# 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.udp;  **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.udp;  **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。

# 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));