**设计模式之Chain of Responsibility(职责链)**

[**板桥里人**](http://www.jdon.com/aboutme.htm) http://www.jdon.com 2002/04/21(转载请保留)

**Chain of Responsibility定义**  
Chain of Responsibility(CoR) 是用一系列类(classes)试图处理一个请求request,这些类之间是一个松散的耦合,唯一共同点是在他们之间传递request. 也就是说，来了一个请求，A类先处理，如果没有处理，就传递到B类处理，如果没有处理，就传递到C类处理，就这样象一个链条(chain)一样传递下去。

**如何使用?**  
虽然这一段是如何使用CoR,但是也是演示什么是CoR.

有一个Handler接口:

public interface Handler{  
　　public void handleRequest();  
}

这是一个处理request的事例， 如果有多种request,比如 请求帮助 请求打印 或请求格式化：  
  
最先想到的解决方案是：在接口中增加多个请求：  
public interface Handler{  
　　public void handleHelp();  
　　public void handlePrint();  
　　public void handleFormat();

}

具体是一段实现接口Handler代码：  
public class ConcreteHandler implements Handler{  
　　private Handler successor;

　　public ConcreteHandler(Handler successor){  
　　this.successor=successor;  
}

　　public void handleHelp(){  
　　　　//具体处理请求Help的代码  
　　　　...  
　　}

　　public void handlePrint(){  
　　　　//如果是print 转去处理Print  
　　　　successor.handlePrint();  
　　}  
　　public void handleFormat(){  
　　　　//如果是Format 转去处理format  
　　　　successor.handleFormat();  
　　}

}  
一共有三个这样的具体实现类，上面是处理help,还有处理Print 处理Format这大概是我们最常用的编程思路。

虽然思路简单明了，但是有一个扩展问题，如果我们需要再增加一个请求request种类,需要修改接口及其每一个实现。

第二方案:将每种request都变成一个接口，因此我们有以下代码 ：

public interface HelpHandler{  
　　public void handleHelp();  
}

public interface PrintHandler{  
　　public void handlePrint();  
}

public interface FormatHandler{  
　　public void handleFormat();  
}

public class ConcreteHandler  
　　implements HelpHandler,PrintHandler,FormatHandlet{  
　　private HelpHandler helpSuccessor;  
　　private PrintHandler printSuccessor;  
　　private FormatHandler formatSuccessor;

　　public ConcreteHandler(HelpHandler helpSuccessor,PrintHandler printSuccessor,FormatHandler 　　　　　　　　　　　　formatSuccessor)  
　　{  
　　　　this.helpSuccessor=helpSuccessor;  
　　　　this.printSuccessor=printSuccessor;  
　　　　this.formatSuccessor=formatSuccessor;  
　　}

　　public void handleHelp(){  
　　　　.......  
　　}

　　public void handlePrint(){this.printSuccessor=printSuccessor;}

　　public void handleFormat(){this.formatSuccessor=formatSuccessor;}

}

这个办法在增加新的请求request情况下，只是节省了接口的修改量，接口实现ConcreteHandler还需要修改。而且代码显然不简单美丽。

解决方案3: 在Handler接口中只使用一个参数化方法：  
public interface Handler{  
　　public void handleRequest(String request);  
}  
那么Handler实现代码如下：  
public class ConcreteHandler implements Handler{  
　　private Handler successor;

　　public ConcreteHandler(Handler successor){  
　　　　this.successor=successor;  
　　}

　　public void handleRequest(String request){  
　　　　if (request.equals("Help")){  
　　　　　　//这里是处理Help的具体代码  
　　　　}else  
　　　　　　//传递到下一个  
　　　　　　successor.handle(request);

　　　　}  
　　}

}

这里先假设request是String类型，如果不是怎么办？当然我们可以创建一个专门类Request

最后解决方案:接口Handler的代码如下：  
public interface Handler{  
　　public void handleRequest(Request request);  
}  
Request类的定义:  
public class Request{  
　　private String type;  
  
　　public Request(String type){this.type=type;}

　　public String getType(){return type;}

　　public void execute(){  
　　　　//request真正具体行为代码  
　　}  
}  
那么Handler实现代码如下：  
public class ConcreteHandler implements Handler{  
　　private Handler successor;

　　public ConcreteHandler(Handler successor){  
　　　　this.successor=successor;  
　　}

　　public void handleRequest(Request request){  
　　　　if (request instanceof HelpRequest){  
　　　　　　//这里是处理Help的具体代码  
　　　　}else if (request instanceof PrintRequst){  
　　　　　　request.execute();  
　　　　}else  
　　　　　　//传递到下一个  
　　　　　　successor.handle(request);

　　　　}  
　　}

}

这个解决方案就是CoR, 在一个链上,都有相应职责的类,因此叫**Chain of Responsibility**.

**CoR的优点：**  
因为无法预知来自外界（客户端）的请求是属于哪种类型，每个类如果碰到它不能处理的请求只要放弃就可以。

缺点是效率低，因为一个请求的完成可能要遍历到最后才可能完成，当然也可以用树的概念优化。 在Java AWT1.0中，对于鼠标按键事情的处理就是使用CoR,到Java.1.1以后，就使用Observer代替CoR

扩展性差，因为在CoR中，一定要有一个统一的接口Handler.局限性就在这里。

**与Command模式区别：**

Command 模式需要事先协商客户端和服务器端的调用关系，比如 1 代表 start 2 代表 move 等，这些 都是封装在 request 中，到达服务器端再分解。

CoR 模式就无需这种事先约定，服务器端可以使用 CoR 模式进行客户端请求的猜测，一个个猜测 试验。