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
4. **【应用】网络编程概述**
   1. 【理解】能够阐述IP地址和端口号的概念
   2. 【理解】能够理解网络模型的概念
5. **【应用】UDP协议**
   1. 【应用】能够独立实现UDP的发送端和接收端
   2. 【应用】能够独立实现UDP聊天大厅案例
6. **【应用】TCP协议**
   1. 【应用】 能够独立实现TCP的客户端和服务端
   2. 【应用】 能够独立实现服务端向客户端发送数据
   3. 【应用】 能够独立实现服务端循环读取客户端发送的数据
   4. 【应用】 能够独立实现上传文件案例

# 多线程安全问题产生及解决方案

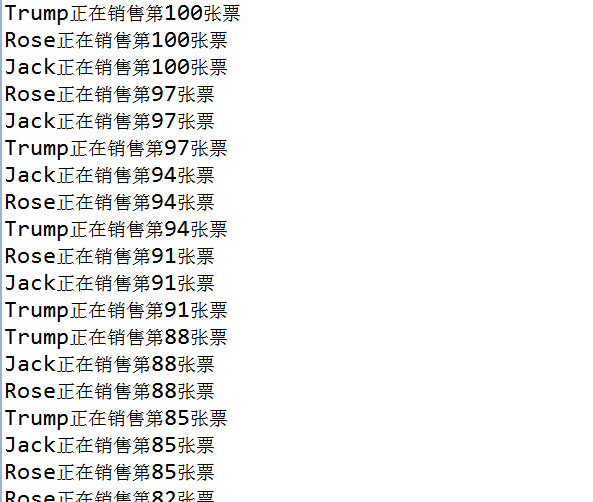
## 多线程安全问题产生

### 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

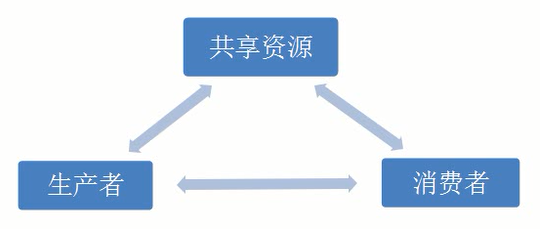
|  |
| --- |
| **package** com.igeek\_02\_02\_02;  /\*\*  \* **@ClassName**: Ticket  \* **@Description**: 卖票类  \* **@date** 2018年2月4日 下午2:14:50  \* Company www.igeekhome.com  \*  \* 定义卖票的线程执行目标类  \* 不同同步代码块使用相同锁对象，依然可以起到同步效果  \*/  **public** **class** Ticket **implements** Runnable{  //在成员变量位置将票定义为数字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。

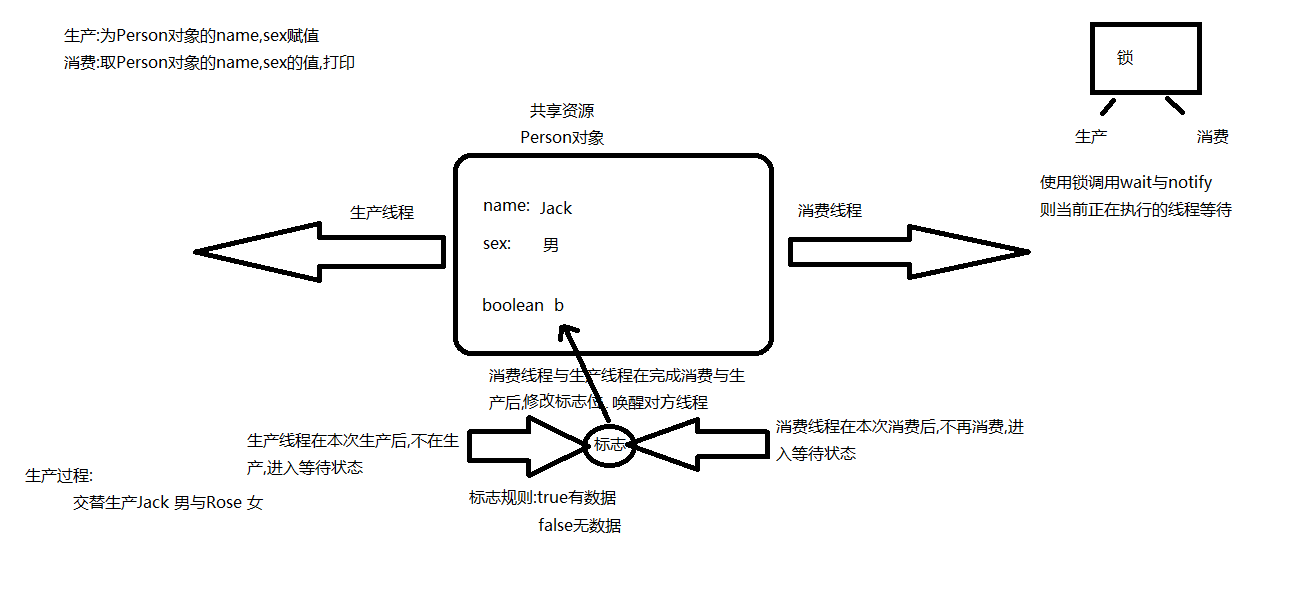
notifyAll()：唤醒全部，可以将线程池中的所有wait()线程都唤醒。

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){

}

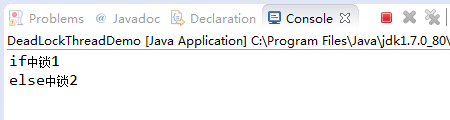
}

## 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

# 网络编程概述

## IP地址&InetAddress

IP是每台电脑在互联网上的唯一标识符。

一个IPV4的地址是由四段0—255的数字组成：192.168.0.100。

Java中可以使用InetAddress类表示一个IP地址。

常见方法：

public static InetAddress **getLocalHost**()

throws UnknownHostException 获取本地主机

public static InetAddress **getByName**(String host)

throws UnknownHostException 依据主机名获取主机

public String **getHostName**() 获取主机名称

public String **getHostAddress**() 获取主机字符串形式的IP

其他：

ipconfig用于DOS获取计算机IP设置。

ping用于判断两台计算机连接是否通畅。

127.0.0.1与localhost类似，均代表本机地址。

xxx.xxx.xxx.255 广播地址，即该网段下所有用户均可以被通知到。

### 案例代码十:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.net.InetAddress;  **import** java.net.UnknownHostException;  /\*\*  \* **@ClassName**: IPDemo  \* **@Description**: IP描述  \* **@date** 2018年2月6日 下午1:03:43  \* Company www.igeekhome.com  \*  \* IP地址是一台电脑在网络当中最常用的唯一标识，  \* 也可以使用主机名(计算机名)在网络中找到电脑主机，但是主机名有可能重复，会带来许多不安全的因素，弃用主机名，  \* 主要使用IP地址  \*  \* 由四个网段组成：  \* 其本质为：  \* 4个字节的二进制数  \* 二进制数 11111111 10101010 01010101 11110000  \* 十进制 数字1 数字2 数字3 数字4  \*  \* 由于每段数字均由一个字节翻译成十进制数，所有最大为255  \*  \* Java中的InetAddress代表地址，主要代表IP地址  \*  \* 获取地址对象：  \* public static InetAddress getLocalHost() throws UnknownHostException：  \* 获取本地主机  \* public static InetAddress getByName(String host) throws UnknownHostException：  \* 依据主机名(或者是IP地址字符串)获取主机  \*  \* 通过地址对象获取IP字符串或者主机名称字符串:  \* public String getHostName(): 获取主机名称  \* public String getHostAddress(): 获取主机字符串形式的IP  \*  \*/  **public** **class** IPDemo {  **public** **static** **void** main(String[] args) **throws** UnknownHostException {  //获取本地IP地址  InetAddress localHost = InetAddress.*getLocalHost*();    //获取IP字符串或者主机名称字符串  String hostName = localHost.getHostName();  System.***out***.println(hostName);    String hostAddress = localHost.getHostAddress();  System.***out***.println(hostAddress);    //获取其他主机IP地址  InetAddress byName = InetAddress.*getByName*("SKY-20170629LIV");  System.***out***.println(byName);    InetAddress byName2 = InetAddress.*getByName*("192.168.2.99");  System.***out***.println(byName2);  }  } |

## 端口号

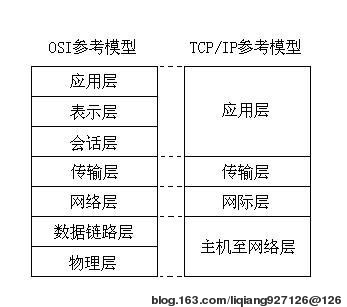
通过IP地址可以连接到指定计算机，但如果想访问目标计算机中的某个应用程序，还需要指定端口号。在计算机中，不同的应用程序是通过端口号区分的。端口号是用两个字节（16位的二进制数）表示的，它的取值范围是0~65535，其中，0~1023之间的端口号用于一些知名的网络服务和应用，用户的普通应用程序需要使用1024以上的端口号，从而避免端口号被另外一个应用或服务所占用。

## 网络模型

网络模型是计算机网络通讯规范，一般符合以下规则：

OSI（Open System Interconnection开放系统互连）参考模型；

TCP/IP参考模型。



## Scoket套接字

在Internet上的主机一般运行了多个服务软件，同时提供几种服务。每种服务都打开一个Socket，并绑定到一个端口上，不同的端口对应于不同的服务。

Socket就是为网络编程提供的一种机制，又叫套接字编程。Socket需要理解以下几点内容：

通信的两端都有Socket；

网络通信其实就是Socket间的通信；

数据在两个Socket间通过IO传输；

Socket编程一般使用TCP与UDP两种协议，这里主要介绍TCP。

# UDP协议

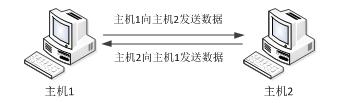
## 概述

UDP是无连接通信协议，即在数据传输时，数据的发送端和接收端不建立逻辑连接。简单来说，当一台计算机向另外一台计算机发送数据时，发送端不会确认接收端是否存在，就会发出数据，同样接收端在收到数据时，也不会向发送端反馈是否收到数据。

由于使用UDP协议消耗资源小，通信效率高，所以通常都会用于音频、视频和普通数据的传输，例如视频会议都使用UDP协议，因为这种情况即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。

但是在使用UDP协议传送数据时，由于UDP的面向无连接性，不能保证数据的完整性，因此在传输重要数据时不建议使用UDP协议。

UDP的交换过程如下图所示。



## UDP协议实现

发送和接收均可以使用的Socket类：

DatagramSocket

数据打包的相关类：

DatagramPacket

使用UDP发送或接收数据的步骤：

1、建立发送端，接收端；

2、创建数据；

3、建立数据包；

4、调用Socket的发送、接收方法；

5、关闭Socket。

需求：教师的电脑的一个程序发送数据，一个程序接收数据，使用的教师本机的IP。

### 发送端

#### 案例代码十一:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.io.IOException;  **import** java.net.DatagramPacket;  **import** java.net.DatagramSocket;  **import** java.net.InetAddress;  **import** java.net.SocketException;  **import** java.net.UnknownHostException;  /\*\*  \* **@ClassName**: UDPSend  \* **@Description**: UDP的发送端  \* **@date** 2018年2月6日 下午4:08:39  \* Company www.igeekhome.com  \*  \* 发送和接收均可以使用的Socket类：  \* DatagramSocket  \*  \* 数据打包的相关类：  \* DatagramPacket  \*  \* 使用UDP发送数据的步骤：  \* 1、建立发送端，空参构造；  \* 2、创建数据；  \* 3、建立数据包；  \* public DatagramPacket(byte[] buf, 数据打包成的字节数组  int length, 数据的长度  InetAddress address, 目标程序IP地址  int port) 目标程序端口  \* 4、调用Socket的发送；  \* public void send(DatagramPacket p) 发送数据报包  \* 5、关闭Socket。  \*  \* ps：发送端与接收端是两个对立的运行程序。  \*/  **public** **class** UDPSend {  **public** **static** **void** main(String[] args) **throws** IOException {  //1、建立发送端  DatagramSocket ds = **new** DatagramSocket();    //2、创建数据  String s = "i love java";    //3、建立数据包  **byte**[] data = s.getBytes();  **int** length = data.length;  InetAddress address = InetAddress.*getByName*("192.168.2.99");  **int** port = 12345;  DatagramPacket dp = **new** DatagramPacket(data , length, address, port );    //4、调用Socket的发送  ds.send(dp);    //5、关闭Socket  ds.close();  }  } |

### 接收端

#### 案例代码十二:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.io.IOException;  **import** java.net.DatagramPacket;  **import** java.net.DatagramSocket;  **import** java.net.InetAddress;  **import** java.net.SocketException;  /\*\*  \* **@ClassName**: UDPReceive  \* **@Description**: UDP的接收端  \* **@date** 2018年2月6日 下午4:31:45  \* Company www.igeekhome.com  \*  \* 发送和接收均可以使用的Socket类：  \* DatagramSocket  \*  \* 数据打包的相关类：  \* DatagramPacket  \*  \* 使用UDP接收数据的步骤：  \* 1、建立接收端；  \* public DatagramSocket(int port): 指定程序端口号，方便发送端寻找  \* 2、建立数据包，用于接收数据；  \* public DatagramPacket(byte[] buf, int length)  \* 3、调用Socket的接收方法；  \* 4、解析数据 ；  \* public byte[] getData(): 获取数据报包中的数据  \* public int getLength()： 获取数据报包中的数据长度  \* public InetAddress getAddress(): 获取发送方的IP  \* public int getPort()： 获取发送方的端口号  \* 5、关闭Socket。  \*  \* ps：发送端与接收端是两个对立的运行程序。  \*/  **public** **class** UDPReceive {  **public** **static** **void** main(String[] args) **throws** IOException {  //1、建立接收端  DatagramSocket ds = **new** DatagramSocket(12345);    //2、建立数据包，用于接收数据  **byte**[] buf = **new** **byte**[1024];  DatagramPacket dp = **new** DatagramPacket(buf, buf.length);    //3、调用Socket的接收方法  ds.receive(dp);    //4、解析数据  InetAddress sendAddress = dp.getAddress();  String sendIP = sendAddress.getHostAddress();  **int** sendPort = dp.getPort();    **byte**[] data = dp.getData();  **int** length = dp.getLength();    //将字节数组转成字符串  String content = **new** String(data, 0, length);    System.***out***.println("收到来自"+sendIP+"的，端口号为"+sendPort+"的数据："+content);    //5、关闭Socket  ds.close();    }  } |

## UDP案例

需求：

发送端：不断的向接收端发送数据

接收端：不断的接收从发送端发送过来的数据

### 案例代码是十三:

|  |
| --- |
| **package** com.igeek\_03;  **import** java.io.IOException;  **import** java.net.DatagramPacket;  **import** java.net.DatagramSocket;  **import** java.net.InetAddress;  **import** java.net.SocketException;  **import** java.net.UnknownHostException;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: UDPSend  \* **@Description**: UDP的发送端  \* **@date** 2018年2月6日 下午4:08:39  \* Company www.igeekhome.com  \*  \* 反复向接收端发送数据  \*/  **public** **class** UDPSend {  **public** **static** **void** main(String[] args) **throws** IOException {  //1、建立发送端  DatagramSocket ds = **new** DatagramSocket(54321);    //反复发送数据  **while**(**true**){  //2、创建数据  Scanner scanner = **new** Scanner(System.***in***);  System.***out***.print("请输入发送的内容：");  String s = scanner.nextLine();    //3、建立数据包  **byte**[] data = s.getBytes();  **int** length = data.length;  InetAddress address = InetAddress.*getByName*("192.168.2.99");  **int** port = 12345;  DatagramPacket dp = **new** DatagramPacket(data , length, address, port );    //4、调用Socket的发送  ds.send(dp);  }    }  } |

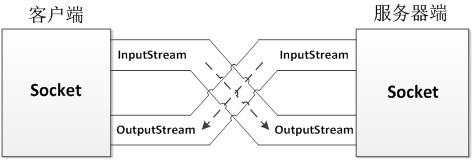
|  |
| --- |
| **package** com.igeek\_03;  **import** java.io.IOException;  **import** java.net.DatagramPacket;  **import** java.net.DatagramSocket;  **import** java.net.InetAddress;  **import** java.net.SocketException;  /\*\*  \* **@ClassName**: UDPReceive  \* **@Description**: UDP的接收端  \* **@date** 2018年2月6日 下午4:31:45  \* Company www.igeekhome.com  \*  \* 反复接收从发送端发送过来的数据  \*/  **public** **class** UDPReceive {  **public** **static** **void** main(String[] args) **throws** IOException {  //1、建立接收端  DatagramSocket ds = **new** DatagramSocket(12345);    //反复接收数据  **while**(**true**){  //2、建立数据包，用于接收数据  **byte**[] buf = **new** **byte**[1024];  DatagramPacket dp = **new** DatagramPacket(buf, buf.length);    //3、调用Socket的接收方法  ds.receive(dp);    //4、解析数据  InetAddress sendAddress = dp.getAddress();  String sendIP = sendAddress.getHostAddress();  **int** sendPort = dp.getPort();    **byte**[] data = dp.getData();  **int** length = dp.getLength();    //将字节数组转成字符串  String content = **new** String(data, 0, length);    System.***out***.println("收到来自"+sendIP+"的，端口号为"+sendPort+"的数据："+content);  }    }  } |

# TCP协议

## 概述:

TCP协议通过三次握手协议将客户端与服务器端连接，两端使用各自的Socket对象。Socket对象中包含了IO流，供数据传输。

即：TCP协议在客户端与服务器端通过Socket组成了IO通道，数据在IO通道中传输，客户端传输的数据成队列方式写出，服务器端接收，同样服务器端传输的数据成队列方式写出，客户端接收。



## TCP协议实现

Socket客户端和ServerSocket服务器端：

1、建立客户端和服务器端；

2、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输；

（如果是服务器端，则需要添加一步操作：通过Socket服务获取Socket，再获取其中的IO流）

3、关闭socket。

注意点：

1、服务器端开启后等待客户端访问，可以不关闭；

2、一个服务端可以对应多个客户端；

3、不同客户端间通信可以通过服务器端中转信息。

### TCP客户端

#### 案例代码十四:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.io.IOException;  **import** java.io.OutputStream;  **import** java.net.Socket;  **import** java.net.UnknownHostException;  /\*\*  \* **@ClassName**: Client  \* **@Description**: 客户端  \* **@date** 2018年2月6日 下午7:31:14  \* Company www.igeekhome.com  \*  \* 客户端发送信息，服务端接收信息  \*  \* Socket客户端：  \* 1、建立客户端；  \* public Socket(String host, int port)： 指定服务器IP与端口  \* 2、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输；  \* public OutputStream getOutputStream()： 输出数据  \* public InputStream getInputStream()： 输入数据  \* 3、关闭socket。  \*/  **public** **class** Client {  **public** **static** **void** main(String[] args) **throws** UnknownHostException, IOException {  //1、建立客户端  Socket socket = **new** Socket("192.168.2.99", 55555);    //2、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输  OutputStream out = socket.getOutputStream();    //输出数据  out.write(97);    //3、关闭socket  socket.close();  }  } |

### TCP服务端

#### 案例代码十五:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.net.ServerSocket;  **import** java.net.Socket;  /\*\*  \* **@ClassName**: Server  \* **@Description**: 服务器端  \* **@date** 2018年2月6日 下午7:39:07  \* Company www.igeekhome.com  \*  \* 客户端发送信息，服务端接收信息  \*  \* ServerSocket服务器端：  \* 1、建立服务器端；  \* public ServerSocket(int port)： 指定服务器端端口，方便客户端访问  \* 2、由于是服务器端，则需要添加一步操作：通过Socket服务获取Socket，再获取其中的IO流  \* public Socket accept()： 接收一个socket对象  \* 3、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输；  \* public OutputStream getOutputStream()： 输出数据  \* public InputStream getInputStream()： 输入数据  \* 4、关闭socket。  \*/  **public** **class** Server {  **public** **static** **void** main(String[] args) **throws** IOException {  //1、建立服务器端；  ServerSocket serverSocket = **new** ServerSocket(55555);    //2、由于是服务器端，则需要添加一步操作：通过Socket服务获取Socket，再获取其中的IO流  Socket socket = serverSocket.accept();    //3、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输；  InputStream in = socket.getInputStream();    **int** read = in.read();    System.***out***.println(read);  System.***out***.println((**char**)read);    //4、关闭socket。  socket.close();  serverSocket.close();  }  } |

## 服务端向客户端发送数据

### 案例代码十六:

|  |
| --- |
| **package** com.igeek\_03;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **import** java.net.Socket;  **import** java.net.UnknownHostException;  /\*\*  \* **@ClassName**: Client  \* **@Description**: 客户端  \* **@date** 2018年2月6日 下午7:52:58  \* Company www.igeekhome.com  \*  \* 服务器响应数据  \*  \* 客户端只需要按照顺序，在输出数据后，再获取输入流，输入数据即可  \*/  **public** **class** Client {  **public** **static** **void** main(String[] args) **throws** UnknownHostException, IOException {  //1、建立客户端；  Socket socket = **new** Socket("192.168.2.99", 55555);    //2、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输；  OutputStream out = socket.getOutputStream();    //输出数据  out.write(97);    //获取输入流，输入响应数据  InputStream in = socket.getInputStream();  **int** read = in.read();    System.***out***.println(read);  System.***out***.println((**char**)read);    //3、关闭socket。  socket.close();  }  } |

|  |
| --- |
| **package** com.igeek\_03;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **import** java.net.ServerSocket;  **import** java.net.Socket;  /\*\*  \* **@ClassName**: Server  \* **@Description**: 服务器端  \* **@date** 2018年2月6日 下午8:00:43  \* Company www.igeekhome.com  \*  \* 服务器响应数据  \*  \* 服务器端只需要按照顺序，在完成输入数据后，再获取输出流，输出数据即可  \*/  **public** **class** Server {  **public** **static** **void** main(String[] args) **throws** IOException {  //1、建立服务器端；  ServerSocket serverSocket = **new** ServerSocket(55555);    //2、由于是服务器端，则需要添加一步操作：通过Socket服务获取Socket，再获取其中的IO流  Socket socket = serverSocket.accept();    //3、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输；  InputStream in = socket.getInputStream();    **int** read = in.read();    System.***out***.println(read);  System.***out***.println((**char**)read);    //获取输出流，输出响应数据  OutputStream out = socket.getOutputStream();  out.write(98);    //4、关闭socket。  socket.close();  serverSocket.close();  }  } |

## 服务端循环读取客户端发送的数据

### 案例代码十七:

|  |
| --- |
| **package** com.igeek\_04;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **import** java.net.Socket;  **import** java.net.UnknownHostException;  /\*\*  \* **@ClassName**: Client  \* **@Description**: 客户端  \* **@date** 2018年2月6日 下午7:52:58  \* Company www.igeekhome.com  \*  \* 客户端发送一个字符串i love java信息，接收服务器端返回一个字母o的响应  \*  \* public void shutdownOutput(): 禁止输出流，告知对方  \*/  **public** **class** Client {  **public** **static** **void** main(String[] args) **throws** UnknownHostException, IOException {  //1、建立客户端；  Socket socket = **new** Socket("192.168.2.99", 55555);    //2、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输；  OutputStream out = socket.getOutputStream();    //输出数据  String content = "i love java";  out.write(content.getBytes());    //告知服务器，不再发送数据了  socket.shutdownOutput();    //获取输入流，输入响应数据  InputStream in = socket.getInputStream();  **int** read = in.read();    System.***out***.println(read);  System.***out***.println((**char**)read);    //3、关闭socket。  socket.close();  }  } |

|  |
| --- |
| **package** com.igeek\_04;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **import** java.net.ServerSocket;  **import** java.net.Socket;  /\*\*  \* **@ClassName**: Server  \* **@Description**: 服务器端  \* **@date** 2018年2月6日 下午8:00:43  \* Company www.igeekhome.com  \*  \* 客户端发送一个字符串i love java信息，接收服务器端返回一个字母o的响应  \*/  **public** **class** Server {  **public** **static** **void** main(String[] args) **throws** IOException {  //1、建立服务器端；  ServerSocket serverSocket = **new** ServerSocket(55555);    //2、由于是服务器端，则需要添加一步操作：通过Socket服务获取Socket，再获取其中的IO流  Socket socket = serverSocket.accept();    //3、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输；  InputStream in = socket.getInputStream();    //反复使用一个字节接收  **int** read = -1;    **while**((read=in.read())!=-1){  System.***out***.print((**char**)read);  }    //获取输出流，输出响应数据  OutputStream out = socket.getOutputStream();  String content ="o";  out.write(content.getBytes());    //4、关闭socket。  socket.close();  serverSocket.close();  }  } |

## 上传文件案例

需求：将文件上传到指定IP的主机（这里的客户端和服务器都使用同一个机器）。

### 案例代码十八:

|  |
| --- |
| **package** com.igeek\_05;  **import** java.io.FileInputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **import** java.net.Socket;  **import** java.net.UnknownHostException;  /\*\*  \* **@ClassName**: Client  \* **@Description**: 客户端  \* **@date** 2018年2月6日 下午7:52:58  \* Company www.igeekhome.com  \*  \* 文件上传，客户端发送的数据是一个文件当中的数据  \*  \* 读取文件内容，发送给服务器端  \*/  **public** **class** Client {  **public** **static** **void** main(String[] args) **throws** UnknownHostException, IOException {  //1、建立客户端；  Socket socket = **new** Socket("192.168.2.99", 55555);    //2、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输；  OutputStream out = socket.getOutputStream();    //获取资源文件输入流对象  FileInputStream fis = **new** FileInputStream("ASCII码表完整版.doc");    //读取文件内容  **int** b=-1;    **while**((b=fis.read())!=-1){  //输出数据给服务器  out.write(b);  }    //关闭资源文件输入流  fis.close();    //告知服务器，不再发送数据了  socket.shutdownOutput();    //获取输入流，输入响应数据  InputStream in = socket.getInputStream();  **int** read = in.read();    System.***out***.println(read);  System.***out***.println((**char**)read);    //3、关闭socket。  socket.close();  }  } |

|  |
| --- |
| **package** com.igeek\_05;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **import** java.net.ServerSocket;  **import** java.net.Socket;  /\*\*  \* **@ClassName**: Server  \* **@Description**: 服务器端  \* **@date** 2018年2月6日 下午8:00:43  \* Company www.igeekhome.com  \*  \* 文件上传，服务器端接收文件中的数据，向服务器端的一个文件中写出  \*  \* 接收文件内容，写出到某个文件  \*/  **public** **class** Server {  **public** **static** **void** main(String[] args) **throws** IOException {  //1、建立服务器端；  ServerSocket serverSocket = **new** ServerSocket(55555);    //2、由于是服务器端，则需要添加一步操作：通过Socket服务获取Socket，再获取其中的IO流  Socket socket = serverSocket.accept();    //3、建立连接后，通过Socket中的IO流(Socket流)进行数据的传输；  InputStream in = socket.getInputStream();    //获取上传文件在服务器端的那个文件输出流  FileOutputStream fos = **new** FileOutputStream("ASCII码表完整版copy.doc");    //反复使用一个字节接收  **int** b = -1;    **while**((b=in.read())!=-1){  //每上传一个字节，就写到服务器端这个文件中一个字节  //最终这个文件就接收了所有上传的数据，即文件上传  fos.write(b);  }    //关闭文件流对象  fos.close();    //获取输出流，输出响应数据  OutputStream out = socket.getOutputStream();  String content ="o";  out.write(content.getBytes());    //4、关闭socket。  socket.close();  serverSocket.close();  }  } |

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

4、网络编程概述

5、使用UDP协议完成数据通信

6、DatagramSocket和DatagramPacket的使用

7、使用TCP协议完成数据通信

8、套接字Socket和ServerSocket的使用