# 1. 网络编程概述

网络编程用于实现网络互连的不同计算机上运行的程序间进行数据交换。计算机网络体系结构：TCP/IP模型。建议先学习计算机网络知识。

网络通信的三要素：

（1）IP地址：网络中设备的标识。用于找到设备。

（2）端口号：设备中不同软件进程的标识，用于找到具体的应用进程。

（3）通信协议：就是通信的规则，常用的是TCP和UDP协议。我们编程关心的是应用层的协议。

网络编程还会涉及到IO和多线程的知识，IO用于数据的传输，多线程是为了处理多个网络请求。

# 2. InetAddress和套接字Socket

InetAddress其实就是Java封装了IP地址对象，而套接字（Socket）就是IP值和端口号的组合。通信的两端是通过套接字Socket进行连接，两个Socket间通过IO方式传输数据。

Java中，UDP和TCP的客户端/服务端Socket是有所区别的，下面讲。这里先讲下InetAddress，因为套接字Socket是要用到InetAddress的。

InetAddress类没有构造方法，可以使用类中的静态方法getByName()得到该类的对象：

public static InetAddress getByName(String host)：根据主机名或者IP地址的字符串形式得到InetAddress对象。

之后，通过getHostAddress()方法可得到此对象对应的主机地址；通过getHostName()方法得到主机名。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.net.InetAddress; **import** java.net.UnknownHostException;  **public class** Demo {  **public static void** main(String[] args) **throws** UnknownHostException {  InetAddress ipObj = InetAddress.*getByName*(**"192.168.1.1"**);  System.***out***.println(ipObj.getHostName());  System.***out***.println(ipObj.getHostAddress());  } } |

# 3. UDP通信

UDP协议是用户数据报协议，是无连接的传输层协议。

UDP通信不分服务端与客户端，只分为发送端与接收端。UDP通信的要点是：创建DatagramSocket，建立DatagramPacket数据包，最后调用receive()方法或者send()方法接受或者发送数据包。

发送端Socket连接接收端Socket就能建立连接。而接收端需要监听自己的端口，等待发送端连接，管理很多过来的请求。

下面是一个UDP简单的例子。主要是要看例子！

（1）UDP接收端程序：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException; **import** java.net.DatagramPacket; **import** java.net.DatagramSocket; **import** java.net.InetAddress;  **public class** MainReceive {  **public static void** main(String[] args) **throws** IOException {  *// 1. 创建UDP接收端Socket。  // 用DatagramSocket类，只要指明监听的本机端口即可。这里用10000* DatagramSocket ds = **new** DatagramSocket(10000);  *// 2.创建一个数据包作为接收数据的容器。  // 需要新建一个字节数组传递进去，并指明数组长度* **byte**[] data = **new byte**[4 \* 1024]; *// 4K的大小* DatagramPacket dp = **new** DatagramPacket(data, data.**length**);  *// 3. 用DatagramSocket接收数据到数据包中，用receive方法* ds.receive(dp);  */\*  API对receive()方法的解释：  从此套接字接收数据报包。当此方法返回时，DatagramPacket的缓冲区填充了接收的数据。数据报包也包含发送方的 IP 地址和端口号。此方法在接收到数据报前一直阻塞。数据报包对象的length 字段包含所接收信息的长度。如果信息比包的长度长，该信息将被截断。  \*/  // 也就是说此时若发送端传递了数据，那么DatagramPacket中不仅存放了数据，还存放了发送端信息   // 获得发送端的IP地址和端口信息：* InetAddress ip = dp.getAddress();  **int** port = dp.getPort();  *// 获得传送来的数据* **byte**[] receiveData = dp.getData();  *// 传送过来的是普通文本的话，可解析为字符串。注：dp.getLength()获得的是有效长度* String dataStr = **new** String(receiveData, 0, dp.getLength());   *// 输出上述的信息* System.***out***.printf(**"客户端地址：%s，端口：%d，数据：%s"**, ip.getHostAddress(), port, dataStr);  ds.close(); // 关闭  } } |

（2）发送端程序：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException; **import** java.net.\*;  **public class** MainSend {  **public static void** main(String[] args) **throws** IOException {  *// 1. 建立UDP发送端Socket，还是用DatagramSocket  // 发送端不需监听。直接创建对象。发送时才指定连接的Socket信息* DatagramSocket ds = **new** DatagramSocket();  *// 2. 同样，创建DatagramPacket对象用于设置发送的信息  // 使用的构造为：public DatagramPacket(byte[] buf, int length, InetAddress addr, int port)。正好包含了以下信息：数据、长度、远程主机IP和端口号* **byte**[] buf = **"hello 你好啊"**.getBytes();  DatagramPacket dp = **new** DatagramPacket(buf, buf.**length**, InetAddress.*getByName*(**"127.0.0.1"**), 10000);  *// 调用Socket的send()方法发送数据包* ds.send(dp);  *// 释放Socket连接* ds.close();  System.***out***.println(**"已发送！"**);  } } |

上述程序应该先运行MainReceive程序，因为recevice方法会阻塞，也就是如果没有程序连接过来，程序会一直等待。所以先运行MainReceive程序，再运行SendReceive程序发送数据。这样MainReceive程序中就会打印出传递过来的信息。注意的是，MainReceive程序使用的端口是10000，但是MainSend程序也有自己的端口，所以MainReceive输出的链接过来的程序的端口不是10000，而是系统自动分配的，这要清楚。

中文接收是没问题的，因为是字节全部转换的。

上述中，MainReceive只能监听一次程序数据的传递，一旦MainSend程序运行，MainReceive接收到数据后也会停止程序。我们可以使用while循环来让MainReceive一直receive进行监听。

在代码上看来，UDP不区分服务端和客户端，因此在编程时，服务端和客户端如果要互相传输数据，都是通过receive()和send()方法来接收和发送数据。下面是示例：

实现的功能为：服务端在循环中不断等待接收数据，若成功收到数据则给此客户端发送反馈数据；而客户端发送完数据并得到反馈后就结束程序。

服务端代码：

|  |
| --- |
| **package** com.demo.udp;  **import** java.io.IOException; **import** java.net.DatagramPacket; **import** java.net.DatagramSocket; **import** java.util.Arrays;  **public class** Server {  **public static void** main(String[] args) **throws** IOException, InterruptedException {  DatagramSocket ds = **new** DatagramSocket(10000);  **byte**[] data = **new byte**[1024]; *// 一次课接收/发送1024个字节* DatagramPacket dp = **new** DatagramPacket(data, data.**length**);  **while** (**true**) {  *// 在循环中接收和发送数据。* ds.receive(dp);  System.***out***.println(**new** String(dp.getData(), 0, dp.getLength())); *// 打印接收到的数据  // 向客户端发送反馈信息。发送的流程同样是设置数据并调用send()发送  // 但注意：  // 1. 发送反馈给客户端时，需要使用同一个DatagramPacket，即上面的dp，因为dp中有对方的信息，否则不知道对方是谁。  // 2. 不能直接使用dp.setData("server received".getBytes())这种方式  // 原因是一旦这样做，dp中可存储的数据长度只能是15个字节了。这样下次客户端发送超过15个字节的数据就会被截取。  // 因此我们要保留dp中可存数据的长度。我们通过将要发送的字节数据“拷贝”到原来的data数组中实现保留长度。* Arrays.*fill*(data, (**byte**) 0); *// 需要先把data先全部清空！！！否则下面可能会发送多余数据* **byte**[] sendData = **"server received"**.getBytes(); *// 待发送的数据* System.*arraycopy*(sendData, 0, data, 0, sendData.**length**); *// 拷贝数组到data中* dp.setData(data); *// 重新设置data数据* ds.send(dp); *// 发送* Thread.*sleep*(200);  }  *//ds.close(); // 上面用了while(true)，这里执行不到了...* } } |

客户端代码：

|  |
| --- |
| **package** com.demo.udp;  **import** java.io.IOException; **import** java.net.DatagramPacket; **import** java.net.DatagramSocket; **import** java.net.InetAddress;  **public class** Client {  **public static void** main(String[] args) **throws** IOException {  DatagramSocket ds = **new** DatagramSocket();  **byte**[] buf = **"我们建立了连接，传输了数据"**.getBytes();  DatagramPacket dp = **new** DatagramPacket(buf, buf.**length**, InetAddress.*getByName*(**"127.0.0.1"**), 10000);  ds.send(dp); *// 发送数据* System.***out***.println(**"数据已发送"**);  **byte**[] data = **new byte**[1024]; *// 也规定一次可传输1024字节* dp.setData(data);  ds.receive(dp); *// 接收数据。  // 下面接收数据的处理比较简单，只要得到获取的数据即可。* System.***out***.println(**"收到服务器数据："** + **new** String(dp.getData(), 0, dp.getLength()));  ds.close();  } } |

# 4. TCP通信

TCP是面向连接的、可靠的协议。一般连接的双方一个充当服务器，一个充当客户端。客户端请求服务器相应数据，而服务器返回给客户端数据。当然，两端都能发送和接收数据。

TCP通信的要点就是：创建Socket；拿到Socket中的输入/输出流；发送/接收此数据。

两端Socket有所区别，即服务端Socket（ServerSocket）和客户端Socket。

例子：

（1）TCP接收端：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.BufferedReader; **import** java.io.IOException; **import** java.io.InputStream; **import** java.io.InputStreamReader; **import** java.net.ServerSocket; **import** java.net.Socket;  **public class** MainReceive {  **public static void** main(String[] args) **throws** IOException {  *// 1. 创建服务器端Socket，指定监听端口。  // 这里使用的是ServerSocket类* ServerSocket ss = **new** ServerSocket(10000);  *// 2.用accept方法监听客户端连接，此方法返回一个TCP客户端Socket对象。直接用Socket类型接收  // 若无连接，则一直阻塞。* Socket client = ss.accept();  *// 与UDP不同，此时Socket中就有了需要的所有信息   // 获得IP和端口* String ip = client.getInetAddress().getHostAddress();  **int** port = client.getPort();  System.***out***.printf(**"IP：%s，端口：%d\n"**, ip, port);  *// 获得数据的输入流来读取传递过来的数据。* InputStream is = client.getInputStream();  *// IO方式读取数据。这里用BufferedReader包装一下，防止中文乱码* BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));  String data = **null**;  **while**((data = br.readLine()) != **null**) {  System.***out***.println(data);  }  *// 关闭client和server的Socket* client.close(); *// 会自动关闭上述的流* ss.close();  } } |

（2）TCP发送端：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException; **import** java.io.OutputStream; **import** java.net.Socket;  **public class** MainSend {  **public static void** main(String[] args) **throws** IOException {  *// 1. 创建客户端Socket，指定服务器的IP和端口* Socket s = **new** Socket(**"127.0.0.1"**, 10086);  *// 拿到Socket的输出流，写数据* OutputStream os = s.getOutputStream();  os.write(**"TCP 你好啊"**.getBytes());  *// 释放资源* s.close();  } } |

Socket也能拿到输出流OutputStream也能拿到InputStream，可以互相发送数据。我们在这里同样用TCP实现数据交互的示例：

实现的功能为：服务端在循环中不断等待接收数据，若成功收到数据则给此客户端发送反馈数据；而客户端发送完数据并得到反馈后就结束程序。

服务端代码：

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.ServerSocket; **import** java.net.Socket;  **public class** Server {  **public static void** main(String[] args) **throws** IOException, InterruptedException {  ServerSocket ss = **new** ServerSocket(10000);  **while** (**true**) {  Socket client = ss.accept();  *// 获得数据* InputStream is = client.getInputStream();  BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));  String data = **null**;  **while** ((data = br.readLine()) != **null**) {  System.***out***.println(data);  }  client.shutdownInput(); *// shutdownInput()，重要！！！  // 发送反馈数据* OutputStream os = client.getOutputStream();  os.write(**"data received"**.getBytes());  client.close();  Thread.*sleep*(200);  }  *// 关闭ServerSocket  //ss.close();* } } |

客户端代码：

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.Socket;  **public class** Client {  **public static void** main(String[] args) **throws** IOException {  Socket s = **new** Socket(**"127.0.0.1"**, 10000);  *// 1. 通过输出流和write()发送数据* OutputStream os = s.getOutputStream();  os.write(**"TCP 你好啊"**.getBytes());  s.shutdownOutput(); *// shutdownOutput()，重要！！！  // 2. 再通过输入流和read()读取反馈的数据* InputStream is = s.getInputStream();  BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));  String data = **null**;  **while** ((data = br.readLine()) != **null**) {  System.***out***.println(data);  }  *// 释放资源* s.close();  } } |

值得注意的是，在上述的服务器和客户端代码中，如果要在接收（或发送）后，还需要再发送（或接收）数据，那么必须要在此之前调用shutdownInput()或者shutdownOutput()方法，表示当前接收（或发送）数据操作已经完毕，可以进行下一波数据传递。否则，如果不使用shutdownInput()或者shutdownOutput()，将导致程序的IO阻塞，双方程序都无法正常进行。

实际上，上述的**服务器代码**还是有很大的改进空间的。因为Socket监听客户端（accept）、使用IO流发送和接收数据是阻塞的，只有有Socket连接、IO流可写和可读数据时，程序才会执行，否则一直是阻塞状态。为了能让服务器具有同时响应多个Socket连接的能力，一般服务器端需要进行多线程处理。例如改进为如下代码：

（1）HandleTcp线程类：接收客户端Socket，处理具体的IO。

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.Socket;  **public class** HandleTcp **extends** Thread {  **private** Socket **client**;   *// 接收客户端socket* **public** HandleTcp(Socket client) {  **this**.**client** = client;  }   @Override  **public void** run() {  **try** {  *// 获得数据* InputStream is = **client**.getInputStream();  BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));  String data = **null**;  **while** ((data = br.readLine()) != **null**) {  System.***out***.println(data);  }  **client**.shutdownInput(); *// shutdownInput()，重要！！！  // 发送反馈数据* OutputStream os = **client**.getOutputStream();  os.write(**"data received"**.getBytes());  } **catch** (Exception ex) {   } **finally** {  **try** {  **client**.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } } |

（2）TcpServer线程类：不断接收客户端Socket，然后再将Socket传递给HandleTcp具体处理。

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.IOException; **import** java.net.ServerSocket; **import** java.net.Socket;  **public class** TcpServer **extends** Thread {  **private** ServerSocket **serverSocket**;   *// 接收serverSocket* **public** TcpServer(ServerSocket serverSocket) {  **this**.**serverSocket** = serverSocket;  }   @Override  **public void** run() {  **try** {  **while** (**true**) {  Socket client = **serverSocket**.accept();  **new** HandleTcp(client).start();  }  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

（3）主类：创建TcpServer线程执行，同时也可运行其他代码。注意，监听端口new ServerSocket(10000)不能写在上述的循环中，因为只能new一次，否则会导致端口占用问题。

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.ServerSocket;  **public class** Server {  **public static void** main(String[] args) **throws** IOException {  ServerSocket serverSocket = **new** ServerSocket(10000);  **new** TcpServer(serverSocket).start();  System.***out***.println(**"其他程序正常运行"**);  } } |

上述多线程无需考虑线程安全问题，因为每个客户端Socket不是共享的，无共享变量。

# 1. 图形用户界面（GUI）

Java也可以开发图形用户界面（Graphics User Interface）的程序。使用图形界面程序更直观、更方便，所见即所得。不过我们学习的重点不在于图形用户界面程序，而是以后的互联网开发。因此这里是少量涉及，介绍某几个重要概念。不过更适合开发桌面应用程序的还是C、C++和.Net，效率高。

## 1.1 概述

GUI中常用的类在下面的两个包中（这里不涉及JavaFX）：

（1）java.awt包：Abstract Window ToolKit (抽象窗口工具包)，通过调用本地系统方法实现功能，和操作系统关联较强，属重量级组件。虽然Java是跨平台的，但使用awt包创建GUI程序时，在不同平台下显示的效果是有差别的，因为依赖的是平台本身的组件。

（2）javax.swing包：在AWT的基础上，建立的一套图形界面系统，其中的图形组件完全由Java实现。这样swing包中就是和操作系统平台无关的图形组件了，使用swing能让GUI程序在不同平台上展示的效果相同。swing在javax包下，在JDK中，所有在javax中的包都是“扩展包”。

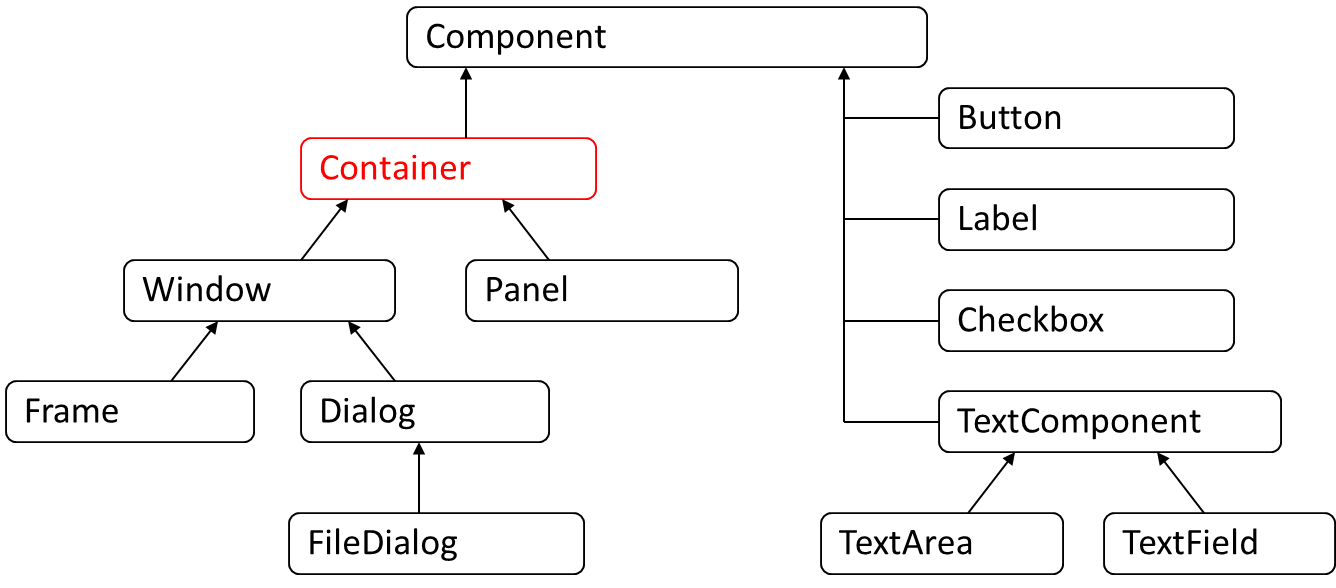
提倡多使用swing包下的图形组件。awt组件和swing组件的名称和使用方法是一致的，为了区分，swing中的所有类名前都加上了一个字母J。比如awt中的类Frame对应swing中的类JFrame。

几乎每个awt控件类都有对应的swing组件类，那么为什么不废除掉awt包呢？

原因有二：一是为了软件的兼容性；而是因为awt包中提供了事件处理(event)相关的类，这部分类并没有在swing中重新提供，因为各个平台的事件处理是有差异的，无法做到统一。

需要注意的是，图形界面不是学习的重点，重点在于其中的编程思想。

GUI的继承体系图：



通过上图可知，所有的组件都继承自Component，Component下有两类组件，一类是容器组件（Container），一类是非容器组件（Button、Label等）。容器组件用于“装”非容器组件的，因此先学容器组件，再在容器中“装”非容器组件进行展示。

## 1.2 一个简单的窗体

我们一般不使用Window类，因为API介绍说Window是没有边界的，因此我们用Frame类创建一个窗体。用setTitle()设置标题，用setSize()设置窗体大小等等。都可在API中查看。注意窗体默认是不显示的，需要调用setVisible()，设置可见性为true才能看到窗口。具体内容见代码：

|  |
| --- |
| **package** com.zhang.test;  **import** java.awt.\*;  **public class** Demo {  **public static void** main(String[] args) {  *// 新建窗体对象* Frame f1 = **new** Frame();  f1.setTitle(**"第一个窗体"**); *// 设置标题* f1.setSize(400, 300); *// 设置窗体宽和高* f1.setLocation(400, 200); *// 设置窗体左上角坐标。* f1.setBackground(Color.***darkGray***); *// 设置背景颜色* f1.setVisible(**true**); *// 设置窗体显示* } } |

其中setLocation(x, y)设置窗体左上角在屏幕中的坐标，那么坐标原点就是屏幕的左上角，向右是x轴，向下是y轴。setBackground中设置颜色，用Color对象中提供的默认变量即可。

实际上，setSize()方法中还能传递Dimension对象（尺寸对象），此对象实际就是封装了宽高属性；setLocation()方法中还能传递Point对象(坐标点对象)，此对象实际就是封装了x和y两个坐标值。例如：

|  |
| --- |
| **package** com.zhang.test;  **import** java.awt.\*;  **public class** Demo {  **public static void** main(String[] args) {  *// 新建窗体对象* Frame f2 = **new** Frame();  f2.setTitle(**"第而个窗体"**);  f2.setSize(**new** Dimension(400, 300));  f2.setLocation(**new** Point(400, 200));  f2.setVisible(**true**);  } } |

还可以直接用setBounds方法一次性设置好窗体的大小和坐标。

|  |
| --- |
| **package** com.zhang.test;  **import** java.awt.\*;  **public class** Demo {  **public static void** main(String[] args) {  *// Frame的构造可以直接设置标题* Frame f3 = **new** Frame(**"这是第三个窗体"**);  *// 设置x, y, width, height* f3.setBounds(400, 200, 400, 300);  f3.setVisible(**true**);  } } |

因此有时可多看看API，找到更好用的方法。

简单的窗体创建过程：Frame f = new Frame();f.setVisible(true);在显示之前可以设置标题、尺寸等。

有时，使用上面的awt包中类，显示中文有问题，可改用swing即显示正常。

## 1.3 设置窗体居中

窗体一般都是居中显示的，我们可设置窗体位置来让窗体居中。公式是（x, y） = （(屏幕宽度 - 窗体宽度) / 2, (屏幕高度 - 窗体高度) / 2）。

为了适应不同设备，我们需要获得设备的屏幕宽高（分辨率），用awt包下的ToolKit工具类。ToolKit是一个抽象类，但是它提供了一个getDefaultToolKit()方法来获得一个默认的工具箱。然后调用ToolKit对象的getSreenSize()方法即可得到Dimension对象。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** javax.swing.\*; **import** java.awt.\*;  **public class** Demo {  **public static void** main(String[] args) {  JFrame jFrame = **new** JFrame(**"我的窗体"**);  jFrame.setSize(1200, 600);  *setFrameCenter*(jFrame);  jFrame.setVisible(**true**);  }   */\*\*  \* 将窗体居中显示。窗体最好预先设置了宽高  \** ***@param jFrame*** *窗体  \*/* **public static void** setFrameCenter(JFrame jFrame) {  *// 先获得屏幕分辨率* Toolkit toolkit = Toolkit.*getDefaultToolkit*();  Dimension dimension = toolkit.getScreenSize();  *// 设置窗体居中* **int** x = ((**int**)dimension.getWidth() - jFrame.getWidth()) / 2;  **int** y = ((**int**)dimension.getHeight() - jFrame.getHeight()) / 2;  jFrame.setLocation(x, y);  } } |

## 1.4 事件监听机制

运行上面程序时，会发现点击窗体关闭按钮是没有反应的，只能在IDE中强制关闭程序（如果是JFrame，默认点击关闭按钮时，窗体会隐藏掉，但是程序还是没有被关闭，可在IDE中看到）。如果想让窗口关闭正常，那就就要进行事件处理。即要程序监听窗体关闭事件，并且程序要处理该事件。

在GUI中经常用到事件处理。事件处理有以下要点：

（1）拥有事件源，就是事件发生的地方，比如一个按钮。

（2）给事件源添加（注册）对某个事件的监听，这个事件是API定义好的，比如按钮点击事件、窗体关闭事件。

（3）给注册的监听事件编写事件处理，比如点击一个按钮要进行什么样的操作。

现在实现窗口的关闭。使用addWindowListener方法给事件源（即窗口）添加监听。此方法中传递一个WindowListener接口的实现类。此接口中就定义了窗体可以处理的各种事件。我们可以用匿名内部类。对于窗体关闭，我们只要处理WindowListener接口中的windowClosing事件，其他方法不需要实现。

|  |
| --- |
| **package** com.zhang.test;  **import** java.awt.\*; **import** java.awt.event.WindowEvent; **import** java.awt.event.WindowListener;  **public class** Demo {  **public static void** main(String[] args) {  *// Frame的构造可以直接设置标题* Frame f3 = **new** Frame(**"这是第三个窗体"**);  *// 设置x, y, width, height* f3.setBounds(400, 200, 400, 300);  f3.setVisible(**true**);  f3.addWindowListener(**new** WindowListener() {  @Override  **public void** windowOpened(WindowEvent e) {  }  @Override  **public void** windowClosing(WindowEvent e) {  *// 在此事件中关闭窗体。调用窗体对象的dispose()方法即可* f3.dispose();  }  @Override  **public void** windowClosed(WindowEvent e) {  }  @Override  **public void** windowIconified(WindowEvent e) {  }  @Override  **public void** windowDeiconified(WindowEvent e) {   }  @Override  **public void** windowActivated(WindowEvent e) {  }  @Override  **public void** windowDeactivated(WindowEvent e) {  }  });  } } |

上面就是在窗体关闭事件中关闭窗体，最后，程序将执行结束。

在处理事件时，能通过事件对象WindowEvent拿到事件源等对象以便使用，例如，上述的windowClosing方法中可以这么写：

|  |
| --- |
| @Override **public void** windowClosing(WindowEvent e) {  *// 通过WindowEvent拿到事件源对象  // 拿到的这个frame其实就是f3对象。* Frame frame = (Frame) e.getSource();  frame.dispose(); } |

对于窗体关闭，JDK提供了更方便的方法，即调用setDefaultCloseOperation(int operation)方法即可实现关闭窗体。其中operation可以传递WindowConstants接口中提供的常量值，即DO\_NOTHING\_ON\_CLOSE代表0，点击关闭时什么也不做；HIDE\_ON\_CLOSE代表1，即关闭时隐藏；DISPOSE\_ON\_CLOSE代表2，即关闭此窗口（若程序只剩下这一个窗口，就相当于关闭程序）；EXIT\_ON\_CLOSE代表3，即关闭此程序。Java中常用这些常量来限定某些参数值。这里常用的就是2和3（一般使用对应的常量名让代码更可读）。例子：

|  |
| --- |
| jFrame.setDefaultCloseOperation(WindowConstants.***DISPOSE\_ON\_CLOSE***); |

## 1.5 适配器Adapter

上面写事件处理时，要实现接口所有的方法，太麻烦。有什么方法能只要重写接口中的一个方法呢？这就通过适配器来实现。

实现适配器的步骤：

（1）首先定义好接口（上述程序中，此接口已经由JDK定义好）；

（2）用一个抽象类来实现此接口，实现接口的所有方法。注意是实现，里面可以不写什么，但是一定要实现。那么，当需要用到此接口的实现时，直接用此抽象类的实现类对象即可。此时，此抽象类的实现类需要重写什么方法，就重写什么方法，因为虽然是抽象类，但是他们都“实现”了，因此只要重写需要的方法即可。这样就实现了适配器，该抽象类就是适配器。

实际上对于WindowListener，JDK也写好了适配器WindowAdapter。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.awt.\*; **import** java.awt.event.WindowAdapter; **import** java.awt.event.WindowEvent;  **public class** Demo {  **public static void** main(String[] args) {  *// Frame的构造可以直接设置标题* Frame f3 = **new** Frame(**"这是第三个窗体"**);  *// 设置x, y, width, height* f3.setBounds(400, 200, 400, 300);  f3.setVisible(**true**);  f3.addWindowListener(**new** WindowAdapter() {  @Override  **public void** windowClosing(WindowEvent e) {  f3.dispose()  }  });  } } |

当然，基本上JDK也都提供了事件处理配套的适配器，除非此事件处理接口只有一个方法。

其他例子：用户点击一下按钮，就改变窗体的背景颜色（红和绿）。

|  |
| --- |
| **package** com.zhang.test;  **import** java.awt.\*; **import** java.awt.event.ActionEvent; **import** java.awt.event.ActionListener; **import** java.awt.event.WindowAdapter; **import** java.awt.event.WindowEvent;  **public class** Demo {  **public static void** main(String[] args) {  Frame f = **new** Frame(**"背景变色"**);  f.setBounds(400, 300, 400, 200);  *// 窗口关闭事件* f.addWindowListener(**new** WindowAdapter() {  @Override  **public void** windowClosing(WindowEvent e) {  f.dispose();  }  });  f.setBackground(Color.***red***);  f.setVisible(**true**);  *//设置布局方式为流式布局，否则下面的按钮会占据整个区域（了解即可）* f.setLayout(**new** FlowLayout());   *// 添加按钮* Button btn = **new** Button(**"Change Background Color"**);  f.add(btn); *// 向窗体中添加按钮  // 按钮点击事件 这个ActionListener就没有适配器* btn.addActionListener(**new** ActionListener() {  @Override  **public void** actionPerformed(ActionEvent e) {  **if**(f.getBackground() == Color.***red***) {  *// 红色就绿* f.setBackground(Color.***green***);  } **else** {  *// 变红* f.setBackground(Color.***red***);  }  }  });  } } |

## 1.6 对话框

使用Dialog来弹出对话框。JDialog的构造函数是：JDialog(Frame owner, String title, boolean modal)。

owner表示对话框所属窗体，可以为空；

title表示对话框标题；

modal表示模式，若为true，则程序不允许操作主窗体，只能关闭此对话框才能操作，如果为false，则对话框和主窗体都能操作。例子：

|  |
| --- |
| **public static void** main(String[] args) {  JFrame jFrame = **null**;  JDialog jDialog = **new** JDialog(jFrame, **"欢迎"**, **true**);  jDialog.setBounds(300, 200, 400, 300);  jDialog.setVisible(**true**); } |

注意，不能直接使用new Dialog(null, “欢迎”, true)，因为JDialog还有一个构造参数为：

public JDialog(Dialog owner, String title, boolean modal)。如果直接传递null，会不知道具体使用哪个构造函数。因此可以用一个窗体或者Dialog类型“包装”null。

一般，经常用JOptionPane来弹出“实用”的对话框。

（1）消息对话框。

使用静态方法：showMessageDialog(Component parentComponent, Object message, String title, int messageType)。参数1是父容器，可为null，参数2是消息字符串，参数3是标题，参数4用来规定消息框的类型，可用JOptionPane中的静态常量表示，有：

ERROR\_MESSAGE：错误对话框图标；

INFORMATION\_MESSAGE：信息对话框图标

WARNING\_MESSAG：警告对话框图标

QUESTION\_MESSAGE：危险对话框图标

PLAIN\_MESSAGE：无对话框图标

（2）确认对话框，使用方法：showConfirmDialog(Component parentComponent, Object message, String title, int optionType)。最后一个参数是确定对话框选项，查看文档即可知道传入参数。方法会根据用户选择的选项返回对应的数字，比如点击“是”则返回1。

（3）输入对话框：showInputDialog(...)，参数和上述方法类似，返回用户输入的值。若用户点击“取消”没有输入值，则返回值为null。

图形界面到此为止。可自行研究文件对话框、非容器组件、菜单和界面布局等内容。但是不建议深入研究，Oracle已经不再更新awt和swing了，有兴趣可了解下JavaFX。

# 5. 设计模式

## 5.1 设计模式简介

设计模式（Design pattern）是一套代码设计经验的总结。使用设计模式是为了可重用代码、保证代码可靠性。设计模式是一种思想。

设计模式和具体的语言无关，学习设计模式就是要建立面向对象的思想，尽可能的面向接口编程，低耦合，高内聚，使设计的程序可复用。

一共有23种设计模式，这里只介绍几种较常用的。

## 5.2 简单工厂模式

简单工厂模式就是定义一个具体的工厂类，其中提供静态方法，根据参数来创建并返回一个具体的对象实例。

这样客户端不负责对象的创建，明确了各个类的职责。缺点是由该工厂类创建所有对象，如果程序新增了类，就需要修改工厂类，不利于维护。

例子：现在用工厂类AnimalFactory来创建所有的动物对象。这里还有抽象类Animal、猫类Cat、狗类Dog。

（1）抽象类Animal

|  |
| --- |
| **package** com.zhang.test;  **public abstract class** Animal {  **private** String **name**; *// 名字  // 怎么说话的* **public abstract void** speak();   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  } } |

（2）猫类

|  |
| --- |
| **package** com.zhang.test;  **public class** Cat **extends** Animal {  @Override  **public void** speak() {  System.***out***.println(**"猫叫：miao~"**);  } } |

（3）狗类

|  |
| --- |
| **package** com.zhang.test;  **public class** Dog **extends** Animal {  @Override  **public void** speak() {  System.***out***.println(**"狗叫：wang!"**);  } } |

（4）AnimalFactory工厂类

|  |
| --- |
| **package** com.zhang.test;  **public class** AnimalFactory {  *// 根据参数的不同，返回不同的对象。但是统一用Animal返回。  // 使用者可以强转类型的。* **public static** Animal getAnimal(String type) {  **if**(type.equalsIgnoreCase(**"cat"**)) {  **return new** Cat();  }  **if**(type.equalsIgnoreCase(**"dog"**)) {  **return new** Dog();  }  *// 不能创建对象返回null* System.***out***.printf(**"ERROR：类%s不存在，未能创建对象。\n"**, type);  **return null**;  } } |

（5）主类测试

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  Cat cat = (Cat) AnimalFactory.*getAnimal*(**"cat"**);  Dog dog = (Dog) AnimalFactory.*getAnimal*(**"dog"**);  AnimalFactory.*getAnimal*(**"monkey"**); *// 会提示不能创建猴子类* cat.speak();  dog.speak();  } } |

## 5.3 工厂方法模式

工厂方法模式是简单工厂的改进，但是需要额外编写代码。

工厂方法模式中，先定义一个抽象的总的工厂类，提供抽象方法创建对象。然后，为每个具体的类都编写工厂类，每个具体的工厂类继承自抽象的总工厂类，实现其中的创建方法。

这样以后可以统一用抽象的工厂类接收一个具体的工厂类来产生对象。

当以后程序中有新的类时，需要再写下此类的工厂类即可，不需要对总的工厂类进行修改，提高了可维护性。

例：将上面的简单工厂改变为工厂方法模式。需要重写总的工厂类和各个具体类的工厂。而Animal、Cat和Dog类不需要改变。

（1）先改变AnimalFactory为总的抽象工厂（用接口也行）：

|  |
| --- |
| **package** com.zhang.test;  **public abstract class** AnimalFactory {  *// 此时不需要传递类型参数* **public abstract** Animal getAnimal(); } |

（2）Cat工厂：

|  |
| --- |
| **package** com.zhang.test;  **public class** CatFactory **extends** AnimalFactory {  @Override  **public** Animal getAnimal() {  **return new** Cat();  } } |

（3）Dog工厂：

|  |
| --- |
| **package** com.zhang.test;  **public class** DogFactory **extends** AnimalFactory {  @Override  **public** Animal getAnimal() {  **return new** Dog();  } } |

（4）主类测试：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 猫工厂* AnimalFactory catFactory = **new** CatFactory();  *// 狗工厂* AnimalFactory dogFactory = **new** DogFactory();   Cat cat = (Cat) catFactory.getAnimal();  Dog dog = (Dog) dogFactory.getAnimal();  cat.speak();  dog.speak();  } } |

## 5.4 单例模式

单例模式就是确保该类的对象在程序中只能有一个。也就是说，该类的实例必须在类中创建，并且返回给外部使用。

在系统内存中只存在该类的一个对象，可以节约系统资源，对于一些需要频繁创建和销毁的对象，单例模式可以提高系统的性能。要考虑的是此类是否适合用作单例，因为单例同一时刻只存储一份对象数据。

为了保证只有一个对象呗创建，那么需要将此类构造函数私有化，禁止外界创建变量，在类的内部，直接用一个静态成员变量接收此类的对象。最后，通过一个类的静态方法向外界返回此成员变量。

根据此对象的创建时机不同，可分为恶汉式单例和懒汉式单例。

### 5.4.1 恶汉式单例

恶汉式单例就是在类加载使就创建对象，直接new一个本类对象给静态变量。例如一个学生单例（只是例子，学生不适合用单例）：

|  |
| --- |
| **package** com.zhang.test;  **public class** Student {  *// 私有化构造* **private** Student() {}  **private** String **name**;  *// 直接创建本类对象，并赋值给静态成员，用private修饰，避免外界直接拿到* **private static** Student *student* = **new** Student();  *// 对外界提供获得对象方法* **public static** Student getInstance() {  **return** *student*;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  } } |

测试例子输出结果为true，因为是同一个对象。

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  Student stu1 = Student.*getInstance*();  Student stu2 = Student.*getInstance*();  System.***out***.println(stu1 == stu2);  } } |

### 5.4.2 懒汉式单例

懒汉式单例就是在第一次调用getInstance()方法时才创建对象。实现方式就是先将静态变量设置为null，在getInstance()方法内部再给静态变量赋值（需要进行为空判断）。即：

|  |
| --- |
| **package** com.zhang.test;  **public class** Student {  *// 私有化构造* **private** Student() {}  **private** String **name**;  *// 先设置为null* **private static** Student *student* = **null**;   **public static** Student getInstance() {  *// 这里新建对象，需要判断* **if**(*student* == **null**){  *student* = **new** Student();  }  **return** *student*;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  } } |

不过懒汉式有线程安全问题，因为如果第一次调用时，有多个线程同时进入了getInstance()方法，那么就会创建多个Student实例，因此需要在getInstance()方法上加上synchronized关键字即可。

### 5.4.3 Runtime类

JDK中，Runtime类就是典型的单例模式。看Runtime类的源码，可以知道它采用的是恶汉式单例。

可以使用Runtime对象的exec()方法来执行系统命令，比如：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  Runtime runtime = Runtime.*getRuntime*();  runtime.exec(**"notepad"**); *// 会打开Windows记事本* } } |

执行部分命令时，会出错，提示需要更高的权限，那么执行时，只需要在命令前加上“cmd /c”即可，比如：runtime.exec("cmd /c diskpart")。

Runtime对象还有以下方法：

（1）exit(status)：退出虚拟机

（2）long freeMemory()：获取可用的内存数

（3）gc()：指示调用垃圾回收器

（4）long maxMemory()：获取JVM最大内存容量

（5）long totalMemory()：获取JVM总内存

## 5.5 模板设计模式

模板设计模式就是在一个抽象类中定义一个通用的方法骨架，在方法中调用一个抽象的方法。该抽象方法的具体实现由具体的子类实现，这样既能执行通用的步骤，又能根据需要执行不同的具体步骤。

例1：计算不同代码的运行时间。

（1）计算时间的抽象类CodeRunTime。

|  |
| --- |
| **package** com.zhang.test;  **public abstract class** CodeRunTime {  *// 这个通用的计算代码运行时间的方法。不允许修改，用final* **public final long** getRunTime() {  **long** startTime = System.*currentTimeMillis*();  *// 运行“代码”* code();  **long** endTime = System.*currentTimeMillis*();  **return** endTime - startTime;  }   *// 这是“代码”。用抽象方法代替。由子类实现* **public abstract void** code(); } |

（2）具体的不同的代码所在的类。

|  |
| --- |
| **package** com.zhang.test; *// 继承CodeRunTime* **public class** CodeDemo **extends** CodeRunTime {  @Override  **public void** code() {  *// 要执行的代码* **int** sum = 0;  *// 循环太少的话，执行时间是0。只怪咱电脑太快。* **for**(**int** i = 0; i < 200000000; i+=1) {  sum += i;  }  } } |

如果有其他不同的代码类，还可以写类继承抽象类CodeRunTime。

（3）主类测试：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 可得到代码运行的时间* CodeDemo crt = **new** CodeDemo();  System.***out***.println(**"时间："** + crt.getRunTime());  } } |

很多事可以用模板来做，当这类事的基本步骤相同，只是一部分不同时。

## 5.6 装饰设计模式

装饰设计模式，用构造方法传递的方式，传递需要装饰的对象。这个具体类继承同一个接口，可以增强B对象的方法功能。这是继承的替代方法，用于方法增强。

使用装饰模式，可以提供比继承更灵活的扩展对象的功能，它可以动态的添加对象的功能，并且可以随意的组合这些功能。

例子：Phone类有拨打电话的功能。现在给产品加上铃声和录音的功能。

（1）手机接口：

|  |
| --- |
| **package** com.zhang.test;  **public interface** Phone {  **public void** call() ; } |

（2）普通手机：

|  |
| --- |
| **package** com.zhang.test;  **public class** SimplePhone **implements** Phone {  @Override  **public void** call() {  System.***out***.println(**"打电话"**);  } } |

（3）带铃声的手机

|  |
| --- |
| **package** com.zhang.test; *// 带有铃声的手机* **public class** RingPhone **implements** Phone {  **private** Phone **p**;    **public** RingPhone(Phone p) {  **this**.**p** = p;  }  **public void** call() {  System.***out***.println(**"铃声"**);  **p**.call();  } } |

（4）带录音的手机

|  |
| --- |
| **package** com.zhang.test; **public class** RecPhone **implements** Phone {  **private** Phone **p**;  **public** RecPhone(Phone p) {  **this**.**p** = p;  }  *// 打电话功能* **public void** call() {  System.***out***.println(**"我这手机能录音"**);  **p**.call();  } } |

（5）主类测试。可以根据需要进行组合。

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 只有铃声的手机* Phone ringPhone = **new** RingPhone(**new** SimplePhone());  ringPhone.call();  *// 带铃声和录音的手机* Phone recRingPhone = **new** RingPhone(**new** RecPhone(**new** SimplePhone()));  recRingPhone.call();  } } |

也就是每个类装饰完之后，他还是手机。其实装饰设计模式见过，就是在IO流中，比如：

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));