多线程

分析

进程和线程的区别

在早期的DOS操作系统，一旦中毒，系统就会崩溃，因为DOS是单进程操作系统，同一时间只允许一个进行使用系统资源（CPU，内存，IO）。

现在的Windows操作系统，引入了多进程概念，在同一时间可以执行多个进程，所有中毒之后还可以继续使用。

进程再往下细分，就到了线程，线程是程序执行的最小单元。

java实现多线程的方式

> java语言是为数不多的支持多线程的编程语言
> 实现多线程一般使用以下三种方式：

1. 继承Thread类
2. 实现Runnable接口（重点）
3. 实现Callable接口

继承Thread类

Java为我们提供了一个线程支持类叫做Thread，该类的子类叫做线程类，可以实现多线程操作：

```
class MyThread1 extends Thread{

}
```

此时，该线程类并没有可执行的代码，需要重写Thread类中的run()方法：

```
class MyThread1 extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("i="+i);
        }
    }
}
```

启动线程：

并不是直接调用run()方法，而是调用`start()`才叫做启动线程：

```
MyThread1 mt1 = new MyThread1();
MyThread1 mt2 = new  MyThread1();
mt1.start();
mt2.start();
```

此时我们发现，执行的结果并不是每次都保持一致，说明线程一旦启动，很难控制，线程执行的先后顺序，取决于CPU。CPU在同一个时间点只能执行一个线程

实现Runnable接口

由于java中单继承的特性，如果一个类一旦继承Thread类，就不可以继承其他类，所有java提供了另外一种实现多线程的方式：

```
class MyRunnable implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName()+"--i="+i);
        }
    }

}
```

此时需要注意，启动一个线程必须通过Thread类中的start()方法，观察Thread类中的构造方法：

```
public Thread(Runnable target)
```

发现可以接收Runnable对象，所有我们通过以下方式启动：

```
MyRunnable runnable = new MyRunnable();
Thread t1 = new Thread(runnable);
Thread t2 = new Thread(runnable);
t1.start();
t2.start();
```

此时，线程成功启动。

实现Callable接口

和以上两种方式不同，次方法线程会有一个返回值

```
class MyCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        Thread.sleep(2000);
        return "Hello";
    }

}
```

启动并获取返回值：

```
MyCallable callable = new MyCallable();
FutureTask<String> futureTask =
        new FutureTask<String>(callable);
new Thread(futureTask).start();
System.out.println("线程启动。。。");
try {
    //获取线程的返回值
    System.out.println(futureTask.get());
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

执行结果：

```
线程启动。。。
Hello
```

执行之后我们发现，会在两秒后输出call 方法的返回值

线程的命名和取得

命名

1. 通过Thread类中定义的setName方法:`public final synchronized void setName(String name)`
2. 通过Thread类的构造方法
    - 一个参数：`public Thread(String name)`
    - 两个参数：`public Thread(Runnable target, String name)`

例子

```
MyThread mt1 = new MyThread("线程A");
MyThread mt2 = new MyThread("线程B");
mt1.start();
mt2.start();

MyRunnable runnable = new MyRunnable();
Thread t1 = new Thread(runnable);
Thread t2 = new Thread(runnable, "线程B");
t1.setName("线程A");
t1.start();
t2.start();
```

取得

调用Thread类中的getName()方法：

1. 当通过继承Thread类实现多线是可以直接调用getName()方法
2. 当通过实现Runnable接口实现多线程时通过Thread类中的静态方法currentThread()，先获取当前线程，然后在

getName();

例子

```
getName();
Thread.currentThread().getName();
```

线程生命周期

> 与人有生老病死一样，线程也同样要经历新建、就绪、运行(活动)、阻塞和死亡五种不同的状态。这五种状态都可以通过Thread类中的方法进行控制

新建状态(New Thread)

在Java 语言中使用new 操作符创建一个线程后，该线程仅仅是一个空对象，它具备了线程的一些特征，但此时系统没有为其分配资源，这时的线程处于创建状态。

线程处于创建状态时，可通过Thread类的方法来设置线程各种属性，如线程的优先级（setPriority）、线程名(setName)和线程的类型（setDaemon）等。

就绪状态(Runnable)

使用start() 方法启动一个线程后，系统为该线程分配了除CPU 外的所需资源，使该线程处于就绪状态。此外，如果某个线程执行了yield() 方法，那么该线程会被暂时剥夺CPU 资源，重新进入就绪状态。

运行状态(Running)

Java运行系统通过调度选中一个处于就绪状态的线程，使其占有CPU 并转为运行状态。此时，系统真正执行线程的run() 方法。

可以通过Thread类的isAlive方法来判断线程是否处于就绪/运行状态：当线程处于就绪/运行状态时，isAlive返回true，当isAlive返回false时，可能线程处于阻塞状态，也可能处于停止状态。

阻塞状态(Blocked)

一个正在运行的线程因某些原因不能继续运行时，就进入阻塞状态

死亡状态(Dead)

线程在run() 方法执行结束后进入死亡状态。此外，如果线程执行了interrupt()或stop() 方法，那么它也会以异常退出的方式进入死亡状态。

线程的休眠

> 所谓线程的休眠就是让线程的执行速度变慢。休眠的方法：

```
public static void sleep(long millis) throws InterruptedException
```

例子

代码

```
public class ThreadDemo1 {

    public static void main(String[] args) {
        MyThread mt1 = new MyThread("线程A");
        mt1.start();
    }
}

class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(1000);//休眠1秒
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(getName() + "--i=" + i);
        }
    }
}
```

执行结果

```
线程A--i=0
线程A--i=1
线程A--i=2
线程A--i=3
线程A--i=4
```

分析

此时我们发现，因为所以执行到run方法的线程都会休眠，所以执行速度变慢

线程优先级

之前我们说过，线程启动后并不是立刻执行的，需要等待CPU进行调度，但是调度的时候本身也是存在‘优先’级的，优先级高的线程有可能被先执行。

设置方法

如果想要设置线程的优先级，可以使用：

```
public final void setPriority(int newPriority)
```

此方法需要接收一个整形的数字，这个数字可以设置以下三个内容：

1. 最高优先级：`public static final int MAX_PRIORITY`
2. 中等优先级：`public static final int NORM_PRIORITY`
3. 最低优先级：`public static final int MIN_PRIORITY`

例子

线程类代码

```
class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            try {
                Thread.sleep(1000);//休眠1秒
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(getName() + "--i=" + i);
        }
    }
}
```

直接启动

```
MyThread t1 = new MyThread("线程1");
MyThread t2 = new MyThread("线程2");
MyThread t3 = new MyThread("线程3");
t1.start();
t2.start();
t3.start();
```

执行结果如下（并不是每次执行都是如下结果）

```
线程1--i=0
线程2--i=0
线程3--i=0
线程1--i=1
线程3--i=1
线程2--i=1
线程1--i=2
线程3--i=2
线程2--i=2
```

设置优先级后启动

此时，我们为线程设置如下优先级

- 线程1-->最低
- 线程2-->中等
- 线程3-->最低

```
MyThread t1 = new MyThread("线程1");
MyThread t2 = new MyThread("线程2");
MyThread t3 = new MyThread("线程3");
t1.setPriority(Thread.MIN_PRIORITY);//最低
t2.setPriority(Thread.NORM_PRIORITY);//中等
t3.setPriority(Thread.MIN_PRIORITY);//最高
t1.start();
t2.start();
t3.start();
```

执行结果如下：

```
线程3--i=0
线程2--i=0
线程1--i=0
线程3--i=1
线程2--i=1
线程1--i=1
线程2--i=2
线程1--i=2
线程3--i=2
```

此时，多执行几次我们可以发现，并不是每次都是线程3先抢占到资源，说明了优先级高的线程只是有可能先被执行，并不是一定会先被执行

线程之间共享数据

> 模拟窗口售票操作，一共5张票，两个窗口同时售票

第一种方案

示例代码

```
public class ShareDataDemo {

    /*
     * 模拟窗口售票，一共5张票，两个窗口
     */
    public static void main(String[] args) {
        new TicketThread("1号窗口").start();
        new TicketThread("2号窗口").start();
    }

}
class TicketThread extends Thread{
    private int ticket = 5;//5张票
    public TicketThread(String name) {
        super(name);
    }
    @Override
    public void run() {
        while(true){
            System.out.println(getName()+"卖了第"+ticket--+"张票");
            if(ticket <= 0){
                break;
            }
        }
    }
}
```

执行结果

```
1号窗口卖了第5张票
2号窗口卖了第5张票
1号窗口卖了第4张票
2号窗口卖了第4张票
1号窗口卖了第3张票
1号窗口卖了第2张票
1号窗口卖了第1张票
2号窗口卖了第3张票
2号窗口卖了第2张票
2号窗口卖了第1张票
```

分析

此时，发现每一张票都被卖了两次，原因是，我们new了两个线程类，每一个线程类都有自己的ticket属性，也就是说，两个窗口各有5张票。

第二种方案

示例代码

```
public class ShareDataDemo {

    /*
     * 模拟窗口售票，一共5张票，两个窗口
     */
    public static void main(String[] args) {
        TicketRunnable runnable = new TicketRunnable();
        new Thread(runnable, "1号窗口").start();;
        new Thread(runnable, "2号窗口").start();
    }

}
class TicketRunnable implements Runnable{

    private int ticket = 5;//5张票

    @Override
    public void run() {
        while(true){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()+"卖了第"+ticket--+"张票");
            if(ticket <= 0){
                break;
            }
        }
    }
}
```

执行结果

```
1号窗口卖了第4张票
2号窗口卖了第5张票
2号窗口卖了第3张票
1号窗口卖了第2张票
2号窗口卖了第1张票
1号窗口卖了第0张票
```

分析

此时，发现并不是每张票都被卖了两次，基本上实现了进行之间的数据共享，但是偶尔还存在重复卖出或者卖出第0张的问题。

线程同步

为什么需要线程同步

- 线程同步是为了防止多个线程访问一个数据对象时，对数据造成破坏

- 线程的同步是保证多线程安全访问竞争资源的一种手段

synchronized关键字

- 同步语句块：

```
synchronized(锁定对象){
  //代码
}
```

- 同步方法：

```
访问修饰符 synchronized 返回值类型 方法名(参数列表){
  //方法体
}
```

同步和锁定

- Java中每个对象都有一个内置锁。
- 当程序运行到非静态的synchronized同步方法上时，自动获得与正在执行代码类的当前实例（this实例）有关的锁；当程序运行到synchronized同步代码块时，自动获得锁定对象的锁。
- 获得一个对象的锁也称为获取锁、锁定对象、在对象上锁定或在对象上同步。当程序运行到synchronized同步方法或代码块时该对象锁才起作用。
- 一个对象只有一个锁。所以，如果一个线程获得该锁，就没有其他线程可以获得锁，直到第一个线程释放锁。这也意味着任何其他线程都不能进入synchronized方法或代码块，直到该锁被释放。释放锁是指持锁线程退出了synchronized同步方法或代码块。

例子

在上述多个窗口同时售票的例子中，还存在问题需要使用同步才能修复。我们应该保证从判断剩余票数到售票的过程同一时间只有一个线程可以进行操作，方法如下：

使用同步语句块

```
public class ShareDataDemo {

    public static void main(String[] args) {
        TicketRunnable runnable = new TicketRunnable();
        new Thread(runnable, "1号窗口").start();
        new Thread(runnable, "2号窗口").start();
        new Thread(runnable, "3号窗口").start();
        new Thread(runnable, "4号窗口").start();
    }

}

class TicketRunnable implements Runnable {

    private int ticket = 10;// 5张票

    @Override
    public void run() {
        while(true){
            synchronized(this){
                if(ticket > 0){//1.判断票数是否可卖
                    try {
                        Thread.sleep(1000);//2.延时0.2秒
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    //3.售票
                    System.out.println(Thread.currentThread().getName()+"卖出了第"+ticket--+"张
票");
                }else{
                    break;
                }
            }
        }
    }
}
```

执行结果

```
1号窗口卖出了第10张票
4号窗口卖出了第9张票
3号窗口卖出了第8张票
3号窗口卖出了第7张票
2号窗口卖出了第6张票
2号窗口卖出了第5张票
3号窗口卖出了第4张票
4号窗口卖出了第3张票
4号窗口卖出了第2张票
4号窗口卖出了第1张票
```

使用同步方法

```
public class ShareDataDemo {

    public static void main(String[] args) {
        TicketRunnable runnable = new TicketRunnable();
        new Thread(runnable, "1号窗口").start();
        new Thread(runnable, "2号窗口").start();
        new Thread(runnable, "3号窗口").start();
        new Thread(runnable, "4号窗口").start();
    }

}

class TicketRunnable implements Runnable {

    private int ticket = 10;// 5张票

    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            sale();
        }
    }
    private synchronized void sale(){
        if(ticket > 0){//1.判断票数是否可卖
            try {
                Thread.sleep(1000);//2.延时0.2秒
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //3.售票
            System.out.println(Thread.currentThread().getName()+"卖出了第"+ticket--+"张票");
        }
    }
}
```

执行结果

```
1号窗口卖出了第10张票
1号窗口卖出了第9张票
1号窗口卖出了第8张票
1号窗口卖出了第7张票
4号窗口卖出了第6张票
2号窗口卖出了第5张票
3号窗口卖出了第4张票
2号窗口卖出了第3张票
2号窗口卖出了第2张票
2号窗口卖出了第1张票
```

总结

通过上面的例子可以发现，当添加同步之后，程序的执行速度会变慢，但是可以保证数据的安全，这种操作我们叫做线程安全的线程操作。反之，叫做非线程安全的线程操作

线程死锁

产生死锁的原因

当一个线程已经获取了对象1的锁，同时又想获取对象2的锁。而此时另一个线程当前已经持有了对象二的锁，而又想获取对象1的锁。这种互相等待对方释放锁的过程，会导致“死锁”。

例子

```
public class DeadLockDemo implements Runnable{

    private A a = new A();
    private B b = new B();

    public DeadLockDemo() {
        new Thread(this).start();
        b.say(a);
    }

    public static void main(String[] args) {
        new DeadLockDemo();
    }

    @Override
    public void run() {
        a.say(b);
    }
}
class A{
    public synchronized void say(B b){
        System.out.println("A说：你先放人，我就给钱！");
        b.get();
    }
    public synchronized void get(){
        System.out.println("A:救了人，付出了钱！");
    }
}
class B{
    public synchronized void say(A a){
        System.out.println("B说：你先给钱，我就放人！");
        a.get();
    }
    public synchronized void get(){
        System.out.println("B:放了人，拿到了钱！");
    }
}
```

程序执行之后会输出：

```
B说：你先给钱，我就放人！
A说：你先放人，我就给钱！
```

可以发现两个线程正在互相等待，等待对方持有的锁被释放，这种现象叫做死锁。

线程通讯

概述

> 让多个线程按照一定的次序来访问共享资源

Object类对多线程的支持

> Java提供了3个重要方法巧妙解决线程间的通信问题。这3个方法定义在Object类中，分别是：wait()、notify()和notifyAll()。

1. `wait()` :使调用该方法的线程释放共享资源的锁，然后从运行态退出，进入等待队列，直到被再次唤醒。
2. `notify()` :以唤醒等待队列中第一个等待同一共享资源的线程，并使该线程退出等待队列，进入可运行态。
3. `notifyAll()` :使所有正在等待队列中等待同一共享资源的线程从等待状态退出，进入可运行状态，此时，优先级最高的那个线程最先执行。

生产者和消费者的例子

产品类

```java
/**
 * 产品
 */
class Info {
    private String title;
    private String content;
    private boolean flag = true;
    //true：可以生产，不可以取走
    //false：可以取走，不可以生产

    public synchronized void set(String title, String content){
        //判断是否可以生产
        if(this.flag == false){//不可以生产
            try {
                this.wait();//等待消费者取走
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.title = title;
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.content = content;
        this.flag = false;//修改状态：可以取走
        this.notify();//唤醒其他线程
    }

    public synchronized void get(){
        //先判断是否可以取走
        if(this.flag == true){//不可以
            try {
                this.wait();//等待生产者生产
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(this.title+"--"+this.content);
        this.flag = true;//修改状态：可以生产
        this.notify();//唤醒其他线程
    }
}
```

生产者线程

```java
/**
 * 生产者
 */
class Productor implements Runnable {

    private Info info;

    public Productor(Info info) {
        this.info = info;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            if (i % 2 == 0) {// 偶数
                this.info.set("高鹏", "好学生一个");
            } else {// 奇数
                this.info.set("张三", "坏学生一个");
            }
        }
    }
}
```

消费者线程

```java
/**
 * 消费者
 */
class Customer implements Runnable {

    private Info info;

    public Customer(Info info) {
        this.info = info;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            this.info.get();
        }
    }
}
```

开始执行

```
public class Test {
    public static void main(String[] args) {
        Info info = new Info();
        new Thread(new Productor(info)).start();
        new Thread(new Customer(info)).start();
    }
}
```

执行结果

```
高鹏--好学生一个
张三--坏学生一个
高鹏--好学生一个
张三--坏学生一个
高鹏--好学生一个
张三--坏学生一个
高鹏--好学生一个
张三--坏学生一个
高鹏--好学生一个
张三--坏学生一个
...
```

分析

在上述代码中，我们通过为商品类添加了一个boolean的标记配合Object类定义的wait()和notify()方法实现了生产者每生产一个产品，消费者取走一个产品