**多线程之间实现通讯**

# 课程目标

多线程之间如何通讯

wait、notify、notifyAll()方法

lock

停止线程

守护线程

Join方法

优先级

Yield

# 多线程之间如何实现通讯

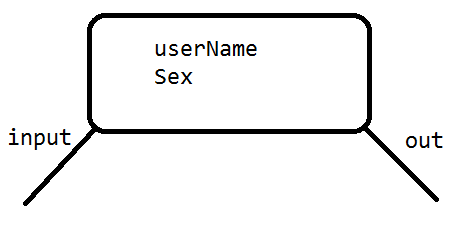
## 2.1 什么是多线程之间通讯？

多线程之间通讯，其实就是多个线程在操作同一个资源，但是操作的动作不同。

画图演示

2.2 多线程之间通讯需求

需求:第一个线程写入(input)用户，另一个线程取读取(out)用户.实现读一个，写一个操作。



2.3代码实现基本实现

### 2.3.1共享资源源实体类

|  |
| --- |
| **class** Res {  **public** String userSex;  **public** String userName;  } |

### 2.3.2输入线程资源

|  |
| --- |
| **class** IntThrad **extends** Thread {  **private** Res res;  **public** IntThrad(Res res) {  **this**.res = res;  }  @Override  **public** **void** run() {  **int** count = 0;  **while** (**true**) {  **if** (count == 0) {  res.userName = "余胜军";  res.userSex = "男";  } **else** {  res.userName = "小紅";  res.userSex = "女";  }  count = (count + 1) % 2;  }  }  } |

### 2.3.3输出线程

|  |
| --- |
| **class** OutThread **extends** Thread {  **private** Res res;  **public** OutThread(Res res) {  **this**.res = res;  }  @Override  **public** **void** run() {  **while** (**true**) {  System.***out***.println(res.userName + "--" + res.userSex);  }  }  } |

### 2.3.4运行代码

|  |
| --- |
| Res res = **new** Res();  IntThrad intThrad = **new** IntThrad(res);  OutThread outThread = **new** OutThread(res);  intThrad.start();  outThread.start(); |

### 2.3.5运行代码

|  |
| --- |
|  |

注意：数据发生错乱，造成线程安全问题

### 2.3.6解决线程安全问题

#### 2.3.6.1 IntThrad 加上synchronized

|  |
| --- |
| **class** IntThrad **extends** Thread {  **private** Res res;  **public** IntThrad(Res res) {  **this**.res = res;  }  @Override  **public** **void** run() {  **int** count = 0;  **while** (**true**) {  **synchronized** (res) {  **if** (count == 0) {  res.userName = "余胜军";  res.userSex = "男";  } **else** {  res.userName = "小紅";  res.userSex = "女";  }  count = (count + 1) % 2;  }  }  }  } |

#### 2.3.6.2 输出线程加上synchronized

|  |
| --- |
| **class** OutThread **extends** Thread {  **private** Res res;  **public** OutThread(Res res) {  **this**.res = res;  }  @Override  **public** **void** run() {  **while** (**true**) {  **synchronized** (res) {  System.***out***.println(res.userName + "--" + res.userSex);  }  }  }  } |

# 2.4 wait()、notify、notifyAll()方法

wait()、notify()、notifyAll()是三个定义在Object类里的方法，可以用来控制线程的状态。

这三个方法最终调用的都是jvm级的native方法。随着jvm运行平台的不同可能有些许差异。

* 如果对象调用了wait方法就会使持有该对象的线程把该对象的控制权交出去，然后处于等待状态。
* 如果对象调用了notify方法就会通知某个正在等待这个对象的控制权的线程可以继续运行。
* 如果对象调用了notifyAll方法就会通知所有等待这个对象控制权的线程继续运行。

|  |
| --- |
| **class** Res {  **public** String userSex;  **public** String userName;  //线程通讯标识  **public** **boolean** flag = **false**;  } |

|  |
| --- |
| **class** IntThrad **extends** Thread {  **private** Res res;  **public** IntThrad(Res res) {  **this**.res = res;  }  @Override  **public** **void** run() {  **int** count = 0;  **while** (**true**) {  **synchronized** (res) {  **if** (res.flag) {  **try** {  // 当前线程变为等待，但是可以释放锁  res.wait();  } **catch** (Exception e) {  }  }  **if** (count == 0) {  res.userName = "余胜军";  res.userSex = "男";  } **else** {  res.userName = "小紅";  res.userSex = "女";  }  count = (count + 1) % 2;  res.flag = **true**;  // 唤醒当前线程  res.notify();  }  }  }  } |

|  |
| --- |
| **class** OutThread **extends** Thread {  **private** Res res;  **public** OutThread(Res res) {  **this**.res = res;  }  @Override  **public** **void** run() {  **while** (**true**) {  **synchronized** (res) {  **if** (!res.flag) {  **try** {  res.wait();  } **catch** (Exception e) {  // **TODO**: handle exception  }  }  System.***out***.println(res.userName + "--" + res.userSex);  res.flag = **false**;  res.notify();  }  }  }  } |

|  |
| --- |
| **public** **class** ThreaCommun {  **public** **static** **void** main(String[] args) {  Res res = **new** Res();  IntThrad intThrad = **new** IntThrad(res);  OutThread outThread = **new** OutThread(res);  intThrad.start();  outThread.start();  }  } |

# 2.5 wait与wait区别?

对于sleep()方法，我们首先要知道该方法是属于Thread类中的。而wait()方法，则是属于Object类中的。

sleep()方法导致了程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。

在调用sleep()方法的过程中，线程不会释放对象锁。

而当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备

获取对象锁进入运行状态。

# JDK1.5-Lock

**在 jdk1.5 之后，并发包中新增了 Lock 接口(以及相关实现类)用来实现锁功能，Lock 接口提供了与 synchronized 关键字类似的同步功能，但需要在使用时手动获取锁和释放锁。**

## 3.1 、Lock写法:

|  |
| --- |
| Lock **lock** = **new** ReentrantLock();  **lock**.**lock**();  **try**{  *//可能会出现线程安全的操作*  }**finally**{  *//一定在finally中释放锁*  *//也不能把获取锁在try中进行，因为有可能在获取锁的时候抛出异常*  **lock**.ublock();  } |

## 3.2、Lock 接口与 synchronized 关键字的区别

**Lock 接口可以尝试非阻塞地获取锁 当前线程尝试获取锁。如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁。  
\*Lock 接口能被中断地获取锁 与 synchronized 不同，获取到锁的线程能够响应中断，当获取到的锁的线程被中断时，中断异常将会被抛出，同时锁会被释放。**

**Lock 接口在指定的截止时间之前获取锁，如果截止时间到了依旧无法获取锁，则返回。**

## 3.3 Condition用法

**Condition的功能类似于在传统的线程技术中的,Object.wait()和Object.notify()的功能,**

**代码:**

|  |
| --- |
| Condition condition = lock.newCondition();res. condition.await(); 类似waitres. Condition. Signal() 类似notify |

# 如何停止线程？

## 4.1 停止线程思路

  1.  使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。

    2.  使用stop方法强行终止线程（这个方法不推荐使用，因为stop和suspend、resume一样，也可能发生不可预料的结果）。

    3.  使用interrupt方法中断线程。

代码:

|  |
| --- |
| **class** StopThread **implements** Runnable {  **private** **boolean** flag = **true**;  @Override  **public** **synchronized** **void** run() {  **while** (flag) {  **try** {  wait();  } **catch** (Exception e) {  //e.printStackTrace();  stopThread();  }  System.***out***.println("thread run..");  }  }  /\*\*  \*  \* **@methodDesc**: 功能描述:(停止线程)  \* **@author**: 余胜军  \* **@param**:  \* **@createTime**:2017年8月20日 下午8:07:34  \* **@returnType**: void  \* **@copyright**:上海每特教育科技有限公司  \*/  **public** **void** stopThread() {  flag = **false**;  }  }  /\*\*  \*  \* **@classDesc**: 功能描述:(停止线程)  \* **@author**: 余胜军  \* **@createTime**: 2017年8月20日 下午8:05:25  \* **@version**: v1.0  \* **@copyright**:上海每特教育科技有限公司  \*/  **public** **class** StopThreadDemo {  **public** **static** **void** main(String[] args) {  StopThread stopThread1 = **new** StopThread();  Thread thread1 = **new** Thread(stopThread1);  Thread thread2 = **new** Thread(stopThread1);  thread1.start();  thread2.start();  **int** i = 0;  **while** (**true**) {  System.***out***.println("thread main..");  **if** (i == 300) {  // stopThread1.stopThread();  thread1.interrupt();  thread2.interrupt();  **break**;  }  i++;  }  }  } |

# 守护线程

Java中有两种线程，一种是用户线程，另一种是守护线程。

当进程不存在或主线程停止，守护线程也会被停止。

使用setDaemon(true)方法设置为守护线程

|  |
| --- |
| \*\*  \*  \* 什么是守护线程? 守护线程 进程线程(主线程挂了) 守护线程也会被自动销毁.  \*  \* **@classDesc**: 功能描述:(守护线程)  \* **@author**: 余胜军  \* **@createTime**: 2017年8月20日 下午8:55:58  \* **@version**: v1.0  \* **@copyright**:上海每特教育科技有限公司  \*/  **public** **class** DaemonThread {  **public** **static** **void** main(String[] args) {  Thread thread = **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **while** (**true**) {  **try** {  Thread.*sleep*(100);  } **catch** (Exception e) {  // **TODO**: handle exception  }  System.***out***.println("我是子线程...");  }  }  });  thread.setDaemon(**true**);  thread.start();  **for** (**int** i = 0; i < 10; i++) {  **try** {  Thread.*sleep*(100);  } **catch** (Exception e) {  }  System.***out***.println("我是主线程");  }  System.***out***.println("主线程执行完毕!");  }  } |

# join()方法作用

join作用是让其他线程变为等待

## 6.1需求:

创建一个线程，子线程执行完毕后，主线程才能执行。

|  |
| --- |
| **class** JoinThread **implements** Runnable {  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + "---i:" + i);  }  }  }  /\*\*  \*  \* **@classDesc**: 功能描述:(Join方法)  \* **@author**: 余胜军  \* **@createTime**: 2017年8月20日 下午9:23:30  \* **@version**: v1.0  \* **@copyright**:上海每特教育科技有限公司  \*/  **public** **class** JoinThreadDemo {  **public** **static** **void** main(String[] args) {  JoinThread joinThread = **new** JoinThread();  Thread t1 = **new** Thread(joinThread);  Thread t2 = **new** Thread(joinThread);  t1.start();  t2.start();  **try** {  //其他线程变为等待状态，等t1线程执行完成之后才能执行join方法。  t1.join();  } **catch** (Exception e) {  }  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println("main ---i:" + i);  }  }  } |

# 优先级

现代操作系统基本采用时分的形式调度运行的线程，线程分配得到的时间片的多少决定了线程使用处理器资源的多少，也对应了线程优先级这个概念。在JAVA线程中，通过一个int priority来控制优先级，范围为1-10，其中10最高，默认值为5。下面是源码（基于1.8）中关于priority的一些量和方法。

|  |
| --- |
| **class** PrioritytThread **implements** Runnable {  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().toString() + "---i:" + i);  }  }  }  /\*\*  \*  \* **@classDesc**: 功能描述:(Join方法)  \* **@author**: 余胜军  \* **@createTime**: 2017年8月20日 下午9:23:30  \* **@version**: v1.0  \* **@copyright**:上海每特教育科技有限公司  \*/  **public** **class** ThreadDemo4 {  **public** **static** **void** main(String[] args) {  PrioritytThread prioritytThread = **new** PrioritytThread();  Thread t1 = **new** Thread(prioritytThread);  Thread t2 = **new** Thread(prioritytThread);  t1.start();  // 注意设置了优先级， 不代表每次都一定会被执行。 只是CPU调度会有限分配  t1.setPriority(10);  t2.start();    }  } |

# Yield方法

Thread.yield()方法的作用：暂停当前正在执行的线程，并执行其他线程。（可能没有效果）

yield()让当前正在运行的线程回到可运行状态，以允许具有相同优先级的其他线程获得运行的机会。因此，使用yield()的目的是让具有相同优先级的线程之间能够适当的轮换执行。但是，实际中无法保证yield()达到让步的目的，因为，让步的线程可能被线程调度程序再次选中。

结论：大多数情况下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。

# 练习题

9.1有T1、T2、T3三个线程，如何怎样保证T2在T1执行完后执行，T3在T2执行完后执行？