Java进阶1 第8天

**【学习目标】理解、了解、应用、记忆**

通过今天的学习，参训学员能够：（解释的时候说出二级目标的掌握程度）

1. **【应用】多线程安全问题产生及解决方案**
   1. 【理解】能够独立分析多线程的安全问题
   2. 【应用】能够使用同步代码块解决多线程安全问题
   3. 【应用】能够使用同步方法解决多线程的安全问题
2. **【应用】多线程等待唤醒机制**
   1. 【应用】能够独立完成等待唤醒机制的案例
   2. 【理解】能够独立独立分析等待唤醒机制代码执行流程
3. **【应用】死锁**
   1. 【理解】 能够阐述死锁的定义
   2. 【应用】 能够独立写出一个死锁案例
   3. 【应用】 能够独立阐述线程各个状态之间的转换关系

# 多线程安全问题产生及解决方案

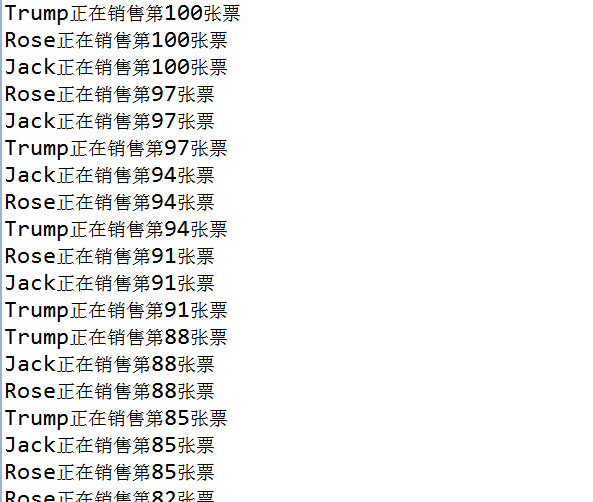
## 多线程安全问题产生

### 问题演示

#### 案例代码一:

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: TicketDemo  \* **@Description**: 多线程安全问题  \* **@date** 2018年2月4日 下午1:59:57  \* Company www.igeekhome.com  \*  \* 使用第二种创建并开启线程的方式，方便数据共享  \*  \* Thread方法:  \* public static void sleep(long millis) throws InterruptedException  \* 线程睡眠  \*  \* 安全问题原理分析：  \* 共享数据  \* 操作共享数据  \* 一次完整的操作过程，应该不可以被分割执行，即该完整操作过程期间，不应该被其他线程抢夺CPU  \*/  **public** **class** TicketDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  //创建卖票的线程执行目标对象  Ticket ticket = **new** Ticket();    //使用该卖票线程执行目标对象创建多个线程  Thread thread = **new** Thread(ticket,"Jack");  Thread thread2 = **new** Thread(ticket,"Rose");  Thread thread3 = **new** Thread(ticket,"Trump");    //开启多个线程  thread.start();  thread2.start();  thread3.start();    //sleep方法演示  **for**(**int** i=0;i<50;i++){  //线程暂停250毫秒  Thread.*sleep*(250);  System.***out***.println("main:"+i);  }  }  } |

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Ticket  \* **@Description**: 卖票类  \* **@date** 2018年2月4日 下午2:14:50  \* Company www.igeekhome.com  \*  \* 定义卖票的线程执行目标类  \*/  **public** **class** Ticket **implements** Runnable{  //在成员变量位置将票定义为数字100  /\*\*  \* **@Fields** number :票数  \*/  **private** **int** number = 100;    /\*\*  \* **@Title**: run  \* **@Description**: 完成卖票的线程逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //车站不停地在卖票  **while**(**true**){  //线程暂停20毫秒  **try** {  Thread.*sleep*(20);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    //有票就买票  **if**(number>0){  String threadName=Thread.*currentThread*().getName();  System.***out***.println(threadName+"正在销售第"+(number)+"张票");  //T  //R  //J  number--;  }**else**{  //没有票，就跳出循环，不再卖票  **break**;  }  }  }  } |



运行结果发现：

票出现了重复的票

### 产生原因分析

当多个线程操作共享的数据时，共同修改该数据。可能出现，当第1个线程想操作A状态的数据时，发现已经被第二个线程修改为了B状态，于是无法完成本想完成的任务。

## 多线程安全问题解决

### 利用同步代码块解决

Java中使用synchronized关键字来解决，将一个完整动作使用synchronized包裹。

同步代码块: 在代码块声明上，加上synchronized

synchronized (锁对象) {

可能会产生线程安全问题的代码

}

同步代码块中的锁对象可以是任意的对象；但多个线程操作相同数据时，要使用同一个锁对象才能够保证线程安全，避免安全问题。

同步解决方案解释：

即线程A中操作数据的代码与线程B中操作数据的代码均使用synchronized包裹，并使用相同的锁对象。

线程A先进入了同步块，这时线程B会等待线程A中synchronized包裹的代码执行完毕后再执行，此时线程A已经操作完了代码，反之线程A也会等待线程B。

#### 案例代码二:

利用同步代码块解决卖票的安全问题:

|  |
| --- |
| **package** com.igeek\_02\_01\_01;  /\*\*  \* **@ClassName**: TicketDemo  \* **@Description**: 多线程安全问题的解决方案  \* **@date** 2018年2月4日 下午1:59:57  \* Company www.igeekhome.com  \*  \* 多线程安全问题的解决方案：  \* Java中使用synchronized关键字来解决，将一个完整动作使用synchronized包裹。  \*  \* 同步代码块格式：  \* synchronized (锁对象) {  \* 可能会产生线程安全问题的代码  \* }  \*  \* 同步代码块中的锁对象可以是任意的对象；但多个线程操作相同数据时，要使用同一个锁对象才能够保证线程安全，避免安全问题。  \*  \* 同步解决方案解释：  \* 即线程A中操作数据的代码与线程B中操作数据的代码均使用synchronized包裹，并使用相同的锁对象。  \* 线程A先进入了同步块，这时线程B会等待线程A中synchronized包裹的代码执行完毕后再执行，此时线程A已经操作完了代码，反之线程A也会等待线程B。  \*/  **public** **class** TicketDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  //创建卖票的线程执行目标对象  Ticket ticket = **new** Ticket();    //使用该卖票线程执行目标对象创建多个线程  Thread thread = **new** Thread(ticket,"Jack");  Thread thread2 = **new** Thread(ticket,"Rose");  Thread thread3 = **new** Thread(ticket,"Trump");    //开启多个线程  thread.start();  thread2.start();  thread3.start();  }  } |

|  |
| --- |
| **package** com.igeek\_02\_01\_01;  /\*\*  \* **@ClassName**: Ticket  \* **@Description**: 卖票类  \* **@date** 2018年2月4日 下午2:14:50  \* Company www.igeekhome.com  \*  \* 定义卖票的线程执行目标类  \*/  **public** **class** Ticket **implements** Runnable{  //在成员变量位置将票定义为数字100  /\*\*  \* **@Fields** number :票数  \*/  **private** **int** number = 100;    //在成员位置定义锁对象  **private** Object lock = **new** Object();    /\*\*  \* **@Title**: run  \* **@Description**: 完成卖票的线程逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //车站不停地在卖票  **while**(**true**){  //T  //将一个完整动作使用synchronized同步代码块包裹  //J  //R  **synchronized**(lock) {  //线程暂停20毫秒  **try** {  Thread.*sleep*(20);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    //有票就买票  **if**(number>0){  String threadName = Thread.*currentThread*().getName();  System.***out***.println(threadName+"正在销售第"+(number)+"张票");    number--;  }**else**{  //没有票，就跳出循环，不再卖票  **break**;  }  }    }  }  } |

#### 案例代码三:

不同同步代码块使用相同锁对象，依然可以起到同步效果：

|  |
| --- |
| **package** com.igeek\_02\_01\_02;  /\*\*  \* **@ClassName**: Ticket  \* **@Description**: 卖票类  \* **@date** 2018年2月4日 下午2:14:50  \* Company www.igeekhome.com  \*  \* 定义卖票的线程执行目标类  \*/  **public** **class** Ticket **implements** Runnable{  //在成员变量位置将票定义为数字100  /\*\*  \* **@Fields** number :票数  \*/  **private** **int** number = 100;    //在成员位置定义锁对象  **private** Object lock = **new** Object();    /\*\*  \* **@Title**: run  \* **@Description**: 完成卖票的线程逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //车站不停地在卖票  **while**(**true**){  //T  //将一个完整动作使用synchronized同步代码块包裹  //J  //R  **synchronized**(lock) {  //线程暂停20毫秒  **try** {  Thread.*sleep*(20);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    //有票就买票  **if**(number>0){  String threadName = Thread.*currentThread*().getName();  System.***out***.println(threadName+"正在销售第"+(number)+"张票");    number--;  }**else**{  //没有票，就跳出循环，不再卖票  **break**;  }  }    }  }  } |

|  |
| --- |
| **package** com.igeek\_02\_01\_02;  /\*\*  \* **@ClassName**: TicketDemo  \* **@Description**: 多线程安全问题的解决方案  \* **@date** 2018年2月4日 下午1:59:57  \* Company www.igeekhome.com  \*  \* 多线程安全问题的解决方案：  \* Java中使用synchronized关键字来解决，将一个完整动作使用synchronized包裹。  \*  \* 同步代码块格式：  \* synchronized (锁对象) {  \* 可能会产生线程安全问题的代码  \* }  \*  \* 同步代码块中的锁对象可以是任意的对象；但多个线程操作相同数据时，要使用同一个锁对象才能够保证线程安全，避免安全问题。  \*  \* 同步解决方案解释：  \* 即线程A中操作数据的代码与线程B中操作数据的代码均使用synchronized包裹，并使用相同的锁对象。  \* 线程A先进入了同步块，这时线程B会等待线程A中synchronized包裹的代码执行完毕后再执行，此时线程A已经操作完了代码，反之线程A也会等待线程B。  \*  \* 不同同步代码块使用相同锁对象，依然可以起到同步效果  \*/  **public** **class** TicketDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  //创建卖票的线程执行目标对象  Ticket ticket = **new** Ticket();    //使用该卖票线程执行目标对象创建多个线程  Thread thread = **new** Thread(ticket,"Jack");  Thread thread2 = **new** Thread(ticket,"Rose");  Thread thread3 = **new** Thread(ticket,"Trump");    //开启多个线程  thread.start();  thread2.start();  thread3.start();  }  } |

### 利用同步方法解决

#### 案例代码四:

A：同步方法：在方法声明上，返回值前加上synchronized

public synchronized void method(){

可能会产生线程安全问题的代码

}

同步方法中的锁对象是this

|  |
| --- |
| **package** com.igeek\_02\_02\_01;  /\*\*  \* **@ClassName**: Ticket  \* **@Description**: 卖票类  \* **@date** 2018年2月4日 下午2:14:50  \* Company www.igeekhome.com  \*  \* 定义卖票的线程执行目标类  \* 不同同步代码块使用相同锁对象，依然可以起到同步效果  \*/  **public** **class** Ticket **implements** Runnable{  //在成员变量位置将票定义为数字100  /\*\*  \* **@Fields** number :票数  \*/  **private** **int** number = 100;    //在成员位置定义锁对象  **private** Object lock = **new** Object();    //定义一个标志位，让不同的线程执行不同的代码块  **private** **int** x = 0;    /\*\*  \* **@Title**: run  \* **@Description**: 完成卖票的线程逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //车站不停地在卖票  **while**(**true**){  //T    **if**(x%2==0){  //将一个完整动作使用synchronized同步代码块包裹  **synchronized**(**this**) {  //R  //线程暂停20毫秒  **try** {  Thread.*sleep*(20);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    //有票就买票  **if**(number>0){  String threadName = Thread.*currentThread*().getName();  System.***out***.println(threadName+"正在销售第"+(number)+"张票");    number--;  }  }  }**else**{  //J  //调用sell方法  **this**.sell();  }    **if**(number<=0){  //没有票，就跳出循环，不再卖票  **break**;  }    x++;  }  }    /\*\*  \* **@Title**: sell  \* **@Description**: 定义卖票的同步方法  \*/  **public** **synchronized** **void** sell(){  //线程暂停20毫秒  **try** {  Thread.*sleep*(20);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    //有票就买票  **if**(number>0){  String threadName = Thread.*currentThread*().getName();  System.***out***.println(threadName+"正在销售第"+(number)+"张票");    number--;  }  }  } |

|  |
| --- |
| **package** com.igeek\_02\_02\_01;  /\*\*  \* **@ClassName**: TicketDemo  \* **@Description**: 多线程安全问题的解决方案  \* **@date** 2018年2月4日 下午1:59:57  \* Company www.igeekhome.com  \*  \* 多线程安全问题的解决方案：  \* Java中使用synchronized关键字来解决，将一个完整动作使用synchronized包裹。  \*  \* 同步方法：  \* 同步方法的格式：在方法声明上，返回值前加上synchronized关键字，该同步方法中的锁对象是this  \*  \* 同步解决方案解释：  \* 即线程A中操作数据的代码与线程B中操作数据的代码均使用synchronized包裹，并使用相同的锁对象。  \* 线程A先进入了同步块，这时线程B会等待线程A中synchronized包裹的代码执行完毕后再执行，此时线程A已经操作完了代码，反之线程A也会等待线程B。  \*  \* 不同同步代码块使用相同锁对象，依然可以起到同步效果  \*  \*/  **public** **class** TicketDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  //创建卖票的线程执行目标对象  Ticket ticket = **new** Ticket();    //使用该卖票线程执行目标对象创建多个线程  Thread thread = **new** Thread(ticket,"Jack");  Thread thread2 = **new** Thread(ticket,"Rose");  Thread thread3 = **new** Thread(ticket,"Trump");    //开启多个线程  thread.start();  thread2.start();  thread3.start();  }  } |

#### 案例代码五:

B：静态同步方法：在方法声明上，加上static synchronized

public static synchronized void method(){

可能会产生线程安全问题的代码

}

静态同步方法中的锁对象是 类名.class

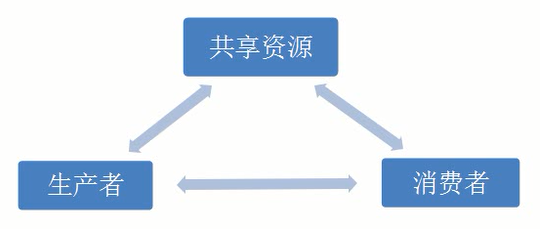
|  |
| --- |
| **package** com.igeek\_02\_02\_02;  /\*\*  \* **@ClassName**: Ticket  \* **@Description**: 卖票类  \* **@date** 2018年2月4日 下午2:14:50  \* Company www.igeekhome.com  \*  \* 定义卖票的线程执行目标类  \* 不同同步代码块使用相同锁对象，依然可以起到同步效果  \*/  **public** **class** Ticket **implements** Runnable{  //在成员变量位置将票定义为数字100  /\*\*  \* **@Fields** number :票数  \*/  **private** **static** **int** *number* = 100;    //在成员位置定义锁对象  **private** Object lock = **new** Object();    //定义一个标志位，让不同的线程执行不同的代码块  **private** **int** x = 0;    /\*\*  \* **@Title**: run  \* **@Description**: 完成卖票的线程逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //车站不停地在卖票  **while**(**true**){  //T    **if**(x%2==0){  //将一个完整动作使用synchronized同步代码块包裹  **synchronized**(Ticket.**class**) {  //R  //线程暂停20毫秒  **try** {  Thread.*sleep*(20);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    //有票就买票  **if**(*number*>0){  String threadName = Thread.*currentThread*().getName();  System.***out***.println(threadName+"正在销售第"+(*number*)+"张票");    *number*--;  }  }  }**else**{  //J  //调用sell方法  Ticket.*sell*();  }    **if**(*number*<=0){  //没有票，就跳出循环，不再卖票  **break**;  }    x++;  }  }    /\*\*  \* **@Title**: sell  \* **@Description**: 定义卖票的静态同步方法  \*/  **public** **static** **synchronized** **void** sell(){    //线程暂停20毫秒  **try** {  Thread.*sleep*(20);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    //有票就买票  **if**(*number*>0){  String threadName = Thread.*currentThread*().getName();  System.***out***.println(threadName+"正在销售第"+(*number*)+"张票");    *number*--;  }  }  } |

|  |
| --- |
| **package** com.igeek\_02\_02\_02;  /\*\*  \* **@ClassName**: TicketDemo  \* **@Description**: 多线程安全问题的解决方案  \* **@date** 2018年2月4日 下午1:59:57  \* Company www.igeekhome.com  \*  \* 多线程安全问题的解决方案：  \* Java中使用synchronized关键字来解决，将一个完整动作使用synchronized包裹。  \*  \* 同步方法：  \* 同步方法的格式：在方法声明上，返回值前加上synchronized关键字，该同步方法中的锁对象是this  \* 如果方法是静态方法：则该方法的锁是该类的字节码文件对象：类名.class  \*  \* 同步解决方案解释：  \* 即线程A中操作数据的代码与线程B中操作数据的代码均使用synchronized包裹，并使用相同的锁对象。  \* 线程A先进入了同步块，这时线程B会等待线程A中synchronized包裹的代码执行完毕后再执行，此时线程A已经操作完了代码，反之线程A也会等待线程B。  \*  \* 不同同步代码块使用相同锁对象，依然可以起到同步效果  \*  \*/  **public** **class** TicketDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  //创建卖票的线程执行目标对象  Ticket ticket = **new** Ticket();    //使用该卖票线程执行目标对象创建多个线程  Thread thread = **new** Thread(ticket,"Jack");  Thread thread2 = **new** Thread(ticket,"Rose");  Thread thread3 = **new** Thread(ticket,"Trump");    //开启多个线程  thread.start();  thread2.start();  thread3.start();  }  } |

# 多线程等待唤醒机制

## 等待唤醒机制

在开始讲解等待唤醒机制之前，有必要搞清一个概念——线程之间的通信：



A：多个线程在处理同一个资源，但是处理的动作（线程的任务）却不相同。通过一定的手段使各个线程能有效的利用资源。而这种手段即——等待唤醒机制。



B：等待唤醒机制所涉及到的方法：

wait()：等待，将正在执行的线程释放其执行资格和执行权，并存储到线程池中。

notify()：唤醒，唤醒线程池中被wait()的线程，一次唤醒一个，而且是任意的。

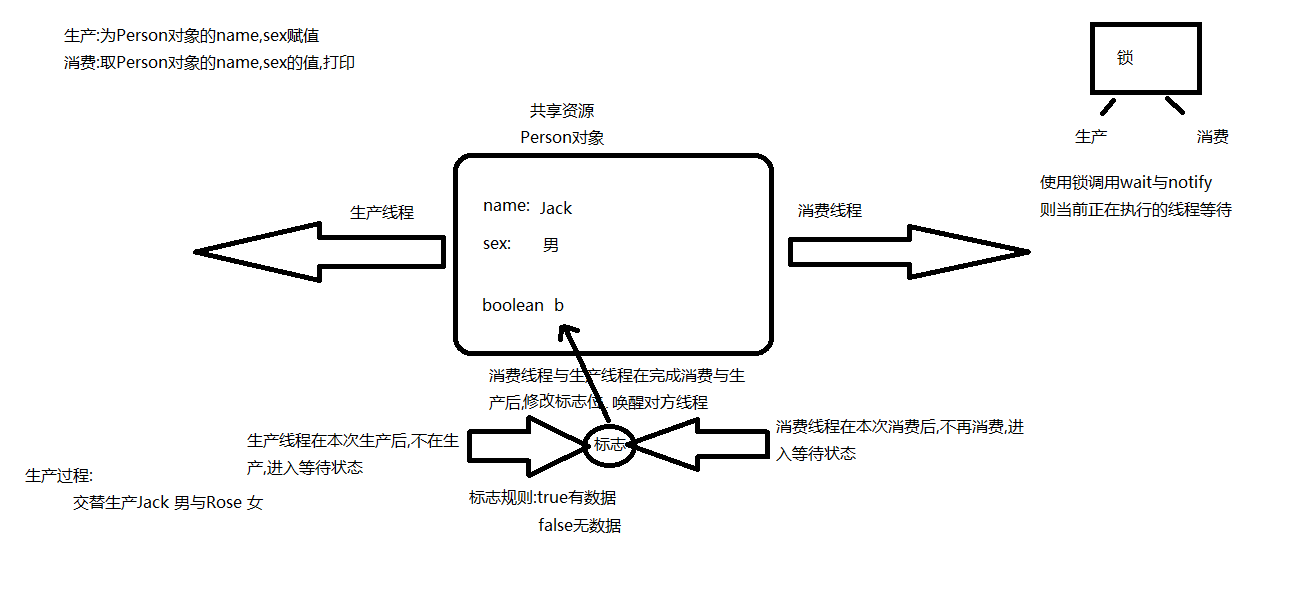
notifyAll()：唤醒全部，可以将线程池中的所有wait()线程都唤醒。

其实，所谓唤醒的意思就是让线程池中的线程具备执行资格。必须注意的是，这些方法都是在同步中才有效。同时这些方法在使用时必须标明所属锁，这样才可以明确出这些方法操作的到底是哪个锁上的线程。

C：仔细查看Java API之后，发现这些方法并不定义在Thread中，也没定义在Runnable接口中，却被定义在了Object类中，为什么这些操作线程的方法定义在Object类中？

因为这些方法在使用时，必须要标明所属的锁，而锁又可以是任意对象。能被任意对象调用的方法一定定义在Object类中。

## 等待唤醒机制案例



如上图说示，生产线程向共享资源中输入name，sex 值, 输出线程从共享资源中取出name，sex值输出，先要完成的任务是：

A：当生产线程发现共享资源中没有数据时，开始输入；输入完成后，叫醒消费线程来输出；

如果生产线程发现有数据，就wait()；

B：当消费线程发现共享资源中没有数据时，就wait()；

如果发现有数据时，就输出；输出完成后，叫醒生产线程来输入数据。

### 设计出生产线程,消费线程以及共享资源

#### 案例代码六:

|  |
| --- |
| **package** com.igeek\_02\_01;  /\*\*  \* **@ClassName**: WaitNotifyDemo  \* **@Description**: 等待唤醒机制  \* **@date** 2018年2月5日 下午2:20:55  \* Company www.igeekhome.com  \*  \* 开启生产者与消费者线程的主线程逻辑  \*  \* 生产者与消费者的线程执行目标可以通过构造方法传入共同操作的同一个资源对象  \*  \* 生产者生成：  \* 消费者消费：  \*/  **public** **class** WaitNotifyDemo {  **public** **static** **void** main(String[] args) {  //共享资源资源  Person person = **new** Person();    //创建生产者线程执行目标  PutIn putIn = **new** PutIn(person);    //创建消费者线程执行目标  GetOut getOut = **new** GetOut(person);    //开启生产者线程  Thread putInThread = **new** Thread(putIn);  putInThread.start();    //开启消费者线程  Thread getOutThread = **new** Thread(getOut);  getOutThread.start();  }  } |

|  |
| --- |
| **package** com.igeek\_02\_01;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person资源类  \* **@date** 2018年2月5日 下午2:23:19  \* Company www.igeekhome.com  \*  \* 定义共享资源Person，其中有两个成员变量：name、sex  \*/  **public** **class** Person {  //不使用private修饰符，可以简化后续操作  /\*\*  \* **@Fields** name : 姓名  \*/  String name;  /\*\*  \* **@Fields** sex : 性别  \*/  String sex;  /\*\*  \* **@Title**: Person  \*/  **public** Person() {  **super**();  }    } |

|  |
| --- |
| **package** com.igeek\_02\_01;  /\*\*  \* **@ClassName**: PutIn  \* **@Description**: 生产者  \* **@date** 2018年2月5日 下午2:25:17  \* Company www.igeekhome.com  \*  \* 生产者线程执行目标类  \*  \* 反复为Person对象赋值属性：Jack 男 与 Rose 女  \*/  **public** **class** PutIn **implements** Runnable{    **private** Person person;    //通过构造方法，从外界接收共享的那个资源  **public** PutIn(Person person){  **this**.person=person;  }  /\*\*  \* **@Title**: run  \* **@Description**: 实现接口中的run方法  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //使用死循环达到反复生成的目的  //如果是奇数，生成Jack 男；如果是偶数，生成Rose 女；  //Person person = new Person();  **int** i=0;  **while**(**true**){    //线程睡眠100毫秒  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    //判断生产内容  **if**(i%2==1){  person.name="Jack";  person.sex="男";  }**else**{  person.name="Rose";  person.sex="女";  }    System.***out***.println(person);  System.***out***.println("生产了："+person.name+"-"+person.sex);    //每次生成完，更改奇偶对应的整数  i++;  }  }  } |

|  |
| --- |
| **package** com.igeek\_02\_01;  /\*\*  \* **@ClassName**: GetOut  \* **@Description**: 消费者  \* **@date** 2018年2月5日 下午2:29:38  \* Company www.igeekhome.com  \*  \* 消费者线程执行目标类  \*  \* 反复获取Person对象的属性并打印  \*/  **public** **class** GetOut **implements** Runnable{    **private** Person person;    //通过构造方法，从外界接收共享的那个资源  **public** GetOut(Person person){  **this**.person=person;  }  /\*\*  \* **@Title**: run  \* **@Description**: 实现接口中的run方法  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //使用死循环达到反复生成的目的  //Person person = new Person();  **while**(**true**){    //线程睡眠100毫秒  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    String name = person.name;  String sex= person.sex;    System.***out***.println(person);  System.***out***.println("消费了："+name+"-"+sex);  }  }  } |

### 生产线程,消费线程产生的安全问题解决

#### 案例代码七:

|  |
| --- |
| **package** com.igeek\_02\_02;  /\*\*  \* **@ClassName**: WaitNotifyDemo  \* **@Description**: 等待唤醒机制  \* **@date** 2018年2月5日 下午2:20:55  \* Company www.igeekhome.com  \*  \* 开启生产者与消费者线程的主线程逻辑  \*  \* 生产者与消费者的线程执行目标可以通过构造方法传入共同操作的同一个资源对象  \*  \* 生产者和消费者线程安全解决：  \* 将生产过程和消费过程都使用synchronized包裹代码块，并且使用共享对象作为锁。  \*  \* 生产者生成：  \* 消费者消费：  \*/  **public** **class** WaitNotifyDemo {  **public** **static** **void** main(String[] args) {  //共享资源资源  Person person = **new** Person();    //创建生产者线程执行目标  PutIn putIn = **new** PutIn(person);    //创建消费者线程执行目标  GetOut getOut = **new** GetOut(person);    //开启生产者线程  Thread putInThread = **new** Thread(putIn);  putInThread.start();    //开启消费者线程  Thread getOutThread = **new** Thread(getOut);  getOutThread.start();  }  } |

|  |
| --- |
| **package** com.igeek\_02\_02;  /\*\*  \* **@ClassName**: PutIn  \* **@Description**: 生产者  \* **@date** 2018年2月5日 下午2:25:17  \* Company www.igeekhome.com  \*  \* 生产者线程执行目标类  \*  \* 反复为Person对象赋值属性：Jack 男 与 Rose 女  \*/  **public** **class** PutIn **implements** Runnable{    **private** Person person;    //通过构造方法，从外界接收共享的那个资源  **public** PutIn(Person person){  **this**.person=person;  }  /\*\*  \* **@Title**: run  \* **@Description**: 实现接口中的run方法  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //使用死循环达到反复生成的目的  //如果是奇数，生成Jack 男；如果是偶数，生成Rose 女；  //Person person = new Person();  **int** i=0;  **while**(**true**){    //使用共享对象作为锁  **synchronized**(person){  //判断生产内容  **if**(i%2==1){  person.name="Jack";  //线程睡眠100毫秒  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  person.sex="男";  }**else**{  person.name="Rose";  //线程睡眠100毫秒  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  person.sex="女";  }    System.***out***.println(person);  System.***out***.println("生产了："+person.name+"-"+person.sex);  }    //每次生成完，更改奇偶对应的整数  i++;  }  }  } |

|  |
| --- |
| **package** com.igeek\_02\_02;  /\*\*  \* **@ClassName**: GetOut  \* **@Description**: 消费者  \* **@date** 2018年2月5日 下午2:29:38  \* Company www.igeekhome.com  \*  \* 消费者线程执行目标类  \*  \* 反复获取Person对象的属性并打印  \*/  **public** **class** GetOut **implements** Runnable{    **private** Person person;    //通过构造方法，从外界接收共享的那个资源  **public** GetOut(Person person){  **this**.person=person;  }  /\*\*  \* **@Title**: run  \* **@Description**: 实现接口中的run方法  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //使用死循环达到反复生成的目的  //Person person = new Person();  **while**(**true**){    //使用共享对象作为锁  **synchronized**(person){  //线程睡眠100毫秒  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    String name = person.name;  String sex= person.sex;    System.***out***.println(person);  System.***out***.println("消费了："+name+"-"+sex);  }  }  }  } |

|  |
| --- |
| **package** com.igeek\_02\_02;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person资源类  \* **@date** 2018年2月5日 下午2:23:19  \* Company www.igeekhome.com  \*  \* 定义共享资源Person，其中有两个成员变量：name、sex  \*/  **public** **class** Person {  //不使用private修饰符，可以简化后续操作  /\*\*  \* **@Fields** name : 姓名  \*/  String name;  /\*\*  \* **@Fields** sex : 性别  \*/  String sex;  /\*\*  \* **@Title**: Person  \*/  **public** Person() {  **super**();  }    } |

### 加入等待唤醒机制,实现生产一次,消费一次

#### 案例代码八:

|  |
| --- |
| **package** com.igeek\_02\_03;  /\*\*  \* **@ClassName**: WaitNotifyDemo  \* **@Description**: 等待唤醒机制  \* **@date** 2018年2月5日 下午2:20:55  \* Company www.igeekhome.com  \*  \* 开启生产者与消费者线程的主线程逻辑  \*  \* 生产者与消费者的线程执行目标可以通过构造方法传入共同操作的同一个资源对象  \*  \* 生产者和消费者线程安全解决：  \* 将生产过程和消费过程都使用synchronized包裹代码块，并且使用共享对象作为锁。  \*  \* Object类的方法：  \* public final void wait() throws InterruptedException：让线程等待；  \* public final void notify()：唤醒等待的线程  \*  \* 解决等待唤醒操作：  \* 1、在Person类中定义标志位；  \* 2、  \* 在生产者线程执行目标中，先判断是否有数据  \* 有：生产者线程等待(使用锁wait)  \* 无：正常生产  \* 该标记位为true  \* 唤醒消费者  \* 3、  \* 在消费者线程执行目标中，先判断是否有数据  \* 有：正常消费  \* 无：消费者线程等待(使用锁wait)  \* 该标记位为false  \* 唤醒生产者  \*  \* sleep与wait方法是否都释放锁：  \* sleep由于线程会苏醒，所以不释放锁；  \* wait释放锁，由其他线程唤醒等待的这个线程，被唤醒后，会继续执行wait代码后边的代码  \*  \* 生产者生成：  \* 消费者消费：  \*/  **public** **class** WaitNotifyDemo {  **public** **static** **void** main(String[] args) {  //共享资源资源  Person person = **new** Person();    //创建生产者线程执行目标  PutIn putIn = **new** PutIn(person);    //创建消费者线程执行目标  GetOut getOut = **new** GetOut(person);    //开启生产者线程  Thread putInThread = **new** Thread(putIn);  putInThread.start();    //开启消费者线程  Thread getOutThread = **new** Thread(getOut);  getOutThread.start();  }  } |

|  |
| --- |
| **package** com.igeek\_02\_03;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person资源类  \* **@date** 2018年2月5日 下午2:23:19  \* Company www.igeekhome.com  \*  \* 定义共享资源Person，其中有两个成员变量：name、sex  \*/  **public** **class** Person {  //不使用private修饰符，可以简化后续操作  /\*\*  \* **@Fields** name : 姓名  \*/  String name;  /\*\*  \* **@Fields** sex : 性别  \*/  String sex;    /\*\*  \* **@Fields** flag : 定义布尔值，作为判断哪个线程等待的标记位：true有数据，生产者等待；false无数据，消费者等待；  \*/  **boolean** flag;  /\*\*  \* **@Title**: Person  \*/  **public** Person() {  **super**();  }    } |

|  |
| --- |
| **package** com.igeek\_02\_03;  /\*\*  \* **@ClassName**: PutIn  \* **@Description**: 生产者  \* **@date** 2018年2月5日 下午2:25:17  \* Company www.igeekhome.com  \*  \* 生产者线程执行目标类  \*  \* 反复为Person对象赋值属性：Jack 男 与 Rose 女  \*/  **public** **class** PutIn **implements** Runnable{    **private** Person person;    //通过构造方法，从外界接收共享的那个资源  **public** PutIn(Person person){  **this**.person=person;  }  /\*\*  \* **@Title**: run  \* **@Description**: 实现接口中的run方法  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //使用死循环达到反复生成的目的  //如果是奇数，生成Jack 男；如果是偶数，生成Rose 女；  //Person person = new Person();  **int** i=0;  **while**(**true**){    //使用共享对象作为锁  **synchronized**(person){    //判断标志位  **while**(person.flag){  //如果有数据，就等待  **try** {  person.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    //如果没有数据或者刚刚被唤醒    //判断生产内容  **if**(i%2==1){  person.name="Jack";  //线程睡眠100毫秒  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  person.sex="男";  }**else**{  person.name="Rose";  //线程睡眠100毫秒  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  person.sex="女";  }    System.***out***.println(person);  System.***out***.println("生产了："+person.name+"-"+person.sex);    //该标记位为true  person.flag=**true**;    //如果有消费者等待，就唤醒消费者；如果没有就无需唤醒，相当于这句代码没有执行  person.notify();  }    //每次生成完，更改奇偶对应的整数  i++;  }  }  } |

|  |
| --- |
| **package** com.igeek\_02\_03;  /\*\*  \* **@ClassName**: GetOut  \* **@Description**: 消费者  \* **@date** 2018年2月5日 下午2:29:38  \* Company www.igeekhome.com  \*  \* 消费者线程执行目标类  \*  \* 反复获取Person对象的属性并打印  \*/  **public** **class** GetOut **implements** Runnable{    **private** Person person;    //通过构造方法，从外界接收共享的那个资源  **public** GetOut(Person person){  **this**.person=person;  }  /\*\*  \* **@Title**: run  \* **@Description**: 实现接口中的run方法  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //使用死循环达到反复生成的目的  //Person person = new Person();  **while**(**true**){    //使用共享对象作为锁  **synchronized**(person){    //判断标记位  **while**(!person.flag){  //如果没有数据，就等待  **try** {  person.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    //如果有数据或者刚刚被唤醒    //线程睡眠100毫秒  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    String name = person.name;  String sex= person.sex;    System.***out***.println(person);  System.***out***.println("消费了："+name+"-"+sex);    //该标记位为false  person.flag=**false**;    //如果有生产者等待，就唤醒生产者；如果没有就无需唤醒，相当于这句代码没有执行  person.notify();  }  }  }  } |

# 死锁

## 死锁概述:

同步锁使用的弊端：当线程任务中出现了多个同步(多个锁)时，如果同步中嵌套了其他的同步。这时容易引发一种现象：程序出现无限等待，这种现象我们称为死锁。这种情况能避免就避免掉。

synchronzied(A锁){

synchronized(B锁){

}

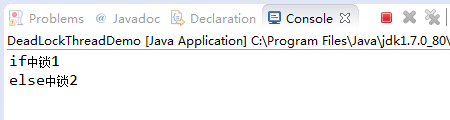
}

## 死锁实现

### 案例代码九:

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: DeadLockThreadDemo  \* **@Description**: 死锁的测试类  \* **@date** 2018年2月5日 下午4:24:31  \* Company www.igeekhome.com  \*  \*/  **public** **class** DeadLockThreadDemo {  //定义了锁1与锁2的对象  **public** **static** Object *LOCK1* = **new** Object();  **public** **static** Object *LOCK2* = **new** Object();    **public** **static** **void** main(String[] args) {  DeadLockThread thread1 = **new** DeadLockThread(**true**);  DeadLockThread thread2 = **new** DeadLockThread(**false**);    thread1.start();  thread2.start();  }  } |

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: DeadLockThread  \* **@Description**: 死锁  \* **@date** 2018年2月5日 下午4:23:59  \* Company www.igeekhome.com  \*  \*/  **public** **class** DeadLockThread **extends** Thread{    **boolean** flag;//定义标记，用来指定要执行的代码    **public** DeadLockThread(**boolean** flag) {  **this**.flag = flag;  }    @Override  **public** **void** run() {  **if**(flag) {//flag赋值为true时，执行的代码  **synchronized** (DeadLockThreadDemo.*LOCK1*) {  System.***out***.println("if中锁1");  **synchronized** (DeadLockThreadDemo.*LOCK2*) {  System.***out***.println("if中锁2");  }  }  } **else** {//flag赋值为false时，执行的代码  **synchronized** (DeadLockThreadDemo.*LOCK2*) {  System.***out***.println("else中锁2");  **synchronized** (DeadLockThreadDemo.*LOCK1*) {  System.***out***.println("else中锁1");  }  }  }  }  } |



Thread-0线程 持有LOCK1 锁 ,请求 LOCK2 锁

Thread-1线程 持有LOCK2 锁 ,请求 LOCK1 锁

两者都在等待对方释放锁,结果都不释放,进入永久的相互等待

重点和总结

1、多线程的安全问题及解决方案

2、多线程中等待唤醒机制

3、死锁的问题