



模式

王翔 孙逊 著

工程化实现及扩展

(设计模式C#版)



电子工业出版社

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.NET Framework 有深入了解，因为扩展主要是结合 C#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 C#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 C#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”把各种设计模式进行了一次集中展示，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。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

模式——工程化实现及扩展：设计模式 C#版 / 王翔，孙逊著. —北京：电子工业出版社，2012.4

ISBN 978-7-121-15639-7

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

中国版本图书馆 CIP 数据核字（2011）第 280710 号

策划编辑：张春雨

责任编辑：付 睿

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：25.5 字数：653 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 zltz@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个性。

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，但效果不一定好，因为

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

相信读者也发现了，用一个语言写 **Hello World** 是一回事，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很多实现方式与《设计模式》介绍的内容不一致，但别忘了《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，如果我们“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 C# 设计模式分册，我试图用最 C# 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，本书分为以下 5 个部分。

- 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、C# 语言面向对象扩展特性，以及 Java 和 C# 语法特性的简单对比。

- 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。

- 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。

- 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、封装为独立对象的办法，削弱调用过程中的耦合关系。

- 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 and 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 .NET Framework 平台特性加以实现的过程。

不管读者之前对于模式是否有所尝试，我希望读者不妨浏览这章，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，您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，你给予我直面挑战、战胜挑战的信心和力量。

不过，受到开发年限和项目经验的限制，本书在很多地方难免会有疏漏和不足之处，希望能够听到读者的批评和建议。

王翔

目 录

第一篇 预备知识——发掘 C#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） 7
- 1.4 接口隔离原则（ISP） 8
- 1.5 迪米特法则（LoD: Law of Demeter、LKP） 9
- 1.6 开闭原则（OCP） 11
- 1.7 小结 15
- 1.8 自我检验 15

第 2 章 重新研读 C#语言 16

- 2.1 说明 17
- 2.2 C# 部分语法内容扩展 18
 - 2.2.1 命名空间（Namespace） 18
 - 2.2.2 简洁的异步调用机制——委托（Delegate）和事件（Event） 21
 - 2.2.3 考验算法的抽象能力——泛型（Generics） 29
 - 2.2.4 用贴“标签”的方式扩展对象特性 35
 - 2.2.5 可重载运算符（Overloadable Operators）与
转换运算符（Conversion Operators） 38
- 2.3 面向插件架构的配置系统设计 41
 - 2.3.1 认识.NET Framework 提供的主要配置实体类 42
 - 2.3.2 小结 43
 - 2.3.3 自我检验 43
- 2.4 依赖注入 44
 - 2.4.1 背景介绍 44

2.4.2 示例情景.....	45
2.4.3 构造注入 (Constructor)	47
2.4.4 设值注入 (Setter)	48
2.4.5 接口注入.....	48
2.4.6 基于属性的注入方式 (Attributer)	50
2.4.7 小结.....	52
2.4.8 自我检验.....	53
2.5 连贯接口 (Fluent Interface)	53
第3章 Java 和 C#.....	56
3.1 说明	57
3.2 枚举 (Enum)	57
3.3 泛型 (Generics)	61
3.4 属性和标注 (Attribute and Annotation)	63
3.5 操作符重载和类型转换重载.....	64
3.6 委托、事件、匿名方法.....	65
3.7 Lamada 和 LINQ	68
3.8 小结	73
第二篇 创建型模式——管理对象实例的构造过程	74
第4章 工厂&工厂方法模式.....	75
4.1 说明	76
4.2 简单工厂	76
4.2.1 最简单的工厂类.....	76
4.2.2 简单工厂的局限性.....	79
4.3 经典回顾	79
4.4 解耦工厂类型与客户程序.....	81
4.5 基于配置文件的工厂	83
4.5.1 基于配置文件解耦工厂接口和具体工厂类型	83
4.5.2 基于配置文件解耦工厂类型和具体工作产品	84
4.6 典型工程化实现.....	86
4.7 小结	89
4.8 自我检验	90
第5章 单件模式	91
5.1 说明	92

5.2	经典回顾	93
5.3	线程安全的单件模式	96
5.4	细节决定成败	98
5.5	细颗粒度的单件模式	99
5.5.1	背景讨论	99
5.5.2	解决桌面应用中细颗粒度单件模式问题	100
5.5.3	解决 Web 应用中细颗粒度单件模式问题	101
5.6	分布式环境下的单件模式	102
5.7	单件模式的使用问题	105
5.8	小结	105
第 6 章	抽象工厂模式	106
6.1	说明	107
6.2	经典回顾	108
6.3	解决经典模式的硬伤	110
6.4	基于委托的生产外包	112
6.5	小结	114
第 7 章	创建者模式	116
7.1	说明	117
7.2	经典回顾	118
7.3	为 Builder 打个标签	121
7.4	具有装配/卸载能力的 Builder	126
7.5	连贯接口形式的 Builder	126
7.6	小结	130
7.7	自我检验	130
第 8 章	原型模式	131
8.1	说明	132
8.2	经典回顾	132
8.3	表面模仿还是深入模仿	136
8.3.1	概念	136
8.3.2	制作实现克隆的工具类型	137
8.3.3	简单自定义复制过程	139
8.3.4	细颗粒度自定义复制过程	140
8.4	小结	142
8.5	自我检验	143

第三篇 结构型模式——组织灵活的对象体系	145
第 9 章 适配器模式	146
9.1 说明	147
9.2 经典回顾	148
9.3 类型转换符实现适配	151
9.4 组适配器	152
9.5 用配置约定适配过程	154
9.6 面向数据的适配机制	156
9.7 小结	158
9.8 自我检验	158
第 10 章 桥模式	161
10.1 说明	162
10.2 经典回顾	163
10.3 分解复杂性的多级桥关系	166
10.4 看着“图纸”造桥	171
10.5 具有约束关系的桥	174
10.6 小结	175
10.7 自我检验	176
第 11 章 组合模式	177
11.1 说明	178
11.2 经典回顾	179
11.3 用迭代器遍历组合类型	183
11.4 适于 XML 信息的组合模式	185
11.5 分布式“部分—整体”环境	188
11.6 小结	188
11.7 自我检验	189
第 12 章 装饰模式	190
12.1 说明	191
12.2 经典回顾	192
12.3 卸载装饰	195
12.4 通过配置和创建者完成装饰过程	196

12.5 把装饰类型做成标签.....	198
12.5.1 更“彻底”的属性注入.....	198
12.5.2 方式 1: 采用.NET 平台自带的 AOP 机制实现.....	199
12.5.3 自定义代理拦截框架方式.....	202
12.5.4 进一步分析.....	205
12.6 小结.....	205
第 13 章 外观模式.....	207
13.1 说明.....	208
13.2 经典回顾.....	208
13.3 平台、开发语言无关的抽象 Facade 接口——WSDL.....	211
13.4 小结.....	212
第 14 章 享元模式.....	214
14.1 说明.....	215
14.2 经典回顾.....	216
14.3 制订共享计划.....	219
14.4 通过委托和队列实现异步享元.....	219
14.5 小结.....	220
第 15 章 代理模式.....	221
15.1 说明.....	222
15.2 经典回顾.....	222
15.3 远程代理.....	224
15.4 小结.....	226
第四篇 行为型模式——算法、控制流的对象化操作.....	227
第 16 章 职责链模式.....	228
16.1 说明.....	229
16.2 经典回顾.....	229
16.3 非链表方式定义职责链.....	233
16.4 小结.....	237
第 17 章 模板方法模式.....	239
17.1 说明.....	240
17.2 经典回顾.....	241

17.3 方法的模板——委托.....	243
17.4 类和接口的模板——泛型.....	243
17.5 系统架构的模板——配置.....	244
17.6 小结	245
17.7 自我检验	245
第 18 章 解释器模式	247
18.1 说明	248
18.2 经典回顾	249
18.3 采用正则表达式.....	252
18.4 采用字典	255
18.5 多级解释器系统.....	258
18.6 用 XSD 解释自定义业务语言	260
18.7 小结	261
18.8 自我检验	261
第 19 章 命令模式	262
19.1 说明	263
19.2 经典回顾	264
19.3 打包命令对象	266
19.4 异步命令操作	270
19.5 命令操作队列	272
19.6 小结	273
19.7 自我检验	273
第 20 章 迭代器模式	276
20.1 说明	277
20.2 经典回顾	278
20.3 .NET 内置的迭代器.....	279
20.4 小结	282
20.5 自我检验	282
第 21 章 中介者模式	283
21.1 说明	284
21.2 经典回顾	285
21.3 基于委托和事件的松耦合中介者	288
21.4 根据配置动态协调通知关系.....	290

21.5	小结	292
21.6	自我检验	293
第 22 章	备忘录模式	294
22.1	说明	295
22.2	经典回顾	296
22.3	把备忘压栈	300
22.4	备忘录的序列化和持久化	302
22.5	小结	305
22.6	自我检验	305
第 23 章	观察者模式	307
23.1	说明	308
23.2	经典回顾	310
23.3	.NET 内置的观察者机制——事件	313
23.4	具有观察者机制的集合类型	315
23.5	面向服务接口的观察者	316
23.6	小结	318
23.7	自我检验	318
第 24 章	状态模式	320
24.1	说明	321
24.2	经典回顾	322
24.3	状态的序列化和持久化	326
24.4	主动状态对象	328
24.5	用 WF 完成更易于编排的状态模式	329
24.6	小结	330
24.7	自我检验	330
第 25 章	策略模式	332
25.1	说明	333
25.2	经典回顾	334
25.3	策略模式与解释器模式的协作	335
25.4	.NET 自带的策略接口	337
25.5	小结	338
第 26 章	访问者模式	339

26.1	说明	340
26.2	经典回顾	340
26.3	借助反射或 Dynamic 实现访问者	343
26.4	用委托实现工程化的访问者	346
26.5	小结	347
26.6	自我检验	347
第五篇 GOF 综合练习		349
第 27 章 GOF 部分阶段实践		350
27.1	回顾 GOF	351
27.2	需求的提出	351
27.3	第一轮技术分析	352
27.4	第二轮技术分析	354
27.5	第三轮技术分析	356
27.6	示例实现	357
27.7	验证逻辑的有效性	369
27.8	小结	380
27.9	后记	381
附录 Java 和 C#关键字对照表		382

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 把装饰类型做成标签
- 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型体系快速膨胀的情况。

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

以手机为例：

之所以称它为手机，首要原因是可以在移动中拨打电话和收发短信息，按照单一职责原则，我们姑且定义具有这两个功能的一类对象是 **IMobile**。

后来手机可以听 MP3，这时的手机就成了 **MobileWithMP3**：IMobile、IMP3。

接着，出现了智能手机，除了之前的功能外，它还可以导航，于是就成了 **MobileWithMP3AndGPS**：Imobile、IMP3、IGPS。

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 C# 编码实现过程中由于受到单继承的约束，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，比如新增 3 个接口的实现，就需要 8 个类型（包括 **MobileBase**），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，而装饰模式解决的就是这类问题。



什么是 has a 实例？比如那个 **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 `IMP3 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”一章提到的里氏替换原则（LSP）。

由于 C# 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，比如留给项目自己

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。

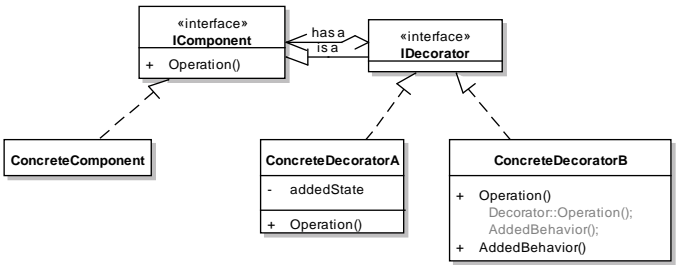


图 12-1 经典装饰模式的静态结构

根据上面的 Notepad 的示例要求，我们的设计如图 12-2 所示。

C# 抽象部分

```

public interface IText
{
    string Content { get; }
}

public interface IDecorator : IText { }
public abstract class DecoratorBase : IDecorator    // is a
{
    /// has a
    protected IText target;

    public DecoratorBase(IText target) { this.target = target; }
    public abstract string Content { get; }
}

```

C# 具体装饰类型

```

/// 具体装饰类
/// 属于“马甲”
public class BoldDecorator : DecoratorBase
{
    public BoldDecorator(IText target) : base(target) { }

    public override string Content
    {
        get { return ChangeToBoldFont(target.Content); }
    }

    public string ChangeToBoldFont(string content)
    {
        return "<b>" + content + "</b>";
    }
}

/// 具体装饰类
/// 属于“马甲”
public class ColorDecorator : DecoratorBase
{
    public ColorDecorator(IText target) : base(target) { }
    public override string Content
    {
        get { return AddColorTag(target.Content); }
    }

    public string AddColorTag(string content)
    {
        return "<color>" + target.Content + "</color>";
    }
}

/// 具体装饰类
/// 属于“口罩”
public class BlockAllDecorator : DecoratorBase

```

```
{
    public BlockAllDecorator(IText target) : base(target) { }
    public override string Content { get { return string.Empty; }}
}
```

/// 实体对象类型

```
public class TextObject : IText
{
    public string Content { get { return "hello"; } }
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    //建立对象，并对其进行两次装饰
    IText text = new TextObject();
    text = new BoldDecorator(text);
    text = new ColorDecorator(text);
    Assert.AreEqual<string>("<color><b>hello</b></color>", text.Content);

    //建立对象，只对其进行一次装饰
    text = new TextObject();
    text = new ColorDecorator(text);
    Assert.AreEqual<string>("<color>hello</color>", text.Content);

    //通过装饰，撤销某些操作
    text = new BlockAllDecorator(text);
    Assert.IsTrue(string.IsNullOrEmpty(text.Content));
}
```

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 `IComponent` 定义的方法，但同时又会保留一个对 `IComponent` 的引用。`IComponent` 接口方法的实现其实是通过自己保存的 `IComponent` 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：

`IComponent` 不要直接或间接地使用 `IDecorator`，因为它不应该知道 `IDecorator` 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。

`IDecorator` 也仅仅认识 `IComponent`。抽象依赖于抽象，知识最少。

某个 `ConcreteDecorator` 最好也不知道 `ConcreteComponent` 的存在，否则概念上该 `ConcreteDecorator` 只能服务于这个 `ConcreteComponent`（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。

运行态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，IComponent 提交给客户程序的是最外层 Concrete Decorator 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。

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.NET 程序集不是太难。

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，就像家里装修一样，不能说墙纸贴上去就再也不揭了。这一点对于那些需要长时间驻留内存、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 MSIL 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，我们可以用些技巧，比如，在 IDecorator 定义时增加一个类似名为 Enabled 的属性方法即可。

12.4 通过配置和创建者完成装饰过程

一般我们实现装饰模式都要使用前面那个“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。不过我们分析一下，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
IText text = new TextObject();
text = new BoldDecorator(text);
text = new ColorDecorator(text);
...
```

大家可以看到，采用这种所谓的经典装饰过程，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。而且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，这种做法并不可取。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再“痛定思痛”的情景。我们在之前的示例中已经“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些改变。

下面我们来看一个示例。

C# 登记客户类型与相应装饰类型配置关系的 Assembly 类型

```
///装饰类型的装配器
public class DecoratorAssembler
{
    ///登记装饰不同类型需要使用的一组 Concrete Decorator 类型
    static IDictionary<Type, IEnumerable<Type>> dictionary =
        new Dictionary<Type, IEnumerable<Type>>();

    ///实际项目中这个加载过程可由配置完成
    static DecoratorAssembler()
    {
        #region 配置相关的装饰类型
        var types = newList<Type>()
        {
            typeof (BoldDecorator),
            typeof (ColorDecorator)
        };
        dictionary.Add(typeof (TextObject), types);
        #endregion
    }

    ///按照需要构造的客户类型选择相应的 Decorator 列表
    public IEnumerable<Type> this[Type type]
    {
        get
        {
            if (type == null) throw new ArgumentNullException("type");
            IEnumerable<Type> result;
            return dictionary.TryGetValue(type, out result) ? result : null;
        }
    }
}
```

C# 完成装饰过程的 Builder 类型

```
///为目标类型做“装饰”的创建者
public class DecoratorBuilder
```

```

{
    static readonly DecoratorAssembler assembly = new DecoratorAssembler();

    /// 构造
    public IText BuildUp(IText target)
    {
        if (target == null) throw new ArgumentNullException("target");
        var types = assembly[target.GetType()];
        if ((types != null) && (types.Count() > 0))
            types.ToList().ForEach(x => target = (IText)Activator.CreateInstance(x,
            target));
        return target;
    }
}

```

Unit Test

```

[TestMethod]
public void Test()
{
    // 修改后的 IText 仅仅依赖于一个 Builder 类型
    IText text = new TextObject();
    text = (new DecoratorBuilder()).BuildUp(text);
    Assert.AreEqual<string>("<color><b>hello</b></color>", text.Content);
}

```

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了我们从外部配置目标类型及其相关装饰类型的机会。

12.5 把装饰类型做成标签

尽管通过前面 has-a + is-a 的方式可以达到装饰效果，但这种“八股”的实现方式对于用户而言还是烦琐，用户需要增加的功能都需要额外编写装饰类型。

下面我们将采用对于客户程序而言更简便的“贴标签”方式完成装饰过程。

不过为了避免歧义，这里先做个辨析：尽管在“重新研读 C#语言”一章我们已经完成一个基于属性的装饰示例，但这还是“一事一议”的处理方法，因此所有操作都是针对 DirectorAttribute 属性的，本节我们要尝试用更通用的方式解决这个问题。当然客户程序用着简单一般都意味着支持框架的开发要增加很多工作量，正所谓“台上一分钟，台下十年功”，尽管本节只是用 .NET 自带的横切机制通过属性对类型扩展，这不需要“十年功”，但能够熟练调用 C# 动态生成 MSIL 也要几年功底。



本节内容相对复杂，一般也仅在一些超大型项目的核心环节零星使用，建议先了解 MSIL 及 .NET 动态编译的相关内容后再阅读本节。

为了降低本节的难度，本次改版没有直接编码动态生成、编译 MSIL 的拦截类型，而是通过集成微软 Unity Block 完成该工作。但为了了解其拦截机制的实现，还是建议读者尝试阅读该类库的源码。

12.5.1 更“彻底”的属性注入

在前面的章节里，我们其实已经多次使用属性作为类型的装饰性描述，不过前面章节采用的都是通过定制面向专门接口的创建型模式完成的，这种实现方式本身并不通用，毕竟随着应用集成、项目中共享组件重用等方法的出现，很难做到系统中所有的对象都继承自固定的一个甚至几个接口，我们不应该把对这些装饰属性的解析和控制过程完全基于接口方法，而对同样适用于这个方式的其他方法无可奈何。一个更为普遍的办法就是采用“拦截（Interception）”机制，借助透明代理定制方法上标记了装饰属性的调用过程。

概念上这个透明代理类的主要作用就是把客户程序对目标类型的直接调用拆成两步甚至多步调用，而对客户程序而言，它感觉不到这个调用过程被重发了，这也是区别于前面章节示例的地方。它除了向客户程序交付它需要的接口外，还需要提供一个操作该对象实例的支持对象。新的调用关系如图 12-3 所示。

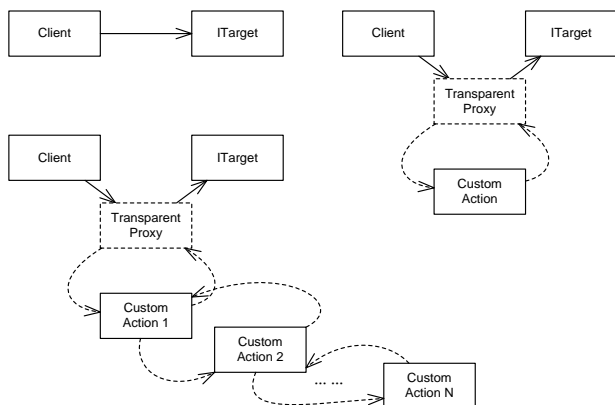


图 12-3 基于属性的动态代理装饰过程

这里相对经典设计模式而言存在一个灰色地带，即装饰模式和后面要提到的代理模式的界定问题。《设计模式》一书中装饰模式采用的是包含方式，而代理模式采用的是继承（或实现公共接口）方式。这里把装饰性设计为属性的主要目的是根

据 C#语言的特点, 提供一个耦合度更低、后续开发人员使用更方便的途径, 而所使用的动态代理则仅仅是一个后台支撑机制, 是实现这种途径的手段。



实际执行步骤其实更复杂, 客户程序调用 Transparent Proxy , 而 Transparent Proxy 会调用 Real Proxy , Real Proxy 后续还需要完成一系列 Message Sink 操作。

至于如何实现动态代理, 我们可以采用很多方式, 本章选择其中两个技术实现难度较大, 但对客户程序而言更易用的方式。

方式 1: 继承 MarshalByRefObject 或 ContextBoundObject。

方式 2: 动态编译。

12.5.2 方式 1 : 采用 .NET 平台自带的 AOP 机制实现

CLR 本身对外提供了对 AOP 机制的支持, 但用它完成装饰属性有一个非常不利的限制——客户类型必须继承自 MarshalByRefObject 或 ContextBoundObject, 对于待装饰的目标类型而言, 等于放弃了唯一一次的继承机会。

下面是一个示例, 其代码如下:

C# 定义装饰属性的基类

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Property)]
abstract class DecoratorAttributeBase : Attribute
{
    public abstract void Intercept(object target);
}
```

C# 定义代理类

```
class CustomProxy<T> : RealProxy, IDisposable where T : MarshalByRefObject
{
    ///构造过程中把 Proxy 需要操作的内容与实际目标对象实例 Attach 到一起
    public CustomProxy(T target) : base(target.GetType())
    {
        AttachServer(target);
    }

    ///析构过程则借助 Proxy 和目标对象实例的 Attach , 便于 GC 回收
    public void Dispose() { DetachServer(); }

    public static T Create(T target)
    {
        if (target == null) throw new ArgumentNullException("target");
        return (T)(new CustomProxy<T>(target).GetTransparentProxy());
    }

    ///实际执行的拦截, 并根据装饰属性进行定制处理
    public override IMessage Invoke(IMessage msg)
    {
```

```

var caller =
    new MethodCallMessageWrapper((IMethodCallMessage)msg);

    //提取实际宿主对象
var method = (MethodInfo)caller.MethodBase;
    T target = (T)GetUnwrappedServer();
var attributes = (DecoratorAttributeBase[])
    method.GetCustomAttributes(typeof(DecoratorAttributeBase),
        true);
    if (attributes.Length > 0)
        foreach (DecoratorAttributeBase attribute in attributes)
            attribute.Intercept(caller);
var ret = method.Invoke(target, caller.Args);

    //拦截处理后，继续回到宿主对象的调用过程
return new ReturnMessage(ret, caller.Args, caller.ArgCount,
    caller.LogicalCallContext, caller);
}
}

```

C# 定义具体装饰属性

```

[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
class ArgumentTypeRestrictionAttribute : DecoratorAttributeBase
{
    Type type;
    public ArgumentTypeRestrictionAttribute(Type type) { this.type = type; }
    public override void Intercept(object target)
    {
        var caller = (MethodCallMessageWrapper)target;
        if (caller.ArgCount == 0) return;
        for (var i = 0; i < caller.ArgCount; i++)
        {
            var arg = caller.Args[i];
            if ((arg.GetType() != type) &&
                (!arg.GetType().IsAssignableFrom(type)))
                throw new ArgumentException(i.ToString());
        }
    }
}

[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
class ArgumentNotEmptyAttribute : DecoratorAttributeBase
{
    public override void Intercept(object target)
    {
        var caller = (MethodCallMessageWrapper)target;
        if (caller.ArgCount == 0) return;
        foreach (var arg in caller.Args)
            if (string.IsNullOrEmpty((string)arg))
                throw new ArgumentException();
    }
}

```

C# 定义业务对象

```

class User : MarshalByRefObject
{
    string name;
}

```



```

        string title;

        [ArgumentTypeRestriction(typeof(string))]    //提供拦截入口
        [ArgumentNotEmpty()]                        //提供拦截入口
        public void SetUserInfo(object name, object title)
        {
            this.name = (string)name;
            this.title = (string)title;
        }
    }
}

```

Unit Test

```

[TestMethod]
public void Test()
{
    User user = CustomProxy<User>.Create(new User());
    user.SetUserInfo("joe", "manager"); //成功

    try
    {
        user.SetUserInfo(20, "manager");
    }
    catch (Exception exception)
    {
        // 因第一个参数类型异常被拦截后抛出异常
        Assert.IsInstanceOfType(exception, typeof(ArgumentException));
    }

    try
    {
        user.SetUserInfo("", "manager");
    }
    catch (Exception exception)
    {
        // 因 name 为空被拦截后抛出异常
        Assert.IsInstanceOfType(exception, typeof(ArgumentException));
    }
}

```

从上面的示例中不难看出，通过.NET Remoting 和 MarshalByRefObject 的组合，借助代理类可以拦截调用过程，并“横切”楔入额外的控制逻辑。但遗憾的是这种方式把 C# 类型唯一一次继承机会让给 MarshalByRefObject，侵入性过高。

尽管有很多不足，不过在项目中采用该方法实现一个透明的装饰属性框架，成本相对较低。另外，上面的示例中并没有提供对 params 参数、属性方法和带参数构造函数的支持，更为复杂的工程化实现可以参考微软的 Policy Injection 和 Unity 代码块。

12.5.3 自定义代理拦截框架方式

如果希望在.NET 平台实现一个侵入性低的框架，则需要通过一些异常复杂的、非

常“动态”的手段实现装饰对象的包装。概念上.NET Framework CLR 实际执行的是底层 MSIL，因此部分主流框架通过 System.Reflection.Emit 命名空间下的对象，实现动态装配和动态编译。采用该方式的主要意义是根据用户提供的类型实例，**使用 MSIL 动态装配出一个新的类型，然后对这个类型进行编译**。而新的类型在执行具体方法、属性方法、委托和事件的同时，调用动态装配的外界方法（或方法列表），如图 12-4 所示。

该方式的执行步骤如下：

（1）在执行前，框架通过 System.Reflection.Emit 根据目标类型生成新的动态类型。生成动态类型的过程包括创建构造函数、创建方法、属性方法和事件，并且为外部“横切”机制提供入口。

（2）执行过程中，客户程序实际调用的是动态生成的新类型。

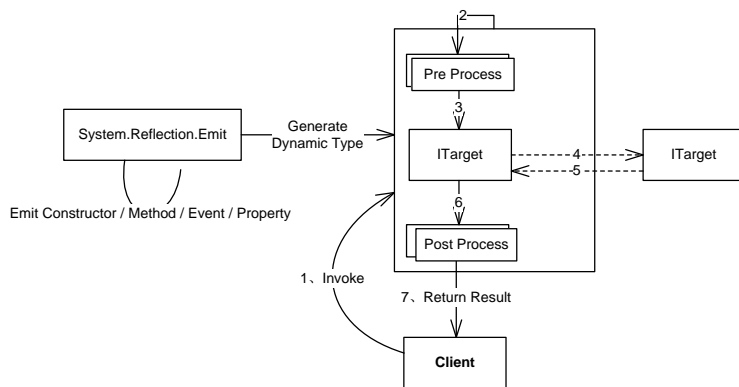


图 12-4 动态包装类型的执行过程

下面我们以微软的Unity Block（Unity Block可以从 <http://unity.codeplex.com/> 获得）为例，看一个通过动态编译MSIL获得新类型的使用效果示例，这里 InterfaceInterceptor就是一个借助动态生成MSIL获得目标类型的拦截器。

C# 使用自定义的 AOP Decorator Attribute 装饰目标类型

```

///待装饰的目标类型接口
public interface IBizObject
{
    ///待装饰的内容只要定义在接口即可，不用逐个定义在每个实体类上
    [Log]
    [Security]
    int GetValue();

    int GetYear();
}

///待装饰的目标类型
public class BizObject : IBizObject

```

```

{
    public static readonly int Value = new Random().Next();

    /// 接口中已经被标注为需要装饰的方法
    public int GetValue() { return Value; }

    /// 接口中声明为无须装饰的方法
    public int GetYear()
    {
        return DateTime.Now.Year;
    }
}

#region 自定义的装饰属性

public class LogAttribute : HandlerAttribute, ICallHandler
{
    public override ICallHandler CreateHandler(IUnityContainer container)
    {
        return this;
    }

    public IMethodReturn Invoke(IMethodInvocation input,
        GetNextHandlerDelegate getNext)
    {
        {
            Trace.WriteLine("log...");
            return getNext()(input, getNext);
        }
    }
}

public class SecurityAttribute : HandlerAttribute, ICallHandler
{
    public override ICallHandler CreateHandler(IUnityContainer container)
    {
        return this;
    }

    public IMethodReturn Invoke(IMethodInvocation input,
        GetNextHandlerDelegate getNext)
    {
        {
            Trace.WriteLine("machine name");
            Trace.WriteLine("domain name");
            return getNext()(input, getNext);
        }
    }
}

#endregion

```

C# Unit Test

```

[TestMethod]
public void TestInjectionType()
{
    var container =
        new UnityContainer().AddNewExtension<Microsoft.Practices.
        Unity.InterceptionExtension.Interception>();
}

```

```
//Unity 动态编译并增加装饰类型
container.RegisterType<IBizObject, BizObject>().

Configure<Microsoft.Practices.Unity.InterceptionExtension.Interception>().
    SetInterceptorFor<IBizObject>(new InterfaceInterceptor());

var bizObject = container.Resolve<IBizObject>();

// 调用装饰后的方法
Trace.WriteLine("\nInvoke GetValue()\n-----");
Assert.AreEqual<int>(BizObject.Value, bizObject.GetValue());

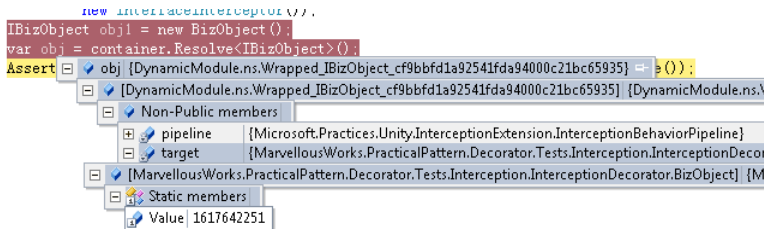
// 调用未装饰的方法
Trace.WriteLine("\nInvoke GetValue()\n-----");
Assert.AreEqual<int>(DateTime.Now.Year, bizObject.GetYear());
}
```

未通过 InterfaceInterceptor 动态编译的类型:



```
IBizObject obj1 = new BizObject();
var obj = container.Resolve<IBizObject>();
Assert.AreEqual<int>(obj.GetValue(), BizObject.Value);
```

通过 InterfaceInterceptor 动态编译的类型:



```
IBizObject obj1 = new BizObject();
var obj = container.Resolve<IBizObject>();
Assert.AreEqual<int>(obj.GetValue(), BizObject.Value);
```

Output 窗口

其中，高亮部分是动态编译后增加的执行内容。

```
----- Test started: Assembly: Decorator.Tests.dll -----
```

```
Invoke GetValue()
```

```
-----
log...
machine name
domain name
```

```
Invoke GetValue()
```

```
-----
1 passed, 0 failed, 0 skipped, took 1.10 seconds (MSTest 10.0).
```

通过上面的示例不难看出动态编译的“威力”。通过这个机制几乎可以在客户程序完全不知情的情况下直接为目标类型增加新的功能，而且可以根据需要精确选

择会影响的方法。如 Output 窗口显示的那样，GetValue()方法标记为会被装饰，所以执行时动态重新编译了 LogAttribute 和 SecurityAttribute 的新增功能；而 GetYear()没有标记为需要装饰，所以它的执行无异。

这个示例中装饰属性仅仅定义在接口部分，对于实体类型没有直接侵入性，但从执行效果看，同样可以获得接口声明的内容。这是我们在“重新研读 C#语言”一章无法实现的能力，毕竟在那个示例中还要为实体类的方法上贴一个 DirectorAttribute 的“标签”。

12.5.4 进一步分析

由上面两个示例不难看出，通过 .NET Remoting 的 MarshalByReference 或 System.Reflection.Emit 命名空间下的对象，可以通过透明代理把装饰类型加载到目标对象上。尤其对于后者，项目中很多非功能性需求可以在不修改外部业务逻辑的情况下，通过给业务对象“点缀”装饰属性后即可增加新的控制功能。

不过上面的示例用于实际项目中也有一定欠缺，主要是性能问题，这是由反射和动态加载所导致的。一些更成熟的框架中会借助缓冲把动态创建的代理类实例保存在内存中，同时把动态生成的 Assembly 保存在文件系统中，从而最大程度地提高执行效率。当然，也可以采用前面单件模式的方式，管理动态编译出的实例数量。

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，比如，.NET Framework 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，主要是因为追踪和调试相对困难，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，本章还引入了拦截机制，使用属性以“横切”的方式装饰目标类型和接口，最大程度地减少装饰类型与目标实体类型间的依赖。不过此方式的使用代价是**实现成本很高**，而且执行中如果不借助缓冲，**其执行效率也会受到一定影响**。但不管怎么说，用动态编译 MSIL 实现装饰类型站在用户角度很好，只不过不是每个人都

“玩”得起的。

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
- 27.7 验证逻辑的有效性
- 27.8 小结
- 27.9 后记

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，这么做的原因还是“唯一不变的是变化本身”。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：

(1) 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。

(2) 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。

(3) 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，实现该目标有一个前提，那就是待构造的对象先被抽象。

(4) 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。

(5) 如果不确定性来自执行过程，则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，比如某些操作、交互过程过于复杂，这时就需要对它们分解。行为型的思维方法就是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：当在应用某个模式的时候，如果同时涉及特定情景的话，可以考虑采用另外某个模式。

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。

不过项目进度摆在那里，而且根据我们自己的经验在某几个点总会有那些变动，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也未尝不是一个方法。

前面我们每章都在谈一个特定的模式，但这样可能整体感不强，为了加深对 GOF 的认识，本章我们做个综合练习，尝试设计一个比较通用的用户认证模块。

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方式，已知的包括：

- USB Key（或 IC 卡）方式。
- 用户名口令方式（UserName / Password）。
- 集成活动目录（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，用户认证过程是一个比较敏感的过程，可能会不断增加额外的非功能性需求，例如记录审计日志、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。

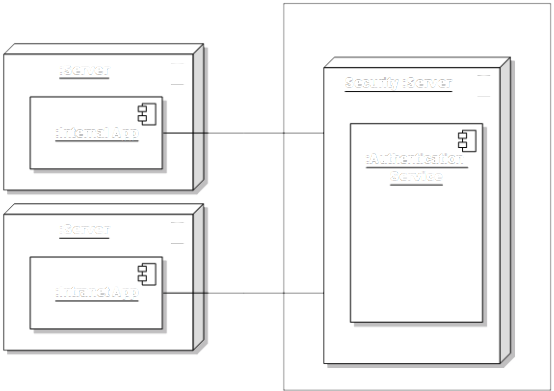


图 27-1 认证系统部署结构

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，完整的示例模型我们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，我们先做第一轮技术分析，大体描绘出如图 27-2 所示的认证服务的交互过程。

- (1) 客户程序调用一个名为“认证子 (IAuthenticator)”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 (IAuthenticationProvider) 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。

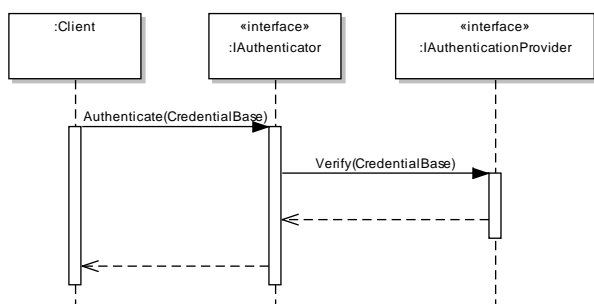


图 27-2 认证服务 Layer 1 的交互过程

参与执行的两个接口的静态结构如图 27-3 所示。

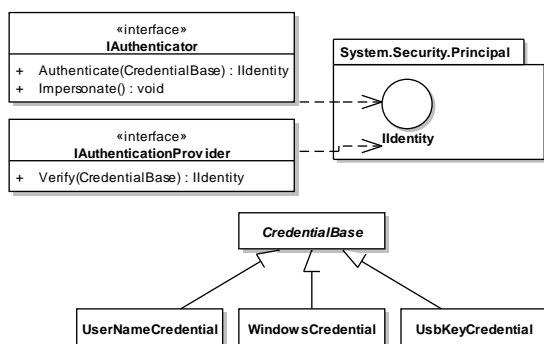


图 27-3 认证服务 Layer 1 的静态结构

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，IAuthenticator 通过选择指定的 IAuthenticationProvider 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 IAuthenticationProvider 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 IAuthenticationProvider 存在很大的不确定性。

考虑到不同 IAuthenticationProvider 运行时需要一定的配置信息，因此 IAuthenticationProvider 对象的创建过程不是简单地 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如下一些设计模式技巧：

增加一个工厂类型，隔离 IAuthenticator 和具体 IAuthenticationProvider 的关系。

根据认证系统对于 USB Key 的使用技术要求，设计 IAuthenticationProvider 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 IAuthenticationProvider。

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，我们将进入第二轮的技术分析。

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。

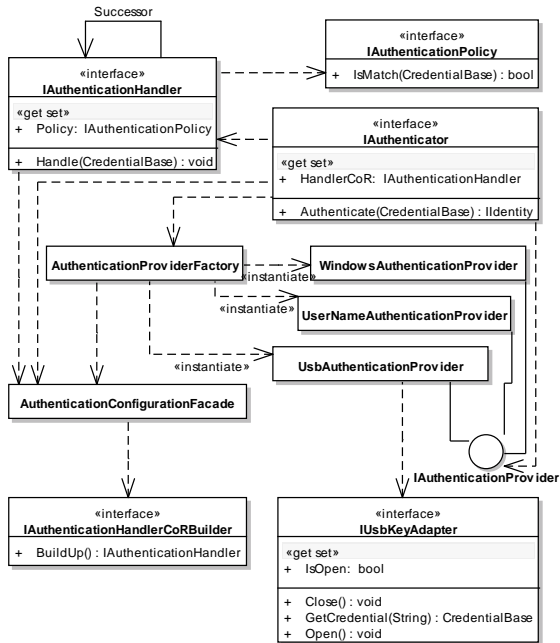


图 27-4 认证服务 Layer 2 的静态过程（为了简化，将部分 Layer 1 已有的部分省略）

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，但其目的是为了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 AuthenticationProviderFactory 工厂类型，用于隔离 IAuthenticator 与具体 IAuthenticationProvider 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 IUsbAdapter，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 IAuthenticationPolicy，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 IAuthenticationHandler 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 IAuthenticationHandler 职责链的装配问题，抽象出 IAuthenticationHandlerCoRBuilder 接口完成该部分工作，确保 AuthenticationProviderFactory 反馈的结果不仅仅是 new() 出一个

	IAuthenticationHandler 实例，还完成对其后继节点和 IAuthenticationPolicy 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 AuthenticationConfigurationFacade 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**。

为了更有效地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 IAuthenticationHandlerCoRBuilder 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，一旦发现配置变化后及时通知 IAuthenticationHandlerCoRBuilder，让它根据最新的配置更新控制策略 CoR 的装配过程。

另外，远程访问和本地访问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。

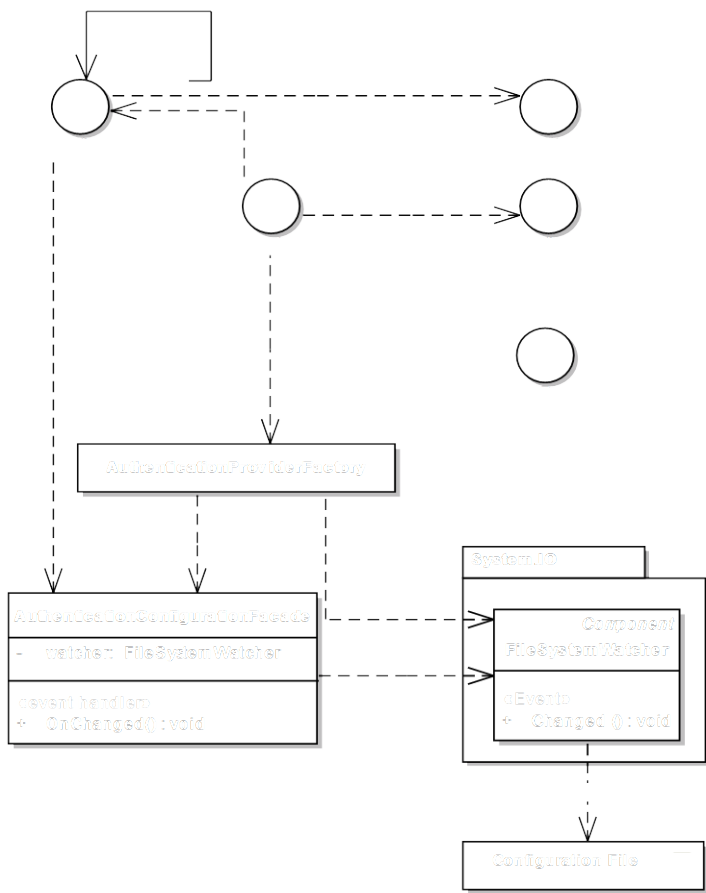


图 27-5 认证服务 Layer 3 的静态过程（为了简化，将部分 Layer 1、Layer 2 已有的部分省略）

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。由于 C#中委托简化了观察者模式的实现方式，这里采用 System.IO.FileSystemWatcher 通过委托机制完成

（续表）


模式	用途
代理模式	考虑到远程访问中接口颗粒度一般较粗，远程访问受到通信协议的不同，实现方式差别迥异，因

(Proxy)	<p>此设计上定义独立的远程接口供远程客户程序调用。</p> <p>尽管我们没有采用《设计模式》中标准的代理模式实现，但在实际项目中可以通过配置不同的 [ServiceContact] 定义哪些可以由远程代理实现、哪些只针对本地。</p> <p>甚至我们可以通过设置某些方法为 [OperationContact]，另外一些方法非 [OperationContact]，可以在一个 interface 中同时表示远程接口和本地接口，这表示实际项目中往往并不需要真正独立的 IAuthenticationProviderRemote 接口</p>
---------	---

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 的构造过程也需要一定的代价，为此将它也收回到 AuthenticationConfigurationFacade 中，由该外观类型缓存并根据配置信息变化动态更新。

完成第三轮技术分析后，下面我们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 个命名空间组织相关类型，划分如图 27-6 所示。

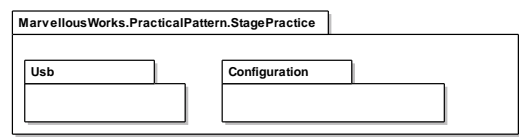


图 27-6 认证服务实现部分的功能划分

其中，每个命名空间的职能如下：

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。

MarvellousWorks.PracticalPattern.StagePractice.Usb: 仅针对 USB Key 方式，定义约束外部第三方驱动力的接口定义。

MarvellousWorks.PracticalPattern.StagePractice.Configuration: 定义各类配置对象。

2. 配置结构设计

接着，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，我们采用以配置为中心的开发思路，在实际编码前完成配置文件的结构。

配置文件的优势我们在“重新研读 C#语言”一章中已经介绍过，而且对于什么内容需要定义在配置文件中也做了说明，下面我们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，包括以下 6 项：

IAuthenticator：定义认证服务整体对外的功能接口（认证子）。

IAuthenticationHandlerCoRBuilder：定义装配 **IAuthenticationHandler** 职责链的创建者。

IUsbKeyAdapter：USB key 驱动适配接口。

IAuthenticationPolicy：定义认证过程非功能性控制策略对象。

IAuthenticationProvider：定义针对具体凭证类型的认证功能提供者。

IAuthenticationHandler：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：**IAuthenticator** 针对具体特定的认证服务运行而言是单一的；**IUsbKeyAdapter** 在一段时间内也是唯一的；**IAuthenticationHandlerCoRBuilder** 功能固定，只需要构造一个 **IAuthenticationHandler** 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，我们定义配置文件的初步结构如下：

App.Config

```
<sectionGroupname="stagePractice">

  <sectionname="authenticator"type="System.Configuration.SingleTagSectionHandler"/>
  <sectionname="handlerCorBuilder"type="System.Configuration.SingleTagSectionHandler"/>
  <sectionname="usbAdapter"type="System.Configuration.SingleTagSectionHandler"/>

  <sectionname
="credentials"type="System.Configuration.DictionarySectionHandler"/>
  <sectionname
="providers"type="System.Configuration.DictionarySectionHandler"/>
  <sectionname
="policies"type="System.Configuration.DictionarySectionHandler"/>
  <sectionname="handlers"type="
System.Configuration.DictionarySectionHandler"/>

</sectionGroup>
```


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组“stagePractice”，将每个单项配置项定义为一个独立的 System.Configuration.SingleTagSectionHandler，而将每个集合性质的配置项统一定义为 System.Configuration.DictionarySectionHandler。之所以没有完全自定义配置节，目的是尽量采用 .NET Framework 内置的配置类型，减少编码量。

A 尽管可以用一个 System.Configuration.SingleTagSectionHandler 同时定义 authenticator、handlerCorBuilder、usbAdapter 3 个配置项，但出于配置文件清晰的考虑，将它们独立分节。

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

所以，进一步整理后配置结构（Schema）调整如下：

App.Config

```
<?xmlversion="1.0"encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroupname="stagePractice"type=" ">
      <sectionname="authenticator"type="System.Configuration.SingleTagSectionHandler"/>
      <sectionname="credentials"type="System.Configuration.DictionarySectionHandler"/>
      <sectionname="providers"type="System.Configuration.DictionarySectionHandler"/>
      <sectionname="policies"type="System.Configuration.DictionarySectionHandler"/>
      <sectionname="handlers"type="自定义配置节" />
    </sectionGroup>
  </configSections>

  <stagePractice>

    <authenticator />
    <credentials/>
    <policies/>
    <handlersbuilder=" " />
    <providers/>

  </stagePractice>
</configuration>
```

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，进行如下设计：统一约定将 System.Configuration.DictionarySectionHandler 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 AssemblyQualifiedName

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 `IAuthenticationHandler` 配置元素设置 3~4 个属性，因此需要为它定制一套独立的配置元素和配置元素集合。

为了从名称上区分“handlers”配置节和“handlers”元素集合，我们将配置节的名称修改为“handlerCoR”，而将它下面的配置元素集合保留命名为“handlers”。

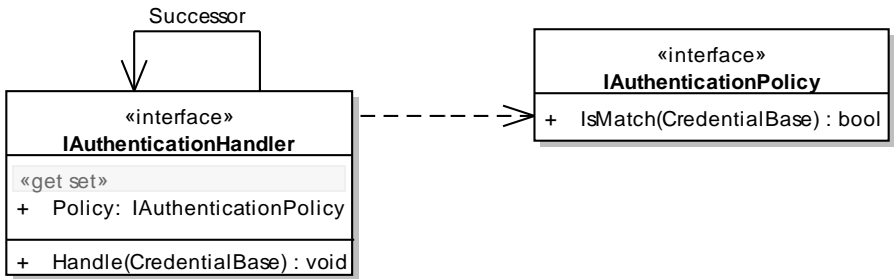


图 27-7 认证过程中非功能性控制措施及适用策略的静态关系

这样，我们为其设计的配置元素结构如下：

App.Config

```
<handlerCoRbuilder=" ">
<handlers>
<addkey=" " seq=" " po=" " value=" " />
</handlers>
</handlerCoR>
```

按照以下描述，“key”属性表示该 `IAuthenticationHandler` 的逻辑名称，“value”属性表示对应实体类型的 `AssemblyQualifiedName` 类型名称，而“seq”属性代表该 `IAuthenticationHandler` 在整个职责链中的执行次序，“po（Policy）”表示该 `IAuthenticationHandler` 适用的 `IAuthenticationPolicy` 的逻辑名称。

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 `AuthenticationConfigurationFacade` 类型向上层逻辑提供配置访问接口。

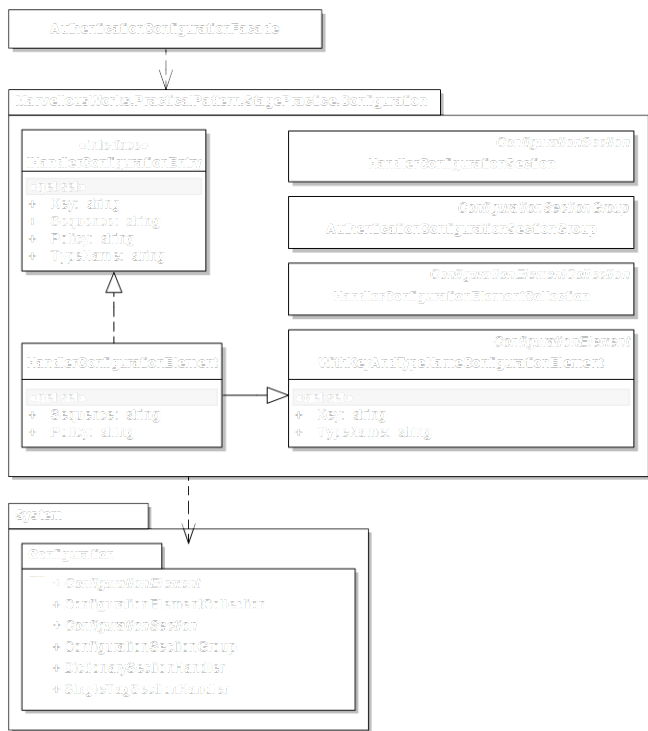


图 27-8 认证服务配置子系统的静态关系

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。因此如你看到的那样，上述部分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“削足适履”，硬着头皮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 `AuthenticationConfigurationFacade` 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 中的依赖关系，我们先要设计、实现用于装配 `IAAuthenticationHandler` 的 `IAAuthenticationHandlerCoRBuilder`。
- (2) 完成 `AuthenticationConfigurationFacade` 主干逻辑设计。
- (3) 为保证 `AuthenticationConfigurationFacade` 能够根据配置修改动态更新，再为其增加 `File Watcher` 的实现部分。

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

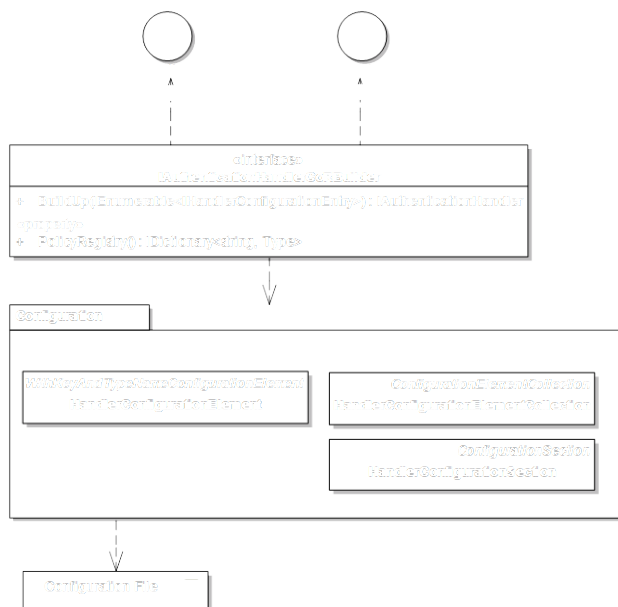


图 27-9 `IAuthenticationHandlerCoRBuilder` 静态结构

尽管我们定义了 `IAuthenticationHandlerCoRBuilder`，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。

C# `IAuthenticationHandlerCoRBuilder`

```

/// 装配 IAuthenticationHandler CoR 的创建者
public interface IAuthenticationHandlerCoRBuilder
{
    /// 装配
    /// <returns> IAuthenticationHandler CoR 的入口节点 </returns>
    IAuthenticationHandler BuildUp(IEnumerable<IHandlerConfigurationEntry>
    config);

    /// 非决定是否执行某个 Handler 的非功能性控制策略名称和类型的映射关系
    IDictionary<string, Type> PolicyRegistry { get; set; }
}
  
```

C# `AuthenticationHandlerCoRBuilder`

```

using System;
using System.Collections.Generic;
using System.Linq;
  
```

```

using MarvellousWorks.PracticalPattern.StagePractice.Configuration;
namespace MarvellousWorks.PracticalPattern.StagePractice
{
    ///<summary>
    ///<see cref="IAuthenticationHandlerCoRBuilder"/>
    ///</summary>
    public class AuthenticationHandlerCoRBuilder :
        IAuthenticationHandlerCoRBuilder
    {
        public IDictionary<string, Type> PolicyRegistry { get; set; }

        ///<summary>
        ///构建 Handler CoR
        ///</summary>
        ///<param name="config">登记 handler 的配置信息</param>
        ///<returns></returns>
        public virtual IAuthenticationHandler
        BuildUp(IEnumerable<IHandlerConfigurationEntry> config)
        {
            if ((config == null) || (config.Count() == 0)) return null;

            //按照配置的执行次序排列非功能性控制 handler
            config.OrderByDescending((x) => Convert.ToInt32(x.Sequence));
            var entries =
                (from e in config
                 orderby Convert.ToInt32(e.Sequence)
                 select new
                 {
                     Handler =
                     (IAuthenticationHandler)Activator.CreateInstance(Type.GetType(e.TypeName)),
                     Policy = (IAuthenticationPolicy)Activator.CreateInstance
                     (PolicyRegistry[e.Policy])
                 }).ToArray();

            //定义 Handler CoR 返回结果的根节点
            var root = entries.First().Handler;
            root.Policy = entries.First().Policy;
            var current = root;

            //建立 Handler CoR 的链表关系
            if (entries.Count() > 1)
            {
                for (var i = 1; i < entries.Count(); i++)
                {
                    entries[i].Handler.Policy = entries[i].Policy;
                    entries[i - 1].Handler.Successor = entries[i].Handler;
                }
                root.Successor = entries[1].Handler;
            }
            return root;
        }
    }
}

```

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

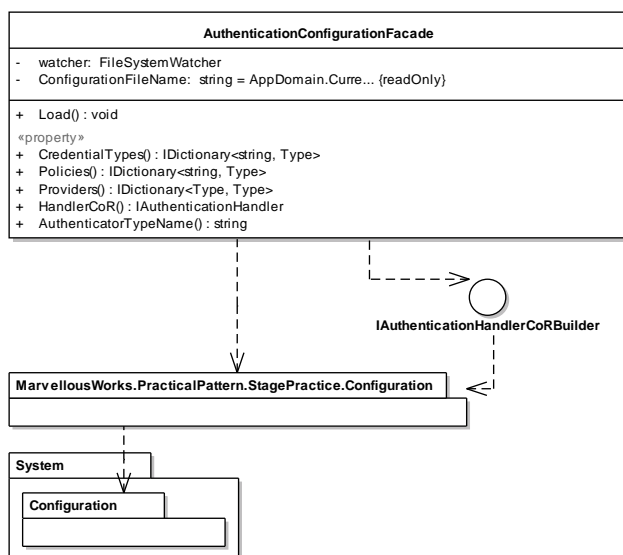


图 27-10 配置访问外观类型的静态结构

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 `Load()` 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。下面是裁剪后的实现片段：

C#

```

///认证过程相关配置访问对象
///<remarks>为了简化示例中仅支持 WinForm 应用</remarks>
public static class AuthenticationConfigurationFacade
{
    static FileSystemWatcher watcher;

    #region 简化配置访问的属性方法
    public static IDictionary<string, Type> CredentialTypes { get; private set; }
    public static IDictionary<string, Type> Policies { get; private set; }
    public static IDictionary<Type, Type> Providers { get; private set; }
    public static IAAuthenticationHandler HandlerCoR { get; private set; }
    public static string AuthenticatorTypeName { get; private set; }
    #endregion

    static AuthenticationConfigurationFacade()
    {
        Load();
    }

    ///<summary>
    ///读取配置文件
    ///</summary>
    public static void Load()
    {

```

```

ConfigurationManager.RefreshSection(GetFullSectionName(Constant.ProvidersSectionName));
ConfigurationManager.RefreshSection(GetFullSectionName(Constant.CredentialSectionName));
ConfigurationManager.RefreshSection(GetFullSectionName(Constant.PoliciesSectionName));
ConfigurationManager.RefreshSection(GetFullSectionName(Constant.HandlerCoRSectionName));

// 加载名称与 Credential 类型的对应关系
    CredentialTypes =
LoadDictionaryConfigSection<string>(Constant.CredentialSectionName,
typeof(CredentialBase));
// 加载认证非功能性控制策略类型列表
    Policies =
LoadDictionaryConfigSection<string>(Constant.PoliciesSectionName,
typeof(IAuthenticationPolicy));
// 加载不同认证凭证所需的认证 Provider 类型
    Providers =
LoadDictionaryConfigSection<Type>(Constant.ProvidersSectionName,
typeof(IAuthenticationProvider), (x) => CredentialTypes[x]);
// 加载认证子类型名称
    AuthenticatorTypeName = GetAuthenticatorTypeName();
// 加载各种非功能性控制措施的配置信息
    HandlerCoR = GetHandlerCoR();
}

#region Helper Methods

static IAuthenticationHandler GetHandlerCoR()
{
    var section = (ConfigurationManager.OpenExeConfiguration
(ConfigurationUserLevel.None).GetSectionGroup(Constant.SectionGroupName)
as AuthenticationConfigurationSectionGroup).Handlders; ;
    var handlerBuilder = (IAuthenticationHandlerCoRBuilder)(Activator.
CreateInstance(Type.GetType(section.BuilderTypeName)));
    handlerBuilder.PolicyRegistry = Policies;
    return handlerBuilder.BuildUp(section.GetHandlers());
}

///<summary>
///根据项目特点，读取 System.Configuration.DictionarySectionHandler 配置节的公共方法
///</summary>
///<typeparam name="TKey">转化后的配置集合信息的键值类型</typeparam>
///<param name="sectionName">配置节名称</param>
///<param name="exceptedType">"value"属性定义的 AssemblyQualifiedName 类型名称
    所期望的目标类型</param>
///<param name="keyConverter">配置键值 string 类型到 TKey 的转换委托函数</param>
///<returns>转化后的配置集合信息</returns>
static IDictionary<TKey, Type> LoadDictionaryConfigSection<TKey>(string
sectionName, Type exceptedType, Func<string, TKey> keyConverter)
{
    if (string.IsNullOrEmpty(sectionName))

```

```

thrownewAbandonedMutexException("sectionName");
if(exceptedType == null) thrownewArgumentNullException("exceptedType");

IDictionary<TKey, Type> result = newDictionary<TKey, Type>();
// 读取配置文件中的 System.Configuration.DictionarySectionHandler 信息
var config = ConvertDictionaryConfigurationSection
(ConfigurationManager.GetSection(GetFullSectionName(sectionName)));
if (config == null) thrownewConfigurationErrorsException(sectionName);
foreach (var entry in config)
{
    var type = Type.GetType(entry.Value);
    CheckTypeAssignment(type, exceptedType);
    if (keyConverter == null)
        result.Add((TKey)Convert.ChangeType(entry.Key, typeof(TKey)),
            type);
    else
        result.Add(keyConverter(entry.Key), type);
}
return result;
}

staticIDictionary<TKey, Type> LoadDictionaryConfigSection<TKey>(string
sectionName, Type exceptedType)
{
    return LoadDictionaryConfigSection<TKey>(sectionName, exceptedType, null);
}

staticIEnumerable<KeyValuePair<string, string>>
ConvertDictionaryConfigurationSection(object target)
{
    if (target == null) returnnull;
    var config = target asHashtable;
    var result = newDictionary<string, string>();
    foreach (DictionaryEntry entry in config)
        result.Add(entry.Key.ToString(), entry.Value.ToString());
    return result;
}

///验证类型转换
staticvoid CheckTypeAssignment(Type source, Type target)
{
    if ((source == null) || (target == null))
        thrownewArgumentNullException();
    if (!target.IsAssignableFrom(source))
        thrownewInvalidCastException(
            string.Format(Resources.ExceptionCanNotConvertType, source.FullName,
                target.FullName));
}

...

#endregion
}

```

然后，为 AuthenticationConfigurationFacade 增加“动态”更新的功能，选择通过 File Watcher 回调 Load() 的方法动态更新缓存的配置信息，而 File Watcher 对 Load()

方法的绑定则定义在 `AuthenticationConfigurationFacade` 的静态构造函数中。

C#

```
static AuthenticationConfigurationFacade()
{
    Load();

    //绑定 File Watcher
    watcher = new FileSystemWatcher(Environment.CurrentDirectory);
    watcher.NotifyFilter = NotifyFilters.LastWrite;
    watcher.Filter = "*.config";
    watcher.Changed += (x, y) => Load();    // File Watcher 回调方法
    watcher.EnableRaisingEvents = true;    //启动 File Watcher
}
```

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 `AuthenticationHandlerCoRBuilder` 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，我们还要增加两个工厂类型。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。

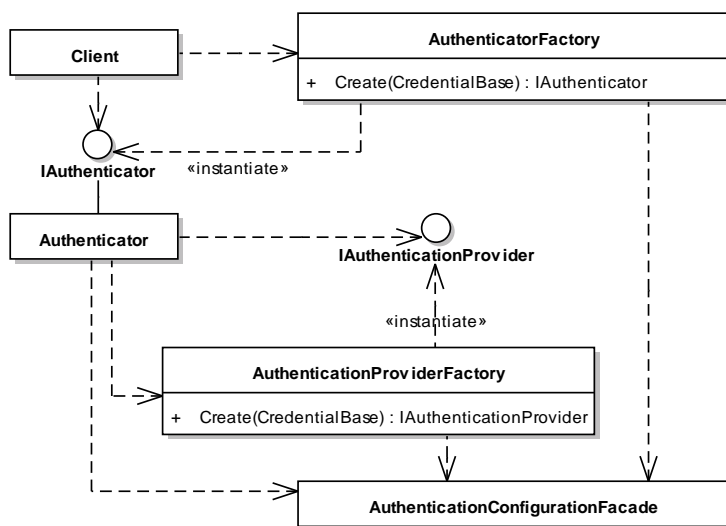


图 27-11 认证服务的静态结构

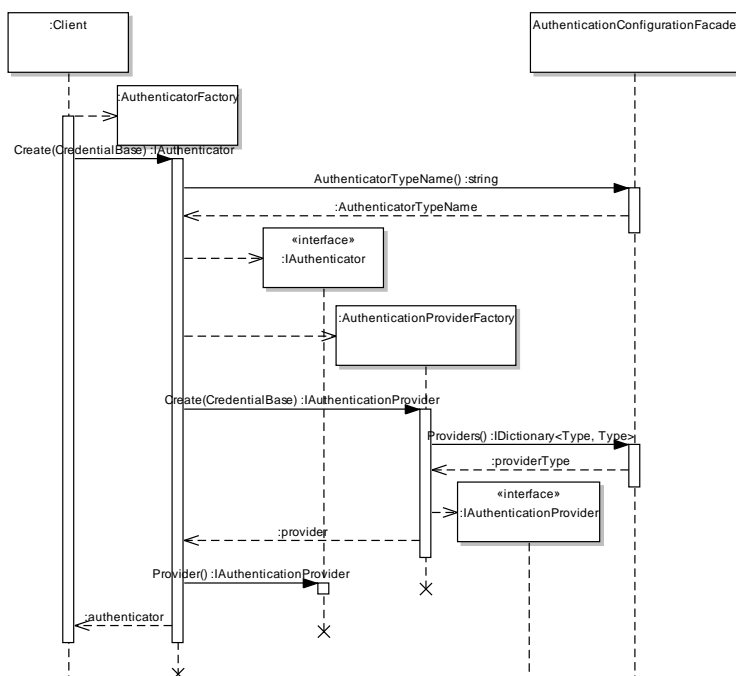


图 27-12 实例化认证服务实体类型的动态过程

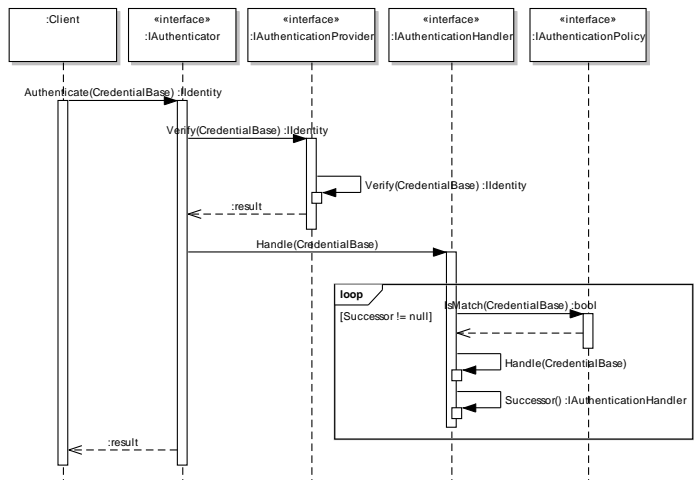


图 27-13 认证服务的认证过程

认证的实际代码如下（已去除输出调试信息的部分）：

C#

```
///默认的实现
public class Authenticator : IAuthenticator
{
    ///默认的实现过程
    public virtual IIdentity Authenticate(CredentialBase credential)
    {
        ///标准的验证过程
        var result = Provider.Verify(credential);
        ///非功能性控制策略
        if (AuthenticationConfigurationFacade.HandlerCoR != null)
            AuthenticationConfigurationFacade.HandlerCoR.Handle(credential);

        return result;
    }

    ///Provider 类型
    public IAuthenticationProvider Provider { get; set; }
}
```

27.7 验证逻辑的有效性

1. 编写测试类型

为了验证上述逻辑的有效性，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。

C#

```
///<summary>
///凭据抽象类型
///</summary>
publicabstractclassCredentialBase{}

publicpartialclassUserNameCredential : CredentialBase { }
publicpartialclassWindowsCredential : CredentialBase { }
publicpartialclassUsbKeyCredential : CredentialBase { }

classCustomsCredential : CredentialBase{}
```

(2) 示例用户身份标识类型。

C#

```
classIdentityMock : IIdentity
{
    publicbool IsAuthenticated { get; set; }
    publicstring Name { get; set; }
    publicstring AuthenticationType { get; set; }

    publicstring State { get; set; }

    publicoverridestring ToString()
    {
        returnstring.Format("\n\tIsAuthencated : {0}\n\t Name : {1}\n\t Credential
Type : {2}\n\t State : {3}", IsAuthenticated, Name, AuthenticationType, State);
    }
}
```

(3) 示例非功能性控制策略。

C#

```
classAuthenticationPolicyMockBase : IAuthenticationPolicy
{
    protectedIList<Type> ApplyToCredentialTypes { get; set; }

    publicbool IsMatch(CredentialBase credential)
    {
        if (credential == null) thrownewArgumentNullException("credential");
        bool result;
        if (ApplyToCredentialTypes.Count == 0)
            result = false;
        else
```

```

        result = ApplyToCredentialTypes.Contains(credential.GetType());
    if(result)
    Trace.WriteLine(string.Format(Resources.OutputFormat, "Policy",
    GetType().Name, "match"));
    return result;
    }
}

///仅适于 USBKey 类型凭证
classUsbKeyPolicy : AuthenticationPolicyMockBase
{
    public UsbKeyPolicy()
    {
        ApplyToCredentialTypes = newList<Type>(){typeof(UsbKeyCredential)};
    }
}

///仅适于用户名/口令类型凭证
classUserNamePolicy : AuthenticationPolicyMockBase
{
    public UserNamePolicy()
    {
        ApplyToCredentialTypes = newList<Type>()
        { typeof(UserNameCredential) };
    }
}

///适于用户名/口令和 USBKey 两种凭证类型
classUserNameAndUsbKeyPolicy : AuthenticationPolicyMockBase
{
    public UserNameAndUsbKeyPolicy()
    {
        ApplyToCredentialTypes = newList<Type>() { typeof(UserNameCredential),
        typeof(UsbKeyCredential) };
    }
}

///适用于所有凭证类型
classAllPolicy : IAuthenticationPolicy
{
    publicbool IsMatch(CredentialBase credential)
    {
        Trace.WriteLine(string.Format(Resources.OutputFormat, "Policy",
        GetType().Name, "match"));
        returntrue;
    }
}

```

(4) 示例非功能性控制措施。

C#

```

classAuthenticationHandlerMockBase : IAuthenticationHandler
{
    protectedFunc<object> GetValue { get; set; }
    public AuthenticationHandlerMockBase(Func<object> GetValue){this.GetValue =

```

```

GetValue;}

public IAuthenticationPolicy Policy { get; set; }
public IAuthenticationHandler Successor { get; set; }

public virtual void Handle(CredentialBase credential)
{
    if (credential == null) throw new ArgumentNullException("credential");
    if (Policy.IsMatch(credential))
    {
        Trace.WriteLine(string.Format(Resources.OutputFormat, "Handler",
            GetType().Name, GetValue().ToString()));
    }
    if (Successor != null)
        Successor.Handle(credential);
}

class LogProcessorCountHandler : AuthenticationHandlerMockBase
{
    public LogProcessorCountHandler(): base(() =>Environment.ProcessorCount){}
}

class LogClrVersionHandler : AuthenticationHandlerMockBase
{
    public LogClrVersionHandler() : base(() =>Environment.ProcessorCount) { }
}

class LogOsVersionHandler : AuthenticationHandlerMockBase
{
    public LogOsVersionHandler() : base(() =>Environment.OSVersion) { }
}

```

(5) 示例的第三方 USB Key 驱动和 USB Key 适配器。

C#

```

using System.Diagnostics;
namespace External
{
    ///模拟第三方提供的接口不兼容的驱动程序
    class UsbDriver
    {
        bool available;
        public bool Avaliable {
            get
            {
                Trace.WriteLine("External UsbDriver.Avaliable.get()");
                return available;
            }
            set
            {
                Trace.WriteLine("External UsbDriver.Avaliable.set()");
                available = value;
            }
        }
    }
}

```

```

namespace MarvellousWorks.PracticalPattern.StagePractice.Tests.Usb
{
    using System.ServiceModel.Security;
    using MarvellousWorks.PracticalPattern.StagePractice.Usb;
    using External;

    publicclass UsbAdapter : IUsbKeyAdapter
    {
        publicconststring Pin = "hello";
        publicconststring Name = "external";

        UsbDriver driver = newUsbDriver();

        publicvoid Close()
        {
            Trace.WriteLine("UsbAdapter.Close()");
            driver.Avaliable = false;
        }

        publicCredentialBase GetCredential(string pin)
        {
            Trace.WriteLine("UsbAdapter.GetCredential()");
            if(!string.Equals(pin, Pin)) thrownewSecurityAccessDeniedException();
            returnnewUsbKeyCredential();
        }

        publicbool IsOpen
        {
            get { return driver.Avaliable; }
            set{ driver.Avaliable = value;}
        }

        publicvoid Open()
        {
            Trace.WriteLine("UsbAdapter.Open()");
            if(!IsOpen)
                driver.Avaliable = true;
        }
    }
}

```

(6) 示例认证服务提供者类型。

C#

```

abstractclass AuthenticationProviderMockBase : IAuthenticationProvider
{
    publicabstractstring AuthenticationType { get; }
    publicabstractType CredentialType { get; }

    publicconststring Name = "mock";
    publicconststring State = "provider";

    publicvirtualIIdentity Verify(CredentialBase credential)
    {
        if((credential == null) || (credential.GetType() != CredentialType))
            thrownewArgumentException("credential");
    }
}

```

```

return new IdentityMock()
{
    AuthenticationType = AuthenticationType,
    Name = Name,
    State = State,
    IsAuthenticated = true
};
}
}

class UsbKeyProvider : AuthenticationProviderMockBase
{
    public override string AuthenticationType { get { return "usb"; } }
    public override Type CredentialType { get { return typeof(UsbKeyCredential); } }

    public override IIdentity Verify(CredentialBase credential)
    {
        var adapter = new UsbAdapter();
        if (!adapter.IsOpen)
            adapter.Open();
        if (credential.GetType() != adapter.GetCredential(UsbAdapter.Pin).GetType())
            throw new NotSupportedException();
        var result = base.Verify(credential);
        adapter.Close();
        return result;
    }
}

class WindowsProvider : AuthenticationProviderMockBase
{
    public override string AuthenticationType { get { return "win"; } }
    public override Type CredentialType { get { return typeof(WindowsCredential); } }
}

class UserNameProvider : AuthenticationProviderMockBase
{
    public override string AuthenticationType { get { return "userName"; } }
    public override Type CredentialType { get { return typeof(UserNameCredential); } }
}

```

(7) 示例认证子。

C#

```
class NewAuthenticator : Authenticator{}
```

2. 执行单元测试

(1) 示例程序配置文件。

单元测试的配置文件如下（为了简化已经去除自定义代码的命名空间和 Assembly 文件名称）：

App.Config

```
<?xmlversion="1.0" encoding="utf-8" ?>
```



```

<configuration>
<configSections>
<sectionGroupname="stagePractice" type="
AuthenticationConfigurationSectionGroup">
<sectionname="authenticator" type="System.Configuration.SingleTagSectionHan
dler" />
<sectionname
="credentials" type="System.Configuration.DictionarySectionHandler" />
<sectionname
="providers" type="System.Configuration.DictionarySectionHandler" />
<sectionname
="policies" type="System.Configuration.DictionarySectionHandler" />
<sectionname="handlerCoR" type=" HandlerConfigurationSection,
StagePractice" />
</sectionGroup>
</configSections>

<stagePractice>

<authenticatorvalue=" NewAuthenticator"/>

<!--认证凭据名称和凭据类型的对应关系-->
<credentials>
<addkey="usb"value="UsbKeyCredential"/>
<addkey="windows"value="WindowsCredential"/>
<addkey="userName"value="UserNameCredential"/>
</credentials>

<!--与认证相关的各种策略-->
<policies>
<addkey="usbPo"value="UsbKeyPolicy"/>
<addkey="usbAndUserNamePo"value="UserNameAndUsbKeyPolicy"/>
<addkey="userNamePo"value="UserNamePolicy"/>
<addkey="allPo"value="AllPolicy"/>
</policies>

<handlerCoRbuilder="NewHandlerCoRBuilder">
<handlers>
<addkey="processor"seq="3"po="usbPo"value="LogProcessorCountHandler"/>
<addkey="clr"seq="1"po="allPo"value="LogClrVersionHandler"/>
<addkey="os"seq="2"po="usbAndUserNamePo"value="LogOsVersionHandler"/>
</handlers>
</handlerCoR>

<providers>
<addkey="usb"value="UsbKeyProvider"/>
<addkey="windows"value="WindowsProvider"/>
<addkey="userName"value="UserNameProvider"/>
</providers>
</stagePractice>

</configuration>

```

(2) 验证读取配置信息。

首先，验证读取配置信息。

C# Unit Test

///确认正常读取 Provider

```
[TestMethod]
public void TestLoadAuthenticationProvider()
{
    Assert.AreEqual<Type>(AuthenticationConfigurationFacade.Providers[typeof(UsbKeyCredential)],
        typeof(UsbKeyProvider));
    Assert.AreEqual<Type>(AuthenticationConfigurationFacade.Providers[typeof(WindowsCredential)],
        typeof(WindowsProvider));
    Assert.AreEqual<Type>(AuthenticationConfigurationFacade.Providers[typeof(UsernameCredential)],
        typeof(UsernameProvider));
}
```

///确认正常读取 Policy

```
[TestMethod]
public void TestLoadAuthenticationPolicy()
{
    Assert.AreEqual<Type>(AuthenticationConfigurationFacade.Policies["usbPo"],
        typeof(UsbKeyPolicy));
    Assert.AreEqual<Type>(AuthenticationConfigurationFacade.Policies["allPo"],
        typeof(AllPolicy));
}
```

///确认正常读取 Handler

```
[TestMethod]
public void TestLoadAuthenticationHandler()
{
    var handlerRoot = AuthenticationConfigurationFacade.HandlerCoR;
    Assert.IsInstanceOfType(handlerRoot, typeof(LogClrVersionHandler));
    Assert.IsInstanceOfType(handlerRoot.Policy, typeof(AllPolicy));
    Assert.IsInstanceOfType(handlerRoot.Successor,
        typeof(LogOsVersionHandler));
    Assert.IsInstanceOfType(handlerRoot.Successor.Successor,
        typeof(LogProcessorCountHandler));
}
```

///确认正常读取 Authenticator

```
[TestMethod]
public void TestLoadAuthenticatorType()
{
    Assert.AreEqual<Type>(typeof(NewAuthenticator),
        Type.GetType(AuthenticationConfigurationFacade.AuthenticatorTypeName));
}
```

///确认正常读取 Handler CoR Builder

```
[TestMethod]
public void TestLoadConfigHandlerCoRBuilderType()
{
    var doc = new XmlDocument();
    doc.Load(ConfigFileName);
    var typeName =

    doc.SelectSingleNode(HandlerBuilderXPath).Attributes[Constant.BuilderItem]
```

```
.Value;
Assert.AreEqual<Type>(typeof(NewHandlerCoRBuilder),
Type.GetType(typeName));
Assert.IsTrue(typeof(IAuthenticationHandlerCoRBuilder).IsAssignableFrom(
typeof(NewHandlerCoRBuilder)));
}
```

(3) 验证动态加载配置信息。

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

C# Unit Test

```
///<summary>
///验证 AuthenticationConfigurationFacade 的 file watcher 能否根据配置文件修改动态
加载最新的配置
///</summary>
///<remarks>
///为了跟踪效果，建议每次执行前重新 build 一下该项目，更新 App.Config
///另外，应检查是否启动了多个 UnitTest 进程阻止修改.config 文件
///</remarks>
[TestMethod]
public void TestConfigFileWatcher()
{
    var credentials = AuthenticationConfigurationFacade.CredentialTypes;
    var providers = AuthenticationConfigurationFacade.Providers;

    //确认原始配置信息是不同的
    var key1 = credentials.ElementAt(0).Key;
    var key2 = credentials.ElementAt(1).Key;
    Assert.AreNotEqual<Type>(providers[credentials[key1]],
providers[credentials[key2]]);

    //修改配置信息
    ModifyConfigFile(key1, key2);

    // 等待 File Watcher 生效
    Thread.Sleep(SleepMilliseconds);

    //重新获取最新的配置信息
    providers = AuthenticationConfigurationFacade.Providers;
    //确认配置信息已经更新
    Assert.AreEqual<Type>(providers[credentials[key1]],
providers[credentials[key2]]);
}

void ModifyConfigFile(string keySource, string keyTarget)
{
    var doc = new XmlDocument();
    doc.Load(ConfigFileName);
    var nodes = doc.SelectNodes(ProvidersXPath);
    var nodeSource = GetNodeByKey(keySource, nodes);
```

```

var nodeTarget = GetNodeByKey(keyTarget, nodes);
nodeTarget.Attributes[ValueItem].Value =
nodeSource.Attributes[ValueItem].Value;
doc.Save(ConfigFileName); // 保存对配置文件的修改
}

```

Output

```

----- Test started: Assembly: StagePractice.Tests.dll -----
1 passed, 0 failed, 0 skipped, took 13.04 seconds (MSTest 10.0).

```

上面的单元测试确认 **AuthenticationConfigurationFacade** 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 **File Watcher** 有机会对更新做出响应。

(4) 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 **IAuthenticator** 接口及其默认实现类型的有效性。这里我们选择以下 3 类凭证进行验证：

USB Key。

用户名/口令。

自定义的 **CustomsCredential**。

配置文件对于 3 类认证凭证的描述如下：

App.Config

```

<policies>
<add key="usbPo" value="UsbKeyPolicy" />
<add key="usbAndUserNamePo" value="UserNameAndUsbKeyPolicy" />
<add key="userNamePo" value="UserNamePolicy" />
<add key="allPo" value="AllPolicy" />
</policies>

<handlerCoR builder="NewHandlerCoRBuilder">
<handlers>
<add key="processor" seq="3" po="usbPo" value="LogProcessorCountHandler" />
<add key="clr" seq="1" po="allPo" value="LogClrVersionHandler" />
<add key="os" seq="2" po="usbAndUserNamePo" value="LogOsVersionHandler" />
</handlers>
</handlerCoR>

<providers>
<add key="usb" value="UsbKeyProvider" />
<add key="windows" value="WindowsProvider" />
<add key="userName" value="UserNameProvider" />
</providers>

```

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“processor”、“clr”、“os”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“usbAndUserNamePo”和“allPo”两个策略，因此会执行“clr”、“os”两个非功能性措施的Handler。

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。

相应的单元测试也验证了配置文件中的控制要求：

C# Unit Test

```
[TestClass]
public class AuthenticatorFixture
{
    /// 验证 USB Key 类凭证
    [TestMethod]
    public void TestUsbKeyAuthentication()
    {
        Trace.WriteLine("TestUsbKeyAuthentication");
        var credential = new UsbKeyCredential();
        new AuthenticatorFactory().Create(credential).Authenticate(credential);
    }

    /// 验证用户名/口令类凭证
    [TestMethod]
    public void TestUserNameAuthentication()
    {
        Trace.WriteLine("TestUserNameAuthentication");
        var credential = new UserNameCredential();
        new AuthenticatorFactory().Create(credential).Authenticate(credential);
    }

    /// 新增的自定义凭证类型
    /// <exception cref="NotSupportedException">因为没有定义相应的认证服务 Provider，
    因此会抛出异常</exception>
    [TestMethod]
    [ExpectedException(typeof(NotSupportedException))]
    public void TestCustomsAuthentication()
    {
        Trace.WriteLine("TestCustomsAuthentication");
        var credential = new CustomsCredential();
        new AuthenticatorFactory().Create(credential).Authenticate(credential);
    }
}
```

Output

```
----- Test started: Assembly: StagePractice.Tests.dll -----

TestUsbKeyAuthentication
credential type is UsbKeyCredential
External UsbDriver.Avaliable.get()
UsbAdapter.Open()
```

```

External UsbDriver.Avaliable.get()
External UsbDriver.Avaliable.set()
UsbAdapter.GetCredential()
UsbAdapter.Close()
External UsbDriver.Avaliable.set()
[Policy] AllPolicy : match
[Handler] LogClrVersionHandler : 2
[Policy] UserNameAndUsbKeyPolicy : match
[Handler] LogOsVersionHandler : Microsoft Windows NT 6.1.7600.0
[Policy] UsbKeyPolicy : match
[Handler] LogProcessorCountHandler : 2
authentication result :
    IsAuthencated : True
    Name : mock
    Credential Type : usb
    State : provider

TestUserNameAuthentication
credential type is UserNameCredential
[Policy] AllPolicy : match
[Handler] LogClrVersionHandler : 2
[Policy] UserNameAndUsbKeyPolicy : match
[Handler] LogOsVersionHandler : Microsoft Windows NT 6.1.7600.0
authentication result :
    IsAuthencated : True
    Name : mock
    Credential Type : userName
    State : provider

TestCustomsAuthentication
credential type is CustomsCredential

3 passed, 0 failed, 0 skipped, took 0.68 seconds (MSTest 10.0).

```

(5) 测试结论。

综上所述，通过一系列单元测试我们从各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。

27.8 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.NET 环境下模式应用技巧的体会。前面我们综合运用的模式及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。

下面，我们再次对本章前面的设计思路进行归纳，包括设计和实现两个部分。

1. 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。

2. 实现部分

首先，针对认证服务这个比较技术化的系统，我们遵循了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 .NET Framework 自带的配置类型完成，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。

27.9 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，入手也会慢一些，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 C# 版）》中将看到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 Java 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 JavaEE 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一个非常不错的选择。博采众长，而不是受厂商的影响将自己定死在一个平台上，这能帮助我们更快地成长。视野决定高度，做开发同样如此。