Lab5: Java多线程编程2

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

1.1 synchronized关键字

- synchronized是Java中解决并发问题的方法之一，其能达成以下效果：
  - 原子性：确保线程互斥的访问同步代码；
  - 可见性：保证共享变量的修改能够及时可见，在Java内存模型中，不同线程拥有自己的本地内存，而本地内存是主内存的副本。如果线程修改了本地内存而没有去更新主内存，那么就无法保证可见性。synchronized在解锁前会将本地内存修改的内容刷新到主内存中，而在加锁前则将会清空工作内存中此变量的值，并重新从主内存读取读取保证了可见性；
  - 有序性：有效解决重排序问题，一个解锁操作先行发生于后面对同一个锁的加锁操作。
- synchronized表示当前线程单独占有对象object1(即锁对象，锁对象可以是任意对象，实际上锁的是每个对象拥有的Object Monitor)，如果有其他线程试图占有object1，就会等待，直到当前线程释放对object1的占用。

```
Object object1 = new Object();
synchronized (object1){
  //此处的代码只有占有了object1后才可以执行
}
```

- synchronized同样也可以修饰整个方法，被修饰的方法称为同步方法，其作用的范围是整个方法，如上所示。

```
public synchronized void functionName(){}
```

- 可重入性：所谓可重入就是说某个线程已经获得某个锁，可以再次获取锁而不会出现死锁。 可重入性在尝试获取对象锁时，如果当前线程已经拥有了此对象的锁，则把锁的计数器加一。

```
class Test {
    public static void main(String[] args) {
        Test t = new Test();
        synchronized(t) {
            synchronized(t) {
                System.out.println("made it!");
            }
        }
    }
}
```

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。

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。

```
public class TestMax {
    public static void main(String[] args) throws InterruptedException {
        Res res = new Res();

        int threadNum = 3;
        Thread[] threads = new Thread[threadNum];

        for (int i = 0; i < threadNum; i++) {
            threads[i] = new Thread(new MyThread(i, res));
        }

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

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

        System.out.println(res.max_idx);
    }
}

class MyThread implements Runnable {

    static int[] seeds = {1234567, 2345671, 3456712};

    MyThread(int id, Res _res) {
```

```
        Random r = new Random(seeds[id]);
        list = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
            list.add(r.nextInt(10000));
        }
        res = _res;
    }

    @Override
    public void run() {
        //TODO
    }

    ArrayList<Integer> list;
    Res res;
}

class Res {
    int max_idx;
}
```

1.2 死锁

- 死锁是因为代码设计的原因导致的系统问题
- 且在java中如果使用synchronized解决同步问题，由于不可抢占条件，可能出现多线程互相阻塞等待，无法主动或者被动停止阻塞，没有办法自动解锁。

1.2.1 代码验证

- 有如下TestDeadLock测试类，运行main函数可观察到死锁现象

```
public class TestDeadLock {
    public static void main(String[] args) throws InterruptedException {
        PlusMinus plusMinus1 = new PlusMinus();
        plusMinus1.num = 1000;
        PlusMinus plusMinus2 = new PlusMinus();
        plusMinus2.num = 1000;
        MyThread thread1 = new MyThread(plusMinus1, plusMinus2, 1);
        MyThread thread2 = new MyThread(plusMinus1, plusMinus2, 2);

        Thread t1 = new Thread(thread1);
        Thread t2 = new Thread(thread2);

        t1.start();
        t2.start();

        t1.join();
        t2.join();
```

```
    }
}

class MyThread implements Runnable {

    @Override
    public void run() {
        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");
                }
            }
        }else if (tid == 2) {
            synchronized (pm2) {
                System.out.println("thread" + tid + "正在占用 plusMinus2");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("thread" + tid + "试图继续占用 plusMinus1");
                System.out.println("thread" + tid + "等待中...");
                synchronized (pm1) {
                    System.out.println("thread" + tid + "成功占用了 plusMinus1");
                }
            }

        }
    }

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

    PlusMinus pm1;
    PlusMinus pm2;
    int tid;
}
```

```
class PlusMinus {
    public int num;

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

    public void minusOne() {
        num = num - 1;
    }

    public int printNum() {
        return num;
    }
}
```

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。

2. 线程交互

- 在多线程的场景下，线程间可能涉及交互。
- 比如对于一个int变量，有两个线程对它操作，一个每次加一，一个每次减一。对于减一的线程，如果它发现变量值为一时就停止它的操作，直到变量大于一才继续减。
- 现有`PlusMinusOne`类

```
class PlusMinusOne {
    volatile int num;

    public void plusOne() {
        synchronized (this) {
            this.num = this.num + 1;
            printNum();
        }
    }

    public void minusOne() {
        synchronized (this) {
            this.num = this.num - 1;
            printNum();
        }
    }

    public void printNum() {
        System.out.println("num = " + this.num);
    }
}
```

- 编写`TestInteract`测试类

```
public class TestInteract {
    public static void main(String[] args) {
        PlusMinusOne pmo = new PlusMinusOne();
        pmo.num = 50;

        Thread t1 = new Thread() {
            public void run() {
                while (true) {
                    while (pmo.num == 1) {
                        continue;
                    }
                    pmo.minusOne();
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

            }
        };
        t1.start();
        Thread t2 = new Thread() {
            public void run() {
                while (true) {
                    pmo.plusOne();
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };
        t2.start();
    }
}
```

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。

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。

2.1 使用wait和notify

- 跑了上面代码，可以发现上述写法不是好的解决方式，因为会大量占用CPU，导致整体性能下降
- `wait()`方法和`notify()`、`notifyAll()`方法，是Object上的方法。
- `wait()`:让占用了这个同步对象的线程，临时释放当前的占用，并且等待。 可以看出wait的调用应在synchronized块里，且前提是先获得了锁。
- `notify()`:唤醒一个等待在这个同步对象上的线程。
- `notifyAll()`:唤醒所有等待在这个同步对象上的线程。
- 以下代码通过`wait()`和`notifyAll()`实现了：
  - 线程t1可以调用`getProduct()`从`productQueue`中获取product。如果队列为空，则`getProduct()`应该等待，直到队列中至少有一个product时再返回。
  - 线程t2可以调用`addProduct()`不断往队列中添加product。

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。

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?

```
import java.util.LinkedList;
import java.util.Queue;

public class Test {
    public static void main(String[] args) {
        ProductFactory pf = new ProductFactory();
        Thread t1 = new Thread() {
            public void run() {
                while (true) {
                    try {
                        String s = pf.getProduct();
                        System.out.println("t1 get product: " + s);
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

            }
        };
        Thread t2 = new Thread() {
            public void run() {
                while (true) {
                    try {
                        String s = "product";
                        pf.addProduct(s);
                        System.out.println("t2 add product: " + s);
```

```
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    };
    t1.start();
    t2.start();
  }
}

class ProductFactory {
    Queue<String> productQueue = new LinkedList<>();

    public synchronized void addProduct(String s) {
        productQueue.add(s);
        this.notifyAll(); // 唤醒所有在this锁等待的线程
    }

    public synchronized String getProduct() throws InterruptedException {
        while (productQueue.isEmpty()) {
            // 释放this锁
            this.wait();
            // 重新获取this锁
        }
        return productQueue.remove();
    }
}
```

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。