设计模式之 Chain of Responsibility(职责链)

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。无疑这降低了类之间的耦合性。

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。