Broadview®
www.broadview.com.cn 博文视点

设计模式

基于C#的工程化实现及扩展

王 翔 著

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

谨以此书献给我最敬重的爷爷、献给养育我成长的父母，以及各位家人和一直以来关心、帮助和支持我的各位朋友。

作者访谈录

博文小林　　王　翔

针对王翔老师的新书《设计模式——基于C#的工程化实现及扩展》的出版，博文视点编辑小林对他进行了邮件专访，现将博文的编辑与王老师的对话整理成文，以飨读者。

博文小林

王翔老师，您好！您即将出版的《设计模式——基于 C#的工程化实现及扩展》这本新书，是您结合项目经验撰写而成的。市面上已经有一些关于 23 种设计模式的书，有的已经获得了市场和读者的认可，您在文中重新介绍这 23 种模式时融入了哪些新元素，这些元素对读者会有哪些帮助？

王　翔

设计模式是套思想，我是一名从事开发的人员，有时候会觉得如何更巧妙地结合应用功能完成实现须要多多斟酌。

在我的师傅、同事和领导的帮助下，这几年做了一些.NET 的项目，我酷爱 C#语言，在使用中发现 C#实现一些设计模式的时候有很多特色。

虽然本书很多模式的扩展都很有限，但我希望将其作为一个引子，能够与喜欢模式 & C#语言的朋友有个讨论的基础，不断完善这部分内容。

从以上写作目的出发，我主要希望该书能够对实际工作有以下帮助：

1. 打破一些固有的套路。

2. 用 C#以简洁、直接的手段解决易于变化的问题。
3. 不要仅仅将依赖关系定格在对象体系上，应更多考虑到应用开发、运维不同生命周期中参与者的工作特点，将依赖拓宽到对象、配置体系、数据存储和服务体系。
4. 面向 Web、面向混合信息体系、面向服务。

博文小林

每个程序员都是有独立性格的个体，这些性格会给学习带来方法和思路上的影响，您认为程序员学习和使用设计模式来进行开发的时候最须关注的要点是什么？

王　翔

同为程序开发人员，我们在对待自己工作的时候总或多或少有些“止于至善”的心结。

代码、类库、应用框架不仅仅是老板和项目经理眼中的产品，更是我们敝帚自珍的工作成果，虽然我们可以接受“唯一不变的就是变化本身”，但修改自己的代码，尤其是因为上游需求不确定带来这种压力的时候，总不是愉快的。

我们要借鉴并应用那些成熟的套路，将变化抽象并集中在几个点，然后把它们交给运维人员来处理，而我们把时间更多地放在创造性的工作上。

所以，模式是现成的，但实现套路要靠自己。

如果说开发中我们用什么态度使用模式最重要，我自己的体会有两点：

1. 敏思（不唯套路，同时不囿于局部）。
2. 厉行（如果在适应变化、根据上下文需要应用某些或某个模式的时候，能够用语法支持的特性、FCL 完成，则绝不随便构造一整套类型系统，而是直达目的）。

博文小林

《设计模式——基于C#的工程化实现及扩展》一书中，我们看到了您多年经验的积累和总结。那么对于一个设计模式的初学者，您对他（她）们在学习设计模式和本书时有什么好的建议？在遇到难以捉摸的问题时有什么比较通用且有效的解决办法？

王　翔

我是个比较循规蹈矩的人，所以我的建议可能收效会慢点。我认为设计模式虽然是

思想，但需要对一个面向对象语言有比较扎实的基础。

以我自己为例。

接触.NET 是因为有个项目我师傅觉得用.NET 来实现不错，然后我去师傅那里抱佛脚，师傅打开.NET Framework SDK Documentation，指着语言参考的 C#部分和 Programming with.NET Framework 说：“这是你要的”。

不知道是不是有意指点我，记得那段时间他的 MSN Personal Message 是“聪明人懂得下笨功夫”。

既然师傅这么安排，那我就一头扎进.NET SDK，7 个月后.NET 项目延期了，不过我把那些文档全部看完了，C#语言参考还看了两遍，所有的示例代码也都调试过了。

然后看到《设计模式：可复用面向对象软件的基础》，第一遍看的时候，经常觉得“哦，这个太 Cool，有收获”，但第二遍看的时候，有很多地方会觉得“这么麻烦，用 C#，一下子就 OK 了”。

所以，设计模式是思想性的东西，但需要在语言上多做些准备，学习的时候要敢于否定自己以前已经很熟悉的套路，甚至是经典中提出的套路，自己给自己一个不断突破的目标，经过批评和自我批评之后再看《设计模式：可复用面向对象软件的基础》，应该就有更多心得和收获了。

至于难点，我师傅也有句话——“重复的力量”，看不清楚就敲击键盘调试它，几遍之后会突然有一个顿悟的时刻。本书概念上最难的是 Visitor 模式，前面在 Observer 和 Proxy 也有个跳跃式的过程，您也可以用“重复的力量”击溃它。

博文小林

您在全国海关信息中心从事过很多大型的开发项目，您日常负责的工作主要是哪些？本书重视设计模式的工程化实现和扩展，这些工程化的内容与您的工作有哪些联系？在书中您是如何体现工程化思想的？

王　翔

我的工作职责主要有三个部分：

1. 作为开发部门的高级技术架构师，主要负责行业业务系统的开发。
2. 作为信息安全工作组的成员，参与公钥项目的实施和扩展。
3. 作为优化小组的成员，参与行业主要业务系统的优化，务求 Do more with less。

本书涉及的内容基本上都是从这几年工作经历中抽象、简化出来的，虽然书名有

“工程化”，但其实更偏重“工程化实施的功能特性”，因为容错、保密、异常处理、高可用、锁和并发管理基本上没有体现在示例中。

不过本书中也提到了如何透明地加入这些机制，并且还介绍了可以即插即用的模式，希望您不要错过。

我工作的行业是一个快速变化的行业，最近几年每年的增长率都在 20%以上，同时又随着全球化、区域化而快速变化。因为业务上的变化要求，技术上必须做出相应的处理，因此发现《设计模式：可复用面向对象软件的基础》后，我就再也没有放弃过它。

这几年行业的标准化程度有了质的跨越，外部环境中 Web 2.0、SOA、云计算和纳米计算的概念也在兴起之中，所以书中很多地方提到的 XML 方式，既是顺应需要，又是不得已为之。希望其中一些粗浅的方法也能用在别的行业中。

博文小林

在经典的 23 种设计模式之外，您在书中还扩展了一些架构模式如 Web Service 模式，介绍这些并不属于经典模式的模式，您是出于哪方面的考虑？在深入这些模式之前需要有哪些必要的知识准备？

王　翔

项目需要，估计别人的项目中也已经越来越多地涉及了这些内容。

完成一个项目，不同类型的模式可能要兼容并包，“尺有所短，寸有所长”。

我有一点体会，不管什么类型的模式，哪怕它叫“架构”，都应该有个“作业面”（借鉴地质、能源行业常用的词），每次经过抽象和简化，不管什么模式其实面对的都是一个相对有限的小场景，头脑中保存的不是按层次分类的模式，而是一个长长的列表，然后根据上下文选择合适的，不要受架构层次一定要用所谓的架构模式、类层次一定要用《设计模式：可复用面向对象软件的基础》中那些对象关系的羁绊，基于组件化、服务化，我们已经可以较容易地拿捏出这个“作业面”。

至于这些扩展部分，准备知识恐怕还是经典的 GOF23 和相关领域的开发体会。

另外，比较遗憾，没能在本书中把数据访问模式、集成模式、公钥体系中的信息安全模式、XML 设计模式和数据库设计模式涵盖进来，如果有幸再版，我希望可以把它们补充进来。

推荐序 1

如果你要开发一个小型的系统，整个系统只有两三人，系统活不过五年，商业逻辑单纯，程序代码不超过万行，那么你随便做，影响不大。反之，若要架构大型系统，你需要慎思，套用模型与架构，将前人的经验当作基石，这样系统设计才不至于陷入发散。

然而，你不会一开始就做大系统，那样风险太高。因此，要练习，最好从小系统开始使用模型与架构，这样才能检讨与学习，日后方能在大系统中自如运用。

企业信息系统现今面临着大量的整合需求，需要提供深入的分析应用和灵活的应变流程。但系统整合的复杂度是彼此系统复杂度的乘积，系统间的安全、弹性、效率、扩充性、可用性……彼此互相掣肘，此时，企业需要接触广、想得深、能定方向的架构师。而熟悉设计模式是架构师的养成基础，要求对于问题的分类与解法有一定的认知。

有经验的设计者们，抽象出系统开发的原则与标准问题的设计解法，而 GOF 于十几年前提出的 23 种模式是其中的佼佼者。但毕竟空有概念，仍难落实到你日常使用的程序语言中来。坊间许多图书作者利用不同的程序语言，例如 C++、Visual Basic、.NET、Java 等，实现 GOF 的 23 种模式，配合 UML 的模型说明，让你可以方便地应用在自己的开发环境中。

本书的作者王翔有多年的开发经验，参与过多个千万乃至十亿行代码的大工程，他将经验融于设计模式中，以 C#重新实现了 GOF 的模式，同时加入了新近的设计想法，如 SOA 与 Web Services 等，以及相对于其他设计模式而言较新的.NET Framework 实现技术，如泛型、3.0 的 WCF 等。在本书中他除了正向地以 C#展现多个不同用途的模式外，还提供了日后可重复验证与测试的单元测试码。

系统分析与设计是门艺术，问题的解法与何为问题是交织的，而各模式的搭配使用技巧不同，要领存乎一心，须要巧思与反复琢磨，方可有好的解法。本书立意明确，除了告诉你问题的类型与解法，还提供了可以立即演绎的程序代码。相信这本案头的工具书可以提供你一个不错的思维模式，帮你造就有弹性、能扩充、易维护的软件实体。

须要提醒你的是，抽象化的思考、封装与重用的设计神髓在心中，而不是落在纸上的程序代码，阅读此书时，不要停止在仅 Copy and Paste 程序代码。

微软 MVP，台湾恒逸资讯资深讲师，数据库铁人　胡百敬

推荐序 2

且看《笑傲江湖》中风清扬的独孤九剑：有进无退，招招都是进攻，使攻敌不得不守。虽只一剑九式，却是变化无穷，学到后来，前后式融会贯通，更是威力大增。能料到他要出甚么招，反招却抢在他头里。敌人手还没提起，你长剑已指向他的要害，他再快也没你快。“料敌机先”和“活学活用”这八个字，正是这套剑法的精要所在。

设计模式可以当做软件开发中的独孤九剑。在软件设计中最大的敌人就是需求不断在变化，需求变化无休无止，软件交付日期也就无限期地延迟。我们无法做到以不变应万变，但如果能提前预见到一些变化，就能用很小的代价来应对剧烈的变化。GOF 总结的经典设计模式虽只有 23 种，但不管是创建型模式、结构型模式还是行为型模式，归根结底都是在寻找软件中可能的变化，并封装这些变化。“料敌机先”为设计模式精髓之一，只不过这里的敌是需求的变化而已。

预测到了变化，我们需要运用抽象的手段对其进行封装，如何抽象、如何封装需要具体问题具体分析，不能一概而论，从重构到模式是目前使用设计模式最好的方式。对于设计模式如果不能够做到灵活自如地运用，不仅威力大减，甚至于弄巧成拙，“活学活用”为设计模式精髓之二。

然而，仅仅学会了剑法永远也无法达到武功的最高境界，正如学会了设计模式也无法登上软件设计领域之巅，要经过大量的实战才行，在实战中提高剑法，在实战中体会如何“料敌机先”，如何“活学活用”。《设计模式——基于 C#的工程化实现及扩展》正是这样一本教你进行设计模式实战的好书，作者从 GOF 23 种经典设计模式开始，带你走进模式的大门，小到细粒度的基础模式，大到粗粒度的架构模式，本书都做了详尽的讲解。如果您还在为了软件需求的无尽变化而烦恼不断，为了在软件设计领域更上一层楼而苦苦思索，希望本书能带给您一些启发。

最后，特别感谢王翔为大家带来了这样一本设计模式的经典之作。

微软 MVP，博客园专家，IT168 专栏作者　李会军

序　言

我有幸 5 岁开始学习编程。成为一名软件开发人员是从初中开始确立的一个目标。大学毕业后因为工作的关系，开始使用 Visual C++、Visual Basic 进行开发，并在师傅的教导下学习 C#。初识 C#的时候我总会将它和之前接触的语言进行类比，而且一直用 C#以很生硬的方式完成工作任务。不过随着行业的快速发展，来自各方面的变化往往在项目中期就不期而至，在很被动地完成几个项目后，我开始寻找尽可能灵活应付这些问题的方法，Enterprise Library、《Design Patterns: Elements of Reusable Object-Oriented Software》和《Patterns of Enterprise Application Architecture》是对我帮助最大的三项资料。通过对它们的了解，我发现 C#中充满灵性的内容——托管环境下的自由、灵活，我变得酷爱 C#。

模式是一个非常有趣的话题，它是对特定前提下重复出现问题的一个普遍解答，它是一种思想，使用得当也会对设计、实施提供帮助，从这个角度看它又是实实在在的生产率。最近几年单位用.NET 开发的项目越来越多，规模也越来越大，自己经常感觉到需要把一些内容记录下来，并在与同事、同行分享的过程中修正、提炼它们，这也是写本书的主要动力。区别于其他类似的图书，本书强调面向工程化处理，偏重具体实现，同时结合越来越普遍使用的 XML 技术及.NET 3.0+的技术进行了扩展和完善。在完成 GOF4 的 23 种模式后，我感觉到仅仅用这些“搭”项目是不够的，因此继续把一些架构模式、Web 服务模式、成例补充进去。不过比较遗憾的是因为时间的关系，忽略了数据库模式、数据访问模式、集成模式。

您可以直接用示例代码做练习，关于本书的示例代码您可通过以下链接免费下载：*http://bv.csdn.net/resource/sjms.rar*。为了便于了解每个知识点，建议您使用 TestDriven.Net 逐个运行相关章节示例代码的单元测试，涉及数据库访问的时候，您还需要用到微软的 Northwind 官方示例数据库。

希望本书能对您的开发有所帮助，当面对各种“不可抗拒”的变化时，您可以从中获得一些启发，能够简洁并直接地应对它们。另外，希望有机会与您就模式和 C#语言进行沟通和讨论，书中存在的问题和错误也请您不吝指正。

高级架构师　王　翔

电子书 目 录

导读 I

第 1 篇 预备知识——发掘用 C#语言进行面向对象化设计的潜力 1

第 1 章 重新研读 C#语言 3

1.1 说明 4

1.2 C# 部分语法内容扩展 4

1.2.1 命名空间（Namespace） 4

1.2.2 简洁的异步通知机制——委托（Delegate） 6

1.2.3 考验你的算法抽象能力——泛型（Generics） 12

1.2.4 用作标签的方式扩展对象特性——属性（Attribute） 18

1.2.5 用索引器简化的 C#类型信息访问 22

1.2.6 融入 C#语言的迭代机制——迭代器（Iterator） 29

1.3 可重载运算符（Overloadable Operators）与转换运算符（Conversion Operators） 35

1.3.1 The Day After Someday 35

1.3.2 用于有限的状态迭代 35

1.3.3 操作集合 36

1.3.4 类型的适配 37

1.3.5 小结 38

1.4 面向插件架构和现场部署的配置系统设计 39

1.4.1 认识.NET Framework 提供的主要配置实体类 40

1.4.2 应用实例 41

1.4.3 小结 48

1.5 实现依赖注入 48

1.5.1 背景介绍 48

1.5.2 示例情景 48

1.5.3 Constructor 注入 51

1.5.4 Setter 注入 51

1.5.5 接口注入 52

1.5.6 基于 Attribute 实现注入——Attributer 53

1.5.7 小结.....55
第 2 篇 创建型模式——管理并隔离对象实例的构造过程.....73
第 3 章 工厂&工厂方法模式.....75
3.1 简单工厂.....76
3.1.1 最简单的工厂类.....76
3.1.2 根据规格加工产品——参数化工厂.....79
3.1.3 简单工厂的局限性.....80
3.2 经典回顾.....80
3.3 解耦 Concrete Factory 与客户程序.....82
3.4 基于配置文件的 Factory.....83
3.5 批量工厂.....85
3.5.1 开发情景.....85
3.5.2 定义产品类型容器.....87
3.5.3 定义批量工厂和产品类型容器.....88
3.5.4 增设生产指导顾问——Director.....89
3.5.5 由 Director 指导的客户程序.....90
3.6 基于类型参数的 Generic Factory.....91
3.7 委托工厂类型.....94
3.8 小结.....95
第 4 章 单件模式.....97
4.1 经典回顾.....99
4.2 线程安全的 Singleton.....102
4.3 细节决定成败.....103
4.4 细颗粒度 Singleton.....105
4.4.1 背景讨论.....105
4.4.2 解决桌面应用中细颗粒度 Singleton 问题.....105
4.4.3 解决 Web 应用中细颗粒度 Singleton 问题.....107
4.4.4 更通用的细颗粒度 Singleton.....108
4.5 自动更新的 Singleton.....110
4.6 参数化的 Singleton.....110
4.7 跨进程的 Singleton.....112
4.8 Singleton 的扩展——Singleton-N.....114
4.8.1 定义具有执行状态的抽象对象.....115
4.8.2 定义相应的 Singleton-N 实例集合.....115
4.8.3 在基本 Singleton 模式实现的框架下引入实例集合.....116
4.9 引入配置文件管理 Singleton.....117
4.10 基于类型参数的 Generic Singleton.....118

4.11 由工厂类型协助 Singleton 实例管理.....119
4.12 小结.....120
第 3 篇 结构型模式——针对变化组织灵活的对象体系.....181
第 8 章 适配器模式.....183
8.1 说明.....184
8.2 经典回顾.....185
8.3 进一步扩展适配范围的组适配器.....187
8.4 Adapter——Adapter 互联模式.....190
8.4.1 分析.....190
8.4.2 方式 1：客户程序直接调度 Adapter.....192
8.4.3 方式 2：基于标准约定调度 Adapter.....192
8.4.4 方式 3：借助反射和约定完成异步调用.....193
8.5 用配置约定适配过程.....194
8.6 XML 数据的专用适配机制.....196
8.7 小结.....198
第 11 章 装饰模式.....225
11.1 说明.....226
11.2 经典回顾.....226
11.3 具有自我更新特征的装饰模式.....230
11.3.1 分析.....230
11.3.2 抽象装饰接口.....231
11.3.3 抽象状态接口.....232
11.3.4 依据当前状态修改装饰.....233
11.3.5 测试验证.....233
11.4 设计 Decorator 与 Builder 协作的产物.....234
11.5 把 Decorator 做成标签.....236
11.5.1 更“彻底”的 Attribute 注入.....236
11.5.2 方式 1：采用.NET 平台自带的 AOP 机制实现.....237
11.5.3 方式 2：自定义代理拦截框架方式.....239
11.5.4 进一步分析.....242
11.6 小结.....243
第 12 章 外观模式.....245
12.1 说明.....246
12.2 经典回顾.....246
12.3 Facade 接口.....249
12.4 Remote Facade.....250
12.5 面向性能考虑的升级版 Remote Facade——Data Transfer Object 模式.....253

12.6 平台、开发语言无关的抽象 Facade 接口——WSDL 256
12.7 让使用者更加方便的 Fluent Interface 设计 257
12.8 小结 258
第 4 篇 行为型模式——算法、控制流和通信关系的对象化处理 289
第 17 章 解释器模式 315
17.1 说明 316
17.2 经典回顾 317
17.3 采用正则表达式 320
17.4 采用字典 324
17.5 采用 XSD 328
17.6 用 XSD 解释定制的业务语言 329
17.7 小结 330
第 22 章 观察者模式 383
22.1 说明 384
22.2 经典回顾 386
22.3 .NET 内置的 Observer 机制——事件 391
22.4 具有 Observer 的集合类型 392
22.5 面向服务接口的 Observer 394
22.6 小结 397
第 5 篇 小颗粒度基础模式和应用案例——服务于细节的基础性模式 431
第 6 篇 部分架构模式——面向应用全局的模式化处理 483
第 30 章 出版—预订模式 511
30.1 说明 512
30.2 生活中无处不在的“预订” 512
30.2.1 面向单一主题的本地观察者模式 512
30.2.2 增加 Proxy 实现面向单一主题的分布式观察者模式 513
30.2.3 用出版者集中管理预订 514
30.2.4 面向物理环境设计更多出版预订模式 517
30.3 示例 518
30.3.1 数据实体模型部分 518
30.3.2 业务实体模型部分 521
30.3.3 具体实体对象部分 523
30.3.4 单元测试 527
30.4 小结 528
第 7 篇 部分 Web 和 Web Service 模式——面向服务开发中的模式化处理 549
第 37 章 Web 服务事件监控器模式 605

37.1 说明 606
37.2 如何为普通 Web Service 封装事件机制 607
37.3 示例 609
37.4 小结 613
附录 A 面向关系数据和 XML 数据的领域逻辑模式 627
A.1 说明 628
A.2 实现业务领域逻辑的主要方法 628
A.2.1 整体逻辑结构 628
A.2.2 性能改进 629
A.2.3 面向关系数据库的业务服务设计 630
A.2.4 面向 XML 数据的扩展设计 630
A.2.5 配置机制设计 631
A.3 示例 632
A.3.1 示例情景 632
A.3.2 测试内容准备 633
A.3.3 实际测试过程 634
A.4 小结 635

关于本书更详细的目录请访问：http://bv.csdn.net/resource/sjmsml.doc

作者简介：

王翔，软件架构师，主要从事.NET、XML、公钥基础设施的开发。专注于数据（尤其是 XML 信息）的生产、加工、交换、提炼等过程。最近参与了一系列有关应用密码技术和 PKI 环境保护信息系统数据安全的项目。

最喜欢数学，平常案头总是摆一本数学练习题。闲暇时间喜欢写作，通过发表多种技术文章与国内外同行交流各种数据应用经验。

项目间隙经常到各海滨城市徒步旅行、野外露营、出海航行、极限运动，这几年烹饪也渐渐成为个人主要爱好。

博客园博客：http://www.cnblogs.com/callwangxiang/

导　读

本书的应用背景

面向对象设计模式，也就是本书简称的“设计模式”，是软件设计过程中，通过面向对象的方法对相近似的问题，在指定上下文范围内给予的指导性解决方案。模式的主要价值在于它们是以往经验的浓缩，尤其在我们建立复杂系统的时候，借鉴和采用模式可以让我们少走弯路，其设计比较灵活并具有不错的扩展性。

和 15 年前相比，现在的开发工作更强调对于业务变化的适应性。虽然有各种方法学帮助我们以尽量小的代价适应这些变化，但相信没有多少人愿意对一个已经基本完成的系统进行结构性设计修改，即便修改也希望尽可能地集中在一个点上。但现实的情况总和我们作对：业务实体这个本应该相对稳定的对象，在信息化快速建设过程中被赋予了多变的特质；业务流程和操作功能总是进行着“家常”式的变化；为了适应更广泛的服务对象，我们开发的产品需要不断集成更多的第三方产品、需要支持更多的 IT 产品；最后，还有会更加多变的未来。技术上虽然各种开发方法、架构技术都试图让 80%的开发人员仅仅关注业务逻辑的实现，但很多情况下这都是美好的愿景。简言之，开发人员处在一个夹缝之中，如果不尽量让自己的设计更具弹性，则很容易让本已经满负荷的工作不断加码。

对于准一次性代码而言，应用设计模式常常会成为负担；但对于公共库、公共平台、领域通用软件而言，合理使用设计模式则是避免“坏”设计的一个有效途径：

- 避免僵化。
- 增加重用性。
- 以适度的复杂性应对可预见的变化。

因此，如果您要“坚守”某一块代码，而且该代码总是受到打压不得不适应各种变化的话，您可以在抽象的基础上发掘变化的诱因，如果需要，参照贴近的模式设计并实现。那么为什么基于.NET Framework 呢？因为.NET Framework 已经被 MySpace 等众多成功的电子商务站点所验证，它完全可以支撑大型应用运行维护；至于选择 C#而不是 Java 和 C++，也许是因为 C#语言更优美吧。

谁应该读本书

本书的目标读者是对 C#语言和.NET Framework 平台有一定了解或应用经验的用户，尤其适于那些希望能基于模式技术在设计和开发方面应对更多挑战的用户。对于具有多年项目经验的架构师而言，本书的内容可能有些肤浅，不过您可以把本书当成一个小备忘录。

本书具体读者对象如下：

- 初学设计模式的读者

本书对于您可能有一个小的跨度，不过这并不影响您使用本书。您可以在对 C#语言语法了解的基础上，在第 1 章中补充一些 C#面向对象开发的高级知识，第 1 章中的依赖注入部分则可以先跳过。第 2 章还会为您补充一些具体的开发专题，您可以结合自己项目的需要做一套类似的小工具类型。

接着，创建型模式、结构型模式可以作为您树立并强化设计模式思想的途径。

- 中级.NET 开发工程师

您已经具有了 C#开发的经验，这时候不妨浏览一下第 1 章的内容，因为它除了进一步强化 C#语言对象化特性外，还有一些规范性、扩展性的内容。然后您可以继续学习 GOF23。建议您在学习 GOF23 后，不妨在第 26、27 章稍作停顿，回顾并对之前的了解做汇总和检查，最好结合项目中一个比较核心的类库（例如：数据访问、报文交接、通知机制……）设计一个自己的 Show case，环境可以考虑得复杂些，通过三四个迭代的开发，从中体会出 GOF23 种主要模式的适用环境。

接着，您可以根据工作的需要，继续学习后面的架构、Web Service 模式。

- 高级.NET 开发工程师

您已经在.NET 平台完成了一批规模化的项目，这时候您可以浏览本书前面的部分，不过在学习后面的架构模式和 Web Service 模式前，您不妨在第 1 章（依赖注入部分）、第 4 章、第 11 章、第 12 章、第 14 章、第 17 章、第 22 章、第 24 章和第 25 章的扩展部分稍微多花些时间，因为后续的内容很大程度上与这些章节有关系，而且它们通常在开发技巧上进行了提炼和延伸。

- .NET 平台架构师

也许您可以从附录的两章入手，其中一章介绍了领域逻辑的实现技巧（RDBMS / XML DB），另一章介绍了 XML 应用建模。推荐这两章的目的主要是技术平台，尤其是企业信息本身的变化可能需要我们思路上有些变化。至于前面 GOF23 的内容，您可能已经烂熟于心，可以先跳过。不过如果有时间，您不妨对第 11 章、第 14 章、第 17 章和第 22 章中的扩展部分稍微留意一下。接下来，您就可以直接学习后面的架构模式、Web Service 模式了。

阅读本书需要的基础知识

Example 就够了么

设计模式是一种设计思想，表达这种思想最简洁的方式就是类图，至于说明其实现上的技巧，Example 就够了。但 Demo 和实际工程应用还是有一段差距的，原因不多，但每一个都需要在 Example 之余好好考虑：

- 数据类型是 int、string，还是 DateTime？或者是<T>？
- 能支撑负载要求的并发么？
- 是不是应该用 Delegate 解决常规的异步调用？
- 如何进行运行维护？哪些允许被配置？
- 静态类、匿名方法，还有基于 Attribute 的开发是不是也可以用于实现设计模式呢？

此外，设计模式的一个亮点就是提高代码的可重用性，如果设计一套比较适合实际工程使用的设计模式库，可以重复八股式反复 Example 的工作。

设计原则

作为面向对象基本设计原则的忠实体现，设计模式帮助我们在学习过程中不断强化以下五项原则性设计要求。

- 单一职责原则（SRP）：一个类应该有且仅有一个引起变化的因素。
- 开放封闭原则（OCP）：对扩展开放，对修改封闭。
- Liskov 替换原则（LSP）：子类可以替换为它的基类。
- 依赖倒置原则（DIP）：高层模块不应该依赖于低层模块，二者都应该依赖于抽象。抽象不应该依赖于细节。细节应该依赖于抽象。
- 接口隔离原则（ISP）：一个类对另外一个类的依赖建立在最小的接口上。

而在工程上，SRP 和 ISP 常常被不经意破坏。因为设计时变化因素还没有被真正识别，因此最初设计的接口从最终的实现看，本身是可以分解的；另一个考虑就是为了开发“省事”，参数固定为某个接口，即使以后该接口被丰富了，也不需要修改下游代码。本书由于是专门描述每个设计模式工程化实现的，不涉及上层具体业务处理，因此设计上严格贯彻这五项原则，Test Project 设计也依据这五项原则展开。

约定

本书示例全部采用 VSTS 2005 编写，运行平台为.NET Framework 3.0，对于模式的每个实现的测试均采用 VSTS Test Project 的 Unit Test 方式编写。由于 VSTS 2005 自带的 Class

Designer 对 UML 的展示相对很有限，因此笔者选择用 IBM Rational Rose 或 Sparx 的 Enterprise Architect 绘制 UML。本书假设了一个叫“MarvellousWorks”的公司，并按照层次关系设定所有实例的根命名空间为 MarvellousWorks.PracticalPattern。此外，示例编码上区别于其他设计模式图书有如下不同。

- 命名规范上依照的是《Design Guidelines for Developing Class Libraries》(.NET Framework 2.0 & 3.0，本书简称为 Design Guideline)，而不是 Java 命名法、C++匈牙利命名法。
- 笔者更倾向于用 Property，而不是一对 set / get 方法。
- 对于集合类型基于 key 的检索，笔者喜欢用 Indexer 而不是类似 GetValue(TKey)的方法。
- 为了减少使用者的编码量，笔者喜欢将修饰性的类设计为 Attribute，而不是单纯的 Interface。模式实现上一般也采用 Interface<T>到 Abstract Class<T>再到 Concrete Class 的方式，务求在保持抽象性的基础上尽量减少子类实现的编码量。
- 如果某些机制的定义已经在.NET Framework 中提供了，那么尽量用平台自己的。
- 与扩展无关的、与 Assembly 内部其他类调用无关的属性、方法等一律声明为 private。
- 此外，由于本书很多介绍都是基于代码的说明，为了尽量减少行文中代码的行数，许多接口声明、方法、属性的写法都采用了很不规范的单行书写方式，例如：

C#

```
using System;
namespace MarvellousWorks.PracticalPattern.Concept.Generics
{
   public interface ITarget { void Request();}
   public interface IAdaptee { void SpecifiedRequest();}

   public abstract class GenericAdapterBase<T> : ITarget
      where T : IAdaptee, new()
   {
      public virtual void Request()
      {
         new T().SpecifiedRequest();
      }
   }
}
```

如果您经常负责代码复查，并且对代码书写方式很敏感，笔者在此对您可能会感到的不畅致歉。

相对于其他介绍设计模式思想的图书，本书重点在于如何借助.NET Framework 2.0 & 3.0 平台和 C# 2.0 实现这些模式，并且根据笔者在工程中遇到的一些情况，介绍如何扩展实现体系。

本书如何组织

本书主要基于 C# 2.0 的语法（但在部分内容上采用.NET 3.0 扩展的部分新增语法），试

图将 GOF 23 种模式以一种可工程化的公共库而非 Example 的方式呈现给读者。内容划分为 8 个部分展开。

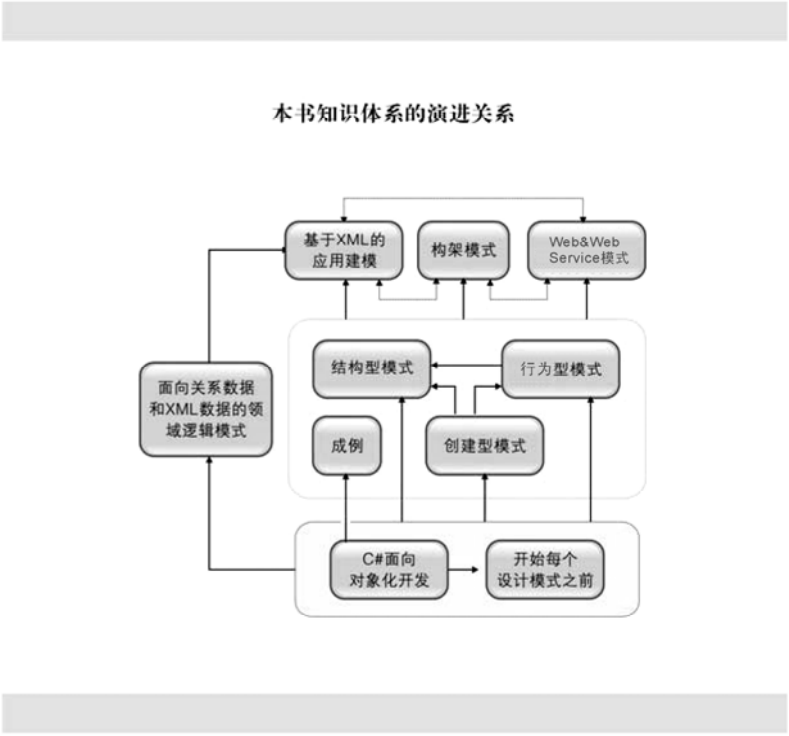


图 1

“C#面向对象化开发和扩展”是本书讨论的内容基础。图 1 中不同知识领域的技术层次以其在图中位置的高低表示，知识领域间的学习曲线关系以箭头方向而定（但部分章节内的细节内容可能有交叉）。

另外，XML 技术作为 SOA、Web 2.0 时代的主要数据形式、实体结构、处理逻辑、语义信息、领域语言载体，一直是本书所倡导和推荐的，因此“面向关系数据和 XML 数据的领域逻辑模式”及最终的“基于 XML 的应用建模”在“C#面向对象化开发和扩展”之上，贯穿全书模式介绍的始终。

第 1 篇　预备知识——发掘用 C#语言进行面向对象化设计的潜力

第 1 篇主要对本书进行一些概括性介绍。

第 1 章　重新研读 C#语言

虽然本书不是一本介绍 C#语言的书，但是由于工程中，特别是规模相对较大的工程中常常需要使用一些 C#语言特有的高级特性，这些特性的介绍又要结合面向对象开发基础之

上扩展的一些诸如“异步通知机制”、“配置—对象映射机制”、“操作符重载”等特性，所以专门增加了一章，介绍 C#语言这类开发的梗概内容。

第 2 章　开始每个设计模式之前

为了切合本书的副标题，如我们一般项目的开发习惯一样，这一章主要准备一些后面每个模式具体编码中需要使用的公共机制的实现。

第 2 篇　创建型模式——管理并隔离对象实例的构造过程

创建型模式为了隔离客户程序与具体类型实例化的依赖关系，通过将实例化职责委托他方对象的办法，保证客户程序（或外部系统）在获得期望具体类型实例的同时不必发生直接的引用。

第 3 章　工厂&工厂方法模式

工厂方法的意图非常明确，它把类的实例化过程延迟到子类，将 new()的工作交给工厂完成。同时，增加一个抽象的工厂定义，解决一系列具有统一通用工厂方法的实体工厂问题。在.NET 平台中，我们可以借助配置、泛型和委托的方法在实现经典模式目的的同时，获得工厂类型与客户程序间更加松散的构造过程。

第 4 章　单件模式

使用 Singleton 模式的意图就是要保证一个类只有一个实例，同时提供客户程序一个访问它的全局访问点。虽然经典 Singleton 模式旨在控制实例的数量，但受到种种“隐性”破坏因素的威胁，我们还需要实施各种针对性的实例数量控制手段。

第 5 章　抽象工厂模式

提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们的具体类型。借助类型映射器、配置、委托等手段都可以进一步扩展抽象工厂构造“一组相关或相互依赖对象接口”的能力。

第 6 章　创建者模式

创建者模式可以将一个产品的内部表象与产品的生成过程分割开来，从而使一个建造过程生成具有不同的内部表象的产品对象。根据上下文需要，我们除了要设计具有装配能力的创建者，必要的时候还要能够有序拆解目标实例。

第 7 章　原型模式

用原型实例指定创建对象的种类，并且通过拷贝（克隆）这些原型创建新的对象。克隆过程可“繁”可“简”，而且我们往往希望对于克隆过程实施更深入的控制，为了简化开发，我们还要准备点工具类完成不同的克隆过程。

第 3 篇　结构型模式——针对变化组织灵活的对象体系

结构型模式的重点在于如何通过灵活的体系组织不同的对象，并在此基础上完成更为复杂的类型（或类型系统），而参与组合的各类型之间始终保持尽量松散的结构关系。

第 8 章　适配器模式

适配器模式的意图就是通过接口转换，使本来不兼容的接口可以协作。一次适配也许不能满足我们的需要，项目中我们往往需要做适配器间的互联，随着 XML 数据的广泛使用，我们还需要专门的适配器设计思路。

第 9 章　桥模式

一直觉得在设计模式中，桥模式（或称之为桥接模式、桥梁模式）的意图表述是最普遍正确但又难于明确化的：“将抽象部分与它的实现部分分离，使它们都可以独立地变化”。通常我们应用中的类型关系会越来越复杂，一座桥也许满足不了我们的需要，那就不妨多搭几座，然后把它们连起来。

第 10 章　组合模式

组合模式需要解决的主要意图是把具有“整体—部分”关系的对象组织成层次结构，客户程序在使用的时候可以用同一套方法处理单个个体或容器对象。项目中，虽然实体对象和 XML 对象都具有“整体—部分”关系，但处理的手段却是各异的。

第 11 章　装饰模式

装饰模式的意图非常明确：动态地为对象增加新的职责。Decorator 与 Builder 的协作可以令对象构造过程更加丰富多彩，在.NET 平台上我们还可以用更简单的 Attribute 方式实现这些过程。

第 12 章　外观模式

外观模式的意图很明确：为子系统中的一组接口提供一个高层接口，该接口使子系统更易于使用。它的主要动机是减少“子系统”内部与外部间对象通信的依赖复杂程度。远程访问需要好的外观，Web Service 访问需要好的外观，就连编码我们也可以面向类型使用者提供好的外观。

第 13 章　享元模式

享元模式的意图是通过共享有效支持大量细颗粒度对象。管理并有效共享对象实例的使用是中、大型应用需要慎重设计的问题，基于享元模式正好提示我们一个对象池的好思路。

第 14 章　代理模式

通过一个代理对象，控制其他对象访问目标对象。为了让我们的逻辑更多关注于当前的处理，而把各种复杂的网络调用、数据访问、缓存等透明地提供给客户程序，代理类型需要多种多样的扩展。

第 4 篇　行为型模式——算法、控制流和通信关系的对象化处理

行为型模式关注于应用运行过程中算法的提供和通信关系的梳理。

第 15 章　职责链模式

职责链模式用于对目标对象施与一系列操作的情况，为了避免调用双方和操作之间的耦

合关系，可以把这些操作组成一个链表，通过遍历链表找到适合处理的操作。链式处理帮助我们把加工过程联动起来，不过有时候这还不够，还需要在我们不直接调用的情况下，把这个联动过程自动化，并覆盖到更大范围之中。

第 16 章　模板方法模式

模板方法模式是面向对象系统中非常朴实的一种模式，体现出面向对象设计中继承和多态的基本特征。由于设计的需要，我们往往会在初期规划一些较粗颗粒度的算法，而且对参与计算的对象进行抽象，明确算法会使用到哪些方法，每个方法可以提供哪些支持，但此时每个方法本身并没有细化；随着开发过程的展开，我们可能会具体实现每个方法，或者对最初的一些方法进行替换，覆盖上新的内容，这样就在一个相对固定的算法框架下，通过子类（或其他方法）的变化，实现了算法的差异性。本章扩展了类型的模板、方法的模板、程序执行框架的模板。

第 17 章　解释器模式

项目中我们常常会发现有些问题（业务或技术上的）出现得非常频繁，而且它们也基本上以一些规律性的方式进行变化。对于这类问题，如果编写一个对象类进行处理，随着业务变更，我们将频繁地修改代码、编译、部署。与其反复做这种工作，不如把它们抽象为一个语言，借助解释器在更高层次上以更灵活的方式适应变化。实现解释器的方法多种多样，对于不同的“特定领域语言”我们也能找到一些捷径。

第 18 章　命令模式

命令模式是对一类对象公共操作的抽象，它们具有相同的方法签名，所以具有类似操作的对象就可以被抽象出来，成为一个抽象的“命令”对象。这样实际操作的调用者就不是和一组对象打交道，它只需要依赖于这个“命令”对象的方法签名，并根据这个操作签名调用相关的方法。把处理抽象为具体的命令对象还不够，要根据不同的应用环境，我们还需要赋予它异步、打包、队列的支持。

第 19 章　迭代器模式

迭代器模式要求我们从使用者的角度考虑如何设计对外访问内部对象接口。即便我们组织的对象系统内部结构很复杂，但对于客户程序而言最简单的方式莫过于通过 for / foreach 循环依次遍历，至于遍历过程中的次序、分类筛选等则由目标类型自己封装。而 C#语言中就有现成的迭代器。

第 20 章　中介者模式

中介者模式是在“依赖倒置”原则基础上进一步扩展出来的，它主要解决 M:N 依赖关系。松散协调 1:N 的通知本来就不容易，为了在 Web Service 时代“规规矩矩”地完成这个过程，我们还要采用专用的手段。

第 21 章　备忘录模式

备忘录模式的意图是在保证对象封装性的前提下，获取对象内部的状态，并交给第三方

对象保存起来，后续运行中根据需要可以把对象恢复成之前的某个状态。此外，还须妥善地保存备忘信息及定义备忘信息的获取手段。

第 22 章　观察者模式

观察者模式主要用于 1:N 的通知发布机制，它希望解决一个对象状态变化时可以及时告知相关依赖对象的问题，令它们也可以做出响应。.NET 有内置的观察者机制，我们除了可完成个体对象间的异步通知，还可以针对集合类型、服务接口扩展不同的观察者。

第 23 章　状态模式

状态模式的意图就是引入独立的状态管理类型，由后者负责在对象类型状态变化的时候相应改变对象的行为，这样从外部看上去对象的执行逻辑就好像被修改了一样。状态模式可以主动更迭，可以与用户进行交互，也可以扩展为复杂的工作流机制。

第 24 章　策略模式

策略模式的意图很明确：定义一系列算法，把它们封装起来，确保它们可以根据需要相互替换。不过，好策略要实施有方。

第 25 章　访问者模式

访问者模式的意图描述比较复杂：对于一系列元素，在不改变各元素的前提下增加新的功能（操作）。正所谓“外来的和尚好念经”，当类型间产生双依赖关系的时候，我们就用访问者来解决。

第 5 篇　小颗粒度基础模式和应用案例——服务于细节的基础性模式

相对前面介绍的创建型、结构型、行为型模式，我们往往在项目中局部还要应用一些非常小的基础模式（或被称为 idiom），它们虽然在实现上，很多时候技巧性不如 GOF 模式，但经常会令我们的代码使用起来更“舒服”，构造的数据结构、算法结构也更加清晰，因此专门选择其中几个贴近.NET 类型且项目中常常会用到的“小”模式进行介绍。

第 26 章　成例

也许有些模式化的处理很“小”，但它们很实用。

第 27 章　GOF 总结及应用案例

完成了 GOF23 和一系列“小”模式的介绍后，为了加深对模式应用技巧的认识，本部分通过一个案例——数据源无关的通用数据访问机制，介绍如何在项目中具体应用上述模式。虽然很多内容看上去有些做作，但具体项目中您可以根据需要和相关位置的变化进行裁减，而不像这个案例中“为了模式而模式”。

第 6 篇　部分架构模式——面向应用全局的模式化处理

除 了上面介绍的 GOF，项目中我们还往往会用到一些涉及应用全局的模式化的实现方法。现在已经被普遍应用的 N 层模式及某些关键性框架产品采用的“微内核”模式，由于其概念性很强且已被绝大部分开发人员所了解，但实现技巧不一而足，所以并没有被纳入本

部分。本章选择部分有代表性的模式，介绍它们的模式意图和具体应用情景，并介绍如何基于.NET 和 C#实现它们。

第 28 章　MVC 模式

用 MVC 分解应用中数据、控制和展现的关系。

第 29 章　管道—过滤器模式

当需要一个线性的连续处理时，我们不妨把消息和数据投入到一个管子里——筛选它。

第 30 章　出版—预订模式

“预订”是生活中常常遇到的情况，针对不同预订要求、不同执行环境，我们需要灵活设计。

第 31 章　Broker 模式

用 Broker 梳理 M:N 个应用的交互。

第 32 章　消息总线模式

面对交织在一起的企业应用环境，我们为它们准备一个总线，需要交互的都通过总线完成。

第 7 篇　部分 Web 和 Web Service 模式——面向服务开发中的模式化处理

前面我们的讨论一般都集中在本地应用的范畴内，随着 SOA 和 Web 2.0 概念逐步被业界所推崇，Web 和 Web Service 应用也日趋普及。面对新挑战，我们可以用一些专门的模式化方法应对。本部分是一些针对该领域的模式设计技术。

第 33 章　页面控制器模式

用对象化的方法，把页面中的处理抽象为独立的控制对象。

第 34 章　实现 Web 服务依赖倒置

要实现服务间的松散耦合，先实现服务接口间的依赖倒置。

第 35 章　Web 服务适配器模式

采用适配器完成服务接口间的适配。

第 36 章　Web 服务数据传输对象模式

采用数据传输对象缓解频繁服务调用中中间信息的缓冲。

第 37 章　Web 服务事件监控器模式

通过增加监控器，为本来无状态的服务调用赋予异步的 Observer 能力。

第 38 章　Web 服务拦截过滤器模式

通过横切手段完成服务调用过程中外部机制的透明注入。

附录 A　面向关系数据和 XML 数据的领域逻辑模式

在领域驱动设计（DDD：Domain Driven Design）中，实现业务逻辑层主要有三种模式

Transaction Script、Domain Module 和 Table Module。随着业务逻辑复杂程度的增加，采用各模式实现的工作量变化趋势有所不同；根据应用特点不同，三种模式也各有优势。应用建设初期选择的实现模式随着业务需求和历史数据量的变化可能需要进行调整，此时要增加一个适应性机制，保证在尽量不影响客户程序的前提下，选择合适的实现模式。随着 XML 数据使用日趋广泛，须借助 XPath、XQuery 和 XSL 为层次型数据增加专门的扩展机制，使得基于 XML 数据源的业务逻辑也可以采用上述三种模式实现。

附录 B　基于 XML 的应用建模

介绍如何采用一些 XML 技术服务应用于开发中的主要领域。

致谢

首先，要感谢工作单位的各位领导、同事及带我进入.NET 开发领域的师傅——沈嵘，在有机会经历这些规模化的.NET 项目之后，我才对.NET 开发有了一些了解和认识，并且可以在后续项目中不断强化和完善这些认识，进一步总结一些模式化的处置手段。

另外，博文视点的各位老师和编辑也多次给予我无私的帮助，尤其是周筠老师、徐定翔编辑和陈琼编辑。作为我的第一本书，最初的书稿存在很多不足，是博文视点的各位老师和编辑不厌其烦的指导和建议才让本书的内容更加“规矩”。还要感谢李会军、王涛、蒋波涛等朋友对本书的关心和支持，正是他们无私的帮助，才使得我可以纠正并调整书中很多不足之处。

由于笔者工作年限不长，对.NET 社区，尤其是.NET 开源社区了解有限，所以书中难免存在不足、错误，也请专家、读者不吝指正。

第 1 章

重新研读 C#语言

1.1 说明

1.2 C# 部分语法内容扩展

1.3 可重载运算符（Overloadable Operators）与转换运算符（Conversion Operators）

1.4 面向插件架构和现场部署的配置系统设计

1.5 实现依赖注入

1.1 说明

本章以工程化使用为目的，对 C#和.NET Framework 提供的几个平时开发时不引人注意的特征进行介绍，它们对于提高代码扩展性、灵活性很关键，务求用“很 C#”的方式解决以往设计模式 Example 之外必须面临的一些问题。它们主要包括：

- Namespace（命名空间）；
- Delegate（委托）；
- Generics（泛型）；
- Attribute（属性）；
- Indexer（索引器）；
- Iterator（迭代器）；
- Overloadable Operators（可重载运算符）与 Conversion Operators（转换运算符）；
- Configuration（对象化配置访问）；

由于降低类间耦合关系一直是设计上控制变化范围的常用手段之一，所以在设计模式之外补充有关用 C#实现“依赖注入”（Dependency Injection）的方法。

有关设计模式在 Threading（多线程）模型下实现需注意的内容，笔者将在相关章节的工程化分析部分介绍。

1.2 C# 部分语法内容扩展

1.2.1 命名空间（Namespace）

尴尬的现实状况

是否有很好的命名空间规划是工程化代码与非工程化代码一个很明显的区别。

尤其对于大型的组织而言，如果涉及的产品线、项目、公共平台很多，如何通过命名空间把所有的代码资源有效地组织起来，恐怕是实施项目前要考虑的主要问题。作为一个树形体系，最好有组织级统一的分类标准，目的很明确——用的时候能很容易找到。

做到这点并不容易，原因如下：

- 习惯中很难改变的缩写命名：CAD 很容易在声明的时候被命名为.***CAD，但事实上 Design Guideline 建议的是 Cad，这样使用者和定义者之间就存在一些小小的错位。在 99%的情况下这个问题不会发生，因为您只要一个点，IntelliSense 就帮您列出来了，另外的 1%则发生在后绑定调用的情况下。

- 不统一的命名：涉及加密的库，可能 A 被定义为 Encrypt，B 被定义为 Crypto 或 Cryptography。来自匈牙利命名法的“遗毒”也常常成为新旧开发人员统一命名的障碍，而且会直接影响到命名空间的定义。
- 组织或您上司的认同：技术总监不认为命名规范是他需要关心的事情，下面的架构师更关心的是结构，再下面的项目经理关注的是资源调度和进度，具体的实施人员恐怕没有多少机会规定其他同事该怎么命名，那么谁来关心命名空间呢？

企业.NET 类型系统的命名空间规划示例

无论如何，即便没有办法在组织级统一命名空间，为了您所带领的团队现在做的工作在以后能更容易地被应用，或者仅仅为你自己的职业生涯好好“储蓄”，在动笔编写第一行程序之前，先规划好命名空间吧。

可参考的建议来自 Design Guideline，示例代码如下：

Txt

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]
例如：Microsoft.WindowsMobile.DirectX.
```

本书选择了 MarvellousWorks.PracticalPattern 作为根命名空间，但前提是假设这个公司很小，相关的项目（或产品）很少，而且没有多少组织级通用的代码资源。如果假设它为一个大型软件企业，套用笔者自己组织的命名空间，总体命名空间的规划情况如下：

C#（一级命名空间）

```
namespace MarvellousWorks.Application
namespace MarvellousWorks.Foundation
namespace MarvellousWorks.Framework
namespace MarvellousWorks.Utility
namespace MarvellousWorks.Training
```

- Application：代表项目或产品。
- Foundation：代表公共库，类似 Enterprise Library 之类的公共基础库、基于企业设备和操作系统平台的通用的图形处理引擎等，但都是纯粹的 Class Library，没有 UI 元素。
- Framework：组织通用的框架，基于 Foundation 之上，面向某个开发领域扩充的 Class Library 和控件，其本身不能独立运行，但完全可以集成在具体的项目或产品中，比如通用的授权框架、完全 Ajax 化的前后端组件、报表和打印中间件。
- Utility：企业内部各种工具，比如现场故障排查工具、Dump 和日志分析工具。
- Training：完全面向培训用途，是对企业自身 Application、Foundation、Framework（甚至 Utility）使用的 Examples。

几个一级命名空间的布局如图 1-1 所示。

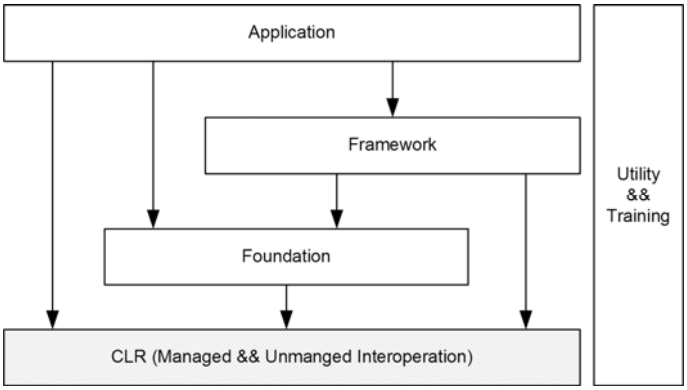


图 1-1 一个建议的一级命名空间布局和调用关系

本次的 PracticalPattern 作为实际工程化代码，不属于 MarvellousWorks.Training，而应划入 MarvellousWorks.Foundation，并将它作为最核心的算法框架，放入 MarvellousWorks.Foundation.Core.PracticalPattern。

小结

不论您最终如何定义命名空间，其实它体现的是您意志中对代码资源的规划，如果您觉得工程化的设计模式不是您希望开发团队要考虑的内容，没关系，您可以忽略这部分内容。

这样 MarvellousWorks.Foundation.Accessories.PracticalPattern

这样 MarvellousWorks.Foundation.Core.Accessories.PracticalPattern

或者这样 TestSolution. PracticalPattern

…… ……

1.2.2 简洁的异步通知机制——委托（Delegate）

用 Delegate 完成异步调用

Delegate 可用于对方法的引用，它可以作为方法的参数出现，也可以作为类的成员出现，尤其在异步调用中，Delegate 可以定义回调目标方法，示例代码如下：

C#

```
using System;
using System.Collections.Generic;
using System.Threading;
namespace MarvellousWorks.PracticalPattern.Concept.Delegating
{
    public class AsyncInvoker
    {
        // 记录异步执行的结果
        private IList<string> output = new List<string>();
```

```
        public AsyncInvoker()
        {
            Timer slowTimer = new Timer(new TimerCallback(OnTimerInterval),
               "slow", 2500, 2500);
            Timer fastTimer = new Timer(new TimerCallback(OnTimerInterval),
               "fast", 2000, 2000);
            output.Add("method");
        }

        private void OnTimerInterval(object state)
        {
            output.Add(state as string);
        }

        public IList<string> Output { get { return output; } }
    }
}
```

Unit Test

```
        [TestMethod()]
        public void Test()
        {
            AsyncInvoker asyncInvoker = new AsyncInvoker();
            System.Threading.Thread.Sleep(3000);
            Assert.AreEqual<string>("method", asyncInvoker.Output[0]);
            Assert.AreEqual<string>("fast", asyncInvoker.Output[1]);
            Assert.AreEqual<string>("slow", asyncInvoker.Output[2]);
        }
```

在上面的例子中，TimerCallback 就是一个 Delegate，它定义了每个 Timer 触发时需要回调的方法。由于它与主流程间是异步执行的，因此从测试结果看，主流程首先执行完成，而两个快慢 Timer 则先后执行。除此之外，上述事例还表达了一个非常重要的意图：Delegate 是对具体方法的抽象，它屏蔽了 Delegate 的调用者与实际执行方法间的关联关系。例如上例中调用者是 Timer，而执行方法是某个 AsyncInvoker 实例的 OnTimerInterval 方法。很多行为型模式可以采用 Delegate 这种抽象的操作方法表示。

对 n 的通知

进一步，通过 Delegate 集合可以实现一个对象与多个抽象方法的 1:1:n（调用者：Delegate 集合：抽象方法）的关系，如图 1-2 所示。

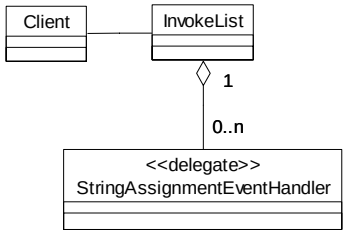


图 1-2 用 Delegate 集合实现客户程序与多个抽象方法调用的关联

其示例代码如下：

C#

```
using System;
using System.Collections.Generic;
namespace MarvellousWorks.PracticalPattern.Concept.Delegating
{
   public delegate void StringAssignmentEventHandler(); //抽象的操作方法

   public class InvokeList
   {
      private IList<StringAssignmentEventHandler> handlers;
      private string[] message = new string[3];

      public InvokeList()
      {
         // 绑定一组抽象方法
         handlers = new List<StringAssignmentEventHandler>();
         handlers.Add(AppendHello);
         handlers.Add(AppendComma);
         handlers.Add(AppendWorld);
      }

      public void Invoke()
      {
         foreach (StringAssignmentEventHandler handler in handlers)
            handler();
      }

      public string this[int index] { get { return message[index]; } }

      // 具体操作方法
      public void AppendHello() { message[0] = "hello"; }
      public void AppendComma() { message[1] = ","; }
      public void AppendWorld() { message[2] = "world"; }
   }
}
```

Unit Test

```
      [TestMethod()]
      public void Test()
      {
         string message = string.Empty;
         InvokeList list = new InvokeList();
         list.Invoke();
         Assert.AreEqual<string>("hello,world",list[0]+list[1]+list[2]);
      }
```

不过，上面的实现方式并不是“原汁原味”的.NET 做法，因为 delegate 声明的 Delegate 类型其实本身继承自 System.MulticastDelegate，从名字上不难发现它表示广播（见图 1-3），也就是它的调用列表中可以拥有多个委托，同时它重载了“+=”和“–=”运算符以便于使用，所以更为简单的写法如下：

C#

```
public MulticastDelegateInvoker()
{
    StringAssignmentEventHandler handler = null;
    handler += new StringAssignmentEventHandler(AppendHello);
    handler += new StringAssignmentEventHandler(AppendComma);
    handler += new StringAssignmentEventHandler(AppendWorld);
    handler.Invoke();
}
```

IL (StringAssignmentEventHandler)


```
.class public auto ansi sealed StringAssignmentEventHandler
   extends [mscorlib]System.MulticastDelegate{}
```

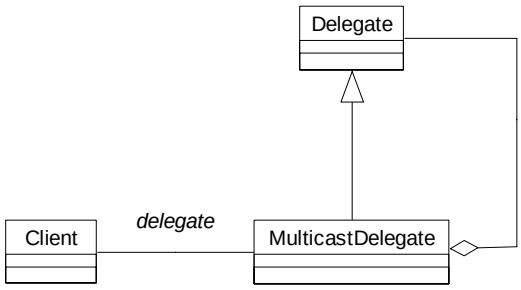


图 1-3 基于 MulticastDelegate 的广播结构

调用匿名方法

方法，也就是一小段可以重用的处理过程，一般我们都会把它独立编写出来，但如果这个逻辑本身非常简单，而且其变化范围仅仅是一个非常小的局部，或者如上面两个例子那样需要把抽象的处理作为一个参数传递给其他逻辑调用，那么匿名方法（Anonymous Method）是个很不错的选择。此外，匿名方法结合反射会大大提升公共库的灵活性。

从工程角度看，匿名方法无论在创建型模式、行为型模式还是结构型模式的实现中都大有用处，另外在架构模式中（包括 Pipeline、Page Controller、Gateway 等），匿名方法非常适合数量较大，但逻辑控制很短的调用。

笔者喜欢匿名方法的主要原因是用它写的代码看起来比较简洁，上例构造函数用匿名方法稍作修改后如下所示：

C#

```
public InvokeList()
{
    StringAssignmentEventHandler handler = null;
    handler += delegate { message[0] = "hello"; };
    handler += delegate { message[1] = ","; };
    handler += delegate { message[2] = "world"; };
    handler.Invoke();
}
```

调用重载方法

上例中 StringAssignmentEventHandler 定义了参数为空的方法，虽然在实际工程中最初定义模式时可以严格遵守一个特定的 Delegate 描述，但随着软件的升级，难免会要求新的定义，最明显的就是需要调用的方法有新的重载。虽然在经典的设计模式介绍中会列举出关键的方法名称，但似乎从未提示过如何处理重载的情况（如 Observer 的 Notify()方法）。为此，在实际工程中，笔者会采用以下几个办法：

- 声明 Delegate 参数为 params object[] parameters，这种方法虽然特别通用，甚至可以称之为 Smart Delegate，但怎么看都觉得别扭，原因是重载的方法参数类型不一定相同（值型、引用型都可能存在），因此只好不用强类型，也无法使用泛型。同时把 params object[] parameters 放到目标方法上，目标方法内部再根据参数信息自己选择该调用哪个方法反馈的 Delegate。这时整个系统内部仅须声明一个参数为 params object[] parameters 的“万金油”Delegate 也就够了，但这样会让每个目标类的实现非常 Ugly，如图 1-4 所示。

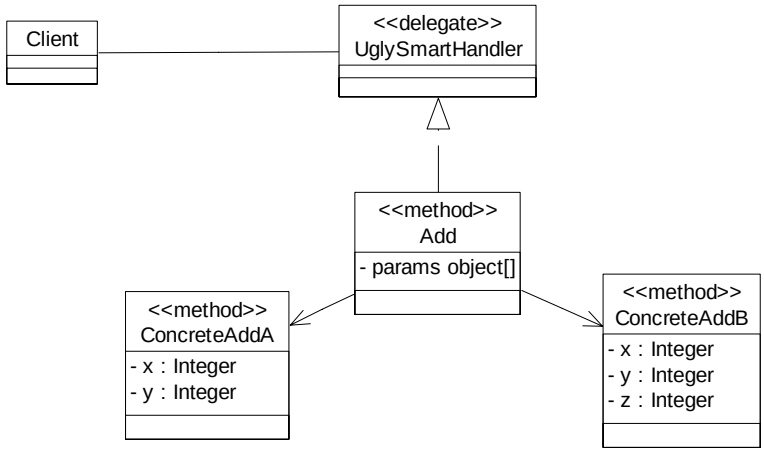


图 1-4 不怎么中看的智能委托实现

UML 中没有所谓的“方法图”，但为了表示逻辑组成的关系，笔者对方法采用了类似类图的表示。这里 Delegate 为纯虚方法定义，而具体目标方法被视为其实现类，对方法而言相应的属性代表方法的参数。

- 其次就是类似 ObjectBuilder 那种很重磅的实现方式，具体调用哪个目标方法全部由配置系统决定。
- 一般情况下，笔者喜欢用抽象类 System.Delegate 完成。例如图 1-5 中 C1、C2、C3 虽然分别对方法 A、S、M 都有重载，实际操作上下文仅须使用三个参数的重载方法，通过组合 System.Delegate 就可以实现该调用要求。

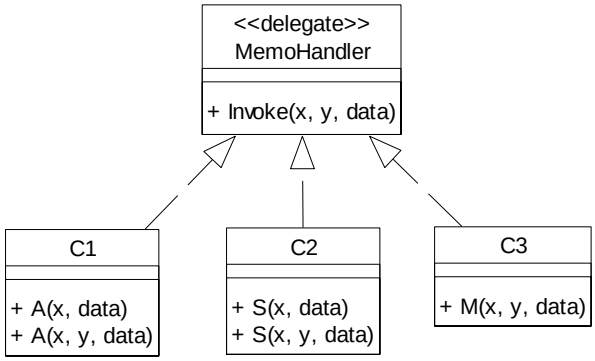


图 1-5 改进的智能委托

UglySmartDelegateInvoker.cs 和 UglySmartDelegateInvokerTest.cs 的示例代码如下：

C#

```
using System;
using System.Collections.Generic;
namespace MarvellousWorks.PracticalPattern.Concept.Delegating
{
   public delegate void MemoHandler(int x, int y,
     IDictionary<string, int> data);

   // 对具有重载的多个目标方法 Delegate
   public class OverloadableDelegateInvoker
   {
      private MemoHandler handler;

      public OverloadableDelegateInvoker()
      {
         Type type = typeof(MemoHandler);
         Delegate d = Delegate.CreateDelegate(type, new C1(), "A");
         d = Delegate.Combine(d, Delegate.CreateDelegate(type, new C2(),"S"));
         d = Delegate.Combine(d, Delegate.CreateDelegate(type, new C3(),"M"));
         handler = (MemoHandler)d;
      }

      public void Memo(int x, int y, IDictionary<string, int> data)
```

```
        { handler(x, y, data); }
    }
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    int result = 10;
    int expected = result;
    OverloadableDelegateInvoker invoker =
      new OverloadableDelegateInvoker();
    IDictionary<string, int> data = new Dictionary<string, int>();
    invoker.Memo(1, 2, data);
    Assert.AreEqual<int>(1 + 2, data["A"]);
    Assert.AreEqual<int>(1 - 2, data["S"]);
    Assert.AreEqual<int>(1 * 2, data["M"]);
}
```

小结

由于大量设计模式的书籍都采用了 Java 语言，即便冠名为 C#，从编码行文看实际上也采用了类 Java 的描述，相关的设计模式更倾向于用 Interface 而非 Delegate，因此在实际的.NET 工程中存在如下不便的地方：

- 如果将异步调用任务交给统一的线程池机制来维护，则需要增加额外开发量，不如直接使用.NET Framework 包装好的如 TimerCallback 之类的 Delegate。
- C#的 Delegate 自身就具有组合特征，可以通过组合加组播的方式，自动代理客户程序对多个目标方法的抽象调用。
- Interface 可以使用的抽象方法描述是固定的，而使用 Delegate，在运行过程中，可以根据动态要求自动适配目标方法。

1.2.3 考验你的算法抽象能力——泛型（Generics）

抽象及抽象能力支持

自第一次使用 STL 之后，笔者一直特别喜欢具有模板类的语言，主要原因有三个：

- 算法更抽象。
- 嵌套之后（例如 C<T<TKey, TValue>, V>），可以表示抽象之上的进一步抽象。
- 强参数类型无论在开发过程、编译过程、运行过程中都有优势。

除此之外，还有个人感觉上的原因——看 Generics 的代码很酷。

先谈抽象，也许已经有太多的技术资料讨论如何 Care 业务，没错，那始终是第一位的，不过作为开发人员，还要更多地顾及自己那始终在抚摸键盘的手指，以及办公桌上的时钟，如果一项任务能通过抽象这个途径减少一些开发量，何乐而不为呢？如果您是组织内部的平

台组成员，负责公共开发平台建设或通用行业中间件设计，那么抽象就不是 optional 而是 essential 的事情了，原因很简单——您大概只能知道别人会怎么重用您的工作，大概知道他们要什么样的内容，但您不能替他们做完一切。

那么，C#提供给了我们什么样的抽象能力支持呢？

- Class：提供了对现实世界的抽象。
- Interface：提供了对 Class 行为的抽象。
- Delegate：对方法的抽象。
- Attribute：对类型元数据的抽象。
- Generics：给上述因素进一步、进两步、直至进 n 步抽象的机会。

应用于模式

首先，各种创建型模式可以从中受益。Factory 可以加工出抽象类型来——Interface、Abstract Class，甚至在某个语境层面的父类 Class，但 Factory 自身的写法非常一致，如果为接口 1 写一个、接口 2 写一个……接口 100 也写一个，内心应该是个冲击——“我，我……我写了重复的代码”，修改的时候也一样。**当然，如果您的老板是按照代码行数计算绩效或发薪的话，另当别论**。在不使用配置系统的情况下，先考虑用 typeName 对具有无参构造函数的类提供一个相对通用的 Factory，这里的 typeName 采用的是可以被 System.Type 类识别的名称，并由 System.Activator 动态生成，例如：

- TopNamespace.SubNameSpace.ContainingClass, MyAssembly
- ContainingClass, MyAssembly, Version=1.3.0.0, PublicKeyToken=b17a5c561934e089

示例代码如下：

C#

```
using System;
namespace MarvellousWorks.PracticalPattern.Concept.Generics
{
    public static class RawGenericFactory
    {
        public static T Create<T>(string typeName)
        {
            return (T)Activator.CreateInstance(Type.GetType(typeName));
        }
    }
}
```

Unit Test

```
    [TestClass()]
    public class RawGenericFactoryTest
    {
        interface IProduct { }
        class ConcreteProduct : IProduct { }
```

```
        [TestMethod]
        public void Test()
        {
            string typeName = typeof(ConcreteProduct).AssemblyQualifiedName;
            IProduct product = RawGenericFactory.Create<IProduct>(typeName);
            Assert.IsNotNull(product);
            Assert.AreEqual<string>(typeName,
               product.GetType().AssemblyQualifiedName);
        }
    }
```

或者可以把 Factory 按照以往的方式，设计为非静态类。示例代码如下：

C#

```
public class RawGenericFactory<T>
{
    public T Create(string typeName)
    {
        return (T)Activator.CreateInstance(Type.GetType(typeName));
    }
}
```

Unit Test

```
IProduct product = new RawGenericFactory<IProduct>().Create(typeName);
```

类似的功能可以通过通用的 System.Object 类完成，但通过类型转换前置则是在保证 Factory 通用性的前提下，与客户程序间仅交付“规约”的产品——抽象类型的本意。相对而言，毕竟使用 Factory 的客户代码相对定义 Factory 自身的代码出现频率要高一些，把须要多次执行的类型转换成执行一次，下家使用的时候多少也轻松一点（或仅是一点点）。在实际工程中，typeName 一般都是根据调用上下文转译过来的，工厂方法一般借助配置访问部分获得实际需要生产的目标类型。

如果这个构造过程比较复杂，还要考虑用 Builder 模式来装配，经典的 Builder 模式使用通常的 C#实现方式，如图 1-6 所示。

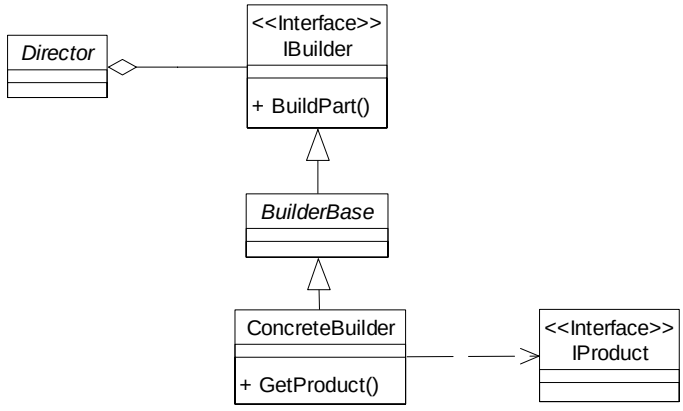


图 1-6 经典创建者模式的静态结构

但实际工程中除了 BuildPartA()、BuildPartB()之外，还需要增加下面几阶段处理，比如：

- Pre Constructor：整个 BuildUp 过程之前，获得每个 Part 的配置信息和配置中有关如何把 Part 装配为 IProduct 的说明信息，尤其是 BuildPart 的次序。
- Initialization：完成每个 Part 之后的装配过程。
- Post Initialization：装配之后是否需要考虑把实例池化、是否需要捆绑必要的监控属性（Instrument Attribute）、是否需要生成日志或做好串行化准备等事情。

如果把它们全部交给 Director 完成，那么 Director 就太辛苦了，不妨把每个阶段抽象为特殊的接口，每个阶段自身又可以由很多的抽象步骤完成，这样，Director 还是做“主持大局”的工作，而无须事事“亲历亲为”，保证 Builder 框架的相对稳定，如图 1-7 所示。它们将结合配置访问在 Builder 模式部分详细介绍其内容。

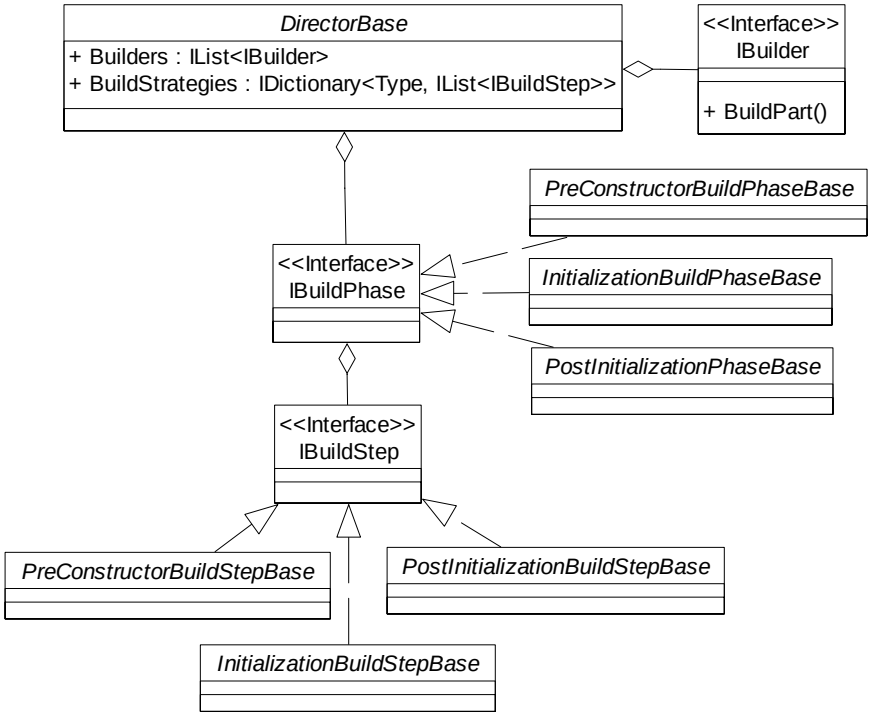


图 1-7 改进后的创建者模式的静态结构

这样，DirectorBase 仅须保存两个集合——Builders 和 BuildStrategies。其中 Builders 是 IBuilder 的集合，而 BuildStrategies 则提供了分类保存 IBuildPhase（及其相关 IBuildStep）的功能。那么使用 Generics 与全部保存为 System.Object 有多少区别呢？

- 便于阅读，需要什么类型的抽象类型（接口或抽象类），一目了然。
- 编译的时候就进行强类型检查，而不是在运行过程中才获得一个 InvalidCastException。

- Build 负责装备需要的类型，是应用中相对最繁忙的类型之一，如上 Build 涉及多个 Build Phase，每个 Build Phase 又可能包括多个 Build Step，即便做 System.Object 与具体接口的类型转换只需要增加很小的负载，但如果用于比较繁忙的业务情景，就不再会是个小负载了。

Generics 在结构型和行为型模式中的作用主要还是提高算法的适用性，例如：Adapter 完成的是接口的转换，从更抽象的角度看，就是一个 Type Cast（类型转换），所以用泛型可以定义出一个普适的 IGenericAdapter，示例代码如下：

C#

```
using System;
namespace MarvellousWorks.PracticalPattern.Concept.Generics
{
   public interface ITarget { void Request();}
   public interface IAdaptee { void SpecifiedRequest();}

   public abstract class GenericAdapterBase<T> : ITarget
      where T : IAdaptee, new()
   {
      public virtual void Request()
      {
         new T().SpecifiedRequest();
      }
   }
}
```

容器

泛型的另一个主要用途是建立各种强类型的集合类型，也就是定义容器中具体内容的类型。设计模式涉及管理“一组”对象的不是少数，仅最经典的描述中就包括了：

- Builder 的 Director；
- Composite；
- Flyweight 自己的 Factory；
- Command 的 Invoker；
- Memento 的 Care Keeper；
- State 和 Strategy 的 Context。

除此之外，实际工程中还要考虑容器的访问方式，即需要 IDictionary<K, V>、IList<T>、Stack<T>和 Queue<T>之类的类型。如果不使用泛型，那么后面的应用就纯属于“碰运气”访问，因为无法确定别人会怎么用你的公共库，至于到底外面传入的是什么？不清楚，都被容器包成为 System.Object 了，您可以假设它是实现了某个接口的类实例。但如果一个企业的公共代码库处处都基于“估计”，这个产品就太不靠谱了。相对一般的泛型类而言，泛型对集合类意义更重要：**接口和参数更明确，而且不仅仅停留在 UML 的图纸上**。

类型参数约束

另一个更为清晰的定义手段来自类型参数的约束机制，除了 MSDN 介绍的控制实例化过程外，还有如下注意事项：

- 当类成员使用相同类型参数的同时，该类型参数的约束也同样适用于相关成员。示例代码如下：

C#

```
namespace MarvellousWorks.PracticalPattern.Concept.Generics
{
   public interface IOrganization { }

   public abstract class UserBase<TKey, TOrganization>
      where TOrganization : class, IOrganization, new()
   {
      public abstract TOrganization Organization { get;}  // method
      public abstract void Promotion(TOrganization newOrganization);
      // property
      delegate TOrganization OrganizationChangedHandler(); // delegate
   }
}
```

- 参数约束不适用于 Attribute。

类型参数和类型参数约束的搭配让您的设计“收放自如”：类型参数通过抽象具体操作类型，让类和接口更加通用——“放”；但是您的算法又必须“有的放矢”，适用于特定的类型，并且当别人使用这些算法的时候，能通过一个“点”，即可由 IntelliSense 反射出一些东西，让编码更为快捷，这就需要对类型参数做限制——“收”。

本书中，类型参数约束的使用会更加频繁，作为 Example 没关系，无参的构造函数、无参的方法、还有 void 的返回值就可以了，只要能 Console.WriteLine；但是作为工程化代码，没有 where T : IProduct, new()的类型约束，在 Factory 里面 new T()的时候就会提示编译错误，更不会返回实例结果。所以，如果您不是自己练手，而是真的要把设计模式应用于具体工程，笔者有两个建议：

1. 设计每一个模式角色类的时候，要根据客户程序的需要，反复斟酌类型参数约束。
2. 除了容器类以外，尽可能不要在生产代码里出现无约束的类型参数（“裸类型约束”）。

除此之外，如果您把设计模式应用到组织的公共开发库，可能还有如“where T : A || B”或“where T : A && B”之类的代码，也就是说，您希望某个特定算法仅仅被某几个特定抽象类型使用。很可惜，.NET 不支持。那么变通的办法和我们用设计模式思想解耦其他关联的办法一样——引入新对象。示例代码如下：

C#

```
namespace MarvellousWorks.PracticalPattern.Concept.Generics
{
```

```
    // 解决类似 where T : A || B 的需要
    interface INewComer { }
    class OrA : INewComer { }
    class OrB : INewComer { }
    class OrClient<T> where T : INewComer { }

    // 解决类似 where T : A && B 的需要
    interface Layer11 { }
    interface Layer12 { }
    interface Layer2 : Layer11, Layer12 { }
    class AndA : Layer11 { }
    class AndB : Layer12 { }
    class AndClient<T> where T : Layer12 { }
}
```

不过重复写出 INewComer、ILayer 比较费时费神，可以用 Visual Studio .NET 的“Extract Interface”菜单快速解决这个问题，如图 1-8 所示。

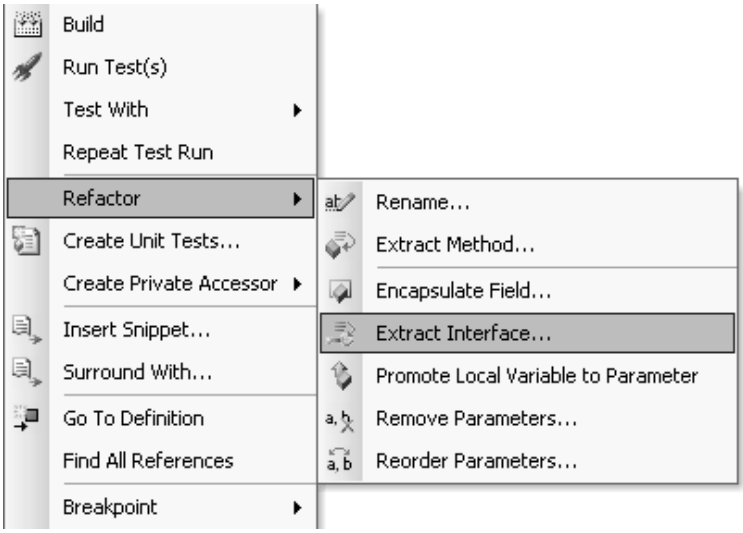


图 1-8 使用 Visual Studio .NET 菜单简化代码框架的生成

小结

如果您的代码将被反复重用，只要进度允许，尽量泛型吧，因为：

- 省去 Cast 过来 Cast 过去后，效率提高了。
- 抽象给您的代码带来更多的适应性。
- 减少接口和参数的歧义。

1.2.4 用作标签的方式扩展对象特性——属性（Attribute）

从使用者角度看，恐怕没有比 Attribute 更方便的了，基于属性的编码完全站在实际逻辑

外面，如果说经典设计模式中 Decorator 通过套接在不生成子类的情况下为类添加职责，那么 Attribute 则通过一个更简洁的方法为类“装饰”出职责和特性的机制。

用 Attribute 指导模式

这里以 Builder 模式为例。首先，按照经典的做法定义抽象 Builder 和一个实体 Builder，在此为了后续 Unit Test 的方便，增加了一个 Log 属性，记录每个 Build Part 的调用过程。代码如下：

C#

```
// Builder 抽象行为定义
public interface IAttributedBuilder
{
    IList<string> Log { get;}     // 记录 Builder 的执行情况
    void BuildPartA();
    void BuildPartB();
    void BuildPartC();
}

public class AttributedBuilder : IAttributedBuilder
{
    private IList<string> log = new List<string>();
    public IList<string> Log { get { return log; } }

    public void BuildPartA() { log.Add("a"); }
    public void BuildPartB() { log.Add("b"); }
    public void BuildPartC() { log.Add("c"); }
}
```

此后，将 Director 指导 Builder 组装的每个步骤通过 DirectorAttribute 属性类来表示，而 Director 在 BuildUp 的过程中，通过反射获得相关 Builder 的 DirectorAttribute 列表，对列表进行优先级排序后，执行每个 DirectorAttribute 指向的 Build Part 方法。代码如下：

C#

```
// 通过 attribute 扩展 Director
[AttributeUsage(AttributeTargets.Class, AllowMultiple =true)]
public sealed class DirectorAttribute : Attribute,
  IComparable<DirectorAttribute>
{
    private int priority;    // 执行优先级
    private string method;
    public DirectorAttribute(int priority, string method)
    {
        this.priority = priority;
        this.method = method;
    }
    public int Priority { get { return this.priority; } }
    public string Method { get { return this.method; } }

    // 提供按照优先级比较的 ICompare<T> 实现，由于 Array.Sort<T>
    // 实际是升序排列，而 Array.Reverse 是完全反转，因此这里调整了
```

```
    // 比较的方式为 “输入参数优先级-当前实例优先级”
    public int CompareTo(DirectorAttribute attribute)
    {
        return attribute.priority - this.priority;
    }
}

public class Director
{
    public void BuildUp(IAttributedBuilder builder)
    {
        // 获取 Builder 的 DirectorAttribute 属性
        object[] attributes = builder.GetType().GetCustomAttributes(
          typeof(DirectorAttribute), false);
        if (attributes.Length <= 0) return;
        DirectorAttribute[] directors =
          new DirectorAttribute[attributes.Length];
        for (int i = 0; i < attributes.Length; i++)
            directors[i] = (DirectorAttribute)attributes[i];
        // 按每个 DirectorAttribute 优先级逆序排序后，逐个执行
        Array.Sort<DirectorAttribute>(directors);
        foreach (DirectorAttribute attribute in directors)
            InvokeBuildPartMethod(builder, attribute);
    }

    // helper method : 按照 DirectorAttribute 的要求，执行相关的 Builder 方法
    private void InvokeBuildPartMethod(
        IAttributedBuilder builder, DirectorAttribute attribute)
    {
        switch (attribute.Method)
        {
            case "BuildPartA": builder.BuildPartA(); break;
            case "BuildPartB": builder.BuildPartB(); break;
            case "BuildPartC": builder.BuildPartC(); break;
        }
    }
}
```

接着，用做好的 DirectorAttribute 来定义 Builder 的装配过程。示例代码如下：

C#

```
[Director(1, "BuildPartB")]
[Director(2, "BuildPartA")]
public class AttributedBuilder : IAttributedBuilder
```

Unit Test

```
    [TestMethod]
    public void Test()
    {
        IAttributedBuilder builder = new AttributedBuilder();
        Director director = new Director();
        director.BuildUp(builder);
        Assert.AreEqual<string>("a", builder.Log[0]);
        Assert.AreEqual<string>("b", builder.Log[1]);
    }
```

如果要修改 Builder 的装配过程，仅需要增加、维护相关属性即可，例如下列代码：

C#

```
[Director(3, "BuildPartA")]
[Director(2, "BuildPartB")]
[Director(1, "BuildPartC")]
public class AttributedBuilder : IAttributedBuilder
```

Unit Test

```
[TestMethod]
public void Test()
{
    IAttributedBuilder builder = new AttributedBuilder();
    Director director = new Director();
    director.BuildUp(builder);
    Assert.AreEqual<string>("a", builder.Log[0]);
    Assert.AreEqual<string>("b", builder.Log[1]);
    Assert.AreEqual<string>("c", builder.Log[2]);
}
```

实际工程中，Attribute 常常会和反射、配置一同使用，比如[Director(2, "BuildPartA")]中优先级和方法名称都可以在配置文件定义。虽然看起来开发阶段增加了一些额外的代码工作（例如 Director 和 DirectorAttribute 的编码），但从使用者角度看，减少了反复定义 Director 相关 BuildUp 装配的过程。对于其他行为型和结构型模式，Attribute 同样可以从很多方面扩展，减少客户程序使用的操作复杂程度。比如：

- 把 State 声明为属性。
- 把各类 Proxy 需要的控制属性通过 Attribute 体现出来。

进一步方便客户程序调用

虽然可通过 Attribute 扩展模式中相关角色类的特征，但如果 BuildUp 的不是三个 Part，而是一架飞机，罗列 40 多个 DirectorAttribute 就似乎不太合适，代码看起来也太“涩”了；另一个问题，动态地增加 DirectorAttribute 需要反复地编译代码，把本来的一点点方便都掩盖了。

解决方法：让 Attribute 具有组合特性，把配置拉进来。这样做带来的变化就是可以改配置。

参考上面 Delegate 部分的 MulticastDelegate，可以扩展出具有 Multicast 特性的 Attribute，它采用经典的 Composite 来完成，如图 1-9 所示。

这样，每个 IAttributedBuilder 关联的仅仅是一个 IDirectorAttribute，它可能是一个原子的 DirectorAttribute，也可能是代表了一组 DirectorAttributeBase 的 MulticastDirectorAttribute，甚至一个复杂的 DirectorAttribute 树。即便相对比较复杂也没有关系，毕竟我们的配置文件本身就是 XML 的，它就是棵树。这样，如果 IAttributedBuilder 的属性需要修改，比如增加或删除一个 DirectorAttribute，修改配置文件即可，始终可以保持 IAttributedBuilder 与 IDirectorAttribute 的 1:1。

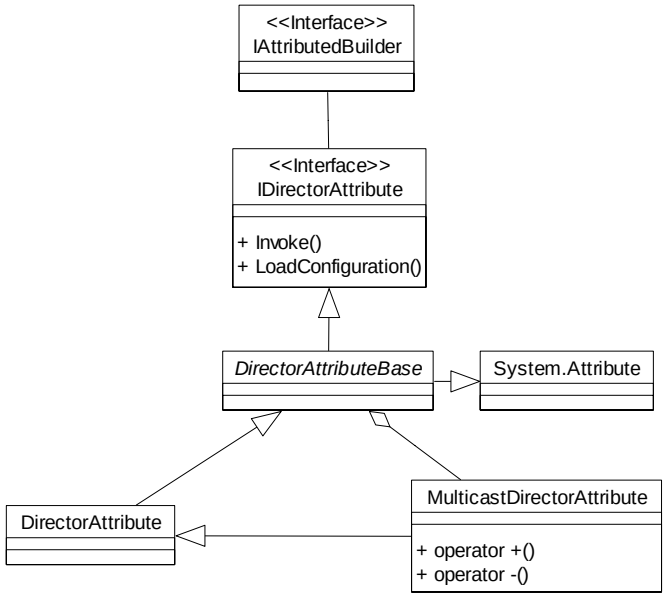


图 1-9 具有组合关系和 Attribute 构造器结构

小结

也许更多的时候，我们感觉基于 Attribute 的开发仅仅是新瓶装旧酒，OO 本质没有什么变化，但在工程上，它是个“让利”给下家的途径——用着更方便、写着更简单。

1.2.5 用索引器简化的 C#类型信息访问

索引器的确是很精致的语法元素。

服务于集合

索引器的出现，让我们首次感觉到真正意义上的容器类型或集合类型的存在。在它出现之前，要达到类似的效果，主要有两种选择：

1. 保存为数组，然后按照 for(int i=0; i<arr.Length; i++)读取。
2. 提供一组 get***(idx)和 put/set***(idx)的方法。

虽然 Java 借鉴 C#已经改进了很多，但是在 Indexer 上没有变化，相应的集合类型都是通过 add、remove、get 之类的方法读取。C#中的 Indexer 给人一种更“透彻”的感觉，集合类型就是集合类型，有自己专用但又最简洁的访问方式，而且同一种类型可以有不同的索引访问方式。

企业应用中存在非常多的编码：经营单位编码、货币类型编码、工作流步骤编码等，应用中常常把它们作为参数使用。回想一下我们在建立这些表的时候仅仅有一个 PK（Primary

Key），还是常常在 PK 之外，增加一些 IDX（索引）呢？可能就一个 PK，也可能是 PK + IList<IDX>；Indexer 也一样，可以有一个，也可以有多个。

封装单列集合的访问

即便是单列集合，其应用也非常普遍，以 MSDN 的 Index Tab 为例，输入 Array 之后的效果如图 1-10 所示。

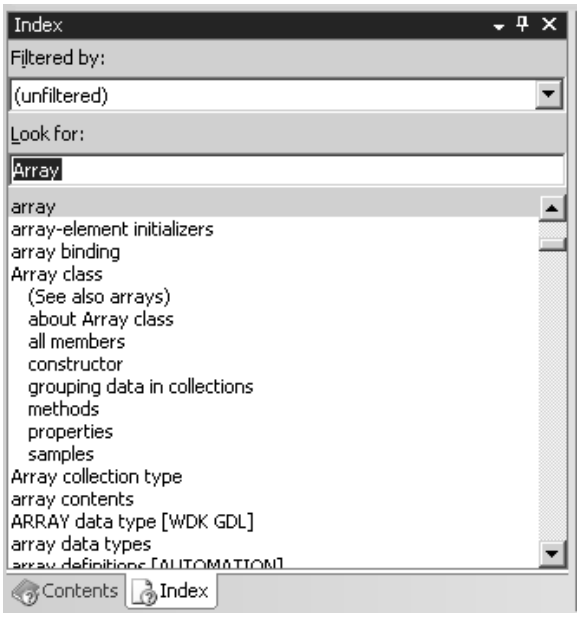


图 1-10 MSDN Library 的索引查询结果

也就是说，索引不仅仅是我们常规思维中 1:1 的精确匹配，应用中尤其涉及和用户交互的过程，即便是单列数据也常常会在 Index 时出现 1:n 的情况。示例代码如下：

C#

```
using System;
namespace MarvellousWorks.PracticalPattern.Concept.Indexer
{
   public class SingleColumnCollection
   {
      private static string[] countries = new string[] { "china",
         "chile", "uk" };

      public string this[int index] { get { return countries[index]; } }
      public string[] this[string name]
      {
         get
         {
            if((countries == null) || (countries.Length <= 0)) return null;
            return Array.FindAll<string>(countries,
```

```
                delegate(string candicate) { return
                  candicate.StartsWith(name); });
            }
        }
    }
}
```

Unit Test

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MarvellousWorks.PracticalPattern.Concept.Indexer;
namespace MarvellousWorks.PracticalPattern.Concept.Test
{
    [TestClass()]
    public class SimpleColumnCollectionTest
    {
        [TestMethod]
        public void Test()
        {
            SingleColumnCollection c = new SingleColumnCollection();
            Assert.AreEqual<string>("china", c[0]);
            Assert.AreEqual<int>(2, c["ch"].Length); // 命中 china 和 chile 两项
            Assert.AreEqual<string>("china", c["ch"][0]);
        }
    }
}
```

当然，应用中还存在另一种解决办法——“长长的流水账”方式，WinForm 方式还好，但如果是网页（见图 1-11）——“反正我把知道的都告诉你了，耐心等待，然后自己选吧；至于还想看到里面的方法，算了吧，那么多内容一次都告诉你我嘴皮子都得磨破了。”

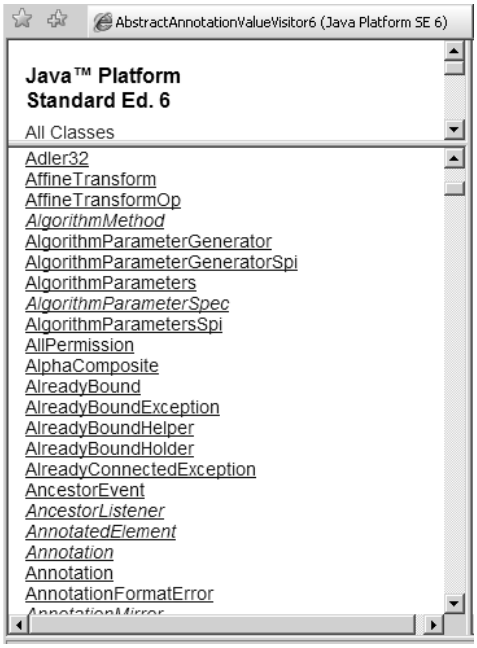


图 1-11 JavaDoc 的索引结果

多列的集合

多列的集合平时用得最多的恐怕非 System.Data 下的 DataSet 和 DataTable 莫属了，我们看看 Indexer “over Indexer 再 over Indexer”之后的效果，示例代码如下：

C#

```
using System;
using System.Data;
namespace MarvellousWorks.PracticalPattern.Concept.Indexer
{
    public class MultiColumnCollection
    {
        private static DataSet data = new DataSet();
        static MultiColumnCollection()
        {
            data.Tables.Add("Data");
            data.Tables[0].Columns.Add("name");
            data.Tables[0].Columns.Add("gender");
            data.Tables[0].Rows.Add(new string[] { "joe", "male" });
            data.Tables[0].Rows.Add(new string[] { "alice", "female" });
        }
        public static DataSet Data { get { return data; } }
    }
}
```

Unit Test

```
    [TestClass()]
    public class MultiColumnCollectionTest
    {
        [TestMethod]
        public void Test()
        {
            Assert.AreEqual("joe",
               MultiColumnCollection.Data.Tables[0].Rows[0]["name"]);
            Assert.AreEqual("female",
               MultiColumnCollection.Data.Tables[0].Rows[1][1]);
        }
    }
```

实现类似 RDBMS 中联合主键或唯一性索引的访问

“索引器”这个名称使我们很自然地联想到 RDBMS（关系数据库）中的索引，就如我们在设计数据库逻辑结构的过程一样，为了唯一标注每条记录，常常会用到主键或唯一性索引，而构成它们属性（列）的可能是一项也可能是几项的联合。.NET 平台为了跨层调用的方便，从一开始就支持离线的 DataSet 和基于 DOM 的 XML 解析数据，随着.NET 平台升级到 2.0，对象化的配置类型也可以提供基于内存缓冲信息的访问。应用可能要求包装类型提供基于联合索引的查询（尤其对于属性较多、关系复杂的实体），而索引器又成了一个非常优雅的封装方式。

比如：一个员工实体包括“FirstName”、“FamilyName”、“Title”3 项属性，我们需要包装一个 Staff 类型管理全部的员工信息，如图 1-12 所示。

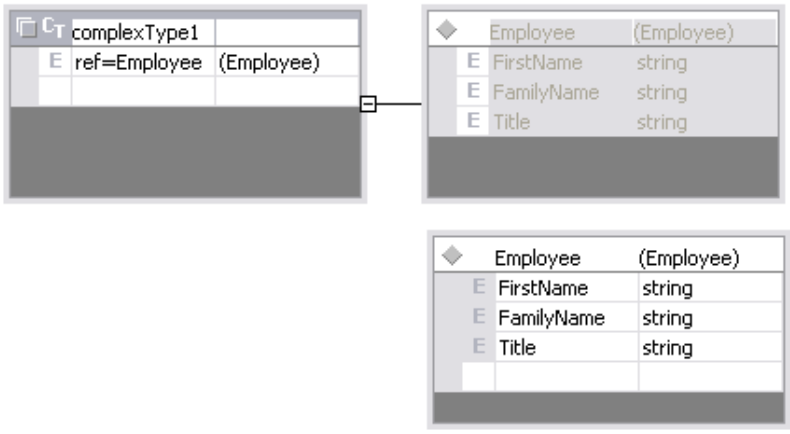


图 1-12 具有联合主键用户实体

同时根据 UI 绑定或其他功能检索的需要，我们会根据它的联合主键（FirstName + FamilyName）提供一个索引器以访问具体的员工记录。示例如下：

1. 完成具有操作联合索引的类

C#

```
/// 具有联合索引特点的实体类型
public struct Employee
{
    public string FirstName;    // PK Field
    public string FamilyName;   // PK Field
    public string Title;

    public Employee(DataRow row)
    {
        this.FirstName = row["FirstName"] as string;
        this.FamilyName = row["FamilyName"] as string;
        this.Title = row["Title"] as string;
    }
}

public class Staff
{
    static DataTable data = new DataTable();

    /// 数据准备
    /// <remarks>实际数据应该会从数据库等持久层渠道获得</remarks>
    static Staff()
    {
        data.Columns.Add("FirstName");
        data.Columns.Add("FamilyName");
        data.Columns.Add("Title");
        // pk : familyname + firstname
        data.PrimaryKey = new DataColumn[] { data.Columns[0],
          data.Columns[1] };
```

```
        data.Rows.Add("Jane", "Doe", "Sales Manger");
        data.Rows.Add("John", "Doe", "Vice President");
        data.Rows.Add("Rupert", "Muck", "President");
        data.Rows.Add("John", "Smith", "Logistics Engineer");
    }

    /// 基于联合 PK 检索
    public Employee this[string firstName, string familyName]
    {
        get
        {
            DataRow row = data.Rows.Find(new object[] { firstName,
              familyName });
            return new Employee(row);
        }
    }
}
```

2. 通过单元测试验证

Unit Test

```
[TestMethod]
public void FindStaff()
{
    Staff staff = new Staff();
    Employee employee = staff["John", "Doe"];
    string exptected = "Vice President";
    Assert.AreEqual<string>(exptected, employee.Title);
}
```

通过委托传递索引规则

如上文，对于检索规则固定的情况而言，我们可以通过在索引器内部硬编码完成，但如果要完成一些更为公共的类库，往往还要“授之以渔”，即除了告诉它“要检索”之外，还要告知检索策略和规则。在这方面 C#是非常有优势的，因为它有对象化的托管委托类型（delegate），而且.NET Framework FCL 部分也提供了很多现成的委托，所以我们不妨善加利用。

这时候，我们会发现索引器的功能更加强大，就像在使用 SQL 语句的 WHERE 子句一样，根据需要以灵活的方式对目标数据进行筛选。示例代码如下：

C#

```
public class Dashboard
{
    float[] temps = new float[10] {
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F, 61.3F, 65.9F, 62.1F, 59.2F, 57.5F };

    /// 与 SQL 语句中 Where 子句的效果非常类似
    /// <param name="predicate">传入的检索规则</param>
    public float this[Predicate<float> predicate]
    {
        get
```

```
        {
            float[] matches = Array.FindAll<float>(temps, predicate);
            return matches[0];
        }
    }
}
```

Unit Test

```
[TestMethod]
public void FindData()
{
    float expected = 65.9F;
    Dashboard dashboard = new Dashboard();
    float actual = dashboard[
        delegate(float data)    // Predicate<float>的委托
        {
            return data > 63F;
        }];
    Assert.AreEqual<float>(expected, actual);

    expected = 56.7F;
    actual = dashboard[
        delegate(float data)    // 更换规则
        {
            return ((data < 63F) && (data > 56.5F));
        }];
    Assert.AreEqual<float>(expected, actual);
}
```

不过，在实际使用中 WHERE 子句可能还会包括不只一条的限制条件，对此，索引器一样可以完成。例如，WHERE 语句可以定义为下列代码形式：

C#

```
/// 与 SQL 语句中一组 Where 子句的效果非常类似
public float this[params Predicate<float>[] predicates]
{
    get
    {
        // 具体实现可以参考上面的例子，基本上和我们写 SQL 的 Where 类似
        // 具体实现略过
        throw new NotSupportedException();
    }
}
```

LINQ 时代的索引器：乍一看，索引器似乎已经越来越接近于 LINQ 通过 Lamada 表达式完成的功能，不过有一些区别。

- 定位上索引器一般面向单条检索结果，而不是批量结果（尽管我们可以让索引器返回一个 IEnumerable<T>）。
- 从封装和客户程序使用的角度看，LINQ 有各种内置并被优化的 LINQ to 系列，而索引器给客户程序的是一种更贴近业务语义、更加直观的形式，因为客户程序无须编写 LINQ 查询，按照键值检索即可。

不过，把两者结合使用倒是一个非常不错的组合，索引器做接口，LINQ 完成内部检索逻辑，客户程序在无须记住具体方法名称的前提下，按照键值检索即可，索引器内部则依托 LINQ to 系列的基础，提供对各种异构数据源的访问。

小结

索引器具有“上善若水”的语言特性，当客户程序访问集合信息的时候，索引器会让代码显得异常的简洁、朴实，示例代码如下：

Conversation

```
客户程序：“C，你是个集合类型么？”
集合类型：“是”
客户程序：“那好，给我第 3 项。”
集合类型：“C[2]，拿去~~~”
```

就像我们设计接口时会根据业务领域把各类型的职能分解一样，操作类型同样可以根据访问内容的不同，选择使用不同的访问方法，比如：

- 索引器：承担各种检索和查找的工作。
- 属性（Property）：承担“它的……特性是……”或“它们的……特质是……”的工作，用来标注某个实例特性（成员属性）或静态特性（静态属性）。
- 普通的方法：承担“让它处理……”的职能。
- 而事件定位于“当……发生的时候，要作些……”。

受到惯性影响，我们常常把索引器作为一个仅按照编号反馈结果的入口，但就如 SQL 中的 WHERE 子句，我们可以做的其实很多。善用它就能令我们的程序更加亲切、更加清晰。

1.2.6 融入 C#语言的迭代机制——迭代器（Iterator）

除了 Template 这个最自然面向对象的语言特性外，迭代器恐怕是 C#可以提供的最简单的设计模式了。.NET Framework 有自己的 IEnumerator 接口，还有 Java 觉得不错也“学”过去的 foreach。迭代器的作用很明确：提供顺序访问一个内部组成相对复杂类型各个元素的手段，而客户程序无须了解那个复杂的内部组成。

工程环境中，除了 Composite、Iterator 两个基本的模式可以利用迭代器完成外，出于运行态检查或运行监控的需要，Facade、Flyweight、Interpreter、Mediator、Memento 等类的行为性、结构型模式同样可以通过迭代器简化设计，因为它们内部都需要组织一系列对象，而遍历每个成员又是经常性的操作。

基本应用

对同一个事物，不同人会有不同的看法。

对象也一样，虽然都是遍历，但是不同语境下、不同上下文限制下，要遍历的内容可能完全不同。正好，Iterator 提供了这样一个机会，对于一个类型，您可以提供多个 IEnumerable（泛型或非泛型的），同时还可以提供一个最基本的 IEnumerator（泛型或非泛型的）。示例代码如下：

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
namespace MarvellousWorks.PracticalPattern.Concept.Iterating
{
    public class RawIterator
    {
        private int[] data = new int[]{0, 1, 2, 3, 4};

        // 最简单的基于数组的全部遍历
        // 如果客户程序需要强类型的返回值，可以采用泛型声明 public IEnumerator<int>
        public IEnumerator GetEnumerator()
        {
            foreach (int item in data)
                yield return item;
        }

        // 返回某个区间内数据的 IEnumerable
        public IEnumerable GetRange(int start, int end)
        {
            for (int i = start; i <= end; i++)
                yield return data[i];
        }

        // 手工“捏”出来的 IEnumerable<string>
        public IEnumerable<string> Greeting
        {
            get
            {
                yield return "hello";
                yield return "world";
                yield return "!";
            }
        }
    }
}
```

Unit Test

```
    [TestClass()]
    public class RawIteratorTest
    {
        [TestMethod]
        public void Test()
        {
            int count = 0;  // 测试 IEnumerator
            RawIterator iterator = new RawIterator();
            foreach (int item in iterator)
                Assert.AreEqual<int>(count++, item);
```

```
            count = 1;      // 测试具有参数控制的 IEnumerable
            foreach (int item in iterator.GetRange(1, 3))
               Assert.AreEqual<int>(count++, item);

            string[] data = new string[] { "hello", "world", "!" };
            count = 0;      // 测试手工 “捏” 出来的 IEnumerable
            foreach (string item in iterator.Greeting)
               Assert.AreEqual<string>(data[count++], item);
        }
    }
```

简化复杂结构的遍历

当然，如果仅用迭代器遍历数组有点“杀鸡用牛刀”的感觉，迭代器最适用的场景还是访问复杂结构。这里假想一种情形——早上上班铃响的那一瞬间，想想员工都在什么地方？如图 1-13 所示。

- 在电梯里，马上就要到办公楼层的，那么我们假设他们都很有礼貌地被保存在一个栈里。
- 在楼下走廊和楼上走廊的，那么也假设他们很有礼貌地被保存在一个队列里。
- 到了办公室里马上就进入了一个有上下级关系的组织，假设他们被保存在二叉树里。
- 还有就是那一刻还没有打卡，而列入迟到名单的，姑且算被保存在一个数组里。

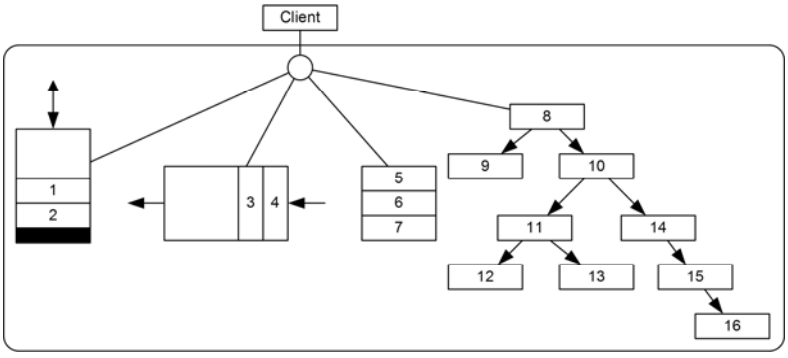


图 1-13 一个示例的内部对象组织结构

虽然从客户程序端角度来看，其内部结构似乎有些复杂，不过就像我们解决其他复杂问题一样，“分而治之”就好了，关键是为每个部分找到它们的 IEnumerator。

- Stack 和 Stack<T>都可以通过 GetEnumerator()获得。
- Queue、Queue<T>、Ayyay 也可以通过 GetEnumerator()获得。
- 至于那个二叉树，只能自己做了，不过就和遍历二叉树一样，安排好次序即可。
- 在确定了每个部分可以获得 IEnumerator 之后，需要有一个具有组合关系的容器来保存所有的类型，即便有更多的关系需要遍历也没关系，因为它有组合关系，把它们组合在一起，最后客户程序看到的也就只有一个对象（类似上文的 MulticastDelegate）。

1. 设计每个员工——ObjectWithName，其代码如下：

C#

```
public class ObjectWithName
{
    private string name;
    public ObjectWithName(string name) { this.name = name; }
    public override string ToString() { return name; }
}
```

2. 设计那个保存员工信息的二叉树，其代码如下：

C#

```
public class BinaryTreeNode : ObjectWithName
{
    private string name;
    public BinaryTreeNode(string name) : base(name) { }

    public BinaryTreeNode Left = null;
    public BinaryTreeNode Right = null;
    public IEnumerator GetEnumerator()
    {
        yield return this;
        if (Left != null)
            foreach (ObjectWithName item in Left)
                yield return item;
        if (Right != null)
            foreach (ObjectWithName item in Right)
                yield return item;
    }
}
```

3. 设计具有组合关系的 CompositeIterator，其代码如下：

C#

```
using System;
using System.IO;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
namespace MarvellousWorks.PracticalPattern.Concept.Iterating
{
    public class CompositeIterator
    {
        // 为每个可以遍历对象提供的容器
        // 由于类行为 object ， 所以 CompositeIterator 自身也可以嵌套
        private IDictionary<object, IEnumerator> items = new Dictionary<object,
            IEnumerator>();
        public void Add(object data) { items.Add(data, GetEnumerator(data)); }

        // 对外提供可以遍历的 IEnumerator
        public IEnumerator GetEnumerator()
        {
            if ((items != null) && (items.Count > 0))
```

```
            foreach (IEnumerator item in items.Values)
                while (item.MoveNext())
                    yield return item.Current;
        }

        // 获取 IEnumerator
        public static IEnumerator GetEnumerator(object data)
        {
            if (data == null) throw new NullReferenceException();
            Type type = data.GetType();

            // 是否为 Stack
            if (type.IsAssignableFrom(typeof(Stack))
                || type.IsAssignableFrom(typeof(Stack<ObjectWithName>)))
                return DynamicInvokeEnumerator(data);

            // 是否为 Queue
            if (type.IsAssignableFrom(typeof(Queue))
                || type.IsAssignableFrom(typeof(Queue<ObjectWithName>)))
                return DynamicInvokeEnumerator(data);

            // 是否为 Array
            if ((type.IsArray) && (type.GetElementType().IsAssignableFrom(
              typeof(ObjectWithName))))
                return ((ObjectWithName[])data).GetEnumerator();

            // 是否为二叉树
            if (type.IsAssignableFrom(typeof(BinaryTreeNode)))
                return ((BinaryTreeNode)data).GetEnumerator();

            throw new NotSupportedException();
        }

        // 通过反射动态调用相关实例的 GetEnumerator 方法获取 IEnumerator
        private static IEnumerator DynamicInvokeEnumerator(object data)
        {
            if (data == null) throw new NullReferenceException();
            Type type = data.GetType();
            return (IEnumerator)type.InvokeMember("GetEnumerator",
                BindingFlags.InvokeMethod, null, data, null);
        }
    }
}
```

4. 单元测试验证，其代码如下：

C#

```
    [TestClass()]
    public class CompositeIteratorTest
    {
        [TestMethod]
        public void Test()
        {
            #region 准备测试数据
            // stack<T>
            Stack<ObjectWithName> stack = new Stack<ObjectWithName>();
            stack.Push(new ObjectWithName("2"));
```

```
            stack.Push(new ObjectWithName("1"));
            // Queue<T>
            Queue<ObjectWithName> queue = new Queue<ObjectWithName>();
            queue.Enqueue(new ObjectWithName("3"));
            queue.Enqueue(new ObjectWithName("4"));
            // T[]
            ObjectWithName[] array = new ObjectWithName[3];
            array[0] = new ObjectWithName("5");
            array[1] = new ObjectWithName("6");
            array[2] = new ObjectWithName("7");
            // BinaryTree
            BinaryTreeNode root = new BinaryTreeNode("8");
            root.Left = new BinaryTreeNode("9");
            root.Right = new BinaryTreeNode("10");
            root.Right.Left = new BinaryTreeNode("11");
            root.Right.Left.Left = new BinaryTreeNode("12");
            root.Right.Left.Right = new BinaryTreeNode("13");
            root.Right.Right = new BinaryTreeNode("14");
            root.Right.Right.Right = new BinaryTreeNode("15");
            root.Right.Right.Right.Right = new BinaryTreeNode("16");

            // 合并所有 IEnumerator 对象
            CompositeIterator iterator = new CompositeIterator();
            iterator.Add(stack);
            iterator.Add(queue);
            iterator.Add(array);
            iterator.Add(root);
            #endregion

            int count = 0;
            foreach (ObjectWithName obj in iterator)
                Assert.AreEqual<string>((++count).ToString(),
                  obj.ToString());
        }
    }
```

小结

无论您所面对的对象内部结构多么复杂，如果您的任务是封装它们，而不只是把它们全部 public 了，则尽可能地提供一个 Iterator，可以的话按照需要使用的领域各提供一个 Iterator（见图 1-14），这样您的继任者，当然也可能是“朝花夕拾”的您自己，一定会对当时所做的工作心存感激……

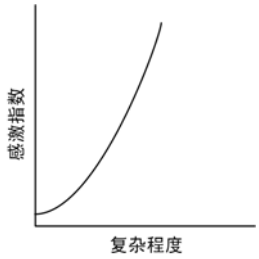


图 1-14 通过加入迭代器以缓解和便利复杂对象访问

1.3 可重载运算符（Overloadable Operators）与转换运算符（Conversion Operators）

1.3.1 The Day After Someday

假设有天一上班，领导把你叫过去，说：“用户的计费系统需要修改，除了价格 * 数量外，还要减去折扣，今天中午前最好更新上去”。也就是：

Math

```
Total = Price * Quantity 变成了
Total = Price * Quantity – Discount
```

你心想：“这有什么难的？马上就可修改好”。

但是发现了一个意想不到的情况：

- 因为不知名的原因，.NET Framework 不支持+、–、*、/和=了。
- Automatic Updates 已经帮您把公司所有的机器更新过了。

不过有个好消息，之前公司某人封装了一个 DoubleNumber 类。这么看来没有别的选择了——“我相信第一眼的感觉”，于是您也在很短的时间内更新了程序（方案 1），数日后一切正常后，您又用=和算术计算符重写了这个修改（方案 2）：

C# （方案 1）

```
// 没办法，没有= ，new 出来的 DoubleNumber 没地方存放
new DoubleNumber().Multiple(price, quantity, out result);
new DoubleNumber().Substract(result, discount, out result);
```

C# （方案 2）

```
return price * quantity - discount;
```

对方案 1 有什么感觉呢？恐怕用一个字来形容是“乱”，用两个字来形容则是“麻烦”。不管是喜欢还是发自内心的厌恶，我们都学了很多年数学，已经习惯了等号、不等号还有加减乘除了，对于纯粹计算的类型，您可以调用某些计算方法完成，当然最简单的加减乘除可能还是直接用我们最习惯的数学运算符更直观。比起某些通用的开发语言来说，使用 C#的我们还算幸运，可以重载运算符，MSDN 中教科书式的复数类（Complex）相信大家都已经很熟悉了，那么同样的方便性是不是也可以用于其他工程化编码中呢？

可以。

1.3.2 用于有限的状态迭代

四季的更迭“不以尧存，不以桀亡”，对于季节类型而言，它本身具有很明显的状态特

征。其示例代码如下：

C#

```
namespace MarvellousWorks.PracticalPattern.Concept.Operator
{
    public class Season
    {
        public static readonly string[] Seasons =
            new string[] { "Spring", "Summer", "Autumn", "Winter" };
        private int current;
        public Season() { current = default(int); }
        public override string ToString() { return Seasons[current]; }

        public static Season operator ++(Season season)
        {
            season.current = (season.current + 1) % 4;
            return season;
        }
        public static implicit operator string(Season season) { return
           season.ToString(); }
    }
}
```

Unit Test

```
    [TestClass()]
    public class SeasonTest
    {
        [TestMethod]
        public void Test()
        {
            Season season = new Season();
            Assert.AreEqual<string>(Season.Seasons[0], season);
            season++;
            season++;
            Assert.AreEqual<string>(Season.Seasons[2], season);
        }
    }
```

同样地，对于常规意义上的状态图而言，它一样可以区分出前置和后置，到底要转移到哪种状态，需要依靠上下文和规则而定，虽然不像比例中的四季那样严格地周而复始，但一样可以++和--，只不过增加一个因素而已。

1.3.3 操作集合

又碰到集合了。上面说的 MulticastDelegate 和.NET Framework 自身的状态机制都有现成的操作集合的范本：需要 Add 就 +=，需要 Remove 就–=。就如在上文提到的委托一样，设计模式中很多地方都可以用重载运算符的办法，让客户程序操作的时候更简洁一些：

- Chain of Responsibility：增加一个链表节点。
- Command：给 Invoker 增加一个命名对象。
- Memento：增加一个“备忘”的内容项。

- ……

就像 Add 和 Remove 可以被重载一样，我们在使用可重载操作符的时候，也要“不拘一格”。其示例代码如下：

C#

```
using System.Collections.Generic;
namespace MarvellousWorks.PracticalPattern.Concept.Operator
{
    public class ErrorEntity
    {
        private IList<string> messages = new List<string>();
        private IList<int> codes = new List<int>();

        public static ErrorEntity operator +(ErrorEntity entity,
          string message)
        {
            entity.messages.Add(message);
            return entity;
        }
        public static ErrorEntity operator +(ErrorEntity entity, int code)
        {
            entity.codes.Add(code);
            return entity;
        }

        public IList<string> Messages { get { return messages; } }
        public IList<int> Codes { get { return codes; } }
    }
}
```

Unit Test

```
    [TestClass()]
    public class ErrorEntityTest
    {
        [TestMethod]
        public void Test()
        {
            ErrorEntity entity = new ErrorEntity();
            entity += "hello";
            entity += 1;
            entity += 2;
            Assert.AreEqual<int>(1, entity.Messages.Count);
            Assert.AreEqual<int>(2, entity.Codes.Count);
        }
    }
```

1.3.4 类型的适配

设计模式中 Adapter 承担的是不兼容类型间的适配工作，一般同类书籍中的实现操作会把 Adapter 设计为继承 Target 的独立类，但如果有个机制，确保在需要的时候它可以自动地转换为目标接口，是不是也可以达到类似的效果呢？其示例代码如下：

C#

```
using System;
using System.Collections.Generic;
namespace MarvellousWorks.PracticalPattern.Concept.Operator
{
    public class Adaptee
    {
        // 不兼容的接口方法
        public int Code { get { return new Random().Next(); } }
    }

    public class Target
    {
        private int data;
        public Target(int data){this.data = data;}
        // 目标接口方法
        public int Data { get{return data;}}
        // 隐式类型转换进行适配
        public static implicit operator Target(Adaptee adaptee)
        {
            return new Target(adaptee.Code);
        }
    }
}
```

Unit Test

```
    [TestClass()]
    public class AdapteeTest
    {
        [TestMethod]
        public void Test()
        {
            Adaptee adaptee = new Adaptee();
            Target target = adaptee;
            Assert.AreEqual<int>(adaptee.Code, target.Data);
            List<Target> targets = new List<Target>();
            targets.Add(adaptee);
            targets.Add(adaptee);
            Assert.AreEqual<int>(adaptee.Code, targets[1].Data);
        }
    }
```

这种做法有什么局限呢？那就是子类“沾不上光”，原因在于转换运算符是 static 的，子类不能按照继承自动获取这个类型转换的特性。这也就等于提醒我们：如果工程中 Target 和 Adaptee 可以保证 1:1，那么无所谓，直接进行类型转换就可以了；如果觉得隐式转换可能不安全，没关系，定义为显式的即可；如果 Target 本身就是个接口或抽象类，则最好打消这个想法，尽管抽象类也可以定义转换运算符。

1.3.5 小结

就像本节一开始的“The Day After Someday”一样，没有=、+、*的时候，我们很可能会有“别扭”的感觉；反言之，如果在必要的地方，无论是进行运算符重载还是使用类型转

换符，都会为编码提供不少的便利。

1.4 面向插件架构和现场部署的配置系统设计

工程化代码和 Example 代码另一个最大的区别就是有关配置的处理，Example 可以“捏”些参数，甚至定义几个 const，但是在工程化代码的编写中，除非您的成果应用领域非常狭小，而且极少需要重新编译并部署，用户也仅仅是有限的几个点，那么无所谓，您全部使用硬编码好了；如果不是，您可能要考虑给代码一个“活口”，通过配置给您的软件更自由、更具适应性的机会。

配置可以有很多种形式：

- App.config 或自己写的 XML。
- 存在数据库里的参数本身也可以被视为配置。
- .ini 文件和.inf 文件。
- 令人又爱又恨的注册表。
- 互联网和 SOA 时代，配置更是无所不在，很有可能费用的计算要基于外管局网页的汇率。

应用使用它们的途径也很多：

- 彻底“放任自流”，应用实体自己封装相关的访问措施。
- 通过通用的配置访问机制，隔离配置的实际物理存储，确保每个应用实体封装的时候仅关心逻辑存储上的各种配置信息。
- 彻底让应用逻辑不知道配置信息的存在，“哪里有配置？满眼都是对象。”这方面 Enterprise Library 在组合 ObjectBuilder + Configuration + Instrument 方面作了很好的表率，不过就是稍微重了些。

本书采用一种折中的办法，毕竟设计模式公共库关系的内容本身几乎不涉及具体数据源、日志持久对象之类的内容，相关的配置主要集中在动态加载的类型、相关集合类型的容量控制等，同时为了充分利用.NET Framework 平台的现有机制，相关的配置布局采用完全基于 App.Config 的自定义配置解决。为了便于客户程序根据需要选择的模式，每个模式自己独立定义出一个配置节组，名称和相关的 Class Library 项目的默认命名空间保持一致（不过按照.NET 的命名习惯，命名采用首字母小写的方式）。例如，Builder 模式：

C# Class Library Default Namespace

```
namespace MarvellousWorks.PracticalPattern.Builder
```

相应地配置 sectionGroup

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
```

```
  <configSections>
    <sectionGroup name="marvellousWorks.practicalPattern.builder" .../>
  </configSections>
  <marvellousWorks.practicalPattern.builder/>
</configuration>
```

1.4.1 认识.NET Framework 提供的主要配置实体类

本书中，各设计模式的配置全部基于.NET Framework 提供的配置实体，并通过.NET 自己的 app.config 解析完成，会涉及的实体基类如下：

- System.Configuration.ConfigurationSectionGroup（配置节组）

  仅仅是为了逻辑上组织各个配置节，虽然您可以不使用，或者整个项目就使用一个配置节，但为了确保您的 Assembly 可以在其他地方被反复重用，最好将 SectionGroup 与 Assembly 进行一对一“搭对”，甚至当 Assembly 由于历史原因 Façade 的东西太多的时候，一个 Assembly 也可以划分为几个 SectionGroup。至于划分标准，建议类比 SRP 原则，以 Assembly 的每个服务特性为边界，明确划分 SectionGroup。

- System.Configuration.ConfigurationSection（配置节）

  维护一个独立的配置内容，与 ConfigurationElement、ConfigurationElementCollection 不同，自定义的 ConfigurationSection 需要在<configuration><configSections/> </configuration>下注册，同时.NET Framework 会以<section/>或其父节点<sectionGroup>作为解析一块配置的入口点。

- System.Configuration.ConfigurationElementCollection（配置元素集合）

  配置元素集合本身不需要在<configuration><configSections/></configuration>下注册，它是一组配置元素的父节点。

- System.Configuration.ConfigurationElement（配置元素）

笔者感觉.NET 明确区分这四个类主要是便于开发人员更明确地操作配置文件。实际设计中，笔者建议按照以下步骤设计并解析自己的配置文件：

1. 首先完成配置文件的“骨架”，即先完成配置节组和配置节的操作。
2. 用配置元素集合填充相应的配置节。
3. 设计每个配置元素需要包括的属性。
4. 从最下层开始逐个设计每个配置元素类，这仅需根据需要的属性描述出每一个类即可。根据以往的经验，好像开发人员在做这部分“照葫芦画瓢”的过程中往往忽略了使用面向对象的继承特性，因此无论是出于后续扩展的需要，还是根据添加如监控之类的企业开发要求，笔者强烈建议：**一定要在每个具体配置元素类之下**、System.Configuration.ConfigurationElement **之上，设计一个自己应用的抽象基类**。

本书的主要目的是建立工程化的企业设计模式库，每个模式都要增加独立的配置类，为了便于客户程序的使用和扩展，不仅要有一个统一的配置元素抽象基类，在多数模式中还会继承出一个模式内的专用抽象基类。

5. 配置元素集合类本身。这相对要简单些，毕竟它的基本作用就是个容器。出于相同的目的，也要有个“基”，而且最好是个泛型的“基”。
6. 按照最初的“骨架”，实现配置节和配置节组，并把相关的配置元素或配置元素集合组装起来。
7. 在单元测试项目里，生成一个相应的 App.Config 文件，并用测试信息填充，通过 Unit Test 确认您的 Object – Configuration Mapping 设计可以正常运行。

这里有个技巧，因为默认 VSTS 的 Test Project 生成的目标 App.Config 与实际的 Assembly 不搭对，会在加载配置对象的时候出错，因此需要修改 Test Project 属性，在 Build Events 的 pre-build event command line 增加如下代码及如图 1-15 所示。

Command Line

```
copy /Y $(ProjectDir)app.config $(TargetPath).config
```

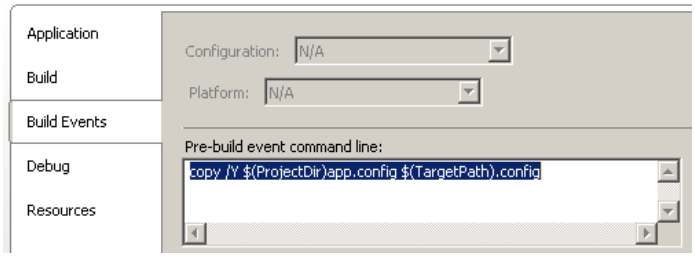


图 1-15 配置项目属性，保证 Unit Test 项目的 App.Config 可以被正确地部署到目标目录

1.4.2 应用实例

下面以本章 Delegate、Generics 两部分的大纲为例，通过 System.Configuration 相关机制提供一个解析范例。

1. 定义 App.Config 框架，包括一个 ConfigurationSectionGroup 和两个 ConfigurationSection，每个配置节下面包括一个<examples>和<diagrams>配置元素集合，委托介绍部分还有一个 Reflector.Net 的截图<picture>，其示例代码如下：

App.Config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- fake 代表还没有定义对象类型，临时声明一个 PlaceHolder -->
    <sectionGroup name="marvellousWorks.practicalPattern.concept"
```

```
        type="fake">
      <section name="delegate" type="fake"/>
      <section name="generics" type="fake"/>
    </sectionGroup>
  </configSections>

  <!-- 具体配置部分 -->
  <marvellousWorks.practicalPattern.concept>
    <delegate>
      <examples/>
      <diagrams/>
      <pictures/>
    </delegate>
    <generics>
      <examples/>
      <diagrams/>
    </generics>
  </marvellousWorks.practicalPattern.concept>
</configuration>
```

2. 定义每个<example>和<diagram>仅包括“name”（Required）和“description”（Optional）两个 attribute，而<picture>还需要增加一个是否为彩色的属性“colorized”（Required），相应的配置元素解析类如图 1-16 所示。

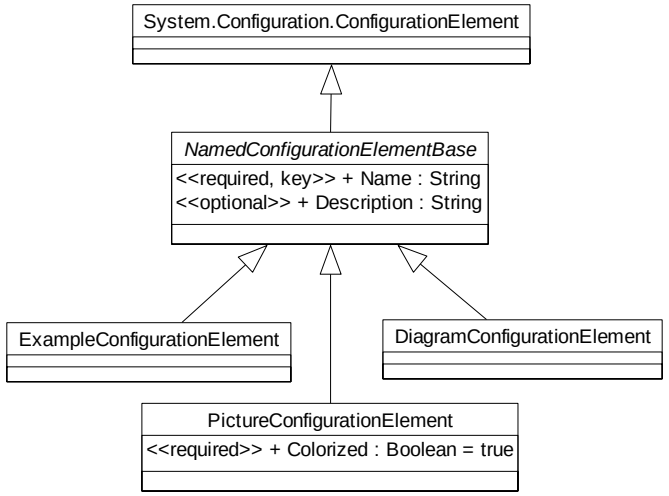


图 1-16 配置对象结构

示例代码如下：

C#

```
using System;
using System.Configuration;
namespace MarvellousWorks.PracticalPattern.Concept.Configurating
{
    // 定义具有 name 和 description 属性的配置元素
    // name 属性作为 ConfigurationElementCollection 中相应的 key
```

```
    public abstract class NamedConfigurationElementBase :
       ConfigurationElement
    {
        private const string NameItem = "name";
        private const string DescriptionItem = "description";

        [ConfigurationProperty(NameItem, IsKey=true, IsRequired=true)]
        public virtual string Name { get { return base[NameItem] as string; } }

        [ConfigurationProperty(DescriptionItem, IsRequired = false)]
        public virtual string Description { get { return base[DescriptionItem]
          as string; } }
    }

    public class ExampleConfigurationElement :
        NamedConfigurationElementBase { }
    public class DiagramConfigurationElement :
        NamedConfigurationElementBase { }
    public class PictureConfigurationElement : NamedConfigurationElementBase
    {
        private const string ColorizedItem = "colorized";

        [ConfigurationProperty(ColorizedItem, IsRequired = true)]
        public bool Colorized { get { return (bool)base[ColorizedItem]; } }
    }
}
```

3. 完成保存相应配置元素的容器——配置元素集合。

泛型，还是泛型。另外，声明这些容器为 AddRemoveClearMap，如图 1-17 所示。

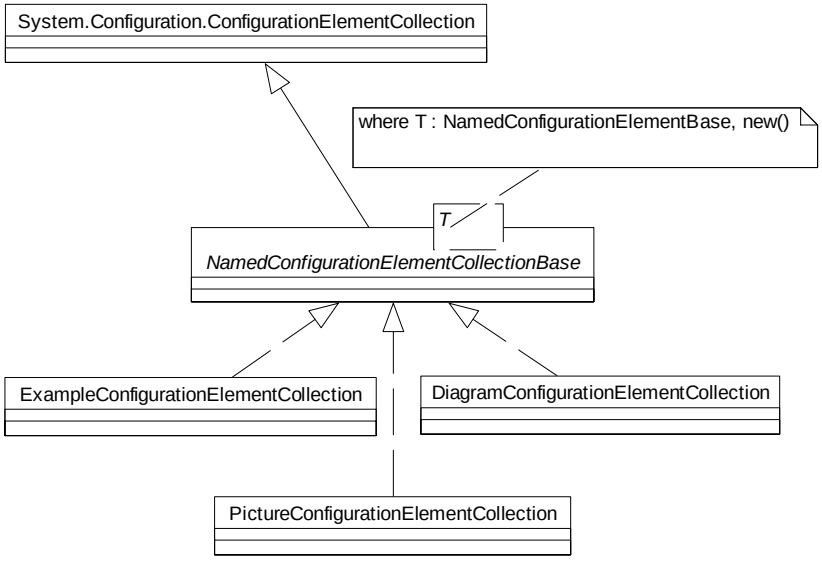


图 1-17 泛型化的配置对象结构

示例代码如下：

C#

```
using System;
using System.Configuration;
namespace MarvellousWorks.PracticalPattern.Concept.Configurating
{
    // 定义包括 NamedConfigurationElementBase 的
       ConfiugrationElementCollection
    [ConfigurationCollection(typeof(NamedConfigurationElementBase),
    CollectionType = ConfigurationElementCollectionType.AddRemoveClearMap)]
    public abstract class NamedConfigurationElementCollectionBase<T>
        : ConfigurationElementCollection
        where T : NamedConfigurationElementBase, new()
    {
        // 外部通过 index 获取集合中特定的 configurationelement
        public T this[int index] { get { return (T)base.BaseGet(index); } }
        public new T this[string name] { get { return (T)base.BaseGet(name); } }

        // 创建一个新的 NamedConfiugrationElement 实例
        protected override ConfigurationElement CreateNewElement() { return new
            T(); }

        // 获取集合中某个特定 NamedConfiugrationElement 的 key (Name 属性)
        protected override object GetElementKey(ConfigurationElement element)
        {
            return (element as T).Name;
        }
    }

    public class ExampleConfigurationElementCollection : NamedConfiguration-
       ElementCollectionBase<ExampleConfigurationElement> { }
    public class DiagramConfigurationElementCollection : NamedConfiguration-
       ElementCollectionBase<DiagramConfigurationElement> { }
    public class PictureConfigurationElementCollection : NamedConfiguration-
       ElementCollectionBase<PictureConfigurationElement> { }
}
```

4. 设计每个配置节的解析类，同时把相应的配置元素集合组装上去，如图 1-18 所示。

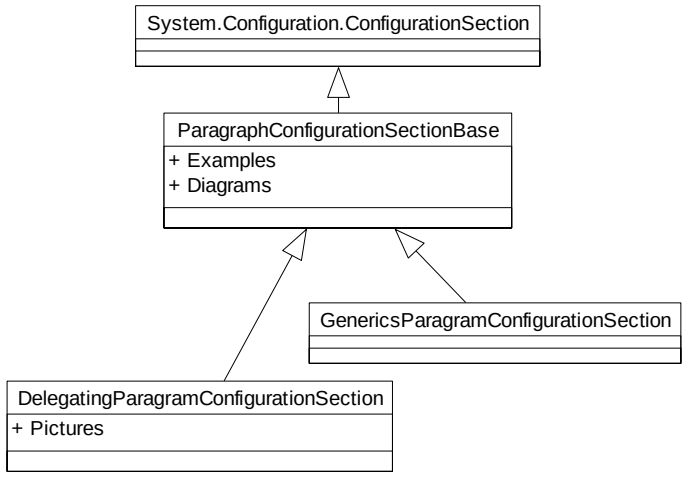


图 1-18 配置节部分的结构

示例代码如下：

C#

```
using System;
using System.Configuration;
namespace MarvellousWorks.PracticalPattern.Concept.Configurating
{
   // 文章段落的配置节类型包括:
   // 1. <examples>的 ConfigurationElementCollection (optional)
   // 1. <diagrams>的 ConfigurationElementCollection (optional)
   public abstract class ParagraphConfigurationSectionBase :
      ConfigurationSection
   {
      private const string ExamplesItem = "examples";
      private const string DiagramsItem = "diagrams";

      [ConfigurationProperty(ExamplesItem, IsRequired=false)]
      public virtual ExampleConfigurationElementCollection Examples
      {
         get
         {
            return base[ExamplesItem] as
              ExampleConfigurationElementCollection;
         }
      }

      [ConfigurationProperty(DiagramsItem, IsRequired=false)]
      public virtual DiagramConfigurationElementCollection Diagrams
      {
         get
         {
            return base[DiagramsItem] as
              DiagramConfigurationElementCollection;
         }
      }
   }

   public class DelegatingParagramConfigurationSection
      : ParagraphConfigurationSectionBase
   {
      private const string PicturesItem = "pictures";

      [ConfigurationProperty(PicturesItem, IsRequired = false)]
      public virtual PictureConfigurationElementCollection Pictures
      {
         get
         {
            return base[PicturesItem] as
              PictureConfigurationElementCollection;
         }
      }
   }
   public class GenericsParagramConfigurationSection
      : ParagraphConfigurationSectionBase { }
}
```

5. 用配置节组把“冰糖葫芦串一串”，其代码如下：

C#

```
using System;
using System.Configuration;
namespace MarvellousWorks.PracticalPattern.Concept.Configurating
{
    // 整个配置节组的对象, 包括<delegating> 和<generics> 两个配置节
    //  <sectionGroup name="marvellousWorks.practicalPattern.concept"/>
    public class ChapterConfigurationSectionGroup : ConfigurationSectionGroup
    {
        private const string DelegatingItem = "delegating";
        private const string GenericsItem = "generics";
        public ChapterConfigurationSectionGroup() : base() { }

        [ConfigurationProperty(DelegatingItem, IsRequired=true)]
        public virtual DelegatingParagramConfigurationSection Delegating
        {
            get{return base.Sections[DelegatingItem]
                as DelegatingParagramConfigurationSection;}
        }
        [ConfigurationProperty(GenericsItem, IsRequired=true)]
        public virtual GenericsParagramConfigurationSection Generics
        {
            get { return base.Sections[GenericsItem]
                as GenericsParagramConfigurationSection; }
        }
    }
}
```

6. 最后通过一个公共的 ConfigurationBroker 类统一对外每个配置节的访问，在 Test Project 的 App.Config 中填充测试配置信息，并通过 Unit Test 进行验证确认，其代码如下：

C#

```
    // 用于调度 App.Config 相关 Configuration 的 Broker 类型
    public static class ConfigurationBroker
    {
        private static ChapterConfigurationSectionGroup group;

        static ConfigurationBroker()
        {
            Configuration config = ConfigurationManager.OpenExeConfiguration(
                ConfigurationUserLevel.None);
            group = (ChapterConfigurationSectionGroup)config.GetSectionGroup
                  ("marvellousWorks.practicalPattern.concept");
        }

        public static DelegatingParagramConfigurationSection Delegating
        { get { return group.Delegating; } }
        public static GenericsParagramConfigurationSection Generics
        { get { return group.Generics; } }
    }
```

App.Config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- fake 代表还没有定义对象类型，临时声明一个 PlaceHolder -->
```

```
  <sectionGroup name="marvellousWorks.practicalPattern.concept" type=
     "MarvellousWorks.PracticalPattern.Concept.Configurating.
     ChapterConfigurationSectionGroup,Concept">
    <section name="delegating" type=
       "MarvellousWorks.PracticalPattern.Concept.Configurating.
       DelegatingParagramConfigurationSection,Concept"/>
    <section name="generics" type=
       "MarvellousWorks.PracticalPattern.Concept.Configurating.
       GenericsParagramConfigurationSection,Concept"/>
  </sectionGroup>
 </configSections>

 <!-- 具体配置部分 -->
 <marvellousWorks.practicalPattern.concept>
  <delegating>
   <examples>
    <add name="AsyncInvoke" description="用 Delegate 完成异步调用"/>
    <add name="MulticastNotify" description="1 对 n 的通知"/>
    <add name="AnonymousMethod" description="调用匿名方法"/>
   </examples>
   <diagrams>
    <add name="DelegateList" description="Delegate 集合与多个抽象方法关联"/>
    <add name="CompositeDelegate" description="MulticastDelegate 实现机制"/>
   </diagrams>
   <pictures>
    <add name="EventHandler" colorized="true"/>
   </pictures>
  </delegating>
  <generics>
   <examples>
    <add name="RawGenericFactory" description="通用工厂类"/>
    <add name="GenericAdapter"/>
   </examples>
   <diagrams>
    <add name="ClassicBuilder" description="经典 Builder 模式实现"/>
   </diagrams>
  </generics>
 </marvellousWorks.practicalPattern.concept>
</configuration>
```

Unit Test

```
[TestClass()]
public class ConfigurationBrokerTest
{
    [TestMethod]
    public void Test()
    {
        DelegatingParagramConfigurationSection s1 =
          ConfigurationBroker.Delegating;
        Assert.IsTrue(s1.Pictures["EventHandler"].Colorized);
        Assert.AreEqual<string>( "1 对 n 的通知",
          s1.Examples["MulticastNotify"].Description);
        GenericsParagramConfigurationSection s2 =
          ConfigurationBroker.Generics;
        Assert.AreEqual<int>(1, s2.Diagrams.Count);
    }
}
```

1.4.3 小结

按照.NET Framework“4 件套”的八股要求编写代码，相信您可以更轻松地为应用、为自己的 Assembly 建立起完整的配置支持。只要在编码的时候，如果突然想到“这个数、这个连接串、这个 Uri 是不是放到生产环境才能大概知道”，不要多想了，增加个配置节或配置元素，交给运维的人去管理吧。

编码的时候多八股几行代码，系统上线以后，相对会省心一些，尤其当你的代码被无数次复用，或者作为一个销路不错的产品的关键组成时，也许体会会更好一些。

不过，之前提到的单一职责原则同样适用，每一个配置项服务的内容尽量单一些，否则，一个配置项负责的职责增多以后，相应的变化因素也随之增加。

1.5 实现依赖注入

1.5.1 背景介绍

设计模式中，尤其是结构型模式很多时候解决的就是对象间的依赖关系，变依赖具体为依赖抽象。平时开发中如果发现客户程序依赖某个（或某类）对象，我们常常会对它们进行一次抽象，形成抽象的抽象类、接口，这样客户程序就可以摆脱所依赖的具体类型。

这个过程中有个环节被忽略了——谁来选择客户程序需要的满足抽象类型的具体类型呢？通过后面的介绍你会发现很多时候创建型模式可以比较优雅地解决这个问题。但另一问题出现了，如果您设计的不是具体业务逻辑，而是公共库或框架程序，这时候您是一个“服务方”，不是您调用那些构造类型，而是它们把抽象类型传给您，怎么松散地把加工好的抽象类型传递给客户程序就是另一回事了。

这个情形也就是常说的“控制反转”，IOC：Inverse of Control；框架程序与抽象类型的调用关系就像常说的好莱坞规则：Don’t call me, I’ll call you.

参考 Martin Fowler 在《Inversion of Control Containers and the Dependency Injection pattern》一文，我们可以采用“依赖注入”的方式将加工好的抽象类型实例“注入”到客户程序中，本书的示例也将大量采用这种方式将各种依赖项“注入”到模式实现的外部——客户程序。下面我们结合一个具体的示例看看为什么需要依赖注入，以及 Martin Fowler 文中提到的三种经典方式，然后依据 C#语言的特质，再扩展出一个基于 Attribter 方式注入（参考原有的 Setter 命名，这里将基于 Attribute 的方法称为 Attributer）。

1.5.2 示例情景

客户程序需要一个提供 System.DateTime 类型当前系统时间的对象，然后根据需要仅仅

把其中的年份部分提取出来，因此最初的实现代码如下：

C#

```
using System;
using System.Diagnostics;
namespace
MarvellousWorks.PracticalPattern.Concept.DependencyInjection.Example1
{
    class TimeProvider
    {
        public DateTime CurrentDate { get { return DateTime.Now; } }
    }

    public class Client
    {
        public int GetYear()
        {
            TimeProvider timeProvier = new TimeProvider();
            return timeProvier.CurrentDate.Year;
        }
    }
}
```

后来因为某种原因，发现使用.NET Framework 自带的日期类型精度不够，需要提供其他来源的 TimeProvider，确保在不同精度要求的功能模块中使用不同的 TimeProvider。这样问题集中在 TimeProvider 的变化会影响客户程序，但其实客户程序仅需要抽象地使用其获取当前时间的方法。为此，增加一个抽象接口，确保客户程序仅依赖这个接口 ITimeProvider，由于这部分客户程序仅需要精确到年，因此它可以使用一个名为 SystemTimeProvider (:ITimeProvider)的类型。新的实现代码如下：

C#

```
using System;
namespace
   MarvellousWorks.PracticalPattern.Concept.DependencyInjection.Example2
{
    interface ITimeProvider
    {
        DateTime CurrentDate { get;}
    }

    class TimeProvider : ITimeProvider
    {
        public DateTime CurrentDate { get { return DateTime.Now; } }
    }

    public class Client
    {
        public int GetYear()
        {
            ITimeProvider timeProvier = new TimeProvider();
            return timeProvier.CurrentDate.Year;
        }
```

```
    }
}
```

这样看上去客户程序后续处理权都依赖于抽象的 ITimeProvider，问题似乎解决了？没有，它还要知道具体的 SystemTimeProvider。因此，需要增加一个对象，由它选择某种方式把 ITimeProvider 实例传递给客户程序，这个对象被称为 Assembler。新的结构如图 1-19 所示。

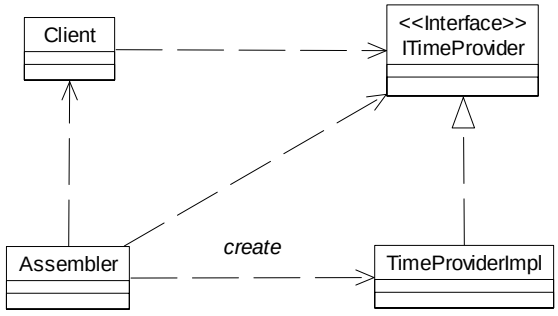


图 1-19 增建装配对象后新的依赖关系

其中，Assembler 的职责如下：

- 知道每个具体 TimeProviderImpl 的类型。
- 可根据客户程序的需要，将抽象 ITimeProvider 反馈给客户程序。
- 本身还负责对 TimeProviderImpl 的创建。

下面是一个 Assembler 的示例实现：

C#

```
public class Assembler
{
    /// <summary>
    /// 保存“抽象类型/实体类型”对应关系的字典
    /// </summary>
    private static Dictionary<Type, Type> dictionary =
       new Dictionary<Type, Type>();

    static Assembler()
    {
        // 注册抽象类型需要使用的实体类型
        // 实际的配置信息可以从外层机制获得,例如通过配置定义
        dictionary.Add(typeof(ITimeProvider), typeof(SystemTimeProvider));
    }

    /// 根据客户程序需要的抽象类型选择相应的实体类型，并返回类型实例
    /// <returns>实体类型实例</returns>
    public object Create(Type type)    // 主要用于非泛型方式调用
    {
        if ((type == null) || !dictionary.ContainsKey(type)) throw new
           NullReferenceException();
```

```
        Type targetType = dictionary[type];
        return Activator.CreateInstance(targetType);
    }

    /// <typeparam name="T">抽象类型（抽象类/接口/或者某种基类）</typeparam>
    public T Create<T>()    // 主要用于泛型方式调用
    {
        return (T)Create(typeof(T));
    }
}
```

1.5.3 Constructor 注入

构造函数注入，顾名思义，就是在构造函数的时候，通过 Assembler 或其他机制把抽象类型作为参数传递给客户类型。这种方式虽然相对其他方式有些粗糙，而且仅在构造过程中通过“一锤子”方式设置好，但很多时候我们设计上正好就需要这种 Read Only 的注入方式。其实现方式如下：

C#

```
/// 在构造函数中注入
class Client
{
    private ITimeProvider timeProvider;
    public Client(ITimeProvider timeProvider)
    {
        this.timeProvider = timeProvider;
    }
}
```

Unit Test

```
[TestClass]
public class TestClient
{
    [TestMethod]
    public void Test()
    {
        ITimeProvider timeProvider =
           (new Assembler()).Create<ITimeProvider>();
        Assert.IsNotNull(timeProvider);     // 确认可以正常获得抽象类型实例
        Client client = new Client(timeProvider);   // 在构造函数中注入
    }
}
```

1.5.4 Setter 注入

Setter 注入是通过属性赋值的办法解决的，由于 Java 等很多语言中没有真正的属性，所以 Setter 注入一般通过一个 set()方法实现，C#语言由于本身就有可写属性，所以实现起来更简洁，更像 Setter。相比较 Constructor 方式而言，Setter 给了客户类型后续修改的机会，它比较适应于客户类型实例存活时间较长，但 Assembler 修改抽象类型指定的具体实体类型相

对较快的情景；不过也可以由客户程序根据需要动态设置所需的类型。其实现方式如下：

C#

```
/// 通过 Setter 实现注入
class Client
{
    private ITimeProvider timeProvider;
    public ITimeProvider TimeProvider
    {
        get { return this.timeProvider; }    // getter 本身和以 Setter 方式实现
                                                注入没有关系
        set { this.timeProvider = value; }  // Setter
    }
}
```

Unit Test

```
[TestClass]
public class TestClient
{
    [TestMethod]
    public void Test()
    {
        ITimeProvider timeProvider =
          (new Assembler()).Create<ITimeProvider>();
        Assert.IsNotNull(timeProvider);      // 确认可以正常获得抽象类型实例
        Client client = new Client();
        client.TimeProvider = timeProvider; // 通过 Setter 实现注入
    }
}
```

1.5.5 接口注入

接口注入是将包括抽象类型注入的入口以方法的形式定义在一个接口里，如果客户类型需要实现这个注入过程，则实现这个接口，客户类型自己考虑如何把抽象类型“引入”内部。实际上接口注入有很强的侵入性，除了要求客户类型增加需要的 Setter 或 Constructor 注入的代码外，还显式地定义了一个新的接口并要求客户类型实现它。除非还有更外层容器使用的要求，或者有完善的配置系统，可以通过反射动态实现接口方式注入，否则笔者并不建议采用接口注入方式。

既然 Martin Fowler 文中提到了这个实现方式，就给出如下示例：

C#

```
/// 定义需要注入 ITimeProvider 的类型
interface IObjectWithTimeProvider
{
    ITimeProvider TimeProvider { get;set;}
}

/// 通过接口方式注入
```

```
class Client : IObjectWithTimeProvider
{
    private ITimeProvider timeProvider;
    /// IObjectWithTimeProvider Members
    public ITimeProvider TimeProvider
    {
        get { return this.timeProvider; }
        set { this.timeProvider = value; }
    }
}
```

Unit Test

```
[TestClass]
public class TestClient
{
    [TestMethod]
    public void Test()
    {
        ITimeProvider timeProvider = (new
           Assembler()).Create<ITimeProvider>();
        Assert.IsNotNull(timeProvider);     // 确认可以正常获得抽象类型实例
        IObjectWithTimeProvider objectWithTimeProvider = new Client();
        objectWithTimeProvider.TimeProvider = timeProvider; // 通过接口方式注入
    }
}
```

1.5.6 基于 Attribute 实现注入——Attributer

如果做个归纳，Martin Fowler 之前所介绍的三种模式都是在对象部分进行扩展的，随着语言的发展（.NET 从 1.0 开始，Java 从 5 开始），很多在类元数据层次扩展的机制相继出现，比如 C#可以通过 Attribute 将附加的内容注入到对象上。直观上的客户对象有可能在使用上做出让步以适应这种变化，但这违背了依赖注入的初衷，三个角色（客户对象、Assembler、抽象类型）之中两个不能变，那只好在 Assembler 上下功夫，谁叫它的职责就是负责组装呢？

为了实现上的简洁，上面三个经典实现方式实际将抽象对象注入到客户类型都是在客户程序中（也就是那三个 Unit Test 部分）完成的，其实同样可以把这个工作交给 Assembler 完成；而对于 Attribute 方式注入，最简单的方式则是直接把实现了抽象类型的 Attribute 定义在客户类型上，例如：

C#

```
(错误的实现情况)
    class SystemTimeAttribute : Attribute, ITimeProvider { … }

    [SystemTime]
    class Client { … }
```

相信您也发现了，这样虽然把客户类型需要的 ITimeProvider 通过“贴标签”的方式告诉它了，但事实上又把客户程序与 SystemTimeAttribute“绑”上了，它们紧密地耦合在一起。参考上面的三个实现，当抽象类型与客户对象耦合的时候我们引入了 Assembler，当

Attribute 方式出现类似的情况时，我们写个 AttributeAssembler 不就行了么？还不行。设计上要把 Attribute 设计成一个“通道”，考虑到扩展和通用性，它本身要协助 AttributeAssembler 完成 ITimeProvider 的装配，最好还可以同时装载其他抽象类型来修饰客户类型。示例代码如下：

C#

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
sealed class DecoratorAttribute : Attribute
{
    /// 实现客户类型实际需要的抽象类型的实体类型实例，即得注入客户类型的内容
    public readonly object Injector;
    private Type type;

    public DecoratorAttribute(Type type)
    {
        if (type == null) throw new ArgumentNullException("type");
        this.type = type;
        Injector = (new Assembler()).Create(this.type);
    }

    /// 客户类型需要的抽象对象类型
    public Type Type { get { return this.type; } }
}

/// 帮助客户类型和客户程序获取其 Attribute 定义中需要的抽象类型实例的工具类
static class AttributeHelper
{
    public static T Injector<T>(object target)
        where T : class
    {
        if (target == null) throw new ArgumentNullException("target");
        Type targetType = target.GetType();
        object[] attributes = targetType.GetCustomAttributes(
            typeof(DecoratorAttribute), false);
        if ((attributes == null) || (attributes.Length <= 0)) return null ;
        foreach (DecoratorAttribute attribute in
           (DecoratorAttribute[])attributes)
            if (attribute.Type == typeof(T))
                return (T)attribute.Injector;
        return null;
    }
}

[Decorator(typeof(ITimeProvider))]
// 应用 Attribute，定义需要将 ITimeProvider 通过它注入
class Client
{
    public int GetYear()
    {
        // 与其他方式注入不同的是，这里使用的 ITimeProvider 来自自己的 Attribute
        ITimeProvider provider =
           AttributeHelper.Injector<ITimeProvider>(this);
        return provider.CurrentDate.Year;
    }
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    Client client = new Client();
    Assert.IsTrue(client.GetYear() > 0);
}
```

1.5.7 小结

依赖注入虽然被 Martin Fowler 称为一个模式，但平时使用中，它更多地作为一项实现技巧出现，开发中很多时候需要借助这项技巧把各个设计模式所加工的成果传递给客户程序。各种实现方式虽然最终目标一致，但在使用特性上有很多区别。

- Constructor 方式：它的注入是一次性的，当客户类型构造的时候就确定了。它很适合那种生命期不长的对象，比如在其存续期间不需要重新适配的对象。此外，相对 Setter 方式而言，在实现上 Constructor 可以节省很多代码；
- Setter 方式：一个很灵活的实现方式，对于生命期较长的客户对象而言，可以在运行过程中随时适配；
- 接口方式：作为注入方式具有侵入性，很大程度上它适于需要同时约束一批客户类型的情况；
- 属性方式：随着开发语言的发展引入的新方式，它本身具有范围较小的固定内容侵入性（一个 DecoratorAttribute），它也很适合需要同时约束一批客户类型情景。它本身实现相对复杂一些，但客户类型使用的时候非常方便——“打标签”即可。

第3章

工厂&工厂方法模式

3.1 简单工厂

3.2 经典回顾

3.3 解耦 Concrete Factory 与客户程序

3.4 基于配置文件的 Factory

3.5 批量工厂

3.6 基于类型参数的 Generic Factory

3.7 委托工厂类型

3.8 小结

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses..

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

笔者一直认为工厂方法是整个创建型模式中最为典型的、也是最具启发效果的，它告诉我们使用一个变化频率比较高的类不必忙着 new()，而要依赖一个抽象的类型（抽象类或接口）。根据准备知识的介绍，.NET 平台上 Delegate 也是一个抽象，与抽象类型不同，它是对一类方法的抽象，而不像前两者是对一组方法的抽象。哪一种更好呢？

尺有所长，寸有所短。

如果需要的仅仅是某个特定的操作，那么大可不必按照抽象类型来加工，反馈一个 Delegate 实例就可以了；如果需要的是抽象业务实体，或者是具有一组“操作 + 属性”的抽象类型，那么就循规蹈矩好了。

使用工厂方法的主要动机来自于“变化”，应用的哪些组成会快速变化呢？不一定，多数项目会有一个相对稳定的核心，无论是被叫为 Framework，还是现在更时髦的、感觉上更底层的 Foundation，这个部分相对比较稳定，言外之意其他部分都会“相对”变化比较频繁。

我们可能期待架构师设计一个很灵活的架构，这样开发的时候就可以“填空”，而且是 Plug && Play 方式的；我们也可以期待需求分析人员把需求分析得特别透彻，这样我们代码真的 Write Once Run Always 了。当然，既然是 Team 工作，就有上下家，周围还有共同奋斗的同事们，我们希望上家的变化尽可能地少，同时出于人际工程学的需要，我们也尽量不为下家找麻烦。

很难。

基于接口的开发虽然不能解决上述那么多问题，但起码可以从很多方面减轻这些工作负担。如果想贯彻这种思想，那么首先要从类型构造上解决问题，否则后面的讨论很容易都成为空谈。（当然，可以通过正交的方法，或者一般被称为“拦截”的方式在具体方法执行上提供灵活性。后面章节会对此进行描述。）

3.1 简单工厂

3.1.1 最简单的工厂类

作为 Factory Method、Abstract Factory 的一个“预备”，首先用最朴实的方式完成一个工厂（其结构如图 3-1 所示），体现工厂与抽象类型间的构造关系，其代码如下：

C#

```
using System;
namespace MarvellousWorks.PracticalPattern.FactoryMethod.Example1
```

```
{
    public interface IProduct { }
    public class ConcreteProductA : IProduct { }
    public class ConcreteProductB : IProduct { }

    public class Factory
    {
        public IProduct Create()
        {
            // 工厂决定到底实例化哪个子类
            return new ConcreteProductA();
        }
    }
}
```

Unit Test

```
参考 FactoryMethod.Test 项目 Example1 的 FactoryTest
/// <summary>
/// 说明可以按照要求生成抽象类型，但具体实例化哪个类型由工厂决定
/// </summary>
[TestMethod]
public void Test()
{
    Factory factory = new Factory();
    IProduct product = factory.Create();
    Assert.AreEqual<Type>(product.GetType(), typeof(ConcreteProductA));
}
```

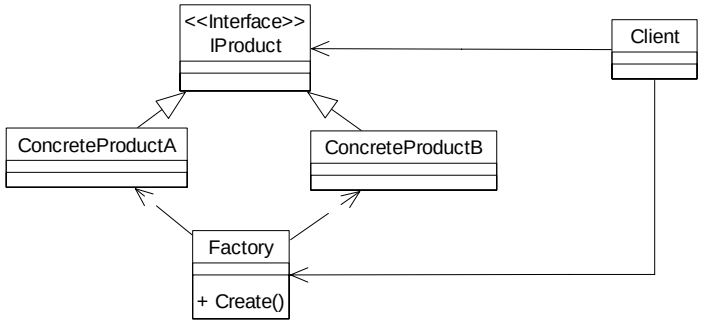


图 3-1 简单工厂的静态结构

这个工厂与直接使用 new()有什么不同？假如把它放到工程中应用有什么不尽如人意的地方？

- 通过 IProduct 隔离了客户程序与具体 ConcreteProductX 的依赖关系，在客户程序视野内根本就没有 ConcreteProductX。
- 即使 ConcreteProductX 增加、删除方法或属性，也无妨大局，只要按照要求实现了 IProduct 就可以，Client 无须关心 ConcreteProductX 的变化（确切地说它也关心不着，因为看不到）。
- 相对直接写个 ConcreteProductX 而言，要平白地多写一个工厂出来，尤其当需要 IProduct

频繁变化的时候，客户程序也闲不下来。

- 好的需求分析师可以在实施之前分析清楚 85%的需求，好的架构师在把这些需求转换为实际技术框架的时候大概能做到 90%的忠于需求，作为开发人员，设计的时候能够详细刻画 95%的内容就很不错了，这样 100% – 85% * 90% * 95% = 27.3%，也就是说，即便您身在一个精英团队，也可能有 1/4 的内容到了编码的时候还无法得到准确分析，那么对应用的某个局部领域而言，很难有效抽象这个 IProduct；但 Factory 提示在第一遍 Draft 的时候，我们就可以直接 new()，但在复查或迭代的过程中一定要尽量找到 IProduct，然后套个 Factory，在公共代码部分，更是如此。

此外，还有个效率问题。如果构造一个 Factory 实例，并用它获取一个抽象类型实例后就不再使用，资源上有些浪费，尤其当这个过程非常频繁的时候。工程中可以通过如下几种方式解决：

- 把工厂实例作为参数注入到操作中，而不是在每个方法内部自行创建，其代码如下：

C# Enterprise Library Data Access Application Block Database.cs 片断节选

```
private DbProviderFactory dbProviderFactory;
protected Database(string connectionString,
   DbProviderFactory dbProviderFactory)
{
    this.dbProviderFactory = dbProviderFactory;
}

public virtual DbConnection CreateConnection()
{
    DbConnection newConnection = dbProviderFactory.CreateConnection();
    newConnection.ConnectionString = ConnectionString;
    return newConnection;
}
```

- 把工厂设计为 Singleton 方式，因为工厂的职责相对单一，所有客户端需要的加工过程使用的都是一个唯一的共享实例。
- 使用静态类。虽然在经典的设计模式书籍示例中并没有采用，但它未尝不是一个有效节省资源使用的途径，不过切记：静态类不能被继承它只能从 object 继承，没其他可能。所以看起来静态类更像以前的 API 集合，有些不那么“面向对象”的味道。其示例代码如下：

C#

```
using System;
namespace MarvellousWorks.PracticalPattern.FactoryMethod.Example1
{
    public enum Category
    {
        A,
        B
    }
```

```
    public static class ProductFactory
    {
        public static IProduct Create(Category category)
        {
            switch (category)
            {
                case Category.A:
                    return new ConcreteProductA();
                case Category.B:
                    return new ConcreteProductB();
                default:
                    throw new NotSupportedException();
            }
        }
    }
}
```

Unit Test

```
/// <summary>
/// 说明静态工厂可以根据提供的目标类型枚举变量选择需要实例化的类型
/// </summary>
[TestMethod]
public void Test()
{
    IProduct product = ProductFactory.Create(Category.B);
    Assert.IsNotNull(product);
    Assert.AreEqual<Type>(typeof(ConcreteProductB), product.GetType());
}
```

静态工厂的静态结构如图 3-2 所示。

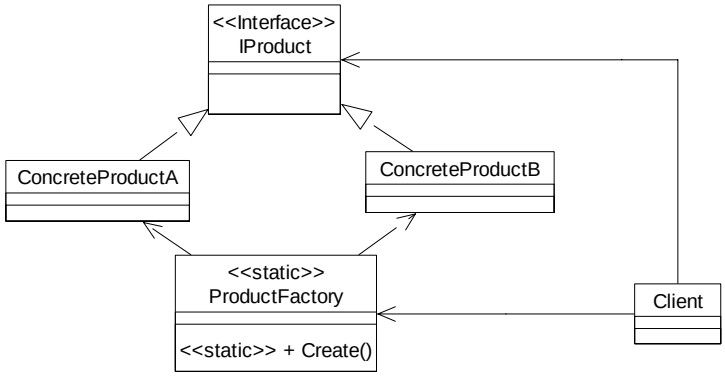


图 3-2 静态工厂的静态结构

3.1.2 根据规格加工产品——参数化工厂

本节中最初那个程序（IProduct、Factory）只能是个 Example，充其量也就是个模型。为什么？我们有两个 ConcreteProductX，如果没有任何机制告诉 Factory 该构造谁，工厂只能采用上面那个“认 A 为亲”，或者随机产生 A/B 之中某一个实例的方法，这实在不靠谱。如果想让工厂在运行态有效选择符合抽象类型要求的某个实例，最简单的机制莫过于传递一

个参数，它可能是一个字符串、一个 HRESULT 值、一个枚举值，或者是类似 Enterprise Library 中的某个 IConfigurationSource……，例如在上例静态类工厂的实现中就显式地传递了一个 Category 类型的枚举参数。

3.1.3 简单工厂的局限性

上例的简单工厂比较优雅地解决了外部 new()的问题，它把目标实例的创建工作交给一个外部的工厂完成，是设计模式化思想一个很不错的引子。但如果应用中需要工厂的类型只有一个，而且工厂的职责又非常单纯——就是一个 new()的替代品，类似我们面向对象中最普遍的思路，这时候就需要进一步抽象了，于是出现了新的发展：工厂方法模式和抽象工厂模式。

3.2 经典回顾

作为简单工厂的一个扩展，工厂方法的意图也非常明确，它把类的实例化过程延迟到子类，将 new()的工作交给工厂完成。同时，增加一个抽象的工厂定义，解决一系列具有统一通用工厂方法的实体工厂问题。它适合如下情景：

- 客户程序需要隔离它与需要创建的具体类型间的耦合关系。
- 很多情况下，客户程序在开发过程中还无法预知生产环境中实际要提供给客户程序创建的具体类型。
- 将创建性工作隔离在客户程序之外，客户程序仅需要执行自己的业务逻辑，把这部分职责交给外部对象完成。
- 如果目标对象虽然可以统一抽象为某个抽象类型，但它们的继承关系太过复杂，层次性比较复杂，这时同样可以通过工厂方法解决这个问题。

工厂方法主要有四个角色。

- 抽象产品类型（Product）：工厂要加工的对象所具有的抽象特征实体。
- 具体产品类型（Concrete Product）：实现客户程序所需要的抽象特质的类型，它就是工厂需要延迟实例的备选对象。
- 声明的抽象工厂类型（Creator）：定义一个工厂方法的默认实现，它返回抽象产品类型 Product。
- 重新定义创建过程的具体工厂类型（Concrete Creator）：返回具体产品类型 Concrete Product 的具体工厂。

于是，经典的工厂方法设计结构如图 3-3 所示。

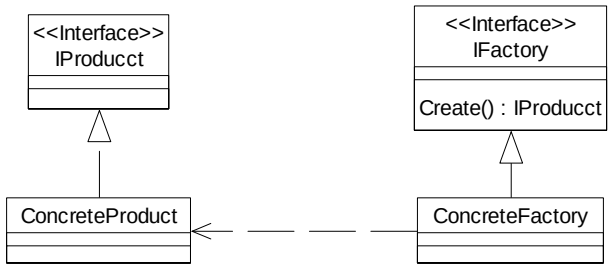


图 3-3 经典工厂方法模式的静态结构

示例代码如下：

C# 抽象产品类型与具体产品类型

```
/// 抽象产品类型
public interface IProduct
{
    string Name { get;}     // 约定的抽象产品所必须具有的特征
}

/// 具体产品类型
public class ProductA : IProduct
{
    public string Name { get { return "A"; } }
}
public class ProductB : IProduct
{
    public string Name { get { return "B"; } }
}
```

C# 抽象的工厂类型描述

```
public interface IFactory
{
    IProduct Create();  //  每个工厂所需要具有的工厂方法——创建产品
}
```

C# 两个实体的工厂类型

```
public class FactoryA : IFactory
{
    public IProduct Create()
    {
        return new ProductA();
    }
}

public class FactoryB : IFactory
{
    public IProduct Create()
    {
        return new ProductB();
    }
}
```

通过这个例子我们可以看到，当客户程序作为 IProduct 消费者的时候，抽象类型在虚拟构造（Virtual Constructor）的时候到底“挂”哪个实体类型由 Factory 决定。客户程序需要使用 IProduct 的时候通过访问 IFactory 所定义的 Create()方法就可以获得一个目标产品实例，直观上客户程序可以把频繁变化的某个 Product 隔离在自己的视线之外，自身不会受到它的影响。

3.3 解耦 Concrete Factory 与客户程序

虽然经典工厂方法模式告诉了我们最后的结果，通过 Product、Concrete Product、Factory 和 Concrete Factory 四个角色解决了客户程序对 Product 的获取问题，但没有把客户程序和 Factory 连起来，也就是说，没有介绍如何把 Factory 放到客户程序里。示例代码如下：

C# 两个实体的工厂类型

```
class Client
{
    public void SomeMethod()
    {
        IFactory factory = new FactoryA();
        // 获得了抽象 Factory 的同时与 FactoryA 产生依赖
        IProduct product = factory.Create();
        // 后续操作仅依赖抽象的 IFactory 和 IProduct;
        // ......
    }
}
```

如果让客户程序直接来 new()某个 Concrete Factory，由于 Concrete Factory 依赖于 Concrete Product，因此还是形成了间接的实体对象依赖关系，背离了这个模式的初始意图。怎么办？在前面的介绍中笔者提到了“依赖注入”的概念，也就是通过 Assembler 把客户程序需要的某个 IFactory 传递给它。因此，静态实现结构要在经典工厂方法模式上做些修改，如图 3-4 所示。

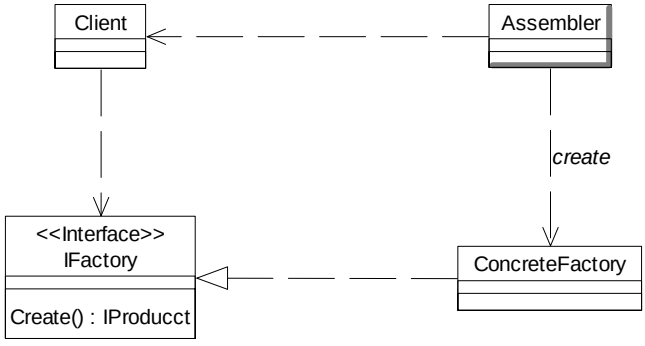


图 3-4 增加了装配对象的工厂方法静态结构

实际工程中很多时候不得不做这项工作——将 Concrete Product 与客户程序分离，其原因是我们不想让上游开发人员的针头线脑的修改迫使我们必须重新 Check Out 代码，编译、重新单元测试后再 Check In，这种活动在项目开发的中、后期常常是非常令人厌恶的，所以我们把这些“烂摊子”统统甩给 Assembler。参考第 1 章“实现依赖注入”部分的介绍，我们给那个版本的 Assembler 多增加一个注册项，然后按照适合的注入实现方式把 IFactory 传给客户程序，其代码如下：

C# 为 Assembler 增加有关 IFactory 的注册项

```
static Assembler()
{
    // 注册抽象类型需要使用的实体类型
    // 实际的配置信息可以从外层机制获得,例如通过配置定义
    dictionary.Add(typeof(ITimeProvider), typeof(SystemTimeProvider));
    dictionary.Add(typeof(IFactory), typeof(FactoryA));
}
```

C# 相应客户程序就可以修改为依赖 Assembler 的方式

```
class Client
{
    private IFactory factory;
    public Client(IFactory factory)
    // 外部机制将 IFactory 借助 Assembler 以 Setter 方式注入
    {
        if (factory== null) throw new ArgumentNullException("factory");
        this. factory= factory;
    }

    public void AnotherMethod()
    {
        IProduct product = factory.Create();
        // ... ...
    }
}
```

3.4 基于配置文件的 Factory

上面的方法虽然已经解决了很多问题，但工程中我们往往会要求 Assembler 知道更多的 IFactory / Concrete Factory 配置关系，而且考虑到应用的 Plug && Play 要求，新写的 Concrete Factory 类型最好保存在一个额外的程序集里面，不涉及既有的已部署好的应用。为了不用反复引用并编译代码，而且可以把这些涉及运行维护的事务性工作交到系统管理员手里，一个常见的选择就是把 Assembler 需要加载的 IFactory/Concrete Factory 列表保存到配置文件里，相应的配置文件设计如下：

App.Config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
```

```
    <section name ="marvellousWorks.practicalPattern.factoryMethod.
      customFactories" type=
      "System.Configuration.NameValueSectionHandler"/>
  </configSections>
  <marvellousWorks.practicalPattern.factoryMethod.customFactories>
    <add
      key="MarvellousWorks.PracticalPattern.FactoryMethod.Classic.IFactory,
         MarvellousWorks.PracticalPattern.FactoryMethod"
      value="MarvellousWorks.PracticalPattern.FactoryMethod.Classic.
         FactoryA, MarvellousWorks.PracticalPattern.FactoryMethod"/>
  </marvellousWorks.practicalPattern.factoryMethod.customFactories>
</configuration>
```

由于这里 IFactory / Concrete Factory 仅仅是个映射关系列表，因此为了实现简介并不采用第 1 章配置部分介绍的自定义配置节的办法，而是直接使用.NET Framework 自带的 System.Configuration. DictionarySectionHandler 实现。

Assembler 则将提取 IFactory / Concrete Factory 的过程修改为通过配置实现，其代码如下：

C# 通过配置文件实现的 Assembler

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Collections.Specialized;
namespace MarvellousWorks.PracticalPattern.FactoryMethod.Classic
{
    public class Assembler
    {
        /// 配置节名称
        private const string SectionName =
           "marvellousWorks.practicalPattern.factoryMethod.customFactories";
        /// IFactory 在配置文件中的键值
        private const string FactoryTypeName = "IFactory";
        /// 保存“抽象类型/实体类型”对应关系的字典
        private static Dictionary<Type, Type> dictionary =
           new Dictionary<Type, Type>();

    static Assembler()
    {
        // 通过配置文件加载相关“抽象产品类型”/“实体产品类型”的映射关系
        NameValueCollection collection = (NameValueCollection)
           ConfigurationSettings.GetConfig(SectionName);
        for (int i = 0; i < collection.Count; i++)
        {
            string target = collection.GetKey(i);
            string source = collection[i];
            dictionary.Add(Type.GetType(target), Type.GetType(source));
        }
    }
    /// … …
}
```

Unit Test

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
```

```
using MarvellousWorks.PracticalPattern.FactoryMethod;
using MarvellousWorks.PracticalPattern.FactoryMethod.Classic;
namespace FactoryMethod.Test.Classic
{
    public class Client
    {
        private IFactory factory;
        public Client(IFactory factory) // 将 IFactory 通过 Constructor 方式注入
        {
            if (factory == null) throw new ArgumentNullException("factory");
            this.factory = factory;
        }

        public IProduct GetProduct() { return factory.Create(); }
    }

    [TestClass]
    class TestClient
    {
        [TestMethod]
        public void Test()
        {
            IFactory factory = (new Assembler()).Create<IFactory>();
            Client client = new Client(factory);  // 注入 IFactory
            IProduct product = client.GetProduct(); // 通过 IFactory 获取 IProduct
            Assert.AreEqual<string>("A", product.Name); // 配置中选择为 FactoryA
        }
    }
}
```

这样，即便在系统上线后，如果需要修改 IFactory 的具体类型，一样可以通过增加新的程序集，在生产环境中更新相关 IFactory 需要使用的具体工厂类型。这种方式下，客户程序成为一个可以动态加载的框架，外部机制很容易通过配置将新完成的程序集部署到运行环境中，而在经典的设计模式实现中，如果需要把模式工程化，很多时候需要借助外部的配置机制。

3.5 批量工厂

3.5.1 开发情景

实际项目中经常需要加工一批对象，这时候如果按部就班地采用单件生产模式来生成，效率上相对较低，最好专门设计独立的批量工厂。

其实，“工厂”这个名称体现的也是这个思想，现实中除了造船、造卫星、造航空母舰之类的工厂是单件生产外，大多数工厂都是一批批生产产品的，很难想象调用“啤酒 Factory”的 Create()方法的时候，要经过工厂的一系列处理，最后才产生一瓶啤酒，如果有人蹬个自行车准备批发一件啤酒的时候，要等 24 次序列处理，这也太累了吧。那我们看看如果准备批量生产产品，大体上需要哪些步骤？其步骤如图 3-5 所示。

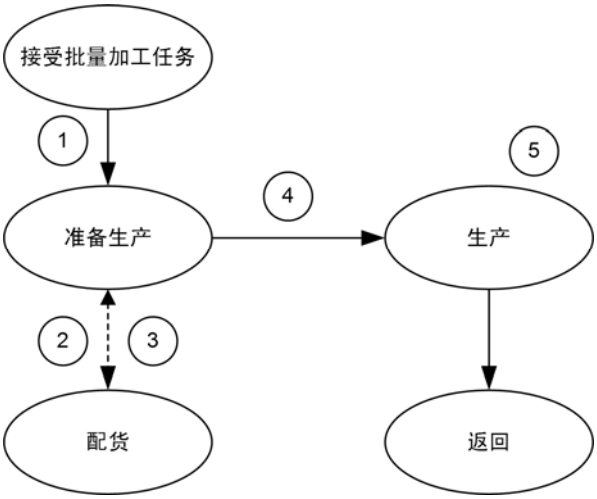


图 3-5 一个批量生产的处理工序示意图

实际项目中的开发过程也与此类似，但需要新增两个步骤。

- 准备生产：之前的实现中，实际加工对象的步骤其实就一个——new()，但之前必须编写很多访问配置文件，寻找抽象产品类型需要实例化的产品类型，项目中经常还需要增加很多其他例如记录日志、权限检查等之类的操作，这些步骤都被划分到“准备生产”过程里。
- 配货：这是个可选步骤。在此以实际生产过程做类比，比如我们生产 5000 部手机，但这 5000 部手机并不一定都是黑色的，照顾到女性顾客的需要，可能其中要包括 2000 部红色的，因此后续实际生产不是 for(int i=0; i<5000; i++)的单纯 new()，还需要增加一个配货步骤。

参考经典设计模式其他模式的实现，依据单一职责原则，需要增加三个对象。

- 指导生产的抽象类型（Director）：它告诉客户程序在保持 IFactory 的条件下，生产某类产品的数量，同时由于具体某类产品是由 Concrete Factory 决定的，因此客户程序实际获得 IFactory 还需要由 Director 告诉 Assembler 并进行替换。例如上面那个手机的情景，就需要 Director 在用完当前 BlackMobileFactory 生产 3000 部黑色手机后，将 Client 的 IFactory 换成 RedMobileFactory，继续生产 2000 部红色手机。
- Director 的每个步骤称为一个决策（Decision）：它包括两个信息，例如当决策生产 2000 部红色手机的时候，它包括数量（2000）和实际加工对象（RedMobileFactory）两项，客户程序 Decision 记载的批次数量 2000 调用 RedMobileFactory 的 Create()方法。
- 为批量的 IProduct 增加一个集合容器类型，项目中也可以直接使用.NET Framework 提供的既有集合类型（或泛型集合类型），同时修改 Concrete Factory 的加工方式，变返回

某个单独的 IProduct 为返回 IProduct 集合。

另外，参考前面的设计，由于项目中批量的加工任务不一定只有一类产品，因此把 Director 这个“军师”放到客户程序中的任务，还是交给 Assembly 完成好了。下面是其实现步骤。

3.5.2 定义产品类型容器

产品容器的对象结构如图 3-6 所示。

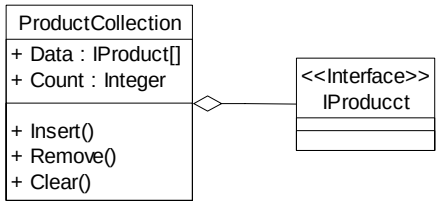


图 3-6 产品容器对象结构

示例代码如下：

C# 装载 IProduct 的容器类型

```
public class ProductCollection
{
    private IList<IProduct> data = new List<IProduct>();

    /// 对外的集合操作方法
    public void Insert(IProduct item) { data.Add(item); }
    public void Insert(IProduct[] items)    // 批量添加
    {
        if ((items == null) || (items.Length == 0)) return;
        foreach (IProduct item in items)
            data.Add(item);
    }
    public void Remove(IProduct item) { data.Remove(item); }
    public void Clear() { data.Clear(); }

    /// 获取所有 IProduct 内容的属性
    public IProduct[] Data
    {
        get
        {
            if ((data == null) || (data.Count == 0)) return null;
            IProduct[] result = new IProduct[data.Count];
            data.CopyTo(result, 0);
            return result;
        }
    }

    /// 当前集合内的元素数量
```

```
    public int Count { get { return data.Count; } }

    /// 为了便于操作，重载的运算符
    public static ProductCollection operator +(
    ProductCollection collection, IProduct[] items)
    {
        ProductCollection result = new ProductCollection();
        if (!((collection == null)||(collection.Count ==
           0)))result.Insert(collection.Data);
        if (!((items == null) || (items.Length == 0))) result.Insert(items);
        return result;
    }
    public static ProductCollection operator +(
    ProductCollection source, ProductCollection target)
    {
        ProductCollection result = new ProductCollection();
        if (!((source == null) || (source.Count == 0)))
            result.Insert(source.Data);
        if (!((target == null) || (target.Count == 0)))
            result.Insert(target.Data);
        return result;
    }
}
```

3.5.3 定义批量工厂和产品类型容器

在经典工厂方法模式上增加一个产品容器，如图 3-7 所示。

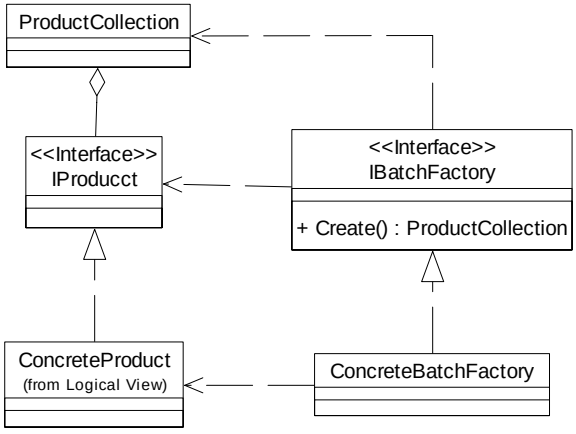


图 3-7 增加了产品容器的批量工厂对象结构

示例代码如下：

C# 定义并实现批量产品加工工厂

```
using System;
using MarvellousWorks.PracticalPattern.FactoryMethod;
namespace MarvellousWorks.PracticalPattern.FactoryMethod.Batch
{
```

```
    public interface IBatchFactory
    {
        /// <param name="quantity">待加工的产品数量</param>
        ProductCollection Create(int quantity);
    }

    /// 为了方便扩展提供的抽象基类
    /// <typeparam name="T">Concrete Product 类型</typeparam>
    public class BatchProductFactoryBase<T> : IBatchFactory
        where T : IProduct, new()
    {
        public virtual ProductCollection Create(int quantity)
        {
            if (quantity <= 0) throw new ArgumentException();
            ProductCollection collection = new ProductCollection();
            for (int i = 0; i < quantity; i++) collection.Insert(new T());
            return collection;
        }
    }

    /// 两个实体批量生产工厂类型
    public class BatchProductAFactory : BatchProductFactoryBase<ProductA> { }
    public class BatchProductBFactory : BatchProductFactoryBase<ProductB> { }
}
```

3.5.4 增设生产指导顾问——Director

Director 本身组织了一系列的 Decision 信息，它相当于指导客户程序调度不同实体工厂生产产品的指挥者，为了让客户程序的调用成为一个连续的过程，Director 可以采用迭代器组织每个 Decision，这样客户程序就可以用一种连续的线性方式完成每个 Decision 所记录的生产任务了，其代码如下：

C# 定义 Director 和 Decision

```
public abstract class DecisionBase
{
    protected IBatchFactory factory;
    protected int quantity;
    public DecisionBase(IBatchFactory factory, int quantity)
    {
        this.factory = factory;
        this.quantity = quantity;
    }

    public virtual IBatchFactory Factory { get { return factory; } }
    public virtual int Quantity { get { return quantity; } }
}

public abstract class DirectorBase
{
    protected IList<DecisionBase> decisions = new List<DecisionBase>();

    /// 实际项目中，最好将每个 Director 需要添加的 Decision 也定义在配置文件中，
    /// 这样增加新的 Decision 项都在后台完成，而不需要 Assembler 显式调用该方法补充
```

```
    protected virtual void Insert(DecisionBase decision)
    {
        if ((decision == null) || (decision.Factory == null) ||
           (decision.Quantity < 0))
            throw new ArgumentException("decision");
        decisions.Add(decision);
    }
    /// 便于客户程序使用增加的迭代器
    public virtual IEnumerable<DecisionBase> Decisions
    {
        get { return decisions; }
    }
}
```

3.5.5 由 Director 指导的客户程序

在完成了三个辅助类型（ProductCollection、Director、Decision）的设计之后，就可以设计由 Director 指导生产的新客户程序，示例代码如下：

C#

```
这个示例采用硬编码方式，没有通过 Assembler 获取 Director

class ProductADecision : DecisionBase
{
    public ProductADecision() : base(new BatchProductAFactory(), 2) { }
}
class ProductBDecision : DecisionBase
{
    public ProductBDecision() : base(new BatchProductBFactory(), 3) { }
}

class ProductDirector : DirectorBase
{
    public ProductDirector()
    {
        base.Insert(new ProductADecision());
        base.Insert(new ProductBDecision());
    }
}

class Client
{
    /// 实际项目中，可以通过 Assembler 从外部把 Director 注入
    private DirectorBase director = new ProductDirector();
    public IProduct[] Produce()
    {
        ProductCollection collection = new ProductCollection();
        foreach (DecisionBase decision in director.Decisions)
            collection += decision.Factory.Create(decision.Quantity);
        return collection.Data;
    }
}
```

Unit Test

```
[TestClass]
```

```
public class TestBatchFactory
{
    [TestMethod]
    public void Test()
    {
        Client client = new Client();
        IProduct[] products = client.Produce();
        Assert.AreEqual<int>(2 + 3, products.Length);
        for (int i = 0; i < 2; i++) Assert.AreEqual<string>("A",
           products[i].Name);
        for (int i = 2; i < 5; i++) Assert.AreEqual<string>("B",
           products[i].Name);
    }
}
```

通过上面的示例不难发现，虽然经典面向对象设计模式实现中并没有告诉我们如何批量创建对象，但通过增加新的监管对象（Director、Decision、ProductCollection）可以很容易地实现该需求。图 3-8 显示了该方式下的执行时序。

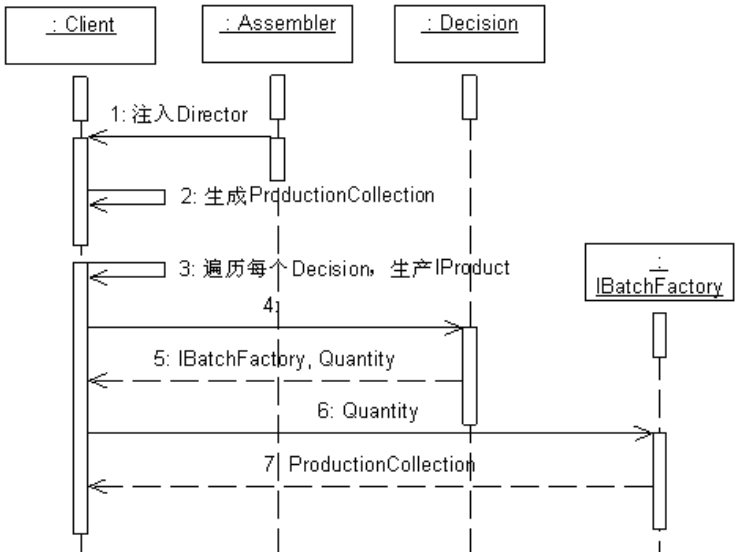


图 3-8 批量工厂的执行时序

随着本书后续设计模式的介绍，你可以发现其实 Decision 就是一个 Strategy，而 Director 本身可以作为客户程序的外部机制（Observer），异步指导客户程序的执行。

3.6 基于类型参数的 Generic Factory

随着泛型的使用，IProduct 的含义也被扩展了，项目中经常需要提供 IProduct<T>，甚至 IProduct<T1, T2, …>之类的泛型抽象产品类型。这些类型在类库的最外层往往被赋予了具体的类型参数，但在内层类库部分，或者在高度抽象的通用算法部分，往往会继续保持泛型的

抽象类型。为了保证工厂类型的加工过程的通用性，也需要设计具有泛型的 Generic Factory。

为了说明使用泛型工厂方法的技术需要，我们看看下面这个假想的情形。

假想情景

应用中需要一个自定义的链表结构，它被称为 ILinkList<T>（T 为 string, 值类型或实现了 IComparable 接口的类型），区别于我们通常向链表插入节点的时候已经按照键值作了升序或降序的处理，它本身就增加在链表的尾部，并提供了一个 Sort()方法，用于将自己内部节点进行排序。客户程序有时候用它保存一个学生历年的语文成绩，语法为 ILinkList<double>，有时候用它保存一个班级内学生姓名，语法为 ILinkList<string>，但其 Soft()方法是通用的。如果我们要实现 ILinkList<string>或 ILinkList<double>的实体工厂类型，有两种选择:

- 选择一：定义一个 ChineseMarkFactory 和一个 StudentNameFactory 工厂，分别生成 ILinkList<double>和 ILinkList<string>。
- 选择二：定义一个 LinkListFactory<T>，由客户程序根据需要提供具体的类型参数。

两种方式哪个更好呢？相信您也发现了——“视情况而定”，也就是上面说的：如果直接服务于最外层业务逻辑，那么选择第一种方式最好；如果继续用于内层抽象算法描述，那么选择第二种方式好像可继续保持算法的通用性。

其实在上面的批量对象工厂部分已经遇到了类似问题，就是因为可能返回 IProduct，可能返回 ProductCollection，所以我们定义了 IFactory 和 IBatchFactory，但既然它们令客户程序需要的类型选择延迟到子类，为什么就不能把非常通用的 new()也延迟到自己的子类呢？那些工厂都是 IFactory<T>的某个子类，大家都有一个通用的方法 T Create()，这样的代码多整齐。这样在统一体系之下，工厂方法模式部分的各个工厂可以实现的代码如下：

C#

```
/// 抽象的泛型工厂类型
public interface IFactory<T>
{
    T Create();
}

public abstract class FactoryBase<T> : IFactory<T>
    where T : new()
{
    /// 由于批量工厂的可能应用概率比较小，因此默认为实现单个产品的工厂
    public virtual T Create()
    {
        return new T();
    }
}

/// 生产单个产品的实体工厂
public class ProductAFactory : FactoryBase<ProductA> { }
public class ProductBFactory : FactoryBase<ProductB> { }
```

```
/// 生产批量产品工厂的抽象定义
public abstract class BatchFactoryBase<TCollection, TItem> :
FactoryBase<TCollection>
    where TCollection : ProductCollection, new()
    where TItem : IProduct, new()
{
    protected int quantity;
    public virtual int Quantity { set { this.quantity = value; } }

    public override TCollection Create()
    {
        if (quantity <= 0) throw new ArgumentException("quantity");
        TCollection collection = new TCollection();
        for (int i = 0; i < quantity; i++)
            collection.Insert(new TItem());
        return collection;
    }
}

/// 生产批量产品的实体工厂
public class BatchProductAFactory : BatchFactoryBase<ProductCollection,
  ProductA> { }
public class BatchProductBFactory : BatchFactoryBase<ProductCollection,
  ProductB> { }
```

上例是选择后一种实现方式的结果，那么相对前一种方式而言，它有什么好处呢？

- 最大的好处是整个体系只有一个根对象 IFactory<T>，整个体系统一。那么是不是上面的 IBatchFactory（Create()方法返回 ProductCollection）和 IFactory（Create()方法返回 IProduct）就不能做到只有一个根呢？也不是，它们都可以继承自 IFactory，只不过 Create()方法返回一个 object 就可以了，但这样不利于客户程序的调用，因为返回结果是 object。为了后续开发方便（也就是点一个点之后，通过 IntelliSense，就可以把具体产品类型的属性列出来），还需要增加一个强制类型转换，检查工作在运行态完成，而不是在变异过程完成。
- 另一个好处是隐性的，如果项目规模比较大，常常会感到别扭的地方就在于实际加工方法的命名，如果不做任何约束，开发人员可以写成 Create()、CreateInstance()或 Constructor()，还有各种名目的 CreateProducts()、CreateProductCollection()、CreateProductList()等，有些时候要靠猜才能确认。与其“大杂烩”，不如大家都规规矩矩地用一个统一的名称，当然，如果子类有特殊情况，可以自己增加，但最基本的方法名称大家都用一个。
- 如果您设计的是某种框架（也就是具有常说的那种 IOC 特质——“Don’t call me, I’ll call you”的类库），而且还常常会用到反射调用某个工厂生产产品，那么统一名称可以令您写出一个很通用的反射发现 Create()的机制，否则这个事情就麻烦了。

除此之外，笔者偏爱这种方式的原因更单纯，因为它是泛型的，看起来更有抽象的味道，比较 Cool。

3.7 委托工厂类型

相对而言，上面的实现还和经典的实现比较“贴”，毕竟加工的是一个类型对象，但在C#实现设计模式的时候，很多行为型和结构型模式需要的内容就是类的方法。受到语言的限制，Java 等很多语言必须用一个接口，然后由工厂把接口加工出来，但 C#有更经济的解决办法——委托。

你需要的是某个特征的方法、好定义、好规则（即委托），我找到合适的就给你挂上。

结合前面章节的介绍，您已经看到委托可以做很多非常方便的调用，无论是同步的还是异步的、无论是点对点调用还是组播调用，但是还有很多人认为委托是对面向对象概念的破坏，因为骨子里它就是个方法指针，但.NET 平台把它们包装为一个个对象，所以我们才可以很容易地实现很多异步机制和组播，而且执行过程中很多时候要比创建业务类型在效率上划算得多。“君子性非异也，善假于物也”，是否使用委托，您可根据需要和开发习惯自己选择。

这里提供一个最简单的 Delegate 工厂方法实现，上面提到的依赖注入机制、基于配置文件实现等工厂化实现技巧，与之非常类似，本书不赘述。其代码如下：

C#

```
/// 委托本质上就是对具体执行方法的抽象，它相当于 Product 的角色
public delegate int CalculateHandler (params int[] items);

class Calculator
{
    /// 这个方法相当于 Delegate Factory 看到的 Concrete Product
    public int Add(params int[] items)
    {
        int result = 0;
        foreach (int item in items)
            result += item;
        return result;
    }
}

/// Concrete Factory
public class CalculateHandlerFactory : IFactory<CalculateHandler>
{
    public CalculateHandler Create()
    {
        return (new Calculator()).Add;
    }
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    IFactory<CalculateHandler> factory = new CalculateHandlerFactory();
    CalculateHandler handler = factory.Create();
```

```
    Assert.AreEqual<int>(1 + 2 + 3, handler(1, 2, 3));
}
```

3.8 小结

经典工厂方法给我们设计应用一个很好的模式化 new()思维，通过引入新的对象，在很大程度上解放了客户程序对具体类型的依赖，其方法就是延迟构造到子类，但依赖关系是始终存在的，解放一个对象的时候等于把麻烦转嫁到另一个对象上。为此，工程中往往最后把推不掉的依赖关系推到配置文件上，也就是推到.NET Framework 上，这样经典的工厂方法模式往往在项目中被演绎成“工厂方法 + 依赖注入 + 配置文件访问”的组合方式。

考虑到性能因素，或者干脆为了省去多次调用的麻烦，工程中往往会有生成一批对象的需要，比如生成具有某些配置选项的多个线程，这时候就会用到批量对象工厂。但如前面章节所讨论的，写 Demo 和完成一个项目是两码事，其中一个关键的因素就是人，为了让整个实施过程中工厂的实现“中规中矩”，有时候要统一一些内容，泛型工厂能从根上约束整个工厂体系的加工方法命名，也许它看起来需要适应一段时间，但在使用上起码多了个选择。

最后就是工厂要为委托服务的问题，与 IProduct、Concrete Product 的模式不同，委托自己就是抽象类型，工厂在创建的时候不是 new()个类，关键是找到匹配的方法。

第 4 章

单件模式

4.1　经典回顾

4.2　线程安全的 Singleton

4.3　细节决定成败

4.4　细颗粒度 Singleton

4.5　自动更新的 Singleton

4.6　参数化的 Singleton

4.7　跨进程的 Singleton

4.8　Singleton 的扩展——Singleton-N

4.9　引入配置文件管理 Singleton

4.10　基于类型参数的 Generic Singleton

4.11　由工厂类型协助 Singleton 实例管理

4.12　小结

Ensure a class only has one instance, and provide a global point of access to.

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

Singleton（单件）模式是几个创建型模式中最独立的一个，它的主要特点不是根据客户程序调用生成一个新的实例，而是控制某个类型的实例数量——唯一一个。很多时候我们需要在应用中保存一个唯一的实例，比如您公司有很多同事都需要通过 MSN 上网发送、接收实时消息，为了集中控制公司所有的请求都通过代理服务器的 HTTP 端口发送出去，虽然在那台机器的 80 端口上您已经安排了非常多的工作，不仅仅是 MSN，还有网页浏览、B2B 的 Web Service 调用等，但最终一个端口在同一时间内只能有一个进程打开。为了防止这种“一窝蜂”情况下，某个进程打开后没有很好关闭 HTTP 端口，导致其他人都无法使用的情况，您最后可能要决定用一个进程控制它，如果其他人需要服务——排队，而这个唯一进程也就是那台机器计算机软件环境中的一个 Singleton（单件）。数据库中自增字段也有类似的情况，它不会因为 A、B 的调用在数据库不同会话中（无论是专用服务器模式还是共享服务器模式）出现 A：1000、1001、1002，B：1000、1001、……9970 的情况，1000 ~ 9970 间的每一个号都可能出现在 A 或 B 中的某一个会话里，这时候那个自增字段的序号生成部分也是一个 Singleton。

实现单件的方式有很多，但大体上可以划分为两种。

- 外部方式：客户程序在使用某些全局性（或语义上下文范围内的全局性）的对象时，做些“Try-Create-Store-Use”的工作，如果没有，就自己创建一个，但是仍把它搁在那个全局的位置上，如果原本有，就直接拿一个现成的。其代码如下：

C# 客户程序保证的 Singleton 方式

```
class Target{}

class Client
{
   private static IList<Target> list = new List<Target>();
   private Target target;

   public void SomeMethod()
   {
      if (list.Count == 0)
      {
         target = new Target();
         list.Add(target);
      }
      else
         target = list[0];
   }

   public Target Instance { get { return target; } }
}
```

Unit Test

```
[TestClass]
public class TestClientSideSingleton
```

```
{
   [TestMethod]
   public void Test()
   {
      Client client1 = new Client();
      client1.SomeMethod();
      Client client2 = new Client();
      client2.SomeMethod();
      Assert.AreEqual<int>(client1.
        Instance.GetHashCode(), client2.Instance.GetHashCode());
   }
}
```

- 内部方式：类型自己控制生成实例的数量，无论客户程序是否 Try 过了，类型自己就是一个实例，客户程序使用的都是这个现成的唯一实例。

相比较而言，外部方式实在不“靠谱”，毕竟应用中的类型不止一个，即便某个类型可以恪守这种先 Try 的方式，仍然可能有其他类型会用到它，很难要求所有相关类型都做这种唯一性检查，最终结果就是无法唯一。内部方式把干扰因素排除在类型之外，相对更保险一些。随着集群、多处理器和多核处理器的展开，想通过简单的类型内部控制，保持其真正的 Singleton 越来越难，确切来说，这个单件的“单”字是有上下文和语义范围限制的。

本章除了介绍经典 Singleton 之外，还将根据实际技术趋势介绍如何根据不同类型生产环境实现某种语义范围的 Singleton。

4.1 经典回顾

使用 Singleton 模式的意图就是要保证一个类只有一个实例，同时提供客户程序一个访问它的全局访问点。进一步考虑，由于类可以被继承，因此 Singleton 要保证“对外”（非直接或间接继承关系）提供一个统一的访问点。最初看到这个需求您可能会直接写一个静态的全局变量，虽然声明为静态是提供全局访问的一个技巧，但它也仅解决了部分问题，没有真正限制客户程序实例化的数量。

实际上，Singleton 模式要做的就是通过控制类型实例的创建，确保后续使用的都是之前创建好的一个实例，通过这样的一个封装，客户程序就无须知道该类型实现的内部细节。

在逻辑模型上，Singleton 模式很直观（在整个 23 个模式中是最简单的了），如图 4-1 所示。

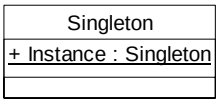


图 4-1 单件对象的静态结构

其中的参与者就只有 Singleton 自己，它的职责是定义一个 Instance 操作（类成员或类方

法、类属性），作为客户程序访问 Singleton 的一个，而且是唯一的一个实例。客户程序如果需要调用 Singleton 类型的其他实例方法，也仅可以通过 Instance 完成。其代码如下：

C++ 《Design Patterns : Elements of Reusable Object-Oriented Software》的描述

```
// Declaration
class Singleton {
public:
   static Singleton* Instance();
protected:
   Singleton();
private:
   static Singleton* _instance;
}

// Implementation
Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance() {
   if (_instance == 0) {
      _instance = new Singleton;
   }
   return _instance;
}
```

C# 把静态方法 Instance()转换成静态属性后的 C#描述

```
public class Singleton
{
   private static Singleton instance;   // 唯一实例
   protected Singleton() { }   // 封闭客户程序的直接实例化
   public static Singleton Instance
   {
      get
      {
         if (instance == null)
            instance = new Singleton();
         return instance;
      }
   }
}
```

让我们回头看看这段 C#代码：

- 这个 Singleton 类仅有一个公共类属性——Instance，区别于普通的类，Singleton 类仅有一个 Protected 的构造函数，客户程序不可以直接实例化 Singleton，即类属性 Instance 就是客户程序访问“private static Singleton instance”的唯一入口。
- 其中的那个 if 完成的控制部分则是控制实例数量的部分，只要 instance 被实例化过它就不会再生成新的实例，直接把一个既有的实例反馈给客户程序。

大面上讲这段代码已经可以满足最初 Singleton 模式的设计要求，在大多数情况下那段代码也可以很好地工作。但在多线程环境下，这种实现方式存在很多潜在的缺陷，一个最直接的问题就位于 if 部分，当多个线程几乎同时调用 Singleton 类的 Instance 静态属性的时候，instance 成员可能还没有被实例化，因此它被创建了多次，而且最终 Singleton 类中保存的是

最后创建的那个实例，各个线程引用的对象不同，这违背了我们“唯一实例”的初衷。

这种情况往往是我们在一般调试中最头疼的事情，因为不像其他逻辑错误，可以反复重现，设计多线程应用的一个原则就是“不要臆测每个并发任务的执行次序”，但这里要模拟出两个线程同时进入，很多时候要靠多次测试中的运气。毕竟很多时候相对线程实际执行前那个等待 CPU 调度并被实际执行的过程而言，这里的 new Singleton() 可能太短暂了。

另外，对于之前从事过 C++开发的同行，您可能觉得静态私有成员 instance 应该命名为_instance 或 s_instance，虽然很长时间里笔者也经常使用那个命名方式，但本书准备全部采用《Design Guideline》的方式，如果与您的编码习惯有所冲突，请您见谅。

在综合执行效率和线程同步的考虑后，我们采用一个 double check 的方式修正上面的代码如下：

C# 增加 Double Check 后的 Singleton

```
public class Singleton
{
   protected Singleton() { }
   private static volatile Singleton instance = null;
   /// Lazy 方式创建唯一实例的过程
   public static Singleton Instance()
   {
      if (instance == null)          // 外层 if
         lock (typeof(Singleton))    // 多线程中共享资源同步
            if (instance == null)    // 内层 if
               instance = new Singleton();
      return instance;
   }
}
```

为区别上面的代码，这里有几处是需要注意的：

- 虽然是多线程环境，但如果没有那个外层 if，客户程序每次执行的时候都需要先被 lock 住 Singleton 类型，但实际在绝大多数情况下，运行起来的时候这个 instance 并不为空，每次都锁定 Singleton 类型，效率太差，这个 lock 很可能就成了整个应用的瓶颈。
- lock 加内层的 if 部分等于组成了一个相对线程安全的实例构造小环境。
- 一旦唯一的一个实例在那个线程安全的“小环境”里被创建之后，后续新发起的调用都无须经过那个 lock 部分，直接在外层 if 判断之后就可获得既有的唯一实例引用。
- 其中 volatile 关键字也是个要点，它表示字段可能被多个并发执行线程修改。声明为 volatile 的字段不受编译器优化（一般情况下默认的编译优化假定由单个线程访问）的限制，这样可以确保该字段在任何时间呈现的都是最新的值，也就是在被 lock 之后，如果还没有真正完成 new Singleton()，新加入的线程看到的 instance 都是 null。

上面是经典的 Singleton 用 C#参照 C++和 Java“照猫画虎”的实现，凭心而论有些繁琐。

由于.NET Framework 在设计上自身就有很多类似并发控制和单实例的访问要求，所以也有很多更简洁但同时也更“靠谱”的实现方式。就像我们前面介绍的，只要保证好对外的这个 Instance 静态属性不变的前提，客户程序可以继续使用它。下面介绍如何基于.NET Framework 对 Singleton 的内部实现进行改造。

4.2 线程安全的 Singleton

上面的 Double Check 方式实现虽然基本解决了多线程情况下的 Singleton 问题，但从我们的经验来看，那个构造过程为什么不放到静态构造函数里？毕竟它是整个类的构造部分。一个原因是编译器可能会“好心地”把静态成员 instance 的构造次序重排。C++在静态成员的构造过程中存在一些多义性，这应该也是最初《设计模式：可复用面向对象软件的基础》那本书里没有采用静态构造函数的一个原因；C#相对好很多，因为通过指定的编写方法可以明确地规定静态成员的构造顺序。考虑到 Singleton 实例仅由一个内部静态成员保存，如果它本身在构造过程中不需要借鉴其他外部机制或不需要准备很多构造参数，也可以直接用下述方式定义：

C#

```
class Singleton
{
   private Singleton() { }
   public static readonly Singleton Instance = new Singleton();
}
```

您可能觉得这次的实现比之前的要简单得多，但它确实是多线程环境下，C#实现 Singleton 一个非常棒的实现方式，怎么做到的呢？

- 它省去了上面示例中那个 laze 构造过程，由于 Instance 是类的公共静态成员，因此相当于它会在类第一次被用到的时候被构造，同样的原因也就可以省去把它放在静态构造函数里的过程。
- 这里实例构造函数被彻底定义为私有的，所以客户程序和子类无法额外构造新的实例，所有的访问通过公共静态成员 Instance 获得唯一实例的引用，符合 Singleton 的设计意图。
- 至于怎么保证多线程 Singleton 的安全，从 C#层面不好判断，我们不妨从 IL 层面来看看，代码如下：

IL

```
.class private auto ansi beforefieldinit Singleton extends
[mscorlib]System.Object
{
   .method private hidebysig specialname rtspecialname static void .cctor()
     cil managed
   {
      .maxstack 8
      L_0000: newobj instance void Singleton::.ctor()
      L_0005: stsfld class Singleton Singleton::Instance
```

```
        L_000a: ret
    }
 … …
    .field public static initonly class Singleton Instance
}
```

首先，这里有个 beforefieldinit 的修饰符，它告诉 CLR 只有当这里的静态成员（或称之为字段，Field）在静态构造函数执行后才生效，因此即便有很多线程试图引用 Instance，也需要等静态构造函数执行并把静态成员 Instance 实例化之后才可以使用。（这里 IL 层面的静态构造是 C#编译器隐式添加的。）

其次，虽然静态属性 Instance 被定义为公共的，但它是只读的（IL 这里的 initonly），因此一旦创建，它就不能被任何线程修改，也就不用作 Double Check。

因此，很简短的 C#代码就实现了一个 Singleton，并且它是线程安全的。很 Cool。

但这种异常简练的方式相对最经典的 Lazy 构造方式或 Double Check 方式的优势都有个适用范围，如果工程中这个类有些静态方法，它们很可能会在调用静态属性 Instance 之前就被客户程序用到，同时哪怕类型只有一个实例也是内存中的庞然大物，那么我们没必要在刚开始时就为其腾出很大一块地方，如果后面始终没有对其进行调用，这不就白白浪费了么？而且除非当前进程终止，否则 GC 几乎没有机会来回收这块“空置不用”的内存。

下面我们用这种方式来实现一个 Singleton 常用的情景——计数器，示例代码如下：

C#

```
public class Counter
{
    private Counter() { }
    public static readonly Counter Instance = new Counter();

    private int value;
    public int Next { get { return ++value; } }
    public void Reset() { value = 0; }
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    Counter.Instance.Reset();
    Assert.AreEqual<int>(1, Counter.Instance.Next);
    Assert.AreEqual<int>(2, Counter.Instance.Next);
    Assert.AreEqual<int>(3, Counter.Instance.Next);
}
```

4.3 细节决定成败

上面的 Singleton 类型继承自 System.Object，但实际项目中很多时候它都会继承自某些

类型，或者实现了某些接口，这里要特别注意一些因素，因为它们会打破我们费尽心思实现的 Singleton，笔者将这种情况称为“Singleton 变质”。下面是最常见的两个导致“Singleton 变质”的情景：

- 不要实现 ICloneable 接口或继承自其相关的子类，否则客户程序可以跳过已经隐蔽起来的类构造函数。下面的示例说明通过 ICloneable 接口的克隆过程导致私有构造函数失效，CLR 通过内存结构的复制生成了一个新的实例，最终导致并非单一实例存在。

C# 会导致变质的情景

```
public class BaseEntity : System.ICloneable
{
    public object Clone()   // 对当前实例进行克隆
    {
        return this.MemberwiseClone();  // 例如采用这种方式克隆
    }
}

public class Singleton : BaseEntity
{
    // ... ...
}
```

- 严防序列化。对于远程访问，往往需要把复杂的对象序列化后进行传递，但是序列化本身会导致 Singleton 特性的破坏，因为序列化事实上完成了 Singleton 对象的拷贝。所以不能对期望具有 Singleton 特性的类型声明 SerializableAttribute 属性。下面是一个错用情形——为 Singleton 类型增加了 SerializableAttribute 属性：

C# 会导致变质的情景

```
[Serializable]
public class Singleton
{
// ... ...

    /// 把 Singleton 实例通过二进制串行化为字符串
    public static string SerializeToString(Singleton graph)
    {
        MemoryStream memoryStream = new MemoryStream();
        formatter.Serialize(memoryStream, graph);
        Byte[] arrGraph = memoryStream.ToArray();
        return Convert.ToBase64String(arrGraph);
    }

    /// 通过二进制反串行化从字符串回复出 Singleton 实例
    public static Singleton DeserializeFromString(string serializedGraph)
    {
        Byte[] arrGraph = Convert.FromBase64String(serializedGraph);
        MemoryStream memoryStream = new MemoryStream(arrGraph);

        return (Singleton)formatter.Deserialize(memoryStream);
   }
}
```

4.4 细颗粒度 Singleton

4.4.1 背景讨论

4.2 节讨论的是线程安全的 Singleton 实现，但项目中我们往往需要更粗或更细颗粒度的 Singleton，比如某个线程是长时间运行的后台任务，它本身存在很多模块和中间处理，但每个线程都希望有自己的线程内单独 Singleton 对象，其他线程也独立操作自己的线程内 Singleton，这样线程级 Singleton 的实例总数 = 1（每个线程内部唯一的一个）* N（线程数）= N。

.NET 程序可以通过把静态成员标示为 System.ThreadStaticAttribute，以确保它指示静态字段的值对于每个线程都是唯一的，但这对于 Windows Form 程序很有效，对于 Web Form、ASP.NET Web Service 等 Web 类应用则不适用，因为它们是在同一个 IIS 线程下分割的执行区域，客户端调用时传递的对象是在 HttpContext 中共享的，也就是说，它本身不可以简单地通过 System.ThreadStaticAttribute 实现。不仅如此，使用 System.ThreadStaticAttribute 也不能很潇洒地套用前面的内容写成：

C#

```
[ThreadStatic]
public static readonly Singleton Instance = new Singleton();
```

按照.NET 的设计要求，不要为标记该属性的字段指定初始值，因为这样的初始化只会发生一次，因此在类构造函数执行时只会影响一个线程。在不指定初始值的情况下，如果它是值类型，可依赖初始化为其默认值的字段，如果它是引用类型，则可依赖初始化为 null。也就是说，在多线程情况下，除了第一个实例外，其他线程虽然也期望通过这个方式获得唯一实例，但其实获得的是一个 null，不能用。

4.4.2 解决桌面应用中细颗粒度 Singleton 问题

对于 Windows Forms，可以通过 System.ThreadStaticAttribute 比较容易地告诉 CLR 其中的静态唯一属性 Instance 仅在本线程内部静态，但麻烦的是如何构造它。正如上面背景介绍部分所说的，不能把它放到整个类的静态构造函数里，也不能直接初始化，那么怎么办？还好，如果这里不适用 4.2 节介绍的那个很 cool 的实现，我们就退回到最经典的那个 lazy 方式加载 Singleton 实例的方法。您可能觉得这样的线程不安全了吧？那种实现方式确实不是线程安全的，但我们这里的 Singleton 构造本身就已经运行在一个线程里面了，用那种不安全的方式在线程内部实现自己“一亩三分地”范围内 Singleton 的对象反而安全了。新的实现如下：

C#

```
public class Singleton
{
    private Singleton() { }

    [ThreadStatic]  // 说明每个 Instance 仅在当前线程内为静态
    private static Singleton instance;

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
                instance = new Singleton();
            return instance;
        }
    }
}
```

Unit Test

```
/// 每个线程需要执行的目标对象定义
/// 同时在它内部完成线程内部是否 Singleton 的情况
class Work
{
    public static IList<int> Log = new List<int>();
    /// 每个线程的执行部分定义
    public void Procedure()
    {
        Singleton s1 = Singleton.Instance;
        Singleton s2 = Singleton.Instance;
        // 证明可以正常构造实例
        Assert.IsNotNull(s1);
        Assert.IsNotNull(s2);
        // 验证当前线程执行体内部两次引用的是否为同一个实例
        Assert.AreEqual<int>(s1.GetHashCode(), s2.GetHashCode());
        //登记当前线程所使用的 Singleton 对象标识
        Log.Add(s1.GetHashCode());
    }
}

[TestClass]
public class TestSingleton
{
    private const int ThreadCount = 3;
    [TestMethod]
    public void Test()
    {
        // 创建一定数量的线程执行体
        Thread[] threads = new Thread[ThreadCount];
        for (int i = 0; i < ThreadCount; i++)
        {
            ThreadStart work = new ThreadStart((new Work()).Procedure);
            threads[i] = new Thread(work);
        }
        // 执行线程
        foreach (Thread thread in threads) thread.Start();
```

```
        // 终止线程并作其他清理工作
        // ... ...

        // 判断是否不同线程内部的 Singleton 实例是不同的
        for (int i = 0; i < ThreadCount - 1; i++)
            for (int j = i + 1; j < ThreadCount; j++)
                Assert.AreNotEqual<int>(Work.Log[i], Work.Log[j]);
    }
}
```

下面我们分析单元测试代码说明的问题：

- 在 Work.Procedure()方法中，两次调用了 Singleton 类的 Instance 静态属性，经过验证是同一个 Singleton 类实例。同时由于 Singleton 类的构造函数定义为私有，所以线程（客户程序）无法自己实例化 Singleton 类，因此也实现了该模式的设计意图。
- 通过对每个线程内部使用的 Singleton 实例登记并检查，确认不同线程内部其实掌握的是不同实例的引用，因此满足我们需要实现的细颗粒度（线程级）的需求。

4.4.3 解决 Web 应用中细颗粒度 Singleton 问题

上面用 ThreadStatic 虽然解决了 Windows Form 的问题，但对于 Web Form 应用而言并不适用，原因是 Web Form 应用中每个会话的本地全局区域不是线程，而是自己的 HttpContext，因此，相应的 Singleton 实例也应该被保存在这个位置。实现上我们只需要做少许的修改，就可以完成一个 Web Form 下的细颗粒度 Singleton 设计：

这里的 Web Form 应用包括 ASP.NET Application、ASP.NET Web Service、ASP.NET AJAX 等相关应用。但示例并没有在.NET Compact Framework 和.NET Micro Framework 的环境下进行过验证。

C#

```
public class Singleton
{
    /// <summary>
    /// 足够复杂的一个 key 值，用于和 HttpContext 中的其他内容相区别
    /// </summary>
    private const string Key = "marvellousWorks.practical.singleton";
    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            // 基于 HttpContext 的 Lazy 实例化过程
            Singleton instance = (Singleton)HttpContext.Current.Items[Key];
            if (instance == null)
            {
                instance = new Singleton();
                HttpContext.Current.Items[Key] = instance;
            }
```

```
            return instance;
        }
    }
}
```

Unit Test

```
using System;
using System.Web;
using MarvellousWorks.PracticalPattern.SingletonPattern.WebContext;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace SingletonPattern.Test.Web
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            Singleton s1 = Singleton.Instance;
            Singleton s2 = Singleton.Instance;
            // 确认获得的 Singleton 实例引用确实已经被实例化了
            Assert.IsNotNull(s1);
            Assert.IsNotNull(s2);
            // 确认两个引用调用的是同一个 Singleton 实例
            Assert.AreEqual<int>(s1.GetHashCode(), s2.GetHashCode());
            // 显示出当前 Singleton 实例的标识，用于比较与其他
            // HttpContext 环境下的 Singleton 实例其实是不同的实例
            instanceHashCode.Text = s1.GetHashCode().ToString();
        }
    }
}
```

在浏览器中的执行效果如图 4-2 所示。

浏览器效果

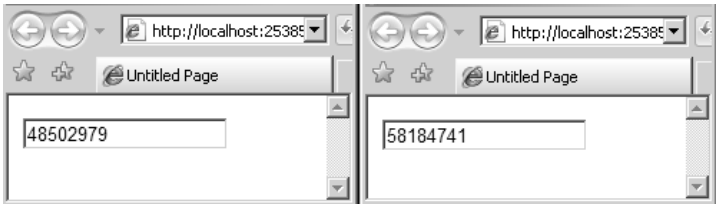


图 4-2 浏览器中的执行效果

同上，这段单元测试验证了 Web Form 下的细颗粒度 Singleton，将唯一实例的存储位置从当前线程迁移到 HttpContext，一样可以实现细颗粒度的 Singleton 设计意图。

4.4.4 更通用的细颗粒度 Singleton

就像我们在第 2 章说的一样，如果你是一个公共库或是公共平台的设计者，您很难预料到自己的类库会运行在 Windows Form 还是 Web Form 环境下，但 Singleton 模式作为很多公共机制，最常用的包括计数器、时钟等，又常常会成为其他类库的基础，尤其当涉及业务领

域逻辑的时候，很难在开发过程就约定死运行的模式。怎么办？

借助我们在第 2 章实现的公共工具，不妨作一个 2 in 1 的细颗粒度 Singleton（听起来有点像早年的任天堂游戏卡），不过就像我们提到的面向对象设计的单一职责原则一样，把两者合并在一起会产生一些比较难看的冗余代码，不过 Singleton 与其他设计模式有个很显著的区别——它不太希望被外部机制实例化，因为它要保持唯一性，因此一些常用的依赖倒置技巧在这里又显得不太适用。这里实现一个稍有些冗余的 Web Form + Windows Form 2 in 1 的细颗粒度 Singleton，如图 4-3 所示。

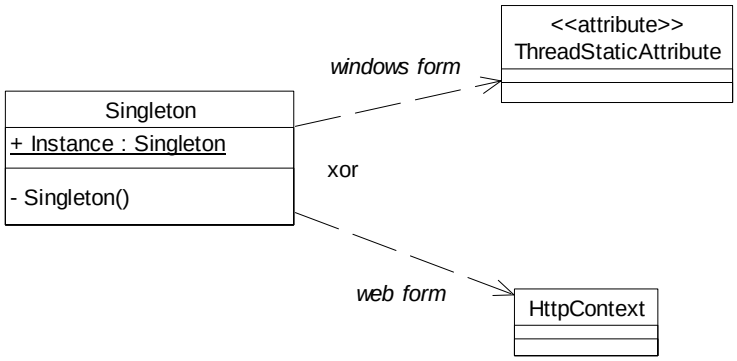


图 4-3 细颗粒度单件模式的静态结构

示例代码如下：

C#

```
using System;
using System.Web;
using MarvellousWorks.PracticalPattern.Common;
namespace MarvellousWorks.PracticalPattern.SingletonPattern.Combined
{
    public class Singleton
    {
        private const string Key = "marvellousWorks.practical.singleton";
        private Singleton() { }    // 对外封闭构造
        [ThreadStatic]
        private static Singleton instance;

        public static Singleton Instance
        {
            get
            {
                // 通过之前准备的 GenericContext 中非官方的方法
                // 判断当前执行模式是 Web Form 还是非 Web Form
                // 本方法没有在 .NET 的 CF 和 MF 上验证过
                if (GenericContext.CheckWhetherIsWeb())    // Web Form
                {
                    // 基于 HttpContext 的 Lazy 实例化过程
                    Singleton instance =
                      (Singleton)HttpContext.Current.Items[Key];
```

```
                if (instance == null)
                {
                    instance = new Singleton();
                    HttpContext.Current.Items[Key] = instance;
                }
                return instance;
            }
            else  // 非 Web Form 方式
            {
                if (instance == null)
                    instance = new Singleton();
                return instance;
            }
        }
    }
}
```

4.5 自动更新的 Singleton

似乎 Singleton 从经典实现开始就一直是一个持续存在的对象，但现实中持久不变的内容还是放到数据库、XML、报文里面更好些，内存中的对象很多时候还是要被刷新的，工程中 Singleton 的那个唯一实例很多时候也有类似的需求。实现上更新的触发原因很多，有可能是来自外部的通知，也可能是系统内部的时钟，有时候也经常同时提供外部、内部的通知机制。

4.6 参数化的 Singleton

很多时候 Singleton 对唯一实例的构造过程有些“莽撞”，因为要么 new()一个无参数的构造函数，要么按照指定参数序列构造一个硬编码参数方式的构造函数，这确实不够灵活，即便可以把很多判断逻辑放到静态构造函数或那个 Lazy 构造过程，但毕竟这都是在开发过程中定义的。

C# 硬编码方式实现僵化的参数化 Singleton

```
public class Singleton
{
    /// 参数化构造函数
    private string message;
    private Singleton(string message) { this.message = message; }
    public string Message { get { return this.message; } }

    /// 硬编码方式实现的参数化构造函数
    private static Singleton instance;
    public static Singleton Instance
    {
        get
        {
            if (instance == null)
```

```
            lock(typeof(Singleton))
               if (instance == null)
                  if ((DateTime.Now.DayOfWeek == DayOfWeek.Sunday) ||
                     (DateTime.Now.DayOfWeek == DayOfWeek.Saturday))
                     instance = new Singleton("weekend");
                  else
                     instance = new Singleton("work day");
         return instance;
      }
   }
}
```

既然直接硬编码太僵化，很多时候第一直觉是参考“依赖注入”的模式，把需要的参数注入到唯一实例中，如本书第 1 章介绍的，注入在.NET 环境下有四种方式。

- 构造函数方式：这种方式在 Singleton 行不通，因为不允许客户程序控制 Singleton 类型的实例化过程。
- Setter 方式：通过暴露出公共实例属性（public 或某些情况下 internal），客户程序就可以修改唯一实例的内容，实现参数化不成问题，但同时这种方式有点太过“自由”了，无论哪里的客户程序只要可以看到相关属性就能修改，这很容易失控。这种方式备选。
- 接口方式：侵入性太强，而且接口方法的实现全部都在类内部，本身就是硬编码过程，解决不了问题。
- Attributer 方式：将通用的 Attribte 作为桥梁，可以在外部将需要的信息注入 Singleton 类型，供构造过程参考。这是可行的，但其代价相对于 Singleton 模式而言有点大。这种方式备选。

既然外部注入方式感觉上都不是很理想，还是让 Singleton“内强”好了，一个不错的选择是借助访问配置系统（配置文件、ini 文件或数据库等）。例如上面的那个问题，我们可以通过把 message 的内容配置在 App.Config 文件里来解决，其代码如下：

App.Config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="parameterizedSingletonMessage" value="Hello world"/>
  </appSettings>
</configuration>
```

C#

```
public class Singleton
{
   /// 其他处理
   /// ……

   /// 通过访问配置实现的参数化 Singleton 实例构造
   private static Singleton instance;
   public static Singleton Instance
   {
```

```
        get
        {
            if (instance == null)
                lock(typeof(Singleton))
                    if (instance == null)
                    {
                        /// 读取配置并进行实例化
                        string key = "parameterizedSingletonMessage";
                        string message = ConfigurationManager.AppSettings[key];
                        instance = new Singleton(message);
                    }
            return instance;
        }
    }
}
```

UnitTest

```
Assert.AreEqual<string>("Hello world", Singleton.Instance.Message);
```

4.7 跨进程的 Singleton

以上所有的讨论都是限制在一个进程内的，无论是那个很精简的线程安全实现还是细颗粒度的实现，但随着软件可用性要求的进一步提升，很多关键应用都需要通过 Cluster 实现应用的 Scale out。这下麻烦更大了，因为之前仅仅在一个线程内部，为了实现一个线程安全的唯一实例就需要大费周折地做个Double Check或取巧地采用.NET Framework的内置机制，但如果应用运行在不同服务器，甚至跨越网络，则很难控制那个 instance==null 的判断。我们看看 Lazy 构造、Double Check 和那个很精简的方式谁可以胜任：

- 经典 Singleton 模式中的那个 Lazy 构造过程：多个进程执行某些任务本身就是个并行性的工作，在单个线程或进程内部，判断 instance == null 可能仅仅是一瞬间的事情，但我们在上面讨论过仍然可能导致两个线程一起进入这个 if 部分，在跨线程环境下这个问题更明显，判断自己内存中 instance == null，然后再看看别人是不是也 instance==null 才敢实例化，这个时间相对 instance = new Singleton()而言往往要长得多，因此更不“靠谱”。这个方式暂不考虑。
- Double Check 方式：发现自己进程内 instance==null，然后就要把自己锁起来，接着去看看别人那边是不是做好了，如果已做好，直接引用那边的；如果没做好，就让那边也先锁上，然后自己再踏踏实实地 new Singleton()。听上去不错，不过别忘了，那边很可能也是这么想的。如果两边都同时这么想，您想将出现什么情况？

凭借开发经验您不难看出，图 4-4 里虚线的调用其实是完成不了的，但是两边又都持僵持的态度，满足一方持有部分争用资源但又试图锁定对方争用资源的典型情形，很容易形成死锁，相信您也很不愿看到这种情况的发生。

- 那个精简的方式呢？还是算了吧，它压根儿就不考虑对方是否已经构造好了，直接就 new Singleton()了。

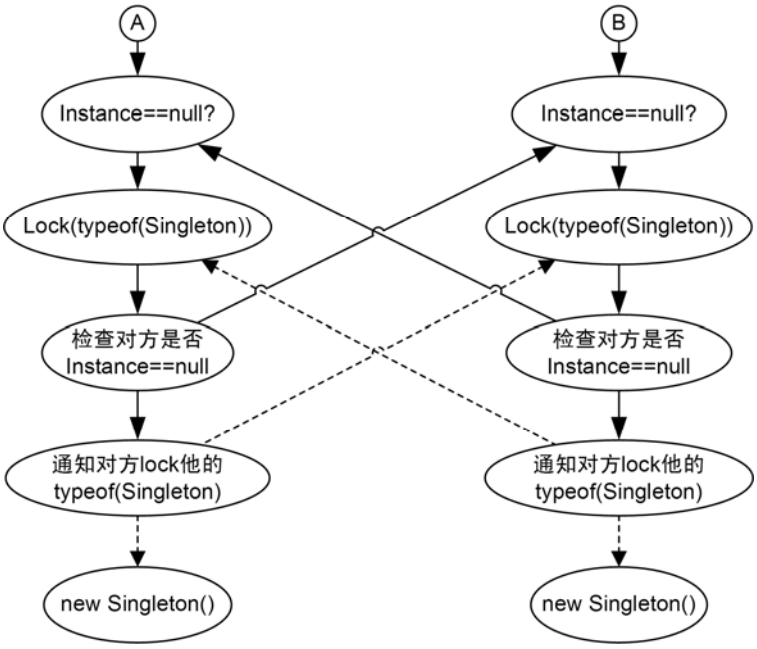


图 4-4 跨进程单件模式的静态结构

考虑解决办法前，我们还是看看一般一个 Cluster 是怎么做的，因为如果要做负载均衡，很多时候它要记录 Cluster 中每个节点的响应情况，然后按照配置好的比例分配负载，其中一个关键的因素就是有个共享机制，也就是要有 3rd（第三方）。这里也一样，当两者（甚至更多）内部都保存一个具有 new Singleton()能力的类型后，常规的 Singleton 模式很难控制它们的实现，这时候不如直接剥夺它们 new Singleton()的能力，两边作为一个代理向本进程内部的实体提供 Singleton 实体的引用，把构造工作抽象出去。新的设计如图 4-5 所示。

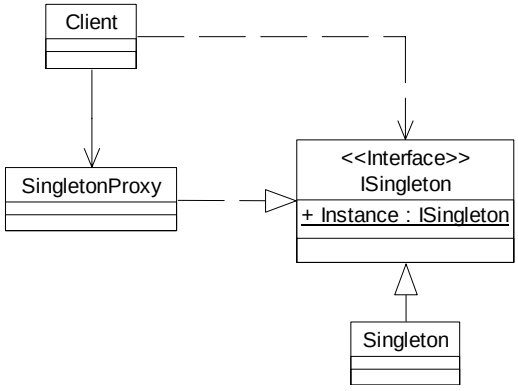


图 4-5 增加了远程代理的单件对象静态结构

ISingleton 需要部署在客户节点，构造 Singleton 类型节点，代理类所在节点和客户程序

部分（它们可能位于同一台机器上，也可能运行在分布式网络环境下）。SingletonProxy 作为每个进程内部的 ISingleton 访问代理，负责传达客户程序对 ISingleton 类型的调用请求。至于那个真正的 Singleton 类，则可以按部就班地套用 Double Check 或精简型方式，将自己锁在进程的内存空间里踏踏实实地创建并暴露出那个唯一实例。如果是 Cluster 环境，Singleton 可能运行在 Controller 上，也可以运行在某个 Controller + Member 的机器上；如果仅仅是一台机器的多个进程间通信，它可以“寄生”在某个并行进程内部，或者自己在一个单独的进程里。

另外，还有一个半 Singleton 的方式，因为 Singleton 类型中可变的内容本质上是它的成员和它自身需要操作的数据，如果把这些内容通过某种方式固定到一个持久层，即便在每个进程内部 new 出一个 Singleton 实例，其实达到的效果是相同的。在现代的企业应用中，这部分工作往往是借助数据库数据复制或同步实现的，它相当于 Singleton 可变内容的后移，虽然往往可以达到类似的效果，但因为实例不唯一，所以与经典设计模式中 Singleton 的定义有出入，这里笔者就叫它“半”Singleton——Semi-Singleton。

4.8 Singleton 的扩展——Singleton-N

Singleton 模式明确定义了唯一的实例数量，但发掘其设计意图不难发现，它的根本目的是保证某些操作的一致性，比如更新计数器、更新整个应用级的某些全局性信息。就像我们可能正慢条斯理地完成一件件工作，但 12 月 28 日上司突然要求您在元旦前提交 10 个部门的总结，而且每份总结所需的调研时间为 2 天零 7 个小时。虽然套用以前的模板，通过填写关键业绩可以在 1 小时内完成 10 份总结，但是从总体上来说您还是来不及在元旦前提交，因为只有您一个人在有效时间内才能完成大量的工作。使用 Singleton，很多时候也会导致应用出现这样的瓶颈，那么我们可以考虑作一个扩展，让 Singleton 内部产生 N 个实例，大家分担工作量，这就不至于一个人累垮而且还拖累大家。您可能会质疑，这和一般的类有什么不同？不就等于按需 new()了么？其实是有些区别的，这里的 N 是个相对固定的数量，而且也是最多允许出现的数量。比如：N=5，那么 1、2、3、4 个实例都是可以的，5 也可以，但当 5 个实例都处于忙碌状态、再次进行 new()的时候，系统就会告诉您“超出服务容量了”。作为一种扩展，这里将它称为 Singleton-N 模式。

您可能注意到了上面描述的几个关键点：

- 最多 N。
- “如果处于忙碌状态”。
- 按需 new()。

其实，从某种程度上来说，Singleton-N 是 ObjectPool 模式的一个预备模式，它除了完成

创建型模式最一般的 new()之外，还要负责检查和数量控制，而且往往在一个多线程环境下进行，因此，在设计上要为它增加一些“助手”。至于实现方法，经典的 Lazy 构造方式不适合并发环境，PASS；精简的那个方式，不适合构造多个对象，虽然它可以构造一个数组，但一句话的时候构造不了数组中的每个实例，如果把那个精简的代码展开放到静态构造函数里一个个进行实例化，又不能满足“按需 new()”的性能要求，所以也 PASS；最后看来，Double Check 方式暂时还有改造的空间。为了不打破 Double Chcek 框架，并且依据单一职责原则，我们要增加一个类 WorkItemCollection，它必须满足下面的条件：

- 它本身是个集合类型。
- 最多存放某种类型的 N 个实例。
- 集合中的每个实例都可以通过状态标示自身处于忙碌状态还是处于闲暇状态，同时可以根据外部的反馈，修正自己的状态。
- 可以告诉外界，是否还有空位，或者是否可以找出一个“热心”的闲着的对象。
- 至于 new()么，WorkItemCollection 还是别管了，否则容易让 Singleton-N 类型的构造过程失控。

为了简洁，下面的示例中没有在 WorkItemCollection 中增加锁机制，Singleton 类型实现上也省略了 Double Check，仅仅实现了一个 SingletonN 实例容器的功能性部分。

4.8.1 定义具有执行状态的抽象对象

定义具有执行状态的抽象对象示例代码如下：

C#

```
/// 实例的执行状况
public enum Status
{
   Busy,   // 被客户程序占用
   Free    // 没有被客户程序占用
}

interface IWorkItem
{
   Status Status { get; set;}
   void DeActivate();  // 放弃使用
}
```

4.8.2 定义相应的 Singleton-N 实例集合

定义相应的 Singleton-N 实例集合的示例代码如下：

C#

```
class WorkItemCollection<T>
```

```
    where T : class, IWorkItem
{
    /// 定义最多允许保存的实例数量 N
    protected int max;
    protected IList<T> items = new List<T>();
    public WorkItemCollection(int max) { this.max = max; }

    /// 外部获得 T 类型实例的入口
    public virtual T GetWorkItem()
    {
        if((items == null) || (items.Count == 0)) return null;
        // 可能的话，对外反馈一个现成实例
        foreach(T item in items)
            if (item.Status == Status.Free)
            {
                item.Status = Status.Busy;
                return item;
            }
        return null;    // 虽然有现成的实例，但都处于忙碌状态，所以返回 null
    }

    /// 新增一个类型
    public virtual void Add(T item)
    {
        if (item == null) throw new ArgumentNullException("item");
        if (!CouldAddNewInstance) throw new OverflowException();
        item.Status = Status.Free;  // 默认状态
        items.Add(item);
    }

    /// 判断是否可以增加新的实例
    public virtual bool CouldAddNewInstance {
       get { return (items.Count < max); } }
}
```

4.8.3 在基本 Singleton 模式实现的框架下引入实例集合

示例代码如下：

C#

```
public class SingletonN : IWorkItem
{
    private const int MaxInstance = 2;    // 定义 Singleton-N 的这个 N
    private Status status = Status.Free;  // 初始状态
    public void DeActivate() { this.status = Status.Free; }
    public Status Status
    {
        get { return this.status; }
        set { this.status = value; }
}

    private static WorkItemCollection<SingletonN> collection =
       new WorkItemCollection<SingletonN>(MaxInstance);
    private SingletonN() { }
    public static SingletonN Instance
```

```
    {
        get
        {
            // 在基本实现框架不变的情况下，引入集合实现 Singleton-N 的多个实例对象管理
            SingletonN instance = collection.GetWorkItem();
            if (instance == null)
                if (!collection.CouldAddNewInstance)
                    return null;
                else
                {
                    instance = new SingletonN();
                    collection.Add(instance);
                }
            instance.Status = Status.Busy; // 激活使用
            return instance;
        }
    }
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    SingletonN s1 = SingletonN.Instance;
    SingletonN s2 = SingletonN.Instance;
    SingletonN s3 = SingletonN.Instance;

    Assert.IsNull(s3);  // 超出容量，所以不能获得实例引用
    Assert.AreNotEqual<int>(s1.GetHashCode(), s2.GetHashCode());
    // 两个不同实例
    s1.DeActivate();
    s3 = SingletonN.Instance;
    Assert.IsNotNull(s3);   // 有了空间，所以可以获得引用
    s2.DeActivate();
    Assert.IsNotNull(s3);   // 有了空间，所以可以获得引用
    // s3 虽然获得了新的引用，但其实是之前已经创建的某个现成的
    Assert.IsTrue((s3.GetHashCode() == s1.GetHashCode()) ||
        (s3.GetHashCode() == s2.GetHashCode()));
}
```

从上面的示例不难看出，Singleton-N 相对传统 Singleton 模式而言，增建了多个实例的管理和执行调度，但与 Object Pool 不同的是它没有实现对象销毁机制，算是一个半成品。实际上应用中如果出现需要使用 Singleton-N 的情况往往和设计上的类型间职责划分有一定关系，如果方法间本身不存在明显的耦合情况，那么完全可以把计算上比较耗时的部分提取出新的对象，它们本身并不需要 Singleton；但如果方法间本身难于拆解，或者出于安全或封装的需要，不可以对外暴露出中间结果，本身计算比较复杂，涉及调用共享争用资源的时隙又很小，这便是 Singleton-N 大展身手的时候了。

4.9 引入配置文件管理 Singleton

上面的很多示例其实都有可通过配置完成的部分，而且工程中也需要这种生产环境的管

理机制，例如：Singleton-N 的这个 N 可能就需要按照服务器执行能力和业务负载进行调整；之前在定义“自动更新的 Singleton”部分，如果采用内置时钟方式，那么最好将超时时间也写到配置里；参数化 Singleton 部分本身也可能把很多初始化参数写在某个配置节或配置元素集合里。

不管涉及的配置访问部分工作有多少种，建议在工程中采用或扩展.NET Framework 自己的配置对象系统，这里有个关键角色 ConfigurationManager，它相当于应用和配置系统的桥梁，通过它可以直接或间接地获得.NET Framework 提供的“4 件套”配置对象（ConfigurationSectionGroup、ConfigurationSection、ConfigurationElement、ConfigurationElementCollection），一些简单的配置信息可放在 AppSetting 部分，如图 4-6 所示。

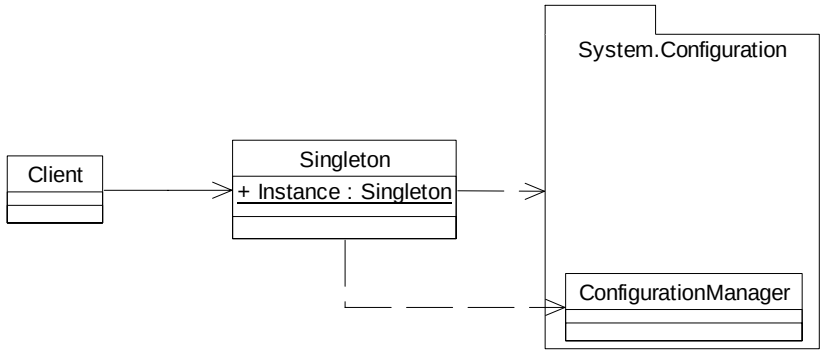


图 4-6 基于配置的单件对象静态结构

4.10 基于类型参数的 Generic Singleton

有时候，项目中需要批量地产生很多“不是那么规矩”的 Singleton 类型，也就是说，虽然内部提供了唯一的 Singleton 实例，但偶尔还是可以允许某些外部机制额外构造它的实例。这么做有什么意义呢？

- 首先，毕竟实现 Singleton 的方法相对很固定，虽然本章演绎出了很多扩展，但其实基本过程就是踏踏实实地生成一个实例，如果在类的继承关系上提供一个模板式的实现方式，很多时候可以节省客户代码；
- 其次，项目大了后很多时候就要讲规矩，比如本章访问的唯一实例全都是通过静态公共属性进行的。您可以检查之前参与的项目，称之为 GetInstance()、Instance()、Singleton()的方法估计也林林总总，问题和第 3 章说的 Factory 方法一样，必要的时候可以定一个接口，并实现一个抽象基类，统一所有 Singleton 类型的访问点。由于事先需要部分开放子类的构造过程，因此这种 Singleton 有点“不那么规矩”。
- 其三，便于客户程序使用。那个唯一的实例作为静态的服务访问点，但是并不排斥客户

程序从实例层面操作 Singleton 类的方法。（虽然实现方式不同，不过效果上您可以类比 string 类，它既提供静态的 Equals()方法，也支持实例级 Equals()方法。）

下面是这种准 Singleton 的实现方式代码：

C# 定义抽象部分

```
/// 定义一个非泛型的抽象基础
/// 否则类型约束上，只能 T : class, new()，相对而言约束不够严谨。
public interface ISingleton{}

public abstract class SingletonBase<T> : ISingleton
   where T : ISingleton, new()
{
   protected static T instance = new T();
   public static T Instance { get { return instance; } }
}
```

C# 批量加工出一批访问点的很“规矩”的 Singleton 类型

```
/// 利用现有基础可以快速地构造出一批具有 public static T Instance
/// 类型特征的准 Singleton 类型，从整体上统一 Singleton 方式访问的入口
class SingletonA : SingletonBase<SingletonA> { }
class SingletonB : SingletonBase<SingletonB> { }
class SingletonC : SingletonBase<SingletonC> { }
```

Unit Test

```
[TestMethod]
public void Test()
{
   /// 使用传统 Singleton 方式访问，一样可以保证唯一性
   SingletonA sa1 = SingletonA.Instance;
   SingletonA sa2 = SingletonA.Instance;
   Assert.AreEqual<int>(sa1.GetHashCode(), sa2.GetHashCode());
   /// 也可以绕过 Instance 静态属性，直接实例化那些准 Singleton 类型
   SingletonA sa3 = new SingletonA();
   Assert.IsNotNull(sa3);
   Assert.AreNotEqual<int>(sa1.GetHashCode(), sa3.GetHashCode());
}
```

4.11 由工厂类型协助 Singleton 实例管理

如果按照经典方式来定义 Singleton，那么客户程序无法动态生成 Singleton 类型实例，使用其他外部工厂类也于事无补，因为构造函数对外部封闭。但如果适当放宽一下限制，则可以通过工厂实现 Concrete Singleton 类型与抽象 ISingleton 延迟加载，实现客户程序与 Concrete Singleton 的依赖倒置。比如下述情景：

- 客户程序对于各 Concrete Singleton 类型是通过程序集引用获得的，因此可以把各个 Concrete Singleton 类型与相关的 Singleton Factory 定义在同一个程序集里，Concrete Singleton 做到 internal 可见即可，这样也可达到客户程序无法直接 new()的目的。例如上面的泛型 Singleton 示例中，SingletonA、SingletonB 其实就是在.NET 默认约定的访问

控制符——internal 中被定义的。

- 通过在 Concrete Singleton 类型的构造函数上增加某些安全检查，例如：权限检查属性、Role 检查等，事实上也可以封闭外部客户程序的实例化过程。代码如下：

C# && Unit Test

```
class SingletonFactory
{
   public ISingleton Create() { return SingletonA.Instance; }
}

[TestMethod]
public void TestSingletonFactory()
{
   ISingleton singleton = (new SingletonFactory()).Create();
   Assert.IsNotNull(singleton);
}
```

4.12 小结

作为创建型模式中一个很独特的模式，Singleton 更多地强调可构造的类型数量和唯一实例的方式，工程上实现 Singleton 的挑战主要来自于它的运行环境和需要保持 Singleton 特质的颗粒度。随着对应用高可用性要求的提出，部署环境中 Singleton 往往需要在更广范围内保持其特性，如果仅通过应用的设计来解决这类问题往往又非常复杂（喧宾夺主了），为降低技术实施风险可以借助数据库、共享进程等方式近似解决分布式环境下 Singleton 的实现。

经典的设计模式实现很多时候都是基于许多理想化的假设，例如所有类型只有无参数的构造函数，但项目中往往不是这样的；另外，由于 Singleton 的构造过程比较封闭，因此很多时候借助配置机制把参照信息提供给 Singleton 类型。

理想方式和项目实际往往有冲突，对于开发人员而言最直接的一个冲突恐怕就是重复编码的工作量了，稍微放宽一些 Singleton 类型的构造限制有时候还是可以考虑的。

最后，抛开各种讨论，我们还是重温一下 Singleton 模式在.NET 平台一个最简洁但又相对线程安全的实现方式：

C#

```
class Singleton
{
   private Singleton() { }
   public static readonly Singleton Instance = new Singleton();
}
```

您可能觉得看上去这和使用静态类几乎没有什么区别，确实有点，不过这里没有把其他实例方法写进去，同时这个类可以被继承，而不像静态类那样一步到终点，不留进一步扩展的余地。不过静态类也有自己的优势，效率相对要高一些。

第 8 章

适配器模式

8.1 说明

8.2 经典回顾

8.3 进一步扩展适配范围的组适配器

8.4 Adapter——Adapter 互联模式

8.5 用配置约定适配过程

8.6 XML 数据的专用适配机制

8.7 小结

8.1 说明

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces..

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

作为结构型模式的第 1 章，我们先回顾一下之前的创建型模式主要采用了哪些方法避免客户程序与 new()的具体对象直接依赖呢？其实关键就是借助接口，让客户程序依赖于抽象。但现实中我们定义的接口在一个环境中可以用，在另一个环境中则不一定合乎规则；反过来也一样，毕竟现在很少有项目完全重头建立自己的类库，很多时候会用第三方一些现成的中间件，它们的接口在我们的项目里不一定合用。这种情况下一个很现实的问题就是如何改造这些接口，主要有三种方式：

- 抛开它们，自己重新做，当您觉得那些东西“太烂了”或“技术太陈旧”的时候，这可能是首先考虑到的。
- 修改它们，这个可能需要更大的勇气，尤其当您的这个提议意味着很多资金、他人工作量、技术实施风险的时候，这个方案常常是阻力重重。
- “借他山之石”，但从外部做些适应性修改，确保旧接口（既有接口）通过外面增加的新对象适合新接口（目标接口）的需要。

由于 C#更新得比较快，很多时候虽然我们觉得第一种方式比较耗费资源，但为了一个看上去“合心”的框架，选中它的比例并不小；第三种方案也许更适合一些规模大些的方案，毕竟之前四五年的投资不是说放弃就放弃的。第二个方案也许是最头疼的，大部分开发人员恐怕都会避开它。上面的话题谈大了，我们回到具体某个接口上，当我们发现某个已经实施过的接口不合用的时候，也许不能把它删掉或进行修改（无论是自己项目的还是外部引入的），因为这牵涉很多其他部分，此时第三种方案看上去不错。这时候适配器模式就派上用场了，它的概念过程如图 8-1 所示。

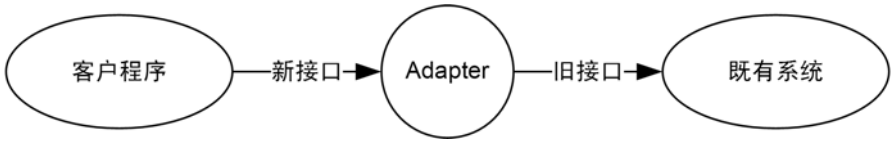


图 8-1 适配器的典型应用场景

不难看出适配器主要有三个作用：

- 完成旧接口到新接口的转换。
- 将“既有系统”进行封装，逻辑上客户程序不知道“既有系统”的存在，将变化隔离在 Adapter 部分。

- 如果客户程序需要迁移，仅需要在 Adapter 部分做修改。

8.2 经典回顾

适配器模式的意图就是通过接口转换，使本来不兼容的接口可以协作。

无论进行 WebForm 开发还是 WinForm 开发，相信您都积攒了一些常用的类，在具体项目中它们可以被反复重用，但也常常碰到因为接口参数、返回值类型不匹配需要做小幅修改的情况。适配器模式给我们另一个思路，不是对现有类和接口进行改造，而是通过增加新对象完成转换工作。这个模式主要用于下述环境：

- 我们需要复用一些已经存在的类，但这些类提供的接口不满足我们新的需要，尤其之前的这些类是我们花了很大成本完成的，它们经过项目考验表现确实一直很稳定。
- 我们封装一些公共的类，它们会经常被重用，但我们不确定目标环境需要何种接口。
- 新的接口可能需要的接口功能是之前几个类所能提供的功能，而且新接口与它们不兼容。

前面两个要求比较好理解，一个是别人给您的，另一个是您计划给别人的，对于第三个要求，您可能觉得和单一职责原则相违背，为什么一个接口需要其他好几个不兼容接口的支持呢？这是出于封装的需要，注意这里不是把那几个接口的方法“一字排开”，而是根据新接口的需要，特定方法会使用既有接口的某（几）个方法，仅从新接口的功能角度看，它的职责还是单一的。

实现这个目标，一般有两种方式：一是通过多重继承让适配器类具有既有类型的特点，同时也可以根据客户程序的需要，满足新接口的需要，一般称之为类适配器；另一种是在适配器里保存一个既有类型的引用（保存一个它的实例或在方法中根据需要 new()出一个实例），它自身按照客户程序的要求，实现新接口，这种方式被称为对象适配器。类适配器和对象适配器的静态结构分别如图 8-2、图 8-3 所示。

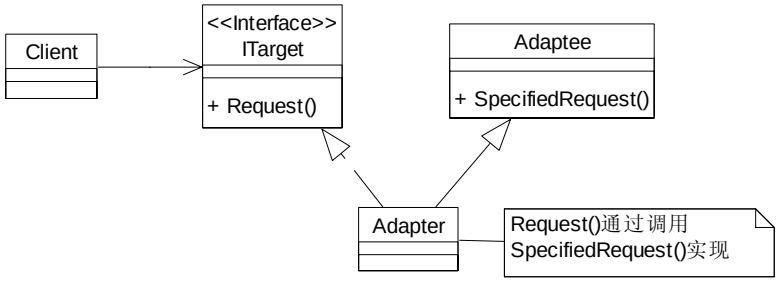


图 8-2 类适配器模式

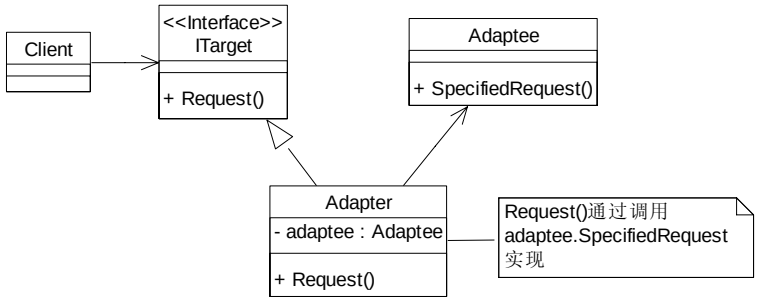


图 8-3 对象适配器模式

这其中主要包括以下几个角色。

- Target（ITarget）：客户所期待的接口。
- Adaptee：需要被适配的类型。
- Adapter：适配器，主要完成 Adaptee 到 ITarget 的转换。

从这个结构不难看出，类适配器和对象适配器最大的区别在于对既有类型的使用上，这和我们如何扩展新功能的思路相关。对象适配器相对而言是被推荐的方式，因为这种方式结构更加松。下面我们看一下两种方式的示例，其代码如下：

C# 定义 ITarget 和 Adaptee

```
public interface ITarget
{
    void Request();
}

public class Adaptee
{
    void SpecifiedRequest() { }
}
```

C# 类适配器

```
public class Adapter : Adaptee, ITarget
{
    public void Request()
    {
        // 其他操作
        base.SpecifiedRequest();    // 调用 Adaptee
    }
}
```

C# 对象适配器

```
public class Adapter : ITarget
{
    private Adaptee adaptee;    // Adaptee 对象
    public void Request()
    {
        // 其他操作
        adaptee.SpecifiedRequest();    // 调用 Adaptee
```

```
    }
}
```

通过上面的示例，结合 C#语言特点，我们可以看出它们的区别，如表 8-1 所示。

表 8-1

Class Adapter	Object Adapter
基于继承概念	基于对象组合的思路
适配器之前不能继承自其他类，Target 只能是接口形式的 ITarget	Target 可能是 ITarget（接口）、TargetBase（抽象类），甚至是实体类，只要 Adapter 满足不继承两个或两个以上类的限制
可以覆盖 Adaptee 的某些方法	无法覆盖 Adaptee 的方法
虽然不可以适配子类，但可以通过覆盖修改某些方法，部分情况下可以达到适配子类同样的效果	不仅可以适配 Adaptee，还可以适配 Adaptee 的任何子类

本章下面的内容将以对象适配器方式扩展为主，个别情况会采用多接口实现方式，尽量避免类适配器。

相信在开始学习 C#、C++等面向对象语言的时候，您总是先接触到继承概念，老师或书籍总是通过继承让我们相信面向对象开发可以大大节省编码量，因为你如果需要一个新特性的类型时，扩展并添加一两个方法就可以了。但经过几年工作后，我们发现继承不总是那么好，尤其当您的设计处于继承结构的中下层的时候，每次修改的时候总是牵扯面太大。然后我们借鉴 Delphi 早期类库设计的技巧，把底层类的大部分方法都定义为虚方法，修改上面任何一个不合适层，结果随着项目代码规模的扩大，开发期最灵活的设计往往造成运行环境最难排查的问题。最后，我们接受了基于接口编程的思路，设计模式也一直倡导组合优于继承的概念，尤其在 C#、Java 等语言彻底不支持多继承之后，我们才把目光从“堆积”类变成“捆绑”接口。如果在一开始学习的时候就有人提示一下多好。

8.3 进一步扩展适配范围的组适配器

上面介绍的是经典的适配器模式，下面我们要对其作些扩展。首先，您在用数码伴侣么？我没有，不过我看同事用过（我是个喜欢纯粹的人，不喜欢组合功能的产品）。想想看它的功能是什么？通过读卡器，可以把多种存储载体上的内容保存在一个硬盘里。

我们项目中也有类似的情况，比如我们在开发产品时，客户可能会把相关的参数和日志保存在 ORACLE、SQL Server、DB2、MySQL 等数据库，但我们几乎不会用到类似 ORACLE Sequence 之类的数据库产品相关特性，只需要 SQL-92 就可以了，这时候我们有两个选择，一是做一个类似 Enterprise Library Db Block 的数据访问块，然后为每个数据库做个 DataAdapter，另一种就是完全基于抽象数据类型做个公用的数据访问块，至于它执行的 SQL 语句，选择 4 个数据库能支持的最小集即可。哪个方案更好一些呢？它们各有优缺点。第一

种方案优势是效率高，可以充分利用数据库产品的特性；第二种方案是一个中庸的方案，如以后增加了 Sybase 数据库，只要它符合 ADO.NET 2.0 的类型系统，一样可以使用，但需要在配置文件上登记新的 Provider Name。

不难看出，采用后者其实要做的是一个同时适应一组 Adaptee 的 Adapter，不妨叫它“组适配器”。逻辑上它应该实现的结构（采用对象适配器方式，限于单继承的要求，类适配器不能用）如图 8-4 所示。

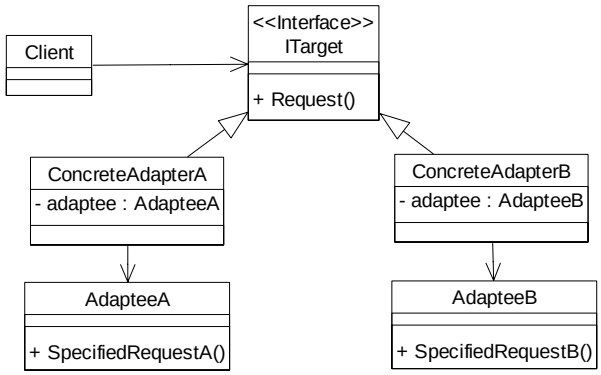


图 8-4 1∶N 的简单适配器模式

在此似乎少了些什么，回想一下前面的创建型模式，是不是少了个工厂类型？为了清晰，忽略掉下面的 AdapteeA 和 AdapteeB，增加一个简单工厂类型即可。调整后的组适配器的结构如图 8-5 所示。

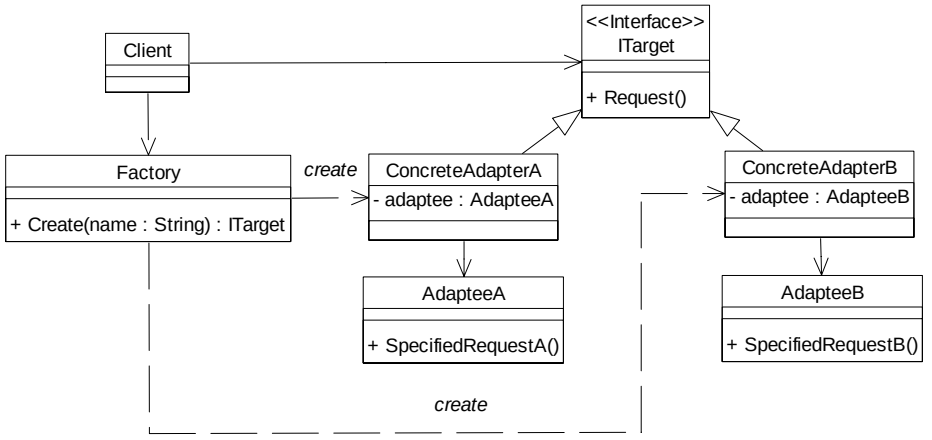


图 8-5 1∶N 的组适配器模式

下面我们来看一个组适配器的示例，其代码如下：

C# 两个接口各异的 Adaptee

```
public class OracleDatabase
```

```
{
    /// specified request
    public string GetDatabaseName() { return "oracle"; }
}

public class SqlServerDatabase
{
    /// specified request
    public string DbName { get { return "SQL Server"; } }
}
```

C# 定义抽象 Adapter 和两个具体 Adapter

```
public interface IDatabaseAdapter
{
    string ProviderName { get;}
}

public class OracleAdapter : IDatabaseAdapter
{
    private OracleDatabase adaptee = new OracleDatabase();
    public string ProviderName { get { return adaptee.GetDatabaseName(); } }
}

public class SqlServerAdapter : IDatabaseAdapter
{
    private SqlServerDatabase adaptee = new SqlServerDatabase();
    public string ProviderName { get { return adaptee.DbName; } }
}
```

C# 定义工厂类型

```
public sealed class DatabaseAdapterFactory
{
    class DatabaseAdapterMapper
    {
        private static IDictionary<string, Type> dictionary =
            new Dictionary<string, Type>();
        /// 其中的内容可以从配置文件提取
        static DatabaseAdapterMapper()
        {
            dictionary.Clear();
            dictionary.Add("oracle", typeof(OracleAdapter));
            dictionary.Add("sqlserver", typeof(SqlServerAdapter));
        }

        /// 根据数据库类型，获得指定的 Adapter 类型名称
        public Type this[string name]
        {
            get
            {
                if (!dictionary.ContainsKey(name))
                    throw new NotSupportedException(name);
                return dictionary[name];
            }
        }
    }

    private DatabaseAdapterMapper mapper = new DatabaseAdapterMapper();
    /// 根据数据库类型返回 DatabaseAdapter
    public IDatabaseAdapter Create(string name)
```

```
    {
        return (IDatabaseAdapter)(Activator.CreateInstance(mapper[name]));
    }
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    DatabaseAdapterFactory factory = new DatabaseAdapterFactory();
    IDatabaseAdapter adapter = factory.Create("oracle");
    Assert.AreEqual<string>("oracle", adapter.ProviderName);
    Assert.AreEqual<Type>(typeof(OracleAdapter), adapter.GetType());
}
```

通过上面的分析不难看出，适配一组 Adaptee 相对经典适配器模式而言，主要工作在于如何组织和构造每个具体 Adapter 类型，不过之前的创建型模式我们已经做了很多的练习了。

8.4 Adapter——Adapter 互联模式

8.4.1 分析

随着松散耦合概念越来越普遍，加之各种标准化协议的出现，很多时候我们面临的不仅仅是既有类型和新接口的关系，而常常会出现 Adapter 对 Adapter 的情况，比如：我们用 C# 从 SQL Server 中提取一些数据，经过加工后再保存到 ORACLE，这个过程其实是 Adapter 之间的连接。从部署的层次看，这种问题一般都是通过系统管理员配置解决的，比如数据库的同步、活动目录的组策略下发等，但如果这些 Adapter 就是应用自己的定义的内容，不同 Adaptee 适配之后可以基于相同的 ITarget 访问，我们是否可以为它们之间的互联提供更简便的方式呢？

先看一下常规思路，两个 Adapter（客户程序逻辑上把它们看作两个 Adaptee）互联，我们会增加第三个对象完成这个转发过程，既然从逻辑角度看它们的结构一样，那么它们之间的调用就变成如图 8-6 所示的结构。

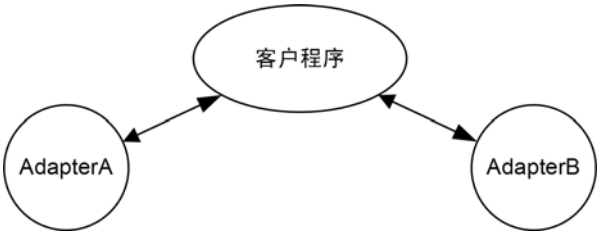


图 8-6 具有多适配器的应用结构

不过这样似乎还不够好，毕竟客户程序需要同时与两个对象频繁地打交道，而这个打交道的过程非常固定，且都是基于 ITraget 接口的操作，其中有可能需要对中间结果进行临时

的缓存，这时候不妨抽象出一个名为 EndPoint 的对象，由它专门负责与这两个（甚至更多）Adapter 打交道，结构调整如图 8-7 所示。

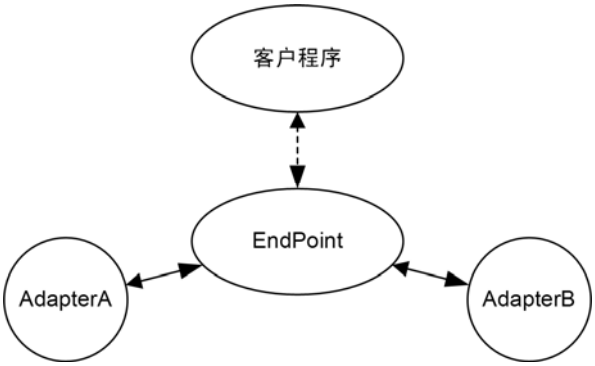


图 8-7 增加了 Endpoint 后的多适配器应用的结构

这时候客户程序与 Endpoint 之间的交互可能是同步的也可能是异步的，至于调用方法的描述相对也简单了，对于同步调用，需要客户程序从 Endpoint 获得两个 Adapter 的引用，然后调用即可；如果是异步方式，可以把客户程序提交给 Endpoint 的任务，表示成如下方式。

- Direction：谁调用谁？A-> B 还是 B->A？因为大家都是一样的接口，所以称呼为 Adapter Invoke Adater 不能区分出调用关系。
- Request Method：发起方执行的方法。
- Response Method：响应方执行的方法。
- Need Relay：是否需要传递 Request Method 计算的中间结果。

为了能体现出效果，我们给上一个示例中的 OracleDatabase 和 SqlServerDatabase 增加读、写数据的功能，对 IDatabaseAdapter 也相应作些调整，其代码如下：

C# 调整后的 Adapter 和 Adaptee

```
public class OracleDatabase
{
    public string GetDatabaseName() { return "oracle"; }
    public int Select() { return (new Random()).Next(); }
    public void Add(int data) { }
}
// SqlServerDatabase 也作类似地修改

public interface IDatabaseAdapter
{
    string ProviderName { get;}
    int GetData();
    void SetData(int data);
}

public class OracleAdapter : IDatabaseAdapter
{
```

```
    private OracleDatabase adaptee = new OracleDatabase();
    public string ProviderName { get { return adaptee.GetDatabaseName(); } }
    public int GetData() { return adaptee.Select(); }
    public void SetData(int data) { adaptee.Add(data); }
}
// SqlServerAdapter 也作类似的修改
```

8.4.2 方式 1：客户程序直接调度 Adapter

其示例代码如下：

Unit Test

```
[TestMethod]
public void Test()
{
    DatabaseAdapterFactory factory = new DatabaseAdapterFactory();
    IDatabaseAdapter a = factory.Create("oracle");
    IDatabaseAdapter b = factory.Create("sqlserver");
    b.SetData(a.GetData());
}
```

不难发现，在这种方式下，客户程序最好依赖于一个构造型类型，否则客户程序将直接和具体的 Adapter 类型产生依赖。如果每个 Adapter 在使用前必须执行一些配置工作，可能单独使用 Factory 还不够，最好是在 Create()方法里嵌入一个 Builder 来完成 Adapter 的组装工作。相对而言这种方式实现起来比较“廉价”，对于客户程序而言，需要依赖适配器组机制。

8.4.3 方式 2：基于标准约定调度 Adapter

其示例代码如下：

Unit Test

```
class Endpoint
{
    private IList<IDatabaseAdapter> adapters = new List<IDatabaseAdapter>();
    public Endpoint()
    {
        DatabaseAdapterFactory factory = new DatabaseAdapterFactory();
        adapters.Add(factory.Create("sqlserver"));
        adapters.Add(factory.Create("oracle"));

        // 构造函数部分执行其他 Adapter 的构造、准备工作的相关任务
    }
    public IDatabaseAdapter this[int index] { get { return adapters[index]; } }
}

[TestClass]
public class TestSyncEndpoint
{
    [TestMethod]
    public void Test()
    {
```

```
        Endpoint endPoint = new Endpoint();
        endPoint[0].SetData(endPoint[1].GetData());
    }
}
```

不难看出第三种方式采用了很“老套”的伎俩——“如果某一部分工作比较麻烦，就做个对象把事情丢出去”。这里客户程序把获取、构造、配置不同 Adapter 的任务丢给了 Endpoint，自己更关注与业务逻辑的实现（把 A 的数据导入 B）。项目中这种情况常常发生，比如把某个数据库的数据作为日志写到文件里面，或者在 OA 系统里把一个用户发出的任务交给另一个用户（这里您可以把 ORM 机制理解为 RDBMS 基于 SQL 的接口到实体对象系统的 Adapter）。它们的特点都很类似，就是在相同对等的 Adapter 间进行操作。

不过，有可能你的项目就是处理各种数据交换、迁移的，为每一种操作都定义 EndPoint，成本比较高，那么我们可以考虑把操作借助反射异步化，让 Endpoint 变得更通用，而各种客户程序可以使用同一个 Endpoint。

8.4.4 方式 3：借助反射和约定完成异步调用

其示例代码如下：

Unit Test

```
delegate void AfterInvokeHandler(int requestIndex, int responseIndex,
    string requestMethod, string responseMethod);

class Endpoint
{
/// … …

    /// 通过反射，执行一项具体的任务
    public void Invoke(int requestIndex, string requestMethod,
        int responseIndex,
        string responseMethod, bool needRelay, AfterInvokeHandler callback)
    {
        object requester = adapters[requestIndex];
        object responser = adapters[responseIndex];
        MethodInfo request = requester.GetType().GetMethod(requestMethod);
        MethodInfo response = responser.GetType().GetMethod(responseMethod);
        object result = response.Invoke(responser, null);
        if (needRelay)
            request.Invoke(requester, new object[] { result });
        else
            request.Invoke(requester, null);
        // 异步回调，通知执行完毕
        callback(responseIndex, responseIndex, requestMethod,
            responseMethod);
    }
}

[TestClass]
public class TestAsyncEndpoint
{
```

```
    [TestMethod]
    public void Test()
    {
        Endpoint endPoint = new Endpoint();
        // 相当于调用 endPoint[0].SetData(endPoint[1].GetData());
        endPoint.Invoke(0, "SetData", 1, "GetData", true, Log);
    }
    private void Log(int requestIndex, int responseIndex,
        string requestMethod, string responseMethod) { }
}
```

相对第二个方式而言，这里引入了反射机制，其实在 Invoke()方法里，根本就没有 IDataAdapter 之类的类型，也就是说它是通用的，您甚至可以把 Invoke 作为一个调用队列的入口，它的公共仅仅是把需要执行的任务放进队列，至于“排好队”的任务执行后该通知谁由那个委托变量完成即可。虽然实施上稍微增加了几行代码，但它更适合 Adapter 间批量后台任务的执行，而此时客户程序可以说：“我不关心过程，告诉我结果”。

相对于一般的“Client-Adapter”方式而言，“Adapter-Adapter”方式可以做很多更灵活的处理，毕竟双方都是“知己知彼”的对等抽象结构。

8.5 用配置约定适配过程

相对前面的工厂、创建者模式而言，适配器模式的变化主要体现在 Adapter 类型上，因为对客户程序而言，它需要的抽象接口 ITarget 是固定的，您只须告诉它用哪个实现了 ITarget 的类型即可，额外的可能需要把 Adapter 的一些执行参数配置上去即可。我们看一个现成的例子：.NET Framework 中关于 Data Provider 的设置。

首先，在 Machine.Config 中有个名为<system.data>的 ConfigurationSection，它的下面有一个名为<DbProviderFactories>的节点（ConfigurationElemeentCollection），它登记了每个 DbProvider 的逻辑名称和具体的 Qualified 名称，其代码如下：

Machine.Config

```
<configuration>
  <configSections>
    <section name="system.data" type=" "/>
  </configSections>
  <system.data>
    <DbProviderFactories>
      <add name="SqlClient Data Provider"invariant=" " description="."type=" "/>
        … …
    </DbProviderFactories>
  </system.data>
</configuration>
```

然后，我们在应用中选择需要的 DbProvider，这其实相当于选择适合某个特定数据库的一组符合 DbConnection、DbDataAdapter、DbCommand、DbTransaction……要求的 Adapter

类型，由于它们是“一系列相关或具有依赖关系的”，所以没有直接返回具体的 SqlConnection、SqlCommand，而是返回了一个抽象工厂，由这个工厂加工具体的适配器，其代码如下。

App.Config

```
<configuration>
  <connectionStrings>
    <add name="sales" providerName="System.Data.SqlClient" connectionString=
"" />
  </connectionStrings>
</configuration>
```

C#

```
ConnectionStringSettings setting =
ConfigurationManager.ConnectionStrings["sales"];
DbProviderFactory factory =
DbProviderFactories.GetFactory(setting.ProviderName);
DbConnection connection = factory.CreateConnection();
connection.ConnectionString = setting.ConnectionString;
```

其实 ADO.NET 2.0 的配置给了我们一个不错的适配器类型配置思路，总结如下：

- 首先，客户程序明确自己需要的接口 ITarget。
- 设计 App.Config（或 Web.Config，有可能还会涉及 Machine.Config）的结构，考虑到扩展和需要，可以把这个配置部分设计为一个 ConfigurationElementCollection，可以动态增加。
- 考虑 Adapter 还需要调整哪些特性，比如数据访问中的 Package Size 之类的参数，如果参数的数量和结构非常稳定，可以设计为每个 ITarget 配置节点下面的子 ConfigurationElement，但如果不同 ITarget 类型需要的参数数量不同，内容也不一定，则在每个 ITarget 节点下增加一个 ConfigurationElementCollection。
- 我们完成或采购第三方实现 ITarget 的类库。
- 为这些类库里面实现了 ITarget 的类型起一个逻辑名称，并把它们的 Quanlified 名称登记在每个 ITarget 的 ConfigurationElement 中，相关的参数顺次登记在子节点中。
- 按照我们在创建型模式已经介绍的方法，增加一个工厂类，按照逻辑名称创建 ITarget。

您也可以把认为“固定”的参数定义在 ITarget 的 ConfigurationElement 上，不过除非是非常必要而且固定的，否则还是算了。把一个 Name/Value Pair 传递给具体 Adapter，后者将酌情办理。

为什么要给每个具体 Adapter 起个逻辑名称呢？这是为了让你的应用能与具体的 Adaptee 有效隔离，比如您的业务是操作销售数据，那么开发最好基于一个名为“Sales”的逻辑数据库，至于系统上线后，把它放在 ORACLE 里还是 SQL Server 里，放在哪台服务器上，进行配置即可，当然您的应用首先要使用通用的 SQL 标准完成。）

8.6 XML 数据的专用适配机制

随着 Internet 的发展，有一个技术的推动作用甚至超过.NET、Java、数据库，它就是 XML 技术家族。在 XML 世界里，XML 是数据、是 Schema、是定义 Schema 的元语言，它还是指令系统和执行逻辑（借助跨平台的解释器），是检索和查询语言，是 UI 语言。

以往我们谈到经典适配器模式的时候，其实更关心的是方法和方法参数、方法返回值，但在 XML 的世界里，交换的是消息，即 XML 的各种消息。要让两个系统互联最主要的因素有两个：XML 化的接口描述和数据 Schema。前者我们采用传统的方法处理就可以了，因为无论是 Visual Studio.NET 还是 Eclipse 都可以帮忙做很多代理类，后者一般是两个 Schema 间的转换，是客户程序一方根据自己需要的 Schema，调用服务端某种返回结果 Schema 的适配过程。这里有两个选择：

- 就像 20 年前我们处理 EDI 报文一样，把文本中的内容逐项提取出来，然后放到目标文本中，只不过对于 XML 这种层次型数据，我们可以使用 XPath。
- 采用 XML 的方式，使用 XSLT。

除非处理逻辑特别复杂，复杂到必须调用 XML 技术之外的处理，否则我一般都倾向于使用 XSLT。下面是一个我常称之为“黄金公式”的内容，它适用的场景是 XML：

XML (符合 Source XSD) -> XSLT (Source XSD to Target XSD) -> XML (符合 Target XSD)

当然，您也知道选择“新”一点技术还要过项目经理那道关，对吧，他们的理由总是压倒性的——“风险”、“进度”。本章仅介绍基本 XML 数据的适配过程，在 Web Service 设计模式中，还会介绍 Web Service 方法间的适配。

例如，Internet 上某个电子商务 Web Service 提供的订单录入接口输入和回执数据 Schema 分别为 DomainData.xsd 和 DomainDataTarget.xsd。我们按照意图，通过某个图形化的 XSTL 完成 Order.xsd 到 OrderReceipt.xsd 的设计工作（其实就是拖曳），如图 8-8 所示。

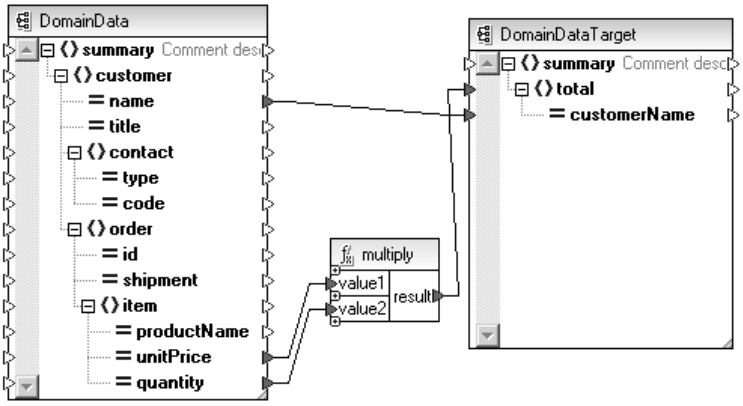


图 8-8 Order.xsd -> OrderReceipt.xsd

然后，以通用的 XSLT Adapter 处理它。由于其中 XSLT 的工作非常通用，所以我们继续向工具箱 Common 项目中增加一个 XmlHelper 的工具类，借助这个工具类，我们就可以完成订单 Adapter 类型，其代码如下：

C# XmlHelper

```
public static class XmlHelper
{
    /// 保存所有被加载并编译的 XSLT 实例缓冲
    private static Dictionary<string, XslCompiledTransform> transforms =
        new Dictionary<string, XslCompiledTransform>();

    /// 根据 XSLT 的定义完成 XML 文件的转换
    public static XmlDocument Transform(string xsltFile, XmlDocument source)
    {
        XslCompiledTransform tranform = GetTransform(xsltFile);
        using (MemoryStream stream = new MemoryStream())
        {
            tranform.Transform(source, null, stream);
            stream.Position = 0;
            XmlDocument target = new XmlDocument();
            target.Load(stream);
            return target;
        }
    }

    private static XslCompiledTransform GetTransform(string xsltFile)
    {
        // 根据缓冲情况获取 XSLT 实例
        XslCompiledTransform transform;
        if (!transforms.TryGetValue(xsltFile, out transform))
        {
            transform = new XslCompiledTransform();
            transform.Load(xsltFile);
            transforms.Add(xsltFile, transform);
        }
        return transform;
    }
}
```

C# 基于 XSLT 实现的 Adapter

```
public abstract class XsltAdapterBase
{
    protected string xslt;
    public virtual XmlDocument Transform(XmlDocument source)
    {
        return XmlHelper.Transform(xslt, source);
    }
}

public class OrderAdapter : XsltAdapterBase
{
    public OrderAdapter() { base.xslt = "Order.xslt"; }
}
```

区别于以往的适配概念，XSLT 关注的是复杂参数或数据的适配过程，它的出现和

Internet 有关。以往我们的软件运行在一个部门或企业内部，但现在越来越多的应用运行在 Internet 上，您内部可能需要检索一个产品信息，但不同 Web Service 提供商的 Schema 千奇百怪，除了使用它们的 WSDL 调用数据外，还经常要对不同 Schema 的数据进行处理。但“唯一不变的就是变化本身”，更何况您很难约束对方的情况，为了不回到以往重复编写编码、拆解报文的情景，我们更多地基于外部标准完成这个工作。试想示例中如果商家的 xsd 变化了，为了保持我们内部应用的稳定，可以修改 XSLT，至于那个 Adapter 还可以继续使用，因为它把变化放在“外面”了。

类似地，如果您没有使用 XML 数据库，也经常存在不同数据表间的数据操作，您可以采用上面的 Adapter-Adapter 互连方式，通过 C#代码解决这个问题，也可以把它们写在存储过程里，使用 Adapter 调用这个存储过程。采用这种方法同样可以完成数据部分的适配（或称之为转换）。

8.7 小结

再次回到适配器模式的意图：解决接口间的不兼容问题。

考虑到工程中不同情况，我们做了如下几个扩展：

- 为了做到 1 对多的适配，我们通过“组”的方式，把满足不同 Adaptee 的适配器组织在一起，然后按照名称（逻辑名称、类型名称或其他可以区分的名称）借助一个工厂完成随需适配的过程。
- 考虑到很多时候我们的开发就是在 Adapter 之上的，进行 Adapter 间的调用，我们增加了 Endpoint。
- 由于 Adapter 的工作就是 Adaptee 的兼容性修改，我们参考了 ADO.NET 2.0 的做法，介绍了如何实现数据 Adapter。
- 最后，我们把适配的概念从以往印象中单纯的方法适配，扩展为复杂参数结构的适配，用 XSLT 的方法解决 XML 数据间的适配过程。

第 11 章

装饰模式

11.1 说明

11.2 经典回顾

11.3 具有自我更新特征的装饰模式

11.4 设计 Decorator 与 Builder 协作的产物

11.5 把 Decorator 做成标签

11.6 小结

11.1 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*

装饰模式是设计模式中实现技巧性非常明显的一个模式，它很多时候也会用来应对类型在继承过程中快速膨胀的情况，而导致膨胀的原因是我们往往需要为类型增加新的职责（功能），而这恰恰是软件在开发和维护阶段过程中最经常的变化。

以我的手机（或称之为移动电话）为例：

- 之所以称它为手机，恐怕首要原因是可以在移动中拨打电话，以及收发短信息，我第一个手机的功能也就是拨打、接听电话和收发短信（因为那部手机不能输入中文，所以短信用得不多），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。
- 上班后，我发现很多同事的手机可以听 MP3，这时候的手机就成了 MobileWithMP3：IMobile、IMP3。
- 接着，我换了部多普达，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特定对象的一个方面，在软件维护的过程中，我们经常需要某些类型添加新的接口，随着接口的增加，实现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很多时候不是我们可以承受的。为了避免出现这种情况，之前我们会考虑采用组合方式解决，但客户程序又需要从不同角度看待组合后的类型，也就是可以通过某种接口调用这个子类。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。

has a 实例：比如那个 MobileWithMP3 就是一个手机“还有”MP3 功能；is a 实例：客户程序可以用 IMP3 mp3 = new MobileWithMP3()的方式使用这个子类。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。

11.2 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，然后增加了一堆需求：

- 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（功能或属性）是需要动态增加的。装饰模式给出的解决方案如图 11-1 所示。

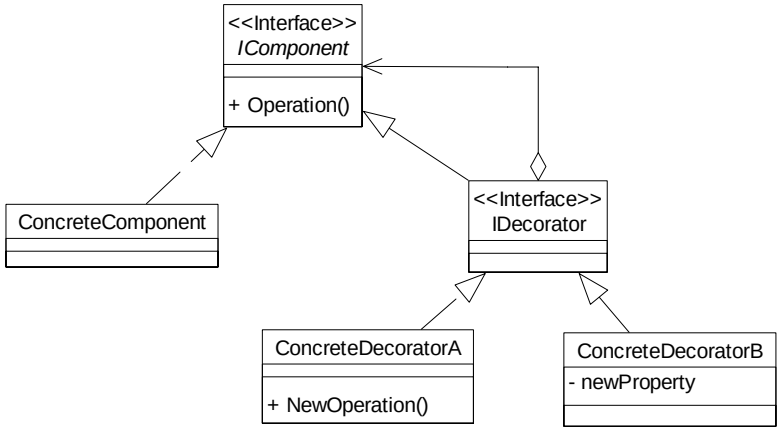


图 11-1 经典装饰模式的静态结构

我们从概念上理解一下这个结构：

- 首先，我们需要的是显示文字，这时候可以指定一个名为 IText 的接口，它有名为 Display() 的方法。
- 然后，我们把所有需要用来修饰的类型也抽象出一个名为 IDecorator 的接口，它继承自这个 IText，因此其实体类必须实现 Content 属性。
- 接着，我们把没有任何新增功能的 IText 做出一个“毛坯”的实体类型，起名为 Text。
- 最后，我们把可以令字体 Bold()、SetColor()的方法填充到每个具体的修饰类型中。

这样概念上当 Text 需要某种新增功能的时候，直接套上某个具体修饰类型就可以了。这个修饰过程最直观的描述是穿上一层又一层的马甲。那么该模式主要适用于哪些情景呢？

- 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 IComponent 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，而客户程序只负责根据 IComponent 的方法调用。
- 屏蔽某些职责，也就是在套用某个装饰类型的时候，并不增加新的特征，而只把既有方法屏蔽。
- 避免出现为了适应变化而子类膨胀的情况。

方法屏蔽：即并不调用它，这样在客户程序拿到的最终结果里，虽然可以按照 IComponent 的声明调用该方法，但实际上本来应该执行的操作并没有执行。

下面我们看一个实现示例，示例代码如下：

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# 具体类型实现

```
/// 具体装饰类
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
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
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));
}
```

从上面的示例不难看出，装饰模式实现上特别有技巧，它的声明要实现 IComponent 定义的方法，但同时又会保留一个 IComponent 的成员，IComponent 接口方法的实现其实是通过自己保存的那个 IComponent 成员完成的，自己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，有时候我们用它“撤销”某些功能。项目中我们有三个要点必须把握：

- IComponent 不要直接或间接地使用 IDecorator，因为它不知道 IDecorator 的存在。
- IDecorator 也仅仅认识 IComponent。
- 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，当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，最后 IComponent 的类型是最外层 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。

11.3 具有自我更新特征的装饰模式

11.3.1 分析

还记得 Word 么，我们把一段文件变成 Bold 之后，难道就再没有机会 Unbold 了么？不是的。再次点击“B”按钮或“CTRL_B”即可，概念上要求我们的装饰类具有这样的能力，示例代码如下：

C#

```
// 建立对象，并对其进行两次装饰
IText d = new TextObject();
IText d1 = new BoldDecorator(d);
IText d2 = new ColorDecorator(d1);
IText d3 = new BoldDecorator (d2);  // 第二次调用，相当于撤销这个装饰类

IText result = d3;
Assert.AreEqual<string>("<color>hello</color>", result.Content);
```

对于 ColorDecorator 也一样，它也要允许我们根据需要随时修改文字内容。也就是说，尽管装饰模式的意图中提到“动态”，但在经典设计模式中示例都是增加的，我们在项目中往往还需要处理“卸载”和“重新定义”的情况。

您可能觉得这个很容易，找到那个装饰类型，然后进行修改，不就可以了？没错，不过前提是必须先找到装饰类型。我们看看之前示例中的调用次序，如图 11-2 所示。

为了可以对装饰类的内容更新，我们需要做些准备工作：

- 虽然时序上有调用关系，但调用是通过 has a 的一个 IText 对象来传递的，对象间并没有直接的线索，首先要做到可以定位到具体某个装饰类。
- 不同装饰类更新的内容不同，比如：BoldDecorator 更新的是一个 bool 状态，而 ColorDecorator 更新的是一个 Color 的枚举，必须能适应这个变化。
- 既然装饰模式意图中明确要求“动态”执行，那么这个更新过程也必须可以动态完成。

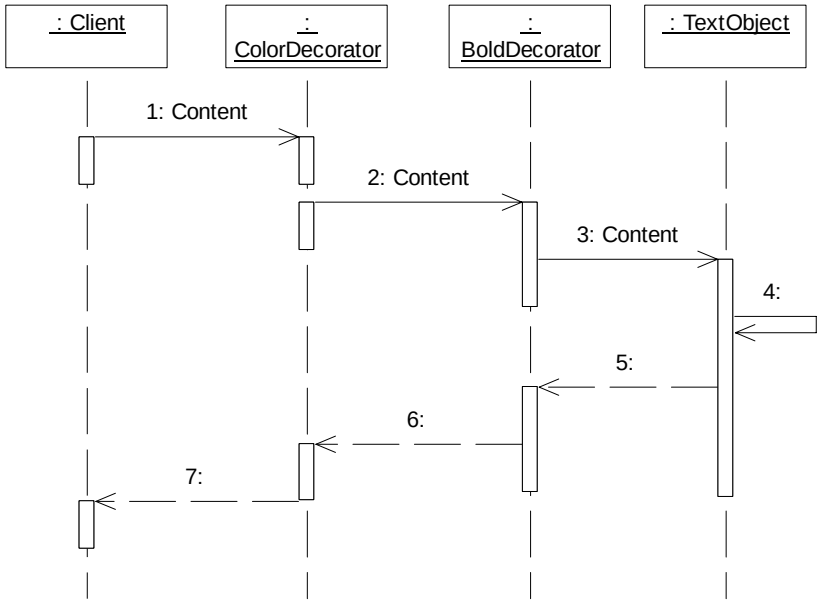


图 11-2 装饰模式示例的执行时序

11.3.2 抽象装饰接口

示例代码如下：

C#

```
/// 定义每个 Decorator 会影响的内容
public interface IState
{
    /// 比较新定义的状态是否与现有的状态一致
    bool Equals(IState newState);
}

public interface IDecorator : IText
{
    /// 记录当前 Decorator 需要维护的状态信息
    IState State { get; set;}

    /// 用于更新 Decorator 的方法
    /// 类型参数 T 主要用于定位具体匹配的 Decorator 对象
    void Refresh<T>(IState newState) where T : IDecorator;
}

public abstract class DecoratorBase : IDecorator    // is a
{
    /// has a
    protected IText target;
    public DecoratorBase(IText target) { this.target = target; }

    public abstract string Content { get;}
```

```
    /// 有关 Decorator 状态的属性及更新操作
    protected IState state;
    public virtual IState State
    {
        get { return this.state; }
        set { this.state = value; }
    }
    public virtual void Refresh<T>(IState newState) where T : IDecorator
    {
        if (this.GetType() == typeof(T))
        {
            if(newState == null) State = null;
            if((State != null) && (!State.Equals(newState)))
                State = newState;
            return;
        }
        // 通过递归继续寻找与 T 匹配的 Decorator 类型
        if (target != null)
            ((IDecorator)target).Refresh<T>(newState);
    }
}
```

新定义的 IDecorator 接口中，把它的内容抽象为一个名为 IState 的接口，这样各个具体装饰类型可以在一个框架下定义自己的相关内容。

11.3.3 抽象状态接口

示例代码如下：

C#

```
public class BoldState : IState
{
    public bool IsBold;
    public bool Equals(IState newState)
    {
        if (newState == null) return false;
        return ((BoldState)newState).IsBold == IsBold;
    }
}

public class ColorState : IState
{
    public Color Color = Color.Black;
    public bool Equals(IState newState)
    {
        if (newState == null) return false;
        return (((ColorState)newState).Color == Color);
    }
}
```

这里新定义的 IState 代表每个装饰类型所有可被外界在运行时“动态”修改的内容，比如 BoldDecorator 仅仅修改是否为“粗体”这个属性，而 ColorDecorator 只修改文字颜色。

实际项目中，您可以根据需要定义更为复杂的 IState 实体。

11.3.4 依据当前状态修改装饰

示例代码如下：

C#

```
/// 具体装饰类
public class BoldDecorator : DecoratorBase
{
    public BoldDecorator(IText target) : base(target) { base.state =
       new BoldState(); }
    public override string Content
    {
        get
        {
            if (((BoldState)State).IsBold)
                return "<b>" + target.Content + "</b>";
            else
                return target.Content;
        }
    }
}

///  具体装饰类
public class ColorDecorator : DecoratorBase
{
    public ColorDecorator(IText target) : base(target) { base.state =
       new ColorState(); }
    public override string Content
    {
        get
        {
            string colorName = (((ColorState)State).Color).Name;
            return "<" + colorName + ">" + target.Content + "</" + colorName + ">";
        }
    }
}
```

每个装饰类对 IText 和内层 IDecorator 传递出来的内容，根据自己 IState 对象的定义进行装饰。

11.3.5 测试验证

示例代码如下：

Unit Test

```
[TestMethod]
public void Test()
{
    // 建立对象，并对其进行两次装饰。bold = false, color= black
```

```
    IText text = new TextObject();
    text = new BoldDecorator(text);
    text = new ColorDecorator(text);
    Assert.AreEqual<string>("<Black>hello</Black>", text.Content);

    // 动态找到需要更新的 Decorator 并修改相应属性
    // bold = false, color = red
    ColorState newColorState = new ColorState();
    newColorState.Color = Color.Red;
    IDecorator root = (IDecorator)text;
    root.Refresh<ColorDecorator>(newColorState);
    Assert.AreEqual<string>("<Red>hello</Red>", text.Content);

    // 动态找到需要更新的 Decorator 并修改相应属性
    // bold = true, color = red
    BoldState newBoldState = new BoldState();
    newBoldState.IsBold = true;
    root.Refresh<BoldDecorator>(newBoldState);
    Assert.AreEqual<string>("<Red><b>hello</b></Red>", text.Content);
}
```

从上面的示例不难看出，由于装饰模式结构的特殊性，对其进行更新的过程也同样需要按照 has a -> is a 的方式交替检索定位。由于更新过程是在 Decorator 结构已经创建之后进行的，因此这个示例等于部分借鉴了后面会介绍的行为型模式。

11.4 设计 Decorator 与 Builder 协作的产物

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);
… …
```

为了隔离客户程序与这一组装饰类型的直接引用，在“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。

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。我们在之前的示例已经“中规中矩”地完成了经典装饰模式的实现，但回头看看会发现 IComponent 随着 IDecorator 数量的增加而变化，在此基础上我们要考虑做些改变。

还记得我们在第 1 章介绍的“依赖注入”么？这里客户程序逻辑上仅需要使用一组 T:IDecorator, new()，但为了“套用”装饰模式的实现，所以与一组 ConcreteDecorator 产生依赖，这种情况下我们会把这些任务交给一个独立的装配对象，由它协助客户程序组织这批 ConcreteDecorator，并把它们交给 Builder。这样，我们完成一个示例，示例代码如下：

C# 登记客户类型与相应装饰类型配置关系的 Assembly 类型

```
public class DecoratorAssembly
{
    private static IDictionary<Type, IList<Type>> dictionary =
        new Dictionary<Type, IList<Type>>();

    /// 项目中这个加载过程可以借助配置完成
    static DecoratorAssembly()
    {
        IList<Type> types = new List<Type>();
        types.Add(typeof(BoldDecorator));
        types.Add(typeof(ColorDecorator));
        dictionary.Add(typeof(TextObject), types);
    }

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

C# 完成装饰过程的 Builder 类型

```
public class DecoratorBuilder
{
    private DecoratorAssembly assembly = new DecoratorAssembly();

    public IText BuildUp(IText target)
    {
        if (target == null) throw new ArgumentNullException("target");

        IList<Type> types = assembly[target.GetType()];
        if ((types != null) && (types.Count > 0))
            foreach (Type type in types)
                // 相当于 text = new ColorDecorator(text);
                target = (IText)Activator.CreateInstance(type, 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);
}
```

从示例里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前 1:1 的依赖关系，而这里 Builder 类型可以进一步设计为泛型 Builder，这样可以减少多个客户类型使用装饰模式时“繁文缛节”的过程；另外，装配对象的加入也给我们从外部配置客户类型及其相关装饰类型的机会。

11.5 把 Decorator 做成标签

11.5.1 更“彻底”的 Attribute 注入

在前面的章节里我们其实已经多次使用 Attribute 作为类型修饰性描述，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对象完成，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式无可奈何。一个更为普遍的办法就是采用“拦截”（Interception）方法，借助透明代理对某些标记装饰属性（Attribute）的方法调用过程进行定制。

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，它感觉不到这个重发的过程，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支持对象。新的调用关系如图 11-3 所示。

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问题，经典设计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，而采用动态代理则是一个后台支撑机制，是实现这种途径的手段。

实际执行步骤相对更为复杂，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。

.NET 的 Attribute 和 Property 一般都被译为“属性”，而在比较大型的.NET 应用中两者又常常同时出现，因此很容易混淆。在同时出现两个概念的时候，本书采用“属性”和“属性方法”分别指向 Attribute 和 Property，尽量避免误会。

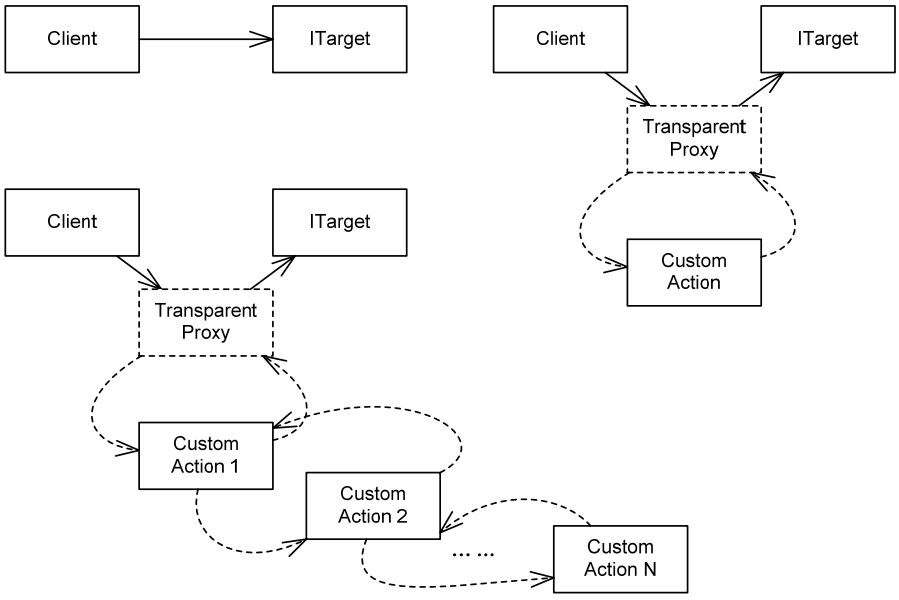


图 11-3 基于属性的动态代理装饰过程

11.5.2 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，不过如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。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)
    {
        MethodCallMessageWrapper caller =
            new MethodCallMessageWrapper((IMethodCallMessage)msg);

        // 提取实际宿主对象
        MethodInfo method = (MethodInfo)caller.MethodBase;
        T target = (T)GetUnwrappedServer();
        DecoratorAttributeBase[] attributes = (DecoratorAttributeBase[])
            method.GetCustomAttributes(typeof(DecoratorAttributeBase),
              true);
        if (attributes.Length > 0)
            foreach (DecoratorAttributeBase attribute in attributes)
               attribute.Intercept(caller);
        object 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
{
    private Type type;
    public ArgumentTypeRestrictionAttribute(Type type) { this.type = type; }
    public override void Intercept(object target)
    {
        MethodCallMessageWrapper caller = (MethodCallMessageWrapper)target;
        if (caller.ArgCount == 0) return;
        for (int i = 0; i < caller.ArgCount; i++)
        {
            object 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)
    {
        MethodCallMessageWrapper caller = (MethodCallMessageWrapper)target;
        if(caller.ArgCount == 0) return;
            foreach(object arg in caller.Args)
                if(string.IsNullOrEmpty((string)arg))
                    throw new ArgumentException();
    }
}
```

C# 定义业务对象

```
class User : MarshalByRefObject
```

```
{
    private string name;
    private 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.AreEqual<string>("0", exception.Message);
    }

    try
    {
        user.SetUserInfo("", "manager");
    }
    catch (Exception exception)
    {
        // 因 name 为空被拦截后抛出异常
        Assert.AreEqual<string>("string is null or empty", exception.Message);
    }
}
```

从上面的示例不难看出，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”锲入额外的控制逻辑。但比较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、对于属性方法和构造函数的支持，更为复杂的工程化实现可以参考微软的 Policy Injection 和 Unity 代码块。

11.5.3 方式 2：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，则需要通过一些更复杂的、更“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，动态使用 MSIL 装配出一个新的类型，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 11-4 所示。其执行步骤如下：

- 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。
- 执行过程中，客户程序实际调用的是动态生成的新类型。

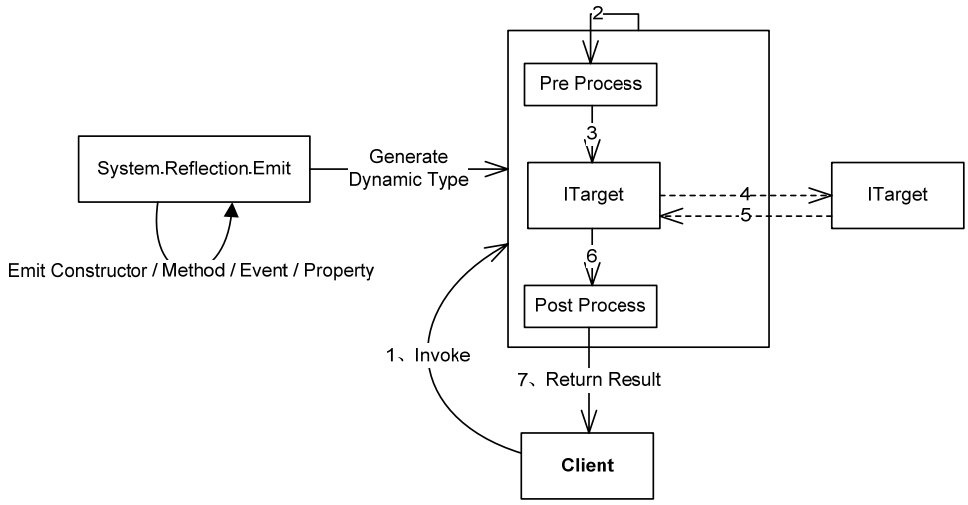


图 11-4 动态包装类型的执行过程

下面是一个简化后（但相对其他示例已算很复杂）的示例，其代码如下：

C# 定义实际执行其他方法的委托

```
public delegate object MethodCall(object target, MethodBase method,
   object[] parameters, DecoratorAttribute[] attributes);
```

C# 定义各种装饰属性

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Property |
   AttributeTargets.Interface, Inherited = true)]
public abstract class DecoratorAttribute : Attribute
{
   public abstract void Process(object target, MethodBase method,
     object[] parameters);
}

/// 代表执行前和执行后外部“横切”机制的抽象对象
public abstract class BeforeDecoratorAttribute : DecoratorAttribute { }
public abstract class AfterDecoratorAttribute : DecoratorAttribute { }
```

C# 定义装饰对象注入器

完整示例请参考
MarvellousWorks.PracticalPattern.DecoratorPattern.Interception 下的
DecoratorInjector

```
public class DecoratorInjector
{
    public const string AssemblyName = "TEMP_DYNAMIC_ASSEMBLY";
    public const string ClassName = "TEMP_CLASS_NAME";

    public static object InjectHandlerMethod(object target, MethodBase method,
        object[] parameters, DecoratorAttribute[] attributes)
    {
        object returnValue = null;
        foreach (DecoratorAttribute attribute in attributes)
            if (attribute is BeforeDecoratorAttribute)
                attribute.Process(target, method, parameters);
        returnValue = target.GetType().GetMethod(method.Name).Invoke(target,
          parameters);
        foreach (DecoratorAttribute attribute in attributes)
            if (attribute is AfterDecoratorAttribute)
                attribute.Process(target, method, parameters);
        return returnValue;
    }

    public static object Create(object target, Type interfaceType)
    {
        Type proxyType = EmiProxyType(target.GetType(), interfaceType);
        return Activator.CreateInstance(proxyType, new object[] { target,
          interfaceType });
    }

    /// 通过动态生成的 MSIL 形成构造方法
    private static void EmitConstructor(TypeBuilder typeBuilder,
    FieldBuilder target, FieldBuilder iface)
    {
        Type objType = Type.GetType("System.Object");
        ConstructorInfo objCtor = objType.GetConstructor(new Type[0]);
        ConstructorBuilder pointCtor = typeBuilder.DefineConstructor(
        MethodAttributes.Public,
            CallingConventions.Standard, new Type[] { typeof(object),
              typeof(Type) });

        ILGenerator ctorIL = pointCtor.GetILGenerator();
        ctorIL.Emit(OpCodes.Ldarg_0);
        ctorIL.Emit(OpCodes.Call, objCtor);
        ctorIL.Emit(OpCodes.Ldarg_0);
        ctorIL.Emit(OpCodes.Ldarg_1);
        ctorIL.Emit(OpCodes.Stfld, target);
        ctorIL.Emit(OpCodes.Ldarg_0);
        ctorIL.Emit(OpCodes.Ldarg_2);
        ctorIL.Emit(OpCodes.Stfld, iface);
        ctorIL.Emit(OpCodes.Ret);
    }
    ……
}
```

C# 定义外部横切机制

```
public class LogAttribute : BeforeDecoratorAttribute
{
    public override void Process(object target, MethodBase method,
      object[] parameters)
    {
        Trace.WriteLine(method.Name);
    }
}

public class ParameterCountAttribute : BeforeDecoratorAttribute
{
    public override void Process(object target, MethodBase method,
      object[] parameters)
    {
        Trace.WriteLine(target.GetType().Name);
    }
}
```

Unit Test

```
public interface IBizObject
{
    /// 对对象功能的装饰采用“横切”而非传统的继承方式获得，
    /// 中间代理对象的构造是隐式，而且是由 DecoratorInjector 包装的，
    /// 从外部角度看，对象的扩展是在 Meta 信息部分扩展，而且有关的约
    /// 束定义在接口而非实体类层次
    [Log]
    [ParameterCount]
    int GetValue();
}

public class BizObject : IBizObject
{
    public int GetValue() { return 0; }
}

[TestMethod]
public void Test()
{
    IBizObject obj = (IBizObject)DecoratorInjector.Create(
    new BizObject(), typeof(IBizObject));
    int val = obj.GetValue();
    Assert.AreEqual<int>(0, val);
}
```

11.5.4 进一步分析

通过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，可以借助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，通过配置业务类型的装饰属性即可实施新的控制。

不过上面的示例用于实际项目中还有很多欠缺，主要是性能问题，该问题是由反射和动

态加载所导致的。在类似 Castle 等一些更成熟的框架中会更多借助缓冲，把动态创建的代理类实例保存在内存里，同时把动态生成的 Assembly 保存在文件系统中，以最大程度地提高执行效率。

11.6 小结

作为整个设计模式中非常讲究技巧的一个模式，经典实现中装饰模式通过继承方式实现了对新功能的拓展。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，为此本章借鉴后面要提到的状态模式实现“动态”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，以“横切”使用 Attribute 的方式实现装饰过程，最大程度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和动态 Assembly 生成等办法，执行效率也是必须关注的问题。

第 12 章

外观模式

12.1 说明

12.2 经典回顾

12.3 Facade 接口

12.4 Remote Facade

12.5 面向性能考虑的升级版 Remote Facade——Data Transfer Object 模式

12.6 平台、开发语言无关的抽象 Facade 接口——WSDL

12.7 让使用者更加方便的 Fluent Interface 设计

12.8 小结

12.1 说明

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

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

外观模式（或门面模式、包装模式）是设计模式中非常朴素地体现面向对象“封装”概念的模式，它的基本原理是将复杂的内部实现以统一接口的方式暴露出来，最大程度地减少客户程序对某些子系统内部众多对象的依赖关系。

随着软件技术的发展，虽然主观上我们更多采用面向对象技术进行开发，但事实上经过上两个 10 年的积累，很多时候我们不得不通过访问本地 API、调用 COM/COM+对象的方式进行工作，同时由于多种外界信息源和遗留系统的存在，很多时候我们难于获得一个纯粹的面向对象类型体系，为此实际项目中我们已经潜移默化地多次用到了外观模式。

从更高的角度看，SOA 和 Enterprise 2.0 所倡导的软件即服务的概念，其实也在一个新的层次上演绎着外观模式的理念——屏蔽语言、平台、开发工具和网络的差异性，以统一的服务概念供外部使用；从更低的角度看，为了便于开发人员使用我们包装的类型，很多时候也需要通过重载和连贯接口（Fluent Interface）的方式，将最“舒服”的类型外观提供给别人，在满足多态差异性的同时，在更小的子系统范围内——类内部实现更简明的外部接口。

以我们使用的计算机为例，虽然不同计算机内部组成不同，但对于用户而言他直接接触的一般就是显示器、键盘、鼠标和一些多媒体外设，对于计算内部包括哪些组成，这些组成如何协作完成这些计算任务并不了解，这时候这些看得见、摸得着的设备就成了计算机的“外观类型”。我们做的项目也一样。对于很多浏览器客户端项目而言，即便项目内部存在数据访问、日志、消息交换等非常多的子系统和模块，但对于用户而言它交互的接口只有一个浏览器，这时候浏览器成了“外观”。

12.2 经典回顾

外观模式的意图很明确：为子系统中的一组接口提供一个高层接口，该接口使子系统更易于使用。它的主要动机是减少“子系统”内部与外部间对象通信的依赖复杂程度，如图 12-1 所示。

这里的“子系统”是一个概念上的划分，它由一组功能上非常内聚的对象组成，与我们设计系统时常说的按照业务功能划分的“子系统”概念有些区别，该“子系统”更多时候是按照技术实现中对象关系形成的，有时候我们甚至可以把一个非常复杂的对象称为“子系统”。

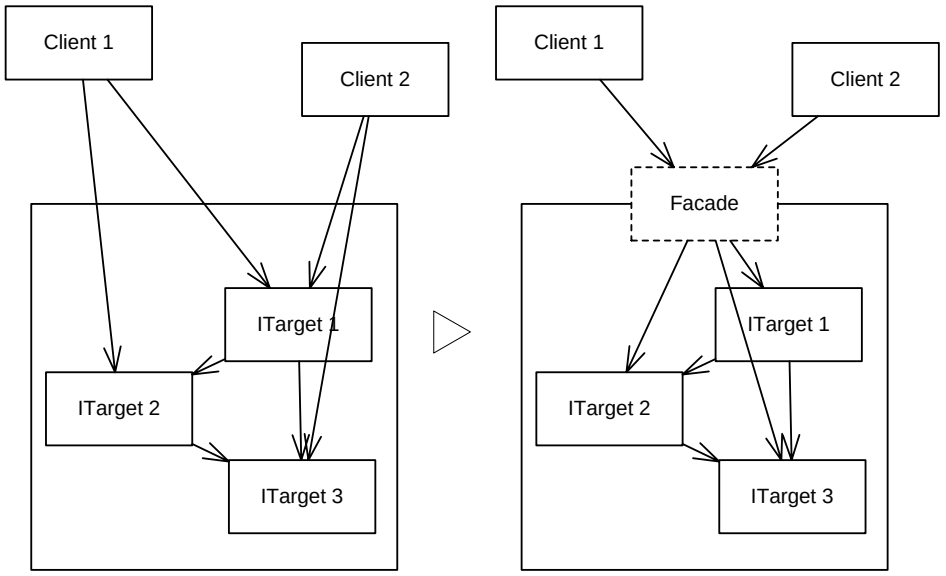


图 12-1 使用外观模式前后的对象间通信关系

从经典外观模式的动机不难看出，它的主要作用是对外屏蔽内部对象间协作的复杂性，那么把它用于项目实施有什么具体优势呢？

- “子系统”可能会随着开发过程的深入或客户需求的变化而越来越复杂，但从客户程序看，它需要的接口（也就是那个外观类型）规格相对固定，因此外观的隔离作用更利于客户程序的稳定。
- 应用中经常存在很多的“子系统”，为了避免客户程序使用中与过多“子系统”内部对象产生依赖，从高层逻辑角度，有必要每个“子系统”封装集中的外观对象。
- 在保证外观对象稳定的情况下，子系统自身可以根据运行环境需要做迁移。
- 另外一个就是涉及开发颗粒度的问题，本着 2/8 原则大部分情况“子系统”内部虽然有很多类型，能够提供非常细颗粒的调用支持，但客户程序往往只需要按照固定的流程操作这些对象，为此不妨把这些 default 的处理过程以外观对象的方式呈现给客户程序，同时，不封闭细颗粒度定制调用。
- 随着软件规模的扩大，一些“子系统”往往分别由不同的团队独立开发，采用外观接口可大大简化将它们装配成应用的过程，同时可提高“子系统”的重用性。

下面我们以最经常进行的数据访问为例，看看如何在 ADO.NET System.Data.Common 命名空间的基础上实现一个简单的数据驱动无关的 DataFacade，示例代码如下：

App.Config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
```

```
  <connectionStrings>
    <add name="AdventureWorks" providerName="System.Data.SqlClient"
      connectionString="…"/>
  </connectionStrings>
</configuration>
```

C# DataFacade

```
public class DataFacade
{
    private const string dbName = "AdventureWorks";
    private static DbProviderFactory factory;
    private static string connectionString;

    /// 通过访问配置找到服务于具体数据库 Provider 的抽象工厂
    static DataFacade()
    {
        ConnectionStringSettings settings =
          ConfigurationManager.ConnectionStrings[dbName];
        factory = DbProviderFactories.GetFactory(settings.ProviderName);
        connectionString = settings.ConnectionString;
    }

    /// Helper method
    private static DbConnection CreateConnection()
    {
        DbConnection connection = factory.CreateConnection();
        connection.ConnectionString = DataFacade.connectionString;
        return connection;
    }

    /// 封装数据访问“子系统”的内部对象
    /// 通过协同 DbConnection、DbCommand、DbDataAdapter 和 DataSet，对外提供简单的
        查询接口
    public DataSet ExecuteQuery(string sql)
    {
        if (string.IsNullOrEmpty(sql)) throw new
          ArgumentNullException("sql");
        using (DbConnection connection = CreateConnection())
        {
            DbCommand command = connection.CreateCommand();
            command.CommandText = sql;
            command.CommandType = CommandType.Text;
            DbDataAdapter adapter = factory.CreateDataAdapter();
            adapter.SelectCommand = command;
            DataSet result = new DataSet();
            adapter.Fill(result);
            return result;
        }
    }
}
```

Unit Test （测试采用 SQL Server 2005 的示例数据库 AdventureWorks）

```
[TestMethod]
public void Test()
{
    DataFacade facade = new DataFacade();
    DataSet result = facade.ExecuteQuery(
    "SELECT TOP 10  CurrencyCode, Name FROM. Sales.Currency");
```

```
    Assert.AreEqual<int>(1, result.Tables.Count);
}
```

上面的示例中笔者有意以一个数据驱动无关的方式实现数据访问，目的是希望对外观模式应用的技巧进行进一步探讨：

1. 如果外观类型直接依赖于具体类型，比如 SqlConnection、SqlCommand 和 SqlDataAdapter 虽然适用于 SQL Server 数据库，但并不能充分实现外观模式的众多优势，比如，当数据源切换到 ORACLE 后，DataFacade 必须重新编译，而客户程序对其的引用也需要进行变化（除非保证强名不变或采用 Late Binding 方式）。为了避免这个问题，外观接口对“子系统”的内部调用最好依从“依赖于抽象”的原则，使用抽象类型或接口，而不是面向具体类型，即便这样，由于所依赖的常常是“一组相关”的对象，根据前面章节的介绍，也很有必要增加一个抽象工厂。
2. 由于外观类型经常被作为一个打包的 API 使用，而且具体执行过程中间接调用“子系统”内部对象完成，因此很多时候使用中只需要保留一个 Singleton 或根据处理能力定义的 Singleton-N 入口实例即可，或者索性定义为一个静态类（Static Class）。

12.3 Facade 接口

上面的示例展示了经典外观模式中一个外观类的设计，但进一步思考不难发现“变化”的风险也同时集中在 DataFacade 一个类型上。随着数据类型的扩大，以后基于文件系统、WMI 监控平台提供查询时，重新写一个新的 DbProvider 的代价要比直接在 ExecuteQuery 方法上作扩展大得多，为此需要修改 DataFacade，相应的客户程序也需要重新调整。

另外，还有一个问题，随着 SOA 和 Enterprise 2.0 概念的推广，“子系统”往往成为跨越物理边界的系统，这时候如果仅仅做一个 Facade 类型，由于客户程序与“子系统”物理上的分离，部署在客户程序端的 Facade 没有实际访问资源的能力。

综合上述因素，实际项目中使用外观类也许不如使用外观接口更利于隔离“子系统”和客户程序，同时需要增加一个构造该接口的工厂类或其他创建机制。新的结构如图 12-2 所示。

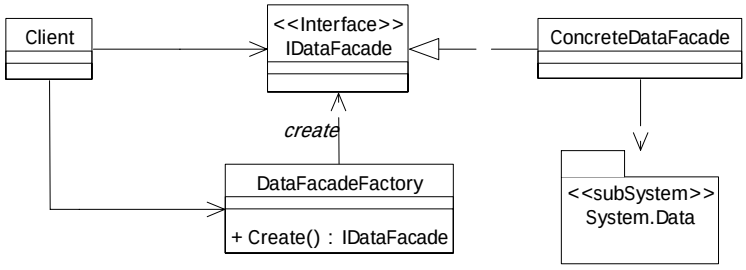


图 12-2 外观接口配置外观工厂机制的静态结构

12.4 Remote Facade

把外观类型需要的功能抽象为接口后，原则上外观类型的执行与客户程序就可以完全分离了，不仅从对象抽象角度看可以进行这个分离，而且跨进程、跨物理位置的调用也成为可能，这时 Facade 变成了 Remote Facade。

虽然 Remote Facade 模式被普遍接受是在 Martin Fowler 撰写《Patterns of Enterprise Application Architecture》之后，但 Windows 平台从 COM 过渡到 DCOM 的过程已经很好地演绎了如何对 COM 这种扁平接口进行封装，通过增加跨进程对象实现“Remote”，.NET 推出的时候也同时提供了 Socket、Enterprise Service、Web Service 和.NET Remoting 四种方式，我一直认为.NET Remoting 是对 Remote Facade 最好的诠释，WCF 更多地关注于远程调用之外的事情，所以用来介绍设计模式反而显得“浮华”。

为了真正演示在.NET Remote 之后基于外观接口的应用在部署方面的优势，我们采用如表 12-1 所示的项目布局。

表 12-1

名称	类型	功能	部署位置	依赖于
Common	Class Library	定义 Facade 接口	客户程序 远程服务端	
BusinessLogic	Class Library	实现 Facade 接口定义的数据访问要求	远程服务端	Common
Host	Client Console	作为远程服务进程	远程服务端	Common BusinessLogic
Client	Windows Application	根据 Common 约定的 Facade 接口消费相关服务	客户程序	Common

编写配置文件的时候感觉有点作茧自缚——名称太长了，所以 Remote Facade 破例采用了比较简短的命名空间。

示例代码如下：

C# 定义 Remote Facade 的接口

该接口位于 Common 项目，其 Assembly 会同时部署到客户程序和远程服务端

```
public interface IDataFacade
{
    DataSet ExecuteQuery(string sql);
}
```

C# 定义具体完成数据访问的 Facade 类型

```
public class DataFacade : System.MarshalByRefObject, IDataFacade
…
```

C# & App.Config 实现并配置远程服务端

```
<configuration>
  <connectionStrings>
    <add name="AdventureWorks" providerName="System.Data.SqlClient"
        connectionString="Data Source = (local); Initial Catalog =
          AdventureWorks;
         Integrated Security = SSPI"/>
  </connectionStrings>
   <system.runtime.remoting>
      <application>
        <service>
          <wellknown mode="Singleton"
             type="Test.Rem.Lib.DataFacade,
             RemLib" objectUri="RemotableType.rem"/>
        </service>
        <channels>
          <channel ref="http" port="9000">
                <serverProviders>
                   <formatter ref="soap"/>
                </serverProviders>
          </channel>
        </channels>
      </application>
   </system.runtime.remoting>
</configuration>

public static void Main(string[] args)
{
    string configFile =
       AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
    RemotingConfiguration.Configure(configFile, false);
    Console.WriteLine("Host started and listening ...");
    Console.ReadLine();
}
```

C# & App.Config 实现并配置客户程序

```
<configuration>
   <system.runtime.remoting>
      <application>
        <client>
          <wellknown type="Test.Rem.Common.IDataFacade, RemCommon"
                   url="http://localhost:9000/RemotableType.rem"/>
        </client>
        <channels>
          <channel ref="http">
              <clientProviders>
                 <formatter ref="soap"/>
              </clientProviders>
          </channel>
        </channels>
      </application>
   </system.runtime.remoting>
</configuration>

private IDataFacade remObj;

private object CreateWellKnownType(string typeName)
{
```

```
    WellKnownClientTypeEntry[] entries =
    RemotingConfiguration.GetRegisteredWellKnownClientTypes();
    foreach (WellKnownClientTypeEntry entry in entries)
        if (string.Equals(entry.TypeName, typeName))
            return Activator.GetObject(Type.GetType(typeName),
              entry.ObjectUrl);
    return null;
}

private void ClientForm_Load(object sender, System.EventArgs e)
{
    string configFile =
      AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
    txtConfigFile.Text = configFile;
    RemotingConfiguration.Configure(configFile, false);
    string typeName = "Test.Rem.Common.IDataFacade";
    remObj = (IDataFacade)RemHelper.CreateWellKnownType(typeName);
    txtMessage.Text = "Hello world";
}

private void btnQuery_Click(object sender, EventArgs e)
{
    DataSet result = remObj.ExecuteQuery(txtQuery.Text);
    if((result == null) || (result.Tables[0].Rows.Count == 0)) return;
    txtMessage.Text = "";
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 10; i++)
        sb.Append(Convert.ToString(result.Tables[0].Rows[i][1]) +
          Environment.NewLine);
    txtMessage.Text = sb.ToString();
}
```

执行效果如图 12-3 所示。

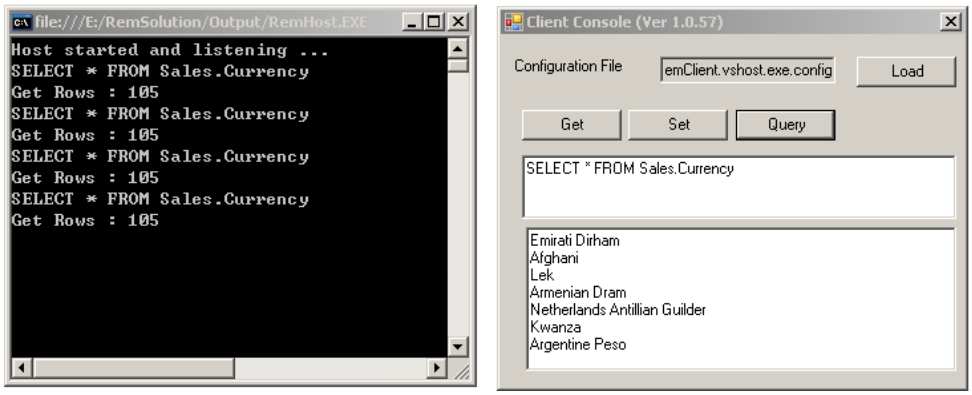


图 12-3 基于.NET Remoting 实现的远程外观对象

不难看出，外观包装的工作其实还是在服务端进程运行，因为相关的 Console 输出信息是写在服务端进程中的；而客户程序也仅仅依赖于外观接口（IDataFacade），借助.NET Remoting 远程调用具体的外观类型（DataFacade）。

这里虽然采用了.NET Remoting，但主要用它完成远程调用中很多复杂的操作，具体项目中完全可以通过 Socket（但必须对 Socket 消息面向对象进行序列化定制）、Web Service 和 Enterprise Service 等多种方式完成，不过为了使必要的组件运行在合适的环境下，必须将外观接口和具体的外观类型相分离。

12.5 面向性能考虑的升级版 Remote Facade——Data Transfer Object 模式

上面的实现仅仅解决了调用远程“子系统”能否连通的问题，并没有针对安全性、性能等其他因素进行考虑，安全性是个没有止境的话题，所以这里也不谈了，针对访问性能，有个问题值得考虑。上面的例子中我们采用 DataSet 方式一次性提取全部数据，返回结果相对单一，但受到数据条目的限制，对于比较大的查询往往不建议采用 DataSet，很多情况下需要提供更利于“细水长流”的 DataReader 方式。下面是直接套用外观接口部分完成的示例，其代码如下：

C# 定义数据访问的 Facade 接口

```
public interface IDataFacade
{
    DbDataReader ExecuteQuery(string sql);
}
```

C# 定义具体完成数据访问的 Facade 类型

```
/// 封装数据访问“子系统”的内部对象
/// 通过协同 DbConnection、DbCommand、DbDataAdapter 和 DataSet，对外提供简单的查询接口
public DbDataReader ExecuteQuery(string sql)
{
    if (string.IsNullOrEmpty(sql)) throw new ArgumentNullException("sql");

    DbConnection connection = CreateConnection();
    DbCommand command = connection.CreateCommand();
    command.CommandText = sql;
    command.CommandType = CommandType.Text;
    connection.Open();

    return command.ExecuteReader();
}
```

Unit Test

```
[TestMethod]
public void TestExecuteDbDataReaderWithoutDTO()
{
    IDataFacade facade = new DataFacade();
    using (DbDataReader reader = facade.ExecuteQuery(
    "SELECT TOP 10 CurrencyCode, Name FROM Sales.Currency"))
    {
        Assert.IsTrue(reader.HasRows);
        while (reader.Read())
```

```
            Trace.WriteLine(Convert.ToString(reader[0]));
        }
    }
```

这里有一个问题，即如果需要遍历的内容太多，而客户程序与实际生成 DbDataReader 的服务器之间有比较大的延时时，每次 reader.Read()都需要一个往复调用，代价比较大，为此有必要在服务端发送前增加一个对调用“打包”的对象，它负责把几次调用合并为一个反馈结果。Martin Flower 在他的《Patterns of Enterprise Application Architecture》中介绍过一个名为 DTO（Data Transfer Object）的模式，它专门适用于这个上下文，而 DTO 对象的职责就是完成“调用打包”。

DTO 与外观模式有什么关系呢？我们现在的应用日趋“松散耦合”，松散到很多以往安装在本地的 COM 和 ActiveX 都已经被封装成服务分散在整个互联网上。上面提到的对“子系统”外部调用“打包”只是一方面，外观对象在封装“子系统”内部的时候也需要“打包”。另外考虑到如果让外观对象直接与 DTO 对象交互，将会额外引发它们之间的依赖关系，而对于同一外观对象，常常也会采用不同的 DTO 按照不同主题进行“打包”，因此很有必要在外观对象和 DTO 对象之外增加一个把它们装配起来的机制，参考第 1 章提到的依赖注入，在这里增加一个装配对象。新的结构如图 12-4 所示。

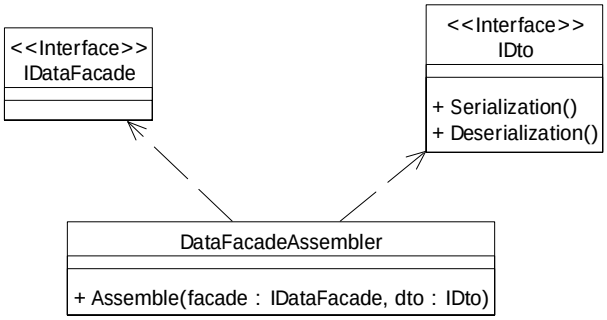


图 12-4 增加 DTO 之后的远程外观结构布局

区别于本地外观对象，这里有几点需要说明的：

- 因为调用过程往往是远程的（跨进程、服务器甚至公共网），因此操作的是外观接口而不是具体外观类。
- 同样的原因，在 DTO 对很多单次调用的结果“打包”后，还在 DTO 接口上定义了序列化操作（如果调用是双向的，也可以考虑增加反序列化操作），这个过程可能简单到仅仅用一个集合类型把它们聚集在一起，也可能复杂到通过 SOAP 包装后，再使用 XML-Signature、WS-Security 等机制生成一个满足 Web Service 使用规格的 SOAP 消息。
- 负责装配的对象的职责也很简单，即把它们装配起来，简言之就是告诉 DTO 对象，它

需要“打包”谁的调用。

下面是一个示例，其代码如下：

C# 定义 DOT 对象接口

```
/// 返回结果为 XmlDocument 方式的 DTO 接口定义
public interface IXmlDataDto
{
    XmlDocument GetCurrency(string sql);
    IDataFacade Facade { set;}
}
```

C# 定义 DOT 对象

```
public class XmlDataDto : IXmlDataDto
{
    private IDataFacade facade;

    /// 为 DTO 对象指定需要打包的对象
    public IDataFacade Facade { set { this.facade = value; } }

    /// 调用外观对象，并将请求结果以自定义的 XML 序列化方式打包
    public XmlDocument GetCurrency(string sql)
    {
        if (facade == null) throw new ArgumentNullException("facade");
        if (string.IsNullOrEmpty(sql)) throw new
           ArgumentNullException("sql");

        DbDataReader reader = facade.ExecuteQuery(sql);
        if (!reader.HasRows) return null;
        XmlDocument doc = new XmlDocument();
        XmlElement root = doc.CreateElement("root");
        doc.AppendChild(root);

        while (reader.Read())
        {
            XmlElement element = doc.CreateElement("currency");
            element.SetAttribute("code", reader.GetString(0));
            // currency code
            element.SetAttribute("name", reader.GetString(1));
            // currency code
            root.AppendChild(element);
        }

        return doc;
    }
}
```

C# 定义装配对象

```
public class DataFacadeAssembler
{
    /// 实际的过程可能非常复杂，Assembler 需要通过各种本地或远程方法
    /// 获得一个 DTO 对象接口的引用，而且考虑到效率因素，DTO 本身应该相对
    /// 客户程序而言，到实际外观对象更近，否则 DTO 的“打包”反而成为累赘
    public IXmlDataDto CreateDto(IDataFacade facade)
    {
        if (facade == null) throw new ArgumentNullException("facade");
```

```
        IXmlDataDto dto = new XmlDataDto();
        dto.Facade = facade;
        return dto;
    }
}
```

Unit Test

```
[TestMethod]
public void TestExecuteDbDataReaderWithDTO()
{
    IDataFacade facade = new DataFacade();
    DataFacadeAssembler assembler = new DataFacadeAssembler();
    IXmlDataDto dto = new DataFacadeAssembler().CreateDto(facade);
    XmlDocument doc =
        dto.GetCurrency("SELECT TOP 10  CurrencyCode, Name FROM
          Sales.Currency");
    Assert.IsNotNull(doc);
}
```

上面的示例虽然很简单，但代表了实际项目运行中一个经常遇到的情况——完成一个业务操作的数据不只一条，甚至不在同一个物理位置。比如做一个订票的系统，往往需要通过 DTO 将来自多家航空售票机构的数据“打包”。虽然相对直接的 Remote Facade 而言，增加 DTO 会带来一定编码量，但客户程序起码可以从下面几处获益：

- 变多次调用为一次调用，获得性能优势。
- 外观模式起到了一个入口的作用，而 DTO 则起到了单一数据结果的作用。
- DTO 与外观接口间为 M:N 的关系，一个 DTO 可能同时汇总多个外观接口，同一个外观接口也可能服务于多个 DTO，而客户程序仅根据它的需要定义 DTO 的规格。

12.6 平台、开发语言无关的抽象 Facade 接口——WSDL

再进一层就到了 SOA 和 Enterprise 2.0 时代的外观模式，这时候“子系统”的概念更加广泛：

- 不同的数据源。
- 不同的应用。
- 不同语言和操作系统平台。

如何描述这些“子系统”组成？显然可以继续上面 Remote Facade 的方法，自定义各种外观接口，但应用领域会受到限制。那么就走标准化道路，此时 WSDL 是一个非常不错的思路，它提供了把这些复杂东西包装起来的统一外观接口描述，而对于使用者而言，它还提供了这个外观接口的位置。如果数据对象比较复杂，则还需要增加一个 XSD 描述的对象实体。这样 WSDL + XSD 打破了一定要用 C#、C++、IDL、Java 这类语言定义接口的惯例，提供了一个更适合互联网时代的 Remote Facade 模式实现手段。

12.7 让使用者更加方便的 Fluent Interface 设计

前面谈的问题似乎越来越大，回到编码我们可以发现外观模式的思想不仅可以放大，同样可以缩小，其中一个最直接的例子就是如何让方法调用的编写使用起来更方便。这时候“子系统”被缩小为一个类内部，而“子系统”的组成也变成这个方法内部需要调用的一系列方法、属性和委托，我们可以把普通的接口设计成更利于使用的“连贯接口”。

Fluent Interface，也被称为连续接口，虽然 Fluent 很少被直接翻译成“连贯”，但根据 FI 的用意，我觉得它不仅仅有简化编码的作用，同时也是保持对象间贯通一致的方式，加之国内对它还没有很权威的命名，所以本书采用“连贯接口”的称呼。

虽然连贯接口的思想可以被用于各种接口、不同模式代码的设计，但由于外观模式是 GOF23 中包容性最复杂的一个模式，因此在这里介绍小“子系统”范围内的外观——连贯接口方法，目的是不仅要包装“子系统”，更要让用户通过这个外观类型更方便地操作“子系统”内部的一系列对象。其代码如下：

C# 增加具有 Fluent 特性的接口方法

```
public interface IDataFacade
{
DataSet ExecuteQuery(string sql);

    /// 具有 Fluent 特性的接口方法
    IDataFacade AddNewCurrency(string code, string name);
}
```

C# 实现该方法

```
/// 为支持连贯操作设计的接口方法
public IDataFacade AddNewCurrency(string code, string name)
{
    string sql = "INSERT INTO Sales.Currency( CurrencyCode, [Name]) VALUES ";
    sql += " ('" + code + "', '" + name + "')";

    using (DbConnection connection = CreateConnection())
    {
        connection.Open();
        DbCommand command = connection.CreateCommand();
        command.CommandText = sql;
        command.CommandType = CommandType.Text;
        if (command.ExecuteNonQuery() != 1)
            throw new ApplicationException("Failed");
}

    return this;    // 保证调用连贯的关键
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    IDataFacade facade = new DataFacade();
```

```
    DataSet result = facade.ExecuteQuery("SELECT * FROM Sales.Currency");
    int currentRows = result.Tables[0].Rows.Count;

    ///连续添加条记录
    facade.AddNewCurrency("DW1", "Known1").
        AddNewCurrency("DW2", "Known2").
        AddNewCurrency("DW3", "Known3").
        AddNewCurrency("DW4", "Known4").
        AddNewCurrency("DW5", "Known5");
    result = facade.ExecuteQuery("SELECT * FROM Sales.Currency");
    Assert.AreEqual<int>(currentRows + 5, result.Tables[0].Rows.Count);
}
```

上面的单元测试显示出连贯接口在客户程序使用上的便利性。上面的示例没有在提交前“打包”调用，因此虽然使用上方便了，但实际的执行效率会很不理想，不过可以参考上面的 DTO 为这 5 次添加记录的操作进行打包，并增加一个 Submit()方法一次性地提交 5 次操作。

12.8 小结

外观模式的作用是将复杂“子系统”内部的各种外部调用通过一个集中外观对象包装并暴露出来。随着应用的发展，“子系统”的概念本身也被外延到“远程子系统”甚至是远程服务对象（互联网环境下，很多时候是 Web Service），本章针对这些情况进行了 Facade Interface、Remote Facade 和 WSDL 的讨论，考虑性能的需要还介绍了如何结合 DTO 把琐碎的调用“打包”以提高访问效率。最后，我们又把“子系统”的概念收缩到一个类内部，考虑如何通过 Fluent Interface 的方式设计支持连贯操作的接口。

第 17 章

解释器模式

17.1　说明

17.2　经典回顾

17.3　采用正则表达式

17.4　采用字典

17.5　采用 XSD

17.6　用 XSD 解释定制的业务语言

17.7　小结

17.1 说明

Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language..

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

解释器模式本身具有很强的业务适应性，它的实质就是“用双方熟悉的语言交流”，也就是保证当应用或对象与用户或其他对象交流的时候，采用最具实效性的方式完成，不过这些“语言”，计算机可能无法直接理解，因此需要增加一个解释器（也有可能是编译器）来完成这个过渡的工作。随着领域语言（DSL：Domain Specified Language）的盛行，以解释器模式完成更高级的开发环境或开发语言就显得愈发必要了。

下面，我们根据《编译原理》看看一个标准的编译过程，如图 17-1 所示。

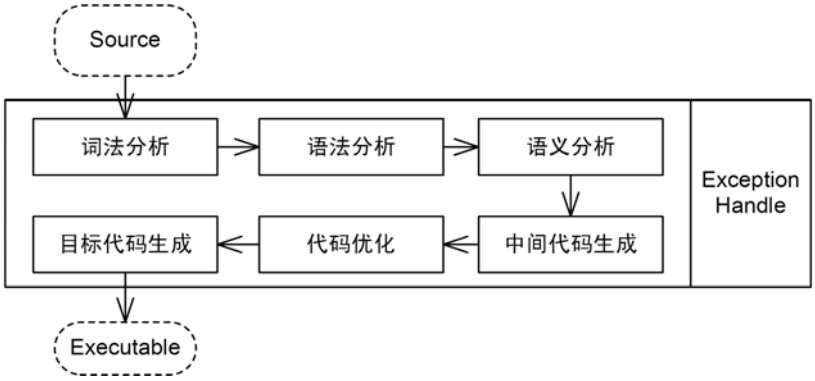


图 17-1 编译过程

其中每个步骤完成的工作如下。

- 词法分析：逐个字符读入并识别出各个单词。
- 语法分析：将单词序列分解成各类语法短语。
- 语义分析：审查源码是否有语义错误。
- 中间代码生成：将源码变成一种内部表示。（例如，.NET 语言将变成 IL。）
- 代码优化：对中间代码进行改善。
- 目标代码生成：把优化后的中间代码变成可执行的内容。

工程中我们可以根据应用需求，以及从长远考虑需要如何“贴近”业务的角度，选择以上任何一个步骤为切入点，另外还可以充分利用.NET 平台、正则表达式、XML 相关技术简化这些处理，从而为快速完成贴合业务使用的解释器努力。同时，在经典解释

器模式中，并没有对现在普遍使用的多芯多核系统作特别说明，而我们生产环境的服务器已经基本上被替换成多处理器环境，因此设计支持并行运算的解释器模式也是实际项目中要特别设计的内容，毕竟如何充分利用硬件的并行能力，不应该也不能只从编码层面考虑。

项目中我们常常会发现有些问题（业务或技术上的）出现得非常频繁，而且它们的变化也基本上以一些规律性的方式进行变化。对于这类问题，如果编写一个对象类进行处理，随着业务变更，我们将频繁地修改代码、编译、部署。与其反复做这种工作，不如把它们抽象为一个语言（语法定义可能很简单，也可能很复杂，如 BPEL 之类的协议或 VBScript 之类的语言）。这样我们的业务逻辑执行过程变成如图 17-2 所示。

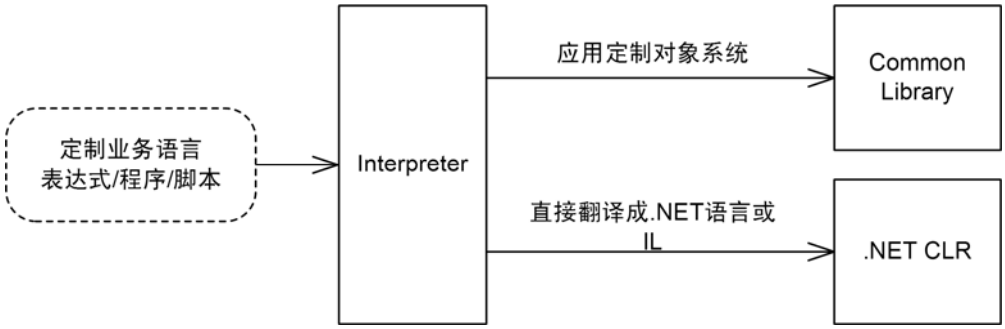


图 17-2 用解释器实现业务语言与运行环境间的映射

17.2 经典回顾

解释器模式适用于哪些环境呢？

- 虽然相关操作频繁出现，而且也有一定规律可循，但如果通过大量层次性的类来表示这种操作，设计上显得比较复杂。
- 执行上对效率的要求不是特别高，但对于灵活性的要求非常高。

我们来看一个例子，此例中，某个应用的授权系统除了控制用户、角色、功能和职责范围外，还需要与业务数据进行结合。

比如：虽然部门经理可以审批员工的办公用品申请，但如果某个申请单的金额大于 1 万，那么部门经理无法点击“审批”按钮。授权规则如下：

Expression

```
本部门员工（申请单） AND （申请单.金额<=10000）
或用自定义函数（UDF: User Defined Function）表示为
IsSubordination(Issue.EmployeeID) AND (Issue.Fee <= 10000)
```

类似的规则常常会发生更改，比如增加一条：如果员工本身是行政助理，他收集全部门办公用品单，为了简化手续，每个部门的办公用品可以由他一个人挂名申请，因此金额可以大于 1 万。这样我们需要修改这个表达式。所以，在比较大型项目的实施上，我们可以考虑增加一个能读懂这个表达式的子系统，在牺牲一些效率的情况下，专门解释执行类似的表达式。

该模式的静态结构如图 17-3 所示。

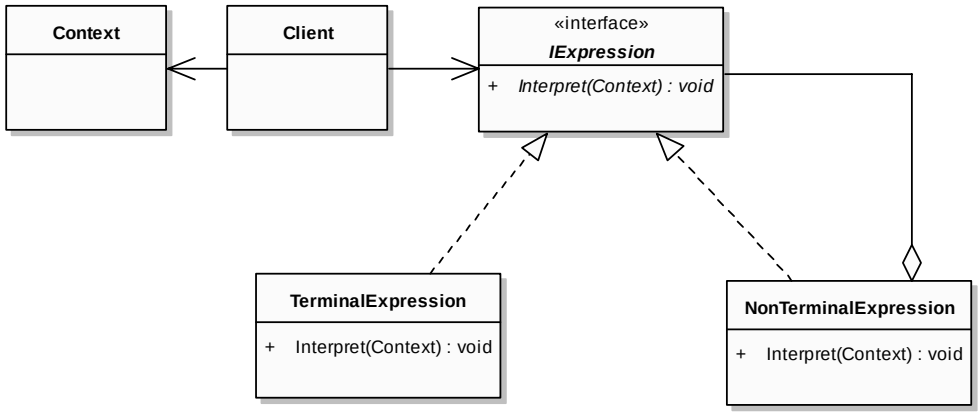


图 17-3 经典解释器模式的静态结构

下面我们用一个仅支持个位整数加、减法的小计算器程序，看一下如何实现经典的解释器模式，其代码如下：

C# 抽象的表达式（语言）对象及 Context 对象

```
/// 用于保存计算过程的中间结果及当前执行的操作符
public class Context
{
    public int Value;
    public char Operator;
}

/// 表示所有表达式的抽象接口
public interface IExpression
{
    /// 用 Context 负责保存中间结果
    void Evaluate(Context context);
}
```

C# 拆分后的表达式元素，包括操作符和操作数

```
/// 表示所有操作符
public class Operator : IExpression
{
    private char op;
    public Operator(char op) { this.op = op; }
```

```
    public virtual void Evaluate(Context context) { context.Operator = op; }
}

/// 表示所有操作数
public class Operand : IExpression
{
int num;

public Operand(int num) { this.num = num; }

    /// 根据操作符执行计算
    public virtual void Evaluate(Context c)
    {
        switch (c.Operator)
        {
            case '\0': c.Value = num; break;
            case '+': c.Value += num; break;
            case '-': c.Value -= num; break;
        }
    }
}
```

C#

```
/// 解析器
public class Calculator
{
    public int Calculate(string expression)
    {
        Context context = new Context();
        IList<IExpression> tree = new List<IExpression>();

        // 词法和语法分析
        char[] elements = expression.ToCharArray();
        foreach (char c in elements)
        {
            if ((c == '+') || (c == '-'))
                tree.Add(new Operator(c));
            else
                tree.Add(new Operand((int)(c - 48)));
        }

        // 遍历执行每个中间过程
        foreach (IExpression exp in tree)
            exp.Evaluate(context);
        return context.Value;
    }
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    Calculator calculator = new Calculator();
    Assert.AreEqual<int>(1 + 3 - 2, calculator.Calculate("1+3-2"));
}
```

上面的例子采用一种相对“中规中矩”的方式实现经典的解释器模式，其中我们不难看出几个关键对象的作用。

- Context：由于解释过程经常会存在递归和迭代，所以这里 Context 相当于一个全局的容器，我们可以把各种需要全局使用的内容、属性保存在 Context 里面。

示例中有两个属性，Value 表示中间结果，Operator 代表计算符，而且是当前计算要用的。

- IExpression：所有计算表达式的抽象接口，为了简化计算过程，它只有一个方法 Evaluate()，表示当前表达式节点及其分支下所有节点的计算。
- 根据计算的不同，我们在 IExpression 之上重新定义了新的计算对象，包括操作符和操作数。

不过上面的示例仅针对一个最简单的计算，下面我们介绍一些满足更复杂语言的“捷径”。

17.3 采用正则表达式

对于很多开发人员而言，平时做得最多的定制化计算可以用正则表达式（Regular Expressions）完成。正则表达式本身是基于模式匹配的，它解决了开发中最经常做的工作——搜索（及根据搜索结果进行的替换）。如果应用对行业的数据信息有明确要求，这时候正则就非常有用了。

下面是一些常见的情景：

- 电话号码。
- 邮政编码。
- 身份证编号（及社保编号）。
- IP 地址。
- URL。
- HTML、JavaScript 中的某些 Tag 或 Comment。
- 信用卡编号。
- ……

按照面向对象的体系，我们可以把正则表达式作为解析器模式的一个具体化情况，如图 17-4 所示。

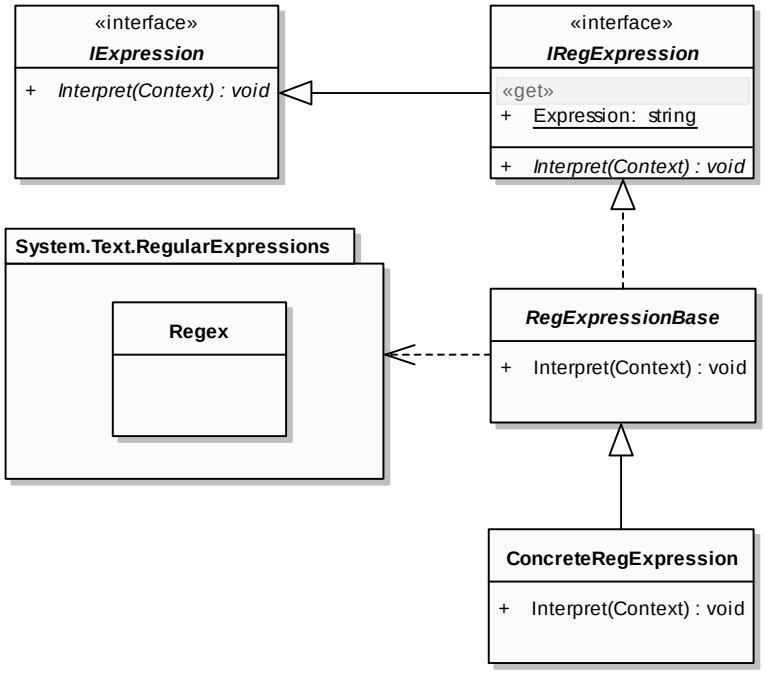


图 17-4 基于正则表达式的解释器

至于如何把每个 IRegExpression 用于解析整段的业务表达式，可以参考上面的示例，这里仅介绍每个具体正则表达式解析对象的设计，其代码如下：

C# 定义 Context 和表达式解析接口

```
/// 用于保存计算过程的中间结果及当前执行的操作符
/// 根据 Regex 的功能操作符包括: Matches(M)、Replace(R)
public class Context
{
    /// 文本内容
    public string Content;

    /// 'M' Matches /'R' Replace
    public char Operator;

    /// 匹配字符串集合
    public IList<string> Matches = new List<string>();

    /// 用于替换的文本内容
    public string Replacement;
}

/// 采用正则表达式方式表示的抽象接口
public interface IRegExpression : IExpression
{
```

```
    /// 是否匹配
    bool IsMatch(string content);
}
```

C# 为减轻每个具体正则表达式解析类型的负担而增加的抽象基类

```
/// 采用正则表达式方式表示的抽象基类
public abstract class RegExpressionBase : IRegExpression
{
    protected Regex regex;
    public RegExpressionBase(string expression)
    {
        regex = new Regex(expression,
          RegexOptions.Compiled | RegexOptions.IgnoreCase);
    }

    public virtual bool IsMatch(string content) {
       return regex.IsMatch(content); }

    /// 解析表达式
    public virtual void Evaluate(Context context)
    {
        if (context == null) throw new ArgumentNullException("context");
        switch (context.Operator)
        {
            case 'M':
                EvaluateMatch(context);
                break;
            case 'R':
                EvaluateReplace(context);
                break;
            default:
                throw new ArgumentException();
        }
    }

    /// 通过 Match 方式解析表达式，可以被子类覆盖
    protected virtual void EvaluateMatch(Context context)
    {
        context.Matches.Clear();
        MatchCollection coll = regex.Matches(context.Content);
        if (coll.Count == 0) return;
        foreach (Match match in coll)
            context.Matches.Add(match.Value);
    }

    /// 通过 Replace 方法替换内容，可以被子类覆盖
    protected virtual void EvaluateReplace(Context context)
    {
        context.Content = regex.Replace(context.Content, context.Replacement);
    }
}
```

C# 具体正则表达式解析类型

```
/// 电子邮件
public class EmailRegExpression : RegExpressionBase
{
    public EmailRegExpression()
        : base(@"\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*") { }
}

/// 汉字
public class ChineseRegExpression : RegExpressionBase
{
    public ChineseRegExpression()
        :base(@"^[\u4e00-\u9fa5]{0,}$") { }
}

/// 普通的字符串
public class StringRegExpression : RegExpressionBase
{
    public StringRegExpression(string expression) : base(expression) { }
}
```

Unit Test

```
[TestMethod]
public void TestEmailExpression()
{
    Context context = new Context();
    context.Content = "zhao@hotmail.com, liu@gmail.com, test";

    context.Operator = 'M'; // 查找匹配信息
    IRegExpression expression = new EmailRegExpression();
    expression.Evaluate(context);

    // 确认是否发现其中包括两个 Email 信息
    Assert.AreEqual<int>(2, context.Matches.Count);
    Assert.AreEqual<string>("zhao@hotmail.com", context.Matches[0]);
    Assert.AreEqual<string>("liu@gmail.com", context.Matches[1]);
}

[TestMethod]
public void TestChineseExpression()
{
    Context context = new Context();
    context.Content = "测试";
    context.Operator = 'M'; // 查找匹配信息

    IRegExpression expression = new ChineseRegExpression();
    expression.Evaluate(context);
    Assert.AreEqual<int>(1, context.Matches.Count);
    context.Content = "测试 test";
    expression.Evaluate(context);
    Assert.AreEqual<int>(0, context.Matches.Count);
```

```
}

[TestMethod]
public void TestReplacement()
{
    Context context = new Context();
    context.Content = "a   b   c   d";
    context.Operator = 'R'; // 查找匹配信息

    context.Replacement = ",";
    IRegExpression expression = new StringRegExpression(@"\s+");
    expression.Evaluate(context);
    Assert.AreEqual<string>("a,b,c,d", context.Content);
}
```

正则表达式是个强有力的工具，通过它我们可以比较好地结合经典解释器模式的实现框架，简化相关搜索、替换的工作，为高层解析器调度机制的实现提供便利。同时这里的正则表达式也可以为后面的策略模式提供很多便利，尤其是 IsMatch()对批量策略匹配进行检验的时候更有效。

17.4 采用字典

项目中还有很多情景是需要进行信息翻译的，无论是输入信息还是输出信息，或者是进行“业务/技术”内容转换的，采用解释器模式都是必要的，可以把它们作为特殊的表达式（IExpression）对象来使用。

主要包括下述情况：

- 参数的翻译。
- 多区域或多参数系统的翻译。
- 数据字段的翻译。
- ……

在完成这些内容的翻译后，我们可以继续沿用最初那个计算器示例，把它们组织到解释器的统一调度和管理之下，为完成相对复杂的业务语言解析提供便利。这些基于字典信息的数量差异很大，而且实际项目中会采用枚举、配置文件、参数文件、资源文件、数据库、XML 文档等多种方式组织，因此在 IExpression 之外，为了便于描述和组织不同字典信息的存储，还需要额外抽象出一个 IDictionaryStore 对象。使用中翻译过程常常是双向的，比如：UI 显示部分进行参数到汉字说明的翻译，而 UI 的请求提交服务端部分又进行汉字说明到参数的翻译，所以不同于一般的 IDictionary<K, T>，这里还需要借助 Context 对象实现从 Value 到 Key 的逆向翻译。

因此，新的静态结构如图 17-5 所示。

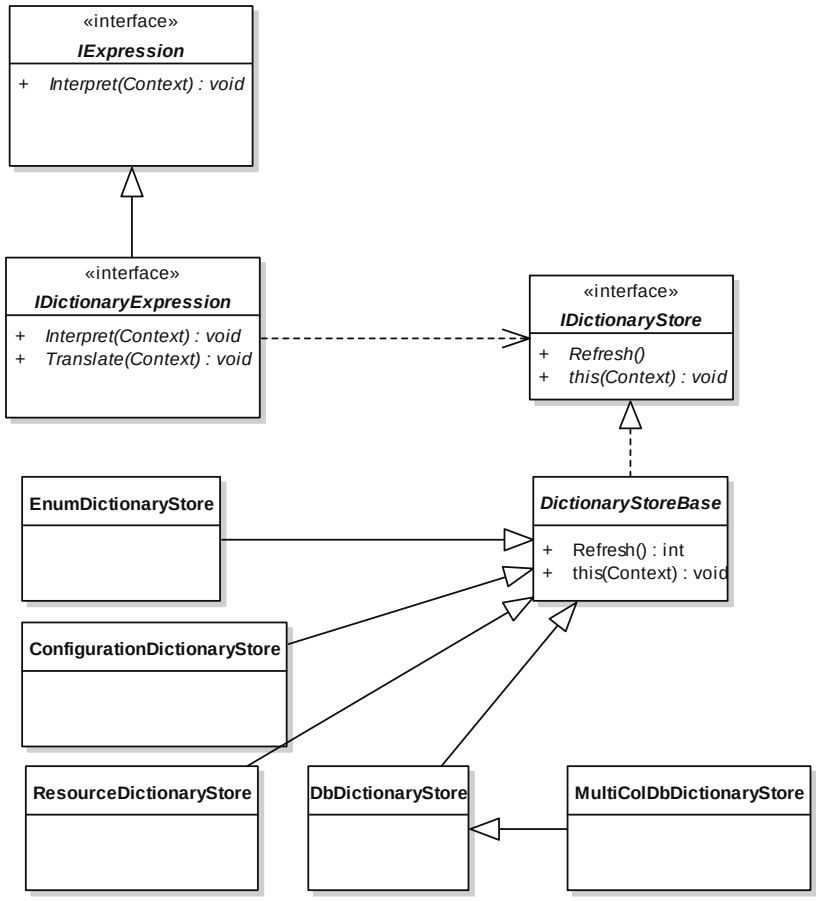


图 17-5 基于字典的解释器

下面我们以简单的枚举和 IDictionary<string, string>为示例，看一下基于字典的解释器表达式实现，其代码如下：

C# 定义 Context 和表达式解析接口

```
/// 适用于字典信息的 Context 对象
public class Context
{
    public object Key;
    public object Value;

    /// 'T'(to) 根据 Value 获得 Key
    /// 'F'(from) 根据 Key 获得 Value
    public char Operator;
}

/// 抽象字典信息存储访问对象
public interface IDictionaryStore
{
```

```
    ///基于配置、数据库等信息的 Store 对象可以通过
    /// 该方法重新加载相应的缓冲信息
    void Refersh();

    /// 根据 Context 定义的 Key/Value 及访问方向提取信息
    void Find(Context context);
}

/// 具有字典信息的表达式对象接口
public interface IDictionaryExpression : IExpression
{
    IDictionaryStore Store { get; set;}
}
```

C# 具体正则表达式解析类型

```
public class SimpleDictionaryExpression : IDictionaryExpression
{
    protected IDictionaryStore store;
    public virtual IDictionaryStore Store
    {
        get { return store; }
        set { store = value; }
    }

    public virtual void Evaluate(Context context)
    {
        store.Find(context);
    }
}
```

C# 具体的 IDictionaryStore 类型

```
/// 基于枚举项的 DictionaryStore
public class EnumDictionaryStore<T> : IDictionaryStore
{
public void Refersh(){} // 对于枚举类型暂时用不到

    /// 根据字典操作方向进行字符串(value)/枚举值(key)的翻译工作
    public void Find(Context context)
    {
        switch (context.Operator)
        {
            case 'F':   // 'F'(from) 根据 Key 获得 Value
                context.Value = ((T)(context.Key)).ToString();
                break;
            case 'T':   // 'T'(to) 根据 Value 获得 Key
                context.Key = Enum.Parse(typeof(T), (string)context.Value);
                break;
            default:
                throw new ArgumentException();
        }
    }
}

/// 基于 Dictionary<string, string>的 DictionaryStore
public class StringDictionaryStore : IDictionaryStore
{
```

```
    public void Refersh() { }
    // 对于 Dictionary<string, string> 类型，暂时用不到该方法

    /// 定义数据内容
    protected IDictionary<string, string> data;
    public virtual IDictionary<string, string> Data
    {
        get { return data; }
        set { data = value; }
    }

    /// 根据字典操作方向进行字符串(value)/枚举值(key)的翻译工作
    /// 即便存在不同的 Key 对应同一个 Value 的情况，也只返回找到的第一个 Key
    public void Find(Context context)
    {
        if(Data == null) throw new NullReferenceException("Data");
        string value;
        switch (context.Operator)
        {
            case 'F':   // 'F'(from) 根据 Key 获得 Value
                if (!Data.TryGetValue((string)context.Key, out value))
                    context.Value = string.Empty;
                else
                    context.Value = value;
                break;
            case 'T':   // 'T'(to) 根据 Value 获得 Key
                value = (string)context.Value;
                foreach (string key in Data.Keys)
                    if (string.Equals(Data[key], value))
                    {
                        context.Key = key;
                        return;
                    }
                context.Key = string.Empty;
                break;
            default:
                throw new ArgumentException();
        }
    }
}
```

Unit Test

```
enum Color { Red, Green, Blue }

private IDictionary<string, string> data;
private void PreProcess()
{
    data = new Dictionary<string, string>();
    data.Add("R", "Red");
    data.Add("G", "Green");
    data.Add("B", "Blue");
}

[TestMethod]
public void Test()
{
```

```
    // 准备环境
    IDictionaryExpression expression = new SimpleDictionaryExpression();
    IDictionaryStore store = new EnumDictionaryStore<Color>();
    expression.Store = store;
    Context context = new Context();

    // 测试 Enum 从 Key 到 Value 的解析过程
    context.Key = Color.Red;
    context.Operator = 'F'; // from key to value
    expression.Evaluate(context);
    Assert.AreEqual<string>("Red", context.Value as string);

    // 测试 Enum 从 Value 到 Key 的解析过程
    context.Value = "Blue";
    context.Operator = 'T';
    expression.Evaluate(context);
    Assert.AreEqual<Color>(Color.Blue, (Color)(context.Key));

    // 测试另一个 DictionaryStore 框架的解析能力
    StringDictionaryStore another = new StringDictionaryStore();
    PreProcess();
    another.Data = data;
    expression.Store = another;
    expression.Evaluate(context);
    Assert.AreEqual<string>("B", context.Key as string);
}
```

上面示例中我们定义了基于字典的解释器表达式对象，实际项目中它们有什么作用呢？以本章最前面的例子来考察这个问题：

Expression

```
本部门员工（申请单） AND （申请单.金额<=10000）
或用自定义函数（UDF: User Defined Function）表示为
IsSubordination(Issue.EmployeeID) AND (Issue.Fee <= 10000)
```

实际的业务规则可能保存在数据库里，而“Issue.EmployeeID”和“Issue.Fee”则将在词法解析后用来根据 DataTable、DataRow 或实际的 ORM 对象进行翻译，根据当前业务实体被翻译成它们的值，这样外层业务规则引擎只需要通过翻译自定义函数（IsSubordination）就可以实现对这个非常业务化规则的解析过程。不过“IsSubordination(Issue.EmployeeID) AND (Issue.Fee <= 10000)”似乎对于业务人员不够友好，它们更倾向于上面中文的那个表示方式“本部门员工（申请单） AND （申请单.金额<=10000）”，这时候我们可以用一个字典表达式对象完成 UI 的输入、显示、提取、提交的过程，操作步骤和本节示例雷同。

17.5 采用 XSD

上面的准备对于一两个项目也许是适用的，但对于一些信息化程度更高的企业而言恐怕就不够了，原因在于后者内部可能同时有很多部门，仓储部门的领域语言与财务部门的领域

语言也许存在很大差异，按照解释器模式的说明，我们可能需要实现多个语言的解释器。不过，我们也可以参考.NET 的实现，在同一套 CLS 上基于一个 IL 实现多种.NET 语言的混合开发。

实现企业（甚至行业）应用时，我们可以在每个具体解释器上实现更加通用的框架，这个框架只能对一些“原语”性的语言进行解析，上面是各个具体的解释器，它们的作用就是把不同业务领域的语言翻译成中间语言（“原语”语言）。

以税务为例，税单和发票是两个主要的业务领域，另外在办公环境还有一套主要针对公文的领域语言，但是从它们的语义看都可能存在“提取”、“保存”、“是否匹配”等业务表达式，这些内容就可以抽象成一个独立的中间语言。新的解释器结构如图 17-6 所示。

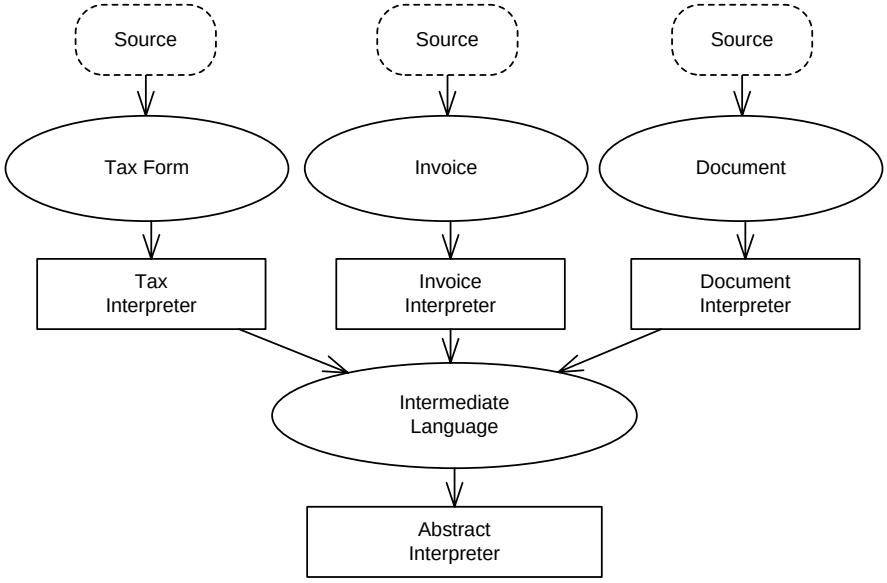


图 17-6 服务于更多领域语言的解释器体系

17.6 用 XSD 解释定制的业务语言

另外，对于更复杂的业务语言，以往我们表示语法时一般采用语法树的形式，源码需要经过一系列相对复杂的流程完成词法、语法解析之后才真正进行实际业务语义的翻译。不过随着 XML 技术的发展，我们可以比较容易地把树型结构换成以 XSD（XML Schema）方式描述，判断一段代码是否满足 XSD 定义的语法树要求，只需要通过 XSD 的验证即可，至于解释器提取每个表达式则可以通过 DOM 方式完成。

例如，我们可以定义如图 17-7 所示的语法。

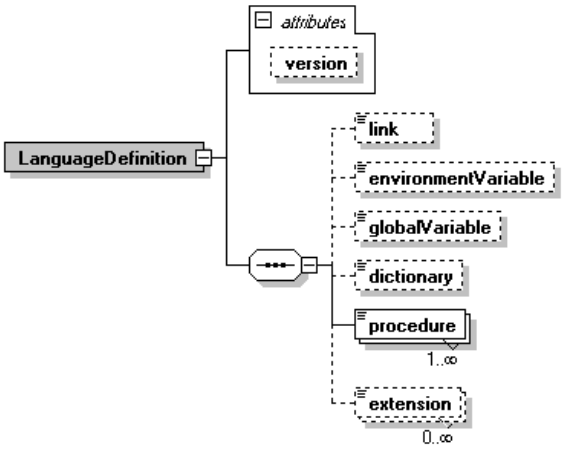


图 17-7 用 XSD 定义的语法树的解释器

其中：

- <LanguageDefinition>是每段语言的根节点。
- <link>相当于我们常用的 Reference，用于引用外部其他语言片断或 WS-*。
- <environmentVariable>主要用于对客户端、服务端环境信息的提取，包括用户名称、服务器地址等。
- <globalVariable>相当于一般开发语言里的头文件，定义一些常量、通用的数据类型等。
- <dictionary>主要提供各种字典表达式及参数信息。
- <procedure>实际的业务逻辑执行部分。
- <extension>扩展部分。

然后我们在这个 XSD 语法定义上增加一个解析器，这样就可以应付功能特性相对更完善、但语言本身也更复杂的环境了。虽然实现这样一个解析器的代价相对很大，但后续的业务逻辑可以用自己定义的业务语言编写，对于一些大型或需要长期维护的业务系统而言，这个代价也许是值得的。

17.7 小结

大部分情况下，解释器模式被用来解决单纯堆叠类结构难于应付业务变化的问题，这里有两个关键点：自定义的语言和那个 Context 对象，它们是贯穿解释器始终的对象，至于解释器的骨架则是由一个个表达式对象完成的，解释器的作用是把 Context 放进去，然后调度一个个表达式对象，直至完成整个语言的解释过程。

第 22 章

观察者模式

22.1 说明

22.2 经典回顾

22.3 .NET 内置的 Observer 机制——事件

22.4 具有 Observer 的集合类型

22.5 面向服务接口的 Observer

22.6 小结

22.1 说明

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically..

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

观察者模式主要用于 1:N 的通知发布机制，它希望解决一个对象状态变化时可以及时告知相关依赖对象的问题，令它们也可以做出响应。生活中类似的案例很多，大至自上而下的政策调整、奥运赛场信息的发布，小至项目中一个需求点的调整，我们都可以发现存在类似的通知要求，而且无论是会议、人力还是工具，其中都存在一个角色，它的作用就是保持对相关问题（或称之为主题）的关注，在问题发生变化的时候是“TA”把消息通知相关各方。观察者模式采用的就是类似的方法，它抽象一类对象（观察者）专门负责“盯着”目标对象，当目标对象状态有变动的时候，每个观察者就会获得通知并做出响应。观察者模式解决的也是调用通知关系带来的依赖。

用一段代码作为示例也许更为直观，这里假设类型 X 状态更新后要同时告知 A、B、C 三个类型，那么一个直观的实现方式如下：

C#

```
/// <summary>
/// A\B\C 为希望获得 X 通知的类型
/// </summary>
public class A
{
    public int Data;
    public void Update(int data) { this.Data = data; }
}
public class B
{
    public int Count;
    public void NotifyCount(int data) { this.Count = data; }
}
public class C
{
    public int N;
    public void Set(int data) { this.N = data; }
}

/// X 在自己属性更新的时候需要同时通知多个对象
public class X
{
    private int data;

    public A instanceA;
    public B instanceB;
    public C instanceC;

    public void SetData(int data)
    {
        this.data = data;
```

```
        instanceA.Update(data);
        instanceB.NotifyCount(data);
        instanceC.Set(data);
    }
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    A a = new A();
    B b = new B();
    C c = new C();
    X x = new X();

    x.instanceA = a;
    x.instanceB = b;
    x.instanceC = c;

    x.SetData(10);
    Assert.AreEqual<int>(10, a.Data);
    Assert.AreEqual<int>(10, b.Count);
    Assert.AreEqual<int>(10, c.N);
}
```

这里类型 X 只需引用 A、B、C 三个类型即可，但这样会给类型 X 带来似乎过多的依赖关系，因为它需要同时引用三个类型。不难发现，其实 A、B、C 三个类型本身都有类似的执行更新的方法，如果对它们进行抽象就可以让 X 仅仅依赖于一个抽象类型，于是对代码进行如下修改：

C#

```
/// 对类型 A/B/C 做抽象后的接口
public interface IUpdatableObject
{
    int Data { get;}
    void Update(int newData);
}

/// 具体的待更新类型
public class A : IUpdatableObject
{
    private int data;
    public int Data{get { return this.data;}}
    public void Update(int newData) { this.data = newData; }
}
// B, C……

public class X
{
    private IUpdatableObject[] objects = new IUpdatableObject[3];

    public IUpdatableObject this[int index] { set { objects[index] =
      value; } }

    private int data;
```

```
    public void Update(int newData)
    {
        this.data = newData;
        foreach (IUpdatableObject obj in objects)
            obj.Update(newData);
    }
}
```

Unit Test

```
[TestMethod]
public void Test()
{
    X x = new X();
    IUpdatableObject a = new A();
    IUpdatableObject b = new B();
    IUpdatableObject c = new C();
    x[0] = a;
    x[1] = b;
    x[2] = c;

    x.Update(10);
    Assert.AreEqual<int>(10, a.Data);
    Assert.AreEqual<int>(10, b.Data);
    Assert.AreEqual<int>(10, c.Data);
}
```

这样做有什么区别呢？我们会发现如果后面示例中 A\B\C 实例可以通过其他方式注入，客户程序仅需要依赖一个抽象的接口——IUpdatableObject。变化后的结构如图 22-1 所示。

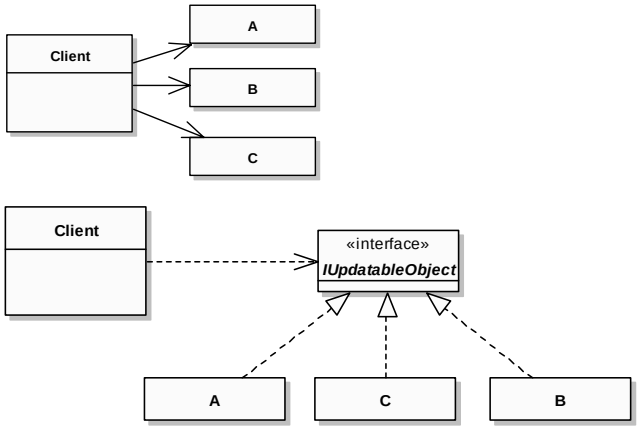


图 22-1 抽象前后的 1:N 通知机制

通过一个简单的抽象，客户程序所依赖的类型变成一个，这样为其带来进一步扩展的灵活性，而且不容易因为 A\B\C 的变化受到影响。

22.2 经典回顾

上面的分析还有一个问题没有说明，即对目标对象的状态修改的抽象，不仅存在修改某

个对象需要其他几个对象的情况，而且很多对象都存在类似的情况。根据我们面向对象的经验，这时候非常有必要对目标对象本身也进行一次抽象，这样任何符合类似特征的对象都可以采用类似的方式做到松散耦合的通知。观察者模式的静态结构如图 22-2 所示。

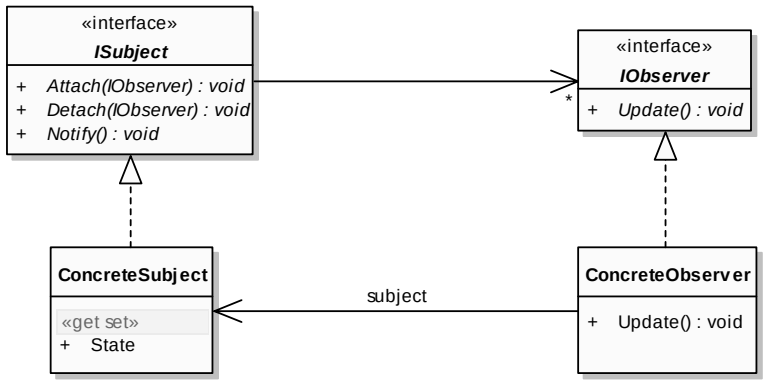


图 22-2 经典观察者模式的静态结构

该模式不仅对目标对象进行了抽象，同时对它保持关注的对象（观察者）也进行了抽象，通过在目标对象维护一个观察者对象列表的方式，当状态变更的时候进行逐个通知。它的主要应用情景如下：

- 当存在一类对象通知关系上依赖于另一类对象的时候，把它们进行抽象，确保两类对象的具体实现都可以相对独立的变化，但它们交互的接口保持稳定。
- 一个类型状态变化时，需要通知未知数量的其他对象，例如：我们第一个例子中 X 明确知道有 A\B\C 各一个实例需要通知，但如果增加 D 类型或有两个 A 需要通知，X 就需要频繁修改了。
- 最重要的是提供了目标对象与希望获得通知的对象间松散耦合。

下面我们看一个示例，其代码如下：

C# 抽象定义部分

```
/// 观察者类型接口
public interface IObserver<T>
{
    void Update(SubjectBase<T> subject);
}

/// 目标对象抽象类型
public abstract class SubjectBase<T>
{
    /// 登记所有需要通知的观察者
    protected IList<IObserver<T>> observers = new List<IObserver<T>>();

    protected T state;
    public virtual T State{get{return state;}}
```

```
    /// Attach
    public static SubjectBase<T> operator +(
    SubjectBase<T> subject, IObserver<T> observer)
    {
        subject.observers.Add(observer);
        return subject;
    }

    /// Detach
    public static SubjectBase<T> operator -(
    SubjectBase<T> subject, IObserver<T> observer)
    {
        subject.observers.Remove(observer);
        return subject;
    }

    /// 更新各观察者
    public virtual void Notify()
    {
        foreach (IObserver<T> observer in observers)
            observer.Update(this);
    }

    /// 供客户程序对目标对象进行操作的方法
    public virtual void Update(T state)
    {
        this.state = state;
        Notify();   // 触发对外通知
    }
}
```

C# 具体类型部分

```
/// 具体目标类型
public class Subject<T> : SubjectBase<T> { }

/// 具体观察者类型
public class Observer<T> : IObserver<T>
{
    public T State;
    public void Update(SubjectBase<T> subject)
    {
        this.State = subject.State;
    }
}
```

Unit Test

```
[TestClass]
public class TestObserver
{
    /// <summary>
    /// 验证目标类型对观察者类型的:N 通知
    /// </summary>
    [TestMethod]
    public void TestMulticst()
    {
        SubjectBase<int> subject = new SubjectA<int>();
        Observer<int> observer1 = new Observer<int>();
```

```
        observer1.State = 10;
        Observer<int> observer2 = new Observer<int>();
        observer2.State = 20;

        // Attach Observer
        subject += observer1;
        subject += observer2;

        // 确认通知的有效性
        subject.Update(1);
        Assert.AreEqual<int>(1, observer1.State);
        Assert.AreEqual<int>(1, observer2.State);

        // 确认变更通知列表后的有效性
        subject -= observer1;
        subject.Update(5);
        Assert.AreNotEqual<int>(5, observer1.State);
        Assert.AreEqual<int>(5, observer2.State);
    }

    /// <summary>
    /// 验证同一个观察者对象可以同时“观察”多个目标对象
    /// </summary>
    [TestMethod]
    public void TestMultiSubject()
    {
        SubjectBase<int> subjectA = new SubjectA<int>();
        SubjectBase<int> subjectB = new SubjectA<int>();
        Observer<int> observer = new Observer<int>();
        observer.State = 20;
        subjectA += observer;
        subjectB += observer;

        subjectA.Update(10);
        Assert.AreEqual<int>(10, observer.State);
        subjectB.Update(5);
        Assert.AreEqual<int>(5, observer.State);
    }
}
```

我们分析一下这个例子：

- SubjectBase 本身不知道会有哪些 IObserver 类型希望获得它的更新通知。
- 具体的 Observer 类型也并不需要真正了解目标类型是什么，知道它们是 SubjectcBase 即可。
- SubjectBase 通过一个观察者列表逐个通知，单元测试验证其有效。
- 同样，每一个观察者由于仅依赖于抽象的目标类型，因此一个观察者实际上可以跟踪多个目标类型。

下面我们讨论一下观察者如何更新自己数据，当目标对象状态更新的时候，观察者可以通过以下两种方式更新信息：

- 目标对象在通知里把需要更新的信息作为参数提供给 IObserver 的 Update()方法。

- 目标对象仅仅告诉观察者有新的状态，至于该状态是什么，观察者自己访问目标对象来获取。

前者我们称之为“推”方式，更新的数据是目标对象硬塞给观察者的；后者被称为“拉”模式，是观察者主动从目标对象拽下来的。从面向对象来看，它们的优劣如下：

- “推”方式每次都会把通知以广播方式发送给所有观察者，所有观察者只能被动接受，如果通知的内容比较多，多个观察者同时接受可能会对网络、内存（可能还会涉及 I/O）有比较大的影响。
- “拉”方式更加自由，它只要知道“有情况”就可以了，至于什么时候获取、获取哪些内容、甚至是否要获取都可以自主决定，但这也带来两个问题，如果某个观察者“不紧不慢”，它可能会漏掉之前通知的内容；另外它必须保存一个对目标对象的引用，而且需要了解目标对象的结构，即产生了一定依赖。

项目中将这两种方式区分开来，其更主要原因来自于安全性要求，原则上我们都希望高信任区域可以读写低信任区域，而低信任区域不能写高信任区域，很多时候连读都不允许，这时候“推”模式比较适合高信任区域不信任低信任区域的写方式，而“拉”一般要求高信任区域信任某个低信任区域的访问，或者就是高信任区域访问低信任区域。

我们上面的示例其实是综合两者优缺点后一个折中的办法，整体上看比较贴近“推”方式，但推送的内容并不是实际的状态，而是一个对抽象目标对象的引用，观察者可以根据需要通过这个引用访问到状态，从这个角度看又是“拉”方式（见图 22-3）。

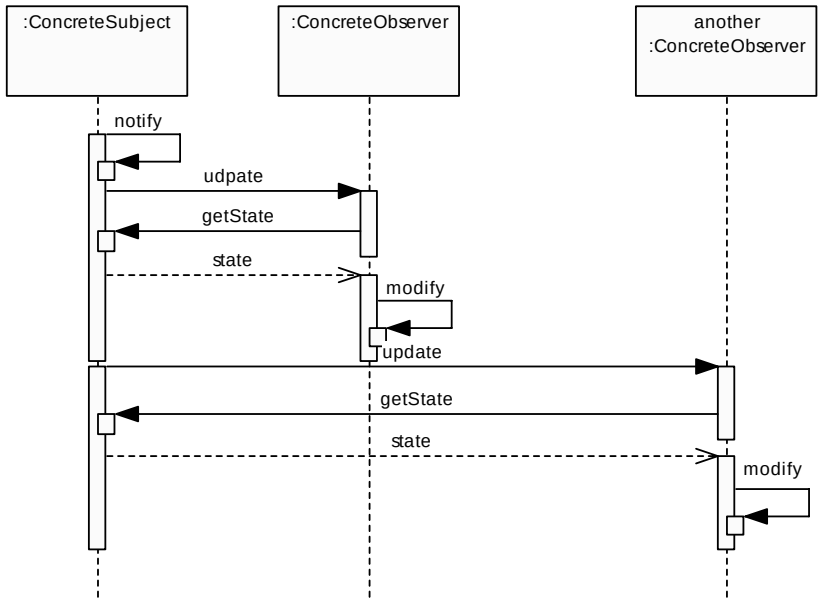


图 22-3 观察者模式的时序

22.3 .NET 内置的 Observer 机制——事件

前面的章节里我们已经多次介绍过委托和事件，从其执行效果看，.NET 的事件处理过程本身就是个观察者模式的范例，前面讨论过，虽然从直接的编码角度看，.NET CLR 对上层众多事件的调度属于中介者方式，但从应用层的使用看事件所定义的委托类型本身就是个抽象的观察者，相对于上面示例中的观察者接口而言，它令应用结构更加松散耦合，实际上直接依赖的除了作为目标对象的 EventArgs 以外就是.NET CLR 自己了。

C#

```
/// 具体的 Subject
public class UserEventArgs : EventArgs
{
    private string name;
    public string Name { get { return this.name; } }

    public UserEventArgs(string name) { this.name = name; }
}

public class User
{
    public event EventHandler<UserEventArgs> NameChanged;

    private string name;
    public string Name
    {
        get { return name; }
        set{
            name = value;
            // notify
            NameChanged(this, new UserEventArgs(value));
        }
    }
}
```

Unit Test

```
private void OnNameChanged(object send, UserEventArgs args)
{
    Assert.AreEqual<string>("joe", args.Name);
}

[TestMethod]
public void Test()
{
    User user = new User();
    user.NameChanged += this.OnNameChanged;
    user.Name = "joe";
}
```

上面是一个简单的.NET 事件，其中观察者是满足.NET 事件委托要求（例如 NameChanged）所定义的具体方法， 相对于我们常说的抽象观察者接口和具体观察者对象而言，.NET 的委托机制更为简单，从编码上看就是抽象一个方法签名。

不难看出这个结构更加松散，客户程序不必实现一个 IObserver 接口，.NET 事件机制是.NET 内置的，观察者与目标对象之间也是松散耦合的，观察者可以随时通过重载的+=、–=的建立与目标对象的联系，同时所有的目标类型都是继承自 EventArgs 的，它是通用的。对于使用普通委托的情况而言，每个 delegate 定义其实可视为使用 MulticastDelegate 做组播，它可以满足我们对更多委托参数（即更多主题内容）的要求。

因此，我们建议：除非类型参数或配置体系有专门的要求，必须要定义自己的 IObserver 接口，项目中可以考虑用松散的委托或事件方式设计观察者。甚至我们在设计应用的时候，尤其是对应用交互部分可以采用事件驱动（EDA：Event-Driven Architect）方式，业务实体对象也好，业务流程对象也好，把对外部应用的切入点设计为各种事件，这样它们执行中可以按照自己的方式处理，至于会有多少人“盯着”其行为，操作本身并不理会。

22.4 具有 Observer 的集合类型

工程中很多时候我们会使用集合类型（例如：IList<T>、IDictionary<TKey, TValue>）来缓存一系列信息，这些信息往往会被多个模块共享，如果缓存信息发生变更，有可能需要通知到各使用方，我们可以采用为集合类型增加事件的方式令它们具有 Observer 机制。

.NET Framework 提供的常用集合类型——Dictionary<TKey, TValue>、List<T>，并没有提供更新事件，估计其本意也是为了提高集合操作效率，所以这里就采用两套相对简便的方式来实现示例。

其代码如下：

C# 类型结构定义

```
/// 用于保存集合操作中操作条目信息的时间参数
public class DictionaryEventArgs<TKey, TValue> : EventArgs
{
    private TKey key;
    private TValue value;

    public DictionaryEventArgs(TKey key, TValue value)
    {
        this.key = key;
        this.value = value;
    }

    public TKey Key { get { return key; } }
    public TValue Value { get { return value; } }
}

/// 具有操作事件的 IDictionary<TKey, TValue>接口
public interface IObserverableDictionary<TKey, TValue> :
    IDictionary<TKey, TValue>
{
    EventHandler<DictionaryEventArgs<TKey, TValue>> NewItemAdded { get; set;}
```

```
}

/// 一种比较简单的实现方式
public class ObserverableDictionary<TKey, TValue> :
   Dictionary<TKey, TValue>, IObserverableDictionary<TKey, TValue>
{
   protected EventHandler<DictionaryEventArgs<TKey, TValue>> newItemAdded;
   public EventHandler<DictionaryEventArgs<TKey, TValue>> NewItemAdded
   {
      get { return newItemAdded; }
      set { newItemAdded = value; }
   }

   /// 为既有操作增加事件
   public new void Add(TKey key, TValue value)
   {
      base.Add(key, value);
      if (NewItemAdded != null)
         NewItemAdded(this, new DictionaryEventArgs<TKey, TValue>(key,
           value));
   }
}
```

Unit Test

```
[TestClass]
public class TestObserver
{
   string key = "hello";
   string value = "world";

   public void Validate(object sender,
     DictionaryEventArgs<string, string> args)
   {
      Assert.IsNotNull(sender);
      Type expectedType = typeof(ObserverableDictionary<string, string>);
      Assert.AreEqual<Type>(expectedType, sender.GetType());

      Assert.IsNotNull(args);
      expectedType = typeof(DictionaryEventArgs<string, string>);
      Assert.AreEqual<Type>(expectedType, args.GetType());

      Assert.AreEqual<string>(key, args.Key);
      Assert.AreEqual<string>(value, args.Value);
   }

   [TestMethod]
   public void Test()
   {
      IObserverableDictionary<string, string> dictionary =
         new ObserverableDictionary<string, string>();
      dictionary.NewItemAdded += this.Validate;
      dictionary.Add(key, value);
   }
}
```

22.5 面向服务接口的 Observer

在 Web Service 世界中同样可以借助事件机制完成两个服务对象间的通知——观察方式，比较主要的标准是微软联合相关厂商推出的 WS-Eventing，不过它采用的是观察者模式的一个更进一步的变体——出版/预定模式，出版/预定模式我们会在后面的架构模式部分作介绍，本章先提一下服务接口环境下实现的概念步骤。

这很大程度上与服务调用中，发出通知的源对象难于管理需要通知的各个对象有关，采用下面的直接方式，源对象至少需要了解如下内容：

- 每个目标对象的地址（对于 Web Service 而言，大部分是 URL）。
- 如何与每个目标对象绑定，双方的通过什么协议、接口关联在一起。
- ……

服务环境下的通知关系如图 22-4 所示。

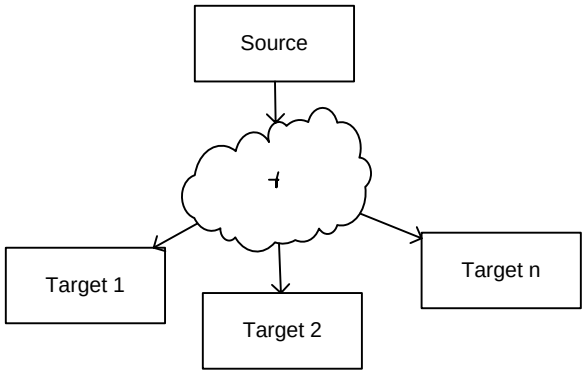


图 22-4 服务环境下的通知关系示意图

由于服务源服务接口需要服务的对象还不确定的内容非常多，也可能非常杂，如果让它实现服务的同时又去管理这些“杂事”，对其运行有不利的影响，而且承担的职责太多，同时会产生过多的对外依赖关系。虽然通过 WS-MEX（Web Service MetadataExchange）可以让源与每个目标对象间完成对消息交换方式的协商，但还是免不了 1:N 的依赖关系。还有一个问题，即我们设计的服务接口往往不像之前示例中那么简单，需要通知的事情（正常情况、异常情况、延期超时、追加通知，同一个业务领域下的不同主体……）也比较多，相应的目标方关注的内容也不一样（例如，虽然银行提供了催缴电话费的短信服务，但这些服务不能每次都“群”，还是要有的放矢）。

所以，很有必要安排一个观察者负责关注源的动向，另外还需要增加一个持久化机制，把每个目标方关注的内容和源实际可能通知的事件类别进行登记并匹配，这样即便有各种类型、各种适用个性化的通知，也可以“按需”完成通知。在 WS-Eventing 中规定了几个

概念。

- Event Source：实际发生事件的源。
- Event Message：通知本身作为事件的消息，在松散的服务接口间传递。
- Event Sink：实际接受 Event Message 的目标方，类似 WCF 中的概念体系，它也是用 ABC（Address、Binding、Contract）描述的（ABC 的组合常被称为 ServiceEndpoint）。

下面我们看一下一般设计服务接口层次 WS-Eventing 系统的结构，如图 22-5 所示。

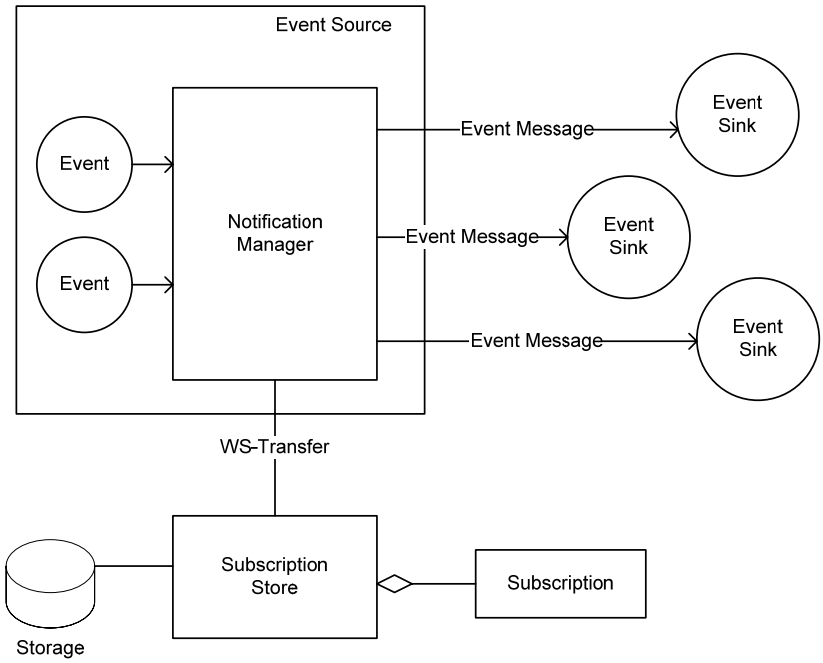


图 22-5 WS-Eventing 的系统结构

这里 SubscriptionStore 是个抽象的概念预订信息访问对象，它保存的信息包括哪些 EventSink 会对何种事件感兴趣，需要接收该事件，而通知管理机制在收到新的通知事件的时候，往往采用 WS-Transfer 方式提取、查找相关信息。至于具体用什么技术实现 WS-Transfer，则没有明确要求，比如用数据访问、用 XPath 查找一个 XML 文件都可以。

该方式的时序如图 22-6 所示。

考虑到 NotificationManager 主要负责有关运行过程中的执行，为了和 SubscriptionStore 打交道，之前还需要有个对象管理对预定信息的收集、登记、撤销，所以我们在上面系统结构上增加一个 SubscriptionManager 的对象，它和 NotificaionManager 没有直接的交互，两者仅基于 SubscriptionStore 实现各自独立的处理，如图 22-7 所示。

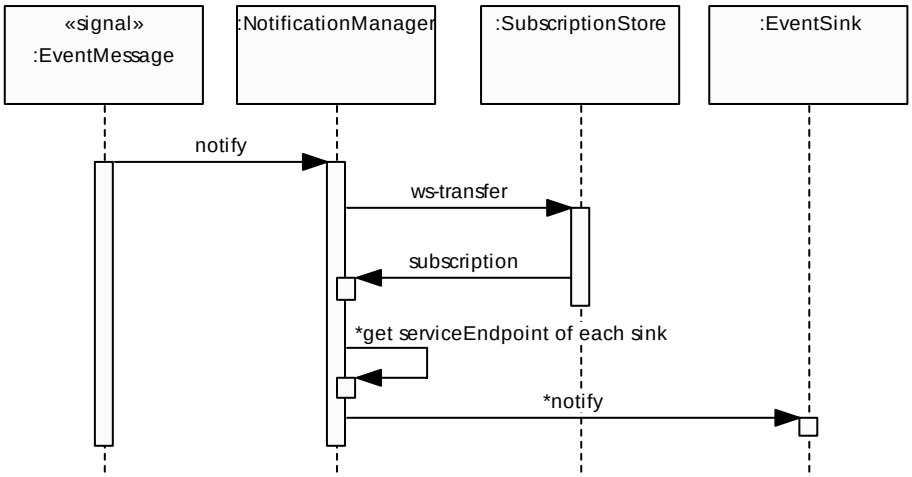


图 22-6 通知时序

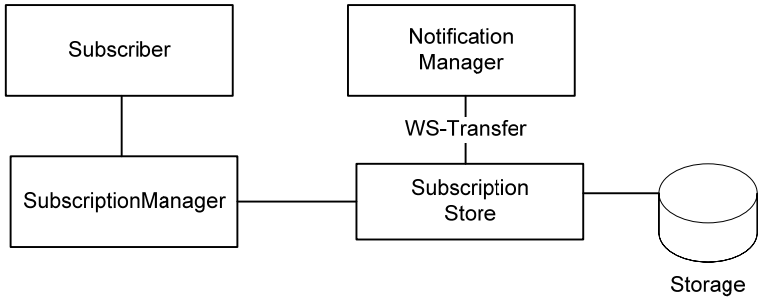


图 22-7 增加了预定情况持久化之后的结构

这时候实际执行之前的注册过程就变成如图 22-8 所示。

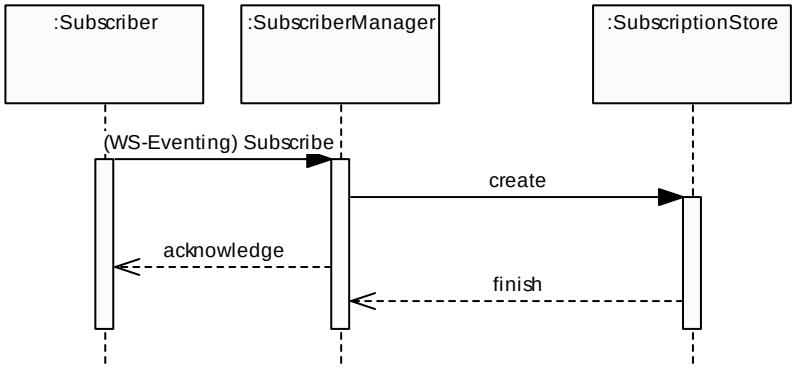


图 22-8 增加了预定信息持久化机制之后的执行时序

这里有意把 SubscriberManager 的回复写在下一个操作之前，其意在说明 Subscriber 的预定过程可以是异步的，也可以同步执行。

22.6 小结

本章介绍了观察者模式，由于它可以比较有效地降低调用关系中 1:N 的依赖，同时对无法预知数量的观察者提供支持，所以无论在.NET Framework 还是在项目实施中都被普遍使用。

但与此同时，观察者模式也会产生一些不太好的后果，例如：

- 降低了性能：不使用该模式的时候对象间是直接引用的调用过程，而且引用关系在编译阶段就决定的。
- 内存泄漏：即便每个主题对象上的所有观察者都已经失效了，但它们没有调用 Detach 方法，因此无论是观察者本身还是主题对象间都可能因为相互的引用关系无法被 GC 回收。
- 测试和调试更加困难：采用观察者模式会令调用关系的依赖不如之前的直接调用明显，因此在测试或调试的时候我们需要时刻注意每一个 Attach 和 Detach 过程，而且还要注意 Notify 当前遍历的位置，否则难于跟踪并找到当前实际执行的类型。

除非特殊需要，很多时候.NET 环境下开发并非必须定义独立的 IObserver 接口，我们可以直接采用.NET 的事件机制完成。

第30章

出版—预订模式

30.1 说明

30.2 生活中无处不在的“预订”

30.3 示例

30.4 小结

30.1 说明

生活中我们往往会“预订”很多事情，预订生日蛋糕和鲜花，预订机票、火车票、船票和比赛门票……这些事情其实都有同一个规律可循：

- 首先，我们想到哪些内容是我们关心或需要的，而且很多时候是在对后续情况预期基础上的。
- 我们找到一些单位（人/机构/网站），看看它们可以提供哪些资料（消息/实物）的预订。
- 从这些资料列表中我们选择需要的内容，并告知它们提供给谁、以什么方式（上门送/发短信告知/还是我们自己上门去取）、需要的紧急程度（特快专递/平邮）。
- 对于很多商用情景，我们还要告知在多长时间内、配货情况出现问题的补救或异常决策。
- 然后，支付相应的费用后，我们就可以等着“预订”的资料。

当然，即便您不订票、不订花、不订报刊杂志，一个最普遍的预订情况也会出现在您去饭店吃饭的时候。

上面的过程相当于把资料提供方和需求方分开：供应方可以按照自己的节奏处理事情，只不过在某些关键的位置知会一声；而需求方也不用一天到晚“盯着”供应方，等现成的即可。

您会发现，所谓“出版—预订”模式（有时候也被称为“预订模式”、“订阅模式”）要解决的其实就是如何异步方式向多个对象同步其属性或状态的问题，这个与之前的观察者模式没有区别。确实，按照设计模式的划分，它们只是同一模式的两个名称而已，由于在产品级软件、服务中大部分情况按照“出版—预订”命名，所以本章是在观察者模式之后的进一步工程化扩展，区别于观察者模式一章，这里主要介绍如何设计“出版者”（Publisher，或称之为“发布者”）、“预订者”（Subscriber），以及它们如何管理众多预订任务的问题，另外针对处理性能的考虑，还会增加一些第三方对象使得结构更符合高负载环境的使用。

30.2 生活中无处不在的“预订”

有关异步预订的原理我们在观察者模式中已经作了介绍。下面讨论一下当这个结构放在分布式应用环境中，同时作为项目一个集中的对外信息服务的时候，还需要增加什么。

30.2.1 面向单一主题的本地观察者模式

这一模式的组成结构如图 30-1 所示。

由图不难发现，该状态下我们有个假设——本地运行。而事实上发布消息的往往在服务器一方，而预订的可能是另一个服务或最终用户，它们之间很难保证这种直接的二进制对象

依赖，即便我们为它们赋予远程处理能力，那也会令 ISubject 和 IObserver 实体对象过于复杂、职责过重。

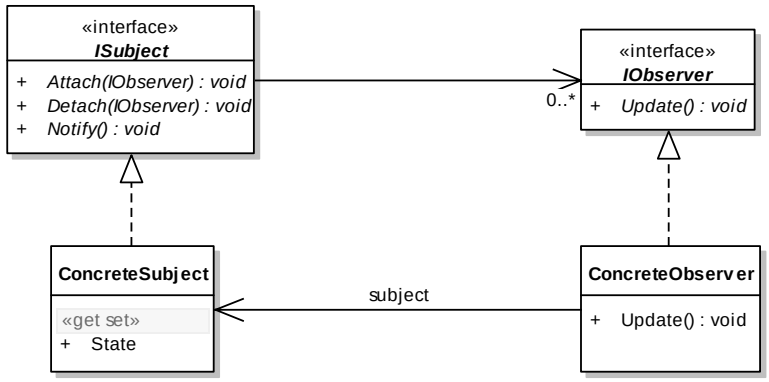


图 30-1 面向单一主题的本地观察者模式

30.2.2 增加 Proxy 实现面向单一主题的分布式观察者模式

此模式的组成结构如图 30-2 所示。

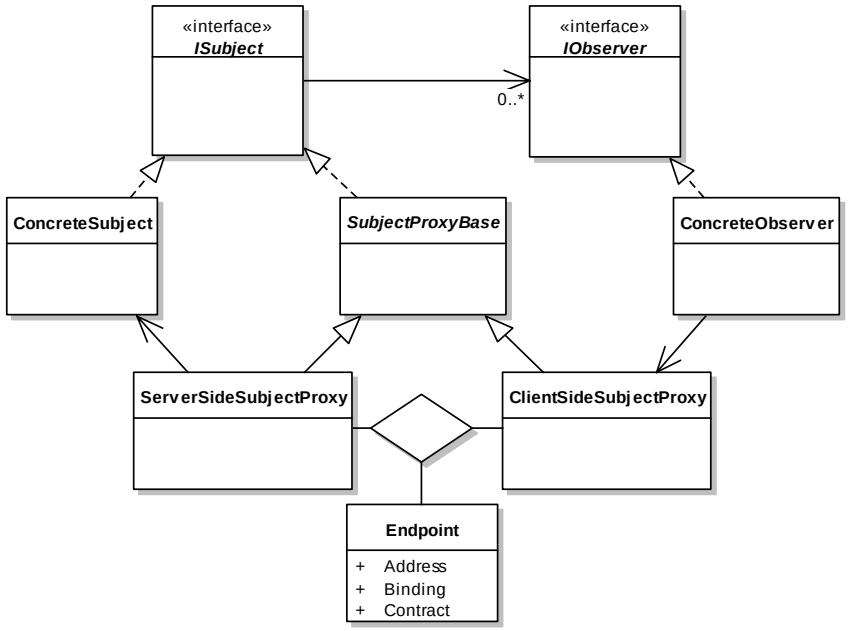


图 30-2 增加代理类型后的面向单一主题的远程观察者模式

之前我们提到，为了比较“透明”地解决有关远程访问的问题可以考虑在边界上设置代理对象，并在代理模式一章用 WCF 实现了一个远程代理，这里我们可以用类似的方法解决。

此模式区别于本地方式，增加了服务端和客户端的 SubjectProxy 类型。它们之间根据 EndPoint 的 ABC 关联在一起，由于开始的预订过程及后面的通知发布过程需要双向的通信过程，因此服务端和客户端的 SubjectProxy 可能会产生双向的调用关系。

30.2.3 用出版者集中管理预订

上面的示例虽然对于发布一个主题内容比较合适，但在项目中对外信息发布、数据预订下发等工作一般都是相对集中在某个功能模块内部，即该模块往往需要同时提供多个主题的管理。为了表示区别，这里我们按照“出版—预订”的称呼，把一系列服务能够提供的主题内容集中在一个统一的出版者（Publisher）上。按照单一职责原则还要为它增加一个出版事件列表（可发布主题的列表），把关心各个主题的对象统一称为预订者（Subscriber）。

这里有关出版者/预订者的设计上有个取舍的问题：设计一系列面向不同主体的预订者还是设计一个通用的预订者。前者您可以采用之前观察者模式的方式，只不过在出版者增加一个具有分类的观察者列表；后者有个前提就是所有的主题传递给预订者采用抽象兼容的方式。预订者本身不负责复杂的信息拆解、验证等工作，它的作用就是单纯的事前预订、事中接收，至于收到后采用 CoR 方式还是管道方式处理与预订者无关。采用后者还有一个意义，就是对于预订信息的聚合作用，因为它可以不仅仅关注于一个出版者。它可以把 Google 的地图、eBey 的支付和自己的洗车服务结合在一起，与之相连的客户程序只从这一个统一的入口操作相关服务。至于以后基于商务上的考虑，把 Google 的地图换成其他地图服务只须在预订者及其处理管道进行一些调整即可。后者的静态结构如图 30-3 所示。

出版者也有类似的问题，不过在面向服务的环境下，不同服务往往通过 url + 统一的 WSDL 来区分，比如一个门户网站：它可能通过 *http://www.<sitename>.com/publisher/auto* 发布汽车信息，而 *http://www.<sitename>.com/publisher/stock* 发布股票信息。

为了与一般“出版—预订”介绍材料一致，这里主题我们采用英文 Article 而非 Subject。

图 30-3 中的相关说明如下：

- IPublisher：出版者，它定义了出版者需要具有的基本功能，包括提供发布信息列表、接收和撤销预订者的预订等功能。
- ISubscriber：预订者，根据（1 个或多个）出版者发布的内容，预订相关信息。
- NotificationGenerator：负责在 IPublisher 生成预订事件的时候，比照相关预订者，生成通知的对象。
- Distributor：获得通知事件之后，根据事件中说明的预订者 Endpoint，把通知下发给它（/它们）。

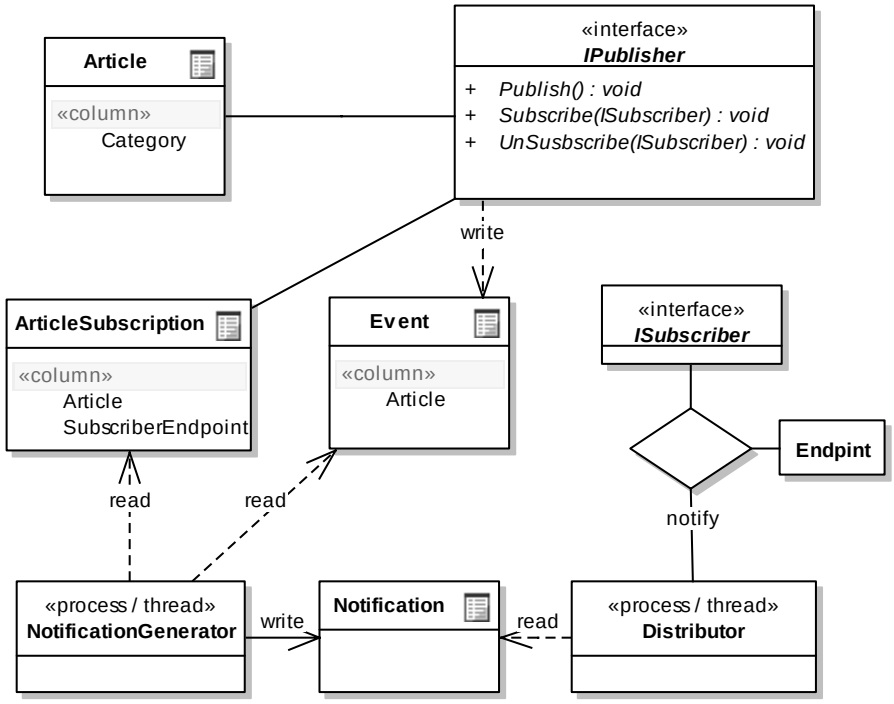


图 30-3 增加了持久层机制后的“出版—预订”体系

之所以增加这些内容，主要是实际项目中我们的出版者、预订者往往并不像我们在观察者模式介绍的那样，大家都在一个进程内部执行，而需要发布的信息在流转过程中一般也都需要“落地”（即持久化到类似关系数据库、报文等介质中），因此需要增加一些持久化对象来保存中间结果，包括：

- Article：出版者提供的可供预订的信息列表。
- ArticleSubscription：保存 Article 与预订者对应关系的列表。
- Event：出版者在运行过程中，在特定执行步骤下抛出的预订事件。
- Notification：根据预订规则，确认实际预订事件中实际有预订载体的内容，并附加相关预订者 Endpoint 信息的通知列表。

由于实际项目中，出版者往往又是随着主要的生产数据库的数据更新对外提供预订事件的，因此出于尽量减轻其额外负载的考虑，又增加了其他对象协助进行通知的生成和发送。

前期的预订内容发布和预订过程如图 30-4 所示。

预订事件生成并根据之前的预订者定义的内容生成通知，其发送过程时序如图 30-5 所示。

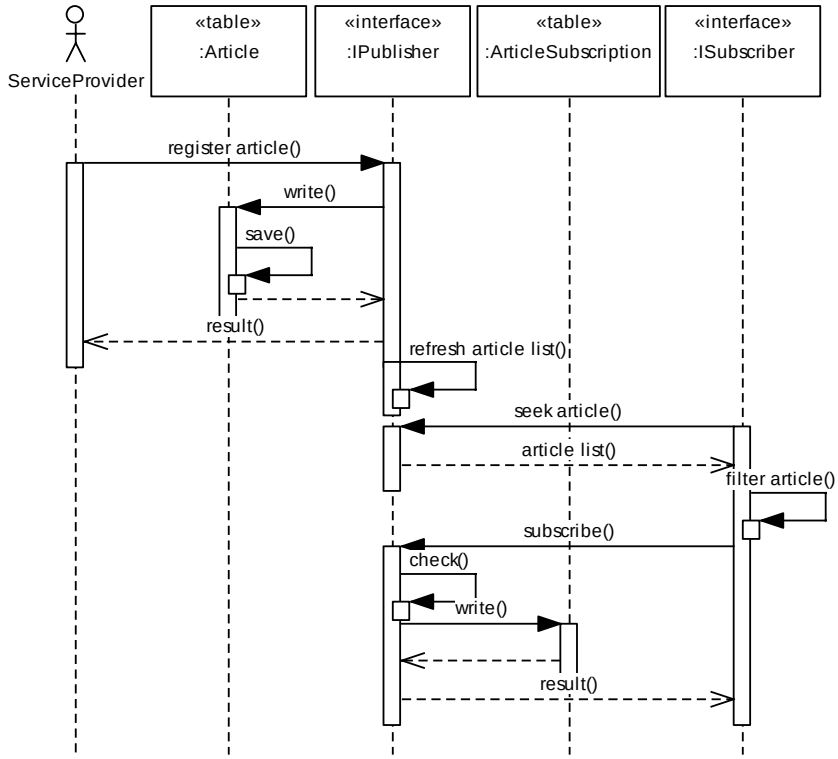


图 30-4 通知生成过程时序

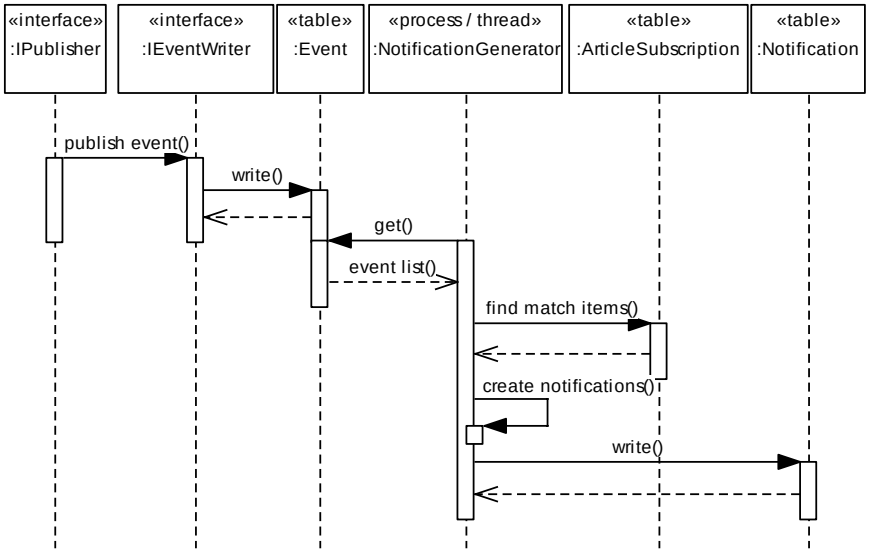


图 30-5 通知发送过程时序

然后，Distributor 根据每条通知的内容及其中预订者的 Endpoint 信息把通知发送给它。

30.2.4 面向物理环境设计更多出版预订模式

上面的设计并没有过多考虑有关负载、网络区域划分等实际的项目的问题。面向具体物理环境我们还有很多“出版—预订”部署选择：

1. 双向（或多节点双向）同步（如图 30-6 所示）。

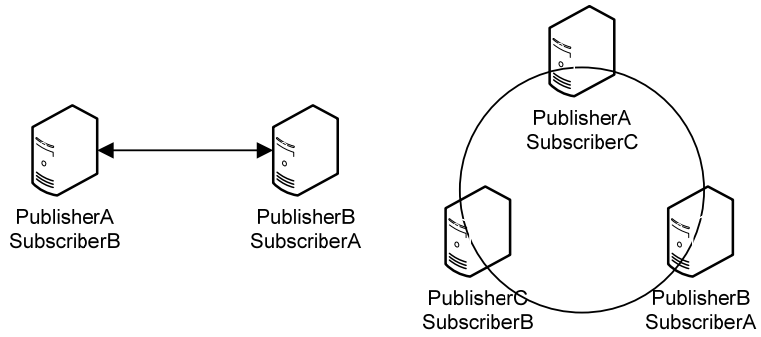


图 30-6 同步或多节点双向同步

这里设计上每个节点（或者是内部对象）同时具有 IPulisher 和 ISubscriber 的功能，采用这种方式主要是面向信息同步，尤其对于异地容灾中心的设计更加必要。

2. 二级（或多级）发布（如图 30-7 所示）。

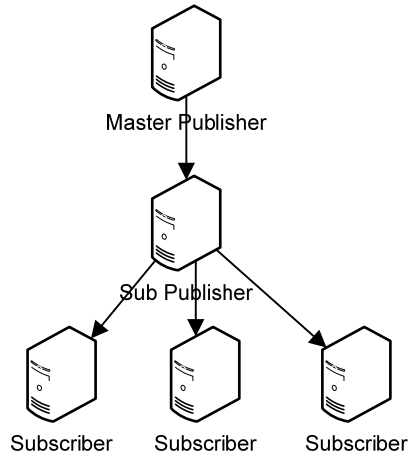


图 30-7 多级发布

采用这种方式的原因也很多，例如：

- 信息系统物理上分成不同的大区域，每个区域内部都有独立同时又有数据同步要求的各类应用。
- 主出版者自身对外负载支撑能力有限，需要子出版者接力作二次发布。
- 出版者的内容针对性比较强，比如：全国的销售情况都会及时发布到每个系统，但对于区域数据中心而言，它只关心本区域的数据，因此子出版者除了二次发布之外，还起到信息筛选的作用。
- 主出版者的网络速度相对较慢，而预订对象相对集中在某一个网速较高的区域。
- 信息的预订方虽然很多，但是从主出版者部分提取数据租用网络带宽比较贵，而次级出版者部分相对廉价。

3. 集中式预订者（如图 30-8 所示）。

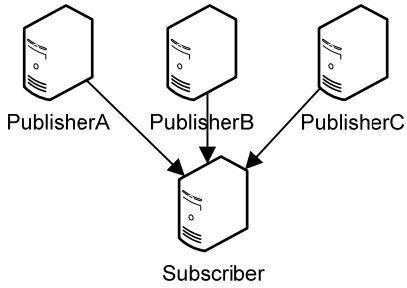


图 30-8 集中式预订

另外，就是上面提到的集中式预订者，它主要用于数据服务比较分散，但新的应用相对集中的情况，集中式预订者的主要职能就是汇聚。

30.3 示例

有关远程 Proxy 的示例，我们之前在代理模式一章已经作了介绍，这里作为示例我们按照“用出版者管理预订”的介绍模拟一个单进程的示例。为了简化，其中有关持久化对象我们采用内存集合类型代替。

30.3.1 数据实体模型部分

示例代码如下：

C#

```
/// 便于按照 Key 从持久层检索对象的抽象
public interface IObjectWithKey
{
```

```
    string Key { get;}
}

/// 预订发布内容对象
/// 其内容可以是各种通用的数据类型，例如:
/// <list type="bullet">
///     <item>string</item>
///     <item>DataSet</item>
///     <item>Stream</item>
///     <item>XmlDocument</item>
/// </list>
/// 这里为了简化，采用了 string
public class Article : IObjectWithKey
{
    private string category;
    public string Category
    {
        get { return category; }
        set { category = value; }
    }

    private string content;
    public string Content
    {
        get{return content;}
        set{content = value;}
    }

    public Article(string category, string content)
    {
        this.category = category;
        this.content = content;
    }

    #region IObjectWithKey Members
    public virtual string Key
    {
        get { return category; }
    }
    #endregion
}


/// Publishder 抛出的预订事件
public class Event : IObjectWithKey
{
    public string id = Guid.NewGuid().ToString();
    public string ID
    {
        get { return id; }
    }

    private Article article;
    public Article Article
    {
        get { return article; }
    }

    public Event(Article article)
```

```
        {
            this.article = article;
        }

        #region IObjectWithKey Members
        public virtual string Key
        {
            get { return ID; }
        }
        #endregion
    }


    /// 订阅情况
    public class ArticleSubscription : IObjectWithKey
    {
        private Article article;
        public Article Article
        {
            get { return article; }
        }

        private ISubscriber subscriber;
        public ISubscriber Subscriber
        {
            get { return subscriber; }
        }

        public ArticleSubscription(Article article, ISubscriber subscriber)
        {
            this.article = article;
            this.subscriber = subscriber;
        }

        #region IObjectWithKey Members
        public virtual string Key
        {
            get
            {
                return ((IObjectWithKey)article).Key +
                    subscriber.GetHashCode().ToString();
            }
        }
        #endregion
    }

    /// 通知
    /// <remarks>
    /// 由于通知是事件发生后，根据预订条件筛选出来的内容，
    /// 这里为了简化示例，采用直接继承的方式
    /// </remarks>
    public class Notification : IObjectWithKey
    {
        private Event e;
        public Event Event
        {
            get { return e; }
        }
```

```
    private ISubscriber subscriber;
    public ISubscriber Subscriber
    {
        get { return subscriber; }
    }

    public Notification(Event e, ISubscriber subscriber)
    {
        this.e = e;
        this.subscriber = subscriber;
    }

    #region IObjectWithKey Members
    public string Key
    {
        get { return e.ID + subscriber.GetHashCode().ToString(); }
    }
    #endregion
}
```

30.3.2 业务实体模型部分

示例代码如下：

C#

```
/// 抽象 Subscriber 接口
public interface ISubscriber
{
    /// 接收预订
    void Enqueue(Article article);

    /// 检查是否有某个分类的新预订结果
    string Peek(string category);

    /// 获取某个分类新的预订结果
    string Dequeue(string category);
}

/// 抽象 Subscriber 对象
public abstract class SubscriberBase : ISubscriber
{
    /// 按分类整理的接收队列
    /// <remarks>
    /// 实际项目中这个队列可能就是个内存缓冲，也可能是具体的持久化机制
    /// </remarks>
    protected IDictionary<string, Queue<string>> queue
        = new Dictionary<string, Queue<string>>();

    public virtual void Enqueue(Article article)
    {
        if (article == null) throw new ArgumentNullException("article");
        string category = article.Category;
        if (!queue.ContainsKey(category))
            queue.Add(category, new Queue<string>());
        queue[category].Enqueue(article.Content);
    }
```

```
        public virtual string Peek(string category)
        {
            if (!queue.ContainsKey(category))
                return null;
            if (queue[category].Count == 0)
                return null;
            return queue[category].Peek();
        }

        public virtual string Dequeue(string category)
        {
            if (!queue.ContainsKey(category))
                return null;
            if (queue[category].Count == 0)
                return null;
            return queue[category].Dequeue();
        }
    }

    /// 抽象 Publisher 接口
    public interface IPublisher
    {
        /// 增加预订项
        void Subscribe(Article article, ISubscriber subscriber);

        /// 撤销预订项
        void Unsubscribe(Article article, ISubscriber subscriber);

        /// 获取所有可发布的内容列表
        IEnumerator<Article> Articles { get;}

        /// 发布预订事件
        void Publish(Article article);
    }

    public abstract class PublisherBase : IPublisher
    {
        protected IKeyedObjectStore<Article> articleStore;
        protected IKeyedObjectStore<ArticleSubscription> subscriptionStore;
        protected IKeyedObjectStore<Event> eventStore;

        public virtual void Subscribe(Article article, ISubscriber subscriber)
        {
            if (article == null) throw new ArgumentNullException("article");
            if (subscriber == null) throw new ArgumentNullException("subscriber");
            ArticleSubscription subscription = new ArticleSubscription(article,
              subscriber);
            string key = ((IObjectWithKey)subscription).Key;
            if (subscriptionStore.Select(key) == null)
                subscriptionStore.Save(subscription);
        }

        public virtual void Unsubscribe(Article article, ISubscriber subscriber)
        {
            if (article == null) throw new ArgumentNullException("article");
            if (subscriber == null) throw new ArgumentNullException("subscriber");
            ArticleSubscription subscription = new ArticleSubscription(article,
              subscriber);
```

```
            subscriptionStore.Remove(((IObjectWithKey)subscription).Key);
        }

        public virtual IEnumerator<Article> Articles
        {
            get
            {
                foreach (Article article in articleStore)
                    yield return article;
            }
        }

        public virtual void Publish(Article article)
        {
            if (article == null) throw new ArgumentNullException("article");
            Event e = new Event(article);
            eventStore.Save(e);
        }
    }

    /// 抽象的持久层对象
    public interface IKeyedObjectStore<T>  where T : class, IObjectWithKey
    {
        /// 保存
        void Save(T target);

        /// 按照 Key 提取
        T Select(string key);

        /// 删除
        void Remove(string key);

        /// 遍历
        IEnumerator GetEnumerator();
    }
```

30.3.3 具体实体对象部分

由于示例中 NotificationGenerator 和 Distributor 与其他部分在一个进程中执行，为了体现它们的动态处理能力，示例在预订事件列表和通知事件列表上增加了更新事件。

C#

```
/// 示例用的测试用内存持久化机制
public class KeyedObjectStore<T> : IKeyedObjectStore<T>
    where T : class, IObjectWithKey
{
    protected IDictionary<string, T> data = new Dictionary<string, T>();

    public virtual void Save(T target)
    {
        if (target == null) throw new ArgumentNullException("target");
        data.Add(target.Key, target);
    }

    public virtual T Select(string key)
```

```
        {
            if (string.IsNullOrEmpty(key)) throw new
              ArgumentNullException("key");
            T result;
            if (data.TryGetValue(key, out result))
                return result;
            else
                return null;
        }

        public virtual void Remove(string key)
        {
            data.Remove(key);
        }

        public IEnumerator GetEnumerator()
        {
            foreach (T value in data.Values)
                yield return value;
        }
    }

    /// 示例用的预订事件持久化
    public class ExtEventArgs : EventArgs
    {
        private Event e;
        public Event Event
        {
            get { return e; }
        }

        public ExtEventArgs(Event e)
        {
            this.e = e;
        }
    }

    public class EventStore : KeyedObjectStore<Event>
    {
        public EventHandler<ExtEventArgs> NewEventOccured;

        public override void Save(Event target)
        {
            base.Save(target);
            if (NewEventOccured != null)
                NewEventOccured(this, new ExtEventArgs(target));
        }
    }

    /// 示例用的通知事件持久化
    public class NotificationEventArgs : EventArgs
    {
        private Notification notification;
        public Notification Notification
        {
            get { return notification; }
        }

        public NotificationEventArgs(Notification notification)
```

```
    {
        this.notification = notification;
    }
}
public class NotificationStore : KeyedObjectStore<Notification>
{
    public event EventHandler<NotificationEventArgs> NewNotificationOccured;

    public override void Save(Notification target)
    {
        base.Save(target);
        if (NewNotificationOccured != null)
            NewNotificationOccured(this, new NotificationEventArgs(target));
    }
}

/// 示例用的发布信息持久化
public class ArticleStore : KeyedObjectStore<Article>
{
}

/// 示例用的预订情况持久化
public class ArticleSubscriptionStore :
KeyedObjectStore<ArticleSubscription>
{
}

/// 示例用的出版者
public class Publisher : PublisherBase
{
    public ArticleStore ArticleStore
    {
        set { articleStore = value; }
    }

    public ArticleSubscriptionStore ArticleSubscriptionStore
    {
        set { subscriptionStore = value; }
    }

    public EventStore EventStore
    {
        set { eventStore = value; }
    }
}

/// 示例用的预订者
public class Subscriber : SubscriberBase
{
    public IDictionary<string, Queue<string>> Queue
    {
        get { return queue; }
    }
}

/// 示例用的通知生成器
public class NotificationGenerator
{
    private EventStore eventStore;
```

```
    private NotificationStore notificationStore;
    private ArticleSubscriptionStore articleSubscriptionStore;

    public NotificationGenerator(
        EventStore eventStore,
        NotificationStore notificationStore,
        ArticleSubscriptionStore articleSubscriptionStore)
    {
        this.eventStore = eventStore;
        this.notificationStore = notificationStore;
        this.articleSubscriptionStore = articleSubscriptionStore;

        // 接收新的预订事件
        eventStore.NewEventOccured += OnNewEventOccured;
    }

    /// 筛选并生成通知
    public void OnNewEventOccured(object send, ExtEventArgs args)
    {
        Event e = args.Event;
        string articleKey = e.Article.Key;
        foreach (ArticleSubscription articleSubscription in
           articleSubscriptionStore)
        {
            string subscriptionArticleKey =
               ((IObjectWithKey)(articleSubscription.Article)).Key;
            if(string.Equals(articleKey, subscriptionArticleKey))
            {
                Notification notification =
                    new Notification(e, articleSubscription.Subscriber);
                notificationStore.Save(notification);
            }
        }
    }
}

/// 示例用的通知发送对象
public class Distributor
{
    private NotificationStore notificationStore;
    public Distributor(NotificationStore notificationStore)
    {
        this.notificationStore = notificationStore;
        // 发送新的通知情况
        notificationStore.NewNotificationOccured +=
            OnNewNotificationOccured;
    }

    /// 发送新的通知
    public void OnNewNotificationOccured(object send,
       NotificationEventArgs args)
    {
        Article article = args.Notification.Event.Article;
        ISubscriber subscriber = args.Notification.Subscriber;
        subscriber.Enqueue(article);
    }
}
```

30.3.4 单元测试

示例代码如下：

C#

```
#region Private Fields
private ArticleStore articleStore;
private Article articleX;
private Article articleY;
private Article articleZ;
private ArticleSubscriptionStore articleSubscriptionStore;
private EventStore eventStore;
private NotificationStore notificationStore;
private Publisher publisher;
private Subscriber subscriberA;
private Subscriber subscriberB;
private NotificationGenerator generator;
private Distributor distributor;
#endregion

#region PreProcess
#region 初始化持久层，增加新的预订事件
void InitPersistence()
{
    articleStore = new ArticleStore();
    articleX = new Article("X", string.Empty);
    articleY = new Article("Y", string.Empty);
    articleZ = new Article("Z", string.Empty);
    articleStore.Save(articleX);
    articleStore.Save(articleY);
    articleStore.Save(articleZ);
    articleSubscriptionStore = new ArticleSubscriptionStore();
    eventStore = new EventStore();
    notificationStore = new NotificationStore();
}
#endregion

#region 组装出版—预订机制
void AssemblyPubSub()
{
    // 构造出版者
    publisher = new Publisher();
    publisher.ArticleStore = articleStore;
    publisher.ArticleSubscriptionStore = articleSubscriptionStore;
    publisher.EventStore = eventStore;
    // 构造预订者
    subscriberA = new Subscriber();
    subscriberB = new Subscriber();
    // 通知生成和分发
    generator = new NotificationGenerator(eventStore,
        notificationStore, articleSubscriptionStore);
    distributor = new Distributor(notificationStore);
    // 预订新的事件
    publisher.Subscribe(articleX, subscriberA);
    publisher.Subscribe(articleY, subscriberA);
    publisher.Subscribe(articleY, subscriberB);
```

```
    publisher.Subscribe(articleZ, subscriberB);
}
#endregion

#region 执行发布
void PublishEvent()
{
    publisher.Publish(new Article("X", "1"));
    publisher.Publish(new Article("X", "2"));
    publisher.Publish(new Article("Y", "3"));
    publisher.Publish(new Article("Z", "4"));
}
#endregion
#endregion

[TestMethod]
public void Test()
{
    InitPersistence();
    AssemblyPubSub();
    PublishEvent();

    // 验证出版—预订体系执行效果
    Assert.AreEqual<int>(2, subscriberA.Queue.Count);
    Assert.AreEqual<string>(subscriberA.Dequeue("X"), "1");
    Assert.AreEqual<string>(subscriberA.Dequeue("X"), "2");
    Assert.AreEqual<string>(subscriberA.Dequeue("Y"), "3");

    Assert.AreEqual<int>(2, subscriberB.Queue.Count);
    Assert.AreEqual<string>(subscriberB.Dequeue("Y"), "3");
    Assert.AreEqual<string>(subscriberB.Dequeue("Z"), "4");
}
```

分析一下上面示例的内容：

- 出版者可以提供多个发布内容，预订者可以预订不同内容并及时接受通知。
- 虽然没有正式系统那样设计主动方式执行的 NotificationGenerator 和 Distributor，但借助.NET 事件机制示例中的对象同样可以在持久化信息更新的时候提取到出版者最新发布的内容，并实现“事件发布→事件持久化→通知生成→通知持久化→通知发布至预订方”的过程。
- 借助 LINQ、XQuery、ADO.NET 等措施，实际项目中可以进一步扩展每一个持久化机制，使其变成一个实际可执行的主动消息发布机制。

30.4 小结

通过增加新的执行对象和持久化机制，本章试图在经典观察者模式基础上实现一个更贴近物理部署环境的“出版—预订”机制，它们包括：

- 可发布信息持久化。
- 预订者预订情况持久化。

- 通知信息持久化。
- 发布信息持久化。
- 用于将发布信息转换为通知的 NotificationGenerator。
- 用于将通知发布到预订者的 Distributor。
- 此外，对于更大规模的信息发布环境，还可以增加多级出版者、集中式预订者等对象。

随着 Ent2 和 Web2 类型应用的普及，信息服务更多从服务端“推”到客户端，因此“出版—预订”模式在项目中的使用会更加普遍。

第37章

Web服务事件监控器模式

37.1 说明

37.2 如何为普通 Web Service 封装事件机制

37.3 示例

37.4 小结

37.1 说明

我们设计 Web Service 的目的总是希望它对外提供某些服务。但限于设计能力和运行环境的关系，我们不能假设它们都提供了主动的信息服务，即类似我们前面提到的“出版—预订”方式那样按照我们关注的主题和获取方式主动向我们提供信息。绝大多数的 Web Service 应用还是要我们自己从它那里获取信息的。两者的区别如图 37-1 所示。

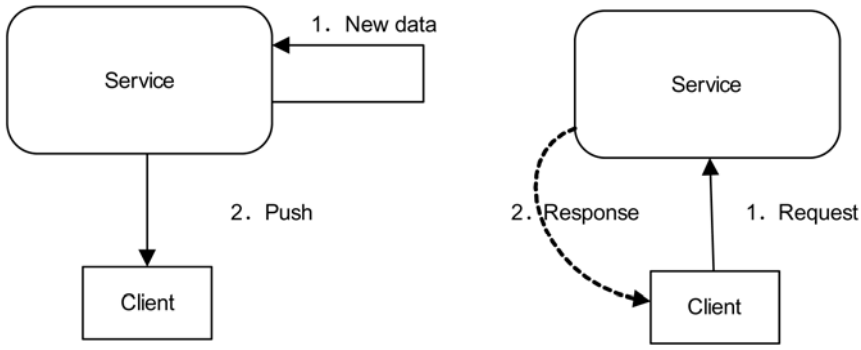


图 37-1 主动方式和被动方式的服务提供过程

对于后面这种情况如果让每个客户程序都执行一个循环任务，从某种程度上而言会破坏程序自身的执行过程，而且对于服务端而言也是个不小的压力。因此，基于我们之前积累的经验，是否可以在服务外围增加一个对象，令客户程序继续享受到预订方式的信息获取，同时可以根据服务端相关主题的变化，准实时地更新相关客户端呢？解决这类问题的一个方法就是本章要讨论的事件监控器模式。

从调用需求看，事件监控器起到承上启下的作用。一方面它作为一个主动客户端机制，准实时地访问服务端；另一方面它又作为服务端通过“出版—预订”方式向下游客户端发送数据。另外，这个准实时到底要“准”到什么程度，要视服务端承载能力和客户端相应要求而定，因此最好有个可以配置的 Interval（时间间隔）参数。至于事件监控器怎么知道哪些信息变化了呢？服务端一般不会保存它们，下游客户程序只需要获得更新的通知，因此这个确认到底是否变化的工作就放在了中间——事件监控器中来完成。

采用这种外围包装方式有什么好处呢？

- 由于事实上支持 WS-Event 和 WS-Notification 的协议的 Web Service 应用非常少，即便服务端支持，对应的客户端也有限，所以不妨使用这种手工“捏”出来的事件机制，便于下游应用的使用。
- 给客户端应用带来更大的自由度，下游客户程序可以“自选”需要的内容，借助事件监控器定制自己需要访问的内容。

- 由于 SOA 环境下很多 Web Service 都是外部产品或是其他开发团队构造的，如果修改它们的内容，让它们主动地提供信息服务比较困难。同时，直接集成它们比较困难，不如采用外部改造的办法。
- 便于支持不同隔离区域的情况，企业环境中为了治理的需要，可能需要把某些服务部署在相对独立的区域中，对其他区域服务的时候需要限制其访问会话数量，防止因为这些“副业”影响“主业”的情况，因此通过在边界提供一个可调配的事件监控器也利于掌控对关键服务的使用情况。

37.2 如何为普通 Web Service 封装事件机制

基于上面的讨论，事件监控器模式的静态结构如图 37-2 所示。

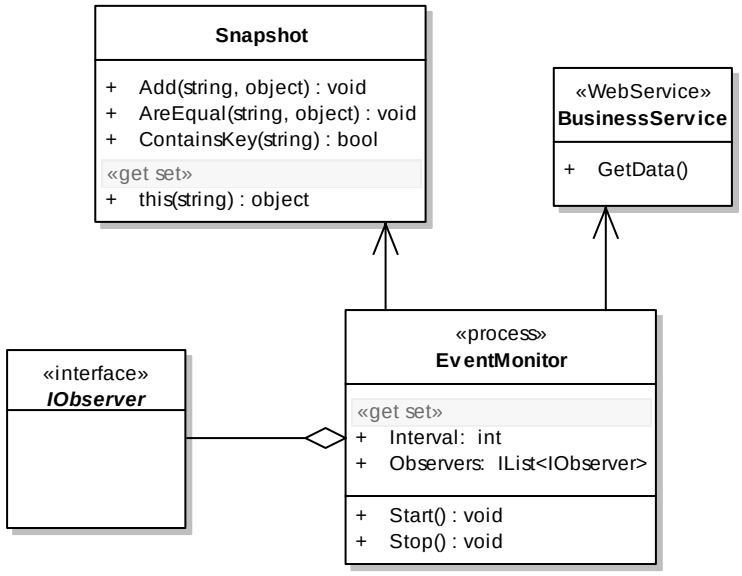


图 37-2 Web 服务事件监控器模式的静态结构

其中每个对象的职责如下：

- IObserver：类似之前观察者模式中对应角色的作用，替客户程序“盯着”EventMonitor 通知需要更新内容的对象。
- EventMonitor：主动访问具体业务服务。
- Snapshot：快照，主要作用是存留。存留之前保存的各个信息，用于在 EventMonitor 获取新信息后比照是否进行了修改。
- BusinessService：则是具体加工业务信息的一端，它虽然会对外提供数据查询接口，但

本身并不会主动提交。

它们的执行时序如图 37-3 所示。

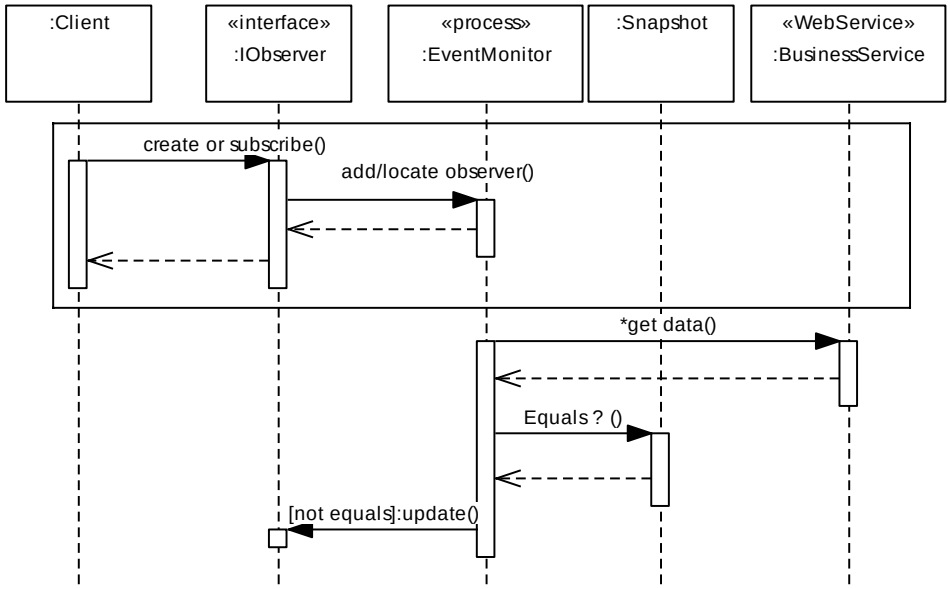


图 37-3 Web 服务事件监控器的执行时序

那么使用上面的方式有什么优势呢？

- 下游客户程序只需要等着现成的更新通知。
- 一个事件监控器可以为一系列观察者服务，允许它们预订相同或各异的主题。
- 一个事件监控器可以负责向多个服务“打听”情况。
- 相对而言，如果关心同一主题的下游客户程序较多，借助事件监控器可以减轻实际服务端“汹涌”的 Web Service 轮循调用。
- 随着多核多芯服务器的普及，可以把事件监控器设计为具有并发轮循、并发通知的体系。

但是这么做也有一个最大的隐患——丢数据，毕竟这种准实时并非真的实时，尤其对于一些数据突发性明显的情景。BusinessService 可能平时闲着，数据都是晚上批量导入甚至是直接 BCV 分离出来的。这样白天事件监控器基本上都是“白忙活”，晚上导入数据的过程又“目不暇接”，造成数据变更情况丢失，这可能会影响到下游客户程序的处理逻辑。

BCV：business continuance volume，EMC 等厂商提供的快速磁盘快照技术。短时间内在一个分离的介质复制出另一个磁盘信息的技术，如果事件监控器关注的是 BCV 的目标磁盘，很可能瞬时获得一个非常大的更新结果，导致整个事件监控器体系过载。

37.3 示例

下面作为示例，我们结合 WCF 实现一个简化结构：

- BusinessService 就是个 WCF 的服务程序，内部通过一个 Timer 定时修改其一系列状态信息。
- EventMonitor 的轮循也是一个 Timer 定期触发对 WCF 客户端访问。
- 下游客户程序就是由事件机制作为 Observer 通知的同一进程的一些对象。

示例代码如下：

C#

```
/// 业务服务接口
 [ServiceContract]
public interface  IMessage
{
    [OperationContract]
    string GetID();

    [OperationContract]
    string GetTitle();
}
```

C# 服务端程序

```
/// 用于模拟的具有动态信息更新能力的注册表对象
static class MessageRegistry
{
    /// 业务信息内容
    public static string ID = string.Empty;
    public static string Title = string.Empty;

    /// 触发更新的 Timer
    private static Timer timer;

    static void UpdateID(Object sender, ElapsedEventArgs args)
    {
        if(string.IsNullOrEmpty(Title))
            Title = "MarvellousWorks";
        MessageRegistry.ID = Guid.NewGuid().ToString();
    }

    static MessageRegistry()
    {
        timer = new Timer(1000);
        timer.Elapsed += UpdateID;
        timer.Enabled = true;
    }
}

/// WCF 服务端实体对象
class MessageService : IMessage
{
    public string GetID()
```

```
    {
        return MessageRegistry.ID;
    }

    public string GetTitle()
    {
        return MessageRegistry.Title;
    }
}

class Program
{
    public static void Main()
    {
        using (ServiceHost host = new ServiceHost(typeof(MessageService)))
        {
            host.Open();
            Console.WriteLine("Service started ...");
            Console.ReadLine();
        }
    }
}
```

C# 客户端 EventMonitor 及 Snapshot 部分

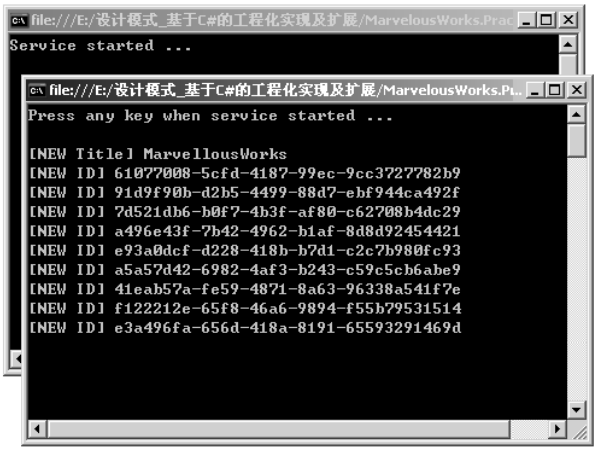
```
/// Snapshot 对象
static class Snapshot
{
    public const string IDItem = "ID";
    public const string TitleItem = "Title";

    private static IDictionary<string, string> dictionary =
        new Dictionary<string, string>();

    static Snapshot()
    {
        dictionary.Add(IDItem, string.Empty);
        dictionary.Add(TitleItem, string.Empty);
    }

    /// 鉴别新监控到的信息是否与快照信息相符
    public static bool Equals(string key, string monitorValue)
    {
        if (string.IsNullOrEmpty(key)) throw new
           ArgumentNullException("key");
        if (!dictionary.ContainsKey(key)) throw new ArgumentException("cannot
           find item");
        bool isEqual = string.Equals(dictionary[key], monitorValue);
        // 根据最新的数据快照更新自身信息
        // 实际处理中这个操作也可以由 EventMonitor 发起
        if (!isEqual)
            dictionary[key] = monitorValue;
        return isEqual;
    }
}

#region 事件监控器
class MonitorEventArgs : EventArgs
{
```

```
    private string newValue;
    public string NewValue{get{return newValue;}}

    public MonitorEventArgs(string newValue)
    {
        this.newValue = newValue;
    }
}

class EventMonitor
{
    private Timer timer;
    private MessageClient client;

    #region EventMonitor 的监控频率
    private int interval;
    public int Interval
    {
        get { return interval; }
        set { interval = value; }
    }
    #endregion

    #region 对外的预订事件
    public event EventHandler<MonitorEventArgs> IDChanged;
    public event EventHandler<MonitorEventArgs> TitleChanged;
    #endregion

    public EventMonitor(MessageClient client)
    {
        this.client = client;
        timer = new Timer(1000);
        timer.Elapsed += Refersh;
    }

    /// <summary>
    /// 轮循执行时执行的逻辑
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="args"></param>
    void Refersh(Object sender, ElapsedEventArgs args)
    {
        string id = client.GetID();
        string title = client.GetTitle();

        if (TitleChanged != null)
        {
            if (!Snapshot.Equals(Snapshot.TitleItem, title))
                TitleChanged(this, new MonitorEventArgs(title));
        }

        if (IDChanged != null)
        {
            if (!Snapshot.Equals(Snapshot.IDItem, id))
                IDChanged(this, new MonitorEventArgs(id));
        }
    }

    public void Start()
```

```
    {
        timer.Elapsed += Refersh;
        timer.Enabled = true;
    }

    public void Stop()
    {
        timer.Enabled = false;
    }
}
#endregion
```

Unit Test

```
class Program
{
    static void OnIDChanged(object sender, MonitorEventArgs args)
    {
        Console.WriteLine("[NEW ID] " + args.NewValue);
    }
    static void OnTitleChanged(Object sender, MonitorEventArgs args)
    {
        Console.WriteLine("[NEW Title] " + args.NewValue);
    }

    static void Main(string[] args)
    {
        using (MessageClient client = new MessageClient())
        {
            Console.WriteLine("Press any key when service started ...");
            Console.ReadLine();
            EventMonitor moitor = new EventMonitor(client);

            // 预订更新通知
            moitor.TitleChanged += OnTitleChanged;
            moitor.IDChanged += OnIDChanged;
            moitor.Start();
            Console.ReadLine();
        }
    }
}
```



由示例结果看，Title 属性由于只更新了一次，因此事件监控器只通知下游客户程序它被更新一次。而 ID 属性因为持续更新，所以下游客户程序会持续获得更新通知。

37.4 小结

事件监控器模式是一种有效包装其他 Web Service 的手段，境界上有点类似“辛苦我一个，幸福千万人”。因为它既承担了对服务端（甚至是一批服务端）的轮循调用，同时又负责通知一批下游客户程序。

事件监控器模式必须可控，尤其对于企业内部的应用而言，因为过于频繁就很容易“压垮”服务端，从“有效的二级通知机制”演变成“准分布式拒绝服务攻击了”。

附录 A

面向关系数据和 XML 数据的领域逻辑模式

A.1 说明

A.2 实现业务领域逻辑的主要方法

A.3 示例

A.4 小结

A.1 说明

在领域驱动设计（DDD：Domain Driven Design）中，实现业务逻辑层主要有 Transaction Script、Domain Module 和 Table Module 三种模式。随着业务逻辑复杂程度的增加，采用各模式实现的工作量变化趋势有所不同；根据应用特点不同，三种模式也各有优势。

- Transaction Script：业务逻辑直接用 SQL 脚本与数据库交互，实现简单，但是限于 SQL 面向过程化的特点，完成复杂业务逻辑时工作量较大。
- Domain Module：将业务数据封装为业务对象，适于业务逻辑复杂的应用，但需要 O/R 映射的支持。
- Table Module：将业务数据组织成数据表方式，虽然对象化特征不如 Domain Module 明显，但适于展现层使用。

实现所需工作量与业务逻辑复杂度的关系如图 A-1 所示。

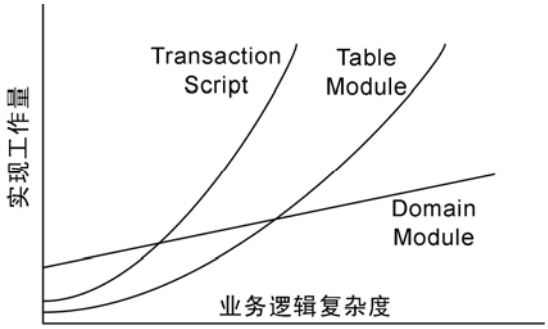


图 A-1 实现所需工作量与业务逻辑复杂度的关系

应用建设初期选择的实现模式随着业务需求和历史数据量的变化可能需要进行调整，此时要增加一个适应性机制，保证在尽量不影响客户程序的前提下，选择合适的实现模式。随着 XML 数据使用日趋广泛，须借助 XPath、XQuery 和 XSL 为层次型数据增加专门的扩展机制，使得基于 XML 数据源的业务逻辑也可以采用上述三种模式实现。

A.2 实现业务领域逻辑的主要方法

A.2.1 整体逻辑结构

总体适配机制如图 A-2 所示。

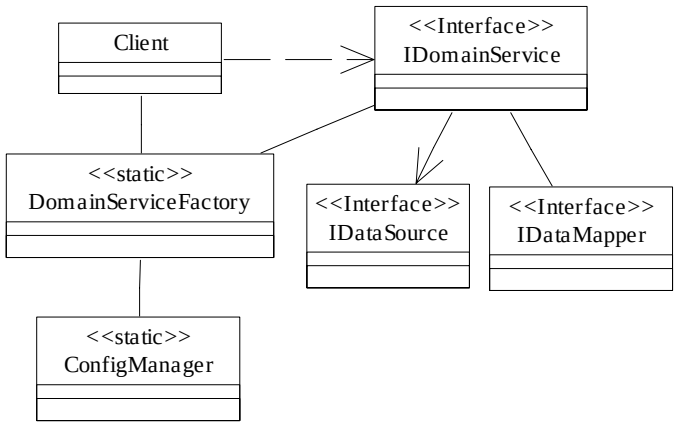


图 A-2 总体实现结构

为业务服务增加抽象接口 IDomainService，客户程序通过 DomainServiceFactory 获得该抽象接口。这样客户程序不依赖具体的业务服务实体类，仅依赖抽象的服务接口，当下层实现模式调整时，不影响客户程序；为了让框架可以同时适应关系数据库和 XML 数据，增加了抽象接口 IDataSource，代表不同的数据源对象；IDataMapper 负责根据不同的数据源，完成关系数据或 XML 数据与业务对象的映射；为了减少 DomainServiceFactory 与具体业务服务对象产生依赖，增加配置管理对象 ConfigManager，由它获取指定的业务服务的实体类名称，并通过反射机制动态生成目标实例。

A.2.2 性能改进

由于业务实体经常会对应到具有 Master-Detail 关系的多个表，而且有些复杂业务实体本身会包含一组或几组其他业务实体，出于性能考虑，为了避免 IDataMapper 在映射过程中频繁调用数据源逐个生成子业务实体，需要在 IDataMapper 与数据源之间增加一个 DTO（Data Transfer Object）对象 IDataTransferObject，通过将调用打包的办法，减少频繁的远程调用，如图 A-3 所示。

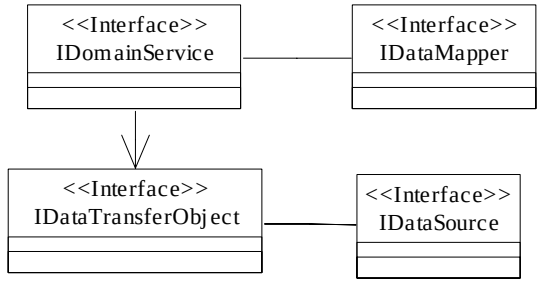


图 A-3 借助 DTO，Domain Module 对象间接访问数据源

A.2.3 面向关系数据库的业务服务设计

为了将业务实体纳入适配机制的管理，依据依赖倒置原则，先对各模式实现的业务实体进行抽象，如图 A-4 所示。

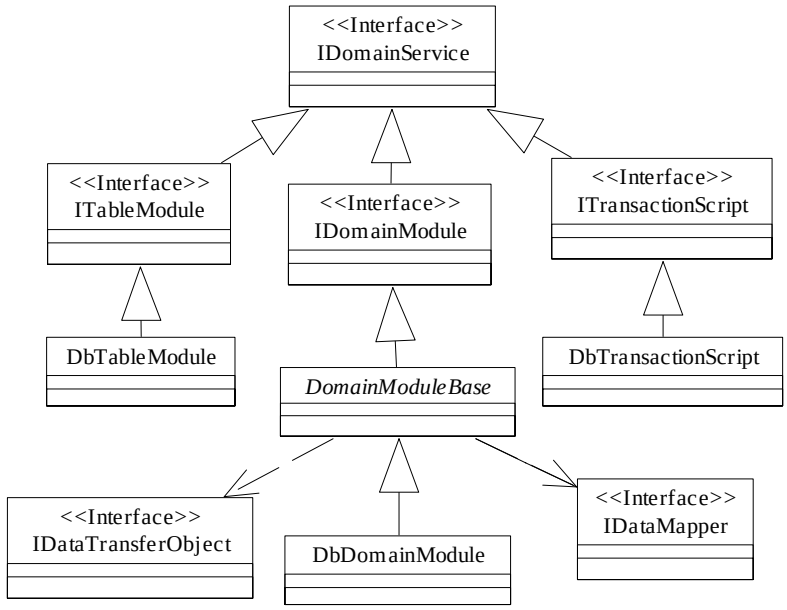


图 A-4 关系数据库方式下的适配机制

如图 A-4 所示，设计上为每种模式实现的业务对象抽象独立接口，并编写对应的关系数据库实现类；Domain Module 需要调度数据映射和 DTO 进行关系数据与业务实体的映射；增加抽象基类 DomainModuleBase，通过调用 IDataMapper 和 IDataTransferObject 完成数据提取和映射工作。可用的实现技术如表 A-1 所示。

表 A-1 三个模式在关系信息下的典型实现方式

模式	执行方式	返回结果
TranScript	SQL	查询结果或统计值
DomainModule	SQL 和关系数据映射	业务对象
TableModule	SQL	DataTable 或 DataSet

A.2.4 面向 XML 数据的扩展设计

XML 数据方式下的适配机制如图 A-5 所示。

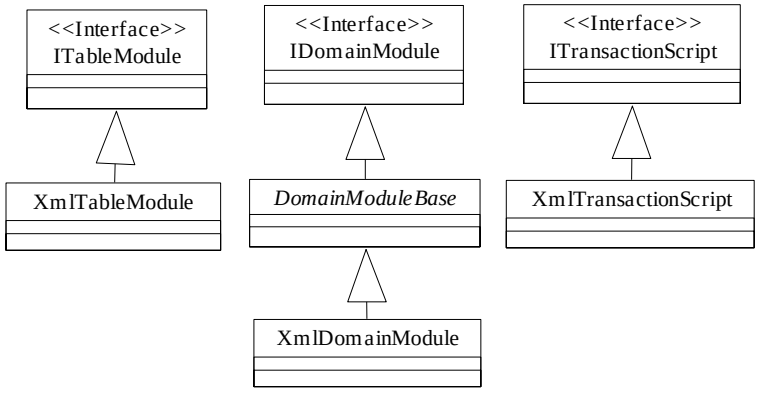


图 A-5 XML 数据方式下的适配机制

由于 XML 的层次特征，三个模式的实现技术与关系数据库不同，如表 A-2 所示。

表 A-2 三个模式在 XML 信息下的典型实现方式

模式	执行方式	返回结果
TranScript	SQL	查询结果或统计值
DomainModule	SQL 和关系数据映射	业务对象
TableModule	SQL	DataTable 或 DataSet

A.2.5 配置机制设计

通过增加服务接口工厂类的方式隔离客户程序与具体业务服务实体类间的依赖，工厂类通过配置管理 ConfigManager 获得每个目标服务接口对应的实体类名称，借助反射动态包装目标服务接口（见图 A-6）。静态结构和客户程序获得业务服务接口的时序关系如图 A-7 所示。

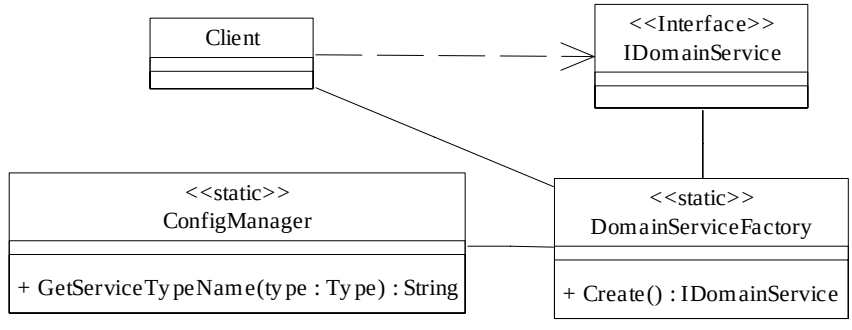


图 A-6 配置管理机制

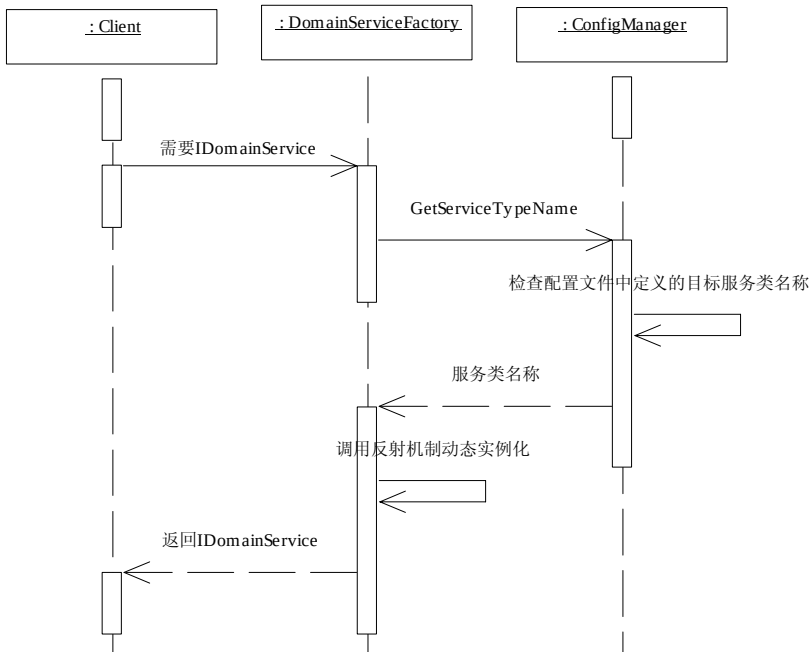


图 A-7 客户程序获得业务服务接口的时序关系

A.3 示例

A.3.1 示例情景

为了比较三种模式实现特点的不同，测试中设计了两个具有 Master-Detail 特征的业务实体：Customer 和 Order，两者之间也存在 1:N 的关系，对应的关系数据库和 XML 数据实现如图 A-8、图 A-9、图 A-10 所示。

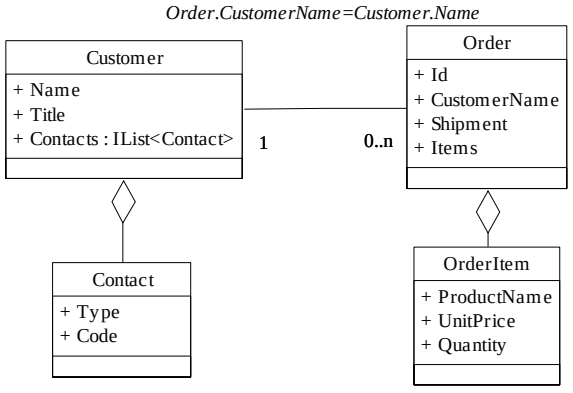


图 A-8 业务实体

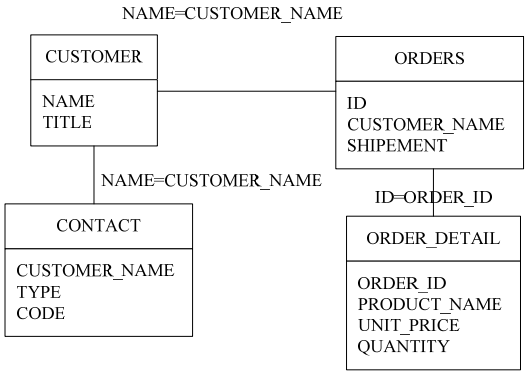


图 A-9 关系数据库方式下业务实体实现

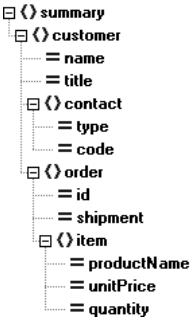


图 A-10 XML 数据方式下业务实体的实现

目标服务是一个根据客户名称，返回其所有订单明细项小计之和的接口。

A.3.2 测试内容准备

针对关系数据库和 XML 数据方式的不同， XML 数据的 Transaction Script 模式为了计算简单，采用 XSLT 生成“客户名称—订单项明细小计”的中间过程，如图 A-11 所示。

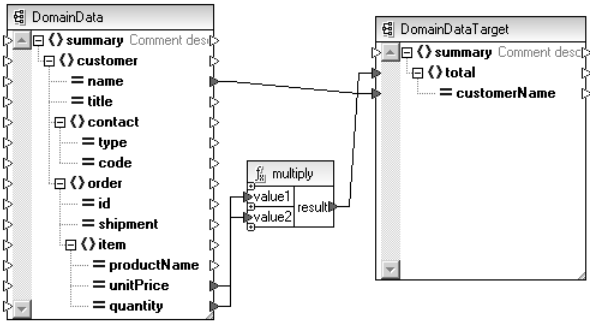


图 A-11 XML Transaction Script 方式下生成中间结果的 XSLT

两种数据模型下三个实现模式的计算方法如表 A-3 所示。

表 A-3

RDBMS	采用 SQL 方式
TranScript	SELECT SUM(UNIT_PRICE* QUANTITY) FROM ORDER_DETAIL WHERE ORDER_ID IN (SELECT ID FROM ORDERS WHERE CUSTOMER_NAME = '?')
DomainModule	$\sum_{i,j}(Customer.Order[i].Item[j].UnitPrice * Quantity)$
TableModule	$\sum_{i}(DataTable.Rows[i].UnitPrice * Quantity)$
XML	采用 XPath 方式
TranScript	针对中间数据执行 XPath 表达式 sum(/summary/total[@customerName='<name>']获得统计结果
DomainModule	$\sum_{i,j}(Customer.Order[i].Item[j].UnitPrice * Quantity)$
TableModule	先通过/summary/customer [@name='<name>'"]/order/item 获得该用户所有订单明细项的 XMLNodeList，然后汇总每项小计

A.3.3 实际测试过程

测试数据如表 A-4 所示。

表 A-4

客户 joe			
订单编号	产品	单价	数量
10000	desk	200	30
10000	chair	50	200
10000	cup	2.1	20000
10000	room	500000	1

通过修改 ConfigManager 中实现业务服务的实体类名称，结合数据库调用监控获得如表 A-5 中所示数据。

表 A-5

RDBMS	Name	Total	SQL 数量
Tran Script	joe	558000	1
Domain Module	joe	558000	1
Table Module	joe	558000	1
XML			
Tran Script	joe	558000	
Domain Module	joe	558000	
Table Module	joe	558000	

测试结果分析如下：

- 借助适配机制，当目标实现模式甚至数据模型修改时，客户程序保持稳定，修改内容控制在配置文件部分，不影响客户程序的业务逻辑。
- 使用不同模式设计完成的业务对象，借助适配机制和 XML 数据扩展机制，在关系数据库和 XML 数据方式下，可完成同样的服务功能。
- 通过 DTO 组件的调用打包作用，可以将包括两份订单、四项订单明细的信息一次性提取，将多次调用打包为一次调用，减少了网络往复。

A.4 小结

依据抽象设计的适配机制可以从一定程度上减少客户程序与业务逻辑的耦合程度，当部署、运行环境变化时，可通过调整配置选择合适的业务逻辑实现模式，并且不需要客户程序联动修改；DTO 对象的加入可以减少分布式调用中的往复次数，对应用性能的提升有利。实际工程中，由于业务逻辑实体往往需要被多个客户程序调用，实际项目中需要有效的并发机制配合。