Broadview
www.broadview.com.cn

模式工程化实现及扩展

（设计模式Java版）

王翔 孙逊 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

设计模式不是一门适合空谈的技术，它来自于开发人员的工程实践又服务于工程实践。

本书并不是一本面向入门者的读物，因为它需要结合工程实践介绍如何发现模式灵感、如何应用模式技术。不过作为一本介绍设计模式的书，它并不需要读者对于庞大的 JDK 有深入了解，因为扩展主要是结合 Java 语法完成的，配合书中的实例，相信读者不仅能够熟练应用设计模式技术，也能令自己的 Java 语言上一个台阶。

为了降低学习门槛，本书第一部分除了介绍面向对象设计原则外，还充实了一些 Java 语言的介绍，但这些内容并不是枯燥的讲解，读者可以在阅读中通过一系列动手练习尽快吸收这些理论并将它们转化为自己的技能。本书最后一部分的“GOF 综合练习”把各种设计模式做了一次集中展示，目的是让读者把分散的模式知识融合在一起，能够将书本知识真正用于改善一个“准”生产型模块的实现。

本书内容生动，示例贴近中型、大型项目实践，通过一个个“四两拨千斤”的示例练习可以让读者有一气读完的兴趣。

图书在版编目（CIP）数据

模式：工程化实现及扩展：设计模式 Java 版 / 王翔，孙逊著.—北京：电子工业出版社，2012.4
ISBN 978-7-121-15638-0

Ⅰ. ①模… Ⅱ. ①王… ②孙… Ⅲ. ①Java 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2011）第 280674 号

策划编辑：张春雨
责任编辑：李云静
特约编辑：赵树刚
印　　刷：
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：26　字数：666 千字
印　　次：2012 年 4 月第 1 次印刷
印　　数：4000 册　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

如同每个人都有其个性一样，每种开发语言也有自己的特点。

项目中，我们固然可以机械地将一种语言的开发经验套用到另一种语言中，但效果不一定好，因为：

- 语言有自己的短处：用短处去实现需求不仅费时费力，结果也不理想。
- 语言有自己的长处：为了沿用以前的经验而“削足适履”，没有用到语言的精要，结果暴殄天物。

相信读者也发现了，用一种语言写一个应用是一回事，写好一个应用则完全是另一回事，这就是工程化代码和“玩具”代码的区别。教科书上的知识落实到工程上时不能按图索骥，需要考虑开发语言和目标环境，设计模式也不例外。

也许读者会觉得本书中很多实现方式与《设计模式》介绍的内容不一致，这是因为《设计模式》一书出版至今已近 20 年，其间无论是开发语言还是技术平台均已经“换了人间”。GOF 23 个模式的思想不仅影响着我们，更影响着走在技术前沿的语言设计者、平台设计者，他们也在工作中潜移默化地把模式思想融入自己的工作成果，作为用户，我们“推却”别人的盛情，所有事情都从“车轮”做起，多少有点不经济。

作为本系列的 Java 设计模式分册，我试图用最 Java 的方式将自己对于设计模式的理解呈献给读者，而且实现上务求简洁、直接。结构上，本书分为 5 个部分：

（1）第一部分：预备知识

包括面向对象设计原则中“面向类”的部分、Java 语言面向对象扩展特性，以及 Java 和 C#语法特性的简单对比。

（2）第二部分：创建型模式

主要介绍如何创建对象，如何将客户程序与创建过程的“变化”有效隔离。

（3）第三部分：结构型模式

从静态结构出发，分析导致类型结构相互依赖的原因，通过将静态变化因素抽

象、封装为独立对象的办法，梳理对象结构关系。

（4）第四部分：行为型模式

从动态机制出发，分析导致类型调用过程的依赖因素，通过将调用关系、调用过程抽象、封装为独立对象的方法，削弱调用过程中的耦合关系。

（5）第五部分：GOF 综合练习

为了便于读者从整体上体会模式化设计思路和实现技巧，这部分通过一个综合性的示例向读者展示如何发现变化、抽象变化、应用模式并最终结合 Java SE 平台特性落地实现的过程。

不管之前对于模式是否有所尝试，我希望读者不妨浏览一下这些内容，毕竟模式思想转化为模式设计思路，再转化为模式应用技巧是一个渐进的过程，必须实际动手才会加深印象，然后才可能进一步开阔思路。本章示例设计上变化因素较多，需要三类模式的综合运用，务求能起到抛砖引玉的效果。

感谢多年培养、帮助我的领导和同事们，多年富有挑战、共同拼搏的项目经历使我能够完成这本书。

感谢我和我妻子共同的父母，他们一直给予我无私的关心和照顾，还教会我学会从生活中发掘无穷的技术灵感。

最后，感谢我挚爱的妻子，是她给予我直面挑战、战胜挑战的信心和力量。

受到自己开发年限和项目经验的限制，本书难免有疏漏和不足之处，希望能够听到您的批评和建议。

王翔

目　录

第一篇　预备知识——发掘 Java 语言的面向对象设计潜力 1

第 1 章　面向对象设计原则 2
1.1　说明 3
1.2　单一职责原则（SRP） 4
1.3　里氏替换原则（LSP）和依赖倒置原则（DIP） 6
1.4　接口隔离原则（ISP） 7
1.5　迪米特法则（Law of Demeter，LoD） 9
1.6　开闭原则（OCP） 10
1.7　小结 13
1.8　自我检验 14
第 2 章　重新研读 Java 语言 15
2.1　说明 16
2.2　Java 部分语法内容扩展 16
2.2.1　规划和组织代码——包 16
2.2.2　正式命名的常量契约——枚举 19
2.2.3　考验算法的抽象能力——泛型 20
2.2.4　用贴标签的方式扩展对象特性——标注 26
2.3　面向插件架构的配置系统设计 30
2.4　依赖注入 33
2.4.1　背景介绍 33
2.4.2　示例情景 34
2.4.3　构造注入（Constructor） 36
2.4.4　设值注入（Setter） 36
2.4.5　接口注入 37
2.4.6　小结 38
2.4.7　自我检验 39
2.5　连贯接口（Fluent Interface） 40

2.6 自我检验 41
第 3 章 Java 和 C# 43
3.1 说明 44
3.2 枚举 44
3.3 泛型 48
3.4 属性和标注 50
3.5 操作符重载和类型转换重载 52
3.6 委托、事件、匿名方法 52
3.7 Lamada 和 LINQ 56
3.8 小结 61

第二篇 创建型模式——管理对象实例的构造过程 62

第 4 章 工厂及工厂方法模式 63
4.1 说明 64
4.2 简单工厂 64
4.2.1 最简单的工厂类 64
4.2.2 简单工厂的局限性 67
4.3 经典回顾 68
4.4 解耦工厂类型与客户程序 69
4.5 基于配置文件的工厂 73
4.5.1 基于配置文件解耦工厂接口和具体工厂类型 73
4.5.2 基于配置文件解耦工厂类型和具体工作产品 74
4.6 批量工厂 77
4.7 典型工程化实现 78
4.8 小结 80
4.9 Java 中的典型实现 81
4.10 自我检验 81
第 5 章 单件模式 82
5.1 说明 83
5.2 经典回顾 84
5.3 枚举方式的单件模式 88
5.4 细节决定成败 89
5.5 线程级单件模式 92
5.6 分布式环境下的单件模式 94
5.7 单件模式的使用问题 96
5.8 小结 97

5.9 Java 中的典型实现 97
第 6 章 抽象工厂模式 98
6.1 说明 99
6.2 经典回顾 100
6.3 解决经典模式的硬伤 102
6.4 委托生产外包 105
6.5 小结 109
6.6 Java 中的典型实现 110
第 7 章 创建者模式 111
7.1 说明 112
7.2 经典回顾 113
7.3 为 Builder 贴个标签 116
7.4 具有装配/卸裁能力的 Builder 117
7.5 连贯接口形式的 Builder 118
7.6 小结 122
7.7 Java 中的典型实现 122
7.8 自我检验 122
第 8 章 原型模式 123
8.1 说明 124
8.2 经典回顾 124
8.3 表面模仿还是深入模仿 129
8.3.1 概念 129
8.3.2 “纯手工”实现深层复制 130
8.3.3 制作实现序列化工具类型 133
8.3.4 简单自定义复制过程 135
8.3.5 细颗粒度自定义复制过程 135
8.4 小结 138
8.5 自我检验 138

第三篇 结构型模式——组织灵活的对象体系 140

第 9 章 适配器模式 141
9.1 说明 142
9.2 经典回顾 143
9.3 组适配器 146
9.4 用配置约定适配过程 149
9.5 面向数据的适配机制 151

9.6 小结 154
9.7 Java 中的典型实现 154
9.8 自我检验 154
第 10 章 桥模式 157
10.1 说明 158
10.2 经典回顾 159
10.3 分解复杂性的多级桥关系 163
10.4 具有分支的桥 164
10.5 看着“图纸”造桥 168
10.6 具有约束关系的桥 171
10.7 小结 173
10.8 自我检验 173
第 11 章 组合模式 175
11.1 说明 176
11.2 经典回顾 177
11.3 适于 XML 信息的组合模式 183
11.4 分布式“部分—整体”环境 185
11.5 小结 185
11.6 Java 中的典型实现 186
11.7 自我检验 186
第 12 章 装饰模式 187
12.1 说明 188
12.2 经典回顾 188
12.3 卸载装饰 194
12.4 通过配置和创建者完成装饰过程 194
12.5 Java 中的典型实现 197
12.6 小结 198
第 13 章 外观模式 199
13.1 说明 200
13.2 经典回顾 200
13.3 平台、开发语言无关的抽象外观接口——WSDL 203
13.4 Java 中的典型实现 204
13.5 小结 204
第 14 章 享元模式 205
14.1 说明 206
14.2 经典回顾 206
14.3 枚举享元方式 212

14.4 制订共享计划 214
14.5 通过“委托—代理”关系和队列实现异步享元 214
14.6 小结 214
第 15 章 代理模式 215
15.1 说明 216
15.2 经典回顾 216
15.3 远程代理 219
15.4 动态代理 220
15.5 Java 中的典型实现 223
15.6 小结 224
第四篇 行为型模式——算法、控制流的对象化操作 225
第 16 章 职责链模式 226
16.1 说明 227
16.2 经典回顾 227
16.3 非链表方式定义职责链 232
16.4 小结 238
16.5 Java 中的典型实现 239
第 17 章 模板方法模式 240
17.1 说明 241
17.2 经典回顾 242
17.3 类和接口的模板——泛型 247
17.4 系统架构的模板——配置 248
17.5 小结 248
17.6 Java 中的典型实现 249
17.7 自我检验 249
第 18 章 解释器模式 250
18.1 说明 251
18.2 经典回顾 252
18.3 采用正则表达式 256
18.4 采用字典 261
18.5 多级解释器系统 266
18.6 用 XSD 解释自定义业务语言 267
18.7 小结 268
18.8 Java 中的典型实现 269
18.9 自我检验 269

第 19 章 命令模式 270
19.1 说明 271
19.2 经典回顾 272
19.3 打包命令对象 275
19.4 异步命令操作 281
19.5 命令操作队列 284
19.6 小结 284
19.7 Java 中的典型实现 285
19.8 自我检验 285
第 20 章 迭代器模式 286
20.1 说明 287
20.2 经典回顾 288
20.3 Java 内置的迭代器 289
20.4 小结 292
20.5 自我检验 292
第 21 章 中介者模式 293
21.1 说明 294
21.2 经典回顾 295
21.3 根据配置动态协调通知关系 300
21.4 小结 303
21.5 Java 中的典型实现 303
21.6 自我检验 304
第 22 章 备忘录模式 305
22.1 说明 306
22.2 经典回顾 307
22.3 把备忘压栈 312
22.4 备忘录的序列化和持久化 314
22.5 小结 318
22.6 Java 中的典型实现 319
22.7 自我检验 319
第 23 章 观察者模式 320
23.1 说明 321
23.2 经典回顾 324
23.3 面向服务接口的观察者 328
23.4 小结 330
23.5 Java 中的典型实现 330

23.6 自我检验 331
第 24 章 状态模式 332
24.1 说明 333
24.2 经典回顾 334
24.3 状态的序列化和持久化 339
24.4 主动状态对象 341
24.5 小结 342
24.6 自我检验 342
第 25 章 策略模式 343
25.1 说明 344
25.2 经典回顾 345
25.3 策略模式与解释器模式的协作 347
25.4 Java 中的典型实现 348
25.5 小结 348
第 26 章 访问者模式 349
26.1 说明 350
26.2 经典回顾 350
26.3 借助反射或 Dynamic 实现访问者 355
26.4 Java 中的典型实现 358
26.5 小结 358
26.6 自我检验 359

第五篇 GOF 综合练习 360

第 27 章 GOF 部分阶段实践 361
27.1 回顾 GOF 362
27.2 需求的提出 363
27.3 第一轮技术分析 364
27.4 第二轮技术分析 365
27.5 第三轮技术分析 367
27.6 示例实现 368
附录 A Java 和 C#关键字对照表 394

第 12 章

装饰模式

12.1 说明

12.2 经典回顾

12.3 卸载装饰

12.4 通过配置和创建者完成装饰过程

12.5 Java 中的典型实现

12.6 小结

12.1 说明

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

—*Design Patterns : Elements of Reusable Object-Oriented Software*

装饰模式非常强调实现技巧，我们一般用它应对类体系快速膨胀的情况。

在项目中，是什么原因导致类型体系会快速膨胀呢？在多数情况下是因为我们经常要为类型增加新的职责（功能），尤其在软件开发和维护阶段，这方面需求更为普遍。

面向对象中每一个接口代表我们看待对象的一个特定方面。在 Java 编码实现过程中由于受到单继承的约束，我们通常也会将期望扩展的功能定义为新的接口，进而随着接口不断增加，实现这些接口的子类也在快速膨胀，如新增 3 个接口的实现，就需要 8 个类型（包括 MobileImpl），4 个接口则是 16 个类型，这种几何基数的增长我们承受不了。为了避免出现这种情况，之前我们会考虑采用组合接口的方式解决，但客户程序又需要从不同角度看待组合后的类型，也就是可以根据里氏替换原则用某个接口调用这个子类。所以面临的问题是，既要 has a、又要 is a，装饰模式解决的就是这类问题。

什么是 has a 实例？如那个 MobileWithMP3 就是一个手机“还有”MP3 功能。

那什么是 is a 实例？客户程序可以用 MP3 mp3 = new MobileWithMP3()的方式使用这个子类。也就是我们在“面向对象设计原则”一章中提到的里氏替换原则（LSP）。

由于 Java 没有多继承，因此它的 is a 表现为“最多继承一个基类 + 实现一系列接口”的方式，本书在类图和示例中一般都会先抽象接口，为的是在满足客户程序需要的基础上，尽量把这唯一的继承机会保留下来，如留给项目自己的公共抽象基类。

另外，从表象上看，**桥模式解决的问题也是因为继承导致出现过多子类对象，但它的诱因不是因为增加新的功能，而是对象自身现有机制沿着多个维度变化，是“既有”部分不稳定，而不是为了“增加”新的。**

12.2 经典回顾

装饰模式的意图非常明确：动态为对象增加新的职责。

这里有两个关键词：动态和增加，也就是说，这些新增的功能不是直接从父类继承或是硬编码写进去的，而是在运行过程中通过某种方式动态组装上去的。例如：我们在录入文字的时候，最初只需要一个 Notepad 功能的软件，然后增加了很多需求：

字体可以加粗。

文字可以显示为不同颜色。

字号可以调整。

字间距可以调整。

……

不仅如此，到底如何使用这些新加功能，需要在客户使用过程中进行选择，也就是说，新的职责（功能或成员）是需要动态增加的。为了解决这类问题，装饰模式给出的解决方案如图 12-1 所示。

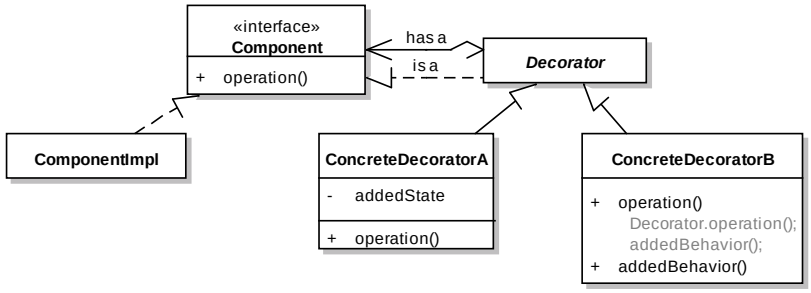

图 12-1 经典装饰模式的静态结构

根据 Notepad 的示例要求，设计如图 12-2 所示的静态结构。

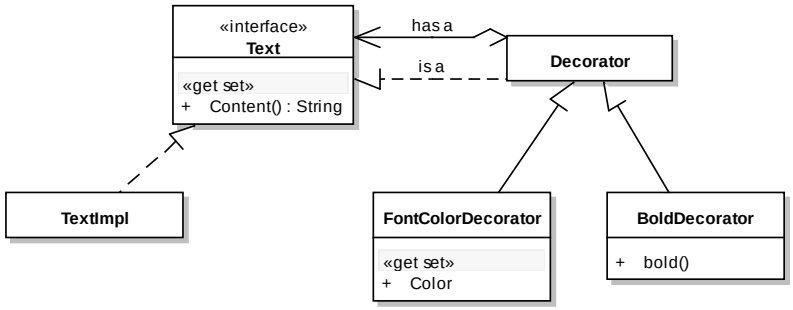

图 12-2 Notepad 应用装饰模式后的静态结构

区别于经典装饰模式设计，在图 12-2 中我们将 Decorator 定义为具体类，而不是抽象类，主要是为了简化示例结构。

首先，我们需要的是显示文字，这时可以指定一个名为Text 的接口，它有名为Content 的读/写属性。

然后，我们把所有需要用来装饰的类型抽象出一个名为 Decorator 的接口，它继承自这个 Text，因此其实体类必须实现 Content 属性方法。

接着，我们把没有任何新增功能的 Text 做出一个“毛坯”的实体类型，命名为 TextImpl。

最后，我们把操作字体 bold()、setColor()的方法填充到每个具体的装饰类型中。

这样，概念上当 TextImpl 需要某种新增功能时，直接为其套上某个具体装饰类型就可以了。这就好像给 TextImpl 类穿上一层又一层的“马甲”。该模式这么做主要是为了适用于哪些情景呢？

在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

毕竟客户程序依赖的仅仅是 Component 接口，至于这个接口被做过什么装饰，只有实施装饰的对象才知道，而客户程序只是依据 Component 的接口方法调用它，这样，装饰类在给 Component 穿上新“马甲”的同时也会随着客户程序的调用一并执行了。

屏蔽某些职责，也就是在套用某个装饰类型时，并不增加新的特征，只把既有方法屏蔽。

也就是说，装饰类不仅能否充当“马甲”，也能起到“口罩”的作用，让 Component 现有的某些方法“闭嘴”。尽管我们使用装饰模式一般是为了增加功能（做“加法”），但并不排斥它也具有方法屏蔽的作用（做“减法”），只不过平时用的比较少而已。

避免因不断调整职责导致类型快速膨胀的情况。

下面看一段示例代码：

Java 抽象部分

```
public interface Text {
    String getContent();
    void setContent(String content);
}

/** implements Text 说明 Decorator is a Text*/
public class Decorator implements Text{

    /** 构造方式注入 has a 的 text 接口 */
    private Text text;
    public Decorator(Text text){
        this.text = text;
    }

    @Override
    public String getContent(){
        return text.getContent();
    }

    @Override
    public void setContent(String content){
        text.setContent(content);
    }
}
```

Java 具体装饰类型

```
/**
 * 具体装饰类，属于“马甲”
 */
class BoldDecorator extends Decorator{

    public BoldDecorator(Text text){
        super(text);
    }

    public String bold(String data){
        return "<b>" + data + "</b>";
    }

    @Override
    public String getContent() {
        return bold(super.getContent());
    }

    @Override
    public void setContent(String content) {
        super.setContent(content);
    }
```

```
}

/**
 * 具体装饰类
 * 属于“马甲”
 */
class ColorDecorator extends Decorator{

    public ColorDecorator(Text text){
        super(text);
    }

    public String setColor(String data){
        return "<color>" + data + "</color>";
    }

    @Override
    public String getContent() {
        return setColor(super.getContent());
    }

    @Override
    public void setContent(String content) {
        super.setContent(content);
    }

}

/**
 * 具体装饰类
 * 属于“口罩”
 */
class BlockAllDecorator  extends Decorator{

    public BlockAllDecorator (Text text){
        super(text);
    }

    @Override
    public String getContent() {
        return null;
    }

    @Override
    public void setContent(String content) {
        super.setContent(content);
    }

}
```

Unit Test

```
Text text;

@Before
public void setUp(){
    text = new TextImpl();
```

```
}

/** 验证套用单个装饰对象的效果 */
@Test
public void testSingleDecorator(){
    text = new Decorator(text);

    //下面开始套用装饰模式，开始给 text 穿“马甲”
    text = new BoldDecorator(text);
    text.setContent("H");
    assertEquals("<b>H</b>", text.getContent());
}

/** 验证套用多个装饰对象的效果 */
@Test
public void testMultipleDecorators(){
    text = new Decorator(text);

    //下面开始套用装饰模式，开始给 text 穿“马甲”
    text = new BoldDecorator(text);
    text = new ColorDecorator(text);
    text = new BoldDecorator(text);
    text.setContent("H");
    assertEquals("<b><color><b>H</b></color></b>",text.getContent());
}

/** 验证装饰类型的撤销（/“口罩”）效果*/
@Test
public void testBlockEffectDecorator(){
    text = new Decorator(text);

    //下面开始套用装饰模式，开始给 text 穿“马甲”
    text = new BoldDecorator(text);
    text = new ColorDecorator(text);
    text.setContent("H");
    assertEquals("<color><b>H</b></color>", text.getContent());

    //下面开始套用装饰模式，开始给 text 戴“口罩”
    text = new BlockAllDecorator(text);
    assertNull(text.getContent());
}
```

从上面的示例中不难看出，装饰模式实现上特别有技巧，也很“八股”，它的声明要实现 Component 定义的方法，但同时也会保留一个对 Component 的引用，Component 接口方法的实现其实是通过自己保存的 Component 成员完成的，而装饰类只是在这个基础上增加一些额外的处理。而且，使用装饰模式不仅仅是为了“增加”新的功能，有时候我们也用它“撤销”某些功能。项目中我们有 3 个要点必须把握：

Component 不要直接或间接地使用 Decorator，因为它不应该知道 Decorator 的存在，**装饰模式的要义在于通过外部 has a + is a 的方式对目标类型进行扩展，对于待装饰对象本身不应有太多要求。**

Decorator 也仅仅认识 Component。**抽象依赖于抽象、知识最少。**

某个 ConcreteDecorator 最好也不知道 ComponentImpl 的存在，因为 ConcreteDecorator 只能服务于这个 ComponentImpl（及其子类）。

此外，使用装饰模式解决一些难题的同时，我们也要看到这个模式的缺点：

开发阶段需要编写很多 ConcreteDecorator 类型。

运行时动态组装带来的结果就是排查故障比较困难，从实际角度看，Component 提交给客户程序的是最外层 ConcreteDecorator 的类型，但它的执行过程是一系列 ConcreteDecorator 处理后的结果，追踪和调试相对困难。

在实际项目中，我们往往会将一些通用的功能做成装饰类型单独编译，而且一般也鼓励这么做，因为可以减少重复开发，但这样会人为增加排查和调试的难度。好在反编译 Java byte code 不是太难。

12.3 卸载装饰

配合前面创建者模式中提到的装配/卸载思路，我们也可为 Concrete Decorator 增加卸载功能。**这对于那些需要长时间驻留内存并需要根据外部情况动态装饰/卸载装饰的对象尤为重要。**

与前面章节 Builder 的卸载不同，Decorator 同时有 has a 和 is a 两层关系，所以原理上卸载某个 Decorator 需要重新调整继承关系，修改 has a 的指向，感觉这样才是比较彻底的卸载装饰。但工程中类似的实现往往涉及复杂的动态 Byte Code 编译，成本和代价远远高于示例中那样从毛坯重新装修。不过，我们可以使用一些技巧，如在 Decorator 定义时增加类似名为 Enabled 属性或者参考后面要介绍的备忘录模式，在每个操作执行前，就记录未来专门用于还原的内容，间接等价于卸载了某个装饰类型的装饰效果。

12.4 通过配置和创建者完成装饰过程

一般，实现装饰模式都要使用“一层套一层”的办法，由于它的“动态”性，每次要“套用”的数量不同，因此，一般的介绍中没有使用创建者模式协助完成这

个“多个步骤”的创建过程。不过分析一下，如果不涉及复杂构造参数的情况，其实装饰模式的实现过程是比较固定的。其代码如下：

Unit Test

```
// 建立对象，并对其进行两次装饰
@Test
public void testBuildDecorators(){
    text = new Decorator(text);
    text = new BoldDecorator(text);
    text = new ColorDecorator(text);
}
```

客户程序其实已经显式依赖了所有的具体装饰类型，而实际项目中需要考虑隔离客户程序对这一组具体装饰类型的直接引用。考虑到 “过程相对稳定”的前提下需要通过“多个步骤”解决，是不是可以启用创建者模式协助解决呢？是的。

不仅如此，为了便于在生产环境中动态调整这些装饰类型，我们可以将所有装饰类型定义在配置文件中，由创建者根据配置信息动态完成装饰过程。从经典装饰模式实现中我们可以发现，对于创建者而言，如果能掌握装饰类型的构造函数，一般情况下即可完成类似的工作。

学习设计模式很忌讳“模式先行”，即在遇到问题的时候先考虑如何套用模式，这种做法不可取。模式一般用于在开发中已经发现问题，尤其是发现变化并多次修改后的情景。我们在之前的示例已经“中规中矩”地完成了经典装饰模式的实现，但回头看看才发现客户程序会不断受到 ConcreteDecorator 的影响，在此基础上我们要考虑做一些改变。

综合上面的需要，我们尝试引入创建者模式解决客户程序对各个具体装饰类型的直接依赖，示例代码如下：

marvellousworks.practicalpattern.xml 定义装饰模式的配置节（已剔除掉其他模式配置定义部分）

```
<?xml version="1.0" encoding="UTF-8"?>
<practicalpatterns>

    <configSections>
       <add name="decorator" type="…….DecoratorConfigSection"/>
    </configSections>

<!—装饰模式的配置节-->
    <decorator>
        <!—所有已经安装的装饰类型的配置元素集合-->
        <decorators>
```

```
            <add name="bold" type="…….BoldDecorator"/>
            <add name="color" type="…….ColorDecorator"/>
            <add name="block" type="…….BlockAllDecorator"/>
        </decorators>
        <!—运行时会实际执行的装饰类型的配置元素集合-->
        <runtime>
            <add name="bold"/>
            <add name="color"/>
        </runtime>
    </decorator>

</practicalpatterns>
```

Java 完成装饰模式配置解析(OCM)类型

```
public class DecoratorConfigSection
    extends ConfigSection{

    public static final String NAME = "decorator";
    private static final String DECORATORS_ITEM = "decorators";
    private static final String RUNTIME_ITEM = "runtime";


    //保存 Decorator 名称与类型的映射关系
    protected Map<String, Class<?>>;
    //保存实际运行时生效的 Decoraotr 列表
    protected List<Class<?>> runtimeDecorators;

    public DecoratorConfigSection()…}

    /**
     * 配置文件中登记的已经安装的装饰类型
     * @return   映射关系
     */
    public Map<String, Class<?>> getDecoratorMappings(){
        return decoratorMappings;
    }

    /**
     * 配置文件中登记运行时生效的装饰类型
     * @return
     */
    public List<Class<?>> getRuntimeDecorators(){
        return runtimeDecorators;
    }
}
```

Java 完成实现装饰过程的创建者对象

```
/** 完成实现装饰过程的创建者对象 */
public class DecoratorBuilder {
```

```
    //配置文件中定义的各个装饰类型的构造函数对象
    static List<Constructor<?>> constructors = null;

    static{
         try {
             DecoratorConfigSection section = (DecoratorConfigSection)
             (ConfigBroker.getSection(DecoratorConfigSection.NAME));
             if((section.getRuntimeDecorators() != null) ||
                 (section.getRuntimeDecorators().size() > 0)){
                 constructors = new ArrayList<>();
                 Factory factory = new FactoryImpl();
                 for(Class<?> clazz :section.getRuntimeDecorators()){
                     constructors.add(
                        factory.getConstructor(clazz,
                            new Class[]{Text.class}));
                 }
             }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /** 根据配置信息完成装饰过程 */
    public Text buildUp(Text text)
        throws …{
        if(text == null)
            return null;
        text = new Decorator(text);
        if((constructors != null) && (constructors.size() > 0)){
            for(Constructor<?> constructor : constructors)
                text = (Text)(constructor.newInstance(text));
        }
        return text;
    }


    /**
     * 获得所有运行时装饰类型的构造函数列表
     * @return   构造函数列表
     */
    public List<Constructor<?>> constructors (){
        return constructors;
    }
}
```

Unit Test

```
public class DecoratorBuilderFixture {
    DecoratorBuilder builder;


    @Before
    public void setUp(){
        builder = new DecoratorBuilder();
        //输出构造函数的信息
```

```
        for(Constructor<?> constructor : builder.constructors())
            trace(TextHelper.toString(constructor));
    }

    @Test
    public void testDecoratorProcessByBuilder() …{
        Text text = builder.buildUp(new TextImpl());
        text.setContent("H");
        //验证准从配置文件定义要求，仅依次执行 Bold 和 Color 两个装饰类型
        assertEquals("<color><b>H</b></color>", text.getContent());
    }

}
```

Console 窗口

```
BoldDecorator(Text)
ColorDecorator(Text)
```

从示例中不难看出，通过引入创建者，客户类型与多个装饰类型的依赖关系变成了客户类型与创建者之间 1:1 的依赖关系，而这里 Builder 类型可以进一步设计为泛型 Builder（例如：定义为 DecoratorBuilder<T>），这样可以减少多个客户类型使用装饰模式时“繁文缛节”的构造过程；另外，装配对象的加入也给我们从外部配置目标类型及其相关装饰类型的机会。

12.5 Java 中的典型实现

下面的类型、方法采用了装饰模式，感兴趣的读者可以通过反编译参考。

java.io.InputStream、OutputStream,、Reader、Writer 及它们的子类。

javax.servlet.http.HttpServletRequestWrapper 和 HttpServletResponseWrapper。

12.6 小结

作为整个设计模式中非常讲究技巧的一个模式，经典装饰模式通过继承方式实现了对新功能的拓展，如 I/O 处理中的 Stream 家族就是典型的装饰模式，加密、缓存、压缩等机制都通过专门的 has a + is a 的子类完成。**但在实际项目中，装饰模式的应用比例不高，这主要是因为追踪和调试相对困难，在多数情况下项目中更愿意采用多接口组合的方式。**

不过根据应用运行环境的要求，单纯的装饰模式一般只能一味增加和扩充对象特性，无法根据上下文要求动态装载、卸载这些扩充的特性，为此本章借鉴前面的创建者模式讨论了能加载、卸载的装饰过程。

根据开发语言的特点，为了简化装饰过程，尽量减少继承带来的类型依赖，大型项目中还会配合拦截机制将装饰内容直接做成标注，配合代理类型以“横切”的方式装饰目标类型和接口，最大限度地减少装饰类型与目标实体类型间的依赖，Java EE 中这种方式被普遍采用。不过此方式的使用代价是**实现成本很高，**执行过程中如果不借助缓冲，**其执行效率也会受到一定的影响**。

第 27 章

GOF 部分阶段实践

27.1　回顾 GOF

27.2　需求的提出

27.3　第一轮技术分析

27.4　第二轮技术分析

27.5　第三轮技术分析

27.6　示例实现

27.1 回顾 GOF

GOF 给我们最大的启示就是先把紧密耦合在一起的对象分解，然后通过抽象或者增加第三个对象的方法再把它们联系在一起，实现一个更加松散的结构，经过一系列模式的学习我们发现 GOF 各模式的分析过程存在相似性：

首先，分析所面临的问题，然后把其中关键对象的静态结构和动态结构用类 UML 图形表示出来。该步骤让我们从具体的业务环境过渡到软件领域。

找到其中变化的部分，分析是哪个（或哪几个）对象在导致变化。

如果对象的不确定性来自于构造过程，那么借助创建型模式，控制目标对象的数量或者把创建工作交给一个独立的对象完成，这样，业务对象自身不会为了构造其他对象产生直接依赖。当然，实现该目标有个前提，那就是待构造的对象先被抽象。

如果对象的不确定性来自结构，则可以考虑结构型模式，用一个相对抽象的对象协调结构上的依赖关系。

如果不确定性来自执行过程，可以考虑行为型模式，行为型模式多数时候是对执行逻辑复杂性的进一步分解，如如果某些操作、交互过程过于复杂，这时就需要对它们进行分解。行为型的思维方法就把这些复杂而且易变操作和交互对象化，同时抽象它们的行为，确保这些对象后续可以被替换。

经过前面的介绍，我们也能感觉到模式间存在转换关系，设计模式中也给出了模式之间协作的概览图。实践中，导致变化的因素往往不止一个，所以概览图相当于一个指南：当我们在应用某个模式的时候，如果同时涉及特定情景的话，那么就可以考虑采用另外某种模式。

很多时候我们都被灌输一种信念：设计模式仅仅是一种设计思想。

根据我们自己的经验，在某几个点总会有一些变动，很多情况下根据经验应用模式方法去解决这些可预见的变化反而是比较“偷懒”的途径，偶尔根据上下文套用模式解决方法（甚至于“形而上学”）也未尝不是一个办法。

前面每章都在谈一个特定的模式，但这样可能整体感不强，为了加深读者对 GOF 的认识，本章我们进行综合练习，尝试设计一个比较通用的用户认证模块。

本章作为示例，并没有采用 JAAS（Java 认证和授权服务：Java Authentication and Authorization Service）的认证、授权框架，而是自己定义了一个模拟的认证模块，实际项目中为了认证模块的标准化，同时也便于与其他商用中间件产品集成，还是建议采用 JAAS 的框架进行扩展。

27.2 需求的提出

示例项目的需求如下：

MarvellousWorks 公司要开发一个用户认证模块。

用户认证的凭据有很多种方式，已知的包括如下几种：

- USB Key（或 IC 卡）方式。
- 用户名口令方式（UserName / Password）。
- 集成微软公司的活动目录（Active Directory）方式。

MarvellousWorks 公司的 USB Key 是外购第三方厂商的，设备驱动也是对方提供的。

对于 MarvellousWorks 公司来说，用户认证过程是个比较敏感的过程，可能会不断增加额外的非功能性需求，如记录审计日志、监控登录响应时间等。

认证过程及采取何种非功能性控制措施要与 MarvellousWorks 公司整体的安全策略一致，而且要根据策略调整及时“热”（在线）调整并生效。

为了更明确地划清中间件服务器与认证服务器的边界，认证机制设计上就要提供一定的远程访问能力。

MarvellousWorks 公司认证系统最终的部署结构大致如图 27-1 所示。

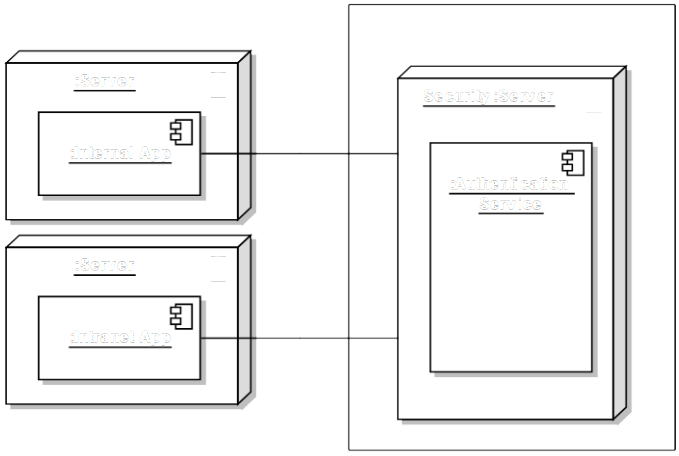

图 27-1 认证系统部署结构

本册作为 GOF 部分介绍，仅讨论认证服务（Authentication Service）进程内逻辑的实现梗概，完整的示例模型将在下册完成架构模式的介绍后进一步展开。

27.3 第一轮技术分析

根据描述的需求，我们先作为第一轮技术分析，大体描绘出如图 27-2 所示的认证服务的交互过程：

（1）即客户程序调用一个名为“认证子（Authenticator）”的对象执行认证。

（2）但事实上认证子除了自身做一些管理工作外，实际的认证过程是交给认证服务提供者（Provider）来完成的。

（3）认证服务提供者将认证结果反馈给认证子，然后认证子向客户程序反馈。

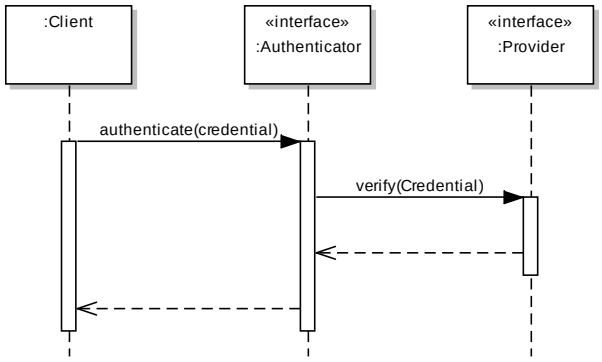

图 27-2 认证服务 Layer 1 的交互过程

而参与执行的两个接口的静态结构如图 27-3 所示。

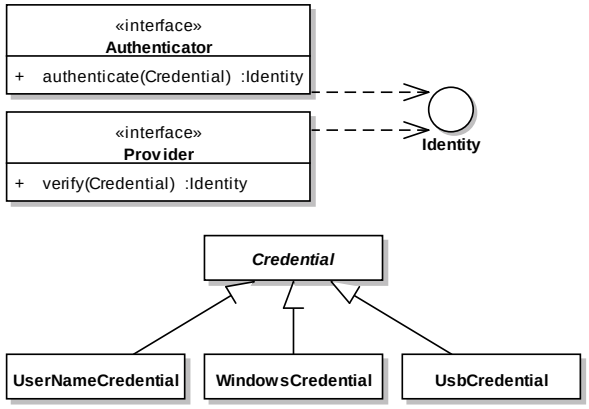

图 27-3 认证服务 Layer 1 的静态结构

虽然上面的设计大致体现了面向对象设计的封装、继承和多态的要求，但有些细节很值得推敲：

图 27-2 中， Authenticator 通过选择指定的 Provider 完成对特定类型凭据的认证工作，但谁来做这个“选择”？如果让 Authenticator 完成，那么它就会和具体的 Provider 实体类产生依赖。

对于 USB Key 认证方式，由于设备和驱动是第三方厂商提供的，因此适用于它的 Provider 存在很大的不确定性。

考虑到不同 Provider 运行时需要一定的配置信息，因此 Provider 对象的创建过程不是简单的一 new()而就。

现有的设计中没有考虑到非功能性控制需求。

回忆前面介绍的 GOF 内容，我们可以在这些变化点应用一些设计模式技巧：

增加一个工厂类型，隔离 Authenticator 和具体 Provider 的关系。

根据认证系统对于 USB Key 的使用技术要求设计 Provider 需要的访问接口，通过增加适配器**隔离底层驱动接口的变化。**

提供具有“装配”能力的构造类型，将配置信息写入 Provider。

将非功能性控制策略进行抽象，同时增加“逐个筛选”的方式，隔离认证过程与具体控制要求。

融合这些调整内容后，我们进入第二轮技术分析。

27.4 第二轮技术分析

增加上述模式处理后，认证服务新的静态结构如图 27-4 所示。

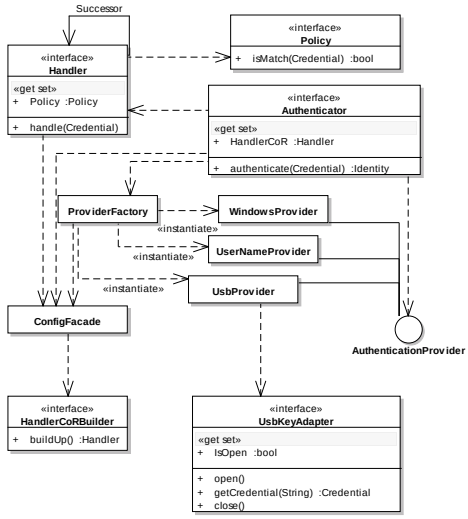

图 27-4 认证服务 Layer 2 的静态过程（为了简化，将部分 Layer 1 已有的部分省略）

在图 27-4 中，我们通过一些模式技巧对原有设计进行优化。直观上，相对图 27-3 而言图 27-4 的复杂度增加了，但其目的是解决前述设计中的一些问题，如表 27-1 所示。

表 27-1 第二次技术分析中模式用途关系

模式	用途
桥模式（Bridge）	首先，从认证服务局部架构分析看，由于影响认证的变化因素比较多，因此采用桥模式“分片包干”，将变化因素逐个定义为不同的接口。 而且，由于变化的因素不是单一方向的正交变化，因此采用具有分支的桥模式
简单工厂模式(Simple Factory)	新增 ProviderFactory 工厂类型，用于隔离 Authenticator 与具体 Provider 类型的直接依赖关系
适配器模式（Adapter）	为保证上层认证逻辑的稳定，对 USB Key 设备驱动的接入行为进行约定，定义抽象的适配器接口 UsbAdapter，要求第三方供应商实现
策略模式（Strategy）	将认证过程各种非功能性的控制策略抽象为认证策略 Policy，通过 isMatch()方法确认策略是否适用于具体的凭证类型
职责链模式（CoR）	为了简化各种非功能性控制策略的适用过程，将所有的 Handler 类型实例以链式方式组织，以“逐个筛选”的方式隔离认证过程与具体非功能性控制策略的依赖关系
创建者模式（Builder）	为了解决 Handler 职责链的装配问题，抽象出 HandlerCoRBuilder 接口完成该部分工作，确保 ProviderFactory 反馈的结果不仅只是 new()出一个 Handler 实例，还完成对其后继节点和 Policy 的装配工作
外观模式（Façade）	作为一个面向长期运维的模拟生产系统，为了便于管理，需要定义一组相对复杂的配置结构，但上层逻辑又需要通过一些简单的接口获得配置信息，为此引入外观模式，将配置信息的访问集中于 ConfigFacade 类型

完成第二轮技术分析后，我们还有两个主要的需求没有实现：

认证过程及采取何种非功能性控制措施要与 MarvellousWorks 公司整体的安全策略一致，而且要**根据策略调整及时“热”（在线）调整并生效。**

为了划清普通中间件服务器与认证服务器的边界，认证机制要**提供一定的远程访问控制能力。**

为了让 HandlerCoRBuilder 能够随时根据配置要求“在线”调整装配过程，需要有一个对象专门“盯着”配置信息，一旦发现配置变化就及时通知 HandlerCoRBuilder，让它根据最新的配置更新控制措施 CoR 的装配过程。

另外，远程访问和本地访问在多数情况下颗粒度不同：本地访问时，由于处理都在进程内部，因此可以提供一系列细颗粒度的访问接口；远程访问时由于通信效率较低，一般设计较粗颗粒度的接口，为了约束远程访问接口的颗粒度，但又避免陷入具体远程通信技术细节，需要考虑采用模式从结构上隔离远程接口内容。

针对上述问题，我们继续推演现有设计，完成第三轮技术分析。

27.5 第三轮技术分析

新的静态结构如图 27-5 所示。

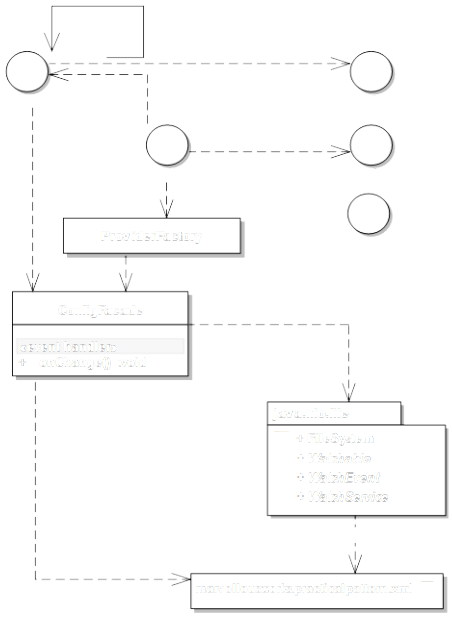

图 27-5 认证服务 Layer 3 的静态过程（为了简化，将部分 Layer 1、2 已有的部分省略）

在图 27-5 中，我们继续通过模式技巧对第二次技术分析的设计进行优化，处理内容如表 27-2 所示。

表 27-2 第三次技术分析中模式用途关系

模式	用途
观察者模式（Observer）	为了能够根据配置变化及时更新对象构造过程，设计上采用观察者模式“盯住”配置文件的变化。 这里采用 Java SE 7 中 java.nio.file 包的对象完成
代理模式（Proxy）	考虑到远程访问中接口颗粒度一般较粗，远程访问由于通信协议的不同，实现方式差别迥异，因此设计上定义独立的远程接口供远程客户程序调用。 实际项目中可以通过配置不同的 EJB 3.x 标注定义哪些可以由远程代理实现，哪些只针对本地。 实际项目中往往并不需要真正独立的 ProviderRemote 接口，因为很多工作可以通过 EJB 3.x 的标注完成

其中，由于非功能性控制策略属于全局的控制措施，其每次构造控制策略 CoR 构造过程也需要一定的代价，为此将它也收回到 ConfigFacade 中，由该外观类型缓存并根据配置信息的变化动态更新。

完成第三轮技术分析后，下面通过一个实例程序验证上述设计，确认上述设计是否真的能够适应需求部分的变化。

27.6 示例实现

考虑本册主要针对 GOF23 部分，因此示例中没有实现远程代理部分，这部分内容纳入后续架构模式的示例。

1．功能划分

为了自顶向下划分示例逻辑，我们通过 3 个包组织相关类型，划分如图 27-6 所示。

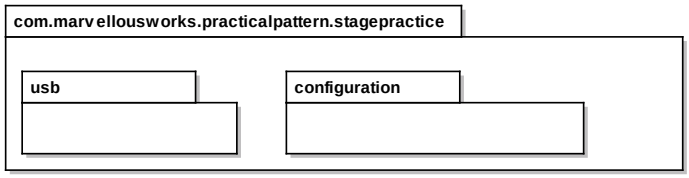

图 27-6 认证服务实现部分的功能划分

其中，每个包的职能如下：

com.marvellousworks.practicalpattern.stagepractice：保存认证服务运行所需要的公共接口和部分内置实现类型。

com.marvellousworks.practicalpattern.stagepractice.usb：仅针对 USB Key 方式，定义约束外部第三方驱动的接口定义。

com.marvellousworks.practicalpattern.stagepractice.configuration：定义各类配置对象。

2．配置结构设计

为了便于前述技术分析的内容可以通过“即插即用”的方式接入，为认证服务后续的扩展提供基础，我们采用配置为中心的开发思路，在实际编码前先完成配置文件的结构。

配置文件的优势在“重新研读 Java 语言”一章中已经介绍，而且对于什么内容需要定义在配置文件中也做了说明，下面根据第三轮技术设计的结果，考察其中需要通过配置文件接入的接口，包括：

Authenticator：定义认证服务整体对外的功能接口（认证子）。

HandlerCoRBuilder：定义装配 Handler 职责链的创建者。

UsbKeyAdapter：USB Key 驱动适配接口。

Policy：定义认证过程非功能性控制策略对象。

Provider：定义针对具体凭证类型的认证功能提供者。

Handler：定义认证过程非功能性控制处理对象。

对于一个运行系统，上述 6 个接口中：Authenticator 针对具体特定的认证服务运行时而言是单一的；UsbKeyAdapter 在一段时间内也是唯一的；HandlerCoRBuilder 功能固定，只需要构造一个 Handler 的链式结构，因此也是单一的。而另外 3 个则会同时存在多个可选项。另外，考虑到凭证类型除了之前定义的“USB Key”、“用户名口令”、“活动目录”外，后续还可能不断增加，因此也存在多个选项。

这样，我们定义的配置结构如下：

marvellousworks.practicalpattern.xml　　示例程序的配置节

```
<stagepractice>

  <add name="authenticator" type="配置元素"/>
  <add name="handlerCorBuilder" type="配置元素"/>
  <add name="usbAdapter" type="配置元素"/>

  <add name ="credentials" type="配置元素集合"/>
  <add name ="providers" type="配置元素集合"/>
  <add name ="policies" type="配置元素集合"/>
  <add name="handlers" type="配置元素集合"/>

</stagepractice>
```

其中，为了便于和公司的其他服务集成，整个认证服务定义了自己独立的配置节“stagePractice”，将每个单项配置项定义为一个个独立的配置元素，而将每个集合性质的配置项统一定义为一个配置元素集合。

进一步，考虑到 handlerCorBuilder 用于构建 Handler，且仅与“handlers”项配置节有关系，因此可以将它作为“handlers”配置元素的属性定义；而 UsbKeyAdapter 只适用于 USB Key 凭证的认证方式，因此只适合用于特定的 Provider，而不应作为一个配置项独立出现。

所以，进一步整理后的配置如下：

marvellousworks.practicalpattern.xml 示例程序的配置节

```
<stagepractice>

  <add name="authenticator" type="配置元素"/>

  <add name ="credentials" type="配置元素集合"/>
  <add name ="providers" type="配置元素集合"/>
  <add name ="policies" type="配置元素集合"/>
  <add name="handlers" type="配置元素集合" builder="className"/>

</stagepractice>
```

再进一步，为了便于后续各种工具的界面显示和用户操作管理，做如下设计：

统一约定将每个配置元素集合中各个配置元素的“key”属性作为配置元素的逻辑名称，而用“value”属性登记对应实体类型的类型名称。

根据图 27-7 中非功能性控制措施及适用策略的静态关系，我们需要为每个 Handler 配置元素设置 3~4 个属性，因此需要为它定制一个独立的配置元素和配置元素集合。

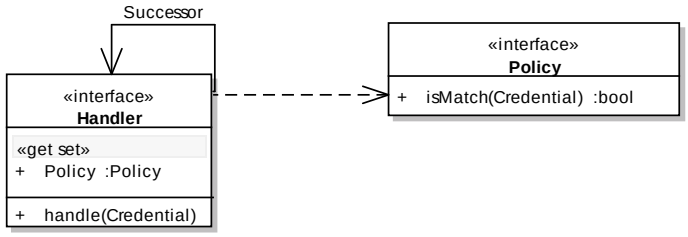

图 27-7 认证过程中非功能性控制措施及适用策略的静态关系

这样，我们为其设计的配置元素结构如下：

marvellousworks.practicalpattern.xml 示例程序的配置节

```
<stagepractice>

  <add key="authenticator" value="配置元素"/>

  <add key ="credentials" value="配置元素集合">
    …
  </add>
  <add key ="providers" value="配置元素集合">
    …
  </add>
```

```
  <add key ="policies" value="配置元素集合">
    …
  </add>
  <add key="handlers" value="配置元素集合" builder="className">
    <add key="" value="" seq="" po=""/>
    <add key="" value="" seq="" po=""/>
    …
  </add>

</stagepractice>
```

按照以上描述，每个 handler 配置元素中“key“属性表示该 Handler 的逻辑名称 “value”属性表示对应实体类型的类型名称，而“seq”属性代表该 Handler 在整个职责链中的执行次序，“po”（Policy）表示该 Handler 适用的 Policy 的逻辑名称。

基于上述分析，将各个确定的配置元素 `Authenticator` 和其他几个配置元素集合定义为独立命名的配置元素后，配置元素结构简化为：

marvellousworks.practicalpattern.xml 示例程序的配置节

```
<stagepractice>
  <authenticator value="……"/>
  <credentials>
    …
  </credentials>
  <providers>
    …
  </providers>
  <policies>
    …
  </policies>
  <handlers builder="className">
    <add key="" value="" seq="" po=""/>
    <add key="" value="" seq="" po=""/>
    …
  </handlers>
</stagepractice>
```

基于上述分析，我们配置部分的静态结构如图 27-8 所示。由于涉及的类型较多，且自成“子系统”，为了减少上层认证逻辑与底层配置对象的依赖关系，我们采用外观模式，通过单一的 ConfigFacade 类型向上层逻辑提供配置访问接口。

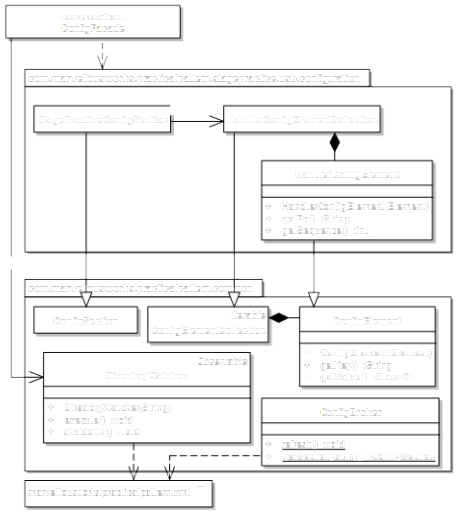

图 27-8 认证服务配置子系统的静态关系

以往我们常常会提醒自己重构并优化编写的代码，但事实上配置文件结构是否合理很大程度上会影响代码的结构及代码的重构，上述我们还对配置文件的结构做了几次重构和优化，目标是确保包装配置访问的代码不会为了一个很“别扭”的配置结构而“削足适履”，不得不去编写配置访问类型和配置访问逻辑。

3．实现配置访问外观类型

上面，我们完成了对于配置结构的设计，但这只是停留在“纸面上”，为了让 ConfigFacade 变成一个可以运行的类型，我们还需要编码实现它。过程如下：

（1）根据图 27-4 所示的依赖关系，我们先要设计、实现用于装配 Handler 的 HandlerCoRBuilder。

（2）完成 ConfigFacade 的主干逻辑。

（3）为了保证 ConfigFacade 能够根据配置修改动态更新，再为其增加 Directory Watcher 的实现部分。

首先，设计 HandlerCoRBuilder 部分，静态结构如图 27-9 所示。

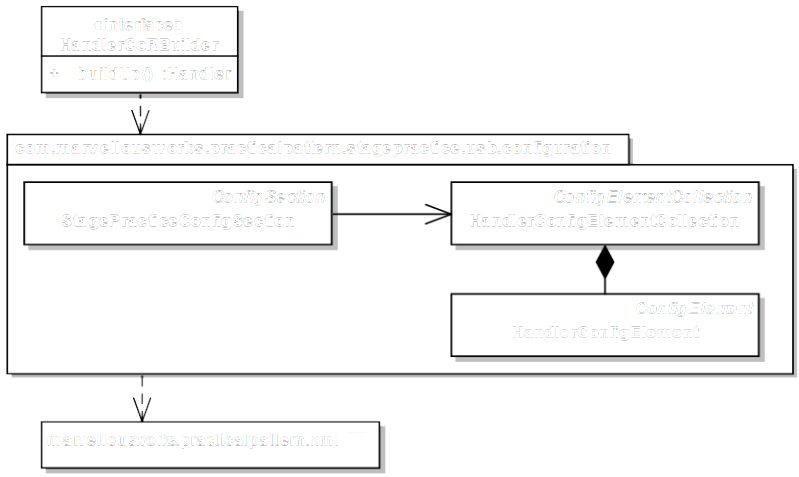

图 27-9 HandlerCoRBuilder 静态结构

尽管我们定义了 HandlerCoRBuilder，允许外部程序独立定义其实现，但在实际项目中，尤其对于类似示例这样的公共代码库，一般我们还是会补充一个功能最“原始”、“简单”的默认实现类，一方面是简化客户程序的使用；另一方面也便于通过单元测试验证自己设计的接口是否真的合适。

Java HandlerCoRBuilder

```
/** 装配 Handler CoR 的创建者*/
public interface HandlerCoRBuilder {

    /**装配
     * @return Handler CoR 的入口节点*/
    Handler buildUp();
}
```

Java HandlerCoRBuilderImpl

```
public class HandlerCoRBuilderImpl implements HandlerCoRBuilder{

    private StagePracticeConfigSection section;

    @Override
    public Handler buildUp(){
    trace("HandlerCoRBuilderImpl.buildUp()");
    HandlerConfigElement[] config = section.getHandlers();
    if((config == null) || (config.length == 0))
        return null;
```

```
    /**  建立 Handler CoR 的链表关系*/
    Handler root = null;
    //先定位到链表的最后一个节点
    root = add(config[config.length - 1], null);
    //然后依次将各节点接入链表
    if(config.length > 1)
        for(int i=config.length-2; i>=0; i--)
            root = add(config[i], root);
    traceHandlerCoR(root);
    return root;
    }

    /** 在 Handler CoR 链表中追加一个新的 Handler 节点*/
    private Handler add(HandlerConfigElement element,Handler successor)
        throws
            InstantiationException,
            IllegalAccessException{

        return ((Handler)(element.getValue().newInstance()))
            .setPolicy((Policy)(section.getPolicyType(element.getPo())
            .newInstance()))
            .setSuccessor(successor);
    }

    …
}
```

下面开始设计 ConfigFacade 的主干逻辑，通过它调度各配置类型及 HandlerCoRBuilder，静态关系如图 27-10 所示。

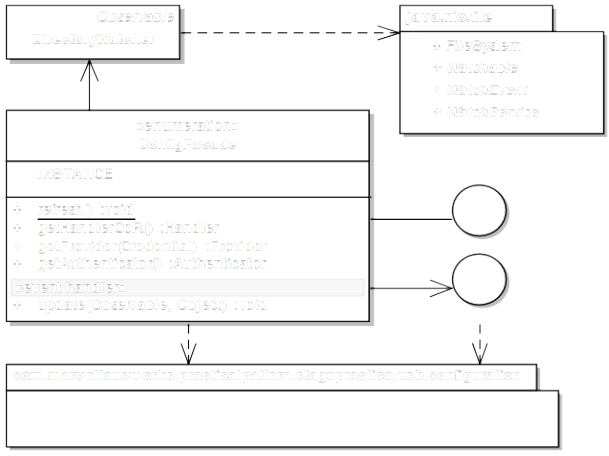

图 27-10 配置访问外观类型的静态结构

其中，静态私有成员 watcher 相当于配置文件的观察者，用于“盯着”配置文件是否变化，而 refresh ()方法则负责加载配置信息，然后将配置信息转化为上层应用所需的简化数据结构。实现上，DirectoryWatcher 封装了 java.nio.file 包的相关接口，负责将文件系统的变更作为通知对外发布。为了充分利用 Java SE 内置的观察者机制，实现上还将 DirectoryWatcher 设计为 java.util.Observable 的子类，同时令 ConfigFacade 实现了 java.util.Observer 接口，这样，两者就可以平滑地建立起通知发布后的接收通道，确保 ConfigFacade 可以及时根据配置文件的变化调用 refresh() 方法更新配置信息。

下面是裁剪后的实现片段：

Java

```
/** 为了便于客户程序访问配置系统设计的 Facade 类型 */
public enum ConfigFacade implements Observer{

    /** 单件方式定义的 ConfigFacade */
    INSTANCE;

    //为减少重复建立 Handler CoR，缓存用于共享的 Handler CoR 实例
    private static Handler handlerCoR;
    //认证服务实际使用的认证子类型
    private static Class<? extends Authenticator> authenticatorType;
    //登记 Credential 及其对应的 Provider
    private static Map<Class<? extends Credential>, Provider> providers;

    static{
        refresh();
    }

    /** 根据配置文件的变化更新各缓存对象 */
    public static void refresh()……

    /** 获得 Handler CoR */
    public Handler getHandlerCoR(){return handlerCoR;}

    /**
     * 根据 Credential 类型返回适用的 Provider 实例
     * 相当于 ProviderFactory 应尽的职责
     */
    public Provider getProvider(Credential credential)…

    /** 根据配置信息获得 Authenticator 类型实例 */
    public Authenticator getAuthenticator() …
}
```

然后，我们为 ConfigFacade 增加“动态”更新的功能。

Java DiretoryWatcher

```
/** 文件系统事件类型*/
public final class FileSystemEventArgs {
    private final String fileName;
    private final Kind<?> kind;

    public FileSystemEventArgs(String fileName, Kind<?> kind){
        this.fileName = fileName;
        this.kind = kind;
    }

    /** 文件的路径 */
    public String getFileName(){return fileName;}

    /** 操作类型：变更、创建、删除 */
    public Kind getKind(){return kind;}
}

/**
 * 监控一个目录内文件的更新、创建和删除事件（不包括子目录）
 *
 * 由于 Java 没有类似.NET 源生的事件机制
 * 因此实现上采用了 Java SE 自带的 Observer/Observable，对外抛出“假”事件
 * 适于 Java SE 7 及以后版本
 */
public class DirectoryWatcher extends Observable{

    private WatchService watcher;
    private Path path;
    private WatchKey key;
    private Executor executor = Executors.newSingleThreadExecutor();

    /**
     * 借助线程池将 Directory Watcher 作为独立的后台任务执行
     * 对于文件系统的更新则通过 Observable/Observer 的预定关系发布
     */
    FutureTask<Integer> task = new FutureTask<Integer>(
            new Callable<Integer>(){
                public Integer call() throws InterruptedException{
                    processEvents();
                    return Integer.valueOf(0);}});
    @SuppressWarnings("unchecked")
    static <T> WatchEvent<T> cast(WatchEvent<?> event) {
        return (WatchEvent<T>) event;
    }

    /**
     * 构造
     * @param dir 待监控的目录
     */
    public DirectoryWatcher(String dir) throws IOException {
        watcher = FileSystems.getDefault().newWatchService();
```

```
        path = Paths.get(dir);
        //监控目录内文件的更新、创建和删除事件
        key = path.register(watcher, ENTRY_MODIFY);
    }
    /** 启动监控过程 */
    public void execute(){
        //通过线程池启动一个额外的线程加载 Watching 过程
        executor.execute(task);
    }
    /** 监控文件系统事件 */
    void processEvents() {
        while (true) // 等待直到获得事件信号
            …
    }

    /** 通知各个 Observer，目录有新的更新事件*/
    void notifiy(String fileName, Kind<?> kind){
        //标注目录已经被做了更改
        setChanged();
        //主动通知各个观察者目标对象状态的变更
        //这里采用的是观察者模式的“推”方式
        notifyObservers(new FileSystemEventArgs(fileName, kind));
    }
}
```

Java 为 ConfigFacade 增加动态功能

```
public enum ConfigFacade implements Observer{

    private static DirectoryWatcher watcher;
    /**
     * 单件方式定义的 ConfigFacade
     */
    INSTANCE;
    static{
        try {
            refresh();
            watcher = new DirectoryWatcher(USER_DIR);
            watcher.addObserver(INSTANCE);
            watcher.execute();
        } catch (XPathExpressionException | InstantiationException
                | IllegalAccessException | IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * event handler
     * 响应配置文件的修改
     */
    @Override
    public void update(Observable o, Object eventArgs) {
        FileSystemEventArgs args = (FileSystemEventArgs) eventArgs;
```

```
        if((args.getKind() != ENTRY_MODIFY) ||
(!args.getFileName().equals(ConfigBroker.FILE_NAME)))
            return;

        //如果被修改了，重新加载配置内容
        refresh();
    }

    …
}
```

4．实现认证服务逻辑

上面我们花费大量时间完成了一个可配置、可动态更新的执行基础，目的是确保整个认证服务的灵活性和可扩展性，对于一般的业务功能，这样代价比较大，也不一定值得，但对于一些基础的公共库或者公共服务却比较值得。另外，实际项目中即便对于公共库或者公共服务也不需每个接口都变成外部可配置的。可以考虑一个折中的办法，**就是在发布公共库、公共服务的时候，对可扩展的接口提供一个默认的实现类型**，类似前面 HandlerCoRBuilder 那样。

下面，我们开始编写认证服务主干逻辑，但在此之前为了隔离认证服务主体与具体认证服务提供者（Provider）的依赖关系，也为了隔离客户程序与认证服务实现类型的依赖关系，我们还要增加工厂类型。这样，从认证服务上层看，静态和动态结构分别如图 27-11~图 27-13 所示。

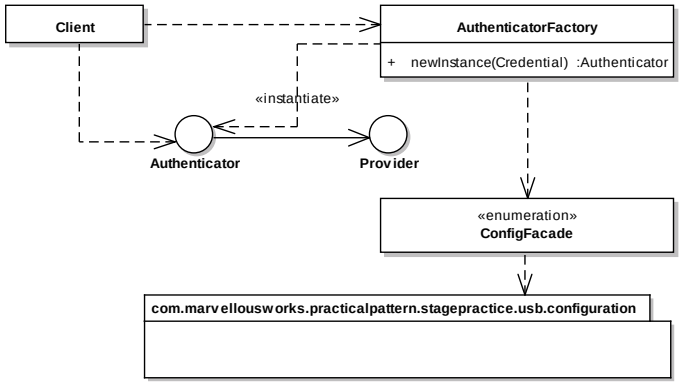

图 27-11 认证服务的静态结构

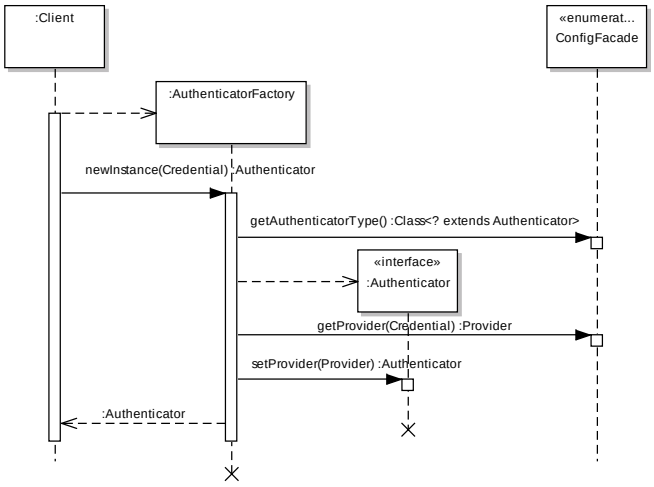

图 27-12 实例化认证服务实体类型的动态过程

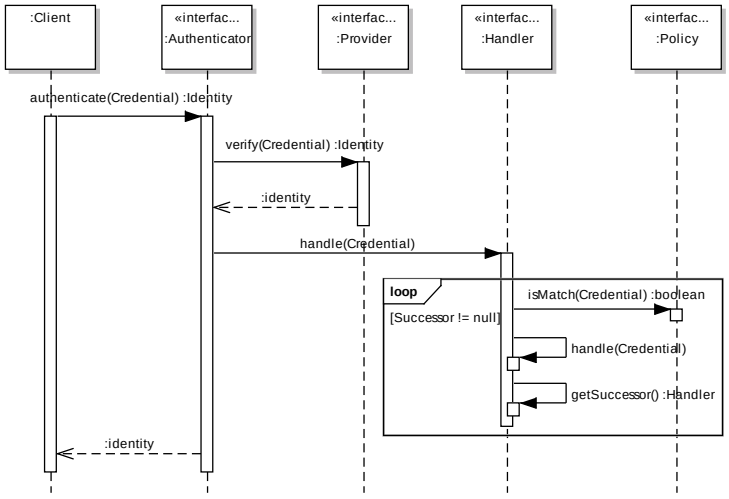

图 27-13 认证服务的认证过程

认证的实际代码如下：（已去除输出调试信息的部分）

Java 默认的认证子实现

```
public class AuthenticatorImpl implements Authenticator{

    private Provider provider;

    /**认证过程*/
    @Override
```

```
    public Identity authenticate(Credential credential) {
        //基本的验证过程
        Identity result = provider.verify(credential);
        //执行非功能性控制措施
        Handler handlerCoR;
        if((handlerCoR = ConfigFacade.INSTANCE.getHandlerCoR()) != null)
            handlerCoR.handle(credential);
        return result;
    }

    @Override
    public Provider getProvider() {return provider;  }

    @Override
    public Authenticator setProvider(Provider provider) {
        this.provider = provider;
        return this;
    }
}
```

1）编写测试类型

为了验证上述逻辑的有效性，下面还需要编写一系列测试类型并通过单元测试验证。

① 示例认证凭据类型

Java

```
/** 认证凭据*/
public interface Credential {
    /** 认证凭据颁布的对象名称 */
    String issueTo();
}


/** Usbkey 认证凭据*/
public class UsbCredential implements Credential{…}
/** 活动目录方式的认证凭据*/
public class WindowsCredential  implements Credential{…}
/** 用户名/口令方式的认证凭据*/
public class UserNameCredential implements Credential{…}


/** 新增的自定义认证凭据类型*/
public class CustomCredential implements Credential{…}
```

② 示例非功能性控制措施

Java 接口

```
/** 认证过程非功能性控制措施的接口 */
```

```
public interface Handler {
    /** 所适用的策略 */
    Policy getPolicy();
    Handler setPolicy(Policy policy);
    /** 处理*/
    void handle(Credential credential);

    /** 后继节点 */
    Handler getSuccessor();
    Handler setSuccessor(Handler successor);
}
```

Java 抽象基类

```
public class abstract AbstractHandler implements Handler{
    private Policy policy;
    private Handler successor;

    @Override
    public void handle(Credential credential) {
        notNull(credential);
        notNull(policy);
        if(policy.isMatch(credential))
            internalHandle(credential);

        //执行 CoR 后续的其他非功能性操作
        if(getSuccessor() != null)
            getSuccessor().handle(credential);
    }

    /** 实际需要处理的控制措施，交给子类实现 */
    public abstract void internalHandle(Credential credential);

    …
}
```

Java 具体类型

```
/** 输出当前的 Java Class 版本 */
public class LogClassVersionHandler extends AbstractHandler{
    @Override
    public void internalHandle(Credential credential) {
        trace("JAVA_CLASS_VERSION = %s", JAVA_CLASS_VERSION);
    }
}

/** 输出操作系统版本信息*/
public class LogOSHandler extends AbstractHandler{
    @Override
    public void internalHandle(Credential credential) {
        trace("OS = %s", OS);
    }
}
```

```
/** 输出 Java 运行环境提供商信息*/
public class LogVendorHandler extends AbstractHandler{
    @Override
    public void internalHandle(Credential credential) {
        trace("JAVA_VENDOR = %s", JAVA_VENDOR);
    }
}
```

③ 示例策略类型

Java 接口

```
/** 认证过程非功能性控制策略对象*/
public interface Policy {
    /** 策略是否适用 */
    boolean isMatch(Credential credential);
}
```

Java 抽象基类

```
public abstract class AbstractPolicy implements Policy{

    /**
     * 示例为了简便，仅通过 Credential 类型是否匹配作为 isMatch()
     * 判定条件，实际项目中 isMatch()可以根据很多其他因素定义，例如：
     *   > Credential 类型
     *   > Credential 的内容和各个成员
     *   > 当前的运行环境上下文
     *   >   …
     */
    @Override
    public boolean isMatch(Credential credential) {
        notNull(credential);
        if(applyTo().size() == 0)
            return false;
        else
            return applyTo().contains(credential.getClass());
    }

    /** 适用的认证凭证类型或类型列表*/
    public abstract List<Class<?>> applyTo();
}
```

Java 具体基类

```
/** 仅适于 USB Key 类型凭证 */
public class UsbPo extends AbstractPolicy{
    private static List<Class<?>> list = Arrays.asList(
        new Class<?>[]{UsbCredential.class});

    @Override
    public List<Class<?>> applyTo() {return list;}
```

```
}

/** 适于用户名口令和 USB Key 两种凭证类型*/
public class UserNameAndUsbPo extends AbstractPolicy{
    private static List<Class<?>> list = Arrays.asList(
        new Class<?>[]{UserNameCredential.class, UsbCredential.class});

    @Override
    public List<Class<?>> applyTo() {return list;}
}

/** 适于用户名口令凭证类型*/
public class UserNamePo extends AbstractPolicy{
    private static List<Class<?>> list = Arrays.asList(
        new Class<?>[]{UserNameCredential.class});

    @Override
    public List<Class<?>> applyTo() {return list;}
}

/** 适用于所有凭证类型 */
public class AllPo implements Policy{
    @Override
    public boolean isMatch(Credential credential) {
        notNull(credential);
        return true;
    }
}
```

④ 示例的第三方 USB Key 驱动和 USB Key 适配器

Java

```
package external;

/**模拟第三方提供的接口不兼容的驱动程序*/
public class UsbDriver {

    private boolean enabled;

    public boolean isEnabled(){return enabled;}
    public void enable(){    enabled = true;  }
    public void disable(){   enabled = false;}
}
```

```
package com.marvellousworks.practicalpattern.stagepractice.usb;

/**
 * USB Key 驱动适配接口
 * 为了隔绝第三方厂商的不确定性，认证服务先手定义的访问接口
 */
public interface UsbAdapter {
    /** 设备是否打开 */
    boolean isOpen();
```

```
    /** 打开 USB Key 设备端口 */
    void open();

    /**
     * 获取 USB 设备内凭证
     * @param pin 与驱动通信的 pin
     */
    Credential getCredential(String pin) throws InvalidKeyException;

    /** 关闭 USB Key 设备端口 */
    void close();
}
/** 一个 USB Key 驱动适配器的实现类*/
public class UsbAdapterImpl implements UsbAdapter{…}
```

⑤ 示例认证服务提供者类型

Java 接口

```
/** 针对具体凭证类型的认证功能的提供者 */
public interface Provider {
    /** 验证凭证信息，反馈认证结果*/
    Identity verify(Credential credential);

    /** 该 Provider 支持的认证凭据类型 */
    Class<? extends  Credential> getCredentialType();
}
```

Java 抽象基类

```
public abstract class AbstractProvider implements Provider{

    @Override
    public Identity verify(Credential credential) {
        notNull(credential);
        notNull(getCredentialType());
        if(credential.getClass() != getCredentialType())
            throw new
UnsupportedOperationException(credential.getClass().getName());
        return internalVerify(credential);
    }

    /** 供子类实现的实际认证过程 */
    public abstract Identity internalVerify(Credential credential);
}
```

Java 具体类型

```
public class UserNameProvider extends AbstractProvider{…}
public class WindowsProvider extends AbstractProvider{…}
public class UsbNameProvider extends AbstractProvider{…}
```

2）执行单元测试

单元测试的配置文件如下：（为了简化，已经去除完整的包路径）

marvellousworks.practicalpattern.xml

```
<?xml version="1.0" encoding="gb2312"?>
<!--全局配置节组-->
<practicalpatterns>

    <!--登记各模式对应配置节的解析类型-->
    <configSections>
        <add name="stagepractice"
            type="StagePracticeConfigSection"/>
    </configSections>

    <!--GOF 综合示例部分的配置节-->
    <stagepractice>

        <!--登记认证子的类型-->
        <authenticator value="AuthenticatorImpl"/>

        <!--Credential 配置-->
        <credentials>
            <add key="username" value="UserNameCredential"/>
            <add key="windows" value="WindowsCredential"/>
            <add key="usb" value="UsbCredential"/>
            <add key="custom" value="CustomCredential"/>
        </credentials>

        <!--Provider 配置-->
        <providers>
            <add key="username" value="UserNameProvider"/>
            <add key="windows" value="WindowsProvider"/>
            <add key="usb" value=" UsbProvider"/>
        </providers>

        <policies>
            <!--仅适用于 USB Key 方式-->
            <add key="usb" value=" UsbPo"/>
            <!--适用于 USB Key 和用户名口令方式-->
            <add key="userAndUsb" value=" UserNameAndUsbPo"/>
            <!--仅适用于用户名口令方式-->
            <add key="username" value=" UserNamePo"/>
            <!--全适用-->
            <add key="all" value=" AllPo"/>
        </policies>

        <handlers builder=" HandlerCoRBuilderImpl">
            <!--适于 usb Policy 规则-->
            <!--输出当前操作系统名称和版本-->
            <add key="logOS" seq="3" po="usb" value=" LogOSHandler"/>
```

```
            <!--适于 all Policy 规则-->
            <!--输出当前 Java 类库版本-->
            <add key="logJV" seq="1" po="all" value="
LogClassVersionHandler"/>
            <!--适于 userAndUsb Policy 规则-->
            <!--输出 Java 运行环境提供商信息-->
            <add key="logVendor" seq="2" po="userAndUsb" value="
LogVendorHandler"/>
        </handlers>

    </stagepractice>

</practicalpatterns>
```

① 验证读取配置信息

首先，验证读取配置信息。

Unit Test

```
StagePracticeConfigSection section;

@Before
public void setUp() throws XPathExpressionException, InstantiationException,
IllegalAccessException{
    section = (StagePracticeConfigSection)
ConfigBroker.getSection(StagePracticeConfigSection.NAME);
}

/** 验证根据名称获得 credential 类型 */
@Test
public void testGetCredentials(){
    assertEquals(UserNameCredential.class,
section.getCredentialType("username"));
    assertEquals(WindowsCredential.class,
section.getCredentialType("windows"));
    assertEquals(UsbCredential.class, section.getCredentialType("usb"));
    assertEquals(CustomCredential.class,
section.getCredentialType("custom"));
}

@Test(expected = NullPointerException.class)
public void testGetNonCredential(){
    section.getCredentialType(UNKNOWN);
}

/** 验证根据名称获得 Provider 类型 */
@Test
public void testGetProvider(){
    assertEquals(UserNameProvider.class,
section.getProviderType("username"));
    assertEquals(WindowsProvider.class,
section.getProviderType("windows"));
    assertEquals(UsbProvider.class, section.getProviderType("usb"));
}
```

```
@Test(expected = NullPointerException.class)
public void testGetNonProvider(){
    section.getProviderType(UNKNOWN);
}

/** 验证根据名称获得 Policy 类型 */
@Test
public void testGetPolicy(){
    assertEquals(UserNameAndUsbPo.class,
section.getPolicyType("userAndUsb"));
    assertEquals(UsbPo.class, section.getPolicyType("usb"));
    assertEquals(AllPo.class, section.getPolicyType("all"));
    assertEquals(UserNamePo.class, section.getPolicyType("username"));
}

@Test(expected = NullPointerException.class)
public void testGetNonPolicy(){
    section.getPolicyType(UNKNOWN);
}

/** 验证根据名称获得 Handler 类型 */
@Test
public void testGetHandler(){
    HandlerConfigElement element;

    element = section.getHandler("logOS");
    assertEquals(LogOSHandler.class, element.getValue());
    assertEquals(3, element.getSequence());
    assertEquals("usb", element.getPo());

    element = section.getHandler("logJV");
    assertEquals(LogClassVersionHandler.class, element.getValue());
    assertEquals(1, element.getSequence());
    assertEquals("all", element.getPo());

    element = section.getHandler("logVendor");
    assertEquals(LogVendorHandler.class, element.getValue());
    assertEquals(2, element.getSequence());
    assertEquals("userAndUsb", element.getPo());

    assertNull(section.getHandler(UNKNOWN));
}

@Test
public void testGetHandlers(){
    HandlerConfigElement[] elements = section.getHandlers();

    //验证各个 Handler 是按照次序排序好的
    if((elements == null) || (elements.length == 0))
        return;
    if(elements.length == 1)
        return;
    for(int i=0; i<elements.length -1 ; i++)
        assertTrue(elements[i].getSequence() <=
elements[i+1].getSequence());
}
```

```
/**
 * 验证获得 HandlerCoRBuilder 类型
 * 确认其构造 CoR 与配置文件定义的一致性
*/
@Test
public void testGetHandlerCoRBuilderType() throws InstantiationException,
IllegalAccessException{
    Class<? extends HandlerCoRBuilder> type = section.getBuilderType();
    assertEquals(type, HandlerCoRBuilderImpl.class);
    assertTrue(type.newInstance() instanceof HandlerCoRBuilder);
    // 验证 Handler CoR 中 Handler 的次序是否正确
    Class<?>[] handlerTypes = new Class<?>[]{
            LogClassVersionHandler.class,
            LogVendorHandler.class,
            LogOSHandler.class};
    Handler current = ((HandlerCoRBuilder)(type.newInstance())).buildUp();
    for(Class<?> handlerType : handlerTypes){
        assertEquals(handlerType, current.getClass());
        current = current.getSuccessor();
    }
    //确认正好遍历完 Handler CoR
    assertNull(current);

}

/** 验证获得 Authenticator 类型 */
@Test
public void testGetAuthenticatorType() throws InstantiationException,
IllegalAccessException{
    Class<? extends Authenticator> type = section.getAuthenticatorType();
    assertEquals(type, AuthenticatorImpl.class);
    assertTrue(type.newInstance() instanceof Authenticator);
}
```

② 验证动态加载配置信息

然后，验证 ConfigFacade 能否在配置文件修改的时候“动态”加载最新配置信息。

Unit Test

```
public class ConfigurationWatcherFixture {

    private static FileSystem f = FileSystems.getDefault();

    static void copy(String source, String target) throws IOException{
        Files.copy(f.getPath(source), f.getPath(target), REPLACE_EXISTING);
    }

    private File tempFile;

    @Before
    public void setUp() throws IOException, XPathExpressionException,
InstantiationException, IllegalAccessException{
        tempFile = File.createTempFile("test", ".txt");
```

```
    }

    @After
    public void tearDown() throws IOException{
        //删除临时文件
        Files.delete(tempFile.toPath());
    }

    /** 验证 ConfigFacade 能否根据配置文件的修改动态加载最新的配置 */
    @Test
    public void testWatchConfigFile()
        throws InterruptedException, IOException{
        //通过调用 ConfigFacade 触发其静态构造函数的执行过程
        //从后台启动 DirectoryWatcher 的监控过程
        ConfigFacade.INSTANCE.getAuthenticatorType();

        //将现有的配置文件备份为一个临时文件
        copy(ConfigBroker.FILE_NAME, tempFile.toString());

        //将临时文件复制，触发 DirectoryWatcher 的通知操作
        copy(tempFile.toString(), ConfigBroker.FILE_NAME);

        //延迟等待后台任务的执行
        Thread.sleep(2000);
        trace("finished");
    }
}
```

Console 窗口 （精简无关 trace 信息后的内容）

```
ConfigFacade.<clinit>()
ConfigFacade.refresh()
StagePracticeConfigSection.<init>()
ConfigFacade.update()
ConfigFacade.refresh()
StagePracticeConfigSection.<init>()
finished
```

上面的单元测试确认 **ConfigFacade 可以根据配置文件的变化动态加载最新的配置信息，**测试过程中通过 Thread.Sleep();方法对测试线程做了简短的停歇，确保 ConfigFacade 和 DirectoryWatcher 有机会对更新做出响应。

③ 验证认证服务示例的有效性

最后，作为整个示例集成测试内容，我们验证 Authenticator 接口及其默认实现类型的有效性。这里我们选择 4 类凭证进行验证：

USB Key。

用户名/口令。

活动目录 Windows 账号。

自定义的 CustomCredential，但没有支持该类型的 Provider，因此无法作为认证凭据使用。

而配置文件对于 3 类认证凭证的描述如下：

marvellousworks.practicalpattern.xml

```
<providers>
<add key="username" value=" UserNameProvider"/>
<add key="windows" value=" WindowsProvider"/>
<add key="usb" value=" UsbProvider"/>
</providers>

<policies>
<add key="usb" value=" UsbPo"/>
<add key="userAndUsb" value=" UserNameAndUsbPo"/>
<add key="username" value=" UserNamePo"/>
<add key="all" value=" AllPo"/>
</policies>

<handlers builder=" HandlerCoRBuilderImpl">
<add key="logOS" seq="3" po="usb" value=" LogOSHandler"/>
<add key="logJV" seq="1" po="all" value=" LogClassVersionHandler"/>
<add key="logVendor" seq="2" po="userAndUsb" value="LogVendorHandler"/>
</handlers>
```

按照配置内容，各类认证凭证的验证过程如下。

USB Key：适用于“usbPo“ “allPo”、“usbAndUserNamePo”3 个策略，因此会执行“logOS”、“logJV”、“logVendor”3 个非功能性措施的 Handler。另外，还会执行适配第三方 USB Key 驱动的过程。

用户名/口令：适用于“UserNameAndUsbPo “ 、“UserNamePo”和“AllPo”3 个策略，但只会执行 “logJV”、“logVendor”两个非功能性措施的 Handler。

活动目录 Windows 账号：仅适用于“AllPo“策略，因此只会执行“logJV”、一个非功能性措施的 Handler。

自定义的 CustomsCredential：尽管适于“AllPo”策略，但它没有对应的 Provider，因此不会被 Authenticator 调用，自然也不会触发非功能性措施的 Handler。

相应地，单元测试也验证了配置文件中的控制要求：

Java

```
public class AuthenticatorFixture {

    private static AuthenticatorFactory factory;

    @BeforeClass
    public static void classSetUp()
        throws InstantiationException, IllegalAccessException{
        factory = new AuthenticatorFactory();
```

```
        trace("认证服务启动过程");
        trace("Authenticator 类型是 %s",
    ConfigFacade.INSTANCE.getAuthenticatorType().getSimpleName());
        trace("\n\n");
    }

    @After
    public void cleanUp(){trace("\n\n");}

    @Test
    public void testUsbKeyAuthentication(){
        trace("验证 USB Key 凭证方式");
        test(new UsbCredential());
    }

    @Test
    public void testUserNameAuthentication(){
        trace("验证用户名/口令凭证方式");
        test(new UserNameCredential());
    }

    @Test
    public void testWindowsAuthentication(){
        trace("验证 Windows 活动目录凭证方式");
        test(new WindowsCredential());
    }

    /**因为没有定义相应的认证服务 Provider，因此会抛出异常*/
    @Test(expected = UnsupportedOperationException.class)
    public void testCustomAuthentication(){
        trace("验证新增的自定义凭证方式");
        test(new CustomCredential());
    }

    private void test(Credential credential){
        try {
            factory.newInstance(credential).authenticate(credential);
        } catch (InstantiationException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

Console 窗口

```
/**认证服务启动过程*/
ConfigFacade.<clinit>()
ConfigFacade.refresh()
StagePracticeConfigSection.<init>()
StagePracticeConfigSection.<init>()
HandlerCoRBuilderImpl.buildUp()
LogOSHandler.setPolicy(UsbPo)
LogVendorHandler.setPolicy(UserNameAndUsbPo)
LogClassVersionHandler.setPolicy(AllPo)
```

```
HandlerCoR=LogClassVersionHandler->LogVendorHandler->LogOSHandler->NULL
Authenticator 类型是 AuthenticatorImpl
/**验证 USB Key 凭证方式*/
AuthenticatorFactory.newInstance(UsbCredential)
ConfigFacade.getProvider(UsbCredential)
AuthenticatorImpl.setProvider(UsbProvider)
AuthenticatorImpl.authenticate(UsbCredential)
/**执行基本的验证过程*/
UsbProvider.verify(UsbCredential)
UsbDriver.isEnabled() = false
UsbAdapter.isOpen() = false
UsbDriver.isEnabled() = false
UsbAdapterImpl.open()
UsbDriver.enable()
/**读取 USB Key 内认证凭证*/
UsbAdapterImpl.getCredential()
/**成功读取 USB Key 内认证凭据*/
UsbAdapterImpl.close()
UsbDriver.disable()
/**执行非功能性控制措施*/
ConfigFacade.getHandlerCoR()
LogClassVersionHandler.handle(UsbCredential)
UsbCredential 适用策略 AllPo
JAVA_CLASS_VERSION = 51.0
LogVendorHandler.handle(UsbCredential)
UsbCredential 适用策略 UserNameAndUsbPo
JAVA_VENDOR = Oracle Corporation
LogOSHandler.handle(UsbCredential)
UsbCredential 适用策略 UsbPo
OS = Windows 7 6.1
/**验证用户名/口令凭证方式*/
AuthenticatorFactory.newInstance(UserNameCredential)
ConfigFacade.getProvider(UserNameCredential)
AuthenticatorImpl.setProvider(UserNameProvider)
AuthenticatorImpl.authenticate(UserNameCredential)
/**执行基本的验证过程*/
UserNameProvider.verify(UserNameCredential)
/**执行非功能性控制措施*/
ConfigFacade.getHandlerCoR()
LogClassVersionHandler.handle(UserNameCredential)
UserNameCredential 适用策略 AllPo
JAVA_CLASS_VERSION = 51.0
LogVendorHandler.handle(UserNameCredential)
UserNameCredential 适用策略 UserNameAndUsbPo
JAVA_VENDOR = Oracle Corporation
LogOSHandler.handle(UserNameCredential)
/**验证 Windows 活动目录凭证方式*/
AuthenticatorFactory.newInstance(WindowsCredential)
ConfigFacade.getProvider(WindowsCredential)
AuthenticatorImpl.setProvider(WindowsProvider)
AuthenticatorImpl.authenticate(WindowsCredential)
```

```
/**执行基本的验证过程*/
WindowsProvider.verify(WindowsCredential)
/**执行非功能性控制措施*/
ConfigFacade.getHandlerCoR()
LogClassVersionHandler.handle(WindowsCredential)
WindowsCredential 适用策略  AllPo
JAVA_CLASS_VERSION = 51.0
LogVendorHandler.handle(WindowsCredential)
LogOSHandler.handle(WindowsCredential)
/**验证新增的自定义凭证方式*/
AuthenticatorFactory.newInstance(CustomCredential)
ConfigFacade.getProvider(CustomCredential)
```

④ 测试结论

综上所示，通过一系列单元测试我们从各个方面验证了认证服务现有结构的有效性、可扩展性和动态性。

5．小结

本章“中规中矩”地通过一系列模式设计了一个“准生产型”的认证服务系统。作为本册 GOF 部分的最后一章，希望这个练习能帮助读者加深对 Java SE 环境下模式应用技巧的体会。上面我们综合运用了如下模式，每个模式的功能如下。

桥模式：规划认证服务局部架构。

简单工厂模式：构造抽象接口。

适配器模式：适配第三方厂商提供的 USB Key。

策略模式：确认各个非功能性控制措施的适用性。

职责链模式：组织不确定数量的非功能性控制措施。

创建者模式：构造职责链。

外观模式：隔离复杂的配置系统。

观察者模式：“盯住”配置信息的变化，随时更新缓存的配置信息。

代理模式：为远程访问做准备。

单件模式：确保配置访问外观类型只有一个实例和一个公共访问点。

下面，我们再次对本章前面的设计思路做个归纳：包括设计和实现两个部分。

设计部分：

（1）建模主干流程。

（2）根据需求寻找主干流程中的不稳定节点，抽象接口。

（3）应用模式，解耦构造过程、类型结构、执行时序中不稳定的依赖关系。

（4）增加外观类型封装那些难以通过其他 GOF 结构型模式整理的部分。

实现部分：

首先，遵循以配置为中心的实现过程：

（1）逐个排查 “不稳定节点”的接口，根据“不稳定”程度及项目上线后开发、运维人员的分工，确认是否需要抽象为配置节。

（2）确认配置内容是单个配置元素还是配置元素集合。

（3）梳理配置对象的开发范围，明确哪些可以通过 JDK 自带的 XML 类型完成，哪些需要定制开发。

（4）定义配置文件 Schema。

（5）优化配置文件 Schema。

其次，完成应用逻辑部分：

（1）完成应用主干逻辑。

（2）逐步装配各类“附件”逻辑。

最后，正确性是第一位的，**作为功能部分的开发人员还应该逐个完成主要类型、模块的单元测试，需要从客户程序的角度验证服务接口的整体集成的有效性。**

6．后记

相对而言，从经典教科书式的书籍入手可能会离平台远一些，对于平台优势的体会要弱一些，入手也会慢点，这也是本系列一直坚持“工程化实现及扩展”的主要原因。

为了体现本书的特色，本章我们对 GOF 部分做了一个综合回顾，相信通过这个示例读者已经对 GOF 的综合运用有了更为深入的体会。但这些操作基本都是在本地内存中执行的，而我们实际面临的项目大部分都是分布式应用，本系列的下册《模式——工程化实现及扩展（架构模式 Java 版）》中将介绍分布式环境下一系列更高层的模式技巧。但是，这些架构模式大部分也是在 GOF 的基础上进行组合，出于性能、通信、并发、安全性的考虑进一步扩展的模式，学好、用好 GOF 是进一步了解架构模式的台阶。

另外，建议读者了解本系列 C#分册的内容，毕竟随着项目规模的膨胀，很多项目往往需要综合使用多种开发语言和开发平台。包括.NET 和 Java EE 在内的这些平台可以说各有优势，体会这些内容除了从学习编程语言和开发框架入手外，从设计模式、架构模式角度入手也是个非常不错的选择。博采众长而不是受厂商的影响将自己定死在一个平台，这能帮助我们更快地成长。视野决定高度，做开发一样如此。