第一部分

仙人指路
——设计模式简介

Java 程序员工资高吗?

年薪 10 万是一个很正常的薪资水平，你的目标应该是年薪 30 万+，这样就可以在北京、深圳、上海等城市拥有自己的房子、车子。可以让垂垂老矣的父母更好的享受人生；经常带着爱人去看爱情海...

我为什么学不好，找不到高薪工作？

也许你会在学无所用之中度过一生，因为你不知道大型企业现在和未来需要什么样的人才，**方向错了一切努力等于白费**。

如何才能找到高人指点迷津？

Java 学习群：72030155，一年 365 天，每天晚上 8：30-11:30 有**腾讯、百度、阿里内部首席架构师来免费讲课**

我们为什么每天都要免费讲课？

1、口碑宣传

2、塑造品牌

3、免费课程都很棒，那么有更好的收费课程，你也会选择的

第 1 章　设计模式概述

1.1　设计模式是什么

战国·邹·孟轲《孟子·离娄上》说："离娄之明，公输子之巧，不以规矩，不能成方圆。"这就是说，做任何事都要有一定的规矩、规则、做法，否则无法成功。

基于此，人们在设计工作中逐渐摸索出一些规则。这些规则来源于设计实践中的共性和本质。

因而，"一个设计模式就是一个已被记录的最佳实践或一个解决方案，这个最佳实践或解决方案已被成功应用在许多环境中，它解决了在某种特定情境中重复发生的某个问题。"[1]

为了解决软件企业设计水平不高的问题，专家和富有经验的设计者以设计模式形式将他们的经验传授给我们，我们通过学习，消化和吸收设计模式的精华。最终，既学会了好的设计，也提升了软件开发的质量。

虽然，软件的设计模式是从欧美传到中国，但是设计模式中的许多内容都相当符合中国文化的思维模式。因此，作者祝愿读者在阅读本文之后，可以从学习模式发展到应用模式，最终可以达到创造自己模式的境界。

1.2　软件设计模式的发展历程

建筑领域的建筑师 Christopher Alexander 于 20 世纪 70 年代后期对建筑设计的相似性进行了研究，他发现并创造了模式这一概念。

软件模式从建筑师 Christopher Alexander 的思想上进化而来。Kent Beck 和 Ward Cunningham 将 Christopher Alexander 的思想应用到软件领域，他们记下了最初的一些模式（UI 方面的）；第一个发表的关于在开发中使用模式的著述是 Erich Gamma 于 1991 年发表的一篇论文；而最著名的则是 Erich Gamma、Richard Helm、Ralph Johnson 、John Vlissides 这四位作者合作编写的《设计模式：可复用面向对象软件的基本》，这篇文章发表于 1995 年。从此之后，软件设计模式研究进入了空前的繁荣期，在软件工程的各个领域被不断应用。

1.3　作者阐述软件设计模式的主要方式

设计模式，在软件工程方面使项目经理、系统设计师、系统开发人员等角色可以快速地复用成功的设计方案或设计体系。

为了清晰地阐述设计模式，本书主要从名称、结构、解决方案以及效果方面进行重点讲解。

---

[1] Partha Kuchana. Java 软件体系结构设计模式标准指南. 王卫军，楚宁志，译. 北京：电子工业出版社，2006，2

名称：包括模式的名字或别名，它可以帮助我们思考，方便我们与他人进行沟通。
结构：包括 UML 的各种图形、对象之间的交互关系以及 Java 代码等等。
解决方案：解决实际软件设计问题的元素组合。
效果：描述模式应用的效果及具体问题具体使用相应模式的策略。

第二部分

设计红宝书
——设计模式原则详解

第 2 章　设计原则之开闭原则

2.1　何谓开闭原则

开闭原则（Open Closed Principle，OCP）是指“软件实体应当对扩展开放，对修改关闭（Software entities should be open for extension, but closed for modification）”。[1]这个概念是大师 Bertrand Meyer 在 1988 年提出的。

“对于扩展是开放的”这意味着模块的行为是可以扩展的。“对于修改是封闭的”对模块行为进行扩展时，不必改动模块的源代码或者二进制代码。[2]

我个人认为开闭原则从软件系统的角度来说：软件系统中包含的各种功能组件（例如菜单(menu)、模块（Modules）、类（Classes）以及功能（Functions）等等）应该在不修改原有代码的基础上，添加新功能。同时，对于经常发生变化的有关值，可以进行封装。至于开闭原则中的“开”，可以认为对于组件功能的扩展是开放的，是允许对其进行功能扩展的。开闭原则中的“闭”，可以认为对于原有代码的修改是关闭的，即不需要改动原有的代码。

简而言之，我们可以把开闭原则理解成——可以根据需求随意增加新的类，但是不要修改原有的类。

例如，软件公司人力资源部现有绩效专员、培训发展专员和薪酬福利专员，由于业务发展的需要，缺少一个人事助理。此时，人力资源部部门经理通过招收一个新人任职“人事助理”便可满足变化了的需求，而不必变动人力资源部的原有人员，如图 2.1 所示。

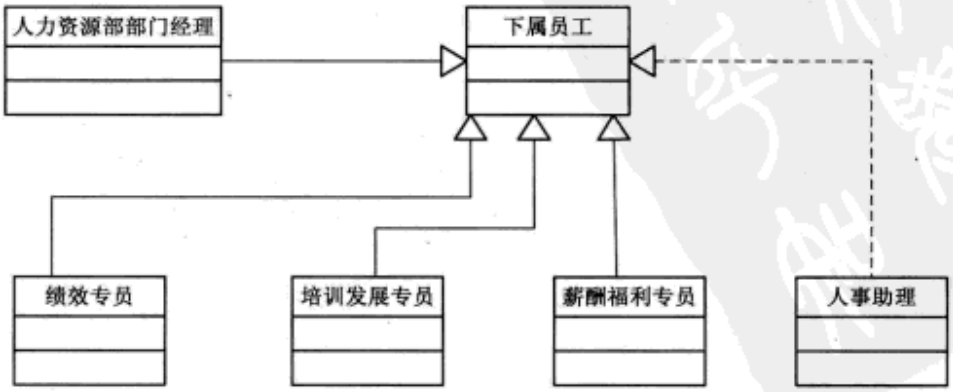

图 2.1

[1] Bertrand Meyer. Object-Oriented Software Construction. New Jersey：Prentice Hall，1988，23
[2] Robert C. Martin. 敏捷软件开发：原则、模式与实践. 邓辉，译. 北京：清华大学出版社，2003，9

2.2 为何要遵循开闭原则

为什么要遵循开闭原则呢？因为开闭原则是一切设计原则的基础，它是判断面向对象设计是否正确的最基本的原理之一。它在软件可用性上，非常灵活。它可以通过不断的增加新模块满足不断变化的新需求！并且由于开闭原则规定对软件原来的模块不要修改，因此不用担心软件的稳定性。如果一个软件项目某些功能已不符合新的需求了，我们可以重新开发，并且只需将现有的功能替换掉。比如说财政系统的算法发生了一些变化，我们可以在不改变原有代码的情况下，重新实现一个新算法将原有的算法替换下来。比如说一个拥有“增加”、“删除”、“查询”功能的模块，现在要增加“修改”功能，此时，我们只需要新开发“修改”功能，而不需要改变原有模块的内容。

总之，如果一个软件系统遵循开闭原则去设计，那么它至少具备以下好处：

- 可扩展程度高，非常灵活。无论是在软件开发过程中，还是在软件开发完成以后，我们都可以在软件中进行扩展，加入新的功能。如此一来，软件可以随着不断增加新模块满足不断变化的新需求！
- 可维护性强，无须修改原有代码。因此，变化的软件系统有一定的稳定性和延续性。

2.3 如何实现开闭原则

实现开闭原则的核心之处在于“抽象”。由于软件项目中，需求变更比较频繁，这要求我们系统设计师在设计时，要区分那些是变化的部分，那些是不变的部分。对于不变部分，我们可以把这些不变的部分加以抽象成不变的接口。对于变化部分，我们可以进行评估和分类，每一个可变的因素都单独进行封装。

即便如此，由于需求的变化，设计师在设计阶段要整理清楚所有的系统行为，这本身是不现实的。正如，歌手陈旭在“哥只是一个传说”一歌中所唱的“请你不要再迷恋哥，哥只是一个传说”。为此，我们只能在某些组件，在某种程度上符合开闭原则的要求。

当然，歌手许美静在“阳光总在风雨后”一歌中所唱的“阳光总在风雨后，请相信有彩虹”也相当感人。我们软件系统建设中，可以尽量地达到开闭原则。

至于开闭原则的具体抽象实现方面，可以通过 Java 的抽象类与接口去展现。对于软件系统的功能扩展，我们可以通过继承、重载或者委托等手段实现。

以抽象类为例，对于一些具备相同功能的类进行抽象处理，将公用部分的功能放入抽象类中，所有的操作可以调用子类。这样，如果对系统进行功能扩展时，只需根据抽象类去实现新的子类。

以接口为例，接口定义子类需要实现的接口函数。这样，如果要改换原有的实现，只需要改换一个实现类。

关于开闭原则的应用，在本书后面各个设计模式中都有充分的体现。

2.4 应用反思——出售鞋类

某商店需要开发商品应用系统，以方便管理。其中，以出售鞋类为例，商店的鞋类分男鞋和

女鞋，需要进行统一的折扣管理，如图 2.2 所示。

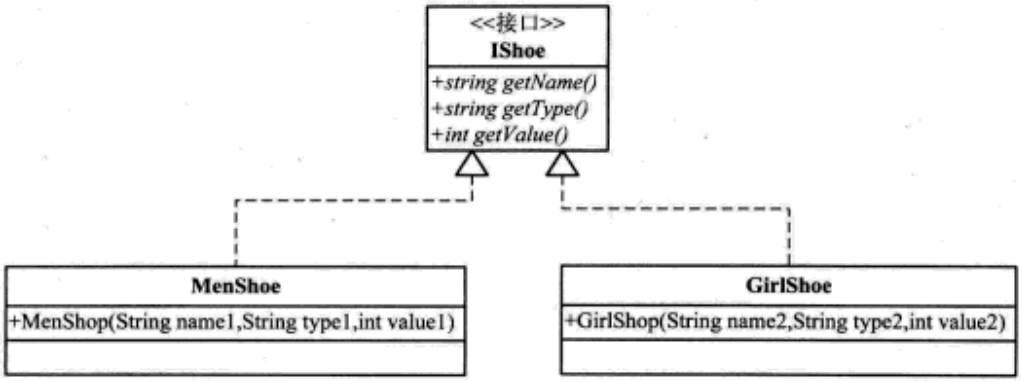

图 2.2

本应用反思的工程名为“程序 2.4.1”，源代码如下所示。

1. IShoe 定义了三个数据属性：鞋名、鞋类、价格

```
package principle.ocp;

public interface IShoe {
    public String getName();//鞋名
    public String getType();//鞋类
    public int getValue();//价格
}
```

2. MenShoe 实现接口 IShoe

```
package principle.ocp;

public class MenShoe implements IShoe
{
    private String name;//男鞋名
    private String type;//男鞋类
    private int value;//男鞋价格
    public  MenShoe(String name1,String type1,int value1)
    {
        this.name = name1;
        this.type = type1;
        this.value = value1;
    }
    public String getName() {
        return this.name;
    }
    public String getType() {
        return this.type;
    }
    public int getValue() {
        return this.value;
```

```
    }
}
```

3. GirlShoe 实现接口 IShoe

```
package principle.ocp;

public class GirlShoe implements IShoe {
    private String name;//女鞋名
    private String type;//女鞋类
    private int value;//女鞋价格
    public  GirlShoe(String name2,String type2,int value2)
    {
        this.name = name2;
        this.type = type2;
        this.value = value2;
    }
    public String getName() {
        return this.name;
    }
    public String getType() {
        return this.type;
    }
    public int getValue() {
        return this.value;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setType(String type) {
        this.type = type;
    }
    public void setValue(int value) {
        this.value = value;
    }
    }
```

由于新店开张，为了提高人气，扩大商店知名度，店老板决定对鞋类进行降价促销：

- 男鞋原价大于 200 元打 8 折，原价大于 150 元打 8.5 折，其他的价格一律打 9 折。
- 女鞋原价大于 200 元打 7.5 折，原价大于 150 元打 8 折，其他的价格一律打 8.5 折。

对此，我们应该如何设计比较合理呢？请看下面 3 个方案的比较：

方案一：在接口 IShoe 中增加 GetPriceCut ()用于处理降价，此时 MenShoe 类和 GirlShoe 类也需要修改。此方式不符合开闭原则，并且改动量比较大。

方案二：在 MenShoe、GirlShoe 中各增加一个方法，相对方案一要好一些，但是还不是最好的方案。

方案三：增加新的两个类，比如增加 PriceCutMenShoe 用于处理男鞋的降价，PriceCutGirlShoe 用于处理女鞋的降价计算。此方案，修改量少，可扩展性强，最符合开闭原则。

根据方案三的要求，原类图变为如图 2.3 所示。

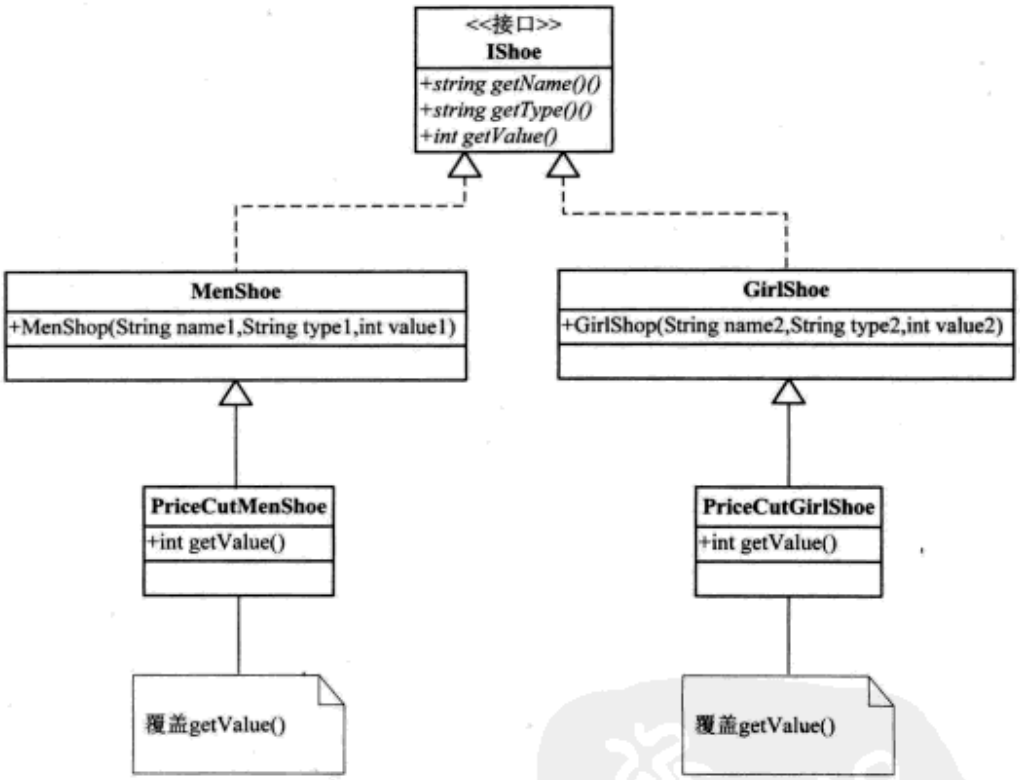

图 2.3

不修改原有工程代码，新增 PriceCutMenShoe 类。

本应用反思的工程名为“程序 2.4.2”，源代码如下所示。

（1）PriceCutMenShoe 继承 MenShoe 类，并扩展了功能。

```
package principle.ocp;

public class PriceCutMenShoe  extends MenShoe
{
    public PriceCutMenShoe(String name1, String type1, int value1)
    {
        super(name1, type1, value1);
    }
   public int getValue()
   {
    int primeCost=super.getValue();
    int cutPrice=0;
    if(primeCost>200)
```

```
        {//如果原价大于200元，则打8折
            cutPrice=cutPrice*80/100;
        }
        if(primeCost>150)
        {//如果原价大于150元，则打8.5折
            cutPrice=cutPrice*85/100;
        }
        else//其他，则打9折
        {
            cutPrice=cutPrice*90/100;
        }
            return cutPrice;
        }
    }
```

（2）PriceCutGirlShoe 继承 GirlShoe 类，并扩展了功能。

不修改原有代码，新增的 PriceCutGirlShoe 类如下：

```
package principle.ocp;

public class PriceCutGirlShoe extends GirlShoe
{

    public PriceCutGirlShoe(String name2, String type2, int value2)
    {
        super(name2, type2, value2);
    }
    public int getValue()
    {
    int primeCost=super.getValue();
    int cutPrice=0;
    if(primeCost>200)
    {//如果原价大于200元，则打7.5折
        cutPrice=cutPrice*75/100;
        System.out.println("如果原价大于200元，则打7.5折");
    }
    if(primeCost>150)
    {//如果原价大于150元，则打8折
        cutPrice=cutPrice*80/100;
        System.out.println("如果原价大于150元，则打8折");
    }
    else//其他，则打8.5折
    {
        cutPrice=cutPrice*85/100;
        System.out.println("其他，则打8.5折");
    }
    return primeCost;
    }
```

```
}
```

（3）ocp.jsp，验证开闭原则。

```
<%@ page contentType="text/html;charset=GB2312"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312"
/>
        <title>开闭原则</title>
    </head>
    <body>
        <%@ page pageEncoding="GB2312" language="java" import="principle.ocp.*"%>

        <%
            PriceCutGirlShoe a = new PriceCutGirlShoe("女鞋名", "女鞋类型", 160);
            out.println("鞋价为:" + a.getValue());
        %><br />
    </body>
</html>
```

在浏览器上打开页面，执行代码，显示结果如下所示：

```
鞋价为：160
```

2.5 开闭原则与 Struts

Struts 运用 MVC（Model-View-Controller，模型-视图-控制器）技术处理各类任务和事件，其控制器由 ActionServlet 和 Action 构成。

ActionServlet 采以 J2EE 核心控制器中的前端控制器模式为应用系统提供了一个强有力的中心支撑点，用于控制。这里，所谓前端控制器，它主要负责的事情是处理全体用户的相关请求，当控制器对有关请求进行校验与通过之后，可运用重定向的方式才可访问所需对象。

Action 则主要用于业务模型的调用，并承担客户端请求与业务逻辑之间的交互控制。其业务的框架内容细节由模型去打理。

如此一来，此类清晰的分层结构就能良好地实现开闭原则在软件系统中的应用。

第 3 章　单一职责原则

3.1　何谓单一职责原则

单一职责原则（single responsibility principle，SRP）是指“就一个类而言，应该仅有一个引起它变化的原因。”[1]

换言之，一个类，只需按职责进行功能设计。此原则从软件工程角度分析，符合“高内聚低耦合”的标准。并且，我们在接口的设计方面也同样符合单一职责的设计原则。

为了更好地理解此项原则，我们可以从以下两点进行分析：

- 防止相同类型的职责，分离到不同的类中。即我们需要提高代码的可重用性。
- 同一个类无须编制多余的职责。即做事专一，遵守中华民族“从一而终”的优良品德。

比如，足球场上的裁判执行竞赛规则，主裁只能担任主裁，助理裁判员只能担任助理裁判员，而不能既担任主裁又担任助理裁判员。下面我们分别来看一下违反 SRP 和符合 SRP 的设计是什么样子的。

（1）违反 SRP 的设计，如图 3.1 所示。

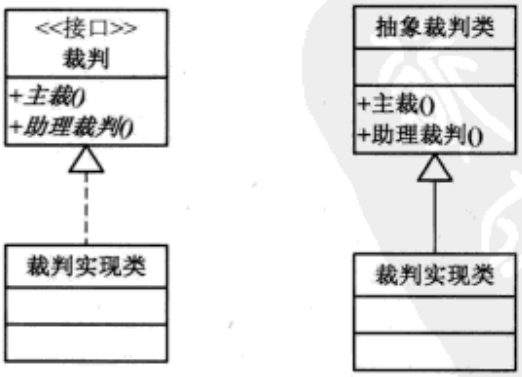

图 3.1

（2）符合 SRP 的设计，如图 3.2、图 3.3 所示。

[1] Robert C. Martin. 敏捷软件开发：原则、模式与实践. 邓辉，译. 北京：清华大学出版社，2003，88

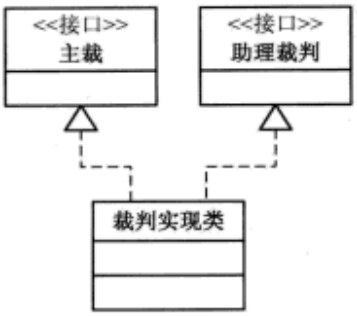

图 3.2

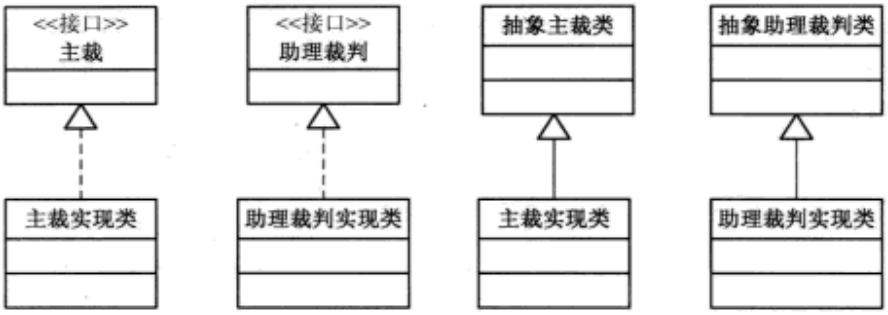

图 3.3

图 3.2 中一个“裁判实现类”实现主裁接口和助理裁判接口，此方法并无不可。然而，图 3.3 则更符合单一职责原则，并且实际设计开发中比较常见。因而，我推荐使用图 3.3 所示的这种一一对应的方法。

3.2 为何遵循单一职责

现阶段，我国软件行业的需求在不断的变化。如果一个类或接口拥有超过一个以上的职责，那么其中一个职责的变化将有可能导致其他职责的弱化。

就像一个公司的老总，我们不能指望把他既当管理人员又当技术专家、业务专家、公关专家，我必须对他的职责进行明确。老总就是老总，只要他做好管理职责，我们就认为他是个好老总。明确职责后，通过运用单一职责将带来一些优势：

- 有助于清晰的理清设计与编码的思路。
- 有助于简化维护、编码、测试的流程。
- 复杂问题简单化，有利于代码的重用。
- 职责之间消除耦合后，有利于系统的扩展。

3.3 如何实现单一职责

如何合理地使用单一职责原则，是一个“仁者见仁，智者见智”的问题。为了科学地实现单一职责，我们不妨从以下两个角度进行分析：

（1）从面向对象的原理方面进行分析。

一个类通俗来讲，它是按照对象的属性和运动规律的相似性，将相近的对象合并而成。

那么这个类，它只实现自身该做的事项，即我们可以认为它只是实现自己的职责。比如，我们系统架构设计人员，在架构设计中经常会运用分层的概念。其中每个层次的分类，各自实现各自的功能，这些功能我们可以理解成为所谓的职责。

（2）从业务功能的角度进行分析。

在我们应用系统的开发过程中，会涉及到各种各样的功能模块。比如常见的“增加”、“删除”、“修改”、“查询”功能，这些都属于操作职责，此时我们可以把它封装成一个类。不过，在设计类时需要考虑高内聚、低耦合，并且要注意类的大小范围要适中。

为何类的大小要适中呢？因为，如果一个类的范围过小，那么“增加”、“删除”、“修改”、“查询”可以分成四个类，结果必然会导致接口过多，影响开发进度；如果类的范围过大，一个类可以包括多个功能的职责，结果必然会导致这个类维护复杂。

3.4 应用反思——产品报表

某公司项目组为了锻炼新手，布置一个产品报表的模块让几个新人去设计开发。这个产品报表模块的属性包括生产年月、产品名称、产品类型、品牌、产品价格和产地，具体功能只需实现查询。此时，几个新人中，出现两种不同的实现方式。

方式一，把所有属性与功能封装在一个接口中，利用实现类去实现接口，如图 3.4 所示

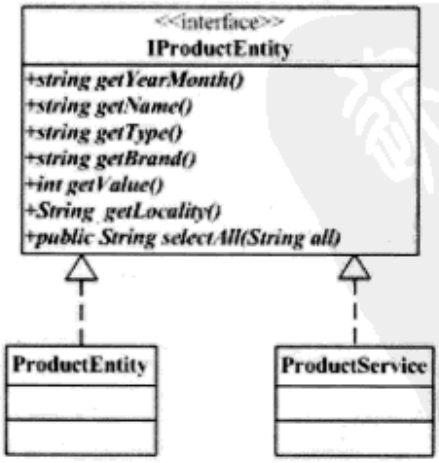

图 3.4

本应用反思的工程名为“程序 3.4.1”，源代码如下所示。

（1）IProductEntity 既包含产品的各个属性又包含查询功能。

```
package principle.srp1;

//单一职责原则应用 1
```

```
public interface IProductEntity {
    public String getYearMonth(String YearMonth);//生产年月
    public String getName(String Name );//产品名称
    public String getType(String Type);//产品类型
    public String getBrand(String Brand);//品牌
    public int getValue(int Value);//产品价格
    public String getLocality(String Locality);//  产地
    ////////////////////////////////////////////////
    public String selectAll(String all);//查询所有信息
}
```

（2）ProductEntity 类实现接口 IProductEntity。

```
package principle.srp1;

public class ProductEntity implements IProductEntity{

    public String getName(String Name) {
        String Name1="手机";
        System.out.println(Name+"功能性");
        return Name1;
    }

    public String getYearMonth(String YearMonth) {
        String YearMonth1="报表日期";
        System.out.println(YearMonth+"年月");
        return YearMonth1;
    }

    public String getBrand(String Brand ) {
        String Brand1="品牌";
        System.out.println(Brand1+"眼镜");
        return Brand1;
    }

    public String getLocality(String Locality) {
        String Locality1="产地";
        System.out.println(Locality1+"产品产地");
        return Locality1;
    }

    public String getType(String Type) {
        String Type1="产品类型";
        System.out.println(Type1+"产品产地");
        return Type1;
    }

    public int getValue(int Value) {
        int Value1=20;
```

```
        System.out.println(Value1+"价格");
        return Value1;
    }
    public String selectAll(String all) {
        String all1="全部查询";
        System.out.println(all1+"查询");
        return all1;
    }
}
```

（3）ProductService 类实现接口 IProductEntity。

```
package principle.srp1;

public class ProductService implements IProductEntity {
    public String selectAll(String all) {
        String all1 = "全部查询";
        System.out.println(all1 + "查询");
        return all1;
    }

    public String getBrand(String Brand) {
        return null;
    }

    public String getLocality(String Locality) {
        return null;
    }

    public String getName(String Name) {
        return null;
    }

    public String getType(String Type) {
        return null;
    }

    public int getValue(int Value) {
        return 0;
    }

    public String getYearMonth(String YearMonth) {
        return null;
    }
}
```

（4）srp1.jsp，调用未使用单一职责模式时展现的页面。

```
<%@ page contentType="text/html;charset=GB2312"%>
```

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312"
/>
        <title>单一职责原则</title>
    </head>
    <body>
        <%@ page pageEncoding="GB2312" language="java" import="principle.srp1.*"%>

        <%
        ProductEntity  productEntity=new ProductEntity();
        out.println("产品名称: 1.TCL"+productEntity.getName("TCL"));
        out.println("2.厦新"+productEntity.getName("厦新"));
        out.println("3.华为"+productEntity.getName("华为"));
        %><br />
    </body>
</html>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
产品名称: 1.TCL 手机 2.厦新手机 3.华为手机
```

方式二，把产品属性封装在一个接口，产品业务操作功能封装在另一个接口，如图 3.5 所示。

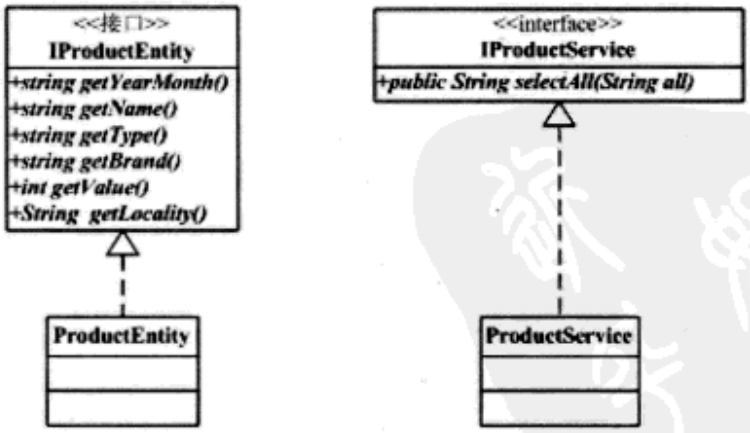

图 3.5

本应用反思的工程名为“程序 3.4.2”，源代码如下所示。

（1）IProductEntity 包含产品的各个属性。

```
package principle.srp2;
public interface IProductEntity {
    public String getYearMonth(String YearMonth);//生产年月
    public String getName(String Name );//产品名称
    public String getType(String Type);//产品类型
    public String getBrand(String Brand);//品牌
    public int getValue(int Value);//产品价格
```

```
    public String getLocality(String Locality);//  产地
}
```

（2）ProductEntity 类实现接口 IProductEntity。

```
package principle.srp2;

public class ProductEntity implements IProductEntity{

    public String getName(String Name) {
        String Name1="手机";
        System.out.println(Name+"功能性");
        return Name1;
    }

    public String getYearMonth(String YearMonth) {
        String YearMonth1="报表日期";
        System.out.println(YearMonth+"年月");
        return YearMonth1;
    }

    public String getBrand(String Brand ) {
        String Brand1="品牌";
        System.out.println(Brand1+"眼镜");
        return Brand1;
    }

    public String getLocality(String Locality) {
        String Locality1="产地";
        System.out.println(Locality1+"产品产地");
        return Locality1;
    }

    public String getType(String Type) {
        String Type1="产品类型";
        System.out.println(Type1+"产品产地");
        return Type1;
    }

    public int getValue(int Value) {
        int Value1=20;
        System.out.println(Value1+"价格");
        return Value1;
    }
}
```

（3）IProductService 包含产品的查询功能。

```
package principle.srp2;

public interface IProductService {
    public String selectAll(String all);//查询所有信息
}
```

（4）ProductService 类实现接口 IProductService。

```
package principle.srp2;

public class ProductService implements IProductService{
    public String selectAll(String all) {
        String all1="全部查询";
        System.out.println(all1+"查询");
        return all1;
    }
}
```

（5）srp2.jsp，调用使用单一职责模式时展现的页面。

```
<%@ page contentType="text/html;charset=GB2312"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
        <title>单一职责原则</title>
    </head>
    <body>
        <%@ page pageEncoding="GB2312" language="java" import="principle.srp2.*"%>

        <%
         ProductEntity pe = new ProductEntity();
         out.println("产品和查询功能:1.TCL 手机 "+ pe.getName("TCL 手机"));
         ProductService  ps=new  ProductService();
         out.println("2.查询 "+ ps.selectAll("查询"));
        %><br />
    </body>
</html>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
产品和查询功能：1.TCL 手机 手机 2.查询 全部查询
```

从设计原则的角度考虑，两种设计方式的比对分析如下：

- 方式一，违背了单一职责原则。因为单一职责原则告诉我们，一个类应该只有一个引起该类变化的原因，这样有利提高类的可维护性和可复用性。
- 方式二，接口与实现类一一对应，符合单一职责原则，是一个较好的实现方式。

3.5 单一职责原则与 Spring

在笔者负责山西某市的烟草物流绩效平台开发过程中，遇到了指标权重计算和显示问题。有两个开发人员认为，在系统后台编制包括计算和显示功能的同一个 Java 接口类，通过 list 将计算和显示传给前台客户端即可。此方式从单一职责原则角度分析并不合理，因为两个功能的职责差异很大。最终，我和系统架构师运用单一职责原则，将计算和显示功能设置为两个接口类。主要的类与配置如下：

（1）创建计算接口。

```
public interface Icount{
    void int count; //用于指标权重的计算
}
```

（2）创建显示接口。

```
public interface Ishow{//
    void List pendTaskList;// 显示列表
}
```

（3）创建实现计算、显示接口的类，运用组件使配置文件的数据传递给前台客户端代码。其中，计算接口类的 xml 配置如下：

```
<category title = "a">
<item name="8">
<implement>计算实现类之 spring 配置名</implement>
<url>相关 url</url>
<type>sms</type>
<isCalculate>指标权重计算</isCalculate>
</item>
```

显示接口类的 xml 配置与计算接口类同理。

配置文件根据项目的情况也会产生变化，我们需要合理的规划。当 xml 配置文件之 item 中的属性产生变化时，显示功能需要统一变化，而计算类则也许无须变化。因而，本节在 Spring 中采以职责分离（即分离配置文件的职责，实行单一职责原则）的方式。

第 4 章　里氏代换原则

4.1　何谓里氏替换原则

里氏替换原则（Liskov Substitution Principle，LSP）由 Barbara Liskov 于 1988 年提出，它的完整表达是："若对每个类型 S 的对象 o1，都存在一个类型 T 的对象 o2，使得在所有针对编写的程序 P 中，用 o1 替换 o2 后，程序 P 行为功能不变，则 S 是 T 的子类型。"[1]

其实从通俗的角度来讲，我们可以这样认为，"如果对每一个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2 存在，使得以 T1 定义的所有程序 P 在所有的对象 o1 都替换成 o2 时，程序 P 的行为也没有变化，那么类型 T2 是类型 T1 的子类型。" 简言之，"子类型必须能够替换掉它们的基类型"。[2]

比如人类，分为男人和女人。他们各自继承了人的特性，并替换了人的一部分特性，因而形成男人与女人，如图 4.1 所示。

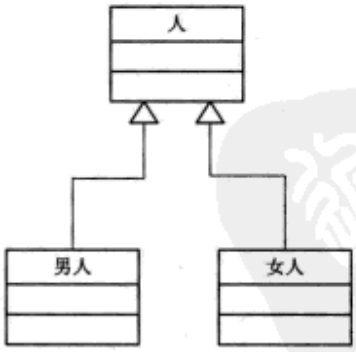

图 4.1

为了更好地理解此项原则，我们也可以从以下几点进行分析：

子类必须全部继承基类的功能。比如汽车，分为卡车、轿车、客车等。卡车、轿车、客车必须全部继承汽车的功能，否则车子开不起来。

充许子类进行个性化设置，比如卡车、轿车、客车各有其自身的特点。

覆写或实现基类方法时输入的参数可以自由设置，比如卡车载货的大小等。

子类中调用类，需要调用基类或接口。比如卡车、轿车、客车不调用其基类或接口则无法成为有用的车。

---

[1] Robert C. Martin. 敏捷软件开发：原则、模式与实践. 邓辉，译. 北京：清华大学出版社，2003，102
[2] Robert C. Martin. 敏捷软件开发：原则、模式与实践. 邓辉，译. 北京：清华大学出版社，2003，102

4.2 为何要实现里氏代换

目前，继承是面向对象思想中的一个重要的特点。它具有以下一些优势与问题：

（1）优势

它可以减少重复编码，从而实现代码的重用性。

子类与基类，可以相似，也可以有其各自的不同之处。

基类，可以提高代码开放性。

（2）问题

当父类方法与属性进行变动时，其子类也要随之修改。如果没有一个科学的规范，则其结果会导致大量的代码需要重构。

基于此，由于里氏代换原则以继承为基础，如果对其进行合理的规范化，其结果可以大大地降低代码的重构。因此，我们需要实现里氏代换。

4.3 如何实现里氏代换

我们在涉及类的继承设计时，可以从“父类不能替换子类，而子类可以替换父类”的思路引导下，进行里氏代换原则的实现。

首先，正确地进行继承设计。所有基类的方法都要在其子类中得到实现或者重写，并且子类不能写出与业务功能实现无关的多余方法或实现。

其次，最优的继承层次设计。当其他应用类调用业务功能类时，应该先调用其业务功能的基类而不应该直接调用业务功能的子类对象。

关于里氏代换原则的应用，在本书后面讲到策略模式（Strategy）、合成模式（Composite）、代理模式（Proxy）时，都会有充分的体现。

4.4 应用反思——子类调用父类

为了测试 Java 工程师对里氏替换原则的理解程度，我为某公司设计了一道题，其内容为：“假设有一个基类 SuperLiskov 和一个子类 SubLiskov,子类调用父类。请设计类图与编制代码。”

结果测试并不理想，大部分工程师们没有写对或只有写对一部分。

1. 错误的写法

其中最典型的错误表现为如图 4.2 所示之类的问题。

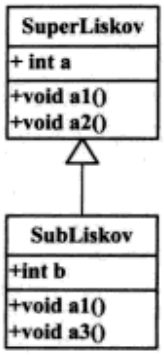

图 4.2

本应用情景的工程名为“程序 4.4.1”，源代码如下所示。

（1）基类：SuperLiskov

```
package principle.lsp1;

public class SuperLiskov {
    public int a;
    public int b;
    public void a1()
    { }
    public void a2()
    { }
}
```

（2）子类：SuperLiskov

```
package principle.lsp1;

public class SubLiskov extends SuperLiskov {
    public int b;
    public void a1() { }
    public void a3() { }
}
```

（3）测试类：lsp1.jsp

```
<%@ page contentType="text/html;charset=GB2312"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312"
/>
        <title>里氏替换原则</title>
```

```
    </head>
    <body>
        <%@ page pageEncoding="GB2312" language="java" import="principle.lsp1.*"%>

        <%
        SuperLiskov  ob=new  SuperLiskov(); //(1)
        ob=new SubLiskov();  //(2)
        ob.a=1;  //(3)
        ob.b=2;  //(4)
        ob.a1(); //(5)
        ob.a2(); //(6)
        ((SubLiskov) ob).a3(); //(7)

        %><br />
    </body>
</html>
```

在浏览器上打开页面，执行代码后不显示结果。原因在于：

程序中，子类改写了父类中的 a1()方法，添加了自己的属性 b 和方法 a3()。语句（1）中变量 ob 的静态类型是 SuperLiskov，动态类型是 SuperLiskov；语句（2）中变量 ob 的静态类型不变，动态类型是 SubLiskov（子类类型），子类对象的地址可以存放在父类类型的引用变量中，即父类引用变量引用子类对象；语句（3）、（5）、（6）是正确的，语句（4）、（7）是错误的，因为对于引用的合法性依赖于变量的静态类型，属性 b 和方法 a3()在 SuperLiskov 中不存在，故有错，还应该注意参数的匹配；语句（5）调用的 a1()是子类的 a1()，语句（6）调用的 a2()是父类的 a2()，这是因为对于消息的响应取决于变量的动态类型，因此 a1()是 SubLiskov 中的 a1()。

2. 正确的写法

这一道题类图的正确写法如图 4.3 所示。

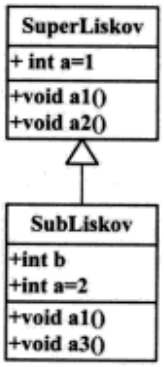

图 4.3

本应用情景的工程名为“程序 4.4.2”，源代码如下所示。

（1）基类：SuperLiskov

```
package principle.lsp2;

public class SuperLiskov {
    public int a=1;
    public void a1()
    {
    }
    public void a2()
    {
    }
    }
```

（2）子类

```
package principle.lsp2;

public class SubLiskov extends SuperLiskov{
    public int b;
    public int a=2;
    public void a1()
    { }
    public void a3()
    { }
    }
```

（3）测试类：lsp2.jsp

```
<%@ page contentType="text/html;charset=GB2312"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312"
/>
        <title>里氏替换原则</title>
    </head>
    <body>
        <%@ page pageEncoding="GB2312" language="java" import="principle.lsp2.*"%>

        <%
        SuperLiskov ob=new SuperLiskov(); //(1)
        ob=new SubLiskov();  //(2)
        ob.a1(); //(5)
        ob.a2(); //(6)
        out.println("结果为:" +  (ob.a));
```

```
        %><br />
    </body>
</html>
```

在浏览器上打开页面，执行代码，显示结果为 1。

因为程序中，子类改写了父类中的 a1()方法，添加了自己的属性 b 和方法 a3()。语句（1）中变量 ob 的静态类型是 SuperLiskov，动态类型是 SuperLiskov；语句（2）中变量 ob 的静态类型不变，动态类型是 SubLiskov（子类类型），子类对象的地址可以存放在父类类型的引用变量中，即父类引用变量引用子类对象；如果在父类和子类中存在属性的覆盖，则通过 ob（父类对象名）访问父类中被覆盖的属性。

4.5 里氏代换原则与 Struts 以及 Spring

1. 里氏代换原则与 struts

Struts 的 ActioMessagesStruts 具备消息传递功能，其核心元素如图 4.4 所示。

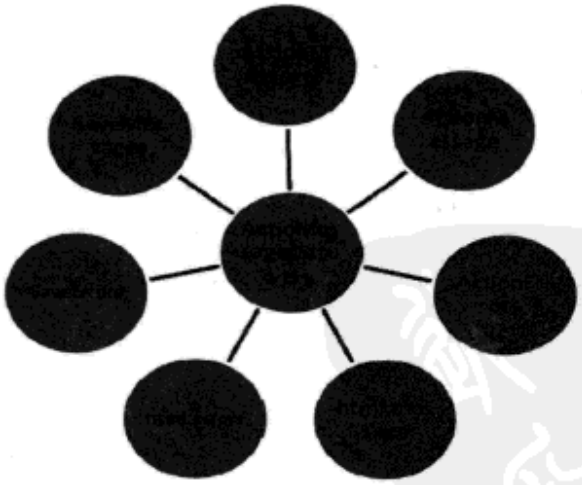

图 4.4

ActionMessage 的容器包括“ActionErrors 与 ActionMessages”，后者的实例均可相互设置为容器，并且 ActionErrors 也可运用 errors.add(msgs)使自己成为 ActionMessages（父类实例）的容器。此处，较好地运用了“里氏代换”原则的原理，即这种做法实现了在父类涉及的场所子类可安全的替代。

2. 里氏代换原则与 Spring

依据里氏代换原则的原理展开分析，如果可以接收到父类的场所，则能够接收到子类。以猫为例，白猫与黑猫都要睡觉与捕捉老鼠。

（1）建立猫接口（包括白猫 WhiteCat 与黑猫 BlackCat）。

```
package lsp;
public interface Cat {
    public void catchMouse();// 抓老鼠
    public void sleep();//睡觉
}
```

（2）建立猫实现类（包括白猫和黑猫）。

```
//白猫
package lsp;
public class WhiteCat  implements Cat {
    public void sleep() {
        System.out.println("白猫睡觉了!");
    }
    public void catchMouse() {
        System.out.println("大老鼠你跑不了");
    }
}
//黑猫
package lsp;
public class BlackCat implements Cat {
    public void sleep() {
        System.out.println("黑猫睡觉了!");
    }
    public void catchMouse() {
       System.out.println("抓老鼠了!");
    }
}
```

（3）创建 bean 映射配置文件，例如 jackbean.xml（此文件名可随意取，但在初使化时须指定给初使化程序）。

```
    <?xml version="1.0" encoding="gb2312"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.sprintframework.org/dtd/spring-beans.dtd">
<beans>
<bean id="whiteCat" class="lsp.WhiteCat"/>
<bean id="blackCat" class="lsp.BlackCat"/>
</beans>
```

（4）设置测试代码类。

```
    public void doIt(){
        /*获取 bean 映射配置文件，将配置文件与所用当前测试类放置于同一目录结构中*/
        ApplicationContext  ctx=new  FileSystemXmlApplicationContext(getClass().
getResource("jackbean.xml").toString());
        Cat cat=null;
```

```
        cat=(Cat)ctx.getBean("whiteCat");
        cat.catchMouse();
        cat.sleep();
        cat=(Cat)ctx.getBean("blackCat");
        cat.catchMouse();
        cat.sleep();
    }
```

经过 getBean 返回的对象（如 WhiteCat 或者 BlackCat），均可赋于接口 Cat；此时接口类调用其方法时，由于此时父类实际上接受了子类的对象，因而此时调用了指定子类的方法，即 Spring 实现了“Cat cat=new WhiteCat();”。当然，子类通过 Spring 工厂映射配置生成而无须显示。

因此，spring 实现了里氏代换原则，即可接受父类的地方，也可接受子类。

第 5 章　依赖倒换原则

5.1　何谓依赖倒换原则

依赖倒换原则（Dependence Inversion Principle，DIP）它有两种定义。

- 定义一："高层模块不应该依赖于低层模块，二者都应该依赖于抽象。抽象不应该依赖于细节，细节应当依赖于抽象。"[1]简言之，我们在设计系统时，需要运用抽象来分析，而不必一开始关注类的细节。
- 定义二："要针对接口编程，不要针对实现编程。"[2]简言之，我们运用依赖倒换原则时，可以通过接口与抽象类进行各种变量、参数、方法等的声明。并且，禁止实现类去做以上各种声明。

综上所述，我们可以把依赖倒换原则理解成现实生活中的一些情景，比如孕妇与龙凤胎。有孕妇，才能谈到生产龙凤胎。孕妇是抽象，龙凤胎(包括男婴、女婴)是实现，具体如图 5.1 所示。

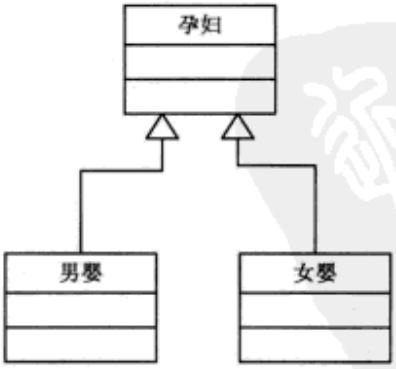

图 5.1

5.2　为何要实现依赖倒换

为何要遵循依赖倒换原则，我们可以从以下两点进行分析：

（1）倒转的缘由

当前，在我们软件项目开发过程中，为了使代码重用。我们经常会把一些通用的函数和功能

[1] Robert C. Martin. 敏捷软件开发：原则、模式与实践. 邓辉，译. 北京：清华大学出版社，2003，116
[2] 阎宏. Java 与模式. 北京：电子工业出版社，2002，87

写成通用模块，以方便有需要的开发人员调用。

如果我们的通用模块设计的不合理，则无法发挥其功能。比如，我们在类型转换功能类中封装 Oracle 格式的代码，而我们新项目使用的是 SQL Server。此时，调用该通用模块必然会出错。为了解决这一问题，我们必然要重构代码，此时依赖倒换则派上用场。我们通过在接口中定义 Oracle、SQL Server 等数据库的转换方法，在 SQL Server 具体实现类中实现接口方法即可。如此一来，无论将来新项目使用 Oracle 还是其他数据库，通用模块都无须修改或增加代码。

（2）依赖接口的作用

作为完整抽象描述的接口，其作用我们可以通过一个通俗的实例进行理解。比如当前居民用户家中，以插口方式进行网络式连接的电信 IPTV 电视盒。如果 IPTV 的机房网络连接进行了变换，此时他们只需在机房中进行变换，而我们的 IPTV 电视盒仍沿用原插口。如果不是通过插口来使用电视盒，而是通过直接与电视站进行线路连接；则在网络方式发生变化时，线路很有可能要进行改造，并且增加了安全风险。

由此可见，接口大大地提高了系统的扩展性和灵活性。

5.3 如何实现依赖倒换

如何使用依赖倒换原则，是一个值得让人深思的课题。我们不妨从以下几个方面去考虑：

- 从实现类与抽象类的角度进行分析。我们在运用具体实现类去继承抽象类时，需要保证引用‘基类’之处可修改成其子类。
- 从层次关系角度进行分析。需要定义清晰的层次关系，使每个层次通过接口的方式进行。
- 从对象构造角度进行分析。如果创建的是动态的对象，则使用依赖倒换。如果创建的是一些静态的具体类并且变化率很低，则我们无须再创建其基类去继承，以规避维护多余代码的风险。

综上所述，我们在使用依赖倒换原则时，需要在完全理解定义的基础上运用分层的概念，通过具体实现类去调用接口或抽象类，最终达到高效、清晰、系统扩展灵活的效果。

5.4 应用反思——Java 程序员招聘

某外贸小公司刚成立不久，为了开展 B2C 业务需要进行网站开发，公司老板委托我帮忙招收高级与普通 Java 程序员各一名。此时，我设计了一些题目，其中，以一个员工管理小模块作为设计考题。具体内容如下：

本公司人员包括管理层与业务员，现因业务需要招收两名 Java 程序员（请根据依赖倒换原则对公司人员管理进行设计，只需创建 Java 类图与简要代码）。

笔试的程序员中主要有两种回答：

1. 违反倒转依赖原则的回答

设计四个类，管理层、业务员、高级程序员和普通程序员各创建一个类图，如图 5.2 所示。

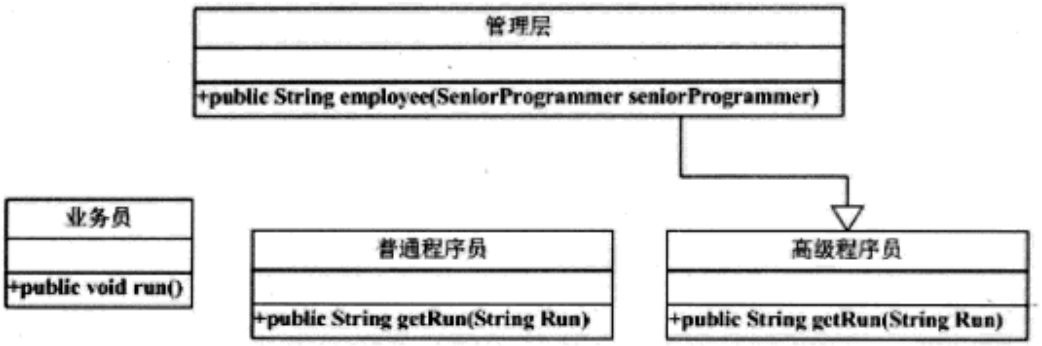

图 5.2

本应用情景的工程名为“程序 5.4.1”，源代码如下所示。

（1）管理层

```
package principle.dipl;

public class Employee { //管理层
    //管理层只给高级程序员李四工作，不给普通程序员工作
    private static final String Run = "李四";
    public String employee(SeniorProgrammer seniorProgrammer){
        String Run1="工作";
         seniorProgrammer.getRun(Run);
        return Run1;
    }
    }
```

（2）业务员

```
package principle.dipl;

public class Salesman { //业务员
    public void run() { //业务员工作
        System.out.println("业务人员开始干活...");
    }
}
```

（3）高级程序员

```
package principle.dipl;

public class SeniorProgrammer { //Programmer {//程序员
    public String getRun(String Run){
        String Run1="工作";
        System.out.println(Run+"高级程序员开始工作...");
          return Run1;
        }
    }
```

（4）普通程序员

```
package principle.dip1;

public class CommonProgrammer { //普通程序员
    public String getRun(String Run){
        String Run1="工作";
        System.out.println(Run+"普通程序员开始工作...");
          return Run1;
        }
    }
```

（5）测试类，dip1.jsp

```
<%@ page contentType="text/html;charset=GB2312"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312"
/>
        <title>依赖倒换原则</title>
    </head>
    <body>
        <%@ page pageEncoding="GB2312" language="java" import="principle.dip1.*"%>

        <%
    Employee zhangSan = new Employee();
    SeniorProgrammer lishi = new SeniorProgrammer();
    out.println("管理人员张三派高级程序员李四:"+zhangSan.employee(lishi) );
        %><br />
    </body>
</html>
```

在浏览器上打开页面，执行代码，显示结果为 1。

```
管理人员张三派高级程序员李四:工作
```

2. 符合倒转依赖的回答（如图 5.3 所示）

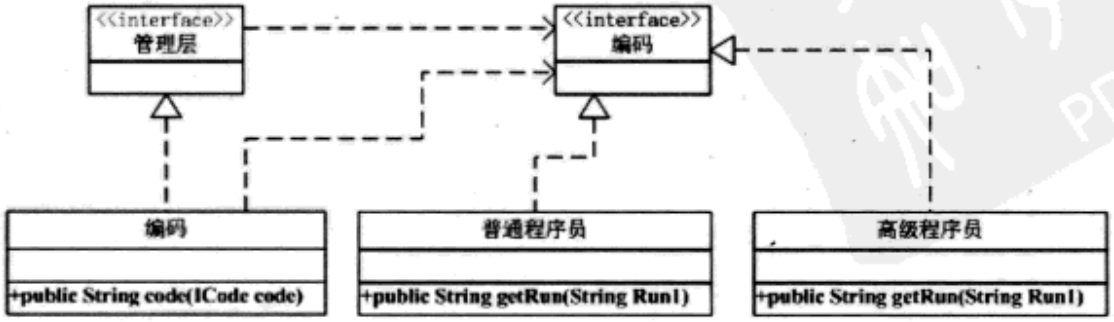

图 5.3

定义管理层接口、编码接口、编码类、普通程序员类和高级程序员类。

本应用反思的工程名为“程序 5.4.2”，源代码如下所示。

（1）管理层接口

```
package principle.dip2;

public interface IEmployee {   //管理人员让程序员编码
    public String code(ICode code);
    }
```

（2）编码接口

```
package principle.dip2;

public interface ICode { //编码
    public String getRun(String Run);
    }
```

（3）编码类

```
package principle.dip2;

public class Code implements IEmployee{ //程序员编码
    private static final String Run = "编码活动";
    public String code(ICode code){
        System.out.println(code+"程序员开始编码...");
         return code.getRun(Run);
    }
    }
```

（4）普通程序员类

```
package principle.dip2;

public class CommonProgrammer implements ICode{ //普通程序员
    private static final String Run = "JACK";
    public String getRun(String Run1){
        String Run2="工作";
        System.out.println(Run+"普通程序员开始工作...");
          return Run2;
        }
    }
```

（5）高级程序员类

```
package principle.dip2;
```

```
public class SeniorProgrammer implements ICode{ //高级程序员
    private static final String Run = "李四";
    public String getRun(String Run1) {
        String Run2="工作";
        System.out.println(Run+"高级程序员开始工作...");
          return Run2;
    }
    }
```

（6）测试类，dip2.jsp

```
<%@ page contentType="text/html;charset=GB2312"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312"
/>
        <title>单一职责原则</title>
    </head>
    <body>
        <%@ page pageEncoding="GB2312" language="java" import="principle.dip2.*"%>

        <%
        IEmployee employee = new Code();
        ICode code = new SeniorProgrammer();
        ICode code1 = new CommonProgrammer();
        //高级程序员张三开始编码
        out.println(""+employee.code(code));
        //普通程序员李四开始编码
        out.println("管理人员张三派高级程序员李四:"+employee.code(code1) );
        %><br />
    </body>
</html>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
工作管理人员张三派高级程序员李四：工作
```

5.5 依赖倒换原则在 Spring 中的应用

由于 Spring 具备 IoC 的功能，因而当软件设计开发人员在调用某一 Java 类时无须了解具体的实现，只需调用某接口之相关方法即可。下面我们来看一个通过 WebService 进行查询得到结果的具体实例，如图 5.4 所示。

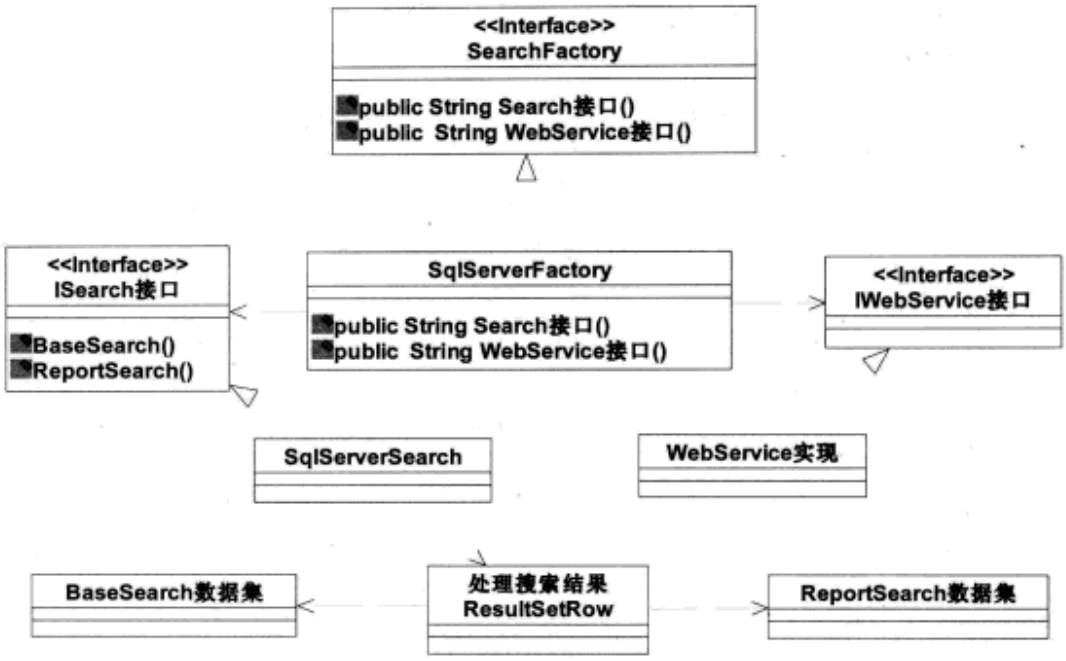

图 5.4

上图中，SearchFactory 作为搜索工厂接口，ISearch 接口是搜索接口，软件设计开发人员在访问此类方法时，运用 Spring 的 context 文件进行配置，便可使具体程序编码无须了解具体实现类即可进行调用。

第 6 章　接口隔离原则

6.1　何谓接口隔离原则

接口隔离原则简称 ISP（ISP--Interface Segregation Principle）。它主要有以下两个定义：

- 定义一："不应该强迫客户依赖于它们不用的方法。"[1]
- 定义二："一个类对另一个类的依赖性应当是建立在最小的接口上。"[2]

细品以上两个定义，我个人认为它们体现的意思基本一致。它们都是在说定义接口的表达要准确，不要创建多余的方法。基于此，在现实生活中不难发现接口隔离的案例。比如，将我们的手机构建成一个接口，如图 6.1 所示。

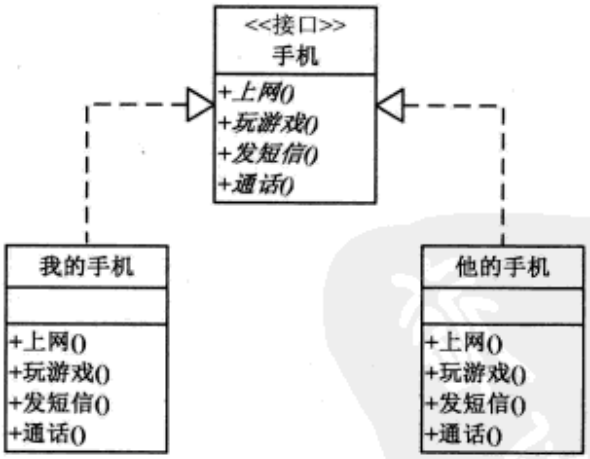

图 6.1

此时，我的手机和他的手机要调用手机接口，则会显示接口的所有方法。但是，我的手机与他的手机如果用处不同，则需要将接口重新规划。比如，我的手机用于上网和玩游戏，他的手机用于发短信和通话，则需要规划成两个接口，如图 6.2 所示。

[1] Robert C. Martin. 敏捷软件开发：原则、模式与实践. 邓辉，译. 北京：清华大学出版社，2003，125
[2] 闻宏. Java 与模式. 北京：电子工业出版社，2002，97

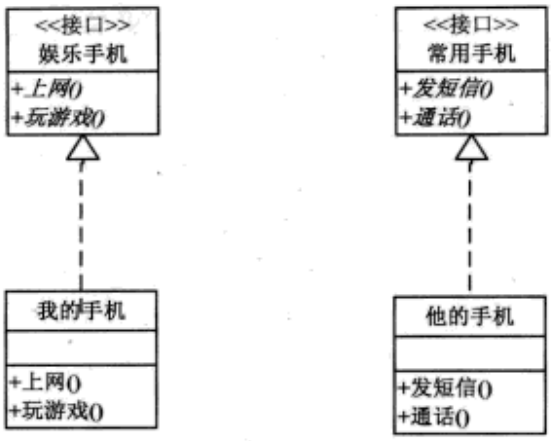

图 6.2

6.2 为何要实现接口隔离

ISP 理解起来很简单，我们可以把接口理解成角色，但是在实际应用开发中却经常有人违反此原则。比如以下两点：

- 我们经常会发现有一些公司的 Java 代码中有一个大接口，里面放着一大堆方法，其中有些方法根本就没有作用。其结果必然导致开发人员将不需要实现的方法多次放置在接口中，造成一定程度的代码冗余。
- 我们在系统开发时，如果有一个职责改变了，那么我们就去修改这个接口？这个接口有多少个实现类，我们就要去修改多少个类。如果我们运用接口隔离原则，一开始就设计角色独立的接口，这种情况就不会出现。

基于此，为了解决此类问题，实现接口隔离原则是一个较好的方法。

6.3 如何实现接口隔离

如何使用接口隔离原则，是一个值得让人深思的课题。我们不妨从以下几个方面去考虑：

首先，从业务逻辑角度考虑接口，我们可以把某类功能也设计成接口。比如，在各类影片中主演正角、反角的演员，“正角”、“反角”是接口，“主演正角”、“反角的演员”则是接口的实现。

其次，根据场合和调用者的情况，消除无关的方法，只提供同类型角色的接口。

再次，我们对客户程序进行有效区分，并对其对应的接口进行变化。比如，当客户程序又乱又杂，此时我们就需要对其进行分离。随着客户程序的分离，其对应的接口也随之变化。

如此一来，分离的客户程序与其接口则是同一角色对应的接口隔离模式。

6.4 应用反思——商品管理功能设计

某化妆品公司需要开发一个 B2C 网站，其中有一个商品管理的功能要求如下：

- 门户网站，只需查询方法；
- 管理系统后台，需要“增加、修改、查询、删除”功能。

此时，设计师该如何运用接口隔离原则比较合理呢？

1. 不合适的设计（如图 6.3 所示）

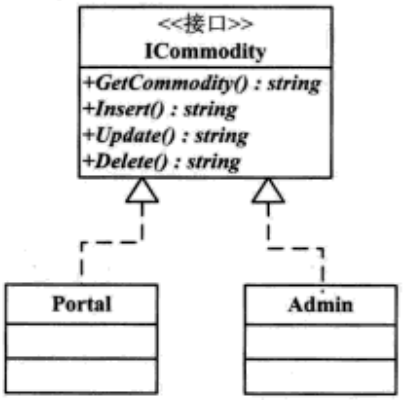

图 6.3

本应用情景的工程名为“程序 6.4.1”，源代码如下所示。

（1）ICommodity 包含商品“增加”、“修改”、“查询”、“删除”的各个功能。

```
package principle.isp1;

public interface ICommodity {
    public String GetCommodity(String Commodity);//method1 查询
    public String Insert(String Insert);//method2 插入
    public String Update(String Update);//method3 修改
    public String Delete(String Delete);//method4 删除
}
```

（2）Portal 类实现接口 ICommodity。

```
package principle.isp1;

public class Portal implements ICommodity {
    public String GetCommodity(String Commodity) {
        String a = "xx";
        System.out.println("类 Portal 实现接口 ICommodity 的查询方法");
        return a;
    }
```

```
        //对于类 Portal 来说，Insert\Update\Delete 不是必需的，但是由于接口 ICommodity
中有这 3 个方法，
        //所以在实现过程中即使这 3 个方法的方法体为空，也要将这 3 个没有作用的方法进行实现。
        public String Insert(String Insert) {
            String b = "不需要";
            System.out.println("类 Portal 不需要插入方法");
            return b;
        }
        public String Update(String Update) {
            String c = "null";
            System.out.println("类 Portal 不需要修改方法");
            return c;
        }
        public String Delete(String Delete) {
            String d="null";
            System.out.println("类 Portal 不需要 Delete 方法");
            return d;
        }
    }
```

（3）Admin 类实现接口 ICommodity。

```
package principle.ispl;

public class Admin implements ICommodity {
    public String GetCommodity(String Commodity) {
        String a = "a1";
        System.out.println("类 Admin 实现接口 ICommodity 的查询方法");
        return a;
    }

    public String Insert(String Insert) {
        String b = "需要";
        System.out.println("类 Admin 实现接口 ICommodity 的插入方法");
        return b;
    }

    public String Update(String Update) {
        String c = "c1";
        System.out.println("类 Admin 实现接口 ICommodity 的修改方法");
        return c;
    }

    public String Delete(String Delete) {
        String d = "Delete";
        System.out.println("类 Admin 实现接口 ICommodity 的 Delete 方法");
        return d;
    }
```

```
}

isp1.jsp，测试未使用接口隔离原则时的情况

<%@ page contentType="text/html;charset=GB2312"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312"
/>
        <title>未使用接口隔离原则</title>
    </head>
    <body>

        <%@ page pageEncoding="GB2312" language="java" import="principle.isp1.*"%>
        <%
        Portal b=new Portal();
        out.println("类 Portal 的 Insert 方法: "+b.Insert(""));
        %>
        <br>
        <%
        Admin d=new Admin();
        out.println("类 Admin 的 Insert 方法: " +d.Insert(""));
        %><br />
    </body>
</html>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
类 Portal 的 Insert 方法: 不需要类 Admin 的 Insert 方法: 需要
```

2. 合适的设计，如图 6.4 所示

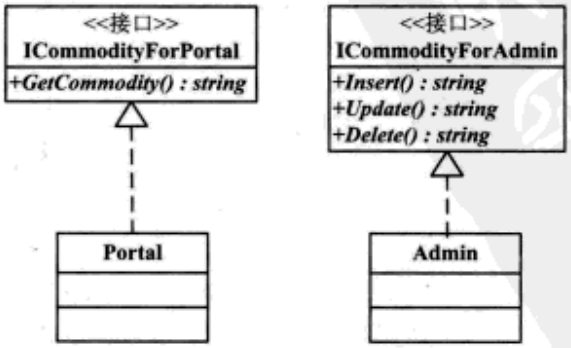

图 6.4

本应用反思的工程名为“程序 6.4.2”，源代码如下所示。

（1）ICommodityForPortal 包含商品“查询”的各个功能。

```
package principle.isp2;
public interface ICommodityForPortal {
```

```
    public String GetCommodity(String Commodity);//查询
  }
```

（2）ICommodityForAdmin 包含商品“增加”、“修改”、“删除”的各个功能。

```
package principle.isp2;

public interface ICommodityForAdmin {
    public String GetCommodity(String Commodity);//method1 查询
    public String Insert(String Insert);//method2 插入
    public String Update(String Update);//method3 修改
    public String Delete(String Delete);//method4 删除
  }
```

（3）Portal 类实现接口 ICommodityForPortal。

```
package principle.isp2;

 public class Portal implements ICommodityForPortal{
        public String GetCommodity(String Commodity) {
            String a = "查询";
            System.out.println("类 Portal 实现接口 ICommodity 的查询方法");
            return a;
        }
    }
```

（4）Admin 类实现接口 ICommodityForAdmin。

```
package principle.isp2;

public class Admin implements  ICommodityForAdmin{
    public String GetCommodity(String Commodity) {
        String a = "查询";
        System.out.println("类 Admin 实现接口 ICommodity 的查询方法");
        return a;
    }
    public String Insert(String Insert) {
        String b = "插入";
        System.out.println("类 Admin 实现接口 ICommodity 的插入方法");
        return b;
    }
    public String Update(String Update) {
        String c = "修改";
        System.out.println("类 Admin 实现接口 ICommodity 的修改方法");
        return c;
    }
    public String Delete(String Delete) {
```

```
        String d = "删除 ";
        System.out.println("类 Admin 实现接口 ICommodity 的删除方法");
        return d;
    }
    }
```

（5）isp2.jsp，测试使用接口隔离原则时的情况。

```
<%@ page contentType="text/html;charset=GB2312"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312"
/>
        <title>使用接口隔离原则</title>
    </head>
    <body>

        <%@ page pageEncoding="GB2312" language="java" import="principle.isp2.*"%>

        <%
        Portal b=new Portal();
        out.println("类 Portal 的方法: "+b.GetCommodity(""));
        %>
        <br>
        <%
        Admin d=new Admin();
        out.println("类 Admin 的方法: " +d.GetCommodity(""));
        out.println(", " +d.Insert(""));
        out.println(", " +d.Update(""));
        out.println(", " +d.Delete(""));
        %><br />
    </body>
</html>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
类 Portal 的方法: 查询类 Admin 的方法: 查询、插入、修改、删除
```

3. 比对分析：

从设计原则的角度考虑，上面两种设计方式的对比如下：

- 方式一，违背了接口隔离原则。因为单一职责原则告诉我们，类间的依赖关系应该建立在最小的接口上。这样有利提高类的可维护性和可复用性。
- 方式二，接口与实现类角色一一对应，符合接口隔离原则，是一个较好的实现方式。

6.5 接口隔离原则在 Spring 中的应用

在 Spring 框架中 org.springframework.beans.factory.BeanFactory 是 IoC 容器的核心组成部分，大家在调用 BeanFactory 接口及主要的相关接口时，就可以发现它们体现了接口隔离原则的相关原理。如图 6.5 所示。

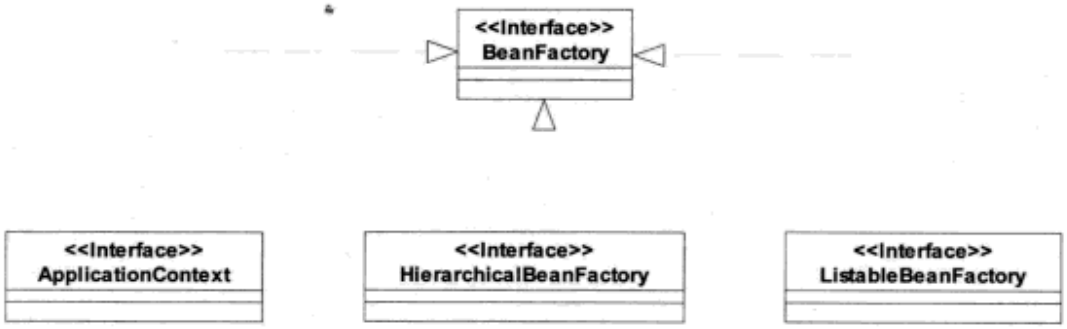

图 6.5

BeanFactory 提供管理多个对象的各个方法都很简洁，继承者只要使用他们所需要的内容即可。

ApplicationContext 实现了 BeanFactory 的相关方法，并可将来自 BeanFactory 的功能进行进一步的提升。

HierarchicalBeanFactory 接口在实现 BeanFactory 的基础上，将 BeanFactory 可以集成父容器的功能进行了具体的落实，并且可以方便其他程序有效地创建工厂链。

ListableBeanFactory 接口主要实现了 BeanFactory 中能够列表的相关 Bean，我们可以将它理解为 IoC 容器的门面之一。

第 7 章　迪米特法则

7.1　何谓迪米特法则

“迪米特法则（Law of Demeter）又叫做最少知识原则（Least Knowledge Principle，LKP），它是指一个对象应当对其他对象有尽可能少的了解，不必与不相识的人直接联系”。[1]

这个概念是由美国 Northeastern University 的 Ian Holland 于 1987 年秋天提出的，后由 UML 的创始者之一 Booch 等普及。

为了切实地理解此原则，我们可以从一个租房的例子去分析。比如租客需要租借三室二厅的住房，他不必直接到房东家去租，而是可以通过一个房产中介处去了解情况，这样就减少了租房者与住房之间的耦合，如图 7.1 所示。

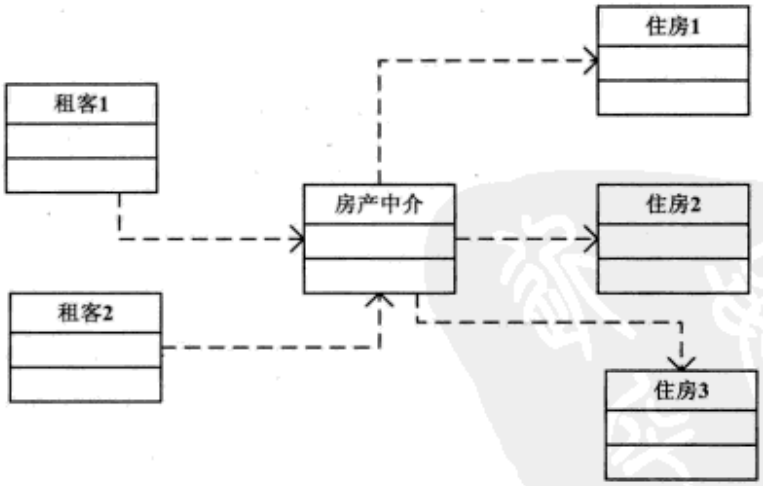

图 7.1

此外，迪米特法则主要分为两大类。

（1）第一类：狭义的迪米特法则

“如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的方法的话，可以通过第三者转发这个调用。”[2]

比如在谈生意时，为防止被对方所骗，甲、乙双方可通过双方熟识的第三方进行联系。

---

[1] 闾宏. Java 与模式. 北京：电子工业出版社，2002，109
[2] 闾宏. Java 与模式. 北京：电子工业出版社，2002，109

（2）第二类：广义的迪米特法则

“一个模块设计得好坏，一个重要标志就是该模块在多大程度上将自己的内部数据与实现的有关细节隐藏起来。”[1]

比如，我们可以从古代代表皇帝出巡的钦差大人与老百姓之间的关系进行分析。

钦差大人是皇帝为了解民间疾苦而设的，其过程为老百姓直接向钦差反映生活问题，皇帝再，通过钦差的报告了解各地百姓的民间百态，以便能更好地治理国家。

7.2 为何要实现迪米特

当今社会，各行各业都有一定的行业规则。比如，一个全职雇员只能与一家单位签订固定劳动合同。如果签订多个劳动合同，则会对雇员的社保与公积金等方面造成重大影响。

明确各个对象的关系后，通过运用迪米特法则将带来一些优势：

- 迪米特法则有利于降低类之间的耦合度。由于类与类之间去除了依赖关系，则各个软件功能模块之间相互独立。
- 遵循迪米特法则进行系统设计时，如果系统需要扩展，则更加符合开闭原则对修改关闭的要求。
- 使用迪米特法则将系统的内部数据与实现细节隐藏，从而使各个功能子模块远离耦合，最终达到提高代码的重用性和可维护性。

7.3 如何实现迪米特

迪米特法则的主要目标在于控制信息的加载，因此我们在系统设计的过程中要关注以下一些问题：

- 设计者对类进行分类设计时，需要创建低耦合的类层次关系，使类之间的耦合度越来越低，从而达到高度重用的效果。
- 设计者对类进行构造时，每个类之间都需要降低成员之间的访问程度。特别是实体类，我们需要尽量降低它的访问权限。我们可以开放取值和赋值的方法让外界间接访问自己的属性，但是我们如果把实体类定义成 public，那么客户程序就可能会调用这个类。此时，实体类被删除，则会导致一些客户程序出错。
- 尽量把一些类设计成不变的类，以方便功能的实现。
- 注意与依赖倒换原则相结合。因为，过度使用迪米特法则会产生大量的中间类，这将导致系统维护变得复杂。此时，可使用依赖倒换原则来解决，通过在调用方和被调用方之间增加一个抽象层，被调用方在遵循抽象层的前提下就可以自由变化，此时抽象层成了调用方的“朋友”。

[1] 阎宏. Java 与模式. 北京：电子工业出版社，2002，116

7.4 应用反思——地下党单线联系

在解放前的白色恐怖时代，中共地下党员为了提高保密程度，实行单线联络方式。某男性党员只需与联络人联系，而联络人则与某女性党员直接联系。针对这一情况，我们运用迪米特法则如何设计会比较合理呢？

1. 纯迪米特法则设计，如图 7.2 所示

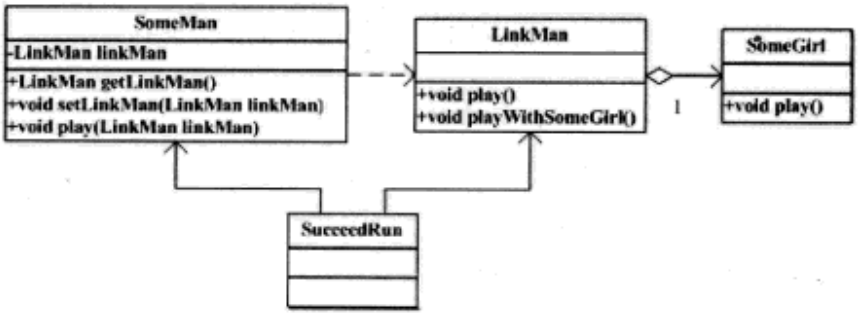

图 7.2

本应用反思的工程名为“程序 7.4.1”，源代码如下所示。

（1）某男性党员

```
package principle.lkp1;

public class SomeMan {
    private LinkMan linkMan;
        public LinkMan getLinkMan()
        {
            return linkMan;
        }
        public void setLinkMan(LinkMan linkMan)
        {
            this.linkMan = linkMan;
        }
        public void play(LinkMan linkMan)
        {
        System.out.println("someman ok");
        linkMan.play();
        }
}
```

（2）联络人

```
package principle.lkp1;

public class LinkMan {    //联络人
```

```
    public void play()
    {
        System.out.println("introducer ok");
    }
    public void playWithSomeGirl()
    {
        SomeGirl someGirl = new SomeGirl();
        someGirl.play();
    }
}
```

（3）某女性党员

```
package principle.lkp1;

public class SomeGirl {   ////某女
    public String play(){
        return "someGirl play";
    }
}
```

(4)成功完成任务

```
package principle.lkp1;

public class SucceedRun {   //成功完成任务
    public static void main(String[] args)
    {
        SomeMan jack = new SomeMan();
        jack.setLinkMan(new LinkMan());
        jack.getLinkMan().playWithSomeGirl();
    }
}
```

lkp1.jsp，测试未使用迪米特法则时的情况

```
<%@ page contentType="text/html;charset=GB2312"%>
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>迪米特法则</title>
</head>

<body>
<%@ page pageEncoding="GB2312" language="java" import="principle.lkp1.*" %>

<%
  SomeGirl x=new SomeGirl();
  out.println(  x.play() + ' ' );
```

```
%><br />

</body>
</html>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
someGirl play
```

2. 与依赖倒换相结合的方法，如图 7.3 所示

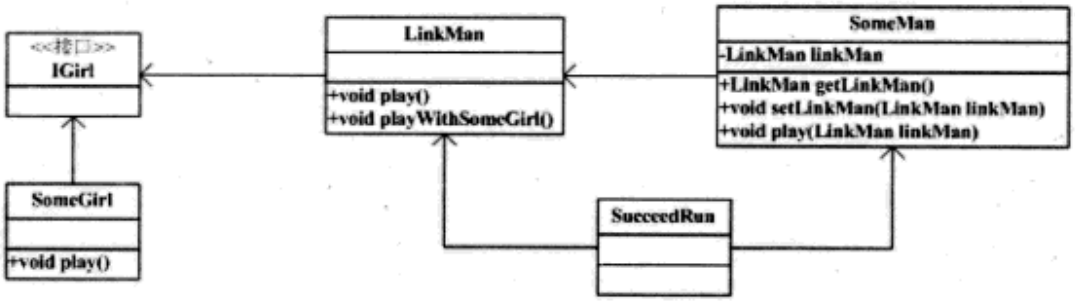

图 7.3

本应用反思的工程名为“程序 7.4.2”，源代码如下所示。

（1）某男性党员

```
package principle.lkp2;

public class SomeMan {   //某男性党员
    private LinkMan linkMan;
        public LinkMan getLinkMan()
        {
            return linkMan;
        }
        public void setLinkMan(LinkMan linkMan)
        {
            this.linkMan = linkMan;
        }
        public void play(LinkMan linkMan)
        {
        System.out.println("someman ok");
        linkMan.play();
        }
}
```

（2）联络人

```
package principle.lkp2;

public class LinkMan {   //联络人
```

```
    public void play()
    {
        System.out.println("introducer ok");
    }
    public String playWithSomeGirl()
    {
        SomeGirl someGirl = new SomeGirl();
        String a="aha";
         return someGirl.play(a);
    }
}
```

（3）抽象女党员接口

```
package principle.lkp2;

public interface IGirl {//抽象女党员接口
    public abstract String play(String a);
}
```

（4）某女性党员

```
package principle.lkp2;

public class SomeGirl implements IGirl {   ////某女性党员
    public String play(String a){
     String a1="someGirl play";
        System.out.println("someGirl play");
         return a1;
    }
}
```

（5）成功完成任务

```
package principle.lkp2;

public class SucceedRun {   //完成任务
    public static void main(String[] args)
    {
        SomeMan jack = new SomeMan();
        jack.setLinkMan(new LinkMan());
        jack.getLinkMan().playWithSomeGirl();
    }
}

lkp2.jsp，测试使用迪米特法则时的情况
<%@ page contentType="text/html;charset=GB2312"%>
```

```
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>迪米特法则</title>
</head>

<body>
<%@ page pageEncoding="GB2312" language="java" import="principle.lkp2.*" %>

<%
   SomeMan jack = new SomeMan();
   jack.setLinkMan(new LinkMan());
   out.println(jack.getLinkMan().playWithSomeGirl());
%><br />

</body>
</html>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
someGirl play
```

7.5 迪米特法则在 Spring 中的应用

IoC 作为 Spring 框架的核心部分，其名称叫控制反转，即将依赖关系进行了转移。

从技术结构角度而言，IoC 无须使用程序编码就可对各类程序之间的关系进行控制，并且是通过容器即可进行处理。需要注意，此处的 IoC 容器能够提供给组件需要运行的相关环境，并可对组件的有关声明周期进行维护与指导。

例如，组件 X 依赖另一组件 Y，则 Y 拥有控制权，但是应用程序无须了解依赖的具体实现，只需创建某种服务的相关对象进行松散耦合。

第三部分

设计创建派
——细说创建型模式

第 8 章　FactoryMethod（工厂方法）模式

8.1　概述

工厂方法模式又叫虚拟构造（Virtual Constructor）模式或者多态工厂（Polymorphic Factory）模式。它的正式定义为："工厂方法模式定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。"[1]

为了较好地理解此模式，我们可以从一个服装厂生产产品的例子去分析。比如一个服装厂由成人服装厂和未成人服装厂组成，其产品包括上装、裤子和连衣裙，如图 8.1 所示。

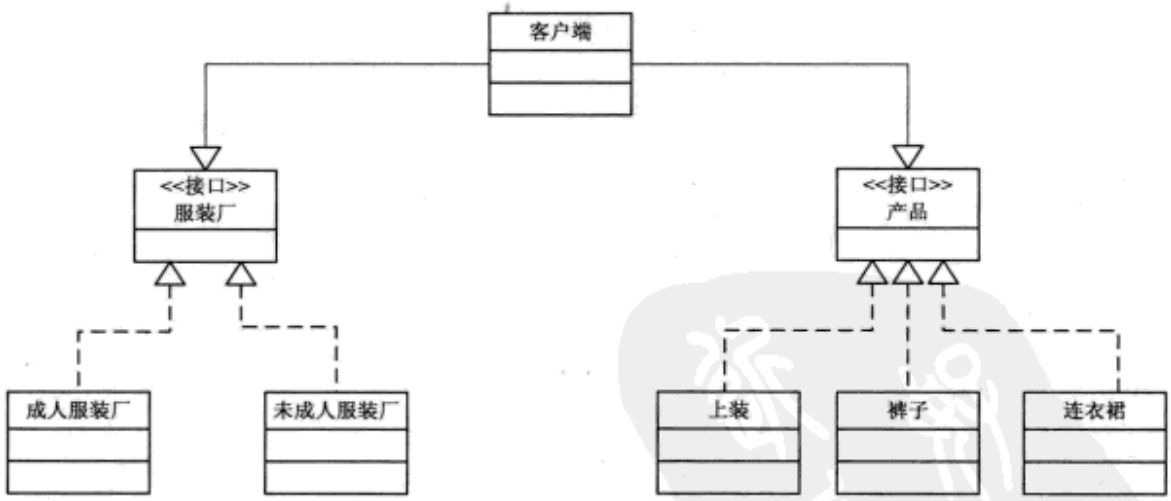

图 8.1

根据，目前业界公认的看法。工厂方法模式结构，由"抽象工厂、具体工厂、抽象产品、具体产品"角色组成。

"抽象工厂是工厂方法模式的核心，它是与应用程序无关的。任何在模式中创建对象的工厂类必须实现这个接口。在实际的系统中，这个角色也常常使用抽象 Java 类实现。

具体工厂是实现了抽象工厂接口的具体 Java 类。

抽象产品是工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口。

具体产品是实现了抽象产品角色所声明的接口。"[2]

---

[1] Eric Freeman. Head First 设计模式. 中文版. Oreily Taiwan 公司，译. 北京：中国电力出版社，2007，134
[2] 阎宏. Java 与模式. 北京：电子工业出版社，2002，153~154

图 8.1 中，服装厂是抽象工厂，成人服装厂、未成人服装厂是具体工厂。产品是抽象产品，上装、裤子、连衣裙是具体产品。

8.2 应用优势与时机

工厂方法具备以下一些优势：

- 在软件开发过程中，工厂方法通过使用产品的接口去实现功能，并且其实现的任务可延迟到子类中完成。
- 添加新产品无须修改接口。
- 隔断了客户端与具体产品的依赖关系，提高了产品的可扩展性。

基于此，我认为可以在以下几种情形下采用工厂方法模式进行软件设计与实施。

- 当客户端程序不了解调用几个类进行实例化时，我们可以运用工厂方法模式来创建一个接口，用于控制对一些类进行实例化。
- 当“父类”考虑使用其子类去指定创建的对象时。
- 当一个类不清楚它需要建立的对象类时，可通过其子类来实现。

8.3 应用情境——小明评先进

在某机械配件厂新来一个叫小明的青年技术工人。由于他既在生产上提出了创新的方法，又在工作上任劳任怨。因此，他被大家两次评选为先进个人。针对这一情况，本例运用工厂方法模式进行设计，如图 8.2 所示。

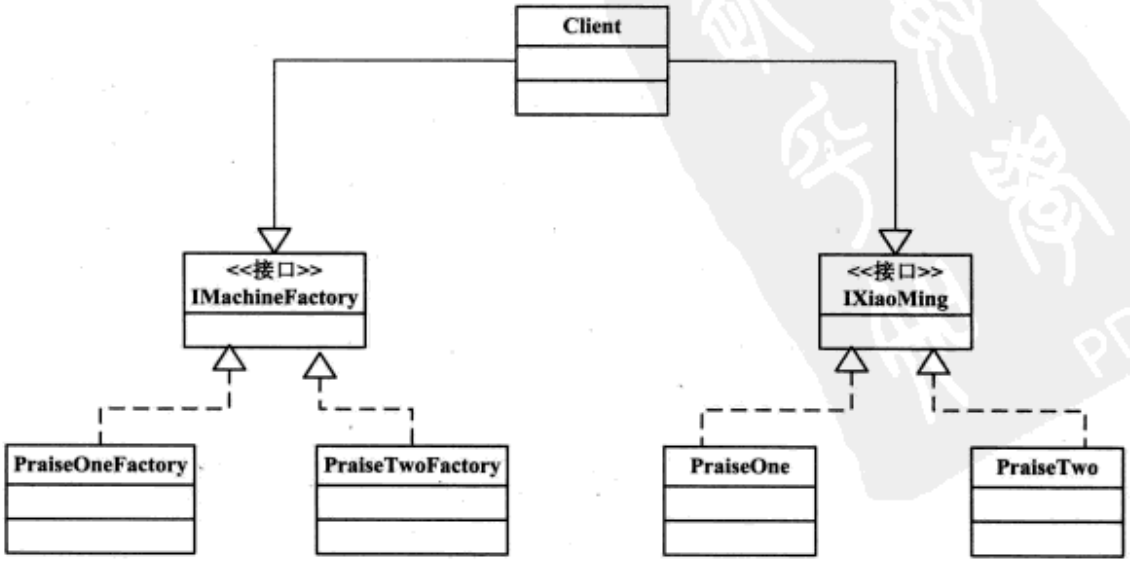

图 8.2

本应用情景的工程名为“程序 8.3.1”，源代码如下所示。

（1）机械厂

```
package model.factorymethod;

public interface IMachineFactory
{
        IXiaoMing CreateXiaoMing();
}
```

（2）机械厂表扬 1

```
package model.factorymethod;

public class PraiseOneFactory implements IMachineFactory{
    public IXiaoMing CreateXiaoMing() {
        return new PraiseOne();
    }
}
```

（3）机械厂表扬 2

```
package model.factorymethod;

public class PraiseTwoFactory implements IMachineFactory {
    public IXiaoMing CreateXiaoMing() {
        return new PraiseTwo();
    }
}
```

（4）小明

```
package model.factorymethod;

public interface IXiaoMing {
        public void SkillOne();
        public void SkillTwo();
        public void Industry();
}
```

（5）表扬 1

```
package model.factorymethod;

public class PraiseOne implements IXiaoMing
{
```

```
        public void SkillOne(){
            System.out.println("提高了工作效率");
        }
         public void SkillTwo(){
                System.out.println("降低了原料成本");
        }
          public void Industry(){
                 System.out.println("肯吃苦，勤劳");
        }
    }
```

（6）表扬 2

```
package model.factorymethod;

public class PraiseTwo implements IXiaoMing
{
        public void SkillOne(){
            System.out.println("提高了工作效率");
        }
         public void SkillTwo(){
                System.out.println("降低了原料成本");
        }
          public void Industry(){
                 System.out.println("肯吃苦，勤劳");
        }
}
```

（7）客户端 test.jsp

```
<HTML>
<!--
    工厂方法模式实现时，客户端需要决定实例化那个工厂来实现运算类。因此，存在需要选择判断的问题。
    然而，工厂方法把简单工厂的内部逻辑判断移到了客户端代码来进行。
    开发人员想要添加的功能，本来是修改工厂类，而现在是修改客户端。
    @author jianghc
-->

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    工厂方法模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.factorymethod.*"
%>
```

```
    <jsp:useBean id="IF" scope="session" class="model.factorymethod.
IndustryFactory"/>
    <jsp:useBean id="BF" scope="session" class="model.factorymethod.
BecilityFactory"/>

    <br />
    <br />
    <h4>
    <%
        XiaoMing becility=BF.CreateXiaoMing();
        out.println("Becility");
    %>
    </h4>
    <% out.println("SkillOne: " + becility.SkillOne()); %><br />
    <% out.println("SkillTwo: " + becility.SkillTwo()); %><br />
    <% out.println("Industry: " + becility.Industry()); %>
    <br />
    <br />
    <br />

    <h4>
    <%
        XiaoMing Industry=IF.CreateXiaoMing();
        out.println("Industry");
    %>
    </h4>
    <% out.println("SkillOne: " + Industry.SkillOne()); %><br />
    <% out.println("SkillTwo: " + Industry.SkillTwo()); %><br />
    <% out.println("Industry: " + Industry.Industry()); %>
    </BODY>
    </HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
Becility
SkillOne: 提高了工作效率
SkillTwo: 降低了原料成本
Industry: 肯吃苦，勤劳

Industry
SkillOne: 提高了工作效率
SkillTwo: 降低了原料成本
Industry: 肯吃苦，勤劳
```

8.4 工厂方法与开闭原则

工厂方法模式，通过优化简单工厂模式，实现“开闭”原则。

8.5 工厂方法模式与简单工厂

目前业界的看法，“简单工厂模式就是由一个工厂类根据传入的参量决定创建出哪一种产品类的实例。”[1]因此，我们在使用简单工厂模式时，只需传递参数给简单工厂即可。

由于简单工厂比较简单，此处不再细述。简单工厂和工厂方法的相同与区别之处如表 8.1 所示。

表 8.1 简单工厂和工厂方法的相同与区别之处

模式名	简单工厂	工厂方法
是否创建型模式	是	是
是否把对象的实例化部分抽取出来	是	是
是否增强系统扩展性	是	是
核心	一个具体类	一个抽象工厂类
新增产品	如果系统增加新的产品，就必须修改工厂类	如果系统增加新的产品，只需要实现新的对应工厂即可。
是否符合开闭原则	否	是

8.6 工厂方法模式与 Spring

Spring 框架的 FactoryBean 接口，可以体现工厂方法模式的运用，如图 8.3 所示。

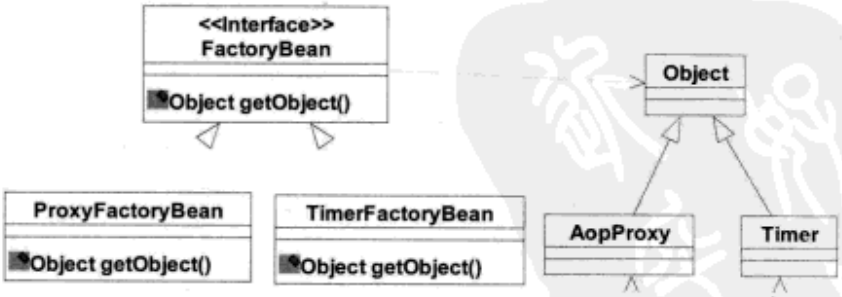

图 8.3

FactoryBean 是抽象工厂，ProxyFactoryBean、TimerFactoryBean 是具体工厂。

Object 是抽象产品，AopProxy、Timer 是具体产品。

[1] 阎宏. Java 与模式. 北京：电子工业出版社，2002，134

第 9 章　AbstractFactory（抽象工厂）模式

9.1　概述

抽象工厂模式是指“提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类。”[1]

此类模式在实际项目开发中，由于需求的变化，需要创建多个对象。此时，如何防止过多的创建方法，以避免客户程序和服务端的紧耦合，这就需要我们运用抽象工厂模式进行解决，具体解决方式可通过创建接口或抽象类，由具体类去实现或继承。

为了较好地理解此模式，我们可以从一个计算机厂生产两种产品的例子去分析。比如一个计算机厂由计算机一厂和计算机二厂组成，其 CPU 产品包括 CPU1 型和 CPU2 型，内存条产品包括内存条 1 型和内存条 2 型，如图 9.1 所示。

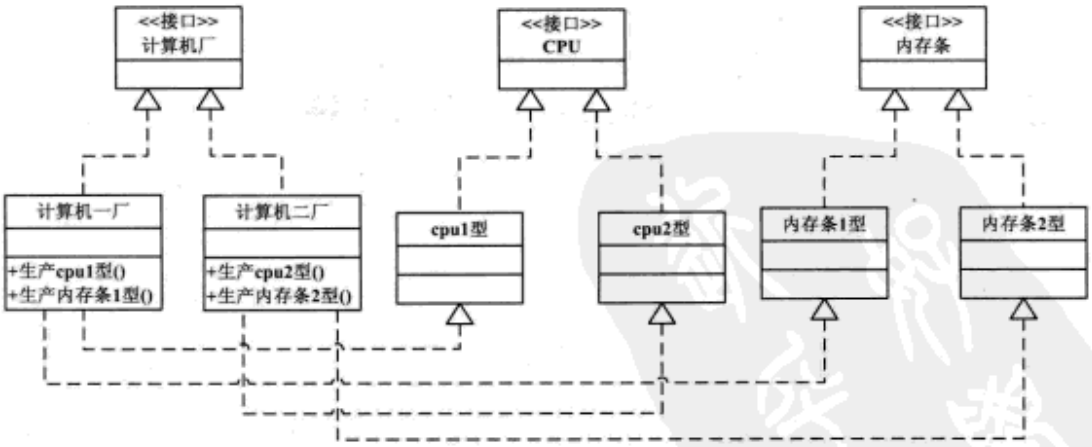

图 9.1

根据目前业界公认的看法，抽象工厂模式结构由抽象工厂、具体工厂和产品族组成。

图 9.1 中，计算机厂是抽象工厂，计算机一厂、计算机二厂是具体工厂；抽象产品 CPU 和具体产品 CPU1 型、CPU2 型，抽象产品内存条和具体产品内存条 1 型、内存条 2 型则是产品族。一个抽象产品与具体产品组成一个产品等级结构。

这里，“抽象工厂”、“具体工厂”、“抽象产品”、“具体产品”与第 8 章说明的一致。

产品等级结构即产品的继承结构，如图 9.1 所示接口是 CPU，其实现类有 CPU1 型和 CPU2 型，则 CPU 与具体的 CPU1、CPU2 之间构成了一个产品等级结构。如果 CPU 是抽象类则其是父

[1] Eric Freeman. Head First 设计模式. 中文版. Oreily Taiwan 公司，译. 北京：中国电力出版社，2007，156

类，而具体的 CPU1、CPU2 是其子类。三者共同构成一个产品等级结构。

产品族：在抽象工厂模式中，产品族是指由同一个公司生产的，位于不同产品等级结构中的一组产品，如 TCL 公司生产的 TCL 手机、TCL 空调。TCL 手机位于手机产品等级结构中，TCL 空调位于空调产品等级结构中。

9.2 应用优势与时机

抽象工厂具备以下一些优势：

- 它可分离具体类的生成，使用户无须了解何种对象被创建。因此，如需变换具体工厂则会变得更简单。
- 它在一个完整的系统中，可使模块之间尽可能地保持独立存在。
- 它可使客户端调用一个产品族的同一个对象，而无须改变。

基于此，我认为可以在以下几种情形下采用抽象工厂模式进行软件设计与实施。

- 当独立的软件系统需要单独进行产品的创建和展示时。
- 当独立的软件系统由产品树套餐中的单个进行配置时。
- 联合使用相关产品对象时。
- 当我们需要建立一个项目框架的通用库时。

9.3 应用情境——男女平等

当前社会，男女平等，由于生活的需要男生与女生一般都需要学习。基于此，本例特结合抽象工厂模式进行实例描述，如图 9.2 所示。

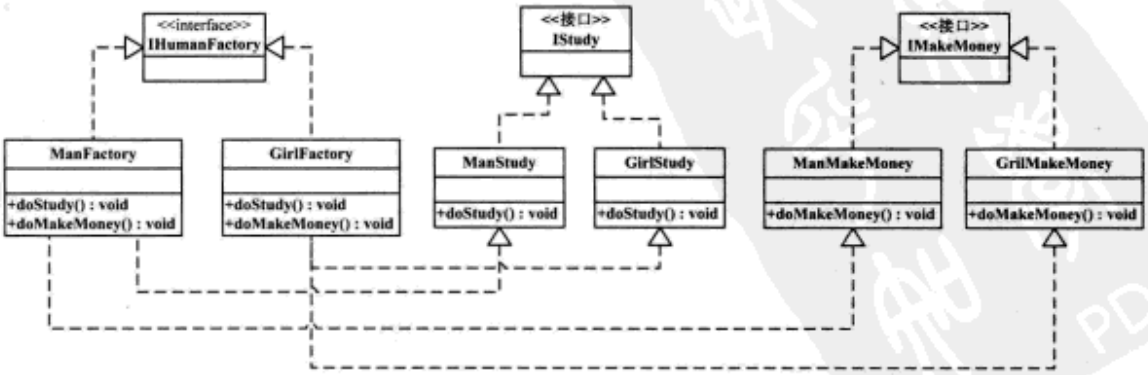

图 9.2

本应用情景的工程名为“程序 9.3.1”，源代码如下所示。

（1）人类工厂接口类

```
package model.abstractfactory;
```

```
///人类工厂接口类
public interface IHumanFactory {
    IStudy getStudy();
    IMakeMoney getMakeMoney();
}
```

（2）女生学习和赚钱工厂类

```
package model.abstractfactory;
//女生学习和赚钱工厂类
public class GirlFactory implements IHumanFactory {
    public IStudy getStudy() {
        return new GirlStudy();
    }
    public IMakeMoney getMakeMoney() {
        return new GrilMakeMoney();
    }
}
```

（3）男生学习和赚钱工厂类

```
package model.abstractfactory;
//男生学习和赚钱工厂类
public class ManFactory implements IHumanFactory {
    public IStudy getStudy() {
        return new ManStudy();
    }
    public IMakeMoney getMakeMoney() {
        return new ManMakeMoney();
    }
}
```

（4）学习接口类

```
package model.abstractfactory;
///学习接口类
public interface IStudy {
    public String  doStudy();
}
```

（5）女生学习类

```
package model.abstractfactory;
//女生学习类
public class GirlStudy implements IStudy {
    public String doStudy() {
    String a="女生学习";
```

```
        System.out.println("女生学习!");
         return a;
    }
}
```

（6）男生学习类

```
package model.abstractfactory;
//男生学习类
public class ManStudy implements IStudy {
    public String doStudy() {
     String a="男生学习!";
        System.out.println("男生学习!");
         return a;
    }
}
```

（7）赚钱接口类

```
package model.abstractfactory;
///赚钱接口类
public interface IMakeMoney {
    public String doMakeMoney();
}
```

（8）女生赚钱类

```
package model.abstractfactory;
//女生赚钱类
public class GrilMakeMoney implements IMakeMoney {
    public String doMakeMoney() {
        String girl="女生赚钱";
        System.out.println("女生赚钱!");
        return girl;
    }
}
```

（9）男生赚钱类

```
package model.abstractfactory;
//男生赚钱类
public class ManMakeMoney implements IMakeMoney {
    public String doMakeMoney() {
    String man="男生赚钱";
        System.out.println("男生赚钱!");
        return man;
    }
```

```
}
```

（10）客户端 test.jsp

```
<HTML>
<!--客户端调用类@author jianghc  -->
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    抽象工厂模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.abstractfactory.*" %>
<br />
<br />
<h4>
<%
IHumanFactory manStudyFactory = new ManFactory();
IHumanFactory girlStudyFactory = new GirlFactory();
IHumanFactory manMakeMoneyFactory = new ManFactory();
IHumanFactory girlMakeMoneyFactory = new GirlFactory();
%>
</h4>
<% out.println("manStudyFactory: " + manStudyFactory.getStudy().doStudy());
%><br />
<%                    out.println("girlStudyFactory:                    "
+girlStudyFactory.getStudy().doStudy());%><br />
<%                    out.println("manMakeMoneyFactory:                 "
+manMakeMoneyFactory.getMakeMoney().doMakeMoney());%><br />
<%
out.println("girlMakeMoneyFactory:"+girlMakeMoneyFactory.getMakeMoney().doMake
Money());%>
<br />
<br />
<br />
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
显示结果:
男生学习!
女生学习!
男生赚钱!
女生赚钱!
```

9.4 抽象工厂与开闭原则

根据本书开闭原则的定义可知，如果要设计成遵循此原则的产品，可通过“添加新的产品族”来实现。因为，在产品等级结构数量不变时，由于工厂等级结构与产品等级结构是平行的关系，因此，添加新的产品族，只需在产品等级结构与工厂等级结构中增加新的元素或属性即可。也就是设计者，只需在系统中添加新的具体工厂类，而无须修改现有的工厂角色或产品角色。

当然，有人也许会想：既然“添加新的产品族”符合开闭原则，那我们也可以通过“添加新的产品等级结构”去实现开闭原则。诚然，这个想法不错。如果我们在产品族总量不变时，添加新的产品等级结构。如此一来，增加了一个与当前产品等级结构平行的新产品等级结构。此时，每个工厂类新增一个工厂方法，即在抽象类或接口中增加了一个新的办法，此时，“继承的子类”或“实现类”都将会改变。显然，这不符合开闭原则的要求。

综上所述，为了遵循开闭原则，抽象工厂只支持添加新产品，而不支持添加新的产品等级结构。

9.5 抽象工厂与工厂方法

表 9.1 抽象工厂和工厂方法的相同与区别之处

模式名	抽象工厂类	工厂方法
是否创建型模式	是	是
是否一个抽象工厂类，可以派生出多个具体工厂类	是	是
抽象产品类	多个抽象产品类，每个抽象产品类可以派生出多个具体产品类	一个抽象产品类，可以派生出多个具体产品类
具体工厂类	每个具体工厂类可以创建多个具体产品类的实例	每个具体工厂类只能创建一个具体产品类的实例

9.6 抽象工厂模式与 Spring

IoC 容器作为 Spring 框架的重要组成部分，由于其即可管理各类普通 Bean 的实例，也可管理工厂的实例。

那么，在软件设计开发人员进行软件平台开发时，只需使客户端程序分离与其所调用对象的实现类与具体工厂类，即可实现抽象工厂模式。也就是说，IoC 容器仅需通过将客户端代码与抽象工厂类进行耦合，就体现了抽象工厂模式的相关原理。

第 10 章　Singleton（单例）模式

10.1　概述

单例模式是指“确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。”[1] 它提供全局访问的方法。

此模式相对比较简单，在现实生活中往往可以运用其去解释一些问题。比如在一个中学里面，一个班级只能有一个班主任。因此，某初三（1）班的学生蒋东、张彭、李飞、陈科均共享班主任的一个实例，如图 10.1 所示。

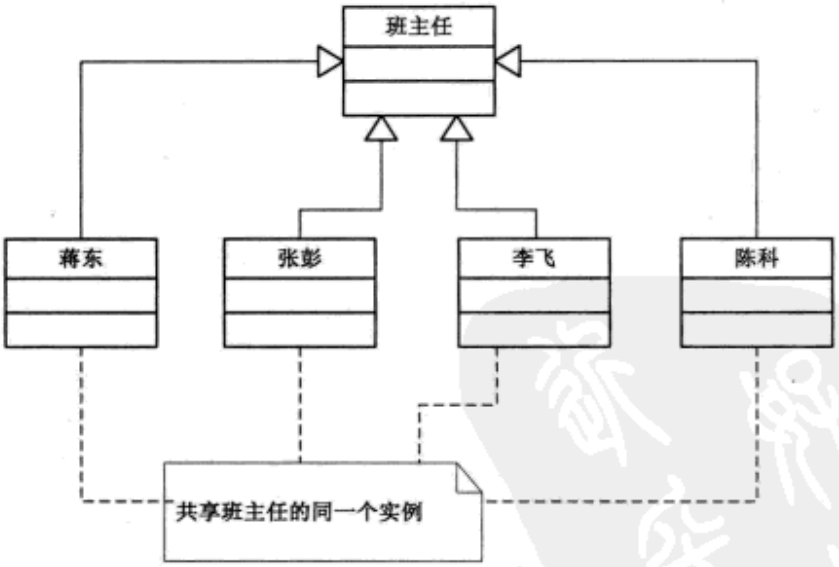

图 10.1

当然，单例模式也分饿汉式和懒汉式，两者的异同点如表 10.1 所示。

表 10.1

类别	相同点	不同点
饿汉式	在类创建的同时就已经创建好一个静态的对象供系统使用，并且以后不再改变	线程安全
懒汉式		在创建实例对象时不加上 synchronized 则会导致对对象的访问不是线程安全

[1] 阎宏. Java 与模式. 北京：电子工业出版社，2002，209

10.2 应用优势与时机

单例具备以下一些优势：

实例数量设置为一个，有利于节约内存空间。

可为整个程序框架提供共享变量，最终提高代码的重用性。

可减少 Java 程序命名的空间。

基于此，我认为可以在以下几种情形下采用单例模式进行软件设计与实施。

- 当软件系统中多个程序只需调用同一实例对象时。
- 无须修改客户端程序，即可使用一个由子类进行扩展的实例时。
- 当系统的性能需要进一步优化时。

10.3 应用情境——饿汉

为了清晰地描述“单例模式”，本例通过编制一个饿汉式、懒汉式单例类与非单例类，运用测试类的方式对单例模式的作用进行展现，如图 10.2 所示。

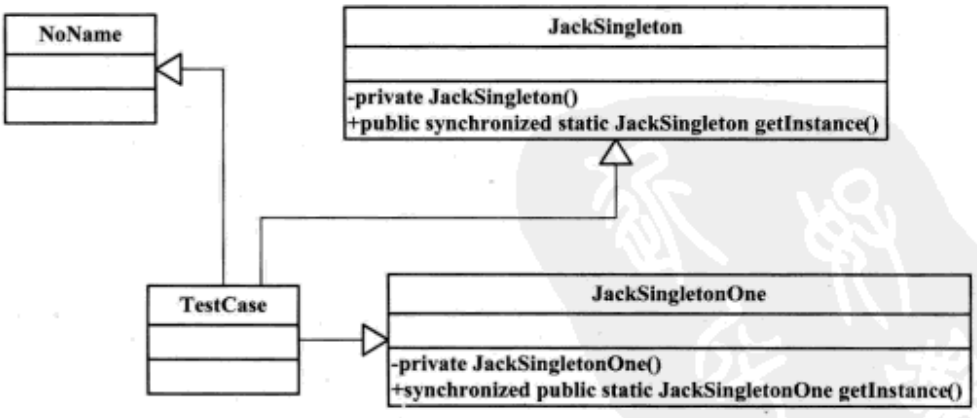

图 10.2

本应用情景的工程名为“程序 10.3.1”，源代码如下所示。

（1）饿汉式单例类

```
package model.singleton;
/*饿汉式单例类*/
public class JackSingleton
{
    private static final JackSingleton jackSingleton = new JackSingleton();
    private JackSingleton()
    {
    }
```

```
    public synchronized static JackSingleton getInstance()
    {
        return jackSingleton;
    }
}
```

（2）懒汉式单例类

```
package model.singleton;

/*懒汉式单例类*/
public class JackSingletonOne {
    private static JackSingletonOne lazySingleton = null;

    private JackSingletonOne() {
    }

    synchronized public static JackSingletonOne getInstance() {
        if (lazySingleton == null) {
            lazySingleton = new JackSingletonOne();
        }
        return lazySingleton;
    }
}
```

（3）非单例模式类

```
package model.singleton;
/*非单例模式类*/
public class NoName {
    public NoName() {
    }
    private int no;
    private String name;
    public int getNO() {
        return no;
    }
    public void setNo(int no) {
        this.no = no;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

（4）单例模式测试类 TestCase.jsp

```
<HTML>
<!--
    @author jianghc
-->
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    单例模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.singleton.*" %>

<br />
<br />
<h4>
<%
//运用饿汉式单例模式，来调用类的实例
 out.println("1.饿汉式单例模式调用类的实例:");
 out.println("*****************************************");
 for(int x=0;x<5;x++)
 {
   int y=x+1;
  out.println("第 "+y+" 次"+"得到的对象值为"+JackSingleton.getInstance());
 }
%><br>
<%
//运用懒汉式单例模式，来调用类的实例
 out.println("2.懒汉单例模式调用类的实例:");
 out.println("*****************************************");
 for(int x=0;x<5;x++)
 {
   int y=x+1;
  out.println("第 "+y+" 次"+"得到的对象值为"+JackSingletonOne.getInstance());
 }
 %><br>
 <%
//不使用单例模式，来调用类的实例
 out.println("3.非单例模式调用类的实例: ");
 out.println("*****************************************");
 for(int x=0;x<5;x++)
 {
     int z=x+1;
     NoName noName=new NoName();
```

```
        out.println("第 "+z+" 次"+"得到的对象值为"+noName);
    }
    //}
    %><br>
    </h4>

    <br />
    <br />
    <br />

    <h4>

    </BODY>
    </HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
1.饿汉式单例模式调用类的实例：******************************************** 第 1 次得到的对象值为 model.singleton.JackSingleton@1091c3f 第 2 次得到的对象值为 model.singleton.JackSingleton@1091c3f 第 3 次得到的对象值为 model.singleton.JackSingleton@1091c3f 第 4 次得到的对象值为 model.singleton.JackSingleton@1091c3f 第 5 次得到的对象值为 model.singleton.JackSingleton@1091c3f

2.懒汉单例模式调用类的实例：******************************************** 第 1 次得到的对象值为 model.singleton.JackSingletonOne@1b7c9d1 第 2 次得到的对象值为 model.singleton.JackSingletonOne@1b7c9d1 第 3 次得到的对象值为 model.singleton.JackSingletonOne@1b7c9d1 第 4 次得到的对象值为 model.singleton.JackSingletonOne@1b7c9d1 第 5 次得到的对象值为 model.singleton.JackSingletonOne@1b7c9d1

3.非单例模式调用类的实例：******************************************** 第 1 次得到的对象值为 model.singleton.NoName@a498d0 第 2 次得到的对象值为 model.singleton.NoName@2ca90c 第 3 次得到的对象值为 model.singleton.NoName@e873b 第 4 次得到的对象值为 model.singleton.NoName@1d07324 第 5 次得到的对象值为 model.singleton.NoName@3777c8
```

10.4 单例模式与 Struts

目前 struts 主流的框架包括 Struts1 和 Struts2，本节以依托各类请求调用相关业务组件的 Action 类作为例子，进行单例模式的阐述（由于 Struts2 的 Action 对象具备使所有请求均会产生对应的一个实例的功能，因而我们可以将其理解为非单态模式）。

在 Struts1 的 Action 中实行的是线程安全的单例模式，其 Action 只可运用一个实例来处理相关的全部请求。并且 Struts1 具备使所有模块均可拥有生命周期的功能，尤其要注意的是各个模块中的 Action 都需要共同分享同一生命周期。

综上所述，为了便于理解，其实我们也可以这么认为，“struts 1 的数据不可在 Action 中编制，所以其可实现单例模式；而 Struts 2 的数据可在 Action 中编制，因而其不能运用单例模式。”

第 11 章　Builder（建造者）模式

11.1　概述

建造者模式是指“将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示”。[1]

粗看其定义，大家可能有点迷惑不解。其实，我们可以把它形象化。

“复杂对象的构建与它的表示分离”可理解成一辆汽车，无论使用何种品牌的部件，只要能正常安装即可。

“使得同样的构建过程可以创建不同的表示”可理解成同样的部件可采取多种安装方式。

为了便于读者进一步理解建造者模式，本节通过模式的角色与实例进行解说，建造者模式由以下 4 个角色构成。

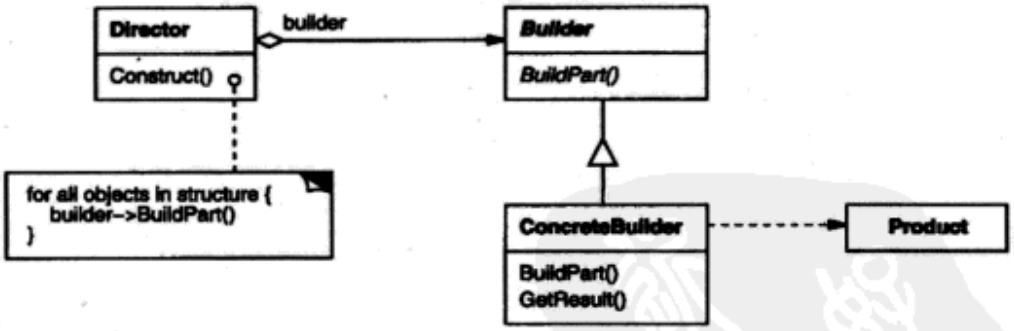

图 11.1

（1）Builder（抽象建造者角色）

- 为创建一个 Product 对象的各个部件指定抽象接口。

（2）ConcreteBuilder（具体建造者角色）

- 实现 Builder 的接口以构造和装配该产品的各个部件。
- 定义并明确它所创建的表示。
- 提供一个检索产品的接口。

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，63

（3）Director（导演者角色）

- 构造一个使用 Builder 接口的对象。

（4）Product（产品角色）

- 表示被构造的复杂对象。ConcreteBuilder 创建该产品的内部表示并定义它的装配过程。
- 包含定义组成部件的类，包括将这些部件装配成最终产品的接口。[1]

本章通过一个电冰箱制造厂的例子，运用图解的方式来诠释建造者模式，如图 11.2 所示。在这个电冰箱制造厂的例子中各个角色说明如下：

- 抽象的建造者：制造一台电冰箱需要那些零件
- 具体建造者：制造出各个零件，返还电冰箱。
- 导演者：厂长，调派具体建造者制造零件，生产电冰箱。
- 产品角色：电冰箱。

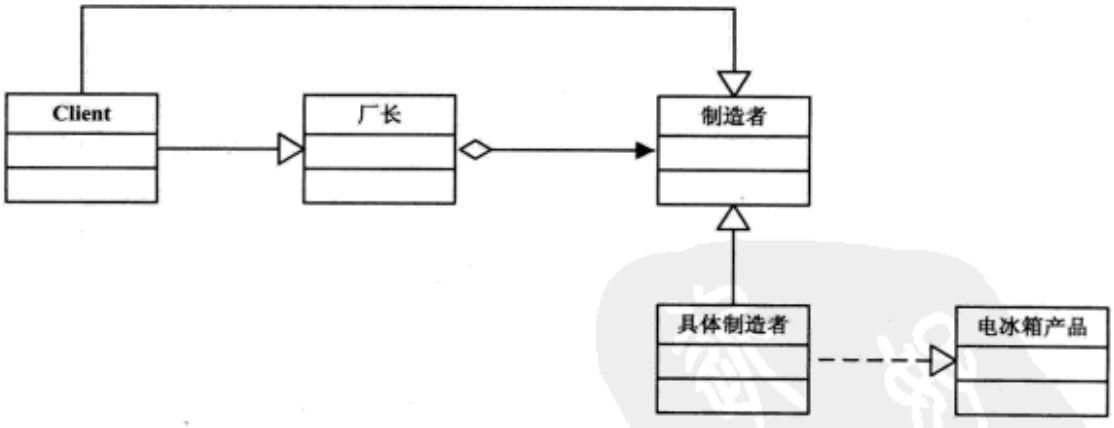

图 11.2

厂长即导演者，制造者即抽象的建造者，具体制造者即具体建造者，电冰箱产品即产品角色。

11.2 应用优势与时机

建造者具备以下一些优势：

- 客户端无须了解产品内部情况。
- 通过新的具体建造者，即可改变产品的内部结构。
- 四个角色构成相互独立，具备高扩展性。
- 通过对建造过程的细化，可降低项目维护的风险。

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，65

基于此，我认为可以在以下几种情形下采用建造者模式进行软件设计与实施。

- 产品对象有一个较复杂的内部结构，并且其具体的构建方法面临着复杂的变化时。
- 产品类的构造过程中，可使被构造对象有不同表现时。
- 创建对象时，需要调用软件系统的其他对象。

11.3 应用情境——西门子冰箱产品设计

为了考察系统设计师对建造者模式的理解程度，我设计了一个问题。假设西门子冰箱制造厂产品升级，由厂长策划，技术工人改装新的压缩机。如果用建造者模式，请问如何设计比较合理？有人给出的设计如图 11.3 所示。

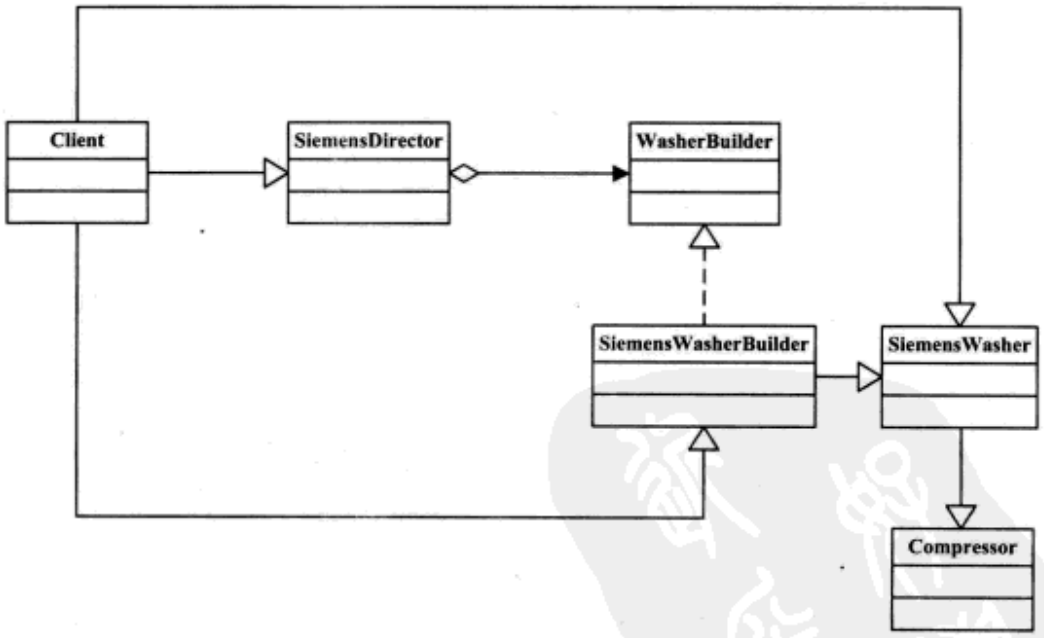

图 11.3

本应用情景的工程名为“程序 11.3.1”，源代码如下所示。

（1）西门子导演者

```
package model.builder;
/*
 * 西门子导演者 导演者：
这里，由导演者指导建造者以一定的方式生产西门子冰箱
 */
public class SiemensDirector {
    WasherBuilder builder;
    /**
     * 为导演者配置一个建造者
     */
```

```
    public SiemensDirector(WasherBuilder builder){
        this.builder = builder;
    }
    /**
     * 按照一定的方式或规则建造冰箱
     */
    public void contruct(){
        builder.buildeWasherSkeleton();
        builder.buildeWasherEngine();
        builder.buildeWasherWheels();
        builder.buildeWasherBody();
    }
}
```

（2）建造冰箱的框架

```
package model.builder;
public interface WasherBuilder {

    /**
     * 建造冰箱的框架 抽象建造者:
     */
    public void buildeWasherSkeleton();

    /**
     * 给冰箱装上压缩机
     */
    public void buildeWasherEngine();

    /**
     * 给冰箱装上轮子
     */
    public void buildeWasherWheels();

    /**
     * 安装冰箱身（包含冰箱门、涂颜色）
     */
    public void buildeWasherBody();
}
```

（3）具体建造者

```
package model.builder;
/*
 * 具体建造者 建造者:（只是改变自己的内部实现--把西门子旧压缩机换成西门子新压缩机）
 */
public class SiemensWasherBuilder implements WasherBuilder {
    private SiemensWasher siemensWasher = new SiemensWasher();
    /***
      安装冰箱身（包含冰箱门、涂颜色）*/
```

```
    public void buildeWasherBody() {
        siemensWasher.setWasherDoor("冰箱门");
        siemensWasher.setWasherColor("银色");
    }
    /**
     * 给冰箱装上压缩机
     */
    public void buildeWasherEngine() {
        Compressor audiEngine = new Compressor("新型压缩机");
        siemensWasher.setWasherEngine(audiEngine);
    }
    /**
     * 建造冰箱的框架
     */
    public void buildeWasherSkeleton() {
        siemensWasher.setWasherSkeleton("冰箱框架");
    }
    /**
     * 给汽冰箱装上轮子
     */
    public void buildeWasherWheels() {
        siemensWasher.setWasherWheels("冰箱轮");
    }
    public SiemensWasher retrieveWasher(){
        return siemensWasher;
    }
}
```

（4）西门子冰箱

```
package model.builder;
/*
 * 产品   西门子冰箱(产品):
 */
public class SiemensWasher {
    private String washerSkeleton;
    private Compressor washerEngine;
    private String washerWheels;
    private String washerDoor;
    private String washerColor;
    public SiemensWasher(){}
    /**
     * 冰箱的颜色
     */
    public void setWasherColor(String washerColor) {
        this.washerColor = washerColor;
    }
    /**
     * 冰箱门
     */
```

```
        public void setWasherDoor(String washerDoor) {
            this.washerDoor = washerDoor;
        }
        /**
         * 冰箱压缩机，压缩机在这里是一个对象
         */
        public void setWasherEngine(Compressor washerEngine) {
            this.washerEngine = washerEngine;
        }
        /**
         * 冰箱的框架
         */
        public void setWasherSkeleton(String washerSkeleton) {
            this.washerSkeleton = washerSkeleton;
        }
        /**
         * 冰箱轮子
         */
        public void setWasherWheels(String washerWheels) {
            this.washerWheels =washerWheels;
        }
        /**
         * 一个冰箱对象的描述
         */
        public String toString(){
            return  washerSkeleton+  ","  +  washerEngine.toString()  +","  +
washerWheels + "," +
            washerDoor + "," + washerColor  ;
        }
    }
```

（5）一个部件对象（压缩机）冰箱的核心

```
package model.builder;
/*
 * 一个部件对象（压缩机）冰箱的核心
 */
public class Compressor {
    private String name;
    public Compressor(String name){
        this.name = name;
    }
    public String toString(){
        return this.name;
    }
}
```

（6）客户端：test.jsp

```
<HTML>
```

```
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    建造者模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.builder.*" %>

<br />
<br />
<h4>
<%
//      创建一个建造者对象
        SiemensWasherBuilder builder = new SiemensWasherBuilder();
//      创建一个导演者，并为它配置一个建造者
        SiemensDirector directorOne = new SiemensDirector(builder);
//      导演者将通知建造者去创建产品
//      在这个过程中，建造者将会根据导演者的请求，去创建组织并创建产品对象
        directorOne.contruct();
//      从具体建造者中检索产品（返回的是冰箱的轮子）。
        SiemensWasher siemensWasher = builder.retrieveWasher();
        %>
</h4>
<% out.println(siemensWasher);  %><br />
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
冰箱框架，新型压缩机，冰箱轮，冰箱门，银色
```

11.4 建造者与抽象工厂

建造者和抽象工厂的相同与区别之处如表 11.1 所示。

表 11.1 建造者和抽象工厂的相同与区别之处

模式名	建造者	抽象工厂类
是否创建型模式	是	是
是否将对象创建过程与使用过程相分离	是	是

（续表）

模式名	建造者	抽象工厂类
用户需要使用对象时，是否只需了解创建什么，而无须了解对象如何创建。	是	是
创建过程有所改动时，是否只对修改创建过程进行，而无须修改使用过程。	是	是
构建过程	运用共同的构建过程形成有差异的表示。	运用不同的构建过程形成有差异的表示。
返回	返回整体的一个产品。	返回一些相关的产品。
客户端	客户端指挥抽象建造者类去生成对象，并且合成某些类来构成建造类。	客户端运用抽象工厂生成自身需要的对象。

第 12 章　Prototype（原型）模式

12.1　概述

原型模式是指“用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。”[1]

粗看其定义，大家可能有点迷惑不解。其实，我们可以把它简化。

“用原型实例指定创建对象的种类”可理解为“新建一个对象，运用一个原型指定对象的种类。”

“通过拷贝这些原型创建新的对象”可理解为“通过现有对象的复制实现新对象的创建”。

为了便于读者进一步理解原型模式，本章通过一个钥匙的故事进行解说。话说当年，我在大学课堂夜自修时，突然接到舍友告知丢失宿舍钥匙的来电。于是我马上去小卖部找师傅配钥匙。只见师傅把我的钥匙放在机器上，一按按钮一下子就出来两个一模一样的钥匙，如图 12.1 所示。

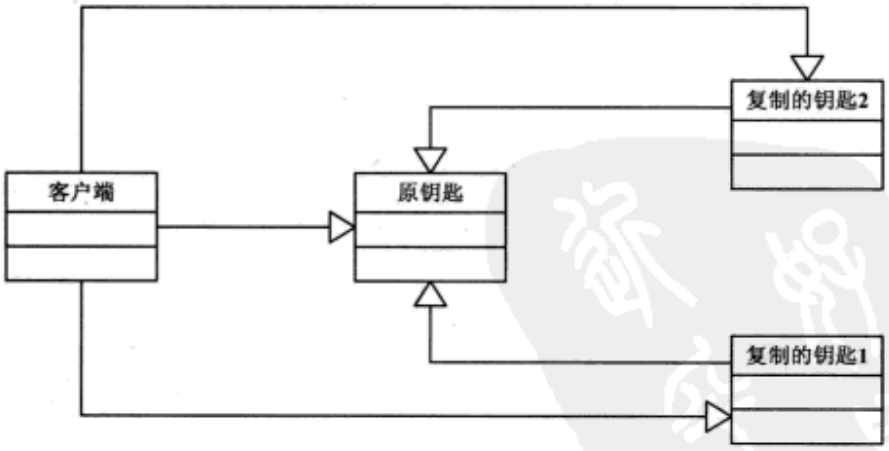

图 12.1

根据，目前业界公认的看法。原型模式结构，由“客户角色”、“抽象原型角色”、“具体原型角色”组成。

- 客户角色：客户类提出创建对象的请求。
- 抽象原型角色：这是一个抽象角色，通常由一个 Java 接口或 Java 抽象类实现。此角色给出所有的具体原型类所需的接口。
- 具体原型角色：被复制的对象。此角色需要实现抽象的原型角色所要求的接口。[2]

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，77

[2] 阎宏. Java 与模式. 北京：电子工业出版社，2002，323

在图 12.1 中，客户端是客户角色，原钥匙即抽象原型角色，复制的钥匙 1、钥匙 2 即复制的具体原型角色。

12.2 应用优势与时机

原型模式具备以下一些优势：

- 允许动态地增加或减少产品类。由于创建产品类实例的方法是产品类内部具有的，因此增加新产品对整个结构没有影响。
- 提供简化的创建结构。
- 具有给一个应用软件动态加载新功能的能力。
- 产品类不需要非得有任何事先确定的等级结构，因为原型模式适用于任何的等级结构。[1]

基于此，我认为可以在以下几种情形下采用原型模式进行软件设计与实施。

- 当系统创建动态加载的对象并且产品具有一定的层次性时。
- 当产品结构经常变化时。
- 当需要复制一个现有的对象进行新对象的生成时。

12.3 应用情境——克隆猪

当代社会，克隆技术相当发达。本应用情境，主要通过猪的克隆进行原型模式的阐述，如图 12.2 所示。

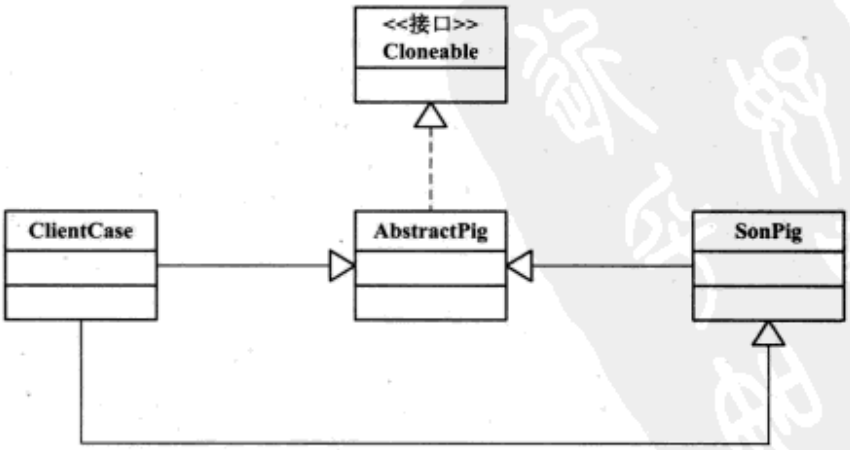

图 12.2

本应用情景的工程名为“程序 12.3.1”，源代码如下所示。

（1）原型类

[1] 阎宏. Java 与模式. 北京：电子工业出版社，2002，341

```
package model.prototype;
/**
 * 原型类(以猪为例) ,注意要实现Cloneable接口
 *
 * @author jackjiang
 *
 */
public abstract class AbstractPig implements Cloneable
{
    public String pigName;
    public String getPigName()
    {
        return this.pigName;
    }
    public void setPigName(String pigName)
    {
        this.pigName = pigName;
    }
    /** *//**
     * 重写clone()方法
     */
    @Override
    public Object clone()
    {
        try
        {
            return super.clone();
        } catch (CloneNotSupportedException e)
        {
            System.out.println("此对象不支持复制");
        }
        return null;
    }
}
```

（2）实现类

```
package model.prototype;
/** *//**
 * 给原型对象赋值
 *
 * @author jackjiang
 *
 */
public class SonPig extends AbstractPig
{
    public SonPig()
    {
        setPigName("zhejiangPig");
```

```
    }
}
```

（3）客户端，test.jsp

```
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    原型模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.prototype.*" %>

<br />
<br />
<h4>
<%
AbstractPig pig = new SonPig();
out.println(pig.getPigName());
// 通过对象的clone()方法,即可获得对象的一个copy.
AbstractPig pig2 = (AbstractPig) pig.clone();
%>
</h4>
<%out.println("拷贝: "+pig2.getPigName()); %><br />
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
    zhejiangPig
拷贝: zhejiangPig
```

12.4 原型模式与 Spring

基于 Spring 框架的 Bean，可在 context.xml 文件中用单例模式和原型模式进行创建。原型模式的创建方式说明如下：

在 Spring 框架的 context.xml 文件中编制以下内容：

```
<?xml version="1.0" encoding="GBK"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id="test" class="spring.bean.scope.prototype.TestBean" scope="prototype"/>
    </beans>
    在 context.xml 文件中移除 scope="prototype" ，将其修改为 singleton="false"。
```

因为 context.xml 中只支持 prototype 与 singleton 模式，当 singleton 设置为 false 时，自然此处运用了原型模式。

第四部分

设计结构派
——细说结构型模式

第 13 章　Adapter（适配器）模式

13.1　概述

适配器模式是指“将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。其别名为包装器（Wrapper）。”[1]适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。

粗看其定义，大家也许会有点疑惑。其实，我们可以将它通俗化。

所谓适配器模式，我们可理解为“将两种没有联系的功能代码，通过中间代码相结合。”

为了便于读者进一步理解适配器模式，本章通过模式的结构与实例进行解说。

1．结构

（1）类适配器

类适配器使用多重继承对一个接口与另一个接口进行匹配，如图 13.1 所示。

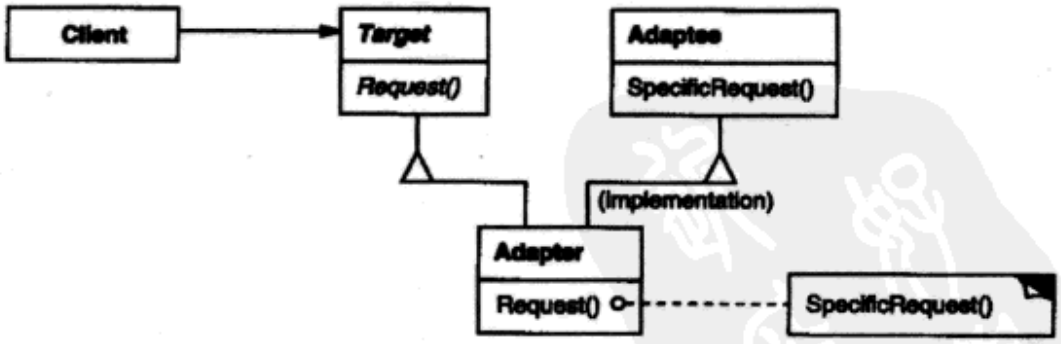

图 13.1

（2）对象适配器

对象匹配依赖于对象组合，如图 13.2 所示。

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，92

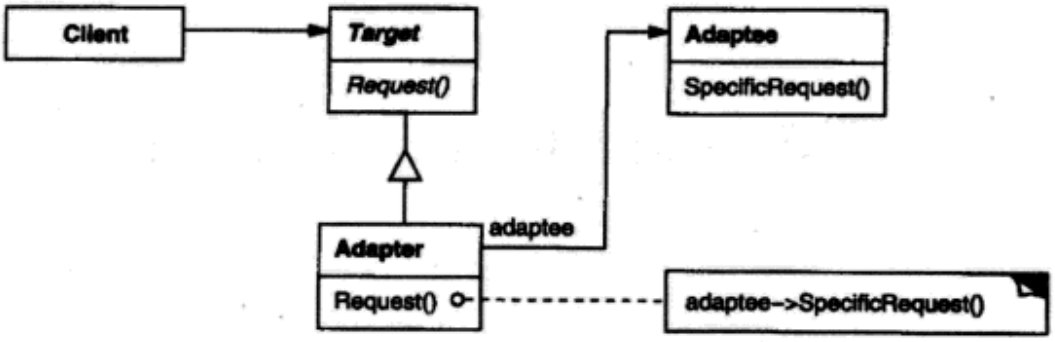

图 13.2

（3）适配器参与者

- 目标角色 Target（类或接口）：定义 Client 使用的与特定领域相关的接口。
- Client：与符合 Target 接口的对象协同。
- 源角色 Adaptee（类或接口）：定义一个已经存在的接口，这个接口需要适配。
- 适配器角色 Adapter（适配者类）：对 Adaptee 的接口与 Target 接口进行适配。[1]

2. 实例

（1）类适配器实例

国内足球俱乐部请洋教练的现象越来越普遍，为了使洋教练与足球队员进行更好的沟通，需要翻译进行训练技巧的解说，如图 13.3 所示。

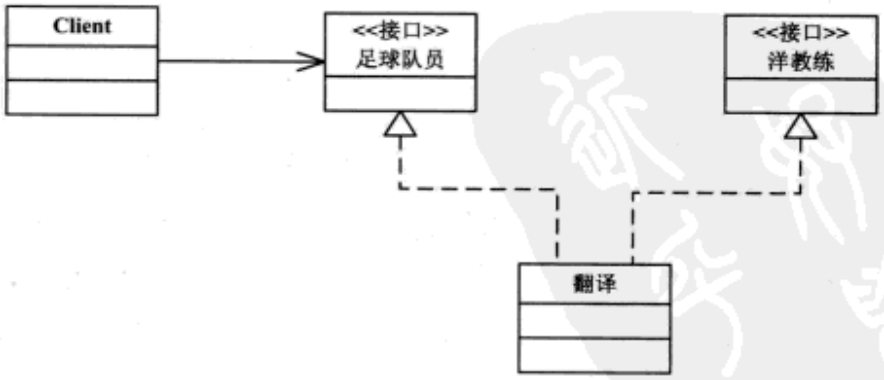

图 13.3

- 足球队员接口即 Target。
- 洋教练接口即 Adaptee。
- 翻译即 Adapter。

（2）对象适配器实例

某外包公司接手一个软件维护项目，需要开发一个 FTP 上传模块。原系统已定义好了 FTP 界

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，93~94

面类。由于时间紧迫，开发人员需要调用已有的 FTP 上传、下载方法。此方法封装在通用类库中，没有源代码。使用适配器模式，程序的结构如图 13.4 所示。

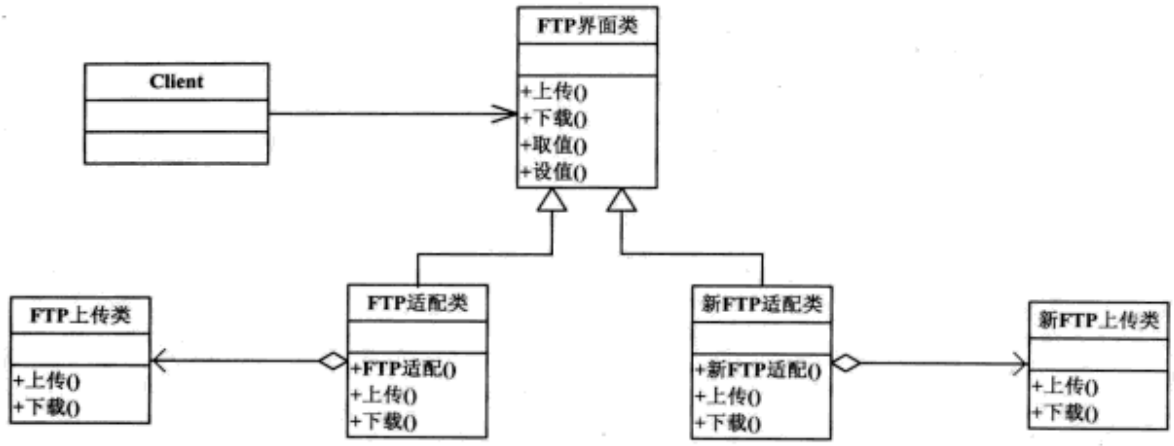

图 13.4

- FTP 界面类即 Target。
- FTP 上传类、新 FTP 上传类即 Adaptee。
- FTP 适配类、新 FTP 适配类即 Adapter。

13.2 应用优势与时机

适配器具备以下一些优势：

- 适配器类，可使任何两个无直接联系的类在一起运行。
- 目标（Target）角色可通过被适配者（Adaptee）具体实现。
- 适配器类，大大提高了类的复用度与灵活性。

基于此，我认为可以在以下几种情形下采用适配器模式进行软件设计与实施。

- 软件系统有使用已有类的要求，但是类的接口不适合。
- 设计师需要设计一个可多次使用的类，并且与其他没有直接关系或不可知的类进行关联。
- 调用现有的某些子类，无须派生它们的每个接口。

13.3 应用情境——鞋子生产业务扩展

鞋子生产商可以生产鞋子，服装生产商可以生产服装，某鞋子生产公司决定将业务扩展到服装领域。下面介绍 3 种实现方式。

1. 传统的做法

增加一个服装生产的接口，并在实现中增加生产服装的实现，如图 13.5 所示。

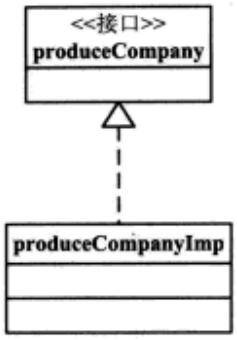

图 13.5

本应用情景的工程名为“程序 13.3.1”，源代码如下所示。

（1）公司产品接口

```
package model.adapter.noadapter;

public interface produceCompany{
    String getShoe();
    String getDress();
}
```

（2）接口实现类

```
package model.adapter.noadapter;

public class produceCompanyImp implements produceCompany{
   public String getShoe() {
       String a="Shoe";
       return a;
   }
   public String getDress() {
       String b="Dress";
       return b;
   }
}
```

（3）客户端（test.jsp）

```
<HTML>
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    不使用适配器模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
```

```
<%@ page pageEncoding="GB2312" language="java" import="model.adapter.
noadapter.*" %>

<br />
<br />
<h4>
<%
//      创建一个对象
        produceCompanyImp obj = new produceCompanyImp();
        %>
</h4>
<% out.println("公司产品:"+ obj.getDress());
  out.println( obj.getShoe());%><br />
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
公司产品: Dress Shoe
```

2. 类适配器（如图 13.6 所示）

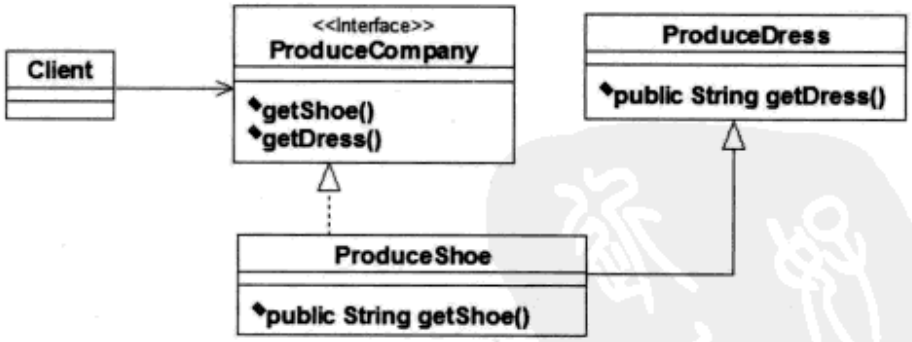

图 13.6

本应用情景的工程名为“程序 13.3.2”，源代码如下所示。

（1）目标类

```
package model.adapter.classadapter;
/*
 Target 目标
 * */
public interface ProduceCompany{
        String getShoe();
        String getDress();
    }
```

（2）适配器类

```
package model.adapter.classadapter;
/**
 * Adapter 适配器
 * */
public class ProduceShoe extends ProduceDress implements ProduceCompany {
    public String getShoe() {
        System.out.println("--鞋--");
        return "shoe";
    };
}
```

（3）被适配器类

```
package model.adapter.classadapter;
/**
 * Adaptee 被适配器
 * */
public class ProduceDress {
    public String getDress(){
        System.out.println("--服装--");
        return "dress";
    };
}
```

（4）客户端（classtest.jsp）

```
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    类适配器模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.adapter.
classadapter.*" %>

<br />
<br />
<h4>
<%
//      创建一个对象
            ProduceShoe p = new ProduceShoe();
```

```
        %>
</h4>
<% out.println("公司产品:"+  p.getShoe());
  out.println(  p.getDress());%><br />
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
公司产品: shoe dress
```

3. 对象适配器（如图 13.7 所示）

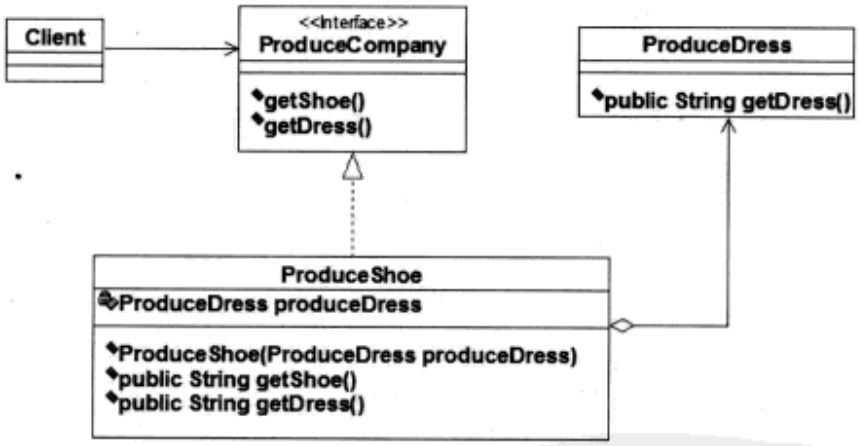

图 13.7

本应用情景的工程名为“程序 13.3.3”，源代码如下所示。

（1）目标类

与类适配器中的目标类相同。

（2）适配器类

```
/**
 * Adapter 适配器  对象适配器与类适配器的区别，只是本处用组合的方式引用 ProduceDress
 * */
public class ProduceShoe  implements ProduceCompany {
    ProduceDress produceDress;
    ProduceShoe(ProduceDress produceDress){
            this.produceDress = produceDress ;
        }
   public String getShoe() {
       System.out.println("--鞋--");
       return "shoe";
   }
    public String getDress() {
```

```
        System.out.println("--衣服--");
         return "dress";
      };
   }
```

（3）被适配器类

与类适配器中的被适配器类相同。

（4）客户端（objecttest.jsp）

```
<HTML>
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    对象适配器模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.adapter.
objectadapter.*" %>

<br />
<br />
<h4>
<%
//      创建一个对象
          ProduceShoe p = new ProduceShoe(null);
          p.getShoe();
          p.getDress();
        %>
</h4>
<% out.println("公司产品:"+ p.getShoe());
  out.println( p.getDress());%><br />
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
公司产品:shoe dress
```

4. 推荐方式

综合“传统的做法、类适配器、对象适配器”的代码结构，推荐使用对象适配器。

13.4 Spring 与 Hibernate 在适配器模式中的应用

当前，不少 IT 公司运用 Spring 整合 Hibernate 进行系统开发。对于 Spring 而言，它提供了 HibernateDaoSupport、HibernateTemplate、HibernateCallBack 等工具类或接口来支撑 DAO 组件的实现。本节以 Spring 结合 HibernateDaoSupport 为例进行适配器模式的阐述，如图 13.8 所示。

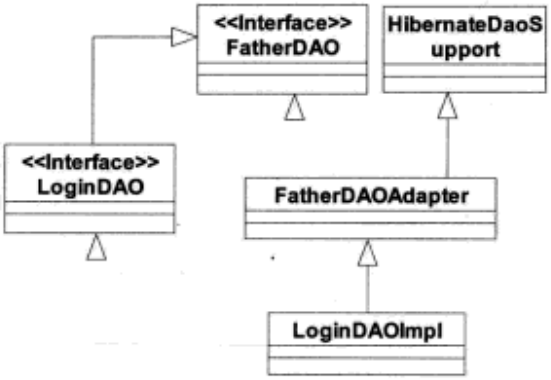

图 13.8

（1）父 DAO 接口

```
package spring.adpater;
import java.util.List;
public interface FatherDAO
{
    public void saveInfo(Object obj1);
    public void delInfo(Object obj2);
    public Object findByNo(String no);
    public List findAll();
}
```

（2）通用 DAO 适配器

```
package spring.adpater;
import java.util.List;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
    public abstract class FatherDAOAdapter extends HibernateDaoSupport
implements FatherDAO
{
    protected FatherDAOAdapter() {
    }
    public void saveInfo(Object obj1) {
    getHibernateTemplate().saveOrUpdate(obj1);
    }
    public void delInfo(Object obj2) {
    getHibernateTemplate().delete(obj2);
    }
```

```
        public List findAll(String objName) {
        String SearchString = "from " + objName;
        return getHibernateTemplate().find(SearchString);
        }
    }
```

（3）登录 DAO 接口

```
    package spring.adpater;
    import java.util.List;
    import spring.adpater.FatherDAO;
        public interface LoginDAO extends FatherDAO
        {
        public List findByName(String name);
        public List findByPassword(String password);
        }
```

（4）登录 DAO 实现类

```
    package spring.adpater;
    import java.util.List;
    import spring.adpater.FatherDAOAdapter;
    import spring.adpater.LoginDAO;
        public abstract class LoginDAOImpl extends FatherDAOAdapter implements
LoginDAO
    {
        public static final String Login = "spring.adpater.Login";
        public static final String NAME = "name";
        public static final String PASSWORD = "password";
        public List findAll() {
        return super.findAll(NAME);
        }
    }
```

第 14 章　Bridge（桥接）模式

14.1　概述

桥接模式是指："将抽象部分与它的实现部分分离，使它们都可以独立地变化。"[1]粗看其定义，大家也许会有点疑惑。其实，我们可以将它通俗化。

所谓桥接模式可理解为"将抽象部分与其实现部分分开，使之任意增减，而无须受其他约束"。为了便于读者进一步理解建造者模式，本章通过模式的角色与实例进行解说。

1. 模式的角色（如图 14.1 所示）

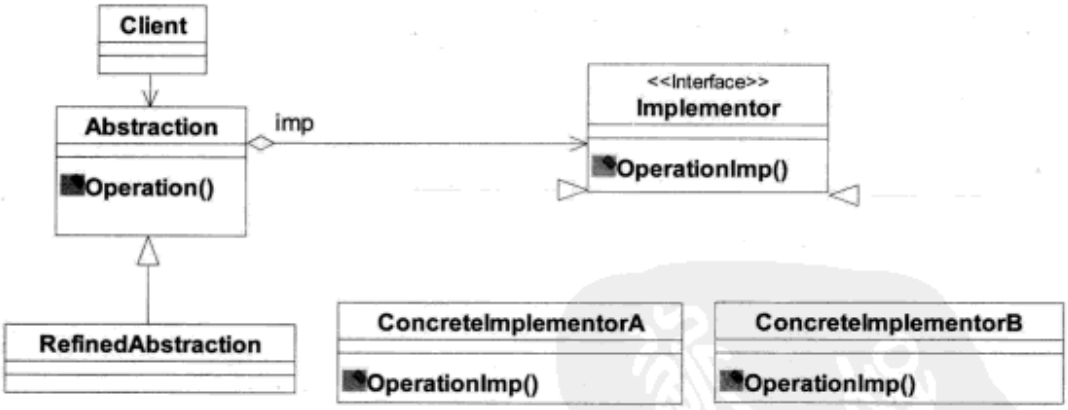

图 14.1

- Abstraction：定义抽象类的接口，维护一个指向 Implementor 类型对象的指针，将 Client 的请求转发给它的 Implementor。RefinedAbstraction 扩充由 Abstraction 定义的接口。
- Implementor：定义实现类的接口，该接口不一定要与 Abstraction 的接口完全一致，事实上这两个接口可以完全不同。一般来讲，Implementor 接口仅提供基本操作，而 Abstraction 则定义了基于这些基本操作的较高层次的操作。
- ConcreteImplementor：实现 Implementor 接口并定义它的具体实现。[2]

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，100

[2] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，101~102

2.实例

当前，此类模式在实际生活中有着较好的运用空间。比如，我们日常生活中经常使用的门。从其材质角度划分可包括塑料门、塑钢门、木门，从其形式角度划分可包括平开门、推拉门，折叠门。基于此，本实例可设计为如图 14.2 所示。

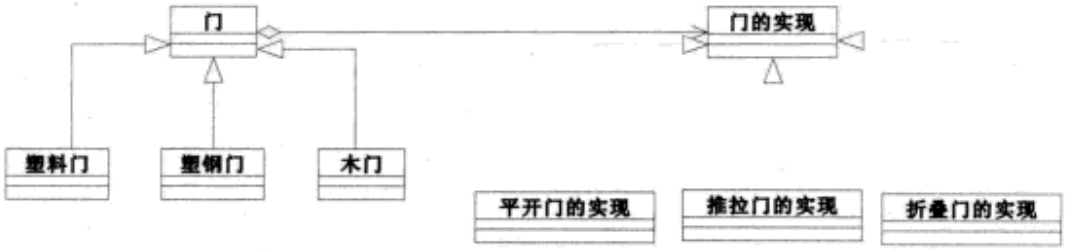

图 14.2

“门”即 Abstraction，“塑料门、塑钢门、木门” 即 RefinedAbstraction，“门的实现”即 Implementor，“平开门的实现、推拉门的实现、折叠门的实现” 即 ConcreteImplementor。

14.2 应用优势与时机

桥接具备以下一些优势：

- 使接口与实现各自独立。
- 使接口实现类的扩展性大大增强。
- 保护了部分实现内容，在扩展与变更实现内容时，无须重新编译原客户程序。
- 基于此，我认为可以在以下几种情形下采用桥接模式进行软件设计与实施。
- 需要增强抽象角色与具体角色的灵活性，以避免两者之间的直接关联。
- 实现类的变动，不影响客户端的使用。
- 抽象接口与类的实现通过组合，均可在子类上进行扩展。

14.3 应用情境——房子

当代社会，房子是每个老百姓的生活大事。本应用情境运用桥接模式展现房子与公寓、别墅之间的关系，如图 14.3 所示。

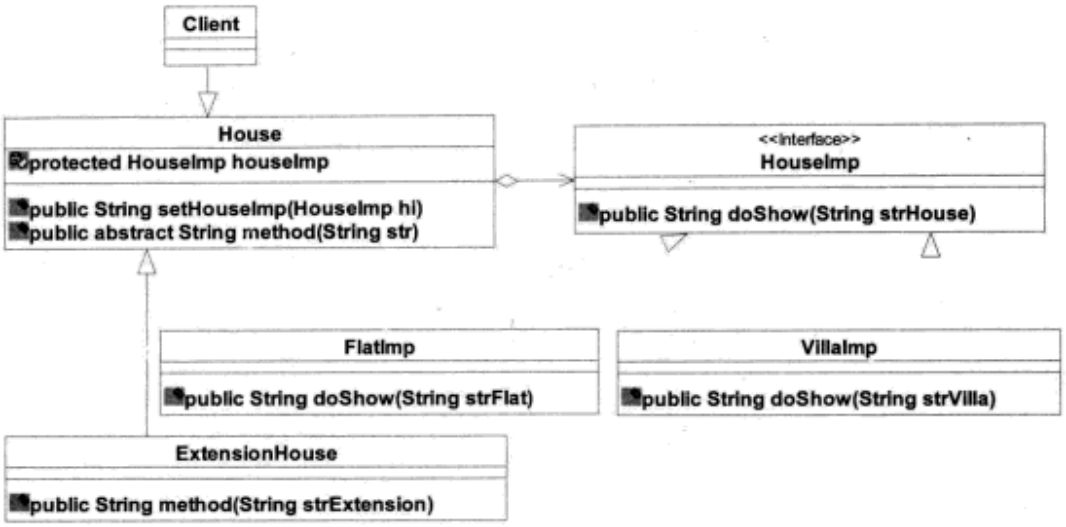

图 14.3

本应用情景的工程名为“程序 14.3.1”，源代码如下所示。

（1）房子抽象类

```
package model.bridge;
//Abstraction  定义抽象类的接口，维护一个指向实现者(implementor)接口类型(HouseImp)的指针
public abstract class House{
    protected HouseImp houseImp;
    public String setHouseImp(HouseImp hi){
     houseImp=hi;
        return null;
    }
    public abstract String method(String str);
}
```

（2）房子扩展类

```
package model.bridge;
//RefinedAbstraction(ExtensionHouse): 扩充由Abstraction定义的接口。
public class ExtensionHouse extends House{
    public String method(String strExtension){
       String e=strExtension+"\nEXTENSION HOUSE";
       return this.houseImp.doShow(e);
    }
}
```

（3）房子实现类接口

```
package model.bridge;
/* Implementor(HouseImp):定义实现类的接口，该接口不一定要与Abstraction的接口完全一致，
事实上这两个接口可以完全不同。一般来讲Implementor接口仅提供基本操作，而Abstraction则定义
了这些基本操作的较高层次的操作。*/
public interface HouseImp{
    public String doShow(String strHouse);
}
```

（4）公寓实现类

```
package model.bridge;
//ConcreteImplementor(FlatImp): 实现ImageImp并定义它的具体实现。-公寓
public class FlatImp implements HouseImp{
    public String doShow(String strFlat){
     String a="房子属于 公寓!";
       System.out.println(strFlat+" 房子属于 公寓!");
     return a;
    }
}
```

（5）别墅实现类

```
package model.bridge;
//ConcreteImplementor(VillaImp): 实现ImageImp并定义它的具体实现。-别墅
public class VillaImp implements HouseImp{
    public String doShow(String strVilla){
       String v="房子属于 别墅!";
       System.out.println(strVilla+" 房子属于 别墅!");
        return v;
    }
}
```

（6）客户端（test.jsp）

```
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    桥接模式
    </TITLE>
</HEAD>
```

```
<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.bridge.*" %>

<br />
<br />
<h4>
<%
      HouseImp villaimp = new VillaImp();
      ExtensionHouse ext = new ExtensionHouse();
      ext.setHouseImp(villaimp);
      ext.method("Show1:");
        %>
</h4>
<% out.println("房子类别:"+ ext.method("Show1:")); %><br />
  <%
      HouseImp flatimp = new FlatImp();
      ext.setHouseImp(flatimp);
      ext.method("Show2:");
        %>
        <% out.println(  ext.method("Show2:"));%><br />
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
房子类别：房子属于 别墅！房子属于 公寓！
```

第 15 章　Composite（组合）模式

15.1　概述

组合模式又称合成模式。国外设计模式大师 GOF 把合成模式定义为："将对象组合成树形结构以表示'部分-整体'的层次结构。"[1]

粗看其定义，大家可能有点迷惑不解。其实，我们可以把它简化。

"将对象组合成树形结构以表示'部分-整体'的层次结构"可理解为"通过树形结构，同等显示多个独立的对象以及他们复合而成的合成对象。"

为了便于读者进一步理解组合模式，本章通过模式的结构与实例进行解说。

1. 结构（如图 15.1 所示）

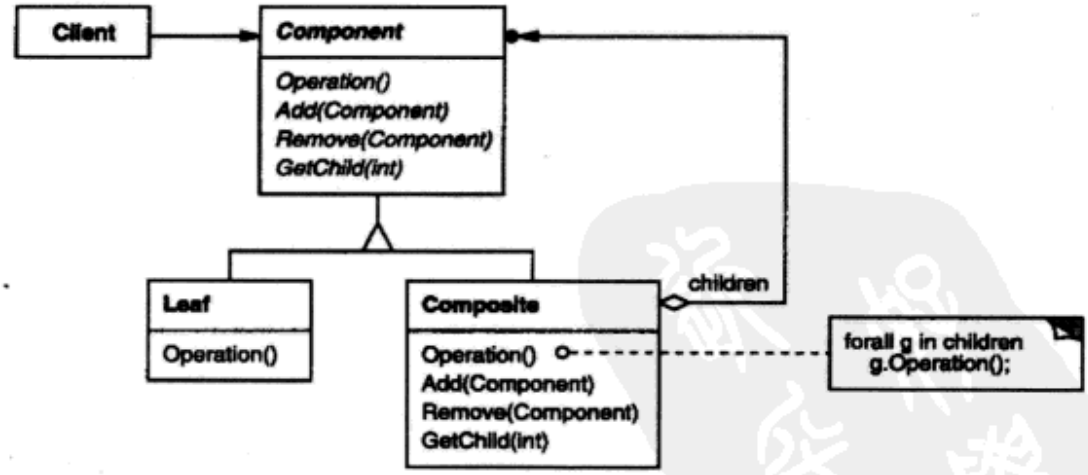

图 15.1

- Component：为组合中的对象声明接口。在适当的情况下，实现所有类共有接口的缺省行为，声明一个接口用于访问和管理 Component 的子组件。在递归结构中定义一个接口(可选)，用于访问一个父部件并在合适的情况下实现它。
- Leaf：在组合中表示叶节点对象，叶节点没有子节点，在组合中定义图元对象的行为。
- Composite：定义有子部件的那些部件的行为。存储子部件，在 Component 接口中实现与子部件有关的操作。

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，107

- Client：通过 Component 接口操纵组合部件的对象。[1]

2. 实例

本章通过一个“人事管理系统”的“员工管理、统计报表”模块为例，运用图解的方式来诠释组合模式，如图 15.2 所示。

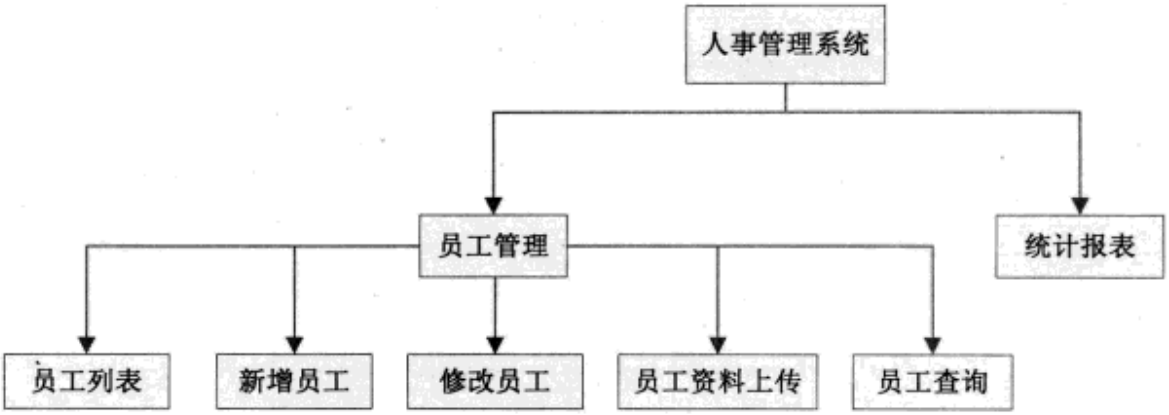

图 15.2

人事管理系统是 Component。

员工管理包括“员工列表”、“新增员工”、“修改员工”、“员工资料上传”、“员工查询”子菜单（即有内部树结构），因此其可称之为树枝节点（即 Composite）。

统计报表没有其他子菜单（即没有内部树结构），因此，它只是一种树叶节点（即 Leaf）。

以上树枝节点和树叶节点根据相同的地位共同组合成人事管理系统，这就是组合模式在实际软件开发中的一个现实案例。

15.2 应用优势与时机

组合具备以下一些优势：

- 树型结构易于增加组合对象。
- 无论分层对象是否复杂，均可清晰定义，并且可方便的添加新的构件。
- 客户端可简便的共同调用组合或单个对象。

基于此，我认为可以在以下几种情形下采用组合模式进行软件设计与实施。

- 当开发树型结构的系统时。
- 当客户要求软件系统需要展现全局与个体的关系时。
- 当系统需要共同使用组合结构的全体对象时。

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，108~109

15.3 应用情境——军官级别

当代社会，我国军官级别呈树形结构。本应用情境运用组合模式，展现排长与副排长、班长之间的关系，如图 15.3 所示。

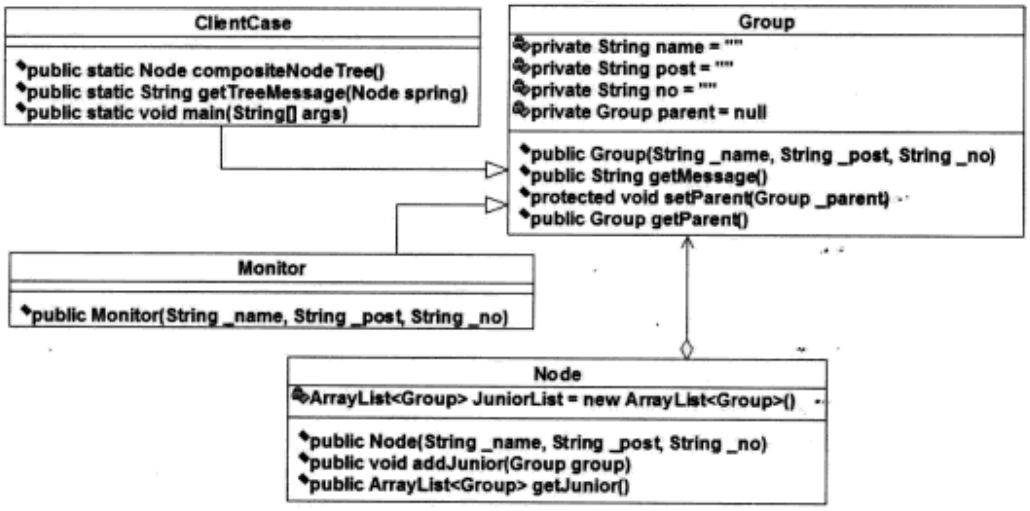

图 15.3

本应用情景的工程名为“程序 15.3.1”，源代码如下所示。

（1）排级单位抽象类

```
package model.composite;

/**
 * 定义一个军队排级单位的抽象类-即 Component
 */
public abstract class Group {
    private String name ="";    //姓名
    private String post ="";//官职
    private String no ="";//编号
    private Group parent = null;//父节点初始化
    //实际软件项目中构造函数的定义
    public Group(String _name,String _post,String _no){
        this.name = _name;
        this.post = _post;
        this.no = _no;
    }
    //获得军官信息
    public String getMessage(){
        String message = "";
        message = "姓名：" + this.name;
        message = message + "\t 官职："+ this.post;
        message = message + "\t 编号：" + this.no;
```

```
        return message;
    }
    //设置父节点
    protected void setParent(Group _parent){
        this.parent = _parent;
    }
    //获取父节点
    public Group getParent(){
        return this.parent;
    }
}
```

（2）叶子类

```
package model.composite;

/**
 * 班长-无下属军官的军官--Leaf
 */
public class Monitor extends Group {
    //构造函数
    public Monitor(String _name,String _post,String _no){
        super(_name,_post,_no);
    }
}
```

（3）节点类

```
package model.composite;

import java.util.ArrayList;
/**
 * 节点类
 */
public class Node extends Group {
    //上级军官下属包括那些下级军官和下下级军官
    ArrayList<Group> JuniorList = new ArrayList<Group>();
    //构造函数,定义参数
    public Node(String _name,String _post,String _no){
        super(_name,_post,_no);
    }
    //增加一个下属，可能是下级军官，也可能是个下下级军官
    public void addJunior(Group group) {
        group.setParent(this); //设置父节点
        this.JuniorList.add(group);
    }
    //上级拥有的下属
    public ArrayList<Group> getJunior() {
```

```
            return this.JuniorList;
        }
    }
```

（4）客户端测试类

```
package model.composite;

import java.util.ArrayList;
/**
 * 展现树型结构:
 * 一级节点是排长 platoonOfficer，二级节点是副排长 platoonSergeant，三级节点是班长
monitor
 */
public class ClientCase {
    //实现树型结构
    public static Node compositeNodeTree(){
        //一级节点：排长
        Node node = new Node("杰克","排长","A0001");//姓名，官职，编号
        //二级节点：二个副排长
        Node platoonSergeant1 = new Node("李东","副排长","A0002");
        Node platoonSergeant2 = new Node("蒋飞","副排长","A0003");
        //三级节点：班长
        Monitor A0002_1 = new Monitor("陈小小","班长","A0002_1");
        Monitor A0002_2 = new Monitor("李梅","班长","A0002_2");
        Monitor A0002_3 = new Monitor("司徒青云","班长","A0002_3");
        Monitor A0003_1 = new Monitor("金虹","班长","A0003_1");
        Monitor A0003_2 = new Monitor("麻小遇","班长","A0003_2");
        Monitor A0003_3 = new Monitor("林东悄","班长","A0003_3");
        //设置下级-------------------
        //排长下属二个副排长
        node.addJunior(platoonSergeant1);
        node.addJunior(platoonSergeant2);
        //二个副排长的下属
        platoonSergeant1.addJunior(A0002_1);
        platoonSergeant1.addJunior(A0002_2);
        platoonSergeant1.addJunior(A0002_3);
        platoonSergeant2.addJunior(A0003_1);
        platoonSergeant2.addJunior(A0003_2);
        platoonSergeant2.addJunior(A0003_3);
        return node;
    }
    //为了遍历全树,需要通过根节点，遍历出所有的节点。
    public static String getTreeMessage(Node spring){
        ArrayList<Group> juniorList = spring.getJunior();
        String message = "";
        for(Group g :juniorList){
            if(g instanceof Monitor){ //是无下属的军官就直接获得信息
                message = message+ g.getMessage()+"\n";
```

```
                }else{ //是个有下属的下级军官
                    message = message +g.getMessage() +"\n"+
getTreeMessage((Node)g);
                }
            }
            return message;
        }
        //主方法用于打印信息。
        public static void main(String[] args)
        {
            //实现树型结构
            Node platoonOfficer = compositeNodeTree();
            //一级节点-排长:
            System.out.println(platoonOfficer.getMessage());
            //所有军官信息
            System.out.println(getTreeMessage(platoonOfficer));
        }

    }
```

（5）客户 Web 页面（test.jsp）

```
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    组合模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.composite.*" %>

<br />
<br />
<h4>
<%
//实现树型结构
Node platoonOfficer = ClientCase.compositeNodeTree();
//一级节点-排长:
%>
<% out.println("我国军官级别呈树形结构:"+ platoonOfficer.getMessage());
%><br >

<%out.println( ClientCase.getTreeMessage(platoonOfficer));%><br />
</h4>
</BODY>
```

```
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
我国军官级别呈树形结构：姓名：杰克 官职：排长 编号：A0001
姓名：李东 官职：副排长 编号：A0002 姓名：陈小小 官职：班长 编号：A0002_1 姓名：李梅 官职：班长 编号：A0002_2 姓名：司徒青云 官职：班长 编号：A0002_3 姓名：蒋飞 官职：副排长 编号：A0003 姓名：金虹 官职：班长 编号：A0003_1 姓名：麻小遇 官职：班长 编号：A0003_2 姓名：林东悄 官职：班长 编号：A0003_3
```

15.4 组合模式与 Struts

目前，Struts 框架的模板标签运用了组成模式的相关原理。例如“Tiles 是表示层建筑构件。Definition 存储了一组 Attribute，把屏幕描述分离成一个具有独立标志的对象。声明一个基础屏幕 Definition，然后从这个基础 Definition 继承，创建其他 Definition，使之成为扩展类。如果修改基础屏幕 Definition。那会使继承自它的 Definition 也将改变，这样就把继承和封装的面向对象引入了动态页面中。但是主体内容不同，则只需要把标签替换为新的值。通过传递附加或者替换 Attribute，Definition 在部署时可以重载。”

此外，一般而言 Struts 框架中在未指定 Key 的情形下，在 struts-config.xml 文件中只可配置单个资源文件，为了使各个模块均具备自身的资源文件，我们可以运用组合模式解决。具体文件与程序如下所示。

（1）struts-config.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN" "http://struts.apache.org/dtds/struts-config_1_2.dtd">
    <message-resources  factory=" struts.composite. ManyInfoOriginFactory "/>
```

（2）工厂类

```
package struts.composite;

import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;
import org.apache.struts.util.MessageResources;
import org.apache.struts.util.PropertyMessageResources;
import org.apache.struts.util.PropertyMessageResourcesFactory;

public class ManyInfoOriginFactory extends PropertyMessageResourcesFactory {
    public MessageResources produceOrigin(String deploy) {
        ManyInfoOrigin manyInfo = new ManyInfoOrigin(this,
                deploy, this.returnNull);
        String[] deploys = getDeploy(deploy);
        for (int a = 0; a < deploys.length; a++) {
```

```
            MessageResources info = new PropertyMessageResources(this,
                    deploys[a], this.returnNull);
            manyInfo.createInfoOrigin(info);
        }
        return manyInfo;
    }
    private String[] getDeploy(String mDeploy) {
        if (mDeploy.equals("") )
        {
            return new String[0];
        }
        StringTokenizer stringTokenizer = new StringTokenizer(mDeploy, ", ; ");
        List fruit = new ArrayList();
        while (stringTokenizer.hasMoreTokens()) {
            fruit.add(stringTokenizer.nextToken());
        }
        return (String[]) fruit.toArray(new String[0]);
    }
}
```

（3）普通类

```
package struts.composite;

import java.util.ArrayList;
import java.util.List;
import org.apache.struts.util.MessageResources;
import org.apache.struts.util.MessageResourcesFactory;
import org.apache.struts.util.PropertyMessageResources;

public class ManyInfoOrigin extends PropertyMessageResources {
    private List infoOrigin = new ArrayList();
    public void createInfoOrigin(MessageResources mInfoOrigin) {
        infoOrigin.add(mInfoOrigin);
    }
    public ManyInfoOrigin(MessageResourcesFactory infoFactory, String deploy)
{
        this(infoFactory, deploy, false);
    }
    public ManyInfoOrigin(MessageResourcesFactory infofactory,
            String deploy, boolean isEmpty) {
        super(infofactory, deploy, isEmpty);
    }
}
```

第 16 章　Decorator（装饰）模式

16.1　概述

装饰模式别名也叫包装器，国外设计模式大师组合 GOF 把装饰模式定义为“动态地给对象添加一些额外的职责。”[1]

就其功能而言，装饰模式比生成子类更灵活。粗看其定义，大家可能有点迷惑不解。其实，我们可以把它形象化。所谓“动态地给对象添加一些额外的职责”，可理解为保持原有功能不变并进行扩展。

为了便于读者进一步理解装饰模式，本章通过模式的结构与实例进行解说。

1. 装饰模式结构（如图 16.1 所示）

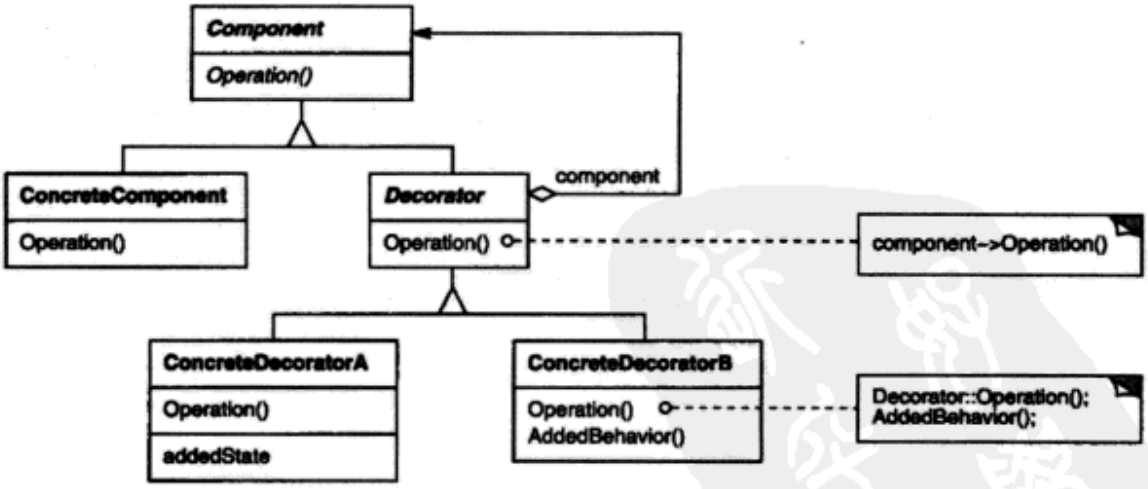

图 16.1

- Component：定义一个对象接口，可以给这些对象动态地添加职责。
- ConcreteComponent：定义一个对象，可以给这个对象添加一些职责。
- Decorator：维持一个指向 Component 对象的指针，并定义一个与 Component 接口一致的接口。
- ConcreteDecorator：向组件添加职责。[2]

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，115

[2] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，116~117

2. 实例

我们举一个实例来说明这个模式，比如，一家公司的行政助理打印 word 与 pdf 格式的文件，如图 16.2 所示。

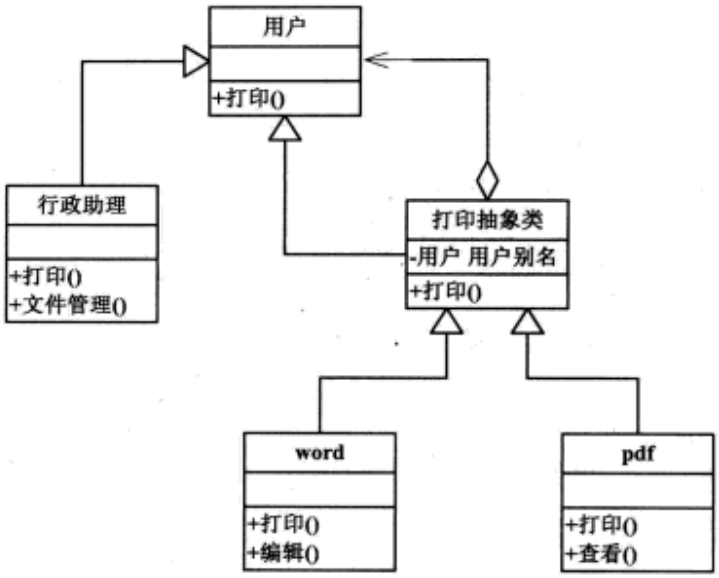

图 16.2

用户即 Component。行政助理即 ConcreteComponent。打印抽象类即 Decorator ，继承于用户类，同时用户类对象（上图中的用户别名）又成为其属性之一。word、pdf 即 ConcreteDecorator，它们分别继承打印抽象类。

16.2 应用优势与时机

装饰具备以下一些优势：

- 通过在类中移除装饰功能，从而使其更加精简。
- 提供比单独的继承、组合更强大、更灵活性的功能。
- 区别类的核心职责与装饰职责，降低了程序的耦合度。
- 无须修改组件，即可添加面向该具体组件的新具体装饰。

基于此，我认为可以在以下几种情形下采用装饰模式进行软件设计与实施。

- 当程序需要扩展现有类的功能或职责时。
- 当对象需要添加灵活性较高的功能时。
- 单独使用继承或组合存在维护复杂、代码过多等问题时。

16.3 应用情境——员工考核排名与分数设计

为了帮助市场部员工小张，了解 2010 年本人的员工考核排名与分数情况，特通过装饰者模式，进行设计。

1. 传统的方式

运用继承，进行处理，如图 16.3 所示。

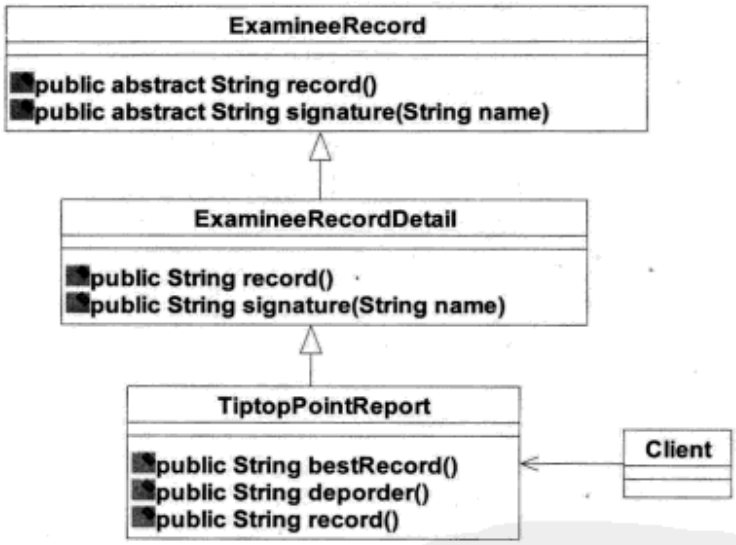

图 16.3

本应用情景的工程名为“程序 16.3.1”，源代码如下所示。

（1）考核情况抽象类

```
package model.decorator1;
/**
 * @author jianghc 考核情况
 */
public abstract class ExamineeRecord {
    //小张考核情况
    public abstract String record();
    //评价人签字
    public abstract String signature(String name);
}
```

（2）考核情况类

```
package model.decorator1;
/**
```

```
 * @author jianghc 小张考核情况
 */
public class ExamineeRecorDetail extends ExamineeRecord {
//小张考核情况
    public String record() {
        String a="员工:小张   部门: 市场 ";
        String b="考核情况:   平均分 3.5  2010 年度考核分  3.6";
        String c=" .......";
        System.out.println("员工:小张   部门: 市场");
        System.out.println("考核情况:   平均分 3.5  2010 年度考核分  3.6");
        System.out.println("  .......");
        return a+b+c;
    }
    //评价人签字
    public String signature(String name) {
        String b="评价人签字";
        System.out.println("评价人签字为: "+name);
        return b+name;
    }
}
```

（3）告知员工部门最高考核分

```
package model.decorator1;
/*
 @author jianghc
  告知员工部门最高考核分
 */
public class TiptopPointReport extends ExamineeRecorDetail {
    public String bestRecord(){
        String a="此次考核,最高平均分为 4.2, 2010 年度最高考核分为 4.1。";
      System.out.println("此次考核,最高平均分为 4.2, 2010 年度最高考核分为 4.1。");
    return a;
 }
    public String deporder(){
        String b="部门排名第 7...";
          System.out.println("部门排名第 7...");
          return b;
     }
    public String record(){
        this.bestRecord();
        super.record();
        this.deporder();
        return this.bestRecord()+super.record()+this.deporder();
    }
}
```

（4）客户端（员工查看考核情况：test.jsp）

```
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    未使用装饰模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.decorator1.*" %>

<br />
<br />
<h4>
<%
//考核情况
ExamineeRecord er = new TiptopPointReport();
%>
<% out.println("查看考核情况:"+er.record()); %><br >

<%out.println("签字:"+er.signature("李东"));%><br />
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
查看考核情况：此次考核，最高平均分为 4.2，2010 年度最高考核分为 4.1。员工：小张 部门：市场
考核情况：平均分 3.5 2010 年度考核分 3.6 .......部门排名第 7...
签字评价人签字：李东
```

（5）存在的问题

YearExamineeRecord 只有一个子类，如果子类有很多个，那么系统维护将要累死人。所以此方法不推荐。那么，我可以使用装饰模式进行解决。

2. 装饰模式方式（如图 16.4 所示）

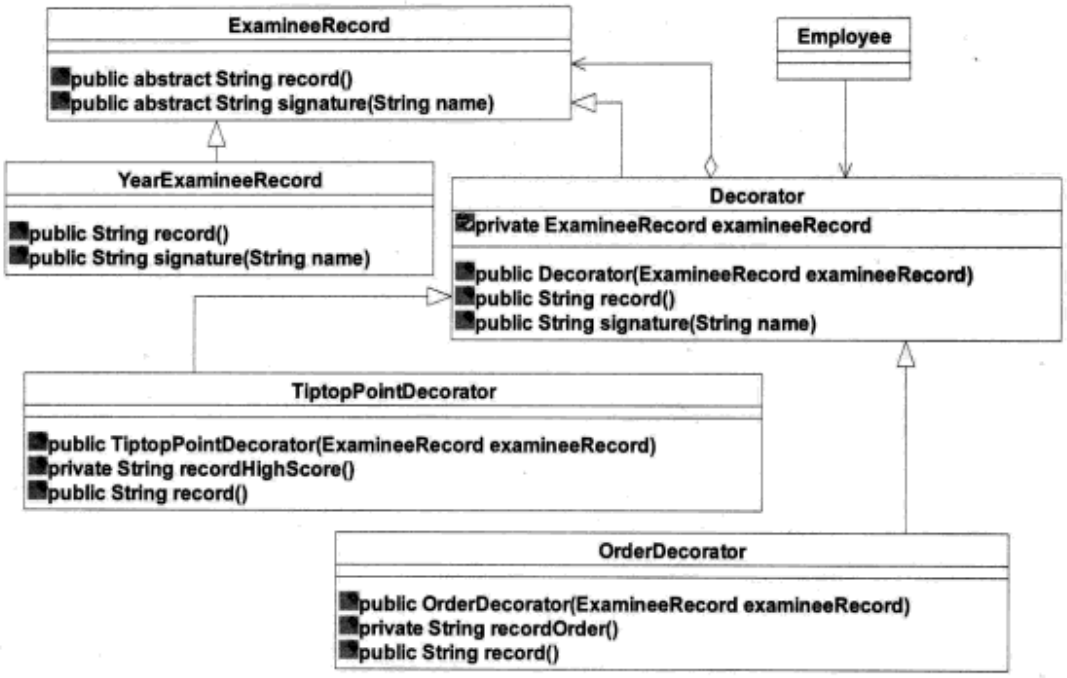

图 16.4

本应用情景的工程名为“程序 16.3.2”，源代码如下所示。

（1）考核抽象类

```
package model.decorator2;

/*
 @author jianghc
 考核报告  Component  定义一个对象接口，可以给这些对象动态地添加职责。
 考核抽象类
 */
public abstract class ExamineeRecord {
    // 考核情况显示
    public abstract String record();
    //需要评价人签字
    public abstract String signature(String name);
}
```

（2）员工考核类

```
package model.decorator2;

/*
```

```
 @author jianghc  ConcreteComponent - 定义一个对象，可以给这个对象添加一些职责。
 员工小张的考核情况，员工考核类
 */
public class YearExamineeRecord extends ExamineeRecord {
    //员工小张的考核情况
    public String record() {
        String a1="员工:小张   部门：市场";
        String a2="考核情况：  平均分 3.5  2010 年度考核分  3.6";
        String a3="  .......";
        System.out.println("员工:小张   部门：市场");
        System.out.println("考核情况：  平均分 3.5  2010 年度考核分  3.6");
        System.out.println("  .......");
        return a1+a2+a3;
    }
    //考核评价人
    public String signature(String name) {
        String a="评价人";
        System.out.println("评价人："+name);
        return a+name;
    }
}
```

（3）装饰考核报告

```
package model.decorator2;

/*
 @author jianghc
    Decorator  - 维持一个指向 Component 对象的指针，并定义一个与 Component 接口一致的
接口。
    装饰类，装饰考核报告。
 */
public abstract class Decorator extends ExamineeRecord{
    //  考核报告
    private ExamineeRecord er;
    // 用于传送考核报告的构造函数
    public Decorator(ExamineeRecord er){
        this.er = er;
    }
    // 显示考核报告
    public String record(){
        return this.er.record();
    }
    // 评价人签字
    public String signature(String name){
        return  this.er.signature(name);
    }

}
```

（4）最高考核分类装饰

```
package model.decorator2;

/*
 @author jianghc   ConcreteDecorator - 向组件添加职责
  告知员工部门最高考核分类装饰
 */
public class TiptopPointDecorator extends Decorator {
    //构造函数
    public TiptopPointDecorator(ExamineeRecord er){
        super(er);
    }
    //部门的最高考核分数情况
    private String reportHighScore(){
        String a="此次考核,最高平均分为 4.2, 2010 年度最高考核分为 4.1。";
        System.out.println("此次考核,最高平均分为 4.2,2010 年度最高考核分为 4.1。");
        return a;
    }
//部门最高考核分与员工的考核分
    public String record(){
        this.reportHighScore();
        super.record();
        return this.reportHighScore()+super.record();
    }
}
```

（5）排名装饰类

```
package model.decorator2;

/*
 @author jianghc   ConcreteDecorator- 向组件添加职责
  排名装饰
 */
public class OrderDecorator extends Decorator {
    //构造函数
    public OrderDecorator(ExamineeRecord er){
        super(er);
    }
    //员工在部门的排名情况
    private String recordOrder(){
        String a="部门排名第 7...";
        System.out.println("部门排名第 7...");
        return a;
    }
    //查看考核分数与部门排名
    public String record(){
        super.record();
```

```
        this.recordOrder();
        return super.record()+this.recordOrder();
    }
}
```

（6）客户端（Employee 即员工查看考核明细，test.jsp）

```
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    装饰模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.decorator2.*" %>

<br />
<br />
<h4>
<%
//考核情况
        // 考核报告
        ExamineeRecord er;
        er = new YearExamineeRecord();  //员工现有考核情况
        //部门考核最高分
        er = new TiptopPointDecorator(er);
        //考核排名
        er = new OrderDecorator(er);
        //考核情况显示
        //er.record();
        //er.signature("袁东");  //评价人姓名
%>
<% out.println("查看考核情况:"+er.record()); %><br >

<%out.println("签字"+er.signature("李东"));%><br />
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
查看考核情况:此次考核,最高平均分为 4.2，2010 年度最高考核分为 4.1。员工:小张 部门：市场考核情况： 平均分 3.5 2010 年度考核分 3.6 .......部门排名第 7...
签字评价人李东
```

16.4 装饰模式与 Struts

在 Struts 框架中，继承于 ActionConfig 的 ActionMapping 使业务逻辑分为几种类型。当某一 HTTP 请求到来的时候，ActionSevlet 将于 ActionMapping 的相关目录中通过 path 属性方便快捷的实现搜寻所需内容之目标。

并且，“全部的 ActionMapping 对象都是存放在一个集合中。为了更好地使用 Action, ActionMapping 对象被用来作为 Action 对象的修饰符。它将一个或者多个 URI 交给一个 Action, 同时可以根据哪个 URI 被调用来传递不同的配置信息给 Action。即 ActionMapping 可以动态赋予一个特定的 Action 对象一些附加的要求和新功能，这些就是装饰模式的要求。该设计模式相对于子类化方法扩展功能而言，具有更高的灵活性。”

此外，ActionMapping 的配置简例如下所示。

```
<action-mappings>
<action path="/user" type="com.test.UserAction">
</action>
<action path="/search" type="com.test.SearchAction">
</action>
</action-mappings>
```

第 17 章　Facade（门面）模式

17.1　概述

门面模式又称外观模式。其定义是："外部与一个子系统的通信必须通过一个统一的门面对象进行。"[1]

粗看其定义，大家可能有点迷惑不解。其实，我们可以把它理解为"一个子系统，只能对应一个具有单个实例的门面类。"

当然，一个由多个子系统组成的大系统，可以有多个门面类。

基于对定义的理解，我认为门面模式的主要目标是为了清理客户使用接口，实现对接口的精简。虽然不符合接口隔离原则，但是如果合理地使用此模式，对软件系统的设计还是蛮有好处。

为了便于读者进一步理解门面模式，本章通过模式的结构与实例进行解说。

1. 结构（如图 17.1 所示）

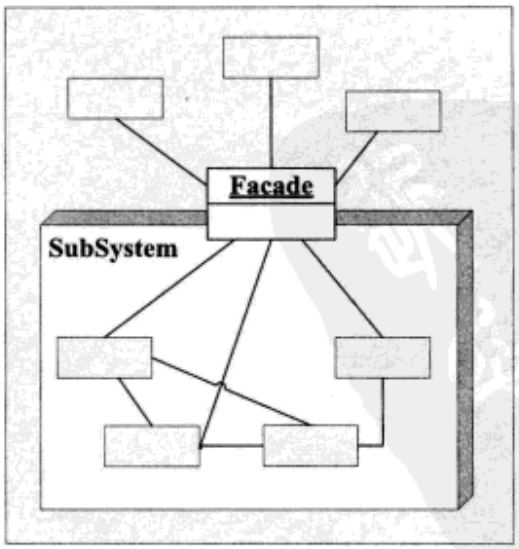

图 17.1

- 门面（Facade）角色：客户端可以调用这个角色的方法。此角色知晓相关的（一个或者多个）子系统的功能和责任。在正常情况下，本角色会将所有从客户端发来的请求委派到相应的子系统去。

[1] 阎宏. Java 与模式. 北京：电子工业出版社，2002，561

- 子系统（subsystem）角色：可以同时有一个或者多个子系统。每一个子系统都不是一个单独的类，而是一个类的集合。每一个子系统都可以被客户端直接调用，或者被门面角色调用。子系统并不知道门面的存在，对于子系统而言，门面仅仅是另外一个客户端而已。[1]

2. 实例

一家医药公司，由小变大，其应用子系统不断增加。然而，各个子系统的登录各自独立。这将给公司的管理与信息安全方面带来隐患。一个合理的规划可以是，为员工设计一个只需统一登录一次即可进入多个子系统的单点登录功能，以降低了安全的风险和管理的消耗。设计图形如图 17.2 所示。

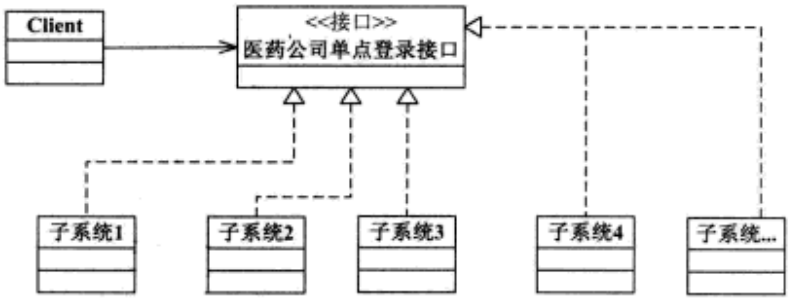

图 17.2

医药公司单点登录接口即 Façade 接口。

子系统 1、子系统 2、子系统 3、子系统 4、子系统...，即子系统。

17.2 应用优势与时机

门面模式具备以下一些优势：

- 门面模式，通过隐藏子系统的组件，降低客户处理的对象数量，以提升各个子系统的应用便捷性。
- 门面模式降低子系统与客户端程序的耦合度，促进了系统的可扩展性与可维护性。
- 门面模式通过层次化的架构方式，可提升整个系统的稳定性。

基于此，我认为可以在以下几种情形下采用门面模式进行软件设计与实施。

- 当设计者要求隐藏旧系统时，可使用门面模式的接口去实现。
- 如果子系统之间依赖性较高，可通过门面模式进行降低。
- 当子系统相当复杂时，可通过门面模式提供较简单的接口，以提高子系统的易用性与稳定性。

[1] 阎宏. Java 与模式. 北京：电子工业出版社，2002，563

17.3 应用情境——电力公司上门服务设置

用电变更、用电报修、用电增容、付费等是电力公司营销系统的各个营销业务子系统，孤寡残疾老人由于身体原因，办理此类业务非常不便。为了解决这一问题，电力公司可通过设置上门服务。由上门服务人员，为孤寡残疾老人代办用电变更、用电报修、用电增容、付费业务。孤寡残疾老人只与上门服务人员联系，如图 17.3 所示。

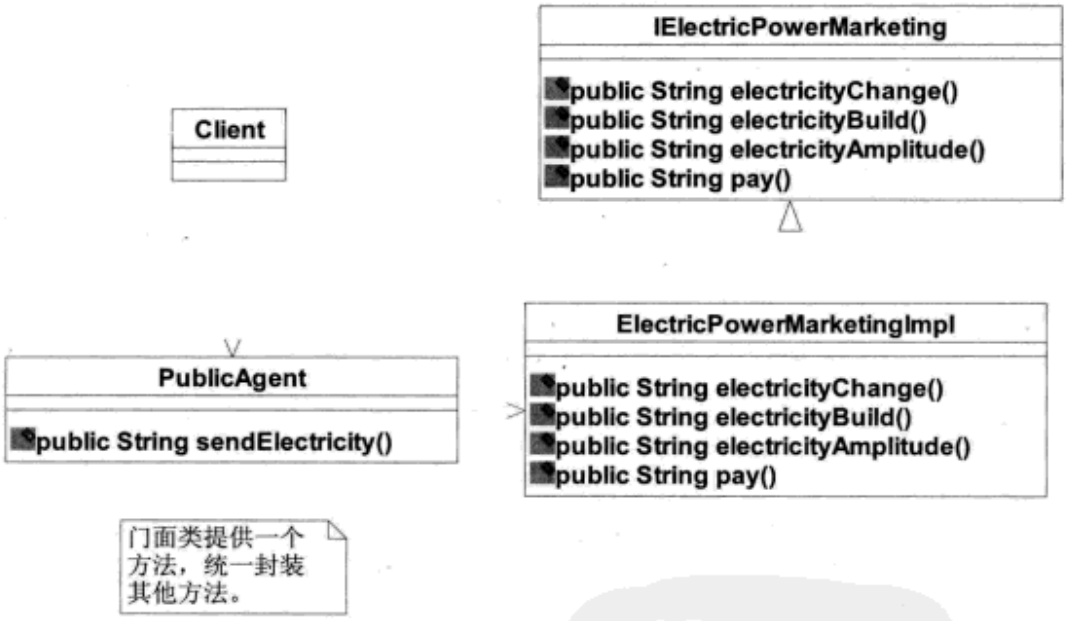

图 17.3

本应用情景的工程名为“程序 17.3.1”，源代码如下所示。

（1）电力营销业务接口

```
package model.facade;

/**
 * @author jianghc
 */
public interface IElectricPowerMarketing {
    //代办用电变更
    public String electricityChange( );
    //用电报修
    public String electricityBuild( );
    //用电增容
    public String electricityAmplitude();
    //付费
    public String pay();
}
```

（2）电力营销业务实现类

```
package model.facade;

/**
 * @author jianghc
 *///用电变更、用电报修、用电增容、付费业务
public class ElectricPowerMarketingImpl implements IElectricPowerMarketing {
    //用电变更
    public String electricityChange() {
        String a="代办用电变更....";
        System.out.println("代办用电变更....");
        return a;
    }
    //用电报修
    public String electricityBuild( ) {
        String b="代办用电报修....";
        System.out.println("代办用电报修...." );
        return b;
    }
    //用电增容
    public String electricityAmplitude() {
        String c="代办用电增容....";
        System.out.println("代办用电增容....");
        return c;
    }
    //付费
    public String pay() {
        String d="代办付费..";
        System.out.println("代办付费...");
        return d;
    }
}
```

（3）代办人（门面类）

```
package model.facade;

/**
 * @author jianghc 代办人
 */
public class PublicAgent {
    private IElectricPowerMarketing electricPowerMarketing = new
ElectricPowerMarketingImpl();

    // 上门服务人员，为孤寡残疾老人提供代办用电变更、用电报修、用电增容、付费业务一体化服
务。
    public String sendElectricity() {
        // 代办用电办更
```

```
        electricPowerMarketing.electricityChange();// 用电变更
        electricPowerMarketing.electricityBuild();// 用电报修
        electricPowerMarketing.electricityAmplitude();// 用电增容
        electricPowerMarketing.pay();// 付费
        return electricPowerMarketing.electricityChange()
                + electricPowerMarketing.electricityBuild()
                + electricPowerMarketing.electricityAmplitude()
                + electricPowerMarketing.pay();
    }
}
```

（4）客户端（test.jsp）

```
<HTML>
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    门面模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.facade.*" %>

<br />
<br />
<h4>
<%
//上门服务人员-代办人
PublicAgent publicAgent = new PublicAgent();
%>
<% out.println("上门服务:"+publicAgent.sendElectricity()); %><br >
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
上门服务：代办用电变更....代办用电报修....代办用电增容....代办付费..
```

17.4 门面模式在 Spring 与 Hibernate 中的应用

在 Spring 的 HibernateTemplate 类中，如果软件设计开发人员运用 HibernateTemplate 的 find()方法进行信息搜索，那么此行代码就可获取信息搜索返回的 List。但 find()方法其实未将以下代码进行公开：

```
Session sn = sf.openSession();
```

```
Query qe = sn.createQuery(hql);
for(int a = 0 ; a < args.length ; a++)
{
   qe.setParameter(a , Object obj)
}
qe.list();
```

从此处的代码可以看出，HibernateTemplate 类包含了 SessionFactory、Session、Query 等各种类的门面，它使客户端代码实现持久化查询时，仅需调用 HibernateTemplate 门面类的相关方法。

第 18 章　Flyweight（享元）模式

18.1　概述

享元模式是指“运用共享技术有效地支持大量细粒度的对象。”[1]

粗看其意图，大家可能有点迷惑不解。其实，我们可以把它形象化。所谓“运用共享技术有效地支持大量细粒度的对象”我们可理解为，当细粒度对象的数量过多时运行的代价相当高，此时运用共享技术可大大降低运行的代价。

为了便于读者进一步理解享元模式，本章通过模式的结构与实例进行解说。

1. 结构

（1）单纯享元模式结构

此模式，全部享元对象均可共享，如图 18.1 所示。

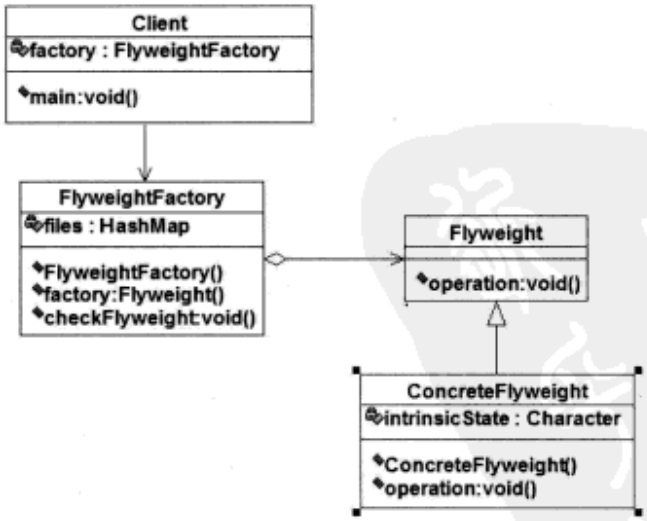

图 18.1

- 抽象享元角色（Flyweight）：为具体享元角色规定了必须实现的方法，而外蕴状态就是以参数的形式通过此方法传入。在 Java 中可以由抽象类、接口来担当。
- 具体享元角色（ConcreteFlyweight）：实现抽象角色规定的方法。如果存在内蕴状态，就负责为内蕴状态提供存储空间

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，128

- 享元工厂角色（FlyweightFactory）：负责创建和管理享元角色。要想达到共享的目的，这个角色的实现是关键！（files 可用 HashMap 或 HashTable）
- 客户端角色（Client）：维护对所有享元对象的引用，而且还需要存储对应的外蕴状态。[1]

（2）复合享元模式结构

此模式运用单纯享元，接合合成模式加以复合，如图 18.2 所示。

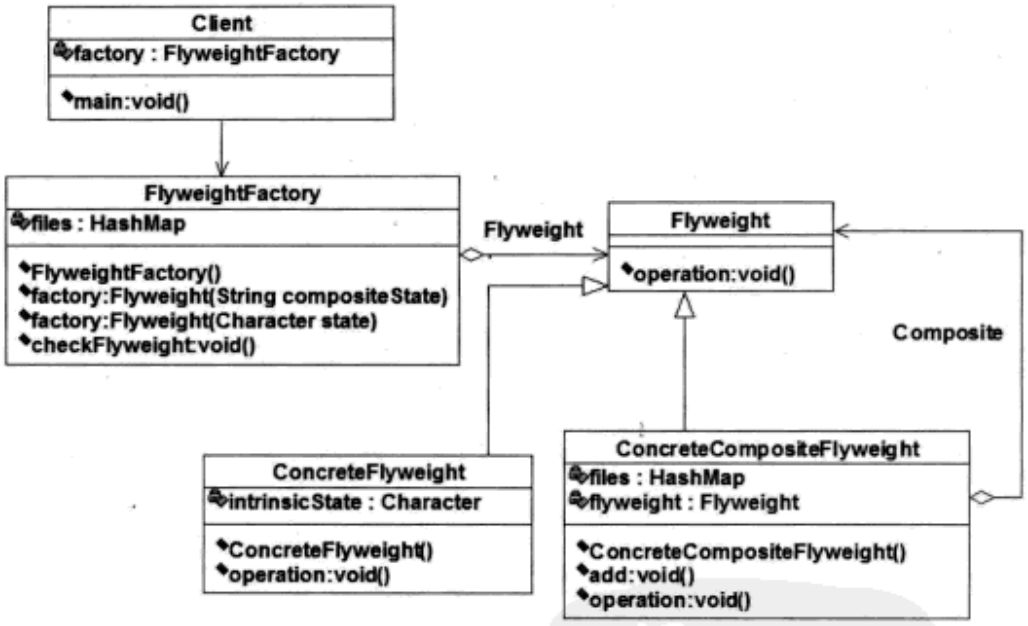

图 18.2

- 抽象享元角色（Flyweight）：它是所有的具体享元类的超类。
- 具体享元角色（ConcreteFlyweight）：实现抽象享元角色所规定的接口。
- 复合享元（UnsharableFlyweight）：其代表的对象是不可共享的，但一个复合享元对象可以分解成为多个本身是单纯享元对象的组合。比如，图 18.2 中的 ConcreteCompositeFlyweight。
- 享元工厂角色（FlyweightFactory）：负责创建与管理享元角色。
- 客户端角色（Client）：本角色需要自行存储所有享元对象的外蕴状态。[2]

2. 实例

（1）单享元模式实例

饮料店中有一系列的冰激凌用于外卖，冰激凌有内蕴状态，即冰激凌的味道；冰激凌没有环境因素，即没有外蕴状态。如果开发一个管理系统为每一个冰激凌都建立一个独立的对象，那么将产生大量的对象。此时，合理的处理方式是将冰激凌按照种类（即“味道”）划分，每一种味道的

[1] 阎宏. Java 与模式. 北京：电子工业出版社，2002，528~529
[2] 阎宏. Java 与模式. 北京：电子工业出版社，2002，532~533

冰激凌只创建一个对象并实行共享。所谓共享，主要是指冰激凌味道的共享，制造方法的共享等。因此，享元模式对饮料店而言，无须为每一份冰激凌单独调制。当有需要时，店主可一次性地调制出足够当天使用的某风味冰激凌。

基于此，运用单享元模式设计如图 18.3 所示。

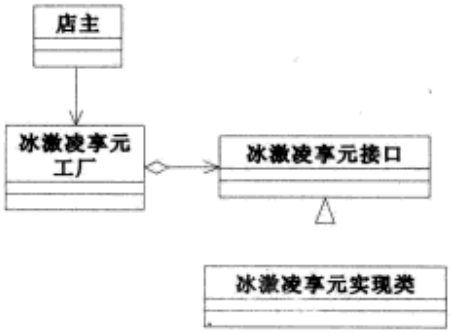

图 18.3

店主即客户端角色，冰激凌享元工厂即享元工厂角色，冰激凌享元接口即抽象享元角色，冰激凌享元实现类即具体享元角色。

（2）多享元模式实例

饮料店中的冰激凌仅用于外卖，客户有意见并提出希望在店中有相应的桌子用来消费冰激凌。此时，冰激凌的桌子即为外蕴状态。此时，单享元模式需要扩展为多享元模式。

基于此，运用多享元模式设计如图 18.4 所示。

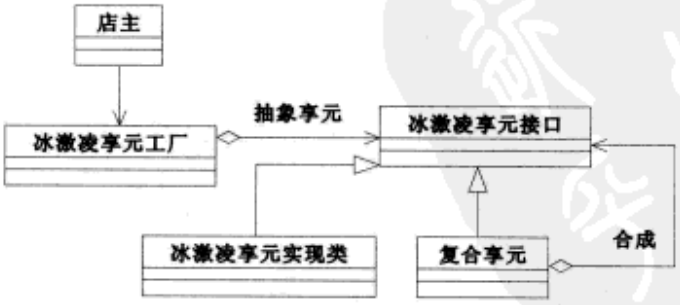

图 18.4

店主即客户端角色，冰激凌享元工厂即享元工厂角色，冰激凌享元接口即抽象享元角色，冰激凌享元实现类即具体享元角色，复合享元即复合享元角色。

18.2 应用优势与时机

享元模式的优势在于：通过减少内存对象的数量，节省内存空间。基于此，我认为可以在以下几种情形下采用享元模式进行软件设计与实施。

- 软件系统中对象过多时。
- 软件系统中的对象花费内存过多时。
- 软件系统的对象状态，其绝大部分可外部化时。
- 根据对象状态进行分类时，在去掉外部状态时，每个分类均可用一个对象取代。
- 软件系统无须依靠对象的标志。

18.3 应用情境——word 文档字体样式的修改

当我们对 word 文档的文字字体进行编辑时，可通过对字体样式的修改阐述单享元模式与多享元模式的区别与理念。

1. 单享元模式（如图 18.5 所示）

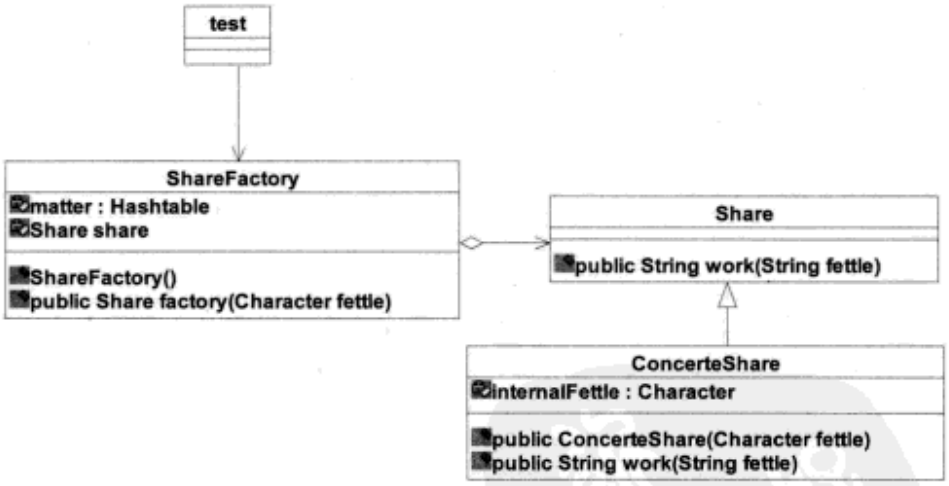

图 18.5

本应用情景的工程名为“程序 18.3.1”，源代码如下所示。

（1）享元工厂角色类

```
package  model.flyweight1;

import java.util.Hashtable;
//属于 FlyweightFactory 角色
public class ShareFactory {
    private Hashtable matter = new Hashtable();
    private Share share;
    public ShareFactory(){

    }
    //式样
    public Share factory(Character fettle){
        if(matter.containsKey(fettle)){
```

```
            return (Share)matter.get(fettle);
        }else{
         share = new ConcerteShare(fettle);
         matter.put(fettle, share);
            return share;
        }
    }
}
```

（2）抽象享元角色接口

```
package model.flyweight1;
//属于 Flyweight 角色
public interface Share {
    public String work(String fettle);
}
```

（3）具体享元角色类

```
package model.flyweight1;

//属于 ConcerteFlyweight 角色
public class ConcerteShare implements Share{
    //内蕴状态
    private Character internalFettle = null;
    //外蕴状态参数传进
    public ConcerteShare(Character fettle) {
        this.internalFettle = fettle;
    }
    public String work(String fettle) {
        String a="internalFettle="+internalFettle+",Exterior Fettle=";
        System.out.println("internalFettle="+internalFettle+",Exterior
Fettle="+fettle);
        return a+fettle;
    }
}
```

（4）客户角色类（test.jsp）

```
//属于 Client 角色
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    单享元模式
    </TITLE>
```

```
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.flyweight1.*" %>

<br />
<br />
<h4>
<%
ShareFactory factory=new ShareFactory();
Share share=factory.factory(new Character('中'));
%>
<% out.println("字体:"+share.work("黑体")); %><br >
<% out.println("字体:"+share.work("宋体")); %><br >
<%
share=factory.factory(new Character('外'));
out.println("字体:"+share.work("仿宋"));%><br />
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
字体: internalFettle=中，Exterior Fettle=黑体
字体: internalFettle=中，Exterior Fettle=宋体
字体: internalFettle=外，Exterior Fettle=仿宋
```

显示结果证明，内蕴即汉字“中”共享一次，外蕴即字体不断改变。

2. 多享元模式（如图 18.6 所示）

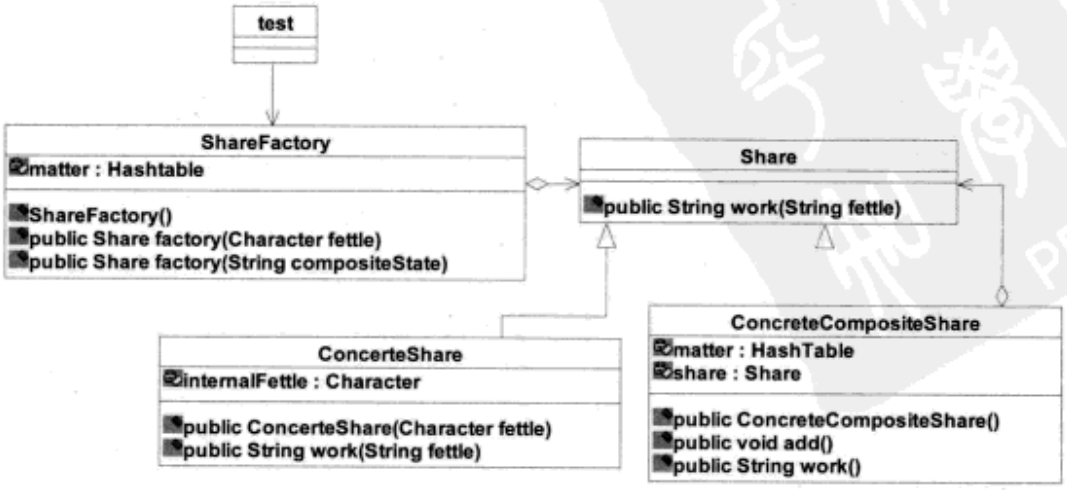

图 18.6

本应用情景的工程名为“程序 18.3.2”，源代码如下所示。

（1）享元工厂角色类

```
package model.flyweight2;

import java.util.Hashtable;
//FlyweightFactory 角色
public class ShareFactory
{
  private Hashtable matter=new Hashtable();
  public ShareFactory() {  }
  //单纯享元工厂方法，所需状态以参量形式传入
  public Share factory(Character fettle)
  {
    if(matter.containsKey(fettle))
    {
      return (Share)matter.get(fettle);
    } else {
      Share share=new ConcreteShare(fettle);
      matter.put(fettle,share);
      return share;
    }
  }
  //享元工厂方法,运用 String 参量传入状态
  public Share factory(String compositeState)
  {
    ConcreteCompositeShare compositeShare=new ConcreteCompositeShare();
    int length=compositeState.length();
    Character fettle=null;
    for(int i=0;i<length;i++)
    {
      fettle=new Character(compositeState.charAt(i));
      System.out.println("factory("+fettle+")");
      compositeShare.add(fettle,this.factory(fettle));
    }
    return compositeShare;
  }
}
```

（2）抽象享元角色接口

```
package model.flyweight2;
//属于 Flyweight 角色
public interface Share {
    public String work(String fettle);
    }
```

（3）具体享元角色类

```
package model.flyweight2;
//属于 ConcerteFlyweight 角色
public class ConcreteShare implements Share
    {
     private Character intrinsicFettle=null;
     //传入内蕴状态作为参数
    public ConcreteShare(Character fettle)
    {
      this.intrinsicFettle=fettle;
     }
     //传入外蕴状态
     public String work(String fellte)
     {
    String work="Intrinsic fellte="+intrinsicFettle+",Exterior fellte=";
      System.out.println("Intrinsic    fellte="+intrinsicFettle+",Exterior
fellte="+fellte);
      return work+fellte;
     }
    }
```

（4）复合享元角色

```
package model.flyweight2;

import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;
//UnsharableFlyweight 角色
public class ConcreteCompositeShare implements Share{
        private Hashtable matter=new Hashtable(10);
        private Share share;
        public ConcreteCompositeShare() { }
        //添加一个新的单纯享元对象到聚集中
        public void add(Character key,Share share)
        {
         matter.put(key,share);
        }
        //传入外蕴状态
        public String work(String exteriorFettle)
        {
          Share share=null;
          for(Iterator it=matter.entrySet().iterator();it.hasNext();)
          {
            Map.Entry e=(Map.Entry)it.next();
            share=(Share)e.getValue();
            share.work(exteriorFettle); //外蕴状态
          }
```

```
            return exteriorFettle;
        }
    }
```

（5）客户角色类（test.jsp）

```
//属于Client角色
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    多享元模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.flyweight2.*" %>

<br />
<br />
<h4>
<%
ShareFactory factory=new ShareFactory();
Share share=(Share) factory.factory("东西南");
//share.work("宋体");
%>

<% out.println("内蕴东西南不同，外蕴均为字体:"+share.work("宋体")); %><br >
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
内蕴东西南不同，外蕴均为字体：宋体
```

此结果证明，复合享元对象分成若干个单纯享元对象处理，其外蕴是相同的。

18.4 享元模式与 Struts

回顾以往开发的 Struts 项目，笔者认为 Struts1 中的 Action 类体现了共享模式的相关原理。

例如，我们在 Action 中声明了字段并设置了相关数值，如果初始值为 88，当客户端有人访问页面将值设为 77，那么第 2 个使用者调用此字段时，得到的结果将是 88，而不是修改后的 77。

原因在于，Action 声明字段（属性）对于全体请求均是共享的。当然此处的共享模式应用的是否合适，这个要结合项目需求。如果，项目需要一个永不改变的数值，那么此模式在这里就比较

合理。如果项目需要对字段进行数值修改，我们也有解决方法。

比如，我们可以通过创建 RequestProcessor 的子类 TestRequestProcessor，将 Struts 现有 RequestProcessor 类 processActionCreate()中的以下部分删除。

```
instance = (Action) actions.get(className);
if (instance != null) {
    if (log.isTraceEnabled()) {
        log.trace("  Returning existing Action instance");
    }
    return (instance);
}
```

剩下所有代码均放置于 TestRequestProcessor 的 processActionCreate()中即可。

第 19 章　Proxy（代理）模式

19.1　概述

代理模式是指："为其他对象提供一种代理以控制对这个对象的访问。"[1]

粗看其定义，大家可能有点迷惑不解。其实，我们可以把它简化。所谓"为其他对象提供一种代理以控制对这个对象的访问"可理解为"代理对象去除无须了解的服务，实现客户与目标对象之间的关联。

为了便于读者进一步理解代理模式，本章通过一个结构和实例进行解说。

1. 结构（如图 19.1 所示）

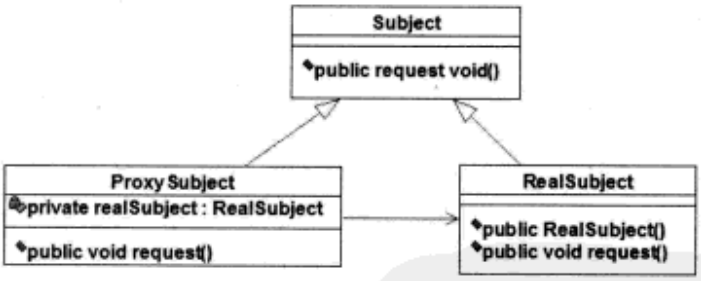

图 19.1

- 抽象主题角色（Subject）：声明了代理主题和真实主题的公共接口，使任何需要真实主题的地方都能用代理主题代替。
- 代理主题角色（ProxySubject）：含有真实主题的引用，从而可以在任何时候操作真实主题对象，代理主题角色提供和真实主题相同的接口，使它可以随时代替真实主题。代理主题通过持有真实主题的引用，不仅可控制真实主题的创建或删除，还可在真实主题被调用前进行拦截，或在调用后进行某些操作。
- 真实代理对象（RelSubject）：定义了代理角色所代表的具体对象。[2]

2. 实例

当前，房地产行业是一个充满着代理服务的行业。一般房产商开盘出售新房都是由代理公司帮忙销售。由此，房产商、代理公司、买房人构成了一个代理模式的关系。

基于此，运用代理模式设计如图 19.2 所示。

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，137

[2] 阎宏. Java 与模式. 北京：电子工业出版社，2002，498~499

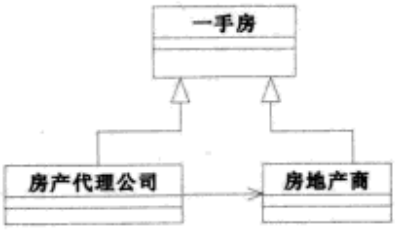

图 19.2

一手房即抽象主题角色（Subject）：声明了房产代理主题和房地产商真实主题的公共接口。

房产代理公司即代理主题角色（ProxySubject），房地产商即真实代理对象，因为它让房产代理公司代理销售一手房。

19.2 应用优势与时机

代理模式具备以下一些优势：

- 运用中介的方式，关联调用方与被调方，有利于减轻系统的耦合度。
- 无须修改现有代码，即可对需要的一个对象进行控制与访问。

基于此，我认为可以在以下几种情形下采用代理模式进行软件设计与实施。

- 调用程序的原有方法，其已有方法需要修改或优化时。
- 客户端的访问权限受到限制，其用户的权限需要验证时。
- 计算程序较复杂，运算的时间较长并且在计算进程中需要展现中间的运算结果时。

19.3 应用情境——员工知识培训遇到的事

为了进一步加强员工的业务知识，某公司特开展了员工知识培训讲座。其中，上级主管领导对参与人员的数量进行了分配。但是，业务分析师李东被客户临时叫走。因而，需要一个人去代替参与会议。此时运用代理模式，设计如图 19.3 所示。

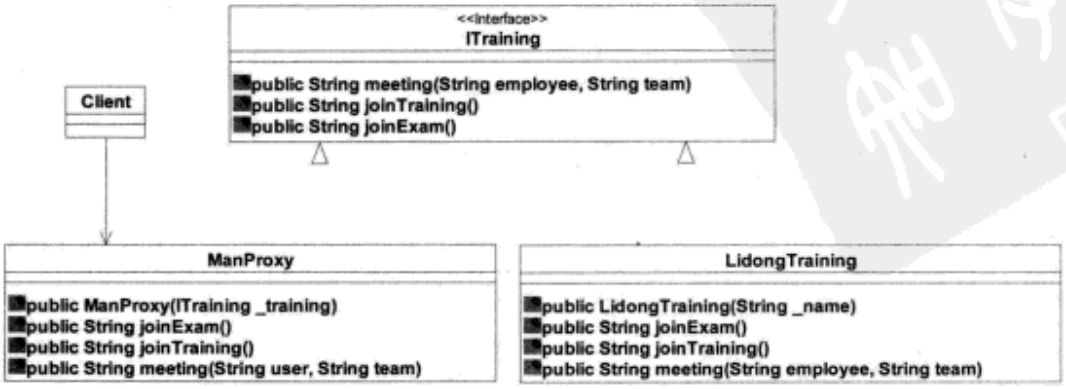

图 19.3

本应用情景的工程名为“程序 19.3.1”，源代码如下所示。

（1）抽象主题角色

```
package model.proxy;
/**  抽象主题
 * @author jackjiang
 */
public interface ITraining {
    //进入培训讲座会议室
    public String meeting(String employee,String team);
    //参与培训
    public String joinTraining();
   //参与考试
    public String joinExam();
}
```

（2）代理主题角色

```
package model.proxy;

/** 代理类
 * @author jackjiang 代替开会者
 */
public class ManProxy implements ITraining {
    private ITraining training = null;
    //构造函数传递谁代替李东参加培训
    public ManProxy(ITraining _training){
        this.training = _training;
    }
    public String joinExam() {//参与培训
        return this.training.joinExam();
    }
    public String joinTraining() {//参与考试
        return this.training.joinTraining();
    }
    public String meeting(String user, String team) {
        return this.training.meeting(user, team);
    }
}
```

（3）真实代理对象

```
package model.proxy;

/**  真实实现类
 * @author jackjiang
 */
```

```
public class LidongTraining implements ITraining {
    private String name = "";
    //通过构造函数传递名称
    public LidongTraining(String _name){
        this.name = _name;
    }
    public String joinExam() {
        String joinExam="在培训！";
        System.out.println(this.name + "在培训！");
        return this.name+joinExam;
    }
    //参与考试
    public String joinTraining() {
        String joinTraining="参与考试！";
        System.out.println(this.name + " 参与考试！");
        return this.name+joinTraining;
    }
    //进入培训讲座会议室，前提
    public String meeting(String employee, String team) {
        String a="参会者名为";
        String b="的员工";
        String c="进入会议室成功！";
        System.out.println("参会者名为"+employee + " 的员工 " + this.name + "
进入会议室成功！");
        return a+employee+b+ this.name +c;
    }
}
```

（4）客户端（test.jsp）

```
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    代理模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.proxy.*" %>

<br />
<br />
<h4>
<%
//定义参会者 lidong
ITraining training = new LidongTraining("李东");
```

```
//然后再定义一个代练者
ITraining proxy = new ManProxy(training);
//开始培训，记下时间戳
%>
<% out.println("开始时间：2011-8-2 13:00"); %><br >
<%out.println(proxy.meeting("lidong", "team"));//参加培训会的人
out.println(proxy.joinTraining());//参加培训
out.println(proxy.joinExam());//考试
 %><br >
<% out.println("结束时间：2011-8-2 15:30"); %><br >
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
开始时间：2011-8-2 13:00
参会者名为 lidong 的员工李东进入会议室成功！李东参与考试！李东在培训！
结束时间：2011-8-2 15:30
```

19.4 代理模式与适配器模式

表 19.1 代理模式和适配器模式的相同与区别之处

模式名	代理	适配器
是否替一个对象提供间接性质的访问	是	是
接口	实现和目标对象相同的接口	主要用来处理接口间不匹配的问题，它往往替所适配的对象提供一个不同的接口。

19.5 代理模式与装饰模式

表 19.2 代理模式和装饰模式的相同与区别之处

模式名	代理	装饰
是否需要为转调其他对象的前后，执行一定的功能	是	是
目的	主要目的是控制对对象的访问	动态的为某个类型添加新的职责，也就是说为了动态的增加功能。

19.6 代理模式在 Spring 中的 AOP 实现

代理模式分为静态代理和动态代理。

静态代理类：由程序员创建或由特定工具自动生成源代码，再对其编译。在程序运行前，代

理类的.class 文件就已经存在了。动态代理类：在程序运行时，运用反射机制动态创建而成。

本文运用动态代理对 Spring 框架的 AOP 进行实现。具体实例可见以下代码：

（1）动态代理类

```
package spring.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class DynaProxyHi implements InvocationHandler {
    //需要处理的相关对象,例如 Hi
    private Object delegate;
    //可以动态的创建方法被处理之后的有关对象
    public Object bind(Object delegate) {
        this.delegate = delegate;
        return Proxy.newProxyInstance(
                this.delegate.getClass().getClassLoader(), this.delegate
                        .getClass().getInterfaces(), this);
    }
    //需要处理对象的各个方法均由此方法交与 JVM 调用。（动态方法）
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        Object fruit = null;
        try {
            //本条语句交于 JVM 执行原有方法（即反射机制）
            fruit = method.invoke(this.delegate, args);
        } catch (Exception e) {
            e.printStackTrace();
        }
        //将返回方法与返回值交于调用方
        return fruit;
    }
}
```

（2）普通接口

```
public interface IHi {
    public String info1(String A);
    public String info2(String B);
}
```

（3）普通接口实现类

```
package spring.proxy;

public class Hi implements IHi{
    public String info1(String A) {
        return A;
```

```
    }
    public String info2(String B) {
        return B;
    }
}
```

（4）客户端

```
package spring.proxy;

public class Client {
        public static void main(String[] args) {
            IHi hi = (IHi)new DynaProxyHi().bind(new Hi());
            System.out.print(hi.info1("A"));
            System.out.print(hi.info2("B"));
        }
    }
```

显示结果：

```
AB
```

第五部分

设计行为派
——细说行为型模式

第 20 章　Chain of Responsibility

（责任链）模式

20.1　概述

责任链模式是指："使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。"[1]

粗看其定义，大家可能有点迷惑不解。其实，我们可以把它理解为"为多个对象提供办理同一请求的时机，以实现发送者与接受者的解耦。并且，多个对象组成的责任链，在办理客户的请求时，自身能处理则处理，如果不能处理则必须传递至下一个对象，直至解决为止。"

为了便于读者进一步理解责任链模式，本章通过模式的结构与实例进行解说。

1. 结构（如图 20.1 所示）

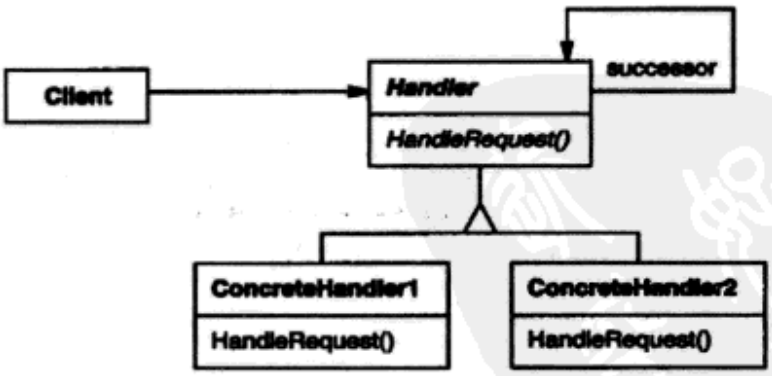

图 20.1

- 处理者（Handler）：定义一个处理请求的接口。（接口或抽象类）实现后继链（可选）。
- 具体处理者（ConcreteHandler）：处理它所负责的请求，可访问它的后继者。如果可处理该请求，就处理之；否则将该请求转发给它的后继者。
- Client：向链上的具体处理者对象提交请求。[2]

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，147

[2] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，149~150

2. 实例

粗看一下责任链模式，不禁联想到当前社会的有些不良现象。

回想 2005 年某日，我在浙江省某地级市民政局进行需求调研时，有一个 70 多岁的老大爷，拄着拐杖进入处长办公室。一见到处长，老大爷就大声诉说委屈："自己年纪已大，家中唯一的养子不愿赡养他。村委会又不愿为其提供低保，如要办理低保，让他去找镇民政所。去了镇民政所，镇民政所的工作人员又让他去找县民政局。去了县民政局，县民政局的工作人员又让他去找地区民政局。"听完老大爷的讲诉后，处长让他去县民政局申请低保。在老大爷离开后，处长说了句话："这不是我的职责范围。"

最后，老大爷跑了无数部门，但是全部无果而终，如图 20.2 所示。

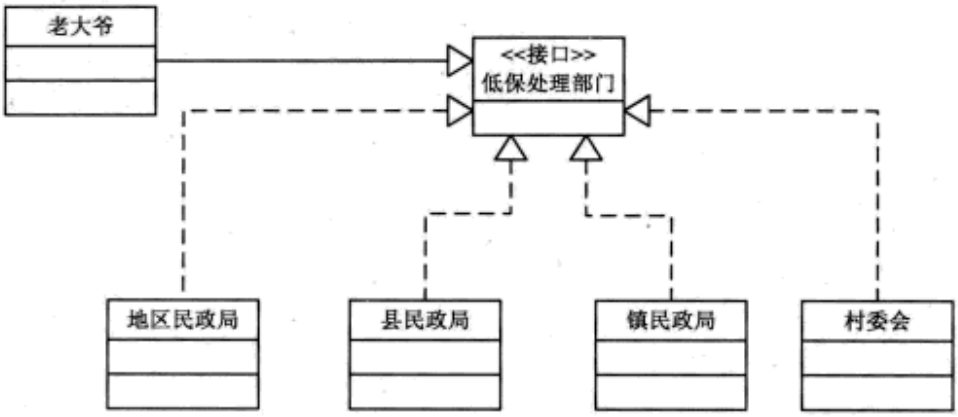

图 20.2

责任链模式就是此类"推卸"责任的模式，你的事情，在我这能处理掉就处理，处理有麻烦就推给其他对象。

20.2 优势与时机

责任链模式具备以下一些优势：

- 责任链的所有对象只能与后面的对象发生耦合度较低的关联，如此一来，编制处理者与责任链程序则显得相当简单。
- 责任链模式：分离了请求与处理，使客户无须了解自己的请求为何对象处理。
- 责任链模式：促使应用程序可灵活地更改处理对象的前后次序，并且可灵活地更改处理对象的职责。

基于此，我认为可以在以下几种情形下采用责任链模式进行软件设计与实施。

- 多个对象均可处理单个请求，并且处理该请求的对象在运行时自动定位。
- 客户无须指定特点接收者，即可对众多对象发送请求。
- 动态确定，可处理单个请求的对象群。

20.3 提升方向

需要合理的把握性能，由于责任链从链头遍历到链尾，如果链过长，则会导致程序性能降低。调试并不快捷，如果链条较长，过程较多，调试时的逻辑会相当繁琐。客户提交的请求未必一定能处理，当请求在处理链结束点时程序无法处理。

20.4 应用情境——知识平台的权限设计

某地的一个软件公司，其知识平台的菜单访问权限由业务经理、组长、配置管理员进行管理。除程序员可访问菜单，其他人员均无权访问。

其中，业务经理可授与程序员查看菜单编号为 50 及以上的内容；组长可授与程序员查看菜单编号为 50 以内的内容；配置管理员可授与程序员查看菜单编号为 30 以内的内容。

针对此情境，我们系统设计师可运用责任链模式进行设计，如图 20.3 所示。

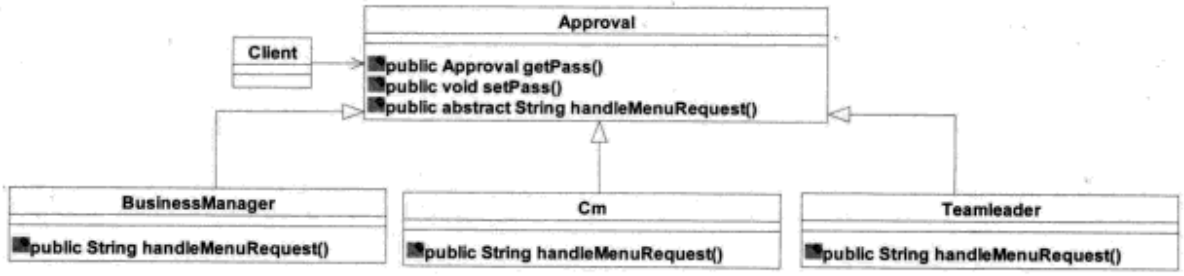

图 20.3

本应用情景的工程名为“程序 20.4.1”，源代码如下所示。

（1）审批（处理者抽象类）

```
package model.responsibility;
//approval 审批 抽象处理者(Handler)角色
public abstract class Approval {
    /**
     *拥有处理请求的下一个对象
     */
    protected Approval pass = null;
    /**
     * 取值方法
     */
    public Approval getPass() {
        return pass;
    }
    /**
     *  为下一个处理请求的对象进行设置
     */
    public void setPass(Approval pass) {
```

```
        this.pass = pass;
    }
    /**
     * 处理菜单查看的申请
     * @param person    申请人
     * @param workId    菜单编号
     * @return          充许或不充许的情况
     */
    public abstract String handleMenuRequest(String person , int menuId);
}
```

（2）业务经理（处理者实现类）

```
package model.responsibility;
//具体处理者(ConcreteHandler)角色：具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。
//由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。
public class BusinessManager extends Approval {
    public String handleMenuRequest(String person, int menuId) {
        String str = "";
        //业务经理审批>=50 编号的菜单
        if(menuId >= 50)
        {
            //只充许程序员
            if("程序员".equals(person))
            {
                str = "充许：业务经理授与(" + person + ")查看菜单内容，菜单编号为" + menuId + "";
            }else
            {
                //不充许其他人
                str = "不充许：业务经理不授与(" + person + ")查看菜单内容，菜单编号为" + menuId + "";
            }
        }else
        {
            //如有后继处理对象，继续传递
            if(getPass() != null)
            {
                return getPass().handleMenuRequest(person, menuId);
            }
        }
        return str;
    }
}
```

（3）配置管理员（处理者实现类）

```
package model.responsibility;
//具体处理者(ConcreteHandler)角色：具体处理者接到请求后，可以选择将请求处理掉，或者将请
```

```
求传给下家。
    //由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。
    public class Cm extends Approval {
        public String handleMenuRequest(String person, int menuId) {
            String str = "";
            //配置管理员充许查看 30 以内的菜单
            if(menuId < 30)
            {
                //只充许程序员
                if("程序员".equals(person))
                {
                    str = "充许：配置管理员授与(" + person + ")查看菜单内容，菜单编号为" +
menuId + "";
                }else
                {
                    //不充许其他人
                    str = "不充许：配置管理员不授与(" + person + ")查看菜单内容，菜单编号
为" + menuId + "";
                }
            }else
            {
                //超过 30，传给上一级处理
                if(getPass() != null)
                {
                    return getPass().handleMenuRequest(person, menuId);
                }
            }
            return str;
        }
    }
```

（4）组长（处理者实现类）

```
    package model.responsibility;
    //具体处理者(ConcreteHandler)角色：具体处理者接到请求后，可以选择将请求处理掉，或者将请
求传给下家。
    //由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。
    public class Teamleader extends Approval {
        public String handleMenuRequest(String person, int menuId) {
            String str = "";
            //组长只审批 50 之内的菜单查看权
            if(menuId < 50)
            {
                //充许程序员
                if("程序员".equals(person))
                {
                    str = "充许：组长授与(" + person + ")查看菜单内容，菜单编号为" + menuId
+ " ";
                }else
                {
```

```
                //不充许他人
                str = "不充许：组长不授与(" + person + ")查看菜单内容，菜单编号为" +
menuId + " ";
            }
        }else
        {
            //超过 50，继续传递给更高层解决
            if(getPass() != null)
            {
                return getPass().handleMenuRequest(person, menuId);
            }
        }
        return str;
    }
}
```

（5）客户端（test.jsp）

```
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    责任链模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.responsibility.*"
%>

<br />
<br />
<h4>
<%
    //先要组装责任链
Approval a1 = new BusinessManager();
Approval a2 = new Teamleader();
Approval a3 = new Cm();
a3.setPass(a2);
a2.setPass(a1);
%>
<%
//测试
String one = a3.handleMenuRequest("程序员", 29);
out.println("one = " + one);    %><br >
<%
//测试
String two = a3.handleMenuRequest("业务员", 29);
```

```
out.println("two = " + two);%><br >
<%
//测试
String three = a3.handleMenuRequest("程序员", 40);
out.println("three = " + three);    %><br >
<%
//测试
String four = a3.handleMenuRequest("业务员", 40);
out.println("four = " + four);      %><br >
<%
//测试
String five = a3.handleMenuRequest("程序员", 55);
out.println("five = " + five); %><br >
<%
//测试
String six = a3.handleMenuRequest("业务员", 55);
out.println("six = " + six);    %><br >
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
one = 充许：配置管理员授与(程序员)查看菜单内容，菜单编号为 29
two = 不充许：配置管理员不授与(业务员)查看菜单内容，菜单编号为 29
three = 充许：组长授与(程序员)查看菜单内容，菜单编号为 40
four = 不充许：组长不授与(业务员)查看菜单内容，菜单编号为 40
five = 充许：业务经理授与(程序员)查看菜单内容，菜单编号为 55
six = 不充许：业务经理不授与(业务员)查看菜单内容，菜单编号为 55
```

20.5 责任链模式与 Struts

Struts2 框架的拦截器结构所进行的相关设计，体现了责任链模式的具体实现。

首先，Struts2 拦截器结构类似于某种堆栈。当一个请求发起时，一个堆栈执行多个拦截器和 Action。

其次，拦截器结构中 Action 被放置于堆栈的底部。并且，堆栈一向所奉行的“先进后出”原则将使多个拦截器取出来先执行，才能使 Action 再取出来执行。如此一来，必然形成了某种递归性质的调用关系。

第 21 章　Command（命令）模式

21.1　概述

命令模式是指："将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。"[1]

粗看其定义，大家可能有点迷惑不解。其实，我们可以把它理解为"一个命令即一个操作，请求方与接收方各自独立操作，请求方无须了解接收方的接口以及执行细节。"

为了便于读者进一步理解命令模式，本章通过模式的结构与实例进行解说。

1. 命令模式结构（如图 21.1 所示）

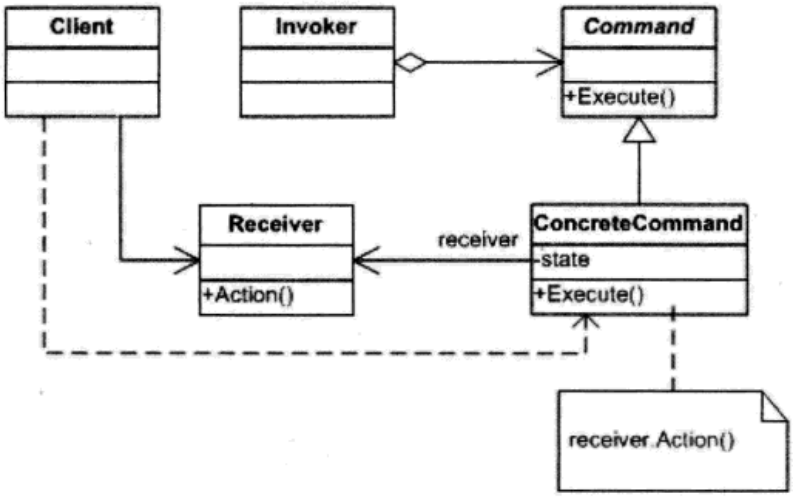

图 21.1

- Command（命令）：声明执行操作的接口。
- ConcreteCommand（命令实现）：将一个接收者对象绑定于一个动作。调用接收者相应的操作，以实现 Execute。
- Client：创建一个具体命令对象并设定它的接收者。
- Invoker（调用者）：要求该命令执行这个请求。
- Receiver（做事的人）：知道如何实施与执行一个请求相关的操作。任何类都可能作为一个接收者。[2]

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，154~155

[2] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，157

2. 实例

最近十来年，我国房地产事业发展的十分迅猛。作为房地产行业一份子的建筑公司，其整个运作过程就是一个命令模式的流程。

比如，总经理下达一个造房命令，部门经理接到命令后，再指派给下属工程队长。这样的需求，根据命令模式进行设计的类图如图 21.2 所示。

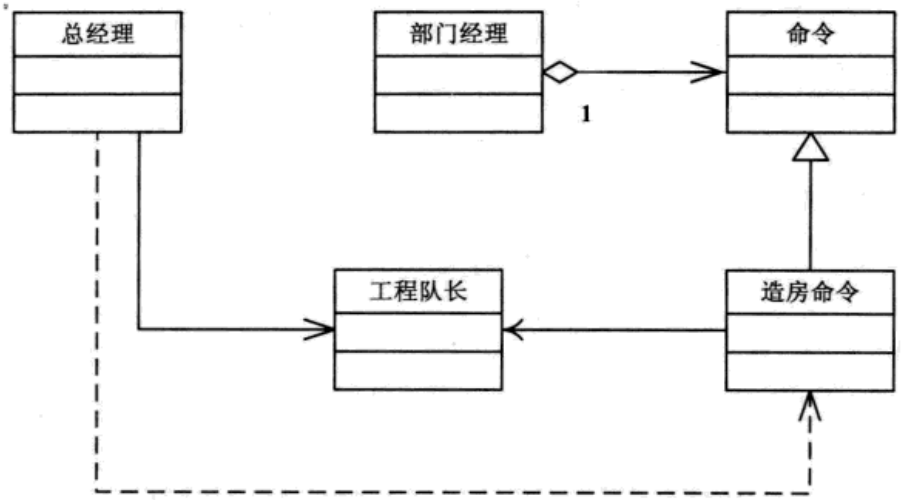

图 21.2

总经理即 client，部门经理即 Invoker（调用人），命令即 Command（命令），造房命令即 ConcreteCommand（命令实现），工程队长即 Receiver（做事的人）。

21.2 优势与时机

命令模式我认为具备以下一些优势：

- 命令模式将请求方与接收方分开，有利于降低系统的耦合度。
- 命令类，可以方便维护与扩展。
- 添加新的命令实现类，变得更加简单。
- 根据系统的需求，控制命令状态和添加命令日志显得比较简单。
- 多个命令对象聚合起来，可以构造成一个命令集合。

基于此，可在以下几种情形下采用命令模式进行软件设计与实施。

- 抽象需要做的事件，以用作对象的参数时。
- 在不同的时间段，需要指定或排序时。
- 需要取消命令操作时。
- 可修改日志，并为系统崩溃的数据更新做准备时。
- 可构造一个新的交易类型时。

21.3 提升方向

可能会导致，具体命令类数量过多，而影响系统的实现。当某些命令十分相似时，可结合原型模式进行开发。

21.4 应用情境——豆浆机制作饮料

当前市场上，豆浆机制作饮料的功能键主要包括干豆/湿豆豆浆、五谷豆浆、果蔬冷饮、玉米汁。要使用命令模式实现豆浆机制作饮料的过程。

基本思路如下：

- 豆浆机上的功能键就相当于是命令对象。
- 豆浆相当于是 Invoker。
- 电路相当于接收者对象。
- 命令对象持有一个接收者对象，就相当于是给豆浆的按钮连上了一根连接线。
- 当豆浆上的按钮被按下的时候，豆浆就把这个命令通过连接线发送出去。

因此，电路类才是真正实现开机功能的地方，是真正执行命令的地方，也就是“接收者”。命令的实现对象其实是个“虚”的实现，就如同那根连接线，它不知如何实现，只是把命令传递给连接线连到的电路。使用命令模式来设计的类图如图 21.3 所示。

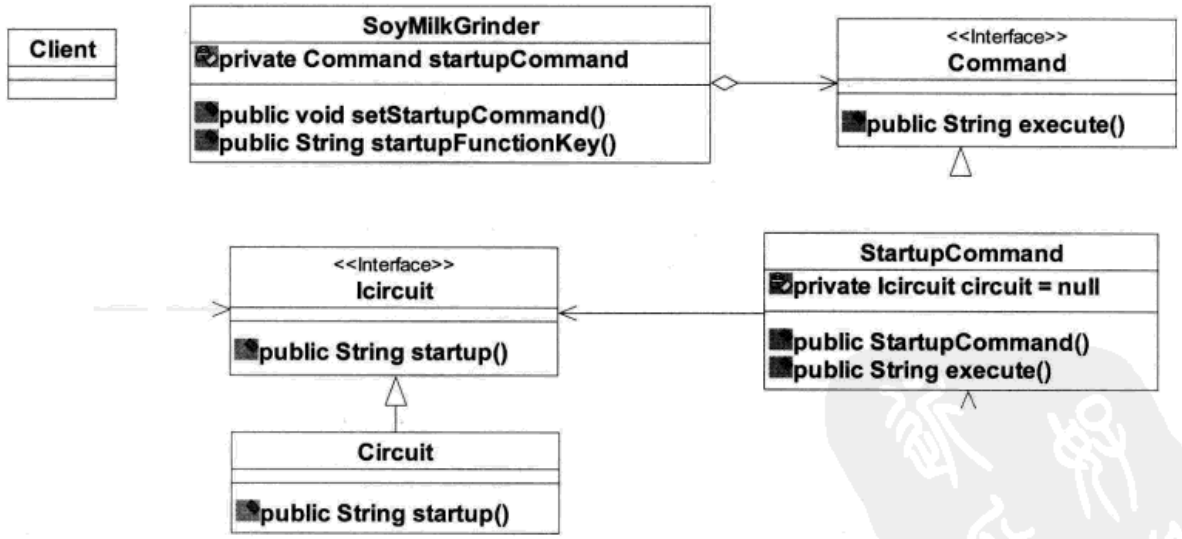

图 21.3

本应用情景的工程名为“程序 21.4.1”，源代码如下所示。

（1）Invoker（调用者）类

```
package model.command;
```

```
/**
 * 拥有功能键对应命令对象的豆浆机对象 invoker
 */
public class SoyMilkGrinder {
    /**
     * 启动命令对象
     */
    private Command startupCommand;
    /**
     * 设置启动机器命令对象
     * @param command 启动机器命令对象
     */
    public void setStartupCommand(Command command){
        this.startupCommand = command;
    }
    /**
     *为客户提供，接收并响应用户请求的功能。也就是按下功能键，进行的方法。
     */
    public String startupFunctionKey(){
        return startupCommand.execute();
    }
}
```

（2）命令接口类

```
package model.command;

/**
 * 命令接口，声明执行的操作
 */
public interface Command {
    /**
     * 执行命令对应的操作
     */
    public String execute();
}
```

（3）命令实现类

```
package model.command;

/**
 * 具有启动机器命令的实现类，实现 Command 接口，通过调用接收者的方法来实现命令。
 */
public class StartupCommand implements Command{
    /**
```

```
     * 真正实现命令的接收者—电路对象
     */
    private Icircuit circuit = null;
    /**传入电路对象方法
     * @param circuit 电路对象
     */
    public StartupCommand(Icircuit circuit) {
       this.circuit = circuit;
    }
    public String execute() {
     return this.circuit.startup();
    }
}
```

（4）Receiver 接口

```
package model.command;

// Receiver 接口
public interface Icircuit {
    /**
     * 电路的启动功能
     */
    public String startup();
}
```

（5）Receiver 接口的实现类

```
package model.command;

/**
 * 电路类，启动命令的真正实现者， Receiver 接口的实现类
 */
public class Circuit implements Icircuit{
    /**
     * 真正的启动命令的实现
     */
    public String startup(){
     String a="当前，电路正在启动，请稍等！";
     String b="电源已接通！";
     String c="检查机器！";
     String d="应用程序载入！";
     String e="机器正常运转起来！";
     String f="豆浆机已经正常打开，请正常操作！";
       System.out.println("当前，电路正在启动，请稍等！");
       System.out.println("电源已接通！");
       System.out.println("检查机器！");
       System.out.println("应用程序载入！");
```

```
        System.out.println("机器正常运转起来！");
        System.out.println("豆浆机已经正常打开，请正常操作！");
        return a+b+c+d+e+f;
    }
}
```

（6）客户端（test.jsp）

```
<HTML>
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    命令模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.command.*" %>
<br />
<br />
<h4>
<%
//A.将命令和真正的实现组合起来，也就是组装机器，将豆浆机功能键上的连接线与电路相连接。
Icircuit circuit = new Circuit();
StartupCommand startupCommand = new StartupCommand(circuit);
//B.为豆浆机的功能键设置对应的命令，使功能键与命令相匹配。
SoyMilkGrinder soyMilkGrinder = new SoyMilkGrinder();
soyMilkGrinder.setStartupCommand(startupCommand);%>
</h4>
<% //C.按下豆浆机上的功能键
out.println(soyMilkGrinder.startupFunctionKey());   %><br />
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
当前，电路正在启动，请稍等！电源已接通！检查机器！应用程序载入！机器正常运转起来！豆浆机已经正常打开，请正常操作！
```

21.5 命令模式与 Struts

在 Struts 框架中的 Action 类是作为响应“用户请求层与后台业务逻辑层”之间的关联点，它可一一对应客户所需的业务代理。当 RequestProcessor 类预备进行请求处理时将生成 Action 的有关实例，并通过 processActionPerform()方法对 Action 的 execute()方法进行相应的调用。

Action 的 execute()方法通过调用模型的有关业务方法，实现客户请求的有关业务逻辑，并以执行效果为依据将请求转发至其余恰当的 Web 组件。execute()的简要配置与代码如下所示。

（1）Strust-config.xml 的配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts
Configuration 1.2//EN" "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<struts-config>
  <data-sources />
  <form-beans >
    <form-bean name="manForm" type="com.test.ManForm" />
  </form-beans>
  <global-exceptions />
  <global-forwards />
  <action-mappings >
    <action
      attribute="manForm"
      input="/man.jsp"
      name="manForm"
      parameter="status"
      path="/man"
      scope="request"
      validate ="false"
      type="com.distributaction.struts.action.ManAction" />
  </action-mappings>
<message-resources
parameter="com.distributaction.struts.ApplicationResources" />
</struts-config>
```

（2）ManAction 的 execute()代码（命令角色）

```
//search 和 del 是 execute()方法
    public ActionForward search(ActionMapping mapping, ActionForm form,
          HttpServletRequest request, HttpServletResponse response) {
      ManForm manForm = (ManForm) form;
      String name  = manForm.getName();
      System.out.println("查询方法:"+ name);
      return null;//
    }
    public ActionForward del(ActionMapping mapping, ActionForm form,
          HttpServletRequest request, HttpServletResponse response) {
      ManForm manForm = (ManForm) form;
      String name  = manForm.getName();
      System.out.println("删除方法:"+ name);
      return null;//是 execute()方法
    }
```

由此得出，Action 类运用了命令模式。

第 22 章　Interpreter（解释器）模式

22.1　概述

解释器模式是指，“给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。”[1]

粗看其意图，大家就能从字面上就可以得出结论。

解释器模式其实就是对需要解释的语言进行解释，它在实际软件项目开发过程中并不常见。

为了便于读者进一步理解解释器模式，本章通过模式的结构与实例进行解说。

1. 结构（如图 22.1 所示）

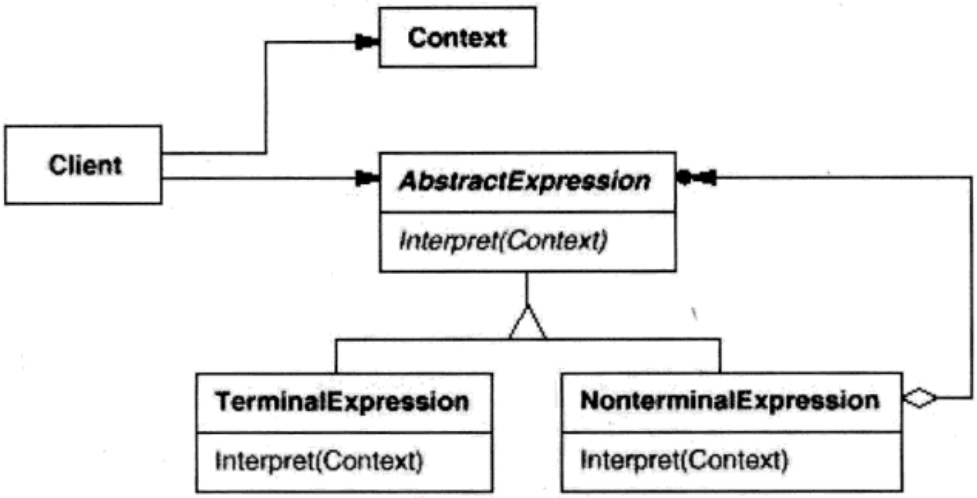

图 22.1

- AbstractExpression（抽象表达式，可用接口或抽象类）：声明一个抽象的解释操作，这个接口为抽象语法树中所有的节点所共享。
- TerminalExpression（终结符表达式）：实现与文法中的终结符相关联的解释操作，一个句子中的每个终结符需要该类的一个实例。
- NonterminalExpression（非终结符表达式）：为文法中的非终结符实现解释(Interpret)操作。
- Context（上下文）：包含解释器之外的一些全局信息。

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，162

- Client（客户）：构建(或被给定)表示该文法定义的语言中一个特定的句子的抽象语法树。该抽象语法树由 NonterminalExpression 和 TerminalExpression 的实例装配而成，调用解释操作。[1]

2. 实例

软件系统中，偶尔会有涉及到需要使用解释器的时候，由于项目成本、时间等因素的影响。具体使用何种解释器就需要进行选择。如图 22.2 所示。

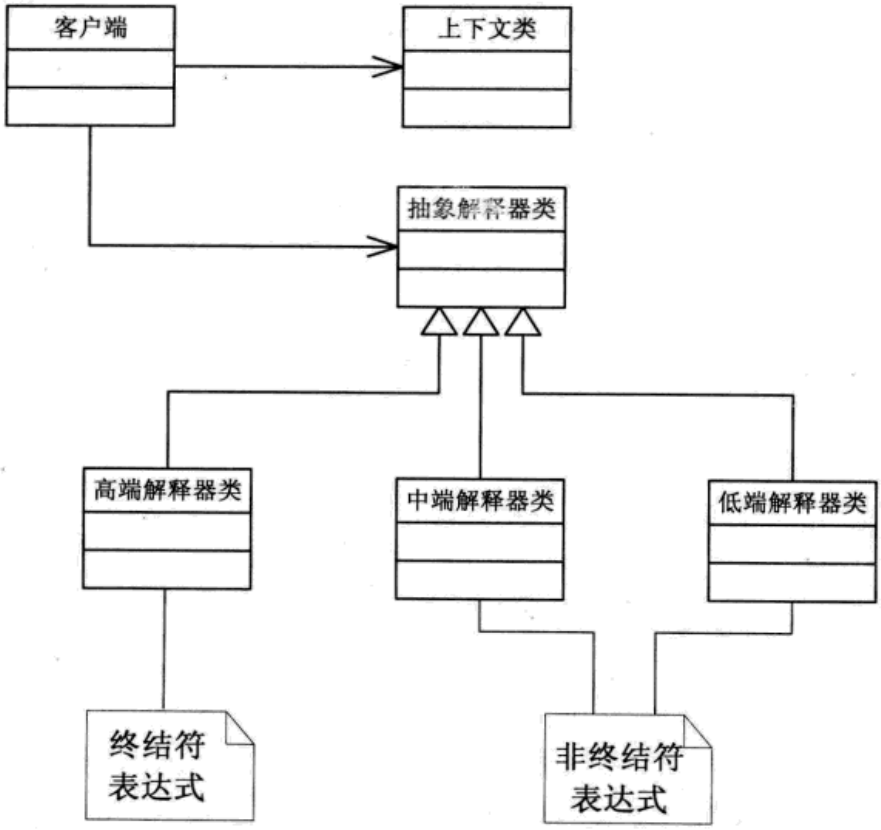

图 22.2

客户端即 Client，上下文类即 Context，抽象解释器类即 AbstractExpression，高端解释器类即 TerminalExpression，中端解释器类、低端解释器类即 NonterminalExpression。

22.2 优势与时机

解释器模式具备的优势：

- 解释器模式，具备良好的扩展性；通过修改非终结符类，即可维护语法规则。通过新添非终结符类，即可扩展语法结构。
- 解释器中内容不同的类，实现了不同的规则。因此，简化了语法规则的实现。

[1] Erich Gamm.设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，163~164

基于此，我认为可以在以下几种情形下采用解释器模式进行软件设计与实施。

- 需要运用抽象语法树， 对一个语言进行解释时。
- 多次出现相同的问题时。比如 WebService 的数据传送，其数据格式可做共性设计。
- 需要进行形式转换时。比如语言的字符格式转换。

22.3 提升方向

如果文法过于复杂，则需要产生大量的规则处理类，由此将导致，程序维护过于庞大。如果规则类过多，由于它们需要递归调用抽象表达式，这将导致系统性能大大降低。

22.4 应用情境——逻辑判断

对于从事软件开发的朋友来说，当我们软件项目中使用 or、and 以及 no 进行逻辑判断时，可运用解释器模式进行解释，如图 22.3 所示。

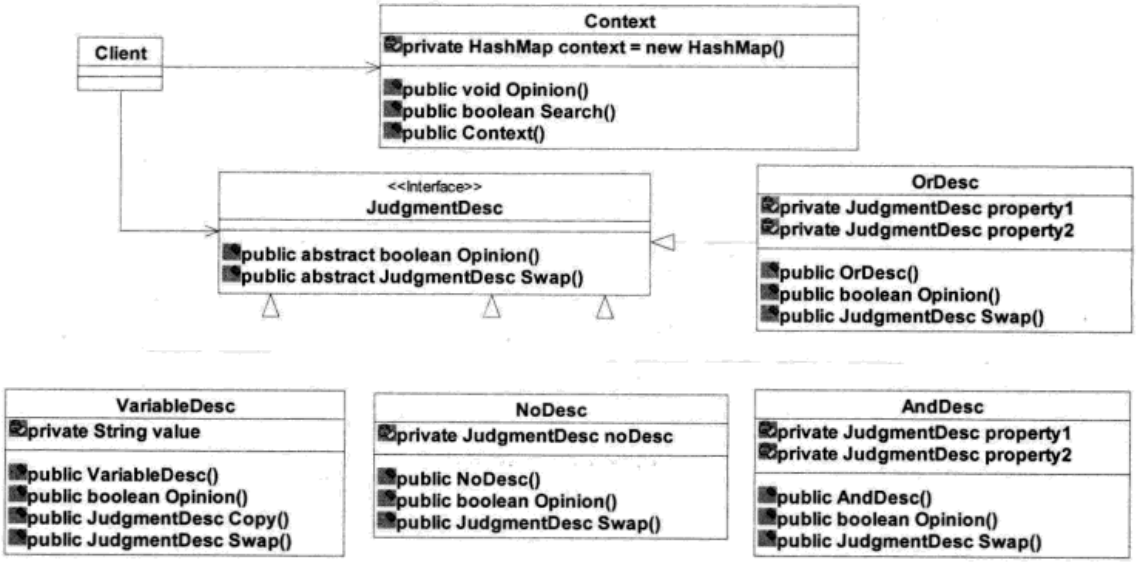

图 22.3

本应用情景的工程名为“程序 22.4.1”，源代码如下所示。

（1）抽象表达式

```
package model.interpreter;
/*
 * AbstractExpression-抽象表达式
 */
public interface JudgmentDesc {//评判
    public abstract boolean Opinion(Context context);//判断逻辑
    public abstract JudgmentDesc Swap(String value, JudgmentDesc desc);//交换
}
```

（2）终结符表达式

```
package model.interpreter;
/**
 * TerminalExpression(终结符表达式)
 */
import java.util.*;
public class VariableDesc implements JudgmentDesc {
    private String value;
    public VariableDesc(String _value) {
     value = _value;
    }
    public boolean Opinion(Context context) {
        return context.Search(value);
    }
    public JudgmentDesc Copy() {
        return new VariableDesc(value);
    }
    public JudgmentDesc Swap(String operor, JudgmentDesc desc) {
        if(operor.equals(value)) {
            return new VariableDesc(value);
        }
         return desc;
    }
}
```

（3）非终结符表达式（NoDesc）

```
package model.interpreter;
/**
 *    属于 NonterminalExpression 非终结符表达式
 */
public class NoDesc implements JudgmentDesc {
    private JudgmentDesc noDesc;
    public NoDesc(JudgmentDesc judgmentDesc) {
     noDesc = judgmentDesc;
    }
    public boolean Opinion(Context context) {//判断逻辑
        return !(noDesc.Opinion(context));
    }
    public JudgmentDesc Swap(String operor, JudgmentDesc desc) {//替换逻辑
        return new NoDesc(noDesc.Swap(operor, desc));
    }
}
```

（4）非终结符表达式（AndDesc）

```
package model.interpreter;
```

```
/**
 * 属于 NonterminalExpression 非终结符表达式
 */
public class AndDesc implements JudgmentDesc {
    private JudgmentDesc property1;//属性 1
    private JudgmentDesc property2;//属性 2
    public AndDesc(JudgmentDesc desc1, JudgmentDesc desc2) {
     property1 = desc1;
     property2 = desc2;
    }
    public boolean Opinion(Context context) {//意见
        return property1.Opinion(context) &&
        property2.Opinion(context);
    }
    public JudgmentDesc Swap(String value, JudgmentDesc desc) {//交换
        return new AndDesc(
             property1.Swap(value, desc),
             property2.Swap(value, desc)
                 );
    }
}
```

（5）非终结符表达式（OrDesc）

```
package model.interpreter;
/**
 * 属于 NonterminalExpression 非终结符表达式
 */
public class OrDesc implements JudgmentDesc {
    private JudgmentDesc property1;
    private JudgmentDesc property2;
    public OrDesc(JudgmentDesc judgment1, JudgmentDesc judgment2) {
     property1 = judgment1;
     property2 = judgment2;
    }
    public boolean Opinion(Context context) {//判断逻辑
        return property1.Opinion(context) ||
        property2.Opinion(context);
    }
    public JudgmentDesc Swap(String operor, JudgmentDesc desc) {//替换逻辑
        return new OrDesc(
             property1.Swap(operor, desc),
             property2.Swap(operor, desc)
                 );
    }
}
```

（6）上下文类

```
package model.interpreter;
/**
 * Context-上下文类
 */
import java.util.*;

public class Context  {
    private HashMap context = new HashMap();
    public void Opinion(String name, boolean value) {
        context.put(name, new Boolean(value));
    }
    public boolean Search(String name) {
        return ((Boolean)context.get(name)).booleanValue();
    }
    public Context() {
    }
}
```

（7）客户端类（test.jsp）

```
<HTML>

<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    解释器模式
    </TITLE>
</HEAD>

<BODY BGCOLOR="white">

<%@ page pageEncoding="GB2312" language="java" import="model.interpreter.*"
%>

<br />
<br />
<h4>
<%
Context context = new Context();
VariableDesc vd1 = new VariableDesc("X");
VariableDesc vd2 = new VariableDesc("Y");
VariableDesc vdTure = new VariableDesc("true");
VariableDesc vdFalse = new VariableDesc("false");
context.Opinion("true", true);
context.Opinion("false", false);
context.Opinion("X", false);
context.Opinion("Y", true);
```

```
//----AND 表达式----
JudgmentDesc description1 = new AndDesc(
    new AndDesc(vdTure, vd2),
    new AndDesc(vd2, new NoDesc(vd1))
);
boolean outcome1 = description1.Opinion(context);
//System.out.println("结果: AND 的逻辑为 " + outcome1);

%>
<% out.println("结果: AND 的逻辑为 " + outcome1);    %><br >
<%//----Or 表达式----
JudgmentDesc description2 = new OrDesc(
          new OrDesc(vdTure, vd1),
          new OrDesc(vd2, new NoDesc(vd1))
        );
        boolean outcome2 = description2.Opinion(context);
        //System.out.println("结果: OR 的逻辑为" + outcome2);
        //------------------
%>
<% out.println("结果: OR 的逻辑为" + outcome2);  %><br >
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
结果: AND 的逻辑为 true
结果: OR 的逻辑为 true
```

第 23 章　Iterator（迭代器）模式

23.1　概述

迭代者模式是指："提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。"[1]

粗看迭代者模式的意图，大家可能有点迷惑不解。其实，我们可以把它简化。

"提供一种方法顺序访问一个聚合对象中各个元素"可理解为"通过运用一个接口有序的访问一个数据集合或列表。"

"不需暴露该对象的内部表示"可理解为"无须显示数据集合或列表的明细信息"。

为了便于读者进一步理解迭代者模式，本章通过模式的结构、分类与实例进行解说。

1. 结构

常规迭代子如图 23.1 所示。

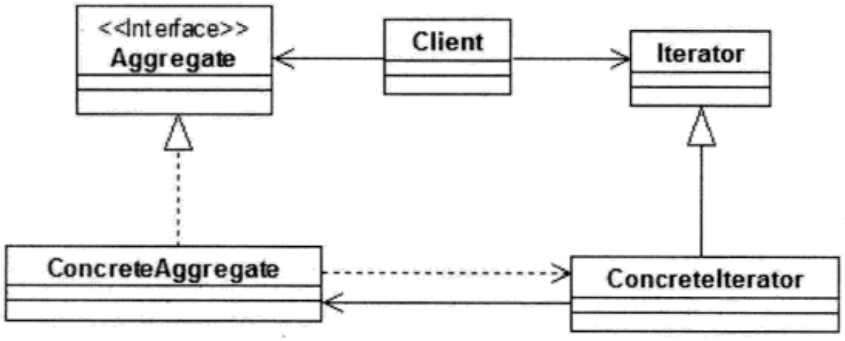

图 23.1

- Itertor（迭代器，可用接口或抽象类）：定义访问和遍历元素的接口。
- ConcreteIterator（具体迭代器）：实现迭代器接口，对该聚合遍历时跟踪当前位置。
- Aggregate（聚合，可用接口或抽象类）：聚合定义创建相应迭代器对象的接口。
- Concrete Aggregate（具体聚合）：具体聚合实现创建相应迭代器的接口，该操作返回 ConcreteIterator 的一个适当的实例。[2]

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，171

[2] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，172~173

（1）白箱聚集外禀迭代子

它为外部供应访问自身内部元素的接口，令外禀迭代子可运用聚集供应的方法实现迭代功能，如图 23.2 所示。

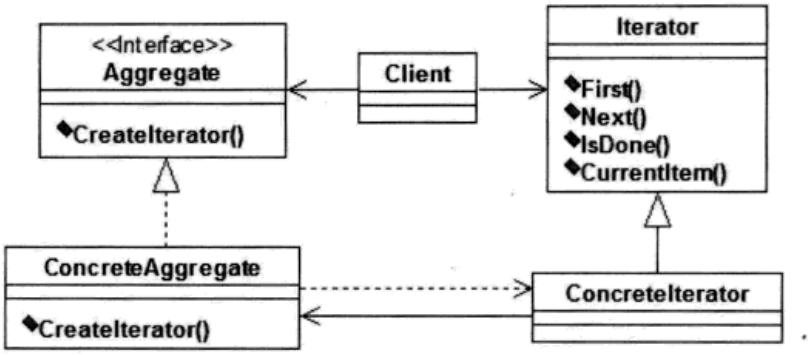

图 23.2

（2）黑箱聚集内禀迭代子

黑箱聚集不为外界提供遍历自身的元素的接口，因而聚集的成员只能被聚集内部的方法访问。由于内禀迭代子恰好是聚集的成员，因此可访问聚集元素。内部迭代器可顺序访问全体集合的某些方法，无须用户发出任何明确的请求便可直接对每个元素进行操作，如图 23.3 所示。

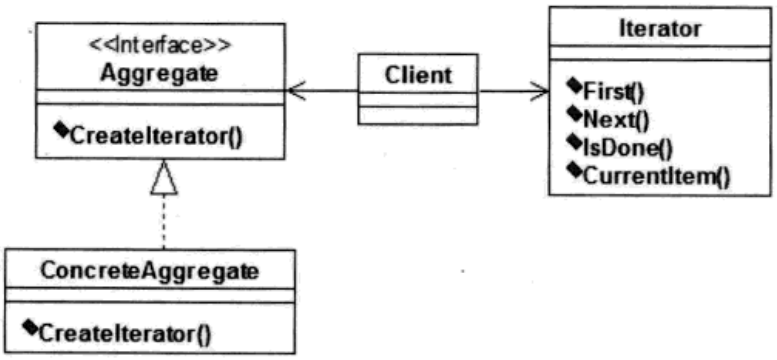

图 23.3

2. 实例

在永乐家电的商场中有一些手机柜台，如果我们运用迭代器模式，该如何把柜台上的手机依次输出手机名。

此实例，我们可设计的类图如下所示。

（1）白箱聚集外禀迭代子（如图 23.4 所示）

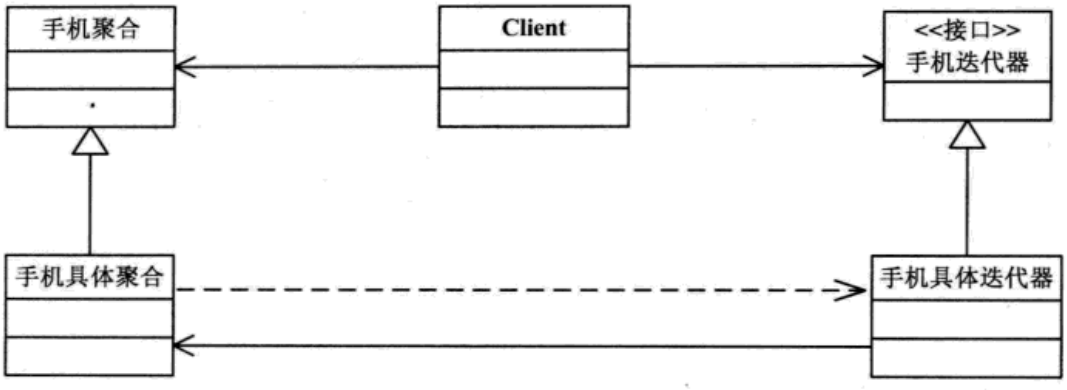

图 23.4

（2）黑箱聚集内禀迭代子（如图 23.5 所示）

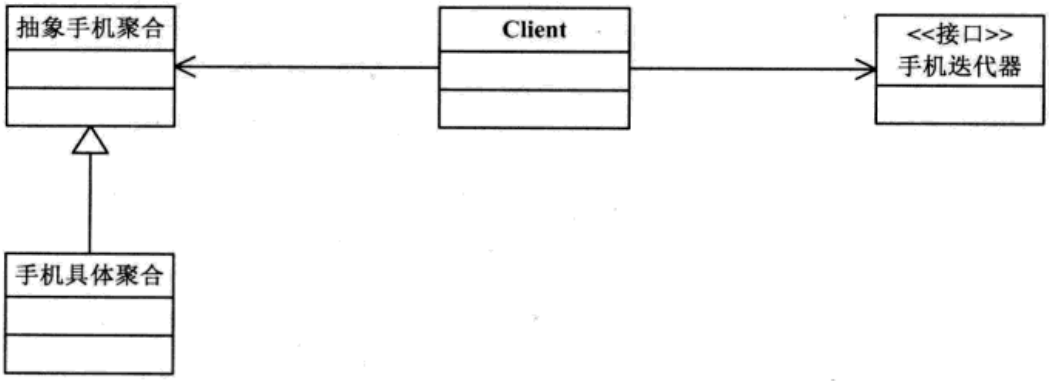

图 23.5

23.2 优势与时机

迭代器模式具备的优势：

- 可通过多种方式遍历一个聚合，并且不同的实现方式达到不同的效果。
- 由于迭代器具备遍历接口，因而聚合无须提供遍历接口。
- 同一聚合中可实现多个遍历，并且每个迭代器均可有遍历状态。

基于此，我认为可以在以下几种情形下采用迭代器模式进行软件设计与实施。

- 当需要访问某一聚合对象的内容，而不必显示内部表示时。
- 需要多次遍历容器对象时。
- 提供某一统一的接口，方便遍历各不相同的聚合结构时。

23.3 提升方向

迭代器模式包括存储数据和遍历数据两大独立功能，当添加一个聚合类时，将增加相应的新迭代器类。如果类的数量成对增长，则会导致系统过于复杂。

尽管迭代需要按照一定的线性顺序活动，但是聚集的元素的顺序不确定。如此一来，将有可能导致客户端由于调用聚集元素，而产生错误的结果。

迭代子元素一般都是对象类，因此，客户端调用时，必须要明确元素类型才可使用。

23.4 应用情境——邻居小张餐饮店的日常成本支出

某周六，我正在家看书，突然听到敲门声。我打开门一看，原来是邻居小张。“张老板你好，今天有何贵干呀？”我说。“见外，见外了兄弟，有个事儿请你帮个忙。”小张说。接着我俩一起去了书房，“情况是这样，我现在开了三个餐饮店，你看是否可以给我搞个功能，让我查一查平时的成本支出呢。”小张说。

“没问题，两天帮你搞定！”我愉快的答应道。

我俩又聊了十几分钟的闲话，小张就走了，于是我就开始设计类图并进行编码，类图如图 23.6 所示。

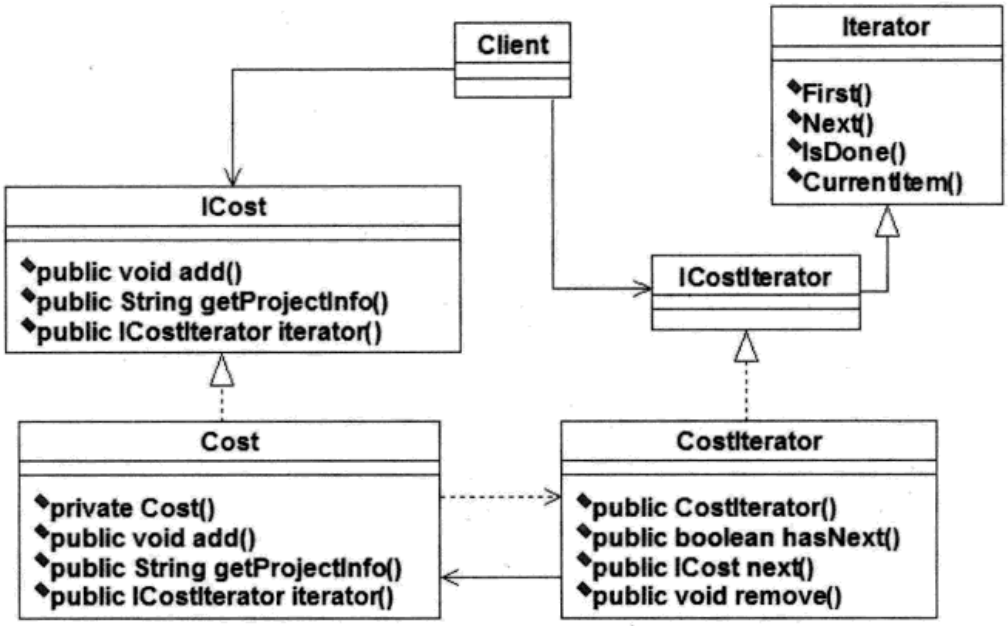

图 23.6

本应用情景的工程名为“程序 23.4.1”，源代码如下所示。

（1）Itertor（迭代器）

```
package model.iterator;
import java.util.Iterator;

/**
 * @author jianghc
 * 定义个 Iterator 接口
 */
public interface ICostIterator extends Iterator {
```

```
}
```

（2）ConcreteIterator（具体迭代器）

```
package model.iterator;
import java.util.ArrayList;

/**
 * @author jianghc
 * 定义一个迭代器
 */
public class CostIterator implements ICostIterator {
    //所有的项目都放在这里ArrayList中
    private ArrayList<ICost> costList = new ArrayList<ICost>();
    private int currentItem = 0;
    //构造函数出入costList
    public CostIterator(ArrayList<ICost> projectList){
        this.costList = projectList;
    }
    //判断是否还有元素，必须实现
    public boolean hasNext() {
        //定义一个返回值
        boolean b = true;
        if(this.currentItem>=costList.size()
this.costList.get(this.currentItem) == null){
            b =false;
        }
        return b;
    }
    //取得下一个值
    public ICost next() {
        return (ICost)this.costList.get(this.currentItem++);
    }
    //删除一个对象
    public void remove() {
        //暂时没有使用到
    }
}
```

（3）Aggregate（聚合，可用接口或抽象类）

```
package model.iterator;
/**
 * @author jianghc
 * 定义一个接口，所有的成本都是一个接口- 聚合(Aggregate可使用接口或抽象类)
 */
public interface ICost {
    //增加项目
```

```
    public void add(String name,int num,int cost);
    //小张查看成本信息
    public String getProjectInfo();
    //取得一个可被遍历对象
    public ICostIterator iterator();
}
```

（4）Concrete Aggregate（具体聚合）

```
package model.iterator;
import java.util.ArrayList;

/**
 * @author jianghc
 * 成本信息类--具体聚合(ConcreteAggregate)
 */
public class Cost implements ICost {
    //定义一个成本列表
    private ArrayList<ICost> costList = new ArrayList<ICost>();
    //成本内容
    private String content = "";
    //员工数量
    private int num = 0;
    //费用
    private int charge = 0;
    public Cost(){

    }
    //定义一个构造函数，把所有老板需要看到的信息存储起来
    private Cost(String content,int num,int charge){
        //赋值到类的成员变量中
        this.content = content;
        this.num = num;
        this.charge=charge;
    }
    //增加项目
    public void add(String content,int num,int charge){
        this.costList.add(new Cost(content,num,charge));
    }
    //得到项目的信息
    public String getProjectInfo() {
        String info = "";
        //成本内容
        info = info+ "成本名称: " + this.content;
        //人力成本数
        info = info + "\t 人力成本: "+ this.num;
        //费用
        info = info+ "\t 费用: "+ this.charge;
        return info;
```

```
    }
    //产生一个遍历对象
    public ICostIterator iterator(){
        return new CostIterator(this.costList);
    }

}
```

（5）客户端（test.jsp）

```
/**
 * @author jianghc
 * 老板来看项目信息了 客户端  Boss
 */

<HTML>
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    迭代器模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.iterator.*" %>
<br />
<br />
<h4>
<%
//定义一个 List，存放所有的项目对象
ICost cost = new Cost();
//采购支出
cost.add("采购",10,100000);
//人力支出
cost.add("人力",100,10000000);
//纳税支出
cost.add("纳税",10000,1000000000);
//这边 30 个项目
for(int i=4;i<34;i++){
    cost.add("第"+i+"个支出",i*5,i*1000000);
}
//遍历一下 ArrayList，把所有的数据都取出
ICostIterator projectIterator = cost.iterator();
while(projectIterator.hasNext()){
    ICost p = (ICost)projectIterator.next();
%>
<%
  out.println(p.getProjectInfo());
}
  %><br >
```

```
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
成本名称：采购 人力成本：10 费用：100000 成本名称：人力 人力成本：100 费用：10000000 成本名称：纳税 人力成本：10000 费用：1000000000 成本名称：第 4 个支出 人力成本：20 费用：4000000 成本名称：第 5 个支出 人力成本：25 费用：5000000 成本名称：第 6 个支出 人力成本：30 费用：6000000 成本名称：第 7 个支出 人力成本：35 费用：7000000 成本名称：第 8 个支出 人力成本：40 费用：8000000 成本名称：第 9 个支出 人力成本：45 费用：9000000 成本名称：第 10 个支出 人力成本：50 费用：10000000 成本名称：第 11 个支出 人力成本：55 费用：11000000 成本名称：第 12 个支出 人力成本：60 费用：12000000 成本名称：第 13 个支出 人力成本：65 费用：13000000 成本名称：第 14 个支出 人力成本：70 费用：14000000 成本名称：第 15 个支出 人力成本：75 费用：15000000 成本名称：第 16 个支出 人力成本：80 费用：16000000 成本名称：第 17 个支出 人力成本：85 费用：17000000 成本名称：第 18 个支出 人力成本：90 费用：18000000 成本名称：第 19 个支出 人力成本：95 费用：19000000 成本名称：第 20 个支出 人力成本：100 费用：20000000 成本名称：第 21 个支出 人力成本：105 费用：21000000 成本名称：第 22 个支出 人力成本：110 费用：22000000 成本名称：第 23 个支出 人力成本：115 费用：23000000 成本名称：第 24 个支出 人力成本：120 费用：24000000 成本名称：第 25 个支出 人力成本：125 费用：25000000 成本名称：第 26 个支出 人力成本：130 费用：26000000 成本名称：第 27 个支出 人力成本：135 费用：27000000 成本名称：第 28 个支出 人力成本：140 费用：28000000 成本名称：第 29 个支出 人力成本：145 费用：29000000 成本名称：第 30 个支出 人力成本：150 费用：30000000 成本名称：第 31 个支出 人力成本：155 费用：31000000 成本名称：第 32 个支出 人力成本：160 费用：32000000 成本名称：第 33 个支出 人力成本：165 费用：33000000
```

23.5 迭代器模式与 Struts

Struts 框架的标签可以体现出迭代器模式。

大家可以查看以下实例：

（1）jsp 页面

```
<%@ page contentType="text/html; charset=gb2312" %>
<html>
<head>
Test
</head>
<body>
<%
 if(count.size()==0)
 out.print("缺少会员评论");
 if(count.size()>0)
 {
 int size=count.size();
 (size>6)
 size=6;
```

```
 for(int a=0;a<size;a++)
 {
 member meb = (member) count.get(a);
%>
</body>
</html>
```

（2）Struts 部分

接下来读取与<%=meb.getMemberNo() %>的属性相近的 JSP 部分内容，运用 Struts2 的标签进行编写，<s:iterator value="memberCore2" >//memberCore2 是某个从 Action 中读取的 List<member>。

此后需要迭代，可以运用下面所示代码进行迭代器中元素数量的统计。

```
<s:if test="%{memberCore2.size == 0}">
  没有相关纪录!
 </s:if>
 <s:iterator value="memberCore2" status="info1">
   <s:if test="#info1.count <=6">
      <s:property value="businessmanNo" />
   </s:if>
 </s:iterator>
```

第 24 章　Mediator（中介者）模式

24.1　概述

所谓中介者模式是指“用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使期耦合松散，而且可以独立地改变它们之间的交互”。[1]

粗看此模式，大家觉得比较容易理解，确实，中介者模式可理解为：“将每个组件间的复杂交互，运用当中某一‘中介者’来完成。”

为了方便，进一步理解，本章从结构与实例的角度进行阐述。

1. 结构，如图 24.1 所示

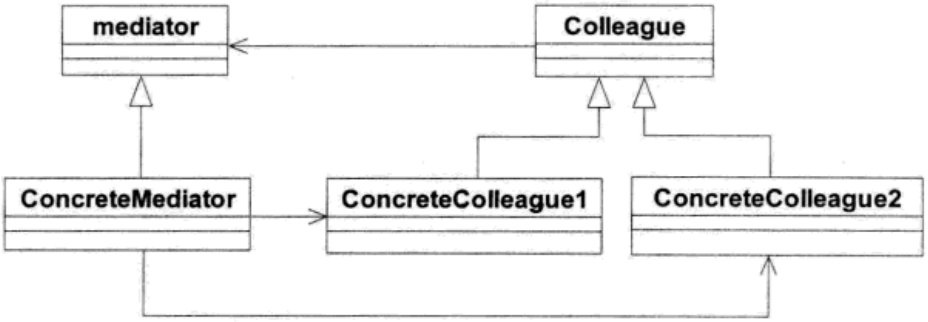

图 24.1

- 中介者（Mediator，可接口或抽象类）：定义一个接口用于与各同事对象通信。
- 具体中介者（ConcreteMediator）：通过协调各同事对象实现协作行为。了解并维护它的各个同事。
- 同事（Colleague，可接口或抽象类）：每一个同事类都知道它的中介者对象，每一个同事对象需在与其他的同事通信的时候与它的中介者通信。
- 具体同事（ConcreteColleague）：继承于抽象同事类，每一个具体同事类都很清楚它自己在小范围内的行为，而不知道它在大范围内的目的。[2]

2. 实例

一个省级的电力公司会有很多部门和员工，为完成一些任务同事之间需要较多的配合。比如

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，181

[2] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，183~184

用电申请业务，其业扩流程，涉及不少部门与员工。如果由每个同事多次寻找与自己相关的同事沟通，这会造成工作效率的大大降低。基于此，运用中介者模式设计如图 24.2 所示。

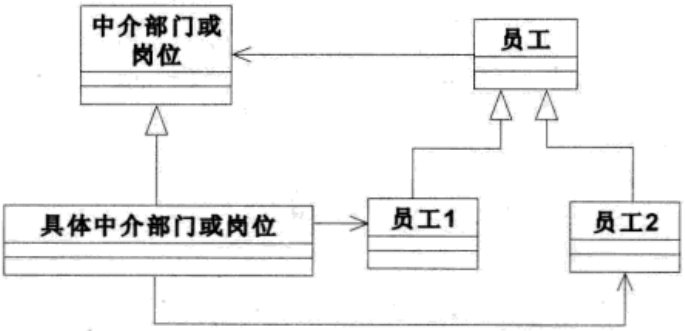

图 24.2

中介部门或岗位即中介者角色，具体中介部门或岗位即具体中介者角色，员工即同事角色，员工 1、员工 2 即具体同事角色。

24.2 优势与时机

中介者模式具备的优势：

- 将系统模块细分为多个对象，使对象间的耦合度降低。
- 当多个对象间需要变更其通信行为时，只需运用某一具体中介者即可实现同事代码的重用，并且无须变更具体同事的程序。
- 具体中介者的应用，使任意具体同事程序的修改，与其他同事无关。

基于此，我认为可以在以下几种情形下采用中介者模式进行软件设计与实施。

- 因多个对象间的交互方式十分复杂，其依赖关系将有可能导致软件系统无法维护时。
- 因某一对象调用过多对象，导致无法复用此对象时。
- 希望减少子类，以提高代码的重要性时（即创建某一分布在众多类中的行为时）。

24.3 提升方向

如果对象细分过多，则集中控制的方法将有可能导致程序可读性降低。如果中介程序的通信出现问题，则有可能导致系统性能降低。

24.4 应用情境——两个高中女生比拼男友的故事

两高中女生，一个叫蒋佳，一个叫张余。两人都是杭州某职业高中高二同班同学。她俩曾经是形影不离的好友。但是在今年 4 月份，两人的关系开始急速转变。双方由于比拼谁的男友更帅，而发生了口角。

针对此情境，我们系统设计师可运用中介者模式进行设计。具体类图可设计为如图 24.3 所示。

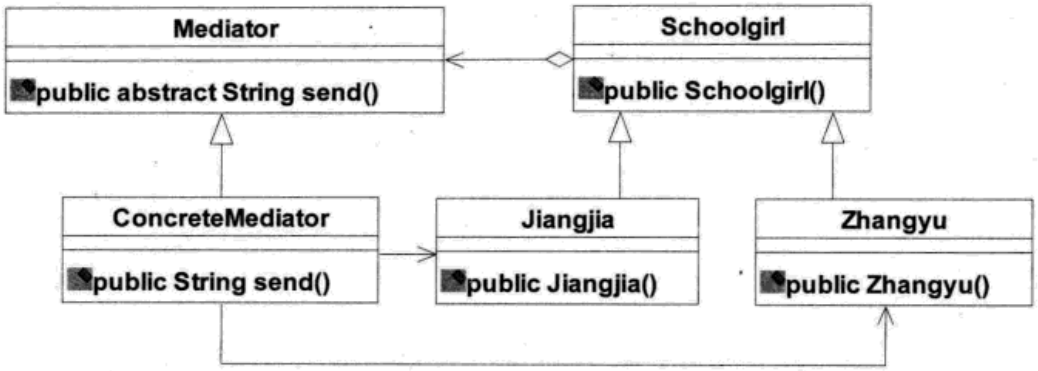

图 24.3

本应用情景的工程名为“程序 24.4.1”，源代码如下所示。

（1）中介者（Mediator）

```
package model.mediator;

/**
 * 抽象中介者类,此角色定义统一接口用于各同事角色间通信
 */
public abstract class Mediator {
    // 创建获取同事对象、发送消息的一个抽象发送消息方法。
    public abstract String send(String info,Schoolgirl schoolgirl);
}
```

（2）同事（Colleague）

```
package model.mediator;

/**
 * 抽象同事类 -学校女生
 */
public abstract class Schoolgirl {
    protected Mediator mediator;
    // 创建获取中介者对象的方法
    public Schoolgirl(Mediator mediator){
        this.mediator = mediator;
    }
}
```

（3）具体同事 1（ConcreteColleague）

```
/**
 * 具体同事对象 1,其角色必须了解对应的具体中介者角色，
 * 以便与其他同事角色通信时，通过中介者角色实现协作。
```

```
 */
public class Jiangjia extends Schoolgirl{
    public Jiangjia(Mediator mediator) {
        super(mediator);
    }
    public String send(String info){
        // 发送消息时通常由中介者发送出去
        return mediator.send(info, this);
    }
    public String notifyInfo(String info){
        String a="蒋佳说: ";
        System.out.println("蒋佳说: "+info);
        return a+info;
    }
}
```

（4）具体同事 2（ConcreteColleague）

```
/**
 * 具体同事对象 2,其角色必须了解对应的具体中介者角色,
 * 以便与其他同事角色通信时，通过中介者角色实现协作。
 */
public class Zhangyu extends Schoolgirl{

    public Zhangyu(Mediator mediator) {
        super(mediator);
    }

    public String send(String info){
        // 发送消息时通常是由中介者发送出去
        return mediator.send(info, this);
    }

    public String notifyInfo(String info){
        String a="张余说: ";
        System.out.println("张余说: "+info);
        return a+info;
    }
}
```

（5）具体中介者（ConcreteMediator）

```
/**
 * 具体中介者类，通过了解全体具体同事对象来协调各同事角色，从而达到协作的目标。
 */
public class ConcreteMediator extends Mediator{
    // 具体同事对象 1
    private Jiangjia jiangjia;
```

```
    // 具体同事对象 2
    private Zhangyu zhangyu;

    // 重写发送消息方法，以对象为依据进行选择判断，从而达到通知对象的目的。
    @Override
    public String send(String info, Schoolgirl schoolgirl) {
        if(schoolgirl == jiangjia){
         zhangyu.notifyInfo(info);
        }else{
         jiangjia.notifyInfo(info);
        }
         return info+schoolgirl.toString();
    }
    public Jiangjia getJiangjia() {
        return jiangjia;
    }
    public void setJiangjia(Jiangjia jiangjia) {
        this.jiangjia = jiangjia;
    }
    public Zhangyu getZhangyu() {
        return zhangyu;
    }
    public void setZhangyu(Zhangyu zhangyu) {
        this.zhangyu = zhangyu;
    }
}
```

（6）客户端（test.jsp）

```
<HTML>
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    中介者模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.mediator.*" %>
<br />
<br />
<h4>
<%
// 具体中介者对象
ConcreteMediator cm = new ConcreteMediator();
// 使两个具体同事类了解中介者对象
Jiangjia jj1 = new Jiangjia(cm);
Zhangyu zy2 = new Zhangyu(cm);
// 使中介者了解各个具体同事类对象
cm.setJiangjia(jj1);
```

```
cm.setZhangyu(zy2);
%>
<%
out.println("蒋佳:"+String.valueOf(jj1.send("瞧我男朋友，长得很帅噢！")));
%>
<br >
  <%
  out.println("张余:"+String.valueOf(zy2.send("切，我男朋友比你的更帅！")));
  %>
 <br >
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
蒋佳：瞧我男朋友，长得很帅噢！model.mediator.Jiangjia@106ab9a
张余：切，我男朋友比你的更帅！model.mediator.Zhangyu@ec1745
```

24.5 中介者模式与 Struts

从 MVC 的角度展开分析，控制器可以认为是一种中介者。那么作为 Struts 控制器类的 Action，其必然成为 Web 页面与业务之间的中介者。

此外，从 struts 的工作原理角度进行分析。在软件技术开发人员配置好 struts-config.xml 并完成解析后，那么当某一线程进行调用 ActionServlet 类的动作时，ActionServlet 可以使此线程进行访问并分配至相匹配的继承 Action 类当中，这良好地体现了中介者模式的相关概念与原理。

第 25 章　Memento（备忘录）模式

25.1　概述

所谓备忘录（Memento）模式是指："在不破坏封装性的前提下，捕获一个对象的内部状态并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。"[1]

粗看其定义，大家可能有点迷惑不解。其实，我们可以把它理解为"在保留原有对象封装的同时，依托一个对象存储另一对象的内部状态。从而在合适的时机，将原有对象还原。"

为了便于读者进一步理解备忘录模式，本章通过模式的结构与实例进行解说。

1. 结构

常规备忘录类图如图 25.1 所示。

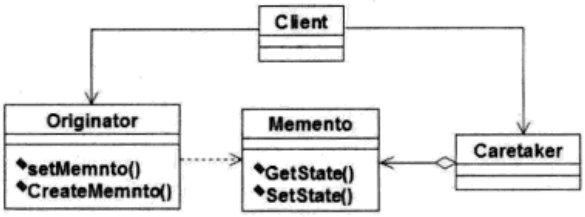

图 25.1

Memento（备忘录）：通过赋值可存储 Originator 对象的某些内部状态，并且能屏蔽 Originator 以外的其他对象访问备忘录 Memento。备忘录有两个接口，Caretaker 只能看到备忘录的窄接口，只能将备忘录传递给其他对象；Originator 能够看到一个宽接口，允许它访问返回到先前状态所需的所有数据。

Originator（发起人）：赋值创建一个备忘录 Memento，以记录当前时刻它的内部状态，并可使用备忘录恢复内部状态。它可视情况将 Memento 存储一些 Originator 内部状态。

Caretaker（负责人，即管理者）：负责保存好备忘录 Memento，不能对备忘录的内容进行操作或检查。[2]

（1）白箱备忘录

所谓白箱备忘录是指备忘录角色提供一个接口，方便所有对象的使用，并且备忘录角色的内部存储状态为所有对象开放。其结构与图 25.1 所示一致。

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，188

[2] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，189~190

（2）黑箱备忘录

当 Memento 成为 Originator 的内部类时，Memento 对象可封装在 Originator 中。并且在外部设计一个接口 MementoIF 为 Caretaker 及其他对象服务，如图 25.2 所示。

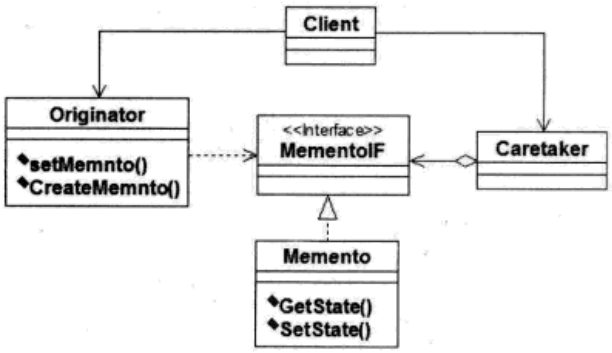

图 25.2

2. 实例

目前，传统的网上交易，消费者在按确认键后将只能是付款。此种情况下，对于消费者并不公平。为了防止商家弄虚作假，为了避免消费者的合同权益受到侵害。我国工商部门要求国内一些电子商务平台开发撤消商品买入的功能，以赋予消费者在合理的时间段内享受无条件的商品撤消权。

此实例，我们可设计的类图如下。

（1）箱备忘录（如图 25.3 所示）。

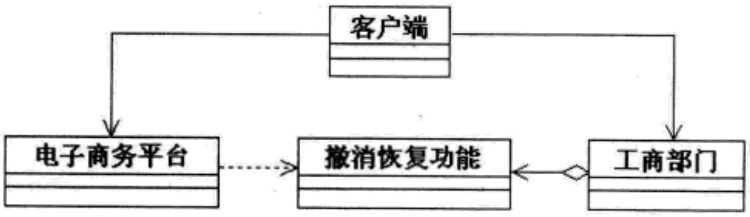

图 25.3

客户端即 Client，电子商务平台即 Originator，撤消恢复功能即 Memento，工商部门即 Caretaker。

（2）黑箱备忘录（如图 25.4 所示）。

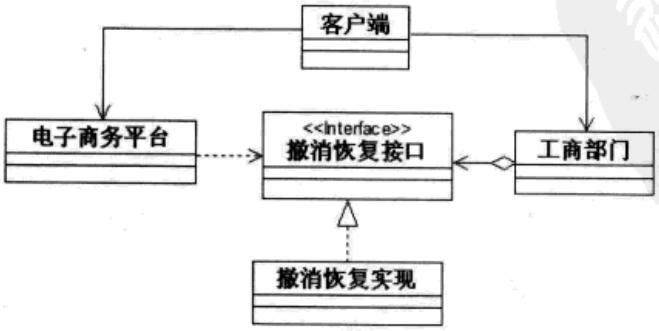

图 25.4

客户端即 Client，电子商务平台即 Originator，撤消恢复接口即 MementoIF，撤消恢复实现 Memento，工商部门即 Caretaker。

25.2 优势与时机

备忘录模式具备的优势：

- 为客户提供返回至某一历史时刻的机会。在最新状况有变或不稳定时，可运用备忘录去恢复某时段的状态。
- 将发起人的内部信息与外部对象隔离，从而确保 Memento 封装的不变性。
- 基于此，我认为可以在以下几种情形下采用备忘录模式进行软件设计与实施。
- 需要储存某一对象在某个时间点的状态，以便有需要时可及时恢复。
- 当某一接口使其他对象取得某些状态，并且显示对象的实现过程并可能。
- 改变对象的封装性。

25.3 提升方向

当类的成员变量不断增加时，其占有的内存量也随之上升。并且，每次对某一对象状态的保存都将消费一定的内存资源。由此，将导致用户需要增加内存与硬盘空间。

25.4 应用情境——版本控制

当前，在软件行业的版本控制系统中往往缺少不了撤销的功能。比如当申请人在一个版本控制系统中提交多次版本后，如果由于客户与领导的要求需要恢复至第 2 版，则需要申请人选择历史版本的第 2 版并将当前版本覆盖。

针对此情境，我们系统设计师可运用“备忘录模式”进行设计。具体类图可设计为白箱备忘录和黑箱备忘录：

1. 白箱备忘录（如图 25.5 所示）

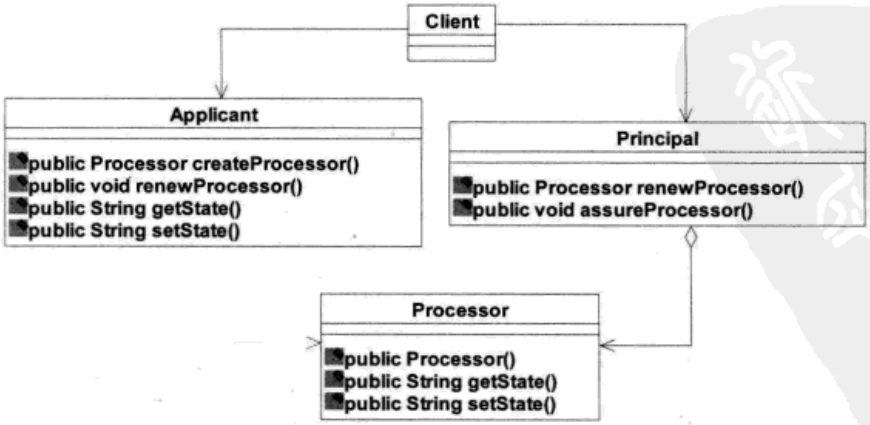

图 25.5

本应用情景的工程名为“程序 25.4.1”，源代码如下所示。

（1）发起人角色-申请人

```
package model.memento1;

//宽接口和白箱:
//发起人角色  -申请人
public class Applicant {
    private String state;

    //运用工厂方法，返还至某一新备忘录对象
    public Processor createProcessor() {
        return new Processor(state);
    }

    //申请人(即发起人)可恢复至备忘录对象中保存的状态
    public void renewProcessor(Processor processor) {
        this.state = processor.getState();
    }

    //状态取值
    public String getState() {
        return this.state;
    }

    //状态设值
    public String setState(String state) {
        String a="当前状态 = ";
        this.state = state;
        System.out.println("当前状态 = " + this.state);
        return a+state;
    }
}
```

（2）负责人角色

```
package model.memento1;

//负责人角色  Principal
public class Principal{
    private Processor processor;

    //备忘录恢复取值
    public Processor renewProcessor(){
        return this.processor;
    }

    //备忘录保证取值
```

```
    public void assureProcessor(Processor processor){
        this.processor = processor;
    }
}
```

（3）备忘录角色-数据处理机

```
package model.memento1;

//备忘录角色  processor  数据处理机
public class Processor{
    private String state;

    public Processor(String state){
        this.state = state;
    }

    public String getState(){
        return this.state;
    }

    public String setState(String state){
        return this.state = state;
    }
}
```

（4）客户端（test.jsp）

```
<HTML>
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    白箱备忘录
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.memento1.*" %>
<br />
<br />
<h4>
<%
 Applicant applicant = new Applicant();  //申请人对象
 Principal principal= new Principal();  //负责人角色

%>
<%
out.println("恢复至:"+String.valueOf(applicant.setState("第 2 版"))); //申请人
对象的状态
```

```
    principal.assureProcessor(applicant.createProcessor());//创建备忘录对象，并存储
申请人对象状态
    %>
    <br >
      <%
      out.println("将当前版本:"+String.valueOf(applicant.setState("覆盖"))) ;  //
变更申请人对象状态
      applicant.renewProcessor(principal.renewProcessor()); //恢复申请人对象状态
      %>
     <br >
    </h4>
    </BODY>
    </HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
恢复至：当前状态 = 第 2 版
将当前版本：当前状态 = 覆盖
```

2. 黑箱备忘录（如图 25.6 所示）

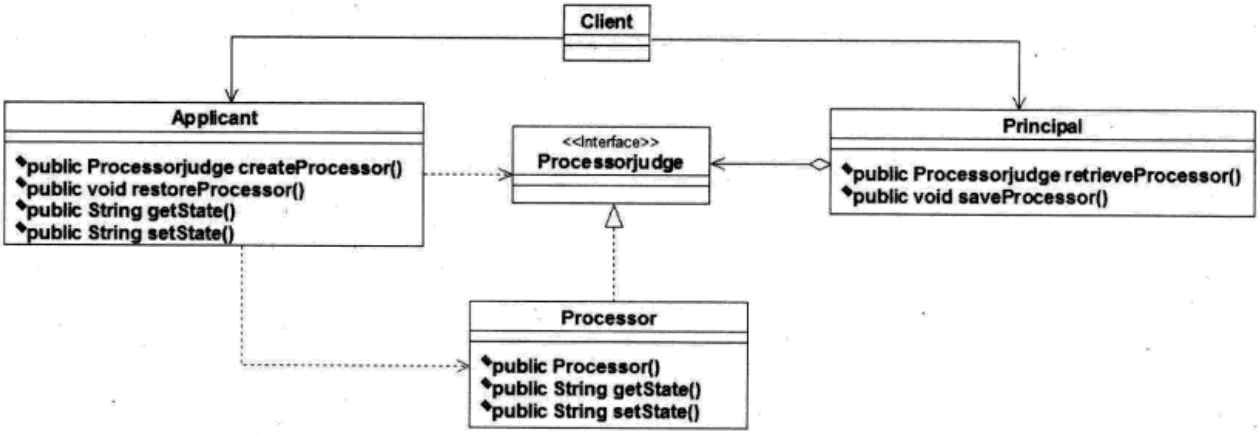

图 25.6

本应用情景的工程名为“程序 25.4.2”，源代码如下所示。

（1）窄接口或者黑箱实现

```
package model.memento2;

//窄接口或者黑箱实现
//发起人角色    Originator
public class Applicant {
    private String state;

    public Applicant() {
    }

    // 运用工厂方法，返还某一新备忘录对象
```

```
    public Processorjudge createProcessor() {
        return new Processor(this.state);
    }

    // 使发起人恢复至备忘录对象已记录之状态
    public void restoreProcessor(Processorjudge processorjudge) {
        Processor a_processor = (Processor) processorjudge;
        this.setState(a_processor.getState());
    }

    public String getState() {
        return this.state;
    }

    public String setState(String state) {
        this.state = state;
        System.out.println("state = " + state);
        return state;
    }

    protected class Processor implements Processorjudge { // Processor 备忘录
角色
        private String savedState;

        private Processor(String severalState) {
            savedState = severalState;
        }

        private String setState(String severalState) {
            return savedState = severalState;
        }

        private String getState() {
            return savedState;
        }
    }
```

（2）备忘录角色接口

```
package model.memento2;

// 备忘录角色
public interface Processorjudge {
}
```

（3）负责人角色

```
package model.memento2;
```

```
////负责人角色
public class Principal {
    private Processorjudge processorjudge; //备忘录角色

    public Processorjudge retrieveProcessor() { //恢复备忘录
        return this.processorjudge;
    }

    public void saveProcessor(Processorjudge processorjudge) {
        this.processorjudge = processorjudge;
    }
}
```

（4）备忘录角色

```
package model.memento2;

//备忘录角色  processor  数据处理机
public class Processor implements Processorjudge{
    private String state;

    public Processor(String state){
        this.state = state;
    }

    public String getState(){
        return this.state;
    }

    public String setState(String state){
        return this.state = state;
    }
}
```

（5）客户端（test.jsp）

```
<HTML>
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    黑箱备忘录
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.memento2.*" %>
<br />
<br />
```

```
<h4>
<%
  Applicant applicant = new Applicant();
  Principal principal = new Principal();
%>
<%
out.println("恢复至:"+String.valueOf(applicant.setState("第 2 版"))); //改变申请人对象的状态
principal.saveProcessor(applicant.createProcessor());//创建备忘录对象，并存储申请人对象状态
%>
<br >
  <%
  out.println("将当前版本:"+String.valueOf(applicant.setState("覆盖"))) ; //变更申请人对象状态
  applicant.restoreProcessor(principal.retrieveProcessor()); //恢复申请人对象状态
  %>
 <br >
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
恢复至：第 2 版
将当前版本：覆盖
```

第 26 章　Observer（观察者）模式

26.1　概述

所谓观察者（Observer）模式是指："定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。"[1]

粗看其意图，大家可能有点迷惑不解。其实，我们可以把它简化。

所谓观察者模式的定义，可理解为"如果被观察的对象产生一些变更，则会通告观察者去具体执行相关的操作"。

为了便于读者进一步理解观察者模式，本章通过模式的结构与实例进行解说。

1. 结构（如图 26.1 所示）

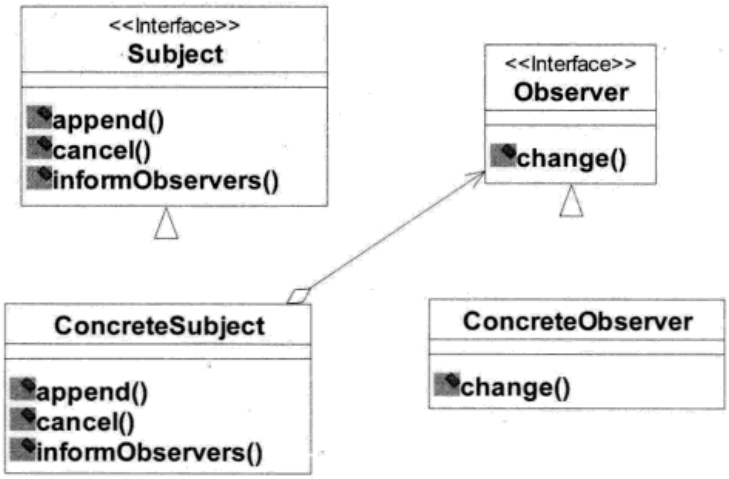

图 26.1

- 抽象主题（Subject）角色：主题角色可将全体观察者对象的引用存入至某一个列表中，并且任意主题拥有的观察者数量不受限制。主题提供某一接口可用于添加或撤销观察者对象。主题角色我们也可称之为抽象被观察者(Observable)角色；通常可通过某一抽象类或某一接口的方式去实现。
- 抽象观察者（Observer）角色：通过定义某一类接口帮助全体具体观察者，在获取通知信息的同时及时改变自身。
- 具体主题（ConcreteSubject）角色：通过将对具体观察者对象产生作用的内部状态进行保

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，194

存。当此类状态产生变化时会对它的观察者发送某类通知信息；并且具体主题角色我们也可称之为具体被观察者角色。

- 具体观察者（ConcreteObserver）角色：存贮某一面向具体主题对象的引用以及某一与主题状态类别相同的状态。遵照抽象观察者角色的需求去变更自身的接口，从而实现具体观察者角色状态与主题状态的统一。

2. 实例

股票交易软件中，每个股票投资者都可设置价格来买卖股票。交易系统观察投资者的价格，进行处理。每次系统统计一个投资者新的出价都将对当前的股价产生影响，并且通知所有投资人以方便他们设置新的买卖价格，如图 26.2 所示。

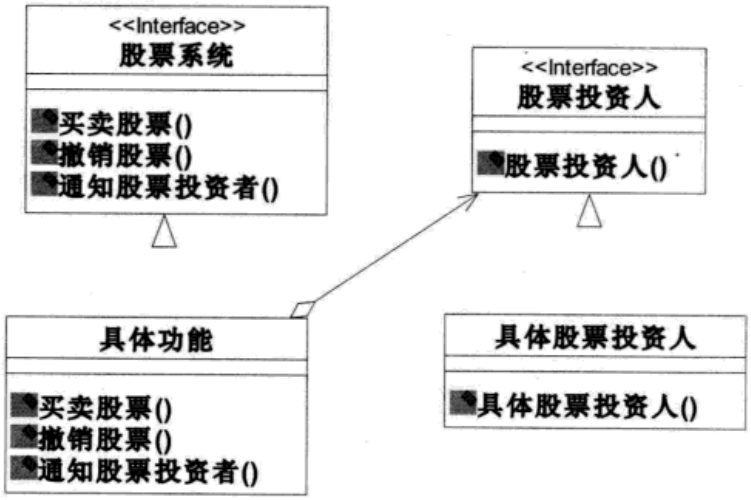

图 26.2

股票系统即抽象主题（Subject）角色，具体功能即具体主题（ConcreteSubject）角色，股票投资人即抽象观察者（Observer）角色，具体股票投资人即具体观察者（ConcreteObserver）角色。

26.2 优势与时机

观察者模式具备的优势：

- 通过将表示层与数据逻辑层的层次关系各自独立出来，使更新消息的传递机制更加稳定。
- 观察者与被观察者通过某一抽象的耦合关系进行联系，使被观察者无须了解某一具体的观察者。

基于此，我认为可以在以下情形下采用观察者模式进行软件设计与实施。

- 如果某一对象在更改自身的同时，还需同时更改其他数量无法估计的对象时。
- 为了降低各个程序层次之间的耦合度，抽象模型的两个方面，被封装在各自独立的对象中，使两者的使用更加灵活时。

26.3 提升方向

抽象出来的接口类使每一个外观对象均需继承它，并且具体观察者未必是“更新”方法所需调用者。因而，会造成一定程序资源上的浪费。当然，我们也可用 Adapter 模式来处理外观对象未继承抽象类或接口有问题，并且不想变动外观对象类的情况。但是，设计会更加复杂，出错的可能性也将增大。

26.4 应用情境——邮箱书讯通知

打开我经常使用的邮箱，过个三五天总是会收到一些在线书城的书讯通知。究其原因，目前在线书城网站推出最新书籍报价主动告知政策，凡是用电子邮箱注册登记的用户，均可免费享受 E-mail 书类价格实时告知服务。对于我这种喜爱研读计算机书籍的人而言，是一种好事。此情形从观察者模式的定义角度分析，十分符合其要求。本应用情境以某书从 100 元降为 85 元为例，如图 26.3 所示。

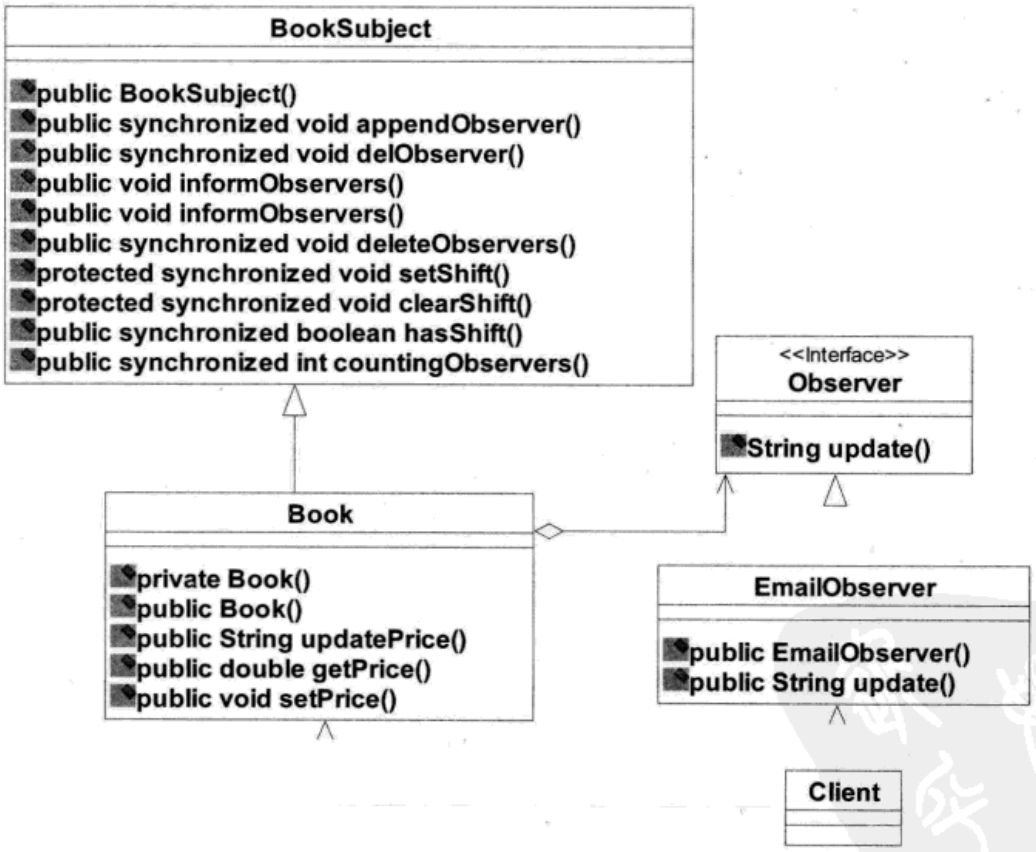

图 26.3

本应用情景的工程名为“程序 26.4.1”；源代码如下所示。

（1）被观察者基类

```
package model.observer;

import java.util.ArrayList;
import java.util.List;

public abstract class BookSubject {
      //状态
      private boolean shift = false;

      //观察者集合
      private List<Observer> observers;

      public BookSubject() {
        observers = new ArrayList<Observer>(0);
      }

      //添加一个观察者
      public synchronized void appendObserver(Observer observer) {
        if (observer == null)
          throw new NullPointerException();
        if (!observers.contains(observer)) {
          observers.add(observer);
        }
      }

      //删除一个观察者
      public synchronized void delObserver(Observer observer) {
        observers.remove(observer);
      }

      //通知观察者
      public void informObservers() {
        informObservers(null);
      }

      //触发连接到这个对象的所有观察者
      public void informObservers(Object argObj) {
        synchronized (this) {
          if (!shift)
            return;
          clearShift();
        }

        for (Observer observer : observers) {
          observer.update(this, argObj);
        }
      }
```

```
    //清除观察名单，该对象不再有任何观察员。
    public synchronized void deleteObservers() {
      observers.clear();
    }

    //标志着观察对象为已更改
    protected synchronized void setShift() {
        shift = true;
    }

    //说明此对象已不再改变，或者它已通知其所有观察者的最新变化
    protected synchronized void clearShift() {
        shift = false;
    }

    public synchronized boolean hasShift() {
      return shift;
    }

    public synchronized int countingObservers() {//统计观察者
      return observers.size();
    }
  }
```

（2）被观察者子类

```
package model.observer;

public class Book extends BookSubject {
    // 构造器
    private Book() {
    }

    // 构造器
    public Book(double value) {
        this.value = value;
    }

    // 书籍价格
    private double value;

    // 变更书籍价格
    public String updatePrice(double price) {
        String b=String.valueOf(price);
        if (this.value != price) {
            this.value = price;
            setShift();
```

```
        }
         informObservers();
        return b;
    }

    public double getPrice() {
        return value;
    }

    public void setPrice(double value) {
        this.value = value;
    }
}
```

（3）抽象观察者

```
package model.observer;

//观察者
public interface Observer {
      String update(BookSubject o, Object argObj);
    }
```

（4）具体观察者（E-mail 观察者）

```
package model.observer;

//email 观察者

public class EmailObserver implements Observer {

      //运用构造器为 Book 放置邮件发送观察者
      public EmailObserver(Book book) {
        book.appendObserver(this);
      }

      public String update(BookSubject book, Object argObj) {
          String a="Java 编程思想价格变为";
          String b=String.valueOf(((Book)book).getPrice());
          String c="元 ; 通过 E-mail 发送给所有邮箱注册用户.";
        System.out.println("Java 编程思想价格变为 "
            + ((Book)book).getPrice()
            + "元 ; 通过 E-mail 发送给所有邮箱注册用户.");
        return a+b+c;
      }
    }
```

（5）客户端（test.jsp）

```
<HTML>
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    观察者模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.observer.*" %>
<br />
<br />
<h4>
<%
//新书原价 100 元
Book book = new Book(100);
%>
<%
out.println("通过 EMAIL 发送给所有邮箱注册用户:"+String.valueOf(new
EmailObserver(book)));//电邮 observer
%>
<br >
<%
out.println("Java 编程思想价格变为:"+String.valueOf(book.updatePrice(85)+"元
")); //增加产品价格
%>
 <br >
</h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
通过 E-mail 发送给所有邮箱注册用户:model.observer.EmailObserver@1248513
Java 编程思想价格变为:85.0 元
```

附注：1248513 是随机数，每次运行时数据各不相同。

26.5 观察者模式与 Spring

Spring 框架有别于以往传统的观察者模式，它可在 bean factory 中通过配置实现此模式。Bean factory 所配备的 xml 配置文件如下所示。

```
<?xml version="1.0" encoding="GBK"?>
   <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
```

```
        <beans>
        <bean id="replacer1" class="ReplacerSriber1"/>
        <bean id="replacer2" class="ReplacerSriber2"/>
        <bean id="info" class="Speech"/>
        <bean id="ligatureInfoReplacer"

class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="targetObject"><ref local="info"/></property>
        <property name="targetMethod"><value>addListener</value></property>
        <property name="arguments">
        <list>
        <ref bean="replacer1"/>
        <ref bean="replacer2"/>
        </list>
        </property>
        </bean>
        </beans>
```

此处运用了 org.springframework.beans.factory.config.MethodInvokingFactoryBean 的工厂类；将 replacer1 和 replacer2 动态的注入至 info 之中，达到了观察者在 xml 中的可配置性。

具体处理的 Java 类如下所示。

（1）测试类

```
package spring.observer;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class Test {
    public static void main(String[] args) {
        ApplicationContext ac = new FileSystemXmlApplicationContext(
                "/test/test.xml");
        Info pub = (Info) ac.getBean("pub");
        pub.setMessage("大家好啊！！！");
    }
}
```

（2）演讲类

```
package spring.observer;
public class Speech {

}
```

（3）信息类

```
package spring.observer;
public class Info {
    public void setMessage(String a) {
    }
}
```

第 27 章　State（状态）模式

27.1　概述

所谓状态模式是指“充许一个对象在其内部状态改变时改变它的行为。”[1]

粗看其定义，大家可能会有点疑惑。其实，我们可以将状态模式理解为“运用一种状态处理多种不同状态，当不同状态的算法相同而选择不同时，则需要单独解决状态的选择。即通过转化成有差异的类用于需选择的对象，并且使用相同的算法调用共用方法解决”。

为了进一步说明状态模式，本文运用结构与实例进行讲解。

1. 结构（如图 27.1 所示）

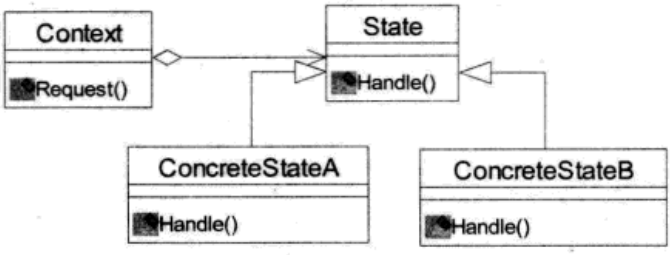

图 27.1

- Context（环境，可使用接口或抽象类）：定义客户感兴趣的接口。维护一个 ConcreteState 子类的实例，这个实例定义当前状态。
- State（状态）：定义一个接口以封装与 Context 的一个特定状态相关的行为。
- ConcreteStatesubclasses（具体状态子类）：每一子类实现一个与 Context 的一个状态相关的行为。[2]

2. 实例

自动存取款 ATM 机是一个高度精密的机电一体化装置，利用磁性代码卡或智能卡实现金融交易的自助服务。其可提供现金提取、现金存入以及现金查询、资金划拨等功能，如图 27.2 所示。

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，63

[2] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，63

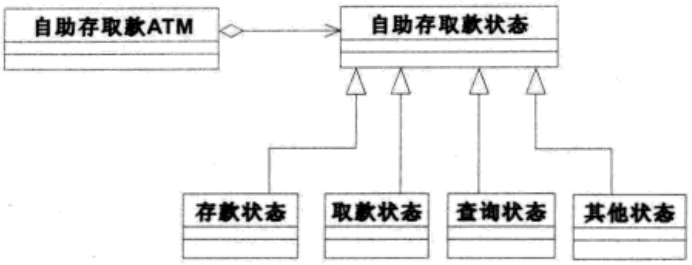

图 27.2

自助存取款 ATM 即 Context，自助存取款状态即 State，存款状态、取款状态、查询状态、其他状态即 ConcreteStatesubclasses。

27.2 优势与时机

状态模式具备的优势：

- 使 If 或 Case 等判断语句的数量与长度得到了很好的控制。
- 具体状态与其对应的行为封装在状态角色中，使添加新的状态变得更加方便。
- 基于此，我认为可以在以下几种情形下采用状态模式进行软件设计与实施。
- 当某一对象的状态可决定其行为时，它需要在运行时依靠状态变更行为。
- 当条件判断语句十分复杂时，可将复杂的语句分为一些分支并将每一分支纳入单个独立类中，同时将对象状态作为某一对象，此对象不因其他对象变化而变化。

27.3 提升方向

状态如果过多，则每一状态对应的具体状态类会结构分散并且代码较难理解。而且，将导致出现大量的状态类，使维护的工作量大大增加。

27.4 应用情境——报警系统设计

如果我们设计一个报警系统，当前只有高（high）、中（middle）、低（low）三种声音。对于一个经验不足的开发人员而言，他也许会把类图和代码编成如图 27.3 所示。

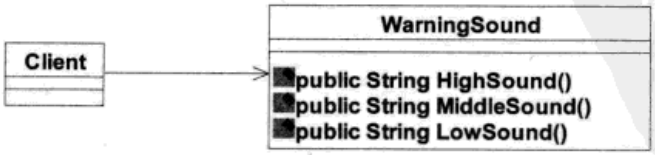

图 27.3

本应用情景的工程名为“程序 27.4.1”，源代码如下所示。

（1）报警声音

```
package model.state1;

//高high、中middle、低low
public class WarningSound{
    public String HighSound()
    {
     String HighSound="高音";
        System.out.println("高音");
        return HighSound;
   }
    public String MiddleSound()
    {
         String MiddleSound="中音";
        System.out.println("中音");
       return MiddleSound;
    }
    public String LowSound()
    {
        String LowSound="低音";
        System.out.println("低音");
        return LowSound;
    }
}
```

（2）客户端（test.jsp）

```
<HTML>
<!--
    @author jianghc
-->
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    未运用状态模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.state1.*" %>
<br />
<br />
<h4>
<%
   WarningSound ws=new WarningSound();
    out.print("不正常:"+ws.MiddleSound());
 %><br>
<%
```

```
    out.print("重大异常:"+ws.HighSound());
 %><br>
 <%
    out.print("正常:"+ws.LowSound());
 %><br>
</h4>
<br />
<br />
<br />
<h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
不正常：中音
重大异常：高音
正常：低音
```

在类图‘图 27.3’设计的基础上完成了设计的功能，但是如果状态（高中低声音）增加了呢？因而，程序 27.4.1 并不适合，适合的程序如图 27.4 所示。

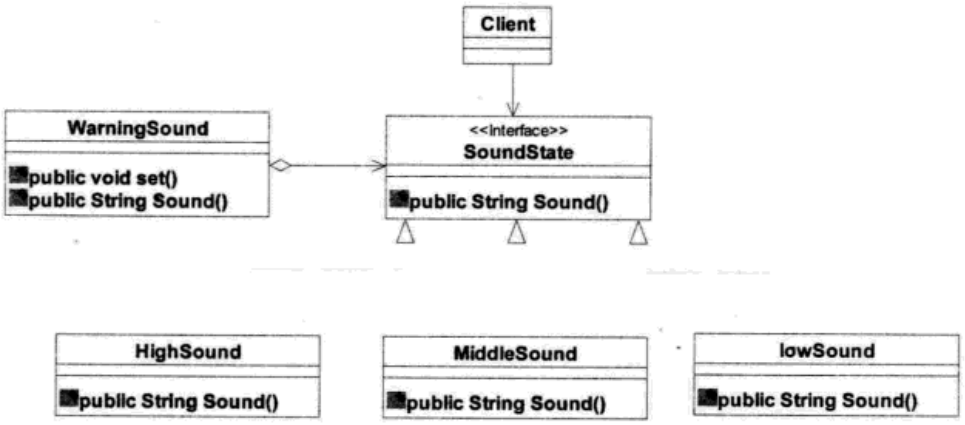

图 27.4

本应用情景的工程名为“程序 27.4.2”，源代码如下所示。

（1）报警声音类（Context 环境类）

```
package model.state2;

public class WarningSound { //报警声音类-Context 环境类
    private SoundState state = new MiddleSound();
    public void setState(SoundState state) {
        this.state = state;
    }
    public String Sound() {
        return state.Sound();
```

```
    }
}
```

（2）状态接口（声音）

```
package model.state2;

interface SoundState{
    public String Sound();
}
```

（3）具体状态子类（高音）

```
package model.state2;

public class HighSound implements SoundState{
    public String Sound(){
     String HighSound="高音";
        System.out.println("高音");
        return HighSound;
}
}
```

（4）具体状态子类（中音）

```
package model.state2;

public class MiddleSound implements SoundState{
    public String Sound(){
         String MiddleSound="中音";
        System.out.println("中音");
       return MiddleSound;
}
}
```

（5）具体状态子类（低音）

```
package model.state2;

public class LowSound implements SoundState{
    public String Sound(){
    String LowSound="低音";
    System.out.println("低音");
    return LowSound;
}
}
```

（6）客户端（test2.jsp）

```
<HTML>
<!--
    @author jianghc
-->
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    状态模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.state2.*" %>
<br />
<br />
<h4>
<%
    WarningSound warning = new WarningSound();
%><br>
<%
out.println("不正常:"+warning.Sound());
%><br>
<%
warning.setState(new HighSound());
out.println("重大异常:"+warning.Sound());
%><br>
<%
warning.setState(new LowSound());
out.println("正常:"+warning.Sound());
%>
</h4>
<br />
<br />
<h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
不正常：中音
重大异常：高音
正常：低音
```

第 28 章　Strategy（策略）模式

28.1　概述

所谓策略模式指“对算法的包装，把使用算法的责任和算法本身分割开来，委派给不同的对象管理。策略模式通常把一个系列的算法包装到一系列的策略类里面，作为一个抽象策略类的子类”。

粗看其定义，大家可能会有点疑惑。其实，我们可以将策略模式理解为“定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。”[1]

为了进一步说明策略模式，本文运用结构与实例进行讲解。

1. 结构

策略模式的结构，如图 28.1 所示。

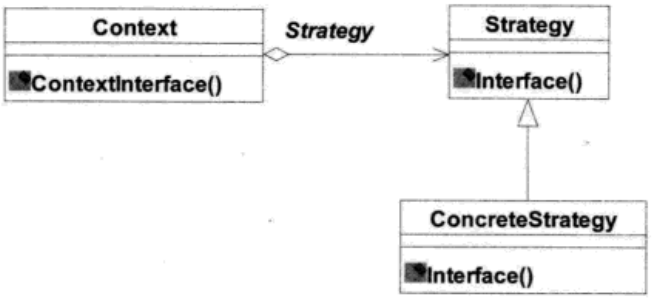

图 28.1

- 环境（Context）角色：持有一个 Strategy 类的引用。
- 抽象策略（Strategy）角色：这是一个抽象角色，通常由一个接口或抽象类实现此角色给出所有的具体策略类所需的接口。
- 具体策略（ConcreteStrategy）角色：包装了相关的算法或行为。[2]

2. 实例

对于上班族而言，出行离不开各类交通工具。有人选择私家车，有人选择出租车，有人选择公交车，有人选择地铁等等。其出行策略各不相同，以策略模式为例，如图 28.2 所示。

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，208

[2] 阎宏. Java 与模式. 北京：电子工业出版社，2002，622

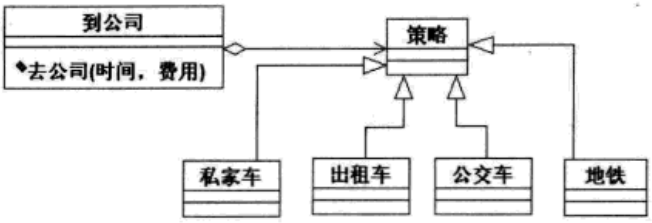

图 28.2

28.2 优势与时机

策略模式具备的优势：

- 为相关算法族的管理提供了相应的方法和手段。即运用策略类将通用代码移植至父类内部，从而去除冗余代码。
- 通过提供一种可替代继承的方法，从而实现继承的代码可重用性优点。并且，进一步提升代码的可扩展性。
- 无须编写任何多重条件转移语句，从而使代码更加简洁。
- 基于此，我认为可以在以下几种情形下采用策略模式进行软件设计与实施。
- 当软件系统的类过多并且类之间的区别在于其行为时，即动态的使某一对象在多个行为中选择一种。
- 当软件系统必须动态的在多个算法中选择一个时，即使用共性的接口并使客户端可选择某一具体算法类以及持有某一数据类型为抽象算法类的对象即可。
- 当软件系统的算法所运行的数据无须客户端了解时，即阻断客户端无须了解的数据即可。

28.3 提升方向

对于客户端而言，它需要了解全部策略类，并且根据需要自我选择某一策略类去使用。如果策略类过多，则需要运用享元模式去缩减对象的数量。

由于策略接口的各个算法地位是相同的，那将导致在程序运行时只能调用某一算法。为解决此类问题，如在处理大型商场搞折扣加返现金的活动时，可运用装饰模式之类的方式处理嵌套使用多个算法的问题。

28.4 应用情境——数据预测

某电力公司做软科学课题项目，要求对未来五年的用电客户量进行预测。预测的数据以 2009 年 1 至 12 月的用电客户数据为基础。

基于对各种预测模型的了解，研究人员张玉重点选择了三种预测策略，如图 28.3 所示。

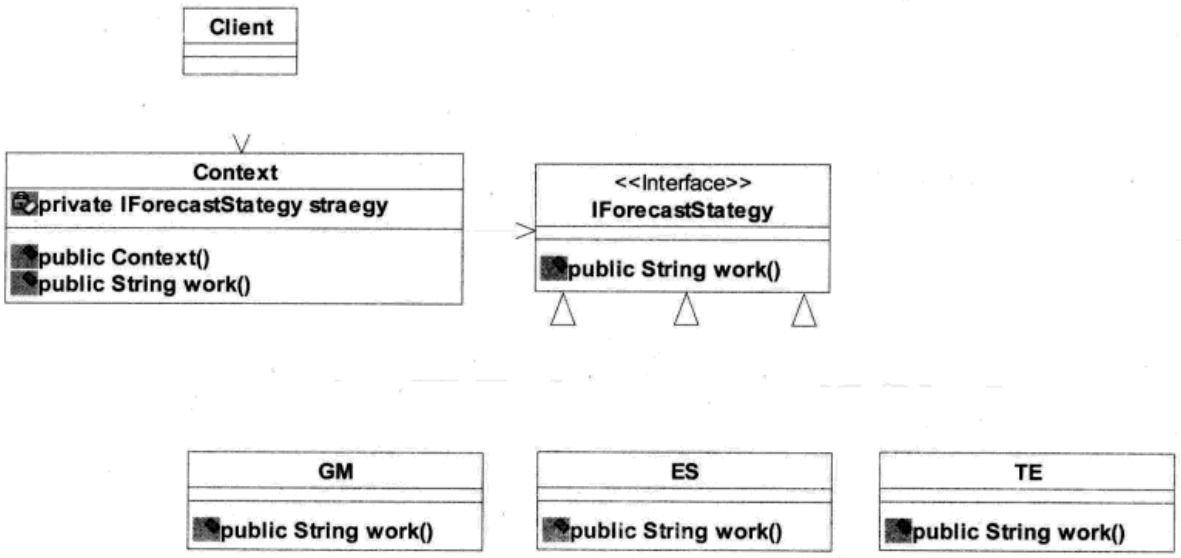

图 28.3

本应用情景的工程名为“程序 28.4.1”，源代码如下所示。

（1）环境类

```
package model.strategy;

/**
 * @author jackjiang
 */
public class Context {
    // 构造函数，选择所需预测策略
    private IForecastStategy straegy;
    public Context(IForecastStategy forecastStategy) {
        this.straegy = forecastStategy;
    }
    // 运用预测策略，开始。
    public String work() {
        return  this.straegy.work();
    }
}
```

（2）抽象策略（预测策略接口）

```
package model.strategy;

/**
 * @author jackjiang 预测策略接口
 */
public interface IForecastStategy {
    //三个预测模型均为某一可执行算法
    public String work();
```

```
}
```

（3）具体策略（灰色模型，GM）

```
package model.strategy;

/**
 * @author jackjiang
 */
public class GM implements IForecastStategy {
    public String work() {
    String ES="灰色模型:将已知的数据序列按照某种规则构成动态或非动态的白色模块，再按照某种变化、解法来求解未来的数据。";
    System.out.println("灰色模型:将已知的数据序列按照某种规则构成动态或非动态的白色模块，再按照某种变化、解法来求解未来的数据。");
    return ES;
    }
}
```

（4）具体策略（指数平滑法 Exponential Smoothing，ES）

```
package model.strategy;

/**
 * @author jackjiang
 */
public class ES implements IForecastStategy {
    public String work() {
    String ES="指数平滑法：以某种指标的本期实际数和本期预测数为基础，引入一个简化的加权因子，即平滑系数，以求得平均数！";
    System.out.println("指数平滑法：以某种指标的本期实际数和本期预测数为基础，引入一个简化的加权因子，即平滑系数，以求得平均数！");
    return ES;
    }
}
```

（5）具体策略（趋势外推法 Trend extrapolation，即 TE）

```
package model.strategy;

/**
 * @author jackjiang
 */
public class TE implements IForecastStategy {
    public String work() {
        String TE="趋势外推法：根据过去和现在的发展趋势推断未来的一类方法的总称！";
```

```
            System.out.println("趋势外推法：根据过去和现在的发展趋势推断未来的一类方法的
总称！");
            return TE;
        }
    }
```

（6）客户端（test1.jsp）

```
<HTML>
<!--
    @author jianghc
-->
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    策略模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.strategy.*" %>
<br />
<br />
<h4>
<%
Context context;
 %><br>
 <%
//first
    out.print("first, GM 即灰色模型。");
 %><br>
  <%  context = new Context(new GM()); //取得灰色模型
    out.print(context.work());//运行
    %><br>
 <%
//second
 out.print("second, ES 即指数平滑法。");
 %><br>
  <%  context = new Context(new ES());
  out.print(context.work());//运行
  %><br>
 <% //third
 out.print("third, TE 即趋势外推法。");
  %><br>
   <%  context = new Context(new TE());
   out.print(context.work());//运行
   /*
    * 由于预测模型过多，张玉重要了解了三个。
    */
    %><br>
```

```
<br>
</h4>
<br />
<br />
<br />
<h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
first，GM 即灰色模型。
灰色模型:将已知的数据序列按照某种规则构成动态或非动态的白色模块，再按照某种变化、解法来求解未来的数据。
second，ES 即指数平滑法。
指数平滑法：以某种指标的本期实际数和本期预测数为基础，引入一个简化的加权因子，即平滑系数，以求得平均数！
third，TE 即趋势外推法。
趋势外推法：根据过去和现在的发展趋势推断未来的一类方法的总称！
```

28.5 模式扩展

当电力公司领导要求添加第四种预测模型时，该怎么扩展最便捷呢？

合适的方式是，新增加一种预测模型的策略实现，然后通过实现抽象策略接口来扩展。在遵循开闭原则的基础下，只需在客户端调用最新的相关内容与策略实现即可，并且已有的实现均无须修改。

增加具体策略 BP 类。

```
package model.strategy;

    public class BP implements IForecastStategy {
        public String work() {
        String BP="BP 神经网络算法：是一种按误差逆传播算法训练的多层前馈网络，是目前应用最广泛的神经网络模型之一";
        System.out.println("BP 神经网络算法：是一种按误差逆传播算法训练的多层前馈网络，是目前应用最广泛的神经网络模型之一！");
        return BP;
        }
    }

修改客户端(test2.jsp)
<HTML>
<!--
    @author jianghc
```

```
-->
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    策略模式扩展
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.strategy2.*" %>
<br />
<br />
<h4>
<%
Context context;
 %><br>
 <%
//four
    out.print("four, BP 即 BP 神经网络算法。");
 %><br>
  <%  context = new Context(new BP()); //取得 BP 神经网络算法
    out.print(context.work());//运行
    %><br>
<br>
</h4>
<br />
<br />
<br />
<h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
four, BP 即 BP 神经网络算法。
```

BP 神经网络算法是一种按误差逆传播算法训练的多层前馈网络，是目前应用最广泛的神经网络模型之一。

28.6 策略模式与桥接模式

策略模式与桥接模式比较相近，但两者为处理不同类型的问题设计。

表 28.1 策略模式与桥接模式的相同与区别之处

模式名	策略模式	桥接模式
是否面向接口编程	是	是
模式范围	属于行为型模式	属于结构型模式
强调重点	1. 强调行为 2. 强调算法可替代	1. 强调结构 2. 强调抽象与实现的分离

28.7 策略模式与 Struts

当前，在 Web 应用程序的开发过程中，对表现层、逻辑层、数据层等对象的校验无处不在。这中间必然会产生一些冗余程序，使系统的上线与维护增加了不少工作量。

为了提高编码的生产力，David Winterfeldt 研制了 Validator 框架，并于 2002 年 11 月初进行了发布与使用。此框架的产生，大大提高了校检代码重复使用率的效率。

例如，我们既能在某个 Validator 类中创建某个单独的验证规则，也可将多个规则组汇总为一定规模的规则集合。并且，Validator 类中能够清晰地创建一定数量的验证规则组，使其具体某类规则无须涵盖客户端相关的任意状态信息。

如此一来，包含了 Validator 的 Struts 框架具备了策略模式功能，当 Struts 的多种页面组件（如多选框、表单、列表单元格）调用了数据库的相关数据进行校验时，在 Validator 框架的 validator-rules.xml 和 validation.xml 配置文件中进行相应校验配置信息的维护即可。

第 29 章　TemplateMethod（模板方法）模式

29.1　概述

所谓模板方法模式是指："定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。"[1]

粗看其定义，大家也许会有一些疑惑，其实我们可以这样来理解："通过在某一类中定义某一算法，即可将此算法的某些要求置入子类中实现。"当然我们也可以这样认为："设计某一抽象类时可将逻辑控制的一部分内容用具体方法实现，并且定义某些抽象方法以方便子类去实现逻辑控制未实现的另一部分。此时，各种不同的子类可运用不同的方法实现抽象方法，从而保证未实现逻辑的差异化实现。"

为了进一步说明模板模式，本文运用结构与实例进行讲解。

1. 结构（如图 29.1 所示）

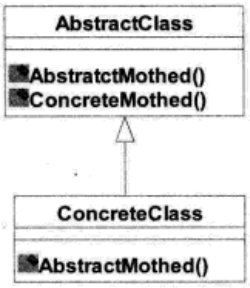

图 29.1

- 抽象模板角色（Abstract Template）：定义了一个或多个抽象操作以便让子类实现，定义并实现了一个模板方法。
- 具体模板角色（Concrete Template）：实现父类所定义的一个或多个抽象方法。每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色都可以给出这些抽象方法的不同实现。[2]

2. 实例

家具设计师在设计新家具时会运用模板方法。比如一个沙发，它包括木架、海绵、面料（皮和布）、五金、靠垫。只有在客户提出个性化需求时，才产生不同的沙发样式，如图 29.2 所示。

---

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，214

[2] 阎宏. Java 与模式. 北京：电子工业出版社，2002，642

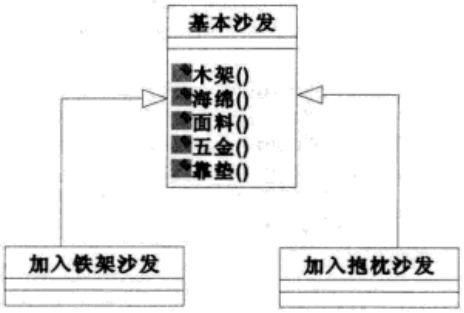

图 29.2

29.2 优势与时机

模板模式具备的优势：

- 将不变代码置于父类，将可变代码置于子类，有利于提高代码的重用性。
- 可根据需要在子类中实现某类算法，并对父类无任何影响。
- 可在子类中实现一些独特的方法，有利于代码的扩展。

基于此，我认为可以在以下几种情形下采用模板模式进行软件设计与实施。

- 需要一次就实现某一算法的不变部分，并由子类来实现代码的可变部分时。
- 为防止代码重复而将公共方法置于公共类时。
- 需要对代码的扩展进行监控时。（原因在于，模板方法仅限于在特定点调用 Hook 操作）

29.3 提升方向

当算法的框架变更时，将导致对抽象类进行修改。每个具体实现均需定义子类，其结果将使类的数目增加较频。

29.4 应用情境——办公室故事

2011 年 9 月 2 号，在某软件公司办公室，设计师小王刚从外面回来，准备喝口水时。就被部门助理一个电话叫到业务经理老张的办公室。

当小王坐在老张办公桌对面了解了情况后，也不禁心情大好。原来老张接到民政局的一个软件开发项目，此项目小王手上已有十分相近的原型。根据客户的需求，目前只需要做成两个版本即可。这两个版本，一个给民政局领导用，一个给普通员工用。所谓两个版本均继承至原型，无非是版本 1、版本 2 的页面显示的内容有点不同而已。

此项目成本低利润高，小王从中可获得不少的奖金。

针对这一情况，运用模板模式的设计类图如图 29.3 所示。

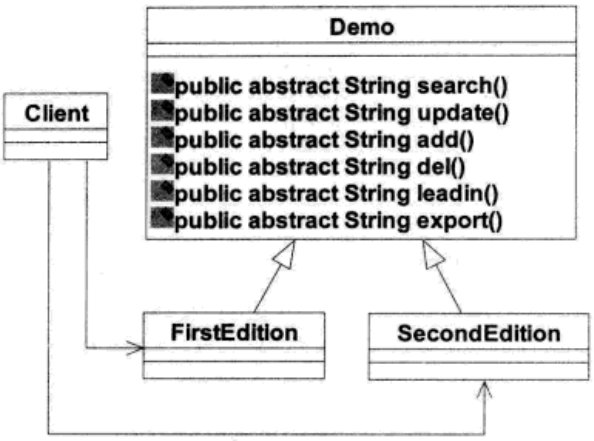

图 29.3

本应用情景的工程名为“程序 29.4.1”，源代码如下所示。

（1）模板抽象类

```
package model.templateMethod1;

/**
 * @author jackjiang
 */
public abstract class Demo {

    /*
     *设计图所含主体内容
     */
    public abstract String search();//查询

    public abstract String update();//修改

    public abstract String add();//添加

    public abstract String del();//删除

    public abstract String leadin();//导入

    public abstract String export();//导出

}
```

（2）第一具体模板类

```
package model.templateMethod1;

/**
 *@author jackjiang
```

```
 */
public class FirstEdition extends Demo {
    @Override
    public String add() {
        String add="新增页面类型 1";
        System.out.println("新增页面类型 1");
        return add;
    }
    @Override
    public String del() {
        String del="删除页面类型 1";
        System.out.println("删除页面类型 1");
        return del;
    }
    @Override
    public String export() {
        String export="导出页面类型 1";
        System.out.println("导出页面类型 1");
        return export;
    }
    @Override
    public String leadin() {
        String leadin="导入页面类型 1";
        System.out.println("导入页面类型 1");
        return leadin;
    }
    @Override
    public String search() {
        String search="查询页面类型 1";
        System.out.println("查询页面类型 1");
        return search;
    }
    @Override
    public String update() {
        String update="修改页面类型 1";
        System.out.println("修改页面类型 1");
        return update;
    }
}
```

（3）第二具体模板类

```
package model.templateMethod1;

/**
 * @author jackjiang
 */
public class SecondEdition extends Demo {
    @Override
```

```
    public String add() {
        String add="新增页面类型 2";
        System.out.println("新增页面类型 2");
        return add;
    }
    @Override
    public String del() {
        String del="删除页面类型 2";
        System.out.println("删除页面类型 2");
        return del;
    }
    @Override
    public String export() {
        String export="导出页面类型 2";
        System.out.println("导出页面类型 2");
        return export;
    }
    @Override
    public String leadin() {
        String leadin="导入页面类型 2";
        System.out.println("导入页面类型 2");
        return leadin;
    }
    @Override
    public String search() {
        String search="查询页面类型 2";
        System.out.println("查询页面类型 2");
        return  search;
    }
    @Override
    public String update() {
        String update="修改页面类型 2";
        System.out.println("修改页面类型 2");
        return update;
    }
}
```

（4）客户端（test1.jsp）

```
<HTML>
<!--
    @author jianghc
-->
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    模版方法模式
    </TITLE>
```

```
    </HEAD>
    <BODY BGCOLOR="white">
    <%@ page pageEncoding="GB2312" language="java" import="model.
templateMethod1.*" %>
    <br />
    <br />
    <h4>
    <%
    // 版本 1，给领导用的。
    FirstEdition f1 = new FirstEdition();
     %><br>
      <%
     out.print("版本 1，给领导用:");
     %><br>
     <%
     out.print(f1.add()+" ");
     out.print(f1.del()+" ");
     out.print(f1.export()+" ");
     out.print(f1.search()+" ");
     out.print(f1.leadin()+" ");
     out.print(f1.update()+" ");
     %><br>
      <% //版本，给员工用的。
      SecondEdition f2 = new SecondEdition();
     %><br>
       <%
     out.print("版本 2，给员工用:");
     %><br>
     <%
     out.print(f2.add()+" ");
     out.print(f2.del()+" ");
     out.print(f2.export()+" ");
     out.print(f2.search()+" ");
     out.print(f2.leadin()+" ");
     out.print(f2.update()+" ");
     %><br>
    </h4>
    <br />
    <br />
    <br />
    <h4>
    </BODY>
    </HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
版本 1，给领导用:
新增页面类型 1 删除页面类型 1 导出页面类型 1 查询页面类型 1 导入页面类型 1 修改页面类型 1
版本 2，给员工用:
```

新增页面类型 2 删除页面类型 2 导出页面类型 2 查询页面类型 2 导入页面类型 2 修改页面类型 2

29.5 模式扩展

当针对民政局领导与普通员工的两个版本开发好之后，民政局的信息科长要求版本的发布，需要支持 war 与 ear 两种格式。此时，我们可结合模板方法与装饰模式进行设计，如图 29.4 所示。

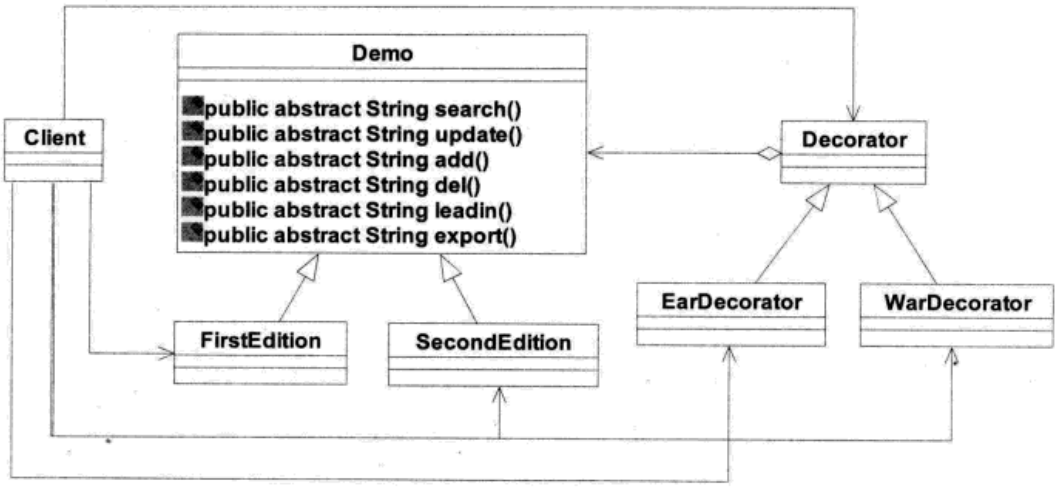

图 29.4

本应用情景的工程名为“程序 29.4.2”，源代码如下所示。

（1）模板抽象类

与程序 29.4.1 一致。

（2）第一具体模板类

与程序 29.4.1 一致。

（3）第二具体模板类

与程序 29.4.1 一致。

（4）抽象装饰类

```
package model.templateMethod2;

abstract public class Decorator extends Demo{//发布抽象装饰类
    private Demo demo;
    public Decorator(Demo demo){
        super();
        this.demo=demo;
    }
    @Override
    public String add() {
```

```
        return demo.add();
    }
    @Override
    public String del() {
        return demo.del();
    }

    @Override
    public String export() {
        return demo.export();
    }
    @Override
    public String leadin() {
        return demo.leadin();
    }
    @Override
    public String search() {
        return demo.search();
    }
    @Override
    public String update() {
        return demo.update();
    }
}
```

（5）具体装饰类 1（WAR 发布）

```
package model.templateMethod2;

public class WarDecorator extends Decorator{
//war 发布
    public WarDecorator(Demo demo) {
        super(demo);
    }
    @Override
    public String add() {
        String add="将新增页面发布为 war 格式";
        super.add();
        System.out.println("将新增页面发布为 war 格式");
        return add;
    }
    @Override
    public String del() {
        String del="将删除页面发布为 war 格式";
        super.del();
        System.out.println("将删除页面发布为 war 格式");
        return del;
    }
    @Override
```

```
    public String export() {
        String export="将导出页面发布为 war 格式";
        super.export();
        System.out.println("将导出页面发布为 war 格式");
        return export;
    }
    @Override
    public String leadin() {
        String leadin="将导入页面发布为 war 格式";
        super.leadin();
        System.out.println("将导入页面发布为 war 格式");
        return leadin;
    }
    @Override
    public String search() {
        String leadin="将查询页面发布为 war 格式";
        super.leadin();
        System.out.println("将查询页面发布为 war 格式");
        return leadin;
    }
    @Override
    public String update() {
        String leadin="将修改页面发布为 war 格式";
        super.leadin();
        System.out.println("将修改页面发布为 war 格式");
        return leadin;
    }
}
```

（6）具体装饰类 2（EAR 发布）

```
package model.templateMethod2;

public class EarDecorator extends Decorator{
    //ear 发布
        public EarDecorator(Demo demo) {
            super(demo);
        }
        @Override
        public String add() {
            String add="将新增页面发布为 ear 格式";
            super.add();
            System.out.println("将新增页面发布为 ear 格式");
            return add;
        }
        @Override
        public String del() {
            String del="将删除页面发布为 ear 格式";
            super.del();
```

```
            System.out.println("将删除页面发布为 ear 格式");
            return del;
        }
        @Override
        public String export() {
            String export="将导出页面发布为 ear 格式";
            super.export();
            System.out.println("将导出页面发布为 ear 格式");
            return export;
        }
        @Override
        public String leadin() {
            String leadin="将导入页面发布为 ear 格式";
            super.leadin();
            System.out.println("将导入页面发布为 ear 格式");
            return leadin;
        }
        @Override
        public String search() {
            String leadin="将查询页面发布为 ear 格式";
            super.leadin();
            System.out.println("将查询页面发布为 ear 格式");
            return leadin;
        }
        @Override
        public String update() {
            String leadin="将修改页面发布为 ear 格式";
            super.leadin();
            System.out.println("将修改页面发布为 ear 格式");
            return leadin;
        }
    }
```

（7）客户端（test2.jsp）

```
<HTML>
<!--
    @author jianghc
-->
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    模版方法模式扩展
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.templateMethod2.*" %>
<br />
<br />
```

```
<h4>
<%
//版本 1，给领导用的,WAR 发布
Decorator demod1=new WarDecorator(new FirstEdition());
%><br>
 <%
out.print("版本 1，给领导用的,WAR 发布。");
%><br>
<%
out.print(demod1.add()+" ");
out.print(demod1.del()+" ");
out.print(demod1.export()+" ");
out.print(demod1.search()+" ");
out.print(demod1.leadin()+" ");
out.print(demod1.update()+" ");
%><br>
 <% //版本 2，给员工用的,EAR 发布。
 Decorator demod2=new EarDecorator(new SecondEdition());
%><br>
  <%
out.print("版本 2，给员工用的,EAR 发布。");
%><br>
<%
out.print(demod2.add()+" ");
out.print(demod2.del()+" ");
out.print(demod2.export()+" ");
out.print(demod2.search()+" ");
out.print(demod2.leadin()+" ");
out.print(demod2.update()+" ");
%><br>
</h4>
<br />
<br />
<br />
<h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
版本 1，给领导用的，WAR 发布。
将新增页面发布为 war 格式 将删除页面发布为 war 格式 将导出页面发布为 war 格式 将查询页面发布为 war 格式将 导入页面发布为 war 格式 将修改页面发布为 war 格式
版本 2，给员工用的，EAR 发布。
将新增页面发布为 ear 格式 将删除页面发布为 ear 格式 将导出页面发布为 ear 格式 将查询页面发布为 ear 格式 将导入页面发布为 ear 格式 将修改页面发布为 ear 格式
```

29.6 模板方法模式与策略模式

模板模式与策略有一定相似之处，但两者实现方式有一定区别，如表 29.1 所示。

表 29.1 模板模式与策略模式的相同与区别之处

模式名	模板方法模式	策略模式
是否行为型模式	是	是
是否算法和上下文解耦	是	是
如何代码重用	运用继承来实现代码重用	运用委托来实现代码重用。并且委托比继承具有更大的灵活性
重点	封装算法骨架	分离并封装算法实现

29.7 模板方法模式在 Spring、Hibernate 中的应用

当前，在 Spring 框架中，其外接扩展（如 JNDI、JMS 以及 Hiberate 等有关扩展）都采用模板方法模式进行扩展，即对模板方法模式进行了一定的创新。

本节运用模板方法模式在 Hibernate 中进行代码的优化，以达到提高系统扩展性与增强代码易读性之效果。例如，在 HibernateTemplate 类中经常进行数据库的增、删、改、查等操作，如果某个操作就对应继承相应的子类，那多余的代码实在过多。Spring 将 HibernateCallback 接口类与模板方法结合，即可实现模板方法模式的创新。

当然，Spring 框架模板方法模式的创新无须继承传统模板方法的抽象类，它运用了有关回调函数 HibernateCallback 接口类中相关的 doInHibernate 方法的相关隐藏类设置成参数，通过关联 HibernateTemplate 类中名为 execute()的相关模板方法进行应用。具体简要代码如下：

```
public Object oracleExecute(String oracle)
{
      return getHibernateTemplate().execute(new HibernateCallback(){
            public Object doInHibernate(Session oraclesen)
            {
                  return null;
            }
      }
}
```

第 30 章　Visitor（访问者）模式

30.1　概述

访问者模式是指："表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。"[1]

粗看其意图，大家可能有点迷惑不解。其实，我们可以把它简化。所谓访问者模式的定义，可理解为"通过保持原有程序的结构，运用增加'访问者'来实现对现有程序功能的优化"。

为了便于读者进一步理解访问者模式，本章通过模式的结构与实例进行解说。

1. 结构（如图 30.1 所示）

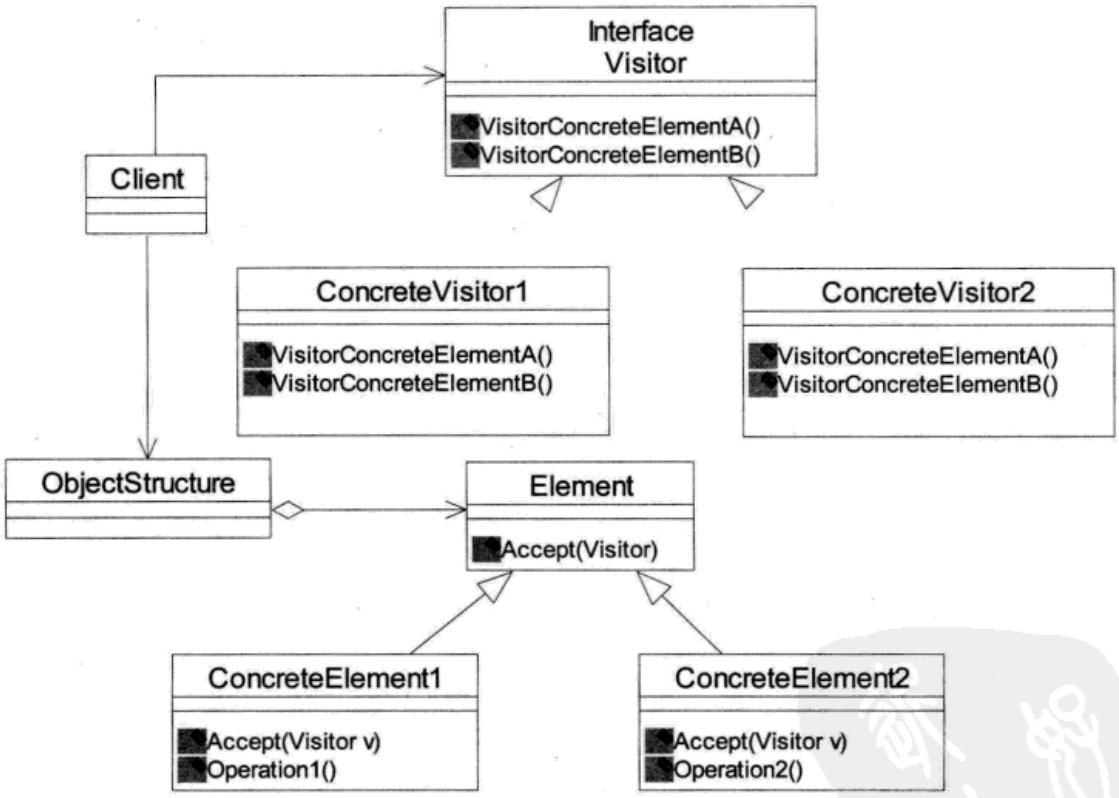

图 30.1

- 访问者角色（Visitor）：为该对象结构中具体元素角色声明一个访问操作接口。该操作接口的名字和参数标识了发送访问请求给具体访问者的具体元素角色。这样访问者就可以

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，218

通过该元素角色的特定接口直接访问它。

- 具体访问者角色（Concrete Visitor）：实现每个由访问者角色（Visitor）声明的操作。
- 元素角色（Element）：定义一个 Accept 操作，它以一个访问者为参数。
- 具体元素角色（Concrete Element）：实现由元素角色提供的 Accept 操作。
- 对象结构角色（Object Structure）：这是使用访问者模式必备的角色。它要具备以下特征：能枚举它的元素；可以提供一个高层的接口以允许该访问者访问它的元素；可以是一个复合（组合模式）或是一个集合，如一个列表或一个无序集合。[1]

2. 实例

当前，我国软件行业的发展前景十分良好。对于一家软件公司而言，其核心部门我认为是营销部与研发部。只有研发部研发出产品，营销部销售出产品，公司才能生存与壮大。基于此，根据访问者模式的原理，设计如图 30.2 所示。

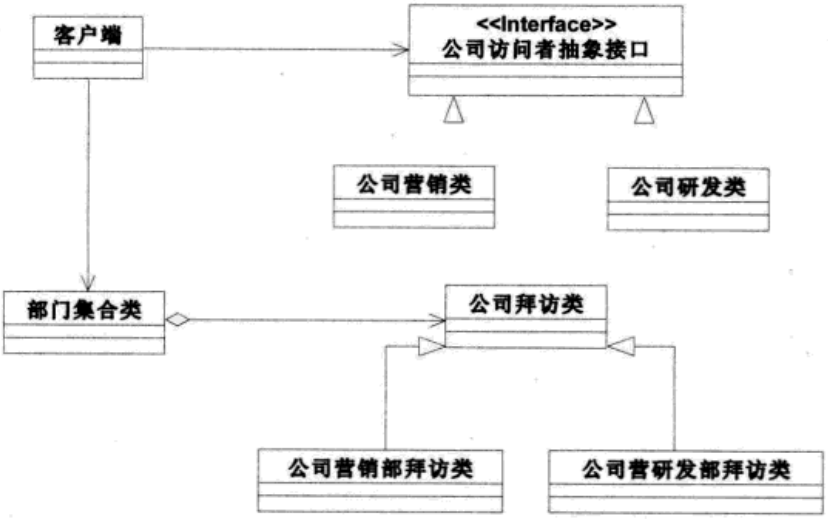

图 30.2

客户端即 Client，公司访问者抽象接口即 Visitor，公司营销类、公司研发类即 ConcreteVisitor，部门集合类即 ObjectStructure，公司拜访类即 Element，公司营销部拜访类、公司营研发部拜访类即 ConcreteElement。

30.2 优势与时机

访问者模式具备的优势：

- 使软件系统，添加操作功能变得十分简单。原因在于，添加操作功能无非添加某一新的访问类而已。
- 通过访问者对象集中管理相关行为，有利于代码的维护。

[1] Erich Gamm. 设计模式：可复用面向对象软件的基础. 李英军，马晓星，蔡敏，刘建中，译. 北京：机械工业出版社，2000，220

- 可任意访问不同等级结构的成员类，使代码的灵活性增加。
- 基于此，我认为可以在以下几种情形下采用访问者模式进行软件设计与实施。
- 当对象结构所包含的类型过多并且缺少统一接口时。
- 期望在对象结构中添加新的操作而无须修改旧有代码时。
- 当对象结构某些类型关联较多并且其实现需要经常变动时。

30.3 提升方向

使添加新的节点变得更加麻烦。原因在于，当添加新的节点时需要在抽象访问者角色中添加某一新的抽象操作，并在每一个具体访问者类中添加对应的具体操作。

影响了封装的隐蔽性。原因在于，当对象访问者调用某一节点对象的操作时会显露出某些自身存在的操作与内部状态。

具体元素角色与访问者之间可传递的信息量并不大，这将一定程度上限制访问者模式的应用。

30.4 应用情境——马和驴子的简介

马和驴子是两种属于马科的动物。它们有着各自不同的特征。本节以伊犁马与滚沙驴为例，进行特征阐述。

伊犁马："一般体高 144～148 厘米，体重 400～450 千克。它体格高大，结构匀称，头部小巧而伶俐，眼大眸明，头颈高昂，四肢强健。"

滚沙驴："一般体高 107 厘米，体重 140～190 千克，结构良好，体躯短小，腹部稍大，被毛粗刚。头大而长，额宽突，背腰平，胸窄浅，四肢结实，蹄小而圆。有青、灰、黑等多种毛色。"

针对这一情况，运用访问者模式的设计如图 30.3 所示。

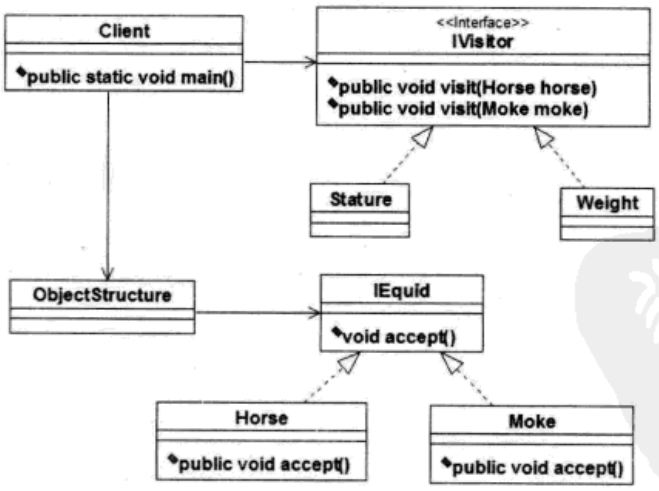

图 30.3

本应用情景的工程名为"程序 30.4.1"，源代码如下所示。

（1）访问者

```
package model.visitor1;

//访问者角色
public interface IVisitor {
    public String visit(Horse horse);
    public String visit(Moke moke);
}
```

（2）具体访问者角色-身高

```
package model.visitor1;

//具体访问者角色-身高
public class Stature implements IVisitor {
//伊犁马、滚沙驴身高的表现
    public String visit(Horse horse) {
        String a="伊犁马，一般体高 144～148 厘米。";
        System.out.println("伊犁马，一般体高 144～148 厘米。");
        return a;
    }
    public String visit(Moke moke) {
        String b="滚沙驴，一般体高 107 厘米。";
        System.out.println("滚沙驴，一般体高 107 厘米。");
        return b;
    }
}
```

（3）具体访问者角色-体重

```
package model.visitor1;

//具体访问者角色-体重
public class Weight implements IVisitor {
    //伊犁马、滚沙驴体重的表现
    public String visit(Horse horse) {
        String a="伊犁马，体重 400～450 千克。";
        System.out.println("伊犁马，体重 400～450 千克。");
        return a;
    }
    public String visit(Moke moke) {
        String b="滚沙驴，体重 140～190 千克。";
        System.out.println("滚沙驴，体重 140～190 千克。");
        return b;
    }
}
```

（4）对象结构

```
package model.visitor1;

import java.util.*;
//对象结构角色
public class ObjectStructure {
   private List<IEquid> elements = new ArrayList<IEquid>();
   public String increase(IEquid element){ //增加increase
       return String.valueOf(elements.add(element));
   }
   public String discerption(IEquid element){  //discerption 分开,移除
      return  String.valueOf(elements.remove(elements));    //移除
   }
   //遍历各种类型的具体元素，并运行其accept方法
   public String show(IVisitor visitor){
      Horse horse = null;
      String a=visitor.visit(horse);
      Moke moke = null;
      String b=visitor.visit(moke);
      for(IEquid e:elements){
         e.accept(visitor);
      }
      return a+b;
   }
}
```

（5）无素角色-马科

```
package model.visitor1;

//元素角色-马科
public interface IEquid {
        String accept(IVisitor visitor);
    }
```

（6）具体元素角色-马

```
package model.visitor1;

// 具体元素角色-马
public class Horse implements IEquid {
    public String accept(IVisitor visitor) {
       return visitor.visit(this).toString();
    }
}
```

（7）具体元素角色-驴

```
package model.visitor1;

//具体元素角色-驴
public class Moke implements IEquid {
    public String accept(IVisitor visitor) {
        return visitor.visit(this);
    }
}
```

（8）客户端（test1.jsp）

```
<HTML>
<!--
    @author jianghc
-->
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    访问者模式
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.visitor1.*" %>
<br />
<br />
<h4>
<%
ObjectStructure obj = new ObjectStructure();  //依赖 ObjectStructure
%><br>
 <%
//具体元素的实例化
 obj.increase(new Horse());
 obj.increase(new Moke());
 %><br>
<%
//在处理身高时，各元素的不同反应
IVisitor stature = new Stature();          //依赖 Visitor 接口
%><br>
<%
out.print(obj.show(stature)+" ");
%><br>
<%
//在处理体重时，各元素的不同反应
IVisitor weight = new Weight();          //依赖 Visitor 接口
```

```
  %><br>
   <%
   out.print(obj.show(weight)+" ");
  %><br>
   <br>
 </h4>
 <br />
 <br />
 <h4>
 </BODY>
 </HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
伊犁马，一般体高 144～148 厘米.滚沙驴，一般体高 107 厘米。
伊犁马,体重 400～450 千克.滚沙驴，体重 140～190 千克。
```

30.5 模式扩展

当马与驴子产生感情后会生出一种新的动物，我们叫它为骡子，在已有访问者架构下该如何处理呢？此时，我们需要增加一个骡子类 mule 来实现马科接口。

比较不好的办法是我们需要修改访问者接口与所有具体访问者类。因为，Visitor 方法中没有包含访问骡子对象的行为，这将促使我们去手工在 Visito 中添加有骡子对象，这将违反“开放-封闭”原则。

比较好的办法是去除 Visitor 对马科的依赖关系，如图 30.4 所示。

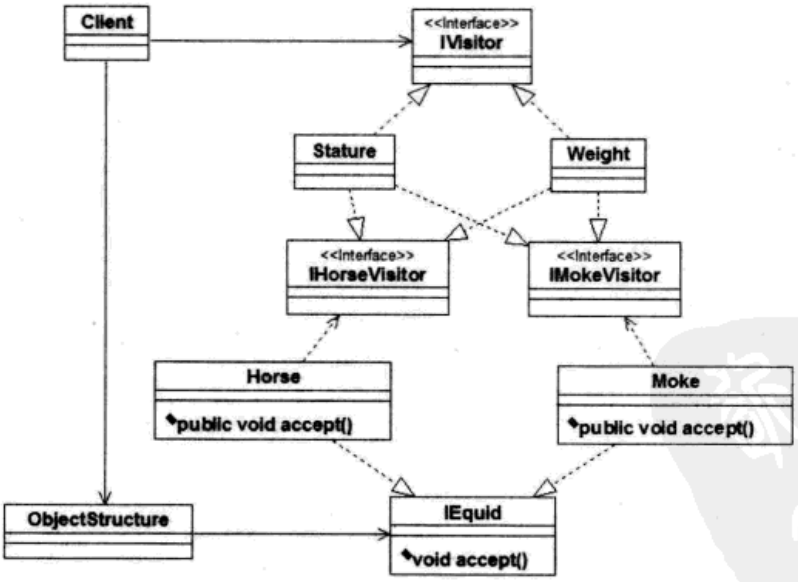

图 30.4

本应用情景的工程名为“程序 30.5.1”，源代码如下所示。

（1）访问者

```
package model.visitor2;

//访问者
public interface IVisitor {
   //无抽象方法
}
```

（2）具体访问者角色—身高

```
package model.visitor2;

//具体访问者角色-身高
public  class Stature implements IVisitor,IMokeVisitor,IHorseVisitor
//伊犁马、滚沙驴身高的表现
    public String visit(Horse horse) {
        String a="伊犁马，一般体高 144～148 厘米。";
        System.out.println("伊犁马，一般体高 144～148 厘米。");
        return a;
    }
    public String visit(Moke moke) {
        String b="滚沙驴，一般体高 107 厘米。";
        System.out.println("滚沙驴，一般体高 107 厘米。");
        return b;
    }
}
```

（3）具体访问者角色-体重

```
package model.visitor2;

//具体访问者角色-体重
public class Weight implements IVisitor,IMokeVisitor,IHorseVisitor {
    //伊犁马、滚沙驴体重的表现
    public String visit(Horse horse) {
        String a="伊犁马，体重 400～450 千克。";
        System.out.println("伊犁马，体重 400～450 千克。");
        return a;
    }
    public String visit(Moke moke) {
       String b="滚沙驴，体重 140～190 千克。";
        System.out.println("滚沙驴，体重 140～190 千克。");
        return b;
    }
```

```
}
```

（4）新增马接口

```
package model.visitor2;

//新增马接口
public interface IHorseVisitor {
    public String visit(Horse horse);
}
```

（5）新增驴接口

```
package model.visitor2;

//新增驴接口
public interface IMokeVisitor {
    public String visit(Moke moke);
}
```

（6）具体元素角色-马类

```
package model.visitor2;

//具体元素角色-马
public class Horse implements IEquid {
    public void accept(IVisitor visitor) {
        if (visitor instanceof IHorseVisitor) {
            IHorseVisitor Ihv = (IHorseVisitor) visitor;
            Ihv.visit(this);
        }
    }
}
```

（7）具体元素角色-驴类

```
package model.visitor2;

//具体元素角色-驴
public class Moke implements IEquid {
    public void accept(IVisitor visitor) {
        if (visitor instanceof IMokeVisitor) {
            IMokeVisitor Imv = (IMokeVisitor) visitor;
```

```
            Imv.visit(this);
        }
    }
}
```

（8）元素角色-马科

```
package model.visitor2;

//元素角色-马科
public interface IEquid {
        void accept(IVisitor visitor);
    }
```

（9）对象结构

```
package model.visitor2;

import java.util.*;
////对象结构角色
public class ObjectStructure {
    private List<IEquid> elements = new ArrayList<IEquid>();
    public void increase(IEquid element) { //增加increase
        elements.add(element);
    }
    public void discerption(IEquid element) { //discerption分开,移除
        elements.remove(elements); //移除
    }
    //遍历各种类型的具体元素，并运行其accept方法
    public String show(IVisitor visitor) {
        String a1="";
        String a2="";
        String a3="";
        String a4="";
          if(visitor instanceof Stature ){
              Moke moke = null;
              IMokeVisitor Imv = (Stature) visitor;
            a1=Imv.visit(moke);
          }
          if(visitor instanceof Stature ){
              Horse horse = null;
              IHorseVisitor Ihv = (Stature) visitor;
            a2=Ihv.visit(horse);
          }
```

```
            if(visitor instanceof Weight ){
                Moke moke = null;
                IMokeVisitor Imv = (Weight) visitor;
              a3=Imv.visit(moke);
            }
            if(visitor instanceof Weight ){
                Horse horse = null;
                IHorseVisitor Ihv = (Weight) visitor;
              a4=Ihv.visit(horse);
            }
           for (IEquid equid : elements) {
              equid.accept(visitor);
           }
           return a1+a2+a3+a4;
       }
   }
```

（10）客户端（test2.jsp）

```
<HTML>
<!--
    @author jianghc
-->
<HEAD>
 <meta charset=GB2312" />
    <TITLE>
    访问者模式扩展上半部分
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
<%@ page pageEncoding="GB2312" language="java" import="model.visitor2.*" %>
<br />
<br />
<h4>
<%
ObjectStructure obj = new ObjectStructure();  //依赖ObjectStructure
%><br>
 <%
//具体元素的实例化
 obj.increase(new Horse());
 obj.increase(new Moke());
 %><br>
<%
//在处理身高时，各元素的不同反应
IVisitor stature = new Stature();          //依赖Visitor接口
```

```
obj.show(stature);
%><br>
<%
out.print(obj.show(stature)+" ");
//身高

%><br>
<%
//在处理体重时，各元素的不同反应
IVisitor weight = new Weight();          //依赖 Visitor 接口
//obj.show(weight);
 %><br>
  <%
  out.print(obj.show(weight)+" ");
 %><br>
  <br>
</h4>
<br />
<br />
<h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
滚沙驴，一般体高 107 厘米。伊犁马，一般体高 144～148 厘米。
滚沙驴，体重 140～190 千克。伊犁马，体重 400～450 千克。
```

接下来，我们增加新的行为，即增加一个具体访问者即骡子的角色！

本应用情景的工程名为“程序 30.5.2”，源代码如下所示。

（1）具体元素角色-骡子类（新增此类）

```
package model.visitor3;

//具体元素角色-骡子
public class Mule implements IEquid {
    public void accept(IVisitor visitor) {
        if (visitor instanceof IMuleVisitor) {
            IMuleVisitor Imv = (IMuleVisitor) visitor;
            Imv.visit(this);
        }
        //视情况判断，else 不一定要写
        else {
            String s = visitor.getClass().getName();
```

```
                String state = s.substring(s.lastIndexOf(".") + 1, s.length());
                System.out.println("骡子在" + state + "时没有设置体重!!");
            }
        }
    }
```

（2）骡子接口（新增此接口类）

```
package model.visitor3;

//新增骡子接口
public interface IMuleVisitor {
    public String visit(Mule mule);
}
```

（3）具体访问者角色—身高类（修改）

```
package model.visitor3;

//具体访问者角色-身高
public  class Stature implements
IVisitor,IMokeVisitor,IHorseVisitor,IMuleVisitor {
//伊犁马、滚沙驴身高的表现
    public String visit(Horse horse) {
        String a="伊犁马，一般体高 144～148 厘米。";
        System.out.println("伊犁马，一般体高 144～148 厘米。");
        return a;
    }
    public String visit(Moke moke) {
        String b="滚沙驴，一般体高 107 厘米。";
        System.out.println("滚沙驴，一般体高 107 厘米。");
        return b;
    }
    public String visit(Mule mule) {
        String c="骡子，英国迷你骡子身高只有 50 厘米。";
        System.out.println("骡子，英国迷你骡子身高只有 50 厘米。");
        return c;
    }
}
```

（4）具体访问者角色—体重类

```
package model.visitor3;
```

```
    //具体访问者角色-体重
    public class Weight implements
IVisitor,IMokeVisitor,IHorseVisitor,IMuleVisitor {
        //伊犁马、滚沙驴体重的表现
        public String visit(Horse horse) {
            String a="伊犁马，体重 400～450 千克。";
            System.out.println("伊犁马，体重 400～450 千克。");
            return a;
        }
        public String visit(Moke moke) {
           String b="滚沙驴，体重 140～190 千克。";
            System.out.println("滚沙驴，体重 140～190 千克。");
            return b;
        }
        public String visit(Mule mule) {
           String c="骡子体重未知!!";
            System.out.println("骡子体重未知!!");
            return c;
        }
    }
```

（5）客户端 test3.jsp（修改）

```
    <HTML>
    <!--
        @author jianghc
    -->
    <HEAD>
     <meta charset=GB2312" />
        <TITLE>
        访问者模式扩展下半部分
        </TITLE>
    </HEAD>
    <BODY BGCOLOR="white">
    <%@ page pageEncoding="GB2312" language="java" import="model.visitor3.*" %>
    <br />
    <br />
    <h4>
    <%
    ObjectStructure obj = new ObjectStructure();  //依赖 ObjectStructure
    %><br>
     <%
    //具体元素的实例化
     obj.increase(new Horse());
     obj.increase(new Moke());
```

```
 obj.increase(new Mule());      //新增一种具体元素 mule
 %><br>
<%
//在处理身高时，各元素的不同反应
IVisitor stature = new Stature();          //依赖 Visitor 接口
%><br>
<%
out.print(obj.show(stature)+" ");
%><br>
<%
//在处理体重时，各元素的不同反应
IVisitor weight = new Weight();          //依赖 Visitor 接口
 %><br>
  <%
  out.print(obj.show(weight)+" ");
 %><br>
  <br>
</h4>
<br />
<br />
<h4>
</BODY>
</HTML>
```

在浏览器上打开页面，执行代码，显示结果如下所示。

```
滚沙驴，一般体高 107 厘米。伊犁马，一般体高 144～148 厘米。骡子，英国迷你骡子身高只有 50 厘米。
滚沙驴，体重 140～190 千克。伊犁马，体重 400～450 千克。骡子体重未知!!
```

第六部分

设计模式应用思维

第 31 章　设计模式实战

31.1　示例项目概述

良好的运用设计模式可使软件设计开发人员有效地提高工作效率和降低技术难度，从而为软件系统的开发与维护提供扎实的实现基础。

本章节依托 J2EE 技术与 Struts、Hibernate 框架，结合“单例”、“装饰”、“命令”、“中介者”、策略”等模式进行软件实战项目的阐述。

为了便于读者的理解，此处通过常规的“增、删、改、查”功能页面的实现，进行层次化的介绍，以使大家能有一个清晰的认识。

31.2　需求分析

某商业公司，由于业务的飞速发展，需要对各个分公司的各类产品类型进行维护。此时，信息部的需求分析师，将需求划分为通过 Web 方式进行访问的“增、删、改、查”功能，并决定将产品类型维护的实现主要由数据库与 Java Web 代码来完成。

- 数据库主要运用 MySQL5。
- Java Web 代码主要运用 Struts（1.2 版本）、Hibrnate(3.0 版本)、jsp、css、js。
- 项目环境配置主要运用 JDK1.6 和 Tomcat6 。
- 开发工具主要使用 Myeclipse，读者可以使用自己熟悉的其他 IDE。

31.3　功能开发实现

31.3.1　数据库表结构

本示例项目使用的表结构参看下面的 SQL 语句：

```
DROP TABLE IF EXISTS 'product';
CREATE TABLE 'product'(
  'ID' int(4) NOT NULL AUTO_INCREMENT,
  'ProductName' char(40) DEFAULT NULL,
  'ProductRemark' text,
  PRIMARY KEY ('ID')
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=gb2312;
```

大家可以安装 Navicat for MySQL 之类的图形化工具，进入命令行界面进行 SQL 语句的创建。

31.3.2 Java Web 程序结构

1. 程序的总体结构

程序的总体结构如图 31.1 所示。

图 31.1

- Struts 框架：test.struts、test.action 和 test.form。
- Hibernate 框架：test.entity 和 hibernate.cfg.xml。
- J2SE 包：test.base、test.filter 和 test.service。
- 日志配置：log4j.properties。
- WebRoot：tld、jsp、js、css、struts-config.xml、validation.xml、validator-rules.xml、web.xml 等等。
- org 配置文件：displaytag.properties。

2. 程序层次关系

程序层次关系如图 31.2 所示。

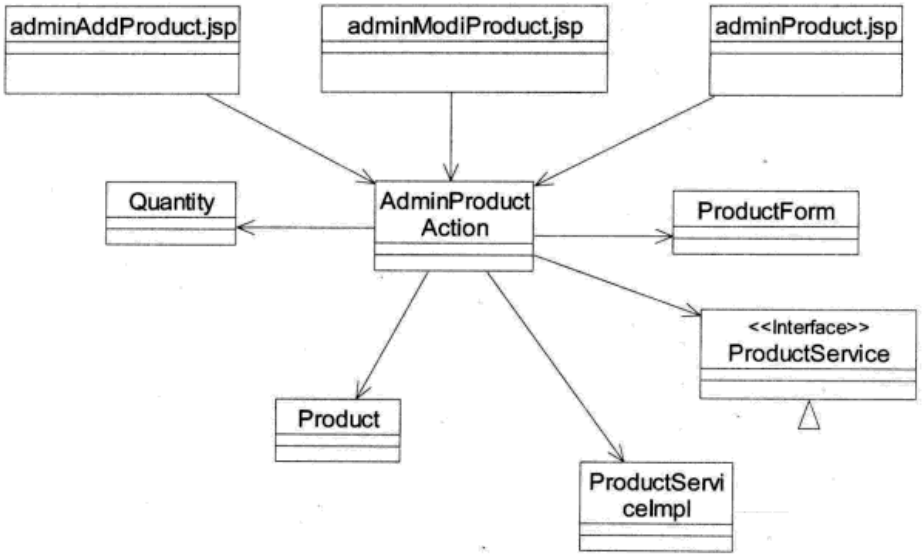

图 31.2

31.3.3 Java Web 程序详述

本节说明设计模式在 Java Web 程序中的使用，示例的工程名为“程序 31.33”，源代码的构成分别说明如下：

（1）Action 类

```
package test.action;

import java.util.*;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import test.base.*;
import test.entity.*;
import test.form.ProductForm;
import test.service.*;

// 产品类型维护
public class AdminProductAction extends BasisAction {
    public ActionForward addProductInit(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        this.resetToken(request);
        request.setAttribute("productForm", null);
        this.saveToken(request);
        return mapping.findForward("addProduct");
    }
    //新增
    public ActionForward addProduct(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
```

```
            ProductForm productForm = (ProductForm) form;
            Product product = new Product();
            product.setProductName(productForm.getProductName().trim());
            product.setProductRemark(productForm.getProductRemark().trim());
            ProductService service = new ProductServiceImpl();
            ActionMessages msgs = new ActionMessages();
            try{
                System.out.println("/////////" + isTokenValid(request));
                if (!isTokenValid(request))
                {
                    msgs.add("addProductStatus", new ActionMessage(
                            Quantity.ADDPRODUCT_ALREADY_KEY));//
                    saveErrors(request, msgs);
                    return mapping.getInputForward();
                }
                else{
                    this.resetToken(request);
                boolean status = service.addProduct(product);
                if (status){
                    msgs.add("addProductStatus",new
ActionMessage(Quantity.ADDPRODUCT_SUC_KEY));
                }else{
                    msgs.add("addProductStatus",new
ActionMessage(Quantity.ADDPRODUCT_FAIL_KEY));
                }
                saveErrors(request, msgs);
                }
            }catch(Exception ex){
                logger.info("调用AdminProductAction类之addProduct方法报错: \n");
                ex.printStackTrace();
            }
            return mapping.findForward("addAdminProduct");
        }
        //查看
        public ActionForward checkProduct(ActionMapping mapping, ActionForm form,
                HttpServletRequest request, HttpServletResponse response) {
            List chklist = null;
            ProductService service = new ProductServiceImpl();
            try{
                chklist = service.checkProduct();
                request.setAttribute("productList", chklist);
            }catch(Exception ex){
                logger.info("调用AdminProductAction类之checkProduct方法报错:\n");
                ex.printStackTrace();
            }
            return mapping.findForward("check");
        }
        //删除
        public ActionForward delProduct(ActionMapping mapping, ActionForm form,
```

```
                HttpServletRequest request, HttpServletResponse response) {
            ActionMessages msgs = new ActionMessages();
            ProductService service = new ProductServiceImpl();
            String sId = request.getParameter("id");
            Integer id = null;
            if(sId!=null){
                id = new Integer(sId);
            }else{
                id = new Integer(0);
            }
            try{
                boolean status = service.delProduct(id);
                if (status){
                    msgs.add("delProductStatus",new
ActionMessage(Quantity.DELPRODUCT_SUC_KEY));
                }else{
                    msgs.add("delProductStatus",new
ActionMessage(Quantity.DELPRODUCT_FAIL_KEY));
                }
                saveErrors(request, msgs);
            }catch(Exception ex){
                logger.info("调用 AdminProductAction 之 delProduct 方法报错: \n");
                ex.printStackTrace();
            }
            return mapping.findForward("delete");
        }
        //载入
        public ActionForward loadProduct(ActionMapping mapping, ActionForm form,
                HttpServletRequest request, HttpServletResponse response) {
            ActionMessages msgs = new ActionMessages();
            ProductService service = new ProductServiceImpl();
            Product product = null;
            String sId = request.getParameter("id");
            Integer id = null;
            if(sId!=null){
                id = new Integer(sId);
            }else{
                id = new Integer(0);
            }
            try{
                product = service.loadProduct(id);
                if(product!=null){
                    form = new ProductForm();

    ((ProductForm)form).setProductName(product.getProductName().trim());

    ((ProductForm)form).setProductRemark(product.getProductRemark().trim());
                    request.setAttribute("productForm", form);
                    request.setAttribute("id", product.getId());
```

```
            }
        }catch(Exception ex){
            logger.info("调用 AdminProductAction 之 loadProduct 方法报错: \n");
            ex.printStackTrace();
        }
        return mapping.findForward("modify");
    }
    //修改
    public ActionForward modiProduct(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        ProductForm productForm = (ProductForm) form;
        Product product = null;
        ProductService service = new ProductServiceImpl();
        ActionMessages msgs = new ActionMessages();
        String sId = request.getParameter("id");
        Integer id = null;
        if(sId!=null){
            id = new Integer(sId);
        }else{
            id = new Integer(0);
        }
        try{
            product = service.loadProduct(id);
            if (product!=null){

    product.setProductName(productForm.getProductName().trim());

    product.setProductRemark(productForm.getProductRemark().trim());
                boolean status = service.updateProduct(product);
                if (status){
                    msgs.add("modiProductStatus",new
ActionMessage(Quantity.MODIPRODUCT_SUC_KEY));
                }else{
                    msgs.add("modiProductStatus",new
ActionMessage(Quantity.MODIPRODUCT_FAIL_KEY));
                }
            }else{
                msgs.add("modiProductStatus",new
ActionMessage(Quantity.MODIPRODUCT_FAIL_KEY));
            }
            saveErrors(request, msgs);
        }catch(Exception ex){
            logger.info("调用 AdminProductAction 类之 modiProduct 方法报错: \n");
            ex.printStackTrace();
        }
        return mapping.findForward("modify");
    }
}
```

本代码基于 struts1.2 进行开发，应用了单例模式的相关原理，具体原因可参见第 10.4 节的“单例模式与 Struts”部分。

上面代码中：

```
public ActionForward addProductInit
public ActionForward addProduct
public ActionForward checkProduct
public ActionForward delProduct
public ActionForward loadProduct
public ActionForward modiProduct
```

等方法属于 Action 的 execute()方法，因而它们均运用了命令模式。具体原因可参见第 21.5 节“命令模式与 Struts”部分。

（2）Hibernate 实体类

```
package test.entity;

//产品类别实体类

public class Product implements java.io.Serializable {
    private Integer id;
    private String productName;
    private String productRemark;
    public Product() {
    }
    public Product(String productName, String productRemark) {
        this.productName = productName;
        this.productRemark = productRemark;
    }
    public Integer getId() {
        return this.id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getProductName() {
        return this.productName;
    }
    public void setProductName(String productName) {
        this.productName = productName;
    }
    public String getProductRemark() {
        return this.productRemark;
    }
    public void setProductRemark(String productRemark) {
        this.productRemark = productRemark;
    }
}
```

（3）Hibernate 映射文件（Product.hbm.xml）

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="test.entity.Product" table="product" catalog="test">
        <id name="id" type="java.lang.Integer">
            <column name="ID" />
            <generator class="identity" />
        </id>
        <property name="productName" type="java.lang.String">
            <column name="ProductName" length="40" />
        </property>
        <property name="productRemark" type="java.lang.String">
            <column name="ProductRemark" length="65535" />
        </property>
    </class>
</hibernate-mapping>
```

（4）Struts 的表单类

```
package test.form;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.validator.ValidatorForm;

//表单类
public class ProductForm extends ValidatorForm {
    private String productRemark;
    private String productName;
    /**
     * Method validate
     * @param mapping
     * @param request
     * @return ActionErrors
     */
    @Override
    public ActionErrors validate(ActionMapping mapping,
            HttpServletRequest request) {
        return null;
    }
    @Override
    public void reset(ActionMapping mapping, HttpServletRequest request) {
    }
    public String getProductRemark() {
```

```
        return productRemark;
    }
    public void setProductRemark(String productRemark) {
        this.productRemark = productRemark;
    }
    public String getProductName() {
        return productName;
    }
    public void setProductName(String productName) {
        this.productName = productName;
    }
}
```

（5）业务接口类

```
package test.service;

import java.util.*;
import test.entity.*;

//产品类别业务接口
public interface ProductService {
    //浏览产品类别
    public List checkProduct() throws Exception;
    // 载入所需产品类别
    public Product loadProduct(Integer id) throws Exception;
    //删除所需产品类别
    public boolean delProduct(Integer id) throws Exception;
    //新增所需产品类别
    public boolean addProduct(Product product) throws Exception;
    // 更新所需产品类别
    public boolean updateProduct(Product product) throws Exception;
    // 统计记录数量
    public int countRecord(String hql) throws Exception;
}
```

（6）业务接口实现类

```
package test.service;

import java.util.List;
import org.hibernate.*;
import test.base.BasisLog;
import test.entity.*;

// 产品类别业务接口实现
public class ProductServiceImpl extends BasisLog implements ProductService {
    //新增产品类别
    public boolean addProduct(Product product) throws Exception {
```

```
        Session session = TestSessionFactory.getSession();
        Transaction ts = null;
        boolean state = false;
        try{
            ts = session.beginTransaction();
            session.save(product);
            ts.commit();
            state = true;
        }catch(Exception ex){
            if(ts!=null)ts.rollback();
            logger.info("调用 ProductServiceImpl 类之 addProduct 方法报错: \n");
            ex.printStackTrace();
        }finally{
            TestSessionFactory.closeSession();
        }
        return state;
    }
    // 查看产品类别
    public List checkProduct() throws Exception {
        Session session = TestSessionFactory.getSession();
        Transaction ts = null;
        List chklist = null;
        try{
            Query query = session.createQuery("from Product as a order by
a.id");
            ts = session.beginTransaction();
            chklist = query.list();
            ts.commit();
            if
(!Hibernate.isInitialized(chklist))Hibernate.initialize(chklist);
        }catch(Exception ex){
            if(ts!=null)ts.rollback();
            logger.info("调用 ProductServiceImpl 类之 checkProduct 方法报错:\n");
            ex.printStackTrace();
        }finally{
            TestSessionFactory.closeSession();
        }
        return chklist;
    }
    // 删除所需产品类别
    public boolean delProduct(Integer id) throws Exception {
        Session session = TestSessionFactory.getSession();
        Transaction ts = null;
        boolean state = false;
        try{
            ts = session.beginTransaction();
            Product product = (Product)session.load(Product.class, id);
            session.delete(product);
            ts.commit();
```

```
            state = true;
        }catch(Exception ex){
            if(ts!=null)ts.rollback();
            logger.info("调用 ProductServiceImpl 类之 delProduct 方法报错: \n");
            ex.printStackTrace();
        }finally{
            TestSessionFactory.closeSession();
        }
        return state;
    }
    // 载入所需产品类别
    public Product loadProduct(Integer id) throws Exception {
        Session session = TestSessionFactory.getSession();
        Transaction ts = null;
        Product product = null;
        try{
            ts = session.beginTransaction();
            product = (Product)session.get(Product.class, id);
            ts.commit();
        }catch(Exception ex){
            if(ts!=null)ts.rollback();
            logger.info("调用 ProductServiceImpl 类之 loadProduct 方法报错: \n");
            ex.printStackTrace();
        }finally{
            TestSessionFactory.closeSession();
        }
        return product;
    }
    // 更新产品类别
    public boolean updateProduct(Product product) throws Exception {
        Session session = TestSessionFactory.getSession();
        Transaction ts = null;
        boolean state = false;
        try{
            ts = session.beginTransaction();
            session.update(product);
            ts.commit();
            state = true;
        }catch(Exception ex){
            if(ts!=null)ts.rollback();
            logger.info("调用 ProductServiceImpl 类之 updateProduct 方法报错:
\n");
            ex.printStackTrace();
        }finally{
            TestSessionFactory.closeSession();
        }
        return state;
    }
    //统计记录数量
```

```
    public int countRecord(String hql) throws Exception {
        Session session = TestSessionFactory.getSession();
        Transaction ts = null;
        int count = 0;
        try{
            ts = session.beginTransaction();
            Query query = session.createQuery(hql);
            query.setMaxResults(1);
            count = ((Integer)query.uniqueResult()).intValue();
            ts.commit();
        }catch(Exception ex){
            if(ts!=null)ts.rollback();
            logger.info("调用 ProductServiceImpl 类之 countRecord 方法报错: \n");
            ex.printStackTrace();
        }finally{
            TestSessionFactory.closeSession();
        }
        return count;
    }
}
```

（7）查询列表页面（adminProduct.jsp）

```
<%@ page language="java" pageEncoding="gb2312"%>
<%@ taglib uri="/struts-bean" prefix="bean"%>
<%@ taglib uri="/struts-html" prefix="html"%>
<%@ taglib uri="http://displaytag.sf.net" prefix="display" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
    <head>
        <title>产品类别管理</title>
        <link rel="stylesheet" type="text/css" href="../CSS/stylesheet.css">
        <link rel="stylesheet" type="text/css" href="../CSS/displaytag.css"
/>
    <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
    </head>
  <body>
    <c:set var="label1"><bean:message key="product.table.label1"/></c:set>
    <c:set var="label2"><bean:message key="product.table.label2"/></c:set>
    <c:set var="label3"><bean:message key="product.table.label3"/></c:set>

    <table  border="0"  align="center"  cellpadding="0"  cellspacing="0"
style="background-color:lightblue; border:0px;">
      <tr>
        <td height="38" class="productTitle" align="center"><bean:message
key="product.table.title"/></td>
      </tr>
      <tr>
        <td height="28" class="blueText" align="center">
            <html:link forward="addAdminProduct">
```

```
                        <span                    class="blueText"><bean:message
key="product.table.add"/></span>
                </html:link>
            </td>
        </tr>
        <tr>
          <td height="30" align="center">
                <display:table   name="productList"   id="row"   pagesize="10"
export="false"                                          class="displaytag"
requestURI="/Admin/adminProduct.do?method=checkProduct" >
                    <display:column  property="productName"  title="${label1}"
sortable="true" headerClass="sortable" style="text-align:center;"/>
                    <display:column  property="productRemark"  title="${label2}"
sortable="true" headerClass="sortable" style="text-align:center;"/>
                    <display:column        title="${label3}"        media="html"
style="text-align:center;">
                        <html:link
page="/Admin/adminProduct.do?method=loadProduct"
                                paramId="id"
                                paramName="row"
                                paramProperty="id">
                            <bean:message key="product.table.modify"/>
                        </html:link>
                        <html:link
page="/Admin/adminProduct.do?method=delProduct"
                                paramId="id"
                                paramName="row"
                                paramProperty="id">
                            <bean:message key="product.table.delete"/>
                        </html:link>
                    </display:column>
                </display:table>
            </td>
        </tr>
        <tr>
          <td   height="28"   align="center"   class="redText"><html:errors
property="delProductStatus"/></td>
        </tr>
      </table>
      </body>
    </html>
```

在浏览器上打开页面，执行代码，显示结果如图 31.3 所示。单击“新增产品类别”链接，出现图 31.4 所示“新增产品类别”页面。单击“修改产品类别”链接，出现图 31.5 所示“修改产品类别”页面。

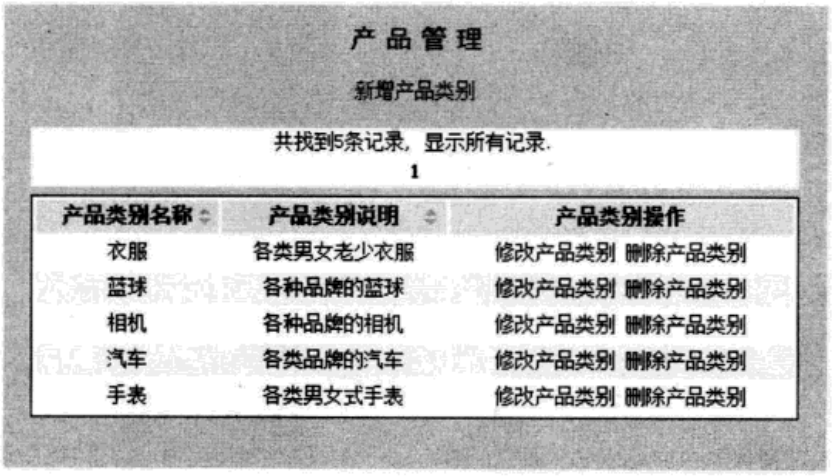

图 31.3

（8）新增页面（adminAddProduct.jsp）

```
    <%@ page contentType="text/html; charset=gb2312" %>
    <%@ taglib uri="/struts-bean" prefix="bean" %>
    <%@ taglib uri="/struts-html" prefix="html" %>
    <jsp:useBean id="JSONRPCBridge" scope="session" class="com.metaparadigm.
jsonrpc.JSONRPCBridge"/>
    <jsp:useBean id="ajax" class="test.base.AjaxObj"></jsp:useBean>
    <%
        JSONRPCBridge.registerObject("ajax",ajax);
    %>
    <html>
    <head>
    <title>产品类别管理</title>
    <link href="../CSS/stylesheet.css" rel="stylesheet" type="text/css">
    <script type="text/javascript" src="../JS/jsonrpc.js"></script>
    </head>
    <body style="background-color:lightblue; border:0px;">
    <html:javascript formName="productForm"/>
    <html:form action="/Admin/adminProduct.do?method=addProduct" onsubmit="return
 validateCateForm(this);">
      <table width="500" border="0" align="center" cellpadding="0" cellspacing=
"0">
       <tr height="39">
          <td colspan="2" class="productTitle" align="center">
           <bean:message key="product.table.add"/>
          </td>
        </tr>
        <tr height="29">
          <td width="150" align="right"><bean:message key="product.table.
label1"/>: </td>
          <td><html:text property="productName" size="41" styleClass="textBox"
onblur="checkProductName()"/></td>
        </tr>
```

```
        <tr height="29">
          <td valign="top"  align="right"><bean:message key="product.table.
label2"/>: </td>
          <td><html:textarea  property="productRemark"  cols="40"  rows="10"
styleClass="textBox"/></td>
        </tr>
        <tr height="29">
          <td colspan="2" align="center">
               <html:reset><bean:message key="reset.text"/></html:reset>
               <html:submit><bean:message key="submit.text"/></html:submit>
           </td>
        </tr>
        <tr>
          <td height="29" align="center" colspan="2" class="redText">
               <html:errors property="addProductStatus"/>
           </td>
        </tr>
      </table>
    </html:form>
    <script language="javascript">
        jsonrpc = new JSONRpcClient("../JSON-RPC");
        //检查产品类别名称是否有效
        function checkProductName(){
            var lname = document.all.productName.value;
            if (!jsonrpc.ajax.chkProductName(lname)){
                alert('抱歉，本产品分类已被占用，请重新输入！');
                document.all.productName.focus();
            }
        }
    </script>
    </body>
    </html>
```

在浏览器上打开页面，执行代码，显示结果如图 31.4 所示。

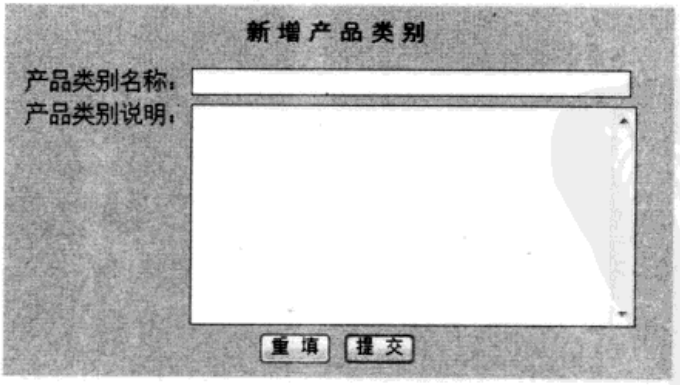

图 31.4

（9）修改页面（adminModiProduct.jsp）

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib uri="/struts-bean" prefix="bean" %>
<%@ taglib uri="/struts-html" prefix="html" %>
<html>
<head>
<title>产品类别管理</title>
<link href="../CSS/stylesheet.css" rel="stylesheet" type="text/css">
</head>
<body style="background-color:lightblue; border:0px;">
<html:javascript formName="productForm"/>
<html:form action="/Admin/adminProduct.do?method=modiProduct" onsubmit=
"return validateCateForm(this);">
  <table width="600" border="0" align="center" cellpadding="0" cellspacing=
"0">
    <tr height="38">
      <td colspan="2" class="productTitle" align="center">
       <bean:message key="product.table.modify"/>
      </td>
    </tr>
    <tr height="28">
      <td width="160" align="right"><bean:message key="product.table.
label1"/>: </td>
      <td><html:text property="productName" size="41" styleClass=
"textBox"/></td>
    </tr>
    <tr height="28">
      <td valign="top"  align="right"><bean:message key="product.table.
label2"/>: </td>
      <td><html:textarea property="productRemark" cols="40" rows="10" styleClass=
"textBox"/></td>
    </tr>
    <tr height="28">
      <td colspan="2" align="center">
          <html:reset><bean:message key="reset.text"/></html:reset>
          <html:submit><bean:message key="submit.text"/></html:submit>
       </td>
    </tr>
    <tr>
      <td height="28" align="center" colspan="2" class="redText">
          <html:errors property="modiProductStatus"/>
       </td>
    </tr>
  </table>
  <input type="hidden" name="id" value="<%=request.getAttribute("id")%>">
</html:form>
</body>
</html>
```

在浏览器上打开页面，执行代码，显示结果如图 31.5 所示。

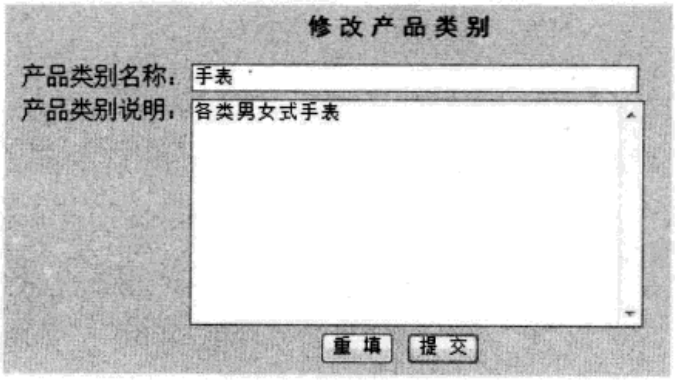

图 31.5

（10）struts-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts
Configuration 1.2//EN" "http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
  <data-sources />
  <form-beans >
    <form-bean name="productForm" type="test.form.ProductForm" />
  </form-beans>

  <global-exceptions />
  <global-forwards >
    <forward name="addAdminProduct" path="/Admin/adminProduct.do?method=
addProductInit" />
  </global-forwards>

  <action-mappings >

    <action
      attribute="productForm"
      input="/Admin/adminAddProduct.jsp"
      name="productForm"
      parameter="method"
      path="/Admin/adminProduct"
      scope="request"
      type="test.action.AdminProductAction">
      <forward name="check" path="/Admin/adminProduct.jsp" />
      <forward name="modify" path="/Admin/adminModiProduct.jsp" />
      <forward name="delete" path="/Admin/adminProduct.do?method=
checkProduct" />
      <forward name="addProduct" path="/Admin/adminAddProduct.jsp" />
    </action>
```

```
    </action-mappings>
    <message-resources parameter="test.struts.ApplicationResources" />

    <!--配置 struts-menu 插件-->
    <plug-in className="net.sf.navigator.menu.MenuPlugIn">
      <set-property property="menuConfig" value="/WEB-INF/menu-config.xml"/>
    </plug-in>

    <!--配置 Validator 插件-->
    <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
      <set-property property="pathnames" value="/WEB-INF/validator-rules.xml,/
WEB-INF/validation.xml" />
    </plug-in>

  </struts-config>
```

本处主要运用了装饰模式，具体原因可参见第 16.4 节“装饰模式与 Struts”部分。

（11）其他文件

其他文件还包括 web.xml、hibernate.cfg.xml、validator-rules.xml、validation.xm、Application Resources.properties 等配置文件，js 脚本文件和 ss 样式文件。这些文件的使用说明如下：

- web.xml、hibernate.cfg.xml，validator-rules.xml 和 validation.xm 按常规进行配置。
- css、js 与 jsp 页面相结合。
- ApplicationResources.properties 需要与 jsp 页面的 struts 标签对应。
- validation.xm 运用了策略模式，具体原因可参见 28 章“28.7 策略模式与 Struts”部分。

这里，我们给出 validation.xm 的配置，如下所示。

```
<form-validation>
    <global>
    <!-- 配置常量表达式 -->
      <constant>
          <constant-name>phone</constant-name>
          <constant-value>^\(?(\d{3})\)?[-| ]?(\d{8})$</constant-value>
      </constant>
    </global>
    <formset>
        <form name = "productForm">
            <field
                property = "productName"
                depends = "required">
                <arg0 key = "product.table.label1"/>
            </field>
            <field
                property = "productRemark"
                depends = "required">
                <arg0 key = "product.table.label2"/>
```

```
            </field>
        </form>
    </formset>
</form-validation>
```

31.4 本章小结

本章从常规功能入手，通过一个实例详细介绍了设计模式在 Java Web 项目中的应用，希望对广大软件设计开发人员的成长和软件公司的发展产生积极的借鉴作用。