Lab4: Java多线程编程1

一、实验目的

- 熟悉Java多线程编程
- 熟悉并掌握线程创建和线程间交互

二、实验任务

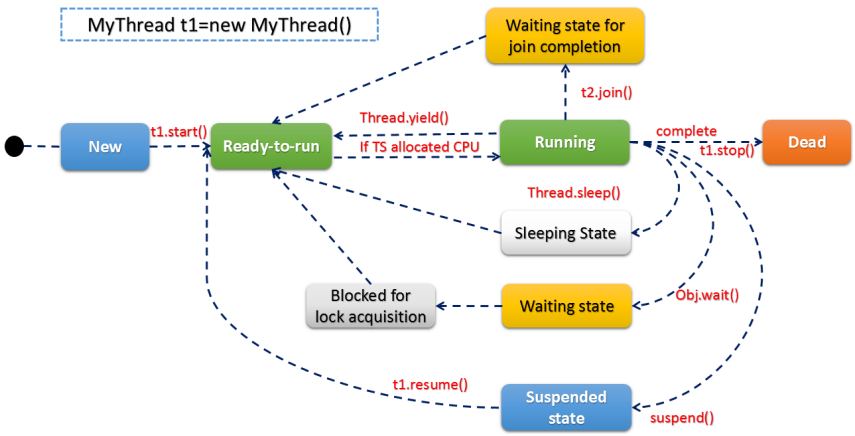
- 熟悉创建线程方法，继承线程类，实现Runnable接口 (匿名类不涉及)
- 使用`sleep()`、`join()`、`yield()`方法对线程进行控制
- 初步接触多线程编程

三、使用环境

- IntelliJ IDEA
- JDK 版本: Java 19

四、实验过程

Java thread life cycle:



1. 创建线程

1.1 继承Thread类创建线程

通过继承 **Thread** 类来创建并启动多线程的一般步骤如下：

1. 定义Thread类的子类，并重写该类的`run()`方法，该方法的方法体就是线程需要完成的任务，即线程的执行体。
2. 创建Thread子类的实例，创建一个线程对象。
3. 调用线程的`start()`方法，启动线程。

示例代码如下所示：

```
class ThreadTest01 extends Thread{

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }

    public static void main(String[] args){
        ThreadTest01 mthread1 = new ThreadTest01();
        ThreadTest01 mthread2 = new ThreadTest01();
        ThreadTest01 mthread3 = new ThreadTest01();

        mthread1.start();
        mthread2.start();
        mthread3.start();
    }
}
```

Task1: 使用继承Thread类的方式，编写ThreadTest类，改写`run()`方法，方法逻辑为每隔1秒打印`Thread.currentThread().getId()`，循环10次。实例化两个ThreadTest对象，并调用`start()`方法，代码及运行结果附在实验报告中。

1.2 实现Runnable接口创建线程

通过实现 **Runnable** 接口创建并启动线程一般步骤如下：

1. 定义Runnable接口的实现类，一样要重写`run()`方法，这个`run()`方法和Thread中的`run()`方法一样是线程的执行体。
2. 创建Runnable实现类的实例，并用这个实例作为Thread的target来创建Thread对象，这个Thread对象才是真正的线程对象。
3. 第三步依然是通过调用线程对象的`start()`方法来启动线程。

示例代码如下所示：

```
public class ThreadTest02 implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }

   public static void main(String[] args){
       ThreadTest02 tr = new ThreadTest02();
       Thread thread1 = new Thread(tr, "线程1");
       Thread thread2 = new Thread(tr, "线程2");
       Thread thread3 = new Thread(tr, "线程3");

       thread1.start();
```

```
        thread2.start();
        thread3.start();
    }
}
```

Task2: 给出以下`BattleObject`、`Battle`、`TestBattle`类，请改写`Battle`类，实现Runnable接口，`run()`方法逻辑为让`bo1`调用`attackHero(bo2)`，直到`bo2`的状态为`isDestoryed()`，请完成代码后使用`TestBattle`进行测试，将实现代码段及运行结果附在实验报告中。

```
public class BattleObject {
    public String name;
    public float hp;
    public int attack;
    public void attackHero(BattleObject bo) {
        try {
            // 每次攻击暂停
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        bo.hp -= attack;
        System.out.printf("%s 正在攻击 %s, %s 的耐久还剩 %.2f\n", name,
                bo.name, bo.name, bo.hp);
        if (bo.isDestoryed())
            System.out.println(bo.name + "被消灭。");
    }
    public boolean isDestoryed() {
        return 0 >= hp;
    }
}
```

```
public class Battle{
    private BattleObject bo1;
    private BattleObject bo2;

    public Battle(BattleObject bo1, BattleObject bo2) {
        this.bo1 = bo1;
        this.bo2 = bo2;
    }
    // TODO
}
```

```
public class TestBattle {
    public static void main(String[] args) {
        BattleObject bo1 = new BattleObject();
        bo1.name = "Object1";
        bo1.hp = 600;
        bo1.attack = 50;
        BattleObject bo2 = new BattleObject();
        bo2.name = "Object2";
        bo2.hp = 500;
        bo2.attack = 40;
        // TODO
    }
}
```

2. 线程控制

2.1 线程join

`join()`是使当前线程暂停执行，等待调用该方法的线程结束后再继续执行本线程，可以实现一个线程等待另一个线程执行完毕以后再执行，例如，若在A线程中调用了B线程的`join()`方法，只有当B线程执行完毕时，A线程才能继续执行。

```
...
public static void main(String[] args) throws InterruptedException {
    ThreadTest01 thread1 = new ThreadTest01();
    thread1.start();
    thread1.join();
    ...
}
...
```

`join()`可用作同步，`join()`和`start()`调用顺序问题：`join()`方法必须在线程`start()`方法调用之后调用才有意义。一个线程都还未开始运行，同步是不具有任何意义的。

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

```
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");
```

```
        // TODO
    }
}
```

2.2 守护线程

Java中有一种线程只在后台运行，为其他线程提供服务，这种线程就是守护线程（Daemon Thread）。JVM判断程序是否执行结束的标准是所有的前台线程（用户线程）执行完毕了，而不管后台线程（守护线程）的状态。

```
public static void main(String[] args) {
    ThreadTest01 thread1 = new ThreadTest01();
    thread1.setDaemon(true);
    thread1.start();
}
```

守护线程的作用是为其他线程提供便利服务，如负责线程调度的线程，守护线程最典型的应用就是GC (垃圾回收器)。守护线程不应该去访问固有资源，如文件、数据库，因为不知在何时守护线程可能就结束了。

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。并将关键代码和结果写到实验报告中。

```
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{
        // TODO
        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");
            }
        }}
}
```

2.3 Thread.yield()

线程让步，用于正在执行的线程，使线程让出CPU资源，回到一个准备抢占cpu的线程队列。

```
class MyRunnable implements Runnable{
    @Override
    public void run(){
        for(int i = 1; i < 100; i++){
            ...
            Thread.yield();
        }}
}
```

`yield()`方法会让线程回到就绪状态，直至等到CPU重新分配资源。

3. 线程安全初探

3.1 可见性

Task5: 给出`TestVolatile`类，测试`main`方法，观察运行结果，并尝试分析结果。

```
// if variable is not volatile, this example may not be terminated
// but this behaviour may differ on some machines
class TestVolatile extends Thread{
    //volatile
//    volatile boolean sayHello = true;
    boolean sayHello = true;

    public void run() {
        long count=0;
        while (sayHello) {
            count++;
        }

        System.out.println("Thread terminated." + count);
    }

    public static void main(String[] args) throws InterruptedException {
        TestVolatile t = new TestVolatile();
        t.start();
        Thread.sleep(1000);
        System.out.println("after main func sleeping...");
        t.sayHello = false;
        t.join();
        System.out.println("sayHello set to " + t.sayHello);
    }
}
```

3.2 可见性&原子性

Task6: 给出`PlusMinus`、`TestPlus`、`Plus`三个类，描述`TestPlus`的`main`方法的运行逻辑，并多次运行，观察输出结果，并尝试分析结果。

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

```
public class TestPlus {
    public static void main(String[] args) throws InterruptedException {
        PlusMinus plusMinus = new PlusMinus();
        plusMinus.num = 0;

        int threadNum = 10;
        Thread[] plusThreads = new Thread[threadNum];

        for(int i=0;i<threadNum;i++){
            plusThreads[i] = new Plus(plusMinus);
        }

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

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

        System.out.println(plusMinus.printNum());
    }
}

class Plus extends Thread{
    Plus(PlusMinus pm){
        this.plusMinus = pm;
    }
    @Override
    public void run(){
```

```
        for(int i=0;i<10000;i++){
            plusMinus.plusOne();
        }
    }
    PlusMinus plusMinus;
}
```