性)，Java 使用的是**抢占式调度**。大部分操作系统都支持多进程并发运行，现在的操作系统几乎都支持

同时运行多个程序。实际上，CPU(中央处理器)使用抢占式调度模式在多个线程间进行着高速的切换。对于 CPU 的一个核而言，某个时刻，只能执行一个线程，而 CPU 的在多个线程间切换速度相对我们的感觉要快，看上去就是在同一时刻运行。

- 多线程程序并不能提高程序的运行速度，但能够提高程序运行效率，让 CPU 的使用率更高。

操作：

- 设置线程的优先级：



四、Thread 类

Java 使用 `java.lang.Thread` 类代表线程，所有的线程对象都必须是 `Thread` 类或其子类的实例。API 中该类中定义了有关线程的一些方法，具体如下：

- 构造方法：

- `public Thread()` : 分配一个新的线程对象。
- `public Thread(String name)` : 分配一个指定名字的新的线程对象。
- `public Thread(Runnable target)` : 分配一个带有指定目标新的线程对象。
- `public Thread(Runnable target, String name)` : 分配一个带有指定目标新的线程对象并指定名字。

- 常用方法：

- `public String getName()` : 获取当前线程名称。

- `public void start()` :导致此线程开始执行; Java 虚拟机调用此线程的 `run` 方法。
- `public void run()` :此线程要执行的任务在此处定义代码。
- `public static void sleep(long millis)` :使当前正在执行的线程以指定的毫秒数暂停（暂时停止执行）。
- `public static Thread currentThread()` :返回对当前正在执行的线程对象的引用。

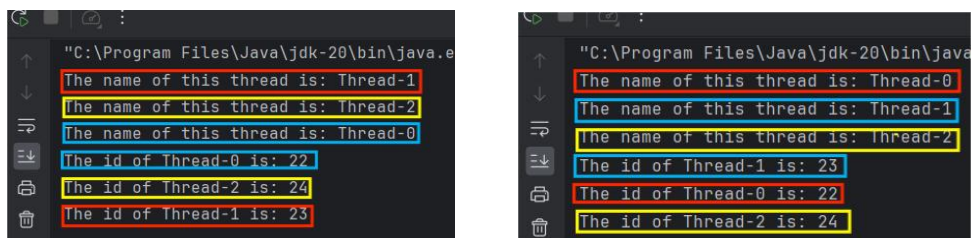
五、创建线程类方法一-----继承 `Thread`

Java 中通过继承 `Thread` 类来**创建并启动多线程**，步骤如下：

1. 定义 `Thread` 类的子类，并重写该类的 `run()`方法，该 `run()`方法的方法体就代表了线程需要完成的任务，因此把 `run()`方法称为线程执行体。
2. 创建 `Thread` 子类的实例，即创建了线程对象
3. 调用线程对象的 `start()`方法来启动该线程

Task1 代码&运行结果截图：

- 运行结果：每次运行代码，每语句的打印顺序不同，打印 id 和打印 name 顺序也不对应



- 代码：

```

1 package HW4;
2
3 public class thread01 extends Thread {
4     public void run() {
5         System.out.println("The name of this thread is: " + Thread.currentThread().getName());
6         System.out.println("The id of " + Thread.currentThread().getName() + " is: " + Thread.currentThread().threadId());
7     }
8
9     public static void main(String[] args) {
10         thread01 th1 = new thread01();
11         thread01 th2 = new thread01();
12         thread01 th3 = new thread01();
13         th1.start();
14         th2.start();
15         th3.start();
16     }
17 }

```

- 多次运行结果不同原因：

代码中，创建了三线程 `th1`、`th2` 和 `th3`，并且它们都是并发运行的。虽然调用了它们的 `start()`方

法，但是线程的启动顺序和执行顺序是不确定的。因此，每次运行程序时，这三个线程的打印顺序可能会有所不同。

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的形式为 `a1b23c456d7891.....z.....`（数字1-9循环2次）。将关键代码和总结的内容写到实验报告中。

储备知识:

一、创建线程方法二-----`Runnable`

采用 `java.lang.Runnable` 接口，只需要**重写 `run` 方法**即可，步骤如下：

1. 定义 `Runnable` 接口的实现类，并重写该接口的 `run()`方法，该 `run()`方法的方法体同样是该线程的线程执行体。

2. 创建 `Runnable` 实现类的实例，并以此实例作为 `Thread` 的 `target` 来创建 `Thread` 对象，该 `Thread` 对象才是真正

的线程对象。

3. 调用线程对象的 `start()`方法来启动线程。

二、`Thread` 和 `Runnable` 的联系

- 通过实现 `Runnable` 接口，使得该类有了多线程类的特征。`run()`方法是多线程程序的一个执行目标。所有的多线程代码都在 `run` 方法里面。`Thread` 类实际上也是实现了 `Runnable` 接口的类。
- 在启动的多线程的时候，需要先通过 `Thread` 类的构造方法 `Thread(Runnable target)` 构造出对象，然后调用 `Thread` 对象的 `start()`方法来运行多线程代码。
- 实际上所有的多线程代码都是通过运行 `Thread` 的 `start()`方法来运行的。因此，**不管是继承 `Thread` 类还是实现 `Runnable` 接口来实现多线程，最终还是通过 `Thread` 的对象的 API 来控制线程的**，`Thread` 类的 API 是进行多线程编程的基础。

Tips: `Runnable` 对象仅仅作为 `Thread` 对象的 **target**，`Runnable` 实现类里包含的 `run()`方法仅作为**线程执行体**。而实际的**线程对象依然是 `Thread` 实例**，只是该 `Thread` 线程负责执行其 `target` 的 `run()`方法。

三、`Thread` 和 `Runnable` 的区别

- 如果一个类继承 `Thread`，则不适合资源共享。但是如果实现了 `Runnable` 接口的话，则很容易的实现资源共享。

➤ **实现 Runnable 接口比继承 Thread 类所具有的优势：**

1. 适合多个相同的程序代码的线程去共享同一个资源。
2. 可以避免 java 中的单继承的局限性。
3. 增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和线程独立。
4. 线程池只能放入实现 Runnable 或 Callable 类线程，不能直接放入继承 Thread 的类。

➤ 在 java 中，每次程序运行**至少启动 2 个线程**。一个是 main 线程，一个是垃圾收集线程。因为每当使用 java 命令执行一个类的时候，实际上都会启动一个 JVM，每一个 JVM 其实就是在操作系统中启动了一个进程。

Task2 代码&运行结果截图：

➤ 运行结果：每次输出打印的字母和数字输出顺序均不同。

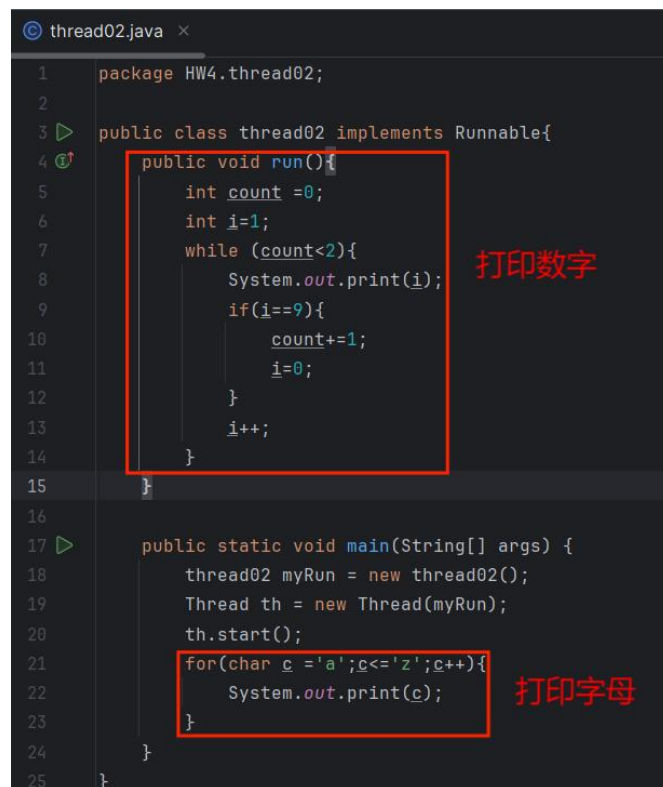


```
运行 thread02 第一次运行:
"C:\Program Files\Java\jdk-20\bin\java.exe"
abcdefghijklmnopqrstuvwxyz
123456789123456789
进程已结束，退出代码为 0
```



```
运行 thread02 第二次运行:
"C:\Program Files\Java\jdk-20\bin\java.exe"
123456789123456789
abcdefghijklmnopqrstuvwxyz
进程已结束，退出代码为 0
```

➤ 代码：



```
thread02.java
1 package HW4.thread02;
2
3 public class thread02 implements Runnable{
4     public void run(){
5         int count =0;
6         int i=1;
7         while (count<2){
8             System.out.print(i); 打印数字
9             if(i==9){
10                 count+=1;
11                 i=0;
12             }
13             i++;
14         }
15     }
16
17     public static void main(String[] args) {
18         thread02 myRun = new thread02();
19         Thread th = new Thread(myRun);
20         th.start();
21         for(char c = 'a'; c<='z'; c++){ 打印字母
22             System.out.print(c);
23         }
24     }
25 }
```

➤ 结果不同原因：

1. 线程调度的随机性：不同的操作系统和 JVM 对线程调度的策略不同，可能导致不同时间点上不同线

程获得 CPU 执行权的顺序不同。

2. 线程之间的竞争：由于两个线程共享资源（例如标准输出），可能会出现竞争条件，导致输出的顺序不确定。

Task3: 完善代码，用 join 方法该写 ThreadTest03 类实现正常的逻辑，并将关键代码和结果写到实验报告中。

储备知识：

一、pdf 示例注释与否差异

```
public class Main {  
    public static void main(String[] args) throws InterruptedException {  
        var t = new Thread() -> {  
            for (int i = 1; i <= 100; i++) {  
                System.out.println(i);  
            }  
        };  
        t.start();  
        t.join(); // 试试看把这行注释掉  
        for (int i = -100; i <= -1; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

➤ 该例注释前后输出有差别：注释前，先打印完 1~100，然后打印-100~-1；注释后 1~100 和-100~-1 交叉打印，打印具体数字的顺序不固定。

➤ 不同结果原因：

当注释掉 t.join()语句时，主线程和子线程会**并发执行**。主线程启动子线程后就会继续执行自己的循环打印-100 到-1 的操作，而子线程也会开始执行它的循环打印 1 到 100 的操作。由于主线程和子线程是并发执行的，因此它们的输出会交错在一起，并且由于线程调度的随机性，每次运行的输出结果可能稍有不同。

当取消注释 t.join()语句时，**主线程会等待子线程执行完成后再继续执行**。这样，子线程会先完成循环打印 1 到 100 的操作，然后主线程才开始执行循环打印-100 到-1 的操作。因此，输出结果会按照子线程和主线程的顺序依次输出，不会交错。

二、创建线程方法三-----匿名内部类

- 使用线程的匿名内部类方式，可以方便的实现每个线程执行不同的线程任务操作。使用匿名内部类的方式实现 Runnable 接口，重新 Runnable 接口中的 run 方法，下面有两种形式（第一种更加简洁）：

```
public class niming {
    public static void main(String[] args) {
        //format1
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 20; i++) {
                    System.out.println("thread:" + i);
                }
            }
        }).start();

        //format2
        Runnable r = new Runnable(){
            public void run(){
                for (int i = 0; i < 20; i++) {
                    System.out.println("thread:"+i);
                }
            }
        }; //---这个整体 相当于new MyRunnable()
        new Thread(r).start();

        for (int i = 0; i < 20; i++) {
            System.out.println("main:" + i);
        }
    }
}
```

三、创建线程方法四-----Lambda 表达式

- Lambda 表达式标准格式：(参数类型 参数名称) -> { 代码语句 }
- **省略规则。**在 Lambda 标准格式的基础上，使用省略写法的规则为：
1. 小括号内参数的类型可以省略；
 2. 如果小括号内有且仅有一个参，则小括号可以省略；
 3. 如果大括号内有且仅有一个语句，则无论是否有返回值，都可以省略大括号、return 关键字及语句分号。
- Lambda 表达式使用场景：
1. 使用在**函数式接口**中：用于**替代函数式接口的匿名类实现**，可以更加简洁地实现函数式接口的抽象方法。
 2. 使用在集合操作中：Java 8 引入的 Stream API 提供了丰富的集合操作方法，结合 Lambda 表达式可以更加便捷地进行数据处理、筛选和转换。

3. 使用在**并发编程**中：在多线程编程中，Lambda 表达式可以简化线程的创建和管理，例如**使用 Runnable 接口或者 Callable 接口来创建线程**。
4. 使用在事件处理中：Swing 和 JavaFX 等 GUI 编程中，Lambda 表达式可以替代传统的事件监听器，使得代码更加简洁和易读。
5. 使用在函数式编程中：Lambda 表达式支持函数式编程范式，可以用于编写更为函数式、声明式的代码。

➤ **创建线程时：** () -> System.out.println("多线程任务执行！")

1. 前面的一对小括号即 run 方法的参数（无），代表不需要任何条件；
2. 中间的一个箭头代表将前面的参数传递给后面的代码；
3. 后面的输出语句可以替换成其他在线程完成的逻辑代码。

将三、匿名内部类方法代码修改为 Lambda 表达式：

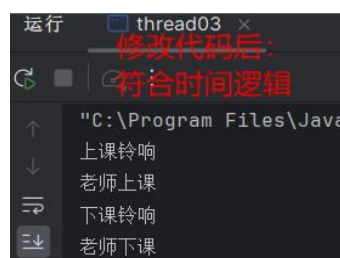
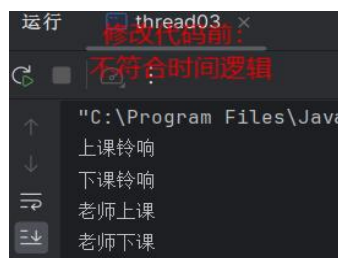
```
new Thread(() -> {  
    for (int i = 0; i < 20; i++) {  
        System.out.println("thread:" + i);  
    }  
}).start();
```

四、join() 方法

- 作用：让一个线程等待另一个线程执行结束。当在一个线程中调用另一个线程的 join() 方法时，当前线程会被阻塞，直到被调用 join() 方法的线程执行完成或者超时。
- **接受超时参数：** t.join(time)，表示最多等待 time 毫秒，如果在这个时间内 t 线程没有执行完成，剩余线程也会继续执行。

Task3 代码&运行结果截图：

- 运行结果：修改前，上课铃声响后，下课铃声就响了，老师还没上课，不符合时间逻辑；修改后，才符合实际情况。



➤ 代码：

```
thread03.java ×
1 package HW4;
2
3 public class thread03 implements Runnable {
4     @Override
5     public void run() {
6         System.out.println(Thread.currentThread().getName());
7     }
8
9     public static void main(String[] args) throws InterruptedException {
10         thread03 join = new thread03();
11         Thread thread1 = new Thread(join, name: "上课铃响");
12         Thread thread2 = new Thread(join, name: "老师上课");
13         Thread thread3 = new Thread(join, name: "下课铃响");
14         Thread thread4 = new Thread(join, name: "老师下课");
15         thread1.start();
16         thread1.join();
17         thread2.start();
18         thread2.join();
19         thread3.start();
20         thread3.join();
21         thread4.start();
22     }
23 }
```

➤ 修改代码前后输出结果不同原因：

当注释掉 join() 方法时，各个线程启动后会并发执行，它们之间的执行顺序是不确定的，因此输出结果会有所不同。而当使用 join() 方法时，主线程会等待调用 join() 的线程执行完毕后再继续往下执行，这样就能保证线程的执行顺序，因此输出结果会按照代码中的顺序依次打印。

任务4: 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。

储备知识：

一、守护线程

- 守护线程 (Daemon Thread) 是一种在程序运行时在后台提供服务的线程，它并不阻止程序的终止。当所有的非守护线程结束时，程序会自动退出，不会等待守护线程执行完毕。守护线程通常被用来执行一些后台任务，例如垃圾回收等。
- 设置方法：线程名称.setDaemon(true) 方法将线程设置为守护线程

- 当一个线程被设置为守护线程后，它会**随着主线程（非守护线程）的结束而结束**。如果所有的非守护线程都结束了，那么守护线程也会随之结束。

二、垃圾回收

- 一种自动管理内存的机制，用于在程序运行时识别和回收不再被程序所使用的内存，从而避免内存泄漏和提高内存利用率，由虚拟机（JVM）负责执行的。
- 操作原理：当程序在运行过程中创建对象时，这些对象会被分配在堆内存中。当某个对象不再被引用，即没有任何引用指向该对象时，该对象就成为垃圾。垃圾回收器会定期检查程序中的对象，找出这些不再被引用的对象，并释放它们占用的内存空间，使得这部分内存可以被重新利用。
- 一般不需要显式地调用垃圾回收，**因为 JVM 会根据需要自动触发垃圾回收操作**。但是，在某些特殊情况下，可以通过 `System.gc()` 方法来建议 JVM 执行垃圾回收。

三、主动结束守护线程

- 可以调用线程对象的 `join()` 方法或者设置线程的标志位来终止线程的执行：

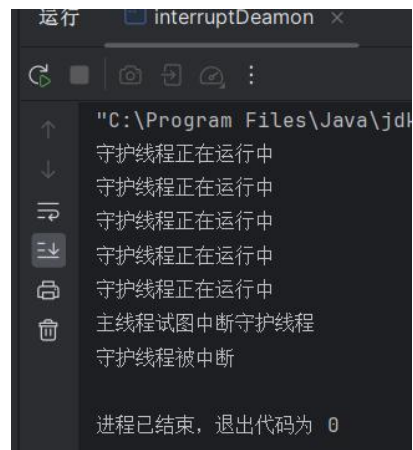
1. 使用 `interrupt()` 方法：在守护线程的执行代码中，可以定期检查线程的中断状态，如果线程被中断，则结束线程的执行。可以通过调用 `interrupt()` 方法来中断线程。

- 示例代码：

```
public class InterruptDaemon {
    public static void main(String[] args) throws InterruptedException {
        Thread daemonThread = new Thread(() -> {
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println("守护线程正在运行中");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("守护线程被中断");
                    return;
                }
            }
        });
        daemonThread.setDaemon(true); // 设置为守护线程
        daemonThread.start();

        // 主线程执行一段时间后中断守护线程
        Thread.sleep(5000);
        System.out.println("主线程试图中断守护线程");
        daemonThread.interrupt(); // 中断守护线程
    }
}
```

➤ 运行结果：

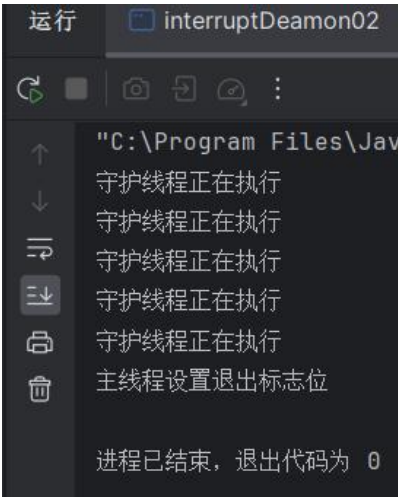


2. 设置标志位：在守护线程的执行循环中，通过检查一个标志位来决定是否退出循环从而结束线程的执行。

➤ 示例代码：

```
interruptDeamon02.java x
3 public class interruptDeamon02 extends Thread {
4     2 个用法
5     private volatile boolean exitFlag = false;
6
7     public void run() {
8         while (!exitFlag) {
9             System.out.println("守护线程正在执行");
10            try {
11                Thread.sleep(1000);
12            } catch (InterruptedException e) {
13                e.printStackTrace();
14            }
15        }
16        System.out.println("守护线程已退出");
17    }
18
19    1 个用法
20    public void setExitFlag(boolean exitFlag) {
21        this.exitFlag = exitFlag;
22    }
23
24    public static void main(String[] args) throws InterruptedException {
25        interruptDeamon02 daemonThread = new interruptDeamon02();
26        daemonThread.setDaemon(true); // 设置为守护线程
27        daemonThread.start();
28
29        // 主线程执行一段时间后设置退出标志位
30        Thread.sleep(5000);
31        System.out.println("主线程设置退出标志位");
32        daemonThread.setExitFlag(true); // 设置退出标志位
33    }
34 }
```

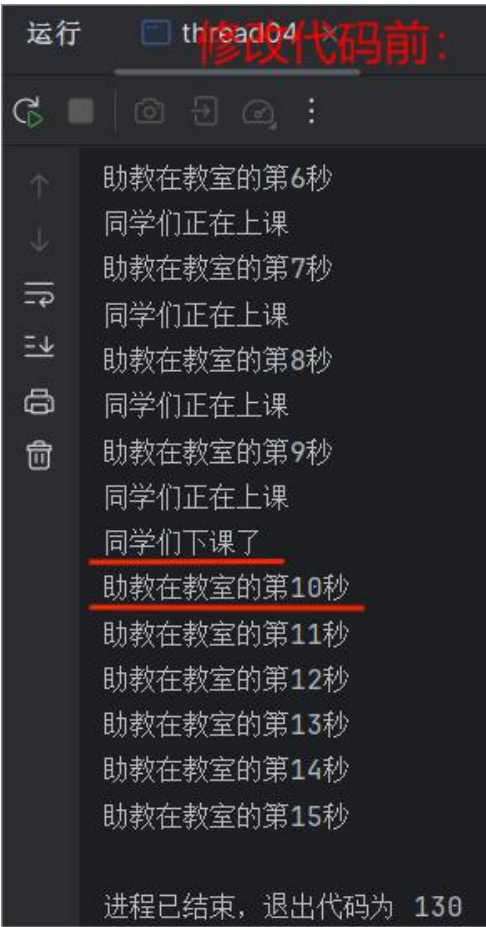

➤ 运行结果：



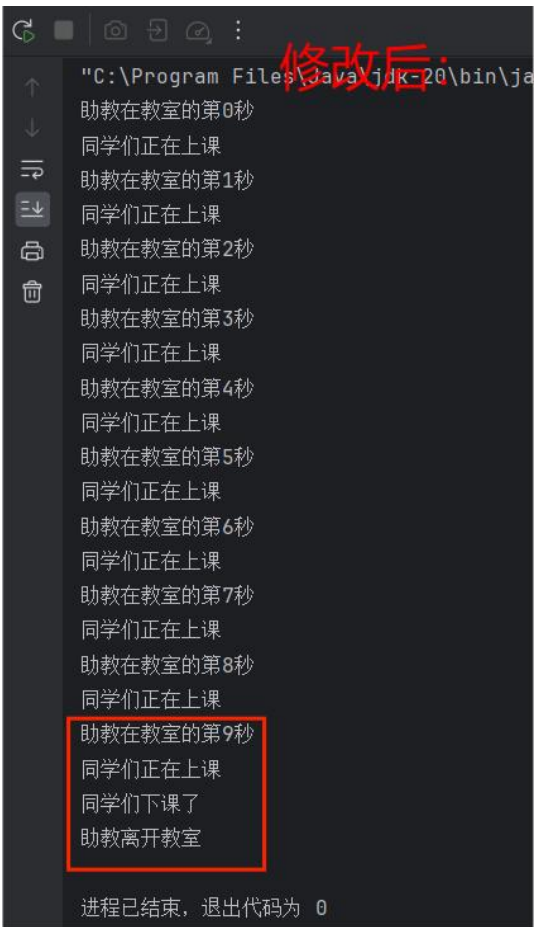
```
运行 interruptDeamon02
守护线程正在执行
守护线程正在执行
守护线程正在执行
守护线程正在执行
守护线程正在执行
主线程设置退出标志位
进程已结束，退出代码为 0
```

Task4 运行结果&代码截图：

- 运行结果：修改前，即使课程结束，同学们已经下课，但是助教仍在教室，不符合逻辑，修改后，才符合实际情况。



```
运行 thread04
修改代码前:
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
助教在教室的第10秒
助教在教室的第11秒
助教在教室的第12秒
助教在教室的第13秒
助教在教室的第14秒
助教在教室的第15秒
进程已结束，退出代码为 130
```



```
运行 thread04
修改后:
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
助教离开教室
进程已结束，退出代码为 0
```

➤ 代码：

```
thread04.java x
1 package HW4;
2
3 public class thread04 implements Runnable {
4     @Override
5     public void run() {
6         int worktime = 0;
7         while (true) {
8             System.out.println("助教在教室的第" + worktime + "秒");
9             try {
10                 Thread.currentThread().sleep(1000);
11             } catch (InterruptedException e) {
12                 e.printStackTrace();
13             }
14             worktime++;
15         }
16     }
17
18     public static void main(String[] args) throws InterruptedException {
19         thread04 inClassroom = new thread04();
20         Thread thread = new Thread(inClassroom, "助教");
21         thread.setDaemon(true);
22         thread.start();
23
24         for (int i = 0; i < 10; i++) {
25             thread.sleep(1000);
26             System.out.println("同学们正在上课");
27             if (i == 9) {
28                 System.out.println("同学们下课了");
29             }
30         }
31         System.out.println("助教离开教室");
32     }
33 }
```

➤ 前后结果不同原因：

当线程是非守护线程时，即使主线程（main 线程）执行完毕，非守护线程仍然会继续执行。因此，即使主线程执行完了，“助教”线程仍然会一直输出“助教在教室的第 X 秒”，直到程序手动结束或者发生异常；而当线程被设置为守护线程，当主线程（main 线程）执行完毕后，守护线程也会随之结束。因此，主线程执行完毕后，“助教”线程会立刻停止输出，程序结束。

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。将关键代码和总结的内容写到实验报告中，并总结线程让步的特点。

储备知识：

一、线程让步特点

➤ 线程让步 (yield) :一种线程调度的手段, 通过让出 CPU 执行时间, 帮助调度器更好地分配 CPU 资源, 提高多线程程序的效率和公平性。但过度使用线程让步也可能导致线程之间的频繁切换, 影响程序性能。

➤ 特点:

1. 主动性: 线程让步是由线程自身主动调用 `Thread.yield()` 方法来实现的, 表示当前线程愿意让出 CPU 的执行权限, 使其他线程有机会获得执行机会。

2. 不释放锁: 并不会释放已经获取的锁, 因此在让步后, 线程仍然保持对锁的持有, 其他线程无法获得该锁。

3. 不确定性: 不保证其他线程一定会被调度执行, 只是增加了其他线程被调度执行的可能性。

4. 轻量级: 线程让步是一种轻量级的线程调度方式, 在某些情况下可以提高多线程程序的效率, 避免出现线程饥饿现象。

5. 平台相关性: 具体行为和效果可能会因操作系统和虚拟机的不同而有所差异, 因此在编写多线程程序时需要注意平台相关性。

二、 线程调度方法

1. `yield()` 方法: 线程可以通过调用 `Thread.yield()` 方法来让出 CPU 时间片, 让其他具有相同优先级的线程有机会获得执行。调用 `yield()` 方法并不会释放锁, 而是告诉调度器自己愿意让出 CPU 时间, 但可能会再次被调度执行。

2. `sleep()` 方法: 线程可以通过调用 `Thread.sleep()` 方法让出 CPU 时间片, 并进入休眠状态一段时间。在这段时间内, 线程不会占用 CPU 资源, 但仍然持有锁, 因此其他线程无法获取这些锁。

3. `join()` 方法: 一个线程可以调用另一个线程的 `join()` 方法, 使得当前线程进入等待状态, 直到另一个线程执行完毕。这种方式可以用于协调多个线程的执行顺序。

4. `wait()` 和 `notify()/notifyAll()` 方法: 线程可以通过调用对象的 `wait()` 方法让出 CPU 时间, 并进入等待状态, 直到其他线程调用相同对象的 `notify()` 或 `notifyAll()` 方法唤醒它。这种机制常用于线程间的通信和协作。

5. 自旋等待: 线程在特定的条件下可以选择自旋等待, 即反复检查某个条件是否满足, 从而避免进入阻塞状态。这种方式适用于短暂的等待情况, 可以减少线程切换带来的开销。

三、 线程优先级

➤ 定义：线程优先级是一个整数，范围通常是 1 到 10 之间。线程的优先级越高，意味着它更可能在竞争 CPU 时间时被选择执行。

➤ 特点：

1. 范围：Java 线程的优先级范围通常是 1 到 10，其中 1 是最低优先级，10 是最高优先级。线程的默认优先级通常是 5。

2. 继承性：当一个线程创建了另一个线程时，新线程的优先级将与创建它的线程相同。

3. 可更改性：允许通过 `Thread.setPriority()` 方法动态地改变线程的优先级。但是，这种更改并不一定会被操作系统完全支持。

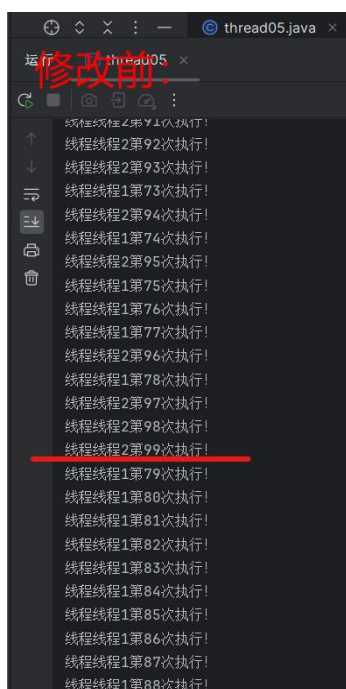
4. 平台相关性：线程优先级的实现在不同的操作系统和 Java 虚拟机中可能会有所不同，因此在编写依赖于优先级的多线程代码时需要注意平台相关性。

5. 不保证性：Java 规范没有强制要求虚拟机必须按照优先级来调度线程。虽然高优先级的线程在大多数情况下会比低优先级的线程更优先执行，但并不保证绝对的执行顺序。

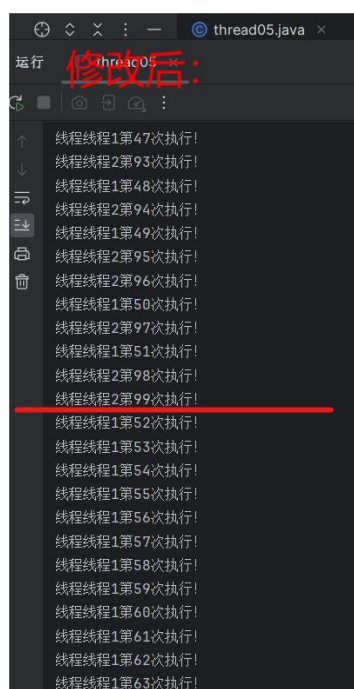
6. 反对滥用：滥用线程优先级会导致不可预测的结果，并且在不同的平台上可能表现不一致。

Task5 运行结果& 代码截图：

➤ 运行结果：可以看出线程 2 最后一次执行完成时间上，修改后比修改前更靠前。

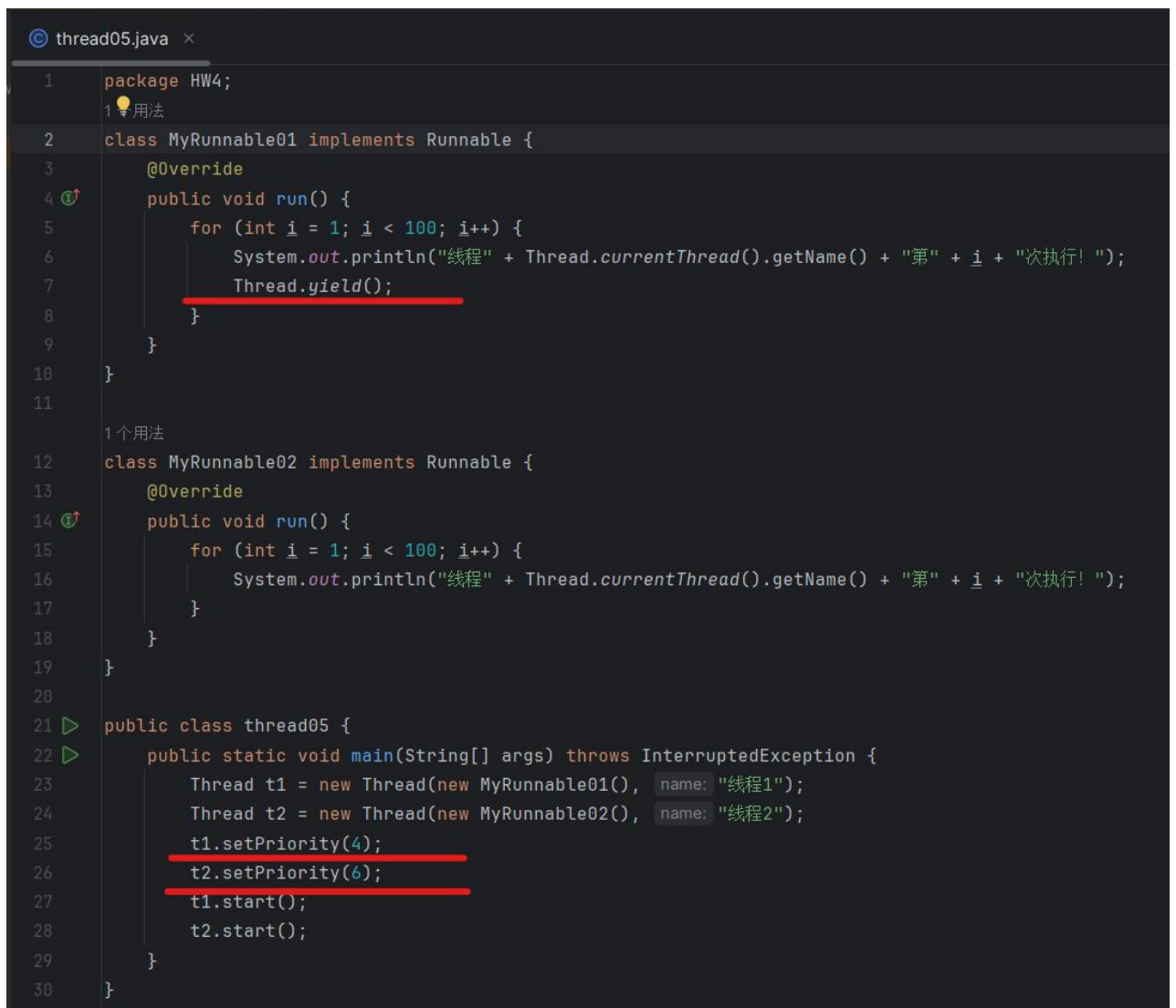


```
运行 thread05
修改前:
线程线程2第92次执行!
线程线程2第93次执行!
线程线程1第73次执行!
线程线程2第94次执行!
线程线程1第74次执行!
线程线程2第95次执行!
线程线程1第75次执行!
线程线程1第76次执行!
线程线程1第77次执行!
线程线程2第96次执行!
线程线程1第78次执行!
线程线程2第97次执行!
线程线程2第98次执行!
线程线程2第99次执行!
线程线程1第79次执行!
线程线程1第80次执行!
线程线程1第81次执行!
线程线程1第82次执行!
线程线程1第83次执行!
线程线程1第84次执行!
线程线程1第85次执行!
线程线程1第86次执行!
线程线程1第87次执行!
线程线程1第88次执行!
```



```
运行 thread05
修改后:
线程线程1第47次执行!
线程线程2第93次执行!
线程线程1第48次执行!
线程线程2第94次执行!
线程线程1第49次执行!
线程线程2第95次执行!
线程线程2第96次执行!
线程线程1第50次执行!
线程线程2第97次执行!
线程线程1第51次执行!
线程线程2第98次执行!
线程线程2第99次执行!
线程线程1第52次执行!
线程线程1第53次执行!
线程线程1第54次执行!
线程线程1第55次执行!
线程线程1第56次执行!
线程线程1第57次执行!
线程线程1第58次执行!
线程线程1第59次执行!
线程线程1第60次执行!
线程线程1第61次执行!
线程线程1第62次执行!
线程线程1第63次执行!
```

➤ 代码截图：



```
1 package HW4;
2 class MyRunnable01 implements Runnable {
3     @Override
4     public void run() {
5         for (int i = 1; i < 100; i++) {
6             System.out.println("线程" + Thread.currentThread().getName() + "第" + i + "次执行! ");
7             Thread.yield();
8         }
9     }
10 }
11
12 class MyRunnable02 implements Runnable {
13     @Override
14     public void run() {
15         for (int i = 1; i < 100; i++) {
16             System.out.println("线程" + Thread.currentThread().getName() + "第" + i + "次执行! ");
17         }
18     }
19 }
20
21 public class thread05 {
22     public static void main(String[] args) throws InterruptedException {
23         Thread t1 = new Thread(new MyRunnable01(), name: "线程1");
24         Thread t2 = new Thread(new MyRunnable02(), name: "线程2");
25         t1.setPriority(4);
26         t2.setPriority(6);
27         t1.start();
28         t2.start();
29     }
30 }
```

➤ 修改前后结果不同原因：

修改前，两个线程的优先级没有被显式设置，因此它们会使用默认的优先级。默认情况下，线程的优先级是 5；修改后，通过设置线程的优先级，线程"线程 1"和"线程 2"分别被设置为 4 和 6 的优先级。这样一来，会影响线程调度器在调度这两个线程时的顺序和频率，线程 2 可能会获得更多的执行机会。但这并不是绝对的，因为高优先级只是一个提示，具体的调度顺序还受到操作系统和线程调度器的影响。

二、Java 多线程编程_2 题目要求及功能实现情况

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。

储备知识：

一、 *volatile* 关键字

- 用来修饰变量，告诉编译器和虚拟机，这个变量是可能被多个线程同时访问的，因此不要进行各种优化。

具体来说，主要有以下两个作用：

1. 保证可见性：当一个变量被声明为 *volatile* 后，每次这个变量发生改变时，都会**强制**将该变量的最新值刷新到主内存中，而且每次使用这个变量时，都会从主内存中**重新读取**。这样可以保证不同线程之间对变量的修改是可见的，即一个线程修改了变量的值，其他线程能够立即看到最新的值。

2. 禁止指令重排序：*volatile* 关键字修饰的变量，可以**防止指令重排序优化**，保证了程序的有序性。这样就避免了多线程环境下可能出现的意外情况。

运行 Task1 后两种可能结果截图：

- 运行结果：每次运行大概率输出结果均不相同。

```
"C:\Program Files\Java\jdk-2  
所有线程结束后的 num 值为：1009
```

```
"C:\Program Files\Java\jdk-  
所有线程结束后的 num 值为：997
```

- 结果不同原因：

每次运行输出结果不同的原因是由于在 *PlusMinus* 类中的 *num* 变量没有进行线程同步操作（比如使用锁或者原子类），导致多个线程对 *num* 同时进行读写操作时可能会出现竞态条件，从而导致结果不确定。

Task2: 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，将修改后的 *PlusMinus* 基础类代码附在实验报告中。

储备知识：

一、 线程同步方式一----- *synchronized* 关键字

- 同步代码块：*synchronized* 关键字可以用于方法中的某个区块中，表示只对这个区块的资源实行互斥访问。格式：***synchronized*(同步锁){ 需要同步操作的代码 }**
- 同步锁：对象的同步锁可以想象为在对象上标记了一个锁。
 1. 锁对象可以是任意类型。
 2. 多个线程对象要使用同一把锁。
 3. 在任何时候,最多允许一个线程拥有同步锁,谁拿到锁就进入代码块,其他的线程只能在外等着

(BLOCKED)。

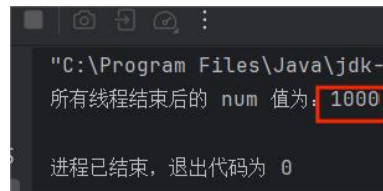
- 同步方法：使用 `synchronized` 修饰的方法，保证 A 线程执行该方法的时候，其他线程只能在方法外等着。格式：**`public synchronized void method(){ 可能会产生线程安全问题的代码 }`**
- 同步锁：对于非 `static` 方法，同步锁就是 `this`；对于 `static` 方法，我们使用当前方法所在类的字节码对象 (类名.class)。

二、 线程同步方式一----- Lock 锁

- 原理：Lock 锁由 `java.util.concurrent.locks.Lock` 提供，具有比 `synchronized` 代码块和 `synchronized` 方法更广泛的锁定操作，除了同步代码块/同步方法具有的功能，Lock 更体现面向对象。
- Lock 锁也称同步锁，加锁与释放锁方法：
 1. `public void lock()` :加同步锁。
 2. `public void unlock()` :释放同步锁。

Task2 运行结果& 代码截图：

- 修改后运行结果：



- 代码：

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

➤ 修改前后打印结果不一致原因：

在修改前的情况下，由于多个线程同时对 num 进行加一和减一操作，没有进行同步控制，可能会导致数据不一致的问题；而修改后，通过添加同步锁，保证了每次对 num 的操作都是互斥的，不会出现多个线程同时修改 num 的情况，从而保证了数据的一致性。

• **Task3:** 设计 3 个线程发生死锁的场景并编写代码，将关键代码和结果附在实验报告中。

储备知识：

一、 线程死锁

- Java 中线程发生死锁的情况通常是由于多个线程之间相互等待对方释放资源而无法继续执行的情况。
- 典型场景：多个线程互相持有对方需要的资源，并且不释放已经持有的资源，导致彼此无法继续执行。
- pdf 示例运行代码后输出结果截图如下：

可以看出：输出一直停留在 thread1 等待中阶段，无法正常退出，证明此时出现了线程死锁情况。

```
"C:\Program Files\Java\jdk-20\bin\java.exe"
thread1 正在占用 plusMinus1
thread2 正在占用 plusMinus2
thread2 试图继续占用 plusMinus1
thread1 试图继续占用 plusMinus2
thread1 等待中...
```

➤ 原因为：

线程 thread1 和 thread2 在获取锁的顺序上存在问题，导致了相互之间的等待。

当 thread1 先获取了 plusMinus1 的锁并尝试获取 plusMinus2 的锁时，如果此时 plusMinus2 已经被 thread2 获取了，那么 thread1 就会被阻塞，等待 plusMinus2 的锁释放。

同时，当 thread2 先获取了 plusMinus2 的锁并尝试获取 plusMinus1 的锁时，如果此时 plusMinus1 已经被 thread1 获取了，那么 thread2 就会被阻塞，等待 plusMinus1 的锁释放。

这样，thread1 和 thread2 互相等待对方释放锁，导致了死锁的发生。

Task3 运行结果&代码截图：

➤ 代码：

```
myExample.java x PlusMinus.java

1 package HW5.hw03;
2
3 public class myExample {
4     public static void main(String[] args) {
5         PlusMinus pm1 = new PlusMinus();
6         pm1.num = 1000;
7         PlusMinus pm2 = new PlusMinus();
8         pm2.num = 1000;
9         PlusMinus pm3 = new PlusMinus();
10        pm3.num = 1000;
11
12        Thread th1 = new Thread(() -> {
13            synchronized (pm1) {
14                System.out.println("Thread1 正在占用pm1");
15                try {
16                    Thread.sleep(1000);
17                } catch (InterruptedException e) {
18                    e.printStackTrace();
19                }
20                System.out.println("Thread1 试图继续占用pm2");
21                System.out.println("Thread1 等待中...");
22                synchronized (pm2) {
23                    System.out.println("Thread1 成功占用了pm2");
24                }
25            }
26        });
27        Thread th2 = new Thread(() -> {
28            synchronized (pm2) {
29                System.out.println("Thread2 正在占用pm2");
30                System.out.println("Thread2 试图继续占用pm3");
31                System.out.println("Thread2 等待中...");
32            }
33            synchronized (pm3) {
34                System.out.println("Thread2 成功占用了pm3");
35            }
36        });
37        Thread th3 = new Thread(() -> {
38            synchronized (pm3) {
39                System.out.println("Thread3 正在占用pm3");
40                System.out.println("Thread3 试图继续占用pm1");
41                System.out.println("Thread3 等待中...");
42                synchronized (pm1) {
43                    System.out.println("Thread3 成功占用了pm1");
44                }
45            }
46        });
47        th1.start();
48        th2.start();
49        th3.start();
50    }
51 }
```

➤ 运行结果截图：



➤ 出现结果原因：

这段代码存在死锁的原因是三个线程分别试图占用三个对象（pm1、pm2、pm3）的锁，并且它们的获取锁的顺序不同，可能导致循环等待的情况，从而导致死锁。

当 th1 线程先获取了 pm1 对象的锁并尝试获取 pm2 对象的锁时，如果此时 pm2 对象已经被 th2 线程获取了，那么 th1 线程就会被阻塞，等待 pm2 对象的锁释放。

同时，当 th2 线程先获取了 pm2 对象的锁并尝试获取 pm3 对象的锁时，如果此时 pm3 对象已经被 th3 线程获取了，那么 th2 线程就会被阻塞，等待 pm3 对象的锁释放。

最终，当 th3 线程尝试获取 pm1 对象的锁时，由于 pm1 对象已经被 th1 线程获取了，而 th1 线程又在等待 pm2 对象的锁，形成了一个闭环，导致所有线程都被阻塞，从而发生死锁。

Task3 实现过程中出现错误的代码及原因分析：

➤ 代码：


```
myExample.java × wrongExample.java × PlusMinus.java

1 package HW5.hw03;
2
3 public class wrongExample {
4     public static void main(String[] args) {
5         PlusMinus pm1 = new PlusMinus();
6         pm1.num = 1000;
7         PlusMinus pm2 = new PlusMinus();
8         pm2.num = 1000;
9         PlusMinus pm3 = new PlusMinus();
10        pm3.num = 1000;
11
12        new Thread(() -> {
13            synchronized (pm1) {
14                System.out.println("Thread1 正在占用pm1");
15                try {
16                    Thread.sleep(1000);
17                } catch (InterruptedException e) {
18                    e.printStackTrace();
19                }
20                System.out.println("Thread1 试图继续占用pm2");
21                System.out.println("Thread1 等待中...");
22                synchronized (pm2) {
23                    System.out.println("Thread1 成功占用了pm2");
24                }
25            }
26        }).start();
27        new Thread(() -> {
28            synchronized (pm2) {
29                System.out.println("Thread2 正在占用pm2");
30                System.out.println("Thread2 试图继续占用pm3");
31                System.out.println("Thread2 等待中...");
32                synchronized (pm3) {
33                    System.out.println("Thread2 成功占用了pm3");
34                }
35            }
36        }).start();
37        new Thread(() -> {
38            synchronized (pm3) {
39                System.out.println("Thread3 正在占用pm3");
40                System.out.println("Thread3 试图继续占用pm1");
41                System.out.println("Thread3 等待中...");
42                synchronized (pm1) {
43                    System.out.println("Thread3 成功占用了pm1");
44                }
45            }
46        }).start();
47    }
48 }
```

➤ 未出现死锁现象原因:

- **Task4:** 阐述 synchronized 关键字在实例方法上的作用，然后运行本代码，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。

一、线程交互

1. **共享内存**：多个线程可以通过共享内存来进行通信，它们可以读取和修改同一块内存区域的数据。

2. 线程间的通信方式:

wait()、notify()、notifyAll()：基本线程间通信的方式，基于对象的监视器机制。线程可以调用 wait() 方法来等待某个条件的满足，而其他线程可以调用 notify() 或 notifyAll() 方法来通知等

待的线程条件已经满足。

管道 (PipedInputStream 和 PipedOutputStream) : 允许在两个线程之间进行管道通信。

信号量 (Semaphore) : 用于控制同时访问某个资源的线程数量, 可以用来解决某些并发问题, 如资源的有限访问。

倒计时门栓 (CountDownLatch) : 允许一个或多个线程等待其他线程完成操作。

循环屏障 (CyclicBarrier) : 允许一组线程相互等待, 直到到达某个公共屏障点。

线程池 (ThreadPoolExecutor) : 可以实现线程的复用和管理。

3. 线程间协作的模式:

生产者-消费者模式: 一组生产者线程生成数据, 另一组消费者线程消费数据, 通过共享的数据缓冲区进行交互。

读者-写者模式: 多个读线程可以同时读取共享数据, 但写线程在写入数据时需要独占资源。

任务分工模式: 主线程将任务分发给多个工作线程, 并等待它们完成任务后收集结果。

Task4 问题答案& 结果截图:

➤ synchronized 关键字在实例方法上的作用:

synchronized 关键字用于实现线程同步, 确保对 num 变量的读写操作是原子的, 避免了多线程环境下可能出现的数据不一致性问题。

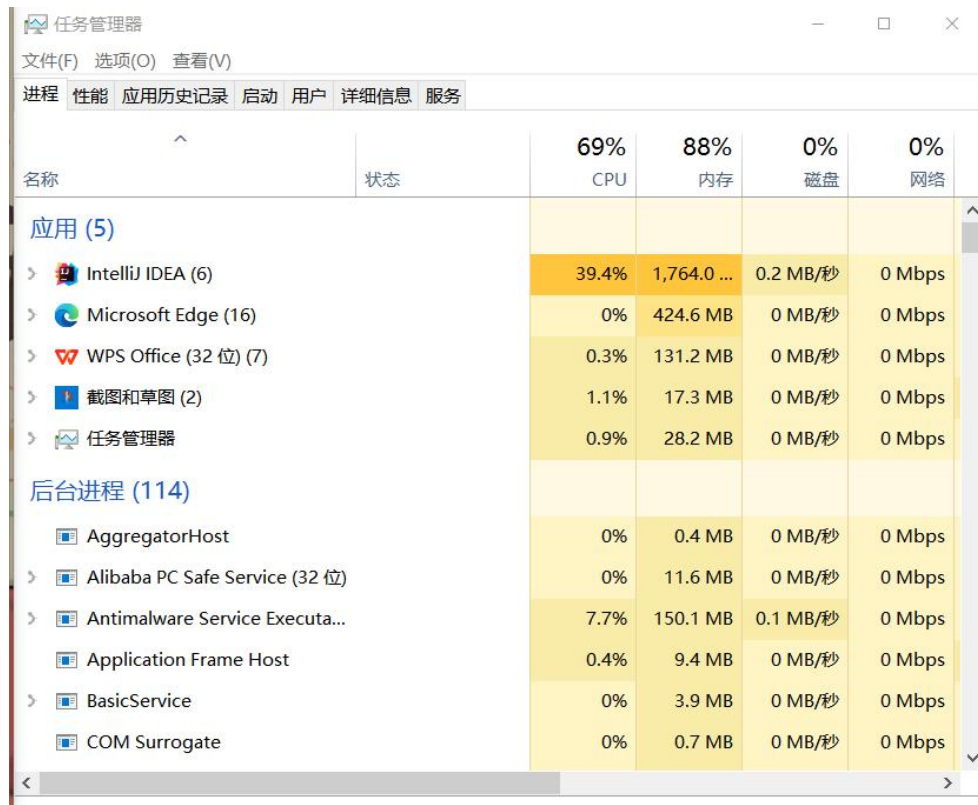
PlusMinus02 类中的 plusOne() 和 minusOne() 方法都使用了 synchronized 关键字修饰, 这意味着每次只有一个线程能够获得该对象的锁, 进入 synchronized 块中执行, 从而保证了对 num 的操作是原子性的。这样可以避免多个线程同时修改 num 导致的竞态条件问题。

在 InteractTest 类的 main 方法中, 创建了两个线程 th1 和 th2 分别执行 plusMinus.minusOne() 和 plusMinus.plusOne() 方法。由于这两个方法都是同步的, 因此当一个线程执行其中一个方法时, 另一个线程必须等待, 直到前一个线程释放锁才能执行对应的方法。

➤ CPU 的使用情况:

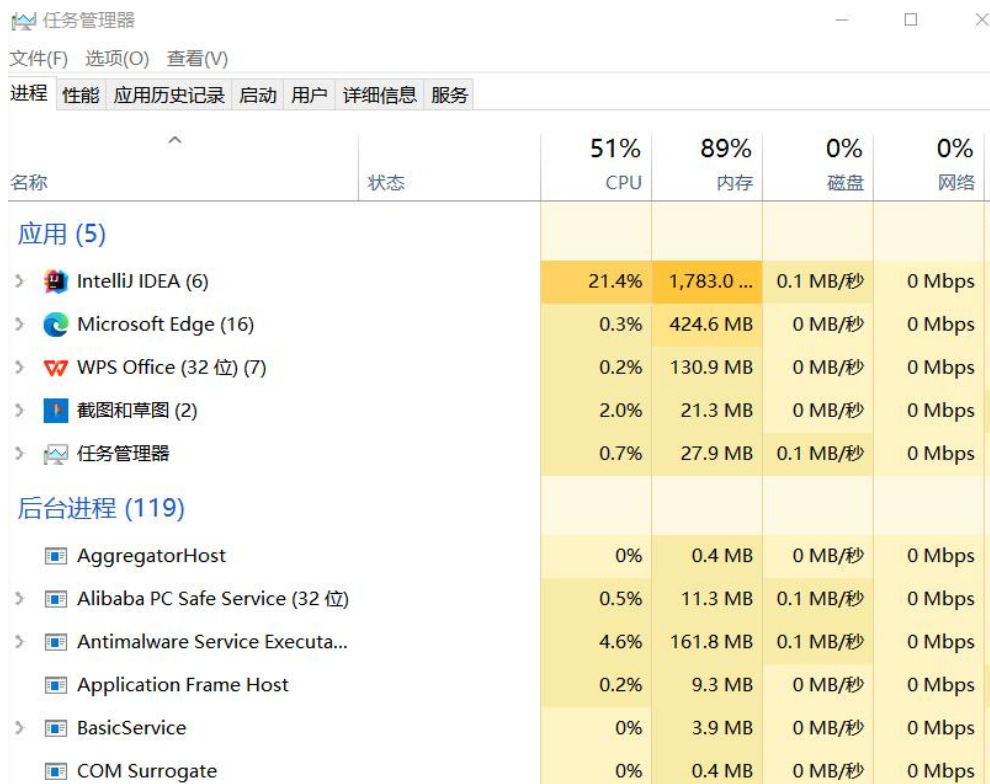
右键 windows 打开任务管理器, 查看 CPU 使用情况如下:

- 刚开始运行时, IDEA 迅速占用 CPU, 并且一度使 CPU 使用达到 70%左右



名称		69% CPU	88% 内存	0% 磁盘	0% 网络
应用 (5)					
IntelliJ IDEA (6)		39.4%	1,764.0 ...	0.2 MB/秒	0 Mbps
Microsoft Edge (16)		0%	424.6 MB	0 MB/秒	0 Mbps
WPS Office (32 位) (7)		0.3%	131.2 MB	0 MB/秒	0 Mbps
截图和草图 (2)		1.1%	17.3 MB	0 MB/秒	0 Mbps
任务管理器		0.9%	28.2 MB	0 MB/秒	0 Mbps
后台进程 (114)					
AggregatorHost		0%	0.4 MB	0 MB/秒	0 Mbps
Alibaba PC Safe Service (32 位)		0%	11.6 MB	0 MB/秒	0 Mbps
Antimalware Service Executa...		7.7%	150.1 MB	0.1 MB/秒	0 Mbps
Application Frame Host		0.4%	9.4 MB	0 MB/秒	0 Mbps
BasicService		0%	3.9 MB	0 MB/秒	0 Mbps
COM Surrogate		0%	0.7 MB	0 MB/秒	0 Mbps

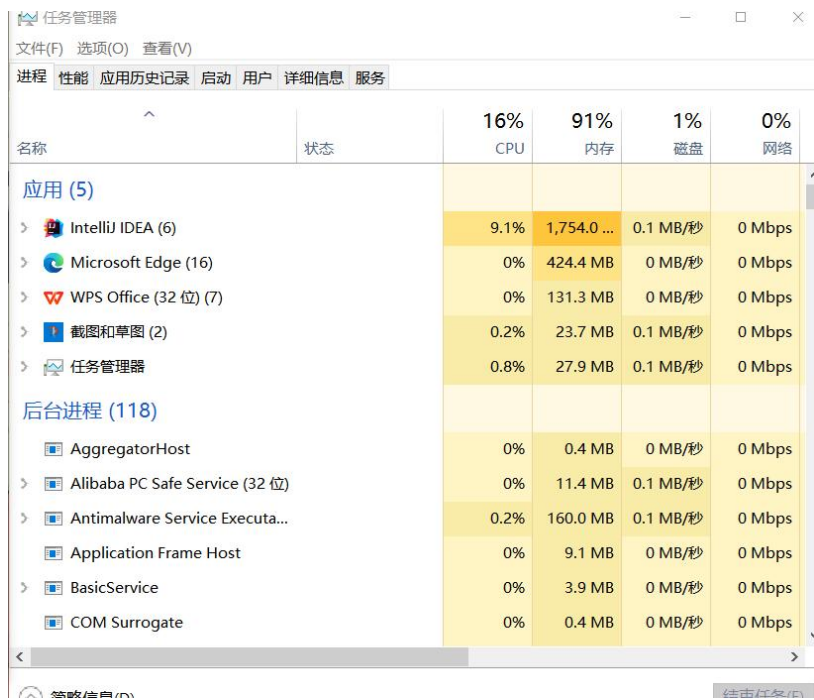
- 后来输出 num 值逐渐递减时, IDEA 占用 CPU 逐渐降低, CPU 使用也逐渐降低



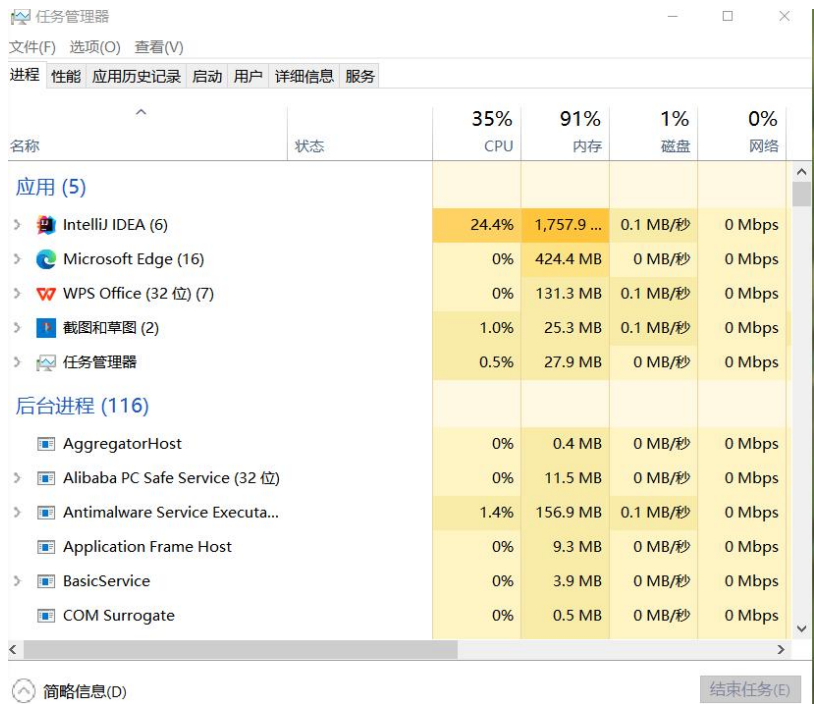
名称		51% CPU	89% 内存	0% 磁盘	0% 网络
应用 (5)					
IntelliJ IDEA (6)		21.4%	1,783.0 ...	0.1 MB/秒	0 Mbps
Microsoft Edge (16)		0.3%	424.6 MB	0 MB/秒	0 Mbps
WPS Office (32 位) (7)		0.2%	130.9 MB	0 MB/秒	0 Mbps
截图和草图 (2)		2.0%	21.3 MB	0 MB/秒	0 Mbps
任务管理器		0.7%	27.9 MB	0.1 MB/秒	0 Mbps
后台进程 (119)					
AggregatorHost		0%	0.4 MB	0 MB/秒	0 Mbps
Alibaba PC Safe Service (32 位)		0.5%	11.3 MB	0.1 MB/秒	0 Mbps
Antimalware Service Executa...		4.6%	161.8 MB	0.1 MB/秒	0 Mbps
Application Frame Host		0.2%	9.3 MB	0 MB/秒	0 Mbps
BasicService		0%	3.9 MB	0 MB/秒	0 Mbps
COM Surrogate		0%	0.4 MB	0 MB/秒	0 Mbps

- 然后当 num 刚开始在 1 和 2 之间循环时, IDEA 占用 CPU 达到最低值, CPU 使用率也达到最低, 大概

15%左右



- 最后当 num 在 1 和 2 之间循环时间增加，IDEA 占用 CPU 达到稳定，CPU 使用率也基本波动不大，大概 33%~37%之间震荡



- 实验结果截图：从结果中可以看出开始输出 num 值为 $1000-1=999$ ，然后运行了一次 $999+1=1000$ ，紧接着就一直执行-1 操作，直至减到 1 之后，再执行+1 操作，后-1，+1，-1，+1... 一直循环下去，不会自动退出。


```
"C:\Program Files\Java\jdk-20
num = 999
num = 1000
num = 999
num = 998
num = 997
num = 996
num = 995
num = 994
num = 995
num = 994
num = 993
num = 992
num = 991
num = 990
```

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

- **Task5:** 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 关键字的作用。

Task5 问题答案& 结果截图:

➤ 实验结果截图：刚开始跟去掉前的输出结果相同，999 先+1 到 1000，再逐渐递减到 1，后面有所不同，1 又开始+1，2 再+1.....逐渐递加 1，如果不手动结束，就会一直加下去。

```
"C:\Program Files\Java\jdk-20
num = 999
num = 1000
num = 999
num = 998
num = 997
num = 996
num = 995
num = 994
num = 995
num = 994
num = 993
num = 992
num = 991
num = 990
num = 989
num = 988
num = 989
num = 988
num = 987
num = 986
num = 985
num = 984
num = 983
num = 982
num = 983
num = 982
```

```
num = 13
num = 12
num = 11
num = 10
num = 9
num = 8
num = 9
num = 8
num = 7
num = 6
num = 5
num = 4
num = 3
num = 2
num = 3
num = 2
num = 1
num = 2
num = 3
num = 4
num = 5
num = 6
num = 7
num = 8
num = 9
num = 10
num = 11
```

```
num = 8305
num = 8306
num = 8307
num = 8308
num = 8309
num = 8310
num = 8311
num = 8312
num = 8313
num = 8314
num = 8315
num = 8316
num = 8317
num = 8318
num = 8319
num = 8320
num = 8321
num = 8322
num = 8323
num = 8324
num = 8325
num = 8326
num = 8327
num = 8328
num = 8329
num = 8330
```

➤ volatile 关键字作用：

通用作用在 task1 储备知识一中有提到，现分析，在该例子中 volatile 关键字作用。

在这段代码中，PlusMinus02 中的 volatile 关键字用于修饰 num 变量。volatile 关键字的主要作用是告诉编译器和运行时环境，这个变量是可能被多个线程同时访问的，因此不要进行线程本地缓存，而应该直接从主内存中读取和写入。

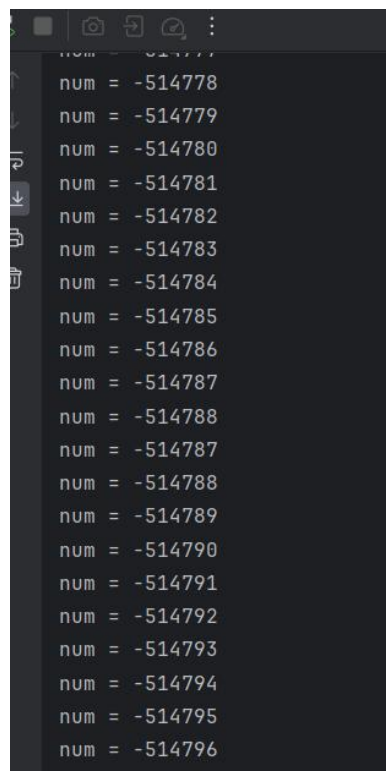
当一个线程修改了 num 的值之后，其他线程能够立即看到这个修改，而不会使用自己线程本地的缓存值。对 num 进行的读写操作都是原子的。这意味着即使是复合操作（比如 $\text{num} = \text{num} + 1$ 或 $\text{num} = \text{num} - 1$ ），也会被视为一个单独的操作，不会被其他线程中断。

- **Task6:** 在 Task4 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。

➤ 增加数量：我在代码中增加了 6 个减一操作，sleep () 中整数是随机的。

➤ 是否发生错误&运行结果：

发生错误，从输出结果中可以看出，在 Task4 输出结果基础上，递减到 1 时，程序并未停止，而是继续往下减到负数，我运行了大概二十多分钟，一直在递减，最后手动停止程序。



```
num = -514778
num = -514779
num = -514780
num = -514781
num = -514782
num = -514783
num = -514784
num = -514785
num = -514786
num = -514787
num = -514788
num = -514787
num = -514788
num = -514789
num = -514790
num = -514791
num = -514792
num = -514793
num = -514794
num = -514795
num = -514796
```

➤ 错误原因：

1. 竞态条件：在多线程环境下，即使添加了条件判断，仍然存在竞态条件的问题。多个线程同时对共享资源 `num` 进行读写操作，由于操作的执行顺序不确定，可能导致程序出现不正确的结果。即使减 1 的线程在判断条件为真时不执行减 1 操作，但在它执行减 1 操作之前，其他线程可能已经将 `num` 的值减小到 1 以下，导致最终的结果是负数。

2. 忙等待：减 1 的线程中使用了 `continue` 语句，导致线程处于忙等待状态。即使 `num` 的值为 1 时，线程也会立即再次进入循环，而不会释放 CPU 资源。这会消耗大量的 CPU 资源，并且无法有效地阻止其他线程继续修改 `num` 的值。

➤ 解决方法：

在 `PlusMinus02` 里就添加判断操作，判断 `num` 是否大于一，只有满足时，再执行减操作，代码如下：

```
3 public class PlusMinus02 {
4     10 个用法
5     volatile int num;
6     //int num;
7
8     2 个用法
9     public void plusOne() {
10         synchronized (this) {
11             this.num = this.num + 1;
12             printNum();
13         }
14     }
15
16     8 个用法
17     public void minusOne() {
18         synchronized (this) {
19             /*this.num = this.num - 1;
20             printNum();*/
21             if (this.num > 1) { // 添加判断，当num已经为1时，停止减1操作
22                 this.num = this.num - 1;
23                 printNum();
24             }
25         }
26     }
27
28     2 个用法
29     > public void printNum() { System.out.println("num = " + this.num); }
30 }
```

- **Task7:** 使用 `wait` 和 `notify` 修改 4.2 的代码，实现原先相同功能，并将修改后的完整段和实验结果以及 CPU 占用情况截图附在实验报告中。

储备知识:

一、 等待唤醒机制

➤ 等待唤醒机制是多线程编程中一种用于线程间通信的机制。就是在一个线程进行了规定操作后，就进入等待状态 (`wait()`)，等待其他线程执行完他们的指定代码过后 再将其唤醒 (`notify()`) ;在有多线程进行等待时， 如果需要，可以使用 `notifyAll()`来唤醒所有的等待线程。`wait/notify` 就是线程间的一种协作机制。

➤ 等待唤醒机制方法：

1. `wait`: 线程不再活动，不再参与调度，进入 `wait set` 中，因此不会浪费 CPU 资源，也不会去竞争锁了，这时的线程状态即是 `WAITING`。它还要等着别的线程执行一个特别的动作，也即是“通知 (`notify`) ” 在这个对象上等待的线程从 `wait set` 中释放出来，重新进入到调度队列 (`ready queue`) 中。

2. `notify`: 选取所通知对象的 `wait set` 中的一个线程释放

3. `notifyAll`: 释放所通知对象的 `wait set` 上的全部线程

➤ 如果能获取锁，线程就从 `WAITING` 状态变成 `RUNNABLE` 状态；否则，从 `wait set` 出来，又进入 `entry set`，线程就从 `WAITING` 状态又变成 `BLOCKED` 状态。

➤ 调用 `wait` 和 `notify` 方法需要注意的细节：

1. `wait` 方法与 `notify` 方法必须要由同一个锁对象调用。因为：对应的锁对象可以通过 `notify` 唤醒使用同一个锁对象调用的 `wait` 方法后的线程。

2. `wait` 方法与 `notify` 方法是属于 `Object` 类的方法的。因为：锁对象可以是任意对象，而任意对象的所属类都是继承了 `Object` 类的。

3. `wait` 方法与 `notify` 方法必须要在同步代码块或者是同步函数中使用。因为：必须要通过锁对象调用这 2 个方法。

Task7 代码&结果&CPU 截图:

➤ 代码：

```
PlusMinus02.java x Main.java
1 package HW5.hw07;
2
3 2 个用法
3 public class PlusMinus02 {
4     7 个用法
4     private int num;
5     //volatile int num;
6     4 个用法
6     private final Object lock = new Object(); //加锁
7
8     1 个用法
8     public PlusMinus02(int n){
9         //添加构造方法，方便设置num
10        this.num=n;
11    }
12
13    1 个用法
13    public void plusOne() {
14        synchronized (lock) {
15            this.num = this.num + 1;
16            printNum();
17            //通知在lock等待的线程
18            lock.notify();
19        }
20    }
21
22    1 个用法
22    public void minusOne() {
23        synchronized (lock) {
24            if(this.num==1){
25                try{
26                    lock.wait();
27                }catch (InterruptedException e){
28                    e.printStackTrace();
29                }
30            }
31            this.num = this.num - 1;
32            printNum();
33        }
34    }
35
36    2 个用法
36    public void printNum() {
37        System.out.println("num = " + this.num);
38    }
39 }
40
```

➤ 实验输出结果：

与 Task4 输出结果完全相同，都是 999, 1000, 999, 998.....(递减一)...2,1,2,1,2,1....(一直在 1,2 之间

循环), 不会自动退出, 除非手动结束程序。

```
"C:\Program Files\
num = 999
num = 1000
num = 999
num = 998
num = 997
num = 996
num = 995
num = 994
num = 995
num = 994
num = 993
num = 992

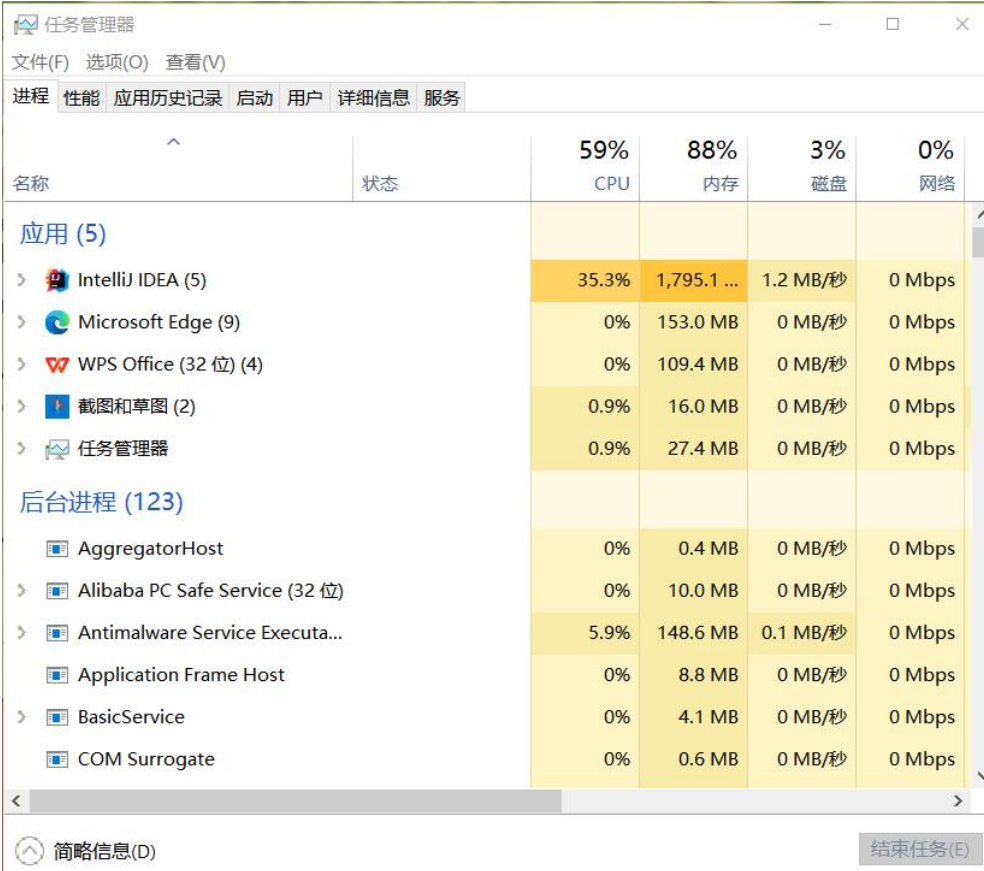
... ..

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
```

➤ CPU 占用情况截图:

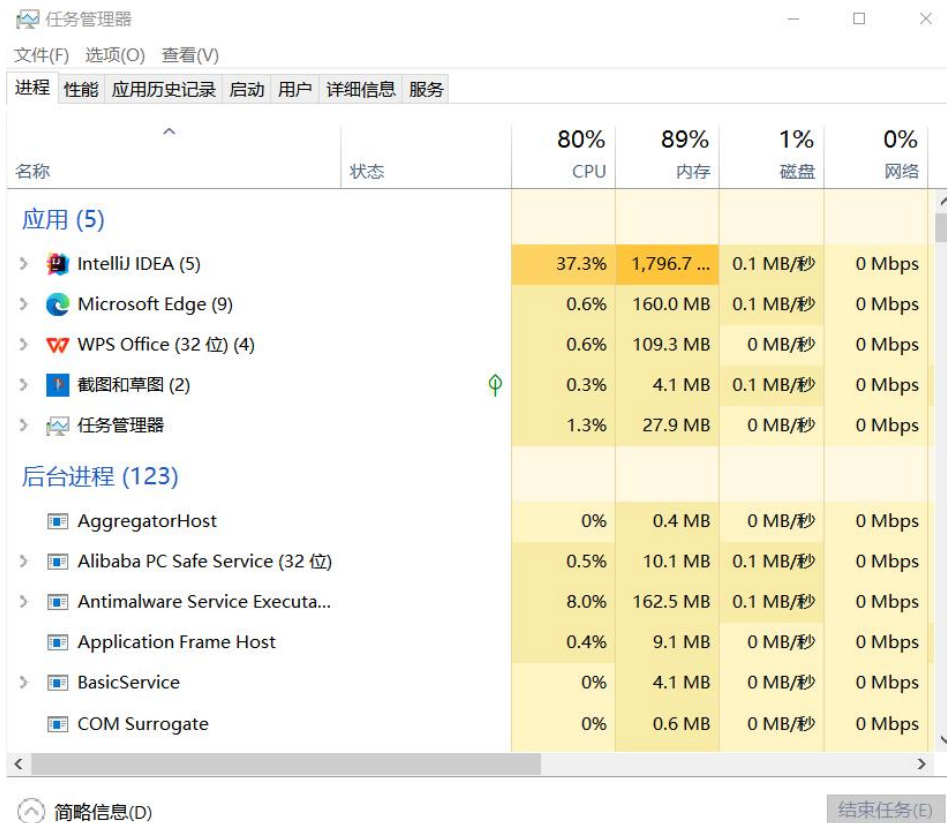
右键 windows 打开任务管理器, 查看 CPU 使用情况如下:

- 刚开始运行时, IDEA 迅速占用 CPU, IDEA 占有率和 CPU 总使用率都迅速提高



		59%	88%	3%	0%
		CPU	内存	磁盘	网络
应用 (5)					
IntelliJ IDEA (5)		35.3%	1,795.1 ...	1.2 MB/秒	0 Mbps
Microsoft Edge (9)		0%	153.0 MB	0 MB/秒	0 Mbps
WPS Office (32 位) (4)		0%	109.4 MB	0 MB/秒	0 Mbps
截图和草图 (2)		0.9%	16.0 MB	0 MB/秒	0 Mbps
任务管理器		0.9%	27.4 MB	0 MB/秒	0 Mbps
后台进程 (123)					
AggregatorHost		0%	0.4 MB	0 MB/秒	0 Mbps
Alibaba PC Safe Service (32 位)		0%	10.0 MB	0 MB/秒	0 Mbps
Antimalware Service Executa...		5.9%	148.6 MB	0.1 MB/秒	0 Mbps
Application Frame Host		0%	8.8 MB	0 MB/秒	0 Mbps
BasicService		0%	4.1 MB	0 MB/秒	0 Mbps
COM Surrogate		0%	0.6 MB	0 MB/秒	0 Mbps

- IDEA 占用 CPU 率逐步提升, 并且一度使 CPU 总使用率达到 80%左右



任务管理器

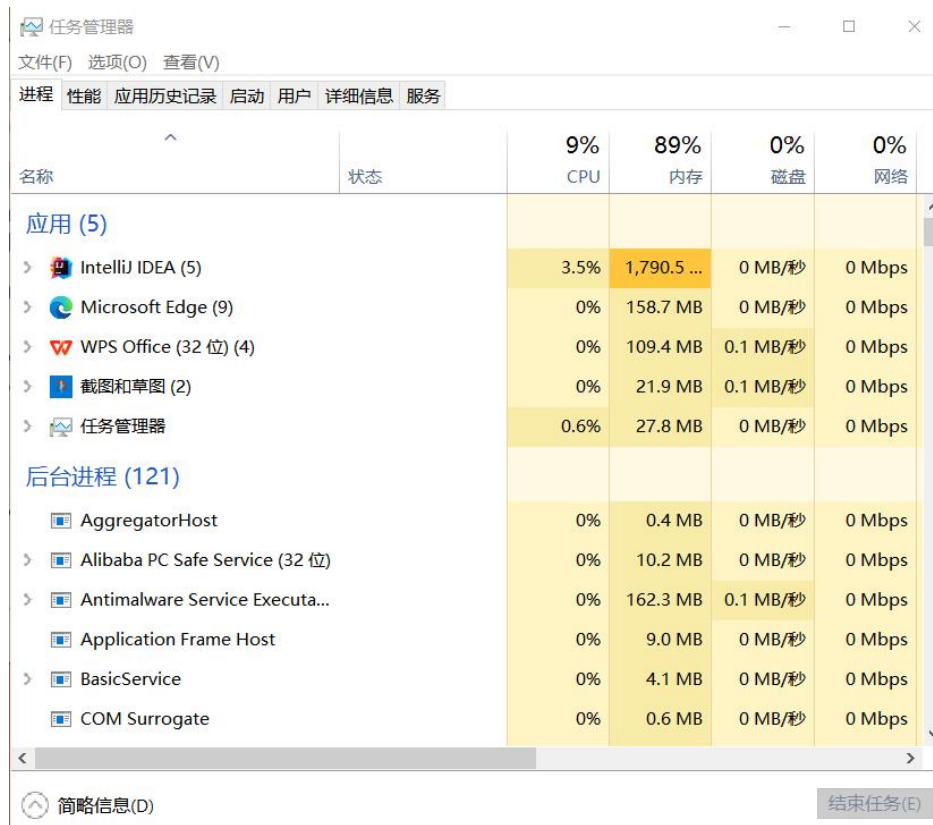
文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	状态	80% CPU	89% 内存	1% 磁盘	0% 网络
应用 (5)					
IntelliJ IDEA (5)		37.3%	1,796.7 ...	0.1 MB/秒	0 Mbps
Microsoft Edge (9)		0.6%	160.0 MB	0.1 MB/秒	0 Mbps
WPS Office (32 位) (4)		0.6%	109.3 MB	0 MB/秒	0 Mbps
截图和草图 (2)		0.3%	4.1 MB	0.1 MB/秒	0 Mbps
任务管理器		1.3%	27.9 MB	0 MB/秒	0 Mbps
后台进程 (123)					
AggregatorHost		0%	0.4 MB	0 MB/秒	0 Mbps
Alibaba PC Safe Service (32 位)		0.5%	10.1 MB	0.1 MB/秒	0 Mbps
Antimalware Service Executa...		8.0%	162.5 MB	0.1 MB/秒	0 Mbps
Application Frame Host		0.4%	9.1 MB	0 MB/秒	0 Mbps
BasicService		0%	4.1 MB	0 MB/秒	0 Mbps
COM Surrogate		0%	0.6 MB	0 MB/秒	0 Mbps

简略信息(D) 结束任务(E)

- 随着 num 逐渐递减至 1，IDEA 占用率也逐渐降低，占用 CPU 一度达到最小值，CPU 总使用率也降到 9%



任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	状态	9% CPU	89% 内存	0% 磁盘	0% 网络
应用 (5)					
IntelliJ IDEA (5)		3.5%	1,790.5 ...	0 MB/秒	0 Mbps
Microsoft Edge (9)		0%	158.7 MB	0 MB/秒	0 Mbps
WPS Office (32 位) (4)		0%	109.4 MB	0.1 MB/秒	0 Mbps
截图和草图 (2)		0%	21.9 MB	0.1 MB/秒	0 Mbps
任务管理器		0.6%	27.8 MB	0 MB/秒	0 Mbps
后台进程 (121)					
AggregatorHost		0%	0.4 MB	0 MB/秒	0 Mbps
Alibaba PC Safe Service (32 位)		0%	10.2 MB	0 MB/秒	0 Mbps
Antimalware Service Executa...		0%	162.3 MB	0.1 MB/秒	0 Mbps
Application Frame Host		0%	9.0 MB	0 MB/秒	0 Mbps
BasicService		0%	4.1 MB	0 MB/秒	0 Mbps
COM Surrogate		0%	0.6 MB	0 MB/秒	0 Mbps

简略信息(D) 结束任务(E)

- 最后 num 在 1 与 2 之间循环时间久了之后，IDEA 占用 CPU 以及 CPU 总使用率趋于平缓。CPU 总使用率稳定在 15%左右。

名称	状态	15% CPU	89% 内存	0% 磁盘	0% 网络
应用 (5)					
IntelliJ IDEA (5)		3.4%	1,789.8 ...	0.1 MB/秒	0 Mbps
Microsoft Edge (9)		0%	158.7 MB	0.1 MB/秒	0 Mbps
WPS Office (32 位) (4)		0.1%	109.3 MB	0 MB/秒	0 Mbps
截图和草图 (2)		1.0%	22.0 MB	0.1 MB/秒	0 Mbps
任务管理器		0.9%	27.9 MB	0 MB/秒	0 Mbps
后台进程 (122)					
AggregatorHost		0%	0.4 MB	0 MB/秒	0 Mbps
Alibaba PC Safe Service (32 位)		0.1%	10.2 MB	0 MB/秒	0 Mbps
Antimalware Service Executa...		0%	148.9 MB	0.1 MB/秒	0 Mbps
Application Frame Host		0.1%	9.1 MB	0 MB/秒	0 Mbps
BasicService		0%	4.1 MB	0 MB/秒	0 Mbps
COM Surrogate		0%	0.6 MB	0 MB/秒	0 Mbps

- Task8 (选做)：**编写多线程程序，模拟车站三个窗口同时卖票，包括购票（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。

储备知识：

一、Semaphore 集合

- Semaphore 是一种线程同步的工具，它主要用于控制同时访问某个特定资源的线程数量。
- 作用原理：Semaphore 维护了一个许可证集合，线程需要在访问资源之前从 Semaphore 获取许可证，许可证数量表示允许同时访问资源的线程数量。当一个线程获取到许可证时，Semaphore 会减少相应数量的许可证数量；当线程释放许可证时，Semaphore 会增加相应数量的许可证数量。

➤ 作用：

1. **控制并发线程数量：** 通过控制许可证的数量，Semaphore 可以限制同时访问某个资源的线程数

量，从而控制并发访问的程度，避免资源过度竞争导致的性能问题。

2. 线程间通信： Semaphore 的许可证数量可以被动态地调整，可以用于线程间的通信和协调，例如在生产者-消费者模型中控制生产者和消费者的速率。

3. 保护临界区： Semaphore 可以用来保护临界区，多个线程需要获取许可证才能进入临界区，确保在同一时间只有有限数量的线程可以访问临界区，从而避免竞态条件和数据不一致性。

➤ 代码：

```
© TicketsSell.java ×
1 package HW5.hw08;
2
3 import java.util.Scanner;
4 import java.util.concurrent.Semaphore;
5
6 public class TicketsSell {
7
8     6个用法
9     private static final int WINDOWSNUM = 3;
10    6个用法
11    private static int totalTickets = 100;
12    2个用法
13    private static Semaphore semaSale = new Semaphore(permits: 1);
14    2个用法
15    private static Semaphore semaRefund = new Semaphore(permits: 1);
16
17    2个用法
18    private static TicketSell[] sellers = new TicketSell[WINDOWSNUM];
19
20    public static void main(String[] args) {
21        Scanner scanner = new Scanner(System.in);
22        System.out.println("欢迎使用票务系统! 请输入指令: <窗口> <指令> <票数>, 或输入 '退出' 结束程序。");
23        System.out.println("窗口范围: 1-" + WINDOWSNUM);
24        System.out.println("指令: '购票' 或 '退票'");
25        System.out.println("票数范围: 1-20");
26
27        for (int i = 0; i < WINDOWSNUM; i++) {
28            sellers[i] = new TicketSell(i + 1);
29            new Thread(sellers[i]).start();
30        }
31
32        new Thread(() -> {
33            while (true) {
34                try {
35                    Thread.sleep(millis: 100000);
36                    addTicket(i: (int) (Math.random() * 5) + 1);
37                } catch (InterruptedException e) {
38                    e.printStackTrace();
39                }
40            }
41        }).start();
42
43        while (true) {
44            String input = scanner.nextLine().trim();
45            if (input.equalsIgnoreCase(anotherString: "退出")) {
46                System.out.println("谢谢使用, 再见!");
47            }
48        }
49    }
50}
```



```
42         break;
43     }
44
45     String[] inputs = input.split(regex: "\\s+");
46     if (inputs.length != 3) {
47         System.out.println("无效指令格式, 请重新输入。");
48         continue;
49     }
50
51     try {
52         int windowNumber = Integer.parseInt(inputs[0]);
53         String action = inputs[1];
54         int numTickets = Integer.parseInt(inputs[2]);
55
56         if (windowNumber < 1 || windowNumber > WINDOWSNUM) {
57             System.out.println("窗口号无效, 请输入范围内的窗口号。");
58             continue;
59         }
60
61         if (!action.equalsIgnoreCase(anotherString: "购票") && !action.equalsIgnoreCase(anotherString: "退票")) {
62             System.out.println("无效指令, 请输入 '购票' 或 '退票'。");
63             continue;
64         }
65
66         if (numTickets < 1 || numTickets > 20) {
67             System.out.println("票数无效, 请输入范围内的票数。");
68             continue;
69         }
70
71         TicketSell[] sellers = new TicketSell(WINDOWSNUM);
72         for (int i = 0; i < WINDOWSNUM; i++) {
73             sellers[i] = new TicketSell(n: i + 1);
74             new Thread(sellers[i]).start();
75         }
76
77         if (action.equalsIgnoreCase(anotherString: "购票")) {
78             sellers[windowNumber - 1].sellTicket(numTickets); // Sell tickets
79         } else {
80             sellers[windowNumber - 1].refundTicket(numTickets); // Refund tickets
81         }
82     } catch (NumberFormatException e) {
83         System.out.println("输入格式错误, 请输入整数。");
84     }
85 }
86
87 scanner.close();
88 }
89
90 1个用法
91 private static void addTicket(int n) {
92     totalTickets += n;
93     System.out.println("有新进票数: " + n + "票。到达售票处");
94 }
95
96 6个用法
97 static class TicketSell implements Runnable {
98     4个用法
99     private int winNum;
100
101     2个用法
102     public TicketSell(int n) {
103         this.winNum = n;
104     }
105
106     public void run() {
107         Thread.currentThread().setName("售票窗口-" + winNum);
108         //System.out.println(Thread.currentThread().getName() + "开始工作");
109     }
110 }
```



```

106
107     while (true) {
108         try {
109             Thread.sleep( millis: 1000);
110         } catch (InterruptedException e) {
111             e.printStackTrace();
112         }
113     }
114 }
115
116 1个用法
117 private void sellTicket(int n) {
118     try {
119         semaSale.acquire();
120         if (totalTickets >= n) {
121             totalTickets -= n;
122             System.out.println("窗口" + this.winNum + "接受购票订单, 出售" + n + "票。剩余可购总票数: " + totalTickets);
123         } else {
124             System.out.println("现无可购余票, 请耐心等待新票、退票或者遗憾离开。");
125         }
126     } catch (InterruptedException e) {
127         e.printStackTrace();
128     } finally {
129         semaSale.release();
130     }
131 }
132
133 1个用法
134 private void refundTicket(int n) {
135     try {
136         semaRefund.acquire();
137         totalTickets += n;
138         System.out.println("窗口" + this.winNum + "接受退票订单, 退回 " + n + "票。剩余可购总票数: " + totalTickets);
139     } catch (InterruptedException e) {
140         e.printStackTrace();
141     } finally {
142         semaRefund.release();
143     }
144 }

```

➤ 输出结果:

红框表示参数输入出错, 蓝框则为正确

```

"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java id
欢迎使用票务系统! 请输入指令: <窗口> <指令> <票数>, 或输入 '退出' 结束程序。
窗口范围: 1-3
指令: '购票' 或 '退票'
票数范围: 1-20
1 购票 4
窗口1接受购票订单: 出售4票。剩余可购总票数: 96
2 购票 25
票数无效, 请输入范围内的票数。
3 退票 2
窗口3接受退票订单: 退回 2票。剩余可购总票数: 98
8 购票 10
窗口号无效, 请输入范围内的窗口号。
1 2 3 4 5
无效指令格式, 请重新输入。
1 购票 4
无效指令, 请输入 '购票' 或 '退票'。
有新进票数: 4票。到达售票处
退出
谢谢使用, 再见!

```

过段时间, 就会有新进票

三、总结

拓展知识:

一、 线程状态

- 当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在 API 中 `java.lang.Thread.State` 这个枚举中给出了六种线程状态：

线程状态	导致状态发生条件
NEW(新建)	线程刚被创建，但是并未启动。还没调用start方法。
Runnable(可运行)	线程可以在java虚拟机中运行的状态，可能正在运行自己代码，也可能没有，这取决于操作系统处理器。
Blocked(锁阻塞)	当一个线程试图获取一个对象锁，而该对象锁被其他的线程持有，则该线程进入Blocked状态；当该线程持有锁时，该线程将变成Runnable状态。
Waiting(无限等待)	一个线程在等待另一个线程执行一个（唤醒）动作时，该线程进入Waiting状态。进入这个状态后是不能自动唤醒的，必须等待另一个线程调用notify或者notifyAll方法才能够唤醒。
Timed Waiting(计时等待)	同waiting状态，有几个方法有超时参数，调用他们将进入Timed Waiting状态。这一状态将一直保持到超时期满或者接收到唤醒通知。带有超时参数的常用方法有Thread.sleep、Object.wait。
Terminated(被终止)	因为run方法正常退出而死亡，或者因为没有捕获的异常终止了run方法而死亡。

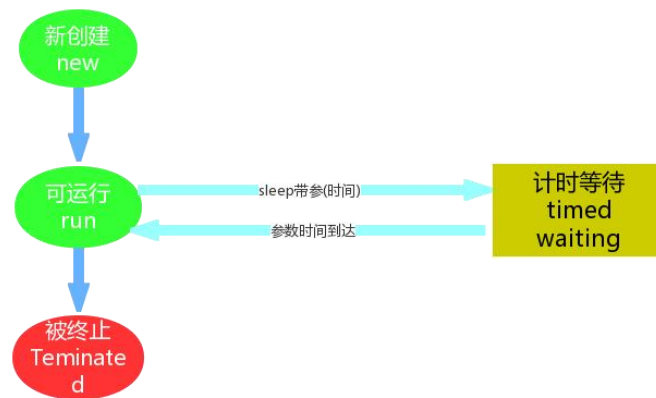
- Timed Waiting（计时等待）：一个正在限时等待另一个线程执行一个（唤醒）动作的线程处于这一状态。

1. 进入 TIMED_WAITING 状态的一种常见情形是调用的 sleep 方法，单独的线程也可以调用，不一定非要有协作关系。

2. 为了让其他线程有机会执行，可以将 Thread.sleep()的调用放线程 run()之内。这样才能保证该线程执行过程中会睡眠。

3. sleep 与锁无关，线程睡眠到期自动苏醒，并返回到 Runnable（可运行）状态。

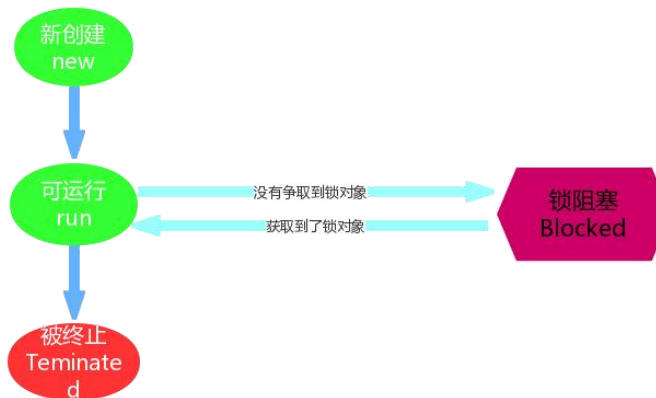
4. sleep()中指定的时间是线程不会运行的最短时间。因此，sleep()方法不能保证该线程睡眠到期后就开始立刻执行。



➤ **BLOCKED (锁阻塞)**：一个正在阻塞等待一个监视器锁（锁对象）的线程处于这一状态。

1. 由 Runnable 状态进入 Blocked 状态：假设线程 A 与线程 B 代码中使用同一锁，如果线程 A 获取到锁，线程 A 进入到 Runnable 状态，那么线程 B 就进入到 Blocked 锁阻塞状态。

2. Waiting 以及 Time Waiting 状态也会在某种情况下进入阻塞状态。



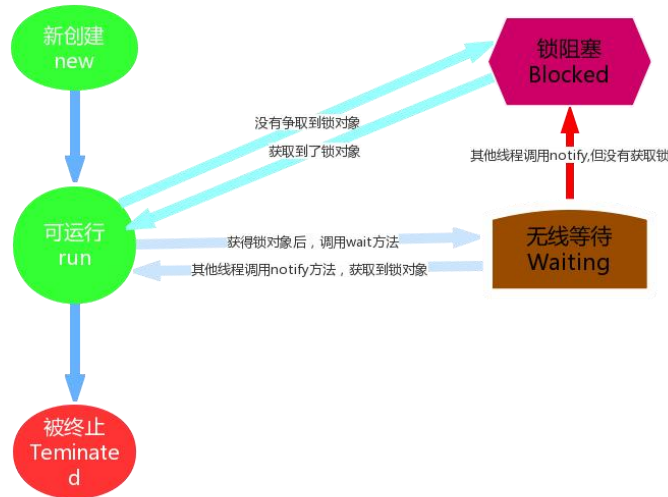
➤ **Waiting (无限等待)**：一个正在无限期等待另一个线程执行一个特别的（唤醒）动作的线程处于这一状态。

1. 一个调用了某个对象的 `Object.wait` 方法的线程会等待另一个线程调用此对象的 `Object.notify()` 方法 或 `Object.notifyAll()` 方法。

2. 可以理解为多个线程之间的协作关系，多个线程会争取锁，同时相互之间又存在协作关系。

3. 当多个线程协作时，比如 A，B 线程，如果 A 线程在 Runnable（可运行）状态中调用了 `wait()` 方法那么 A 线程就进入了 Waiting（无限等待）状态，同时失去了同步锁。假如这个时候 B 线程获取到了同步锁，在运行状态中调用了 `notify()` 方法，那么就会将无限等待的 A 线程唤醒。注意是唤醒，如果获取到锁对

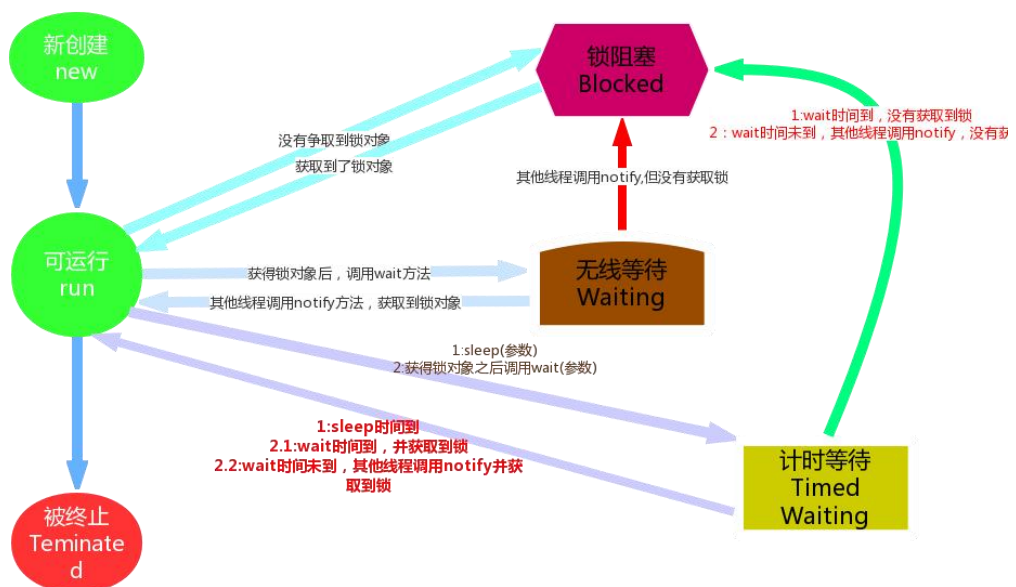
象，那么 A 线程唤醒后就进入 Runnable（可运行）状态；如果没有获取锁对象，那么就进入到 Blocked（锁阻塞状态）。



➤ Timed Waiting（计时等待）与 Waiting（无限等待）：

Waiting（无限等待）状态中 wait 方法是空参的，而 timed waiting（计时等待）中 wait 方法是带参的。这种带参的方法，其实是一种**倒计时操作**，如果没有得到（唤醒）通知，那么线程就处于 Timed Waiting 状态,直到倒计时完毕自动醒来；如果在倒计时期间得到（唤醒）通知，那么线程从 Timed Waiting 状态立刻唤醒。

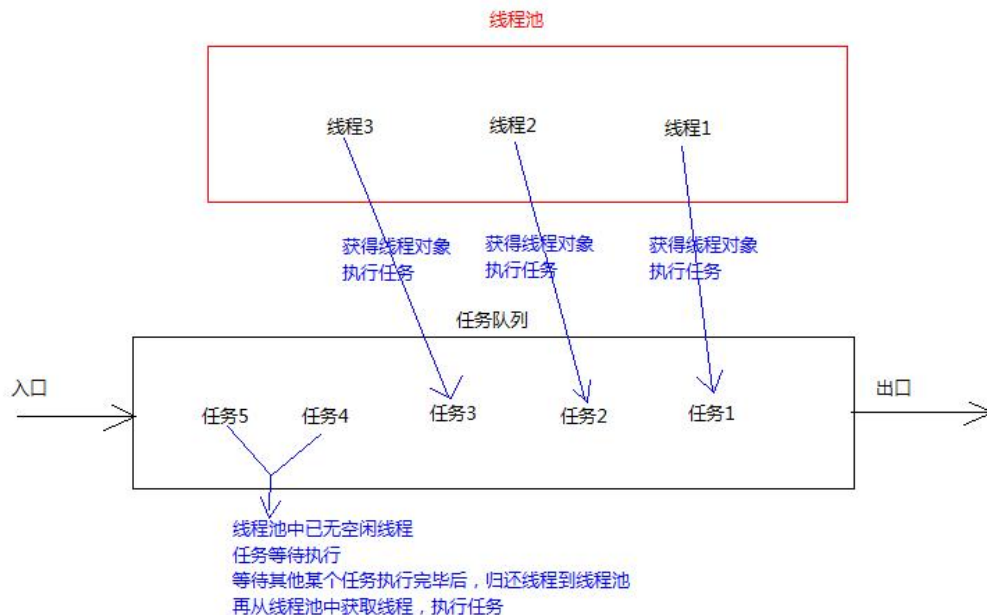
➤ 总体线程状态图示：



二、线程池

➤ 概念：一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。

➤ 图示：



➤ 合理使用的好处：

1. 降低资源消耗。减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。
2. 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。
3. 提高线程的可管理性。可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约 1MB 内存，线程开的越多，消耗的内存也就越大，最后死机)。

➤ 使用方法：

1. 接口：Java 里面线程池的顶级接口是 `java.util.concurrent.Executor`，但是严格意义上讲 `Executor` 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 `java.util.concurrent.ExecutorService`。

2. 在 `java.util.concurrent.Executors` 线程工厂类里面提供了一些静态工厂，生成一些常用的线程

池。官方建议使用 **Executors 工程类**来创建线程池对象。

3. 创建线程池的方法：public static ExecutorService newFixedThreadPool(int nThreads) : 返回线程池对象。(创建的是有界线程池,也就是池中的线程个数可以指定最大数量)

4. 使用线程池的方法：public Future<?> submit(Runnable task) :获取线程池中的某一个线程对象，并执行。（Future 接口：用来记录线程任务执行完毕后产生的结果）

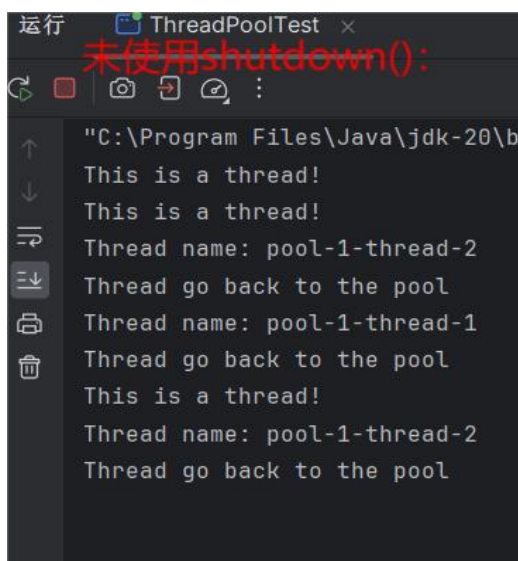
5. 使用线程池中线程对象的步骤：

- 1) 创建线程池对象。
- 2) 创建 Runnable 接口子类对象。(task)
- 3) 提交 Runnable 接口子类对象。(take task)
- 4) 关闭线程池(一般不做)。

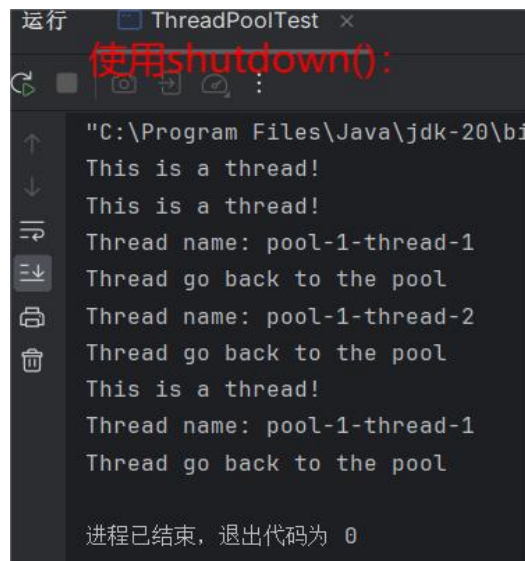
➤ 示例代码：

```
ThreadPoolTest.java x
7 public static void main(String[] args) {
8     //自己创建线程对象
9     /*new Thread()->{
10         System.out.println("This is a thread!");
11         try{
12             Thread.sleep(1000);
13         }catch (InterruptedException e){
14             e.printStackTrace();
15         }
16         System.out.println("Thread name: "+Thread.currentThread().getName());
17         System.out.println("Thread go back to the pool");
18     }).start();*/
19     //通过线程池获取线程对象
20     //创建线程池对象
21     ExecutorService service= Executors.newFixedThreadPool( nThreads: 2);
22     Runnable r =()->{
23         System.out.println("This is a thread!");
24         try{
25             Thread.sleep( millis: 1000);
26         }catch (InterruptedException e){
27             e.printStackTrace();
28         }
29         System.out.println("Thread name: "+Thread.currentThread().getName());
30         System.out.println("Thread go back to the pool");
31     };
32     service.submit(r); //获取第一个
33     service.submit(r); //获取第二个
34     service.submit(r);
35     //submit方法调用结束后，程序并不终止，是因为线程池控制了线程的关闭。将使用完的线程又归还到了线程池中
36
37     // 关闭线程池
38     //service.shutdown();
39 }
```

- 输出结果：没有手动 shutdown()时，程序并不会自己结束；调用 shutdown()后，可以自动退出程序。



```
运行 ThreadPoolTest x
未使用shutdown():
"C:\Program Files\Java\jdk-20\b
This is a thread!
This is a thread!
Thread name: pool-1-thread-2
Thread go back to the pool
Thread name: pool-1-thread-1
Thread go back to the pool
This is a thread!
Thread name: pool-1-thread-2
Thread go back to the pool
```



```
运行 ThreadPoolTest x
使用shutdown():
"C:\Program Files\Java\jdk-20\bi
This is a thread!
This is a thread!
Thread name: pool-1-thread-1
Thread go back to the pool
Thread name: pool-1-thread-2
Thread go back to the pool
This is a thread!
Thread name: pool-1-thread-1
Thread go back to the pool
进程已结束，退出代码为 0
```

总结:

通过本次实验，让我加深对 Java 多线程编程的理解和应用。学会使用常用的方式创建线程，并了解如何使用 join()、yield() 等方法对线程进行控制。能够使用 volatile 和 synchronized 关键字进行线程同步，以及使用 wait() 和 notify() 方法进行线程交互。

这两次的 task 主要包括以下几个方面：

- 使用常用的两种方式创建线程：通过继承 Thread 类和实现 Runnable 接口，分别创建线程，我还拓展了其他的一些便捷方式。
- 使用 join()、yield() 等方法对线程进行控制：了解和掌握 join() 方法阻塞调用线程、yield() 方法暂停当前线程的功能。
- 使用 volatile 和 synchronized 关键字进行线程同步：了解 volatile 关键字保证变量的可见性，synchronized 关键字实现线程的同步访问。还有一些其他的线程同步或者控制的技巧。
- 使用 wait() 和 notify() 方法进行线程交互：通过 wait() 方法使线程进入等待状态，notify() 方法唤醒等待线程，实现线程的交互操作。

总之，我深入理解了 Java 多线程编程的重要性和基本原理，掌握了创建线程、控制线程、同步线程和线程交互等核心技术。同时，我也意识到多线程编程在实际项目中的广泛应用，对提高程序的并发性能和响应速度具有重要意义。